第一部分　Windows 程序设计模式

第 1 章　Windows 概念介绍

1.1　什么是 Windows

Microsoft Windows 3.1 是一个运行在 MS-DOS 或 PC-DOS 3.1 以上版本下的图形环境。在标准模式下，Windows 需要一台配备 80286(或更高)处理器，640K 常规内存，及 256K 扩展内存，6M 自由磁盘空间(建议为 9M)，至少一个软盘驱动器的微机。

在 386 增强模式下运行 Windows，要求一个 80386 处理器(或更高)，640K 常规内存，及 1024K 扩展内存，8M 自由磁盘空间(建议为 10M)，至少一个软盘驱动器。在 386 增强模式下，各方面的需求更高。

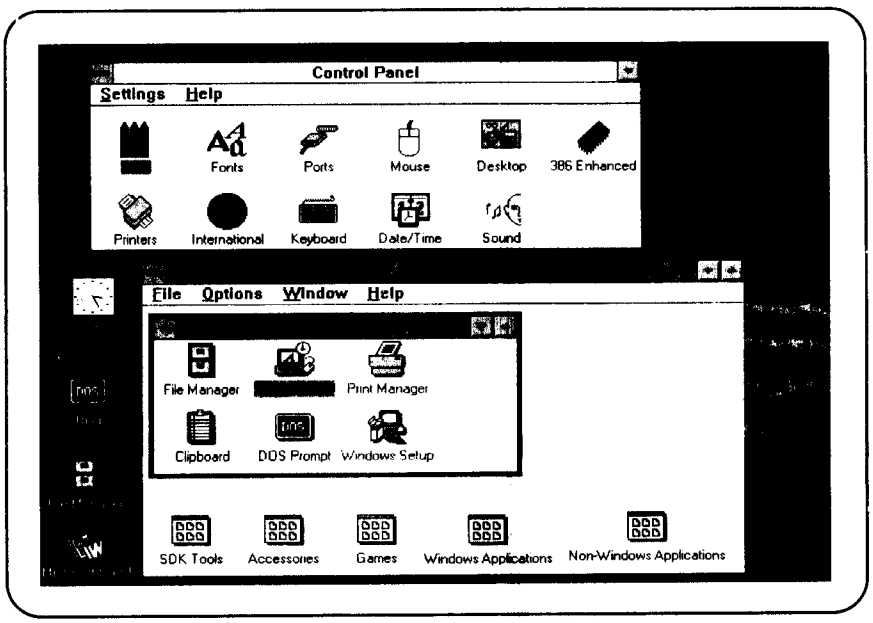

图 1.1　Windows 桌面图示

Windows 提供一个多任务图形窗口环境，可以同时运行多个程序。每个程序的全部输出显示在屏幕上一个矩形区域内，每个矩形区域称作一个窗口。整个屏幕形象化为“桌面”。用户可以像在一张真正的书桌上放置纸张一样在桌面上排放窗口。例如，可以在相邻的窗口内运行两个程序，而将其它程序暂时搁置起来，如图 1.1 所示。图 1.1 中的所有程序正在运行，包括那些以图符搁置起来的程序。大多数程序在用户未与之交互时是被搁置起来的，但有些程序也可能以图符状态运行。例如，一个电子表格应用程序在图符态时仍要继续计算表格中的值。Windows 的时钟程序即使成为图符也能继续修改钟面。

对程序设计人员来说，Windows 为开发易用的、一致的用户界面提供了广泛的支持，是一个丰富的程序设计环境。Windows 为开发者提供了菜单、对话框、列表框、滚动条、按钮和其它的用户界面元素。Windows 提供了设备无关的图形，使用户可以不必详细了解运行所依赖的硬件平台便可编写程序。在微机上这种设备无关扩展成基本的硬件设备。除了图形显示之外，程序设计人员还可以以一种设备无关的方式访问键盘、鼠标、打印机、系统时钟及串行通信口。

1.2 用户交互的历史回顾

最初，计算机利用触发开关和灯实现用户交互。要想将一个程序装入计算机，用户需要用一种适当的方式设置很多触发开关，然后按一个按钮将一个字符、字节、或字装入计算机的内存。无须多言，这种方法速度很慢而且极易出错。计算机的输出在灯板上以二进制显示。

随着打孔纸带、打孔纸卡和行打印机的出现，用户交互的技术有了重大的改进。利用这种改进，当纸带或纸卡被正确地打孔之后，可以保证较准确地装入程序并且产生可读形式的输出。除了类似大量的打孔纸卡或打孔纸在打印机里发生阻塞这样的事件之外，与计算机的交互不再象以前那样乏味及容易出错。

但是，与计算机的交互仍然很困难。记录程序的大量纸卡通常要保存到积累了足够的任务可以分批地运行为止。实现在计算机上一天多次运行某一项任务是一件值得庆贺的事情。电传打字电报机(TTYs)以及后来终端(当时被称作玻璃 TTYs)的出现使用户首次可以与计算机直接交互。

最初，显示终端只是被看作是打印终端，用户只能看见已打印过的最后 24 行左右的内容。事实上，IBM 微机及其兼容机上的 DOS 操作系统仍然以这种方式处理显示。通常当用户请求目录列表时，在终端上一闪而过，用户看到的只是长长的目录列表的最后几行。

随着对良好的用户界面的不断需求，计算机更多地被用作一种随机访问设备，而不是一种顺序的行式设备例如打印终端。程序开始在屏幕上显示表格并且从输入区域接受输入。但程序员仍在多方面受到限制。通常假定一个显示屏由 24 行 80 列组成。这个假定至今仍然保留。

第一台 IBM 微机的文本显示是 25 行×80 列。要求 24 行×80 列显示的程序可以很容易地在这种微机上运行。另外，第 25 行可用于该程序的 PC 专用版本。诸如彩色图形适配器(CGA)，增强型图形适配器(EGA)，VGA，超 VGA 及 8514 适配器等可以按不同的

行列数显示图形及文本。较早的一些程序在利用这些显示器时运行很奇特，通常只利用显示器上的头 25 行和左边 80 列。对程序设计人员来说，在使用这些显示器的图形方式时更难于处理。在物理上不存在对图形的内部支持，因此每个程序设计人员必须重新构造图形方式。当一个程序可以在某一个图形显示器上正确运行之后，还必须努力寻找一种新的方法使该程序可以在另一个不同的显示器上显示。

Microsoft Windows 作为解决这些问题的一种方案于 1983 年 11 月公布。1985 年 11 月公布 Windows 版本 1.01。接下来的两年里对 Windows 作了许多小的修订。在这段时间里，几个其它的软件制造商发行了几个以图形或窗口环境运行程序的软件产品。尽管其中一些产品因为不受欢迎而被淘汰，但其它的产品却继续与 Microsoft Windows 共存。

