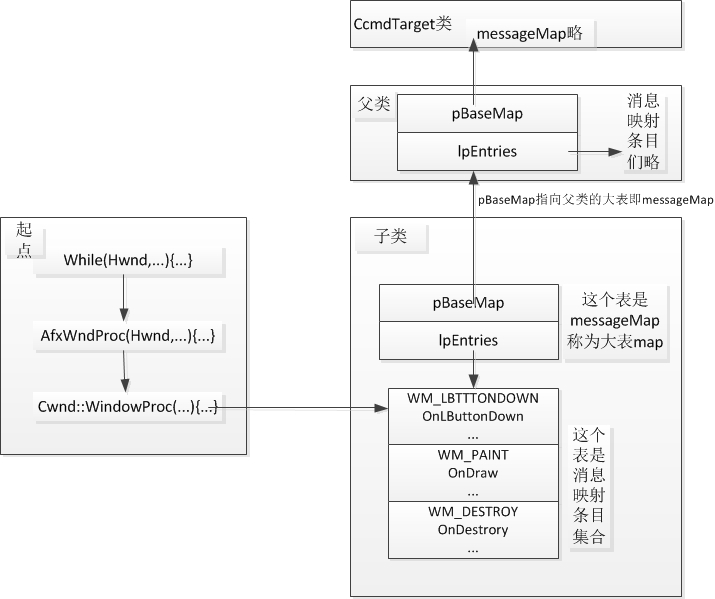
**图解MFC基本框架（深入消息映射机制）**

**首先，先看整体的消息流向图：**



上图解释：

 起点是消息循环，在winmain函数中（mfc中winmain函数是隐含的调用的，在app全局对象构造完后紧接着调用winmain函数），while循环中不断从应用程序队列中取消息，当取得一个消息时（含HWnd句柄），调用全局的AfxWndProc窗口函数（含有HWnd句柄作为参数），这个全局的函数根据HWnd得到具体的窗体wnd对象，然后调用对象的Cwnd::WindowProc窗口函数（此时不含HWnd句柄作为参数了）。至此就从全局进入到具体窗体对象的窗口函数了。然后Cwnd::WindowProc遍历具体窗口类（对象）的消息映射条目集合（又称为窗口的消息映射表），一旦对应的条目，就通过对应条目的函数指针成员调用具体的消息处理函数了，但是如果遍历完后没找到，则通过messageMap（我称为大表map，更准确的说是大节点，因为一个messageMap就是一个链表的节点）的pBaseMap成员找到其父类的messageMap，通过这个大节点map的lpEntries成员找到父类的消息映射条目集合表，然后遍历这个表，如果找到某个条目则执行，然后结束，否则继续向上一层父类重复上述过程，这就是mfc独有的消息映射机制，它模仿了虚函数的实现（即从子类向父类的执行流向）。【但之所以不用虚函数而用这么复杂的机制，应该是如果用虚函数，则每个类（准确的说是具体的子类）都背负一个庞大的虚函数表，而这个机制的消息映射条目集合只是按需添加需要处理的消息，效率更高了（对子类来说效率高了，如果是向上传递到最顶层的父类的消息查找，则会比虚函数表效率更低）。】

另外，补充一下，对于while消息循环

DispatchMessage(&msg);  //把消息路由给（发送给）操作系统，最后由操作系统负责调用具体的消息处理函数（窗口函数）

通过上面语句才把消息队列的消息最终发送到某个窗口函数函数的（比如WndProc），而这些窗口函数不是说定义好后，操作系统就能识别的，必须通过窗口类注册过后才能被操作系统识别，进而把msg发送到某个注册过的窗口类对应的窗口函数【应该是通过HWnd来找到注册过的窗口类的，还记得win32应用程序框架的CreateWindow(LPCTSTR lpClassName,...)函数吗，说明每个建立好的窗口都应该保存有窗口类的名字lpClassName】。至于窗口类怎么和窗口函数对应，就得去复习wn32应用程序框架了，下面简单复习下：

WNDCLASS wc;  //定义一个窗口类的实例，即一个空清单

wc.lpfnWndProc = WndProc; //窗口函数的首地址记录下来（窗口类也和一个窗口函数就一一对应了）

......

