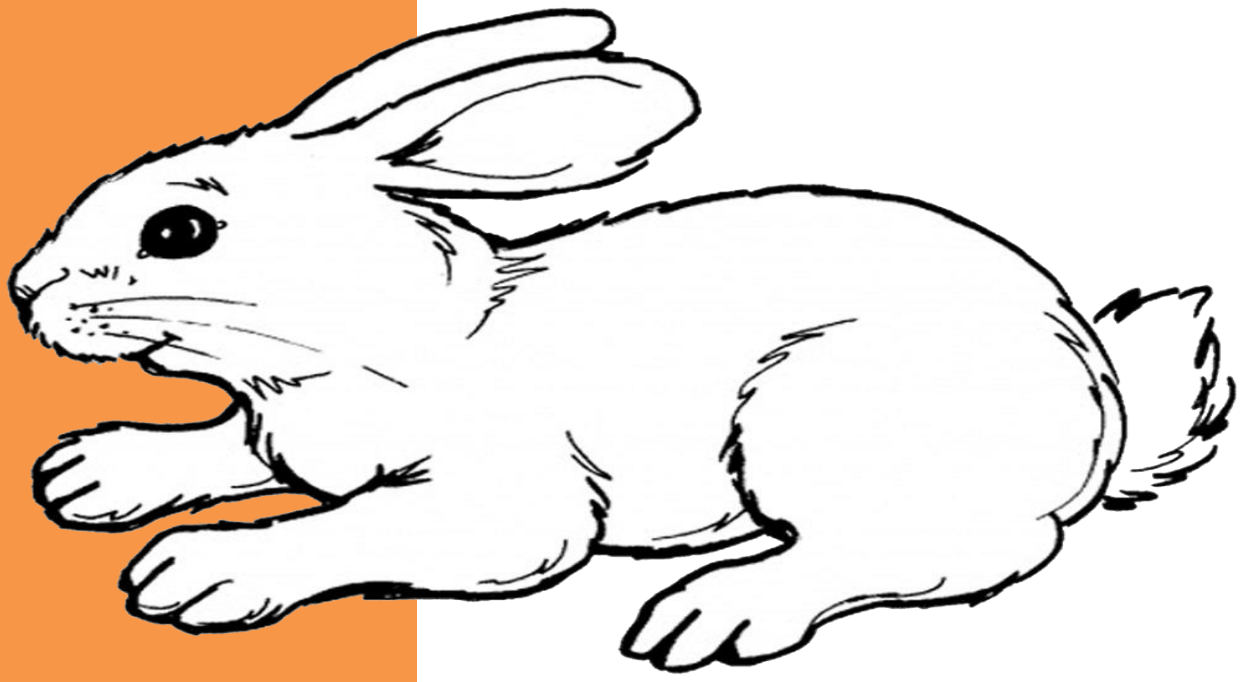
.NET 控件开发基础



汪 明　　著

目　录

第一章　.NET 基础 1

1.1 类和对象 1

1.1.1 类的用途 1

1.1.2 类和类型 3

1.2 堆栈和变量类型 8

1.2.1 Stack 和 Heap 8

1.2.2 值类型和引用类型 9

1.3 UI 类 11

1.3.1 控件 11

1.3.2 组件 14

1.3.3 控件 Z-Index 15

1.4 自定义事件 16

1.5 本章小结 18

第二章　控件基础 19

2.1 控件的概念 19

2.2 控件的类型 19

2.3 控件的继承 20

2.3.1 继承 Windows 窗体控件 20

2.3.2 继承 UserControl 20

2.3.3 继承 Control 21

2.4 控件设计时属性 22

2.4.1 字段和属性 23

2.4.2 Attribute 用法 25

2.4.3 自定义 Attribute 26

2.5 控件设计时支持 27
2.6.1 UITypeEditor 28
2.6.2 TypeConverters 32
2.6.3 Custom Designer 35
2.6 颜色 37
2.7 字体 40
2.8 鼠标和键盘 42
2.8.1 键盘 42
2.8.2 鼠标 45
2.9 本章小结 46
第三章 GDI+基础 47
3.1 GDI+用途 47
3.2 GDI+绘制 48
3.3 绘制和重绘 50
3.4 刷新 52
3.5 大小调整与重绘 53
3.6 Graphic 类 55
3.7 GraphicPath 类 61
3.8 Region 类 64
3.9 坐标体系和变换 67
3.10 双缓冲 73
3.11 局部刷新 73
3.12 命中测试 75

3.13 不规则窗体 77
3.14 本章小结 78
第四章 Form 控件开发 81
4.1 LabelTextBox 控件 81
4.1.1 控件功能 81
4.1.2 控件设计 81
4.1.3 控件开发 82
4.1.4 控件应用 88
4.2 IPTextBox 控件 89
4.2.1 控件功能 90
4.2.2 控件设计 90
4.2.3 控件开发 91
4.2.4 控件应用 96
4.3 ChineseMoneyTextBox 控件 97
4.3.1 控件功能 97
4.3.2 控件设计 98
4.3.3 控件开发 98
4.3.4 控件应用 100
4.4 ImageTextBox 控件 102
4.4.1 控件功能 102
4.4.2 控件设计 102
4.4.3 控件开发 103
4.4.4 控件应用 111
4.5 ImageButton 控件 113
4.5.1 控件功能 113
4.5.2 控件设计 113
4.5.3 控件开发 113

4.5.4 控件应用 118
4.6 IconCheckBox 控件 119
4.6.1 控件功能 119
4.6.2 控件设计 119
4.6.3 控件开发 120
4.6.4 控件应用 124
4.7 ToggleButton 控件 125
4.7.1 控件功能 125
4.7.2 控件设计 125
4.7.3 控件开发 125
4.7.4 控件应用 133
4.8 IconCaptionPanel 控件 135
4.8.1 控件功能 135
4.8.2 控件设计 135
4.8.3 控件开发 136
4.8.4 控件应用 142
4.9 IconTabControl 控件 142
4.9.1 控件功能 143
4.9.2 控件设计 143
4.9.3 控件开发 143
4.9.4 控件应用 148
4.10 IconMessageBox 控件 150
4.10.1 控件功能 150
4.10.2 控件设计 150
4.10.3 控件开发 151
4.10.4 控件应用 156
4.11 FlatRoundImage 控件 157
4.11.1 控件功能 158

4.11.2 控件设计 158
4.11.3 控件开发 158
4.11.4 控件应用 161
4.12 FlatDateTimePicker 控件 163
4.12.1 控件功能 163
4.12.2 控件设计 163
4.12.3 控件开发 163
4.12.4 控件应用 166
4.13 FlatDataGridView 168
4.13.1 控件功能 168
4.13.2 控件设计 168
4.13.3 控件开发 168
4.13.4 控件应用 170
4.14 本章小结 171
第五章 Form 高级主题 173
5.1 数据库交互 173
5.1.1 强类型数据集 174
5.1.2 UI 数据绑定 174
5.1.3 数据操作方法 175
5.2 反射 178
5.3 插件机制 180
5.3.1 定义接口 181
5.3.2 插件加载 181
5.4 动态属性 183
5.6 C#闭包 185
5.7 C#扩展方法 186
5.7.1 变量前缀$ 186

5.7.2 正则表达式捕获变量 187
5.7.3 用反射获取属性的值 187
5.7.4 string 方法扩展实现 188
5.8 C#方法链 189
5.9 C#动态编译 191
5.10 本章小结 197

第一章 .NET 基础

在正式介绍Windows Forms自定义控件开发之前，有必要简要介绍一下.NET相关基础知识，如果你对这部分内容非常了解，可以跳过本章内容，但是如果你不是特别熟悉，也希望你能耐心的阅读本章内容，这对于你掌握后续章节的内容有一定的帮助。

1.1 类和对象

在面向对象编程相关课程中，基本上都会提及类和对象。.NET中的C#是一种面向对象的语言，在C#语言中类和对象是最基本的概念。一般来说，类将数据（一般表现为属性和字段）以及这些数据上的操作（一般表现为方法或函数）封装在一起。类是对象的抽象，而对象是类的具体实例。类、对象、实体和现象之间的关系如图1.1所示：

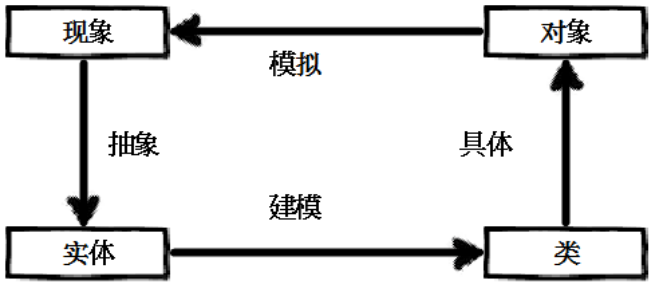


图 1.1 类、对象、实体和现象之间的关系

1.1.1 类的用途

在.NET中，类可以用New方法来创建，从创建方法上来说，不同类的创建几乎没有任何区别。但是在不同的应用场景中，不同的类所承担的作用却不同。一般来说，类最常用的几种用途为：

- 类可以对现实实体进行建模

现实世界中有很多的实体，例如人、购物单、物料等。在面向对象编程中，可以用类对实体进行建模，从而来实例化出各种具体的实体对象。例如现实世界中的人，一般具备名称、年龄、性别和身份证号码等信息，那么在计算机世界中，可以通过一个Man类对现实中的人实体进行建模，Man类代码如下所示：

```
public class Man
{
      public string Name { get; set; }
      public int Age { get; set; }
      public string Sex { get; set; }
      public string ID { get; set; }
}
```

■ 类可以作为可复用的编程抽象

这是类最常用的一种用途。例如TextBox类就可以表示文本框控件，TextBox类不与具体的文本对象直接进行关联，而是将文本相关的属性和方法进行了封装，从而可以通过配置属性和方法来创建出各种文本效果。通过TextBox类可以创建不同内容和颜色的文本，代码如下所示：

```
TextBox textBox1 = new TextBox();
textBox1.ForeColor = Color.Red;
textBox1.Text = "Hello";
TextBox textBox2 = new TextBox();
textBox2.ForeColor = Color.Black;
textBox2.Text = "world";
textBox2.Focus();
```

■ 类可以整合相关的函数

某些类仅仅是一些静态方法的集合体，我们不需要创建该类的实例就可以使用它。在绝大部分的应用程序中，都有一个DBHelper静态类，里面封装了大量的静态方法来处理程序与数据库交互的逻辑，可以直接进行调用。下面的ASCIICode静态类实现ASCII码转字符和字符转ASCII码的方法，代码如下：

```
public static class ASCIICode
{
       // ASCII码转字符
       public static string ASCIIChr(int asciiCode)
       {
           if (asciiCode >= 0 && asciiCode <= 255)
           {
               System.Text.ASCIIEncoding asciiEncoding =
               new System.Text.ASCIIEncoding();
               byte[] byteArray = new byte[] { (byte)asciiCode };
               string strCharacter =
asciiEncoding.GetString(byteArray);
               return (strCharacter);
           }
           else
```

```
            {
                throw new Exception("ASCII码无效");
            }
        }
        //字符转ASCII码
        public static int Asc(string character)
        {
            if (character.Length == 1)
            {
                System.Text.ASCIIEncoding asciiEncoding =
                 new System.Text.ASCIIEncoding();
                int intAsciiCode =
(int)asciiEncoding.GetBytes(character)[0];
                return (intAsciiCode);
            }
            else
            {
                throw new Exception("字符无效");
            }
        }
 }
```

1.1.2 类和类型

在.NET中除了上面提到类和对象的基本概念之外，还有一个非常重要的概念，那就是类型(Type)，类型包含：

■ 结构

结构（Struct）和类不同，它是值类型，而类是引用类型。当我们将一个Struct变量赋值给另一个Struct变量时，实际上.NET内部是将该Struct的内容复制到另一个对象。当比较两个Struct变量时，是对其内容进行比较，下面的代码演示了Struct变量是值类型，而非引用类型，当变量赋值后进行了属性修改（ms2.ID = 2），但不影响原有变量的属性值（ms1.ID=1）：

```
myStruct ms1 = new myStruct();
ms1.ID = 1;
ms1.Name = "jack";
myStruct ms2 = ms1;
ms2.ID = 2;//ms1.ID=1
```

■ 类

类（Class）是引用类型，当两个类变量进行赋值后，会指向同一个对象，修改其中一

个类变量的属性会同时修改另外一个类变量的同一属性。下面的代码演示了Class变量是引用类型，而非值类型，当Class变量赋值后进行了属性修改（mc2.ID = 2），则影响原有变量的属性值（mc1.ID=2）：

```
myClass mc1 = new myClass();
mc1.ID = 1;
mc1.Name = "jack";
myClass mc2 = mc1;
mc2.ID = 2;//mc1.ID=2
```

但必须注意的是，类可以通过重载其默认的引用类型行为。例如String是一个类，但是.NET重载了赋值和判断相等的操作，从而让String类像值类型一样工作。下面的代码可以看到String类的行为是值类型，当赋值后，原有String变量的值没有变化：

```
string s1 = "hello";
string s2 = s1;
s2 = "world";//s1 = "hello"
String s3 =new String("hello".ToCharArray());
String s4 = s3;
s4 = "world";//s3="hello"
```

Array类是引用类型，当进行Array类的变量赋值时，也是指向同一个对象，修改一个变量的值会影响到另一个变量的值，代码如下：

```
int[] arr1 = new int[] { 1, 2, 3, 4 };
int[] arr2 = arr1;
arr2[0] = 6;//arr1[0]=6
```

■ 委托

MSDN中对委托Delegate的解释为：委托类似于C或C++中的函数指针。使用委托使程序员可以将方法引用封装在委托对象内。然后可以将该委托对象传递给可调用所引用方法的代码，而不必在编译时知道将调用哪个方法。与C或C++中的函数指针不同，委托是面向对象且类型安全的。这个解释非常拗口，其实委托是指向同一约定契约（一致的函数参数个数和类型及函数返回类型）方法的一种引用类型变量。委托特别用于实现事件和回调方法，指向的函数可在运行时被改变。所有的委托都派生自 System.Delegate 类。为了理解上述解释，我们可以通过一个例子来说明，DelegateClass类中定义了一个int myDelegate(int a, int b)的委托和一个int Sum(int a, int b)函数，其函数契约为int fn函数名(int 变量1, int 变量2)，只要符合这个约定契约的函数都可以被myDelegate委托进行引用。换句话说，符合这个约定契约的函数都可以赋值给myDelegate委托，显然sum函数符合该委托的约定契约，具体示例代码如下：

```
class DelegateClass
{
    //声明一个Delegates
    public delegate int myDelegate(int a, int b);
    //声明一个符合myDelegate的函数
    public int Sum(int a, int b)
    {
        return a + b;
    }
}
class Program
{
    static void Main(string[] args)
    {
        DelegateClass objMyclass = new DelegateClass();
        //创建委托引用
        DelegateClass.myDelegate objSum = null;
        //将委托引用指向Sum函数
        objSum = objMyclass.Sum;
        //调用
        if (objSum != null)
        {
            Console.WriteLine("Sum=" + objSum(5, 8));
            Console.ReadLine();
        }
    }
}
```

委托和事件关系密切。事件是一种特殊的委托，或者说是受限制的委托，只能施加+=和-=操作符。

■ 枚举

枚举enum可以将一组整数常量进行有意义的命名，且可以将命名赋值给变量。如果我们需要定义一个变量，该变量的值表示一周中的一天，且该变量只能存储七个有意义的值（周一到周日）。那么此变量使用枚举进行定义将比较合适。

默认情况下，枚举中每个元素的基础类型是 int。可以使用冒号指定另一种数值类型。如果不为枚举数列表中的元素指定值，则它们的值将以1为增量自动递增。下面我们定义一个TimeOfDay枚举类型来表示一天中的上午，下午和晚上，代码如下：

```
public enum TimeOfDay
{
        Moning = 0,
        Afternoon = 1,
```

```
        Evening = 2,
    };
```

我们可以用枚举变量类型的ToString()来获取枚举的文本：

```
TimeOfDay time = TimeOfDay.Afternoon;
//输出 Afternoon
Console.WriteLine(time.ToString());
```

我们也可以根据给定的枚举类型的文本值，通过Enum.Parse方法来获取枚举对应文本的值：

```
TimeOfDay time2 = (TimeOfDay)
Enum.Parse(typeof(TimeOfDay), "afternoon", true);
Console.WriteLine((int)time2);//输出1
```

类似的，可以根据给定的枚举类型的值，通过Enum.GetName方法来获取指定值对应的文本值：

```
string enumName = Enum.GetName(typeof(TimeOfDay), 0);
```

最后，我们用Enum.GetNames方法来获取所有的枚举类型的文本：

```
string names = "";
foreach (string name in Enum.GetNames(typeof(TimeOfDay)))
{
    names += name;
}
```

在软件的权限管理中，常常会出现权限相互包含的现象。一般来说权限有View、Add、Delete、Save和All，其中All = View | Add | Delete | Save，All可集合其他所有的权限。为了让枚举类型支持All = View | Add | Delete | Save操作，需要在枚举类型类型上用[Flags]进行标注：

```
[Flags]
public enum AccessRights
{
    View = 0x01,
    Add = 0x02,
    Delete = 0x04,
    Save = 0x08,
    All = View | Add | Delete | Save
}
//声明枚举变量
var rightView = AccessRights.View;
var rightAll = AccessRights.All;
if (rightAll.HasFlag(rightView))
{
     Console.WriteLine("Has View Right");
```

```
}
else
{
    Console.WriteLine("Has No View Right");
}
```

■ 接口

接口Interface只包含方法、委托或事件的签名。换句话说，接口只是负责完成定义的操作，而不去实现具体的细节。接口除了可以包含方法之外，还可以包含属性、索引器、事件等，而且这些成员都被定义为公有的。除此之外，不能包含任何其他的成员，例如：常量、域、构造函数、析构函数、静态成员。一个类可以直接继承多个接口，但只能直接继承一个类（包括抽象类）。从类型上来说，接口是引用类型的，类似于类。接口和抽象类的相似之处有三点：

1、不能实例化；

2、包含未实现的方法声明；

3、派生类必须实现未实现的方法，抽象类是抽象方法，接口则是所有成员；

下面我们定义一个设备接口IDevice，设备接口有一个方法为Read()，代码如下：

```
public interface IDevice
{
    void Read();
}
```

然后定义继承IDevice 接口的USB类和HDD类：

```
public class USB : IDevice
{
    public void Read()
    {
        Console.WriteLine("Read USB");
    }
}
public class HDD : IDevice
{
    public void Read()
    {
        Console.WriteLine("Read HDD");
    }
}
```

有了上述的接口和类后，我们可以定义一个设备控制器类，该类负责对各种继承IDevice接口的硬件进行读取操作：

```
public class DeviceController
{
    private IDevice device;
    public DeviceController(IDevice device)
    {
        this.device = device;
    }
    public void Read()
    {
        device.Read();
    }
}
DeviceController deviceCtrl = new DeviceController(new USB());
deviceCtrl.Read();
deviceCtrl = new DeviceController(new HDD());
deviceCtrl.Read();
```

接口在各类系统中都有大量的应用，特别对于框架型的类库中，一般都要约定接口内容，方便各模块进行耦合。接口是一组规范，有了规范程序才能更好的协作。

1.2 堆栈和变量类型

对.NET 基本概念的理解和掌握对于提升编程水平来说非常重要。上面介绍了类、类型、枚举、委托等基础概念，这一节我们对.NET 的 Stack(栈)和 Heap(堆)进行阐述。

计算机的内存可以分为代码块内存、stack 内存和 heap 内存。代码块内存是在加载程序时存放程序机器代码的地方。Stack 一般存放函数内的局部变量。而 heap 存放全局变量和类对象实例等。若只是声明一个对象，则先在栈内存中为其分配地址空间，若再实例化它，则在堆内存中为其分配空间。

1.2.1 Stack 和 Heap

由于计算机的内存分配过程比较抽象，下面举一个简单的程序片段，来图解不同类型（值类型和引用类型）的变量创建对 Stack 和 Heap 内存的影响，如图 1.2 所示：

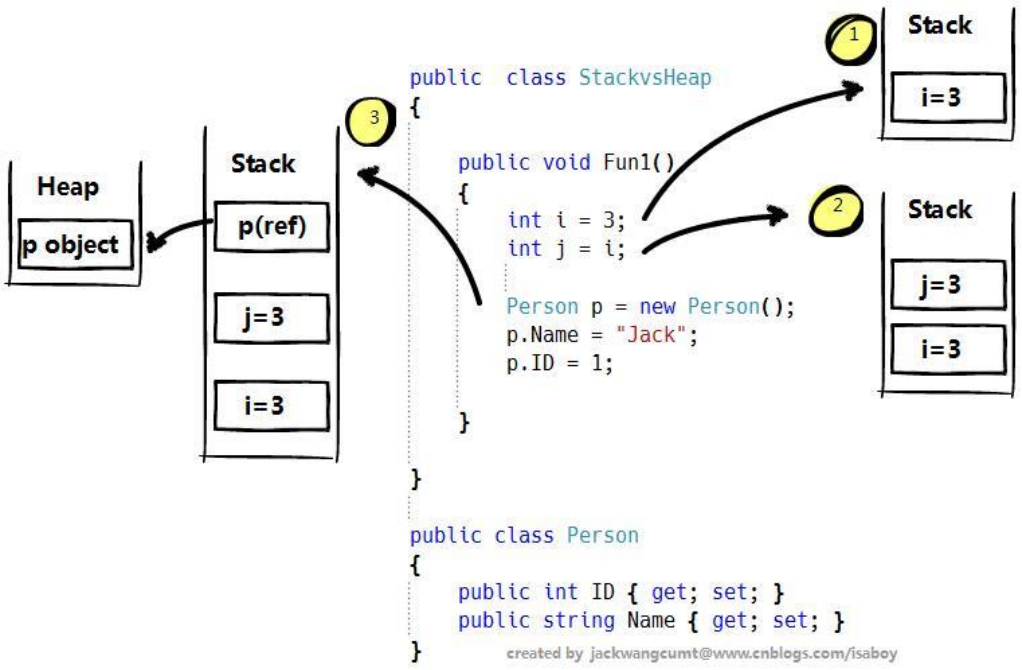


图 1.2 Stack 和 Heap

假设我们调用 StackvsHeap 类中的 Fun1 方法，当执行第一行代码,即 int i=3;由于在.NET 中，除了 string（表现为值类型）、object、class、delegate 和 interface 外，其他的类型为数值类型，一般（不是全部）存放在 Stack 内存中。此处 int i=3 是函数内的非静态变量，而数值型为非引用类型，即会在 Stack 内存中分配一个区域来存放该变量。

当执行第二句语句，即 int j=i;.NET 也会在 Stack 内存中分配一个区域来存放该变量，而且地址块在 i=3 之上（LIFO）。

当执行第三句，即 Person p = new Person();时，我们可以分为两步来看：

1）在 Stack 上分配一个 Person 类型的 p 引用变量（指向 Heap 上的地址）；

2）在 Heap 上分配一个空间来存储 Person 类的实例数据；

1.2.2 值类型和引用类型

理解了上面的过程，现在理解值类型和引用类型变量的区别则更加容易，如图 1.3 所示：

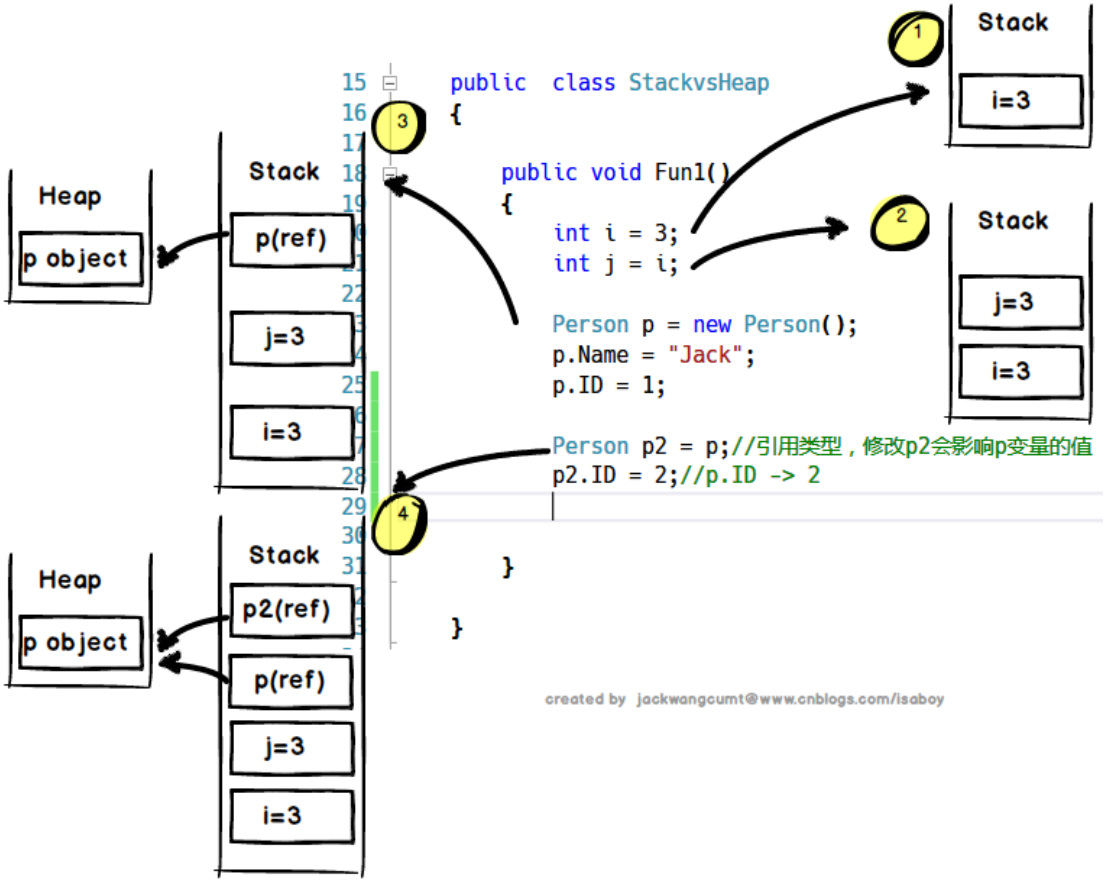


图 1.3 值类型和引用类型

由于 int j=i 中是 int 类型，为值类型变量，则 j=3 为 i=3 的拷贝，因此，修改 i 不会修改 j，修改 j 也不会修改 i;

而 Person p2=p 中 Person 为类，是引用类型，因此 p2 和 p 指向同一个 heap 地址块，因此，修改 p2 的值会影响 p 的值。

下面给出常见.NET 类型对应的类型，如表 1.1 所示：

表 1.1 值类型和引用类型一览

类别	类型	描述
值类型	简单类型	整形 Int
		字符型 char
		浮点型 float 和 double
		高精度小数:decimal
		布尔型:bool
	枚举类型	Enum

	结构类型	Struct
引用类型	类类型	基类 Object 为的所有类型
		string
		class
	接口类型	Inteface
	数值类型	Int[],string[]等
	委托类型	delegate

1.3 UI 类

1.3.1 控件

在.NET 中，所有的控件都是类，所有的窗体控件都在 System.Windows.Forms 命名空间下。控件类和其他类一样，都可以实例化并设置其属性，也可以调用其方法等，但和普通的类不同的是，所有的窗体控件追溯到源头，都是继承自 System.Windows.Forms.Control 基类，这使得控件具备一些特有的基本功能，例如绘图和鼠标相关的事件。

其实，控件类之所以能根据属性配置来绘制 UI，是因为 Windows Forms 引擎能够识别控件的相关信息并进行渲染。Windows Forms 引擎可以根据操作系统的消息来操作窗体，其中包含对控件的属性进行设置和调用控件的方法来绘制 UI。虽然这些过程 Windows Forms 引擎会自动帮我们处理，但是其中的一些属性和事件等依然对用户公开，并允许我们进行重载，从而使得控件从样式和内容上都可以被定制。

另外，控件之间存在上下级的层级关系，控件可以相互包含，由于所有窗体控件的基类都是 System.Windows.Forms.Control，在 System.Windows.Forms.Control 类中提供了一个 Controls 属性，该属性是个集合类型，可以包含其他的控件。如下图 1.4 所示：

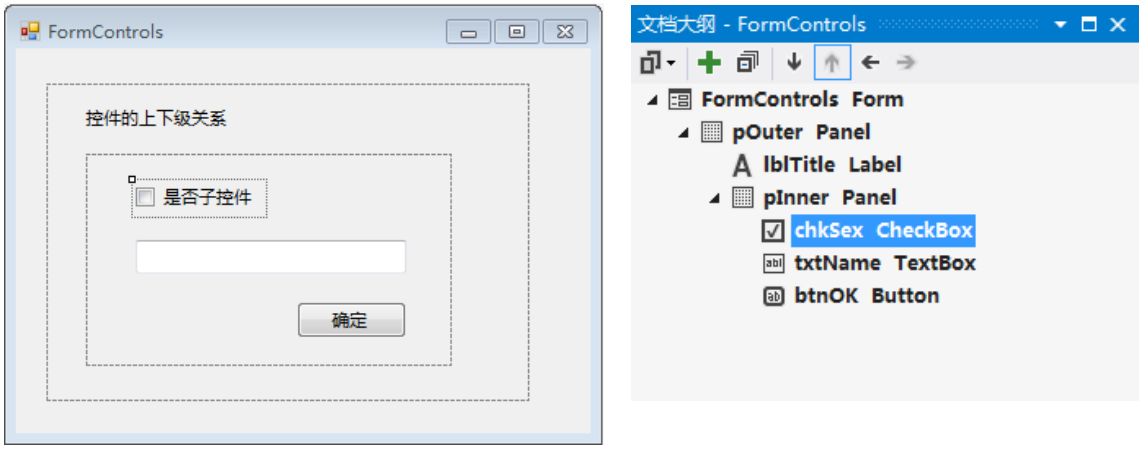


图 1.4 控件层级关系

在 Visual Studio IDE 中用文档大纲可以查看控件的上下级关系。我们可以通过控件的 Controls 属性中的 Add 和 Remove 方法来实现控件的添加和删除。下面的代码演示了在窗体 frmName 以及 frmName 窗体的 pContain 面板控件上对子控件的操作：

```
frmName.Controls.Add(txtTextBox);
frmName.pContain.Controls.Add(txtTextBox);
frmName.pContain.Controls.Remove (txtTextBox);
```

控件在使用过程中，为了动态的修改其属性或调用其方法，必须要对控件进行访问，其中每个控件都有一个 Name 属性唯一标识一个实例化的控件。

```
//遍历控件，移除名称为txtDept的控件
foreach (Control ctrl in Controls)
{
    if (ctrl.Name == "txtDept")
    {
        Controls.Remove(ctrl);
    }
}
```

下面用递归方法来清空文本框的文本：

```
private void ClearControls(Control topControl)
{
    if (topControl.GetType() == typeof(TextBox))
    {
        topControl.Text = "";
    }
    else
    {
        foreach (Control childControl in topControl.Controls)
        {
            ClearControls(childControl);
        }
```

```
    }
}
ClearControls(this);
```

下面给出控件的基本属性和方法，如表1.2所示：

表 1.2 控件的基本属性和方法

属性或方法	描述
Name	控件的简要名称，可作为控件的唯一标识
Tag	可存放 object 对象，一般用于存储控件的额外信息
Controls	存放控件所有子控件的集合
Invoke()	多线程编程中使用，在拥有此控件的基础窗口句柄的线程上执行指定的委托。
InvokeRequired	多线程编程中使用，获取一个值，该值指示调用方在对控件进行方法调用时是否必须调用 Invoke 方法，因为调用方位于创建控件所在的线程以外的线程中。
DesignMode	获取一个值，用以指示 Component 当前是否处于设计模式。
Disposing	获取一个值，该值指示 Control 基类是否在释放进程中。
DataBindings	为该控件获取数据绑定。
Cursor	获取或设置当鼠标指针位于控件上时显示的光标。
Parent	获取或设置控件的父容器。
Size	获取或设置控件的高度和宽度。
TopLevelControl	获取没有另一个 Windows 窗体控件作为其父级的父控件。通常，这是控件所在的最外面的 Form。
Width	获取或设置控件的宽度。
IsDisposed	获取一个值，该值指示控件是否已经被释放。
Height	获取或设置控件的高度。
DoubleBuffered	获取或设置一个值，该值指示此控件是否应使用辅助缓冲区重绘其图面，以减少或避免闪烁。
Font	获取或设置控件显示的文字的字体。
HasChildren	获取一个值，该值指示控件是否包含一个或多个子控件。
ForeColor	获取或设置控件的前景色。

BackColor	获取或设置控件的背景色。
Bounds	获取或设置控件（包括其非工作区元素）相对于其父控件的大小和位置（以像素为单位）。
AllowDrop	获取或设置一个值，该值指示控件是否可以接受用户拖放到它上面的数据。
Capture	获取或设置一个值，该值指示控件是否已捕获鼠标。
ClientRectangle	获取表示控件的工作区的矩形。
GetChildAtPoint()	检索位于指定坐标处的子控件。
Contains()	
Invalidate()	具有多个重载，可以使控件的整个图面无效并导致重绘控件，也可以重绘局部区域。
ControlAdded	在将新控件添加到 Control.ControlCollection 时发生，可以用于界面自动布局。
ControlRemoved	在从 Control.ControlCollection 移除控件时发生，可以用于界面自动布局。
Update()	使控件重绘其工作区内的无效区域。
UpdateZOrder()	按控件的父级的 Z 顺序更新控件。
WndProc(Message)	处理 Windows 消息。

1.3.2 组件

在.NET 中，可以放在窗体上的不是只有控件，组件也可以放到窗体上。当然组件是"不可见"的，在窗体上不占用实际的空间。 其实控件是一种特殊的组件，控件继承自组件。组件有 2 个必备的特征：一个是能在设计时进行配置，另外一个就是必须要能释放资源。组件的继承关系如图 1.5 所示：

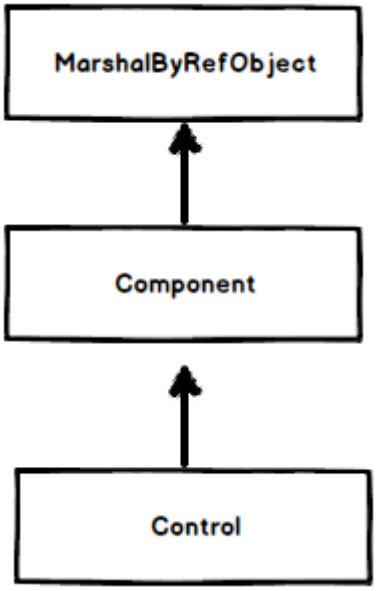


图 1.5 控件的继承关系

1.3.3 控件 Z-Index

在.NET 中窗体开发中，有些控件在布局上会重叠，一般来说，在窗体设计时，先拖放到窗体上的控件会在底层，后拖放到窗体上的会在顶层。有时候要达到某种效果需要动态调整控件的显示顺序，那么此时必须要用到 Controls.SetChildIndex(ChildControl, Z-Index)来设置。

下面看一个例子，如图 1.6 所示。默认的时候左边方块 pictureBox2 在底层，右边的方块 pictureBox3 在顶层。下面通过以下代码进行设置将二者的位置进行变换：

```
this.Controls.SetChildIndex(this.pictureBox2, 0);
this.Controls.SetChildIndex(this.pictureBox3, 1);
```

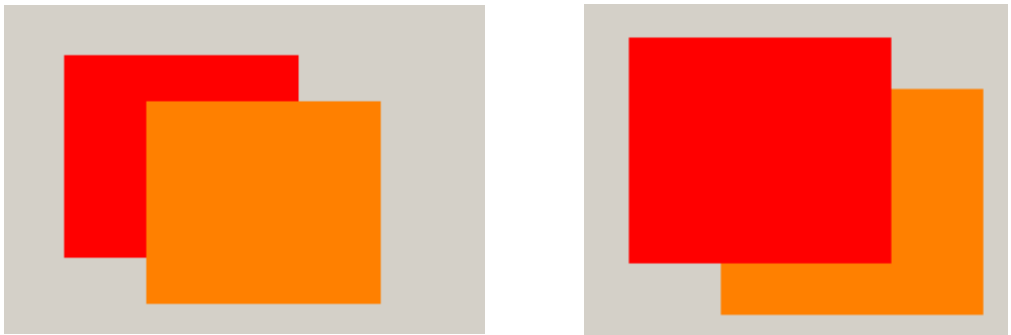


图 1.6 控件的 Z-Index

1.4 自定义事件

Windows Forms 事件属于一种发布订阅模式 。订阅发布模式定义了一种一对多的依赖关系，让多个订阅者对象同时监听某一个主体对象。这个主体对象在自身状态发生变化时，会通知所有订阅者对象，使它们能够自动更新自己的状态。 当一个对象的改变需要同时改变其他对象，而且无需关心具体有多少对象需要改变时，就特别适合用此种模式。下面将演示如何在窗体上自定义一个事件(custom event)：

一般自定义的事件都有一个参数，继承自 EventArgs。此处我们自定一个 CustomEventArgs 类，通过自定义字段来存储参数的值：

```
public class CustomEventArgs:EventArgs
{
    //自定义字段用于存储值
    public object Tag;
    public string Message;
    public CustomEventArgs()
    {
    }
    public CustomEventArgs(string message, object tag)
    {
        Message = message;
        Tag = tag;
    }
}
```

然后我们在 userControlAttribute1 控件中，定义一个 OnProperChanged 的自定义事件，在控件的属性 NickName 变化时进行事件触发：

```
[
BrowsableAttribute(true),
BindableAttribute(false),
CategoryAttribute("自定义事件"),
DescriptionAttribute("当属性(Age和NickName)值变化时触发该事件")
]
//自定义事件
public event EventHandler<CustomEventArgs> OnProperChanged;
private string _name = "";
[
BrowsableAttribute(true),
BindableAttribute(false),
CategoryAttribute("自定义事件"),
```

```
DescriptionAttribute("姓名")
]
public string NickName
{
    get { return _name; }
    set {
        if (value != _name)
        {
            if (OnProperChanged != null)
            {
                //old value vs new value
                CustomEventArgs customEventArgs =
                    new CustomEventArgs(_name, value);
                //触发事件
                OnProperChanged(this, customEventArgs);
            }
            _name = value;
        }
    }
}
```

最后在窗体上添加一个 userControlAttribute1 控件，并在窗体加载事件中将属性值进行赋值并且订阅 OnProperChanged 事件。由于赋值和默认值不同且有对象订阅该事件，就会调用事件 OnProperChanged,同时将控件的属性 NickName 和 Age 进行属性值变更，在 OnProperChanged 事件中，我们可以访问 CustomEventArgs 自定义事件参数的相关值，例如 e.Message 和 e.Tag 分别表示旧值和新值：

```
private void Form4_Load(object sender, EventArgs e)
{
    //属性值改变
    this.userControlAttribute1.Age = 26;
    this.userControlAttribute1.NickName = "jackwang";
}
//OnProperChanged事件
private void userControlAttribute1_OnProperChanged(object sender,
CustomEventArgs e)
{
    object oldvalue = e.Message;
    object newvalue = e.Tag;
}
```

1.5 本章小结

本章主要就.NET 中的基本概念做了一个介绍，并比较了 C#中值类型变量和引用类型的区别，二者在堆栈分配上是不同的。值类型变量赋值后两个变量是独立的对象，修改一个后不影响另一个的值。而引用类型赋值后二者指向同一个对象，修改其中一个会影响另一个。

另外简要介绍了 UI 类，其中包含控件和组件等，最后讲解了如何实现自定义事件。理解和掌握本章的知识对于理解后续章节是有一定帮助的。下一章我们重点讲解控件的相关知识。

第二章　控件基础

在开始正式介绍如何开发自定义控件之前，有必要先了解一下控件开发的基础知识。下面从控件的概念、分类和开发模式上对控件做一个基本的概述。

2.1 控件的概念

所谓的控件（Control）就是对数据和方法的封装。一般来说，控件都有自己的属性、方法和事件。控件的属性是存储控件数据的容器，控件的方法则是实现控件所需的功能。为了提高开发的效率和界面的标准化，很多软件公司都有一套自己封装的 UI 控件，内部员工（程序员）大部分时间都是在使用这些控件，很少有机会来开发这些自定义控件。

控件的设计和开发是一项比较繁重的工作，同时对程序员的要求也比较高，需要掌握 GDI+编程、事件模型、Windows API 和面向对象开发等知识。一个使用比较方便的控件，其背后往往是大量的代码。虽然开发自定义控件是一个比较复杂的过程，但也是一个一劳永逸的过程。开发控件的最大意义在于封装重复的工作，其次是可以扩充现有控件的功能。

另外，和控件概念容易混淆的是组件（Component）。一般来说，控件是具有用户界面的可视化的组件。例如窗体中的文本框、列表框。而组件一般用户逻辑处理，比如实现数据绑定、逻辑计算和计时触发等功能。

2.2 控件的类型

以 Windows Forms 控件来说，控件通常有三种类型：

- 复合控件（Composite Controls）

复合控件将现有的各种控件组合起来，形成一个新的控件，将多个控件的功能集中起来。

- 扩展控件（Extended Controls）

扩展控件是在现有控件的控件的基础上派生出一个新的控件，为原有控件增加新的功能或者修改原有控件的功能。

- 自定义控件（Custom Controls）

自定义控件直接从 System.Windows.Forms.Control 类派生出来。Control 类提供控件所需要的所有基本功能，包括键盘和鼠标的事件处理。自定义控件是最灵活最强大的

方法，但是对开发者的要求也比较高，你必须为 Control 类的 OnPaint 事件写代码，你也可以重写 Control 类的 WndProc 方法，处理更底层的 Windows 消息，所以你必须掌握一定的 GDI＋和 Windows API 知识。 后面的章节将重点介绍 GDI+相关知识，并用案例分别讲解如何自定义控件。Custom Controls 自定义控件称为自绘制控件（Own-Drawing Controls）比较合适。本书的自定义控件是一个广泛的概念，包含上面的三种控件类型。

2.3 控件的继承

一般来说，控件的开发不是从零开始的，而是根据需要实现的控件功能，来选择合适的控件类型，再根据控件类型来继承相关的类来实现，这样可以大大降低开发控件的难度和成本。

2.3.1 继承 Windows 窗体控件

自定义控件可以继承自任何现有的 Windows 窗体控件，此方法可以保留 Windows 窗体控件所有已有的功能和属性，控件开发人员只需要添加自定义的属性、方法或其他功能扩展即可。例如，如果现在需要自定义一个只接受数值型的数值文本框，那么开发此数值文本框可以创建一个从 TextBox 派生的控件，并为此控件添加一个验证输入是否为数值的判断方法，每当文本框中的文本发生更改时就会调用此方法，另外如果需要定制数值文本框的外观，可以通过重写基类 TextBox 的 OnPaint 方法将自定义外观添加到此控件上。

