==========
第二课　窗口与消息
==========

一、窗口类及其注册方法
----

1. 窗口类是包含窗口各种参数信息的结构体(WNDCLASS)。
   每个窗口都属于特定的窗口类，
   且该窗口类必须注册到操作系统中(RegisterClass)。
   每个窗口类必须都有唯一的名称(WNDCLASS::lpszClassName)。

2. 系统窗口类：系统预定义的窗口类，
   应用程序可以直接使用。如：
   按钮(BUTTON)、编辑框(EDIT)等。

范例：WinButton

3. 应用程序全局窗口类：用户自己定义，
   可在当前进程的所有模块(exe及其所加载的各dll)中使用。
   易导致类名冲突，谨慎使用。

4. 应用程序局部窗口类：用户自己定义，
   只能在当前进程的本模块(exe/dll)中使用。
   不同模块即使定义并注册了同名的窗口类，亦无妨，
   自己用自己的。

注意：一个进程可以包含多个模块(exe/dll)，
每个模块都拥有独立的实例句柄(内存映像)。

5. RegisterClass和RegisterClassEx函数：
   将窗口类注册到操作系统中。带Ex是不带Ex的增强版。

```
ATOM RegisterClassEx (CONST WNDCLASSEX* lpwcx);
```

ATOM即unsigned short，成功返回所注册窗口类的唯一标识码(非0)，
失败返回0。

```
typedef struct {
    UINT      cbSize;         // 结构体字节数*
    UINT      style;          // 窗口类风格
    WNDPROC   lpfnWndProc;    // 窗口过程函数指针
    int       cbClsExtra;     // 窗口类附加数据缓冲区字节数
    int       cbWndExtra;     // 窗口附加数据缓冲区字节数
    HINSTANCE hInstance;      // 当前应用程序实例句柄
    HICON     hIcon;          // 大图标句柄(Alt+Tab)
    HICON     hIconSm;        // 小图标句柄(标题栏左上角)*
    HCURSOR   hCursor;        // 光标句柄
    HBRUSH    hbrBackground; // 刷子句柄
    LPCTSTR   lpszMenuName;  // 菜单资源名
    LPCTSTR   lpszClassName; // 窗口类名
}   WNDCLASSEX, *PWNDCLASSEX;
```

(*) 相对于WNDCLASS结构增加的成员。

6. 若WNDCLASSEX::style成员中包含CS_GLOBALCLASS位，

则为应用程序全局窗口类，否则为应用程序局部窗口类。

7. 更多窗口类风格

```
CS_GLOBALCLASS     - 应用程序全局窗口类
CS_BYTEALIGNCLIENT - 窗口客户区水平位置按8像素对齐
CS_BYTEALIGNWINDOW - 窗口水平位置按8像素对齐
CS_HREDRAW         - 窗口水平变化时重绘
CS_VREDRAW         - 窗口垂直变化时重绘
CS_CLASSDC         - 同窗口类窗口共享同一个设备上下文
CS_PARENTDC        - 使用父窗口的设备上下文
CS_OWNDC           - 使用自己的设备上下文
CS_SAVEBITS        - 允许窗口被保存为位图，提高窗口绘图效率，
                     但是耗费内存资源
CS_DBLCLKS         - 允许窗口接收鼠标双击消息
CS_NOCLOSE         - 没有关闭按钮
```

二、注册窗口类与创建窗口

```
-----------------------
```

```
typedef struct {
    ...
    HINSTANCE hInstance;     // 当前应用程序实例句柄
    ...
    LPCTSTR   lpszClassName; // 窗口类名
    ...
}   WNDCLASSEX, *PWNDCLASSEX;

WNDCLASSEX wcex = {0};
wcex.hInstance = hInstance;
wcex.lpszClassName = "MainWnd";
...
RegisterClassEx (&wcex);
```

局部窗口类列表：

```
+--------------+--------------+--------------+
|  局部窗口类1  |  局部窗口类2  |              |
| lpszClassName| lpszClassName|      ...     |
|   hInstance  |   hInstance  |              |
+--------------+--------------+--------------+
```

