Lewolf
保留版权
签名未验证

第四章 C++ Builder 语言的扩充

上一章讲述了面向状态的基本思想和方法，也针对 C++ Builder 语言给出了一些简单的例子，但是作为一种新的编程思想。必须有特定编译器的支持，面向状态也一样，尽管在很多开发工具中都已经支持这种方法，但各个编译器差别还是很大的。本书是针对 C++ Builder 语言而编写的，因此有必要对 C++ Builder 语言有一定的了解，至少应该对支持面向状态的语言扩充有一个较为深入的了解。

本书并不是 C++ Builder 的参考书或使用指南，只是针对 C++ Builder 来讲解面向状态的设计思想，因此本章对于 C++ Builder 中关于面向状态的语言扩充会进行详细的讲解，而其他内容不作详细说明。需要对 C++ Builder 进行深入了解的读者请参考相关书籍，已经非常熟悉 C++ Builder 的读者可以越过本章。

本章第三节针对 VCL 的基本类库的结构进行了一个简单的讲解，这将有助于读者更深入的了解 VCL 的处理机制，也会更加深入的理解这种新的技术。

1 关键字扩充

在 C++ Builder 中，有许多关键字是对 Ansi C++中关键字的扩展，其功能和作用和标准的 C++语言类似，也有一些关键字是 Ansi C++中所没有的，而且只在 C++Builder 中有效的，确切的讲是在 VCL 类库中才有效的。在 C++ Builder 5 中，大概有以下 36 个扩充的关键字：

__asm	_asm	asm	__automated	__cdecl	_cdecl
cdecl	__classid	__closure	__declspec	__except	__export
_export	__fastcall	_fastcall	__finally	__import	_import
__inline	__int16	__int32	__int64	__int8	__msfastcall
__msreturn	__pascal	_pascal	pascal	__property	__published
__stdcall	_stdcall	__thread	__try	unsigned __int64	__rtti

其中 asm、__rtti 等关键字在 Ansi C++中有相同的型式，作用也相同，但在 Ansi C++和 C++Builder 中对最终代码的影响会稍微有所不同；inline、try 等在 Ansi C++中有类似的型式，作用也相似；而__published、__property、__msfastcall 等关键字是 Ansi C++中所没有的，而且只在 C++ Builder 中有效，像__published、__property、__closure 等部分关键字是和本书所讲内容关系比较大的关键字，下面的内容将对它们作一些详细的介绍，其它关键字只作一些简单的描述，读者可以查阅相关书籍或者查看 C++ Builder 的联机帮助。

1.1 Ansi C++关键字的扩充

在 C++ Builder 中，有一些扩充的关键字是在 Ansi C++中本来就有相对应的关键字，它们的功能和作用基本相同，形状也相似，比如 asm、inline 等关键字，本书中只作简单描述，不作详细说明。

1.1.1 __asm、_asm、asm 关键字

__asm、_asm、asm 关键字是用来标记汇编代码的关键字，在 Ansi C++中也有相同的型式，用法和功能也相同，例如：

```
__asm  MOV  AX 12;
__asm{
        ADD   EAX EBX;
        MOV   Mem EAX;
     }
```

通常 C++ Builder 在编译的过程中，首先是将 C++代码编译成汇编代码，然后再启动汇编代码编译器将汇编代码编译成目标代码，这个过程是在内部执行的，而且也不会生成汇编代码的文件。但若在程序中使用__asm 关键字插入汇编代码，C++ Builder 在编译过程中会显式的分作两个过程来完成——首先启动 C++编译器将代码编译成汇编代码，并且将生成的汇编代码保存为扩展名是.asm 的同名汇编文件，然后启动汇编语言编译器将.asm 文件编译成目标代码。生成中间文件有助于程序员分析插入的汇编代码，也可以帮助程序员分析函数调用规则即调用过程，本书中多处例子编写时作者就是采用插入空汇编代码来验证调用过程的。

1.1.2 __rtti 关键字

__rtti 关键字是运行时类型判断 RTTI（Runtime type identification）选项的声明关键字，RTTI 在 C++ Builder 中是缺省有效的，可以在 Project 的 Option 中使之无效，也可以通过命令行的-RT-选项使之无效，在 RTTI 无效时，可以通过代码前缀__rtti 关键字使某个类具有 RTTI 特性。

RTTI 是类的多态性的一个选项，主要用于运行时使用 typeid 关键字获取对象的表示或类型。当 RTTI 选项有效时，一个类的指针或者引用的类型总是实际指向的对象的类型，而不一定是声明时的类型，而 RTTI 选项无效时，一个类的指针或者引用其类型可以是其声明时的类型，而不是实际指向的对象的类型。比如：

```
class A
{
   ……
```

```
};
class B : public A
{
   ……
};
A* pA;
B* pB;
B ClassB;
pA = &ClassB;
pB = &ClassB;
```

RTTI 有效时：
typeid（pA） 返回为类型 B，typeid（pB） 返回为类型 B；
RTTI 无效时：
typeid（pA） 返回为类型 A，typeid（pB） 返回为类型 B；

1.1.3 __inline 关键字

__inline 关键字在 C++ Builder 中扩充的关键字，在 Ansi C++中有相同作用的关键字 inline，两者的作用基本上是一样的，使用与定义内联函数，所不同的是，inline 关键字只能在 C++的函数中使用，而使用__inline 则可以定义 C 语言或者 C++语言的内联函数。

1.2 类声明列表关键字__declspec

__declspec 关键字是 C++ Builder 中提高修饰符灵活性的一个关键字，Ansi C++中没有相对应的关键字，通常__declspec 需要和其他修饰性关键字同时使用，其语法如下：

```
__declspec(<decl-modifier>);
```

其中 decl-modifier 可以为以下几种：

dllexport	**dllimport**	**naked**	**noreturn**	**nothrow**
novtable	**property**	**selectany**	**thread**	**uuid**

dllexport 是作为动态链接库输出函数的表示关键字，这为 C++ Builder 语言和 Microsoft C++以及标准的 Windows 程序提供了良好的兼容性，在 Windows 程序设计中，动态链接库的输出函数或数据需要在模块控制文件中定义，Microsoft 的 C++提供了__export 关键字，C++ Builder 中也包含了这个关键字，使用__export 将使不必使用模块控制文件来定义输出函数，在 C++ Builder 中也可以通过更加灵活方便的修饰符__declspec(dllexport)来定义输出函数。

dllimport 修饰符和 dllexport 类似，是用与动态链接库的输入函数和数据的关键字，__declspec(dllimport)和__import 关键字作用一样，但使用上更加方便。

naked 关键字用于定义函数，__declspec(naked)表明被修饰的函数不按照常规的方式建立和使用堆栈，而且调用函数以及函数返回时不保护任何寄存器的内容，这一点使所修饰的函数特性类似汇编语言，在后面的内容中，将要讲述 C++ Builder 中几种函数的调用规则，而 __declspec(naked)修饰正是这几种调用规则之外的类型——实际上是一种没有规则的规则，需要程序员清楚的知道并实现函数调用时参数的传递以及返回值的传递过程，并且要程序员要负责编写代码实现调用前和调用后对寄存器的保护和恢复。因而__declspec(naked)关键字通常都和内嵌的汇编语言结合在一起使用，而且多用于设备驱动等底层的开发。

需要注意的是__declspec(naked)修饰符不能和任意一种函数修饰符如：__fastcall、__stdcall 等合用，只能单独使用。

noreturn 关键字用来通知编译器，该函数调用是不返回的。通常编译器在会变过程中会分析代码，对某些不能运行到的代码会发出警告或报错，如果这种情况是由于调用的函数没有返回而造成的，则可以通过声明__declspec(noreturn)函数来避免编译器报错。

nothrow 关键字用于修饰函数，告诉编译器该函数不会产生例外，使编译器不为其生成同步的例外跟踪处理代码，从而有效的降低代码长度，提高运行效率。下面三个函数的效果是一样的：

```
#define WINAPI __declspec(nothrow) __stdcall
void WINAPI foo1();
void __declspec(nothrow) __stdcall foo2();
void __stdcall foo3() throw();
```

其中 foo3 是标准 C++的声明语法，foo2 是 C++ Builder 的扩充声明语法，foo1 是采用宏定义的 C++ Builder 声明。当处理多个相同类型的函数时，采用宏定义要高效得多。

novtable 关键字用于类的定义声明，__declspec(novtable)只能用于纯粹的接口类，这些类永远不会自身实例化，例如某些纯虚基类就是这样的，当定义成__declspec(novtable)类的时侯，编译器不为其产生虚拟函数分配表（vtable），编译器连接时会去处所有关于该类的虚拟函数连接，从而有效的降低代码长度。

property 关键字用于定义“虚拟数据成员”，和后面要介绍__property 关键字类似，但功能似乎要比__property 弱得多，而且实际中好像也没有多大意义，因为 Borland 也好像只在帮助文件中出现过这样的用法。其语法如下：

```
__declspec(property(get=get_func_name)) declarator
__declspec(property(put=put_func_name)) declarator
__declspec(property(get=get_func_name,put=put_func_name)) declarator
```

同__property 关键字一样，__declspec(property())也只能用于类的成员定义中，比如可以定义一个虚拟数组成员 X：

```
__declspec(property(get=GetX, put=PutX)) int x[];
i = p->x[a][b]
```

实际上就是：

```
i=p->GetX(a, b),

p->x[a][b] = i
```

实际上就是：

```
p->PutX(a, b, i);
```

selectany 用于头文件中对变量声明且初始化的情况，通常可执行文件或者动态链接库对全局变量只能进行一次初始化，当一个变量在头文件中初始化并被不同的源 cpp 文件引用时就会产生错误，selectany 就是告诉编译器初始化时“任选其一即可”例如在头文件中：

```
__declspec(selectany) int x1 = 1;
```

是正确的，x1 被声明并正确初始化。

```
const __declspec(selectany) int x2 = 2;
```

在 C++语言中这是不正确的，因为 C++中 const 缺省的代表着静态变量，不具备 extern 特性，因此不能正确初始化，但是在 C 语言中，这是正确的，因为 const 在 C 语言中缺省的不是静态变量。

```
extern const __declspec(selectany) int x3=3;
```

是正确的，const 被重载具有 extern 特性。

```
extern const int x4;
const __declspec(selectany) int x4=4;
```

正确，和上面的情况相同。

```
extern __declspec(selectany) int x5;
```

是不正确的，因为 x5 没有初始化，__declspec(selectany)只能用于变量的初始化过程。

thread 用于“进呈局部变量”的定义，__declspec(thread)和__thread 关键字的作用一样，关于“进呈局部变量”参见__thread 关键字的介绍。

uuid 用于将一个完整的 COM 对象标记为一个类或者结构类型，主要用于分布式编程，uuid 的参数可以是注册对象的 GUID 字符串，也可以是带有 {} 分界符的字符串，例如：

```
struct __declspec(uuid("00000000-0000-0000-c000-000000000046")) IUnknown;
struct __declspec(uuid("{00020400-0000-0000-c000-000000000046}"))
                                                        IDispatch;
```

可以使用__uuidof 关键字来获取和__declspec(uuid(..))修饰符一样的功能，但

在 C++ Builder 中__declspec 关键字可以带来很多方便。

通常，C++语言中的修饰符必须处于被修饰对象的前面，并且相邻的位置，使用__declspec 关键字可以消除这种修饰符位置的强约束性，比如输出一个函数：

使用__export 关键字只能采用以下形式：

```
void __export f(void);
```

而不能是：

```
__export void f(void);
```

型式，但是采用__declspec 关键字，则下面几种声明语句都是正确的：

```
void __declspec(dllexport) f(void);
__declspec(dllexport)void f(void);
class __declspec(dllexport) ClassName { };
```

1.3 C++ Builder 扩充变量类型

在 C++ Builder 中允许程序员指定整型变量的长度，以二进制位数来表示，但仅限于以下几种类型

__int8	八位整型。
__int16	十六位整型。
__int32	三十二位整型。
__int64	六十四位整型。
unsigned __int64	六十四位无符号整型。

而且，在使用扩展整型变量时，必须对数值加上适当的后缀，__int8 、__int16、__int32、__int64、unsigned__int64 的后缀分别为 i8、i16、i32、i64、ui64，例如：

```
__int8 c = 127i8;
__int32 i = 123456789i32;
__int64 big = 12345654321i64;
```

1.4 函数调用规则与返回

所谓调用规则是指函数采用什么样的方式来传递参数和返回值的，在高级语言中有两大类比较常用的调用规则，分别是 C 调用规则和 Pascal 调用规则，许多编译器为了提高程序性能，在这两种规则的基础上进行了一些扩充。

C++ Builder 中支持五种不同的调用规则，它们分别是：C 调用规则、Pascal 调用规则、Win32 标准调用规则、寄存器调用规则和 Microsoft 的寄存器调用规则。除了这五种调用规则之外。C++ Builder 还允许使用“自定义”类型的调用规则，就是前面所讲到的__declspec(naked)关键字修饰符。

下面就这五种调用规则逐一介绍。

1.4.1 __cdecl、_cdecl、cdecl 关键字

这是在 C++ Builder 中特有的关键字，__cdecl、_cdecl、cdecl 可以用来修饰变量或者函数，其含义是指定的变量或者函数使用 C 语言的调用规则和命名规则。在 C++ Builder 中 C 调用规则是缺省的设置,可以通过 Project 的 Option 菜单在 Advanced Compiler 标签中来改变缺省设置。

对于 C 命名规则的变量或函数，具有大小些敏感、编译后变量名称前下划线前导符，这与 Pascal 语言是不一样的。对于函数，C 调用规则要求使用堆栈传递参数，参数传递的顺序为自右向左依次压入堆栈，由调用者自行清除堆栈，C 调用规则允许向函数传递不定参数，但这种调用方式系统需要更多的开销来完成参数的匹配，除非特殊用途一般尽量不要使用，这种调用方式最典型的例子是控制台程序中的 main 函数，程序运行时可以不带参数，也可以带多个参数。

在一个独立的程序中，采用那种调用规则或者命名规则影响并不十分大，但是在多种语言混合编程，或者编写带有数据输出或函数输出的可执行模块（可执行文件和动态链接库）时，命名规则和调用规则显得尤为重要，使用不当则可能导致程序无法正常运行，甚至使系统崩溃。

__cdecl、_cdecl、cdecl 的使用语法如下：

```
cdecl <data/function definition>;
_cdecl <data/function definition>;
__cdecl <data/function definition>;
```

例如：

```
int cdecl I;          //编译后标识符为“_I”
void __cdecl Fun(); //编译后函数名为“_Fun”
```

1.4.2 __fastcall、_fastcall 关键字

__fastcall 和_fastcall 也是 C++ Builder 中特有的关键字，只能用于修饰函数，其作用是指定函数使用“寄存器”调用规则，使用语法如下：

```
return-type _fastcall function-name(parm-list)
return-type __fastcall function-name(parm-list)
```

例如:

```
void __fastcall GetName(int Index);
```

使用“寄存器”调用规则的函数，其参数传递顺序为自左向右依次传递，并且将前三个参数尽可能的使用 EAX、EBX、ECX 这三个寄存器传递，对于前三个参数中浮点类型、

结构类型等超过四个字节的变量，和第四个及以后的参数则采用堆栈来传递，因此采用“寄存器”调用规则的函数只能传递固定数量的参数。

在 C++ Builder 中，所有属于 VCL 的成员函数，必须是__fastcall 类型，编译器将“寄存器”调用规则和 C 调用规则、Pascal 调用规则以及 Win32 的标准等其它调用规则是同等对待的，因此__fastcall 关键字不能和__cdecl、__pascal、__stdcall 等关键字联合使用，也不能和__export 关键字同时使用（实际上使用__fastcall 规则的函数是允许输出的，只是这种调用规则编写的 dll 或可执行模块只能被 C++ Builder 或者 Delphi 开发的程序载入并调用输出的函数）。

被指定为“寄存器”调用的函数会在编译时被冠以前导符@，这将使得 C 及 C++的函数命名过于混乱，但是这种调用规则有很高的执行效率，因此在 C++ Builder 中几乎所有 VCL 的成员函数都使用这种调用规则。

但是__fastcall 和_fastcall 是 C++ Builder 之前的 C++编译器所没有的调用规则，为了保证兼容性，可以使程序连接以前版本的编译器生成的库文件，C++Builder 提供了-VC 命令行选项，缺省状态下，该选项是关闭的，当需要时，可以通过命令行编译并选择-VC 选项。

1.4.3 __msfastcall、__msreturn 关键字

__msfastcall 和__msreturn 也是 C++ Builder 中特有的关键字，使用这两个关键字可以为程序提供和 Microsoft 兼容的“寄存器调用”规则以及和 Microsoft 兼容的函数返回的规则。

被__msfastcall 修饰的函数，其参数传递顺序为前两个参数如果长度小于 4 字节，则分别使用 ECX 和 EDX 寄存器传递，其余的参数和前两个不能使用寄存器传递的参数则按照从右向左的顺序通过堆栈来传递，堆栈由被调用函数负责清除。

