Lewolf
保留版权
签名未验证

第三章 面向状态的程序设计思想

从本章开始将介绍面向状态的程序设计思想和一些常用的方法，思想是关键，方法是途经，因此建议读者需要仔细阅读本章。从本章开始，将会根据不同的情况据一些简单的例子，来讲解面向状态的思想和方法，有些甚至于是比较弱智的例子，而采用常规方法或者面向对象的方法可能会更简单一些，但请读者一定要注意，这里的例子仅仅是为了体现如何实现面向状态，所以在例子上力求简单、明了，正如我国著名的象棋棋谱《局中密》一样，看似平淡无奇，近似弱智，但是按照棋谱一步一步深入下去，若能逐一旁通，便会发现其中奥妙无穷，受益匪浅。本书也希望通过一些简单的例子能给读者了解面向状态提供最简捷的途径，通过练习，举一反三而最终达到熟练掌握的目的。

1 从 Hello World 开始

很多计算机书籍在介绍一种开发工具时，总会从 HelloWorld 开始，本章也将沿袭这一习惯，从一个稍微复杂一点的 HelloWorld 开始，给读者建立一个面向状态的感性认识，然后在一步一步的深入问题，最后从理性的角度认识并掌握面向状态的技术。

本书面对的读者是对 C++ Builder 有一定基础的，因此在这个例子中，涉及到 C++ Builder 的一些语法、规则等不多介绍，而重点放在与常规不同的实现方法上，不熟悉的读者应该先阅读有关 C++ Builder 的书籍。此外，这个例子非常简单，但却贯穿这本章内容的重要思想，因此建议读者不要因为是一个弱智的例子而轻易越过。

面向状态并不是一个新的理论，而是在应用的早已出现的概念，尽管很多书籍都没有这样称呼，但这是一个存在的事实。如果你使用过 Visual Basic 或者 Delphi 或者 C++ Builder，那么可以肯定你已经使用过面向状态的技术，前面第一章我们也说过，面向状态正是从这些可视化编程工具中产生的，只不过那时候你可能只是被动的使用，而没有真正的领会其中的奥妙和思想，更谈不上主动的使用面向状态的技术。接下来的内容将让你全面的了解面向状态的技术，并教会你如何主动使用它。

1.1 问题与分析

本例要实现的功能如下：

(1) 在屏幕上显示一个窗体，窗体中包含一个 Button 和一个 Label，Label 显示“HelloWorld!”

(2) 当鼠标点击 Button 时，窗体底色由银灰色变为白色，并且 Label 的字符串变



图 3.1 HelloWorld 的运行界面

为“This is New State HelloWorld”，当再次点击 Button 时，窗体及字符还原。



图 3.2 HelloWorld 的新状态

如图 3.1 和图 3.2 所示：

对该例进行分析，不难发现，当鼠标点击 Button 时，Form 中有两项内容需要改变，一是 Form 的底色，另一个是 Label 显示的字符串，而且这两项内容始终相关联在一起，当 Form 底色为银灰色时 Label 为“HelloWorld!”，当 Form 的底色为白色时，Label 为“This is New State HelloWorld”，我们可以认为这是 Form 的两个不同状态，而且在鼠标点击了 Button 以后，程序必须知道到前 Form 处于什么样的状态，然后决定什么样的操作。按照常规的做法，我们可能需要定义一个 bool 变量：IsNewState 来记录当前 Form 的状态，在 OnButton1Click 事件中写入要执行的代码，尽管我们可以利用 Form 的颜色来判断 Form 当前处于什么样的状态，但这种做法显然不够规范，而且是一种不负责任的做法，在复杂的程序中是很危险的。

今天我们将采用与上面做法稍微不同的实现方法，这就是面向状态的方法，从表面上看，代码变动不大，而且似乎更加冗长，但实际上这是一种根本的变革，我们先看采用新的方法是如何实现的。

1.2 Hello World 的实现

本例子中只包含一个 Form，Form 中可视化构件只有一个 Button 和一个 Label，其它成员是手工创建的，因此我们并不给出 Unit1.dfm 文件，只给出 Unit1.h 和 Unit1.cpp，源文件如下：

Uint1.h 文件

```
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------
class TForm1 : public TForm
{
__published:
    TButton *Button1;
    TLabel *Label1;
```

```
    void __fastcall Button1Click(TObject *Sender);
private:
    bool FIsNewState;          //定义一个Bool变量，记录State
protected:
    void __fastcall FSetIsNewState (bool Value);
                               //为FisNewState创建设置函数
public:
    __property bool IsNewState = {read = FIsNewState,
                                  write = FSetIsNewState};
                               //定义一个属性（Property）供程序使用
    __fastcall TForm1(TComponent* Owner);
};
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------
#endif
```

Unit1.cpp 文件

```
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
                  : TForm(Owner)
{                 //一个空的构造函数
}
//---------------------------------------------------------------------
void __fastcall TForm1::FSetIsNewState (bool Value)
{
    if( Value == FIsNewState )
      return;
    if( Value )
    {
        Label1->Caption = "This is New State Hello World";
        Color = clWhite;
```

```
    }
    else
    {
        Label1->Caption = "Hello World!";
        Color = clBtnFace;
    }
    FIsNewState = Value;
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    IsNewState = !IsNewState;      //令属性 IsNewState 与原来相反
}
```

在这一段代码中，有几个明显的特点：

（1） Button1Click 中没有实际操作的代码，而仅有一条赋值语句。

（2） bool 变量 FIsNewState 为私有成员，不可以被其它对象直接访问。

（3） 多含了一个 IsNewState 属性（Property）。

（4） 多了一个成员函数 FSetIsNewState。

Property 是 C++ Builder 中特有的一个关键字，在英文中的含义为属性、道具、财产、所有物、性质、特性等，在这段代码中理解为道具等为合适一些，但大多数的书籍中均将其翻译为属性，为了保持一致，我们在本书中也称之为属性。

关于__property 关键字以及其用法我们在后面章节中详细讲解，这里我只要知道 IsNewState 是一个属性，它并不是实实在在的变量，也就是说程序并不会给它分配内存空间，使用 IsNewState 属性的目的是：程序员可以将其当成一般变量一样给其赋值。而这个赋值过程会通过 FSetIsNewState 函数最终反映在 FIsNewState 变量中。读者可以验证 Button1Click 中的代码：

```
IsNewState = !IsNewState;       与
FsetIsNewState(!FisNewSate);
```

在编译后是完全相同的，从使用的角度讲，第一种方法显然直观得多，但这并不是最根本的变革，最根本的变革是 Button1Click 摆脱了与其相关性较差的代码，而将这些任务交给了属性 IsNewState，由其相关的配套函数 FSetIsNewState 在背后进行，属性 IsNewState 不仅记录了当前 Form 所处的状态，还将这个状态的表现行为与之紧密相连。因此程序员可以分别的去考虑在什么样的状态下程序有什么样的表现行为，以及在什么时候要改变程序的状态。这就是面向状态的方法。

实际上，这个例子中除了用户定义的 IsNewState 属性以外，Color 和 Caption 也分别是 Form 和 Label 的属性，因此在 FSetIsNewState 函数中，也基本都是赋值语

句，而真正的执行中，Color 的改变除了改变保存 Color 真实变量的值以外，还需要对 Form 进行重绘，Caption 的赋值也需要对画布的文字进行重新输出。可以想象，如果没有这种机制，要实现这样一个简单的过程并不是那么简单。而且代码之间的逻辑也会变得相对复杂化。

2 什么是面向状态

上一节我们从一个简单的例子接触了面向状态的程序设计，这一节将讲述面向状态的一些概念和思想，在了解面向状态之前，首先从状态来了解起。因为程序设计实际上是将人类日常活动计算机化，让计算机代替人类作一些事务，所以，程序设计和自然物质世界是分不开的，和人类的思维过程是分不开的，这一节我们先从状态的概念了解起。

2.1 状态的概念

状态在面向状态的程序中有着特殊的地位，如同面向对象一样，状态也是对自然界的一种抽象，或者是一种描述方法，而状态也和对象有着非常紧密的联系，下面我们将从自然界对状态的定义开始学习。

2.1.1 什么是状态

状态在自然界是一个非常广泛的概念，指的是物体所处的状况和形态，可以是动态的，也可以是静态的，比如物理上可以是指物体的运动状态，这是一个动态的概念，也可以是指物体的通电状态、发光状态等，这些都是静态的。更广义的理解比如：心理状态、身体状态、竞技状态等都是自然界对状态的定义。

从另一方面来看，状态又是一种抽象的概念，只有和人类的思维结合在一起才有实际的意义，比如物体的运动状态，并不属于客观物质世界实实在在存在的物质体系，它不像一棵树、一根草那样，是一个客观存在的物质，而是经过人类抽象而形成的一个概念，而它的存在性又是客观的，并不会因为人类没有意识到运动的状态时，物体就不会运动。

从这两方面观察，我们可以这样理解状态：

状态是描述事物动态特征的变量，从上面的例子我们可以看出，无论这状态是动态的还是静态的，它所描述的都是物质世界经过某种运动或者作用后的结果，比如运动状态是描述物体受到外界力量以后的表现行为；心理状态可能会受到自身或者外界的某种影响而改变。总之状态所描述的都是事物的动态特征。

将这一点引申到程序设计中，不难发现在程序设计中，状态的动态特征更加明显。在程序的运行过程中仔细分析不难发现，每个过程的运行结果都可以是多个状态的组合，比如一个绘图程序，程序的状态可能是某个按钮有效或者无效，屏幕显示某个对象被选中或未选中，如果用鼠标点击某个图形对象，将其选中，这时屏幕显示图形被选中，同时“剪切”、“删除”、“拷贝”等按钮将变成有效，当未选中任何图形时，这些按钮则是无效的。记录或者表示一个按钮是否有效的变量便是程序中的一个状态，而这些状态最终反映的是程序的动态特征。

如果说对象和类是认识客观世界一个静态的抽象，那么状态则描述的是客观世界的一个动态过程。

状态是事物本身的特征，正如前面所提到的，不管你是否认识到事物处于某种状态，也不管对它的状态是否有定义，它的状态都是客观存在的，因为物质由于运动而改变状态也是可观存在的，因此说状态是事物本身的特征。在程序设计中一样，程序的运行状态也是客观存在的，但是对于状态的抽象却可以是多种多样的。在自然界人们首先认识的是事物，其次才是事物的状态：只有对“车”先有了定义，才可能认识到“车的运动状态”。在程序设计中，只有在采用了面向对象技术之后，状态才有了更加精确的含义，状态应该是某个对象的状态，没有对象的抽象就没有状态更精确的抽象。在一个程序设计过程中，可以有多种不同的对象的抽象方式，因此也具有多种状态的抽象方式。

状态是程序设计中必不可少变量，不管程序设计经过了怎么样的发展，状态变量始终是程序中不可却少的变量，即使在机器代码时代，程序中也会有一些环境变量来记录当前程序运行的一些情况，也可以将这些环境变量认为是状态——属于整个程序的状态，只是其含义和复杂程度比面向对象以后的状态要弱得多。

到了面向对象产生以后，状态的必要性越来越明显，因为程序本身是一个运动的过程，需要时刻记录当前的运行状况，尤其是在图形界面出现以后，程序的状态就显得更加明显。现在众多的软件开发工具中，尤其是图形化的开发界面，状态更是一个不可缺少的变量。

2.1.2 状态和对象的关系

前面已经对状态的概念有了一定的了解，对于程序设计而言，必须明确状态在程序中的作用和关系。

在第二章介绍面向对象的时侯，已经讲过，程序的设计过程实际上就是一个抽象过程，将所要完成的任务通过适当的模型进行描述，然后再用计算机语言表达出来，在后完成所需要的功能。而这个模型的建立过程很大一部分内容就是对象的抽象过程。

根据前面我们对状态的理解，可以看出在程序设计中对象和状态有着以下关系：

状态属于对象的特征：正如前面所说的，面向对象是对客观事物的一种静态描述，对象的成员是类的固有属性，包括成员变量和成员函数；而状态恰恰描述的是对象动态的特征，是在程序运行中所处的不同状况。因此状态也是对象的固有特征，比如一个对象包含着一个整型的成员，当这个成员被赋值为“0”或者“1”时，这个对象将处于两种不同的状态。

有对象必然有状态：

状态是一种事物运动的一种抽象表达，就像上一个例子，有了对象。有了对象的成员，就已经涉及到了这个对象的状态，这是客观存在的。在面向对象中，我们可能将精力主要放在成员和成员函数的关系上，并没有太多的关注这个对象的状态，或者说并没有将对象的状态放在第一位，而在面向状态的方法中，可能会将更多的精力用来分析对象的状态以及处理对象和对象之间的状态的相互关系。

面向对象从面向对象发展而成的：面向状态并不是开始就有的，而是在面向对象技术产生之后才发展起来的，当一个程序的对象比较多的时候，各个对象之间的联系会变得非

常复杂，任何一个操作，或者操作的结果都会影响到其他的对象，仅仅采用面向对象的分析方法，要将一个程序中所有的逻辑关系表达清楚会有一定的难度。面向状态就是在这种情况下产生的，从后面的例子中我们会发现，如果采用了面向状态的分析方法，可以将程序中的逻辑关系简单化，使一些复杂逻辑关系转化为相对简单逻辑，从而使程序更加清晰、明了，也更容易维护。

状态具有一般成员的特性又区别于一般成员：尽管在程序中存在对象，就有对象的状态，但是并不是所有的成员都需要用对象的状态来描述。在这里我们可以先将状态理解为也是对象的成员，但它又有区别于一般成员的特点，在后面的内容里将要介绍 C++Builer 中类的“属性”，通过对“属性”的了解和应用，会进一步理解状态和对象的关系。

2.1.3 状态的分类

作为对象的一个成员，状态可以是任意的变量类型，包括编程语言本身固有的数据类型和用户扩展的数据类型，甚至是其他的对象，但从生存周期和表现形态上大致可以分作以下几种类型：

持续状态和非持续状态：

所谓持续状态，指的是状态的生存周期比较长而已，比如汽车在匀速行使的过程中，其运动速度就是一个持续状态，而此时的位置就是一个非持续的状态，是在不断的在变化，同样的还有一些状态为瞬间状态，也属于非持续状态，比如汽车的点火过程，关车门的过程等。

在程序中也有类似现实中的持续状态和非持续状态，比如一个对象的颜色，可能是持续不变的，这时候颜色便是一个持续的状态，也可能是在连续变化的，这时候颜色便是一个非持续状态。再比如按钮是否被按下这个状态也有不同的表现，一种情况是按钮本身自己可以弹起来，当鼠标键按下时，按钮也按下去，当鼠标弹起时，按钮也弹起来，这种按钮的状态便是一种非持续状态；另一种按钮是自己不能弹起来，鼠标点击后按钮被按下去，只有再次点击该按钮时或者其他按钮被按下时才能弹起来，这种按钮的状态就是一种持续状态。

实际上持续状态和非持续状态是相互关联的，位置的匀速移动实际上表明物体有一个恒定的速度。也就是说一种持续状态可能是另一种非持续状态的运动结果，而一个瞬间状态的作用可能使一个持续状态发生改变。这在数学上表示为运动函数和其导数的关系。

连续状态和离散状态：

所谓连续状态和离散状态是从状态的表现特征上分析的结果，很多状态，象速度、位置等都是一种可以连续取值的量，属于连续状态；而发动机状态、车门状态等只有两种或者几种有限的状态取值，属于离散状态。

由于计算机的特点，在程序中不管是连续的状态还是离散的状态，要对其进行处理或者建立约束关系，都要转化成离散状态。在前面举的汽车的例子中，就很明显，汽车的速度是一个连续的状态，而和车门状态、发动机状态建立约束关系时，只能用车速为不为“零”或者车速“大于”多少和“小于”多少等来描述，这实际上就是将连续状态转换为离散状态。

这一特点实际上也表明了计算机程序设计的另一个原则：建立约束和反约束关系的条件只能是持续状态。比如车门的状态对车速是一个约束条件，而车速则对车门的状态是一个反约束条件，我们建立它们之间的约束和反约束关系时，只能表示为车速大于零时，车门必须是关的，车门为打开状态时，车速必须为零。在这个过程里将车速表示为离散状态的同时也是将车速的状态转化为持续状态来表示，因为不管车速怎么样变化，车速是否为零的状态都将是一个持续的状态。

对于状态的分类这只是一个大致的分法，是从状态的表现特征和生存周期来区分的，而且这些不同类型的状态也可以相互转化，在程序不同的运行时刻也可能属于不同的类型。从另一方面讲对于状态的抽象是根据实际需要进行的，跟程序设计者的经验和习惯都是相关的。习惯上在程序设计时，总是将一些持续的状态称之为“属性”，而将另一些非持续的状态称之为“事件”，“事件”实际上也是一种特殊的“属性”，但不能作为约束条件，而是引发一些动作，或引起某些状态的改变。这也正好表明了“属性”和“事件”之间的数学关系。

2.2 状态的特征

前面我们从自然界对状态的定义开始分析，已经逐渐了解在程序中的状态的基本含义，但在接受面向状态的概念之前，我们还需要对状态的特征进行更深入的了解，因为程序设计中要处理好各种状态之间的关系，就需要找出更适合计算机的描述方法。

尽管程序设计是对自然界事物的运动规律采用计算机语言描述而形成的，但程序必然有不同于自然界事物运动的特征，有其自己独特的规律，计算机程序设计中的状态也和自然界中的状态有着不同的地方。下面所讲的状态的特征，就是指在程序设计中所谓状态的一些主要特征，它并不违背自然界对状态的定义，但又不等同于自然界对状态的定义。

2.2.1 状态是抽象的概念

对于状态的抽象这一特征，我们已经不止一次的提到，在程序设计中状态的出现也是经过抽象而形成的。

在本章一开始我们举的例子 Hello World 中 IsNewState 就是一个状态，它记载着当前程序所运行的一种“状态”，这个状态所涉及的只有 Form 的颜色和标题字符串的内容，这是一个简单的例子，但是从这个例子中可以看出：Form 的颜色和标题的内容本身并没有什么联系，但是在程序运行过程中，它们是被联系在一起的，是同步的呈现着不同的表现形式，这种表现形式经过抽象便是我们所看到的 IsNewState。显然，这个抽象过程使得程序看起来更加“复杂”，但整个程序的结构会更加清晰，而且，IsNewState 本身也封装了这个状态的具体实现过程，甚至包括这个状态所要成立的条件。

2.2.2 状态描述的是动态特征

前面已经说过，状态本身描述的就是一个动态的特征，这一点在程序设计中也是这样的，前面也举过一些例子说明程序中的状态是描述程序的动态特征，从 Hello World 的例子中我们也可以看出，IsNewState 和程序的运行是相关的，IsNewState 本身是一

个变量（严格的讲，这并不是一个变量，在 C++ Builder 中冠以__property 关键字，我们通常称之为属性，在程序中可以当变量一样使用，但程序并不为其分配内存，在下一章将有详细的将解），它并不是程序运行的“数据”，而是记录着程序运行的状况。如果程序运行的自始至终 Form 都将是一个颜色，标题也不会变化，那么 IsNewState 就没有任何实际意义。IsNewState 不同于一般变量的另一个特点是，它还封装着状态的实现过程，这一点我们称作面向状态的封装性，在本章的后面和下一章将作以详细介绍。

