

第六部分 窗 口

第26章 窗口消息

本章介绍Microsoft Windows的消息系统是如何支持带有图形用户界面的应用程序的。在设计Windows 2000或Windows 98所使用的窗口系统时，微软有两个主要目标：

- 尽可能保持与过去 16位Windows的兼容性，便于开发人员移植他们已有的 16位Windows程序。
- 使窗口系统强壮，一个线程不会对系统中其他线程产生不利影响。

但是，这两个目标是直接相互冲突的。在 16位Windows系统中，向窗口发送一个消息总是按同步方式执行的：发送程序要在接受消息的窗口完全处理完消息之后才能继续运行。这通常是一个所期望的特性。但是，如果接收消息的窗口花很长的时间来处理消息或者出现挂起，则发送程序就不能再执行。这意味着系统是不强壮的。

这种冲突给微软的设计人员带来了一定的困难。他们的解决方案是两个相互冲突目标之间的出色折衷方案。如果在阅读本章时记住这两个目标，你就会更多地理解微软为什么会做出这样的设计。

我们从一些基本原则开始讨论。Windows允许一个进程至多建立 10 000个不同类型的用户对象（User object）：图符、光标、窗口类、菜单、加速键表等等。当一个线程调用一个函数来建立某个对象时，则该对象就归这个线程的进程所拥有。这样，当进程结束时，如果没有明确删除这个对象，则操作系统会自动删除这个对象。对窗口和挂钩（hook）这两种User对象，它们分别由建立窗口和安装挂钩的线程所拥有。如果一个线程建立一个窗口或安装一个挂钩，然后线程结束，操作系统会自动删除窗口或卸载挂钩。

这种线程拥有关系的概念对窗口有重要的意义：建立窗口的线程必须是为窗口处理所有消息的线程。为了使这个概念更加明确具体，可以想像一个线程建立了一个窗口，然后就结束了。在这种情况下，窗口不会收到一个 WM_DESTROY或WM_NCDESTROY消息，因为线程已经结束，不可能被用来使窗口接收和处理这些消息。

这也意味着每个线程，如果它至少建立了一个窗口，都由系统对它分配一个消息队列。这个队列用于窗口消息的派送（dispatch）。为了使窗口接收这些消息，线程必须有它自己的消息循环。本章要考查每个线程的消息队列。特别是要看看消息是如何被放置在队列中的，以及线程如何从队列中取出消息并处理它们。

26.1 线程的消息队列

前面已经说过，Windows的一个主要目标是为程序的运行提供一个强壮的环境。为实现这个目标，要保证每个线程运行在一个环境中，在这个环境中每个线程都相信自己是唯一运行的线程。更确切地说，每个线程必须有完全不受其他线程影响的消息队列。而且，每个线程必须有一个模拟环境，使线程可以维持它自己的键盘焦点（keyboard focus）、窗口激活、鼠标捕获等概念。

当一个线程第一次被建立时，系统假定线程不会被用于任何与用户相关的任务。这样可以减少线程对系统资源的要求。但是，一旦这个线程调用一个与图形用户界面有关的函数（例如检查它的消息队列或建立一个窗口），系统就会为该线程分配一些另外的资源，以便它能够执行与用户界面有关的任务。特别是，系统分配一个THREADINFO结构，并将这个数据结构与线程联系起来。

这个THREADINFO结构包含一组成员变量，利用这组成员，线程可以认为它是在自己独占的环境中运行。THREADINFO是一个内部的、未公开的数据结构，用来指定线程的登记消息队列（posted-message queue）、发送消息队列（send-message queue）、应答消息队列（reply-message queue）、虚拟输入队列（virtualized-input queue）、唤醒标志（wake flag）以及用来描述线程局部输入状态的若干变量。图26-1描述了THREADINFO结构和与之相联系三个线程。

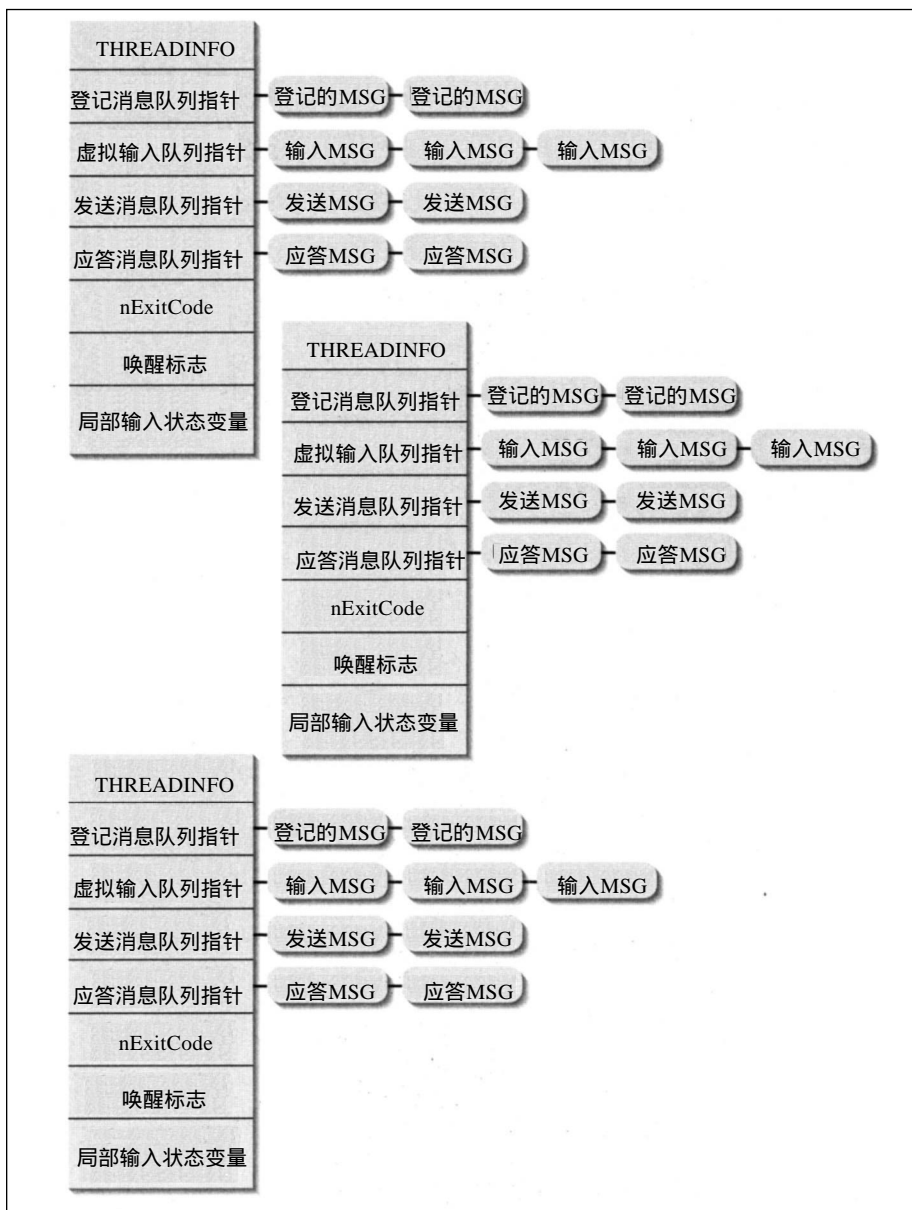


图26-1 三个线程及相应的THREADINFO结构

这个THREADINFO结构是窗口消息系统的基础，在阅读下面各节内容时，应该参考该图。

26.2 将消息发送到线程的消息队列中

当线程有了与之相联系的 THREADINFO结构时，线程就有了自己的消息队列集合。如果一个进程建立了三个线程，并且所有这些线程都调用 CreateWindow，则将有三个消息队列集合。消息被放置在线程的登记消息队列中，这要通过调用 PostMessage函数来完成：

```
BOOL PostMessage(  
    HWND hwnd,  
    UINT uMsg,  
    WPARAM wParam,  
    LPARAM lParam);
```

