
北京希望电脑公司出品
C++ Builder
核 心 编 程 技 术
“九五”国家重点电子出版物规划项目
希望计算机知识普及系列

C++ Builder 3核心编程技术

北京希望电脑公司　总策划
徐新华　编著
希望图书创作室　审校

北京希望电脑公司　出品
1998

内 容 简 介

本书全面深入地介绍了当今最热门的编程技术，包括COM、ActiveX、Web服务器应用程序，以及Internet上的WinSock、UDP、HTTP、HTML、FTP、SMTP、POP3、NNTP等协议。本书由16章组成，主要内容包括面向对象编程，组件对象模型(COM)，ActiveX框架，“Type Library”编辑器，创建ActiveX，OLE自动化，使用WinSock,使用FTP控件，使用UDP控件，使用HTTP控件，使用HTML控件，使用SMTP控件，使用POP控件，使用NNTP控件，创建Web服务器应用程序，Web服务器的细节。该书主要是为那些想在Internet/Intranet领域编程的读者写的，不是一本C++ Builder 3的入门参考书，本书虽然是针对应用和开发C++ Builder 3的技术人员编写，但其中很多内容具有普遍性，对使用其他开发工具的读者也有一定的参考作用。

C++ Builder 3核心编程技术
北京希望电脑公司　总策划
徐新华　编著
希望图书创作室　审校
责任编辑　战晓雷
北京希望电脑公司　出品
北京海淀路82号（100080）
北京媛明印刷厂　印刷
新华书店、新华书店音像发行所、各地书店、软件专卖店经销
＊　＊　＊　＊　＊
1998年9月第1版　　1998年9月第1次印刷
开本：787×1092 1/16　　印张：16
字数：368千字　　印数：1-5000
新出音管[1997] 96号
ISBN 7-980008-38-3/TP.09
（1CD，含配套书）

前　　言

作为一个传统的程序员，您也许已经能熟练地使用Delphi、C++ Builder、VB、PowerBuilder等工具开发Windows应用程序和数据库应用程序。随着因特网在国内的迅速流行，许多程序员迫切需要了解有关在Internet/Intranet上编程的知识，而国内关于这方面的书籍很少，事实上，由于技术发展得太快，连那些专业提供开发工具的厂商如Borland、Microsoft、Sybase也是刚刚把Internet/Intranet编程技术加到它们的产品中。本书就是针对使用C++ Builder 3的程序员编写的，深入透彻地介绍C++ Builder 3在Internet/Intranet领域的编程技术。

第一章作为本书的基础篇详细介绍C++ Builder 3面向对象的编程思想。面向对象是C++ Builder 3的精髓，它贯穿着本书的始终。

第二章介绍组件对象模型即COM，包括COM的结构、GUID、CLSID、IID、引用计数、虚拟方法表、IUnknown接口、类工厂、调度接口、双重接口等概念。

第三章介绍Borland引为自豪的ActiveX考技术。ActiveX考实际上是一组有继承关系的类的通称，ActiveX考中封装了许多COM和ActiveX的接口和函数，使开发基于COM和ActiveX的应用变得非常简单，代码具有良好的重用性。

第四章介绍“Type Library”编辑器，这是C++ Builder 3的一个可视化RAD

工具，可以很方便地编辑COM对象中的数据类型、接口和成员，并自动生成相应的代码。

第五章详细讲述ActiveX考、ActiveForm和特性页的创建、注册、安装和在Web上发布等细节，着重从代码级来剖析ActiveX的奥秘。

第六章介绍OLE自动化服务器，先从客户的角度看看怎样操纵自动化服务器，然后实际创建了一个OLE自动化服务器。由于C++ Builder 3采用了ActiveX考技术，实现了与类型信息的无缝连接，因此，创建自动化服务器是很方便的。

第七章到第十四章涉及到Internet/Intranet上的许多领域如WinSock、UDP、FTP、HTML、HTTP、SMTP、POP3、NNTP。

第十五章和第十六章介绍怎样创建Web服务器应用程序，包括Web模块、动作项、HTTP请求消息、HTTP响应消息、HTML模板、页面生成器、访问数据库等内容。

由于本书所讲述的内容比较新，加上本人的水平有限，时间又很紧，书中可能难免有错误，请读者予以批评指正，也欢迎读者随时与我联系，共同探讨技术问题。我的联系电话是010-62987260，E-Mail是p_inprise@mail.263.net.cn。

徐新华

1998.6.20

目　　录

16.8　操纵 Web 服务器应用程序
16.9　Web 服务器与数据库

第一章　面向对象编程

C++ Builder 3 的最大特点就是面向对象，在面向对象领域，Borland®一直处于世界领先的地位。C++ Builder 3 的面向对象主要体现在三个方面，一是编程语言最接近ISO C++标准，二是全面支持基于元件的应用程序开发，三是借助于对象库可实现代码重用。

本章从一些基本概念开始，详细介绍C++ Builder 3面向对象的编程思想，并初步介绍VCL的结构。C++ Builder 3中的元件很多，尽管这些元件千差万别，但它们都是从几个公共基类继承下来的，因此，这些元件具有某种程度的相似性，为了节省篇幅，本书按照这样的思路编写，即先介绍公共基类，后面将只介绍每个元件特有的特性和方法。

1.1　什么是对象

什么是对象？这得先从什么是类说起。类(Class)是一种数据类型，与一般的数据类型不同的是，类除了包含数据以外，还包含了操纵数据的方法，类把数据和方法封装在一起。在一个类中，可以包含不同类型的数据，这一点与C++的结构相似。

类具有可继承性，如果要创建一个新的类，只要选择一个基类再加上自己

的成员就派生出一个新的类。事实上，C++ Builder 3中所有的类都是从一个共同的基类继承下来的，基类的非私有成员自动成为派生类的成员。类的继承还具有传递性，例如，假设T3继承了T2，而T2又继承了T1，可以认为T3也继承了T1。

在C++ Builder 3中，所有的类都是从TObject继承下来的，TObject是一个抽象类，它的派生类可以对TObject的方法包括构造和析构重载，TObject也被称为默认祖先类。

类只是一种数据类型，要使用类，一般还得声明类的变量(某些特殊的情况下可以直接对类进行操作)。需要指出的是，类的变量和类的实例从严格意义上讲，不完全是一个概念。类的实例(Instance)是一块动态分配的内存区域，一般我们把类的实例称为对象(Object)，同一个类，可以有多个实例存在，每个对象都有自己的数据，但方法是共享的。

类的变量实际上是一个指针，指向类的实例，要访问对象的成员，就是通过类的变量来引用的。同一个类可以有多个变量，多个变量可以引用同一个对象。

下面我们还是通过一个典型的程序示例把类和对象的概念讲清楚，当您使用“File”菜单上的“New Application”命令时，C++ Builder 3将创建一个新的项目，默认情况下，这个项目中只有一个空白的Form和一个单元文件及头文件。

在C++ Builder 3中，Form就是一个非常典型的对象，Form的直接基类是TForm，我们先看看头文件中是怎样声明Form的：

```
//---------------------------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
_published:
private:
public:
_fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

可以看出，头文件中声明了一个类叫TForm1，它的基类是TForm。由于Form现在是空白的，也没有建立任何事件句柄，因此，TForm1中除了构造外没有其它数据和方法。
如果您对类的概念理解得比较透彻的话，您就知道，TForm1中实际上是有成员的，因为TForm1是从TForm继承下来的，TForm的成员也就成为TForm1的成员。
声明了TForm1后，头文件中声明了TForm1类型的变量Form1，以后就是用Form1来引用这个Form。注意：在调用TForm1的构造之前，也就是说创建TForm1的实例之前，Form1这个变量基本上是没意义的。
我们再看看单元文件的代码是怎样的：

```
    //---------------------------------------------------------------------------
------
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
```

```
{
}
//---------------------------------------------------------------------------
```
由于现在Form是空白的，单元文件中只有TForm1的构造函数，您可以在构造函数中加入任何代码，只要不引用Form本身，因为在构造函数中，类的实例还没有创建好。

假设您向Form中加入一个按钮(TButton)，名称是Button1，双击此按钮，C++ Builder 3就自动建立一个处理OnClick事件的句柄。这时候，头文件就变成这样：

```
//---------------------------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
_published:
```

```
TButton *Button1;
void _fastcall Button1Click(TObject *Sender);
private:
public:
_fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

可以看出，头文件在TForm1的_published部分声明了一个TButton类型的变量叫Button1，同时声明了一个无返回的函数Button1Click作为处理OnClick事件的句柄。

我们再看看单元文件的代码有什么变化：

```
//---------------------------------------------------------------------------
#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::Button1Click(TObject *Sender)
{

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

可以看出，单元文件唯一的变化就是给出了事件句柄的实体，目前实体中没有代码，您可以加入任何代码，包括对Form的引用。程序示例如下：

```
void_fastcall TForm1::Button1Click(TObject *Sender)
{
Form1->Color = clGreen;
}
```

1.2　修改元件的名称

一般来说，您最好在设计期用Object Inspector修改元件的名称，而且最好在您刚刚把元件加到Form上时就改。在设计期修改元件的名称有一个好处，C++ Builder 3会自动更新源代码中所有引用元件名的地方，也就是说，您不用担心是否有不一致的地方。例如，您可以把Form的名称从Form1改为MainForm，头文件会相应变化：

```
//---------------------------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class TMainForm : public TForm
{
_published:
TButton *Button1;
void _fastcall Button1Click(TObject *Sender);
private:
public:
_fastcall TMainForm(TComponent* Owner);
```

```
};
//---------------------------------------------------------------------------
extern PACKAGE TMainForm *MainForm;
//---------------------------------------------------------------------------
#endif
```
可以看出，所有原先是Form1的地方现在都改为MainForm，包括类名。
完全可以预料，单元文件的代码也改过来了：
```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;
//---------------------------------------------------------------------------
__fastcall TMainForm::TMainForm(TComponent* Owner)
: TForm(Owner)
{

}
```

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

void _fastcall TMainForm::Button1Click(TObject *Sender)
{
Form1->Color = clGreen;
}
//---------------------------------------------------------------------------
```

要注意的是，如果您在事件句柄Button1Click中写了代码，而且还用老名字引用了Form，当您修改Form的名称时，这部分代码不会自动改变，因为代码是您自己键入的。这时候，您就要手动更改对Form的引用，否则编译不能通过。

1.3 对象的作用域问题

在C++ Builder 3中要特别注意对象的作用域问题，因为这关系到对象的成员是否可见和可访问。总的原则是，在类的方法中，类的成员都是可见的，例如，假设Form(类名叫TForm1)上有一个按钮Button1，由于C++ Builder 3把处理Form及Form上元件事件的句柄作为TForm1的一个方法，因此，在处理按钮的OnClick事件的句柄中，您可以任意访问TForm1的所有成员，而且不需要用TForm1的变量来引用，例如：

```
_fastcall TForm1::Button1Click(TObject *Sender);
{
```

```
Color = clFuchsia;
Button1->Color = clLime;
}
```

由于Color是TForm的成员，也就成为TForm1的成员，因此，您可以直接对Color特性赋值，而不需要这么写(尽管这样写编译器也不认为出错)：

```
Form1->Color = clFuchsia;
```

但是，如果要访问Form上某个元件的成员，例如Button1的Color特性，您就必须加上对象限定符，否则编译器就认为您要访问的是TForm1的Color特性。

也许您要问，既然访问按钮的Color特性时要加上Button1限定符，为什么不在Button1前加Form1限定符呢？这是因为，Button1被声明为TForm1的一个数据成员，在类的方法中，类的成员总是可见的、可访问的，因此，不需要加Form1限定符。

假设当前项目中有两个Form，分别是Form1和 Form2，如果要在Form1中访问Form2上的Button2元件，该怎样访问呢？

由于Button2已超出Form1的作用域，因此，在访问Button2时您必须加上Form2限定符，程序示例如下：

```
Form2->Button2->Color = clLime;
```

如果您以为这样就可以了，那就错了，因为Form1根本就不知道Form2是什么东西，您还需要把Form2的头文件包含到Form1的单元文件中。

1.4 类成员的可见性

面向对象编程的重要特征之一就是隐藏复杂性，C++ Builder 3通过_published、private、public、protected和automated等几个关键字使类成员具有不同的可见性，例如：

```
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
_published:
TButton *Button1;
void _fastcall Button1Click(TObject *Sender);
private:
public:
_fastcall TForm1(TComponent* Owner);
};
```

上例中，数据Button1和方法Button1Click是在_published部分声明的，而TForm1的构造函数是在public部分声明的，那么这几个关键字究竟是什么含义呢？

在public部分声明的成员是公共的，也就是说，它们虽然是在某个类中声明的，但其它类的实例也可以访问。例如，假设应用程序由两个Form构成，

相应的单元是Unit1和Unit2，您希望Unit2能共享Unit1中的整型变量Count，您可以把Count在TForm1类中的public部分声明，然后把Unit1的头文件包含到Unit2中。
不过，除非您必须把某个成员在不同类之间共享，一般来说尽量不要把成员声明在类的public部分，以防止程序意外地不正确地修改了数据。
在private部分声明的成员是私有的，它们只能被同一个类中的方法访问，就好象局部变量一样，对于其它类包括它的派生类，private部分声明的成员是不可见的，这就是面向对象编程中的数据保护机制，程序员不必知道类实现的细节，只需关心类的接口。
protected与private有些类似，在protected部分声明的成员是私有的，不同的是，在protected部分声明的成员在它的派生类中是可见的，并且成为派生类的私有成员。
在protected部分声明的成员通常是方法，这样既可以在派生类中访问这些方法，又不必知道方法实现的细节。
在_published部分声明的成员，其可见性与在public部分声明的成员的可见性是一样的，它们都是公共的，不同的是，_published主要用于声明Form上的元件和事件句柄，它是由IDE自动维护的。
在automated部分声明的成员，其可见性与在public部分声明的成员的可见性是一样的，它们都是公共的，唯一的区别在于，在automated部分声明的方法和特性将生成OLE自动化的类型信息。不过，在C++ Builder 3中，automated的作用不大。

1.5 对象的相互赋值

尽管类是一种复杂的数据类型，既有数据又有方法，但是，您完全可以象对一般的变量那样，用赋值号把类的变量赋给另一个类的变量，只要这两个类的类型是一致的或相容的，例如，假设Form1和Form2都是TForm的变量，您可以这么写：

```
Form2 = Form1;
```

因为Form1和Form2的类型都是TForm。

您还可以把派生类的变量赋给基类的变量，例如，假设TMyForm是这样声明的：

```
class TMyForm : public TForm
{
_published:
    TButton *Button1;
    TEdit *Edit1;
    TDBGrid *DBGrid1;
    TDatabase *Database1;
private:
public:
    virtual _fastcall TMyForm(TComponent* Owner);
};
```

然后分别声明了TMyForm和TForm的变量：

```
TMyForm *MyForm;
TForm *AForm;
```

由于TMyForm是从TForm继承下来的，因此您可以这样赋值：

```
AForm = MyForm;
```

不知您注意到没有，每一个事件句柄中都有一个Sender参数，它的类型是TObject，大家知道，VCL的所有对象都是从TObject继承下来的，因此，VCL的所有对象都能赋值给Sender参数，这就是为什么Sender参数能代表所有VCL对象的原因。

Sender参数主要用于在运行期判断元件的类型，程序示例如下：

```
if (typeid(*Sender) = = typeid(TEdit))
DoSomething;
else
DoSomethingElse;
```

1.6 自己创建一个对象

C++Builder 3已经在VCL中声明了众多的对象，不过，有时候您可能需要自己声明一个类，然后再创建类的实例即对象，因为VCL中的对象未必能完全满足您的编程需要。例如，您可以声明一个TEmployee类：

```
class TEmployee : public TObject
{
private:
      char Name[25];
      char Title[25];
      double HourlyRate;
public:
      double CalculatePayAmount( );
      TEmployee( );
};
```

类的声明一般放在头文件中，然后您得在要用到这个类的单元文件中声明类的变量：

```
TEmployee *Employee;
```

TEmployee只是一种数据类型，要使用这个类，您必须创建类的实例：

```
Employee = new TEmployee;
```

new会自动调用TEmployee的构造函数，现在您就可以访问TEmployee中的成员了。

前面讲过，对象实际上是一块内存区域，换句话说就是Windows的资源。如果您不再需要用到TEmployee的对象，您应当及时地删除它，以释放它所占用的资源。

要删除TEmployee的对象，您可以调用delete：

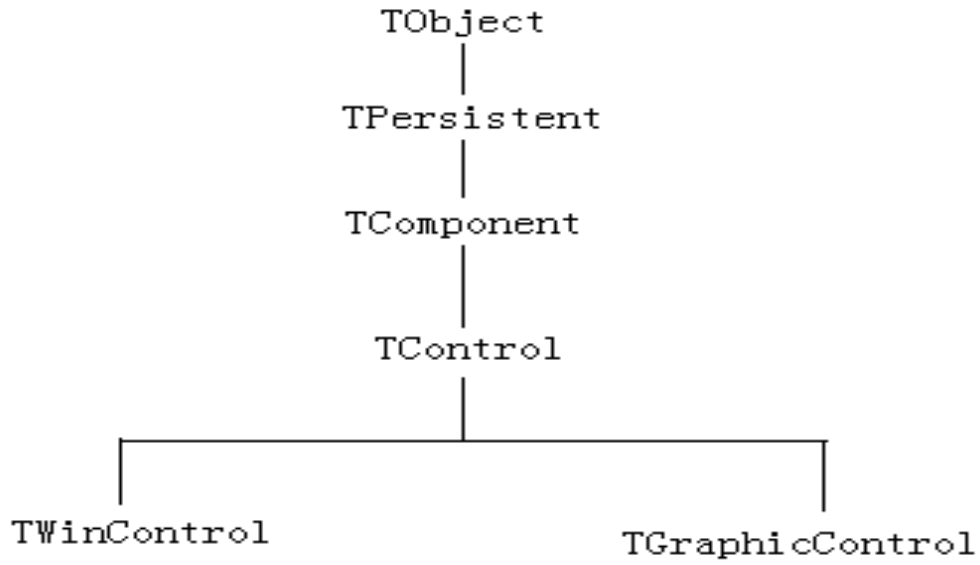
```
delete Employee;
```

不过，您要考虑到这样一种情况，就是如果程序出现异常，程序将非正常退出，这种情况下，程序可能执行不到删除对象的地方。为了保证资源能可靠地得到释放，您就要用到C++的异常处理机制，把删除对象的代码放到try..catch块中。

要说明的是，当您把元件选项板上的元件加到Form上时，C++ Builder 3会自动创建元件的实例，当程序正常退出的时候，C++ Builder 3会自动删除元件的实例，这一切都用不着您自己编写代码，除非您是自己在运行期动态地把一个元件加到Form上。

1.7 VCL 的 结 构

VCL(Visual Component Library的缩写)是C++ Builder 3的核心，VCL是完全面向对象的，VCL中的所有对象都存在着继承与被继承的关系，下图是VCL的示意：



从示意图可以看出，最顶层的是TObject，它是一切对象的基类。
TPersistent是TObject的下一级继承者，它是一个抽象类，主要用于提供对象之间的相互赋值和读写流的能力。
TComponent又是TPersistent的继承者，这个类是VCL中所有元件的祖先类，TComponent定义了所有元件最基本的行为。
尽管所有元件都是从TComponent继承下来的，但直接继承的只有几个非可视的元件如TTimer和TDataSource等，其它元件则是从TComponent的下级TControl继承下来的，从TControl继承下来的元件是可视元件，可视元件也称为控件。TControl定义了所有控件的基本属性，包括特性、方法和事件等。
TWinControl和TGraphicControl这两个类都是从TControl继承下来的，这两个类的区别正如它们取的名字一样，从TWinControl继承的主要是按钮、对话框、列表框等有窗口句柄的控件，这些控件占用Windows的资源，并且允

许用户输入。而从TGraphicControl继承的元件例如TLabel、TSpeedButton等，没有窗口句柄，不占用Windows资源，也不能接受键盘的输入，使用这一类元件的好处在于节约资源。
VCL的面向对象还体现在它的可扩展性，您可以选择其中一个元件作为基类，派生出一个新的类(元件)，并把新创建的元件加入到VCL中。

1.8 TObject

TObject是VCL中所有对象的基类，它定义了操纵对象的基本方法，其中，有的是类方法，用于返回对象的类型信息，有的是虚拟方法，能够在派生类中重载，有的仅用于C++ Builder 3的IDE自身调用，而不能被程序调用。TObject没有数据成员。
请读者注意本书的编写习惯，在介绍一个类时，先介绍类的继承关系，再按字母顺序介绍类的特性、方法和事件，可能会作一些取舍。

AfterConstruction 函数

声明：virtual void _fastcall AfterConstruction( );
应用程序不要直接调用这个函数，因为这个函数是当类的构造函数执行完毕时自动调用的，但您可以重载这个虚拟函数。

BeforeDestruction

声明：virtual void _fastcall BeforeDestruction( );

应用程序不要直接调用这个函数，因为这个函数是在类的构造函数马上就要执行时自动调用的，但您可以重载这个虚拟函数。

ClassName 函数

声明：static ShortString _fastcall ClassName(TClass cls);
ShortString _fastcall ClassName( ){ return ClassName(ClassType( ));}
这个函数返回一个对象实例的类名。注意：您可以用基类的变量来引用派生类的对象实例，用此变量来调用ClassName时，返回的是派生类的类名，而不是变量本身的类型。
假设Form上有一个按钮(TButton)、标签(TLabel)、复选框(TCheckBox)，当用户单击其中一个任何控件，要使控件的类名显示在Form的标题栏上，您当然应当首先为这些控件建立处理OnClick事件的句柄，然后这样写代码：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
Form1->Caption = String(Button1->ClassName( ));
}

void _fastcall TForm1::CheckBox1Click(TObject *Sender)
{
Form1->Caption = String(CheckBox1->ClassName( ));
}
```

```
void _fastcall TForm1::Label1Click(TObject *Sender)
{
Form1->Caption = String(Label1->ClassName( ));
}
```

ClassNameIs 函数

声明：static bool _fastcall ClassNameIs(TClass cls, const AnsiString string);
这个函数用于判断对象的类型是否是指定的类型，如是，就返回True。

DefaultHandler 函数

声明：virtual void _fastcall DefaultHandler(void* Message);
如果一个对象处理某个消息，DefaultHandler能提供对该消息的默认处理。派生类可以重载这个方法，例如，TWinControl就重载了DefaultHandler，并且换名为DefWindowProc。

Dispatch 函数

声明：virtual void _fastcall Dispatch(void *Message);
这个函数用于调用对象的消息处理方法，究竟调用哪个方法，由Message参数指定的消息来决定。如果对象中没有找到处理该消息的方法，Dispatch就从基类中查找，如果一直到最顶层还没有找到，就调用DefaultHandler。
Message是个无类型的参数，从语法上讲，您可以传递任何类型的数据，不过，最好是TMessage结构，它是这样声明的：

```
struct TMessage
{
Cardinal Msg;
union
{
      struct
      {
            Word WParamLo;
            Word WParamHi;
            Word LParamLo;
            Word LParamHi;
            Word ResultLo;
            Word ResultHi;
      };
      struct
      {
            int WParam;
            int LParam;
            int Result;
      };
      };
};
```

FreeInstance 函数

声明：virtual void _fastcall FreeInstance( );
这个函数是由析构函数自动调用的，您一般不要直接调用它，除非派生类重载了NewInstance，您也要相应地重载FreeInstance。
FreeInstance是与NewInstance配对使用的，NewInstance用于建立一个新的对象实例，而FreeInstance用于删除NewInstance建立的对象实例。

InheritsFrom 函数

声明：bool _fastcall InheritsFrom(TClass aClass) {return InheritsFrom (ClassType ( ), aClass);}
这个函数用于判断两个对象的继承关系，如果aClass参数指定的类是对象的基类，这个函数就返回True，否则就返回False。注意这个函数与ClassNameIs函数的区别，ClassNameIs要求类名精确匹配，而InheritsFrom函数只要求是继承关系。

NewInstance 函数

声明：virtual TObject* _fastcall NewInstance(TClass cls);
这个函数用于为对象实例分配内存并返回对象实例的指针。注意：类的构造函数会自动调用NewInstance，应用程序不能直接调用这个函数，但派生类可以重载NewInstance，相应地，派生类也必须重载FreeInstance，因为这两个函数是配对使用的。

SafeCallException 函数

声明：virtual HResult _fastcall SafeCallException(TObject *ExceptObject, void *Except- Addr);
实现COM和OLE接口的类应当重载这个函数以处理异常。

1.9 TPersistent

TPersistent提供了对象之间相互赋值和读写流的能力。TPersistent是一个抽象类，不要直接创建TPersistent的对象实例。TPersistent的大部分方法是从TObject继承的，因此，这里只介绍几个TPersistent特有的方法。

Assign 函数

声明：virtual void _fastcall Assign(TPersistent* Source);
这个函数用于把Source参数指定的对象的数据复制给自己，调用的形式是这样的：

```
Destination->Assign(Source);
```

Source参数的类型是TPersistent，因此，只要是TPersistent的派生类都可以相互赋值，只有一些直接从TObject继承下来的类如TComObject、TAutoObject等才是不可赋值的。
要注意的是，对象之间的相互赋值是有前提的，那就是两个对象必须是赋值相容的，两个对象的类型要么完全一致，要么具有继承关系。

您也可以用赋值语句在对象之间相互赋值，赋值形式是这样的：
Destination = Source;
但要注意，这与调用Assign函数含义不同，赋值语句的含义是用目标变量引用源变量所引用的对象实例，而调用Assign则是把源对象的数据复制给目标对象。

AssignTo 函数

声明：virtual void _fastcall AssignTo(TPersistent* Dest);
这个函数与Assign恰好相反，用于把自己的数据复制给Dest参数指定的对象。

DefineProperties 函数

声明：virtual void _fastcall DefineProperties(TFiler* Filer);
这是个虚拟函数，TPersistent的派生类重载它是为了把对象的非公开数据写到流例如Form文件中，Filer参数指定要写的流。

1.10 TComponent

TComponent是C++ Builder 3中所有元件的抽象基类，它提供了元件的基本特征：元件可以出现在元件选项板上，可以被加到Form上，可以拥有和管理其它元件。
TComponent是个抽象类，不要直接创建它的实例。尽管C++ Builder 3的所

有元件都是从TComponent继承下来的，但直接继承于TComponent的只是少数几个非可视的元件，大部分元件是从TComponent的派生类TControl继承下来的。

ComObject特性

声明：_property _di_IUnknown ComObject;
这个只读的特性返回支持COM的VCL元件的接口引用。如果VCL元件不是作为一个COM组件的外套，访问这个特性将导致EComponentError异常。

ComponentCount特性

声明：_property int ComponentCount;
这个只读的特性返回元件拥有的子元件的数目，例如，对于TForm元件来说，它的ComponentCount特性就是Form上元件的个数。
元件拥有的子元件放在Components特性中，这是个由所有子元件组成的数组，其索引从0开始，程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
AnsiString nameString("TButton");
char compBuf[10];
TButton * button;
for(int i=0; i < ComponentCount; i++)
{
```

```
        if (Components[i]->ClassNameIs(nameString))
        {
                //cast the component to a TButton *
                button = (TButton *)Components[i];
                button->Font->Name = "Courier";
                itoa(ComponentCount, compBuf, 10);
                Edit1->Text = AnsiString(compBuf) + AnsiString(" components");
        }
}
}
```

ComponentIndex 特性

声明： _property int ComponentIndex;
这个只读的特性返回某个子元件在元件数组中的索引，程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
char index[10];
char *string = new char[strlen("The index of the button is ")];
itoa(Button1->ComponentIndex, index, 10);
strcpy(string, "The index of the button is ");
strcat(string, index);
Edit1->Text = string;
```

```
delete string;
}
```

上例把Form1上的Button1元件的索引值转换成字符串后显示在编辑框Edit1中。

Components特性

声明：_property TComponent* Components[int Index];
这个只读的特性就是由元件所拥有的所有子元件组成的数组，程序示例请参照前面。

ComponentState特性

声明：_property TComponentState ComponentState;
这个只读的特性返回元件当前的状态。TComponentState是个集合类型，包含下列元素：csLoading, csReading, csWriting, csDestroying, csDesigning,csAncestor, csUpdating, csFixups。

ComponentStyle特性

声明：_property TComponentStyle ComponentStyle;
这个只读的特性返回元件的风格，如果返回csInheritable，表示这个元件是可继承的。

Name特性

声明：Property Name: TComponentName;
这个特性就是元件的名称。当您把一个元件放到Form上时，C++ Builder 3给这个元件一个默认的名字，如Button1、Edit1等，当然您可以另取一个名字。在同一个项目中，元件的名字不能重名，如果不是很有必要，不要在运行期修改元件的名字。

Owner特性

声明：_property TComponent* Owner;
这个只读的特性返回元件的拥有者。当拥有者被删除时，所有被拥有的元件也将被删除。例如，当Form被删除时，Form上的元件都将被删除。

Tag特性

声明：_property int Tag;
这个特性通常用于给一组元件加上相异的序号，这样您就可以在运行期识别它们。例如，假设Form上有十个按钮，它们有一个共同的事件句柄，为了知道是哪个按钮被按下，您可以事先给这十个按钮编号，然后用Tag特性判断是哪个按钮被按下，程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
if (typeid(*Sender) = = typeid(TButton))
      if (((TButton *)Sender)->Tag = = 1)
```

```
        ((TButton *)Sender)->Caption = "My tag is 1";
}
```

TComponent 构造函数

声明：_fastcall virtual TComponent(TComponent* AOwner);
这个构造函数做了这么几件事情：创建和初始化元件的实例，指定元件的拥有者(由AOwner参数指定)，把ComponentStyle特性设为csInheritable表示此元件是可继承的。
对于元件来说，一般不需要显式地调用这个构造函数，因为当您把一个元件放到Form上时，元件的实例是自动创建的。不过，TComponent的派生类可以重载这个虚拟函数。

~TComponent 析构函数

声明：_fastcall virtual ~TComponent(void);
析构函数用于删除元件的实例，包括该元件所拥有的元件。不过，一般不需要显式地删除元件的实例，因为当应用程序终止时，会自动删除应用程序所拥有的Form，进而删除Form上的元件。当然，如果元件是在运行期动态引入的，您得用delete关键字来删除元件的实例。对于Form来说。最好调用release来释放它的实例。注意：千万不要在元件的事件句柄中删除元件自己或元件的拥有者，否则会导致意想不到的结果。

FindComponent 函数

声明：TComponent* _fastcall FindComponent(const System:: AnsiString AName);
这个函数从元件所拥有的元件(实际上就是从Components数组)中查找一个元件，其名称是AName参数(大小写不敏感)。如找到，就返回这个元件。

FreeNotification 函数

声明：void _fastcall FreeNotification(TComponent* AComponent);
这个函数通知AComponent参数指定的元件：本元件将要被删除了。例如，元件A被赋值给元件B的某特性，当元件A将要被删除时，就要通知元件B。不过，如果元件A和元件B位于同一个Form内，通知是自动进行的。如果两个元件不在同一个Form内，并且存在依赖关系，才需要调用FreeNotification。

GetParentComponent 函数

声明：DYNAMIC TComponent* _fastcall GetParentComponent(void);
这是个动态方法，派生类重载它是为了返回一个特定的“父”类型。

HasParent 函数

声明：DYNAMIC bool _fastcall HasParent(void);
这个函数判断元件是否有“父”，“父”通常是元件所在的容器如Form、Panel等。

InsertComponent 函数

声明：void _fastcall InsertComponent(TComponent* AComponent);
这个函数向元件的Components数组中增加一个AComponent参数指定的元件，这个元件要么没有名字，要么与Components数组中现有的元件相异。
程序示例如下：

```
void MoveToModule(TForm*Source, TDataModule *Dest)
{
int I;
TComponent*Temp;
for (I = Source->ComponentCount - 1; I >= 0; I--)
{
      Temp = Source->Components[I];
      if (dynamic_cast<TControl *>(Temp) == NULL)
      {
            Source->RemoveComponent(Temp);
                  Dest->InsertComponent(Temp);
      }
}
}
```

MoveToModule函数的作用就是把Form上所有非可视的元件移到数据模块上，函数通过一个类型强制转换来判断元件是否属于可视的控件，如不是，

就调用RemoveComponent从Form上删除这个控件，再调用InsertComponent把控件插入到数据模块上。

RemoveComponent函数

声明：void _fastcall RemoveComponent(TComponent* AComponent);
这个函数从元件的Components数组中删除AComponent参数指定的元件。注意：在设计期，无论您把一个元件加到Form上，还是从Form上移走一个元件，InsertComponent或RemoveComponent都是自动调用的。

1.11 TControl

TControl是从TComponent继承下来的，作为C++ Builder 3中所有控件的基类。所谓控件，是指那些在运行期可见的元件，例如，TButton元件在运行期就是个按钮，TEdit元件在运行期就是一个编辑框。对于控件来说，您可以设置它的外观，包括尺寸、位置、光标形状、提示信息，并且能够响应鼠标事件。

Align特性

声明：_property TAlign Align;
这个特性用于设置控件在其容器上的排列方式，TAlign是个枚举类型，可以是以下值：

- alNone　　控件仍然在其原先的位置。

- alTop　控件移到容器的顶部，其宽度总是填满容器的水平尺寸。
- alBottom　控件移到容器的底部，宽度总是填满容器的水平尺寸。
- alLeft　控件移到容器的左部，其高度总是填满容器的竖直尺寸。
- alRight　控件移到容器的右部，其高度总是填满容器的竖直尺寸。
- alClient　控件总是填满整个容器的剩余客户区，这时候如果要选择容器元件，只能在Object Inspector顶端的对象列表中找到容器元件。

BoundsRect 特性

声明：_property Windows::TRect BoundsRect;

这个特性返回控件的边界矩形，该矩形以控件在其“父”控件上的坐标表示，坐标原点就是“父”控件的左上角。程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
TRect MyRect = ActiveControl->BoundsRect;
MyRect.Right = MyRect.Left + 2 * (MyRect.Right - MyRect.Left);
MyRect.Bottom = MyRect.Top + (MyRect.Bottom - MyRect.Top) / 2;
ActiveControl->BoundsRect = MyRect;
}
```

上述代码把Form上当前控件的宽度扩大一倍，高度缩小一倍。

Caption 特性

声明：_property System::AnsiString Caption;
这个特性用于设置控件的标识字符串，默认就是控件的名字。有时候为了兼容键盘操作，您可能需要给控件设一个加速键，例如，如果把一个按钮的Caption特性设为“&Open”，显示在屏幕上就是“Open”，用户按Alt+O就相当于单击这个按钮。

ClientHeight 特性

声明：_property int ClientHeight;
这个特性用于设置控件的客户区高度(以像素为单位)。客户区的高度一般就是控件的高度，但对Form来说，客户区高度等于Form的高度减去标题栏、边框以及滚杆的高度。

ClientOrigin 特性

声明：_property tagPOINT ClientOrigin;
这个特性返回控件的客户区的坐标原点，实际上就是客户区的左上角坐标。

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
char buf[10];
itoa(Button1->ClientOrigin.y, buf, 10);
AnsiString aBuf(buf);
Edit1->Text = aBuf;
```

}

ClientRect 特性

声明：_property Windows::TRect ClientRect;
这个特性返回控件的客户区的矩形，该矩形的左上角坐标总是(0,0)，右下角的坐标就是控件的高度和宽度，相当于Rect(0, 0, ClientWidth, ClientHeight)。程序示例如下：

```
MyControl->Canvas->MoveTo(MyControl->ClientRect.Left, MyControl->ClientRect.Top);
MyControl->Canvas->LineTo(MyControl->ClientRect.Right,
    MyControl->ClientRect.Bottom);
```

上述程序首先把光标移到控件客户区的左上角，再画一根直线到客户区的右下角。

ClientWidth 特性

声明：_property int ClientWidth;
这个特性返回控件的客户区的宽度(以像素为单位)，相当于ClientRect.Right。

Color 特性

声明：_property Graphics::TColor Color;
这个特性用于设置控件的背景颜色，可以是RGB值，也可以是颜色常量，也可以从“颜色”对话框的返回值获取颜色，程序示例如下：

```
Button1->Color = RGB(0,255,0);
Edit1->Color = clMaroon;
if (ColorDialog1->Execute( )) Shape1->Color = ColorDialog1->Color;
```

ControlStyle 特性

声明：_property TControlStyle ControlStyle;

这个特性用于设置控件的风格。一般来说，控件的风格在创建时设好后一般是不改的。

Cursor 特性

声明：_property TCursor Cursor;

这个特性用于设置当鼠标进入控件的客户区时光标的形状，可以设为以下常量：

DesktopFont 特性

声明：_property bool DesktopFont;
如果这个特性设为True，就把Screen对象的IconFont特性赋给控件的Font特性，这是为了保证整个桌面的字体一致。

DragCursor 特性

声明：_property TCursor DragCursor;
这个特性用于设置在进行鼠标拖放操作时光标的形状。

DragMode 特性

声明：_property TDragMode DragMode;
这个特性用于设置鼠标拖放的方式，可以是以下值：

- dmAutomatic　只要在控件上按下鼠标就进入拖放操作，不需要调用BeginDrag。
- dmManual　需要调用BeginDrag，拖放才可以进行。

Enabled 特性

声明：_property bool Enabled;
正常情况下，Enabled特性设为True。如果把Enabled特性设为False，控件就不能响应鼠标、键盘以及定时器，同时控件本身也变灰显示。

Font 特性

声明：_property Graphics::TFont* Font;
这个特性用于设置控件上文本的字体。在设计期，您可以用Object Inspector修改控件的Font特性。要在运行期修改Font特性，程序示例如下：

```
void _fastcall TForm1::FormCreate(TObject *Sender)
{
for (int i = 0; i < Screen->Fonts->Count; i++)
ComboBox1->Items->Add(Screen->Fonts->Strings[i]);
}
```

```
void _fastcall TForm1::ComboBox1Click(TObject *Sender)
{
RichEdit1->Font->Name = ComboBox1->Items->Strings[ComboBox1->ItemIndex];
}
```

上述程序首先在Form创建的时候把屏幕所支持的字体的名称列在组合框ComboBox1中，当用户从组合框中选择一种字体后，RTF编辑器RichEdit1中就相应地改变字体。

Height特性

声明：_property int Height;
这个特性用于设置控件的高度(以像素为单位)，一般就是控件客户区的高度。

Hint特性

声明：_property System::AnsiString Hint;
当鼠标在控件上稍作停留时，控件的旁边将弹出一个提示窗口，一般是显示关于这个控件的简短描述，描述的内容就是由Hint特性设置的，同时将触发Application的OnHint事件，如果程序处理这个事件的话，就可以在状态栏上进一步显示详细的描述。
注意：要显示Hint，控件的ShowHint特性必须设为True。如果控件的ShowHint特性为True，但没有指定Hint的内容，将借用其“父”控件的Hint。

Left 特性

声明：_property int Left;
这个特性是控件相对于它的“父”控件的左上角的横坐标(以像素为单位)。

MouseCapture 特性

声明：_property bool MouseCapture;
如果这个特性设为True，控件就能够捕捉鼠标的动作。

Parent 特性

声明：_property TWinControl* Parent;
这个特性用于设置或返回控件的“父”控件。在设计期，当您把一个控件加到Form或其它容器上时，Form或容器就自动成为该控件的“父”。在运行期，当您通过代码创建了一个控件的实例后，一定要指定这个控件的“父”，程序示例如下：

```
TButton *MyButton;
MyButton = new TButton(Application);
MyButton->Parent = Form1;
```

ParentColor 特性

声明：_property bool ParentColor;
如果这个特性设为True，表示控件使用“父”控件的背景颜色，这是为了使所有的控件都使用相同的背景颜色。如果这个特性设为False，控件将使

用自己的背景颜色。

ParentFont 特性

声明：_property bool ParentFont;
这个特性与ParentColor特性类似。如果ParentFont设为True，表示所有控件都使用其“父”控件的字体，否则就使用自己的字体。

ParentShowHint 特性

声明：_property bool ParentShowHint;
如果这个特性设为True，控件使用其“父”控件的ShowHint特性值，换句话说，只要“父”控件的ShowHint特性为True，所有子控件将显示提示窗口，如果“父”控件的ShowHint特性为False，所有子控件不显示提示窗口。如果ParentShowHint特性设为False，控件就可以自己决定是否要显示提示窗口。

PopupMenu 特性

声明：_property Menus::TPopupMenu* PopupMenu;
这个特性用于指定一个浮动的弹出式菜单，当用户单击鼠标右键时将弹出这个菜单。

ShowHint 特性

声明：_property bool ShowHint;

如果这个特性设为True，当鼠标在控件上稍作停留时，将弹出一个提示窗口，提示的内容由Hint特性设置。

Text 特性

声明：_property System::AnsiString Text;
这个特性并不是所有控件都有，如果有的话，默认就是控件的名称。对于TEdit、TMemo等编辑类控件来说，Text特性就是编辑框中的内容。

Top 特性

声明：_property int Top;
这个特性是控件相对于它的“父”控件的左上角的纵坐标。

Visible 特性

声明：_property bool Visible;
这个特性用于决定控件是否要显示在屏幕上。如果设为True，控件在屏幕上就是可见的，相当于调用Show。如果设为False，在屏幕上就找不到这个控件，相当于调用Hide。

Width 特性

声明：_property int Width;
这个特性是控件的宽度(以像素为单位)。对于TTabSheet来说，改变这个特性是无效的。

WindowProc 特性

声明：_property TWndMethod WindowProc;
其中，TWndMethod是这样声明的：

```
typedef void _fastcall (_closure *TWndMethod)(Messages::TMessage &Message);
```

这个特性用于指定一个函数，用这个函数来响应Message参数指定的Windows消息。在对这个特性赋值之前，首先要保存该特性原先的值(默认是WndProc)。

WindowText 特性

声明：_property char* WindowText;
这个特性与Text特性区别不大，但Text特性可以重载。

BeginDrag 函数

声明：void _fastcall BeginDrag(bool Immediate);
这个函数用于开始拖放。如果Immediate参数设为True，拖放立即开始，光标变成DragCursor特性指定的形状。如果Immediate参数设为False，用户必须拖动一段距离(5个像素)后光标才改变形状开始拖放，这是考虑到有时候用户单击这个控件并不是为了拖放。

BringToFront 函数

声明：void _fastcall BringToFront(void);
这个函数把控件在它的“父”控件范围内推到最前端。如果控件的Visible

特性原来是False，现在也变成True。注意：如果一个窗口类控件被另一个窗口类控件覆盖，或者一个非窗口类控件被另一个非窗口类控件覆盖，调用BringToFront是有效的，但如果是非窗口类控件被窗口类控件覆盖，即使调用BringToFront也无效。
假设有两个Form，分别是Form1和Form2，Form1上有一个按钮，Form2作为一个工具选项板，单击这个按钮使工具选项板出现在屏幕的前端：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
if (!Form2->Visible)
{
Form2->Visible = true;
Form2->BringToFront( );
}
}
```

ChangeScale 函数

声明：DYNAMIC void _fastcall ChangeScale(int M, int D);
这个函数把控件按M/D的比例缩放，同时也改变了控件的位置，相当于同时修改了控件的Top、Left、Width和Height特性。假设要把控件的尺寸增大33%，程序可以这么写：
ChangeScale(133,100);或者ChangeScale(4,3);
如果多次重复调用ChangeScale，每次将在前一次缩放的基础上再次缩放，

最后有可能使控件变得非常大或非常小。

Click 函数

声明：DYNAMIC void _fastcall Click(void);
这个函数模拟鼠标的单击行为，从而触发OnClick事件。程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
Button2->Click( );
}

void _fastcall TForm1::Button2Click(TObject *Sender)
{
MessageBeep(0);
}
```

ClientToScreen 函数

声明：tagPOINT _fastcall ClientToScreen(const tagPOINT &Point);
这个函数用于把以控件的客户区为参照的坐标转化成以屏幕为参照的坐标。
假设Form上有两个编辑框，分别是Edit1和Edit2，当用户在Form上单击鼠标时，就把鼠标所点的位置的X坐标显示在Edit1中，把Y坐标显示在Edit2中，程序这么写：

```
void _fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
TShiftState Shift, int X, int Y)
{
TPoint P, Q;
P.x = X;
P.y = Y;
Q = ClientToScreen(P);
char xCoord[40];
char yCoord[40];
itoa(Q.x, xCoord, 10);
itoa(Q.y, yCoord, 10);
AnsiString xString(xCoord);
AnsiString yString(yCoord);
Edit1->Text = xString + " is the X screen coordinate";
Edit2->Text = yString + " is the Y screen coordinate";
}
```

DblClick 函数

声明：DYNAMIC void _fastcall DblClick(void);
这个函数与Click类似，用于模拟鼠标的双击行为，从而触发控件的OnDblClick事件。

DefaultHandler 函数

声明：virtual void _fastcall DefaultHandler(void *Message);
这个函数提供了处理Message参数指定的Windows消息的默认句柄，如果控件本身没有处理这个消息的话。
DefaultHandler是虚拟函数，派生类以重载它。后面将要讲到，WndProc函数也提供这样的默认句柄，那么这两者之间有什么区别呢？
DefaultHandler是在没有发现有处理某消息的句柄存在的前提下才调用的，而对于WndProc来说，即使有处理某消息的句柄存在，也要首先调用WndProc。

DragDrop 函数

声明：DYNAMIC void _fastcall DragDrop(System::TObject* Source, int X, int Y);
这个函数将调用处理OnDragDrop事件的句柄(如果有的话)。

Dragging 函数

声明：bool _fastcall Dragging(void);
如果控件正在被进行拖放操作，这个函数就返回True。假设Form上有三个复选框，当用户拖动其中一个复选框时，Form的背景颜色就发生改变：

```
void _fastcall TForm1::FormActivate(TObject *Sender)
{
CheckBox1->DragMode = dmAutomatic;
```

```
CheckBox2->DragMode = dmAutomatic;
CheckBox3->DragMode = dmAutomatic;
}
void _fastcall TForm1::FormDragOver(TObject *Sender, TObject *Source, int X, int Y,
TDragState State, bool &Accept)
{
if (CheckBox1->Dragging( )) Color = clAqua;
if (CheckBox2->Dragging( )) Color = clYellow;
if (CheckBox3->Dragging( )) Color = clLime;
}
```

DragOver 函数

声明：DYNAMIC void _fastcall DragOver(System::TObject* Source, int X, int Y, TDragState State, bool &Accept);
这个函数将调用处理OnDragOver事件的句柄(如果有的话)。

EndDrag 函数

声明：void _fastcall EndDrag(bool Drop);
这个函数与BeginDrag配对使用，用于结束拖放，其中Drop参数为True时，表示释放被拖放的对象，Drop为False时表示不释放对象，拖放取消。

GetTextBuf 函数

声明：int _fastcall GetTextBuf(char * Buffer, int BufSize);
这个函数把控件上的文字复制到Buffer指定的缓冲区中，BufSize参数指定要复制的字符个数，函数返回实际复制的字符数。程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
int Size = Edit1->GetTextLen( );
char *Buffer = new char[Size+1];
Edit1->GetTextBuf(Buffer,Size+1);
delete Buffer;
}
```

GetTextLen 函数

声明：Function GetTextLen: Integer;
这个函数返回控件上文本的长度，程序示例见GetTextBuf函数。

HasParent 函数

声明：DYNAMIC bool _fastcall HasParent(void);
如果控件有“父”，这个函数就返回True。一般来说，控件总是有“父”的。

Hide 函数

声明：void _fastcall Hide(void);
这个函数把控件暂时隐藏起来，相当于把Visible特性设为False。

Invalidate 函数

声明：virtual void _fastcall Invalidate(void);
这个函数通知Windows重画控件的表面。

Perform 函数

声明：int _fastcall Perform(Cardinal Msg, int WParam, int LParam);
这个函数向控件自己发一个消息，Msg指定消息，WParam和LParam指定参数。

Refresh 函数

声明：void _fastcall Refresh(void);
这个函数通知Windows重画控件的表面，与Repaint函数的功能相似。

Repaint 函数

声明：virtual void _fastcall Repaint(void);
这个函数通知Windows重画控件的表面，与Refresh函数的功能相似。如果ControlStyle 特性中包含csOpaque元素，这个函数只重画控件自己。如果ControlStyle 特性中不包含csOpaque元素，这个函数还将重画被控件部分遮

住的控件的可见部分。

ScreenToClient 函数

声明：tagPOINT _fastcall ScreenToClient(const tagPOINT &Point);
这个函数把以屏幕为参照的坐标转化为以控件的客户区为参照的坐标。程序示例如下：

```
TPoint ScreenOrgin, ClientPoint;
ScreenOrgin.x = 0;
ScreenOrgin.y = 0;
ClientPoint = Button1->ScreenToClient(ScreenOrgin);
```

上述程序把屏幕坐标(0,0)转换为以Button1的客户区为参照的坐标。

SendCancelMode 函数

声明：void _fastcall SendCancelMode(TControl* Sender);
这个函数用于取消控件的模式状态。假设Form上有一个组合框(TComboBox)，当用户打开它的下拉列表时，组合框就处于模式状态，此时如果用户不“卷”起下拉列表，就不能把输入焦点移走。调用SendCancelMode相当于用户把下拉列表“卷”起。

SendToBack 函数

声明：void _fastcall SendToBack(void);
这个函数与BringToFront正好相反，用于把一个控件放到最后端，如果控件

本来有输入焦点，输入焦点也将被移去。注意：如果是非窗口类控件被窗口类控件覆盖，即使调用SendToBack，窗口类控件始终在非窗口类控件的前端。

SetBounds 函数

声明：virtual void _fastcall SetBounds(int ALeft, int ATop,int AWidth, int AHeight);
这个函数同时设置控件的Left、Top、Width和Height特性。从最终效果看，调用SetBounds与分别修改Left、Top、Width和Height特性是一样的，但后者会使控件多次重画。

SetTextBuf 函数

声明：void _fastcall SetTextBuf(char* Buffer);
这个函数用于把控件上的文字改为Buffer参数指定的文字，程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
Button1->SetTextBuf("You clicked me");
}

void _fastcall TForm1::FormActivate(TObject *Sender)
{
Button1->Caption = "Click me";
```

}

SetZOrder 函数

声明：DYNAMIC void _fastcall SetZOrder(bool TopMost);
这个函数用于改变控件在其“父”控件上的层叠顺序(Z-Order)。如果TopMost参数是True，控件就移到最顶层，如果TopMost参数是False，控件就移到最底层。

Show 函数

声明：void _fastcall Show(void);
这个函数相当于把控件的Visible特性设为True，并把它推向屏幕的前端。

OnClick 事件

声明：_property Classes::TNotifyEvent OnClick
typedef void _fastcall (_closure *TNotifyEvent)(System::TObject* Sender);
这个事件发生在用户单击控件的时候，下列情况下也可能产生这个事件：

- 用户在栅格、列表、树状视图中用键盘上的方向键选择项目。
- 当按钮或复选框具有输入焦点时按下空格键。
- 当前Form上的某个按钮定义为默认按钮时按下Enter键。
- 当前Form上的某个按钮定义为Cancel按钮时按下Esc键。
- 用户按下了按钮或复选框的快捷键。
- 把单选按钮的Checked特性设为True。

- 改变复选框的Checked特性。
- 用户单击Form上的空白处。

OnDblClick 事件

声明：_property Classes::TNotifyEvent OnDblClick;
当用户双击控件时，将触发这个事件。

OnDragDrop 事件

声明：_property TDragDropEvent OnDragDrop;
其中，TDragDropEvent是这样声明的：

```
typedef void _fastcall (_closure *TDragDropEvent)(System::TObject* Sender, System:: TObject* Source,
int X, int Y);
```

当用户释放被拖曳的对象时将触发这个事件，其中，Sender参数是被拖曳的对象释放时的目标控件，Source参数是被拖曳的对象，X和Y参数是鼠标释放时的坐标。
鼠标拖放是非常有趣的。假设Form上有一个列表框和三个标签，三个标签的字体各不相同，当用户拖曳某个标签到列表框时希望列表框的字体变成该标签的字体，示例如下：

```
void _fastcall TForm1::ListBox1DragDrop(TObject *Sender, TObject *Source, int X, int Y)
{
if (Sender->ClassNameIs("TListBox") && Source->ClassNameIs("TLabel"))
{
```

```
TListBox *DestList = (TListBox *)Sender;
DestList->Font = ((TLabel *)Source)->Font;
DestList->Color = ((TLabel *)Source)->Color;
}
}
```

OnDragOver 事件

声明：_property TDragOverEvent OnDragOver;
其中，TDragOverEvent是这样声明的：

```
typedef void _fastcall (_closure *TDragOverEvent)(System::TObject* Sender, System::TObject* Source, int X, int Y, TDragState State, bool &Accept);
```

当用户拖着对象经过控件时将触发这个事件，在事件句柄中，您可以设置控件是否允许接受对象，如果允许的话就把Accept参数设为True。为了直观地表示是否允许接受，通常要改变光标的形状来表示是否允许，要改变光标的形状，修改控件的DragCursor特性。

OnEndDrag 事件

声明：_property TEndDragEvent OnEndDrag;
其中，TEndDragEvent是这样声明的：

```
typedef void _fastcall (_closure *TEndDragEvent)(System::TObject* Sender, System::TObject* Target, int X, int Y);
```

这个事件发生在拖放结束，拖放结束可能是因为用户释放了鼠标或者被取

消。如果拖曳的对象被控件接受，Target参数就是这个控件，否则Target参数为NULL。
注意：上述三个事件中，前二个发生于目标控件上，第三个发生于被拖放的对象。

OnMouseDown事件

声明：_property TMouseEvent OnMouseDown;
其中，TMouseEvent是这样声明的：

```
typedef void _fastcall (_closure *TMouseEvent)(System::TObject* Sender, TMouseButton Button, Classes::TShiftState Shift, int X, int Y);
```

当用户在控件上按下了鼠标将触发这个事件，其中Button参数表示鼠标上哪个按钮被按下，其值可能是mbLeft、mbRight或mbMiddle。Shift参数表示是否有Shift、Ctrl和Alt键按下，其值可能是ssShift、ssAlt、ssCtrl、ssLeft、ssRight、ssMiddle、ssDouble。

OnMouseMove事件

声明：_property TMouseMoveEvent OnMouseMove;
其中，TMouseMoveEvent是这样声明的：

```
typedef void _fastcall (_closure *TMouseMoveEvent)(System::TObject* Sender, Classes::TShiftState Shift, int X, int Y);
```

当用户在控件上移动鼠标的时候将触发这个事件，Shift参数的含义见上。

OnMouseUp 事件

声明： _property TMouseEvent OnMouseUp;
其中，TMouseEvent是这样声明的：

```
typedef void _fastcall (_closure *TMouseEvent)(System::TObject* Sender, TMouseButton Button, Classes::TShiftState Shift, int X, int Y);
```

当用户在控件上释放鼠标时将触发这个事件，注意：释放和按下可能不在同一个控件上。
下面通过一个程序示例帮助您理解OnMouseDown、OnMouseMove和OnMouseUp事件。
假设Form上有一个状态栏，状态栏分成四个窗格。要让用户在Form上用鼠标画一个矩形，并且在状态栏的四个窗格中显示矩形四个角的坐标，程序这么写：

```
int StartX, StartY;

void _fastcall TForm1::Button1MouseDown(TObject *Sender, TMouseButton Button, TShiftState Shift, int X, int Y)
{
StartX = X;
StartY = Y;
}

void _fastcall TForm1::FormMouseMove(TObject *Sender, TShiftState Shift, int X, int Y)
```

```
{
if (Shift.Contains(ssLeft))
{
if (Y > StartY)
{
StatusBar1->Panels[0]->Text = "Top: " + IntToStr(StartY);
StatusBar1->Panels[2]->Text = "Bottom: " + IntToStr(Y);
}
else
{
StatusBar1->Panels[0]->Text = "Top: " + IntToStr(Y);
StatusBar1->Panels[2].Text = "Bottom: " + IntToStr(StartY);
}
if (X > StartX)
{
StatusBar1->Panels[1]->Text = "Left: " + IntToStr(StartX);
StatusBar1->Panels[3]->Text = "Right: " + IntToStr(X);
}
else
{
StatusBar1->Panels[1]->Text = "Left: " + IntToStr(X);
StatusBar1->Panels[3]->Text = "Right: " + IntToStr(StartX);
```

```
}
}
}
void _fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button, TShiftState
Shift, int X, int Y)
{
Form1->Canvas->Rectangle(StartX, StartY, X, Y);
StatusBar1->Panels[0]->Text = "";
StatusBar1->Panels[1]->Text = "";
StatusBar1->Panels[2]->Text = "";
StatusBar1->Panels[3]->Text = "";
}
```

1.12 TWinControl

TWinControl是所有窗口类控件的基类。窗口类控件具有以下几个特征：可以有输入焦点，用户可以用键盘与控件交互，可以作为其它控件的容器，有窗口句柄。不过，窗口类控件实际上并不是直接从TWinControl继承的，而是从TCustomControl继承的。

Brush 特性

声明：_property Graphics::TBrush* Brush;

这个特性返回控件的画布所使用的刷子(TBrush对象)。

ClientOrigin 特性

声明：_property tagPOINT ClientOrigin;
这个特性返回控件的客户区的左上角的屏幕坐标。

ClientRect 特性

声明：_property Windows::TRect ClientRect;
这个特性返回控件的客户区的矩形，相当于Rect(0, 0, ClientWidth, ClientHeight)。

ControlCount 特性

声明：_property int ControlCount;
这个只读的特性返回子控件的数目，子控件的索引是从0开始的，ControlCount特性的值总是比最大索引值大1。

Controls 特性

声明：_property TControl* Controls[int Index];
这个只读的特性是由控件的子控件组成的数组，注意:不要把它同元件的Components特性混淆，假设把一个按钮加到一个分组框内，这个按钮的拥有者仍然是Form，但其“父”控件是分组框而不是Form。我们通过一个程序示例来说明Controls特性的用法。

这个程序的思路是这样的，假设Form上有一个分组框和一个加/减控件(TUpDown)，分组框内有若干个控件，当用户单击加/减控件的箭头时，分组框内的控件就上下移动。

```
void _fastcall TForm1::UpDown1Click(TObject *Sender, TUDBtnType Button)
{
int I;
TControl *ChildControl;
for (I = 0; I < GroupBox1->ControlCount; I++)
{
ChildControl = GroupBox1->Controls[I];
if (Button == btNext)
ChildControl->Top = ChildControl->Top + 15;
else
ChildControl->Top = ChildControl->Top - 15;
}
}
```

Ctl3D 特性

声明：_property bool Ctl3D;
这个特性用于设置控件是否具有3-D的外观，默认是True。

Handle 特性

声明：_property HWND Handle;
这个只读的特性非常有用，它能够返回控件的窗口句柄，在调用Windows的API时可能要用到这个句柄，例如，下面的程序中就调用了Windows的ShowWindow函数：

```
ShowWindow(Form2->Handle, SW_SHOWMINIMIZED);
```

HelpContext 特性

声明：_property Classes::THelpContext HelpContext;
典型的Windows应用程序应当具有这样的功能，当用户在控件上按下F1键时将显示关于此控件的帮助。帮助系统的每一屏都有一个唯一的上下文编号，HelpContext特性就是用于设置上下文编号。如果HelpContext设为0，控件将从它的“父”控件继承上下文编号。

ImeMode 特性

声明：_property TImeMode ImeMode;
其中，TImeMode是这样声明的：

```
enum TImeMode { imDisable, imClose, imOpen, imDontCare, imSAlpha, imAlpha,
imHira, imSKata, imKata, imChinese, imSHanguel, imHanguel };
```

这个特性是C++ Builder 3新加的，用于设置输入方式(IME)。如果只输入英文字母和数字，建议设为imClose或imDontCare。如果要输入汉字，建议设为imChinese。

ImeName 特性

声明：_property System::AnsiString ImeName;
这个特性用于指定输入法，相当于在Windows的“控件面板”中双击“输入法”图标，然后在“输入法”框内选择一种输入法，如“智能ABC输入法”、“微软拼音输入法”等。
如果ImeName特性指定的输入法不存在或不能使用，就用程序启动时的输入法替代。

ParentCtl3D 特性

声明：_property bool ParentCtl3D;
如果这个特性设为True，表示控件使用它的“父”控件的Ctl3D特性，如果ParentCtl3D特性设为False，控件使用自己的Ctl3D特性。

ParentWindow 特性

声明：_property HWND ParentWindow;
这个特性用于指定一个非VCL窗口作为控件的“父”，前提是控件的Parent特性为NULL即原先没有“父”。 这个特性经常用于把ActiveX控件插入到所指定的窗口中。

Showing 特性

声明：_property bool Showing;
这个只读的特性返回控件是否显示在屏幕上。

TabOrder 特性

声明：_property TTabOrder TabOrder;
这个特性设置控件在其“父”控件上的制表键停顺序(Tab Order)，所谓制表键停顺序是指当用户按Tab键时，输入焦点在子控件上依次移动的顺序，默认情况下这个顺序就是控件被加到“父”控件上的顺序。TabOrder从0开始，每个控件的TabOrder值必须是相异的，如果您把一个控件的TabOrder特性设为另一个控件的TabOrder特性值，C++ Builder 3将自动给每个控件重新编号。注意：只有当TabStop特性为True时TabOrder才是有效的。

TabStop 特性

声明：_property bool TabStop;
这个特性用于设置当用户按Tab键时输入焦点要不要移到控件上。假设Form上有3个按钮和1个列表框，如果希望按Tab键时输入焦点只在3个按钮上移动，不希望移到列表框上，您可以把列表框的TabStop特性设为False。

WindowHandle 特性

声明：_property HWND WindowHandle;
这个特性返回控件的窗口句柄，相当于Handle特性，不过，WindowHandle特性是在TWinControl的Protected部分声明的，对于窗口类控件来说，只能在控件内部访问。另一个不同点是，Hnadle特性是只读的，而WindowHandle特性既可以读也可以写。

AlignControls 函数

声明：virtual void _fastcall AlignControls(TControl* AControl, Windows::TRect &Rect);
这个函数在控件的客户区的指定区域按Align特性重新排列某个子控件，AControl参数指定要排列的子控件，Rect指定客户区的某块区域。

Broadcast 函数

声明：void _fastcall Broadcast(void *Message);
这个函数用于向所有的子控件广播一个消息。

CanFocus 函数

声明：bool _fastcall CanFocus(void);
这个函数返回控件是否能接受输入焦点，只有当控件和它的“父”控件的Visible特性和Enabled特性都为True时，CanFocus函数才返回True，否则就返回False。

ChangeScale 函数

声明：DYNAMIC void _fastcall ChangeScale(int M, int D);
这个函数把控件按M/D的比例缩放。

ContainsControl 函数

声明：bool _fastcall ContainsControl(TControl* Control);

这个函数返回Control参数指定的控件是否是本控件的子控件，程序示例如下：

```
    void _fastcall TForm1::Button1Click(TObject *Sender)
{
if(Form1->ContainsControl(ListBox1))
    Label1->Caption = "The form contains ListBox1";
}
```

ControlAtPos 函数

声明：TControl* _fastcall ControlAtPos(const tagPOINT &Pos, bool AllowDisabled);
这个函数返回控件的指定坐标点处的子控件，如果该点处没有子控件，函数返回NULL，AllowDisabled参数指定搜索子控件时是否要包括当前禁止的子控件。

CreateHandle 函数

声明：virtual void _fastcall CreateHandle(void);
这个函数用于为控件建立窗口句柄(如果还没有建立的话)。注意：如果控件的Parent特性为NULL，调用这个函数将触发异常。

CreateParams 函数

声明：virtual void _fastcall CreateParams(TCreateParams &Params);

其中，TCreateParams是这样声明的：

```
struct TCreateParams
{
char *Caption;
int Style;
int ExStyle;
int X;
int Y;
int Width;
int Height;
HWND WndParent;
void *Param;
tagWNDCLASSA WindowClass;
char WinClassName[64];
};
```

这个函数用于初始化TCreateParams结构，也就是给它的每个成员赋值。

CreateParented 构造

声明：constructor CreateParented(ParentWindow: HWnd);
这个函数创建一个控件，作为ParentWindow参数指定的非VCL窗口的子控件。

CreateSubClass 函数

声明：void _fastcall CreateSubClass(TCreateParams &Params, char * ControlClassName);
这个函数用于创建一个窗口类控件，ControlClassName参数指定基类的名称，Params参数用于初始化TCreateParams对象。

CreateWindowHandle 函数

声明：virtual void _fastcall CreateWindowHandle(const TCreateParams &Params);
这个函数创建控件的窗口句柄，Params参数用于传递已初始化的TCreateParams对象。

CreateWnd 函数

声明：virtual void _fastcall CreateWnd(void);
这个函数先调用CreateParams初始化参数，再调用CreateWindowHandle创建控件的窗口句柄，并调整控件的尺寸，再调用Perform传递WM_SETFONT消息来设置字体。

DefaultHandler 函数

声明：virtual void _fastcall DefaultHandler(void *Message);
这个函数提供了对所有消息的默认处理，您可以重载这个虚拟函数。
Message参数是无类型的，您可以强制转换为TMessage类型，再分析出

WParam、LParam和Result。如果Result不等于零，表示消息已处理。如果Result等于零，表示消息还没有处理，您可以处理这个消息，然后您应当把Result设为一个非零数，表示消息已处理。

DestroyHandle 函数

声明：void _fastcall DestroyHandle(void);
这个函数删除控件的窗口句柄，但不删除控件本身。如果控件还有一些窗口类的子控件，这个函数将同时删除这些子控件的窗口句柄。
以后，您可以调用CreateHandle重新创建控件的窗口句柄。

DestroyWindowHandle 函数

声明：virtual void _fastcall DestroyWindowHandle(void);
这个函数删除由CreateWindowHandle创建的窗口句柄。

DestroyWnd 函数

声明：virtual void _fastcall DestroyWnd(void);
这个函数首先保存控件中的文字，释放设备描述表，最后调用DestroyWindowHandle删除窗口句柄。

DisableAlign 函数

声明：void _fastcall DisableAlign(void);
这个函数将暂时禁止控件上的子控件重新排列。当您正在对Form进行缩放

时，为了避免在缩放函数中这些子控件重新排列，可以先调用DisableAlign函数，缩放好后再调用EnableAlign函数重新允许排列，这样可以加快缩放的速度。

EnableAlign 函数

声明：void _fastcall EnableAlign(void);
这个函数允许控件上的子控件重新排列，一般与DisableAlign函数配合使用。

FindNextControl 函数

声明：TWinControl* _fastcall FindNextControl(TWinControl* CurControl, bool GoForward, bool CheckTabStop, bool CheckParent);
这个函数返回控件的一个子控件，该子控件的TabOrder顺序排在CurControl参数指定的子控件的后面。如果CurControl参数指定的不是控件的子控件，函数返回控件的第一个子控件。如果GoForward参数设为True，表示按TabOrder递增的顺序查找。如果CheckTabStop参数为True，表示只查找TabStop特性为True的子控件。如果CheckParent参数为True，表示只查找Parent特性为该“父”控件的子控件。

Focused 函数

声明：bool _fastcall Focused(void);
这个函数返回控件是否具有输入焦点。

GetTabOrderList 函数

声明：DYNAMIC void _fastcall GetTabOrderList(Classes::TList* List);
这个函数用于把子控件(必须是TabStop特性为True的子控件)放在一个列表中。

GetTopParentHandle 函数

声明：HWND _fastcall GetTopParentHandle(void);
这个函数返回它最顶层的“父”控件的窗口句柄，换句话说，这个“父”控件的Parent特性是NULL。处在最顶层的也有可能是非VCL窗口，此时函数就返回这个窗口。

HandleAllocated 函数

声明：bool _fastcall HandleAllocated(void);
这个函数返回控件的窗口句柄是否已存在。注意：您也可以用Handle特性判断控件的窗口句柄是否存在，但是，如果窗口句柄不存在，访问Hnadle特性将自动创建窗口句柄。

HandleNeeded 函数

声明：void _fastcall HandleNeeded(void);
这个函数用于建立控件的窗口句柄(如果句柄不存在的话)。

InsertControl 函数

声明：void _fastcall InsertControl(TControl* AControl);
这个函数用于在控件中插入一个AControl参数指定的子控件，程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
if (Button1->Parent == GroupBox1)
{
     GroupBox1->RemoveControl(Button1);
     GroupBox2->InsertControl(Button1);
}
else
{
     GroupBox2->RemoveControl(Button1);
     GroupBox1->InsertControl(Button1);
}
}
```

Invalidate 函数

声明：virtual void _fastcall Invalidate(void);
这个函数强制控件重新显示一遍，因为控件的某些区域可能已有变化。

PaintTo 函数

声明：void _fastcall PaintTo(HDC DC, int X, int Y);
这个函数把控件在指定的设备描述表的指定位置重画，DC参数指定设备描述表，X和Y参数指定左上角坐标。

PaletteChanged 函数

声明：DYNAMIC bool _fastcall PaletteChanged(bool Foreground);
您可以重载这个函数，当Windows的系统调色板发生变化时，就调用这个函数重新实现控件及其子控件的调色板。有必要的话，还要用新的调色板重画控件。
在同一个时刻，只有一个窗口是活动的。Windows用前景调色板重画活动的窗口，用背景调色板重画其它非活动的窗口。背景调色板与控件的逻辑调色板大致相似，每个控件可以分别实现不同的背景调色板，而前景调色板只能有一个。
如果ForeGround参数设为True，表示Form已经被激活，正在重新实现前景调色板。如果ForeGround参数设为False，表示另外一个应用程序或Form已经改变了前景调色板，凡是与这个前景调色板相关的控件应当重新实现一个新的背景调色板。
如果控件没有建立逻辑调色板，调用GetPalette将返回零，调用PaletteChanged也不会重新实现一个新的调色板，只是把Windows已改变系统调色板的消息通知子控件。如果没有一个子控件建立了逻辑调色板，PaletteChanged函数就返回False。

Realign 函数

声明：void _fastcall Realign(void);
这个函数强制控件重新排列它的子控件。如果每个子控件的Align特性都设为alNone，调用Realign无效。

RemoveControl 函数

声明：void _fastcall RemoveControl(TControl* AControl);
这个函数把AControl参数指定的子控件从Controls数组中删除，换句话说，就是取消了“父子”关系。

ResetIme 函数

声明：void _fastcall ResetIme(void);
这个函数用于恢复应用程序启动时的输入法(IME)。前面讲过，输入法只在控件具有输入焦点时才是有效的，因此，当控件失去输入焦点时，就会自动调用ResetIme。

ResetImeComposition 函数

声明：bool _fastcall ResetImeComposition(int Action);
这个函数让IME窗口进行一个由Action参数指定的任务，Action参数可以设为以下值：CPS_CANCEL、CPS_COMPLETE、CPS_CONVERT、CPS_REVERT。

ScaleBy 函数

声明：void _fastcall ScaleBy(int M, int D);
这个函数把控件按M/D的比例缩放。

ScaleControls 函数

声明：void _fastcall ScaleControls(int M, int D);
这个函数把控件的所有子控件按M/D的比例缩放。

ScrollBy 函数

声明：void _fastcall ScrollBy(int DeltaX, int DeltaY);
这个函数一般不用，因为您可以给控件加上滚杆让它自动滚动，如果您不想用滚杆而要自己滚动，可以调用这个函数，其中DeltaX表示水平方向上滚动的像素数，正数表示向右。DeltaY参数表示竖直方向上滚动的像素数，正数表示向下。

SelectFirst 函数

声明：void _fastcall SelectFirst(void);
这个函数用于选择控件的第一个子控件，如果找到，就激活该子控件。

SelectNext 函数

声明：void _fastcall SelectNext(TWinControl* CurControl, bool GoForward, bool Check-TabStop);

这个函数用于选择CurControl参数指定的子控件的下一个子控件，GoForward、CheckTabStop参数的含义见前。

SetBounds 函数

声明：virtual void _fastcall SetBounds(int ALeft, int ATop, int AWidth, int AHeight);
这个函数用于设置控件的边界矩形，其中ALeft和ATop指定矩形的左上角坐标，AWidth和AHeight分别指定矩形的宽度和高度。

SetChildOrder 函数

声明：DYNAMIC void _fastcall SetChildOrder(Classes::TComponent* Child, int Order);
这个函数用于改变子控件在Controls数组中的顺序，Child参数指定子控件，Order参数指定新的序号。其它子控件的顺序将自动调整。

SetFocus 函数

声明：virtual void _fastcall SetFocus(void);
调用这个函数使控件具有输入焦点。

ShowControl 函数

声明：virtual void _fastcall ShowControl(TControl* AControl);
这个函数使AControl参数指定的子控件是可见的，相当于该子控件调用

Show。

OnEnter 事件

声明：_property Classes::TNotifyEvent OnEnter;
这个事件发生在控件从非活动状态变成活动状态的时候。如果是因为切换Form或切换应用程序导致Form或应用程序上的控件被激活，不会触发这个事件。
对于容器元件如TPanel和TGroupBox来说，当容器上的某个子控件被激活时容器本身也被激活，容器的OnEnter事件发生在子控件的OnEnter事件之前。

OnExit 事件

声明：_property Classes::TNotifyEvent OnExit;
这个事件发生在控件从活动状态变成非活动状态的时候。如果是因为切换Form或切换应用程序导致Form或应用程序上的控件的输入焦点被移走，不会触发这个事件。
对于容器元件来说，当输入焦点从容器的某个子控件移到容器外时，容器本身也失去输入焦点，容器的OnExit事件发生在子控件的OnExit事件之后。

OnKeyDown 事件

声明：_property TKeyEvent OnKeyDown;
其中，TKeyEvent是这样声明的：