一般来说，处于下列情况时建议从 Windows 窗体控件继承：

- 大多数所需的功能已经与现有的 Windows 窗体控件相同
- 不需要自定义图形接口，或者想为现有控件设计一个新的外观

2.3.2 继承 UserControl

用户控件（UserControl）是封装在公共容器内的 Windows 窗体控件的集合。此容器包含与每个 Windows 窗体控件相关联的所有固有功能，允许控件开发人员有选择地公开和绑定内部的属性。例如，如果现在需要开发一个 IP 文本框控件，只允许输入 IP 地址。那么开发此 IP 控件可以通过继承用户控件，然后在用户控件上添加 4 个 TextBox，在 TextBox 中间放入一个 Label（设置其 Text 为.的符号），再通过检测每个文本框的文本变更时用户

输入字符的合法性，即用户输入的字符只允许为 0~255。

一般来说，如果要将若干个 Windows 窗体控件的功能合成一个可重新使用的单元，则建议从 UserControl 类继承。

2.3.3 继承 Control

创建一个新的控件，除了上面介绍的继承自现有窗体控件和用户控件类以外，也可以通过继承 Control 类从头开发一个控件。Control 类提供控件（例如事件）所需的所有基本功能，但不提供控件特定的功能或图形界面。与通过从用户控件或现有 Windows 窗体控件继承创建控件相比，通过从 Control 类继承创建控件需要耗费更多的心思和精力。因为控件开发人员必须为控件的 OnPaint 事件编写代码以及所需的任何功能特定代码，但同时也允许控件开发人员根据自己的需要，灵活地自定义调整控件。例如，如果需要开发一个时钟控件，此控件模拟时钟的外观和操作，通过添加 Timer 组件，并响应内部计时器组件的 Tick 事件来每秒调用自定义绘图方法来模拟时钟指针运动。

一般来说，处于下列情况时建议从 Control 类继承：

- 想要提供控件的自定义 UI 样式
- 需要实现无法从标准控件获得的自定义功能

在 Viusal Studio IDE 中，我们可以通过 Windows Forms 下的模板创建上述三种类型的控件，如下图 2.1 所示：

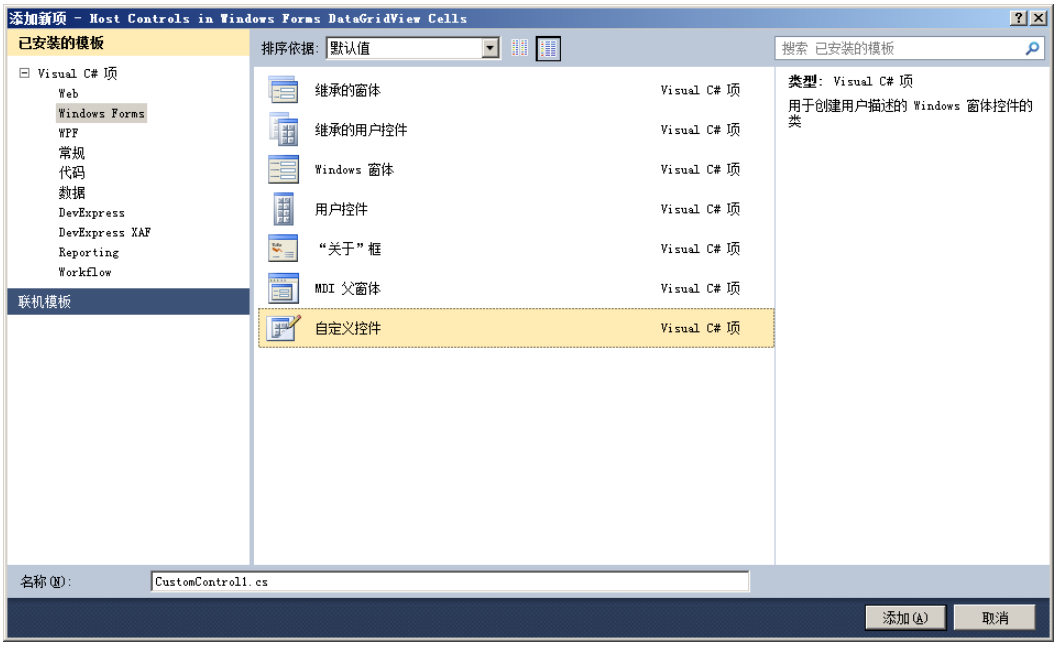


图 2.1 Viusal Studio Windows Forms 模板

2.4 控件设计时属性

属性 Attribute 类将预定义的系统信息或用户定义的自定义信息与目标元素相关联。目标元素可以是程序集、类、构造函数、委托、枚举、事件、字段、接口、方法、可移植可执行文件模块、参数、属性 (Property)、返回值、结构或其他属性 (Attribute)。

属性 Attribute 所提供的信息也称为元数据。元数据可由应用程序在运行时进行检查以控制程序处理数据的方式，也可以由外部工具在运行前检查以控制应用程序处理或维护自身的方式。

所有属性 Attribute 类型都直接或间接地从 Attribute 类派生。属性可应用于任何目标元素。多个属性可应用于同一目标元素。使用 AttributeTargets 类可以指定属性所应用到的目标元素。

在控件开发过程中，自定义的属性 (Property) 或者自定义的事件（Event）往往需要添加设计时属性 Attribute，下表 2.1 对常用的设计时 Attribute 进行了说明：

表 2.1 设计时 Attribute 一览

属性 (Attribute)	应用于	说明
BrowsableAttribute	属性和事件	指定属性 (Property) 或事件是否应该显示在属性 (Property) 浏览器中。
CategoryAttribute	属性和事件	指定类别的名称，在该类别中将对属性 (Property) 或事件进行分组。当使用了类别时，组件属性 (Property) 和事件可以按逻辑分组显示在属性 (Property) 浏览器中。
DescriptionAttribute	属性和事件	定义一小块文本，该文本将在用户选择属性 (Property) 或事件时显示在属性 (Property) 浏览器底部。
BindableAttribute	属性	指定是否要绑定到该属性 (Property)。
DefaultPropertyAttribute	属性	指定组件的默认属性 (Property)。当用户单击控件时，将在属性 (Property) 浏览器中选定该属性 (Property)。

DefaultValueAttribute	属性	为属性（Property）设置一个简单的默认值。
EditorAttribute	属性	指定在可视设计器中编辑（更改）属性（Property）时要使用的编辑器。
LocalizableAttribute	属性	指定属性（Property）可本地化。当用户要本地化某个窗体时，任何具有该属性（Attribute）的属性（Property）都将自动永久驻留到资源文件中。
DesignerSerializationVisibilityAttribute	属性	指定显示在属性（Property）浏览器中的属性（Property）是否应该（以及如何）永久驻留在代码中。
TypeConverterAttribute	属性	指定将属性（Property）的类型转换为另一个数据类型时要使用的类型转换器。
DefaultEventAttribute	事件	指定组件的默认事件。这是当用户单击组件时在属性（Property）浏览器中选定的事件。

2.4.1 字段和属性

在控件类中，一般都有字段 Fields 和属性 Properties，二者看起来有点类似，但是二者还是有区别的，请看下面的一段代码：

```
//fields
private string name;
private string level;
private bool reviewed;
//property read-only
public string Name
{
    get { return name; }
}
```

一般来说，Fields 和 Properties 的区别有：

- Fields 可以作为 out 和 ref 参数的输入；但是 Properties 不能。
- Properties 在运行时可能会抛出异常；但是 Fields 不会。
- Properties 可能会有副作用或者会消耗比较长的时间来执行；但是 Fields 不会

有副作用并且总是按照给定的类型来得到期望的结果。

■ Properties 支持不同的访问策略（包括验证等），通过 getters 和 setters 实现；但是 Fields 不支持 (但可以设置字段为 Readonly)。

■ 在反射操作时， Properties 和 Fields 是不同的 MemberTypes，所以获取的方法也不同，字段用 GetFields 获取，属性用 GetProperties 获取。

■ 在 WPF 中，数据只能绑定到公共的 Properties；但是绑定到公共的 Fields 不起作用。

由于编译器被显式设计为识别预定义关键字，因此传统上我们没有机会创建自己的关键字。但是，公共语言运行库允许我们添加类似关键字的描述性声明，称为属性 (Attribute) 来批注编程元素，例如类型、字段、方法和属性 (Property)。

按照 MSDN 的说法，在运行库编译代码时，属性 (Attribute) 相关代码被转换为 Microsoft 中间语言 (MSIL)，并同编译器生成的元数据一起被放到可移植可执行 (PE) 文件的内部。属性 (Attribute) 使我们得以向元数据中放置额外的描述性信息，并可使用反射服务来提取该信息。属性 (Attribute) 描述如何将数据序列化，指定用于强制安全性的特性，并限制实时 (JIT) 编译器的优化，从而使代码易于调试。属性 (Attribute) 还可以记录文件名或代码作者，或在窗体开发阶段控制控件和成员的可见性。

可使用属性 (Attribute) 以几乎所有可能的方式描述代码，并以富有创造性的新方式影响运行库行为。使用属性可以向 C#或其他任何以运行库为目标的语言添加自己的描述性元素，而不必重新编写编译器。基于以上的机制，很多数据库实体框架都应用到自定义的属性 (Attribute)，来标识实体类对于的数据库表信息或者字段类型。

我们可以为控件上的多个成员（例如属性 Property）指定该属性 (Attribute)。如果已将 BindableAttribute 设置为 true 来标记属性，则应引发该属性的属性更改通知。这意味着，如果属性 (Property) 的 Bindable 设置为 Yes，则支持双向数据绑定。如果 Bindable 设置是 No，则我们虽然可以绑定到该属性 (Property)，但它不应该显示在默认的要绑定到的属性 (Property) 集中，因为它不一定引发属性 (Property) 更改通知。

注意
当使用设置为 true 的 BindableAttribute 标记某个属性 (Property) 时，此属性 (Attribute) 的值会被设置为常数成员 Yes。对于使用设置为 false 的 BindableAttribute 标记的

属性（Property），则此值为 No。因此，若要在代码中检查该属性（Attribute）的值，必须将该属性（Attribute）指定为 BindableAttribute.Yes 或 BindableAttribute.No。

2.4.2 Attribute 用法

上面阐述了属性（Property）和属性（Attribute），由于二者都为属性，让人感到非常的迷糊。英文上 Property 和 Attribute 是不同的，但是翻译为中文都是属性。这一节重点阐述的是属性 Attribute，它在控件开发中非常重要，下面给出示例代码：

```
private int _age = 28;
[
BrowsableAttribute(true),
BindableAttribute(false),
CategoryAttribute("自定义属性"),
DescriptionAttribute("年龄，默认为28岁"),
DefaultValueAttribute(28),
]
public int Age
{
    get { return _age; }
    set { _age = value; }
}
```

上述代码中的 BrowsableAttribute(true)也可以写成 Browsable(true)，二者是等价的，表示被此 Attribute 标注的 Age 属性为可见的。CategoryAttribute("自定义属性")等价于 Category("自定义属性")，表示被此 Attribute 标注的 Age 属性归为自定义属性类别。DescriptionAttribute("年龄，默认为 28 岁")等价于 Description("年龄，默认为 28 岁"),用于属性描述。BindableAttribute(false)表示被此 Attribute 标注的属性不能被绑定。上面的代码所在的控件在 Visual Studio IDE 的属性浏览器中对应的关系为下图 2.2 所示。

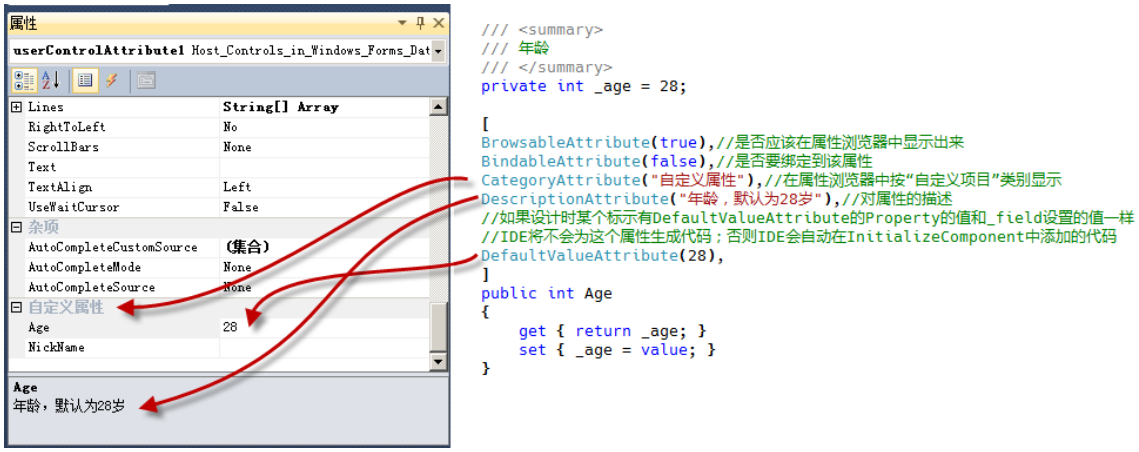


图 2.2 Viusal Studio 属性和控件属性标注的对应关系

2.4.3 自定义 Attribute

在自定义开发中，我们还可以自定义 Attribute，自定义的 Attribute 必须继承 System.Attribute 基类，下面的自定义控件 UserControl 中就用到了自定义的 CustomAttribute（内部简单的配置 Browsable 属性），Custom(true)表示 jackid 属性可见，代码如下：

```
public partial class UserControl1 : UserControl
{
    public UserControl1()
    {
        InitializeComponent();
    }
    [
        Custom(true)//CustomAttribute(true)
    ]
    public int jackid
    {
        get;
        set;
    }
}
//自定义Attribute
[AttributeUsage(AttributeTargets.All)]
public sealed class CustomAttribute : Attribute
{
    public CustomAttribute(bool browsable)
    {
        this.Browsable = browsable;
```

```
    }
    public bool Browsable { get; set; }
}
```

此属性定义说明了下列几点：

■ 属性类必须声明为公共类。

■ 按照约定，属性类的名称以单词 Attribute 结尾。虽然并不要求这样，但出于可读性目的，建议采用此约定。应用属性时，可以选择是否包含 Attribute 一词。

■ 在.NET C#语言中所有属性类都必须直接或间接地从 System.Attribute 继承。

2.5 控件设计时支持

上面介绍了自定义控件设计时属性的使用，这一节介绍下一个重要的知识点就是如何实现控件的设计时支持（Design-Time Support）。控件设计时支持可以让用户开发扩展功能去配置控件的属性和方法。一般来说，为了方便自定义控件的属性配置，都需在控件设计时，用可视化界面去修改配置控件的属性。在 .NET 框架中有相关基础类或者接口让开发人员可以定制设计时支持功能扩展。

.NET 提供了三种常见的设计时支持方式：

■ UITypeEditor

UI 类型编辑器可以提供一个自定义用户界面 (UI)，以便在设计时编辑属性的值并显示属性值的表示形式。UI 类型编辑器是特定于具体类型的，并且提供了用户界面，以便在设计时配置该编辑器支持的属性。UI 类型编辑器可以显示“Windows 窗体”或下拉配置界面以便配置控件的属性。

■ Designers

设计器可以在设计时自定义组件的行为，包括它的外观、初始化以及如何与用户交互。对于选定的组件，设计器可以添加、移除或替换属性浏览器中列出的属性。设计器可以提供用户定义的方法，这些方法可以链接到某些组件事件，或从自定义菜单命令或 DesignerVerb 中执行。设计器还可以使用由设计时环境提供的服务。

■ TypeConverters

类型转换器可用于字符串到值的转换，或用于在设计时和运行时进行数据类型之间的双向翻译。 在宿主（如窗体设计器中的属性浏览器）中，类型转换器允许以文本形式向用户表示属性值，并且可以将用户输入的文本转换为相应数据类型的值。类型转换器和 UI 类型编

辑器在设计时和运行时都可使用，而设计器只能在设计时使用。

.NET 提供的三种常见的设计时支持方式可用下图 2.3 表示：

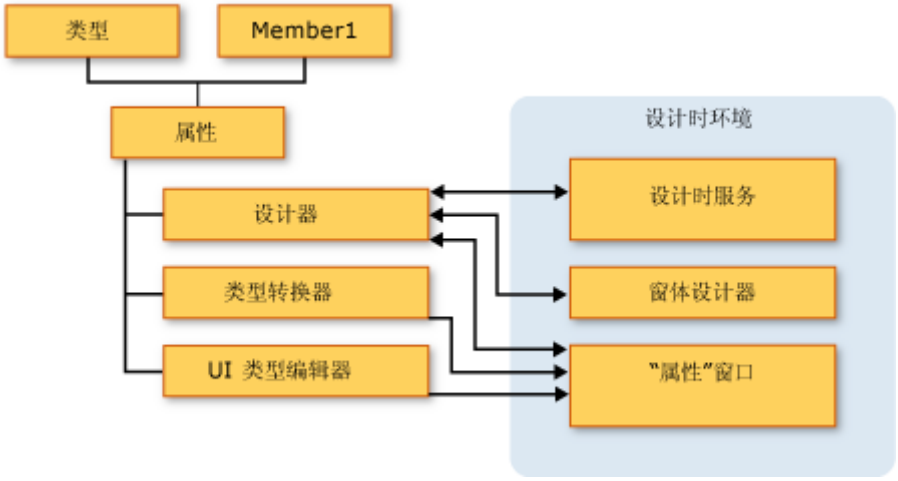


图 2.3 .NET Framework 中的设计时结构

.NET Framework 为在设计时环境中自定义组件行为和用户界面提供了接口和类。设计时环境通常包括用于排列组件的窗体设计器和用于配置组件的属性值的属性浏览器。设计时环境通常还会提供设计时机制可以访问和使用的设计时服务。

2.5.1 UITypeEditor

UITypeEditor 类提供一种基类，可以从该基类派生和进行扩展，以便为设计时环境实现自定义类型编辑器。在文本框值编辑器不足以有效地选择某些类型的值的情况下，自定义类型编辑器非常有用。若要实现自定义设计时 UI 类型编辑器，必须执行下列步骤：

1. 定义一个从 UITypeEditor 派生的类。
2. 重写 EditValue 方法以处理用户界面、用户输入操作以及值的分配。
3. 重写 GetEditStyle 方法，以便将编辑器将使用的编辑器样式的类型通知给属性窗口。

```
namespace System.Drawing.Design
{
    //指定System.Drawing.Design.UITypeEditor 的值编辑样式
    public enum UITypeEditorEditStyle
    {
        //不提供任何交互用户界面 (UI) 组件。
        None = 1,
        //模式窗口
        Modal = 2,
```

```
        //显示一个下拉箭头按钮，并在下拉对话框中承载用户界面 (UI)。
        DropDown = 3,
    }
}
```

下表 2.2 到表 2.5 列出了与控件绘图相关命名空间中的一些重要类型。

表 2.2 System.Drawing.Design 重要类型

类型	说明
UITypeEditor	提供用于实现值编辑器的基类。
IToolboxService	提供管理和查询开发环境中的"工具箱"的方法和属性。
ToolboxItem	提供工具箱项的基实现。

表 2.3 System.ComponentModel.Design 重要类型

类型	说明
DesignerActionList	为类型提供基类，这些类型定义用于创建智能标记面板的项目的列表。
DesignSurface	为设计组件提供一个用户界面。
IDesigner	提供构建自定义设计器的基本框架。
IRootDesigner	提供对根级设计器视图技术的支持。
IExtenderProviderService	提供一个接口，用于在设计时添加和移除扩展程序提供程序。
UndoEngine	指定一般撤消/重复功能。

表 2.4 System.Windows.Forms.Design 重要类型

类型	说明
IWindowsFormsEditorService	提供一个接口，供 UI 类型编辑器用来在设计模式下显示 Windows 窗体，或显示 PropertyGrid 控件中的下拉区域中的控件。
ControlDesigner	基设计器类，用于扩展 Control 的设计模式行为。
DocumentDesigner	基设计器类，用于扩展支持嵌套控件并接收滚动消息的 Control 的设计模式行为，并为其提供根级设计模式视图。

表 2.5 System.Windows.Forms.Design.Behavior 重要类型

类型	说明
BehaviorService	管理设计器中的用户界面。
Behavior	表示由 BehaviorService 管理的 Behavior 对象。
Adorner	管理与用户界面相关的 Glyph 对象的集合。此类不能被继承。
Glyph	表示一个由 Adorner 管理的用户界面 (UI) 实体。

对于自定义的控件，如何为某个属性配置一个 UITypeEditor 来作为设计时属性编辑器呢？其实只需要在某个属性上用 Editor 属性进行标注即可。

```
[Editor(typeof(WidthEditor),typeof(UITypeEditor))]
public int RectHeight
```

下面我们自定义一个 RectControl 的控件，并用 WidthEditor 类（此类没有给出代码）作为 UITypeEditor，如图 2.4 所示：

```
using System;
using System.Collections;
```

```
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Windows.Forms;
using System.Drawing.Design;
namespace MingForm.Controls
{
    public class RectControl : System.Windows.Forms.UserControl
    {
        private System.ComponentModel.Container components = null;
        private int width =60;
        private int height =60;
        /// <summary>
        /// RectWidth属性，并指定WidthEditor为属性编辑器
        /// </summary>
        [Description("宽度"), Category("UITypeEditor"),
        Editor(typeof(WidthEditor),typeof(UITypeEditor))]
        public int RectWidth
        {
            get
            {
                return this.width;
            }
            set
            {
                if( value > 0)
                {
                    this.width = value;
                    //重绘
                    this.Invalidate();
                }
            }
        }
        /// <summary>
        /// RectHeight属性，并指定WidthEditor为属性编辑器
        /// </summary>
        ///
        [Description("高度"),Category("UITypeEditor"),
        Editor(typeof(WidthEditor),typeof(UITypeEditor))]
        public int RectHeight
        {
            get
            {
                return this.height;
```

```
            }
            set
            {
                if( value > 0)
                {
                    this.height = value;
                    this.Invalidate();
                }
            }
        }
        public RectControl()
        {
            InitializeComponent();
        }
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if( components != null )
                    components.Dispose();
            }
            base.Dispose( disposing );
        }
        #region Component Designer generated code
        private void InitializeComponent()
        {
            this.Name = "RectangleControl";
            this.Paint += new
System.Windows.Forms.PaintEventHandler(this.RectangleControl_Paint)
;
        }
        #endregion
        private void RectangleControl_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
        {
            Brush br = new SolidBrush(Color.Red);

    e.Graphics.FillRectangle(br,0,0,this.RectWidth,this.RectHeight);
        }
    }
}
```

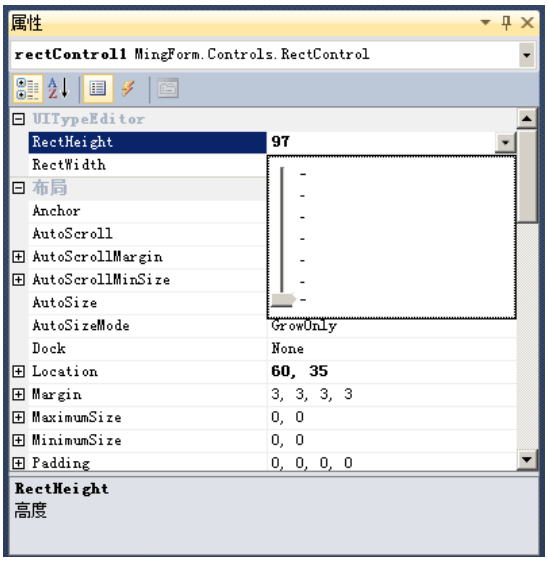


图 2.4 UITypeEditor 示例

2.5.2 TypeConverters

在属性窗口中，像字体和大小等属性是一个内置的对象，其有若干个属性构成，在属性窗口中，这个属性前面都会有一个+号，可以点击展开并对其每个属性进行配置。对于自定义控件来说，如果控件中包含类似的属性，那么可以通过 TypeConverters 来实现属性窗口呈现+号的效果。

假设我们已经定义了一个 Person 类，其中包含两个属性（Name 和 PhoneNum）。然后我们定义一个继承自 TypeConverter 的 PersonTypeConverter 类型转换器，代码如下：

```
using System;
using System.ComponentModel;
namespace MingForm.Controls
{
    public class PersonTypeConverter : TypeConverter
    {
        public PersonTypeConverter()
        {
        }
        public override bool GetPropertiesSupported(
            ITypeDescriptorContext context)
        {
            return true;
        }
        public override PropertyDescriptorCollection GetProperties(
            ITypeDescriptorContext context, object value,
```

```
            Attribute[] attributes)
        {
            return TypeDescriptor.GetProperties(typeof(Person));
        }
    }
}
```

然后我们定义一个继承自UserControl的PersonControl控件，该控件上面用两个TextBox分别显示关联Person的信息，代码如下：

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Windows.Forms;
namespace MingForm.Controls
{
    public class PersonControl : System.Windows.Forms.UserControl
    {
        private System.ComponentModel.Container components = null;
        private System.Windows.Forms.Label label3;
        private System.Windows.Forms.Label label4;
        private System.Windows.Forms.TextBox txtPhnNumber;
        private System.Windows.Forms.TextBox txtName;
        #region
        private Person persondata = new Person();
        [DesignerSerializationVisibility(
DesignerSerializationVisibility.Content)]
        public Person PersonData
        {
            get
            {
                return this.persondata;
            }
            set
            {
                if(value!=null)
                {
                    this.persondata = value;
                    DisplayData();
                }
            }
        }
        #endregion
```

```
        public PersonControl()
        {
            InitializeComponent();
            this.persondata.PropertyChanged +=
                new
Person.PropertyChangedEventHandler(persondata_PropertyChanged);
        }
        void persondata_PropertyChanged(string propertyname)
        {
            DisplayData();
        }
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if( components != null )
                    components.Dispose();
            }
            base.Dispose( disposing );
        }
        #region Component Designer generated code
        private void InitializeComponent()
        {
            this.label3 = new System.Windows.Forms.Label();
            this.txtName = new System.Windows.Forms.TextBox();
            this.label4 = new System.Windows.Forms.Label();
            this.txtPhnNumber = new System.Windows.Forms.TextBox();
            this.SuspendLayout();
            //… 其他配置
            this.ResumeLayout(false);
            this.PerformLayout();
        }
        #endregion
        private void txtPhnNumber_TextChanged(object sender,
System.EventArgs e)
        {
            persondata.PhoneNum = txtPhnNumber.Text;
        }
        private void DisplayData()
        {
            txtName.Text = persondata.Name;
            txtPhnNumber.Text = persondata.PhoneNum;
        }
        private void txtName_TextChanged(object sender,
```

```
System.EventArgs e)
        {
            persondata.Name = txtName.Text;
        }

    }
}
```

当我们在Visual Studio IDE中对PersonControl进行设计时，我们可以在属性面板中通过修改PersonData的数据，来讲配置反应到PersonControl控件上，如下图2.5所示：

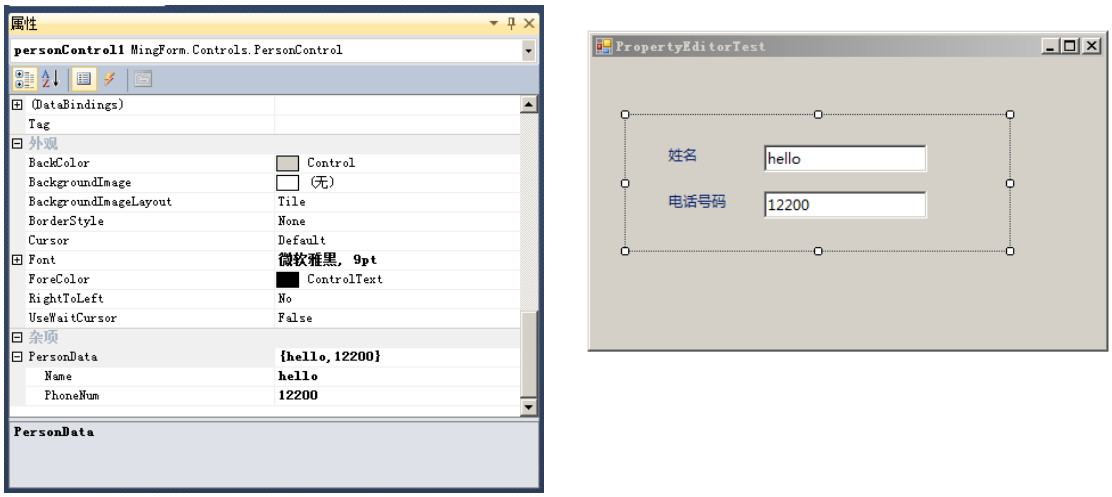


图 2.5 TypeConverters 示例

2.5.3 Custom Designer

上面分别介绍了用 UITypeConverter 和 TypeConverter 实现控件的设计时支持功能扩展，下面介绍另外一种方式，即用自定义的 Designer 来实现控件的设计时支持。首先创建一个继承自 System.Windows.Forms.Design 命名空间下 ControlDesigner 类的 PersonControlDesigner 类，由于 System.ComponentModel.Design 下也有此类，因此必须明确指明，这里用 using FD=System.Windows.Forms.Design;创建一个别名 FD，可以简化代码。此类的核心是在 DesignerVerbCollection 实例 actions 中添加一个 DesignerVerb 对象，这里将动作命名为样式配置，并且绑定了一个事件处理程序，可以弹出一个窗体，并将窗体上设置的值回传到控件，代码如下：

```
using System;
using System.Windows.Forms;
using System.Collections;
using System.Drawing;
```

```
using FD=System.Windows.Forms.Design;
using System.ComponentModel.Design;
namespace MingForm.Controls
{
    public class PersonControlDesigner:FD.ControlDesigner
    {
        public PersonControlDesigner()
        {
        }
        #region UserDefinedVariables
        private DesignerVerbCollection actions;
        public override DesignerVerbCollection Verbs
        {
            get
            {
                if(actions == null)
                {
                    actions = new DesignerVerbCollection();
                    actions.Add(new DesignerVerb("样式配置", new
EventHandler(ChangeDisplay)));
                }
                return actions;
            }
        }
        #endregion
        #region EventHandlers
        void ChangeDisplay(object sender, EventArgs e)
        {
            Formater formt = new Formater();
            formt.ShowDialog();
            //传值
            Control.BackColor = formt.Style.BackGroudColor;
        }
        #endregion
    }
}
```

下面创建一个 PersonControl 类，它继承自 UserControl，PersonControl 类用属性 Designer(typeof(PersonControlDesigner))进行标识，表明其用上面定义的 PersonControlDesigner 来作为 Designer 来配置其 BackColor 属性，如图 2.6 所示。

```
[Designer(typeof(PersonControlDesigner))]
public class PersonControl : System.Windows.Forms.UserControl
```

```
{
    // … 省略
}
```

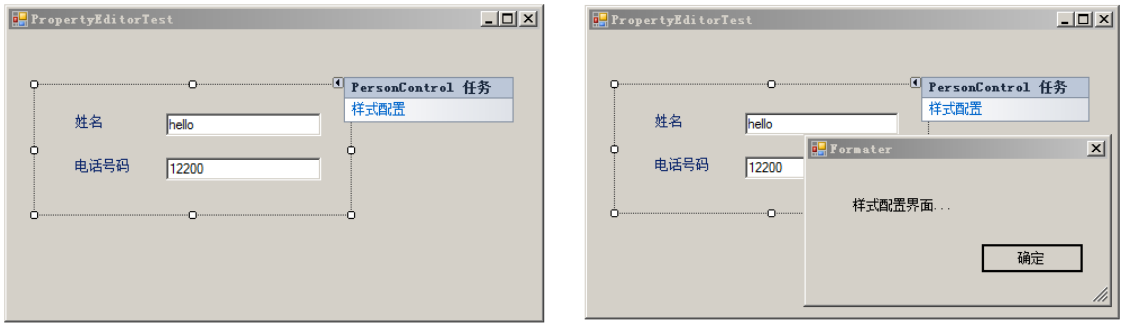


图 2.6 Custom Designer 示例

2.6 颜色

在.NET 中,所有的 Windows Forms 控件都有 ForeColor 和 BackColor 两个属性。在字面上看 ForeColor 和 BackColor 分别为控件的前景色和背景色。但对于不同的控件，这两个属性的实际意义存在一些差异。

Color 结构(不是类)定义在 System.Drawing 命名空间下，我们可以非常方便的创建 Color 对象。一般来说，创建 Color 有以下几种方式,请看下表 2.6:

表 2.6 创建 Color 的几种方式

方法	描述	示例
FromArgb	ARGB (alpha, red, green, blue)来定义颜色，其中每个参数范围为 0 到 255。定一个为透明度值，其他三个参数依次为红色、绿色和蓝色的分量值。	`Color c = Color.FromArgb(255, 66, 66, 66);`
Color 预定义	获取.NET 预定义在 Color 结构中的颜色。	`Color c = Color.Red;`
FromWin32	将 Windows 颜色值翻译成 GDI+System.Drawing.Color 结构。	`Color c = ColorTranslator.FromWin32(0XFF00);`
FromHtml	将 HTML 颜色表示形式翻译成	`Color c = ColorTranslator.FromHtml("#FF0000");`

	GDI+ System.Drawing.Color 结构。	
FromOle	将 OLE 颜色值翻译成 GDI+ System.Drawing.Color 结构。	Color c = ColorTranslator.FromOle(0xDD00);
SystemColors	从系统设置的颜色体系中获取颜色	Color c = SystemColors.ActiveCaptionText;

举例来说，下面可以用 ColorTranslator.FromHtml 方法来给 txtUserName 控件设置前景色和背景色：

```
this.txtUserName.ForeColor = ColorTranslator.FromHtml("#FF0000");
this.txtUserName.BackColor = ColorTranslator.FromHtml("#999999");
```

下面看一个例子来说明如何 System.Enum.GetNames 方法来获取所有已知颜色的名字，并通过 System.Enum.Parse 和 System.Drawing.Color.FromKnownColor 根据指定颜色名称获取 Color 对象。

下面给出已知颜色的列表，通过下拉列表的选择，将选择的颜色填充到 pictureBox1 的 BackColor 中，核心代码如下：

```
void cbbColors_SelectedIndexChanged(object sender, EventArgs e)
{
    KnownColor selectedColor;
    selectedColor = (KnownColor)System.Enum.Parse(
    typeof(KnownColor), cbbColors.Text);
    this.pictureBox1.BackColor =
        System.Drawing.Color.FromKnownColor(selectedColor);

}
private void FrmColor_Load(object sender, EventArgs e)
{
    string[] _colorNames;
    _colorNames = System.Enum.GetNames(typeof(KnownColor));
    cbbColors.Items.AddRange(_colorNames);
    cbbColors.SelectedIndex = 0;
}
```

运行该窗体显示如图 2.7 所示：

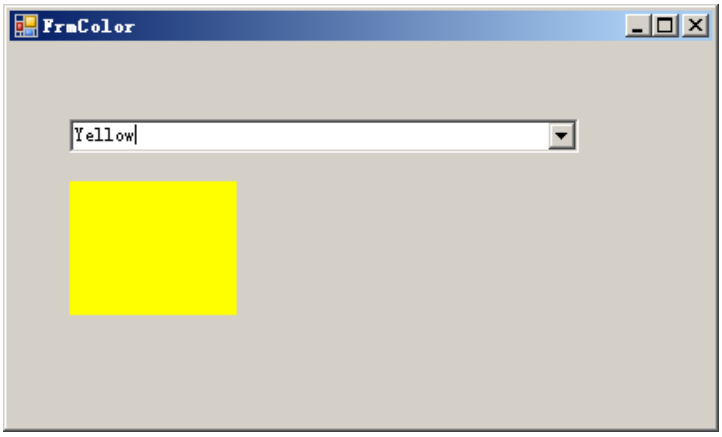


图 2.7 Color 选择示例

在 Web 应用中，很多时候为了 UI 的美观，会将元素的背景设置为透明，但是 Windows Forms 与 Web 不同，我们必须要需要注意的是，虽然程序可以通过 Color.Transparent 或者 Color.FromArgb(0,Color.Red)来设置某个控件背景为透明，但是.NET 对于控件透明背景支持的并不好，不是真正能做到透明背景。下面用一个例子来说明，如图 2.8 所示。

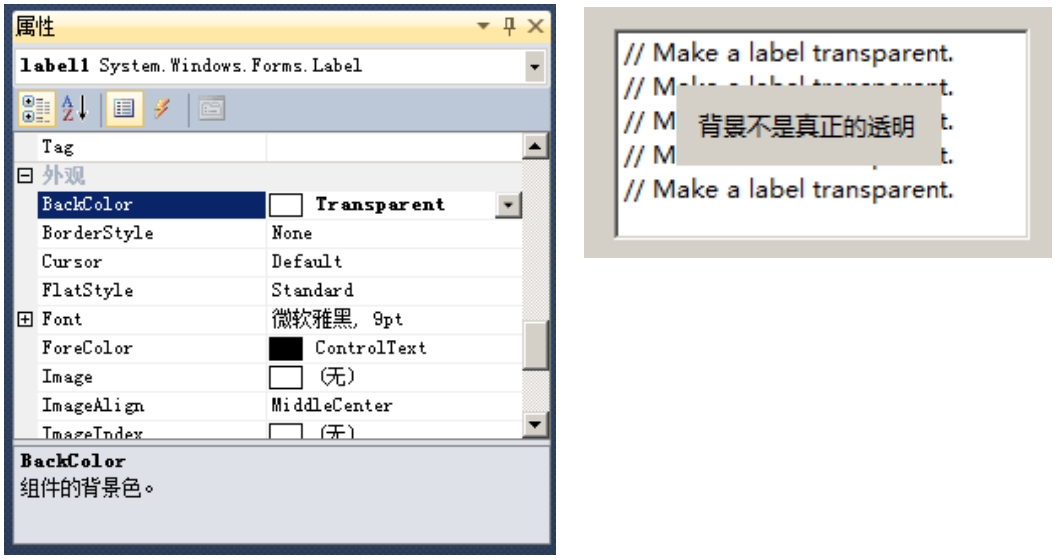


图 2.8 透明背景的设置

虽然我设置了 Label1 的背景为 Color.Transparent，但是界面上 Label1 的背景依然存在（Window 主题为经典，其他的主题现实效果上可能会有区别），不是真正的透明。这种问题目前没有什么好的解决方案，只能通过 GDI+ 自定义绘制（Owner-drawn Controls）来解决。

2.7 字体

说到字体，都是针对文本而言的，文本对于窗体控件来说非常重要，一般都是用来显示信息或者说明的。 文本的样式很大一部分受到字体的影响，例如可以通过 Ctrl.Font = new Font("Segoe UI", 12)来配置控件 Ctrl 的字体。

随着技术的发展，web 上以前的图片按钮现在逐步换成了图标字体，这些图标字体是矢量图，矢量图意味着每个图标都能在所有大小的屏幕上完美呈现，可以随时更改大小和颜色，而且不失真，真心给人一种"高大上"的感觉。由于 Font Awesome 是完全免费的，无论个人还是商业使用，因此这种字体库使用的比较多。Font Awesome 一个字体文件包含了非常多的实用图标，可以助你完整表达 web 页面上每个动作的含义（图优于表，表优于文字）。

Font Awesome 是完全从头设计的整套图标，完全和 Bootstrap 兼容，他们是一组很好的搭档。可以用 CSS 很方便的进行使用，Font Awesome 的部分字体编码（http://fontawesome.io/3.2.1/cheatsheet/）体现如下图 2.9 所示：

图 2.9 fontawesome 字体截图

但是问题来了，上面阐述的都是在 Web 页面中进行应用，但是如何在 Winform 界面中使用呢（WPF 也可以）？当然第一步就是下载 Font Awesome 到本地，并安装该字体，字体名称为 FontAwesome。下面给出示例代码如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
```

```
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WinFormFont
{
    public partial class IconFontDemo : Form
    {
        public IconFontDemo()
        {
            InitializeComponent();
            //必须是unicode码
            this.label1.Text = "\uF028";
            this.label1.Font = new Font("FontAwesome", 16);
            this.label1.ForeColor = Color.Green;
            //必须是unicode码
            this.textBox1.Text = "\uF029 jackwangcumt";
            this.textBox1.Font = new Font("FontAwesome", 16);
            this.textBox1.ForeColor = Color.Black;
        }
        private void button1_Click(object sender, EventArgs e)
        {
            int fontsize = int.Parse(this.textBox2.Text);
            this.label1.Font = new Font("FontAwesome", fontsize);
            this.textBox1.Font = new Font("FontAwesome", fontsize);
        }
    }
}
```

我们可以更改字体的大小，然后单击刷新按钮，可以看到字体图标会根据字体大小的设置而做出相应的变化，如图 2.10 所示：

图 2.10 fontawesome 字体示例

那么肯定有人会问，如果我想使用一种图形字体，但是我怎么知道该图形字体所对应的 unicode 编码是什么呢？针对这个问题，我们可以通过在 word 中插入符号，然后选择该字体，然后点选需要的图标，看一下对应的 unicode 编码即可，如图 2.11 所示：

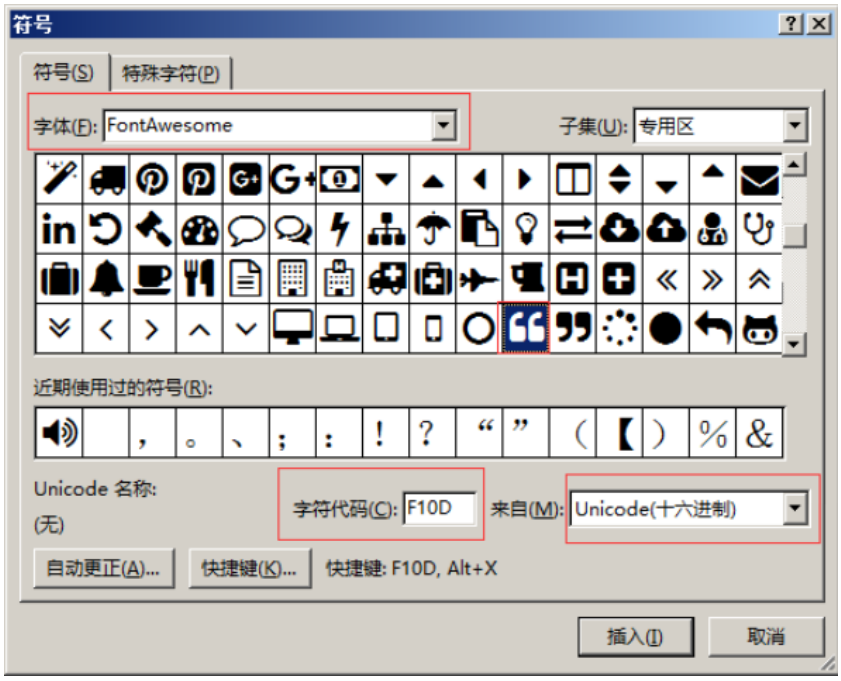


图 2.11 word 中 fontawesome 字体编码

2.8 鼠标和键盘

一般来说，控件提供一些内置的事件来响应鼠标和键盘的操作，来达到与人的交互。其中就包含鼠标的单击和移动，键盘的按键和组合等信息。

2.8.1 键盘

键盘与控件的交互是比较常见的，一般来说主要通过三个事件来交互： KeyDown、KeyPress 和 KeyUp。常用的 KeyDown 和 KeyUp 事件可以获取键盘按键的信息，而 KeyPress 一般用于输入的验证和限制输入（例如只允许输入数值），这些事件的执行顺序如图 2.12 所示：