当一个线程调用这个函数时，系统要确定是哪一个线程建立了用 hwnd参数标识的窗口。然后系统分配一块内存，将这个消息参数存储在这块内存中，并将这块内存增加到相应线程的登记消息队列中。并且，这个函数还设置 QS_POSTMESSAGE唤醒位（后面会简单讨论）。函数PostMessage在登记了消息之后立即返回，调用该函数的线程不知道登记的消息是否被指定窗口的窗口过程所处理。实际上，有可能这个指定的窗口永远不会收到登记的消息。如果建立这个特定窗口的线程在处理完它的消息队列中的所有消息之前就结束了，就会发生这种事。

还可以通过调用 PostThreadMessage将消息放置在线程的登记消息队列中。

```
BOOL PostThreadMessage(  
    DWORD dwThreadId,  
    UINT uMsg,  
    WPARAM wParam,  
    LPARAM lParam);
```

注意 可以通过调用 GetWindowsThreadProcessId来确定是哪个线程建立了一个窗口。

```
DWORD GetWindowThreadProcessId(  
    HWND hwnd,  
    PDWORD pdwProcessId);
```

这个函数返回线程的ID，这个线程建立了hwnd参数所标识的窗口。线程ID在全系统范围内是唯一的。还可以通过对pdwProcessId参数传递一个DWORD地址来获取拥有该线程的进程ID，这个进程ID在全系统范围内也是唯一的。通常，我们不需要进程ID，只须对这个参数传递一个NULL。

PostThreadMessage函数所期望的线程由第一个参数 dwThreadId所标记。当消息被设置到队列中时，MSG结构的hwnd成员将设置成NULL。当程序要在主消息循环中执行一些特殊处理时要调用这个函数。

要对线程编写主消息循环以便在 GetMessage或PeekMessage取出一个消息时，主消息循环代码检查hwnd是否为NULL，并检查MSG结构的msg成员来执行特殊的处理。如果线程确定了该消息不被指派给一个窗口，则不调用 DispatchMessage，消息循环继续取下一个消息。

像PostMessage函数一样，PostThreadMessage在向线程的队列登记了消息之后就立即返回。调用该函数的线程不知道消息是否被处理。

向线程的队列发送消息的函数还有 PostQuitMessage：

```
VOID PostQuitMessage(int nExitCode);
```

为了终止线程的消息循环，可以调用这个函数。调用 PostQuitMessage类似于调用：

```
PostThreadMessage(GetCurrentThreadId(), WM_QUIT, nExitCode, 0);
```

但是，PostQuitMessage并不实际登记一个消息到任何一个 THREADINFO 结构的队列。只是在内部，PostQuitMessage 设定 QS_QUIT 唤醒标志（后面将要讨论），并设置 THREADINFO 结构的 nExitCode 成员。因为这些操作永远不会失败，所以 PostQuitMessage 的原型被定义成返回 VOID。

26.3 向窗口发送消息

使用 SendMessage 函数可以将窗口消息直接发送给了一个窗口过程：

```
LRESULT SendMessage(  
    HWND hwnd,  
    UINT uMsg,  
    WPARAM wParam,  
    LPARAM lParam);
```

窗口过程将处理这个消息。只有当消息被处理之后，SendMessage 才能返回到调用程序。由于具有这种同步特性，比之 PostMessage 或 PostThreadMessage，SendMessage 用得更频繁。调用这个函数的线程在下一行代码执行之前就知道窗口消息已经被完全处理。

SendMessage 是如何工作的呢？如果调用 SendMessage 的线程向该线程所建立的一个窗口发送一个消息，SendMessage 就很简单：它只是调用指定窗口的窗口过程，将其作为一个子例程。当窗口过程完成对消息的处理时，它向 SendMessage 返回一个值。SendMessage 再将这个值返回给调用线程。

但是，当一个线程向其他线程所建立的窗口发送消息时，SendMessage 的内部工作就复杂得多（即使两个线程在同一进程中也是如此）。Windows 要求建立窗口的线程处理窗口的消息。所以当一条线程调用 SendMessage 向一个由其他进程所建立的窗口发送一个消息，也就是向其他的线程发送消息，发送线程不可能处理窗口消息，因为发送线程不是运行在接收进程的地址空间中，因此不能访问相应窗口过程的代码和数据。实际上，发送线程要挂起，而由另外的线程处理消息。所以为了向其他线程建立的窗口发送一个窗口消息，系统必须执行下面将讨论的动作。

首先，发送的消息要追加到接收线程的发送消息队列，同时还为这个线程设定 QS_SENDMESSAGE 标志（后面将讨论）。其次，如果接收线程已经在执行代码并且没有等待消息（如调用 GetMessage、PeekMessage 或 WaitMessage），发送的消息不会被处理，系统不能中断线程来立即处理消息。当接收进程在等待消息时，系统首先检查 QS_SENDMESSAGE 唤醒标志是否被设定，如果是，系统扫描发送消息队列中消息的列表，并找到第一个发送的消息。有可能在这个队列中有几个发送的消息。例如，几个线程可以同时向一个窗口分别发送消息。当发生这样的事时，系统只是将这些消息追加到接收线程的发送消息队列中。

当接收线程等待消息时，系统从发送消息队列中取出第一个消息并调用适当的窗口过程来处理消息。如果在发送消息队列中再没有消息了，则 QS_SENDMESSAGE 唤醒标志被关闭。当接收线程处理消息的时候，调用 SendMessage 的线程被设置成空闲状态（idle），等待一个消息出现在它的应答消息队列中。在发送的消息处理之后，窗口过程的返回值被登记到发送线程的应答消息队列中。发送线程现在被唤醒，取出包含在应答消息队列中的返回值。这个返回值就是调用 SendMessage 的返回值。这时，发送线程继续正常执行。

当一个线程等待 SendMessage 返回时，它基本上是处于空闲状态。但它可以执行一个任务：如果系统中另外一个线程向一个窗口发送消息，这个窗口是由这个等待 SendMessage 返回的线

程所建立的，则系统要立即处理发送的消息。在这种情况下，系统不必等待线程去调用 GetMessage、Peek Message或WaitMessage。

由于Windows使用上述方法处理线程之间发送的消息，所以有可能造成线程挂起（hang）。例如，当处理发送消息的线程含有错误时，会导致进入死循环。那么对于调用 SendMessage的线程会发生什么事呢？它会恢复执行吗？这是否意味着一个程序中的 bug会导致另一个程序挂起？答案是确实有这种可能。

利用4个函数——SendMessageTimeout、SendMessageCallback、SendNotifyMessage和ReplyMessage，可以编写保护性代码防止出现这种情况。第一个函数是SendMessageTimeout：

```
LRESULT SendMessageTimeout(  
    HWND hwnd,  
    UINT uMsg,  
    WPARAM wParam,  
    LPARAM lParam,  
    UINT fuFlags,  
    UINT uTimeout,  
    PDWORD_PTR pdwResult);
```

利用SendMessageTimeout函数，可以规定等待其他线程答回你消息的时间最大值。前4个参数与传递给SendMessage的参数相同。对fuFlags参数，可以传递值SMTO_NORMAL(定义为0)、SMTO_ABORTIFHUNG、SMTO_BLOCK、SMTO_NOTIMEOUTIFNOTHUNG或这些标志的组合。

SMTO_ABORTIFHUNG标志是告诉SendMessageTimeout去查看接收消息的线程是否处于挂起状态，如果是，就立即返回。SMTO_NOTIMEOUTIFNOTHUNG标志使函数在接收消息的线程没有挂起时不考虑等待时间限定值。SMTO_BLOCK标志使调用线程在SendMessageTimeout返回之前，不再处理任何其他发送来的消息。SMTO_NORMAL标志在Winuser.h中定义成0，如果不想指定任何其他标志及组合，就使用这个标志。

