第一部分　基　　础

无论读者是否已经读过本系列的书籍，或者已经具备了多年的编程经验，我们仍将在这一部分回顾一下所需要的基本知识，其目的就在于能够使读者更好地理解本书的实例。编写程序常常是一种需要尝试不同方法以达到最终目的的工作。通常情况下，了解用 MFC来做什么涉及到对4个基本概念的理解：Windows API怎样创建窗口；MFC如何封装并改进Windows API；MFC如何与窗口通信以及MFC是怎样控制绘图任务的。除了这些概念以外，本部分还将讨论一下工具栏和状态栏。最后我们将讨论一下 MFC如何同非Windows构件进行通信，如串行口和Internet站点。

本部分包括的章节介绍如下。

第1章　概述

本章概述MFC如何封装并改进Windows API。如果读者已经阅读过本系列书籍的前一本，将会发现该章是对那些版本基础部分的一些回顾。本书包含这一章是为了保持本书对高层次读者的独立性。

第2章　控件条

本章将讨论MFC支持的控件条。标准的控件条包括工具栏、状态栏和伸缩条 (Rebar)等。MFC增加了对话条和停靠栏。该章还要探讨 MFC如何避免控件条之间以及它们和视之间互相覆盖的技术内幕。

第3章　通信

本章讨论应用程序与外部世界的不同通信方式。其中最基础的窗口消息将在第一章中讨论。本章还涉及其他一些通信途径，包括局域网、 Internet通信、串行和并行端口、 DDE、Windows挂钩和管道等。

第 1 章　概　　述

本章将回顾Windows应用程序的基本知识，包括应用程序如何创建窗口、窗口之间如何进行对话以及如何在窗口内绘图。然后将讨论微软基础类库 (MFC)以及Developer Studio怎样使创建窗口应用程序的工作变得容易起来。

1.1　Windows基础

当Windows操作系统启动应用程序时，它首先创建一个程序线程，该线程一般只是一个

可执行内存的管理模块，而这些内存则与系统中其他应用程序分享执行时间。如果这个应用程序要通过显示屏幕与用户交互，那么这个程序线程便需要负责创建显示在屏幕上的窗口。

程序线程通过调用操作系统的应用程序编程接口 (API)来创建这些窗口。实际调用的函数是::CreateWindowEx()，这个函数需要下列参数：屏幕位置、窗口大小以及即将创建的窗口的风格。

1.1.1 窗口类结构

线程创建的多数窗口具有类似的风格 (例如按钮)，这些类似的风格已经被集成为一个名为窗口类(Windows Class)的结构。注意这是一个结构，而不是一个 C++类。在创建窗口时必须设定窗口类。也可以使用其他的窗口风格，并且分别设定各自的窗口类结构。

1.1.2 消息

如果用户单击了一个窗口，操作系统就会向这个窗口发送一个消息来把这一事件通知给它。每个窗口用自己的窗口处理过程来处理窗口消息，举个例子，一个按钮的窗口处理过程可能向它的主窗口发送一个消息告诉它需要做什么事情。

每个窗口的处理过程还负责在屏幕上绘制属于自己的窗口。操作系统在绘制窗口时会向目标窗口发送WM_PAINT消息。

所有类似的窗口具备同样的窗口处理过程，例如，所有的按钮控件使用同样的窗口处理过程，因此所有的按钮看起来外表都很类似，其行为也类似。这种情况下的窗口处理过程位于操作系统内。它的地址在窗口的窗口类结构内指定。所有的按钮控件都由同样的窗口类创建，这个窗口类结构的名字叫做 BUTTON。

1.1.3 客户区和非客户区

窗口处理过程在屏幕上绘制一个窗口时实际上绘制了两个部分：客户区和非客户区。为了绘制非客户区(nonclient area)，窗口处理过程总是调用所有窗口过程都需要调用的相同的操作系统处理过程。该过程接下来还需要绘制框架、菜单条以及标题栏等等多数窗口共同具有的内容。过程所绘制的东西取决于窗口的风格。例如，由于按钮的风格被指定为不用绘制其非客户区，所以按钮窗口上就不会看见框架和菜单。

窗口的客户区(client area)总是由窗口自己的窗口处理过程绘制，也可能由操作系统来完成这件事，例如所有的按钮都由同样的处理过程来绘制，或者由创建者自己来绘制图像或列表。

1.1.4 重叠窗口、弹出窗口和子窗口

除了窗口类以外，还有成百上千种窗口风格供用户指定窗口的绘制及其行为。其中有 3种最重要的风格创建了对应3种最基本的窗口类型：重叠窗口、弹出窗口和子窗口。

■ 重叠窗口(overlapped window)，具有应用程序主窗口的全部特点。它的非客户区包括一个可伸缩的框架、菜单条、标题栏和最小化、最大化按钮。

■ 弹出窗口(popup window)，具有消息框或者对话框的全部特点。它的非客户区包括一个固定大小的框架和一个标题栏。

■ 子窗口(child window)，具有类似按钮控件的全部特点。它没有非客户区，窗口的处理过程负责绘制窗口的每个部分。

这些窗口在其行为上表现不同，这将在以后讨论。

1.1.5 父窗口和宿主窗口

由于用户界面可能会由好多个窗口组成，所以由程序线程控制它们将是很困难的，例如，如果用户将一个应用程序最小化，那么程序线程应该对那些组成用户界面的所有最小化窗口负责吗？实际上并没有采取直接控制的方式，应用程序创建的每个窗口都通过调用::CreateWindowEx()分配了一个控制窗口。如果这个控制窗口被最小化，那么所有被其控制的窗口都会由操作系统自动地最小化。如果控制窗口被销毁，那么每个被控制窗口也随之被销毁。

每个被控窗口也可以作为其他窗口的控制窗口，结果最小化或者销毁某些窗口都只影响用户界面的一部分。无论是什么窗口或者位于何处，程序员都可以在它内部创建另外的窗口。

子窗口的控制窗口叫做父窗口 (parent window)。父窗口剪切其子窗口，也就是子窗口不能在其父窗口以外绘制。当用户与类似按钮的子窗口交互的时候，这些子窗口将自动地生成消息并发送到其父窗口。这就使得控件可以在父窗口的窗口处理过程中被集中处理。