RegisterClass(&wc); //窗口类实例的注册

通过上面的语句即注册窗口类了，同时这个窗口类也和一个窗口函数一一对应了。

解决了上面所说的操作系统只是把注册过的窗口类对应的窗口函数的问题后，就有疑问了，是不是每个窗体包括控件都注册成了窗体类了，好像没看到这样的代码啊？况且已经知道每个窗口包括控件都有自己的窗口函数即Cwnd::WindowProc函数，那么如果有上百个控件，岂不是系统中保存了上百个窗口类（对应了窗口函数），效率能高吗?其实是mfc更改了每个窗口对应的窗口类，让每个窗口wnd的窗口类对应的窗口函数都是唯一的全局的AfxWndProc函数，所以操作系统通过DispatchMessage转发消息时只需要识别这一个窗口函数就行了，以后转到哪个具体的窗口函数就是上面图中所示的东西了。

**其次，下面补充一个自己写的一个简单的仿MFC框架的完整的例子（vs2010下）：**

（用消息映射机制实现的一个简单窗体CMyWnd，它继承于CFrameWnd）

就只有一个cpp文件（为了简化过程，把类声明和实现写到一个文件中了），内容如下：

首先，新建一个空win32应用程序，然后添加一个cpp文件，在文件中书写如下代码：

#include <afxwin.h>

class CMyWnd:public CFrameWnd

{

private:

　　char\* ShowText;  //声明一个字符串变量为数据成员

public:

　　afx\_msg void OnPaint();  //声明WM\_PAINT消息处理函数，注意afx\_msg只是一个占位符供类向导定位用

　　afx\_msg void OnLButtonDown();  //声明鼠标左键按下消息处理函数

　　DECLARE\_MESSAGE\_MAP();  //声明消息映射。这个宏本质是给类添加了：声明静态的\_messageEntries[]数组和messageMap指针，可能还有其他的东西

　　afx\_msg void OnLButtonDown(UINT nFlags, CPoint point);

private:  
　　CButton btn;  //一个按钮作为成员变量

};

//消息映射的实现

BEGIN\_MESSAGE\_MAP(CMyWnd,CFrameWnd)  //初始化初始化声明中定义的消息映射条目集合\_messageEntries[]和messageMap

　　ON\_WM\_PAINT()  ON\_WM\_LBUTTONDOWN()

END\_MESSAGE\_MAP()

/\*对BEGIN\_MESSAGE\_MAP和END\_MESSAGE\_MAP更详细的说明，大概内容如下：

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

theClass::messageMap =          //【填写子类的大表或大节点messageMap，这个节点里面就放了两个指针，分别指向父类的大节点以及本类的消息映射条目集合的首地址\_messageEntries[0]】

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

theClass::\_messageEntries[] =   //【填写子类的消息映射条目集合。每一个条目主要包含两个东西：消息标志如WM\_PAINT和消息处理函数的指针。

　　{   //【这里是根据“ON\_WM\_PAINT()”之类的宏展开，一条一条的往数组中填充消息映射条目】}

};  //【END\_MESSAGE\_MAP()宏，先简单的理解为就是一个右大括号和一个分号，当然其实际上代表更多，要不然也不会带有小括号了。

\*/

//WM\_PAINT消息处理函数的实现

void CMyWnd::OnPaint()

{

　　CPaintDC dc(this);

　　dc.TextOut(20,20,ShowText);

}

//WM\_LBUTTONDOWN消息处理函数的实现

void CMyWnd::OnLButtonDown(UINT nFlags, CPoint point)

{

　　ShowText="有消息映射表的程序";  //当鼠标按下时显示的字符串

　　InvalidateRect(NULL,TRUE);  //通知更新或重绘窗口

　　CFrameWnd::OnLButtonDown(nFlags, point);

}

//OnCreate类的函数都是在窗体创建完了之后调用的，所以可以在这个函数里创建一些按钮。

BOOL CMyWnd::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext\* pContext)  
{

　　 //CButton btn;   注意，不能把btn定义成局部的，否则函数执行完后就把btn类销毁了，而btn类会自动销毁相关的资源，所以btn就显示不出来了。

　　//对控件类的窗体来说，光构造对象还不行，还必须要初始化才能显示，初始化用的都是create或createEx之类的函数。这里btn已经是类的成员变量了，所以构造就省略了。  
 　　btn.Create(L"my按钮",WS\_CHILD|BS\_DEFPUSHBUTTON,CRect(50,120,50+100,120+100),this,123);   //最后一个参数123是控件的ID号，随便取一个值即可。 按钮是正方形的。  
 　　btn.ShowWindow(SW\_SHOWNORMAL);  
　　 return CFrameWnd::OnCreateClient(lpcs, pContext);  
}

//程序员由CWinApp派生的应用程序类

class CMyApp:public CWinApp

{

public:

　　BOOL InitInstance();

};

BOOL CMyApp::InitInstance()

{

　　CMyWnd\* pMainWnd=new CMyWnd;

　　pMainWnd->Create(0,L"窗口标题");  //这个函数中完成了注册窗体类和创建一个具体的窗体，当然中间可能经过了多个mfc的函数，机制很复杂的

　　pMainWnd->ShowWindow(m\_nCmdShow);

　　pMainWnd->UpdateWindow();

　　m\_pMainWnd=pMainWnd;

　　return TRUE;

}

//定义APP全局对象

CMyApp MyApp;

