Lewolf
保留版权
签名未验证

第二篇 应用 C++ Builder 的面向状态

第五章 使用 C++ Builder

前面几章讲述了面向状态的基本思想和 C++ Builder 的语言特征，侧重于知识性，从这一章开始，我们将要针对应用 C++ Builder 来介绍如何使用面向状态。有一点还要声明，本书并不是 C++ Builder 的使用手册，因此本章并不对如何使用 C++ Builder 进行详细的讲解，而主要是就面向状态的一些使用方法进行讲述，读者需要有一定的使用基础和编程经验，否则阅读本章并无多大帮助。

面向状态的方法是源于组件技术的，本章要讲的也是在 VCL 组件设计中如何使用 C++ Builder 的面向状态，在应用程序中使用面向状态技术和组件设计中大同小异，而且更简单一些，能够设计组件的时候自然而然的能够在应用程序中使用面向状态的技术。

1 C++ Builder 中的命名约定

任何一种编程工具，都会有自己的命名规则或者叫命名约定，C++ Builder 也不例外，实际上，关于变量的名称和程序本身并没有多大关系，也不会产生实质性影响，但是采用一套好的命名规则对程序的设计会带来许多便利。

熟悉 Windows SDK 的读者都知道在 SDK 中采用了一种叫做“匈牙利命名规则”的约定，在这种约定中可以非常容易的通过变量的名称来判断变量属于那一种数据类型，比如 hWnd 是一个 HWND 类型的窗口句柄；lpText 是一个 LPCTSTR 类型，其含义是指向字符串的 long point 长指针，HWND 和 LPCTSTR 是在 Windows.h 中预定义的数据类型，其名称的含义也显而易见。

在 C++ Builder 中，Borland 公司提供了一种更加适合面向状态的命名规则，在这种规则里面，也许你并不能根据其名称判断数据或者属性是什么数据类型，但从其名称可以很容易的判断出其用途或者在对象中真正的含义。

C++ Builder 并不对如何命名方法或参数施加限制，但是遵守这一约定可以方便应用程序开发人员使用方法。你也可以完全不遵守 Borland 公司的命名规则，而在开发程序或者组件的时候使用一种自己熟悉的约定，但是从一致性来考虑，建议使用 Borland 在 VCL 中的命名规则是非常有必要的，为此首先要了解 C++ Builder 的命名规则。

C++ Builder 中使用名词、动词、动名词甚至是介词的组合方式来对一个变量、属性、方法等进行命名，使其具有一个非常明确的含义，比如 Active 表示的是一个 bool 类型的变量（属性），Enabled 同样，而 Show 则理解为是一个方法用于显示某些内容更为合理，同样 OnShow 则应该理解为显示时触发的一个事件，OnShowing 是显示的过程

中的一个事件。

显然这种命名规则对于某些对象和面向状态的程序设计更为适合一些，下面我们就 C++ Builder 中类、属性、事件和方法的命名规则作个简单介绍。

1.1 类的命名

在 VCL 中，类的命名最为简单，所有的 VCL 类库中除了 Exception 和其派生类之外，其类名都是以“T”开头的，象 TObject、TButton、TPicture 等，“T”后面跟随的是能够明确表示类用途的词汇或词汇组合。除了类以外，其它的数据类型也大多数是以 T 开头的比如，TColor 为 enum 类型，而在 Graphics.h 中预定义了部分 Color 的取值如：clBlack，代表黑色，clBlue 代表蓝色等，其中 cl 为 Color 的缩写，Blue 是蓝色。也有一些数据类型不是以“T”开头的比如 AnsiString 等，这些大多数是从 Object Pascal 转换过来的。

类实例的命名也非常简单，在 IDE 中 C++ Builder 会自动根据类名词为使用 IDE 创建的类实例命名，大多数类的实例都是根据创建的数量在类名称的后面加上序号并去掉前面的“T”,比如 TButton 的实例为 Button1、Button2 等，另外一些向 TMenuItem，则是根据 Caption 中的字符，去掉中间的空格再加上序号的，如果 Caption 中的字符不是 Ascii 字符，则 IDE 会自己对 MemuItem 进行命名。类实例的名称一般不需要自行定义也是可行的,但如果希望类实例更具有明显的含义,可以在 IDE 中修改类实例的名称，其变量的名称也会跟随改变，比如输入用户姓名的 TEdit 实例，可以为其取名 UserNameEdit 等。

1.2 属性的命名

在 C++ Builder 中，属性的类型允许为任意类型，可以是 C++语言中的基本数据类型，也可以是 VCL 类或其它任意用户定义的数据类型，而对于属性，不仅仅是属性显示的名称，可能还包括了内部存储数据和存取方法等名称，我们逐一了解。

1.2.1 属性名称

一般来说，属性的名称一个可以表达明确的含义，通过名称可以知道属性的功能或者描述的状态，而且需要尽可能的通过其名称来判断属性的数据类型，属性的名称通常有以下几种：

名词

由单个名词构成，这些名词往往是具有明确含义的词汇如：Count、Menu、Canvas、Designer、Position、Icon、Monitor、Parent、Child 等，象 Count 就很容易理解为数量，其类型应该是 int 类型，Menu、Canvas、Icon 等本身是类名的一部分，这些属性的类型应该是这些类的类型，Parent、Child 本身不是类名的一部分，基本数据类型也不能直接何其对应，这些属性的类型一个是和对象相关的类型，很可能和自身相同。

动词

直接使用单个动词的属性很少，因为单个动词看起来更象是一个方法，在 VCL 中也确实如此，动词往往是以被动形式出现在属性名称中，比如：Scaled，而被动形式也可以理解为形容词，因此这种属性和形容词作属性名称类似，多是 bool 类型的，Scaled 表示是否按照比例作某件事。

形容词

单个形容词作属性名称的是比较多的，象 Active、Floating、Visible、Enable 等，形容词所描述的状态往往是非此即彼的，因此这种属性的类型大多数是 **bool** 类型。

词汇组合

词汇组合是属性名称中使用最多的形式，因为单个词汇往往不能表达复杂的含义，为了更加明确属性的含义，使用词汇组合可以达到这个目的。

在词汇组合的属性中，属性的数据类型则可以根据被修饰的主要部分进行判断，比如 ActiveControl、ActiveMDIChild、ClientHandle、ClientRect、HelpFile，这几个属性前面词汇都是修饰用的，最后一个词才是主要部分，其类型可以通过最后一个词来判断。

词汇组合的属性名称中有一些是数据类型本身名称的一部分如：FormState、FormStyle 等，这些实际上和单个名词的情况是一样的。

另外一些词汇组合的是由动词和其它词汇构成的，比如：HasChild、HasParent、IsEmpty 等，这些名称看起来更象是方法，往往也是只读属性，这种情况既可以作为方法处理，也可以作为属性来处理，但我们建议当做属性来处理。

1.2.2 内部存储变量名称

前面也讲过，大多数属性在其内部都有各自的存储数据，来保存真实的数据信息，这些数据原则上都是私有数据，在 VCL 中内部数据的命名规则是属性名称前面加上“ F ”，这也是一个习惯问题，采用和 Borland 一致的命名约定，在设计程序和组件的时候可以不必费心的考虑某个属性是 VCL 固有的还是自己扩充的，这样以来在成员函数中引用内部数据的时候只需要在属性名称前面加上“ F ”就行。

1.2.3 存取函数的命名

对于属性，另外一个命名约定就是存取函数的名称，在 VCL 中所有的存取函数均是保护成员，通常并不需要显式调用，在引用属性时被隐含调用。

属性的读取函数的命名原则为属性名称前面加上“ Get ”，存储函数的名称为属性名称前面加上“ Set ”，这正好也是一个判断原则，如果你设计的方法名称中出现以“ Get ”或者“ Set ”开头的，就应该考虑是否将其改为属性。

1.3 事件的命名

事件本身就是一种属性，但又和一般属性有着不同之处，我们知道事件表示的是状态的改变，是一种瞬间状态，可以理解为“ 当……怎么样 ”的时候，或者“ 在……之前 ”等，因此事件往往是由介词“ On ”开头的，后面紧跟着相应的动作，通常是动词、形容词或词

汇组合,比如 OnCreate、OnShow、OnPaint 等,有一些事件是以‘ Before ’或者‘ After ”开头的，表示“ 在……之前 ”或“ 在……之后 ”。

事件的处理函数名称也非常简单，IDE 通常是将对象名称和事件名称组合在一起构成事件处理函数的名称，比如 Button1 的 OnClick 事件，在 IDE 中生成的处理函数的名称为：Button1Click；

1.4 方法的命名

方法的命名更加直观，通常可以由能够表达操作内容的单个动词构成，也可以是动词和其它词汇的组合。一般来说，方法是相对独立的一个操作或动作，其名称至少包含一个动词，如果名称中没有动词，则需要考虑是否应该作为属性来处理。方法的名称也可以是名称、形容词甚至是副词，这时，其名称表达的是该方法执行后的结果或者状态。

由单个动词构成的方法如 TForm 类中的：

Close、Hide、Print、Release 等等。

由其他类型的单个词汇构成的方法有：

Next、Cascade、Previous、Tile 等。

词汇组合为名称的方法更多，如：

ArrangeIcons 、 CloseQuery 、 SendCancelMode 、 DefocusControl 、 FocusControl、GetFormImage 等等。

一般来讲，方法的命名应该遵循以下原则：

具有描述性，能够确切表达动作的含义。

名称应能够反映出返回值的性质。尽管对于作为程序员的你而言，返回某个对象水平位置的函数命名为 X 是很自然的，但用 GetHorizontalPosition 作为名字更易理解。

如果为 void 返回类型，方法名称应该具有一个主动动词或者是动作执行后的结果。

尽管 C++ Builder 对变量、对象、类、属性、事件和方法的名称并没有限制，但是使用 C++ Builder 的命名规则会有助于更快更好的开发程序，了解这个约定对阅读后面的章节也有帮助。

2 创建属性

属性是面向状态中最重要的概念，属性体现着对象的状态，要使用面向状态的技术，就必须深刻的理解属性，并能够熟练的创建和应用属性。而且在组件中，可见部分大多数是属性。即使不使用面向状态的方法，也应该对属性有所了解。接下来的内容是在组件设计中如何创建属性及使用属性，在应用程序中创建和使用属性和这类似。

2.1 属性的必要性

前面讲过，属性就如同变量一样，程序员可以象使用变量一样对他们进行读取和设置，唯一与变量不同的地方是不能把属性作为参数按引用传递给方法。但是属性不仅提供了面

向状态的可能，而且比简单的数据成员更强有力的能力，这是因为：

- 应用程序员可以在设计时设置属性的值。与方法只能在运行时有效不同，属性使得应用程序员可以在应用程序运行之前定制组件。属性显示在对象观察器中，这简化了应用程序开发人员的工作。构造对象，你可以让 C++ Builder 从对象观察器中读取值，而不是使用几个参数。当属性值被设置时，对象观察器将进行验证。这一点是 C++ Builder 能实现可视化程序设计的关键。
- 属性可以隐藏实现细节。例如，内部被加密的数据在显示为属性值时将被解密。虽然，值只是简单的数字和字符，但组件却可能需要在数据库中查询或进行复杂的计算才能获得。属性使你可以把复杂的事情弄得看上去只是简单的设置，而看上去犹如设置数据成员的东西实际上却调用了精心编制的方法。
- 属性可以是虚拟的，因此，应用程序开发人员看上去差不多的属性实际上可能由不同的组件用不同的方式实现的。比如 Active 对不同的组件，实现的方法和效果都是不同的。
- 属性封装了实现过程，体现了对象的状态，一个简单的例子是所有控件都有的 Top 属性。设置 Top 属性值并不仅仅只是改变了内存中的值，而且会导致控件重定位与重画。
- 设置属性值引起的反应并不局限于单个组件，例如，设置一个快速按钮的 Down 属性为 true 会使得与它同组的其他快速按钮的 Down 属性被设为 false。

2.2 属性的类型

C++ Builder 中属性可以是任何类型的。不同类型的属性在对象观察器中的显示是不同的。设计时对象观察器在创建属性时将对属性的赋值进行验证，下表显示了不同类型的属性在对象察看器中显示的方式。

表 5.1 对象观察器如何显示属性属性类型对象观察器的处理

简单型（Sample）	数字、字符和字符串属性被显示为数字、字符和字符串。应用程序开发人员能直接编辑它们的值
属性类型对象观察器的处理枚举型（Enumerated）	枚举型（包括 bool 型）属性显示为可编辑的字符串。应用程序开发人员既可以通过双击值域，也可以通过下拉式列表框选择可能的值
集合型（Set）	集合型属性显示为集合（sets）。应用程序开发人员通过双击该属性象处理布尔值一样扩展、处理集合的每个元素
对象型（Object）	本身就是类的属性，自身就有自己属性的编辑器，由组件注册过程指定。如果作为属性的类有自己的发布部分，对象观察器允许应用程序开发人员扩展（通过双击）列表包含并编辑它们。对象属性必须从 TPersistent 派生而来
数组型（Array）	数组属性必须有自己的属性编辑器。对象观察器对它们的编辑没有内置支持。当你注册你的组件时，你能指定一个属性编辑器

2.3 发布继承的属性

所有组件都继承它们祖先的属性。当从已有的类派生组件时，新组件继承了直系祖先的所有属性。如果你从抽象类派生组件，它继承的属性可以是 public 或 protected，但没有 published。为了使 protected 或 public 属性在设计时可以在对象观察器中可见，需要把该属性重新声明为 published。重新声明意味着在派生类的声明中加入继承属性的声明。

例如，从 TWinControl 控件派生组件，继承它的 DockSite 保护属性。通过在新组件中重新声明 DockSite，能把 protected 改为 public 或 published。

下面的代码显示如何重新声明 DockSite 为 published，从而使它在设计时可见：

```
class PACKAGE TMyComponent : public TWinControl
{
__published:
    __property DockSite;
};
```

在重新声明属性时，只能指定属性名，而无需指定属性类型和其他在 2.4“定义属性”中出现的信息，但你能为该属性声明新的缺省值并指明是否存储该属性。重新声明属性可以降低属性的限制等级，但不能提高限制等级。因此，可以把 protected 改为 public，但不能把 public 属性声明为 protected。

