# **[windows消息机制（MFC）](https://www.cnblogs.com/findumars/p/3948427.html)**

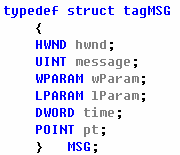
# **[windows消息机制（MFC）](http://blog.csdn.net/kongfuxionghao/article/details/35882533)**

**消息分类与消息队列**

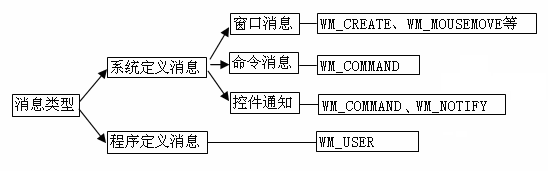
Windows中，消息使用统一的结构体（MSG）来存放信息，其中message表明消息的具体的类型，

而wParam，lParam是其最灵活的两个变量，为不同的消息类型时，存放数据的含义也不一样。

time表示产生消息的时间，pt表示产生消息时鼠标的位置。



**按照类型，Windows将消息分为：**



(0) 消息ID范围

系统定义消息ID范围：[0x0000, 0x03ff]  
用户自定义的消息ID范围：   
WM\_USER: 0x0400-0x7FFF (例：WM\_USER+10)   
WM\_APP(winver> 4.0)：0x8000-0xBFFF (例：WM\_APP+4)   
RegisterWindowMessage：0xC000-0xFFFF【用来和其他应用程序通信，为了ID的唯一性，使用::RegisterWindowMessage来得到该范围的消息ID 】

(1) 窗口消息：即与窗口的内部运作有关的消息，如创建窗口，绘制窗口，销毁窗口等。

     可以是一般的窗口，也可以是MainFrame,Dialog,控件等。

     如：WM\_CREATE, WM\_PAINT, WM\_MOUSEMOVE, WM\_CTLCOLOR, WM\_HSCROLL等

(2) 当用户从菜单选中一个命令项目、按下一个快捷键或者点击工具栏上的一个按钮，都将发送WM\_COMMAND命令消息。

     LOWORD(wParam)表示菜单项，工具栏按钮或控件的ID；如果是控件, HIWORD(wParam)表示控件消息类型。

     #define LOWORD(l) ((WORD)(l))

     #define HIWORD(l) ((WORD)(((DWORD)(l) >> 16) & 0xFFFF))

(3) 随着控件的种类越来越多，越来越复杂（如列表控件、树控件等），仅仅将wParam，lParam将视为一个32位无符号整数，已经装不下太多信息了。

    为了给父窗口发送更多的信息，微软定义了一个新的WM\_NOTIFY消息来扩展WM\_COMMAND消息。

    WM\_NOTIFY消息仍然使用MSG消息结构，只是此时wParam为控件ID，lParam为一个NMHDR指针，

    不同的控件可以按照规则对NMHDR进行扩充，因此WM\_NOTIFY消息传送的信息量可以相当的大。

注：Window 9x 版及以后的新控件通告消息不再通过WM\_COMMAND 传送，而是通过WM\_NOTIFY 传送，  
      但是老控件的通告消息， 比如CBN\_SELCHANGE 还是通过WM\_COMMAND 消息发送。

(4) windwos也允许程序员定义自己的消息，使用SendMessage或PostMessage来发送消息。

**windows消息还可以分为：**

(1) 队列消息(Queued Messages)   
消息会先保存在消息队列中，消息循环会从此队列中取出消息并分发到各窗口处理   
如：WM\_PAINT，WM\_TIMER，WM\_CREATE，WM\_QUIT，以及鼠标，键盘消息等。  
其中，WM\_PAINT，WM\_TIMER只有在队列中没有其他消息的时候才会被处理，  
WM\_PAINT消息还会被合并以提高效率。其他所有消息以先进先出（FIFO）的方式被处理。

(2) 非队列消息(NonQueued Messages)   
消息会绕过系统消息队列和线程消息队列，直接发送到窗口过程进行处理   
如：WM\_ACTIVATE, WM\_SETFOCUS, WM\_SETCURSOR，WM\_WINDOWPOSCHANGED

**Windows系统的整个消息系统分为3个层级：**

    ① Windows内核的系统消息队列

    ② App的UI线程消息队列

    ③ 处理消息的窗体对象

Windows内核维护着一个全局的系统消息队列；按照线程的不同，系统消息队列中的消息会分发到应用程序的UI线程的消息队列中；

应用程序的每一个UI线程都有自己的消息循环，会不停地从自己的消息队列取出消息，并发送给Windows窗体对象；

每一个窗体对象都使用**窗体过程函数（WindowProc）**来处理接收到的各种消息。

[IMG_258](https://www.cnblogs.com/findumars/p/javascript:void(0);)

[IMG_259](http://blog.csdn.net/kongfuxionghao/article/details/35882533)

1 LRESULT CALLBACK WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

2 {

3 PAINTSTRUCT ps;

4 HDC hdc;

5

6 switch (message)

7 {

8 case WM\_COMMAND:

9 break;

10 case WM\_PAINT:

11 hdc = BeginPaint(hWnd, &ps);

12 // TODO: 在此添加任意绘图代码...