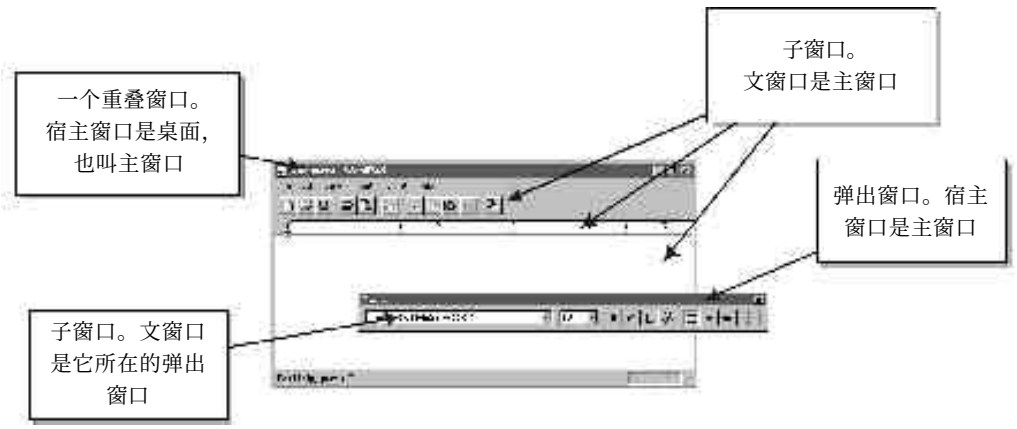

图1-1 构成一个Windows应用程序界面的窗口

弹出窗口和重叠窗口的控制窗口叫做宿主窗口 (owner window)。与父窗口不同，宿主窗口并不限制属于它的窗口。然而，当最小化宿主窗口的时候，它的所属窗口也将被最小化。但是，当宿主窗口隐藏的时候，它的所属窗口却仍然显示。请参见图 1-1以了解组成Windows应用程序的各种窗口。

1.2 Windows消息

上面提到，每个窗口由其自己的窗口处理过程响应来自操作系统或者其他窗口的消息。例如，用户用鼠标单击了某个窗口，那么操作系统可能就会向这个窗口发送一个消息。如果这是一个标识为Load File的按钮窗口，那么它的窗口处理过程就可能向应用的主窗口发送一个消息通知加载文件。主窗口处理过程将加载文件并在其客户区显示文件内容来作为响应。

接下来讨论消息是如何传送的。

1.2.1 发送或寄送消息

传送消息到窗口有两种方式：发送 (send)或者寄送(post)。这两种方式之间的主要差别在于被寄送的消息不必立即处理。被寄送的消息放置于一个先入先出的队列里等待应用程序空闲的时候处理，而被发送的消息需要立即处理。实际上，发送消息到窗口处理过程和直接调用窗口处理过程两者之间几乎没有任何不同。只是，你可以要求操作系统截获所有为达到某个目的而在应用程序中被发送的消息，但不能截获对窗口处理过程的直接调用。

与用户输入相对应的消息(如鼠标单击和按下一个按键)通常都是以寄送的方式传送，以便于这些用户输入可以由运行较缓慢的系统进行缓冲处理。而其余的所有消息都是被发送的。在以上的例子中，系统寄送了鼠标单击消息，而按钮窗口则向其主窗口发送了 Load File消息。

1.2.2 消息类型

有3种类型的消息：窗口消息、命令消息和控件通知消息：

■ 窗口消息(Window message)是由操作系统和控制其他窗口的窗口所使用的消息。这一类的消息有Create、Destroy和Move等等。上例中的鼠标单击消息也是一种窗口消息。

■ 命令消息(Command message)是一种特殊的窗口消息，它从一个窗口发送到另一个窗口以处理来自用户的请求。在以上例子中，从按钮窗口发送到主窗口的消息就是命令消息。

■ 控件通知消息(control notification) 是最后一种消息类型。它类似命令消息，当用户与控件窗口交互时，这一类消息就从控件窗口发送到其主窗口。但是，这种消息的目的并不在于处理用户命令，而是为了让主窗口能够改变控件，如加载并显示更多的数据。以上所述的例子中并没有控件通知消息，但是，假如按钮发送了鼠标单击消息给它的主窗口，那么这个消息也可以看作是一个控件通知消息。一个普通的鼠标单击消息可以由主窗口直接处理，然后由控件窗口处理。

1.2.3 接收消息

窗口处理过程看起来与其他函数和方法没有任何不同。消息到来后，按照消息类型排序进行处理。其中的参数则由调用函数提供以进一步区分消息。命令消息由 wParam中的命令ID分类。DefWindowProc()函数则发送任何程序员都不会去处理的消息给操作系统。所有传送到窗口的消息都将通过这个函数，甚至包括绘制窗口非客户区的消息 —— 尽管最终它们都将绕过DefWindowProc()函数。

一个主窗口的处理过程实例如下：

```
MainWndProc( HWND hWnd,UINT message,WPARAM wParam,LPARAM lParam )
{
   switch( message )
   {
   case WM_CREATE:
      :  :  :
         break;
   case WM_PAINT:
      :  :  :
         break;
   case WM_COMMAND:
```

```
        switch (id)
        {
            case IDC_LOAD_FILE:
                :   :   :
                break;
        }
            break;
    default:
        return( DefWindowProc( hWnd, message, wParam, lParam ) );
    }
    return( NULL );
}
```

1.2.4 窗口处理函数的子类化

上面提到过，在窗口类中定义了窗口处理过程的地址。由窗口类所创建的窗口将把它们的消息传递给窗口处理过程。如果程序员使用的是一个由系统提供的窗口类，并要增加自己对窗口的特殊处理，就需要使用子类化 (swbolassing)。

将窗口指向自己的窗口处理过程，便对窗口进行了子类化，这样所有的消息都可以由程序员自己处理了。如果只想处理一个或者两个消息，只需简单地将剩余消息传递给初始的窗口处理过程即可。我们注意到，采用子类化并没有修改原先的窗口类，而是直接对窗口对象进行了修改，这个对象保存了一份窗口处理过程地址的拷贝。

与此相反，超类化(superclass-ing)则修改了原始的窗口类，然后将其应用于创建窗口，但由于可以更方便、安全地使用 MFC来获得由超类化带来的好处，所以这种方法就很少采用了。

请参见图1-2了解子类化概念。

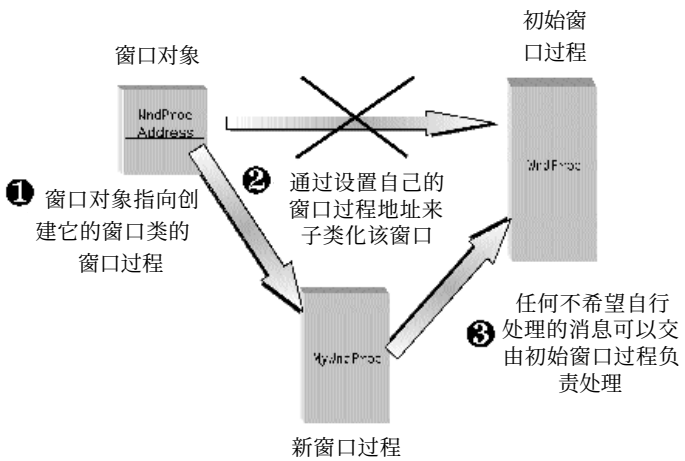

图1-2 窗口处理函数的子类化

1.3 窗口绘图

Windows API为窗口绘图提供了好几种调用函数。它们是点、弧、图形和圆绘图函数以及图形填充和位图绘制等函数。正是因为采用了绘图 API，所以程序员必须负责传递坐标数值、颜色、宽度和绘图位置，而 API函数则负责剩余的工作。为了简化对 Windows API的图像函数调用，一些参数被固化到了一个名为设备环境的可重用对象中。