2.4 定义属性

下面的内容将阐述如何声明新属性以及解释一些标准组件遵循的约定，主要包含以下几个方面：

2.4.1 属性声明

在组件类的声明中声明属性。声明属性的时候，有三件事情需要明确：属性名称、属性类型、读写属性值的方法。如果没有写方法，那么该属性为只读属性。

在设计时，对象观察器可以编辑声明在__published 定义域中的属性。已发布的属性值与组件一同保存于窗体文件中。public 部分声明的属性可以在运行时访问，由程序代码读取或设置。下面是 Count 的属性声明：

```
class PACKAGE TMyComponent : public TComponent
{
private:
    int FCount;                                    //存储属性的数据成员
    int __fastcall GetCount();                     //属性的读取方法
    void __fastcall SetCount( int ACount );        //属性的存储方法
```

```
public:
    __property int Count = {read=GetCount, write=SetCount};
                                                    //声明属性
     //...
};
```

2.4.2 内部数据存储

在前面的章节中我们已经介绍过，对于大多数属性都有一个用于存储数据的数据成员，该成员必须和属性是同一类型，但如何存储属性的数据是没有限制的，也就是说这个用于存储数据的成员可以是私有的、是受保护的、也可以是完全公开的。

一般的 C++ Builder 的组件对于属性内部存储数据遵循下列约定：

- 属性数据存储于数据成员中，这也是我们推荐的，大多数属性都将数据存储在数据成员中，也可以使用读取方法引用其它组件的属性。
- 存储属性的私有数据成员只有组件自身才能访问。派生组件应使用继承的属性，而不要直接访问属性内部的数据存储。这一点对于 C++ Builder 来说是非常不幸的，因为当父类的属性定义为只读属性的时候，其派生类没有办法对其内部数据进行修改，而在 Object Pascal 中派生类可以对父类的私有成员进行存取，Borland 公司在开发 VCL 的时候，并没有考虑到这两者之间的差异，从而给使用 C++ Builder 的程序员在组件设计上（多存在于对原有 VCL 组件功能的扩充时）带来了许多不如 Delphi 灵活的地方。
- 数据成员标识由 F 和紧接着的属性名组成，这就是我们前面说的命名规则。例如，TControl 中 Width 属性的原始数据存于 FWidth 数据成员中。

有个原则强调了上述约定：只有属性的方法可以访问属性的数据。如果某一方法或其他属性想改变属性的数据，它应遵循上述原则，而不要直接访问存储的数据。这保证了继承的属性可以改变而不会导致派生组件无效。实际上灵活性和语言的约束是相互联系和对立的，强的语言约束可以保证程序设计更少的出错，但灵活性差一些，灵活性提高的时候，却可能带来程序潜在错误的发生，在后面的例子中我们就使用了公开的成员作为属性的内部存储数据，目的是为了在其它对象能够访问和设置内部数据。

2.4.3 直接访问

使数据可以被访问的最简单方法就是直接访问。也就是说，属性声明中指定赋值和读取的 read 和 write 部分直接访问内部存储的数据，而不是通过某一访问方法做这些事。直接访问对那些在对象观察器中可见的、改变值不会立即触发处理的属性非常有用，但并不实用。通常，属性声明中的 read 部分会直接访问数据，但 write 部分则使用访问方法。当组件值被改变时，这点使得组件可以更新状态。

下面的组件类型声明显示了一个属性在 read 和 write 部分中使用直接访问。

```
class PACKAGE TSampleComponent : public TComponent
```

```
{
private:                              //内部存储数据是私有的
    bool FReadOnly;                   //声明私有的数据成员，用于存储属性
                                      // ...
__published:                          //__publish 使属性在设计时可见
    __property bool ReadOnly = {read=FReadOnly, write=FReadOnly};
};                                    //声明属性
```

这个属性只是提供了能够在 Object Inspector 中设置和修改，以及将设计时指定的数据存储在资源文件中的能力，对于资深的程序员来说毫无意义。

2.4.4 访问方法

可以在属性声明的 read 和 write 部分指定访问方法，而不是数据成员。访问方法应是 protected，而且通常声明为 virtual，允许派生类重载。不要把访问方法声明为 public，把它们声明为 protected，确保应用程序开发人员不会无意中通过它们修改了属性。

下面的例子就是上面提到的 Count 属性的访问方法，在这里 Count 同时具有 read 和 write 方法：

```
class PACKAGE TMyComponent : public TComponent
{
private:
    int FCount;
    int __fastcall GetCount();
    void __fastcall SetCount( int ACount );
public:
    __property int Count = {read=GetCount, write=SetCount};
     //...
};
```

下面是设置方法：

```
int __fastcall GetCount()
{
    return FCount;                    //直接返回存储的数据
}

void __fastcall SetCount( int ACount )
{
    if( FCount == Value )
      return;        //如果设置的值和原来的值相同，则直接返回，因为后面对于状态更
```

```
                    //新的代码可能会有较大的系统开销，或是不必要的操作。
    //  ...           需要针对 Count 属性的改变而需要更新的代码
    FCount = Value;  //更新内部存储数据
}
```

1．读方法

读方法通常都是无参数的函数（带有索引的属性或者数组属性除外），它返回与属性的类型相同的值。根据约定，函数的名字是 Get 后紧接属性名。例如，Count 属性的读方法为 GetCount。使用读取方法时，往往需要在读取函数对内部存储数据操作，生成适当类型的值。读方法一般没有参数，仅有的例外是数组和使用索引分类的属性。如果你没有声明读取部分，属性就是只写的。只写属性非常少见。

2．写方法

写方法是个只有一个参数的函数（带有索引的属性或者数组属性除外），它的参数类型与属性类型相同。参数可以按值传递，也可以按引用传递，对参数名也没有限制。根据约定，写方法的名字是 Set 后紧跟属性名称。

例如，Count 属性的写方法为 SetCount。传递的参数即是属性的新值。写方法在需要时将适当的数据存储于属性内部存储区。写方法一般只有一个参数，仅有的例外是数组和使用索引分类的属性，这两个属性以索引值作为第二个参数（使用索引分类创建一个读写方法被几个属性共享）。

如果你没有声明写方法，属性就是只读的。为了在设计时能使用__published 属性，你必须将它们定义为读/写类型的。通常在改变属性之前，写方法需要检测新的属性值与原有值是否相同。

2.4.5 缺省属性值

当声明了属性之后，可以为它设置缺省值。C++ Builder 使用缺省值确定是否把属性存储在窗体文件中。如果没有为属性指定缺省值，C++ Builder 总是存储该属性。为了为属性声明缺省值，在属性名后加一个等号和一对大括号，大括号中包含 default 关键字与缺省值。例如：

```
__property bool IsTrue = {/*...*/,default=true};
```

注意声明缺省值并非就是将属性设置为该值，组件的构造函数应对属性进行初始化。另外，由于对象总是把它们的数据成员设为 0，因此，无需严格要求构造函数非要把缺省值为 0 整数属性设为 0、把字符串属性设为空、把缺省值为 false 的布尔型属性设为 false 不可（参见前面对完整性的要求）。

可以为属性指定无缺省值，当重声明属性时，可以指定该属性没有缺省值，即使继承的属性已指定了一个，也可以这么做。为了将属性指定为无缺省值，在属性名后加一个等号和一对大括号，大括号中为 nodefault 关键字。

例如：

```
__property bool IsTrue = { /*  */,nodefault };
```

在第一次声明属性时，你无需包含 nodefault，没有声明缺省值就意味着无缺省值。下面是个组件的声明，该组件有个 bool 型属性 IsTue，缺省值为 true，构造函数设置了此值。

```
class PACKAGE TMyComponent : public TComponent
{
private:
    bool FIsTrue;
public:
    virtual __fastcall TMyComponent( TComponent* Owner );
__published:
     __property bool IsTrue = {read=FIsTrue, write=FIsTrue, default=true};
};

__fastcall TMyComponent::TMyComponent( TComponent* Owner )
          : TComponent( Owner )
{
    FIsTrue = true;                  //初始化属性的缺省值，
      //需要注意，初始化属性尽量不要对属性进行，而是对属性的内部存储数据进行操作，
      //因为设置属性通常会引发更新动作，而在构造函数没有执行完以前，某些更新操作是
      //不能进行的，也是没有必要的!! 后面 2.6.2 中例子有所不同。
}
```

指定缺省值的目的是为了在程序在存储和载入属性的时候判断是否需要将设计时的属性值存储在资源中，读者可以通过后面 2.6 的内容了解到，缺省值可以有效的压缩不必要的重复过程和程序可执行文件的体积。

2.5 创建数组属性

某些属性可能需要向数组那样使用索引，这些属性在使用上和数组比较类似，我们称之为数组属性，但并不是指内部存储数据为属性，或者将属性当做 C++中的数组一样使用，数组属性只能按照单个数组元素的方式来使用。

例如，TMemo 的 Lines 属性就是一个由备注文本组成的、可以索引的列表，你可以把它当作字符串数组。Lines 属性为在大量数据（备注文本）中访问特定的元素（一个字符串）提供了很自然的访问手段。除了下面的例外，数组属性同其他属性一样声明：

- 声明包含至少一个规定类型的索引。索引通常都是整型。
- 如果指定了属性声明中的 read 和 write 部分，那么它们必须为方法，而不是数据成员。任何数组属性的读和写方法都有一个额外的参数标识索引，该参数必须与声明中指定的索引类型、次序相同。数组属性和数组有一些重要的不同之处。

不像数组，数组属性的索引并非一定是整型，例如，可以以字符串为索引。另外，只能引用数组属性的单个元素，而不能引用属性的整个范围。

下面显示如何声明用整型作为索引返回字符串的属性。

```
class PACKAGE TDemoComponent : public TComponent
{
private:
   System::AnsiString __fastcall GetNumberSize(int Index);
public:
   __property System::AnsiString NumberSize[int Index] =
                                                {read=GetNumberSize};
   // ...
};
```

下面是 .CPP 文件中的 GetNumberSize 方法：

```
System::AnsiString __fastcall TDemoComponent::GetNumberSize(int Index)
{
   System::AnsiString Result;
   switch (Index)
   {
     case 0:
       Result = "Zero";
       break;
     case 100:
       Result = "Medium";
       break;
     case 1000:
       Result = "Large";
       break;
     default: Result = "Unknown size";
   }
   return Result;
}
```

2.6 存储和载入属性

C++ Builder 将窗体和窗体的组件存储在窗体文件（.DFM，参见 C++ Builder 中资源使用部分的内容）中。编译后窗体文件使用二进制数据描述窗体和它的组件，这就

是窗体的资源。如果 C++ Builder 开发员将你的组件加入他的窗体，那么当窗体被存储时，你的组件应有能力将自己的属性写入窗体文件中。同时，当 C++ Builder 需要载入窗体或组件作为程序的一段代码被执行时，组件也必须能从窗体文件中恢复自己的属性。通常对于这些，你无需做任何工作，因为你的组件已经继承了这种能力。但有时，设计一个组件可能想换种存储方式或者改变载入时的初始化方式，这组件设计中实在经常碰到的，也是有必要的，因此，对于一个优秀的 C++ Builder 程序员应理解它的底层处理机制。包括以下几个方面：

2.6.1 使用存储-载入机制

窗体描述是由一系列窗体属性和类似的组件属性描述组成的，存储在窗体文件中。每个组件，包括窗体自身，都需自己存储与载入自己的描述。缺省情况下，当组件存储自身时，组件按次序存储 public 属性和与缺省值不同的 published 属性。而载入时，组件首先构造自己，并在构造函数中将所有属性设置为缺省值，然后读取已存储的非缺省属性值。缺省机制可以满足大多数组件的需要，组件开发员通常无需做任何工作。但也有几个途径可定制存储和载入机制以满足自身的需要。

2.6.2 指定缺省值

只有属性值与它的缺省值不同时，C++ Builder 才会存储它们。如果不指定缺省值，C++ Builder 就假定该属性没有缺省值，这意味着而不管是什么值，组件都要存储它们的属性。要指定属性的缺省值参见前面内容。

如上所属，也可以在重新声明时指定缺省值。实际上，重新声明属性的一个目的就是修改指定的缺省值。注意指定缺省属性值并不意味着在对象创建时会将该值赋给它。你必须通过构造函数对属性进行必要的赋值。如果构造函数没有对某属性赋值，那么无论该属性值曾被假定为什么值，在内存中都会被设置为 0。也就是说，数字值设为 0，布尔值设为 false，而指针设为 NULL，诸如此类。如果你有什么不放心，那么处于完整性考虑最好在构造函数中设置属性值。

下面的代码是个组件的声明，该组件在构造函数中将 Align 属性设置为缺省值。在这个例子中，新组件是个标准的面板组件，用于窗体的状态条，因此它缺省的对齐方式是对齐窗口的底部。

```
class PACKAGE TMyStatusBar : public TPanel
{
public:
    virtual __fastcall TMyStatusBar(TComponent* AOwner);
__published:
    __property Align = {default=alBottom};
};
```

.CPP 文件中 TMyStatusBar 组件的构造函数如下：

```
__fastcall TMyStatusBar::TMyStatusBar (TComponent* AOwner)
                : TPanel(AOwner)
{
    Align = alBottom;
}
```

在这个例子中，Align 属性是通过派生而来的，在 TMyStatusBar 的构造函数中这个属性是可用的，当改变 Align 属性是会引发相应的更新动作，但是引发的是父类 TPanel 的更新动作，从前面章节关于 C++ Builder 类的构造过程中，我们可以知道，TPanel 在 TMyStatusBar 构造之前已经被创建，因此可以直接对 Align 属性进行操作，而且无法对 Align 属性的内部数据 FAlign 进行直接操作,这和 2.4.5 的例子中的注解有所不同。

这里就出现了前面说过的问题，如果一个父类的属性是只读的，在其派生类中无法对其缺省值进行修改，而且不能访问原始的数据成员，而在 Delphi 中，这是允许的，读者可以思考以下，对于这种情况应该如何处理。

2.6.3 决定存储内容

可以控制 C++ Builder 是否存储组件属性。缺省情况下，类声明中所有的 published 属性都会被存储。但可以指定不存储某一属性，或者指定一个程序在运行时决定是否存储该属性。要控制属性的存储可以使用 stored 关键字。