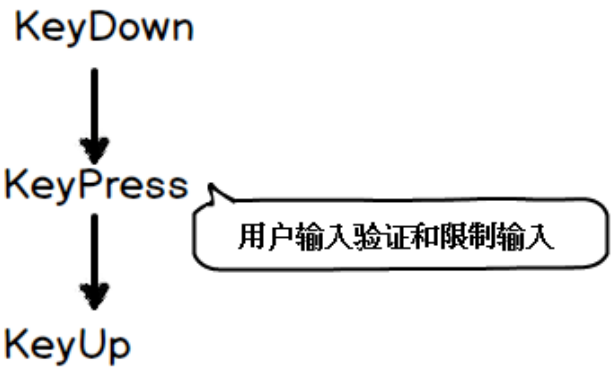


图 2.12　KeyUp、KeyPress 和 KeyDown 的相应顺序

常用的 KeyDown、KeyUp 和 KeyPress 事件的描述如表 2.7 所示：

表 2.7　KeyUp、KeyPress 和 KeyDown 事件描述

事件	描述
KeyDown	当键盘某个键被按下后触发
KeyPress	在输入字符出现之前触发
KeyUp	当键盘某个键被释放后触发

从字面上看 KeyDown 和 KeyPress 比较容易混淆，那么二者有什么区别呢？下面通过一个例子来说明二者的区别。假设用户依次按下键盘的 Ctrl 键和 A 键。那么 KeyDown 事件会触发 2 次而 KeyPress 只触发 1 次。

```
private void cbbColors_KeyDown(object sender, KeyEventArgs e)
{
    this.Text = "Key Code: " + e.KeyCode.ToString();
    this.Text += "Key Value: " + e.KeyValue.ToString();
}
private void cbbColors_KeyPress(object sender, KeyPressEventArgs e)
{
        this.Text = "Key Press:"+ e.KeyChar.ToString();
}
private void cbbColors_KeyUp(object sender, KeyEventArgs e)
{
    this.Text = "Key Code: " + e.KeyCode.ToString();
    this.Text += "Key Value: " + e.KeyValue.ToString();
}
```

获取 Alt, Control 和 Shift 修饰符可以用以下方法来获取：

```
private void cbbColors_KeyDown(object sender, KeyEventArgs e)
{
    //用Modifiers获取 Alt, Ctrl和Shift键
    if ((e.Modifiers & Keys.Shift) == Keys.Shift)
    {
```

```
        this.Text = "Shift";
    }
    //更简单的方式来获取Alt, Ctrl和Shift键
    if (e.Alt)
    {
        this.Text = "Alt";
    }
}
```

对于键盘上的 Caps Lock, Scroll Lock 和 Num Lock 键，需要用下面的方法来检测：

```
private void cbbColors_KeyDown(object sender, KeyEventArgs e)
{
    if (Control.IsKeyLocked(Keys.CapsLock))
    {
        //Caps Lock开启
    }
    if (Control.IsKeyLocked(Keys.NumLock))
    {
        // Num Lock开启
    }
}
```

但是对于覆盖和插入模式的判定，必须要用到非托管（unmanaged）的 win32 API GetKeyState 方法来获取：

```
//调用 Win32 API
[DllImport("User32.dll")]
private static extern short GetKeyState(System.Windows.Forms.Keys key)
private void cbbColors_KeyDown(object sender, KeyEventArgs e)
{
    if (GetKeyState(Keys.Insert) == 1)
    {
        Console.WriteLine("Overwrite");
        // Overwrite模式
    }
    else
    {
        // Insert模式
        Console.WriteLine("Insert");
    }
}
```

2.8.2 鼠标

除了键盘与控件的交互外，鼠标和控件进行交互也是非常常见的。鼠标事件可以获取鼠标单击的信息和移动的信息，示例代码如下：

```
private void cbbColors_MouseMove(object sender, MouseEventArgs e)
{
    if ((e.Button & MouseButtons.Right) == MouseButtons.Right)
    {
        if ((e.Button & MouseButtons.Left) == MouseButtons.Left)
        {
            //逻辑处理
        }
    }
}
private void cbbColors_MouseUp(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Right)
    {
        //逻辑处理
    }
    if (e.Button == MouseButtons.Left)
    {
        //逻辑处理
    }
}
```

每个控件都提供 MousePosition, MouseButtons 和 ModifierKeys 属性，其中 MouseButtons 和 ModifierKeys 属性获取鼠标和键盘的相关信息，而 MousePosition 属性则返回当前鼠标所在的位置信息（注意不是事件触发的位置）。另外非常需要注意的是，MousePosition 属性使用的屏幕坐标，不是控件坐标，不过二者的坐标可以通过 Form.PointToClient()和 Form.ClientToPoint()进行互相转换，代码如下：

```
private void cbbColors_MouseMove(object sender, MouseEventArgs e)
{
    Point mousep = MousePosition;
    Point s2c = PointToClient(mousep);
    Point c2s = PointToScreen(s2c);
}
```

2.9 本章小结

本章首先阐述了控件（Control）和组件（Component）的概念和区别，并介绍了Windows Forms 控件的三种类型：

- 复合控件（Composite Controls）；
- 扩展控件（Extended Controls）
- 自定义控件（Custom Controls）

然后阐述了控件的设计时属性和如何自定义 Attribute。在此基础上，介绍了如何实现控件的设计时支持（Design-Time Support）。控件设计时支持可以让用户开发扩展功能去配置控件的属性和方法。.NET 提供了三种常见的设计时支持方式：

- UITypeEditor
- Designers
- TypeConverters

最后我们介绍了颜色、字体和鼠标和键盘等知识，这些基础知识对于后续的控件开发将大有好处。

第三章 GDI+基础

GDI 是 Graphics Device Interface 的缩写，含义是图形设备接口，它的主要任务是负责系统与绘图程序之间的信息交换，处理所有 Windows 程序的图形输出。

在 Windows 操作系统下，绝大多数具备图形界面的应用程序都离不开 GDI，我们利用 GDI 所提供的众多函数就可以方便的在屏幕、打印机及其它输出设备上输出图形，文本等操作。GDI+对 GDI 进行了性能优化，并添加了许多新的功能。GDI+使得应用程序开发人员在输出屏幕和打印机信息的时候无需考虑具体显示设备的细节，GDI+使得图形硬件和应用程序相互隔离，从而使开发人员编写设备无关的应用程序变得非常容易。

GDI+的核心是 Graphics 对象，Graphics 类定义了绘制和填充图形对象的方法和属性。Graphics 类的属性（字段）很多，具体可参见 MSDN。Graphics 类的方法分为三类：绘制、填充及其他。

3.1 GDI+用途

一般来说，GDI+在以下场景中具有广泛的应用：

2D 向量图：利用 GDI+,我们可以绘制直线、曲线、矩形和椭圆等形状，并且可以绘制路径，并填充区域。

图片：GDI+可以将位图图片渲染到界面上，同时支持对位图进行各类操作（例如缩放、旋转等）

字体排印：GDI+可以让我们渲染出光滑的反锯齿的文本，同时可以设置其大小、字体、颜色和方向等。

下面给出 GDI+命名控件及其功能，如表 3.1 所示。

表 3.1 GDI+命名空间及描述

命名空间	描述
System.Drawing	提供基本的 GDI+功能，包含的 Graphics 类可以绘制线、矩形等。同时可以定义位图、颜色、字体、笔刷和笔画等。
System.Drawing.Drawing2D	提供高级的 2 维 GDI+功能，其中包含对绘图质量的设置、渐变填充和 GraphicsPath 等
System.Drawing.Imaging	提供对位图和向量图的操作
System.Drawing.Text	提供访问当前安装的可用字体集合
System.Drawing.Printing	提供将 GDI+绘制的内容输出到打印机上的功能

3.2 GDI+绘制

既然GDI+在界面绘制上占有重要的地位，那么问题来了，如何才能调用GDI+绘制对象呢？常用的方式为重载控件的OnPaint事件，然后在事件内进行GDI+绘制，请看下面的代码：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace GDIDemo
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            //绘图无效
            Pen drawingPen = new Pen(Color.Black, 3);
            this.CreateGraphics().DrawArc(
                drawingPen, 36, 19, 100, 100, 40, 180);
            drawingPen.Dispose();
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            Pen drawingPen = new Pen(Color.Black, 3);
            Rectangle rec=new Rectangle(10,10,120,80);
            e.Graphics.DrawRectangle(drawingPen, rec);
            drawingPen.Dispose();
            base.OnPaint(e);
        }
        private void button1_Click(object sender, EventArgs e)
        {
            Pen drawingPen = new Pen(Color.Red, 15);
            Graphics g = this.CreateGraphics();
```

```
            Rectangle rec = new Rectangle(60, 60, 120, 80);
            g.DrawRectangle(drawingPen, rec);
            //释放非托管的资源
            drawingPen.Dispose();
            g.Dispose();
        }
    }
}
```

运行该窗体，结果如图3.1所示。

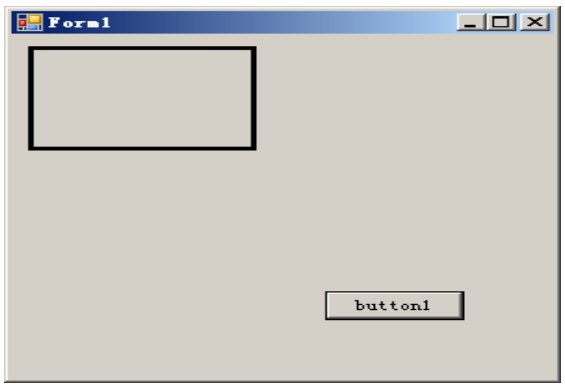


图3.1 GDI+OnPaint事件绘制示例

值得注意的是，在窗体 Load 事件中，进行的绘制并未反映到 UI 上，因为事件执行顺序为先 Load 再 OnPaint，虽然 Load 里面的代码正常的执行，但是也会被 OnPaint 重新“抹掉”。我们利用 GDI+进行绘图时，先要获取 Graphics 对象，一般可以在 OnPaint 事件中通过 e.Graphics 进行获取，在 OnPaint 事件我们不需要手动释放资源，.NET 会自动帮我们处理。但是如果是用 Control.CreateGraphics()方法来创建的 Graphics 对象，那么必须要手动进行资源释放，究其原因是因为 Graphics 对象采用的非托管的系统资源。

我们单击 button1 按钮，会在界面上绘制第二个方块，但是这种在 Click 事件中绘图的方法和在 OnPaint 事件中绘图的方法有着较大的区别，当你将窗体最小化或者隐藏（最大化或者窗体大小调整不影响）再最大化或者显示时，第二个方块就自动消失。对于这种很是奇怪的现象，对于刚接触 GDI+的人来说，确实有点摸不着头脑，究其原因是当最小化或者隐藏窗体时，操作系统将窗体进行了重绘，也就是再次调用了 Paint 事件，原有的绘图都被重新抹掉。

单击按钮button1后，结果如图3.2所示。

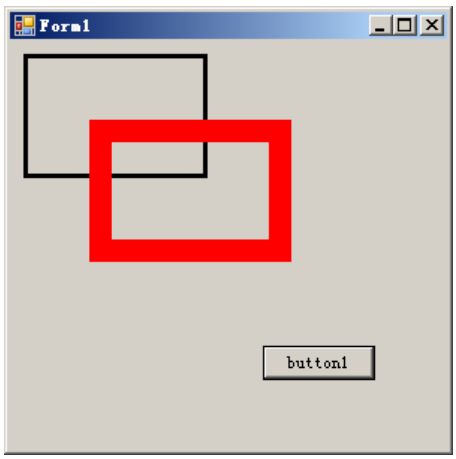


图3.2 GDI+绘制示例

3.3 绘制和重绘

接着上面的话题，当窗体最小化或者隐藏时，操作系统就会将窗体的 UI 资源在内存中释放掉，然后当窗体恢复时，操作系统会发送消息让窗体调用 Control.OnPaint()方法来重新创建窗体 UI。这种架构之所以这样处理，是由于以前的计算机资源不像现在这样比较"富裕"，图形资源占用大量的内存等资源，如果不及时释放掉，会严重消耗掉内存。如果有一个窗体被另外一个窗体遮挡住一部分，那么被遮挡区域的控件将被重绘。

因此，基于历史遗留的架构设计，对于窗体或者控件的绘制逻辑建议放在 Paint 事件中，放在其他地方往往不能按照预期进行绘制。那么又一个问题来了，如果我们就是需要在其他方法中调用绘图逻辑，那该怎么办呢？其实解决方法是将绘图逻辑依然放在 Paint 事件，其他方法可以配置相关属性，然后通过 Invalidate()来调用 Paint 事件进行绘图即可。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace GDIDemo
{
    public partial class Form2 : Form
    {
        public Form2()
        {
            InitializeComponent();
```

```
        }
        private void Form2_Load(object sender, EventArgs e)
        {
            string[] _colorNames;
            _colorNames = System.Enum.GetNames(typeof(KnownColor));
            cbbColors.Items.AddRange(_colorNames);
            cbbColors.SelectedIndex = 0;
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            Pen drawingPen = new Pen(_color, 3);
            Rectangle rec = new Rectangle(80, 80, 120, 80);
            e.Graphics.DrawRectangle(drawingPen, rec);
            Brush brush=new SolidBrush(_color);
            e.Graphics.FillRectangle(brush,rec);
            drawingPen.Dispose();
            base.OnPaint(e);
        }
        private Color _color = Color.Red;
        private void cbbColors_SelectedIndexChanged(object sender,
EventArgs e)
        {
            KnownColor selectedColor;
            selectedColor = (KnownColor)System.Enum.Parse(
            typeof(KnownColor), cbbColors.Text);
            _color=System.Drawing.Color.FromKnownColor(
                      selectedColor);
            //重绘
            this.Invalidate();
        }
    }
}
```

运行该窗体，结果如图3.3所示。

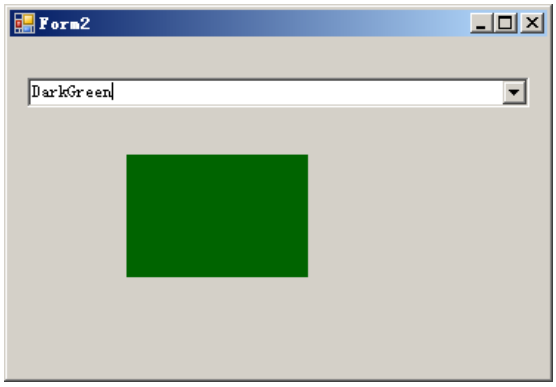


图3.3 GDI+重绘示例

这里还必须多说一句，我们在处理绘制逻辑时，不应该在其他方法中直接调用 Paint 事件处理程序或者 OnPaint() 方法。特别是绘图逻辑非常复杂的情况，往往会导致性能的降低和不期望的结果出现。对于这种需求，推荐使用 Invalidate()方法。Invalidate()方法可以重绘之前，可以保存相关变量的信息，如果 Invalidate()方法多次，往往会进行合理排队，按期望的顺序执行。Invalidate()方法如果连续执行多次，往往只需要 Paint 事件一次，但是如果直接调用 Paint 事件事件，那么调用几次就是重绘几次，因此这两种处理方法的性能存在不小的差距。

```
private int size;
private void button1_Click(object sender, EventArgs e)
{
    for (int i = 0; i < 100; i++)
    {
        size = i;
        //只重绘1次，而不是100次
        Invalidate();
    }
}
protected override void OnPaint(PaintEventArgs e)
{
    Pen pen = new Pen(Color.Red, 3);
    Rectangle rect = new Rectangle(80, 80, size, size);
    e.Graphics.DrawRectangle(pen, rect);
    pen.Dispose();
    base.OnPaint(e);
    System.Threading.Thread.Sleep(20);
}
```

3.4 刷新

上面的例子提到，虽然循环调用 100 次 Invalidate()方法，但是实际上.NET 内部只调用 1 次 OnPaint 方法，也就是直接绘制出 size=99 的矩形。这种方式在某些情景下确实能够提高性能，但是在某些情景下（例如需要看到这个矩形大小的变化情况），那么必须能在每次调用 Invalidate()方法时，都调用一次 OnPaint 方法。那么这种情况该如何实现呢？

```
private int size;
private void button1_Click(object sender, EventArgs e)
{
    for (int i = 0; i < 100; i++)
    {
```

```
            size = i;
            //重绘 100次
            Refresh();
        }
    }
    protected override void OnPaint(PaintEventArgs e)
    {
        Pen pen = new Pen(Color.Red, 3);
        Rectangle rect = new Rectangle(80, 80, size, size);
        e.Graphics.DrawRectangle(pen, rect);
        pen.Dispose();
        base.OnPaint(e);
        System.Threading.Thread.Sleep(20);
    }
```

用 Refresh()方法强制刷新 UI 即可实现。另外等价的一种写法是：

```
Invalidate(true);
Update();
```

3.5 大小调整与重绘

前面提到，窗体被遮挡一部分的时候，只会重绘被遮挡的那部分控件，这种设计在某些情况下确实能提高性能，但是在某些情况下（特别是窗体大小调整的时候）往往这种设计会出现奇怪的问题。为了更加直观的说明这个问题，下面看一个例子。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace GDIDemo
{
    public partial class Form3 : Form
    {
        public Form3()
        {
            InitializeComponent();
        }
        protected override void OnPaint(PaintEventArgs e)
        {
```

```
            base.OnPaint(e);
            Pen pen = new Pen(Color.Red, 2);
            e.Graphics.DrawEllipse(pen, new Rectangle(new Point(0, 0),
            this.ClientSize));
            pen.Dispose();
        }
    }
}
```

窗体我们在 OnPaint 事件中进行了绘图，当我们调整窗体大小的时候，界面会出现很奇怪的情况，如图 3.4 所示。

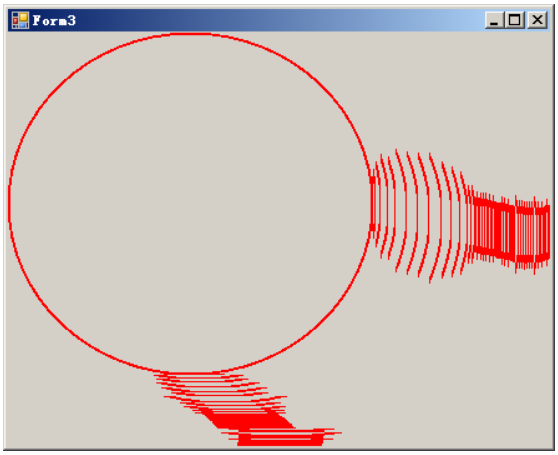


图3.4 GDI+大小变化后部分重绘示例

这个问题出现的原因是.NET 框架中的窗体假设只需要绘制隐藏或者恢复的区域，其他区域无需绘制，但是在窗体大小变化的情况下，基于这种假设就是不正确的。好在我们可以有多种方法来解决这个问题：

第一种就是在用 SetStyle(ControlStyles.ResizeRedraw,true); 设置窗体的 ResizeRedraw 为 True,这个设置会让窗体在大小变化时，自动进行窗体的重绘。

另外一种就是在 OnSizeChanged 事件中，调用 Invalidate()方法来重绘。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace GDIDemo
{
    public partial class Form3 : Form
    {
```

```
        public Form3()
        {
            InitializeComponent();
            //SetStyle(ControlStyles.ResizeRedraw, true);
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            base.OnPaint(e);
            Pen pen = new Pen(Color.Red, 2);
            e.Graphics.DrawEllipse(pen, new Rectangle(new Point(0, 0),
            this.ClientSize));
            pen.Dispose();
        }
        protected override void OnSizeChanged(EventArgs e)
        {
            base.OnSizeChanged(e);
            this.Invalidate();
        }
    }
}
```

运行该窗体，结果如图3.5所示。

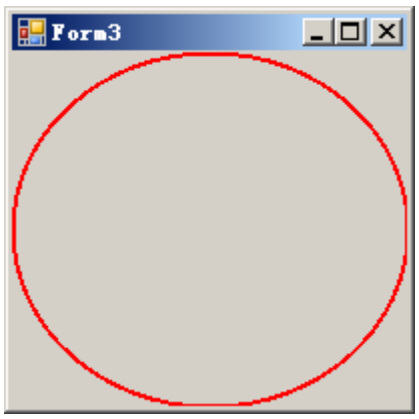


图3.5 GDI+大小变化后全部重绘示例

3.6 Graphic 类

需要先创建 Graphics 对象，然后才可以使用 GDI+ 绘制线条和形状、呈现文本或显示与操作图像。 Graphics 对象表示 GDI+ 绘图表面，并且该对象是用于创建图形图像的对象。处理图形包括两个步骤：

1. 创建 Graphics 对象。
2. 使用 Graphics 对象绘制线条和形状、呈现文本或显示与操作图像。

创建图形对象 Graphics 的方式有：

- Paint 事件

在窗体或控件的 Paint 事件中接收对图形对象的引用，作为 PaintEventArgs 的一部分。在为控件创建绘制代码时，通常会使用此方法来获取对图形对象的引用。同样，我们也可以在处理 PrintDocument 的 PrintPage 事件时获取 PrintPageEventArgs 对象中的属性作为图形对象。

- CreateGraphics

调用某控件或窗体的 CreateGraphics 方法以获取对 Graphics 对象的引用，该对象表示该控件或窗体的绘图图面。如果想在已存在的窗体或控件上绘图，请使用此方法。

- Graphics.FromImage

由从 Image 继承的任何对象创建 Graphics 对象。此方法在我们需要更改已存在的图像时十分有用。

下面的部分给出了有关 Graphic 的基础属性和方法，如表 3.2 和表 3.3 所示。

表 3.2 Graphic 基础属性

名称	说明
Clip	获取或设置 Region，它限定此 Graphics 的绘图区域。
ClipBounds	获取一个 RectangleF 结构，该结构限定此 Graphics 的剪辑区域。
CompositingMode	获取一个值，该值指定如何将合成图像绘制到此 Graphics。
CompositingQuality	获取或设置绘制到此 Graphics 的合成图像的呈现质量。
DpiX	获取此 Graphics 的水平分辨率。
DpiY	获取此 Graphics 的垂直分辨率。
InterpolationMode	获取或设置与此 Graphics 关联的插补模式。
IsClipEmpty	获取一个值，该值指示此 Graphics 的剪辑区域是否为空。
IsVisibleClipEmpty	获取一个值，该值指示此 Graphics 的可见剪辑区域是否为空。
PageScale	获取或设置此 Graphics 的世界单位和页单位之间的比例。
PageUnit	获取或设置用于此 Graphics 中的页坐标的度量单位。
PixelOffsetMode	获取或设置一个值，该值指定在呈现此 Graphics 的过程中像素如何偏移。
RenderingOrigin	为抵色处理和阴影画笔获取或设置此 Graphics 的呈现原点。
SmoothingMode	获取或设置此 Graphics 的呈现质量。
TextContrast	获取或设置呈现文本的灰度校正值。
TextRenderingHint	获取或设置与此 Graphics 关联的文本的呈现模式。
Transform	获取或设置此 Graphics 的几何世界变换的副本。
VisibleClipBounds	获取此 Graphics 的可见剪辑区域的边框。

表 3.3 Graphic 基础方法

名称	说明
BeginContainer()	保存具有此 Graphics 的当前状态的图形容器，然后打开并使用新的图形容器。
BeginContainer(Rectangle, Rectangle, GraphicsUnit)	保存具有此 Graphics 的当前状态的图形容器，然后打开并使用具有指定缩放变形的新图形容器。
Clear(Color)	清除整个绘图面并以指定背景色填充。
CopyFromScreen(Point, Point, Size)	执行颜色数据（对应于由像素组成的矩形）从屏幕到 Graphics 的绘图图面的位块传输。
CreateObjRef(Type)	创建一个对象，该对象包含生成用于与远程对象进行通信的代理所需的全部相关信息。
Dispose()	释放此 Graphics 使用的所有资源。
DrawArc(Pen, Rectangle, Single, Single)	绘制一段弧线，它表示 Rectangle 结构指定的椭圆的一部分。
DrawBeziers(Pen, Point[])	用 Point 结构数组绘制一系列贝塞尔样条。
DrawClosedCurve(Pen, Point[])	绘制由 Point 结构的数组定义的闭合基数样条。
DrawCurve(Pen, Point[])	绘制经过一组指定的 Point 结构的基数样条。
DrawEllipse(Pen, Int32, Int32, Int32, Int32)	绘制一个由边框定义的椭圆，该边框由矩形的左上角坐标、高度和宽度指定。
DrawIcon(Icon, Rectangle)	在 Icon 结构指定的区域内绘制指定的 Rectangle 表示的图像。
DrawIconUnstretched(Icon, Rectangle)	绘制指定的 Icon 表示的图像，而不缩放该图像。
DrawImage(Image, Rectangle)	在指定位置并且按指定大小绘制指定的 Image。
DrawImageUnscaled(Image, Rectangle)	在指定的位置使用图像的原始物理大小绘制指定的图像。
DrawLine(Pen, Point, Point)	绘制一条连接两个 Point 结构的线。
DrawPath(Pen, GraphicsPath)	绘制 GraphicsPath。
DrawPie(Pen, Rectangle, Single, Single)	绘制由一个 Rectangle 结构和两条射线所指定的椭圆定义的扇形。
DrawPolygon(Pen, Point[])	绘制由一组 Point 结构定义的多边形。
DrawRectangle(Pen, Rectangle)	绘制由 Rectangle 结构指定的矩形。
DrawString(String, Font, Brush, PointF, StringFormat)	使用指定 StringFormat 的格式化特性，用指定的 Brush 和 Font 对象在指定的位置绘制指定的文本字符串。
EndContainer(GraphicsContainer)	关闭当前图形容器，并将此 Graphics 的状态还原到通过调用 BeginContainer 方法保存的状态。
ExcludeClip(Rectangle)	更新此 Graphics 的剪辑区域，以排除 Rectangle 结构所指定的区域。
FillClosedCurve(Brush, Point[])	填充由 Point 结构数组定义的闭合基数样条曲线的内部。
FillEllipse(Brush, Rectangle)	填充 Rectangle 结构指定的边框所定义的椭圆的内部。
FillPath(Brush, GraphicsPath)	填充 GraphicsPath 的内部。

FillPolygon(Brush, PointF[])	填充 PointF 结构指定的点数组所定义的多边形的内部。
FillRectangle(Brush, Rectangle)	填充 Rectangle 结构指定的矩形的内部。
FillRegion(Brush, Region)	填充 Region 的内部。
Flush()	强制执行所有挂起的图形操作并立即返回而不等待操作完成。
FromHdc(IntPtr)	从设备上下文的指定句柄创建新的 Graphics。
FromImage(Image)	从指定的 Image 创建新的 Graphics。
GetHdc()	获取与此 Graphics 关联的设备上下文的句柄。
GetNearestColor(Color)	获取与指定的 Color 结构最接近的颜色。
GetType()	获取当前实例的 Type。
IntersectClip(Rectangle)	将此 Graphics 的剪辑区域更新为当前剪辑区域与指定 Rectangle 结构的交集。
IsVisible(Int32, Int32)	指示由一对坐标指定的点是否包含在此 Graphics 的可见剪辑区域内。
MeasureCharacterRanges(String, Font, RectangleF, StringFormat)	获取 Region 对象的数组，其中每个对象将字符位置的范围限定在指定字符串内。
MeasureString(String, Font)	测量用指定的 Font 绘制的指定字符串。
MultiplyTransform(Matrix)	将此 Graphics 的世界变换乘以指定的 Matrix。
ReleaseHdc()	释放通过以前对此 Graphics 的 GetHdc 方法的调用获得的设备上下文句柄。
ReleaseHdc(IntPtr)	释放通过以前对此 Graphics 的 GetHdc 方法的调用获得的设备上下文句柄。
ResetClip()	将此 Graphics 的剪辑区域重置为无限区域。
ResetTransform()	将此 Graphics 的世界变换矩阵重置为单位矩阵。
Restore(GraphicsState)	将此 Graphics 的状态还原到 GraphicsState 表示的状态。
RotateTransform(Single)	将指定旋转应用于此 Graphics 的变换矩阵。
Save()	保存此 Graphics 的当前状态，并用 GraphicsState 标识保存的状态。
ScaleTransform(Single, Single)	将指定的缩放操作应用于此 Graphics 的变换矩阵，方法是将该对象的变换矩阵左乘该缩放矩阵。
SetClip(Graphics)	将此 Graphics 的剪辑区域设置为指定 Graphics 的 Clip 属性。
TransformPoints(CoordinateSpace, CoordinateSpace, Point[])	使用此 Graphics 的当前世界变换和页变换，将点数组从一个坐标空间转换到另一个坐标空间。
TranslateClip(Int32, Int32)	将此 Graphics 的剪辑区域沿水平方向和垂直方向平移指定的量。
TranslateTransform(Single, Single)	通过使此 Graphics 的变换矩阵左乘指定的平移来更改坐标系统的原点。

Graphic 类有相关属性可以配置绘图的质量，默认情况下绘图一般有锯齿，但是我们可以将其绘图 SmoothingMode 设置为 HighQuality 或者 AntiAlias，即可较为明显的消除锯齿。下面我们用 Graphic 类在不同的绘图质量下绘制椭圆：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Drawing.Drawing2D
using System.Drawing.Text;
using System.Windows.Forms;
namespace CustomControlsTest
{
    public partial class Form3 : Form
    {
        public Form3()
        {
            InitializeComponent();
        }
        private void button1_Click(object sender, EventArgs e)
        {
            Pen p = new Pen(Color.Black, 5);
            Graphics g = this.groupBox1.CreateGraphics();
            SizeF size = this.groupBox1.Size;
            g.SmoothingMode = SmoothingMode.None;
            Rectangle rec=this.groupBox1.ClientRectangle;
            Rectangle rec2=new Rectangle(10,15,rec.Width-20,rec.Height-20);
            g.DrawArc(p, rec2, 0, 360);
            g = this.groupBox2.CreateGraphics();
           // g.SmoothingMode = SmoothingMode.AntiAlias;
            g.SmoothingMode = SmoothingMode.HighQuality;
            rec = this.groupBox2.ClientRectangle;
            rec2 = new Rectangle(10, 15, rec.Width - 20, rec.Height - 20);
            g.DrawArc(p, rec2, 0, 360);
            g = this.groupBox3.CreateGraphics();
            g.TextRenderingHint = TextRenderingHint.ClearTypeGridFit;
           // StringFormat sf = new StringFormat(StringFormatFlags.NoWrap);
            StringFormat CenterSF = new StringFormat
             {
                 Alignment = StringAlignment.Center,
                 LineAlignment = StringAlignment.Center
             };
            g.DrawString("DrawString", new Font("微软雅黑", 20),
              new SolidBrush(Color.Black), 100, 50, CenterSF);
            g = this.groupBox4.CreateGraphics();
```

```
            g.TextRenderingHint = TextRenderingHint.SingleBitPerPixelGridFit;
            g.DrawString("DrawString", new Font("微软雅黑", 20),
              new SolidBrush(Color.Black), 100, 50, CenterSF);
            g.Dispose();
            p.Dispose();
        }
    }
}
```

运行该窗体，界面如图 3.6 所示。

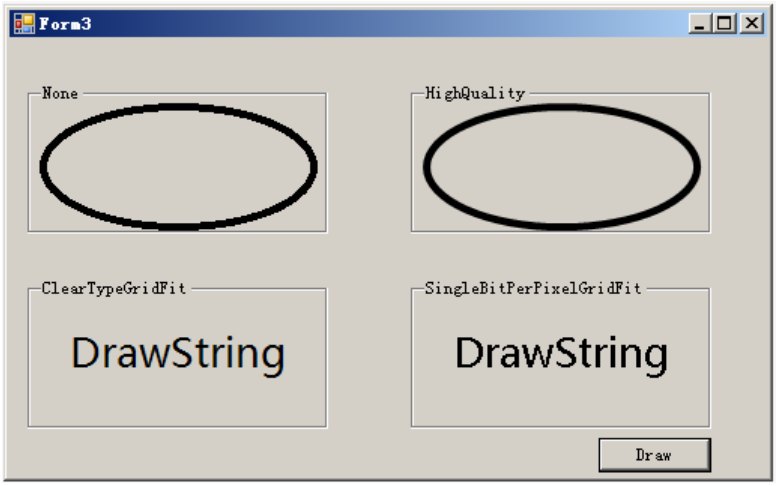


图 3.6 不同绘图质量下的绘图效果

在自定义控件开发中，有些情况下，需要输出不同格式的文本，Graphic 对象可以对文本按不同方向进行输出，也可以设置文本的颜色和字体：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace CustomControlsTest
{
    public partial class Form5 : Form
    {
        public Form5()
        {
            InitializeComponent();
        }
        protected override void OnPaint(PaintEventArgs e)
        {
```

```
            StringFormat stringFormat = new StringFormat();
            stringFormat.Alignment = StringAlignment.Center;
            stringFormat.LineAlignment = StringAlignment.Center;
            // 垂直方向且不换行
            stringFormat.FormatFlags =
          StringFormatFlags.DirectionVertical | StringFormatFlags.NoWrap;
            //截断后用省略号...
            stringFormat.Trimming = StringTrimming.EllipsisCharacter;
            Rectangle rec = this.ClientRectangle;
            e.Graphics.DrawString("GDI含义是图形设备接口，主要任务是负责系统与绘图程序之间
的信息交换，处理所有Windows程序的图形输出。",
                new Font("微软雅黑", 12), new SolidBrush(Color.Black),
                rec, stringFormat);
            base.OnPaint(e);
        }
    }
}
```

运行该窗体，界面如图 3.7 所示。

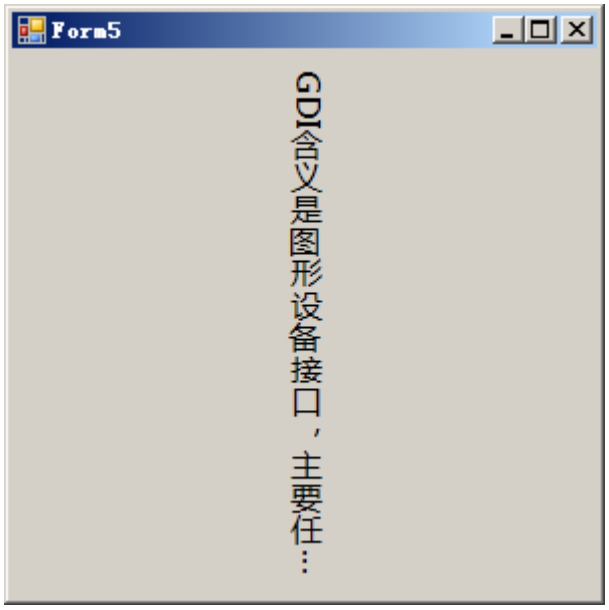


图 3.7 Graphic文本绘制示例

3.7 GraphicPath 类

GraphicPath 由一系列相互连接的直线、曲线连接起来组成的开放（非闭合）图形。创建路径时就会隐式创建一个新图形（由上面的直线、曲线等组成）。也可以显示地声明 StartFigure。图形具有方向，其先后顺序加入的直线、曲线等就表明了次序。一般图形路径是开放的，由起点，到最后图形（终点）。也可以用 ClosedFigure 显式声明为闭合图

形（比如填充和剪辑时要用）：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace CustomControlsTest
{
    public partial class Form5 : Form
    {
        public Form5()
        {
            InitializeComponent();
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            System.Drawing.Drawing2D.GraphicsPath gp =
              new System.Drawing.Drawing2D.GraphicsPath();
            // 创建一个开口的图形
            gp.AddLine(10, 10, 10, 50);
            gp.AddLine(10, 50, 50, 50);
            gp.AddLine(50, 50, 50, 10);
            // 开始一个新的图形
            gp.StartFigure();
            gp.AddLine(60, 10, 60, 50);
            gp.AddLine(60, 50, 100, 50);
            gp.AddLine(100, 50, 100, 10);
            gp.CloseFigure();
            Rectangle r = new Rectangle(110, 10, 40, 40);
            gp.AddEllipse(r);
            //填充path
            e.Graphics.FillPath(Brushes. Red, gp);
            //绘制path
            e.Graphics.DrawPath(Pens.Black, gp);
            //释放资源
            gp.Dispose();
            base.OnPaint(e);
        }
    }
}
```

运行该窗体，界面如图 3.8 所示。

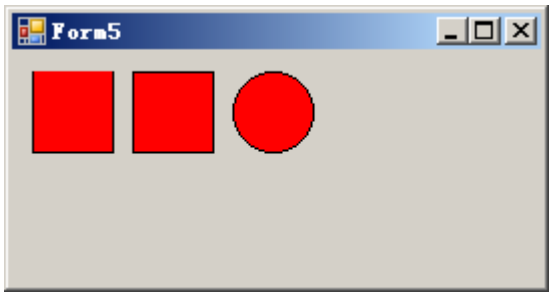


图 3.8 GraphicPath绘制示例

GraphicPath 除了可以组合各种直线、矩形等形状，更加灵活的是可以用 GraphicPath.AddString 添加字符串，然后对字符串构成的 path 进行操作，请看下面的代码：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Drawing.Drawing2D;
namespace CustomControlsTest
{
    public partial class Form5 : Form
    {
        public Form5()
        {
            InitializeComponent();
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            GraphicsPath path = new GraphicsPath();
            path.AddString("I \uF08A China",
              new FontFamily("FontAwesome"), 0, 70, new Point(10, 30),
              new StringFormat());
            e.Graphics.DrawPath(Pens.Blue, path);
            //设置剪辑Clip
            Region clippingRegion = new Region(path);
            e.Graphics.Clip = clippingRegion;
            //笔刷样式
            HatchStyle brushStyle = HatchStyle.WideDownwardDiagonal;
            HatchBrush brush =new HatchBrush(brushStyle, Color.Red, Color.Pink);
```

```
            e.Graphics.FillRectangle(brush, this.ClientRectangle);
            clippingRegion.Dispose();
            path.Dispose();
            base.OnPaint(e);
        }
    }
}
```

运行该窗体，界面如图 3.9 所示。

图 3.9 GraphicPath String绘制示例

3.8 Region 类

GDI+绘图过程中我们有时候需要让绘图实现类似于 XOR 那种颜色透明叠加的效果，或者制作无规则窗体控件等，这时可以采用 Region 对象。Region 对象需引用 System.Drawing.Region 空间。Region 常用的有以下方法：Union、Xor、Exclude、Intersect 等，可实现交集、差集、并集等。

```
Region region1 = new Region(new Rectangle(0, 0, 100, 100));
Region region2 = new Region(new Rectangle(50, 50, 100, 100));
//填充region
SolidBrush brush1 = new SolidBrush(Color.FromArgb(180, 255, 0, 0));
e.Graphics.FillRegion(brush1, region1);
SolidBrush brush2 = new SolidBrush(Color.FromArgb(180, 0, 255, 0));
e.Graphics.FillRegion(brush2, region2);
```

上述代码创建了两个半透明的色块，有一部分相互交叠，如图 3.10 所示。

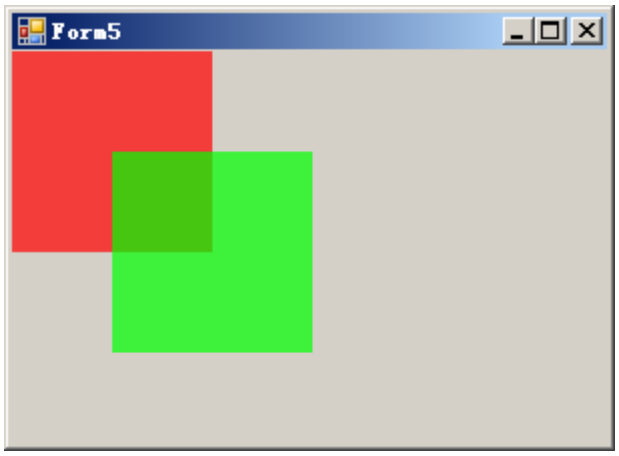


图 3.10 Region填充示例

```
Region region1 = new Region(new Rectangle(0, 0, 100, 100));
Region region2 = new Region(new Rectangle(50, 50, 100, 100));
//用Xor函数合并区域，得到的是两个区域未重叠部分的区域.
region1.Xor(region2);
SolidBrush brush = new SolidBrush(Color.Red);
e.Graphics.FillRegion(brush, region1);
```

上述代码创建了两个相互交叠的色块，Region 进行 Xor 操作后，如图 3.11 所示。

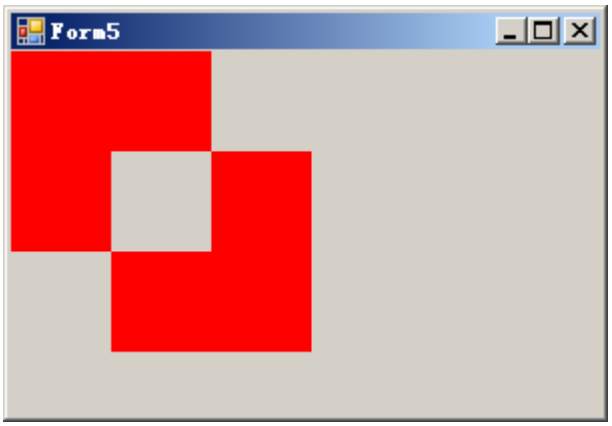


图 3.11 Region Xor填充示例

```
Region region1 = new Region(new Rectangle(0, 0, 100, 100));
Region region2 = new Region(new Rectangle(50, 50, 100, 100));
//在region2中排除region1和region2交叠的部分
region1.Complement(region2);
SolidBrush brush = new SolidBrush(Color.Red);
e.Graphics.FillRegion(brush, region1);
```