13 EndPaint(hWnd, &ps);

14 break;

15 case WM\_DESTROY:

16 PostQuitMessage(0);

17 break;

18 default:

19 return DefWindowProc(hWnd, message, wParam, lParam);

20 }

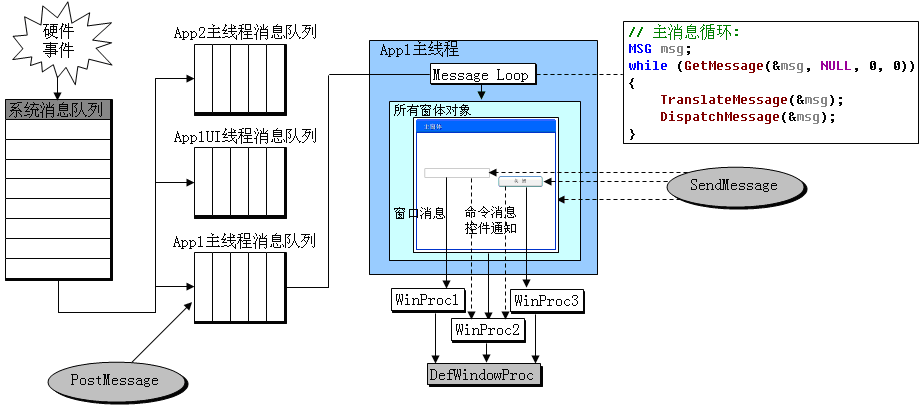
21 return 0;

22 }

[IMG_260](http://blog.csdn.net/kongfuxionghao/article/details/35882533)

[IMG_261](https://www.cnblogs.com/findumars/p/javascript:void(0);)

需要的话，在WindowProc中，可以用::GetMessageTime获取当前消息产生的时间，  
用::GetMessagePos获取当前消息产生时鼠标光标所在的位置。



(1) 各个**窗口消息**由各个窗体（或控件）自身的**WindowProc（虚函数）**接收并处理。

(2) **WM\_COMMAND命令消息**统一由**当前活动主窗口的WindowProc接收**，经过绕行后，可被其他的**CCmdTarget对象**处理。

(3) **WM\_COMMAND控件通知**统一由**子窗口（控件）的父窗口的WindowProc**接收并处理，也可以进行绕行被其他的**CCmdTarget对象**处理。

     （例如：CFormView具备接受WM\_COMMAND控件通知的条件，又具备把WM\_COMMAND消息派发给关联文档对象处理的能力，

         所以给CFormView的WM\_COMMAND控件通知是可以让文档对象处理的。）

     另外，**WM\_COMMAND控制通知**会先调用ReflectLastMsg反射**通知子窗口（控件）**，如果子窗口（控件）处理了该消息并返回TRUE，则消息会停止分发；

     否则，会继续调用OnCmdMsg进行命令发送（如同WM\_COMMAND命令消息一样）。

注：WM\_COMMAND命令消息与WM\_COMMAND控件通知的相似之处：  
WM\_COMMAND命令消息和WM\_COMMAND控制通知都是由WindowProc给OnCommand处理，  
OnCommand通过wParam和lParam参数区分是命令消息或通知消息，然后送给OnCmdMsg处理。  
事实上，BN\_CLICKED控件通知消息的处理和WM\_COMMAND命令消息的处理完全一样。  
因为该消息的通知代码是0，ON\_BN\_CLICKED(id，memberfunction)和ON\_COMMAND(id，memberfunction)是等同的。

（4）**WM\_NOTIFY消息**只是对**WM\_COMMAND控件通知**进行了扩展，与WM\_COMMAND控件通知具有相同的特点。

**SendMessage与PostMessage**

PostMessage 把消息投递到消息队列后，立即返回；   
SendMessage把消息直接送到窗口过程处理，处理完才返回。

**GetMessage与PeekMessage**

GetMessage 有消息且该消息不为WM\_QUIT，返回TRUE。  
　　            有消息且该消息为WM\_QUIT，返回FALSE。  
                  没有消息时，挂起该UI线程，控制权交还给系统。  
PeekMessage 有消息返回TRUE，如果没有消息返回FALSE；不会阻塞。  
                   是否从消息队列中删除此消息（PM\_REMOVE），由函数参数来指定。

要想在没有消息时做一些工作，就必须使用PeekMessage来抓取消息，以便在没有消息时，能在OnIdle中执行空闲操作（如下）：

[IMG_263](https://www.cnblogs.com/findumars/p/javascript:void(0);)

[IMG_264](http://blog.csdn.net/kongfuxionghao/article/details/35882533)

1 while (TRUE)

2 {

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

4 {

5 if (msg.message == WM\_QUIT)

6 break;

7 TranslateMessage(&msg);

8 DispatchMessage(&msg);

9 }

10 else

11 {

12 OnIdle();

13 }

14 }

[IMG_265](http://blog.csdn.net/kongfuxionghao/article/details/35882533)