以上就是整个cpp文件中的内容。在编译之前，修改vs2010项目的配置属性->常规->MFC的使用选项选择使用共享或静态的mfc库，最后编译运行即可，运行效果如下图：



 另外，上面的CMyWnd::OnCreateClient函数是虚函数而且是创建窗体过程中系统自己发ON\_CREATE消息然后调用的，这是因为窗体类是继承与CFrameWnd类的，而CFrameWnd类都有默认的这个虚函数，但是如果是CView类，则默认没有OnCreate函数，也就是**对于CView来说，默认情况下是不把(ON\_CREATE，对应的函数地址)这个的消息映射条目加入到\_messageEntries[]数组中的（而CFrameWnd类的OnCreate函数对应的消息映射条目默认是加入到那个数组的），**所以需要手动添加消息处理函数，而不像CMyWnd::OnCreateClient那样只需要改写对应的虚函数OnCreateClient就行了。

【**补充孙鑫教程中了解到的MFC标准的单文档应用程序框架流程**：**首先是全局MyApp对象的构造函数调用**（即全局对象的构造），**然后操作系统启动winmain函数**，在这个winmain函数中调用afxMain函数，进入afxMain函数中：先是调用了**MyApp对象的InitInstance函数**，最后调用**MyApp对象的Run函数启动消息循环**。

对于InitInstance函数内部调用了一些主要的函数：首先是通过shell命令即“ParseCommandLine(cmdInfo)这句”调用了AfxEndDeferRegisterClass()即afx框架的注册窗口类函数，然后shell命令又调用了CWnd::PreCreateWindiw()函数，而这个PreCreateWindiw函数中又一次调用AfxEndDeferRegisterClass函数（之所以调用了两次可能是为了处理涉及文档框架等细节的东西），在这里可能修改窗口类保存的窗口处理函数的地址。然后shell又调用了CFrame::LoadFrame函数，这个函数（具体作用不重要先不管）调用CFrame::Create函数创建一个具体的窗体（实际山create中调用了createEx来完成的），特别注意的是CFrame::Create中还调用了子类CMainFrame::PreCreateWindow函数目的是为了修改窗体的样式，注意虽然总共调用了两个PreCreateWindow函数实际上调用的是不同类的函数，第一次是为了实现afx框架的窗口类的建立，而第二次是为了让用户在创建窗口之前有机会修改窗体的外观。

至此窗口就创建完成了，**窗口创建的所有工作都是由shell命令完成**的，最后回到InitInstance函数调用窗口的**ShowWindow和UpdateWindow来显示窗口**。】

**第三、下面深入讲解细节原理：**

一、消息映射不用深入掌握情况下的用法

 先复习下msg结构体的定义，共六个成员：

typedef struct tagMSG {  
HWND hwnd;  
UINT message;  //消息的类型（即事件的类型）  
WPARAM wParam;  //附加或称数据参数1  
LPARAM lParam;  
DWORD time;  
POINT pt;  
} MSG;

再复习下窗口过程的定义（四个参数，分别是msg的前四个成员）：

LRESULT CALLBACK MainWndProc (    
HWND hwnd,// 窗口句柄  
UINT msg,// 消息标识  
WPARAM wParam,//消息参数1  
LPARAM lParam//消息参数2  
)

消息的分类（从发送途径上看）和传递过程：

消息分两种：队列消息和非队列消息。队列消息送到系统消息队列，然后到线程消息队列；非队列消息直接送给目的窗口过程。  
这里，对消息队列阐述如下：  
Windows维护一个系统消息队列（System message queue），每个GUI线程有一个线程消息队列（Thread message queue）。  
鼠标、键盘事件由鼠标或键盘驱动程序转换成输入消息并把消息放进系统消息队列，例如WM\_MOUSEMOVE、WM\_LBUTTONUP、WM\_KEYDOWN、WM\_CHAR等等。Windows每次从系统消息队列移走一个消息，确定它是送给哪个窗口的和这个窗口是由哪个线程创建的，然后，把它放进窗口创建线程的线程消息队列。线程消息队列接收送给该线程所创建窗口的消息。线程从消息队列取出消息，通过Windows把它送给适当的窗口过程来处理。  
除了键盘、鼠标消息以外，队列消息还有WM\_PAINT、WM\_TIMER和WM\_QUIT。  
这些队列消息以外的绝大多数消息是非队列消息。

从消息的来源来看，可以分为：系统定义的消息和应用程序定义的消息。  
系统消息ID的范围是从0到WM\_USER-1，或0X80000到0XBFFFF；应用程序消息从WM\_USER（0X0400）到0X7FFF，或0XC000到0XFFFF；WM\_USER到0X7FFF范围的消息由应用程序自己使用；0XC000到0XFFFF范围的消息用来和其他应用程序通信，为了ID的唯一性，使用::RegisterWindowMessage来得到该范围的消息ID。