上述代码创建了两个相互交叠的色块，Region 进行 Complement 操作后，如图 3.12 所示。

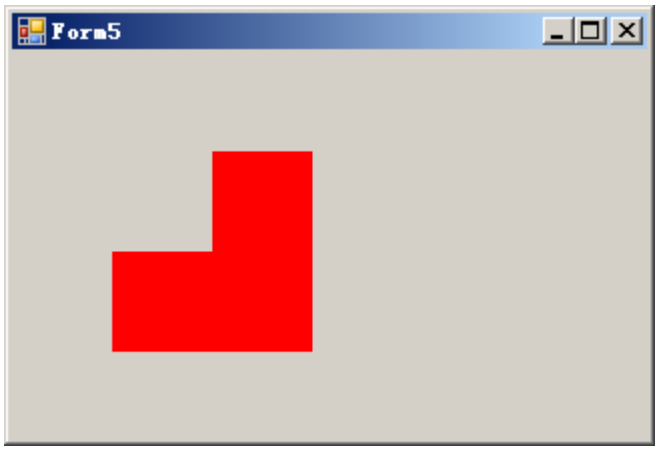


图 3.12 Region Complement填充示例

```
Region region1 = new Region(new Rectangle(0, 0, 100, 100));
Region region2 = new Region(new Rectangle(50, 50, 100, 100));
//用Intersect获取region1和region2交叠的部分
region1.Intersect(region2);
SolidBrush brush = new SolidBrush(Color.Red);
e.Graphics.FillRegion(brush, region1);
```

上述代码创建了两个相互交叠的色块，Region 进行 Intersect 操作后，如图 3.13 所示。

图 3.13 Region Intersect填充示例

```
Region region1 = new Region(new Rectangle(0, 0, 100, 100));
Region region2 = new Region(new Rectangle(50, 50, 100, 100));
//用Union获取region1和region2的合并区域
region1.Union(region2);
SolidBrush brush = new SolidBrush(Color.Red);
e.Graphics.FillRegion(brush, region1);
```

上述代码创建了两个相互交叠的色块，Complement 进行 Union 操作后，如图 3.14 所示。

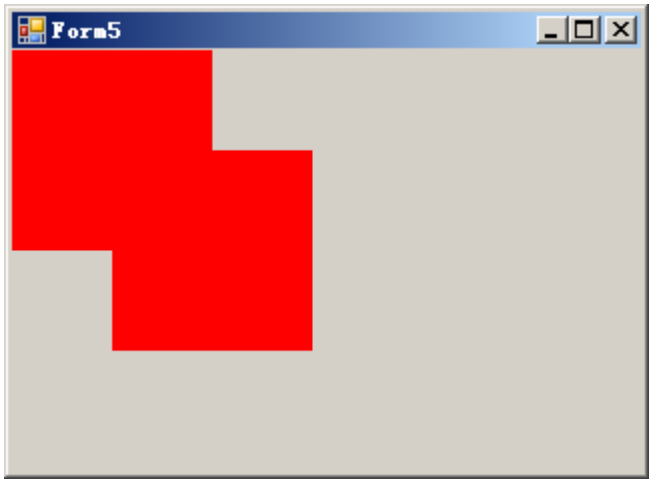


图 3.14 Region Union填充示例

3.9 坐标体系和变换

在Windows Forms中，坐标点有两个公共整型属性，用大写X和Y表示（也就是数学上熟悉的横坐标、纵坐标，但是和数学的坐标计量规则不同）。在Windows Forms 中的坐标系分为三类：

- 屏幕坐标系，以显示屏左上角为(0,0)点的坐标
- 窗体坐标系，以 Windows Forms 窗体左上角为(0,0)点
- 窗体控件坐标系，以该控件的左上角为(0,0)点

以上三类的坐标原点都是其左上角（屏幕左上角、窗体左上角、控件左上角）。在定义控件或者窗体各子控件的布局时，必须要了解这些坐标体系，不然就会出现布局混乱的情况。下面我们用一个例子来说明坐标体系。

这个例子稍微有点复杂，它用AutoScrollMinSize属性设置了最小滚动所需的大小，这个例子绘制一个矩形和椭圆，该窗体的大小默认比AutoScrollMinSize小，这样就可以出现滚动条，代码如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace ReflectionDemo
{
```

```
public partial class Form1 : Form
{
    private Point rectangleTopLeft = new Point(0, 0);
    private Size rectangleSize = new Size(400, 300);
    private Point ellipseTopLeft = new Point(100, 300);
    private Size ellipseSize = new Size(200, 200);
    private Pen bluePen = new Pen(Color.Blue, 3);
    private Pen redPen = new Pen(Color.Red, 2);
    public Form1()
    {
        InitializeComponent();
        //显示scrollviewer
        this.AutoScrollMinSize = new Size(400, 600);
    }
    private void FrmScrollable_Load(object sender, EventArgs e)
    {
        //滚动到指定的位置
        this.HorizontalScroll.Value = 100;
        this.VerticalScroll.Value = 300;
    }
    protected override void OnPaint(PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        Size ScrollOffset = new Size(this.AutoScrollPosition);
        this.Text = ScrollOffset.ToString();
        if (e.ClipRectangle.Top + ScrollOffset.Width < 400 ||
           e.ClipRectangle.Left + ScrollOffset.Height < 600)
        {
            Rectangle RectangleArea = new Rectangle
               (rectangleTopLeft + ScrollOffset, rectangleSize);
            Rectangle EllipseArea = new Rectangle
               (ellipseTopLeft + ScrollOffset, ellipseSize);
            g.DrawRectangle(bluePen, RectangleArea);
            g.DrawEllipse(redPen, EllipseArea);
        }
        base.OnPaint(e);

    }
}
}
```

用代码this.HorizontalScroll.Value = 100和this.VerticalScroll.Value = 300将窗体的可见区域左上点位到坐标（100,300）处，如下图3.15所示：

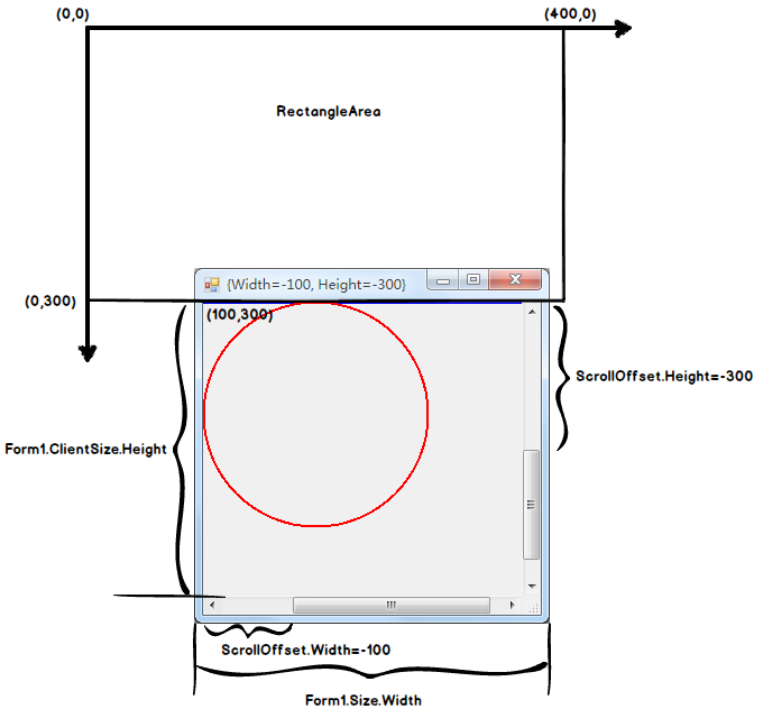


图 3.15 Form在滚动条情况下的几何关系

了解了窗体坐标体系后，我们再来说一下如何进行坐标变换，变换常用的为平移、旋转和缩放。我们先看一下平移变换如何影响GDI+绘制图形。我们先定义一个绘制矩形的方法，该方法就是绘制固定大小和位置的矩形。然后通过坐标平移后来调用该绘制矩形的方法，看一下会有什么效果，代码如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace CustomControlsTest
{
    public partial class Form6 : Form
    {
        public Form6()
        {
            InitializeComponent();
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            base.OnPaint(e);
            //移动坐标系并绘制矩形
```

```
            DrawRectangle(e.Graphics);
            e.Graphics.TranslateTransform(60, 60);
            DrawRectangle(e.Graphics);
            e.Graphics.TranslateTransform(-60, 60);
            DrawRectangle(e.Graphics);
            e.Graphics.TranslateTransform(120, 10);
            DrawRectangle(e.Graphics);
        }
        private void DrawRectangle(Graphics g)
        {
            Pen drawingPen = new Pen(Color.Red, 2);
            //绘制并填充固定位置和大小的矩形
            g.DrawRectangle(drawingPen, new Rectangle(0, 0, 60, 60));
            g.FillRectangle(Brushes.Red, new Rectangle(0, 0, 60, 60));
            drawingPen.Dispose();
        }
    }
}
```

运行窗体，结果如图 3.18 所示。

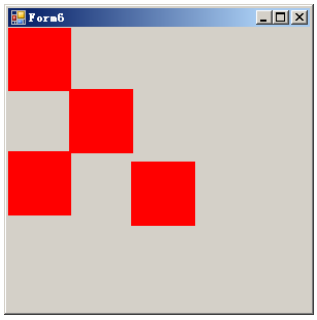


图 3.16 坐标平移绘图示例

通过输出结果可以发现，虽然绘制矩形的方法绘制的矩形的位置和大小都是固定的，但是界面上却出现了 4 个位置的矩形，e.Graphics.TranslateTransform(X, Y)方法会将坐标原点移动到原有坐标的(X,Y)处，而且该方法是有累加效果，如果第一次移动为（20,30），第二次移动为（30,40），那么叠加效果为（50,70）。

接下来我们看一下坐标的旋转变换对于绘图的影响，还是接着上面的例子，将平移变换改成旋转变换，代码如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
```

```
using System.Text;
using System.Windows.Forms;
namespace CustomControlsTest
{
    public partial class Form5 : Form
    {
        public Form5()
        {
            InitializeComponent();
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            base.OnPaint(e);
            e.Graphics.TranslateTransform(120,120);
            DrawRectangle(e.Graphics);
            e.Graphics.RotateTransform(45);
            DrawRectangle(e.Graphics);
        }
        private void DrawRectangle(Graphics g)
        {
            Pen drawingPen = new Pen(Color.Red, 2);
            //固定位置和大小
            g.DrawRectangle(drawingPen, new Rectangle(0, 0, 60, 60));
            drawingPen.Dispose();
        }
    }
}
```

运行窗体，结果如图 3.19 所示。

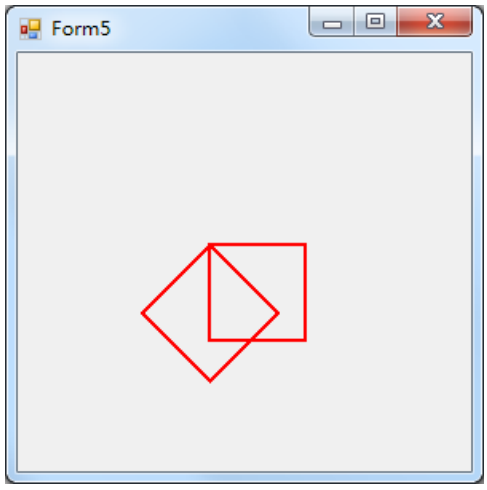


图 3.17 坐标旋转绘图示例

通过输出结果可以发现，虽然绘制矩形的方法绘制的矩形的位置和大小都是固定的，但

是界面上却出现了 2 个角度的矩形，e.Graphics.RotateTransform(degree)方法安按照顺时针方向旋转，旋转的点为当前坐标系的原点（0,0）为了直观的观察，先用 e.Graphics.TranslateTransform(120,120)将当前坐标系的原点移动到(120,120)，然后进行绘图和旋转，当执行 e.Graphics.RotateTransform(45)方法时会在窗体坐标点（120,120）处进行旋转 45 度。

最后我们将一下如何用 GDI+实现缩放。下面就用 Rectangle.Inflate()方法对矩形进行放大，代码如下所示；

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace CustomControlsTest
{
    public partial class Form6 : Form
    {
        public Form6()
        {
            InitializeComponent();
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            base.OnPaint(e);
            e.Graphics.TranslateTransform(120, 20);
            DrawRectangle(e.Graphics);
        }
        private void DrawRectangle(Graphics g)
        {
            Pen drawingPen = new Pen(Color.Red, 2);
            Pen drawingPen2 = new Pen(Color.Black, 2);
            Rectangle rec=new Rectangle(0, 0, 60, 60);
            Rectangle rec2=rec;
            //缩放
            rec2.Inflate(6,6);
            g.DrawRectangle(drawingPen,rec );
            g.DrawRectangle(drawingPen2,rec2);
            drawingPen.Dispose();
        }
```

```
    }
}
```

运行该示例，界面如图 3.18 所示：

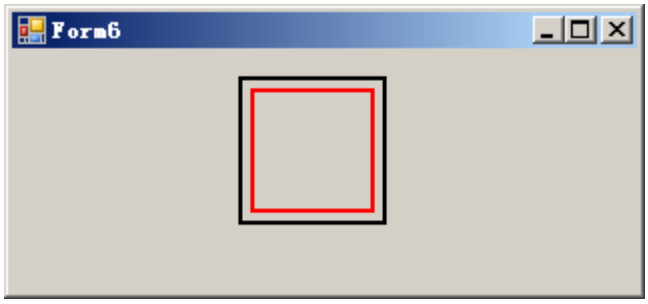


图 3.18 绘图缩放示例

通过输出结果可以发现，rec2.Inflate(6,6)让原有的矩形向外扩大了 6 个单位。

3.10 双缓冲

在 GDI+绘图过程中，如果不做双缓冲（Double Buffering）处理，那么可能会发现窗体在大小调整等操作导致界面重绘的时候，界面会闪烁，为了解决这个闪烁的问题，建议将控件的双缓冲机制打开：

```
//不重绘控件背景
this.SetStyle(ControlStyles.AllPaintingInWmPaint, true);
//控件开启双缓冲
this.DoubleBuffered = true;
```

另外，我们自定义控件的时候，也可以创建双缓冲控件，对于开启双缓冲的控件来说，可以在内存中创建图形，然后将其创建好的图形一次性绘制到界面上，从而减少闪烁。下面的代码将大量应用于第四章的自定义绘制的控件开发中：

```
Bitmap bitmap = new Bitmap(this.ClientRectangle.Width,
this.ClientRectangle.Height);
Graphics  g = Graphics.FromImage(bitmap);
//将内存中创建完成的图形拷贝到界面上
e.Graphics.DrawImageUnscaled(bitmap, 0, 0);
g.Dispose();
bitmap.Dispose();
```

3.11 局部刷新

我们知道，在Web应用中,Ajax技术可以让页面进行局部刷新，从而提高用户体验和处理效率，其实在Windows Forms中也有“局部刷新”的概念，Windows Forms中的局部刷新也

能降低资源占用，提高界面渲染效率。请看下面的代码：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Drawing.Drawing2D;
namespace CustomControlsTest
{
    public partial class Form9 : Form
    {
        public Form9()
        {
            InitializeComponent();
            this.MouseDown += Form9_MouseDown;
        }
        private int num = 0;
        void Form9_MouseDown(object sender, MouseEventArgs e)
        {
            if (e.Button == MouseButtons.Left)
            {
                Rectangle rec = new Rectangle(e.X, e.Y, 20, 20);
                listRec.Add(rec);
                rec.Inflate(2, 2);
                //刷新局部
                //Invalidate(rec);
                //全部刷新，当有90个以上有明显闪烁
                //Invalidate();
            }
            else if (e.Button == MouseButtons.Right)
            {
                num = 0;
                foreach (Rectangle rec in listRec)
                {
                    num++;
                    if (rec.Contains(new Point(e.X, e.Y)))
                    {
                        this.Text = string.Format("Click Rect # {0}", num.ToString());
                    }
                }
            }
```

```
        }
        private List<Rectangle> listRec = new List<Rectangle>();
        private GraphicsPath path;
        protected override void OnPaint(PaintEventArgs e)
        {
            base.OnPaint(e);
            foreach (Rectangle rec in listRec)
            {
                DrawRect(e.Graphics, rec);
            }
        }
        private void DrawRect(Graphics g, Rectangle rec)
        {
           g.DrawRectangle(Pens.Black, rec);
        }
    }
}
```

运行窗体，结果如图 3.19 所示。

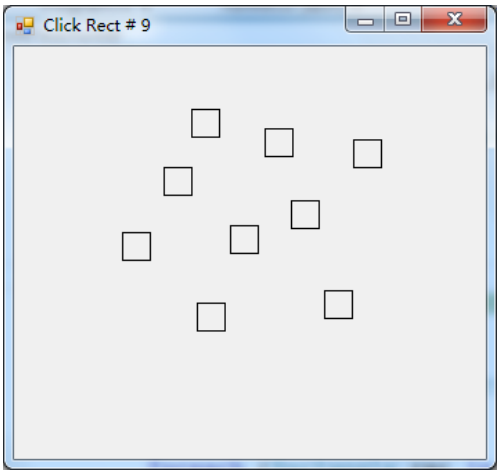


图 3.19 局部刷新示例

上面的代码中，Invalidate(rec)可以让界面只刷新对于区域的内容，而Invalidate()则刷新整个界面。对于上面的代码，再测试过程中，当有90个级以上的矩形时，再次单击界面区域添加矩形时，可以发现有较为明显的闪烁，但是用Invalidate(rec)方法来进行局部刷新却没有见闪烁。

3.12 命中测试

命中测试（Hit Testing）在某些高级的自定义控件中会用到，当自定义控件中有多个需要交互的区域且需要相应的要求不同，那么控件必须能够区别当前用户所命中的区域是什

么才能正确做出响应。举个例子，加入我们开发一个地图控件，地图上不同区域代表不同的地理位置，需要用不同的颜色进行显示，而且当用户鼠标进入到对于的地理位置后，需要提示该地区的人口信息，那么程序就需要用到命中测试来确定当前用户的鼠标进入到哪个区域，下面给出一个示例代码：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Drawing.Drawing2D;
namespace CustomControlsTest
{
    public partial class Form9 : Form
    {
        public Form9()
        {
            InitializeComponent();
            this.MouseDown += Form9_MouseDown;
        }
        private int num = 0;
        void Form9_MouseDown(object sender, MouseEventArgs e)
        {
            if (path1.IsVisible(e.X, e.Y))
            {
                this.Text = "Click In Path 1";
            }
            if (path2.IsVisible(e.X, e.Y))
            {
                this.Text = "Click In Path 2";
            }
        }
        private List<Rectangle> listRec = new List<Rectangle>();
        private GraphicsPath path1;
        private GraphicsPath path2;
        protected override void OnPaint(PaintEventArgs e)
        {
            base.OnPaint(e);
            e.Graphics.SmoothingMode = SmoothingMode.AntiAlias;
            path1 = new System.Drawing.Drawing2D.GraphicsPath();
```

```
            path1.StartFigure();
            path1.AddArc(20, 20, 90, 90, 30, 60);
            path1.AddLine(20, 50, 70, 230);
            path1.CloseFigure();
            e.Graphics.FillPath(Brushes.White, path1);
            path2 = new GraphicsPath();
            path2.AddRectangle(new Rectangle(150, 60, 90, 90));
            e.Graphics.FillPath(Brushes.Black, path1);
            e.Graphics.FillPath(Brushes.Black, path2);
        }
        private void DrawRect(Graphics g, Rectangle rec)
        {
           g.DrawRectangle(Pens.Black, rec);
        }
    }
}
```

运行窗体，结果如图 3.20 所示。

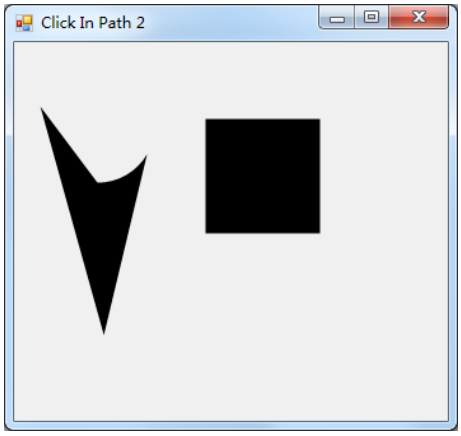


图 3.20 命中测试示例

当用户在区域单击时，程序会用path1.IsVisible(e.X, e.Y)和path2.IsVisible(e.X, e.Y)来判断当前的鼠标坐标位于哪个区域中，然后做出相应，在窗口的标题上显示单击的path为1还是2。

3.13 不规则窗体

这一节我们讨论一个比较有意思的话题，窗体一般都是矩形的，但是窗体能不能是一个不规则的形状呢？答案是肯定的。我们可以通过设置窗体的Region 属性来让窗体显示出很有个性的不规则形状，形状由Region决定，下面给出一个示例代码：

```
using System;
using System.Collections.Generic;
```

```
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Ling;
using System.Text;
using System.Windows.Forms;
using System.Drawing.Drawing2D;
namespace CustomControlsTest
{
    public partial class Form10 : Form
    {
        public Form10()
        {
            InitializeComponent();
        }
        private void Form10_Load(object sender, EventArgs e)
        {
            this.BackColor = Color.DarkBlue;
            GraphicsPath path = new GraphicsPath();
            path.AddEllipse(30,30, this.Width/2, this.Height/2);
            path.AddEllipse(80, 80, this.Width / 2, this.Height / 2);
             //将path定义的区域作为窗体的可见区域（不规则窗体）
            this.Region = new Region(path);
            this.TopMost = true;
        }
    }
}
```

运行该示例，界面如图3.21所示：

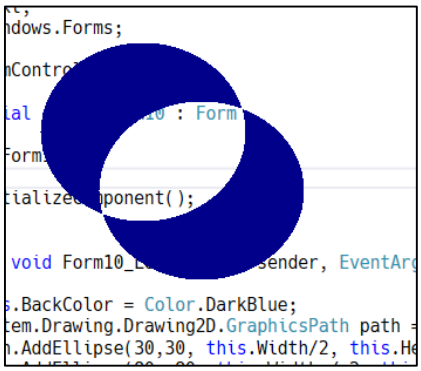


图 3.21 不规则窗体示例

3.14 本章小结

本章是本书的一个重点，较为全面的介绍了GDI+的知识和应用。利用GDI+,我们可以绘

制直线、曲线、矩形和椭圆等形状，并且可以绘制路径，并填充区域。也可以利用GDI+对绘图进行平移、缩放和旋转，也可以对文本进行绘制。

另外介绍了坐标体系和变换，以及为了减少界面闪烁而启用的双缓冲机制，为了提高绘图效率，可以在绘图时进行局部刷新，而不是重绘整个界面。最后介绍了如何进行命中测试和实现不规则的窗体。

第四章 Form 控件开发

上一章重点对 GDI+相关知识做了较为全面的介绍，这些 GDI+知识是这章控件开发的基础，在阅读本章之前，请务必阅读和理解上一章内容。这一章我们将重点讲解如何开发自定义的 Windows Forms 控件。

4.1 LabelTextBox 控件

在大部分的窗体开发中，我们都需要一个卡片窗体来编辑单据，卡片窗体上大量出现标签（Label）和文本框(TextBox)配对出现的情况。基于原生的窗体控件，我们每次都需要拖动这两个控件。那么我们能不能将 Label 和 TextBox 控件组合成一个控件呢？答案是肯定的。下面就自定义一个组合控件，这里我们取一个直观的名字为 LabelTextBox 控件。

在做任何管理信息系统之前，我们必须要了解和明确需求。类似的，开发自定义控件，我们第一步也是要明确控件的功能和 UI 设计。下面的内容都是按照控件功能(功能性需求)、控件设计（UI 设计）、控件开发（代码）和控件应用（用例代码）这几部分来对自定义控件开发进行讲解。

4.1.1 控件功能

LabelTextBox 控件的核心功能是：一次拖放，可以同时生成一个 Label 和 TextBox 控件，并可以分别设置 Label 和 TextBox 的 Text 属性。当控件大小变化时，可以自动将 Label 和 TextBox 居中显示。为了扩展 TextBox 的功能，提供验证规则配置，可以用正则表达式来控制可输入的文本内容，比如 Email 等，当输入不符合配置时，文本框下面会出现一个红色的波浪线以提醒用户输入内容校验失败。

4.1.2 控件设计

第一章提到 WinForm 控件通常有三种类型：复合控件（Composite Controls）、扩展控件（Extended Controls）和自定义控件（Custom Controls）。其中复合控件就是将现有的各种控件组合起来，形成一个新的控件。通过上面的控件功能可以知道，这里就是组合 Label 和 TextBox 控件的功能。因此，LabelTextBox 控件可以说是一种复合控

件。

在 Visual Studio 2010 中（除非特殊说明，所有控件开发所用的 Visual Studio 版本为 2010），复合控件的创建可以通过【用户控件】模板来创建，也就是继承自 UserControl。

在实际生产中，一般都是将自定义的控件编译为一个 dll 来使用，也就是放在一个类库中。由于窗体控件开发需要引入一些标准类库外的窗体控件相关的库（System.Drawing 和 System.Windows.Forms 等），为了减少导入库的操作，可以新建一个窗体应用程序，然后将项目属性中的输出类型设置为类库即可。

另外为了测试控件，需要创建一个窗体应用程序来对自定义的控件进行调试。此应用程序只需要引入上面的类库即可。

根据前面的功能要求，LabelTextBox 控件的 UI 设计如下图 4.1 所示：

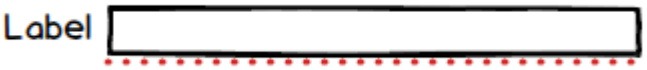


图 4.1 LabelTextBox 控件 UI

4.1.3 控件开发

按照上面的建议，首先创建一个名为 CustomControls 的窗体应用程序（后续除非有特殊说明，不然控件测试和演示都是在该项目下），然后添加一个名为 MingForm.Controls 的窗体应用程序（后续除非有特殊说明，不然控件的开发都在该项目下），并且在其属性页中设置其输出类型为类库，删除默认的 Form1 窗体。这两个应用程序的目标框架都设置为.NET Framework 4。

在 MingForm.Controls 项目下新建一个 Images 文件夹，存放需要的图片资源。创建的控件可以配置一个图标（16x16 像素），此图标一般命名和自定义控件同名（后缀不同而已），并作为嵌入的资源进行处理。最后创建一个名为 LabelTextBox 的 UserControl 控件，如下图 4.2 所示。

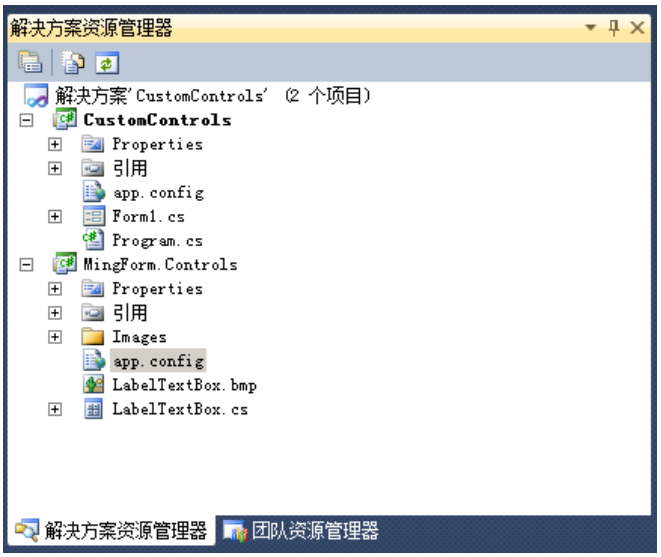


图 4.2 项目结构

创建 LabelTextBox 后，我们根据该控件的 UI 设计对其进行界面进行布局，控件设计界面如图 4.3 所示：

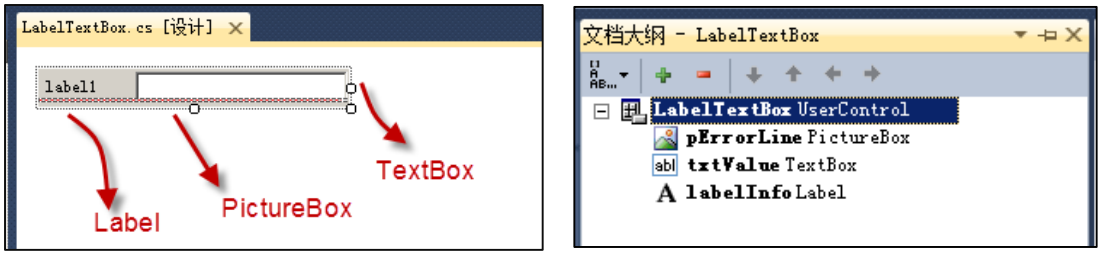


图 4.3 LabelTextBox UI 实现

最左边拖放一个 Label 控件(labelInfo),后面拖放一个 TextBox 控件(txtValue),下面拖放一个 PictureBox 控件（pErrorLine），用于显示波浪线，默认其可见设置为隐藏。

下面给出 LabelTextBox 的核心代码：

```
[Browsable(true)]
[Category("LabelText")]
[Description("标签的文本")]
public string LabelText
{
    get
    {
        return this.labelInfo.Text;
    }
    set
    {
        this.labelInfo.Text = value;
```

```
    }
}
[Browsable(true)]
[Category("LabelText")]
[Description("文本")]
[DefaultValue("LabelText")]
public override string Text
{
    get
    {
        return this.txtValue.Text;
    }
    set
    {
        this.txtValue.Text = value;
    }
}
private bool _isError = true;
[Browsable(true)]
[Category("LabelText")]
[Description("是否显示波浪线")]
[DefaultValue(false)]
public bool isError
{
    get
    {
        return _isError;
    }
    set
    {
        if(value==true)
        {
            this.pErrorLine.Visible = true;
        }
        else
        {
            this.pErrorLine.Visible = false;
        }
        _isError = value;
    }
}
//支持的验证类型
public enum TYPE_DATA
{
```

```
  CUSTOM,
  INTEGER,
  DECIMAL,
  DECIMAL_RESTRICTIVE,
  PERCENT,
  STRING,
  EMAIL,
  URL
}
[Browsable(true)]
[Category("LabelText")]
[Description("TextBox验证类型")]
public TYPE_DATA TypeOfData
{
    get
    {
        return this._typedata;
    }
    set
    {
        this._typedata = value;
    }
}
[Browsable(true)]
[Category("TextChanged Custom Event")]
public new event EventHandler TextChanged;
protected override void OnSizeChanged(EventArgs e)
{
    base.OnSizeChanged(e);
    this.Height = this.txtValue.Height+2;
    this.labelInfo.Left = 1;
    this.labelInfo.Top = (this.Height - this.labelInfo.Height) / 2;
    this.labelInfo.Width = LabelWidth;
    this.txtValue.Top = (this.Height - this.txtValue.Height) / 2;
    this.txtValue.Left = this.labelInfo.Right + GapWidth;
    this.txtValue.Width = this.Width - this.labelInfo.Width - _gapWidth - 2;
    // GapWidth为Label和TextBox的间距属性
    this.pErrorLine.Width = this.txtValue.Width-2;
    this.pErrorLine.Left = this.txtValue.Left + 1;
    this.pErrorLine.Top = this.Height - 5;
}
//字体更改后，重新布局控件位置
protected override void OnFontChanged(EventArgs e)
{
```

```
    base.OnFontChanged(e);
    OnSizeChanged(e);
}
public LabelTextBox()
{
    InitializeComponent();
    //激活double buffering
    this.SetStyle(ControlStyles.OptimizedDoubleBuffer |
        ControlStyles.AllPaintingInWmPaint, true);
    this.txtValue.TextChanged += txtValue_TextChanged;
}
void txtValue_TextChanged(object sender, EventArgs e)
{
    //校验
    if (this.isValidText)
    {
        this._isValid = true;
        this.isError = false;
    }
    else
    {
        this._isValid = false;
        this.isError = true;
    }
    if(TextChanged!=null)
    {
        //触发该控件的TextChanged事件
        TextChanged(sender, e);
    }
}
```

上述代码中省略了根据枚举类型 TYPE_DATA 进行输入验证的具体实现，有兴趣的读者可以自行实现。我们编译整个解决方案通过的话，然后在 CustomControls 的窗体应用程序新建一个 Form 窗体，我们可以将该控件拖放到界面上，点选后，我们可以在 Visual Studio 属性面板中看到 LabelTextBox 控件的自定义属性（为了便于设置，自定义属性都放在同一个类别 LabelText 下，生产环境下，建议将其按照实际的类别进行归类）如下图 4.4 所示：

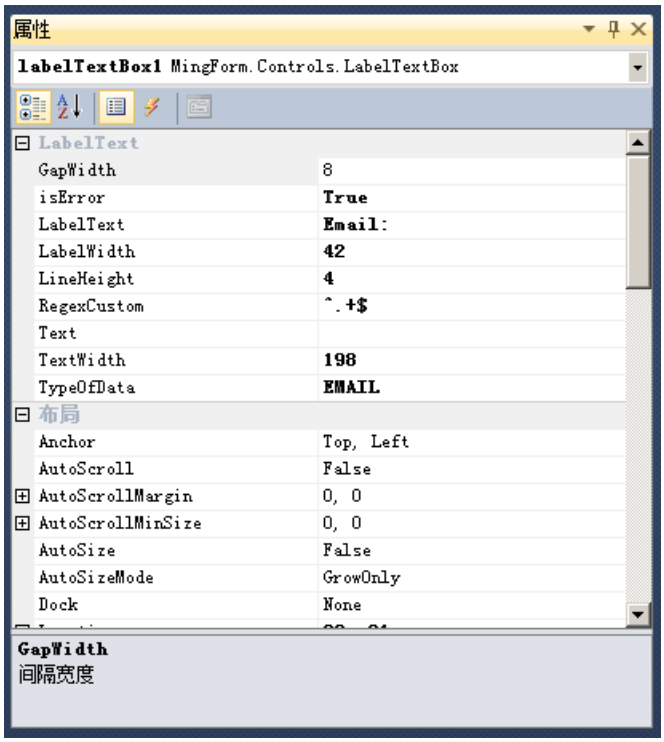


图 4.4 LabelTextBox 属性

在实际生产环境中，自定义的控件往往都有自己的一个个性图标（一般为一个扩展名为.bmp 的图片），下面给出配置控件图标的代码：

```
using System.Drawing;
[ToolboxItem(true)]
[ToolboxBitmap(typeof(LabelTextBox), "LabelTextBox.bmp")]
public partial class LabelTextBox : UserControl
```

其中 ToolboxItem(true)设置该控件在工具栏上可见，ToolboxBitmap 配置控件的图标。我们加上这些属性再重新编译后，看到工具箱中的 LabelTextBox 图标还是默认的图标，并未发生变化，如下图 4.5 所示：

图 4.5 LabelTextBox 控件默认图标

这里必须需要注意的是：由于性能的原因，工具箱的自动填充（控件开发过程中一般都为此模式）区域中的组件不显示自定义位图，并且不支持 ToolboxBitmapAttribute。若要在“工具箱”中显示自定义组件的图标，请使用“选择工具箱项”对话框加载您的组件（一般用于控件正式发布的生产环境中）。当我们手动添加该 MingForm.Controls.dll

到工具箱中（命名为 myCustomControls）后，可以发现控件的自定义图标生效，如下图 4.6 所示：

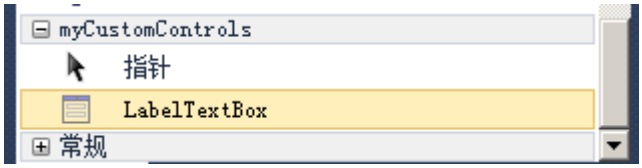


图 4.6 LabelTextBox 控件图标

4.1.4 控件应用

控件编码完成后，我们可以对该控件进行测试和应用，我们在 CustomControls 窗体应用程序中，将自定义的复合控件 LabelTextBox 拖放到 Form1 窗体设计界面上，并配置 LabelText 属性为 Email:，TypeOfData 属性为 EMAIL，如下图 4.7 所示：

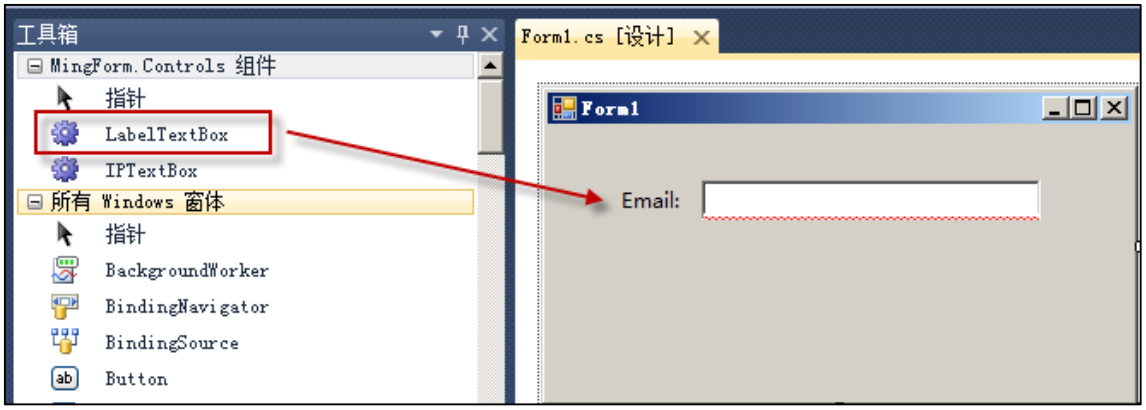


图 4.7 LabelTextBox 控件拖放

Form1 窗体的代码如下所示：

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
    //自定义控件LabelTextBox的TextChanged事件
    private void labelTextBox1_TextChanged(object sender, EventArgs e)
    {
        this.Text = this.labelTextBox1.Text;
    }
    private void Form1_Load(object sender, EventArgs e)
    {
        this.labelTextBox1.Text = "jackwang@gmail.com";
        this.labelTextBox1.TypeOfData =
```

```
                MingForm.Controls.TYPE_DATA.EMAIL;
        this.labelTextBox1.LabelText = "Email:";
        //不起作用，因为LabelWidth和TextWidth会根据labelTextBox1大小动态计算
        //this.labelTextBox1.LabelWidth = 45;
        //this.labelTextBox1.TextWidth = 198;
    }
}
```

运行窗体，结果如图 4.8 所示。

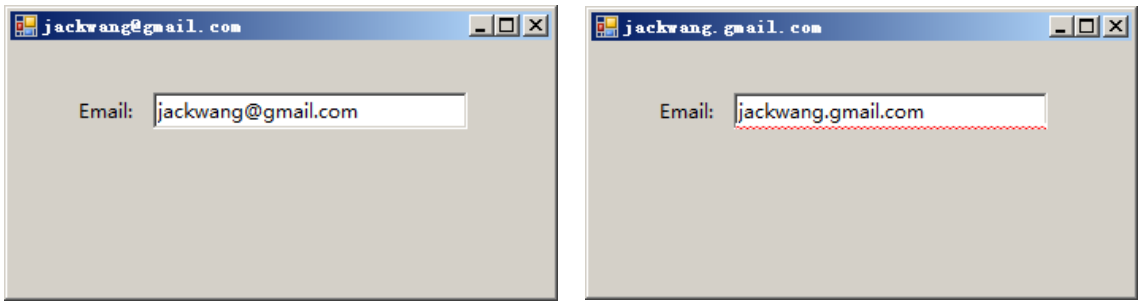


图 4.8 LabelTextBox 控件示例

运行 Form1 窗体，窗体加载后控件 labelTextBox1 的文本为有效的 Email 格式：jackwang@gmail.com，可以看到 labelTextBox1 并未显示错误提示的红色波浪线，如果我们文本修改为无效的 Email,那么 labelTextBox1 会显示红色波浪线。具体如图 4.8 所示。

4.2 IPTextBox 控件

上一节介绍了如何用组合控件的方式开发了一个自定义控件，这一节再介绍一种利用组合控件的方式开发的控件，即 IPTextBox。在 window 上有时候需要配置 IP 地址，IP 地址的输入需要满足 IP 地址的规范，而且中间需要用.进行分隔。如图 4.9 所示。

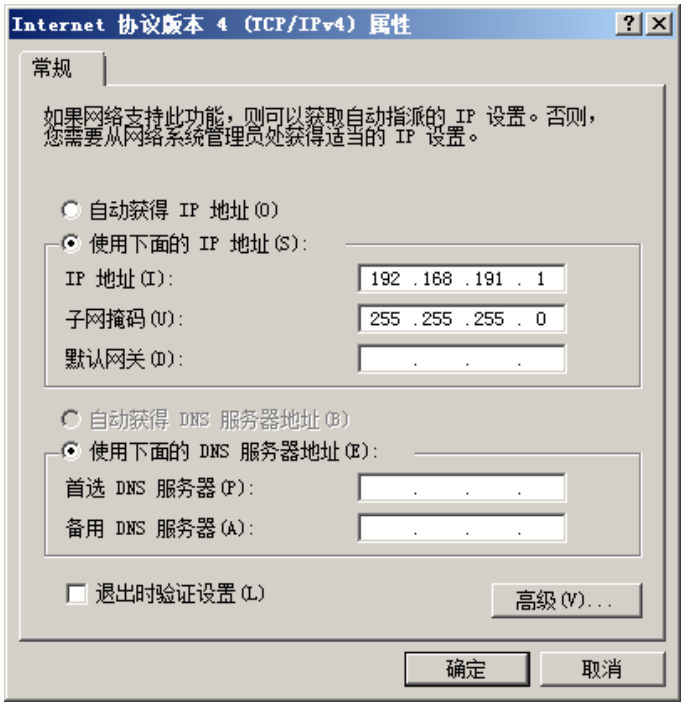


图 4.9 Windows 中 IP 地址控件

4.2.1 控件功能

IPTextBox 控件的核心功能是：只允许输入合法的 IP 地址，且每个 IP 地址段之间用英文点进行分隔。另外就是可以通过 IPText 属性来设置和获取 IP 地址文本，最后自定义一个 isEanbled 属性来控制该控件是否可以编辑。

4.2.2 控件设计

根据 IPTextBox 控件的功能要求，我们可以设计其 UI 如下图 4.10 所示：

图 4.10 IPTextBox UI 设计

根据这个 UI 设计，我们可以创建一个继承自 UserControl 的 IPTextBox 控件，在设计面板上拖放四个 TextBox 和三个 Label 控件。

为了界面整体更加美观，将所有的文本框的 BackColor 都设置为和 panel1 的 BackColor 一致，另外 panel1 的 Dock 属性设置为 Fill，当控件大小调整后，可以自适度高度和宽度。IPTextBox 的文档大纲如图 4.11 所示：

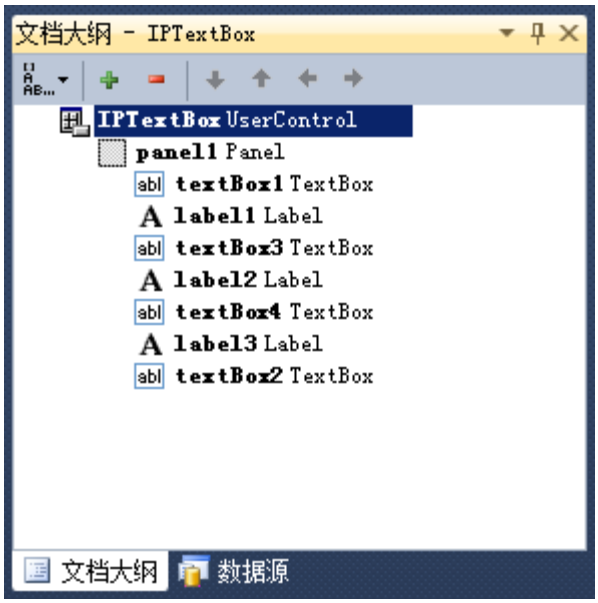


图 4.11 IPTextBox UI 文档大纲

由文档大纲可以清晰看到，所有的文本框和 Label 控件都是放在 Panel1 容器中，可以通过 panel1 的属性来控制该控件一些表现，例如是否可以编辑，如果设置 Panel1 的 Enabled 属性为 false，那么其他控件也是不可编辑的。

另外，必须注意设置四个文本框的 Tab 顺序，从操作简便性上来说，用户很可能通过 Tab 键来进行输入位置切换。

4.2.3 控件开发

上面提到，IPTextBox 控件通过 IPText 属性来设置和获取 IP 地址文本，另外通过 isEanbled 属性来控制该控件是否可以编辑。因此 IPTextBox 控件中必须要创建 IPText 和 isEnabled 这两个属性，核心代码如下：