在 1987 年 11 月公布的 Windows 2.0 中，Windows 应用程序开始从以前 1.x 版本的“铺瓦式”窗口向以后的 Windows 版本中的“重叠式”窗口转换。对 Windows 2.0 中在外观和用户界面的交互上的以及其它的一些修改，是为了与 OS/2 显示管理程序的用户界面相一致。

1990 年 5 月 Windows 3.0 出现，它的出现为用户和开发人员都创造了许多对窗口环境很重要的条件。从最开始的显示，用户注意到桌面的外观有了一个明显的改进。许多图符和按钮具有三维的外观。在菜单和对话框中使用了按比例分隔的字体。Windows 3.0 不再受 640K 内存的限制而可以利用扩展内存。在 Intel 80286 及更高的处理器上是 1M 字节以上的可用内存。多个、复杂的应用程序现在可以同时在一个彩色图形显示器上运行。用户界面的发展经历了一条漫长的道路。

Window 3.1——本书的主题——公布于 1992 年 4 月，进一步改进了 Microsoft Windows 的环境。现在 Windows 3.1 更易于用户安装和配置。Windows 3.1 包含一个指导部分，可指导新用户学会如何从头开始高效使用 Windows。

常用的对话框，例如 Open，Save As 和 Print，现在在整个系统中统一起来。用户可以在自己的 Windows 应用程序中使用这些相同的常用对话框。当用户利用这些常用的对话框来实现常用的功能时，可以使自己的应用程序被新的用户熟悉。用户找到应用程序，利用相同的用户界面来实现同另一个应用程序相同的功能(保存一个文件，打印一个文档，等等)。

Windows 3.1 文件管理程序的速度较快，可以使用户从文件管理程序中将文件拖动到其它的应用程序中。用户可以使自己的 Windows 3.1 应用程序能够拖动和安置，这样在适当时候可以将用户拖动来的文件进行安置。

Windows 3.1 引入了 TrueType 可变比例字体。利用 TrueType 字体可以使用户程序在屏幕上和任意类型的打印机上显示或打印大字体。当用户程序利用 TrueType 字体时，屏幕上的显示非常接近于打印出来的文本外观。TrueType 字体是可变比例的，意味着用户可以选择任意的字体大小而不必为每种字体的各种大小各自存储位图文件。Windows 包含下列的 TrueType 字体(常规体、黑体、斜体和粗斜体)的标准集：

- Times New Roman
- Arial
- Courier New
- Symbol

用户和程序设计人员将会欣赏 Windows 3.1 增加的可靠性——尽管原因各不相同。Windows 3.1 非常强壮。出错的应用程序将很难使 Windows 3.1 的系统受到冲击。当一个出错的应用程序导致了一个致命的 UAE(不可恢复的程序错误)时,用户通常可以利用 Windows 3.1 的 Application Reboot(应用程序重新登录)的功能将出错的程序关闭,而不用重新启动 Windows 或重新登录整个计算机。这个功能节省了许多调试 Windows 应用程序的时间。

Windows 3.1 也提供了参数合法性检查。当用户从一个 Windows 应用程序中调用 Windows 时,常常必须传递许多的参数。先前的 Windows 版本都认为所传递的参数是合法的,应用程序可能传递非法的参数(例如一个非法的窗口句柄),Windows 在使用这些参数的过程中有可能破坏了内存。Windows 3.1 比较可靠,它在使用参数之前先检验应用程序所传递的参数的合法性。当用户指定了非法的参数时,Windows 3.1 将向用户的 3.1 程序返回一个错误。

在 Windows 3.1 下编写利用动态数据交换(DDE)的程序更为容易。Windows 3.1 包含 DDE 管理程序库(DDEML),它为复杂的 DDE 通信任务提供一个高层的过程性的接口。Windows 3.1 还包含对对象链接和嵌入(OLE)的支持。在用户程序中适当的地方利用 OLE,可以使用户将自己的程序和其它程序连成一个整体。

本书的主题是开发 Windows 3.1 应用程序。Windows 应用程序可以概括地划分为三类——一类必须在 Windows 3.0 下以实模式运行的程序,一类必须在 Windows 3.0 版本或更高版本下但以保护模式运行的程序,以及一类必须在 Windows 3.1 版本或更高版本下运行的程序。

由于实模式已不用,事实上在 Windows 3.1 中已经不再出现,因此不再说明对实模式 Windows 应用程序所加的许多限制。本书在描述用户可以在原有的应用程序中找到而在保护模式应用程序中不需要的代码时会偶尔提到实模式 Windows 应用程序。最常见的例子是对不可丢弃的动态申请内存块的连续加锁和解锁。

多数当前的 Windows 应用程序只在保护模式下运行,即便在运行 Windows 3.0 时也是如此。只有保护模式才可以为应用程序的运行提供足够的内存。所有将来的 Windows 应用程序将只在保护模式下运行,因为在 Windows 中不再包括实模式。

本书可指导用户利用 Windows 3.0 和 3.1 的技巧进行 Windows 保护模式程序设计。本书中的所有例程都可以在 Windows 3.0 和 3.1 两种环境下正确运行。调用了 Windows 3.1 特有函数的例程显然不能在不具备该函数的 Windows 3.0 下良好运行。同样,依赖于 Windows 3.1 某个特有功能例如拖动和安置的应用程序,在 Windows 3.0 下运行时也将不具备这种功能。例如,Windows 3.1 文件管理程序知道如何在一个 Windows 3.1 可安置的应用程序上进行拖动和安置,但是 Windows 3.0 文件管理程序却不能。

1.3 Windows 程序和典型的 DOS 程序间的差别

在一个 Windows 程序和一个典型的 DOS 程序之间存在着很多差别。Windows 应用程序必须共享系统资源。DOS 应用程序一般不需要共享。Windows 应用程序产生图形输出。DOS 应用程序一般产生文本显示。Windows 应用程序接收输入和管理内存的方式都

与 DOS 应用程序很不相同。Windows 应用程序不要求细节的硬件知识，但 DOS 应用程序通常只能在一种类型的设备上运行。

1.3.1 资源共享

从一开始 Windows 应用程序的设计就共享它们所运行的系统的资源。诸如主存，处理器，显示器，键盘，硬盘，和软盘驱动器等等的资源被所有运行在 Windows 下的应用程序共享。为了实现这种共享，一个 Windows 应用程序必须只能通过 Windows 的应用程序接口(API)与计算机的资源进行交互。只有通过 Windows API 来访问资源才能使 Windows 控制这些资源。这种限制可使多个程序能同时在 Windows 环境下运行。

相比之下，DOS 应用程序一般希望系统的所有资源都可用于该程序。一个 DOS 应用程序可以申请所有可用的内存而不必考虑其它的应用程序。它可以直接访问串行和并行端口。DOS 应用程序不需要设计成共享系统资源。