__msreturn 是使用 Microsoft 兼容的“寄存器调用”返回规则，在这种返回规则中，返回值的长度在 4～8 个字节时，将采用 EAX/EDX 寄存器返回函数值。

1.4.4 __pascal、_pascal、pascal 关键字

__pascal、_pascal、pascal 关键字表示一个变量或者函数遵循 Pascal 的调用和命名规则。

在 Pascal 的命名规则中，对大小写并不敏感，而且不象 C 调用规则，会在变量和函数明成前面加前导符。Pascal 调用规则中，参数的传递是按照从左到右的顺序依次压入堆栈，并且由被调用函数自行清除堆栈。

Pascal 调用规则破坏了 C 语言的本身的调用体系，同时也带来了更多的灵活性，被表示为__pascal 调用规则的函数，编译后的目标文件中，名称会被全部转换为大写字母，在 Pascal 语言中，过程和函数的书写是对大小些不敏感的，但是在 C++ Builder 中采用 Pascal 调用规则的函数，在 C++源文件中仍然需要遵循 C 语言中对大小些必须严格一致的要求，Pascal 调用规则的大小写不敏感性只能体现在动态链接库、或者输出、输入函数以及数据的时侯。也正是这样的原因，对一些采用 ObjectPascal 编写的第三方控

件或者书写不是很规范的 Pascal 程序代码，使用 C++ Builder 编译的时候，会出现找不到函数原型的错误，实际上可能只是程序中函数名称的大小些不一致造成的。

1.4.5 __stdcall、_stdcall 关键字

在 Windows 程序设计中，有一种 Win32 的标准调用规则，基本上所有的 Windows API 函数都采用了这种 Win32 的标准调用规则，在讲__declspec(nothrow)关键字时的例子，其中宏定义就是 C++ Builder 中 winuser.h 宏定义。Win32 标准调用在 C++ Builder 中以__stdcall、_stdcall 关键字来标识。

在__stdcall、_stdcall 调用规则中，函数的参数是按照从右向左的顺序依次压入堆栈，所有的参数均采用堆栈传递，而且是被调用函数负责清除堆栈。在参数的传递顺序上，Win32 标准调用规则和 C 调用规则一样，从右向左，但和 C 调用规则不同的是，Win32 的标准调用要求调用者必须严格按照被调用函数的参数数量及类型进行参数传递。

关于 C++ Builder 中函数调用规则以及命名规则的详细内容，读者参考相关书籍，这里不做过多讲解，下表给出了 C++ Builder 中几种调用规则的一个简单比较，具有汇编语言基础的读者可以参考本书光盘中相关代码自行分析区别。

关键字	调用规则	参数传递方向	返回	参数寄存器	堆栈的清除
__cdecl	C 调用规则	从右向左	EAX	无	调用者
__fastcall	寄存器	从左向右	EAX	EAX、EBX、ECX	被调用者
__stdcall	Win32 标准	从右向左	EAX	无	被调用者
__pascal	Pascal	从左向右	EAX	无	被调用者
__msfastcall	Ms 寄存器	从右向左	EAX/EDX	ECX、EDX	被调用者

表 4.1 C++ Builder 中几种调用规则的比较

1.5 函数和数据的输入输出

在 Windows 程序中，任何一个可执行模块的函数和数据都可以输入和输出，但需要在模块控制文件（.def 文件）中定义。C++ Builder 提供了_export、__export、__import、_import 等关键字，可以使程序员不必使用.def 文件来定义数据和函数的输入输出。它们的作用分别和__declspec(dllexport)、__declspec(dllimport) 相当。

被定义成__export 的函数，编译器连接时会将列入这个可执行模块的输出列表（export table），这将增加了代码的长度并降低了执行效率，但同时为编程提供了方便，如果不采用 export 关键字，动态链接库会按照函数的顺序进行输出，这样在采用 Loadlibrary 动态加载可执行模块时，就不能使用函数名获取函数的入口，而必须采用函数的输出序号，关于函数输出和输入的详细内容请参考 Windows SDK。

1.6 __try、__except、__finally 关键字

例外处理机制是 C++语言引入的一种程序的保护措施，可以有效的防止意外的执行错误，比如内存溢出，“0”除数等，从而保护系统不会因为程序出错而崩溃。try、throw 等关键字就是为例外执行处理而引入的，在 C++ Builder 中 Borland 公司提供了比 Ansi C++更为强大的处理机制，下面就 C++ Builder 中例外处理的关键字作以简单介绍，关于 C++异常处理的详细内容我们在后面另行介绍。

在 Ansi C++中关于异常处理的关键字在 C 语言中是不受支持的，因而在 C 与 C++语言的混合编程中，对异常的处理是比较困难的，而 C++ Builder 提供了几个新的关键字，来支持 C 语言的异常处理，这就是__try、__except、__finally，__try 关键字在 C++语言中和 try 的作用是相同的，其区别是__try 关键字支持 C 语言中的应用，__except、__finally 是扩充的关键字，可以用于 C 和 C++的异常处理。

异常处理，通常是由一个 try 关键字和其后的 catch 关键字组成，try 后面的语句或语句组合是可能出现异常的代码，catch 关键字用于捕捉异常，__except 是用于 C 程序产生的异常处理，而__finally 关键字后面的语句是不管 try 语句中的执行状况如何，都需要执行的动作。下面的例子说明了如何使用这几个关键字：

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
class Exception
{    //定义一个异常类
public:
    Exception(char* s = "Unknown"){what = strdup(s); }
    Exception(const Exception& e ){what = strdup(e.what);}
   ~Exception(){free(what); }
    char* msg()const{return what; }
private:
    char* what;
};

int main()
{
    float e, f, g;
    try
    {
      try
      {
        f = 1.0;
```

```
            g = 0.0;
            try
            {
               puts("Another exception:");
               e = f / g;    //制造一个异常；
            }
            __except(EXCEPTION_EXECUTE_HANDLER)
            {   //处理 C 类型的异常
               puts("Caught a C-based exception.");
               throw(Exception("Hardware error: Divide by 0"));
            } //抛出例外，被 catch 捕获；
         }
         catch(const Exception& e)
         {
             printf("Caught C++ Exception[ %s ]\n", e.msg());
         }
      }
      __finally
      {
          puts("C++ allows __finally too!");
      }
      return e;
   }
```

上面的程序运行后的结构如下：

```
Another exception:
Caught a C-based exception.
Caught C++ exception[Hardware error: Divide by 0]
C++ allows __finally too!
```

在这个例子中先定义了一个异常的类 Exception，然后在 main 函数中使用了两个 try 语句组合，其中一个是基于 C 语言的异常处理，另一个是基于 C++的异常处理，结果是这两个异常处理过程均被触发。

前面也提到过 VCL 类库不建议使用太多的条件判断来完成一个操作，而更推荐使用异常处理机制来确保程序的正确执行和出错时的条件复原。异常处理机制也是面向状态的程序设计中必不可少的，后面将专门就 C++ Builder 的异常处理进行讲解。

1.7 VCL 类库关键字扩充

VCL 是 C++ Builder 基本类库，其中不少内容是和以前的 OWL 以及 MFC 大不相同的，面向状态也正是建立在 VCL 基础上的，因此也有一些语法以及关键字是 Ansi C++

不具备的，这正是面向状态最需要的关键字，下面就 VCL 中使用的特殊关键字作以描述。

1.7.1 __classid 关键字

__classid 关键字实际上是一个操作符，编译器使用这个关键字以生成指向特定的 classname 列表的指针。

也就是说这个操作符被用来从一个类获得元类，其语法是：

__**classid**(classname)

比如在每一个 C++ Builder 的程序中，WinMain 函数中都可能有这样一条语句：

```
Application->CreateForm(__classid(TForm1), &Form1);
```

用来自动创建一个窗体 TForm1 的实例 Form1，CreateForm 的原型为：

```
void __fastcall CreateForm(System::TMetaClass* InstanceClass,
        void *Reference);
```

CreateForm 函数用于运行时动态创建一个窗体，元类 TMetaClass 是 Object Pascal 中类引用在 C++ Builder 中的体现，因此__classid 关键字实际上是为支持 Object Pascal 语言而引入的。通常这个关键字程序员不必直接使用，编译器往往会自动在必要的时侯生成相应代码。

比如组件开发时，__classid 用于注册属性编辑器，组件，和类，并可用于 TObject 的 InheritsFrom 方法。下列代码说明创建从 TWinControl 派生的一个新组件时使用 __classid：

```
namespace MyWndctrl
{
   void __fastcall PACKAGE Register()
   {
     TComponentClass classes[1] = {__classid(MyWndCtrl)};
     RegisterComponents("Additional", classes, 0);
   }
}
```

在这个例子中，RegisterComponents 要使用一个 TComponentClass 类型的数组作为参数，来决定要将那些组件安装在相应的组件面板，TComponentClass 是指向 TComponent 及其派生类的元类指针，它的定义如下：

```
typedef TMetaClass* TComponentClass;
```

这和上一个例子中的作用实际上是一样的。

1.7.2 __closure 关键字

__closure 关键字被用来声明一个特殊类型的指针指向成员函数。不同于一般的 C++ 成员函数指针，__closure 是包含了对象指针的。

在 Ansi C++中，可把一个派生类的实例分配给基类指针；但是，不能把一个派生类的成员函数分配给基类成员函数指针。

下列代码说明这种情况：

```
class base
{
public:
    void func(int x);
};
class derived: public base
{
public:
    void new_func(int i);
};
void (base::*bptr)(int);
bptr = &derived::new_func;   // 这是非法的
```

然而，__closure 语言扩展允许在 C++ Builder 中使用这种用法。__closure 将指向成员函数的指针与指向类实例的指针联系起来。当调用相联的成员函数时，指向类实例的指针被作为 this 指针使用。__closure 指针的声明与普通函数指针的声明一样，只是在被定义的标识符前增加__closure 关键字就可以。例如：

```
struct MyObject
{
    double MemFunc(int);
};
double func1(MyObject *obj)
{
    //定义一个closure属性的函数指针，返回Double，参数为整型。
    double ( __closure *myClosure )(int);
    //初始化指针
    myClosure = obj -> MemFunc;
    //用这个闭合指针调用函数
    return myClosure(1);
```

```
}
```

__closure 在 C++Builder 中总是与事件一起使用。前面我们也讲过了，事件是 VCL 中消息处理的基础，也是描述状态的一个必不可少的属性，我们来看对一个事件的初始化过程。通常设计时指定的事件处理句柄都被 C++ Builder 保存在资源文件中（C++ Builder 的资源文件为 .dfm），程序启动时自动的初始化，然而事件也可以在运行时初始化，甚至是在运行时动态的改变，前面的例子中我们已经有这样的应用，这是这种灵活的处理方式，才使面向状态更加完整。我们看一下代码：

Form1 的头文件：

```
#include <Buttons.hpp>
//---------------------------------------------------------
class TForm1 : public TForm
{
__published:
private:
public:
    void __fastcall MyOnMessageHandle(tagMSG &Msg, bool
          &Handled);
    __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------
#endif
```

Form1 的单元文件：

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

```
}
//---------------------------------------------------------
void __fastcall TForm1::MyOnMessageHandle(tagMSG &Msg, bool &Handled)
{
   //...   消息处理代码；
}
```

在这个例子中 Application->OnMessage 的原型如下：

```
typedef void __fastcall (__closure *TMessageEvent)(tagMSG &Msg,
     bool &Handled);
__property TMessageEvent OnMessage = {read=FOnMessage,
                                      write=FOnMessage};
```

TMessageEvent 是一个 __closure 的函数指针类型，OnMessage 是 Application 对象的成员，在 Form1 中定义了一个与之相符合的消息处理函数：MyOnMessageHandle，Form1 在创建的时侯对 Application->OnMessage 进行初始化。从这里可看出，实际上 Form1 和 Application 是两个不相干的对象，但是 Application 的函数指针成员却指向了 Form1 的成员函数，这在 Ansi C++中是绝对不允许的，但是在 C++ Builder 中这样却是正确的。由此可见__closure 指针的出现为对象 Application 提供了更加有好和灵活的对外接口,也是更加符和客观规律的接口，通过事件这一特殊属性的接口，才使面向状态成为可能。

1.7.3 __property 关键字

在上一章中，我们已经接触过__property 关键字，它主要应用与创建和声明属性，前面也讲过属性只能在类中被声明。

对于属性数组（当<prop dimlist>被使用时），数组的索引可以是任何类型。

其语法：

```
<property declaration> ::=
   __property<type> <id> [ <prop dim list> ] = "{" <prop attrib list> "}"
```

其中<prop dim list>是可选的，只有声明一个数组属性的时侯才需要，成为索引参数，可以为任何类型的变量，它的格式如下：

```
<prop dim list> ::= "[" <type> [ <id> ] "]" [ <prop dim list> ]
```

索引列表可以由一个或者多个索引参数组成。

属性声明中<prop attrib list>为特征列表，其格式如下：

```
<prop attrib list> ::= <prop attrib> [ , <prop attrib list> ]
```

其中<prop attrib>有以下六中特征：

```
read = <data/function id>
write = <data/function id>
stored = <data/function id/bool constant>
default = <constant>
nodefault
index = <const int expression>
```

这六个特征各有用途，其中 read 和 write 可以使用在 C++ Builder 中任何的类中，其他几个只能使用在 VCL 风格的类中，而且通常都是在编写组件时使用的。

read 部分指定了该属性的读取操作，可以是一个和属性类型相同数据变量，也可以是一个函数句柄，称作读取函数，该函数必须有着和该属性类型相同的返回值，属性为数组时，读取函数至少包含一个索引参数，其参数的类型和数量与<prop dim list>中的类型和数量一一对应。

write 部分指定了属性的存取操作，可以是一个和属性类型相同的数据变量，也可以是一个函数句柄，称作存储函数或者写函数，该函数一般无返回值，但是必须带有一个和属性类型一致的参数，如果属性是数组的时侯，存储函数的参数中应该包含一个和属性类型一致的参数，并且是第一个参数，后面的参变数应该和<prop dim list>中的类型和数量一一对应。

stored 用于 VCL 组件中，表明这个属性在设计时所设置的取值需要被保存在资源文件中,可以为一个 bool 类型的变量也可以为一个具有 bool 类型返回值的函数,关于 VCL 组件的存储机制以及 C++ Builder 对资源的使用在后面的章节介绍，这里仅仅对关键字的使用作一了解。

default 用于指定一个缺省的取值，应该是和属性类型相同的一个常量。该缺省值是供编译器使用的，当设计程序时，在 Object Inspector 中为组件属性设置的取值和缺省值相等的时侯，编译器将不管 stored 部分为真还是为假，都不会将属性取值存储在资源文件中。default 部分并不能为属性进行初始化，也就是说，定义了 default 部分的属性，必须在构造函数中对该属性进行正确初始化。

nodefault 表示属性没有缺省值，不能和 default 部分同时出现。指定了一个属性没有缺省值，编译器会将为该属性设置的任何取值都存储在资源文件中。

index 部分定义了一个索引值，必须为一个整型的常量。索引值为多个属性共用相同的存储函数和读取函数提供了可能，当一个属性定义了索引的时侯，其读取函数和存储函数必须将索引作为第一个参数，后面的参数和不带有索引项的属性相同。

通常一个属性的特征列表中，至少应该包含一个 read 部分，这样该属性可以被其他对象当做变量一样访问，只包含 read 部分的属性是只读的，不能写入，可读可写的属性

应该至少包含 read 和 write 两个部分。下面这一段代码显示了如何使用__property 关键字。

```
class PropertyDemo : public TObject
{
private:
    int Fi0;
    int Fi1;
    int Fi2;
    int Fi3;
    int Fi4;
    int Fi5;
protected:
    void __fastcall SetFi2(int Value){ /*...*/ };
    int  __fastcall GetFi3(){ return Fi3; };
    void __fastcall SetFi3(int Value){ /*...*/ };
    int  __fastcall GetFi4(){ return Fi4; };
    void __fastcall SetFi4(int Value){ /*...*/ };
    int  __fastcall GetFi5(){ return Fi5; };
    void __fastcall SetFi5(int Value){ /*...*/ };
public:
    __property int PropI0 = {read=Fi};
    __property int PropI1 = {read=Fi,write=Fi,default=1};
    __property int PropI2 = {read=Fi,write=SetFi};
    __property int PropI3 = {read=GetFi3,write=SetFi3};
    __property int PropI4 = {read=GetFi4,write=SetFi4,
                             default=4};
    __property int PropI5 = {read=GetFi5,write=SetFi5,
                             default=5,stored=true};
    __fastcall PropertyDemo();
};

__fastcall PropertyDemo:: PropertyDemo ()
           : TObject ()
{
    /*其他初始化内容*/
    Fi1 = 1;     //对属性 PropI1 的 default 取值初始化
    Fi4 = 4;     //对属性 PropI4 的 default 取值初始化
    Fi5 = 5;     //对属性 PropI5 的 default 取值初始化
```