```
//默认显示在IPText的值
private string mText = "192.168.180.2";
//自定义IPText属性，让该控件可以获取和设置IP地址
[Browsable(true)]
[Description("设置和获取IPTextBox控件的IP地址")]
//DefaultValue此处的默认值，不作用于控件，
//而是当设置的值与DefaultValue不同时，值加粗显示
[DefaultValue("192.168.180.18")]
[Category("自定义")]
public string IPText
{
  get
```

```
  {
     return mText;
  }
set
{
   mText = value;
   if (value != "" && value != null)
   {
      try
      {
         string[] pieces = new string[4];
         pieces = value.ToString().Split(".".ToCharArray(), 4);
         textBox1.Text = pieces[0];
         textBox2.Text = pieces[1];
         textBox3.Text = pieces[2];
         textBox4.Text = pieces[3];
      }
      catch
      {
         //如果输入格式不符合要求，则清空
         textBox1.Text = "";
         textBox2.Text = "";
         textBox3.Text = "";
         textBox4.Text = "";
      }
   }
   else
   {
      textBox1.Text = "";
      textBox2.Text = "";
      textBox3.Text = "";
      textBox4.Text = "";
   }
 }
}
//初始化默认值,控件可用
private bool mEnabled = true;
[Browsable(true)]
[Description("设置和获取IPTextBox控件是否禁用"),
DefaultValue(true), Category("自定义")]
public  bool isEnabled
{
  get
  {
```

```
        return mEnabled;
    }
    set
    {
        mEnabled = value;
        this.panel1.Enabled = mEnabled;
    }
}
```

为了界面美观，在 IPTextBox 控件的构造函数中，通过设置所有四个文本框的 BorderStyle 为 BorderStyle.None 来去掉 textBox 控件默认的边框，同时需要统一背景色，最后为了让该组合控件看起来像一个控件，设置 panel1 的 BorderStyle 属性为 BorderStyle.Fixed3D，核心代码如下：

```
//去掉边框
this.textBox1.BorderStyle = BorderStyle.None;
this.textBox2.BorderStyle = BorderStyle.None;
this.textBox3.BorderStyle = BorderStyle.None;
this.textBox4.BorderStyle = BorderStyle.None;
//统一背景色
this.panel1.BackColor = System.Drawing.SystemColors.Window;
// iptextbox外边框
this.panel1.BorderStyle = BorderStyle.Fixed3D;
```

当控件设置为不可编辑时，会调用 EnabledChanged 事件处理逻辑，将相关背景设置为 SystemColors.Control 的颜色，为了保证 IP 的合法性，在 KeyPress 事件中必须对输入的字符进行验证，只有合法的字符（0 到 255）才能显示在文本框中，否则输入无效，并提示消息，代码如下：

```
private void label_EnabledChanged(object sender, System.EventArgs e)
{
    Label lbl = (Label)sender;
    if (lbl.Enabled)
        lbl.BackColor = SystemColors.Window;
    else
        lbl.BackColor = SystemColors.Control;
}
private void panel1_EnabledChanged(object sender, System.EventArgs e)
{
    Panel pan = (Panel)sender;
    if (pan.Enabled)
        pan.BackColor = SystemColors.Window;
    else
        pan.BackColor = SystemColors.Control;
```

```
}
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{
//只能输入英文 '.', 数值, 或者backspace(e.KeyChar == 8)
if (e.KeyChar.ToString() == "." ||
    Char.IsDigit(e.KeyChar) || e.KeyChar == 8)
{
    //如果输入的是 '.'
    if (e.KeyChar.ToString() == ".")
    {
        //如果输入有效则自动将焦点移调下一个IP段文本框
        if (textBox1.Text != "" &&
             textBox1.Text.Length != textBox1.SelectionLength)
        {
            if (IsValid(textBox1.Text))
                textBox1.Focus();
            else
                textBox1.SelectAll();
        }
        e.Handled = true;
    }
    else if (textBox1.SelectionLength != textBox1.Text.Length)
    {
        if (textBox1.Text.Length == 2)
        {
            if (e.KeyChar == 8)
                textBox1.Text.Remove(textBox1.Text.Length - 1, 1);
            else if (!IsValid(textBox1.Text + e.KeyChar.ToString()))
            {
                textBox1.SelectAll();
                e.Handled = true;
            }
            else
            {
                textBox1.Focus();
            }
        }
    }
}
else
{
    //取消，不进行任何处理
    e.Handled = true;
}
```

```
}
//验证传入的参数是否是有效的IP地址段
private bool IsValid(string inString)
{
    try
    {
        int theValue = int.Parse(inString);
        if (theValue >= 0 && theValue <= 255)
            return true;
        else
        {
            MessageBox.Show("IP地址段必须介于 0~255", "提示");
            return false;
        }
    }
    catch
    {
        return false;
    }
}
//在属性面板中进行隐藏，并且重写后只读，不允许通过该属性进行设置
[Browsable(false), EditorBrowsable(EditorBrowsableState.Never)]
[Description("在属性面板中隐藏掉该属性")]
[Category("自定义")]
public new  bool Enabled
{
    get
    {
        return mEnabled;
    }
}
```

当控件本身具有基本的 Enabled 属性，这里我们定义了一个自定义的 isEnabled 属性，为了防止混淆，这里我们在原有的 Enabled 属性上通过加入 EditorBrowsable 标注来配置该属性在 Visual Studio 属性面板中为不可见(EditorBrowsableState.Never)。IPTextBox 控件在属性面板中显示如图 4.12 所示：

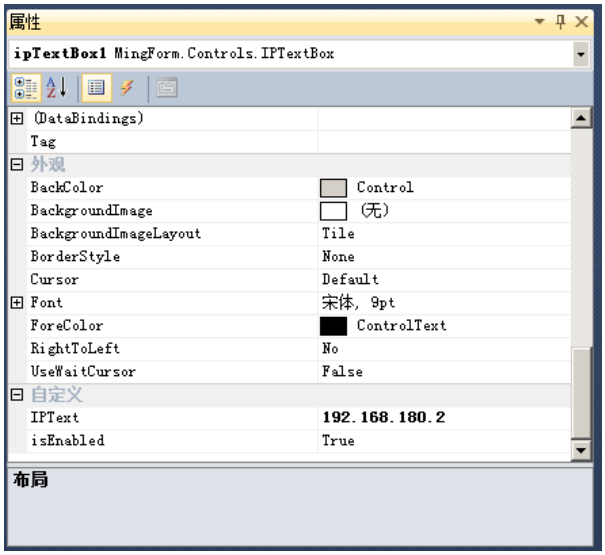


图 4.12 IPTextBox 属性

4.2.4 控件应用

编译后， Visual Studio IDE 可以自动生成一个控件，在测试项目中新创建一个窗体 IPTextBoxTest，从工具箱中将 IPTextBox 控件拖放到窗体上，如图 4.13 所示。

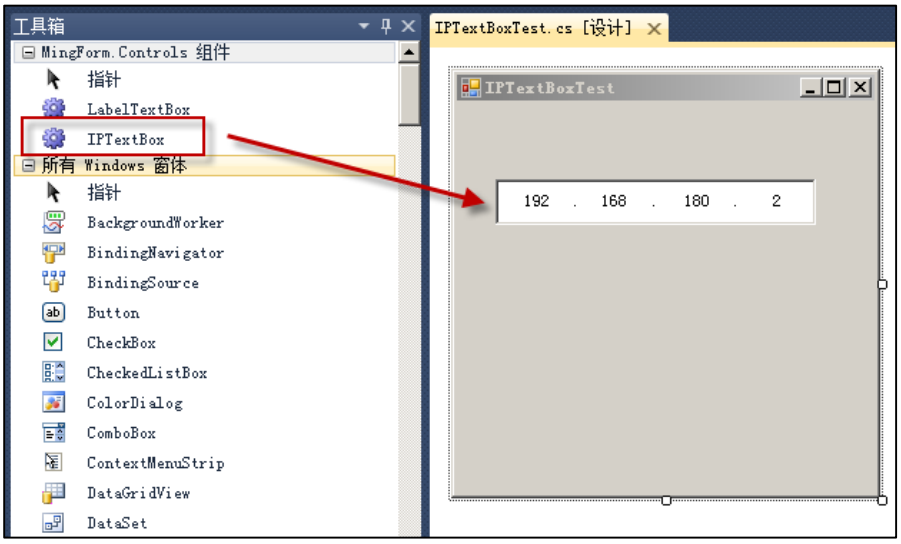


图 4.13 IPTextBox 控件拖放

另外可以用代码的方式进行属性设置，如下所示：

```
public partial class IPTextBoxTest : Form
{
    public IPTextBoxTest()
    {
        InitializeComponent();
    }
```

```
    private void IPTextBoxTest_Load(object sender, EventArgs e)
    {
        this.ipTextBox1.IPText = "192.168.180.180";
        this.ipTextBox1.isEnabled = false;
        string iptext = this.ipTextBox1.IPText;
    }
}
```

运行 IPTextBoxTest 窗体，如图 4.14 所示：

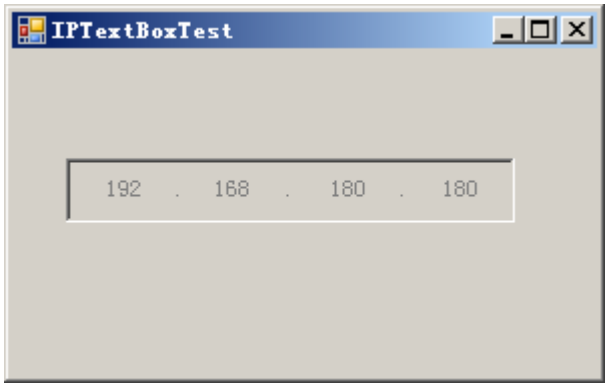


图 4.14 IPTextBox 控件示例

4.3 ChineseMoneyTextBox 控件

前两节介绍了如何用组合控件的方式开发自定义控件，这一节再介绍一种控件的方式开发方式，即扩展控件。扩展控件即继承自现有的基本控件，通过创建属性和方法来扩展原有控件的功能。

一般在财务上，很多金额都需要转化成大写的格式，有些人对于 0 到 9 的大写也不尽然都会写，下面开发一个自定义的 ChineseMoneyTextBox 控件，可以实现将输入的数值转化到大写格式的金额，例如 123456789.99 即为壹亿贰仟叁佰肆拾伍万陆仟柒佰捌拾玖圆玖角玖分。

4.3.1 控件功能

ChineseMoneyTextBox 控件的核心功能是：可以实现将输入的数值转化到大写格式的金额，例如 123456789.99 即为壹亿贰仟叁佰肆拾伍万陆仟柒佰捌拾玖圆玖角玖分。通过获取控件的 Text 属性可以获取到大写格式的信息。

4.3.2 控件设计

根据功能描述，ChineseMoneyTextBox 控件可以继承 TextBox 控件进行扩展即可，UI 如图 4.15 所示：

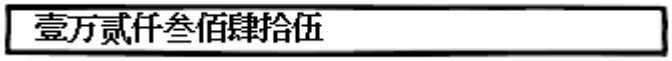


图 4.15 ChineseMoneyTextBox 控件 UI

4.3.3 控件开发

在控件项目中创建一个继承 TextBox 的 ChineseMoneyTextBox 控件，核心代码如下：

```
public partial class ChineseMoneyTextBox : TextBox
{
    public ChineseMoneyTextBox()
    {
        InitializeComponent();
        SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.UserPaint |
ControlStyles.ResizeRedraw | ControlStyles.OptimizedDoubleBuffer, true);
        DoubleBuffered = true;
        Size = new Size(380, 30);
        Location = new Point(10, 38);
        MinimumSize = new Size(200, 20);
        this.TextChanged +=
            new EventHandler(ChineseMoneyTextBox_TextChanged);
    }

    void ChineseMoneyTextBox_TextChanged(object sender, EventArgs e)
    {
        //验证是否为有效值（数值），此处省略
        //重绘
        Invalidate();
    }
    protected override void OnPaint(PaintEventArgs pe)
    {
        base.OnPaint(pe);
        int w = this.Width / 2;
        //计量对应字体下文本的大小
        SizeF sf = pe.Graphics.MeasureString(this.Text, this.Font);
```

```
        pe.Graphics.DrawString(this.Text, this.Font,
             Brushes.Red, 2, this.Height / 2 - sf.Height / 2);
    }
    public override string Text
    {
        get
        {
            try
            {
                double digits = double.Parse(base.Text);
              //静态帮助类，实现小写金额和大写金额的互相转换
                return ToolHelper.Digit2ChineseMoney(digits).ToString();
            }
            catch (Exception ex)
            {
                return "";
            }
        }
        set
        {
            base.Text = value;
        }
    }
    public override string ToString()
    {
        return this.Text;
    }
}
//静态帮助类，实现小写金额和大写金额的互相转换
public static class ToolHelper
{

    public static string Digit2ChineseMoney(double digits)
    {
        //将小写金额转换成大写金额
        string[] MyScale = { "分", "角", "圆", "拾", "佰", "仟", "万", "拾", "佰", "
仟", "亿", "拾", "佰", "仟", "兆", "拾", "佰", "仟" };
        string[] MyBase = { "零", "壹", "贰", "叁", "肆", "伍", "陆", "柒", "捌", "
玖" };
        string moneyStr = "";
        bool isPoint = false;
        string moneyDigital = digits.ToString();
        if (moneyDigital.IndexOf(".") != -1)
```

```
        {
            moneyDigital = moneyDigital.Remove(moneyDigital.IndexOf("."), 1);
            isPoint = true;
        }
        for (int i = moneyDigital.Length; i > 0; i--)
        {
            int MyData = Convert.ToInt16(moneyDigital[moneyDigital.Length -
i].ToString());
            moneyStr += MyBase[MyData];
            if (isPoint == true)
            {
                moneyStr += MyScale[i - 1];
            }
            else
            {
                moneyStr += MyScale[i + 1];
            }
        }
        while (moneyStr.Contains("零零"))
        {
            moneyStr = moneyStr.Replace("零零", "零");
            moneyStr = moneyStr.Replace("零亿", "亿");
            moneyStr = moneyStr.Replace("亿万", "亿");
            moneyStr = moneyStr.Replace("零万", "万");
            moneyStr = moneyStr.Replace("零仟", "零");
            moneyStr = moneyStr.Replace("零佰", "零");
            moneyStr = moneyStr.Replace("零拾", "零");
            while (moneyStr.Contains("零零"))
            {
                moneyStr = moneyStr.Replace("零零", "零");
            }
            moneyStr = moneyStr.Replace("零圆", "圆");
            moneyStr = moneyStr.Replace("零角", "");
            moneyStr = moneyStr.Replace("零分", "");
            moneyStr = moneyStr + "整";
        }
        return moneyStr;
    }
}
```

4.3.4 控件应用

编译后， Visual Studio IDE 可以自动生成一个控件，新创建一个窗体

ChineseMoneyTextBoxTest，从工具箱中将 ChineseMoneyTextBox 控件拖放到该窗体上，如图 4.16 所示。

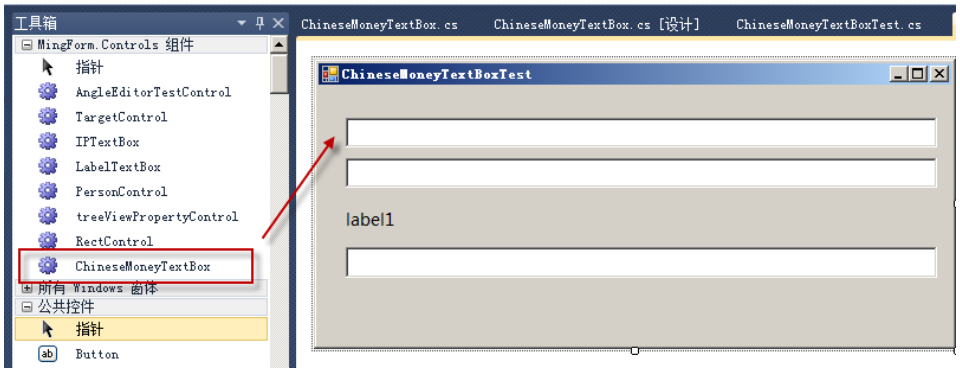


图 4.16 ChineseMoneyTextBox 控件拖放

ChineseMoneyTextBoxTest 的核心代码如下所示：

```
public ChineseMoneyTextBoxTest()
{
    InitializeComponent();
    this.chineseMoneyTextBox1.Text = "123456789.99";
}

private void chineseMoneyTextBox1_TextChanged(object sender, EventArgs e)
{
    this.label1.Text = this.chineseMoneyTextBox1.Text;
    this.textBox1.Text = this.chineseMoneyTextBox1.Text;
}
private void ChineseMoneyTextBoxTest_Load(object sender, EventArgs e)
{
    this.chineseMoneyTextBox1.BackColor = Color.White;
}
```

运行 ChineseMoneyTextBoxTest 窗体，如下图 4.17 所示：

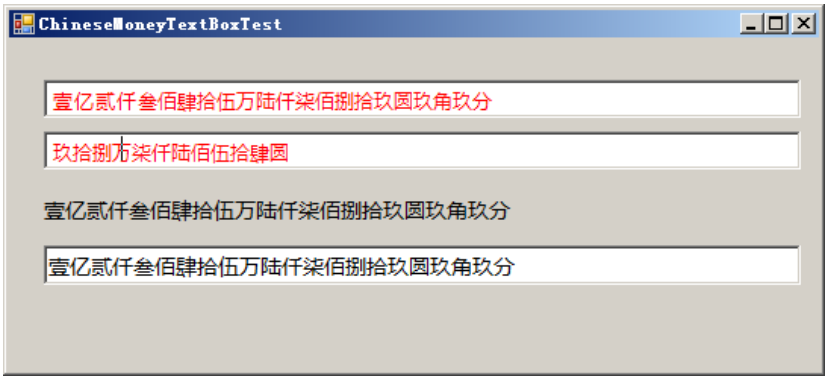


图 4.17 ChineseMoneyTextBox 控件示例

4.4 ImageTextBox 控件

前面提到，控件开发主要有三种方式：组合控件方式；扩展控件方式和自定义控件方式。前面已经分别介绍了组合控件和扩展控件开发的实例。下面介绍自定义控件开发方式。自定义控件直接从 System.Windows.Forms.Control 类派生出来。Control 类提供控件所需要的所有基本功能，包括键盘和鼠标的事件处理。自定义控件是最灵活最强大的方法，必须为 Control 类的 OnPaint 事件写代码。

4.4.1 控件功能

ImageTextBox 控件的核心功能是：在普通 TextBox 控件的基础上，左边能设置字符图标（是矢量图，可以更改大小和颜色），为了实现类似这样的效果，可以用 Wingdings 和 Marlett 等字体。 Wingdings 和 Marlett 是一个符号字体系列，它将许多字母渲染成各式各样的符号。图 4.18 就是 word 中的 Wingdings 符号。

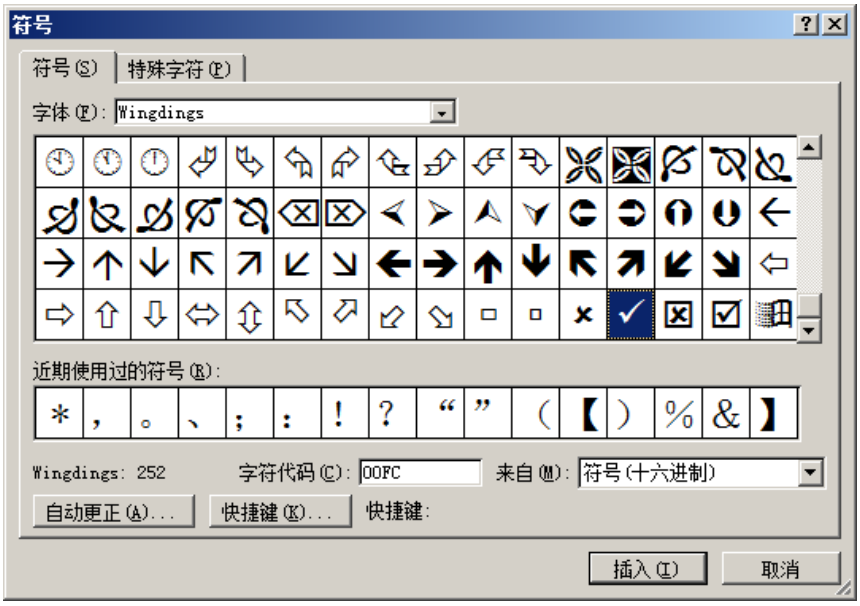


图 4.18 Wingdings 字体符号

利用字体字符可以在设计时或者运行时对显示的字体进行设置。同时可以设置颜色。当然大小也可以进行设置（这里不实现）。

4.4.2 控件设计

根据功能描述，可以对 ImageTextBox 控件进行 UI 设计，整个控件由两部分组成，左

边为图标显示区域，右边为文本编辑区域。UI 如图 4.19 所示：

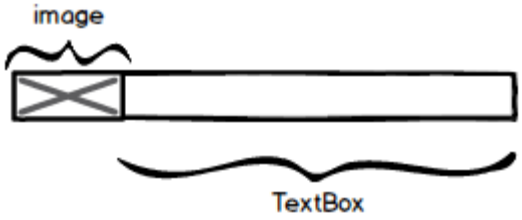


图 4.19 ImageTextBox 控件 UI

4.4.3 控件开发

首先在 Visual Studio IDE 中创建一个名为 ImageTextBox 的组件，它继承自 System.Windows.Forms.Control，创建方法是在项目上右击，然后在右键菜单中选择【组件】进行创建，如下图 4.20 所示。

图 4.20 Visual Studio 添加组件

新创建的组件没有可见的 UI，可以看到 Visual Studio 在设计视图时显示如图 4.21 所示的信息：

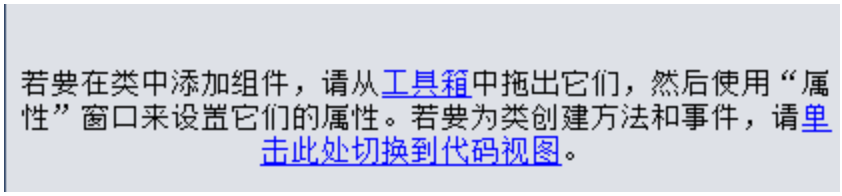


图 4.21 Visual Studio 组件设计视图信息

在该控件即后续的自定义控件开发中，很多都需要和鼠标进行交互，绘制控件时，需要知道当前鼠标的状态，因此定义一个公共的 MouseState 枚举，后续的控件可能不给出该枚举代码。

```
public enum MouseState : byte
{
```

```
        None = 0,
        Over = 1,
        Down = 2,
        Block = 3
    }
```

ImageTextBox 控件的代码如下：

```
using System;
using System.ComponentModel;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Text;
using System.Windows.Forms;
namespace MingForm.Controls
{
    [DefaultEvent("TextChanged")]
    public class ImageTextBox : Control
    {
       public static Color BGColor = Color.FromArgb(35, 168, 109);
        private int W;
        private int H;
        private MouseState State = MouseState.None;
        private System.Windows.Forms.TextBox TB;
        private HorizontalAlignment _TextAlign = HorizontalAlignment.Left;
       [Category("ImageTextBox")]
        public HorizontalAlignment TextAlign
        {
            get { return _TextAlign; }
            set
            {
                _TextAlign = value;
                if (TB != null)
                {
                    TB.TextAlign = value;
                }
            }
        }
        private int _MaxLength = 32767;
       [Category("ImageTextBox")]
        public int MaxLength
        {
            get { return _MaxLength; }
            set
            {
                _MaxLength = value;
```

```
            if (TB != null)
            {
                TB.MaxLength = value;
            }
        }
    }
    private bool _ReadOnly;
[Category("ImageTextBox")]
    public bool ReadOnly
    {
        get { return _ReadOnly; }
        set
        {
            _ReadOnly = value;
            if (TB != null)
            {
                TB.ReadOnly = value;
            }
        }
    }
    private bool _UseSystemPasswordChar;
[Category("ImageTextBox")]
    public bool UseSystemPasswordChar
    {
        get { return _UseSystemPasswordChar; }
        set
        {
            _UseSystemPasswordChar = value;
            if (TB != null)
            {
                TB.UseSystemPasswordChar = value;
            }
        }
    }
    private bool _Multiline;
[Category("ImageTextBox")]
    public bool Multiline
    {
        get { return _Multiline; }
        set
        {
            _Multiline = value;
            if (TB != null)
            {
```

```
                TB.Multiline = value;
                if (value)
                {
                    TB.Height = Height - 11;
                }
                else
                {
                    Height = TB.Height + 11;
                }
            }
        }
    }
    private bool _FocusOnHover = false;
    [Category("ImageTextBox")]
    public bool FocusOnHover
    {
        get { return _FocusOnHover; }
        set { _FocusOnHover = value; }
    }
    [Category("ImageTextBox")]
    public override string Text
    {
        get { return base.Text; }
        set
        {
            base.Text = value;
            if (TB != null)
            {
                TB.Text = value;
            }
        }
    }
    [Category("ImageTextBox")]
    public override Font Font
    {
        get { return base.Font; }
        set
        {
            base.Font = value;
            if (TB != null)
            {
                TB.Font = value;
                TB.Location = new Point(3, 5);
                TB.Width = Width - 6;
```

```
                if (!_Multiline)
                {
                    Height = TB.Height + 11;
                }
            }
        }
    }
    //添加文本框
    protected override void OnCreateControl()
    {
        base.OnCreateControl();
        if (!Controls.Contains(TB))
        {
            Controls.Add(TB);
        }
    }
    private void OnBaseTextChanged(object s, EventArgs e)
    {
        Text = TB.Text;
    }
    private void OnBaseKeyDown(object s, KeyEventArgs e)
    {
        if (e.Control && e.KeyCode == Keys.A)
        {
            TB.SelectAll();
            e.SuppressKeyPress = true;
        }
        if (e.Control && e.KeyCode == Keys.C)
        {
            TB.Copy();
            e.SuppressKeyPress = true;
        }
    }
    protected override void OnResize(EventArgs e)
    {
       //位置和大小重新布局
        TB.Location = new Point(45, 5);
        TB.Width = Width - 50;
        if (_Multiline)
        {
            TB.Height = Height - 11;
        }
        else
        {
```

```
            Height = TB.Height + 11;
        }
        base.OnResize(e);
    }
    [Category("ImageTextBox")]
    public Color TextColor
    {
        get { return _TextColor; }
        set { _TextColor = value; }
    }
    public override Color ForeColor
    {
        get { return _TextColor; }
        set { _TextColor = value; }
    }
    private string iconText = "ü";
    [Category("ImageTextBox"), Browsable(true)]
    [Description("icon font text")]
    public string IconText
    {
       get { return iconText; }
       set { iconText = value; Invalidate(); }
    }
    protected override void OnMouseDown(MouseEventArgs e)
    {
        base.OnMouseDown(e);
        State = MouseState.Down;
        Invalidate();
    }
    protected override void OnMouseUp(MouseEventArgs e)
    {
        base.OnMouseUp(e);
        State = MouseState.Over;
        TB.Focus();
        Invalidate();
    }
    protected override void OnMouseEnter(EventArgs e)
    {
        base.OnMouseEnter(e);
        State = MouseState.Over;
        if(FocusOnHover) TB.Focus();
        Invalidate();
    }
    protected override void OnMouseLeave(EventArgs e)
```

```
        {
            base.OnMouseLeave(e);
            State = MouseState.None;
            Invalidate();
        }
        private Color _BaseColor = Color.FromArgb(45, 47, 49);
        private Color _TextColor = Color.FromArgb(192, 192, 192);
        private Color _BorderColor = BGColor;
       public ImageTextBox()
        {
            SetStyle(ControlStyles.AllPaintingInWmPaint |
ControlStyles.UserPaint | ControlStyles.ResizeRedraw |
ControlStyles.OptimizedDoubleBuffer |
ControlStyles.SupportsTransparentBackColor, true);
            DoubleBuffered = true;
            BackColor = Color.Transparent;
            TB = new System.Windows.Forms.TextBox();
            TB.Font = new Font("Segoe UI", 10);
            TB.Text = Text;
            TB.BackColor = _BaseColor;
            TB.ForeColor = _TextColor;
            TB.MaxLength = _MaxLength;
            TB.Multiline = _Multiline;
            TB.ReadOnly = _ReadOnly;
            TB.UseSystemPasswordChar = _UseSystemPasswordChar;
            TB.BorderStyle = BorderStyle.None;
            TB.Location = new Point(45, 5);
            TB.Width = Width - 50;
            TB.Cursor = Cursors.IBeam;
            if (_Multiline)
            {
                TB.Height = Height - 11;
            }
            else
            {
                Height = TB.Height + 11;
            }
            TB.TextChanged += OnBaseTextChanged;
            TB.KeyDown += OnBaseKeyDown;
        }
        protected override void OnPaint(PaintEventArgs e)
        {
          _BorderColor = BGColor;
            //创建一个和控件同大小的位图
```

```
  Bitmap B = new Bitmap(Width, Height);
  //从图形创建Graphics
  Graphics G = Graphics.FromImage(B);
  //GDI+绘图时，绘制路径会在右边和下面扩展出1像素的区域，要减去
  W = Width - 1;
  H = Height - 1;
  //创建和控件等大小的矩形
  Rectangle Base = new Rectangle(0, 0, W, H);
  //设置高质量的绘图和文本显示方式
  var _withG = G;
  _withG.SmoothingMode = SmoothingMode.HighQuality;
  _withG.PixelOffsetMode = PixelOffsetMode.HighQuality;
  _withG.TextRenderingHint = TextRenderingHint.ClearTypeGridFit;
  //以BackColor填充控件绘图区域
  _withG.Clear(BackColor);
  //以Rectangle 创建宽度为40和控件等高的按钮，存放图标
Rectangle Button = new Rectangle(0, 0, 40, H);
GraphicsPath GP = new GraphicsPath();
GraphicsPath GP2 = new GraphicsPath();

  //将TB控件的背景色和文本颜色和控件统一
  TB.BackColor = _BaseColor;
  TB.ForeColor = _TextColor;
  // 用BaseColor填充控件区域
  _withG.FillRectangle(new SolidBrush(_BaseColor), Base);
//重设区域剪辑
GP.Reset();
GP.AddRectangle(Button);
//将定义的图标按钮区域作为剪辑区域
_withG.SetClip(GP);
//填充图标区域
_withG.FillRectangle(new SolidBrush(BGColor), Button);
_withG.ResetClip();
//绘制图标
_withG.DrawString(IconText, new Font("Wingdings", 22),
      new SolidBrush(_BaseColor),
      new Rectangle(0, 0, 40, H), new StringFormat
{
    Alignment = StringAlignment.Center,
    LineAlignment = StringAlignment.Near
});
  base.OnPaint(e);
//因为类是引用类型，释放资源G,同时释放_withG
  G.Dispose();
```

```
            e.Graphics.InterpolationMode = InterpolationMode.HighQualityBicubic;
            //将内存中的绘图绘制到UI界面上，注意用的e.Graphics而不是_withG
            e.Graphics.DrawImageUnscaled(B, 0, 0);
            B.Dispose();
        }
    }
}
```

4.4.4 控件应用

编译后，Visual Studio IDE 可以自动生成一个控件，新创建一个窗体 ImageTextBoxTest，从工具箱中将 ImageTextBox 控件拖放到界面上，并配置相关属性，如图 4.22 所示：

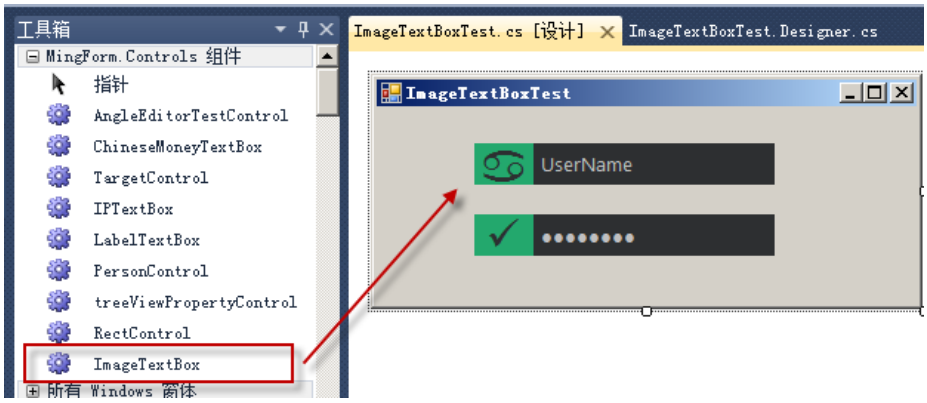


图 4.22 ImageTextBox 控件拖放

我们选择该控件后，可以配置 IconText 属性，此文本为 Wingdings 字符字体对应的字符编码，通过设置不同的 IconText，可以显示不同的图标，如下图 4.23 所示：

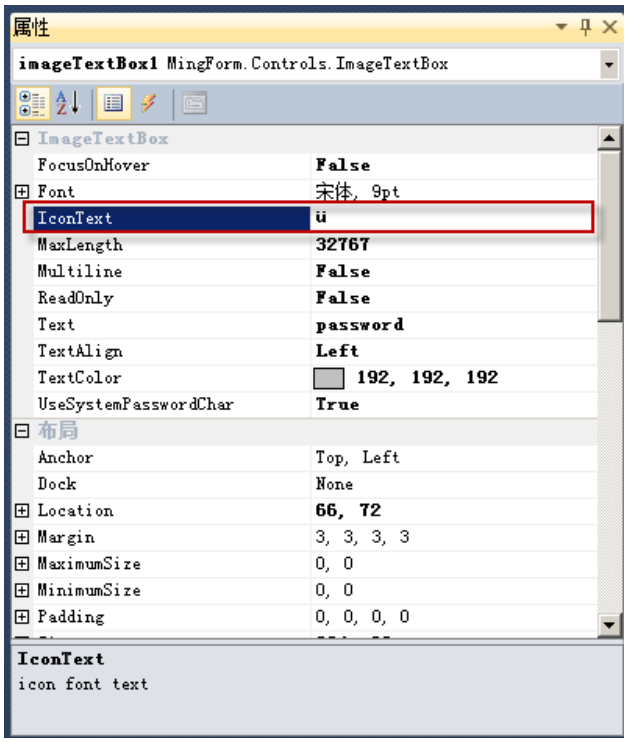


图 4.23 ImageTextBox 控件属性

ImageTextBoxTest 窗体代码如下：

```
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace CustomControls
{
    public partial class ImageTextBoxTest : Form
    {
        public ImageTextBoxTest()
        {
            InitializeComponent();
            this.imageTextBox1.Text = "UserName";
            this.imageTextBox1.IconText = "a";
            //密码框
            this.imageTextBox2.IconText = "ü";
            this.imageTextBox2.Text = "password";
            this.imageTextBox2.UseSystemPasswordChar = true;

        }
    }
}
```

运行 ImageTextBoxTest 窗体，界面如图 4.24 所示：

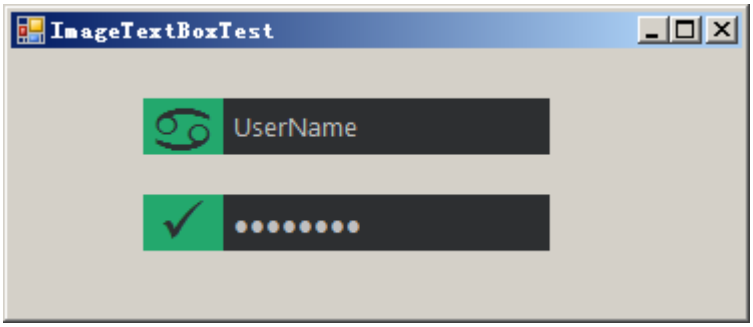


图 4.24 ImageTextBox 控件示例

4.5 ImageButton 控件

上一节介绍了开发自定义的 ImageTextBox 控件，可以在文本的左边显示一个字符图标，其实在上述控件代码的基础上，稍作修改，即可实现另一个自定义控件让它左边依然显示一个字符图标，但是右边是一个类似按钮的控件，可以响应单击事件。

4.5.1 控件功能

ImageButton 控件的核心功能是： 在普通的 Button 基础上，让左边能添加一个文字图标，右边显示文本内容，单击可以出发单击事件。

4.5.2 控件设计

根据功能描述，我们可以将 ImageButton 控件的 UI 设计如下图 4.25 所示：

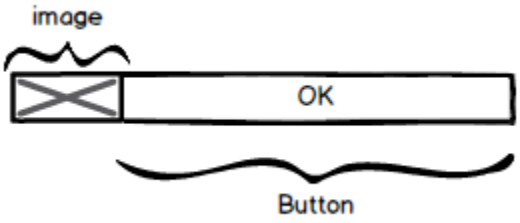


图 4.25 ImageButton 控件示例

4.5.3 控件开发

首先在 Visual Studio IDE 中创建一个名为 ImageButton 的组件，它继承自 System.Windows.Forms.Control，代码如下。

```
using System;
using System.ComponentModel;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Text;
using System.Windows.Forms;
namespace MingForm.Controls
{
    [DefaultEvent("Click")]
    public class ImageButton : Control
    {
       public static Color BGColor = Color.FromArgb(35, 168, 109);
        private int W;
        private int H;
        private MouseState State = MouseState.None;
        private System.Windows.Forms.Button BT;
       [Category("ImageButton")]
        public override string Text
        {
            get { return base.Text; }
            set
            {
                base.Text = value;
                if (BT != null)
                {
                    BT.Text = value;
                }
            }
        }
       [Category("ImageButton")]
        public override Font Font
        {
            get { return base.Font; }
            set
            {
                base.Font = value;
                if (BT != null)
                {
                    BT.Font = value;
                    BT.Location = new Point(0, 0);
                    BT.Width = Width - 6;
                }
            }
        }
```

```
 protected override void OnCreateControl()
 {
     base.OnCreateControl();
     if (!Controls.Contains(BT))
     {
         Controls.Add(BT);
     }
 }
 private void OnBaseTextChanged(object s, EventArgs e)
 {
     Text = BT.Text;
 }
 protected override void OnResize(EventArgs e)
 {
     BT.Location = new Point(40, -1);
     BT.Width = Width - 40;
   BT.Height = Height;
     base.OnResize(e);
 }
[Category("ImageButton")]
 public Color TextColor
 {
     get { return _TextColor; }
     set { _TextColor = value; }
 }
 public override Color ForeColor
 {
     get { return _TextColor; }
     set { _TextColor = value; }
 }
private string iconText = "ü";
[Category("ImageButton"), Browsable(true)]
[Description("icon font text")]
public string IconText
{
   get { return iconText; }
   set { iconText = value; Invalidate(); }
}
 protected override void OnMouseDown(MouseEventArgs e)
 {
     base.OnMouseDown(e);
     State = MouseState.Down;
     Invalidate();
 }
```

```
        protected override void OnMouseUp(MouseEventArgs e)
        {
            base.OnMouseUp(e);
            State = MouseState.Over;
            BT.Focus();
            Invalidate();
        }
        protected override void OnMouseEnter(EventArgs e)
        {
            base.OnMouseEnter(e);
            State = MouseState.Over;
            Invalidate();
        }
        protected override void OnMouseLeave(EventArgs e)
        {
            base.OnMouseLeave(e);
            State = MouseState.None;
            Invalidate();
        }
        private Color _BaseColor = Color.FromArgb(45, 47, 49);
        private Color _TextColor = Color.FromArgb(192, 192, 192);
        private Color _BorderColor = BGColor;
      public ImageButton()
        {
            SetStyle(ControlStyles.AllPaintingInWmPaint |
ControlStyles.UserPaint | ControlStyles.ResizeRedraw |
ControlStyles.OptimizedDoubleBuffer |
ControlStyles.SupportsTransparentBackColor, true);
            DoubleBuffered = true;
            BackColor = Color.Transparent;
            BT = new System.Windows.Forms.Button();
            BT.Font = new Font("Segoe UI", 10);
            BT.Text = Text;
            BT.BackColor = _BaseColor;
            BT.ForeColor = _TextColor;
          //隐藏掉边框
            BT.FlatStyle =FlatStyle.Flat;
          BT.FlatAppearance.BorderSize = 0;
            BT.Location = new Point(45, 5);
            BT.Width = Width - 50;
          this.Cursor = Cursors.Hand;
          BT.Height = Height;
            BT.TextChanged += OnBaseTextChanged;
          BT.Click += new EventHandler(BT_Click);
```

```
 }
void BT_Click(object sender, EventArgs e)
{
    //以BT控件的单击事件触发本控件的单击事件
    this.InvokeOnClick(this, e);
}
 protected override void OnPaint(PaintEventArgs e)
 {
    _BorderColor = BGColor;
      Bitmap B = new Bitmap(Width, Height);
      Graphics G = Graphics.FromImage(B);
      W = Width - 1;
      H = Height - 1;
      Rectangle Base = new Rectangle(0, 0, W, H);
      var _withG = G;
      _withG.SmoothingMode = SmoothingMode.HighQuality;
      _withG.PixelOffsetMode = PixelOffsetMode.HighQuality;
      _withG.TextRenderingHint = TextRenderingHint.ClearTypeGridFit;
      _withG.Clear(BackColor);
    Rectangle Button = new Rectangle(0, 0, 40, H);
    GraphicsPath GP = new GraphicsPath();
    GraphicsPath GP2 = new GraphicsPath();
    BT.BackColor = _BaseColor;
    BT.ForeColor = _TextColor;
      _withG.FillRectangle(new SolidBrush(_BaseColor), Base);
    GP.Reset();
    GP.AddRectangle(Button);
    _withG.SetClip(GP);
    _withG.FillRectangle(new SolidBrush(BGColor), Button);
    _withG.ResetClip();
    _withG.DrawString(IconText, new Font("Wingdings", 22),
      new SolidBrush(_BaseColor),
      new Rectangle(0,5, 40, H), new StringFormat
    {
       Alignment = StringAlignment.Center,
       LineAlignment = StringAlignment.Center
    });
      base.OnPaint(e);
      G.Dispose();
      e.Graphics.InterpolationMode = InterpolationMode.HighQualityBicubic;
      e.Graphics.DrawImageUnscaled(B, 0, 0);
      B.Dispose();
 }
```