2.2.3 状态之间存在约束关系

状态是描述程序的动态特征，状态之间的约束关系实际上反映的是程序运行过程的相互制约关系，及在什么样的条件下可以进行什么样的操作，或者要进行某种操作需要具备哪些条件。在 Hello World 例子中 IsNewState 实际上没有任何约束条件，但它却是另外状态 Form1->Color 和 Lable1->Caption 这两个状态的约束条件。前面我们也说过，Color 和 Caption 实际上也是一种“属性”，在 Form 和 Lable 的头文件中均被冠以__property 关键字。Color 和 Caption 这两个属性状态要受到 IsNewState 的制约。

状态之间的约束关系也存在着好多种类型，在 Hello World 中的 IsNewState 对 Color 和 Caption 实际上是一种不完全的约束关系，因为 Color 和 Caption 除了受到 IsNewSate 的制约之外，本身还有一定的自由度，也就是说程序还可以在其他的条件下单独的改变 Caption 或者 Color，这也使得 IsNewState 成为一个不完整的约束条件，因为 IsNewState 不能完整的反映出程序所处的“状态”。

在一些更为复杂的关系中，状态之间不仅有约束关系，还存在反约束关系。比如在一个程序中可能需要操作员登录，之后才能根据其权限进行其他操作，象数据录入、查询等，这时登录权限就是进行数据录入和查询的约束条件，同时查询和数据录入也会是登录的一种反约束条件，要退出登录状态必须先退出录入或者查询状态。

从这里也可以看出利用这种约束关系来分析程序的运行，会使其逻辑更加简单一些，这也是我们为什么要进行面向状态程序设计的原因之一。

2.2.4 状态之间存在层次关系

状态之间的关系本身可能是非常复杂的，对一个应用软件而言或者一个程序模块，其中的状态可能也是成千上百的，而这些状态之间的关系也是有密有疏。划分状态之间的层次是为了更好的分析状态之间的关系。

前面所讲到的登录和数据录入、查询便是状态层次关系的一个很好的例证，登录是数据录入和查询的必要条件，那么从程序整体上讲，登录是更高一层次的状态，而数据录入和查询则是低一层次的状态。

建立层次的目的是为了简化分析过程和相互关系，因此在状态的层次结构上应该符合以下原则：

- 状态的层次是以其所辖范围或者生存周期进行划分的，这实际上已经包含了一种制约的关系。从这一角度讲高层次的状态对低层次的状态应该具有正向的约束

力。

- 只允许相邻层次之间的状态或者同一层次之间的状态存在直接的相互作用关系。这一要求实际上是为了使状态之间不会存在环形约束或者复杂的约束关系。
- 允许低层次状态对高层次状态具有反约束作用。

在这几个原则下建立状态的层次结构，或者是在程序建立各种约束关系时遵循层次之间的这几个原则，则可以使程序各个状态之间的关系相对简化，使程序的复杂逻辑变为相对简单的逻辑关系，也可以避免程序中出现的一些逻辑漏洞。

2.3 面向状态的概念

前面已经讲述了状态的定义、概念和特征，以及状态在程序中的一些表现特点，接下来的内容将要介绍本书的核心问题——面向状态的程序设计。

所谓面向状态就是指在程序设计中，状态作为设计者和程序之间的第一界面，然后围绕着状态的各种关系和逻辑结构，最终形成一个满足要求的应用程序。正如在面向对象的设计中，程序员始终是将对象和类放在了第一位，一切都是围绕着对象来完成的。但是，面向状态是由面向对象发展而来的，在面向状态的设计方法中，必不可少的需要以对象作为基础。所以，面向状态的设计方法需要具备以下两个基本条件。

2.3.1 以状态为核心

在本书的第二章，我们介绍过几种不同的程序设计思想，其中有一些是把过程放在了第一位的，有的是将数据放在了第一位，但在过去的这些经典思想中，基本还停留在一个较低层次。直到面向对象的技术才将数据和过程有机的融合在了一起，通过数据和方法的封装、对象的继承等技术手段，实现了现实空间和计算机语言的进一步统一，使计算机更加真实的描述了客观事物，但这些并没有能够完全的对事物的运动特征进行详细的描述，因为在面向对象的技术中始终是以对象为核心的，而构建对象也是程序开发的很大一部分内容，尽管很多的 SDK 中包含了非常强大的类库可供使用，但是对这些类库的使用仍然是以对象为核心的，前面我们也讲过，面向对象的技术很难将程序运行的状态进行动态的描述，这就出现了本书需要介绍的面向状态的技术。

针对程序运行的动态特征我们称之为状态，在面向状态的技术中就是以状态为程序开发的核心，程序设计者的主要精力将是放在对与程序状态之间的关系的处理上。而面向状态的技术是从面向对象的技术发展而来的，它本身具有面向对象的特征，但和面向对象又有很大的区别。在面向对象技术中，对象的接口是一些公开的成员和公开的成员函数，而在面向状态的技术中，一个对象的对外接口可能主要是由状态来构成的，这些状态在 C++ Builder 中通常称作属性（property）或事件（Events），实际上事件也是一种特殊的属性，它描述了对象状态的另一种特征，在后面的章节中我们详细讲解。总之在面向状态的技术中，程序设计是以状态为核心而进行设计的。

在 C++ Builder 中对象的对外接口还有一种就是方法（Method），熟悉数据库的读者都知道 Table->Active = **true** 和 Table->Open()执行后的效果是等价的，但这是两种不同的接口，这两种接口所不同的是如果采用方法（Method）作为接口，其执行

结果可以是带返回值的,程序员可以通过代码判断执行结果是否正确,而属性(Property)的设置，是没有返回值的。但是在 C++ Builder 中并不提倡使用方法的返回值作为判断执行结果的依据，而是通过“抛出例外”来处理程序遇到的错误，因此 C++ Builder 中 Table->Open()方法实际上是一个没有返回值的函数,上面的 Table->Active=**true** 和 Table->Open()是两个完全等价的操作。

2.3.2 以对象为基本构成单位

前面也已经说过，面向状态是从面向对象的技术发展扩充而形成的，其本质上还是基于对象的程序开发，但不同的是更注重于程序运行时各个状态之间的相互关系，并且采用了一定的新技术，使得对象之间的关系更加清晰，逻辑关系变得更加简单，使程序更符合客观事物的本身规律，从而最终提高程序的性能和开发速度。

状态并不是孤立的存在着，是对一个对象的特征抽象而生成的，因此状态必须是依附于对象而存在的，离开了状态的主体谈状态是毫无意义的。在 C++语言中对象和类相互依存的，但从更广义的角度理解，一个运行的程序也是一个对象，是 Windows 中的一个对象，这个对象由很多的元素组成，每一个元素也就是一个更小单位的对象。比如一个程序可能由好几个 Form 组成，每一个 Form 本身就是一个对象，而这个 Form 中可能又包含了很多个组件……

总之在面向状态的程序设计中，对象是一个程序的基本构成单位，这一点和面向对象是一致的。

实际上从广义的角度理解对象，可以认为对象就是一种数据类型或者数据结构的实例，尽管在 C++语言中所有的类都被冠以 **class** 关键字声明，但是没有冠以 **class** 关键字的数据类型，甚至是 C 语言的基本数据类型比如 **int**、**double** 等都可以理解为一种“类”,只是这些‘类’属于最最基本的‘类’,而且没有任何独立于外界的操作,不像 **class** 定义的类具有成员函数，也不具备继承、封装、多态等特性。举一个现实中更加形象的例子，仍然以汽车为例：

组成汽车的对象有车身、车轮、发动机等，而车身、车轮、发动机又都是由更低层次的对象构成的，如发动机是由气缸、活塞、曲轴、气门等构成……直到最后无法再细划分的对象，象铆钉、螺丝、活塞环、气缸盖等这些基本的零件就好像 C 语言中基本的数据类型一样。在面向对象之前的程序设计如同交给你一堆基本的零件：铆钉、螺丝、活塞环、气缸盖……然后统一组装成一辆汽车；而在面向对象的程序设计中好比将这些基本零件在各个车间组装成本成品：车轮、发动机、车身等大的构件之后，再集中起来总装成一辆完整的汽车；那么面向状态的程序设计如同给车加装了 ABS 防抱死、自动换档、故障报警的功能。这个比喻并不是十分切切，但可以帮助读者理解对象在面向状态中的地位。

2.4 面向状态的优点

面向状态的技术并不是最近才出现的，而是在可视化编程出现以后就已经发展起来的。最早的面向状态应该算是 Microsoft 的 Visal Basic，但 VB 和 VF 一样属于解释性编译器，并不是真正编译器，直到 Borland 推出了 Delphi 之后，面向状态才真正的

深入到了编程语言中，C++ Builder 是 Borland 在 Delphi 之后推出的基于 C++语言的可视化编译器，其性能已经得到大家的认可，我们就 C++ Builer 来看一下使用面向状态都有那些优点。

2.4.1 消除相关性

所谓相关性就是指相互之间的关联，其主体可以是两个对象、对象和外界或者对象内部的属性或状态等。换句话说：消除相关性就是使对象更加独立，尽量减少外界对自身的限制。

在本章一开始的例子：HelloWorld 中，属性 IsNewState 的创建就是一个很好的消除相关性的例子，Button1 的 OnClick 事件仅仅是 Button1 响应鼠标而触发的一个事件,其处理函数 Button1Click 和 Label1->Caption 以及 Form->Color 并无直接关系，程序员想要做的事当鼠标点击 Button1 时改变 Form 的状态，如果没有定义 IsNewState 属性,显然 Label1->Caption 和 Form->Color 都要和 Button1Click 联系在一起，这就增加了程序之间的相关性。相反的，定义 IsNewState 恰恰是将必须关联在一起的 Label1->Caption 和 Form->Color 抽象成一个独立的状态，而且 IsNewState 属性还封装了其状态改变的条件和实现方法,这样不仅消除了相关性还使程序的结构更加清晰、明了。

再比如 TCanvas 类就是一个很好的例子,TCanvas 对象的 Handle 属性实际上是一个 DC（device context）句柄，在 Windows 程序中 Device Context 是必须谨慎使用的一个资源，因为 Windows 中的 DC 资源是有限的，每个程序使用以后必须在适当的时候将其释放，不仅如此，API 函数中对于 Device Context 的使用还有诸多的限制，因此程序设计时必须考虑的面面俱到，否则很容易出现错误。而使用了 TCanvas 对象以后，你完全不必担心 Device Context 资源是否会被耗尽，对于 Brush、Pen 等绘图的对象也不必担心出现冲突，这些在 C++ Builder 中早已被封装在 TCanvas 的内部，你只要专心去设计如何在 Canvas 上绘制图形就行了。

另外一个典型的例子是 TWinControl 的 Handle 属性，这是一个窗口的句柄，熟悉早期 Windows 程序设计的读者都知道，一个窗口在调用 API 函数 CreateWindow 之前，其句柄并不存在，而错误往往就出现在这里，程序要保证每一次访问窗口句柄的时侯该句柄都是有效的，将会是非常繁琐的事情。但是使用 TWinControl 及其派生类就不会出现这样的问题，C++ Builder 中属性的特点可以保证任何时候调用 Handle 都会得到一个有效的句柄，因为属性 Handle 封装了其实现的条件和实现过程，若该窗口没有被创建，则对象会自动调用 CreateWindow 来创建窗口，并设置 Handle 为相应的句柄。

可能很多读者都注意到在 C++ Builder 中使用属性时，只需要关心发生直接关系的内容就行了，而不必要关心一个属性改变状态时所需要具备的条件和到底是怎样实现状态变化的。这就是消除相关性带来的直接的便利。假如没有 C++ Builder 中属性的这种特点，在 HelloWorld 的例子中，Button11Click 函数不仅仅是对 Color 和 Caption 进行设置，至少还应该向窗体发送一个刷新的消息，或者有一个刷新函数的调用。

2.4.2 简化逻辑关系

消除相关性的同时，就等于已经化简了逻辑关系。从 HelloWorld 这个例子中也可以看出，IsNewState 是一个可要可不要的属性，但是要和不要的结果却相差很大，相比较下，使用属性 IsNewState 无疑是增加了一个中间环节，但这个环节的加入不仅是消除了相关性而将 IsNewState 独立的抽象出来，更重要的是抽象出 IsNewState 后，该属性和其周围的元素的逻辑关系将变得非常简单，在这个例子中，简化逻辑关系的效果也许并不明显，我们来看一个逻辑稍微复杂一点的例子。

比如一个音频播放器，可能会有六个功能按钮：暂停、停止、快进、播放、快退、录音。这几个按钮的功能和相互作用关系分别如下：

暂停按钮：是属于一个相对自由的按钮，无论播放器处于播放、录音、快进、快退还是未工作状态，该按钮均可以被按下，按下以后保持状态，而且暂停按钮只能通过再次按下时自行弹起来。

停止按钮：该按钮和暂停按钮互不影响，并且只有后面四个按钮中的一个被按下时才有效，否则按钮为无效。停止按钮被按下以后自行弹起，状态不保持。

快进按钮：允许在未工作状态和播放状态被按下，因此除了录音状态始终有效，暂停按钮对其不起作用。快进、播放、快退、录音四个按钮之间为相互排斥的按钮，即不能被同时按下，但允许同时弹起来。

播放按钮：除了录音状态下，该按钮均是有效的，但不同的是如果暂停键被按下的化，播放按钮并不动作，而是处于预动作状态，诸如打开音频设备，准备音频文件等工作都已进行完毕。

快退按钮：同快进按钮一样，值时操作的内容相反，暂停键也对快退不起作用。

录音按钮：该按钮和播放按钮类似，当系统处于播放状态时不能进行录音即录音按钮为无效状态，同样暂停键按下时，录音工作并不真正开始，而是准备好所有的录音条件。

这是一个非常典型的状态相互约束的问题，诸如此类的比如 Edit 菜单中的 Delete、Paste、Copy、Cut 等都和是否选中内容以及剪切板的内容有关。而处理这类问题往往会出现顾此失彼的现象，如果一个程序中存在大量的按钮、菜单，它们之间又有着各种各样的约束关系，处理起来就会显得非常繁琐，这是由于各种对象之间的逻辑关系复杂化而造成的，我们试着采用分析状态的方法来对同样一个问题进行分析。

针对这个例子，程序的状态之间的影响可以分作两种类型，一种是按钮的状态和运行状态之间的关系，也就是说在什么情况下，那个按钮是有效（Enable）的，在什么情况下又是无效的；另一种是在不同的运行状态下对按钮动作所产生的影响。按钮的状态和程序运行状态之间的关系可以用下表 3.1 和 3.2 来表示：

按钮有效性（Enabled 属性）和程序运行状态之间的关系

按钮	运行状态					
	播放	快进	快退	录音	停止	暂停
暂停	1	1	1	1	1	—
停止	1	1	1	1	—	1

快进	1	—	1	0	1	1
播放	—	1	1	0	1	1
快退	1	1	—	0	1	1
录音	0	1	1	—	1	1

表中“0”表示无效，“1”表示有效。

表 3.1 按钮有效性和各按钮状态的关系

按钮动作影响的状态和程序运行之间的关系

按钮	运行状态					
	播放	快进	快退	录音	停止	暂停
暂停	1	0	0	1	0	—
停止	1	1	1	1	—	—
快进	1	—	1	0	1	1
播放	—	1	1	0	1	0
快退	1	1	—	0	1	1
录音	0	1	1	—	1	0

表中“0”表示未产生动作，或者动作执行不彻底，“1”表示动作彻底执行。

表 3.2 按钮动作和各按钮之间的关系

比较两个表格，不难发现只有暂停的行和列不相符合，其他的行和列的值均完全一致，这正说明了暂停和运行状态的关系更加复杂。另一方面这两个表格也体现出了状态之间的另一种关系，就是约束的层次关系，显然按钮的状态对按钮的动作所影响的状态有约束力，从层次上讲按钮有效性的状态比按钮的动作影响的状态高了一个层次，因为只有一个按钮有效时才可能产生动作，在上面两个表中表现为只有第一个表中为“1”的单元格，第二个表格中才可能为“1”。

除了这两个表描述的相互关系之外，按钮还有一个状态就是是否被“按下”，有些按钮被按下的状态是保持的，除了“停止”按钮以外，其他的按钮都是这样，但是它们弹起来的条件却是不一样的，暂停按钮是自己弹起来的，另外四个可以相互弹起来，也可以被停止按钮弹起来，这几种关系可以通过设置 SpeedButton 的 Down 属性来实现，而且也比较简单，这里不列出来了。

根据上面的分析，我们可以设想定义六个属性来完成这些功能：WorkState、IsPause、IsPlay、IsPrev、IsNext、IsRec。WorkState 用来记录工作状态，其他几个属性用来记录按钮操作后的影响的状态，为 WorkState 约束的状态。

整个程序包括三个.CPP 文件，分别为：Player.cpp、Unit1.cpp、PropertyFun.cpp。

其中 Player.cpp 为程序的主文件，Unit1.cpp 为 Form1 的单元文件，PropertyFun.cpp 为一个辅助 cpp 文件，将建立属性的一些受保护的函数放在这个单元，这样作的目的是将属性的建立和属性的使用分开放置，有利于设计，当然也完全可以

将它们放在一个文件中。

程序的代码如下：

资源文件“Unit1.dfm”

```
object Form1: TForm1
  Left = 284
  Top = 133
  Width = 297
  Height = 77
  Caption = 'Player'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -13
  Font.Name = 'MS Sans Serif'
  Font.Style = [ ]
  OldCreateOrder = False
  OnCreate = FormCreate
  PixelsPerInch = 96
  TextHeight = 16
  object ToolBar1: TToolBar
    Left = 0
    Top = 0
    Width = 289
    Height = 29
    Caption = 'ToolBar1'
    TabOrder = 0
    object Pause: TSpeedButton
      Left = 0
      Top = 2
      Width = 48
      Height = 22
      AllowAllUp = True
      GroupIndex = 1
      Caption = '暂停'
      OnClick = PauseClick
    end
    object Stop: TSpeedButton
      Left = 48
```

```
    Top = 2
    Width = 48
    Height = 22
    GroupIndex = 2
    Caption = '停止'
    OnClick = StopClick
  end
  object Next: TSpeedButton
    Left = 96
    Top = 2
    Width = 48
    Height = 22
    GroupIndex = 2
    Caption = '快进'
    OnClick = NextClick
  end
  object Play: TSpeedButton
    Left = 144
    Top = 2
    Width = 48
    Height = 22
    GroupIndex = 2
    Caption = '播放'
    OnClick = PlayClick
  end
  object Prev: TSpeedButton
    Left = 192
    Top = 2
    Width = 48
    Height = 22
    GroupIndex = 2
    Caption = '快退'
    OnClick = PrevClick
  end
  object Record: TSpeedButton
    Left = 240
    Top = 2
    Width = 48
    Height = 22
```

```
      GroupIndex = 2
      Caption = '录音'
      OnClick = RecordClick
    end
  end
  object StatusBar1: TStatusBar
    Left = 0
    Top = 30
    Width = 289
    Height = 20
    Panels = <
      item
        Text = '工作状态：'
        Width = 68
      end
      item
        Width = 50
      end>
    SimplePanel = False
  end
end
```