```
}
```

在 VCL 风格的类中（从 TObject 派生而来）上面这几个属性的声明都是正确的，其中：

PropI0 是一个只读属性，其 read 部分直接和一个数据变量 Fi0 对应；

PropI1 是一个可读可写的属性，其 read 和 write 部分都直接和数据变量 Fi1 对应，该属性的缺省取值为 1，在构造函数中必须对属性进行初始化 Fi1 = 1；

PropI2 是一个可读可写的属性，其 read 部分直接和数据变量对应，而 write 部分和一个存储函数 SetFi2 对应，该函数带有一个参数，其类型 PropI2 相同。

PropI3 是一个可读可写的属性，其 read 和 write 部分分别和两个函数对应，read 部分的 GetFi3 是一个返回值和 PropI3 相同的无参数函数，SetFi3 同 PropI2。

PropI4 是一个可读可写的属性，其 read 和 write 部分分别和两个函数对应，read 部分的 GetFi4 是一个返回值和 PropI4 相同的无参数函数，SetFi4 同 PropI2。该属性的缺省取值为 4，在构造函数中必须对属性进行初始化 Fi4 = 4；

PropI5 是一个可读可写的属性，前 read、write 和 default 部分和 PropI4 类似，在 stored 部分定义为 true，代码上和 PropI4 基本一样，但是如果设计时使用该组件，其生成的可执行文件的资源是不一样的。

下面的代码显示了如何使用<prop dim list>参数列表来构建一个 AnsiString 的数组属性。这个属性使用时表现形式和一个数组比较接近。

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

AnsiString __fastcall TDemo::GetNumberSize(int Index)
{
  AnsiString Result;
  switch (Index)
  {
    case 0:
      Result = "Zero";
      break;
    case 100:
```

```
        Result = "Medium";
        break;
      case 1000:
        Result = "Large";
        break;
      default: Result = "Unknown size";
    }
    return Result;
}
```

在这个例子中TDemo只有一个属性NumberSize,类型为一个AnsiString的数组，并且是一个只读属性。数组属性的 read 部分只能是读取函数，而不能和一个数组直接对应。

下面的代码显示了如何使用 Index 部分，使多各属性使用同一个读取和存储函数：

```
class TSampleCalendar : public TCustomGrid
{
private:
   TDateTime FDate;
    //……其它成员，用于显示日历
   int   __fastcall GetDateElement(int Index); //索引作为参数
   Void __fastcall SetDateElement(int Index, int Value);
                              //索引作为第一个参数
public:
   __property int Day = {read=GetDateElement,
                    write=SetDateElement, index=3, nodefault};
   __property int Month = {read=GetDateElement,
                    write=SetDateElement, index=2, nodefault};
   __property int Year = {read=GetDateElement,
                    write=SetDateElement, index=1, nodefault};
    //……其它成员函数，用于显示日历
};
```

由于日期的每一个元素（年、月、日）都是一个整数类型，而且每一个元素存储时都要进行编码变换，这种变换具有一定的共性，因此采用带有索引的属性可以避免重复的运算代码，读取函数 GetDateElement 的代码如下：

```
int __fastcall TSampleCalendar::GetDateElement(int Index)
{
```

```
    unsigned short AYear, AMonth, ADay;
    int result;
    FDate.DecodeDate(&AYear, &AMonth, &Aday); //
    switch (Index)
    {
      case 1:
         result = AYear;
         break;
      case 2:
         result = AMonth;
         break;
      case 3:
         result = ADay;
         break;
      default:
         result = -1;
    }
    return result;
}
```

存储函数 SetDateElement **的代码如下：**

```
void __fastcall TSampleCalendar::SetDateElement(int Index,
                                        int Value)
{
    unsigned short AYear, AMonth, ADay;
    if (Value > 0)   //有效的元素必须是大于“0”的
    {
      FDate.DecodeDate(&AYear, &AMonth, &ADay); //获取原始元素
      switch (Index)
      {
        case 1:
           AYear = Value;
           break;
        case 2:
           AMonth = Value;
           break;
        case 3:
           ADay = Value;
```

```
            break;
         default:
            return;     //无效的索引项，则不需要设置新的日期
       }
     }
     FDate = TDateTime(AYear, AMonth, ADay); //对新的元素编码
     Refresh();        //刷新图形界面
}
```

这个例子是一个日历控件 TSampleCalendar 的部分代码，控件中使用了年、月、日三个元素将日期类型的变量分解开来表示，并且采用了有效的索引来共享日期编码和转换过程。

从前面的几个例子，读者就__property 关键字的使用应该由一个大致的了解。在上面的两个例子中都使用了 Index 作为参数，但是两者有根本的不同，第一个例子中的 Index 相当于“ 数组 ”属性的数组下表,可以为任何类型的变量 ;而第二个例子中的 Index 是属性声明部分的 index 定义，使用 index 部分可以使多个属性使用相同的读写函数，该变量只能为整型，这一点读者一定要区分清楚。关于属性的更进一步内容，我们放在有关组件设计内容里面和实际例子结合起来作更深入的了解。通常可以通过属性的一些特征将它们与数据成员区分开来，一般来说属性具有以下特征：

- 可以通过一个标识符（属性）连接到特定的数据或者读、写方法，程序中并不存在为属性分配的存储空间；
- 可以为属性设置缺省值（仅在组件设计中有效）；
- 可以存储在一个窗体文件中（仅在组件设计中有效）；
- 可以扩展一个在基类中被定义了的属性。

1.7.4 __published 关键字

__published 是 C++ Builder 中特有的一个关键字，而且__published 关键字只能用于 VCL 类库，即从 TObject 派生的类中，用于声明变量或者函数为 “ 发布 ” 类型，同__public 关键字类似。声明在__published 域中的成员，其可见性规则与公共的成员相同，所不同的是__published 域中的成员会生成运行时类型信息（RTTI），RTTI 是 Object Pascal 风格的，它将允许应用程序能够动态地查询数据成员，成员函数和未知类型的属性。

在组件中使用__published 可以使组件被安装后，设计时__published 域中的属性可以在 Object Inspector 中被查看和更改，TCustomForm 是一个例外，因为它和它的派生类并不是一个常规的控件，通常是作为别的控件的容器而出现，只有其派生类在程序设计时，__published 域中声明的类变量、成员函数是在 Object Inspector 可见的，而设计组件的时侯，组件的属性等都不可以在 Object Inspector 查看。另外，TCustomForm 及其派生类的类成员变量在__published 域中只能为 VCL 可视组件、必

须是指针对象、而且在相应的资源文件中必须有对应的对象，也就是必须在 IDE 的 Form 中创建，__published 域中的指针对象不需要进行初始化，程序运行时会自动调用相应的资源进行初始化，这个域中的成员函数也具有可视性，可以在 Object Inspector 分配给相应类型的事件，Object Inspector 会对所有的属性、成员函数进行过滤，只显示类型匹配的属性或函数。

在__published 域不允许有构造函数或析构函数。属性、Object Pascal 内在的或 VCL 派生的数据成员、成员函数及闭合都允许在__published 域中。__published 域中定义的属性不能是数组属性，而且类型必须是顺序类型，real 类型、字符串类型、小集合类型、对象类型或方法指针类型。

1.7.5 __declspec 关键字扩展

在 C++ Builder 中，__declspec 关键字扩展的一些参数为 VCL 提供语言支持。这些参数全部是为了兼容 Object Pacal 而引入的，下面将分别简单作以介绍，这些参数在 sysmac.h 中均有对应的宏定义。通常程序员不会用到这些关键字和宏定义，C++ Builder 在编译 Object Pascal 单元时会自动生成，当确实需要增加它们时，应该尽量使用宏。

1.__declspec（delphiclass ）

delphiclass 参数用于从 TObject 派生的类的声明。这些类在创建时将遵循下列 VCL 的兼容性：

- VCL 兼容的 RTTI。
- VCL 兼容的构造函数/析构函数行为。
- VCL 兼容的异常处理。

一个 VCL 兼容的类有下列限制：

- 不允许虚基类。
- 不允许多重继承。
- 必须使用全局的 new 操作符或与之相当的动态地分配。
- 必须有一个析构函数。
- 拷贝构造函数和赋值操作符对于 VCL 派生类不会是编译器生成的。

如果编译器需要知道类是从 TObject 派生的，从 Object Pascal 转换的类声明将需要这个修饰符。

2.__declspec（delphireturn）

delphireturn 参数只在 C++ Builder 中被 VCL 内部使用。它用于 C++ Builder 创建的类的声明，这些类用于支持那些没有原本的 C++类型的 Object Pascal 内嵌的数据类型和语言结构。包括 Currency、AnsiString、Variant、TDateTime 及 Set。delphireturn 参数将 C++类标记为在作为函数调用的参数和返回值方面是 VCL 兼容处理的。当 Object Pascal 和 C++需要按值传递一个结构给函数时，需要使用这个修饰符。

3.__declspec（dynamic）

dynamic 参数被用于动态函数的声明。动态函数仅存储在定义它们的对象的

vtables 中，而不是在派生 vtables 中，除此之外都与虚拟函数类似。如果调用一个动态函数，而函数没在对象中定义，其祖先的 vtables 将被寻找直到函数被发现。动态函数只对从 TObject 派生的类有效。

4.__declspec（hidesbase）

当移植 Object Pascal 的虚拟和重载的函数到 C++ Builder 时，hidesbase 参数维护 Object Pascal 程序语义。在 Object Pascal 中，基类的虚拟函数能以相同名字的函数在派生类出现，但是它是一个新的函数，与原来的一个没有显式关系。

编译器使用在 sysmac.h 定义的 HIDESBASE 宏，指定这些类型的函数声明是完全独立的。例如：在一段 Object Pascal 的代码中，如果基类 T1 声明一个虚拟函数 func，它没有参数，同时它的派生类 T2 以相同的名字声明了一个函数，当使用 DCC32 – jphn 命令编译后，在 HPP 文件中将生成如下的定义：

```
virtual void T1::func(void);
HIDESBASE void T2::func(void);
```

若没有 HIDESBASE 声明，C++程序语义指出虚拟函数 T1::func()被 T2::func()重载。

5.__declspec（package）

package 参数指示定义类的代码可被编译到包中。当在 IDE 创建包时，这个修饰符由编译器自动生成。关于包的更多信息，参见后面章节。

6.__declspec（pascalimplementation）

pascalimplementation 参数指出定义类的代码以 Object Pascal 实现。这个修饰符在一个带有.hpp 扩展名的 Object Pascal 可移植头文件中出现。

1.8 其他 C++ Builder 扩充

1.8.1 __automated 关键字

这个关键字和 public、private 及 protected 关键字是类似的，对于 automated 成员和 public 成员相比较而言，它们的可视性规则是相同的，所不同的是声明的定义域不同，这个关键字就是指定为 automated 定义域，例如：

```
class myclass : TAutoObject
{
//这一部分用于声明成员函数和属性
// and properties that need OLE automation information.
__automated :
    // __dispid 关键字指示了联合的 OLE 自动化对象，派发 id 1000 给成员函数 func。
    void __fastcall func(void)  __dispid(1000);
};
```

声明在 automated 定义域的成员才可能具有建立 OLE 自动化对象的能力，和普通的成员函数和属性相比较，__automated 关键字后的成员和属性又具有以下的要求：

- 对于成员函数，其返回值（若有）和参数的类型必须是 Currency、OleVariant、DelphiInterface、double、int、float、short、String、TDateTime、Variant 和 unsigned short 等类型之一，否则编译器会报错。
- 成员函数必须采用__fastcall 调用规则。
- 成员函数可以为虚拟函数
- 在成员函数的参数列表后面可以加上__dispid(constant int expression)关键字。
- 属性的声明部分只能包含__dispid、read 和 write 部分，而不能象一般 VCL 类包含 index、stored、default 和 nodefault 等部分。
- 属性的存取部分列表必须是成员函数，而不能是成员标识符。
- 属性的存取函数必须是__fastcall 调用规则的。
- 属性的重载是不允许的。

1.8.2 __thread 关键字

和 C++的其他方法一样，TThread 的 Execute 方法及其调用的任何函数，都有其自己的局部变量。Execute 和其调用的函数一样也能访问程序的全局变量。这为线程之间的通讯提供了非常方便的途径。但是，在某些情况下希望各个线程在使用全局变量时互不影响，但在线程中有时需要使用全局变量，而不需要额外的参数传递，这时就可以使用__thread 关键字来定义一个线程局部（thread-local）变量，例如：

```
int __thread x;
```

就是声明一个整型变量，对应用程序中的各个线程来说它是私有的，当 Execute 执行的时候，会从全局变量 x 获取一份拷贝，这个拷贝在其内部是全局变量，但又不会影响到原来的变量以及其他线程获取新的拷贝。__thread 关键字只能用于全局变量或静态变量。指针和函数等变量不能作为“线程局部”变量。“在写入时复制”类型的变量，如 AnsiStrings 也不能作为“线程局部”变量。需要运行时初始化或运行时完成的程序元素不能被声明为__thread 类型。下面的声明就是非法的，原因是需要运行时初始化。

```
int f();
int __thread x = f();   //非法
但是
int __thread x = 50;    //合法
```

是合法的，因为它的初始化不是在运行时完成的。另外，类的实例也不能作为“线程局部”因为类需要运行时调用构造函数进行初始化，例如：

```
class X
{
   X( );
   ~X( );
};
X __thread myclass;   //非法
```

需要注意的是，线程在获取“线程局部”变量的拷贝时，只和最初声明的初始值有关，也就是说，线程中的和非线程中的函数和例程之间，对__thread 关键字修饰的全局变量也是相互独立的，不管 TThread 创建的时，“线程局部”变量的取值如何，线程所获取拷贝的值总是声明该变量时初始化的取值，因为主进程实际上也是一个特殊的线程，它也是从“线程局部”变量那里获取了一份拷贝而已。

2 VCL 类库层次结构

C++ Builder 使用了 VCL 作为编译器的 FrameWorks，因此对 C++ Builder 使用的娴熟程度很大程度的决定于对 VCL 类库的了解。可以这样说，一个应用程序所能用到基本上 Borland 也已经想到了，并且已经渗透在 VCL 类库中，我曾碰到的情况是很多网友提出了一个特殊要求、或者特殊功能的应用，非常困惑，不知道使用 C++ Builder 该如何解决，结果是使用 VCL 本身的属性、事件和方法，不需要进行任何扩充，就可以非常容易的实现，而困惑的原因正是对 VCL 本身的了解不够透彻。

本书并不准备对 VCL 的使用进行详细的讲解，而且内容也非常的多，关于 VCL 类库的更深入的了解需要读者参考有关专门的书籍或者 C++ Builder 的联机帮助。本节内容将主要针对 VCL 类库的层次结构作一简单介绍，并将主要的分类的一些重要的属性、事件和方法作一介绍，这些内容是对于使用面向状态设计和后面分析例程所必须的。而且通过对 VCL 结构的了解，可以使读者建立一个关于 VCL 整体框架的认识。

2.1 区分 VCL 类库、组件和控件

在进行深入的了解以前必须先了解几个概念，这就是 VCL 类库、组件和控件，在前面几章中，这几个概念都出现过，但并没有明确其含义，很多书籍中对此也没有做明确的解释，有时候也很模糊。在本书后面章节中所出现的这几个名词，都具有以下特定的含义：

VCL 类库，VCL 类库指的是从 TObject 派生的类，所有的 VCL 组件、控件都是 VCL 类库的一部分，但并不是所有的 VCL 类库都是组件或者控件，在 C++ Builder 中 FrameWorks 除了 VCL 类库以外还有一些类库不是从 TObject 派生而来，这些多用于 C++语言对 VCL 类库的支持而新增加的基本类库如 AnsiString 等，或者是为应用 ActiveX 等组件而添加的基本类库，它们都属于 C++ Builder 的 FrameWorks 的一部分，但并不是本书的内容，本书也不作过多描述。

组件，严格的讲是 VCL 组件，本书中所提到的组件均指由 TComponent 派生而来的

类库，TComponent 定义组件必须具有的一些基本特征，大致来讲组件应该具有以下特征：

- 能够被注册在组件面板上，并且可以通过 Form 设计器进行设置，也就是设计时可用。
- 可以作为其他组件的拥有者（Owner），或者可以拥有别的组件。
- 增强从流构建或者归档，这一点说明组件是可以在 dfm 文件被记录存储的。而非组件的类库则可能不具备这种能力。
- 可以在 New Objects 对话框中使用 ActiveX 向导转换成为 ActiveX 控件或者其他的 Com 控件。

控件，本书中所说的控件也是指 VCL 控件，VCL 控件是从 TControl 类派生而来的类库，所谓控件是一种特殊的组件，因为 TControl 是从 TComponent 派生而来。控件具有组件的特征，也有自己的特点，和一般组件最大的区别是控件均应该是“可视的”，即程序运行时控件是操作员可以看到的，并且可以响应用户的操作，即通过适当的操作完成某些功能。控件一般是可以接收 Windows 消息并能够进行处理的。

