引　言

Microsoft Windows 是一种快速成长的应用程序开发平台。自从 1990 年 5 月发行 Windows 3.0 以来，软件开发公司已经开发了几千种 Windows 应用程序。同时，Microsoft 公司已通过对 Windows 编程 API 扩充和增加其鲁棒性扩展了 Windows 平台。

Windows 3.1 在当今的 Windows 产品中表现出最大的优越性。本书覆盖了两个 Windows API 扩展领域：动态数据交换管理库(DDEML)和对象链接及嵌入库(OLE)。表 0.1 表示了 Windows API 的进化过程。

表 0.1　Windows API 的进化

版本号	发行日期	函数调用的数目
Windows 1.0	11/85	379
Windows 2.0	11/87	458
Windows 3.0	5/90	578
Windows 3.1	4/92	771

附录 B 只给出部分的 Windows API，因为 Multimedia 和 Pen Extension 包含了另外 275 个函数。Windows 应用程序的开发者用大量的工作用来理解若干 468 新函数，它们为所有 1046 个函数的一部分。本书可帮助需要理解 DDEML 和 OLE API 的软件开发者。

0.1　概述

本书是为中级 Windows 程序员进入高级程序员而准备的，他们需要学习有关 DDEML 和 OLE API 的有关知识。本书是在假定读者已经掌握了初级编程教程中的编程规则而写成的。本书每章都附有实例程序，它们的重点都在 OLE 或 DDEML 上，而不解释基本的 Windows 编程。

本书分为二大部分，每部分包括四章内容。第一部分覆盖了 DDEML API。第 1 章是基本概念，用基于消息的 DDE 术语回顾了 DDE 协议。第 2 章至第 4 章通过使用详尽的实际应用程序描述 DDEML API；第 2 章采用一个 DDEML 服务器实用程序作为讨论的焦点。第 3 章通过介绍 DDEML 客户应用程序完成客户/服务器的 DDE 程序的循环体。第 4 章解释了为监视器应用程序而对 DDEML API 所做的扩充。

第二部分讨论 OLE。第 5 章是基本概念部分，在其中介绍开发 OLE 应用程序的结构和过程。第 6 章完成一个 OLE 服务器实用程序的例程。OLE 客户应用程序在第 7 章中介绍。本书最后一章是开发 OLE 对象处理程序的入门。

撰写本书最困难的莫过于提供用于示范完成 DDEML 和 OLE 应用程序的例程。其结果就是在本书中占有很大篇幅的应用程序，对于 OLE 应用程序尤其突出。从个人的立场

来说，我更喜欢阅读配有支持正文的原代码。但从实践的观点出发，我又不得不使本书限制在一定的篇幅内。因此，书中很明显地表现为程序代码多于正文。这两种内容的组合应该能给读者提供编写 OLE 和 DDEML 应用程序所需的详尽资料。

我采用中存储模式来编译书中的实用程序，因为我把这些程序看成实际应用程序的起点。因我没有要生成多大的应用程序这样一个概念，中存储模式更恰当一些，因为采用中存储模式编写比起小存储模式更容易生成特大规模的应用程序。

0.2 所需的编程工具

读者需要 Windows 3.1 和 C 或者 C++编译器，它包括了开发 Windows 3.1 应用程序所需的库和头文件，或者 Microsoft Windows 软件开发工具包(SDK)。我使用下列开发工具来编写书中的实例程序：

Windows 3.1

Microsoft C 6.0a

Microsoft Windows 3.1 软件开发工具包(SDK)

开发 Windows 3.1 应用程序最小的硬件需要和运行 Windows 3.1 的相同。它们是 80286(或更高)PC、带有 640K 常规内存、外加 256K 扩展内存、Windows 支持的显示适配器、鼠标和一个能保存 Windows 3.1(至少 6M)和开发工具的硬盘。

我的硬件配置是这样：

Gateway 2000 486/33

16M 内存

200M 硬盘

1024K Diamond SpeedStar VGA 适配器

NEC 多重同步的 3D 监视器

单色适配器和监视器

Microsoft 鼠标器。

我使用单色监视器来进行调试。这是在 Windows 3.0 下使用 CodeView 调试器需要双屏而设置的。但是，我发现使用两个监视器更富有成效，尽管 3.1 版的 SDK 包含了为单监视器而设计的 CodeView。

