路由事件的演变史

终于敢写路由事件（routed event）了，原因很简单。因为我在研究一个东西，喜欢研究一个东西的演变，因此在研究路由事件的时候，我不得不先从事件（event）开始。由于自己有过 Win32 的背景，于是很自然地联想到消息（Message）。

在这一章，我将带着各位从 Win32，MFC，WinForm 和 WPF 这样一路游览过来，路上无疑会看到函数指针（Function Pointer）,回调函数（Callback Function），成员函数指针（Member Function Pointer）,委托（Delegate），事件（Event）和路由事件（Routed Event）,真的很有趣……

1. 程序模块如何沟通

人和人需要沟通。程序世界里模块和模块之间也需要沟通。人的沟通方式多种多样，有高山流水神交型，也有在一块吃喝玩乐，搞搞娱乐节目。但是模块和模块之间沟通归纳起来只有两种。我们把提供功能的模块成之为服务模块 S，把享受功能的模块称之为客户模块 C。那么一种沟通方式是服务模块 S 暴露一些接口，供客户模块 C 调用。

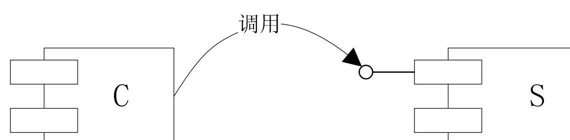

图 1-1 第一种沟通方式

第一种沟通方式是可以解决绝大部分问题。但是还是有一部分问题还是不能解决的。主要是服务模块 S 服务完之后有可能需要通知给客户模块 C，C 可能需要针对这些通知做出响应。这样说，可能还是有些抽象。我不妨举个现实中的例子。

比如，我去火车站买票，告诉售票员需要当天郑州到北京的火车票，售票员会告诉我有如下几种车票：

表 1-1 郑州到北京火车票

郑州到北京和谐号	6:54am——11:58am	213 元
郑州到北京的特快车（硬座）	7:30am——14:50pm	94 元
郑州到北京的快车（硬卧）	10:58pm——次日 6:30am	175 元
…….	…….	

如果我有急事，那么需要选择和谐号，如果我想尽可能的便宜可能会选择票价为 94 元的郑州到北京的特快车（硬座），如果我想不耽误白天的工作，则我可能选择睡一晚

上到达等等。总之是根据具体要求的不同我会选择最佳的车票。这个实际问题如何用程序语言描述呢？

按照模块之间的第一种沟通方式，售票员（模块 S）会暴露给我（模块 C）一个接口 GetTicket，具体参数有时间（Time）,起始城市（Start City）和终点城市（End City）。

```
GetTicket(time, start city, end city);
```

但是这样还是不够的，因为从郑州到北京有多趟列车。实际情况是售票员会询问："尊敬的顾客同志，今天从郑州到北京的火车票有和谐号、特快车和快车，请问你要哪一趟列车？"（当然如果遇到比较猛的大妈级售票员也许会说："要哪一趟？啊 不知道，查清楚了再过来买啊，耽误事嘛，下一位。"）

于是这个接口要增加一个参数是表明选择的依据（chooseby），里面的内部实现为：

```
GetTicket(time, start city, end city,chooseby)
{
        // 1 查询从郑州到北京的所有火车
        …….
        // 2 根据不同条件返回合适的车票
        switch（chooseby）
        {
        case "急啊":
              return "郑州到北京和谐号";
        case "一定要便宜":
             return "郑州到北京的特快车（硬座）";
        case "晚上在火车上睡一觉":
             return "郑州到北京的快车（硬卧）";
        default:
              return "想清楚再来买，下一位";
        }
}
```

我的问题解决了，可是排在我后面的老大娘想在北京转车去长春，再后面的中年男子需要赶回去给老婆弄饭……于是 GetTicket 的 switch….case 越来越庞大（难怪售票员阿姨有时候心情不好 也真难为她们了。）于是售票员没有办法了，说："你们给我一个选择的依据吧"。于是所有的人在买票的时候，都需要提供一个 BestChoice 的方法，售票员会帮助你查询到满足时间，起始城市和终点城市的所有车次提供给你，然后由你自己去选择。

```
Ticket BestChoice (Ticket[]);
GetTicket(time, start city, end city, BestChoice)
{
        // 1 查询从郑州到北京的所有火车
        …….
        // 2 查询到的火车次数供客户自己选择
        return BestChoice(…)；
}
```

这种方式即是模块之间的第二种通讯方式，享受服务的客户（C）都必须提供一个规定好的接口供服务者调用。

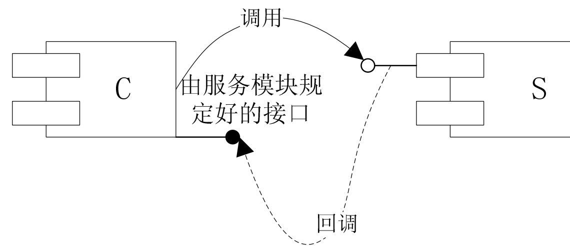

图 1-2 第二种沟通方式

从第一个 Windows 程序开始，我们普遍采用第二种方式。系统总是找了一个地方，然后程序员们就开始写，如果点击了按钮怎么办?如果鼠标移动了怎么办?等等。

在 Win32 里，这个地方是窗口过程函数，系统借助回调函数的机制来实现对其调用。

在 MFC 里，这个地方是每个类的消息响应（Message Handlers）函数，MFC 借助类成员员函数指针来实现其调用；

在 WinForm 里，这个地方是事件响应函数（event hanlders），.Net 通过委托来实现其调用；

在 WPF 里，这个地方仍然是事件响应函数，不同的是多出来一种新的事件，路由事件。

下面几节是一次 Windows 的时间旅行......

2. Win32 和回调函数

从DOS时代过渡到Windows时代是一个重大的转折点，和过去DOS最大的不同是Windows是“以消息为基础，以事件驱动之”（message based, event driven）。关于Windows机制，候大师曾经画过一副非常经典的图[1]，如下所示：

[1] 侯捷《深入浅出MFC第二版》是一本关于MFC非常经典的书，不过这幅图还是有些细节上的问题，可以参见我的一篇文章《永远的窗口》。

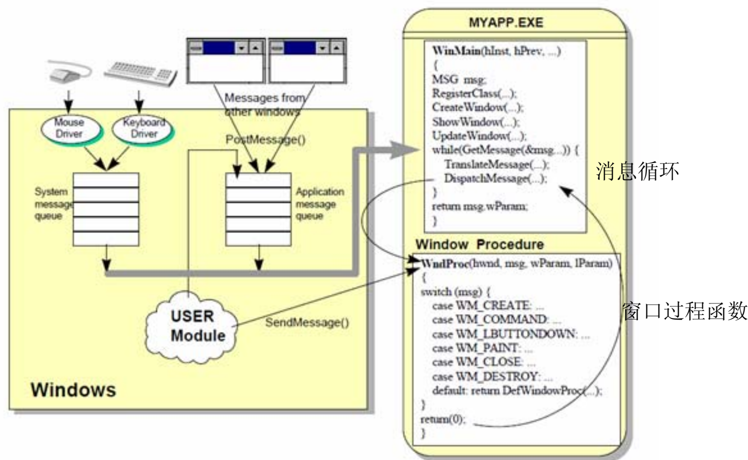

图 2-1 windows 程序与操作系统关系（侯捷《深入浅出 MFC 第二版》）

如果有过 Win32 编程背景的读者，相信这一幅图就是你心中想的确未必能画出来的一幅图。整个系统有一个系统消息队列，应用程序有一个对应的应用程序消息队列，用户移动鼠标，点击键盘，都会通过驱动把这种外部响应转换为相应的消息，还有用户点击窗口上的菜单等，也会触发消息。这些消息都会放在消息队列中，然后消息循环会一个一个取出来然后给相应的窗口过程函数进行处理。

