Lewolf
保留版权
签名未验证

第六章 控件设计示例

前面几章已经就面向状态的思想以及 C++ Builder 中如何使用面向状态的技术作了一些介绍，通过阅读前面内容，相信读者已经对 C++ Builder 中提供的这种新的编程技术有所了解。本章将根据前面的内容有针对性的提供一些控件设计的示例，其目的在于示范如何应用前面所讲的内容，而不在于应用这些控件,因此本章所给的控件都是一些非常简单，几乎没有使用价值的示范，但通过这些控件的学习可以是读者掌握如何创建控件以及更好的使用 C++ Builder 的这些新的特征。

1 组合控件 TPickEdit

所谓组合控件是指有多个控件经过组合形成一个新的控件来完成一些特定的功能，实际上 VCL 的大多数控件都是由多个控件或组件组合而成的，下面给出的 TPickEdit 就是一个由 TEdit 和 TSpeedButton 组合而成的控件。

这是一个非常简单的控件，最初是一位网友问起如何实现类似 MS Office 的 Excel 中数据选择功能。Excel 的数据选择如图 6.1 所示：

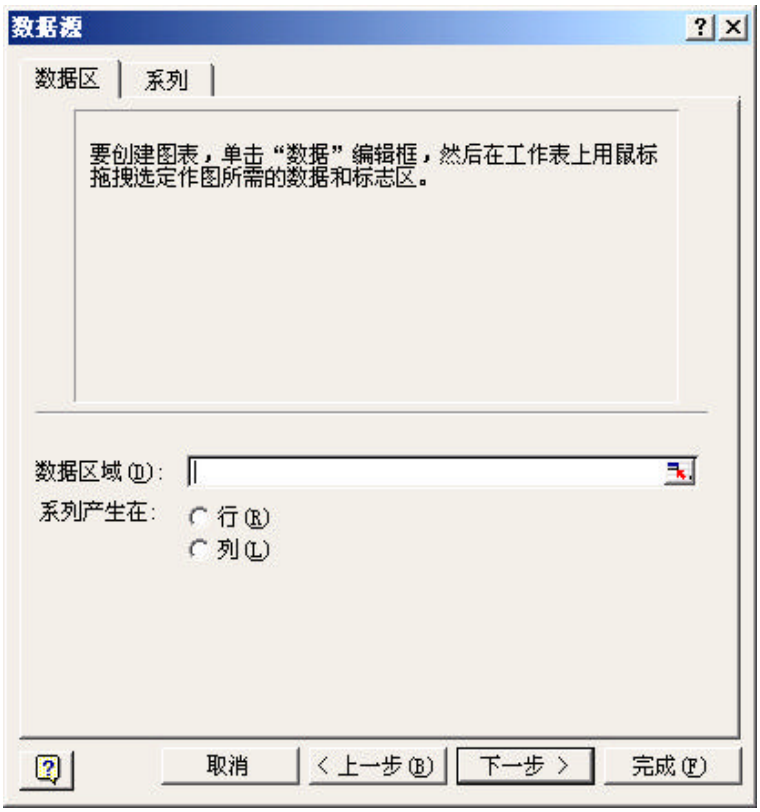


图 6.1 Excel 中数据拾取窗体

“数据区域”输入框是一个带有一个 Button 的 Edit 输入框，当鼠标点击了右边的 Button 时，整个窗口会缩小成下图的形状，再次点击 Button 时，窗体会复原。



图 6.2 Excel 中数据拾取窗体缩小状态

熟悉 C++ Builder 的读者都知道，要实现这样的功能非常简单，利用 Form 的 AutoSize，控件的 Visible、Align、Top、Left、Height 和 Width 等属性按照特定的顺序设置，同时在程序中需要保存原始的控件位置就可以完成。

但我们知道我们在 Excel 中这样的数据选择框至少有十多处，而且每一个窗体中数据选择框的数量、位置等都不一样，因此有多处使用到这个功能时，将会出现大量的重复代码。作一个非常简单的控件来替代重复的代码设计，是一个不错的选择。不仅如此，使用控件会让代码修改和维护也变得非常简单，使程序的结构也变得更加清晰。我们来看如何实现。

1.1 控件的分析

要实现这样的功能，最主要的就是这个带有 Button 的 Edit 选择框，我们将其命名为 TPickEdit，含义是用于拾取数据的，我们来分析如何实现。

- 从表面上看，这个 TPickEdit 在 Button 接受鼠标的 Ckick 事件后缩小，再次点击时复原，但实际上是反映了 TPickEdit 的两个状态：缩小化和复原状态。这个状态可以使用 bool 类型属性 Mined 来描述，Mined 为 true 时，父窗体 Form 为缩小化，当 Mined 为 false 时，父窗体 Form 复原。
- 除了点击 Button 的按钮可以使父窗体缩小和复原之外，允许其它条件改变 Mined 属性的状态，比如用户拾取数据后按“回车”键等，因此 Mined 属性应该是发布的属性，是 TPickEdit 的一个公共接口。
- 除了 Edit 本身的属性以外，TPickEdit 包含的 SpeedButton 应该允许用户指定部分属性，如 OnClick、Graphy、Cusor 等，但 Button 的大部分属性是不允许用户改变，比如 Left、Height、Top 等，这些属性是 TPickEdit 根据自身的状态来调整的，因此，不应该将 Button 整体作为 TSpeedButton 类型属性出现，而是将其中部分属性映射到 TPickEdit 的属性中。
- 除了上述特征之外，TPickEdit 还需要解决一些辅助性的功能，比如 Edit 框中字符显示的范围等。

正象前面所说的那样，这个控件其实并没有什么新的东西，只是将一些其它控件的某些状态联系在一起，共同构成一个新的属性 Mined。我们来看一下这个 Mined 属性和其它控件的属性具有怎么样的约束关系：

- Mined 属性对自身的影响并不多，更多的是和父窗体以及父窗体上的其它控件发生联系，这种关系是依靠 Parent 属性来建立的。
- Mined 为 true 时代表父窗体将缩小化，这个状态的变化是由 TPickEdit 造成的，因此 TPickEdit 必需有义务能够将原始状态记录下来，并且能够完全还原。

- 当 Mined 为 true 时，TPickEdit 的宽度和父窗体的 ClientRect 的宽度相等，高度不变，父窗体的 ClientRect 的高度和 TPickEdit 的高度相等，其它控件的 Visible 均为 false。这个过程的实现可以利用 Align 属性，AutoSize 属性，Left、Top、Witdh 和 Height 属性来实现，实际上就是和其它状态之间的关系。

1.2 源代码

接下来是整个 TPickEdit 的源代码，在本书所附的光盘中有全部内容，读者可以参考前面相关内容阅读并理解。

1.2.1 声明部分

PickEdit.h

```
#ifndef PickEditH
#define PickEditH
//---------------------------------------------------------------
#include <Buttons.hpp>
#include <Menus.hpp>
#include <Graphics.hpp>
#include <Forms.hpp>
#include <SysUtils.hpp>
#include <Messages.hpp>
#include <Controls.hpp>
#include <ExtCtrls.hpp>
#include <StdCtrls.hpp>
#include <Classes.hpp>
#include <Windows.hpp>
#include <System.hpp>
//---------------------------------------------------------------
class PACKAGE TPickEdit : public TCustomEdit
{
private:
    int FOrgX;              //原始位置、大小属性
    int FOrgY;
    int FOrgWidth;
    int FOrgHeight;
    int FOrgParentHeight;   //父窗体高度
    bool FMined;            //是否缩小化
```

```
protected:
    TSpeedButton * Button;  //添加一个按钮
    MESSAGE void __fastcall WMSize(TWMSize &Message);
                            //新的事件响应函数
    MESSAGE void __fastcall CMEnter(TWMNoParams &Message); //……
    void __fastcall BtnMouseDown(TObject *Sender, TMouseButton Button,
                             TShiftState Shift, int X, int Y );
                            //响应 Button 的 MouseDown 事件
    TNotifyEvent __fastcall GetButtonClick();                    //  A
    void __fastcall         SetButtonClick( TNotifyEvent Value );//  A
    TCursor __fastcall      GetCursor();                         //  B
    void __fastcall         SetCursor( TCursor Value );          //  B
    Graphics::TBitmap* __fastcall GetGlyph();                    //  C
    void __fastcall        SetGlyph( Graphics::TBitmap* Value ); //  C
/*************************************************************
    A、B、C 分别将 TSpeedButton 的 OnClick 事件、Cursor 属性和
    Glyph 属性映射传递给 新的控件 TPickEdit。
/*************************************************************/
    void __fastcall SetMined(bool Value); //设置 Mined 属性的函数
    void __fastcall SetButtonPosition();  //界面控制用
    void __fastcall BtnClick(TObject *Sender);
                                          //响应 Button 的 OnClick 事件
    void __fastcall CreateParams(TCreateParams &Params);
       //用于窗体建立时传递参数,
    void __fastcall SetEditBounder(void);//控制 Edit 中字符有效的显示范围
    void __fastcall CreateWnd(void);     //重载 该函数
    virtual void __fastcall SetEnabled(bool Value);
                                          //下面建立 Windows 消息影射表
    BEGIN_MESSAGE_MAP
      VCL_MESSAGE_HANDLER(WM_SIZE, TWMSize, WMSize);
      VCL_MESSAGE_HANDLER(CM_ENTER, TWMNoParams, CMEnter);
    END_MESSAGE_MAP(TCustomEdit);
public:
    TForm *ParaForm;                      //父窗体句柄
    TList *Other;
    __fastcall TPickEdit(TComponent* Owner);
__published:
    __property TCursor ButtonCur = {read = GetCursor,write = SetCursor };
          //将 TSpeedButton 的Cursor 属性传递给 PickEdit,
```

```
    __property TNotifyEvent OnButtonClick = {read = GetButtonClick,
                              write = SetButtonClick};
            // 同上，事件传递
    __property Graphics::TBitmap* ButtonGlyph =
                              {read=GetGlyph,write=SetGlyph};
            //同上
    __property bool Mined = {read = FMined,write = SetMined};
            //建立并发布 Mined 属性。
    __property Align;                  // 以下继承 发布 父类部分属性和事件
    __property AutoSize;
    __property Font;
    __property Color;
    __property Hint;
    __property Enabled;
    __property Text;
    __property OnClick;
    __property OnMouseDown;
    __property OnMouseMove;
    __property OnDblClick;
    __property OnKeyDown;
    __property OnKeyPress;
};
#endif
```

1.2.2 实现部分

.CPP 文件中实现部分

```
#include <vcl.h>
#pragma hdrstop
#include "PickEdit.h"
#pragma package(smart_init)
//---------------------------------------------------------------
static inline void ValidCtrCheck(TPickEdit *)
{
    new TPickEdit(NULL);
}
//---------------------------------------------------------------
__fastcall TPickEdit::TPickEdit(TComponent* Owner)
```

```
        : TCustomEdit(Owner)
{                    //构造函数，创建 Button 并初始化
    ParaForm = (TForm *)Owner;
    Button = new TSpeedButton( (TComponent *) this );
    SetButtonPosition();
    Button->Glyph->Handle = LoadBitmap(HInstance, "SpinDown");
    Button->OnMouseDown = BtnMouseDown;
    Button->OnClick = BtnClick;
    Button->Cursor = crHandPoint;
    Button->Parent = this;
}
//---------------------------------------------------------------
namespace Pickedit
{                    //注册控件
    void __fastcall PACKAGE Register()
    {
       TComponentClass classes[1] = {__classid(TPickEdit)};
       RegisterComponents("MyInterFace", classes, 0);
    }
}
//---------------------------------------------------------------

void __fastcall TPickEdit::SetEditBounder(void)
{                   //用于调整 TPickEdit 的字符显示区域
    TRect Loc;
    SendMessage(Handle, EM_GETRECT, 0, long(&Loc));
    Loc.Bottom = ClientHeight + 1;  //这是 Windows 的一个绘图 Bug，需要 + 1
    Loc.Right = ClientWidth - Button->Width;
    Loc.Top = 0;
    Loc.Left = 0;
    SendMessage(Handle, EM_SETRECTNP , 0, long(&Loc));
}
//---------------------------------------------------------------

void __fastcall TPickEdit::CreateWnd()
{                    //重载 CreateWnd() 函数
    TCustomEdit::CreateWnd();
    SetEditBounder();
}
```

```
//---------------------------------------------------------------

void __fastcall TPickEdit::CMEnter(TWMNoParams &Message)
{                   //CM_ENTER 消息的响应函数
   if (AutoSelect && !(ControlState.Contains(csLButtonDown)))
     SelectAll();
}
//---------------------------------------------------------------

void __fastcall TPickEdit::WMSize(TWMSize &Message)
{                   //WM_SIZE 消息的响应函数
   SetButtonPosition();   //重新 Button 的位置、大小
   SetEditBounder();      //以及 Edit 框的字符显示宽度
   return;
}
//---------------------------------------------------------------

void __fastcall TPickEdit::CreateParams(TCreateParams &Params)
{                   //创建窗体的参数；设置控件创建时的参数，至少应该设置
   TCustomEdit::CreateParams(Params);  // ES_MULTILINE，否则，
   Params.Style &= ~WS_BORDER;          //无法限制 Edit 框中字符
   Params.Style |=  ES_MULTILINE | WS_CLIPCHILDREN;   //的显示宽度
   Params.Height = 21;
}
//---------------------------------------------------------------

void __fastcall TPickEdit::BtnMouseDown(TObject *Sender,
           TMouseButton Button,TShiftState Shift, int X, int Y)
{
   //该函数预留为空
}
//---------------------------------------------------------------

void __fastcall TPickEdit::BtnClick(TObject *Sender)
{
   Mined = !Mined;
}
//---------------------------------------------------------------
```

```
TCursor __fastcall TPickEdit::GetCursor()
{
    return Button->Cursor;
}
//---------------------------------------------------------------

void __fastcall TPickEdit::SetCursor( TCursor Value )
{
    Button->Cursor = Value;
}
//---------------------------------------------------------------

TNotifyEvent __fastcall TPickEdit::GetButtonClick()
{
    return Button->OnClick;
}
//---------------------------------------------------------------

void __fastcall TPickEdit::SetButtonClick( TNotifyEvent Value )
{
    Button->OnClick = Value;
}
//---------------------------------------------------------------

void __fastcall TPickEdit::SetMined(bool Value)
{                    //Mined 属性的设置函数
    int I;
    if(!Other)
       Other = new TList;
    I = ParaForm->ControlCount;              //获得父窗体控件的数量
    Other->Count = I;
    if( FMined && !Value )                   //已经缩小化的情况
    {
       Align = alNone;                       //恢复对齐方式
       ParaForm->AutoSize = false;           //恢复父窗体属性
       Parent->Height = FOrgParentHeight;    //同上
       Width = FOrgWidth;                    //恢复 位置属性
       Height = FOrgHeight;                  //同上
       Top = FOrgY;                          //同上
```

```
        Left = FOrgX;                          //同上
        FMined = Value;                        //保存状态
        for( ;I > 0; I-- )
        {                                      //恢复父窗体其他控件的可视性
          ParaForm->Controls[I - 1]->Visible = bool(Other->Items[I - 1]);
        }
      }
      else if( Value )                         // 反之
      {
        for( ;I > 0; I-- )
        {                                      //隐藏父窗体所有控件
          Other->Items[I - ] = (void*)(ParaForm->Controls[I - ]->Visible);
          ParaForm->Controls[I - ]->Visible = false;
        }
        Visible = true;                        //显示自身
        FOrgParentHeight = Parent->Height ;    //备份父窗体高度属性
        FOrgWidth = Width;                     //备份自身位置属性
        FOrgHeight = Height;                   //同上
        FOrgY = Top;                           //同上
        FOrgX = Left;                          //同上
        Align = alTop;                         //向上靠近缩紧
        FMined = Value;                        //保存状态
        ParaForm->AutoSize = true;             //设置父窗体自动调整大小
      }
}
//---------------------------------------------------------------

void __fastcall TPickEdit::SetButtonPosition()
{                  //当控件大小改变时正确设置 Button 的位置
     //  this->Height = 21;   用于限制 TPickEdit 的高度为 21，这里没有限制
    Button->Top = 0;
    Button->Width = 21;
    Button->Left = this->Width - Button->Width - 2;      //留了边框的距离
    Button->Height = this->Height - 2;                   //留了边框的距离
}
//---------------------------------------------------------------

Graphics::TBitmap* __fastcall TPickEdit::GetGlyph()
{
```

```
    return Button->Glyph;
}
//---------------------------------------------------------------

void __fastcall TPickEdit::SetGlyph( Graphics::TBitmap* Value )
{
    Button->Glyph = Value;
}
//---------------------------------------------------------------

void __fastcall TPickEdit::SetEnabled(bool Value)
{                   //重载SetEnabled方法，将Enabled属性传递到Button中
    TCustomEdit::SetEnabled(Value);
    if( Button )
       Button->Enabled = Value;
}
```