```
typedef void _fastcall (_closure *TKeyEvent)(System::TObject* Sender, Word &Key, Classes::TShiftState
```

Shift);

当用户在有输入焦点的控件上按下一个键将触发这个事件，包括功能键、键组合以及鼠标按钮，其中Key参数表示键的虚拟码，Shift参数请参见OnMouseDown事件。

```
    void _fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key, TShiftState
Shift)
{
if (Key = = VK_ESCAPE && Printer( )->Printing)
{
    Printer( )->Abort( );
    MessageDlg("Printing aborted", mtInformation, TMsgDlgButtons( ) << mbOK,0);
}
}
```

这段程序先判断用户按下的是否为ESC键，并且是否正在打印，如是的话，就终止打印。您必须把Form的KeyPreview特性设为True，这样OnKeyDown事件才会被Form1调用。

OnKeyUp 事件

声明：_property TKeyEvent OnKeyUp

这个事件发生在用户释放了键，参数含义见前。程序示例如下：

```
TColor FormColor;
```

```
void _fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key, TShiftState Shift)
{
FormColor = Form1->Color;
Form1->Color = clAqua;
}

void _fastcall TForm1::FormKeyUp(TObject *Sender, WORD &Key, TShiftState Shift)
{
Form1->Color = FormColor;
}
```

上述这段程序的思路是，当用户按下任意一个键，就把Form的背景颜色改为clAqua，当用户释放这个键，就把Form的背景颜色恢复成原来的颜色。

OnKeyPress事件

声明：_property TKeyPressEvent OnKeyPress;
其中，TKeyPressEvent是这样声明的：

```
typedef void _fastcall (_closure *TKeyPressEvent)(System::TObject* Sender, char &Key);
```

当用户按下了某个字符键就触发这个事件，其中Key参数表示被按下的字符。

```
void _fastcall TForm1::FormKeyPress(TObject *Sender, char &Key)
{
```

```
char keyString[25];
KeyString[0] = Key;
strcpy(&keyString[1], " Was Pressed");
Application->MessageBox(keyString , "Key Press", MB_OK);
}
```

1.13 TGraphicControl

TGraphicControl也是从TControl继承下来的，它是C++ Builder 3中所有非窗口类控件的基类。非窗口类控件的特点是：没有窗口句柄，不接受键盘输入，不能作为其它控件的容器，不能用Tab键来选择。典型的非窗口类控件有TBevel、TImage、TPaintBox、TShape、TSpeedButton、TSplitter、TCustomLabel、TToolButton等。
TGraphicControl继承了TControl的所有特性、方法和事件，增加了Canvas特性用于在控件的表面绘图，增加了虚拟的Paint方法用于响应WM_PAINT消息。

Canvas特性

声明：_property Graphics::TCanvas* Canvas;
这个特性返回控件的画布(Canvas)，通过这块画布，您就可以在控件的表面绘图，如画直线、矩形、椭圆、填充或输出文字。Canvas特性是TCanvas对象，该对象中定义了众多的特性和方法，用于实现各种绘图功能。程序示

例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
Image1->Canvas->Brush->Color = clBlack;
Image1->Canvas->Brush->Style = bsDiagCross;
Image1->Canvas->Ellipse(0, 0, Image1->Width, Image1->Height);
}
```

Paint 方法

声明： virtual void _fastcall Paint(void);
TGraphicControl的派生类应当重载这个函数。当非窗口类控件收到WM_PAINT消息，首先要对TCanvas对象初始化，例如设置画笔、刷子等，然后调用Paint画出控件的图像。

1.14 TCustomControl

TCustomControl是个很有意思的类，它是从TWinControl继承下来的抽象类，但它又有Canvas特性和Paint方法，也就是说，TCustomControl同时具有窗口类控件和非窗口类控件的特征。如果要创建自定义的元件，一般是选TCustomControl作为祖先类。

Canvas 特性

声明：_property Graphics::TCanvas* Canvas;
这个特性就是控件的画布，通过这块画布，您就可以在控件的表面绘图。

PaintWindow 函数

声明：virtual void _fastcall PaintWindow(HDC DC);
当控件收到WM_PAINT消息时就会自动调用这个函数，PaintWindow提供了画布要使用的设备描述表(DC)，并且调用Paint重画控件的表面，最后删除设备描述表。

Paint 函数

声明：virtual void _fastcall Paint(void);
这个函数是由PaintWindow调用的，用于重画控件的表面。

第二章　组件对象模型(COM)

组件对象模型(Component Object Model，简称COM)是组件对象之间相互接口的规范，凡是遵循COM接口规范的对象彼此之间能相互通讯和交互，即使这些对象由不同的厂商、不同的语言、不同的Windows版本甚至在不同的机器上编写和建立。

C++ Builder 3自动支持COM接口规范，也就是说，用C++ Builder 3创建的对象可以与用Visual Basic、Java、Delphi及其它语言实现的对象交互。本章详细介绍组件对象模型的体系结构，并实际创建一个COM服务器。

2.1　几个基本概念

软件重用是业界追求的目标，人们一直希望能够象搭积木一样随意“装配”应用程序，组件对象就充当了积木的角色。所谓组件对象，实际上就是预定义好的、能完成一定功能的服务或接口。问题是，这些组件对象如何与应用程序、如何与其它组件对象共存并相互通讯和交互，这就需要制定一个规范，让这些组件对象按统一的标准方式工作。

COM是个二进制规范，它与源代码无关，这样，即使COM对象是由不同的编程语言创建的，运行在不同的进程空间和不同的操作系统平台，这些对

象也能相互通讯。COM既是规范，也是实现，它以COM库的形式提供了访问COM对象核心功能的标准接口以及一组API函数，这些API用于实现创建和管理COM对象的功能。

COM本质上仍然是客户/服务器模式，客户(通常是应用程序)请求创建COM对象并通过COM对象的接口操纵COM对象，服务器根据客户的请求创建并管理COM对象。客户和服务器这两种角色并不是绝对的。

组件对象与一般意义上的对象既相似也有区别，一般意义上的对象是一种把数据和操纵数据的方法封装在一起的数据类型的实例，而组件对象使用接口(Interface)而不是方法来描述自己并提供服务。所谓接口，其精确定义是“基于对象的一组语义上相关的功能”，实际上一个纯虚类，真正实现接口的是接口对象(Interface Object)。一个COM对象可以只有一个接口，也可以有许多接口，例如，ActiveX控件一般就有多个接口，客户可以从很多方面来操纵ActiveX控件。

接口是客户与服务器通讯的唯一途径，客户只能通过接口访问接口对象中的方法，而不能访问该对象中的数据。如果一个组件对象有多个接口，通过一个接口不能直接访问其它接口，但是，COM模型提供了这样一种策略，就是允许客户调用COM库中的QueryInterface函数去查询组件对象所支持的其它接口，从这个意义上讲，组件对象有点象接口对象的经纪人(Broker)。当客户调用QueryInterface函数后，如果组件对象正好支持要查询的接口，函数就返回该接口的指针。如果组件对象不支持该接口，函数就返回一个出错信息。所以，QueryInterface函数是很有用的，它可以动态了解组件对象所支持的接口。

接口实际上是面向对象编程思想的一种体现，它隐藏了COM对象实现服务的细节，COM对象可以完全独立于访问它的客户，只要接口本身保持不变。如果实在需要修改和更新接口方式，只要重新增加一个新的接口，这样，对于只知道老接口的客户来说，就得到了最大程度的保护和兼容。

2.2 客户和服务器

COM本质上仍然是客户/服务器模式。COM客户通常是EXE，也可能是DLL，甚至就是Windows自己。COM客户一般应独立于COM服务器，因为COM客户并不知道COM服务器在哪儿、怎样把它唤醒，甚至有没有这样的COM服务器都不知道。

要得到某个COM服务器的组件对象的服务，客户首先要请求Windows去查找组件对象在哪儿，找到以后把接口的指针传递给客户。那么Windows是怎么知道COM服务器在哪儿的呢？原来，它是从系统注册表中查找COM服务器的位置。反过来说，COM服务器必须在Windows的注册表中登记注册。

COM服务器实际上是组件对象的容器，根据COM服务器与COM客户是否运行在同一个进程地址空间，COM服务器分为三类：In-Process服务器、Local服务器和Remote服务器。In-Process服务器通常是DLL，它映射到客户的进程地址空间中运行。Local服务器通常是EXE，有它自己的进程空间，但与COM客户在同一个机器上。Remote服务器可以是EXE也可以是DLL，它运行在与COM客户不同的机器上。

对于Remote服务器来说，它已经跨越了机器边界，如果仍然用COM来描述

这种模型显然是不够的，Microsoft扩展了COM模型，推出了称为Distributed COM(简称DCOM)的模型。DCOM最初是在Visual Basic 4.0的专业版中被引入的，目前只有Windows NT 4.0才真正实现了DCOM，Windows 95必须另外安装一个程序才能支持DCOM。

为什么客户能够跨进程边界甚至跨机器边界访问组件对象，这就是COM模型中一种称为Marshaling的策略。Marshaling实际上是一种打包技术，先把要跨边界传递的函数调用信息进行必要的转换并打包，然后传递给另一个进程甚至另一个机器，到达目的地后再解包并调用当地的组件对象。不过，这种跨进程边界或机器边界的传递是相当费劲的。如果COM服务器是DLL的话，就用不着Marshaling，因为客户和COM服务器在同一个进程地址空间，这种情况下，客户只要搜索虚拟方法表。

2.3 认识GUID、CLSID、IID

一个复杂的系统中，可能充斥着大量的组件对象，每个组件对象可能又有大量的接口，为了保证这些接口彼此不会冲突，Microsoft规定用GUID来标识组件对象。所谓GUID，是Globally Unique Identifier的缩写，意为全局唯一标识符。GUID可以标识组件对象的类，这时候GUID也称为CLSID(Class Identifier的缩写)。GUID也可以标识组件对象的接口，这时候GUID也称为IID(Interface Identifier的缩写)。

GUID是一个128位整数(16字节)，绝对能保证每一个组件对象具有唯一的GUID。

2.4 引 用 计 数

引用计数是一种机制，使组件对象具有一定的“智能性”，它的工作原理是这样的：当接口对象第一次创建时，引用计数的初始值为1。当再有一个客户请求获得接口对象的指针时，就调用AddRef，使该计数加1。当某客户不再需要组件对象的服务时，它应当调用Release，注意，Release并不真正释放接口对象，因为可能还有其它客户正在使用，Release只是使引用计数减1。只有当引用计数正好减为零时，接口对象才被删除。

下面我们举例来说明引用计数的作用，假设客户A向服务器请求IMalloc接口，服务器收到请求后，首先看该接口对象是否存在，如果没有，就创建一个接口对象，并调用AddRef使引用计数变为1，同时把该接口对象的指针传递给客户A。假设这时候客户B也加入进来，并且也是请求IMalloc接口，由于此时IMalloc接口对象已存在，服务器只是简单地返回一个指针，并且调用AddRef使引用计数变为2。当客户A不再需要IMalloc接口时，它就调用Release试图释放这个接口。显然，这时候不能删除IMalloc接口对象，因为客户B还在用着呢。正是有了引用计数这种机制，才使得服务器知道怎样管理自己的接口、何时该删除接口对象。引用计数这种机制也带来一个问题，就是调用AddRef和Release不能出现混乱，一旦出现混乱，可能导致接口对象永远不被删除或者过早地被删除。

2.5 什么是IUnknown接口

Addref、Release、QueryInterface这三个函数是COM对象提供的三个核心服务，这三个函数构成IUnknown接口。在C++ Builder 3中，IUnknown接口是这样声明的：

```
IUnknown = Interface
['{00000000-0000-0000-C000-000000000046}']
Function QueryInterface(const IID: TGUID; out Obj): Integer; Stdcall;
Function _AddRef: Integer; Stdcall;
Function _Release: Integer; Stdcall;
End;
```

注意：本书有的地方按照Object Pascal的语法来声明接口，请读者自己与C++对照。

正如TObject是C++ Builder 3中所有类的基类一样，IUnknown接口是COM对象所有接口的基本接口，这样就能保证凡是取得了接口对象指针的客户总是能访问COM对象的核心服务如AddRef、Release和QueryInterface，这三个核心服务管理着接口对象的生存期。

_AddRef函数和_Release函数比较简单，都没有参数，返回类型都是整型。而QueryInterface函数比较复杂，它有两个参数，IID参数给出要查询的接口标识符，Obj是一个无类型参数，如果COM对象不支持所查询的接口，Obj参数返回NULL。

这里顺便介绍一下COM模型中称为Interface Aggregation即接口聚合的概

念，前面我们在介绍C++ Builder 3的面向对象编程思想时讲到，通过继承(Inheritance)可以实现软件重用。而在COM模型中没有继承的概念，而是通过Interface Aggregation技术把多个接口聚合起来，共同完成某一复杂的功能。

2.6 DLL形式的COM服务器

COM服务器通常是DLL形式，DLL中应当输出四个例程让Windows调用，一个是DLLCanUnloadNow函数，用于判断DLL当前能否卸载，这个函数通常这样定义：

```
STDAPI _export DllCanUnloadNow(void)
{
return (_Module.GetLockCount( )= =0) ? S_OK : S_FALSE;
}
```

这个函数首先调用GetLockCount函数，如果COM服务器中至少有一个组件对象在被引用，DllCanUnloadNow函数就返回S_FALSE，表示此时不允许卸载。如果GetLockCount函数返回零，DllCanUnloadNow函数就返回S_OK，表示DLL可以卸载了。

DLL中要输出的另一个例程是DLLGetClassObject函数，这个函数在Windows把DLL调入内存后立即调用，用于判断客户的请求是否应该响应以及如何响应。

```
STDAPI _export DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
{
return _Module.GetClassObject(rclsid, riid, ppv);
}
```

DLL中还要输出DllRegisterServer函数用于注册COM服务器：

```
STDAPI _export DllRegisterServer(void)
{
return _Module.RegisterServer(TRUE);
}
```

DLL中还要输出DllUnregisterServer函数用于撤消注册COM服务器：

```
STDAPI _export DllUnregisterServer(void)
{
return _Module.UnregisterServer( );
}
```

要让Windows能检索到COM服务器，COM服务器必须在Windows的注册表中登记注册。您可以借助于一个实用程序REGSVR32.EXE，这是个命令行程序。不幸的是，这个程序只能在Windows NT 3.51或4.0中找到，Windows 95却没有这样的程序。

如果没有REGSVR32.EXE程序，您可以用一个文本编辑器建立一个“注册表项目”文件，扩展名是.REG。注册表项目文件应当遵循一定的格式，您可以参考下面的例子：

```
REGEDIT4
```

```
[HKEY_CLASSES_ROOT\CLSID\{0AA17140-310E-11D0-A45E-444553540000}]
@="MyCOMSever"
[HKEY_CLASSES_ROOT\CLSID\{0AA17140-310E-11D0-A45E-
444553540000}\InProcServer32]
@="C:\\Borland\\C++ Builder 3\\COMServer\\MyCOMServer.DLL"
```

建立了注册表项目文件后，您可以在资源管理器或文件管理器中双击这个文件，Windows就会把注册表项目文件中的信息加到注册表中。

注册好COM服务器后，您可以打开Windows的注册表查看COM服务器的注册情况，在HKEY_CLASSES_ROOT\CLSID下找到该服务器的CLSID。找到该服务器的CLSID后，您还可以看到这样的子主键：InProcServer32，这个子主键的键值就是COM服务器的路径。

2.7 接　　口

所谓接口，实际上就是一组特性和方法，通过调用这些特性和方法，外部代码就可以访问对象，并且不需要知道对象具体是怎样实现的。实际上，接口本身并不实现自己，接口的方法都是纯虚的。真正实现接口的是另外一个类，我们称为接口对象。

在C++ Builder 3中，接口自动与COM(Component Object Model)规范兼容，也就是说，用C++ Builder 3创建的对象具有COM对象的一切特征。由于COM具有语言无关性，因此，C++ Builder 3的对象可以与Java、Visual Basic、Delphi

的对象彼此交互。
声明接口时可以指定一个祖先接口。如果祖先接口省略不写，表示继承的是IUnknown接口，也就是说，IUnknown是默认祖先接口，就好象TObject是默认祖先类一样。
IUnknown提供了COM库的核心服务，包括引用计数和查询其它接口的功能。
任何一个接口，除了它自身的成员外，还继承了它的祖先接口中的成员，当然肯定包括IUnknown接口中的成员。
要说明的是，在COM中，继承并不是代码重用的手段，因为虽然一个接口从它的祖先接口中继承了方法，但是并没有继承实现祖先接口的代码，也就是说，接口必须重新实现祖先接口中的方法，就好象祖先接口中的方法都是纯虚的一样。
声明了接口后，还要具体实现接口才能被外部代码访问。要实现接口，先要声明一个特殊的类，这个特殊的类我们称为实现类。实现类的祖先一般是TInterfacedObject。
声明了一个实现类后，要实现接口中的方法，您必须把接口中的方法显式或隐式地映射成实现类的相容的方法，然后再在这个实现类中实现这些方法。

2.8 调度接口

后面我们将介绍的OLE自动化对象，实际上就是实现调度接口的对象。调

度接口的成员可以是特性，也可以是方法。与一般的接口相比，您不能用一个类去实现调度接口，换句话说，在声明实现类时，不能把调度接口列在要实现的接口列表中，因为调度接口中的方法仅仅是充当了IDispatch接口中的Invoke函数的原型，调度接口并不需要实现自己。
IDispatch接口中的Invoke函数是加在成员身上的外套，它的第一个参数就是真正要访问的成员的识别号，那么怎样传递成员的参数和返回值(如果有的话)呢？不用担心，Invoke函数具有这样的机制，它的参数是不固定的，因为成员的参数是不固定的。
使用调度接口最大的优势在于它内建对Marshaling的支持，从而使跨进程边界和跨机器边界的通讯成为可能。更重要的是，Marshaling是自动的，也就是说，不管是OLE自动化服务器端还是客户端，程序员可以不管Invoke究竟是怎样实现的细节。
对于OLE自动化对象来说，并不需要事先知道它的类型库，因为IDispatch接口的GetIDsOfNames函数能够在运行期把名称转换为识别号。同样，在编译期，编译器也不进行类型匹配检查，这样有可能导致调用失败。要说明的是，凡是用C++ Builder 3的对象库中的向导创建的OLE自动化对象总是有类型库，以避免过分的盲目性。
如果OLE自动化对象有类型库的话，也就是说，在编译期成员的识别号就是确定的，这时候您可以直接在程序中读取或指定识别号，这就是所谓的“ID Binding”。由于这时候只需要调用Invoke函数，不需要调用GetIDsOfNames函数，因此，程序的速度加快。使用“ID Binding”的另一个好处是，编译器可以进行类型匹配检查，凡是有错误的代码在编译期就

被报告出来，而不是在运行期才发现。使用“ID Binding”这种方式也有一个缺点，直接用识别号来调用成员不太直观，而且无法检查识别号和名称是否对应。

2.9 双 重 接 口

在OLE自动化操作中，有的自动化客户(或称控制程序)希望通过虚拟方法表来访问自动化对象(或称服务程序)，而有的自动化客户则希望只在运行期通过IDispatch接口来访问自动化对象，例如，有些解释型的宏编程语言和脚本工具，它们只在运行期才是有意义的。为了同时支持上述两种访问方式，C++ Builder 3引入了双重接口的概念。

所谓双重接口(Dual Interface)，是指一种能同时支持编译期虚拟方法表和运行期动态装订的接口，双重接口同时以虚拟方法表和IDispatch接口两种方式暴露它的方法。

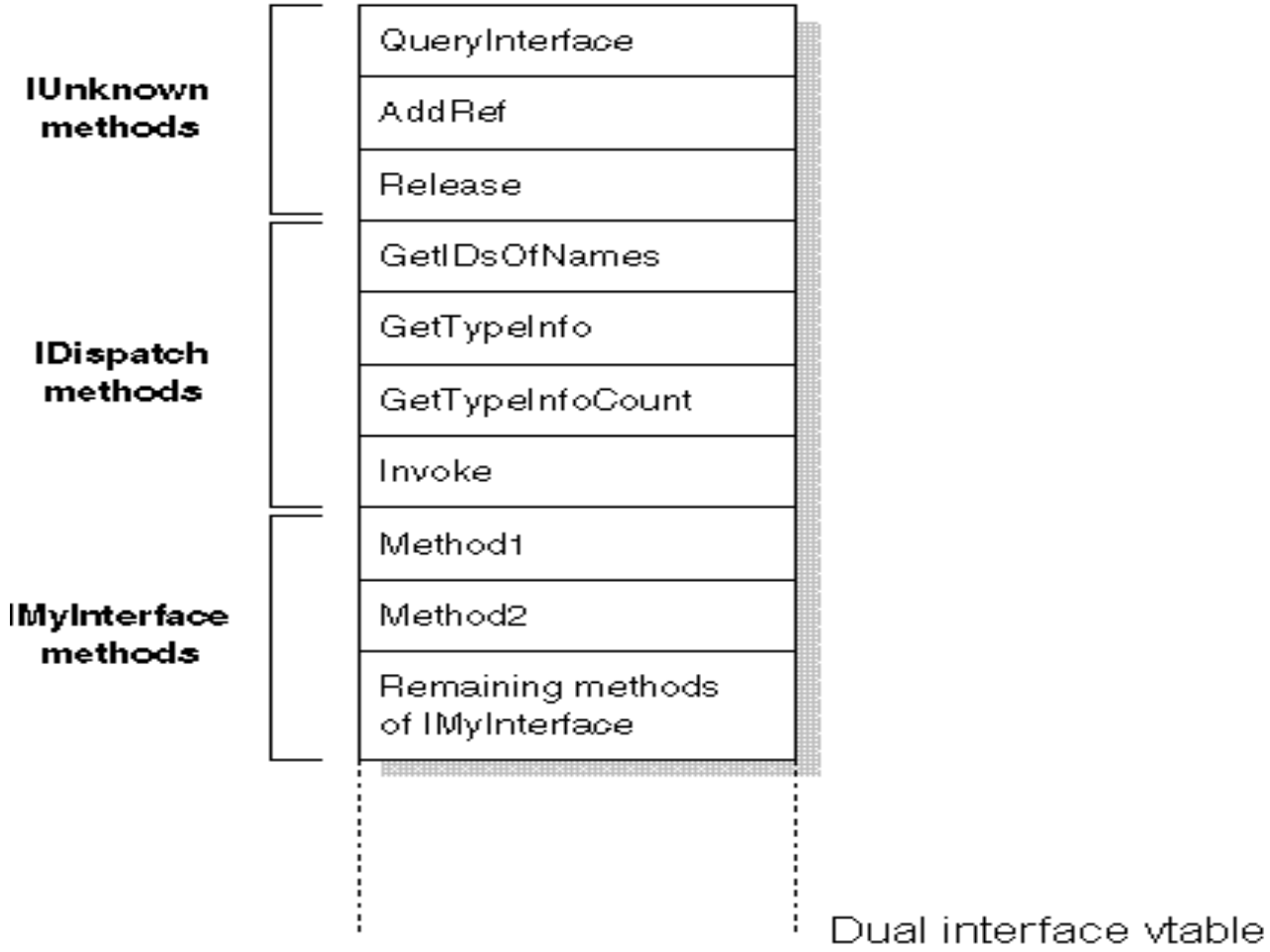
在写客户程序的时候并不知道自动化对象有哪些成员，也就是说无法获得自动化对象的类型信息，这时候就要通过IDispatch接口间接地访问自动化对象。

虚拟方法表实际上是一块内存区域，这块区域是由编译器建立的，存放了自动化对象的成员的地址，头三个是IUnknown接口中的虚拟方法即_AddRef、_Release和QueryInterface，如果自动化对象支持IDispatch接口的话，接下来就是IDispatch接口中的四个方法，然后才是自动化对象本身的方法。我们用一个图来表示虚拟方法表的结构。

双重接口综合了虚拟方法表和IDispatch接口的优势，这样，OLE客户程序就可以用编译型的语言如C++、Object Pascal等来写，因为在写程序的时候就知道要调用自动化对象的哪些特性和方法，而且访问自动化对象是直接的，速度较快。采用虚拟方法表这种方式还有个好处是，可以让编译器进行类型检查。

不过，使用双重接口也有若干个限制，因为双重接口必须继承于IDispatch接口，方法的参数类型和返回类型必须是可自动操作的，这就意味着，C++语言中许多非常有用的数据类型派不上用场。如果需要返回某些信息，只能通过方法的最后一个参数。

总的来说，双重接口比单独用IDispatch接口或单独用虚拟方法表要具有一些优势，因此，用C++ Builder 3创建 OLE自动化对象时，默认总是生成双重接口。



2.10 对接口的引用

接口可以通过接口的变量来引用，由于接口是一种特殊的“数据类型”，因此，对接口的引用实际上是对实现类的引用。对于外部代码来说，通过

接口引用就可以调用接口中的方法，而不需要知道接口究竟是怎样实现的，也就是说，实现类实际上是隐藏在背后的东西。这正是面向对象编程思想的体现。要说明的是，接口的实现类中除了映射了接口的方法以外，还可以有自己的数据和方法，不过，通过对接口的引用并不能访问这些“内部”的数据和方法，这些数据和方法是实现类内部使用的。

System单元中的TInterfacedObject用于实现IUnknown接口，它的实例可以动态赋值给任何接口的引用。细心的读者可能发现，接口的实现类一般都以TInterfacedObject为祖先类，从语法上讲，您也可以把其它VCL类作为祖先类，不过，如果所选的祖先类并不是用于实现接口的，您就不能把类的实例赋值给某接口的引用，否则编译器会报错。

用实现IUnknown接口的类如TInterfacedObject作为祖先类还有一个好处是，IUnknown接口中的QueryInterface方法具有版本自升级功能，当一个组件对象升级的时候，通过调用QueryInterface可以查询到组件对象新增加的接口或原有接口的新版本，而对于已有的客户代码来说，完全可以继续保留，不需要重新编译。由于接口本身并不实现自己，真正实现接口的是接口对象，而接口对象对客户代码来说是隐藏的，这样，接口的开发者就可以单独给接口对象升级，而不会影响到客户代码。

第三章　ActiveX 框架

Borland®公司在面向对象领域一直处于世界领先地位，ActiveX 框架就是他的又一杰作。ActiveX框架中封装了许多COM和ActiveX的接口和函数，使开发基于COM和ActiveX的应用变得非常简单。不仅如此，ActiveX 框架还允许过去基于TWinControl的控件转换成为ActiveX控件，从而使大量的基于VCL的控件可以应用到其它开发环境，如Microsoft的Visual Basic 4.0和5.0、Visual C++ 4.x、Internet Explorer 3.0和4.0、ActiveX Control Pad、FrontPage 98以及Borland的Delphi 3、IntraBuilder、Paradox 8等。

3.1　什么是 ActiveX 框架

ActiveX框架实际上是一组有继承关系的类的统称，这些类分别用于实现简单的COM对象、带类型信息描述的COM对象、OLE自动化服务器、ActiveX控件、ActiveForm以及特性页等。这些类及其继承关系如下：

```
TComObject                          //简单的COM对象
TTypedComObject                     //带类型信息描述的COM对象
    TAutoObject                     //OLE自动化服务器的基类
        TActiveXControl             //任何一个ActiveX控件的直接上级
```

```
TComServerObject                    //TComServer的虚拟基类
TComServer                  //COM服务器的基类

TCustomForm                         //TForm、TActiveForm的共同基类
TActiveForm                 //复合的ActiveX控件

TCustomForm                         //特性页实际上也是Form
TPropertyPage               //特性页的基类

TComObjectFactory                   //用于注册简单的COM对象
TActiveXPropertyPageFactory         //用于注册ActiveX控件的特性页
TTypedComObjectFactory              //用于注册带类型信息描述的COM对象
    TAutoObjcetFactory          //用于注册OLE自动化服务器
        TActiveXControlFactory      //用于注册ActiveX控件
            TactiveFormFactory  //用于注册ActiveForm
```

3.2 TInterfacedObject

TInterfacedObject是这样声明的：

```
class DELPHICLASS TInterfacedObject : TObject
```

```
{
private:
      void *IUnknown;
      int   FRefCount;
protected:
      virtual HResult _stdcall QueryInterface(const GUID& IID, void *Obj);
      virtual int _stdcall _AddRef(void);
      virtual int _stdcall _Release(void);
public:
      _property int RefCount;
  };
```

可以看出，TInterfacedObject是直接从TObject继承下来的，用于实现IUnknown接口。我们知道，IUnknown接口是C++ Builder 3中所有接口的默认祖先接口，因此，TInterfacedObject往往用于作为接口对象的基类。

TInterfacedObject没有也不需要类工厂，因此，那些从TInterfacedObject继承下来的接口对象应当调用自己的构造函数来创建接口对象的实例。

根据类型相容的规则，从TInterfacedObject继承下来的派生类的实例可以赋值给TInterfacedObject的变量，但赋值并不总是成功的。

3.3 TComObject

TComObject是C++ Builder 3中非常重要的一个类，通常用于作为简单的COM对象的基类。TComObject同时支持IUnknown接口和ISupportErrorInfo接口。TComObject在ComObj单元中是这样声明的：

```
class PASCALIMPLEMENTATION TComObject : public System::TObject
{
typedef System::TObject inherited;
private:
    void *_IUnknown;
    void *_ISupportErrorInfo;
public:
    _fastcall TComObject(void);
    _fastcall TComObject(const _di_IUnknown Controller);
    _fastcall TComObject(TComObjectFactory* Factory, const _di_IUnknown Controller);
    _fastcall virtual ~TComObject(void);
    virtual void _fastcall Initialize(void);
    virtual int _stdcall ObjAddRef(void);
    virtual HRESULT _stdcall ObjQueryInterface(const GUID &IID, void *Obj);
    virtual int _stdcall ObjRelease(void);
    virtual HRESULT _fastcall SafeCallException(System::TObject* ExceptObject, void *
    ExceptAddr);
```

```
    _property _di_IUnknown Controller;
    _property TComObjectFactory* Factory;
    _property int RefCount = {read=FRefCount, nodefault};
};
```

TComObject有三个构造函数，这意味着它可以有三种创建方式：

- 没有参数的构造函数，用于把它实例化成一个简单的COM对象。
- 带1个参数的构造函数，用于把它实例化成聚合的一部分。
- 带2个参数的构造函数，用于把它实例化成一个简单的COM对象，在创建实例的同时还对实例进行初始化。

TComObject同时支持IUnknown接口和ISupportErrorInfo接口，这就是所谓的“接口聚合”。这样，一个OLE自动化客户程序就可以查询到错误信息，并且保证错误信息能沿着调用链逐级传播。通过实现ISupportErrorInfo接口中的InterfaceSupportsErrorInfo方法，TComObject支持IErrorInfo策略，因而也支持OLE异常处理和SafeCall调用约定。

Controller 特性

声明：_property _di_IUnknown Controller;
这个特性用于在一个聚合的对象内返回IUnknown接口，如果返回NULL表示对象不是聚合的一部分。

Factory 特性

声明：_property TComObjectFactory* Factory;

这个特性返回COM对象的类工厂，类工厂用于创建TComObject的实例。

RefCount

声明：_property int RefCount;
这个特性返回COM对象的引用计数，它是由IUnknown接口的AddRef和Release自动维护的。

Initialize 函数

声明：virtual void _fastcall Initialize(void);
COM对象可以重载这个虚拟方法，对实例进行初始化。注意：调用构造函数时已经对实例进行了默认的初始化，因此，Initialize不要重复这些初始化的内容。

ObjAddRef 函数

声明：virtual int _stdcall ObjAddRef(void);
这个函数是从IUnknown接口中的_AddRef映射过来的，用于重新实现_AddRef。

ObjRelease 函数

声明：virtual int _stdcall ObjRelease(void);
这个函数是从IUnknown接口中的_Release映射过来的，用于重新实现_Release。

ObjQueryInterface 函数

声明：virtual HRESULT _stdcall ObjQueryInterface(const GUID &IID, void *Obj);
这个函数是从IUnknown接口中的QueryInterface映射过来的，用于重新实现QueryInterface。QueryInterface函数用于判断COM对象是否支持IID参数指定的接口。如果COM对象不是聚合的一部分，就用ObjQueryInterface来实现IUnknown的QueryInter-face。

3.4 TTypedComObject

TTypedComObject是从TComObject继承下来的，支持IProvideClassInfo接口。在ComObj单元中，TTypedComObject是这样声明的：

```
class PASCALIMPLEMENTATION TTypedComObject : public Comobj::TComObject
{
typedef Comobj::TComObject inherited;
private:
    void *_IProvideClassInfo;
public:
    operator IProvideClassInfo*(void)
    { return (IProvideClassInfo*)&_IProvideClassInfo; }
public:
```

```
    _fastcall TTypedComObject(void) : Comobj::TComObject( ) { }
    _fastcall TTypedComObject(const _di_IUnknown Controller) : Comobj::
    TComObject(Controller) { }
    _fastcall TTypedComObject(TComObjectFactory* Factory, const _di_IUnknown
    Controller) : Comobj::TComObject(Factory, Controller) { }
    _fastcall virtual ~TTypedComObject(void) { }
};
```

TTypedComObject最大的特点是支持IProvideClassInfo接口，可以直接读取类型信息，而不需要事先调入类型库。正是由于这个特点，凡是以TTypedComObject为基类的COM对象必须在类型库中提供类型信息描述。

3.5 TAutoObject

TAutoObject用于作为OLE自动化服务器的基类，同时，由于ActiveX本质上也是基于OLE自动化，因此，TAutoObject也是TActiveXControl的祖先类。在ComObj单元中，TAutoObject是这样声明的：

```
class PASCALIMPLEMENTATION TAutoObject : public Comobj::TTypedComObject
{
typedef Comobj::TTypedComObject inherited;
private:
    void *_IDispatch;
```

```
public:
    _fastcall TAutoObject(void) : Comobj::TTypedComObject( ) { }
    _fastcall TAutoObject(const _di_IUnknown Controller) : Comobj::
    TTypedComObject(Controller) { }
    _fastcall TAutoObject(TComObjectFactory* Factory, const _di_IUnknown Controller) :
    Comobj::TTypedComObject(Factory, Controller) { }
    _fastcall virtual ~TAutoObject(void) { }
};
```

可以看出，TAutoObject是从TTypedComObject继承下来的，支持IDispatch接口。TAutoObject还提供了自动的Marshaling功能，凡是用TAutoObject作为基类所建立的COM对象可以跨进程边界传递函数调用。如果不用TAutoObject作基类的话，COM对象就要自己实现Marshaling功能，先要打包，到达目的地后再解包。

TAutoObject的对象需要建立类型库，因为它支持双重接口。同时，由于TAutoObject间接地继承了IProvideClassInfo接口，也需要类型库。TAutoObject有相应的类工厂TAutoObjectFactory，如果要创建的OLE自动化对象是内部使用的，就不需要类工厂，此时应当选TAutoIntfObject为基类。

3.6 TAutoIntfObject

有的OLE自动化对象仅仅是内部使用的，这种自动化对象不在Windows的注

册表中注册，也就是说，它不需要类工厂，这时候，自动化对象应当以TAutoIntfObject为基类。
TAutoIntfObject是从TInterfacedObject继承下来的，支持IDispatch接口和IsupportErr-orInfo接口，在ComObj单元中，TAutoIntfObject是这样声明的：

```
class PASCALIMPLEMENTATION TAutoIntfObject : public System::TInterfacedObject
{
typedef System::TInterfacedObject inherited;
private:
	void *_IDispatch;
	void *_ISupportErrorInfo;
public:
	_fastcall TAutoIntfObject(const _di_ITypeLib TypeLib, const GUID &DispIntf);
	virtual HRESULT _fastcall SafeCallException(System::TObject* ExceptObject, void *
	ExceptAddr);
	_property System::PInterfaceEntry DispIntfEntry ;
	_property _di_ITypeInfo DispTypeInfo ;
	_property GUID DispIID ;
public:
	_fastcall virtual ~TAutoIntfObject(void) { }
};
```

可以看出，TAutoIntfObject与TAutoObject具有相似性，它们都直接或间接地支持IDispatch接口和ISupportErrorInfo接口，因而都支持OLE异常处理。

不同的是，TAutoIntfObject没有类工厂，要创建TAutoIntfObject的实例，必须调用它的构造函数。

3.7 TActiveXControl

TActiveXControl作为ActiveX控件的抽象基类，它是从TAutoObject继承下来的，而TAutoObject又是从TComObject继承下来的，因此，ActiveX控件本质上也是COM对象，支持OLE自动化操作，具有访问类型信息的能力。所有的ActiveX控件必须建立类型库。此外，TActiveXControl还提供了把基于VCL的控件转换为ActiveX控件的能力。

在AxCtrls单元中，TActiveXControl是这样声明的：

```
class PASCALIMPLEMENTATION TActiveXControl : public Comobj::TAutoObject
{
typedef Comobj::TAutoObject inherited;
private:
    void *_IPersistPropertyBag;
    void *_IPersistStreamInit;
    void *_IPersistStorage;
    void *_IObjectSafety;
    void *_IOleObject;
    void *_IOleControl;
    void *_IOleInPlaceObject;
```

```
    void *_IOleInPlaceActiveObject;
    void *_IViewObject;
    void *_IViewObject2;
    void *_IPerPropertyBrowsing;
    void *_ISpecifyPropertyPages;
    void *_ISimpleFrameSite;
public:
    _fastcall virtual ~TActiveXControl(void);
    virtual void _fastcall Initialize(void);
    virtual HRESULT _stdcall ObjQueryInterface(const GUID &IID, void *Obj);
    bool _fastcall PropRequestEdit(const System::WideString PropertyName);
    void _fastcall PropChanged(const System::WideString PropertyName);
    _property Controls::TWinControl* Control;
public:
    _fastcall TActiveXControl(void) : Comobj::TAutoObject( ) { }
        _fastcall TActiveXControl(const _di_IUnknown Controller) : Comobj::
    TAutoObject(Controller) { }
    _fastcall TActiveXControl(Comobj::TComObjectFactory* Factory, const
    _di_IUnknown Controller) : Comobj::TAutoObject(Factory, Controller) { }
};
```

TActiveXControl定义了ActiveX控件的核心行为和需要的各种接口，用TActiveXControl为基类的ActiveX控件可以嵌入到任何一种ActiveX容器中，

包括Internet Explorer、Visual Basic、PowerBuilder、Paradox、Borland C++、IntraBuilder以及C++ Builder 3自身。
在C++ Builder 3中，要创建一个ActiveX控件很简单，对象库中提供了创建ActiveX控件的向导，您只要指定一个已有的VCL控件，向导就会把它转换为ActiveX控件。
要注意的是，TActiveXControl本身是个抽象类，不能直接创建它的实例，您只能以TActiveXControl为基类派生出一个类，然后再创建该派生类的实例即ActiveX控件。不过，真正创建ActiveX控件的实际上是类工厂。

Control特性

声明：_property Controls::TWinControl* Control;
这个只读的特性表明ActiveX控件是从哪个VCL控件转换而来的。

DefinePropertyPages函数

声明：virtual void _fastcall DefinePropertyPages(TDefinePropertyPage &DefineProperty- Page);
当ActiveX的特性页将要显示之前会调用这个函数。因此，如果要使ActiveX控件能显示特性页，您必须重载这个函数。这个函数的名称取得不太好，它唯一的参数是DefinePropertyPage，与函数名仅差一个字母。DefinePropertyPage参数本身是个回调函数，这个回调函数也只有一个参数，就是要显示的特性页对象的类标识符。如果ActiveX控件的特性比较多，特性页通常做成多页的对话框，您应当事先把特性归类，放在不同的页上。

GetPropertyString 函数

声明：virtual bool _fastcall GetPropertyString(int DispID, System::AnsiString &S);
当Object Inspector试图用一个字符串来表达某个特性当前的值时会调用这个函数。如果ActiveX控件有多个特性，就会调用此函数多次。
DispID参数是特性的识别号，如果需要让Object Inspector以一个特殊的字符串显示特性的值，就让S参数返回这个字符串，并让函数本身返回True。如果不需要Object Inspector以一个特殊的字符串显示特性的值，就无须对S参数赋值，并返回False，此时Object Inspector将以标准的方式显示特性的值。

GetPropertyStrings 函数

声明：virtual bool _fastcall GetPropertyStrings(int DispID, Classes::TStrings* Strings);
当用户在Object Inspector中试图打开一个特性的值的列表时会调用这个函数。
DispID参数是特性的识别号，如果需要显示一个下拉列表，就对Strings参数赋值，并让函数返回True。要注意的是，列表中的每一个值应当是唯一的。如果不需要显示下拉列表，就不必对Strings参数赋值，并让函数返回False。

GetPropertyValue 函数

声明：virtual void _fastcall GetPropertyValue(int DispID, int Cookie, OleVariant &Value);

如果您重载了GetPropertyStrings函数，就要配套重载GetPropertyValue，当用户从下拉列表中选择了一个字符串，要把用户选择的字符串转换为数值，并通过Value参数返回。DispID参数是特性的识别号，Cookie参数是所选的项在列表中的序号。

InitializeControl函数

声明：virtual void _fastcall InitializeControl(void);
当ActiveX控件刚刚创建好后就会调用此函数，这样您就有机会对ActiveX控件初始化，一般是把事件句柄赋给ActiveX控件的事件特性，程序示例如下：

```
void InitializeControl( )
{
m_VclCtl->OnClick = ClickEvent;
}
```

LoadFromStream 函数

声明：virtual void _fastcall LoadFromStream(const _di_IStream Stream);
C++ Builder 3已默认实现了这个方法，用于从流中读取ActiveX控件的状态。您可以重载这个方法，读取任何通过SaveToStream函数保存在流中的数据。Stream参数是要读写的流，要从流中读取数据，调用Stream->Read( )。注意：您读取的数据必须与用SaveToStream写入的数据匹配。

SaveToStream 函数

声明：virtual void _fastcall SaveToStream(const _di_IStream Stream);
C++ Builder 3已默认实现了这个方法，用于把ActiveX控件的状态写到流中。您可以重载这个方法，把其它任何要保存的数据保存到流中。
Stream参数是要读写的流，要把数据保存到流中，调用Stream->Write( )。

PerformVerb 函数

声明：virtual void _fastcall PerformVerb(int Verb);
如果您调用了TActiveXControlFactory的AddVerb函数给ActiveX控件增加了一个动作，您应当重载这个方法具体实现这个动作，当用户在ActiveX控件上单击鼠标右键，在弹出的菜单中选择该动作项时，就会调用这个方法，其中，Verb参数是动作项的编号。

Initialize 函数

声明：virtual void _fastcall Initialize(void);
这个函数有点类似于构造函数，用于创建类工厂对象，并且把此对象与TActiveXControl联系起来，以便操纵ActiveX控件。

ObjQueryInterface 函数

声明：virtual HRESULT _stdcall ObjQueryInterface(const GUID &IID, void *Obj);
这个函数实际上是映射了IUnknown接口中的QueryInterface函数，用于返回

IID参数指定的接口。

PropChanged 函数

声明：void _fastcall PropChanged(const System::WideString PropertyName);
调用此函数将触发OnChanged事件，表示PropertyName参数指定的特性的值已改变。

PropRequestEdit 函数

声明：bool _fastcall PropRequestEdit(const System::WideString PropertyName);
这个函数返回PropertyName参数指定的特性是否可以编辑，返回True表示可以编辑。

3.8 TComServerObject

TComServerObject是从TObject直接继承下来的，作为TComServer的基类。在ComObj单元中，TComServerObject是这样声明的：

```
class PASCALIMPLEMENTATION TComServerObject : public System::TObject
{
public:
    _property System::AnsiString HelpFileName;
    _property System::AnsiString ServerFileName;
    _property System::AnsiString ServerKey;
```

```
    _property System::AnsiString ServerName;
        _property _di_ITypeLib TypeLib;
    _fastcall TComServerObject(void) : System::TObject( ) { }
    _fastcall virtual ~TComServerObject(void) { }
};
```

可以看出，TComServerObject所有的方法都是虚拟的和抽象的，这意味着，您不能直接创建它的实例。TComServerObject的5个公共的特性都是只读的，真正的含义取决于TComServer怎样实现上述几个虚拟和抽象的方法。

3.9 TComServer

TComServer是COM服务器的基类，COM服务器可以理解为组件对象的容器，从形式上讲，COM服务器可以是In-Process型即DLL，也可以是Local型即EXE。

ComServ单元中声明了一个ComServer变量，并自动创建了TComServer的实例，就象Forms单元中的Application变量一样，您用不着再专门创建TComServer的实例。

在ComServ单元中，TComServer是这样声明的(按Object Pascal的语法)：

```
TComServer = Class(TComServerObject)
Private
    {私有的成员此处省略}
Protected
```

```
    {重载TComServerObject中的虚拟方法}
    Function CountObject(Created: Boolean): Integer; override;
    Function CountFactory(Created: Boolean): Integer; override;
    Function GetHelpFileName: string; override;
    Function GetServerFileName: string; override;
    Function GetServerKey: string; override;
    Function GetServerName: string; override;
    Function GetTypeLib: ITypeLib; override;
Public
    Constructor Create;
    Destructor Destroy; override;
    Procedure Initialize;
    Procedure LoadTypeLib;
    Procedure SetServerName(const Name: string);
    Procedure UpdateRegistry(Register: Boolean);
    Property IsInprocServer: Boolean read FIsInprocServer write FIsInprocServer;
    Property ObjectCount: Integer read FObjectCount;
    Property StartMode: TStartMode read FStartMode;
    Property OnLastRelease: TLastReleaseEvent read FOnLastRelease write
        FOnLastRelease;
End;
```

另外，ComServ单元中还声明了几个全局例程，用于自动管理COM服务器的注册、撤消注册以及卸载。如果COM服务器是In-Process型，必须输出这几个例程：DllGetClassObject、DllCanUnloadNow、DllRegisterServer、

DllUnregisterServer。
ComServer变量用TComClassManager管理类工厂，TComClassManager是在ComObj单元中声明的，这个单元已经自动声明了TComClassManager的实例ComClassManager变量。

HelpFileName特性

声明：_property System::AnsiString HelpFileName;
这个特性用于返回与COM服务器模块关联的帮助文件名。

IsInprocServer特性

声明：_property bool IsInprocServer;
如果COM服务器是In-Process型的服务器，这个特性就返回True。In-Process型的COM服务器一般是DLL，而Local型的服务器一般是EXE。

ObjectCount特性

声明：_property int ObjectCount;
这个特性返回有多少个对象当前依赖于此COM服务器。如果这个特性返回零，表示当前没有哪个对象在使用COM服务器，此时，COM服务器就可以卸载了。当COM服务器卸载时，将触发OnLastRelease事件。可见，ObjectCount类似于引用计数。

ServerFileName 特性

声明：_property System::AnsiString ServerFileName;
这个特性返回COM服务器模块的文件名，包括完整的路径。

ServerKey 特性

声明：_property System::AnsiString ServerKey;
如果这个特性返回“InprocServer32”，表示COM服务器是In-Process型服务器，如果这个特性返回“LocalServer32”，表示COM服务器是Local型服务器。

StartMode 特性

声明：_property TStartMode StartMode;
这个特性用于返回COM服务器被启动的原因，可以是以下值(含命令行开关)：

返回值	命令行开关	含义
smAutomation	embedding	由Windows响应COM客户的请求而启动

续表

smRegServer	regserver	仅仅是为了注册而启动
smStandalone	---	用户把COM服务器作为独立的程序启动
smUnregServer	unregserver	仅仅是为了撤消注册而启动

ServerName特性

声明：_property System::AnsiString ServerName;
这个特性返回COM服务器的名称。

TypeLib特性

声明：_property ITypeLib TypeLib;
这个特性返回COM服务器的类型库。

LoadTypeLib函数

声明：void _fastcall LoadTypeLib(void);
这个函数把COM服务器的类型库调入，如果类型库还没有准备好，就注册类型库。如果LoadTypeLib调用失败，将触发EOleSysError异常，此时，您可以访问有关错误码：

E_OUTOFMEMORY	内存不够
E_INVALIDARG	部分参数是非法的
TYPE_E_IOERROR	无法写文件
TYPE_E_INVALIDSTATE	类型库还没有打开
TYPE_E_INVDATAREAD	无法读文件
TYPE_E_UNSUPFORMAT	类型库的格式是老的
TYPE_E_UNKNOWNLCID	在OLE支持DLLs中没有找到LCID
TYPE_E_CANTLOADLIBRARY	类型库或DLL无法调入

SetServerName 函数

声明：virtual void _fastcall SetServerName(const System::AnsiString Name);
这个函数用于设置COM服务器的名称。不过，如果COM服务器有类型库的话，这个函数就什么也不干，因为服务器的名称与类型库的名称应当一致。

UpdateRegistry 函数

声明：virtual void _fastcall UpdateRegistry(bool Register);
这个函数用COM服务器最新的状态更新注册表，如果Register参数设为True表示注册，如果Register参数设为False，表示撤消注册。

OnLastRelease事件

声明：_property TLastReleaseEvent OnLastRelease;
其中，TLastReleaseEvent是这样声明的：

```
typedef void _fastcall (_closure * TLastReleaseEvent)(TObject* Sender,bool & Shutdown);
```

当使用COM服务器的最后一个对象被删除后，也就是ObjectCount特性减到零后，COM服务器就可以卸载了，此时将触发OnLastRelease事件。在处理此事件的句柄中，如果把ShutDown参数设为True，表示真的要卸载COM服务器。如果把Shutdown参数设为False，表示不卸载COM服务器。

3.10 TActiveForm

TActiveForm用于作为ActiveForm的基类，关于ActiveForm的概念后面详细阐述，这里您可以简单地把它看作做是一个复合的ActiveX控件，可以嵌入到Web页面中传递。从本质上讲，ActiveForm也是Form，只不过，ActiveForm上的控件都是内部使用的，对于客户程序来说，ActiveForm是一个整体。
C++ Builder 3在对象库中提供了创建ActiveForm的向导，创建好后，您可以使用“Project”菜单上的“Web Deploy”命令发布ActiveForm。
在AxCtrls单元中，TActiveForm是这样声明的(按Object Pascal的语法声明)：

```
TActiveForm = Class(TCustomForm)
Private
```

```
    FAxBorderStyle: TActiveFormBorderStyle;
    Procedure SetAxBorderStyle(Value: TActiveFormBorderStyle);
Protected
    Procedure DefinePropertyPages(DefinePropertyPage:TDefinePropertyPage);virtual;
    Procedure CreateParams(var Params: TCreateParams); override;
    Procedure EventSinkChanged(const EventSink: IUnknown); virtual;
    Procedure Initialize; virtual;
Public
    Constructor Create(AOwner: TComponent); override;
    Function WantChildKey(Child:TControl;var Message:TMessage):Boolean;override;
Published
    Property AxBorderStyle: TActiveFormBorderStyle read FAxBorderStyle
    {以下特性和事件都是从TCustomForm继承的}
    …
End;
```

除了从TCustomForm继承下来的特性和事件外，TActiveForm中唯一公开的特性是AxBorderStyle，用于设置ActiveForm的边框样式，可以设为以下值：afbNone(没有边框)、afbSingle(普通的单细线边框)、afbSunken(凹进的3D边框)、afbRaised(突起的3D边框)。

TCustomForm中有几个可重载的方法，其中，DefinePropertyPages、EventSinkChanged和Initialize前面已经讲过，CreateParams一般不需要重载。

TCustomForm中有一个WantChildKey函数，它是这样声明的：

virtual bool _fastcall WantChildKey(TControl Child, TMessage &Message)
当Form上的控件接收到键盘输入时会调用这个函数，如果让这个函数返回True，该控件不对消息进行任何处理，只是把消息传递给Form。

3.11 TPropertyPage

TPropertyPage用于作为ActiveX控件和ActiveForm的特性页。TPropertyPage也是从TCustomForm继承下来的，因此，特性页本质上也是Form。与普通的Form不同的是，特性页通常是模式Form，而且能够读取ActiveX控件或ActiveForm的特性值到特性页上，或者把用户在特性页上的修改写回到ActiveX控件或ActiveForm中。
TPropertyPage是个虚拟基类，您不能直接创建它的实例，您只能以TPropertyPage为基类，在派生类中重载一些方法，然后再创建派生类的实例。
在AxCtrls单元中，TPropertyPage是这样声明的：

```
class PASCALIMPLEMENTATION TPropertyPage : public Forms::TCustomForm
{
typedef Forms::TCustomForm inherited;
private:
    TPropertyPageImpl* FActiveXPropertyPage;
    System::OleVariant FOleObject;
    Classes::TInterfaceList* FOleObjects;
```

```
    HIDESBASE MESSAGE void _fastcall CMChanged(TCMChanged &Msg);
public:
    _fastcall virtual TPropertyPage(Classes::TComponent* AOwner);
    _fastcall virtual ~TPropertyPage(void);
    void _fastcall Modified(void);
    virtual void _fastcall UpdateObject(void);
    virtual void _fastcall UpdatePropertyPage(void);
    _property System::OleVariant OleObject = {read=FOleObject};
    _property Classes::TInterfaceList* OleObjects;
    void _fastcall EnumCtlProps(const GUID &PropType,TStrings* PropNames);
_published:
    _property ActiveControl ;
    //以下特性和事件都是从TCustomForm继承的
    ...
public:
    _fastcall TPropertyPage(Classes::TComponent* AOwner, int Dummy) : Forms::
    TCustomForm(AOwner, Dummy) { }
    _fastcall TPropertyPage(HWND ParentWindow) : Forms::
    TCustomForm(ParentWindow) { }
};
```

创建特性页需要重载UpdateObject和UpdatePropertyPage两个函数，前者用于把用户在特性页中的修改写回到ActiveX控件或ActiveForm中，后者用于读

取ActiveX控件或ActiveForm的特性值到特性页上。

OleObject特性

声明：_property System::OleVariant OleObject;
这个只读的特性返回特性页正在编辑的ActiveX控件或ActiveForm，这样，您就可以访问ActiveX控件或ActiveForm的特性或方法。

3.12 TComObjectFactory

在ComObj单元中，TComObjectFactory是这样声明的：

```
class PASCALIMPLEMENTATION TComObjectFactory : public System::TObject
{
typedef System::TObject inherited;
private:
    void *_IUnknown;
    void *_IClassFactory;
    void *_IClassFactory2;
public:
    _fastcall TComObjectFactory(TComServerObject* ComServer, System::
    TMetaClass* ComClass, const GUID &ClassID, const System::AnsiString
    ClassName, const System::AnsiString Description, TClassInstancing Instancing);
    _fastcall virtual ~TComObjectFactory(void);
```

```
    TComObject* _fastcall CreateComObject(const _di_IUnknown Controller);
    void _fastcall RegisterClassObject(void);
    virtual void _fastcall UpdateRegistry(bool Register);
    _property GUID ClassID ;
    _property System::AnsiString ClassName ;
    _property System::TMetaClass* ComClass ;
    _property TComServerObject* ComServer ;
    _property System::AnsiString Description ;
    _property GUID ErrorIID ;
    _property System::WideString LicString ;
    _property System::AnsiString ProgID ;
    _property TClassInstancing Instancing ;
    _property bool ShowErrors;
    _property bool SupportsLicensing ;
};
```

TComObjectFactory就是前面一再提到的类工厂。从上述声明可以看出，TcomObject-Factory同时支持IUknown、IClassFactory和IClassFactory2接口。类工厂的作用是创建和输出COM对象的实例，并在Windows的注册表中注册COM对象。类工厂本身的实例一般应当在项目文件中建立，也可以由Windows的CoCreateClassObject函数建立，那么到底该怎样选择呢？用构造函数适合于只建立类工厂的一个实例，而CoCreateClassObject函数适合于建立多个实例。

COM服务器所拥有的类工厂由TComClassManager来管理，在ComObj单元中已自动创建了TComClassManager的实例叫ComClassManager。

ClassID 特性

声明：_property GUID ClassID;
这个只读的特性返回类工厂所要创建实例的COM对象的类标识符CLSID。在调用TComObjectFactory的构造函数时，必须指定COM对象的类标识符(ClassID参数)。

ClassName 特性

声明：_property System::AnsiString ClassName;
这个只读的特性返回类工厂所要创建实例的COM对象的类名。在调用TComObjectFactory的构造函数时，必须指定COM对象的类名(ClassName参数)。

ComClass 特性

声明：_property System::TMetaClass* ComClass;
这个只读的特性返回类工厂所要创建实例的COM对象的类型，TComClassManager 的GetFactoryFromClass函数要用到COM对象的类型。在调用TComObjectFactory的构造函数时，必须指定COM对象的类型(ComClass参数)。

ComServer 特性

声明：property ComServer: TComServerObject;
这个只读的特性返回拥有这个类工厂的COM服务器。在调用TComObjectFactory的构造函数时，必须指定COM服务器(ComServer参数)。

Description 特性

声明：_property System::AnsiString Description;
这个只读的特性从COM对象的类型库中返回关于此对象的描述。当类工厂在注册表中注册COM对象时，需要指定一个字符串来描述要注册的COM对象。在调用TComObjectFactory的构造函数时，必须指定此字符串(Description参数)。

Instancing 特性

声明：_property TClassInstancing Instancing;
这个只读的特性返回COM对象被创建的方式，返回ciInternal表示COM对象是COM服务器内部使用的，返回ciSingleInstance表示在同一个时刻COM对象只有一个实例，返回ciMultiInstance表示在同一个COM对象可以有多个实例。

ProgID 特性

声明：_property System::AnsiString ProgID;
这个只读的特性返回类工厂的ProgID，凡是可以插入到OLE容器的对象都

有其ProgID。

SupportsLicensing特性

声明：_property bool SupportsLicensing;
如果这个特性设为True，类工厂要创建的COM对象的实例将支持“许可”检查功能。

CreateComObject函数

声明：virtual TComObject* _fastcall CreateComObject(const _di_IUnknown Controller);
这个函数用于创建COM对象的实例，它实际上是调用TComObject的一个构造函数，并把类工厂自己传递给CreateFromFactory的Factory参数。

RegisterClassObject函数

声明：void _fastcall RegisterClassObject(void);
这个函数用于注册类工厂所创建的COM对象。如果没有注册成功，将触发EOleSysError异常，可能的错误码有：E_INVALIDARG表示可能参数是非法的，E_OUTOFMEMORY表示内存不够，E_UNEXPECTED表示是未预计到的错误，CO_E_OBJISREG表示COM对象已经注册。只有EXE类型的COM服务器应当调用RegisterClassObject来注册COM对象，如果COM服务器是DLL，不要调用RegisterClassObject，而是在项目文件中实现并输出DllGetClassObject函数。

UpdateRegistry 函数

声明： virtual void _fastcall UpdateRegistry(bool Register);
这个函数也是用于注册类工厂创建的COM对象，不同的是，如果Register参数设为False，就撤消对COM对象的注册。

3.13 TTypedComObjectFactory

TTypedComObjectFactory是从TComObjectFactory直接继承下来的，如果说，TComObjectFactory是对应于TComObject的类工厂的话，那么，TTypedComObjectFactory就是对应于TTypedComObject的类工厂。
在ComObj单元中，TTypedComObjectFactory是这样声明的：

```
    class PASCALIMPLEMENTATION TTypedComObjectFactory :
public Comobj::TComObjectFactory
{
typedef Comobj::TComObjectFactory inherited;
public:
    _fastcall TTypedComObjectFactory(TComServerObject* ComServer, System::
    TMetaClass* TypedComClass, GUID &ClassID, TClassInstancing Instancing);
    _di_ITypeInfo _fastcall GetInterfaceTypeInfo(int TypeFlags);
    virtual void _fastcall UpdateRegistry(bool Register);
    _property _di_ITypeInfo ClassInfo ;
```

```
public:
    _fastcall virtual ~TTypedComObjectFactory(void) { }
};
```

根据类的传递规则，TTypedComObjectFactory也支持IUnknown、IClassFactory和IClassFactory2接口。与TComObjectFactory相比，TTypedComObjectFactory的构造函数稍有不同，并增加了ClassInfo特性和GetInterfaceTypeInfo函数。

ClassInfo 特性

声明：_property _di_ITypeInfo ClassInfo;
TTypedComObject的特点是无须调入类型库就可以直接读取类型信息，这个特性用于返回类型信息的接口引用。ITypeInfo接口用于从类型库中提取有关信息。

GetInterfaceTypeInfo 函数

声明：_di_ITypeInfo _fastcall GetInterfaceTypeInfo(int TypeFlags);
这个函数从由类工厂创建的TTypedComObject的实例中检索类型信息。

3.14 TActiveXPropertyPageFactory

TActiveXPropertyPageFactory也是直接从TComObjectFactory继承下来的，因此，它与TTypedComObjectFactory是平级的，它是ActiveX控件和ActiveForm

的特性页的类工厂。
在AxCtrls单元中，TActiveXPropertyPageFactory是这样声明的(按Object Pascal的语法声明)：

```
TActiveXPropertyPageFactory = Class(TComObjectFactory)
Protected
    Function CreateComObject(const Controller: IUnknown): TComObject; override;
Public
    Constructor Create(ComServer: TComServerObject; PropertyPageClass:
    TPropertyPageClass; const ClassID: TGUID);
End;
```

特性页也是COM对象(TPropertyPage)，它也需要用类工厂来创建实例。

构造函数

声明：constructor Create(ComServer: TComServerObject; PropertyPageClass: Tproperty PageClass; const ClassID: TGUID);
构造函数用于创建类工厂的实例，其中，ComServer参数指定COM服务器，通常把ComServ单元中的全局变量ComServer传递给ComServer参数。PropertyPageClass参数是特性页对象的类名，这个类必须是从TPropertyPage继承下来的。ClassID参数用于指定特性页对象的GUID，通常先声明一个常量，然后用这个常量作为ClassID参数。

3.15 TAutoObjectFactory

OLE自动化服务器也是COM对象，因此也需要类工厂。TAutoObjectFactory是TAutoObject的类工厂。在ComObj单元中，TAutoObjectFactory是这样声明的：

```
    class PASCALIMPLEMENTATION TAutoObjectFactory :
public Comobj::TTypedComObjectFactory
{
typedef Comobj::TTypedComObjectFactory inherited;
public:
    _fastcall TAutoObjectFactory(TComServerObject* ComServer, System::
    TMetaClass* AutoClass, const GUID &ClassID, TClassInstancing Instancing);
    virtual System::PInterfaceEntry _fastcall GetIntfEntry(const GUID &Guid);
    _property System::PInterfaceEntry DispIntfEntry = {read=FDispIntfEntry};
    _property _di_ITypeInfo DispTypeInfo = {read=FDispTypeInfo};
public:
    _fastcall virtual ~TAutoObjectFactory(void) { }
};
```

根据类传递的规则，TAutoObjectFactory支持IClassFactory和IClassFactory2接口。

DispIntfEntry 特性

声明：_property System::PInterfaceEntry DispIntfEntry;
其中，PInterfaceEntry是这样声明的：

```
struct PACKAGE TInterfaceEntry
{
TGUID IID;
void* VTable;
int IOffset;
};
typedef TInterfaceEntry *PInterfaceEntry;
```

这个只读的特性返回OLE自动化对象实现的调度接口的有关信息，实际上是一个记录的指针。这个特性在创建类工厂的实例时被初始化。

DispTypeInfo 特性

声明：_property _di_ITypeInfo DispTypeInfo;
这个只读的特性返回一个ITypeInfo接口的引用，通过这个接口可以读取OLE自动化对象的类型信息。

GetIntfEntry 函数

声明：virtual System::PInterfaceEntry _fastcall GetIntfEntry(const GUID &Guid);

这个函数返回Guid参数指定的接口的有关信息，实际上是一个记录的指针。
注意：这个函数一般只在内部调用，程序不要直接调用它。

3.16 TActiveXControlFactory

ActiveX控件是典型的COM对象，因此，ActiveX控件也需要用类工厂创建实例并注册，TActiveXControlFactory就是ActiveX控件的类工厂。
在AxCtrls单元中，TActiveXControlFactory是这样声明的：

```
    class PASCALIMPLEMENTATION TActiveXControlFactory :
public Comobj::TAutoObjectFactory
{
typedef Comobj::TAutoObjectFactory inherited;
public:
    _fastcall TActiveXControlFactory(Comobj::TComServerObject* ComServer,
    TMetaClass* ActiveXControlClass, System::TMetaClass* WinControlClass, const
    GUID &ClassID, int ToolboxBitmapID, const AnsiString LicStr, int MiscStatus);
    _fastcall virtual ~TActiveXControlFactory(void);
    void _fastcall AddVerb(int Verb, const System::AnsiString VerbName);
    virtual void _fastcall UpdateRegistry(bool Register);
    _property GUID EventIID;
    _property _di_ITypeInfo EventTypeInfo;
    _property int MiscStatus ;
```

```
    _property int ToolboxBitmapID ;
    _property System::TMetaClass* WinControlClass ;
};
```

EventIID 特性

声明：_property GUID EventIID;
这个只读的特性返回ActiveX控件的事件接口的GUID。

EventTypeInfo 特性

声明：_property _di_ITypeInfo EventTypeInfo;
这个只读的特性返回ActiveX控件的事件接口的类型信息。

MiscStatus 特性

声明：_property int MiscStatus;
这个只读的特性返回注册表中CLSID键值下的条目。

ToolboxBitmapID 特性

声明：_property int ToolboxBitmapID;
这个只读的特性返回用于代表ActiveX控件的图标的资源ID。

WinControlClass 特性

声明：_property System::TMetaClass* WinControlClass;

用ActiveX框架可以把一个基于VCL的控件转换为ActiveX控件，这个只读的特性返回对应的VCL控件。

AddVerb 函数

声明：void _fastcall AddVerb(int Verb, const System::AnsiString VerbName);
用鼠标右键单击ActiveX控件，将弹出一个快捷菜单，该菜单上的命令也称动作就是通过这个函数事先设置的。其中，Verb参数是个整数，用于设置动作的编号，自定义的动作通常设为正数。C++ Builder 3预定义了一些动作(负数或零)：

- OLEIVERB_PRIMARY
- OLEIVERB_SHOW
- OLEIVERB_OPEN
- OLEIVERB_HIDE
- OLEIVERB_UIACTIVATE
- OLEIVERB_INPLACEACTIVATE
- OLEIVERB_DISCARDUNDOSTATE
- OLEIVERB_PROPERTIES

VerbName参数是个字符串，用于设置动作名，实际上就是出现在快捷菜单中的命令名。字符串中可以出现&符号，&符号后面的那个字符将带下划线显示。

TActiveXControlFactory 的构造函数

声明：_fastcall TActiveXControlFactory(Comobj::TComServerObjec*ComServer, System::TMetaClass* ActiveXControlClass, System::TMetaClass* WinControlClass, const GUID &Class- ID, int ToolboxBitmapID, const System::AnsiStringLicStr, int MiscStatus);

这个构造函数用于创建类工厂的实例，其中，ComServer参数用于指定COM服务器，通常设为ComServ单元中的全局变量ComServer。ActiveXControlClass参数是ActiveX控件的类名，它是从TActiveXControl继承下来的。WinControlClass参数用于指定VCL控件的类名，它是从TWinControl继承下来的。ClassID参数用于指定ActiveX控件的GUID，LicStr参数是ActiveX控件的许可字符串，如果ActiveX控件不需要进行许可检查，LicStr参数可以设为空。MiscStatus参数用于设置存储到注册表中的状态位。

UpdateRegistry 函数

声明：virtual void _fastcall UpdateRegistry(bool Register);

这个函数用于在注册表中注册ActiveX控件或撤消注册，如果Register参数设为True，表示要进行注册。如果Register参数设为False，表示要撤消注册。一般不需要重载这个函数，除非您想在注册表中存储更多的信息。如果要重载这个函数并且Register参数设为True，首先要调用Inherited，如果Register参数设为False，要在最后调用Inherited。

3.17 TActiveFormFactory

ActiveForm可以理解为复合的ActiveX控件，也是典型的COM对象，因此，ActiveForm也需要类工厂，TActiveFormFactory就是TActiveForm的类工厂。在AxCtrls单元中，TActiveFormFactory是这样声明的(按Object Pascal的语法声明)：

```
TActiveFormFactory = Class(TActiveXControlFactory)
Public
    Function GetIntfEntry(Guid: TGUID): PInterfaceEntry; override;
End;
```

可见，TActiveFormFactory是从TActiveXControlFactory继承下来的，重载了GetIntfEntry函数。

至此，ActiveX框技术已经全部介绍完毕，也许有的读者会觉得上述内容很深奥，不太好理解，事实上的确如此，读者只有深刻领会了C++ Builder 3面向对象的编程思想以及“对象接口”的语法，才能真正掌握ActiveX框架的结构和精髓。

第四章　“Type Library”编辑器

用C++ Builder 3创建OLE自动化对象、ActiveX控件以及ActiveForm的关键在于用“Type Library”编辑器建立和编辑类型库。所谓类型库，实际上是一个文件，包含了COM对象中的数据类型、接口、成员函数等信息，当您用C++ Builder 3提供的向导创建OLE自动化对象、ActiveX控件以及ActiveForm时，这些向导会自动生成类型库文件，并作为资源连接到所要创建的DLL或EXE中，让其它应用程序或其它编程工具识别和使用。

4.1　关于类型库的概述

类型库的主要内容是关于COM对象的数据类型、接口和成员函数的信息，包括接口的标识符IID和类标识符CLSID以及每个成员的识别号DispID。此外，类型库中还包括自定义接口的类型信息、由ActiveX服务器输出的例程以及对其它类型库的引用。

如果仍然使用传统的开发工具，您不得不用脚本语言如IDL或ODL来描述类型库，并通过编译器运行脚本。现在有了C++ Builder 3，您就轻松多了，C++ Builder 3提供了一个非常好的实用工具叫“Type Library”编辑器，您用不着学习IDL或ODL，因为“Type Library”编辑器采用可视化的手段，让您

交互生成和编辑类型库中的信息，并自动生成C++源代码。类型库文件实际上包括两个部分，一个是已编译的类型库，扩展名是.TLB，另一个是相应的接口源文件，扩展名是.CPP。