前面说过，一个线程在等待发送的消息返回时可以被中断，以便处理另一个发送来的消息。使用SMTO_BLOCK标志阻止系统允许这种中断。仅当线程在等待处理发送的消息的时候（不能处理别的发送消息），才使用这个标志。使用SMTO_BLOCK可能会产生死锁情况，直到等待时间期满。例如，如果你的线程向另外一个线程发送一个消息，而这个线程又需要向你的线程发送消息。在这种情况下，两个线程都不能继续执行，并且都将永远挂起。

SendMessageTimeout函数中的uTimeout参数指定等待应答消息时间的毫秒数。如果这个函数执行成功，返回TRUE，消息的结果复制到一个缓冲区中，该缓冲区的地址由pdwResult参数指定。

顺便提一下，这个函数在WinUser.h头文件中的原型是不正确的。这个函数的原型应该被定义成返回一个BOOL型值，因为LRESULT实际是通过函数的一个参数返回的。这会引起一些问题，因为如果对函数传递一个无效的窗口句柄或者等待超时，SendMessageTimeout都会返回FALSE。要知道函数失败详细情况的唯一办法是调用 GetLastError。如果函数是由于等待超时而失败，则GetLastError为0（ERROR_SUCCESS）。如果对参数传递了一个无效句柄，GetLastError为1400（ERROR_INVALID_WINDOW_HANDLE）。

如果调用SendMessageTimeout向调用线程所建立的窗口发送一个消息，系统只是调用这个窗口的窗口过程，并将返回值赋给pdwResult。因为所有的处理都必须发生在一个线程里，调用SendMessageTimeout函数之后出现的代码要等消息被处理完之后才能开始执行。

用来在线程间发送消息的第二个函数是SendMessageCallback：

```
BOOL SendMessageCallback(  
    HWND hwnd,  
    UINT uMsg,  
    WPARAM wParam,  
    LPARAM lParam,  
    SENDASYNCPROC pfnResultCallback,  
    ULONG_PTR dwData);
```

同样，前4个参数同SendMessage中使用的一样。当一个线程调用SendMessageCallback时，该函数发送消息到接收线程的发送消息队列，并立即返回使发送线程可以继续执行。当接收线程完成对消息的处理时，一个消息被登记到发送线程的应答消息队列中。然后，系统通过调用一个函数将这个应答通知给发送线程，该函数是使用下面的原型编写的。

```
VOID CALLBACK ResultCallback(  
    HWND hwnd,  
    UINT uMsg,  
    ULONG_PTR dwData,  
    LRESULT lResult);
```

必须将这个函数的地址传递给SendMessageCallback函数作为pfnResultCallback参数值。当调用这个函数时，要把完成消息处理的窗口的句柄传递到第一个参数，将消息值传递给第二个参数。第三个参数dwData，总是取传递到SendMessageCallback函数的dwData参数的值。系统只是取出对SendMessageCallback函数指定的参数值，再直接传递到ResultCallback函数。ResultCallback函数的最后一个参数是处理消息的窗口过程返回的结果。

因为SendMessageCallback在执行线程间发送时会立即返回，所以在接收线程完成对消息的处理时不是立即调用这个回调函数。而是由接收线程先将一个消息登记到发送线程的应答消息队列。发送线程在下次调用GetMessage、PeekMessage、WaitMessage或某个SendMessage*函数时，消息从应答消息队列中取出，并执行ResultCallback函数。

SendMessageCallback函数还有另外一个用处。Windows提供了一种广播消息的方法，用这种方法你可以向系统中所有现存的重叠（overlapped）窗口广播一个消息。这可以通过调用SendMessage函数，对参数hwnd传递HWND_BROADCAST（定义为-1）。使用这种方法广播的消息，其返回值我们并不感兴趣，因为SendMessage函数只能返回一个LRESULT。但使用SendMessageCallback，就可以向每一个重叠窗口广播消息，并查看每一个返回结果。对每一个处理消息的窗口的返回结果都要调用ResultCallback函数。

如果调用SendMessageCallback向一个由调用线程所建立的窗口发送一个消息，系统立即调用窗口过程，并且在消息被处理之后，系统调用ResultCallback函数。在ResultCallback函数返回之后，系统从调用SendMessageCallback的后面的代码行开始执行。

线程间发送消息的第三个函数是SendNotifyMessage：

```
BOOL SendNotifyMessage(  
    HWND hwnd,  
    UINT uMsg,  
    WPARAM wParam,  
    LPARAM lParam);
```

SendNotifyMessage将一个消息置于接收线程的发送消息队列中，并立即返回到调用线程。这一点与PostMessage函数一样，但SendNotifyMessage在两方面与PostMessage不同。

首先，SendNotifyMessage是向另外的线程所建立的窗口发送消息，发送的消息比起接收线程消息队列中存放的登记消息有更高的优先级。换句话说，由SendNotifyMessage函数存放

在队列中的消息总是在PostMessage函数登记到队列中的消息之前取出。

其次，当向一个由调用进程建立的窗口发送消息时，SendNotifyMessage同SendMessage函数完全一样：SendNotifyMessage在消息被处理完之后才能返回。

我们已经知道，发送给窗口的大多数消息是用于通知的目的。也就是，发送消息是因为窗口需要知道某个状态已经发生变化，在程序能够继续执行之前，窗口要做某种处理。例如，WM_ACTIVATE、WM_DESTROY、WM_ENABLE、WM_SIZE、WM_SETFOCUS和WM_MOVE等都是系统发送给窗口的通知，而不是登记的消息。这些消息是系统对窗口的通知，因此系统不需要停止运行以等待窗口过程处理这些消息。与此相对应，如果系统向一个窗口发送一个WM_CREATE消息，则在窗口处理完这个消息之前，系统必须等待。如果返回值是-1，则不再建立窗口。

第四个用于线程发送消息的函数是ReplyMessage：

```
BOOL ReplyMessage(LRESULT lResult);
```

这个函数与前面讨论过的三个函数不同。线程使用 SendMessageTimeout、SendMessageCallback和SendNotifyMessage发送消息，是为了保护自己以免被挂起。而线程调用 ReplyMessage是为了接收窗口消息。当一个线程调用 ReplyMessage时，它是要告诉系统，为了知道消息结果，它已经完成了足够的工作，结果应该包装起来并登记到发送线程的应答消息队列中。这将使发送线程醒来，获得结果，并继续执行。

调用ReplyMessage的线程在lResult参数中指出消息处理的结果。在调用ReplyMessage之后，发送消息的线程恢复执行，而处理消息的线程继续处理消息。两个线程都不会被挂起，都可以正常执行。当处理消息的线程从它的窗口过程返回时，它返回的任何值都被忽略。

这里的问题是，ReplyMessage必须在接收消息的窗口过程中调用，而不是由调用某个 Send * 函数的线程调用。为了编写保护性代码，最好不要用前面讨论过的三个 Send * 函数中的一个代替对SendMessage的调用，而是依靠窗口过程的实现者来调用 ReplyMessage。

还应该知道，如果在处理一个由同一线程发送来的消息时调用 ReplyMessage，则该函数什么也不做。实际上，这就是ReplyMessage的返回值所指出的。如果你在处理线程间的消息发送时调用了 ReplyMessage，则它返回 TRUE，如果你在处理线程内的消息发送时调用 ReplyMessage，它返回FALSE。

有时候，你可能想知道究竟是在处理线程间的消息发送，还是在处理线程内的消息发送。为了搞清楚这一点，可以调用InSendMessage：

```
BOOL InSendMessage();
```