这个文件所定义的 Form 如下图 3.3 所示：



图 3.3Player 界面

预定义头文件“Unit1.h”

```
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Buttons.hpp>
```

```
#include <ComCtrls.hpp>
#include <ToolWin.hpp>
//---------------------------------------------------------------------
enum RecState{_STOP = 1,_PLAY,_PREV,_NEXT,_REC};
//定义列举类型的数据供属性 WorkState 使用，列举从“1”开始是为了避开程序
//初始化对属性的影响/，因为程序在给变量初始化时赋值为“0”。
//---------------------------------------------------------------------
class TForm1 : public TForm
{
__published:    // IDE-managed Components
        TToolBar *ToolBar1;
        TSpeedButton *Pause;
        TSpeedButton *Stop;
        TSpeedButton *Next;
        TSpeedButton *Play;
        TSpeedButton *Prev;
        TSpeedButton *Record;
        TStatusBar *StatusBar1;
        void __fastcall StopClick(TObject *Sender);
        void __fastcall FormCreate(TObject *Sender);
        void __fastcall NextClick(TObject *Sender);
        void __fastcall PlayClick(TObject *Sender);
        void __fastcall PrevClick(TObject *Sender);
        void __fastcall RecordClick(TObject *Sender);
        void __fastcall PauseClick(TObject *Sender);
private:    // User declarations
        RecState  FState;
        bool    FIsPause;
        bool    FIsPlay;
        bool    FIsPrev;
        bool    FIsNext;
        bool    FIsRec;
        //声明各属性的存储变量；
protected:     // Protected 域是手工后增加的
        void __fastcall FSetState(RecState Value);
        void __fastcall FSetPause(bool Value);
        void __fastcall FSetIsPlay(bool Value);
        void __fastcall FSetIsPrev(bool Value);
        void __fastcall FSetIsNext(bool Value);
```

```
        void __fastcall FSetIsRec(bool Value);
        //声明各属性的设置函数；
public:     // User declarations
        __property RecState WorkState = {read = FState,write = FSetState};
        __property bool IsPause = {read = FIsPause,write = FSetPause};
        __property bool IsPlay =  {read = FIsPlay,write = FSetIsPlay};
        __property bool IsPrev =  {read = FIsPrev,write = FSetIsPrev};
        __property bool IsNext =  {read = FIsNext,write = FSetIsNext};
        __property bool IsRec =  {read = FIsRec,write = FSetIsRec};
        //声明属性；
        __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------
#endif
```

附加 CPP 单元“ PropertyFun.cpp”

```
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"

void __fastcall TForm1::FSetState(RecState Value)
{   //WorkState 的设置函数
    if(Value == FState)
       return;      //这是一个安全措施，使程序引用属性时不需要考虑目前该属性
                    //处于什么状态；
    switch ( Value )
    {
      case _STOP:   //表示设置到未工作状态
        //Pause->Enabled = true;   Pause 的状态和运行状态无关
        Stop->Enabled = false;
        Next->Enabled = true;
        Play->Enabled = true;
        Prev->Enabled = true;
        Record->Enabled = true;
        //设置按钮的有效性
        Next->Down = false;
```

```
    Play->Down = false;
    Prev->Down = false;
    Record->Down = false;
    //设置按钮是否被按下
    IsPlay = false;
    IsRec = false;
    IsPrev = false;
    IsNext = false;
    //设置底层次的属性，使其与工作状态 WorkState 相吻合；
    StatusBar1->Panels->Items[1]->Text = "";  //用于测试的状态栏
  break;
  case _PLAY:    //表示设置到播放状态
    //Pause->Enabled = true;
    Stop->Enabled = true;
    Next->Enabled = true;
    Play->Enabled = true;
    Prev->Enabled = true;
    Record->Enabled = false;
    IsRec = false;
    IsPrev = false;
    IsNext = false;
    IsPlay = true;
  break;
  case _PREV:    //表示设置到快退状态
    //Pause->Enabled = true;
    Stop->Enabled = true;
    Next->Enabled = true;
    Play->Enabled = true;
    Prev->Enabled = true;
    Record->Enabled = true;
    IsRec = false;
    IsNext = false;
    IsPlay = false;
    IsPrev = true;
  break;
  case _NEXT:    //表示设置到快进状态
    //Pause->Enabled = true;
    Stop->Enabled = true;
    Next->Enabled = true;
```

```
        Play->Enabled = true;
        Prev->Enabled = true;
        Record->Enabled = true;
        IsRec = false;
        IsPrev = false;
        IsPlay = false;
        IsNext= true;
      break;
      case _REC:    //表示设置到录音状态
        //Pause->Enabled = true;
        Stop->Enabled = true;
        Next->Enabled = false;
        Play->Enabled = false;
        Prev->Enabled = false;
        Record->Enabled = true;
        IsPlay = false;
        IsPrev = false;
        IsNext = false;
        IsRec = true;
      break;
    }
    FState = Value;     //设置存储变量的内容
}
//---------------------------------------------------------------------
void __fastcall TForm1::FSetPause(bool Value)
{   //设置运行到暂停的状态
    if(FIsPause == Value)
       return;
    switch (WorkState)     //暂停和当前的工作状态有关
    {                      //快进、快退对暂停操作无影响
      case _PLAY:          //当处于播放状态
        if(Value)
        {
          //此处添加暂停播放的代码；
          StatusBar1->Panels->Items[1]->Text = "播放暂停";
        }
        else
        {
          //此处添加暂停恢复播放的代码；
```

```
            StatusBar1->Panels->Items[1]->Text = "正在播放";
          }
        break;
        case _REC:       //当处于录音状态
          if(Value)
          {
            //此处添加暂停录音的代码；
            StatusBar1->Panels->Items[1]->Text = "录音暂停";
          }
          else
          {
            //此处添加暂停恢复录音的代码；
            StatusBar1->Panels->Items[1]->Text = "正在录音";
          }
        break;
      }
      FIsPause = Value;
      Pause->Down = FIsPause;     //该行代码可以省略
}
//---------------------------------------------------------------------
void __fastcall TForm1::FSetIsPlay(bool Value)
{   //设置程序为播放状态，播放状态受到暂停属性的约束
    if(FIsPlay == Value)
      return;
    if(IsPause)
    {
      //添加播放并被暂停代码；
      StatusBar1->Panels->Items[1]->Text = "暂停播放";
    }
    else
    {
      //添加播放代码；
      StatusBar1->Panels->Items[1]->Text = "正在播放";
    }
    FIsPlay = Value;
    Play->Down = FIsPlay;    //该行代码可以省略
}
//---------------------------------------------------------------------
void __fastcall TForm1::FSetIsPrev(bool Value)
```

```
{   //设置快退状态
    if(FIsPrev == Value)
      return;
    //添加快退代码；
    FIsPrev = Value;
    Prev->Down = FIsPrev;      //该行代码可以省略
    StatusBar1->Panels->Items[1]->Text = "快速退回";
}
//---------------------------------------------------------------------
void __fastcall TForm1::FSetIsNext(bool Value)
{   //快进的设置函数
    if(FIsNext == Value)
      return;
    //添加快进代码；
    FIsNext = Value;
    Next->Down = FIsNext;      //该行代码可以省略
    StatusBar1->Panels->Items[1]->Text = "快速前进";
}
//---------------------------------------------------------------------
void __fastcall TForm1::FSetIsRec(bool Value)
{
    if(FIsRec == Value)
      return;
    if(IsPause)
    {
       //添加录音并被暂停代码；
       StatusBar1->Panels->Items[1]->Text = "暂停录音";
    }
    else
    {
       //添加播放代码；
       StatusBar1->Panels->Items[1]->Text = "正在录音";
    }
    FIsRec = Value;
    Record->Down = FIsRec;     //该行代码可以省略
}
```

在这个文件中，多处对按钮的 Down 属性的设置是可以省略的，但笔者建议加上更好一些，该例中去掉这些代码对程序没有任何影响，因为引用这些属性的只有按钮的动作，

而按钮的动作已经可以正确的设置按钮的 Down 属性（关于 SpeedButton 的属性和用法详见联机帮助内容），但是作为一个程序应该尽量的保证状态对应的完整性。IsPause 和 Pause->Down 是严格对应的，但是若有其他动作，比如由一个菜单项引发动作来设置 IsPause 为 True，那么 Pause->Down = FIsPause 就显得非常重要。

单元文件“Unit1.cpp”

```
//---------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------
void __fastcall TForm1::StopClick(TObject *Sender)
{
    WorkState = _STOP;
}
//---------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    WorkState = _STOP;
}
//---------------------------------------------------------------------
void __fastcall TForm1::NextClick(TObject *Sender)
{
    WorkState = _NEXT;
}
//---------------------------------------------------------------------
void __fastcall TForm1::PlayClick(TObject *Sender)
{
    WorkState = _PLAY;
}
```

```
//---------------------------------------------------------------------
void __fastcall TForm1::PrevClick(TObject *Sender)
{
    WorkState = _PREV;
}
//---------------------------------------------------------------------
void __fastcall TForm1::RecordClick(TObject *Sender)
{
    WorkState = _REC;
}
//---------------------------------------------------------------------
void __fastcall TForm1::PauseClick(TObject *Sender)
{
    IsPause = !IsPause;
}
```

从这里也可以看出，这个单元只是 Form 的各种事件对属性的简单引用。

这是一个完整的例子，只是缺少了其中部分关于具体操作代码，而这部分代码和本书要将书的内容关系不大，有兴趣的读者可以自行完善。从这个例子中可以看出采用了针对程序运行状态的分析之后，使程序内部的逻辑关系非常清晰，将一些复杂的逻辑关系用若干简单逻辑进行组合而表达了，而这种组合关系正是经过抽象后的状态。

2.4.3 提高程序的可维护性

对于程序的维护一直是软件开发的一个重要内容，任何程序设计出来都不能保证没有丝毫的缺限，而且维护本身还包含解决程序分析时遗留的缺限。由于面向状态的技术本身是对事物更加真实的抽象模型，更加符合客观事物的规律，而且最大限度的消除了程序之间的相关性，并且力图使用最简单的逻辑关系完成复杂的功能，因此在程序的维护上将更加容易、可行。

仍然就上面 Player 的例子而言，假如暂停按钮不光是对播放和录音其作用，而且还对快进和快退起作用，那么只需要将 IsPause、IsPrev 和 IsNext 三个属性的设置函数稍加修改就可以，其他代码完全不用动，因为也只有这三个状态和原先有差异。修改后的函数如下：

```
void __fastcall TForm1::FSetPause(bool Value)
{
    if(FIsPause == Value)
       return;
    switch (WorkState)
    {
```

```
case _PLAY:
   if(Value)
   {
      //此处添加暂停播放的代码；
      StatusBar1->Panels->Items[1]->Text = "播放暂停";
   }
   else
   {
      //此处添加暂停恢复播放的代码；
      StatusBar1->Panels->Items[1]->Text = "正在播放";
   }
break;
case _REC:
   if(Value)
   {
      //此处添加暂停录音的代码；
      StatusBar1->Panels->Items[1]->Text = "录音暂停";
   }
   else
   {
      //此处添加暂停恢复录音的代码；
      StatusBar1->Panels->Items[1]->Text = "正在录音";
   }
break;
case _PREV:   //此处至 Switch 结尾是增加的内容
   if(Value)
   {
      //……
      StatusBar1->Panels->Items[1]->Text = "快退暂停";
   }
   else
   {
      //……
      StatusBar1->Panels->Items[1]->Text = "快速回退";
   }
break;
case _NEXT:
   if(Value)
   {
```

```
            //……
            StatusBar1->Panels->Items[1]->Text = "快进暂停";
         }
         else
         {
            //……
            StatusBar1->Panels->Items[1]->Text = "快速前进";
         }
      break;
    }
    FIsPause = Value;
    Pause->Down = FIsPause;
}
//---------------------------------------------------------------------
void __fastcall TForm1::FSetIsPrev(bool Value)
{
    if(FIsPrev == Value)
      return;    //以下if是修改的内容
    if(IsPause)
    {
       //……
       StatusBar1->Panels->Items[1]->Text = "回退暂停";
    }
    else
    {
       //……
       StatusBar1->Panels->Items[1]->Text = "快速回退";
    }
    FIsPrev = Value;
    Prev->Down = FIsPrev;
}
//---------------------------------------------------------------------
void __fastcall TForm1::FSetIsNext(bool Value)
{
    if(FIsNext == Value)
      return;    //以下if是修改的内容
    if(IsPause)
    {
       //……
```

```
        StatusBar1->Panels->Items[1]->Text = "快进暂停";
    }
    else
    {
        //……
        StatusBar1->Panels->Items[1]->Text = "快速前进";
    }
    FIsNext = Value;
    Next->Down = FIsNext;
}
```

读者可以验证，在不考虑具体执行代码的情况下（以省略号代替的部分），将前面的 Player 例子修改成上面这三个函数绝对不会超过 5 分钟，而且不会出现错误。可维护性高正是面向状态的一个显著特征。

2.4.4 提高程序的可扩展性

和可维护性一样，程序的可扩展性也是衡量一个软件优略的重要指标，有其对商业软件而言往往是不断地提高、不断地完善功能。结构不合理的软件，很可能在扩充功能时不得不将以前的代码推翻而重新编写，但这种情况在采用了面向状态的分析方法之后，出现的几率将要小得多，除非扩展的内容与原来大相径庭。

前面 Player 的例子是一个用于声音文件的播放程序，假如现在要扩展为播放多种格式的文件，比如视频文件、Midi 文件或其他的流式多媒体文件。那么如何对程序进行修改，我们从下面两点考虑：

- 几个按钮对不同的多媒体文件表现形式是一样的，最多是某些类型不存在“录制”功能，比如对于 Flash 动画文件，但也可以将 Record 定义为新建一个 Flash 对象并调用相应的编辑软件或者其他什么的，在这里我们不作过多考虑，认为所有按钮不同类型的文件的表现行为都一样。
- 播放不同类型的文件只是具体的核心操作有所不同。

根据这两个条件我们看如何来修改程序。由于篇幅的关系，这里并不给出完整的代码，仅就 Play 功能播放 Avi 文件给出一个修改思路：

- 要增加一个关于文件类型的属性 MediaType；
- 定义一个新的列举数据类型，或使用宏定义，提供给 MediaType 不同的状态；

如 `enum MDType{_WAV = 1,_AVI , /*…其他类型*/};`

- 定义一个关于播放处理的函数句柄类型：TMEvents；
- 增加一个关于播放的“事件”；

如：`__property TMEvents OnMDPlay={read = …};`

- 修改 FSetIsPlay 函数，并使用触发 OnMDPlay 事件来完成具体的播放操作；
- 在 MedisType 的设置函数 FSetMediaType 中写入根据不同文件类型改变

OnMDPlay 句柄的代码；

在这个修改思路里面我们使用了面向状态的另一个非常重要的概念——事件，严格的讲，事件也是一种属性（Property）只是这种属性其类型必须是函数的句柄，而且通常是无返回值的函数句柄。需要对事件（Events）详细了解，请参阅后面有关章节。

采用了这种结构要对程序进行扩充是非常容易的事情——需要增加一种文件类型的支持，只需要在 MediaType 的定义中增加该类型的标识码，然后编写一个 TMEvents 类型的处理函数，最后在 FSetMediaType 函数中增加挂入该类型处理函数句柄的就行了。采用类似的技术编写现在比较流行的“插件”也是非常容易的事。

总之面向状态的技术给我们提供了很多解决问题的新方法和新思路，只要使用得当，就会获得事半功倍的效果。

3 面向状态的基本思想

前面对面向状态的基本概念和优点作了一些介绍，读者应该有一个初步的了解，那么到底什么是面向状态的设计思想呢？这需要结合编译器及编程语言本身的特点来进一步了解。

C++ Builder 中的可视化组件（VCL）是基于属性、方法、事件（PEM）的模式，PEM 模式定义了数据成员（Property），对数据成员的操作函数（Methods）以及类的操作数之间的作用（Events）。VCL 为分层结构的类库，是用 Object Pascal 编写并将之用于 C++ Builder 的 IDE 中，通过对 VCL 可视化类库的使用，可以使你非常方便运用 C++ Builder 的组件面板和对象查看器建立你的应用程序，而不需要手工编写很多的代码。这正是 VCL 的最初目的，但 VCL 的作用远不止于此，PEM 模式不仅可以应用到可视化组件之中，也可以使用到任意的代码单元中，只要这个单元运用了 VCL 类库，我们前面所举的两个例子就是这样的。

所谓面向状态的思想实际上一直贯穿于 VCL 类库中，我们将通过了解 VCL 的运行机制来了解面向状态程序设计思想。

3.1 基于事件驱动的程序设计

VCL 组件是建立在 EDP 技术基础上的，EDP（Event driven programming）技术是指程序基于事件驱动的，在这种方式中，程序员并不需要知道用户下一个操作会是什么，而所有的过程都是由事件触发的，比如在 Windows 中消息队列及消息处理机制，便是一种由事件驱动的程序处理机制。但是在 VCL 中对这种事件构造要比 Windows 复杂的多，也更有效地多，通常 VCL 组件的事件都是通过抽象而形成的，能够独立并完整的描述一个状态的改变情况，如 PageControl 的 Onchanged 事件，可以由用户鼠标点击 Pages 标签而触发，也可以是由于用户按 Ctrl + Tab 键而触发。

在进一步了解 VCL 的事件至前有必要对不同程序设计的运行机制做更深入的了解，只

有深刻的理解了 VCL 的处理机制才可能更加得心应手的使用之。

3.1.1 单任务操作系统的处理机制

单任务操作系统是 PC 中出现最早的操作系统，典型的代表示 MS-DOS，当时的程序设计都有一个显著的特点，就是程序主要使用顺序的过程驱动方式。因为当时的程序基本上是基于字符界面的，而且大多数应用都会是独占方式运行，即每一时刻时能运行一个程序，即使在 Linux 或者 Unix 等多任务操作系统中，也是由系统为每一个程序创建一个“虚拟机”，程序的切换只是在不同的“虚拟机”之间切换。

正因为程序运行是独占的，机器的所有资源都是该程序独自使用，因此大多是程序都是按照顺序过程驱动的，比如一个程序包含了四个功能模块 A、B、C、D，其中 C 模块为判断并和 B 模块组成循环，那么程序的执行顺序是可以预测的，流程如图 3.4 所示：

程序的执行顺序总共有两个：要么沿 A→B→C→D 执行要么沿 A→B→C→B（循环过程）→D 执行。

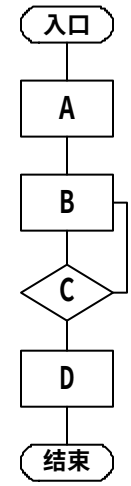


图 3.4Dos 程序的执行流程

尽管在后也出现了许多优秀的图形界面，但整体来说程序的处理机制仍然和按照顺序结构执行的，界面越是复杂，处理起来难度也就越大。最令人头疼的是这种结构在现实中是很难找到的，也就是说它并不完全符合客观规律，因此程序设计的难度也相对较大。

3.1.2 多任务操作系统的处理机制

正是由于这个原因，在多任务操作系统尤其是图形界面的操作系统中，普遍的引入的消息处理机制。图形界面允许用户同时运行多个应用程序，而计算机的资源包括鼠标、键盘等都是多各程序共同使用的，因此操作系统有必要统一管理、协调各个程序占用的资源。

在 XWindows 中通常是操作系统向应用程序发送 Signal（信号），而在 Windows 中则是通过 Message（消息）处理机制来完成的，由于 Signal 和 Message 的产生是随机的，用户的鼠标可能点击这个按钮也可能点击那个按钮或者菜单，是有一定的不可预知性的，程序也是根据收到的不同消息或信号来调用相应的模块。这使程序功能得到大大的增强，也更符合客观规律。