1.3.1 设备环境

图像设备环境(Device Context)是一个简单的对象，它包含了对绘图而言比较共同的属性，

例如绘图位置、线宽、填充模式的颜色等等。这个对象可以一次设置然后多次重复使用。实际上并不需要自己创建设备环境，只要调用几个可能的 API函数，系统就可以返回已经被预先准备好的设备环境值。例如，系统为屏幕创建的设备环境包含屏幕上将用于绘图的颜色以及绘图工具的当前位置，程序员则可以在该位置进行绘图并设置颜色。

1.3.2 绘图工具

设备环境并没有包括绘图所需的全部特征。有几个特征存在于设备环境可以引用的附加图像对象中。这些对象都各自代表了某一特定绘图工具的特征(如：画笔的色彩和宽度、画刷采用的模式等)。这些工具包括绘制直线的画笔；用某种模式填充封闭区域的画刷；确定文本绘制效果的字体以及确定使用何种颜色的调色板等等。另外的两种绘图工具：位图和区域则显得更抽象一些。位图工具看起来像画刷，除了它只能用位图模式填充一个区域。而区域工具则类似于画笔工具，但它起的是剪切作用，例如可以使用区域工具从一个位图中将“ STOP ”单词分割出来。

1.3.3 映射模式

设备环境保持跟踪程序员采用的映射模式。设置映射模式可以指定调用参数中以英寸或者厘米作为度量单位的坐标 x和y，每个绘图函数则可以自动确定绘制多少像素。可用的映射模式如下表所示。

表1-1 可用的映射模式

模 式	使 用 说 明
MM_TEXT	这是缺省的映射模式，坐标值x和y精确地等于一个屏幕像素或者一个打印机的打印点，y的正方向沿屏幕或者打印页向下
MM_HIENGLISH	x和y值为屏幕或者打印页上一英寸的 1/1000。窗口决定当前屏幕设备应该有多少个像素等于1/1000英寸。Y的方向则为沿屏幕或者打印页向上
MM_LOENGLISH	x和y值为设备上一英寸的1/1000，y向上为正
MM_HIMETRIC	x和y值为设备上一毫米的1/100，y向上为正
MM_LOMETRIC	x和y值为设备上一毫米的1/10，y向上为正
MM_TWIPS	x和y值为设备上一英寸的1/1440，y向上为正。这个模式通常应用于绘制文本，一 twip等于一个字体点的1/20
MM_ANISOTROPIC	程序员通过设置接下来将讨论的窗口和视口以决定 x和y代表多少像素
MM_ISOTROPIC	同上，但x和y代表的像素数必须相等

1.3.4 窗口视和视口视

MM_ANISOTROPIC和MM_ISOTROPIC映射模式允许程序员自定义将坐标 x和y换算为像素数的转换比率。这个工作由定义两种叫做视的矩形来完成。首先应该定义一个代表将要绘制的整个区域(例如：0，0，1000，1000)的矩形，然后定义一个代表那些最终出现绘图结果的屏幕或打印机上的对等坐标 (例如：0，0，500，500)的矩形。第一个矩形叫做窗口视

(Window View)，第二个矩形叫做视口视(Viewport View)。如果在设备环境中定义了这两个矩形，并使用上述两种映射模式之一，那么使用窗口视坐标的图形将自动地被转换为使用视口视的坐标。除了视口视的坐标之外，程序员可以缩小、放大以及颠倒自己的图形，而不需要改变其他任何东西。

1.3.5 逻辑单位和设备单位

使用除MM_TEXT以外的绘图模式绘图的时候，传递给绘图函数的坐标采用逻辑单位(Logical Unit)。逻辑单位可以是英寸、厘米或者像素。绘图函数本身则用设备单位绘图。举一个例子，直线绘图函数可能使用254个像素代表一个1英寸的逻辑单位，这里的数字1就是逻辑单位数值，而数字254则是设备单位(Device Unit)数值。这并不会成为一个问题，除非打算让用户能够使用鼠标绘图。鼠标传回给应用程序的任何坐标都是以设备单位计量的数值，因此必须使用一些Windows API将这些坐标值转换为逻辑单位数值。

1.3.6 绘图函数

Windows API具有多个绘图函数，举例如下：

- 画点函数：如SetPixel()。
- 画线函数：如LineTo()、Arc()、Polyline()。
- 画图形函数：如Rectangle()、Polygon()、Ellipse()。
- 填充和图形反转函数：如FillRect()、InvertRect()、FillRgn()。
- 滚屏函数：ScrollDC()。
- 文本绘制函数：如TextOut()、DrawText()。
- 位图和图标绘制函数：如DrawIcon()、BitBlt()。

1.3.7 抖动和非抖动颜色

所有可用的绘图函数，包括从画线到画图形和文本，都需要使用颜色。但是，除非系统具备足够的显示内存，否则颜色将必须利用抖动 (Dithered)方式。所谓抖动颜色实际上是一些主要颜色的集合，在显示的时候通过几个颜色相互混合以获得某种理想的色彩。

对大多数情况，抖动颜色已经足够了。然而，由于抖动颜色显得比较模糊，对图像应用程序而言通常不足以达到要求。图像应用程序不希望线条与其包围的图形颜色相互渗透混合。对图像应用程序可以有以下几种选择方案：

- 增加更多的显示内存。需要颜色抖动的理由在于，虽然屏幕上的每个像素都具有自己的RGB颜色。但一般的系统都没有足够的显示内存来为每个像素存储其 RGB值。例如，设每个像素为32位，那么一个800×600像素的屏幕就需要2M字节的显示内存以容纳所有的颜色值。
- 只用标准颜色绘图。Windows使用的显示卡保证了至少能定义20种标准颜色，因为它需要使用这些颜色来创建抖动效果。
- 配置自己的颜色。除了标准颜色以外，显示卡还有一些空间可定义200多种颜色，可以在设备环境调色板内定义这些颜色并只用这些颜色绘图。多数图像应用都采用了这种方法，此方法将在实例31中得到详细描述。

1.3.8 设备无关位图

每个像素都有自己的RGB颜色，每种颜色的数值范围可以为0～254。一排同样颜色(例如黑色)的像素在屏幕上显示一条直线，一个具有不同颜色像素的矩形就可以创建任何图像。将这些像素的色彩数值存到一个文件中就得到了一个位图文件。这个文件的头不仅要指示文件包含了多少颜色值，而且还要指出多少颜色值组成一排。

如果位图文件中的每个颜色值都包含完整的 RGB数值，那么，由于这个颜色值完全在位图中得到定义，这个文件就是一个设备无关位图。如果每个颜色值实际上都是对某个颜色表的字节索引，那么，在它同时包含了这个颜色表的情况下，这个文件仍然是设备无关的。像这样的颜色索引常用于压缩位图的大小。一个 8位索引只占用32位RGB值空间的四分之一。

设备无关位图(Device Independent Bitmaps)由对颜色表的索引组成，这个颜色表在系统的显卡中被定义。这就是在上节的第 3步为显示非抖动颜色而配置的颜色表，如果某个位图指向它的话，那么这个颜色表将不能独立于设备之外而存在。