这个函数的名字不能够确切说明它究竟做什么事。初看时，你会以为，当线程在处理一个发送的消息时，该函数返回 TRUE，而在处理一个登记的消息时，它返回 FALSE。如果这样想你就错了。这个函数在线程处理线程间发送的消息时，返回 TRUE，而在线程处理线程内发送的或登记的消息时，返回 FALSE。InSendMessage和ReplyMessage的返回值是一样的。

还可以调用另外一个函数来确定窗口过程正在处理的消息类型：

```
DWORD InSendMessageEx(PVOID pvReserved);
```

当调用这个函数时，必须对 pvReserved参数传递 NULL。这个函数的返回值指出正在处理的消息的类型。如果返回值是 ISMEX_NOSEND（定义为0），表示线程正在处理一个线程内发送的或登记的消息。如果返回值不是 ISMEX_NOSEND，就是表26-1中描述的位标志的组合。

表26-1 位标志的组合

标 志	描 述
ISMEX_SEND	线程在处理一个线程间发送的消息，该消息是用 SendMessage 或 Send Message Timeout 函数发送的。如果没有设定 ISMEX_REPLIED 标志，发送线程被阻塞，等待应答
ISMEX_NOTIFY	线程在处理一个线程间发送的消息，该消息是用 Send Notify Message 函数发送的。发送线程不等待应答，也不会阻塞
ISMEX_CALLBACK	线程在处理线程间发送的消息，该消息是用 SendMessage Callback 发送的。发送线程不等待应答，也不会被阻塞
ISMEX_REPLIED	线程在处理线程间发送的消息，并已经调用 Reply Message。发送线程不会被阻塞

26.4 唤醒一个线程

当一个线程调用 GetMessage 或 WaitMessage，但没有对这个线程或这个线程所建立窗口的消息时，系统可以挂起这个线程，这样就不再给它分配 CPU 时间。当有一个消息登记或发送到这个线程，系统要设置一个唤醒标志，指出现在要给这个线程分配 CPU 时间，以便处理消息。正常情况下，如果用户不按键或移动鼠标，就没有消息发送给任何窗口。这意味着系统中大多数线程没有被分配给 CPU 时间。

26.4.1 队列状态标志

当一个线程正在运行时，它可以通过调用 GetQueueStatus 函数来查询队列的状态：

```
DWORD GetQueueStatus(UINT fuFlags);
```

参数 fuFlags 是一个标志或一组由 OR 连接起来的标志，用来测试特定的唤醒位。表 26-2 给出了各个标志取值及含义。

表26-2 标志取值及含义

标 志	队列中的消息
QS_KEY	WM_KEYUP、WM_KEYDOWN、WM_SYSKEYUP 或 WM_SYSKEYDOWN
QS_MOUSEMOVE	WM_MOUSEMOVE
QS_MOUSEBUTTON	WM_?BUTTON* (其中 ? 代表 L、M 或 R、* 代表 DOWN、UP 或 DBLCLK)
QS_MOUSE	同 QS_MOUSEMOVE QS_MOUSEBUTTON
QS_INPUT	同 QS_MOUSE QS_KEY
QS_PAINT	WM_PAINT
QS_TIMER	WM_TIMER
QS_HOTKEY	WM_HOTKEY
QS_POSTMESSAGE	登记的消息 (不同于硬件输入事件)。当队列在期望的消息过滤器范围内没有登记的消息时，这个标志要消除。除此之外，这个标志与 QS_ALLPOSTMESSAGE 相同
QS_ALLPOSTMESSAGE	登记的消息 (不同于硬件输入事件)。当队列完全没有登记的消息时 (在任何消息过滤器范围)，该标志被清除。除此之外，该标志与 QS_POSTMESSAGE 相同
QS_ALLEVENTS	同 QS_INPUT QS_POSTMESSAGE QS_TIMER QS_PAINT QS_HOTKEY
QS_QUIT	已调用 PostQuitMessage。注意这个标志没有公开，所以在 WinUser.h 文件中没有。它由系统在内部使用
QS_SENDMESSAGE	由另一个线程发送的消息
QS_ALLINPUT	同 QS_ALLEVENTS QS_SENDMESSAGE

当调用GetQueueStatus函数时，fuFlags将队列中要检查的消息的类型告诉GetQueueStatus。用OR连接的QS_*标识符的数量越少，调用执行的就越快。当GetQueueStatus返回时，线程的队列中当前消息的类型在返回值的高字（两字节）中。这个返回的标志的集合总是所想要的标志集的子集。例如，对下面的调用：

```
BOOL fPaintMsgWaiting = HIWORD(GetQueueStatus(QS_TIMER)) & QS_PAINT;
```

fPaintMsgWaiting的值总是FALSE，不论队列中是否有一个WM_PAINT消息在等待，因为GetQueueStatus的参数中没有将QS_PAINT指定为一个标志。

GetQueueStatus返回值的低字指出已经添加到队列中，并且在上一次对函数GetQueueStatus、GetMessage或PeekMessage调用以来还没有处理的消息的类型。

不是所有的唤醒标志都由系统平等对待。对于QS_MOUSEMOVE标志，只要队列中存在一个未处理的WM_MOUSEMOVE消息，这个标志就要被设置。当GetMessage或PeekMessage（利用PM_REMOVE）从队列中放入新的WM_MOUSEMOVE消息之前，这个标志被关闭。QS_KEY、QS_MOUSEBUTTON和QS_HOTKEY标志都根据相应的消息按与此相同的方式处理。

QS_PAINT标志的处理与此不同。如果线程建立的一个窗口有无效的区域，QS_PAINT标志被设置。当这个线程建立的所有窗口所占据的区域变成无效时（通常由于对ValidateRect、ValidateRegion或BeginPaint的调用而引起），QS_PAINT标志就被关闭。只有当线程建立的所有窗口都无效时，这个标志才关闭。调用GetMessage或PeekMessage对这个唤醒标志没有影响。

当线程的登记消息队列中至少有一个消息时，QS_POSTMESSAGE标志就被设置。这不包括线程的虚拟输入队列中的硬件事件消息。当线程的登记消息队列中的所有消息都已经处理，队列变空时，这个标志被复位。

每当一个定时器（由线程所建立）报时（go off），QS_TIMER标志就被设置。在GetMessage或PeekMessage返回WM_TIMER事件之后，QS_TIMER标志被复位，直到定时器再次报时。

QS_SENDMESSAGE标志指出有一个消息在线程的发送消息队中。系统在内部使用这个标志，用来确认和处理线程之间发送的消息。对于一个线程向自身发送的消息，不设置这个标志。虽然可以使用QS_SENDMESSAGE标志，但很少需要这样做。笔者还从未见到一个程序使用这个标志。

还有一个未公开的队列状态标志QS_QUIT。当一个线程调用PostQuitMessage时，QS_QUIT标志就被设置。系统并不实际向线程的消息队列追加一个WM_QUIT消息。GetQueueStatus函数也不返回这个标志的状态。

26.4.2 从线程的队列中提取消息的算法

当一个线程调用GetMessage或PeekMessage时，系统必须检查线程的队列状态标志的情况，并确定应该处理哪个消息。图26-2和下面叙述的步骤说明了系统是如何确定线程应该处理的下一个消息的情况。

1) 如果QS_SENDMESSAGE标志被设置，系统向相应的窗口过程发送消息。GetMessage或PeekMessage函数在内部进行这种处理，并且在窗口过程处理完消息之后不返回到线程，这些函数要等待其他要处理的消息。

2) 如果消息在线程的登记消息队列中，函数GetMessage或PeekMessage填充传递给它们的MSG结构，然后函数返回。这时，线程的消息循环通常调用DispatchMessage，让相应的窗口过程来处理消息。

3) 如果QS_QUIT标志被设置。GetMessage或PeekMessage返回一个WM_QUIT消息（其中wParam参数是规定的退出代码）并复位QS_QUIT标志。