0.3 书中使用的约定

我使用少量来自 Windows 3.1 头文件中新的定义类型(typedef)。比如，用 CALLBACK 代替 FAR PASCAL，以及 HINSTANCE 代替 HANDLE 用于实例变量类型。我认为在文档提供的 Windows 程序中这些类型更为优越。这些类型给 Windows 3.1 SDK 增加了 STRICT 编程选项。我不使用 STRICT 编程选项，因为这可能将另外的重点对象引入本书。

0.4 范例程序

本书所有的实例程序都使用 Microsoft C6.0a 编译。要采用 Microsoft C/C++7.0 编译器时，必须对原代码作少量的修改。这些改变在配套的磁盘上的文件 README.NOW 中概要给出。

第一部分
动态数据交换

第 1 章

动态数据交换的概念

动态数据交换(DDE)是进程间的通讯(interprocess communication)方法。进程间的通讯(IPC)包括进程之间及同步事件之间的数据传递。DDE 使用共享内存来实现进程之间的数据交换及使用协议达到传递数据的同步。Windows 也使用裁剪板、动态链接库(DLL)和对象的链接和嵌入来实现进程间的通讯。

裁剪板包含一些函数,它们使用全局内存在进程间传递数据。这个全局内存和裁剪板程序是不同的,它是从全局内存中显示数据的应用程序。裁剪板典型的应用是传递基于同一时间进程之间的数据。这种操作的一个例子是从电子表格(spread sheet)拷贝一个单元区,并且将这单元复制到文字处理文档中。下次裁剪板访问就更象截取或拷贝不同的数据到裁剪板,而不是复制相同的数据。

DLL 模块包含 3 代码、数据以及多重 Windows 程序可访问的 Windows 资源。多重 Windows 程序可共享一个 DLL 实例,但是,在每一时刻只能有一个 DLL 实例存在。DLL 和应用程序利用全局内存块共享数据。要实现此目的,应用程序调用 DLL 函数分配全局内存块,应用程序和 DLL 就使用这个全局内存块交换数据。当应用程序调用 DLL 中的函数终止时,此全局内存块就释放了。这个全局内存块存在的生命期也就仅仅是应用程序调用 DLL 的时间长度。因此,两个应用程序不能访问由 DLL 分配的同一个全局内存块。

OLE 是一种协议,它使用了 DDE 协议。从根本上来说,OLE 协议是 DDE 执行命令集。这种关系也是本书中在 OLE 之前讨论 DDE 的原因之一。因此,当阅读本书时,切记 DDE 是 OLE 的全集。随后的主题开始 DDE 的讨论:

- ☐ DDE 协议
- ☐ DDE 消息
- ☐ 动态数据交换库

1.1 DDE 协议

DDE 协议是一组规则集,所有的 DDE 应用程序都必须遵循。在编写应用程序时,忽视 DDE 协议是可能的;如果没有遵守 DDE 协议,结果是不可预测的。这倒也是事实:当你编写一个 DDE 应用程序集时,它工作在一个闭合的环境内,而不和另外的 DDE 应用程序发生相互作用。有时要更新不符合协议的应用程序,又要它和其它的 DDE 应用程序通讯,这种更改是困难的,因为其它应用程序接受标准的通讯方式。此外,如果用户通过遵循协议的应用程序访问不遵循 DDE 协议的 DDE 应用程序,这个应用程序不可能工作正确,或者甚至可能挂起系统。换句话说,要想节省时间则头痛的事在后头:编写 DDE 应用程序应严格遵守 DDE 协议。

DDE 协议可应用于两种类型的 DDE 应用程序。第一种是基于消息的 DDE 应用程

序，第二种是动态数据交换管理库(DDEML)应用程序。DDEML 应用程序使用动态链接库(DLL)，此库是和 Windows 3.1 一起发行的。我将利用基于消息的 DDE 来描述 DDE 协议。它比使用 DDEML 更复杂，但是它表示了在外壳下的 DDEML 工作情况。另外，理解基于消息的 DDE 给理解 OLE 提供了良好的基础，因为 OLE 在它的内核中也使用了基于消息的 DDE。