> 如果希望详细了解 Windows 编程模型，那么可以参见我的一篇文章《永远的窗口》，这里面甚至也指出了侯捷大师这幅图的一点小小瑕疵。当然还有一本经典书是是 Charles Petztold 写的《Windows 程序设计》。

应用程序注册窗口类，创建窗口之后，就进入了消息循环，这个时候窗口所需要做的事情就是两个字——“等待”。操作系统发现需要应用程序做一些事情的时候，会以消息的形式通知应用程序。比如需要重绘了，操作系统会给应用程序发送 WM_PAINT 消息，鼠标左键点击了一下窗口，操作系统会给应用程序发送 WM_LBUTTONDOWN 消息等。至于应用程序是否响应这些消息，怎么响应这些消息，操作系统是不予理睬的。我们前面已经说过操作系统需要安排一个地方，让程序员去做这些事情。于是操作系统提供了一个需要程序员实现的函数接口。这个函数接口就是大名鼎鼎的窗口过程函数（WndProc），它是一个回调函数。

一个回调函数往往是用函数指针实现的。我们可以看到窗口过程函数的声明就是一个函数指针[2]。在注册窗口类的时候，窗口过程函数的地址存储在该函数指针里，之后系统需要调用窗口过程函数，调用这个指针即可。

[2] 函数指针的概念，可以参见任何一本讲解C语言或者C++语言的书，我在这里推荐Stanley B.Lippman《C++Primer第 4 版》

```
// 窗口过程函数
LRESULT CALLBACK WndProc(HWND hwnd, UINT message……);
// 应用程序的入口
int APIENTRY WinMain(HINSTANCE hInstance,……)
{
    // TODO: Place code here.
……
    WNDCLASS wndclass;
    static TCHAR szAppName[] = TEXT("HelloWin");

    // 系统通过一个字符串来唯一标示一个窗口类
    wndclass.lpszClassName = szAppName;
    // 窗口类结构体 lpfnWndProc 成员存储了当前窗口过程函数的地址
    wndclass.lpfnWndProc = WndProc;
    …….

    // 注册窗口类，系统已经将该类型窗口和窗口过程关联起来
    if(!RegisterClass(&wndclass))
    {
        return 0;
    }
……..
}

// winuser.h
typedef struct tagWNDCLASSA {
    UINT            style;
    WNDPROC         lpfnWndProc;
   ……
    LPCSTR          lpszClassName;
} WNDCLASSA, *PWNDCLASSA, NEAR *NPWNDCLASSA, FAR *LPWNDCLASSA;

typedef LRESULT (CALLBACK* WNDPROC)(HWND, UINT, WPARAM, LPARAM);
```

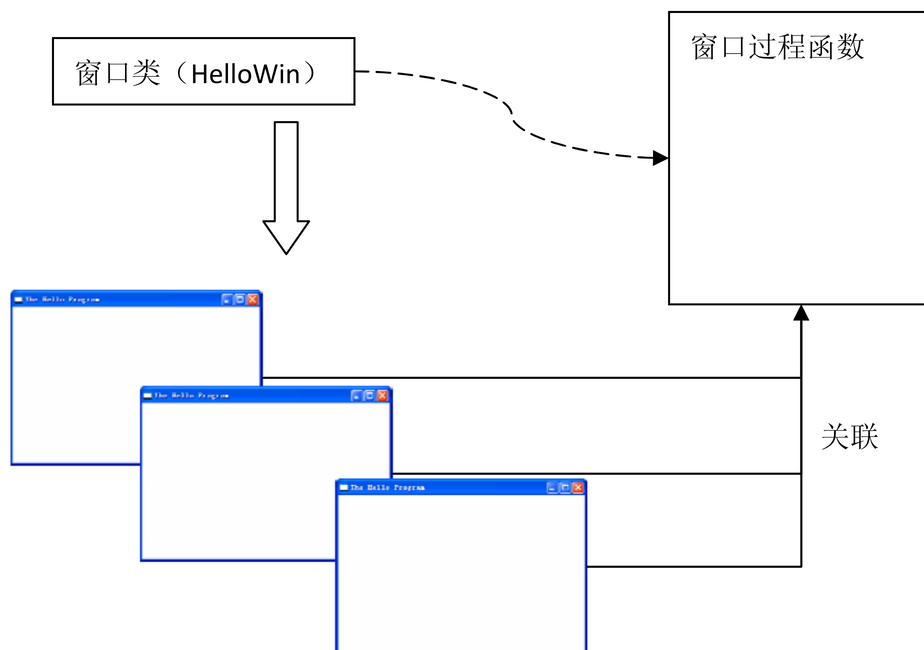

图 2-2

虽然没有见过微软操作系统的源码，但其内部一定会有如下代码来调用窗口过程函数：

```
.. lpfnWndProc(hwnd,……);
```

3. MFC 和类的成员函数指针

在 MFC 里，一切变得复杂了。传统的 WinMain 不见了，窗口过程函数不见了。取而代之的是 CWinApp,CDocument 和 CMainFrame 等等。

3.1 在 MFC 里添加消息响应函数

第二节里已经看到，Win32 程序里对消息进行响应，是在窗口过程函数里增加相应的消息响应代码。

```
LRESULT CALLBACK WndProc(HWND hWnd……)
{
    switch (message)
    {
        case WM_COMMAND:
            …….
```

```
            case WM_PAINT:
                 .......
                break;
            case WM_DESTROY:
                PostQuitMessage(0);
                break;
            default:
                return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

消息越多，窗口过程函数的 switch……case 会越来越长。MFC 里已经看不到窗口过程函数，也看不到长长的 switch……case，消息响应都被 MFC 封装到了一个一个类对应的消息响应函数当中。

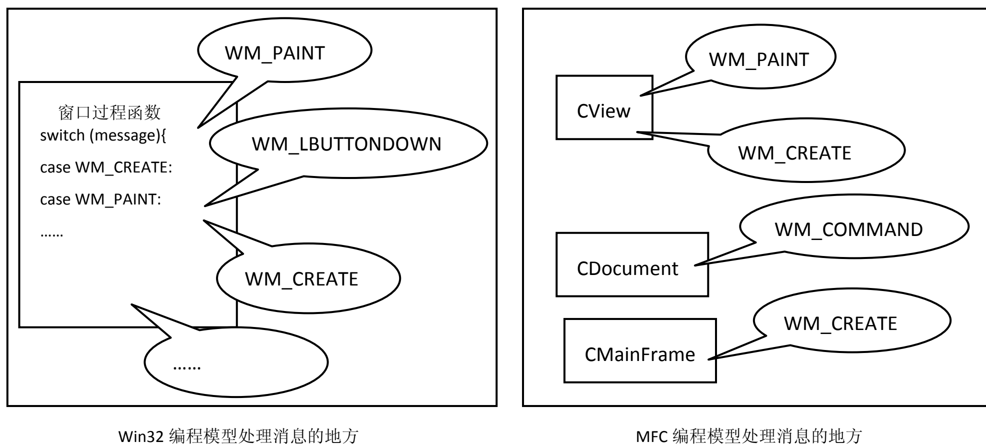

图 3-1 Win32 和 MFC 响应消息的对比

我们以在一个 View 视图添加鼠标左键消息为例。

1. 在 VC6 环境下，新建一个单文档工程名为 HelloWorldMFC；
2. 通过 MFC 类向导为 CHelloWorldMFCView 添加鼠标左键消息；

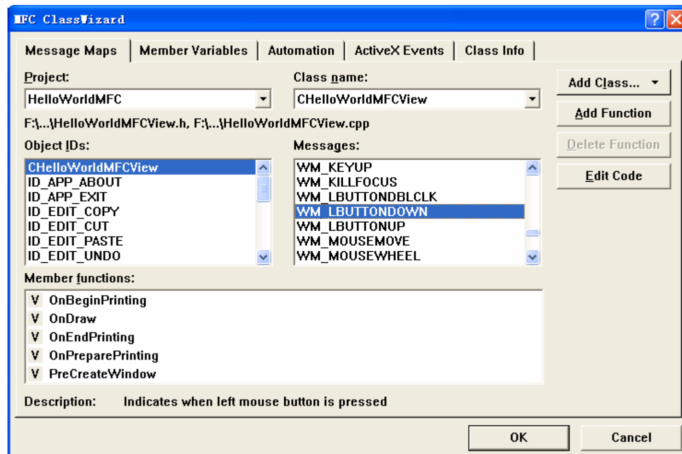

图 3-2 MFC 类向导（从菜单视图-MFC 类向导或者 Ctrl+w 调用）

3. 点击 Edit Code 按钮，VC6 会自动切换到 HelloWorldMFC.cpp 文件 OnLButtonDown 函数，程序员需要在该函数里添加鼠标左键响应的代码。