什么情况下COM对象需要有类型库？总的原则是，如果COM对象需要向外部代码暴露，该COM对象就要有类型库，具体来说，下列情况下需要创建类型库：

- 凡是支持虚拟方法表的对象需要创建类型库，因为对虚拟方法表的引用必须在编译期就是确定的，也就是说，必须在类型库中描述虚拟方法表。
- 凡是支持IProvideClassInfo接口的对象如TTypedComObject的派生类必须创建类型库，因为外部代码可能会直接读取类型库中的信息。
- 凡是ActiveX控件需要创建类型库，类型库必须作为资源连接到ActiveX控件的DLL或OCX文件中。
- OLE自动化服务器必须提供类型库，OLE客户才能实现所谓的“ID Binding”。
- 有时尽管类型库不是必需的，但可以用于标识拖放的对象，最好也要建立类型库。

如果COM对象仅仅是内部使用的，就用不着创建类型库，您可以直接在一个单元文件中描述COM对象的有关接口，只要符合接口的语法。

类型库创建好后，C++ Builder 3的编译器、“Type Library”编辑器自身就可以访问类型库，这是通过三个特殊的接口ITypeLib、ITypeInfo和ITypeComp实现的，其中，ITypeLib接口提供了访问类型库中的信息以及通过ITypeInfo接口直接读取类型信息的能力。ActiveX框架中绝大多数的类都支持ITypeLib

接口，TTypedComObject还支持ITypeInfo接口。
即使应用程序并不需要类型库，但使用类型库能带来以下方便和好处：

- 编译器能够在编译期进行类型匹配检查，并及时报错。
- 支持所谓的“ID Binding”，从而加快方法调用的速度。
- 可以用类似于“Type Library”编辑器的工具打开类型库，查看COM对象内部的情况。
- 可以调用RegisterTypeLib函数在Windows的注册表中注册暴露的对象，也可以调用UnRegisterTypeLib函数从注册表中撤消注册。

4.2 创建一个新的类型库

大多数情况下，您并不需要自己创建类型库，因为当您用C++ Builder 3的向导创建ActiveX控件、ActiveForm、OLE自动化对象时，向导会自动生成类型库。但是，如果您想单独创建一个类型库，作为资源连接到OCX、DLL或EXE中，就要自己创建类型库。
要创建一个新的类型库，使用“File”菜单上的“New”命令打开“New Items”对话框(实际上就是C++ Builder 3的对象库)，翻到“ActiveX”页，如图4.1所示。

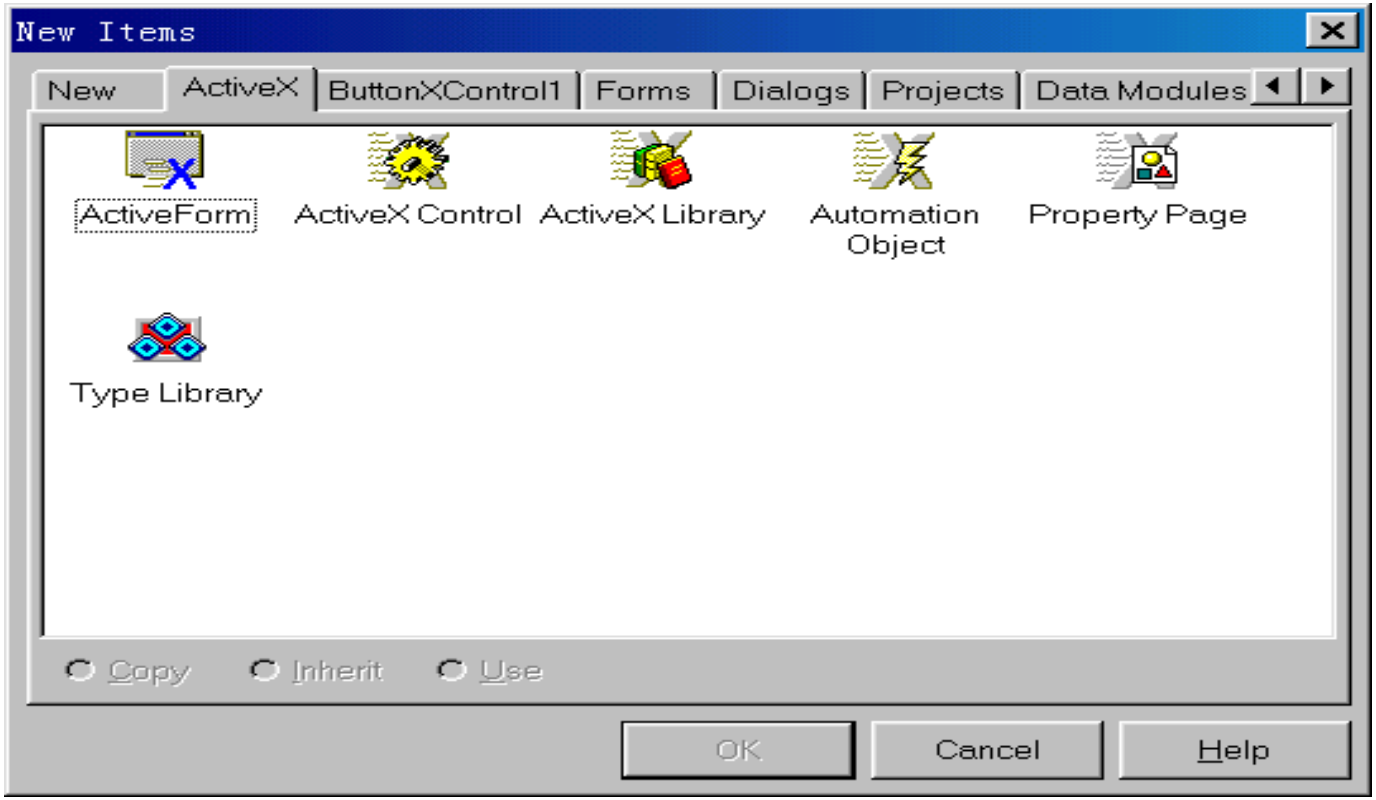


图 4.1　创建一个新的类型库

双击“Type Library”图标，C++ Builder 3将创建一个新的空白的类型库，并自动打开“Type Library”编辑器，提示您输入类型库的名称，如图4.2所示：

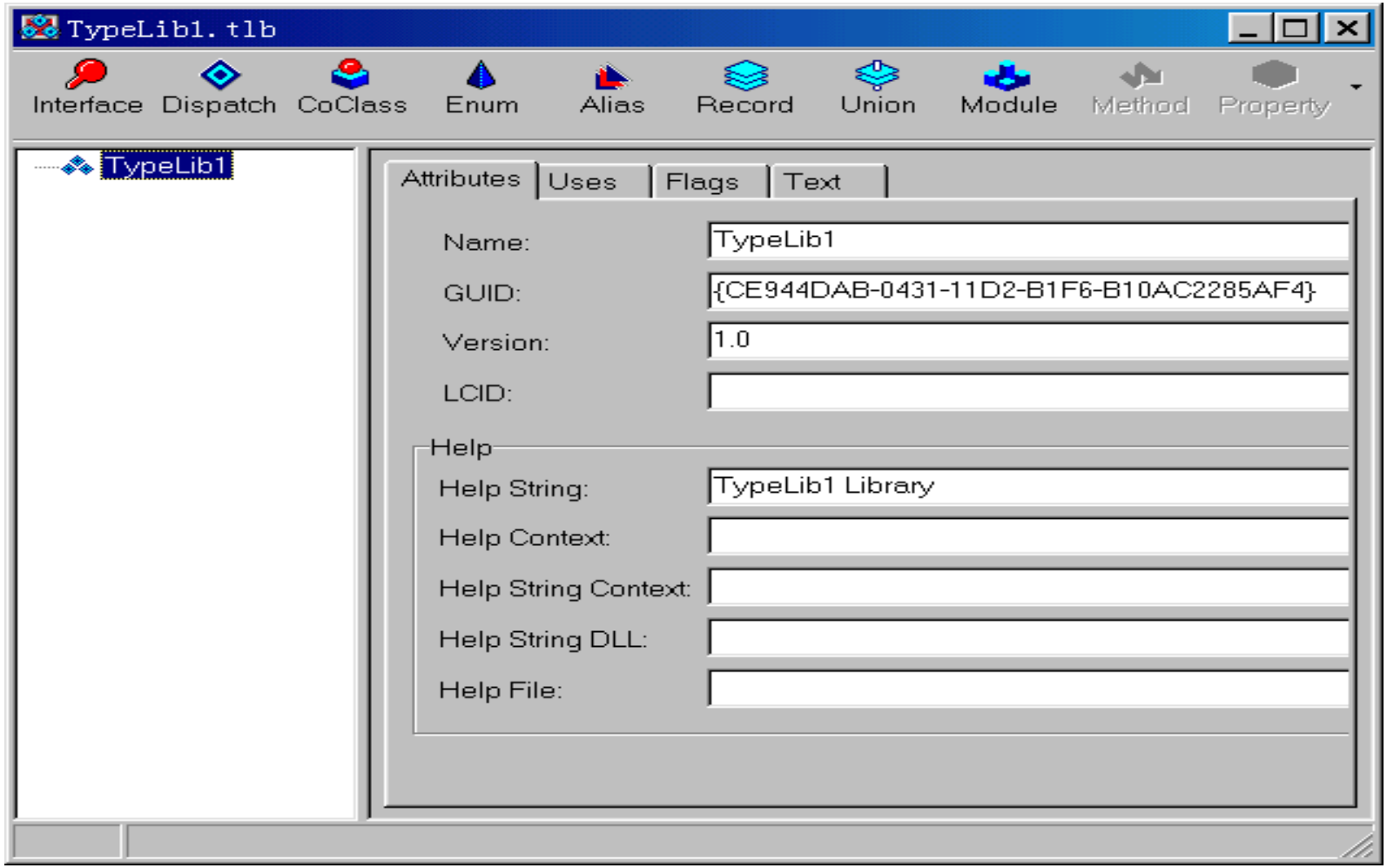


图 4.2　一个新的空白的类型库

给类型库重新取个名字，然后您就可以在类型库中加入Interface、DispInterface、CoClass、Enum、Property和Method，后面将详细介绍怎样使用“Type Library”编辑器。

4.3 “Type Library”编辑器的窗口

当您用C++ Builder 3的向导创建ActiveX控件、ActiveForm、OLE自动化对象时，向导会自动创建类型库。当然，由向导创建的类型库只支持C++ Builder 3内建的功能，如果您想加入一些自定义的功能并且要向外部代码暴露这些功能，您就需要编辑类型库。

要打开和编辑当前项目的类型库，您可以直接使用“View”菜单上的“Type Library”命令。要打开和编辑非当前项目的类型库，您可以使用“File”菜单上的“Open”命令，在“File of Type”框内选择“Type Library(.TLB)”，然后指定一个类型库文件。

下面我们以创建一个ActiveX控件为例。假设我们要创建的ActiveX控件是基于VCL中的TButton元件，打开的“Type Library”编辑器，如图4.3所示。

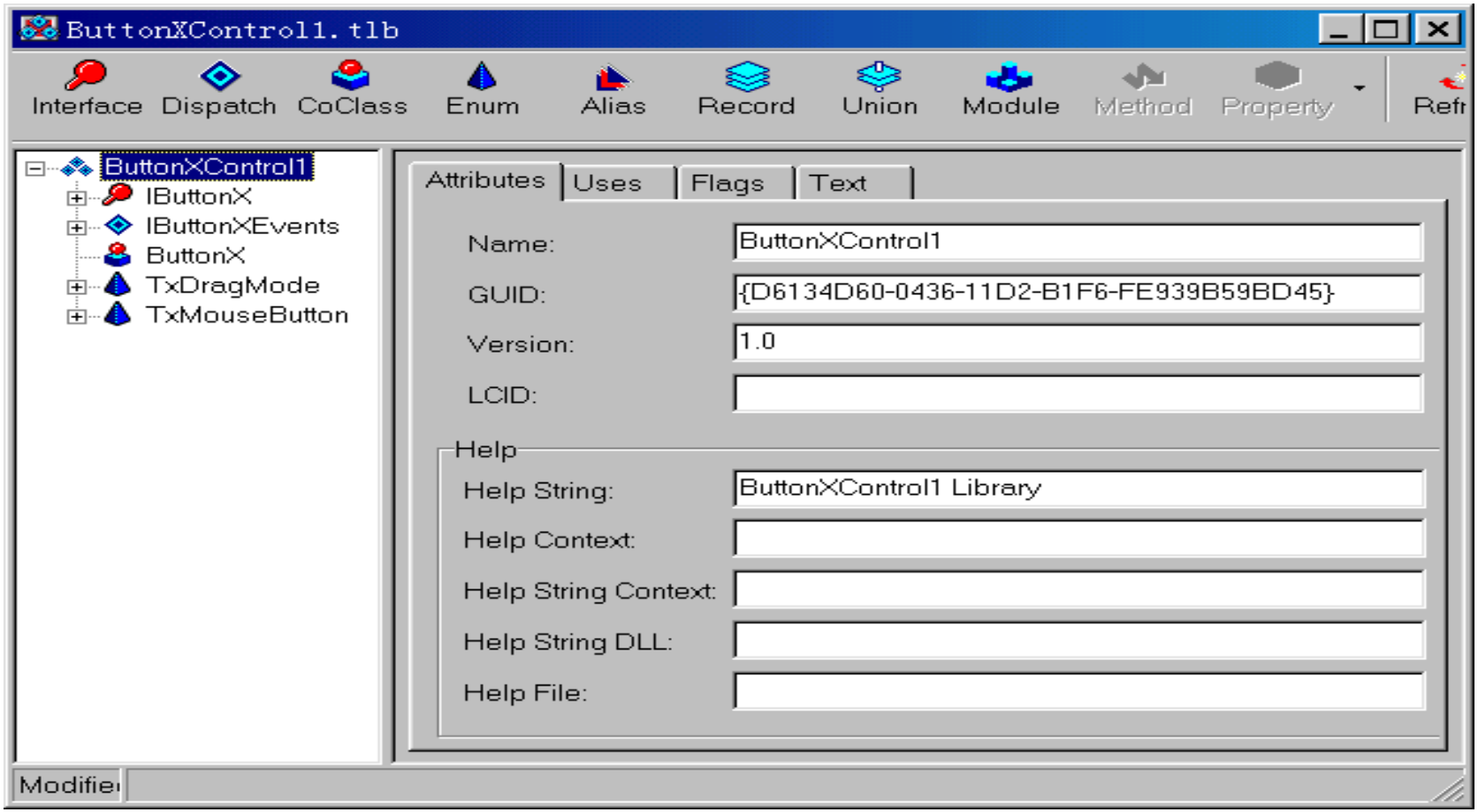


图 4.3 “Type Library”编辑器

“Type Library”编辑器没有菜单，只有一个工具栏。除了工具栏外，“Type Library”编辑器分成左右两个窗格，左边的窗格以树状视图的方式显示类型库的结构，最顶层的节点是有关整个类型库的信息，下面是类型库中包

含的对象(可增减)。右边的窗格是个多页控件，随着在左边窗格中选择的对象的不同，右边的窗格将显示不同的信息。
左边的窗格也叫对象列表窗格，类型库中的所有对象以树状视图的形式列在此窗格中，包括Interface、Dispinterface、CoClass、Property、Method、Enum、Record、类型定义、模块等，其中有的对象是不可编辑的或者说是只读的。在“Type Library”编辑器中，不同类型的对象以不同的图标表示。
当您用“Type Library”编辑器打开一个类型库特别是其它工具创建的类型库时，很可能会发生转换错误，因为有的OLE对象中的数据类型或其它文法不能被C++支持，这时候，“Type Library”编辑器就会在一个专门的错误记录窗格中显示有关的错误信息。

4.4 类型库的一般信息

在“Type Library”编辑器的对象列表窗格中，最顶层的节点就是类型库，如选择这个节点，右边的窗格中将显示“Attributes”页、“Uses”页、“Flags”页和“Text”页。
“Attributes”页显示类型库的一般信息，如图4.4所示。

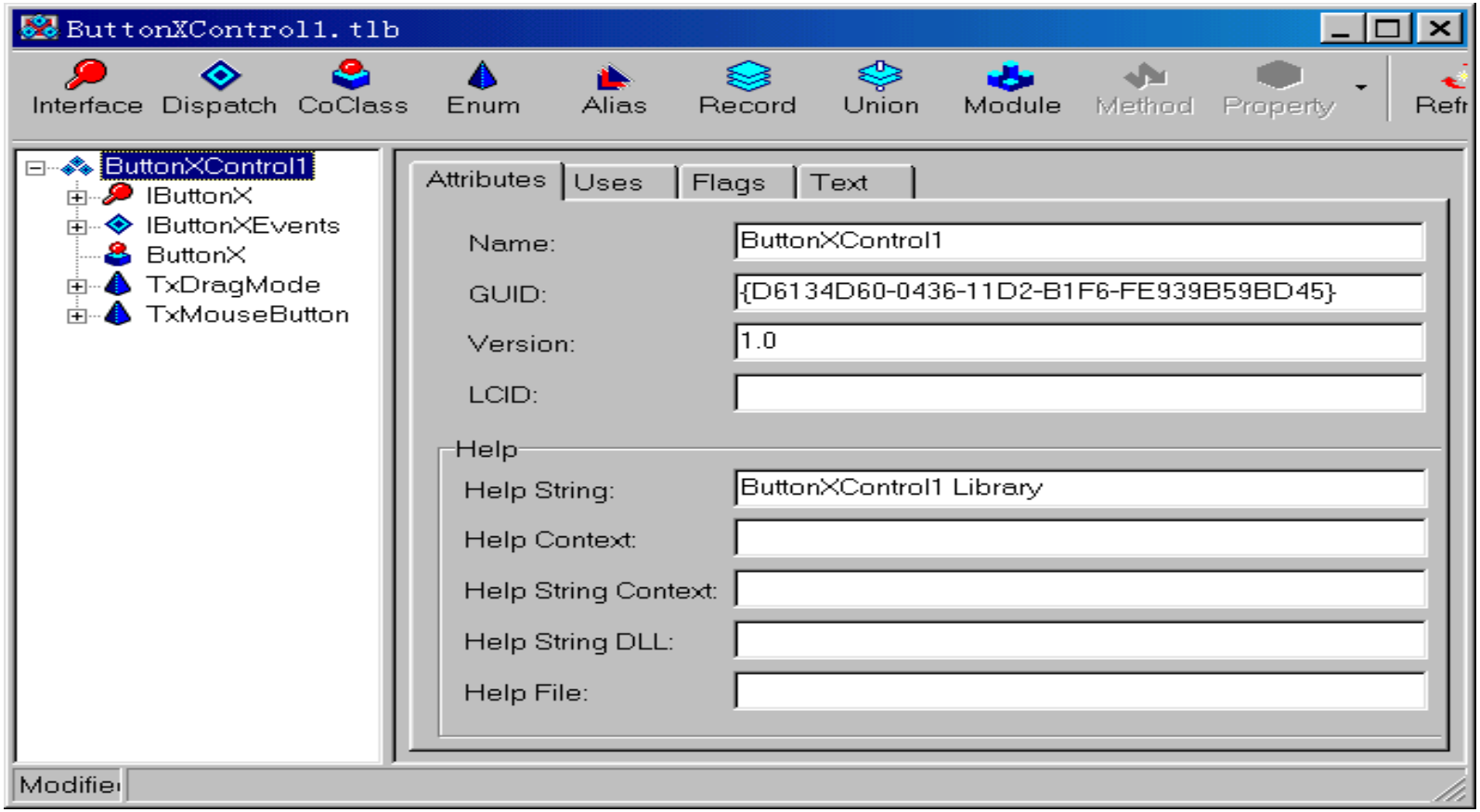


图 4.4　类型库的“Attributes”页

- Name　　类型库的文件名，中间不要有空格和标点符号，扩展名是.TLB。
- GUID　　类型库的全局识别号(128位存储)。
- Version　　类型库的版本号，格式是*n.m*，其中*n*是主版本号，*m*是副版

本号。

- LCID　　类型库中字符串的语言(这个框是只读的)。
- Help String　输入类型库的简短描述(最好不要让它空着)。
- Help File　　指定类型库的帮助文件。
- Help Context　指定类型库的主题在帮助文件中的上下文编号。

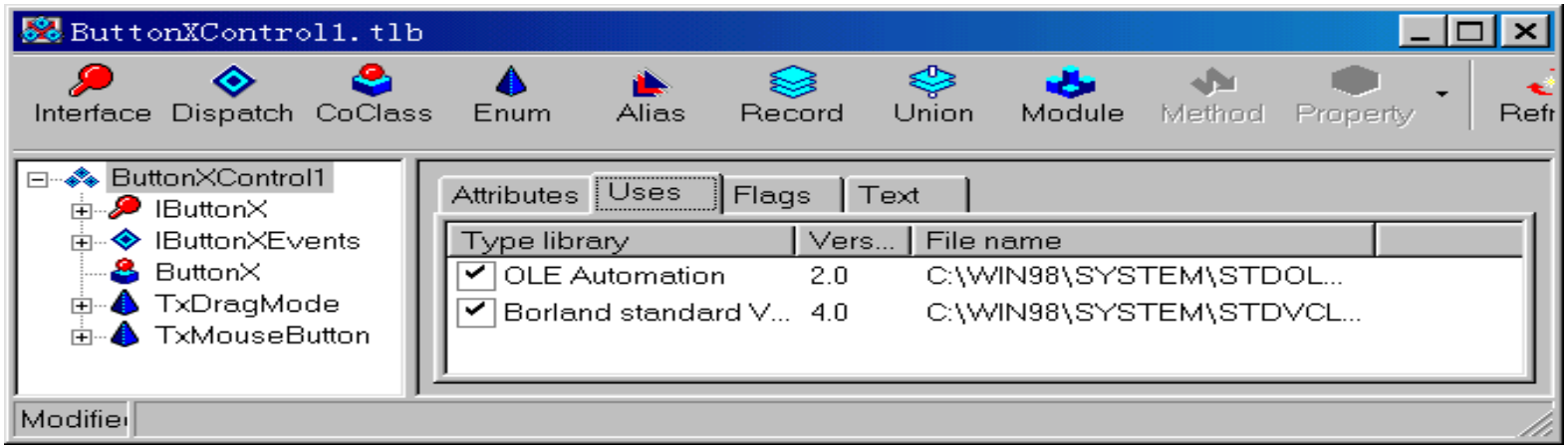


图 4.5　引用其它类型库

“Uses”页显示的是本类型库还引用了哪些其它类型库，如图4.5所示。通过引用其它类型库，可以借用其它类型库中已定义好的类型信息。例如，STDOLE32.TLB文件中定义了IDispatch接口，在您的类型库中只要引用一下就可以了，而不必什么都从头定义。

要打开某个被引用的类型库，只要在某个类型库上单击鼠标右键，在弹出的菜单中选择“View Type Library”命令，C++ Builder 3将打开另一个“Type Library”编辑器，显示该类型库的信息。如果在弹出的菜单中选择“Show All

Type Libraries”命令，将显示所有已注册的类型库，要引用其中一个类型库，只要选中该类型库前面的复选框。如果要让“Uses”页只显示正在引用的类型库，在弹出的菜单中选择“Show Selected”命令。

C++ Builder 3的“Type Library”编辑器增加了“Text”页，用于显示类型库中某个对象的IDL脚本，如图4.6所示。

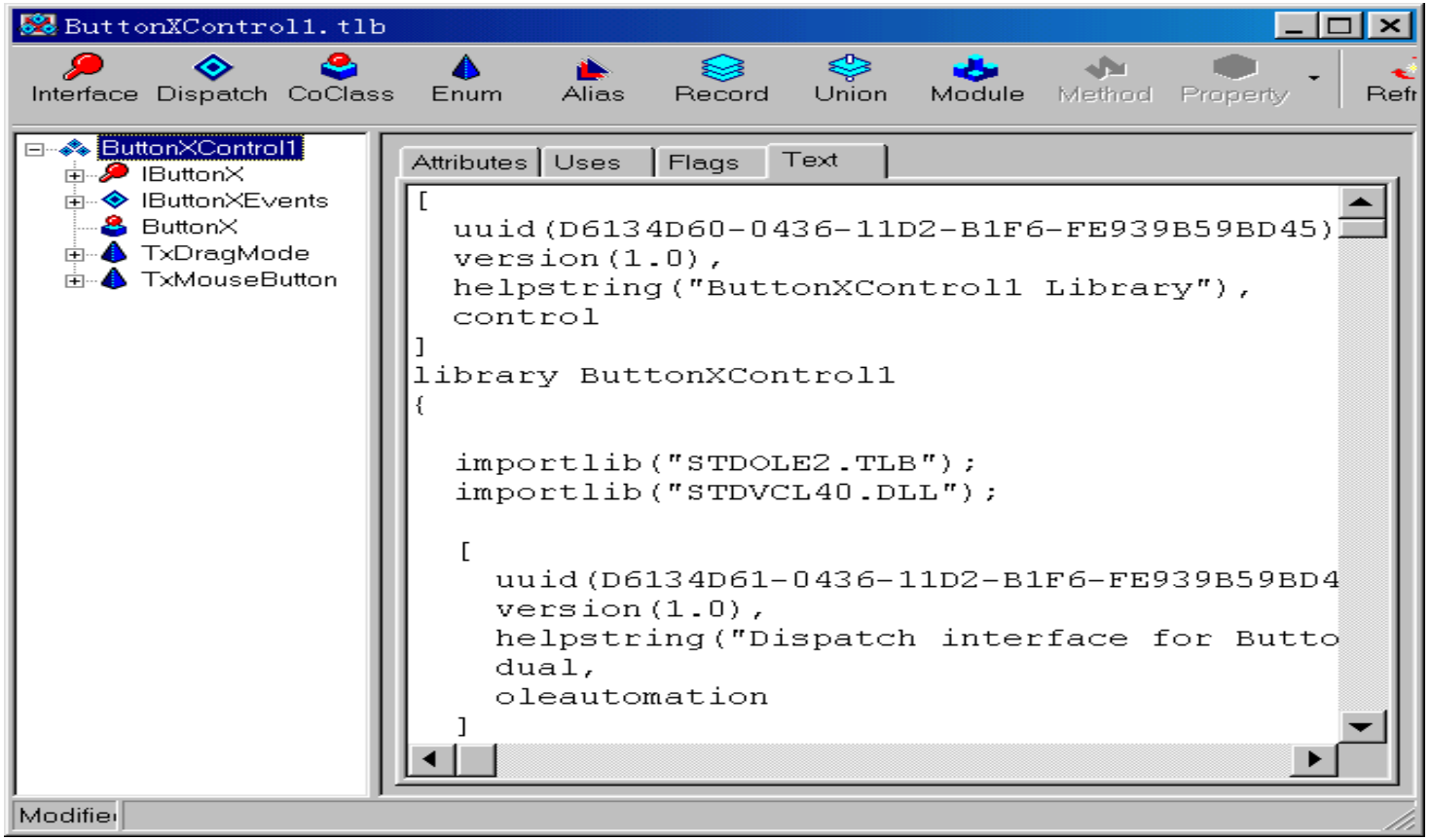


图 4.6　显示 IDL 脚本

您在其它页上所做的任何修改，都将反映到“Text”页上。您也可以在“Text”页上直接修改代码，其它页上将同步修改。这就是说，“Type Library”编辑器是双向的工具。不过，您得对IDL的文法非常熟悉，并且只能使用已定义的标识符。

“Flags”页用于设置类型库的标志，如图4.7所示。

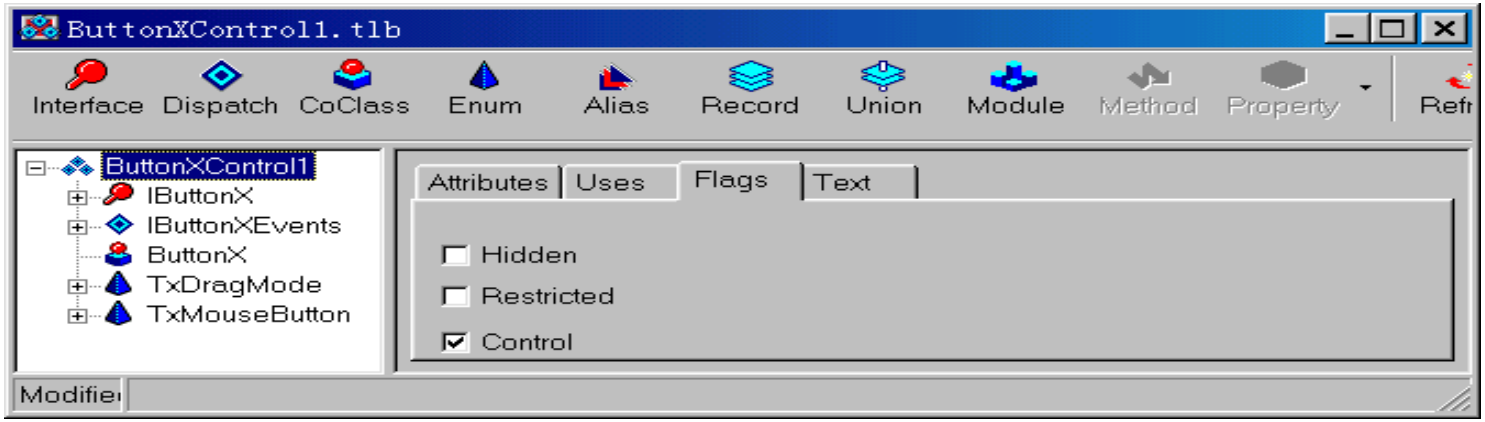


图 4.7　设置类型库的标志

如选中“Hidden”复选框，表示类型信息虽然存在，但却是隐藏的，不能显示出来。

如选中“Restricted”复选框，表示不允许宏编程环境如Visual Basic使用此类型库。

如选中“Control”复选框，表示类型库描述的是一个控件。

4.5 接　　口

要在类型库中加入一个接口，先选择最顶层的类型库节点，然后单击工具栏上的“Interface”图标，也可以在类型库节点上单击鼠标右键，在弹出的菜单中选择“New”，再选择“Interface”。新加入的接口的名称默认是Interface1，您可以给它换名。

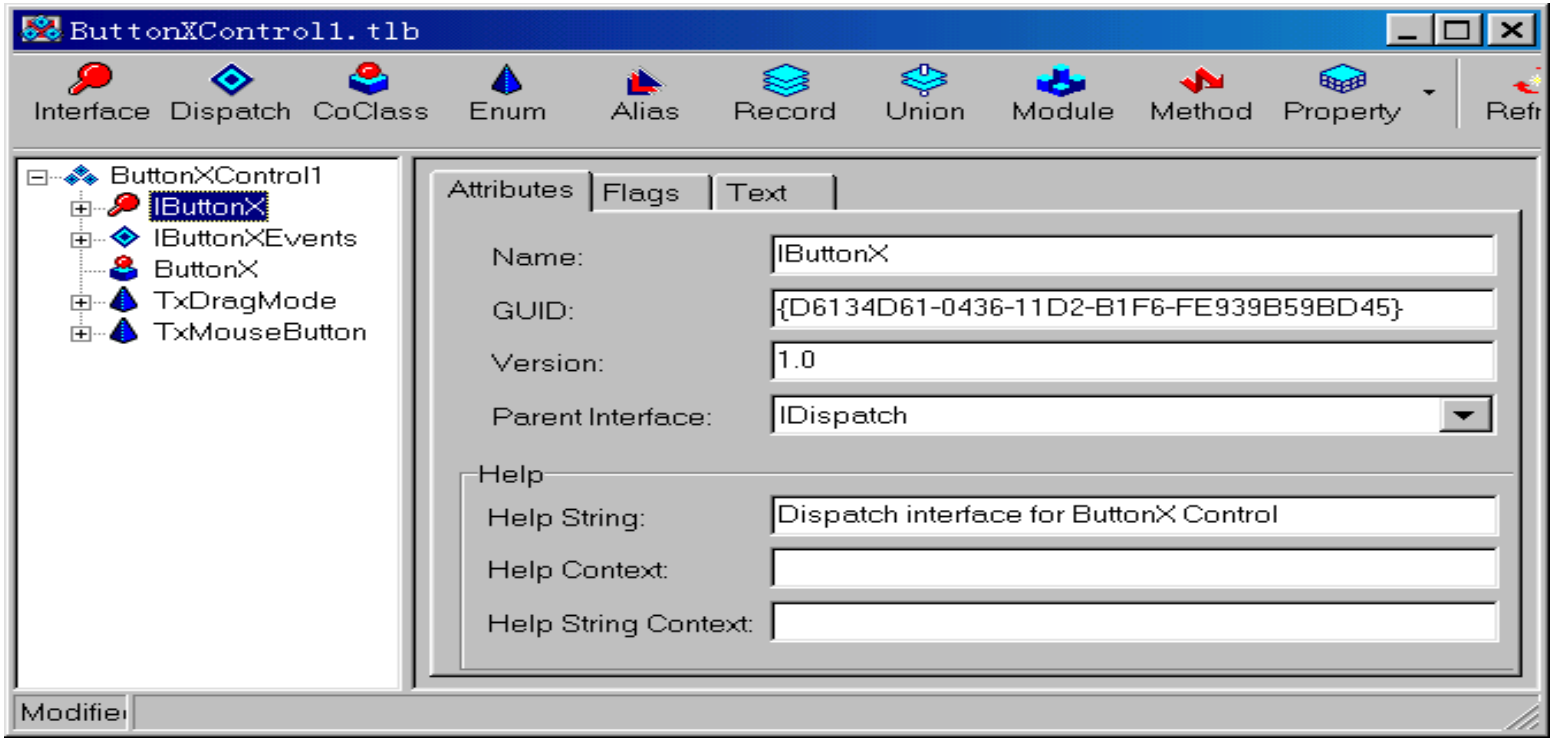


图 4.8　类型库接口的属性

下面我们还是以ButtonXControl的IButtonX节点(带  符号)为例，介绍怎样设置接口的属性和怎样向接口中加入成员。

“Attributes”页显示接口的属性，如图4.8所示。

- Name　　接口的名称。
- GUID　　接口的全局识别号(128位存储)。
- Version　　接口的版本号，格式是*n.m*，*n*是主版本号，*m*是副版本号。
- Parent Interface　祖先接口的名称。
- Help String　　关于该接口的简短描述。
- Help Context　　接口的主题在帮助中的上下文编号。

“Flags”页用于设置接口的标志，如图4.9所示。

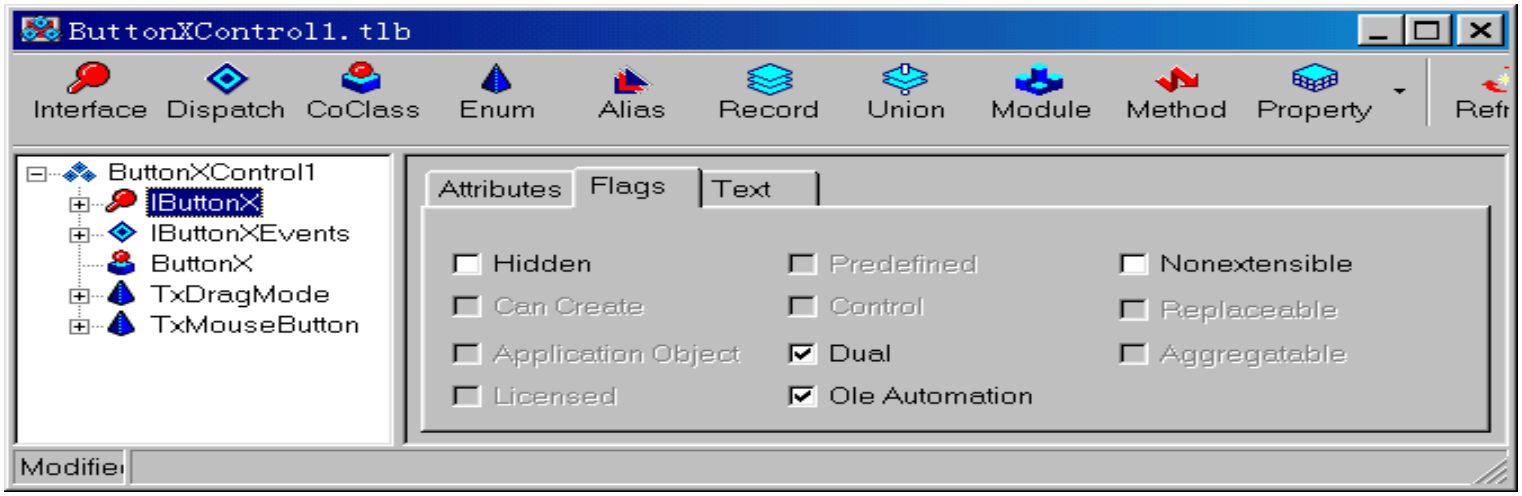


图 4.9　设置接口的标志

如选中“Hidden”复选框，表示接口虽然存在，但不能显示出来。

如选中“NoneExtensible”复选框，表示IDispatch接口只实现列在接口描述表中的特性和方法，不能作为其它接口的祖先接口。

如选中“Dual”复选框，表示接口是一个双重接口，同时支持IDispatch接口和虚拟方法表。

如选中“OLE Automation”复选框，表示接口只能使用与OLE自动化相容的数据类型。

4.6　在接口中加入成员

要向接口中加入成员(指特性或方法)有两种方式，一是在对象列表窗格中选

择一个接口，然后单击工具栏上的“Property”或“Method”图标。第二种方式是直接在“Text”页上按IDL的文法键入特性或方法的声明(参见图4.10)。上述两种操作方式是等价的，因为“Type Library”编辑器是一个双向的工具。

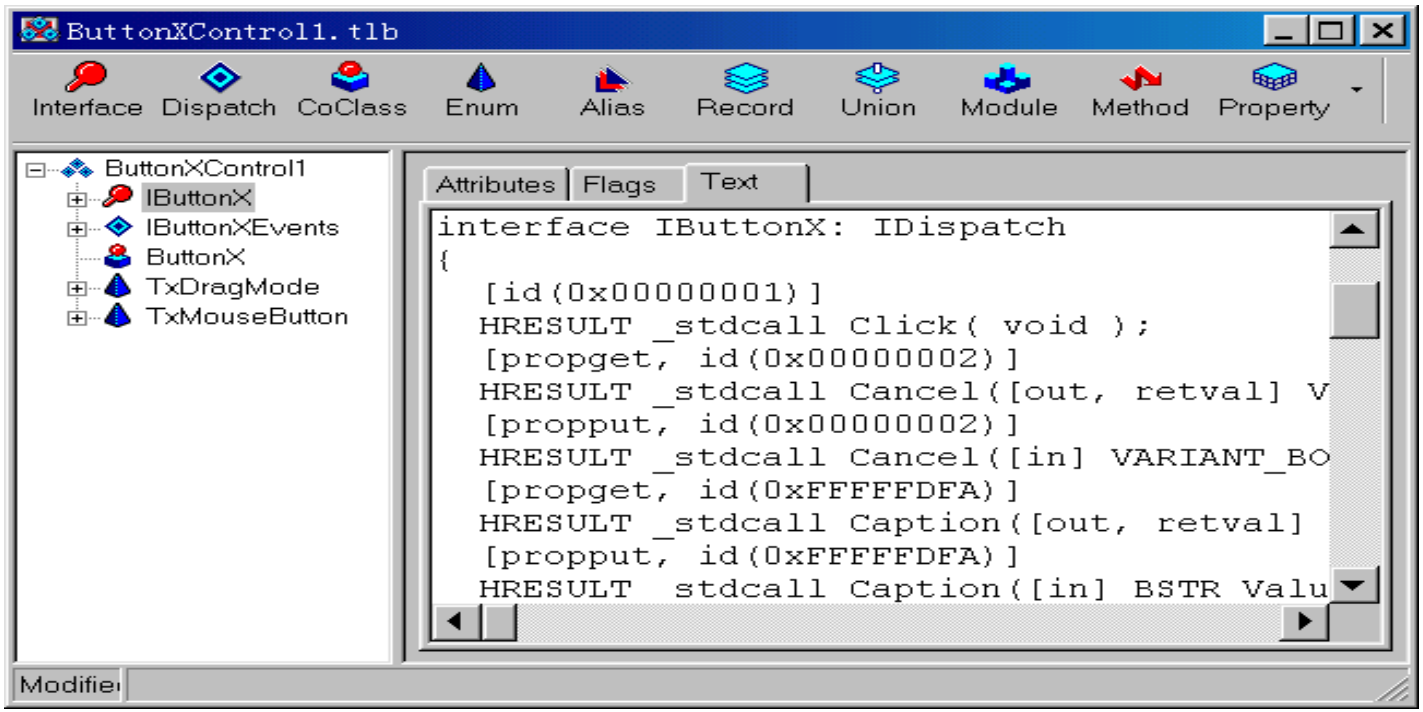


图 4.10　在接口中加入成员

加入了成员后，您就可以在对象列表中扩展接口节点，选择其中一个特性或方法，然后在右边的窗格中显示并编辑该成员的属性、参数和标志。“Attribute”页用于显示成员的属性，如图4.11所示。

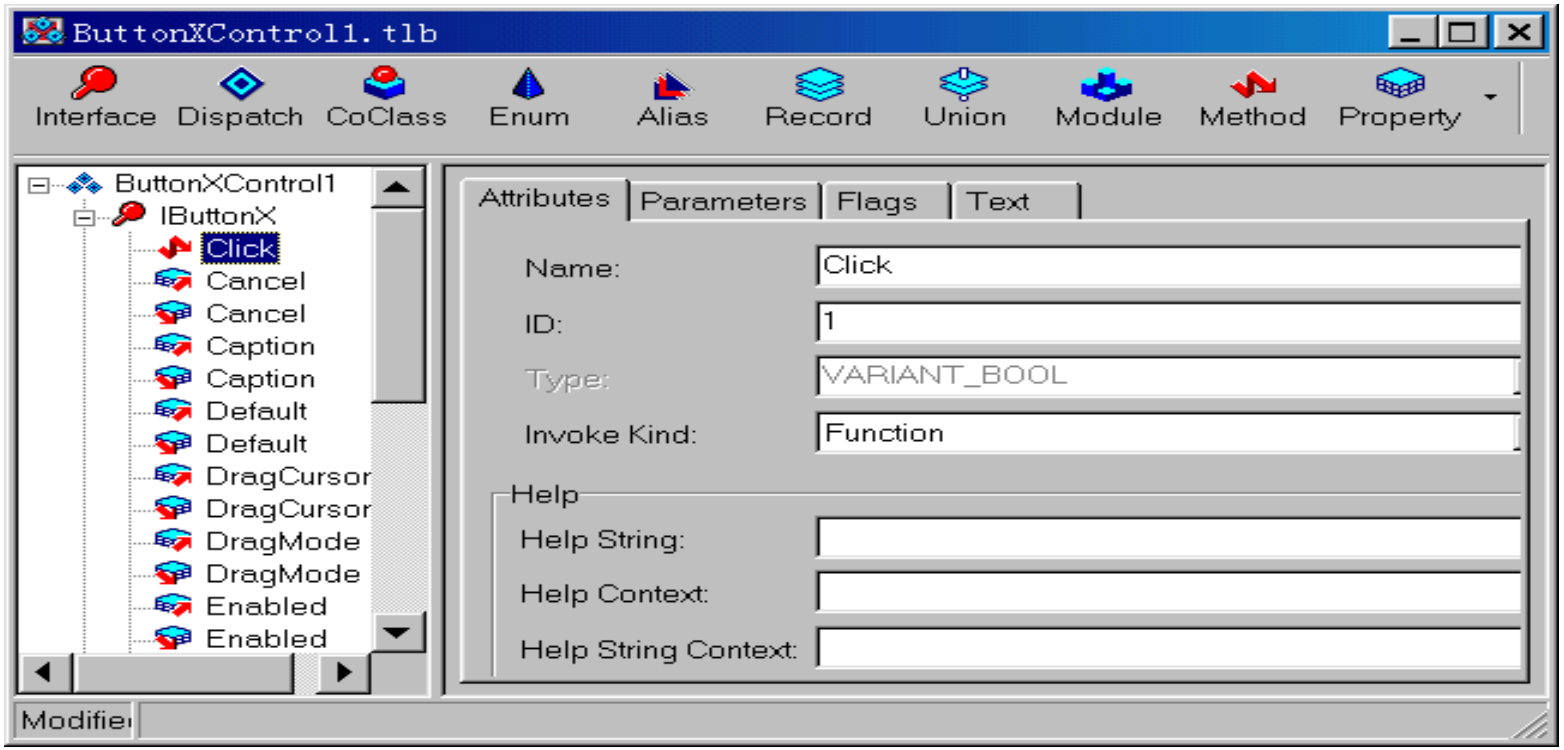


图 4.11　接口成员的属性

- Name　　成员的名称。
- ID　　成员的分配号。
- Invoke kind　　成员是方法还是特性。
- Help String　　关于该特性或方法的简短描述。
- Help Context　　特性或方法的主题在帮助文件中的上下文编号。

“Parameters”页用于设置函数的参数和返回值，如图4.12所示。

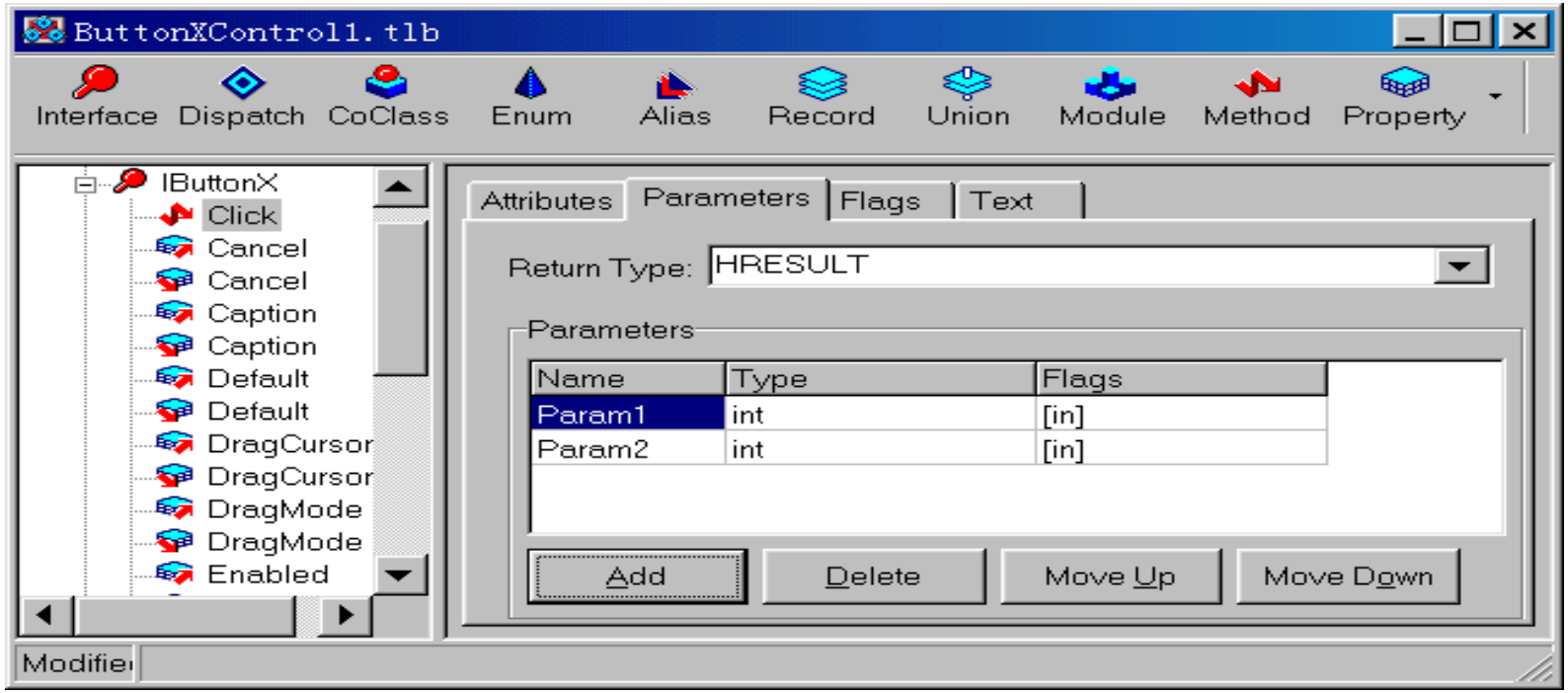


图 4.12　设置函数的参数和返回值

单击“Add”按钮可以增加一个参数，单击“Delete”按钮可以删减一个参数。

“Name”栏用于指定参数名称，“Type”栏用于指定参数的数据类型。要改变参数的标志，双击“Flags”栏，C++ Builder 3将显示“Parameters Flags”对话框，如图4.13所示。

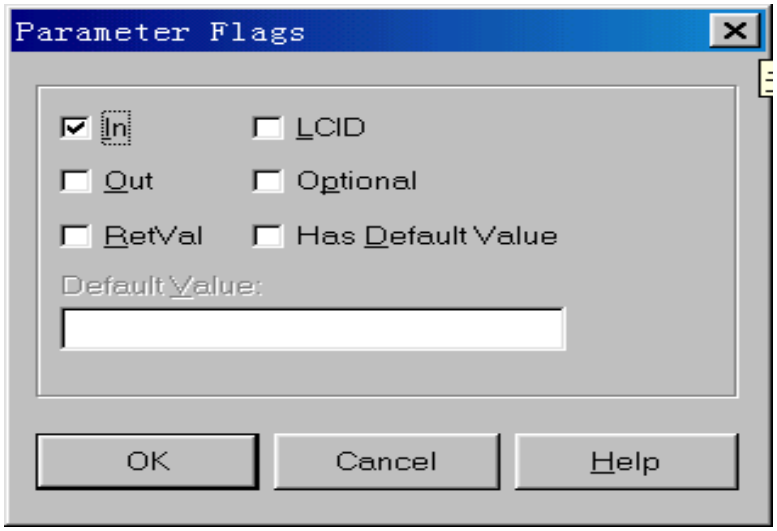


图 4.13　设置参数的标志

“Flags”页用于设置接口成员的标志，如图4.14所示。

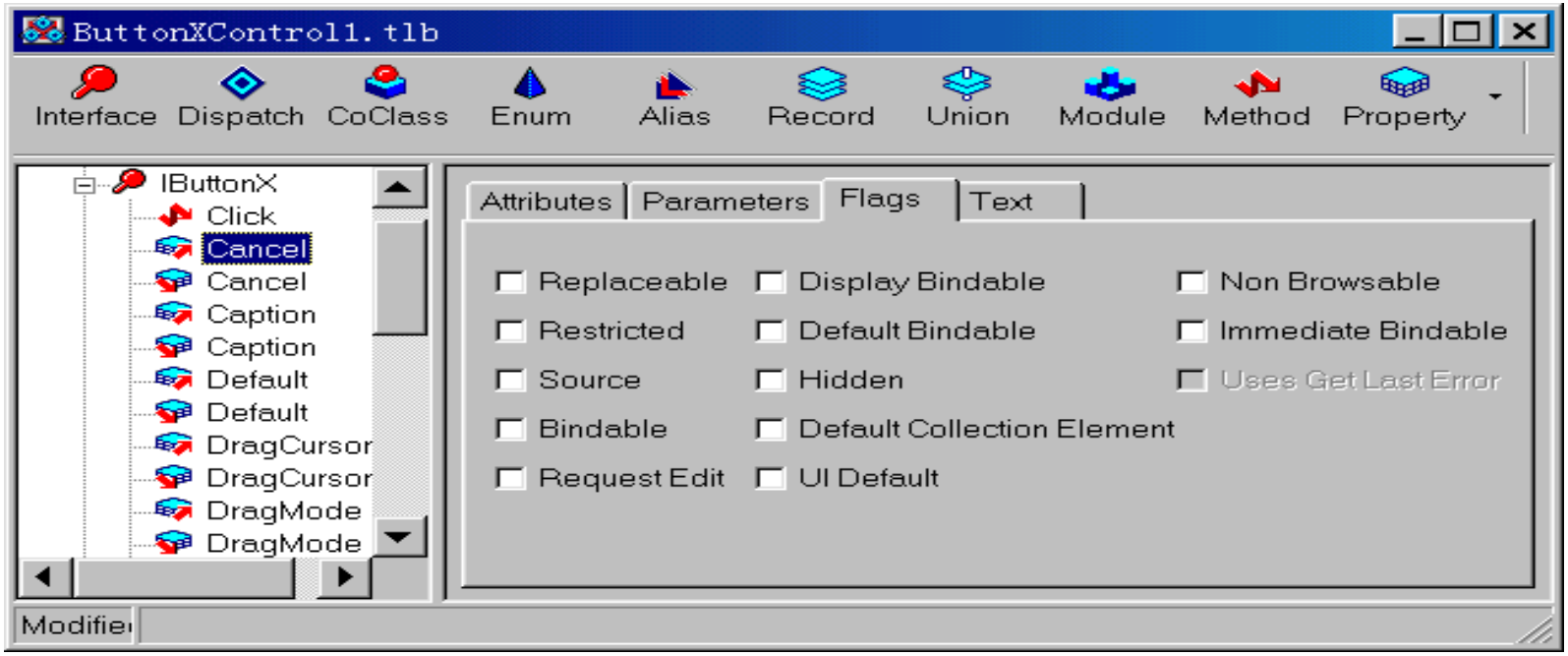


图 4.14　设置接口成员的标志

如选中“Replaceable”复选框，表示支持IConnectionPointWithDefault。

如选中“Restricted”复选框，不允许宏编程环境如Visual Basic使用该特性或方法。

如选中“Source”复选框，表示成员返回一个对象或VARIANT作为事件的源。

如选中“Bindable”复选框，表示特性支持数据装订。

如选中“Request Edit”复选框，表示特性支持OnRequestEdit事件。

如选中“Display Bindable”复选框，表示特性是可装订的。

如选中“Hidden”复选框，表示特性虽然存在，但不能显示出来。

如选中“Default Collection Element”复选框，允许在Visual Basic中对代码优化。
如选中“Non Browsable”复选框，特性只出现在不需要显示值的浏览器中。
另外，对于OLE自动化对象的特性来说，还有个“Write By Reference”复选框，如选中此复选框，表示特性通过指针而不是值来传递，这样可以改善程序的性能。

4.7 调度接口

调度接口实际上也是接口，只不过这种类型的接口中的成员可以通过IDispatch接口的Invoke函数调用。调度接口中的成员通常是事件句柄，如OnClick、OnKeyPress等。
下面我们还是以ButtonXControl的IButtonXEvents节点(带  符号)为例，介绍调度接口的用法，如图4.15所示。

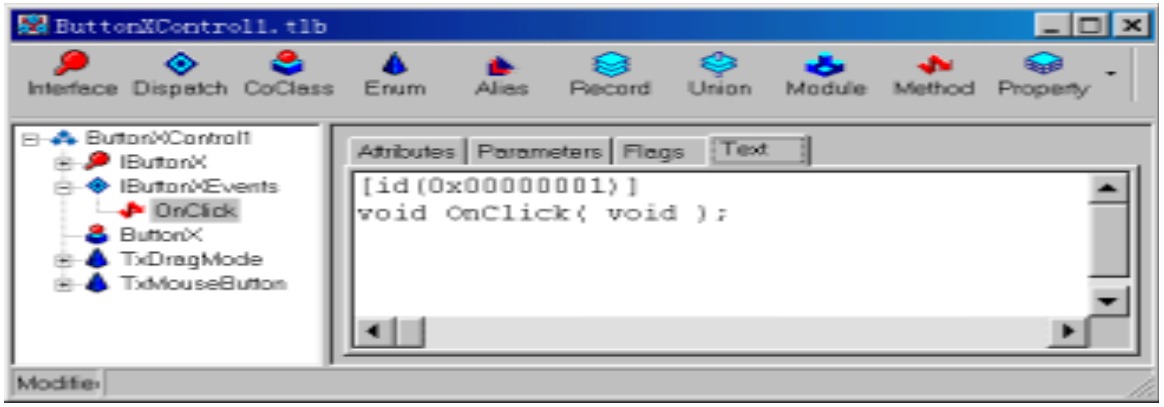


图 4.15　调度接口的成员

从“Type Library”编辑器的角度看，调度接口与一般的接口在用法上没有什么区别，它的“Attributes”页、“Parameters”页、“Flags”页和“Text”页请参见上一节。唯一要注意的是，调度接口中的事件句柄不能有返回值。

4.8　类型库枚举

所谓类型库枚举，其概念与C++中的枚举非常相似，也是一组相关的常量。您可以在“Type Library”编辑器中把枚举作为一个自定义的数据类型使用。要在类型库中加入一个枚举，单击工具栏上的“Enum”图标，新的枚举刚开始的时候是空的，您必须自己声明枚举常量，并且只能在“Text”页上直接键入，如图4.16所示。

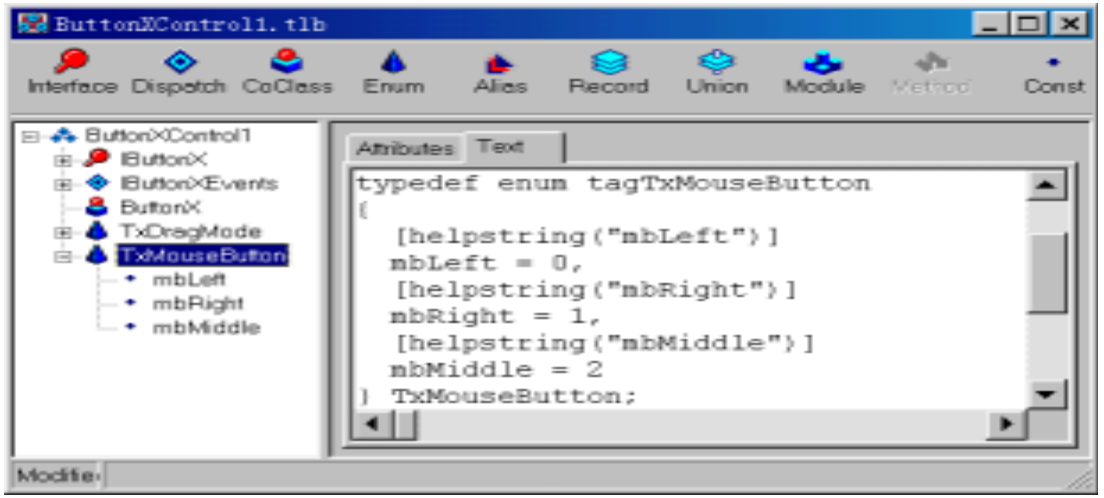


图 4.16　键入枚举常量

可以看出，枚举TxMouseButton声明了三个枚举常量，分别是mbLeft、mbRight、mbMiddle，其值分别是0、1、2。请注意声明的格式。

“Attributes”页用于显示和编辑类型库枚举的属性，如图4.17所示。

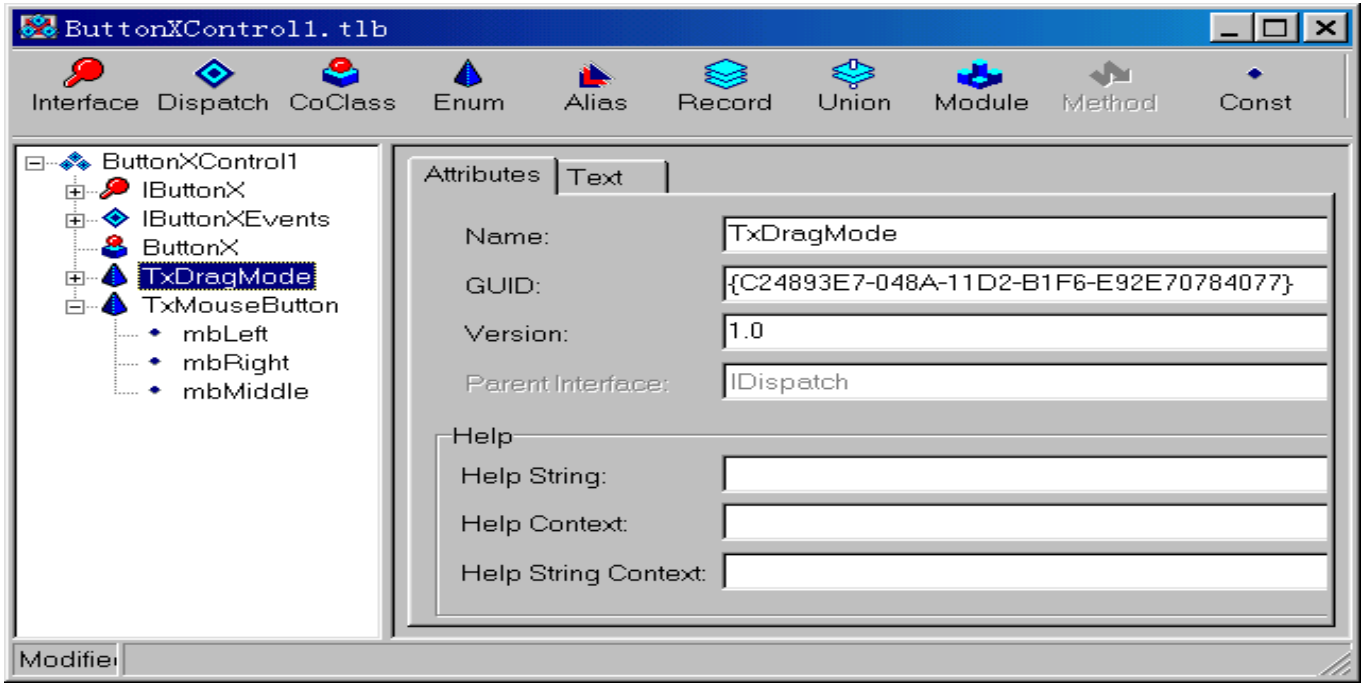


图 4.17　类型库枚举的属性

- Name　　类型库枚举的名称。
- GUID　　类型库枚举的全局识别号(128位存储)。
- Help String　　类型库枚举的简短描述(最好不要让它空着)。
- Help Context　　类型库枚举的主题在帮助文件中的上下文编号。
- Version　　类型库枚举的版本号,格式是*n.m*,*n*是主版本号，*m*是副版本号。

4.9 组件类(CoClass)

类型库的组件类(Component Class，简称CoClass)描述的是整个COM对象，它统一管理COM对象的接口和调度接口。

要在类型库中加入一个组件类，单击工具栏上的“CoClass”图标。新加入的组件类默认是空的，要在组件类中加入一个成员，在“Implements”页上单击鼠标右键，在弹出的菜单中选择“Insert Interface”命令打开“Insert Interface”对话框，这个对话框列出了类型库中所有已定义的接口，也包括本类型库引用的其它类型库中的接口，选择其中一个接口，单击OK按钮。

要删除一个接口，在弹出的菜单中选择“Remove Interface”命令。

在对象列表中选择一个组件类，右边的窗格中将显示“Attributes”页、“Implements”页、“Flags”页、“Text”，其中，“Flags”页用于设置组件类的标志，如图4.18所示。

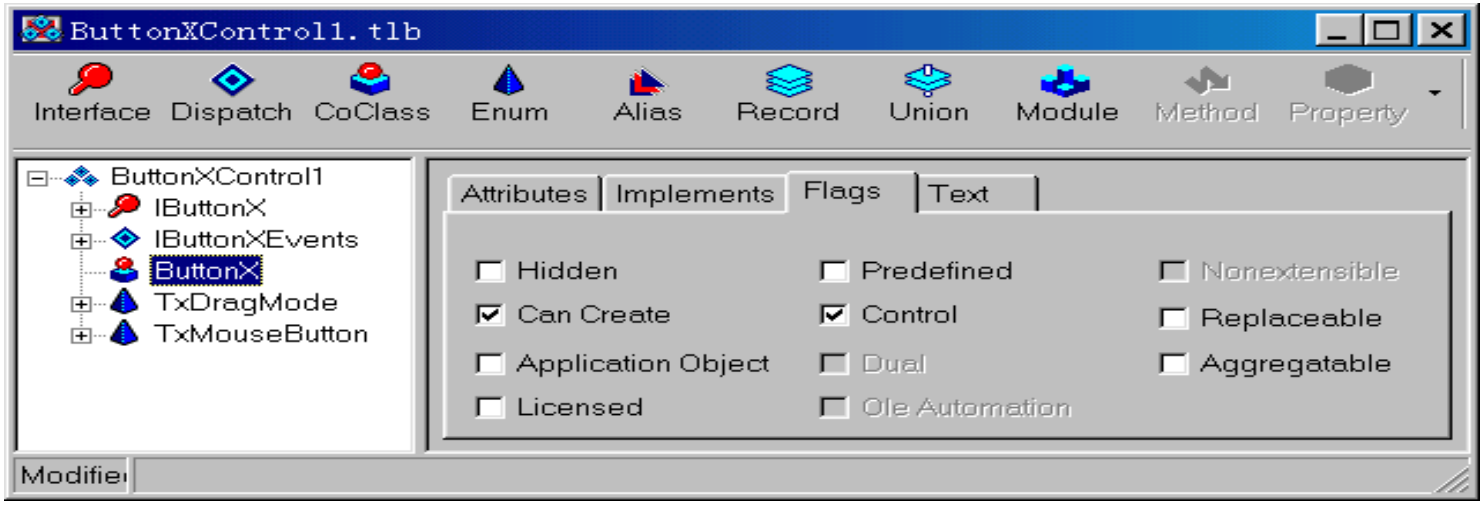


图 4.18　设置组件类的标志

如选中“Hidden”复选框，表示虽然接口存在但隐藏显示。

如选中“Can Create”复选框，表示可以由CoCreateInstance创建实例。

如选中“Application Object”复选框，表示组件类只用于Out-of-Process类型的自动化服务器。

如选中“Licensed”复选框(一般只适用于ActiveX控件)，表示在设计期或运行期都需要许可，组件类的实例只能由IClassFactory2创建。

如选中“Predefined”复选框，表示客户程序应当自动创建OLE对象的单个实例。

如选中“Control”复选框，表示此组件类描述的是一个OLE控件。

如选中“Aggregatable”复选框，表示组件类的成员可以被聚合。

如选中“Replaceable”复选框，表示此组件类支持IConnectionPointWithDefault接口。

“Implements”页用于增加或删除构成组件类的接口，如图4.19所示。

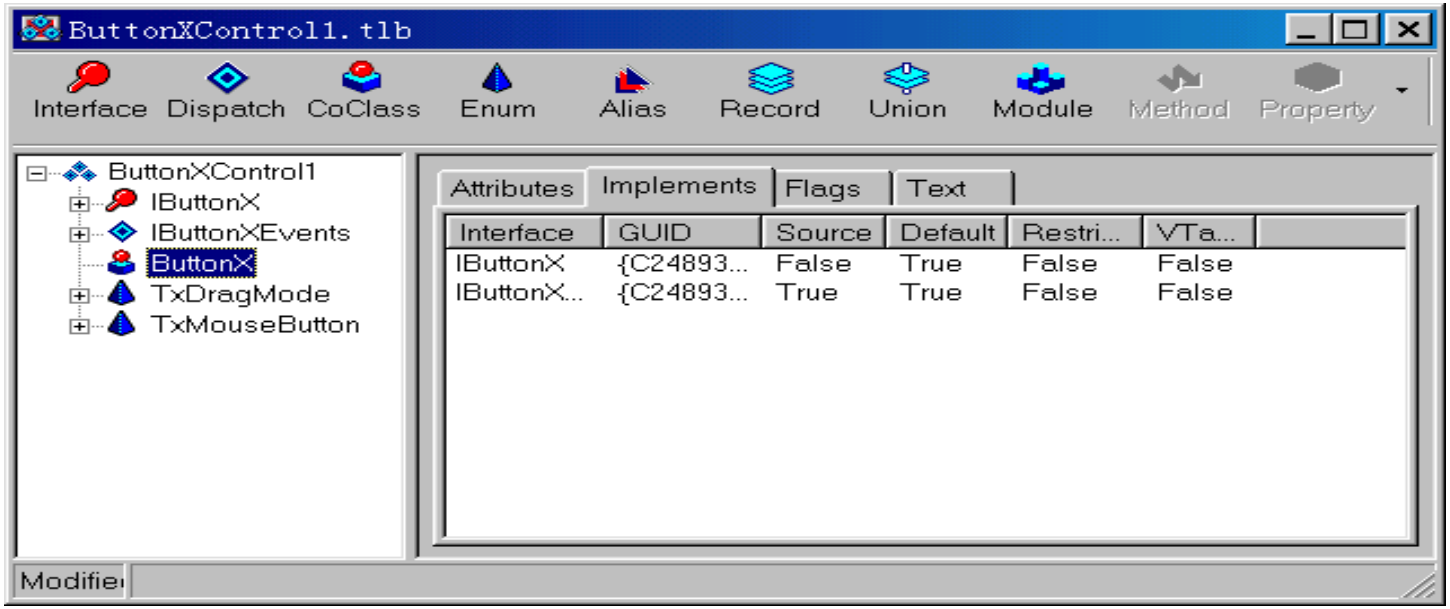


图 4.19　组件类的成员

“Interface”栏是组件类要实现的接口名称。“GUID”栏是接口的全局识别号。“Source”栏为True表示这个接口是事件的源。“Default”栏为True表示这个接口是组件类的默认接口，一个组件类最多有两个默认接口，其中一个是源接口，另一个是槽接口。“Restricted”栏为True表示这个接口不能被宏编程环境使用，一个接口的“Default”栏和“Restricted”栏不能同时为True。“VTable”栏为True表示支持虚拟方法表调用。

4.10 把类型库引入到当前项目中

您可以使用“Project”菜单上的“Import Type Library”命令把已注册的类型库(包括其它工具生成的类型库)引入到当前项目中，并且自动生成相应的C++源文件，也就是说，“Import Type Library”命令能够把二进制的类型库裹上一层C++的外套。

使用“Project”菜单上的“Import Type Library”命令将打开“Import Type Library”对话框，如图4.20所示。

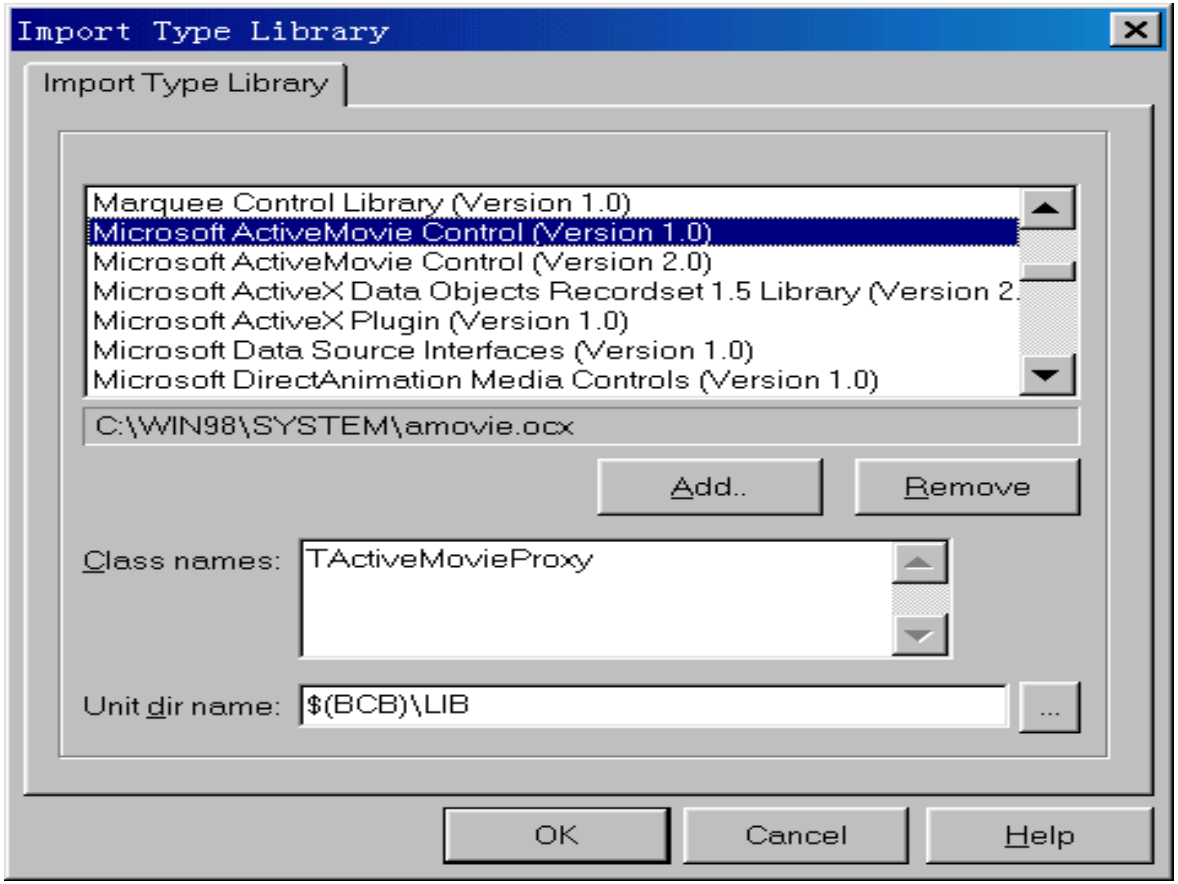


图 4.20　引入类型库

这个对话框列出了所有已注册的类型库，以及每个类型库所在的路径和文件名，“Class Names”框显示所选类型库中的控件的类名。

C++ Builder 3能够从引入的类型库中提取出有关的信息，并自动生成相应的C++源文件，该源文件默认的位置在C:\Program Files\Borland\CBuilder 3\LIB目录，您可以单击“Unit Dir Name”框右边的省略号按钮重新指定一个位

