雨人 C++，MFC,ObjectARX学习笔记

个人资料

它装有他装明月照大江

访问： 12538次
积分： 195
等级： BLOG 2
排名： 千里之外

原创： 9篇 转载： 3篇
译文： 0篇 评论： 0条

MFC中CObject

标签： mfc command windows 文档 pascal class

2011-11-22 20:38 2112人阅读 评论(0) 收藏 举报

CObject类

CObject 是大多数MFC类的根类或基类。CObject类有很多有用的特性：对运行时类信息的支持，对动态创建的支持，对串行化的支持，对象诊断输出，等等。 MFC从CObject派生出许多类，具备其中的一个或者多个特性。程序员也可以从CObject类派生出自己的类，利用CObject类的这些特性。

本章将讨论MFC如何设计CObject类的这些特性。首先，考察CObject类的定义，分析其结构和方法（成员变量和成员函数）对CObject特性的支持。然后，讨论CObject特性及其实现机制。

CObject的结构

以下是CObject类的定义：

```
class CObject
{
public:
//与动态创建相关的函数
virtual CRuntimeClass* GetRuntimeClass() const;
析构函数
virtual ~CObject(); // virtual destructors are necessary
//与构造函数相关的内存分配函数，可以用于DEBUG下输出诊断信息
void* PASCAL operator new(size_t nSize);
void* PASCAL operator new(size_t, void* p);
void PASCAL operator delete(void* p);
#if defined(_DEBUG) && !defined(_AFX_NO_DEBUG_CRT)
void* PASCAL operator new(size_t nSize, LPCSTR lpszFileName, int nLine);
#endif
//缺省情况下，复制构造函数和赋值构造函数是不可用的
//如果程序员通过传值或者赋值来传递对象，将得到一个编译错误
protected:
//缺省构造函数
CObject();
private:
//复制构造函数，私有
CObject(const CObject& objectSrc); // no implementation
//赋值构造函数，私有
void operator=(const CObject& objectSrc); // no implementation
// Attributes
public:
//与运行时类信息、串行化相关的函数
BOOL IsSerializable() const;
BOOL IsKindOf(const CRuntimeClass* pClass) const;
// Overridables
```

CFileDialog的使用 (0)
关于对话框中static文本控 (0)
MFC进度条编程控制（转 (0)
MFC中CObject (0)
通过ObjectARX的运行时 (0)
MFC：Tab Control 控件((0)


```
virtual void Serialize(CArchive& ar);
// 诊断函数
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
// Implementation
public:
//与动态创建对象相关的函数
static const AFX_DATA CRuntimeClass classCObject;
#ifdef _AFXDLL
static CRuntimeClass* PASCAL _GetBaseClass();
#endif
};
```

由上可以看出，CObject定义了一个CRuntimeClass类型的静态成员变量：

CRuntimeClass classCObject

还定义了几组函数：

构造函数析构函数类，

诊断函数，

与运行时类信息相关的函数，

与串行化相关的函数。

其 中，一个静态函数：_GetBaseClass；五个虚拟函数：析构函数、GetRuntimeClass、Serialize、AssertValid、Dump。这些虚拟函数，在CObject的派生类中应该有更具体的实现。必要的话，派生类实现它们时可能要求先调用基类的实 现，例如Serialize和Dump就要求这样。

静态成员变量classCObject和相关函数实现了对CObjet特性的支持。

CObject类的特性

下面，对三种特性分别描述，并说明程序员在派生类中支持这些特性的方法。

对运行时类信息的支持

该特性用于在运行时确定一个对象是否属于一特定类（是该类的实例），或者从一个特定类派生来的。CObject提供IsKindOf函数来实现这个功能。

从CObject派生的类要具有这样的特性，需要：

定义该类时，在类说明中使用DECLARE_DYNAMIC（CLASSNMAE）宏；

在类的实现文件中使用IMPLEMENT_DYNAMIC(CLASSNAME，BASECLASS)宏。

对动态创建的支持

前面提到了动态创建的概念，就是运行时创建指定类的实例。在MFC中大量使用，如前所述框架窗口对象、视对象，还有文档对象都需要由文档模板类(CDocTemplate)对象来动态的创建。