很可能出现的程序流程如图 3.5：

在 Windows 中，对于消息的处理过程是这样的：

- 系统由一个消息管理器，负责接收所有的消息，并将消息分类，分发给相应的应用程序或者线程。
- 每个应用程序都包含一个

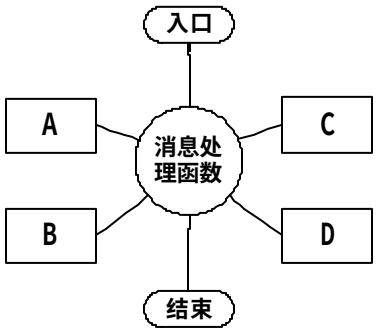


图 3.5 Windows 系统的处理流程

TranslateMessage 和 DispatchMessage 函数，TranslateMessage 负责将消息中的 virtual-key 翻译成字符消息，DispatchMessage 负责分派消息，并将其放入消息队列。

- 消息队列使系统分配给应用程序或者线程的资源，队列中消息并不立即被执行，而是由系统根据当前的资源情况，在适当的时候调用相应的处理函数。也就是说应用程序将处理权是交给操作系统的，而不是自己决定什么时候处理，什么时候不处理的。
- 系统对应用程序功能的调用是通过所谓的回调（CallBack）函数来完成的，每个应用程序或者线程至少包含一个以上的回调函数，一个典型的应用程序的回调函数可能如下形式：

```
LRESULT FAR PASCAL _export WndProc(HWND hWnd, UINT message,
                WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
      case WM_QUIT:
        //……
      case WM_DESTROY:  // message: window being destroyed
        PostQuitMessage(0);
      break;
      case /*其它消息*/
        //处理过程
      break;
    ... ...
      default:           // 调用Windows缺省的处理函数；
        return (DefWindowProc(hWnd, message, wParam, lParam));
    }
    return 0;
}
```

这正是图 3.5 中消息处理函数的那个圆的内容，但这个函数实际上是 Windows 系统来调用的，而不是程序自己来调用。对于 Windows 的消息处理机制可以用下面的图 3.6 来表示：

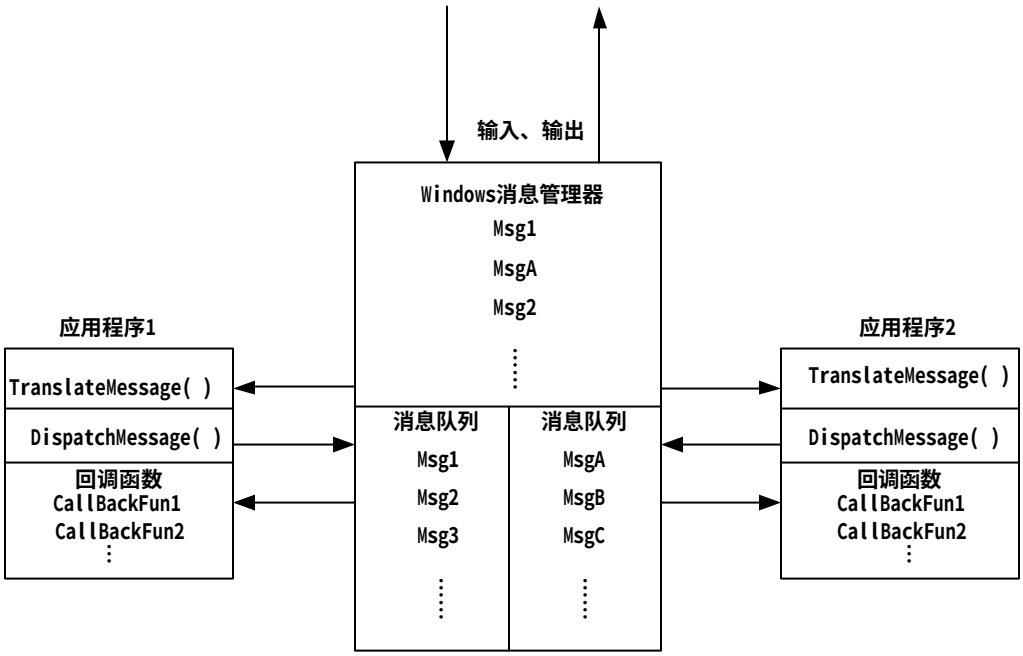


图3.6 Windows的消息处理机制

这是更贴近操作系统的消息处理机制，无论是采用什么语言或者技术来编写 Windwos 程序，都必须遵循这个规则，但现在编程的时候，我们很少再去编写程序的回调函数，而是通过 Windows FrameWorks（框架）来完成的，使程序员可以从更高一层次驾驭消息。

3.1.3 其他 Windows FrameWorks 处理机制

FrameWorks 实际上就是一组 C++类库的组合，包含了基本界面元素和相应的消息处理机制。在 VCL 框架出现以前，Windows FrameWorks 分为两大阵容，一是 Borland 公司推出的 OWL（Object Windows Library）和微软公司的 MFC（Microsoft Foundation Classes）。在 OWL 和 MFC 的最底层仍然使用类似 Switch…Case 结构来处理系统的消息，但在写代码时可以采用被框架封装以后的更为简单的处理机制。

在 MFC 中消息处理最基本的也是使用最多的是：BEGIN_MESSAGE_MAP（消息映射表）宏。对该宏的定义如下：

```
#define BEGIN_MESSAGE_MAP(theClass, baseClass) \
    const AFX_MSGMAP* theClass::GetMessageMap( ) const \
    { return &theClass::messageMap; } \
    AFX_COMDAT AFX_DATADEF const AFX_MSGMAP theClass::messageMap = \
    { &baseClass::messageMap, &theClass::_messageEntries[0] }; \
    AFX_COMDAT const AFX_MSGMAP_ENTRY theClass::_messageEntries[] = \
    { \
```

消息映射表的结尾使用 END_MESSAGE_MAP 宏，定义如下：

```
#define END_MESSAGE_MAP( ) \
    {0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
    }; \
```

由这两个宏的定义可以看出，实际上在每个 MFC 类中都包含一个_messageEntries 数组，用以存放消息处理函数的代码入口，当消息表接到一个消息定义时，首先检查父类的消息映射表是否包含对该消息处理函数的定义，如果有则给该消息分配与派生类中同名的处理函数，如果没有则根据消息的定义宏取分配派生类中相应的处理函数。

一个程序的 BEGIN_MESSAGE_MAP 定义可能如下：

```
BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
    //{{AFX_MSG_MAP(CMainFrame)
    ON_WM_CREATE( )
    //}}AFX_MSG_MAP
END_MESSAGE_MAP( )
```

这一段代码告诉编译器在 CMainFrame 类中，WM_CREATE 系统事件将被注册使用，并且在 Cpp 单元文件中将有一个 OnCreate(LPCREATESTRUCT lpCreateStruct)函数在 CMainFrame 的对象在接收到 WM_CREATE 事件后被调用。

WM_CREATE 是一个系统事件发出的消息，即该消息是程序运行时是系统自己产生的，在 CMDIFrameWnd 类中已经包含了对该消息的处理方法 OnCreate 函数，CMainFrame 是 CMDIFrameWnd 的派生类，因此也继承了相应的消息映射表，缺省条件下 WM_CREATE 消息将会调用父类的 OnCreat 函数，上面的宏将会使父类的 OnCreat 函数被派生类的 OnCreat 函数所代替。因此在 CMainFrame 类的 OnCreat 函数中至少要有一次对父类 OnCreat 函数的调用。

Windows 中更多的消息是属于用户事件所发出的消息，即消息是由用户的操作而引发的，象键盘、鼠标等操作引发的消息等都属于用户事件。用户事件往往在父类中没有消息映射，对于消息映射表中消息的定义必须给出完整的参数，如下面的代码就是为 CWinApp 的派生类 CBatchApp 定义一个响应 ON_COMMAND 事件中 About 菜单 ID_APP_ABOUT 消息的处理函数 OnAppAbout。

```
BEGIN_MESSAGE_MAP(CBatchApp, CWinApp)
    //{{AFX_MSG_MAP(CBatchApp)
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP( )
```

在 Borland C++中的 OWL 框架中，通常使用事件响应表（Response Table），总共有 4 个响应表的宏定义，分别为：

```
DEFINE_RESPONSE_TABLE
DEFINE_RESPONSE_TABLE1
DEFINE_RESPONSE_TABLE2
DEFINE_RESPONSE_TABLE3
```

这几个宏的区别是所带的参数不同，后面的数字代表该类由几个父类派生而来的，它们功能基本是相同的。DEFINE_RESPONSE_TABLE3 的定义如下：

```
#define DEFINE_RESPONSE_TABLE3(cls, base1, base2, base3)\
  bool  cls::Find(TEventInfo& eventInfo, TEqualOperator equal)\
   {eventInfo.Object = (GENERIC*)this;\
     return SearchEntries((TGenericTableEntry __RTFAR*)__entries,
                                              eventInfo, equal) ||\
         base1::Find(eventInfo, equal) ||\
         base2::Find(eventInfo, equal) ||\
         base3::Find(eventInfo, equal);}\
  DEFINE_RESPONSE_TABLE_ENTRIES(cls)
```

结尾的 END_RESPONSE_TABLE 宏定义如下：

```
#define END_RESPONSE_TABLE\
  {0, 0, 0, 0}}
```

其中 DEFINE_RESPONSE_TABLE_ENTRIES 也是一个宏，定义如下：

```
#define DEFINE_RESPONSE_TABLE_ENTRIES(cls)\
  TResponseTableEntry< cls > __RTFAR  cls::__entries[ ] = {
```

比较一下不难发现，MFC 和 OWL 采用的是非常接近的处理方法，尽管内部处理机制仍有很大区别，而且性能和使用方法也不一样，但是这两种 FrameWorks 都属于同一类型的。由于篇幅的关系，我们在这里不对 OWL 和 MFC 的消息处理作更多的介绍，需要进一步了解的读者参阅相关书籍。

3.1.4 VCL 类库的处理机制

Borland 的 VCL 的消息机制相对于前两种 FrameWorks 有了很大的提高，在 VCL

中大多数情况下用户不必和 Windows 消息直接打交道，而是和事件（Events）打交道。

Events 不能简单的理解为 Windows 消息的映射，它不仅可以是 Windows 消息触发的，而且还可以是抽象的事件，比如控件的 OnChange、OnClick 事件都不是消息的直接映射，实际上抽象的事件在应用中更具有使用价值。关于 VCL 中的事件我们在后面详细介绍，这里仅就 VCL 的消息处理机制和 Windows API 及 OWL／MFC 作一比较，API 的消息出来机制如图 3.7 所示：

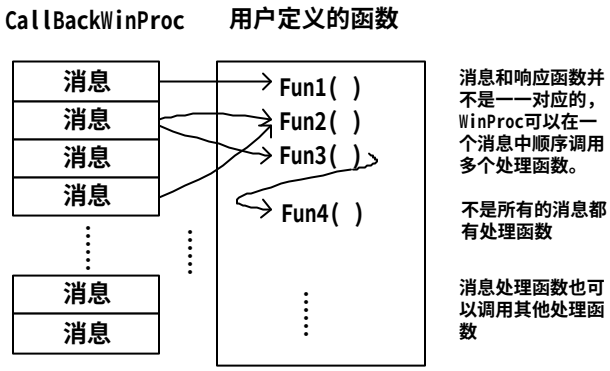


图 3.7 使用 Windows API 的消息处理机制

图中实线箭头表示直接对应关系，从上图以及回调函数的例子中可以看出，在 Windows API 处理消息的时侯，其控制权是由回调函数来掌握的，用户定义的处理函数都受到它的指派，而且在 CallBackWinProc 中本身就可以处理某些 Windows 消息，很难作到针对对象进行消息分派。

在 MFC 和 OWL 中不再直接使用回调函数，消息映射或者响应表将 Windows 消息和相应的处理函数对应起来，这种对应关系是根据类来作的，可以区分消息的来源和使用者。但消息映射表是设计时建立的，运行时很难动态的修改消息的相应函数。在 MFC 和 OWL 中也有部分“消息”属于经过处理后的抽象事件，但由于本身这种消息派发机制的限制，功能比 VCL 中要弱得多，没有从根本上得到提升，没有本质上的飞跃，如下图 3.8 所示。

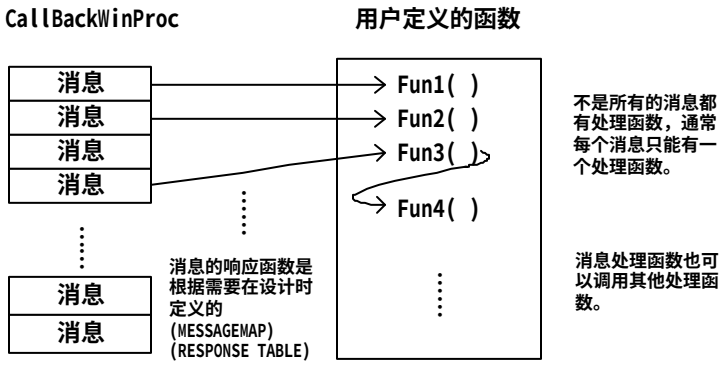


图 3.8 OWL 和 MFC 的消息处理机制

下图 3.9 是关于 VCL 类库中消息处理的示意，图中的虚线箭头表示可以重新定向的函数和事件的对应关系，双点划线表示消息是经过深度加工而触发事件。

事件在 VCL 中是预先定义好的处理函数的句柄，对于需要处理的事件，程序只需要将句柄指向相应的处理函数就可以了。VCL 中的事件大多数都是经过处理的，过滤掉了 Windows 消息中的多余信息，并提供所必须的某些信息，最终提供给处理函数有针对性

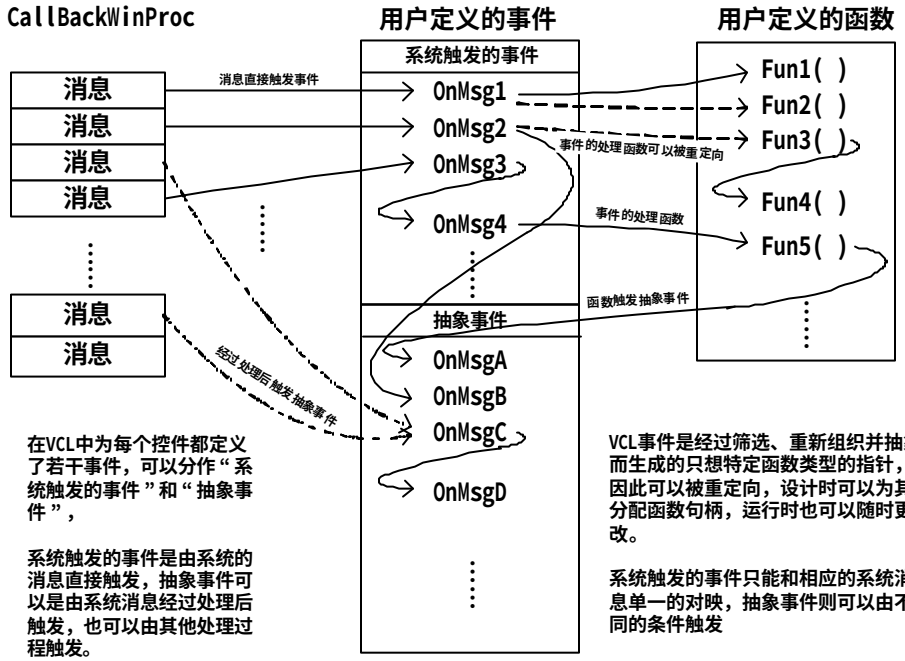


图 3.9 VCL 类库的消息处理机制

地、特定的数据包，比如 OnClick 事件通常是由鼠标左键弹起而触发，但在 TWinControl 类派生的控件中，也可以是由控件获得焦点后按空格键或回车键而触发，同时 OnClick 事件不需要任何关于鼠标和键盘的信息，而更希望得到一个事件发起者的信息，因此 OnClick 事件通常只需要一个参数 TObject* Sender 来告知处理函数该事件是属于谁发起的。

在 VCL 类库中保留了对 OWL 和 MFC 的支持，因此在程序中也可以使用消息映射表和事件响应表。此外 C++ Builder 的 VCL 类库本身中间层的消息处理也使用了 BEGIN_MESSAGE_MAP 宏，宏定义的名称和使用方法同 MFC 的宏完全相同，但它们的内部处理机制是完全不同的，在编写包含 MFC 类库的程序时，应该避免使用 VCL 类库的消息映射表，取而代之的是直接重载 Dispatch 函数（VCL 的 BEGIN_MESSAGE_MAP 宏实际上是重载 Dispatch 函数）或者重定向 TControl 派生类的 WindowProc 属性。

观察 VCL 关于 BEGIN_MESSAGE_MAP 宏的定义：

```
#define BEGIN_MESSAGE_MAP virtual void __fastcall Dispatch(void *Message)\
   {                                                \
     switch (((PMessage)Message)->Msg)              \
     {
```

宏 VCL_MESSAGE_HANDLER 的定义：

```
#define VCL_MESSAGE_HANDLER(msg,type,meth)       \
    case    msg:              \
      meth(*((type *)Message));     \
    break;
```

结束宏 END_MESSAGE_MAP 的定义：

```
#define END_MESSAGE_MAP(base)          default:    \
       base::Dispatch(Message);    \
       break;                 \
     }                       \
}
```

就不难理解 VCL 最底层的消息处理机制了——消息映射表预编译后的代码正是 Windows 回调函数的那种形式：

```
BEGIN_MESSAGE_MAP
  VCL_MESSAGE_HANDLER(WM_SIZE, TWMSize, WMSize);
  VCL_MESSAGE_HANDLER(WM_SETFOCUS, TWMSetFocus, WMSetFocus);
END_MESSAGE_MAP(TCustomControl);
```

预编译后将会被翻译为：

```
virtual void __fastcall Dispatch(void *Message)
{
    switch  ( ( ( PMessage )Message )->Msg )
    {
       case WM_SIZE:
          WMSize(*((TWMSize*)Message));
       break;
       case WM_SETFOCUS:
          WMSetFocus (*((TWMSetFocus *)Message));
       break;
       default:
          TCustomControl::Dispatch(Message);
       break;
```

```
    }
}
```

基于 VCL 类库的程序，各个功能模块基本上都是通过事件驱动并调用适当的函数而完成的，事件并不等同于 Windows 中的消息。在面向状态的程序设计中，事件也是一个非常重要的概念，要对事件更深入的了解，需要结合 VCL 中另一个重要的概念——属性，以及属性和状态、事件和状态之间的关系，我们在后面的内容对 VCL 的事件做更详细的介绍。

关于对 Windows 消息的响应，VCL 支持使用 API 的方式，大多数的 VCL 事件是经过过滤、加工而成的，很大一部分 Windows 消息没有事件和之对应，但这些事件有时是必须进行处理的，这时可以使用消息映射表或者使用 WindowProc 属性重定向某个对象的消息处理函数。另外 C++ Builder 中 TApplication 提供了更加贴近底层的处理接口，OnMeesage 事件，这个事件是整个程序中所有消息的必经之路，通过它可以截获该程序的所有消息，并可以决定是否允许后面的代码继续处理某个消息。这也已看出 VCL 在 Windows 消息方面有着非常强大的处理能力。