需要注意的是并不是所有的控件在程序中都是“可见的”，实际上从 TControl 派生来的类库中，大多数的并不是可见的，也并不出现在组件面板中，它们大多是某一类最终可视控件的共性的集合体，而且往往是虚拟基类。比如 TControl 是所有控件的基类，它定义了所有控件的共同特征，但实际中永远不会直接使用 TControl。

2.2 TObject 基类

TObject 是所有 VCL 类库共同的、最原始的基类，TObject 并没有任何数据成员，只有一些成员函数，这些成员函数定义了 VCL 类库的基本行为规范，因此在 TObject 类中的成员函数大多数都是虚拟函数，而且很多都是纯虚拟函数，这是这个原因，使得 TObject 的子孙将沿袭了 VCL 风格的特征，而区别于一般的类。

在 TObject 中并没有成员，因此其构造函数也将是空的，这是一个特例，包括 TObject 的派生类 Tpersistent 也是一个具有空构造函数的类。下面将就 TObject 的成员函数作以简单介绍，详细了解请参阅联机帮助。

ClassInfo、ClassName、ClassNameIs、ClassParent、ClassType：

这几个函数都是提供类信息的函数，这是体现 VCL 风格的重要函数，其中前四个均有两个版本，一个是静态函数，另一个是为了使用方便而扩展的函数，如 ClassInfo 的原型如下：

```
typedef TMetaClass* TClass;
static void * __fastcall ClassInfo(TClass cls);
void * __fastcall ClassInfo(){return ClassInfo(ClassType()); }
```

ClassType 用于动态的判断对象的类型，与__classid 相当，但是__classid 操作符需要使用对象的名称作为参数，只能静态的进行类型判断。

Dispatch、DefaultHandler：

提供对象进行消息处理的虚拟函数，Dispatch 会自动分发消息到相应的处理函数，

当指定的消息没有处理句柄时将会调用 DefaultHandler 进行处理，需要注意的是这并不是 Windows 的消息处理，而是一种类似 Widnwos 消息的处理机制，通过重载这两个函数，可以使类库响应用户定义的特定消息。Dispatch、DefaultHandler 的参数均为 void*类型，在 TWinControl 中正是重载了这两个函数，是其成为 Windows 消息的处理函数。

FieldAddress：

这是一个供对象内部使用的函数，用来提供定义在__publis 定义域中字段的地址，通常是在组件的流式构造和归档中使用，用户不需显式调用。

Free：

用于销毁对象，一般不需要直接调用，在 C++ Builder 中应当使用 delete 关键字进行销毁，但是在 VCL 类库中允许直接使用 Free 进行销毁。

InheritsFrom：

用于判断两个对象是否属于派生和继承关系。

InitInstance：

用于初始化一个新分配的对象实例，将所有成员置“零”，然后初始化该对象的虚拟函数列表指针。程序中不需要直接调用该函数，当重载 NewInstance 函数时，必须在最后一行代码中调用 InitInstance，此外 InitInstance 是一个非虚拟函数，不能够重载，对成员的初始化需要在构造函数中进行。

MethodAddress

获取一个发布（published）的方法的地址，程序也不需要直接调用该函数，而只有一个对象从 Stream 构造时才会使用到这个方法，是由系统自行完成的。通常是在 VCL 类库的事件被初始化是使用该函数，将资源文件中的方法名称转换为相应的入口地址。

MethodName

用于获取一个指定入口地址的函数的名称，这和 MethodAddress 是一个相反的过程，用于 VCL 类库流式输出的时侯。关于 VCL 的流式输入输出见后面 TStream 类库的介绍。

SafeCallException

用于处理 OLE 的异常情况。对于 COM 接口的异常处理需要重载该函数。

TObject 是所有 VCL 类库的基类，简言之，TObject 具有以下 VCL 的共同特征和行为：

- 对于 VCL 对象创建和销毁的共有特征。
- 对 VCL 对象的类型信息的支持，以及发布属性的运行时类型的支持。
- 对消息处理的支持

大多数 VCL 类库的基本行为都是从 TObject 类的方法继承而来，而且 TObject 的大多数的方法都是供 VCL 类库内部使用的，用户一般不会直接调用。但在一些复杂的组件或者控件中，它们的表现行为要比 TObject 中的复杂的多，这通常是通过重载 TObject 的方法实现的。作为一个 C++ Builder 的程序员，有必要对 VCL 作深入的了解，尤其是希望使用 C++ Builder 开发组件的程序员。

2.2.1 TObject 的派生类

下面就 TObject 的直接派生类作一简单介绍。在这些直接派生类中，很多类都是可以直接使用的，它们都是一些简单类，由一个共同特征——都属于“暂时”类，指的是，这些类都没有用来在对象被销毁之前保存状态的方法，它们不是固有的。

从 TObject 直接派生的类大约有以下 50 个：

TPersistent	TBasicActionLink
TInterfacedObject	TStream
Exception	TWebRequest
TWebResponse	TBasicActionLink
TBits	TFiler
TList	TThreadList
TConversion	TOrderedList
TDragObject	TMouse
TDataSetDesigner	TFields
TLookupList	TPaintControl
TBookmarkList	TBDECallback
TDBError	TParamList
TSessionList	TDSTableProducerEditor
TPropertyCategory	TPropertyCategoryList
TPropertyEditor	TPropertyFilter
TMonitor	TSharedImage
TChangeLink	TCustomIniFile
TMask	TPrinter
TXMLData	TEnumPropDesc
OutlineError	TUpdateTree
Tregistry	TCustomWinSocket
Tthread	TDataBlockInterpreter
TSynchroObject	TLanguages
TWebComponentList	TLayoutAttributes
TWebContentOptions	TMultiReadExclusiveWriteSynchronizer

在这些派生类中，有一些是封装了数据结构的，比如：TBits、TList、TStack、TQueue 等；有一些是封装了硬件设备的操作，比如 TPrinter、TMonitor、TMouse；还有一些是针对某种特殊应用程序的。在这些派生类中有三个分支是 VCL 中比较重要的，它们分别是 TStream、Exception 和 TPersistent。关于其他派生类的详细说明请参考联机帮助。

2.2.2 TStream

TStream 是这一分支里其它类型的典型代表。TStream 是所有流对象的基类，使用流可以与磁盘文件、动态内存等等各种存储介质进行读写操作。TStream 也是定义了流的基本特征，通常并不能直接使用，而是针对不同的流源而使用不同的 TStream 派生类：

TFileStream	用于文件操作的流。
TStringStream	用于字符串流的操作，通常是位于内存中的字符串。
TMemoryStream	存储器流，通常用于内存缓冲。
TBlobStream	用于数据库的大二进制块字段的读写。
TWinSocketStream	用于读写 Socket 的流。
TOleStream	用于 COM 接口读写的流。

不同用途的流，在结构和用法上稍有不同，但差别并不特别大，这里仅就流的共同属性和方法作以简单介绍。

属性：

Position

这是一个可读可写的属性，用于记录当前的存取位置，以字节为单位。

Size

也是一个可读可写的属性，用于确定流的大小，对于 TStream，设置 Size 是无效的，其大小是由内部的处理过程完成的，某些派生类的 Size 属性被重载，如 TMemoryStream 是允许设置 Size 来改变流的大小。

方法：

在 TStream 的方法中，定义了流操作的基本行为，大致如下：

~TStream、TStream

构造、析构函数，其中构造函数是一个纯虚函数，必须在派生类中实现。

CopyFrom

原型为：`int __fastcall CopyFrom(TStream* Source, int Count);`

从指定的的源流中拷贝指定数量的字节内容。该函数会覆盖之前的数据，如果指定的数量为 0，则将 Source 的 Position 置零，并完整拷贝整个源流的内容，如果 Count 不为零，则从 Source 的当前 Position 开始拷贝。函数返回值为实际拷贝的字节数。使用该函数可以在不同类型的流之间传递数据。

Read

原型为：`virtual int __fastcall Read(void *Buffer, int Count) = 0;`

是一个纯虚函数，提供给派生类重载，函数从流中读取指定数量字节的内容到缓冲区。每个派生类必须对该函数进行实例化的重载，因为这是一切流读取操作的基本行为，ReadBuffer,ReadComponent 都是通过调用 Read 来实现最终的读取。

ReadBuffer

原型为：`void __fastcall ReadBuffer(void *Buffer, int Count);`

和 Read 类似，通常用于已知的，相对固定的 Count 的读取，如结构、类等，该函数也用于内部调用实现 Stream 的拷贝，与 Read 不同的是这个函数具有 VCL 的异常处理能力，当不能获取指定的 Count 字节时，会抛出一个异常。

Seek

原型为：`virtual int __fastcall Seek(int Offset, Word Origin) = 0;`

是一个纯虚函数，作用是将 Position 移动指定的字节数 Offset，Origin 表明指定偏移的其实位置，可以为：

soFromBeginning 从流的起始位置，此时 Offset 必须>= 0。

soFromCurrent 从当前位置移动后的 Position 为 Position + Offset。

soFromEnd 从流的结尾，此时 Offset 必须<= 0。

SetSize

原型为：`virtual void __fastcall SetSize(int NewSize);`

提供一个虚拟函数，用于改变流的 Size，该函数也是 Size 属性的存储函数，在 TStream 中为空函数，其派生类根据实际情况重载。

Write

原型为：

`virtual int __fastcall Write(const void *Buffer, int Count) = 0;`

一个和 Read 相对应的纯虚函数，作用正好相反。

WriteBuffer

原型为：`void __fastcall WriteBuffer(const void *Buffer, int Count);`

和 ReadBuffer 的作用相反。

TStream 的派生类在属性和方法上和 TStream 稍有差别，读者要详细了解可参考联机帮助文件。

2.2.3 Exception

Exception 提供了 VCL 中异常处理的异常基类，这也是在 VCL 中仅有的不是以 T 开头命名的类库，从 Exception 派生的异常类有几十种，它们覆盖了数学、文件 IO、非法类型、数据库、内存等等 VCL 类库所涉及到的异常范围。这些派生类也均以 E 开头命名。尽管这些派生类有者不同的用途和特征，但实际上它们和 Exception 的型式是完全相同的，方法也是完全相同的（除了构造函数和析构函数），因此我们这里仅仅对 Exception 进行一些讲解，关于 C++ Builder 异常处理读者可以参考后面的异常处理章节。

属性：

HelpContext

一个整型的可读可写属性，用以标记上下文相关的帮助内容的 ID，该帮助是和相应的异常类型关联在一起。

Message

AnsiString 类型的可读可写属性，用于 Exception 对话框中显示异常类型的文本

信息，该属性可以是在构造函数中作为参数传递而获取，也可以当做动态的参数而创建，或者按照动态或静态的格式化参数从流中读取。

方法：

除了继承 TObject 的方法以外，Exception 只有构造函数和析构函数，析构函数为一个虚拟函数，而构造函数有多个不同的型式，分别如下：

```
__fastcall Exception(const AnsiString Msg);
__fastcall Exception(const AnsiString Msg, const System::TVarRec * Args,
                     const int Args_Size);
__fastcall Exception(int Ident);
__fastcall Exception(int Ident, const TVarRec * Args, const int Args_Size);
__fastcall Exception(const AnsiString Msg, int AHelpContext);
__fastcall Exception(const AnsiString Msg, const System::TVarRec * Args,
                     const int Args_Size, int AHelpContext);
__fastcall Exception(int Ident, int AHelpContext);
__fastcall Exception(int Ident, const System::TVarRec * Args, const int
                     Args_Size, int AHelpContext);
```

这些构造函数有不同的构造方式，供不同的派生类使用，Exception 的构造不能使用 new 关键字，否则不会产生例外，取而代之的是 throw 关键字，如：

```
throw Exception("My error message");
```

将构造一个 Exception 异常类，抛给当前的执行代码被 catch 等关键字中的代码捕获并处理，其 Message 信息为"My error message"。

2.2.4 TPersistent

TPersistent 为 TObject 派生类中最为重要的一个类，所有的 VCL 可视化类库都从它派生而来，TPersistent 同 TObject 一样，没有任何成员，但 TPersistent 扩展了 TObject 的固有特征，是其具备以下特点：

- 定义了针对非发布的数据存储到流或者从流载入的过程。
- 提供了为属性赋值的方法。
- 提供了从另一个类中拷贝赋值的方法。

TPersistent 最大的特点就是提供了流的存储和载入机制，从 TPersistent 派生的类中都继承了两个重要的方法 SaveToStream 和 LoadFromStream。这些方法向对象提供固有特性。例如：当窗体设计器需要创建 DFM 文件（存储窗体及其组件信息的文件）时，它循环检查窗体的组件数组，并对其中所有的组件都调用 SaveToStream 方法。每个组件都“知道”如何将被改变的属性写入流中（如文本文件）。反过来，若窗体设计器需要从 DFM 文件中装入组件属性时，它循环检查窗体的组件数组，并对其中所有的组件都

调用 LoadFromStream 方法。因此，任何 TPersistent 的派生类都有保存状态信息和需要时还原它们的能力。

TPersistent 扩充的方法有：

Assign

原型为：**virtual void __fastcall** Assign(TPersistent* Source);

提供从相似的类中拷贝对象的方法，前面说过在 VCL 类库中没有类似 Ansi C++中的拷贝构造函数，Assign 可以看作是弥补这一缺陷而引入的。这是一个虚拟函数，不同的派生类需要根据实际情况重载该函数。

使用 Assign 可以从其他对象中复制属性或者其他的特征，通常使用如下：

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

这一个语句将告诉 Destination 对象从 Source 复制内容 Destination 中。

多数从 TPersistent 派生的类都需要重载 Assign 函数来处理从相似的对象中拷贝内容，当一个类重载 Assign 函数时，遇到 Source 中有自身不能处理的成员或者属性，需要调用父类的 Assign 函数进行处理。如果 Destination 中没有重载 Assign 函数，该方法会通过 TPersistent 的 Assign 函数来调用 Source 类的 AssignTo 函数进行处理，Source 类中如果没有定义 AssignTo 函数，则 TPersistent 的 Assign 函数会抛出一个异常 EconvertError。

通常情况下 Destination = Source 和 Destination->Assign(Source)是完全不同的，前者是一种引用的拷贝，两个类完全相同，而且是指向同一个对象，而后者是对内容的拷贝，执行的结果是 Destination 和 Source 是两个对象，而且这种操作允许对象的类型不同，这取决于程序员这设计 Assign 方法时的心情。比如 TBitmap 就可以通过 Assign 拷贝 TJPEGImage，这就是由 TJPEGImage 组件的设计师决定的。

在另一种情况下 Destination = Source 和 Destination->Assign(Source)是等价的，这只出现在一个类的属性也是 VCL 对象，并且这个属性的存储函数使用了 Assign 方法来设置属性的存储值。这种要求是必须对，因为在一个组件中属性为 VCL 对象时，实际上这两个对象构成了一种逻辑包含关系，属性只是表达了这种关系，而不是需要根本上作一份拷贝。

AssignTo

原型为：**virtual void __fastcall** AssignTo(TPersistent* Dest);

提供向指定的 Dest 目标对象复制内容的方法，正如在 Assign 方法中提到的，使用 AssignTo 方法可以扩展某些已有类的 Assign 方法。

当创建一个新的 VCL 类时，需要重载 Assign 函数，对每一个已经存在的 VCL 类库都要逐一考虑，若希望能够从其中复制数据的类，就要在 Assign 方法中加入相应的代码，但是对于逆向过程，希望从已有的类中拷贝新建类的数据时，由于已有类的 Assign 方法没有定义对新建类的 Assign 处理代码，TPersistent 会自动调用新建类的 AssignTo 方法。因此，在一个新建的 VCL 类中，需要针对每一个需要相互能够复制数据的已有类的重载 AssignTo 方法。

TPersistent 提供了 Assign 和 AssignTo 方法的相互协作关系，这使得一个新建的 VCL 类不需要依赖于已有类库的方法和特征而实现之间数据的相互复制。由此也可以看出 VCL 类库中的这种机制比 C++中的拷贝构造函数功能也要强大得多。

比如两个不同的类，A 是 VCL 中固有的类，B 是用户新建的类：

```
B->Assign(A);
```

若 B 中重载了 Assign 方法来处理复制 A 类的数据，则调用 B 的 Assign 方法并正确返回，相反的：

```
A->Assign(B);
```

由于 A 中预先没有定义处理复制 B 的 Assign 代码，会最终调用 TPersistent::Assign 方法来处理，TPersistent::Assign 则自动调用：

```
B->AssignTo(A);
```

来处理这个过程，如果 AssignTo 没有定义复制到 A 的代码，TPersistent 将抛出异常。

DefineProperties

原型为：**virtual void __fastcall** DefineProperties(TFiler* Filer);

提供了一个用于存储和载入未发布数据的接口。

在 TPersistent 的派生类中可以通过重载这个函数来未发布的数据到流中，缺省的情况下，存储一个对象时，只将所发布的属性进行存储，而读取时，也是将发布的属性从流中读出，并赋给相应的属性，希望能够对未发布的数据进行存取就需要重载 DefineProperties 方法，但并不是必须要重载。