[IMG_266](https://www.cnblogs.com/findumars/p/javascript:void(0);)

例如：MFC使用OnIdle函数来清理一些临时对象及未使用的动态链接库。

只有在OnIdle返回之后程序才能继续处理用户的输入，因此不应在OnIdle进行较长的任务。

**MFC消息处理**

在CWnd中，MFC使用OnWndMsg来分别处理各类消息：

如果是WM\_COMMAND消息，交给OnCommand处理；然后返回。

如果是WM\_NOTIFY消息，交给OnNotify处理；然后返回。

如果是WM\_ACTIVATE消息，先交给\_AfxHandleActivate处理，再继续下面的处理。

如果是WM\_SETCURSOR消息，先交给\_AfxHandleSetCursor处理，然后返回。

如果是其他的窗口消息（包括WM\_ACTIVATE消息），则

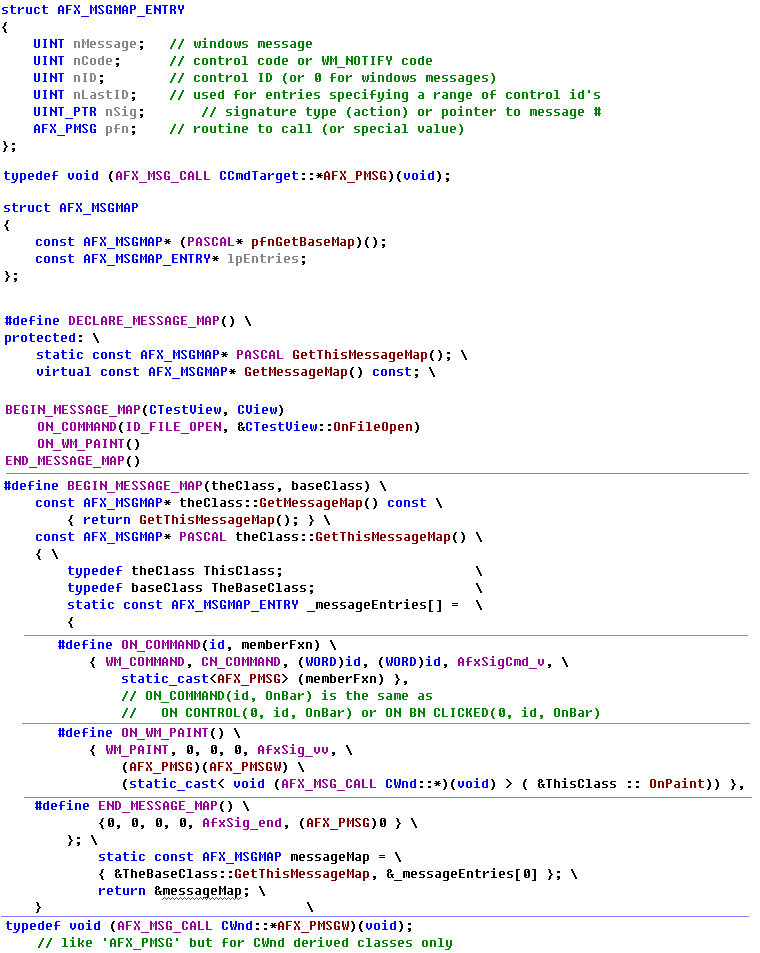
  首先在消息缓冲池（一个hash表，用于加快消息处理函数的查找）进行消息匹配，  
    若匹配成功，则调用相应的消息处理函数；  
    若不成功，则在消息目标的消息映射数组中进行查找匹配，看它是否能处理当前消息。  
  如果消息目标处理了该消息，则会匹配到消息处理函数，调用它进行处理；

