第5章　理解Windows消息

本章内容：

- 什么是消息
- 消息的类型
- Windows的消息系统是如何工作的
- Delphi的消息系统
- 消息处理
- 发送自己的消息
- 非标准的消息
- 一个消息系统的剖析：VCL
- 消息与事件之间的关系

尽管VCL组件可以通过Object Pascal事件来响应Win32消息，但是，作为一个Win32程序员，理解Windows的消息系统还是非常必要的。

作为一个Delphi应用程序开发者，你将发现，在大部分情况下使用 VCL的事件就已经足够了，只有少数情况才需要深入到Win32消息处理的世界。然而，作为一个 Delphi组件开发者，你将和消息成为好朋友，因为你不得不直接处理许多Windows消息。

5.1　什么是消息

消息，就是指Windows发出的一个通知，告诉应用程序某个事情发生了。例如，单击鼠标、改变窗口尺寸、按下键盘上的一个键都会使Windows发送一个消息给应用程序。

消息本身是作为一个记录传递给应用程序的，这个记录中包含了消息的类型以及其他信息。例如，对于单击鼠标所产生的消息来说，这个记录中包含了单击鼠标时的坐标。这个记录类型叫做 TMsg，它在Windows单元中是这样声明的：

```
type
  TMsg = packed record
    hwnd: HWND;          //窗口句柄
    message: UINT;       //消息常量标识符
    wParam: WPARAM;  // 32位消息的特定附加信息
    lParam: LPARAM;     // 32位消息的特定附加信息
    time: DWORD;         //消息创建时的时间
    pt: TPoint;          //消息创建时的鼠标位置
end;
```

消息中有什么？

是否觉得一个消息记录中的信息像希腊语一样？如果是这样，那么看一看下面的解释：

hwnd　32位的窗口句柄。窗口可以是任何类型的屏幕对象，因为Win32能够维护大多数可视对象的句柄(窗口、对话框、按钮、编辑框等)。

message　用于区别其他消息的常量值，这些常量可以是Windows单元中预定义的常量，也

可以是自定义的常量。

wParam 通常是一个与消息有关的常量值，也可能是窗口或控件的句柄。

lParam 通常是一个指向内存中数据的指针。由于WParam、lParam和Pointer都是32位的，因此，它们之间可以相互转换。

现在你已对消息的组成有了一定了解，下面来看看各种类型的消息。

5.2 消息的类型

Win32中预定义了一些消息常量。消息常量由 TMsg记录的message域来传递。这些常量是在Messages单元中定义的。消息常量往往以字母 WM打头，代表Windows Message。表5-1列出了一些常用的Windows消息以及它们的含义和值。

表5-1 常用的Windows消息

消息标识符	值	含 义
WM_ACTIVATE	$0006	窗口被激活或被取消激活
WM_CHAR	$0102	按下某个键，并且已经发送了 WM_KEYDOWN和WM_KEYUP消息
WM_CLOSE	$0010	窗口将要关闭
WM_KEYDOWN	$0100	按下一个键
WM_KEYUP	$0101	按键被释放
WM_LBUTTONDOWN	$0201	按下鼠标左键
WM_MOUSEMOVE	$0200	鼠标移动
WM_PAINT	$000F	窗口的客户区需要重画
WM_TIMER	$0113	发生了定时器事件
WM_QUIT	$0012	程序将要退出

5.3 Windows消息系统是如何工作的

Windows的消息系统是由3个部分组成的：

- 消息队列。Windows能够为所有的应用程序维护一个消息队列。应用程序必须从消息队列中获取消息，然后分派给某个窗口。
- 消息循环。通过这个循环机制应用程序从消息队列中检索消息，再把它分派给适当的窗口，然后继续从消息队列中检索下一条消息，再分派给适当的窗口，依次进行。
- 窗口过程。每个窗口都有一个窗口过程来接收传递给窗口的消息，它的任务就是获取消息然后响应它。窗口过程是一个回调函数；处理了一个消息后，它通常要返回一个值给 Windows。

注意 回调函数是程序中的一种函数，它是由Windows或外部模块调用的。

一个消息从产生到被一个窗口响应，其中有5个步骤：

1) 系统中发生了某个事件。
2) Windows把这个事件翻译为消息，然后把它放到消息队列中。
3) 应用程序从消息队列中接收到这个消息，把它存放在 TMsg记录中。
4) 应用程序把消息传递给一个适当的窗口的窗口过程。
5) 窗口过程响应这个消息并进行处理。

步骤3和4构成了应用程序的消息循环。消息循环往往是 Windows应用程序的核心，因为消息循环使一个应用程序能够响应外部的事件。消息循环的任务就是从消息队列中检索消息，然后把消息传递