重载该方法需要注意：

- 需要调用父类的 DefineProperties 方法。
- 调用 filer 对象的 DefineProperty 方法。
- 调用 filer 的 DefineBinaryProperty 方法。

GetNamePath

原型为：**DYNAMIC** AnsiString **__fastcall** GetNamePath( );

用于返回 Object Inspector 中对象的名称的信息，这是一个内部使用的函数，用户不需直接调用，对于 components，该函数返回其名称；对于 TCollectionItem 对象，该函数返回其宿主组件、所书属性以及 collection 中的 Index 索引值。

GetOwner

原型为：**DYNAMIC** TPersistent* **__fastcall** GetOwner(**void**);

用于返回一个对象的所有者。

GetOwner 被 GetNamePath 使用，以获取 persistent 对象的拥有者，

TPersistent 本身的 GetOwner 返回为 NULL。

对于 TCollection 对象，GetOwner 方法返回的是 collection 的拥有者；对于 collectionitems，该方法返回的是 collectionitems 嵌入的 collection；对于组件，该方法返回的是 Owner 属性。

关于 TCollection 以及 TCollectionItems，详见后面内容。

2.3 TPersistent 分支

在 TPersistent 分支中包含的类有以下 28 个：

TGraphic	**TComponent**
TCollectionItem	**TCollection**
TStrings	**TClipboard**
TIconOptions	**TListItem**
TListItems	**TMonthCalColors**
TParaAttributes	**TTextAttributes**
TTreeNode	**TTreeNodes**
TSizeConstraints	**TDataLink**
TColumnTitle	**TControlScrollBar**
TGraphicsObject	**TCanvas**
TPicture	**THTMLTagAttributes**
TCaptionAttributes	**TGridAttributes**
TGridRowAttributes	**TBaseArray**
TOleServer	**TAbstractSocket**

在这些派生类中，比较重要的类有：

- TGraphicsObject，图形对象的基类，封装了 Windows 图形对象：TBrush、TFont、TPen。
- TGraphic，图标、位图、图元文件等能存储和显示可视图像的对象的基类，封装了 TBitmap、TIcon 和 TMetaFile。
- TCollection 是 TCollectionItem 容器的基类。
- TCollectionItem，包含特殊预定义项的索引集。
- TComponent，所有组件的共同基类。

下面就这几个类进行简单的介绍，了解其它的派生类请读者参考联机帮助。

2.3.1 TComponent

TComponent 是所有 VCL 组件的共同基类，组件就是在设计时可在窗体中操纵的对象。虽然 VCL 名为可视组件库，但其实包含的大部分类库为非可视化的组件。VCL 组件都是固有的对象，继承了 TPersistent 的特征，其功能主要有：

- 可出现在组件面板上并可在窗体设计器中修改。
- 可拥有和管理其他的组件。

- 增强的流及文件管理能力。
- 可通过 New Objects 对话框中 ActiveX 页的向导程序转换成 ActiveX 控件或其他 COM 对象。

TComponent 扮演了一个所有组件都可插入的标准“总线”的角色。TComponent 通过 Name 和 Owner 属性及一些方法规定组件在设计时的行为。所有派生于 TComponent 的类都有 Name 和 Owner 属性。Owner 属性值为组件的拥有者，组件的拥有者负责该决定该组件的销毁时刻。下面对 TComponent 的属性、方法作个简单介绍，TComponent 中没有事件。

TComponent 中的属性、方法都比较多，从其派生的组件也都有着非常多的属性和方法，由于篇幅的原因，这里不可能一一介绍，而且有一些内部使用的方法和属性等，也不是经常使用的，这里仅对比较重要的和常用的属性、方法进行简单介绍，后面将要介绍的类库同样只对重要和常用的属性、方法和事件作个简单介绍。

属性：

ComponentCount

一个整型只读属性，表明该组件所拥有的组件的数量，TComponent 的特征之一就是可以拥有和管理其他组件，这是依赖于 Owner 属性的，若组件 A 的 Owner 属性为 B，则 B 就是 A 的拥有者，当 B 销毁时，B 负责将 A 也销毁。B 的 ComponentCount 属性记载了所有 Owner 为 B 的组件数量。

ComponentIndex

一个整型的可读可写的属性，用于记录组件在其拥有者中组件数组的索引位置，通常联合 Components 同时使用，用于遍历一个组件拥有的所有组件。ComponentIndex 的最大取值为其拥有者 ComponentCount 属性减 1。

Components

一个 TComponent 数组属性，只读。记录了一个组件所拥有的所有组件。

ComponentState

一个 TComponentState 类型的只读属性，用于记录当前组件的状态，以表明允许或禁止某些操作或行为，这是由组件内部负责更新和设置的，对于新设计的一个组件，如果有需要禁止的行为就需要针对 ComponentState 属性添加相应代码。TComponentState 的状态有以下取值：

csDesigning csDestroying csFixups csFreeNotification
csInline csLoading csReading csUpdating
csWriting csDesignInstance

详细信息参阅联机帮助。

ComponentStyle

管理一个组件行为的属性，TComponentStyle 类型，只读。

TComponentStyle 可供选择的取值有 csInheritable, csCheckPropAvail。该属性用于决定一个组件是否能够被继承，或者需要检测属性的可读性。

Name

AnsiString 类型的可读可写属性，用以记录一个组件的名称。也可以通过设置该属

性来改变一个组件的名称。需要注意的是，一般情况，设计时在 C++ Builder 的 IDE 中创建的组件都会在相应 Form 的__publish 定义域，而且其声明的指针标识符和组件的名称相同，但这是两个不同的概念，一个是变量名称，只是一个标识符，而另一个是字符型的成员，记录这一个组件的名称信息。Name 属性在运行时是可变的，而指针变量会在编译时被替换成相应的存储器地址。

Owner

一个 TComponent 类型的只读属性，用来记录一个组件的拥有者。

Owner 和拥有者的 Components 属性共同构成了一种逻辑包含关系，一般情况，设计时创建的组件，其 Owner 就是容纳这个组件的 Form 窗体、或者 DataModule 等组件，而 Form 或 DataModule 的 Components 属性中就一一记录着每一个所拥有的组件。

Owner 在设计时和运行时都不允许改变，一个组件在创建时就必须指定其拥有者，一旦建立这种关系，直到这个组件被销毁都是不能改变的。

在程序中，设计时创建的组件是必须依赖于 Owner 而存在，当一个组件被销毁时，它的 Components 属性中所记录的所有组件也将一并销毁。一个程序中，当某个 Form 被销毁时。这个 Form 所容纳的组件随着一起销毁，程序中所有的 Form 其 Owner 均为 Application 对象，一个程序结束时，Application 将被销毁，所包含的所有 Form 也将被销毁。

程序中允许动态创建 VCL 类库的对象其 Owner 为空，但是程序中必须有程序员自行负责对该组件进行销毁，否则一旦创建直到程序运行结束时，这个组件所占用的内存才会被释放。

Tag

一个整型的可读可写属性，这个属性并没有特定的含义但很有用处的属性，为程序开发者保留了一个 4 字节的成员，用来完成一些特定功能，比如利用该属性存储一个同类型的指针，就可以构成一个单向链表等等。

方法：

ChangeName

原型为：`void __fastcall ChangeName (const AnsiString NewName);`

改变私有的内部 Name 属性的存储值，NewName 为要改变的新名称，在程序中不能直接使用该方法，而是使用 Name 属性进行改变，该方法不能重载。

DefineProperties

原型为：`virtual void __fastcall DefineProperties(TFiler* Filer);`

这是从 TPersistent 派生而来的方法，作用相同，在 TComponent 中，该函数用来构造了两个虚拟属性，或叫做伪属性：Top 和 Left，所有的组件在设计时是允许可见的，设计时创建的组件会在 dfm 文件中保存它们的位置 top 和 left，然而这两个属性在运行时是不需要的，因为只有控件在运行时才是真正可见的。

DestroyComponents

原型为：`void __fastcall DestroyComponents(void);`

用于销毁所有拥有的组件，该方法不需直接调用。

Destroying

原型为：**void __fastcall** Destroying(**void**);

Destroying 设置 ComponentState 属性到 csDestroying 标记，并且将所拥有的组件的 ComponentState 属性也一并设置，一般不需直接调用。

ExecuteAction

原型为：

DYNAMIC bool __fastcall ExecuteAction(TBasicAction* Action);

用来调用一个 Action 动作，并将该组件作为这个动作的目标。这个调用过程遵循以下原则：若 Action 的 OnExecute 事件没有指定处理函数，将会调用 Application 的 ExecuteAction 方法，Application 的 ExecuteAction 方法首先触发 Application 对象的 OnActionExecute 事件，若该事件没有处理函数，则触发 OnExecute 事件，若 OnExecute 同样没有处理函数时，将会调用当前活动控件的 ExecuteAction 方法。

FindComponent

原型为：

TComponent* **__fastcall** FindComponent(**const** AnsiString AName);

用来查找指定名称的组件是否包含在 Components 内。由于 Object Pascal 的缘故，这个函数中并不区分 AName 参数的大小写。

TComponent

原型为：**__fastcall virtual** TComponent(TComponent* AOwner);

TComponent 的构造函数，所有的组件在设计时创建的，都不要在程序中使用代码进行实例化，这个工作在程序运行时会自动完成。运行时创建的组件必须使用 new 关键字来实例化并指定 AOwner 参数作为组件的拥有者，允许 AOwner 为空。

2.3.2 TGraphic

TGraphic 是一个抽象基类，不能被实例化，但它定义了图形对象的许多共同特征。TGraphic 的派生类对很多方法进行了重载来适应不同格式的图像。从 TGraphic 派生的图像类支持3种格式bitmap、icon和metafile等，另一个与之相关的类是 TPicture，如同一个容器一样，使用 TPicture 可以在不知道格式的情况接受 TGraphic 支持的格式。

属性：

Empty

布尔类型的只读属性，记录着图像是否为空。

Height

整型可读可写的属性，以像素为单位，表示着图像的高度。

Modified

布尔类型可读可写的属性，记录了图像文件是否被编辑或改变，在 TGraphic 的派生类中，只有 TBitmap 中这个属性是有效的，其他的派生类不管是否被改变，Modified 都不能为 true。

Palette

HPALETTE 类型的可读可写属性，定义了图像的调色板，若图形格式不需要调色板，该属性可以为空。

PaletteModified

布尔类型的可读可写的属性，记录着调色板是否被改变。

Transparent

布尔类型的可读可写的属性,决定图像是否为透明的,在 metafile 和 icon 格式中，这个属性无效，因为这些格式始终都要求是透明的，但象 bitmap 格式的图像，是否透明要取决于该属性。

Width

整型可读可写的属性，以像素为单位，表示者图像的宽度。

方法：

LoadFromClipboardFormat

原型为：
```
virtual void __fastcall LoadFromClipboardFormat(Word
                    AFormat, int AData, HPALETTE APalette) = 0;
```
一个纯虚函数，用于从有效的剪贴板中载入图像，派生类需要重载。

LoadFromFile

原型为：
```
virtual void __fastcall LoadFromFile(const AnsiString FileName);
```
虚拟函数，用于从指定的文件中载入图像。

LoadFromStream

原型为：
```
virtual void __fastcall LoadFromStream(Classes::Tstream
                                        *Stream) = 0;
```
纯虚拟函数，提供从流中载入图像的方法。

SaveToClipboardFormat

原型为：
```
virtual void __fastcall SaveToClipboardFormat(Word
                    &AFormat,int &AData,HPALETTE &APalette) = 0;
```
纯虚拟函数,和 LoadFromClipboardFormat 作用相反,用于复制图像到剪贴板板。

SaveToFile

原型为：
```
virtual void __fastcall SaveToFile(const AnsiString Filename);
```
虚拟函数，和 LoadFromFile 的作用相反。

SaveToStream

原型为：
```
virtual void __fastcall SaveToStream(Classes::TStream*
                                    Stream) = 0;
```
纯虚拟函数，作用和 LoadFromStream 相反。

事件：

OnChange

TNotifyEvent 类型的事件，当图像被改变的时侯触发。

OnProgress

触发于图像被处理的过程，原型如下：

```
enum TProgressStage {psStarting, psRunning, psEnding};
typedef void __fastcall (__closure *TProgressEvent)(System::TObject*
                Sender,TProgressStage Stage, Byte PercentDone, bool
                RedrawNow, constWindows::TRect &R, const AnsiString Msg);
__property TProgressEvent OnProgress = {read=FOnProgress,
                write=FOnProgress};
```

使用 OnProgress 事件可以在图像被处理的过程中及时的向用户回馈到到前的处理状况，如使用 ProgressBar 来显示进度等情况。

2.3.3 TCollection

在 VCL 类库中，TCollection 和 TCollectionItem 是两个比较重要的类，它们并不是组件，也不能容纳在 Form 中，但是设计时仍然是可用的，有专门 TCollectionItem 的编辑器可以对其进行编辑,在 Object Inspector 中也是可见的，而且设置的属性也会被相应的存储在 dfm 文件中，但和组件不同的是它们在 Form 的头文件中并不存在相应的声明，因此相应的在代码中的引用也不能象组件一样直接使用名称，比如在 TStatusBar 中：

有一个 TStatusPanels 类型的属性 Panels，其 Items 属性容纳了众多的 TStatusPanel 对象，程序中使用 StatusPanel 时只能采用如下代码：

```
StatusPanel->Panels->Items[1]->Text = "Test"；  而不能使用象：
StatusPanel1->Text = "Test"；
```

等类似的代码，因为在 Form 的头文件的声明中不会包含 TStatusPanel 类型的变量。

在上面的代码中，TStatusPanels 是 TCollection 的派生类，TStatusPanel 是 TCollectionItem 的派生类,TCollection 是 TCollectionItem 的一个容器类，通过这两个类，在 VCL 组件之间建立了另一种逻辑包含关系，而且是使用的比较多的一种关系，要深入的理解这中包含关系，可以仔细分析 StatusPanel->Panels 的使用以及后面关于 C++ Builder 资源的相关内容，这里先就 TCollection 的重要属性和方法作个简单介绍。

属性：

Count

整型只读属性，返回 Collection 中项目的数量。所有的 CollectionItem 项目都记录在 TCollection 的 Items 属性中，以数组型式存放。

ItemClass

一个 TMetaClass 类型的只读属性，用以记录 Items 中项目的类型。

Items

可读可写的 TCollectionItem 指针类型属性，存储 TCollection 中容纳的 CollectionItem，可以以数组的方式访问每一个 Items 项目。

方法：

Add

原型为：TCollectionItem* __fastcall Add(void);

该方法用于创建一个新的 TCollectionItem 并且将其添加到 Items 数组中，方法的返回值即新建的 Item。该方法会自动更新 Count、Items 等属性。

BeginUpdate、EndUpdate

原型为：void __fastcall BeginUpdate(void);

void __fastcall EndUpdate(void);

这两个方法用于冻结和解冻屏幕的刷新。通常情况下 TCollection 和 TCollectionItem 组合的对象非常容易和屏幕的显示内容关联在一起，而且每一个更新动作都会触发对屏幕的刷新，但这在某些情况下是不希望的，比如批量更新一个 Items 时，屏幕会频繁闪烁，而且执行效率也会随之降低，这时候，可以使用 BeginUpdate 将屏幕刷新挂起，等批量更新结束后再使用 EndUpdate 恢复屏幕刷新。

Clear

原型为：void __fastcall Clear(void);

用以清除 Items 中的所有 CollectionItem。

Delete

原型为：void __fastcall Delete(int Index);

用于删除指定索引号的 Items 元素。

Insert

原型为：TCollectionItem* __fastcall Insert(int Index);

该方法用于创建一个新的 TCollectionItem 并且将其插入到 Items 数组中，与 Add 方法不同的是，该方法的 Index 参数指定了一个索引号，新建的 Item 将插入到相应的位置，Add 则是添加在数组的末尾。

TCollection 对象中没有事件。

2.3.4 TCollectionItem

属性：

Collection

一个 TCollection 类型的只读属性，指定该 TCollectionItem 的容器 TCollection。这个属性在构件 TcollectionItem 时就被指定，之后不允许改变。TCollectionItem 的 Collection 属性和 TCollection 的 Items 属性,相互构成了一种特定的逻辑包含关系。

DisplayName

一个 AnsiString 类型的可读可写属性，这个属性类似于组件的 Name 属性，它是一个 Item 出现在 Collection editor 中的名称，Collection editor 专门用于编辑 TCollection 的 Items 属性。该属性不能象 Name 一样在代码中引用，只能用于

Collection 编辑器，因为 Item 在程序的头文件中并没有声明的原型，程序在运行时根据 dfm 文件中的相关信息进行创建并初始化。

Index

整型可读可写属性，用于标记一个 TCollectionItem 在其容器的 Items 属性中的位置，该属性允许改变，这意味着运行时可以改变一个 Item 在 Items 数组中的位置，Index 属性的改变不会影响到其他 Item 的相对位置关系，比如：