- 4) 如果消息在线程的虚拟输入队列, 函数 GetMessage 或 PeekMessage 返回硬件输入消息。
- 5) 如果 QS_PAINT 标志被设置, GetMessage 或 PeekMessage 为相应的窗口返回一个 WM_PAINT 消息。
- 6) 如果 QS_TIMER 标志被设置, GetMessage 或 PeekMessage 返回一个 WM_TIMER 消息。

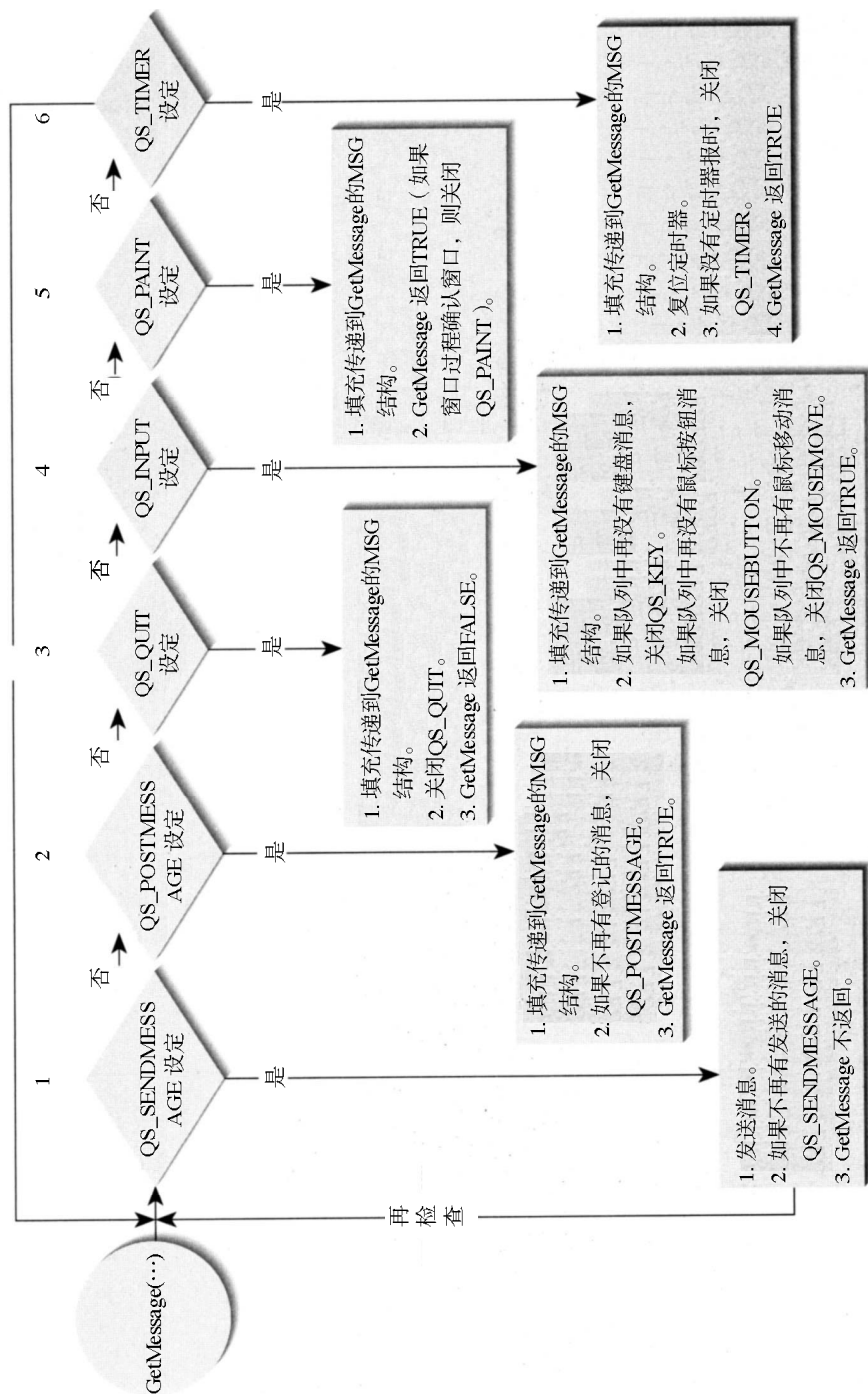


图26-2 从线程队列中提取消息的算法

尽管很难令人相信，但确有理由这样做。微软在设计这个算法时有一个大前提，就是应用程序应该是用户驱动的，用户通过建立硬件输入事件（键盘和鼠标操作）来驱动应用程序。在使用应用程序时，用户可能按一个鼠标按钮，引起一系列要发生的事件。应用程序通过向线程的消息队列中登记消息使每个个别的事件发生。

所以如果按鼠标按钮，处理 WM_LBUTTONDOWN 消息的窗口可能向不同的窗口投送三个消息。由于是硬件事件引发三个软件事件，所以系统要在读取用户的下一个硬件事件之前，先处理这些软件事件。这也说明了为什么登记消息队列要在虚拟输入队列之前检查。

这种事件序列的一个很好的例子是调用 TranslateMessage 函数。这个函数检查是否有一个 WM_KEYDOWN 或一个 WM_SYSKEYDOWN 消息从输入队列中取出。如果有一个这样的消息被取出，系统检查虚键（virtual key）信息是否能转换成等价的字符。如果虚键信息能够转换，TranslateMessage 调用 PostMessage 将一个 WM_CHAR 消息或一个 WM_SYSCHAR 消息放置在登记消息队列中。下次调用 GetMessage 时，系统首先检查登记消息队列中的内容，如果其中有消息存在，从队列中取出消息并将其返回。返回的消息将是 WM_CHAR 消息或 WM_SYSCHAR 消息。再下一次调用 GetMessage 时，系统检查登记消息队列，发现队列已空。系统再检查输入队列，在其中找到 WM_ (SYS) KEYUP 消息。GetMessage 返回这个消息。

由于系统是按这种方式工作，下面的硬件事件序列 WM_KEYDOWN、WM_KEYUP 生成下面的到窗口过程的消息序列（假定虚键信息可以转换成等价的字符）：

```
WM_KEYDOWN
WM_CHAR
WM_KEYUP
```

现在我们再回过头来讨论系统如何确定从 GetMessage 或 PeekMessage 返回的消息。在系统检查了登记消息队列之后，但尚未检查虚拟输入队列时，它要检查 QS_QUIT 标志。我们知道，当线程调用 PostQuitMessage 时设置 QS_QUIT 标志。调用 PostQuitMessage 类似于（但不相同）调用 PostThreadMessage。PostThreadMessage 将消息放置在消息队列的尾端，并使消息在检查输入队列之前被处理。为什么 PostQuitMessage 设置一个标志，而不是将 WM_QUIT 消息放入消息队列中？有两个理由。

第一，在低内存（low memory）情况下，登记一个消息有可能失败。如果一个程序想退出，它应该被允许退出，即使是在低内存的情况下。第二个理由是使用标志可使线程在线程的消息循环结束前完成对所有其他登记消息的处理。例如对下面的代码段，WM_USER 消息将先于 WM_QUIT 消息从队列中取出，尽管 WM_USER 消息是在调用 PostQuitMessage 之后登记到队列中的。

```
case WM_CLOSE:
    PostQuitMessage(0);
    PostMessage(hwnd, WM_USER, 0, 0);
```

最后两个消息是 WM_PAINT 和 WM_TIMER。因为画屏幕是一个慢过程，所以 WM_PAINT 消息的优先级低。如果每当窗口变得无效时就发送一个 WM_PAINT 消息，系统运行就会太慢。在键盘输入之后放置 WM_PAINT 消息，系统会运行得很快。例如，选择一个调用对话框的菜单项，从框中选定一个项，在对话框出现在屏幕上之前一直按 Enter。如果你的按键速度足够快，按键消息总是先于任何 WM_PAINT 消息从队列中取出。当按 Enter 接受对话框的选项，对话框窗口被清除，系统复位 QS_PAINT 标志。