3.2 属性与状态

“属性（property）”是 VCL 中最重要的概念，Proeprty 的英文含义为：财产、所有物、所有权、性质、特性、道具，通常我们都将其翻译为属性，但从 VCL 中 Property 所实现的功能上看，称之为“道具”似乎更为合理，而从 Property 和状态的关系上看，称作“属性”却更恰当一些，Property 正确的含义应该是“道具”和“属性”的结合体。为了和习惯上的叫法统一，本书内 VCL 的 Property 均称之为“属性”。关于面向对象的内容，很多书籍中均将对象的成员等称为“对象的属性”，本书第二章中没有采用这样的称呼就是为了避免和 VCL 面向状态中的“属性”相混淆。

属性最初目的是为了使 IDE 在对象查看器（Object Inspector）中能够对对象的某些状态在设计时进行设置，为此 Borland 在 C++ Builder 中对 C++语言进行了扩展，并引入了属性的一些相关处理机制，包括引入事件，在此基础上便产生了面向状态分析和设计方法。我们来看一下属性的一些特点。

3.2.1 属性封装了数据

属性本身并不是一种变量，更不是对象的成员，尽管在使用上属性和一般的成员非常相似，程序在运行的时侯，并不会为属性分配存储单元，对属性的操作通过 C++ Builder 的处理机制最终实际上是对封装在内部的其他数据进行操作的，从这一点意义上讲 Property 更像是一个道具，而不是一个真是的变量。

细心读者可能已经发现在 C++ Builder 中所有 VCL 类库的__publish：定义域和 public：定义域中定义的数据基本上都是成员函数或者属性或者事件，成员一般只出现在 private：或者 protected：定义域中，可以说在 VCL 中所有的数据成员都是受保护的，其对外接口只有属性、方法和事件。

我们来看一下一个典型的属性的定义，这是 HelloWorld 中的部分代码：

```
class TForm1 : public TForm
{
__published:
        //……
private:
       bool FIsNewState;
protected:
       void __fastcall FSetIsNewState (bool Value);
public:      // User declarations
        __property bool IsNewState = {read = FIsNewState,
                                      write = FSetIsNewState};
        //……
};
```

__property 关键字后面的内容是定义了一个 bool 类型的属性 IsNewState，括号中的内容是为该属性指定了该属性的读取和存储的内容，读取内容 read = FisNewState 表示在读取该属性值的时候，所读取的内容是私有成员 FIsNewState 的内容，存储内容 write = FSetIsNewState 表示在为该属性赋值时，实际调用了受保护的设置函数 FSetIsNewState。关于属性的定义是比较多种多样的，__property 关键字的详细使用方法在后面 C++ Builder 的语言扩充中介绍，读者在这里只要知道：属性本身不是一种实实在在的变量，对它的操作只是形式上的。正如前面提到的：

```
IsNewState = !IsNewState;  与
FSetIsNewState (!FIsNewSate) ;
```

在编译后的可执行代码是完全一样的，但这种形式上的改变却带来了革命性的变化—用全新的处理方法和思维方式解决问题。

属性和变量的使用方法基本是相同的，程序设计时完全可以将它看作是一般的变量，只有一个小小的例外，就是不能将属性以引用的方式传递给其它函数，因为属性不是变量，没有系统为其分配的内存单元，因此引用是一个无效的操作。

另一个值的注意的问题是属性的数据类型，从表现形式上看，属性可以为任意类型的数据，包括 C++语言本身的基本数据类型和用户自定义的数据类型，但对于数组类型的属性处理是比较特殊的，因为 C++语言中不允许将数组作为一个函数的返回值或者是参数。这样一来属性所封装的数据中不能包括数组类型，这一点希望读者能有所注意。

3.2.2 属性封装了实现过程

属性不是变量，却封装着特定的数据，这是一个特点，除此之外还有另一个特点是属性封装了特定的处理过程，或者叫实现过程。如何来理解这一特点呢，我们回想一下

FSetIsNewState 函数：

```
void __fastcall TForm1::FSetIsNewState (bool Value)
{
    if( Value == FIsNewState )
      return;
    if( Value )
    {
        Label1->Caption = "This is New State Hello World";
        Color = clWhite;
    }
    else
    {
        Label1->Caption = "Hello World!";
        Color = clBtnFace;
    }
    FIsNewState = Value;
}
```

当出现 IsNewState = !IsNewState;语句时编译器会自动调用 FSetIsNewState（!FIsNewSate）；函数，FSetIsNewState 不仅是对 IsNewState 封装的数据 FIsNewState 进行设置，而且还将 IsNewState 的表现性状一起更新，从这一方面讲将 Property 称之为属性也是合理的。

这只是一个简单的处理过程的封装，复杂的可能还会有对属性约束条件的检查和设置，举个很简单的例子：

在 Table 的属性中包含一个 DataBase 属性和 Active 属性，DataBase 属性可以是一个 TDataBase 组件，TDataBase 组件中也包含一个 Active 属性。它们的约束关系为：

TdataBase->Active 为 TTable->Active 的必要条件。

因此在 Table->Active = **true** 的执行过程中应该包含类似 Table->DataBase->Active = **true** 的处理过程。这样做的目的就是前面我们说到的消除相关性。

属性不仅在定义的存储（write）部分可以封装执行过程，也可以在读取部分（read）封装特定的处理过程，象前面提到的 TWindControl 的 Handle 属性，其定义原型为：

```
__property HWND Handle = {read=GetHandle, nodefault};
```

这里读取部分为一个函数句柄 GetHandle，当程序读取 Handle 属性时，GetHandle 会检查当前存储的数据 FHandle 是否有效，若有效则返回 FHandle，无效则创建窗口并设置和返回 FHandle 为新创建的窗口句柄。

3.2.3 属性是状态的直接体现

从这里已经可以看出，属性是对象状态的直接反映。

属性和状态的关系就像类和对象的关系一样，属性是状态抽象的表现形式，而状态则是属性的具体表现行为。

采用属性来标记对象的状态比采用变量来标记显然要先进得多，状态的标记必须是公开的，声明在 public:定义域，能够被其他对象或者函数访问，而属性的存储数据是声明于 private:定义域，外界只能通过属性或者成员函数来进行访问，这就保证了属性的取值和状态是严格一致的，从而不必担心状态的标记已被修改而状态并没有刷新。因此说属性是状态的直接体现。

3.2.4 属性的分类

属性与成员变量比较，还有一个特殊的地方，就是属性可以根据实际需要定义成不同的类型，这些属性没有本质上的区别，但在使用上可能会稍有不同。大致的属性可以分作读写型属性、只读型属性和抽象属性。

读写型属性

读写型属性使用起来和一般变量一样可以读取属性的取值，也可以为属性赋值，定义属性时，一般读取（read）和存储（write）部分都有内容，通常读取部分直接和存储的数据关联，而存储部分是由一个设置函数来完成的。这类属性在现实中如同仪表盘上带指示灯的按钮一样，本身可以操作，同时也显示了状态信息。

在 VCL 组件中大部分属性都是这种类型，比如 Form 的 Color、Font、Height、Top 等属性都属于读写型属性。

只读型属性

只读属性在定义属性的时候，存储部分没有内容，TControl 的 Handle 属性就是这样的类型，通常读取部分是由读取函数来完成的，也可以是直接和存储的数据关联。这类的属性在使用时只能当 Const 变量一样读取属性的取值，而不能为其赋值。现实中仪表盘上的仪表、指示灯等就属于这种情况，不能对其直接操作，只能用来显示一些状态信息。

组件中只读属性并不是特别多，但却是不可缺少的，它能提供为对象提供非常重要的状态信息，比如 Form 中的 Active、ActiveMDIChild、Canvas、ClientHandle、ClientRect、Floating、Monitor 等属性都属于这种类型。

虚拟属性

所谓虚拟属性，指的是属性表述了一个抽象状态，这种情况在 VCL 组件里面非常的少，虚拟属性并没有存储数据和其对应，而表现的可能是多个状态按照某种逻辑组合抽象而成的状态，这种情况在现实中是存在的，比如仪表中的电功率表，它不是由一个自然的物理量来显示的，而是由电流和电压共同确定的。

虚拟属性可以是只读的，也可以是读写型的，但是读写型的虚拟属性其设置函数的处理要比一般的属性的设置函数复杂得多，通常会有附加条件，比如现实中对功率改变的附加条件可以是定电阻改变功率、定电压改变功率和定电流改变功率。

这种属性在实际编程时用的很少，也应该尽量避免使用，或者通过增加中间属性的方

法转换成更简单的逻关系。

3.2.5 属性的继承性

在 C++ Builder 中，属性还有一个重要的特点，就是具有 C++语言中类的继承性和多态性。__publish:是 C++ Builder 中增加的一个声明的域，声明在__publish:域中的属性和 public:定义域中的属性或者成员有着相同的外部访问权限，而且是被发布的，可以在 IDE 的 Object Inspector 中访问，但继承特性稍微不同。关于属性的继承性和__publish:关键字的使用方法在后面的章节作详细介绍。

属性的多态性体现在不同的类的同一属性可能具有不同的表现形态，比如 Active 在很多组件中都有，但是 Active 的意义和实现方法却相差的很远。

3.3 事件的概念

C++ Builder 中另一个重要的概念就是事件，前面对事件也有所接触，接下来将重点就事件作一些概念性的介绍，事件的定义以及使用方法也在后面针对 C++ Builder 语言语法扩充的内容中详细介绍。

在 C++Builder 中,事件通常都是以 On 前缀开始的,比如 OnClick、OnMouseDown 等等都是事件，从字面上理解似乎是当…的时侯，这是 C++ Builder 中的一个缺省约定，就象属性的设置函数以 FSet 或者 Set 开头后面跟属性的名称一样，编程序的时候也可以完全不按照这种约定来定义事件，但这种可以顾名思义的命名习惯显然更有利于程序的开发和设计。下面我们看一下 C++ Builder 中事件有什么样的特征。

3.3.1 事件是一种特殊的属性

在 C++ Builder 中事件实际上就是一种属性，但属于一种非常特殊的属性，和一般的属性有较大的区别，从其特征和用途上来看，事件是一个更适合的称谓。我们来看一个典型事件 OnClick 的定义：

```
__property Classes::TNotifyEvent OnClick = {read=FOnClick, write=FOnClick,
                stored=IsOnClickStored};
```

从形式上看这和定义一个属性的形式非常类似，但这种形式上的一致并不代表着本质上的一致。对于一个组件在对象查看器中属性通常被显示在 Propertys 标签页中，而事件则被显示在 Events 标签页中，双击对象查看器中的事件，IDE 会自动创建一个和事件匹配的处理函数，并将光标移动到相应的位置，这仅仅是在表现形式上的不同。对于一个事件要求其类型必须是函数的指针类型，上面的 TNotifyEvent 就是一个函数指针，其定义如下：

```
typedef void __fastcall (__closure *TNotifyEvent)( TObject* Sender);
```

这里又涉及到 C++ Builder 扩充的另一个关键字__closure，这是对事件继承性的另一个要求，关于__closure 关键字的详细介绍参见后面章节。

通常情况下事件是可读写的，其读写部分和存储部分可以直接和存储数据关联（上面的 OnClick 就是这样），也可以是由读取函数和设置函数来完成。因此事件在程序运行的时候是允许改变取值的，这正是 C++ Builder 中对消息处理和其它 FrameWorks 最大的区别，这也使得事件了对操作系统的依赖和约束，可以更加独立的描述一件事情或者一个过程，使用也更加灵活。

3.3.2 事件是抽象的瞬间状态

前面我们将状态分为持续状态、非持续状态、离散状态和连续状态，在计算机中连续状态通常也需要转化为非连续状态而使用数字量来描述，而持续状态通常是用属性来描述的，非持续状态正是用我们这里要讲的事件来描述的。

在 C++ Builder 中的事件中有 OnClick、OnChange、OnActive 等事件，都属于“当…发生的时侯”型的事件，还有一些是“当…发生之前”（Befoer…）或“当…发生之后”（After…）等形式的事件，但仔细观察，这些所有的事件都描述的是一种瞬间状态，是一种非持续的状态。

但在 C++ Builder 中几乎所有的事件都是经过一定的抽象而形成的，它是对象的一种内在的因素，对象本身就是抽象的，因而事件也是需要抽象而形成，尽管 VCL 中好多的事件都是 Windows 消息的直接映射，但是反映到 VCL 组件中时也是具有抽象性的，所有的事件都是针对某个对象而言的，并不是整个程序。

读者最需要区分的就是事件和 Windows 的消息，这属于两个完全不同的概念，有着非常大的本质区别，由于事件是 VCL 消息处理机制的一部分，很多 C++ Builder 用户都没有正确区分两者的关系，但实际上 VCL 的事件从其产生的那一刻起，就比 Windows 消息有了质的飞跃。正确认识事件对学习和使用 VCL 类库有着非常大的影响。

3.3.3 事件的分类

事件是 C++ Builder 中同属性一样重要的语言支持，跟属性一样，事件是针对对象而进行表述的，当然可以有系统消息触发，也可以后其它运行状态而触发，抛开对象，仅从触发的条件来看，事件大致可以分作以系统事件、用户事件和抽象事件等三种类别。

系统事件

系统事件指的是这个事件是由系统消息直接触发的，是对系统消息的一个直接映射，在图 3.9 中可以看出，系统事件是和系统消息单一对映的，即一个系统消息只允许触发一个系统事件，但系统事件可以由其他条件触发。

在 C++ Builder 中这类事件是非常多的，比如 OnPaint、OnTimer、OnMaxim、OnResize 等，在进行程序设计的时候，定义这类的事件也比较简单，处理过程相对也简单。

但系统事件并不等同于简单的消息处理，通常对一个系统消息也要进行加工处理，象 OnPaint、OnTimer 这些事件完全不需要消息中的任何参数，因此在这些事件的处理函

数中不需要定义任何与系统消息有关的参数。

用户事件

用户事件是通过用户的操作所触发的，但并不是 Windows 消息的直接映射，通常对一些消息进行一些简单加工而形成的。用户事件有一定的抽象性，但是和 Windows 消息的关系也比较紧密，一般来讲用户事件更能描述一个过程或操作的结果，用户事件不一定对应一条消息，也可能由多个消息而触发，或者是不同的消息都能够触发，典型的例子是 OnClick 事件，而且在不同的组件中 OnClick 触发的条件也是不同的。比如：

OnClick 事件在 Button 中可以是用户点击鼠标左键，也可以是 Button 获得焦点后按空格键或者回车键。

在 Form 中如果包含缺省按钮（Default Property）和取消按钮（Cancel Property），则按动回车和 ESC 键分别可以触发缺省按钮和取消按钮的 OnClick 事件。

而在 Grid、List、ComboBox 等组件中用户选择不同的 Item 都会触发 OnClick 事件。

显然这种针对对象的事件比起针对 Windows 消息的处理方式要有效和方便得多，VCL 中的用户事件也非常多，象 OnKeyPress、OnDblClick、OnMouseDown 等等，这些事件和系统事件一样，往往经过将消息中的参数进行加工，象 OnClick 事件不需要消息的参数，OnKeyPress 事件将键盘扫描码转换成 Virtual Key 代码，并包含了功能键（Shift,、Alt 和 Ctrl）的信息，使 OnKeyPress 事件甚至可以响应 Mouse 的点击消息。

用户事件是 VCL 中使用比较多的事件，也是对 VCL 类库来说必须要掌握的基本事件。

抽象事件

抽象事件应该说也是一种用户事件，是由用户的操作所触发的，但是抽象事件比起前面所说的用户事件要更具有抽象性，和系统消息的相关性更小，一般来说，抽象事件不是直接由系统消息触发的，但是可能是很多系统消息对一个对象状态产生的相同的影响，有时候甚至抽象事件并不是系统消息触发的，而是在其他状态改变时触发的。

在 C++ Builder 的 VCL 中这类事件非常多，比如组件的 OnChange 事件，Form 的 OnActivate、OnClose、OnCloseQuery、OnCreate、OnDeactivate、OnDestroy、OnHelp、OnShow 等事件都属于这一种类型。

抽象事件在 VCL 中的作用是无法替代的，它使事件完全属于对象的范畴，而和系统消息没有多大的关系，它反映的是对象的特征，而不是反映了用户的操作。程序员完全不必知道 OnClose 事件是怎么产生的，也许是系统的任务管理器发出的、也许是用户点击关闭菜单或者是其它操作发出的，但这些操作共同的特征是关闭该 Form，这对于程序员来说，也仅需要关心在接收到关闭指令时需要做什么善后事情。

诸如此类的例子非常的多，抽象事件正是 VCL 事件中最大的突破和革新，熟练掌握和巧妙利用，可以对改善程序结构和性能有非常大的帮助。

3.3.4 事件的继承

同 VCL 中的属性一样，事件也是具有继承性的，而且事件也具有多态性，事件的多态

性表现在不同的类中，事件具体的响应条件可能有很大的差别。

对于事件的继承区别于一般成员的继承，也区别于一般属性的继承，而且在不同的定义域__publish:、public:等之间的继承与标准的 C++语言有很大的区别。这主要是由于事件是一种函数句柄，在标准的 C++语言里，可以将一个父类的指针指向其一个派生类的实例，如：

```
TCostumerForm * Form = new TForm ( ParentHandle);
        // TForm 由 TCostumerForm 派生而来；
```

将是合法的，但是将一个父类中的成员函数的指针变量指向派生类的成员函数这是不允许的，如：

```
class base
{
public:
    void func(int x);
};
class derived: public base
{
public:
    void new_func(int i);
};
void (base::*bptr)(int);        //定义一个指向父类成员函数的指针
bptr = &derived::new_func;      //这是非法的；
```

父类的事件在继承到派生类的时侯，将会出现这种情况，但在 VCL 类库中对事件这样操作是允许的，这需要依赖于前面提到过的关键字：__closure。此外，事件的继承还有一些其它需要注意的地方，我们将在后面内容详细讲解，这里读者只需要了解事件也具有继承性。

3.3.5 属性和事件的关系

从程序的执行上，可以这样认为，事件就是提供了一个处理函数的接口，这个接口会在适当的时候自动的调用，当然调用之前必须检验从处理句柄的有效性。例如事件 OnClose 的触发：

```
begin
    if Assigned(FOnClose) then FOnClose(Self, Action);
end;
```

注：这是 TForm 中 OnClose 触发的一段源码，是由 Object Pascal 编写的 FOnClose 是 OnClose 事件的内部存储数据，翻译成 C++ 代码如下：

```
{
   if( FonClose )
     FonClose(this,Action);
}
```

那么从这本书要讲述的面向状态的设计思想来分析，事件处于什么样的一个地位呢？根据前面对状态、属性以及事件的介绍，再结合 C++ Builder 中的事件的实例，不难理解：事件实际上就是体现了持续状态的变化。所有的事件都是由状态改变触发的。这就是为什么在 C++ Builder 中事件都是以 On 前缀开头的，因为这样更容易理解为：“当…的时侯”。

- 对于系统事件可以理解为系统状态改变的时候触发的，如：

OnTimer 事件就是系统时间改变的过程中当达到我们设定的步进值时触发。

OnKeyDown 可以理解为键盘状态改变的时候触发的。

- 用户事件和系统事件类似，但是更贴近于对象的状态，如：

Button 的 OnClick 事件反映的是 Button 状态处于被按下并弹起来的时候触发的，而在 ListView 中又可以反映鼠标点击或类似点击效果的被选中时触发。