给适当的窗口。如果消息队列中没有消息， Windows就允许其他应用程序处理它们的消息。图 5-1显示了这5个步骤。

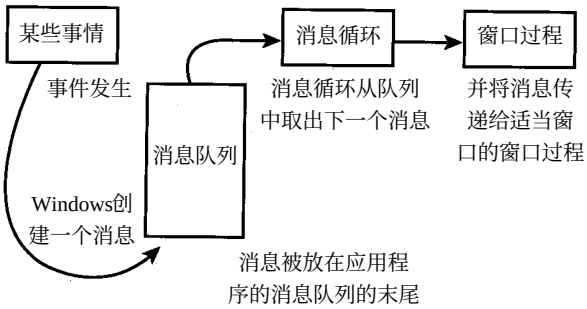

图5-1 Windows消息系统

5.4 Delphi的消息系统

VCL替你处理了许多Windows消息系统的细节。例如，消息循环已经封装到VCL的Forms单元，因此，用不着考虑怎样从消息队列中检索消息然后再把它指派给某个窗口。 Delphi还把Windows的TMsg记录中的信息映射为TMessage记录：

```
type
  TMessage = record
    Msg: Cardinal;
    case Integer of
      0: (
        WParam: Longint;
        LParam: Longint;
        Result: Longint);
      1: (
        WParamLo: Word;
        WParamHi: Word;
        LParamLo: Word;
        LParamHi: Word;
        ResultLo: Word;
        ResultHi: Word);
  end;
```

需要注意的是， TMessage记录中的信息要比TMsg中的信息少一点，这是因为Delphi把有些信息封装起来，TMessage中只包含了处理消息所必需的信息。

特别要注意的是， TMessage记录中包含了一个Result域。正如前面提到的那样，某些信息需要窗口过程返回一个值。在Delphi中，要返回一个值，只要对 TMessage记录的Result域赋值。后面的5.5.2节“ 对Result域赋值 ”将详细解释这一点。

特殊的消息记录

除了通用的TMessage记录外， Delphi为每个Windows消息定义了一个特殊的消息记录，这样就不必从wParam域和 lParam域中分解出有关信息。这些特殊的记录可以在Messages单元中找到。作为一个例子， 下面是一个鼠标的消息记录：

```
type
  TWMMouse = record
    Msg: Cardinal;
```

```
    Keys: Longint;
    case Integer of
      0: (
        XPos: Smallint;
        YPos: Smallint);
      1: (
        Pos: TSmallPoint;
        Result: Longint);
  end;
```

其他鼠标消息(如 WM_LBUTTONDOWN和 WM_RBUTTONUP)，它们的消息记录像下面这样声明：

```
TWMRButtonUp = TWMMouse;
TWMLButtonDown = TWMMouse;
```

注意 几乎每一个Windows消息都定义一个消息记录。消息记录的命名是在消息标识符的前面加上大写字母T并去掉下划线。例如，与消息WM_SETFONT对应的消息记录名称叫TWMSetFont。顺便说一下，TMessage记录对所有的消息都是适用的，而特殊的消息记录只适用于某些消息。

5.5 消息处理

所谓消息处理，就是应用程序以某种方式响应 Windows消息。在一个标准的Windows应用程序中，消息是由窗口过程处理的。不过， Delphi封装了窗口过程，这使得消息处理简单多了。以前，一个窗口程序需要处理各种消息，而在 Delphi中，每个消息都有各自的过程。用于消息处理的过程必须满足下列3个条件：

- 这个过程必须是一个对象中的方法。
- 这个过程必须有一个var参数，变量的类型是TMessage或其他特殊的消息记录。
- 声明这个过程时，必须使用message指示符，后面跟要处理的消息的常量值。

下面是声明一个处理WM_PAINT消息的过程的代码：

```
procedure WMPaint(var Msg: TWMPaint); message WM_PAINT;
```

注意 给用于消息处理的过程命名时采用这样的约定：过程名与消息的标识符一致，但不要全部大写，也不要有下划线。

作为一个例子，下面写一个处理WM_PAINT消息的过程。

首先，创建一个新的空白的项目。然后访问这个项目的代码编辑窗口并在 TForm对象的private部分加入下面这一行。

```
procedure WMPaint(var Msg: TWMPaint); message WM_PAINT;
```

然后，在单元的implementation部分实现这个过程。记住，要在这个过程的名称前加 TForm1，但不需要message指示符。

```
procedure TForm1.WMPaint(var Msg: TWMPaint);
begin
  Beep;
  inherited;
end;
```