最后一个消息 WM_TIMER，比 WM_PAINT 的优先级还低。为理解这一点，想一想有一个程序用每个 WM_TIMER 消息来更新它的显示画屏。如果定时器消息来的太快，则显示画屏就

没有机会重画自己。在 WM_TIMER消息之前先处理 WM_PAINT消息，就可以避免这个问题，程序总能更新它的显示画屏。

注意 要记住 GetMessage或 PeekMessage 函数只检查唤醒标志和调用线程的消息队列。这意味着一个线程不能从与其他线程挂接的队列中取得消息，包括同一进程内其他线程的消息。

26.4.3 利用内核对象或队列状态标志唤醒线程

GetMessage或 PeekMessage 函数导致一个线程睡眠，直到该线程需要处理一个与用户界面 (UI) 相关的任务。有时候，若能让线程被唤醒去处理一个与 UI 有关的任务或其他任务，就会带来许多方便。例如，一个线程可能启动一个长时间运行的操作，并可以让用户取消这个操作。这个线程需要知道何时操作结束 (与 UI 无关的任务)，或用户是否按了 Cancel 按钮 (与 UI 相关的任务) 来结束操作。

一个线程可以调用 MsgWaitForMultipleObjects 或 MsgWaitForMultipleObjectsEx 函数，使线程等待它自己的消息。

```
DWORD MsgWaitForMultipleObjects(  
    DWORD nCount,  
    PHANDLE phObjects,  
    BOOL fWaitAll,  
    DWORD dwMilliseconds,  
    DWORD dwWakeMask);
```

```
DWORD MsgWaitForMultipleObjectsEx(  
    DWORD nCount,  
    PHANDLE phObjects,  
    DWORD dwMilliseconds,  
    DWORD dwWakeMask,  
    DWORD dwFlags);
```

这两个函数类似于 WaitForMultipleObjects 函数 (在第9章讨论过)。不同之处是，当一个内核对象变成有信号状态 (signaled) 或当一个窗口消息需要派送到调用线程建立的窗口时，这两个函数用于线程调度。

在内部，系统只是向内核句柄的数组添加一个事件内核对象。dwWakeMask 参数告诉系统何时让事件成为有信号状态。dwWakeMask 参数的可能取值的合法范围与可传递到 GetQueueStatus 函数的参数值一样。

正常情况下，当 WaitForMultipleObjects 函数返回时，它返回变成有信号状态的对象的索引以满足调用 (WAIT_OBJECT_0 到 WAIT_OBJECT_0 + nCount - 1)。增加 dwWakeMask 参数就如同向调用增加又一个句柄。如果由于唤醒掩码，MsgWaitForMultipleObjects(Ex) 被满足，返回值将是 WAIT_OBJECT_0 + nCount。

这里是一个例子，说明如何调用 MsgWaitForMultipleObjects：

```
MsgWaitForMultipleObjects(0, NULL, TRUE, INFINITE, QS_INPUT);
```

这条语句的意思是没有传递任何同步对象的句柄，因为 nCount 和 phObjects 参数设定了 0 和 NULL。这里让函数等待所有要变成有信号状态的对象，但只指定了一个要等待的对象，参数 fWaitAll 可以变成 FALSE，而不会改变这个调用的作用。这里还告诉系统，程序将等待，不论等多长时间，直到有键盘消息或鼠标消息出现在调用线程的输入队列中。

当你要用 `MsgWaitForMultipleObjects` 函数做某些事的时候，就会发现这个函数缺少许多重要的特性。因此微软不得不又开发了 `MsgWaitForMultipleObjectsEx` 函数。`MsgWaitForMultipleObjectsEx` 是 `MsgWaitForMultipleObjects` 的一个超集（superset）。新的特性是通过 `dwFlags` 参数引进的。对这个参数，可以指定下面标志的任意组合（见表 26-3）。

表26-3 dwFlags 参数的标志

标 志	描 述
MWMO_WAITALL	函数等待所有要变成有信号状态的内核对象及要出现在线程队列中的特定消息。如果没有这个标志，函数等待直到有一个内核对象变成 signaled，或指定的消息出现在线程的队列中
MWMO_ALERTABLE	函数在一个可报警状态等待
MWMO_INPUTAVAILABLE	当任何指定的消息在线程的队列中时，函数醒来（本节后面详细解释）

如果不要任何这些附加的特性，可对参数 `dwFlags` 传递零（0）。

下面是有关 `MsgWaitForMultipleObjects (Ex)` 的一些重要内容：

- 由于这个函数只是向内核句柄的数组增加一个内部事件内核对象，`nCount` 参数的最大值是 `MAXIMUM_WAIT_OBJECT` 减1或63。
- 当对 `fWaitAll` 参数传递 `FALSE` 时，那么当一个内核对象是有信号的（signaled），或当指定的消息类型出现在线程的队列时，函数返回。
- 当对 `fWaitAll` 参数传递 `TRUE` 时，那么当所有内核对象成为有信号状态，并且指定的消息类型出现在线程的队列中时，函数返回。这种行为似乎使许多开发人员感到惊讶。开发人员希望有一种办法，当所有内核对象变成有信号的或者当指定的消息类型出现在线程的队列中时，可以唤醒线程。但没有函数能够这样。
- 当调用这两个函数时，实际是查看是否有指定类型的新消息被放入调用线程的队列。

注意，上述最后一条会使许多开发人员吃惊。这里有一个例子。假定一个线程的队列目前包含有两个按键消息。如果这个线程现在要调用 `MsgWaitForMultipleObjects (Ex)`，其中 `dwWakeMask` 参数设置成 `QS_INPUT`，线程将被唤醒，从队列中取出第一个按键消息，并处理这个消息。现在，如果这个线程要再调用 `MsgWaitForMultipleObjects (Ex)`，线程将不会被唤醒，因为线程的队列中没有“新”的消息。

对开发人员来说，这已变成了一个主要问题，为此微软增加了 `MWMO_INPUTAVAILABLE` 标志，这只用于 `MsgWaitForMultipleObjectsEx`，而不适用于 `MsgWaitForMultipleObjects`。

这里是一个例子，讲述如何适当地编码一个使用 `MsgWaitForMultipleObjectsEx` 的消息循环：

```

BOOL fQuit = FALSE;           // Should the loop terminate?

while (!fQuit) {
    // Wake when the kernel object is signaled OR
    // if we have to process a UI message.
    DWORD dwResult = MsgWaitForMultipleObjectsEx(1, &hEvent,
        INFINITE, QS_ALLEVENTS, MWMO_INPUTAVAILABLE);

    switch (dwResult) {
        case WAIT_OBJECT_0:    // The event became signaled.
            break;

        case WAIT_OBJECT_0 + 1: // A message is in our queue.
    
```



```
// Dispatch all of the messages.
MSG msg;
while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {

    if (msg.message == WM_QUIT) {
        // A WM_QUIT message, exit the loop
        fQuit = TRUE;
    } else {
        // Translate and dispatch the message.
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
} // Our queue is empty.
break;
}
} // End of while loop
```

26.5 通过消息发送数据

本节将讨论系统如何利用窗口消息在进程之间传送数据。一些窗口消息在其 lParam 参数中指出了内存块的地址。例如，WM_SETTEXT 消息使用 lParam 参数作为指向一个以零结尾的字符串的指针，这个字符串为窗口规定了新的文本标题串。考虑下面的调用：

```
SendMessage(FindWindow(NULL, "Calculator"), WM_SETTEXT,
0, (LPARAM) "A Test Caption");
```

这个调用看起来不会有害。它确定 Calculator 程序窗口的窗口句柄，并试图将窗口的标题改成“A Test Caption”。但我们要仔细看一看究竟会发生什么。