> **注**：由于基于消息的 DDE 和 DDEML 应用程序都使用 DDE 协议，这两种应用程序都可和其它的应用程序通讯。但是，假如任何这些类型的应用程序不遵守 DDE 协议，应用程序之间的通讯都将失败。

1.1.1 会话的同步

DDE 应用程序可分成四种类型：客户、服务器、客户/服务器和监视器。DDE 会话发生在客户应用程序和服务器应用程序之间。客户应用程序从服务器应用程序请求数据或服务。服务器应用程序响应客户应用程序的数据或服务请求。客户/服务器应用程序既是客户应用程序又是服务器应用程序，因此它请求并提供信息。监视器应用程序可解释所有 DDE 应用程序的 DDE 消息但不能执行它们。监视器应用程序为调试目的设置是有用的。

DDE 应用程序可拥有多重进发会话。DDE 协议规定会话中的消息必须同步控制，但是，应用程序可在不同的会话之间异步切换。这样的例子是应用程序拥有两个会话，它们接收四条消息。消息 1、2 和 4 为初次会话，而消息 3 是第二次会话。消息 1、2 和 4 顺序地处理，但是应用程序可在其它任何消息之前或之后处理消息 3，假定应用程序已经接收到了第三个消息。

DDE 应用程序必须唯一地定义所有的会话。客户和服务器应用程序窗口句柄定义一个会话，因此客户应用程序可和多重服务器会话并用窗口句柄对来控制该会话。但是，如果客户应用程序要和单个服务器进行多重对话，那又怎么样？如果仅仅只有一个客户/服务器窗口句柄对，管理这些会话是不可能的。可通过在 DDE 应用程序中为每一个会话产生一个新的窗口来排除这种情况。读者可确信，假如你的程序为每个会话建立一个新窗口，你就拥有唯一的适合每个会话的窗口句柄对。

1.1.2 应用程序名、主题名和项名

DDE 应用程序采用三层识别系统以从其它的 DDE 应用程序来辨认它们本身。应用程序名是在层次结构的顶部。应用程序名附属于服务器应用程序。主题名更深刻地定义了服务器应用程序。服务器应用程序可支持一个或多个主题名。每个主题名可拥有一个或多个项名，它在主题名内定义详细的内容。

一个三层识别系统的例子是服务器应用程序从杂货商场中检索报价。服务器的可执行文件名 QUOTES.EXE。但是，服务器应用程序使用应用程序名 StockQuotes。服务器支持多个商场以及每个商场的多个货架。服务器应用程序的主题名是 NYSE、AMEX 和 NASDAQ。这些主题之中是项名，项名识别货架，诸如 IBM 和 Intel。客户应用程序要得

到 Intel 货架值，就必须在它的 DDE 通讯之中使用应用程序名 StockQuotes、主题名 NASDAQ 和项名 Intel。

1.1.3 会话的初始化

除了用正确的顺序处理消息之外，对应用程序在适当的时候发送或邮寄是重要的。当客户应用程序用应用程序名和主题名发送 WM_DDE_INITIATE 消息初始化会话时，DDE 会话就开始了。要开始会话，也必须有服务器应用程序响应 WM_DDE_INITIATE 消息。但是，为了会话工作，客户应用程序必须提供一些有关本身的信息以及哪个服务器是它所需的。因此，当客户应用程序发送 WM_DDE_INITIATE 消息时，它传递窗口句柄和为会话指定应用程序和主题。

可以认为开始 DDE 会话和开始与一群站在黑暗房子里的人会话相似，如果知道他们的名字，你可以说："George，这是 Joe。你愿意讨论用汇编语言来编制 Windows 程序吗？"很多事都可能发生在此时此刻。George 可能熄灯之前刚好离开房间，因此，就不可能从 George 那得到答复。或许 George 可能认为你有毛病而忽视了你的问题，因为他不可能花费时间去用汇编语言编制 Windows 程序，而让你独自去讨论这个问题。或者人们都头痛，并且他们之中的三人都叫 George。在这种情况下你可能得到他们三个人的答复。此外，这里还有最后一种情况：只有一个 George 在房子里，而他要和你讨论。

当你了解到房子里的人并且也有了主意和他们谈论些什么，这就好了。但是，如果你不了解 他们之间的任何人或他们的兴趣，怎么办？最礼貌的事是说："我是 Joe。有谁想讨论点什么事吗？"那么每个想讨论的人都会说出他们的名字及他们想要和你讨论的问题来答复你，你可以得到和 DDE 应用程序通过发送 WM_DDE_INITIATE 消息及指定 NULL 作为应用程序名和主题名相近的结果。