全局窗口类列表：

```
+--------------+--------------+--------------+
|  全局窗口类1  |  全局窗口类2  |      ...     |
| lpszClassName| lpszClassName|              |
+--------------+--------------+--------------+
```

系统窗口类列表：

```
+--------------+--------------+--------------+
|  系统窗口类1  |  系统窗口类2  |      ...     |
| lpszClassName| lpszClassName|              |
+--------------+--------------+--------------+
```

```
HWND hwndMain = CreateWindow ("MainWnd", ..., hInstance, ...);

HWND CreateWindow (LPCTSTR lpClassName, ..., HANDLE hInstance, ...)
```

```
{
    if (局部窗口类列表中存在lpszClassName和hInstance成员与
        lpClassName和hInstance参数相匹配的元素)
        根据该元素创建窗口并返回其句柄;
    else
    if (全局窗口类列表中存在lpszClassName成员与
        lpClassName参数相匹配的元素)
        根据该元素创建窗口并返回其句柄;
    else
    if (系统窗口类列表中存在lpszClassName成员与
        lpClassName参数相匹配的元素)
        根据该元素创建窗口并返回其句柄;
    else
        return NULL;
}
```

三、窗口类APIs

1. 注册窗口类：RegisterClass/RegisterClassEx。
2. 获取类信息：GetClassInfo。
3. 注销窗口类：UnregisterClass。进程结束时，其所注册的全部窗口类都会被自动注销。

范例：WinReg

四、创建窗口APIs

1. CreateWindow

```
HWND CreateWindow (
    LPCTSTR   lpClassName,  // 窗口类名
    LPCTSTR   lpWindowName, // 窗口标题栏信息
    DWORD     dwStyle,      // 窗口风格
    int       x,            // 窗口左上角水平坐标
    int       y,            // 窗口左上角垂直坐标
    int       nWidth,       // 窗口宽度
    int       nHeight,      // 窗口高度
    HWND      hWndParent,   // 父窗口句柄
    HMENU     hMenu,        // 菜单句柄
    HINSTANCE hInstance,    // 当前应用程序实例句柄
    LPVOID    lpParam       // 附加数据
);
```

成功返回所创建窗口的句柄，失败返回NULL。

2. CreateWindowEx - CreateWindow的加强版

```
HWND CreateWindowEx (
    DWORD     dwExStyle,    // 窗口扩展风格*
    LPCTSTR   lpClassName,  // 窗口类名
    LPCTSTR   lpWindowName, // 窗口标题栏信息
    DWORD     dwStyle,      // 窗口基本风格
    int       x,            // 窗口左上角水平坐标
```

```
    int       y,            // 窗口左上角垂直坐标
    int       nWidth,       // 窗口宽度
    int       nHeight,      // 窗口高度
    HWND      hWndParent,   // 父窗口句柄
    HMENU     hMenu,        // 菜单句柄
    HINSTANCE hInstance,    // 当前应用程序实例句柄
    LPVOID    lpParam       // 附加数据
);
```

成功返回所创建窗口的句柄，失败返回NULL。

3. 窗口扩展风格

```
WS_EX_CLIENTEDGE - 凹入效果的窗口边框
WS_EX_MDICHILD   - MDI子窗口
```

4. 窗口基本风格

```
WS_BORDER           - 边框
WS_CAPTION          - 标题
WS_CHILD            - 子窗口
WS_CHILDWINDOW      - 同WS_CHILD
WS_CLIPCHILDREN     - 绘制父窗口时，不绘制被子窗口覆盖的部分。
                      应用于父窗口
WS_CLIPSIBLINGS     - 绘制子窗口的，不绘制被兄弟窗口覆盖的部分。
                      应用于子窗口
WS_DISABLE          - 不可用
WS_DLGFRAME         - 对话框边框
WS_GROUP            - 控件组的首控件
WS_HSCROLL          - 水平滚动条
WS_ICONIC           - 最小化
WS_MAXIMIZE         - 最大化
WS_MAXIMIZEBOX      - 最大化按钮
WS_OVERLAPPED       - 交叠——带标题栏和边框
WS_OVERLAPPEDWINDOW - 交叠、标题、系统菜单、带尺寸框的粗边框、
                      最小化按钮、最大化按钮
WS_POPUP            - 弹出式
WS_POPUPWINDOW      - 边框、弹出式、系统菜单
WS_SIZEBOX          - 带尺寸框的粗边框
WS_SYSMENU          - 系统菜单
WS_TABSTOP          - 可用Tab键切换的控件
WS_THICKFRAME       - 同WS_SIZEBOX
WS_TILED            - 同WS_OVERLAPPED
WS_TILEDWINDOW      - 同WS_OVERLAPPEDWINDOW
WS_VISIBLE          - 初始可见
WS_VSCROLL          - 垂直滚动条
```