1.3 控件的详解

在这个例子中，大部分函数不需要作解释，

ValidCtrCheck Register 方法是控件必需的，而且是由 IDE 自动生成的；

GetGlyph、SetGlyph、GetCursor、SetCursor、GetButtonClick、SetButtonClick 等都是属性的读取函数，也非常简单；

SetEnabled 是重载了父类的 SetEnabled 函数，目的在于能将 Enabled 属性传递到 Button 中；

SetButtonPosition 是用于调整 Button 的位置及大小的方法，使 Button 在 TPickEdit 的大小改变时能够及时适应，因此在 TPickEdit 中必需有 WM_SIZE 消息的响应函数 WMSize；

BtnMouseDown、BtnClick 都是为 Button 的事件而创建的方法，其中 BtnMouseDown 预留，BtnClick 为 SpeedButton 的 OnCkick 处理函数；

TPickEdit 构造过程需要解决以下几个问题，创建 Edit 中的 Button 按钮，设置 OnClick 事件句柄、图片、光标等属性，并且设置 Parent 属性。TPickEdit 并没有显式的声明析构函数，也就是并没有显式的释放 Button，这并不是一个 Bug，读者可能注意到了，在构造函数中创建 TSpeedButton 的时候，指定了 this 作为 Button 的 Owner，前面讲过 Owner 的作用就是负责释放自己所拥有的组件，因此当 TPickEdit 的缺省析构函数被调用时，Button 会自动被释放。在后面的例子 TDataPick 中有不同的情况，请读者注意。

SetEditBounder 并不是一个属性的写函数，而是用于设置 Edit 中字符显示范围的函数。要设置字符的显示范围需要一个条件 :即该 Edit 类必需是 ES_MULTILINE 风格，TEdit 并非如此，属于 ES_SIGLELINE 风格，而且 TEdit 已经将大部分父类继承来的属性发布

了,而这些属性并不一定是我们 TPickEdit 需要发布的,因此 TPickEdit 没有从 TEdit 派生，而是从 TCustomEdit 直接派生而来，同时还需要重载 CreateParams 方法来改变窗体类的风格。另外,和大多数 TWinControl 派生类一样,TPickEdit 重载了 CreateWnd 方法，其目的只有一个，就是负责在创建 TPickEdit 窗体时改变字符的显示范围，尽管在 WM_SIZE 消息的响应函数中有了显示范围设置的功能，但是并不能将这个任务完全寄希望于用户设计时改变缺省的大小,应该在 PickEdit 窗体创建的时候就进行字符显示范围的设置。

CMEnter 是 CM_ENTER 消息的响应函数，这是在 TPickEdit 获得焦点的时候自动的选择了全部字符，但不包含使用鼠标点击而获得焦点。

SetMined 方法是 TPickEdit 的一个重要设置方法，它分为两个部分，一是设置到缩小化状态，另一种是复原状态。在 TPickEdit 中使用了 TList 类来保存每一个父窗体中 Child 控件的 Visible 属性（并不一定是所有的控件都是运行时可见，因此需要对 Visible 逐一保存），父窗体的 Height、自身的原始位置大小等，由 TPickEdit 的私有成员保存,父窗体的 Width 由 TPickEdit 的新 Width 保存 TPickEdit 的 Align 属性为 alTop 时，父窗体的 Width 不会受到父窗体 AutoSize 属性改变的影响，但这必需保证 Parent 的 AutoSize 改变为 **true** 之前,TPickEdit 的 Align 必需已经是 alTop，相反的 TPickEdit 的 Align 属性恢复 alNone 之前，必需将 Parent 的 AutoSize 恢复为 **false**；

至此 TPickEdit 已经是一个可以使用和发布的控件了，但是还有一个小小的问题：我们在 Button 的 OnClick 事件中改变了 Mined 属性的状态，同时又将 Button 的 OnClick 事件映射到了 TPickEdit 的 OnButtonClick 事件中，这种直接映射会导致用户定义了 OnButtonClick 事件的处理句柄时，我们在控件内部定义的 BtnClick 方法将不会得到执行，也即失去了点击 Button 而改变 Mined 状态的能力，尽管用户仍然可以在 OnButtonClick 事件的处理句柄中改变 Mined 的状态，但这毕竟不太妥当，我们对这个 Bug 作一个修复：

- 保留用户可以自定义 OnButtonClick 事件。
- 用户定义了 OnButtonClick 事件后，点击 Button 仍然可以改变 Mined 属性的状态。
- 在用户定义 OnButtonClick 事件的处理句柄中，允许用户屏蔽上面的功能。

我们看看该如何来作。

- 首先是不能再采用直接映射的方式，创建 OnButtonClick 事件，需要去掉 OnButtonClick 的读取方法 GetButtonClick,建立一个 FOnButtonClick 私有数据用来存储用户的 OnButtonClick 句柄。
- 在 BtnClick 方法中调用 FOnButtonClick 句柄。
- 为了能够让用户屏蔽掉点击 Button 改变 Mined 属性的功能，需要创建一个新的 **closure** 类型 TNotifyHandledEvent，该 **closure** 可以向用户的句柄传递一个 **bool** 类型的引用参数 Handled，BtnClick 根据 Handled 参数的值来判断是否进行 Mined 属性的设置。

完成以上功能，代码需要做如下改动：

定义新的事件类型：

```
typedef void __fastcall (__closure *TNotifyHandledEvent)( System::TObject*
              Sender,bool &Handled);
```

在类的声明部分增加 FOnButtonClick 私有成员：

```
class PACKAGE TPickEdit : public TCustomEdit
{
private:
        //.....
       TNotifyHandledEvent FOnButtonClick;
}
```

修改 OnButtonClick 属性为：

```
__property TNotifyHandledEvent OnButtonClick = {read = FOnButtonClick,
                      write = SetButtonClick};
```

修改 SetButtonClick 方法为：

```
void __fastcall TPickEdit::SetButtonClick( TNotifyHandledEvent Value )
{
    FOnButtonClick = Value;
}
```

修改 BtnClick 方法为：

```
void __fastcall TPickEdit::BtnClick(TObject *Sender)
{
    bool Handled = false;
    if( FOnButtonClick )
       FOnButtonClick( Sender,Handled );
    if( !Handled )
       Mined = !Mined;
}
```

删去 GetButtonClick 方法；

从这里我们可以看出，对于 C++ Builder 而言，修改和扩充一段代码的内容是非常

方便的，这也是面向状态的一个最大的优点。

在这一节中，我们通过了对 TPickEdit 控件的学习，已经了解了属性的创建、属性的发布；事件的创建、事件的触发、自定义事件；允许用户屏蔽控件的却省动作；窗体控件基本设置；对继承属性的发布；通过消息映射表对 Windows 消息的响应；通过重载父类虚拟函数对父类已经处理的 Windows 消息的响应；子控件属性和事件向父控件属性和事件的映射等内容。在下一节将要介绍的是关于图形应用的控件设计。

2 图形控件 TDataPick

这一节将要介绍的控件 TDataPick 是一个用于图形显示方面的控件,在这个控件里，我们采用了从 TCustomControl 派生，TCustomControl 是从 TWinControl 派生的窗体控件，具有窗口句柄，能够接收 Windows 的键盘消息，并且提供了 Canvas 属性和 Paint 方法，允许用户对其画布进行绘图方面的处理，下图是 TDataPick 控件运行后的

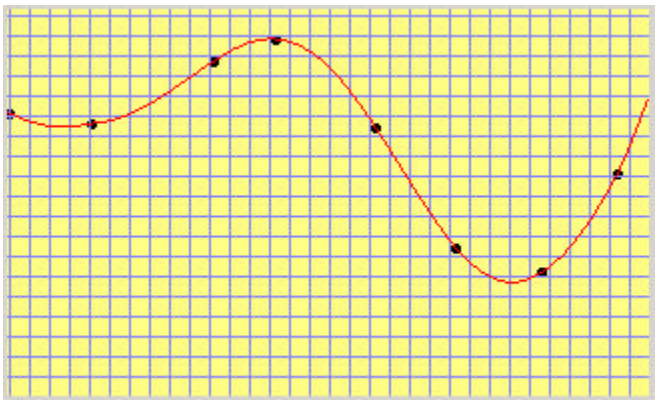


图 6.3 TDataPick 控件运行后的图形

屏幕拷贝。

这个形状可能读者都不会陌生，在很多优秀的软件中都有类似的应用，比如 Adobe 的 PhotoShop 中就有很多关于曲线数据选择，和 TDataPick 控件比较相似，还有著名的声音处理软件 CoolEdit 也是多次使用了类似的控件，诸如此类的应用还非常的多。

很多读者可能都知道，利用 C++ Builder 来实现这样的功能并不复杂，但是在一个程序或者还希望在其它的程序中使用，则编制成一个控件使用是非常不错的。我们来看如何在 C++ Builder 中如何制作这样一个控件。

2.1 控件的分析

这个控件的主要用途是将用户提供的一些属性控制点（图中的黑色园点），经过某种数学的拟合运算，转换成相应的曲线数据，并将拟合后的数据反映到绘图界面上。

在 TDataPick 中，属性控制点记录着用户提供的原始点位信息，并将属性控制点显示在绘图界面上，使用绘图函数根据 TDataPick 的属性控制点的数据将其显示出来就可

以达到希望的效果,这是常规的 SKD 编程中经常使用的方法,但从这个控件的特点来分析,并不合理,因为属性控制点除了纪录着 TDataPick 的点位数据之外,还有一些自己的特性,比如用户希望鼠标在经过点位的时候显示不同的 Cursor;或者用户希望可以通过鼠标的操作,来拖动属性控制点改变位置,使用 TDataPick 的鼠标消息,根据属性控制点的数据判断也可以完成这样的功能,但是从其结构上看,属性控制点显然表达为一个新的对象更为合理一些,这个对象恰好可以利用从 TGraphicControl 派生的图形控件 TShape 控件。因此,在 TDataPick 控件中包含了一个新的属性类型:TPointData。

在 TDataPick 控件中,Point(TPointData 的实例)的数量并不是固定的,而是根据用户的需要增加或者删除,因此在 TDataPick 中必须考虑对 Point 的增加、删除以及控制等能力。

这是对 TDataPick 控件的简单的分析,是该控件的基本功能。除此之外,TDataPick 还需要考虑的是其它一些可能需要允许用户进行控制或者修改的特征,比如,曲线的颜色、背景的颜色、属性控制点的颜色,属性控制点是否显示被选中、是否显示网格、网格的颜色等等用以适合不同用途的特征。

在我们这个例子中,将完成以下的功能:

- 允许用户指定曲线、属性控制点、网格、背景、选中后的属性控制点等特征的颜色;
- 允许用户指定是否绘制网格,允许用户指定是否显示属性控制点;
- 允许用户改变网格的间距;
- 允许用户指定属性控制点的鼠标 Cursor
- 允许用户点击鼠标左键,在鼠标位置创建新的 Point;
- 允许用户用右键点击属性控制点将其删除;
- 允许用户使用鼠标拖动属性控制点改变其位置;
- 允许用户在设计时使用鼠标控制起始点和终点的位置;

2.2 源代码及注解

2.2.1 声明部分

以下是在 DataPick.h 中的声明部分:

```
//---------------------------------------------------------------------
#ifndef DataPickH
#define DataPickH
//---------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>

class PACKAGE TPointData : public TShape
```

```
{
private:
    int x ;                          //纪录鼠标按下时的位置坐标
    int y ;
    bool IsMouseDown;                //纪录鼠标是否被按下
    bool FSelected;                  //Selected 属性的存储变量
protected:                           //Selected 的设置函数
    void __fastcall SetSelected(bool Value);
public:                              //声明 Selected 属性
    __property bool Selected = {read=FSelected,write=SetSelected};
    __fastcall TPointData( TComponent* Owner );  //构造函数
    __fastcall TPointData::TPointData( TComponent* Owner,int X,int Y );
    virtual void __fastcall Paint(void);      //重载绘图函数 Paint
        //下面三个函数为重载 TControl 的动态函数来实现鼠标消息响应的
    DYNAMIC void __fastcall MouseMove(TShiftState Shift,
                              int X, int Y);
    DYNAMIC void __fastcall MouseDown(TMouseButton Button,
                              TShiftState Shift, int X, int Y);
    DYNAMIC void __fastcall MouseUp(TMouseButton Button,
                              TShiftState Shift, int X, int Y);
} ;
//---------------------------------------------------------------------------

class PACKAGE TDataPick : public TCustomControl
{
private:
    bool     FShowPoint;     //下面声明了属性的内部存储变量
    TColor   FForeColor;
    TColor   FPointColor;
    TColor   FGridColor;
    TColor   FSelectColor;
    bool     FGrided;
    TCursor  FPointCur;
    unsigned short FGridInterval;
                             //供内部使用的 TPointData 类型的数据变量
    TPointData    *Point;
protected:                   //以下时属性的设置函数
    void __fastcall  FSetFGridInterval(unsigned short Value);
    void __fastcall  FSetShowPoint(bool Value);
```

```
    void __fastcall  FSetForeColor(TColor Value);
    void __fastcall  FSetPointColor(TColor Value);
    void __fastcall  FSetGridColor(TColor Value);
    void __fastcall  FSetGrided(bool Value);
    void __fastcall  FSetPointCur(TCursor Value);
    void __fastcall  FSetSelectColor(TColor Value);
    int  __fastcall  FGetPoint(int Index);       //带有索引的读取函数
    void __fastcall  FSetPoint(int Index,int Value);//带有索引的设置函数
public:
    int              DataCount;
    TPointData     * FStartPoint;        //起始点位的内部存储变量
    TPointData     * FEndPoint;          //终点的内部存储变量
    TPointData     * Selection;          //纪录当前被选择中的Point
    double         * DataX;              //用于拟合的数据数组（动态分配内存）
    double         * DataY;              //用于拟合的数据数组（动态分配内存）
    int              LeftLimit;          //纪录被选中Point移动位置的左边界
    int              RightLimit;         //纪录被选中Point移动位置的右边界
    void __fastcall  WMMouse(TWMMouse Msg);  //鼠标的响应函数
    __fastcall       TDataPick(TComponent* Owner);
    __fastcall       ~TDataPick(void );
    virtual void __fastcall Paint(void);     //重载的Paint函数
__published:
    __property   OnEnter;                    //继承发布父类的属性
    __property   OnExit;
    __property   OnKeyDown;
    __property   OnKeyPress;
    __property   OnKeyUp;
    __property   Color;                      //以下是自定义的属性
    __property int StartPointX = {read=FGetPoint,write=
                            FSetPoint,index=1};
    __property int StartPointY = {read=FGetPoint,write=
                            FSetPoint,index=2};
    __property int EndPointX = {read=FGetPoint,write=
                            FSetPoint,index=3};
    __property int EndPointY = {read=FGetPoint,write=
                            FSetPoint,index=4};
      //以上四个属性记录了起始点和终点的位置坐标,以便设计时指定的位置可以
      //被存储在资源中,最终反映在运行时
    __property TCursor PointCur = {read=FPointCur,write=FSetPointCur};
```

```
    __property unsigned short GridInterval = {read = FGridInterval,write=
                                FSetFGridInterval,default=10};
    __property bool   ShowPoint = {read = FShowPoint,write =
                                    FSetShowPoint,default=true};
    __property TColor ForeColor = {read = FForeColor,write =
                                    FSetForeColor,default=clBlack};
    __property TColor PointColor = {read = FPointColor,write =
                                    FSetPointColor,default=clBlack};
    __property TColor SelectColor = {read = FSelectColor,write =
                                    FSetSelectColor,default=clYellow};
    __property TColor GridColor = {read = FGridColor,write =
                                    FSetGridColor,default=clSilver};
    __property bool    Grided = {read = FGrided,write =
                                    FSetGrided,default=false};
    BEGIN_MESSAGE_MAP   //对于鼠标消息采用了消息映射表的方式实现响应
      VCL_MESSAGE_HANDLER(WM_LBUTTONDOWN, TWMMouse, WMMouse);
    END_MESSAGE_MAP(TCustomControl);
};
//---------------------------------------------------------------------
#endif
```

2.2.2 实现部分代码

DataPick.cpp 文件的全部代码及部分注解：

```
#include <vcl.h>
#pragma hdrstop
#include "DataPick.h"
#pragma package(smart_init)

double Enlgr(double DataX[],double DataY[],int DataCount,double ResultX)
{           //用于数据拟合的数学处理函数,参考相关 C 语言算法的书籍
    int i,j,k,m;              //DataX 为采样点 X 轴坐标,DataY 为 Y 轴坐标
    double Result,s;          //DataCount 为采样点的数据数量
    Result = 0;               //ResultX 为需要插值的 X 轴坐标
    if( DataCount < 1 )       //返回值为插值
       return(Result);
    if( DataCount == 1 )
    {
```

```
        Result = DataY[0];
        return(Result);
    }
    if( DataCount == 2 )
    {
        Result=(DataY[0]*(ResultX-DataX[1])-DataY[1]*(ResultX-DataX[0]))/
                        (DataX[0]-DataX[1]);
        return(Result);
    }
    i = 0;
    while((DataX[i] < ResultX) && (i < DataCount))
        i = i + 1;
    k = i - 4;
    if (k < 0)
        k = 0;
    m = i + 3;
    if ( m > DataCount - 1 )
        m = DataCount - 1;
    for( i = k;i <= m; i++ )
    {
        s=1.0;
        for(j = k;j <= m; j++)
          if(j != i)
             s = s * (ResultX - DataX[j]) / (DataX[i] - DataX[j]);
          Result=Result+s*DataY[i];
    }
    return Result;
}
//---------------------------------------------------------------------

static inline void ValidCtrCheck(TDataPick *)
{
    new TDataPick(NULL);
}
//---------------------------------------------------------------------

__fastcall TDataPick::TDataPick(TComponent* Owner)
                      : TCustomControl(Owner)
{               //TDataPick 的构造函数
```

```
    Color = clWhite;     //创建并初始化起始点和终点
    Width = 185;
    Height = 105;
    LeftLimit = 0;
    RightLimit = Width;
    FShowPoint = true;
    FStartPoint = new TPointData(NULL,1,Height/2);
      //TPointData 的 Owner 为空将使 TPointData 在设计时能够独立响应鼠标消息
    FStartPoint->Parent = this;     //但是必须由析构函数负责释放
    FEndPoint = new TPointData(NULL,Width,Height/2);
    FEndPoint->Parent = this;
    FGridInterval = 10;
    FGridColor = clSilver;
    FForeColor = clBlack;
    FPointColor = clBlack;
    FPointCur = crHandPoint;
}
//---------------------------------------------------------------------

__fastcall TDataPick::~TDataPick( )
{
    delete FStartPoint;
    delete FEndPoint;
    //TDataPick 的析构函数,负责清除起始点和终点
}
//---------------------------------------------------------------------

namespace Datapick
{
    void __fastcall PACKAGE Register()
    {
        TComponentClass classes[1] = {__classid(TDataPick)};
        RegisterComponents("MyInterFace", classes, 0);
    }
}
//---------------------------------------------------------------------

void __fastcall TPointData::SetSelected(bool Value)
{             //TPointData 的 Selected 属性设置函数
```

```
    if( FSelected != Value )
    {
       if( Value )
       {
         if(((TDataPick *)Parent)->Selection )
           ((TDataPick *)Parent)->Selection->Selected = false;
         ((TDataPick *)Parent)->Selection = this;
       }
       FSelected = Value;   //设置自身的同时必须更新TDataPick的Selection属性
       Paint();
    }
}
//---------------------------------------------------------------------

void __fastcall TPointData::MouseMove(TShiftState Shift,
     int X, int Y)
{         //TPointData的鼠标移动消息的响应函数
    if( IsMouseDown )
    {     //根据鼠标是否被点下来判断是否移动属性控制点
       this->Left = this->Left + X - x;
       this->Top = this->Top + Y - y;
       //判断是否超出边界限定,
       if( Left <= ((TDataPick*)Parent)->LeftLimit )
           Left = ((TDataPick*)Parent)->LeftLimit + 1;
       if( Left >= ((TDataPick*)Parent)->RightLimit )
           Left = ((TDataPick*)Parent)->RightLimit - 1;
       ((TDataPick*)Parent)->Paint();
    }
}
//---------------------------------------------------------------------

void __fastcall TPointData::MouseDown(TMouseButton Button,
                              TShiftState Shift, int X, int Y)
{
    //TPointData的鼠标按下的响应函数
    ((TDataPick*)Parent)->RightLimit = ((TDataPick*)Parent)->Width;
    ((TDataPick*)Parent)->LeftLimit = 0;
    if( Button == mbRight )
    {  //右键则删除该属性点
```

```
        TDataPick * Temp = (TDataPick*)Parent; //纪录 Parent
        if(this != Temp->FStartPoint && this != Temp->FEndPoint)
        {
          delete this;    //销毁自身
          Temp->Paint();  //更新 Parent 窗体
        }
      }
      else
      { //若是左键按下则增加新的属性控制点位
        IsMouseDown = true;  //纪录鼠标是否被按下
        x = X;
        y = Y;              //纪录鼠标按下的位置
        for(int i = 0; i < Parent->ControlCount; i++)
        {       //对鼠标 Point 移动的 X 轴做出限制
          if( Parent->Controls[i]->Left < Left )
          {
            if( Parent->Controls[i]->Left >
                          ((TDataPick*)Parent)->LeftLimit)
              ((TDataPick*)Parent)->LeftLimit =
                          Parent->Controls[i]->Left;
          }
          else if( Parent->Controls[i]->Left > Left )
          {
             if( Parent->Controls[i]->Left <
                          ((TDataPick*)Parent)->RightLimit)
               ((TDataPick*)Parent)->RightLimit =
                           Parent->Controls[i]->Left;
          }    //更改被选中的 Point
        }
        Selected = true;
      }
    }
    //---------------------------------------------------------------------

    void __fastcall TPointData::MouseUp(TMouseButton Button,
                                TShiftState Shift, int X, int Y)
    {
      IsMouseDown = false;  //鼠标按键释放消息的响应函数
      ((TDataPick*)Parent)->Paint();
```

```
}
//---------------------------------------------------------------------

__fastcall TPointData::TPointData( TComponent* Owner )
   : TShape(Owner)
{         //TPointData 的构造函数
    Width = 5;
    Height = 5;
    IsMouseDown = false;
}
//---------------------------------------------------------------------

__fastcall TPointData::TPointData( TComponent* Owner,int X,int Y )
                        :TShape(Owner)
{       //TPointData 带有相关信息的构造函数
    Width = 5;
    Height = 5;
    x = 2;
    y = 2;
    Left = X - 2;
    Top = Y - 2;
    Cursor = crHandPoint;
    IsMouseDown = false;
}
//---------------------------------------------------------------------

void __fastcall TPointData::Paint(void)
{      //TPointData 的绘图函数,用于显示 Point,重载父类的函数
    if( FSelected )
    {
       Shape = stRectangle;
       Pen->Color = ((TDataPick *)Parent)->SelectColor;
    }
    else
    {
       Pen->Color = ((TDataPick *)Parent)->PointColor;
       Shape = stCircle;
    }
    Brush->Color = ((TDataPick *)Parent)->PointColor;
```

```
    TShape::Paint();  //调用父类的绘图函数
}
//---------------------------------------------------------------------

void __fastcall TDataPick::WMMouse(TWMMouse Msg)
{         //TDataPick 的鼠标按下消息的响应函数,通过消息映射表实现
    if( Msg.Keys == MK_LBUTTON )
    {     //左键则创建新的 TPointData 实例
       Point = new TPointData(this,Msg.XPos,Msg.YPos);
       Point->Parent = this;
       Point->Cursor = FPointCur;
       Point->Brush->Color = PointColor;
       Point->Visible = ShowPoint;
       Point->Selected = true;
       if( Selection )
          Selection->Selected = false;
       Selection = Point;
       Paint();
    }
}
//---------------------------------------------------------------------

void __fastcall TDataPick::Paint(void)
{   //TDataPick 的绘图函数,重载父类函数
    int i;       //临时变量
    int TPos;    //临时变量
    int Pos;     //临时变量
    Graphics::TBitmap * BmpTemp = new Graphics::TBitmap;
    BmpTemp->Width = Width;  //使用幕后位图进行绘图
    BmpTemp->Height = Height;
    BmpTemp->Canvas->Brush->Color = Color;
    BmpTemp->Canvas->FillRect(ClientRect); //填充背景
    if(FGrided)
    {  //绘制网格的代码
       BmpTemp->Canvas->Pen->Color = FGridColor;
       for(i = Height;i > 0 ;i -= FGridInterval)
       {  //绘制水平网格
          BmpTemp->Canvas->MoveTo(0,i);
          BmpTemp->Canvas->LineTo(Width,i);
```

```
    }
    for(i = Width;i > 0 ;i -= FGridInterval)
    {  //绘制竖直网格
       BmpTemp->Canvas->MoveTo(i,0);
       BmpTemp->Canvas->LineTo(i,Height);
    }
}  //以下是将屏幕显示的 TPointData 的坐标转换成为用于拟合插值的数据
DataCount = ControlCount;
DataX = new double[DataCount];
DataY = new double[DataCount];
for(i = 0; i< DataCount;i++)
{
   if( i >= 0 || i == DataCount - 1)
   {   //TPointData 的大小为 5，属性点的中心向右、下方各偏移两个像素
     DataX[i] = Controls[i]->Left + 2;
     DataY[i] = Controls[i]->Top + 2;
   }
}
double Temp;  //以下部分是排序，按照 X 的升序排列
for( Pos = 0; Pos< DataCount - 1 ; Pos++)
{
   TPos = Pos;
   for( i = Pos + 1; i < DataCount; i++)
   {
      if( DataX[i] < DataX[TPos] )
         TPos = i;
   }    //交换数据
   Temp = DataX[ TPos ];
   DataX[ TPos ] = DataX[ Pos ];
   DataX[ Pos ] = Temp;     //交换数据
   Temp = DataY[ TPos ];
   DataY[ TPos ] = DataY[ Pos ];
   DataY[ Pos ] = Temp;
}   //以下是绘制曲线
BmpTemp->Canvas->Pen->Color = ForeColor;
BmpTemp->Canvas->MoveTo( 0,Enlgr( DataX,DataY,DataCount,0 ));
for( i = 1; i < Width; i++ )
{       //Temp 为计算出插值
   Temp = Enlgr( DataX,DataY,DataCount,i );
```

```
      BmpTemp->Canvas->LineTo( i,Temp );
    }       //将窗体上的控件也绘制幕后位图上
    PaintControls(BmpTemp->Canvas->Handle,0);
    TCustomControl::Paint();    //调用父类的绘图函数
    Canvas->Draw(0,0,BmpTemp); //将幕后位图的内容反映到画布上
    delete BmpTemp;  //销毁幕后位图
    delete DataX;    //清除临时变量,双精度数组
    delete DataY;    //清除临时变量,双精度数组
}

//---------------------------------------------------------------------
void __fastcall TDataPick::FSetShowPoint(bool Value)
{    //ShowPoint 属性的设置函数
    if(FShowPoint == Value)
      return;
    FShowPoint = Value;
    Paint();
}
//---------------------------------------------------------------------

void __fastcall TDataPick::FSetForeColor(TColor Value)
{   //曲线颜色 ForeColor 的设置函数
    if(FForeColor == Value)
      return;
    FForeColor = Value;
    Paint();
}
//---------------------------------------------------------------------

void __fastcall TDataPick::FSetPointColor(TColor Value)
{   //属性控制点颜色 PointColor 属性的设置函数
    if(FPointColor == Value)
      return;
    FPointColor = Value;
    Paint();
}
//---------------------------------------------------------------------

void __fastcall TDataPick::FSetGridColor(TColor Value)
```

```
{   //网格颜色 GridColor 属性的设置函数
    if(FGridColor == Value)
      return;
    FGridColor = Value;
    Paint();
}
//---------------------------------------------------------------------

void __fastcall TDataPick::FSetGrided(bool Value)
{   //是否显示网格的属性 Grided 的设置函数
    if(FGrided == Value )
      return;
    FGrided = Value;
    Paint();
}
//---------------------------------------------------------------------

void __fastcall TDataPick::FSetFGridInterval(unsigned short Value)
{   //网格间距的设置函数
    if(FGridInterval == Value || !Value)
      return;
    FGridInterval = Value;
    Paint();
}
//---------------------------------------------------------------------

void __fastcall TDataPick::FSetPointCur(TCursor Value)
{   //属性控制点鼠标 PointCur 的设置函数
    if(FPointCur == Value)
      return;
    else
    {
       FPointCur = Value;
       for(int i = 0; i < ControlCount; i++)
       {  //逐一改变每个子控件的鼠标
          Controls[i]->Cursor = FPointCur;
       }
    }
}
```

```
//---------------------------------------------------------------------

void __fastcall  TDataPick::FSetSelectColor(TColor Value)
{   //选中的属性点的边框颜色的设置函数
    if( FSelectColor != Value )
    {
       FSelectColor = Value;
       Paint();
    }
}
//---------------------------------------------------------------------

int __fastcall  TDataPick::FGetPoint(int Index)
{   //用于保存起始点和终点坐标的属性的读取函数
    if(Index == 1)
       return FStartPoint->Left + 2;
    if(Index == 2)
       return FStartPoint->Top + 2;
    if(Index == 3)
       return FEndPoint->Left + 2;
    if(Index == 4)
       return FEndPoint->Top + 2;
}
//---------------------------------------------------------------------

void __fastcall  TDataPick::FSetPoint(int Index,int Value)
{   //用于保存起始点和终点坐标的属性的设置函数
    if(Index == 1)
       FStartPoint->Left = Value - 2;
    if(Index == 2)
       FStartPoint->Top = Value - 2;
    if(Index == 3)
       FEndPoint->Left = Value - 2;
    if(Index == 4)
       FEndPoint->Top = Value - 2;
    Paint();
}
```