工作良好的 DDE 应用程序必须处理发送初始化消息所带来的各种可能的结果。最有可能发生的情形是初始化消息得不到任何响应。这时，客户应用程序必须提示用户启动服务器应用程序；否则，客户应用程序不可能继续工作。第二种情形是有多个服务器响应初始化消息。客户应用程序不得不发送 WM_DDE_TERMINATE 消息来终止所有不需要的会话。

1.1.4 会话中的交换

建立了 DDE 会话之后，客户和服务器应用程序开始他们真正的工作：交换数据和执行服务。交换数据似乎适合动态数据交换应用程序，但是服务器应用程序又可完成服务。服务器可能要打开文件或者和别的计算机连接并装入文件。WM_DDE_EXECUTE 消息指定客户应用程序需要服务器应用程序执行服务。WM_DDE_EXECUTE 消息和串一起传递，它包含了服务器应用程序的命令。DDE 协议为串定义了格式；但是，它没有定义串的具体内容。

数据交换可以发生在三种方式中。首先，客户应用程序向服务器应用程序申请数据在同一时间基础上。在这种情况下，客户应用程序发送 WM_DDE_REQUEST 消息，接着服务器应用程序用 WM_DDE_ACK 或 WM_DDE_DATA 消息对请求作出响应。如果服务

器应用程序用 WM_DDE_DATA 消息响应，客户应用程序可访问数据；如果服务器应用程序不能响应请求，它用 WM_DDE_ACK 消息来发送拒绝应答。

第二种交换数据的方法是客户应用程序向服务器应用程序发送数据。当这一事件发生时，客户应用程序使用 WM_DDE_POKE 将数据放置到服务器应用程序，服务器应用程序接收到这个消息时，它用 WM_DDE_ACK 消息来响应客户应用程序，表明它是否接收到数据。

第三种数据交换的方式是在这种情况下发生的：服务器应用程序报告客户应用程序有项改变了值。这个请求可产生两种形式。第一种形式，服务器应用程序发送通知给客户应用程序说明数据项发生了变化，但服务器不发送数据。第二种形式，服务器在每个数据变化时刻都发送数据。客户应用程序请求这项工作通过向服务器应用程序发送 WM_DDE_ADVISE 消息来实现。服务器应用程序用 WM_DDE_ACK 消息响应此消息。然后，每当数据变化，服务器应用程序就发送 WM_DDE_DATA 消息。如果客户应用程序需要的只是通知，WM_DDE_DATA 消息不包括数据。当数据项（如位图）要花时间来完成时后一种方法是有用的。客户应用程序不再需要从服务器接收数据或通知时，它发送 WM_DDE_UNADVISE 消息。

交换数据的方法常常和项链接有关。链接表明数据如何交换。如果客户应用程序申请数据，并且服务器应用程序立即给客户应用程序发送数据，这是冷链接（cold link）。温链接（warm link）是服务器应用程序通知客户应用程序数据项变化了值，但并没有将这个值发送给客户应用程序。热链接（hot link）发生在每当这个值发生变化时服务器应用程序发送新的数据项值给客户应用程序。

1.1.5 会话终止

客户应用程序或者服务器应用程序都能终止会话。服务器应用程序在它要终止会话时，发送 WM_DDE_TERMINATE 消息。当同伴在会话中接收到这一消息时，它也应该邮寄 WM_DDE_TERMINATE 消息。这项工作必须发生在应用程序调用 **PostQuitMessage**( )函数并且离开它的主消息循环体之前。但是，在会话过程中它可发生在任何时刻。

1.1.6 DDE 消息

DDE 协议的核心是 DDE 消息。DDE 协议定义何时、怎样及何地使用 DDE 消息。DDE 消息格式规定 wParam 包含了指向发送窗口的句柄。lParam 参数包含了高位字和低位字，它们确定 DDE 消息所规定的信息。必须使用 **SendMessage**( )发出 WM_DDE_INITIATE 消息，并且用 WM_DDE_ACK 消息应答 WM_DDE_INITIATE 消息。另外，使用 **PostMessage**( )发送其它的所有 DDE 消息。**SendMessage**( )或 **PostMessage**( )函数中的句柄在 DDE 会话存在之后是指向服务器或客户应用程序的句柄，当客户应用程序试图初始化对话时，为－1，**SendMessage**( )函数中的句柄为－1 表示传播消息给系统中所有的非子窗口。