1.3.2 图形用户界面

Windows 应用程序以图形用户界面运行。由于多个应用程序可以同时运行，因此每个应用程序都通过一个“窗口”来访问图形显示器。应用程序通过这个窗口与用户进行交互。Windows 应用程序不限制每个程序只有一个窗口。一个 Windows 应用程序在图形显示器上可以同时有一个以上的窗口。

Windows 还提供了一个丰富的输入控制集合，作为图形用户界面的组成部分。其中包括有对话框、组合框、各种类型的按钮、菜单和滚动条。图 1.2 显示了一个包含组合框、按钮、圆按钮、一个滚动条和三个定制(又称自画式)控制的对话框。屏幕上显示的对话框可用来向用户显示信息和请求输入。换句话说，它们为程序设计人员提供了一种“包装”与用户之间的对话的方式。按钮、检取框和圆按钮可以供用户选择程序选项的真/假、开/关、使能/禁止的类型，不需要在用户和应用程序之间进行传统的问答对话。必要时可以显示菜单，使用户能够确定程序行为。滚动条可使用户向应用程序指示当前窗口中显示的是一个文件、文档或其它输出显示的那一部分。

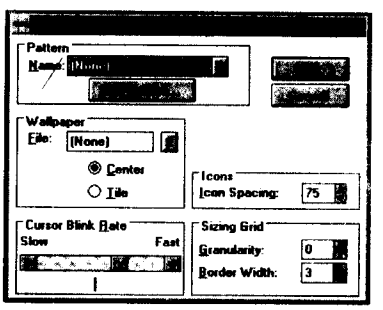

图 1.2 Windows 部分输入控制

1.3.3 输入设备

在一个新的 Windows 环境下的程序，用户输入是设计人员所遇到的最大差异。Windows 控制输入设备并在多个并行的应用程序相互之间根据请求从设备中分配输入。Windows 应用程序不能根据需要读输入设备。Windows 应用程序的构造是在用户产生输入时接收输入，它不能在需要输入时请求输入。

Windows 应用程序被告知用户何时按下或释放一个键。当鼠标光标在应用程序的窗口内移动时，一个消息流发送给该应用程序，随时向应用程序通知鼠标光标的当前位置。当按下或释放一个鼠标按钮时，Windows 将该事件通知应用程序并告知它在该事件发生

时，鼠标光标正点在窗口中的位置。举几个例子，当用户选择菜单、检取一个检取框、按下一个按钮或滚动一个滚动条时，应有程序接收一个连续的消息流以准确地得知用户正在进行的动作。

在这一点上，Windows 应用程序与标准 DOS 应用程序很不相同。Windows 环境要求应用程序的构造与标准 DOS 程序的构造有很大差异。由于 Windows 所提供的日益丰富的环境，Windows 应用程序的编写未必比标准 DOS 程序更困难。在 Windows 下，各种结构是陌生的，而且旧的习惯和方法也不再可行，但是在掌握了 Windows 程序设计中所涉及到的新概念之后，用户就能够编写出比许多基于文本的 DOS 应用程序更容易使用的应用程序。

1.3.4 内存管理

内存是在 Windows 环境下所有应用程序共享的另一资源。与标准 C 程序所不同的是，标准 C 程序通常认为整个计算机的内存都可为自己所用，而多个 Windows 应用程序之间却必须共享可用的内存。Windows 应用程序不需要在初始时保存应用程序所申请的全部内存并将它保持到程序终止时，只是在需要存储时进行申请，并且一旦不再需要就马上释放掉。

Windows 提供了两种内存资源：局部内存，一般用于小容量申请；全局内存，一般用于大容量申请。当用户从这些资源中申请一块内存时，可能有一些特定的限制。例如，用户可能需要将一个内存块从一个应用程序传递给另一个应用程序，这时除非用户从全局内存池中申请存储并在申请时通知 Windows，否则将不能实现。

Windows 努力最有效地利用可用的内存。在 Windows 内存管理中，Windows 可以把一个内存块移动到内存中的不同的位置上。实际上，它甚至可以完全丢弃一个内存块。这就要求 Windows 应用程序谨慎地协调与 Windows 之间的动态内存利用。

程序在需要一块内存，以一种特有方式使用时，必须告知 Windows。旧版的 Windows 要求用户在访问此内存块之前随时询问 Windows 该内存块的当前位置，并在用户完成访问该内存块时通知 Windows。

随着只以保护模式运行的应用程序的出现，这种要求就不再是必要的。但在使用可丢弃的内存块时，用户仍必须遵循这条协议，以使 Windows 知道它什么时候能够和不能够丢弃内存。但是，在更一般的情况下，只以保护模式运行的应用程序可以一次获得一个动态存储块的地址，程序中就使用这个地址访问内存块。最后，当此内存块不再需要时，用户必须告知 Windows，以便它能被回收成为可用的动态存储。

Windows 下的内存管理不是标准 C 程序所要求的简单模式。正是由于这些复杂性，多个应用程序可以同时运行，而在以前只能一次运行一个程序。但是，Windows 应用程序中不完善的内存管理可能阻止其它所有的 Windows 应用程序及 Windows 本身的运行。在全书中讨论正确的 Windows 资源管理(包括内存)。每当一个常见的资源管理问题出现时，将在书中指出并给出正确的处理方法。

1.3.5 设备无关图形

最后，Windows 提供设备无关图形操作。只要一个 Windows 程序可以在图形显示器

上画线和圆，那么同样的程序也能在被 Windows 支持的所有图形显示器上正确地画出这些图形。事实上，这种设备无关性并不只限于显示器。在显示器上产生一个饼图的相同的图形操作也能在一个点阵打印机、激光打印机、绘图仪，或 Windows 支持的其它输出设备上画出饼图来。

1.4 Windows 程序设计模式

Windows 程序设计素有难学之说。这种说法部分是由于丰富的程序设计环境。大量的 Windows API 函数很容易使人不知所措。学习 Windows 程序设计时存在两大障碍。Windows 程序要求一种与传统的 DOS 程序所用不同问题的概念模式。另外，基于这种概念模式编写程序所使用的工具并不直接支持所用的概念。

1.4.1 概念模式

Windows 应用程序是一些处理各种形式的用户输入并向用户提供精致的图形输出的程序。应用程序显示的窗口必须响应用户的动作。菜单必须弹出并使能选择。按钮必须按下并在释放时跳回。检取框必须检取和不检取自己。一般地，Windows 中的对象必须响应操作。这引申出关于 Windows 应用程序是一个对象集合的观点。这恰恰就是面向对象式程序设计所提倡的观点。

面向对象式程序设计的概念和图形程序设计环境一起发源于 Xerox Palo Alto 研究中心(PARC)。在面向对象式程序设计中，程序设计人员创建抽象的数据类型。这些抽象的数据类型(通常称作对象)由一个数据结构和一些使用这个数据结构的相关函数(通常称作方法)组成。一般地，一个对象的数据结构在它提供给外部的方法之外是完全未知的。这种技术称作数据封装，它可以使该对象的内部结构根据需要而改变。只要由对象的方法提供的外部界面保持不变，那么程序的其余部分不需要知道该对象内部的状况或如何实现它的功能。