请注意上述代码中的inherited语句。当想传递消息给祖先对象的处理过程时，请调用inherited语句。这个语句的作用是把消息传递给TForm中处理WM_PAINT消息的过程。

注意 调用inherited时不需要给出祖先类中方法的名称。因为方法的名称并不重要，Delphi是根据消息的常量来区分过程的。

清单5-1列出了一个完整的单元，它演示了怎样处理 WM_PAINT消息。要创建这个项目是很简单

的，只要先创建一个空白的项目，然后在TForm1对象的声明中声明一个过程。

清单5-1 GetMess：消息处理举例

```
unit GMMain;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  TForm1 = class(TForm)
  private
    procedure WMPaint(var Msg: TWMPaint); message WM_PAINT;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.WMPaint(var Msg: TWMPaint);
begin
  MessageBeep(0);
  inherited;
end;

end.
```

当WM_PAINT消息发生时，它就被传递给WMPaint过程，WMPaint过程则调用MessageBeep()来发出声音，然后把消息传递给祖先类的过程。

MessageBeep(): **穷人的调试器**

这里稍稍偏一点题，讲一下MessageBeep()过程。这个过程是Win32 API中最有用和最直观的例程之一。它的使用很简单，只要传递一个预定义的常量，Windows就会使PC机的喇叭响一下(如果有声卡，将演奏一个WAV文件)。MessageBeep过程经常用于调试程序。

如果想知道程序是否执行到了某个地方，但又不想劳驾调试器和断点，MessageBeep()是最适合的。因为它不需要句柄或其他任何Windows资源，可以在代码的任何地方使用它，就像一位著名人士曾经说的：“MessageBeep()是为那些不熟悉调试器的人设计的”。如果有声卡，只要传递一个预定义的常量给MessageBeep()，就能演奏一段声音。这些常量的定义在Win32 API帮助中MessageBeep()主题的下面。

如果觉得这个过程的名称和参数太长。也可以用SysUtils单元中的Beep()过程代替。Beep()实际上调用了MessageBeep()，但参数设为0。

5.5.1 消息处理：不是无约定的

与响应Delphi事件不同的是，处理Windows消息不是无约定的。当在处理一个消息时，Windows总是指望做一些事情。VCL已经内置了基本的消息处理功能，必须要做的就是调用inherited。可以这样认为：写一个消息处理过程是为了做一些你要做的事情，调用inherited是为了做一些Windows要做

的事情。

注意 在处理消息时，还有些事情是有限制的。例如，在处理WM_KILLFOCUS消息时，就不允许将焦点设给另一个控件。

为了证明inherited的作用，可以在清单5-1中把WMPaint()方法中的inherited去掉，即代码变成：

```
procedure TForm1.WMPaint(var Msg: TWMPaint);
begin
  MessageBeep(0);
end;
```

因为这样Windows就没有机会对WM_PAINT消息进行基本的处理，所以，窗体就不会被画出来。

有时候，可能不希望调用inherited。例如，在处理WM_SYSCOMMAND消息时，为了防止窗口被最大化或最小化，就不能调用inherited。

5.5.2 对Result域赋值

当处理某些Windows消息时，Windows希望返回一个值。典型的例子是WM_CTLCOLOR消息。当处理这个消息时，Windows希望返回一个画刷的句柄，Windows用这个画刷来画对话框或控件(Delphi为每个组件提供了Color属性，这个例子只是为了说明问题)。要返回一个值，只要在消息处理过程中调用了inherited之后对TMessage(或其他消息记录)中的Result域赋值。例如，当处理WM_CTLCOLOR消息时，可以这样返回一个画刷的句柄：

```
procedure TForm1.WMCtlColor(var Msg: TWMCtlColor);
var
  BrushHand: hBrush;
begin
  inherited;
  { 创建一个画刷的句柄，并放进BrushHand变量 }
  Msg.Result := BrushHand;
end;
```

5.5.3 TApplication的OnMessage事件

要处理Windows的消息，也可以利用TApplication的OnMessage事件。建立了响应OnMessage事件的处理过程后，只要从消息队列中检索到一个消息，就会触发OnMessage事件。这样，在Windows本身对消息处理之前就会调用响应OnMessage事件的处理过程。响应Application.OnMessage事件的处理过程必须是TMessageEvent类型，可以这样声明：

```
procedure SomeObject.AppMessageHandler(var Msg: TMsg;
  var Handled: Boolean);
```

所有消息参数都是通过Msg传递过来的(注意前面讲过Msg参数是Windows TMsg记录类型)。要对Handled参数赋一个布尔值，以表明是否已经处理了这个消息。