```
void CHelloWorldMFCView::OnLButtonDown(UINT nFlags, CPoint point)
{
  // TODO: Add your message handler code here and/or call default

  CView::OnLButtonDown(nFlags, point);
 }
```

MFC 为我们隐藏了一些实现细节。实际上需要完整地添加一个消息响应函数，需要在代码里做如下工作，以在 CHelloWorldMFCView 里添加鼠标左键消息响应函数：

1. 在 CHelloWorldMFCView 类中声明 DECLARE_MESSAGE_MAP()宏（这个在工程自动生成之初，系统已经自动添加）；
2. 在 CHelloWorldMFCView 类中声明鼠标左键消息响应函数；

```
protected:
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
```

3. 在 HelloWorldMFCView.cpp 文件中，Begin_Message_Map 和 End_Message_Map 宏中间添加 ON_WM_LBUTTONDOWN 宏；

```
BEGIN_MESSAGE_MAP(CHelloWorldMFCView, CView)
  //{{AFX_MSG_MAP(CHelloWorldMFCView)
  ON_WM_LBUTTONDOWN()
  //}}AFX_MSG_MAP
  // Standard printing commands
  ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
  ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
  ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
   END_MESSAGE_MAP()
```

4. 在 HelloWorldMFCView.cpp 文件中，添加鼠标左键消息响应函数的实现。

```
void CHelloWorldMFCView::OnLButtonDown(UINT nFlags, CPoint point)
{
        // TODO: Add your message handler code here and/or call default

        CView::OnLButtonDown(nFlags, point);
}
```

3.2 MFC 消息映射的秘密

上面的一切都是表象，MFC 通过消息映射的方式，把消息响应映射到一个一个类的消息函数里去了。一个庞大的 WinProc 就被 MFC 分解于无形中。

现在我将带着诸位揭开 MFC 消息映射下的秘密。

MFC消息映射的机制相当复杂，我们在这儿不探求窗口消息是如何利用那些有趣的宏上下游走，上窜下跳的。事实上对这一部分讲述的书籍也相当得多，也相当得精彩。其中侯捷的《深入浅出MFC第二版》和George Shepherd的《深入解析MFC》[3]（MFC Internals: Inside the Microsoft Foundation Architecture）应该算上经典之作。当然如果你自觉得外文功底还不错，我还推荐Paul DiLascia在 95 年 7 月Microsoft Systems Journal上发表的一篇名为" Meandering Through the Maze of MFC Message and Command Routing"文章。

在这个地方先说几个结论性的东西：

结论之一，MFC里的三个宏DECLARE_MESSAGE_MAP, BEGIN_MESSAGE_MAP和END_MESSAGE_MAP，组织成了一个很大的消息映射网，里面每个实体（AFX_MSG_ENTRY）将消息和消息映射函数关联起来，图 3-3展现的是MFC消息映射网的冰山一角。

[3] 这本书是探求MFC源码的经典之作，现在基本上在市面上找不到这样一本书。我也是获得电子版然后打印成书。

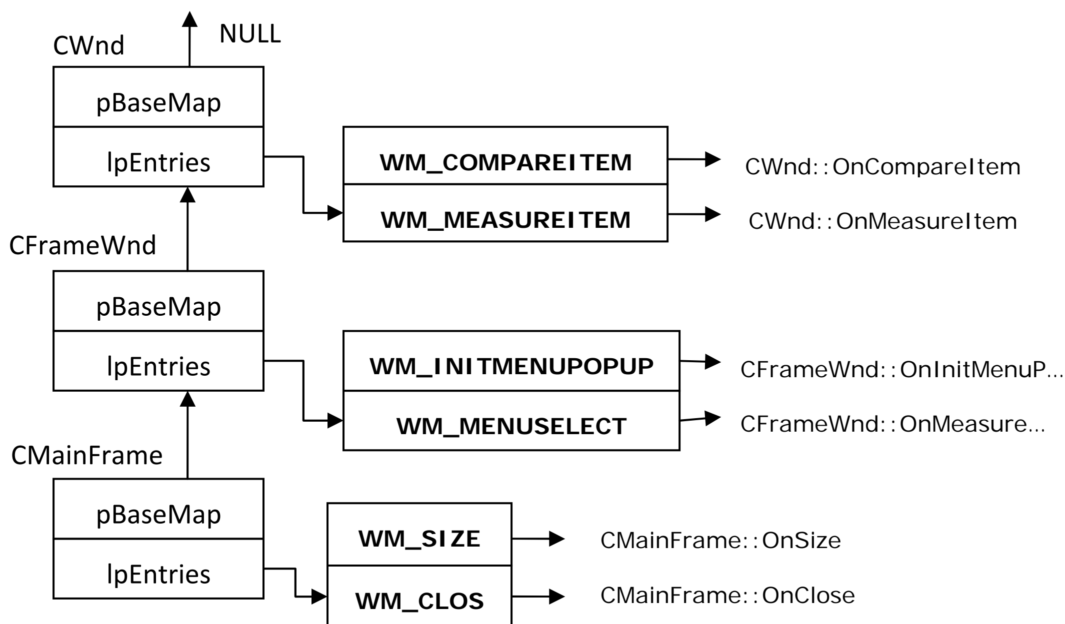

图 3-3 MFC 消息映射网的冰山一角

结论之二，MFC 经过七弯八拐，将窗口过程函数绝大部分事情引入到 CWnd 的 OnWndMsg 函数。

有了这两个结论，问题变成了 MFC 如何将 OnWndMsg 函数正确地分给不同的消息响应函数。常人的想法一般是：

1. 找到该消息对应的消息响应函数；
2. 然后调用该消息响应函数。

MFC 的想法与常人无异。但有一个极为严重的问题会阻拦我们的实现。那就是消息的形式不同造成消息响应的函数声明会有很大的不同，比如下面几个常见的窗口消息。

表 3-1 几个常见的窗口消息

消息	消息参数说明	相应的消息响应函数声明
窗口创建消息：WM_CREATE	wParam 该参数没有用到 lParam 指向 CREATESTRUCT 结构的指针	afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
窗口绘制消息：WM_PAINT	wParam 该参数没有用到 lParam 该参数也没有用到	afx_msg void OnPaint();
鼠标左键消息： WM_LBUTTONDOWN	wParam 表明是否有虚拟键按下，比如 Ctrl 键或者 Shift 键是否按下 lParam 低字节表示的是 x 坐标，高字节表示的是 y 坐标	afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
……	……	……

除此之外，消息响应函数还属于不同类的成员函数，比如CWnd有OnCreate成员函数，CView也有OnCreate函数等等[4]。

于是常人在这儿止步了，但是 MFC 的设计者比常人高明之处，在于他们找到了切割消息响应的刀——类成员函数指针。

现在让我们把所有的焦点都集中在类成员函数指针上面。

3.3 MFC 里的类成员函数指针

类成员函数指针可以说是C++语言里极为生涩的一块。很多熟悉C++语言的程序员也会感到陌生。甚至有的时候它不同于传统的指针，在 32 位操作系统上也不是一个 4 字节的地址[5]（Don Clugston, 2004）。如果想要掌握类成员函数指针的用法，可以参见Lippman的C++Primer**第 4 版** 18.3 节，注意一定是第 4 版，前 3 版都没有涉及到类成员函数指针问题。如果想要有更深入的理解，则需要参阅Lippman另外一本非常高阶的技术书籍Inside the C++ Object Model（中文名为深度探索C++对象模型）3.6 和 4.4 节。当然Don Clugston也对类成员函数指针作了相当深入的剖析（Don Clugston, 2004）。我强烈推荐希望一探究竟的程序员看看他写的这篇文章。

回到 MFC，前面我们说过 MFC 的三个宏编织了一张巨大的消息映射网。而这里面最小的单元是 AFX_MSGMAP_ENTRY，它是一个结构体，目前我们需要关注的是这里面两个成员变量，一个是 nMessage,它表示的就是消息类型，比如 WM_PAINT 等。而 pfn 表示的是一个类成员的函数指针，在这里消息和消息响应函数连接了起来。