这些概念很自然地应用于 Windows 应用程序。在 Windows 中有许多对象。Windows 拥有画笔——一类具有宽度、颜色、虚线风格的对象，常用于画线。Windows 拥有画刷——一类具有颜色和某种模式的对象，用于绘制区域。Windows 还拥有菜单、对话框和其它许多种类的对象。但第一个和最基本的对象是“窗口”。

显示在屏幕上的窗口必定拥有与其相关联的数据。毕竟，窗口有一个背景颜色、一个标题，还可能有一个菜单和很多其它属性。图 1.3 显示了一个窗口的主要属性。如图所示，Windows 隐藏了这些属性的实现。改变属性的唯一方法是借助于 Windows 提供的外部界面。当用户在光标指向窗口内某处时按下一个鼠标按钮，会发生什么事呢？面向对象的观点认为向该窗口发送此事件消息并由它决定。

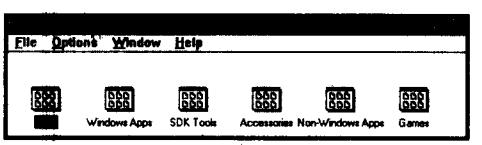

图 1.3 一个窗口的主要属性

面向对象式程序设计还提供多态性——对象具有不同的行为的特性。在图 1.3 的例子中，某个窗口可能以某种方式响应一个按下的鼠标按钮，而另一个窗口的响应方式可能

很不相同。事实上，它可能完全忽略该事件。

面向对象式程序设计还提供继承性。在 Windows 程序设计中，创建一个“正如那边的窗口”的窗口通常是很方便的，除了增加(或减少)的功能。用户可以利用已有的窗口做为模板来设计新的窗口，而不用每次都创建一个新窗口。Windows 支持这种特性。

开发 Windows 应用程序所用的工具并不直接支持这种概念模式。也就是说，用户不能将一种有关系统的基于对象的观点直接映射成一条或数条用 C，Pascal，或汇编语言编写的特殊程序语句。例如，在 C 语言中没有用来说明面向对象式程序设计中所用的对象意义的语句。实际上，Windows 程序设计人员在实现一个可运行的 Windows 应用程序时必须跳过两个障碍。

首先，同所有程序设计一样，用户必须提出一个特定的问题并对它进行抽象，从一个特定的情形转化成所希望的一般行为。Windows 程序设计(面向对象方法)中采用的概念模式，最初对多数程序设计人员都是生疏的。在 Windows 环境下不能采用已试验过的并且正确的连续、过程式的方法。熟悉构造一个以这种新的方式完成程序设计任务的方案需要一定的时间。

在跳过这第一个障碍之后，另一个障碍正在前面等你。C 语言是本书中所采用的程序设计语言及 Microsoft Windows 的软件开发包所支持的语言，它并没有提供直接表示这些面向对象式概念的支持。因此，实际上，用户必须用一种正确模拟面向对象式的方案设计的非面向对象语言来编写程序。

在很多方面，用 FORTRAN 代替 COBOL 语言编写报表程序是一个类似的问题。这是可以实现的，但这样做所要付出的大量努力花费在用一种不是特别适用于要解决的问题的语言来分析问题。当然，反过来也是如此。在 FORTRAN 适用时不应该用 COBOL 语言来编写一个计算性程序。

那么，Windows 是如何实现这部分例程(事件消息)的，作为程序设计人员又如何在受到不特别适用于所做任务的语言的限制情况下编写一个正确响应这些消息的程序呢？答案在于窗口，相关的窗口函数及消息。

1.4.2 窗口及其相关的窗口函数

正如先前提到的，当程序创建一个窗口时，这个窗口具有某些特征——它在屏幕上的位置，一个标题，大小，一个菜单，等等。但是它也具有其它的特征——如何响应各种事件消息。所有的窗口都有一个相关的函数，称作窗口函数，它决定该窗口如何响应一个事件消息。消息的含义即事件的通知。例如，按钮和滚动条控制，它们包含在窗口内但本身也是窗口。因此，它们也具有一个相关的窗口函数，控制按钮或滚动条控制如何响应。滚动条“窗口”通过显式地改变窗口的外观并向它的父窗口发送消息，通知父窗口滚动的请求，来响应一次鼠标点取。

我们已经说过，当一个可能影响某窗口的事件发生时，Windows 将通知该窗口。它的描述通常是各不相同的。也就是说，向窗口发送消息还是窗口接受一条通知该事件的消息，都是取决于用户的观点。实际上，Windows 调用该窗口的窗口函数，利用函数调用中的参数，描述已发生的事件来传递信息。因此，窗口，相关的窗口函数及窗口接收的消息是互相关联的。

什么样的消息发送给窗口？它们来自何处？先来回答后一个问题，它们来源于许多不同的资源。虽然多数消息来源于 Windows 操作环境，但一个应用程序可以从一个窗口向另一个窗口发送消息。窗口也可以向自身发送消息。Windows 应用程序利用消息来通知自己将来要做某事。以后，当消息传到时，应用程序将执行预期的动作。应用程序可以向另一个完全不同的应用程序发送消息。通常一条来自 Windows 的消息会产生一批派生的消息。Windows 也发送一些向窗口请求信息的消息。由于以适当的顺序向适当的目的地发送消息是 Windows 操作的关键，因此我们来看看这是如何实现的。

1.4.3 窗口队列和消息循环

在 Windows 中许多消息来自设备。按下和释放键盘上的键，产生由键盘设备驱动器处理的中断。移动鼠标和点取鼠标按钮，产生由鼠标设备驱动器处理的中断。这些设备驱动器调用 Windows 将这些硬件事件翻译成消息。产生的消息被放进 Windows 系统队列中。

在 Windows 中有两种类型的队列——一个系统队列和应用程序队列。只有一个系统队列。转化成消息的硬件事件被放进系统队列。每个运行的 Windows 应用程序都有自己唯一的应用程序队列。Windows 将系统队列中的消息传送给相应的应用程序队列。程序的应用程序队列包容了该程序所有窗口的全部消息。

Windows 利用系统队列实现共享资源（例如鼠标和键盘）的共享。当一个事件发生时，一条消息被放进系统队列中。然后 Windows 决定哪个应用程序队列应该接收这条消息。这一点的实现方式可以根据事件而变化。Windows 利用输入焦点的概念来决定哪个应用程序队列应该接收这条消息。输入焦点是一种一次仅有一个窗口处理的属性。拥有输入焦点的窗口是所有键盘输入的焦点。键盘消息从系统队列移进拥有输入焦点的窗口所在程序的应用程序队列中。

