1、Windows事件驱动机制

　　我们当中不少使用VC、 Delphi等作为开发语言的程序员是一步步从DOS下的Basic、C++中走过来的，而且大多在刚开始学习编程时也是先从DOS下的编程环境入手的， 因此在习惯了DOS下的过程驱动形式的顺序程序设计方法后，往往在向Windows下的开发环境转型的过程中会对Windows所采取的事件驱动方式感到 无法适应。因为DOS和Windows这两种操作系统的运行机制是截然不同的，DOS下的任何程序都是使用顺序的、过程驱动的程序设计方法。这种程序都有 一个明显的开始、明显的过程以及一个明显的结束，因此通过程序就能直接控制程序事件或过程的全部顺序。即使是在处理异常时，处理过程也仍然是顺序的、过程 驱动的结构。而Windows的驱动方式则是事件驱动的，即程序的流程不是由事件的顺序来控制，而是由事件的发生来控制，所有的事件是无序的，所为一个程 序员，在编写程序时，并不知道用户会先按下哪个按纽，也就不知道程序先触发哪个消息。因此我们的主要任务就是对正在开发的应用程序要发出的或要接收的消息 进行排序和管理。事件驱动程序设计是密切围绕消息的产生与处理而展开的，一条消息是关于发生的事件的消息。

2、Windows的消息循环

　　Windows操作系统为每一个正在 运行的应用程序保持有一个消息队列。当有事件发生后，Windows并不是将这个激发事件直接送给应用程序，而是先将其翻译成一个Windows消息，然 后再把这个消息加入到这个应用程序的消息队列中去。应用程序需要通过消息循环来接收这些消息。在MFC中使用了对WinAPI进行了很好封装的类库，虽然 可以为编程提供一个面向对象的界面，使Windows程序员能够以面象对象的方式进行编程，把那些进行SDK编程时最繁琐的部分提供给程序员，使之专注于 功能的实现，但是由于引入了很好的封装特性，使我们不能直接操纵部分核心代码。对于消息的循环和接收也只是通过类似于下面的消息映射予以很简单的表示：

　 　BEGIN\_MESSAGE\_MAP(CTEMMSView, CFormView)

　　 //{{AFX\_MSG\_MAP(CTEMMSView)

　　ON\_WM\_LBUTTONDOWN()

　　 ON\_COMMAND(ID\_OPENDATA, OnOpenData)

　　ON\_WM\_TIMER()

　　 ON\_WM\_PAINT()

　　//}}AFX\_MSG\_MAP

　　END\_MESSAGE\_MAP()

　 　虽然上述消息映射在编程过程中处理消息非常简练方便，但显然是难于理解消息是如何参与循环和分发的。因此有必要通过SDK（Software Developers Kit，软件开发工具箱）代码深入到被MFC封装的Windows编程的核心中来研究其具体是如何工作的。在SDK编程中，一般是在Windows应用程 序的入口点WinMain函数中添加处理消息循环的代码以检索Windows送来的消息，然后WinMain再把这些消息分配给相应的窗口函数并处理它 们：

　　……

　　MSG msg; //定义消息名

　　while (GetMessage (&msg, NULL, 0, 0))

　　{

　　TranslateMessage (&msg) ; //翻译消息

　　DispatchMessage (&msg) ; //撤去消息

　　}

　 　return msg.wParam ;

　　上述几句虽然简单但却是所有Windows程序的关键代码，担负着获取、解释和分发消息的 任务，下面就重点对其功能和作用进行分析：

　　MSG结构在头文件中定义如下：

　　typedef struct tagMSG

　　{

　　HWND hwnd;

　　UINT message;

　　 WPARAM wParam;

　　LPARAM lParam;

　　DWORD time;

　　 POINT pt;

　　} MSG, \*PMSG;