MFC处理的三类消息：

Windows消息，前缀以“WM\_”打头，WM\_COMMAND例外。Windows消息直接送给MFC窗口过程处理，窗口过程调用对应的消息处理函数。一般，由窗口对象来处理这类消息，也就是说，这类消息处理函数一般是MFC窗口类的成员函数。

控制通知消息，是控制子窗口送给父窗口的WM\_COMMAND通知消息。窗口过程调用对应的消息处理函数。一般，由窗口对象来处理这类消息，也就是说，这类消息处理函数一般是MFC窗口类的成员函数。

命令消息，这是来自菜单、工具条按钮、加速键等用户接口对象的WM\_COMMAND通知消息，属于应用程序自己定义的消息。通过消息映射机制，MFC框架把命令按一定的路径分发给多种类型的对象（具备消息处理能力）处理，如文档、窗口、应用程序、文档模板等对象。能处理消息映射的类必须从CCmdTarget类派生。

一般使用方法，即MFC消息映射的实现方法（即使用过程中的做法，不需要掌握内部运行原理）：

应用程序类的定义（头文件）包含了类似如下的代码：  
//{{AFX\_MSG(CTttApp)          //这句比较特殊的注释是给类向导定位用的，可以删除，不影响编译和运行，但影响类向导的显示  
afx\_msg void OnAppAbout();  //afx\_msg只是个占位符，编译器会忽略，他是给类向导定位用的，可能还有其他作用，所以必不可少  
//}}AFX\_MSG                        //这句比较特殊的注释是给类向导定位用的  
DECLARE\_MESSAGE\_MAP()

应用程序类的实现文件中包含了类似如下的代码：  
BEGIN\_MESSAGE\_MAP(CTApp, CWinApp)  
//{{AFX\_MSG\_MAP(CTttApp)  
ON\_COMMAND(ID\_APP\_ABOUT, OnAppAbout)  //宏就不需要定位用的afx\_msg之类的东西了，指不定这些定位的东西都写在宏里了呢。  
//}}AFX\_MSG\_MAP  
END\_MESSAGE\_MAP()  
头文件里是消息映射和消息处理函数的声明，实现文件里是消息映射的实现和消息处理函数的实现。它表示让应用程序对象处理命令消息ID\_APP\_ABOUT，消息处理函数是OnAppAbout。

void CMyApp::OnAppAbout()

{

......  //函数的具体实现代码略

}

 本文最后还有一个完整的例子。

二、消息映射机制的内部原理：

（1）消息映射声明的解释

#define DECLARE\_MESSAGE\_MAP() \      //一个右斜杠是连接两行的意思，为了便于把代码书写成多行  
private: \  
static const AFX\_MSGMAP\_ENTRY \_messageEntries[]; \  
protected: \  
static AFX\_DATA const AFX\_MSGMAP messageMap; \  
virtual const AFX\_MSGMAP\* GetMessageMap() const;

上面这句消息映射声明的实质是给所在类添加几个静态成员变量和静态或虚拟函数。

成员变量  
有两个成员变量被添加，第一个是\_messageEntries，第二个是messageMap。

\_messageEntries[]是一个AFX\_MSGMAP\_ENTRY 类型的数组变量，是一个静态成员变量，用来容纳类的消息映射条目。一个消息映射条目可以用AFX\_MSGMAP\_ENTRY结构来描述。  
AFX\_MSGMAP\_ENTRY结构的定义如下：  
struct AFX\_MSGMAP\_ENTRY  
{  
UINT nMessage;  //Windows消息ID  
UINT nCode;       //控制消息的通知码  
UINT nID;          //Windows Control的ID  
UINT nLastID;    //如果是一定范围的消息被映射，则nLastID指定其范围  
UINT nSig;         //消息的动作标识  
AFX\_PMSG pfn;  //响应消息时应执行的函数(routine to call (or special value))  
};  
从上述结构可以看出，每条映射有两部分的内容：第一部分是关于消息ID的，包括前四个域；第二部分是关于消息对应的执行函数，包括后两个域。  
在上述结构的六个域中，pfn是一个指向CCmdTarger成员函数的指针。函数指针的类型定义如下：  
typedef void (AFX\_MSG\_CALL CCmdTarget::\*AFX\_PMSG)(void);当使用一条或者多条消息映射条目初始化消息映射数组时，各种不同类型的消息函数都被转换成这样的类型：不接收参数，也不返回参数的类型。因为所有可以有消息映射的类都是从CCmdTarge派生的，所以可以实现这样的转换。nSig是一个标识变量，用来标识不同原型的消息处理函数，每一个不同原型的消息处理函数对应一个不同的nSig。在消息分发时，MFC内部根据nSig把消息派发给对应的成员函数处理，实际上，就是根据nSig的值把pfn还原成相应类型的消息处理函数并执行它。