当输入焦点在窗口间移动时，Windows 将键盘消息从系统队列移到相应的应用程序队列。这是依据消息接着消息的原理实现的，因为某些击键是从一个窗口到另一个窗口改变输入焦点的请求。系统队列中后来的键盘消息可能会去向不同的应用程序队列。

鼠标消息的处理有点不同。鼠标是另一个共享资源。鼠标消息通常发送给鼠标指示器所在的窗口。当多个窗口重叠时，顶层的一个窗口即正被显示的窗口——接收这条鼠标消息。对这条规则的一个例外是涉及到捕获鼠标。当某个 Windows 应用程序捕获鼠标时（由一个 Windows 函数调用实现），Windows 将所有后来的鼠标消息从系统队列移到捕获程序的应用程序队列，无论鼠标正指向屏幕上何处。应用程序最终必须释放被捕获的鼠标并支持其它应用程序使用。

硬件计时器产生由计时器设备驱动器处理的周期性中断。同键盘和鼠标设备驱动器一样，计时器设备驱动器调用 Windows，将此硬件事件翻译成一条消息。与键盘和鼠标消息不同的是，产生的计时器消息被直接放进程序的应用程序队列中。由于多个程序可能请求计时器消息被周期性地向它们的窗口发送一个或多个消息，因此一个计时器中断可以使 Windows 将计时器消息放进多个应用程序队列。实际上，Windows 使硬件计时器成为一个可共享的设备。

程序调用某些 Windows 函数后将其它的消息放进程序的应用程序队列。例如，程序

可以调用 Windows，通知它某个窗口的某个区域不再是最新的。Windows 将一条消息放进该程序的应用程序队列，该消息最终使该程序重画此窗口的这部分无效区域。

既然程序的应用程序队列填满消息，那么程序是如何从该队列中获得消息并将它传送给相应的窗口函数的？用户编写一小段称作消息循环的代码，这段消息循环从应用程序队列中检索输入消息并将它们发送给相应的窗口函数。消息循环连续地检索和发送消息直到它检索到一条特定的标识程序终止的消息时。消息循环是一个 Windows 应用程序的主体。Windows 应用程序启动后，重复地执行消息循环直到被指示停止，然后终止运行。程序调用 Windows 的 GetMessage 函数从它的应用程序队列中检索一条消息。Windows 将消息从此队列中移进该程序内的某个数据区中。

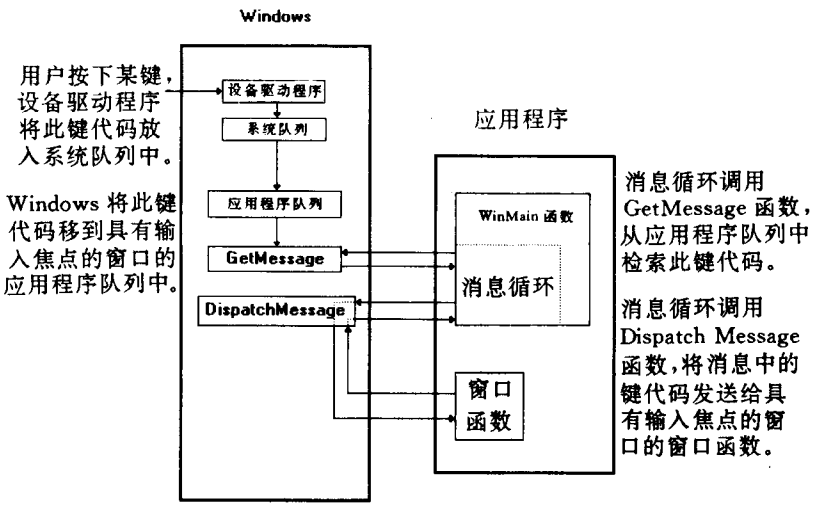

图 1.4　Windows 应用程序的键盘输入

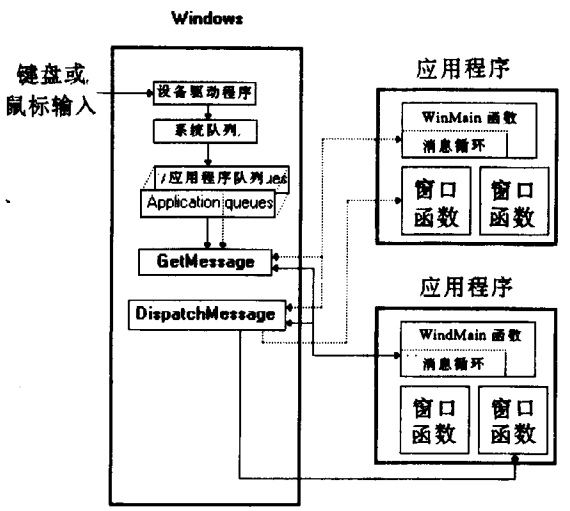

图 1.5　两个 Windows 应用程序的输入

这时程序拥有了消息，但这条消息仍需要被发送给相应的窗口函数。程序是通过调用 Windows 的 DispatchMessage 函数来发送消息的。值得搞清楚的是为什么要调用 Windows 将消息发送给程序内的窗口函数。一个程序可以创建多个窗口。每个窗口可以有它自己唯一的窗口函数，或者多个窗口可以利用相同的窗口函数。除此之外，Windows 提供的窗口类型所用的许多窗口函数根本不在用户应用程序内，而是在 Windows 内部。Windows 中的 DispatchMessage 函数决定获得消息的是程序的哪个窗口函数或 Windows 的内部窗口函数，从而隐藏了所有的复杂性。然后它直接调用正确的窗口函数。图 1.4 显示了键盘输入通过系统的路径，图 1.5 显示了另一个涉及两个应用程序的更复杂例子。

1.5 Windows 操作模式

Windows 3.1 在 Intel 80x86 家族的微处理器上运行。目前这个家族包括 80286，80386 和 80486 微处理器，但不包括 8086，8088，80186 和 80188 微处理器。80286 及其以上的微处理器可以以 Windows 3.1 所要求的保护模式运行。这些不同的处理器的硬件性能从低到高变化极大。Windows 不受该系列中最低级的微处理器性能的限制。一经启动，Windows 将检查用户的系统配置并以最适合该系统的模式运行。Windows 3.1 所支持的两种基本模式是标准模式和 386 增强模式。Windows 3.1 不具备同 Windows 3.0 一样的实模式运行的性能。

当计算机具备一个 80386 或 80486 处理器，至少两兆字节的内存(640K 常规内存和 1024K 扩展内存)时，Windows 通常以 386 增强模式运行。386 增强模式是一种特殊的操作模式，它利用 Intel 80386 和 80486 处理器的虚拟内存性能。虚拟内存性能使 Windows 能够将硬盘空间作为附加的(但较慢的)主存使用。在 Windows 386 增强模式下运行的 Windows 应用程序以处理器的保护模式执行。