当发送 DDE 消息时，lParam 包含了两个 16-位字：低位字和高位字，MAKELONG 宏将这两个字组合成一个 32-位值形式。

```
SendMessage(hwnd,msg,wPraram,MAKELONG((loworder,highorder));
```

DDE 消息的参数 lParam 包含了指向全局内存块的句柄与原子的组合。原子是整数——16-位字——标识字符串。DDE 消息传递原子而不是串，因为那是说明字符数组的更紧凑的方法。

Windows 支持两种不同的原子。第一种是局部原子，在编制 DDE 应用程序时不涉及它，因为局部原子不能在两个应用程序之中共享。第二种类型是全局原子。DDE 应用程序将全局原子存放在 Windows 的全局原子表中。它存在于 Windows DLL 的共享数据段之内。这就允许所有的 Windows 程序访问原子。

在 DDE 应用程序中常使用四种原子函数：**GlobalAddAtom**( )、**GlobalDeleteAtom**( )、**GlobalFindAtom**( )和 **GlobalGetAtom**( )。正如你所怀疑的那样，**GlobalAddAtom**( )给全局原子表增加原子。初次调用 **GlobalAddAtom**( )插入串于全局原子表之中，并且返回唯一值——此原子——它指明了串。**GlobalAddAtom**( )也为每个原子维护引用计数。引用计数初始化时设置成 1。为相同的串调用 **GlobalAddAtom**( )多次时，每次调用引用计数都增加。**GlobalDeleteAtom**( )则减少引用计数。当引用计数达到 0 时，字符串将从全局原子表中删除。

```
ATOM aAtom;

// string is added to table, reference count = 1,
// aAtom contains a unique value

aAtom = GlobalAddAtom(lpsz);

// reference count = 0,string is removed from table

GlobalDeleteAtom(aAtom);
```

GlobalFindAtom( )返回已有串在全局原子表中的原子。如果串没找到，**GlobalFindAtom**( )返回 0。调用 **GlobalFindAtom**( )不改变原子的引用计数；**GlobalGetAtom**( )返回与原子有关的字符串，它返回拷贝串的字节数，或者在串不存在时为 0。

```
ATOM aAtom;

aAtom = GlobalFindAtom(lpsz);
nLen = GlobalGetAtomName(aAtom, lpBuf, sizeof lpBuf);
```

这里是两个 DDE 应用程序典型的例子，它们使用原子初始化会话。

```
// Client Application

ATOM aApplication, aTopic;

aApplication = GlobalAddAtom("GeorgeServer");
aTopic = GlobalAddAtom("WinMASM");
SendMessage(-1,WM_DDE_INITIATE,hwndClient,
```

```
                    MAKELONG(aApplication, aTopic);
GlobalDeleteAtom(aApplication);
GlobalDeleteAtom(aTopic);


// Server Application

case WM_DDE_INITIATE:
     aApplication = GlobalAddAtom("GeorgeServer");
     aTopic = GlobalAddAtom("WinMASM");
     if ((LOWORD(lParam) == aApplication || !LOWORD(lParam)) &&
         (HIWORD(lParam) == aTopic || !HIWORD(lParam)))
          if(!SendMessage(wParam, WM_DDE_ACK, hwndServer,
                          MAKELONG(aApplication,aTopic))
              {
              GlobalDeleteAtom(aApplication);
              GlobalDeleteAtom(aTopic);
              }
      return FALSE;
```

除原子之外，在 DDE 消息中，DDE 应用程序传递指向全局内存块的句柄。全局内存块包含了 DDE 的结构或数据。DDE 结构定义在 dde.h 中，并且它们对应于 DDE 消息。如要为 DDE 应用程序分配全局内存，使用 GMEM_DDESHARE 参数调用 **GlobalAlloc**( )。

```
hData=GlobalAlloc(GHND | GMEM_DDESHARE,sizeof(datablock));
```