2.3 控件的详解

在这个控件中，我们使用了数据处理函数 Enlgr，这是一个根据给定点位坐标，进行插值计算的函数，有关算法的详细信息可以参考相关算法的书籍，这里只需要知道这是一个函数曲线的拟合插值函数，而不是一个任意二维曲线的拟合函数，也就是所给定的参考点，在沿着曲线运动时，其 X 坐标必须是单调递增或者单调递减的，曲线应该是一个以 X 为自变量的单值函数。因此，在曲线的属性控制点中不允许有 X 坐标相等的点，并且在属性控制点移动的时候，我们需要限定左右边界。

ValidCtrCheck 和 Register 是控件必需的函数，这里也不罗嗦。在这个单元里包含了两个类，一个是 TPointData，另一个是 TDataPick，我们分别介绍。

2.3.1 TPointData

这是用于将属性控制点显示在图形中，并使其能够自己响应鼠标消息，因此使用了从 TGraphicControl 的派生类 TShape 作为其基类。TPointData 并不单独出现，而总是作为 TDataPick 的 Child 出现。

TPointData 除了自身原有的属性之外，我们只创建了一个用于记录是否被选定的属性 Selected，是一个 bool 类型的可读写属性，当被选中时，其边框颜色由 Parent 的 SelectColor 属性决定，外形为正方形，当未被选中的时候，外形为圆形，边框和 Point 颜色相同，由 Parent 的 PointColor 决定。TPointData 的 Selected 属性和其他 Point 对象是相互排斥的关系，即同时只能有一个 Point 被选中，对于 Selected 属性的排斥关系可以使用 Parent 的 Broadcast 方法来实现，在本案中，由于存在对各个 Point 逐一筛选确定 Point 的左右边界，因此没有使用 Broadcast 方法广播消息，而是在对每个 Point 筛选时设置其 Selected 属性。SetSelected 方法为 Selected 属性的设置函数，这个方法唯一需要注意的是要避免出现逻辑上的循环调用，尽管 C++语言允许函数递归调用，但是没有出口的递归调用最终会导致堆栈的溢出而出现致命的错误，读者参考 SetSelected 方法源代码自行分析。

MouseDown、MouseMove 和 MouseUP 这三个方法是继承父类的虚拟方法，同时也是响应 TPointData 鼠标消息的方法，这三个方法属于“动态”函数，不同于一般的虚拟函数，“动态”函数在虚拟函数列表的结构和搜索方法上和一般的虚拟函数不同，因此在这三个方法中没有调用父类的同名方法，但同样会触发 OnMouseDown 等事件，若调用了父类的同名方法则会导致循环调用最终堆栈溢出。

MouseDown的主要作用是在鼠标单击的时候选中该 Point(左键)或者删除该Point（右键），并且记录鼠标左键点击时的位置；MouseMove 在鼠标被按下左键是，会随鼠标移动来改变 Point 的位置，实现对 TPointData 对象的鼠标拖动；MouseUp 在鼠标被释放时，更新 IsMouseDown 标记。

TPointData 为 TPointData 的构造函数和析构函数，有两个版本，一个指定了创建的位置，另一个继承了 TShape 的构造函数。

Paint 方法为响应 WM_PAINT 消息的绘图函数，负责根据 Parent 的相关属性和自身的某些属性将Point绘制在 Parent的 Canvas上(参考前面关于TGraphicControl

的介绍），最终调用了父类的 Paint 方法。

2.3.2 TDataPick

在 TDataPick 中定义了若干个新的属性，大多数是供 Point 显示使用的，也有一些是供自身显示使用的，这些属性在源代码的注解中有简单的说明，这里不多介绍，带有 FSet 开头的方法是这些属性的设置函数。这里没有采用 C++Builder 的命名规则（Set 开头），而是在前面加了 F 前导符，只是为了说明命名规则并不是强制性的。

在 TDataPick 的属性中，StratPointX、StartPointY、EndPointX、EndPointY 是用于存储起始点和终点的位置，他们将会根据设计时用户的设置将起始点和终点的位置信息存储在.dfm 文件中，进而可以体现在运行时。需要达到这样的需求还可以使用上一章中介绍的存储未发布的属性，或者对整个 StartPoint 和 EndPoint 进行存储等方式。在本案中，这四个属性共用了一组读写方法 FGetPoint 和 FSetPoint 来从 StartPoint 和 EndPoint 中提取和设置位置信息，Index 是读写方法的索引参数（参考前面关于属性的内容）。

TDataPick(TComponent* Owner)为构造函数，主要完成对缺省属性值的设置、创建起始点和终点对象以及对继承父类的属性的设置，在这个例子中，创建起始点和终点的时候，指定其 Owner 属性为空，这样可以使 TDataPoint 对象在设计时可以响应鼠标消息，但不足之处是 TDataPick 实例销毁的时候不能自行销毁起始点和终点对象的实例，必需在析构函数~TDataPick 中使用 **delete** 关键字显式的销毁，这和上一节例子中的处理方法稍有不同。

WM_Mouse 是 WM_MOUSEDOWN 消息的处理函数，当用户点击左键的时候，会在图形上创建一个新的属性控制点 TPointData 的实例并重新绘制曲线。在 TDataPick 中，使用了消息映射表的方式来响应鼠标消息，也可以使用 TPointData 中重载 MouseDown 方法的方式，这两种方式没有本质的区别，只是不同的两种方式，但是使用消息映射表需要调用父类的消息处理过程才能执行缺省的消息处理方法。在 WMMouse 方法中，我们没有调用父类的消息处理方法，因此不会不会调用系统的缺省操作。

TDataPick 的一个重要方法是从 TCustomeControl 继承来的 Paint 方法，这是用于将需要的图形信息绘制到画布上，在 TDataPick 控件中，绘制的图形比较复杂，同时又有其他 Child 控件需要绘制，直接绘图到 Canvas 上会导致图形闪烁，我们在这里使用了幕后位图进行缓冲来克服这个问题。

Paint 方法主要有三个部分构成，一是绘制背景和网格，二是绘制曲线，三是将 Child 控件绘制到 Canvas 上，网格和 Child 控件是根据需要进行绘图，在绘制曲线的过程中，使用到了插值函数进行曲线的拟合。

至此，TDataPick 的界面部分已经基本构成，但还没有最终完成，因为在这个控件中没有提供给用户数据的输出接口，无论是属性控制点的数据还是经过拟合后的插值数据，都没有提供给用户，为了能够使用这个控件，我们还需要进一步完善。这个工作留给读者来完成，在完成这个工作的时候需要注意的是，必需提供图像坐标和实际数据坐标的转换过程或提供转换比例 Scale 属性，提供获取数据的方法，在某些应用中，比如频谱其 X 轴并不是线性分布，人们通常更习惯使用对数坐标系，也可以考虑增加一个用于描述

坐标系的属性。

3 创建属性编辑器和控件的缺省动作

在控件设计中，一个很重要的内容就是为新建的控件定义缺省动作和定义新类型属性的编辑器，以及定义拷贝粘贴等动作的数据等，前一章我们已经简单介绍过了这些内容，在本节中将就一个非常简单的展示如何为新属性创建个性的编辑器，如何为控件在 IDE 中定义缺省动作和弹出菜单的内容。

3.1 控件简介

这个控件本来是 C++Builder 所附带的一个控件的例子（原控件名称为 TPies，在 Sample 面板中），如右图所视，控件在父窗体上画一个饼状图形，当双击控件或选择弹出菜单的第一项时，弹出"Pie Angels Editor"对 Pies 的 Angels 属性进行编辑。

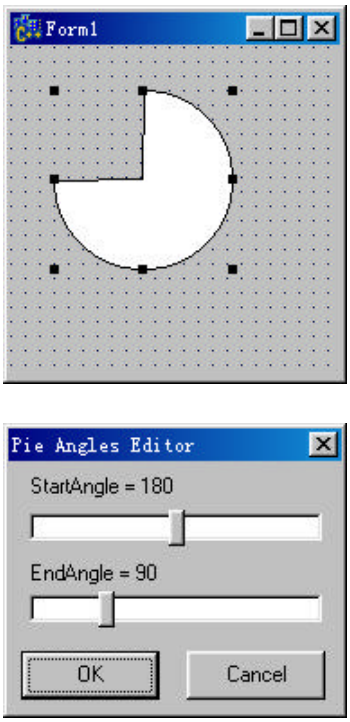


图 6.4 Tangles 的编辑器

Angles 是一个 TAngles 类型的可读写属性，该属性具有两个用于记录角度的属性：StartAngle 和 EndAngle。由于 TAngles 是从 TPersistent 类派生而来，因此该属性没有自己的编辑器，甚至在 Object Inspector 中也不能进行编辑，因此需要对 TAngles 类创建一个编辑器，这个例子中，TAngles 有两种编辑方式：一是通过 Object Inspector 编辑，如同 TFont 那样，其子属性在 Object Inspector 中展开；另一种方式是双击该属性，弹出右图 6.4 中的对话框编辑器。