从CObject派生的类要具有动态创建的功能，需要：

定义该类时，在类说明中使用DECLARE_DYNCREATE（CLASSNMAE）宏；

定义一个不带参数的构造函数（默认构造函数）；

在类的实现文件中使用IMPLEMENT_DYNCREATE（CLASSNAME，BASECLASS）宏；

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。

例如：

```
CRuntimeClass* pRuntimeClass = RUNTIME_CLASS(CNname)
//CName必须有一个缺省构造函数
CObject* pObject = pRuntimeClass->CreateObject();
//用IsKindOf检测是否是CName类的实例
Assert( pObject->IsKindOf(RUNTIME_CLASS(CName));
```

对序列化的支持

“序列化”就是把对象内容存入一个文件或从一个文件中读取对象内容的过程。从CObject派生的类要具有序列化的功能，需要：

定义该类时，在类说明中使用DECLARE_SERIAL（CLASSNMAE）宏；

定义一个不带参数的构造函数（默认构造函数）；
在类的实现文件中使用IMPLEMENT_SERIAL（CLASSNAME，BASECLASS）宏；
覆盖Serialize成员函数。（如果直接调用Serialize函数进行序列化读写，可以省略前面三步。）
对运行时类信息的支持、动态创建的支持、串行化的支持层（不包括直接调用Serailize实现序列化），这三种 功能的层次依次升高。如果对后面的功能支持，必定对前面的功能支持。支持动态创建的话，必定支持运行时类信息；支持序列化，必定支持前面的两个功能，因为 它们的声明和实现都是后者包含前者。

综合示例：

定义一个支持串行化的类CPerson：

```
class CPerson : public CObject
{

    public:

DECLARE_SERIAL( CPerson )
// 缺省构造函数
CPerson(){}{};
CString m_name;
WORD m_number;
void Serialize( CArchive& archive );
// rest of class declaration
};
```

实现该类的成员函数Serialize，覆盖CObject的该函数：

```
void CPerson::Serialize( CArchive& archive )
{
// 先调用基类函数的实现
CObject::Serialize( archive );
// now do the stuff for our specific class
if( archive.IsStoring() )
archive << m_name << m_number;
else
archive >> m_name >> m_number;
}
```

使用运行时类信息：

```
CPerson a;
ASSERT( a.IsKindOf( RUNTIME_CLASS( CPerson ) ) );
ASSERT( a.IsKindOf( RUNTIME_CLASS( CObject ) ) );
```

动态创建：

```
CRuntimeClass* pRuntimeClass = RUNTIME_CLASS(CPerson)
//Cperson有一个缺省构造函数
CObject* pObject = pRuntimeClass->CreateObject();
Assert( pObject->IsKindOf(RUNTIME_CLASS(CPerson));
```

实现**CObject**特性的机制

由上，清楚了CObject的结构，也清楚了从CObject派生新类时程序员使用CObject特性的方法。现在来考察这些方法如何利用CObjet的结构，CObject结构如何支持这些方法。

首先，要揭示DECLARE_DYNAMIC等宏的内容，然后，分析这些宏的作用。

DECLARE_DYNAMIC等宏的定义

MFC提供了DECLARE_DYNAMIC、DECLARE_DYNCREATE、DECLARE_SERIAL声明宏的两种定义，分别用于静态链接到MFC DLL和动态链接到MFC DLL。对应的实现宏IMPLEMNET_XXXX也有两种定义，但是，这里实现宏就不列举了。

MFC对这些宏的定义如下：

```
#ifdef _AFXDLL //动态链接到MFC DLL
#define DECLARE_DYNAMIC(class_name) \
protected: \
```

```
static CRuntimeClass* PASCAL _GetBaseClass(); \
public: \
static const AFX_DATA CRuntimeClass class##class_name; \
virtual CRuntimeClass* GetRuntimeClass() const; \
#define _DECLARE_DYNAMIC(class_name) \
protected: \
static CRuntimeClass* PASCAL _GetBaseClass(); \
public: \
static AFX_DATA CRuntimeClass class##class_name; \
virtual CRuntimeClass* GetRuntimeClass() const; \
#else
#define DECLARE_DYNAMIC(class_name) \
public: \
static const AFX_DATA CRuntimeClass class##class_name; \
virtual CRuntimeClass* GetRuntimeClass() const; \
#define _DECLARE_DYNAMIC(class_name) \
public: \
static AFX_DATA CRuntimeClass class##class_name; \
virtual CRuntimeClass* GetRuntimeClass() const; \
#endif
// not serializable, but dynamically constructable
#define DECLARE_DYNCREATE(class_name) \
DECLARE_DYNAMIC(class_name) \
static CObject* PASCAL CreateObject();
#define DECLARE_SERIAL(class_name) \
_DECLARE_DYNCREATE(class_name) \
friend CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pOb);
```