否则，该消息没有被应用程序处理，OnWndMsg返回FALSE。

**MFC消息映射**

消息映射实际是MFC内建的一个消息分派机制。

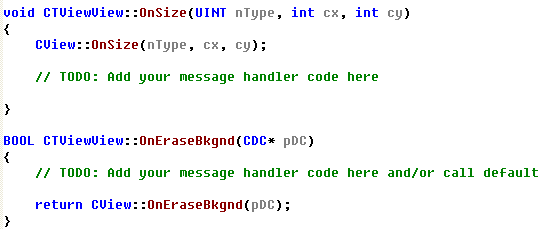
把MFC中的宏进行展开（如下），可以得到消息映射表整个全貌。



注：GetMessageMap为虚函数。  
     {0, 0, 0, 0, AfxSig\_end, (AFX\_PMSG)0}：对象消息映射表的结束标识

**窗口消息**只能由**CWnd对象**来处理，采用向**基类直线上朔**的方式，来查找对应的消息响应函数进行处理。

一旦找到消息响应函数（若有返回值且为TRUE），就停止上朔。因此，我们经常会看到这样的代码：

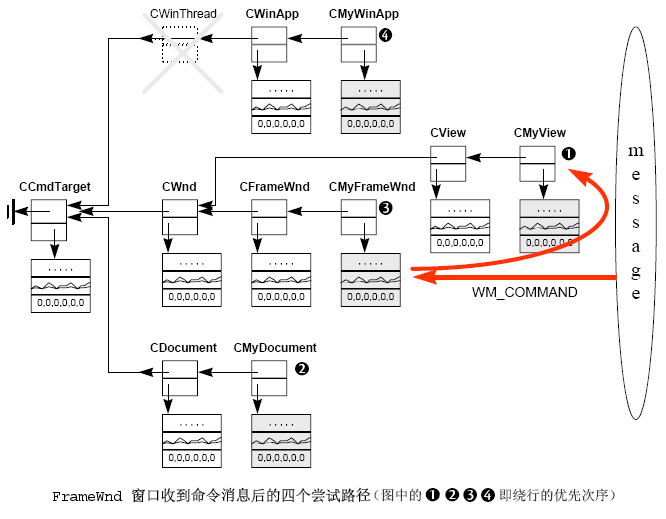


增加一个消息处理函数来写我们的逻辑时，MFC ClassWizard会在该函数之前或之后显示调用其基类对应的函数，保证基类中逻辑被执行。

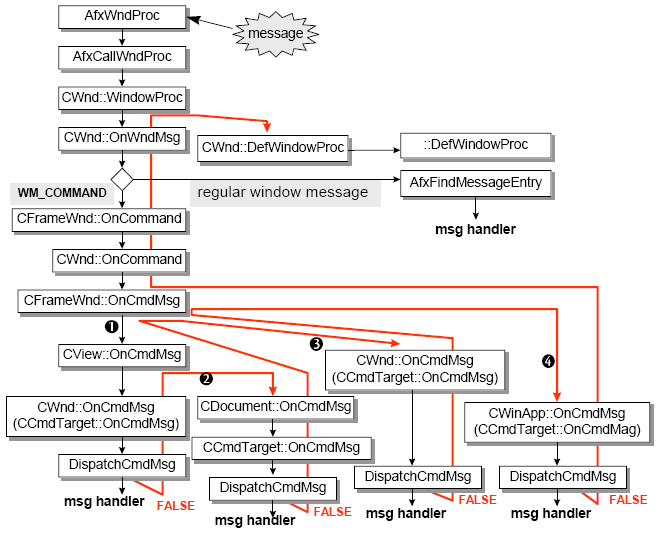
**命令消息**可由**CCmdTarget对象**接收并处理（OnCmdMsg为虚函数），除了向基类直线上朔方式外，还有**命令绕行机制**（要防止形成圈，死循环）。

在某种程度上，**控制通知消息**由窗口对象处理是一种习惯和约定。然而，控件通知消息也是可以有**CCmdTarget对象**接收并处理，并进行**命令绕行**的。

**下图为MFC经典单文档视图框架的命令消息绕行路线：**



函数调用过程如下（如果没有任何对象处理该条WM\_COMMAND消息，最后会被::DefWindowProc处理）。



**非模态对话框的消息处理**

1 static CAboutDlg aboutDlg;

2 aboutDlg.Create(IDD\_ABOUTBOX, this);

3 aboutDlg.ShowWindow(SW\_SHOW);

应用程序只有一个消息循环。

对于窗口消息，非模态对话框（及其子控件）与父窗口（及其子控件）都是用自身的WindowProc函数接收并处理，互不干扰。

对于命令消息，由当前活动主窗口的WindowProc接收（例如：当前活动主窗口为非模态对话框，则命令消息会被非模态对话框接收）。

可以在当前活动主窗口的OnCmdMsg中做命令绕行，使得其他的CCmdTarget对象也可以处理命令消息。

对于控件通知，由其父窗口的WindowProc接收并处理，一般不进行命令绕行被其他的CCmdTarget对象处理。

**模态对话框的消息处理**

1 CAboutDlg aboutDlg;

2 aboutDlg.DoModal();

(1) 模态对话框弹出来后，首先会让父窗口失效，使其不能接受用户的输入（键盘鼠标消息）。

1 EnableWindow(hwndParent, FALSE) ;

(2) 父窗口消息循环被阻塞（会卡在DoModal处，等待返回），由模态对话框的消息循环来接管（因此整个程序不会卡住）。

    接管后，模态对话框的消息循环仍然会将属于父窗口及其子控件的窗口消息（不包括键盘鼠标相关的窗口消息）发送给它们各自的WindowProc窗口函数，进行响应处理。

(3) 模态对话框销毁时（点击IDOK或IDCANCEL），父窗口消息循环重新激活，继续DoModal后的逻辑。

    激活后，父窗口有可以重新接受用户的输入（键盘鼠标消息）。

1 EnableWindow(hwndParent, TRUE) ;

从上面的过程中，我们可以得到如下结论：

对于窗口消息，模态对话框主窗口（及其子控件）与父窗口（及其子控件）都是用自身的WindowProc函数接收并处理，互不干扰。

只是父窗口（及其子控件）无法接受到键盘鼠标消息相关的窗口消息。

对于命令消息，由模态对话框主窗口的WindowProc接收。可以在模态对话框主窗口的OnCmdMsg中做命令绕行，使得其他的CCmdTarget对象也可以处理命令消息。