当计算机具备一个 80286 处理器，至少一兆字节的内存(640K 常规内存和 256K 的扩展内存)时，Windows 以标准模式运行。标准模式定义为 Windows 的正常操作模式。在 Windows 标准模式下运行的 Windows 应用程序也在处理器的保护模式下执行，但是虚拟内存不可用。

当计算机具有一个 8086，8088，80186 或 80188 处理器，至少 640K 常规内存时，3.1 以前的 Windows 版本在实模式下运行。当计算机具备一个 80286，80386 或 80486 处理器，扩展内存少于前面所说的容量时，这些版本的 Windows 也在实模式下运行。

编写一个在实模式下正确运行的 Windows 程序，过去是一项复杂而乏味的任务。并且，实模式对 Windows 应用程序增加了有效大小的限制。现在实模式已经取消，先前的许多神秘不可测的 Windows 程序设计实践就不再是必须的。在全书中，实模式只是在缺少实模式的支持，改变编写 Windows 应用程序的方式时偶尔提及。内存管理代码的受益最大，不再应该也不需要对每次访问内存都连续地加锁和解锁内存块，只需要在申请过程中加锁内存块一次，在释放内存块之前释放它。

最好在标准模式和 386 增强模式下测试一个应用程序。在各模式下 Windows 内存申请采用不同的代码。在一种模式下未被检测到的内存覆盖可能在另一种模式下检测到。要在两种模式下测试一个应用程序，必须具备访问许多不同的计算机或更可取的，一台具备

一个 80386 或 80486 处理器的计算机的权限。在一台采用一个 80386 或 80486 处理器，两兆字节以上的内存的计算机上，Windows 可以以任一种模式运行。

Windows 可以通过命令行上的开关以一种特定的模式启动。例如，要使 Windows 以标准模式启动，敲入 win/s/n。这些开关是区分大小写的。表 1.1 列出了 Windows 承认的命令行开关。

表 1.1　以不同模式启动 Windows 的命令行开关

开关	功　　能
/S	以标准模式启动 Windows。
/3	以 386 增强模式启动 Windows。
/B	创建 BOOTLOG.TXT 文件并记录在 Windows 的初始装载过程期间，设备驱动器、动态链接库和字体文件装载操作的状态。该文件的内容通常指示出 Windows 启动失败的原因，Windows 装载的设备驱动器及未找到的内容。

1.6　程序内存模式

Windows 和 Windows 应用程序运行在 Intel 8086 系列的微处理器上，该系列具有一种具备段式内存寻址表的体系结构。这种体系结构具有深远的影响，甚至对那些用高级语言的程序设计。

段式内存寻址体系结构产生许多折衷方案。用户可以构造只用一个段的程序和尽可能小的程序，但这样的程序不能大于 64K。用户可以构造用了多个段的较大的程序，但这样的程序的执行要比一个段的程序慢。为 Intel 80x86 微处理器的家族编写的程序可以以很多不同的内存模式构造。所选择的内存模式决定了该程序的代码、数据及用于访问代码和数据的指令的最大大小，和得到的程序的速度。通常，内存模式越大(可用的代码和数据空间越多)，得到的可执行程序越大并且越慢。

1.6.1　8086 体系结构

Intel 8086 微处理器是一个 16 位的计算机。通常情况下 16 位限制计算机只能访问 65536 个不同的地址。但是 8086 微处理器可以寻址 1048576 个不同的位置。这是如何实现的？正如你可能已经猜测到的，段式内存寻址表就是答案。

一个段是一块包含从 1 到 65536 字节的连续的内存区域。一个段的开始地址必须是 16 的倍数。16 的倍数的地址称作段界，从一个段界到下一个段界之间的内存称作一个段。一个段从一个段界起始，向上扩展直到但并不包含下一个段界。因此，一个段是一块 16 字节长的连续的内存区域。一个段必须以一个段界起始。

内存地址 0 开始一个段(0 是 16 的倍数)，这个段从内存地址 0 扩展到内存地址 15，包含地址 15(它是 16 字节长的段)。地址 16 开始一个新的扩展段并包括地址 31。这种模式可以使用户能够对内存段进行编号。从地址 0 到地址 15 的段，包括 15，是段 0。从地址 16 到地址 31 的段，包括 31，是段 1。从地址 65536 到地址 65551 的段，包括 65551，是段 4096。最高的可能段号是 65535，因为这是 16 位所能表示的最大数。段 65535 从内存地址

1048560 扩展到内存地址 1048575。

由于段必须在段界上开始，任一个特定的段都可由它起始的内存地址的段编号唯一标识。一个段号是标识段的起始地址的简单段编号。段本身可以在多个段界之间扩展。

正如以前提到过的，一个段在长度上可以从 1 到 65536 字节。因为一个 16 位的数字可以唯一地标识到 65536 的地址，因此可用一个 16 位数来定位一个段内可能包含的每个字节。偏移量是用于标识段内位置的 16 位数，它表示所感兴趣的位置距离段的起始地址的偏移量。在 Intel 8086 系列微处理机上要求用一个段号和一个段内偏移量的组合来定位所有的内存地址。

段是 8086 体系结构中的一个基本部分。该系列的微处理器具有 4 个用来保存被频繁访问的区域的段号的寄存器。CS(代码段)寄存器保存当前执行程序的代码所在地址的段号。SS(堆栈段)寄存器保存当前栈所在地址的段号。DS(数据段)寄存器保存常用数据的段号。最后，ES(附加段)寄存器是一个在必要时使用的空闲寄存器。

除非另外指明，8086 微处理器从 CS 寄存器所指段处取程序代码，从 SS 寄存器所指段处访问栈内数据。ES 寄存器只用于某些特定的硬件指令。段寄存器中装入值后，以后对内存的该段进行的所有访问就不需要提供完整地址中的段号部分。只需要地址的 16 位偏移量部分。内存访问的类型决定了由哪个段寄存器提供地址的段部分。

程序也可以用一个完全指定的地址——一个段和一个段内偏移量。只要提供一个 16 位的段和一个 16 位的偏移量值，程序就能访问内存中的任意地址。但是，这样要求每个地址的存储是只提供一个偏移量值的两倍。利用段和偏移量地址的指令执行起来要比只用偏移量而用段寄存器来指示段值的指令速度慢。

由于这些原因，只由偏移量部分构成的地址称作近地址或近指针。由段和偏移量构成的完全指明的地址称作远地址或远指针。对当前代码段内子程序的调用只需要指明该子程序在段内的偏移量。这叫做段内调用或近调用。对不同段内子程序的调用必须指明该子程序所在的段及其段内偏移量，这叫做段间调用或远调用。

只有当一个 8086 家族的微处理机在实模式下运行时段寄存器内存储的是段号。利用段号限制了微处理器只能寻址 1 兆的内存。当 Windows 在标准模式或 386 增强模式下运行时，处理器运行在保护模式下，使微处理器可以寻址 16 兆的内存。