要建立一个响应OnMessage事件的处理过程，第一步是声明和定义一个TMessageEvent类型的方法。例如，下面就定义了一个方法，用于统计应用程序总共收到多少消息：

```
var
  NumMessages: Integer;

procedure Form1.AppMessageHandler(var Msg: TMsg; var Handled: Boolean);
begin
  Inc(NumMessages);
  Handled := False;
end;
```

第二步也是最后一步，就是把上面这个方法赋给 Applicaton.OnMessage事件。这要在DPR文件中，在创建了项目的窗体之后但调用Application.Run之前加入下面语句：

```
Application.OnMessage := Form1.AppMessageHandler;
```

OnMessage的一个限制是，它响应的是从消息队列中检索到的消息，而不是直接发给某个窗口程序的消息。第13章“核心技术”将介绍怎样克服这个缺陷。

提示 OnMessage事件将捕获到发送给应用程序的所有消息。这是一个非常繁忙的事件(每秒钟可能有数千个消息)。因此，在处理OnMessage事件的处理过程中，不可能什么消息都处理，那样会耗费很多时间，使应用程序变慢。显然，在处理OnMessage事件的处理过程中设断点是非常不明智的。

5.6 发送自己的消息

就像Windows发送消息给应用程序一样，也可以在窗口与控件之间发送消息。 Delphi提供了几种在一个应用程序内部发送消息的方式：调用 Perform()(这种方式不依赖于 Windows API)，以及调用SendMessage()、PostMessage() API函数。

5.6.1 Perform()

VCL的Perform()方法适用于所有的TControl派生对象。 Perform()可以向任何一个窗体或控件发送消息，只需要知道窗体或控件的实例。 Perform()需要传递3个参数：消息标识符、 wParam和 lParam。Perform()是这样声明的：

```
function TControl.Perform(Msg: Cardinal; WParam, LParam: Longint):
  Longint;
```

要给一个窗体或控件发送一个消息，可以参照下面的代码：

```
RetVal := ControlName.Perform(MessageID, wParam, lParam);
```

调用了Perfonn()后，它要等消息得到处理后才返回。 Perform()把3个参数组装成 TMessage记录，然后调用 Dispatch()方法把消息传递给 Windows的消息系统。后面的部分将进一步介绍Dispatch()方法。

5.6.2 sendMessage()和PostMessage()

有的时候，可能需要向一个窗口发送一个消息，而又不知道这个窗口的实例。例如，可能要给一个非 Delphi的窗口发送一个消息，而只有这个窗口的句柄。幸运的是， Windows有两个API函数可以实现这一点：SendMessage()和PostMessage()。这两个函数几乎是一样的，它们的区别是： SendMessage()直接把一个消息发送给窗口过程，等消息被处理后才返回。 PostMessage()只是把消息发送到消息队列，然后立即返回。

SendMessage()和PostMessage()是这样声明的：

```
function SendMessage(hWnd: HWND; Msg: UINT; wParam: WPARAM;
  lParam: LPARAM): LRESULT; stdcall;
function PostMessage(hWnd: HWND; Msg: UINT; wParam: WPARAM;
  lParam: LPARAM): BOOL; stdcall;
```

- hwnd是接收消息的窗口的句柄。
- Msg是消息标识符。
- wParam是32位的特定附加信息。
- lParam是32位的特定附加信息。

注意 尽管SendMessage()和PostMessage()调用方式完全一样，但它们的返回值不一样。

SendMessage()返回此消息被处理的结果值，而PostMessage()返回一个布尔值，表示消息是否已被放到消息队列中。

5.7 非标准的消息

直到现在，我们讨论的还是标准的Windows消息(其标识符是WM_XXX)。然而，另外两种消息也值得讨论；通知消息和用户自定义消息。

5.7.1 通知消息

通知消息(Notification message)是指这样一种消息，一个窗口内的子控件发生了一些事情，需要通知父窗口。通知消息只适用于标准的窗口控件如按钮、列表框、组合框、编辑框，以及 Windows 95公共控件如树状视图、列表视图等。例如，单击或双击一个控件、在控件中选择部分文本、操作控件的滚动条都会产生通知消息。