- 而抽象状态则反映的是更加抽象的状态的改变状况，象 Form 的 OnActivate、OnClose、OnCloseQuery 等，读者稍加分析就不难理解。

从数学和物理的角度来分析，对一个事物或者某时刻运动状态的描述，至少应该知道该点的函数值（或运动状态）和该点的一阶导数（或叫做运动趋势），而在 C++ Builder 中属性和事件恰恰反映了这两者的关系，在程序的运行中，大多数状态都是持续的离散状态，即使连续数据在计算机中也只能用离散的数字量来表示。这样一来对计算机程序的描述只要使用属性和事件就已经相当完善了——研究阶梯函数只需要知道函数值和奇点就可以了。数学描述和状态分析的对比示意见图 3.10：

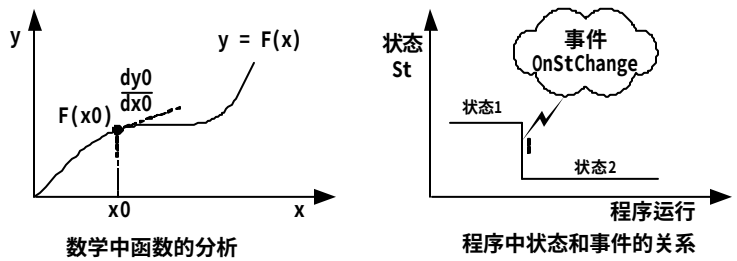


图 3.10 数学分析和面向状态的关系

这个图的表示并不是非常恰当，但可以反映出事件合状态之间的关系，对于 Before…、After…事件可以认为是上面第二个图中 St 曲线的两个拐角处。

在数学中，函数的导数也是一个函数，而在 C++ Builder 中属性是状态的直接体现，

事件是状态变化的反映，而事件也恰恰是一种特殊的属性，这两者之间的吻合并不是偶然现象，而是因为面向状态的方法正是从实际应用中总结出来的。

另外，在 C++ Builder 中完全可以为事件句柄的改变而定义一个触发事件的接口，就是当一个事件的数理函数被换掉时触发另一个事件，比如：

```
class EventsSteps
{
previte:
     TMyEvent FEventsStep1;    //一阶事件存储句柄；
     TMyEvent FEventsStep2;    //二阶事件存储句柄；
protected:
    void __fastcall FSetEVT1(TMyEvent Value);
          //一阶事件的设置函数
public:
    __property TMyEvent OnEventsStep1 = {read = FEventsStep1,write
                                          = FSetEVT1};
    __property TMyEvent OnEventsStep2 = {read = FEventsStep2,
                                         write = FEventsStep2};
}

EventsSteps::FSetEVT1(TMyEvent Value)
{
   if( Value == FEventsStep1 )
      return;
   if(Value ...)      //二阶事件的触发条件；
      if(OnEventsStep2 )
         OnEventsStep2(...);
    FEventsStep1 = Value;
}
```

这种情况在 C++Builder 中是完全允许的，甚至你可以在程序中定义无穷多级事件，这就像函数的二阶导数和高阶导数一样，但在实际程序设计中很少出现这样的需求，也是没有必要的，因为计算机中的状态最终都需要表示为离散的状态，相当于数学中的阶梯函数，因此取其一阶导数就已经足够了。

这里可以看出，C++ Builder 中这种对象的模型是对自然界事物基本规律的真实反映，不仅具有静态数据的抽象，还具备了运行动态的描述，是目前最符合实际规律的抽象模型，因而采用 C++ Builder 作为开发工具的时侯，可以获得更高的开发效率，最大限度的提高软件的性能。面向状态的分析方法正是这种抽象模型的分析方法。

3.4 面向状态的特征

关于面向状态的两个重要的概念属性和事件，我们已经有了一定的了解，但要实现面向对象，还需要在程序设计过程中遵循一定的规律，或者是构成规则，这种规则并不是唯一的，但必须存在。仅有属性和事件是不足以组成面向状态的设计技术，接下来将针对 C++ Builder 语言中 VCL 类库，来讲述面向状态的一些基本特征。通过对这些基本特征的了解，读者可以更好的理解面向状态的设计思想。

3.4.1 抽象性

从本书的一开始，我们就强调抽象性，程序设计本身就是一个将现实抽象到计算机的工作，面向对象的时侯就是这样，在面向状态中更是这样。

在对事物的描述中，类的出现和划分就是一个非常抽象的过程，通过类和对象使得程序设计中的结构、逻辑更加符合客观自然规律，更加符合人的思维习惯和认识过程，而状态是对事物静态特征之外的动态特征更加抽象的描述。这比面向对象中只描述静态特征要先进了许多，因此抽象性也将是面向状态中一个突出的特征。

从前面的一些例子中我们可以看出，不管是属性还是事件，最终都是和对象紧密的结合在一起，成为对象本质特征的一部分。一个简单的属性例，如对数据的直接映射：

```
__property Data = {read = FData,Write = FData};
```

这个属性在实际中唯一的作用是可以在 IDE 环境中设置变量的值，除此之外没有任何实际意义。

正确的划分一个对象的状态并合理的创建属性，可以提高程序的质量，相反的，不合理的状态划分和属性表达，则可能使程序的结构更加复杂，更加混乱，达不到应由的效果，而会适得其反。将抽象性列入面向状态的特征之首，正是为了提醒读者，面向状态是来源于现实，不是空中楼阁，正确的使用是应该尊重客观规律，而不能是盲目设计。

3.4.2 封装性

在面向对象的设计方法中，封装性就是其特点之一，面向状态中仍然遵守着面向对象的原有规则，即封装性、继承性和多态性。这里所说的封装性不是面向对象的封装性，而指的是面向状态比面向对象扩充的部分所具有的特征，也就是对象的状态和其表现特征的封装。

这一点主要体现在属性上，因为属性是状态的直接体现，通过前面的例子，相信读者已经不难理解属性的封装性。从表面上看，采用了 C++ Builder 中属性的封装，使程序中改变了一种书写方式，正像前面举 HelloWorld 的例子：

```
IsNewState = !IsNewState;  与
FSetIsNewState (!FIsNewSate) ;
```

是完全等价的，编译后的机器代码是一模一样的，但实际上这种表面形式的不同代表了一种本质的区别。属性 IsNewState 完全可以独立的描述一种状态，而变量 FIsNewSate 就不可以，必须依赖 FSetIsNewState 函数，这种依赖关系会破坏状态的本来含义。

在 C++ Builder 中属性的设置函数和读取函数通常是被声明在__protected:定义域，而属性才是最终对外公开的接口，这是利用了面向对象的封装性对一个对象的行为进行保护，在标准的 C++中，这种受保护的函数不能随便被外界访问，而在 C++ Builder 中则是通过属性的封装性，使得外界可以间接访问受保护对象，从而实现了状态和表现行为的严格一致。

属性的封装性不仅封装了状态的实现过程也就是表现行为，而且还封装了状态的制约条件，这样可以使对象和属性更加独立，没有任何的附加条件。对于外界而言，对一个对象的操作，仅需要知道我要求作什么操作，而不必关心操作之前对象是否已经具备了操作的条件，这些都是由对象自己去完成的。

3.4.3 约束性

约束性来源于状态之间的约束，前面也已经讲过，状态是描述的事物的动态特征，因此状态之间的制约关系是不可避免的。这种状态之间的约束关系在面向状态的程序设计中必须按照一定的方法和规则来体现出来，这就是面向状态的约束性。

约束性是面向状态中必须遵循的的一个规律，否则对于状态的抽象将完全失去意义。一般来说，属性的设置函数应该包括两个部分的内容，一是对于属性描述的状态的核心操作，另一个是对于该状态的约束条件的检查或设置。约束性以及状态之间的约束关系是靠属性的封装性来保证的，下面看几个 VCL 中典型的属性之间的约束关系：

TDataSet 的 Active 属性,由于 Active 属性受到数据库 Active 等其他条件的约束，因此在这个设置函数中，必须对其约束条件进行判断，或者需要对约束条件进行设置。我们在这里不研究 TDataSet 的 Active 设置函数是如何实现的，因为关于 Active 的定义和内涵涉及到 DataSet 的很多方面，关系也非常复杂，仅就其表现特征进行分析。当一个 DataSet 的数据库（DataBase 属性）被指定为一个 TDataBase 组件时，那么 TDataSet 的 Active 属性将受到 TDataBase 的 Active 属性的约束，用户需要打开一个 DataSet 的时侯，并不需要关心当前的 DataBase 是否已经打开，只要执行：

```
TDataSet->Active = true;
```

就可以，TDataBase->Active 的检查和打开均由 TDataSet->Active 的设置函数来完成，相反的如果用户关闭一个数据库：

```
TDataBase ->Active = false;
```

这时候必须先关闭 DataSet，那么 TDataSet->Active = false;则是由 TDataBase ->Active 的设置函数进行检查并完成关闭工作。

在这个例子中可以看出，不同的属性所表示的状态，基本都是各司其职，属性的设置函数所要关心的是我在做某件事之前需要具备什么样的条件，而这个条件是否具备是别人

的事情。这就是状态之间的约束关系。在这个例子中约束关系是双向的，或者说是约束和反约束关系，因为 TDataBase 比 TDataSet 要高一个层次，先有 DataBase 然后才能有 DataSet，而且一个 TDataBase 可能要约束多个 TDataSet。

对于 TDataSet 而言可能还有其他的状态，比如编辑状态（IsEdit），是受到 Active 约束的，只有 Table 打开的情况下才可以处于编辑状态，这几个状态之间的约束关系可以用下面的图 3.11 来表示：

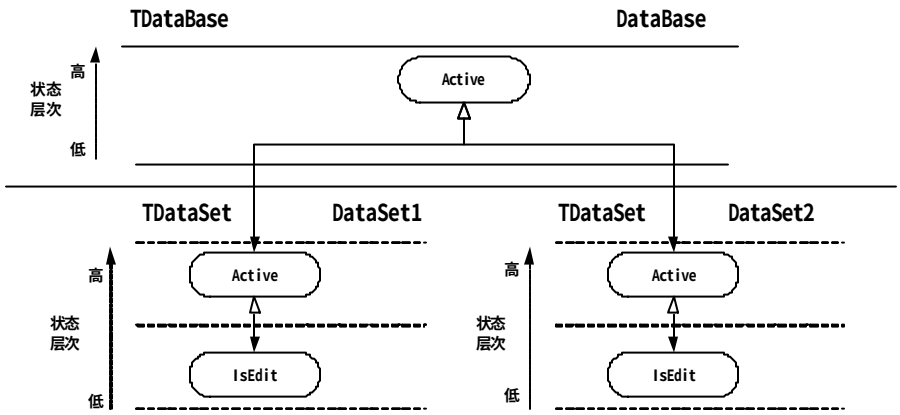


图3.11 状态的约束关系及层次关系

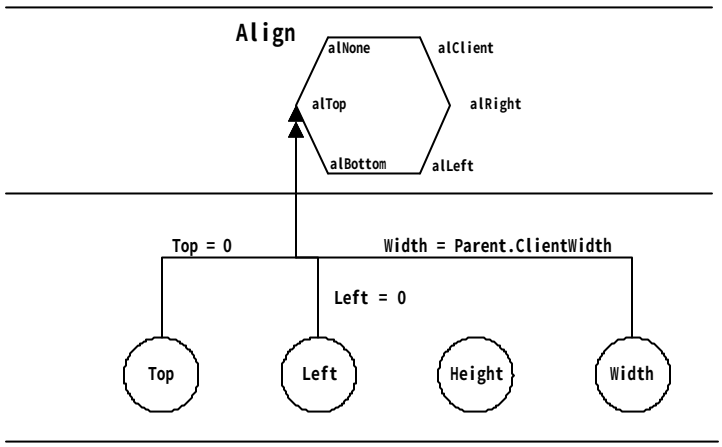
图中圆矩形表示为一个 bool 类型的属性，它有两种状态 true 和 false，箭头表示约束关系，实心箭头表示正逻辑约束，即起点为真是终点为真的必要条件，空心箭头表示反逻辑关系，即起点为假是终点为假的必要条件。建立这样一条约束关系链，并根据我们前面所说的约束性的处理原则，不管 DataSet1 还是 DataSet2 处于编辑状态，当 DataBase 需要关闭的时候，完全不需要考虑，只需要保证 DataSet1 和 DataSet2 的 Active 都为 false，也就是首先关闭 DataSet，在 DataSet1 和 DataSet2 关闭过程中，约束性会自动确保这两个 DateSet 不处于编辑状态，然后再关闭。

这个例子中的约束关系是具有互动性的，是允许约束双方相互改变对方的状态，从其执行的权限上没有高低之分，属于一种平等的关系。在另外一些的约束关系上约束双方不一定的平等的，例如：

在大多数可视化的图形组件中都包含一个“排列”属性 Align，其数据类型是 TAlign，一个枚举类型的数据，包含 alNone、alTop、alBottom、alLeft、alRight、alClient 等取值，从 TControl 派生的组件都缺省的包含该属性，但在某些组件中，这个属性是无效的。对于一个包含 Align 属性的组件，其很多属性都将受其约束，比如 Top、Left、Height、Width，甚至是停靠（Dock）属性等都要受其约束，但是并不允许改变 Align 属性。用图表示它们之间的约束关系如图 3.12 所示：

图中六边形表示 Align 属性有六个独立的状态取值，既 alNone、alTop、alBottom、alLeft、alRight、alClient 等，由于 alNone 实际上对 Top、left、Width、Height 等都没有约束影响，故未画出关系图，圆形表示该属性具有连续的状态取值，比如整型变量、浮点型变量、双精度等。其中双箭头表示连线的起点为终点的必要条件，并且终点对

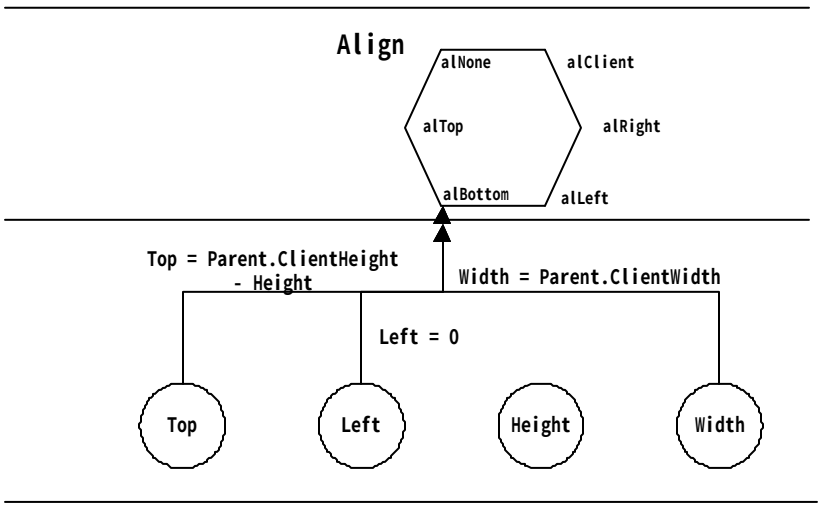
起点存在正向约束关系，不允许起点改变终点的状态。



图：3.12 alTop 的约束关系

在上图中，Align 对 Top、Left、Width、Height 的约束为绝对约束的，因为在 TControl 的属性中，Align 不仅比 Top、Left、Width、Height 层次更高，而且执行级别也要高，程序允许 Align 的状态改变时，相应的改变 Top、Left、Width、Height 的取值，但是不允许它们改变 Align 的状态。上图还显示出 Align 是 Top、Left、Width、Height 的充分条件，当 Align = alTop 时，Top = 0、Left = 0 和 Width = Parent.ClientWidth 是不允许改变的，但并不是必要条件，Align != alTop 时，Top 也可以为 0；

Align 为 aBottom、alLeft、alRight、alClient 时对 Top、Left、Width、Height 的约束关系分别如图 3.13、3.14、3.15：

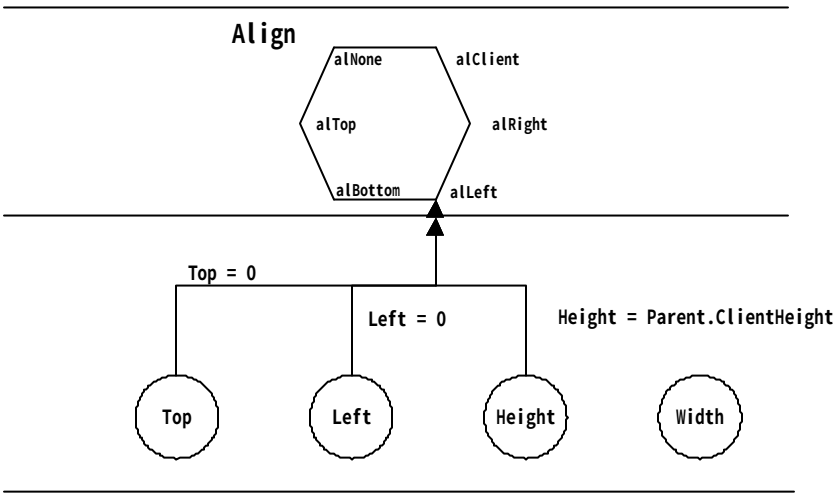


图：3.13 alBottom的约束关系

图中显示 Align = alBottom 是 Top = Parent.ClientHeight - Height、

Left = 0、Width = Parent.ClientWidth 的充分条件，反向为必要条件，Align 不允许被 Top、Left、Width、Height 改变，反之可以。

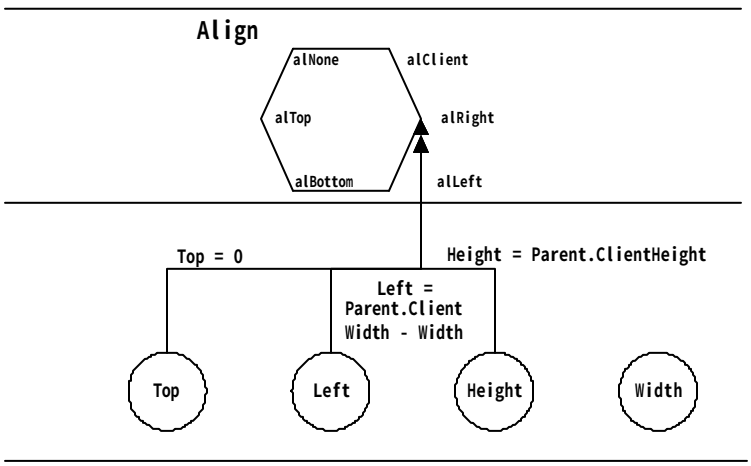
图中的 Height 和 Align = alBottom 无关，即无直接约束关系。