由于这些声明宏都是在CObect派生类的定义中被使用的，所以从这些宏的上述定义中可以看出，DECLARE_DYNAMIC宏给所在类添加了一个CRuntimeClass类型的静态数据成员class##class_name(类名加前缀class，例如，若类名是CPerson，则该变量名称是classCPerson)，且指定为const；两个（使用MFC DLL时，否则，一个）成员函数：虚拟函数GetRuntimeClass和静态函数_GetBaseClass（使用MFC DLL时）。

DECLARE_DYNCREATE宏包含了DECLARE_DYNAMIC，在此基础上，还定义了一个静态成员函数CreateObject。

DECLARE_SERIAL宏则包含了_DECLARE_DYNCREATE，并重载了操作符“>>”（友员函数）。它和前两个宏有所不同的是CRuntimeClass数据成员class##class_name没有被指定为const。

对应地，MFC使用三个宏初始化DECLARE宏所定义的静态变量并实现DECLARE宏所声明的函数：IMPLEMNET_DYNAMIC，IMPLEMNET_DYNCREATE，IMPLEMENT_SERIAL。

首先，这三个宏初始化CRuntimeClass类型的静态成员变量class#class_name。IMPLEMENT_SERIAL不同于其他两个宏，没有指定该变量为const。初始化内容在下节讨论CRuntimeClass时给出。

其次，它实现了DECLARE宏声明的成员函数：

_GetBaseClass()

返回基类的运行时类信息，即基类的CRuntimeClass类型的静态成员变量。这是静态成员函数。

GetRuntimeClass()

返回类自己的运行类信息，即其CRuntimeClass类型的静态成员变量。这是虚拟成员函数。

对于动态创建宏，还有一个静态成员函数CreateObject，它使用C++操作符和类的缺省构造函数创建本类的一个动态对象。

操作符的重载

对于序列化的实现宏IMPLEMENT_SERIAL，还重载了操作符<<和定义了一个静态成员变量

```
static const AFX_CLASSINIT _init_##class_name(RUNTIME_CLASS(class_name));
```

比如，对CPerson来说，该变量是_init_Cperson，其目的在于静态成员在应用程序启动之前被初始化，使得AFX_CLASSINIT类的构造函数被调用，从而通过AFX_CLASSINIT类的构造函数在模块状态的CRuntimeClass链表中插入构造函数参数表示的CRuntimeClass类信息。至于模块状态，在后文有详细的讨论。

重载的操作符函数用来在序列化时从文档中读入该类对象的内容，是一个友员函数。定义如下：

```
CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pOb)
```

```
{
pOb = (class_name*) ar.ReadObject(
RUNTIME_CLASS(class_name));
return ar;
}
```

回顾CObject的定义，它也有一个CRuntimeClass类型的静态成员变量classCObject，因为它本身也支持三个特性。以CObject及其派生类的静态成员变量classCObject为基础，IsKindOf和动态创建等函数才可以起到作用。

这个变量为什么能有这样的用处，这就要分析CRuntimeClass类型变量的结构和内容了。下面，在讨论了CRuntimeClass的结构之后，考察该类型的静态变量被不同的宏初始化之后的内容。

CruntimeClass类的结构与功能

从上面的讨论可以看出，在对CObject特性的支持上，CRuntimeClass类起到了关键作用。下面，考查它的结构和功能。

CRuntimeClass的结构

CruntimeClass的结构如下：

```
Struct CRuntimeClass
{
LPCSTR m_lpszClassName;//类的名字
int m_nObjectSize;//类的大小
UINT m_wSchema;
CObject* (PASCAL* m_pfnCreateObject)();
//pointer to function, equal to newclass.CreateObject()
//after IMPLEMENT
CRuntimeClass* (PASCAL* m_pfnGetBaseClass)();
CRumtieClass* m_pBaseClass;
//operator:
CObject *CreateObject();
BOOL IsDerivedFrom(const CRuntimeClass* pBaseClass) const;
...
}
```