要处理通知消息，可以像前面介绍的那样写一个消息处理过程。表 5-2列出了Win32中标准Windows控件的所有通知消息。

表5-2 Win32中标准Windows控件的通知消息

消 息	含 义
按钮	
BN_CLICKED	用户单击了按钮
BN_DISABLE	按钮被禁止
BN_DOUBLECLICKED	用户双击了按钮
BN_HILITE	用户加亮了按钮
BN_PAINT	按钮应当重画
BN_UNHILITE	加亮应当去掉
组合框	
CBN_CLOSEUP	组合框的列表框被关闭
CBN_DBLCLK	用户双击了一个字符串
CBN_DROPDOWN	组合框的列表框被拉出
CBN_EDITCHANGE	用户修改了编辑框中的文本
CBN_EDITUPDATE	编辑框内的文本即将更新
CBN_ERRSPACE	组合框内存不足
CBN_KILLFOCUS	组合框失去输入焦点
CBN_SELCHANGE	在组合框中选择了一项
CBN_SELENDCANCEL	用户的选择应当被取消
CBN_SELENDOK	用户的选择是合法的
CBN_SETFOCUS	组合框获得输入焦点
编辑框	
EN_CHANGE	编辑框中的文本已更新
EN_ERRSPACE	编辑框内存不足
EN_HSCROLL	用户点击了水平滚动条
EN_KILLFOCUS	编辑框正在失去输入焦点
EN_MAXTEXT	插入的内容被截断
EN_SETFOCUS	编辑框获得输入焦点
EN_UPDATE	编辑框中的文本将要更新
EN_VSCROLL	用户点击了垂直滚动条

(续)

消 息	含 义
列表框	
LBN_DBLCLK	用户双击了一项
LBN_ERRSPACE	列表框内存不够
LBN_KILLFOCUS	列表框正在失去输入焦点
LBN_SELCANCEL	选择被取消
LBN_SELCHANGE	选择了另一项
LBN_SETFOCUS	列表框获得输入焦点

5.7.2 VCL内部的消息

VCL中包含了大量的内部消息。尽管应用程序一般很少用到这些消息，但组件编写者可能会觉得这些消息很有用。这些内部的消息往往以 CM_(代表component message)开头，它们用于管理VCL内部的事物，诸如焦点、颜色、可视性、窗口重建、拖放等。在 Delphi在线帮助的“ Creating Custom Components ”部分可以找到这些消息的完整列表。

5.7.3 用户自定义的消息

有些情况下，一个应用程序可能需要向自己发送消息，或者在两个应用程序之间发送消息。这时你会有这样一个问题：“ 为什么要发送消息而不是直接调用一个过程 ”。这个问题问得好，有这样几个答案。首先，消息可以让你不需要知道接收者的确切类型。同时，消息可以有选择地处理。如果接收者对消息没有做任何处理，不会造成任何后果。最后，消息可以广播给多个接收者，而要同时调用几个过程则比较困难。

1. 在应用程序内发送消息

一个应用程序要发送消息给自己是很容易的，只要调用Perform()、SendMessage()或PostMessage()，并且使消息常量的值为 WM_USER + 100到$7FFF(这个范围是Windows为用户自定义消息保留的)。

```
const
 SX_MYMESSAGE = WM_USER + 100;

begin
  SomeForm.Perform(SX_MYMESSAGE, 0, 0);
  {或者}
  SendMessage(SomeForm.Handle, SX_MYMESSAGE, 0, 0);
  {或者}
  PostMessage(SomeForm.Handle, SX_MYMESSAGE, 0, 0);
  .
  .
  .
end;
```

然后声明和定义一个普通的消息处理过程来处理SX_MYMESSAGE消息：

```
TForm1 = class(TForm)
  .
  .
  .
private
  procedure SXMyMessage(var Msg: TMessage); message SX_MYMESSAGE;
end;

procedure TForm1.SXMyMessage(var Msg: TMessage);
```

```
begin
  MessageDlg('She turned me into a newt!', mtInformation, [mbOk], 0);
end;
```

正如你看到的那样，处理用户自定义的消息与处理标准的 Windows消息几乎没有什么不同。真正的关键在于：必须声明一个消息常量，它的值必须从 WM_USER + 100开始。最好为自定义的消息起一个表明它的用途的名字。

注意 除非应用程序已经建立了相应的消息处理过程，否则，不要发送自定义的消息。由于每个窗口都可以独立地定义消息常量的值，因此，发送自定义消息具有潜在的危险，除非在接受处理时非常地小心。

2. 在应用程序之间发送消息

如果要在两个或多个应用程序之间发送消息，那么最好要调用 RegisterWindowMessage()函数。这个函数能够确保每个应用程序使用一致的消息编号。

RegisterWindowMessage()需要传递一个以null结束的字符串，并返回一个范围从$C000到$FFFF的新的消息常量。这就意味着，在要发送消息的应用程序之间，每个应用程序都必须传递相同的字符串给RegisterWindowMessage()函数；而 Windows也会返回相同的消息常量值。调用RegisterWindowMessage()的真正好处是，对于任何给定的字符串将返回一个在整个系统中都唯一的消息常量，这样，就可以放心地向所有的窗口广播消息。不过，处理这样的消息稍稍麻烦一点，因为只有在运行时才知道消息的标识符，所以无法调用那些标准的API函数，只能覆盖一个控件的WndProc()或DefaultHandler()等方法。第13章将介绍怎样处理这种消息。