A、B、C、D、E 五个 Item 顺序排列在 Items 数组中，它们的 Index 依次为 0、1、2、3、4，当 E 的 Index 改为 0 时，E 在 Items 中的位置将被移动到第一个，A、B、C、D 将会整体向后移动，A、B、C、D、E 的 Index 依次为：1、2、3、4、0。

ID

整型只读属性，是 TCollectionItem 永久的唯一的序列号，通常 ID 会和 Index 一致，但是当一个 TCollectionItem 被进行了删除、移动操作之后，它们将会不再一致，Index 是 TCollectionItem 在 Items 数组中的索引值，就是 Item 在 Items 数组中的位置，ID 是序列号，是一个永久的标识，比如：当一个 Item 被删除后，它的 ID 将不会重新被别的 Item 使用，Index 则可以。

方法：

Changed

原型为：`void __fastcall Changed(bool AllItems);`

用于更新 collection 的其他项目来反映出更新的结果，该方法建立了处于并列关系的 TCollectionItem 对象之间的一个联系的途径，这个方法是自动调用的，但在某些派生类中可能有必要直接调用来强性完成 Items 的更新任务。

GetOwner

原型为：`virtual TPersistent __fastcall GetOwner(void);`

和组件的 GetOwner 类似，但不同的是，TCollectionItem 对象本身并没有 Owner 属性，它多返回的 Owner 指的是 TCollectionItem 对象的容器，也可以使用 Collection 获取 Owner 容器，这里的 Owner 并不象组件的 Owner，需要负责对其拥有的组件的销毁工作。

关于 TCollection 和 TCollectionItem 这种集合及容器关系的类很多，它们都是从这两个类派生而来，有着不同的用途和功能，差别也比较大，这里不可能一一介绍，读者需要详细了解请参考联机帮助。这也是 C++ Builder 中一种非常重要的逻辑包含关系，读者需要结合实际使用仔细理解。

2.3.5 TGraphicsObject

TGraphicsObject 是一个抽象基类，它封装了 Widnows 的图形对象，包括 TBrush、TFont 和 TPen。这几个类都是 Windows 的基本会图工具。一方面，图形对象是提供给应用程序使用的封装后的对象，另一方面，图形对象还将负责爱 Windows 中共享图形资源的任务，下面就 TGraphicsObject 的属性、方法和事件作个简单介绍。

属性：

OwnerCriticalSection

这个属性的定义如下：

```
typedef _RTL_CRITICAL_SECTION TRTLCriticalSection;
__property Windows::PRTLCriticalSection OwnerCriticalSection = {read
                                    =FOwnerLock, write=FOwnerLock};
```

该属性用于获得图形对象的线程锁定状态，在一个多线程的程序，Canvas 对象都有可能被每一个线程访问，为了防止多个线程同时对一个 Canvas 进行操作而引起意外的竞争，C++ Builder 中可以对 Canvas 进行锁定，OwnerCriticalSection 属性就是用于这个目的的。

在一个 Graphic Object 对象对 Canvas 进行操作的时候，不能改变这个属性，可以使用 Lock 方法和 UnLock 方法对一个 Canvas 对象进行锁定和解除锁定。

方法：

Changed

原型为：`DYNAMIC void __fastcall Changed(void);`

当一个 Graophic Object 的属性被改变时，用于执行 OnChanged 事件。所有派生类的属性发生改变时都会调用这个方法。

Lock

原型为：`void __fastcall Lock(void);`

Lock 方法用于阻止同时执行的其他线程对相关的 Canvas 对象进行操作，在使用 UnLock 方法对这个 Canvas 释放之前，其他线程均不能进行操作。

Lock 方法是对多线程协作的一种保护，而且允许嵌套调用，在最后一层的嵌套的 Unlock 被调用之前，Canvas 不会真正的被释放。

这个方法和 OwnerCriticalSection 属性相关，当 OwnerCriticalSection 属性没有被设置的时候，Lock 方法不会真正的锁定 Canvas。TCanvas 对象会自动的设置 OwnerCriticalSection 属性。

Unlock

原型为：`void __fastcall Unlock(void);`

和 Lock 函数对应，用于解除 Lock 锁定的 Canvas。

事件：

OnChange

一个 TNotifyEvent 类型的事件，原型如下：

```
__property Classes::TNotifyEvent OnChange = {read=FOnChange,
                        write=FOnChange};
```

当一个 Graphic Object 发生改变的时候触发该事件，Graphic Object 的实行将反映出这些变化。

2.4 TComponent 分支

TComponent 分支是 VCL 中一个重要的分支，它是所有组件的共同基类。直接由 TComponent 派生的组件并不需要可视化的接口。在这一分支中包含的类主要有：

- TMainMenu 为窗体提供菜单栏及相应的下拉菜单的类。

- TTimer 包含 Windows API 中的计时器函数的类。
- TOpenDialog 、 TSaveDialog 、 TFontDialog 、 TFindDialog 、 TColorDialog 等，是通用 Windows 对话框类。
- TActionList，包含组件或控件，如菜单项和按钮的动作列表的类。
- TScreen，负责管理应用程序创建的窗体和数据模块，当前的活动窗体及其上的活动控件、屏幕尺寸和分辨率，应用程序使用的光标和字体等内容的类。

但也有更多的类不是直接使用的最终类，TComponent 的派生类有以下 37 个。

TControl	**TbasicAction**
TCustomActionList	**TADOCommand**
TCustomConnection	**TdataSet**
TCustomApplicationEvents	**TWebApplication**
TBasicAction	**TCustomContentProducer**
TCustomImageList	**TAppletApplication**
TDataModule	**TField**
TDataSource	**TBatchMove**
TDataSetUpdateObject	**TSession**
TDdeClientConv	**TDdeClientItem**
TDdeServerConv	**TDdeServerItem**
TCommonDialog	**TTimer**
TApplication	**TScreen**
TCustomObjectBroker	**TMenu**
TMenuItem	**TWebButton**
TWebControlGroup	**TWebForm**
TWebDataDisplay	**TCustomDataStore**
TDecisionSource	**TServiceApplication**
TXMLBroker	

这些派生类里面，比较重要的有：

TControl：所有控件的基类，定义了控件所具备的基本特征。

TDataSet、所有数据库表集的基类，包括 BDE、ADO 等数据库的组建。

TApplication 应用程序的类，大多数应用程序应该包含一个 TApplication 对象或者与之相当的对象，比如 TServiceApplication、TWebApplication 等，

TField 字段类，与 TDataset 对象配合使用，用于对数据库表的操作。

在 TComponent 的派生类中，还有很多类都是比较有用，但是限于篇幅，这里不能详细说明，仅就上面提到的类进行简单说明。

2.4.1 TApplication

TApplication 类是标准的 Windows 应用程序类，这个类是非常常用的，却往往被人了解的甚少，原因是 TApplication 并不出现在组建面板中，而且 Application 的

属性也不会出现在 Object Inspector 内，由此以来 TApplication 的许多功能并没有被真正发挥出来。

在一个 Windows 程序中，TApplication 封装系统对应用程序所要求的一些基本操作，如：运行、持续和关闭程序所要进行的基本操作。在 Web 应用程序中，相应的类为 TWebApplication，服务程序为 TServiceApplication 等等。

TApplication 类应该具有以下基本功能：

- Windows 的消息处理能力。
- 提供了上下文相关的在线帮助。
- 菜单和加速键的处理。
- 异常处理能力。
- 包含了操作系统定义的基本构成，如 MainWindow、WindowClass 等等。

在使用 C++ Builder 创建一个应用程序的时候，IDE 会自动根据需要创建的程序类型来声明相应的 Application 类实例。在 TApplication 组件中包含了一些程序基本的事件如：OnActionExecute、OnActionUpdate 等，由于 TApplication 并不出现在组件面板，属性和事件也不出现在 Object Inspector，这些事件也往往被忽略，若希望能够在 IDE 中使用这些事件可以通过创建一个 TApplicationEvents 组件对象，来将 TApplication 的事件映射到 TApplicationEvents 中。

接下来就 TApplication 的属性、方法和事件作以简单介绍，由于 TApplication 也是一个比较“庞大”的类库，这里对一些含义比较明确，通过名称就能够理解其含义的或者使用的较少的属性、方法不再赘述，仅对一些重要的属性、方法和事件作个介绍。

属性：

Active

指定了程序是否处于激活状态。是一个 bool 类型的只读属性。

当程序的，某个 Form 或者程序获得焦点时，程序便处于激活状态，Active 为 true，否则为 false，当程序结束之前，也会将 Active 设置为 false。

Handle

HWND 类型的可读写属性。这个属性提供了程序主窗口句柄访问的接口，在使用 Windows API 函数创建一个窗口或者 MessageBox 的时候，需要指定一个 Windows Handle 作为窗口的父窗口，使用 Application 的 Handle 属性可以提供给 API 一个宿主的窗口。在后面的例子 Sells 中，就使用了 Handle 属性作为动态连接库的一个参数。

MainForm

TForm 类型的只读属性。指定了应用程序的主窗口，一个应用程序只能有一个窗口为主窗口，主窗口会代表着整个应用程序的一些特征，比如最小化、最大化等。在 C++ Builder 中，主窗口总是 Application 创建的第一个 Form，当这个 Form 被关闭时，整个程序也就结束了。MainForm 在运行时是不能改变的，设计时可以通过 Option 选项来设置调整那一个 Form 为主窗口。

ShowMainForm

bool 类型的可读写属性。指定应用程序在启动时是否显示主窗口，缺省的情况下，该属性始终为 true，程序在启动的时候总是显示主窗口，如果一个程序希望在启动的时

候不显示主窗口，则可以通过在 WinMain 函数中 Application->Run 以前设置该属性为 false，并且将主窗口的 Visible 属性设置为 false 即可。

Title

AnsiString 类型的可读写属性。用于标识应用程序的标题，在 Windows 的任务栏，程序的图标上显示的就是 Title 的字符串，缺省的状态下，一个用于程序或者动态连接库的 Title 就是文件名称，可以在 Project | Options 对话框中修改这个缺省值，也可以在程序运行时该这个属性，当一个可执行程序的 Title 为空的时候，可执行执行程序不会出现在 Ctrl+Alt+Del 按键组合的对话框中。

方法：

BringToFront

原型为：`void __fastcall BringToFront(void);`

设置最后一个激活的窗口为桌面的最顶端窗口，不会被其他窗口所覆盖。

CreateForm

原型为：
```
void __fastcall CreateForm(System::TMetaClass
                           * InstanceClass, void *Reference);
```

用于创建一个新的窗口，程序员实际上不需要直接使用这个方法，在 C++ Builder 中,CreateForm 方法是 IDE 自动增加到 WinMain 函数中的,但代码中也允许这样使用，同样在程序中动态创建一个窗体，使用 new 关键字更符合习惯一些。

Initialize

原型为：`void __fastcall Initialize(void);`

提供了一个用于初始化象 OLE 等子系统的机会，在每一个基于 VCL 类库的出现中，Initalize 都是第一个被调用的方法，这个方法会自动调用 InitProc 过程，实际上缺省的，在用于程序中这个方法什么也没有作，因为缺省的 InitProc 指针是空的，可以通过在头文件中增加关于 InitProc 的定义或者在程序中定义自己的初始化过程,并将其值赋给 InitProc 指针。

一个程序中只能有一个 InitProc，当多个头文件都包含 InitProc 定义时，只有最后一个出现的定义是有效的，如果一个程序中确定没有 InitProc，则可以将 Initialize 安全的从应用程序中删除。

MessageBox

原型为：
```
int __fastcall MessageBox(const char * Text, const char
                          *Caption, int Flags);
```

这是一个提供显示用户消息对话框的方法，基本和 API 中的 MessageBox 一样，不同的是在这个方法中不许要指定对话框的父窗口或进程实例，其他的和 API 中的 MessageBox 一样。

Run

原型为：`void __fastcall Run(void);`

该方法通常在 IDE 生成的代码中，实际上在 WinMain 函数中，这个方法是最后调用的，直到程序结束的时候，Run 方法才会返回，该方法主要是建立 Application 对象的消息处理机制。通常需要对程序作的一些特殊处理需要在 Run 方法以前操作或者调用。

Terminate

原型为：`void __fastcall Terminate(void);`

用于结束用于程序。该方法调用了 Windows API **函数** PostQuitMessage **来向程序发送关闭指令，并不意味着程序会立即结束，而是按照正常的顺序和关闭过程来关闭程序，比如在数据没有存储的时候，可以通过代码来阻止** Terminate **方法的进一步完成关闭工作。**

事件：

OnActivate

TNotifyEvent **类型的事件，当用于程序被激活时触发。**OnActivate **一般发生在程序初始化后并正常运行时或者用于程序的焦点从别的用于程序转移到该用于程序的任意一个窗体的时候。填写该事件的处理代码可以在程序激活时进行一些特殊处理。**

OnDeactivate

TNotifyEvent **类型的事件同** OnActive **相反，当一个用于程序失去焦点的时候被触发。**

OnException

原型如下：

```
typedef void __fastcall (__closure
                  *TExceptionEvent)(System::TObject* Sender,
                  Sysutils::Exception* E);
__property TExceptionEvent OnException = {read=FOnException,
                  write=FOnException};
```

当一个没有被定义了处理过程的异常发生时触发。这是一个比较有用的事件，因为在 C++ Builder **中，对于用户没有处理的异常会有缺省的处理行为，比如在** try **语句以外发生的异常，**C++ Builder **会自动的向用户报告一个错误和相应的错误类型，通常这个错误的信息是英文输出的，若希望程序不要出现英文界面就可以使用这个事件对没有在代码中处理的异常进行简单信息输出等处理。**

OnMessage

原型如下：

```
typedef void __fastcall (__closure *TMessageEvent)(tagMSG &Msg,
                        bool &Handled);
__property TMessageEvent OnMessage = {read=FOnMessage,
                        write=FOnMessage};
```

这个事件在用于程序接收到 Windows **消息时触发，这是** C++ Builder **消息处理的一大特色，通过这个事件可以使程序员获得** Windows **消息的最高控制权，这是一个非常有用的事件，但是使用中也必须谨慎，因为每一个** Windows **消息都会触发该事件，对于每秒种出现上千次的消息，代码如果处理不当很可能造成正个程序的性能下降。此外应该注意的是该事件只对** Windows **消息队列中的消息发生作用，而对于不在消息队列中的消息，比如使用** SendMessage **函数发送的消息则不起作用。**

OnShortCut

这是和 OnMessage 事件比较类似的一个事件，原型如下：

```
typedef void __fastcall (__closure
                         *TShortCutEvent)(Messages::TWMKey
                         &Msg, bool&Handled);
__property TShortCutEvent OnShortCut = {read=FOnShortCut,
                          write=FOnShortCut};
```

和 OnMessage 不同的是这只是对 WM_KEYDOWN 消息的一个处理，严格的说是对键盘击键消息的处理，因为 WM_KEYDOWN 消息还包括了鼠标的消息。

使用这个事件可以过滤调一些不需要的消息，而只对键盘消息进行处理，比如定义额外的热键等。这个事件也是具有最高控制权的，它发生在所有控件的 OnKeyDown 和 OnKeyPress 之前。在本书所附的例子 Sells 中使用了 OnMessage 对消息进行全局处理，实际上使用 OnShortCut 会获得更好的效果和更高的程序性能，但出于例子的目的而采用了 OnMessage 事件进行处理,有兴趣的读者可以自行修改代码,使用 OnShortCut 事件。

2.4.2 TField

TField 也是一个比较重要的基类，它是所有数据库字段类型的基类，在 C++ Builder 开发的数据库程序中，TField 扮演着极为重要的角色，通常情况 TField 和 TDataSet 中的 Fields 属性构成了一种逻辑包含关系，这种逻辑关系和 TCollection 与 TCollectionItem 之间的关系比较类似，但也有所不同，因为 TField 是一个组件，因此设计时定义的 TField 对象在头文件中是有声明原型的,可以在程序代码中直接引用，而不需要通过宿主对象来间接访问。TField 对象并不出现在 Form 或其它类似的容器控件中，而使用了和 CollectionItem 编辑器相似的容器进行可视化编辑，表现形式上和 TCollectionItem 非常类似，但这是完全不同两种类型。

在 C++ Builder 中所有的数据库访问都是通过 TField 来完成的，尽管有些时候并不需要明显的在设计时创建 TField 对象（运行时 C++ Builder 会根据需要自动创建），但在大多数情况下，设计时创建 TField 对象会对应用程序的设计带来很大的方便，本书的例子 Sells 中就使用了从 TField 派生而来的对象,读者可参考阅读。下面先就 TField 的一些重要的属性、方法和设计作以简单介绍。

属性：

Calculated

bool 类型的可读写属性,用于指定该字段是否是由 OnCalcFields 事件中的代码计算而获得。缺省情况下这个属性是 false，当一个字段确定是由其它字段或者当前环境中的某些变量计算而得到的话，可以在 OnCalcFields 事件中添加相应的计算代码，并且将 Calculated 设置为 true，该属性在运行时不能改变，只能在设计时确定。