下面的组件声明了 3 个新属性。一个总是被存储，一个不存储，而另一个是否存储由返回 bool 类型方法的值决定：

```
class PACKAGE TSampleComponent : public TComponent
{
protected:
    bool __fastcall StoreIt();
public:
   ...
__published:
    __property int Important = {stored=true};         //总是存储
    __property int Unimportant = {stored=false};      //总是不存储
    __property int Sometimes = {stored=StoreIt};    //是否存储取决于StoreIt
};
```

2.6.4 载入后的初始化

组件从被存储的描述中读取所有的属性值后，将调用 Loaded 虚拟方法执行所需的初

始化。这个调用发生在显示窗体和它的组件之前，因此，你无需担心初始化会导致屏幕闪烁。为了在载入后初始化组件，你要重载 Loaded 方法。注意任何 Loaded 方法做的第一件事就是调用继承的 Loaded 方法，这保证了继承的属性在你初始化你自己的组件之前能正确地初始化，也保证了扩充增强了的属性不会被继承来的 Loaded 方法覆盖而使你徒劳无功。

2.6.5 存储和载入未发布的属性

缺省情况下，在组件中只有 published 属性被存储和载入。但是，存储和载入未发布的属性也是可以的。这意味着你可以拥有相对固定的、且不显示在对象观察器中的属性。并且，你还能够存储和载入由于过于复杂，C++ Builder 不知道如何存储和读取的属性。例如，TString 对象不能依赖 C++ Builder 的自动化行为来存储和载入它描述的字符串，因而必须使用下面的机制。通过加入代码告诉 C++ Builder 如何存储和载入，你可以存储和载入你的组件属性，有以下两种方式：

1.创建方法来存储和载入属性值为了存储和载入未发布的属性，首先必须创建两个方法，一个用于存储属性值，另一个载入属性值。有两种方式可以选择：

- 创建 TWriterProc 类型的方法存储属性值，创建 TReaderProc 类型的方法载入属性值。这个选择允许你使用 C++Builder 内置的支持读写简单类型的能力。如果你的属性值是由 C++ Builder 知道如何读写的类型组成，那么选择这个方式。
- 创建两个 TStreamProc 类型的方法，一个用于存储属性值，一个用于载入属性值。TStreamProc 的参数中有流，你可以用流方法读写属性值。

例如，考虑一个属性，它标识一个在运行时创建的组件。C++ Builder 知道如何写它的值，但却不能自动地做，因为该组件在窗体设计时并未被创建。由于流系统能够存储和载入组件，因此你可以选择第一种方式。下面的方法可以存储和载入动态创建的组件，该组件是 MyComproperty 属性的值：

```
void __fastcall TSampleComponent::LoadCompProperty(TReader *Reader)
{
   if (Reader->ReadBoolean())
      MyCompProperty = Reader->ReadComponent(NULL);
}

void __fastcall TSampleComponent::StoreCompProperty(TWriter *Writer)
{
   if (MyCompProperty)
   {
      Writer->WriteBoolean(true);
      Writer->WriteComponent(MyCompProperty);
   }
```

```
    else
      Writer->WriteBoolean(false);
}
```

2．重载 DefineProperties 方法

如果曾创建过方法来存储和载入属性值，那么就会重载组件的 DefineProperties 方法。当 C++Builder 需要存储和载入组件时，就调用该方法。在 DefineProperties 方法中，必须调用当前归档员对象的 DefineProperties 方法或 DefineBinaryProperty 方法，将存储和载入属性值的过程作为参数传给它。如果使用 TReaderProc 和 TWriterProc 类型过程，则调用归档员对象的 DefineProperty 方法；如果使用 TStreanProc 类型过程，则应调用 DefineBinaryProperty 方法。无论用什么方法定义属性，都要传递一个布尔型参数指示是否需要存储属性值。如果该属性值是继承的或是有缺省值的，则无需存储它。

例如，当使用 TReaderProc 类型的 LoadCompProperty 方法和 TWriterProc 类型的 StoreCompProperty 方法时，重载 DefineProperties 方法如下：

```
void __fastcall TSampleComponent::DefineProperties(TFiler *Filer)
{
   //必须确保父类已经定义了其属性，这个例子中假定是从 Tcomponent 直接派生的
   TComponent::DefineProperties(Filer);
   bool WriteValue;
   if (Filer->Ancestor)            //检查继承来的值
   {
      if ((TSampleComponent *)Filer->Ancestor)->MyCompProperty == NULL)
        WriteValue = (MyCompProperty != NULL);
      else if ((MyCompProperty == NULL) || (((TSampleComponent *)Filer
                                  ->Ancestor)->MyCompProperty->Name !=
                                  MyCompProperty->Name))
         WriteValue = true;
      else
         WriteValue = false;
   }
   else // no inherited value, write property if not null
      WriteValue = (MyCompProperty != NULL);
   Filer->DefineProperty("MyCompProperty ", LoadCompProperty,
                                  StoreCompProperty, WriteValue);
};
```

3 创建事件

事件是一种特殊的属性，反映的是状态的改变，也就是系统或者应用程序所发生的事情（如用户动作或焦点改变）与响应代码之间的连接。响应代码被称为事件处理程序，差不多总是由应用程序开发人员编写。通过事件，应用程序开发人员可以定制组件的行为，而无需改变类自身。这也就是通常所说的代理（delegation）。处理大多数用户动作的事件已内置在所有的标准组件中，但你也可以定义新事件。理解了下面的内容，你就可创建新事件：

- 什么是事件。
- 实现标准事件。
- 定义你自己的事件。

由于事件是用属性实现的，因此，在创建或改变组件的事件之前，应该熟悉属性相关的内容。

3.1 什么是事件

关于什么是事件的问题，在前面讲述面向状态的思想时已经作过详细的介绍，但那是着重于事件和状态的关系以及从事件的作用等角度来讲的。从程序代码的角度来讲，事件是连接发生的事情与响应代码的机制。更为明确地说，对于某一个类实例，事件就是指向方法的 closure。从应用程序开发人员的观点来看，事件只是一个与系统中发生的事情相连的名字，如 OnClick，一些代码与这个名字联系在一起。例如，按钮 Button1 有个 Onclick 方法。缺省情况下，C++ Builder 在该按钮的窗体中生成一个事件处理程序 Button1Click，并将其分配给 OnClick。当该按钮的单击事件发生时，按钮就调用分配给 OnClick 的方法，即 Button1Click。要编写事件，应理解下面的内容：

3.1.1 事件是 closure

C++ Builder 使用 closure 实现事件。前面已经讲过，closure 是种指针类型，该类型的指针指向特殊类实例中的特殊方法。作为组件开发员，你可以把 closure 当作占位符处理：检测事件发生，然后调用用户指定的方法（如果有的话）。closure 拥有一个隐藏的、指向类实例的指针。当用户指定组件事件的处理程序时，不是指定了一个有特殊名字的方法，而是指定了特殊类实例中的特殊方法。通常，特殊的类实例就是包含该组件的窗体，但并非都是如此。

事件最典型的例子是所有的控件都继承了 Click 方法用于处理单击事件：

```
virtual void __fastcall Click(void);
```

如果用户的单击事件处理程序存在，Click 的实现就会调用它。如果用户指定了控件的 OnClick 事件处理程序，那么单击控件将导致该处理程序被调用。如果没有指定处理程序，什么也不会发生。

关于 closure 的详细内容参见前面关键字介绍部分的内容。

3.1.2 事件是属性

组件使用属性实现它们的事件。与其他属性不同，事件不使用方法实现它们的读写部分。相反，事件属性使用与其同类型的私有数据成员采用直接存取的方式，这并不是说事件不能采用读写方法来事件读写部分，而是大多数情况没有必要，因为事件反映的是属性的改变，而事件的改变一般不许要进行再次出发处理过程或者实现更新等动作。

根据约定,数据成员的名字同属性一样,但有一个 F 前缀。例如,OnClick 的 closure 被存储在 TNotifyEvent 类型的 FOnClick 数据成员中，OnClick 事件属性的声明类似：

```
class PACKAGE TControl : public TComponent
{
private:
    TNotifyEvent FOnClick;
    //...
protected:
    __property TNotifyEvent OnClick = {read=FOnClick, write=FOnClick};
    //...
};
```

关于 TNotifyEvent 和其他事件类型，参见联机帮助。同其它任何属性一样，在运行时能设置或改变事件的值。事件是属性的主要用处在于，组件用户可以在对象观察器中为事件指定处理程序，因此对于应用程序中添加事件（主要是区别在组件设计中使用面向状态技术），则可以不必声明为属性，作为一般的成员即可。

3.1.3 事件类型是 closure 指针类型

由于事件是指向事件处理程序的指针，因此事件属性类型必须为 closure 类型。同样，任何作为事件处理程序的代码必须是类中有适当类型的方法。为了与给定类型的事件相容，事件处理程序方法的参数个数、类型、次序、传递方式均应一样。C++ Builder 为标准事件定义了 closure 类型。当你创建自己的事件时，如果合适，你可以使用已有的 closure，或者自己定义一个，例如定义一个自己的 Click 事件的 closure：

```
typedef void __fastcall (__closure *TMyNotifyEvent)(TObject *Sender,
                                        int &MyExtendData);
```

在这个 closure 中比标准的 Click 事件增加了一个 int 类型的引用参数，比如可以通过给数据来记录这个事件被触发的次数等。用户需要自定义 closure 的时候，需要

注意：至少应该包含一个参数就是 TObject *Sender，用来表明事件的触发者，从语言的角度讲，这不是必须的，但是在程序中是非常有必要的，因为事件的处理代码可以根据 Sender 来判断是由谁来触发该事件的，这对于运行时创建的控件是必不可少的。

事件的返回类型是 void，或者说事件只能返回 void 类型。尽管如此，但你可以通过引用型的参数返回需要的值。当你这么做的时候，要保证在调用处理程序之前，传递的参数有正确的值，从而免去用户代码设置该值的麻烦。OnKeyPress 事件处理程序为 TKeyPressEvent 类型，定义了两个参数，一个标识产生该事件的组件，另一个即是引用参数，标识那个键被按下了：

```
typedef void __fastcall (__closure *TKeyPressEvent)(TObject *Sender,
                                        Char &Key);
```

通常，Key 参数含有用户按下的字符，在某些环境中，组件的用户可能想修改该字符。例如，在编辑控件中强制所有的字符为大写。在这种情况下，用户应如下定义按键事件处理程序：

```
void __fastcall TForm1::Edit1KeyPress(TObject *Sender, Char &Key)
{
    Key = UpCase(Key);
}
```

还可以通过使用按引用传递的参数让用户重载缺省的处理。

3.1.4 事件处理程序是可选的

当事件被创建时，要记住开发人员可能使用你的组件，却没有连接处理程序。这意味着你的组件会由于没有对应的事件处理程序而失效或出错。在 Windows 应用程序中，事件总是不断地产生，仅仅在可视组件上移动鼠标就会导致 Windows 发送鼠标移动消息，组件将该消息传至 OnMouseMove 事件。大多数情况下，开发人员不愿处理鼠标移动消息，这会出现问题，就是触发了事件而程序并没有指定处理代码。因此，开发的组件不应对每个事件都要求用户连接一个处理程序。此外，应用程序开发人员可以在事件中编写任何想写的代码。由于精心编码，VCL 组件的事件处理程序极少出现错误。自然，你无法保证自己开发的组件不出现逻辑错误，但至少可以保证数据结构在调用事件处理程序之前已被正确初始化，从而避免了应用程序开发人员访问无效数据。

3.2 实现标准事件

C++ Builder 的组件均继承了 Windows 常用消息的事件，即标准事件。但是这些事件均是内置于控件中，为 protected 类型，因此，应用程序开发人员无法为它们指定处理程序。当你创建控件时，你可以决定是否让你的事件对用户可见。当需要与标准组件

协作时，要考虑下面三个问题：

3.2.1 标识标准事件

标准事件有两类：为所有控件定义的事件和为标准窗口控件定义的事件。

1.为所有控件定义的事件

最基本的事件定义在 TControl 类中，所有控件，无论是窗口、图形，还是定制的，均继承这些事件。

下面是对所有控件均有效的事件：

OnClick	OnDragDrop	OnEndDrag	OnMouseMove
OnDblClick	OnDragOver	OnMouseDown	OnMouseUp

标准事件的调用是在 TControl 中声明的 **protected** 虚拟方法，方法的名字是相应的事件名。例如，OnClick 事件被 Click 方法调用，OnEndDrag 事件被 DoEndDrag 方法调用。下面是 TCustomUpDown 组件 Click 方法 Object Pascal 源代码：

```
procedure TCustomUpDown.Click(Button: TUDBtnType);
begin
   if Assigned(FOnClick) then FOnClick(Self, Button);
end;
```

从这里可以看出，重载相应的调用方法可以扩展组件内部对事件的处理，在需要触发事件的地方调用该方法就可以，当然在你开发组件的时候，也可以不使用重载标准事件的调用方法来触发事件，而是在直接添加如上代码来实现，但这会带来一个缺点，就是控件自身的处理代码将不能得到执行。

2．为标准窗口控件定义的标准事件

除了拥有一般控件都有的事件之外，标准窗口控件（从 TWinControl 派生的控件）还拥有下列事件：

OnEnter	OnKeyDown	OnKeyPress
OnKeyUp	OnExit	

窗口控件的事件同 TControl 中的标准事件一样有相应方法。

3.2.2 使事件可见

TControl 和 TWinControl 中标准事件均声明为 protected，同它们对应的方法一样。如果你继承上述抽象类，并且想使事件在设计时或运行时可见，应将它们重新声明为 public 或 published。重新声明属性不需要它的实现，而是保持原有的实现，重新声明只是改变它的保护等级。可以使用 TControl 中定义的事件，它是不可见的，并将它封装为 public 或 published。

例如，创建一个组件，在设计时封装 OnClick，需要在组件类中加入下列声明：

```
class PACKAGE TMyControl : public TCustomControl
{
   //...
__published:
   __property OnClick;        //使 OnClick 事件在 Object Inspector 中可见
};
```