注意 RegisterWindowMessage()函数的返回值在不同的窗口会话是不同的，返回值只在运行时有意义。

3. 广播消息

TWinControl的派生对象可以调用Broadcast()来向它的子控件广播一个消息。当需要向一组组件发送相同的消息时，你要用得这种技术。例如， Panel1可以给它的所有子控件发送一个叫 UM_FOO的自定义消息，代码如下：

```
var
  M: TMessage;
begin
  with M do
  begin
    Message := UM_FOO;
    wParam := 0;
    lParam := 0;
    Result := 0;
  end;
  Panel1.Broadcast(M);
end;
```

5.8 一个消息系统的剖析：VCL

谈到VCL的消息系统时，除了message指示符外，还有很多值得讨论的。在Windows发出一个消息后，要经过两步才能达到你的消息处理过程(也可能步骤更少)。在这条路线上，你可以操作消息。

首先要讨论的是Application.ProcessMessage()，这个方法包含VCL主消息循环。另一个是Application.OnMessage事件，这个事件是当ProcessMessage()从消息队列中检索到一个消息时触发的。因为发送消息不需要排队，所以不会触发OnMessage事件。

另一个要讨论的是DispatchMessage()，这个API函数用于把消息传递给StdWndProc()函数。它由Win32直接调用以发送消息。StdWndProc()函数起到汇集的作用，它从Windows接收消息，再把消息发送给某个对象。

VCL对象用于接收消息的方法叫MainWndProc()。通过MainWndProc()，可以对消息进行任何特殊的处理。不过，一般情况下很少直接用MainWndProc()来处理消息，除非不想让消息通过VCL的消息系统派发。

离开MainWndProc()后，消息被传递给对象的WndProc()方法，然后进入VCL的派发机制。派发机制即Dispatch()方法把消息派发给一个消息句柄。

消息到达处理该消息的处理过程后，经过处理过程的处理和最后的 inherited语句，消息来到对象的DefaultHandler()。这个方法对消息进行最后的处理，然后把消息传递给Windows的DefWindowProc()函数或其他默认的窗口过程。图5-2显示了VCL的消息处理机制。

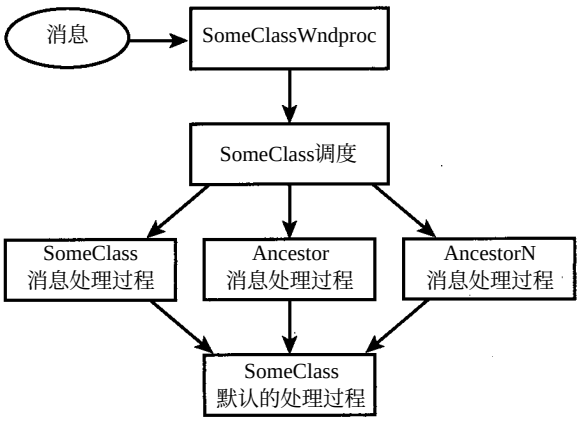

图5-2 VCL的消息系统

注意 处理消息时，最后应当调用inherited，除非不想进行常规的消息处理。

提示 由于所有未处理的消息都会传递给DefaultHandler()，因此，DefaultHandler()最适合于处理应用程序之间的消息，在那里通过调用RegisterWindowMessage()函数得到消息常量的值。

为了更好地理解VCL的消息系统，下面将创建一个小程序，分别用Application.OnMessage事件、WndProc()、消息处理过程和DefaultHandler()来处理消息。这个程序叫CatchIt，它的主窗体如图5-3所示。

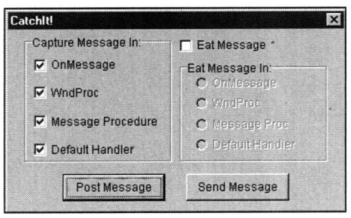

图5-3 CatchIt程序的主窗体

下面的代码中包含了处理PostMessButton和 SendMessButton的 OnClick事件的处理过程。前者调用PostMessage()把一个自定义的消息发送给窗体，后者调用SendMessge()把一个自定义的消息发送给窗体。为了区别，调用PostMessage()时wParam参数设为1，调用SendMessage()时wParam参数设为0。代码如下：

```
procedure TMainForm.PostMessButtonClick(Sender: TObject);
{ posts message to form }
begin
  PostMessage(Handle, SX_MYMESSAGE, 1, 0);
end;
```

```
procedure TMainForm.SendMessButtonClick(Sender: TObject);
{ sends message to form }
begin
  SendMessage(Handle, SX_MYMESSAGE, 0, 0); // send message to form
end;
```