```
    }
}
```

4.5.4 控件应用

编译后，Visual Studio IDE 可以自动生成一个控件，新创建一个窗体 ImageButtonTest，从工具箱中将 ImageButton 控件拖放到设计界面上，如图 4.26 所示：

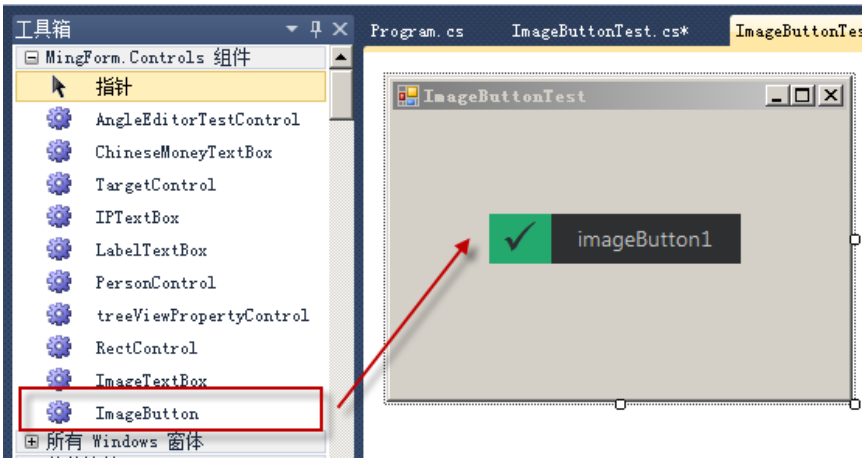


图 4.26 ImageButton 控件拖放

ImageButtonTest 窗体的代码如下所示：

```
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace CustomControls
{
    public partial class ImageButtonTest : Form
    {
        public ImageButtonTest()
        {
            InitializeComponent();
        }

        private void ImageButtonTest_Load(object sender, EventArgs e)
        {
            this.imageButton1.IconText = "ü";
            this.imageButton1.Text = "确 定";
        }

        private void imageButton1_Click(object sender, EventArgs e)
```

```
        {
            this.Text = this.imageButton1.IconText;
        }
    }
}
```

运行 ImageButtonTest 窗体，界面如图 4.27 所示：

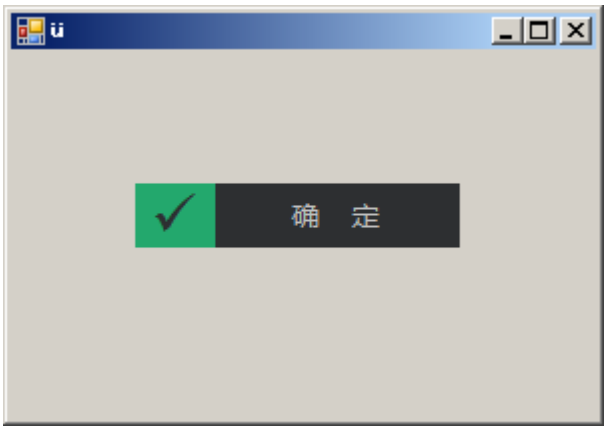


图 4.27 ImageButton 控件示例

4.6 IconCheckBox 控件

上一节介绍了 ImageButton 控件的开发，这一节我们定义一个带图标的 IconCheckBox 控件，该控件继承自 Control，我们还是和前面一样，采取自绘制的方式来定制控件，这样可以最灵活的控制控件的行为和样式。

Windows 自带的 CheckBox 功能虽然基本够用，但是 UI 不美观，下面我们就用自己开发的 IconCheckBox 控件来取代原生的 CheckBox 控件。

4.6.1 控件功能

IconCheckBox 控件的核心功能是：和 CheckBox 控件类似，可以通过属性设置两个状态，然后将 CheckBox 控件样式扁平化，前面的 Check 图标可以自定义（此处定义为对勾图标）。

4.6.2 控件设计

根据控件的功能要求，我们可以将该控件的 UI 设计如图 4.28 所示：

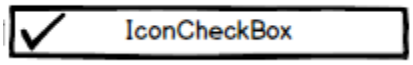


图 4.28 IconCheckBox 控件 UI

4.6.3 控件开发

首先在 Visual Studio IDE 中创建一个名为 IconCheckBox 的组件，它继承自 System.Windows.Forms.Control，代码如下。

```
using System;
using System.ComponentModel;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Text;
using System.Windows.Forms;
namespace MingForm.Controls
{
    //默认属性和事件
    [DefaultProperty("Text")]
     [DefaultEvent("CheckedChanged")]
     public class IconCheckBox : Control
     {
         private int W;
         private int H;
         private MouseState State = MouseState.None;

         private bool _Checked;

         protected override void OnTextChanged(System.EventArgs e)
         {
             base.OnTextChanged(e);
             Invalidate();
         }

         public bool Checked
         {
             get { return _Checked; }
             set
             {
                 _Checked = value;
                 Invalidate();
             }
```

```
    }

    public event CheckedChangedEventHandler CheckedChanged;
    public delegate void CheckedChangedEventHandler(object sender);
    protected override void OnClick(System.EventArgs e)
    {
        _Checked = !_Checked;
        if (CheckedChanged != null)
        {
            CheckedChanged(this);
        }
        base.OnClick(e);
    }
    protected override void OnResize(EventArgs e)
    {
        base.OnResize(e);
        Height = 30;
    }

[Category("IconCheckBox")]
    public Color BaseColor
    {
        get { return _BaseColor; }
        set { _BaseColor = value; }
    }

[Category("IconCheckBox")]
    public Color BorderColor
    {
        get { return _BorderColor; }
        set { _BorderColor = value; }
    }

    protected override void OnMouseDown(MouseEventArgs e)
    {
        base.OnMouseDown(e);
        State = MouseState.Down;
        Invalidate();
    }

    protected override void OnMouseUp(MouseEventArgs e)
    {
        base.OnMouseUp(e);
        State = MouseState.Over;
```

```
            Invalidate();
        }

        protected override void OnMouseEnter(EventArgs e)
        {
            base.OnMouseEnter(e);
            State = MouseState.Over;
            Invalidate();
        }

        protected override void OnMouseLeave(EventArgs e)
        {
            base.OnMouseLeave(e);
            State = MouseState.None;
            Invalidate();
        }

        private Color _BaseColor = Color.FromArgb(45, 47, 49);
        private Color _TextColor = Color.FromArgb(243, 243, 243);
        private Color _BorderColor  = Color.FromArgb(35, 168, 109);

      public IconCheckBox()
        {
            SetStyle(ControlStyles.AllPaintingInWmPaint |
ControlStyles.UserPaint | ControlStyles.ResizeRedraw |
ControlStyles.OptimizedDoubleBuffer, true);
            DoubleBuffered = true;
            BackColor = Color.FromArgb(60, 70, 73);
            Cursor = Cursors.Hand;
            Font = new Font("Segoe UI", 10);
            Size = new Size(200, 30);
        }

      protected override void OnPaint(PaintEventArgs e)
      {

          Bitmap B = new Bitmap(Width, Height);
          Graphics G = Graphics.FromImage(B);
          W = Width - 1;
          H = Height - 1;

          Rectangle Base = new Rectangle(1, 0, Height-1, Height-2);

          var _withG = G;
```

```
            _withG.SmoothingMode = SmoothingMode.HighQuality;
            _withG.TextRenderingHint = TextRenderingHint.ClearTypeGridFit;
            _withG.Clear(BackColor);

            _withG.FillRectangle(new SolidBrush(_BaseColor), Base);

            switch (State)
            {
                case MouseState.Over:
                    _withG.DrawRectangle(new Pen(_BorderColor), Base);
                    break;
                case MouseState.Down:
                    _withG.DrawRectangle(new Pen(_BorderColor), Base);
                    break;
            }
            if (Checked)
            {
                //对勾符号
                _withG.DrawString("ü", new Font("Wingdings", 18), new
SolidBrush(_BorderColor), new Rectangle(5, 7, H - 9, H - 9), new StringFormat
                {
                    Alignment = StringAlignment.Center,
                    LineAlignment = StringAlignment.Center
                });
            }
            if (this.Enabled == false)
            {
                _withG.FillRectangle(new SolidBrush(Color.FromArgb(54, 58, 61)),
Base);
                _withG.DrawString(Text, Font, new SolidBrush(Color.FromArgb(140,
142, 143)), new Rectangle(20, 2, W, H), new StringFormat
                {
                    Alignment = StringAlignment.Center,
                    LineAlignment = StringAlignment.Center
                });
            }
            _withG.DrawString(Text, Font, new SolidBrush(_TextColor), new
Rectangle(20, 2, W, H), new StringFormat
            {
                Alignment = StringAlignment.Center,
                LineAlignment = StringAlignment.Center
            });
            base.OnPaint(e);
            G.Dispose();
```

```
            e.Graphics.InterpolationMode = InterpolationMode.HighQualityBicubic;
            e.Graphics.DrawImageUnscaled(B, 0, 0);
            B.Dispose();
        }
    }
}
```

4.6.4 控件应用

编译后，Visual Studio IDE 可以自动生成一个控件，新创建一个窗体 IconCheckBoxTest，从工具箱中将 IconCheckBox 控件拖放到窗体上，并配置属性，如下图 4.29 所示：

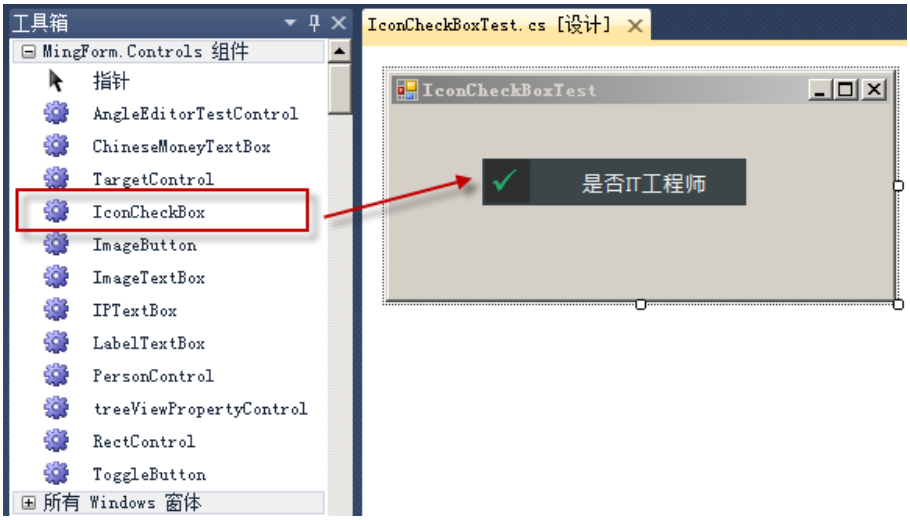


图 4.29 IconCheckBox 控件示例

IconCheckBoxTest 的代码如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace CustomControls
{
    public partial class IconCheckBoxTest : Form
    {
        public IconCheckBoxTest()
        {
```

```
            InitializeComponent();
        }
        private void IconCheckBoxTest_Load(object sender, EventArgs e)
        {
            this.iconCheckBox1.Checked = true;
            this.iconCheckBox1.Text = "是否IT工程师";

        }
    }
}
```

4.7 ToggleButton 控件

上一节介绍了 IconCheckBox 控件，可以表示对立的两个状态，这一节我们再定义一个 ToggleButton 控件对立的两个状态，它们的区别主要在 UI 样式上，ToggleButton 是苹果设备上的控件，样式比较美观，下面我们就开始开发自己的 ToggleButton。

4.7.1 控件功能

ToggleButton 控件的核心功能是：可以表示对立的两个状态，通过单击图标进行状体切换，界面类似于苹果设备的 ToggleButton 样式(这里需要圆角的处理)，同时一个功能点是可以支持多个 UI 样式。

4.7.2 控件设计

根据功能需求，ToggleButton 控件的 UI 设计如下图 4.30 所示：

图 4.30 ToggleButton 控件 UI

4.7.3 控件开发

根据 ToggleButto 的功能需求和 UI 设计，需要圆角边框的处理，这里我们定义一个 GDIHelpers 静态类，来处理圆角绘制和文本格式等业务逻辑。

```
using System;
using System.Drawing;
```

```
using System.Drawing.Drawing2D;
using System.Windows.Forms;
namespace MingForm.Controls
{
    public static class GDIHelpers
    {
        public static Color DefaultColor = Color.FromArgb(35, 168, 109);
        public static readonly StringFormat NearSF = new StringFormat
        {
            Alignment = StringAlignment.Near,
            LineAlignment = StringAlignment.Near
        };
        public static readonly StringFormat CenterSF = new StringFormat
        {
            Alignment = StringAlignment.Center,
            LineAlignment = StringAlignment.Center
        };
        public static GraphicsPath RoundRec(Rectangle Rectangle, int Curve)
        {
            GraphicsPath P = new GraphicsPath();
            int ArcRectangleWidth = Curve * 2;
            P.AddArc(new Rectangle(Rectangle.X, Rectangle.Y, ArcRectangleWidth,
ArcRectangleWidth), -180, 90);
            P.AddArc(new Rectangle(Rectangle.Width - ArcRectangleWidth +
Rectangle.X, Rectangle.Y, ArcRectangleWidth, ArcRectangleWidth), -90, 90);
            P.AddArc(new Rectangle(Rectangle.Width - ArcRectangleWidth +
Rectangle.X, Rectangle.Height - ArcRectangleWidth + Rectangle.Y, ArcRectangleWidth,
ArcRectangleWidth), 0, 90);
            P.AddArc(new Rectangle(Rectangle.X, Rectangle.Height -
ArcRectangleWidth + Rectangle.Y, ArcRectangleWidth, ArcRectangleWidth), 90, 90);
            P.AddLine(new Point(Rectangle.X, Rectangle.Height - ArcRectangleWidth
+ Rectangle.Y), new Point(Rectangle.X, Curve + Rectangle.Y));
            return P;
        }
        public static GraphicsPath RoundRect(float x, float y, float w, float h,
double r = 0.3,
            bool TL = true, bool TR = true, bool BR = true, bool BL = true)
        {
            GraphicsPath functionReturnValue = null;
            float d = Math.Min(w, h) * (float)r;
            float xw = x + w;
            float yh = y + h;
            functionReturnValue = new GraphicsPath();
            var _withGP = functionReturnValue;
```

```
            if (TL)
                _withGP.AddArc(x, y, d, d, 180, 90);
            else
                _withGP.AddLine(x, y, x, y);
            if (TR)
                _withGP.AddArc(xw - d, y, d, d, 270, 90);
            else
                _withGP.AddLine(xw, y, xw, y);
            if (BR)
                _withGP.AddArc(xw - d, yh - d, d, d, 0, 90);
            else
                _withGP.AddLine(xw, yh, xw, yh);
            if (BL)
                _withGP.AddArc(x, yh - d, d, d, 90, 90);
            else
                _withGP.AddLine(x, yh, x, yh);
            _with1.CloseFigure();
            return functionReturnValue;
        }
        public static GraphicsPath DrawArrow(int x, int y, bool flip)
        {
            GraphicsPath GP = new GraphicsPath();
            int W = 12;
            int H = 6;
            if (flip)
            {
                GP.AddLine(x + 1, y, x + W + 1, y);
                GP.AddLine(x + W, y, x + H, y + H - 1);
            }
            else
            {
                GP.AddLine(x, y + H, x + W, y + H);
                GP.AddLine(x + W, y + H, x + H, y);
            }
            GP.CloseFigure();
            return GP;
        }
    }
}
```

首先在 Visual Studio IDE 中创建一个名为 ToggleButton 的组件，它继承自 System.Windows.Forms.Control，代码如下。

```
using System;
using System.ComponentModel;
```

```
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Text;
using System.Windows.Forms;
namespace MingForm.Controls
{
    [DefaultProperty("Text")]
     [DefaultEvent("CheckedChanged")]
     public class ToggleButton : Control
     {
         private int W;
         private int H;
         private _Styles style=_Styles.Style3;
         private bool _Checked = false;
         private MouseState State = MouseState.None;
         public event CheckedChangedEventHandler CheckedChanged;
         public delegate void CheckedChangedEventHandler(object sender);
         [Flags()]
         public enum _Styles
         {
             Style1,
             Style2,
             Style3,
             Style4,
             Style5
         }
        [Category("ToggleButton")]
         public _Styles Styles
         {
             get { return style; }
           set { style = value; Invalidate(); }
         }
        [Category("ToggleButton")]
         public bool Checked
         {
             get { return _Checked; }
           set { _Checked = value; Invalidate(); }
         }

         protected override void OnTextChanged(EventArgs e)
         {
             base.OnTextChanged(e);
             Invalidate();
         }
```

```
        protected override void OnResize(EventArgs e)
        {
            base.OnResize(e);
            Width = 76;
            Height = 33;
        }
        protected override void OnMouseEnter(System.EventArgs e)
        {
            base.OnMouseEnter(e);
            State = MouseState.Over;
            Invalidate();
        }
        protected override void OnMouseDown(System.Windows.Forms.MouseEventArgs
e)
        {
            base.OnMouseDown(e);
            State = MouseState.Down;
            Invalidate();
        }
        protected override void OnMouseLeave(System.EventArgs e)
        {
            base.OnMouseLeave(e);
            State = MouseState.None;
            Invalidate();
        }
        protected override void OnMouseUp(System.Windows.Forms.MouseEventArgs e)
        {
            base.OnMouseUp(e);
            State = MouseState.Over;
            Invalidate();
        }
        protected override void OnClick(EventArgs e)
        {
            base.OnClick(e);
            _Checked = !_Checked;
            if (CheckedChanged != null)
            {
                CheckedChanged(this);
            }
        }
       private Color BaseColor = Color.FromArgb(60, 70, 73);
        private Color BaseColorRed = Color.FromArgb(220, 85, 96);
        private Color BGColor = Color.FromArgb(84, 85, 86);
```

```
        private Color ToggleColor = Color.FromArgb(45, 47, 49);
        private Color TextColor = Color.FromArgb(243, 243, 243);
       private Color LightColor = Color.FromArgb(35, 168, 109);
        public ToggleButton()
        {
            SetStyle(ControlStyles.AllPaintingInWmPaint |
ControlStyles.UserPaint | ControlStyles.ResizeRedraw |
ControlStyles.OptimizedDoubleBuffer |
ControlStyles.SupportsTransparentBackColor, true);
            DoubleBuffered = true;
            BackColor = Color.Transparent;
            Size = new Size(44, Height + 1);
            Cursor = Cursors.Hand;
            Font = new Font("Segoe UI", 10);
            Size = new Size(76, 33);
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            Bitmap B = new Bitmap(Width, Height);
            Graphics G = Graphics.FromImage(B);
            W = Width - 1;
            H = Height - 1;
            GraphicsPath GP = new GraphicsPath();
            GraphicsPath GP2 = new GraphicsPath();
            Rectangle Base = new Rectangle(0, 0, W, H);
            Rectangle Toggle = new Rectangle(Convert.ToInt32(W / 2), 0, 38, H);
            var _withG = G;
            _withG.SmoothingMode = SmoothingMode.HighQuality;
            _withG.PixelOffsetMode = PixelOffsetMode.HighQuality;
            _withG.TextRenderingHint = TextRenderingHint.ClearTypeGridFit;
            _withG.Clear(BackColor);
            //根据样式来绘制UI
            switch (style)
            {
                case _Styles.Style1:
                    GP = GDIHelpers.RoundRec(Base, 6);
                    GP2 = GDIHelpers.RoundRec(Toggle, 6);
                    _withG.FillPath(new SolidBrush(BGColor), GP);
                    _withG.FillPath(new SolidBrush(ToggleColor), GP2);
                    //OFF文本
                    _withG.DrawString("OFF", Font, new SolidBrush(BGColor), new
Rectangle(19, 1, W, H), GDIHelpers.CenterSF);

                    if (Checked)
```

```
                    {
                        //调用GDIHelpers类绘制圆角矩形
                        GP = GDIHelpers.RoundRec(Base, 6);
                        GP2 = GDIHelpers.RoundRec(
                    new Rectangle(Convert.ToInt32(W / 2), 0, 38, H), 6);
                        _withG.FillPath(new SolidBrush(ToggleColor), GP);
                        _withG.FillPath(new SolidBrush(BaseColor), GP2);
                        //ON文本
                        _withG.DrawString("ON", Font, new SolidBrush(BaseColor),
new Rectangle(8, 7, W, H), GDIHelpers.NearSF);
                    }
                    break;
                case _Styles.Style2:
                    //Style 2
                    GP = GDIHelpers.RoundRec(Base, 6);
                    Toggle = new Rectangle(4, 4, 36, H - 8);
                    GP2 = GDIHelpers.RoundRec(Toggle, 4);
                    _withG.FillPath(new SolidBrush(BaseColorRed), GP);
                    _withG.FillPath(new SolidBrush(ToggleColor), GP2);
                    _withG.DrawLine(new Pen(BGColor), 18, 20, 18, 12);
                    _withG.DrawLine(new Pen(BGColor), 22, 20, 22, 12);
                    _withG.DrawLine(new Pen(BGColor), 26, 20, 26, 12);
                    //绘制字符图标
                    _withG.DrawString("r", new Font("Marlett", 8), new
SolidBrush(TextColor), new Rectangle(19, 2, Width, Height), GDIHelpers.CenterSF);

                    if (Checked)
                    {
                        GP = GDIHelpers.RoundRec(Base, 6);
                        Toggle = new Rectangle(
                        Convert.ToInt32(W / 2) - 2, 4, 36, H - 8);
                        GP2 = GDIHelpers.RoundRec(Toggle, 4);
                        _withG.FillPath(new SolidBrush(BaseColor), GP);
                        _withG.FillPath(new SolidBrush(ToggleColor), GP2);
                        _withG.DrawLine(new Pen(BGColor), Convert.ToInt32(W / 2)
+ 12, 20, Convert.ToInt32(W / 2) + 12, 12);
                        _withG.DrawLine(new Pen(BGColor), Convert.ToInt32(W / 2)
+ 16, 20, Convert.ToInt32(W / 2) + 16, 12);
                        _withG.DrawLine(new Pen(BGColor), Convert.ToInt32(W / 2)
+ 20, 20, Convert.ToInt32(W / 2) + 20, 12);
                        //对勾字符图标
                        _withG.DrawString("ü", new Font("Wingdings", 14), new
SolidBrush(TextColor), new Rectangle(8, 7, Width, Height), GDIHelpers.NearSF);
                    }
```

```
                    break;
                case _Styles.Style3:
                    //Style 3
                    GP = GDIHelpers.RoundRec(Base, 16);
                    Toggle = new Rectangle(W - 28, 4, 22, H - 8);
                    GP2.AddEllipse(Toggle);
                    _withG.FillPath(new SolidBrush(ToggleColor), GP);
                _withG.FillPath(new SolidBrush(BaseColor), GP2);
                _withG.DrawString("关", Font, new SolidBrush(BaseColor), new
Rectangle(-12,0, W, H), GDIHelpers.CenterSF);
                    if (Checked)
                    {
                        GP = GDIHelpers.RoundRec(Base, 16);
                        Toggle = new Rectangle(6, 4, 22, H - 8);
                        GP2.Reset();
                        GP2.AddEllipse(Toggle);
                        _withG.FillPath(new SolidBrush(ToggleColor), GP);
                    _withG.FillPath(new SolidBrush(LightColor), GP2);
                    _withG.DrawString("开", Font, new SolidBrush(LightColor), new
Rectangle(12,0, W, H), GDIHelpers.CenterSF);
                    }
                    break;
                case _Styles.Style4:
                 //-- Style 4
                 GP = GDIHelpers.RoundRec(Base, 6);
                 GP2 = GDIHelpers.RoundRec(Toggle, 6);
                 _withG.FillPath(new SolidBrush(BGColor), GP);
                 _withG.FillPath(new SolidBrush(ToggleColor), GP2);
                 //-- Text
                 _withG.DrawString("关", Font, new SolidBrush(BGColor), new
Rectangle(19, 1, W, H), GDIHelpers.CenterSF);
                 if (Checked)
                 {
                     GP = GDIHelpers.RoundRec(Base, 6);
                     GP2 = GDIHelpers.RoundRec(new Rectangle(Convert.ToInt32(W /
2), 0, 38, H), 6);
                     _withG.FillPath(new SolidBrush(ToggleColor), GP);
                     _withG.FillPath(new SolidBrush(BaseColor), GP2);
                     _withG.DrawString("开", Font, new SolidBrush(BaseColor), new
Rectangle(8, 7, W, H), GDIHelpers.NearSF);
                 }
                 break;
                case _Styles.Style5:
                 //Style 5
```

```
                GP = GDIHelpers.RoundRec(Base, 6);
                GP2 = GDIHelpers.RoundRec(Toggle, 6);
                _withG.FillPath(new SolidBrush(BGColor), GP);
                _withG.FillPath(new SolidBrush(LightColor), GP2);
                _withG.DrawString("关", Font, new SolidBrush(BGColor), new
Rectangle(19, 1, W, H), GDIHelpers.CenterSF);
                if (Checked)
                {
                    GP = GDIHelpers.RoundRec(Base, 6);
                    GP2 = GDIHelpers.RoundRec(new Rectangle(Convert.ToInt32(W /
2), 0, 38, H), 6);
                    _withG.FillPath(new SolidBrush(ToggleColor), GP);
                    _withG.FillPath(new SolidBrush(BaseColor), GP2);
                    _withG.DrawString("开", Font, new SolidBrush(BaseColor), new
Rectangle(8, 7, W, H), GDIHelpers.NearSF);
                }
                break;
            }
            base.OnPaint(e);
            G.Dispose();
            e.Graphics.InterpolationMode = InterpolationMode.HighQualityBicubic;
            e.Graphics.DrawImageUnscaled(B, 0, 0);
            B.Dispose();
        }
    }
}
```

4.7.4 控件应用

编译后， Visual Studio IDE 会自动生成了一个 ToggleButton 控件，新创建一个窗体 ToggleButtonTest（窗体标题 ToggelButtonTest 为拼写错误），从工具箱中将 ToggleButton 控件拖放到窗体上，如图 4.31 所示：

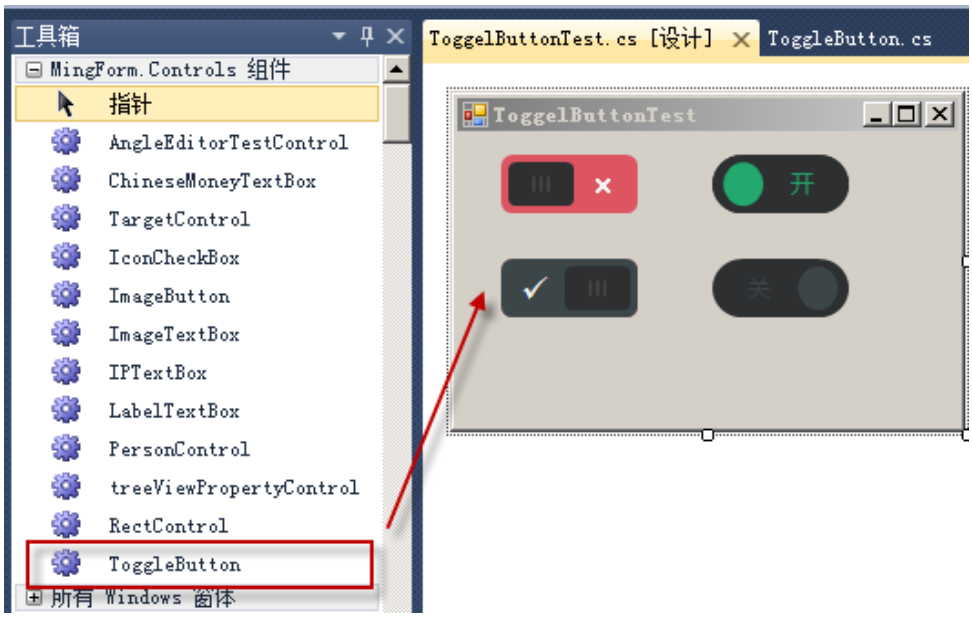


图 4.31 ToggleButton 控件示例

ToggleButtonTest 窗体代码如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace CustomControls
{
    using MingForm.Controls;
    public partial class ToggleButtonTest : Form
    {
        public ToggelButtonTest()
        {
            InitializeComponent();
        }
        private void ToggleButtonTest_Load(object sender, EventArgs e)
        {
            this.toggleButton1.Styles = ToggleButton._Styles.Style3;
            this.toggleButton2.Checked = true;
            this.toggleButton2.Styles = ToggleButton._Styles.Style2;
            this.toggleButton2.Checked = false;
        }
    }
}
```

运行该窗体，界面如图4.32所示：

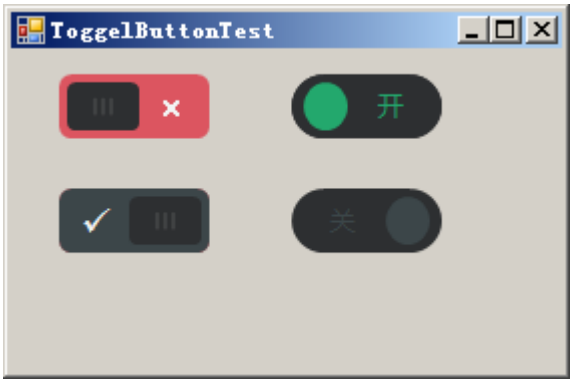


图 4.32 ImageButton 控件示例

4.8 IconCaptionPanel 控件

上面定义的控件不能包含其他子控件，这一节定义一个 IconCaptionPanel 控件，可以包含其他子控件。

4.8.1 控件功能

IconCaptionPanel 控件的核心功能是：和 Panel 控件类似，可以将其他控件包含其中，成为其子控件，另外在 Panel 控件的基础上，添加一个 Caption 标题区域，且可以在标题区域自定义一个图标。

4.8.2 控件设计

根据 IconCaptionPanel 控件的功能需求，我们可以将该控件的 UI 设计如图 4.33 所示：

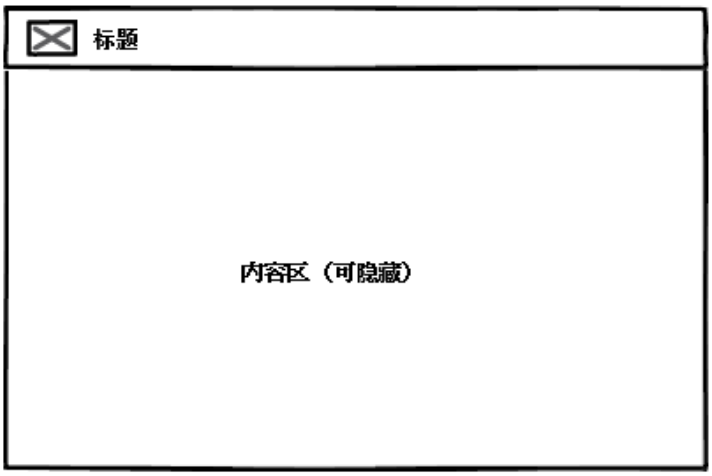


图 4.33 IconCaptionPanel 控件 UI

4.8.3 控件开发

IconPanel 控件我们继承 ScrollableControl，并继承 IContainerControl 接口，IconPanel 控件代码如下：

```
using System;
using System.ComponentModel;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Text;
using System.Windows.Forms;
namespace MingForm.Controls
{
    [DefaultEvent("Click")]
    public class IconPanel : ScrollableControl,IContainerControl
    {
       public static Color BGColor = Color.FromArgb(35, 168, 109);
        private int W;
        private int H;
        private MouseState State = MouseState.None;
        private System.Windows.Forms.Panel panel;
       private Control activeControl;
       [Category("IconPanel")]
        public override string Text
        {
            get { return base.Text; }
            set
            {
                base.Text = value;
                if (panel != null)
                {
                    panel.Text = value;
                }
            }
        }
       [Category("IconPanel")]
        public override Font Font
        {
            get { return base.Font; }
            set
            {
```

```
            base.Font = value;
            if (panel != null)
            {
                panel.Font = value;
                panel.Location = new Point(0, 0);
                panel.Width = Width -2;
            }
        }
    }
private bool isClose = false;
[Category("IconPanel")]
public bool IsClose
{
    get { return isClose; }
    set
    {
        isClose = value;
        Invalidate();
    }
}
 protected override void OnCreateControl()
 {
     base.OnCreateControl();
     if (!Controls.Contains(panel))
     {
         Controls.Add(panel);
     }
 }
 private void OnBaseTextChanged(object s, EventArgs e)
 {
     Text = panel.Text;
 }
private bool isfirst = true;
 protected override void OnResize(EventArgs e)
 {
   panel.Location = new Point(1, 35);
   if (isfirst)
   {
       __orginHeight = Height;
       isfirst = false;
   }
   panel.Width = Width - 2;
   panel.Height = Height - 35;
   this.VerticalScroll.Visible = false;
```

```
            base.OnResize(e);
        }
        [Category("IconPanel")]
        public Color TextColor
        {
            get { return _TextColor; }
            set { _TextColor = value; }
        }
        public override Color ForeColor
        {
            get { return _TextColor; }
            set { _TextColor = value; }
        }
        private string iconText = "ü";
        [Category("IconPanel"), Browsable(true)]
        [Description("icon font text")]
        public string IconText
        {
            get { return iconText; }
            set { iconText = value; Invalidate(); }
        }
        protected override void OnMouseDown(MouseEventArgs e)
        {
            base.OnMouseDown(e);
            State = MouseState.Down;
            Invalidate();
        }
        protected override void OnMouseUp(MouseEventArgs e)
        {
            base.OnMouseUp(e);
            State = MouseState.Over;
            panel.Focus();
            Invalidate();
        }
        protected override void OnMouseEnter(EventArgs e)
        {
            base.OnMouseEnter(e);
            State = MouseState.Over;
            Invalidate();
        }
        protected override void OnMouseLeave(EventArgs e)
        {
            base.OnMouseLeave(e);
            State = MouseState.None;
```

```
            Invalidate();
        }
        private Color _BaseColor = Color.FromArgb(45, 47, 49);
        private Color _TextColor = Color.FromArgb(192, 192, 192);
        private Color _BorderColor = BGColor;
       public IconPanel()
        {
            SetStyle(ControlStyles.AllPaintingInWmPaint |
ControlStyles.UserPaint | ControlStyles.ResizeRedraw |
ControlStyles.OptimizedDoubleBuffer |
ControlStyles.SupportsTransparentBackColor, true);
            DoubleBuffered = true;
            BackColor = Color.Transparent;
            panel = new System.Windows.Forms.Panel();
            panel.Font = new Font("Segoe UI", 10);
            panel.Text = Text;
            panel.BackColor = _BaseColor;
            panel.ForeColor = _TextColor;
           //隐藏掉边框
           panel.BorderStyle = BorderStyle.None;
            panel.Location = new Point(1, 35);
            panel.Width = Width -2;
           this.Cursor = Cursors.Hand;
           panel.Height = Height-36;
            //dock fill控件可以显示标题栏padding top =35实现
          this.Padding = new Padding(0, 35,1,1);
          this.Click += new EventHandler(BT_Click);
          this.PaddingChanged += new EventHandler(IconPanel_PaddingChanged);
          this.AutoScroll = false;
        }
       void IconPanel_PaddingChanged(object sender, EventArgs e)
       {
           this.Padding = new Padding(0, 35, 1, 1);
       }
       private int __orginHeight = 0;
       void BT_Click(object sender, EventArgs e)
       {

           isClose = !isClose;
           if (!isClose)
           {
               panel.Location = new Point(1, 35);
               Height =( __orginHeight==0 ? 260:__orginHeight);
               panel.Width = Width - 2;
```

```
            panel.Height = Height - 35;
            this.panel.Visible = true;
        }
        else
        {
            this.panel.Visible = false;
            this.panel.Height = 0;
            Height = 35;
        }
    }
     protected override void OnPaint(PaintEventArgs e)
     {
        _BorderColor = BGColor;
        if (!isClose)
        {
            this.panel.Visible = true;
        }
        else
        {
            this.panel.Visible = false;
        }
          Bitmap B = new Bitmap(Width, Height);
          Graphics G = Graphics.FromImage(B);
          W = Width - 1;
          H = Height - 1;
          Rectangle Base = new Rectangle(0, 0, W, H);
          var _withG = G;
          _withG.SmoothingMode = SmoothingMode.HighQuality;
          _withG.PixelOffsetMode = PixelOffsetMode.HighQuality;
          _withG.TextRenderingHint = TextRenderingHint.ClearTypeGridFit;
          _withG.Clear(BackColor);
        Rectangle Button = new Rectangle(0, 0, W, 35);
        GraphicsPath GP = new GraphicsPath();
        GraphicsPath GP2 = new GraphicsPath();
        panel.BackColor = _BaseColor;
        panel.ForeColor = _TextColor;
          _withG.FillRectangle(new SolidBrush(_BaseColor), Base);
        GP.Reset();
        GP.AddRectangle(Button);
        _withG.SetClip(GP);
        _withG.FillRectangle(new SolidBrush(BGColor), Button);
        _withG.ResetClip();
        _withG.DrawString(IconText, new Font("Wingdings", 22), new
SolidBrush(_BaseColor), new Rectangle(6, 6, 30, 30), new StringFormat
```

```
            {
                Alignment = StringAlignment.Center,
                LineAlignment = StringAlignment.Center
            });
            _withG.DrawString(Text, new Font("Segoe UI", 12), new
SolidBrush(_BaseColor), new Rectangle(38,3, W-30, 30), new StringFormat
            {
                Alignment = StringAlignment.Near,
                LineAlignment = StringAlignment.Center
            });
              base.OnPaint(e);
              G.Dispose();
              e.Graphics.InterpolationMode = InterpolationMode.HighQualityBicubic;
              e.Graphics.DrawImageUnscaled(B, 0, 0);
              B.Dispose();
         }
        public bool ActivateControl(Control active)
        {
            if (this.Controls.Contains(active))
            {
                active.Select();
                this.ScrollControlIntoView(active);
                this.activeControl = active;
                return true;
            }
            return false;
        }
        public Control ActiveControl
        {
            get
            {
                return activeControl;
            }
            set
            {
                if (this.Controls.Contains(value))
                {
                    activeControl = value;
                }
            }
        }
    }
}
```

4.8.4 控件应用

编译后，Visual Studio IDE 自动生成一个控件，新创建一个窗体 IconPanelTest，从工具箱中将 IconPanel 控件拖放到窗体界面上，并设置其属性，如图 4.34 所示：

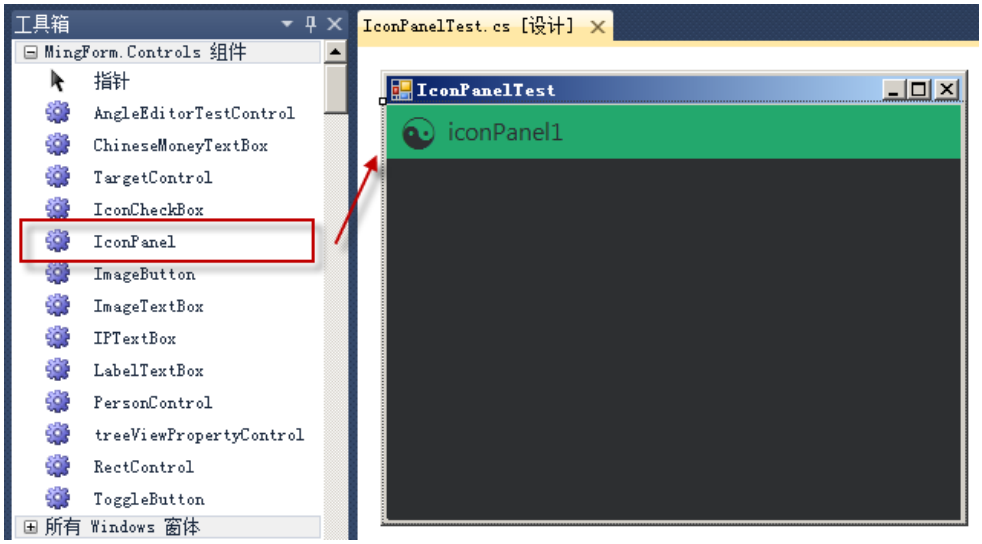


图 4.34 IconPanel 控件拖放

我们可以将一个 datagridview 控件 Fill 到这个 IconPanel 中，当我们单击标题栏时，可以对内容区域进行显示和隐藏的切换，如图 4.35 所示。

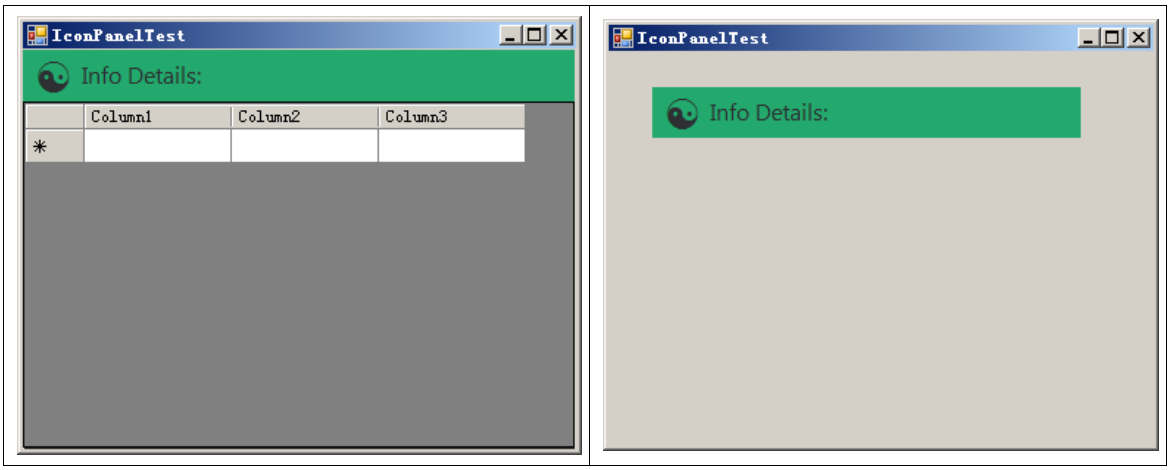


图 4.35 IconPanel 控件示例

4.9 IconTabControl 控件

上一节介绍了 IconPanel 控件的开发，它可以作为其他控件的容器，并且这个 Panel 内容区域可以隐藏。这一节我们介绍如何在原生 TabControl 控件的基础上，用重绘的方式将其 UI 进行美化，并重新重开发一个自定义控件 IconTabControl 控件。

IconTabControl 控件继承自原生 TabControl 控件，我们将其 UI 扁平化，并实现可以在页签的标题前绘制字体图标。

4.9.1 控件功能

IconTabControl 控件的核心功能是：IconTabControl 控件具备原生 TabControl 控件的功能，同时 UI 扁平化，可以在页签的标题前面绘制字体图标。

4.9.2 控件设计

根据 IconTabControl 控件的功能要求，我们可以将其 UI 设计如图 4.36 所示：

图 4.36 IconTabControl 控件 UI

4.9.3 控件开发

我们在 Visual Studio IDE 中添加一个继承自 TabControl 的 IconTabControl 控件，IconTabControl 控件代码如下：