范例：WinCreate

5. 窗口左上角的水平坐标和垂直坐标以及窗口的宽度和高度可取缺省值CW_USEDEFAULT

6. 对WM_DESTROY消息的处理

```
case WM_DESTROY:
    PostQuitMessage (0);
    return 0;
```

当窗口被关闭的瞬间，窗口过程函数会收到WM_DESTROY消息，
此时通过PostQuitMessage函数寄出WM_QUIT消息，
可令GetMessage函数返回FALSE，从而退出消息循环，应用程序进程结束。

```
while (GetMessage (&msg, NULL, 0, 0))
{
    TranslateMessage (&msg);
    DispatchMessage (&msg);
}

void PostQuitMessage (
    int nExitCode // 应用程序的退出码，
                  // 该值将作为WM_QUIT消息的wParam参数
);
```

将WM_QUIT消息放入当前线程的消息队列中。

7. 子窗口的创建

1) 创建子窗口时必须设置父窗口的句柄。

2) 创建子窗口时必须增加WS_CHILD和WS_VISIBLE风格。

范例：WinChild

8. 窗口类附加数据和窗口附加数据

```
typedef struct {
    ...
    int cbClsExtra; // 窗口类附加数据缓冲区字节数
    int cbWndExtra; // 窗口附加数据缓冲区字节数
    ...
}   WNDCLASSEX, *PWNDCLASSEX;
```

为窗口类和窗口提供存放自己数据的内存空间。

1) 窗口类附加数据区

A. 建附加数据缓冲区

int WNDCLASSEX::cbClsExtra - 一般为4字节的倍数

B. 写附加数据缓冲区

```
DWORD SetClassLong (
    HWND hWnd,     // 窗口句柄
    int  nIndex,   // 附加数据索引号，[0, cbClsExtra/4-1]
    LONG dwNewLong // 写入的数据
);
```

返回该位置的原数据。

C. 读附加数据缓冲区

```
DWORD GetClassLong (
    HWND hWnd,   // 窗口句柄
    int  nIndex, // 附加数据索引号，[0, cbClsExtra/4-1]
);
```

返回相应位置的数据。

2) 窗口附加数据区

A. 建附加数据缓冲区

int WNDCLASSEX::cbWndExtra - 一般为4字节的倍数

B. 写附加数据缓冲区

```
DWORD SetWindowLong (
    HWND hWnd,     // 窗口句柄
    int  nIndex,   // 附加数据索引号，[0, cbWndExtra/4-1]
    LONG dwNewLong // 写入的数据
);
```

返回该位置的原数据。

C. 读附加数据缓冲区

```
DWORD GetWindowLong (
    HWND hWnd,   // 窗口句柄
    int  nIndex, // 附加数据索引号，[0, cbWndExtra/4-1]
);
```

返回相应位置的数据。

3) 窗口类附加数据缓冲区和窗口附加数据缓冲区的区别：
   窗口类附加数据缓冲区为该窗口类的所有窗口所共享。
   窗口附加数据缓冲区只属于该窗口所有。

范例：WinExtra

五、显示窗口APIs

```
BOOL ShowWindow (HWND hWnd, int nCmdShow)
{
    根据hWnd参数获取位置、大小、颜色等窗口信息;
    根据窗口信息和nCmdShow参数绘制窗口区域;
}

BOOL UpdateWindow (HWND hWnd)
{
    if (hWnd参数所标识的窗口存在无效区域)
        以WM_PAINT消息直接调用相应的窗口过程函数(注意：不经过消息队列);
```

```
}
```

六、消息基本概念

1. 程序执行机制

1) 过程驱动：程序按照预定顺序执行。

2) 事件驱动：程序根据用户触发的事件执行相应的动作。

3) Win32窗口程序采用事件驱动方式执行——消息机制。

2. 消息

1) 当系统通知窗口工作时，采用消息的方式派发给窗口。
   窗口过程函数接收并处理消息。

2) 消息的组成：

```
窗口句柄        - 消息派发给哪个窗口
消息ID          - 标识具体是什么消息
消息的两个参数  - 消息附带的信息，因不同消息而异
消息产生的时间  - 消息发生的系统时间
消息产生的位置  - 消息发生瞬间的鼠标位置
```

```
typedef struct tagMSG {
    HWND   hwnd;    // 窗口句柄
    UINT   message; // 消息ID
    WPARAM wParam;  // 消息参数
    LPARAM lParam;  // 消息参数
    DWORD  time;    // 消息产生的时间
    POINT  pt;      // 消息产生的位置
}   MSG;
```

注意：MSG结构的前4个成员与窗口过程函数的4个参数完全一致。

3. 消息队列

```
                +-----------+---------+----------+-----+
GetMessage() <- | WM_CREATE | WM_SIZE | WM_PAINT | ... | <- PostMessage()
                +-----------+---------+----------+-----+
```

先进先出。

4. 消息循环

```
MSG msg = {0};
while (GetMessage (&msg, NULL, 0, 0))
{
    TranslateMessage (&msg);
    DispatchMessage (&msg);
```

```
}
```

1) GetMessage函数负责从消息队列中获取消息，并将其填入msg结构体。

```
BOOL GetMessage (
    LPMSG lpMsg,          // 消息结构
    HWND  hWnd,           // 窗口句柄
                          // 非NULL，获取当前线程特定窗口的消息
                          // 为NULL，获取当前线程任意窗口的消息
    UINT  wMsgFilterMin, // 起始消息\
                                  > 若非0，只取闭区间
    UINT  wMsgFilterMax  // 终止消息/  [wMsgFilterMin, wMsgFilterMax]内的消息
);
```

收到WM_QUIT消息返回FALSE，收到其它消息返回TRUE。

```
case WM_DESTROY:
    PostQuitMessage (0);
    break;
```

PostQuitMessage函数即向消息队列放入WM_QUIT消息。

2) TranslateMessage函数负责对部分消息(键盘可见字符按键消息)进行翻译。

```
BOOL TranslateMessage (
    const MSG* lpMsg // 消息结构
);
```

若消息被翻译则返回TRUE，否则返回FALSE。

根据CapsLock键状态判断大小写。

3) DispatchMessage函数首先根据msg中的hwnd成员所标识窗口的窗口类，
   确定相应的窗口过程函数，
   随后以msg中的前4个成员为参数调用该窗口过程函数，
   并返回该窗口过程函数的返回值。

```
LONG DispatchMessage (const MSG* lpmsg)
{
    根据lpmsg->hwnd获取相应的窗口类;
    从窗口类的lpfnWndProc成员确定窗口过程函数的地址;
    return 窗口过程函数 (lpmsg->hwnd, lpmsg->message,
        lpmsg->wParam, lpmsg->lParam);
}
```

4) 一旦GetMessage函数从消息队列中取到WM_QUIT消息，
   即返回FALSE，消息循环结束。

5. 消息处理

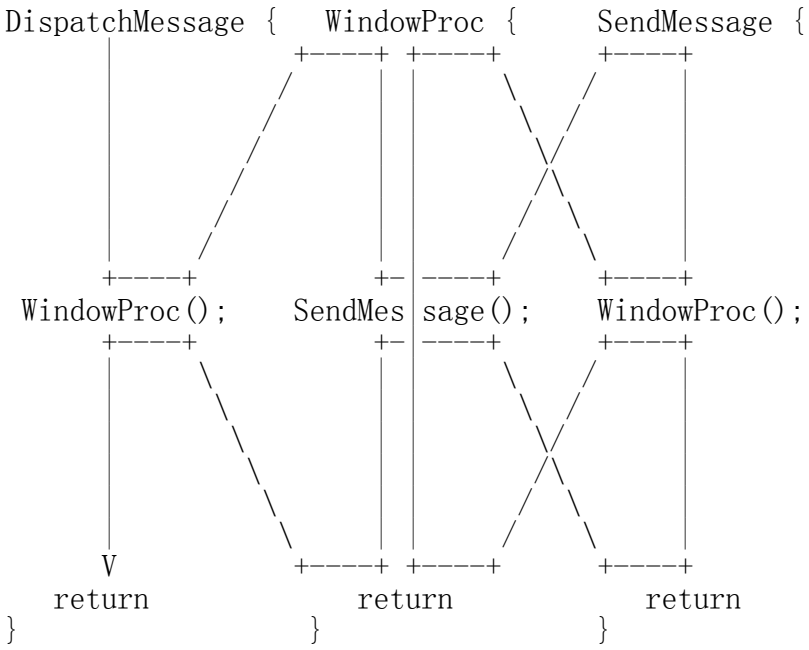
1) 每个窗口都必须具有窗口过程函数。

```
LRESULT CALLBACK WindowProc (
    HWND   hWnd,    // 窗口句柄
    UINT   uMsg,    // 消息标识
```

```
    WPARAM wParam, // 消息参数
    LPARAM lParam  // 消息参数
);
```

窗口过程函数负责对具体消息做具体处理。

2) 窗口过程函数除了被DispatchMessage函数调用外，
   也可能被其它API函数调用，需要考虑重入问题。



3) 窗口过程函数不需要处理的消息，可交由缺省窗口过程函数处理。

```
LRESULT CALLBACK DefWindowProc (
    HWND   hWnd,   // 窗口句柄
    UINT   uMsg,   // 消息标识
    WPARAM wParam, // 消息参数
    LPARAM lParam  // 消息参数
);
```

4) 常见消息

A. WM_DESTROY - 窗口被销毁时发送此消息，无消息参数。
                常用于窗口销毁过程中的善后处理，
                如释放内存资源、寄出WM_QUIT消息等

```
点击窗口的关闭按钮
  -> 发送WM_SYSCOMMAND消息，wParam为SC_CLOSE
    -> DefWindowProc()
      -> CloseWindow()
        -> 发送WM_CLOSE消息
          -> DefWindowProc()
            -> DestroyWindow()
              -> 销毁窗口，发送WM_DESTROY消息
```

B. WM_SYSCOMMAND - 系统命令消息。
                   当点击窗口的最大化、最小化、关闭

等命令按钮或菜单时，收到此消息。
常用于在窗口关闭时提示用户处理
wParam - 具体命令，如SC_CLOSE(关闭)等
lParam - 鼠标位置。LOWORD宏取低字，水平位置，HIWORD宏取高字，垂直位置

范例：WinMsg

C. WM_CREATE - 在CreateWindow/CreateWindowEx函数执行过程中，
收到这个消息，针对此消息的窗口过程函数返回以后，
CreateWindow/CreateWindowEx才会返回。
常用于初始化窗口状态、分配资源，创建子窗口，等等
wParam - 不使用
lParam - CREATESTRUCT结构体指针，保存了CreateWindowEx函数的12个参数

```
HWND CreateWindowEx (
    DWORD     dwExStyle,      // 窗口扩展风格
    LPCTSTR   lpClassName,    // 窗口类名
    LPCTSTR   lpWindowName,   // 窗口标题栏信息
    DWORD     dwStyle,        // 窗口基本风格
    int       x,              // 窗口左上角水平坐标
    int       y,              // 窗口左上角垂直坐标
    int       nWidth,         // 窗口宽度
    int       nHeight,        // 窗口高度
    HWND      hWndParent,     // 父窗口句柄
    HMENU     hMenu,          // 菜单句柄
    HINSTANCE hInstance,      // 当前应用程序实例句柄
    LPVOID    lpParam         // 附加数据
);

typedef struct tagCREATESTRUCT {
    LPVOID    lpCreateParams; // 附加数据
    HINSTANCE hInstance;      // 当前应用程序实例句柄
    HMENU     hMenu;          // 菜单句柄
    HWND      hwndParent;     // 父窗口句柄
    int       cy;             // 窗口高度
    int       cx;             // 窗口宽度
    int       y;              // 窗口左上角垂直坐标
    int       x;              // 窗口左上角水平坐标
    LONG      style;          // 窗口基本风格
    LPCTSTR   lpszName;       // 窗口标题栏信息
    LPCTSTR   lpszClass;      // 窗口类名
    DWORD     dwExStyle;      // 窗口扩展风格
}   CREATESTRUCT;
```

若窗口过程函数在此消息的处理中返回0，则窗口被成功创建，
若返回-1，则窗口创建失败(通过DestroyWindow销毁)，
CreateWindowEx/CreateWindow函数返回NULL。

范例：WinCrt

D. WM_SIZE - 在窗口大小发生变化后，会收到这个消息。
常用于窗口大小变化后调整窗口内各个部分的布局
wParam - 窗口大小变化的原因
lParam - 变化后窗口客户区的大小。LOWORD宽度，HIWORD高度

范例：WinSize

E. WM_QUIT - 用于结束消息循环处理
    wParam - PostQuitMessage函数的参数
    lParam - 不使用

当GetMessage函数收到这个消息后，会返回FALSE，结束消息循环。

F. 其它消息

绘图消息(WM_PAINT)、键盘消息(WM_KEYDOWN等)、
鼠标消息(WM_MOUSEMOVE等)、定时器消息(WM_TIMER)等。

七、获取消息和查看消息
---

1. 获取消息：GetMessage

从系统中获取消息，将消息从系统中移除。
阻塞函数，当系统中没有消息时，该函数会一直等待，
直到成功取得一条消息才返回。

2. 查看消息：PeekMessage

以查看的方式从系统中获取消息，可以不将消息从系统中移除。
非阻塞函数，当系统中没有消息时，该函数会立即返回FALSE。

```
BOOL PeekMessage (
    LPMSG lpMsg,          // 消息结构
    HWND  hWnd,           // 窗口句柄
    UINT  wMsgFilterMin, // 起始消息
    UINT  wMsgFilterMax, // 终止消息
    UINT  wRemoveMsg     // PM_NOREMOVE - 不移除消息，PM_REMOVE - 移除消息
);
```

有消息返回TRUE，否则返回FALSE。

3. 更好的消息循环

```
MSG msg = {0};
for (;;)
    if (PeekMessage (&msg, NULL, 0, 0, PM_NOREMOVE))
    {
        if (! GetMessage (&msg, NULL, 0, 0))
            break;

        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    else
    {
        空闲处理;
    }
```

注意：空闲处理不适宜做过于耗时的操作。

范例：WinPeek

八、发送消息和寄出消息
---

1. 发送消息：SendMessage

不将消息放入消息队列，而是直接调用窗口过程函数，
返回窗口过程函数的返回值。

```
LRESULT SendMessage (
    HWND   hWnd,   // 窗口句柄
    UINT   uMsg,   // 消息ID
    WPARAM wParam, // 消息参数
    LPARAM lParam  // 消息参数
);
```

返回窗口过程函数的返回值。

2. 寄出消息：PostMessage

将消息放入消息队列即返回。

```
BOOL PostMessage (
    HWND   hWnd,   // 窗口句柄
    UINT   uMsg,   // 消息ID
    WPARAM wParam, // 消息参数
    LPARAM lParam  // 消息参数
);
```

成功返回TRUE，失败返回FALSE。

范例：WinMessage

九、消息分类
---

1. 系统消息：0x0000 - 0x03FF

由系统定义的消息，可以在程序中直接使用。

2. 用户消息：0x0400(WM_USER) - 0x7FFF

由用户定义的消息，满足用户自己的需求。
用户自己发出消息，并响应处理。

```
#define WM_EAT WM_USER+1
#define WM_SLEEP WM_USER+2

SendMessage (WM_EAT, ...);
```

```
PostMessage (WM_SLEEP, ...);
```