CRuntimeClass成员变量中有两个是函数指针，还有几个用来保存所在CruntimeClass对象所在类的名字、类的大小（字节数）等。

这些成员变量被三个实现宏初始化，例如：

m_pfnCreateObject，将被初始化指向所在类的静态成员函数CreateObject。CreateObject函数在初始化时由实现宏定义，见上文的说明。

m_pfnGetBaseClass，如果定义了_AFXDLL，则该变量将被初始化指向所在类的成员函数_GetBaseClass。_GetBaseClass在声明宏中声明，在初始化时由实现宏定义，见上文的说明。

下面，分析三个宏对CObject及其派生类的CRuntimeClass类型的成员变量class##class_name初始化的情况，然后讨论CRuntimeClass成员函数的实现。

成员变量class##class_name的内容

IMPLEMENT_DYNCREATE等宏将初始化类的CRuntimeClass类型静态成员变量的各个域，表3-1列出了在动态类信息、动态创建、序列化这三个不同层次下对该静态成员变量的初始化情况：

表3-1 静态成员变量class##class_name的初始化

CRuntimeClass成员变量	动态类信息	动态创建	序列化
m_lpszClassName	类名字符串	类名字符串	类名字符串
m_nObjectSize	类的大小（字节数）	类的大小（字节数）	类的大小（字节数）
m_wShema	0xFFFF	0xFFFF	1、2等，非0
m_pfnCreateObject	NULL	类的成员函数CreateObject	类的成员函数CreateObject

m_pBaseClass	基类的CRuntimeClass变量	基类的CRuntimeClass变量	基类的CRuntimeClass变量
m_pfnGetBaseClass	类的成员函数_GetBaseClass	类的成员函数_GetBaseClass	类的成员函数_GetBaseClass
m_pNextClass	NULL	NULL	NULL

m_wSchema类型是UINT，定义了序列化中保存对象到文档的程序的版本。如果不要求支持序列化特性，该域为0XFFFF，否则，不能为0。

Cobject类本身的静态成员变量classCObject被初始化为：

{ "CObject", sizeof(CObject), 0xffff, NULL, &CObject::_GetBaseClass, NULL };

对初始化内容解释如下：

类名字符串是“CObject”，类的大小是sizeof(CObject)，不要求支持序列化，不支持动态创建。

成员函数CreateObject

回 顾3.2节，动态创建对象是通过语句pRuntimeClass->CreateObject完成的，即调用了CRuntimeClass自己的成 员函数，CreateObject函数又调用m_pfnCreateObject指向的函数来完成动态创建任务，如下所示：

```
CObject* CRuntimeClass::CreateObject()
{
if (m_pfnCreateObject == NULL) //判断函数指针是否空
{
TRACE(_T("Error: Trying to create object which is not ")
_T("DECLARE_DYNCREATE \nor DECLARE_SERIAL: %hs.\n"),
m_lpszClassName);
return NULL;
}
//函数指针非空，继续处理
CObject* pObject = NULL;
TRY
{
pObject = (*m_pfnCreateObject)(); //动态创建对象
}
END_TRY
return pObject;
}
```

成员函数IsDerivedFrom

该函数用来帮助运行时判定一个类是否派生于另一个类，被CObject的成员函数IsKindOf函数所调用。其实现描述如下：

如 果定义了_AFXDLL则，成员函数IsDerivedFrom调用成员函数m_pfnGetBaseClass指向的函数来向上逐层得到基类的 CRuntimeClass类型的静态成员变量，直到某个基类的CRuntimeClass类型的静态成员变量和参数指定的CRuntimeClass变 量一致或者追寻到最上层为止。

如果没有定义_AFXDLL，则使用成员变量m_pBaseClass基类的CRuntimeClass类型的静态成员变量。

程序如下所示：

```
BOOL CRuntimeClass::IsDerivedFrom(
const CRuntimeClass* pBaseClass) const
{
ASSERT(this != NULL);
ASSERT(AfxIsValidAddress(this, sizeof(CRuntimeClass), FALSE));
ASSERT(pBaseClass != NULL);
ASSERT(AfxIsValidAddress(pBaseClass, sizeof(CRuntimeClass), FALSE));
// simple SI case
const CRuntimeClass* pClassThis = this;
while (pClassThis != NULL)//从本类开始向上逐个基类搜索
{
```

```
if (pClassThis == pBaseClass)//若是参数指定的类信息
return TRUE;
//类信息不符合，继续向基类搜索
#ifdef _AFXDLL
pClassThis = (*pClassThis->m_pfnGetBaseClass)();
#else
pClassThis = pClassThis->m_pBaseClass;
#endif
}
return FALSE; // 搜索完毕，没有匹配，返回FALSE。
}
```