置。
要注册新的类型库，单击“Add”按钮，C++ Builder 3将显示“Open Type Library”对话框，选择一个要注册的类型库文件，然后单击“打开”按钮，C++ Builder 3就自动注册该类型库，并加到“Import Type Library”对话框中。
如果您在“Open Type Library”对话框中选择的类型库已注册，C++ Builder 3将在Windows的注册表中重复注册所选的类型库。
要从注册表中撤消对某类型库的注册，单击“Remove”按钮。

4.11 刷新、保存、注册和发布类型库

“Type Library”编辑器是个双向的工具，您既可以用工具栏上的按钮或快捷菜单进行操作，也可以按IDL文法直接键入有关信息，这两种操作方式是等价的。举例来说，假设您在对象列表窗格中选择了一个节点叫Interface1，在右框中翻到“Text”页直接键入特性和方法的声明，也就是说，Interface1有了新的成员，为了直观地显示出该节点有了新成员，您应当刷新类型库。
要刷新类型库，单击工具栏上的“Refresh”按钮，C++ Builder 3将首先对“Type Library”编辑器中的信息进行文法检查，如检查通过，就刷新对象列表窗格中的树状结构，并在内存中刷新相应的C++源文件但不存盘。
要注意的是，如果您把类型库中的某项内容换了名或删除了，刷新类型库时可能会出现重复的或多余的代码，此时，需要您手工把多余的代码删掉。
要保存类型库信息，使用“File”菜单上的“Save”命令，C++ Builder 3首

先对“Type Library”编辑器中的信息进行文法检查，然后把类型库保存到一个扩展名是.TLB的文件中，同时还把类型库中的接口、数据类型以及类的声明保存到一个.CPP文件中。您不要直接修改这个.CPP文件，因为这个文件是由“Type Library”编辑器自动维护的，即使您修改了这个文件，当您保存或刷新类型库信息时，您的修改将作废。

要注册类型库信息，单击工具栏上的“Rigister”按钮，C++ Builder 3首先对“Type Library”编辑器中的信息进行文法检查，然后更新Windows的注册表。

当您用C++ Builder 3提供的向导创建一个ActiveX控件时，向导将自动创建一个类型库，编译器将把类型库作为资源连接到ActiveX控件的.OCX文件中。

要发布类型库可以有两种形式，一是作为单独的资源文件发布，扩展名是.TLB，这是个二进制文件。还有一种方式就是随.OCX文件发布，要把类型库加到.OCX中，您得事先建立一个.RC文件，把类型库列出来，形式如下：

```
1 TYPELIB MyLib1.tlb
2 TYPELIB MyLib2.tlb
```

然后用资源编译器编译这个.RC文件。一个.OCX中可以有多个类型库。

第五章　创建ActiveX控件

C++ Builder 3的一个显著特点就是ActiveX技术。要把ActiveX讲清楚恐怕不是一件很容易的事情，您可以初步把它理解为消肿了的OCX，尤其适合于在Internet上传输。第三章介绍了C++ Builder 3的ActiveX框架，下面就具体介绍怎样用C++ Builder 3创建ActiveX控件、ActiveForm以及怎样在Web上发布ActiveX控件和ActiveForm。

5.1　创建和使用 ActiveX 控件

运用C++ Builder 3的ActiveX框架，创建ActiveX控件实际上就是把已有的基于TWinControl的控件转换为ActiveX控件，如果是基于TGraphicControl的控件，一般不能直接转换，您必须先改写该控件，把它的基类改为TCustomControl，因为TCustomControl兼具TWinControl和TGraphicControl的特征，然后把它重新安装到元件选项板上。

C++ Builder 3为创建ActiveX控件提供了向导。使用“File”菜单上的“New”命令打开“New Items”对话框(实际上就是C++ Builder 3的对象库)，翻到“ActiveX”页，双击“ActiveX Control”图标打开ActiveX控件向导，如图5.1所示。

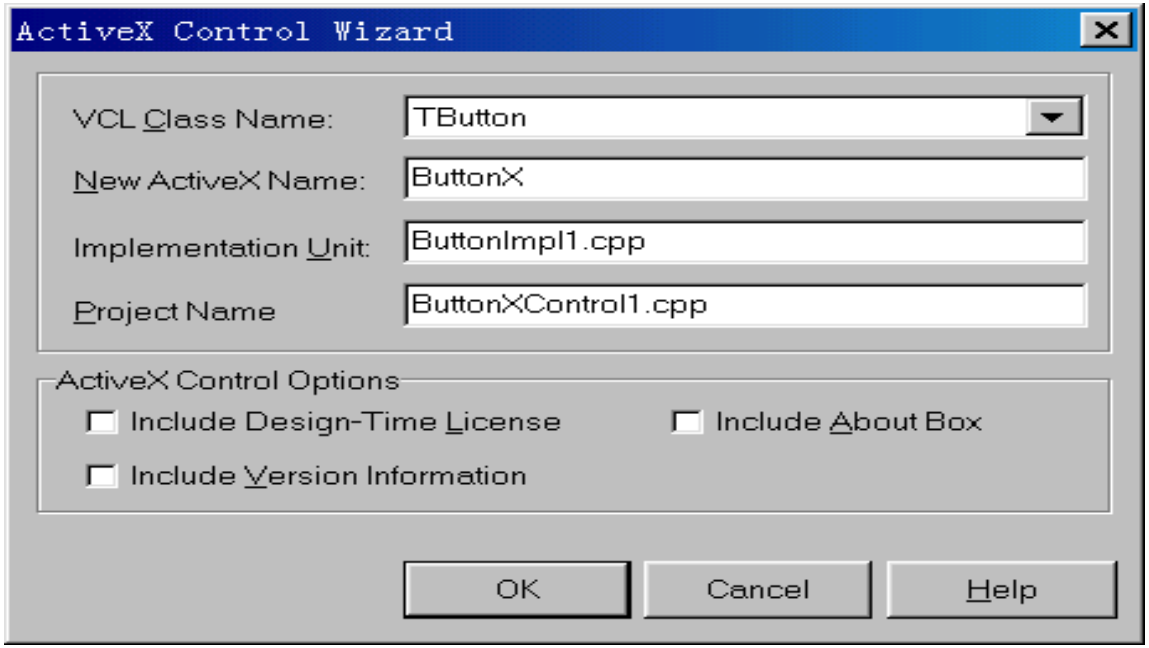


图 5.1　ActiveX 控件向导

在“VCL Class Name”框内选择元件选项板上的一个VCL控件，这里以TButton为例，以便读者与前面介绍的“Type Library”编辑器对照。C++ Builder 3根据您所选的VCL控件自动生成ActiveX控件的名称和实现接口的单元的名称。

ActiveX控件的名称在“New ActiveX Name”框内显示，通常最后一个字母为大写的X，这里是ButtonX。ActiveX控件的实现单元名称在“Implementation Unit”框内显示，这里是ButtonImpl1.CPP。“Project Name”框是此ActiveX控件的项目名称，这里是ButtonXControl1，最后编译此项目得到的.OCX文件就是ButtonXControl1.OCX。

在“ActiveX Control Options”框内有三个复选框，如果选中“Include Design-Time License”复选框，表示在建立此ActiveX控件时将加上许可文件，许可文件的扩展名是.LIC，文件名就是项目名，在这里就是ButtonXControl1.LIC。除非用户拥有此许可文件的合法拷贝，否则，用户只能作为最终用户使用它，无法在开发环境中打开它。

如果一个ActiveX项目中有几个ActiveX控件，并且这几个ActiveX控件都选中了“Include Design-Time Licence”复选框，那么，许可文件仍然只有一个，但每个ActiveX控件在许可文件中都有专门的一个入口。

如果选中“Include Version Information”复选框，表示要在ActiveX控件中包含版本信息。要给ActiveX控件加上版本信息，使用“Project”菜单上的“Options”命令，翻到“VersionInfo”页，如图5.2所示。

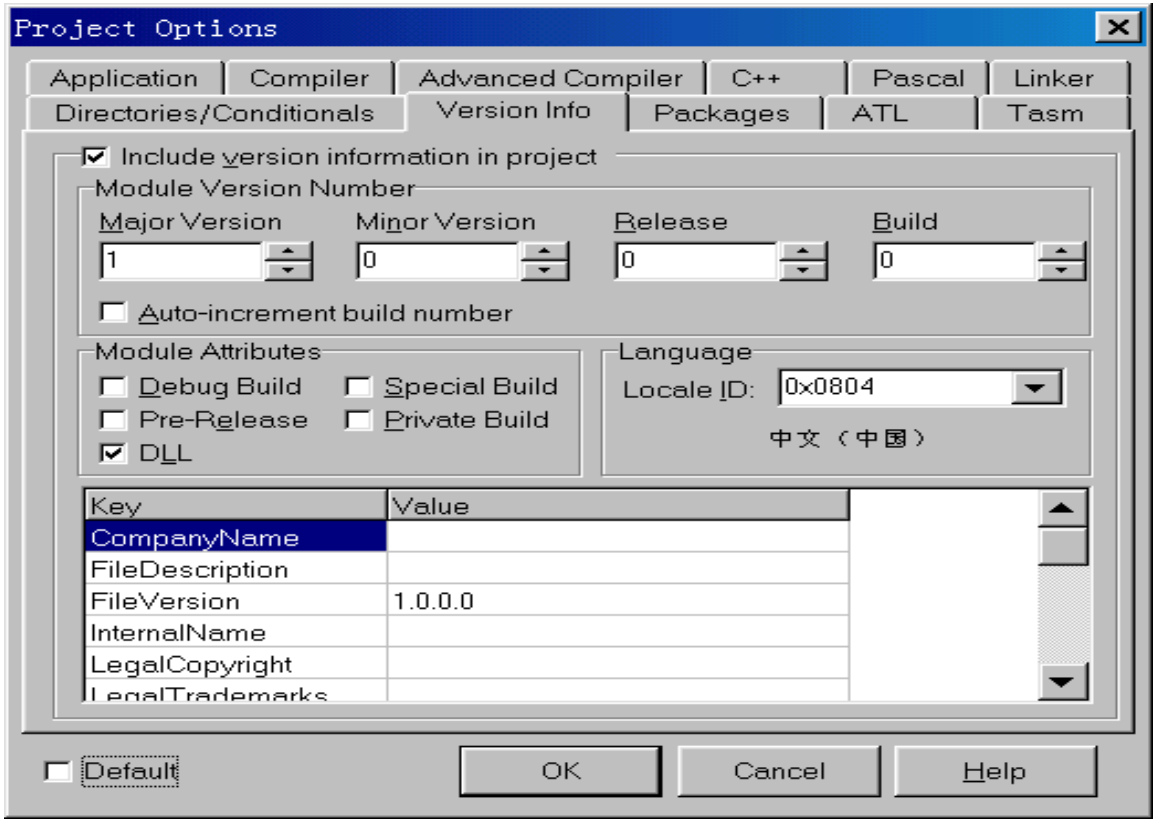


图 5.2　给 ActiveX 控件加上版本信息

如果选中“Include About Box”复选框，表示ActiveX控件中将包含About框。About框实际上就是一个Form，Form上默认的内容有ActiveX控件的名称、ActiveX控件的图标、版权信息以及一个OK按钮，如图5.3所示。您可以自定义About框上的内容。

图 5.3　About 框

上述选项设置好以后，单击OK按钮，ActiveX控件向导可能会显示这样的对话框：

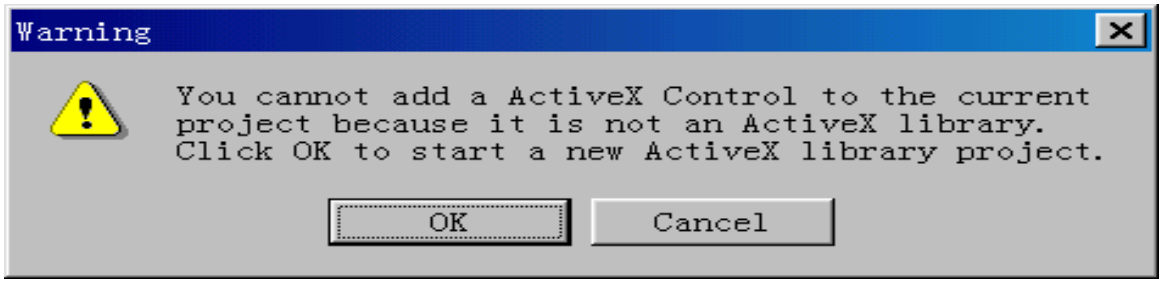


图 5.4　一个警告信息

这个警告的意思是，您必须先建立一个ActiveX项目，然后才能创建ActiveX控件。好在ActiveX控件向导简化了有关工作，您只需单击OK按钮就可以了。当然，您也可以事先在对象库中双击“ActiveX Library”图标建立一个ActiveX项目，然后再用ActiveX控件向导创建ActiveX控件，此时不会出现图5.4所示的警告框。

您还可以把多个ActiveX控件或ActiveForm加到同一个ActiveX项目中。

5.2 向导创建了哪些文件

如果打开资源管理器的话，您就会发现ActiveX控件向导主要生成了下列文件：

- ActiveX项目文件：ButtonXControl1.CPP
- 类型库文件：ButtonXControl1.TLB
- ActiveX控件中的接口实现单元文件：ButtonImpl1.CPP
- 类型库的接口源文件：ButtonXControl1_TLB.CPP
- About框的源文件(可选)：About1.CPP
- About框的Form文件(可选)：About1.DFM
- 许可文件(可选)：ButtonXControl1.LIC
- ActiveX项目的资源文件：ButtonXControl1.RES
- ActiveX项目的ATL文件：ButtonXControl1_ATL.CPP

其中，ATL是Microsoft Active Template Library的简称。

项目文件是ActiveX项目向导自动生成并维护的，一般不要手工修改它。

类型库文件ButtonXControl1.TLB是个二进制文件，只能用“Type Library”编辑器打开，代码编辑器能打开的是类型库的接口源文件ButtonXControl1_TLB.CPP，这个文件是类型库的C++描述，不要直接修改它，因为它是由“Type Library”编辑器维护的，即使您修改了这个文件，刷新类型库时仍会忽略修改。

纯粹从使用的角度看，没有必要知道接口内部的情况，这里是从ActiveX程序员的角度，看看ButtonXControl1中的特性、方法和事件是怎样实现的。

为了节省篇幅，这里选取了有代表性的部分代码。

```
STDMETHODIMP TButtonXImpl::_set_Font(IFontDisp* Value)  // 设置Font特性
{
try
{
      const DISPID dispid = -512;
      if (FireOnRequestEdit(dispid) = = S_FALSE) return S_FALSE;
      SetOleFont(m_VclCtl->Font, Value);
      FireOnChanged(dispid);
}
catch(Exception &e)
{
      return Error(e.Message.c_str( ), IID_IButtonX);
}
return S_OK;
};

STDMETHODIMP TButtonXImpl::get_Font(IFontDisp** Value)  // 读Font特性
{
try
{
```

```
        _di_IFontDisp Temp;
        GetOleFont(m_VclCtl->Font, Temp);
        Temp->AddRef( );
        *Value = Temp;
}
catch(Exception &e)
{
        return Error(e.Message.c_str( ), IID_IButtonX);
}
return S_OK;
};

STDMETHODIMP TButtonXImpl::AboutBox( ) //About函 数
{
try
{
        ShowButtonXAbout( );
}
catch(Exception &e)
{
        return Error(e.Message.c_str( ), IID_IButtonX);
}
```

```
return S_OK;
};
```

至于ShowButtonXAbout函数，则是在About框的源文件中定义的：

```
void ShowButtonXAbout(void)
{
TButtonXAbout* Form;
Form = new TButtonXAbout(NULL);
try
{
     Form->ShowModal( );
}
catch(...)
{
     Form->Free( );
     return;
}
Form->Free( );
}
```

ShowButtonXAbout函数的作用是创建TButtonXAbout即About框的实例，并作为模式对话框显示。

5.3 编辑类型库

ActiveX控件向导能够自动创建类型信息，但并没有把VCL控件中所有公共的(Public)和公开的(Published)的成员都转换到ActiveX控件中，以ButtonXControl为例，TButton元件中的许多特性如Hint、TabOrder就没有加到ButtonXControl中，这可能是因为向导认为没必要，也可能是因为向导不知道怎样转换。

如果您认为有必要把这些遗漏的成员或其它自定义的成员加到ActiveX控件中，可以使用“View”菜单上的“Type Library”命令打开“Type Library”编辑器，如图5.5所示。

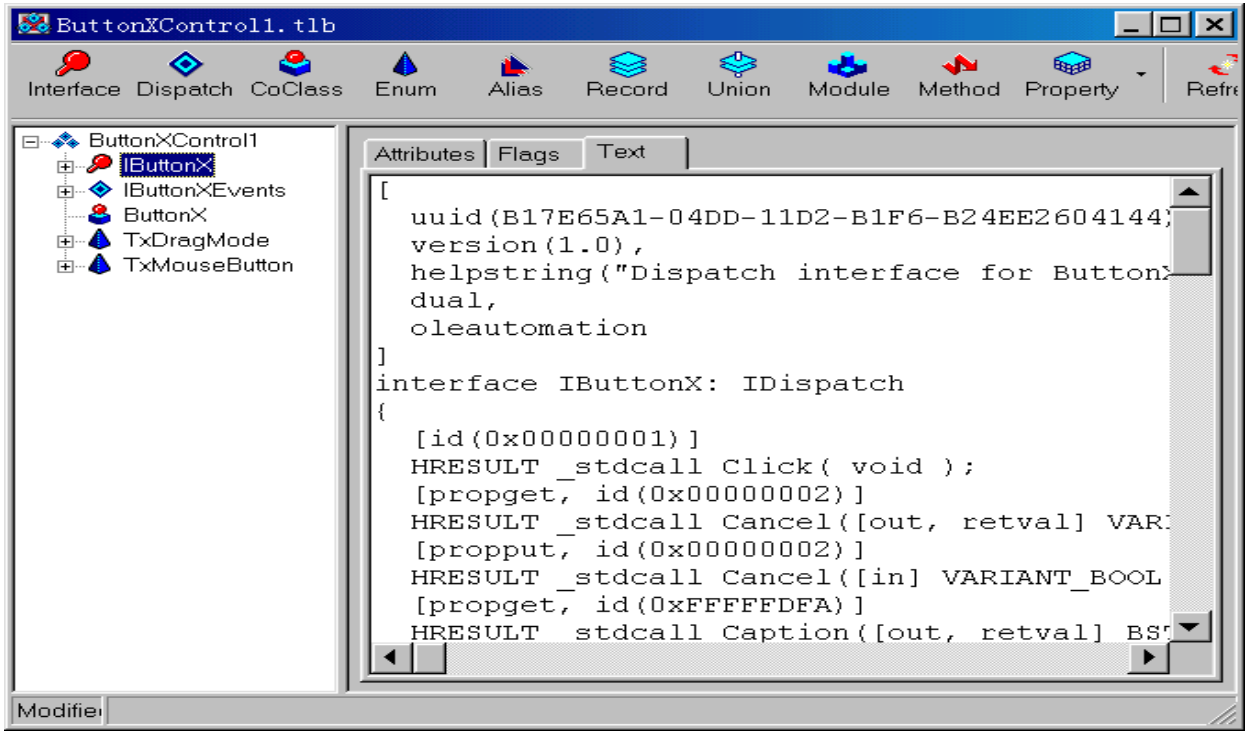


图 5.5 “Type Library”编辑器

要向接口中增加新的成员，在“Type Library”编辑器中选择IButtonX节点或IButtonXEvent节点，然后您可以有三种操作方式：

一是单击工具栏上的“Property”或“Method”图标。

二是在上述两个节点上单击鼠标右键，在弹出的菜单中选择“New”，再选择“Property”或“Method”。

三是翻到“Text”页，按IDL的文法直接键入特性或方法的声明。

无论是哪种方式，操作好后最好单击工具栏上的“Refresh”图标刷新一下类型库。

5.4 创建特性页

特性页非常类似于一个对话框，主要用于让ActiveX控件的用户在设计期修改该控件的特性。要打开特性页，一般是在ActiveX控件上单击鼠标右键，在弹出的菜单中选择“Properties”命令。例如，TNMHTML控件的特性页是这样的：

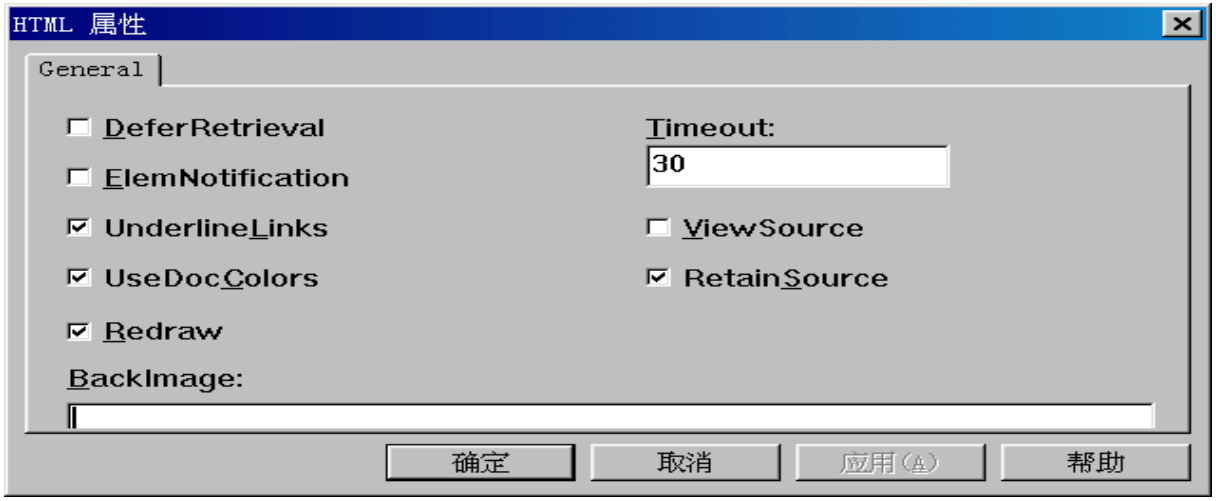


图 5.6 一个 ActiveX 控件的特性页

下面还是以ButtonXControl为例，介绍创建特性页的一般步骤：

使用“File”菜单上的“New”命令，翻到“ActiveX”页，双击“Property Page”图标，向导将创建一个空白的特性页PropertyPage1、一个对应的单元文件Unit1，代码如下：

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include <atl\atlvcl.h>
#include <initguid.h>
#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TPropertyPage1 *PropertyPage1;
//---------------------------------------------------------------------------
_fastcall TPropertyPage1::TPropertyPage1(Classes::TComponent *Component)
: TPropertyPage(Component)
{
//这 是 特 性 页 的 构 造 函 数
}
//---------------------------------------------------------------------------
void _fastcall TPropertyPage1::UpdatePropertyPage(void)
{
//在 这 儿 写 代 码 ， 从 ActiveX控 件 中 把 特 性 值 读 到 特 性 页 上
}
//---------------------------------------------------------------------------
```

```
void _fastcall TPropertyPage1::UpdateObject(void)
{
//在这儿写代码，把特性页上的值写到ActiveX控件中
}
```

新创建的特性页是空白的，您应当根据要编辑的特性把合适的VCL控件加到Form上，例如，对于字符串类型的特性，一般用TEdit元件来编辑，对于字符串列表类型的特性，一般用TMemo元件来编辑。有的特性还需要用多个VCL控件配合起来编辑。

特性页上一般放四个按钮：OK(确定)、Cancel(取消)、Apply(应用)、Help(帮助)。

特性页往往做成多页结构，这样，如果ActiveX控件的特性比较多，通过适当的分类，就显得有条理性，也能避免特性页在屏幕上占的空间太大。

特性页上的每一个VCL控件旁边最好用TLabel做一个标注，否则，用户就不知道这个VCL控件编辑的是什么特性。标注的文字最好就是要编辑的特性名称。

当然，仅仅在VCL控件旁边标注一下是不够的，在特性页设计好后，您必须把ActiveX控件的特性与特性页上的VCL控件联系起来，这就要在特性页单元中重载TPropertyPage的UpdatePropertyPage和UpdateObject，其中，UpdatePropertyPage用于从ActiveX控件中读特性值到特性页上，而UpdateObject正相反，用于把特性页上的值写到ActiveX控件中。

下面以ButtonXControl为例，介绍怎样重载UpdateObject和UpdatePropertyPage，假设特性页上用Edit1编辑ButtonXControl的Caption特

性：

```
//---------------------------------------------------------------------------
void _fastcall TPropertyPage1::UpdatePropertyPage(void)
{
Edit1->Text = OleObject->Caption
}
//---------------------------------------------------------------------------
void _fastcall TPropertyPage1::UpdateObject(void)
{
OleObject->Caption = Edit1->Text;
}
```

有的读者可能对OleObject比较陌生，这个特性属于TPropertyPage，用于返回特性页所编辑的ActiveX控件，这样您就可以访问ActiveX控件中的特性和方法。

最后，要把特性页与ActiveX控件联系起来，这就要在ActiveX控件的实现单元中的BEGIN_PROPERTY_MAP和END_PROPERTY_MAP之间加一个PROP_PAGE宏，并且传递特性页的GUID作为参数。特性页的GUID是由C++ Builder 3自动分配给特性页的，您可以在特性页的单元文件中找到它的GUID，如Class_PropertyPage1、Class_PropertyPage2。

```
BEGIN_PROPERTY_MAP(TActiveFormXImpl)
PROP_PAGE(CLSID_PropertyPage1)
    END_PROPERTY_MAP( )
```

其中，Class_PropertyPage1就是要显示的特性页的GUID，如果一个ActiveX控件有几个特性页，就要分别加入几个宏和几个GUID，例如：

```
BEGIN_PROPERTY_MAP(TActiveFormXImpl)
PROP_PAGE(CLSID_PropertyPage1)
PROP_PAGE(CLSID_PropertyPage2)
END_PROPERTY_MAP( )
```

C++ Builder 3提供了四种标准的特性页：Class_DColorPropPage、Class_DfontProp-Page、Class_DPicturePropPage、Class_DStringPropPage，这四个标准的特性页分别用于编辑颜色、字体、图像和字符串列表。

5.5 注册和安装ActiveX控件

ActiveX控件只有在注册后才能被使用。要注册ActiveX控件，使用“Run”菜单上的“Register ActiveX Server”命令，如果此时ActiveX控件的项目还没有编译或需要重新编译，就先编译此ActiveX项目。该命令实际上是调用ActiveX控件输出的DllRegisterServer函数，这个函数是在ActiveX控件的项目文件中输出的。

如果ActiveX控件注册成功，C++ Builder 3将显示一个信息框，如图5.7所示：

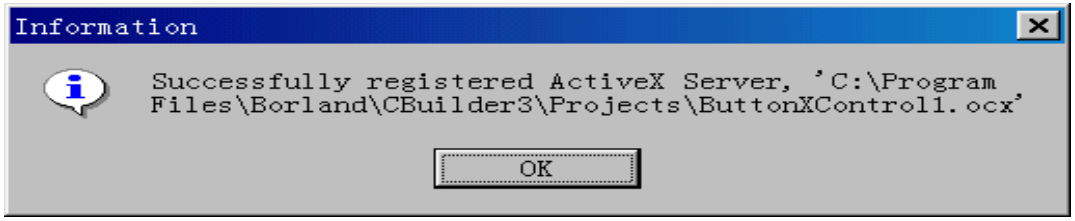


图 5.7　ActiveX 控件已注册成功

要取消ActiveX控件的注册，使用“Run”菜单上的“Unregister ActiveX Server”命令。

要在C++ Builder 3的IDE中使用ActiveX控件，还要把ActiveX控件安装到元件选项板上，以后就可以像使用VCL元件一样使用ActiveX控件。注意：这里所说的ActiveX控件已不限于用C++ Builder 3创建的ActiveX控件，也可以是众多其它厂家创建的ActiveX控件，由于ActiveX控件具有语言无关性，因此，ActiveX控件可以在各种编程环境共享。

要安装ActiveX控件，首先要创建一个包，然后把ActiveX控件加到这个包中。当然，您也可以把ActiveX控件加到元件选项板已有的页上，这种情况下不需要创建一个包。

如果您要把ActiveX控件安装到元件选项板已有的页上，您可以使用“Component”菜单上的“Import ActiveX Control”命令打开“Import ActiveX”对话框，如图5.8所示。

选择一个要安装的ActiveX控件，比如“BottonXControl Library(Version 1.0)”，下面的“Class Names”框显示该ActiveX控件的类名，一般不需要修改，除非与其它ActiveX控件重名。“Palette Page”框用于指定ActiveX控件要安装到元件选项板的哪一页上，一般选“ActiveX”，当然，也可以选

其它页如“Standard”，甚至键入一个不存在的页，C++ Builder 3将在元件选项板上自动增加这一页。

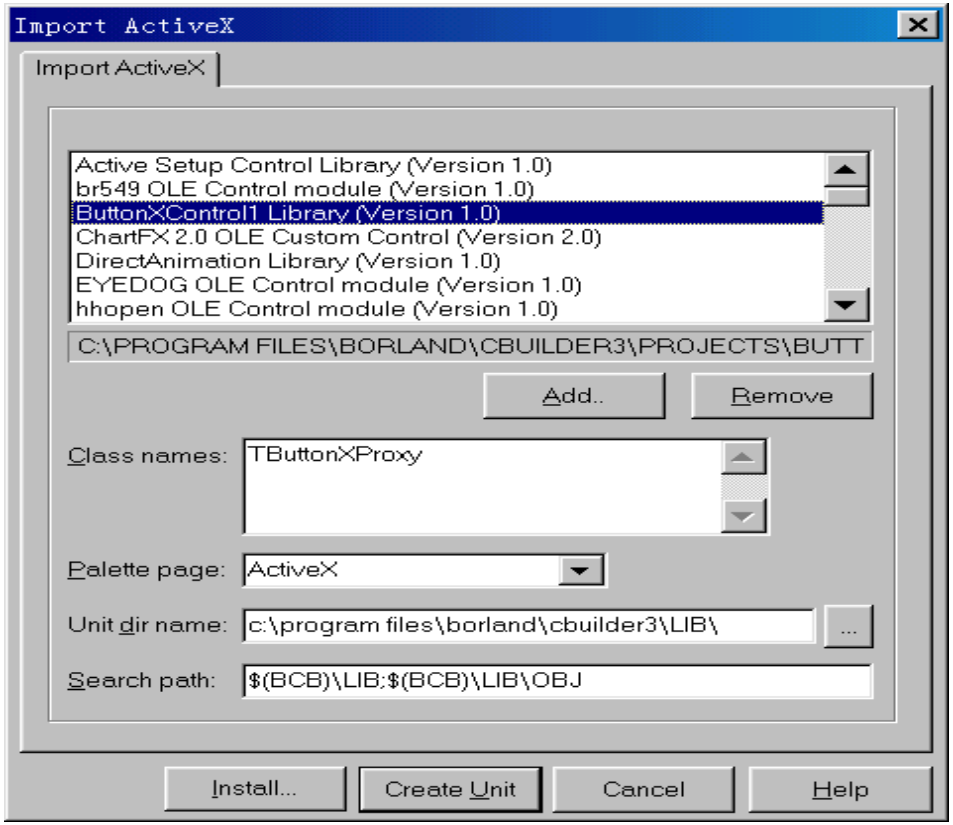


图 5.8　引入 ActiveX 控件

“Unit File Name”框是该ActiveX控件的外套文件(包括完整的路径),“Search Path”框是编译器的搜索路径。一般来说，“Class Names”框、“Palette Page”框、“Unit File Name”框以及“Search Path”框的内容都不需要修改。

单击“Install”按钮，C++ Builder 3将打开“Install”对话框，如图5.9所示。

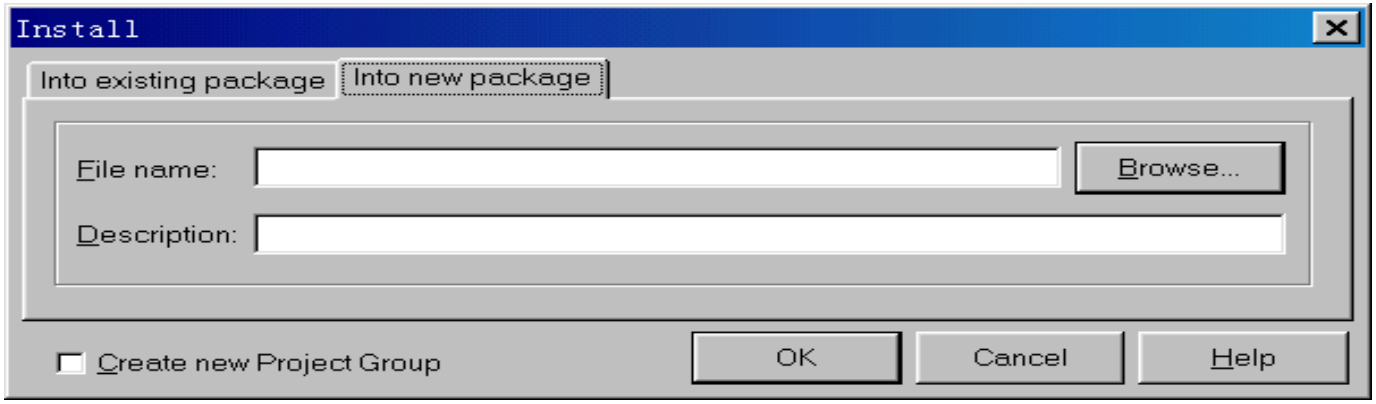


图 5.9　指定要把 ActiveX 控件加到哪一个包中

您既可以把ActiveX加到已有的包中，也可以加到一个新的包中，如果要把ActiveX加到一个新的包中，翻到“Into New Package”页，在“File Name”框内键入包的文件名，在“Description”框内键入关于包的简短描述(可选)，然后单击OK按钮。

“Import ActiveX”对话框列出了所有已注册的ActiveX控件，如果要安装的ActiveX控件还没有注册，您可以单击“Add”按钮打开“Register OLE Control”对话框，选择要注册的ActiveX控件(.OCX)，如图5.10所示。

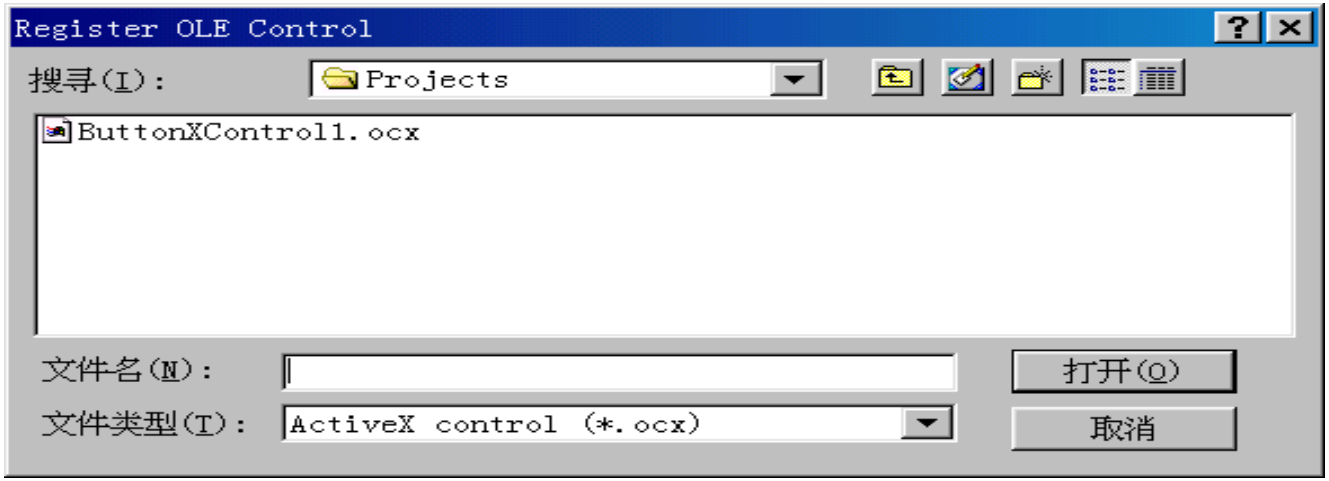


图 5.10　注册新的 ActiveX 控件

5.6　怎样使用 ActiveX 控件

注册和安装了ActiveX控件后，您可以像使用VCL元件那样使用ActiveX控件，包括用Object Inspector修改它的特性，在运行期调用它的方法或响应它的事件。

如果想知道ActiveX控件有哪些方法，每个方法有哪些参数，您可以查阅该ActiveX控件的帮助文件，或者通过“Type Library”编辑器显示它的类型信息。

后面在介绍OLE自动化时将提到，调用自动化对象的方法时可以省略掉部分参数。实际上，某些ActiveX控件也允许省略掉部分参数，但省略的语法

不同。调用ActiveX控件的方法时参数个数不能少，不过，您可以用值为Unassigned或VT_EMPTY的Variant类型的变量去代替您不感兴趣的参数，程序示例如下：

```
Variant V = Unassigned;
ActiveXControl->Method(clRed, V, V, V, 100);
```

这里顺便介绍一下怎样调试ActiveX控件，由于ActiveX控件不是一个可以单独执行的程序，您只能象调试动态链接库那样，先使用“Run”菜单上的“Parameters”命令打开“Run Parameters”对话框，在“Host Application”框内键入一个客户程序，然后使用“Run”菜单上的“Run”命令运行客户程序，在客户程序中操作ActiveX控件，如果您事先在ActiveX控件的实现单元中设置了断点，您就可以通过单步或跟踪等手段调试ActiveX控件。

5.7 ActiveForm

什么是ActiveForm？从本质上讲，ActiveForm也是ActiveX控件，不同的是，一般的ActiveX控件是单一的控件，所谓单一是指它只能从一个VCL控件转换而来。而ActiveForm可以把一个或多个VCL控件转换为一个复合的ActiveX控件。

这项技术为什么取名为ActiveForm？因为需要有一个类似于Form的容器来放这些VCL控件。注意；ActiveForm并不是这些VCL控件的大杂烩，它只输出它自己的特性、方法和事件，而ActiveForm上的VCL控件只是在ActiveForm

内部使用。即使要用到某个VCL控件的特性、方法和事件，也只能在ActiveForm的名义下输出。

ActiveForm上一般放可视的元件，也可以放非可视的元件，如TTable、TQuery等，这样就能用强大的数据库引擎BDE结合C++ Builder 3的Data Broker技术实现分布式数据集。

当ActiveForm在Web上发布的时候，C++ Builder 3会自动生成一个HTML页面包含对该ActiveForm的引用，用户可以通过WWW浏览器如Microsoft Internet Explorer或Netscape Navigator来显示和运行这个ActiveForm。

要创建ActiveForm，其过程与创建ActiveX控件非常相似，为了清晰起见，我们不厌其烦把有关步骤完整地列出来：

首先使用“File”菜单上的“New”命令打开“New Items”对话框，翻到“ActiveX”页，双击“ActiveForm”图标打开ActiveForm向导，如图5.11所示。

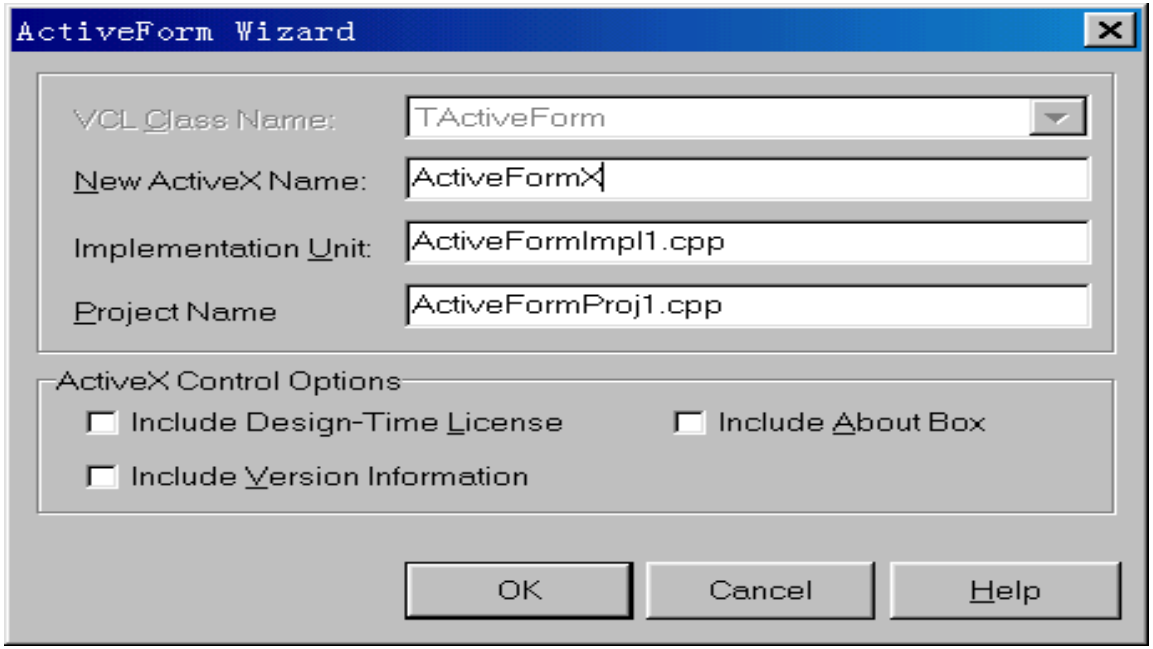


图 5.11　ActiveForm 向导

“New ActiveX Name”框内显示的是ActiveForm的默认名称，“Implementation Unit”框内显示的是ActiveForm的实现单元的默认名称，“Project Name”框内显示的是ActiveForm的项目名称，如果有必要的话，您可以修改上述三个名称。由于ActiveForm只能基于TActiveForm，因此，“VCL Class Name”框是变灰的。

与ActiveX控件一样，如果选中“Include Design-Time License”复选框，表示在建立ActiveForm时将加上许可文件(扩展名是.LIC)，除非用户拥有许可文件的合法拷贝，否则，用户只能作为最终用户使用它，无法在开发环境中打开它。如果选中“Include About Box”复选框，表示ActiveForm中将包

含About框。如果选中“Include Version Information”复选框，表示要在ActiveForm中包含版本信息。单击OK按钮，向导将自动创建类型信息。
下一步就象设计一个单Form的应用程序完全一样，从元件选项板上选择一个或多个、可视或非可视的元件加到ActiveForm上。这里再强调一下，ActiveForm上的元件只是ActiveForm内部使用的，如果您要把ActiveForm上某个元件的特性、方法和事件对外输出，您只能在ActiveForm的名义下输出，这意味着，您必须在ActiveForm的接口中增加新的成员，然后通过编程间接地访问ActiveForm上的元件，也就是说，要给ActiveForm上的元件的特性、方法或事件加上一层外套。此外，您还可能要输出其它自定义的特性、方法和事件，这时候，您就必须编辑ActiveForm的类型库。
要打开ActiveForm的类型库，使用“View”菜单上的“Type Library”命令打开“Type Library”编辑器，然后按照前面介绍的方法加入新的成员。举例来说，假设ActiveForm上有一个编辑框Edit1，要输出编辑框的内容即Text特性，您必须在ActiveForm中增加一个诸如MyText的特性，然后您得在ActiveForm的实现单元中定义下面两个方法：

```
//-----------------------------------------------------------------------
STDMETHODIMP TActiveFormXImpl::set_ MyText(VARIANT_BOOL Value)
{
try
{
      const DISPID dispid = 1;
      if (FireOnRequestEdit(dispid) = = S_FALSE) return S_FALSE;
```

```
        m_VclCtl-> MyText = Value;
        FireOnChanged(dispid);
}
catch(Exception &e)
{
        return Error(e.Message.c_str( ), IID_IActiveFormX);
}
return S_OK;
};
//-----------------------------------------------------------------------
STDMETHODIMP TActiveFormXImpl::get_ MyText (VARIANT_BOOL* Value)
{
try
{
        *Value = m_VclCtl-> MyText;
}
catch(Exception &e)
{
        return Error(e.Message.c_str( ), IID_IActiveFormX);
}
return S_OK;
};
```

有必要的话，还要创建ActiveForm的特性页，这是为了在某些编程环境中可以在设计期修改ActiveForm的特性，最后就是注册ActiveForm，如果要在C++Builder 3的环境中使用的话，需要把它安装到元件选项板上，并把它作为VCL控件一样使用。关于创建特性页、注册和安装ActiveForm的步骤与ActiveX控件是一样的。

5.8 在Web上发布ActiveX

ActiveX控件或ActiveForm设计好后，您可能要通过Web服务器在Internet上发布这个ActiveX控件或ActiveForm。为此，您首先要正确设置有关Web发布的选项。

要设置有关Web发布的选项，使用“Project”菜单上的“Web Deployment Options”命令打开“Web Deployment Options”对话框，如图5.12所示。

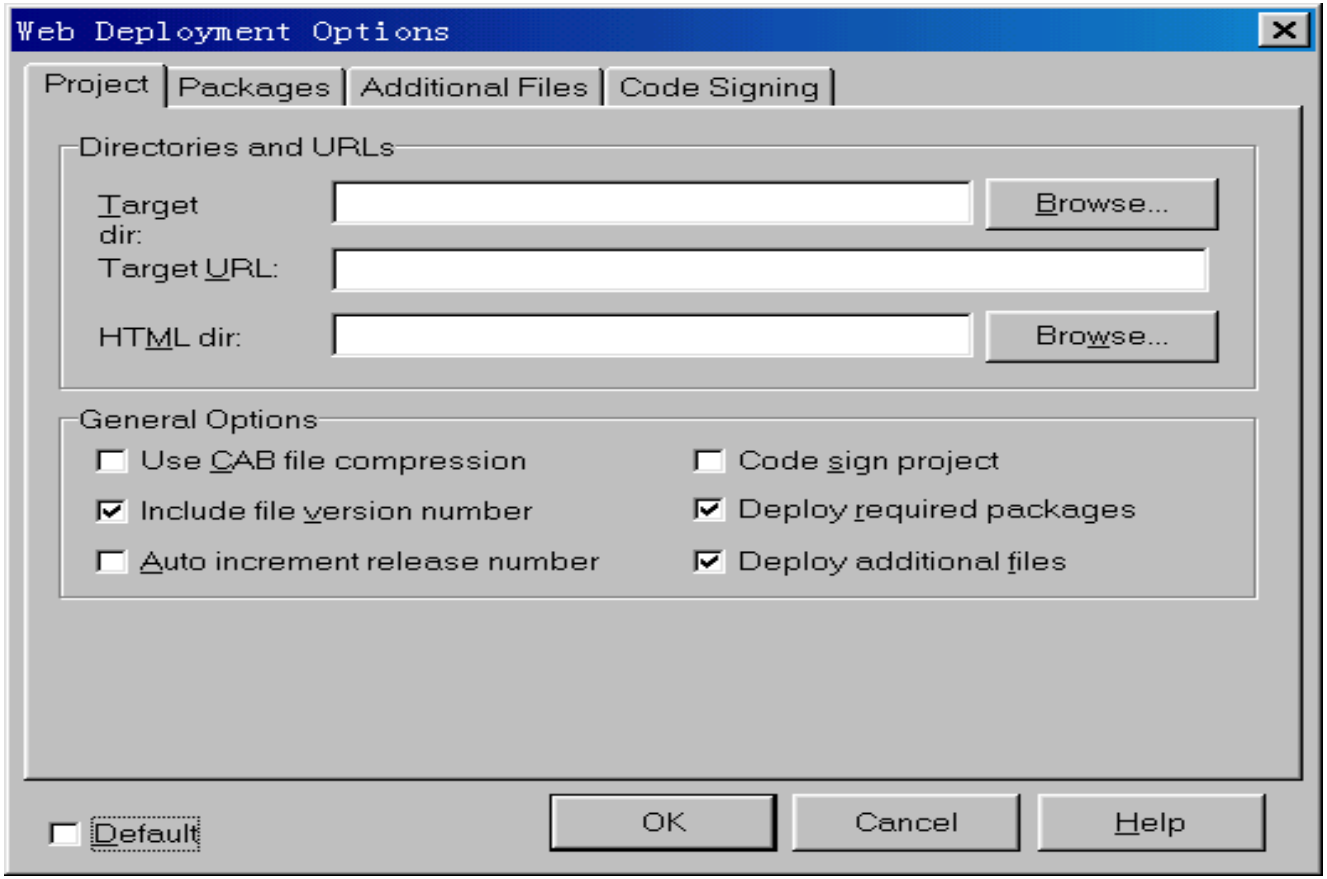


图 5.12　设置 Web 发布的选项

5.8.1　Project页

“Target Dir”框用于指定要发布的ActiveX控件或ActiveForm的.OCX文件在Web服务器上的路径，可以是常规的路径如C:\XXH\MyActiveX\，也可以是通用命名约定UNC如\\MyServer\OCX_Files。

“Target URL”框用于指定要发布的ActiveX控件或ActiveForm在Web服务器上的URL，不包括.OCX本身，如http://mymachine.borland.com/。
“HTML Dir”框用于指定一个HTML文件的路径，可以是常规的路径或UNC，该HTML文件嵌入了要发布的ActiveX控件或ActiveForm，作测试用。如果没有Web服务器，您可以指定一个本地的URL如file:///c:\MyDir\。
您可以把ActiveX控件或ActiveForm的.OCX文件以及相关的包及其它文件捆绑并压缩在一个文件中，即我们常见的.CAB文件，这样能够显著地减少用户下载的时间。
如选中“Use CAB File Compression”复选框，表示要将OCX及其相关的文件压缩成一个CAB文件。
如选中“Include File Version Number”复选框，表示要包含版本号。
如选中“Autoincrement Release Number”复选框，表示要自动增加项目的发布号。
如选中“Code Sign Project”复选框，表示要在.OCX或.CAB文件中包含签名信息。
如选中“Deploy Required Packages”复选框，表示同时发布OCX需要的包。
如选中“Deploy Additional Files”复选框，表示要同时发布其它相关的文件，这些相关的文件由“Web Deployment Options”对话框的“Additional Files”页指定。

5.8.2 Packages页

此页用于指定要随ActiveX或ActiveForm一起发布的包，如图5.13所示。

“Packages used by this Project”列表框内列出了当前的ActiveX项目中使用的运行期包，您可以对其中每一个包分别设置选项。
如选择“Compress In Separate CAB”，表示包单独压缩成一个.CAB文件。
如选择“Compress In Project CAB”，表示把包和OCX压缩在同一个.CAB文件。
如选中“Use File VersionInfo”复选框，表示要把版本信息加到项目的.INF文件中。如选中“Code Sign File”复选框，表示要在包或.CAB文件中加入签名信息。
“Target URL”框用于指定包在Web服务器上的URL，如果这个框空着，WWW浏览器假定能够在本地找到这个包文件。如果WWW浏览器没有找到这个包文件，下载失败。
“Target Directory”框用于指定包在服务器上的路径，可以是常规路径，也可以是UNC。

5.8.3 Additional Files页

此页用于指定要随OCX一起发布的其它文件。如果有包或其它文件随ActiveX控件或ActiveForm一起发布，C++ Builder 3会自动生成 .INF文件。
这一页上的选项与Packages页几乎完全相同，这里不再赘述。

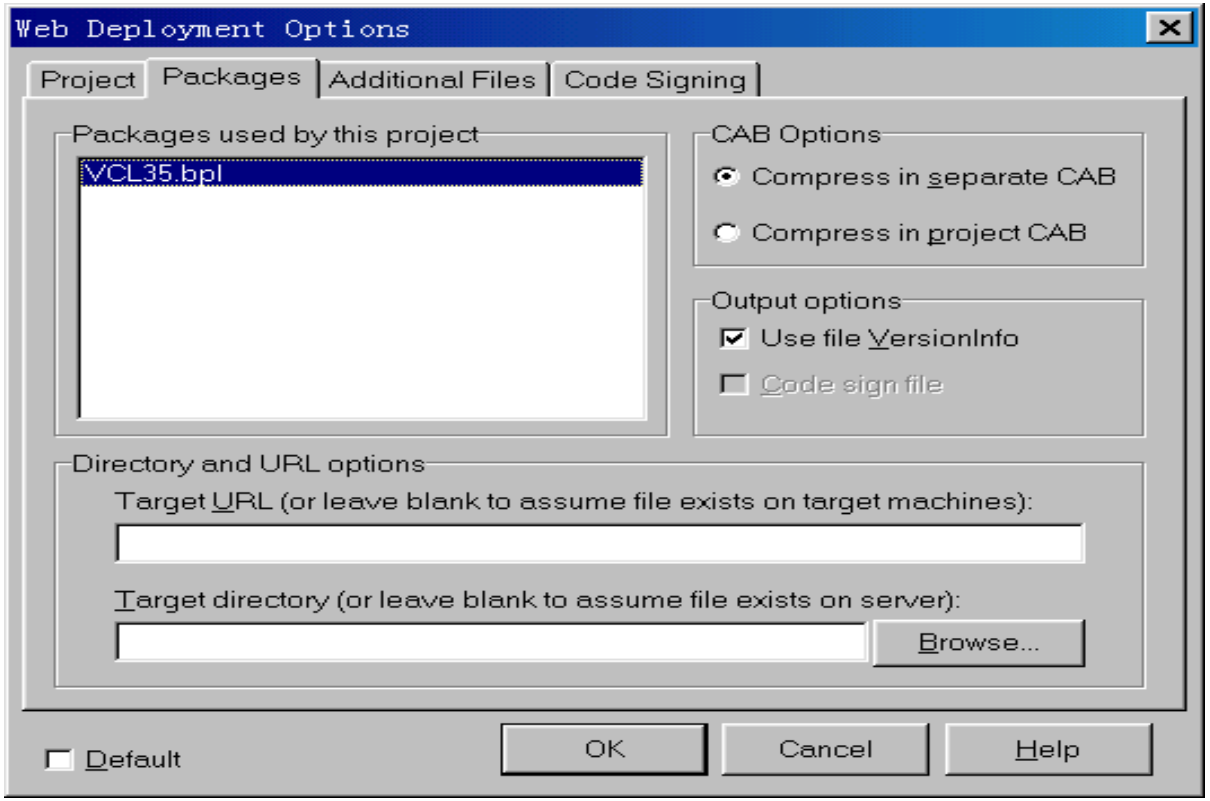


图 5.13　指定要随 ActiveX 一起发布的包

5.8.4　Code Signing页

如果在“Project”页选中了“Code Sign Project”复选框，每个要发布的文件都可以加上一个数字签名，以确认ActiveX控件或ActiveForm的身份，如图5.14所示。

“Credentials file”框用于指定一个证书文件(含路径)，“Private Key”框用于私人密码文件(含路径)，要获得这两种文件，请与Microsoft联系。

“Name of this Application”框用于指定一个应用程序，该应用程序出现在数字证书中。
“Application or company URL”框用于提供一个URL以使Web页面中包含产品或公司的链接。
在“Cryptographic Digest Algorithm”框内，如选择“MD5”表示采用美国RSA实验室的加密算法，生成的密码为128位。目前大多数WWW浏览器都支持这种算法。如选择“SHA 1”表示采用NIST和NSA的加密算法，生成的密码为160位。
设置了有关发布的选项后，就可以在Web上发布ActiveX控件或ActiveForm了。要在Web上发布ActiveX控件或ActiveForm，您可以使用“Project”菜单上的“Web Deploy”命令，这个命令先判断当前的ActiveX项目是否需要重新编译，然后把编译生成的OCX文件和HTML文件分别放到“Target Dir”和“HTML Dir”设定的目录中，供用户通过Web浏览器下载。您也可以自己打开Web浏览器来显示生成的HTML页面。

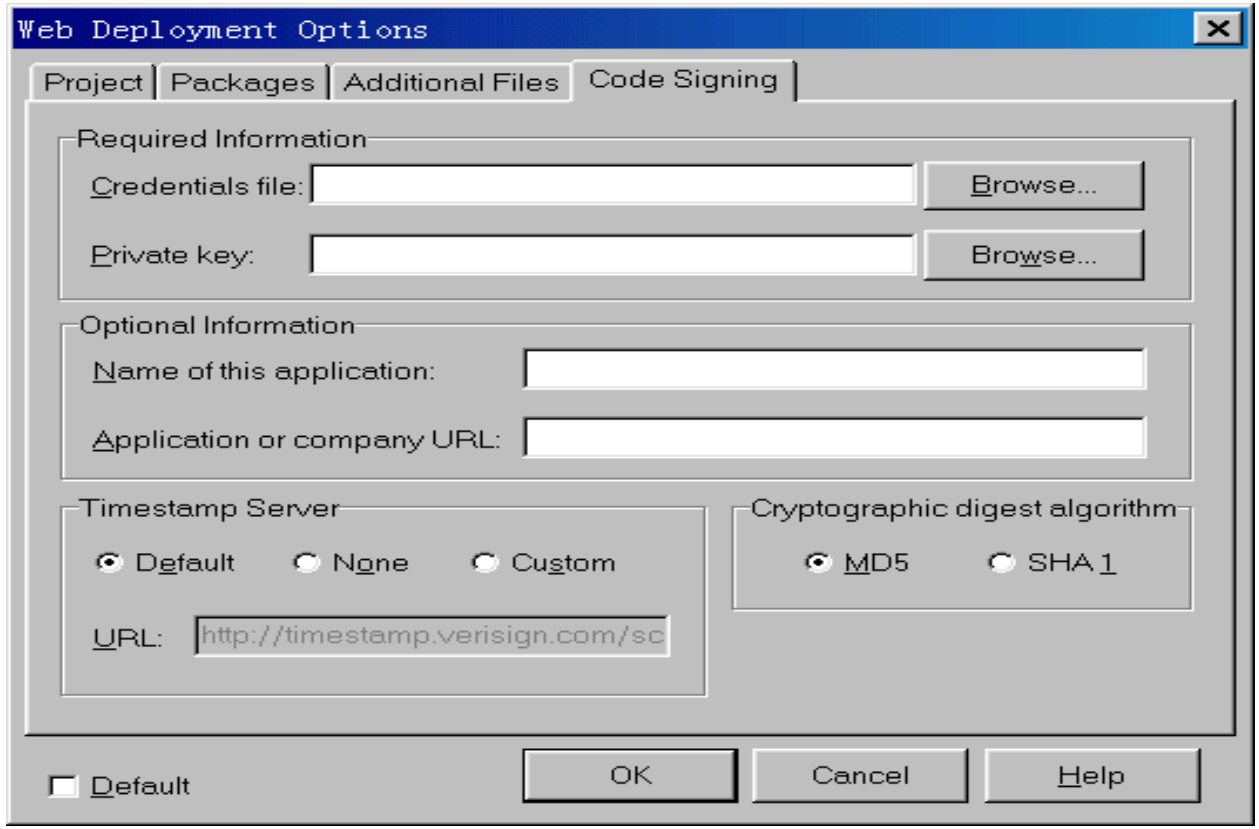

Web Deployment Options
Project
Packages
Additional Files
Code Signing
Required Information
Credentials file:
Browse...
Private key:
Browse...
Optional Information
Name of this application:
Application or company URL:
Timestamp Server
Default
None
Custom
URL:
http://timestamp.verisign.com/sc
Cryptographic digest algorithm
MD5
SHA 1
Default
OK
Cancel
Help

图 5.14　数字签名

第六章　OLE自动化

所谓OLE自动化(OLE Automation)，是Windows应用程序之间相互操纵的一种技巧，被操纵的一端称为自动化对象或自动化服务器，而操纵自动化对象的一端称为自动化控制器或自动化客户。一个应用程序或DLL可以兼具服务器和客户两种角色。本章的重点是怎样操纵OLE自动化对象，怎样创建自动化服务器。

6.1　怎样操纵自动化对象

要操纵自动化对象，您首先要创建自动化控制器或者称自动化客户。在C++ Builder 3中，要创建自动化客户有两种方式：引入类型库和调用Variant::Exec。

6.1.1　通过引入类型库来操纵自动化服务器

要引入自动化服务器的类型库，使用“Project”菜单上的“Import Type Library”命令，C++ Builder 3将打开“Import Type Library”对话框，如图6.1所示。

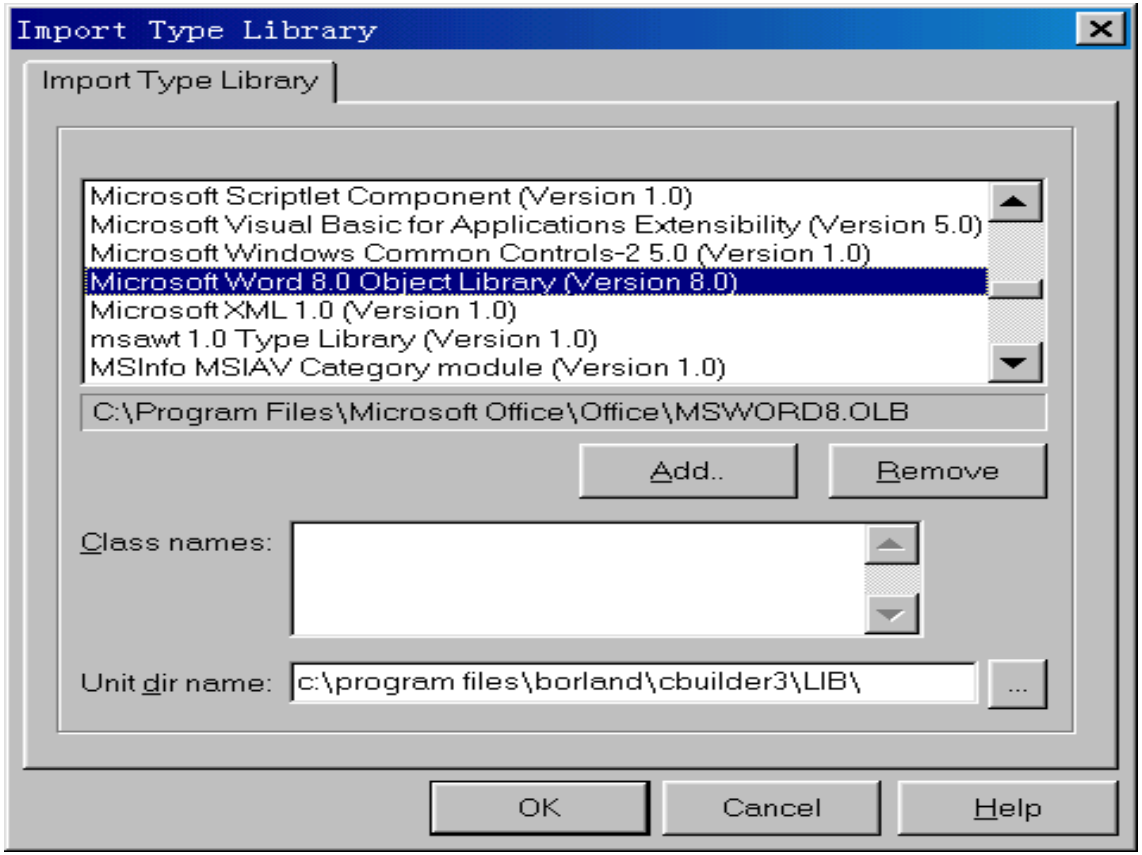


图 6.1　引入类型库

选择其中一个类型库如“Microsoft Word 8.0 Object Library”，然后单击OK按钮。如果要引入的类型库没有列出来，可能是因为还没有注册，此时就单击“Add”按钮先注册。

引入了类型库后，C++ Builder 3会自动生成该类型库的C++外套文件，从中可以看出类型库中的所有GUID常量和组件类(CoClass)。

如果要通过自动化服务器中的双重接口来操纵自动化服务器，您首先要用一个“CoClass客户代理类”来创建自动化对象的一个实例，程序示例如下：

```
TCOMWordBasic wb = CoApplication::Create( );
```

注意：您必须在自动化客户程序中包含C++外套的头文件，否则，上面一行代码将出错，因为编译器找不到CoApplication和TCOMWordBasic的声明。

创建了自动化对象的实例后，您就可以调用自动化对象中的方法，完整的示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
TCOMWordBasic wb = CoApplication::Create( );
if (OpenDialog1->Execute( ))
      wb->FilePrint(Variant::NoParam( ), Variant::NoParam( ), Variant::NoParam( ),
      Variant::NoParam( ), Variant::NoParam( ), Variant::NoParam( ), Variant::NoParam( ),
      Variant::NoParam( ), Variant::NoParam( ), Variant::NoParam( ), Variant::NoParam( ),
      Variant::NoParam( ), Variant(OpenDialog1->FileName), Variant::NoParam( ));
}
```

如果要通过自动化服务器中的调度接口来操纵自动化服务器，首先要用“CoClass客户代理类”来创建自动化对象的一个实例，程序示例如下：

```
    WordBasicDisp wb = CoApplication::Create( );
```

调用Bind函数，并且把'Word.Basic'作为参数传递给它，完整的示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
```

```
{
WordBasicDisp wb = CoApplication::Create( );
wb.Bind('Word.Basic');
if (OpenDialog1->Execute( ))
      wb.FilePrint(Variant::NoParam( ), Variant::NoParam( ), Variant::NoParam( ),
      Variant::NoParam( ), Variant::NoParam( ), Variant::NoParam( ), Variant::NoParam( ),
      Variant::NoParam( ), Variant::NoParam( ), Variant::NoParam( ), Variant::NoParam( ),
      Variant::NoParam( ), Variant(OpenDialog1->FileName), Variant::NoParam( ));
}
```

注意：这里调用wb的方法时用小圆点而不是->。

6.1.2 通过调用Variant::Exec来操纵自动化服务器

如果您不能打开自动化服务器的类型库，您就只能通过调用Variant::Exec来操纵自动化服务器。

首先，您要创建一个Variant类型的变量，用这个变量来引用自动化服务器：

```
Variant wb = Variant::CreateObject('Word.Basic');
```

然后调用Variant::Exec作为外套来执行自动化服务器中的某个方法，完整的示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
Variant wb = Variant::CreateObject('Word.Basic');
if (OpenDialog1->Execute( ))
```

```
    wb.Exec(Procedure("FilePrint") << NamedParam("FileName",
    Variant(OpenDialog1->FileName)))
}
```

最后我们再谈一谈自动化对象的生存期问题，以上述程序示例为例，如果wb变量被声明为全局的，那么这个自动化对象的生存期就与应用程序的生存期是相同的，如果wb变量在某个例程内部声明，该自动化对象的生存期与局部变量的生存期是相同的。因此，一般来说，您没有必要显式地删除自动化对象的实例。

6.1.3 迟后联编

从本质上讲，自动化对象也是类，只不过它是专门实现IDispatch接口的类，因此，访问自动化对象中的特性和方法与访问一般类中的特性和方法是很相似的。

不同的是，访问OLE自动化对象的特性和方法采用的是“迟后联编”，也就是说，不需要事先声明特性和方法的原型，编译器也不进行类型检查，至于特性和方法能不能访问成功要到运行期实际访问后才知道。如果实际访问没有成功，将触发EOleError异常。

造成访问失败的原因可能有这么几种情况：

- 可能是因为自动化对象的实例还没有创建，或者没有取得对自动化对象的引用。
- 特性或方法的标识符在自动化对象中不存在。
- 参数的个数及数据类型不匹配。尽管有的自动化对象支持参数可省略的

调用，但没有遵循有关参数省略的规则(后面将详细介绍怎样省略参数)。

- 方法调用出现在表达式中，但该方法并没有返回一个值。
- 即使没有上述错误，但自动化对象本身也可能出现异常。

6.2 怎样创建自动化服务器

自动化服务器分为三类：一是In-Process型的服务器，二是Out-Of-Process(或称为Local)型的服务器，三是Remote型的服务器。不管是哪种类型的服务器，其基本特征是一致的，都要实现IDispatch接口并输出自动化对象。

所谓In-Process服务器，一般是动态链接库(DLL)，它没有单独的进程地址空间，而是直接映射到客户程序的进程地址空间中运行。创建这一类服务器的好处在于多个自动化客户可以同时共享OLE服务，访问自动化对象的特性和方法是直接的。

所谓Out-Of-Process服务器，一般是可单独执行的应用程序(EXE)，有它自己的进程地址空间，而自动化客户运行在另一个进程地址空间，Word就是一个典型的Out-Of-Process服务器。这种跨进程通信的能力是由IDispatch接口提供的。

所谓Remote服务器，是指服务器与客户程序跨越了机器边界。限于目前的DCOM版本，Remote服务器必须是可执行程序，不过将来也可能是动态链接库。

C++ Builder 3在创建自动化服务器方面作了重大改进，采用了一种全新的体

系结构即ActiveX框架，并且实现了与类型信息的无缝连接。您将会发现，创建自动化服务器实际上主要就是生成类型信息。

如果要创建Out-Of-Process型的自动化服务器，首先要使用“File”菜单上的“New Application”命令开始一个新的项目，因为Out-Of-Process型的自动化服务器是EXE，然后再创建一个自动化对象加到自动化服务器中。

要创建一个自动化对象，使用“File”菜单上的“New”命令打开“New Items”对话框，翻到“ActiveX”页，双击“Automation Object”图标，C++ Builder 3将打开“Automation Object Wizard”对话框，如图6.2所示。

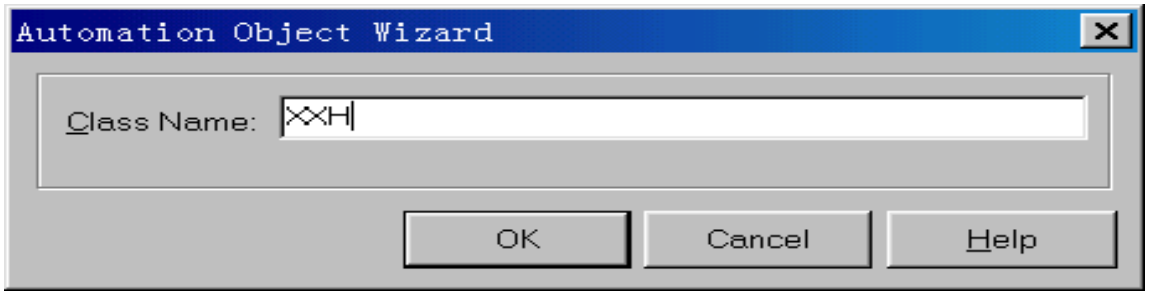


图 6.2　创建自动化对象的向导

在“Class Name”框内输入自动化对象的类名，单击OK按钮，C++ Builder 3将自动创建该自动化对象的类型库，文件名是Project1.TLB，并且自动打开“Type Library”编辑器显示类型信息。最后，先按Ctrl+F9键编译程序，如果没有错误，按F9键运行程序，此服务器将自动在Windows中注册。