对于控件通知，由其父窗口的WindowProc接收并处理，一般不进行命令绕行被其他的CCmdTarget对象处理。

[windows消息机制（MFC）](https://www.cnblogs.com/findumars/p/3948427.html)

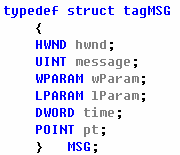
[windows消息机制（MFC）](http://blog.csdn.net/kongfuxionghao/article/details/35882533)

消息分类与消息队列

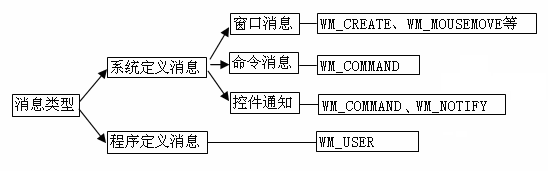
Windows中，消息使用统一的结构体（MSG）来存放信息，其中message表明消息的具体的类型，

而wParam，lParam是其最灵活的两个变量，为不同的消息类型时，存放数据的含义也不一样。

time表示产生消息的时间，pt表示产生消息时鼠标的位置。



按照类型，Windows将消息分为：



(0) 消息ID范围

系统定义消息ID范围：[0x0000, 0x03ff]  
用户自定义的消息ID范围：   
WM\_USER: 0x0400-0x7FFF (例：WM\_USER+10)   
WM\_APP(winver> 4.0)：0x8000-0xBFFF (例：WM\_APP+4)   
RegisterWindowMessage：0xC000-0xFFFF【用来和其他应用程序通信，为了ID的唯一性，使用::RegisterWindowMessage来得到该范围的消息ID 】

(1) 窗口消息：即与窗口的内部运作有关的消息，如创建窗口，绘制窗口，销毁窗口等。

     可以是一般的窗口，也可以是MainFrame,Dialog,控件等。

     如：WM\_CREATE, WM\_PAINT, WM\_MOUSEMOVE, WM\_CTLCOLOR, WM\_HSCROLL等

(2) 当用户从菜单选中一个命令项目、按下一个快捷键或者点击工具栏上的一个按钮，都将发送WM\_COMMAND命令消息。

     LOWORD(wParam)表示菜单项，工具栏按钮或控件的ID；如果是控件, HIWORD(wParam)表示控件消息类型。

     #define LOWORD(l) ((WORD)(l))

     #define HIWORD(l) ((WORD)(((DWORD)(l) >> 16) & 0xFFFF))

(3) 随着控件的种类越来越多，越来越复杂（如列表控件、树控件等），仅仅将wParam，lParam将视为一个32位无符号整数，已经装不下太多信息了。

    为了给父窗口发送更多的信息，微软定义了一个新的WM\_NOTIFY消息来扩展WM\_COMMAND消息。

    WM\_NOTIFY消息仍然使用MSG消息结构，只是此时wParam为控件ID，lParam为一个NMHDR指针，

    不同的控件可以按照规则对NMHDR进行扩充，因此WM\_NOTIFY消息传送的信息量可以相当的大。

注：Window 9x 版及以后的新控件通告消息不再通过WM\_COMMAND 传送，而是通过WM\_NOTIFY 传送，  
      但是老控件的通告消息， 比如CBN\_SELCHANGE 还是通过WM\_COMMAND 消息发送。

(4) windwos也允许程序员定义自己的消息，使用SendMessage或PostMessage来发送消息。

windows消息还可以分为：

(1) 队列消息(Queued Messages)   
消息会先保存在消息队列中，消息循环会从此队列中取出消息并分发到各窗口处理   
如：WM\_PAINT，WM\_TIMER，WM\_CREATE，WM\_QUIT，以及鼠标，键盘消息等。  
其中，WM\_PAINT，WM\_TIMER只有在队列中没有其他消息的时候才会被处理，  
WM\_PAINT消息还会被合并以提高效率。其他所有消息以先进先出（FIFO）的方式被处理。

(2) 非队列消息(NonQueued Messages)   
消息会绕过系统消息队列和线程消息队列，直接发送到窗口过程进行处理   
如：WM\_ACTIVATE, WM\_SETFOCUS, WM\_SETCURSOR，WM\_WINDOWPOSCHANGED

Windows系统的整个消息系统分为3个层级：

    ① Windows内核的系统消息队列

    ② App的UI线程消息队列

    ③ 处理消息的窗体对象

Windows内核维护着一个全局的系统消息队列；按照线程的不同，系统消息队列中的消息会分发到应用程序的UI线程的消息队列中；

应用程序的每一个UI线程都有自己的消息循环，会不停地从自己的消息队列取出消息，并发送给Windows窗体对象；

每一个窗体对象都使用窗体过程函数（WindowProc）来处理接收到的各种消息。

[IMG_258](https://www.cnblogs.com/findumars/p/javascript:void(0);)

[IMG_259](http://blog.csdn.net/kongfuxionghao/article/details/35882533)

1 LRESULT CALLBACK WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

2 {

3 PAINTSTRUCT ps;

4 HDC hdc;

5

6 switch (message)

7 {

8 case WM\_COMMAND:

9 break;

10 case WM\_PAINT:

11 hdc = BeginPaint(hWnd, &ps);

12 // TODO: 在此添加任意绘图代码...