1.3.9 元文件

除了在屏幕或者打印机上绘图，还可以在文件中绘图，这样的文件就是元文件 (Metafile)。元文件的优点在于：无论绘制的内容是什么，它们都将完整地重现。由于元文件可以伸展得到更好的效果，而位图进行伸展时会扭曲变形，所以元文件优于用位图的形式存储图像。

1.3.10 何时绘图

看起来这个话题可能有点傻，但对一个多任务操作系统而言，应用程序之间不得不争夺屏幕上有限的显示空间。哪怕只是在自己的窗口中绘图，也不可能想做什么就做什么。通常只是在窗口接收到 WM_PAINT消息或者WM_DRAWITEM消息(针对所属的控件窗口)的时候才绘图。系统仅在窗口被部分遮盖以及由于关闭其他窗口而显现出来的时候才发送这些消息。或者说，如果需要重画以显示新的信息，系统在这种情况之下将发送这些消息。

1.4 MFC基础

到目前为止，我们只讨论了应用程序可以从 Windows API中所获得的功能。但API并不是面向对象的。例如，使用API不可能在创建一个窗口实例之后再调用其成员函数作用于该窗口。并且，不能从窗口类派生出一个可以加入自己所需功能的类，比方说，不可以增加自己的窗口处理过程。

微软基础类库所要做的就是向应用程序提供可以访问 Windows API的一种模拟面向对象的访问方式。在功能上，每个 MFC类都紧密结合一种Windows资源对象(如窗口)，而API函数则控制该资源。

举个例子， MFC的CWnd类创建并控制窗口。而操作系统一创建窗口就会开辟一块叫做Window对象的管理内存，然后返回该对象的指针，也就是窗口句柄。 MFC CWnd类则以成员变量的形式存储这个句柄，并且由 CWnd的成员函数调用该变量来控制该窗口。例如， CWnd的MoveWindow()成员函数调用Windows API::MoveWindow()来移动属于该窗口句柄的窗口。因为MFC是用C++写成的，所以，程序员可以从CWnd类派生自己的类并在其中增加所需功能。

但是，因为这只是面向对象设计的一个模拟，所以 MFC类对操作系统内部工作的控制能力并不会比任何其他 API调用更多。例如，如果对窗口打开方式的修改是 API的内部功能，那么即使使用MFC类来试图改变这一方式也是不可能的。

创建和销毁MFC类

创建MFC类是一个棘手的问题。对封装了类似窗口系统资源的 MFC类来说，程序员不仅要创建自己的MFC类的实例，还要调用该类的成员函数来创建该系统资源。因此 MFC类的创建几乎总是分成两步走：1)创建一个类的示例；2)创建系统资源。

注意 为什么MFC类不仅仅只是简单地在它们自己的构造函数内创建系统资源呢？这是因为，创建系统资源是否成功是不可能预先知道的，而类的构造函数又难于访问 (它甚至不返回错误状态)，利用成员函数来完成这一工作则要容易得多。

销毁MFC类也同样棘手。如果类的实例首先被销毁，那么它将简单地在其析构函数内销毁资源。然而，如果开始就没有资源，那么 Windows API也就没有办法知道有一个 MFC 类实例必须被销毁。令人惊奇的是，这只是存在于 CWnd类和窗口资源之间的问题，其他类型的资源并不由用户销毁。

但是用户可能通过单击窗口的关闭按钮来关闭窗口，在这种情况下，不知何故， CWnd对象必须知道足够的信息来销毁自己以防止出现内存泄漏。幸运的是，当窗口资源被销毁时，它会发送一个消息，而CWnd对象可以捕获这个消息以用于销毁自身。

因为MFC类对象和系统资源是两种不同实体，所以可以通过编程的方法将两者分开或者重新组合在一起。例如，通过调用 Attach()函数可以将一个CWnd类对象附着于一个已经存在的窗口对象上。所有控制系统资源的 MFC类都具有Attach()函数和Detach()函数。

1.5 Developer Studio基础

为了将 MFC类恰当地运用于应用程序， Developer Studio(开发平台)提供了几种向导(Wizard)工具和编辑器工具：

■ AppWizard 用于生成应用程序所需要的基本类文件。所产生的类都派生于 MFC类，它们在编译后与MFC库链接以创建应用程序。

■ ClassWizard 用于创建应用程序额外的文件或者为已有的类增加新的成员函数。这些被创建的类可以由MFC派生。

■ Dialog Editor 用于创建对话框模板，方法是将控件窗口图标拖到一个空白的框架窗口内。被创建的模板作为应用程序的资源存储，然后用于在运行时候创建对话框。 ClassWizard可以直接从Dialog Editor调用以创建一个对话框类，该类则负责创建对话框。

■ Toolbar Editor 用于创建工具栏和位图资源，而这些资源则又用于创建应用程序的工具栏。

■ Cursor、Icon和Bitmap Editor 是简单的图像编辑器，用于创建应用所使用的光标、图标和位图资源。

■ Menu Editor 用于创建应用中的菜单条和弹出菜单资源。

■ String Editor 用于创建字串资源，它可以将文本字符串从应用程序中分离出来，并且可以很方便地从一种语言转变为另一种语言 (例如从英语转变为法语，而不是从 C++到JAVA)。

■ Text Editor 用于编辑类文件。

1.6 Windows和MFC总结

以上所述说明了Windows、MFC和Developer Studio是如何一起协同工作的。Windows操作系统创建并支持应用程序，其中包括窗口的创建。MFC则在C++类中封装了这一功能，而Developer Studio则负责创建这些类。

现在来回顾一下MFC提供了哪些类。

1.7 基本类

多数MFC类由下列3种基本类派生：CObject、CCmdTarget和CWnd。如上所述，CWnd类封装了创建和控制窗口的 Windows API，它还允许程序员向窗口处理函数中添加自己的消息处理过程。CCmdTarget类则允许没有创建窗口的类也能处理消息，但只能是后面要讨论的所谓命令消息。CObject类为每个从它派生的类提供了许多基本功能，例如得到类对象的大小，将对象存入一个磁盘文件等等。

1. CObject类

CObject 类本身并没有提供什么重要的功能。该类通过 6种相互配合的宏(macro)完成实际工作。这些宏使得类在运行时可以从 Cobject类派生以获得类的名字和对象大小。创建一个这样的类不必要知道类的名字，文档环境存储和接收这种类的实例也不必知道类的名字。

下列宏允许类的实例知道它自己的类名字和运行时的类大小：

```
DECLARE_DYNAMIC( CYourClass )                    // in the .h file
IMPLEMENT_DYNAMIC( CYourClass, CYourBaseClass)   // in the .cpp file
```

使用CObject::GetRuntimeClass()函数可以在运行时使用这些宏来获得与类有关的细节。

另外几个宏包括以上宏的功能，但同时允许类的实例在不知道它的类名字的情况下被创建：

```
DECLARE_DYNCREATE( CYourClass )                  // in the .h file
IMPLEMENT_DYNCREATE( CYourClass, CYourBaseClass )  // in the .cpp file
```