范例：WinUser

3. 应用程序消息：0x8000(WM_APP) - 0xBFFF

应用程序之间通信时使用的消息。

```
#define WM_WORK WM_APP+1
#define WM_REST WM_APP+2
```

4. 系统注册消息：0xC000 - 0xFFFF

通过RegisterWindowMessage函数在系统中注册字符串消息，
多个应用程序使用该消息通信。

十、消息队列
---

1. 消息队列的概念

消息队列是用于存放消息的队列。消息在队列中先进先出。
所有窗口程序都有消息队列。程序可以从消息队列中获取消息。

2. 消息队列的类型

1) 系统消息队列：由系统维护的消息队列，
   存放系统产生的消息，如鼠标、键盘等。

2) 程序消息队列：属于每一个应用程序(线程)的消息队列，
   由应用程序(线程)维护。

3. 消息队列的关系

1) 当鼠标、键盘等产生消息时，
   首先将该消息存放到系统消息队列中。

2) 系统根据系统消息队列中每个消息的具体信息，
   找到对应窗口所属的程序(线程)。

3) 将该消息转发到程序消息队列中。

十一、队列消息与非队列消息
---

1. 队列消息

消息发送后，首先进入消息队列，
然后通过消息循环，从消息队列中获取。

```
 GetMessage() <- 程序消息队列
PeekMessage() <- 程序消息队列
PostMessage() -> 系统消息队列
```

诸如WM_PAINT、键盘、鼠标、定时器等，对顺序要求高，
对时间要求低的消息，常被处理为队列消息。

2. 非队列消息

消息发送后，不经由消息队列，直接调用窗口过程函数。

```
SendMessage() -> 窗口过程函数
```

诸如WM_CREATE、WM_SIZE等，对顺序要求低，
对时间要求高的消息，常被处理为非队列消息。

十二、消息APIs
---

1. GetMessage/PeekMessage

从程序消息队列中获取/查看消息。

```
BOOL GetMessage (LPMSG lpMsg, HWND hWnd,
    UINT wMsgFilterMin, UINT wMsgFilterMax)
{
check_prog_mq:

    if (程序消息队列中有满足hWnd、wMsgFilterMin和wMsgFilterMax限制的消息)
    {
        将该消息填入lpMsg所指向的MSG结构;
        从程序消息队列中删除该消息;
        return lpMsg -> message != WM_QUIT ? TRUE : FALSE;
    }

check_sys_mq:

    if (系统消息队列中有属于本程序的消息)
    {
        要求系统将该消息转发到程序消息队列;
        goto check_prog_mq;
    }

    if (当前程序中存在需要重绘(包含无效区域)的窗口)
    {
        产生一个WM_PAINT消息，追加到系统消息队列的尾部;
        goto check_sys_mq;
    }

    if (当前程序中存在已经到期的定时器)
        if (该定时器存在自己的处理函数)
            调用定时器处理函数;
        else
        {
```

```
            产生一个WM_TIMER消息，追加到系统消息队列的尾部;
            goto check_sys_mq;
        }

    整理资源;

    goto check_prog_mq;
}

BOOL PeekMessage (LPMSG lpMsg, HWND hWnd,
    UINT wMsgFilterMin, UINT wMsgFilterMax, UINT wRemoveMsg)
{
check_prog_mq:

    if (程序消息队列中有满足hWnd、wMsgFilterMin和wMsgFilterMax限制的消息)
    {
        将该消息填入lpMsg所指向的MSG结构;
        if (wRemoveMsg == PM_REMOVE)
            从程序消息队列中删除该消息;
        return TRUE;
    }

check_sys_mq:

    if (系统消息队列中有属于本程序的消息)
    {
        要求系统将该消息转发到程序消息队列;
        goto check_prog_mq;
    }

    if (当前程序中存在需要重绘(包含无效区域)的窗口)
    {
        产生一个WM_PAINT消息，追加到系统消息队列的尾部;
        goto check_sys_mq;
    }

    if (当前程序中存在已经到期的定时器)
        if (该定时器存在自己的处理函数)
            调用定时器处理函数;
        else
        {
            产生一个WM_TIMER消息，追加到系统消息队列的尾部;
            goto check_sys_mq;
        }

    整理资源;

    return FALSE;
}
```