3.2.3 改变标准的事件处理

如果你想改变组件响应某一类事件的方式，你可能会给事件加一些代码。作为应用程序开发员，这正是他们所要做的。而你创建组件时，必须允许使用组件的应用程序员访问事件。这就是与标准事件连接的方法被实现为 protected 的缘由。通过重载方法，你可以修改内部的事件处理。通过调用继承的方法，你可以得到标准的处理过程，包括为应用程序开发员代码服务的事件。调用方法的次序是非常重要的。作为一般原则，首先调用继承的方法，使应用程序开发员的事件处理程序在你定制之前被执行（在某些情况下，会禁止执行定制）。然而，有时需要在调用继承的方法之前先调用你的代码。例如，继承的代码可能依赖组件的状态，而你的代码修改该状态，此时，你应先运行你的代码，然后允许用户的代码对此响应。假定你编写了一个组件，且你改变它响应鼠标单击的方式。你可以重载 protected 方法 Click，而不要像应用程序开发人员那样为它指定事件处理程序 OnClick：

```
void __fastcall TMyControl::Click()
{
   TWinControl::Click();       //调用标准的 Click
   //自定义代码
}
```

3.3 定义你自己的事件

定义一个全新的事件在组件设计中是并不多见的，但是在应用程序中使用面向状态的方法是，出现的机会可能要大一些。然而，当组件的行为完全不同于其他组件时，你需要为它定义事件。定义事件时，需要考虑下列内容：

- 触发事件。
- 定义处理程序类型。
- 声明事件。
- 调用事件。

本书中一个应用程序的例子 Sells 其中报表和系统设置的 Form 窗体都是使用了动态连接库来实现的，实际上载入的接口就是定义了自己的事件。

3.3.1 触发事件

应知道是什么触发了事件。对某些事件，结果是显然的。例如，当用户按下鼠标左键时，Windows 向应用程序发送 WM_LBUTTONDOWN 消息，触发鼠标按下事件。接收到该消息后，组件调用 MouseDown 方法，MouseDown 则调用 OnMouseDown 事件句柄，并将按键作为事件的参数，进而执行用户连接的代码。但是某些事件并不是清楚地与外部事情相连接。例如，滚动条的 OnChange 事件可以有多种触发方式，包括击键、鼠标单击以及其他组件的改动。定义事件时，你要保证所有事情会调用正确的事件。

两类事件你要为两类事情提供事件：用户交互和状态改变。用户交互事件几乎总是由 Windows 传递的消息触发,消息指示组件响应用户动作。状态改变事件也可以与 Windows 消息（例如，焦点改变或允许焦点）相连接，但是也会由属性改变或其他代码触发。对你定义的事件，你拥有完全的控制。小心定义事件让开发人员能够理解和使用它们。

读者请参阅前面章节相关内容理解事件的触发过程。

3.3.2 定义处理程序类型

一旦你确定了事件什么时候发生，你必须定义如何处理事件。这意味着确定事件处理程序的类型。在大多数情况下，定制事件的处理程序要么是简单的通知，要么是专用事件类型。你可以从处理程序中返回信息。

1.简单的通知

通知事件只是告诉你某个事件发生了，不含时间和地点信息。通知使用 TNotifyEvent 类型，该类型只有一个参数，即事件的发送者。所有通知的处理程序都知道事件的种类以及发生事件的组件。例如，单击事件是通知。当你为单击事件编写处理程序时，你所知道的就是那个组件发生了一次单击。通知是单向过程。没有提供反馈用以禁止进一步处理的机制。

2.专用事件类型

在某些情况下，仅知道什么组件发生了什么是不够的。例如，按键事件需要知道用户按下了哪个键。在这种情况下，你需要含有其他信息的处理程序类型。如果事件是为消息定制的，那么所需的参数可以直接从消息参数中得到并经过加工提供给事件句柄。

3.从处理程序返回信息

由于所有的处理程序只返回空值（void）,故获取信息的唯一途径就是传递引用参数。从用户代码返回以后，你的组件可以通过返回的信息决定是否或如何处理事件。

例如，所有的键事件（OnKeyDown、OnKeyUp 和 OnKeyPress）传递引用键值的参数 Key。事件处理程序可以修改 Key，这样作使得应用程序得到的按键与发生事件的按键不同。如前面的例子可以强迫所有的字符大写。

3.3.3 声明事件

一旦你确定了处理程序的类型，就可以为事件声明 closure 和属性了。确保事件的名字是有含义的、描述清楚的，这样用户才能理解事件是干什么用的。多个组件中类似的属性尽量使用统一的名字。事件名以“On”开头 C++ Builder 的大多数事件以“On”

开头，这仅仅只是一个约定，而不是编译器强制的。对象观察器通过查看属性的类型确定该属性是否是事件：所有的 closure 属性（声明于 publish 中的）被认定为事件，并被显示在事件页面上，并允许程序员双击列表框来创建新的处理方法。开发人员习惯通过观察名字的起始是否有“On”来找到事件。使用其他的名字很容易混淆。此外 After 和 Before 也是常用的事件的起始单词。

3.3.4 调用事件

你要集中调用事件，也就是说，在你的组件中创建一个虚拟方法，调用应用程序的事件处理程序（如果有的话），并提供缺省处理。将所有的事件调用集中在一起，当有人从你的组件派生新组件时，通过重载某个方法定制处理，此时，他无需搜遍你的代码寻找你调用事件的地方。

调用事件还需考虑如下两个方面：

- 空处理程序是允许的。
- 用户可以重载缺省的处理。

1.空处理程序是允许的

不要编写会导致空处理程序出错的代码，不要使你的组件依赖于用户的事件处理代码。空处理程序产生如同没有处理程序一样的结果，因此，调用应用程序事件处理程序形状应该如下：

```
if (OnClick)
    OnClick(this);
    //控件的缺省处理
}
```

而且绝对不要这么做：

```
if(OnClick)
    OnClick(this);
else
    //缺省处理
```

2.用户可以重载缺省的处理

对于某些事件，开发人员想取代缺省处理，甚至取消所有响应。为此，你应传递一个引用参数给处理程序，然后检查返回值。这里要遵循的一个约定是：空事件处理程序的影响同根本没有处理程序一样。由于空处理程序并不改变参数值，因此缺省的处理在空处理程序被调用后总会被执行。

例如，处理按键事件时，用户可以通过设置 Key 为空字符来取消组件的缺省处理。逻辑过程如下：

```
if(OnKeyPress)
    OnKeyPress(this, &Key);
if(Key != NULL)
    //执行标准的缺省处理
```

实际的代码与此有些不同，因为它要处理 Windows 消息，但逻辑是一样的。缺省情况下，组件先调用用户指定的处理程序，然后执行标准的处理。如果用户的处理程序将 Key 设为空字符，组件跳过缺省处理。

4 创建方法

组件方法与其他类方法之间并无不同之处，也就是说，它们都是组件类的成员函数。虽然，对如何处理组件的方法本质上没有任何限制，但 C++ Builder 使用了一些应遵守的标准。包括以下内容：

- 避免相关性。
- 命名方法。
- 保护方法。
- 虚拟方法。
- 声明方法。

通常，组件不应有太多的方法，应尽量减少应用程序需要调用的方法。用方法实现的特性常常可以用属性封装，属性提供适合 C++Builder 环境的接口，可以在设计时访问，并且是实现面向状态设计的基础。

4.1 避免相关性

编写组件时，自始至终都要尽可能减少强加给应用程序开发人员的约束。无论他们想做什么都应尽最大可能满足他们的要求。有时候，可能无法做到这一点，但也应尽可能接近这个目标。

下面列出了应避免的各种相关性：

- 用户必须调用的方法。
- 必须按一定次序调用的方法。
- 使组件进入可能会导致某个事件或方法无效的状态与模式的方法。

处理这种情形的最好办法就是确保不提供上述类型的方法。例如，如果调用某个方法会导致组件进入某个状态，在该状态中另一个方法会失效，那么应该重新编写第二个方法（即会出错的那个方法），这样，当组件在该状态中调用该方法时，该方法就首先修正状态，然后执行主要代码。至少，当用户调用某个无效方法时应发送异常。换句话说，如果代码的一部分依赖另一部分，那么你有责任保证当代码被不正确调用时不能产生问题。例如，当用户不适应你的代码时，产生警告消息显然比导致系统错误更好。

4.2 保护方法

类的所有部分，包括数据成员、方法和属性，都有保护等级或者说“可见性”，具体解释参见前面内容。为每个方法选择可见性是相当简单的。组件中大多数方法的可见性是 public 或 protected 的。很少需要 private 方法，除非它是专为本类型的组件定制、甚至由其派生组件都不应访问的。

注意通常没不需要将方法声明为__published。这样做的结果是对于用户来说，与声明方法为 public 没有什么不同。

4.2.1 应为 public 的方法

任何应用程序开发人员需要调用的方法都必须声明为 public。需要记住的是，大多数方法都是在事件处理程序中调用的，因此，方法要避免占有系统资源或导致 Windows 无法响应用户。

类的构造函数和析构函数必须为 public。

4.2.2 应为 protected 的方法

任何为组件实现的方法都应声明为 protected，这样应用程序就无法在错误的时刻调用它们。如果应用程序无需调用你的方法，而派生类却需要调用，那么将它们声明为 protected。例如，假设你的方法依赖于某个以前设置的数据，如果你将该方法声明为 public，那么应用程序就有可能在所需数据设置之前调用它。另一方面，将它声明为 protected，就可保证应用程序不会直接调用它。你可以另建一个 public 方法以保证在调用 protected方法之前,数据会被设置。用于实现属性的方法应声明为 virtual、protected 方法。这样，应用程序开发人员既可以重载该属性的实现，又可以修改它的功能或完全取代它。这样，属性将有充分的多态性。将方法声明为 protected，可以确保开发人员不会偶然地调用它们，或无意中修改属性。

4.3 虚拟方法

对于同一个方法，如果你想根据不同类型调用不同的代码，那么将你的方法声明为 virtual。如果你的组件是由应用程序开发人员直接使用，那么可以将所有的方法声明为 virtual。另一方面，如果组件是可以派生其他组件的抽象类，可以考虑将附加的方法声明为 virtual。这样，派生的组件可以重载它继承的 virtual 方法。

4.4 声明方法

声明组件的方法同声明任何类方法一样。要在组件中声明方法，要做以下的事情：

- 在组件头文件的类声明中加入声明。
- 在单元的.CPP 文件中编写实现代码。

下面是个组件的代码，它定义了两个新方法，一个 protected 方法和一个 public 虚拟方法。.H 文件中的接口定义为：

```
class PACKAGE TSampleComponent : public TControl
{
protected:
    void __fastcall MakeBigger();
public:
    virtual int __fastcall CalculateArea();
    //...
};
```

下面是在 CPP 文件中实现方法的代码

```
void __fastcall TSampleComponent::MakeBigger()
{
    Height = Height + 5;
    Width = Width + 5;
}

int __fastcall TSampleComponent::CalculateArea()
{
    return Width * Height;
}
```

5 在组件中使用图形

Windows 为设备无关图形的绘制提供了强大的图形设备接口（GDI）。但是，GDI 给程序员提出了一些额外的要求，如管理图形资源，正如前面所提到的那样，Windows 只提供了有限的绘图资源，如果不能正确的管理和使用，将会带来灾难性的后果。C++ Builder 提供了更高一级的资源管理能力，是程序员可以将精力集中在有效益的工作上，而不是放在寻找丢失的句柄或者释放资源等问题上。

与任何 Windows API 一起使用时，可以在 C++ Builder 应用程序中调用 GDI 函数。但是你很快会发现使用 C++ Builder 的图形函数封装可以更快更方便。

5.1 图形概述

C++ Builder 在多个层次上封装 Windows API。对作为组件开发员而言，最重要的是组件在屏幕上显示图片的方式。直接调用 Windows GDI 时，需要一个设备场境（device context）句柄，在设备场境中，你选择了各种绘图工具如笔、刷子和字体。在绘图以后，你必须将设备场境恢复到设置之前的原始状态。为了使你无需在低层处理图

形，C++ Builder 提供了一个简单、完备的接口：组件的 Canvas 属性。画布保证拥有一个有效的设备场境，而在你不用它时自动释放它。同样，画布拥有自己的用于描述当前笔、刷子和字体的属性。画布替你管理所有这些资源，因此，你无需再关心创建、选择和释放诸如画笔句柄这样的东西。你只需告诉画布你需要什么样的笔，画布自会处理其他事情。让 C++ Builder 管理图形资源的另一个好处是它能为以后使用资源进行缓冲，这样可以加速重复性的操作。例如，如果你有个程序需要重复地创建、使用和配置某种笔，你不得不在每次使用它时重复这些工作。由于 C++ Builder 能缓冲图形资源，那么很可能你重复使用的工具就在缓冲区中，因而你可以使用它们而无需重新创建。这方面的一个例子是，某个程序有几十个打开的窗体，有几百个控件。每个控件都有一个或多个 TFont 属性，这样就可能存在数百乃至数千个 TFont 对象实例。由于 VCL 的缓冲作用，大多数应用程序只使用两三个字体句柄。

下面有两个例子显示 C++ Builder 图形代码可以简单到什么程度。第一个例子使用标准的 GDI 函数在窗口中画一个蓝边的黄色椭圆，如果你使用其他的开发工具，你很可能就是这么干的；第二个例子是使用 C++ Builder 中的画布画同样的椭圆。

下面是在 Object Windows 中实现的代码：

```
void TMyWindow::Paint(TDC& PaintDC, bool erase, TRect& rect)
{
    HPEN PenHandle, OldPenHandle;
    HBRUSH BrushHandle, OldBrushHandle;
    PenHandle = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
    OldPenHandle = SelectObject(PaintDC, PenHandle);
    BrushHandle = CreateSolidBrush(RGB(255, 255, 0));
    OldBrushHandle = SelectObject(PaintDC, BrushHandle);
    Ellipse(10, 20, 50, 50);
    SelectObject(OldBrushHandle);
    DeleteObject(BrushHandle);
    SelectObject(OldPenHandle);
    DeleteObject(PenHandle);
)
```