```
struct AFX_MSGMAP_ENTRY
{
	UINT nMessage;
	UINT nCode;
	UINT nID;
	UINT nLastID;
	UINT nSig;
	AFX_PMSG pfn;
};
```

请注意 AFX_PMSG 类型，这个类型居然可以有“海纳百川”的气量，可以容纳不同参数，不同类型，甚至是属于不同类的消息响应函数。这个类型真是有无限的诱惑力，让人想一探究竟。继续考察源码。

我们在 AFXWIN.h 里找到 AFX_PMSG 的定义。

---

[4] 有很多人会误解消息响应函数是虚函数，事实上他们不是。

[5] 不过在 32 位操作系统上，MFC里指向类的消息响应函数指针都是 4 字节。

```
typedef void (AFX_MSG_CALL CCmdTarget::*AFX_PMSG)(void);
```

结果让我们失望，又心生疑惑。因为这个类成员函数指针只是一个普通的不能再普通的类成员函数指针，实际上这种指针只能指向如下声明的一类函数。

```
	void CCmdTarget::fun(void);
```

换句话说，AFX_PMSG 成员指针只能指向 CCmdTarget 类的无参数，无返回类型的函数。的确 MFC 里具备消息响应功能的类都必须继承于 CCmdTarget 类，但是各种不同的类型消息响应函数的地址又如何能够存储到这种类型当中呢？

难道 MFC 冥冥之中有神助？继续从源码中探求……

仍然以鼠标左键消息为例，将一个鼠标左键消息映射的最小单元添加到那张巨大的消息映射网里，需要 ON_WM_LBUTTONDOWN 宏。

在 AFX_MSG_.h，可以清楚地查看到该宏的定义。

```
#define ON_WM_LBUTTONDOWN() \
	{ WM_LBUTTONDOWN, 0, 0, 0, AfxSig_vwp, \
		(AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(UINT, CPoint))&OnLButtonDown },
```

在这里你看到了可怕的类型转换动作，足足三次。一个 OnLButtonDown 函数首先转换成 void (AFX_MSG_CALL CWnd::*)(UINT, CPoint)成员函数指针类型，然后再转换成 AFX_PMSGW 类型，再转换成 AFX_PMSG 类型。

AFX_PMSGW 在 AFXWIN.h 定义如下：

```
	typedef void (AFX_MSG_CALL CWnd::*AFX_PMSGW)(void);
```

这种可怕的转换，我们将在下一节讨论。至少我们知道通过这样的三次转换，使得不同类型的消息响应函数地址都可以存储到 AFX_PMSG 类型。实现了消息响应函数存储的“大一统”。

现在大象已经装到了冰箱里，如何把大象完好无损地拿出来？继续探索源码……

下表是 CWnd::OnWndMsg 的部分代码，为了清楚明了，我作了大量的简化。

```
BOOL CWnd::OnWndMsg(UINT message, WPARAM wParam, LPARAM lParam, LRESULT* pResult)
{
	……
	// 获得消息编织的网
	const AFX_MSGMAP* pMessageMap; pMessageMap = GetMessageMap();
	const AFX_MSGMAP_ENTRY* lpEntry;

	// 最终找到现在需要处理的 AFX_MSGMAP_ENTRY 存储在 lpEntry.
```

```
	……
	// 开始交给相应的消息响应函数
LDispatch:
	……
	// MFC 最关键的一步偷天换日的动作
	union MessageMapFunctions mmf;
	mmf.pfn = lpEntry->pfn;
	int nSig;
	nSig = lpEntry->nSig;
	…….
	// 开始根据 nSig，转换成不同的类成员函数指针
	switch (nSig)
	{
	…….
	default:
		ASSERT(FALSE);
		break;
	// 分发给相应的消息响应函数
	case AfxSig_vD:   // OnPaint 或者 OnIconEraseBkgnd
		(this->*mmf.pfn_vD)(CDC::FromHandle((HDC)wParam));
		break;
	……
	}
……
}
```

MFC 偷天换日的最精彩的一步就在于 MessageMapFunctions 这个联合体。在 AFXMSG_.H 定义如下：

```
union MessageMapFunctions
{
	AFX_PMSG pfn;     // generic member function pointer

	// specific type safe variants for WM_COMMAND and WM_NOTIFY messages
	void (AFX_MSG_CALL CCmdTarget::*pfn_COMMAND)();
	BOOL (AFX_MSG_CALL CCmdTarget::*pfn_bCOMMAND)();
	void (AFX_MSG_CALL CCmdTarget::*pfn_COMMAND_RANGE)(UINT);
	BOOL (AFX_MSG_CALL CCmdTarget::*pfn_COMMAND_EX)(UINT);
	……
	void      (AFX_MSG_CALL CWnd::*pfn_vD)(CDC*);
	……
};
```

看出奇妙之处没有？真正的类成员函数指针只有 pfn，但是通过联合可以转换为任何特定的类成员函数指针。而转换的依据则是 AFX_MSGMAP_ENTRY 的另一个成员变量 nsig。它指向了一个枚举变量，如下所示（定义在 AFXMSG_.H）：

```
enum AfxSig
{
	AfxSig_end = 0,        // [marks end of message map]
	AfxSig_bD,           // BOOL (CDC*)
	AfxSig_bb,           // BOOL (BOOL)
	AfxSig_bWww,        // BOOL (CWnd*, UINT, UINT)
	AfxSig_hDWw,        // HBRUSH (CDC*, CWnd*, UINT)
	AfxSig_hDw,         // HBRUSH (CDC*, UINT)
	……