使用CObject::CreateObject()函数可以利用这些宏创建一个类的实例而无需知道它的类名字。

另外几个宏包括以上所有宏的功能，但同时允许类实例在不知道它的类名字的情况下被存贮：

```
DECLARE_SERIAL ( CYourClass )       // in the .h file
IMPLEMENT_SERIAL ( CYourClass , CYourBaseClass , schema )
                                    // in the .cpp file
```

2. CCmdTarget类

从CCmdTarget类派生的类可以接收并处理由应用程序的菜单或者工具栏发出的命令消息。CCmdTarget类将在以后的消息机制部分详细讨论。

3. CWnd类

如上讨论，CWnd类的成员函数封装了负责创建和维护窗口的 Windows API。CWnd类派生于CCmdTarget类，因此能够接收和处理命令消息。所有其他的控制窗口的 MFC 类都由该类派生。

注意 本章所使用的下列字母用于指出MFC类从以上的何种基类派生：

■ O代表派生于CObject。
■ OC代表派生于CObject和CCmdTarget。
■ OCW代表派生于CObject、CCmdTarget和CWnd。

1.8 应用类

AppWizard以下列4种MFC类为基础，为应用程序产生出一些派生类：

1) CWinApp 也就是应用程序的应用类(Application Class)，它负责初始化并运行应用程序，这就是以上讨论的程序线程。

2) CFrameWnd 也就是应用程序的框架类(Frame Class)，它负责显示并跟踪用户命令以及显示应用程序的主窗口。

3) CDocument 应用程序的文档类(Document Class)，它负责加载和维护文档。文档可以是从草稿到网络设置的任何东西。

4) CView 应用程序的视类(View Class)，它负责为文档提供一个或者多个视。

注意 这里使用了应用类、框架类等术语，但本书所指的都是以上4种基类的派生类。

这4种类中的哪些类将包括在应用程序中，取决于所创建应用程序的类型。

■ 对话框应用程序(Dialog Application)，只是简单地拥有作为用户界面的对话框而没有框架、文档或者视类。对话框应用程序仅利用了应用类 CWinApp的派生类。对话框则用 MFC的CDialog类创建，这个类将在以后讨论。

■ 单文档界面应用程序(SDI:Single Document Interface Application)，可以一次加载并编辑一个文档，它使用以上提到的全部 4种基类。

■ 多文档界面应用程序(MDI:Multiple Document Interface Application)，可以一次加载并编辑几个文档，它使用以上提到的全部 4种基类，此外还增加了两个 CFrameWnd的派生类：CMDIFrameWnd和CMDIChildWnd。

1.8.1 文档视

CDocument和CView类的派生类负责文档视。 MFC应用程序是面向文档的，这意味着应用程序负责加载、观察、编辑并存储文档，而文档则可能是文本文件、图形图像或者二进制配置文件。文档类的工作是将文档从磁盘加载到它的成员变量。然后创建一个或者多个视类以显示这些成员变量。文档类仅需为文档类对象创建多个视类对象就可以拥有多个视。因为文档类没有关联的窗口，所以它并不是从 CWnd类派生的，而是从CCmdTarget派生的，因而可以处理命令消息。

1.8.2 CWinApp(OC)

应用类是应用程序运行时创建的第一个对象，并在应用程序执行的过程中最后一个终止。启动后，应用类就负责创建应用程序的其他对象。

■ 针对对话框应用程序，应用类用 CDialog创建一个对话框。
■ 针对SDI应用程序，应用类创建一个或者多个文档模板 (参阅以后内容)，然后使用该模

板打开一个空文档。

■ 针对MDI应用程序，应用类创建一个或者多个文档模板，然后使用该模板在主框架类内打开一个空文档。

应用类派生于CWinApp并从AppWizard中得到类似CXxxApp的类名字，Xxx就是具体应用的名字。

1.8.3 文档模板

文档模板定义了在应用程序打开一个文档时框架类、文档类和视类的创建结果。为了生成一个文档模板，必须为SDI 应用程序创建一个CSingleDocTemplate类或者为MDI应用程序创建一个CMultiDocTemplate类，并用3个类指针对其初始化：

```
pDocTemplate = new CMultiDocTemplate(
    IDR_APPTYPE,
    RUNTIME_CLASS( CAppDoc ),          // Your Document Class
    RUNTIME_CLASS( CChildFrame ),      // Your Frame Class
    RUNTIME_CLASS( CAppView )          // Your View Class
    );
```

其中的RUNTIME_CLASS()宏返回一个指向类的 CRuntimeClass结构的指针，DECLARE_DYNCREATE和IMPLEMENT_DYNCREATE 宏则将该结构添加到类中。文档模板通过创建以上3个类的实例并调用其CRuntimeClass::CreateObject()函数打开文档。

1.8.4 线程

CWinApp类本身从CwinThread类派生。而CWinThread类则封装了创建和维护系统中具体应用程序线程的Windows API。实际上，可以通过创建 CWinThread类的另一个实例以在应用程序中实现多任务。读者可以参考实例 56和实例57。CWinApp类代表了应用程序中的执行主线程。

1.8.5 CFrameWnd(OCW)

框架类是应用程序运行时创建的下一个对象，它负责显示并为应用程序引导用户命令。

对于SDI应用程序，框架类派生于 CFrameWnd类，AppWizard将自动为它分配一个名字：CMainFrame。

对于MDI应用程序，框架类则派生于 CMDIFrameWnd类，AppWizard也为它分配CMainFrame这个名字。同时 MDI应用程序还为每个打开的文档创建一个子框架类 (Child Frame Class)。每个子框架类都派生于CMDIChildWnd，AppWizard自动为子框架类分配名字：CChildFrm。

对话框应用程序没有框架类，正如上面提到的，对话框应用程序由应用类和对话框类组成。

1.8.6 CDocument(OC)

文档类通常是应用程序创建的下一个类，应用程序或者打开一个新文档，或者打开一个已经存在的文档。文档类负责将文档加载到其成员变量中，并允许视类编辑这些成员变量。文档可以包括从图像文件到可编程控制器设置等任何内容。

文档类派生于CDocument类，AppWizard自动为其分配的名字为CXxxDoc，其中Xxx是应用程序的名字。

1.8.7 CView(OCW)

文档类的实例创建之后，紧接着就会创建一个视类的实例。视类负责描述文档类的内容。视类还允许用户编辑文档。分离窗口类CSplitterWnd则允许文档同时具有一个以上的视，这些视可以由好几个同样的或者不一样的视类创建。

AppWizard允许程序员从下列几个基类派生自己的视类，它们是：CTreeView、CEditView、CRichEditView和CListView等等。每一种基类给予应用程序一组不同功能。所有这些类都从CView类派生。无论选择什么基类，AppWizard自动为派生类分配的名字为CXxxView，其中Xxx是应用的名字。

如上所述，可以从这4种基类创建3种类型的MFC应用程序：对话框、SDI和MDI。下面详细介绍一下这些应用程序类型。

1.8.8 对话框应用程序

对话框应用程序由一个应用类和对话框类组成，应用类是由CWinApp派生的，对话框则由一个从对话框类派生的类创建。

1.8.9 SDI应用程序

SDI应用程序由一个派生于CWinApp的应用类、一个派生于CFrameWnd的框架类、一个派生于CDocument的文档类和一个或者一个以上的视类组成，这些视类由一种或几种以CView类为基类的视类派生。