使用这些异常的初始定义，我们可构造和详细地讨论 DDE 消息。表 1.1 总结了关于 DDE 消息的信息。

1. WM_DDE_ACK

WM_DDE_ACK 消息应用程序通知它的会话伙伴已经收到了它的消息。这种情况的例外是 WM_DDE_TERMINATE 消息和有时的 WM_DDE_REQUEST 消息。WM_DDE_ACK 消息有三种格式，一种是响应 WM_DDE_INITIATE 消息，另外是对 WM_DDE_EXECUTE，第三种是针对其它所有 DDE 消息。

当响应 WM_DDE_INITIATE 消息时，服务器应用程序发送 WM_DDE_ACK 消息，带有包含原子的 lParam，此原子标识应用程序和主题名。其它二种 WM_DDE_ACK 消息格式使用 DDEACK 数据结构。

```
typedef struct{
        unsigned bAppReturnCode:8,
              reserved:6,
              fBusy:1,
        fAck:1;
} DDEACK;
```

成员 fAck 被设置为 0 或 1，在这里 1 是应答而 0 是拒绝。如果应用程序正忙并且不能响应 DDE 请求，它设置成员 fBusy 为 1。fBusy 设置只发生在 fAck 为 0 时。bAppReturn-

Code 成员可包含应用程序所指定的返回码。

表 1.1 DDE 消息

DDE 消息	hwnd	wParam	lParam 低位字	lParam 高位字	DEE 结构
WM_DDE_ACK					
响应 INITIATE[1]	hwndClient	hwndServer	aApplication	aTopic	
响应 EXECUTE[2]	hwndClient	hwndServer	wStatus	hCommands	DDEACK
其它	hwndServer 或 hwndClient	hwndServer 或 hwndClient	wStatus	aItem	DDEACK
WM_DDE_ADVISE[2] DDEADVISE	hwndServer	hwndClient	hOptions	aItem	
WM_DDE_DATA[2]	hwndClient	hwndServer	hData	aItem	DDEDATA
WM_DDE_EXECUTE[2]	hwndServer	hwndClient	保留	hCommands	
WM_DDE_INITIATE[1]	-1	hwndClient	aApplication	aTopic	
WM_DDE_POKE[2]	hwndServer	hwndClient	hData	aItem	DDEPOKE
WM_DDE_REQUEST[2]	hwndServer	hwndClient	cfFormat	aItem	
WM_DDE_TERMINATE[2]	hwndServer 或 hwndClient	hwndServer 或 hwndClient	保留	保留	
WM_DDE_UNADVISE[2]	hwndServer	hwndClient	保留	aItem	

1 使用 SendMessage
2 使用 PostMessage

当服务器应用程序响应 WM_DDE_EXECUTE 消息时，它在 lParam 的低位字中使用 DDEACK 数据结构，以及包含命令串的指向全局内存块的句柄，它存在于 lParam 的高位字中。因为 DDEACK 结构正好和 WORD 值大小相同，它可当作数值来传递。其它在 DDE 消息中使用的结构被分配成全局内存块并作为句柄在 lParam 的低位字中传递。最后一个 WM_DDE_ACK 格式也使用在 lParam 的低位字中的 DDEACK 数据结构，但是，lParam 高位字包含了原子，即响应发送项名的原子。

2. WM_DDE_INITIATE 和 WM_DDE_TERMINATE

WM_DDE_INITIATE 和 WM_DDE_TERMINATE 消息标记会话的开始和结束。当客户应用程序要启动会话时，它发送 WM_DDE_INITIATE 消息。WM_DDE_INITIATE 命令的 lParam 的低位字中是指定应用程序名的原子，高位字中是主题名。如果客户应用程序要查询所有的服务器和主题，应用程序名和主题名必须为 NULL。

客户应用程序必须使用 **SendMessage**( )函数——而不是 **PostMessage**( )——来发送 WM_DDE_INITIATE 消息。**SendMessage**( )函数在消息处理结束之前不会返回，但是 **PostMessage**( )函数则立即返回。这个不同对 WM_DDE_INITIATE 消息来说是重要的，因为在 **SendMessage**( )返回之后应用程序调用 **GlobalDeleteAtom**( )来删除应用程序名

和主题名。**SendMessage**( )的使用确保应用程序和主题原子和它们的引用计数是有效的。**PostMessage**( )的使用可能允许客户应用程序在服务器处理 WM_DDE_INITIATE 消息之前删除原子。