	AfxSig_vD,           // void (CDC*)
	……
	AfxSig_vws,
};
```

侯捷在《深入浅出MFC》一书中对MFC这一做法评价到"相当精致，但是也有点儿可怖，是不是？使用MFC 或许应该像吃蜜饯一样； 蜜饯很好吃，但你最好不要看到蜜饯的生产过程！"

我们可以喘一口气了，貌似所有的神秘都在我们面前揭开了。但事实上我们还差一步。为这一步，我足足花了三天时间才明白。

3.4 类成员函数指针的强制转换

3.4.1 为什么不是一次强制转换，而是三次？

在回到 ON_WM_LBUTTONDOWN 这个宏。

```
#define ON_WM_LBUTTONDOWN() \
	{ WM_LBUTTONDOWN, 0, 0, 0, AfxSig_vwp, \
		(AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(UINT,
	CPoint))&OnLButtonDown },
```

前面已经说过它足足做了三次强制转换，为什么不直接做一次强制转换呢，而是三次？

```
(AFX_PMSG) &OnLButtonDown
```

侯捷说是为了类型安全（type-safe）；

Paul DiLascia 也说是为了 type-safe。

我信了，可是我始终没有明白它就怎么着 type-safe 了？

于是我决定做一些修改。我将 OnLButtonDown 函数多增加了一个参数，，然后编译希望能够立刻报错。

```
// 原函数
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
// 修改过的函数
afx_msg void OnLButtonDown(UINT nFlags, CPoint point,int d);
```

遗憾的是，在编译环境下没有报任何错误。于是我又开始期待它在运行时报错。结果还是没有任何错误。我仍不甘心，将其变成 Release 版希望报错。可是还是没有任何错误。当我失望地在应用程序上随便点了一点，错误竟然来了。于是我开始一头冷汗。

其实在程序世界里，一直行，固然是很好；

一直不行，说实话也还不错；

最怕的是，有时行，有时不行；

最最怕得是，平常行，关键时候不行。

程序员担心的是掌握不住自己的程序，很显然这种属于有时行，有时不行的第三境界。航天上有一句名言“在关键时候，小概率事件则是必然事件”，因此这种情况可以大踏步地迈入最高境界，也就是最最怕境界。因此我一头冷汗（也许各位的实验和我结果不太相同，这正是有时行，有时不行的症状。^_^）。

我熟知的编译器至少有两种，于是我很快将这种改动也应用到 VS 平台（VS2005 或者 VS2008）上。VS 平台果然争了一口气，他在编译的时候就告诉了我，这种改动他不行。

```
error C2440: 'static_cast' : cannot convert from 'int (__thiscall CMainFrame::* )......
```

于是我们不得不重新探究 VS 平台里的源码，发现确实不同。VS 平台里对 ON_WM_LBUTTONDOWN 宏定义如下，为了方便对比，我们将 VC6 对 ON_WM_CREATE 列在其右侧。

VS2005 或者 VS 2008 ON_WM_LBUTTONDOWN 宏定义	VC6 ON_WM_LBUTTONDOWN 宏定义
#define ON_WM_LBUTTONDOWN() \ { WM_LBUTTONDOWN, 0, 0, 0, AfxSig_vwp, \ (AFX_PMSG)(AFX_PMSGW) (static_cast< void (AFX_MSG_CALL CWnd::*)(UINT, CPoint) > (&ThisClass :: OnLButtonDown)) },	#define ON_WM_LBUTTONDOWN() \ { WM_LBUTTONDOWN, 0, 0, 0, AfxSig_vwp, \ (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(UINT, CPoint))&OnLButtonDown },

两者的区别在于加黑处。VS 平台使用的是 static_cast 进行强制转换，而 VC6 没有，使用 static_cast 进行转换，编译器可以检查出函数签名(signature)不匹配的情况。

因此我们有理由认为第一重转换，如果使用 static_cast 可以检查类型安全，但是不使用真的不能。

3.4.2 为什么不是二次强制转换，而是三次？

问题又来了，那么为什么不直接做两次强制转换，而是三次呢？比如写成如下代码：

```
#define ON_WM_LBUTTONDOWN() \
	{ WM_LBUTTONDOWN, 0, 0, 0, AfxSig_vwp, \
		(AFX_PMSG) (void (AFX_MSG_CALL CWnd::*)(UINT, CPoint))&OnLButtonDown },
```

为了探求这个问题，我作了很长时间的思考。于是我写了如下代码：

```
#include "stdafx.h"
class A {
public:
	virtual int Afunc() { return 2; };
};
class B : public   A
{
public:
	virtual int Afunc() { return 8; };
	int Bfunc() { return 3; };
};
class C:   public   A
{
public:
	int Cfunc() { return 4; };
};
// D 类使用了多重继承
class D: public B, public C {
public:
	int Dfunc() { return 5; };
};
// E 是个单一继承类，它只继承于 A,C
class E : public C
{
public:
	int Efunc() {return 6;}
};
int main(int argc, char* argv[])
{
```

```
    typedef int (A::*A_fp)(void);
    typedef int (B::*B_fp)(void);
    typedef int (C::*C_fp)(void);
    typedef int (D::*D_fp)(void);
    typedef int (E::*E_fp)(void);

