

附录B 消息分流器、子控件宏和API宏

每当我参加一些会议时，常问一些人是不是使用消息分流器，而回答通常是“ No ”。我再进一步深究这件事，发现很多人不知道消息分流器是干什么用的，甚至没有听说过它。在本书中，通过使用带有消息分流器的 C/C++ 编写示例代码，我想向大家介绍这种不大为人所知但很有用的宏。

消息分流器定义在 Microsoft Visual C++ 中提供的 WindowsX.h 文件里。通常在 Windows.h 文件之后紧接着包含这个文件。WindowsX.h 文件就是一组 #define 指令，建立了一组供我们使用的宏。WindowsX.h 的宏实际上分为三组：消息分流器、子控件宏和 API 宏。这些宏以下述的方式为我们提供帮助：

- 利用这些宏可以减少程序中要做的转换（ casting ）的数量，并可使所要求的转换是无错误的。使用 C/C++ 的 Windows 编程中一个大的问题是所要求的转换数量。你很难看到一个不要求某种转换的 Windows 函数调用。但应该尽量避免使用转换，因为转换阻碍编译器发现代码中的潜在错误。一个转换是在告诉编译程序：“ 我知道我在这里传递了错误的转换，但就要这样做。我知道我在干什么。” 当你做了许多转换时，就很容易出错。编译程序应该尽可能对此提供帮助。
- 使代码的可读性更好。
- 可简化 16 位 Windows、32 位 Windows 和 64 位 Windows 之间的代码移植工作。
- 易于理解（ 只是一些宏 ）
- 这些宏容易结合到已有的代码中。可以不管老的代码而立即在新的代码中使用这些宏。不必修改整个程序。
- 在 C 和 C++ 代码中都可以使用这些宏，尽管当使用 C++ 类时它们不是必需的。
- 如果需要某一个特性，而这些宏不直接支持这个特性，可以根据这个头文件中的宏，很容易地编写自己的宏。
- 不需要参照或记住费解的 Windows 构造。例如，许多 Windows 中的函数，要求一个 long 型参数，其中这个长参数的高字（ high-word ）的值代表一个东西，而其低字（ low-word ）又代表另一个东西。在调用这个函数之前，你必须用两个单独的值构造一个 long 型值。通常利用 WinDef.h 中的 MAKELONG 宏来做这种事。我简直记不清有多少次把两个值的次序给弄反了，造成对函数传递了一个错误的值。而 WindowsX.h 中的宏可以帮我们的忙。

B.1 消息分流器

消息分流器（ message cracker ）使窗口过程的编写更加容易。通常，窗口过程是用一个大的 switch 语句实现的。在我的经验中，我见过有的窗口过程的 switch 语句包含 5 百多行代码。我们都知道按这种方式实现窗口过程是一种坏的习惯，但我们都这么做过。而利用消息分流器可将 switch 语句分成小的函数，每个窗口消息对应一个函数。这样使代码更容易管理。

有关窗口过程的另一个问题是每个消息都有 wParam 和 lParam 参数，并且根据消息的不同，这些参数的意思也不同。在某些情况下，如对 WM_COMMAND 消息，wParam 包含两个不同的值。wParam 参数的高字是通知码，而低字是控件的 ID。或者是反过来？我总是忘了次序。如

果使用消息分流器，就不用记住或查阅这些内容。消息分流器之所以这样命名，是因为它们对任何给定的消息进行分流。为了处理 WM_COMMAND 消息，你只需编写这样一个函数：

```
void C1s_OnCommand(HWND hwnd, int id, HWND hwndCtl,
    UINT codeNotify) {

    switch (id) {

        case ID_SOMELISTBOX:
            if (codeNotify != LBN_SELCHANGE)
                break;

            // Do LBN_SELCHANGE processing.
            break;

        case ID_SOMEBUTTON:
            break;

        :
    }
}
```

这是多么容易！分流器查看消息的 wParam 和 lParam 参数，将参数分开，并调用你的函数。

为了使用消息分流器，必须对你的窗口过程的 switch 语句做一些修改。看一看下面的窗口过程：

```
LRESULT WndProc (HWND hwnd, UINT uMsg,
    WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        HANDLE_MSG(hwnd, WM_COMMAND, C1s_OnCommand);
        HANDLE_MSG(hwnd, WM_PAINT, C1s_OnPaint);
        HANDLE_MSG(hwnd, WM_DESTROY, C1s_OnDestroy);
        default:
            return(DefWindowProc(hwnd, uMsg, wParam, lParam));
    }
}
```

HANDLE_MSG 宏在 WindowsX.h 中是这样定义的：

```
#define HANDLE_MSG(hwnd, message, fn) \
    case (message): \
        return HANDLE_##message((hwnd), (wParam), (lParam), (fn));
```

对于 WM_COMMAND 消息，预处理程序把这一行代码扩展成下面的代码：

```
case (WM_COMMAND):
    return HANDLE_WM_COMMAND((hwnd), (wParam), (lParam),
        (C1s_OnCommand));
```