由于CRuntimeClass类型的成员变量是静态成员变量，所以如果两个类的CruntimeClass成员变量相同，必定是同一个类。这就是IsDerivedFrom和IsKindOf的实现基础。

RUNTIME_CLASS宏

RUNTIME_CLASS宏定义如下：

```
#define RUNTIME_CLASS(class_name) (&class_name::class##class_name)
```

为了方便地得到每个类（Cobject或其派生类）的CRuntimeClass类型的静态成员变量，MFC定义 了这个宏。它返回对类class_name的CRuntimeClass类型成员变量的引用，该成员变量的名称是“class”加上class_name （类的名字）。例如：

RUNTIME_CLASS(CObject)得到对classCObject的引用；

RUNTIME_CLASS(CPerson)得到对class CPerson的引用。

动态类信息、动态创建的原理

MFC对Cobject动态类信息、动态创建的实现原理：

动态类信息、动态创建都建立在给类添加的CRuntimeClass类型的静态成员变量基础上，总结如下。

C++不支持动态创建，但是支持动态对象的创建。动态创建归根到底是创建动态对象，因为从一个类名创建一个该类的实例最终是创建一个以该类为类型的动态对象。其中的关键是从一个类名可以得到创建其动态对象的代码。

在一个类没有任何实例之前，怎么可以得到该类的创建动态对象的代码？借助于C++的静态成员数据技术可达到这个目的：

静态成员数据在程序的入口(main或WinMain)之前初始化。因此，在一个静态成员数据里存放有关类型信息、动态创建函数等，需要的时候，得到这个成员数据就可以了。

不论一个类创建多少实例，静态成员数据只有一份。所有的类的实例共享一个静态成员数据，要判断一个类是否是一个类的实例，只须确认它是否使用了该类的这个静态数据。

从前两节的讨论知道，DECLARE_CREATE等宏定义了一个这样的静态成员变量：类型是CRuntimeClass，命名约定是“calss”加上类名；IMPLEMENT_CREATE等宏初始化该变量；RUNTIME_CLASS宏用来得到该成员变量。

动态类信息的原理在分析CRuntimeClass的成员函数IsDerivedFrom时已经作了解释。

动态创建的过程和原理了，用图表示其过程如下：

注：下面一个方框内的逐级缩进表示逐层调用关系。

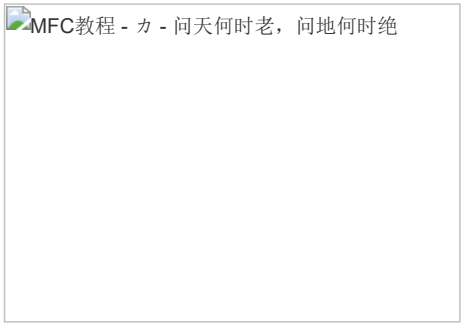
序列化的机制

由上所述可知，一个类要支持实现序列化，使得它的对象可以保存到文档中或者可以从文档中读入到内存中并生成对象，需要使用动态类信息，而且，需要覆盖基类的Serialize虚拟函数来完成其对象的序列化。

仅仅有类的支持是不够的，MFC还提供了一个归档类CArchive来支持简单类型的数据和复杂对象的i

CArchive 在文件和内存对象之间充当一个代理者的角色。它负责按一定的顺序和格式把内存对象写到文件中，或者读出来，可以被看作是一个二进制的流。它和文件类CFile的关系如图3-2所示：

一个CArchive对象在要序列化的对象和存储媒体(storage medium，可以是一个文件或者一个Socke 卜作用。它提供了系列方法来完成序列化，不仅能够把int、float等简单类型数据进行 序列化，而且能够把复杂的数据如string等进行序列化，更重要的是它能把复杂的对象(包括复合对象)进行序列化。这些方法就是重载的操作符> >和<<。对于简单类型，它针对不同类型直接实现不同的读写操作；对于复杂的对象，其每一个支持序列化的类都重载