    A_fp a_fp2 = (A_fp)(E_fp)(&E::Efunc);
    A_fp a_fp = (A_fp)(D_fp)(&D::Dfunc); // error
    return 0;
}
```

E 的成员函数指针可以经过二次强制转换转到 A 的成员函数指针，而 D 则不行。我们不妨看看他们的继承结构。原因正在于他们的继承关系，D 是一个多重继承结构，而 E 是一个单继承结构。E 的成员函数指针即使经过二次强制转换，也能准确无误地转换成 A 的成员函数指针，而 D 确不知道是从 D->B->A，还是从 D->C->A?

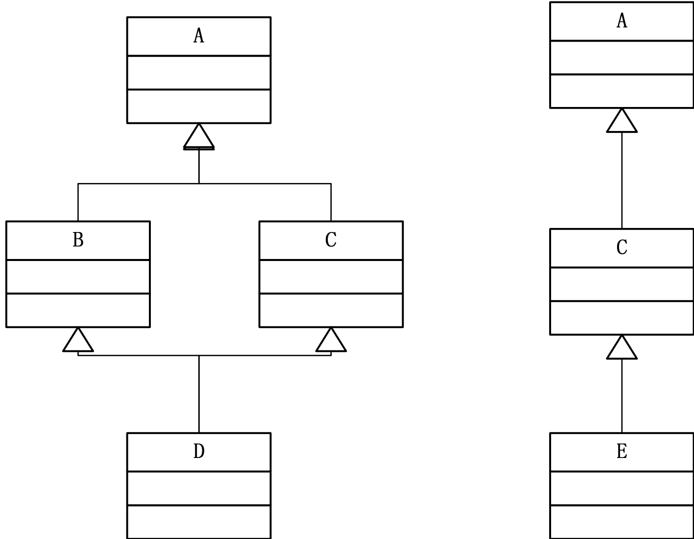

图 3-4 类的继承结构

尽管 MFC 的类层次结构完全是单继承结构，但这样的三次强制转换无疑是一种好习惯的写法。至此所有的秘密昭然若揭。

3.5 关于 MFC 的一点小小感悟

一个女人美的意境是“犹抱琵琶半遮面”。但是程序框架美的意境却全然不同，要么所有的一切都清楚地呈现在眼前，要么一切都封装起来，让你全然不知。前者让程序员有掌控全局的感觉，后者则完全透明，方便调用。换句话说，程序框架美的意境要么是“裸女”要么是 “盛装的美女”。

MFC 显然属于“犹抱琵琶半遮面” 类型，他把很多东西隐藏起来了，但是又留有余地，让人“看不清，断不明”。不可否认，MFC 的影响力是巨大的，他是一个成功的框架，时至今日，仍然有很多企业，科研院所在使用这个框架。但我并不欣赏他。单从 MFC 的消息响应机制来说，很多人都用一个“巨大的迷宫”来形容，由此可见，他带给程序员的困顿。

MFC 使用类成员函数指针，可谓使用到了极致。但是离完美确实相差甚远，至少从以下两点可以窥之。

1. 使用类成员函数指针的确不是类型安全的；
2. MFC 消息响应机制并不“松耦合”。这样一句话太抽象，我们还是打个比方。整个消息好比在一个固定的线路流动。在 MFC 里，程序员可以在需要的地方安插上消息响应函数，截获消息，作出响应，甚至可以从消息的线路上引出一个旁支。但是如果程序员想要完全改变这条线路的形态，则是完全没有可能。

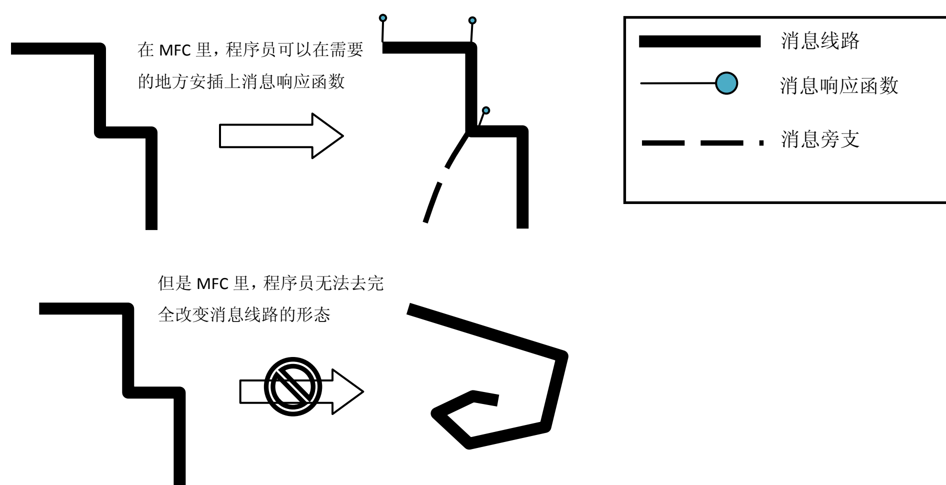

图 3-5 MFC 如何不“松耦合”

4. WinForm 和委托

.Net 平台被描述为微软多年来最重要的新技术一点都不夸张。.Net 的最重要的 Windows 窗体是 WinForm。技术走到了 WinForm，我开始惶恐，因为找不到从 Win32，MFC 继承下来的痕迹，窗口过程自是不必说（完全的看不到），MFC 的那些奇怪的宏业消失殆尽。一个全新的事物总是会让人惶恐的。此外还有一个很重要的原因，那就是我看不见源码。

4.1 如何探索.Net 源码

有过 MFC 编程经验的程序员，知道如果将项目设置 MFC 改成使用静态链接库，则可以调试 MFC 源码。

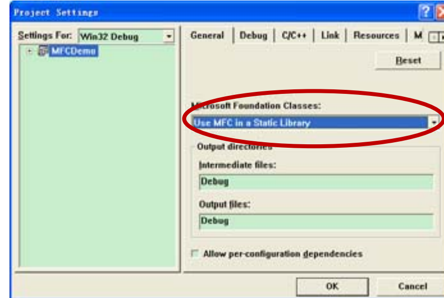

图 4-1 MFC 项目设置

而探索.NetFramework 的源码则没有这么简单，我经历了三个阶段。

第一阶段，我是通过.Net 平台自带的工具 ildasm.exe 来查看 CIL（Common Intermediate Language）源码；CIL 介于 CPU 指令和源代码之间，因此它并没有源代码那么容易理解。

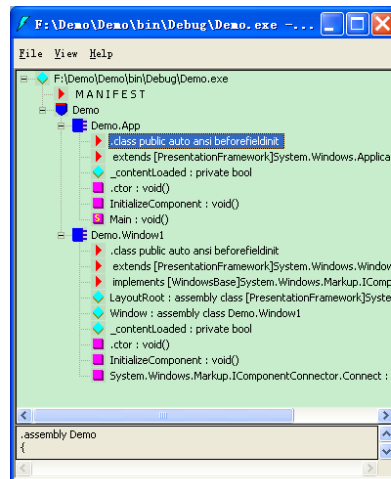

图 4-2 ildasm.exe

第二阶段，Reflector的出现使我立刻弃ildasm，投入到它的怀抱。因为Reflector和ildasm相比，它可以让我们直接看到C#或者Visual Basic的源码。这样的代码当然更受程序员的欢迎。各位可以在http://www.red-gate.com/products/reflector/下载到最新的版本。

但是分析源码，使用这样的工具，我还是会有一些不满足的地方——不能调试。不能调试，意味着我们只能靠猜测，而无法了解程序真正每一步从哪儿走过。

幸运的是，微软在 2008 年年初决定将.NetFramework 代码部分开源。这确实是一个对程序员来说鼓舞人心的事情。这样我顺理成章地进入了第三阶段。

第三阶段，直接调试源码。.NetFramework代码部分开源，需要对VS2008 进行一系列配置之后，才能进行调试。如何进行配置，可以参见ScottGu's 的一篇博文，里面有详细的配置说明（ScottGu,2007）。

4.2 在 WinForm 中寻找过去的痕迹

一个全新的事物因为我们不知道他的底细，所以会感到惶恐。.Net 桌面程序和 MFC 相比有所回归，因为它的程序入口又从一个 Main 开始。但是找不到窗口过程函数，找不到消息循环。更为可怕的是消息作为操作系统和应用程序沟通的载体，似乎退出了历史舞台，取而代之的是事件这一新事物。

于是我开始着手从 WinForm 的荆棘中寻找过去的痕迹，我要找的东西有 3 样：

1. 消息循环在哪儿？
2. 窗口过程函数在哪儿？
3. 在找到上述两点的基础上，我要揭开事件的真实面目。

4.2.1 消息循环在哪儿？

一个典型的 WinForm 程序如下所示：

```
// Form1.cs
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
// Program.cs
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        // 难道所有的秘密都在 Run 里？
        Application.Run(new Form1());
    }
```

直觉告诉我所有的秘密都在 Application.Run 里面。MSDN 进一步证实了我的直觉。他对该函数的解释就是“在当前线程中开始一个消息标准应用程序的消息循环（Begins running a standard application message loop on the current thread.）”。

还犹豫什么，直接调试源码吧！

```
// Application 类
public static void Run(Form mainForm)
{
    ThreadContext.FromCurrent().RunMessageLoop(-1, new ApplicationContext(mainForm));
}

internal void RunMessageLoop(int reason, ApplicationContext context)
{
    IntPtr zero = IntPtr.Zero;
    if (Application.useVisualStyles)
    {
        zero = UnsafeNativeMethods.ThemingScope.Activate();
    }
    try
    {
        this.RunMessageLoopInner(reason, context);
    }
    finally
    {
        UnsafeNativeMethods.ThemingScope.Deactivate(zero);
    }
}

private void RunMessageLoopInner(int reason, ApplicationContext context)
{
……
   if ((!fullModal && !localModal) || ComponentManager is ComponentManager)
   {
        result = ComponentManager.FPushMessageLoop(componentID, reason, 0);
   }
   else if (reason == NativeMethods.MSOCM.msoloopDoEvents || reason ==
   NativeMethods.MSOCM.msoloopDoEventsModal)
   {
        result = LocalModalMessageLoop(null);
   }
   else
   {
        result = LocalModalMessageLoop(currentForm);
   }
……
}

bool UnsafeNativeMethods.IMsoComponentManager.FPushMessageLoop(
                                                       int dwComponentID,
                                                       int reason,
                                                       int pvLoopData          // PVOID
                                                       )
{
```