1.8.10 MDI应用程序

MDI应用程序由一个派生于CWinApp的应用类、一个派生于CMDIFrameWnd的框架类、一个或者一个以上派生于CMDIChildWnd的子框架类和多个文档类和视类组成。其中，每个子框架类的对应文档派生于CDocument类，每个文档对应的一个或者多个视类则派生于CView类。

1.9 其余用户界面类

除了框架类和视类，MFC还提供了几种其他的类来支持用户界面：

- 通用控件类(Common Control Class)，封装了诸如按钮一类的通用控件。
- 菜单类(Menu Class)，是CWnd类为窗口提供的菜单。
- 对话框类(Dialog Class)，封装了对话框和通用对话框。
- 控件条类(Control Bar Class)，封装了控件条(工具栏、对话条和状态栏等)。
- 属性类(Property Class)，封装了属性表和属性页。

1.9.1 通用控件类

通用控件类封装了通用控件(如按钮、列表框等)的功能。这些类派生于CWnd类并继承了

其成员函数如ShowWindow()和MoveWindow()等等。当这些类创建一个窗口时，它们中的每一种都将使用一种通用控件窗口类。例如，当 CButton通用控件类创建一个按钮时，它将使用BUTTON窗口类来创建实际的窗口：

```
Creat(_T("BUTTON"), lpszCaption, dwStyle, rect, pParentwnd, nID);
```

下表列出了这些通用控件类、它们所创建的控件以及所使用的窗口类。

表1-2 MFC通用控件类和它们的窗口类

MFC类	通用控件	Windows类
CAnimateCtrl(OCW)	Animation动画控件	SysAnimate32
CButton(OCW)	按钮控件	BUTTON
CComboBox(OCW)	组合框控件	COMBOBOX
CEdit(OCW)	编辑控件	EDIT
CHeaderCtrl(OCW)	标头控件	SysHeader32
CListBox(OCW)	列表框控件	LISTBOX
CListCtrl(OCW)	列表控件	SysListView32
CProgressCtrl(OCW)	进度控件	msctls_progress32
CScrollBar(OCW)	滚动条控件	SCROLLBAR
CSliderCtrl(OCW)	滑块控件	msctls_trackbar32
CSpinButtonCtrl(OCW)	上/下按钮控件	msctls_updown32
CStatic(OCW)	静态控件	STATIC
CTreeCtrl(OCW)	树型控件	SysTreeView32
CTabCtrl(OCW)	标签控件	SysTabControl32
CDateTimeCtrl(OCW)	日期/时间获取控件	SysDateTimePick32
CMonthCalCtrl(OCW)	日历控件	SysMonthCal32
CHotKeyCtrl(OCW)	热键控件	msctls_hotkey32
CToolTipCtrl(OCW)	工具提示控件	tooltips_class32

并不是所有的通用控件类都只是简单地封装一个通用控件窗口类。有 3种MFC类实际上提供的功能在通用控件内就找不到。表 1-3显示了这些类、派生它们的 MFC类和它们所提供的支持。

表1-3 MFC派生类和它们的基类

MFC类	MFC继承类	新增的功能
CBitmapButton	CButton	对按钮上的位图提供了更好的支持
CCheckListBox	CListBox	列表框中的复选框
CDragListBox	CListBox	列表框中的用户可拖动项目

1.9.2 菜单类(O)

CMenu类封装了创建和维护菜单的 Windows API。CMenu有两个成员函数： Attach()和Detach()。这两个函数允许程序员采用类似 CWnd类对象包含一个已有窗口的方法包含一个已

有菜单。

1.9.3 对话框类

CDialog类封装了创建对话框的Windows API，对话框创建时是一种弹出窗口，可以向对话框类增加在对话框模板中定义的控件窗口。

1.9.4 通用对话框MFC类

MFC库还有6种通用对话框类，它们封装了创建通用对话框的 Windows API。通用对话框是预先加入了控件的对话框，它们以一些普通请求信息提示用户，如要装载和存储的文件名、颜色、字体和打印参数等。它们简化了由程序员自己编写这些对话框的工作，而且向用户展示了他们所熟悉的Windows对话框。

表1-4显示了通用对话框的功能、提供的 Windows API以及封装的MFC通用对话框类。

表1-4 通用对话框类及其使用

通用对话框	Window API调用	MFC类
选择颜色	::ChooseColor()	CColorDialog
打开/保存文件	::GetOpenFileName()	CFileDialog
	::GetSaveFileName()	
“查找”或“替换文本”	::FindText()	CFindReplaceDialog
	::ReplaceText()	
选择字体	::ChooseFont()	CFontDialog
打印页面设置	::PageSetupDlg()	CPageSetupDialog
打印	::PrintDlg()	CPrintDialog

使用通用对话框的实例可以参考实例 23和24。

1.9.5 控件条类 (OCW)

控件条类封装了向应用程序提供工具栏、状态栏、对话条和伸缩条的 Windows API。第2章将更详细地讨论它们。

- CToolBar(OCW)和CToolBarCtrl(OCW)类创建并维护工具栏。
- CStatusBar(OCW)和CStatusBarCtrl(OCW)类创建并维护状态栏。
- CDialogBar(OCW)类创建并维护对话条。
- CRebar(OCW)和CRebarCtrl(OCW)创建并维护伸缩条。

1.9.6 属性类

属性类封装了向应用程序提供属性页和属性表功能的 Windows API。属性表由一个或一个以上的属性页组成，它可以创建Windows用户熟悉的标签视。该视一般用于选择程序选项。

- CPropertySheet(OCW)类创建属性表，CPropertySheet类并不是从CDialog类派生的，但它们很相似。
- CPropertyPage(OCW/CDialog)类创建属性页，它是从CDialog派生的。

1.10 绘图类

CDC类封装了早先讨论的设备环境以及所有需要设备环境的绘图函数 (这个比例不小)。除了CDC类以外，还有另外4种从CDC派生出来的MFC类提供了附加功能，包括：

■ CClientDC类，一般用来方便地创建和销毁一个设备环境。 CClientDC通常在堆栈上创建。它的构造函数通过调用CDC::GetDC()为窗口客户区创建一个设备环境。当子程序返回时，CClientDC的析构函数通过调用CDC::ReleaseDC()函数销毁该设备环境。这样将不会存在忘记释放的设备环境，从而也不会导致内存资源泄漏。

■ CWindowDC类，与CClientDC类相似，但它的工作范围是窗口的非客户区。

■ CPaintDC类，在构造并得到设备环境后将调用 CWnd::BeginPaint()函数，这种情况下的设备环境仅允许绘制无效化的窗口客户区而不是绘制整个窗口。销毁 CPaintDC类在时将调用CWnd::EndPaint()函数。

■ CMetaFileDC类，用于创建元文件，如上所述，元文件是一种磁盘文件，它包含了所有的绘图行为以及绘制图像所需要的绘图模式。可以通过打开一个元文件设备环境来创建元文件，然后使用绘图工具来绘制它，就好像这个文件是计算机屏幕或者打印机设备。所产生的文件可以在其他设备上重新读取以创建图像。