新标题的字符串包含在调用进程的地址空间里。所以这个在调用进程空间的字符串的地址将传递给 lParam 参数。当 Calculator 的窗口的窗口过程收到这个消息时，它要查看 lParam 参数，并要读取这个以零结尾的字符串，使其成为新的标题。

但 lParam 中的地址指向调用进程的地址空间中的字符串，而不是 calculator 的地址空间。这会发生内存存取违规这种严重问题。但当你执行上面的代码时，你会看到执行是成功的，为什么会是这样？

答案是系统特别要检查 WM_SETTEXT 消息，并用与处理其他消息不同的方法来处理这个消息。当调用 SendMessage 时，函数中的代码要检查是否要发送一个 WM_SETTEXT 消息。如果是，就将以零结尾的字符串从调用进程的地址空间放入到一个内存映像文件中，该内存映像文件可在进程间共享。然后再发送消息到其他进程的线程。当接收线程已准备好处理 WM_SETTEXT 消息时，它在自己的地址空间中确定包含新的窗口文本标题的共享内存映像文件的位置，再将 WM_SETTEXT 消息派送到相应的窗口过程。在处理完消息之后，内存映像文件被删除。这样做看起来是不是太麻烦了一些。

幸而大多数消息不要求这种类型的处理。仅当这种消息是程序在进程间发送的消息，特别是消息的 wParam 或 lParam 参数表示一个指向数据结构的指针时，要做这样的处理。

我们再来看另外一个要求系统特殊处理的例子——WM_GETTEXT 消息。假定一个程序包含下面的代码：

```
char szBuf[200];
SendMessage(FindWindow(NULL, "Calculator"), WM_GETTEXT,
sizeof(szBuf), (LPARAM) szBuf);
```

WM_GETTEXT 消息请求 Calculator 的窗口过程用该窗口的标题填充 szBuf 所指定的缓冲区。当一个进程向另一个进程的窗口发送这个消息时，系统实际上必须发送两个消息。首先，系统

要向那个窗口发送一个 WM_GETTEXTLENGTH 消息。窗口过程通过返回窗口标题的字符数来响应这个消息。系统利用这个数字来建立一个内存映像文件，用于在两个进程之间共享。

当内存映像文件被建立时，系统就发送消息来填充它。然后系统再转回到最初调用 SendMessage 的进程，从共享内存映像文件中将数据复制到 szBuf 所指定的缓冲区中，然后从 SendMessage 调用返回。

对于系统已经知道的消息，发送消息时都可以按相应的方式来处理。如果你要建立自己的 (WM_USER + x) 消息，并从一个进程向另一个进程的窗口发送，那又会怎么样？系统不知道你要用内存映像文件并在发送消息时改变指针。为此，微软建立了一个特殊的窗口消息，WM_COPYDATA 以解决这个问题：

```
COPYDATASTRUCT cds;
SendMessage(hwndReceiver, WM_COPYDATA,
    (WPARAM) hwndSender, (LPARAM) &cds);
```

COPYDATASTRUCT 是一个结构，定义在 WinUser.h 文件中，形式如下面的样子：

```
typedef struct tagCOPYDATASTRUCT {
    ULONG_PTR dwData;
    DWORD cbData;
    PVOID lpData;
} COPYDATASTRUCT;
```

当一个进程要向另一个进程的窗口发送一些数据时，必须先初始化 COPYDATASTRUCT 结构。数据成员 dwData 是一个备用的数据项，可以存放任何值。例如，你有可能向另外的进程发送不同类型或不同类别的数据。可以用这个数据来指出要发送数据的内容。

cbData 数据成员规定了向另外的进程发送的字节数，lpData 数据成员指向要发送的第一个字节。lpData 所指向的地址，当然在发送进程的地址空间中。

当 SendMessage 看到要发送一个 WM_COPYDATA 消息时，它建立一个内存映像文件，大小是 cbData 字节，并从发送进程的地址空间中向这个内存映像文件中复制数据。然后再向目的窗口发送消息。在接收消息的窗口过程处理这个消息时，lParam 参数指向已在接收进程地址空间的一个 COPYDATASTRUCT 结构。这个结构的 lpData 成员指向接收进程地址空间中的共享内存映像文件的视图。

关于 WM_COPYDATA 消息，应该注意三个重要问题：

- 只能发送这个消息，不能登记这个消息。不能登记一个 WM_COPYDATA 消息，因为在接收消息的窗口过程处理完消息之后，系统必须释放内存映像文件。如果登记这个消息，系统不知道这个消息何时被处理，所以也不能释放复制的内存块。
- 系统从另外的进程的地址空间中复制数据要花费一些时间。所以不应该让发送程序中运行的其他线程修改这个内存块，直到 SendMessage 调用返回。
- 利用 WM_COPYDATA 消息，可以实现 16 位和 32 位之间的通信。它也能实现 32 位与 64 位之间的通信。这是使新程序同旧程序交流的便捷方法。注意在 Windows 2000 和 Windows 98 上完全支持 WM_COPYDATA。但如果你依然在编写 16 位 Windows 程序，Microsoft Visual C++ 1.52 没有 WM_COPYDATA 消息的定义，也没有 COPYDATASTRUCT 结构的定义。需要手工添加这些代码：

```
// Manually include this in your 16-bit Windows source code.
#define WM_COPYDATA    0x004A
```

```
typedef VOID FAR* PVOID;
typedef struct tagCOPYDATASTRUCT {
    DWORD dwData;
```

```

DWORD cbData;
PVOID lpData;
} COPYDATASTRUCT, FAR* PCOPYDATASTRUCT;

```

在解决进程间的通信问题方面，WM_COPYDATA消息是一个非常好的工具，可以节省程序员的许多时间。关于使用 WM_COPYDATA消息的一个精采例子，见第 22章的 LastMsg BoxInfo示例程序。

CopyData示例程序

清单26-1所列的CopyData程序（“26 CopyData.exe”）说明了如何使用WM_COPYDATA消息从一个程序向另一个程序发送一个数据块。该程序的源代码和资源文件在本书所附光盘的26-CopyData目录下。要看它如何工作，至少需要让两个CopyData的实例运行。每次启动一个CopyData时，它要显示如图26-3所示的对话框。

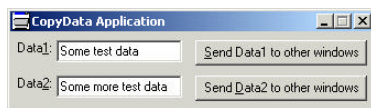


图26-3 CopyData Application 对话框

为了观看从一个程序到另一个程序的数据复制，首先改变 Data1和Data2编辑控制框中的文本。然后点击某个 Send Data* to Other Windows按钮，程序向所有运行的CopyData的实例发送数据。每个实例更新自己的编辑框中的内容来反应新数据。

下面描述CopyData如何工作。当一个用户点击图 26-3中两个按钮中的某一个时，CopyData执行下面的动作。

1) 如果用户点击 Send Data1 To Other Windows按钮，用0来初始化COPYDATASTRUCT的 dwData成员，如果用户点击 Send Data2 To Other Windows按钮，则用1来初始化dwData成员。

2) 从相应的文本框中求取文本串的长度（按字符数计），并加1（对应一个零结束符）。这个值乘以Sizeof(TCHAR)，从字符数转换成字节数。结果存入 COPYDATASTRUCT的cbData成员中。

3) 调用_alloca分配一个内存块，大小足以容纳编辑框中的字符串加上零结束符。这个块的地址存放在 COPYDATASTRUCT结构的lpData成员中。

4) 从编辑框向分配的内存块复制字符串。

这个时候，一切就绪，准备向其他窗口发送数据。为了确定要向哪个窗口发送 WM_COPYDATA消息，CopyData调用FindWindowEx函数传递它自己的对话框标题，以便只有其他的CopyData程序实例才会被枚举。当找到每个实例的窗口时，发送 WM_COPYDATA消息，每个实例更新它的编辑控制框。