下面的 C++ Builder 代码完成同样的事情：

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
    Canvas->Pen->Color = clBlue;
    Canvas->Brush->Color = clYellow;
    Canvas->Ellipse(10, 20, 50, 50);
}
```

显然使用 C++ Builder 中的 TCanvas 画布组件来实现 Windows 的绘图操作要比使用 API 高效、简洁的多。

5.2 使用画布

画布（TCanvas）在多个层次上封装 Windows 图形，包括高层用于画单个直线、形状和文本的函数；中层用于封装画布绘图能力的属性；下层直接访问 Windows GDI。表 5.2 总结了画布的能力。

表 5.2 画布能力摘要

层次	操作	工具
高层	画直线和图形	MoveTo、LineTo、Rectangle 和 Ellipse 等方法
	显示和调节文本	TextOut、TextHeight 和 TextRect 等方法
	填充区域	FillRect 和 FloodFill 方法
中层	定制文本和图形	Pen、Brush 和 Font 属性操作
	像素	Pixels 属性
	拷贝和融合图片	Draw、StrechDraw、BrushCopy 和 CopyRect 方法；CopyMode 属性
下层	直接调用 Windows GDI 函数	Handle 属性

TCanvas 类及其属性和方法的信息参见联机帮助。

5.3 使用图片

在 C++ Builder 中，你所做的大部分图形工作仅限于直接在组件和窗体的画布上绘图。C++ Builder 还提供对单个图形图片，如位图、元文件和图标的处理，包括调色板的自动管理。

下面是在 C++ Builder 中使用图片需要关注的三个重要方面：

- 使用图片、图形或画布。
- 载入和存储图形。
- 处理调色板。

5.3.1 使用图片、图形或画布

C++ Builder 中有三个处理图形的类：

1. TCanvas 标识窗体、图形控件、打印机或位图的位图作图面。画布总是作为其它对象的属性或者工具出现，从来不会是单独的类出现在程序中。
2. TGraphic 标识图形的图片，图片通常存于文件、资源，如位图、图标或元文件中。C++ Builder 定义了 TBitmap、TIcon 和 TMetafile 类，它们均从 TGraphic 派生。用户也可以定义自己的图形类。通过为所有图形定义基本的标准接口，TGraphic 为简化应用程序使用各种图形提供了一个简单的机制。
3. TPicture 是个图形容器，这意味着它可以包含任何图形类。也就是说，

TPicture 类型的项可以包含位图、图标、元文件或用户自定义的图形类型，应用程序可以通过图片类使用同一种方式访问它们。

例如，图片控件有 TPicture 类型的属性 Picture，从而可以显示各种图形。在 VCL 中，图片类总是包含有图形类，而图形类包含画布类（标准的含有画布的图形类是 TBitmap），通常，当处理图片时，仅仅使用了一部分通过 TPicture 显示的图形类，如果需要访问特写的图形类本身，可以参考图片的 Graphic 属性。

5.3.2 载入和存储图形

C++ Builder 中的所有图片和图形都能从文件中载入它们的图像，也能存储它们（也可以存储在不同的文件中）。你可以在任何时候载入和存储图片。调用图片的 LoadFromFile 方法从文件载入图像。调用图片的 SaveToFile 将图像保存到文件。LoadFromFile 和 SaveToFile 方法只有一个参数，即文件名。LoadFromFile 方法使用文件扩展名确定创建和装入的图形对象的类型。SaveToFile 方法保存与图形对象相适应的任何类型的文件。要装入图像控件的位图，将位图文件名作为参数传给图片的 LoadFromFile 方法：

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Image1->Picture->LoadFromFile("c:\\windows\\athena.bmp");
}
```

图片 TPicture 能够识别 BMP 是位图文件的标准扩展名，于是，它创建 TBitmap 图形，然后调用图形的 LoadFromFile 方法。由于图形是个位图，于是它从文件装入位图。

在 C++ Builder 中能够被图片识别的文件类型只有 Icon、Bitmap、Jpeg、Metafile 等几种类型，大多数的图形文件并不能直接在 VCL 中使用，需要载入这些文件可以通过派生图片类，并且重载其载入方法。

除了文件载入之外，VCL 还提供了“流”载入方法，也可以将图形文件存入“流”中，任何从 TStream 派生的类，入资源流、Socket 流、文件流或者是自定义的流都可以作为图形文件的载体。

5.3.3 处理调色板

当 C++ Builder 控件运行在基于调色板的设备上时，它自动支持调色板实现。也就是说，如果你有一个需要用调色板的控件，可以使用从 TControl 继承的两个方法去控制 Windows 调节调色板。

控件的调色板支持下面两个方面：

- 为控件指定调色板。
- 响应调色板改变。

大多数控件不需要调色板，但是那些有“丰富色彩（rich color）”的图形图像（如

图像控件）可能需要与 Windows 交互，然后屏幕设备驱动程序保证控件正确显示。Windows 称这个过程为实现（realizing）调色板。实现调色板是个确保前台窗口最大限度地使用自己的调色板的过程，后台窗口尽可能多地使用自己的调色板，然后将其他的颜色映射为“现实（real）”调色板中最接近的颜色。当窗口移到另一个窗口的前面时，Windows 不断地实现相应的调色板。

注意除了在位图中，C++ Builder 自身并未对创建或维护调色板提供专门的支持。但是，如果你有一个调色板句柄，C++ Builder 控件会替你管理它。

1.为控件指定调色板

为了给控件指定一个调色板，重载控件的 GetPalette 方法返回一个调色板句柄。

要控件指定调色板需要完成下列事情：

- 通知应用程序控件的调色板需要实现
- 指定将要实现的调色板

2.响应调色板改变

如果你的控件通过重载 GetPalette 指定了一个调色板，C++ Builder 会自动响应由 Windows 发送的调色板消息。处理调色板消息的方法是 PaletteChanged。PaletteChanged 的基本功能是确定是在前台还是在后台实现调色板。Windows 处理调色板实现的准则是，最顶层窗口使用前台调色板，其他窗口使用后台调色板。C++ Builder 对调色板实现做了进一步处理，即在同一个窗口中的控件按 Tab 次序实现调色板。只有当你的控件不是在 Tab 次序的第一个而又想使用前台调色板时，你才需要重载该缺省行为。

5.4 幕后位图

当绘制复杂的图形图像时，Windows 编程中一个常用的技术是创建一个幕后位图，在该位图上绘制图像，然后将其从幕后位图中拷贝到屏幕上。使用幕后图像减少了由于重复直接在屏幕上作图而产生的闪烁。

C++ Builder 的位图类表示资源和文件中的位图图像，也能以幕后位图的方式工作。

当绘制复杂的图像时，要避免直接在屏幕画布上绘制。你可以构造一个位图对象，在它的画布上绘制，然后将其拷贝到屏幕的画布上去，而不要直接在窗体或控件的画布上绘制。通常人们在图形控件的 Paint 方法中使用幕后位图。本书的例子中，DataPick 就使用了幕后位图来实现图形的绘制，从而避免了绘制过程中图形频繁闪烁。

C++ Builder 提供了四种不同的方式将图像从一个画布拷贝到另一个。依据需要的效果，可以调用不同的方法。

下面列出了这几种拷贝的方法和效果：

效果	方法
拷贝整个图形	Draw
拷贝图形且重置图形大小	StretchDraw
拷贝画布的一部分	CopyRect
在拷贝中使用光栅操作	BrushCopy

这几种方法在 API 中有相应的函数和其对应，但使用起来要比它们复杂一些，而且根据实际测试的结果，使用 VCL 的图形函数在绘图速度上并不比直接使用 API 慢，而在对

Jpeg 文件的测试中，VCL 的效果比 API 还略胜一筹。

5.5 响应改变

所有图形对象，包括画布和它拥有的对象（笔、刷子和字体）均有内置的方法响应对象的改变。通过使用这些方法，可以使你的组件响应改变重画图像。如果将对象作为设计时组件接口的一部分发布的话，那么响应图形对象的改变就尤其重要。使组件设计时的显示与在对象观察器中的属性设置一致的唯一途径就是响应对象的改变。为了在图形对象中响应改变，要为类的 OnChange 事件指定一个方法。

例如 shape 组件发布了标识自己使用的笔和刷子的属性。组件的构造函数为每个 OnChange 事件指定了一个方法，于是当笔或刷子发生改变时组件将重画自己的图像。shape 组件是用 Object Pascal 编写的，而下面的代码是它的 C++版本，用了个新名字 TMyShape。

下面是头文件中的类声明：

```
class PACKAGE TMyShape : public TGraphicControl
{
private:
protected:
public:
    virtual __fastcall TMyShape(TComponent* Owner);
__published:
    TPen *FPen;
    TBrush *FBrush;
    void __fastcall StyleChanged(TObject *Sender);
};
```

下面是.CPP 文件代码：

```
__fastcall TMyShape::TMyShape(TComponent* Owner)
                                    :TGraphicControl(Owner)
{
    Width = 65;
    Height = 65;
    FPen = new TPen;
    FPen->OnChange = StyleChanged;
    FBrush = new TBrush;
    FBrush->OnChange = StyleChanged;
}
```

```
void __fastcall TMyShape::StyleChanged(TObject *Sender)
{
    Invalidate();
}
```

6 处理消息

在传统的 Windows 编程中，一个关键之处就是处理 Windows 发送给应用程序的消息。C++ Builder 为你处理大多数消息。但是，你还是可以处理 C++ Builder 未处理的消息或创建你自己的消息。

使用消息需要注意下面三个方面：

- 理解消息处理系统。
- 修改消息处理。
- 创建新的消息处理器。

6.1 理解消息处理系统

所有的 C++Builder 类都有内置的消息处理机制，称为消息处理方法或消息处理器。消息处理器的基本思路是，类接收某些消息和分派消息，依据接收的消息调用一系列指定的方法。对某一消息如果没有指定的方法，就进行缺省处理。

下面的框图显示了 VCL 的消息分派系统：

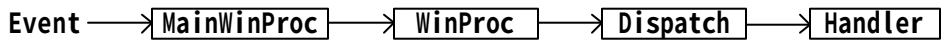


图 5.1 VCL 组件消息派发的顺序

在前面介绍 VCL 类库的时候，我们已经讲过关于消息处理的一些属性和方法等，也对 VCL 的消息处理机制作了本质上的分析，实际的 VCL 消息处理过程遵循着上面图中的顺序，对于消息的响应，可以根据需要在上面的消息链中加入自己的消息处理代码，也可以使用宏定义 MESSAGEMAP 来响应 Windows 消息（实际上 MESSAGEMAP 是对 Dispatch 方法的重载），但有一点需要注意，无论采用什么样的方式，都必须保持上面的消息链不能断，如重载了某个消息处理方法或者重新设置了某个消息处理的属性时，必须保证至少是自己没有处理的消息仍然要传递给原有的消息处理过程。

下面的例子是使用了自定义的消息处理方法来对 Form1 的 WM_COMMND 消息进行处理，采用的是重新设置 WindowProc 属性的方式来在消息链中增加自己的消息处理代码，如下：

.H 文件中定义

```
class TForm1 : public TForm
{
__published:
```

```
        void __fastcall FormCreate(TObject *Sender);
public:
        TWndMethod OldWinProc;
        void __fastcall MyWinProc(Messages::TMessage &Msg);
        __fastcall TForm1(TComponent* Owner);
};
```

CPP 文件中实现代码

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    OldWinProc = WindowProc;    //保存原有的 WindowProc 属性
    WindowProc = MyWinProc;     //设置新的 WindowProc 属性
}
//---------------------------------------------------------

void __fastcall TForm1::MyWinProc(Messages::TMessage &Msg)
{
    if(Msg.Msg == WM_KEYDOWN)
    {     //以下代码只对 WM_KEYDOWN 起作用
       switch (Msg.WParam)
       {
          case VK_F1:
            //...  执行自定义代码，并且不提交给原有处理过程
            Application->MessageBox("OK This is My Proc","OK",0);
            break;
          case VK_F4:   //... 其它的处理键
            //...
            break;
          default :
            OldWinProc(Msg);
            //缺省时，提交给原有处理过程
       }
    }
    else
       OldWinProc(Msg);
       //非 WM_KEYDOWN 消息时，提交给原有处理过程
}
```

在这个例子中使用了 TWinControl 提供的 WindowProc 属性，而 WindowProc 属性对应的处理方法实际上就是 TWinControl 的 WndProc 方法，WndProc 是一个虚拟方法，因此也可以通过重载 WndProc 的方式来实现以上功能，其代码如下：

.H 文件中定义

```
class TForm1 : public TForm
{
public:
       virtual void __fastcall WndProc(Messages::TMessage &Message);
       __fastcall TForm1(TComponent* Owner);
};
```

CPP 文件中实现代码

```
void __fastcall TForm1::WndProc(Messages::TMessage &Message)
{
   if(Message.Msg == WM_KEYDOWN)
   {     //以下代码只对WM_KEYDOWN起作用
      switch (Message.WParam)
      {
         case VK_F1:
           //...  执行自定义代码，并且不提交给原有处理过程
           Application->MessageBox("OK This is My WinProc ","OK",0);
           break;
         case VK_F4:  //...其它的处理键
           //...
           break;
         default :
           TWinControl::WndProc(Message);
           //缺省时，提交给原有处理过程
      }
   }
   else
      TWinControl::WndProc(Message);
      //非WM_KEYDOWN消息时，提交给原有处理过程
}
```

比较这两种方式，发现使用属性比起重载方法要稍微复杂一些，但是使用属性会带来一些灵活性。而且，实际上 WindowProc 是一个没有发布的“事件”，具有事件的所有特

征，因此可以在 Form 中定义一个消息处理方法并将其赋给任何一个 Form 中的窗体控件的 WindowProc 属性，而重载 WndProc 方法只能在用户代码定义的类中使用。并且属性允许运行时更换处理句柄，而重载的方法一旦生成可执行代码就不能再更改。

关于消息处理的其它方面在后面的内容中进一步介绍。

6.1.1 Windows 消息中有什么

可以把 Windows 消息当作包含几个有用数据成员的数据结构。成员中最重要的是个标识消息的整型值。Windows 定义了许多消息，MESSAGES.HPP 中声明了它们的标识符。Windows 程序员早已习惯了使用那些标识消息的 Windows 定义，如 WM_COMMAND 或 WM_PAINT。传统的 Windows 程序有个窗口过程作为系统消息的回调函数。在窗口过程中，通常有一个庞大的 switch 语句，且对每个有待处理的消息有一个 case 语句。附加的信息作为两个参数 wParam 和 lParam 传递给该窗口过程，分别用于“字类型参数”和“long 类型参数”。通常，每个参数都含有不止一个信息，因此需要调用 LOWORD 和 HIWORD 宏取出相应的信息。

例如，调用 HIWORD(lParam)得到该参数的高字。早先，Windows 程序员不得不记住或在 Windows API 中查找每个参数包含的信息。近来，Windows 实现了“消息解析器”以简化处理 Windows 消息和消息参数的语法。通过使用“消息解析器”代替庞大的 switch 语句解析消息参数，你可以简化消息处理函数。如果你在标准的 Windows 程序中包含了 WINDOWSX.H，就可以在你的代码中使用 HANDLE_MSG 宏：