定义在 WindowsX.h 中的各 HANDLE_WM_* 宏是实际的消息分流器。它们分流 wParam 参数和 lParam 参数，执行所有必要的转换，并调用适当的消息函数，如前面例举过的 C1s_OnCommand 函数。HANDLE_WM_COMMAND 宏的定义如下：

```
#define HANDLE_WM_COMMAND(hwnd, wParam, lParam, fn) \
    ( (fn) ((hwnd), (int) (LOWORD(wParam)), (HWND) (lParam), \
        (UINT) HIWORD(wParam)), 0L)
```

当预处理程序扩展这个宏时，其结果是用 wParam 和 lParam 参数的内容分流成各自的部分并经过适当转换，来调用 Cls_OnCommand 函数。

在使用消息分流器来处理一个消息之前，应该打开 WindowsX.h 文件并搜索要处理的消息。例如，如果搜索 WM_COMMAND，将会找到文件中包含下面代码行的部分：

```
/* void Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl,
   UINT codeNotify); */
#define HANDLE_WM_COMMAND(hwnd, wParam, lParam, fn) \
    ((fn)((hwnd), (int)(LOWORD(wParam)), (HWND)(lParam), \
    (UINT)HIWORD(wParam)), 0L)
#define FORWARD_WM_COMMAND(hwnd, id, hwndCtl, codeNotify, fn) \
    (void)(fn)((hwnd), WM_COMMAND, \
    MAKEWPARAM((UINT)(id), (UINT)(codeNotify)), \
    (LPARAM)(HWND)(hwndCtl))
```

第一行是注释行，展示要编写的函数原型。下一行是 HANDLE_WM_* 宏，我们已经讨论过。最后一行是消息转发器（forwarder）。假定在你处理 WM_COMMAND 消息时，你想调用默认的窗口过程，并让它为你做事。这个函数应该是这个样子：

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

    // Do some normal processing.

    // Do default processing.
    FORWARD_WM_COMMAND(hwnd, id, hwndCtl, codeNotify,
        DefWindowProc);
}
```

FORWARD_WM_* 宏将分流开的消息参数重新构造造成等价的 wParam 和 lParam。然后这个宏再调用你提供的函数。在上面的例子中，宏调用 DefWindowProc 函数，但你可以简单地使用 SendMessage 或 PostMessage。实际上，如果你想发送（或登记）一个消息到系统中的任何窗口，可以使用一个 FORWARD_WM_* 宏来帮助合并各个参数。

B.2 子控件宏

子控件宏（Child Control Macro）使发送消息到子控件变得更加容易。这些宏同 FORWARD_WM_* 宏很相似。每个宏的定义以一个控件类型开始（这个控件是要对它发送消息的控件），后面跟一个下横线和消息名。例如，向一个列表框发送一个 LB_GETCOUNT 消息，就使用 WindowsX.h 中的这个宏：

```
#define ListBox_GetCount(hwndCtl) \
    ((int)(DWORD)SendMessage((hwndCtl), LB_GETCOUNT, 0, 0L))
```

关于这个宏我想说两件事。第一，它只用一个参数 hwndCtl，这是列表框的窗口句柄。因为 LB_GETCOUNT 消息忽略了 wParam 和 lParam 参数，你不必再管这些参数。你可以看到，宏只传递了零。第二，当 SendMessage 返回时，结果被转换成 int，所以你不必提供你自己的转换。

关于子控件宏，有一件事我不喜欢，就是这些宏要用控件窗口的句柄。许多时候，你要发送消息到一个控件，而这个控件是一个对话框的子控件。所以最终你总要调用 GetDlgItem，产生这样的代码：

```
int n = ListBox_GetCount(GetDlgItem(hDlg, ID_LISTBOX));
```

比起使用SendDlgItemMessage，这个代码的运行虽然不慢，但你的程序会包含一些额外的代码。这是由于对GetDlgItem的额外调用。如果需要对同一控件发送几个消息，你可能想调用一次GetDlgItem，保存子窗口的句柄，然后调用你需要的所有的宏，见下面的代码：

```
HWND hwndCtl = GetDlgItem(hDlg, ID_LISTBOX);
int n = ListBox_GetCount(hwndCtl);
ListBox_AddString(hwndCtl, "Another string");
:
:
```

如果按这种方式设计你的代码，你的程序会运行得更快，因为这样就不会反复地调用GetDlgItem。如果你的对话框有许多控件并且你要寻找的控件在Z序的结尾，则GetDlgItem可能是个很慢的函数。

B.3 API宏

API宏可以简化某些常用的操作，如建立一种新字体，选择字体到设备环境，保存原来字体的句柄。代码的形式如下：

```
HFONT hfontOrig = (HFONT) SelectObject(hdc, (HGDIOBJ) hfontNew);
```

这个语句要求两个转换以得到没有编译警告错误的编译。在WindowsX.h中有一个宏，正是为了这个用途而设计：

```
#define SelectFont(hdc, hfont) \
((HFONT) SelectObject( (hdc), (HGDIOBJ) (HFONT) (hfont)))
```

如果你使用这个宏，你的程序中的代码行就变成：

```
HFONT hfontOrig = SelectFont(hdc, hfontNew);
```

这行代码更容易读，也不容易出错。

在WindowsX.h中还有其他一些API宏，有助于常用的Windows任务。建议读者了解并使用这些宏。