当处理器在保护模式下运行时，段寄存器内存储的是选子，而不是段号。此时，远指针由选子和偏移量组成，段间调用要求段的选子和段内偏移量。近指针和段内调用同在实模式下相同只用一个段内偏移量。那么为什么开始把段号称作选子呢？

一个段号直接对应一个物理内存地址，而一个选子值则不对应。段号乘以 16 就是该段起始的物理内存地址。选子提供了一个确定段的物理内存地址的方法。微处理器将选子作为下标用于两个内存映射表之一，获得一个描述符。被访问的描述符包含段起始的线性(或虚拟)内存地址(以及其它信息)。这种简单的变化具有远程访问的作用。在标准模式下，线性地址与物理地址相等。在 386 增强模式下，线性地址可以映射到不同的物理地址。

两个段号之间可以作有意义的比较。较高的段号寻址的段起始物理内存地址高于较低的段号所寻址的地址。两个选子之间的比较没有意义。较高的选子值寻址的段，其起始物理内存地址可能高于或低于较低的选子值寻址的段。事实上，两个不同的选子值能够访

问相同的物理内存地址。

段号总是访问 64K 的内存范围(被访问地址实际上是否有内存是另一个问题!)。选子通过描述符间接访问内存,描述符可以限制段小于 64K。在实模式下程序可以用一个远指针指向一个 16K 的内存段,并自由访问超出它所申请的长度的 48K 内存。在保护模式下,微处理器知道段的长度是 16K,并将禁止程序访问超出这个限制。由于这种额外检查,在标准或 386 增强模式下运行的程序中,脱离的指针问题较容易检测。

运行在实模式下的程序可以用任意值作为段号,这有可能改变内存中的随机地址。运行在保护模式下的程序则不可以。处理器将这个任意值翻译成选子,将其作为下标用于描述符数组。由于未用的表项作了这样的标记,微处理器可以检测非法的选子值。处理器知道描述符数组(通常称作描述符表)的大小,所以它还能够检测大于表大小的选子值。

1.6.2 内存模式

程序内存模式是出于编写高级语言的应用程序及使应用程序尽可能高效运行的需要。只用近指针的程序运行比用远指针的程序要快。但是,限制程序只用近指针意谓着程序代码,数据和栈都不能超过 64K 字节。这对小程序的运行有利,但对具有 70K 代码的程序就不利。因为 70K 代码中至少有 6K 必须在另一个段内,对分段的代码的调用必须是段间或远程调用。

但是,由于数据的访问采用的是不同的段寄存器,因此数据的访问可以独立确定。如果同样的程序具有小于 64K 的数据,它仍然可以用近指针进行所有数据访问。这种结构支持程序在访问完全包含在一个段内的数据时利用更高效更快的指令。在访问多个代码段时利用效率较低较慢的指令。较大程序的代码和数据可能都大于 64K,因此需用远程访问数据和代码。

程序内存模式使你能够为所有指针说明一个大小。这样,你可以指导 C 编译器对每个特殊的应用程序采用最佳的寻址方式。利用标准内存模式的缺点是它们产生的不总是最高效的代码。Microsoft C/C++ 7.0 中的 6 种内存模式见表 1.2 所示。微内存模式不能用于 Windows 应用程序。编译一个使用大内存模式或巨大内存模式的模块时为代码和数据段都产生远程地址。大模式远指针只能在段的偏移大小内扩展,最大可以是 64K。巨大模式远指针可以在 64K 间进行扩展。在巨大模式下,编译器产生正确调整段和偏移量或者选子和偏移量(在保护模式下运行时)的代码。巨大模式仅当指针在 64K 间扩展时与大模式有所不同。一般地,对大模式应用的说明与巨大模式相同。

表 1.2 Microsoft C/C++ 7.0 内存模式

模式	代码段	数据段	最大数组大小	注 释
微	1	1	<64K	代码+数据<=64K
小	1	1	64K	代码+数据<=128K
中	多	1	64K	
紧缩	1	多	64K	
大	多	多	64K	
巨大	多	多	不限	

在大模式程序中全部内存都是可寻址的,所以不需要记录每段中的信息。看起来,大

模式似乎是大应用程序明显的选择。但不幸的是，多数据段程序在 Windows 下运行时会遇到其它的限制。

紧缩、大和巨大模式程序在实模式下把自己的数据段进行固定。Windows 不能将固定段在内存中移动。固定的数据段在内存中造成障碍，使内存管理更困难、更耗时。远数据段必须在内存中固定，因为 Windows 没有能力间接引导到远数据段。例如，程序可以保存一个指向远数据段中的数据的远指针。如果 Windows 在实模式下运行时移动这个数据段，远指针的段号部分将不再有效。此时程序并不知道这个远数据段已经移动，所以它不能更新这个远指针。Windows 知道这个远数据段已经移动，但并不知道程序已经保存了一个指向被移动数据的远指针。

Windows 在保护模式下运行时没有这个问题。在保护模式下，被远指针的选子部分访问的描述符中的地址将被更新。因为选子本身保持不变，因此该远指针的所有备份都保持有效。在所有操作模式下，如果程序使用多数据段，Windows 只装入程序的一个实例。

一般地，为避免这些限制，最好对 Windows 程序采用小或中模式。由于各种原因，本书中的例子将采用中模式。大多数的有效大小的 Windows 应用程序是中模式程序。小模式限制应用程序最大为 64K 代码。以小模式启动的程序可以扩展，直到小模式不再满足。如果只是根据编写应用程序时所做的编码假设，采用中模式重新对小模式的例程进行编译，小模式的例程可能仍不能成功运行。中模式的应用程序的编写只比小模式的应用程序稍难一点。例如，中模式的程序可允许指出在编码中潜在的陷阱，对它们可以进行解释并且正确地处理。

1.6.2.1 混合模式程序

在介绍了 Microsoft C 编译器的内存模式后，可以知道 Windows 应用程序并不严格遵循上面提到的任一种内存模式。即使在编写一个特定内存模式的应用程序时，只要调用一个 Windows API 便可以以多种方式脱离这种内存模式。

Windows 3.1 API 包含了 1000 多个函数。当 Windows 扩展，例如包括 Pen Windows 和 Windows 多媒体时，这个数目将会增加。这些函数不可能与应用程序装入一个段内，因为它们必须位于一个不同的段内，必须利用一个远程(段间)调用将控制传送给它们。并且，总体上，Windows 函数可以被看作是一个独立的应用程序。这样，它们拥有自己的代码和数据段。Windows 函数要求的指向应用程序内数据的指针必须是一个远指针。传递给 Windows 函数的近指针会被翻译成 Windows 函数的数据段内的偏移量，而不是应用程序的数据段内的偏移量。事实上，除非传递的是一个远指针，Windows 函数并不知道应用程序的数据段驻留在何处。