```
void MyKeyDownHandler(  HWND hwnd, UINT nVirtKey, BOOL fDown, int CRepeat,
                        UINT flags )
{   //...  }

LRESULT MyWndProc( HWND hwnd, UINT Message, WPARAM wParam, LPARAM lParam )
{
   switch(  Message )
   {
      HANDLE_MSG( hwnd, WM_KEYDOWN, MyKeyDownHandler );
      //...
   }
}
```

HANDLE_MSG 宏的原型如下：

```
#define HANDLE_MSG(hwnd, message, fn)    \
   case (message):return HANDLE_##message((hwnd),(wParam),(lParam),(fn))
```

通过使用这种消息解析，可以很清楚地看到消息被派发给了指定的处理器。你还可以给处理函数的参数表命名有意义的名字。对于 WM_KEYDOWN 消息处理函数来说，将

wParam 参数命名为 nVirtKey 显然有助于理解。

要了解详细的 Windows 消息结构请参阅 Windows SDK 帮助文件。

6.1.2 派发消息

当应用程序创建窗口时，它在 Windows 内核中注册了一个窗口过程。窗口过程就是为该窗口处理消息的例程。传统上，窗口过程包含一个庞大的 switch 语句，该语句对每个有待处理的消息均有一个入口。记住，此处的“窗口”意思是屏幕上的任何东西：每个窗口、每个控件（不包括 VCL 中的图形控件），诸如此类。每次创建新类型的窗口时都不得不创建一个完整的窗口过程。

C++ Builder 采用几种方式简化消息派发：

- 每个组件都继承了完整的消息派发系统。
- 分派系统有缺省处理。只需为需要特殊处理的消息定义处理器。
- 可以依靠继承的方法完成大多数处理，而自己只需修改消息处理的一小部分。

C++ Builder 为应用程序中的每个组件注册了 MainWndProc 方法作为窗口过程。MainWndProc 包含异常处理块，它将 Windows 传来的消息结构传给虚拟方法 WndProc，并调用应用程序类的 HandleException 方法处理所有的异常。MainWndProc 不是虚拟方法，对任何特定的消息也不作特别处理。由于每个 VCL 控件都可以重载 WndProc 以适应自己特殊的需要，因此定制发生在 WndProc 中。WndProc 方法检查任何对处理有影响的条件，因此它可以“屏蔽”不想要的消息。例如，当被拖动时，组件忽略击键消息，因此 TWinControl 的 WndProc 只有在没有被拖动的情形下才传递击键事件。最终，WndProc 调用从 TObject 那儿继承来的虚拟方法 Dispatch 确定该调用哪个方法来处理消息。Dispatch 使用消息结构中的 Msg 数据成员来确定如何派发一个特定的消息。如果组件为该消息定义了处理器，那么 Dispatch 就调用该处理器。如果组件没有为该消息定义处理器，那么 Dispatch 就调用 DefaultHandler。

6.2 修改消息处理

在修改组件的消息处理之前，必须确保这确实是你想做的。C++ Builder 把大多数 Windows 消息转换为组件开发员和组件用户都能处理的事件。应尽可能修改事件处理，而不是修改消息处理。要修改消息处理，需要重载消息处理方法。可以通过屏蔽某一消息以禁止在特定环境下处理该消息。

6.2.1 重载处理器方法

为了修改组件对某一消息的处理，需要重载该消息的处理方法。如果组件还没有处理该消息，可声明一个新的消息处理方法。

要重载消息处理方法可以通过下面两种途径：

1. 在组件中声明一个新方法，新方法的名字与在组件声明的 protected 部分中被它重载的方法的名字一样。比如 Click 方法是用于处理 OnClick 事件的虚拟方法，在派生类中重载 Click 方法可以实现对 OnClick 事件的重载。

2. 使用宏将该方法映射到消息。宏的形式如：

```
BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER(parameter1, parameter2, parameter3)
END_MESSAGE_MAP
```

parameter1 是 Windows 定义的消息索引，parameter2 是消息结构的类型，parameter3 是消息方法的名字。可以在 BEGIN_MESSAGE_MAP 和 END_MESSAGE_MAP 两个宏之间放置任意个想放置的 ESSAGE_HANDLER 宏。

例如，为了重载组件的 WM_PINT 消息的处理，重新声明 WMPaint 方法，然后用上述三个宏将该方法映射到 WM_PAINT 消息：

```
class PACKAGE TMyComponent : public TComponent
{
protected:
    void __fastcall WMPaint(TWMPaint* Message);
BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER(WM_PAINT, TWMPaint, WMPaint)
END_MESSAGE_MAP(TComponent)
};
```

关于 BEGIN_MESSAGE_MAP 和 END_MESSAGE_MAP 宏在前面的章节中已经作了详细的介绍，稍加分析就会发现，经过宏展开后的形式和前面重载 WndProc 的形式基本相同，程序设计时也可以直接重载 Dispatch 来实现消息链的扩展。

比较图 6.1，可以发现，重载 WndProc 或者设置 WindowProc 属性对消息的处理要在 Dispatch 之前进行，因此采用重载 WndProc 方法响应消息要比使用消息映射表的宏更早一步，但使用宏定义可以得到有明确含义的消息结构，因此建议 C++ Builder 的初学者使用消息映射表，而熟悉 SDK 或者有经验的程序员使用重载 WndProc 的方法。

6.2.2 使用消息参数

在消息处理方法的内部，你的组件可以访问消息结构的所有参数。由于传递给消息处理器的是个指针，因此在需要时，处理器可以修改参数值。消息的返回值是被修改的最频繁的参数，该值是 SendMessage 的返回值。

由于 Message 参数的类型随被处理的消息变化，因此你应参考 Windows 消息文档，弄明白单个参数的名字和含义。如果出于某种原因，需要使用消息参数的老式名字（wParam、lParam，诸如此类），可以将 Message 转换为 TMessage 类型，该类型使用老式参数名。关于 TMessage 以及 wParam、lParam 等参数的详细内容请参考联机帮助。

6.2.3 屏蔽消息

在某些环境下，可能希望组件忽略某些消息。这时，你想禁止组件分派消息给自己的处理器。为了屏蔽消息，你要重载虚拟方法 WndProc 或者重新指定 WindowProc 属性，前面 6.1 中的例子就包含了对 F4 以及 F1 键盘消息的屏蔽。

WndProc 方法在将消息传给 Dispatch 方法之前能筛选消息。通过重载 WndProc，你将可以得到一个在分派消息之前过滤消息的机会。对于一个从 TWinControl 派生的控件，重载 WndProc 的代码看上去如下：

```
void __fastcall TMyControl::WndProc(TMessage* Message)
{
   //判断代码
   TWinControl->WndProc(Message);
}
```

其中判断代码可能是一个非常复杂和多样的形式，读者可以参考 6.1 中的例子进行深入的研究。

TControl 组件定义了一个鼠标消息的入口范围，当用户拖动和放下控件时，它可以过滤消息。通常有两种方式重载 WndProc 可以达到上述目的：

- 过滤消息范围，而不是为每个消息指定处理器
- 阻止消息分派，因而处理器根本不会被调用

下面是 TControl 的 WndProc 方法，它在 VCL 中用 Object Pascal 实现：

```
procedure TControl.WndProc(var Message: TMessage);
begin
   ...
   if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST) then
     if Dragging then
       //{ 为 Dragging 消息定义了处理句柄}
       DragMouseMsg(TWMMouse(Message))
     else
       //...            { 按照正常处理 }
     end;
       //...            { 否则亦按照正常处理 }
end;
```

上面这一段代码非常有参考价值，因为 WM_MOUSEFIRST 和 WM_MOUSELAST 很少被人们使用，但却是一个非常有用的消息，有兴趣的读者可以参考 VCL 的源码分析。

6.3 创建新的消息处理器

由于 C++ Builder 为大多数公共 Windows 消息提供了处理器，因此，当你定义了自己的消息的时候，你就需要创建新消息处理器。使用自定义消息有两方面内容：

- 定义自己的消息。
- 声明一个新的消息处理方法。

6.3.1 定义自己的消息

标准组件定义了许多内部使用的消息。定义消息最常见的理由是广播标准的 Windows 消息未能涵盖的信息以及状态改变的通知。

定义一个新消息有两个步骤。如下：

1.声明一个消息标识符

消息标识符是个整型常量。Windows 为自己保留小于 1024 的消息，因此，当你声明消息时，你应从大于 1024 开始。

常量 WM_APP 是自定义消息的起始值。当自定义消息时，应以 WM_APP 为基础。要知道，一些 Windows 标准控件使用了自定义消息范围内的消息。这些控件有列表框、组合框、编辑框和命令按钮。如果你从它们派生组件，并且想为组件定义一个消息，你必须检查 MESSAGES.HPP 文件，弄清楚 Windows 到底为它们定义了什么消息。

下面的代码自定义了两个消息：

```
#define WM_MYFIRSTMESSAGE  (WM_APP + 400)
#define WM_MYSECONDMESSAGE (WM_APP + 401)
```

2.声明一个消息结构类型

这并不是一个必需的过程，但是一个好的消息结构可以使对消息参数的使用更加方便、快捷。在早期的 Windows 程序设计中，程序员不得不牢记每一个消息中 wParam 和 lParam 参数分别传递了什么样的信息，C++ Builder 中 TMessage 结构就是使用了最原始的消息结构，这也是 windows 消息的缺省结构。

声明消息结构类型一般遵循下列约定：

- 命名消息结构类型，名字有个前缀 T 。
- 第一个数据成员为 Msg，类型为 TMsgParam。
- 定义紧接着的两个字节为 Word 型参数，再紧接着的两个字节不用。或者定义紧接着的四个字节为 Longint（WORD）型参数。
- 最后一个数据成员为 Result，类型是 Longint。

例如，下面是键盘消息的消息结构 TWMKey:

```
struct TWMKey
{
    Cardinal Msg;                    //第一个参数，用于标识消息的 ID
    WORD CharCode;                   //第二个参数，相当于 wParam
    WORD Unused;                     //相当于 lParam 参数低字
```

```
    WORD KeyData;                    //相当于 lParam 参数高字
    WORD Result;                     //第四个参数，消息的返回值
};
```

6.3.2 声明一个新的消息处理方法

在很多情况下，需要对某个 Windows 消息进行响应，尽管声明新的消息处理方法通过前面讲过的重载 WndProc 或者重设置 WindowProc 属性等方法均可以实现，但是通常对以下两种情况，建议重新定义新的消息处理方法：

- 你的组件需要处理标准组件未处理的 Windows 消息。
- 需要对自定义的消息进行处理。

要声明一个新的消息处理方法，操作如下：

- 命名消息处理方法，注意不要下划线字符并确保方法返回 void。
- 给消息处理方法传递消息结构类型的指针。
- 使用 BEGIN_MESSAGE_MAP 和 END_MESSAGE_MAP 宏映射方法到消息。
- 在方法实现中编写代码。
- 调用继承的消息处理器。

例如，下面是为自定义消息 CM_CHANGECOLOR 声明的处理器。

```
#define CM_CHANGECOLOR (WM_APP + 400)
class TMyControl : public TControl
{
protected:
    void __fastcall CMChangeColor(TMessage &Message);
BEGIN_MESSAGE_MAP
    MESSAGE_HANDLER(CM_CHANGECOLOR, TMessage, CMChangeColor)
END_MESSAGE_MAP(TControl)
};

void __fastcall TMyControl::CMChangeColor(TMessage &Message)
{
    Color = Message.LParam;            //从 lParam 获取 Color
    TControl::CMChangeColor(Message); //调用继承来的消息处理方法
}
```

7 使组件在设计时可用

C++ Builder 的组件存在于两种不通类型的库包（Package）中，一种是运行时库，

另一种是设计时库，运行时库以.bpl结尾，设计时库以.bpi结尾，要使组件在IDE中可以使用，必需生成设计时库文件，如果应用程序选择了静态链结，则运行时库并不是必需的，通常在新建一个库文件工程时，编译器缺省的设置为同时生成运行时库和设计时库。

下面的内容将描述如何使组件在IDE中可用所需的步骤。一般来讲使组件在设计时可用需要几个步骤：

- 注册组件。
- 添加组件面板位图。
- 为你的组件提供帮助。
- 添加属性编辑器。
- 添加组件编辑器。
- 将组件编译成软件包。

并非每个组件都需要所有的步骤。例如，如果没有定义任何新的属性或事件，无需为它们提供帮助，或者组件仅仅是共内部使用也没有必要生成帮助文件。但任何时候都必须的步骤是注册和编译。一旦你的组件已经注册且已编译成软件包，那么它们就可以分发给其他程序员安装到IDE中。

7.1 注册组件

注册工作是建立在编译单元的基础上，因此如果你在一个编译单元中创建了几个组件，你可以一次全部注册它们。为了注册组件，在单元的.CPP文件中添加一个Register函数。在该函数内部，你注册组件并决定将它们安装到组件面板的什么地方。

如果你是通过在IDE中选择Component | New Component来创建你的组件，那么注册所需的代码已自动添加了。需要注意的是如果第一次编译并生成库文件后，组件注册在那一个Component面板，它将一直不会改变，即使改变RegisterComponents函数（详见后面内容）中的参数，除非是在一个全新的C++ Builder中安装注册。

手工注册组件要经过以下几个步骤：

7.1.1 声明 Register 函数

注册过程包括.CPP文件中编写一个Register函数，且该函数必须位于一个名字空间中，名字空间的名字就是组件文件的名字，除第一个字母以外均为小写。

下面的代码就是一个名字空间中的Register函数。名字空间为Newcomp，文件名为Newcomp.CPP：