```
using System;
using System.ComponentModel;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Text;
using System.Windows.Forms;
namespace MingForm.Controls
{
    public class IconTabControl : TabControl
    {
```

```
public static Color ACColor = Color.FromArgb(35, 168, 109);
 private int W;
 private int H;
 protected override void CreateHandle()
 {
     base.CreateHandle();
     Alignment = TabAlignment.Top;
 }
[Category("IconTabControl")]
public string[] IconTexts
{
   get { return _IconTexts; }
   set {
       if (value != null)
       {
           if (value.Length == TabCount)
           {
               _IconTexts = value; Invalidate();
           }
           else
           {
               MessageBox.Show("IconTexts个数和Tab页个数不匹配");
           }
       }
   }
}
[Category("IconTabControl")]
 public Color BaseColor
 {
     get { return _BaseColor; }
     set { _BaseColor = value; }
 }
[Category("IconTabControl")]
 public Color ActiveColor
 {
     get { return _ActiveColor; }
     set { _ActiveColor = value; }
 }
private string[] _IconTexts =null;
 private Color BGColor = Color.FromArgb(60, 70, 73);
 private Color _BaseColor = Color.FromArgb(45, 47, 49);
 private Color _ActiveColor = Helpers.FlatColor;
 public IconTabControl()
 {
```

```
            SetStyle(ControlStyles.AllPaintingInWmPaint |
ControlStyles.UserPaint | ControlStyles.ResizeRedraw |
ControlStyles.OptimizedDoubleBuffer, true);
            DoubleBuffered = true;
            BackColor = Color.FromArgb(60, 70, 73);
            Font = new Font("Segoe UI", 10);
            SizeMode = TabSizeMode.Fixed;
            ItemSize = new Size(120, 40);
        }
        protected override void OnPaint(PaintEventArgs e)
        {
          _ActiveColor = ACColor;
            Bitmap B = new Bitmap(Width, Height);
            Graphics G = Graphics.FromImage(B);
            W = Width - 1;
            H = Height - 1;
            var _withG = G;
            _withG.SmoothingMode = SmoothingMode.HighQuality;
            _withG.PixelOffsetMode = PixelOffsetMode.HighQuality;
            _withG.TextRenderingHint = TextRenderingHint.ClearTypeGridFit;
            _withG.Clear(_BaseColor);
            try
            {
                SelectedTab.BackColor = BGColor;
            }
            catch
            {
            }
            for (int i = 0; i <= TabCount - 1; i++)
            {
                Rectangle Base = new Rectangle(new Point(GetTabRect(i).Location.X
+ 2, GetTabRect(i).Location.Y), new Size(GetTabRect(i).Width,
GetTabRect(i).Height));
                Rectangle BaseSize = new Rectangle(Base.Location, new
Size(Base.Width, Base.Height));
                if (i == SelectedIndex)
                {
                    _withG.FillRectangle(new SolidBrush(_BaseColor), BaseSize);
                    _withG.FillRectangle(new SolidBrush(_ActiveColor),
BaseSize);
                    if (ImageList != null)
                    {
                        try
                        {
```

```
                            //有对应的ImageList，我们可以流出空间来绘制image
                            if (ImageList.Images[TabPages[i].ImageIndex] != null)
                            {
    _withG.DrawImage(ImageList.Images[TabPages[i].ImageIndex], new
Point(BaseSize.Location.X + 8, BaseSize.Location.Y + 6));
                                _withG.DrawString("     " + TabPages[i].Text,
Font, Brushes.White, BaseSize, Helpers.CenterSF);
                            }
                            else
                            {
                                //绘制页签标题
                                _withG.DrawString(TabPages[i].Text, Font,
Brushes.White, BaseSize, Helpers.CenterSF);
                            }
                        }
                        catch (Exception ex)
                        {
                            throw new Exception(ex.Message);
                        }
                    }
                else if (IconTexts != null)
                {
                    try
                    {
                        if (IconTexts[i]!= "")
                        {
                            //绘制字符图标
                            _withG.DrawString(IconTexts[i], new Font("Wingdings",
22), new SolidBrush(_BaseColor), new Point(BaseSize.Location.X + 8,
BaseSize.Location.Y + 6));
                            //绘制页签标题
                            _withG.DrawString("     " + TabPages[i].Text, Font,
Brushes.White, BaseSize, Helpers.CenterSF);
                        }
                        else
                        {
                            //只绘制页签标题
                            _withG.DrawString(TabPages[i].Text, Font,
Brushes.White, BaseSize, Helpers.CenterSF);
                        }
                    }
                    catch (Exception ex)
                    {
                        throw new Exception(ex.Message);
```

```
                    }
                }
                    else
                    {
                            _withG.DrawString(TabPages[i].Text, Font,
Brushes.White, BaseSize, Helpers.CenterSF);
                    }
                }
                else
                {
                    _withG.FillRectangle(new SolidBrush(_BaseColor), BaseSize);
                    if (ImageList != null)
                    {
                        try
                        {
                            if (ImageList.Images[TabPages[i].ImageIndex] != null)
                            {
    _withG.DrawImage(ImageList.Images[TabPages[i].ImageIndex], new
Point(BaseSize.Location.X + 8, BaseSize.Location.Y + 6));
                                _withG.DrawString("     " + TabPages[i].Text,
Font, new SolidBrush(Color.White), BaseSize, new StringFormat
                                {
                                    LineAlignment = StringAlignment.Center,
                                    Alignment = StringAlignment.Center
                                });
                            }
                            else
                            {
                                _withG.DrawString(TabPages[i].Text, Font, new
SolidBrush(Color.White), BaseSize, new StringFormat
                                {
                                    LineAlignment = StringAlignment.Center,
                                    Alignment = StringAlignment.Center
                                });
                            }
                        }
                        catch (Exception ex)
                        {
                            throw new Exception(ex.Message);
                        }
                    }
                else if (IconTexts != null)
                {
                    try
```

```
                    {
                        if (IconTexts[i] != "")
                        {
                            _withG.DrawString(IconTexts[i], new Font("Wingdings",
22), new SolidBrush(_ActiveColor), new Point(BaseSize.Location.X + 8,
BaseSize.Location.Y + 6));
                            _withG.DrawString("     " + TabPages[i].Text, Font,
Brushes.White, BaseSize, Helpers.CenterSF);
                        }
                        else
                        {
                            _withG.DrawString(TabPages[i].Text, Font,
Brushes.White, BaseSize, Helpers.CenterSF);
                        }
                    }
                    catch (Exception ex)
                    {
                        throw new Exception(ex.Message);
                    }
                }
                else
                {
                    _withG.DrawString(TabPages[i].Text, Font, new
SolidBrush(Color.White), BaseSize, new StringFormat
                    {
                        LineAlignment = StringAlignment.Center,
                        Alignment = StringAlignment.Center
                    });
                }
            }
        }
        base.OnPaint(e);
        G.Dispose();
        e.Graphics.InterpolationMode = InterpolationMode.HighQualityBicubic;
        e.Graphics.DrawImageUnscaled(B, 0, 0);
        B.Dispose();
    }
}
}
```

4.9.4 控件应用

编译后，Visual Studio IDE 自动生成一个控件 IconTabControl，新创建一个窗

体 IconTabControlTest，从工具箱中将 IconTabControl 控件拖放到窗体上，并设置属性和内容，如图 4.37 所示：

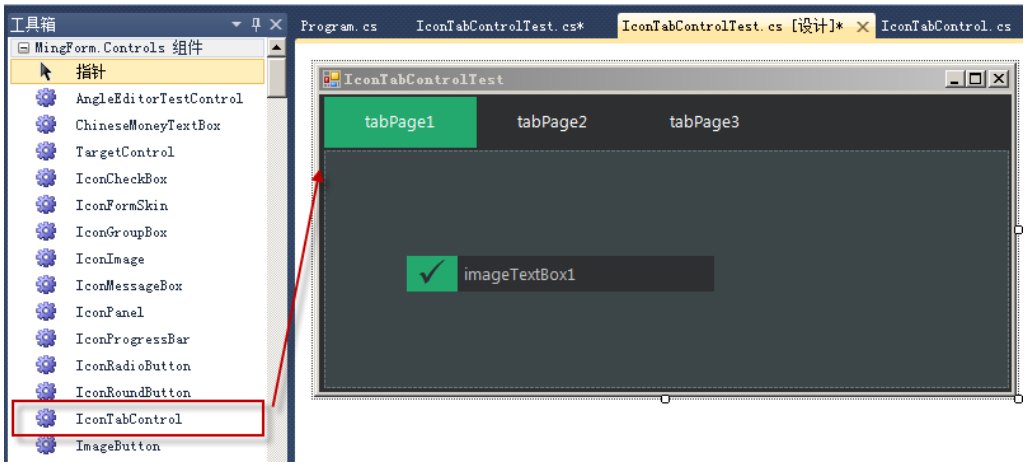


图 4.37 IconTabControl 控件拖放

IconTabControlTest 代码如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace CustomControls
{
    public partial class IconTabControlTest : Form
    {
        public IconTabControlTest()
        {
            InitializeComponent();
        }
        private void IconTabControlTest_Load(object sender, EventArgs e)
        {
            //tab count=3
            this.iconTabControl1.IconTexts = new string[] {"6","8","2"};
            this.iconTabControl1.TabPages[0].Text = "Emp";
            this.iconTabControl1.TabPages[1].Text = "Org";
            this.iconTabControl1.TabPages[2].Text = "File";
        }
    }
}
```

运行IconTabControlTest，界面如图4.38所示：

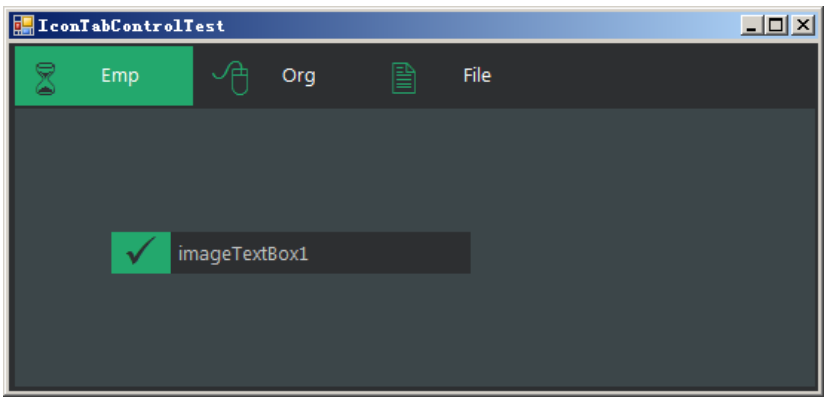


图 4.38 IconTabControl 控件示例

4.10 IconMessageBox 控件

在管理信息系统的日常开发中，经常会根据用户的操作来触发各种后台操作，为了明确告知用户某个操作是否正确处理，需要用消息框进行提示。Windows 自带的 MessageBox 可以实现这样的功能，但是界面美观度欠佳，这一节我们就自定义一个 IconMessageBox 空间，可以替代 MessageBox 进行消息提醒。

4.10.1 控件功能

IconMessageBox 控件的核心功能是：能够方便的通过控件方法来显示和隐藏提示框，同时可以配置提示界面上的图标和文本信息，提示后可以间隔某个时间段自动隐藏，也可以根据用户的操作来手动关闭该提示框。另外就是提示框有类型（信息、成功和错误），不同类型默认的文本颜色是不同的，例如错误文本颜色是红色的，而成功文本 颜色为绿色。

4.10.2 控件设计

根据 IconMessageBox 控件的功能要求，我们可以将其 UI 设计如图 4.39 所示：

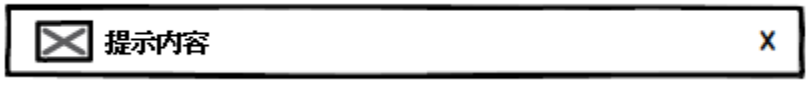


图 4.39 IconMessageBox 控件 UI

4.10.3 控件开发

首先在 Visual Studio IDE 中创建一个名为 IconMessageBox 的组件，它继承自 System.Windows.Forms.Control，代码如下。

```
using System;
using System.ComponentModel;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Text;
using System.Windows.Forms;
namespace MingForm.Controls
{
    public class IconMessageBox : Control
    {
        private int W;
        private int H;
        private _Types _Type;
        private string _Text;
        private MouseState State = MouseState.None;
        private int X;
        private Timer withEventsField_T;
        private Timer T
        {
            get { return withEventsField_T; }
            set
            {
                if (withEventsField_T != null)
                {
                    withEventsField_T.Tick -= T_Tick;
                }
                withEventsField_T = value;
                if (withEventsField_T != null)
                {
                    withEventsField_T.Tick += T_Tick;
                }
            }
        }
        [Flags()]
        public enum _Types
        {
            Success,
            Error,
```

```
        Info
    }
    [Category("IconMessageBox")]
    public _Types BoxType
    {
        get { return _Type; }
        set { _Type = value; }
    }
    [Category("IconMessageBox")]
    public override string Text
    {
        get { return base.Text; }
        set
        {
            base.Text = value;
            if (_Text != null)
            {
                _Text = value;
            }
        }
    }
    [Category("IconMessageBox")]
    public new bool Visible
    {
        get { return base.Visible == false; }
        set { base.Visible = value; }
    }
    protected override void OnTextChanged(EventArgs e)
    {
        base.OnTextChanged(e);
        Invalidate();
    }
    protected override void OnResize(EventArgs e)
    {
        base.OnResize(e);
        Height = 42;
    }
    public void ShowControl(_Types type, string Str, int Interval)
    {
      _Type = type;
        Text = Str;
        this.Visible = true;
        T = new Timer();
        T.Interval = Interval;
```

```
        T.Enabled = true;
    }
    private void T_Tick(object sender, EventArgs e)
    {
        this.Visible = false;
        T.Enabled = false;
        T.Dispose();
    }
    protected override void OnMouseDown(MouseEventArgs e)
    {
        base.OnMouseDown(e);
        State = MouseState.Down;
        Invalidate();
    }
    protected override void OnMouseUp(MouseEventArgs e)
    {
        base.OnMouseUp(e);
        State = MouseState.Over;
        Invalidate();
    }
    protected override void OnMouseEnter(EventArgs e)
    {
        base.OnMouseEnter(e);
        State = MouseState.Over;
        Invalidate();
    }
    protected override void OnMouseLeave(EventArgs e)
    {
        base.OnMouseLeave(e);
        State = MouseState.None;
        Invalidate();
    }
    protected override void OnMouseMove(MouseEventArgs e)
    {
        base.OnMouseMove(e);
        X = e.X;
        Invalidate();
    }
    protected override void OnClick(EventArgs e)
    {
        base.OnClick(e);
        this.Visible = false;
    }
    private Color SuccessColor = Color.FromArgb(60, 85, 79);
```

```
        private Color SuccessText = Color.FromArgb(35, 169, 110);
        private Color ErrorColor = Color.FromArgb(87, 71, 71);
        private Color ErrorText = Color.FromArgb(254, 142, 122);
        private Color InfoColor = Color.FromArgb(70, 91, 94);
        private Color InfoText = Color.FromArgb(97, 185, 186);
        public IconMessageBox()
        {
            SetStyle(ControlStyles.AllPaintingInWmPaint |
ControlStyles.UserPaint | ControlStyles.ResizeRedraw |
ControlStyles.OptimizedDoubleBuffer, true);
            DoubleBuffered = true;
            BackColor = Color.FromArgb(60, 70, 73);
            Size = new Size(300, 42);
            Location = new Point(10, 61);
            Font = new Font("Segoe UI", 10);
            Cursor = Cursors.Hand;
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            Bitmap B = new Bitmap(Width, Height);
            Graphics G = Graphics.FromImage(B);
            W = Width - 1;
            H = Height - 1;
            Rectangle Base = new Rectangle(0, 0, W, H);
            var _withG = G;
            _withG.SmoothingMode = SmoothingMode.HighQuality;
            _withG.TextRenderingHint = TextRenderingHint.ClearTypeGridFit;
            _withG.Clear(BackColor);
            switch (_Type)
            {
                case _Types.Success:
                    _withG.FillRectangle(new SolidBrush(SuccessColor), Base);
                    _withG.FillEllipse(new SolidBrush(SuccessText), new
Rectangle(8, 9, 24, 24));
                    _withG.FillEllipse(new SolidBrush(SuccessColor), new
Rectangle(10, 11, 20, 20));
                    //对勾字体图标
                    _withG.DrawString("ü", new Font("Wingdings", 22), new
SolidBrush(SuccessText), new Rectangle(7, 7, W, H), Helpers.NearSF);
                    _withG.DrawString(Text, Font, new SolidBrush(SuccessText),
new Rectangle(48, 12, W, H), Helpers.NearSF);

                    //X 关闭按钮
                    _withG.FillEllipse(new SolidBrush(Color.FromArgb(35,
```

```
Color.Black)), new Rectangle(W - 30, H - 29, 17, 17));
                    _withG.DrawString("r", new Font("Marlett", 8), new
SolidBrush(SuccessColor), new Rectangle(W - 28, 16, W, H), Helpers.NearSF);
                    switch (State)
                    {
                        case MouseState.Over:
                            _withG.DrawString("r", new Font("Marlett", 8), new
SolidBrush(Color.FromArgb(25, Color.White)), new Rectangle(W - 28, 16, W, H),
Helpers.NearSF);
                            break;
                    }
                    break;
                case _Types.Error:
                    _withG.FillRectangle(new SolidBrush(ErrorColor), Base);
                    _withG.FillEllipse(new SolidBrush(ErrorText), new Rectangle(8,
9, 24, 24));
                    _withG.FillEllipse(new SolidBrush(ErrorColor), new
Rectangle(10, 11, 20, 20));
                    _withG.DrawString("r", new Font("Marlett", 16), new
SolidBrush(ErrorText), new Rectangle(6, 11, W, H), Helpers.NearSF);
                    _withG.DrawString(Text, Font, new SolidBrush(ErrorText), new
Rectangle(48, 12, W, H), Helpers.NearSF);
                    _withG.FillEllipse(new SolidBrush(Color.FromArgb(35,
Color.Black)), new Rectangle(W - 32, H - 29, 17, 17));
                    _withG.DrawString("r", new Font("Marlett", 8), new
SolidBrush(ErrorColor), new Rectangle(W - 30, 17, W, H), Helpers.NearSF);
                    switch (State)
                    {
                        case MouseState.Over:
                            _withG.DrawString("r", new Font("Marlett", 8), new
SolidBrush(Color.FromArgb(25, Color.White)), new Rectangle(W - 30, 15, W, H),
Helpers.NearSF);
                            break;
                    }
                    break;
                case _Types.Info:
                    _withG.FillRectangle(new SolidBrush(InfoColor), Base);
                    _withG.FillEllipse(new SolidBrush(InfoText), new Rectangle(8,
9, 24, 24));
                    _withG.FillEllipse(new SolidBrush(InfoColor), new
Rectangle(10, 11, 20, 20));

                    _withG.DrawString("¡", new Font("Segoe UI", 20,
FontStyle.Bold), new SolidBrush(InfoText), new Rectangle(12, -4, W, H),
```

```
Helpers.NearSF);
                    _withG.DrawString(Text, Font, new SolidBrush(InfoText), new
Rectangle(48, 12, W, H), Helpers.NearSF);
                    _withG.FillEllipse(new SolidBrush(Color.FromArgb(35,
Color.Black)), new Rectangle(W - 32, H - 29, 17, 17));
                    _withG.DrawString("r", new Font("Marlett", 8), new
SolidBrush(InfoColor), new Rectangle(W - 30, 17, W, H), Helpers.NearSF);
                    switch (State)
                    {
                        case MouseState.Over:
                            _withG.DrawString("r", new Font("Marlett", 8), new
SolidBrush(Color.FromArgb(25, Color.White)), new Rectangle(W - 30, 17, W, H),
Helpers.NearSF);
                            break;
                    }
                    break;
            }
            base.OnPaint(e);
            G.Dispose();
            e.Graphics.InterpolationMode = InterpolationMode.HighQualityBicubic;
            e.Graphics.DrawImageUnscaled(B, 0, 0);
            B.Dispose();
        }
    }
}
```

4.10.4 控件应用

编译后，Visual Studio IDE 可以自动生成一个 IconMessageBox 控件，新创建一个窗体 IconMessageBoxTest，从工具箱中将 IconMessageBox 控件拖放到界面上，如图 4.40 所示：

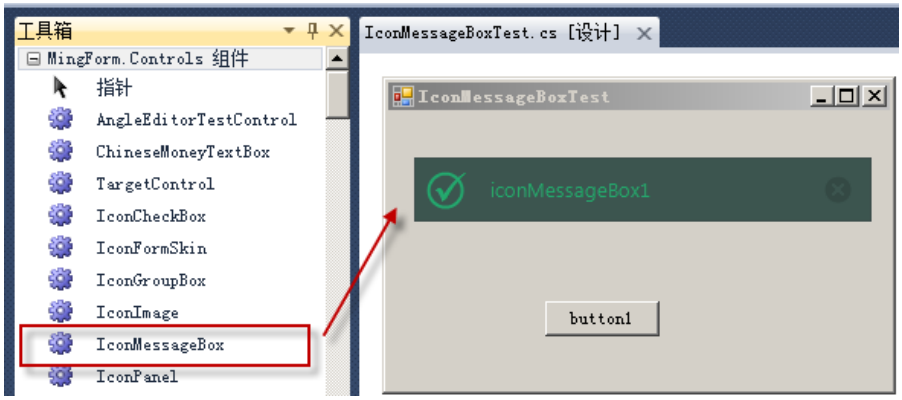


图 4.40 IconMessageBox 控件拖放

IconMessageBoxTest 窗体代码如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace CustomControls
{
    public partial class IconMessageBoxTest : Form
    {
        public IconMessageBoxTest()
        {
            InitializeComponent();
        }
        private void button1_Click(object sender, EventArgs e)
        {
            this.iconMessageBox1.ShowControl(
                MingForm.Controls.IconMessageBox._Types.Success,
                "操作成功",
                3000);
        }
    }
}
```

运行窗体，界面如图4.41所示：

图 4.41 IconMessageBoxT 控件示例

4.11 FlatRoundImage 控件

在很多手机 app 上，登录界面上都有一个个性的头像设置，这个头像是一个圆形的区域，

但是用户上传矩形的头像往往是矩形的，那么如何实现这种效果呢？这一节我们就定义一个 FlatRoundImage 控件。

4.11.1 控件功能

FlatRoundImage 控件的核心功能是：不管要显示的图片是何形状，该控件都是用一个圆形来呈现。控件可以自定义要显示的图片资源。

4.11.2 控件设计

根据 FlatRoundImage 控件的功能需求，其 UI 设计如图 4.42 所示：

图 4.42 FlatRoundImage 控件 UI

4.11.3 控件开发

首先在 Visual Studio IDE 中创建一个名为 FlatRoundImage 的组件，它继承自 System.Windows.Forms.Control，代码如下。

```
using System;
using System.ComponentModel;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Text;
using System.Windows.Forms;

namespace MingForm.Controls
{
    //[Description("Flat圆形图片控件")]
    [DefaultProperty("CurImage")]
    [DefaultEvent("Click")]
```

```
    public class FlatRoundImage : Control
    {
        #region Fields
        private int W;
         private int H;
        private MouseState State = MouseState.None;
        private Color _RoundCircleColor = Color.White;
        private Image _CurImage = Properties.Resources.g;
        private Color _BGColor =
System.Drawing.ColorTranslator.FromHtml("#2C3E50");
        private Color _HoverColor =
System.Drawing.ColorTranslator.FromHtml("#2980B9");
        #endregion
        #region Properties
        [Category("FlatRoundImage")]
        public Color RoundCircleColor
        {
            get { return _RoundCircleColor; }
            set { _RoundCircleColor = value; Invalidate(); }
        }
        [Category("FlatRoundImage")]
        public Color HoverColor
        {
            get { return _HoverColor; }
            set { _HoverColor = value; }
        }
        [Category("FlatRoundImage")]
        public Image CurImage
        {
            get { return _CurImage; }
            set { _CurImage = value; Invalidate(); }
        }
        #endregion
        #region Constructor
        public FlatRoundImage()
         {
             SetStyle(ControlStyles.AllPaintingInWmPaint |
ControlStyles.UserPaint | ControlStyles.ResizeRedraw |
ControlStyles.OptimizedDoubleBuffer, true);
             DoubleBuffered = true;
             ForeColor = Color.White;
            Width = 100;
             Cursor = Cursors.Hand;
            Height = Width;
```

```
            Font = new Font("Segoe UI", 8, FontStyle.Regular);
        }
#endregion
#region override Events
protected override void OnPaint(PaintEventArgs e)
{
    Bitmap B = new Bitmap(Width, Height);
    Graphics G = Graphics.FromImage(B);
    W = Width;
    H = Height;
    Rectangle Base = new Rectangle(0, 0, W, H);
    GraphicsPath GP = new GraphicsPath();
    var _withG = G;
    _withG.Clear(BackColor);
    _withG.SmoothingMode = SmoothingMode.HighQuality;
    _withG.PixelOffsetMode = PixelOffsetMode.HighQuality;
    _withG.TextRenderingHint = TextRenderingHint.ClearTypeGridFit;
    _withG.FillRectangle(new SolidBrush(_BGColor), Base);
    Region region = new System.Drawing.Region();
    GP.AddRectangle(Base);
    region.Union(GP);
    _withG.DrawImage(CurImage, Base);
    Pen p = new Pen(new SolidBrush(RoundCircleColor), 8);
    if (State == MouseState.Over)
    {
        p = new Pen(new SolidBrush(_HoverColor), 9);
    }
    else
    {
        p = new Pen(new SolidBrush(RoundCircleColor), 8);
    }
    GP.AddEllipse(2, 2, Width - 4, Height - 4);
    region.Intersect(GP);
    _withG.DrawEllipse(p, 2, 2, Width - 4, Height - 4);
    _withG.FillRegion(new SolidBrush(BackColor), region);
    G.Dispose();
    e.Graphics.InterpolationMode = InterpolationMode.HighQualityBicubic;
    e.Graphics.DrawImageUnscaled(B, 0, 0);
    B.Dispose();
}
protected override void OnMouseEnter(EventArgs e)
{
    base.OnMouseEnter(e);
    State = MouseState.Over;
```

```
            Invalidate();
        }
        protected override void OnMouseLeave(EventArgs e)
        {
            base.OnMouseLeave(e);
            State = MouseState.None;
            Invalidate();
        }
        protected override void OnResize(EventArgs e)
        {
            base.OnResize(e);
            Height = Width;
        }
        #endregion
    }
}
```

4.11.4 控件应用

编译后，Visual Studio IDE 可以自动生成一个 FlatRoundImage 控件，新创建一个窗体 FlatRoundImageTest，从工具箱中将 FlatRoundImage 控件拖放到设计界面上，如图 4.43 所示：

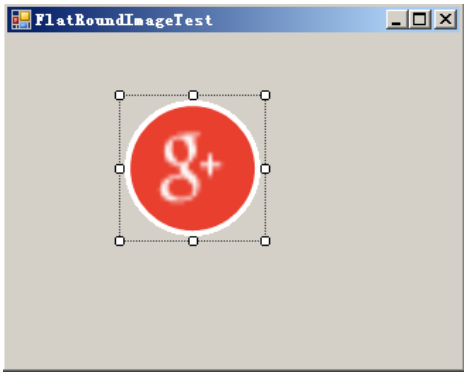


图 4.43 FlatRoundImage 控件示例

我们可以选中 FlatRoundImage 控件，然后配置其属性，我们可以指定其 CurImage 属性来更换图形，如图 4.44 所示：

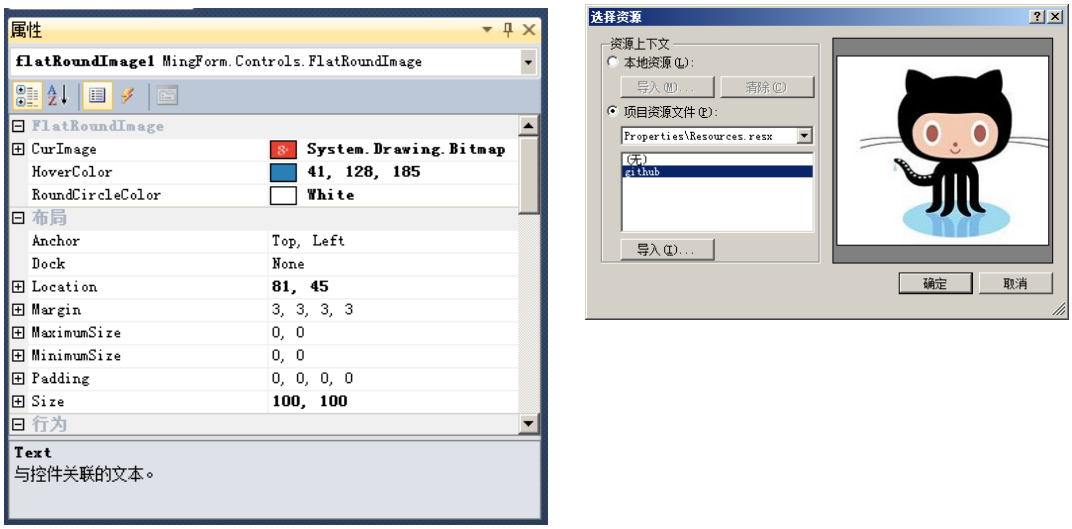


图 4.44 FlatRoundImage 控件示例

FlatRoundImageTest 代码如下：

```
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace CustomControls
{
    public partial class FlatRoundImageTest : Form
    {
        public FlatRoundImageTest()
        {
            InitializeComponent();
        }
        private void FlatRoundImageTest_Load(object sender, EventArgs e)
        {
            this.flatRoundImage1.CurImage =
CustomControls.Properties.Resources.github;
        }
    }
}
```

运行窗体，界面如图 4.45 所示：

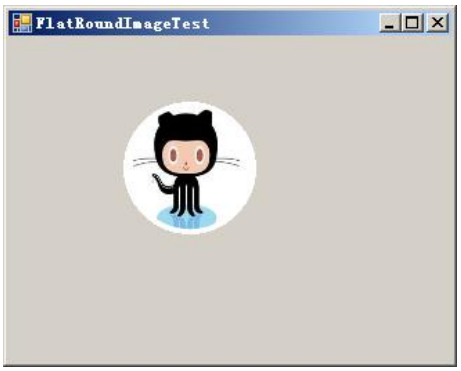


图 4.45 FlatRoundImage 控件示例

4.12 FlatDateTimePicker 控件

在管理信息系统开发过程中，有很多业务单据都需要时间信息，这就需要用到日期控件，这一节我们就自定义一个日期控件 FlatDateTimePicker，从 UI 上美化原生的 DateTimePicker 控件。

4.12.1 控件功能

FlatDateTimePicker 控件的核心功能是：具备 DateTimePicker 的日期选择功能，同时界面上进行美化。

4.12.2 控件设计

根据 FlatDateTimePicker 控件的功能需求，其 UI 可以设计如图 4.46 所示：

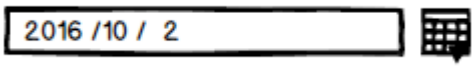


图 4.46 FlatDateTimePicker 控件 UI

4.12.3 控件开发

首先在 VS 中创建一个名为 FlatDateTimePicker 的组件，它继承自 System.Windows.Forms.DateTimePicker，注意此控件绘制日期选择器的图标时，用的是 FontAwesome 字体下的图标，运行该控件之前，请保证运行环境中正确安装该字体库，代码如下。

```
using System;
using System.ComponentModel;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Text;
using System.Windows.Forms;
namespace MingForm.Controls
{
    public class FlatDateTimePicker : DateTimePicker
    {
        private int W;
        private int H;
        private int x;
        private int y;
        private MouseState State = MouseState.None;
        protected override void OnMouseDown(MouseEventArgs e)
        {
            base.OnMouseDown(e);
            State = MouseState.Down;
            Invalidate();
        }
        protected override void OnMouseUp(MouseEventArgs e)
        {
            base.OnMouseUp(e);
            State = MouseState.Over;
            Invalidate();
        }
        protected override void OnMouseEnter(EventArgs e)
        {
            base.OnMouseEnter(e);
            State = MouseState.Over;
            Invalidate();
        }
        protected override void OnMouseLeave(EventArgs e)
        {
            base.OnMouseLeave(e);
            State = MouseState.None;
            Invalidate();
        }
        protected override void OnMouseMove(MouseEventArgs e)
        {
            base.OnMouseMove(e);
            x = e.Location.X;
            y = e.Location.Y;
```

```
            Invalidate();
            if (e.X < Width - 41)
                Cursor = Cursors.IBeam;
            else
                Cursor = Cursors.Hand;
        }
        protected override void OnClick(EventArgs e)
        {
            base.OnClick(e);
            Invalidate();
        }
        [Category("Colors")]
        public Color HoverColor
        {
            get { return _HoverColor; }
            set { _HoverColor = value; }
        }
        protected override void OnResize(EventArgs e)
        {
            base.OnResize(e);
            Height = 30;
        }
        private Color _BaseColor = Color.FromArgb(25, 27, 29);
        private Color _BGColor = Color.FromArgb(45, 47, 49);
        private Color _HoverColor = Color.FromArgb(35, 168, 109);
       public FlatDateTimePicker()
        {
            SetStyle(ControlStyles.AllPaintingInWmPaint |
ControlStyles.UserPaint | ControlStyles.ResizeRedraw |
ControlStyles.OptimizedDoubleBuffer, true);
            DoubleBuffered = true;
            BackColor = Color.FromArgb(45, 45, 48);
            ForeColor = Color.White;
            Cursor = Cursors.Hand;
            Font = new Font("Segoe UI", 8, FontStyle.Regular);
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            Bitmap B = new Bitmap(Width, Height);
            Graphics G = Graphics.FromImage(B);
            W = Width;
            H = Height;
            Rectangle Base = new Rectangle(0, 0, W, H);
            Rectangle Button = new Rectangle(Convert.ToInt32(W - 26), 0, W, H);
```

```
            GraphicsPath GP = new GraphicsPath();
            GraphicsPath GP2 = new GraphicsPath();
            var _withG = G;
            _withG.Clear(Color.FromArgb(45, 45, 48));
            _withG.SmoothingMode = SmoothingMode.HighQuality;
            _withG.PixelOffsetMode = PixelOffsetMode.HighQuality;
            _withG.TextRenderingHint = TextRenderingHint.ClearTypeGridFit;
            _withG.FillRectangle(new SolidBrush(_BGColor), Base);
            GP.Reset();
            GP.AddRectangle(Button);
            _withG.SetClip(GP);
            _withG.FillRectangle(new SolidBrush(_BaseColor), Button);
            _withG.ResetClip();
          //日期后边的图标
          _withG.DrawString("\uF022",new
Font("FontAwesome",16),Brushes.White,new PointF(W -26, 1));
            //绘制日期文本信息
            _withG.DrawString(Text, Font, Brushes.White, new Point(4, 3),
GDIHelpers.NearSF);
            G.Dispose();
            e.Graphics.InterpolationMode = InterpolationMode.HighQualityBicubic;
            e.Graphics.DrawImageUnscaled(B, 0, 0);
            B.Dispose();
        }
    }
}
```

4.12.4 控件应用

编译后， Visual Studio IDE 可以自动生成一个控件 FlatDateTimePicker，新创建一个窗体 FlatDateTimePickerTest，从工具箱中将 FlatDateTimePicker 控件拖放到窗体上，默认控件显示的为当前日期,如图 4.47 所示。

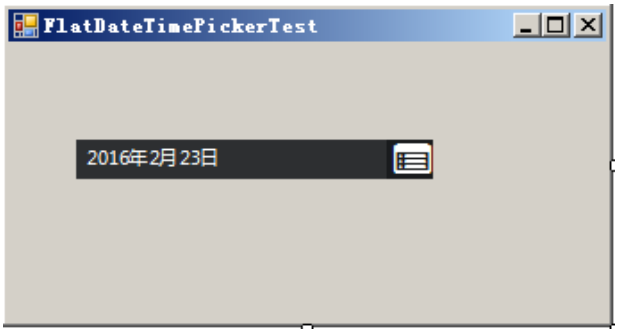


图 4.47 FlatDateTimePicker 控件示例

在 FlatDateTimePickerTest 窗体中编写如下代码：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace CustomControls
{
    public partial class FlatDateTimePickerTest : Form
    {
        public FlatDateTimePickerTest()
        {
            InitializeComponent();
        }
        private void FlatDateTimePickerTest_Load(object sender, EventArgs e)
        {
            this.flatDateTimePicker1.Value =
                  DateTime.Now.Add(new TimeSpan(2, 0, 0, 0));
        }
    }
}
```

运行该窗体，界面如图 4.48 所示：

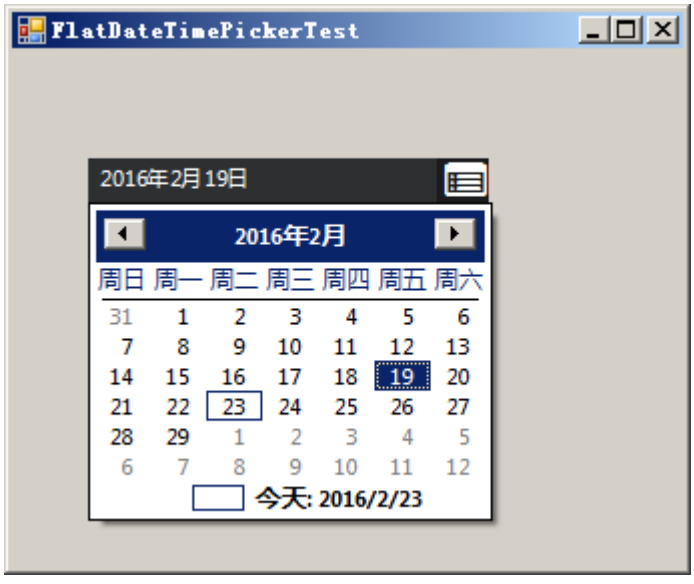


图 4.48 FlatDateTimePicker 控件示例

4.13 FlatDataGridView

日常的管理信息系统中，有很多的表格信息需要显示，这里我们通过自定义样式的方式来定制原生的 DataGridView 控件，让它看起来更加美观，并进行扁平化处理。

4.13.1 控件功能

FlatDataGridView 控件的核心功能是：具备 DataGridView 控件的功能，界面 UI 扁平化。

4.13.2 控件设计

根据 FlatDataGridView 的功能要求，其 UI 设计如图 4.49 所示：

Name (job title)	Age	Nickname	Employee
Giacomo Guilizzoni Founder & CEO	37	Peldi	◉
Marco Botton Tuttofare	34		☑
Mariah Maclachlan Better Half	37	Patata	⊟
Valerie Liberty Head Chef	:)	Val	☑
Guido Jack Guilizzoni	6	The Guids	☐

图 4.49 FlatDataGridView 控件示例

4.13.3 控件开发

首先在 Visual Studio IDE 中创建一个名为 FlatDataGridView 的控件，它继承自 System.Windows.Forms.DataGridView，代码如下。

```
using System;
using System.ComponentModel;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Text;
using System.Windows.Forms;
namespace Ming.Forms.Controls
```

```
{
    public class FlatDataGridView : DataGridView
    {
       public FlatDataGridView()
        {
            DoubleBuffered = true;
          SetFlatStyle(this);
        }
       private void SetFlatStyle(DataGridView dgv)
       {
          System.Drawing.Color _BackColor = Color.FromArgb(60, 70, 73);
          Color SuccessColor = Color.FromArgb(60, 85, 79);
          Color SuccessText = Color.FromArgb(35, 169, 110);
          Color ErrorColor = Color.FromArgb(87, 71, 71);
          Color ErrorText = Color.FromArgb(254, 142, 122);
          Color InfoColor = Color.FromArgb(70, 91, 94);
          Color InfoText = Color.FromArgb(97, 185, 186);
          #region DataGridVeiw Style
          DataGridViewCellStyle dataGridViewRowStyle = new
DataGridViewCellStyle();
          DataGridViewCellStyle dataGridViewHeadersStyle = new
DataGridViewCellStyle();
          dataGridViewRowStyle.Padding = new Padding(2);
          dataGridViewRowStyle.BackColor = SuccessColor;
          dataGridViewRowStyle.Font = new Font("Segoe UI", 10F,
FontStyle.Regular);
          dataGridViewRowStyle.ForeColor = Color.White;
          dataGridViewRowStyle.SelectionBackColor = SuccessText;
          dataGridViewRowStyle.SelectionForeColor = InfoText;
          dataGridViewRowStyle.Alignment =
              DataGridViewContentAlignment.MiddleCenter;
          dgv.AlternatingRowsDefaultCellStyle = dataGridViewRowStyle;
          dgv.RowsDefaultCellStyle = dataGridViewRowStyle;
          dgv.BackgroundColor = SuccessColor;
          dgv.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D;
          dgv.ColumnHeadersBorderStyle = DataGridViewHeaderBorderStyle.Single;
          dataGridViewHeadersStyle.Alignment =
              DataGridViewContentAlignment.MiddleCenter;
          dataGridViewHeadersStyle.BackColor = _BackColor;
          dataGridViewHeadersStyle.Font = new Font("Segoe UI", 10F,
FontStyle.Bold);
          dataGridViewHeadersStyle.ForeColor = Color.White;
          dataGridViewHeadersStyle.SelectionBackColor = SuccessText;
          dataGridViewHeadersStyle.SelectionForeColor = InfoText;
```

```
            dgv.ColumnHeadersHeight = 46;
            dgv.ColumnHeadersDefaultCellStyle = dataGridViewHeadersStyle;
            dgv.ColumnHeadersHeightSizeMode =
DataGridViewColumnHeadersHeightSizeMode.EnableResizing;
            dgv.EnableHeadersVisualStyles = false;
            dgv.GridColor = _BackColor;
            dgv.RowHeadersVisible = false;
            dgv.RowTemplate.Height = 23;
            #endregion
        }
    }
}
```