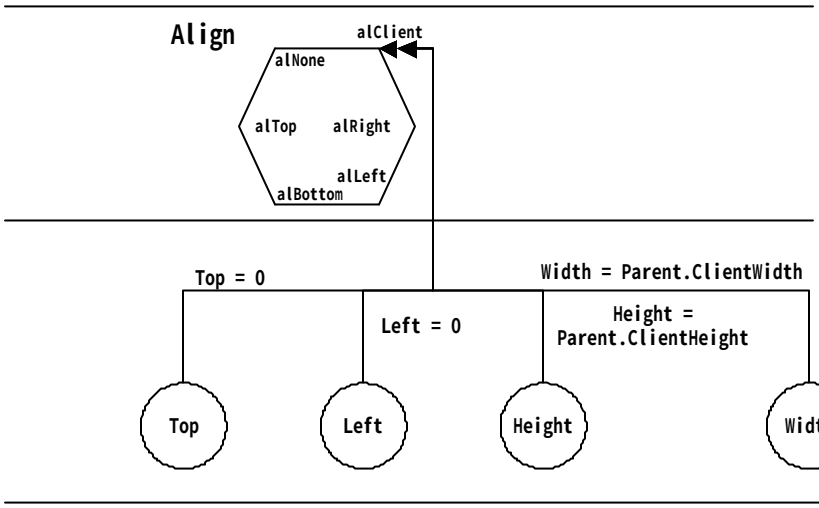
图：3.14 alLeft的约束关系

Align = alLeft 对 Top、Left、Width、Height 的约束关系。其中 Width 和 alLeft 无关。Top、Left、Height 受其约束。

与 Align = alLeft 相似，Left 的取值为 Parent.ClientWidth – Width。



图：3.15 alRight 的约束关系



图：3.16 alClient的约束关系

当 Align = alClient 时，Top、Left、Width、Height 等都受其约束，并满足 Top = 0、Left = 0、Width = Parent.ClientWidth、Height = Parent.ClientHeight。

在上面的几个图中，均没有考虑 Parent 的 Client 区域中包含其它可视组件，或者 TSplitter 组件的情况下。实际中的约束关系要比上图复杂一些，包含了 Parent 的其它组件以及这些组件创建的先后顺序等因素。

关于约束关系还有很多种，而且不适合进行完全的归纳，最重要的是理解这种约束性的存在，并且能够处理各种约束关系。即使同一种约束关系在编写代码的时侯也可以采用不同的方法来处理，只要其逻辑关系没有错，程序就没有漏洞。

实际上 VCL 中处理 TControl 的 Align 属性和 Top、Left、Width、Height 的约束关系时主要采用了两个函数的调用：

SetBounds 和 RequestAlign，关于这两个函数的详细内容读者可以参考 C++ Builder 根目录下的\Source\Vcl\Controls.pas 文件，该单元包含了 Align 属性的原始定义和处理代码。

后面第四节的内容是专门针对 C++ Builder 中 VCL 类库的属性之间存在的制约关系而进行的总结，实际上就是对象的状态之间的约束关系。当然，VCL 中的制约关系并不代表了实际中全部的约束关系类型，第四节的总结也并不一定正确，但通过这些内容，希望读者能够很好的理解状态之间的约束关系的及其处理方法。

3.4.4 完整性

完整性是实现约束性的一个基本要求。

完整性简单来说是指：任何一个程序过程，其约束关系，消息的传递必须是一个闭环的，不允许出现有头无尾的现象。这并不是指在代码的编写上必须完全的前后对应，而是

指在程序设计的时候，必须完整的考虑这种约束关系。

从一方面理解，完整性体现在状态的约束关系存在时，其反约束关系必然也存在，在上面的例子中，DataBase 的 Active 属性和 Table 的 Active 属性就是这种关系，当 Table 设置为 Active 时要求其约束条件 DataBase 的 Active 属性必须成立，反之，当 DataBase 转变状态为非激活状态时，要求 Table 的 Active 属性也必须为 **false**，这就是约束和反约束的关系。在另一个例子中，Top、Left、Width、Height 并不能改变 Align 的状态，代码中会采用不同的处理方式，读者可以参考 Borland 的源代码自行分析，这正是在确保完整性前提下，而采用了更加合理的处理方式，使得最终程序的运行更加符合实际的逻辑关系，这是完整性的另一种表现方式。

从另一方面理解，完整性体现在消息传递的闭合性。也就是说程序在发出一条执行指令的时候，必须是可掌握这个执行过程的有效性，最终是否达到了指令执行的要求，必须将消息反送回去。在 C++ Builder 中默认的情况总是认为执行是有效地，这个消息也并不需要明显的返回，当程序没有成功的执行，或者某些操作失败的时候，才将消息返回，这就像现实中，我们经常说：“你去干什么事吧，有问题了给我打个电话，没问题就不用打了”。在 C++ Builder 中这种类似的处理机制就是“抛出例外”，或者叫“异常处理”，是依靠 **try…catch** 等语句组合来完成的。

这一点也是 C++ Builder 中一个显著的特点，在 Windows 的 SDK 中，大部分 API 函数的返回值，包含着该函数执行情况的信息，程序必须一步一步的逐条判断执行是否成功，然后决定下一步的执行过程，比如在 Windows 系统中为程序分配内存，可能使用的代码如下：

```
LPVOID    lp;      //void 类型指针
HGLOBAL   hp;      //全局对象句柄
hp = GlobalAlloc(GMEM_MOVEABLE,1024);
if(hp)
   lp = GlobalLock(hp);
else
{
   //显示内存分配失败；
   return;
}
if(lp)
   //……
else
{   //显示内存锁定失败
   GlobalFree(hp);
   return;
}
```

这样以来代码会变得冗长而繁琐，而且对于程序而言，并不一定是每个执行过程都需要清楚知道其执行结果，比如上面的内存分配，很可能程序只关心最后是否分配成功，而并不关心是由于什么原因而导致的内存分配失败，显然上面的代码显得过于细致。若采用 C++ Builder 中的 VCL 进行内存的分配，可能会采用如下的代码：

```
try
{
    TStream* lp = new TStream();
    lp->Size = 1024;
}
catch(...)
{
    //分配失败的处理代码
}
```

其中 catch（...）中的“...”可以是很多中类型，这取决于程序的需要，“...”代表了所有的异常都会被捕获，即无论是什么原因产生的“例外”，都将由 catch 语句中的代码进行处理，当然也可以指定例外。这种处理方法显然比采用返函数回值判断的方式更加先进，更加有效，和面向状态的思想最为相符，程序在进行一项操作之前只关心需要什么样的条件，这通过给相应的属性赋值即可，至于该条件是如何实现的，并不是本次操作的需要关心的，要关心的只是该条件是否达到要求，如果没有则按照例外情况来处理。

完整性是体现在设计过程中的，对于设计过程的完整考虑，则允许在实现的代码中出现不完整的情况，这是一个辩证的关系，如何理解这个关系，我们举个例子说明：

在我们前面 Player 的例子中，定义了一个 enum 类型：

```
enum RecState{_STOP = 1,_PLAY,_PREV,_NEXT,_REC};
```

在这里 enum 并不是从 0 开始的，而是从 1 开始，0 是空缺的，而在 RecState 类型的属性中并没有对 0 值的处理。程序在任何状况下启动时，初始状态均不会是 _PLAY,_PREV,_NEXT,_REC 其中之一，确切的讲也不会是_STOP，因为_STOP 是在前几种状态下按停止键后的状态，而程序在初始化时总是将成员缺省的初始化为“0”，因此在 TForm1 的构造函数中我们不需要显式的为 FWorkState 赋值为“0”，这是和实际情况相符合的。相反的假如 RecState 采用了以下的定义：

```
enum RecState{ _PLAY,_PREV,_NEXT,_REC,_STOP,_NONE };
```

其中_NONE 代表了空闲状态，那么从完整性的角度考虑，在 TForm1 的构造函数中必须为 WorkState 进行初始化：

```
FWorkState = _NONE;
```

否则程序将会出现非常大的漏洞。从这里可以看出，面向状态中完整性要求会使代码的逻辑性错误尽可能的少，而且可以设计阶段将状态之间逻辑关系进行约简，最终在保证无逻辑错误的情况下尽量的简化代码，提高程序的执行效率。

3.5 如何面向状态

前面已经介绍了面向状态的基本概念以及面向状态的基本特征，通过对这些基本概念和特征的学习，相信读者对面向状态已经有了一个初步的了解，那么在程序设计中如何来实现面向状态呢？除了遵循面向对象的设计方法之外，面向状态在分析和设计阶段和面向对象有所不同。

分析阶段

在面向状态的设计方法中，分析阶段和面向对象有所不同，主要体现在：面向状态的分析方法中，不仅要考虑对象或者类的成员的静态依存关系，还必须分析对象的运动过程中的运行状态之间的动态约束关系，这是比面向对象更加高级、更加抽象的一种数据模型。

在面向状态的分析过程中，通常是先有静态成员的抽象，然后在根据静态成员的运动特点抽象出描述动态特征的成员，这就是属性。也有一种方式是先根据对象或者根据类要实现的功能，抽象出必要的运行状态，然后根据状态建立属性，最后根据属性的需求在完善整个对象或者类的静态数据成员。这是两种不同的分析方法，但是最终达到的目的是相同的，一个对象或者类的静态特征和动态特征是一致的、统一的，分析过程中，可以根据不同的需要相互调整。

设计阶段

面向状态和面向对象最大的区别就是：采用状态之间的约束关系来处理对象之间复杂的逻辑关系。因此，面向状态的设计过程必须花费很大的精力来处理属性之间的相互制约关系，当属性之间的相互制约关系确定时，实际上整个程序的结构就已经非常清晰。

在面向对象的设计中，编码是在演化阶段进行的，但在面向状态的设计中，这两个阶段的界限竟不是非常的清晰了，面向状态通常可以将代码分作两大类型，一类是描述各种属性之间相互约束关系的代码，可以将其称为框架代码，框架代码描述的一些关系应该是相对固定的，可以在设计阶段进行编码，比如在上面的 `Player` 的例子中，我们给出的代码实际上就是一个框架。另一类是核心操作代码，在一个程序中，核心操作不必要在关心操作会对其他属性产生什么样的影响，不必要再去关心约束条件，在一个完整的系统中，约束条件都是由框架代码处理的，比如 `Player` 例子中没有给出来的播放、录音、快进、快退等操作的核心执行代码，但却能够表达整个程序的逻辑过程。

在面向对象的设计中，设计阶段将会设计出对象和类的原型，包括数据结构和操作方法，而在面向状态的设计中，设计阶段不仅设计出对象和类的原型，还要将各个有关连的对象之间的固定关系，也就是它们之间的动态特征也设计出来。而到了演化阶段，面向状态的设计相对就要简单、容易得多。

总而言之，面向状态是在面向对象的基础上发展而来的，其设计基础仍然遵循面向对象的特征，但面向状态也有自己的特征，设计过程不一定按照一个固定的模式或者方法进

行，最终的目的是为了提高程序的性能和方便设计。面向状态是一个新生的设计思想，在许多设计模式和方法上还有待于进一步完善，这里所讲的也仅仅是给出一个参考。读者可以在实际中完善或提出更加适合设计过程。

4 属性之间的关系

在 VCL 中状态之间的约束关系，可以通过属性的关系中体现出来，而且状态的约束关系最终必须通过属性来体现，前面我们讲述了面向状态的约束性，这一节将针对 VCL 类库中属性的关系进行一些简单的归纳和分类，帮助读者理解状态之间的约束关系。

4.1 同一对象中属性之间的关系

对象是一个相对独立的个体，能够完整的描述某一件事物，状态也必须是对象的状态，因此在属性的关系中，同一对象中属性的关系也是最为突出和重要的。

在 C++ Builder 中对象的属性也可以是另一个对象，比如大多数 VCL 类库中都包含 TFont 对象，从 TControl 派生的类都有一个 Font 属性，其类型为 TFont，TFont 类中又会有其他的属性，它们的关系可以用图 3.17 表示：

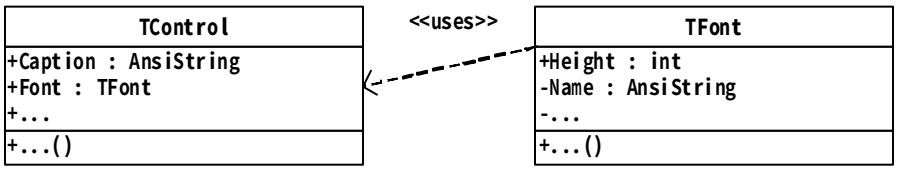


图 3.17 TControl 和 TFont 的关系

在这种关系里面，TFont->Name 和 TControl->Caption 是属于不同的类，但是从 TControl 的层次上分析，Font 本身就是 TControl 的一个属性，因此 Font 的属性 TFont->Name 自然而然也应该是 TControl 的一个属性。而进一步详细划分从 TControl 的属性这一层次分析时，它们仍然隶属不同的类的实例——对象。从这里看出严格区分 TFont->Name 和 TControl->Caption 是否属于同一个对象并没有多达的实际意义。我们作如下的约定：

在 C++ Builder 中已经定义的类，象 TControl 这样的类，其属性的属性和自身的属性，我们都认为是属于同一个对象的属性。

C++ Builder 中不能从个原始定义一次性形成的关系，比如 TToolBar 中的 Button，是需要设计时手工设置或者添加的，将 TToolBar->Button 的属性和 TToolBar 的属性这类关系，认为是不同对象之间的属性。

根据这样的一个约定，我们来看一下同一个对象中属性之间的关系。

4.1.1 并列关系

这是对象的属性中存在最多的一种相互关系，并列关系指的是属性之间没有特定的制约关系，它们的存在不会相互影响。

比如在 TControl 中，属性 Font、Height、Color、Name、Cption 等，之间的关系均为并列关系，甚至包括 Font 的属性 Name、Color、Height 等都和它们是并列的,相互不会影响的,从制约关系上讲它们都是并列的,但从层次上讲 TControl 的 Font、Height 等属于同一个层次,而 Font 的 Name 和 TControl 的 Height 属于不同的层次。

4.1.2 排斥关系

处于排斥关系的属性往往是反映的一个状态的两个或多个不同侧面，排斥关系的属性在同一时刻只能有一个是有效地，而其他的需要忽略而且自动清空。

比如 TDataBase 的 DriverName 和 AliasName，这两个属性从不同的侧面最终描述的都是数据库驱动的信息，当指定了 DriverName 时，AliasName 会自动清空，并依照 Params 属性定义的相关参数数值数据库驱动,当指定了 AliasName 时,DriverName 会自动清空，TDataBase 会根据系统中的别名对数据库的驱动进行配置，并且忽略了 Params 属性关于 DSN、ServerName 等相关的信息。关于 TDataBase 中 DriverName 以及 AliasName 的详细说明请参阅 C++ Builder 的联机帮助。

另一个排斥关系的例子是 TTable 中的 IndexName 和 IndexFieldNames,这两个属性描述的都是 TTable 的关于索引的内容，但同时只能是一个有效。当 IndexName 指定时，IndexFieldNames 将会自动清空，反之 IndexFieldNames 被指定时，IndexName 会自动清空。

我们看一下在 C++ Builder 中如何实现 TTable 的 IndexName 和 IndexFieldNames 的相互排斥关系，VCL 中关于 TTable 的 Object Pascal 的部分源代码如下：

```
function TTable.GetIndexFieldNames: string;
begin
  if FFieldsIndex then Result := FIndexName else Result := '';
end;

function TTable.GetIndexName: string;
begin
  if FFieldsIndex then Result := '' else Result := FIndexName;
end;

//IndexName 和 IndexFieldNames 的读取函数是呈现相反的特型，属性的表现特征为相互排斥的；

procedure TTable.SetIndexName(const Value: string);
begin
  SetIndex(Value, False);
end;
```

```
procedure TTable.SetIndexFieldNames(const Value: string);
begin
  SetIndex(Value, Value <> '');
end;
///IndexName 和 IndexFieldNames 的设置函数殊道同归，最终调用 SetIndex 实现
procedure TTable.SetIndex(const Value: string; FieldsIndex: Boolean);
var
  IndexName, IndexTag: string;
begin
   //……
end;
```

不管 VCL 中是如何实现这种关系，但从最终的表现形式上，也就是最终为用户提供的对外接口来看，它们的排斥关系是显而易见的。TTable 中这两个属性实际上是 TTable 关于索引的两个不同的表现形式。

4.1.3 相互制约

相互制约关系是指属性之间可以相互影响，从层次上讲，处于相互制约的属性可以在同一层次，也可以处于不同的层次。相互制约关系正好体现的是约束与反约束的关系，在 VCL 中，象 TForm 的 Color 和 ParentColor 以及控件的 ShowHint 和 ParentShowHint 都属于这种类型，在程序设计时自定义的属性，大多数会处于这种关系。

以 TPanel 的 ShowHint 和 ParentShowHint 为例，ShowHint 和 ParentShowHint 是从 TControl 继承而来的 bool 类型的属性。当 ParentShowHint 设置为 true 的时侯，ShowHint 必须和父窗口的 ShowHint 一致，ParentShowHint 设置为 false 时，ShowHint 不会发生改变，而当 ShowHint 改变并且与父窗口的 ShowHint 不一致时，则 ParentShowHint 则必须为 false，我们来看一下它们的源代码：

```
procedure TControl.SetParentShowHint(Value: Boolean);
begin
  if FParentShowHint <> Value then
  begin
    FParentShowHint := Value;
    if FParent <> nil then Perform(CM_PARENTSHOWHINTCHANGED, 0, 0);
  end;
end;

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

```
var
  Message: TMessage;
begin
  Message.Msg := Msg;
  Message.WParam := WParam;
  Message.LParam := LParam;
  Message.Result := 0;
  if Self <> nil then WindowProc(Message);
  Result := Message.Result;
end;

procedure TControl.CMParentShowHintChanged(var Message: TMessage);
begin
  if FParentShowHint then
  begin
    SetShowHint(FParent.FShowHint);
    FParentShowHint := True;
  end;
end;
```

在这里，对 ParentShowHint 的设置使用了 TControl 的另一个方法 Perform，这个函数允许绕过 Windows 消息队列使控件可以直接响应指定的消息内容，而最终是调用 WindowProc 对消息进行处理，“ CM_PARENTSHOWHINTCHANGED ”是 VCL 中用户定义的消息，当一个窗口 ShowHint 改变时，会向其子窗口广播该消息。采用这种方式的目的使得 ParentShowHint 为 true 时，ShowHint 和 Parent->ShowHint 也建立约束关系——Parent 的 ShowHint 改变时，自身的 ShowHint 也会随之改变。在控件的 WindowProc 函数中，对“ CM_PARENTSHOWHINTCHANGED ”消息的处理通过调用 CMParentShowHintChanged 函数来完成的，最终也是调用了 ShowHint 的设置函数 SetShowHint。

在这个例子中 ShowHint 和 ParentShowHint 这是一种不完整的制约关系，即：不是所有的条件下都相互约束，在实际中可能存在一种关系，就是两个属性的不同状态下都是相互约束的，称作完整约束，但完整约束意味着这两个属性很可能是等价的，因此实际中这种关系并不多见，往往是为了方便某些应用而将某一个属性在不同的环境重复定义。

4.1.4 单向制约

除了相互制约以外，两个属性之间可能会处于一种单向制约的关系，即：一方对另一方存在制约，反之则不存在制约。比如 TSpeedButton 类的 GroupIndex 和 Down 属性，当 GroupIndex 为“ 0 ”时，Down 属性不允许为“ true ”，而当 GroupIndex 为任意正

整数时，Down 属性则允许为“true”，同时 AllowUp 属性将制约着该组所引的所有 SpeedButton 的 Down 属性是否可以同时弹起来。在这里 GroupIndex 对 Down 属性是一种单向约束，Down 属性对 GroupIndex 属性不具备约束力。TSpeedButton 的 GroupIndex 属性还肩负着另一种职责，这就是在同一个窗体中（或者容器类组件中），处于同一个 GroupIndex 的 SpeedButton 的 Down 属性将会是相互排它的，任何时刻只能有一个 SpeedButton 的 Down 属性为“true”。我们来看一下 TSpeedButton 的部分源代码：