6.3 自动化对象的类型库

如果保存刚刚创建的项目，您就会发现向导自动生成了下列文件：

- 类型库文件：Project1.TLB
- 类型库的接口描述文件：Project1_TLB.CPP
- 自动化对象中接口的实现单元文件：Unit2.PAS

向导自动生成类型库文件Project1.TLB，不过此文件是个二进制文件，只能用“Type Library”编辑器打开，代码编辑器能打开的是类型库的接口描述文件Project1_TLB.CPP，它是由“Type Library”编辑器维护的，您不要直接修改这个文件。

自动化向导能够自动创建TXXH对象的类型信息，不过，刚开始的时候，TXXH对象的接口是空的，要使客户程序能够操纵它，您应当在IXXH接口中加入新的成员。使用“View”菜单上的“Type Library”命令打开“Type Library”编辑器，如图6.3所示。

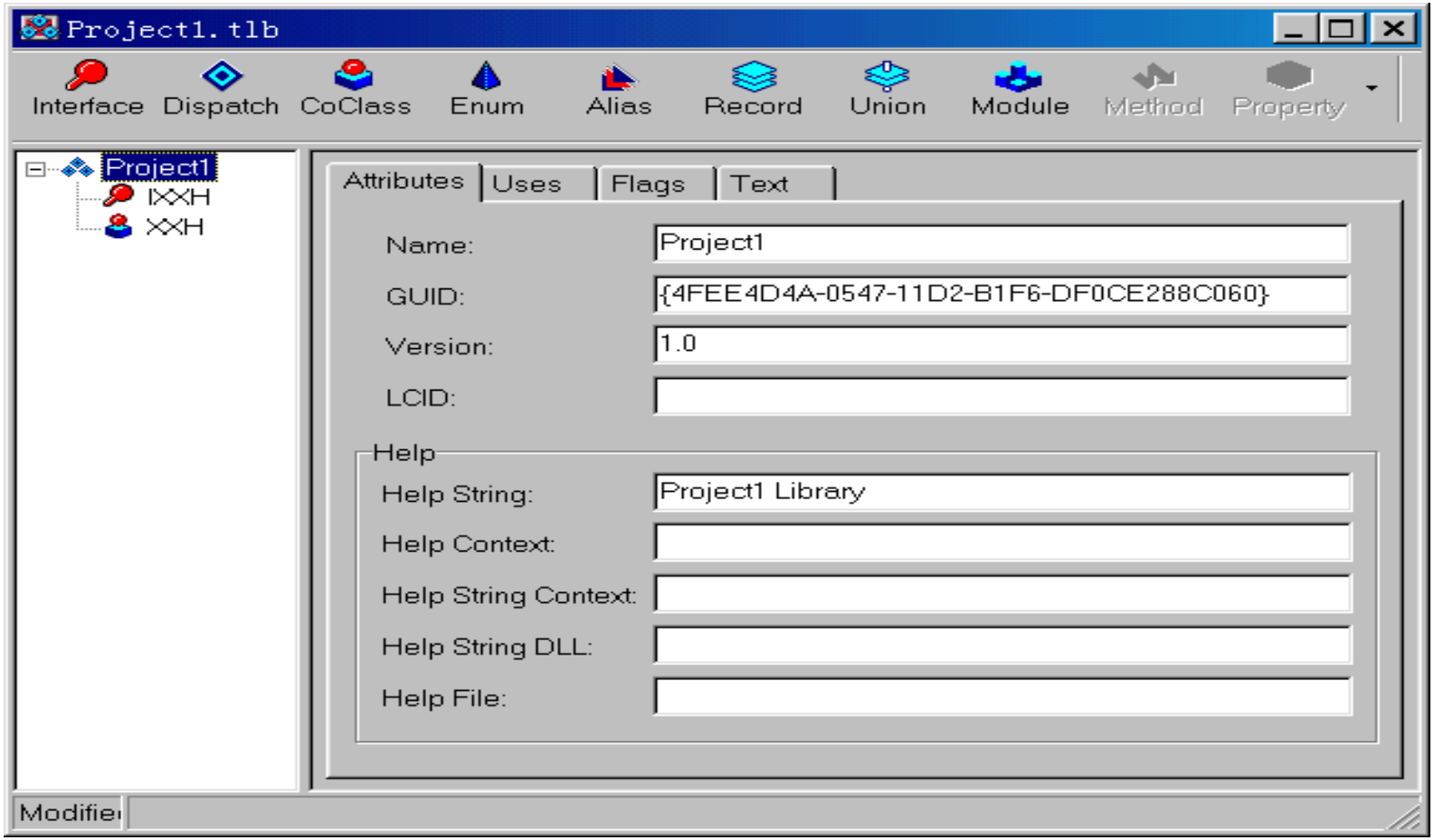


图 6.3 “Type Library”编辑器

要向接口中增加新的成员，在“Type Library”编辑器中选择IXXH节点，然后您可以有三种操作方式：

一是单击工具栏上的“Property”或“Method”图标，然后在“Attributes”页修改成员的属性。

二是在IXXH节点上单击鼠标右键，在弹出的菜单中选择“New”，再选择“Property”或“Method”，然后在“Attributes”页修改成员的属性。
三是翻到“Text”页，按IDL的文法直接键入特性或方法的声明。
假设我们要在IXXH接口中增加一个MyCaption特性，可以单击工具栏上的“Property”图标，然后在“Attributes”页的“Name”框内把特性名改为MyCaption，把数据类型改为WideString，单击工具栏上的“Refresh”按钮刷新类型库，新加入的特性就自动加到类型库的描述文件中，您可以在IXXH接口的声明中找到MyCaption特性的声明：

```
interface IXXH : public IDispatch
{
public:
    virtual HRESULT STDMETHODCALLTYPE get_MyCaption(BSTR* Value) = 0;
    virtual HRESULT STDMETHODCALLTYPE set_MyCaption(BSTR Value) = 0;
}
```

接下来，您可以打开IXXH接口的实现单元Unit2的头文件，找到Get_MyCaption和Set_MyCaption的声明：

```
class ATL_NO_VTABLE TXXHImpl: AUTOOBJECT_IMPL(TXXHImpl, XXH, IXXH)
{
protected:
    STDMETHOD(get_MyCaption(BSTR* Value));
    STDMETHOD(set_MyCaption(BSTR Value));
};
```

在Unit2中自动出现下列代码(注释是笔者加的):

```
//-----------------------------------------------------------------------
STDMETHODIMP TXXHImpl::get_MyCaption(BSTR* Value)
{
try
{
    //在这里写代码，用于读MyCaption特性
}
catch(Exception &e)
{
    return Error(e.Message.c_str( ), IID_IXXH);
}
return S_OK;
};
//-----------------------------------------------------------------------
STDMETHODIMP TXXHImpl::set_MyCaption(BSTR Value)
{
try
{
    //在这里写代码，用于设置MyCaption特性
}
catch(Exception &e)
```

```
{
    return Error(e.Message.c_str( ), IID_IXXH);
}
return S_OK;
};
```

您要做的就是定义Get_MyCaption和Set_MyCaption，用于读写MyCaption特性。

假设我们要在IXXH接口中增加一个Move函数，单击工具栏上的“Method”按钮，然后在“Attributes”页的“Name”框内把函数名改为Move，单击工具栏上的“Refresh”按钮刷新类型库，您可以在IXXH接口的声明中找到Move函数的声明：

```
interface IXXH : public IDispatch
{
public:
    virtual HRESULT STDMETHODCALLTYPE Move(void) ;
};
```

接着，您可以打开IXXH接口的实现单元Unit2的头文件，找到Move函数的声明：

```
class ATL_NO_VTABLE TXXHImpl: AUTOOBJECT_IMPL(TXXHImpl, XXH, IXXH)
{
protected:
    STDMETHOD(Move( ));
```

```
};
```
在Unit2中自动出现下列代码(注释是笔者加的):
```
STDMETHODIMP TXXHImpl::Move( )
{
try
{
     //在这里写代码，用于定义Move函数
}
catch(Exception &e)
{
return Error(e.Message.c_str( ), IID_IXXH);
}
return S_OK;
};
```
假设我们要在类型库中加入一个枚举类型DragMode，单击工具栏上的“Enum”图标，然后在“Attributes”页的“Name”框内把枚举类型名称改为DragMode。注意：GUID框的内容不能修改。刚开始的时候DragMode中还没有定义枚举常量，翻到“Text”页，然后参照下列格式键入枚举常量：
```
typedef enum tagDragMode
{
[helpstring("dmManual")]
dmManual = 0,
```

```
[helpstring("dmAutomatic")]
dmAutomatic = 1
} DragMode;
```

6.4 创建 In-Process 型的自动化服务器

前面讲的是怎样创建Out-Of-Process型的自动化服务器，现在我们要创建一个In-Process型的自动化服务器。由于In-Process型的自动化服务器通常是动态链接库，因此，您首先要创建一个ActiveX项目(库)，然后把自动化对象加到这个项目中。操作步骤是这样的：

使用“File”菜单上的“New”命令打开“New Items”对话框，翻到“ActiveX”页，双击“ActiveX Library”图标，C++ Builder 3将建立一个ActiveX项目，然后使用“File”菜单上的“New”命令再次打开“New Items”对话框，翻到“ActiveX”页，双击“Automation Object”图标打开“Automation Object Wizard”对话框，在“Class Name”框内键入自动化对象的类名，单击OK按钮，C++ Builder 3将自动创建类型库。下面的步骤与创建Out-Of-Process型的自动化服务器相同，此处不再赘述。

6.5 注册和调试自动化对象

对于Out-of-Process型的自动化服务器来说，由于它本身就是个可执行程序，

因此，第一次运行自动化服务器程序的时候，它就自动在Windows中注册。也可以采用这种方式注册自动化服务器，就是使用“Run”菜单上的“Parameters”命令打开“Run Parameters”对话框，在“Run Parameters”框内键入“/regserver”。如果要取消自动化服务器的注册，在“Run Parameters”框内键入“/unregserver”。

对于In-Process型的自动化服务器来说，由于它没法单独运行，要注册In-Process型的自动化服务器，您得使用“Run”菜单上的“Register ActiveX Server”命令。要从Windows中取消自动化服务器的注册，使用“Run”菜单上的“Unregister ActiveX Server”命令。

要调试Out-Of-Process型的自动化服务器比较简单，因为它本身就是个可执行程序，您只要事先在源程序中设好断点，然后启动自动化服务器程序。运行一个客户程序，在客户程序中访问自动化对象的特性和方法，如果遇到断点，C++ Builder 3的调试器就接管对服务器程序的控件权，这样您就可以采用诸如单步、跟踪等手段调试它了。

要调试In-Process型的自动化服务器(DLL或OCX)，首先要使用“Run”菜单上的“Parameters”命令打开“Run Parameters”对话框，在“Host Application”框内指定一个宿主程序(实际上就是前面所说的客户程序)，再用这个宿主程序访问自动化对象。

第七章　使用WinSock

从程序员的角度看，WinSock是一组用C语言写的API，用于通过Internet进行数据传输。虽然现在有很多现成的工具如WWW浏览器、FTP程序可以实现在Internet上传输数据和文件，但是，通过WinSock编程可以获得更大的灵活性。

编写WinSock应用程序本来是很麻烦的，不过，在C++ Builder 3中，您并不需要直接与WinSock中的API打交道，因为C++ Builder 3新增加了TClientSocket元件和TServerSocket元件，这两个元件封装了Windows的有关API，使得对WinSock的访问大大简化。

7.1　关于 Socket 的概述

要通过Internet传输数据，您至少需要一对Socket(意为插座)，其中一个Socket在客户端，另一个Socket在服务器端，一旦客户端和服务器端都接通了Socket，客户端和服务器端就可以相互通信，就好像墙上的电话插孔一样。用Socket建立的连接是建立在TCP/IP协议基础上的，同时也支持其它相关的协议，如XNS、DECnet以及IPX/SPX等。

C++ Builder 3分别用TClientSocket元件和TServerSocket元件来操纵客户端

Socket与服务器端Socket的连接和通信。要说明的是，这两个元件用于管理服务器和客户的连接，本身并不是Socket对象，操纵Socket对象的是TCustomWinSocket及其派生类，如TClientWinSocket、TServerWinSocket和TServerClientWinSocket等。

根据连接启动的方式以及本地Socket要连接的目标，Socket之间的连接可以分为三种类型：客户端连接、监听连接以及服务器端连接。

所谓客户端连接，是指由客户端的Socket提出连接请求，要连接的目标是服务器端的Socket。为此，客户端的Socket必须首先描述它要连接的服务器端Socket(主要是指服务器端Socket的地址和端口号)，然后再定位所要连接的服务器端Socket，找到以后，就向服务器端Socket请求连接。当然，服务器端的Socket此时未必正好处于准备好状态，不过，服务器端的Socket会自动维护客户请求连接的队列，然后在它认为合适的时候向客户端Socket发出“允许连接”(Accept)的信号，这时客户端Socket与服务器端Socket的连接就建立了。

所谓监听连接，服务器端Socket并不定位具体的客户端Socket，而是处于等待连接的状态。当服务器端Socket监听到或者说接收到客户端Socket的连接请求，它就响应客户端Socket的请求建立一个新的Socket句柄并与客户端连接，而服务器端Socket继续处于监听状态，还可以接收其它客户端Socket的连接请求。

所谓服务器端连接，是指当服务器端Socket接收到客户端Socket的连接请求后，就把服务器端Socket的描述发给客户端，一旦客户端确认了此描述，连接就建立了。

7.2 建立服务器端 Socket

把一个TServerSocket元件加到Form或数据模块上，您的应用程序就变成了一个TCP/IP服务器。当服务器处于监听状态时，服务器端的Socket对象用TServerWinSocket来操纵，当客户提出连接请求，服务器响应了连接请求并建立了连接，此时用TServerClientWinSocket来操纵服务器端Socket与客户端Socket的连接。

要使服务器能监听客户的连接请求，您必须首先指定端口号(设置TServerSocket元件的Port特性)，当然，如果服务器提供的是某些标准的服务如FTP、HTTP、FINGER或TIME，这些标准服务的端口号是事先约定的，也就是说，您只要指定了服务类型(设置TServerSocket元件的Service特性)，端口号就已经隐含确定了。如果您同时设置了Port特性和Service特性，C++ Builder 3将忽略Port特性。

指定了端口号或由Service特性隐含指定了端口号后，您就可以调用TServerSocket元件的Open函数开始监听。如果您在设计期把Active特性设为True，服务器就自动进入监听状态，也就是说，把Active特性设为True和调用Open是等价的。

一旦进入了监听状态，您就可以通过TServerSocket元件的Socket特性返回服务器端的Socket对象(TServerWinSocket)，并由此对象获得有关连接的信息，例如，Socket对象的SocketHandle特性可以获得Socket的Windows句柄，在直接调用WinSock的API时需要用到这个句柄。Socket对象的Handle特性可

以访问从Socket连接中检索消息的窗口。
服务器进入监听状态后，当客户提出连接请求，服务器就自动接受请求建立连接并触发OnClientConnect事件。此时，服务器端Socket与客户端Socket的连接由TserverClient-WinSocket来操纵。TServerWinSocket的Connections特性可以返回一个数组，该数组由服务器当前活动的连接(TServerClientWinSocket)组成。
如果要断开连接，只要在服务器端调用Close函数或者把Active特性设为False，注意：这将导致所有与客户的连接都断开，并退出监听状态。如果在客户端调用Close，将只断开该客户与服务器的连接，不影响其它客户的连接。
TServerSocket的直接上级是TCustomServerSocket，而TCustomServerSocket又是从TCustomSocket继承下来的，TCustomSocket又是从TComponent继承下来的，因此，TServerSocket元件是典型的非可视的元件。要注意的是，TCustomSocket是TServerSocket元件的虚拟基类，您不能直接创建TCustomSocket的实例。

7.3 建立客户端 Socket

把一个TClientSocket元件加到Form或数据模块上，您的应用程序就变成一个TCP/IP客户。您可以用TClientWinSocket来操纵客户端的Socket对象。
要说明的是，您可以把TServerSocket元件和TClientSocket元件加到同一个Form或数据模块上，这样，应用程序就兼具TCP/IP服务器和客户两种角色。

要建立与服务器的连接，您首先需指定要连接的服务器。指定服务器可以有两种方式，一是设置Host特性指定服务器的主机名如http://www.wSite.Com，这种方式比较直观，但要花费域名解析的时间。如果您不知道主机名，或者您想节省域名解析的时间，您可以设置Address特性直接指定主机的IP地址如123.197.1.2。这两种方式的效果是等价的，但如果您同时设置了Host特性和Address特性，C++ Builder 3将忽略Address特性。
要建立与服务器的连接，还要指定客户要连接的服务器上的端口号，这里也有两种方式，一是直接设置Port特性指定端口号如1024，也可以设置Service特性隐含确定端口号。如果您同时设置了Port特性和Service特性，C++ Builder 3将忽略Port特性。
指定了服务器以及要连接的端口号后(这称为在客户端描述服务器)，您只要调用Open函数或者把Active特性设为True，客户端Socket就向服务器端Socket提出连接请求。如果服务器此时处于监听状态，就会自动接受请求并建立连接。
一旦建立了与服务器的连接，您就可以用TClientSocket元件的Socket特性返回客户端的Socket对象(TClientWinSocket)，并由此对象获得有关连接的信息，例如，Socket对象的SocketHandle特性可以获得Socket的Windows句柄，如果您要直接调用WinSock的API，就要用到这个句柄。您还可以通过Socket对象的Handle特性访问从Socket连接中检索消息的窗口。您还可以用Socket对象的ASyncStyles特性设置要捕捉哪些消息。
在客户端要断开连接很简单，只要调用Close函数，此时，服务器端将触发OnClientDisconnect事件，而客户端将触发OnDisconnect事件。

TClientSocket元件的直接上级是TCustomSocket，而TCustomSocket又是从TComponent继承下来的，因此，TClientSocket元件也是一个典型的非可视的元件。

7.4 怎样在网络上传输数据

一旦服务器端Socket与客户端Socket之间建立了连接，就可以通过Internet传输数据和文件，这里先要介绍WinSock中“阻塞”与“非阻塞”的概念。

在Windows环境下，一般采用“非阻塞”方式。对于客户端Socket来说，如果把ClientType特性设为ctNonBlocking，表示采用非阻塞方式进行连接。当位于另一端的服务器端Socket试图进行读或写时，客户端Socket就会得到通知(OnRead事件或OnWrite事件)。

对于服务器端Socket来说，如果把ServerType特性设为stNonBlocking，表示采用非阻塞方式进行连接。当位于另一端的客户端Socket试图进行读或写时，服务器端Socket就会得到通知(OnClientRead事件或OnClientWrite事件)。

在响应这些事件的句柄中都有一个Socket参数，这个参数很有用，它就是服务器端或客户端的Socket对象(TClientWinSocket或TServerWinSocket或TServerClientWinSocket)，这些Socket对象的共同基类是TCustomWinSocket，TCustomWinSocket提供了在Internet上传输数据的方法，例如，要读取数据，可以调用ReceiveBuf或ReceiveText，要发送数据，可以调用SendBuf、SendStream、SendText或SendStreamThenDrop，其中，SendStreamThenDrop能够在发送完数据后自动断开Socket的连接。SendStream和

SendStreamThenDrop适合于发送流对象，例如文件，不过，您并不需要显式地删除流对象，因为当Socket之间的连接断开时，Socket会自动释放所有的流对象。

与“非阻塞”方式不同的是，在“阻塞”方式下没有诸如OnRead或OnWrite等异步事件，Socket必须主动去读或写数据，在读或写操作完成之前，应用程序处于等待状态，这样就不可避免地会影响整个应用程序的性能。

对于客户端Socket来说，如果把ClientType特性设为ctBlocking，表示采用阻塞方式进行连接。为了尽可能地减少“阻塞”方式的副作用，您可以把所有涉及到读写的操作放在一个单独的线程中，这样，在32位的Windows环境下其它线程能够继续得到执行。当然，如果在读写函数中客户程序本身没有其它事情要做，您就不必创建一个线程了。

对于服务器端Socket来说，如果把ServerType特性设为stThreadBlocking，表示采用阻塞方式进行连接，C++ Builder 3为每一个阻塞方式的连接自动分配一个新的线程，这样，即使有某个客户正在进行读写操作，其它客户也不必等待。服务器端的Socket用TServerClientThread对象来操纵每一个线程，当线程开始执行时，它就检查位于另一端的客户是否正在试图进行写，如是，就触发OnClientRead事件，如果检查出客户不在试图写，它就触发OnClientWrite事件让服务器去写。

实际上，这种用线程对象来管理连接的方式可以认为是“假阻塞”方式，就编程而言，与“非阻塞”方式是相似的。比较麻烦的是客户端，它怎么知道对方(指服务器端)已准备好读或写呢？这就要用到TWinSocketStream对象，调用这个对象的WaitForData可以保证服务器和客户的同步。用

TWinSocketStream对象进行读或写时，如果在20秒内操作还没有完成，TWinSocketStream对象就认为超时了，它就会自动断开连接，这样就能避免应用程序锁死。正是由于TWinSocketStream对象具有超时保护机制，您也可以把TWinSocketStream对象用于服务器端进行读或写。要注意的是，TWinSocketStream对象不能用于“非阻塞”方式。

也许有的读者要问，既然阻塞方式可能会影响应用程序的性能，虽然用多线程技术可以加以弥补，但又使编程变得复杂，为什么还要用“阻塞”方式呢？这是因为，在阻塞方式下，读或写是主动的，您可以在读或写操作完成后安全地关闭Socket的连接。而在“非阻塞”方式下，读或写是随机的，Socket必须经常保持在连接状态才能捕捉到异步事件。

7.5 TCustomWinSocket

TCustomWinSocket是Socket对象的基类，它是在SCKTCOMP单元中声明的。编写WinSock应用程序有时需要直接操纵Socket对象，因此，这里先介绍TCustomWinSocket。

Addr特性

声明：_property sockaddr_in Addr;

这个只读的特性返回本地Socket的Internet地址，其中，sockaddr_in是一个结构。

ASyncStyles 特性

声明：_property TAsyncStyles ASyncStyles;
这个特性用于设置Socket可以接收到哪些异步事件，其中，TAsyncStyles是一个集合，可以包含下列元素：asRead(当连接已准备好读时)、asWrite(当连接已准备好写时)、asOOB(当超出带宽的数据抵达时)、asAccept(当另一个Socket请求连接时)、asConnect(当与另一个Socket建立了连接时)、asClose(当与另一个Socket的连接终止时)。注意：如果Socket设为非阻塞(Non-Blocking)方式，ASyncStyles特性应至少包含asRead和asWrite这两个元素。

Connected 特性

声明：_property bool Connected;
这个只读的特性返回Socket是否处于连接状态，返回True表示处于连接状态。注意：在使用Socket传输数据之前，首先要访问Connected特性，看看当前是否处于连接状态。

Data 特性

声明：_property void * Data;
这个特性是个无类型指针，指向一个应用程序专用的数据结构，这个数据结构中可能包含了有关客户端的信息，服务器端用此信息判断客户端的连接请求。

Handle 特性

声明：_property HWND Handle;
这个特性很重要，Socket能接收的异步事件都是以Windows消息的形式发给此句柄的。

LocalAddress 特性

声明：_property System::AnsiString LocalAddress;
这个只读的特性返回一个用“点分法”表示的本地Socket的IP地址，如“123.197.1.2”等。一个机器可以有多个IP地址，每个IP地址可以同时用于多个Socket连接，但每一个Socket连接的端口号(LocalPort)必须是相异的，以区别于同一个IP地址的其它连接。

LocalHost 特性

声明：_property System::AnsiString LocalHost;
这个只读的特性返回一个用字符方式表示的本地Socket的IP地址，如“http://www.MySite.Com”等。

LocalPort 特性

声明：_property int LocalPort;
这个只读的特性返回本地Socket的端口号。对于同一个IP地址来说，可以同时用于多个Socket连接，但每个连接的端口号必须是相异的，也就是说，LocalAddress特性和LocalPort特性的组合总是唯一的。

RemoteAddr 特性

声明：_property sockaddr_in RemoteAddr;
这个只读的特性返回本地Socket所连接的另一端Socket的Internet地址，其中，sockaddr_in是一个结构。RemoteAddr特性实际上是RemoteAddress和RemotePort的组合。

RemoteAddress 特性

声明：_property System::AnsiString RemoteAddress;
这个只读的特性以“点分法”的形式表示另一端Socket的IP地址，如123.197.1.2。

RemoteHost 特性

声明：_property System::AnsiString RemoteHost;
这个只读的特性返回另一端Socket的主机名，如http://www.wSite.Com。

RemotePort 特性

声明：_property int RemotePort;
这个只读的特性返回另一端Socket所使用的端口号。注意：如果客户向服务器请求某种特殊的服务，RemotePort特性返回的端口号可能不同于客户请求的端口号。

SocketHandle 特性

声明：_property int SocketHandle;
这个只读的特性返回Socket对象的Windows句柄，在直接调用WinSock的API时需要用到这个句柄。如果Connected特性返回False，也就是说连接还没有建立，SocketHandle特性就返回INVALID_SOCKET。

Close 函数

声明：void _fastcall Close(void);
这个函数用于断开Socket连接(如果Socket之间已建立了连接的话)。对于服务器端Socket来说，调用Close函数将断开与所有客户的连接并退出监听状态。对于客户端Socket来说，将只断开自己与服务器的连接，不会影响到其它客户的连接。

Listen 函数

声明：void _fastcall Listen(const System::AnsiString Name, const System::AnsiString Address, const System::AnsiString Service, Word Port, int QueueSize);
这个函数一般由服务器端Socket调用，使服务器端Socket进入监听状态。Name、Address、Service、Port参数分别取自Host、Address、Service、Port特性。

Lock 函数

声明：void _fastcall Lock(void);
这个函数将阻塞其它正在执行的线程，直到调用了Unlock为止。注意：一般情况下不要调用Lock，因为这会影响应用程序的性能。

Unlock 函数

声明：void _fastcall Unlock(void);
这个函数与Lock函数正相反，用于解除对其它线程的阻塞。

LookupName 函数

声明：in_addr _fastcall LookupName(const System::AnsiString name);
这个函数根据Name参数指定的主机名返回Socket相应的Internet地址，其中，in_addr是一个紧凑的结构。

LookupService 函数

声明：int _fastcall LookupService(const System::AnsiString service);
如果服务器提供的是标准的服务，如FTP、HTTP、FINGER，每一个标准的服务都对应着一个事先约定的端口号，例如，FTP服务对应的端口号就是21，TELNET服务对应的端口号就是23，NNTP服务对应的端口号就是119。这样，只要指定了服务类型，端口号就隐含确定了。这个函数根据Service参数指定的服务类型返回相应的端口号。关于Windows提供的服务及其端口号的详细描述请参见WINDOWS目录中的SERVICES文件。

ReceiveBuf 函数

声明：int _fastcall ReceiveBuf(void *Buf, int Count);
这个函数用于从Socket连接中读取Count个字节到Buf参数中，并返回实际读取的字节数。这个函数一般是在处理Socket对象的OnSocketEvent事件或者TServerSocket元件的OnClientRead事件或者TClientSocket元件的OnRead事件时调用的。
ReceiveBuf函数只适用于非阻塞方式的Socket，对于阻塞方式的Socket，只能用TWinSocketStream来读写信息，TWinSocketStream只在对方准备好的情况下才传输数据。

ReceiveLength 函数

声明：int _fastcall ReceiveLength(void);
这个函数用于估计从Socket连接中读取数据要开辟多大的缓冲区(字节数)。

ReceiveText 函数

声明：System::AnsiString _fastcall ReceiveText( );
这个函数从Socket连接中读取一个字符串并返回这个字符串，这个函数一般在响应Socket对象的OnSocketEvent事件时调用，或者在响应TServerSocket元件的OnClientRead事件时调用。注意：ReceiveText函数只适用于非阻塞方式的Socket对象。

SendBuf 函数

声明：int _fastcall SendBuf(void *Buf, int Count);
这个函数用于把缓冲区Buf中Count个字节发送到Socket连接上，并返回实际发送的字节数。这个函数一般是在响应Socket对象的OnSocketEvent事件时调用，或者在响应TServerSocket元件的OnClientWrite事件或TClientServer元件的OnWrite事件时调用。如果在发送数据的函数中出现错误，连接将终止并触发ESocketError异常。

SendStream 函数

声明：bool _fastcall SendStream(Classes::TStream* AStream);
这个函数把AStream参数指定的流发送到Socket连接中，如果发送成功，函数就返回True。这个函数一般在响应Socket对象的OnSocketEvent事件时调用，或者在响应TServerSocket元件的OnClientWrite事件或TClientSocket元件的OnWrite事件时调用。

SendStreamThenDrop 函数

声明：bool _fastcall SendStreamThenDrop(Classes::TStream* AStream);
这个函数与SendStream函数相似，也是把AStream参数指定的流发送到Socket连接中，如果发送成功，函数就返回True。不同的是，发送完毕后它会自动终止连接。

SendText 函数

声明：void _fastcall SendText(const System::AnsiString S);
这个函数用于发送一个字符串，这个函数一般在响应Socket对象的OnSocketEvent事件时调用，或者在响应TServerSocket元件的OnClientWrite事件或TClientSocket元件的OnWrite事件时调用。如果在发送数据时发生错误，连接将终止并触发ESocketError异常。

OnSocketEvent 事件

声明：_property TSocketEventEvent OnSocketEvent;
其中，TSocketEventEvent是这样声明的：
typedef void _fastcall (_closure * TSocketEventEvent)(System::TObject* Sender, TCustomWinSocket Socket， TSocketEvent SocketEvent);
当Socket对象接收到一个异步事件的通知就触发这个事件，其中，Sender参数和Socket参数似乎是重复的，都是指接收此事件的Socket对象。SocketEvent参数是C++的枚举类型，用于表示事件的种类，可以是以下值之一：seLookup、seConnecting、seConnect、seDisconnect、seListen、seAccept、seWrite、seRead。

OnErrorEvent 事件

声明：_property TSocketErrorEvent OnErrorEvent;
其中，TSocketErrorEvent是这样声明的：
typedef void _fastcall (_closure * TSocketErrorEvent)(System::TObject* Sender, TCustomWinSocket

Socket，TErrorEvent ErrorEvent，int &ErrorCode);
当Socket在创建、使用或关闭连接时发生错误就触发这个事件。一般情况下，Sender参数与Socket参数的含义是一致的。ErrorEvent参数用于表明错误的类型，可以是以下值之一：eeGeneral、eeSend、eeReceive、eeConnect、eeDisconnect、eeAccept。
在处理这个事件的句柄中，如果成功地解决了错误，您就把ErrorCode参数设为0，这样能避免触发ESocketError异常。

7.6 TClientWinSocket

由于Socket分客户端和服务器端两种，因此，一个实际的Socket对象并不是直接从TCustomWinSocket继承下来的，而是从TClientWinSocket或TServerWinSocket或TServerClientWinSocket继承下来的，其中，TClientWinSocket代表客户端的Socket对象，TServerWinSocket代表处于监听状态的服务器端的Socket对象，TServerClientWinSocket代表已连接的服务器端的Socket对象。这里先介绍TClientWinSocket。
TClientWinSocket在Scktcomp单元中是这样声明的：

```
class PASCALIMPLEMENTATION TClientWinSocket :
public Scktcomp::TCustomWinSocket
{
typedef Scktcomp::TCustomWinSocket inherited;
private:
```

```
    TClientType FClientType;
protected:
    virtual void _fastcall Connect(int Socket);
    void _fastcall SetClientType(TClientType Value);
public:
    _property TClientType ClientType ;
public:
    _fastcall TClientWinSocket(int ASocket):
    Scktcomp::TCustomWinSocket(ASocket) { }
    _fastcall virtual ~TClientWinSocket(void) { }
};
```

可以看出，TClientWinSocket是从TCustomWinSocket继承下来的，增加了一个ClientType特性用于区别阻塞方式的Socket和非阻塞方式的Socket。

如果把ClientType特性设为ctNonBlocking，表示客户端Socket为非阻塞方式，可以响应异步的读或写事件(OnSocketEvent)。在进行读或写的操作时，不影响其它线程的执行。

如果把ClientType特性设为ctBlocking，表示客户端Socket为阻塞方式，读或写的操作必须在两端同步进行。在阻塞方式下，您最好把所有涉及到读或写的操作放在一个单独的线程中，这样，在32位的Windows环境下能最大限度地减少对其它线程的影响。在ClientType特性设为ctBlocking的情况下，不会产生异步的读或写事件，此时用TWinSocketStream对象来进行读写操作。TWinSocketStream的超时机制能保证即使在读写操作中出现错误也不

会使整个应用程序挂起。

7.7 TServerWinSocket

TServerWinSocket用于描述处于TCP/IP监听状态的服务器端Socket。一个服务器端Socket可以同时连接多个客户端Socket，与不同客户的连接都有其单独的执行线程。

TServerWinSocket在Scktcomp单元中是这样声明的：

```
class PASCALIMPLEMENTATION TServerWinSocket :
public Scktcomp::TCustomWinSocket
{
typedef Scktcomp::TCustomWinSocket inherited;
private:
    // 私有的成员此处省略
protected:
    // 保护的成员此处省略
public:
    _fastcall TServerWinSocket(int ASocket);
    _fastcall virtual ~TServerWinSocket(void);
    TServerClientThread*    _fastcall    GetClientThread(TServerClientWinSocket*
    ClientSocket);
    _property int ActiveConnections;
```

```
    _property int ActiveThreads;
    _property TCustomWinSocket* Connections[int Index];
    _property int IdleThreads;
    _property TServerType ServerType;
    _property int ThreadCacheSize;
    _property TGetSocketEvent OnGetSocket;
        _property TGetThreadEvent OnGetThread;
    _property TThreadNotifyEvent OnThreadStart;
    _property TThreadNotifyEvent OnThreadEnd;
    _property TSocketNotifyEvent OnClientConnect;
    _property TSocketNotifyEvent OnClientDisconnect;
    _property TSocketNotifyEvent OnClientRead;
    _property TSocketNotifyEvent OnClientWrite;
    _property TSocketErrorEvent OnClientError;
};
```

可以看出，TServerWinSocket也是从TCustomWinSocket继承下来的，不过它增加了一些特性、方法和事件，这是因为处于监听状态的服务器端Socket需要管理多个连接。

ServerType 特性

声明：_property TServerType ServerType;

这个特性用于设置与客户的连接方式，如果设为stThreadBlocking，表示与

每一个客户的连接都自动分配一个线程(TServerClientThread对象)，当服务器端Socket需要读或写时就触发OnClientRead事件或OnClientWrite事件。
如果设为stNonBlocking，表示用非阻塞方式连接每一个客户，每个连接都在一个单独的线程中处理，并且用OnClientRead或OnClientWrite来通知服务器端Socket进行读或写。

ActiveConnections 特性

声明：_property int ActiveConnections;
这个只读的特性返回当前活动的连接数，实际上也就是Connections数组的元素个数。

ActiveThreads 特性

声明：_property int ActiveThreads;
这个只读的特性返回当前正在使用的TServerClientThread对象的个数，服务器端的Socket对象用TServerClientThread来管理与客户连接的每个线程。

Connections 特性

声明：_property TCustomWinSocket* Connections[int Index]
这个特性是一个数组，数组的每一个元素都代表一个当前活动的连接(TServerClientWinSocket对象)，索引为0的元素代表第一个连接，索引为1的元素代表第二个连接，依次类推。该数组的元素个数就是ActiveConnections特性的值。

如果ServerType特性设为stThreadBlocking，与每一个连接(TServerClientWinSocket对象)所对应的线程(TServerClientThread对象)可以通过GetClientThread函数返回。

ThreadCacheSize特性

声明：_property int ThreadCacheSize;
如果ServerType特性设为stThreadBlocking，每一个与客户的连接都将自动分配一个线程，为了改善应用程序的性能，即使连接已断开，C++ Builder 3并不删除对应的线程(TServerClientThread对象)，而是把它保存在缓存中，这样，当建立新的连接时就可以重用在缓存中的线程对象，而不需要新建一个线程对象。
那么，在缓存中保存多少个线程对象呢？这是由ThreadCacheSize特性指定的。ThreadCacheSize特性的值并非越大越好，如果缓存的线程对象太多，实际又用不了这么多，将造成资源的浪费。但如果ThreadCacheSize特性的值设得过小，又将影响应用程序的性能。

IdleThreads特性

声明：_property int IdleThreads;
这个只读的特性返回没有派上用场(空闲)的线程对象的个数。如果IdleThreads特性返回的值过大，说明ThreadCacheSize特性设得过大，这时候您可以参照ActiveThreads特性的值重新调整ThreadCacheSize特性的值。

GetClientThread 函数

声明：TServerClientThread* _fastcall GetClientThread(TServerClientWinSocket* ClientSocket);
这个函数返回由ClientSocket参数指定的连接所对应的线程对象。

OnClientConnect 事件

声明：_property TSocketNotifyEvent OnClientConnect;
其中，TSocketNotifyEvent是这样声明的：

```
typedef void _fastcall (_closure *TSocketNotifyEvent)(System::TObject* Sender, TCustomWinSocket* Socket);
```

当服务器端Socket进入监听状态时将触发OnSocketEvent事件，SocketEvent参数设为seListen。当处于监听状态的服务器端Socket监听到客户的连接请求，就自动接收客户的请求并触发OnGetSocket事件。这时候又将触发OnSocketEvent事件，SocketEvent参数设为seAccept。如果ServerType特性设为stThreadBlocking而缓存中又没有线程对象可以重用，将触发OnGetThread事件，在处理OnGetThread事件的句柄中，如果没有显式地创建一个线程对象，服务器端的Socket对象将自动创建一个线程对象(TServerClientThread)。创建了线程对象后，线程开始执行时又将触发OnThreadStart事件。最后将触发OnClientConnect事件，通知服务器端Socket连接已完成。
注意：在这里Sender参数和Socket参数的含义是不同的，Sender参数是处于监听状态的服务器端Socket对象(TServerWinClient)，而Socket参数是TServerClientWinSocket对象。

OnClientDisconnect 事件

声明：_property TSocketNotifyEvent OnClientDisconnect;
当服务器端Socket与某个客户的连接被断开时将触发这个事件，同时，相应的TServerClientWinSocket对象被删除。如果ServerType特性设为stThreadBlocking，在触发了OnClientDisconnect事件之后还将触发OnThreadEnd事件。

OnClientError 事件

声明：_property TSocketErrorEvent OnClientError;
其中，TSocketErrorEvent是这样声明的：

typedef void _fastcall (_closure * TSocketErrorEvent)(System::TObject* Sender, TCustomWinSocket Socket，TErrorEvent ErrorEvent，int &ErrorCode);

当服务器端Socket对象在建立、使用和终止与某客户的连接时出现错误将触发这个事件。如果在处理此事件的句柄中解决了错误，您可以把ErrorCode参数设为0，这样就能避免触发ESocketError异常。ErrorEvent参数是错误的类型，可以是以下值之一：eeGeneral、eeSend、eeReceive、eeConnect、eeDisconnect、eeAccept。

OnClientRead 事件

声明：_property TSocketNotifyEvent OnClientRead;
C++ Builder 3用这个事件通知服务器端Socket对象去读取有关信息。

OnClientWrite 事件

声明：_property TSocketNotifyEvent OnClientWrite;
C++ Builder 3用这个事件通知服务器端Socket对象去写信息。

OnGetSocket 事件

声明：_property TGetSocketEvent OnGetSocket;
其中，TGetSocketEvent是这样声明的：

typedef void _fastcall (_closure *TGetSocketEvent)(System::TObject* Sender, int Socket, TserverClient-WinSocket* &ClientSocket);
C++ Builder 3用这个事件通知服务器端Socket需要创建一个新的TserverClient-WinSoc-ket对象来管理与客户的连接，创建的TServerClientWinSocket对象由ClientSocket参数返回。

OnGetThread 事件

声明：_property TGetThreadEvent OnGetThread;
其中，TGetThreadEvent是这样声明的：

typedef void _fastcall (_closure *TGetThreadEvent)(System::TObject* Sender, TServerClientWinSocket* ClientSocket, TServerClientThread* &SocketThread);
C++ Builder 3用这个事件通知服务器端Socket需要创建一个新的TServerClientThread对象，新创建的TServerClientThread对象由SocketThread参数返回。

OnThreadEnd 事件

声明：_property TThreadNotifyEvent OnThreadEnd;
如果ServerType特性设为stThreadBlocking，当服务器端Socket与某个客户的连接断开时将触发OnThreadEnd事件。在此事件触发之前已触发了OnClientDisconnect事件。

OnThreadStart 事件

声明：_property TThreadNotifyEvent OnThreadStart;
如果ServerType特性设为stThreadBlocking，当服务器端Socket建立了一个新的连接，新的线程开始执行时就会触发这个事件。

7.8 TServerClientWinSocket

TServerClientWinSocket用于描述与某个客户连接的服务器端Socket对象。有的读者也许要问，TServerWinSocket也是用于描述服务器端的Socket，那么它们两者之间的区别在哪里呢？当您把TServerSocket元件放到Form或数据模块上并进入监听状态，就自动创建了服务器端的Socket对象(TServerWinSocket)。只有当服务器端Socket监听到一个客户的连接请求并接收了连接请求后，才创建一个相应的TServerClientWinSocket对象。当与

该客户的连接断开时，相应的TServerClientWinSocket对象将被删除。
在Scktcomp单元中，TServerClientWinSocket是这样声明的：

```
class PASCALIMPLEMENTATION TServerClientWinSocket :
public Scktcomp::TCustomWinSocket
{
typedef Scktcomp::TCustomWinSocket inherited;
private:
	TServerWinSocket* FServerWinSocket;
public:
	_fastcall TServerClientWinSocket(int Socket, TServerWinSocket* ServerWinSocket);
	_fastcall virtual ~TServerClientWinSocket(void);
	_property TServerWinSocket* ServerWinSocket;
};
```

可以看出，TServerClientWinSocket也是从TCustomWinSocket继承下来的，增加了一个ServerWinSocket特性，并且重载了构造函数和析构函数。

ServerWinSocket特性

声明：_property TServerWinSocket* ServerWinSocket;
这个只读的特性返回处于监听状态的服务器端Socket对象(TServerWinSocket)。

7.9 TWinSocketStream

TWinSocketStream用于在“阻塞”方式下进行读或写操作，也许有的读者要问，Socket对象本身提供了读写的方法，为什么还要用TWinSocketStream呢？这是因为TwinSocket-Stream还提供了同步和超时保护机制。

在Scktcomp单元中，TWinSocketStream是这样声明的：

```
class PASCALIMPLEMENTATION TWinSocketStream : public Classes::TStream
{
typedef Classes::TStream inherited;
private:
	TCustomWinSocket* FSocket;
	int FTimeout;
public:
	_fastcall TWinSocketStream(TCustomWinSocket* ASocket, int TimeOut);
	_fastcall virtual ~TWinSocketStream(void);
	bool _fastcall WaitForData(int Timeout);
	virtual int _fastcall Read(void *Buffer, int Count);
	virtual int _fastcall Write(const void *Buffer, int Count);
	virtual int _fastcall Seek(int Offset, Word Origin);
	_property int TimeOut;
};
```

可以看出，TWinSocketStream是从TStream继承下来的，并且重载了TStream

的构造函数和析构函数，因此，TWinSocketStream具有流操作的能力。

TimeOut特性

声明：_property int TimeOut;
这个特性用于指定一个毫秒数，当读或写操作的时间超过这个数，就认为失败。

构造函数

声明：_fastcall TWinSocketStream(TCustomWinSocket* ASocket, int TimeOut);
在使用TWinSocketStream进行读写操作之前首先要用构造函数创建TWinSocketStream的实例，ASocket参数用于指定要用TWinSocketStream进行读写操作的Socket对象，一般是“阻塞”方式的客户端Socket对象，也可能是“阻塞”方式的服务器端Socket对象，但不能是“非阻塞”方式的Socket对象，否则将触发ESocketError异常。TimeOut参数用于指定超时的门槛值，以毫秒为单位，默认是20000毫秒。程序示例如下：

```
void _fastcall TMyClientThread::Execute( )
{
TWinSocketStream *pStream=new TWinSocketStream(ClientSocket1->Socket,6000);
try
{
    while (!Terminated && ClientSocket1->Active)
    {
```

```
        try
        {
            char buffer[10];
            GetNextRequest(buffer);
            pStream->Write(buffer, strlen(buffer) + 1);
            ...
        }
        catch (Exception &E)
        {
            if (!E.ClassNameIs("EAbort"))
                Synchronize(HandleThreadException( ));
        }
    }
}
_finally
{
    delete pStream;
}
}
```

注意：不要把TWinSocketStream与Socket对象的SendStream和SendStreamThenDrop混淆起来，后者也是针对流对象进行操作，但您并不需要显式地删除流对象，因为当Socket之间的连接断开时，Socket会自动释

放所有的流对象。而TWinSocketStream对象的实例不会自动删除，需要您显式地调用delete删除它。

WaitForData 函数

声明：bool _fastcall WaitForData(int Timeout);
这个函数用于等待对方准备好读或写，TimeOut参数指定等待的时间，以毫秒为单位。如果在规定的时间内对方准备好了，这个函数就返回True，否则就返回False。

Read 函数

声明：virtual int _fastcall Read(void *Buffer, int Count);
这个函数类似于Socket对象的ReadBuf，用于读取Count个字节到Buffer指定的缓冲区中，并返回实际读到的字节数。如果在读的函数中出现错误或连接的速度很慢，整个应用程序不会一直等待，只要读操作占用的时间超过了事先设定的门槛值例如20秒，Read函数就放弃此次读操作并返回0。
如果读操作确实需要占用较长的时间，您要么设置TimeOut特性把超时的门槛值放大，要么把较长的操作分成几次较短的操作，然后逐个调用Read函数。
注意：在调用Read之前一定要先调用WaitForData，以保证对方Socket已准备好。

Write 函数

声明：virtual int _fastcall Write(const void *Buffer, int Count);
这个函数类似于Socket对象的WriteBuf，用于从Buffer参数指定的缓冲区中发送Count个字节，并返回实际发送的字节数。如果在发送的函数中出现错误或连接的速度很慢，整个应用程序不会一直等待，只要写操作占用的时间超过了事先设定的门槛值例如20秒，Write函数就放弃此次读操作并返回0。
如果要发送的数据确实很长，您要么设置TimeOut特性把超时的门槛值放大，要么把较长的操作分成几次较短的操作，然后逐个调用Write函数。
注意：在调用Write之前一定要先调用WaitForData，以保证对方Socket已准备好。

7.10 一个网上交谈(Chat)程序

下面我们剖析一个示范程序，这个程序用TServerSocket元件和TClientSocket元件写了一个网上交谈(Chat)程序。要说明的是，这个程序是用Delphi 3写的，C++ Builder 3的程序员可以从中学到WinSock编程的基本技巧。
Chat程序只有一个Form，这个Form上同时放了TServerSocket元件和TClientSocket元件，表示Chat程序既是TCP/IP服务器，也是TCP/IP客户。为了分别显示交谈双方的内容，Chat程序用了两个TMemo元件。此程序运行后如图7.1所示：

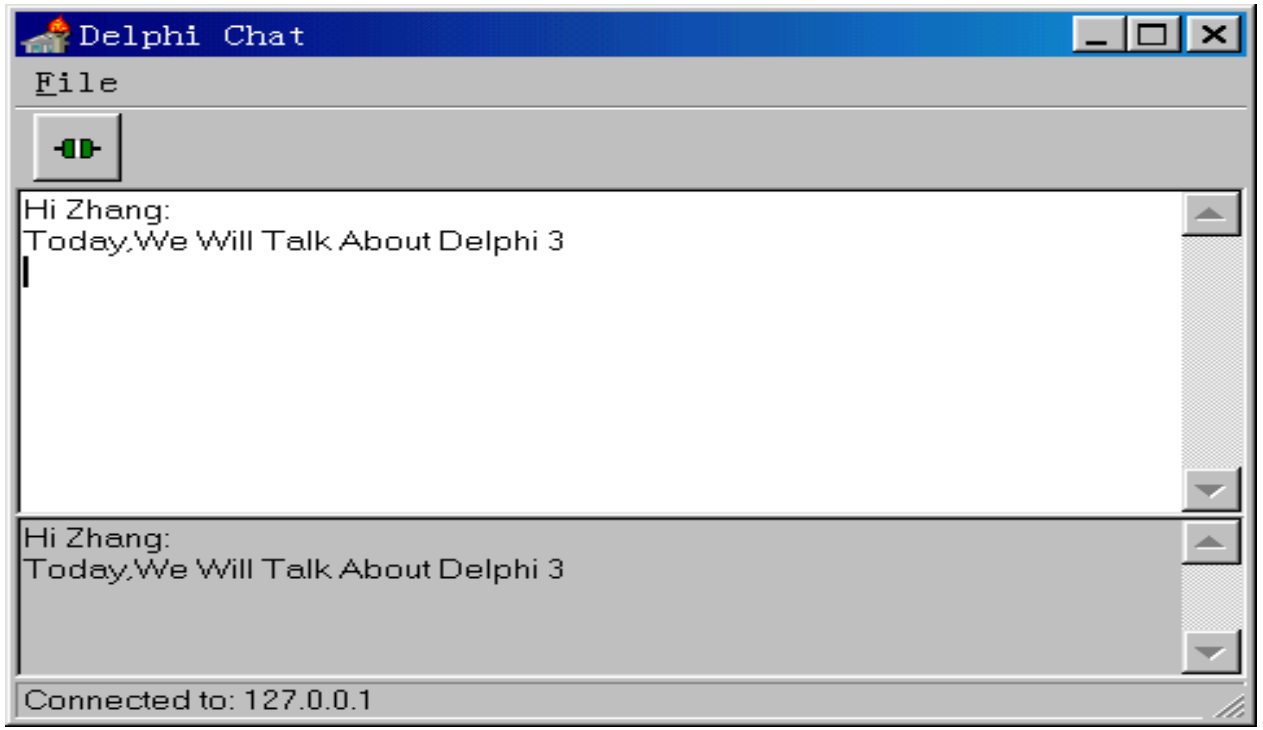


图 7.1　一个网上交谈程序

在网上交谈是一种非标准的服务，因此，交谈双方约定端口号(Port特性)为1024，Socket之间连接类型为非阻塞方式(NonBlocking)。

由于TServerSocket元件的Active特性设为False，因此，Chat程序刚开始运行的时候，首先要使服务器进入监听状态，代码如下：

```
Procedure TChatForm.FormCreate(Sender: TObject);
Begin
FileListenItemClick(nil);
End;
```

FileListenItemClick也是处理“File”菜单上的“Listen”命令的句柄，它是这样定义的：

```
Procedure TChatForm.FileListenItemClick(Sender: TObject);
Begin
FileListenItem.Checked := not FileListenItem.Checked;
If FileListenItem.Checked then Begin
      ClientSocket.Active := False;
      ServerSocket.Active := True;
      Statusbar1.Panels[0].Text := 'Listening...'
End
Else Begin
      If ServerSocket.Active then ServerSocket.Active := False;
      Statusbar1.Panels[0].Text := '';
End;
End;
```

FileListenItemClick实际上是一个切换程序，如果服务器已处于监听状态，就退出监听状态，如果服务器不在监听状态，就让服务器进入监听状态。为了直观起见，当服务器处于监听状态，就在“Listen”命令的左边显示一个✓符号，并且在状态栏上显示“Listening”。如果服务器不在监听状态，就取消“Listen”命令左边的✓符号，同时让状态栏显示空白。

服务器进入监听状态后，客户需要提出连接请求，待服务器接受了连接请求后交谈双方才可以进行交谈，这是通过“File”菜单上的“Connect...”命令或  按钮实现的：

```
Procedure TChatForm.FileConnectItemClick(Sender: TObject);
Begin
If ClientSocket.Active then ClientSocket.Active := False;
If InputQuery('Computer to connect to', 'Address Name:', Server) then
    If Length(Server) > 0 then
    With ClientSocket do Begin
        Host := Server;
        Active := True;
    End;
End;
```

FileConnectItemClick实际上也是一个切换程序，如果客户与服务器已连接，就断开连接，接着在一个询问框内键入要连接的服务器的地址(指主机名)，然后建立连接。

当客户提出连接请求的时候，在服务器端和客户端都将触发OnConnect事件，在客户端是这样处理OnConnect事件的：

```
Procedure TChatForm.ClientSocketConnect(Sender: TObject; Socket: TCustomWinSocket);
Begin
Statusbar1.Panels[0].Text := 'Connected to: ' + Socket.RemoteHost;
End;
```

上面这个句柄实际上就是在状态栏上显示“Connected to:”后跟服务器的主机名。

当服务器监听到客户的连接请求，就把Memo2清空，准备开始交谈，代码

如下：

```
Procedure TChatForm.ServerSocketClientConnect(Sender:TObject;Socket:TCustomWinSocket);
Begin
Memo2.Lines.Clear;
End;
```

当服务器接受了客户的连接请求，将触发OnAccept事件：

```
Procedure TChatForm.ServerSocketAccept(Sender: TObject; Socket: TCustomWinSocket);
Begin
IsServer := True;
Statusbar1.Panels[0].Text := 'Connected to: ' + Socket.RemoteAddress;
End;
```

IsServer是Chat程序中一个私有的布尔类型的变量，当服务器接受了一个客户的连接请求也就是说成为名副其实的服务器后，就把这个私有的变量设为True。

建立了连接后，客户就可以在Memo1中键入字符开始交谈，按下Enter键，除了使文本换行外，还把光标所在的行作为文本发出去，代码如下：

```
Procedure TChatForm.Memo1KeyDown(Sender:TObject; var Key: Word;Shift: TShiftState);
Begin
If Key = VK_Return then
      If IsServer then
            ServerSocket.Socket. Connections[0].SendText(
                  Memo1.Lines[Memo1.Lines. Count - 1])
```

```
    Else
        ClientSocket.Socket.SendText(Memo1.Lines[Memo1.Lines.Count - 1]);
End;
```

上面这段程序首先判断按下的是否是Enter键，如是，再判断IsServer是否为True，如是，就从TServerSocket元件的Socket特性返回服务器端的Socket对象，再由此对象的Connections数组的第一个元素(也只有一个元素)得到TServerClientWinSocket对象，最后调用此对象的SendText把光标所在行的文本发出去。如果IsServer为False，就从TClientSocket元件的Socket特性得到客户端Socket对象，再调用SendText把光标所在行的文本发出去。

当客户端收到文本后就把收到的文本添加在Memo2的最后，代码如下：

```
Procedure TChatForm.ClientSocketRead(Sender: TObject; Socket: TCustomWinSocket);
Begin
Memo2.Lines.Add(Socket.ReceiveText);
End;
```

当服务器端收到文本后也把收到的文本添加在Memo2的最后，代码如下：

```
Procedure TChatForm.ServerSocketClientRead(Sender:TObject;Socket:TCustomWinSocket);
Begin
Memo2.Lines.Add(Socket.ReceiveText);
End;
```

此外，在客户端还增加了处理错误的代码：

```
Procedure TChatForm.ClientSocketError(Sender: TObject; Socket: TCustomWinSocket;
```

```
ErrorEvent: TErrorEvent; var ErrorCode: Integer);
Begin
Memo2.Lines.Add('Error connecting to :' + Server);
ErrorCode := 0;
End;
```

至此，一个能够在网上交谈的程序基本写好了。最后要说明的是，编译此程序后，您可以把执行文件分别在两台计算机上运行，这两台计算机必须有固定的IP地址。也可以把此程序在单机上调试，即使该计算机没有联入Internet。单机调试时，单击  按钮后在“Address Name”框内键入“localhost”，如图7.2所示：

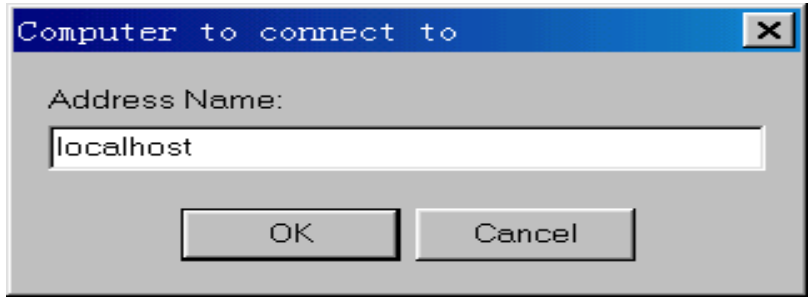


图 7.2　键入服务器的主机名

至于为什么要键入“localhost”，读者可以打开位于Windows目录的HOSTS.SAM文件，那里为IP地址127.0.0.1定义了主机名为“localhost”。

第八章　使用FTP控件

FTP是File Transfer Protocol的缩写，意为文件传输协议，用于在Internet上传输文本和二进制文件。由于FTP的应用非常广泛，现在流行的WWW浏览器如IE、Netscape都支持FTP协议，也就是说，浏览器就是一个简单的FTP客户程序。此外，专业的FTP客户程序也很多，比较著名的是CuteFTP、GetRight等。更值得一提的是，Windows 95已经把FTP作为一项标准的功能集成到操作系统中，用户只要运行FTP命令，就可以得到FTP>提示符，然后就可以象使用DOS命令那样进行FTP的操作。

不过，本章并不是教您如何使用这些现成的工具软件，而是要讲述怎样在应用程序中实现FTP协议，换句话说，就是自己写FTP客户程序。FTP协议本身是很复杂的，幸运的是，C++ Builder 3引进了NetManage公司的一个ActiveX控件，C++的类名叫TNMFTP。下面我们将详细介绍TNMFTP的特性、方法和事件，读者重点要掌握如何登录到FTP服务器上、如何得到和显示FTP服务器上的目录、如何从FTP服务器下载文本文件和二进制文件、如何把本地磁盘中的文本文件和二进制文件上载到FTP服务器。

8.1 FTP 控件的特性

BytesRecvd 特性

声明：property BytesRecvd: longint;
这个特性返回已收到的字节数，与BytesTotal特性配合使用可以显示文件下载的进度。程序示例如下：

```
void _fastcall TForm1::NMFTP1PacketRecvd(TObject *Sender)
{
...
StatusBar1->SimpleText = IntToStr(NMFTP1->BytesRecvd)+
" bytes of "+IntToStr(NMFTP1->BytesTotal)+" received";
}
```

BytesSent 特性

声明：property BytesSent: longint;
这个特性返回已发出的字节数，与BytesTotal特性配合使用可以显示文件上载的进度。程序示例如下：

```
void _fastcall TForm1::NMFTP1PacketSent(TObject *Sender)
{
StatusBar1->SimpleText = IntToStr(NMFTP1->BytesSent)+
" bytes of "+IntToStr(NMFTP1->BytesTotal)+" sent";
```

}

BytesTotal 特性

声明： property BytesTotal: longint;
这个特性返回此次下载或上载总共要传输的字节数，与BytesRecvd特性或BytesSent特性配合使用可以显示传输的进度。

CurrentDir 特性

声明： property CurrentDir: string;
这个特性返回FTP服务器上的当前目录。程序示例如下：

```
void _fastcall TForm1::NMFTP1Success(TCmdType Trans_Type)
{
switch (Trans_Type) {
      case cmdChangeDir:
            StatusBar1->SimpleText = "ChangeDir success";
            Form1.Caption = NMFTP1->CurrentDir;
            break;
      case cmdMakeDir: StatusBar1->SimpleText = "MakeDir success"; break;
      case cmdDelete: StatusBar1->SimpleText = "Delete success"; break;
      case cmdRemoveDir: StatusBar1->SimpleText = "RemoveDir success"; break;
      case cmdList: StatusBar1->SimpleText = "List success"; break;
      case cmdRename: StatusBar1->SimpleText = "Rename success"; break;
```

```
        case cmdUpRestore: StatusBar1->SimpleText = "UploadRestore success"; break;
        case cmdDownRestore:
            StatusBar1->SimpleText="DownloadRestore success";
            break;
        case cmdDownload: StatusBar1->SimpleText = "Download success"; break;
        case cmdUpload: StatusBar1->SimpleText = "Upload success"; break;
        case cmdAppend: StatusBar1->SimpleText = "UploadAppend success"; break;
        case cmdReInit: StatusBar1->SimpleText = "ReInit success"; break;
        case cmdAllocate: StatusBar1->SimpleText = "Allocate success"; break;
        case cmdNList: StatusBar1->SimpleText = "NList success"; break;
        case cmdDoCommand: StatusBar1->SimpleText = "DoCommand success"; break;
        default: ShowMessage("Unrecognized command success."); break;
}
}
```

Host 特性

声明： property Host: String;

这个特性用于指定FTP服务器的主机名或IP地址，如127.0.0.1。程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
NMFTP1->ReportLevel = Status_Basic;
```

```
if (CheckBox1->Checked)
{
     NMFTP1->Proxy = Edit6->Text;
     NMFTP1->ProxyPort = StrToInt(Edit7->Text);
}
NMFTP1->Host = HostTxt->Text;
NMFTP1->Port = StrToInt(PortTxt->Text);
NMFTP1->UserID = UserTxt->Text;
NMFTP1->Password = PassTxt->Text;
NMFTP1->Connect( );
}
```

这个程序用一个复选框让用户选择是否要使用代理服务器，如需要用代理服务器，就把Proxy特性设为代理服务器的IP地址，ProxyPort特性设为代理服务器的端口号。

LocalIP 特性

声明：property: LocalIP: String;

这个只读的特性以“点分法”的形式返回本地计算机的IP地址。如果本地计算机有多个IP地址，这个特性返回其中的第一个。程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
ShowMessage("您的计算机的IP地址是 "+NMFTP1->LocalIP);
```

}

Password 特性

声明：property Password: string;
登录到FTP服务器的时候需要给出用户名和口令(通常是您的E-mail地址)，这个特性用于给出口令。

Port 特性

声明：property Port: integer;
这个特性用于指定FTP服务器的端口号，默认是21，因为FTP是标准服务。

Proxy 特性

声明：property Proxy: String;
这个特性用于指定代理服务器的主机名或IP地址。如果不需要用代理服务器，这个特性必须空着。

ProxyPort 特性

声明：property ProxyPort: integer;
这个特性用于指定代理服务器的端口号，如果不用代理服务器，这个特性必须空着。

RemoteIP 特性

声明：property RemoteIP: string;
这个特性返回以“点分法”表示的FTP服务器的IP地址。程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
ShowMessage("当前连接的是:"+NMFTP1->RemoteIP);
}
```

ReplyNumber 特性

声明：property ReplyNumber: Smallint;
这个只读的特性返回FTP服务器响应客户请求的应答代码。程序示例如下：

```
void _fastcall TForm1::NMFTP1Status(TComponent *Sender, AnsiString Status)
{
if (StatusBar1) StatusBar1->SimpleText = Status;
if (NMFTP1->ReplyNumber = = 530)
    ShowMessage("Requested action not taken");
}
```

ReportLevel 特性

声明：property ReportLevel: integer;
这个特性用于指定状态信息的详细程度，可以设为以下值：Status_None、Status_Informational、Status_Basic、Status_Routines、Status_Debug、

Status_Trace。

Status 特性

声明： property Status: String;
这个特性返回当前的状态信息。

Timeout 特性

声明： property TimeOut: Integer;
这个特性用于指定一个毫秒数，如果超过规定的时间Socket没有响应就触发异常。

TransactionReply 特性

声明： property TransactionReply: String;
这个特性返回上一次命令的执行结果，程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
NMFTP1->DoCommand("SYST");
ShowMessage(NMFTP1->TransactionReply);
}
```

UserId 特性

声明： property UserID: string;

这个特性用于给出用户名，因为有的FTP服务器需要给出用户名。

8.2 FTP控件的方法

Abort函数

声明：procedure Abort;
这个函数用于中止本次下载或上载，但不断开与FTP服务器的连接。

Allocate函数

声明：procedure Allocate(FileSize: integer);
这个函数其实很少用到，用于预先分配一些空间用来存放文件，FileSize参数指定空间的大小，以字节数为单位。程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
int FileHandle;
if (FileHandle = open("C:\\WINNT\\SouthPark.bmp",O_RDONLY | O_BINARY) != -1)
    NMFTP1->Allocate(filelength(FileHandle));
}
```

ChangeDir函数

声明：procedure ChangeDir(DirName: string);

这个函数请求FTP服务器改变当前目录为Directory参数指定的目录。

Connect 函数

声明：procedure Connect;
这个函数很重要，用于登录到FTP服务器，Host、Port、UserID和Password特性必须预先已设置。

Delete 函数

声明：procedure Delete(FileName: string);
这个函数在FTP服务器上删除由FileName参数指定的文件。

Disconnect 函数

声明：procedure Disconnect;
这个函数断开与FTP服务器的连接。

DoCommand 函数

声明：procedure DoCommand(CommandStr: string);
这个函数用于在FTP服务器上执行一条命令(由CommandStr参数指定)。调用了这个函数后，一般要访问TransactionReply或ReplyNumber特性取得执行的结果。

Download 函数

声明：procedure Download(RemoteFile, LocalFile: string);
这个函数用于从FTP服务器下载由RemoteFile参数指定的文件，然后在本地以LocalFile参数指定的名称保存。如果LocalFile参数设为空，表示名称不变。
程序示例如下：

```
void _fastcall TForm1::Button6Click(TObject *Sender)
{
NMFTP1->Mode(MODE_IMAGE);
NMFTP1->Download("file.zip", "c:\\download\file.zip");
}
```

DownloadRestore 函数

声明：procedure DownloadRestore(RemoteFile, LocalFile: string);
TNMFTP控件支持断点续传，这个函数能够接着上次中断的地方继续下载。

List 函数

声明：procedure List;
这个函数用于返回FTP服务器中当前目录下的文件列表，列表中的每一项将触发OnListItem事件。程序示例如下：

```
void _fastcall TForm1::NMFTP1ListItem(AnsiString Listing)
{
Memo1->Lines->Add(Listing);
```

}

MakeDirectory 函数

声明：procedure MakeDirectory(DirectoryName: string);
这个函数在FTP服务器上创建一个目录，目录名由Directory参数指定。

Mode 函数

声明：procedure Mode(TheMode: integer);
这个函数用于设置传输模式，设为MODE_ASCII表示按文本块的方式传输，设为MODE_IMAGE表示按一个字节一个字节地传输(图像)。

Nlist 函数

声明：procedure Nlist;
这个函数用于返回当前目录中的文件名列表，列表中的每一项将触发OnList事件。

Reinitialize 函数

声明：procedure Reinitialize;
如果您要以另一个用户名或口令登录到FTP服务器，就调用这个函数，这个函数将向FTP服务器发出重新初始化的请求。

RemoveDir 函数

声明：procedure RemoveDir(DirectoryName: string);
这个函数在FTP服务器中删除由Directory参数指定的目录。

Rename 函数

声明：procedure Rename(Filename, Filename2: string);
这个函数把FTP服务器当前目录中的一个文件换名，Filename参数是原来的名称，Filename2参数是新的名称。

Upload 函数

声明：procedure Upload(LocalFile, RemoteFile: string);
这个函数用于把本地磁盘上由LocalFile参数指定的文件上载到FTP服务器的当前目录，并且以RemoteFile参数指定的名称保存。如果RemoteFile参数指定的文件已存在，原先的文件将被覆盖。

UploadAppend 函数

声明：procedure UploadAppend(LocalFile, RemoteFile: string);
这个函数与Upload函数相似，不同的是，如果RemoteFile参数指定的文件已存在，上载的文件将添加在原先这个文件的后面。

UploadRestore 函数

声明：procedure UploadRestore(LocalFile, RemoteFile: string; Position: integer);

TNMFTP控件支持断点续传，这个函数能够接着上次中断的地方继续上载，Position参数指定从什么地方开始续传。假设已传输了80个字节，Position参数就设为81。

8.3 FTP控件的事件

FTP控件的事件往往是与调用某个方法相关联的，如果事件被触发，通常表示调用成功，您可以在处理事件的句柄中得到FTP服务器的返回信息。

OnAuthenticationFailed事件

声明：property OnAuthenticationFailed: THandlerEvent;
其中，THandlerEvent是这样声明的：
THandlerEvent = Procedure(var handled: boolean) of Object;
登录到FTP服务器时，如果没有事先给出用户名和口令，或者用户名和口令是非法的，将会触发这个事件。更正了错误后，如果把Handled参数设为True，将再登录一遍。

OnConnect事件

声明：property OnConnect: TNotifyEvent;
当客户已经成功地登录到FTP服务器时将触发这个事件。

OnConnectionFailed 事件

声明： property OnConnectionFailed: TNotifyEvent;
当客户试图登录到FTP服务器失败时将触发这个事件。

OnConnectionRequired 事件

声明： property OnConnectionRequired: THandlerEvent;
FTP控件的大部分函数只能在已连接到FTP服务器上时才是有意义的，如果调用函数时没有连接到FTP服务器，将触发这个事件。在处理这个事件的句柄中，您应当调用Connect函数连接FTP服务器，然后把Handled参数设为True。程序示例如下：

```
void _fastcall TForm1::NMEcho1ConnectionRequired(bool &handled)
{
AnsiString BoxCaption;
AnsiString BoxMsg;
BoxCaption = "Connection Required";
BoxMsg = "Connection Required. Connect?";
if (MessageBox(0, &BoxMsg[1], &BoxCaption[1], MB_YESNO +
MB_ICONEXCLAMATION) = = IDYES)
{
     handled = true;
     NMFTP1->Connect( );
}
```

```
}
```

OnDisconnect 事件

声明：property OnDisconnect: TNotifyEvent;
当客户断开与FTP服务器的连接时将触发这个事件。注意：在处理这个事件的句柄中，如果需要访问VCL元件的话，最好先判断VCL元件是否存在，程序示例如下：

```
void _fastcall TForm1::NMFTP1Disconnect(TObject *Sender)
{
if (StatusBar1 != 0)
     StatusBar1->SimpleText = "Disconnected";
}
```

OnError 事件

声明：property OnError: TOnErrorEvent;
其中，TOnErrorEvent是这样声明的：

```
TOnErrorEvent = procedure(Sender: TComponent; Errno: word; Errmsg: string);
```

如果连接或传输数据时出错将触发这个事件，其中，Errno参数和Errmsg参数分别返回错误编号和错误的简短描述。程序示例如下：

```
void _fastcall TForm1::NMFTP1Error(TComponent *Sender, WORD Errno,
AnsiString Errmsg)
{
```

```
ShowMessage("Error "+IntToStr(Errno)+": "+Errmsg);
}
```

OnFailure 事件

声明： property OnFailure: TFailureEvent;
其中，TFailureEvent是这样声明的：

```
TFailureEvent = Procedure(var handled: boolean;    Trans_Type: TCmdType) of Object;
```

当调用TNMFTP控件的某个函数失败时将触发这个事件。CmdType 参数表明调用的是哪个函数。程序示例如下：

```
void _fastcall TForm1::NMFTP1Failure(bool &handled, TCmdType Trans_Type)
{
switch (Trans_Type) {
      case cmdChangeDir: StatusBar1->SimpleText = "ChangeDir failure"; break;
      case cmdMakeDir: StatusBar1->SimpleText = "MakeDir failure"; break;
      case cmdDelete: StatusBar1->SimpleText = "Delete failure"; break;
      ...
      default: ShowMessage("Unrecognized command failed."); break;
}
}
```

OnHostResolved 事件

声明：property OnHostResolved: TOnHostResolved;
当FTP服务器的主机名已找到(域名解析)就触发这个事件，如果解析没有成功，将触发OnInvalidHost事件。

OnInvalidHost 事件

声明：property OnInvalidHost: THandlerEvent;
当Host特性所指定的主机名是非法的，就触发这个事件。

OnListItem 事件

声明：property OnListItem: TNMListItem;
其中，TNMListItem是这样声明的：