了操作符> >，从前几节可以清楚地看到这点：IMPLEMENT_SERIAL给所在类重载了操作符>>。至于<<操作，就不必每个 序列化类都重载了。

复杂对象的“<<”操作，先搜索本模块状态的CRuntimeClass链表看是否有“< <”第二个参数指定的对象类的运行类信息(搜索过程涉及到模块状态，将在9.5.2节描述)，如果有（无，则返回），则先使用这些信息动态的创建对 象（这就是是序列化类必须提供动态类信息、支持动态创建的原因），然后对该对象调用Serilize函数从存储媒体读入对象内容。

复杂对象的“>>”操作先把对象类的运行类信息写入存储媒体，然后对该对象调用Serilize函数把对象内容写入存储媒体。

在创建CArchive对象时，必须有一个CFile对象，它代表了存储媒介。通常，程序员不必做这个工作，打开 或保存文档时MFC将自动的创建CFile对象和CArchive对象并在适当的时候调用序列化类的Serialize函数。在后面讨论打开(5.3.3.2节)或者关闭(6.1节)文档时将会看到这样的流程。

CArchive对象被创建时，需要指定它是用来读还是用来写，即指定序列化操作的方向。Serialize函数适用CArchive的函数IsStoring判定CArchive是用于读出数据还是写入数据。

在解释实现序列化的方法时，曾经提到如果程序员直接调用Serilize函数完成序列化，而不借助 CArchive的>>和<<操作，则可以不需要动态类信息和动态创建。从上文的论述可以看出，没有CArchive的> >和<<操作，的确不需要动态类信息和动态创建特性。

消息映射的实现

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。

消息的分类

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。

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。

消息结构和消息处理

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 人列中移走）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。

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。

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。

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 与幕，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。

消息映射的定义和实现

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。

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。

在声明与实现的内部

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。
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。
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\