```
namespace Newcomp
{
    void __fastcall PACKAGE Register()
    {
        //注册组件的函数
    }
}
```

在 Register 函数内部为你每个想添加到组件面板的组件调用 RegisterComponents。如果头文件和.CPP 文件包含几个组件，你可以一次把它们添加到组件面板。由 PACKAGE 宏扩展的语句使组件能被导入和导出。

7.1.2 编写 Register 函数

在 Register 函数内部，必须把想添加到组件面板的每一个组件都注册。RegisterComponent 涉及三件重要的事情：

1.指定组件

在 Register 函数内部，声明一个 TComponentClass 类型的数组，该数组包含你要注册的组件。语法形如：

```
TMetaClass classes[1] = {__classid(TNewComponent));  //生成一个组件数组
```

在这个例子中，类数组中只有一个组件，但是你可以把想注册的所有组件都添加到数组中去。例如，下面的代码就有两个组件：

```
TMetaClass classes[2] =
            {__classid(TNewComponent),__classid(TAnotherComponent)};
```

另一种方式是采用单独的语句给组件类数组赋值。下面的语句同前面的语句作用是一样的：

```
TMetaClass classes[2];
classes[0] = __classid(TNewComponent);
classes[1] = __classid(TAnotherComponent);
```

2.指定组件面板的页面

组件面板页面的名字是个 ANSI 字符串，如果你给出的名字标识的页面并不存在，C++ Builder 将用此名创建一个新页面。C++ Builder 把标准页面的名字存储在字符串列表（string-list）资源中，因此国际版本的软件可以用当地的语言为页面命名。如果你想将组件安装于标准页面（指 C++ Builder 中提供的页面）上，应调用 LoadStr 函数获取该字符串资源，传递的参数是标识该页面字符串资源的常量，例如 System 页面的标识常量是 srSystem，关于组件面板的详细信息请参阅联机帮助。

3.使用 RegisterComponents 函数

在 Register 函数内部，调用 RegisterComponents 注册组件类数组中的组件。RegisterComponents 有三个参数：组件面板页面的名字、组件类数组和数组中最后一个入口的索引。

下面的 Register 函数代码位于 NEWCOMP.CPP 文件中，它注册了一个 TMyComponent 组件，并将其放在 Miscellaneous 页面中：

```
namespace Newcomp
{
   void __fastcall PACKAGE Register()
   {
      TMetaClass classes[1] = {__classid(TMyComponent)};
      RegisterComponents("Miscellaneous", classes,0);
  }
}
```

RegisterComponents 的第三个参数是 0，它是最后一个入口的索引（数组大小减去 1）。如果几个组件位于同一页面中，你可以一次注册它们，或者像下面的代码那样在不同的页面中注册：

```
namespace Mycomps
{
   void __fastcall PACKAGE Register()
   {
       //生成包含两个元素的组件数组
      TMetaClass classes1[2] = {__classid(TFirst), __classid(TSecond)};
        //注册 classes1 到 Miscellaneous 组件面板
      RegisterComponents("Miscellaneous", classes1, 1);
      TMetaClass classes2[1];              //声明第二个组件数组
      classes2[0]  = __classid(TThird);    //指定 classes2 数组的第一个元素
        //注册 classes2 数组中的第一个元素所对应的组件到 Samples 组件面板
      RegisterComponents("Samples", classes2, 0);
   }
}
```

例子中声明了两个数组，classes1 和 classes2。在第一次调用 RegisterComponents 中，classes1 有两个元素，所以第三个参数是 1；在第二次调用 RegisterComponents 中，classes2 只有一个元素，所以第三个参数是 0。

在 C++ Builder 中注册组件要比 Delphi 中复杂的多，这是很多 C++ Builder 程序员最恼火的地方，由于 VCL 最早是为 Delphi 而开发的，很多语法特点并没有顾及到 C++语言，读者可以参考前面讲述关于“Object Pascal 语言的 C++对应”中相关内容。

7.2 添加组件面板位图

在组件面板中，每个组件都需要一个位图标识自己。如果没有指定位图，C++Builder使用缺省位由于组件面板位图只是在设计时需要，所以不要将它们编译到组件的编译单元中去。相反，应在一个 Windows 资源文件中提供位图，资源文件名同.CPP 文件名相同，但后缀为.DCR（动态组件资源 dynamic component resource）。可以用 C++Builder 中的图像编辑器创建该位图。每个位图应是 24×24 像素的。为你想安装的每个组件提供一个组件面板位图文件，在每个组件面板位图文件中为你注册的每个组件提供一个位图。位图图像的名字与组件类的名字相同，将组件面板位图文件与编译过的文件放在同一目录中，这样当 C++Builder 在组件面板中安装组件时才能找到所需的位图。如果不想麻烦，也可以将资源编译到库文件中，只需要按照正常的资源创建方法创建一个和组件名称相同的位图资源，并将相应的资源文件包含在工程中。

例如，如果你创建了一个组件 TMyControl，需要创建一个包含 TMYCONTROL 位图的.DCR 文件或.RES 资源文件。资源名是大小写无关的，但是根据约定，通常使用大写。

7.3 为你的组件提供帮助

当在窗体中选择了一个了标准组件，或在对象观察器中选择了属性、事件时，你可以通过按下 F1 键获得该项目的帮助。如果你创建了相应的帮助文件，也可以为你的组件向开发员提供类似的文档，对于一个完整的组件，帮助文件是必不可少的，通常可以根据需要，提供一个小的帮助文件以描述定制的组件，帮助文件将可以成为 C++ Builder 帮助系统的一部分。

可以使用任何工具创建 Windows 帮助文件的源文件（使用.rtf 格式）。然后使用 Microsoft Help Workshop 将 RTF 文件编译成帮助文件，C++ Builder 在 Hellp | Tools 目录中提供了该工具和使用手册。

为组件创作帮助文件有下列几个步骤：

- 创建入口。
- 使组件帮助对上下文敏感。
- 添加组件帮助文件。

1.创建入口

为了能让你的组件帮助无缝地与库中其他组件的帮助集成在一起，考虑下面的约定：

(1)每个组件都应有一个帮助主题，组件的主题应显示声明组件的单元以及组件的简短描述。组件主题应链接第二个窗口，这第二个窗口描述组件在对象层次结构中的位置和它所有属性、事件以及方法的列表。应用程序开发人员通过在窗体中选中组件然后按 F1 键来访问这些主题。你可以把任意一个组件放到窗体上并按 F1 键，看一下该组件的主题。

组件主题必须有一个#脚注，＃脚注有一个用于该主题的唯一值，通过帮助系统唯一地标识每个主组件主题应该有一个 K 脚注，K 脚注用于帮助系统索引关键字，而索引中包含组件的类名。例如，TMemo 组件的关键字脚注是 TMemo。组件主题还应有一个$脚注提供主题的标题，该标题显示在 Topics Found 对话框、Bookmark 对话框和 History 对话框中。

(2) 每个组件都应包含下列二级导航主题

- 一个层次结构主题，链接到该类在组件层次结构中的每个祖先。
- 一个组件所有可用属性的列表，链接到属性描述的入口。
- 一个组件所有可用事件的列表，链接到事件描述的入口。
- 一个组件所有可用方法的列表，链接到方法描述的入口。

可以使用 Alinks 来链接 C++ Builder 帮助系统中的对象类、属性、方法或事件。当链接对象类时，Alinks 使用对象的类名，在类名后紧跟一个下划线和 object 字符串。例如，链接 TCustomPanel 对象，使用：

```
!AL(TCustomPanel_object,1)
```

当链接到属性、方法或事件时，使用对象名作前缀，紧跟着的是下划线和属性、方法或事件的名字。

例如，链接 TControl 控件的 Text 属性，使用：

```
!AL(TControl_Text,1)
```

你可以显示任意一个组件的帮助，然后单击标为层次结构、属性、方法或事件的链接，看看二级导航主题。

(3)每个在组件中声明的属性、方法或事件都应有一个主题

属性、方法或事件的主题应显示它的声明和使用描述。应用程序开发人员可以通过在对象观察器中高亮显示该项目，然后按 F1，或把鼠标放到代码编辑器中的项目名上然后按 F1 来查看这些主题。在对象观察器中任意选择一个属性按 F1，看看它的主题。

属性、事件和方法主题应有一个 K 脚注列出它们的名字以及它的名字与组件名的混合体。因此，TControl 组件的 Text 属性有如下的 K 脚注：

```
Text,TControl;TControl,Text;Text,
```

属性、事件和方法主题还应有一个$脚注显示主题的标题，如 TControl::Text。所有这些主题都有一个唯一的 ID，形如#脚注。

2.使组件帮助对上下文敏感

每个组件、属性、方法和事件主题都必须有一个 A 脚注。当用户选择了组件并且按下 F1，或用户在对象观察器中选择了属性或事件且按下 F1 时，A 脚注用于显示主题。A 脚注必须遵循下列命名约定：

如果是组件的帮助主题，A 脚注由两个被分号分隔的入口组成：

```
ComponentClass_Object;ComponentClass
```

此处，ComponentClass 是组件的类名。

如果是属性或事件的主题，A 脚注由被分号分隔的三个入口组成：

```
ComponentClass_Element;Element_Type;Element
```

此处，ComponentClass 是组件的类名，Element 是属性、方法或事件的名字，Type 指明是方法、属性还是事件。例如，TMyGrid 组件的 Background 属性的 A 脚注形如：

```
TMyGrid_BackgroundColor;BackgroundColor_Property;BackgroundCo
lor
```

3.添加组件帮助文件

使用 bin 目录中的 OpenHelp 工具（即 oh.exe），或使用 IDE 中的 Help |

Customize 把你的帮助文件添加到 C++ Builder 中。可以在 OpenHelp.hlp 文件中找到 OpenHelp 的使用说明，其中包括如何将你的帮助添加到帮助系统中。

关于生成帮助文件，并不是本书的主要内容，这里并不多讲，在本书提供的例程中，有一个 TDataPick 控件，提供了简单的帮助，从本书附带的光盘中可以找到其源文件和帮助文件的所有原始内容，读者可以参考学习。

7.4 添加属性编辑器

对象观察器为所有类型的属性提供缺省的编辑。但是，可以通过编写并注册编辑器为某一属性提供一个替代的编辑器。既可以为编写的组件属性提供专用的编辑器，也可以为某一类型的所有属性提供编辑器。

如果简单一点，一个属性编辑器可以以下列两种形式之一进行工作：显示并允许用户将属性作为文本字符串进行编辑，或显示一个允许以某种形式编辑的对话框。根据被编辑的属性的类型，你可以提供一种或两种方式。

通常编写属性编辑器有以下五个步骤：

- 派生属性编辑器类。
- 将属性作为文本进行编辑。
- 将属性作为整体进行编辑。
- 指定编辑器属性。
- 注册属性编辑器。

7.4.1 派生属性编辑器类

在 DSGNINTF.HPP 中定义了几种属性编辑器，均是从 TPropertyEditor 派生。当创建属性编辑器时，属性编辑器既可以直接从 TPropertyEditor 派生，也可以从下表中描述的属性编辑器类派生。

表 5.2 预定义的属性编辑器类型

类型	适用的属性
TordinalProperty	所有的序数属性编辑器（用于整型、字符、枚举属性编辑）均从 TOrdinalProperty 派生
TintegerProperty	所有整型类型，包括预定义的和用户定义的子域
TcharProperty	字符类型和字符的子域，如‘A’...‘Z’
TenumProperty	任意枚举类型
TfloatProperty	所有的浮点数
TstringProperty	ANSI 字符串
TsetElementProperty	集合中的单个元素，以布尔值形式显示
TsetProperty	所有的集合。集合不能直接编辑，而是展开成集合元素属性列表
TclassProperty	类。显示类名并允许扩展类的属性
TmethodProperty	方法指针，大多数值得注意的事件

TcomponentProperty	同一窗体中的组件。用户不能编辑组件的属性，但是可以指向一个适当类型的组件
TcolorProperty	组件颜色。如果可能显示颜色常量，否则显示十六进制值。下拉式列表框包含颜色常量。双击打开颜色选择对话框
TFontNameProperty	字体名。下拉式列表框包含所有当前安装的字体
TfontProperty	字体。允许将单个字体属性扩展为字体对话框

DSGNINTF.HPP 中还定义了一些用于编辑具有唯一性的属性（如组件名）的属性编辑器。列出的属性编辑器是那些对用户定义的属性非常有用的编辑器。

下面的例子声明一个名为 TMyPropertyEditor 的简单属性编辑器：