```
TNMListItem = Procedure(Listing: string) of Object;
```

当程序调用List函数或NList函数时，每检索到一个文件就会触发这个事件。Listing参数就是检索到的项。对于List函数来说，Listing参数中包括文件名、长度、修改日期和文件属性。对于NList函数来说，Listing参数中只包括文件名。

OnPacketRecvd 事件

声明：property OnPacketRecvd: TNotifyEvent;
在文件下载过程中，每收到一个数据块就会触发这个事件，在处理这个事件的句柄中，您可以用BytesTotal特性和BytesRecvd特性显示下载的进度。

OnPacketSent 事件

声明：property OnPacketSent: TNotifyEvent;
在文件上载过程中，每发出一个数据块就会触发这个事件，在处理这个事件的句柄中，您可以用BytesTotal特性和BytesRecvd特性显示上载的进度。

OnStatus 事件

声明：property OnStatus: TOnStatus;
其中，TOnStatus是这样声明的：
TOnStatus = procedure(Sender: TComponent; Status: string) of Object;
每次当FTP服务器返回状态信息时将触发这个事件，Status参数就是当前的状态。

OnSuccess 事件

声明：property OnSuccess: TSuccessEvent;
其中，TSuccessEvent是这样声明的：
TSuccessEvent = Procedure(Trans_Type: TCmdType) of Object;
当某个命令被成功执行后将触发这个事件，Trans_Type参数就是被执行的命令。

OnTransactionStart 事件

声明：property OnTransactionStart: TNotifyEvent;
当程序调用诸如List、Download、DownloadRestore、Upload、UploadUnique、

UploadAppend、UploadRestore等函数开始传输数据时将触发这个事件。

OnTransactionStop 事件

声明：property OnTransactionStop: TNotifyEvent;
当程序调用了诸如List、Download、DownloadRestore、Upload、UploadUnique、UploadAppend、UploadRestore等函数并且数据已传输完毕时将触发这个事件。

OnUnSupportedFunction 事件

声明：property OnUnSupportedFunction: TUnsupportedEvent;
当程序调用某个函数而FTP服务器不支持该项命令时，就会触发这个事件。

第九章　使用UDP控件

UDP(User Datagram Protocol，意为用户报文协议)也是Internet上广泛采用的通信协议之一，它与TCP协议都是TCP/IP协议的传输层，它们均依赖于IP层进行通信和提供路由功能。不同的是，TCP是面向连接的有序的协议，它的确认和重传机制能保证数据可靠地到达目的地，并且顺序总是正确的。而UDP是非连接的协议，它传送的数据包是独立的，前后没有顺序关系。UDP没有确认机制，只有对报文头标和数据区的简单校验，因此，它不能保证数据传输的可靠性。显然，UDP在可靠性方面不如TCP，但它的效率却比TCP高。在网络质量较好的情况下，UDP也能得到不错的效果。

C++ Builder 3从NetManage公司引进了UDP控件，您不需要掌握UDP协议的细节，就可以访问UDP网络服务。

9.1　使用UDP控件的一般步骤

UDP控件不区分服务器端和客户端，只区分发送端和接收端，UDP控件既可以放在发送端，也可以放在接收端。两个UDP控件也可以放在同一个应用程序中。

与NetManage公司的其它Internet ActiveX控件一样，UDP控件也是基于

WinSock的API编写的，只不过UDP控件侧重处理UDP报文。
使用UDP控件的一般步骤是，首先把两个UDP控件分别放到两个应用程序的Form或数据模块上，然后设置作为接收端的UDP控件的LocalPort特性指定接收数据的端口号，设置作为发送端的TCP控件的RemoteHost特性指定对方的主机名或IP地址如“127.0.0.1”，设置RemotePort特性指定对方的端口号如1024。当然，上述操作也可以在运行期进行。要发送数据，只要调用SendBuffer，当有数据到达时，接收端将触发OnDataReceived事件，在处理此事件的句柄中，调用GetBuffer即可获得数据。

9.2 UDP控件的特性

LocalPort特性

声明：property LocalPort: integer;
这个特性用于指定本地的端口号。注意：在运行期不能修改这个特性。

RemoteHost特性

声明：property RemoteHost: string;
这个特性用于指定UDP报文发送给哪个计算机，可以是主机名，也可以是IP地址。

RemotePort 特性

声明： property RemotePort: integer;
这个特性用于指定对方计算机上使用的端口号。程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
AnsiString TmpStr;
NMUDP1->ReportLevel = Status_Basic;
NMUDP1->RemoteHost = "127.0.0.1";
NMUDP1->RemotePort = 6969;
TmpStr = "This is my data to send";
TMemoryStream *MyStream = new TMemoryStream( );
MyStream->WriteBuffer(&TmpStr, TmpStr.Length( ));
NMUDP1->SendStream(MyStream);
MyStream->Free( );
}
```

对于UDP控件来说，无论是RemotePort特性还是LocalPort特性都要避免使用保留的端口号，最好用1000以上的端口号比较安全。

ReportLevel 特性

声明： property ReportLevel: integer;
这个特性用于指定状态信息的详细程度，可以设为以下值： Status_None、

Status_Informational 、 Status_Basic 、 Status_Routines 、 Status_Debug 、 Status_Trace。

9.3 UDP 控件的方法

UDP控件的方法很少，因为UDP协议不需要事先建立连接，因此，UDP控件没有诸如Listen、Connect、Accept、Close等方法。

ReadBuffer 函数

声明：procedure ReadBuffer(var Buff: Array of char; var length: integer);
这个函数用于接收数据，接收到的数据放在缓冲区中，length参数指定缓冲区的长度。如果缓冲区的长度设得过小，将触发OnBufferInvalid事件。程序示例如下：

```
void _fastcall TForm1::NMUDP1DataReceived(TComponent *Sender, int NumberBytes,
AnsiString FromIP)
{
char *Buff = "                              ";
NMUDP1->ReadBuffer(Buff, NumberBytes );
}
```

ReadStream 函数

声明：procedure ReadStream(DataStream: TStream);

这个函数也是用于接收数据，接收到的数据放在DataStream参数指定的流中。如果DataStream参数指定的流不存在，将触发OnStreamInvalid事件。程序示例如下：

```
void _fastcall TForm1::NMUDP1DataReceived(TComponent *Sender, int NumberBytes,
AnsiString FromIP)
{
TMemoryStream *MyStream;
AnsiString TmpStr = AnsiString::StringOfChar(' ', NumberBytes+1);
MyStream = new TMemoryStream( );
NMUDP1->ReadStream(MyStream);
MyStream->ReadBuffer(&TmpStr,NumberBytes);
TmpStr.SetLength(NumberBytes);
TmpStr=(FromIP + ": " + TmpStr);
Memo1->Lines->Add(TmpStr);
MyStream->Free( );
}
```

SendBuffer 函数

声明：procedure SendBuffer(Buff: Array of char; length: integer);

这个函数把缓冲区中的数据发出去，length参数指定缓冲区的长度。程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
```

```
int BuffLen = 3;
char *Buff = "Hi";
NMUDP1->RemoteHost = Edit2->Text;
NMUDP1->RemotePort = StrToInt(Edit1.Text);
NMUDP1->SendBuffer(Buff, BuffLen);
}
```

SendStream 函数

声明：procedure SendStream(DataStream: TStream);
这个函数把DataStream参数指定的流中的数据发出去，程序示例请见RemotePort特性。

9.4 UDP控件的事件

UDP协议不需要建立连接，因此，UDP控件没有诸如OnAuthenticationFailed、OnConnect、OnConnectionRequest等事件。

OnBufferInvalid 事件

声明：property OnBufferInvalid: TBuffInvalid;
其中，TbuffInvalid是这样声明的：

```
TBuffInvalid = Procedure(var handled: boolean; var Buff: array of char; var length: integer);
```

接收数据时，如果缓冲区设得过小，将触发这个事件。在处理这个事件的句柄中，您可以调整缓冲区的大小，然后把handled参数设为True。

OnDataReceived 事件

声明：property OnDataReceived: TOnReceive;
其中，TOnReceive是这样声明的：

```
TOnReceive = procedure(Sender: TComponent; NumberBytes: Integer; FromIP: string);
```

当有新的数据到来时就会触发这个事件，NumberBytes参数是可以接收的数据长度，FromIP参数表明发送数据的IP地址。如果要接收数据，就调用ReadBuffer函数。

OnDataSend 事件

声明：property OnDataSend: TNotifyEvent;
当程序调用SendBuffer或SendStream函数发送数据时就会触发这个事件。

OnInvalidHost 事件

声明：property OnInvalidHost: THandlerEvent;
当RemoteHost特性所指定的主机名是非法的时，就触发这个事件。程序示例如下：

```
void _fastcall TForm1::NMUDP1InvalidHost(bool &Handled)
{
```

```
AnsiString TmpStr;
if (InputQuery("Invalid Host!", "Specify a new host:", TmpStr))
{
    NMUDP1->RemoteHost = TmpStr;
    Handled = true;
}
}
```

OnStreamInvalid 事件

声明： property OnStreamInvalid: TStreamInvalid;
调用ReadStream函数时，如果DataStream参数指定的流不存在，将触发这个事件。

```
    void _fastcall TForm1::NMUDP1StreamInvalid(bool &handled, TStream *Stream)
{
Stream->Write(MyString[1], Length(MyString));
handled = true;
}
```

第十章　使用HTTP控件

HTTP(Hypertext Transport Protocol，意为超文本传输协议)是Internet上最流行的协议，很多人对HTTP的细节可能不太了解，但事实上我们经常在与HTTP打交道，象大家都非常熟悉的Web浏览器，如Internet Explorer、Netscape Navigator等，它们的作用就是通过HTTP协议从Web服务器中请求HTML文档，然后在客户端解释HTML文档并转换为页面显示出来。现在的问题是，您也许并不想在应用程序中嵌入Web浏览器，也不希望在外部启动现成的浏览器程序，您只是想通过HTTP协议检索HTML文档，然后按您的要求去处理它，C++ Builder 3能做到这一点吗？能，完全能！C++ Builder 3从NetManage公司引进了TNMHTTP控件，用于在不需要浏览页面和图像处理的情况下检索HTTP文档。

10.1　HTTP控件的特性

Body特性

声明：property Body: string;
如果InputFileMode特性设为True，Body特性是HTML文档的文件名。如果

InputFileMode特性设为False，Body特性是检索到的数据实体。

BytesRecvd特性

声明：property BytesRecvd: longint;
这个特性返回已经接收到的数据长度，与BytesTotal特性配合使用可以显示传输进度。

BytesSent特性

声明：property BytesSent: longint;
这个特性返回已经向HTTP服务器发送的数据长度。

BytesTotal特性

声明：property BytesTotal: longint;
这个特性返回总共要传输多少个字节。程序示例如下：

```
void _fastcall TForm1::NMHTTP1PacketRecvd(TObject *Sender)
{
StatusBar1->SimpleText = IntToStr(NMHTTP1->BytesRecvd)+
" bytes of "+IntToStr(NMHTTP1->BytesTotal)+" received";
}
```

CookieIn特性

声明：property CookieIn: string;

这个特性包含了HTTP服务器发来的Cookie，关于Cookies的作用请查阅其它书籍。

Header 特性

声明：property Header: string;
如果InputFileMode特性设为True，Header特性包含的是文件名。如果InputFileMode特性设为False，Header特性包含的是实际检索到的头标信息。

HeaderInfo 特性

声明：property HeaderInfo: THeaderInfo;
这个特性用于指定要发送给HTTP服务器的头标信息，其中，THeaderInfo的Cookie特性指定要发送的Cookie，THeaderInfo的LocalMailAddress特性指定一个E-Mail地址，THeaderInfo的LocalProgram特性指定客户所使用的浏览器名称，THeaderInfo的Password特性用于给出口令，THeaderInfo的Referer特性用于指定要查询的文档的URL，THeaderInfo的UserID特性用于给出用户名。程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
NMHTTP1->TimeOut = 5000;
NMHTTP1->InputFileMode = false;
NMHTTP1->OutputFileMode = false;
NMHTTP1->ReportLevel = Status_Basic;
```

```
if (CheckBox1->Checked)
{
     NMHTTP1->Proxy = Edit11->Text;
     NMHTTP1->ProxyPort = StrToInt(Edit12->Text);
}
NMHTTP1->HeaderInfo->Cookie = Edit5->Text;
NMHTTP1->HeaderInfo->LocalMailAddress = Edit6->Text;
NMHTTP1->HeaderInfo->LocalProgram = Edit7->Text;
NMHTTP1->HeaderInfo->Referer = Edit8->Text;
NMHTTP1->HeaderInfo->UserId = Edit9->Text;
NMHTTP1->HeaderInfo->Password = Edit10->Text;
NMHTTP1->Get(Edit1->Text);
Memo1->Text = NMHTTP1->Body;
Memo2->Text = NMHTTP1->Header;
if (NMHTTP1->CookieIn != "") ShowMessage("Cookie:/n"+NMHTTP1->CookieIn);
}
```

Host 特性

声明：property Host: String;

这个特性用于指定要连接的HTTP服务器的主机名或IP地址。程序示例如下：

```
void _fastcall TForm1::NMHTTP1InvalidHost(bool &handled)
```

```
{
AnsiString NewHost;
if (InputQuery("Invalid Host", "Please Choose another host", NewHost))
{
      NMHTTP1->Host = NewHost;
      handled = true;
}
}
```

InputFileMode 特性

声明： property InputFileMode: boolean;
如果这个特性设为False，Body和Header特性包含的是实际的数据和头标。

LastErrorNo 特性

声明： property LastErrorNo: integer;
这个特性返回最近一次发生的错误的编号。程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
NMHTTP1->Get("http://www.borland.com");
StatusBar1->SimpleText = IntToStr(NMHTTP1->LastErrorNo);
}
```

LocalIP 特性

声明：property LocalIP: string;
这个特性返回客户计算机的IP地址(以点分法表示)。

OutputFileMode 特性

声明：property OutputFileMode: boolean;
如果这个特性设为False，要发送的数据可以直接从一个字符串中取出。

Port 特性

声明：property Port: Longint;
这个特性用于指定要连接的HTTP服务器的端口号，默认是80。

Proxy 特性

声明：property Proxy: String;
这个特性用于指定代理服务器的主机名或IP地址。如果不需要用代理服务器，这个特性必须空着。

ProxyPort 特性

声明：property ProxyPort: integer;
这个特性用于指定代理服务器的端口号，如果不用代理服务器，这个特性必须空着。

RemoteIP 特性

声明：property RemoteIP: string;
这个特性用于指定要连接的HTTP服务器的IP地址。

ReplyNumber 特性

声明：property ReplyNumber: Smallint;
这个只读的特性返回HTTP服务器响应客户请求的应答代码。程序示例如下：

```
void _fastcall TForm1::NMHTTP1Status(TObject *Sender, AnsiString Status)
{
if (StatusBar1 != 0)
{
    StatusBar1->SimpleText = Status;
    if (NMHTTP1->ReplyNumber = = 404)
        StatusBar1->SimpleText = "Object Not Found";
}
}
```

ReportLevel 特性

声明：property ReportLevel: integer;
这个特性用于指定状态信息的详细程度，可以设为以下值：Status_None、Status_Info-rmational、Status_Basic、Status_Routines、Status_Debug、

Status_Trace。

SendHeader 特性

声明： property SendHeader: string;
这个特性包含了要发送给HTTP服务器的头标信息，您可以在处理OnAboutToSend事件的句柄中修改这个特性增加其它信息。程序示例如下：

```
void _fastcall TForm1::NMHTTP1AboutToSend(TObject *Sender)
{
NMHTTP1->SendHeader->Values["Server"] = "www.netmastersllc.com";
}
```

Status 特性

声明： property Status: String;
这个特性返回当前的状态信息。

Timeout 特性

声明： property TimeOut: Integer;
这个特性用于指定一个毫秒数，如果超过规定的时间Socket没有响应就触发异常。

TransactionReply 特性

声明： property TransactionReply: String;

这个特性返回上一次命令的执行结果，程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
NMHTTP1->Get("http://www.netmastersllc.com");
ShowMessage(NMHTTP1->TransactionReply);
}
```

10.2 HTTP控件的方法

HTTP控件的方法很少，这是因为它只是一个HTTP客户，只要拨通了ISP，就可以直接访问HTTP服务器，因此，它没有诸如Connect、Accept、Listen等函数。

Abort函数

声明：procedure Abort;
这个函数用于取消一个正在发出的请求，相当于IE上的Stop按钮。

Copy函数

声明：procedure Copy(URL1, URL2: string);
这个函数把URL1参数指定的文档复制到URL2参数指定的位置。程序示例如下：

```
void _fastcall 1.BitBtn2Click(TObject *Sender)
```

```
{
NMHTTP1->Copy("http://www.crl.com/news/nfront.htm",
                    "http://www.crl.com/news/nback.htm");
}
```

注意：有的HTTP服务器可能不支持这项功能。

Delete 函数

声明：procedure Delete(URL: string);

这个函数删除由URL参数指定的文档。程序示例如下：

```
void _fastcall TForm1::BitBtn2Click(TObject *Sender)
{
NMHTTP1->Delete("http://www.crl.com/news/nfront.htm");
}
```

Get 函数

声明：procedure Get(URL: string);

这个函数用于从HTTP服务器检索由URL参数标识的文档。如果InputFileMode特性设为True，接收到的数据保存在由Body特性指定的文件中，如果InputFileMode特性设为False，接收到的数据作为字符串保存在Body特性中。

Head 函数

声明：procedure Head(URL: string);
这个函数检索URL参数所指定的文档中的头标信息，程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
NMHTTP1->Head("http://www.netmastersllc.com");
Memo1->Text = NMHTTP1->Header;
}
```

Link 函数

声明：procedure Link(URL, link: string);
这个函数把Link参数指定的资源连接到URL参数指定的文档中，程序示例如下：

```
void _fastcall TForm1::BitBtn2Click(TObject *Sender);
{
NMHTTP1->Link("http://www.crl.com/news/nfront.htm",
                    "http://www.crl.com/news/house.gif");
}
```

注意：有的HTTP服务器可能不支持这项功能。

Move 函数

声明：procedure Move(URL1, URL2: string);
这个函数把URL1参数指定的文档移到URL2指定的位置。程序示例如下：

```
void _fastcall TForm1::BitBtn2Click(TObject *Sender)
{
NMHTTP1->Move("http://www.crl.com/news/nfront.htm",
                    "http://www.crl.com/news/nback.htm");
}
```

注意：有的HTTP服务器可能不支持这项功能。

Options 函数

声明：procedure Options(URL: string);
这个函数用于检索URL参数所指定的文档的选项。

Patch 函数

声明：procedure Patch(URL, PatchData: string);
这个函数用PatchData参数指定的数据修补URL参数指定的文档。程序示例如下：

```
void _fastcall TForm1::BitBtn2Click(TObject *Sender)
{
NMHTTP1->Patch("http://www.crl.com/~test/nfront.htm", "entity.stf");
```

```
}
```

Post 函数

声明：procedure Post(URL, PostData: string);
这个函数用于把PostData参数指定的数据传送到URL参数指定的位置。如果OutputFileMode参数设为True，PostData参数是一个文件名，如果OutputFileMode参数设为False，PostData参数是一个字符串。程序示例如下：

```
void _fastcall TForm1.Button1Click(Sender: TObject);
{
NMHTTP1->OutputFileMode = false;
NMHTTP1->Post("http://www.sitetopostto.com/site", "name=ed pass=smith");
}
```

UnLink 函数

声明：procedure UnLink(URL, link: string);
这个函数与Link函数正相反，用于取消URL参数所指定的文档和Link参数所指定的资源之间的连接关系。

10.3 HTTP 控件的事件

OnAboutToSend 事件

声明：property OnAboutToSend: TNotifyEvent;
当客户端将要把头标信息发向HTTP服务器时将触发这个事件。在处理这个事件的句柄中，您可以修改SendHeader特性。程序示例如下：

```
void _fastcall TForm1::NMHTTP1AboutToSend(TObject *Sender)
{
NMHTTP1->SendHeader->Values["Server"] = "www.netmasters.com";
}
```

OnConnect 事件

声明：property OnConnect: TNotifyEvent;
当客户已经成功地连接到HTTP服务器时将触发这个事件。

OnConnectionFailed 事件

声明：property OnConnectionFailed: TNotifyEvent;
当客户试图登录到HTTP服务器失败时将触发这个事件。

OnDisconnect 事件

声明：property OnDisconnect: TNotifyEvent;

当客户断开与HTTP服务器的连接时将触发这个事件。注意：在处理这个事件的句柄中，如果需要访问VCL元件的话，最好先判断VCL元件是否存在，程序示例如下：

```
void _fastcall TForm1::NMHTTP1Disconnect(TObject *Sender)
{
if (StatusBar1 != 0)
      StatusBar1->SimpleText = "disconnected";
}
```

OnFailure 事件

声明： property OnFailure: TFailureEvent;
其中，TFailureEvent是这样声明的：

```
TResultEvent = procedure(Cmd: CmdType);
```

当调用TNMHTTP控件的某个函数失败时将触发这个事件。Cmd参数表明调用的是哪个函数。程序示例如下：

```
void _fastcall TForm1::NMHTTP1Failure(CmdType Cmd)
{
switch(Cmd)
{
      case CmdGET: StatusBar1->SimpleText = "Get Failed"; break;
      case CmdOPTIONS: StatusBar1->SimpleText = "Options Failed"; break;
      case CmdHEAD: StatusBar1->SimpleText = "Head Failed"; break;
```

```
        case CmdPOST: StatusBar1->SimpleText = "Post Failed"; break;
        case CmdPUT: StatusBar1->SimpleText = "Put Failed"; break;
        case CmdPATCH: StatusBar1->SimpleText = "Patch Failed"; break;
        case CmdCOPY: StatusBar1->SimpleText = "Copy Failed"; break;
        case CmdMOVE: StatusBar1->SimpleText = "Move Failed"; break;
        case CmdDELETE: StatusBar1->SimpleText = "Delete Failed"; break;
        case CmdLINK: StatusBar1->SimpleText = "Link Failed"; break;
        case CmdUNLINK: StatusBar1->SimpleText = "UnLink Failed"; break;
        case CmdTRACE: StatusBar1->SimpleText = "Trace Failed"; break;
        case CmdWRAPPED: StatusBar1->SimpleText = "Wrapped Failed"; break;
}
}
```

OnHostResolved 事件

声明： property OnHostResolved: TNotifyEvent;
当HTTP服务器的主机名已找到(域名解析)就触发这个事件，如果解析没有成功，将触发OnInvalidHost事件。

OnInvalidHost 事件

声明： property OnInvalidHost: THandlerEvent;
当Host特性所指定的主机名是非法的时，就触发这个事件。程序示例如下：

```
void _fastcall TForm1::NMHTTP1InvalidHost(bool &handled)
```

```
{
AnsiString NewHost;
if (InputQuery("Invalid Host", "Please Choose another host", NewHost))
{
     NMHTTP1->Host = NewHost;
     handled = true;
}
}
```

OnPacketRecvd 事件

声明：property OnPacketRecvd: TNotifyEvent;
客户每收到一个数据块就会触发这个事件，在处理这个事件的句柄中，您可以用BytesTotal特性和BytesRecvd特性显示接收的进度。

OnPacketSent 事件

声明：property OnPacketSent: TNotifyEvent;
客户每发出一个数据块就会触发这个事件，在处理这个事件的句柄中，您可以用BytesTotal特性和BytesRecvd特性显示发送的进度。

OnRedirect 事件

声明：property OnRedirect: THandlerEvent;
其中，THandlerEvent是这样声明的：

THandlerEvent = Procedure(var handled: boolean) of Object;
当您试图检索一个URL发生重定向时将触发这个事件，如果把handled参数设为False，就从新的URL处检索文档。

OnStatus 事件

声明： property OnStatus: TOnStatus;
其中，TOnStatus是这样声明的：

```
TOnStatus = procedure(Sender: TComponent; Status: string) of Object;
```

每次当HTTP服务器返回状态信息时将触发这个事件，Status参数就是当前的状态。

```
void _fastcall TForm1::NMHTTP1Status(TObject *Sender, AnsiString Status)
{
if (StatusBar1 != 0)
{
      StatusBar1->SimpleText = Status;
      if (NMHTTP1->ReplyNumber == 404)
            StatusBar1->SimpleText = "Object Not Found";
}
}
```

OnSuccess 事件

声明： property OnSuccess: TSuccessEvent;

其中，TSuccessEvent是这样声明的：

```
TResultEvent = procedure(Cmd: CmdType);
```

当某个命令被成功执行后将触发这个事件，Cmd 参数就是被执行的命令。

第十一章　使用HTML控件

上一章我们介绍了怎样通过HTTP控件传输HTML文档，并对HTML文档作了简单的分析和处理。这一章我们要介绍C++ Builder 3从NetManage公司引进的HTML控件，用这个控件可以建立一个强大的HTML语法分析器和页面显示器，这样，无需从外部启动一个现成的浏览器程序，应用程序直接就能浏览Web页面，够酷的吧？

11.1　HTML控件概述

在HTML控件中，要检索HTML文档有四种方式，一是用URL特性指定要检索的文档然后调用RequestDoc，此时将触发OnDoRequestDoc事件。二是在Web页面中单击超级链接，HTML会自动提取出该超级链接的URL，此时将触发OnDoRequestDoc事件。三是选择嵌入的文档，此时将触发OnDoRequestEmbedded事件。四是在窗体上单击“提交(Submission)”按钮，此时将触发OnDoRequestSubmit事件。

HTML控件的功能非常强大，支持所选页面的滚动视图、支持GIF、JPEG、BMP、XBM等格式的图像、支持HTML 2.X及大多数NetScape 2.0和Explorer 2.0扩展、支持用HTTP和文件URL检索文档、允许修改显示风格包括字体和

颜色等、支持DocStream接口用于进行复杂的传输、用事件重载默认的处理、支持页面打印。当然，与IE或Netscape相比，HTML控件还是有些欠缺的，它不支持文本选择和剪贴板操作，不支持代理服务器、不支持FTP功能、没有安全措施、不支持多部分文档提交。

HTML控件也可以只作为非可视的HTML语法分析器，只要把Visible特性设为False。在ElemNotification特性设为True的情况下，每当一个HTML元素要分析的时候就会触发OnDoNewElement事件，在处理OnDoNewElement事件的句柄中把EnableDefault参数设为False，这是为了不再对HTML元素作进一步的处理(因为不需要显示)，这样就大大加快了分析的速度。关于OnDoNewElement事件后面将详细介绍。

11.2 HTML控件的特性

BackColor特性

声明：property BackColor: TColor;

这个特性用于设置默认的背景颜色，RGB值的范围从0到0xFFFFFF。注意：如果UseDocColors特性设为True的话，文档的背景颜色就由DocBackColor特性决定。

BackImage特性

声明：property BackImage: WideString;

这个特性用于指定一个图像的URL，该图像作为文档的背景图案。注意：如果UseDocColors特性设为True，文档的背景图案由文档中的<BODY BACKGROUND=Λ>标记决定。该图案将自动平铺添满整个HTML控件的窗口。

BaseURL 特性

声明：property BaseURL: WideString;
这个只读的特性返回文档中<BASE>标记的值，如果文档中没有<BASE>标记，此特性就返回URL特性的值。

Blocking 特性

声明：property Blocking: WordBool;
如果把这个特性设为True(默认是False)，调用HTML控件的任何一个方法将等到完全执行完毕后才返回，类似于WinSock编程中的“阻塞”。

BlockResult 特性

声明：property BlockResult:(icBlockOK,IcTimedOut,IcErrorExit,IcBlockCancel,IcUserQuit);
这个只读的特性返回上次“阻塞”调用的返回值，可以是下列值之一：icBlockOK(上次调用成功)、IcTimedOut(上次调用超时)、IcErrorExit(上次调用中途出现错误)、IcBlockCancel(上次调用中途被取消)、IcUserQuit(因为用户关闭了程序)。

DeferRetrieval 特性

声明：property DeferRetrieval: WordBool;
如果此特性设为True(默认是False)，除非显式地请求，否则不下载文档中嵌入的对象。

DocBackColor 特性

声明：property DocBackColor: TColor;
这个只读的特性返回文档中<BODY>标记的BGCOLOR属性，也就是文档的背景颜色。如果<BODY>标记中没有BGCOLOR属性，此特性就返回BackColor特性的值。

DocForeColor 特性

声明：property DocForeColor: TColor;
这个只读的特性返回文档中<BODY>标记的TEXT属性，也就是文档的前景(文本)颜色。如果<BODY>标记中没有TEXT属性，此特性就返回ForeColor特性的值。

DocInput 特性

声明：property DocInput: DocInput;
这个只读的特性返回一个DocInput对象。

DocLinkColor 特性

声明：property DocLinkColor: TColor;
这个只读的特性返回文档中<BODY>标记的LINK属性，也就是文档中未访问过的链接的颜色。如果<BODY>标记中没有LINK属性，此特性就返回LinkColor特性的值。

DocOutput 特性

声明：property DocOutput: DocOutput;
这个只读的特性返回一个DocOutput对象。

DocVisitedColor 特性

声明：property DocVisitedColor: TColor;
这个只读的特性返回文档中<BODY>标记的VLINK属性，也就是已访问过的链接的颜色。如果<BODY>标记中没有VLINK属性，此特性就返回VisitedColor特性的值。

Errors 特性

声明：property Errors: icErrors;
这个只读的特性返回最近一次发生的错误(icErrors对象)。

ElemNotification 特性

声明：property ElemNotification: WordBool;

如果您需要把HTML控件作为HTML语法分析器，您应当把这个特性设为True，当一个新的HTML元素需要分析时就会触发OnDoNewElement事件。

FixedFont特性

声明：property FixedFont: TFont;
这个特性用于设置等宽文本的字体，默认采用“Courier New”，字号为10。

Font特性

声明：property Font: TFont;
这个特性用于设置非等宽文本的字体，默认采用“Times New Roman”，字号是12。

ForeColor特性

声明：property ForeColor: TColor;
这个特性用于设置默认的前景颜色，RGB值的范围从0到0xFFFFFF。注意：如果UseDocColors特性设为True的话，文档的前景颜色就由DocForeColor特性决定。

Forms特性

声明：property Forms: HTMLForms;
这个只读的特性返回文档中的窗体(后面将介绍HTMLForms和HTMLForms对象)。

HasSelection 特性

声明：property HasSelection: WordBool;
这个只读的特性返回是否选择了文本。

Heading1Font 特性

声明：property Heading1Font: TFont;
这个特性用于设置包括在一对<H1>标记之间的文本(即一级标题)的字体，默认采用“Times New Roman”，字号是24，加粗。

Heading2Font 特性

声明：property Heading2Font: TFont;
这个特性用于设置包括在一对<H2>标记之间的文本(即二级标题)的字体，默认采用“Times New Roman”，字号是18，加粗。

Heading3Font 特性

声明：property Heading3Font: TFont;
这个特性用于设置包括在一对<H3>标记之间的文本(即三级标题)的字体，默认采用“Times New Roman”，字号是14，加粗。

Heading4Font 特性

声明：property Heading4Font: TFont;
这个特性用于设置包括在一对<H4>标记之间的文本(即四级标题)的字体，

默认采用“Times New Roman”，字号是12，加粗。

Heading5Font特性

声明：property Heading5Font: TFont;
这个特性用于设置包括在一对<H5>标记之间的文本(即五级标题)的字体，默认采用“Times New Roman”，字号是10，加粗。

Heading6Font特性

声明：property Heading6Font: TFont;
这个特性用于设置包括在一对<H6>标记之间的文本(即六级标题)的字体，默认采用“Times New Roman”，字号是8，加粗。

hWnd特性

声明：property hWnd: OLE_HANDLE;
这个只读的特性返回HTML控件的窗口句柄，这样就可以传递WM_COPY等消息。

IsPrintingDone特性

声明：property IsPrintingDone[PageNumber: Integer]: WordBool;
如果文档中指定的页码(PageNumber参数)已经被打印，这个只读的特性就返回True。

LayoutDone 特性

声明：property: WordBool;
如果文档的总体布局已显示出来，嵌入的对象还未下载，这个只读的特性就返回True。

LinkColor 特性

声明：property LinkColor: TColor;
这个特性用于设置未访问过的链接颜色，RGB值的范围从0到0xFFFFFF。如果UseDocColors特性设为True，链接的颜色由DocLinkColor特性决定。

ParseDone 特性

声明：property ParseDone: WordBool;
开始检索文档的时候，此特性为False。当整个文档分析完毕后，这个特性就返回True。

Redraw 特性

声明：property Redraw: WordBool;
如果这个特性设为True，当文档中的数据发生变化或窗口被翻滚时就重画HTML控件的窗口，不过，这可能造成屏幕老是闪烁。

RequestURL 特性

声明：property RequestURL: WideString;

这个只读的特性返回当前要请求的文档的URL，由RequestDoc的URL参数设定。

RetainSource 特性

声明：property RetainSource: WordBool;
如果这个特性设为True，将保存文档的源代码(可以由SourceText特性访问)。如果不需要文档的源代码，最好把这个特性设为False，以节省内存。

RetrieveBytesDone 特性

声明：property RetrieveBytesDone: Integer;
这个只读的特性返回已经检索到文档的多少个字节。

RetrieveBytesTotal 特性

声明：property RetrieveBytesTotal: Integer;
这个只读的特性返回要检索的文档总共有多少个字节。如果DeferRetrieval特性设为True，返回的值不包括文档中嵌入的对象。

SleepTime 特性

声明：property SleepTime: Longint;
这个特性仅用于“阻塞”方式(指Blocking特性设为True)，用于指定检索消息的时间间隔，默认是10毫秒。

SourceText 特性

声明：property SourceText: WideString;
如果RetainSource特性设为True，可以由SourceText特性得到文档的源代码。

TimeOut 特性

声明：property Timeout: Integer;
这个特性用于指定一个时间间隔(以秒为单位)，默认是30秒。在规定的时间间隔内，如果没有收到数据就触发OnTimeout事件(仅对主文档有效，不适用于嵌入的文档)。

TotalHeight、TotalWidth 特性

声明：property TotalHeight, TotalWidth: Integer;
这两个只读的特性分别返回文档的高度和宽度(以像素为单位)，包括因为窗口太小未显示出来的部分。注意：在对文档的分析和布局函数中，这两个特性的值实际上是变化的，当文档分析完毕后(此时将触发OnEndRetrieval事件)，这两个特性的值才固定下来。

UnderlineLinks 特性

声明：property UnderlineLinks: WordBool;
如果这个特性设为True，文档中的链接加上下划线。

URL 特性

声明：property URL: WideString;
当调用RequestDoc并且传递了合法的URL时，就开始检索文档，此时将触发OnBeginRetrieval事件，这时候URL特性就返回正在检索的文档的URL加上端口号。可见，URL特性与RequestURL特性还是有区别的。

UseDocColors 特性

声明：property UseDocColors: WordBool;
如果这个特性设为True，就用文档本身的前景、背景、链接的颜色取代BackColor、ForeColor、LinkColor特性设置的颜色。

ViewSource 特性

声明：property ViewSource: WordBool;
如果这个特性设为True，HTML控件将以源代码的形式显示文档。如果RetainSource特性设为False，HTML控件会自动去检索文档的源代码。

VisitedColor 特性

声明：property VisitedColor: TColor;
这个特性用于设置访问过的链接的颜色，RGB值的范围从0到0xFFFFFF。如果UseDocColors特性设为True，访问过的链接的颜色由DocVisitedColor特性决定。
最后简单介绍一下HTML控件的特性页，如图11.1所示：

图 11.1　HTML 控件的特性页

11.3　HTML 控件的方法

AboutBox 函数

声明：procedure AboutBox;

这个函数用于显示HTML控件的About框，如图11.2所示。

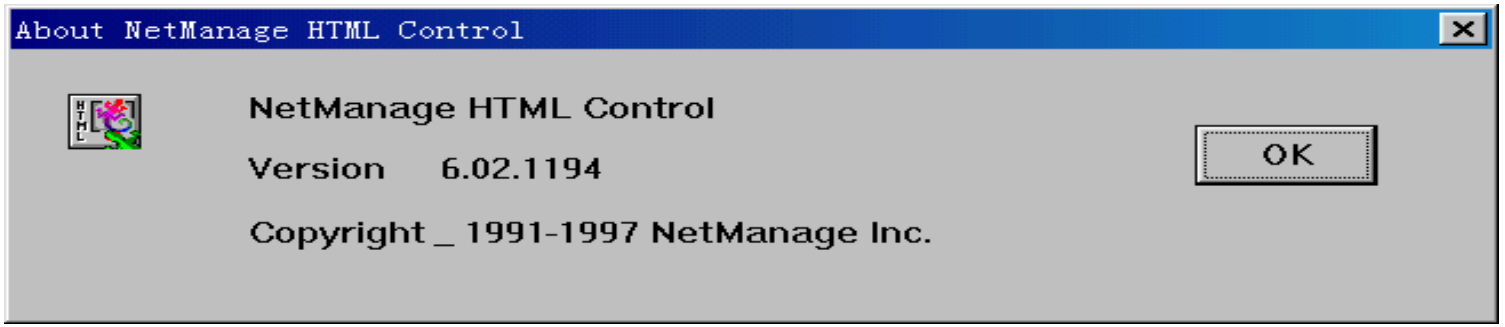


图 11.2 HTML 控件的 About 框

AutoPrint 函数

声明：procedure AutoPrint(hDC: Integer);
这个函数用于打印整个文档并自动插入分页符，其中，hDC参数用于指定打印机的设备描述表，程序示例如下：

```
Printer( )->BeginDoc( );
HTML1->AutoPrint(Printer( )->hDC);
Printer( )->EndDoc;
```

BeginPrinting 函数

声明：procedure BeginPrinting(hDC: Integer; X, Y, Width, Height, DefaultHeaders, DefaultTitle: OleVariant);
这个函数用于在打印前进行设置，其中，hDC参数是打印机的设备描述表，X和Y参数用于设置要打印的区域的起点坐标，Width和Height参数用于要打印的区域的宽度和高度，DefaultHeader和DefaultTitle参数目前暂时不用。程

序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
bool DonePrinting;
Printer( )->BeginDoc( );
HTML1->BeginPrinting(Printer( )->hDC);          //其余参数省略
int Page = 1;
while(DonePrinting = False)
{
      HTML1->PrintPage(Printer( )->hDC, Page);
      Page = Page+1
      DonePrinting = HTML1->IsPrintingDone(Page);
      Printer( )->NewPage( );
}
HTML1->EndPrinting( );
Printer( )->EndDoc( );
};
```

Cancel函数

声明： procedure Cancel(Message: OleVariant);
这个函数用于取消当前的文档检索(包括嵌入的文档)，然后输出一个消息。

EndPrinting 函数

声明：procedure EndPrinting;
这个函数用于结束对打印的设置，下一步应当调用Printer对象的EndDoc函数开始打印文档，程序示例请参见BeginPrinting函数。

GetPlainText 函数

声明：function GetPlainText(Selected, Fancy: WordBool): WideString;
这个函数用于返回文档中当前选择的文本。如果Selected参数设为True，这个函数就只返回当前选择的文本。如果Selected参数设为False，这个函数就返回整个文本。如果Fancy参数设为True，文档中的水平线被转换为文本形式的点划线。

PrintPage 函数

声明：procedure PrintPage(hDC, PageNumber: Integer);
这个函数用于打印文档中指定的页，hDC参数是打印机的设备描述表，PageNumber参数指定要打印的页码。

RequestAllEmbedded 函数

声明：procedure RequestAllEmbedded;
这个函数请求下载所有嵌入的文档，如调用成功，将触发OnDoRequestEmbedded事件。

RequestDoc 函数

声明：procedure RequestDoc(const URL: WideString);
这个函数很重要，用于检索URL参数指定的文档，并触发OnDoRequestDoc事件，同时，RequestURL特性被设为URL参数的值。如果URL参数指定的路径是有效的，就开始检索文档并触发OnBeginRetrieval事件，此时，URL特性就被设为URL参数的值。

SelectAll 函数

声明：procedure SelectAll;
调用这个函数将选择HTML控件的窗口中所有的文本。

11.4 HTML 控件的事件

目前为止，本书介绍的Internet ActiveX控件中只有HTML控件是可视的，也就是说，在运行期是可以看见的，其它均是非可视的。因此，HTML控件具有诸如OnClick、OnKeyDown、OnMouseDown等事件，用于处理用户在窗口上的操作。

OnBeginRetrieval 事件

声明：property OnBeginRetrieval: TNotifyEvent;
当开始检索HTML文档时就触发这个事件。如果用一个进程条(TProgressBar

元件)来显示下载进度的话，此时正好初始化这个进程条。

OnClick 事件

声明：property OnClick: TNotifyEvent;
当用户在窗口上单击鼠标左键时就触发这个事件。

OnDblClick 事件

声明：property OnDblClick: TNotifyEvent;
当用户在窗口上双击鼠标时将触发这个事件。

OnDocInput 事件

声明：property OnDocInput: procedure(Sender: TObject; const DocInput: DocInput);
当DocInput对象的状态发生变化或新的数据块被传输时就触发这个事件。在处理这个事件的句柄中，可以由DocInput参数返回DocInput对象，进而访问DocInput对象的特性和方法。

OnDocOutput 事件

声明：property OnDocOutput: procedure(Sender: TObject; const DocOutput: DocInput);
当DocOutput对象的状态发生变化或新的数据块被传输时就触发这个事件。在处理这个事件的句柄中，可以由DocOutput参数返回DocOutput对象，进

而访问DocOutput对象的特性和方法。

OnDoNewElement事件

声明：property OnDoNewElement: procedure(Sender: TObject; const ElemType: WideString; EndTag:WordBool;const Attrs:HTMLAttrs;const text:WideString; var EnableDefault: WordBool);
当一个新的HTML元素将要分析时就触发OnDoNewElement事件。如果要分析的HTML元素是文档中的文本，ElemType参数为空，而Text参数就是该文本。如果是标记，ElemType参数就是标记类型，此时可以通过Attrs参数(HTMLAttrs对象)得到标记中的属性，而Text参数则为空。如果有一个结束标记的话，EndTag参数就为True，否则就为False。
EnableDefault参数很重要，如果HTML控件只是作为非可视的语法分析器，您应当把这个参数设为False，这样就不再对该元素作进一步的处理。

OnDoRequestDoc事件

声明：property OnDoRequestDoc: procedure(Sender: TObject; const URL: WideString; const Element: HTMLElement; const DocInput: DocInput; var EnableDefault: WordBool);
当调用了RequestDoc(不管URL参数是否有效)或者在窗口中单击某个链接时将触发这个事件，此时，RequestURL参数自动设为URL参数是值或链接的URL。如果要检索的URL是有效的，就开始检索并触发OnBeginRetrieval事件，URL特性设为正在检索的URL。

在OnDoRequestDoc事件中，URL参数是要检索的URL，Element参数目前暂时不用，DocInput参数返回一个DocInput对象。EnableDefault参数的含义见OnDoNewElement事件。

OnDoRequestEmbedded 事件

声明：property OnDoRequestEmbedded: procedure(Sender: TObject; const URL: WideString; const Element: HTMLElement; const DocInput: DocInput; var EnableDefault: WordBool);
如果DeferRetrieval特性设为False，检索文档时要自动检索其中嵌入的文档。如果DeferRetrieval特性设为True，只有调用了RequestAllEmbedded函数才检索嵌入的文档。此事件就是在检索嵌入的文档时触发的，其中，URL参数是嵌入文档的URL，Element参数目前不用，DocInput参数返回一个DocInput对象，EnableDefault参数的含义见上。

OnDoRequestSubmit 事件

声明：property OnDoRequestSubmit: procedure(Sender: TObject; const URL: WideString; const Form: HTMLForm; const DocOutput: DocOutput; var EnableDefault: WordBool);
当用户单击了窗体上的“Submit(提交)”按钮或程序中调用了RequestSubmit时就会触发这个事件。对于目前这个版本的HTML控件来说，窗体上的内容是由用URL编码的字段的值组成的，这些值可以由HTMLForm对象的URLEncodedBody特性访问(后面将介绍HTMLForm对象)。如果窗体的提交

方法是GET而不是POST的话，这个事件的URL参数就是用户请求的动作。Form参数返回一个HTMLForm对象，就是正在提交的窗体。DocOutput参数返回一个DocOutput对象。如果要取消此次提交，就把EnableDefault参数设为False。如果把EnableDefault参数设为True，RequestURL特性将被设为URL参数的值，如果URL是有效的话，URL特性就被设为URL参数的值。

OnEndRetrieval事件

声明：property OnEndRetrieval: TNotifyEvent;
文档检索完毕就触发这个事件，如果用一个进程条显示下载进度的话，此时就是100%。

OnError事件

声明：property OnError:procedure(Sender:TObject;Number:Smallint;var Description:WideStr-ing;
Scode:Integer;const Source,HelpFile:WideString;HelpContext:Integer;var CancelDisplay:WordBool);
当在连接或传输数据的时候出错时就触发这个事件，其中，Number参数是错误代码，Description参数是关于错误的简短描述，Scode参数和Source参数的作用不详，HelpFile参数是帮助文件名，HelpContext是帮助的上下文编号，如果您把CancelDisplay参数设为True(默认)，表示不显示错误信息框。

OnGotFocus 事件

声明： property OnGotFocus: TNotifyEvent;
当HTML控件的窗口获得输入焦点时就触发这个事件，可能是因为用户直接用鼠标单击窗口，或者用TAB键把输入焦点切换到窗口中，也可能是因为程序中调用了SetFocus。

OnKeyDown 事件

声明： property OnKeyDown: procedure(Sender: TObject; var Key: Word; Shift: Tshift-State);
如果输入焦点在HTML控件的窗口中，当用户在窗口中按下一个键就触发这个事件，其中，KeyCode参数是所按键的虚拟码，Shift参数是一个集合，可包括下列元素：ssShift、ssAlt、ssCtrl、ssLeft、ssRight、ssMiddle、ssDouble。

OnKeyPress 事件

声明： Property OnKeyPress: Procedure (Sender: TObject; var Key: Char);
这个事件发生在用户按下了某个字符键，其中Key参数表示被按下的字符。

OnKeyUp 事件

声明： Property OnKeyUp: procedure(Sender: TObject; var Key: Word; Shift: TShiftState);
这个事件发生在用户释放了键，参数含义见OnKeyDown事件。

OnLayoutComplete 事件

声明：property OnLayoutComplete: TNotifyEvent;
当HTML主文档的布局已完成时将触发这个事件，不过，此时嵌入的文档可能还没有检索完毕，但至少每个嵌入文档的长度和位置已确定。

OnLostFocus 事件

声明：property OnLostFocus: TNotifyEvent;
当HTML控件的窗口失去输入焦点时就触发这个事件，可能是因为用户在其它窗口中单击鼠标，或者用TAB键把输入焦点切换到其它窗口中，也可能是因为调用了SetFocus。

OnMouseDown 事件

声明：Property OnMouseDown: TMouseEvent;
其中，TMouseEvent是这样声明的：

```
TMouseEvent = Procedure (Sender: TObject; Button: TMouseButton; Shift: TShiftState; X， Y: Integer) of object;
```

这个事件发生在用户在HTML控件的窗口上按下了鼠标，其中Button参数表示鼠标上哪个按钮被按下，其值可能是mbLeft、mbRight或mbMiddle。Shift参数表示是否有Shift、Ctrl和Alt键被按下，其值可能是ssShift、ssAlt、ssCtrl、ssLeft、ssRight、ssMiddle、ssDouble。

OnMouseMove 事件

声明：Property OnMouseMove: TMouseMoveEvent;
其中，TMouseMoveEvent是这样声明的：
TMouseMoveEvent =Procedure(Sender: TObject; Shift: TShiftState; X，Y: Integer) of object;
这个事件发生在用户在HTML控件的窗口上移动鼠标的时候，Shift参数的含义见上。

OnMouseUp 事件

声明：Property OnMouseUp: TMouseEvent;
其中，TMouseEvent是这样声明的：
TMouseEvent = Procedure (Sender: TObject; Button: TMouseButton; Shift: TShiftState; X，Y: Integer) of object;
当用户在HTML控件的窗口上释放鼠标时将触发这个事件。

OnParseComplete 事件

声明：property OnParseComplete: TNotifyEvent;
当整个HTML文档已分析完毕时将触发这个事件，此时，文档的布局可能还没有完成，嵌入的文档可能还没有检索完毕。

OnTimeOut 事件

声明：property OnTimeOut:procedure(Sender:TObject;event:Smallint;var

Continue:WordBool);
当某个事件在规定的时间内未被处理就会触发这个事件，其中，Event参数就是未被处理的事件，可以是以下值之一：prcConnectTimeout、prcReceiveTimeout、prcUserTimeout。您如果把Continue参数设为True，定时器继续处于活动状态。

OnUpdateRetrieval事件

声明：property OnUpdateRetrieval: TNotifyEvent;
这个事件类似于OnDocOutput事件，在检索文档的函数中将触发这个事件。

11.5 几个与HTML控件有关的对象

在介绍OnDoNewElement事件时曾经提到HTMLAttrs对象，此对象用于收集和管理HTML标记的属性，其Count特性返回标记中有几个属性，其Item函数用于返回其中一个属性(HTMLAttr对象)。HTMLAttr对象的Name特性返回属性名，Value特性返回属性的值。
前面在介绍OnDoRequestSubmit事件时曾经提到HTMLForm对象，它是由HTMLForms对象收集和管理的，与HTMLAttrs对象一样，HTMLForms对象的Count特性返回共有几个窗体，其中每个窗体可以由Item函数返回。下面我们重点介绍HTMLForm对象：

Method 特性

声明：property Method: WideString;
这个只读的特性表示用户提交窗体的方法是“GET”还是“POST”。

URL 特性

声明：property URL: WideString;
如果用户提交窗体的方法是“GET”，这个只读的特性返回用户请求的动作。

URLEncodedBody 特性

声明：property URLEncodedBody: WideString;
通过这个只读的特性可以访问窗体中每个字段的值。

RequestSubmit 函数

声明：procedure RequestSubmit;
这个函数用于请求提交一个窗体，并触发OnDoRequestSubmit事件。

第十二章　使用SMTP控件

E-Mail(电子函件)是Internet上应用最广泛的服务，它为我们提供了方便、快捷、安全而又廉价的通信手段，相信读者对此深有体会。用于收发电子函件的软件也很多，比较著名的有Outlook Express、Eudora Pro、方正飞扬等。这里要讲述的是，如何通过编程在应用程序中收发电子函件而不需要借助于专门的邮件客户程序。

与E-Mail密切相关的两个协议是POP3和SMTP，POP3是Post Office Protocol 3的缩写，用于从“邮局”(POP3服务器)接收电子函件，SMTP是Simple Mail Transfer Protocol的缩写，用于通过SMTP服务器发送电子函件。这两个协议本身是很复杂的，好在C++ Builder 3从NetManage公司引进了POP控件和SMTP控件，使您不需要知道协议的细节，就可以方便地实现这两个协议。本章先介绍SMTP控件，读者应从中掌握怎样登录到SMTP服务器、怎样发送邮件。

12.1 SMTP 控件的特性

EncodeType 特性

声明：property EncodeType: UUMethods;
这个特性用于指定附件的编码方式，设为uuMIME 表示按MIME编码，设为uuCode 表示按UUEncoding编码。

FinalHeader 特性

声明：property FinalHeader: String;
这个特性用于指定最后的消息头标。

Host 特性

声明：property Host: String;
这个特性用于指定SMTP服务器的主机名或IP地址如206.155.164.20。程序示例如下：

```
void _fastcall TForm1::ConnectBtnClick(TObject *Sender)
{
NMSMTP1->ClearParams = true;
NMSMTP1->Host = "inc-net.com";
NMSMTP1->Port = 25;
NMSMTP1->Connect( );
```

```
}
```

LocalIP 特性

声明：property LocalIP: String;
这个特性返回客户计算机的IP地址(点分法)。如果计算机有多个IP地址，这个特性返回其中第一个。程序示例如下：

```
void _fastcall TForm1::ShowIPClick(TObject *Sender)
{
ShowMessage("Your Local IP Address is: "+NMSMTP1->LocalIP);
}
```

Port 特性

声明：property Port: Longint;
这个特性用于指定要连接的SMTP服务器的端口号，默认是25。

PostMessage 特性

声明：property PostMessage: TPostMessage;
这个特性很重要，用于设置要张贴的消息。TPostMessage是一个类，其中：

- FromName　　用于指定发件人的名称
- FromAddress　　用于指定发件人的E-Mail地址
- ReplyToAddress 用于指定回复地址
- Organization　　用于指定发件人所在的组织

- LocalProgram　　用于指定发件人所使用的邮件客户程序
- ToAddress　　这是个TStringList对象，用于指定收件人的E-Mail 地址
- ToCC　　这是个TStringList对象，用于指定要抄送的E-Mail 地址
- ToBCC　　这是个TStringList对象，用于指定要密件抄送的E-Mail 地址
- Attachments　　这是个TStringList对象，用于指定附件
- Body　　这是个TStringList对象，用于指定邮件正文
- Subject　　用于指定邮件的主题
- Date　　用于指定邮件的发送日期，如空着，表示是当前日期

程序示例如下：

```
void _fastcall TForm1::SendBtnClick(TObject *Sender)
{
NMSMTP1->PostMessage->FromAddress = LAddress->Text;
NMSMTP1->PostMessage->FromName = LName->Text;
NMSMTP1->PostMessage->ToAddress->Add(ToField->Text);
NMSMTP1->PostMessage->ToCarbonCopy->Add(CCField->Text);
NMSMTP1->PostMessage->Body->AddStrings(BodyField->Lines);
NMSMTP1->PostMessage->Attachments->AddStrings(AttachmentsField->Lines);
NMSMTP1->PostMessage->Subject = SubjectField->Text;
NMSMTP1->PostMessage->LocalProgram = "NMSMTP Demo";
NMSMTP1->SendMail( );
```

```
}
```

ReplyNumber 特性

声明：property ReplyNumber: Longint;
这个只读的特性返回SMTP服务器响应客户请求的应答代码。程序示例如下：

```
void _fastcall TForm1::NMSMTP1Status(TObject *Sender, AnsiString status)
{
if (StatusBar1 != 0)
      if (NMSMTP1->ReplyNumber == 220)
            StatusBar1->SimpleText = "Service Ready";
}
```

Status 特性

声明：property Status: String;
这个特性返回当前的状态信息。

Timeout 特性

声明：property TimeOut: Integer;
这个特性用于指定一个毫秒数，如果超过规定的时间Socket没有响应就触发异常。

TransactionReply 特性

声明：property TransactionReply: String;
这个特性返回上一次命令的执行结果，程序示例如下：

```
void _fastcall TForm1::NMSMTP1Status(TComponent*Sender, AnsiString status)
{
if (StatusBar1 != 0) StatusBar1->SimpleText = NMSMTP1->TransactionReply;
}
```

UserID 特性

声明：property UserID: string;
这个特性用于指定用户名，因为有的SMTP服务器需要给出用户名。程序示例如下：

```
    void _fastcall TForm1::ConnectBtnClick(TObject *Sender)
{
NMSMTP1->Host = "206.155.164.20";
NMSMTP1->UserID = "Ed";
NMSMTP1->Connect( );
}
```

12.2 SMTP控件的方法

Abort函数

声明：procedure Abort;
这个函数用于中止此次发送，并且断开与SMTP服务器的连接。

ClearParameters函数

声明：procedure ClearParameters;
在填充PostMessage特性前，一般要调用这个函数把PostMessage特性清空。

Connect函数

声明：procedure Connect; override;
这个函数用于登录到SMTP服务器，Host特性和Port特性必须事先已设置，如果主机名是非法的，将触发OnInvalidHost事件。如果连接失败，将触发OnConnectionFailed事件。

Disconnect函数

声明：procedure Disconnect; override;
这个函数用于断开与SMTP服务器的连接，并且将触发OnDisconnect事件。

ExpandList 函数

声明：procedure ExpandList(ListName: String);
这个函数在SMTP服务器上检索一个邮件列表，如果有的话，通过OnMailListReturn事件可以取得这个邮件列表。

SendMail 函数

声明：procedure SendMail;
这个函数用于向SMTP服务器发送邮件，您必须事先已设置Host、Port、PostMessage等特性。

Verify 函数

声明：function Verify(User: String) : Boolean;
这个函数用于校验某个用户名是否合法，如合法，这个函数就返回True。

12.3 SMTP 控件的事件

SMTP控件的事件往往是与调用某个方法相关联的，如果事件被触发，通常表示调用成功，您可以在处理事件的句柄中得到SMTP服务器的返回信息。

OnAttachmentNotFound 事件

声明：property OnAttachmentNotFound: TFileItem;

其中，TFileItem是这样声明的：

```
TFileItem    = procedure (Filename:string) of object;
```

发送邮件时，如果PostMessage特性所指定的附件没有找到就会触发这个事件。

OnAuthenticationFailed 事件

声明： property OnAuthenticationFailed: THandlerEvent;

其中，THandlerEvent是这样声明的：

```
THandlerEvent = Procedure(var handled: boolean) of Object;
```

有的SMTP服务器需要给出用户名，如果没有事先给出用户名或者用户名是非法的，将会触发这个事件。更正了错误后，如果把Handled参数设为True，将再连接一遍。

```
void _fastcall TForm1::NMSMTP1AuthenticationFailed(bool &handled)
{
AnsiString   TmpStr;
if (InputQuery("Authentication Needed", "Enter UserID: ",TmpStr))
{
     handled = true;
     NMSMTP1->UserID = TmpStr;
}
}
```

OnConnect 事件

声明：property OnConnect: TNotifyEvent;
当客户已经成功地连接到SMTP服务器时将触发这个事件。

OnConnectionFailed 事件

声明：property OnConnectionFailed: TNotifyEvent;
当客户试图登录到SMTP服务器失败时将触发这个事件。

OnDisconnect 事件

声明：property OnDisconnect: TNotifyEvent;
当客户断开与SMTP服务器的连接时将触发这个事件。注意：在处理这个事件的句柄中，如果需要访问VCL元件的话，最好先判断VCL元件是否存在。

OnEncodeStart 事件

声明：property OnEncodeStart: TFileItem;
当一个附件将要被编码时将触发这个事件，FileName参数就是被编码的附件。

OnEncodeEnd 事件

声明：property OnEncodeEnd: TFileItem;
当一个附件被编码完毕时将触发这个事件，FileName参数就是被编码的附件。

OnFailure 事件

声明：property OnFailure: TNotifyEvent;
如果邮件由于某种原因没有被成功发送，将触发这个事件。

OnHeaderIncomplete 事件

声明：property OnHeaderIncomplete: THeaderInComplete;
其中，THeaderInComplete是这样声明的：

```
THeaderInComplete = procedure( handled:boolean ; hiType : integer ) of Object;
```

发送邮件时，如果PostMessage特性中给出的头标信息不完整，将触发这个事件。程序示例如下：

```
void _fastcall TForm1::NMSMTP1HeaderIncomplete(bool &handled int hiType)
{
switch(hiType)
{
      case hiToAddressMissing: ShowMessage("Recipient field missing!"); break;
      case hiFromAddressMissing: ShowMessage("Sender field missing!"); break;
}
}
```

OnHostResolved 事件

声明：property OnHostResolved: TNotifyEvent;
当SMTP服务器的主机名已找到(域名解析)就触发这个事件，如果解析没有

成功，将触发OnInvalidHost事件。

OnInvalidHost事件

声明：property OnInvalidHost: THandlerEvent;
当Host特性所指定的主机名是非法的，就触发这个事件。程序示例如下：

```
void _fastcall TForm1::NMSMTPInvalidHost(TObject *Sender, bool &handled)
{
if (InputQuery("Invalid Host","Enter a new host: ",TmpStr))
{
     handled = true;
     NMSMTP1->Host = TmpStr;
}
}
```

OnMailListReturn事件

声明：property OnMailListReturn: TMailListReturn;
其中，TMailListReturn是这样声明的：

```
TMailListReturn = procedure (MailAddress:string) of object;
```

当程序调用ExpandList函数检索邮件列表时，每检索到一项(E-Mail地址)就会触发这个事件，程序示例如下：

```
void _fastcall TForm1::NMSMTP1MailListReturn(AnsiString MailAddress)
{
```

```
AddressList->Lines->Add(MailAddress);
}
```

OnRecipientNotFound 事件

声明： property OnRecipientNotFound: TRecipientNotFound;
其中，TRecipientNotFound是这样声明的：

```
TRecipientNotFound = procedure( Recipient : string ) of Object;
```

如果PostMessage特性中指定的收件人不存在，将触发这个事件。

OnSendStart 事件

声明： property OnSendStart: TNotifyEvent;
邮件将要被发送时将触发这个事件，这样您就有机会最后修改FinalHeader特性。

第十三章　使用POP控件

C++ Builder 3从NetManage公司引进了POP控件，用于从支持POP3协议的服务器包括运行UNIX的主机检索邮件。POP控件支持身份验证登录，能够检索用户邮箱信息如邮箱中有几封信等，POP控件还能够直接在服务器上删除邮件。

13.1　POP控件的特性

AttachFilePath特性

声明：property AttachFilePath: string;
这个特性用于指定存放附件的路径，如果设为空，表示是当前目录。

BytesRecvd特性

声明：property BytesRecvd: longint;
这个特性返回已接收了多少个字节，与BytesTotal特性配合使用可以显示传输的进度。

BytesTotal 特性

声明：property BytesTotal: longint;
这个特性返回本次传输总共有多少个字节。

Connected 特性

声明：property Connected: boolean;
如果已经与POP3服务器连接，这个特性就返回True。程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject*Sender)
{
if (NMPOP31->Connected != true)
{
     NMPOP31->Port = 110;
     NMPOP31->UserID = "Ed";
     NMPOP31->Password = "Smith";
     NMPOP31->Connect( );
     Label10->Caption = "# of Messages: "+IntToStr(NMPOP31->MailCount);
}
}
```

DeleteOnRead 特性

声明：property DeleteOnRead: boolean;
如果这个特性设为True，当邮件被下载后就从服务器中删掉。

Host 特性

声明：property Host: String;
这个特性用于指定POP3服务器的主机名或IP地址，如206.155.164.20。

LocalIP 特性

声明：property LocalIP: String;
这个特性返回客户计算机的IP地址(点分法)。如果计算机有多个IP地址，这个特性返回其中第一个。程序示例如下：

```
    void _fastcall TForm1::Button9Click(TObject *Sender)
{
ShowMessage("Your local IP Address is "+NMPOP31->LocalIP);
}
```

MailCount 特性

声明：property MailCount: Smallint;
这个只读的特性返回用户邮箱中有几封信。注意：此特性只有在通过了身份验证之后才是有效的，此前访问这个特性是非法的。

MailMessage 特性

声明：property MailMessage: TMailMessage;
调用了GetMailMessage函数后，这个特性返回检索到的邮件信息。TMailMessage是一个类，其中：

- Attachments　这是个TStringList对象，包含接收到的附件名称。
- Body　这是个TStringList对象，包含接收到的邮件正文。
- From　这是发件人的E-Mail地址。
- Head　这是个TStringList对象，包含接收到的邮件头标。
- MessageId　这是邮件的识别号如3587A72E.8928F0D5@public.east.cn.net。
- Subject　这是邮件的主题。

程序示例如下：

```
void _fastcall TForm1::Button3Click(TObject *Sender)
{
NMPOP31->GetMailMessage(StrToInt(Edit5->Text));
Edit6->Text = NMPOP31->MailMessage->From;
Edit7->Text = NMPOP31->MailMessage->Subject;
Edit9->Text = NMPOP31->MailMessage->MessageID;
Memo2->Lines->Assign(NMPOP31->MailMessage->Head);
Memo1->Lines->Assign(NMPOP31->MailMessage->Body);
if (NMPOP31->MailMessage->Attachments->Text != "")
    ShowMessage("Attachments\n"+NMPOP31->MailMessage->Attachments->Text);
}
```

Password 特性

声明： property Password: WideString;

这个特性用于设置登录到POP3服务器时的口令。

Port特性

声明：property Port: Longint;
这个特性用于指定要连接的POP3服务器的端口号，默认是110。

RemoteIP特性

声明：property RemoteIP: string;
这个特性返回POP3服务器的IP地址(点分法)。程序示例如下：

```
    void _fastcall TForm1::Button9Click(TObject *Sender)
{
ShowMessage("Currently connected to: "+NMPOP31->RemoteIP);
}
```

ReportLevel特性

声明：property ReportLevel: integer;
这个特性用于指定状态信息的详细程度，可以设为以下值：Status_None、Status_Informational、Status_Basic、Status_Routines、Status_Debug、Status_Trace。

Status特性

声明：property Status: String;

这个特性返回当前的状态信息。

Summary 特性

声明： property Summary: TSummary;
调用了GetSummary函数后，这个特性包含了邮件的统计信息。TSummary是一个类，其中：

- Bytes　　这是邮件的字节数。
- From　　这是发件人的E-Mail地址。
- MessageId　　这是邮件的编号如3587A72E.8928F0D5@public.east.cn.net。
- Subject　　这是邮件的主题。

程序示例如下：

```
void _fastcall TForm1::Button6Click(TObject *Sender)
{
NMPOP31->GetSummary(StrToInt(Edit5->Text));
Edit6->Text = NMPOP31->Summary->From;
Edit7->Text = NMPOP31->Summary->Subject;
Edit8->Text = IntToStr(NMPOP31->Summary->Bytes);
Edit9->Text = NMPOP31->Summary->MessageID;
}
```

Timeout 特性

声明： property TimeOut: Integer;

这个特性用于指定一个毫秒数，如果超过规定的时间Socket没有响应就触发异常。

TransactionReply 特性

声明： property TransactionReply: String;
这个特性返回上一次命令的执行结果，程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
NMPOP31->TimeOut = 20000;
NMPOP31->Connect( );
ShowMessage(NMPOP31->TransactionReply);
}
```

UserId 特性

声明： property UserId: WideString;
这个特性用于设置用户名，与Password特性一起用于身份验证。

13.2 POP 控件的方法

Abort 函数

声明： procedure Abort;

这个函数用于中止此次发送，并且断开与POP3服务器的连接。

Connect函数

声明：procedure Connect;
这个函数很重要，用于登录到POP3服务器，Host、Port、UserID和Password特性必须事先已设置。程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
NMPOP31->AttachFilePath = ".";
NMPOP31->DeleteOnRead = false;
NMPOP31->TimeOut = 20000;
NMPOP31->Host = "mail.myserver.com";
NMPOP31->Port = 110;
NMPOP31->UserID = "Ed";
NMPOP31->Password = "Smith";
NMPOP31->Connect( );
Label10->Caption = "# of Messages: "+IntToStr(NMPOP31->MailCount);
}
```

DeleteMailMessage函数

声明：procedure DeleteMailMessage(MailNumber: integer);
这个函数从POP3服务器中删除一个邮件，MailNumber参数指定要删除的邮

件的编号。注意：调用DeleteMailMessage函数实际上并没有真正删除邮件，除非断开了连接。在断开连接之前，您可以调用Reset函数恢复被删除的邮件。

Disconnect 函数

声明：procedure Disconnect;
这个函数断开与POP3服务器的连接。

GetMailMessage 函数

声明：procedure GetMailMessage(MailNumber: integer);
这个函数用于从POP3服务器下载一个邮件，MailNumber参数指定邮件的编号。

GetSummary 函数

声明：procedure GetSummary(MailNumber: integer);
这个函数将返回一个邮件的统计信息，MailNumber参数指定邮件的编号。

List 函数

声明：procedure List;
这个函数从POP3服务器检索一个列表，该列表中包含每一个邮件的编号和字节数。列表中的每一个项将触发一次OnList事件。

Reset 函数

声明：procedure Reset;
这个函数将恢复服务器中所有标记为“已删除”的邮件。

13.3 POP 控件的事件

POP控件的事件往往是与调用某个方法相关联的，如果事件被触发，通常表示调用成功，您可以在处理事件的句柄中得到POP服务器的返回信息。

OnAuthenticationFailed 事件

声明：property OnAuthenticationFailed: THandlerEvent;
其中，THandlerEvent是这样声明的：

```
THandlerEvent = Procedure(var handled: boolean) of Object;
```

当登录到POP3服务器时，如果没有事先给出用户名和口令，或者用户名和口令是非法的，将会触发这个事件。更正了错误后，如果把Handled参数设为True，将再登录一遍。

OnAuthenticationNeeded 事件

声明：property OnAuthenticationNeeded: THandlerEvent;
如果某个操作需要验证用户的身份就会触发这个事件。

OnConnect 事件

声明：property OnConnect: TNotifyEvent;
当客户已经成功地连接到POP3服务器时将触发这个事件。

OnConnectionFailed 事件

声明：property OnConnectionFailed: TNotifyEvent;
当客户试图登录到POP3服务器失败时将触发这个事件。

OnConnectionRequired 事件

声明：property OnConnectionRequired: THandlerEvent;
POP控件的大部分函数只能在已连接到POP3服务器上时才是有意义的，如果调用函数时没有连接到POP3服务器，将触发这个事件。在处理这个事件的句柄中，您应当调用Connect函数连接POP3服务器，然后把Handled参数设为True。程序示例如下：

```
void _fastcall TForm1::NMPOP31ConnectionRequired(bool &handled)
{
AnsiString BoxCaption = "Connection Required";
AnsiString BoxMsg = "Connection Required. Connect?";
if (MessageBox(0, &BoxMsg[1], &BoxCaption[1], MB_YESNO +
MB_ICONEXCLAMATION) = = IDYES)
{
    handled = true;
```

```
}
}
```

OnDisconnect 事件

声明：property OnDisconnect: TNotifyEvent;
当客户断开与POP3服务器的连接时将触发这个事件。注意：在处理这个事件的句柄中，如果需要访问VCL元件的话，最好先判断VCL元件是否存在。

OnFailure 事件

声明：property OnFailure: TNotifyEvent;
如果删除邮件时出错，将触发这个事件。

OnInvalidHost 事件

声明：property OnInvalidHost: THandlerEvent;
当Host特性所指定的主机名是非法的，就触发这个事件。

OnList 事件

声明：property OnList: TListEvent;
其中，TListEvent是这样声明的：

```
TListEvent = procedure(Msg, Size: integer) of object;
```

当程序调用List函数时，每检索到一个项就会触发一次OnList事件。Msg参数是邮件的编号，Size参数是邮件的字节数。程序示例如下：

```
void _fastcall TForm1->NMPOP31List(int Msg, Size)
{
Memo3->Lines->Add(IntToStr(Msg)+" //"+IntToStr(Size));
}
```

OnPacketRecvd 事件

声明：property OnPacketRecvd: TNotifyEvent;
每收到一个数据块就会触发这个事件，在处理这个事件的句柄中，您可以用BytesTotal特性和BytesRecvd特性显示接收的进度。

OnReset 事件

声明：property OnReset: TNotifyEvent;
当程序调用Reset函数恢复了标记为“已删除”的邮件后将触发这个事件。

OnRetriveStart 事件

声明：property OnRetriveStart: TNotifyEvent;
当程序调用GetMailMessage函数或GetSummary函数检索数据时将触发这个事件。程序示例如下：

```
void _fastcall TForm1::NMPOP31RetriveStart(TObject *Sender)
{
Form1->Cursor = crHourGlass;
StatusBar1->SimpleText = "Retrieval start";
```

```
}
```

OnRetriveEnd 事件

声明： property OnRetriveEnd: TNotifyEvent;
当程序调用GetMailMessage函数或GetSummary函数已成功地检索到数据就触发这个事件。程序示例如下：