```
……
   bool continueLoop = true;
   // 多么熟悉的 While……消息循环
   while (continueLoop)
   {
      bool peeked = UnsafeNativeMethods.PeekMessage(ref msg, NativeMethods.NullHandleRef, 0, 0,
NativeMethods.PM_NOREMOVE);
      if (peeked)
      {
         // GetMessage 从消息队列中拿到消息
         if (msg.hwnd != IntPtr.Zero && SafeNativeMethods.IsWindowUnicode(new HandleRef(null,
         msg.hwnd)))
          {
              unicodeWindow = true;
              UnsafeNativeMethods.GetMessageW(ref msg, NativeMethods.NullHandleRef, 0, 0);
          }
          else
          {
              unicodeWindow = false;
              UnsafeNativeMethods.GetMessageA(ref msg, NativeMethods.NullHandleRef, 0, 0);
           }
          // 遇到 WM_QUIT 一定会退出循环
         if (msg.message == NativeMethods.WM_QUIT)
         {
             ……
             continueLoop = false;
             break;
          }
          // 开始分发消息
          if (!component.FPreTranslateMessage(ref msg))
          {
              UnsafeNativeMethods.TranslateMessage(ref msg);
              if (unicodeWindow)
              {
                  UnsafeNativeMethods.DispatchMessageW(ref msg);
               }
               else
               {
                  UnsafeNativeMethods.DispatchMessageA(ref msg);
                }
          }
      }
      else
      {
          ……
      }
   }
   return !continueLoop;
 }
```

上述代码，我作了大量的简化。我并不力求去探求 Application.Run 里所有的细节，只是寻找到隐藏在里面的消息循环。相信这种“眼见为实”的感觉远比 MSDN 里的一句话来得真切。

4.2.2 窗口过程函数在哪儿？

过去的 Win32，窗口过程函数一目了然。

MFC 通过类成员函数指针将窗口过程函数分解到一个一个类的消息响应函数里。

WinForm 呢？相信万变不离其中，他也应该有一个类似处理消息的函数。这个函数是 Form 的一个保护函数 WndProc。实际上我在调试这部分代码是，也是一副残垣断壁之景，微软的开源还是半遮半掩。我会尽可能地给读者一个较为全面的印象，其实细节倒真的不必计较。我们只是需要找到.Net 在如何切割窗口过程函数。我们以 WM_CREATE 为例。

```
// Form.cs
protected override void WndProc(ref Message m)
{
    switch (m.Msg)
    {
        ……
        case NativeMethods.WM_CREATE:
                    WmCreate(ref m);
                    break;
        case NativeMethods.WM_ERASEBKGND:
                    WmEraseBkgnd(ref m);
                    break;
        default:
                    base.WndProc(ref m);
                    break;
     }
}

private void WmCreate(ref Message m)
{
     ……
     base.WndProc(ref m);
     ……
}

// ContainerControl.cs
// Form 的基类为 ContainerControl，其 WndProc 没有处理 WM_CREATE 消息，继续上溯
protected override void WndProc(ref Message m)
{
   switch (m.Msg)
   {
      case NativeMethods.WM_SETFOCUS:
           WmSetFocus(ref m);
           break;
      default:
           base.WndProc(ref m);
           break;
   }
}
```

```
// ScrollableControl.cs
// ContainerControl 的基类为 ScrollableControl,其 WndProc 也没有处理 WM_CREATE 消息，
// 继续上溯
protected override void WndProc(ref Message m)
{
   switch (m.Msg)
  {
       case NativeMethods.WM_VSCROLL:
                   WmVScroll(ref m);
                   break;
       case NativeMethods.WM_HSCROLL:
                   WmHScroll(ref m);
                   break;
       case NativeMethods.WM_SETTINGCHANGE:
                   WmSettingChange(ref m);
                   break;
       default:
                   base.WndProc(ref m);
                   break;
   }
}

// Control.cs
// 处理 WM_CREATE 消息
protected virtual void WndProc(ref Message m)
{
......
            switch (m.Msg)
           {
           ......
                case NativeMethods.WM_CREATE:
                    WmCreate(ref m);
                    break;
            }
......
}

private void WmCreate(ref Message m)
{
            DefWndProc(ref m);
            if (parent != null)
           {
                parent.UpdateChildZOrder(this);
            }
            UpdateBounds();

            // Let any interested sites know that we've now created a handle
            //
            OnHandleCreated(EventArgs.Empty);

            if (!GetStyle(ControlStyles.CacheText)) {
                text = null;
            }
 }
```

```
protected virtual void OnHandleCreated(EventArgs e)
{
            ……
            EventHandler handler = (EventHandler)Events[EventHandleCreated];
            if (handler != null) handler(this, e);
            ……
}

// Form1.Designer.cs
private void InitializeComponent()
{
            ……

            this.HandleCreated += new System.EventHandler(this.Form1_HandleCreated);
            ……
}

// Form1.cs
private void Form1_HandleCreated(Object sender, EventArgs e)
{
        MessageBox.Show("Control.HandleCreated event.");
}
```

尽管我已经作了大量的简化，但是我还要再进一步简化，只需要一头一尾。如下图所示：

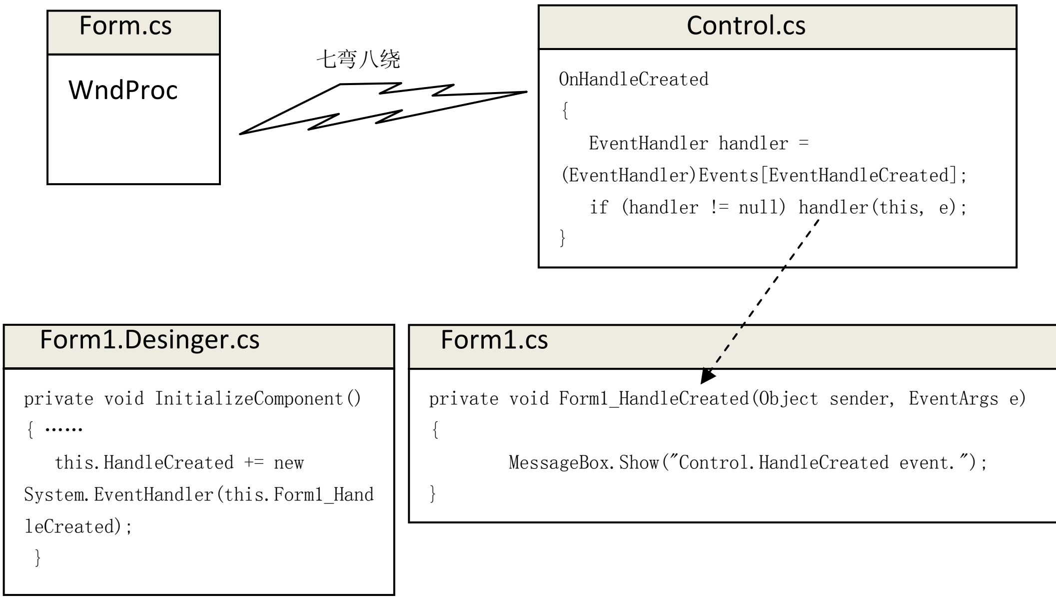

图 4-3 消息是如何被分解的

传统的消息循环被 EventHandler 给切割成一小快一小块。在 MFC 里，消息响应函数切割窗口过程，在 WinForm 里切割的刀换成了事件。何其的相似！

我再总结一下我们整个探索的思路，我们从 Form 的窗口过程函数（WndProc）入手，发现窗口过程被事件切割，而事件的实质是一个预定义的委托。因此，下面我们先从委托认识入手，然后认识事件。

4.3 委托和事件

4.3.1 委托的实质

如果从函数指针，类成员函数指针这样一路认识下来，我们可以说委托的实质是一个指向方法的指针。只不过和函数指针和类成员函数指针不同的地方在于：

1. 它是一个类型安全的对象；
2. 它可以指向全局函数，类的方法，类的静态方法甚至是指向多个方法；
3. 它除了支持同步调用甚至支持异步调用。

函数指针，类成员函数指针和委托一脉相承！

在 C#里用 delegate 关键字来表示一个委托。比如，如下代码声明了一个委托。