这也是一个比较有用的属性，比如一个表中包含了产品的数量和单价，那么合计价格就可以通过计算字段而获得，从程序的逻辑上讲，这也是比较合适的处理方法。

ConstraintErrorMessage

AnsiString 类型的可读写属性，用于指定一个用户的出错信息，这个信息是当该字

段的取值违反了该字段的约束条件时报告给用户的，比如一个正整形字段，为其赋的值小于 0 时，就是这样的错误。对于字段的约束条件在 CustomConstraint 属性中定义，或者是由数据库所决定的约束条件。

CustomConstraint

AnsiString 类型的可读写属性，指定一个额外的字段约束条件，这个属性是按照 ANSI SQL 的语句格式来书写的，比如要求一个整形字段的取值范围为 1~100。则可以将该属性设置为：

```
x > 0 and x < 100
```

通常这个属性被指定时，ConstraintErrorMessage 属性通常也需要针对该约束条件设置相应的输出信息。

DefaultExpression

AnsiString 类型的可读写属性，也是以 ANSI SQL 的格式指定一个用于用户没有设置该字段的时候，添加缺省值的语句，比如某个时间字段当用户没有为新建的记录没有赋值时，指定缺省值为 12 点，可以将该属性设置为：

```
'12:00:00'
```

即可

DisplayLabel

AnsiString 类型的可读写属性，用于指定显示在 DBGrid 控件中字段的标题，缺省的情况下，DisplayLabel 和数据库中字段的名称相同，但大多数情况下，应用程序的 DBGrid 中字段的标题和数据库中的字段的名称并不一样,数据库更习惯于使用英文名称，而 DBGrid 中则更适合使用中文标题，这可以通过设置 DisplayLabel 了实现。

希望更改 DBGrid 的标题也可以通过 DBGrid 的 Columns 属性来设置 Title，但这种方式将不允许在运行的过程中更改字段列表或更换 DBGrid 绑定的数据集，即使通过代码来修改，也要烦琐的多。使用 Columns 属性的另一个不可取之处是，数据集的字段显示特性没有和相应的字段结合起来，而是决定于和其相关性较弱的 DBGrid。

DisplayName

AnsiString 类型的只读属性，指定用于应用程序中用于显示目的的字段名称，其含义是，当应用程序需要输出某些信息，比如产生异常的时候，格式化的字符串中包含的字段信息。缺省情况下，DisplayName 和数据库的字段名称相同，当 DiaplayLabel 被指定时，该属性和 DisplayLabel 相同。

DisplayText

AnsiString 类型的只读属性，这个属性实际上就是经过格式化处理的字段的取值，应用程序中，对一个字段值的引用可以使用 AsBoolean、AsCurrency、AsDateTime、AsFloat、AsInteger、AsString、AsVariant 等属性，但是在界面显示，或者 Web 程序中，可能使用 Text 属性更加方便一些。

这个属性的另外一个用处是提供了应用程序隐藏真实数据的途径，当 TField 对象的 OnGetText 事件被定义的时候，若处理函数的 DisplayText 参数为 true 时，DisplayText 将是经过 OnGetText 事件处理的 Text 参数，关于 OnGetText 属性参见后面内容。

Index

一个整形的可读写属性。指定了 TField 在 DataSet 的 Fields 数组中的位置索引。DataSet 的 Fields 属性和 TField 对象构成了一种于 TCollection 对象类似但并不相同的逻辑包含关系，Fields 属性参见 TDataset 类，包括了数据集中的所有字段以及应用程序增加的字段,Index 属性可以是程序员象 TCollection 与 TCollectionItem 对象那样访问一个数据集的某个字段，比如：

```
Table1->Fields->Fields[1]
```

就是访问 Table1 的第二个字段（索引号符合 C 语言对于数组的习惯，从 0 开始），在 TDataset 对象中还提供了另一种访问途径：使用 FieldByName 方法。另外，由于 TField 本身就是一个组件，因此在程序的头文件中有其声明原型，因此在程序中也可以通过使用 TField 的原型进行访问，这也是最直接、效率最高的访问方式。

通常只在设计时更改 TFiled 对象的 Index 属性。

IsIndexField

一个 bool 类型的只读属性。用于指定该指定是否为索引字段，这是在创建数据库时就已经决定的，在数据集中定义的索引字段其相应的 TField 对象的 IsIndexField 即为 true，反之为 false。在一个 lookup table 中，其 LookupKeyFields 必须是一个索引字段。

KeyFields

一个 AnsiString 类型的可读写属性，用于指定 LookUp 字段（字段类型为 fkLookup 并且字段的 Lookup 属性为 true 时）需要进行匹配查找的字段，当包含多个查找匹配字段的时候，各个字段之间使用分号隔离。

Text

和 DisplayText 类似的属性，不同的是可读可写，用于字段的编辑状态，可以更改，Text 属性同样可以使用 OnGetText 事件重新构建，当 OnGetText 事件句柄的 DisplayText 参数为 false 时,该事件处理函数的 Text 参数将代表着 TFiled 对象的 Text 属性。

ValidChars

原型为：

```
typedef Set<char, 0, 255> TFieldChars;
__property TFieldChars ValidChars = {read=FValidChars, write=FValidChars};
```

用于指定 Text 属性的合法输入字符集，比如整形字段就可以在该属性中指定字符集为'0'~'9'，这样就可以过滤掉非法的字符输入，这一个工作通常在定义数据集字段的时候系统就已经自己完成，可以通过该属性更改法定的字符集。

Value

Variant 类型的可读写属性，提供了在运行时直接访问一个字段数据的途径，比如可以在一个字段为不可视状态下的存储和读取。

Visible

bool 类型的可读写属性，用于指定该字段是否显示于 DBGrid 控件，这个属性对于 DBEdit、DBLabel 等控件不起作用，当一个 DBGrid 使用了 Columns 属性来定义显示列的时候，该属性也不起作用。

方法：

FocusControl

原型为：`void __fastcall FocusControl(void);`

强制于该字段相关联的 Data 控件获取焦点，由于数据字段在失去焦点的时候，字段的数据会随之改变，使用该方法可以确保在没有获得正确的数据之前，字段的数据不会改变，可以在 OnChange 事件中加入相应的判断代码。

IsBlob

原型为：
```
virtual bool __fastcall IsBlob()
                        { IsBlob(__classid(TField))};
```
用于判断该字段是否为 Blob 字段，若是 Blob 字段则对该字段数据的存取应该使用 TBlobStream 对象，而不能使用 GetData 及 SetData 方法存取。

IsValidChar

原型为：`virtual bool __fastcall IsValidChar(char InputChar);`

用于判断输入的字符是否包含在该字段的有效字符集内，这通常是一个 Data 控件需要调用的方法，在新创建一个 TField 派生类的时候，可能需要重载该函数，有效字符集在 ValidChars 属性中定义。对于某些非字符输入的字段，比如大二进制数据，该方法始终返回为 true。

SetFieldType

原型为：`virtual void __fastcall SetFieldType(TFieldType Value);`

提供了一个改变字段数据类型的接口，通常数据集中固有的字段，不需要改变字段的数据类型，只有程序运行时创建的字段，或者是用于新建一个数据集时创建的字段才需要重新设置字段数据类型。

事件：

OnChange

当向一个字段数据缓冲区写入数据时触发的事件，原型为：
```
typedef void __fastcall (__closure *TFieldNotifyEvent)(TField* Sender);
__property TFieldNotifyEvent OnChange = {read=FOnChange, write=FOnChange};
```
当向一个数据缓冲区成功的写入数据的时候，将会按照以下的顺序触发事件：

1. OnValidate 事件率先被触发，来使数据生效。
2. 若 OnValidate 事件的处理函数判断数据有效，将写入数据缓冲区。
3. 若写入过程没有产生异常，则写入后触发 OnChange 事件。

OnGetText

当 DisplayText 或者 Text 属性被访问的时候触发，原型为：
```
typedef void __fastcall (__closure *TFieldGetTextEvent)(TField* Sender,
                          AnsiString &Text, bool DisplayText);
__property TFieldGetTextEvent OnGetText = {read=FOnGetText,
```

```
                                write=FOnGetText};
```

访问 DisplayText 属性时，DisplayText 参数为 true，访问 Text 属性时，DisplayText 参数为 false，若该事件没有指定处理函数句柄，则 DisplayText 和 Text 都等于 AsString 属性。

使用这个事件可以定制数据的显示格式或者隐藏真实数据，例如可以填写该事件的处理代码，将 Currency 类型的数据显示为符合中国习惯的人民币大写金额。

OnSetText

当为 Text 属性赋了一个新值的时候触发，原型为：

```
typedef void __fastcall (__closure *TFieldSetTextEvent)(TField* Sender,
                                const AnsiString Text);
__property TFieldSetTextEvent OnSetText = {read=FOnSetText, write
                                =FOnSetText};
```

这是和 OnGetText 事件相反的一个过程，当该事件没有指定处理函数句柄时，编辑后的 Text 将直接赋给 AsString 属性，可以在 OnGetText 和 OnSetText 事件中定义两个操作相反的过程，对需要保存的数据进行加密，可以参考本书所附的例子。

OnValidate

在数据被写入缓冲区之前触发，原型为：

```
typedef void __fastcall (__closure *TFieldNotifyEvent)(TField* Sender);
__property TFieldNotifyEvent OnValidate = {read=FOnValidate, write
                                                =FOnValidate};
```

该事件通常用于整体判断输入的数据是否有效，和 TField 的 EditMask 属性不同，EditMask 允许对用户输入的字符逐个进行判断。若在该事件中拒绝数据被写入缓冲区，则之后的 OnChange 事件将不会被触发，同时该事件会抛出一个异常。

2.4.3 TDataSet

TDataSet 是所有数据集组件的公共基类，它使用行和列反映了一个数据库中数据集的数据。TDataSet 封装了数据库的引擎，以及数据集所需要的属性、方法和事件。在 TDataSet 中，很多属性、事件和方法都是纯虚拟的，因此 TDataSet 是不允许被实例化的，通常使用的都是从 TDataSet 派生而来的数据集组件，比如 TTable、TQuery 等。

在单层结构的应用程序中，使用的是文件数据库，两层结构的应用程序使用的都是 SQL 数据库，在 C++Builder 中封装后的数据库组件可以消除这两种数据库之间的差别，使程序开发者不需要面对两种不同类型的数据库而编写不同的代码。这都是依赖于 TDataSet 的不同派生类来实现的。

在 TDataSet 的方法中有一些尽管不是纯虚拟函数，但是功能是不完整的，需要在其派生类中进一步来完善，如果开发者需要构建自己的数据集组件，则需要从 TDataSet 的某个派生类中派生新的组件，因此也有必要对 TDataSet 的基本特征有个大致的了解。

属性：

CanModify

一个 bool 类型的只读属性，用于表明 Dataset 是否允许写入数据，TDataSet 的

实例均是可写入的，而其派生类中 TQuery，TStoredProc 和 TTable 等都对这个属性进行了重载，会根据实际的情况来判定是否具有写入权限。

DataSource

一个 TDataSource 类型的只读属性，在 TDataSource 类中有一个 TDataSet 类型的属性，每一个 DataSource 必须为该属性赋予相应的值后才能被其它的数据库控件所使用。这是两个相互对应的属性，当为一个 TDataSource 对象指定一个 DataSet 时，这个 TDataSet 的 DataSource 属性就指向该 TDataSource 对象。

FieldCount

一个整型的只读属性，记录着 DataSet 组件中相关的字段数目。这个属性记录的字段指的是在 Fields 属性列出的字段的数量，对于 TDataSet 对象，Fields 属性中的字段有两种创建方式，一种是在设计时就已经根据需要创建的字段，这些字段的类型一个相关信息会被保存在 dmf 文件中，当程序运行的时候，Fields 属性是根据资源中的信息创建 TField 对象，因此设计好的程序，其 DataCount 是不会发生改变的；当一个 TDataSet 对象的 Fields 属性是动态创建字段的话，所有的字段信息不会在设计时保存到 dmf 文件，而是在 DataSet 被打开的时候，根据数据库的结构来创建相应的字段，因此，一个 TDataSet 对象打开不同的数据库表集的时候，其 Fields 和 FieldCount 都可能是不一样的。

Fields

一个 TFields 类型的只读属性，用于罗列 DataSet 中的所有非合计字段，TFields 组件包含了一个 TField 数组属性，用于存储所有列表的字段，TDataSet 通过 Fields 属性和 TField 对象建立一种逻辑包含关系，由于 TField 和 TDataSet 都是从 TComponent 派生的，因此具有流式存取能力，而且在设计时创建的 TField 对象会在头文件中生成相应的声明原型，因此，程序代码中可以直接通过与名称（Name 属性）相同的变量来访问 TField 对象，这一点和 TCollection 是完全不同的，尽管他们在 IDE 中的表现形式是完全一样的。

TFields 属性中包含的 TField 对象可以是在设计时创建，也可以是在运行时创建，设计时创建的 TField 对象，在运行时 Fields 中包含的 TField 对象将严格按照设计时的结构和类型生成相应的对象，运行时是不可改变的，但同时允许在设计时设置 TField 对象中的许多属性，诸如：Title、Color、Font、甚至是字段名称等等；而运行时创建则是 TDataSet 对象根据打开的数据集的结构自动的生成相应的字段 TField 对象，前面我们也说过了，任何数据库的最终操作都是通过 TField 来完成的，不管是设计时创建还是运行时创建，TField 对象都是绝对存在的，但是运行时创建的 TField 对象，其大部分属性象：DisplayName 等都是缺省的和数据集中的字段名称相同，不利于定制个性化界面或国际化应用程序，它的优点是可以通过一个 TDataSet 对象打开不同结构的数据集。

FieldValues

一个 Variant 类型的可读写属性，这实际上是为程序员提供了一个访问很多记录的途径，FieldValues 是一个数组属性，它的索引为 AnsiString 类型的变量，用于指定字段的名称，其值等于该字段当前记录的值。由于 FieldValues 使用了 Variant 类型，

因此在存取过程中程序员不必去仔细区分字段的数据类型，会由系统自动进行区分，但同时也带来的效率的降低等弊端，可以采用其它的访问方法来提高访问速度，比如使用 Fields 属性的字段列表直接对某个字段访问，并且指明数据的格式 AsString、AsCurrency、AsDate 等。

方法：

CreateBlobStream

原型为：
```
virtual Classes::TStream* __fastcall CreateBlobStream(TField*
                                    Field, TBlobStreamMode Mode);
```

提供了访问 Blob 数据的一种接口，对于大二进制数据的访问在 C++ Builder 中是通过 TBlobStream 对象来实现的，创建 BlobStream 具有两种方式，一种是使用 new 关键字，并在构造函数中指明相应的 Blob 字段，另一种是使用 Blob 字段的 CreateBlobStream 方法。

GetBlobFieldData

原型为：
```
virtual int __fastcall GetBlobFieldData(int FieldNo,
                                    TBlobByteData &Buffer);
```

这个方法从 Blob 字段中读取数据到一个动态数组的 Buffer 中，其返回值为实际读入字节数，FieldNo 指定了读取那个字段的内容。

Locate

原型为：
```
virtual bool __fastcall Locate(const AnsiString KeyFields,
                                const System::Variant &KeyValues,
                                TLocateOptions Options);
```

提供一个虚拟的记录定位的方法，按照提供的 KyFields 字段中值为 KyValues 的记录，其中 Option 为定位的匹配选项，在 C++ Builder 中有精确匹配和部分匹配等选项。该方法找到对应的记录时返回为 true，反之则返回 false，返回为 true 时，查找到的记录被激活为当前记录，Locate 更适合基于文件的数据库，而两层以上的数据库则更适合使用 SQL 语句进行定位或者查找匹配。

Lookup

原型为：
```
virtual System::Variant __fastcall Lookup(const AnsiString KeyFields, const
                    Variant &KeyValues, const AnsiString ResultFields);
```

提供一个虚拟函数用于查找给定字段的特定记录的值，与 Locate 相似，但返回的不是 bool 类型，而是指定字段的取值，KeyFields 指定用于匹配的字段，KeyValues 为该字段匹配的取值，ResultFields 为需要返回其值的字段。

事件：

在 TDataSet 的事件中，大部分是以 After 和 Before 开头的，这些事件有着非常明确的含义，根据事件的名称就可以了解其作用，读者可以参考帮助，这里不多讲。只对 On 开头的事件作个简单介绍。

OnCalcFields

在应用程序需要重新计算字段的值时触发。原型如下：

```
__property TDataSetNotifyEvent OnCalcFields = {read=FOnCalcFields,
                                               write=FOnCalcFields};
```

当 TDataSet 的 AutoCalcFields 属性为 true 时，该事件在以下情况下触发：

- TDataSet 被打开时；
- 一个 TDataSet 对象被设置为 dsEdit 状态时；
- 数据库相关的可视化控件的焦点转移时或者 DBGrid 的 Column 改变时；
- 一条记录从数据库中重新获取数据时；