如果客户应用程序和服务器存在会话，函数 **SendMessage**( )中的句柄参数是指向服务器窗口的句柄。否则，该句柄参数应该为－1，它传播 WM_DDE_INITIATE 消息给系统中所有的非子窗口。

当服务器应用程序接收 WM_DDE_INITIATE 消息时，它必须首先建立它支持的应用程序名和主题名的原子。如果原子和 WM_DDE_INITIATE 消息中的 lParam 具有相同的值，服务器必须用消息 WM_DDE_ACK 响应，它的 lParam 中是服务器应用程序为应用程序建立的原子和主题名。如果 WM_DDE_INITIATE 消息的应用程序或主题名包含 NULL 的原了，服务器必须 使用正确的 WM_DDE_ACK 消息数目作出响应。如果 **SendMessage**( )失败了，服务器必须在发送 WM_DDE_ACK 消息之前删除已建立的原子。表 1.2 给出了服务器应用程序怎样响应 WM_DDE_INITIATE 消息。

表 1.2　响应 WM_DDE_INITIATE

lParam	服务器响应
低位字＝NULL 高位字＝NULL	为应用程序名和主题名建立原子，给服务器支持的每个主题名发送 WM_DDE_ACK 消息
低位字＝NULL 高位字＝aTopic	为应用程序名和主题名建立原子；如果 aTopic 和服务器的主题名之一匹配，给此主题发送一个 WM_DDE_ACK 消息。
低位字＝aApplication 高位字＝NULL	为应用程序名和主题名建立原子；如果 aApplication 和服务器的应用程序名匹配，给服务器支持的每个主题名发送 WM_DDE_ACK 消息。
低位字＝aApplication 高位字＝aTopic	为应用程序名和主题名建立原子；如果 aApplication 和服务器的应用程序名匹配，并且 aTopic 和服务器的主题名之一匹配，给那个应用程序/主题名对发送一个 WM_DDE_ACK 消息。

当服务器成功地发送了 WM_DDE_ACK 消息，会话就开始。这给客户应用程序带来一些复杂性，如果服务器为多重主题发送 WM_DDE_ACK 消息，或者多重服务器响应。客户应用程序则必须确定保留哪个会话和哪些必须终止，哪个给我们带入 WM_DDE_TERMINATE 消息。WM_DDE_TERMINATE 消息终止会话，这个消息的协议是简单的。发送应用程序邮寄 WM_DDE_TERMINATE 消息，接着接收应用程序邮寄另一个 WM_DDE_TERMINATE 消息给发送应用程序。在这个消息中只是句柄和 wParam 发生改变。

3. WM_DDE_POKE

客户应用程序使用消息 WM_DDE_POKE 更新服务器应用程序中的数据，该消息使用一个 DDEPOKE 结构和指定服务器应用程序需要更新项目名的原子。lParam 的低位字包含了指向 DDEPOKE 结构的句柄。lParam 的高位字包含了项名的原子。DDEPOKE 的定义如下：

```
typedef struct {
        unsigned unused:13,
            fRelease:1,
            fReserved:2;
        int cfFormat;
        BYTE Value[1];
} DDEPOKE;
```

客户应用程序分配大小为 DDEPOKE 结构及存放数据的全局内存块。存放数据的大小即是在所选用的裁剪板格式中补偿数据的大小。结构成员 cfFormat 的值是数据的裁剪板格式，换句话说，即是 CF_TEXT 或 CF_BITMAP。fRelease 成员包含 0 或者 1。如果 fRelease 被置成 0，服务器应用程序不能释放全局内存块，反之，服务器应用程序必须释放全局内存块。如果服务器应用程序没有释放全局内存块，删除它则是客户应用程序的责任，这发生在从服务器应用程序接收到 WM_DDE_ACK 消息之后。

注：在 SDK 3.0 以前的 Windows 版本中，DDEPOKE 定义得不正确。以后的 SDK 版本包含了下面的结构，它允许为使用不正确定义的结构的应用程序编程。

```
typedef struc {
    unsigned unused:12,
        fAck:1,
        fRelease:1,
        fReserved:1
        rAckReq:1,
    int cfFormat;
    BYTE rgb[1];
} DDEUP;
```