第二个成员变量的声明  
AFX\_MSGMAP messageMap;是一个AFX\_MSGMAP类型的静态成员变量，从其类型名称和变量名称可以猜出，它是一个包含了消息映射信息的变量。。的确，它把消息映射的信息（消息映射数组）和相关函数打包在一起，也就是说，得到了一个消息处理类的该变量，就得到了它全部的消息映射数据和功能【我称之为消息映射大表】。AFX\_MSGMAP结构的定义如下：

struct AFX\_MSGMAP  
{

//pBaseMap保存基类消息映射入口\_messageEntries的地址【其实是保存基类大表，大表中含有消息映射条目数组的首地址】  
const AFX\_MSGMAP\* pBaseMap;  
//lpEntries保存消息映射入口\_messageEntries的地址  
const AFX\_MSGMAP\_ENTRY\* lpEntries;  
};

从上面的定义可以看出，通过messageMap可以得到类的消息映射数组\_messageEntries和基类消息映射数组的地址。

（2）消息映射实现的解释

下面是BEGIN\_MESSAGE\_MAP宏：

#define BEGIN\_MESSAGE\_MAP(theClass, baseClass) \  
const AFX\_MSGMAP\* theClass::GetMessageMap() const \  
{ return &theClass::messageMap; } \  
AFX\_DATADEF const AFX\_MSGMAP theClass::messageMap = \  
{ &baseClass::messageMap, &theClass::\_messageEntries[0] }; \  
const AFX\_MSGMAP\_ENTRY theClass::\_messageEntries[] = \  
{ \

下面是END\_MESSAGE\_MAP宏：

#define END\_MESSAGE\_MAP() \  
{0, 0, 0, 0, AfxSig\_end, (AFX\_PMSG)0 } \  
};

消息映射实现的实质是初始化声明中定义的静态成员函数\_messageEntries和messageMap。

这样，在进入WinMain函数之前，每个可以响应消息的MFC类都生成了一个消息映射表，程序运行时通过查询该表判断是否需要响应某条消息。

<1>消息映射数组\_messageEntries[]的初始化

消息映射数组的元素是消息映射条目，条目的格式符合结构AFX\_MESSAGE\_ENTRY的描述。所以，要初始化消息映射数组，就必须使用符合该格式的数据来填充.显然，这是一个繁琐的工作。为了简化操作，MFC根据消息的不同和消息处理方式的不同，把消息映射划分成若干类别，每一类的消息映射至少有一个共性：消息处理函数的原型相同。对每一类消息映射，MFC定义了一个宏来简化初始化消息数组的工作。例如，前文提到的ON\_COMMAND宏用来映射命令消息，只要指定命令ID和消息处理函数即可，因为对这类命令消息映射条目，其他四个属性都是固定的。

<2>messageMap的初始化

messageMap的类型是AFX\_MESSMAP。经过初始化，域lpEntries保存了消息映射数组\_messageEntries的地址;pBaseMap保存了基类的消息映射数组的地址。

（3）消息映射宏的解释

下面是ON\_COMMAND宏：

#define ON\_COMMAND(id, memberFxn) \  
{\  
WM\_COMMAND,\  
CN\_COMMAND,\  
(WORD)id,\          //最终是控件的ID值  
(WORD)id,\  
AfxSig\_vv,\        //是AfxSig\_vv是MFC预定义的枚举变量  
(AFX\_PMSG)memberFxn\  
}；

为了简化程序员的工作，MFC定义了一系列的消息映射宏和像AfxSig\_vv这样的枚举变量，以及标准消息处理函数，并且具体地实现这些函数。这里主要讨论消息映射宏，常用的分为  
以下几类。  
<1>用于Windows消息的宏，前缀为“ON\_WM\_”。  
这样的宏不带参数，因为它对应的消息和消息处理函数的函数名称、函数原型是确定的。MFC提供了这类消息处理函数的定义和缺省实现。每个这样的宏处理不同的Windows消息。  
例如：宏ON\_WM\_CREATE()把消息WM\_CREATE映射到OnCreate函数，消息映射条目的第一个成员nMessage指定为要处理的Windows消息的ID，第二个成员nCode指定为0。

<2>用于命令消息的宏ON\_COMMAND  
这类宏带有参数，需要通过参数指定命令ID和消息处理函数。这些消息都映射到WM\_COMMAND上，也就是将消息映射条目的第一个成员nMessage指定为WM\_COMMAND，第二个成员nCode指定为CN\_COMMAND(即0)。消息处理函数的原型是void (void)，不带参数，不返回值。另外对于ON\_COMMAND\_RANGE，这里暂不讨论。