```
void _fastcall TForm1::NMPOP31RetriveEnd(TObject *Sender)
{
Form1->Cursor = crDefault;
StatusBar1->SimpleText = "Retrieval end";
}
```

第十四章　使用NNTP控件

NNTP是Networking News Transfer Protocol的缩写，意为网络新闻传输协议。C++ Builder 3从NetManage公司引进的NNTP控件可以方便地登录到NNTP新闻服务器，下载可用的新闻组列表，选择新闻组，读取新闻组中的文章或者张贴自己的文章。
因为Internet上的新闻组很多，每个新闻组的文章也很多，NNTP控件具有很强的检索功能，可以让用户有选择地下载所需要的文章或者最新的文章，以提高工作效率。

14.1　NNTP控件的特性

AttachFilePath特性

声明：property AttachFilePath: string;
这个特性指定存放附件的路径。如果这个特性设为空，表示是当前目录。

Attachments特性

声明：property Attachments: TStringList;

调用了GetArticle函数后，这个特性返回一个列表，列表中包含附件的名称。
程序示例如下：

```
    void _fastcall TForm1::NMNNTP1Article(TObject *Sender)
{
Memo1->Text = NMNNTP1->Attachments->Text;
Memo2->Text = NMNNTP1->Body->Text;
}
```

Body 特性

声明： property Body: TExStringList;
调用了GetArticle函数或GetArticleBody函数后，这个特性返回文章的正文。

BytesRecvd 特性

声明： property BytesRecvd: longint;
这个特性返回已经接收到的数据长度，与BytesTotal特性配合使用可以显示传输进度。

BytesSent 特性

声明： property BytesSent: longint;
这个特性返回已经向新闻服务器发送的数据长度。

BytesTotal 特性

声明：property BytesTotal: longint;
这个特性返回总共要传输多少个字节。程序示例如下：

```
void _fastcall TForm1::NMNNTP1PacketRecvd(TObject*Sender)
{
StatusBar1->SimpleText = IntToStr(NMNNTP1->BytesRecvd)+
" bytes of "+IntToStr(NMNNTP1->BytesTotal)+" received";
}
```

CacheMode 特性

声明：property CacheMode: TCacheMode;
如果这个特性设为cmRemote，表示只从新闻服务器中检索文章。如果这个特性设为cmLocal，表示只从本地缓存中检索文章。如果这个特性设为cmMixed，表示既可以从新闻服务器中检索文章也可以从本地缓存中检索文章。程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
NMNNTP1->TimeOut = 20000;
NMNNTP1->NewsDir = "E:\\News\\";
NMNNTP1->AttachFilePath = "E:\\News\\";
NMNNTP1->Host = "forums.borland.com";
NMNNTP1->Port = 119;
```

```
if (Edit3->Text != "")
{
     NMNNTP1->UserId = Edit3->Text;
     NMNNTP1->Password = Edit4->Text;
}
if (RadioButton1->Checked)
     NMNNTP1->CacheMode = cmLocal;
if (RadioButton2->Checked)
     NMNNTP1->CacheMode = cmMixed;
if (RadioButton3->Checked)
     NMNNTP1->CacheMode = cmRemote;
if (CheckBox1->Checked)
     NMNNTP1->ParseAttachments = true;
else
     NMNNTP1->ParseAttachments = false;
NMNNTP1->Connect( );
}
```

Connected 特性

声明： property Connected: boolean;
如果当前与新闻服务器处于连接状态，这个特性就返回True。程序示例如下：

```
    void _fastcall TForm1::Button1Click(TObject *Sender)
{
if (NMNNTP1->Connected != true)
    NMNNTP1->Connect( );
}
```

CurrentArticle 特性

声明： property CurrentArticle: integer;
这个特性返回当前文章的编号。程序示例如下：

```
void _fastcall TForm1::BitBtn2Click(TObject *Sender)
{
Screen->Cursor = crHourglass;
NMNNTP1->GetArticle(NMNNTP1->CurrentArticle+1);
StringGrid1->Row = StringGrid1->Row + 1
Screen->Cursor = crDefault;
}
```

GroupList 特性

声明： property GroupList: TStringList;
调用了GetGroupList函数后，这个特性返回一个列表，列表中包含了所有的新闻组名称。程序示例如下：

```
    void _fastcall TForm1::BitBtn6Click(TObject *Sender)
```

```
{
NMNNTP1->GetGroupList( );
Memo3->Lines->Assign(NMNNTP1->GroupList);
ShowMessage("News group update complete");
}
```

Header 特性

声明：property Header: TExStringList;
调用了GetArticle函数或GetArticleHeader函数后，这个特性返回当前文章的标题。

HiMessage 特性

声明：property HiMessage: integer;
调用了SetGroup函数后，这个特性返回当前选择的新闻组中最大的文章编号。

Host 特性

声明：property Host: String;
这个特性用于指定要连接的NNTP服务器的主机名或IP地址，默认值是“news”。

LocalIP 特性

声明：property LocalIP: string;
这个特性返回客户计算机的IP地址。如果计算机有多个IP地址，这个特性返回其中第一个。程序示例如下：

```
    void _fastcall TForm1::Button9Click(TObject*Sender)
{
ShowMessage("Your local IP Address is "+NMNNTP1->LocalIP);
}
```

LoMessage 特性

声明：property LoMessage: integer;
调用了SetGroup函数后，这个特性返回当前选择的新闻组中最小的文章编号，程序示例如下：

```
    void _fastcall TForm1::NMNNTP1GroupSelect(TObject *Sender)
{
Form1->Caption = "NNTP Demo - ["+NMNNTP1->SelectedGroup+"Lo:"+
IntToStr(NMNNTP1->LoMessage)+" Hi:"+IntToStr(NMNNTP1->HiMessage)+"]";
if (NMNNTP1->Posting)
    Form1->Caption = Form1->Caption + " Posting Allowed]"
else
    Form1->Caption = Form1->Caption + " Posting Prohibited]";
}
```

NewsDir 特性

声明：property NewsDir: string;
这个特性用于指定一个目录临时存放文章。如果这个特性设为空，就是当前目录。

ParseAttachments 特性

声明：property ParseAttachments: boolean;
如果这个特性设为True，将自动从文章中把附件解码并分离出来。

Password 特性

声明：property Password: string;
登录到有的新闻服务器时可能需要给出口令。

Port 特性

声明：property Port: Longint;
这个特性用于指定要连接的NNTP服务器的端口号，默认是119。

PostBody 特性

声明：property PostBody: TExStringList;
这个特性用于指定要张贴到新闻组的文章的正文。程序示例如下：

```
    void _fastcall TForm1::Button1Click(TObject *Sender)
{
```

```
NMNNTP1->PostBody->Assign(Memo1->Lines);
NMNNTP1->PostRecord->PrFromAddress = "yogi@booboo.com";
NMNNTP1->PostRecord->PrReplyTo = "yogi@booboo.com";
NMNNTP1->PostRecord->PrSubject = "Test Posting";
NMNNTP1->PostRecord->PrNewsGroups = NMNNTP1->SelectedGroup;
NMNNTP1->Attachments->Assign(ListBox1->Items);
NMNNTP1->PostHeader->Values["Referer"]="www.netmastersllc.com";
NMNNTP1->PostArticle( );
}
```

PostHeader 特性

声明：property PostHeader: TExStringList;
这个特性用于指定要张贴到新闻组的文章的头标。程序示例请参见PostBody特性。

Posting 特性

声明：property Posting: boolean;
如果当前选择的新闻组允许张贴文章，这个特性就返回True。

RemoteIP 特性

声明：property RemoteIP: string;
这个特性返回新闻服务器的IP地址，程序示例如下：

```
    void _fastcall TForm1::Button9Click(TObject *Sender)
{
ShowMessage("Currently connected to: "+NMNNTP1->RemoteIP);
}
```

ReplyNumber 特性

声明：property ReplyNumber : Longint;
这个只读的特性返回NNTP服务器响应客户请求的应答代码。

ReportLevel 特性

声明：property ReportLevel: integer;
这个特性用于指定状态信息的详细程度，可以设为以下值：Status_None、Status_Inf-ormational、Status_Basic、Status_Routines、Status_Debug、Status_Trace。

SelectedGroup 特性

声明：property SelectedGroup: string;
调用了SetGroup 函数后，这个特性返回当前选择的新闻组名。

Status 特性

声明：property Status: String;
这个特性返回当前的状态信息。

Timeout 特性

声明：property TimeOut: Integer;
这个特性用于指定一个毫秒数，如果超过规定的时间Socket没有响应就触发异常。

TransactionReply 特性

声明：property TransactionReply: String;
这个特性返回上一次命令的执行结果，程序示例如下：

```
    void _fastcall TForm1::Button1Click(TObject*Sender)
{
NMNNTP1->Connect( );
ShowMessage(NMNNTP1->TransactionReply);
}
```

UserId 特性

声明：property UserId: WideString;
这个特性用于设置用户名，与Password特性一起用于身份验证。

14.2 NNTP控件的方法

Connect函数

声明：procedure Connect;
这个函数用于登录到NNTP服务器。Host、Port、UserId和Password等特性必须事先设置好。

Disconnect函数

声明：procedure Disconnect;
这个函数断开与新闻服务器的连接。

GetArticle函数

声明：procedure GetArticle(Ref: integer);
这个函数从新闻服务器的当前新闻组中检索一篇文章，Ref参数指定该文章的编号。程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
NMNNTP1->GetArticle(156);
}
```

GetArticleBody 函数

声明：procedure GetArticleBody(Ref: integer);
这个函数用于从新闻服务器的当前新闻组中检索一篇文章的正文，Ref参数指定该文章的编号。

GetArticleHeader 函数

声明：procedure GetArticleHeader(Ref: integer);
这个函数用于从当前新闻组中检索一篇文章的标题，Ref参数指定该文章的编号。

GetArticleList 函数

声明：procedure GetArticleList(All: boolean; ArticleNumber: integer);
这个函数用于从当前选择的新闻组中检索文章的列表，如果All参数设为True，列表中包含新闻组中的所有文章，如果All参数设为False，列表中只包含编号大于ArticleNumber参数的文章。每检索到一个文章，就会触发一次OnHeaderList事件。

GetGroupList 函数

声明：procedure GetGroupList;
这个函数很重要，用于获得新闻服务器中所有新闻组的列表。

PostArticle 函数

声明：procedure PostArticle;
这个函数用于张贴文章，PostBody特性和PostHeader特性必须事先设置好。

SetGroup 函数

声明：procedure SetGroup(Group: String);
这个函数把Group参数指定的新闻组设为当前新闻组。程序示例如下：

```
void _fastcall TForm1::ListBox1DblClick(TObject *Sender);
{
NMNNTP1->SetGroup("alt.pascal.delphi");
}
```

14.3 NNTP 控件的事件

NNTP控件的事件往往是与调用某个方法相关联的，如果事件被触发，通常表示调用成功，您可以在处理事件的句柄中得到NNTP服务器的返回信息。

OnArticle 事件

声明：property OnArticle: TNotifyEvent;
每下载一篇文章就会触发一次这个事件。程序示例如下：

```
void _fastcall TForm1::NMNNTP1Article(TObject *Sender)
```

```
{
Memo1->Text = NMNNTP1->Header->Text;
Memo2->Text = NMNNTP1->Body->Text;
}
```

OnArticleCacheUpdate 事件

声明：property OnArticleCacheUpdate: THandlerEvent;
其中，THandlerEvent是这样声明的：

```
THandlerEvent = Procedure(var handled: boolean) of Object;
```

每下载一篇文章并且该文章需要保存到缓存中时就会触发这个事件。

OnAuthenticationFailed 事件

声明：property OnAuthenticationFailed: THandlerEvent;
其中，THandlerEvent是这样声明的：

```
THandlerEvent = Procedure(var handled: boolean) of Object;
```

当登录到新闻服务器时，如果没有事先给出用户名和口令，或者用户名和口令是非法的，将会触发这个事件。更正了错误后，如果把Handled参数设为True，将再登录一遍。

OnAuthenticationNeeded 事件

声明：property OnAuthenticationNeeded: THandlerEvent;
如果某个操作需要验证用户的身份就会触发这个事件。

OnBody 事件

声明：property OnBody: TNotifyEvent;
每下载到一篇文章的正文就会触发这个事件。程序示例如下：

```
    void _fastcall TForm1::NMNNTP1Body(TObject*Sender)
{
Memo1->Lines->Assign(NMNNTP1->Body);
}
```

OnBodyCacheUpdate 事件

声明：property OnBodyCacheUpdate: THandlerEvent;
每下载一篇文章的正文并且该正文需要保存到缓存中时就会触发这个事件。

OnConnect 事件

声明：property OnConnect: TNotifyEvent;
当客户已经成功地连接到新闻服务器时将触发这个事件。

OnConnectionFailed 事件

声明：property OnConnectionFailed: TNotifyEvent;
当客户试图登录到新闻服务器失败时将触发这个事件。

OnDisconnect 事件

声明：property OnDisconnect: TNotifyEvent;
当客户断开与新闻服务器的连接时将触发这个事件。注意：在处理这个事件的句柄中，如果需要访问VCL元件的话，最好先判断VCL元件是否存在。

OnGroupListCacheUpdate 事件

声明：property OnGroupListCacheUpdate: TGroupRetrievedCacheEvent;
其中，TGroupRetrievedCacheEvent是这样声明的：

```
TGroupRetrievedCacheEvent = Procedure(var Handled: boolean; Name: string; FirstArticle, LastArticle: integer; Posting: boolean) of object;
```

每当检索到一个新闻组时就会触发这个事件。Name参数是新闻组名称，FirstArticle参数是新闻组中第一篇文章的编号，LastArticle参数是新闻组中最后一篇文章的编号，Posting参数为True表示允许张贴文章。程序示例如下：

```
void _fastcall TForm1::NMNNTP1GroupListCacheUpdate(bool &Handled, AnsiString Name, int FirstAr- ticle, LastArticle , bool Posting)
{
ListBox1->Items->Add(Name);
}
```

OnGroupSelect 事件

声明：property OnGroupSelect: TNotifyEvent;

当程序调用SetGroup函数选择了当前新闻组后，就会触发这个事件。

OnGroupSelectRequired 事件

声明：property OnGroupSelectRequired: THandlerEvent;
NNTP控件的大部分操作是针对当前新闻组进行的，如果当前没有指定新闻组，就会触发这个事件。程序示例如下：

```
    void _fastcall TForm1::NMNNTP1GroupSelectRequired(bool &Handled)
{
AnsiString TmpStr;
if (InputQuery("Newsgroup selectoion Required!","Enter NewsGroup Name:",TmpStr))
{
    NMNNTP1->SetGroup(TmpStr);
    handled = true;
}
}
```

OnHeader 事件

声明：property OnHeader: TNotifyEvent;
每检索到一篇文章的头标就会触发这个事件。程序示例如下：

```
void _fastcall TForm1::NMNNTP1Header(TObject *Sender)
{
Memo1->Lines->Assign(NMNNTP1->Header);
```

}

OnHeaderCacheUpdate 事件

声明：property OnHeaderCacheUpdate: THeaderCacheEvent;
每检索到一篇文章的头标并且该头标需要保存到缓存中时就会触发这个事件。

OnInvalidArticle 事件

声明：property OnInvalidArticle: TNotifyEvent;
如果要下载的文章不存在，就会触发这个事件。

OnInvalidHost 事件

声明：property OnInvalidHost: THandlerEvent;
当Host特性所指定的主机名是非法的，就触发这个事件。

OnPosted 事件

声明：property OnPosted: TNotifyEvent;
当一篇文章已经成功地张贴到新闻组中，就会触发这个事件。

第十五章　创建Web服务器应用程序

要创建Web服务器应用程序，必须要有一个Web服务器环境，目前比较流行的Web服务器有基于Windows NT 4.0的IIS(Internet Information Server的缩写)、基于Windows 95的PWS(Persional Web Server的缩写)、Netscape Enterprise Server以及WebSite Server等，我们推荐采用Windows NT 4.0的IIS。下面我们先从静态的HTML页面开始，循序渐进地介绍怎样用C++ Builder 3创建Web服务器应用程序。

15.1　WWW是怎样工作的

WWW的基本结构仍然是客户机/服务器模式，客户端就是Web浏览器，如Internet Explorer，服务器端就是Web服务器如IIS，Web浏览器和Web服务器之间通过HTTP进行通信。所谓HTTP，是Hypertext Transfer Protocol的缩写，意为超文本传输协议，HTTP又是建立在TCP/IP协议(Transmission Control Protocol/Internet Protocol)的基础上。

当客户需要得到服务器的服务时，换句话说就是需要得到一个HTML页面时，就通过Web浏览器向特定的Web服务器发出HTTP请求，对于客户来说，发出HTTP请求也许是非常简单的，他只要用鼠标单击一个超级链接。HTTP

请求分为两种类型，一是GET请求，还有一个是POST请求。GET请求用于检索静态的HTML页面、图形或声音、视频剪辑等，而POST请求用于与Web服务器交互以得到动态的HTML页面，当然，Web服务器端需要运行一个程序(这就是所谓的Web服务器应用程序)，由此程序生成动态的HTML页面。
客户发出的HTTP请求通过Internet传送到Web服务器后，Web服务器就分析其中的URL(统一资源定位串)，然后在服务器的磁盘中查找符合要求的文档，如果找到，就把文档回传给Web浏览器。假设URL是http://www.borland.com:80/custom/index.html，表示协议是HTTP，服务器的主机名是www.borland.com，端口号是80，客户要求检索的是位于custom目录中的index.html文件。关于URL后面将详细介绍。
当Web浏览器收到Web服务器传过来的HTML文档时，就对HTML文档的代码进行语法分析，然后在屏幕上正确地显示出页面。

15.2 静态的HTML页面

所谓静态的HTML页面，是指客户在访问Web服务器时所得到的HTML页面每次都是不变化的。创建这种Web服务器应用程序的一般步骤是：
使用“File”菜单上的“New”命令，翻到“New”页，如图15.1所示。

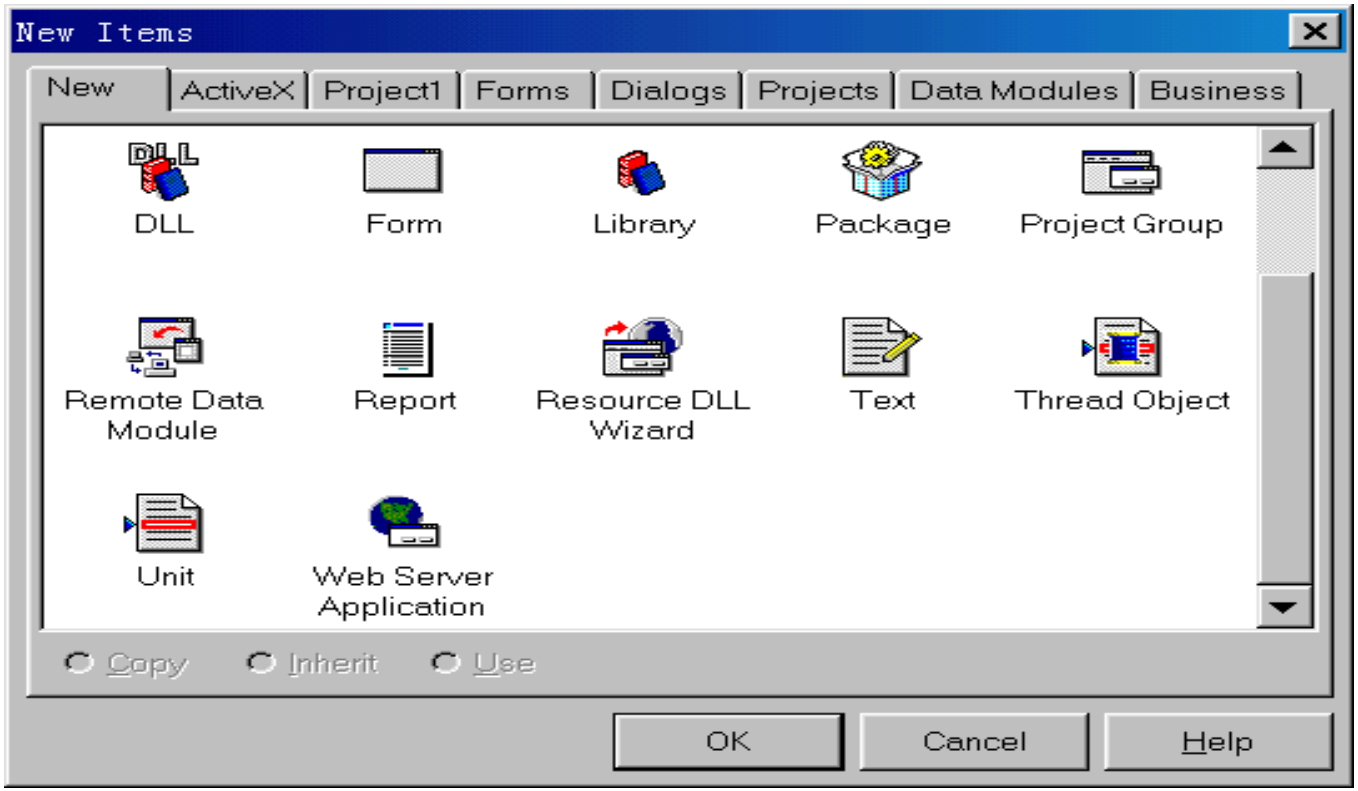


图 15.1　开始创建 Web 服务器应用程序

双击“Web Server Application”图标，C++ Builder 3将打开“New Web Server Application”对话框，让您选择Web服务器应用程序的类型，如图15.2所示。

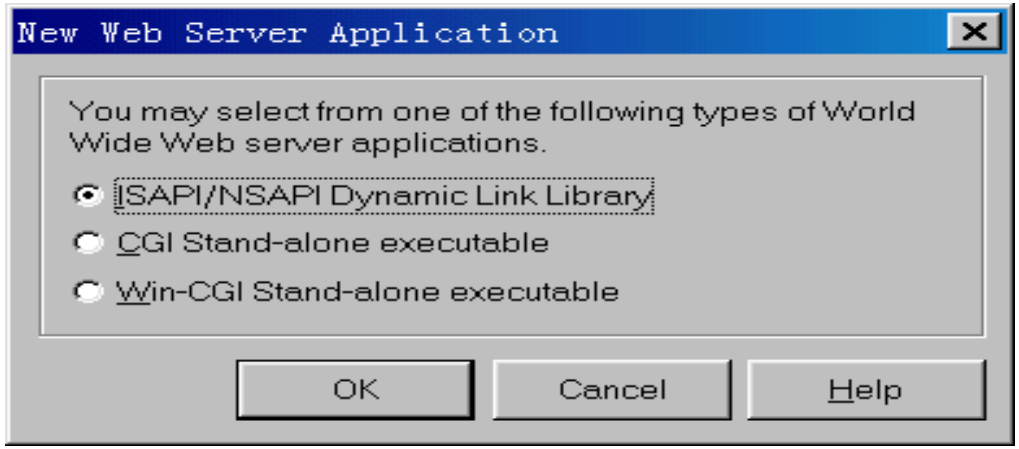


图 15.2　选择 Web 服务器应用程序的类型

可以看出，Web服务器应用程序有三种类型：

一是“ISAPI/NSAPI Dynamic Link Library”(推荐)，表示Web服务器应用程序是一个动态链接库。所谓ISAPI，是Internet Server API的缩写，是由Microsoft定义的编程接口。Netscape也定义了一套编程接口，称为NSAPI，并且宣称支持ISAPI标准。不过，作为一般的程序员来说，没有必要掌握两套编程接口，本书以ISAPI为例。

二是“CGI Stand-alone executable”，表示Web服务器应用程序是一个可单独执行的控件台程序或者说字符界面程序，它与Web服务器之间通过标准的输入输出设备交换数据。所谓CGI，是Common Gateway Interface的缩写，后面将详细介绍怎样写CGI程序。

三是“Win-CGI Stand-alone executable”，表示Web服务器应用程序是一个可单独执行的Windows应用程序。Web服务器通过配置文件(.INI)传递客户

的请求消息，Web服务器应用程序通过文件传递响应结果。

从编程的角度看，Web服务器应用程序由TWebApplication对象来描述，客户的HTTP请求消息由TWebRequest对象来描述，Web服务器应用程序作出的反应由TWebResponse对象来描述。不过，TWebApplication、TWebRequest、TWebResponse实际上还只是祖先类，不同类型的Web服务器应用程序分别对应着不同的派生对象，列表如下：

Web服务器应用程序类型	应用程序对象	客户请求消息	服务器的响应
Microsoft Server DLL (ISAPI)	TISAPIApplication	TISAPIRequest	TISAPIResponse
Netscape Server DLL (NSAPI)	TISAPIApplication	TISAPIRequest	TISAPIResponse
Console CGI application	TCGIApplication	TCGIRequest	TCGIResponse
Windows CGI application	TCGIApplication	TWinCGIRequest	TWinCGIResponse

选择了要创建的服务器应用程序类型后，单击“OK”按钮，C++ Builder 3就建立一个新的项目，项目中目前只有一个空白的Web模块，如图15.3所示。

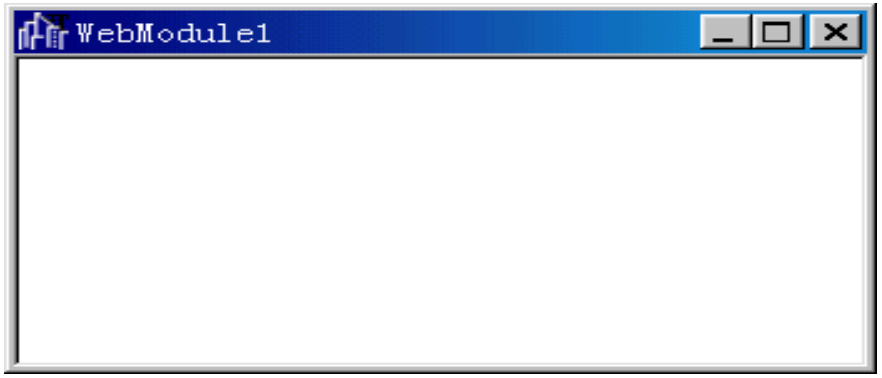


图 15.3　空白的 Web 模块

Web模块(TWebModule对象)与C++ Builder 3中的数据模块(TDataModule对象)非常相似，事实上，TWebModule对象就是从TDataModule对象继承下来的。与数据模块一样，Web模块上通常放一些非可视的数据访问元件如TTable、TQuery、TDataSource等。当然，如果Web模块仅仅作为一个容器，它也没有存在的必要了，因为数据模块完全可以充当这样的角色。Web模块不仅仅是个容器，它还是Web服务器应用程序的调度中心。

每个Web模块上都有一个默认的Web调度器(TWebDispatcher对象)，当客户通过Web浏览器访问Web服务器时，由Web调度器决定指派哪个动作项(TWebActionItem对象)去响应客户的请求，ActionItem可以理解为Web服务器的一个入口。

由此可见，一个Web服务器应用程序应当创建若干个ActionItem以供Web调度器指派。C++ Builder 3用一个专门的动作编辑器来创建和管理ActionItem。要打开动作编辑器，有三种办法：一是直接双击Web模块，二是用鼠标右键单击Web模块，在弹出的菜单中选择“Action Editor”命令，三是在Object

Inspector中找到Actions特性，单击该特性边上的省略号按钮。打开的动作编辑器如图15.4所示。

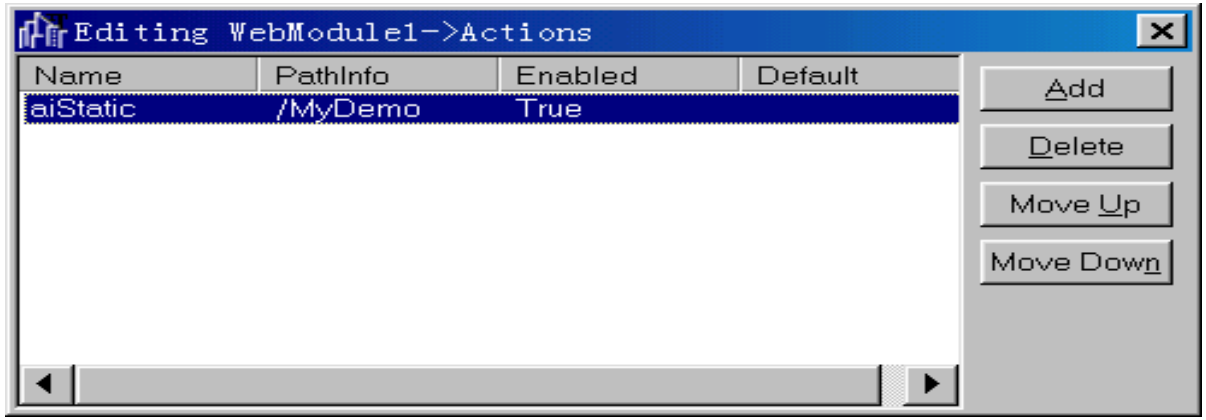


图 15.4　动作编辑器

要创建一个新的ActionItem，单击“Add”按钮，在动作编辑器中选择新创建的ActionItem，此时Object Inspector将显示此ActionItem的特性和事件，首先设置它的Name特性，给此ActionItem取个名字如StaticPage，然后设置它的PathInfo特性指定此ActionItem在Web服务器上的入口路径如“/MyDemo”。

如果要再创建一个ActionItem，重复上述步骤。要注意的是，每个ActionItem应具有其唯一的入口路径。

要删除一个ActionItem，单击“Delete”按钮。

要调整ActionItem的先后顺序，单击“Move Up”按钮或“Move Down”按钮。

前面讲过，指派哪个ActionItem去响应客户的请求是由Web调度器决定的，至于ActionItem怎么响应客户的请求则是在处理OnAction事件的句柄中进行的，您要做的事情就是在事件句柄中用HTML语言描述Web页面。程序示例如下：

```
void _fastcall TWebModule1::WebModule1aiStaticAction(TObject *Sender,
TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
AnsiString HTML = "<HTML>";
HTML = HTML + "<BODY>";
HTML = HTML + "<H1>Hello This Is My Demo</H1>";
HTML = HTML + "<P>My Name Is XXH</P>";
HTML = HTML + "</BODY>";
HTML = HTML + "</HTML>";
Response->Content = HTML;
};
```

可以看出，在处理动作项的OnAction事件的句柄中，可以通过Request参数访问到客户的请求消息，不过，这里只关心Response参数，要响应客户的请求，实际上就是把用HTML代码描述的页面赋值给Response参数(TWebResponse对象)的Content特性，Web调度器会自动把响应结果传递给Web服务器，再由Web服务器传递给客户。

至此，一个简单的Web服务器应用程序创建完毕，您可以通过Web浏览器来测试它。现在的问题是，在Web浏览器的地址框内键入什么URL呢？这取决于两个方面的因素：

一是您怎样保存这个项目，假设您使用“File”菜单上的“Save Project As”命令指定项目名是XXH.BPR，那么编译此项目后所得到的动态链接库就是XXH.DLL(假设Web服务器应用程序的类型是“ISAPI/NSAPI Dynamic Link Library”)。

另一个因素是当前所处的Web服务器环境，如果操作系统是Windows 95，并且用Microsoft的Personal Web Server作为Web服务器，服务器的主机名是www.server.com，所有服务入口都建在scripts目录下，一个完整的URL是：

http://www.server.com/scripts/xxh.dll/MyDemo

如果要在本地测试Web服务器应用程序，您应当先建立个人Web服务器。笔者所用的是Windows 98 Beta2中文版，它内置了Personal Web Server，Web服务器默认的IP地址是127.0.0.1，所有的服务入口都建在scripts目录下，一个完整的URL是：

http://127.0.0.1/scripts/xxh.dll/MyDemo 或

http://localhost/scripts/xxh.dll/MyDemo

您只要键入上述URL，就会看到一个静态的HTML页面，如图15.5所示。

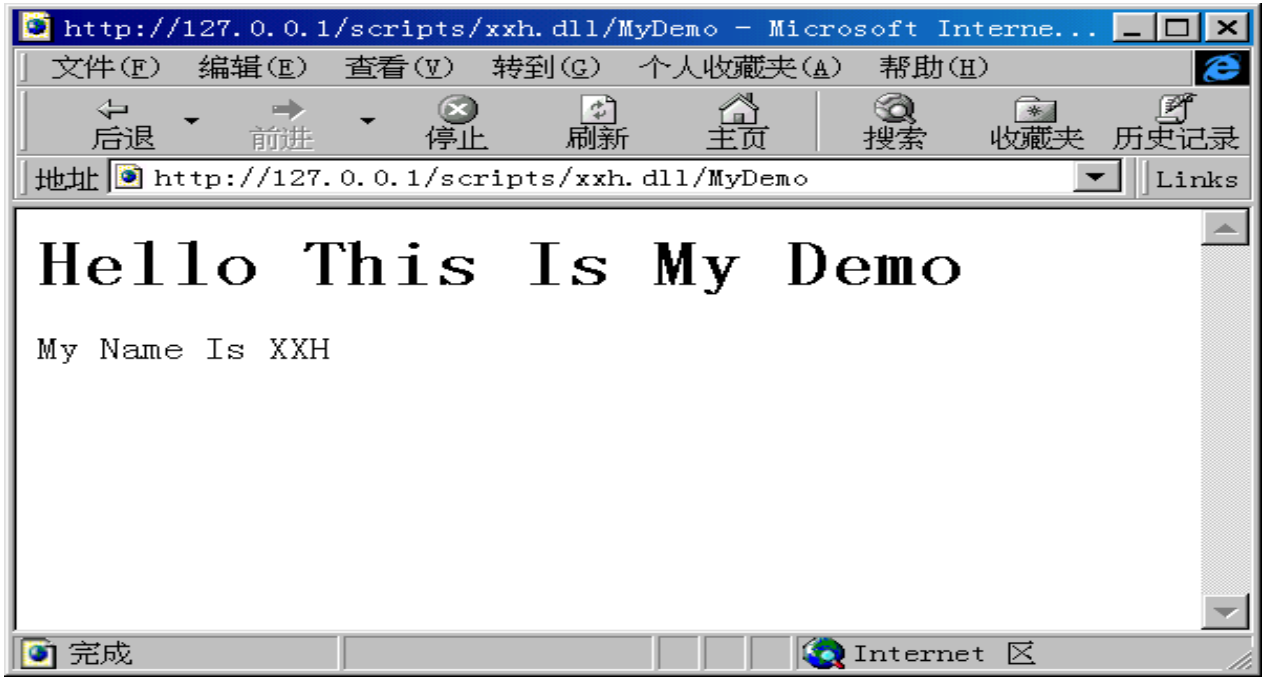


图 15.5　静态的 HTML 页面

15.3　动态的 HTML 页面

所谓动态的HTML页面，是指客户访问Web服务器时所得到的HTML页面每次是不完全相同的，例如，您可以在页面上显示当前的日期和时间。

要创建动态的HTML页面，基本步骤与创建静态的HTML页类似，您可以在上述创建静态HTML页面的基础上，再增加一个ActionItem，然后把它的Name特性设为DynamicPage，把PathInfo特性设为/DateTime，然后响应OnAction

事件，程序示例如下：

```
void _fastcall TWebModule1::WebModule1DynamicPageAction(TObject *Sender,
TWebRequest*Request, TWebResponse *Response, bool &Handled)
{
AnsiString HTML;
HTML = "<HTML>";
HTML = HTML + "<BODY>";
HTML = HTML + "<H1>当前的日期和时间是...</H1>";
HTML = HTML + DateTimeToStr(Now);
HTML = HTML + "</BODY>";
HTML = HTML + "</HTML>";
Response->Content = HTML;
};
```

DateTimeToStr是C++ Builder 3的运行期函数，用于取得当前的日期和时间。上述程序编译后，假设项目名仍然是XXH.DPR，完整的URL如下：

http://127.0.0.1/scripts/xxh.dll/DateTime 或 http://localhost/scripts/xxh.dll/DateTime

在Web浏览器的地址框内键入URL，您就会看到当前的日期和时间，如图15.6所示。

图 15.6　显示当前的日期和时间

15.4　怎样与客户交互

上面介绍的动态页面实际上还不是真正意义上的动态，因为日期和时间的变化并不依赖于用户的交互，一个好的Web服务器应用程序应当具有良好的交互能力。要实现客户与服务器的交互有两种方式：

第一种方式是在Web浏览器的地址框中键入URL时直接附带参数，但键入

的格式比较复杂，如果您使用过搜索引擎如Altavista、Yahoo等，您可能会比较容易理解。例如，假设我们使用Digital公司的Altavista搜索有关主题是Delphi的中文网站，您可以在搜索框内键入“Delphi and Chinese”，然后按Enter键或单击“Search”按钮，此时，Altavista就根据当前的搜索选项自动生成URL并开始搜索，如图15.7所示。

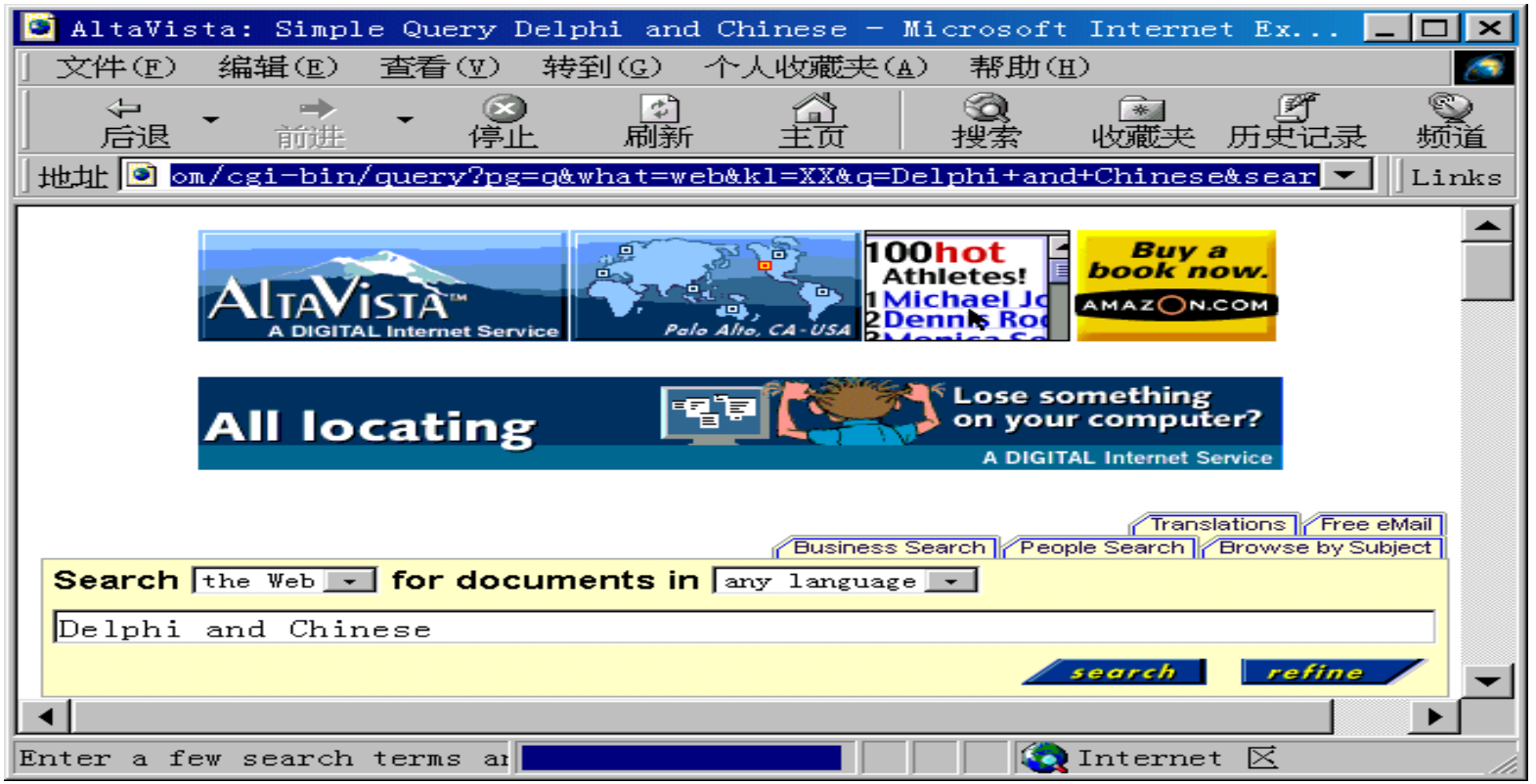


图 15.7　URL 中的参数

要搜索主题是Delphi的中文网站，Altavista自动生成的URL是：

http://altavista.digital.com/cgi-bin/query?pg=q&what=web&kl=XX&q=Delphi+and+Chinese&search.x=34&search.y=7

URL中附带参数的规则是：在参数名前加一个问号，参数名后用一个等号后跟参数的值，如果要附带多个参数，参数与参数之间用&符号隔开，如果参数的值包含空格，必须用加号连接起来。根据上述规则，上面这个URL中有6个参数，其中，q参数的值是Delphi+and+Chinese，其它参数是Altavista这个搜索引擎特定的参数，读者不必管它。

这种方式的缺陷是，如果参数比较多，客户键入URL时容易输错，除非您也象Altavista那样用图形界面让客户操作，然后自动生成URL。

第二种方式是在HTML页面中用窗体(Form)让客户填写有关信息，我们通过一个HTML程序示例来说明怎样用窗体获取客户的输入：

```
<HTML>

<HEAD>
<TITLE>WEB SERVER APPLICATION</TITLE>
</HEAD>

<BODY>

<H1 ALIGN="CENTER">WEB SERVER APPLICATION</H1>
<H2>请在下面两个窗体中填写</H2>
```

```
<P>这个窗体用GET请求</P>
<FORM ACTION= "../SCRIPTS/XXH.DLL/FormDemo" METHOD= "GET">
<P>NAME: <INPUT TYPE= "TEXT" SIZE= "20" NAME= "Name"></P>
<P>ADDRESS: <INPUT TYPE= "TEXT" SIZE= "20" NAME= "Address"></P>
<P>PHONE: <INPUT TYPE= "TEXT" SIZE= "20" NAME= "Phone"></P>
<P>E-MAIL: <INPUT TYPE= "TEXT" SIZE= "25" NAME= "email"></P>
<P><INPUT TYPE= "Submit"><INPUT TYPE= "Reset"></P>
</FORM>

<P>这个窗体用POST请求</P>
<FORM ACTION="../SCRIPTS/XXH.DLL/FormDemo" METHOD= "POST">
<P>NAME: <INPUT TYPE= "TEXT" SIZE= "20" NAME= "Name"></P>
<P>ADDRESS: <INPUT TYPE= "TEXT" SIZE= "20" NAME= "Address"></P>
<P>PHONE: <INPUT TYPE= "TEXT" SIZE= "20" NAME= "Phone"></P>
<P>E-MAIL: <INPUT TYPE= "TEXT" SIZE= "25" NAME= "email"></P>
<P><INPUT TYPE= "Submit"><INPUT TYPE= "Reset"></P>
</FORM>

</BODY>
</HTML>
```

可以看出，此HTML页面上有两个窗体，每个窗体上有四个编辑框用于接受客户的输入，有两个按钮，一个是“提交查询内容”按钮，另一个是“复

原”按钮，如图15.8所示。

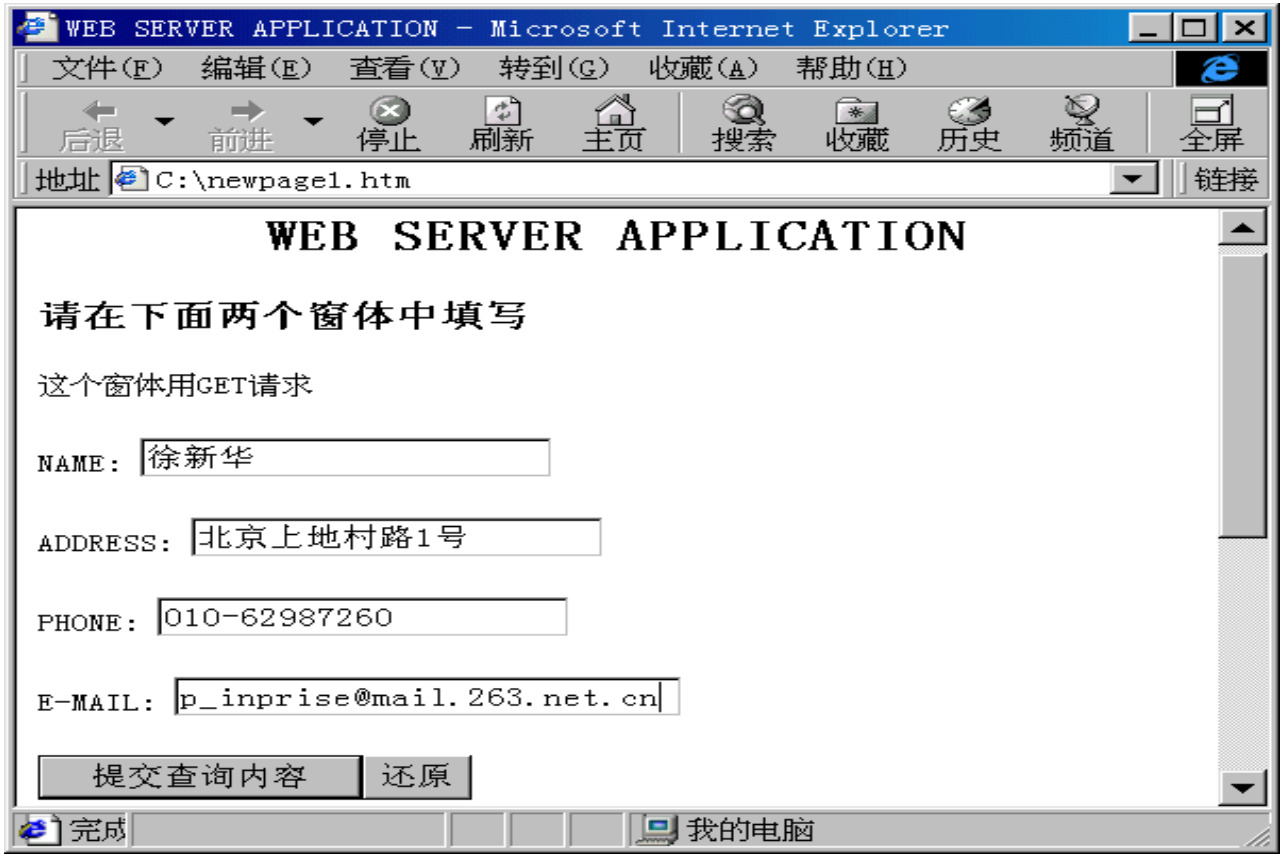


图 15.8　用窗体让用户输入

现在的问题是，Web服务器应用程序怎么获取用户的输入并对用户的输入作出反映？

要实现与客户的交互，首先要从客户的HTTP请求中检索出URL附带的参数，这就要用到动作项的OnAction事件中的Request参数。

Request参数是TWebRequest对象，用于描述客户的HTTP请求，其中，Query特性是GET请求所附带的参数组成的字符串，不过，直接用Query特性不太方便，您可以用QueryFields特性得到分解后的各个参数。同样，Content特性是POST请求所附带的参数组成的字符串，不过，直接用Content特性不太方便，您可以用ContentFields特性得到分解后的各个参数。程序示例如下：

```
void _fastcall TWebModule1::WebModule1DynamicPageAction(TObject *Sender,
TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
TStrings *Data;
Data = Request->QueryFields;
...
};
```

15.5 交互生成页面

您也许是某大公司的商务主管，要在Internet上开展一个调查以获取市场需求等信息，这就要借助于窗体(Form)让客户填写，当客户填写完毕并提交后，服务器一般要把窗体再传一遍，当然这时候窗体中已经有了客户填写的内容，此外，作为礼貌，您应当对客户接受调查表示感谢。现在的问题是，客户输入的信息在您设计Web服务器应用程序时是未知的，也就是说，您没法在程序中直接用HTML语言描述客户填写的内容，而需要在运行期动态地生成Web页面，这就要用到TPageProducer元件。TPageProducer元件是非

常有用的，它能够基于事先建立的HTML模板，分析客户的输入并自动生成动态的HTML页面，因此，这个元件又叫“页面生成器”。下面我们就介绍这个元件的一般用法：

首先要建立一个HTML模板(后面将详细介绍怎样建立HTML模板，怎样在HTML模板中设置标记)，让页面生成器知道应该怎样转换。

然后把一个TPageProducer元件放到Web模块上，您既可以通过HTMLDoc特性(TStrings对象)指定HTML模板，也可以通过HTMLFile特性指定HTML模板所在的文件(扩展名是.HTM或.HTML)，但要注意，您必须正确指定模板文件所在的路径，这里是指在服务器上的路径，而不是您开发这个程序时所用的计算机上的路径。

当TPageProducer元件遇到HTML模板中的标记时就会触发OnHTMLTag事件，这样您就有机会分析当前遇到的是什么标记，然后把客户输入的信息赋给ReplaceText参数，程序示例如下：

```
void _fastcall TWebModule1::PageProducer1HTMLTag(TObject *Sender, TTag Tag, const
AnsiString TagString, TStrings *TagParams, AnsiString &ReplaceText)
{
TStrings *Data = NULL;
switch(MethodType)
{
    case mtPost:
        Data = Request->ContentFields;
        break;
```

```
        case mtGet:
            Data = Request->QueryFields;
            break;
}
ReplaceText = Data->Value[TagString];
};
```

上述这段程序首先根据Request参数(TWebRequest对象)的MethodType特性判断客户的HTTP请求类型是GET还是POST，如果是GET，就通过QueryFields特性得到URL中附带的参数，如果是POST，就通过ContentFields特性得到URL中附带的参数。TagString参数就是当前遇到的标记名，由此得到客户输入的内容并把它赋给ReplaceText参数。

至于怎样把转换得到的Web页面回传给客户，很简单，您可以再加一个ActionItem，设置它的Name特性为aiParseFile，设置PathInfo特性为/ParseFile，然后再响应OnAction事件，程序示例如下：

```
void _fastcall TWebModule1::WebModule1aiParseFileAction(TObject*Sender,
TWebRequest*Request, TWebResponse*Response, bool&Handled)
{
PageProducer1->HTMLFile = "Greeting.html";
Response->Content = PageProducer1->Content( );
};
```

15.6 与数据库的连接

用C++ Builder 3创建的Web服务器应用程序可以方便地访问数据库，并自动管理与数据库的连接。要访问数据库，就要用到TDataSetTableProducer元件和TQueryTableProducer元件，这两个元件实际上起到桥梁的作用，用于从标准的数据集如TTable、TQuery以及TClientDataSet引入数据。TDataSetTableProducer元件和TQueryTableProducer元件在用法上很相似，主要的区别是后者是建立在查询的基础上。下面我们以TDataSetTableProducer元件为例介绍Web服务器应用程序怎样访问数据库。

首先把一个TSession元件加到Web模块上，把它的AutoSessionName特性设为True。为什么要用TSession元件呢？因为Web服务器可能同时连接着几个数据库，需要用TSession提供全局控制。即使一个Web服务器只连接了一个数据库，但由于Web服务器应用程序可能是动态链接库，有可能同时运行着多个实例，实际上相当于同时连接着几个数据库。

把一个TTable元件加到Web模块上，设置它的DatabaseName特性指定要访问的数据库例如DBDEMOS，设置它的TableName特性指定要访问的表例如Customer.DB。如果只对部分字段感兴趣，您可以用字段编辑器选择部分字段。

把一个TDataSetTableProducer元件加到Web模块上，设置它的DataSet特性指定TTable元件，然后再设置它的MaxRows特性指定最多行数例如200，如果数据库的记录很多的话，就只转换MaxRows特性指定的记录数。

与数据库连接只是TDataSetTableProducer元件的一部分功能，事实上，它还

是个页面生成器，它能自动生成一个HTML页面，以表格的形式显示数据库的记录。
当然，如果页面上光有数据显然是不够的，您至少要给表格加一个标题，这是通过设置TDataSetTableProducer元件的Caption特性实现的。如果要在整个表格包括标题的前面加上一些内容，您可以设置TDataSetTableProducer元件的Header特性。如果要在整个表格的末尾加上一些内容，您可以设置TDataSetTableProducer元件的Footer特性。Header特性和Footer特性都是TStrings对象，您可以分别填入一些HTML源代码，并且只能填入HTML代码，不过，编译器并不检查HTML代码的语法，这是由Web浏览器检查的。如果要对Header特性和Footer特性赋值的话，您必须遵守HTML语言的语法规则，例如，Header特性的第一行必须是<HTML>，Footer特性的最后一行必须是</HTML>。
例如，您可以这样给TDataSetTableProducer元件的Header特性赋值：

```
//---------------------------------------------------------
DataSetTableProducer1->Header->Add("<HTML>");
DataSetTableProducer1->Header->Add("<HEAD>");
DataSetTableProducer1->Header->Add("<TITLE>
下面是Borland公司97年的销售统计</TITLE>");
DataSetTableProducer1->Header->Add("</HEAD>");
DataSetTableProducer1->Header->Add("<BODY>");
//---------------------------------------------------------
```

您可以这样给TDataSetTableProducer元件的Footer 特性赋值：

```
//-------------------------------------------------------------
DataSetTableProducer1->Footer->Add("<HR>");
DataSetTableProducer1->Footer->Add("<H1>
注：更详细的情况请访问www.borland.com</H1>");
DataSetTableProducer1->Footer->Add("</BODY>");
DataSetTableProducer1->Footer->Add("</HTML>");
//-------------------------------------------------------------
```

至于怎样把HTML格式的表格回传给客户，很简单，您可以再加一个ActionItem，设置它的Name特性为DataSet，设置PathInfo特性为/ShowTable，然后再响应OnAction事件，程序示例如下：

```
void _fastcall TWebModule1::WebModule1DataSetAction(TObject *Sender,
TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
    …
    DataSetTableProducer1->Caption = "<H1>客户表</H1>";      //设置表格的标题
DataSetTableProducer1->CaptionAlignment = caTop;            //标题的排列方式
DataSetTableProducer1->DataSet->Open( );                    //打开数据集
Response->Content = DataSetTableProducer1->Content( );
DataSetTableProducer1->DataSet->Close( );                   //及时关闭数据集
…
};
```

下面我们再介绍TQueryTableProducer元件的用法，首先要把一个TQuery元

件放到Web模块上，设置它的DatabaseName特性指定要访问的数据库如DBMEMOS，设置它的SQL特性指定要执行的SQL语句，如果您要构造很复杂的多表查询，建议您用SQL Builder来建立查询，并把得到的SQL语句复制给TQuery元件的SQL特性。然后把一个TQueryTableProducer元件放到Web模块上，设置它的Query特性指定TQuery元件。

与TDataSetTableProducer元件一样，TQueryTableProducer元件也是个Web页面生成器，它能自动生成一个HTML文档，页面上以表格的形式显示查询得到的数据。

TQueryTableProducer元件能够自动检索URL所附带的参数，如果某个参数的名称与SQL语句中的某个参数的名称匹配，就把URL参数的值赋值给SQL语句的参数。由此可见，Web服务器应用程序的设计者必须通盘考虑SQL语句的参数和窗体中的输入项，例如，假设SQL语句中有一个CustNo参数，相应地，窗体中也必须有一个CustNo输入项。

现在您可以双击Web模块打开动作编辑器再加入一个ActionItem，设置它的Name特性为Query，设置它的PathInfo特性为/Query，然后响应OnAction事件，程序示例如下：

```
void _fastcall TWebModule1::WebModule1QueryAction(TObject*Sender,
TWebRequest*Request, TWebResponse*Response, bool&Handled)
{
QueryTableProducer1->Caption ="<H1>客户表</H1>";
QueryTableProducer1->Query->Open( );
Response->Content = QueryTableProducer1->Content( );
```

```
QueryTableProducer1->Query->Close( );
};
```

TDataSetTableProducer元件和TQueryTableProducer元件都能自动生成HTML格式的表格，为了给表格增加点色彩，您可以响应OnFormatCell事件，这样您就有机会给表格的每个单元设置不同的背景颜色，程序示例如下：

```
void _fastcall TWebModule1::DataSetTableProducer1FormatCell(TObject *Sender, int CellRow, int Cell- Column, THTMLBgColor &BgColor, THTMLAlign &Align, THTMLVAlign &VAlign, AnsiString &CustomAttrs, AnsiString &CellData)
{
Switch (CellData)
{
     case "CustNo": BgColor = "Blue";
     case "Company": BgColor = "Red";
     case "IfPayed": BgColor = "Green";
}
};
```

目前只有Microsoft的Internet Explorer 3.0或更高版本支持表格单元设置背景颜色。实际上，在OnFormatCell事件中您可以做的事很多，例如，您还可以设置单元格的排列方式。

15.7 怎样调试Web服务器应用程序

Web服务器应用程序是一种特殊的应用程序，其特殊性在于它必须运行在Web服务器环境下。因此，调试Web服务器应用程序要考虑到一些特殊问题，具体的调试方式取决于Web服务器应用程序的类型及其所在的Web服务器环境。

15.7.1 调试ISAPI或NSAPI类型的Web服务器应用程序

ISAPI或NSAPI类型的Web服务器应用程序实际上是动态链接库，在C++ Builder 3中，要调试动态链接库，您首先要定位一个宿主程序(这里就是Web服务器软件)并指定运行参数。在定位宿主程序和指定运行参数之前，Web服务器软件应当还没有运行。

Windows NT 4.0的IIS环境

如果是在Windows NT 4.0的IIS环境下，您可以在C++ Builder 3的IDE中使用“Run”菜单上的“Parameters”命令打开“Run Parameters”对话框，如图15.9所示。

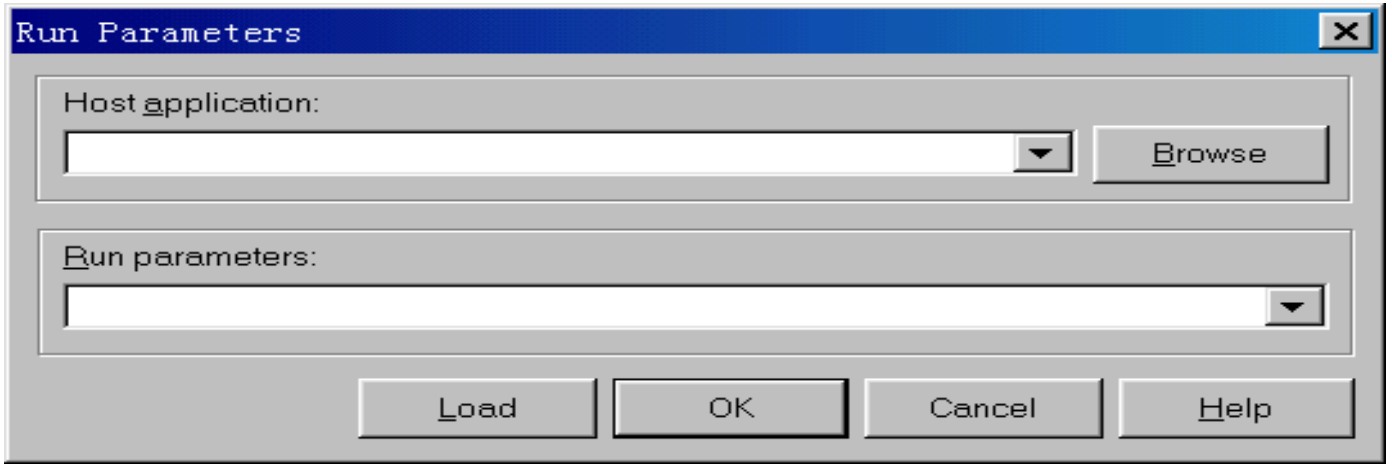


图 15.9　调试 DLL 必须设置运行参数

在“Host Application”框内键入“c:\winnt\system32\inetsrv\inetinfo.exe”，在“Run Parameters”框内键入“-e w3svc”。注意：inetinfo.exe程序的路径可能与上述示例不同。

Windows 95 的 Personal Web Server 环境

如果您的Web服务器环境是Personal Web Server For Windows 95，您应当在“Host Application”框内键入“c:\Program Files\websvc\system\inetsw95.exe”，在“Run Parameters”框内键入“-w3svc”。注意：inetsw95.exe程序的路径可能与上述示例不同。

如果您使用Windows 95 OSR2或Windows 98内置的个人Web服务器，可以参照Personal Web Server For Windows 95环境下的设置。

Netscape Server 2.0 环境

在 Netscape Server 2.0 环境下，您首先要从 ...\CBuilder3\Bin 目录中把 ISAPITER.DLL 文件复制到 C:\Netscape\Server\Nsapi\Examples 目录(实际的路径可能不同)，然后修改位于 C:\Netscape\Server\Httpd-<servername>\Config 目录的配置文件。

在 OBJ.CONF 文件中找到这么一行：

```
Init fn=load-types mime-types=mime.types
```

在这一行后插入：

```
Init funcs="handle-isapi,check-isapi,log-isapi" fn="load-modules" shlib="c:/netscape/server /nsapi/examples/ISAPIter.dll"
```

再在 OBJ.CONF 中的 <Object name=default> 段找到这么一行：

```
NameTrans fn=document-root root="C:/Netscape/Server/docs"
```

在这一行前插入：

```
NameTrans from="/scripts" fn="pfx2dir" dir="C:/Netscape/Server/docs/scripts" name="isapi"
```

然后在 OBJ.CONF 文件的末尾添加这么一个段：

```
<Object name="isapi">
PathCheck fn="check-isapi"
ObjectType fn="force-type" type="magnus-internal/isapi"
Service fn="handle-isapi"
</Object>
```

最后在 MIME.TYPES 文件的末尾添加这么一行：

```
type=magnus-internal/isapi exts=dll
```

现在您可以在C++ Builder 3的IDE中打开“Run Parameters”对话框，在“Host Application”框内键入“c:\Netscape\server\bin\httpd\httpd.exe”，在“Run Parameters”框内键入“c:\Netscape \server\httpd-<servername>\config”。

15.7.2 调试CGI或Win-CGI类型的Web服务器应用程序

CGI或Win-CGI类型的Web服务器应用程序都是单独可执行的程序，其中，CGI类型的Web服务器应用程序通过标准的输入输出设备与Web服务器交换信息，Win-CGI类型的Web服务器应用程序通过文件与Web服务器交换信息，因此，要调试Win-CGI类型的Web服务器应用程序，您可以模拟Web服务器写一个包含客户请求的配置文件，让Win-CGI类型的Web服务器应用程序感觉上就好象有一个Web服务器一样。

还有一种方式，就是先按ISAPI或NSAPI类型创建Web服务器应用程序，然后按上一小节介绍的方法调试动态链接库，调试好后再转换为CGI或Win-CGI类型的应用程序。转换也有两种方式，一种是直接修改项目文件，另一种方式是先创建一个空的CGI或Win-CGI类型的Web服务器应用程序，再把ISAPI或NSAPI类型的Web服务器应用程序中的Web模块及其代码移植过来。需要说明的是，在ISAPI或NSAPI类型的Web服务器应用程序中，有些代码是专门用于处理多线程和Web模块缓存的，这些代码在CGI或Win-CGI类型的Web服务器应用程序中是多余的，应当去掉。

第十六章　Web服务器的细节

如果您仔细阅读了前面一章的内容，并实际创建了一个Web服务器应用程序，您一定会感到，用C++ Builder 3创建Web服务器应用程序是很简单的，C++ Builder 3的确很优秀。实际上，前面一章还只是简单的介绍，忽略了很多细节，这一章我们将进一步介绍细节，包括Web模块和Web调度器、动作项、HTTP请求消息、响应客户、HTML模板和页面生成器等，只要您真正掌握了这些细节，您就成为Web服务器应用程序的设计高手。

16.1　Web服务器应用程序的逻辑结构

Web服务器应用程序实际上是Web服务器在功能上的扩展，就好象Windows应用程序是Windows在功能上的扩展一样。当Web服务器收到客户的HTTP请求时，Web服务器应用程序能够从Web服务器检索到一个HTTP请求消息，Web服务器应用程序要做的就是分析HTTP请求消息，然后作出不同的响应，更准确地说就是生成HTML页面传递给Web服务器，再由Web服务器传递给客户。除此以外，Web服务器应用程序还可以象一般的应用程序一样做任何事情。

C++ Builder 3用TWebRequest对象来描述HTTP请求消息，用TWebResponse

对象来描述Web服务器应用程序作出的反应。Web服务器应用程序中的一个关键部件是Web模块，它收集和管理着一组动作项(TWebActionItem对象)，并根据HTTP请求消息来指派其中一个动作项去响应客户的请求，实际上就是填写TWebResponse对象的Content特性。Web服务器应用程序的逻辑结构如图16.1所示。

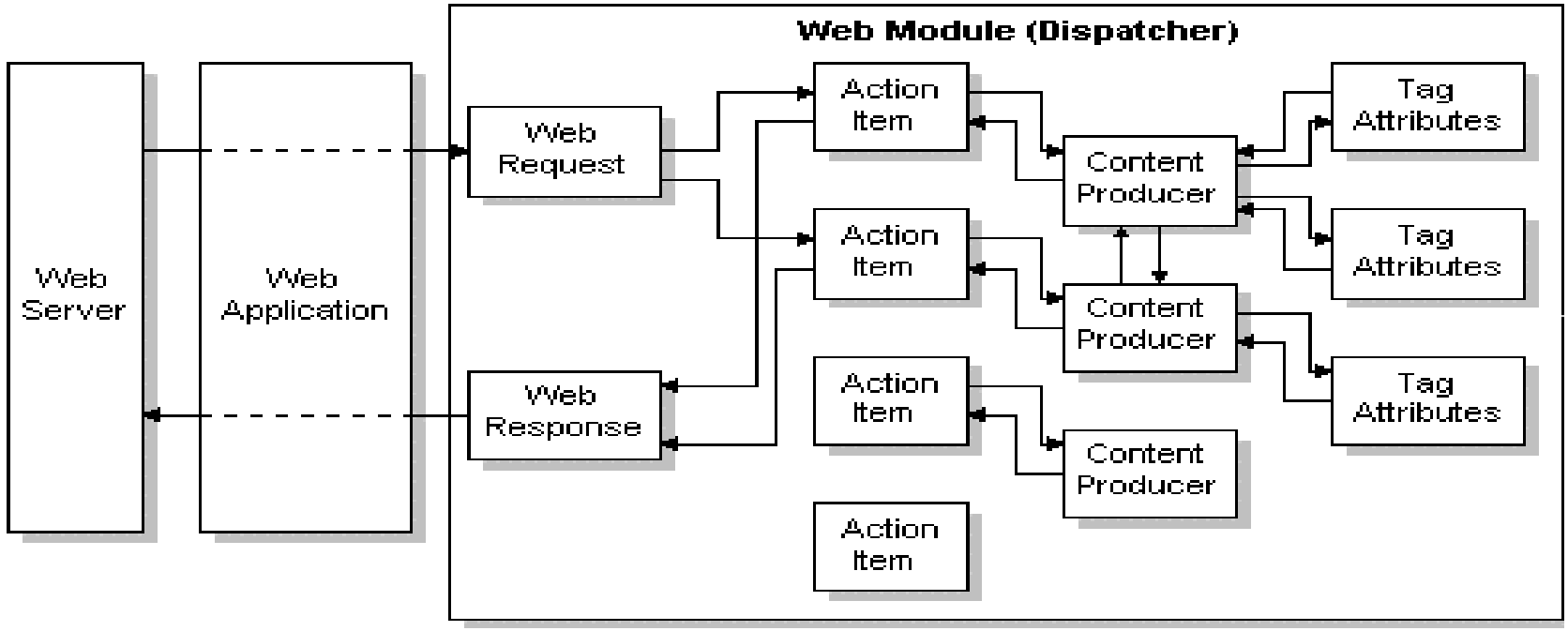


图 16.1　Web 服务器应用程序的逻辑结构

16.2 Web 模 块

当您用C++ Builder 3创建Web服务器应用程序时，它会自动包含一个空白的Web模块。Web模块是Web服务器应用程序的核心部件，它在HTTPAPP单元中是这样声明的：

```
class PASCALIMPLEMENTATION TWebModule :
public Httpapp::TCustomWebDispatcher
{
_published:
    _property Actions ;
    _property BeforeDispatch ;
    _property AfterDispatch ;
};
```

可以看出，Web模块的直接上级是TCustomWebDispatcher，而TCustomWebDispatcher又是从TDataModule继承下来的，因此，Web模块也象传统的数据模块一样，可以放一些非可视的元件如TSession、TTable、TPageProducer、TDataSetTableProducer等。

Web模块不仅仅是个容器，更主要的它还是Web服务器应用程序的调度中心，它收集和管理着一组动作项，由它决定指派哪个动作项去响应客户的请求。要说明的是，如果应用程序中已经有了Web模块，就没必要也不允许再把TWebDispatcher元件加到Web模块上，因为Web模块已经包含了TWebDispatcher元件即Web调度器的功能，事实上，Web模块和

TWebDispatcher元件都是从TCustomWebDispatcher继承下来的。注意：一个Web服务器应用程序只能有一个Web模块或调度中心。

Action特性

声明：_property TWebActionItem* Action[int Index];
通过这个只读的特性可以访问由Web模块收集和管理的每个动作项(TWebActionItem对象)，Index是动作项的索引(序号)，0代表第一个动作项，1代表第二个动作项，依次类推。如果要通过名称来访问动作项，请调用ActionByName函数。

Actions特性

声明：_property TWebActionItems* Actions;
这个特性用于管理动作项，如增加、删减动作项等。
在设计期，您可以单击这个特性边上的省略号按钮打开动作项编辑器。在运行期，您可以调用TWebActionItems对象的Add、Assign、Clear等函数。

Request特性

声明：_property TWebRequest* Request;
这个只读的特性返回Web模块正在处理的HTTP请求消息即TWebRequest对象，Web调度器根据这个特性指派相应的动作项去响应客户的请求。至此，读者应该能够理解为什么在处理TPageProducer元件的OnHTMLTag事件时能够直接引用Request。

Response 特性

声明：_property TWebResponse* Response;
这个只读的特性返回响应HTTP请求消息的TWebResponse对象。在Web服务器应用程序中，经常要对TWebResponse对象的Content特性赋值，读者可以参见上一章的示例。

ActionByName 函数

声明：TWebActionItem* _fastcall ActionByName(const System::AnsiString AName);
这个函数返回一个动作项(TWebActionItem对象)，AName参数指定动作项的名称。

构造函数

声明：_fastcall virtual TWebModule(Classes::TComponent* AOwner);
构造函数用于创建TWebModule对象的实例，然后触发OnCreate事件，这样您就有机会对Web模块初始化。要说明的是，一般情况下并不需要调用构造函数来创建Web模块，因为当您用C++ Builder 3创建Web服务器应用程序时，它会自动创建一个空白的Web模块，一个Web服务器应用程序只允许有一个Web模块。

析构函数

声明：_fastcall virtual ~TWebModule(void) { }

析构函数用于删除Web模块的实例，由Web模块收集和管理的动作项也将一并删除，同时将触发OnDestroy事件。不过，一般情况下并不需要调用析构函数，因为当Web服务器应用程序关闭时，会自动删除Web模块的实例。

OnCreate 事件

声明：_property Classes::TNotifyEvent OnCreate;
正在创建Web模块时将触发这个事件，这样您就有机会对Web模块初始化，例如，假设Web模块上包含一个TTable元件，此时就可以打开数据集。记住一点，Web服务器应用程序中的Web模块就相当于一般应用程序中的Form，因此，Web服务器应用程序的所有初始化工作都可以在处理OnCreate事件的句柄中进行。

OnDestroy 事件

声明：_property Classes::TNotifyEvent OnDestroy;
当Web正在删除的时候将触发这个事件，这样您就有机会做一些“清场”的工作，以保证先前占用的资源能得到释放。

OnBeforeDispatch 事件

声明：_property THTTPMethodEvent BeforeDispatch;
其中，THTTPMethodEvent是这样声明的：