这个控件的属性和实现均没有什么特殊的，主要是对新建的属性增加了编辑器，并定义了缺省动作，我们来看如何实现。

首先是建立一个用于记录起始角度和终止角度的类 TAngles，然后再定义我们的主控件 TMyPie，并且需要为 Angles 对象创建和注册一个 IDE 中使用的属性编辑器 TAnglesProperty，使 Angles 属性可以在 Object Inspector 中按照要求编辑，最后还要创建并注册一个控件的编辑器，控件编辑器可以指定控件在 IDE 中被双击或者点击右键弹出的菜单的动作。

3.2 源代码

读者可以在 C++ Builder 的例子中找到这个例子的源代码，原文分作两个单元，TPies 控件的单元和编辑器等注册单元，作者对其稍作修改，并加入了注解，如下：

3.2.1 声明部分

MyPies.h 文件，其中包含了五个类：TAngles、TMyPie、TAngleEditorDlg、TAnglesProperty、TPieEditor。

```
#ifndef MYPiesHPP
#define MYPiesHPP
#include <StdCtrls.hpp>
#include <DsgnIntf.hpp>
#include <Buttons.hpp>
#include <Windows.hpp>
#include <System.hpp>
#include <StdCtrls.hpp>
#include <Graphics.hpp>
#include <Forms.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <math.h>
#include <ComCtrls.hpp>
#include <memory>

class  PACKAGE TAngles : public TPersistent
{    //定义 TAngles 数据类型，将作为 TPie 的属性
private:
    int FStartAngle;                          //饼状的起始角度
    int FEndAngle;                            //饼状的终止角度
    TNotifyEvent FOnChange;                   //OnChange 事件的内部存储数据
    void __fastcall SetStart(int Value);      //用于设置起始角度
    void __fastcall SetEnd(int Value);        //用于设置终止角度
public:                                       //Assign 用于对象的拷贝
    void __fastcall Assign(TPersistent* Value);
    void __fastcall Changed(void);            //Change 用于反映设置值的变化
    __fastcall TAngles(void);                 //构造函数和析构函数
    __fastcall virtual ~TAngles(void);
__published:                                  //定义需要的属性和事件
    __property int StartAngle = {read=FStartAngle, write=SetStart,
                                 nodefault};
    __property int EndAngle = {read=FEndAngle, write=SetEnd, nodefault};
    __property TNotifyEvent OnChange = {read=FOnChange, write=FOnChange};
```

```
};
//从 TCustomControl 派生 TMyPie
class  PACKAGE TMyPie : public TCustomControl
{
__published:
    TPen    *FPen;                              //Pen 属性的内部变量
    TBrush  *FBrush;                            //Brush 属性的内部变量
    TAngles *FAngles;                           //Angles 属性的内部变量
    virtual void __fastcall Paint(void);        //重载 Paint 方法
    void __fastcall SetBrush(TBrush* Value); //属性的写方法
    void __fastcall SetPen(TPen* Value);        //属性的写方法
    void __fastcall SetAngles(TAngles* Value);   //属性的写方法
    void __fastcall StyleChanged(TObject* Sender);//反映改变的方法
    __property Anchors ;                        //继承父类的属性
    __property Constraints ;                    //继承父类的属性
    __property TAngles* Angles = {read=FAngles, write=SetAngles, nodefault};
    __property TBrush* Brush = {read=FBrush, write=SetBrush, nodefault};
    __property TPen* Pen = {read=FPen, write=SetPen, nodefault};
    __property ShowHint ;                       //继承父类的属性
    __property OnClick ;                        //下同
    __property OnDblClick ;
    __property OnDragDrop ;
    __property OnDragOver ;
    __property OnEndDock ;
    __property OnEndDrag ;
    __property OnMouseDown ;
    __property OnMouseMove ;
    __property OnMouseUp ;
    __property OnStartDock ;
    __property OnStartDrag ;

public:
    __fastcall virtual TMyPie(TComponent* AOwner);
    __fastcall virtual ~TMyPie(void);
};

class  PACKAGE TAngleEditorDlg : public TForm
{       //定义 TAngles 的编辑对话框
__published:
```

```
    TLabel *EAngleLabel;                    //终止角度的显示标题
    TLabel *SAngleLabel;                    //起始角度的显示标题
    TButton *OKButton;                      //确定按钮
    TButton *CancelButton;                  //取消按钮
    TTrackBar *STrackBar;                   //编辑起始角度的TrackBar
    TTrackBar *ETrackBar;                   //编辑终止角度的TrackBar
    void __fastcall CancelClick(TObject* Sender);  //取消按钮的Click句柄
    void __fastcall STrackBarChange(TObject* Sender);
                                            //起始TrackBar的OnChage句柄
    void __fastcall ETrackBarChange(TObject* Sender);
                                            //终止TrackBar的OnChage句柄
private:
    int FOrigStart;          //存储原来的起始角度，当取消编辑时用于恢复原始值
    int FOrigEnd;            //存储原来的终止角度，当取消编辑时用于恢复原始值
    TAngles* FAngles;        //被编辑的TAngles对象
    void __fastcall SetStartAngle(int Value);    //起始角度的设置函数
    void __fastcall SetEndAngle(int Value);      //终止角度的设置函数
    void __fastcall SetAngles(TAngles* Value);   //TAngles的设置函数
public:
    __property TAngles* EditorAngles = {read=FAngles, write=SetAngles,
                                        nodefault};
    __fastcall virtual TAngleEditorDlg(TComponent* AOwner);   //构造函数1
    __fastcall TAngleEditorDlg(TComponent* AOwner, int Dummy);//构造函数2
    __fastcall TAngleEditorDlg(HWND ParentWindow);            //构造函数3
    __fastcall virtual ~TAngleEditorDlg(void);                //析构函数
};

class PACKAGE TAnglesProperty : public TClassProperty
{                                           //定义TAngles的属性编辑器
public:
    virtual void __fastcall Edit();      //重载Edit方法
    virtual TPropertyAttributes __fastcall GetAttributes(void);
                                            //重载GetAttributes方法
    __fastcall virtual ~TAnglesProperty(void);       //析构函数和构造函数
    __fastcall TAnglesProperty(const _di_IDesigner ADesigner,
                               int APropCount);
};

class PACKAGE TPieEditor : public TDefaultEditor
```

```
{       //定义 TAngles 的缺省编辑器，向 IDE 的弹出菜单添加项目
protected:
   virtual void __fastcall EditProperty(TPropertyEditor* PropertyEditor,
                                  bool& Continue, bool& FreeEditor);
public:
   virtual void __fastcall ExecuteVerb(int Index);      //用于执行菜单操作
   virtual AnsiString __fastcall GetVerb(int Index);    //获取菜单标题
   virtual int __fastcall GetVerbCount(void);           //获取菜单数目
   __fastcall virtual TPieEditor(TComponent* AComponent,
                     _di_IDesigner ADesigner);
   __fastcall virtual ~TPieEditor(void);            //构造函数和析构函数
};
#define PI 3.1415926535897932385                    //用于计算角度
#endif
```

3.2.2 实现部分

在 MyPies 单元中代码如下：

```
#pragma hdrstop
#include  "Mypies.h"
#pragma resource "*.dfm"
#pragma package(smart_init)

__fastcall TAngles::TAngles(void) : TPersistent()
{    //空的构造函数
}
//---------------------------------------------------------------------
__fastcall TAngles::~TAngles(void)
{    //空的析构函数
}
//---------------------------------------------------------------------

void __fastcall TAngles::Assign(TPersistent* Value)
{    //Assign 方法用于 TAngles 对象的数据拷贝
    StartAngle = dynamic_cast<TAngles*>(Value)->StartAngle;
    EndAngle = dynamic_cast<TAngles*>(Value)->EndAngle;
}
//---------------------------------------------------------------------
```

```
void __fastcall TAngles::SetStart(int Value)
{  //属性改变通过 Changed 方法放映在界面上（这里只是触发的 OmChange 事件）
   if (Value != FStartAngle)
   {
      FStartAngle = Value;
      Changed();
   }
}
//---------------------------------------------------------------------

void __fastcall TAngles::SetEnd(int Value)
{  //与上面的方法类似
   if (Value != FEndAngle)
   {
      FEndAngle = Value;
      Changed();
   }
}
//---------------------------------------------------------------------

void __fastcall TAngles::Changed()
{  //这是标准的 VCL 事件的处理方法，可以根据需要将 Changed 方法定义为虚拟函数
   if (FOnChange != NULL) //这样可以在其派生类重载来改变 Changed 的处理过程
      FOnChange(this);
}
//---------------------------------------------------------------------

__fastcall TMyPie::TMyPie(TComponent* AOwner): TCustomControl(AOwner)
{  //MyPie 的构造函数，对属性的缺省值和事件进行初始化，
   Width = 100;
   Height = 100;
   FPen = new TPen();  //这里采用了新建对象方式来记录 Pen、Brush 等属性，
   FPen->OnChange = StyleChanged;        //也可以采用前面我们使用过的将子控
   FBrush = new TBrush();                //件的属性映射到父控件的方式
   FBrush->OnChange = StyleChanged;      //但代码需要作适当修改
   FAngles = new TAngles();
   FAngles->OnChange = StyleChanged;
   FAngles->StartAngle = 180;
   FAngles->EndAngle = 90;
```

```
}
//---------------------------------------------------------------------

__fastcall TMyPie::~TMyPie(void)
{
    delete FPen;          //析构函数负责将自己创建并且不能够
    delete FBrush;        //自动销毁的对象销毁
    delete FAngles;
}
//---------------------------------------------------------------------

void __fastcall TMyPie::StyleChanged(TObject* /*Sender*/)
{
    Invalidate();         //本案中只是进行了函数调用的传续
}
//---------------------------------------------------------------------

void __fastcall TMyPie::SetBrush(TBrush* Value)
{   //对象赋值需要拷贝，因为 Value 不一定是永久生存的对象
    FBrush->Assign(Value);
}
//---------------------------------------------------------------------

void __fastcall TMyPie::SetPen(TPen* Value)
{   //同上
    FPen->Assign(Value);
}
//---------------------------------------------------------------------

void __fastcall TMyPie::SetAngles(TAngles* Value)
{   //同上并且进行图形的刷新
    FAngles->Assign(Value);
    Invalidate();
}
//---------------------------------------------------------------------

void __fastcall TMyPie::Paint()
{   //Paint 方法根据起始点和终点以及颜色等在花布上绘制饼状图形
    int StartA, EndA;
```

```
    int midX, midY, stX, stY, endX, endY;
    float sX, sY, eX, eY;
    StartA = FAngles->StartAngle;
    EndA = FAngles->EndAngle;
    midX =  Width/2;
    midY = Height/2;
    sX = cos((StartA / 180.0) * PI);
    sY = sin((StartA / 180.0) * PI);
    eX = cos((EndA / 180.0) * PI);
    eY = sin((EndA / 180.0) * PI);
    stX = floor(sX * 100);
    stY = floor(sY * 100);
    endX = ceil(eX * 100);
    endY = ceil(eY * 100);
    Canvas->Pen = FPen;
    Canvas->Brush = FBrush;
    Canvas->Pie(0,0,
            Width,Height,
             midX + stX, midY - stY,
             midX + endX, midY - endY);

}
//---------------------------------------------------------------------

namespace Mypies
{    //注册控件、组件编辑器以及 TAngles 的属性编辑器等
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TMyPie)};
    RegisterComponents("MyInterFace", classes, 0);
    RegisterComponentEditor(__classid(TMyPie), __classid(TPieEditor));
    RegisterPropertyEditor(__typeinfo(TAngles),
                      NULL,
                      "",
                      __classid(TAnglesProperty));
  }
}
//---------------------------------------------------------------------
```

```
__fastcall TAngleEditorDlg::TAngleEditorDlg(TComponent* AOwner)
                            : TForm(AOwner)
{   //空的构造函数
}
//---------------------------------------------------------------------

__fastcall TAngleEditorDlg::TAngleEditorDlg(TComponent* AOwner,int Dummy)
                           : TForm(AOwner, Dummy)
{   //空的构造函数
}
//---------------------------------------------------------------------

__fastcall TAngleEditorDlg::TAngleEditorDlg(HWND ParentWindow)
                            : TForm(ParentWindow)
{   //空的构造函数
}
//---------------------------------------------------------------------

__fastcall TAngleEditorDlg::~TAngleEditorDlg(void)
{   //空的析构函数
}
//---------------------------------------------------------------------

void __fastcall TAngleEditorDlg::STrackBarChange(TObject* /*Sender*/)
{   //当起点 TrackBar 改变时更新起点角度
    SetStartAngle(STrackBar->Position);
}
//---------------------------------------------------------------------

void __fastcall TAngleEditorDlg::ETrackBarChange(TObject* /*Sender*/)
{   //当终点 TrackBar 改变时更新终点角度
    SetEndAngle(ETrackBar->Position);
}
//---------------------------------------------------------------------

void __fastcall TAngleEditorDlg::SetStartAngle(Integer Value)
{   //设置起始角度的同时改变显式字符，并保证 TrackBar 和 Value 一致
    STrackBar->Position = Value;  //这里不会出现和 OnChange 事件无限次循环调用，
    SAngleLabel->Caption = "StartAngle = " + String(Value);
```

```
    FAngles->StartAngle = Value;     //因为当 STrackBar->Position
                                     //和 Value 相等时，将不会触发 OnChange 事件
}
//---------------------------------------------------------------------

void __fastcall TAngleEditorDlg::SetEndAngle(Integer Value)
{   //同上
    ETrackBar->Position = Value;
    EAngleLabel->Caption = "EndAngle = " + String(Value);
    FAngles->EndAngle = Value;
}
//---------------------------------------------------------------------

void __fastcall TAngleEditorDlg::SetAngles(TAngles* Value)
{   //获取被编辑的对象，并获取其包含属性反映在 Form 中
    FAngles = Value;
    FOrigStart = Value->StartAngle;  //保存原始值
    FOrigEnd = Value->EndAngle;      //保存原始值
    SetStartAngle(Value->StartAngle);
    SetEndAngle(Value->EndAngle);
}
//---------------------------------------------------------------------

void __fastcall TAngleEditorDlg::CancelClick(TObject* /*Sender*/)
{   //取消时恢复原始取值
    SetStartAngle(FOrigStart);
    SetEndAngle(FOrigEnd);
}
//---------------------------------------------------------------------

__fastcall TAnglesProperty::TAnglesProperty(const _di_Idesigner
                              ADesigner,int APropCount)
                             :TClassProperty(ADesigner, APropCount)
{   //空的构造函数
}
//---------------------------------------------------------------------

__fastcall TAnglesProperty::~TAnglesProperty(void)
{  //空的析构函数
```

```
}
//---------------------------------------------------------------------

void __fastcall TAnglesProperty::Edit()
{
   //这里使用了 auto_ptr 来分配对象和内存,当作用域 结束时内存会自动释放.
   std::auto_ptr<TAngleEditorDlg> AngleEditor(
                              new TAngleEditorDlg(Application));
   AngleEditor->EditorAngles = (TAngles*)GetOrdValue();
   AngleEditor->ShowModal();  //将被编辑的属性传给属性编辑对话框,
}
//---------------------------------------------------------------------

TPropertyAttributes __fastcall TAnglesProperty::GetAttributes()
{  //使 TAngles 的属性编辑器具有对话框编辑和子属性展开的能力
   return (TPropertyAttributes() << paDialog << paSubProperties);
}
//---------------------------------------------------------------------

__fastcall TPieEditor::TPieEditor(TComponent* AComponent,
                    _di_IDesigner ADesigner)
                  : TDefaultEditor(AComponent, ADesigner)
{  //空的构造函数
}
//---------------------------------------------------------------------

__fastcall TPieEditor::~TPieEditor(void)
{  //空的析构函数
}
//---------------------------------------------------------------------

void __fastcall TPieEditor::EditProperty(TPropertyEditor* PropertyEditor,
                           bool& Continue, bool& /*FreeEditor*/)
{  //PropertyEditor 将指向 TAnglesProperty 对象,该方法被 Edit 方法调用,
   String PropName(PropertyEditor->GetName()); //获取属性名
   if (strcmpi(PropName.c_str(),"ANGLES") == 0)
   {  //属性名为"ANGLES"时,调用该属性的编辑器(PropertyEditor)的 Edit 方法
      PropertyEditor->Edit();
      Continue = false;     //不再继续枚举其它的属性和事件
```

```
    }
}
//---------------------------------------------------------------------

int __fastcall TPieEditor::GetVerbCount()
{
    return 1;  //获取添加的菜单项的数目
}
//---------------------------------------------------------------------

String __fastcall TPieEditor::GetVerb(Integer Index)
{
    if (Index == 0)    //获取添加的菜单项的显示文本（Caption）
      return "Editor";
    else               //忽略其它非法索引，有效范围为 GetVerbCount - 1 ;
      return String("");
}
//---------------------------------------------------------------------

void __fastcall TPieEditor::ExecuteVerb(Integer Index)
{
    if(Index == 0)     //指向菜单项，Edit 为父类的虚拟函数，将调用属性的编辑器
      Edit();
}
//---------------------------------------------------------------------
```