};
```


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。

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）。

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。

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。

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。
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。
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。

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。

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。MF 肖
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 员
nMessage指定为要处理的Windows消息的ID，第二个成员nCode指定为0。
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。

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。

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。

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。

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。

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。

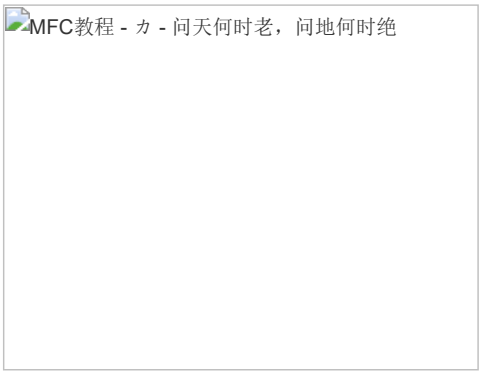


图4-1中的CREATESTRUCT结构类型的变量cs包含了传递给窗口过程的初始化参数。

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GetSuperWndProcA

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[illegible]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。

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。

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。 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。

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

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

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。

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。
消息映射机制完成虚拟函数功能的原理

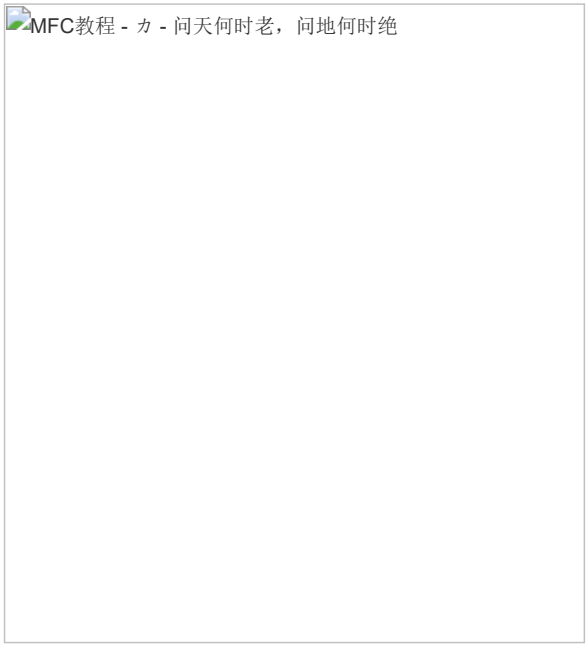
综合对Windows消息的处理来看，MFC使用消息映射机制完成了C++虚拟函数的功能。这主要基于以下几点：
所有处理消息的类从CCmdTarget派生。
使用静态成员变量_messageEntries数组存放消息映射条目，使用静态成员变量messageMap来唯一地区别和得到类的消息映射。
通过GetMessage虚拟函数来获取当前对象的类的messageMap变量，进而得到消息映射入口。
按照先底层，后基层的顺序在类的消息映射数组中搜索消息处理函数。基于这样的机制，一般在覆盖基类的消息处理函数时，应该调用基类的同名函数。
以上论断适合于MFC其他消息处理机制，如对命令消息的处理等。不同的是其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。

对命令消息的接收和处理

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。
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。接着介绍的内部的有关调用，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：

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。

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。

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。

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。

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。

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 处理函数

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，函数逐级都返回，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。

命令消息的派发和消息的多次处理

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。

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 且的多条消息的命令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个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。

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)

调用*CWnd::OnCmdMsg*，让对话框或其基类处理消息。
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。
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。
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。M 对
OnCmdMsg的实现函数的原型如下：
BOOL CMDIFrameWnd::OnCmdMsg(UINT nID, int nCode, void* pExtra,

AFX_CMDHANDLERINFO* pHandlerInfo)

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。
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。

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。

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。
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。

对控制通知消息的接收和处理

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。
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 和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 处理的。
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。

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。

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：

反射消息给控制子窗口处理。

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，而这样，处理函数就 得到了复杂的通知消息信息。

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。

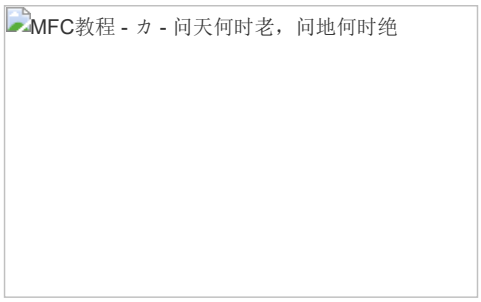
消息处理函数的名字和去掉“WM_”前缀的消息名相同 ，例如WM_HSCROLL反射消息处理函数是Hscroll。

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_HSCROLL”。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：



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_COMMAN[调用 *CWnd::OnCmdMsg* 派发反射消息WM_REFLECT_BASE+WM_COMMAND；对于WM_NOTI 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 II 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。

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。

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对象的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。

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。

类**CCmdUI**

CCmdUI不是从CObject派生，没有基类。

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

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[illegible]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。

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。

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。

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。

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。

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。

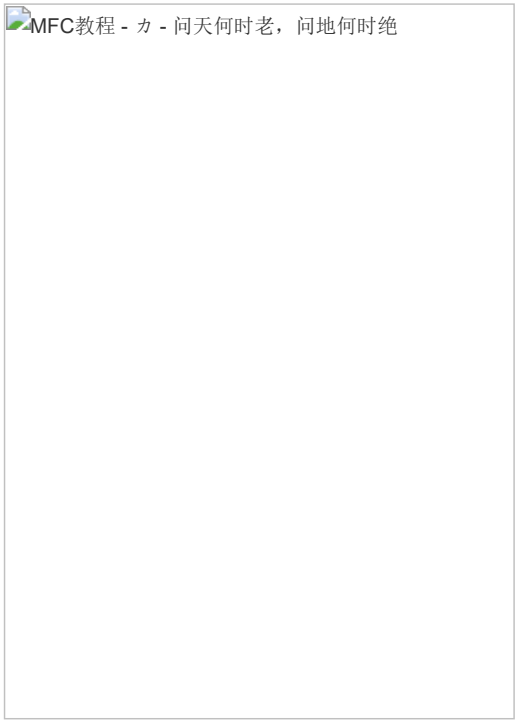
这一节分析了用户界面更新的原理和机制。在后面第13章讨论工具条和状态栏时，将详细的分析这种机制的具体实现。

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，I 中消息预处理机制，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 C

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。

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：

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。

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。

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。

顶 0 踩 0

上一篇 通过ObjectARX的运行时来复制实体
下一篇 MFC进度条编程控制（转）

参考知识库

软件测试知识库
4196 关注 | 310 收录

算法与数据结构知识库
14910 关注 | 2320 收录

猜你在找

windows命令行教程
Windows Server 2012 RMS 文档安全管理
Windows Server 2012 AD RMS 文档安全管理
windows批处理教程
Windows Server 2012 R2 远程桌面服务
转使用stdvector作为管理动态数组的优先选择
使用stdvector作为管理动态数组的优先选择
使用stdvector作为管理动态数组的优先选择
使用stdvector作为管理动态数组的优先选择
MFC架构之CObject类

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC
coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap