

MFC

（基础篇)

目录

[1. MFC入门 4](#_Toc457628858)

[1.1 为什么学习MFC 4](#_Toc457628859)

[1.2 Windows消息机制 4](#_Toc457628860)

[1.2.1 基本概念解释 5](#_Toc457628861)

[1.2.2 Windows 编程模型 9](#_Toc457628862)

[1.3 MFC入门 28](#_Toc457628863)

[1.3.1 MFC是什么? 28](#_Toc457628864)

[1.3.2 编写第一个MFC应用程序 28](#_Toc457628865)

[1.3.3 消息映射 33](#_Toc457628866)

[1.3.4 帮助文档的使用 35](#_Toc457628867)

[1.3.5 Widnows字符集 37](#_Toc457628868)

[1.4 用向导生成一个MFC应用程序 39](#_Toc457628869)

[1.4.1 向导流程 39](#_Toc457628870)

[1.4.2 类视图 41](#_Toc457628871)

[1.4.3 文档/视图结构体系 42](#_Toc457628872)

[1.4.4 消息处理的添加 43](#_Toc457628873)

[1.4.5 MFC框架中一些重要的函数 46](#_Toc457628874)

[1.5 拓展知识点 49](#_Toc457628875)

[2. 基于对话框编程 49](#_Toc457628876)

[2.1 创建基于对话框的 MFC 应用程序框架 50](#_Toc457628877)

[2.2 对话框应用程序框架介绍 51](#_Toc457628878)

[2.2.1 资源视图 51](#_Toc457628879)

[2.2.2 类视图 53](#_Toc457628880)

[2.2.3 设计界面和工具箱 54](#_Toc457628881)

[2.3 模态对话框 54](#_Toc457628882)

[2.4 非模态对话框 58](#_Toc457628883)

[3. 常用控件 60](#_Toc457628884)

[3.1 静态文本框CStatic 60](#_Toc457628885)

[3.2 普通按钮 CButton 63](#_Toc457628886)

[3.3 编辑框CEdit 66](#_Toc457628887)

[3.3.1 关联控件变量 67](#_Toc457628888)

[3.3.2 关联基本类型变量 68](#_Toc457628889)

[3.4 组合框(下拉框) CComboBox 69](#_Toc457628890)

[3.5 列表控件 CListCtrl 71](#_Toc457628891)

[3.6 树控件 CTreeCtrl 74](#_Toc457628892)

[3.7 标签控件 CTabCtrl 80](#_Toc457628893)

[4. 综合案例：销售信息管理系统 83](#_Toc457628894)

1. MFC入门

1.1 为什么学习MFC

如果你是在Windows平台上做GUI开发，MFC(微软基础类库)是一个很好的选择，毕竟Windows累积用户群庞大，市场接受程度高。

但是，学习MFC不仅仅要学习用MFC，还要学习MFC的框架设计思想。

很多公司在一些做了很久的项目上，往往都是有自己的类库、自己的框架，我们只需要在其基础上不断的完善和扩展。如果你不了解类库，你是根本无从下手。这也是我们要学习类库、框架设计的原因。

如果仅仅会用MFC的话，可能在找工作的时候，一旦工作内容离开了MFC，就什么也不会了。学习任何东西都是一样的，学的是方法，而不仅仅是某个技术本身，因为技术肯定不停地更新的，你今天学的现在能用上，但是谁也不能保住以后会怎么样，但是万变不离其中，懂得了方法，学什么都一样。

1.2 Windows消息机制

要想熟练掌握 Windows 应用程序的开发， 首先需要理解 Windows 平台下程序运行的内部机制。如果想要更好的学习掌握 MFC，必须要先了解Windows 程序的内部运行机制，为我们扫清学习路途中的第一个障碍，为进一步学习 MFC 程序打下基础。

1.2.1 基本概念解释

我们在编写标准C程序的时候,经常会调用各种库函数来辅助完成某些功能：初学者使用得最多的C库函数就是printf了，这些库函数是由你所使用的编译器厂商提供的。在Windows平台下，也有类似的函数可供调用：不同的是，这些函数是由Windows操作系统本身提供的。

1) SDK和API

**SDK**： 软件开发工具包（Software Development Kit），一般都是一些被软件工程师用于为特定的软件包、软件框架、硬件平台、操作系统等建立应用软件的开发工具的集合。

**API函数**： Windows操作系统提供给应用程序编程的接口（Application Programming Interface）。

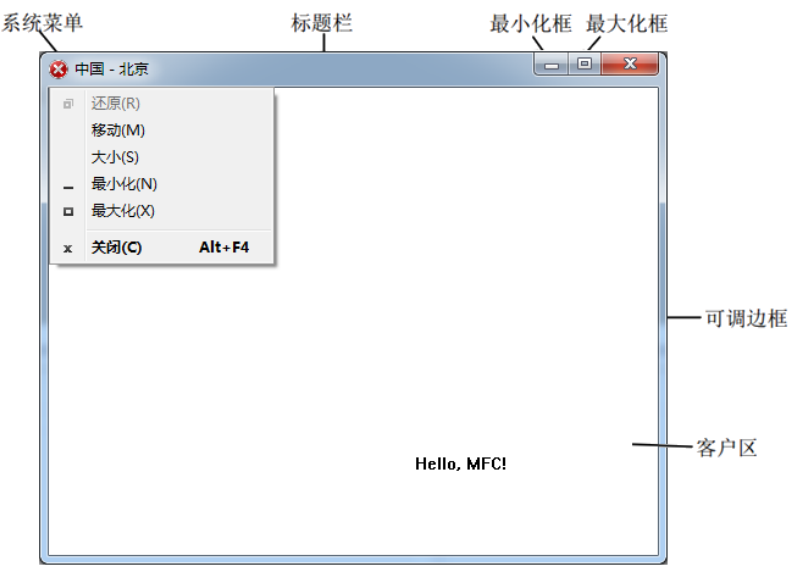
Windows应用程序API函数是通过C语言实现的，所有主要的 Windows 函数都在 Windows.h 头文件中进行了声明。Windows 操作系统提供了 1000 多种 API函数。

2) 窗口和句柄

窗口是 Windows 应用程序中一个非常重要的元素，一个 Windows 应用程序至少要有一个窗口，称为主窗口。

窗口是屏幕上的一块矩形区域，是 Windows 应用程序与用户进行交互的接口。利用窗口可以接收用户的输入、以及显示输出。

一个应用程序窗口通常都包含标题栏、菜单栏、系统菜单、最小化框、最大化框、 可调边框，有的还有滚动条。如下图：



窗口可以分为客户区和非客户区， 如上图。 客户区是窗口的一部分， 应用程序通常在客户区中显示文字或者绘制图形。

标题栏、 菜单栏、 系统菜单、 最小化框和最大化框、 可调边框统称为窗口的非客户区， 它们由 Windows 系统来管理， 而应用程序则主要管理客户区的外观及操作。

窗口可以有一个父窗口， 有父窗口的窗口称为子窗口。除了上图所示类型的窗口外， 对话框和消息框也是一种窗口。 在对话框上通常还包含许多子窗口， 这些子窗口的形式有按钮、 单选按钮、 复选框、 组框、 文本编辑框等。

在 Windows 应用程序中， 窗口是通过窗口句柄（ HWND） 来标识的。 我们要对某个窗口进行操作， 首先就要得到这个窗口的句柄。

句柄（ HANDLE） 是 Windows 程序中一个重要的概念， 使用也非常频繁。 在 Windows 程序中， 有各种各样的资源（ 窗口、 图标、光标,画刷等）， 系统在创建这些资源时会为它们分配内存， 并返回标识这些资源的标识号， 即句柄。 在后面的内容中我们还会看到图标句柄（ HICON）、 光标句柄（ HCURSOR） 和画刷句柄（ HBRUSH）。

3) 消息与消息队列

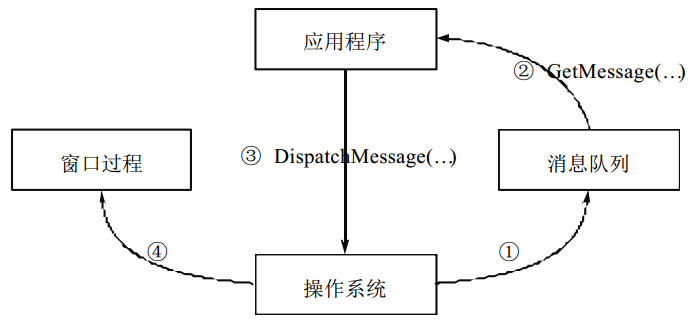
Windows 程序设计是一种完全不同于传统的 DOS 方式的程序设计方法。它是一种事件驱动方式的程序设计模式，主要是基于消息的。

每一个 Windows 应用程序开始执行后， 系统都会为该程序创建一个消息队列， 这个消息队列用来存放该程序创建的窗口的消息。

例如，当用户在窗口中画图的时候，按下鼠标左键，此时，操作系统会感知到这一事件，于是将这个事件包装成一个消息，投递到应用程序的消息队列中，等待应用程序的处理。

然后应用程序通过一个消息循环不断地从消息队列中取出消息，并进行响应。

在这个处理过程中，操作系统也会给应用程序“ 发送消息”。所谓“ 发送消息”，实际上是操作系统调用程序中一个专门负责处理消息的函数，这个函数称为窗口过程。



4) WinMain函数

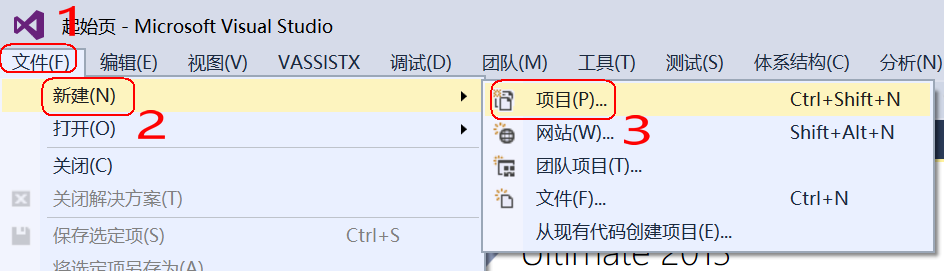
当Windows操作系统启动一个程序时，它调用的就是该程序的WinMain函数（ 实际是由插入到可执行文件中的启动代码调用的）。 WinMain是Windows程序的入口点函数，与DOS程序的入口点函数main的作用相同，当WinMain 函数结束或返回时，Windows应用程序结束。

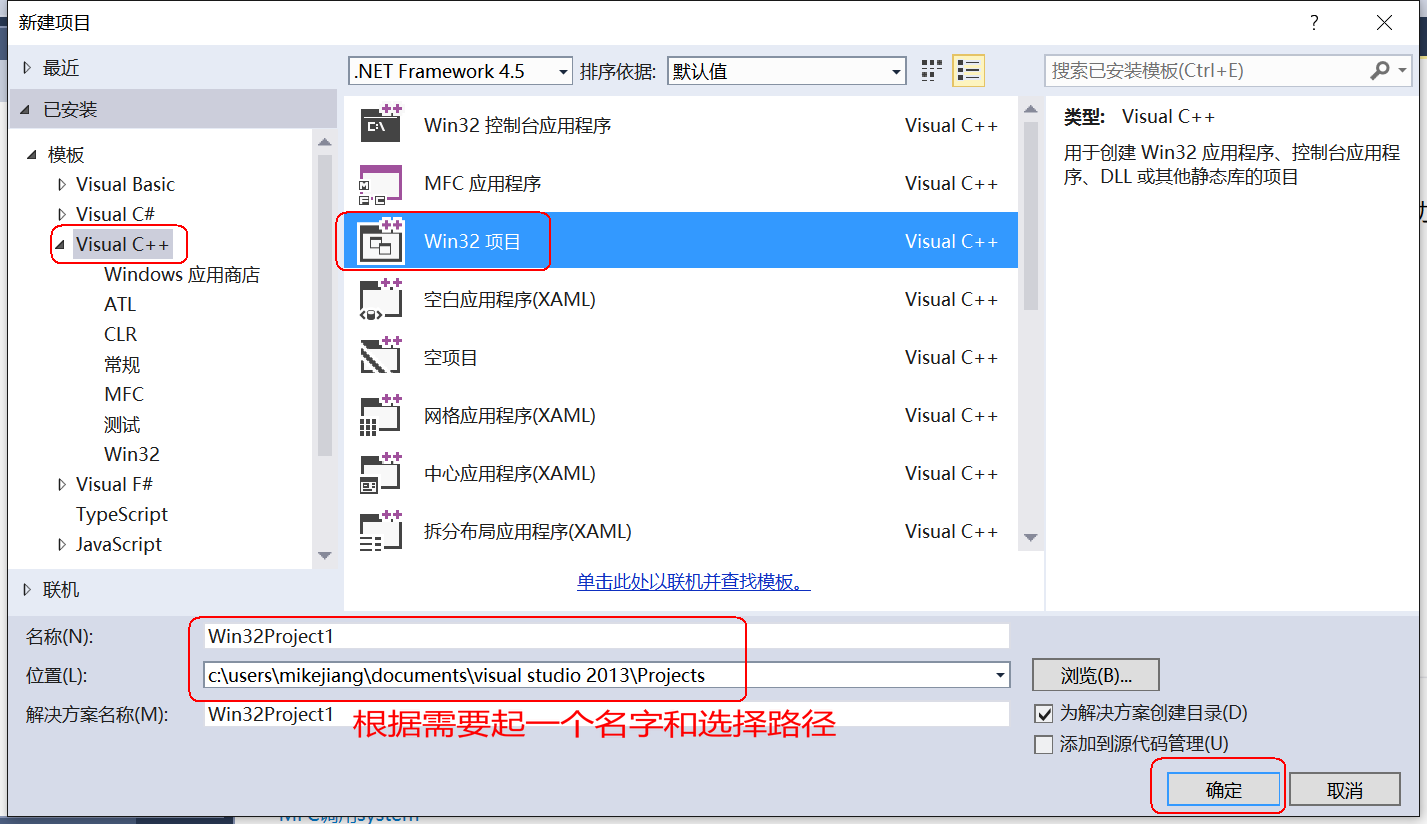
1.2.2 Windows 编程模型

一个完整的Win32程序(#include <windows.h>)，该程序实现的功能是创建一个窗口，并在该窗口中响应键盘及鼠标消息，程序的实现步骤为：

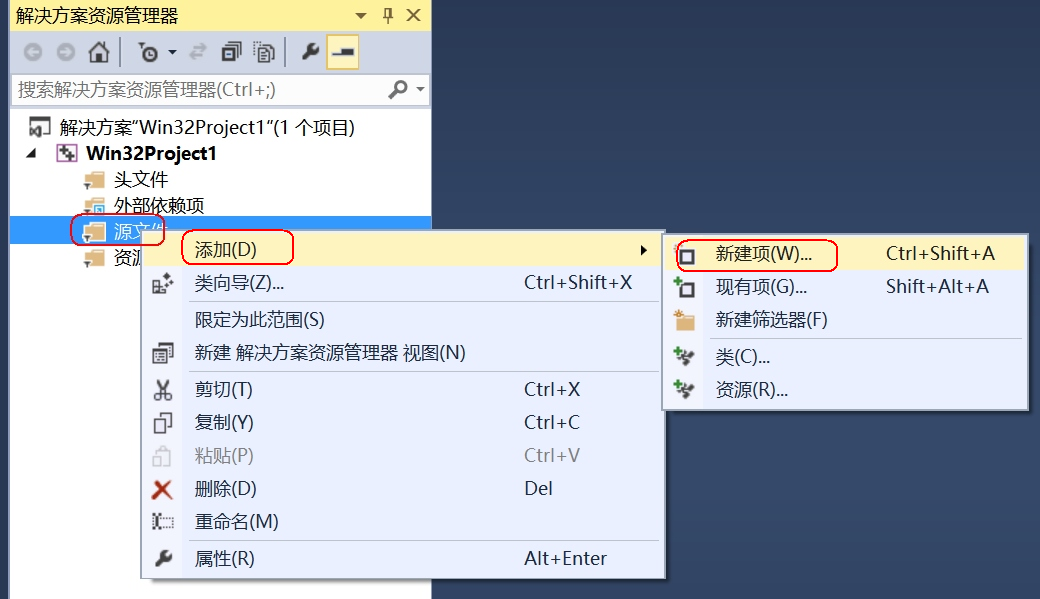
1. WinMain函数的定义
2. 创建一个窗口
3. 进行消息循环
4. 编写窗口过程函数

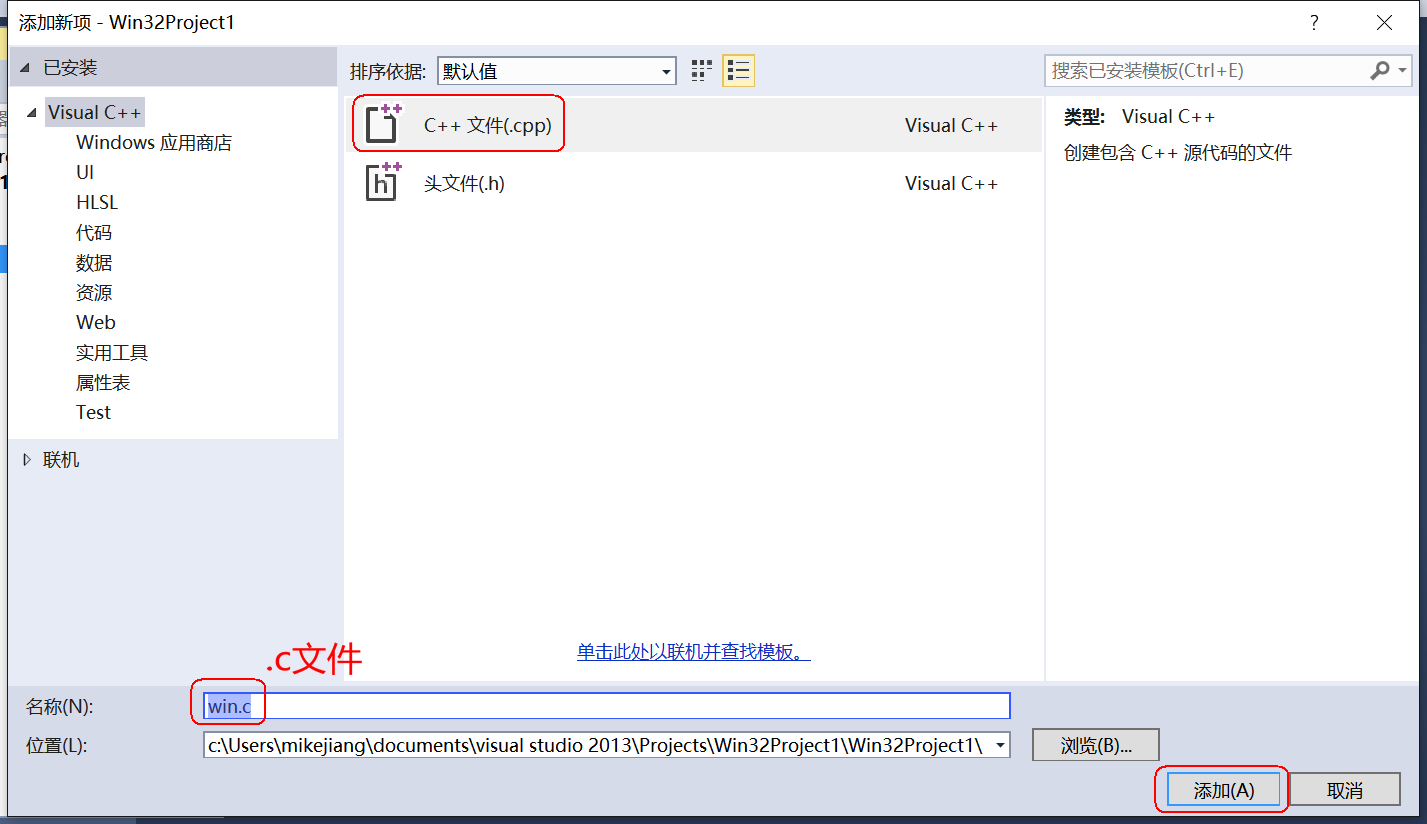
1) 项目的创建











2) WinMain函数的定义

int *WINAPI* WinMain(

*HINSTANCE* hInstance, //应用程序实例

*HINSTANCE* hPrevInstance, //上一个应用程序实例

*LPSTR* lpCmdLine, //命令行参数

int nShowCmd); //窗口显示的样式

* **WINAPI**：是一个宏，它代表的是\_\_stdcall（注意是两个下划线），表示的是参数传递的顺序：从右往左入栈，同时在函数返回前自动清空堆栈。
* **hInstance**：表示该程序当前运行的实例的句柄，这是一个数值。当程序在Windows下运行时，它唯一标识运行中的实例（注意，只有运行中的程序实例， 才有实例句柄）。一个应用程序可以运行多个实例，每运行一个实例，系统都会给该实例分配一个句柄值，并通过hInstance参数传递给 WinMain 函数。
* **hPrevInstance**：表示当前实例的前一个实例的句柄。在Win32环境下，这个参数总是NULL，即在Win32环境下，这个参数不再起作用。
* **lpCmdLine**：是一个以空终止的字符串， 指定传递给应用程序的命令行参数，相当于C或C++中的main函数中的参数char \*argv[]。
* **nShowCmd**：表示一个窗口的显示，表示它是要最大化显示、最小化显示、正常大小显示还是隐藏显示。

3) 创建一个窗口

创建一个完整的窗口，需要经过下面几个步骤：

1. 设计一个窗口类
2. 注册窗口类
3. 创建窗口
4. 显示及更新窗口
5. 设计一个窗口类

一个完整的窗口具有许多特征， 包括光标（鼠标进入该窗口时的形状）、图标、背景色等。窗口的创建过程类似于汽车的制造过程。

我们在生产一个型号的汽车之前， 首先要对该型号的汽车进行设计， 在图纸上画出汽车的结构图， 设计各个零部件， 同时还要给该型号的汽车取一个响亮的名字， 例如“宝马 x6”。

类似地， 在创建一个窗口前， 也必须对该类型的窗口进行设计， 指定窗口的特征。在Windows中，窗口的特征就是由WNDCLASS结构体来定义的，我们只需给WNDCLASS结构体对应的成员赋值，即可完成窗口类的设计。

WNDCLASS结构体的定义如下：

typedef struct \_WNDCLASS{

*UINT* *style*;

*WNDPROC* *lpfnWndProc*;

int *cbClsExtra*;

int *cbWndExtra*;

*HINSTANCE* hInstance;

*HICON* *hIcon*;

*HCURSOR* *hCursor*;

*HBRUSH* *hbrBackground*;

*LPCWSTR* *lpszMenuName*;

*LPCWSTR* *lpszClassName*;

} WNDCLASS;

* **style**：指定窗口的样式(风格)，常用的样式如下：

| **类型** | **含义** |
| --- | --- |
| CS\_HREDRAW | 当窗口水平方向上的宽度发生变化时， 将重新绘制整个窗口。 当窗口发生重绘时， 窗口中的文字和图形将被擦除。如果没有指定这一样式，那么在水平方向上调整窗口宽度时，将不会重绘窗口。 |
| CS\_VREDRAW | 当窗口垂直方向上的高度发生变化时，将重新绘制整个窗口。如果没有指定这一样式，那么在垂直方向上调整窗口高度时，将不会重绘窗口。 |
| CS\_NOCLOSE | 禁用系统菜单的 Close 命令，这将导致窗口没有关闭按钮。 |
| CS\_DBLCLKS | 当用户在窗口中双击鼠标时，向窗口过程发送鼠标双击消息。 |

* **lpfnWndProc**：指定一个窗口回调函数，是一个函数的指针。

当应用程序收到给某一窗口的消息时，就应该调用某一函数来处理这条消息。这一调用过程不用应用程序自己来实施，而由操作系统来完成，但是，回调函数本身的代码必须由应用程序自己完成。对于一条消息，操作系统调用的是接受消息的窗口所属的类型中的lpfnWndProc成员指定的函数。每一种不同类型的窗口都有自己专用的回调函数，该函数就是通过lpfnWndProc成员指定的。

回调函数的定义形式如下：

*LRESULT* *CALLBACK* WindowProc(

*HWND* hWnd, //信息所属的窗口句柄

*UINT* uMsg, //消息类型

*WPARAM* wParam, //附加信息(如键盘哪个键按下)

*LPARAM* lParam //附加信息(如鼠标点击坐标)

);

* **cbClsExtra**：类的附加内存，通常数情况下为0。
* **cbWndExtra**：窗口附加内存，通常情况下为0。
* **hInstance**：当前实例句柄，用WinMain中的形参hInstance为其赋值。
* **hIcon**：指定窗口类的图标句柄，设置为NULL，则使用默认图标，也可用如下函数进行赋值：

*HICON* LoadIcon(*HINSTANCE* hInstance, *LPCTSTR* lpIconName);

如：LoadIcon(*NULL*, *IDI\_WARNING*); //第一个参数为NULL，加载系统默认图标

* **hCursor**：指定窗口类的光标句柄，设置为NULL，则使用默认图标，也可用如下函数进行赋值：

*HCURSOR* LoadCursor(*HINSTANCE* hInstance, *LPCTSTR* lpCursorName);

如：LoadCursor(*NULL*, *IDC\_HELP*); //第一个参数为NULL，加载系统默认光标

* **hbrBackground**：指示窗口的背景颜色，可用如下函数进行赋值：

*HGDIOBJ* GetStockObject(int fnObject);

如：GetStockObject(*WHITE\_BRUSH*);

* **lpszMenuName**：指定菜单资源的名字。如果设置为NULL，那么基于这个窗口类创建的窗口将没有默认菜单。
* **lpszClassName**：指定窗口类的名字。

示例代码如下：

*WNDCLASS* wc; //窗口类变量

wc.*cbClsExtra* = 0; //类附加内存

wc.*cbWndExtra* = 0; //窗口附加内存

wc.*hbrBackground* = (*HBRUSH*)*GetStockObject*(*WHITE\_BRUSH*); //背景色为白色

wc.*hCursor* = (*HCURSOR*)*LoadCursor*(*NULL*, *IDC\_HELP*); //帮助光标

wc.*hIcon* = (*HICON*)*LoadIcon*(*NULL*, *IDI\_WARNING*); //警告图标

wc.*hInstance* = hInstance; //应用程序实例，为WinMain第1个形参

wc.*lpfnWndProc* = WinProc; //窗口过程函数名字

wc.*lpszClassName* = *TEXT*("MyWin"); //类的名字

wc.*lpszMenuName* = *NULL*; //没有菜单

wc.*style* = 0; //类的风格，填0，使用默认风格

1. 注册窗口类

设计完窗口类（WNDCLASS）后， 需要调用RegisterClass函数对其进行注册，注册成功后，才可以创建该类型的窗口。

注册函数的原型声明如下：

*ATOM* RegisterClass(*CONST* *WNDCLASS* \*lpWndClass);

使用示例：RegisterClass(&wc);

1. 创建窗口

设计好窗口类并且将其成功注册之后， 即可用CreateWindow函数产生这种类型的窗口了。

CreateWindow函数的原型声明如下：

*HWND* *CreateWindow*(

*LPCTSTR* lpClassName,

*LPCTSTR* lpWindowName,

*DWORD* *dwStyle*,

int *x*,

int *y*,

int nWidth,

int nHeight,

*HWND* *hWndParent*,

*HMENU* *hMenu*,

*HINSTANCE* hInstance,

*LPVOID* lpParam);

参数说明：

* **lpClassName**：指定窗口类的名称，此名字必须和WNDCLASS的lpszClassName成员指定的名称一样。
* **lpWindowName**：指定窗口的名字，即窗口的标题。
* **dwStyle**：指定创建的窗口的样式，常指定为指WS\_OVERLAPPEDWINDOW类型，这是一种多种窗口类型的组合类型。
* **x, y**：指定窗口左上角的x，y坐标。如果参数x被设为CW\_USEDEFAULT，那么系统为窗口选择默认的左上角坐标并忽略y参数。
* **nWidth，nHeight**：指定窗口窗口的宽度，高度。如果参数nWidth被设为 CW\_USEDEFAULT，那么系统为窗口选择默认的宽度和高度，参数nHeight被忽略。
* **hWndParent**：指定被创建窗口的父窗口句柄，没有父窗口，则设置NULL。
* **hMenu**：指定窗口菜单的句柄，没有，则设置为NULL。
* **hInstance**：窗口所属的应用程序实例的句柄，用WinMain中的形参hInstance为其赋值。
* **lpParam**：作为WM\_CREATE消息的附加参数lParam传入的数据指针。通常设置为NULL。

返回值说明：如果窗口创建成功，CreateWindow函数将返回系统为该窗口分配的句柄，否则，返回NULL。

示例代码：

*HWND* hWnd = *CreateWindow*(

*TEXT*("MyWin"), //窗口类名字

*TEXT*("测试"), //窗口标题

*WS\_OVERLAPPEDWINDOW*, //窗口风格

*CW\_USEDEFAULT*, *CW\_USEDEFAULT*, //窗口x，y坐标，使用默认值

*CW\_USEDEFAULT*, *CW\_USEDEFAULT*, //窗口宽度，高度，使用默认值

*NULL*, //无父窗口

*NULL*, //无菜单

hInstance, //应用程序实例句柄，为WinMain第1个形参

*NULL*); //附件信息，通常设置为NULL

1. 显示及更新窗口

显示窗口函数原型：

*BOOL* *ShowWindow*(*HWND* hWnd, int *nCmdShow*);

更新窗口函数原型：

*BOOL* *UpdateWindow*(*HWND* hWnd);

示例代码：

*ShowWindow*(hWnd, *SW\_SHOWNORMAL*); //SW\_SHOWNORMAL为普通模式

*UpdateWindow*(hWnd);

1. 示例代码

//设计一个窗口类

*WNDCLASS* wc; //窗口类变量

wc.*cbClsExtra* = 0; //类附加内存

wc.*cbWndExtra* = 0; //窗口附加内存

wc.*hbrBackground* = (*HBRUSH*)*GetStockObject*(*WHITE\_BRUSH*); //背景色为白色

wc.*hCursor* = *LoadCursor*(*NULL*, *IDC\_HELP*); //帮助光标

wc.*hIcon* = *LoadIcon*(*NULL*, *IDI\_WARNING*); //警告图标

wc.*hInstance* = hInstance; //应用程序实例，为WinMain第1个形参

wc.*lpfnWndProc* = WinProc; //窗口过程函数名字

wc.*lpszClassName* = *TEXT*("MyWin"); //类的名字

wc.*lpszMenuName* = *NULL*; //没有菜单

wc.*style* = 0; //类的风格，填0，使用默认风格

//注册窗口类

*RegisterClass*(&wc);

//创建窗口

*HWND* hWnd = *CreateWindow*(*TEXT*("MyWin"), *TEXT*("测试"), *WS\_OVERLAPPEDWINDOW*, *CW\_USEDEFAULT*, *CW\_USEDEFAULT*, *CW\_USEDEFAULT*, *CW\_USEDEFAULT*, *NULL*, *NULL*, hInstance, *NULL*);

//显示及更新窗口

*ShowWindow*(hWnd, *SW\_SHOWNORMAL*);

*UpdateWindow*(hWnd);

4) 消息循环

在创建窗口、显示窗口、更新窗口后，我们需要编写一个消息循环，不断地从消息队列中取出消息，并进行响应。

a) 消息结构体

在Windows程序中，消息是由MSG结构体来表示的。MSG结构体的定义如下：

typedef struct tagMSG {

*HWND* *hWnd*;

*UINT* *message*;

*WPARAM* *wParam*;

*LPARAM* *lParam*;

*DWORD* *time*;

*POINT* *pt*;

} MSG;

* **hWnd**：消息所属的窗口。我们通常开发的程序都是窗口应用程序，一个消息一般都是与某个窗口相关联的。例如，在某个活动窗口中按下鼠标左键，产生的按键消息就是发给该窗口的。
* **message**：消息的标识符，是由一个数值来表示的，不同的消息对应不同的数值。Windows将消息对应的数值定义为WM\_XXX宏(WM是Windows Message的缩写)的形式， XXX对应某种消息的英文拼写的大写形式。例如，鼠标左键按下消息是WM\_LBUTTONDOWN，键盘按下消息是WM\_KEYDOWN，字符消息是 WM\_CHAR……。
* **wParam**： 指定消息的附加信息，如键盘按下会触发WM\_KEYDOWN消息，但是，具体按下哪个按键需要wParam区分。
* **lParam**：指定消息的附加信息，如鼠标左击会触发WM\_LBUTTONDOWN消息，但是，具体点击的坐标需要lParam区分。
* **time**：标识一个消息产生时的时间。
* **pt**：表示产生这个消息时光标或鼠标的坐标。

b) 取消息

要从消息队列中取出消息，我们需要调用GetMessage()函数，该函数的原型声明如下：

*BOOL* GetMessage(

*LPMSG* lpMsg,

*HWND* hWnd,

*UINT* wMsgFilterMin,

*UINT* wMsgFilterMax);

参数说明：

* **lpMsg**：指向一个消息结构体(MSG)，GetMessage从线程的消息队列中取出的消息信息将保存在该结构体变量中。
* **hWnd**：指定接收属于哪一个窗口的消息。通常我们将其设置为NULL，用于接收属于调用线程的所有窗口的窗口消息。
* **wMsgFilterMin**：指定消息的最小值。
* **wMsgFilterMax**：指定消息的最大值。如果wMsgFilterMin和wMsgFilterMax都设置为0， 则接收所有消息。

返回值说明：GetMessage函数接收到除 WM\_QUIT 外的消息均返回非零值。对于WM\_QUIT消息，该函数返回零。如果出现了错误，该函数返回-1，例如，当参数hWnd是无效的窗口句柄或lpMsg是无效的指针时。

c) 建立消息循环

*MSG* msg;

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

{

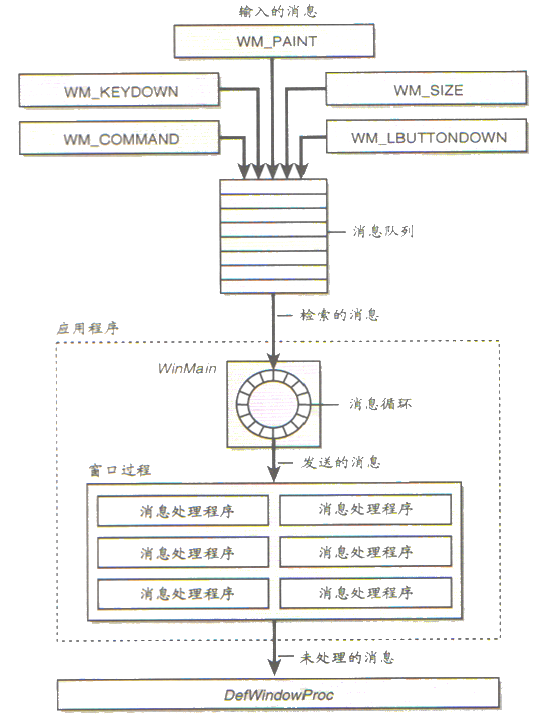
*TranslateMessage*(&msg);

*DispatchMessage*(&msg);

}

* **TranslateMessage**：用于翻译、处理和转换消息并把新消息投放到消息队列中，并且此过程不会影响原来的消息队列。
* **DispatechMessage**：用于把收到的消息传到窗口回调函数进行分析和处理。即将消息传递给操作系统，让操作系统调用窗口回调函数，来对信息进行处理。

d) 消息处理机制



1. 操作系统接收到应用程序的窗口消息，将消息投递到该应用程序的消息队列中。
2. 应用程序在消息循环中调用GetMessage函数从消息队列中取出一条一条的消息。取出消息后，应用程序可以对消息进行一些预处理，例如，放弃对某些消息的响应，或者调用TranslateMessage产生新的消息。
3. 应用程序调用DispatchMessage，将消息回传给操作系统。消息是由 MSG结构体对象来表示的，其中就包含了接收消息的窗口的句柄。因此， DispatchMessage函数总能进行正确的传递。
4. 系统利用WNDCLASS结构体的lpfnWndProc成员保存的窗口过程函数的指针调用窗口过程，对消息进行处理。

5) 窗口过程函数(消息处理函数)

在完成上述步骤后，剩下的工作就是编写一个窗口过程函数，用于处理发送给窗口的消息。

窗口过程函数的名字可以随便取， 如WinProc， 但函数定义的形式必须和下面声明的形式相同：

*LRESULT* *CALLBACK* WinProc( //CALLBACK 和WINAPI 作用一样

*HWND* hWnd, //信息所属的窗口句柄

*UINT* uMsg, //消息类型

*WPARAM* wParam, //附加信息(如键盘哪个键按下)

*LPARAM* lParam //附加信息(如鼠标点击坐标)

);

示例代码：

*LRESULT* *CALLBACK* WinProc(

*HWND* hWnd, //信息所属的窗口句柄

*UINT* uMsg, //消息类型

*WPARAM* wParam, //附加信息(如键盘按键)

*LPARAM* lParam //附加信息(如鼠标点击坐标)

)

{

switch (uMsg)

{

case *WM\_KEYDOWN*: //键盘按下

//……

break;

case *WM\_LBUTTONDOWN*: //鼠标右键按下

//……

break;

case *WM\_PAINT*: //绘图事件

//……

break;

case *WM\_DESTROY*:

*PostQuitMessage*(0);

break;

case *WM\_CLOSE*:

*DestroyWindow*(hWnd);

break;

default:

//以windows默认方式处理

return *DefWindowProc*(hWnd, uMsg, wParam, lParam);

break;

}

return 0;

}

* **DefWindowProc函数**：DefWindowProc函数调用默认的窗口过程，对应用程序没有处理的其他消息提供默认处理。
* **WM\_CLOSE**：对WM\_CLOSE消息的响应并不是必须的，如果应用程序没有对该消息进行响应，系统将把这条消息传给DefWindowProc函数而 DefWindowProc函数则调用DestroyWindow函数来响应这条WM\_CLOSE消息。
* **WM\_DESTROY**：DestroyWindow函数在销毁窗口后，会给窗口过程发送 WM\_DESTROY消息，我们在该消息的响应代码中调用PostQuitMessage函数。

PostQuitMessage函数向应用程序的消息队列中投递一条WM\_QUIT消息并返回。WinMain函数中，GetMessage 函数只有在收到WM\_QUIT消息时才返回0，此时消息循环才结束，程序退出。传递给 PostQuitMessage函数的参数值将作为WM\_QUIT消息的wParam参数，这个值通常用做WinMain函数的返回值。

6) 完整示例代码

#include <windows.h>

//窗口过程函数

*LRESULT* *CALLBACK* WinProc(

*HWND* hWnd, //信息所属的窗口句柄

*UINT* uMsg, //消息类型

*WPARAM* wParam, //附加信息(如键盘按键)

*LPARAM* lParam //附加信息(如鼠标点击坐标)

)

{

switch (uMsg)

{

case *WM\_KEYDOWN*: //键盘按下

*MessageBox*(hWnd, *TEXT*("键盘按下"), *TEXT*("键盘"), *MB\_OK*);

break;

case *WM\_DESTROY*:

*PostQuitMessage*(0);

break;

default:

//以windows默认方式处理

return *DefWindowProc*(hWnd, uMsg, wParam, lParam);

}

return 0;

}

//程序入口地址

int *WINAPI* *WinMain*(

*HINSTANCE* hInstance, //应用程序实例

*HINSTANCE* hPrevInstance, //上一个应用程序实例

*LPSTR* lpCmdLine, //命令行参数

int nShowCmd) //窗口显示的样式

{

//设计一个窗口类

*WNDCLASS* wc; //窗口类变量

wc.*cbClsExtra* = 0; //类附加内存

wc.*cbWndExtra* = 0; //窗口附加内存

wc.*hbrBackground* = (*HBRUSH*)*GetStockObject*(*WHITE\_BRUSH*); //背景色为白色

wc.*hCursor* = *LoadCursor*(*NULL*, *IDC\_HELP*); //帮助光标

wc.*hIcon* = *LoadIcon*(*NULL*, *IDI\_WARNING*); //警告图标

wc.*hInstance* = hInstance; //应用程序实例，为WinMain第1个形参

wc.*lpfnWndProc* = WinProc; //窗口过程函数名字

wc.*lpszClassName* = *TEXT*("MyWin"); //类的名字

wc.*lpszMenuName* = *NULL*; //没有菜单

wc.*style* = 0; //类的风格，填0，使用默认风格

//注册窗口类

*RegisterClass*(&wc);

//创建窗口

*HWND* hWnd = *CreateWindow*(*TEXT*("MyWin"), *TEXT*("测试"), *WS\_OVERLAPPEDWINDOW*, *CW\_USEDEFAULT*, *CW\_USEDEFAULT*, *CW\_USEDEFAULT*, *CW\_USEDEFAULT*, *NULL*, *NULL*, hInstance, *NULL*);

//显示及更新窗口

*ShowWindow*(hWnd, *SW\_SHOWNORMAL*);

*UpdateWindow*(hWnd);

//消息循环

*MSG* msg;

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

{

*TranslateMessage*(&msg); //翻译

*DispatchMessage*(&msg); //分发信息

}

return msg.*wParam*;

}

1.3 MFC入门

1.3.1 MFC是什么?

微软基础类库（英语：Microsoft Foundation Classes，简称MFC）是一个微软公司提供的类库（class libraries），以C++类的形式封装了Windows API，并且包含一个应用程序框架，以减少应用程序开发人员的工作量。其中包含的类包含大量Windows句柄封装类和很多Windows的内建控件和组件的封装类。

MFC把Windows SDK API函数包装成了几百个类，MFC给Windows操作系统提供了面向对象的接口，支持可重用性、自包含性以及其他OPP原则。MFC通过编写类来封装窗口、对话框以及其他对象，引入某些关键的虚函数（覆盖这些虚函数可以改变派生类的功能）来完成，并且MFC设计者使类库带来的总开销降到了最低。

1.3.2 编写第一个MFC应用程序

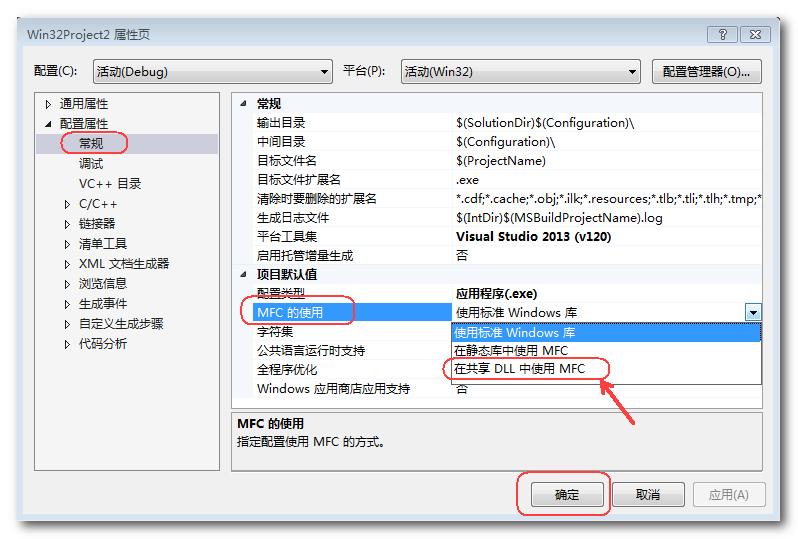
1) 代码的编写

项目的创建和之前一样，只是此次的源文件后缀为.cpp，因为MFC是由C++编写的，编写MFC程序需要包含#include <afxwin.h>头文件。



配置环境后，代码才可编译运行：





2) 程序执行流程

1. 程序开始时，先实例化应用程序对象(有且只有一个)
2. 执行程序的入口函数InitInstance()
3. 给框架类MyFrame对象动态分配空间（自动调用它的构造函数），在其构造函数内部，通过CWnd::Create创建窗口
4. 框架类对象显示窗口CWnd::ShowWindow
5. 框架类对象更新窗口CWnd::UpdateWindow
6. 保存框架类对象指针CWinThread::m\_pMainWnd

3) 代码分析

a) CFrameWnd 框架窗口类

CFrameWnd是从CWnd(窗口基类)派生出来的。CFrameWnd模仿框架窗口行为，我们可以把框架窗口作为顶层窗口看待，它是应用程序与外部世界的主要接口。

如果想要创建一个窗口，可以在此类中调用CWnd::Create或CWnd::CreateEX函数：

virtual *BOOL* Create(

*LPCTSTR* lpszClassName,

*LPCTSTR* lpszWindowName,

*DWORD* dwStyle,

const *RECT*& rect,

*CWnd*\* pParentWnd,

*UINT* nID,

*CCreateContext*\* pContext = *NULL*

);

* Create接收的8个参数后6个有默认值定义。
* **lpszClassName**指定了窗口基于WNDCLASS类的名称，为此将其设定为NULL将创建一个基于已注册的WNDCLASS类的默认框架窗口。
* **lpszWindowName**参数指定将在窗口的标题栏出现的文字。

b) CWinApp应用程序类

MFC应用程序的核心就是基于CWinApp类的应用程序对象。CWinApp提供了消息循环来检索消息并将消息调度给应用程序窗口。它还包括可被覆盖的、用来自定义应用程序行为的主要虚函数。

一个MFC程序可以有且仅有一个应用程序对象，此对象必须声明为在全局范围内有效，以便它在程序开始时即在内存中被实例化。

c) InitInstance函数

CWinApp::InitInstance函数是一个虚函数，其默认操作仅有一条语句：

*BOOL* MyApp::InitInstance()//程序入口地址

{

return *TRUE*;

}

InitInstance的目的是给应用程序提供一个自身初始化的机会，其返回值决定了框架接下来要执行的内容，如果返回FALSE将关闭应用程序，如果初始化正常返回TRUE以便允许程序继续进行。此函数是MFC应用程序的入口。

d) m\_pMainWnd 成员变量

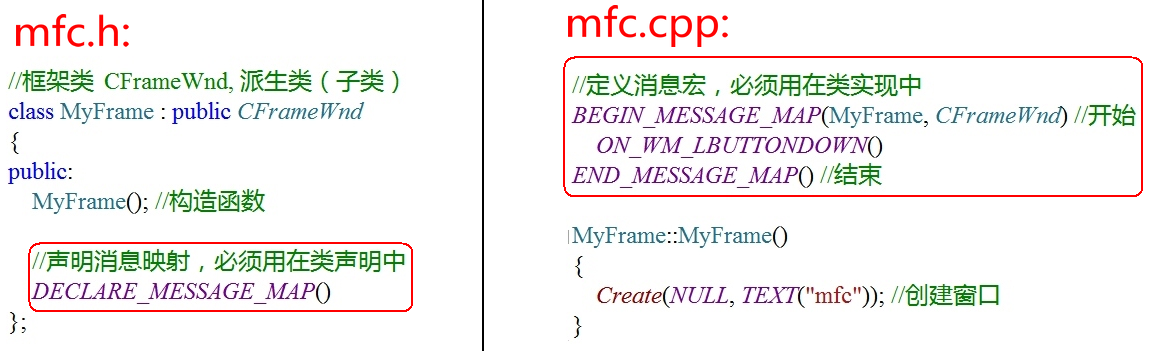
在CWinApp中有一个名为CWinThread::m\_pMainWnd的成员变量。 该变量是一个CWnd类型的指针，它保存了应用程序框架窗口对象的指针。也就是说，是指向CFramWnd对象（框架窗口类对象）的指针。

1.3.3 消息映射

消息映射是一个将消息和成员函数相互关联的表。比如，框架窗口接收到一个鼠标左击消息，MFC将搜索该窗口的消息映射，如果存在一个处理WM\_LBUTTONDOWN消息的处理程序，然后就调用OnLButtonDown。

下面是是将消息映射添加到一个类中所做的全部工作：

1. 所操作类中，声明消息映射宏。
2. 通过放置标识消息的宏来执行消息映射，相应的类将在对BEGIN\_MESSAGE\_MAP和END\_MESSAGE\_MAP的调用之间处理消息。

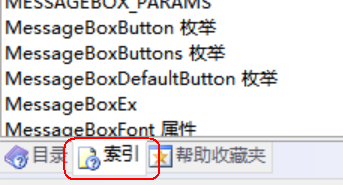


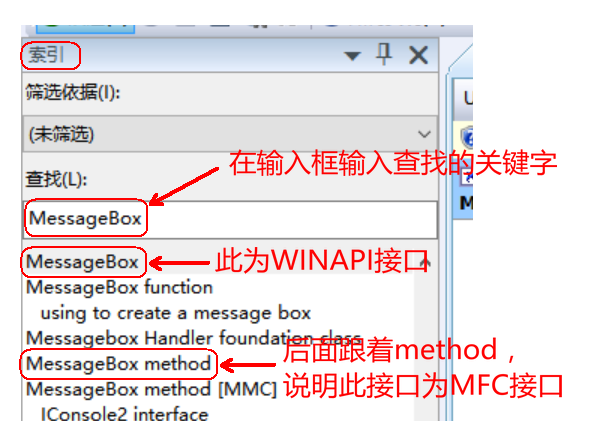
1. 对应消息处理函数分别在类中声明，类外定义：

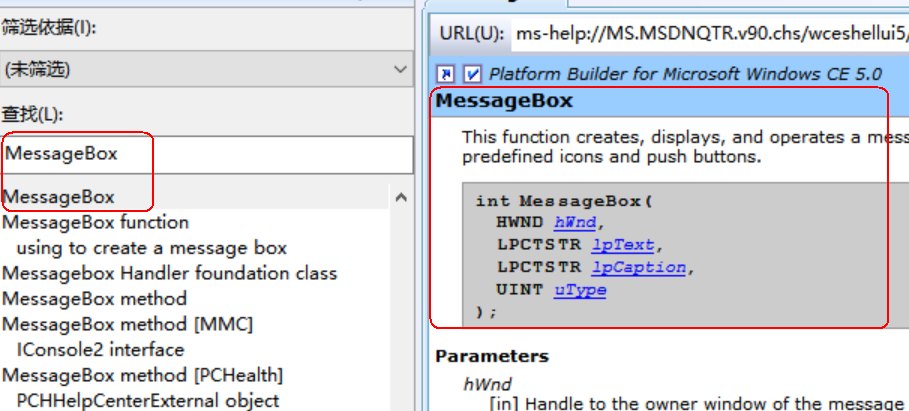


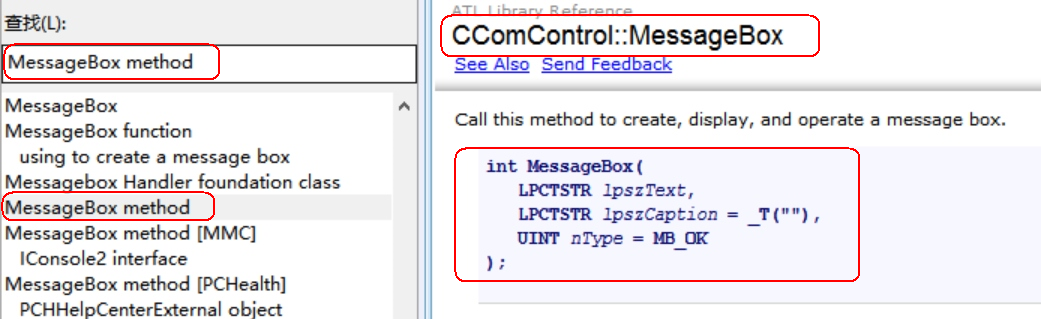
1.3.4 帮助文档的使用

1) MSDN的使用



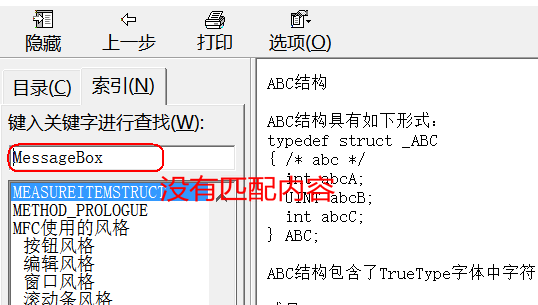


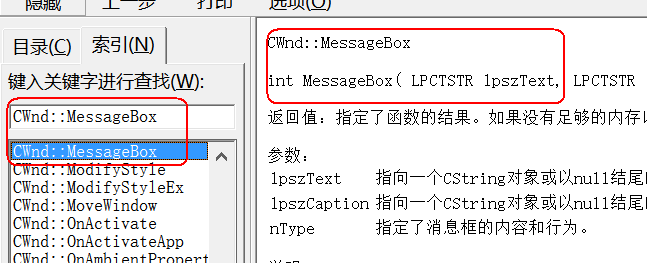




2) VC++之MFC类库中文手册

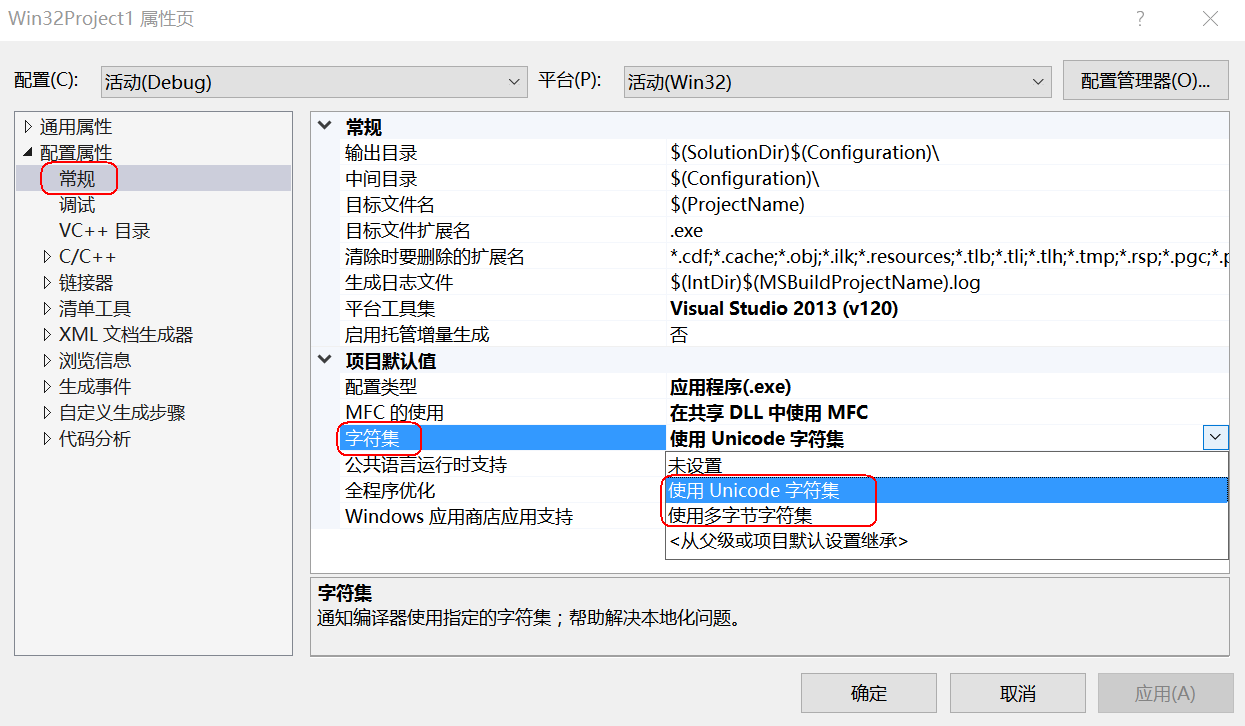
通过此手册查找，必须加上成员所属的类作用域(类名::)，否则，无法查找到匹配的关键字。





1.3.5 Widnows字符集





1) 多字节字符集(8位的ANSI字符集)

在Windows98以及以前的版本使用8位ANSI字符集，它类似于我们程序员熟悉的ASCII字符集。

char sz[] = "ABCDEFG";

char \*psz = "ABCDEFG";

int len = *strlen*(sz);

2) 宽字符集(16位的Unicode字符集)

在WindowsNT和Windows2000后开始使用16位的Unicode字符集，它是ANSI字符集的一个超集。Unicode适用于国际市场销售的应用程序，因为它包含各种各样来自非U.S.字母表的字符，比如中文，日文，韩文，西欧语言等。

//在字符串前加字母L表示将ANSI字符集转换成Unicode字符集。

wchar\_t wsz[] = L"ABCDEFG";

wchar\_t \*pwsz = L"ABCDEFG";

int len = *wcslen*(wsz); //测试宽字节字符串的长度

3) TEXT（\_T）宏

MFC中的TEXT宏可以自动适应字符类型，如果定义了预处理器程序符号\_UNICODE，那么编译器将使用Unicode字符，如果没用定义该预处理器程序符号，那么编译器将使用ANSI字符。

*MessageBox*(*TEXT*("鼠标左键"));

*MessageBox*(*\_T*("鼠标左键"));

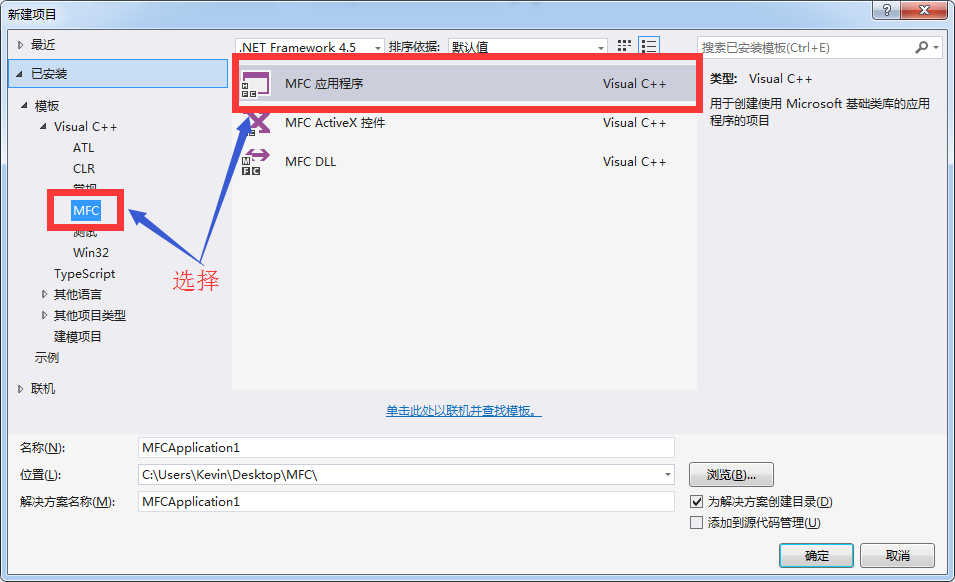
4) TCHAR类型

如果定义了\_UNICODE符号TCHAR将变为wchar\_t类型。如果没用定义\_UNICODE符号，TCHAR将变为普通古老的char类型。

1.4 用向导生成一个MFC应用程序

1.4.1 向导流程

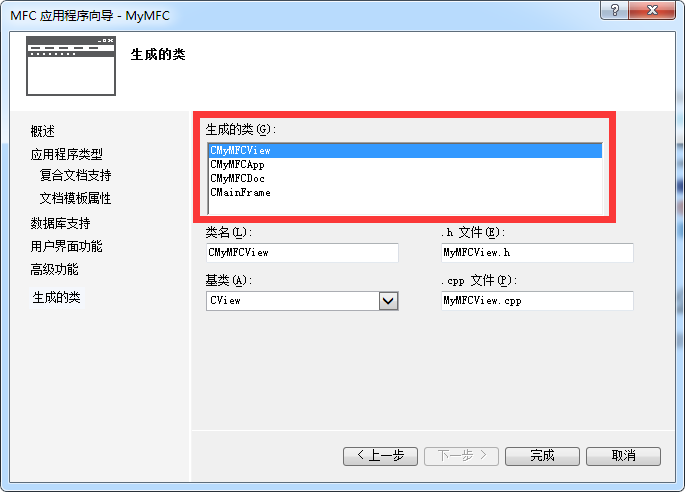
在VS中选择“文件” -- “新建” -- “项目”：



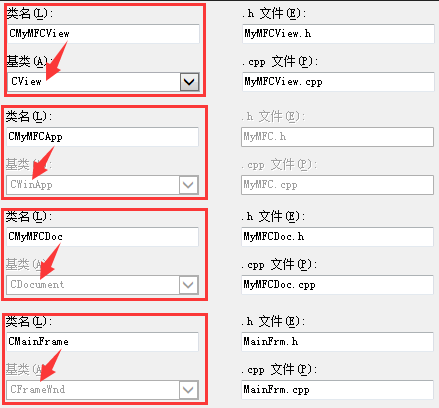
选择 MFC – MFC应用程序，接下来我们创建一个单文档MFC标准类型应用程序。



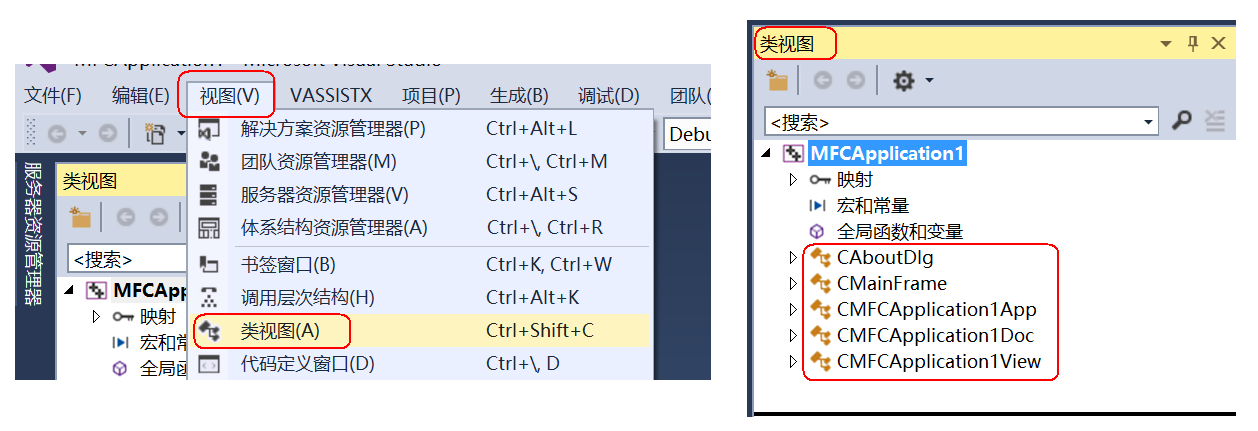
一路按默认值next，到最后一个页面：

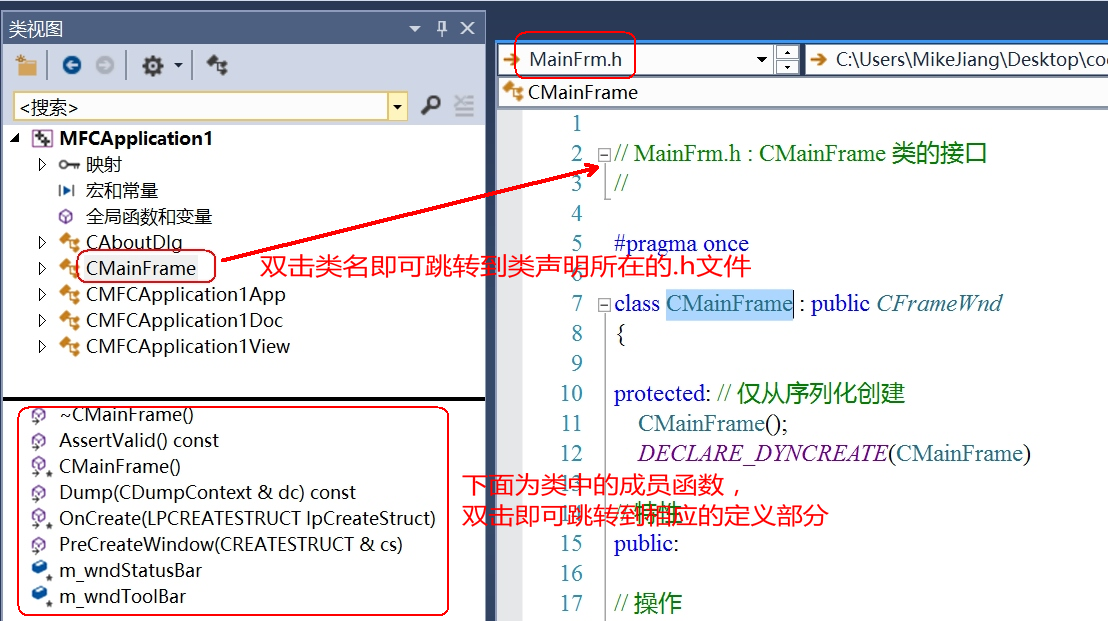


MFC自动为我们生成了四个类，它们的继承关系如下：



1.4.2 类视图



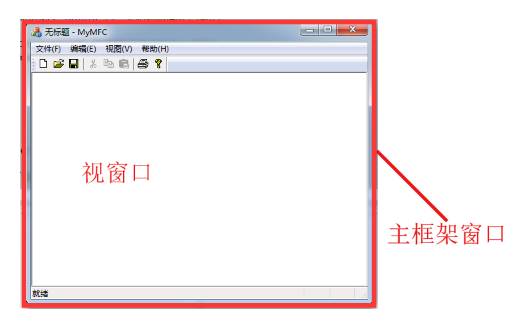


1.4.3 文档/视图结构体系

MFC应用程序框架结构的基石是文档/视图体系结构，它定义了一种程序结构，这种结构依靠文档对象保存应用程序的数据，并依靠视图对象控制视图中显示的数据，把数据本身与它的显示分离开。

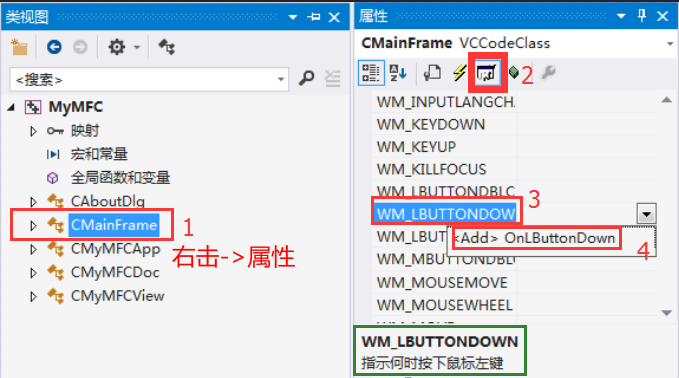
数据的存储和加载由文档类来完成，数据的显示和修改则由视类来完成。 MFC在类CDocument和CView中为稳定视图提供了基础结构。CWinApp、CFrameWnd和其他类与CDocument和CView合作，把所有的片段连在了一起。

CView类也派生于CWnd类，框架窗口是视图窗口的一个父窗口。主框架窗口（CFrameWnd）是整个应用程序外框所包括的部分，即图中粗框以内的内容，而视类窗口只是主框架中空白的地方。



1.4.4 消息处理的添加

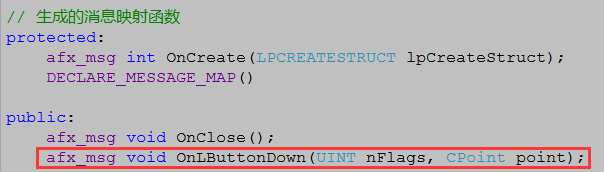
在主框架类中添加WM\_LBUTTONDOWN消息的响应函数，具体操作如下：



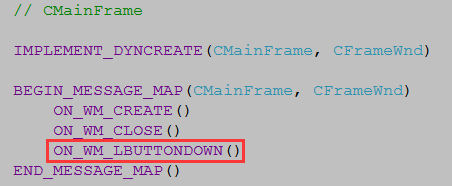
从类视图中找到CMainFrame（继承自CFrameWnd），选择此类然后从属性面板中找到消息按钮，在消息列表中找到WM\_LBUTTONDOWN消息，添加。

**工程文件增加几处改变：**

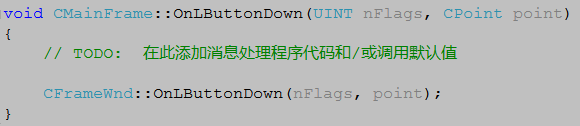
* 第一处：在框架类头文件中添加了鼠标左键消息函数的函数声明



* 第二处：在框架类cpp文件中添加了消息映射宏

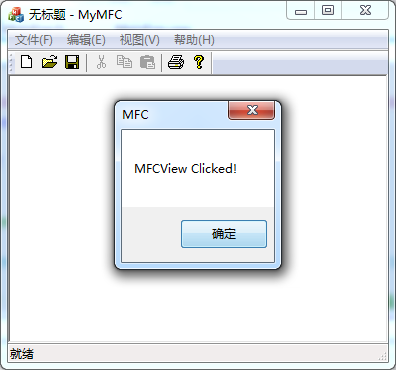


* 第三处：在框架列cpp文件中添加了处理鼠标左键消息的函数定义



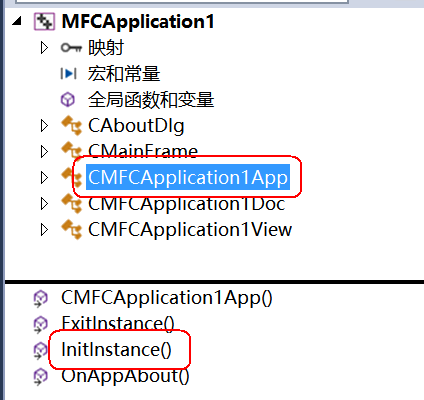
我们再此OnLButtonDown函数中添加一个MessageBox消息，鼠标左键按下弹出一个提示框，然后执行程序。我们会惊奇的发现程序并未如我们所愿弹出消息框。

因为，框架窗口是视窗口的父窗口，那么视类窗口就应该始终覆盖在框架类窗口之上。就好比框架窗口是一面墙，视类窗口就是墙纸，它始终挡在这面墙前边。也就是说，所有操作，包括鼠标单击、鼠标移动等操作都只能有视类窗口捕获。



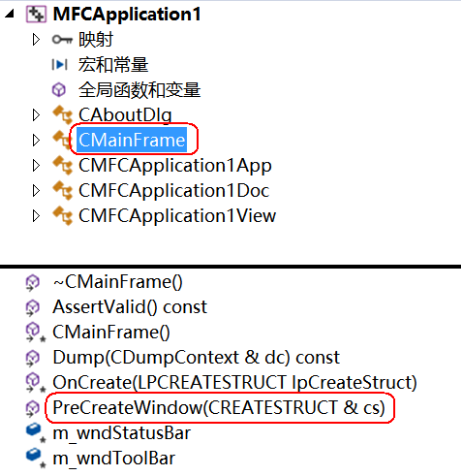
1.4.5 MFC框架中一些重要的函数

1) InitInstance函数



应用程序类的一个虚函数，MFC应用程序的入口。

2) PreCreateWindow函数

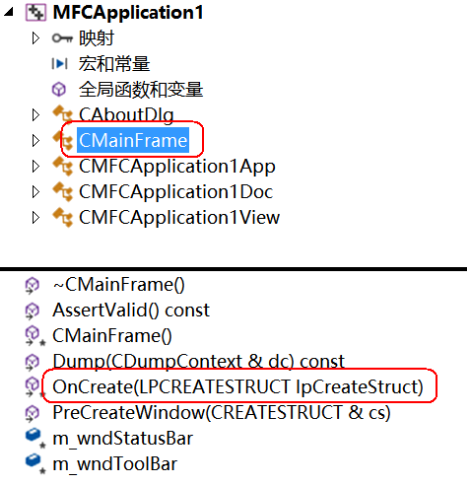


当框架调用CreateEx函数创建窗口时，会首先调用PreCreateWindow函数。

通过修改传递给PreCreateWindow的结构体类型参数CREATESTRUCT，应用程序可以更改用于创建窗口的属性。在产生窗口之前让程序员有机会修改窗口的外观。

最后再调用CreateWindowEx函数完成窗口的创建。

3) OnCreate函数

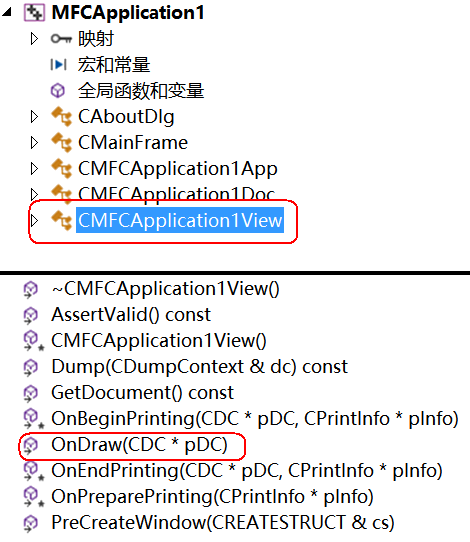


OnCreate是一个消息响应函数，是响应WM\_CREATE消息的一个函数，而WM\_CREATE消息是由Create函数调用的。一个窗口创建（Create）之后，会向操作系统发送WM\_CREATE消息，OnCreate()函数主要是用来响应此消息的。

OnCreate与Create的区别：

* Create()负责注册并产生窗口，像动态创建控件中的Create()一样，窗口创建之后会向操作系统发送WM\_CREATE消息。
* OnCreate()不产生窗口，只是在窗口显示前设置窗口的属性如风格、位置等。
* OnCreate()是消息WM\_CREATE的消息响应函数。

4) OnDraw和OnPaint



OnPaint是WM\_PAINT消息的消息处理函数，在OnPaint中调用OnDraw，一般来说，用户自己的绘图代码应放在OnDraw中。

* OnPaint()是CWnd的类成员，负责响应WM\_PAINT消息。
* OnDraw()是CView的成员函数，没有响应消息的功能。

当视图变得无效时（包括大小的改变，移动，被遮盖等等），Windows发送WM\_PAINT消息。该视图的OnPaint 处理函数通过创建CPaintDC类的DC对象来响应该消息并调用视图的OnDraw成员函数。OnPaint最后也要调用OnDraw,因此一般在OnDraw函数中进行绘制。

通常我们不必编写OnPaint处理函数。当在View类里添加了消息处理OnPaint()时，OnPaint()就会覆盖掉OnDraw()。

1.5 拓展知识点

* MFC中后缀名为Ex的函数都是扩展函数。
* 在MFC中，以Afx为前缀的函数都是全局函数，可以在程序的任何地方调用它们。

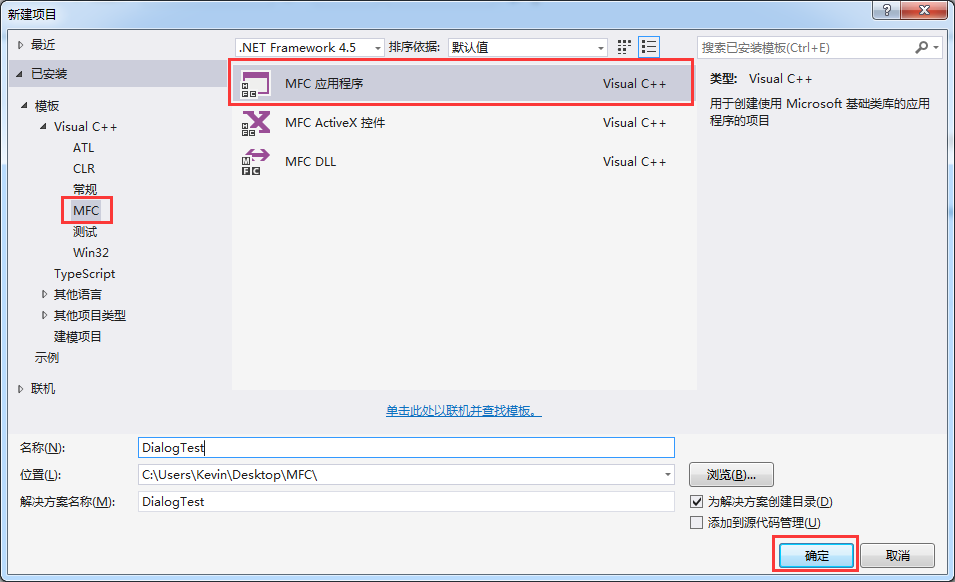
1. 基于对话框编程

对话框是一种特殊类型的窗口，绝大多数Windows程序都通过对话框与用户进行交互。在Visual C++中，对话框既可以单独组成一个简单的应用程序，又可以成为文档/视图结构程序的资源。

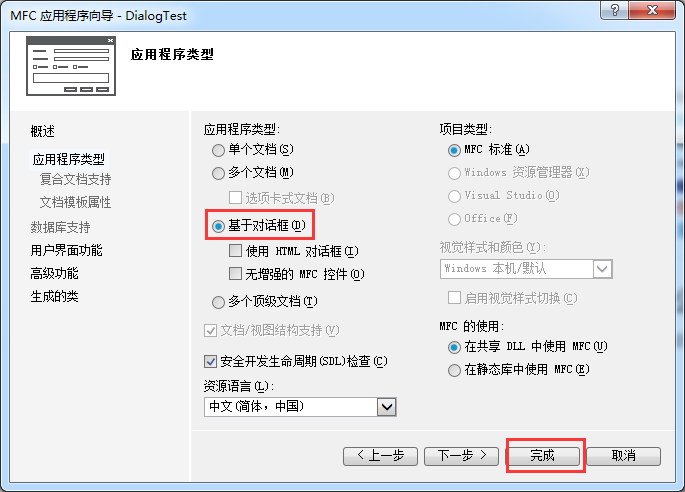
2.1 创建基于对话框的 MFC 应用程序框架

程序的创建过程：

* 选择“文件 | 新建 |　项目”菜单；
* 在“新建项目”对话框中，选择“ MFC 应用程序 ”，输入工程名称，选择“确定”。



* 选择“ 基于对话框”，即创建基于对话框的应用程序，选择“完成”。



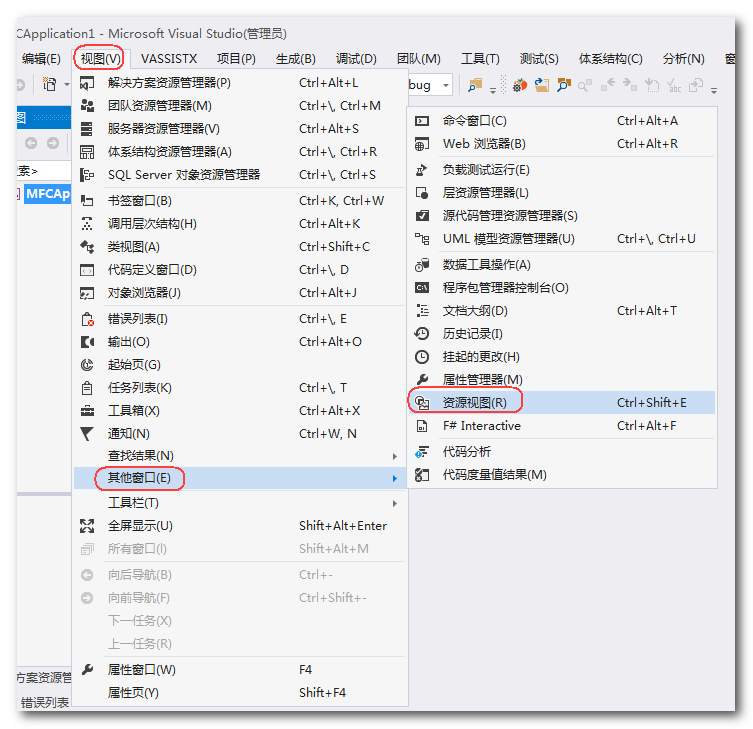
2.2 对话框应用程序框架介绍

2.2.1 资源视图

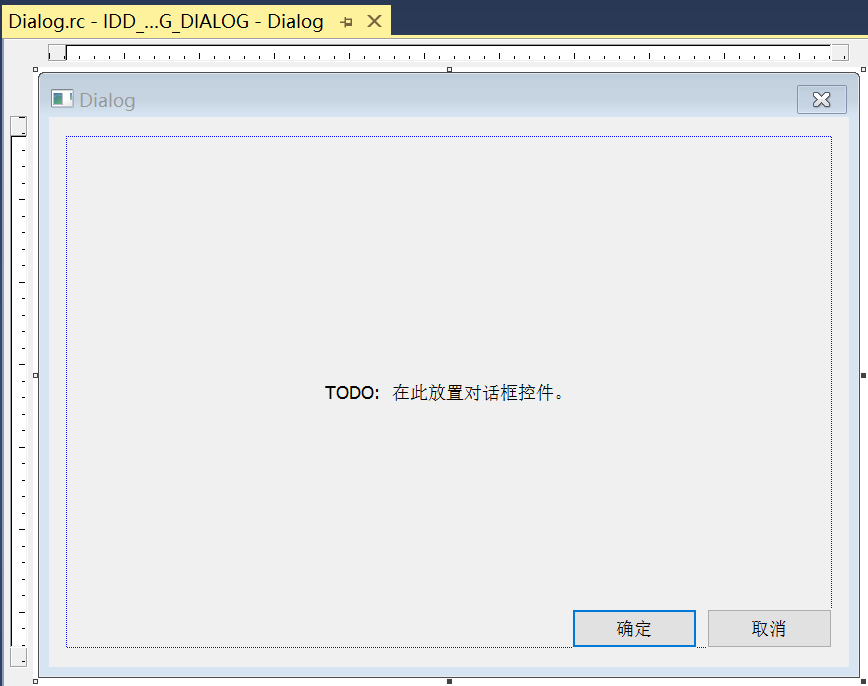
用 AppWizard 创建基于对话框的应用程序框架（假定工程名为 Dialog ）后，项目工作区上增加了一个“资源视图”选项卡。



或者，通过视图找到“资源视图”选项卡：



在 MFC中，与用户进行交互的对话框界面被认为是一种资源。展开“Dialog”，可以看到有一个ID为IDD\_ DIALOG \_DIALOG（中间部分（DIALOG）与项目名称相同）的资源，对应中间的对话框设计界面。不管在何时，只要双击对话框资源的ID，对话框设计界面就会显示在中间。



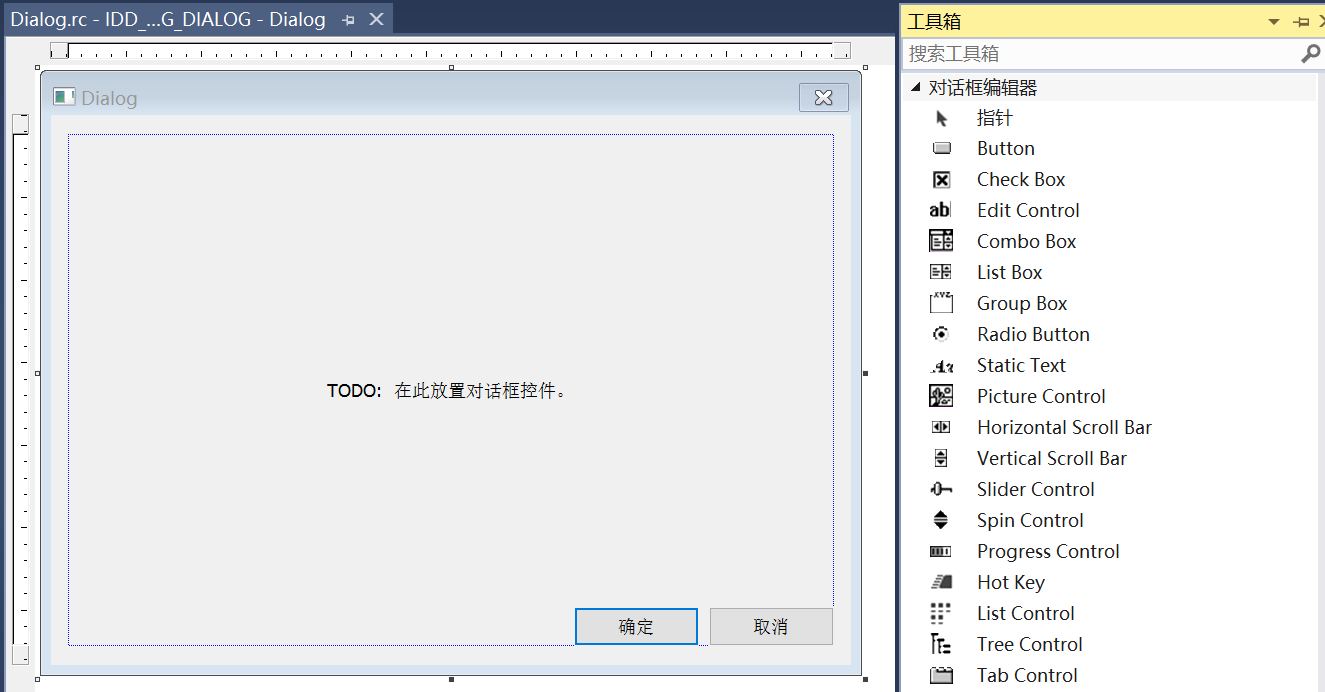
2.2.2 类视图

在类视图中，可以看到生成了3 个类：CAboutDlg、CDialogApp和CDialogDlg。



* **CAboutDlg**：对应生成的版本信息对话框。
* **CDialogApp**：应用程序类，从 CWinApp 继承过来，封装了初始化、运行、终止该程序的代码。
* **CDialogDlg**：对话框类，从CdialogEx继承过来的，在程序运行时看到的对话框就是它的一个具体对象。
  + **DoDataExchange函数**：该函数主要完成对话框数据的交换和校验。
  + **OnInitDialog函数**：相当于对对话框进行初始化处理。

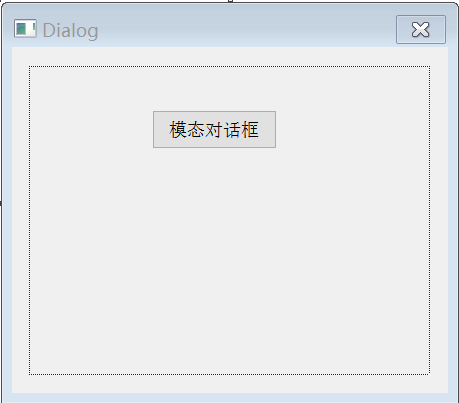
2.2.3 设计界面和工具箱



2.3 模态对话框

当模态对话框显示时，程序会暂停执行，直到关闭这个模态对话框之后，才能执行程序中的其他任务。

1）通过工具箱在界面上放一个Button，双击此按钮即可跳转到按钮处理函数：



//按钮处理函数

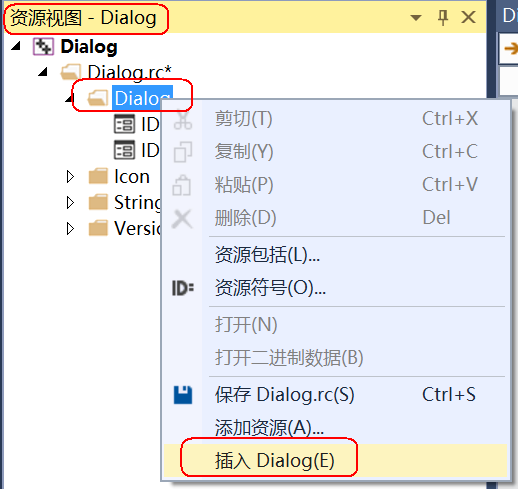
void CDialogDlg::OnBnClickedButton1()

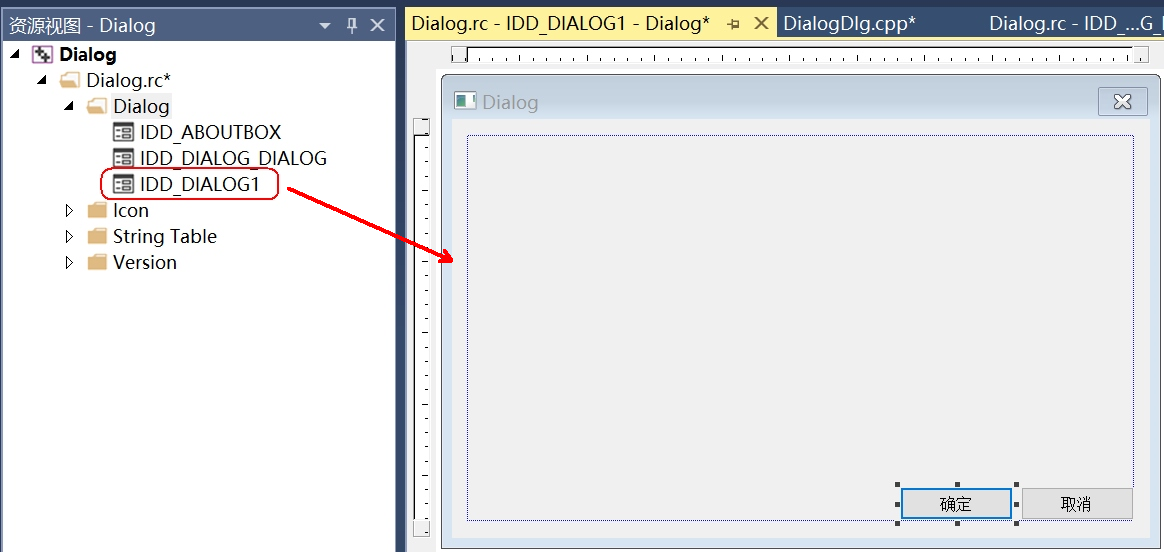
{

// TODO: 在此添加控件通知处理程序代码

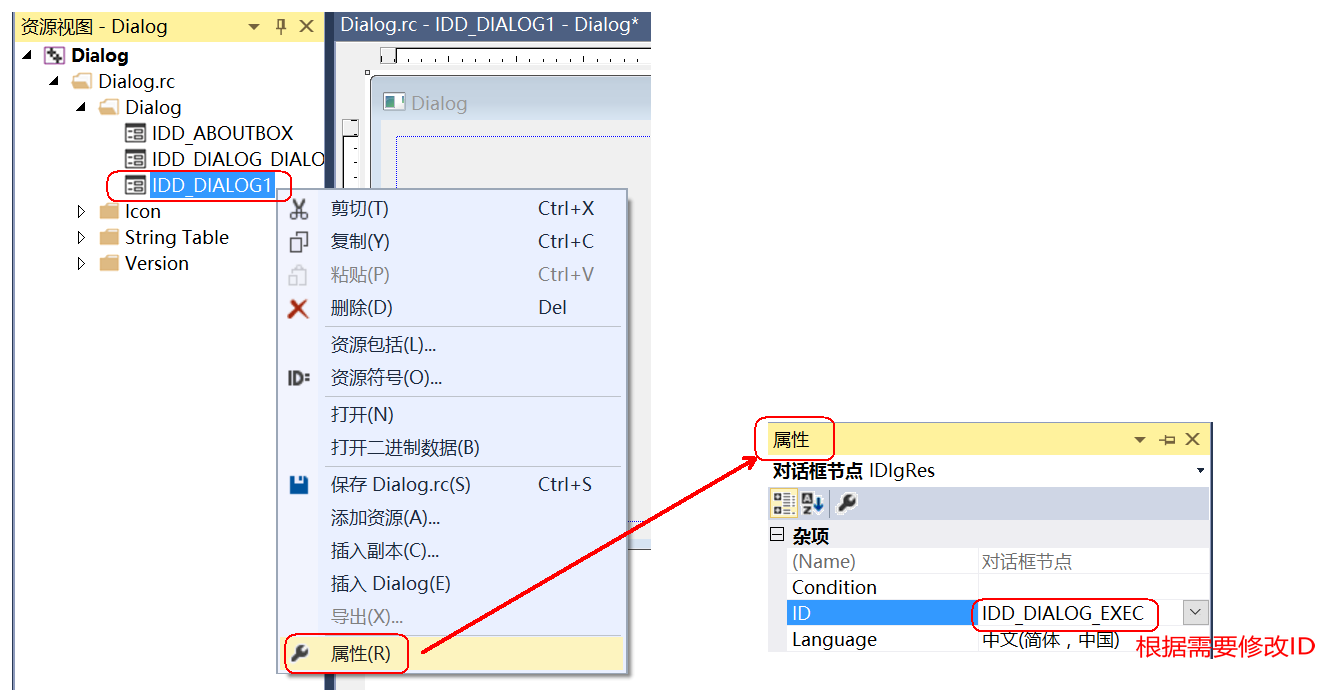
}

2）资源视图 -> Dialog -> 右击 -> 插入 Dialog：

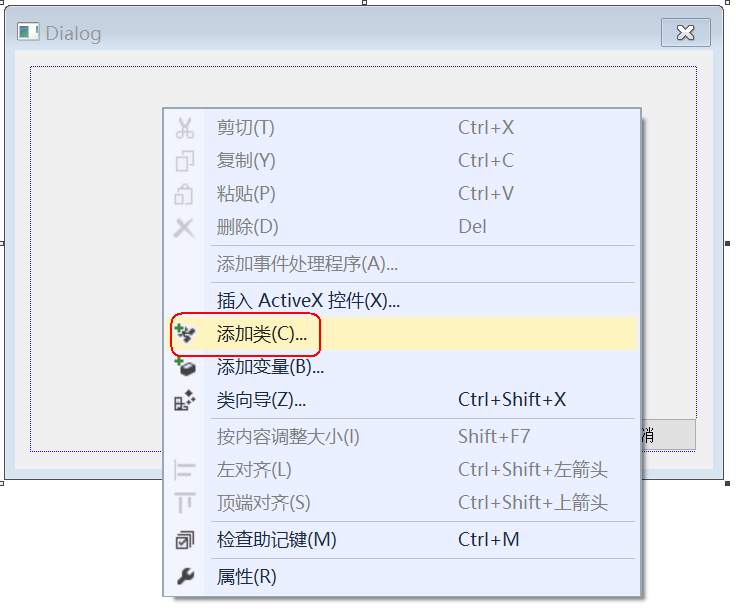


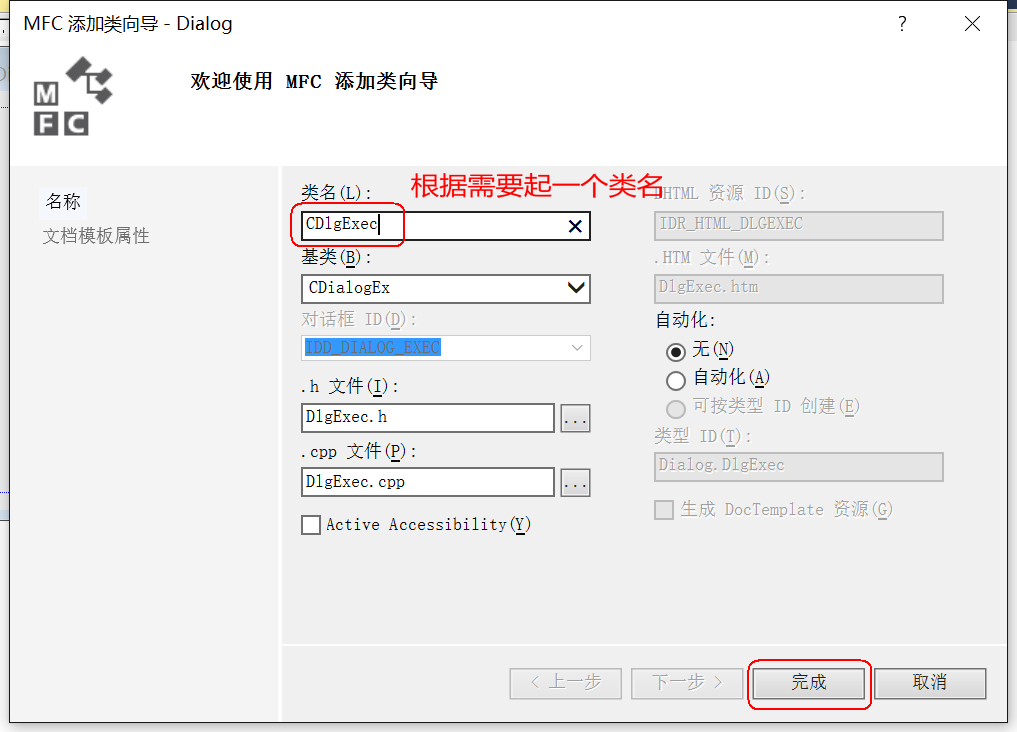


1. 修改对话框ID:

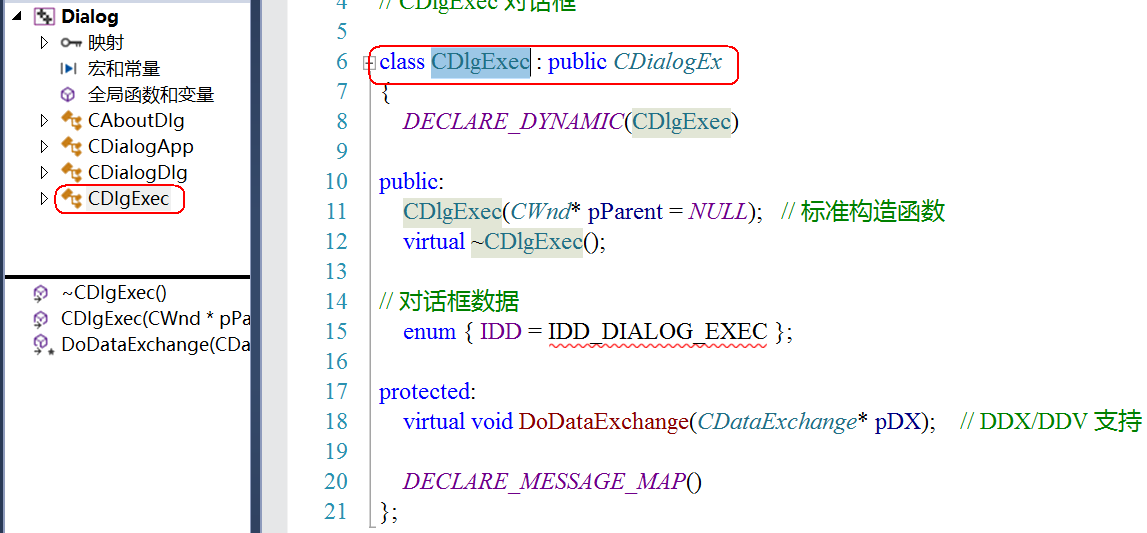


1. 点击对话框模板 -> 右击 -> 添加类：





1. 类视图中多了一个自定义类：



1. 按钮处理函数创建对话框，以模态方式运行。

实现模态对话框的创建需要调用CDialog类的成员函数CDialog::DoModel，该函数的功能就是创建并显示一个对话框:

//启动模态对话框按钮

void CDialogDlg::OnBnClickedButton1()

{

//需要包含头文件：#include "DlgExec.h"

CDlgExec dlg;

dlg.*DoModal*(); //以模态方式运行

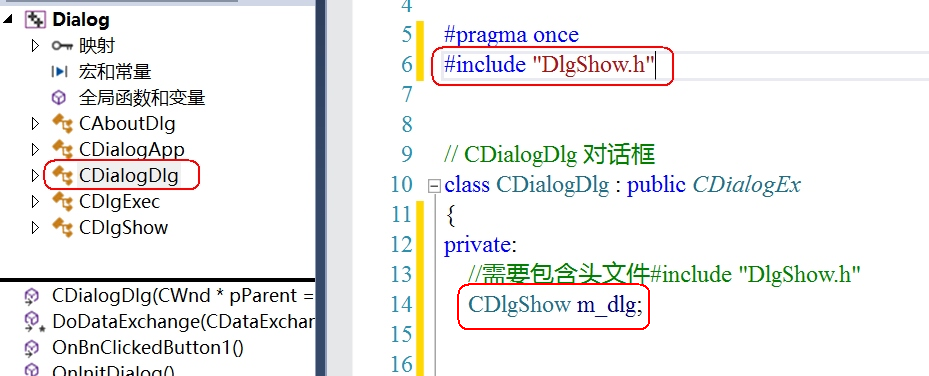
}

2.4 非模态对话框

当非模态对话框显示时，运行转而执行程序中的其他任务，而不用关闭这个对话框。

图形界面操作过程和模态对话框一样，只是，非模态对话框实现方式不一样，先创建(CDialog::Create)一次，然后再显示(CWnd::ShowWindow)。

1. 主对话框.h类中声明对话框对象：



1. 创建对话框放在主对话框类的构造函数或OnCreate()函数，目的只创建一次对话框：

//主对话框构造函数

CDialogDlg::CDialogDlg(*CWnd*\* pParent /\*=NULL\*/)

: *CDialogEx*(CDialogDlg::IDD, pParent)

{

m\_hIcon = *AfxGetApp*()->*LoadIcon*(IDR\_MAINFRAME);

m\_dlg.*Create*(IDD\_DIALOG\_SHOW); //IDD\_DIALOG\_SHOW为对话框ID

}

1. 按钮处理函数显示对话框：

//启动非模态对话框按钮

void CDialogDlg::OnBnClickedButton2()

{

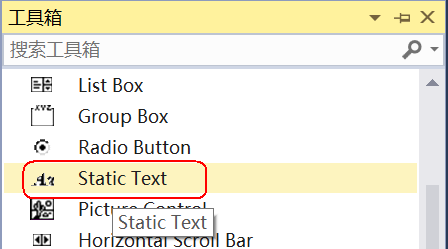
// TODO: 在此添加控件通知处理程序代码

m\_dlg.*ShowWindow*(*SW\_SHOWNORMAL*); //显示非模态对话框

}

3. 常用控件

3.1 静态文本框CStatic



静态文本框是最简单的控件，它主要用来显示文本信息，不能接受用户输入，一般不需要连接变量，也不需要处理消息。

静态文本框的重要属性有：

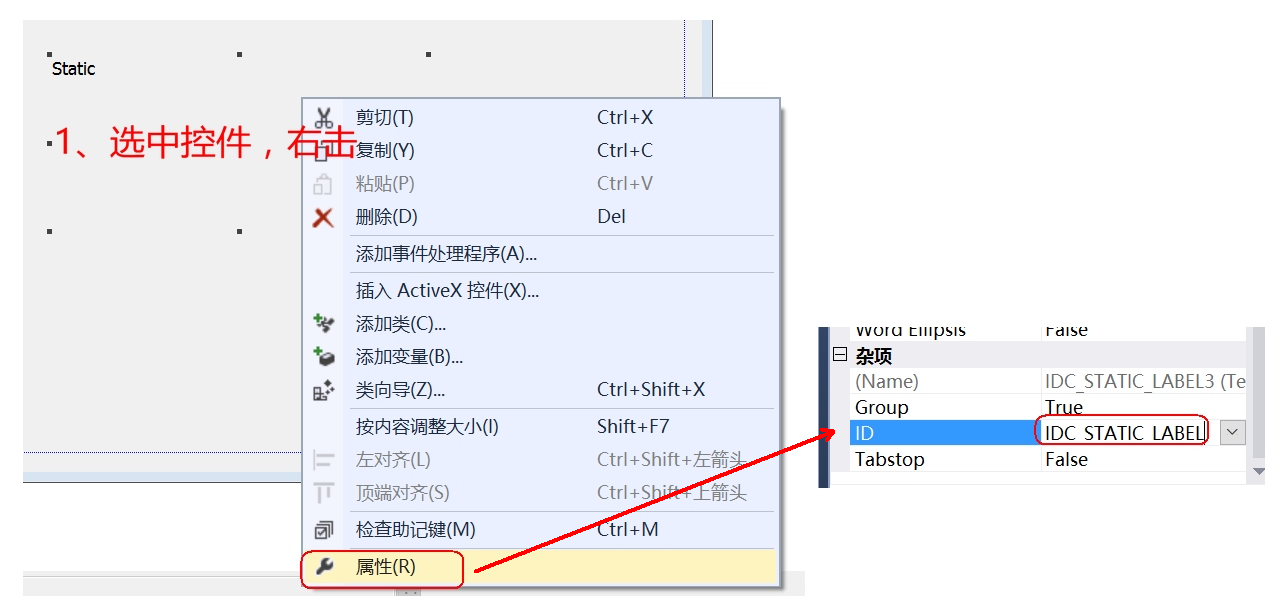
* ID：所有静态文本框的缺省ID都是IDC\_STATIC，静态ID，不响应任何消息（事件）
* Caption：修改显示的内容

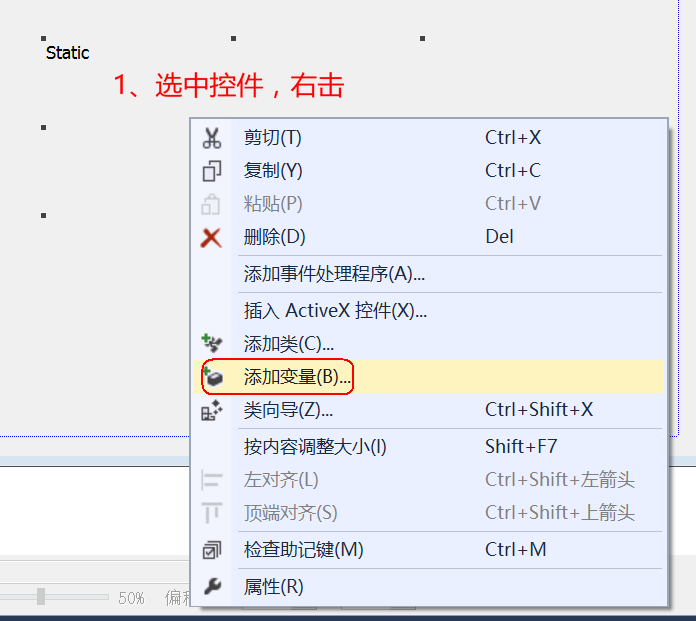
常用接口：

| **接口** | **功能** |
| --- | --- |
| CWnd::SetWindowText | 设置控件内容 |
| CWnd::GetWindowText | 获取控件内容 |
| CStatic::SetBitmap | 设置位图(后缀为bmp的图片) |

**关联控件变量：**

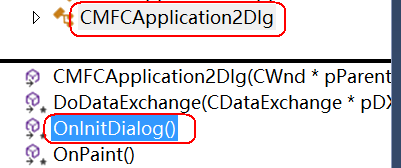
由于XXX\_STATIC静态ID是不能关联变量，故需把ID修改后，再关联变量：







在主对话框类OnInitDialog()中，完成相应接口测试：



//设置静态控件内容为Tom

m\_label.*SetWindowText*(*TEXT*("Tom"));

//获取静态控件的内容

*CString* str;

m\_label.*GetWindowText*(str);

*MessageBox*(str);

//设置静态控件窗口风格为位图居中显示

m\_label.*ModifyStyle*(0xf, *SS\_BITMAP* | *SS\_CENTERIMAGE*);

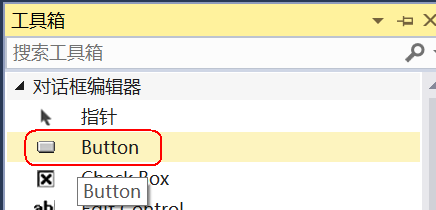
//通过路径获取bitmap句柄

#define HBMP(filepath,*width*,*height*) (*HBITMAP*)*LoadImage*(*AfxGetInstanceHandle*(),filepath,*IMAGE\_BITMAP*,*width*,*height*,*LR\_LOADFROMFILE*|*LR\_CREATEDIBSECTION*)

//静态控件设置bitmap

m\_label.*SetBitmap*(HBMP(*TEXT*("./1.bmp"), 300, 250));

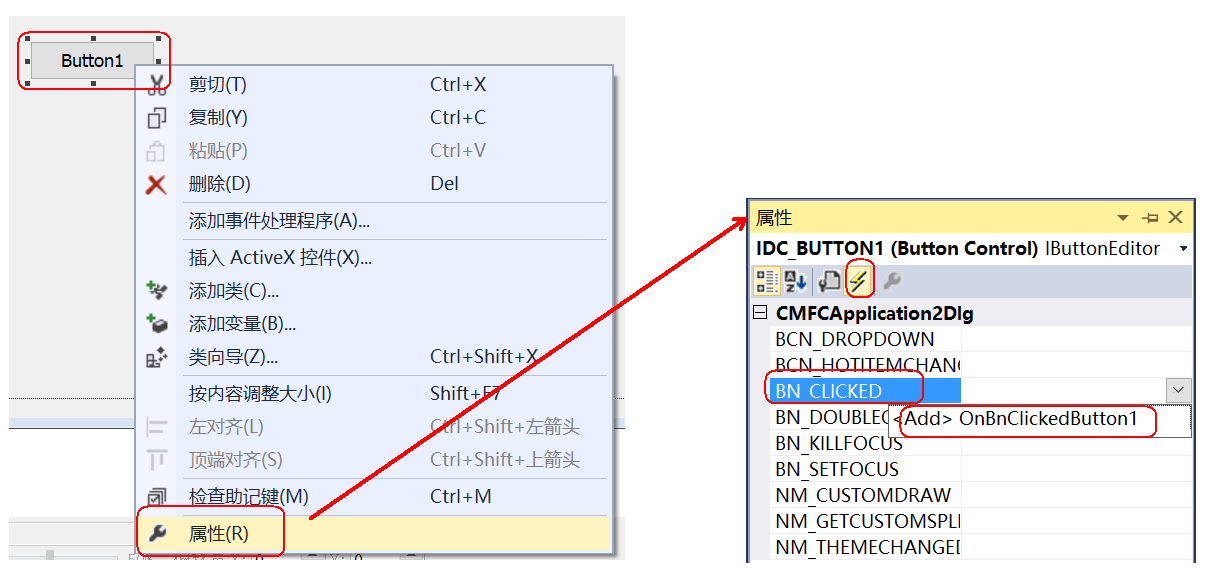
3.2 普通按钮 CButton



按钮是最常见的、应用最广泛的一种控件。在程序执行期间，当单击某个按钮后就会执行相应的消息处理函数。

按钮的主要属性是Caption，来设置在按钮上显示的文本。

命令按钮处理的最多的消息是：BN\_CLICKED，双击按钮即可跳转到处理函数。或者，通过按钮属性 -> 控制事件 -> 选择所需事件，添加处理函数：



//按钮BN\_CLICKED事件处理函数

void CMFCApplication2Dlg::OnBnClickedButton1()

{

// TODO: 在此添加控件通知处理程序代码

}

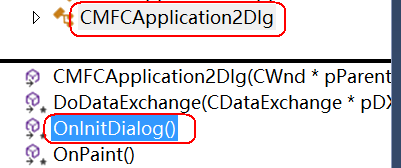
常用接口：

| **接口** | **功能** |
| --- | --- |
| CWnd::SetWindowText | 设置控件内容 |
| CWnd::GetWindowText | 获取控件内容 |
| CWnd::EnableWindow | 设置控件是否变灰 |

关联控件变量：



在主对话框类OnInitDialog()中，完成相应接口测试：



//获取按钮的内容

*CString* str;

m\_button.*GetWindowText*(str);

*MessageBox*(str);

//设置按钮内容

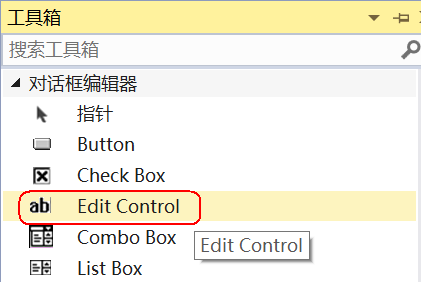
m\_button.*SetWindowText*(*TEXT*("^\_^"));

//设置按钮状态为灰色

m\_button.*EnableWindow*(*FALSE*);

m\_button.*EnableWindow*(*TRUE*);

3.3 编辑框CEdit



常用属性设置：

| **属性** | **含义** |
| --- | --- |
| Number | True只能输入数字 |
| Password | True密码模式 |
| Want return | True接收回车键，自动换行，只有在多行模式下，才能换行 |
| Multiline | True多行模式 |
| Auto VScroll | True 当垂直方向字符太多，自动出现滚动条，同时设置Vertical Scroll才有效 |
| Vertical Scroll | True当垂直方向字符太多，自动出现滚动条，和Auto VScroll配合使用 |
| Horizontal Scroll | True当垂直方向字符太多，自动出现滚动条，和Auto HScroll配合使用 |
| Read Only | True 只读 |

常用接口：

| **接口** | **功能** |
| --- | --- |
| CWnd::SetWindowText | 设置控件内容 |
| CWnd::GetWindowText | 获取控件内容 |

3.3.1 关联控件变量



在主对话框类OnInitDialog()中，完成相应接口测试：

//设置按钮内容

m\_edit.*SetWindowText*(*TEXT*("this is a test"));

//获取按钮的内容

*CString* str;

m\_edit.*GetWindowText*(str);

*MessageBox*(str);

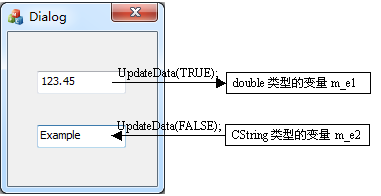
3.3.2 关联基本类型变量



若一个编辑框连接了一个Value类别的变量，则该变量就表示这个编辑框，编辑框中显示的内容就是变量的值。

但是，改变了编辑框的内容并不会自动更新对应的变量的值，同样，改变了变量的值也不会自动刷新编辑框的内容。若要保持一致，需要使用UpdateData()函数更新：

* 若编辑框的内容改变了，则应使用语句UpdateData(TRUE) 获取对话框数据
* 若变量的值改变了，则应使用语句UpdateData(FALSE) 初始化对话框控件



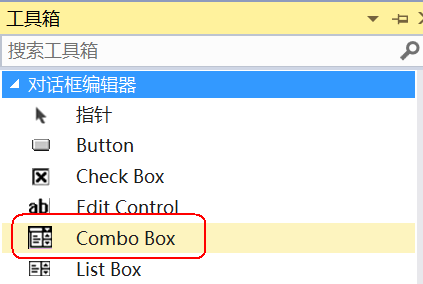
在主对话框类OnInitDialog()中，完成相应代码测试：

m\_e1 = 10.11;

*UpdateData*(*FALSE*); //FALSE说明把m\_e1的值更新到对应的控件中

*UpdateData*(*TRUE*); //TRUE说明把控件的值更新到m\_e1变量中

3.4 组合框(下拉框) CComboBox



常用属性设置：

| **属性** | **含义** |
| --- | --- |
| data | 设置内容，不同内容间用英文的分号“;”分隔 |
| type | 显示风格 |
| Sort | True 内容自动排序 |

常用接口：

| **接口** | **功能** |
| --- | --- |
| CComboBox::AddString | 组合框添加一个字符串 |
| CComboBox::SetCurSel | 设置当前选择项(当前显示第几项)，下标从0开始 |
| CComboBox::GetCurSel | 获取组合框中当前选中项的下标 |
| CComboBox::GetLBText | 获取指定位置的内容 |
| CComboBox::DeleteString | 删除指定位置的字符串 |
| CComboBox::InsertString | 在指定位置插入字符串 |

关联控件变量后，测试接口：



//添加字符串内容

m\_combo.*AddString*(*TEXT*("可乐"));

m\_combo.*AddString*(*TEXT*("雪碧"));

m\_combo.*SetCurSel*(1);//显示显示第1项

//获取组合框中当前选中项的下标

int index = m\_combo.*GetCurSel*();

*CString* str;

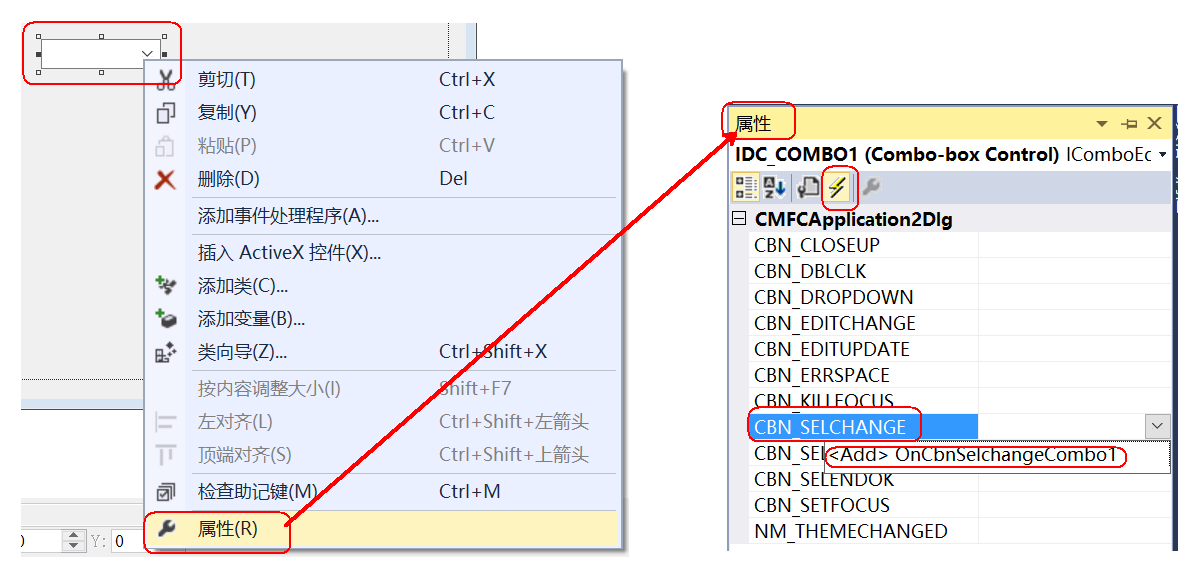
m\_combo.*GetLBText*(index, str); //获取指定下标的内容

*MessageBox*(str);

m\_combo.*DeleteString*(0); //删除第0项字符串

m\_combo.*InsertString*(0, *\_T*("hello")); //在第0位置插入“hello”

组合框常用的事件为：CBN\_SELCHANGE，当选择组合框某一项时，自动触发此事件。



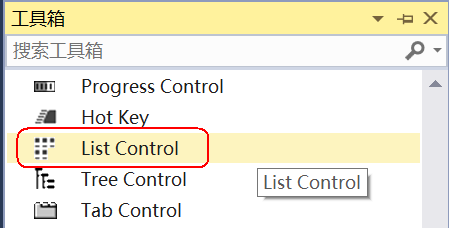
void CMFCApplication2Dlg::OnCbnSelchangeCombo1()

{

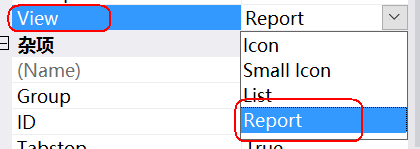
// TODO: 在此添加控件通知处理程序代码

}

3.5 列表控件 CListCtrl



常用属性设置：view -> Report(报表方式)



常用接口：

| **接口** | **功能** |
| --- | --- |
| CListCtrl::SetExtendedStyle | 设置列表风格 |
| CListCtrl::GetExtendedStyle | 获取列表风格 |
| CListCtrl::InsertColumn | 插入某列内容，主要用于设置标题 |
| CListCtrl::InsertItem | 在某行插入新项内容 |
| CListCtrl::SetItemText | 设置某行某列的子项内容 |
| CListCtrl::GetItemText | 获取某行某列的内容 |

关联控件变量后，测试接口：



//设置风格样式

//LVS\_EX\_GRIDLINES 网格

//LVS\_EX\_FULLROWSELECT 选中整行

m\_list.*SetExtendedStyle*(m\_list.*GetExtendedStyle*()

| *LVS\_EX\_GRIDLINES* | *LVS\_EX\_FULLROWSELECT*);

//插入标题

*CString* head[] = { *TEXT*("姓名"), *TEXT*("年龄"), *TEXT*("性别") };

//插入列

m\_list.*InsertColumn*(0, head[0], *LVCFMT\_LEFT*, 100);

m\_list.*InsertColumn*(1, head[1], *LVCFMT\_LEFT*, 100);

m\_list.*InsertColumn*(2, head[2], *LVCFMT\_LEFT*, 100);

//插入正文内容，先确定行，再确定列

for (int i = 0; i < 10; i++)

{

*CString* str;

str.*Format*(*TEXT*("张三\_%d"), i );

//确定行

m\_list.*InsertItem*(i, str);

//设置列

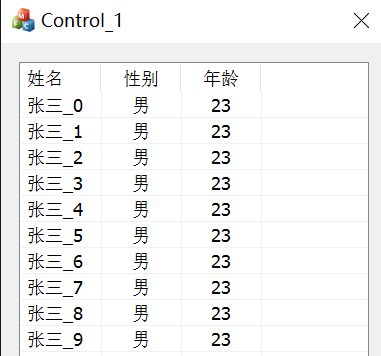
int j = 0;

m\_list.*SetItemText*(i, ++j, *TEXT*("男"));

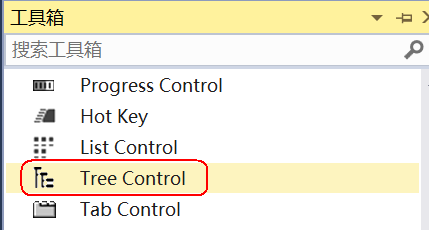
m\_list.*SetItemText*(i, ++j, *TEXT*("23"));

}

程序效果图：



3.6 树控件 CTreeCtrl



常用属性设置：

| **属性** | **含义** |
| --- | --- |
| has buttons | True 有展开按钮 |
| has lines | True 有展开线 |
| lines at root | True 有根节点 |

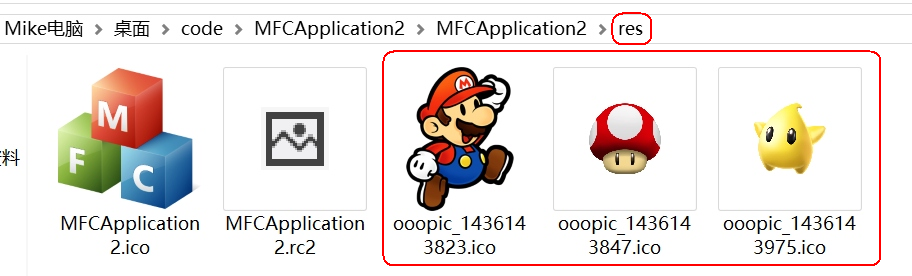
常用接口：

| **接口** | **功能** |
| --- | --- |
| AfxGetApp() | 获取应用程序对象指针 |
| CWinApp::LoadIcon | 加载自定义图标 |
| CImageList::Create | 创建图像列表 |
| CImageList::Add | 图像列表追加图标 |
| CTreeCtrl::SetImageList | 设置图形状态列表 |
| CTreeCtrl::InsertItem | 插入节点 |
| CTreeCtrl::SelectItem | 设置默认选中项 |
| CTreeCtrl::GetSelectedItem | 获取选中项 |
| CTreeCtrl::GetItemText | 获取某项内容 |

1. 关联控件变量



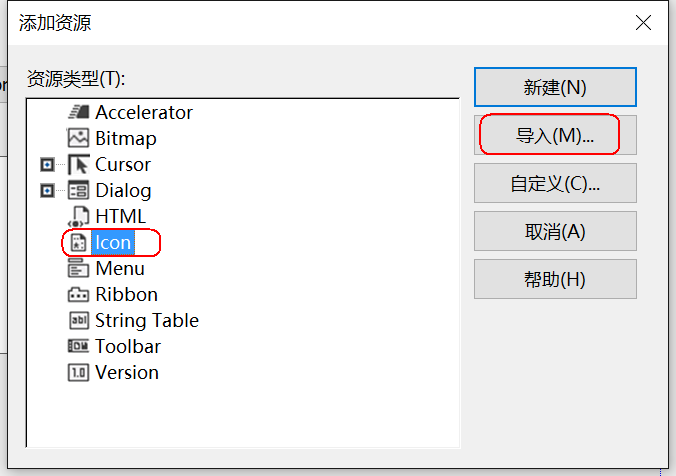
1. 添加图标资源(icon)
2. 把ico资源文件放在项目res文件夹中

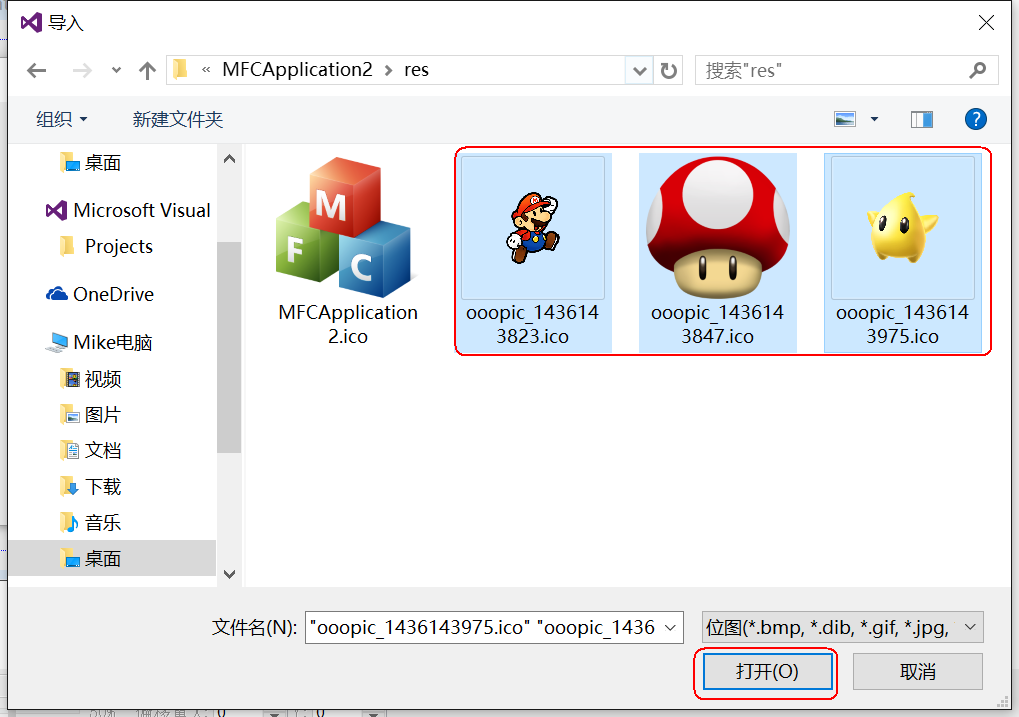


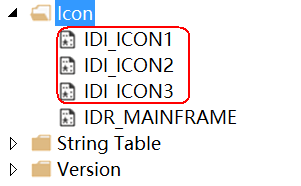
b)资源视图 -> Icon -> 添加资源：



c)导入ico文件







1. 通过代码加载图标

//加载图标

*HICON* icon[3];

icon[0] = *AfxGetApp*()->*LoadIconW*(IDI\_ICON1);

icon[1] = *AfxGetApp*()->*LoadIconW*(IDI\_ICON2);

icon[2] = *AfxGetApp*()->*LoadIconW*(IDI\_ICON3);

4)创建图像列表

a) .h 文件类中定义图形列表（CImageList）对象

*CImageList* m\_imageList; //图像列表

b) OnInitDialog()函数中完成图像列表的创建、图标的追加

//图像列表，程序完毕不能释放， 创建

//30, 30: 图片的宽度和高度

//ILC\_COLOR32：样式

// 3, 3： 有多少图片写多少

m\_imageList.*Create*(30, 30, *ILC\_COLOR32*, 3, 3);

//给图像列表添加图片

for (int i = 0; i < 3; i++)

{

//图片列表加载图标

m\_imageList.*Add*(icon[i]);

}

1. 树控件的相应操作

//树控件设置图片列表

m\_treeCtrl.*SetImageList*(&m\_imageList, *TVSIL\_NORMAL*);

//给树创建节点

//根节点，父节点，子节点

*HTREEITEM* root = m\_treeCtrl.*InsertItem*(*TEXT*("中国"), 0, 0, *NULL*);

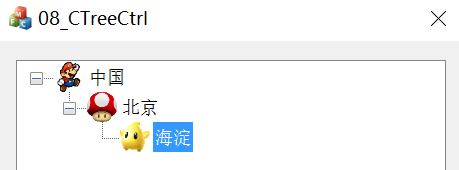
*HTREEITEM* fathter = m\_treeCtrl.*InsertItem*(*TEXT*("北京"), 1, 1, root);

*HTREEITEM* son = m\_treeCtrl.*InsertItem*(*TEXT*("海淀"), 2, 2, fathter);

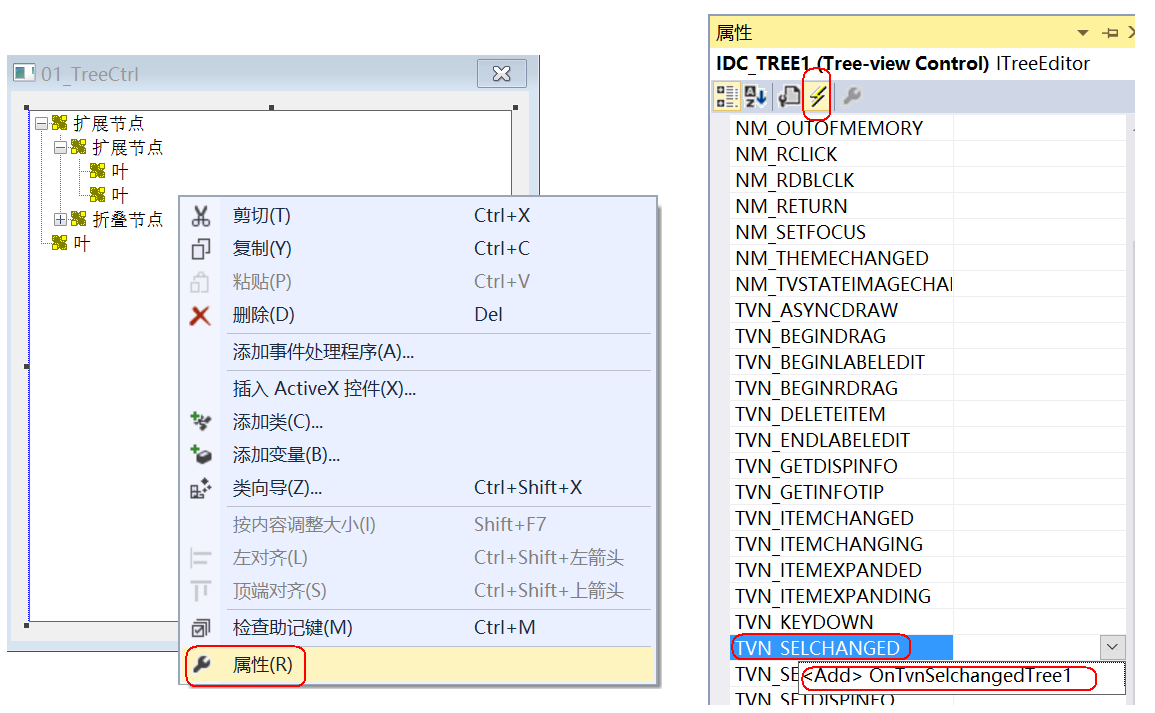
//设置某个节点被选中

m\_treeCtrl.*SelectItem*(fathter);

程序效果图：



树控件常用事件为：TVN\_SELCHANGED，当选择某个节点时，自动触发此事件。



void CMy01\_TreeCtrlDlg::OnTvnSelchangedTree1(*NMHDR* \*pNMHDR, *LRESULT* \*pResult)

{

*LPNMTREEVIEW* pNMTreeView = reinterpret\_cast<*LPNMTREEVIEW*>(pNMHDR);

// TODO: 在此添加控件通知处理程序代码

\*pResult = 0;

*HTREEITEM* selItem;

//获得选择项

selItem = m\_treeCtrl.*GetSelectedItem*();

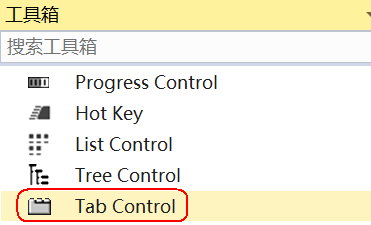
//获取选中的内容

*CString* cs = m\_treeCtrl.*GetItemText*(selItem);

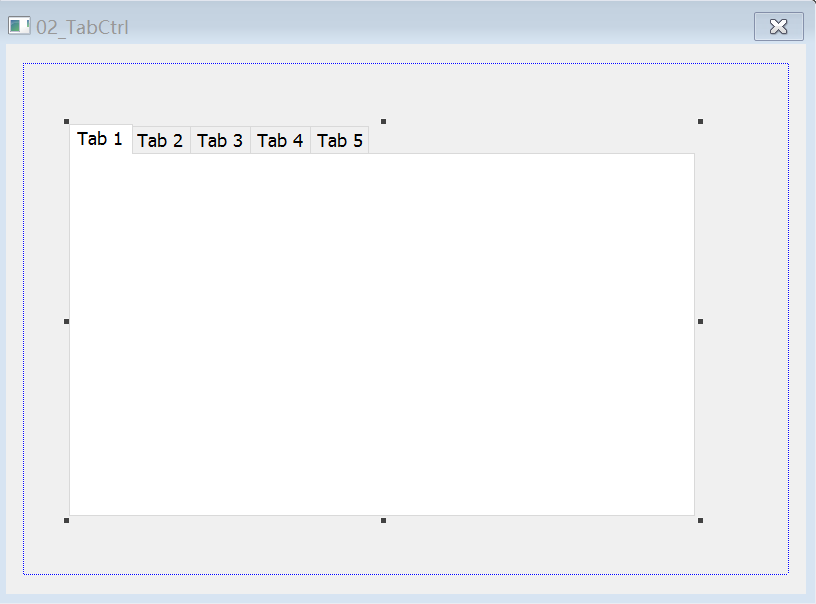
*MessageBox*(cs);

}

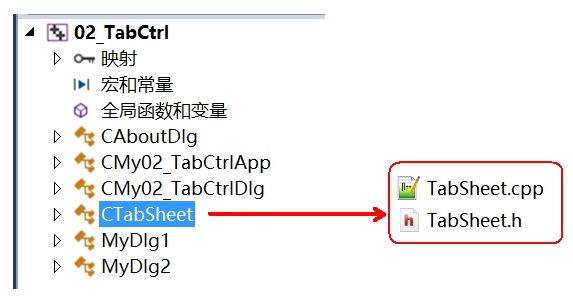
3.7 标签控件 CTabCtrl



1) 在ui工具箱拖放 Tab Control



1. 把 TabSheet.h和TabSheet.cpp 放在项目文件同级目录，并且添加到工程目录中



1. 给ui上 Tab Control 关联Control类型（CTabSheet）



1. 添加对话框

a) 资源视图 -> Dialog -> 右击 -> 插入 Dialog

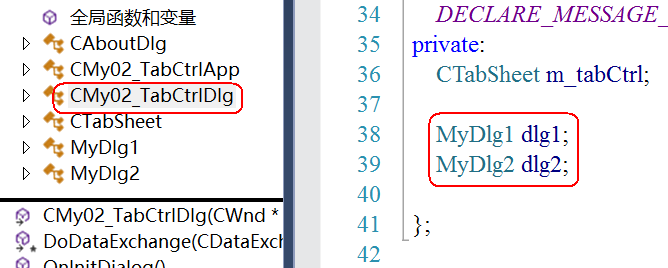
b) 设置相应属性：

Style -> Child (子窗口)

Border -> None (无边框)

c) 自定义类：点击对话框模板 -> 右击 -> 添加类(MyDlg1、MyDlg2)

d) 主对话框类中, 定义自定义类对象，需要相应头文件



e) 主对话框类中 OnInitDialog() 做初始化工作

//给tab控件添加对话框

//IDD\_DIALOG1为dlg1资源ID

m\_tabCtrl.AddPage(*TEXT*("系统管理"), &dlg1, IDD\_DIALOG1);

//IDD\_DIALOG1为dlg2资源ID

m\_tabCtrl.AddPage(*TEXT*("系统设置"), &dlg2, IDD\_DIALOG2);

//显示tab控件

m\_tabCtrl.Show();

程序效果图：



4. 综合案例：销售信息管理系统