注意：WM_TIMER消息有可能因为消息过多而延期生成。

2. TranslateMessage

检查取到的消息，若为按键消息则产生一个字符消息(WM_CHAR)，

重新放入程序消息队列，等待后续获取/查看。

3. DispatchMessage

根据消息的具体信息，找到相应的窗口过程函数，
调用之，并返回其返回值。

4. SendMessage

以消息为参数调用指定窗口的窗口过程函数，
并返回其返回值。SendMessage函数返回表示消息处理完毕。

5. PostMessage

将消息放到消息队列中，立刻返回。用于队列消息。
PostMessage函数返回，消息未必处理完。

十三、窗口绘制消息：WM_PAINT
---

1. WM_PAINT消息

WM_PAINT - 当窗口需要绘制的时候，相应的窗口过程函数会收到此消息
  wParam - 不使用
  lParam - 不使用

首次显示
改变大小
从最小化恢复
最大化
被遮挡部分恢复暴露

2. 窗口的无效区域与无效矩形

1) 窗口中需要重新绘制的区域被称为窗口的无效区域。

2) 窗口所有无效区域的外接矩形称为窗口的无效矩形。

3) 人为设定窗口的无效矩形

```
BOOL InvalidateRect(
    HWND  hWnd,          // 窗口句柄
    CONST RECT* lpRect, // 无效矩形，
                         // NULL表示将整个客户区都设为无效
    BOOL  bErase         // 指示是否发送WM_ERASEBKGND消息重绘窗口背景，
                         // 与BeginPaint函数所输出绘图结构的fErase成员一致
);
```

矩形结构：

```
typedef struct _RECT {
```

```
    LONG left;   // 左上角水平坐标
    LONG top;    // 左上角垂直坐标
    LONG right;  // 右下角水平坐标
    LONG bottom; // 右下角垂直坐标
}   RECT, *PRECT;
```

在程序中，如果需要重绘窗口，
可调用该函数将需要重绘的部分设为无效，
进而引发一次WM_PAINT消息处理。

3. WM_PAINT消息的处理

1) 开始绘图：获取设备上下文句柄，为后续绘图操作做准备。

```
HDC BeginPaint (
    HWND          hwnd,   // 窗口句柄
    LPPAINTSTRUCT lpPaint // 绘图结构
);
```

成功返回设备上下文句柄，否则返回NULL。

```
typedef struct tagPAINTSTRUCT {
    HDC  hdc;
    BOOL fErase;
    RECT rcPaint; // 无效矩形
    BOOL fRestore;
    BOOL fIncUpdate;
    BYTE rgbReserved[32];
}   PAINTSTRUCT, *PPAINTSTRUCT;
```

如：

```
PAINTSTRUCT ps;
HDC hDC = BeginPaint (hWnd, &ps);
...
```

2) 执行绘图

创建GDI对象(画笔、画刷、字体等)
    设置设备上下文
        调用绘图函数
    恢复设备上下文
销毁GDI对象

如：

```
...
HPEN hpenNew = CreatePen (PS_SOLID, 1, RGB (255, 0, 0));
HGDIOBJ hpenOld = SelectObject (hDC, hpenNew);
...
Ellipse (hDC, ...);
...
SelectObject (hDC, hpenOld);
DeleteObject (hpenNew);
...
```

3）结束绘图：释放BeginPaint函数所分配的资源，结束绘图过程。

```
BOOL EndPaint(
    HWND                hWnd,    // 窗口句柄
    CONST PAINTSTRUCT* lpPaint // 绘图结构
);
```

任何时候都返回TRUE。

如：

```
...
EndPaint (hWnd, &ps);
```

范例：WinPaint