清单26-1 CopyData示例程序



CopyData.cpp

```

/*****
Module: CopyData.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
*****/

#include "..\CmnHdr.h" /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>

```

```

#include <malloc.h>
#include "Resource.h"

/////////////////////////////////////////////////////////////////

// WindowsX.h doesn't have a prototype for CIs_OnCopyData, so here it is.
/* BOOL CIs_OnCopyData(HWND hwnd, HWND hwndFrom, PCOPYDATASTRUCT pcds) */

/////////////////////////////////////////////////////////////////

BOOL Dlg_OnCopyData(HWND hwnd, HWND hwndFrom, PCOPYDATASTRUCT cds) {

    Edit_SetText(GetDlgItem(hwnd, cds->dwData ? IDC_DATA2 : IDC_DATA1),
        (PTSTR) cds->lpData);

    return(TRUE);
}

/////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_COPYDATA);

    // Initialize the edit controls with some test data.
    Edit_SetText(GetDlgItem(hwnd, IDC_DATA1), TEXT("Some test data"));
    Edit_SetText(GetDlgItem(hwnd, IDC_DATA2), TEXT("Some more test data"));
    return(TRUE);
}

/////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    switch (id) {
        case IDCANCEL:
            EndDialog(hwnd, id);
            break;

        case IDC_COPYDATA1:
        case IDC_COPYDATA2:
            if (codeNotify != BN_CLICKED)
                break;

            HWND hwndEdit = GetDlgItem(hwnd,
                (id == IDC_COPYDATA1) ? IDC_DATA1 : IDC_DATA2);

            // Prepare the COPYDATASTRUCT.
            COPYDATASTRUCT cds;

```

```

// Indicate which data field we're sending (0=ID_DATA1, 1=ID_DATA2)
cds.dwData = (DWORD) ((id == IDC_COPYDATA1) ? 0 : 1);

// Get the length (in bytes) of the data block we're sending.
cds.cbData = (Edit_GetTextLength(hwndEdit) + 1) * sizeof(TCHAR);

// Allocate a block of memory to hold the string.
cds.lpData = _alloca(cds.cbData);

// Put the edit control's string in the data block.
Edit_GetText(hwndEdit, (PTSTR) cds.lpData, cds.cbData);

// Get the caption of our window.
TCHAR szCaption[100];
GetWindowText(hwnd, szCaption, chDIMOF(szCaption));

// Enumerate through all the top-level windows with the same caption.
HWND hwndT = NULL;
do {
    hwndT = FindWindowEx(NULL, hwndT, NULL, szCaption);
    if (hwndT != NULL) {
        FORWARD_WM_COPYDATA(hwndT, hwnd, &cds, SendMessage);
    }
} while (hwndT != NULL);
break;
}
}

////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
        chHANDLE_DLGMSG(hwnd, WM_COPYDATA, Dlg_OnCopyData);
    }
    return(FALSE);
}

////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_COPYDATA), NULL, Dlg_Proc);
    return(0);
}

//////////////////////////////////// End of File //////////////////////////////////////

CopyData.rc

//Microsoft Developer Studio generated resource script.

```


[illegible]

```
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
////////////////////////////////////////////////

#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//
////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

26.6 Windows如何处理ANSI / Unicode字符和字符串

Windows 98 Windows 98只支持ANSI窗口类和ANSI窗口过程。

当你注册一个新的窗口类时，必须将负责为这个类处理消息的窗口过程的地址告诉系统。对某些消息（如WM_SETTEXT），消息的lParam参数指向一个字符串。在此之前，为了派送消息，使它被正确地处理，系统需要知道窗口过程要求该字符串是ANSI字符串还是Unicode字符串。

告诉系统一个窗口过程是要求ANSI字符串还是Unicode字符串，实际上取决于注册窗口类时所使用的函数。如果构造WNDCLASS结构，并调用RegisterClassA，系统就认为窗口过程要求所有的字符串和字符都属于ANSI。而用RegisterClassW注册窗口类，则系统就向窗口过程派送Unicode字符串和字符。宏RegisterClass对RegisterClassA和RegisterClassW都做了扩展，究竟代表哪一个要看在编译源模块时是否定义了UNICODE。

如果有了一个窗口句柄，就可以确定窗口过程所要求的字符和字符串类型。这可以通过调用下面的函数实现：

```
BOOL IsWindowUnicode(HWND hwnd);
```

如果这个窗口的窗口过程要求Unicode，这个函数返回TRUE，否则返回FALSE。

如果你建立一个ANSI串，并向一个窗口过程要求Unicode串的窗口发送WM_SETTEXT消息，则系统在发送消息之前，为你自动地转换字符串。很少需要调用IsWindowUnicode函数。

如果你对窗口过程派生子类，系统也会为你执行自动的转换。假定一个编辑控制框的窗口过程要求字符和字符串是Unicode。在你的程序的某处建立了一个编辑控制框，并建立窗口过程的子类，这可以调用

```
LONG_PTR SetWindowLongPtrA(  
    HWND hwnd,  
    int nIndex,  
    LONG_PTR dwNewLong);
```

或

```
LONG_PTR SetWindowLongPtrW(  
    HWND hwnd,  
    int nIndex,  
    LONG_PTR dwNewLong);
```

并将nIndex参数设置成GCLP_WNDPROC，dwNewLong参数设置成子类过程的地址。如果这个子类过程要求ANSI字符和字符串会出现什么情况？这可能引起严重的问题。系统决定怎样转换字符串和字符，要取决于究竟是用上面两个函数中的哪一个来建立子类。如果是调用SetWindowLongPtrA，就是告诉系统新的窗口过程（即子类过程）要接收ANSI字符和字符串。实际上，如果在调用SetWindowLongPtrA之后调用IsWindowUnicode函数，将返回FALSE，表示这个子类的编辑窗口过程不再要求Unicode字符和字符串。

但现在又有一个新的问题：如何能够保证原来的窗口过程得到正确的字符和字符串类型？系统需要有两条信息，才能正确地转换字符和字符串。第一条信息就是字符和字符串当前所具有的形式。这可以通过调用CallWindowProcA或CallWindowProcW来告诉系统：

```
LRESULT CallWindowProcA(  
    WNDPROC wndprcPrev,  
    HWND hwnd,  
    UINT uMsg,  
    WPARAM wParam,  
    LPARAM lParam);
```

```
LRESULT CallWindowProcW(  
    WNDPROC wndprcPrev,  
    HWND hwnd,  
    UINT uMsg,  
    WPARAM wParam,  
    LPARAM lParam);
```

如果子类过程要把ANSI字符串传递给原来的窗口过程，子类过程必须调用CallWindowProcA。如果子类过程要把Unicode字符串传递给原来的窗口过程，则子类过程必须调用CallWindowProcW。

系统需要的第二条信息是原来的窗口过程所要求的字符和字符串类型。系统从原来窗口过程的地址获取这个信息。当调用SetWindowLongPtrA或SetWindowLongPtrW函数时，系统要查看是否使用了一个ANSI子类过程派生了一个Unicode窗口过程，或用一个Unicode子类过程派生了一个ANSI窗口过程。如果没有改变所要求的字符串类型，则SetWindowLongPtr只返回原

先窗口过程的地址。如果改变了窗口过程要求的字符和字符串类型，SetWindowLongPtr不是返回原先窗口过程的实际地址，而是返回一个内部子系统数据结构的句柄。

这个数据结构包含原先窗口过程的地址及一个数值，用来指示窗口过程是要求 ANSI还是要求Unicode字符串。当调用 CallWindowProc时，系统要查看是传递了某个内部数据结构的地址，还是传递了一个窗口过程的地址。如果传递了一个窗口过程的地址，则调用原先的窗口过程，不需要执行字符和字符串转换。

如果传递了一个内部数据结构的句柄，则系统要将字符和字符串转换成适当的类型（ANSI或Unicode），然后调用原先的窗口过程。