<3>用于控制通知消息的宏  
这类宏可能带有三个参数，如ON\_CONTROL，就需要指定控制窗口ID，通知码和消息处理函数；也可能带有两个参数，如具体处理特定通知消息的宏ON\_BN\_CLICKED、ON\_LBN\_DBLCLK、ON\_CBN\_EDITCHANGE等，需要指定控制窗口ID和消息处理函数。  
控制通知消息也被映射到WM\_COMMAND上，也就是将消息映射条目的第一个成员的nMessage指定为WM\_COMMAND，但是第二个成员nCode是特定的通知码，第三个成员nID是控制子窗口的ID，第四个成员nLastID等于第三个成员的值。消息处理函数的原型是void (void)，没有参数，不返回值。另外ON\_NOTIFY类似于ON\_CONTROL，这里暂不讨论。

<4>用于用户界面接口状态更新的ON\_UPDATE\_COMMAND\_UI宏  
这类宏被映射到消息WM\_COMMND上，带有两个参数，需要指定用户接口对象ID和消息处理函数。消息映射条目的第一个成员nMessage被指定为WM\_COMMAND，第二个成员nCode被指定为-1，第三个成员nID和第四个成员nLastID都指定为用户接口对象ID。消息处理函数的原型是 void (CCmdUI\*)，参数指向一个CCmdUI对象，不返回值。另外还有ON\_UPDATE\_COMMAND\_UI\_RANGE宏，这里暂不讨论。

<5>用于其他消息的宏  
例如用于用户定义消息的ON\_MESSAGE。这类宏带有参数，需要指定消息ID和消息处理函数。消息映射条目的第一个成员nMessage被指定为消息ID，第二个成员nCode被指定为0，第三个成员nID和第四个成员也是0。消息处理的原型是LRESULT (WPARAM, LPARAM)，参数1和参数2是消息参数wParam和lParam，返回LRESULT类型的值。

<6>扩展消息映射宏  
很多普通消息映射宏都有对应的扩展消息映射宏，例如：ON\_COMMAND对应的ON\_COMMAND\_EX，ON\_ONTIFY对应的ON\_ONTIFY\_EX，等等。扩展宏除了具有普通宏的功能，还有特别的用途，这里暂不讨论。

（4）CcmdTarget类

所有响应消息或事件的类都从它派生。例如，CWinapp，CWnd，CDocument，CView，CDocTemplate，CFrameWnd，等等。CCmdTarget类是MFC处理命令消息的基础、核心。MFC为该类设计了许多成员函数和一些成员数据，基本上是为了解决消息映射问题的。

CCmdTarget有两个与消息映射有密切关系的成员函数：DispatchCmdMsg和OnCmdMsg。  
<1>静态成员函数DispatchCmdMsg  
CCmdTarget的静态成员函数DispatchCmdMsg，用来分发Windows消息。此函数是MFC内部使用的，其原型如下：  
static BOOL DispatchCmdMsg(  
CCmdTarget\* pTarget,  
UINT nID,  
int nCode,  
AFX\_PMSG pfn,  
void\* pExtra,  
UINT nSig,  
AFX\_CMDHANDLERINFO\* pHandlerInfo)

<2>虚拟函数OnCmdMsg  
CCmdTarget的虚拟函数OnCmdMsg，用来传递和发送消息、更新用户界面对象的状态，其原型如下：  
OnCmdMsg(  
UINT nID,  
int nCode,  
void\* pExtra,  
AFX\_CMDHANDLERINFO\* pHandlerInfo)  
框架的命令消息传递机制主要是通过该函数来实现的。

CCmdTarget对OnCmdMsg的默认实现简单描述是：在当前命令目标(this所指，也即处理消息的对象)的类和基类的消息映射数组里搜索指定命令消息的消息处理函数（标准Windows消息不会送到这里处理）。

这里使用虚拟函数GetMessageMap得到命令目标类的消息映射入口数组\_messageEntries，然后在数组里匹配指定的消息映射条目。匹配标准：命令消息ID相同，控制通知代码相同。因为GetMessageMap是虚拟函数，所以可以确认当前命令目标的确切类。  
如果找到了一个匹配的消息映射条目，则使用DispachCmdMsg调用这个处理函数；  
如果没有找到，则使用\_GetBaseMessageMap得到基类的消息映射数组，查找，直到找到或搜寻了所有的基类（到CCmdTarget）为止；  
如果最后没有找到，则返回FASLE。

每个从CCmdTarget派生的命令目标类都可以覆盖OnCmdMsg，利用它来确定是否可以处理某条命令，如果不能，就通过调用下一命令目标的OnCmdMsg，把该命令送给下一个命令目标处理。通常，派生类覆盖OnCmdMsg时，要调用基类的被覆盖的OnCmdMsg。  
在MFC框架中，一些MFC命令目标类覆盖了OnCmdMsg，如框架窗口类覆盖了该函数，实现了MFC的标准命令消息发送路径。