下面这个应用程序演示了在OnMessage、WndProc()、消息处理过程或DefaultHandler()中怎样把消息“吃掉”(即不触发继承的行为，并使消息不再进入VCL的消息系统)。清单5-2列出了这个应用程序的源代码，从中可以看出消息的流向。

清单5-2 CIMain.Pas的代码

```
unit CIMain;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Menus;

const
  SX_MYMESSAGE = WM_USER;                 // 用户自定义的消息
  MessString = '%s message now in %s.';   // 提示用户的字符串

type
  TMainForm = class(TForm)
    GroupBox1: TGroupBox;
    PostMessButton: TButton;
    WndProcCB: TCheckBox;
    MessProcCB: TCheckBox;
    DefHandCB: TCheckBox;
    SendMessButton: TButton;
    AppMsgCB: TCheckBox;
    EatMsgCB: TCheckBox;
    EatMsgGB: TGroupBox;
    OnMsgRB: TRadioButton;
    WndProcRB: TRadioButton;
    MsgProcRB: TRadioButton;
    DefHandlerRB: TRadioButton;
    procedure PostMessButtonClick(Sender: TObject);
    procedure SendMessButtonClick(Sender: TObject);
    procedure EatMsgCBClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure AppMsgCBClick(Sender: TObject);
  private
    { 在应用级处理消息 }
    procedure OnAppMessage(var Msg: TMsg; var Handled: Boolean);
    { 在WndProc级处理消息 }
    procedure WndProc(var Msg: TMessage); override;
    { 在派发后处理消息 }
    procedure SXMyMessage(var Msg: TMessage); message SX_MYMESSAGE;
    { 用DefaultHandler()处理消息 }
    procedure DefaultHandler(var Msg); override;
  end;
var
  MainForm: TMainForm;

implementation
```

```
{$R *.DFM}

const
  // 声明两个字符串，代表消息是被Send或Post
  SendPostStrings: array[0..1] of String = ('Sent', 'Posted');

procedure TMainForm.FormCreate(Sender: TObject);
{ 处理主窗体的OnCreate事件 }
begin
  // 将OnMessage设为自己的OnAppMessage方法
  Application.OnMessage := OnAppMessage;

  // 设置复选框的Tag属性，以保存相应单选按钮的引用

  AppMsgCB.Tag := Longint(OnMsgRB);
  WndProcCB.Tag := Longint(WndProcRB);
  MessProcCB.Tag := Longint(MsgProcRB);
  DefHandCB.Tag := Longint(DefHandlerRB);

  // 设置单选框的Tag属性，以保存相应复选框的引用

  OnMsgRB.Tag := Longint(AppMsgCB);
  WndProcRB.Tag := Longint(WndProcCB);
  MsgProcRB.Tag := Longint(MessProcCB);
  DefHandlerRB.Tag := Longint(DefHandCB);
end;

procedure TMainForm.OnAppMessage(var Msg: TMsg; var Handled: Boolean);
{ 处理OnMessage事件的处理过程 }
begin
  // 检查是否是自定义的消息
  if Msg.Message = SX_MYMESSAGE then
  begin
    if AppMsgCB.Checked then
    begin
      // 让用户知道是什么消息，并设置Handled标志
      ShowMessage(Format(MessString, [SendPostStrings[Msg.WParam],
        'Application.OnMessage']));
      Handled := OnMsgRB.Checked;
    end;
  end;
end;
procedure TMainForm.WndProc(var Msg: TMessage);
{ 窗体的WndProc过程 }
var
  CallInherited: Boolean;
begin
  CallInherited := True;            // 假设我们将调用inherited
  if Msg.Msg = SX_MYMESSAGE then    // 检查是否是用户自定义的消息
  begin
    if WndProcCB.Checked then       // 是否选中了WndProcCB复选框
    begin
      // 让用户知道是什么消息
      ShowMessage(Format(MessString, [SendPostStrings[Msg.WParam],
        'WndProc']));
      // 如果不想“吃掉”消息，就调用inherited。
      CallInherited := not WndProcRB.Checked;
    end;
  end;
```

```
  if CallInherited then inherited WndProc(Msg);
end;

procedure TMainForm.SXMyMessage(var Msg: TMessage);
{ 处理自定义的消息 }
var
  CallInherited: Boolean;
begin
  CallInherited := True;              // 假设将调用inherited
  if MessProcCB.Checked then          // 是否选中了MessProcCB复选框
  begin
    // 让用户知道是什么消息
    ShowMessage(Format(MessString, [SendPostStrings[Msg.WParam],
      'Message Procedure']));
    // 如果不想“吃掉”消息，就调用inherited
    CallInherited := not MsgProcRB.Checked;
  end;
  if CallInherited then Inherited;
end;

procedure TMainForm.DefaultHandler(var Msg);
{ 默认的消息处理处理过程 }
var
  CallInherited: Boolean;
begin
  CallInherited := True;              // 假设要调用inherited
  // 检查是否是用户自定义的消息
  if TMessage(Msg).Msg = SX_MYMESSAGE then   begin
    if DefHandCB.Checked then         // 是否选中了DefHandCB复选框
    begin
      // 让用户知道是什么消息
      ShowMessage(Format(MessString,
        [SendPostStrings[TMessage(Msg).WParam], 'DefaultHandler']));
      // 如果不想“吃掉”消息，就调用inherited。
      CallInherited := not DefHandlerRB.Checked;
    end;
  end;
  if CallInherited then inherited DefaultHandler(Msg);
end;

procedure TMainForm.PostMessButtonClick(Sender: TObject);
{ post消息给窗体 }
begin
  PostMessage(Handle, SX_MYMESSAGE, 1, 0);
end;

procedure TMainForm.SendMessButtonClick(Sender: TObject);
{ send消息给窗体 }
begin
  SendMessage(Handle, SX_MYMESSAGE, 0, 0); // send message to form
end;

procedure TMainForm.AppMsgCBClick(Sender: TObject);
{ 允许/禁止复选框 }
begin
  if EatMsgCB.Checked then
  begin
```

```
      with TRadioButton((Sender as TCheckBox).Tag) do
      begin
        Enabled := TCheckbox(Sender).Checked;
        if not Enabled then Checked := False;
      end;
    end;
  end;

  procedure TMainForm.EatMsgCBClick(Sender: TObject);
  { 禁止/允许单选按钮 }
  var
    i: Integer;
    DoEnable, EatEnabled: Boolean;
  begin
    // 得到禁止/允许标志
    EatEnabled := EatMsgCB.Checked;
    // 遍历GroupBox的所有子控件以禁止/允许和选中/不选中单选按钮
    for i := 0 to EatMsgGB.ControlCount - 1 do
      with EatMsgGB.Controls[i] as TRadioButton do
      begin
        DoEnable := EatEnabled;
        if DoEnable then DoEnable := TCheckbox(Tag).Checked;
        if not DoEnable then Checked := False;
        Enabled := DoEnable;
      end;
  end;

end.
```