3.3 控件的详解

在这个控件中，总共使用了五个类，分别是 TAngles、TMyPie、TAngleEditorDlg、TAnglesProperty、TPieEditor，它们担任者不同的角色，也有着不同作用，下面我们分别作以介绍。

3.3.1 TAngles

这是一个从 TPersistent 派生的类，因此具有将数据存储在.dfm 文件中的能力，在这个类中，有两个属性 StartAngle 和 EndAngle 及一个事件 OnChange，这两个属性分别记录了两个角度用于表示起始和终止角度，然而，TAngles 类不仅是给 TMyPies 使用的，它应该是一种普遍适应的变量类型，而且是支持 VCL 的，并且 TAngles 并不自己将变化反映在界面上，因为它并不是一个控件，但必需为 TAngles 提供一个用于通报或者是反映属性改变的接口，这就是 OnChange 事件，OnChang 事件是标准的

TNotifyEvent 事件类型。

在 TAngles 中，我们对变化的处理方法和 DataPick 控件大不一样，在 DataPick 中，TDataPoint 是我们 TDataPick 专用的，而且并没有希望这个类会被其它的控件作为属性，因此我们在 TDataPoint 的属性设置函数中对 TDataPoint 属性变化作了相应的处理，而 TAngles 将这种变化作成一个接口 OnChange，通过 OnChange 事件可以使使用 TAngles 对象作为属性的控件可以自由的指定 Angles 的值，当 TAngles 变化时，控件作出正确的反应,这就象在 TPickEdit 中我们对 TSpeedButton 的 OnClick 事件的处理一样。这是不同的处理方法，但最终的目的和结果是相同的，可以根据实际情况选择处理方法。

OnChange 事件是由 Change 方法调用的,也就是在 Change 方法中触发 OnChange 事件，前面我们也讲过，这是 VCL 标准的处理方式，也可以将 Change 声明为虚拟函数，这样以来，在 TAngles 的派生类中，可以通过重载 Change 方法使在触发 OnChange 事件之前作某些相应的操作，由此而实现对 OnChange 事件的重载。

在 TAngles 类中，另一个很重要的内容是 Assign 方法，我们知道在 VCL 类库和其派生类中，没有标准 C++的拷贝构造函数，Delphi 中为 VCl 类库供了用于拷贝的方法 Assign 和 AssignTo，在后面的例子 TDBJpegImage 中发现这个方法比拷贝构造更加灵活，关于这两个方法的详细应用读者参考前面有关章节。

3.3.2 TMyPie

这是一个由 TCustomCtrol 派生的控件类，在 C++ Builder 的例程中是从 TGraphicControl 派生而来，这也是我们最终需要向 IDE 注册的控件。在 TMyPie 控件中，有三个新扩展的属性：Pen、Brush、Angles，其中 Pen 和 Brush 是用于显示饼状图形的画笔和画布的刷子，SetBrush 和 SetPen 分别是它们的设置函数，这两个属性也可以采用我们前面是用过的属性映射的方式，TMyPie 中是对 Pen 和 Brush 重新创建了新的对象。

StyleChanged 方法用于反应属性的改变，构造函数负责创建新的 Pen 和 Brush，析构函数则进行销毁，重载的 Paint 方法会相应 WM_PAINT 消息会画布进行绘制。其它属性都是从其父类继承而来。

3.3.3 TAngleEditorDlg

TMyPie 控件和前面介绍的几个控件相比较，最大的特点是为新创建的属性增加了属性编辑器，TAngleEditDlg 是从 TForm 派生而来的对话框，如前面图中所视的那个编辑器，这个 Form 对象中包含了两个 TTrackBar、两个 TLabel 和两个 TButton 对象，这些可以在 IDE 中可视化编辑,包括三个声明在 **publish** 定义域的方法 CancelClick、STackBarChange 和 ETrackBarChange 方法。

另外在 TAngleEditDlg 对话框中声明了一个 TAngles 类型的属性 EditorAngles，它的作用是将被编辑的属性 Angles 引入到对话框以便对其编辑（设置），这个属性在 TMyPie 控件的 Angles 属性被编辑时,由 TAnglesProperty 的编辑方法 Edit 负责获

取并交给 EditorAngles 属性，因此在 Form 中对 EditorAngles 的操作将完全反应在 TMyPie 的 Angles 属性上。EditorAngles 的另一个作用是将 Angles 属性中的起始角度和终止角度同步的反映在 TAnglesEditDlg 的界面上，同时将图形界面上的变化同步的反映到 Angles 属性中，实现编辑器和属性的互动操作。

除此之外，TAngleEditDlg 并没有其它特殊之处，和一般的 Form 基本一样，读者自行理解。当用户在 IDE 中双击控件或者在 Object Inspector 双击 Angles 属性时，TAnglesEditDlg 将会被 TAnglesProperty 的 Edit 方法创建，同时将 Angles 属性传递给 TAngleEditDlg，用户在编辑对话框中的编辑会实时的反映在 Angles 属性和 TMyPie 控件中，用户点击取消时，恢复编辑前的原始值，确定则改变为新的属性值。

3.3.4 TAnglesProperty

TAnglesProperty 是从类编辑器 TClassProperty 派生而来，需要完成的是两个工作，一是让 Angles 属性可以在 Object Inspector 被展开以及允许打开编辑器对话框，这个工作是通过重载 GetAttrbutes 方法来实现的；另一个工作是在 Edit 方法中打开编辑器对话框，也就是反映用户双击属性时的动作。这两件事都很简单，读者参考源码理解。

关于属性编辑器的详细内容请读者参阅联机帮助。

3.3.5 TPieEditor

TPieEditor 是从 TDefualtEditor 派生而来的组件编辑器，组件编辑器的目的是在 IDE 中对组件进行编辑，诸如双击组件、拷贝组件、粘贴组件、点击右键等编辑操作所作的反应，TDefualtEditor 为组件定义了通用的编辑功能，比如剪切、拷贝等等，但某些时候针对不同的控件我们可能经常需要定义个性的编辑功能，比如在拷贝控件的时候将其某些属性的数据一同拷贝，或者指定双击组件的缺省动作以及在快捷菜单中增加其它的编辑功能。

在这个例子中，TPieEditor 在快捷菜单中增加了一个项目“ Editor ”单击 Editor 菜单的时候，IDE 将打开 Angles 的对话框编辑器，向快捷菜单中添加菜单项并且响应需要重载三个方法：ExecuteVerb、GetVerb 和 GetVerbCount，GetVerbCount 是 IDE 用来获取用户添加的菜单项的数量，GetVerb 供 IDE 来获取每一个菜单项的显示字符串，ExecuteVerb 是供 IDE 调用某个自定义菜单项被点击时的接口。在这个例子中 GetVerbCount 返回为“ 1 ”，即有一个用户自定义菜单项，GetVerb 中对应的是 Index 为“ 0 ”的菜单项，返回字符串“ Editor ”。这时，控件在 IDE 中快捷菜单就添加了一个新的项目“ Editor ”，当用户点击该菜单项时，IDE 调用 ExecuteVerb，并以 Index=0 为参数，在这个例子中，当 ExecuteVerb 的 Index 参数为“ 0 ”时，调用 TPieEditor 的 Edit 方法。

Edit 也是一个虚拟函数，是当用户在 IDE 中双击组件时调用的，缺省情况，TDefultEditor 的 Edit 方法首先会寻找该组件的 OnCreate 事件，如果 OnCreate 事件没有发布，再寻找 OnChanged 事件，然后是 OnClick 事件，若这些事件都没有发布，则按照 Object Inspector 中该组件的第一个事件来在 Code Editor 中创建该事

件的处理方法，需要更改双击组件的缺省动作，只需要重载 EditProperty 方法。这个例子中，EditProperty 方法将在双击控件时调用 Angles 的属性编辑器，例中 EditProperty 在 Edit 方法被调用后依次列出组件的每一个属性及事件供编辑,其参数 Continue 决定是否继续枚举，当枚举找到 Angels 时，停止继续枚举，并根据 Angles 属性打开属性编辑器。

TMyPie 控件是一个比较全面的应用 IDE 特征的控件，作为一个组件设计人员，必需掌握属性编辑器、控件编辑器等方面的内容,但是本书并不是针对 VCL 组件开发来编写的，主要是通过组件来掌握如何面向状态，来掌握在 C++ Builder 中如何这种语言所提供的新功能。需要进一步了解属性编辑器和组件编辑器请读者参考相关书籍或者联机帮助。

4 创建数据控件 TDBJpegImage

接下来的一个例子是创建一个用于数据库的控件，在 C++ Builder 中，Borland 为我们提供了非常强大的数据库处理能力，一个非常有用的功能就是数据库和图形显示界面的关联，这不仅仅是一个界面的事情，而更多的是使用 VCL 的面向状态的技术使各个对象之间的关系简化了许多。在 TDBJpegImage 控件中，代码非常的短，但在这段代码的背后隐藏着 Borland 公司的 VCL 中最精湛的技术，也体现了 VCL 强大的功能。

在 Borland 的 VCL 中提供了用于图形存储的数据库控件 TDBImage，但遗憾的是这个 DBImage 并不能支持 Jpeg 格式的图形数据，VCL 还提供了一个用于存取 Jpeg 数据格式的组件 TJPEGImage，既然 VCL 中有了 Jpeg 的处理能力，我们就可以编制一个用于存取 Jpeg 的数据库控件。

之所以选择数据库控件作为这一章的最后一个例子，不仅是在网上经常有网友问起如何在数据库中存储 Jpg 格式的数据，更重要的是在数据库的控件中，各个控件之间存在着更加复杂逻辑关系，而使用了 VCL 你可能根本看不到它们是如何联系在一起，却可以非常默契的配合工作，这正是面向状态所带来的好处。当你需要设计一个新的控件的时候，也许并不需要过多考虑 Borland 是否在设计 VCL 时就考虑了向前兼容，因为回答是肯定没有，但是你却可以很容易的做到向后兼容，当然，要做到这一点，必需对 VCL 有这更深入的了解，同时对面向状态的技术也必需熟练的掌握。