13 EndPaint(hWnd, &ps);

14 break;

15 case WM\_DESTROY:

16 PostQuitMessage(0);

17 break;

18 default:

19 return DefWindowProc(hWnd, message, wParam, lParam);

20 }

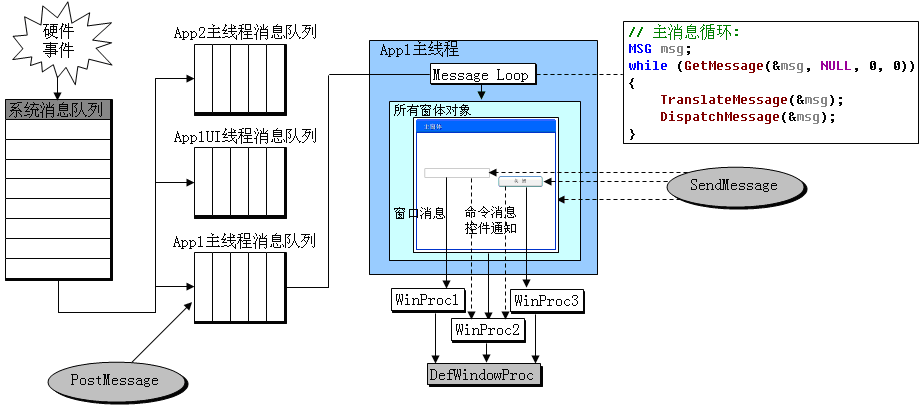
21 return 0;

22 }

[IMG_260](http://blog.csdn.net/kongfuxionghao/article/details/35882533)

[IMG_261](https://www.cnblogs.com/findumars/p/javascript:void(0);)

需要的话，在WindowProc中，可以用::GetMessageTime获取当前消息产生的时间，  
用::GetMessagePos获取当前消息产生时鼠标光标所在的位置。



(1) 各个窗口消息由各个窗体（或控件）自身的WindowProc（虚函数）接收并处理。

(2) WM\_COMMAND命令消息统一由当前活动主窗口的WindowProc接收，经过绕行后，可被其他的CCmdTarget对象处理。

(3) WM\_COMMAND控件通知统一由子窗口（控件）的父窗口的WindowProc接收并处理，也可以进行绕行被其他的CCmdTarget对象处理。

     （例如：CFormView具备接受WM\_COMMAND控件通知的条件，又具备把WM\_COMMAND消息派发给关联文档对象处理的能力，

         所以给CFormView的WM\_COMMAND控件通知是可以让文档对象处理的。）

     另外，WM\_COMMAND控制通知会先调用ReflectLastMsg反射通知子窗口（控件），如果子窗口（控件）处理了该消息并返回TRUE，则消息会停止分发；

     否则，会继续调用OnCmdMsg进行命令发送（如同WM\_COMMAND命令消息一样）。

注：WM\_COMMAND命令消息与WM\_COMMAND控件通知的相似之处：  
WM\_COMMAND命令消息和WM\_COMMAND控制通知都是由WindowProc给OnCommand处理，  
OnCommand通过wParam和lParam参数区分是命令消息或通知消息，然后送给OnCmdMsg处理。  
事实上，BN\_CLICKED控件通知消息的处理和WM\_COMMAND命令消息的处理完全一样。  
因为该消息的通知代码是0，ON\_BN\_CLICKED(id，memberfunction)和ON\_COMMAND(id，memberfunction)是等同的。

（4）WM\_NOTIFY消息只是对WM\_COMMAND控件通知进行了扩展，与WM\_COMMAND控件通知具有相同的特点。

SendMessage与PostMessage

PostMessage 把消息投递到消息队列后，立即返回；   
SendMessage把消息直接送到窗口过程处理，处理完才返回。

GetMessage与PeekMessage

GetMessage 有消息且该消息不为WM\_QUIT，返回TRUE。  
　　            有消息且该消息为WM\_QUIT，返回FALSE。  
                  没有消息时，挂起该UI线程，控制权交还给系统。  
PeekMessage 有消息返回TRUE，如果没有消息返回FALSE；不会阻塞。  
                   是否从消息队列中删除此消息（PM\_REMOVE），由函数参数来指定。

要想在没有消息时做一些工作，就必须使用PeekMessage来抓取消息，以便在没有消息时，能在OnIdle中执行空闲操作（如下）：

[IMG_263](https://www.cnblogs.com/findumars/p/javascript:void(0);)

[IMG_264](http://blog.csdn.net/kongfuxionghao/article/details/35882533)

1 while (TRUE)

2 {

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

4 {

5 if (msg.message == WM\_QUIT)

6 break;

7 TranslateMessage(&msg);

8 DispatchMessage(&msg);

9 }

10 else

11 {

12 OnIdle();

13 }

14 }

[IMG_265](http://blog.csdn.net/kongfuxionghao/article/details/35882533)

[IMG_266](https://www.cnblogs.com/findumars/p/javascript:void(0);)