注意 在消息处理过程中可以调用 inherited来把消息传递给祖先类的消息处理过程，而在WndProc()、DefaultHandler()中调用inherited时，需要后跟一个过程的名称：

```
inherited WndProc(Msg);
```

你可能注意到了，DefaultHandler()需要传递一个无类型的var参数。这是因为DefaultHandler()假设参数的头一个词是消息号，而并不关心传递的其他信息。因此，可以把这个参数强制转换为 TMessage记录，这样就可以访问它的message域。

5.9 消息与事件之间的关系

你现在已经知道了消息的细节。前面曾经提到， VCL的事件系统封装了许多 Windows的消息。Delphi的事件系统是为了更好地与Windows消息接口而设计的。许多VCL的事件都对应着一个WM_XXX消息。表5-3列出了一些VCL事件及其对应的Windows消息。

表5-3 VCL事件和对应的Windows消息

VCL事件	Windows消息
OnActivate	WM_ACTIVATE
OnClick	WM_XBUTTONDOWN
OnCreate	WM_CREATE
OnDblClick	WM_XBUTTONDBLCLICK
OnKeyDown	WM_KEYDOWN
OnKeyPress	WM_CHAR
OnKeyUp	WIN_KEYUP

(续)

VCL事件	Windows消息
OnPaint	WM_PAINT
OnResize	WM_SIZE
OnTimer	WM_TIMER

表5-3可以方便地为你提供事件-消息对照。

提示 应当尽量用事件而不要用消息。由于事件的处理是无约定的，因此，处理事件比处理消息要简单些。

5.10 总结

现在，你应当已经非常清楚地了解了 Win32的消息系统是怎样工作的以及 VCL是怎样封装消息系统的。尽管Delphi的事件系统很出色，但作为一个Win32程序员，了解消息系统仍然是有必要的。

如果希望了解更多的有关处理消息的知识，请看第 21章“ 编写自定义组件 ”。在那一章中，你将看到本章介绍的知识得到实际应用。在下一章中，你将学习如何按一系列标准写 Delphi代码以使代码更有逻辑并便于共享源代码。