绘图工具和类

MFC使用类来封装绘图工具的特性：CPen代表画笔，CBrush代表画刷，CFont代表字体，CPalette代表调色板，CBitmaps代表位图，CRegion则代表区域。这些类中的每一种创建相关联的图像对象，然后将其选入设备环境。

1.11 其他MFC类

并不是所有的MFC类都会影响到用户界面。几种其他的 MFC类还封装了控件文件、数据库和Windows套接字的API。还有一些类负责维护数据集合 (列表、数组和映射等)。

1.11.1 文件类

CFile(O)类封装了创建并维护一个普通文件的 Windows API。有3种MFC类是从CFile类派生并提供了附加功能：

■ CMemFile类允许在内存而不是在磁盘上创建文件。在构造一个CMemClass对象的时候，文件被立即打开，可以使用成员函数对该文件读写，就好像它是一个磁盘文件一样。

■ 除了用于创建文件的内存位于全局堆上之外， CShared File类与CMemFile完全类似。这使得它可以通过剪贴板和DDE共享。

■ CStdioFile类允许读写以回车换行控制符结尾的文本字符串。

1.11.2 CArchive和序列化

CArchive类在序列化的过程中使用 CFile类将文档的类对象存入磁盘。采用序列化，类中的成员变量和整个类对象可以按照某个顺序存入一个归档设备，以后则可以按照同样的顺序恢复。

1.11.3 数据库类

MFC库有一些支持两种数据库的类：

■ ODBC(Open Database Connectivity)类封装了多数数据库开发商所支持的 ODBC API。如果应用程序使用 MFC的ODBC类，它可以支持任何遵守 ODBC标准的数据库管理系统(DBMS:Database Management System)。

■ DAO(Data Access Object)类支持由Microsoft Jet数据库引擎所优化的更新数据库 API。使用该引擎仍然可以访问ODBC兼容数据库或者其他数据库。

1.11.4 ODBC类

有3种ODBC类：

1) CDatabase(O)类使用ODBC API打开DBMS 数据库。在构造一个 CDatabase对象以后，可以使用它的OpenEx()函数与一个数据库建立连接。调用 CDatabase的Close()函数则可以关闭该连接。

2) CRecordset类用于通过数据库连接来存储和接收记录。

3) CDBVariant类代表了记录中的一列，它并不关心其数据类型。

1.11.5 DAO类

DAO类有3种类似ODBC的类，它们是：

1) CDaoDatabase(O)类打开DAO数据库。

2) CDaoRecordSet(O)拥有记录。

3) COleVariant表示记录列。

DAO类还包括以下3种类：

■ CDaoWorkSpace(O)类，管理数据库会话，该会话允许执行事务 (存储到数据库)或者不执行。

■ CDaoQueryDef(O)类，表示一个查询定义。

■ CDaoTableDef(O)类，表示一个表的定义，包括域和索引结构。

1.11.6 数据集合类

数据集合类维护并支持数组、列表和数据对象映射。

■ CArray类和它的派生类支持数据对象数组。一个数组由一个或者多个同样数据对象(如：整型数、类)所组成，它们在内存中相邻，因而可以用简单的索引来访问。 CArray类可以动态地增大或者减小其大小。 CArray类有几种派生类(如CByteArray和CDWordArray)允许创建某种安全类型的数组，当然，可以用 CArray<type，arg_type>模板创建类创建任何安全类型的数组。

■ CList类和它的派生类支持数据对象的链表。链表由一个或者一个以上的相同类型数据对象(如：整型数、类)所组成，它们在内存上不连续，通过双向链接可以前后遍历链表。几种CList类的派生类(如CPtrList和CObList)允许创建安全类型链表。当然，可以用 CList<type，arg_type>模板类创建任何安全类型的CList。

■ CMap类和它的派生类支持数据对象的字典。数据字典在二进制或者文本键值下存储一个或者一个以上同样类型的数据对象 (如：整型数、类)。可以使用该键值接收数据。例如，因为Windows并不跟踪哪个MFC CWnd对象属于哪个窗口。应用程序就可以使用一个 CMap对象类将窗口句柄，与其对应的 CWnd对象相关联。 CMap类有几种派生类 (如CMapWordToPtr, CObToString)允许创建保护类型的字典。当然，可以用 CMap<class KEY，class ARG_KEY，class VALUE，class ARG_VALUE>模板类创建任何安全类型的CMap。

1.11.7 通信类

MFC库提供了一些类以使得应用程序通过网络或 Internet进行通信。这些类将在第 3章详细讨论。

1.12 类的消息机制

本章结束之前最后需要学习的内容是 MFC如何处理消息。如上所述，每个窗口都有一个窗口处理过程处理任何发送到该窗口的消息。同时，窗口处理过程一般还写成使用 switch-case语句的判断模块以对不同消息分类各自处理。实际上，这种判断模块可能会很大，而且难以维护。 MFC的解决方案是将这些消息重定向到 CWnd类的成员变量。这样就可以利用类的安全性和方便性来处理消息。

MFC甚至还可以发送一些消息给其他非 CWnd派生类处理，这在技术上叫做命令路由，以后章节将详细讨论它。

1.12.1 MFC如何接收一个寄送消息

消息既可以被发送也可以被寄送。发送的消息在本质上与直接调用窗口处理过程一样，就好像它是另一个函数。寄送消息则进入一个在应用程序初始化时操作系统所建立的消息队列中。鼠标和键盘单击一般产生寄送消息，然后应用程序一个接一个地将它们从消息队列中删除，并将它们发送到被鼠标单击的窗口或者按键按下时接收输入的窗口。

Windows API提供了两个调用函数， GetMessage()和PeekMessage()，它们允许应用程序从队列中删除消息。 MFC类CWinThread将这些函数调用封装到Run()函数中。 Run()是MFC应用程序开始执行时最后被调用的函数。然后 Run()函数就不断检查消息队列以等待用户的按键或者鼠标单击等操作。 Run()函数还执行一些MFC类的后台维护工作并为程序员提供机会进行自己的维护工作。

读者可以参考实例47和48以充分理解运用这一机制的有关内容。

一旦从消息队列中删除一条寄送消息，并将其发送到一个窗口，对它的处理就与接下来所讨论的发送消息一样了。

1.12.2 MFC如何处理接收的消息

所有MFC被控窗口的基本窗口处理过程是静态函数 AfxWndProc()。只要决定使用MFC将一个窗口子类化，那么该函数的地址就由 MFC将其附着于适当的窗口对象。当一个消息到来时， AfxWndProc()就以某种方法调用成员函数来处理该消息并把其余的消息传递给原始窗口

处理过程。

但除了这种简单的消息处理过程之外，MFC还在消息机制中建立了两类与AfxWndProc()函数不同的附加特性：命令路由和消息反射。

■ 命令路由(Command Routing)，几乎所有由菜单和工具栏产生的命令消息都转发到主框架类窗口进行处理。因为许多这样的消息在其他应用类中可能会得到更好的处理。例如视类或者文档类，所以AfxWndProc()将自动地通过一种名叫命令路由的过程将这些消息发送给应用程序。该过程还允许MFC不用控件窗口(例如，文档类)就能处理命令消息。