（5）MFC窗口过程

所有的消息都送给窗口过程处理，MFC的所有窗口都使用同一窗口过程，消息或者直接由窗口过程调用相应的消息处理函数处理，或者按MFC命令消息派发路径送给指定的命令目标处理。那么，MFC的窗口过程是什么？怎么处理标准Windows消息？怎么实现命令消息的派发？请看下面。

按理说，每一个“窗口类”都有自己的窗口过程。正常情况下使用该“窗口类”创建的窗口都使用它的窗口过程。

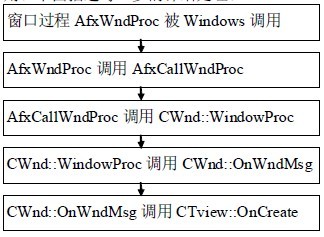
那么，为什么说MFC创建的所有HWND窗口使用同一个窗口过程呢？  
在MFC中，的确所有的窗口都使用同一个窗口过程：AfxWndProc或AfxWndProcBase（如果定义了\_AFXDLL）。它们的原型如下：  
LRESULT CALLBACK  
AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam)   //记住有一个HWND参数  
LRESULT CALLBACK  //不是dll时不用管这个函数  
AfxWndProcBase(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam)

上面的窗口过程和win32的标准窗口过程是一样的，只有四个参数，并且是msg结构的前四个成员，所以四个参数的值其实都是来自一个从消息队列取出的msg结构变量，也即四个参数的值，其实都是操作系给的值到msg，然后传到这四个参数。

实际上在窗体初始化过程中通过复杂的过程（有空研究这个复杂的过程吧，好像涉及到Hook）调用了::SetWindowLong函数人为的设置窗口过程为AfxWndProc，并保存原窗口过程在窗口类成员变量m\_pfnSuper中，这样形成一个窗口过程链。需要的时候，原窗口过程地址可以通过窗口类成员函数GetSuperWndProcAddr得到。  
这样，AfxWndProc就成为CWnd或其派生类的窗口过程。不论队列消息，还是非队列消息，都送到AfxWndProc窗口过程来处理（如果使用MFC DLL，则AfxWndProcBase被调用，然后是AfxWndProc）。经过消息分发之后没有被处理的消息，将送给原窗口过程（m\_pfnSuper）处理。

（6）对Windows消息的接收和处理

Windows消息送给AfxWndProc窗口过程之后，AfxWndProc得到HWND窗口对应的MFC窗口对象，然后，搜索该MFC窗口对象和其基类的消息映射数组，判定它们是否处理当前消息，如果是则调用对应的消息处理函数，否则，进行缺省处理。  
下面，以一个应用程序的视窗口创建时，对WM\_CREATE消息的处理为例，详细地讨论Windows消息的分发过程。  
比如类CTview要处理WM\_CREATE消息，使用ClassWizard加入消息处理函数CTview::OnCreate。下面，看这个函数怎么被调用：



下面是对上面过程的分析：

<1>首先，分析AfxWndProc窗口过程函数。  
􀁺 AfxWndProc的原型如下：  
LRESULT AfxWndProc(HWND hWnd,  
UINT nMsg, WPARAM wParam, LPARAM lParam)  
如果收到的消息nMsg不是WM\_QUERYAFXWNDPROC（该消息被MFC内部用来确认窗口过程是否使用AfxWndProc），则从hWnd得到对应的MFC Windows对象指针pWnd。pWnd所指的MFC窗口对象将负责完成消息的处理。这里，pWnd所指示的对象是MFC视窗口对象，即CTview对象。  
然后，把pWnd和AfxWndProc接受的四个参数传递给函数AfxCallWndProc执行。

<2>AfxCallWndProc原型如下：  
LRESULT AFXAPI AfxCallWndProc(CWnd\* pWnd, HWND hWnd,  
UINT nMsg, WPARAM wParam = 0, LPARAM lParam = 0)  
MFC使用AfxCallWndProc函数把消息送给CWnd类或其派生类的对象。该函数主要是把消息和消息参数(nMsg、wParam、lParam)传递给MFC窗口对象的成员函数WindowProc  
（pWnd->WindowProc）作进一步处理。如果是WM\_INITDIALOG消息，则在调用WindowProc前后要作一些处理。

 <3>WindowProc的函数原型如下：  
LRESULT CWnd::WindowProc(UINT message,  //他是具体窗口的函数了，所以就不需要窗口句柄参数了  
WPARAM wParam, LPARAM lParam)  
这是一个虚拟函数，程序员可以在CWnd的派生类中覆盖它，改变MFC分发消息的方式。例如，MFC的CControlBar就覆盖了WindowProc，对某些消息作了自己的特别处理，其他消息处理由基类的WindowProc函数完成。  
但是在当前例子中，当前对象的类CTview没有覆盖该函数，所以CWnd的WindowProc被调用。  
这个函数把下一步的工作交给OnWndMsg函数来处理。如果OnWndMsg没有处理，则交给DefWindowProc来处理。  
OnWndMsg和DefWindowProc都是CWnd类的虚拟函数。