例如：MFC使用OnIdle函数来清理一些临时对象及未使用的动态链接库。

只有在OnIdle返回之后程序才能继续处理用户的输入，因此不应在OnIdle进行较长的任务。

MFC消息处理

在CWnd中，MFC使用OnWndMsg来分别处理各类消息：

如果是WM\_COMMAND消息，交给OnCommand处理；然后返回。

如果是WM\_NOTIFY消息，交给OnNotify处理；然后返回。

如果是WM\_ACTIVATE消息，先交给\_AfxHandleActivate处理，再继续下面的处理。

如果是WM\_SETCURSOR消息，先交给\_AfxHandleSetCursor处理，然后返回。

如果是其他的窗口消息（包括WM\_ACTIVATE消息），则

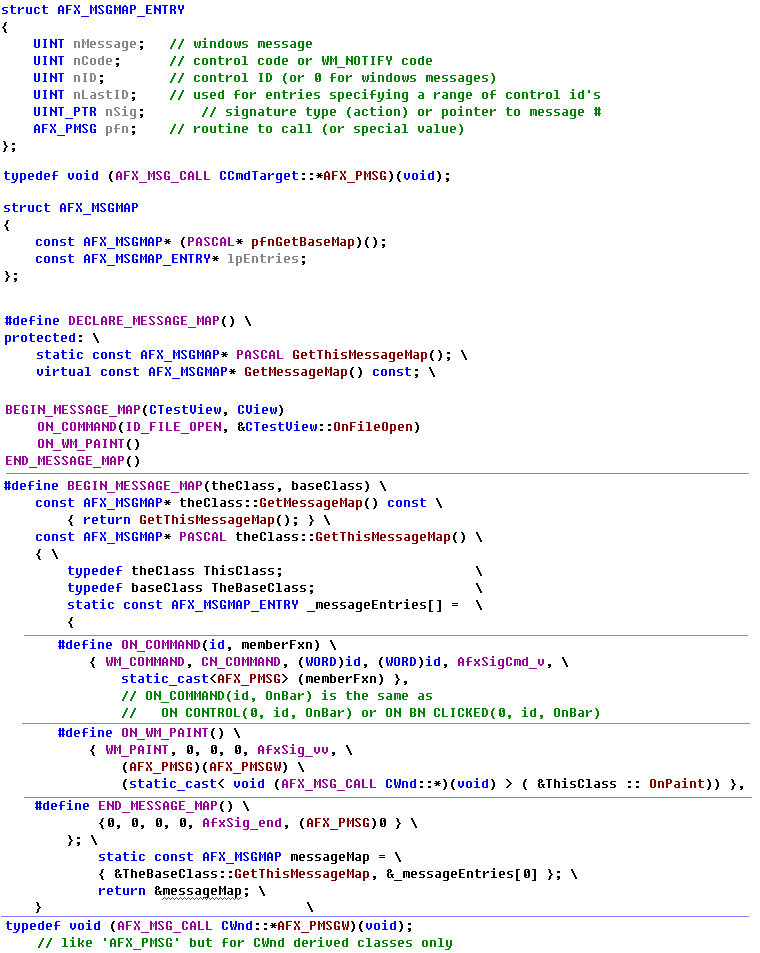
  首先在消息缓冲池（一个hash表，用于加快消息处理函数的查找）进行消息匹配，  
    若匹配成功，则调用相应的消息处理函数；  
    若不成功，则在消息目标的消息映射数组中进行查找匹配，看它是否能处理当前消息。  
  如果消息目标处理了该消息，则会匹配到消息处理函数，调用它进行处理；