■ 消息反射(Message Reflection)，正如上面提到的，控件通知消息的发送目标是控件的父窗口而不是控件自己。因此单独将控件窗口以AfxWndProc()子类化将不会允许在自己的控件类中处理这些消息。因为在控件父窗口中增加控件特定代码是违反面向对象编程原则的，所以父窗口类的AfxWndProc()函数自动地将这些消息弹回控件类以允许其处理通知消息，这个过程就叫作消息反射。

最后要考虑的就是AfxWndProc()的性能问题。如果使用标准的C++技术，那么将到来的消息引导到成员函数的最简单方式就是为每个可能的消息重载基类中的哑元函数。如果从这些类派生并重载了这类函数，这些消息就可以由自己来处理了。但是，任何涉及窗口的活动一般都要产生一大堆窗口消息。如果每一个这样的消息都要通过这些哑元函数来过滤的话，这样大的开销是不允许的。因此在基类中不会有哑元函数，作为一种替代方式，从基类中派生的每个类将具有一个成员函数和它们处理的消息之间的嵌入消息映射。AfxWndProc()则直接访问该消息映射(Message Map)。如果到来的消息没有对应的映射入口，该消息就很快地被转发给原始窗口处理函数。为了提高这个速度，MFC甚至不惜使用汇编程序来处理。

AfxwndProc()函数使用6个辅助函数，并按照下列调用顺序来完成所有以上功能：

1) AfxWndProc()接收消息，找出消息所属的CWnd对象然后调用AfxCallWndProc()函数。

2) AfxCallWndProc()存储消息(消息ID和参数)供今后索引，然后调用WindowProc()函数。被存储的消息用于不处理该消息的情况。

3) WindowProc()运行期间发送消息给OnwndMsg()函数，如果不处理消息则传送给DefWindowProc()函数。

4) OnWndMsg()将消息按照其类型分类，对WM_COMMAND消息，OnWndMsg()调用OnCommand()函数。对WM_NOTIFY消息，OnWndMsg()调用OnNotify()函数。剩下的则是窗口消息，OnWndMsg()于是搜索消息映射找寻对应的消息处理器。如果没有找到，它就将消息返回给WindowProc()函数，该函数则随后将消息发送给DefWindowProc()函数处理。

5) OnCommand()检查消息是否是一个控件通知消息(lParam的值不是NULL)。如果是，OnCommand()将把消息反射回发出通知消息的控件。如果不是控件通知消息，或者控件拒绝被反射的消息，那么此时OnCommand()就调用OnCmdMsg()函数。

6) OnNotify()也将消息反射回控件，如果不成功则调用同样的OnCmdMsg()函数。

7) OnCmdMsg()函数，取决于是什么类接收消息，它会暗中利用命令路由过程以发送命令消息和控件通知消息，例如，如果拥有某个窗口的类是框架类，命令和通知消息就会被发送到视类和文档类中以寻找该消息的消息处理器。

为回顾这一过程，请参见图1-3。

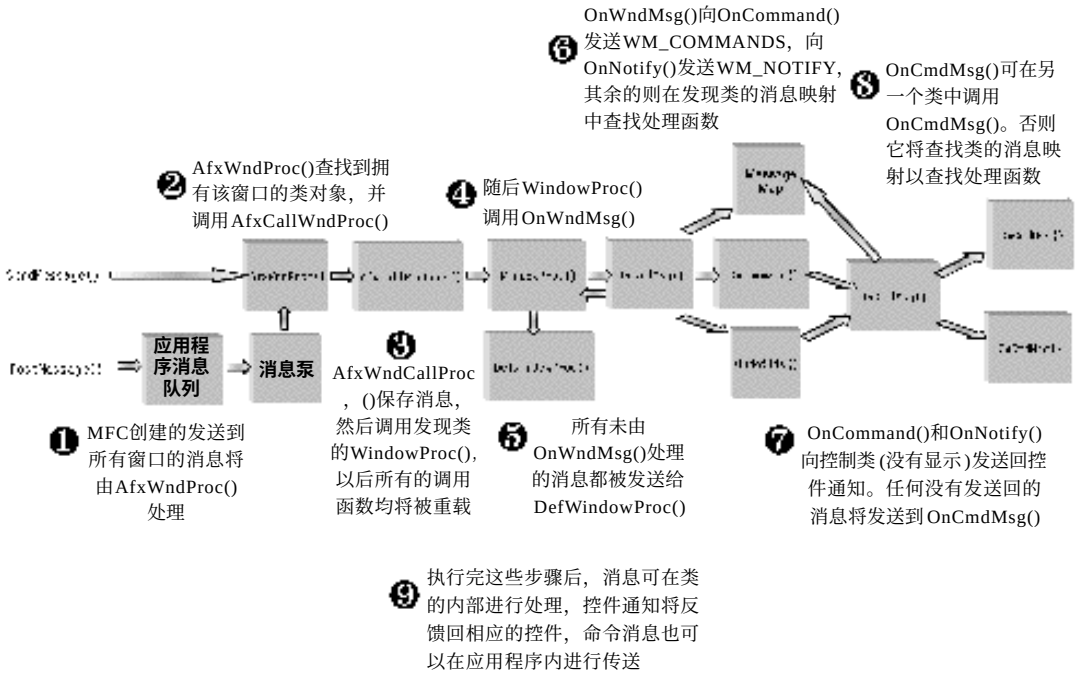

图1-3 MFC消息处理概述

1.12.3 UI对象

OnCmdMsg()不仅是处理命令消息的辅助函数，还用于自动地启用、禁用以及标识菜单项、状态栏窗格和工具栏按钮等。换句话说，既然 OnCmdMsg()函数已经为将命令从菜单项传送到类的成员提供了所有必要的底层支持，那么为什么不让类的成员来修改菜单项的外观呢？

只要用户拉下菜单，主框架类将通过调用 OnCmdMsg()函数遍历该菜单所有的命令 ID。但并不是查找一个成员函数来处理这个命令， OnCmdMsg()函数只是查找一个可能在该菜单项处增加标记或者改变其文本、将其禁用的成员函数。

只要应用程序空闲等待一个新的寄送消息，它就使用 OnCmdMsg()函数遍历所有的工具栏按钮和状态栏窗格，并再次查找用户界面处理器以改变它们的外观。

1.13 小结

本章回顾了Windows应用程序的基础知识，包括其窗口、消息机制和绘图机制，并讨论了MFC如何封装Windows API以提供一种相当不错(但还不是完美)的C++接口来创建Windows应用。此外还回顾了一些重要的MFC类，以及MFC如何引导窗口消息到类的成员函数。

本章讨论了很多内容，但只是希望作为一个已有知识的回顾。如果要了解这方面的更多知识，请参考《Visual C++ MFC编程实例》(已由机械工业出版社出版)。

第2章将深入讨论一个问题：控制条。