4.13.4 控件应用

编译后，Visual Studio IDE 可以自动生成一个 FlatDataGridView 控件，新创建一个窗体 FlatDataGridViewTest，从工具箱中将 FlatDataGridView 控件拖放到窗体上，窗体代码如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace MingForm.Controls
{
    public partial class FlatDataGridViewTest: Form
    {
        public Form5()
        {
            InitializeComponent();
        }
        private void FlatDataGridViewTest_Load(object sender, EventArgs e)
        {
            BindData(this.flatDataGridView1);
        }
        private void BindData(Ming.Forms.Controls.FlatDataGridView dgv)
        {
            DataTable dt = new DataTable();
```

```
            dt.Columns.Add("ID", typeof(string));
            dt.Columns.Add("UserName", typeof(string));
            dt.Columns.Add("Detail", typeof(string));
            for (int i = 0; i < 6; i++)
            {
                dt.Rows.Add(new object[] { "ID" + i.ToString(), "Name" +
i.ToString(), "Details" + i.ToString() });
            }
            dgv.DataSource = dt;
        }
    }
}
```

运行窗体，界面如图 4.50 所示：

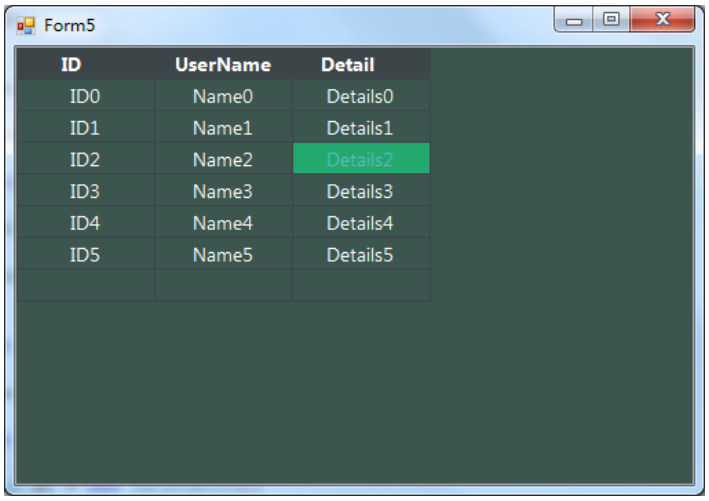


图 4.50 ImageButton 控件示例

4.14 本章小结

本章重点介绍了三种类型控件的开发，基本都给出了完整的代码，用兴趣的读者可以自行实现。其中大部分的控件都是 Flat 样式的，其中也借鉴了 FlatUI 控件的设计元素。通过这些控件的介绍，应该对一些基本的控件开发有一定的了解和掌握，但是生产环境下的控件开发往往非常复杂，大部分需要和数据库进行交互，例如多表头的 DataGridView 控件，能够下来出现多列的 ComBoBOx 控件，能够用于工程管理的甘特图控件，以及能够继续 Excel 公式计算的 SpreadSheet 控件等。

第五章 Form 高级主题

前面章节几乎都是关于控件开发的相关知识，其实 C#中还有很多其他很重要的内容，例如和数据库交互，如何构建动态属性以及如何扩展已有类的方法等。下面从数据库交互、反射、插件机制、动态属性以及 C#闭包和动态编译等角度来阐述 C#。

5.1 数据库交互

C#利用 SqlDataAdapte 对 DataTable 进行批量数据操作,可以让我们大大简化操作数据的代码量,我们几乎不需要循环和不关心用户到底是新增还是修改,更不用编写新增和修改以及删除的 SQL 语句,适配器都帮我们在后台进行了很好的处理.

如果您要通过 SQL Server 存储过程使用 DataAdapter 来编辑或删除数据，请确保不要在存储过程定义中使用 SET NOCOUNT ON。这将使返回的受影响的行数为零，DataAdapter 会将其解释为并发冲突。在许多情况下，以何种顺序向数据源发送通过 DataSet 所做的更改是非常重要的。例如，如果更新了现有行的主键值，并且添加了以新主键值作为外键的新行，则务必要在处理插入之前处理更新。可以使用 DataTable 的 Select 方法来返回仅引用具有特定 RowState 的 DataRow 数组。然后可以将返回的 DataRow 数组传递给 DataAdapter 的 Update 方法来处理已修改的行。通过指定要更新的行的子集，可以控制处理插入、更新和删除的顺序。以下代码确保首先处理表中已删除的行，然后处理已更新的行，然后处理已插入的行。

```
DataTable table = dataSet.Tables["Customers"];
//第一步处理删除.
adapter.Update(table.Select(null, null, DataViewRowState.Deleted));
//接着处理更新
adapter.Update(table.Select(null, null,
    DataViewRowState.ModifiedCurrent));
//最后处理新增
adapter.Update(table.Select(null, null, DataViewRowState.Added));
```

注意
对 DataSet、DataTable 或 DataRow 调用 AcceptChanges 将导致 DataRow 的所有 Original 值被 DataRow 的 Current 值覆盖。如果修改了唯一标识该行的字段值，则在调用 AcceptChanges 后，Original 值将不再匹配数据源中的值。在调用 DataAdapter 的 Update 方法

期间会对每一行自动调用 AcceptChanges。在调用 Update 方法期间，通过先将 DataAdapter 的 AcceptChangesDuringUpdate 属性设置为 false，或为 RowUpdated 事件创建一个事件处理程序并将 Status 设置为 SkipCurrentRow，可以保留原始值。(来自 MSDN)

5.1.1 强类型数据集

数据库的表设计好后,可以用 Visual Studio 连接数据库,并将 myUser 表拖放到数据集设计器中(重命名为 dsUser),默认情况下,数据集会自动生成 Fill 方法,我们这里需要手动进行删除,只保留表结构即可,数据查询和操作逻辑我们进行自定义，如下图 5.1 所示:

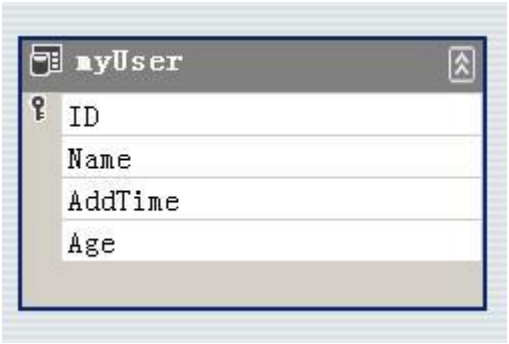


图 5.1 强类型数据集

5.1.2 UI 数据绑定

新建一个窗体,编译后可以从工具栏将上一步新建的 dsUser 强类型数据集拖放到窗体上,另外添加一个 BindingSource 控件(主要用于对 DataGridView 中的数据库和 DataSet 的数据进行绑定)

BindingSource 中选择数据源 DataSource 为 dsUser,然后选择数据集中的一个表 myUser 进行绑定，如图 5.2 所示:

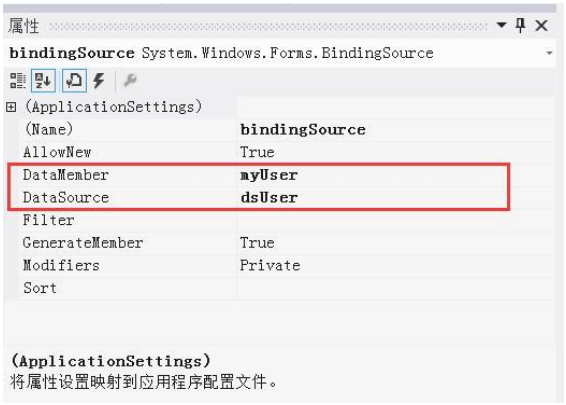


图 5.2 数据绑定

设置 DataGridView 的数据源为 bindingSource，这样通过 bindingSource1 就实现了 DataGridView 和 dsUser 的数据绑定，也就是说在界面 DataGridView 上进行操作，可以同步到 dsUser 中(会在每行上打上标志)

5.1.3 数据操作方法

可以利用 SqlDataAdapter 批量更新 DataTable 中的数据库，也支持表格中同时进行了删除/修改和新增的操作，下面的代码中，用 select * from {0} where 1=2 查询语句为 SqlDataAdapter 提供架构：

```
//生成架构
string selectSQL = string.Format("select * from {0} where 1=2", dt.TableName);
SqlDataAdapter sda = new SqlDataAdapter(selectSQL, ConnectionString)
```

而后用 SqlCommandBuilder 自动构建新增/删除/修改的命令，这样就可以用 sda.Update()方法对数据表进行批量操作了：

```
SqlCommandBuilder scb  =  new SqlCommandBuilder(sda);
sda.Update(dt);
public static void UpdateDataSet(DataSet ds, string tableName)
{
    try
    {
        if (tableName == "")
        {
            throw new ArgumentNullException("tableName 不能为空");
        }
        //生成架构
        string selectSQL = string.Format("select * from {0} where 1=2", tableName);
        using (SqlDataAdapter sda = new SqlDataAdapter(selectSQL,
ConnectionString))
```

```
        {
            SqlCommandBuilder scb = new SqlCommandBuilder(sda);
            sda.UpdateCommand = scb.GetUpdateCommand();
            sda.InsertCommand = scb.GetInsertCommand();
            sda.DeleteCommand = scb.GetDeleteCommand();
            sda.Update(ds, tableName);
            ds.Tables[tableName].AcceptChanges();
            DisposeCommand(sda.UpdateCommand);
            DisposeCommand(sda.InsertCommand);
            DisposeCommand(sda.DeleteCommand);
            sda.Dispose();
            scb.Dispose();
        }
    }
    catch (SqlException e)
    {
        throw e;
    }
}
```

为了确保在用户编辑完成后,立刻将数据同步到数据集中,必须调用 bindingSource 的 EndEdit 方法来同步数据.

```
private void dataGridView1_CellEndEdit(object sender,
DataGridViewCellEventArgs )
{
   this.bindingSource.EndEdit();
}
```

在窗体加载时,调用刷新方法,然后程序会自动将数据绑定到 DataGridView 上:

```
 private bool fnRefresh()
 {
     string sql = string.Format("select * from myUser");
     SqlHelper.GetDataTableBySQL(this.dsUser.myUser, sql);
     return true;
 }
```

我们可以通过 DataGridView 的 EndEdit 方法来获取当前用户是否已经结束编辑:

```
bool isEnd= this.dataGridView1.EndEdit();
```

另外可以通过 GetChanges()方法来获取修改的数据,返回一个 DataTable.

```
DataTable dtModify = this.dsUser.myUser.GetChanges(DataRowState.Modified);
```

调用 UpdateTable 方法,即可对表格进行批量处理(这里就一个表,如果是多个表,那么需要指明表名):

```
private bool Save()
{
```

```
    try
    {
        DataAccess.SqlHelper.UpdateDataSet(this.dsUser, "myUser");
        this.dsUser.myUser.AcceptChanges();//提交数据库
        return true;
    }
    catch (Exception ex)
    {
        return false;
    }

}
```

当用户选择一个单元格时，会获取到当前行的索引，然后调用 RemoveAt()方法进行删除，由于用 bindingSource 进行了绑定，可以直接调用 save()方法进行数据保存：

```
private void bindingNavigatorDeleteItem_Click(object sender, EventArgs e)
{
    if (this.dataGridView1.CurrentRow != null)
    {
        int index = this.dataGridView1.CurrentRow.Index;
        if (index > -1)
        {
            this.dataGridView1.Rows.RemoveAt(index);
            Save();
        }
    }
}
```

当用户单击新增按钮时，我们创建一个强类型的 myUserRow，然后给它附上默认值（特别是主键）

```
//add row
private void toolStripButton1_Click(object sender, EventArgs e)
{
    Common.dsUser.myUserRow dr = this.dsUser.myUser.NewmyUserRow();
    dr["ID"] = System.Guid.NewGuid().ToString();
    dr["Name"] = "Name";
    dr["AddTime"] = DateTime.Now.ToString();
    this.dsUser.myUser.AddmyUserRow(dr);
}
```

5.2 反射

反射，一种计算机处理方式。是程序可以访问、检测和修改它本身状态或行为的一种能力。程序集包含模块，而模块包含类型，类型又包含成员。反射则提供了封装程序集、模块和类型的对象。您可以使用反射动态地创建类型的实例，将类型绑定到现有对象，或从现有对象中获取类型。然后，可以调用类型的方法或访问其字段和属性。

表 5.1 给出了 System.reflection 命名空间包含的几个核心类，通过这些核心类可以实现反射的强大功能。

表 5.1 System.reflection 命名空间包含的几个类：

类	说明
Binder	从候选者列表中选择一个成员，并执行实参类型到形参类型的类型转换。
ConstructorInfo	发现类构造函数的属性并提供对构造函数元数据的访问权。
CustomAttributeData	提供对加载到只反射上下文中的程序集、模块、类型、成员和参数的自定义特性数据的访问。
CustomAttributeExtensions	包含检索自定义特性的静态方法。
EventInfo	发现事件的属性并提供对事件元数据的访问权。
FieldInfo	发现字段属性并提供对字段元数据的访问权。
LocalVariableInfo	发现局部变量的属性并提供对局部变量元数据的访问。
ManifestResourceInfo	提供对清单资源的访问，这些资源是描述应用程序依赖项的 XML 文件。
MemberInfo	获取有关成员属性的信息并提供对成员元数据的访问。
MethodBody	提供对用于方法体的元数据和 MSIL 的访问。
MethodInfo	发现方法的属性并提供对方法元数据的访问。
Module	在模块上执行反射。
ParameterInfo	发现参数属性并提供对参数元数据的访问。
PropertyInfo	发现属性 (Property) 的属性 (Attribute) 并提供对属性 (Property) 元数据的访问。
StrongNameKeyPair	封装对公钥或私钥对的访问，该公钥或私钥对用于为强名称程序集创建签名。

为了对反射有直观且明确的认识，下面我们通过一个例子来说明反射技术如何动态调用类的方法：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;
```

```
namespace ReflectionDemo
{
    public class TestClass
    {
        public string Add(int i, int j)
        {
            return (i + j).ToString();
        }
    }
    public static class ReflectionHelper
    {
        public static string RunByReflection()
        {
            //类库一般为dll
            //string ModuleName = "ReflectionDemo.dll";
            //此处为窗体应用程序,为exe
            string ModuleName = "ReflectionDemo.exe";
            string TypeName = "TestClass";
            string MethodName = "Add";
            string res = "";
            Assembly myAssembly = Assembly.LoadFrom(ModuleName);
            BindingFlags flags = (BindingFlags.NonPublic |
             BindingFlags.Public |BindingFlags.Static |
             BindingFlags.Instance | BindingFlags.DeclaredOnly);
            //获取方法列表
            Module[] myModules = myAssembly.GetModules();
            foreach (Module Mo in myModules)
            {
                if (Mo.Name == ModuleName)
                {
                    Type[] myTypes = Mo.GetTypes();
                    foreach (Type Ty in myTypes)
                    {
                        //类为TestClass
                        if (Ty.Name == TypeName)
                        {
                            MethodInfo[] myMethodInfo = Ty.GetMethods(flags);
                            foreach (MethodInfo Mi in myMethodInfo)
                            {
                                //string Add(int i, int j) 方法
                                if (Mi.Name == MethodName)
                                {
                                    Object obj = Activator.CreateInstance(Ty);
                                    Object o = Mi.Invoke(obj,new object[]{1,2});
```

```
                                res = o.ToString();
                            }
                        }
                    }
                }
            }
            return res;
        }
    }
}
```

当定义好 ReflectionHelper 静态类和 TestClass 类，下面就可以创建一个窗体来测试如何用反射进行动态的函数调用了，如果正确的话，应该返回结果为 3。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace ReflectionDemo
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            //3
            string res = ReflectionHelper.RunByReflection();
        }
    }
}
```

5.3 插件机制

在很多管理信息系统的开发过程中，有一些需求是经常变化的（例如界面布局，业务处理逻辑等）。但现实情况是，我们在编写程序时也很难做到面面俱到，考虑到一切的可能性。如果客户或者用户有新的需求，采用传统技术的话我们不得不重新修改代码并重新编译打包后发布。如果我们的产品有很多客户再用，而且同一个模块的业务操作完全不同，我们就不

得不在同一个业务模块的窗体代码中判断当前的客户，然后根据特定客户才处理特定的业务逻辑。

当我们编写完成测试通过后，提出需求的客户好像问题是解决了，但是可能会由于该模块小的改动，导致其他模块或者其他客户更新程序后出现奇葩的问题。

如果我们做一个基础的业务模块和平台级产品，对于差异的模块我们可以通过插件的机制来实现按照客户进行代码隔离，这样是不是可以保证产品的灵活性和可扩展性，另一方面减少统一修改发布更新包导致其他客户或者模块无法正常使用的副作用。

5.3.1 定义接口

对于插件机制来说，核心的是利用反射技术来动态加载和实例化类，但是一个管理信息系统或者软件产品里面有大量的类信息，必须要约定一个契约，来说明哪些是可以通过插件进行集成的，因此一般的差距机制必须要先定义一个接口来作为契约，当符合契约的话，即进行插件的加载和处理。下面我们先定义一个 ICMFormPlugIn 接口，其中有两个方法，一个用于显示窗体，一个负责关闭窗体。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace CM.Products.WinUI.PlugIn
{
    public interface ICMFormPlugIn
    {
        //Form
        void Show();
        void Close();
    }
}
```

5.3.2 插件加载

对于插件机制来说，核心的是利用反射技术来动态加载和实例化类，但是一个管理信息系统或者软件产品里面有大量的类信息，必须要约定一个契约，来说明哪些是可以通过插件进行集成的，因此一般的差距机制必须要先定义一个接口来作为契约，当符合契约的话，即

进行插件的加载和处理。下面我们先定义一个 ICMFormPlugIn 接口，其中有两个方法，一个用于显示窗体，一个负责关闭窗体。

对于插件窗体，可以继承 ICMFormPlugIn 接口，编译完成后（*.dll）放入插件对应的文件夹（PlugIns）中,然后配置系统菜单（生产环境中，这些信息都存放在数据库中，然后将插件反射需要的核心数据绑定到菜单 TreeNode 上,当用户单击菜单，就可以通过反射技术加载 dll,并调用 ICMFormPlugIn 接口的 Show 方法显示窗体，我这里的插件机制如图 5.3 所示。

```
public static void CreatePlugInDockForm(string strPlugInSubFilePath, string
strFormFullName, string strAssemblyName)
{
    try
    {
        //项目的Assembly选项名称DockSample.dll 即 DockSample
       string path = strAssemblyName;
        //类的全路径名称=> “DockSample.Modules.frm班组日志管理”
        string name = strFormFullName;
        string currentDirectory =
        System.IO.Directory.GetCurrentDirectory();
       if (strPlugInSubFilePath == "")
       {
           currentDirectory = currentDirectory + "\\PlugIns";
       }
       else
       {
           //同样的dll,可以按照打上客户的名称,然后加载对应客户的dll,这样就可以防止冲突
           currentDirectory = currentDirectory + "\\PlugIns" + "\\" +
strPlugInSubFilePath;
       }
       string[] dllFilenames = System.IO.Directory.GetFiles(currentDirectory,
strAssemblyName + ".dll");
       if (dllFilenames.Length == 1)
       {
           Assembly asm = Assembly.LoadFrom(dllFilenames[0]);
           Form frm = (Form)asm.CreateInstance(strFormFullName, true);
           //实现WinUI.PlugIn.ICMFormPlugIn接口
           if (frm.GetType().GetInterface(
                typeof(WinUI.PlugIn.ICMFormPlugIn).FullName) != null)
           {
               frm.Show();
           }
           else
```

```
            {
                MessageBox.Show("没有实现接口");
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
```

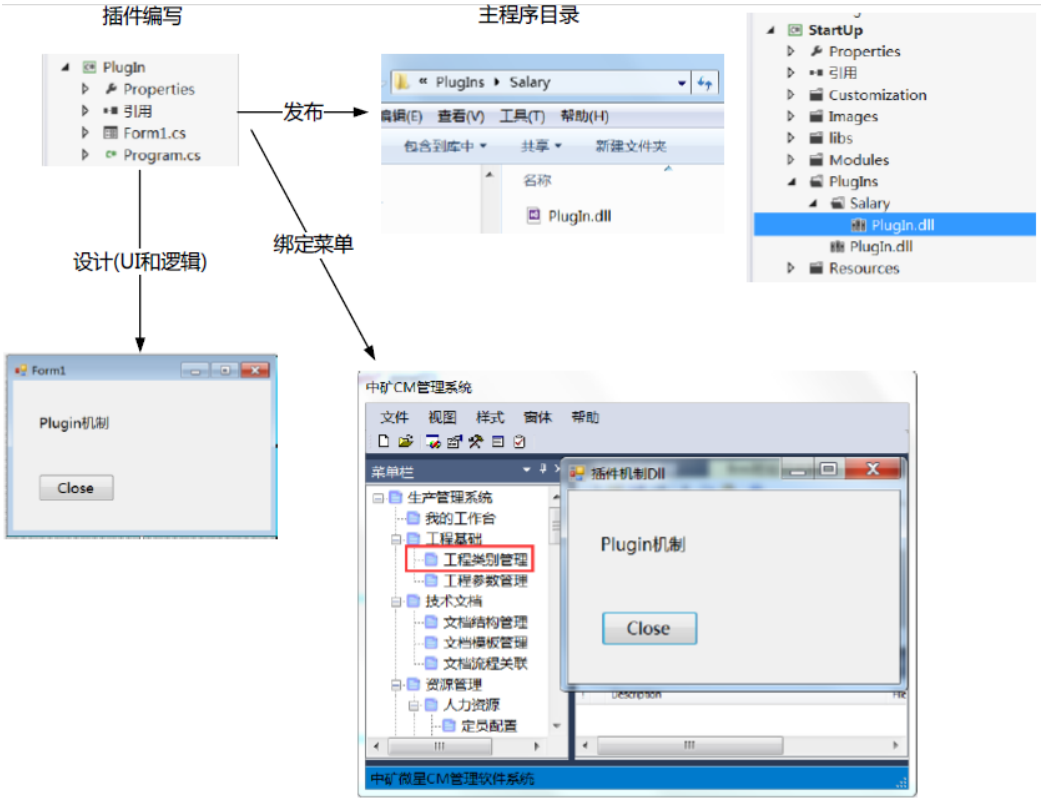


图 5.3 插件发布机制

```
//通过反射实例化出插件窗体
Modules.DynMenu.CreatePlugInDockForm("", "Plugin.Form1", "Plugin");
```

5.4 动态属性

JavaScript 是一门动态语言，可以动态的给对象添加属性和方法，非常方便．那么有没有一种方式可以让 C#也具备动态添加属性和方法的能力，像 Javascript 一样进行编程？下面就介绍一个很不错的框架 ClaySharp 可以实现上述功能．

下面的代码就是用 ClaySharp 构建一个 New 对象后，可以用 New 来动态添加属性：

```
//How to make C# Like JavaScript
```

```
var User = New.Person(
  FirstName: "Jack",
  LastName: "Wang"
);
```

为了让C#可以像JavaScript一样具备动态属性的能力，首先我们需要在项目中引入ClaySharp类库和它依赖的库。然后在控制台程序入口方法所在的类中要引入ClaySharp命名空间：

```
//ClaySharp可以让C#具备JavaScript的动态编程能力
 using ClaySharp;
```

用ClayFactory创建一个New对象，New对象后面可以随意的进行动态对象创建：

```
dynamic New = new ClayFactory();
var person = New.Person();
person["FirstName"] = "Louis";
person["LastName"] = "Dejardin";
person.Age = 26;
```

对象属性可以用[]或者.进行属性创建.当然也可以动态的创建方法：

```
var man = New.JackWang();
man.FirstName = "Jack";
man.LastName = "Wang";
man.SayFullName = new Func<string, string>(x => man.FirstName + man.LastName + x);
//output

Console.WriteLine(man.SayFullName()(" Here!"));
```

ClaySharp还支持用链的方法进行属性创建和赋值：

```
var peoples = New.Array(
    New.Person().FirstName("jack").LastName("wang"),
    New.Person().FirstName("jack").LastName("cumt")
);
Console.WriteLine("{0}", "----------ClaySharp-------------");
foreach (var p in peoples)
{
    Console.WriteLine("{0} {1}", p.FirstName, p.LastName);
}
```

其实在高版本的C#中，有内置的对象ExpandoObject也有类似的功能，只是功能稍微比ClaySharp弱一点：

```
Console.WriteLine("{0}", "-----System.Dynamic.ExpandoObject----");
//---------------ExpandoObject-----------------------------------
dynamic expObject = new System.Dynamic.ExpandoObject();
expObject.NickName = "jack";
//expObject["NickName"] = "jack"; //error
```

```
expObject.RealName = "Wang";
Console.WriteLine("{0} {1}", expObject.NickName, expObject.RealName);
Console.ReadLine();
```

运行该控制台程序，输出结果如图 5.4 所示：

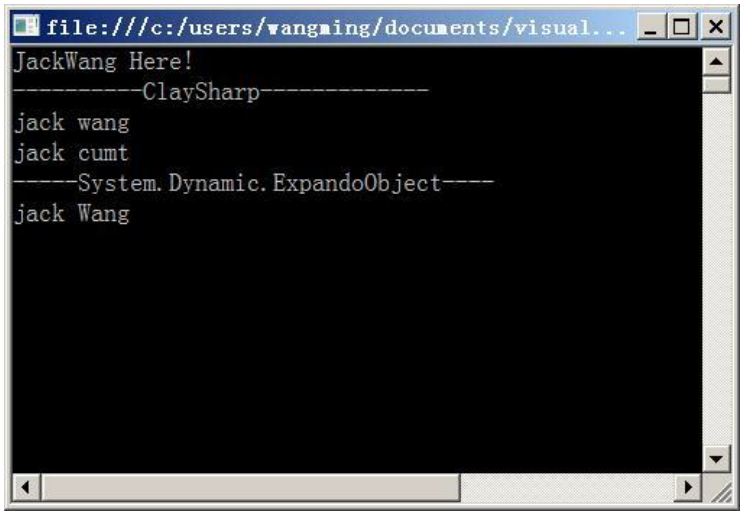


图 5.4 C#动态属性示例

5.6 C#闭包

JavaScript 中一个重要的概念就是闭包，闭包在 JavaScript 中有大量的应用，但是你知道么？C#也可以创建 Closure。下面就介绍一下如何在 C#中创建神奇的闭包。

在这之前，我们必须先知道如何在 C#中定义函数

```
//函数定义，参数为string，返回为string
Func<string, string> myFunc = delegate(string msg)
{
    return "Msg:" + msg;
};
```

利用 Lambda 表达式也可以简化上述的代码，但是效果一样：

```
//Lambda
Func<string, string> myFuncSame = msg => "Msg:" + msg;
```

定义好函数后，可以进行调用：

```
//函数调用
string message= myFuncSame("Hello world");
```

定义一个带外部变量（相对于内嵌函数而言）的嵌套函数，外部函数将内部嵌套的函数进行返回：

```
public static Func<int, int> Func()
{
    var myVar = 1;
    Func<int, int> inc = delegate(int var1)
    {
        //myVar能够记录上一次调用后的状态（值）
        myVar = myVar + 1;
        return var1 + myVar;
    };
    return inc;
}
```

C# Closure 调用如下所示：

```
static void CsharpClosures()
{
    var inc = Func();
    Console.WriteLine(inc(5));//7
    Console.WriteLine(inc(6));//9
}
```

当第二次调用 inc(6)时，此时函数内变量 myVar 并未像第一次调用函数时进行重新初始化（var myVar=1），而是保留了第一次运算的值,即 2，因此 inc(6)返回的结果为（2+1+6）=9

5.7 C#扩展方法

C#自带的 string.Format 可以格式化字符串，但是还是不太好用，由于格式的字符占位符都是数字，当数目较多时容易混淆。其实可以扩展 string 的方法，让 C#的字符串具备其他的方法，下面介绍一个实现类似的扩展方法。

```
String.jQueryStringFormat("hello $world", new {world="cnblog" })
```

5.7.1 变量前缀$

可以仿照 jQuery 中的选择器方法，用作为变量前缀。例如 Ilove$something 中的someting 就是变量，可以将 something 变量的值替换到字符串中。

```
//模板字符串前缀
private static readonly string __prefix = "$";
// $ 正则表达式 $name
    private static readonly Regex VariableRegex = new
Regex(@"\$(@{0,1}[a-zA-Z_\.0-9]+)");
```

5.7.2 正则表达式捕获变量

上面定义了变量的规则，必须是$打头的有效变量，下面将字符串用该正则表达式进行捕获

```
private static IEnumerable<string> GetEnumerateVariables(string s)
{
     var matchCollection = VariableRegex.Matches(s);

     for (int i = 0; i < matchCollection.Count; i++)
     {
        yield return matchCollection[i].Groups[1].Value;
     }
}
```

5.7.3 用反射获取属性的值

传入的对象含有各个属性，写一个方法获取指定属性的值

```
/// <summary>
/// 获取对象的对应属性值
/// </summary>
/// <param name="oValue">包含值的对象</param>
/// <param name="name">属性名</param>
/// <returns></returns>
private static object ValueForName(object oValue, string name)
{
    Type type = oValue.GetType();
    var property = type.GetProperty(name);
    if (property != null)
    {
        return property.GetValue(oValue, new object[0]);
    }

    var field = type.GetField(name);
    if (field != null)
```

```
    {
        return field.GetValue(oValue);
    }
    throw new FormatException("未找到命名参数: " + name);
}
```

5.7.4 string 方法扩展实现

基于上面的准备工作，下面我们定义 string 类的扩展方法 jQueryStringFormat，扩展方法都是用静态方法来实现，第一个参数必须是 this 待扩展的类 变量名：

```
public static string jQueryStringFormat(this String @this, string
sjQueryStringT, object oValue)
{

    //检测验证
    if (string.IsNullOrEmpty(sjQueryStringT))
        return sjQueryStringT;
    if (!sjQueryStringT.Contains(__prefix))
        throw new Exception("字符串中变量不包含$前缀");
    if (oValue == null)
        return sjQueryStringT;

    //解析
    //need  using System.Linq;
    var variables = GetEnumerateVariables(sjQueryStringT).ToArray();
    foreach (string vname in variables)
    {
        //获取值
        string vvalue = ValueForName(oValue, vname).ToString();
        //字符串替换
        sjQueryStringT = sjQueryStringT.Replace("$" + vname, vvalue);
    }
    return sjQueryStringT;
}
```

定义扩展方法完成以后，我们可以进行单元测试，看编写的扩展方法是否正确：

```
/// <summary>
///jQueryStringFormat 的测试
///</summary>
[TestMethod()]
public void jQueryStringFormatTest()
{
```

```
    string @this = ""; // TODO: 初始化为适当的值
    string Name = "JackWang";
    int ID = 100;
    string sjQueryStringT = "exec func($Name,$$ID)"; // TODO: 初始化为适当的值
    object oValue = new { ID, Name }; // TODO: 初始化为适当的值
    string expected = "exec func(JackWang,$100)"; // TODO: 初始化为适当的值
    string actual;
    actual = StringFormat.jQueryStringFormat(@this, sjQueryStringT, oValue);
    Assert.AreEqual(expected, actual);
    //Assert.Inconclusive("验证此测试方法的正确性。");
}
```

5.8 C#方法链

jQuery 的方法链使用起来非常方便，可以简化语句，让代码变得清晰简洁。那 C#的类方法能不能也实现类似的功能呢？基于这样的疑惑，研究了一下 jQuery 的源代码，发现就是需要方法链的函数方法最后返回对象本身即可。既然 javascript 可以，C#应该也是可以的。为了验证，编写一个 jQPerson 类，然后用方法链对其 ID，Name，Age 等属性进行设置，请看下面的代码：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace CSharpMethodLikeJQuery
{
    public class jQPerson
    {
        string Id { set; get; }
        string Name { set; get; }
        int Age { set; get; }
        string Sex { set; get; }
        string Info { set; get; }
        public jQPerson()
        {
        }
        /// <summary>
        /// 设置ID,返回this,即jQPerson实例
        /// </summary>
        /// <param name="Id"></param>
        /// <returns></returns>
```

```
public jQPerson setId(string Id)
{
    this.Id = Id;
    return this;
}
/// <summary>
/// 返回this,即jQPerson实例
/// </summary>
/// <param name="name"></param>
/// <returns></returns>
public jQPerson setName(string name)
{
    this.Name = name;
    return this;
}
/// <summary>
/// 返回this,即jQPerson实例
/// </summary>
/// <param name="age"></param>
/// <returns></returns>
public jQPerson setAge(int age)
{
    this.Age = age;
    return this;
}
/// <summary>
/// 返回this,即jQPerson实例
/// </summary>
/// <param name="sex"></param>
/// <returns></returns>
public jQPerson setSex(string sex)
{
    this.Sex = sex;
    return this;
}
/// <summary>
/// 返回this,即jQPerson实例
/// </summary>
/// <param name="info"></param>
/// <returns></returns>
public jQPerson setInfo(string info)
{
    this.Info = info;
    return this;
```

```
        }
        /// <summary>
        /// tostring输出键值对信息
        /// </summary>
        /// <returns></returns>
        public string toString()
        {
            return string.Format("Id:{0},Name:{1},Age:{2},Sex:{3},Info:{4}",
this.Id, this.Name, this.Age, this.Sex, this.Info);
        }
    }
}
```

然后可以对上面进行单元测试，看方法链是否生效：

```
/// <summary>
///toString 的测试
///</summary>
[TestMethod()]
public void toStringTest()
{
    jQPerson target = new jQPerson();
    target.setId("2")
            .setName("jack")
            .setAge(26)
            .setSex("man")
            .setInfo("ok");
    string expected = "Id:2,Name:jack,Age:26,Sex:man,Info:ok";
    string actual;
    actual = target.toString();
    Assert.AreEqual(expected, actual);
}
```

5.9 C#动态编译

前面介绍了 C#中的反射、动态属性、闭包和扩展方法，这一节介绍另一种比较高级的主题，就是 C#动态编译。前面也提到，如果客户的业务计算逻辑经常变化，而且计算公式需要能够动态配置，那么除了插件外，还有其他的解决方案么？其实动态编译技术对于处理动态变化的公式计算特别有用，下面我们就用动态编译技术来实现一个动态公式计算的示例：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Collections.ObjectModel;
namespace WpfApp_DynCalc
{
    /// <summary>
    /// MainWindow.xaml 的交互逻辑
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            ObservableCollection<Dynformula> ofs = new
ObservableCollection<Dynformula>();
            ofs.Add(new Dynformula {Zb = "A", Value = "1", Formula = "" });
            ofs.Add(new Dynformula { Zb = "B", Value = "2", Formula = "2*A+1" });
            ofs.Add(new Dynformula { Zb = "C", Value = "3", Formula = "B*B" });
            ofs.Add(new Dynformula { Zb = "D", Value = "4", Formula = "C-2" });
            ofs.Add(new Dynformula { Zb = "Z", Value = "5", Formula = "D+C" });
            this.dgrid.ItemsSource = ofs;
        }
        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
            this.lblResult.Text = "计算...";
             this.dgrid.ItemsSource=   DynCalc.CalcScript(ref this.dgrid);
             this.lblResult.Text = "计算完成!";
        }
    }
    public class Dynformula
    {
        private string zb;
        public string Zb
        {
            get { return zb; }
            set { zb = value; }
```

```
        }
        private string value;
        public string Value
        {
            get { return this.value; }
            set { this.value = value; }
        }
        private string formula;
        public string Formula
        {
            get { return formula; }
            set { formula = value; }
        }
    }
}
```

上面定义了一个 Dynformula 类，用于存储业务数据，同时再 WPF 窗体加载后，就将基础数据加载完成。下面我们定义一个 DynCalc 的静态类，该静态类首先必须要引入必要的命名控件：

```
using System.Collections.ObjectModel;
using System.Windows;
using System.Windows.Controls;
using System.Reflection;
using System.Globalization;
using Microsoft.CSharp;
using System.CodeDom;
using System.CodeDom.Compiler;
```

下面是 DynCalc 的静态类的全部代码：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace WpfApp_DynCalc
{
using System.Collections.ObjectModel;
using System.Windows;
using System.Windows.Controls;
using System.Reflection;
using System.Globalization;
using Microsoft.CSharp;
using System.CodeDom;
using System.CodeDom.Compiler;
```

```
   public static class DynCalc
    {
      public static ObservableCollection<Dynformula> CalcScript(ref DataGrid
dgrid)
      {

          CSharpCodeProvider objCSharpCodePrivoder =
            new CSharpCodeProvider();
          ICodeCompiler objICodeCompiler =
            objCSharpCodePrivoder.CreateCompiler();
          CompilerParameters objCompilerParameters =
             new CompilerParameters();
          objCompilerParameters.ReferencedAssemblies.Add("System.dll");
          objCompilerParameters.GenerateExecutable = false;
          objCompilerParameters.GenerateInMemory = true;
          CompilerResults cr =
          objICodeCompiler.CompileAssemblyFromSource(
          objCompilerParameters, GenerateCode(ref dgrid));
          if (cr.Errors.HasErrors)
          {
              Console.WriteLine("编译错误：");
              foreach (CompilerError err in cr.Errors)
              {
                  Console.WriteLine(err.ErrorText);
              }
              return null;
          }
          else
          {
              // 通过反射，调用实例
              Assembly objAssembly = cr.CompiledAssembly;
              object objDynCalc =
        objAssembly.CreateInstance("DynamicCodeGenerate.RunScript");
              ObservableCollection<Dynformula> ofsnew = new
ObservableCollection<Dynformula>();
              //循环datagrid进行公式计算并赋值
              for (int i = 0; i < dgrid.Items.Count; i++)
              {
                  Dynformula item = dgrid.Items[i] as Dynformula;
                  if (item == null)
                  {
                      break;
                  }
                  string zb = item.Zb;
```

```
            PropertyInfo pinfo = objDynCalc.GetType().GetProperty(zb);
            if (pinfo != null && pinfo.CanRead)                {
                //获取属性get值
                object obj_Name = pinfo.GetValue(objDynCalc, null);
                ofsnew.Add(new Dynformula {
                 Zb = item.Zb,
                 Value = obj_Name.ToString(),
                  Formula = item.Formula});
            }
        }
        return ofsnew;
    }
}
/// <summary>
/// 计算逻辑C#脚本动态构建
/// </summary>
/// <param name="dgrid">存有指标以及指标计算公式的datagrid</param>
/// <returns>C#脚本</returns>
 static string GenerateCode(ref DataGrid dgrid)
 {
    StringBuilder sb = new StringBuilder();
    StringBuilder sb构建函数内容 = new StringBuilder();
    sb.Append("using System;");
    sb.Append(Environment.NewLine);
    sb.Append("namespace DynamicCodeGenerate");
    sb.Append(Environment.NewLine);
    sb.Append("{");
    sb.Append(Environment.NewLine);
    sb.Append("    public class RunScript");
    sb.Append(Environment.NewLine);
    sb.Append("    {");
    //------------------------------------------------------------
    for(int i=0;i<dgrid.Items.Count;i++)
    {
        Dynformula item = dgrid.Items[i] as Dynformula;
        if (item == null)
        {
            break;
        }
        string zb = item.Zb;
        sb.Append(Environment.NewLine);
        sb.AppendFormat("       public double _{0};", item.Zb);
        sb.Append(Environment.NewLine);
        sb.AppendFormat("       public double {0}",item.Zb);
```

```
            sb.Append(Environment.NewLine);
            sb.Append("       {");
            sb.Append(Environment.NewLine);

            if (item.Formula.Trim() != "")
            {
                sb.Append("           set{ "+item.Zb+"=value;}" );
                sb.Append(Environment.NewLine);
                sb.Append("           get{return "+ item.Formula + ";}");
            }
            else
            {
                sb.Append("           set{ _" + item.Zb + "=value;}");
                sb.Append(Environment.NewLine);
                sb.Append("           get{return  _"+item.Zb+";} ");
                sb.Append(Environment.NewLine);
                sb构建函数内容.Append("      _" + item.Zb + "=" + item.Value);
            }
            sb.Append(Environment.NewLine);
            sb.Append("       }");
            sb.Append(Environment.NewLine);
        }
        //-------------------------------------------
        //构造函数进行赋值
        sb.Append(Environment.NewLine);
        sb.Append("       public  RunScript()");
        sb.Append(Environment.NewLine);
        sb.Append("       {");
        sb.Append(Environment.NewLine);
        sb.AppendFormat("           {0};",sb构建函数内容.ToString());
        sb.Append(Environment.NewLine);
        sb.Append("       }");
        sb.Append(Environment.NewLine);

        //-------------------------------------
        sb.Append(Environment.NewLine);
        sb.Append("       public string OutPut()");
        sb.Append(Environment.NewLine);
        sb.Append("       {");
        sb.Append(Environment.NewLine);
        sb.Append("           return \"Hello world!\";");
        sb.Append(Environment.NewLine);
        sb.Append("       }");
        //-----------------------------------------
```

```
            sb.Append(Environment.NewLine);
            sb.Append("    }");
            sb.Append(Environment.NewLine);
            sb.Append("}");
            //输出代码拷贝到vs中，可以发现代码的异常，辅助调试
            string code = sb.ToString();
            Console.WriteLine(code);
            return code;
        }
    }
}
```

运行程序，其中值列的值为初始值，点击计算后会根据公式列配置进行动态计算，初始化界面如下，点击计算后程序会根据公式的设置进行值运算界面如图 5.5 所示。

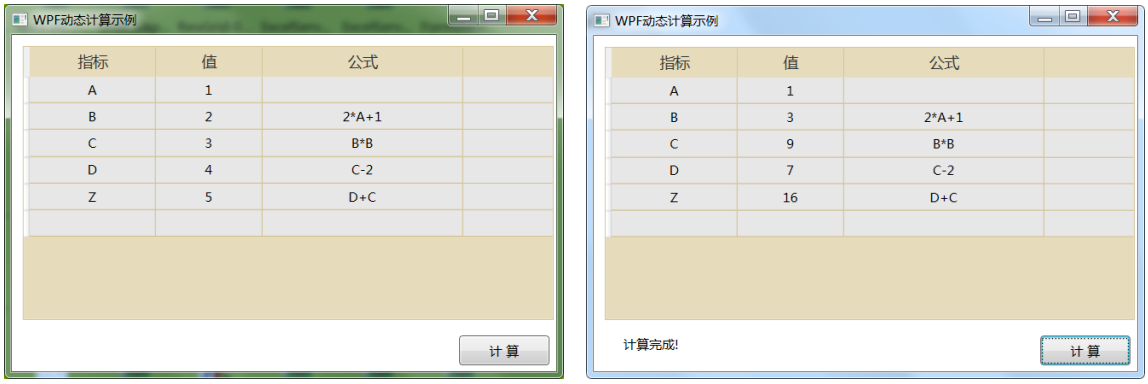


图 5.5 C#动态编译示例

5.10 本章小结

本章作为最后一章，介绍了控件开发之外的 C#主题，其中包括数据库的交互，反射机制、插件机制、动态属性、C#闭包，C#扩展方法，C#方法链和动态编译技术。虽然本章都是这些主题的入门知识，只是抛砖迎玉，但是如果读者感兴趣，可以在这些主题上进行深入学习，这对于提升编程能力将大有裨益。