4.1 控件的分析

我们先来看看一个数据库的控件需要和那些对象发生关系，以及这些个控件之间的联系是什么？在一个数据库程序中，需要使用数据库控件必需具备至少以下几个对象：

- 由 TDataSet 派生而来的数据集组件，如：TTable、TQuery、ADODataSet、ADOTable 等来连接某个数据库中的数据表。
- 至少包含一个 TDataSource 组件,提供给数据集和数据库控件之间的必要连接。
- 一个数据库控件来显示以及编辑数据库中的数据结果，如：TDBGrid、TDBEdit

等数据库控件。

这三个对象是必不可少的，但实际中对于数据库的操作仅有这几个对象还是不够的，诸如 TDataBase、TDataLink、TField 或它们的派生类也是必不可少的，但并不一定需要用户显示的创建和操作、有些是 VCL 在幕后自动进行的，有些则是可以使用其它方式变通使用的。

我们先来看一看最简单的数据库应用，假定包含一个 TDataSet 对象、一个 TDataSource 对象和一个自定义的数据库控件 TDBJpegImage。当需要为 TDBJpegImage 指定数据库时，必需指定一个数据源 TDataSource 对象和使用数据对应的字段名称，TDataSource 能够为数据库控件使用就必需有其连接的数据库表 TDataSet 对象。在这个连接关系中，不是一对一，一个 TDataSet 对象可以作为多个 TDataSource 的 DataSet；而且一个 TDataSource 对象又同时可以作为多个数据库控件的数据源。这种联系并不是为了更加好看，而是要完成数据的更新、编辑、存储等等一系列操作的同步和协调，而在这个联系中，还必须有一个幕后的中间对象 TDataLink 对象。这是这种错综复杂的逻辑关系使得程序设计上必需更多的考虑之间的联系，好在 VCL 使用的面向状态的技术可以是这些关系变得简单一些，使一些复杂的逻辑关系化简为一些简单的逻辑关系。

在这个 TDBJpegImage 中，我们将不再花费大量的篇幅完善控件本身的功能，仅仅提供了一个从剪贴板粘贴图片的接口，而且还是利用 VCL 本身来完成的，有兴趣的读者可以自行扩展功能，如剪切、清除、甚至从数字图像设备直接采集图像源的接口，这里只是希望能通过这个例子是读者更好理解在 VCL 中如何处理这种复杂的逻辑关系。

4.2 源代码

TDBJpegImage 中使用其它 BPI 中的组件 TJPEGImage，建立 Package 工程的时候，必需包含其它相关的 Bpi 文件，详细的信息以及源代码读者可以在本书的附带光盘中找到。

4.2.1 声明部分

该例子中 TDBJpegImage 是从 TCustomControl 派生而来，这样的话控件将可以拥有自己的窗口句柄，可以接受系统的键盘消息，更重要的是可以作为数字图像设备如：视频采集卡、数码相机、扫描仪、电脑眼等设备的回显窗体。如果希望在 TDBJpegImage 中实现更多的编辑功能，如：放大、缩小、区域剪切、拷贝、旋转等操作，使用窗体控件会获得比图形控件更高的执行速度。

下面是 DBJpegImage.h 的内容：

```
#ifndef DBJpegImageH
#define DBJpegImageH
//---------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Controls.hpp>
```

```
#include <Classes.hpp>
#include <Forms.hpp>
#include <Jpeg.hpp>   //这个单元包含了VCL 中对 Jpeg 数据的处理类 TJPEGImage 等
#include <DBCtrls.hpp>
#include <dbtables.hpp>
//---------------------------------------------------------------------
class PACKAGE TDBJpegImage : public TCustomControl
{    //从 TCustomControl 派生而来的 TDBJpegImage
private:
    TFieldDataLink * FDataLink;   //用于响应数据库变化和编辑
    TJPEGImage * FJpg;            //用于存储和读取 Jpeg 格式的图片
protected:
    void __fastcall DataChanged(TObject *Sender);     //反映数据库的改变
    void __fastcall UpdateData(TObject *Sender);      //更新数据库的数据
    AnsiString __fastcall  GetDataField(void);   //DataField 属性的读取函数
    TDataSource* __fastcall GetDataSource(void); //DataSource 的读取函数
    void __fastcall SetDataField(AnsiString Value);  //DataField 的设置函数
    void __fastcall SetDataSource(TDataSource* Value);
                                                  //DataSource 的设置函数
    virtual void __fastcall CreateParams(TCreateParams &Params );
public:        //CreateParams 是重载 TWinControl 的虚拟函数
    __fastcall TDBJpegImage(TComponent* Owner);       //构造函数
    __fastcall ~TDBJpegImage(void);                   //析构函数
    Graphics::TBitmap* FBmp;                          //用于存储图片的数据
    void __fastcall PasteFromCliborad(void);     //用于从粘贴板获取图片数据
    virtual void __fastcall Paint(void);              //绘制控件的函数
__published:              //数据库必需的属性 DataField 和 DataSource
    __property AnsiString   DataField =
                    {read=GetDataField,write=SetDataField,nodefault};
    __property TDataSource* DataSource =
                    {read=GetDataSource,write=SetDataSource,nodefault};
    __property Color ;                                //继承发布父类的属性
    __property Constraints ;                          //同上
    //……                                             //同上
    __property Visible ;                              //同上
    __property OnClick ;                              //继承发布父类的事件
    __property OnContextPopup ;                       //同上
    __property OnDblClick ;                           //同上
    //……                                             //同上
```

```
};
//---------------------------------------------------------------------
#endif
```

除了为TDBJpegImage定义了一个新的绘图函数Paint和一个用于从剪贴板中粘贴数据的PasteFromCliborad方法以外，没有任何新的东西，但这已经可以完成一个最简单的Jpeg数据库图像的存取。

4.2.2 实现部分

```
#include <vcl.h>
#pragma hdrstop

#include "DBJpegImage.h"
#include <clipbrd.hpp>
#pragma package(smart_init)
//---------------------------------------------------------------------

static inline void ValidCtrCheck(TDBJpegImage *)
{
       new TDBJpegImage(NULL);
}
//---------------------------------------------------------------------
__fastcall TDBJpegImage::TDBJpegImage(TComponent* Owner)
       : TCustomControl(Owner)
{
    ControlStyle = ControlStyle << csOpaque << csReplicatable;
    if(!NewStyleControls)                    //设置控件的风格
      ControlStyle = ControlStyle << csFramed;
    FDataLink = new TFieldDataLink;          //创建 DataLink 属性的对象
    FDataLink->Control = this;               //获取 DataLink 的控制权
    FDataLink->OnDataChange = DataChanged;   //设置 OnDataChanged 事件
    FDataLink->OnUpdateData = UpdateData;    //设置 OnUpdateData 事件
    FBmp = new Graphics::TBitmap();          //创建内部位图
    FJpg = new TJPEGImage();                 //创建 Jpeg 图片对象
}
//---------------------------------------------------------------------

void __fastcall TDBJpegImage::CreateParams(TCreateParams &Params )
```

```
{      //使控件具有边框和 3D 效果
    TCustomControl::CreateParams(Params);
    if( Ctl3D )
        Params.ExStyle = Params.ExStyle | WS_EX_CLIENTEDGE;
    else
        Params.Style = Params.Style | WS_BORDER;
    Params.WindowClass.style = Params.WindowClass.style &
                               !(CS_HREDRAW | CS_VREDRAW);
}

//---------------------------------------------------------------------
__fastcall TDBJpegImage::~TDBJpegImage(void)
{  //析构函数
    FDataLink->Control = NULL;
    FDataLink->OnDataChange = NULL;
    FDataLink->OnUpdateData = NULL;
    delete FDataLink;     //销毁构造函数创建的对象
    delete FBmp;
    delete FJpg;
}
//---------------------------------------------------------------------
namespace Dbjpegimage
{  //注册函数
   void __fastcall PACKAGE Register()
   {
      TComponentClass classes[1] = {__classid(TDBJpegImage)};
      RegisterComponents("MyInterFace", classes, 0);
   }
}
//---------------------------------------------------------------------
AnsiString __fastcall  TDBJpegImage::GetDataField(void)
{  //获取 DataField 属性（将 FDataLink 的属性映射到 TDBJpegImage）
    return FDataLink->FieldName;
}
//---------------------------------------------------------------------
TDataSource* __fastcall TDBJpegImage::GetDataSource(void)
{  //获取 DataSource 属性（将 FDataLink 的属性映射到 TDBJpegImage）
    return FDataLink->DataSource;
}
```

```
//---------------------------------------------------------------------
void __fastcall TDBJpegImage::SetDataField(AnsiString Value)
{  //设置 DataField 属性（将 FDataLink 的属性映射到 TDBJpegImage）
   FDataLink->FieldName = Value;
}
//---------------------------------------------------------------------
void __fastcall TDBJpegImage::SetDataSource(TDataSource* Value)
{  //设置 DataSource 属性（将 FDataLink 的属性映射到 TDBJpegImage）
   FDataLink->DataSource = Value;
}
//---------------------------------------------------------------------
void __fastcall TDBJpegImage::DataChanged(TObject *Sender)
{  //当数据库的数据发生变化的时候，将变化情况及时的反映在控件中
   TBlobStream *Stream =
               new TBlobStream((TBlobField*)FDataLink->Field,bmRead);
   try      //创建 Blob 字段的流，更妥当的方法是在创建流之前确
   {        //保字段是 Blob 类型，或者使用 try 关键字来捕获异常
     if( Stream->Size != 0 )
     {     //字段的长度不为 0 时，将数据载入到 Jpeg 对象中
       FJpg->LoadFromStream(Stream);
       FBmp->Assign(FJpg); //将 Jpeg 数据转换成 Bitmap 数据到 FBmp 中
     }
     else          //如果数据为空，则清除内部存储位图（刷成白底）
       FBmp->Canvas->FillRect(ClientRect);
     Paint();     //刷新显示
   }
   catch(...)
   {
                 //发生异常进行适当的处理
   }             //在__finally 组合语句中销毁 Stream 更合理一些，因为
   delete Stream;  //在__finally 中，不管是否发生异常，均会销毁 Stream
}
//---------------------------------------------------------------------
void __fastcall TDBJpegImage::UpdateData(TObject *Sender)
{  //更新被编辑的图片数据到数据库中
   TBlobStream *Stream =
               new TBlobStream((TBlobField*)FDataLink->Field,bmWrite);
   try                           //首先创建 TBlobStream
   {
```

```
        FJpg->Assign(FBmp);           //从内部位图中获取图形数据
        FJpg->Compress();             //压缩 Jpeg 对象的数据
        FJpg->SaveToStream(Stream);   //将 Jpeg 数据写入流
    }
    catch(...)
    {
                                      //进行异常处理
    }
    delete Stream;                    //销毁 Stream
}
//---------------------------------------------------------------------
void __fastcall TDBJpegImage::Paint(void)
{
    Canvas->FillRect(ClientRect);  //清除控件显示的内容
    Canvas->Draw(0,0,FBmp);        //将内部位图的数据显示在控件上
}
void __fastcall TDBJpegImage::PasteFromCliborad(void)
{   //从剪贴板中获取图形数据
    if( FDataLink->Editing )       //只有数据库处于编辑状态才作如下操作
    {
      TClipboard *pCB = Clipboard();    //获取剪贴板
      if (pCB->HasFormat(CF_BITMAP))    //检查数据类型
      {
        try
        {
          FBmp->Assign(pCB);            //获取数据
          Paint();                      //更新显示内容
          UpdateData(this);             //并更新数据库
        }
        catch (...)
        {
                                        //异常处理
        }
      }
    }
}
```

4.3 控件详解

在这个控件中几乎没有任何新的东西，只是将 VCL 类库作了一个简单的组合，但在使

用上则会方便很多，我们还是对该控件的方法作个简单介绍。

4.3.1 TDBJpegImage

ValidCtrCheck、Register 以及 CreateParams 不需要介绍了，前面的控件里出现过；

TDBJpegImage 是 TDBJpegImage 的构造函数，在数据库控件的构造函数中，必需创建一个 TDataLink 或其派生类的对象用于和 TDataSource 对象保持连接关系，并且需要对 TDataLink 对象的相关属性进行设置，主要有 Control 属性和 OnDataChange、OnUpdateData 事件。Control 用以标识 TDataLink 对象所连接的控件，OnDataChange 在数据集的数据发生变化的时候触发，提供给 TDBJpegImage 来更新界面的显示内容，OnUpdateData 在需要更新数据时触发，提供给 TDBJpegImage 将显示的图形保存到数据集中。这两个事件的处理函数是 TDBJpegImage 控件的关键，它们决定了如何将数据库中的数据显示到界面上，以及如何将界面上的数据以特定的格式保存在数据库中，OnDataChange 的处理函数为 DataChange，OnUpdateData 的处理函数为 UpDateData。

析构函数~TDBJpegImage 负责将构造函数创建的对象释放掉，也不需多讲。

以 Get 和 Set 开头的函数是属性的读取和设置函数，前面的控件中我们讲过，读者自行分析。

DataChange 是反应数据集改变时触发的 OnDataChange 事件的处理函数，任意一个数据库控件都需要有这样一个处理函数来将正确的内容显示给用户，比如 DBEdit 控件可以显示字符类型、整型数据、浮点类型数据、货币、时间等不同的类型，那么在 DBEdit 的 DataChange 事件中必需能够针对字段数据的类型，进行数据到文本的转换并显示到 DBEdit 中，TDBJpegImage 的数据类型为 Blob 类型，在 DataChange 函数中，应该包含对数据类型的判断，并且提供异常处理机制，甚至在 DataField 的设置方法中就应该存在对字段数据类型的检测。在这个控件里我们没有这样处理，这是不完整的，因为同一字段的数据类型运行和设计时不同是完全有可能的。按照完整性的要求，在 DataChange 函数中必需先对字段的数据类型进行判断，如果不是 Blob 类型，则抛出异常，提供给使用这进行善后的处理，否则应用程序是很危险的。