当服务器应用程序接收 WM_DDE_POKE 消息时，它必须尝试去设置消息中原子所指定的项。如果服务器成功了，它发送 WM_DDE_ACK 消息，使用 DDEACK 结构成员 fAck 设置成 TRUE。如果服务器是不成功的或者没有识别出原子，它发送 fAck 设置成 FALSE 的 WM_DDE_ACK 消息。

这里是客户应用程序发送 WM_DDE_POKE 消息的原代码片段：

```
ATOM aItem;
HANDLE hData;
char ach[STRINGSIZE];
LPDDEPOKE lpPokeData;

hData = GlobalAlloc(GHND | GMEM_DDESHARE,
                    sizeof(DDEPOKE) + STRINGSIZE);
if(!hData)
    return FALSE;
```

```
lpPokeData = (DDEPOKE FAR *) GlobalLock(hData);
if(!lpPokeData)
     {
     GlobalFree(hData);
     return FALSE;
     }

lstrcpy((LSPTR)&lpPokeData->Value[0], ach);
lpPokeData->cfFormat = CF_TEXT;
lpPokeData->fRelease = FALSE;
GlobalUnlock(hData);
aItem = GlobalAddAtom("Poked Data");

if(!PostMessage(hwndServer, WM_DDE_POKE, hwndClient,
               MAKELONG(hData, aItem)))
     {
     GlobalFree(hData);
     GlobalDeleteAtom(aItem);
     return FALSE;
     }
```

在服务器一方，窗口消息循环看起来应该是这样：

```
case WM_DDE_POKE:
     hData = LOWORD(lParam);
     aItem = HIWORD(lParam);

     lpPokeData = (DDEPOKE FAR *)GlobalLock(hData);
     if(!lpPokeData)
          { /* Negative ack */
          PostMessage(hwndClient, WM_DDE_ACK,hwndServer,
                      MAKELONG(0, aItem));
          return FALSE;
          }
      else
          {
          if(lpPokeData->cfFormat == CF_TEXT)
              strcpy(szItem, lpPokeData->Value);
          else
              { /* negative ack */
              PostMessage(hwndClient, hwndServer, WM_DDE_ACK,
                          MAKELONG(0,aItem);
              return FALSE;
              }

          GlobalUnlock(hData);
```

```
/* positive ack */

PostMessage(hwndClient,WM_DDE_ACK, hwndServer,
            MAKELONG(0x8000, aItem));
}
```

4. WM_DDE_EXECUTE

WM_DDE_EXECUTE 消息允许客户应用程序邮寄命令到服务器应用程序。客户应用程序将命令串存放在全局内存块中，并在 lParam 高位字传递句柄给全局内存块。命令串必须以 NULL 结束，并且符合严格的句法。命令串包含一个或多个操作码串，它们由方括号定界。每个操作码可选择地带有参数。操作码的参数用括号定界，多重参数用逗号隔开。这里是几个例子：

[*command*1]

[*command*2(*parameter*1)]

[*command*3(*parameter*1, *parameter*2, *parameter*3)]

[*command*1][*command*3(*parameter*1, *parameter*2, *parameter*3)]

当服务器应用程序接收 WM_DDE_EXECUTE 消息时，它必须用 WM_DDE_ACK 消息响应。如果服务器成功地执行命令，DDEACK 结构成员 fAck 被置成 TRUE。如果服务器执行命令不成功或者不能辨认这些命令，服务器邮寄 WM_DDE_ACK 时将 fAck 设置为 FALSE。服务器也将命令的句柄回送给客户应用程序，将句柄放置在 lParam 的高位字中。当客户应用程序接收到 WM_DDE_ACK 消息时，它从全局内存中删除这一命令串。

下面的代码段表示了 WM_DDE_EXECUTE 消息的客户方面的情况：

```
char * szExecute;
HANDLE hExecute;
LPSTR lpExecute;

if(! hExecute = GlobalAlloc(GMEM_MOVEABLE | GMEM_DDESHARE,
                            (DWORD)lstrlen(szExecute) + 1)))
    return FALSE;
if(! (lpExecute = GlobalLock(hExecute)))
    {
    GlobalFree(hExecute);
    return FALSE;
    }

lstrcpy(lpExecute, szExecute);
GlobalUnlock(hExecute);
```