```
class PACKAGE TMyPropertyEditor : public TPropertyEditor
{
public:
    virtual bool __fastcall AllEqual(void);
    virtual System::AnsiString __fastcall GetValue(void);
    virtual void __fastcall SetValue(const System::AnsiString Value);
    __fastcall virtual ~TMyPropertyEditor(void) { }
    __fastcall TMyPropertyEditor(void) : Dsgnintf::TPropertyEditor() { }
};
```

7.4.2 将属性作为文本进行编辑

所有的属性都要为对象观察器的显示提供一个字符串标识。大多数属性还允许用户为属性键入一个新值。可以重载属性编辑器类提供的虚拟方法完成真实值和文本描述之间的转换。

重载的方法名为 GetValue 和 SetValue。你的属性编辑器还继承了一系列方法用于不同类型值的赋值和读取，见下表

表 5.3 读写属性值的方法

属性类型	Get 方法	Set 方法
浮点数	GetFloatValue	SetFloatValue
Closure（事件）	GetMethodValue	SetMethodValue
序数类型	GetOrdValue	SetOrdValue
字符串	GetStrValue	SetStrValue

当重载一个 GetValue 方法时，将调用 Get 方法中的一个；当重载一个 SetValue 方法时，将调用 Set 方法中的一个。

1.显示属性值

属性编辑器的 GetValue 方法返回属性当前值的文本描述。对象观察器在该属性的值

列中使用此文本描述。缺省时，GetValue 返回“unknown”。

重载属性编辑器的 GetValue 方法给你的属性提供一个文本描述。

如果属性不是字符串，GetValue 必须将值转换为字符串描述。

2.设置属性值

属性编辑器的 SetValue 方法将用户在对象观察器中键入的字符串转换为适当的值，然后将该值赋给属性。如果得到的字符串描述的并不是正确的属性值，SetValue 会发送异常且不使用不正确的值。

重载属性编辑器的 SetValue 方法以读取将赋给属性的值。

SetValue 应对字符串进行转换，并在调用 Set 方法之前对值进行验证。

7.4.3 将属性作为整体进行编辑

将属性作为整体编辑的意思是，提供一个对话框，当用户双击属性时或者单击属性旁边的编辑按钮（...），弹出对话框，如 Font、Color 等属性的编辑器。

重载属性编辑器类的 Edit 方法以提供整体属性编辑器对话框。

Edit 方法调用的 Set 和 Get 方法与 GetValue 和 SetValue 调用的是一样的。实际上，Edit 既调用 Get 方法，也调用 Set 方法。由于这种编辑器是类型专用的，因此通常不需要将属性值转换为字符串。这类编辑器通常将相应的值当作“接收的值（as retrieved）”一样处理。

当用户单击属性旁边的“...”按钮或双击相应的值列时，对象观察器调用属性编辑器的 Edit 方法。

在实现 Edit 方法时，遵循下列步骤：

- 构造用于该属性的编辑器。
- 读取当前值，并用 Get 方法将其赋给属性。
- 当用户选择了一个新值时，调用 Set 方法将值赋给属性。
- 销毁编辑器。

7.4.4 指定编辑器特性

属性编辑器必须为对象观察器提供信息以决定显示什么工具。例如，对象观察器需要知道属性是否有子属性或能否显示可能值的列表。

重载属性编辑器的 GetAttributes 方法指定编辑器特性。

GetAttributes 方法返回一列 TPropertyAttributes 类型的值，值为下表中值的任意一个或多个，甚至全部：

表 5.4 属性编辑器特性标志

标志	相关的方法	含义
paValueList	GetValue	编辑器可以给出枚举值的列表
PaSubPropertiess	GetPropertie	编辑器有可以显示子属性
PaDialog	Edit	编辑器可以显示用于编辑整个属性的对话框

paMultiSelect	____	当用户选择了多个组件时，该属性应显示
paAutoUpdate	SetValue	对于每个改变都自动更新组件，无需等待确认
PaSortList	____	对象观察器应对值进行排序
PaReadOnly	____	用户不能修改属性值
paRevertable	____	使对象观察器上下文菜单中的 ReverttoInherited 菜单项有效。该菜单项告诉属性编辑器放弃当前属性值并返回到预先确定的缺省值或标准值

Color 属性比大多数属性要多变一些，因为它允许用户在对象观察器通过几种方式选择：键入、从列表中选择和定制的编辑器。因此，TColorProperty 的 GetAttributes 方法在返回值中包含几个属性：

```
virtual __fastcall TPropertyAttributes TColorProperty::GetAttributes()
{
  return TPropertyAttributes() << paMultiSelect << paDialog << paValueList;
}
```

7.4.5 注册属性编辑器

一旦创建了一个属性编辑器，需要在 C++ Builder 中注册它。注册一个属性编辑器会将某一类型的属性与指定的属性编辑器联系起来。可以为给定类型的所有属性注册一个编辑器，也可以为某一特定组件的特定属性注册一个编辑器。

调用 RegisterPropertyEditor 函数注册一个属性编辑器。

RegisterPropertyEditor 有四个参数：

1.一个被编辑属性的类型信息指针。指定类型信息如下：

```
__typeinfo(TMyComponent)
```

2.编辑器应用的组件类型。如果该参数是 null，则编辑器应用于给定类型的所有属性。

3.属性名。只有前一个参数指定了某一特定的组件类型时，该参数才有意义。因此，你可以指定某一属性的名字。

4.编辑特定属性的编辑器类型。

下面的摘录为组件面板中的标准组件注册编辑器：

```
namespace Newcomp
{
   void __fastcall PACKAGE Register()
   {
     RegisterPropertyEditor(__typeinfo(TComponent), 0L, "",
```

```
                __classid(TComponentProperty));
    RegisterPropertyEditor(__typeinfo(TComponentName),
                __classid(TComponent), "Name",
                __classid(TComponentNameProperty));
    RegisterPropertyEditor(__typeinfo(TMenuItem), __classid(TMenu), "",
                __classid(TMenuItemProperty));
    //...
  }
}
```

该函数中的三个语句涵盖了 RegisterPropertyEditor 函数的不同用法：

- 第一个语句是非常典型的。它为 TComponent 类型（或没有注册自己的编辑器的 TComponent 后代）的所有属性注册了属性编辑器 TComponentProperty。通常，当你注册一个属性编辑器时，你就为某一特定的类型创建了一个编辑器，并且想把它应用于这个类型的所有属性，因此，第二参数和第三个参数分别是 NULL 和空字符串。
- 第二个语句是注册中非常特殊的一种。它为特定类型组件的特定属性注册了一个编辑器。在这个例子中，是所有 TComponentName 类型组件的 Name 属性的编辑器。
- 第三个语句比第一个语句特殊，但还没有像第二个语句那样受限制。它为 TMenu 类型组件的所有 TMenuItem 类型属性注册编辑器。

7.5 添加组件编辑器

组件编辑器是当组件在设计器中被双击时，确定作什么样的事情；而当在组件上单击右键时，组件编辑器将向弹出的上下文菜单中添加命令。它还可以把你的组件以定制格式拷贝到剪贴板。

如果你没有给你的组件设计并注册一个编辑器，C++ Builder 会使用缺省的组件编辑器。缺省的组件编辑器由类 TDefaultEditor 实现。TDefaultEditor 并不向组件的上下文菜单添加新项目。当组件被双击时，TDefaultEditor 搜索组件的属性，生成（或定位到）它找到的第一个事件处理程序。

为了向上下文菜单添加项目、改变组件被双击时的动作、或添加新的剪贴板格式，从 TComponentEditor 派生一个新类并注册它应用于你的组件。在你重载的方法中，你可以使用 TComponentEditor 的 Component 属性访问正在编辑的组件。

添加定制的组件编辑器有以下几个步骤：

- 向上下文菜单添加项目。
- 改变双击时的动作。
- 添加剪贴板格式。
- 注册组件编辑器。

7.5.1 向上下文菜单添加项目

当用户右击组件时，组件编辑器的 GetVerbCount 方法和 GetVerb 方法被调用以建立上下文菜单。你可以重载这些方法，向上下文菜单添加命令（动词）。

向上下文菜单添加项目需要下面几个步骤：

1.指定菜单项目

重载 GetVerbCount 方法返回你向上下文菜单中添加的命令数目。重载 GetVerb 方法返回向上下文菜单添加的命令字符串。当重载 GetVerb 时，在字符串中加入&将导致紧接着的字符有一个下划线，且该字符成为该选项的快捷键。如果某个命令将弹出一个对话框，那么在相应字符串后面添加一个省略号（...）。

GetVerb 有一个参数，即命令的索引。

下面的代码重载 GetVerbCount 和 GetVerb 方法，向上下文菜单中添加两个命令：

```
int __fastcall TMyEditor::GetVerbCount(void)
{
   return 2;
}

System::AnsiString __fastcall TMyEditor::GetVerb(int Index)
{
   switch (Index)
   {
     case 0: return "&DoThis ..."; break;
     case 1: return "Do&That"; break;
   }
}
```

注意确保你的 GetVerb 方法为 GetVerbCount 指示的索引返回一个值，否则在组件的弹出菜单中将增加一个空白的菜单项，这并不是组件设计员希望看到的。

2.实现命令

当 GetVerb 提供的命令在设计器中被选择时，ExecuteVerb 方法被调用。在 ExecuteVerb 中实现每个由 GetVerb 提供的命令。你可以提供编辑器的 Component 属性访问正被编辑的组件。

例如，下面的 ExecuteVerb 方法实现前面例子中的 GetVerb 提供的命令：

```
void __fastcall TMyEditor::ExecuteVerb(int Index)
{
   switch (Index)
   {
```

```
        case 0:      //执行 DoThis 菜单
          TMyDialog *MySpecialDialog = new TMyDialog();
          MySpecialDialog->Execute();
          ((TMyComponent *)Component)->ThisProperty =
                    MySpecialDialog->ReturnValue;
          delete MySpecialDialog;
          break;
        case 1:    //执行 DoThat 菜单
          That();  //调用"That"方法
          break;
      }
    }
```

7.5.2 改变双击时的行为

当组件被双击时，编辑器的 Edit 方法被调用。缺省情况下，Edit 执行第一个被添加到上下文菜单中的命令。这样，在前面的例子中，双击组件将执行 DoThis 命令。

虽然执行第一个命令通常是个好主意，但你可能想改变这种缺省行为。例如，你可以提供一个替代动作，如果：

- 你没有向上下文菜单中添加任何命令。
- 当组件被双击时，你想显示一个组合了多个命令的对话框。

重载 Edit 方法指定一个组件被双击时执行的动作。例如，当用户双击组件时，下面的 Edit 方法弹出一个字体对话框：

```
void __fastcall TMyEditor::Edit(void)
{
   TFontDialog *pFontDlg = new TFontDialog(NULL);
   pFontDlg->Execute();
   ((TMyComponent *)Component)->Font = pFontDlg->Font;
   delete pFontDlg;
}
```

注意如果想在双击组件时为事件处理程序弹出代码编辑器，那么应以 TDefaultEditor 作为组件编辑器的基类，而不是 TComponentEditor。这样，要重载的是 TComponentEditor::EditProperty 法而不是 Edit 方法。EditProperty 方法搜索组件的事件处理程序，提出它找到的第一个事件处理程序。你可以改变这一点，让其提出某一指定的事件。例如：

```
void __fastcall TMyEditor::EditProperty(TPropertyEditor* PropertyEditor,
                                        bool &Continue, bool &FreeEditor)
```

```
{
    if (PropertyEditor->ClassNameIs("TMethodProperty") &&
        CompareText(PropertyEditor->GetName, "OnSpecialEvent") == 0)
    {
      TDefaultEditor::EditProperty(PropertyEditor, Continue, FreeEditor);
    }
}
```

7.5.3 添加剪贴板格式

缺省情况下，当某个组件在 IDE 中被选中后用户选择 Copy 时，组件会以 C++Builder 的内部格式被拷贝。它能粘贴到另外一个窗体或数据模块中。通过重载 Copy 方法，你的组件可以在剪贴板中使用另外的格式。

例如，下面的 Copy 方法允许 TImage 组件将它的图片拷贝到剪贴板。C++Builder 将忽略这些图片，但它们可以粘贴到另外的应用程序中。

```
void __fastcall TMyComponentEditor::Copy(void)
{
    WORD AFormat;
    int AData;
    HPALETTE APalette;
    ((TImage *)Component)->Picture->SaveToClipboardFormat(AFormat,
                        AData, APalette);
    TClipboard *pClip = Clipboard();       // 不要清除剪贴板
    pClip->SetAsHandle(AFormat, AData);
}
```

7.5.4 注册组件编辑器

一旦组件编辑器被定义，它就可以被注册用于某个组件类。当该类的某一组件在设计器中被选中时，注册过的编辑器将被应用于该组件。

为了在组件编辑器和组件类之间建立链接，调用 RegisterComponentEditor。RegisterComponentEditor 使用组件的类名与组件编辑器的类名。例如，下面的语句注册了用于 TMyComponent 类组件的 TMyEditor 编辑器：

```
RegisterComponentEditor(__classid( TMyComponent), __classid(TMyEditor));
```

将 RegisterComponentEditor 的调用放在你注册组件用的名字空间中。例如，如果新组件 TMyComponent 和它的编辑器 TMyEditor 均是在 NewComp.cpp 文件中实现的，那么下面的代码（在 NewComp.cpp 文件中）注册了组件以及与之关联的编辑器。

```
namespace Newcomp
{
    void __fastcall PACKAGE Register()
    {
      TMetaClass classes[1] = {__classid(TMyComponent)};
      RegisterComponents("Miscellaneous", classes, 0);
      RegisterComponentEditor(classes[0], __classid(TMyEditor));
    }
}
```

7.6 将组件编译成软件包

一旦你的组件已被注册，那么在你将它们安装到 IDE 之前，必须将它们编译成软件包。一个软件包可以有一个或几个组件，如同定制属性编辑器一样。有关软件包的信息以及如何创建和编译包文件请读者参考联机帮助。将定制组件的源代码单元放到软件包包含列表（Containlist）中。如果你的组件需要其他的软件包，那么把它们放到需求列表（Requireslist）中。

7.7 解决定制组件问题

在注册和安装组件时一个常见的问题是，当软件包成功地安装后，组件没有显示在组件列表中。

最普通的导致组件未能显示在组件列表或组件面板中的原因是：

- 在注册函数中没有使用 PACKAGE 修饰符。
- 在类中没有使用 PACKAGE 修饰符。
- 在 C++源文件中没有使用#pragma package(smart_init)语句。
- 在与源文件同名的名字空间中没有找到注册函数。
- 注册函数没有成功地导出。对.BPL 使用 tdump 寻找导出函数：

```
tdump -ebpl mypack.bpl mypack.dmp
```

在转储的导出部分，你应该看到注册函数（在名字空间内部）被导出。

8 小节

本章从应用的角度讲述了如何使用 C++ Builder 中语言扩充的内容，以及一些处理技巧和经验，对于设计组件的读者来说需要和实践结合起来，在应用中参考本章内容，对于程序的开发者来说，本章部分内容也是需要了解的。

通过这一章的阅读，读者对 C++ Builder 的 VCL 会有一个更深入的了解，这将有助于后面示范程序的阅读。实际，这一章所讲的内容几乎都是和 VCL 组件技术有关的，C++ Builder 的使用包括很多方面，这里也不能一一介绍，因此 C++ Builder 的初学

者应该结合联机帮助或者其它书籍进一步深入了解。

下一章我们通过讲解几个实际组件设计的例子来运用和掌握本章以及前面章节所介绍的内容。