在 DataChange 函数中，首先创建了一个 Blob 类型的 Stream，有两种方式创建，读者可以参考帮助文件获取详细信息。然后将 BlobStream 的数据读入到 JpegImage 对象 FJpg 中，JpegImage 会自行解压，使用 TBitmap 的 Assign 方法可以将 JpegImage 解压后的位图数据复制到位图对象中，这个位图对象 FBmp 供 TDBJpegImage 的绘图方法 Paint 使用，而且 FBmp 不能立即释放，任何时候都要用来存放供 TDBJpegImage 显示内容的数据。这 DataChange 函数中，我们还使用了 try 关键字组合来对数据的读取尝试操作，因为 Blob 类型的数据也不能保证以前存储的数据符合 Jpeg 格式，但对于这个可能的异常行为，我们本身没有作任何特殊处理。

UpdateData 是 DataChange 的反向过程，当用户需要进行数据库数据的更新时，会触发 OnUpdateData 事件，UpdateData 将被调用，要知道你所看到的计算机屏幕上的数据，也许并不是数据库中真正的数据，只有在 UpdateData 被调用之后，这些数据才会真正的写入到数据库中，在一个完整的 UpdateData 函数中，不仅应该具有上面提

到的数据类型的判断和异常处理功能之外，还应该具有判断 TDBJpegImage 的数据是否真的被改变过，这是一个非常有用的判断，因为 Jpeg 数据的存储不仅数据量大，还需要对位图数据进行压缩，系统的开销绝对不允许不考虑，可以在 TDBJpegImage 的属性中加入一个记录编辑状态的属性，比如 IsModify 等，但在这个例子中我们并没有这么作，读者可以自行分析和修改。

在 UpdateData 函数中，我们又用到了那个非常有用的方法 Assign，前面介绍 VCL 类库时，我们已经分析过 Assign 方法和 AssignTo 方法的相互关系和运行机制，JpegImage 和 Bitmap 正好是这样一个例子，当 Borland 设计 TBitmap 组件时，并没有考虑在其后面的版本中支持 Jpg 格式，当后来设计 TjpegImage 时，同时重载了 Assign 和 AssignTo 方法,这样以来可以是 Bitmap 和 JpegImage 能够相互 Assign，当调用 JpegImage 的 Assign 方法拷贝一个 Bitmap 时，是真正的使用了 JpegImage 的 Assign 方法，反过来使用 Bitmap 的 Assign 方法拷贝一个 JpegImage 时，其最终使用的是 JpegImage 的 AssignTo 方法。我们的 TDBJpegImage 其实也应该考虑这个问题，那就是是否允许用户使用 Assign 方法来对 TDBJpegImage 和其它的图形控件，比如 TImage、TBitmap，甚至是剪贴板中的数据进行相互的拷贝，如果希望这样，那就应该考虑该如何重载 Assign 方法和 AssignTo 方法。

Paint 方法是 TDBJpegImage 的绘图函数，相对于前面的例子简单多，我们这里也不多讲，需要注意的是在将位图拷贝到控件的 Canvas 之前，应当清除 Canvas 上以前的内容。这是一个再简单不过的控件,如果希望达到更好的效果,可以考虑在 TDBJpegImage 控件中加入控制属性，比如是否按照比例显示，是否居中以及是否透明等属性，在 Paint 函数中根据属性不同的控制要求而选择不同的绘图效果。

PasteFromCliborad 是从剪贴板粘贴图像的函数，在这个例子中，我们提供更改数据的唯一图形就是通过 Windows 的剪贴板，同样也使用了 Assign 这个非常有用的拷贝函数。对于数据库的编辑，通常的控件是这样处理的：如果控件不是 ReadOnly 的，将允许控件进行数据的编辑等操作，同时还要通报数据集发生了这个变化过程，这个控件里我们没有这么作，读者自行考虑如何实现。

4.3.2 TDataSource

在所有数据库的组件里面，TDataSource 是一个十分重要的组件，C++ Builder 的不同版本中，提供了众多的数据引擎，但是无论使用什么样的数据库引擎，和数据库控件的关联都必需使用 TDataCource 组件来完成。数据库控件和 TDataSet、TDataLink 之间的关系，相对还是比较复杂的，我们这里仅就 TDataSource 组件作一个分析，简单介绍以下这几个组件之间完成数据库的连接所经历的过程。

TDataSet、TDataSource、TDataLink 之间具有两种关联关系，一种如下图，沿箭头方向为多对一的对应关系：TDataSet 的 FdataSource 中记录着每一个和其关联的 TDataSource 对象；TDataSource 的 FDataLink 中记录着每一个和其关联的 TDataLink 对象。

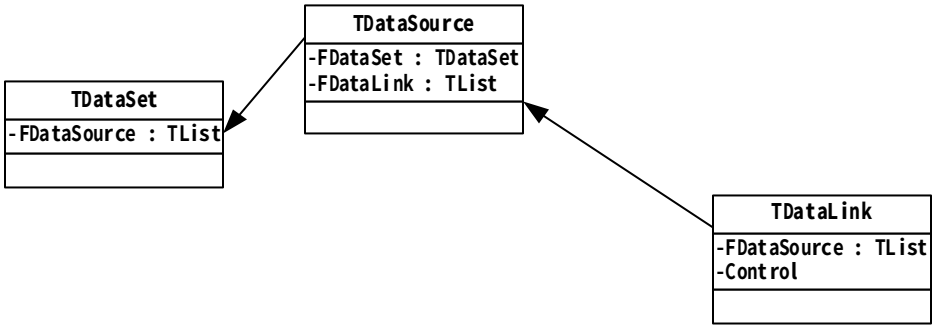


图 6.5 数据库对象之间的对应关系

另一种关系如下图，沿箭头方向是一对多的对应关系，正好是上面关系的反向连接，这两中关系共同组成了双向可追溯的相互关联关系。

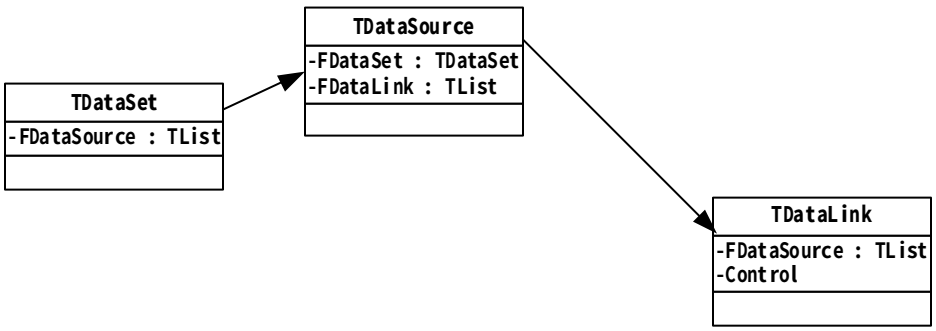


图 6.6 数据库对象之间的反向对应关系

从上面 TDBJpegImage 的源代码中我们已经知道，当建立一个数据库控件和数据源连接时，只需要指定其数据源和需要使用的字段就可以，而实际上当用户设置一个 TDataLink 的 DataSource 属性时，上面的连接关系自动会建立，相关的事件的消息传递等关系也会随之建立。由于这三个对象之间的联系并不是一对一的，因此在任意一个对象的属性被改变的时候，必需保证不能破坏原有的之前和其它同等对象建立的类似的关系。

比如一个 DataSource 可以提供给若干个数据库控件使用，任何一个控件设置其数据源为该 DataSource 对象时，必需保持这个 DataSource 和其它数据库控件之间的关系不受影响，而且必需保证数据库的事件能够正确的分发给每一个数据库控件。

这是 VCL 组件完整性的一个必需的要求，我们这里之来看一看当一个 TDataLink 对象设置其 DataSource 属性时，它们如何建立这种包含关系。完成这个一个连接过程，TDateSource 组件中有以下几个必需的属性或成员：

DataSet，用于记录 TDataSource 对象所连接的 DataSet 数据集。在每个数据集对象中包含一个 TList 的对象用于记录该 DataSet 连接的每一个 TDataSource 对象，和 TDataSource 对象中的 DataSet 相对应，组成一个双向可追溯的逻辑包含关系。

FDataLinks，这是一个 TList 列表类型的私有成员，用于记录 TDataSource 对象连接的 TLink 对象，这个列表不是属性，而且不是公开，是提供给内部使用，并且是由内部的过程完成设置的。在每一个 TDataLink 对象中包含一个 DataSource 属性和 FDataLinks 相对应，组成双向可追溯的逻辑包含关系。当用户设置一个 TDataLink 对

象的 DataSource 属性的时候，DataSource 的设置方法将调用被设置的 DataSource 对象的 AddLinks 方法，将该 TDataLink 对象添加到列表中。这个过程可以用下面的图来表示：

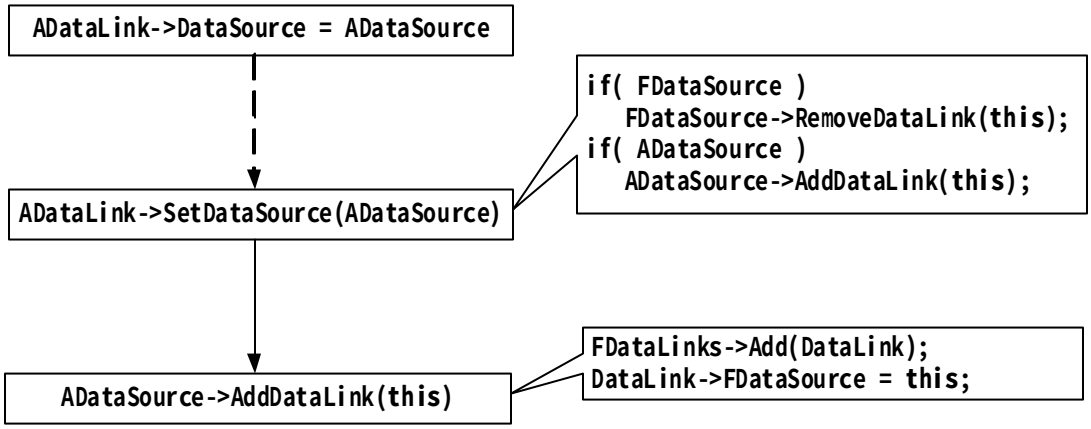


图 6.7 TDataLink 和 TDataSource 关系的建立过程

在这个过程中，我们可以看出 VCL 对于一个属性的设置是非常谨慎而且一致的，每一步都必需保证是有效的，而且不能对使用者提出过多的约束，这样的属性才是完整的，才是具有高可用性的。在 TDataSource 的属性中，象 DataSet 的设置过程和 TDataLink 的方式非常相似，而且过程更加严谨，因为在 TDataSet 的属性中还有其它的 TDataSource 类型的属性如：MasterSource 用于建立 Master-Detail 表索引关系，这时候必需保证不能建立环路包含关系，就是说引用自己的 DataSource 不能作为自身的 MasterSource 索引表源，因此在 TDataSet 的相关设置方法中必需对这个条件进行判断，不能满足条件时抛出异常，而不能将这种条件的判断交给引用者，否则这个属性也将是不完整的。

在上面图中的这个过程中，我们再次遇到了 C++和 Object Pascal 冲突的地方，在 C++代码中

```
ADataSource->AddDataLink(this);
```

方法进行了以下操作：

```
FDataLinks->Add(DataLink);
DataLink->FDataSource = this;    //C++语言中这是非法的
```

我们前面讲过，FDataSource 是属性的内部存储成员变量，一定是私有成员，因此在 DataSource 的方法中对 DataLink 的私有成员的访问是不允许的，但是在 Object Pascal 中，这是允许的。很多网友曾经在网上提问为何 Borland 公司一直使用 Onject Pascal 来编写 VCL，而不使用 C++来编写 VCL，我想一方面的原因就是使用 C++编写 VCL 将必需改变这种保持完整性的逻辑结构，可能会导致兼容性难以保证。这也是读者在

编写组件时需要注意的地方，上面的分析说明如何保证完整性，但是并不能一成不变的拿到 C++ Builder 中来使用，只能通过其它的方式解决这种冲突。

关于 TDataSource 组件更深入的了解。读者可以在 C++ Builder 的源码中找 DB.pas 文件，参考其中代码详细分析。

5 小结

本章主要从控件的编写方面对如何使用面向状态的程序设计进行了一些示范性讲解，其中的例子覆盖了不同方面的应用和不同的语言使用技巧，C++ Builder 的博大精深是远远超过这些的，而且作者的水平也只限于目前的层次，这些例程只是希望能给读者一个抛砖引玉的作用。对于 C++ Builder 的使用很大程度依赖于对 VCL 的了解，本书也只能提供给读者有限的信息。

在本章中并没有按照第三章中所讲的抽象性、封装性、约束性和完整性等特征来分析和设计控件，一方面是因为本章的控件非常简单，主要是提供对语言掌握的一些示例，牵扯到复杂关系的属性比较少；另一方面是正象第三章小结中所说的，对于一种理论的使用，不能局限在这几个字眼上，而是将抽象性、封装性、约束性和完整性等特性在阅读第三章时尽可能多的理解消化，而在使用面向状态的时候，尽可能的不要被这几个特征束缚住手脚，这样才能灵活的运行；再还有就是对于作者胡言乱语的所谓面向状态的设计方法，在分析过程上，包括作者目前也并没有非常完善的分析方法，因此不便在此枉加宣扬，还望读者见谅。