```
procedure TSpeedButton.SetGroupIndex(Value: Integer);
begin
  if FGroupIndex <> Value then
  begin
    FGroupIndex := Value;
    UpdateExclusive;                          //排他性更新调用 UpdateExclusive
  end;
end;            //在此没有对 Down 属性的操作，即 Down 对 GroupIndex 没有反向约束

procedure TSpeedButton.SetDown(Value: Boolean);
begin                                          //Down 属性的设置函数
  if FGroupIndex = 0 then Value := False;   //GroupIndex 为“0”时
                                               //Value 始终应该保持为 false
  if Value <> FDown then    //当 Value 和 Down 属性不一致时，进行以下操作
  begin
    if FDown and (not FAllowAllUp) then Exit;
         //若该 SpeedButton 已经被按下，并且该索引组不允许全部弹起时返回，反之继续；
    FDown := Value;
    if Value then                              //设置后的 Down 属性为“true”时
    begin
      if FState = bsUp then Invalidate;      //原来状态为 Up，则重新绘制；
      FState := bsExclusive   //记录新的状态；State 是另外一个记录组件的属性，
                              //请参阅在线帮助
    end
    else                      //若原来状态不为 Up，则设置并重新绘制；
    begin
      FState := bsUp;
      Repaint;
    end;
    if Value then UpdateExclusive;      //排他性更新，和其他按钮的排它关系
  end;
```

```
end;

procedure TSpeedButton.UpdateExclusive;
var
  Msg: TMessage;
begin     //非“0”索引组，并有父窗体时，传递一个用户自定义消息给父窗体进行广播；
  if (FGroupIndex <> 0) and (Parent <> nil) then
  begin
    Msg.Msg := CM_BUTTONPRESSED;
    Msg.WParam := FGroupIndex;
    Msg.LParam := Longint(Self);
    Msg.Result := 0;
    Parent.Broadcast(Msg);    //消息广播，用于其他子控件的更新。
  end;
end;
```

这里使用了 TWinControl 的一个方法 Broadcast，它可以不经过 Windows 的消息队列而发送相同的消息给每一个由 Parent 属性关联起来的子控件，关于 Parent 属性以及建立的通过其建立的对象之间的关系，参见后面内容。

4.1.5 间接制约

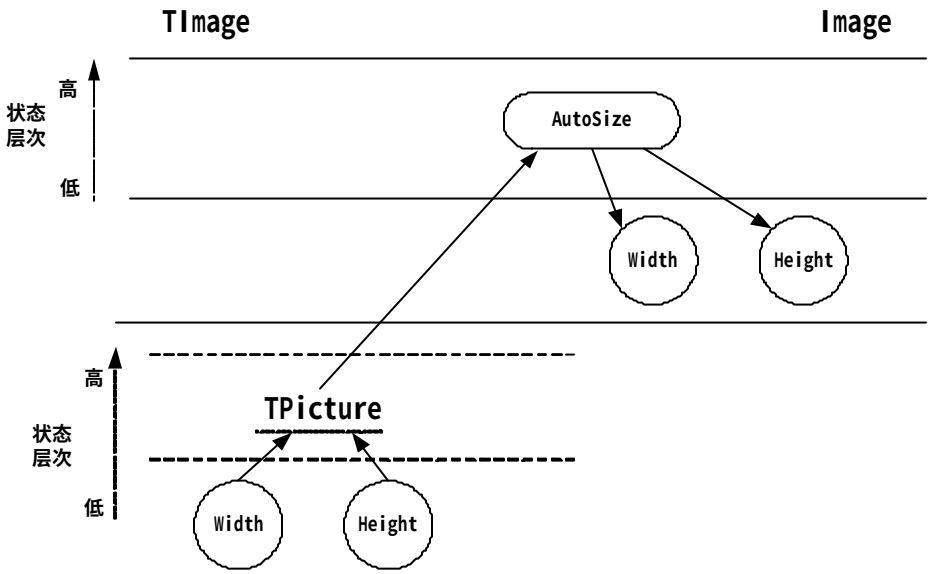
间接制约关系是指两个或者多个属性之间不是直接的建立制约关系，而是通过其他属性而建立的制约关系。

这种关系在 C++ Builder 的 VCL 中也是非常多的，而且处理起来也相对复杂一些，并没有一个特定的方式。比如上面例子中的 TSpeedButton 的 Down 属性，当 GroupIndex 设置为正整数时，同一个窗体中索引相同的 SpeedButton 的 Down 属性之间就会建立相互排它的制约关系，在上面的代码中我们也看到是通过窗体的消息广播机制来完成的。

相类似的例子比如：TImage 的 AutoSize、Height、Width 和 TPicture 的 Height、Width 等，TImage 的 AutoSize、Height、Width 属性是通过继承 TControl 而获得的。TPicture 为 TImage 的一个成员，当 AutoSize 为“true”时，TImage 的 Height 和 Width 属性将和 TPicture 的 Height、Width 相同，而且不允许改动，只有 TPicture 的 Height、Width 改动时，才可以跟随变化。它们的关系可以表示如图 3.18：

实际上 TImage 和 TPicture 关系远比图中复杂得多，单就 AutoSize 和大小属性而言可以简单的表示成图中关系，而且 TPicture 的 Height 和 Width 属性实际上也是受到它的其他属性成员的约束。关于 TImage 的 AutoSize、Height、Width 属性等属性的处理方法涉及到了 TControl 的相关属性，在 TControl 中它们是其派生类的共同

属性，因此在处理方式上需要充分考虑它们的共性，并且必须考虑派生类如何继承，如何重载相关成员函数而获得最佳的处理途径，因此比较复杂，这里不列出来分析，读者有兴趣可以参考 controls.pas 和 extctrls.pas 两个源代码文件。



图：3.18 TImage和TPicture的制约关系

间接制约关系可以是单向制约也可以是双向制约，这与直接的制约关系是比较类似的，这里不再赘述，读者可以参考上面内容理解。

在 VCL 中间接制约关系非常重要，也是容易出现错误的地方，编写控件的时候，这是非常需要注意的，但是在编写应用程序的时候，这种关系可能要少一些，即使有也通常是不同对象之间属性的间接制约。

按照我们上面的约定：VCL 中已经定义的类，其属性的属性和自身的属性，属于同一个对象的属性；而不能从 VCL 的原始定义一次性形成的关系，需要设计时手工设置或者添加的，属于不同对象之间的属性。TImage 和 TPicture 之间的间接制约是同一个对象的属性之间的间接制约，而 SpeedButton 的 Down 属性之间的制约是属于不同对象的属性之间的间接制约。对于不同对象的属性之间的关系，必须结合对象之间的关系进行分析，下面的内容将针对 VCL 中对象之间的关系进行一些简单归纳。

4.2 对象之间的关系

属性是对象的一个重要元素，属性的类型可以是任意的数据类型，也包括 VCL 中的类，从根本上讲对象不可能是一个个孤立的，对象之间必然存在着诸多的联系。根据关联的方式不同，对象之间的关系也可以划分为不同的类型，不同类型的关系在信息传递以及属性的制约上也有所不同。大体上分，对象之间的关系可以分为并列关系、包含关系和逻辑包含关系。逻辑包含本身也是包含关系的一种，但是在 C++ Builder 中这种逻辑包含的关系有着其特殊的意义和作用，VCL 中也采用了几种不同的方式来建立逻辑包含关系，因此这里将逻辑包含分开来讲解。

4.2.1 并列关系

对象的并列关系是指对象完全处于一种等同的位置关系，没有层次的高低之分，也没有其他的逻辑层次关系。

在程序中，可能会有多各 Form 对象，那么这些 Form 之间就是一种并列关系，不存在谁凌驾于谁之上。在同一个 Form 中也会有很多的对象属于并列关系，比如 MainMenu 和 PopupMenu。

另外在 VCL 类库中,也会存在这样的并列关系,比如 TPictrue 组件包含了 Bitmap、Icon、Metafile 等三个不同的属性，这三个属性都是图形类的元素，而且都是由 TGraphic 派生而来，它们在 TPictrue 组件中属于并列关系。

并列关系并不代表着没有相互影响，而且通常并列关系的同一种对象很可能通过其他属性或者上一层对象而建立了某种制约关系，比如 TSpeedButton 就可以通过 GroupIndex 而建立相互制约的关系。

4.2.2 包含关系

包含关系实际上就是一种隶属关系，即一个对象作为另一个对象的成员出现，前面提到的 TControl 和 TControl->Font 就是这样的一种关系，像这样的关系在 VCL 类库中非常多，TPictrue 组件中的 Bitmap、Icon、Metafile 之间相互为并列关系，但它们和 TPictrue 之间的关系就为包含与被包含关系。

包含关系有两种不同的表现形式，一种物理包含，即一个对象直接作为另一个对象的成员出现；另一种形式是引用包含，就是一个对象以指针成员出现，或者说一个对象的引用作为另一个对象的成员出现。

在 C++Builder 的 VCL 中,大多数包含关系都是引用包含,比如 TControl ->Font 的定义形式为：

```
__property TFont * Font;
```

TPictrue 组件中的 Bitmap、Icon、Metafile 属性和 TPictrue 也是引用包含关系，它们的定义形式为：

```
__property TBitmap* Bitmap;
__property TIcon* Icon;
__property TMetafile* Metafile;
```

这种包含关系是 VCL 固有的，正如前面所说的，我们将这些属性和其属性的属性看作是一个对象的属性，但是从对象的内部来分析，这种包含关系仍然是存在的。

在 C++ Builder 中也有一些包含关系是属于物理包含，但相对要少得多，像 AnsiString 类型的属性，几乎在 VCL 中都是直接出现在对象中，比如 Caption 属性、Text 属性等都是 AnsiString 类型的属性。还有一些像枚举类型的属性、TColor 属性等都是直接出现在对象中的，是属于一种物理包含。

严格的讲，引用包含实际上是通过指针或者引用而建立的一种逻辑上的包含，应该属

于下面所说的逻辑包含关系，但是在上面的例子中，这种包含关系是在 VCL 内部建立的，属于 VCL 固有的，通常并不允许改变这种关系，因此我们并不将其归属于逻辑包含。

从设计者的角度来考虑，一个 VCL 的对象一旦建立，它所包含的对象也会相应的建立，或者在需要的时侯会自动建立，设计者完全不需要考虑在代码中或设计时创建这种包含关系，而对于设计者需要在设计时建立的或者在程序中需要通过代码实现的包含关系，我们成作为伪包含或者叫做逻辑包含关系。

4.2.3 逻辑包含关系

逻辑包含关系是指通过某种联系使得多各对象在形式上，存在隶属关系，使一个对象在形式上类似于另一个对象的成员而出现。逻辑包含也可以称作伪包含关系，这是因为逻辑包含不能够反映出真实的对象真实的层次关系，到更像是为了方便使用而定义的别名，或者是实现特定功能而建立的约束关系。从另一方面来讲，逻辑包含关系在为组件建立约束关系的同时，也为满足完整性提供了可能。

逻辑包含是具有客观现实意义的，就像现实中某个人具有双重或多重身份一样，在不同的条件下，处于逻辑包含的对象会有不同的表现形式和含义。通常逻辑包含关系是程序设计者在设计时建立的，或者是在程序运行的过程中通过代码实现的。但并不是所有在设计时或者通过代码建立的都属于逻辑包含，比如 TImage 的 Picture 是一个 TPicture 类型的属性，通常需要设计时指定内容或者执行时载入图片等，但这是一种 VCL 固有的包含关系，当需要的时侯，TImage 会自行创建相应对象并且正确设置。

最常见的逻辑包含关系是通过对象指针来建立的，比如所有 TControl 派生的类都包含一个 PopupMenu 属性，PopupMenu 允许为空，也可以为其指定任意一个 TPopupMenu 对象，甚至是其它 Form 中的 PopupMenu，而且允许程序运行过程中任意的改动，这种关系一旦建立，就会实现其特定的功能。

像这一类的包含关系非常的多，TForm 中的 Menu，TTable 中的 DataBase，TListView 中的 SmallImages、StateImage 等，都属于这种方式建立的包含关系。VCL 中对象指针的属性通常在 Object Inspector 中可以使用下拉框选取，当没有相匹配的对象时，下拉框的选项为空，使用指针建立的包含对象时必须对其有效性进行判断。

在 C++Builder 的 VCL 中，有两个非常重要的逻辑包含关系，一个是由 TComponent 继承而来的 Components 属性和 Owner 属性，另一个是从 TWinControl 继承来的 Controls 属性和由 TControl 继承而来的 Parent 属性。这两组包含关系建立的方式是非常相似的，Components 和 Controls 都是数组类型，Components 数组的元素必须是从 TComponent 继承而来的对象，Controls 数组的元素必须是从 TControl 继承而来的对象，它们分别和包含的控件中的 Owner 属性和 Parent 属性相对应。当一个 TComponent 或其派生类的对象指定了 Owner 以后，它的 Owner 组件的 Components 属性会自动增加一个元素，并且该元素指向这个 TComponent 或其派生类的对象。Controls 和 Parent 和这相似，所不同的是 Owner 和 Components 是一种“拥有关系”，是决定对象的生存关系的属性，Owner 属性一般在一个组件的构造函数中指定，而且不允许改变，是一个只读属性，当一个组件被销毁时，它所拥有的所有组件（Components 数

组列表中的组件），将会被一并销毁。而 Parent 是一个可读写的属性，是用于可视化组件显示关系的属性，能被指定为 Parent 的组件必须是容器类的对象，从 TWinControl 派生而来，当一个控件指定了 Parent 属性以后，它的 Parent 对象的 Controls 属性中会自动增加一个元素，并且该元素指向这个控件，当 Parent 对象移动时，它的子控件位置会跟随移动，Parent 隐藏时，子控件也会跟随隐藏，一个可视化控件没有指定 Parent 时，是不能在屏幕上显示的。

与这种关系类似的状况在 VCL 还有很多,比如 TTable 的 Fields 属性和由 TField 派生的各种数据库字段组件，TPageControl 的 Pages 属性和 TTabSheet 对象，也都是这一种类型的逻辑包含关系。

在 C++ Builder 的 VCL 中，有两个从 TPersistent 派生的 VCL 分支，一个是 TCollectionItem，另一个是 TCollection，它们之间可以建立一种一对多的逻辑包含关系,最典型的例子是 TStatusBar 的 Panels 属性和 TStatusPanels 对象,Panels 为 TStatusPanels 类型的指针，是由 TCollection 派生而来，TStatusPanels 是由 TCollectionItem 派生而来。细心的读者可能都会发现，通过 TCollectionItem 和 TCollection 的派生类们建立的包含关系，其被包含的对象都是在一个特殊的属性编辑器中生成的，而且在单元文件中没有其声明的原型，它的创建是程序根据资源文件自动进行的，因此程序中不能对其进行直接引用，只能通过包含它的对象而间接引用，比如 TStatusBar 的 Panels 只能采用：

```
StatusBar1->Panels->Items[0] ->Text = "Test";
```

的形式进行访问，而不像 TPageControl 中的 Pages 属性和 TTabSheet 对象，可以采用下面形式访问：

```
PageControl1->Pages[0]->Caption = "Test";
```

或者：

```
TabSheet1->Caption = "Test";
```

的形式进行访问。

类似这种关系的像 DBGrid 的 Columns 属性和 TDBGridColumns 对象,TCoolBar 的 Bands 属性和 TCoolBand 对象,TDataSet 的 FieldDefs 属性和 TFieldDef 对象等等。

4.3 不同对象中的属性之间的关系

按照我们前面的约定：不能从 VCL 的原始定义一次性形成的关系，需要设计时手工设置或者添加的，属于不同对象之间的属性。实际上不同对象之间的属性关系才是程序设计时侯最为复杂的处理关系,前面所说的同一个对象之间的属性的关系是 VCL 本身已经建立好的，我们只要理解了并使用正确就行了，但在程序设计时候创建的属性，或者建立不同对象之间的关系，都属于不同对象之间属性的关系。

和同一个和对象之间属性的关系类似，不同对象之间属性的关系也可以分为：排斥关系、并列关系、相互制约、单向制约和间接制约等不同的类型，我们简单作一介绍，读者可以参考同一个对象属性间的关系进行理解。

4.3.1 排斥关系

这种情况应该是比较少见的，排斥关系在这里也是意味着两个属性是描述着同一事物的不同侧面，因此在同一时刻只能有一个是有效地被使用，尽管很少出现，但这种情况也可能出现在应用程序设计或者控件设计的过程中。

4.3.2 并列关系

并列关系指的是属性之间相互没有特别明显的制约关系，而是相互并列存在的，这是程序设计中最常见的一个相互关系。但是并列关系并不是说属性之间不会产生影响，而是这个影响是由程序的其他代码来完成的，它们之间的影响关系并不是固定的，没有典型的代表性。

4.3.3 相互制约

具有相互制约关系的属性，之间的关系是相互影响的，这种关系在程序设计中也是经常出现的，而且在应用程序设计中可能出现的机会更多一些。相互制约关系可以表现为同一层次之间的状态相互约束，也可以表现为不同层次之间的状态存在的约束和反约束关系。

4.3.4 单向制约

单向制约关系，表现为一个属性对另一个属性具有制约关系，反过来被制约的属性并没有对该属性的反向制约关系，在同一个对象中，属性具有这样的关系，在不同的对象中，属性也可以具有这样的关系。

4.3.5 间接制约

同前面讲过的同一对象之间属性的关系一样，间接关系指的是属性之间没有直接的建立制约关系，而是通过其他属性间接的建立了相互制约的关系。这种关系和前面讲的同一对象之间的间接制约关系是完全一样的，不同的是按照是否属于 VCL 本身已经包含的属性关系来区分的。前面说的间接制约从内部分析也是不同对象之间的属性，只是这种关系在 VCL 类库中已经建立，我们将其归属于同一对象的属性，而这里说的间接制约关系是需要在进行程序设计时或者进行控件设计时，由程序员建立的。

5 小结

本章简单的讲述了面向状态的基本思想，并结合实际进行了一些代码的分析，通过将接和代码的分析，帮助读者理解状态的概念、属性的概念以及在 C++ Builder 中关于消息

的处理机制等。

本章第四节针对 VCL 类库，对面向状态中的属性之间的关系进行了一些简单的归纳和总结，其目的是帮助读者深入的了解这种新的程序设计思想，其中部分内容的划分或者归纳并不一定正确，这只是作者个人的理解。而且这种归纳后的关系也并不能代表了所有属性之间的关系，但这种划分可以帮助读者加深理解。在实际的程序设计中，不需要仔细的分析每一个属性和其它属性之间的到底是那种关系，实际中属性的关系可能也比给出的这些关系复杂得多。

尽管本章对面向状态的特征、思想、以及第四节 VCL 类库的中属性的常见关系进行了一些讲解，但是建议读者在阅读时，尽可能多的思考并理解，而在实际程序设计的时侯，不要刻意分析属于那种关系、是否遵循了完整性等，这样会束缚住手脚，正像金庸先生笔下的张无忌，学完太极拳的时候，也全部忘光了，但实际上是掌握了精髓，而不浮于表面。