第 4 章　消息映射的实现

4.1　Windows消息概述

Windows 应用程序的输入由 Windows 系统以消息的形式发送给应用程序的窗口。这些窗口通过窗口过程来接收和处理消息，然后把控制返还给 Windows。

4.1.1　消息的分类

（1）　队列消息和非队列消息

从消息的发送途径上看，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。

（2）　系统消息和应用程序消息

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。

4.1.2　消息结构和消息处理

（1）　消息的结构

为了从消息队列获取消息信息，需要使用 MSG 结构。例如，::GetMessage 函数（从消息队列得到消息并从队列中移走）和::PeekMessage 函数(从消息队列得到消息但是可以不移走)都使用了该结构来保存获得的消息信息。

MSG 结构的定义如下：

```
typedef struct tagMSG {        // msg
    HWND     hwnd;
    UINT     message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD    time;
    POINT    pt;
} MSG;
```

该结构包括了六个成员，用来描述消息的有关属性：

接收消息的窗口句柄、消息标识（ID）、第一个消息参数、第二个消息参数、消息产生的时间、消息产生时鼠标的位置。

（2） 应用程序通过窗口过程来处理消息

如前所述，每个“窗口类”都要登记一个如下形式的窗口过程：

```
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
)
```

应用程序通过窗口过程来处理消息：非队列消息由 Windows 直接送给目的窗口的窗口过程，队列消息由::DispatchMessage 等派发给目的窗口的窗口过程。窗口过程被调用时，接受四个参数：

a window handle（窗口句柄）；

a message identifier（消息标识）；

two 32-bit values called message parameters（两个 32 位的消息参数）；

需要的话，窗口过程用::GetMessageTime 获取消息产生的时间，用::GetMessagePos 获取消息产生时鼠标光标所在的位置。

在窗口过程里，用 switch/case 分支处理语句来识别和处理消息。

（3） 应用程序通过消息循环来获得对消息的处理

每个 GDI 应用程序在主窗口创建之后，都会进入消息循环，接受用户输入、解释和处理消息。

消息循环的结构如下：

```
while (GetMessage(&msg, (HWND) NULL, 0, 0)) {//从消息队列得到消息
    if (hwndDlgModeless == (HWND) NULL ||
        !IsDialogMessage(hwndDlgModeless, &msg) &&
        !TranslateAccelerator(hwndMain, haccel, &msg)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg); //发送消息
}
}
```

消息循环从消息队列中得到消息，如果不是快捷键消息或者对话框消息，就进行消息转换和派发，让目的窗口的窗口过程来处理。

当得到消息 WM_QUIT，或者::GetMessage 出错时，退出消息循环。

（4） MFC 消息处理

使用 MFC 框架编程时，消息发送和处理的本质也如上所述。但是，有一点需要强调的是，所有的 MFC 窗口都使用同一窗口过程，程序员不必去设计和实现自己的窗口过程，而是通过 MFC 提供的一套消息映射机制来处理消息。因此，MFC 简化了程序员编程时处理消息的复杂性。

所谓消息映射，简单地讲，就是让程序员指定要某个 MFC 类（有消息处理能力的类）处理某个消息。MFC 提供了工具 ClassWizard 来帮助实现消息映射，在处理消息的类中添加一些有关消息映射的内容和处理消息的成员函数。程序员将完成消息处理函数，实现所希望的消息处理能力。

如果派生类要覆盖基类的消息处理函数，就用 ClassWizard 在派生类中添加一个消息映射条目，用同样的原型定义一个函数，然后实现该函数。这个函数覆盖派生类的任何基类的同名处理函数。

下面几节将分析 MFC 的消息机制的实现原理和消息处理的过程。为此，首先要分析 ClassWizard 实现消息映射的内幕，然后讨论 MFC 的窗口过程，分析 MFC 窗口过程是如何

实现消息处理的。

4.2 消息映射的定义和实现

4.2.1 MFC处理的三类消息

根据处理函数和处理过程的不同，MFC 主要处理三类消息：

- 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。

需要指出的是，Win32 使用新的 WM_NOFITY 来处理复杂的通知消息。WM_COMMAND 类型的通知消息仅仅能传递一个控制窗口句柄(lparam)、控制窗 ID 和通知代码(wparam)。WM_NOTIFY 能传递任意复杂的信息。

- 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。

在讨论了消息的分类之后，应该是讨论各类消息如何处理的时候了。但是，要知道怎么处理消息，首先要知道如何映射消息。

4.2.2 MFC消息映射的实现方法

MFC 使用 ClassWizard 帮助实现消息映射，它在源码中添加一些消息映射的内容，并声明和实现消息处理函数。现在来分析这些被添加的内容。
在类的定义（头文件）里，它增加了消息处理函数声明，并添加一行声明消息映射的宏 DECLARE_MESSAGE_MAP。
在类的实现（实现文件）里，实现消息处理函数，并使用 IMPLEMENT_MESSAGE_MAP 宏实现消息映射。一般情况下，这些声明和实现是由 MFC 的 ClassWizard 自动来维护的。
看一个例子：
在 AppWizard 产生的应用程序类的源码中，应用程序类的定义（头文件）包含了类似如下的代码：

```
//{{AFX_MSG(CTttApp)
afx_msg void OnAppAbout();
//}}AFX_MSG
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)
//}}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。
为什么这样做之后就完成了一个消息映射？这些声明和实现到底作了些什么呢？接着，将讨论这些问题。

4.2.3 在声明与实现的内部

（1） DECLARE_MESSAGE_MAP 宏：
首先，看 DECLARE_MESSAGE_MAP 宏的内容：

```
#ifdef _AFXDLL
#define DECLARE_MESSAGE_MAP() \
private: \
    static const AFX_MSGMAP_ENTRY _messageEntries[]; \
protected: \
static AFX_DATA const AFX_MSGMAP messageMap; \
static const AFX_MSGMAP* PASCAL _GetBaseMessageMap(); \
virtual const AFX_MSGMAP* GetMessageMap() const; \

#else
#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; \

#endif
```

DECLARE_MESSAGE_MAP 定义了两个版本，分别用于静态或者动态链接到 MFC DLL 的情形。

（2） BEGIN_MESSAE_MAP 宏
然后，看 BEGIN_MESSAE_MAP 宏的内容：

```
#ifdef _AFXDLL
#define BEGIN_MESSAGE_MAP(theClass, baseClass) \
const AFX_MSGMAP* PASCAL theClass::_GetBaseMessageMap() \
{ return &baseClass::messageMap; } \
const AFX_MSGMAP* theClass::GetMessageMap() const \
{ return &theClass::messageMap; } \
AFX_DATADEF const AFX_MSGMAP theClass::messageMap = \
    { &theClass::_GetBaseMessageMap, &theClass::_messageEntries[0] }; \
const AFX_MSGMAP_ENTRY theClass::_messageEntries[] = \
{ \

#else
#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[] = \
{ \


#endif


#define END_MESSAGE_MAP() \
        {0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
    }; \
```

对应地，BEGIN_MESSAGE_MAP 定义了两个版本，分别用于静态或者动态链接到 MFC DLL 的情形。END_MESSAGE_MAP 相对简单，就只有一种定义。

（3） ON_COMMAND 宏

最后，看 ON_COMMAND 宏的内容：

```
#define ON_COMMAND(id, memberFxn) \
{\
    WM_COMMAND,\
    CN_COMMAND,\
    (WORD)id,\
    (WORD)id,\
    AfxSig_vv,\
    (AFX_PMSG)memberFxn\
};
```

4.2.3.1 消息映射声明的解释

在清楚了有关宏的定义之后，现在来分析它们的作用和功能。

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，当然它们是与消息映射相关的变量和函数。

（1） 成员变量

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

- 第一个成员变量的声明：

AFX_MSGMAP_ENTRY _messageEntries[]

这是一个 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
{
//Windows 消息 ID
UINT nMessage;
//控制消息的通知码
UINT nCode;
//Windows Control 的 ID
UINT nID;
//如果是一定范围的消息被映射，则 nLastID 指定其范围
```

```
UINT nLastID;

UINT nSig;//消息的动作标识
//响应消息时应执行的函数(routine to call (or special value))
AFX_PMSG pfn;
};
```

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;

这是一个 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。AFX_MSGMAP 结构的定义如下：

```
struct AFX_MSGMAP
{
//得到基类的消息映射入口地址的数据或者函数
#ifdef _AFXDLL
//pfnGetBaseMap 指向_GetBaseMessageMap 函数
const AFX_MSGMAP* (PASCAL* pfnGetBaseMap)();
#else
//pBaseMap 保存基类消息映射入口_messageEntries 的地址
const AFX_MSGMAP* pBaseMap;
#endif
//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 和函数_GetBaseMessageMap 的地址（不使用 MFC DLL 时，是基类消息映射数组的地址）。

（2） **成员函数**

- _GetBaseMessageMap()

用来得到基类消息映射的函数。

- GetMessageMap()

用来得到自身消息映射的函数。

4.2.3.2 消息映射实现的解释

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，实现所声明的静态或虚拟函数 GetMessageMap、_GetBaseMessageMap。
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） 对消息映射入口表(消息映射数组)的初始化

如前所述，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：如果指定当前类处理某个消息，则把和该消息有关的信息（四个）和消息处理函数的地址及原型组合成为一个消息映射条目，加入到消息映射数组中。
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。ON_COMMAND 宏的初始化内容如下：

```
{WM_COMMAND,
CN_COMMAND,
(WORD)ID_APP_ABOUT,
(WORD)ID_APP_ABOUT,
AfxSig_vv,
(AFX_PMSG)OnAppAbout
}
```

这个消息映射条目的含义是：消息 ID 是 ID_APP_ABOUT，OnAppAbout 被转换成 AFX_PMSG 指针类型，AfxSig_vv 是 MFC 预定义的枚举变量，用来标识 OnAppAbout 的函数类型为参数空(Void)、返回空(Void)。
在消息映射数组的最后，是宏 END_MESSAGE_MAP 的内容，它标识消息处理类的消息映射条目的终止。

（2） 对 messageMap 的初始化

如前所述，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；如果动态链接到 MFC DLL，则 pfnGetBaseMap 保存了_GetBaseMessageMap 成员函数的地址；否则 pBaseMap 保存了基类的消息映射数组的地址。

（3） 对函数的实现

_GetBaseMessageMap()
它返回基类的成员变量 messagMap（当使用 MFC DLL 时），使用该函数得到基类消息映射入口表。

GetMessageMap()：
它返回成员变量 messageMap，使用该函数得到自身消息映射入口表。

顺便说一下，消息映射类的基类 CCmdTarget 也实现了上述和消息映射相关的函数，不过，它的消息映射数组是空的。

既然消息映射宏方便了消息映射的实现，那么有必要详细的讨论消息映射宏。下一节，介绍消息映射宏的分类、用法和用途。

4.2.4 消息映射宏的种类

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。

除了单条命令消息的映射，还有把一定范围的命令消息映射到一个消息处理函数的映射宏 ON_COMMAND_RANGE。这类宏带有参数，需要指定命令 ID 的范围和消息处理函数。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)，第三个成员 nID 和第四个成员 nLastID 指定了映射消息的起止范围。消息处理函数的原型是 void (UINT)，有一个 UINT 类型的参数，表示要处理的命令消息 ID，不返回值。

（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。

还有一类宏处理通知消息 ON_NOTIFY，它类似于 ON_CONTROL，但是控制通知消息被映射到 WM_NOTIFY。消息映射条目的第一个成员的 nMessage 被指定为 WM_NOTIFY，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 (NMHDR*, LRESULT*)，参数 1 是 NMHDR 指针，参数 2 是 LRESULT 指针，用于返回结果，但函数不返回值。

对应地，还有把一定范围的控制子窗口的某个通知消息映射到一个消息处理函数的映射宏，这类宏包括 ON__CONTROL_RANGE 和 ON_NOTIFY_RANGE。这类宏带有参数，需要指定控制子窗口 ID 的范围和通知消息，以及消息处理函数。

对于 ON__CONTROL_RANGE，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 和第四个成员 nLastID 等于指定了控制窗口 ID 的范围。消息处理函数的原型是 void (UINT)，参数表示要处理的通知消息是哪个 ID 的控制子窗口发送的，函数不返回值。

对于 ON__NOTIFY_RANGE，消息映射条目的第一个成员的 nMessage 被指定为 WM_NOTIFY，第二个成员 nCode 是特定的通知码，第三个成员 nID 和第四个成员 nLastID 指定了控制窗口 ID 的范围。消息处理函数的原型是 void (UINT, NMHDR*, LRESULT*)，参数 1 表示要处理的通知消息是哪个 ID 的控制子窗口发送的，参数 2 是 NMHDR 指针，参数 3 是 LRESULT 指针，用于返回结果，但函数不返回值。

（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。

对应地，有更新一定 ID 范围的用户接口对象的宏 ON_UPDATE_COMMAND_UI_RANGE，此宏带有三个参数，用于指定用户接口对象 ID 的范围和消息处理函数。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 用于指定用户接口对象 ID 的范围。消息处理函数的原型是 void (CCmdUI*)，参数指向一个 CCmdUI 对象，函数不返回值。之所以不用当前用户接口对象 ID 作为参数，是因为 CCmdUI 对象包含了有关信息。

（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。关于扩展宏的具体讨论和分析，见 4.4.3.2 节。

作为一个总结，下表列出了这些常用的消息映射宏。

表 4-1 常用的消息映射宏

消息映射宏	用途
ON_COMMAND	把 command message 映射到相应的函数
ON_CONTROL	把 control notification message 映射到相应的函数。MFC 根据不同的控制消息，在此基础上定义了更具体的宏，这样用户在使用时就不需要指定通知代码 ID，如 ON_BN_CLICKED。
ON_MESSAGE	把 user-defined message.映射到相应的函数
ON_REGISTERED_MESSAGE	把 registered user-defined message 映射到相应的函数，实际上 nMessage 等于 0x0C000，nSig 等于宏的消息参数。nSig 的真实值为 Afxsig_lwl。
ON_UPDATE_COMMAND_UI	把 user interface user update command message 映射到相应的函数上。
ON_COMMAND_RANGE	把一定范围内的 command IDs 映射到相应的函数上
ON_UPDATE_COMMAND_UI_RANGE	把一定范围内的 user interface user update command message 映射到相应的函数上
ON_CONTROL_RANGE	把一定范围内的 control notification message 映射到相应的函数上

在表 4-1 中，宏 ON_REGISTERED_MESSAGE 的定义如下：

```
#define ON_REGISTERED_MESSAGE(nMessageVariable, memberFxn) \
{ 0xC000, 0, 0, 0,\
(UINT)(UINT*)(&nMessageVariable), \
/*implied 'AfxSig_lwl'*/ \
(AFX_PMSG)(AFX_PMSGW)(LRESULT\
(AFX_MSG_CALL CWnd::*)\
(WPARAM, LPARAM))&memberFxn }
```

从上面的定义可以看出，实际上，该消息被映射到 WM_COMMAND(0XC000)，指定的 registered 消息 ID 存放在 nSig 域内，nSig 的值在这样的映射条目下隐含地定为 AfxSig_lwl。由于 ID 和正常的 nSig 域存放的值范围不同，所以 MFC 可以判断出是否是 registered 消息映射条目。如果是，则使用 AfxSig_lwl 把消息处理函数转换成参数 1 为 Word、参数 2 为 long、

返回值为 long 的类型。
在介绍完了消息映射的内幕之后，应该讨论消息处理过程了。由于 CCmdTarge 的特殊性和重要性，在 4.3 节先对其作一个大略的介绍。

4.3 CcmdTarget类

除了 CObject 类外，还有一个非常重要的类 CCmdTarget。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，而且，很大一部分是针对 OLE 设计的。在 OLE 应用中，CCmdTarget 是 MFC 处理模块状态的重要环节，它起到了传递模块状态的作用：其构造函数获取当前模块状态，并保存在成员变量 m_pModuleState 里头。关于模块状态，在后面章节讲述。
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)
```

关于此函数将在 4.4.3.2 章节命令消息的处理中作更详细的描述。

（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。其参数描述参见 4.3.3.2 章节 DispacthCMdMessage 的参数描述。
在本书中，命令目标指希望或者可能处理消息的对象；命令目标类指命令目标的类。
CCmdTarget 对 OnCmdMsg 的默认实现：在当前命令目标(this 所指)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。具体实现见后续章节。
必要的话，应用程序也可以覆盖 OnCmdMsg，改变一个或多个类中的发送规定，实现与标准框架发送规定不同的发送路径。例如，在以下情况可以作这样的处理：在要打断发送顺序的类中把命令传给一个非 MFC 默认对象；在新的非默认对象中或在可能要传出命令的命令目标中。
本节对 CCmdTarget 的两个成员函数作一些讨论，是为了对 MFC 的消息处理有一个大致印象。后面 4.4.3.2 节和 4.4.3.3 节将作进一步的讨论。

4.4 MFC窗口过程

前文曾经提到，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？这些都将是下文要回答的问题。

4.4.1 MFC窗口过程的指定

从前面的讨论可知，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。
MFC 的窗口对象在创建 HWND 窗口时，也使用了已经注册的“窗口类”，这些“窗口类”或者使用应用程序提供的窗口过程，或者使用 Windows 提供的窗口过程（例如 Windows 控制窗口、对话框等）。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)

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

这两个函数的原型都如 4.1.1 节描述的窗口过程一样。
如果动态链接到 MFC DLL(定义了_AFXDLL)，则 AfxWndProcBase 被用作窗口过程，否则 AfxWndProc 被用作窗口过程。AfxWndProcBase 首先使用宏 AFX_MANAGE_STATE 设置正确的模块状态，然后调用 AfxWndProc。

下面，假设不使用 MFC DLL，讨论 MFC 如何使用 AfxWndProc 取代各个窗口的原窗口过程。窗口过程的取代发生在窗口创建的过程时，使用了子类化(Subclass)的方法。所以，从窗口的创建过程来考察取代过程。从前面可以知道，窗口创建最终是通过调用 CWnd::CreateEx 函数完成的，分析该函数的流程，如图 4-1 所示。

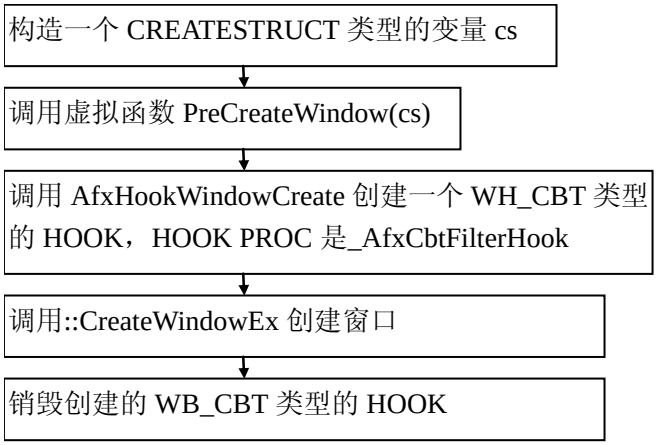

图 4-1　指定 MFC 的窗口过程

图 4-1 中的 CREATESTRUCT 结构类型的变量 cs 包含了传递给窗口过程的初始化参数。CREATESTRUCT 结构描述了创建窗口所需要的信息，定义如下：

```
typedef struct tagCREATESTRUCT {
LPVOID      lpCreateParams; //用来创建窗口的数据
HANDLE      hInstance; //创建窗口的实例
HMENU       hMenu; //窗口菜单
HWND        hwndParent; //父窗口
int         cy; //高度
int         cx; //宽度
int         y; //原点 Y 坐标
int         x;//原点 X 坐标
LONG        style; //窗口风格
LPCSTR      lpszName; //窗口名
LPCSTR      lpszClass; //窗口类
DWORD       dwExStyle; //窗口扩展风格
} CREATESTRUCT;
```

cs 表示的创建参数可以在创建窗口之前被程序员修改，程序员可以覆盖当前窗口类的虚拟成员函数 PreCreateWindow，通过该函数来修改 cs 的 style 域，改变窗口风格。这里 cs 的主要作用是保存创建窗口的各种信息，::CreateWindowEx 函数使用 cs 的各个域作为参数来创建窗口，关于该函数见 2.2.2 节。

在创建窗口之前，创建了一个 WH_CBT 类型的钩子（Hook）。这样，创建窗口时所有的消息都会被钩子过程函数_AfxCbtFilterHook 截获。

AfxCbtFilterHook 函数首先检查是不是希望处理的 Hook——HCBT_CREATEWND。如果是，则先把 MFC 窗口对象（该对象必须已经创建了）和刚刚创建的 Windows 窗口对象捆绑在一起，建立它们之间的映射（见后面模块-线程状态）；然后，调用::SetWindowLong 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，将送给原窗口过程处理。

最后，有一点可能需要解释：为什么不直接指定窗口过程为 AfxWndProc，而要这么大费周折呢？这是因为原窗口过程（“窗口类”指定的窗口过程）常常是必要的，是不可缺少的。

接下来，讨论 AfxWndProc 窗口过程如何使用消息映射数据实现消息映射。Windows 消息和

命令消息的处理不一样，前者没有消息分发的过程。

4.4.2 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。
用第一章的例子，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：
视窗口最终调用::CreateEx 函数来创建。由 Windows 系统发送 WM_CREATE 消息给视的窗口过程 AfxWndProc，参数 1 是创建的视窗口的句柄，参数 2 是消息 ID（WM_CREATE)，参数 3、4 是消息参数。图 4-2 描述了其余的处理过程。图中函数的类属限制并非源码中所具有的，而是根据处理过程得出的判断。例如，“CWnd::WindowProc”表示 CWnd 类的虚拟函数 WindowProc 被调用，并不一定当前对象是 CWnd 类的实例，事实上，它是 CWnd 派生类 CTview 类的实例；而“CTview::OnCreate”表示 CTview 的消息处理函数 OnCreate 被调用。下面描述每一步的详细处理。

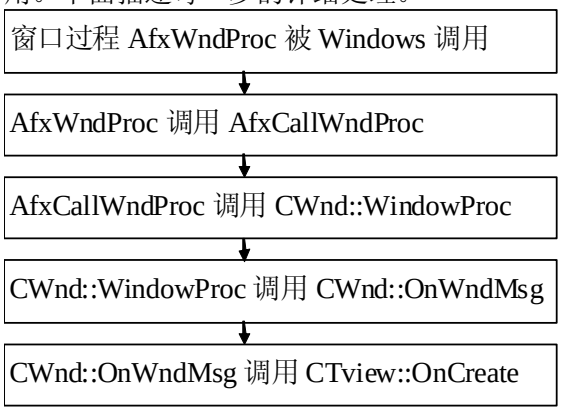

图 4-2　标准 Windows 消息的处理

4.4.2.1 从窗口过程到消息映射

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（该对象必须已存在，是永久性<Permanent>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。

- 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。

WindowProc 的函数原型如下：

```
LRESULT CWnd::WindowProc(UINT message,
    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。

- 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。

关于 Windows 消息和消息处理函数的匹配，见下一节。

缺省处理函数 DefWindowProc 将在讨论对话框等的实现时具体分析。

4.4.2.2 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。
消息匹配是一个比较耗时的任务，为了提高效率，MFC 设计了一个消息缓冲池，把要处理的消息和匹配到的消息映射条目（条目包含了消息处理函数的地址）以及进行消息处理的当前类等信息构成一条缓冲信息，放到缓冲池中。如果以后又有同样的消息需要同一个类处理，则直接从缓冲池查找到对应的消息映射条目就可以了。
MFC 用哈希查找来查询消息映射缓冲池。消息缓冲池相当于一个哈希表，它是应用程序

的一个全局变量，可以放 512 条最新用到的消息映射条目的缓冲信息，每一条缓冲信息是哈希表的一个入口。
采用 AFX_MSG_CACHE 结构描述每条缓冲信息，其定义如下：

```
struct AFX_MSG_CACHE
{
UINT nMsg;
const AFX_MSGMAP_ENTRY* lpEntry;
const AFX_MSGMAP* pMessageMap;
};
```

nMsg 存放消息 ID，每个哈希表入口有不同的 nMsg。
lpEnty 存放和消息 ID 匹配的消息映射条目的地址，它可能是 this 所指对象的类的映射条目，也可能是这个类的某个基类的映射条目，也可能是空。
pMessageMap 存放消息处理函数匹配成功时进行消息处理的当前类（this 所指对象的类）的静态成员变量 messageMap 的地址，它唯一的标识了一个类（每个类的 messageMap 变量都不一样）。
this 所指对象是一个 CWnd 或其派生类的实例，是正在处理消息的 MFC 窗口对象。
哈希查找：使用消息 ID 的值作为关键值进行哈希查找，如果成功，即可从 lpEntry 获得消息映射条目的地址，从而得到消息处理函数及其原型。
如何判断是否成功匹配呢？有两条标准：
第一，当前要处理的消息 message 在哈希表（缓冲池）中有入口；第二，当前窗口对象（this 所指对象）的类的静态变量 messageMap 的地址应该等于本条缓冲信息的 pMessagMap。MFC 通过虚拟函数 GetMessagMap 得到 messageMap 的地址。

如果在消息缓冲池中没有找到匹配，则搜索当前对象的消息映射数组，看是否有合适的消息处理函数。
如果匹配到一个消息处理函数,则把匹配结果加入到消息缓冲池中，即填写该条消息对应的哈希表入口：

```
nMsg=message;
pMessageMap=this->GetMessageMap;
lpEntry=查找结果
```

然后，调用匹配到的消息处理函数。否则（没有找到），使用_GetBaseMessageMap 得到基类的消息映射数组，查找和匹配；直到匹配成功或搜寻了所有的基类（到 CCmdTarget）为止。如果最后没有找到，则也把该条消息的匹配结果加入到缓冲池中。和匹配成功不同的是：指定 lpEntry 为空。这样 OnWndMsg 返回，把控制权返还给 AfxCallWndProc 函数，AfxCallWndProc 将继续调用 DefWndProc 进行缺省处理。

消息映射数组的搜索在 CCmdTarget::OnCmdMsg 函数中也用到了，而且算法相同。为了提高速度，MFC 把和消息映射数组条目逐一比较、匹配的函数 AfxFindMessageEntry 用汇编书写。

```
const AFX_MSGMAP_ENTRY* AFXAPI
AfxFindMessageEntry(const AFX_MSGMAP_ENTRY* lpEntry,
        UINT nMsg, UINT nCode, UINT nID)
```

第一个参数是要搜索的映射数组的入口；第二个参数是 Windows 消息标识；第三个参数是控制通知消息标识；第四个参数是命令消息标识。
对 Windows 消息来说，nMsg 是每条消息不同的，nID 和 nCode 为 0。
对命令消息来说，nMsg 固定为 WM_COMMAND，nID 是每条消息不同，nCode 都是 CN_COMMAND（定义为 0）。
对控制通知消息来说，nMsg 固定为 WM_COMMAND 或者 WM_NOTIFY，nID 和 nCode 是每条消息不同。

对于 Register 消息，nMsg 指定为 0XC000，nID 和 nCode 为 0。在使用函数 AfxFindMessageEntry 得到匹配结果之后，还必须判断 nSig 是否等于 message，只有相等才调用对应的消息处理函数。

4.4.2.3 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;
swithc (value_of_nsig){
    …
    case AfxSig_is:   //OnCreate 就是该类型
        lResult = (this->*mmf.pfn_is)((LPTSTR)lParam);
        break;
    …
default:
    ASSERT(FALSE); break;
}
…
LDispatchRegistered:      // 处理 registered windows messages
    ASSERT(message >= 0xC000);
    mmf.pfn = lpEntry->pfn;
    lResult = (this->*mmf.pfn_lwl)(wParam, lParam);
    …
```

如果消息处理函数有返回值，则返回该结果，否则，返回 TRUE。
对于图 4-1 所示的例子，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。
顺便指出，对于 Registered 窗口消息，消息处理函数都是同一原型，所以都被转换成 lwl 型（关于 Registered 窗口消息的映射，见 4.4.2 节）。

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。

4.4.2.4 消息映射机制完成虚拟函数功能的原理

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。
- 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。

4.4.3 对命令消息的接收和处理

4.4.3.1 MFC标准命令消息的发送

在 SDI 或者 MDI 应用程序中，命令消息由用户界面对象（如菜单、工具条等）产生，然后送给主边框窗口。主边框窗口使用标准 MFC 窗口过程处理命令消息。窗口过程把命令传递给 MFC 主边框窗口对象，开始命令消息的分发。MFC 边框窗口类 CFrameWnd 提供了消息分发的能力。
下面，还是通过一个例子来说明命令消息的处理过程。
使用 AppWizard 产生一个单文档应用程序 t。从 help 菜单选择“About”，就会弹出一个 ABOUT 对话框。下面，讨论从命令消息的发出到对话框弹出的过程。
首先，选择“About”菜单项的动作导致一个 Windows 命令消息 ID_APP_ABOUT 的产生。Windows 系统发送该命令消息到边框窗口，导致它的窗口过程 AfxWndProc 被调用，参数 1 是边框窗口的句柄，参数 2 是消息 ID（即 WM_COMMAND），参数 3、4 是消息参数，参数 3 的值是 ID_APP_ABOUT。接着的系列调用如图 4-3 所示。

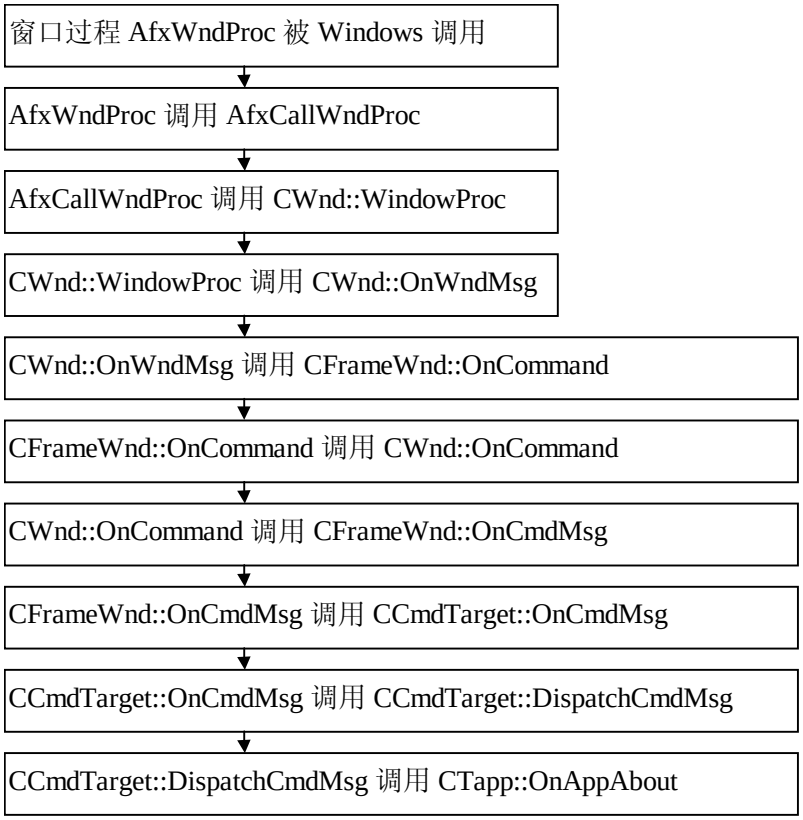

图 4-3　命令消息的处理

下面分别讲述每一层所调用的函数。
前 4 步同对 Windows 消息的处理。这里接受消息的 HWND 窗口是主边框窗口，因此，AfxWndProc 根据 HWND 句柄得到的 MFC 窗口对象是 MFC 边框窗口对象。
在 4.2.2 节谈到，如果 CWnd::OnWndMsg 判断要处理的消息是命令消息(WM_COMMAND)，就调用 OnCommand 进一步处理。由于 OnCommand 是虚拟函数，当前 MFC 窗口对象是边框窗口对象，它的类从 CFrameWnd 类导出，没有覆盖 CWnd 的虚拟函数 OnCommand，而 CFrameWnd 覆盖了 CWnd 的 OnCommand，所以，CFrameWnd 的 OnCommand 被调用。换句话说，CFrameWnd 的 OnCommand 被调用是动态约束的结果。接着介绍的本例子的有关调用，也是通过动态约束而实际发生的函数调用。
接着的有关调用，将不进行为什么调用某个类的虚拟或者消息处理函数的分析。

（1）CFrameWnd 的 OnCommand 函数

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

参数 wParam 的低阶 word 存放了菜单命令 nID 或控制子窗口 ID；如果消息来自控制窗口，高阶 word 存放了控制通知消息；如果消息来自加速键，高阶 word 值为 1；如果消息来自菜单，高阶 word 值为 0。
如果是通知消息，参数 lParam 存放了控制窗口的句柄 hWndCtrl，其他情况下 lParam 是 0。
在这个例子里，低阶 word 是 ID_APP_ABOUT，高阶 word 是 1；lParam 是 0。
MFC 对 CFrameWnd 的缺省实现主要是获得一个机会来检查程序是否运行在 HELP 状态，需要执行上下文帮助，如果不需要，则调用基类的 CWnd::OnCommand 实现正常的命令消息发送。

（2）CWnd 的 OnCommand 函数

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

它按一定的顺序处理命令或者通知消息，如果发送成功，返回 TRUE，否则，FALSE。处理顺序如下：
如果是命令消息，则调用 OnCmdMsg(nID, CN_UPDATE_COMMAND_UI, &state, NULL)测

试 nID 命令是否已经被禁止，如果这样，返回 FALSE；否则，调用 OnCmdMsg 进行命令发送。关于 CN_UPDATE_COMMAND_UI 通知消息，见后面用户界面状态的更新处理。
如果是控制通知消息，则先用 ReflectLastMsg 反射通知消息到子窗口。如果子窗口处理了该消息，则返回 TRUE；否则，调用 OnCmdMsg 进行命令发送。关于通知消息的反射见后面 4.4.4.3 节。OnCommand 给 OnCmdMsg 传递四个参数：nID，即命令消息 ID；nCode，如果是通知消息则为通知代码，如果是命令消息则为 NC_COMMAND(即 0)；其余两个参数为空。

（3）CFrameWnd 的 OnCmdMsg 函数

```
BOOL CFrameWnd::OnCmdMsg(UINT nID, int nCode, void* pExtra,
AFX_CMDHANDLERINFO* pHandlerInfo)
```

参数 1 是命令 ID；如果是通知消息（WM_COMMAND 或者 WM_NOTIFY），则参数 2 表示通知代码，如果是命令消息，参数 2 是 0；如果是 WM_NOTIFY，参数 3 包含了一些额外的信息；参数 4 在正常消息处理中应该是空。
在这个例子里，参数 1 是命令 ID，参数 2 为 0，参数 3 空。
OnCmdMsg 是虚拟函数，CFrameWnd 覆盖了该函数，当前对象（this 所指）是 MFC 单文档的边框窗口对象。故 CFrameWnd 的 OnCmdMsg 被调用。CFrameWnd::OnCmdMsg 在 MFC 消息发送中占有非常重要的地位，MFC 对该函数的缺省实现确定了 MFC 的标准命令发送路径：

第一，送给活动（Active）视处理，调用活动视的 OnCmdMsg。由于当前对象是 MFC 视对象，所以，OnCmdMsg 将搜索 CTview 及其基类的消息映射数组，试图得到相应的处理函数。

第二，如果视对象自己不处理，则视得到和它关联的文档，调用关联文档的 OnCmdMsg。由于当前对象是 MFC 视对象，所以，OnCmdMsg 将搜索 CTdoc 及其基类的消息映射数组，试图得到相应的处理函数。

第三，如果文档对象不处理，则它得到管理文档的文档模板对象，调用文档模板的 OnCmdMsg。由于当前对象是 MFC 文档模板对象，所以，OnCmdMsg 将搜索文档模板类及其基类的消息映射数组，试图得到相应的处理函数。

第四，如果文档模板不处理，则把没有处理的信息逐级返回：文档模板告诉文档对象，文档对象告诉视对象，视对象告诉边框窗口对象。最后，边框窗口得知，视、文档、文档模板都没有处理消息。

第五，CFrameWnd 的 OnCmdMsg 继续调用 *CWnd::OnCmdMsg*（斜体表示有类属限制）来处理消息。由于 CWnd 没有覆盖 OnCmdMsg，故实际上调用了函数 CCmdTarget::OnCmdMsg。由于当前对象是 MFC 边框窗口对象，所以 OnCmdMsg 函数将搜索 CMainFrame 类及其所有基类的消息映射数组，试图得到相应的处理函数。CWnd 没有实现 OnCmdMsg 却指定要执行其 OnCmdMsg 函数，可能是为了以后 MFC 给 CWnd 实现了 OnCmdMsg 之后其他代码不用改变。

这一步是边框窗口自己尝试处理消息。

第六，如果边框窗口对象不处理，则送给应用程序对象处理。调用 CTApp 的 OnCmdMsg，由于实际上 CTApp 及其基类 CWinApp 没有覆盖 OnCmdMsg，故实际上调用了函数 CCmdTarget::OnCmdMsg。由于当前对象是 MFC 应用程序对象，所以 OnCmdMsg 函数将搜索 CTApp 类及其所有基类的的消息映射入口数组，试图得到相应的处理函数

第七，如果应用程序对象不处理，则返回 FALSE，表明没有命令目标处理当前的命令消息。这样，函数逐级别返回，OnCmdMsg 告诉 OnCommand 消息没有被处理，OnCommand 告诉 OnWndMsg 消息没有被处理，OnWndMsg 告诉 WindowProc 消息没有被处理，于是 WindowProc 调用 DefWindowProc 进行缺省处理。

本例子在第六步中，应用程序对 ID_APP_ABOUT 消息作了处理。它找到处理函数 CTApp::OnAbout，使用 DispatchCmdMsg 派发消息给该函数处理。
如果是 MDI 边框窗口，标准发送路径还有一个环节，该环节和第二、三、四步所涉及的 OnCmdMsg 函数，将在下两节再次具体分析。

4.4.3.2 命令消息的派发和消息的多次处理

（1） 命令消息的派发

如前 3.1 所述，CCmdTarget 的静态成员函数 DispatchCmdMsg 用来派发命令消息给指定的命令目标的消息处理函数。

```
static BOOL DispatchCmdMsg(CCmdTarget* pTarget,
    UINT nID, int nCode,
    AFX_PMSG pfn, void* pExtra, UINT nSig,
    AFX_CMDHANDLERINFO* pHandlerInfo)
```

前面在讲 CCmdTarget 时，提到了该函数。这里讲述它的实现：

第一个参数指向处理消息的对象；第二个参数是命令 ID；第三个是通知消息等；第四个是消息处理函数地址；第五个参数用于存放一些有用的信息，根据 nCode 的值表示不同的意义，例如当消息是 WM_NOFITY，指向一个 NMHDR 结构（关于 WM_NOTIFY，参见 4.4.4.2 节通知消息的处理）；第六个参数标识消息处理函数原型；第七个参数是一个指针，指向 AFX_CMDHANDLERINFO 结构。前六个参数（除了第五个外）都是向函数传递信息，第五个和第七个参数是双向的，既向函数传递信息，也可以向调用者返回信息。

关于 AFX_CMDHANDLERINFO 结构：

```
struct AFX_CMDHANDLERINFO
{
CCmdTarget* pTarget;
void (AFX_MSG_CALL CCmdTarget::*pmf)(void);
};
```

第一个成员是一个指向命令目标对象的指针，第二个成员是一个指向 CCmdTarget 成员函数的指针。

该函数的实现流程可以如下描述：

首先，它检查参数 pHandlerInfo 是否空，如果不空，则用 pTarget 和 pfn 填写其指向的结构，返回 TRUE；通常消息处理时传递来的 pHandlerInfo 空，而在使用 OnCmdMsg 来测试某个对象是否处理某条命令时，传递一个非空的 pHandlerInfo 指针。若返回 TRUE，则表示可以处理那条消息。

如果 pHandlerInfo 空，则进行消息处理函数的调用。它根据参数 nSig 的值，把参数 pfn 的类型转换为要调用的消息处理函数的类型。这种指针转换技术和前面讲述的 Windows 消息的处理是一样的。

（2） 消息的多次处理

如果消息处理函数不返回值，则 DispatchCmdMsg 返回 TRUE；否则，DispatchCmdMsg 返回消息处理函数的返回值。这个返回值沿着消息发送相反的路径逐级向上传递，使得各个环节的 OnCmdMsg 和 OnCommand 得到返回的处理结果：TRUE 或者 FALSE，即成功或者失败。

这样就产生了一个问题，如果消息处理函数有意返回一个 FALSE，那么不就传递了一个错误的信息？例如，OnCmdMsg 函数得到 FALSE 返回值，就认为消息没有被处理，它将继续发送消息到下一环节。的确是这样的，但是这不是 MFC 的漏洞，而是有意这么设计的，用来处理一些特别的消息映射宏，实现同一个消息的多次处理。

通常的命令或者通知消息是没有返回值的（见 4.4.2 节的消息映射宏），仅仅一些特殊的消息处理函数具有返回值，这类消息的消息处理函数是使用扩展消息映射宏映射的，例如：

ON_COMMAND 对应的 ON_COMMAND_EX

扩展映射宏和对应的普通映射宏的参数个数相同，含义一样。但是扩展映射宏的消息处理函数的原型和对应的普通映射宏相比，有两个不同之处：一是多了一个 UINT 类型的参数，另外就是有返回值（返回 BOOL 类型）。回顾 4.4.2 章节，范围映射宏 ON_COMMAND_RANGE 的消息处理函数也有一个这样的参数，该参数在两处的含义是一样的，例如：命令消息扩展

映射宏 ON_COMMAND_EX 定义的消息处理函数解释该参数是当前要处理的命令消息 ID。有返回值的意义在于：如果扩展映射宏的消息处理函数返回 FALSE，则导致当前消息被发送给消息路径上的下一个消息目标处理。
综合来看，ON_COMMAND_EX 宏有两个功能：
一是可以把多个命令消息指定给一个消息处理函数处理。这类似于 ON_COMMAND_RANGE 宏的作用。不过，这里的多条消息的命令 ID 或者控制子窗口 ID 可以不连续，每条消息都需要一个 ON_COMMAND_EX 宏。
二是可以让几个消息目标处理同一个命令或者通知或者反射消息。如果消息发送路径上较前的命令目标不处理消息或者处理消息后返回 FALSE，则下一个命令目标将继续处理该消息。

对于通知消息、反射消息，它们也有扩展映射宏，而且上述论断也适合于它们。例如：
ON_NOTIFY 对应的 ON_NOTIFY_EX
ON_CONTROL 对应的 ON_CONTROL_EX
ON_CONTROL_REFLECT 对应的 ON_CONTROL_REFLECT_EX
等等。

范围消息映射宏也有对应的扩展映射宏，例如：
ON_NOTIFY_RANGE 对应的 ON_NOTIFY_EX_RANGE
ON_COMMAND_RANGE 对应的 ON_COMMAND_EX_RANGE
使用这些宏的目的在于利用扩展宏的第二个功能：实现消息的多次处理。

关于扩展消息映射宏的例子，参见 13.2..4.4 节和 13.2.4.6 节。

4.4.3.3 一些消息处理类的OnCmdMsg的实现

从以上论述知道，OnCmdMsg 虚拟函数在 MFC 命令消息的发送中扮演了重要的角色，CFrameWnd 的 OnCmdMsg 实现了 MFC 的标准命令消息发送路径。
那么，就产生一个问题：如果命令消息不送给边框窗口对象，那么就不会有按标准命令发送路径发送消息的过程？答案是肯定的。例如一个菜单被一个对话框窗口所拥有，那么，菜单命令将送给 MFC 对话框窗口对象处理，而不是 MFC 边框窗口处理，当然不会和 CFrameWnd 的处理流程相同。
但是，有一点需要指出，一般标准的 SDI 和 MDI 应用程序，只有主边框窗口拥有菜单和工具条等用户接口对象，只有在用户与用户接口对象进行交互时，才产生命令，产生的命令必然是送给 SDI 或者 MDI 程序的主边框窗口对象处理。
下面，讨论几个 MFC 类覆盖 OnCmdMsg 虚拟函数时的实现。这些类的 OnCmdMsg 或者可能是标准 MFC 命令消息路径的一个环节，或者可能是一个独立的处理过程（对于其中的 MFC 窗口类）。
从分析 CView 的 OnCmdMsg 实现开始。

- CView 的 OnCmdMsg

```
CView::OnCmdMsg(UINT nID, int nCode, void* pExtra,
    AFX_CMDHANDLERINFO* pHandlerInfo)
```

首先，调用 *CWnd::OnCmdMsg*，结果是搜索当前视的类和基类的消息映射数组，搜索顺序是从下层到上层。若某一层实现了对命令消息 nID 的处理，则调用它的实现函数；否则，调用 m_pDocument->OnCmdMsg，把命令消息送给文档类处理。m_pDocument 是和当前视关联的文档对象指针。如果文档对象类实现了 OnCmdMsg，则调用它的覆盖函数；否则，调用基类(例如 CDocument)的 OnCmdMsg。
接着，讨论 CDocument 的实现。

- CDocument 的 OnCmdMsg

```
BOOL CDocument::OnCmdMsg(UINT nID, int nCode, void* pExtra,
    AFX_CMDHANDLERINFO* pHandlerInfo)
```

首先，调用 *CCmdTarget::OnCmdMsg*，导致当前对象(this)的类和基类的消息映射数组被搜索，看是否有对应的消息处理函数可用。如果有，就调用它；如果没有，则调用文档模板的 OnCmdMsg 函数（m_pTemplate->OnCmdMsg）把消息送给文档模板处理。
MFC 文档模板没有覆盖 OnCmdMsg，导致基类 CCmdTarget 的 OnCmdMsg 被调用，看是否有文档模板类或基类实现了对消息的处理。是的话，调用对应的消息处理函数，否则，返回 FALSE。从前面的分析知道，CCmdTarget 类的消息映射数组是空的，所以这里返回 FALSE。

- CDialog 的 OnCmdMsg

```
BOOL CDialog::OnCmdMsg(UINT nID, int nCode, void* pExtra,
    AFX_CMDHANDLERINFO* pHandlerInfo)
```

第一，调用 *CWnd::OnCmdMsg*，让对话框或其基类处理消息。
第二，如果还没有处理，而且是控制消息或系统命令或非命令按钮，则返回 FALSE，不作进一步处理。否则，调用父窗口的 OnCmdmsg(GetParent()->OnCmdmsg)把消息送给父窗口处理。
第三，如果仍然没有处理，则调用当前线程的 OnCmdMsg(GetThread()->OnCmdMsg)把消息送给线程对象处理。
第四，如果最后没有处理，返回 FALSE。

- CMDIFrameWnd 的 OnCmdMsg

对于 MDI 应用程序，MDI 主边框窗口首先是把命令消息发送给活动的 MDI 文档边框窗口进行处理。MDI 主边框窗口对 OnCmdMsg 的实现函数的原型如下：

```
BOOL CMDIFrameWnd::OnCmdMsg(UINT nID, int nCode, void* pExtra,
    AFX_CMDHANDLERINFO* pHandlerInfo)
```

第一，如果有激活的文档边框窗口，则调用它的 OnCmdMsg(MDIGetActive（）->OnCmdMsg)把消息交给它进行处理。MFC 的文档边框窗口类并没有覆盖 OnCmdMsg 函数，所以基类 CFrameWnd 的函数被调用，导致文档边框窗口的活动视、文档边框窗口本身、应用程序对象依次来进行消息处理。
第二，如果文档边框窗口没有处理，调用 CFrameWnd::OnCmdMsg 把消息按标准路径发送，重复第一次的步骤，不过对于 MDI 边框窗口来说不存在活动视，所以省却了让视处理消息的必要；接着让 MDI 边框窗口本身来处理消息，如果它还没有处理，则让应用程序对象进行消息处理——虽然这是一个无用的重复。

除了 CView、CDocument 和 CMDIFrameWnd 类，还有几个 OLE 相关的类覆盖了 OnCmdMsg 函数。OLE 的处理本书暂不涉及，CDialog::OnCmdMsg 将在对话框章节专项讨论其具体实现。

4.4.3.4 一些消息处理类的OnCommand的实现

除了虚拟函数 OnCmdMsg，还有一个虚拟函数 OnCommand 在命令消息的发送中占有重要地位。在处理命令或者通知消息时，OnCommand 被 MFC 窗口过程调用，然后它调用 OnCmdMsg 按一定路径传送消息。除了 CWnd 类和一些 OLE 相关类外，MFC 里主要还有 MDI 边框窗口实现了 OnCommand。

```
BOOL CMDIFrameWnd::OnCommand(WPARAM wParam, LPARAM lParam)
```

第一，如果存在活动的文档边框窗口，则使用 AfxCallWndProc 调用它的窗口过程，把消息送给文档边框窗口来处理。这将导致文档边框窗口的 OnCmdMsg 作如下的处理：
活动视处理消息→与视关联的文档处理消息→本文档边框窗口处理消息→应用程序对象处理消息→文档边框窗口缺省处理
任何一个环节如果处理消息，则不再向下发送消息，处理终止。如果消息仍然没有被处理，就只有交给主边框窗口了。
第二，第一步没有处理命令，继续调用 *CFrameWnd::OnCommand*，将导致 CMDIFrameWnd 的 OnCmdMsg 被调用。从前面的分析知道，将再次把消息送给 MDI 边框

窗口的活动文档边框窗口，第一步的过程除了文档边框窗口缺省处理外都将被重复。具体的处理过程见前文的 CMDIFrameWnd::OnCmdMsg 函数。

第三， 对于 MDI 消息，如果主边框窗口还不处理的话，交给 CMDIFrameWnd 的 DefWindowProc 作缺省处理。

第四， 消息没有处理，返回 FALSE。

上述分析综合了 OnCommand 和 OnCmdMsg 的处理，它们是在 MFC 内部 MDI 边框窗口处理命令消息的完整的流程和标准的步骤。整个处理过程再次表明了边框窗口在处理命令消息时的中心作用。从程序员的角度来看，可以认为整个标准处理路径如下：

活动视处理消息→与视关联的文档处理消息→本文档边框窗口处理消息→应用程序对象处理消息→文档边框窗口缺省处理→MDI 边框窗口处理消息→MDI 边框窗口缺省处理

任何一个环节如果处理消息，不再向下发送消息，急处理终止。

4.4.4 对控制通知消息的接收和处理

4.4.4.1 WM_COMMAND控制通知消息的处理

WM_COMMAND 控制通知消息的处理和 WM_COMMAND 命令消息的处理类似，但是也有不同之处。

首先，分析处理 WM_COMMAND 控制通知消息和命令消息的相似处。如前所述，命令消息和控制通知消息都是由窗口过程给 OnCommand 处理（参见 CWnd::OnWndMsg 的实现），OnCommand 通过 wParam 和 lParam 参数区分是命令消息或通知消息，然后送给 OnCmdMsg 处理（参见 CWnd::OnCommnd 的实现）。

其次，两者的不同之处是：

- 命令消息一般是送给主边框窗口的，这时，边框窗口的 OnCmdMsg 被调用；而控制通知消息送给控制子窗口的父窗口，这时，父窗口的 OnCmdMsg 被调用。
- OnCmdMsg 处理命令消息时，通过命令分发可以由多种命令目标处理，包括非窗口对象如文档对象等；而处理控制通知消息时，不会有消息分发的过程，控制通知消息最终肯定是由窗口对象处理的。

1. 不过，在某种程度上可以说，控制通知消息由窗口对象处理是一种习惯和约定。当使用 ClassWizard 进行消息映射时，它不提供把控制通知消息映射到非窗口对象的机会。但是，手工地添加消息映射，让非窗口对象处理控制通知消息的可能是存在的。例如，对于 CFormView，一方面它具备接受 WM_COMMAND 通知消息的条件，另一方面，具备把 WM_COMMAND 消息派发给关联文档对象处理的能力，所以给 CFormView 的通知消息是可以让文档对象处理的。

2. 事实上，BN_CLICKED 控制通知消息的处理和命令消息的处理完全一样，因为该消息的通知代码是 0，ON_BN_CLICKED(id，memberfunction) 和 ON_COMMAND(id，memberfunction)是等同的。

3. 此外，MFC 的状态更新处理机制就是建立在通知消息可以发送给各种命令目标的基础之上的。关于 MFC 的状态更新处理机制，见后面 4.4.4.4 节的讨论。

- 控制通知消息可以反射给子窗口处理。OnCommand 判定当前消息是 WM_COMAND 通知消息之后，首先它把消息反射给控制子窗口处理，如果子窗口处理了反射消息，OnCommand 不会继续调用 OnCmdMsg 让父窗口对象来处理通知消息。

4.4.4.2 WM_NOTIFY消息及其处理:

（1）WM_NOTIFY 消息

还有一种通知消息 WM_NOTIFY，在 Win32 中用来传递信息复杂的通知消息。WM_NOTIFY 消息怎么来传递复杂的信息呢？WM_NOTIFY 的消息参数 wParam 包含了发送通知消息的控制窗口 ID，另一个参数 lParam 包含了一个指针。该指针指向一个 NMHDR 结构，或者更大的结构，只要它的第一个结构成员是 NMHDR 结构。

NMHDR 结构：

```
typedef struct tagNMHDR {
HWND hwndFrom;
UINT idFrom;
UINT code;
} NMHDR;
```

上述结构有三个成员，分别是发送通知消息的控制窗口的句柄、ID 和通知消息代码。

举一个更大、更复杂的结构例子：列表控制窗发送 LVN_KEYDOWN 控制通知消息，则 lParam 包含了一个指向 LV_KEYDOWN 结构的指针。其结构如下：

```
typedef struct tagLV_KEYDOWN {
NMHDR hdr;
WORD wVKey;
UINT flags;
}LV_KEYDOWN;
```

它的第一个结构成员 hdr 就是 NMHDR 类型。其他成员包含了更多的信息：哪个键被按下，哪些辅助键(SHIFT、CTRL、ALT 等)被按下。

（2）WM_NOTIFY 消息的处理

在分析 CWnd::OnWndMsg 函数时，曾指出当消息是 WM_NOTIFY 时，它把消息传递给 OnNotify 虚拟函数处理。这是一个虚拟函数，类似于 OnCommand，CWnd 和派生类都可以覆盖该函数。OnNotify 的函数原型如下：

```
BOOL CWnd::OnNotify(WPARAM, LPARAM lParam, LRESULT* pResult)
```

参数 1 是发送通知消息的控制窗口 ID，没有被使用；参数 2 是一个指针；参数 3 指向一个 long 类型的数据，用来返回处理结果。

WM_NOTIFY 消息的处理过程如下：

第一，反射消息给控制子窗口处理。

第二，如果子窗口不处理反射消息，则交给 OnCmdMsg 处理。给 OnCmdMsg 的四个参数分别如下：第一个是命令消息 ID，第四个为空；第二个高阶 word 是 WM_NOTIFY，低阶 word 是通知消息；第三个参数是指向 AFX_NOTIFY 结构的指针。第二、三个参数有别于 OnCommand 送给 OnCmdMsg 的参数。

AFX_NOTIFY 结构：

```
struct AFX_NOTIFY
{
LRESULT* pResult;
NMHDR* pNMHDR;
};
```

pNMHDR 的值来源于参数 2 lParam，该结构的域 pResult 用来保存处理结果，域 pNMHDR 用来传递信息。

OnCmdMsg 后续的处理和 WM_COMMAND 通知消息基本相同，只是在派发消息给消息处理函数时，DispatchMsdMsg 的第五个参数 pExtra 指向 OnCmdMsg 传递给它的 AFX_NOTIFY 类型的参数，而不是空指针。这样，处理函数就得到了复杂的通知消息信息。

4.4.4.3 消息反射

（1）消息反射的概念

前面讨论控制通知消息时，曾经多次提到了消息反射。MFC 提供了两种消息反射机制，一种用于 OLE 控件，一种用于 Windows 控制窗口。这里只讨论后一种消息反射。

Windows 控制常常发送通知消息给它们的父窗口，通常控制消息由父窗口处理。但是在 MFC 里头，父窗口在收到这些消息后，或者自己处理，或者反射这些消息给控制窗口自己处理，或者两者都进行处理。如果程序员在父窗口类覆盖了通知消息的处理（假定不调用基类的实现），消息将不会反射给控制子窗口。这种反射机制是 MFC 实现的，便于程序员创建可重用的控制窗口类。

MFC 的 CWnd 类处理以下控制通知消息时，必要或者可能的话，把它们反射给子窗口处理：

WM_CTLCOLOR，

WM_VSCROLL，WM_HSCROLL，

WM_DRAWITEM，WM_MEASUREITEM，

WM_COMPAREITEM，WM_DELETEITEM，

WM_CHARTOITEM，WM_VKEYTOITEM，

WM_COMMAND、WM_NOTIFY。

例如，对 WM_VSCROLL、WM_HSCROLL 消息的处理，其消息处理函数如下：

```
void CWnd::OnHScroll(UINT, UINT, CScrollBar* pScrollBar)
{
    //如果是一个滚动条控制，首先反射消息给它处理
    if (pScrollBar != NULL && pScrollBar->SendChildNotifyLastMsg())
        return;        //控制窗口成功处理了该消息

    Default();
}
```

又如：在讨论 OnCommand 和 OnNofity 函数处理通知消息时，都曾经指出，它们首先调用 ReflectLastMsg 把消息反射给控制窗口处理。

为了利用消息反射的功能，首先需要从适当的 MFC 窗口派生出一个控制窗口类，然后使用 ClassWizard 给它添加消息映射条目，指定它处理感兴趣的反射消息。下面，讨论反射消息映射宏。

上述消息的反射消息映射宏的命名遵循以下格式：“ON”前缀+消息名+“REFLECT”后缀，例如：消息 WM_VSCROLL 的反射消息映射宏是 ON_WM_VSCROLL_REFECT。但是通知消息 WM_COMMAND 和 WM_NOTIFY 是例外，分别为 ON_CONTROL_REFLECT 和 ON_NOFITY_REFLECT 。状态更新通知消息的反射消息映射宏是 ON_UPDATE_COMMAND_UI_REFLECT。

消息处理函数的名字和去掉“WM_”前缀的消息名相同 ，例如 WM_HSCROLL 反射消息处理函数是 Hscroll。

消息处理函数的原型这里不一一列举了。

这些消息映射宏和消息处理函数的原型可以借助于 ClassWizard 自动地添加到程序中。ClassWizard 添加消息处理函数时，可以处理的反射消息前面有一个等号，例如处理 WM_HSCROLL 的反射消息，选择映射消息“=EN_HSC ROLL”。ClassWizard 自动的添加消息映射宏和处理函数到框架文件。

（2）消息反射的处理过程

如果不考虑有 OLE 控件的情况，消息反射的处理流程如下图所示：

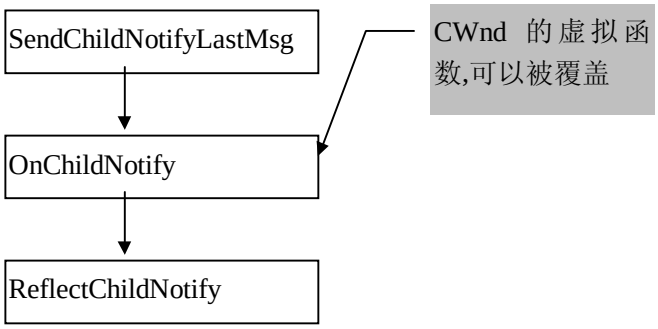

图 4-4　消息反射的处理流程

首先，调用 CWnd 的成员函数 SendChildNotifyLastMsg，它从线程状态得到本线程最近一次获取的消息（关于线程状态，后面第 9 章会详细介绍）和消息参数，并且把这些参数传递给函数 OnChildNotify。注意，当前的 CWnd 对象就是 MFC 控制子窗口对象。

OnChlidNofify 是 CWnd 定义的虚拟函数，不考虑 OLE 控制的话，它仅仅只调用 ReflectChildNotify。OnChlidNofify 可以被覆盖，所以如果程序员希望处理某个控制的通知消息，除了采用消息映射的方法处理通知反射消息以外，还可以覆盖 OnChlidNotify 虚拟函数，如果成功地处理了通知消息，则返回 TRUE。

ReflectChildNotify 是 CWnd 的成员函数，完成反射消息的派发。对于 WM_COMMAND，它直接调用 *CWnd::OnCmdMsg* 派发反射消息 WM_REFLECT_BASE+WM_COMMAND；对于 WM_NOTIFY，它直接调用 *CWnd::OnCmdMsg* 派发反射消息 WM_REFLECT_BASE+WM_NOFITY；对于其他消息，则直接调用 *CWnd::OnWndMsg*（即 *CmdTarge::OnWndMsg*）派发相应的反射消息，例如 WM_REFLECT_BASE+WM_HSCROLL。注意：ReflectChildNotify 直接调用了 CWnd 的 OnCmdMsg 或 OnWndMsg，这样反射消息被直接派发给控制子窗口，省却了消息发送的过程。

接着，控制子窗口如果处理了当前的反射消息，则返回反射消息被成员处理的信息。

（3）一个示例

如果要创建一个编辑框控制，要求它背景使用黄色，其他特性不变，则可以从 CEdit 派生一个类 CYellowEdit，处理通知消息 WM_CTLCOLOR 的反射消息。CYellowEdit 有三个属性，定义如下：

```
CYellowEdit::CYellowEdit()
{
m_clrText = RGB( 0, 0, 0 );
m_clrBkgnd = RGB( 255, 255, 0 );
m_brBkgnd.CreateSolidBrush( m_clrBkgnd );
}
```

使用 ClassWizard 添加反射消息处理函数：

函数原型：

```
afx_msg void HScroll();
```

消息映射宏：

```
ON_WM_CTLCOLOR_REFLECT()
```

函数的框架

```
HBRUSH CYellowEdit::CtlColor(CDC* pDC, UINT nCtlColor)
{
// TODO:添加代码改变设备描述表的属性
// TODO:  如果不再调用父窗口的处理，则返回一个非空的刷子句柄
return NULL;
}
```

添加一些处理到函数 CtlColor 中，如下：

```
pDC->SetTextColor( m_clrText );//设置文本颜色
pDC->SetBkColor( m_clrBkgnd );//设置背景颜色
return m_brBkgnd;                    //返回背景刷
```

这样，如果某个地方需要使用黄色背景的编辑框，则可以使用 CYellowEdit 控制。

4.4.5 对更新命令的接收和处理

用户接口对象如菜单、工具条有多种状态，例如：禁止，可用，选中，未选中，等等。这些状态随着运行条件的变化，由程序来进行更新。虽然程序员可以自己来完成更新，但是 MFC 框架为自动更新用户接口对象提供了一个方便的接口，使用它对程序员来说可能是一个好的选择。

4.4.5.1 实现方法

每一个用户接口对象，如菜单、工具条、控制窗口的子窗口，都由唯一的 ID 号标识，用户和它们交互时，产生相应 ID 号的命令消息。在 MFC 里，一个用户接口对象还可以响应 CN_UPDATE_COMMAND_UI 通知消息。因此，对每个标号 ID 的接口对象，可以有两个处理函数：一个消息处理函数用来处理该对象产生的命令消息 ID，另一个状态更新函数用来处理给该对象的 CN_UPDATE_COMMAND_UID 的通知消息。
使用 ClassWizard 可把状态更新函数加入到某个消息处理类，其结果是：
在类的定义中声明一个状态函数；
在消息映射中使用 ON_UPDATE_COMMAND_UI 宏添加一个映射条目；
在类的实现文件中实现状态更新函数的定义。

ON_UPDATE_COMMAND_UI 给指定 ID 的用户对象指定状态更新函数，例如：
ON_UPDATE_COMMAND_UI(ID_EDIT_COPY, OnUpdateEditCopy)
映射标识号 ID 为 ID_EDIT_COPY 菜单的通知消息 CN_UPDATE_COMMAND_UI 到函数 OnUpdateEditCopy。用于给 EDIT（编辑菜单）的菜单项 ID_EDIT_COPY（复制）添加一个状态处理函数 OnUpdateEditCopy，通过处理通知消息 CN_UPDATE_COMMAND_UI 实现该菜单项的状态更新。
状态处理函数的原型如下：
afxmsg void ClassName::OnUpdateEditPaste(CCmdUI* pCmdUI)
CCmdUI 对象由 MFC 自动地构造。在完善函数的实现时，使用 pCmdUI 对象和 CmdUI 的成员函数实现菜单项 ID_EDIT_COPY 的状态更新，让它变灰或者变亮，也就是禁止或者允许用户使用该菜单项。

4.4.5.2 状态更新命令消息

要讨论 MFC 的状态更新处理，先得了解一条特殊的消息。MFC 的消息映射机制除了处理各种 Windows 消息、控制通知消息、命令消息、反射消息外，还处理一种特别的“通知命令消息”，并通过它来更新菜单、工具栏（包括对话框工具栏）等命令目标的状态。
这种“通知命令消息”是 MFC 内部定义的，消息 ID 是 WM_COMMAND，通知代码是 CN_UPDATE_COMMAND_UI（0XFFFFFFFF）。
它不是一个真正意义上的通知消息，因为没有控制窗口产生这样的通知消息，而是 MFC 自己主动产生，用于送给工具条窗口或者主边框窗口，通知它们更新用户接口对象的状态。
它和标准 WM_COMMAND 命令消息也不相同，因为它有特定的通知代码，而命令消息通知代码是 0。
但是，从消息的处理角度，可以把它看作是一条通知消息。如果是工具条窗口接收该消息，

则在发送机制上它和 WM_COMMAND 控制通知消息是相同的，相当于让工具条窗口处理一条通知消息。如果是边框窗口接收该消息，则在消息的发送机制上它和 WM_COMMAND 命令消息是相同的，可以让任意命令目标处理该消息，也就是说边框窗口可以把该条通知消息发送给任意命令目标处理。
从程序员的角度，可以把它看作一条“状态更新命令消息”，像处理命令消息那样处理该消息。每条命令消息都可以对应有一条“状态更新命令消息”。ClassWizard 也支持让任意消息目标处理“状态更新命令消息”（包括非窗口命令目标），实现用户接口状态的更新。
在这条消息发送时，通过 OnCmdMsg 的第三个参数 pExtra 传递一些信息，表示要更新的用户接口对象。pExtra 指向一个 CCmdUI 对象。这些信息将传递给状态更新命令消息的处理函数。
下面讨论用于更新用户接口对象状态的类 CCmdUI。

4.4.5.3 类CCmdUI

CCmdUI 不是从 CObject 派生，没有基类。

（1） 成员变量

m_nID	用户接口对象的 ID
m_nIndex	用户接口对象的 index
m_pMenu	指向 CCmdUI 对象表示的菜单
m_pSubMenu	指向 CCmdUI 对象表示的子菜单
m_pOther	指向其他发送通知消息的窗口对象
m_pParentMenu	指向 CCmdUI 对象表示的子菜单

（2） 成员函数

Enable(BOOL bOn = TRUE) 禁止用户接口对象或者使之可用
SetCheck(int nCheck = 1) 标记用户接口对象选中或未选中
SetRadio(BOOL bOn = TRUE)
SetText(LPCTSTR lpszText)
ContinueRouting()
还有一个 MFC 内部使用的成员函数：
DoUpdate(CCmdTarget* pTarget, BOOL bDisableIfNoHndler)
其中，参数 1 指向处理接收更新通知的命令目标，一般是边框窗口；参数 2 指示如果没有提供处理函数（例如某个菜单没有对应的命令处理函数），是否禁止用户对象。
DoUpdate 作以下事情：
首先，发送状态更新命令消息给参数 1 表示的命令目标：调用 pTarget->OnCmdMsg（m_nID, CN_UPDATE_COMMAND_UI, this, NULL) 发送 m_nID 对象的通知消息 CN_UPDATE_COMMAND_UI。OnCmdMsg 的参数 3 取值 this，包含了当前要更新的用户接口对象的信息。
然后，如果参数 2 为 TRUE，调用 pTarget->OnCmdMsg(m_nID, CN_COMMAND, this, &info) 测试命令消息 m_nID 是否被处理。这时，OnCmdMsg 的第四个参数非空，表示仅仅是测试，不是真的要派发消息。如果没有提供命令消息 m_nID 的处理函数，则禁止用户对象 m_nID，否则使之可用。
从上面的讨论可以知道：通过其结构，一个 CCmdUI 对象标识它表示了哪一个用户接口对象，如果是菜单接口对象，pMenu 表示了要更新的菜单对象；如果是工具条，pOther 表示了要更新的工具条窗口对象，nID 表示了工具条按钮 ID。
所以，由参数上状态更新消息的消息处理函数就知道要更新什么接口对象的状态。例如，第 1 节的函数 OnUpdateEditPaste，函数参数 pCmdUI 表示一个菜单对象，需要更新该菜单对象的状态。
通过其成员函数，一个 CCmdUI 可以更新、改变用户接口对象的状态。例如，CCmdUI 可以管理菜单和对话框控制的状态，调用 Enable 禁止或者允许菜单或者控制子窗口，等等。
所以，函数 OnUpdateEditPaste 可以直接调用参数的成员函数（如 pCmdUI->Enable）实现菜

单对象的状态更新。
由于接口对象的多样性，其他接口对象将从 CCmdUI 派生出管理自己的类来，覆盖基类的有关成员函数如 Enable 等，提供对自身状态更新的功能。例如管理状态条和工具栏更新的 CStatusCmdUI 类和 CToolCmdUI 类。

4.4.5.4 自动更新用户接口对象状态的机制

MFC 提供了分别用于更新菜单和工具条的两种途径。

（1） 更新菜单状态

当用户对菜单如 File 单击鼠标时，就产生一条 WM_INITMENUPOPUP 消息，边框窗口在菜单下拉之前响应该消息，从而更新该菜单所有项的状态。
在应用程序开始运行时，边框也会收到 WM_INITMENUPOPUP 消息。

（2） 更新工具条等状态

当应用程序进入空闲处理状态时，将发送 WM_IDLEUPDATECMDUI 消息，导致所有的工具条用户对象的状态处理函数被调用，从而改变其状态。WM_IDLEUPDATECMDUI 是 MFC 自己定义和使用的消息。
在窗口初始化时，工具条也会收到 WM_IDLEUPDATECMDUI 消息。

（3） 菜单状态更新的实现

MFC 让边框窗口来响应 WM_INITMENUPOPUP 消息，消息处理函数是 OnInitMenuPopup，其原型如下：

```
afx_msg void CFrameWnd::OnInitMenuPopup( CMenu* pPopupMenu,
    UINT nIndex, BOOL bSysMenu );
```

第一个参数指向一个 CMenu 对象，是当前按击的菜单；第二个参数是菜单索引；第三个参数表示子菜单是否是系统控制菜单。
函数的处理：
如果是系统控制菜单，不作处理；否则，创建 CCmdUI 对象 state，给它的各个成员如 m_pMenu，m_pParentMenu，m_pOther 等赋值。
对该菜单的各个菜单项，调函数 state.DoUpdate，用 CCmdUI 的 DoUpdate 来更新状态。DoUpdate 的第一个参数是 this，表示命令目标是边框窗口；在 CFrameWnd 的成员变量 m_bAutoMenuEnable 为 TRUE 时（表示如果菜单 m_nID 没有对应的消息处理函数或状态更新函数，则禁止它），把 DoUpdate 的第二个参数 bDisableIfNoHndler 置为 TRUE。
顺便指出，m_bAutoMenuEnable 缺省时为 TRUE，所以，应用程序启动时菜单经过初始化处理，没有提供消息处理函数或状态更新函数的菜单项被禁止。

（4） 工具条等状态更新的实现

图 4-5 表示了消息空闲时 MFC 更新用户对象状态的流程：
MFC 提供的缺省空闲处理向顶层窗口(框架窗口)的所有子窗口发送消息 WM_IDLEUPDATECMDUI；MFC 的控制窗口（工具条、状态栏等）实现了对该消息的处理，导致用户对象状态处理函数的调用。

虽然两种途径调用了同一状态处理函数，但是传递的 CCmdUI 参数从内部构成上是不一样的：第一种传递的 CCmdUI 对象表示了一菜单对象，(pMenu 域被赋值)；第二种传递了一个窗口对象(pOther 域被赋值)。同样的状态改变动作，如禁止、允许状态的改变，前者调用了 CMenu 的成员函数 EnableMenuItem，后者使用了 CWnd 的成员函数 EnabelWindow。但是，这些不同由 CCmdUI 对象内部区分、处理，对用户是透明的：不论菜单还是对应的工具条，用户都用同一个状态处理函数使用同样的形式来处理。

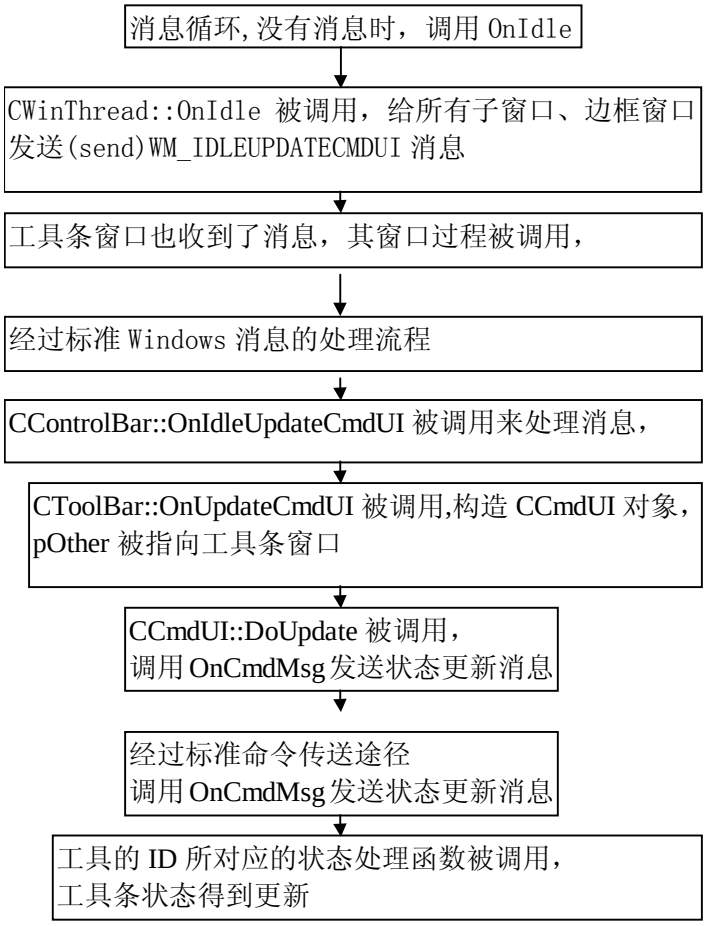

图 4-5　MFC 命令目标状态的更新

这一节分析了用户界面更新的原理和机制。在后面第 13 章讨论工具条和状态栏时，将详细的分析这种机制的具体实现。

4.5　消息的预处理

到现在为止，详细的讨论了 MFC 的消息映射机制。但是，为了提高效率和简化处理，MFC 提供了一种消息预处理机制，如果一条消息在预处理时被过滤掉了（被处理），则不会被派发给目的窗口的窗口过程，更不会进入消息循环了。

显然，能够进行预处理的消息只可能是队列消息，而且必须在消息派发之前进行预处理。因此，MFC 在实现消息循环时，对于得到的每一条消息，首先送给目的窗口、其父窗口、其祖父窗口乃至最顶层父窗口，依次进行预处理，如果没有被处理，则进行消息转换和消息派发，如果某个窗口实现了预处理，则终止。有关实现见后面关于 CWinThread 线程类的章节，CWinThread 的 Run 函数和 PreTranslateMessage 函数以及 CWnd 的函数 WalkPreTranslateTree 实现了上述要求和功能。这里要讨论的是 MFC 窗口类如何进行消息预处理。

CWnd 提供了虚拟函数 PreTranslateMessage 来进行消息预处理。CWnd 的派生类可以覆盖该函数，实现自己的预处理。下面，讨论几个典型的预处理。

首先，是 CWnd 的预处理：

预处理函数的原型为：

```
BOOL CWnd::PreTranslateMessage(MSG* pMsg)
```

CWnd 类主要是处理和过滤 Tooltips 消息。关于该函数的实现和 Tooltips 消息，见后面第 13 章关于工具栏的讨论。

然后，是 CFrameWnd 的预处理：
CFrameWnd 除了调用基类 CWnd 的实现过滤 Tooltips 消息之外，还要判断当前消息是否是键盘快捷键被按下，如果是，则调用函数::TranslateAccelerator(m_hWnd, hAccel, pMsg)处理快捷键。

接着，是 CMDIChildWnd 的预处理：
CMDIChildWnd 的预处理过程和 CFrameWnd 的一样，但是不能依靠基类 CFrameWnd 的实现，必须覆盖它。因为 MDI 子窗口没有菜单，所以它必须在 MDI 边框窗口的上下文中来处理快捷键，它调用了函数::TranslateAccelerator(GetMDIFrame()->m_hWnd, hAccel, pMsg)。

讨论了 MDI 子窗口的预处理后，还要讨论 MDI 边框窗口：
CMDIFrameWnd 的实现除了 CFrameWnd 的实现的功能外，它还要处理 MDI 快捷键（标准 MDI 界面统一使用的系统快捷键）。

在后面，还会讨论 CDialog、CFormView、CToolBar 等的消息预处理及其实现。
至于 CWnd::WalkPreTranslateTree 函数，它从接受消息的窗口开始，逐级向父窗回溯，逐一对各层窗口调用 PreTranslateMessage 函数，直到消息被处理或者到最顶层窗口为止。

4.6 MFC消息映射的回顾

从处理命令消息的过程可以看出，Windows 消息和控制消息的处理要比命令消息的处理简单，因为查找消息处理函数时，后者只要搜索当前窗口对象(this 所指)的类或其基类的消息映射入口表。但是，命令消息就要复杂多了，它沿一定的顺序链查找链上的各个命令目标，每一个被查找的命令目标都要搜索它的类或基类的消息映射入口表。
MFC 通过消息映射的手段，以一种类似 C++虚拟函数的概念向程序员提供了一种处理消息的方式。但是，若使用 C++虚拟函数实现众多的消息，将导致虚拟函数表极其庞大；而使用消息映射，则仅仅感兴趣的消息才加入映射表，这样就要节省资源、提高效率。这套消息映射机制的基础包括以下几个方面：

第一、消息映射入口表的实现：采用了 C++静态成员和虚拟函数的方法来表示和得到一个消息映射类(CCmdTarget 或派生类)的映射表。

第二、消息查找的实现：从低层到高层搜索消息映射入口表，直至根类 CCmdTarget。

第三、消息发送的实现：主要以几个虚拟函数为基础来实现标准 MFC 消息发送路径：OnComamnd、OnNotify、OnWndMsg 和 OnCmdMsg。、

OnWndMsg 是 CWnd 类或其派生类的成员函数，由窗口过程调用。它处理标准的 Windows 消息。
OnCommand 是 CWnd 类或其派生类的成员函数，由 OnWndMsg 调用来处理 WM_COMMAND 消息，实现命令消息或者控制通知消息的发送。如果派生类覆盖该函数，则必须调用基类的实现，否则将不能自动的处理命令消息映射，而且必须使用该函数接受的参数（不是程序员给定值）调用基类的 OnCommand。
OnNotify 是 CWnd 类或其派生类的成员函数，由 OnWndMsg 调用来处理 WM_NOTIFY 消息，实现控制通知消息的发送。
OnCmdMsg 是 CCmdTarget 类或其派生类的成员函数。被 OnCommand 调用，用来实现命令消息发送和派发命令消息到命令消息处理函数。

自动更新用户对象状态是通过 MFC 的命令消息发送机制实现的。
控制消息可以反射给控制窗口处理。
队列消息在发送给窗口过程之前可以进行消息预处理，如果消息被 MFC 窗口对象预处理了，则不会进入消息发送过程。