```
public delegate int BinaryOp (Object target,uint functionAddress);
```

我们可以用 Reflector 查看，以探求 delegate 关键字的实质。

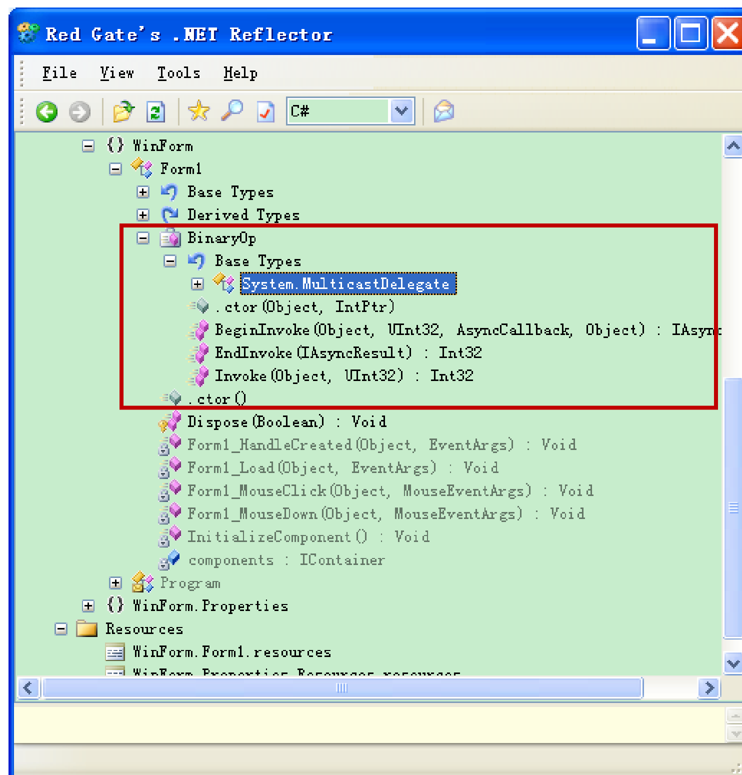

图 4-4 Refelctor 探求 delegate 关键字实质

可以看出来，当编译器遇到 delegate 关键字，实质上是会把 BinaryOp 编译成一个继承多播委托的密闭类。如下所示，则是编译器替我们做的工作。BinaryOp 主要有一个构造函数和三个方法，Invoke 主要用来同步调用，BeginInvoke 和 EndInvoke 主要用来异步调用。Andrew Troelsen 讨论了编译器如何编译 delegate 的完整规则（Andrew Troelsen），而 Jeffery Richter 大师讨论了委托底层的实现。

```
public delegate int BinaryOp (int x, int y);
// 编译器将上面一句话编译成下面的一个继承多播委托的密闭类
sealed class BinaryOp : System.MulticastDelegate
{
    public BinaryOp(object target, uint functionAddress);
    public int Invoke(int x, int y);
    public IAsyncResult BeginInvoke(int x, int y, AsyncCallback cb, object state);
    public int EndInvoke(IAsyncResult result);
}
```

4.3.2 事件的实质

我们用同样的方法来探求 event 关键字的实质。从下面的代码可以看到，op 事件确实是一个 BinaryOp 类型的委托，，它总是被声明为 private。另外，它还有两个方法，分别是 add_op 和 remove_op，这两个方法分别用于注册委托类型的方法和取消注册。

```
public delegate int BinaryOp (int x, int y);
public event BinaryOp op;
// 编译器生成的代码
private BinaryOp op;
public void add_op(BinaryOp value)
{
    this.op = (BinaryOp)Delegate.Combine(this.op,value);
}
public void remove_op(BinaryOp value)
{
    this.op = (BinaryOp)Delegate.Remove(this.op,value);
}
```

这样的封装有何意义呢？经过一番思考，我得出了结论。我们先看看没有事件的情况。

```
public class TestBinaryOp
{
    public delegate int BinaryOp (int x, int y);
    public static int Add(int x,int y) {return x+y;}
    public static int Subtract(int x, int y) {return x-y;}
    public BinaryOp op_delegate;
    public static void main()
    {
        TestBinaryOp testBinaryOp = new TestBinaryOp();
        testBinaryOp.op_delegate = new BinaryOp(Add);
        testBinaryOp.op_delegate += new BinaryOp(Subtarct);
```

```
        testBinaryOp.op_delegate ();
    }
}
```

这样的程序存在如下问题，从 OO 的角度看：op_delegate 作为一个 public 变量被完全暴露出来，外部可以对它进行随意的赋值，严重地破坏了对象的封装性。

于是我们会将该变量进行如下封装。

```
public class TestBinaryOp
{
    public delegate int BinaryOp (int x, int y);
    public static int Add(int x,int y) {return x+y;}
    public static int Subtract(int x, int y) {return x-y;}
    private BinaryOp op_delegate;
    public void add_op_delegate(BinaryOp value)
    {
      this. op_delegate = (BinaryOp)Delegate.Combine(this.op,value);
    }
    public void remove_op_delegate (BinaryOp value)
    {
       this.op_delegate = (BinaryOp)Delegate.Remove(this.op,value);
    }
    public void Trigger()
    {
       If(this.op_delegate != null)
            this.op_delegate();
    }
    public static void main()
    {
        TestBinaryOp testBinaryOp = new TestBinaryOp();
        testBinaryOp.add_op_delegate(new BinaryOp(add));
        testBinaryOp.add_op_delegate(new BinaryOp(Subtract));
        testBinaryOp.Trigger();
    }
}
```

这样的封装还带来了另外一个好处，从订阅和发布的角度看：委托应该由发布者触发，而不应该由客户端来触发。这样的一个好处我要对照上述代码详细解释一下。

main 是客户端程序,它订阅了委托（add_op_delegate）。而 TestBinaryOp 是发布者，它发布了一个委托变量 op_delegate。如果直接在 main 里进行如下的调用。

```
public class TestBinaryOp
{
    public BinaryOp op_delegate;
    public static void main()
    {
        …….
        testBinaryOp.op_delegate ();
    }
}
```

可以吗？答案是当然可以，但是不好。怎么不好呢？打个比方，比如客户者是一个老百姓，他发现路上有个井盖没有了，于是他订阅了事件[6]（“井盖没有了”），如果紧接着调用上述的代码，那意味着这个老百姓要处理这个“井盖没有了”的情况。可以吗？显然可以！合理吗？显然不合理。做这件事情的当然是我们的好政府（事件的发布者）。

4.4 小结

历史演变到现在，真的是臻于完美。从早期的函数指针到类成员函数指针再到委托和事件，其间的演变精彩纷呈。但是技术的革新总是能给我们一次，一次，又一次地惊奇。下一章里，我们正式进入“路由事件”之门……

[6]我们已经说过事件从本质上说是一种封装了的委托。