<4>OnWndMsg的原型如下：  
BOOL CWnd::OnWndMsg( UINT message,  
WPARAM wParam, LPARAM lParam,RESULT\*pResult );  
该函数是虚拟函数。  
和WindowProc一样，由于当前对象的类CTview没有覆盖该函数，所以CWnd的OnWndMsg被调用。  
在CWnd中，MFC使用OnWndMsg来分别处理各类消息：  
如果是WM\_COMMAND消息，交给OnCommand处理；然后返回。  
如果是WM\_NOTIFY消息，交给OnNotify处理；然后返回。  
如果是WM\_ACTIVATE消息，先交给\_AfxHandleActivate处理（后面5.3.3.7节会解释它的处理），再继续下面的处理。  
如果是WM\_SETCURSOR消息，先交给\_AfxHandleSetCursor处理；然后返回。  
如果是其他的Windows消息（包括WM\_ACTIVATE），则首先在消息缓冲池进行消息匹配，若匹配成功，则调用相应的消息处理函数；若不成功，则在消息目标的消息映射数组中进行查找匹配，看它是否处理当前消息。这里，消息目标即CTview对象。如果消息目标处理了该消息，则会匹配到消息处理函数，调用它进行处理；否则，该消息没有被应用程序处理，OnWndMsg返回FALSE。

（7）Windows消息的查找和匹配

CWnd或者派生类的对象调用OnWndMsg搜索本对象或者基类的消息映射数组，寻找当前消息的消息处理函数。如果当前对象或者基类处理了当前消息，则必定在其中一个类的消息映射数组中匹配到当前消息的处理函数。具体过程复杂，暂不讨论。

（8）Windows消息处理函数的调用  
对一个Windows消息，匹配到了一个消息映射条目之后，将调用映射条目所指示的消息处理函数。  
调用处理函数的过程就是转换映射条目的pfn指针为适当的函数类型并执行它：MFC定义了一个成员函数指针mmf，首先把消息处理函数的地址赋值给该函数指针，然后根据消息映射条目的nSig值转换指针的类型。但是，要给函数指针mmf赋值，必须使该指针可以指向所有的消息处理函数，为此则该指针的类型是所有类型的消息处理函数指针的联合体。  
对上述过程，MFC的实现大略如下：  
union MessageMapFunctions mmf；  
mmf.pfn = lpEntry->pfn;  
switch (value\_of\_nsig){  
…  
case AfxSig\_is: //OnCreate就是该类型  
lResult = (this->\*mmf.pfn\_is)((LPTSTR)lParam);  
break;  
…  
default:  
ASSERT(FALSE); break;  
}

分析，对于上面的例子，nSig等于AfxSig\_is，所以将执行语句  
(this->\*mmf.pfn\_is)((LPTSTR)lParam)  
也就是对CTview::OnCreate的调用。

（9）总结

综上所述，标准Windwos消息和应用程序消息中的Registered消息，由窗口过程直接调用相应的处理函数处理：  
如果某个类型的窗口（C++类）处理了某条消息（覆盖了CWnd或直接基类的处理函数），则对应的HWND窗口（Winodws window）收到该消息时就调用该覆盖函数来处理；如果该类窗口没有处理该消息，则调用实现该处理函数最直接的基类（在C++的类层次上接近该类）来处理，上述例子中如果CTview不处理WM\_CREATE消息，则调用上一层的CWnd::OnCreate处理；  
如果基类都不处理该消息，则调用DefWndProc来处理。

综合对Windows消息的处理来看，MFC使用消息映射机制完成了C++虚拟函数的功能。这主要基于以下几点：  
􀁺 所有处理消息的类从CCmdTarget派生。  
􀁺 使用静态成员变量\_messageEntries数组存放消息映射条目，使用静态成员变量messageMap来唯一地区别和得到类的消息映射。【注意，多次提到messageMap，其实messageMap可以成为消息映射大表，但它其实是非常小的，就含有两个成员，一个是pBaseMap用于指向父类的消息映射总表，一个是lpEntries用来指向\_messageEntries[]数组。】  
􀁺 通过GetMessage虚拟函数来获取当前对象的类的messageMap变量，进而得到消息映射入口。  
􀁺 按照先底层，后基层的顺序在类的消息映射数组中搜索消息处理函数。基于这样的机制，一般在覆盖基类的消息处理函数时，应该调用基类的同名函数。  
以上论断适合于MFC其他消息处理机制，如对命令消息的处理等。不同的是其他消息处理有一个命令派发/分发的过程。  
下一节，讨论命令消息的接受和处理。

（10）对命令消息的接收和处理