否则，该消息没有被应用程序处理，OnWndMsg返回FALSE。

MFC消息映射

消息映射实际是MFC内建的一个消息分派机制。

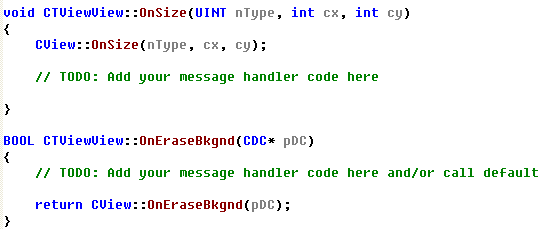
把MFC中的宏进行展开（如下），可以得到消息映射表整个全貌。



注：GetMessageMap为虚函数。  
     {0, 0, 0, 0, AfxSig\_end, (AFX\_PMSG)0}：对象消息映射表的结束标识

窗口消息只能由CWnd对象来处理，采用向基类直线上朔的方式，来查找对应的消息响应函数进行处理。

一旦找到消息响应函数（若有返回值且为TRUE），就停止上朔。因此，我们经常会看到这样的代码：

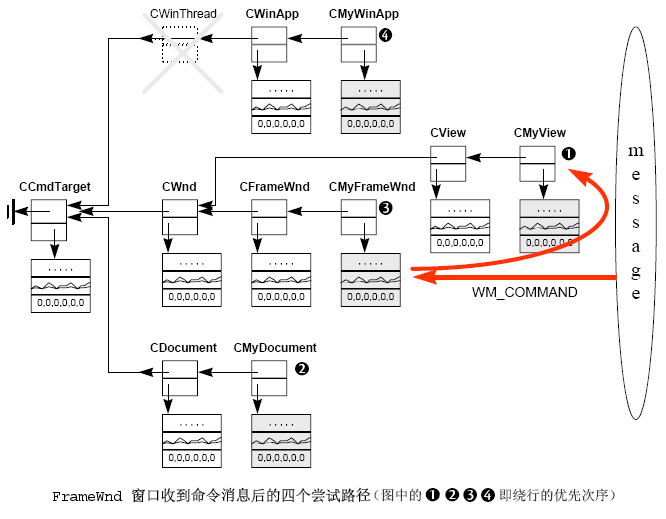


增加一个消息处理函数来写我们的逻辑时，MFC ClassWizard会在该函数之前或之后显示调用其基类对应的函数，保证基类中逻辑被执行。

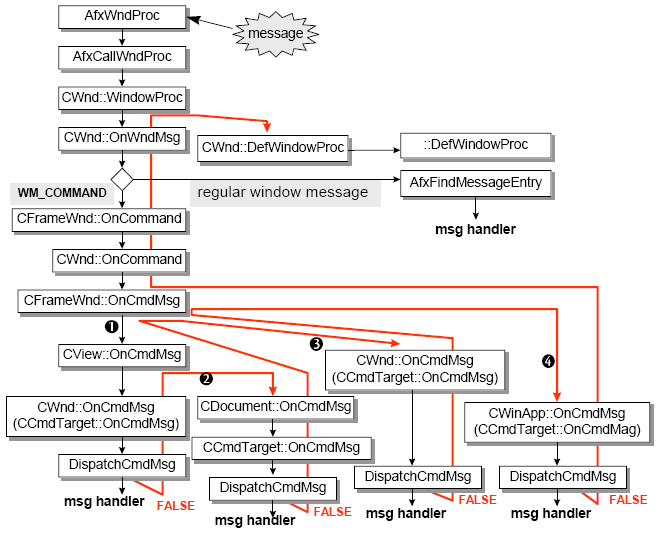
命令消息可由CCmdTarget对象接收并处理（OnCmdMsg为虚函数），除了向基类直线上朔方式外，还有命令绕行机制（要防止形成圈，死循环）。

在某种程度上，控制通知消息由窗口对象处理是一种习惯和约定。然而，控件通知消息也是可以有CCmdTarget对象接收并处理，并进行命令绕行的。

下图为MFC经典单文档视图框架的命令消息绕行路线：



函数调用过程如下（如果没有任何对象处理该条WM\_COMMAND消息，最后会被::DefWindowProc处理）。



非模态对话框的消息处理

1 static CAboutDlg aboutDlg;

2 aboutDlg.Create(IDD\_ABOUTBOX, this);

3 aboutDlg.ShowWindow(SW\_SHOW);

应用程序只有一个消息循环。

对于窗口消息，非模态对话框（及其子控件）与父窗口（及其子控件）都是用自身的WindowProc函数接收并处理，互不干扰。

对于命令消息，由当前活动主窗口的WindowProc接收（例如：当前活动主窗口为非模态对话框，则命令消息会被非模态对话框接收）。

可以在当前活动主窗口的OnCmdMsg中做命令绕行，使得其他的CCmdTarget对象也可以处理命令消息。

对于控件通知，由其父窗口的WindowProc接收并处理，一般不进行命令绕行被其他的CCmdTarget对象处理。

模态对话框的消息处理

1 CAboutDlg aboutDlg;

2 aboutDlg.DoModal();

(1) 模态对话框弹出来后，首先会让父窗口失效，使其不能接受用户的输入（键盘鼠标消息）。

1 EnableWindow(hwndParent, FALSE) ;

(2) 父窗口消息循环被阻塞（会卡在DoModal处，等待返回），由模态对话框的消息循环来接管（因此整个程序不会卡住）。

    接管后，模态对话框的消息循环仍然会将属于父窗口及其子控件的窗口消息（不包括键盘鼠标相关的窗口消息）发送给它们各自的WindowProc窗口函数，进行响应处理。

(3) 模态对话框销毁时（点击IDOK或IDCANCEL），父窗口消息循环重新激活，继续DoModal后的逻辑。

    激活后，父窗口有可以重新接受用户的输入（键盘鼠标消息）。

1 EnableWindow(hwndParent, TRUE) ;

从上面的过程中，我们可以得到如下结论：

对于窗口消息，模态对话框主窗口（及其子控件）与父窗口（及其子控件）都是用自身的WindowProc函数接收并处理，互不干扰。

只是父窗口（及其子控件）无法接受到键盘鼠标消息相关的窗口消息。

对于命令消息，由模态对话框主窗口的WindowProc接收。可以在模态对话框主窗口的OnCmdMsg中做命令绕行，使得其他的CCmdTarget对象也可以处理命令消息。

对于控件通知，由其父窗口的WindowProc接收并处理，一般不进行命令绕行被其他的CCmdTarget对象处理。