```
typedef void _fastcall (_closure *THTTPMethodEvent)(System::TObject* Sender, TWebRequest* Request, TWebResponse* Response, bool &Handled);
```

当Web模块检索到一个HTTP请求消息时就触发这个事件，这样您就有机会在指派具体的动作项之前，先对HTTP请求消息作一个总的处理，然后再响应动作项的OnAction事件对HTTP请求消息作具体的处理。
如果不需要区分HTTP请求消息，您可以在响应OnBeforeDispatch事件的句柄中填充Response参数(TWebResponse对象)，同时把Handled参数设为True，表示消息已处理过，这样，Web模块就不再把HTTP请求消息传递给任何一个动作项，换句话说就是不再触发动作项的OnAction事件。如果您并没有填充Response参数，却把Handled参数设为True，Web模块也不把HTTP请求消息传递给任何一个动作项，但将触发OnAfterDispatch事件，这样您还有一次机会对HTTP请求消息进行处理。

OnAfterDispatch 事件

声明：_property THTTPMethodEvent AfterDispatch;
当Web模块指派了某个动作项并且已经填充了HTTP响应消息(TWebResponse对象)就触发这个事件，此时，HTTP响应消息还没有发出，这样您就有机会对TWebResponse对象作最后的修改，如果您把Handled参数设为False，表示不发出HTTP响应消息。

16.3 Web 调度器

从上一节可以看出，Web模块不仅仅是个容器，更主要的它是Web服务器应用程序的调度中心，起着举足轻重的作用。不过，有时候您可能还得用传

统的数据模块，这是因为，过去可能已经为数据模块写了很多代码，数据模块中封装了许多成熟的商业规则，如果一切从头开始就非常可惜甚至是不可能的，这时候您可以用数据模块代替Web模块，只要把一个TWebDispatcher元件加到数据模块上，这样也能构成Web服务器应用程序的调度中心。事实上，Web模块本来就扮演着容器和调度中心两种角色，现在只是分开实现而已。

TWebDispatcher元件的直接上级是TCustomWebDispatcher，在HTTPAPP单元中，TCustomWebDispatcher是这样声明的：

```
class PASCALIMPLEMENTATION TCustomWebDispatcher : public Forms::TDataModule
{
public:
    TWebActionItem*_fastcall ActionByName(const System::AnsiString AName);
    _property TWebActionItems* Actions;
    _property TWebActionItem* Action[int Index];
    _property TWebRequest* Request;
    _property TWebResponse* Response;
};
```

TWebDispatcher元件的特性、方法和事件请参见Web模块。与Web模块一样，TWebDispatcher元件也有收集和管理动作项的功能，在TWebDispatcher元件上双击鼠标左键或单击鼠标右键再选择“Action Editor”命令，将弹出动作编辑器。

16.4 动 作 项

Web模块用动作项来响应HTTP请求消息，一个动作项就是一个TWebActionItem对象。C++ Builder 3用动作编辑器来管理Web服务器应用程序的动作项，要打开动作编辑器有三种方法，一是直接双击Web模块或TWebDispatcher元件，二是用鼠标右键单击Web模块或TWebDispatcher元件，在弹出的菜单中选择“Action Editor”命令，三是在Object Inspector中找到Actions特性，单击该特性边上的省略号按钮。打开的动作编辑器如图16.2所示。

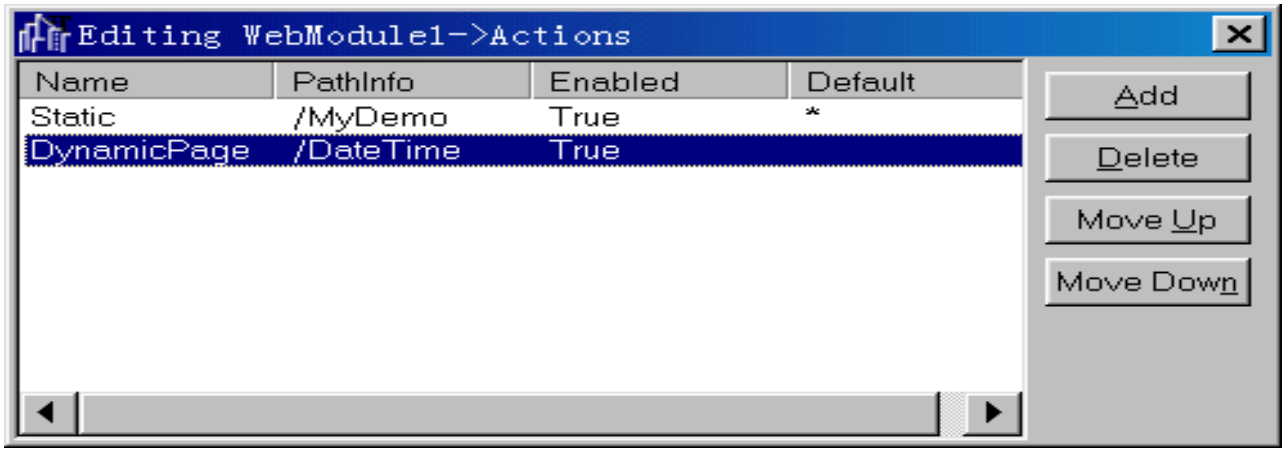


图 16.2 动作编辑器

一个Web服务器应用程序往往有多个动作项，您可以让每个动作项单独完成对HTTP请求消息的响应，也可以只进行一部分响应，让其它动作项共同

参与完成对HTTP请求的响应。最后一个动作项往往比较特殊，通常用于对HTTP响应消息作最后的检查，然后再决定是发送HTTP响应消息还是返回一个错误码。动作项一般不需要显式地发送HTTP响应消息，因为Web服务器应用程序会自动把HTML页面传递给Web服务器。
在HTTPAPP单元中，TWebActionItem对象是这样声明的：

```
class PASCALIMPLEMENTATION TWebActionItem : public Classes::TCollectionItem
{
public:
	_fastcall virtual TWebActionItem(Classes::TCollection* Collection);
	_fastcall virtual ~TWebActionItem(void);
	virtual void _fastcall AssignTo(Classes::TPersistent* Dest);
_published:
	_property bool Default;
	_property bool Enabled;
	_property TMethodType MethodType;
	_property System::AnsiString Name;
	_property System::AnsiString PathInfo;
	_property THTTPMethodEvent OnAction;
};
```

Default特性

声明：_property bool Default;

如果这个特性设为True，此动作项就成为默认的动作项。在动作编辑器中，默认动作项的“Default”栏有一个*号(见图16.2)。
当Web模块检索到一个HTTP请求消息，就在所有的动作项中查找与此消息匹配的动作项，如没有找到，就指派默认的动作项去响应此消息，即使该动作项与HTTP请求消息并不匹配，这就是Default特性的作用。因此，默认的动作项往往是最后一个动作项。
在一个Web服务器应用程序中，只能有一个动作项被指定为默认动作项。对于默认的动作项来说，如果它的Enabled特性设为False，Web调度器先从非默认的动作项中查找匹配的动作项，只有当所有的非默认动作项都不匹配时，才指派默认的动作项。如果默认动作项的Enabled特性设为True，Web调度器将从包括默认动作项在内的所有动作项中查找匹配的动作项，如果都不匹配，就指派默认的动作项。由此可见，默认的动作项有着特殊的作用，处理默认动作项的OnAction事件的句柄应当能够处理任何类型的HTTP请求消息。
注意：一般情况下，您应当在设计期就确定哪个动作项为默认动作项，尽量不要在运行期指定默认动作项，因为这可能导致不可预计的后果。
这里顺便介绍一下匹配的条件，必须同时满足下列两个条件才算是匹配，一是动作项的MethodType特性与HTTP请求消息(TWebRequest对象)的MethodType特性相同，二是动作项的PathInfo特性与HTTP请求消息(TWebRequest对象)的PathInfo特性相同。

Enabled 特性

声明：_property bool Enabled;
如果动作项的Enabled特性设为False，动作项就被暂时禁止，Web调度器将不检查此动作项是否与HTTP请求消息匹配，更不会指派该动作项去响应HTTP请求消息。Enabled特性对默认的动作项无效，如果默认的动作项的Enabled特性设为False，Web调度器将首先从非默认的并且Enabled特性设为True的动作项中查找匹配的动作项，只有当这些动作项都不匹配时，才指派默认的动作项，即使默认的动作项可能正好与HTTP请求消息匹配。

MethodType 特性

声明：_property TMethodType MethodType;
这个特性指定一个方法类型，动作项将只响应该类型的HTTP请求消息。MethodType特性可以设为以下值：mtGet、mtHead、mtPost、mtPut、mtAny。
注意：MethodType特性设为mtAny的动作项最好加在其它动作项的后面，这样能保证先找到精确匹配的动作项。

Name 特性

声明：_property System::AnsiString Name;
这个特性用于给动作项指定名称，每个动作项的名称应该是唯一的，否则将触发异常。

PathInfo 特性

声明：_property System::AnsiString PathInfo;
这个特性用于指定入口路径，动作项将只处理与此入口路径匹配的HTTP请求消息。假设HTTP请求消息的目标URL是：

http://www.TSite.com/art/gallery.dll/mammals?animal=dog&color=black
其中，gallery.dll是Web服务器应用程序，/mammals就是入口路径。

AssignTo 函数

声明：virtual void _fastcall AssignTo(Classes::TPersistent* Dest);
这个函数把一个动作项(本身)赋给由Dest参数指定的另一个动作项，包括动作项的Default、Enabled、MethodType和PathInfo等特性的值。

构造函数

声明：_fastcall virtual TWebActionItem(Classes::TCollection* Collection);
构造函数用于创建TWebActionItem对象的实例并对该实例初始化，Enabled特性设为True，PathInfo特性设为空。

析构函数

声明：_fastcall virtual ~TWebActionItem(void);
析构函数用于删除TWebActionItem对象的实例，不过，一般情况下并不需要调用析构函数，因为当Web服务器应用程序终止时会自动删除Web模块的实例，同时删除由Web模块收集和管理的所有动作项。要显式地删除

TWebActionItem对象的实例，用delete。

OnAction事件

声明：_property THTTPMethodEvent OnAction;
其中，THTTPMethodEvent是这样声明的：

```
typedef void _fastcall (_closure *THTTPMethodEvent)(System::TObject* Sender, TWebRequest*Request, TWebResponse* Response, bool &Handled);
```

这个事件可能是Web服务器应用程序中最重要的事件，上一章已多次提到。当Web调度器检索到一个HTTP请求消息并且找到了匹配的动作项，就触发这个事件，这样您就有机会对客户的请求作出反应。OnAction事件有三个参数：
Request参数是个TWebRequest对象，用于描述HTTP请求消息，动作项正是通过Request参数来分析客户发出的请求。Response参数是个TWebResponse对象，用于描述HTTP响应消息，所谓响应，实际上就是把HTML代码赋给TWebResponse对象的Content特性，这些HTML代码被传送到客户端后，由Web浏览器解释并显示成Web页面。如果此动作项已经完成了对HTTP请求消息的处理，就把Handled参数设为True。如果此动作项没有响应HTTP请求消息或只是部分响应HTTP请求消息，应当把Handled特性设为False，让其它动作项有机会作进一步的处理。
一般来说，您并不需要显式地调用SendRedirect或SendResponse来发送HTTP响应消息，因为Web服务器应用程序会自动把HTML页面传递Web服务器，再传递给客户。

16.5 HTTP请求消息

C++ Builder 3用TWebRequest对象来描述和解释HTTP请求消息，Web服务器应用程序可以从中检索到许多信息，从而作出不同的响应。

16.5.1 HTTP请求消息是怎样传递的

当客户在Web浏览器中单击一个超级链接，Web浏览器就开始收集协议种类、Web服务器的主机名、信息的入口路径、当前的日期和时间、操作系统以及Web浏览器连接状态等信息，然后把这些信息组合成HTTP请求消息发送给Web服务器。

如果Web服务器需要区分HTTP请求消息中的入口路径并分门别类地加以处理，Web服务器必须把HTTP请求消息传递给Web服务器应用程序，更准确地说就是Web模块，具体的传递方式取决于Web服务器应用程序的类型。

对于CGI类型的Web服务器应用程序来说，Web服务器与Web服务器应用程序之间是通过标准的输入输出设备交换信息的，因此，Web服务器直接把HTTP请求消息传递给CGI程序，然后等待CGI程序执行，CGI程序把响应结果直接传递给Web服务器。

对于Win-CGI类型的Web服务器应用程序来说，Web服务器与Web服务器应用程序之间通过文件交换信息，因此，Web服务器首先把HTTP请求消息写到一个配置文件中(扩展名是.INI)，然后执行Win-CGI程序并传递两个文件位置，一个是包含HTTP请求消息的配置文件的位置，另一个是让Win-CGI程序写响应结果的文件的位置。Web服务器会等待Win-CGI程序执行，执行

完毕后，就从指定的文件中读取响应结果。
对于ISAPI或NSAPI类型的Web服务器应用程序来说，当Web服务器检索到一个HTTP请求消息，它就加载动态链接库形式的ISAPI或NSAPI程序(如果ISAPI或NSAPI程序还没有加载的话)，然后把HTTP请求消息以一个结构的形式传递给ISAPI或NSAPI程序。等ISAPI或NSAPI程序执行关闭，就把响应结果直接传递给Web服务器。

16.5.2 TWebRequest是怎样声明的

TWebRequest只是一个虚拟基类，对于不同类型的Web服务器应用程序来说，描述和解释HTTP请求消息的实际上是TWebRequest的派生类，对于ISAPI或NSAPI类型的Web服务器应用程序来说就是TISAPIRequest，对于CGI类型的Web服务器应用程序来说就是TCGIRequest，对于Win-CGI类型的Web服务器应用程序来说就是TWinCGIRequest。
在HTTPAPP单元中，TWebRequest是怎样声明的：

```
class PASCALIMPLEMENTATION TWebRequest : public System::TObject
{
private:
    //私有的成员此处省略
public:
    _fastcall TWebRequest(void);
    _fastcall virtual ~TWebRequest(void);
    virtual int _fastcall ReadClient(void *Buffer, int Count);
```

```
virtual System::AnsiString _fastcall ReadString(int Count);
virtual System::AnsiString _fastcall TranslateURI(const System::AnsiString URI);
virtual int _fastcall WriteClient(void *Buffer, int Count);
virtual bool _fastcall WriteString(const System::AnsiString AString);
void _fastcall ExtractFields(const TCharSet &Separators, const TCharSet &WhiteSpace,
char * Content, Classes::TStrings* Strings);
void _fastcall ExtractContentFields(Classes::TStrings* Strings);
void _fastcall ExtractCookieFields(Classes::TStrings* Strings);
void _fastcall ExtractQueryFields(Classes::TStrings* Strings);
virtual System::AnsiString _fastcall GetFieldByName(const AnsiString Name);
_property TMethodType MethodType;
_property Classes::TStrings* ContentFields;
_property Classes::TStrings* CookieFields;
_property Classes::TStrings* QueryFields;
_property System::AnsiString Method;
_property System::AnsiString ProtocolVersion;
_property System::AnsiString URL;
_property System::AnsiString Query;
_property System::AnsiString PathInfo;
_property System::AnsiString PathTranslated;
_property System::AnsiString Authorization;
_property System::AnsiString CacheControl;
```

```
    __property System::AnsiString Cookie;
    __property System::TDateTime Date;
    __property System::AnsiString Accept;
    __property System::AnsiString From;
    __property System::AnsiString Host;
    __property System::TDateTime IfModifiedSince;
    __property System::AnsiString Referer;
    __property System::AnsiString UserAgent;
    __property System::AnsiString ContentEncoding;
    __property System::AnsiString ContentType;
    __property int ContentLength ;
    __property System::AnsiString ContentVersion;
    __property System::AnsiString Content;
    __property System::AnsiString Connection;
    __property System::AnsiString DerivedFrom;
    __property System::TDateTime Expires;
    __property System::AnsiString Title;
    __property System::AnsiString RemoteAddr;
    __property System::AnsiString RemoteHost;
    __property System::AnsiString ScriptName;
    __property int ServerPort;
};
```

TWebRequest对象的作用就是描述和解释HTTP请求消息，它的特性都是只读的。要说明的是，由于HTTP协议本身还在不断发展之中，因此，并非HTTP请求消息中的所有信息都可以通过TWebRequest对象的特性访问到，例如，有些头标域目前还没法确定，要访问这些头标域，您可以调用GetFieldByName函数。

16.5.3 TWebRequest对象的特性和方法

Accept特性

声明：_property System::AnsiString Accept;
这个特性返回HTTP请求消息中的Accept头标，Accept头标用于描述客户希望接受的媒体类型，例如，假设客户请求在HTML页面中包含得到一段声音，Accept头标可能是这样的字符串“audio/*; q=0.2, audio/basic”，表示希望的媒体类型是audio/basic。
并不是所有的HTTP请求消息都有Accept头标，如果HTTP请求消息中没有Accept头标，Accept特性就返回空，表示客户可接受所有的媒体类型。

Authorization特性

声明：_property System::AnsiString Authorization;
这个特性返回HTTP请求消息中的Authorization头标，Authorization头标描述客户的授权信息，这样Web服务器应用程序就可以根据客户的权限限制客户的某些访问。

CacheControl特性

声明：_property System::AnsiString CacheControl;
这个特性返回HTTP请求消息中的Cache-Control头标，Cache-Control头标用于描述客户发出的缓存机制指示字，HTTP 1.1标准中定义了若干个指示字：no-cache、no-store、max-age=ss、max-stale=ss或max-stale=ss、min-fresh=ss，其中，ss是秒数，一旦缓存的时间超过预定的秒数，就认为HTTP请求是无效的。

Connection特性

声明：_property System::AnsiString Connection;
这个特性返回HTTP请求消息中的Connection头标，Connection头标用于描述客户与Web服务器应用程序之间的连接选项，如果Connection头标是“Keep-Alive”，表示一直保持连接状态，如果Connection头标是“Close”，当响应结束就断开连接。

Content特性

声明：_property System::AnsiString Content;
这个特性返回HTTP请求消息中的内容(未分解)，如果HTTP请求方法是POST，该内容就是要发送到Web服务器的信息，您可以通过ContentFields特性访问分解后的每一个域。如果HTTP请求方法是PUT，该内容就是要替代URL特性所定位的内容。

ContentEncoding 特性

声明：_property System::AnsiString ContentEncoding;
这个特性返回HTTP请求消息中的Content-Encoding头标，Content-Encoding头标用于描述HTTP请求消息中的内容的编码方式，也就是怎样压缩和传输扩展的ASCII字符。如果有多种编码方式的话，这些编码方式按客户指定的顺序依次排列，彼此之间用逗号隔开。

ContentFields 特性

声明：_property Classes::TStrings* ContentFields;
这个特性是TStrings对象，用于返回HTTP请求消息中的内容(已分解)，其中每个字符串的形式是Name = Value，字符串之间用&符号隔开。

ContentLength 特性

声明：_property int ContentLength;
这个特性返回HTTP请求消息中的Content-Length头标，Content-Length头标是HTTP请求消息中内容的长度，如果ContentLength特性返回0表示长度不确定，这可能是因为HTTP请求消息中没有包含任何内容，也可能是因为客户无法确定内容的长度。

ContentType 特性

声明：_property System::AnsiString ContentType;
这个特性返回HTTP请求消息中的Content-Type头标，Content-Type头标用于

描述HTTP请求消息中内容的媒体类型，形式是Type/SubType，例如text/html。

ContentVersion 特性

声明：_property System::AnsiString ContentVersion;
这个特性返回HTTP请求消息中的Content-Version头标，Content-Version头标用于描述HTTP请求消息中内容的版本。大多数HTTP请求消息不需要这个头标。

Cookie 特性

声明：_property System::AnsiString Cookie;
这个特性返回HTTP请求消息中的Cookie头标(未分解)，Cookie头标用于跟踪客户访问站点的踪迹。Cookie头标可能由多个域组成，彼此之间用分号隔开，通过CookieFields特性可以访问Cookie头标的每一个域。

CookieFields 特性

声明：_property Classes::TStrings* CookieFields;
这个特性是TStrings对象，用于访问HTTP请求消息中Cookie头标的每一个域，每个字符串的形式是Name=Value。注意：Web服务器应用程序可以随便给状态变量取名，但不能是Cookie中的域名，换句话说，Cookie中的域名是保留的。

Date 特性

声明：_property System::TDateTime Date;
这个特性返回HTTP请求消息中的Date头标，Date头标用于描述发起HTTP请求消息时的日期和时间，这样，Web服务器应用程序就知道HTTP请求消息中的内容是否太老。

DerivedFrom 特性

声明：_property System::AnsiString DerivedFrom;
这个特性返回HTTP请求消息中的Derived-From头标，Derived-From头标用于描述HTTP请求消息中内容的URI。

Expires 特性

声明：_property System::TDateTime Expires;
这个特性返回HTTP请求消息中的Expires头标，Expires头标实际上是日期和时间，如果超过了这个日期和时间，就认为HTTP请求消息是过期的。注意：如果HTTP请求消息中有Cache-Control头标并且指示字是max-age的话，Expires特性无效。

From 特性

声明：_property System::AnsiString From;
这个特性返回HTTP请求消息中的From头标，From头标很有用，它是客户的e-mail地址，Web服务器应用程序可以记录访问者，或者与客户联系。

Host 特性

声明：_property System::AnsiString Host;
这个特性返回HTTP请求消息中的Host头标，Host头标是客户正在请求的主机名和端口号，假设URL是：http://www.TSite.com/art/gallery.dll/mammals?animal=dog&color=black，主机名就是其中的www.TSite.com。

IfModifiedSince 特性

声明：_property System::TDateTime IfModifiedSince;
这个特性返回HTTP请求消息中的If-Modified-Since头标，Web服务器应用程序应当先判断自If-Modified-Since头标设定的日期和时间以来客户所请求的信息是否修改过。注意：日期和时间以Web服务器端为准。

Method 特性

声明：_property System::AnsiString Method;
这个特性返回HTTP请求消息中的Method头标，Method头标用于描述客户发出HTTP请求的目的，HTTP 1.1标准中定义了下列种类：OPTIONS、GET、HEAD、POST、PUT、DELETE、TRACE等。对于Web服务器应用程序来说，并不是所有种类的HTTP请求都需要响应，不过GET和HEAD是必须响应的。

MethodType 特性

声明：_property TMethodType MethodType;
这个特性的含义与Method特性是基本一致的，只是用法不同。Method特性是字符串，MethodType特性是枚举常量，可以直接用来比较，而字符串的比较相对麻烦些。
如果MethodType特性返回的是mtAny，就得用Method特性才知道确切的方法类型。

PathInfo 特性

声明：_property System::AnsiString PathInfo;
这个特性返回HTTP请求消息中的URL中的入口路径。例如，假设URL是：http://
www.TSite.com/art/gallery.dll/mammals?animal=dog&color=black，入口路径就是/mammals。

PathTranslated 特性

声明：_property System::AnsiString PathTranslated;
这个特性类似于PathInfo特性，不同的是，它还包含完整的路径。

ProtocolVersion 特性

声明：_property System::AnsiString ProtocolVersion;
这个特性返回客户端所使用的HTTP协议的版本号，如果HTTP协议的版本

号不同，HTTP请求消息的解释也稍有不同，例如，对于HTTP协议1.0来说，即使Connection头标没有设为“Close”，客户与Web服务器应用程序的连接也不是永久的。

Query特性

声明：_property System::AnsiString Query;
这个特性返回HTTP请求消息中URL的参数部分，假设URL是：http://www.TSite.com
/art/gallery.dll/mammals?animal=dog&color=black，参数部分就是animal=dog&color=black。

QueryFields特性

声明：_property Classes::TStrings* QueryFields;
这个特性是TStrings对象，返回已分解的HTTP请求消息中URL的参数部分，每个字符串的形式是Name=Value。

Referer特性

声明：_property System::AnsiString Referer;
这个特性返回HTTP请求消息中的Referer头标，Referer头标是资源的URI，Web服务器应用程序用它建立相关的超级链接。

RemoteAddr 特性

声明：_property System::AnsiString RemoteAddr;
这个特性返回客户端的IP地址，例如127.8.34.19。

RemoteHost 特性

声明：_property System::AnsiString RemoteHost;
这个特性返回客户端的主机名(不包括路径)，例如www.borland.com。

ScriptName 特性

声明：_property System::AnsiString ScriptName;
这个特性返回HTTP请求消息中URL的脚本部分，假设URL是：http://www.TSite.com /art/gallery.dll/mammals?animal=dog&color=black，脚本部分就是/art/gallery。

ServerPort 特性

声明：_property int ServerPort;
这个特性返回Web服务器上检索HTTP请求消息的端口号。

Title 特性

声明：_property System::AnsiString Title;
这个特性返回HTTP请求消息中的Title头标，Title头标描述客户要请求的信息的头标。

URL 特性

声明：_property System::AnsiString URL;
前面已多次提到这个特性，在实际应用中，一般要把URL分解成Host、ScriptName、PathInfo和Query等四大部分。

UserAgent 特性

声明：_property System::AnsiString UserAgent;
这个特性返回HTTP请求消息中的User-Agent头标(Web浏览器的名称和版本号)。

ExtractContentFields 函数

声明：void _fastcall ExtractContentFields(Classes::TStrings* Strings);
在MethodType特性返回mtPost的前提下，这个函数把HTTP请求消息中的内容分解成一个个域存到Strings参数中(假设域与域之间用&隔开)。

ExtractCookieFields 函数

声明：void _fastcall ExtractCookieFields(Classes::TStrings* Strings);
这个函数把HTTP请求消息中的Cookie(如果有的话)分解成一个个域存到Strings参数中(假设域与域之间用分号隔开)。

ExtractFields 函数

声明：void _fastcall ExtractFields(const TCharSet &Separators, const TCharSet

&WhiteSpace, char * Content, Classes::TStrings* Strings);
这个函数用于把由多个域组成的字符串分解为一个个域，其中，Separators参数指定一个字符集合，该集合中的字符用于分隔域。WhiteSpace参数也是字符集合，该集合中的字符是要忽略的。Content参数就是要分解的字符串，Strings参数用于存储分解后的域。

ExtractQueryFields 函数

声明：void _fastcall ExtractQueryFields(Classes::TStrings* Strings);
这个函数把HTTP请求消息中的Query分解为一个个域存到Strings参数中，假设域与域之间用分号隔开。

GetFieldByName 函数

声明：virtual System::AnsiString _fastcall GetFieldByName(const AnsiString Name);
一般来说，您可以由TWebRequest对象来访问HTTP请求消息中的头标，不过，由于HTTP协议还在不断地发展，C++ Builder 3不能保证HTTP协议的每一个头标都能由TWebRequest对象来访问。这个函数允许通过名称从HTTP请求消息中返回指定的头标。

ReadClient 函数

声明：virtual int _fastcall ReadClient(void *Buffer, int Count);
这个函数从HTTP请求消息的内容中读取Count个字节到Buffer缓冲区中，并

返回实际返回的字节数。

ReadString 函数

声明：virtual System::AnsiString _fastcall ReadString(int Count);
这个函数从HTTP请求消息的内容中读取Count个字节到一个字符串中。

WriteClient 函数

声明：virtual int _fastcall WriteClient(void *Buffer, int Count);
这个函数从Buffer缓冲区中取Count个字节发送给客户，并返回实际写的字节数。

WriteString 函数

声明：virtual bool _fastcall WriteString(const System::AnsiString AString);
这个函数把一个字符串发送给客户，如果成功就返回True。

16.5.4 TISAPIRequest对象

对于ISAPI或NSAPI类型的Web服务器应用程序来说，描述和解释HTTP请求消息的是TISAPIRequest对象，它在ISAPIAPP单元中是这样声明的：

```
class PASCALIMPLEMENTATION TISAPIRequest : public Httpapp::TWebRequest
{
private:
    //私有的成员此处省略
```

```
public:
    _fastcall TISAPIRequest(Isapi2::PEXTENSION_CONTROL_BLOCK AECB);
    virtual System::AnsiString _fastcall GetFieldByName(const AnsiString Name);
    virtual int _fastcall ReadClient(void *Buffer, int Count);
    virtual System::AnsiString _fastcall ReadString(int Count);
    virtual System::AnsiString _fastcall TranslateURI(const AnsiString URI);
    virtual int _fastcall WriteClient(void *Buffer, int Count);
    virtual bool _fastcall WriteString(const System::AnsiString AString);
    _property Isapi2::PEXTENSION_CONTROL_BLOCK ECB;
    _fastcall virtual ~TISAPIRequest(void) { }
};
```

可以看出，TISAPIRequest是从TWebRequest继承下来的，最主要的变化是增加了ECB特性，并且重载了构造函数用于创建TISAPIRequest对象的实例。

ECB特性

声明：_property Isapi2::PEXTENSION_CONTROL_BLOCK ECB;

前面讲过，Web服务器与ISAPI或NSAPI类型的Web服务器应用程序之间通过一个结构来传递HTTP请求消息，这就是TEXTENSION_CONTROL_BLOCK结构，ECB特性可以返回指向该结构的指针。要说明的是，在实际编程中，您并不一定需要直接与ECB打交道，因为C++ Builder 3已经把HTTP请求消息中的头标分解出来了。关于TEXTENSION_CONTROL_BLOCK结构您可以查阅C++ Builder 3的帮助。

构造函数

声明：_fastcall TISAPIRequest(Isapi2::PEXTENSION_CONTROL_BLOCK AECB);
构造函数用于创建TISAPIRequest对象的实例，要说明的是，您并不需要调用构造函数，因为当ISAPI或NSAPI类型的Web服务器应用程序检索到一个HTTP请求消息，就会自动创建TISAPIRequest对象。这里只是想让您知道，调用构造函数时传递了ECB块。

16.5.5 TCGIRequest对象

对于CGI类型的Web服务器应用程序来说，描述和解释HTTP请求消息的是TCGIRequest对象，它在CGIAPP单元中是这样声明的：

```
class PASCALIMPLEMENTATION TCGIRequest : public Httpapp::TWebRequest
{
private:
    //私有的成员此处省略
protected:
    //保护的成员此处省略
public:
    _fastcall TCGIRequest(void);
    virtual System::AnsiString _fastcall GetFieldByName(const AnsiString Name);
    virtual int _fastcall ReadClient(void *Buffer, int Count);
    virtual System::AnsiString _fastcall ReadString(int Count);
```

```
    virtual System::AnsiString _fastcall TranslateURI(const AnsiString URI);
    virtual int _fastcall WriteClient(void *Buffer, int Count);
    virtual bool _fastcall WriteString(const System::AnsiString AString);
};
```

可以看出，TCGIRequest是从TWebRequest继承下来的，它的特性和方法可以参见前面关于TWebRequest对象的介绍。

16.5.6 TWinCGIRequest对象

对于Win-CGI类型的Web服务器应用程序来说，描述和解释HTTP请求消息的是TWinCGIRequest对象，它在CGIAPP单元中是这样声明的：

```
class PASCALIMPLEMENTATION TWinCGIRequest : public Cgiapp::TCGIRequest
{
private:
    //私有的成员此处省略
public:
    _fastcall TWinCGIRequest(System::AnsiString IniFileName, System::AnsiString
    ContentFile, System::AnsiString OutputFile);
    _fastcall virtual ~TWinCGIRequest(void);
    virtual System::AnsiString _fastcall GetFieldByName(const AnsiString Name);
    virtual int _fastcall ReadClient(void *Buffer, int Count);
    virtual System::AnsiString _fastcall ReadString(int Count);
    virtual System::AnsiString _fastcall TranslateURI(const AnsiString URI);
```

```
    virtual int _fastcall WriteClient(void *Buffer, int Count);
    virtual bool _fastcall WriteString(const System::AnsiString AString);
};
```

可以看出，TWinCGIRequest是从TCGIRequest继承下来的，它的特性和方法可以参见前面关于TWebRequest对象的介绍。

16.6 HTTP响应消息

C++ Builder 3用TWebResponse对象来描述HTTP响应消息。当Web服务器应用程序检索到一个HTTP请求消息，它就创建一个TWebRequest对象来描述和解释HTTP请求消息，同时还创建一个TWebResponse对象准备响应客户的请求。

16.6.1 怎样建立HTTP响应消息

大多数情况下，客户请求的是HTML页面，也就是说，HTTP响应消息中应当包含内容，因此，您至少要在一个OnAction事件的句柄中对TWebResponse对象的Content特性或ContentStream特性赋值。不过，有的情况下，只需要填充一些头标就够了，不需要内容。

要对Content特性或ContentStream特性赋值，您既可以用事先设计到的HTML代码，也可以用TCustomPageProducer的派生对象如TPageProducer或TDataSetTableProducer自动生成页面，然后再赋值给Content特性或ContentStream特性。

Content与ContentStream的区别是，Content只能是HTML代码，C++ Builder 3并不检查这些代码的语法，它只把这些代码当作一个个字符串。如果HTML代码事先存在一个文件中，这时候用ContentStream比较方便，因为您可以用TFileStream对象读写文件。请注意两点，一是读写完毕后您不需要删除TFileStream对象，TWebResponse对象会自动删除它。二是如果ContentStream特性有值的话，Content特性就被忽略。

16.6.2 怎样传递HTTP响应消息

HTTP响应消息首先传递给Web服务器，再由Web服务器传递给客户。对于不同类型的Web服务器应用程序来说，传递HTTP响应消息的方式也是不同的。

对于Win-CGI类型的Web服务器应用程序来说，它是把HTML页面写到一个文件中，把其它信息写到另一个文件中，然后把这两个文件的位置传递给Web服务器，再由Web服务器打开这两个文件，把HTML页面传递给客户。

对于ISAPI或NSAPI类型的Web服务器应用程序来说，它是直接把HTML页面和其它信息传递给Web服务器，再由Web服务器传递给客户。

从这里可以看出，Win-CGI类型的Web服务器应用程序可能要频繁地写磁盘，而ISAPI或NSAPI类型的Web服务器应用程序就没有这方面的开销。

16.6.3 自己传递HTTP响应消息

一般来说，您只要建立HTTP响应消息就够了，并不需要考虑怎样传递HTTP响应消息，因为Web服务器应用程序会自动传递HTTP响应消息。如果您能

够确信已经最终完成了响应，您也可以自己发送HTTP响应消息。TWebResponse对象提供了两个方法SendResponse和SendRedirect，其中，SendResponse用于发送HTTP响应消息包括内容和所有的头标，SendRedirect用于把客户的请求重新定向。注意：如果响应是分几步进行的，在最终完成响应之前不要发送HTTP响应消息，因为这时候的HTTP响应消息是不完整的。

TWebResponse只是个虚拟基类，对于不同类型的Web服务器应用程序来说，描述HTTP响应消息的实际上是TWebResponse的派生类，例如，对于ISAPI或NSAPI类型的Web服务器应用程序来说就是TISAPIResponse，对于CGI类型的Web服务器应用程序来说就是TCGIResponse，对于Win-CGI类型的Web服务器应用程序来说就是TWinCGIResponse。

16.6.4 TWebResponse是怎样声明的

TWebResponse只是一个虚拟基类，对于不同类型的Web服务器应用程序来说，操纵HTTP响应消息的实际上是TWebResponse的派生类，对于ISAPI或NSAPI类型的Web服务器应用程序来说就是TISAPIResponse，对于CGI类型的Web服务器应用程序来说就是TCGIResponse，对于Win-CGI类型的Web服务器应用程序来说就是TWinCGIResponse。

在HTTPAPP单元中，TWebResponse是怎样声明的：

```
class PASCALIMPLEMENTATION TWebResponse : public System::TObject
{
private:
```

```
    //私有的成员此处省略
protected:
    //保护的成员此处省略
public:
    _fastcall TWebResponse(TWebRequest* HTTPRequest);
    _fastcall virtual ~TWebResponse(void);
    System::AnsiString _fastcall GetCustomHeader(const System::AnsiString Name);
    virtual void _fastcall SendResponse(void) = 0;
    virtual void _fastcall SendRedirect(const System::AnsiString URI) = 0;
    virtual void _fastcall SendStream(Classes::TStream* AStream) = 0;
    void _fastcall SetCookieField(Classes::TStrings* Values, const AnsiString ADomain,
    const System::AnsiString APath, TDateTime AExpires, bool ASecure);
    void _fastcall SetCustomHeader(const AnsiString Name, const AnsiString Value);
    _property TCookieCollection* Cookies = {read=FCookies};
    _property TWebRequest* HTTPRequest = {read=FHTTPRequest};
    _property System::AnsiString Version;
    _property System::AnsiString ReasonString;
    _property System::AnsiString Server;
    _property System::AnsiString WWWAuthenticate;
    _property System::AnsiString Realm;
    _property System::AnsiString Allow;
    _property System::AnsiString Location;
```

```
    _property System::AnsiString ContentEncoding;
    _property System::AnsiString ContentType;
    _property System::AnsiString ContentVersion;
    _property System::AnsiString DerivedFrom;
    _property System::AnsiString Title;
    _property int StatusCode;
    _property int ContentLength;
    _property System::TDateTime Date;
    _property System::TDateTime Expires;
    _property System::TDateTime LastModified;
    _property System::AnsiString Content;
    _property Classes::TStream* ContentStream;
    _property System::AnsiString LogMessage;
    _property Classes::TStrings* CustomHeaders;
};
```

16.6.5 TWebResponse对象的特性和方法

Allow特性

声明：_property System::AnsiString Allow;
这个特性向客户表明您这个Web服务器应用程序可以响应哪些方法种类的HTTP请求。Allow是个字符串，方法名称之间用逗号隔开，其中至少要包

含GET和HEAD。

Content特性

声明：_property System::AnsiString Content;
这个特性很重要，用于指定要传递给客户的HTTP响应消息中的内容，一般就是HTML页面。不过，有些类型的请求并不需要HTTP响应消息中包含内容，因此，像OPTIONS、HEAD、POST、PUT、DELETE、TRACE等类型的请求就不需要设置Content特性。

ContentStream特性

声明：_property Classes::TStream* ContentStream;
这个特性与Content特性相似，也是用于指定传递给客户的HTTP响应消息中的内容，不同的是，ContentStream适合于从流中获取内容，而Content特性是个字符串。

ContentEncoding特性

声明：_property System::AnsiString ContentEncoding;
这个特性用于设置HTTP响应消息中内容的编码方式，编码方式之间用逗号隔开。

ContentLength特性

声明：_property int ContentLength;

这个特性用于给出HTTP响应消息中内容的长度(字节数)。如果HTTP响应消息中没有内容或者长度是未知的，就把ContentLength特性设为0。

ContentType 特性

声明：_property System::AnsiString ContentType;
这个特性设置HTTP响应消息中内容的媒体类型，形式是Type/SubType，例如text/html。

ContentVersion 特性

声明：_property System::AnsiString ContentVersion;
这个特性用于设置HTTP响应消息中内容的版本，其含义与ContentType特性所指定的媒体类型有关，假设媒体类型是text/html，ContentVersion就是HTML语言的版本。

Cookies 特性

声明：_property TCookieCollection* Cookies;
Cookies是TCookieCollection对象，用于收集和管理HTTP响应消息中的Cookie，每个Cookie是一个TCookie对象。程序示例如下：

```
//---------------------------------------------------------------------
bool SecureCookies(TWebResponse *Response)
{
for (int I = 0; I < Response->Cookies->Count; I++)
```

```
    if (Response->Cookies->Items[I]->Secure = = true)
        return true;
    return false;
}
```

CustomHeaders 特性

声明：_property Classes::TStrings* CustomHeaders;
如果您要向HTTP响应消息中加入一个头标，但是TWebResponse对象却没有提供相应的特性，此时就要用CustomHeaders自己定义一组头标，每个头标的形式是HeaderName=Value，其中，HeaderName是头标名，Value是该头标的值。

Date 特性

声明：_property System::TDateTime Date;
在开始发送HTTP响应消息之前，把Date特性设为当前的日期和时间，因为有的客户希望知道信息是否陈旧。

DerivedFrom 特性

声明：_property System::AnsiString DerivedFrom;
这个特性用于通知客户HTTP响应消息中内容的URI。HTTP响应消息中的内容一般取自于服务器本身的URL，但也可能取自于其它路径，这时候就要设置DerivedFrom特性。

Expires 特性

声明：_property System::TDateTime Expires;
这个特性设定一个日期和时间，如果客户收到HTTP响应消息超过这个日期和时间，客户可以认为信息是过时的和非法的。

HTTPRequest 特性

声明：_property TWebRequest* HTTPRequest;
这个只读的特性返回当前正在处理的HTTP请求消息，这样您就可以分析其中的头标和内容，从而作出不同的响应。当然，如果是在OnAction事件中，您可以通过Request参数直接访问HTTP请求消息，无须借助于TWebResponse对象的HTTPRequest特性。

LastModified 特性

声明：_property System::TDateTime LastModified;
这个特性用于表明HTTP响应消息中内容的最后修改日期和时间，这样客户就可以先用HEAD方法请求头标，然后判断信息是否最新，如否，就不必传输内容了。

Location 特性

声明：_property System::AnsiString Location;
如果需要由其它URI共同参与完成响应，应当设置Location特性为该URI。

LogMessage 特性

声明：_property System::AnsiString LogMessage;
这个特性用于在HTTP响应消息中附带一个描述性的字符串，例如，您可以把它设为处理客户请求的URI以及内容的日期和时间。

Realm 特性

声明：_property System::AnsiString Realm;
如果客户的访问未被授权或需要支付费用，Web服务器应用程序返回401或402，此时应当设置Realm特性通知客户哪些URI需要口令或支付费用。

ReasonString 特性

声明：_property System::AnsiString ReasonString;
您可以通过StatusCode特性设置状态代码，用ReasonString特性给每个状态代码附带一个简短的说明。

Server 特性

声明：_property System::AnsiString Server;
这个特性用于在HTTP响应消息中给出Web服务器应用程序的名称和版本号，形式为程序名/版本号。

StatusCode 特性

声明：_property int StatusCode;

每一个HTTP响应消息必须包含一个状态代码，以表明响应状态，这是通过StatusCode特性设置的。HTTP协议中定义了若干个标准的状态代码：
100(继续处理)、101(切换协议)、200(　OK)、201(已创建)、202(已接受)、203(未授权的信息)、204(没有内容)、205(重设内容)、206(部分内容)、300(多项选择)、301(永久删除)、302(临时删除)、303(参见)、304(未修改)、305(使用代理)、400(错误的请求)、401(未授权)、402(需支付费用)、403(禁止)、404(未找到)、405(不允许的方法)、406(不可接受)、407(需代理授权)、408(请求超时)、409(冲突)、410(Gone)、411(需要给出长度)、412(预处理失败)、413(实体太大)、414(URI太大)、415(不支持的媒体类型)、500(服务器内部出错)、501(未实现)、502(错误的网关)、503(不能提供服务)、504(网关超时)、505(不支持的HTTP版本)。
除了上述标准的状态代码以外，您也可以定义自己的状态代码。每个状态代码都是三位数，最高的一位表示响应类别，1表示客户的请求已收到，但未能完全处理。2表示客户的请求已收到，并且已成功地处理。3表示需要客户进一步提示才能最终完成响应。4表示客户的请求不能被识别。5表示客户的请求是合法的，但服务器无法处理。
您可以给每一个状态代码附带一个字符串作为简短的说明，这是通过ReasonString特性实现的，不过，对于预定义的状态代码，您不要再附带说明。

Title 特性

声明：_property System::AnsiString Title;

这个特性用于给出HTTP响应消息中内容的头标，通常就是内容的主题。

Version 特性

声明：_property System::AnsiString Version;
这个特性给出HTTP协议的版本号。

GetCustomHeader 函数

声明：System::AnsiString _fastcall GetCustomHeader(const System::AnsiString Name);
这个函数返回某个自定义的头标的值，Name参数是头标的名称。

SendRedirect 函数

声明：virtual void _fastcall SendRedirect(const System::AnsiString URI);
如果您需要把客户的请求重新定向到其它URI，就要调用SendRedirect函数。调用SendRedirect函数相当于先把StatusCode特性设为301，把Location特性设为URI参数，然后调用SendResponse发送HTTP响应消息。

SendResponse 函数

声明：virtual void _fastcall SendResponse(void);
如果您想自己发送HTTP响应消息就调用SendResponse函数。

SendStream 函数

声明：virtual void _fastcall SendStream(Classes::TStream* AStream);
如果HTTP响应消息中的内容是以流的形式给出的，调用SendStream来发送HTTP响应消息就更方便些。注意：在调用SendStream之前，先要调用SendResponse。

SetCookieField 函数

声明：void _fastcall SetCookieField(Classes::TStrings* Values, const System::AnsiString ADomain, const System::AnsiString APath, System::TDateTime AExpires, bool ASecure);
这个函数在HTTP响应消息中加入一个Cookie头标，Values参数就是要加入的Cookie头标，它是TStrings对象，其中每个字符串的形式是Name/Value。ADomain参数和APath参数组成Cookie要发送给哪个URL。AExpires参数给出一个日期和时间，超过这个时间Cookie就是无效的。如果ASecure参数设为True，表示只在安全连接的情况下传递Cookie。

SetCustomHeader 函数

声明：void _fastcall SetCustomHeader(const System::AnsiString Name, const System::An-siString Value);
这个函数向自定义的头标即CustomHeaders特性中加入一个新的头标，或者改变已有头标的值。Name参数是头标的名称，Value参数是头标的值。

16.6.6 TISAPIResponse对象

对于ISAPI或NSAPI类型的Web服务器应用程序来说，操纵HTTP响应消息的是TISAPIResponse对象。在ISAPIAPP单元中，TISAPIResponse是这样声明的：

```
class PASCALIMPLEMENTATION TISAPIResponse : public Httpapp::TWebResponse
{
private:
    //私有的成员此处省略
protected:
    //保护的成员此处省略
public:
    _fastcall TISAPIResponse(Httpapp::TWebRequest* HTTPRequest);
    virtual void _fastcall SendResponse(void);
    virtual void _fastcall SendRedirect(const System::AnsiString URI);
    virtual void _fastcall SendStream(Classes::TStream* AStream);
    _fastcall virtual ~TISAPIResponse(void) { }
};
```

可以看出，TISAPIResponse是从TWebResponse继承下来的，它没有自己的特性和方法，只是重载了TWebResponse的构造函数、SendResponse、SendRedirect、SendStream。

16.6.7 TCGIResponse对象

对于CGI类型的Web服务器应用程序来说，操纵HTTP响应消息的是TCGIResponse对象。在CGIAPP单元中，TCGIResponse是这样声明的：

```
class PASCALIMPLEMENTATION TCGIResponse : public Httpapp::TWebResponse
{
private:
    //私有的成员此处省略
protected:
    //保护的成员此处省略
public:
    _fastcall TCGIResponse(Httpapp::TWebRequest* HTTPRequest);
    virtual void _fastcall SendResponse(void);
    virtual void _fastcall SendRedirect(const System::AnsiString URI);
    virtual void _fastcall SendStream(Classes::TStream* AStream);
    _fastcall virtual ~TCGIResponse(void) { }
};
```

可以看出，TCGIResponse也是从TWebResponse继承下来的，它没有自己的特性和方法，只是重载了TWebResponse的构造函数、SendResponse、SendRedirect、SendStream。

16.6.8 TWinCGIResponse对象

对于Win-CGI类型的Web服务器应用程序来说，操纵HTTP响应消息的是

TWinCGIResponse对象。在CGIAPP单元中，TWinCGIResponse的声明很简单：

```
TWinCGIResponse = Class(TCGIResponse);
```

可以看出，TWinCGIResponse是从TCGIResponse继承下来的，它没有自己的成员。

16.7 页面生成器

TPageProducer元件能够基于一个事先设计好的HTML模板分析客户的输入并自动生成动态的HTML页面，因此TPageProducer元件又叫页面生成器。TPageProducer元件的直接上级是TCustomPageProducer，而TCustomPageProducer又是从TCustomContentProducer继承下来的。

16.7.1 怎样建立HTML模板

要使用页面生成器，首先要建立HTML模板，让页面生成器知道应该怎样转换。所谓HTML模板，就是HTML命令和HTML透明标记组成的序列。关于HTML命令，读者可以参阅有关HTML语言的书籍。一个HTML透明标记的形式是：

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

可以看出，HTML透明标记必须用< >括起来，#与<之间不能有空格，#后是标记名，标记名可以是任何C++的标识符，#与标记名之间也不能有空格。标记名后可以跟一个或多个参数，以进一步指定转换的细节。每个参数的

形式是ParamName=Value，其中，ParamName是参数名，Value是参数的值，参数名、等号、参数值之间均不能有空格，但参数与参数之间要用空格隔开。

预定义的标准标记名

标记名	对应的TTag值	该标记转换为
Link	tgLink	超级链接，相当于HTML中的<A>...</A>序列
Image	tgImage	图像，相当于HTML中的<IMG=...>
Table	tgTable	表格，相当于HTML中的<TABLE>...</TABLE>序列
Image Map	tgImageMap	图像，相当于HTML中的<MAP"...">...</MAP>序列
Object	tgObject	嵌入的ActiveX对象，相当于HTML中的<OBJECT>...</OBJECT>序列
Embed	tgEmbed	与Netscape兼容的插件DLL，相当于HTML中的<EMBED>...</EMBED>序列

预定义的标记名是大小写敏感的。另外，尽管是预定义的标记名，但C++Builder 3并没有提供默认的处理，您得自己处理这些标记。您也可以自己定义标记名，任何自定义的标记对应的TTag值都是tgCustom。下面是一个HTML模板的示例：

```
<HTML>

<HEAD>
<TITLE>这是一个HTML模板示例</TIOTLE>
</HEAD>

<BODY>

<H1>感谢您加入C++ Builder爱好者协会</H1>

<H3>下面是您刚才填写的个人资料</H3>

<TABLE BORDER="0">
<TR>
    <TD>Name: </TD>
    <TD><#Name></TD>
</TR>
<TR>
    <TD>Address: </TD>
    <TD><#Address></TD>
</TR>
<TR>
```

```
    <TD>Phone: </TD>
    <TD><#Phone></TD>
</TR>
<TR>
    <TD>EMail: </TD>
    <TD><#EMail></TD>
</TR>
</TABLE>

<H5>您的资料已保存到会员数据库中，我们将经常与您联系，谢谢</H5>

</BODY>
</HTML>
```

可以看出，上述HTML模板中有四个HTML透明标记，分别是<#Name>、<#Address>、<#Phone>、<#EMail>。

16.7.2 指定HTML模板

建立了HTML模板后，您应当指定HTML模板。指定HTML模板有两种方式：如果HTML模板保存在文件中，用HTMLFile特性比较合适，只要把TPageProducer元件的HTMLFile特性设为HTML模板所在的文件名(可以包括路径)。注意：您在交付此Web服务器应用程序时应一并交付HTML模板文件，否则，Web服务器应用程序就不能正常工作。另外，HTML模板文件的

路径最好让用户自己设定，而不要在程序中写死。
如果HTML模板是用TStrings对象保存的，用HTMLDoc特性比较方便，只要把包含HTML模板的TStrings对象赋值给HTMLDoc特性就行了。
HTMLFile特性和HTMLDoc特性是互斥的，设置了其中一个，另一个就清空。

16.7.3 转换并返回转换后的结果

用HTMLFile特性或HTMLDoc特性指定HTML模板后，TPageProducer提供了三种转换方式：
一是调用Content函数，这个函数能够逐一替换HTML模板中的透明标记，然后返回转换后的结果实际上就是一串HTML命令。
如果HTML模板比较简单，只是一个字符串，您可以调用ContentFromString函数并传递HTML模板作为参数，也能得到转换后的结果。
如果HTML模板可以从流中读取，调用ContentFromStream函数比较方便，由于TBlobStream对象支持流入和流出等操作，您可以把HTML模板存储到数据库的备注字段中，以后再从备注字段中读取HTML模板。

16.7.4 怎样转换（OnHTMLTag事件）

调用Content、ContentFromStream或ContentFromString函数进行转换时，如果在HTML模板中遇到一个HTML标记，就会触发OnHTMLTag事件，这样您就有机会判断当前遇到的是哪个标记，然后再决定怎样替换这个标记。
下面我们详细介绍OnHTMLTag事件。

OnHTMLTag 事件

声明： _property THTMLTagEvent OnHTMLTag;
其中，THTMLTagEvent是这样声明的：

```
typedef void _fastcall (_closure *THTMLTagEvent)(TObject* Sender, TTag Tag, const System::AnsiString TagString, Classes::TStrings*TagParams, AnsiString &ReplaceText);
```

Tag参数告诉您当前遇到的是什么标记，如果Tag参数是tgLink、tgImage、tgTable、tgImageMap、tgObject、tgEmbed中的一个，表示是预定义的标记。如果Tag参数是tgCustom，表示是自定义的标记，此时TagString参数就是自定义的标记名。
不管是预定义的标记还是自定义的标记，如果标记附带了参数的话，您可以通过OnHTMLTag事件的TagParams参数得到标记中的参数，TagParams参数是TStrings对象，其中每个字符串的形式是ParamName=Value，其中，ParamName是参数名，Value是参数值。用TStrings对象的Names特性可以访问标记中的所有参数名，用TStrings对象的Values特性访问标记中的所有参数值。
在处理OnHTMLTag事件的句柄中至少要有这么一句，就是对ReplaceText参数赋值，实际上就是指定要替换标记的文本，该文本既可以是一个普通的字符串，也可以又是一个HTML透明标记，让下一个TPageProducer元件去转换，从而构成转换链。
下面我们通过一个典型的示例来帮助您理解HTML模板和OnHTMLTag事件。假设HTML模板是这样的：

```
<HTML>
```

```
<HEAD><TITLE>Our brand new web site</TITLE></HEAD>

<BODY>
Hello <#UserName>! Welcome to our web site.
</BODY>

</HTML>
```

可以看出，这个HTML模板中只有一个标记，标记名称是UserName。

```
void _fastcall TWebModule1::PageProducer1HTMLTag(TObject*Sender, TTag Tag, const AnsiString TagString, TStrings *TagParams, AnsiString &ReplaceText)
{
if (CompareText(TagString, "UserName") = = 0)
      ReplaceText = (TPageProducer *)Sender->Dispatcher->Request->Content;
};
```

上面这个事件句柄首先判断当前遇到的标记是否UserName，如是，就对ReplaceText参数赋值。这里顺便介绍TPageProducer元件的Dispatcher特性，TPageProducer元件一般应当放在Web模块上，Dispatcher特性返回TPageProducer元件所在的Web模块，由此可以访问HTTP请求消息，进而得到HTTP请求消息中的内容，即客户填写的UserName。

如果程序没有响应OnHTMLTag事件或者在处理OnHTMLTag事件的句柄中没有对ReplaceText参数赋值，HTML透明标记就用空字符串替换。

16.8 操纵Web服务器应用程序

C++ Builder 3用TApplication来操纵应用程序，并且在Forms单元中声明了TApplication的实例叫Application。对于Web服务器应用程序来说，情况就不一样了，C++ Builder 3用TWebApplication及其派生类来操纵Web服务器应用程序。当您用C++ Builder 3提供的向导创建Web服务器应用程序时，向导同时还创建了一个Web服务器应用程序对象，其实例也叫Application。对于CGI或Win-CGI类型的Web服务器应用程序来说，这个实例的类型是TCGIApplication，在CGIApp单元中声明。对于ISAPI或NSAPI类型的Web服务器应用程序来说，这个实例的类型是TISAPIApplication，在ISAPIApp单元中声明。

由此可见，Web服务器应用程序中的Application和普通应用程序中的Application不是一码事，至少它们的类型不同。因此，您不可以在Web服务器应用程序的项目文件中引用某个Form的单元名，因为这个单元会自动声明一个TApplication类型的Application。同样的道理，您不可以把ISAPIApp单元加到CGI或Win-CGI类型的Web服务器应用程序中，也不可以把CGIApp单元加到ISAPI或NSAPI类型的Web服务器应用程序中。

16.8.1 TWebApplication对象

TWebApplication用于操纵Web服务器应用程序的基本行为，包括创建TWebRequest对象和TWebResponse对象，并且把这两个对象传递给Web调度器，再由Web调度器传递给匹配的动作项，由动作项分析和处理HTTP请

求消息，最后发送HTTP响应消息给客户。
在HTTPAPP单元中，TWebApplication是这样声明的：

```
class PASCALIMPLEMENTATION TWebApplication : public Classes::TComponent
{
public:
    _fastcall virtual TWebApplication(Classes::TComponent*AOwner);
    _fastcall virtual ~TWebApplication(void);
    _property int ActiveCount;
    _property bool CacheConnections;
    _property int InactiveCount ;
    _property int MaxConnections;
    _property System::AnsiString Title;
};
```

ActiveCount特性

声明：_property int ActiveCount;
对于ISAPI或NSAPI类型的Web服务器应用程序来说，每收到一个HTTP请求消息，就会自动创建一个新的Web模块实例。如果HTTP请求消息得到处理，Web模块的这个实例要么被删除，要么变成非活动状态备下次再用。
因此，一个Web服务器应用程序中可能存在着Web模块的多个实例，其中有的处于活动状态，有的处于非活动状态，每个活动的实例均在一个单独的线程中。

ActiveCount是个只读的特性，返回当前处于活动状态的Web模块的实例数，实际上也就是Web服务器应用程序当前正在处理的HTTP请求消息数。InactiveCount也是个只读的特性，返回当前处于非活动状态的Web模块的实例数。MaxConnections特性用于设置最多可有多少个活动的Web模块实例，因此，ActiveCount特性的值不会大于MaxConnections。
注意：ActiveCount仅对ISAPI或NSAPI类型的Web服务器应用程序是有效的，因为对于CGI或Win-CGI类型的Web服务器应用程序来说，在同一个时刻只能响应一个HTTP请求消息，也就是说，Web模块只有一个实例。

CacheConnections 特性

声明：_property bool CacheConnections;
如果这个特性设为True，当一个HTTP请求消息被处理后，Web模块的实例不是被删掉，而是保留在内存中备下次再用。这样，当Web服务器应用程序收到下一个HTTP请求消息时，就不必重新创建Web模块的实例，从而提高了程序的性能。
如果这个特性设为False，当一个HTTP请求消息被处理后，Web模块的实例就被删掉，当Web服务器应用程序收到下一个HTTP请求消息时，就要重新创建Web模块的一个实例。这种方式的好处时，每个实例总是反映Web模块的最新状况。

InactiveCount 特性

声明：_property int InactiveCount;

在CacheConnections特性设为True的情况下，这个只读的特性返回当前处于非活动状态的Web模块的实例数。如果CacheConnections特性设为False，InactiveCount特性总是0。

MaxConnections特性

声明：_property int MaxConnections;
这个特性用于设置在同一个时刻最多允许Web模块的几个实例是活动的，换句话说，也就是Web服务器应用程序能同时处理几个HTTP请求消息。
MaxConnections特性的值应当设得比较合适，设得过小显然不行，如果设得过大的话，表面上看，Web服务器应用程序能同时处理很多HTTP请求消息，但同时执行很多线程也会导致应用程序的开销很大，弄不好反而会降低应用程序的性能。
如果当前活动的Web模块的实例数已经达到MaxConnections特性的值，此时若收到一个HTTP请求消息将触发异常。

Title特性

声明：_property System::AnsiString Title;
这个特性用于指定一个字符串，当Web服务器应用程序最小化时此字符串显示在它的图标下面。如果没有设置Title特性，就用Web服务器应用程序的文件名。

Initialize 函数

声明：virtual void _fastcall Initialize(void);
Initialize本来什么也不做，由于用C++ Builder 3创建的应用程序总是先调用Initialize，因此，如果您需要对Web服务器应用程序做一些初始化工作的话，您可以重载Initialize。

16.8.2 TCGIApplication对象

对于CGI或Win-CGI类型的Web服务器应用程序来说，操纵Web服务器应用程序的是TCGIApplication对象。在CGIAPP单元中，TCGIApplication是这样声明的：

```
class PASCALIMPLEMENTATION TCGIApplication : public Httpapp::TWebApplication
{
public:
    virtual void _fastcall Run(void);
    _fastcall virtual TCGIApplication(Classes::TComponent* AOwner) : Httpapp::
    TWebApplication(AOwner) { }
    _fastcall virtual ~TCGIApplication(void) { }
};
```

可以看出，TCGIApplication是从TWebApplication继承下来的，它没有自己的特性和方法，只是重载了TWebApplication中的Run。

16.8.3 TISAPIApplication对象

对于ISAPI或NSAPI类型的Web服务器应用程序来说，操纵Web服务器应用程序的是TISAPIApplication对象。在ISAPIAPP单元中，TISAPIApplication是这样声明的：

```
class PASCALIMPLEMENTATION TISAPIApplication : public
Httpapp::TWebApplication
{
public:
    BOOL _fastcall GetExtensionVersion(Isapi2::THSE_VERSION_INFO &Ver);
    int _fastcall HttpExtensionProc(TEXTENSION_CONTROL_BLOCK &ECB);
    BOOL _fastcall TerminateExtension(int dwFlags);
    _fastcall virtual TISAPIApplication(Classes::TComponent* AOwner) : Httpapp::
    TWebApplication(AOwner) { }
    _fastcall virtual ~TISAPIApplication(void) { }
};
```

可以看出，TISAPIApplication也是从TWebApplication继承下来的，增加了三个函数，这三个函数都是ISAPI或NSAPI类型的Web服务器应用程序必须输出的例程，由Web服务器调用。创建Web服务器应用程序时，项目文件中已自动输出了这三个函数。

16.9 Web服务器与数据库

用C++ Builder 3创建的Web服务器应用程序可以很方便地访问数据库，只要借助于TDataSetTableProducer元件和TQueryTableProducer元件，就可以从标准的数据集中引入数据并动态生成HTML页面。TQueryTableProducer元件还能从HTTP请求消息中检索SQL语句的参数值，从而实现满足特定需求的交互式查询和信息发布。

16.9.1 用TSession管理与数据库的连接

对于ISAPI或NSAPI类型的Web服务器应用程序来说，每一个HTTP请求消息都是在一个单独的线程中处理的。如果Web服务器应用程序连接了数据库的话，每个线程都有可能涉及到数据库连接、断开等问题。为了避免冲突，您应当把一个TSession元件加到Web模块上。每个线程会临时创建Web模块的一个新的实例，由于Web模块上有TSession元件，因此，每个线程就拥有了自己的Session对象。为了使这些Session对象的名称各异，您应当把TSession元件的AutoSessionName特性设为True。

对于CGI或Win-CGI类型的Web服务器应用程序来说，在同一个时刻只能处理一个HTTP请求消息，不存在多线程的问题。因此，如果Web服务器应用程序要连接数据库的话，只要用默认的Session对象就够了，不必把TSession元件加到Web模块上。

16.9.2 TDataSetTableProducer元件

TDataSetTableProducer元件的作用主要有两个，一个是从标准的数据集中引入数据，为此，您首先要把一个TTable元件或TQuery元件或TClientDataset元件加到Web模块上，然后设置TDataSetTableProducer元件的DataSet特性指定数据集。如果要访问的数据库是固定不变的，您可以在设计期设置DataSet特性。如果要访问的数据库是不固定的，甚至要由客户在HTTP请求消息中指定，此时就要在运行期设置DataSet特性。

TDataSetTableProducer元件与数据集之间无需借助于TDataSource元件，因为TDataSetTableProducer元件与一般的数据控件元件如TDBGrid在本质上是不同的，转换得到的HTML页面实际上是静止的文本，无法与数据集实现互动，例如，您无法用数据库导航器按钮来翻滚HTML页面上的记录。

TDataSetTableProducer元件的直接上级是TDSTableProducer，而TDSTableProducer又是从TCustomContentProducer继承下来的，因此，TDataSetTableProducer与前面介绍的TPageProducer具有相似性。下面就介绍TDataSetTableProducer元件的特性、方法和事件。

Caption特性

声明：_property System::AnsiString Caption;

TDataSetTableProducer元件能够把数据集转换为HTML表格，这个特性用于设置表格的头标。Caption特性本身是个字符串，引号内应当是HTML命令。程序示例如下：

```
Caption = "<H1>客户表</H1>";
```

CaptionAlignment 特性

声明：_property THTMLCaptionAlignment CaptionAlignment;
这个特性用于设置标题与HTML表格的位置关系，设为caDefault表示让Web浏览器决定，设为caTop表示标题显示在表格的上面，设为caBottom表示标题显示在表格的下面。

Columns 特性

声明：_property THTMLTableColumns* Columns;
这个特性有点类似于TDBGrid的Columns特性，用于设置表格中每一列的显示属性。在HTML表格中，每一列是一个THTMLTableColumn对象。
在设计期，您可以在Object Inspector中单击Columns特性边上的省略号按钮打开一个HTML表格属性编辑器，直接设置表格中每个列的属性，如图16.3所示。

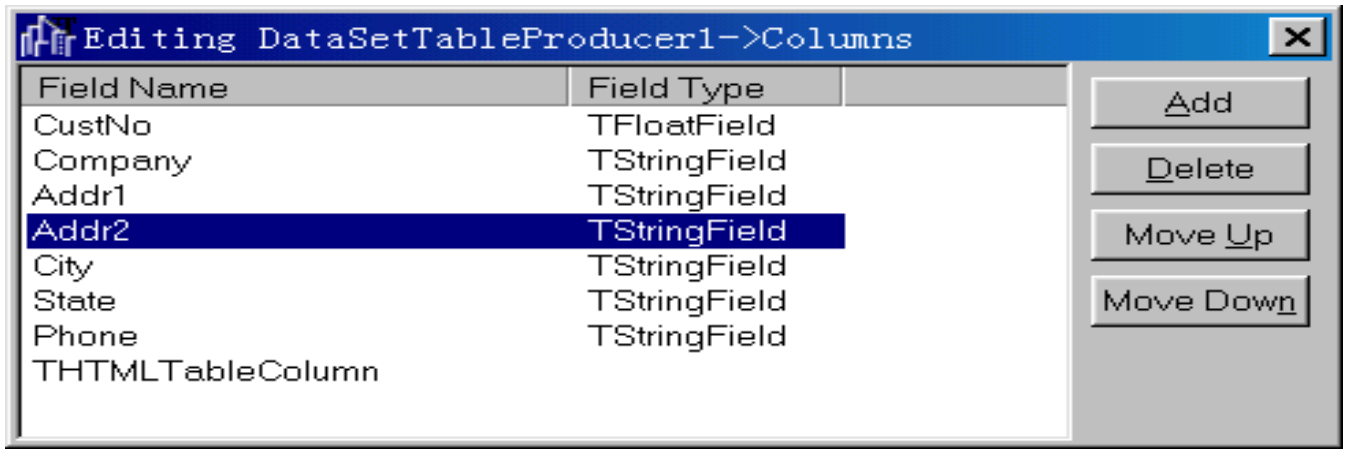


图 16.3　设置 HTML 表格中列的属性

刚开始的时候，HTML表格属性编辑器是空白的，这并不是意味着HTML表格是空白的，相反，C++ Builder 3会把据数据集中的所有字段都显示在HTML表格中。

如果不想显示所有的字段，或者想在设计期设置列的属性，您就得单击“Add”按钮创建一个永久的THTMLTableColumn对象，然后您就可以修改它的属性。

如果要调整字段在HTML表格中的顺序，单击“Move Up”或“Move Down”按钮。

DataSet 特性

声明：_property Db::TDataSet* DataSet;

这个特性用于指定要访问的数据集，可以是TTable、TQuery以及TClientDataSet。

Dispatcher特性

声明：_property TCustomWebDispatcher* Dispatcher;
通过这个特性可以得到TDataSetTableProducer元件所在的Web模块，进而可以访问HTTP请求消息和HTTP响应消息。

Header特性

声明：_property Classes::TStrings* Header;
如果要在整个HTML表格前加上一些内容，可以设置Header特性，这是一个TStrings对象，其中每个字符串必须是一条HTML命令。

Footer特性

声明：_property Classes::TStrings* Footer;
如果要在整个HTML表格的最后加上一些内容，可以设置Footer特性，这是个TStrings对象，每个字符串必须是一条HTML命令。
注意：C++ Builder 3并不检查Header特性和Footer特性中HTML命令的语法，这是由Web浏览器检查的。当您调用TDataSetTableProducer元件的Content函数时，所得到的页面包括Header特性和Footer特性中HTML命令描述的内容加上HTML表格本身。

MaxRows 特性

声明：_property int MaxRows;
这个特性用于设置HTML表格的最大行数，默认是20。如果数据集中的记录数超过MaxRows，超过的记录不会显示。

RowAttributes 特性

声明：_property Httpapp::THTMLTableRowAttributes* RowAttributes;
这个特性(THTMLTableRowAttributes对象)用于设置HTML表格中行的属性，TDataSetTableProducer元件会自动把这些属性转换为相应的HTML标记。

TableAttributes 特性

声明：_property Httpapp::THTMLTableAttributes* TableAttributes;
这个特性(THTMLTableAttributes对象)用于设置整个HTML表格的属性，TDataSetTableProducer元件会自动把这些属性转换为相应的HTML标记。

Content 函数

声明：virtual System::AnsiString _fastcall Content(void);
调用这个函数开始对数据集中的记录进行转换，此时将触发OnCreateContent事件。在处理OnCreateContent事件的句柄中，如果把Continue参数设为True，Content函数就返回Header特性描述的内容，然后是由数据集中的记录转换得到的HTML表格，最后是Footer特性描述的内容。Content函数的返回值可

以直接赋值给TWebResponse对象的Content特性作为HTTP响应消息中的内容。如果把Continue参数设为False，Content函数就返回空。

OnCreateContent事件

声明：_property TCreateContentEvent OnCreateContent;
其中，TCreateContentEvent是这样声明的：
typedef void _fastcall(_closure *TCreateContentEvent) (TObject* Sender, bool &Continue);
当调用Content函数时会触发这个事件。如果把Continue参数设为False，Content函数不再继续转换，并返回空字符串。

OnFormatCell事件

声明：_property THTMLFormatCellEvent OnFormatCell;
其中，THTMLFormatCellEventl是这样声明的：
typedef void _fastcall (_closure *THTMLFormatCellEvent) (System::TObject* Sender, int CellRow, int CellColumn, System::AnsiString &BgColor, Httpapp::THTMLAlign &Align, Httpapp::THTMLVAlign &VAlign, AnsiString &CustomAttrs, System::AnsiString &CellData);
在转换的函数中，每个单元都会触发OnFormatCell事件，这样您就有机会自己定义这个单元中的内容和外观。CellRow参数和CellColumn参数是单元所在的行号和列号。您可以设置BgColor、Align、VAlign、CustomAttrs等参数来改变单元的外观。CellData参数是单元中要显示的内容，默认是字段的值，

您可以修改。

OnGetTableCaption 事件

声明：_property THTMLGetTableCaptionEvent OnGetTableCaption;
其中，THTMLGetTableCaptionEvent是这样声明的：

```
typedef void _fastcall (_closure* THTMLGetTableCaptionEvent) (TObject*Sender, AnsiString &Caption,
THTMLCaptionAlignment &Alignment);
```

当将要生成HTML表格的标题时触发这个事件，您可以设置Caption参数和Alignment参数改变标题的排列方式。

16.9.3 TQueryTableProducer元件

与TDataSetTableProducer元件一样，TQueryTableProducer元件也是个HTML页面生成器，不同的是，TQueryTableProducer元件能够基于对数据库的查询结果生成HTML表格。
对数据库的查询实际上还是通过BDE的查询引擎进行的，因此，您必须把一个TQuery元件放到Web模块上，设置它的DatabaseName特性指定要访问的数据库，设置它的SQL特性指定要执行的SQL语句，如果您要构造很复杂的多表查询，建议您先用C++ Builder 3中的SQL Builder可视化地建立查询，然后把得到的SQL语句复制给TQuery元件的SQL特性，最后设置TQueryTableProducer元件的Query特性指定TQuery元件。
TQueryTableProducer元件能够自动从HTTP请求消息中检索URL所附带的参数，如果客户的请求类型是GET，通过TWebRequest对象的QueryFields特性

可以得到URL中附带的参数，如果客户的请求类型是POST，通过TWebRequest对象的ContentFields特性可以得到URL中附带的参数。如果URL附带的某个参数的名称与SQL语句中某个参数的名称匹配，TQueryTableProducer元件就把该参数的值赋给SQL语句中与之匹配的参数。

TQueryTableProducer元件的特性、方法和事件与TDataSetTableProducer元件几乎完全相同，为了节省篇幅，此处不再赘述。