指向数据的远指针不用于小或中内存模式中。对函数的远程调用不用于小或紧缩内存模式中。Microsoft C/C++ 7.0 编译器识别两个关键字，__near 和__far，支持用户确切地指定一个指针的大小或对函数调用的类型。Windows 程序设计，通常是采用一种标准内存模式并且混合几种非标准(所选择的模式)内存调用，由于这种特性，它通常被称作混合模式程序设计。

1.7 __cdcel 和__pascal 调用顺序

大多数 Windows 函数调用需要向函数传递一些参数。函数的参数以栈的方式传递。把这些参数的拷贝以栈的方式传递称作推参数入栈。移去参数称作从栈中弹出参数。函数调用中的参数可以按从左到右(同 C 语言函数调用中的列出顺序一致)或者从右到左的顺序推入栈中。最终,C 函数通常采用从右到左的顺序推参数入栈(也就是说,第一个——最左——参数是最后推入栈中的参数),而 Windows 函数要求参数按从左到右的顺序推入。

这种差别的原因一部分在于 C 程序设计语言的定义,一部分在于 8086 微处理器家族体系结构的设计。C 语言允许函数带可变数目的参数。在 8086 体系上,按从右到左的顺序向函数推进参数时,将第一个参数置于一个已知的位置,使它能够被调用函数找到。第一个参数被定位之后,所有依次的参数就能被找到。因此,在 8086 家族体系上,由于 C 语言函数可能接收不同数目的参数,参数按从右到左的顺序推入。

函数完成了它的任务之后,推入的参数必须从栈中移出。也就是说,必须从栈中弹出正确数目的参数。编译器在编译时并不知道一个特定的函数可能接收的参数数目。事实上,一个特定的函数在每次被调用时可能接收不同数目的参数。这种用法最基本的例子是 printf 函数。被调用的 C 函数不能清除栈中传递给它的参数,因为参数的数目可能变化,直到运行时才可知。因此,响应性取决于调用程序。函数调用返回之后,从栈中移去与推入栈中数目相等的参数。

这里,有一种稍微更有效的调用顺序。将函数限制固定数目的参数之后,参数可以按两种顺序中的任意一种推入栈中。而且,编译器在编译时知道函数需要的参数数目。这样,编译器能够生成代码在调用程序末尾将参数从栈中移去,而不是在每次调用函数之后移去。只生成一次而不是多次这种代码节省了生成的程序空间。另外,8086 体系结构提供了在函数返回同时从栈中移去参数的硬件支持。

缺省时,Microsoft C 编译器采用支持可变数目的参数的调用顺序生成对函数的调用。它支持 C 语言的定义。但是,Windows 函数带固定数目的参数。因此,Windows 函数采用第二种调用顺序更为有效。Microsoft C 编译器识别两个关键字,__cdecl 和__pascal,使用户能够指定在函数与函数的基础上所采用的调用顺序。__cdecl 关键字告知编译器其后的函数采用 C 调用顺序,__pascal 关键字告知编译器其后函数采用更为有效的调用顺序。在 WINDOWS.H 文件中通过许多 typedefs 将 Windows 函数定义为采用__pascal 调用顺序。

1.8 静态和动态链接

现在,已有足够的知识来选择一种内存模式,指出对那种允许向 Windows 函数传递参数的模式的例外,并在调用函数时采用正确的调用顺序便于找到过去的信息。这里存在两个问题。函数在哪里? 应用程序中的调用如何与隐藏在 Windows 某处的函数相联系?

链接器用来链接应用程序中的调用与它的调用函数。Windows 程序采用一种新的链

接器形式，称作分段可执行链接器。这个链接器由 Microsoft C 5.1，6.x 和 C/C++ 7.0 编译器提供。用户必须用分段可执行链接器来链接 Windows 程序。新的程序设计人员在 Windows 程序设计中常遇到的错误是用错编译器。这是由于 Microsoft C 5.1 编译器软件包含两个链接器——一个链接器和一个分段可执行链接器——其中只有一个用于链接 Windows 程序。分段可执行链接器只是在需要 OS/2 支持时由 C5.1 安装过程安装。要确保你正在使用 5.x 或更高版本的链接器，并在它的标题消息中将自身描述为分段可执行链接器。

在将一个或多个目标模块(.OBJ 文件)链接在一起形成一个可执行程序(.EXE 文件)时，链接器将目标模块内的调用匹配到同一个或另一个目标模块内的函数。在执行完这步之后，如果存有未分解的调用，链接器查寻库文件搜寻丢失的函数。链接器搜寻的库文件是在 LINK 命令行上指定的和在被链接的目标模块内显示命名的库文件。如果在搜寻完所有的命名库文件之后，链接器仍未能找到所调用的函数，链接器将产生一条“未分解的外部调用”消息来告知用户由其程序所调用的一个或多个函数不能被找到。如果链接器找到函数，则从库中将它的目标代码拷贝出来并插入到可执行文件中。这是常用的链接形式，称作静态链接。静态链接要求链接器在链接时知道函数将驻留在内存的何处并且访问构成该函数的目标代码。

但是，链接器在链接时并不知道 Windows 函数驻留在内存的何处。实际上，Windows 函数可能在内存中不断地移动，因此即使链接器及时知道定位点，函数在被调用时并不一定仍位于相同的位置上。而且，Windows 是一个合作环境。多个应用程序同时运行。最好是设计一种所有并行运行的应用程序，都能共享每个 Windows 函数的一个拷贝的方法。这样一种方法已经存在，称作动态链接。

动态链接是链接器用来推延函数的全面分解直至程序执行的时候。无需在链接的时候固定所有的远程调用，链接器将附加信息插入到可执行程序里，向 Windows 通知未分解的函数。当 Windows 装入程序时，被推迟的远程调用被动态地嵌入 Windows 内适当的函数中。链接器确定利用入口库将外部调用动态地链入。

链接器象查询目标代码库一样，查询入口库，搜寻在被链接的程序中出现的未定义的外部调用。但是，入口库中不包含代码。相反，库中包含了一些外部调用所在的 Windows 模块的名称和函数所在的 Windows 模块的名称或在该模块内的入口号。这个信息被拷贝到可执行程序中，链接器认为该外部调用已被分解。之后，当该程序执行时，Windows 利用这个信息分解此远程调用。

链接 Windows 程序所用的入口库是 LIBW.LIB，此库与模式无关，不包含代码。LIBW.LIB 中包含所有可被 Windows 应用程序调用的入口记录。

严格地讲，入口库不过是一些没有代码只包含入口记录的库。库中可以包含在静态链接中所用的函数的目标模块和在动态链接中所用的入口记录。但通常可以很方便地将一个库分解为一个与模式有关的代码库和一个与模式无关的入口库。

1.9 动态链接库

分段可执行的链接器既可产生程序模块也可产生动态链接库(DDL)。程序模块是一