　　其数据成员的具体意义如下：

　　hwnd：消息 将要发送到的那个窗口的句柄，用这个参数可以决定让哪个窗口接收消息。

　　message：消息号，它唯一标识了一种消息类型。每种消息 类型都在Windows文件进行了预定义。

　　wParam：一个32位的消息参数，这个值的确切意义取决于消息本身。

　 　lParam：同上。

　　time：消息放入消息队列中的时间，在这个域中写入的并非当时日期，而是从Windows启动后所测量的时 间值。Windows用

　　这个域来使用消息保持正确的顺序。

　　pt：消息放入消息队列时的鼠标坐标。

　 　消息循环以GetMessage调用开始，它从消息队列中取出一个消息。该函数的四个参数可以有控制地获取消息，第一个参数指定要接收消息的MSG结构 的地址，第二个参数表示窗口句柄，一般将其设置为空，表示要获取该应用程序创建的所有窗口的消息；第三、四参数用于指定消息范围。后面三个参数被设置为默 认值，用于接收发送到属于这个应用程序的任何一个窗口的所有消息。在接收到除WM\_QUIT之外的任何一个消息后，GetMessage()返回 TRUE；如果GetMessage收到一个WM\_QUIT消息，则返回FALSE以退出消息循环，终止程序运行。因此，在接收到WM\_QUIT之前，带 有GetMessage()的消息循环可以一直循环下去。当除WM\_QUIT的消息用GetMessage读入后，首先要经过函数 TranslateMessage()对其进行解释，但对大多数消息来说并不起什么作用。这里起关键作用的是DispatchMessage()函数，把 由GetMessage获取的Windows消息传送给在MSG结构中为窗口所指定的窗口过程。在消息处理函数处理完消息之后，代码又循环到开始去接收另 一个消息，这样就完成了一个完整的消息循环。

　　由于Windows操作系统是一种非剥夺式多任务操作系统。只有在应用程序主动交出 CPU控制权后，Windows才能把控制权交给其他应用程序。在消息循环中，一定要有能交出控制的系统函数才能实现协同式多任务操作。能完成该功能的只 有GetMessage、PeekMessage和WaitMessage这三个函数，如果在应用程序中长期不去调用这三个函数之一其他任务则无法执行。 GetMessage函数在找不到等待应用程序处理的消息时，会自动交出控制权，由Windows把CPU的控制权交给其他等待获取控制权的应用程序。由 于任何Windows应用程序都含有一个消息循环，这种隐式交出控制权的方式可以保证合并各个应用程序共享控制权。一旦发往该应用程序的消息到达应用程序 队列，即开始执行GetMessage语句的下一条语句。使用GetMessage函数的消息循环在消息队列中没有消息时将等待，如果需要，可以利用这段 时间进行I/O端口操作等耗时操作，不过需要在消息循环中使用PeekMessage函数来代替GetMessage。使用PeekMessage的方法 同GetMessage类似，下面是一段使用PeekMessage函数的消息循环的典型例子：

　　MSG msg;

　 　BOOL bDone=FALSE;

　　do{

　　 if(PeekMessage(&msg,NULL,0,0,PM\_REMOVE)){

　　 if(msg.message==WM\_QUIT)

　　bDone=TRUE;

　　else{

　　 TranslateMessage(&msg);

　　DispatchMessage(&msg);

　 　}

　　}

　　//无消息处理，进行长时间操作

　　else{

　　……//长时间操作

　 　}

　　}while(!bDone)

　　……

　　无论应用程序消息队列中是否有消 息，PeekMessage函数都立即返回，如果希望等待新消息入队，可以利用无返回值的函数WaitMessage配合PeekMessage进行消息 循环。

　　四、对Windowds消息的处理

　　窗口过程处理消息通常以switch语句开始，对于它要处理的每一条消 息ID都跟有一条case语句，这在功能上同MFC的消息映射有些类似：

　　switch(uMsgId)

　　{

　 　case WM\_TIMER：

　　//对WM\_TIMER定时器消息的处理过程

　　return 0;

　 　case WM\_LBUTTONDOWN：

　　//对WM\_ LBUTTONDOWN鼠标左键单击消息的处理过程

　　 ruturn 0;

　　……

　　default:

　　//其他消息由这个默认处理函数来处理

　 　return DefWindowProc(hwnd,uMsgId,wParam,lParam);

　　}

　　在处 理完消息后必须返回0，这很重要，否则Windows将要不停地重试下去。对于那些在程序中不准备处理的消息，窗口过程会把它们都扔给 DefWindowProc进行缺省处理，而且还要返回那个函数的返回值。在消息传递层次中，可以认为DefWindowProc函数是最顶层的函数。该 函数发出WM\_SYSCOMMAND消息，由系统执行Windows环境中多数窗口所公用的各种通用操作，如更新窗口的正文标题等等。 在MFC下可以用下述部分代码实现与上述SDK代码相同的功能：

　　BEGIN\_MESSAGE\_MAP(CTEMMSView, CFormView)

　　//{{AFX\_MSG\_MAP(CTEMMSView)

　　 ON\_WM\_LBUTTONDOWN()

　　ON\_WM\_TIMER()

　　//}}AFX\_MSG\_MAP

　 　END\_MESSAGE\_MAP()

　　小结：Windows环境提供有非常丰富的系统资源，在这个基础上可以编制出能满足各种各样目 标功能的应用系统。要深入Windows编程就必须首先对Windows系统的运行机理有很好的认识，本文仅针对Windows的一种重要运行机制--消 息机制作了较深入的剖析和阐述。对培养在Windows下的编程思想有一定的帮助。对某些相关问题的详细论述可以参考MSDN在线帮助的"SDK Reference"部分。

1. 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。
2. 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。
3. 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。
4. 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。
5. 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。
6. 这一步是边框窗口自己尝试处理消息。
7. 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，试图得到相应的处理函数
8. 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，函数逐级别返回，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。