当一个程序允许用户改变数据的时候，该事件可能会被频繁的触发，这时可能会导致程序的反映速度大为降低，可以将 AutoCalcFields 设置为 false 来降低触发的频率。当 DataSet 的 AutoCalcFields 为 false 时，一条记录内部的变化不会触发该事件。

OnDeleteError

当应用程序试图删除数据库的记录，并出现异常的时候触发。原型如下：

```
typedef void __fastcall (__closure *TDataSetErrorEvent)(TDataSet* DataSet,
                         EDatabaseError* E, TDataAction &Action);
__property TDataSetErrorEvent OnDeleteError = {read=FOnDeleteError,
                         write=FOnDeleteError};
```

在该事件的处理句柄中，Action 为处理的动作，当该事件第一次被触发的时候，Action 总是设置为 daFail，如果该事件的处理函数能够正确的解决故障，在函数的退出前，将 Action 设置为 daRetry，这时 delete 操作会重新尝试操作。如果不能正确处理故障，当 Action 为 daFail 时，程序不会发出错误的提示信息，如果希望程序出现提提示信息，可以通过将 Action 设置为 daAbort 来实现。

OnEditError

当一个 DataSet 编辑出错时被触发，原型为：

```
typedef void __fastcall (__closure *TDataSetErrorEvent)(TDataSet* DataSet,
                         EDatabaseError* E, TDataAction &Action);
__property TDataSetErrorEvent OnEditError = {read=FOnEditError,
                         write=FOnEditError};
```

在这个事件的处理函数中也有一个 Action 参数，其用法和作用和 OnDeleteError 事件中的 Action 相同。

OnPostError

当 Post 操作失败时触发，原型为：

```
typedef void __fastcall (__closure *TDataSetErrorEvent)(TDataSet* DataSet,
                         EDatabaseError* E, TDataAction &Action);
__property TDataSetErrorEvent OnPostError = {read=FOnPostError,
                         write=FOnPostError};
```

这个事件和 OnDeleteError 以及 OnEditError 类似，都是在出错时触发的，而且在处理函数中包含了一个 Action，它关系到该事件以后的执行动作，和 Delete 方法中的类似。

2.4.4 TControl

TControl 是所有控件的公共基类，所有的可视化组件都是从 TControl 派生而来的,前面我们也已经提到过,控件是具有用户交互操作能力的组件,同时可以响应 Windows 的消息，因此在 TControl 中需要提供控件的基本行为，包括 Windows 的消息处理能力。

属性：

MouseCapture

一个 bool 类型的可读写属性，用于表明控件是否捕获鼠标事件，当一个控件捕获鼠标的时候，之后发生的鼠标事件都会属于这个控件所有，直到该控件释放了鼠标。用户可以通过这个属性来限制某些操作。

Parent

一个 TWinControl 类型的可读写属性，用来表明一个控件的父窗体。在 Windows 程序中，通常每一个可视的窗体都会有一个父窗体作为容器，父窗体允许为空，为空时这个窗体是属于 Windows 桌面的，比如 Form 就是属于桌面。但通常的控件，包括图形控件和窗体控件都是属于某一个 Form 或者其它容器类控件（必须是从 TWinControl 派生而来）如：TPanel 等，大多数控件没有父窗体的话将不能够被正确的显示。

控件通过 Parent 属性和其父窗体建立了一种用于显示特性的逻辑包含关系，这种关系个接近于直观的表现，当一个控件的父窗体移动时，该控件也会跟随移动位置，父窗体的显示比例改变时，该控件的比例也会随之改变，甚至当控件的 ParentFont、ParentColor 等为 true 时，该控件的 Font、Color 等属性也会随着父窗体改变。但需要注意的是，这只是在显示特性上的逻辑包含，并不一定是“真正”的包含关系，在 VCL 控件的逻辑关系中有三个比较类似但是不同的逻辑包含关系，一个是通过 Parent 建立的显示特性的逻辑包含，一个是通过 Owner 建立的生存周期的逻辑包含关系，另一个是在 C++原文件中声明的位置包含关系，从语言的角度来看，只有原文件中定义的这种关系是实实在在的包含关系。通常在设计时创建的控件，头文件中声明的位置关系和通过 Owner 建立的逻辑关系是一致的，但在运行是创建的控件可以将其 Owner 指定为 NULL，正如前面所说的 Owner 为空的控件其生存周期是由 Application 所掌握的。

Parent 属性和 Controls 属性相对应，当一个控件指定了父窗体时，其父窗体的 Controls 属性中会增加一个元素。

Visible

一个 bool 类型的可读写属性，决定这个控件是否被显示在屏幕上，缺省的情况下控件都是被显示的，其显示的位置是以 Parent 为参考的相对坐标，当父窗体的 Visible 为 false 的时候，所有 Child 控件都不会被显示。

WindowProc

一个 TWndMethod 类型的可读写属性，但实际上该属性从各方面讲更象是一个事件，它为控件的消息处理提供了一个接口，TWndMethod 的原型如下：

```
typedef void __fastcall (__closure *TWndMethod)(Messages::Tmessage
                                                 &Message);
```

尽管 VCL 组件封装了大多数的 Windows 消息，而且进行了归类和加工处理，但是在

某些情况下，开发程序时仍然需要对一些底层的 Windows 消息进行处理，使用 WindowProc 属性可以非常方便是实现这个目的，但是 WindowProc 属性本身并不为空，它包含了一个控件 VCL 的基本消息处理能力，因此在设置该属性前需要将原来的 WindowProc 属性记录下来，并且在新的 WindowProc 属性所指向的 Windows 消息处理函数中调用原来的 WindowProc 句柄，其位置可以根据需要放在最前面或者是最后面，WindowProc 属性对应的消息处理方法为 WndProc。

对于组件的设计时，可以采用重载 WndProc 函数来实现新增加的消息处理过程，前面讲述 VCL 的消息处理机制时，也曾讲过，组件的消息处理可以通过消息映射宏定义来完成，而在 VCL 类库中，则更推荐使用重载 WinProc 方法来实现，在重载的 WinProc 方法中必须记着调用父类的 WinProc 方法，可以通过这种方式来实现对某一个特定的 Windows 消息是同时响应派生类和父类的 WinProc 处理或者只是响应派生类的 WinProc 处理过程，而在消息映射的机制里，我们也可以看出，被映射的 Windows 消息只能响应派生类的消息处理过程，除非在映射的消息处理函数中调用父类的 Dispatch 方法。

WindowText

一个字符串的可读写的属性，在一个控件中包含了三个比较类似的字符型属性，一个是 Caption、一个是 Text，另一个就是 WindowText，它们各有用途，Caption 通常更多的用于固定的标题等，Text 则更多的用于变化的输入或者输出等字符串等，WindopwText 则可以被重载为其它的用途。

在 Edit 控件中，WindowText 为输入的内容，在 ComboBox 中为组合 Edit 框中的内容，在 Button 中为控件的 Name，而在其它的控件中基本上是 Window 的 Title，这时 Windows 系统中对窗体对象的要求。

方法：

BringToFront

原型为：`void __fastcall BringToFront(void);`

其作用是将控件从底层提升到最顶层，这有以下几种情况，一是对 Form 窗体，该方法是有效的，BringToFront 可以改变窗体在屏幕上的叠放次序，但是对于 Form 的情况可能会复杂一些，如果屏幕上存在着其它的应用程序窗体；另一种情况是对于都是从 TWinControl 派生而来的控件，它们都是窗体控件，缺省的情况是控件按照创建的顺序自底向顶排列，即最后创建的窗体显示在最前面，可以通过 BringToFront 方法将排在底层的控件提升到最顶层，这个方法会改变控件在 Parent 数组中的位置；再一种情况是对于都是从 TGraphicControl 中派生而来的控件，它们不属于窗体控件，但是和前一种情况相同，排列次序是根据创建的先后而定的，BringToFront 可以改变它们的叠放次序；最后一种情况是分别由 TWinControl 和 TGraphicControl 派生控件，它们属于不同的控件分支，BringToFront 不起作用，因为所有的窗体控件都会显示在图形控件的上面。

Changed

原型为：`void __fastcall Changed(void);`

向控件发送一个 CM_CHANGED 消息。在属性发生变化以后调用 Changed 方法可以影

响到 Parent 对象，这样可以使其容器对象产生一些必要的调整动作。

ChangeScale

原型为：`DYNAMIC void __fastcall ChangeScale(int M, int D);`

用于调整控件的显示比例，M 为分子，D 为比例的分母，比如 ChangeScale(3,4)将会调整控件的大小为原来的 75%，这个方法在某些情况下是非常有用的，经常在网上听到网友为了界面在不同的桌面分辨率下显示不正常，使用比例放缩可以有效的解决这个问题。

ChangeScale 不仅改变了控件的长宽比例，同时也会改变定点的位置坐标，当一个窗体控件的比例改变时，其 Child 控件也会随之一并改变，关于这方面的内容这里不多讲，读者可以参考本书所附的例程。

Click

原型为：`DYNAMIC void __fastcall Click(void);`

触发 OnClick 事件，Click 最典型的在收到 WM_LBUTTONUP 时系统自动调用，同时 Click 方法会调用 OnClick 事件所关联的处理函数，通过对 Click 方法的重载，可以实现一个控件响应 WM_LBUTTONUP 的额外处理过程，有一些控件 Click 方法不光是在收到 WM_LBUTTONUP 消息时触发，而且会在收到某些于 WM_LBUTTONUP 同等作用的消息时触发，比如热键、加速键或者回车等。

ClientToScreen

原型为：

```
Windows::TPoint __fastcall ClientToScreen(const Windows::TPoint &Point);
```

是将控件中的相对位置坐标转换成屏幕的绝对坐标。

ConstrainedResize

原型为：

```
virtual void __fastcall ConstrainedResize(int &MinWidth, int &MinHeight,
                                          int &MaxWidth, int &MaxHeight);
```

触发一个 OnConstrainedResize 事件，和 Click 方法类似，可以被派生类重载。

DblClick

原型为：`DYNAMIC void __fastcall DblClick(void);`

触发 OnDblClick 事件，和 Click 类似，可以被派生类重载。WM_LBUTTONDBLCLK 消息会调用给方法，在 TControl 的 DblClick 方法中调用 OnDblClick 事件的处理句柄。

DefaultHandler

原型为：`virtual void __fastcall DefaultHandler(void *Message);`

是所有控件的缺省消息处理函数，这和 Windows SDK 编程的缺省处理过程比较类似的，可以通过重载这个函数来实现对一个控件的 Windows 消息做出额外处理，这和 WndProc 重载有这曲艺同工的作用。

MouseDown、MouseMove、MouseUp

原型分别为：

```
DYNAMIC void __fastcall MouseDown(TMouseButton Button, Classes::
```

```
                                    TShiftState Shift, int X, int Y);
DYNAMIC void __fastcall MouseMove(Classes::TShiftState Shift, int X,
                                    int Y);
DYNAMIC void __fastcall MouseUp(TMouseButton Button, Classes::
                                    TShiftState Shift, int X, int Y);
```

触发 OnMouseDown、OnMouseMove、OnMouseUp 事件，和 Click 类似，可以被派生类重载。

Notification

原型为：

```
virtual void __fastcall Notification(Classes::TComponent* AComponent,
                                        Classes::TOperation Operation);
```

这是为对象之间的通讯提供一种途径，Notification 可以将自身的创建、销毁等消息通报给 Owner，派生类可以通过重载这个方法来实现其它的一些通讯。当两个对象通过 Parent、Owner 等属性建立逻辑包含关系的时候，创建、销毁等动作通常都要相互交流，甚至在 DataSet 和 DataSource 等关系也是需要这种信息的交流，否则对象之间的关系或者是相互的约束关系是极不完整的，Notification 为这种关系提供了一种途径。

Perform

原型为：`int __fastcall Perform(Cardinal Msg, int WParam, int LParam);`

例行 Windows 消息的方法，在 VCL 中提供了许多消息通讯的方法和属性，当然也可以使用 Windows API 来向系统的消息队列发送某个消息，但这种情况比较复杂，需要经过 Windows 的处理，Perform 提供了一种途径，使程序需要向某个对象发送某个消息不经过 Windows 的消息队列，而直接使用 Perform 方法来响应。在 VCL 中定义了许多用户消息，这些消息是 Windows 中本身不包含的，因此通过 Perform 来响应消息要高效的多。

Perform 方法通常用在 VCL 内部的消息传递，比如通过 Parent、Owner 等属性建立的逻辑关系，象 ToolBar 和其中的 Button 之间就是使用了 Perform 传递一些消息的。

Refresh、Repaint

原型为：

```
void __fastcall Refresh(void);
virtual void __fastcall Repaint(void);
```

这两个方法的作用是等价实际上是等价的，将强迫控件重绘显示区域。当一个控件的 ControlStyle 属性包含 csOpaque 选项时，Repaint 直接绘制显示区域，反之则是通过 Invalidate 方法来重绘区域。

SendToBack

原型为：`void __fastcall SendToBack(void);`

将控件的叠放顺序改变到最底层，是和 BingToFront 相反的过程。当一个控件获得输入焦点的时候，SendToBack 方法会使其失去焦点。和 BingToFront 方法一样只能对都是窗体控件或者都是图形控件起作用。

Show

原型为：`void __fastcall Show(void);`

改变控件的可视属性，其作用和 `Visible = true;` 语句是相同的，在该方法中包含了对 Parent 对象 Visible 属性的检查工作。

VisibleChanging

原型为：`DYNAMIC void __fastcall VisibleChanging(void);`

提供了一个显示属性改变的响应接口，当一个控件的 Visible 属性需要改变时会自动调用该方法。这个发生在 Visible 改变之前，因此重载该方法，并且在该方法中增加条件的判断和异常行为的处理代码，可以阻止某些非法的显示属性的改变。

WndProc

原型为：`virtual void __fastcall WndProc(Messages::TMessage &Message);`

这实际上就是 WindowProc 属性所指向的处理句柄，它提供了一个控件的消息处理函数，同时这个方法也是虚拟函数，可以被派生类重载，在 TControl 的 WndProc 方法中，提供了基本的鼠标操作的响应，可以根据控件的 ControlStyle、DragMode 等属性来正确处理鼠标消息。

使用重载 WndProc 和重定向 WindowProc 属性基本上可以认为是一回事，但也稍有不同，如果不是用户自定义的类，则不能够对 WndProc 进行重载，因而只能使用重定向 WindowProc 属性。

前面也提到，使用 VCL 所提供的这些扩展的消息处理机制，不仅可以获得最底层的 Windows 消息的处理能力，还可以非常灵活的组织这些消息，可以使派生类具有父类表现特征的同时具有自己的个性。

事件：

在 TControl 的事件中，OnDragDrop、OnDragOver、OnEndDock、OnEndDrag、OnStartDock、OnStartDrag 等都是为控件的拖放而抽象出来的事件，这些是 Windows 本身不具备的，而且是比较一致的，在其派生类中关于拖放的属性、非法等基本都和 TControl 中的相同，对这几个事件我们不多作讲解。

OnCanResize

当试图改变一个控件的大小时触发，该事件在 OnConstrainedResize 事件之前触发，可以在该事件的处理函数中增加适当的代码来阻止随意改变控件的大小。

OnClick

这是一个抽象的事件，在不同的控件中触发的方式可能会有所不同，但有一个共同的触发条件就是单击鼠标的左键可以触发该事件，通常可以触发该事件的条件有：

- 在 Grid 中 OutLine 中选择一个单元或者项目时；
- 在 ComboBox 中按键盘的方向键时；
- 当一个控件获得焦点时按下空格键或者回车键；
- 在一个 Form 中定义了缺省确认按钮是按下了回车键会触发缺省按钮的 OnClick 事件，当一个 Form 中定义了缺省取消按钮时，按下 Esc 键会触发缺省取消键的 OnClick 事件。
- 用户按下了某个控件的热键或者加速键等，会触发相应控件的 OnClick 事件。

- 某些含有 Checked 属性的控件，Checked 属性改变时；
- MenuItem 的 Click 方法被调用时；

在一个 Form 中点击空白区域或者已经处于无效状态的控件，都会触发该 Form 的 OnClick 事件。

关于 OnClick 事件在不同的控件中都可以不一样，用户如果设计控件时，通常需要针对设计的控件来重载这个事件，以满足不同的应用需求。

OnConstrainedResize

当一个控件试图改变其大小的时候触发，在 OnCanResize 事件之后，OnResize 事件之前。每个控件可以定义其约束范围，并保存在 Constraints 属性中，如果需要重新定义约束范围则可以在该事件中设置 Constraints 属性中相关参数。

OnContextPopup

当用户鼠标的右键点击控件时触发，或者是通过其它的条件来弹出 PopupMenu 时触发（比如 Win95 键盘的 Popup 键），这个事件在控件没有指定 PopupMenu 时或者其 AutoPoup 属性设置为 false 的时候特别有用，可以让程序来集中管理加速菜单。

值得注意的是任何控件的 OnContextPopup 总是会触发其 Parent 的 OnContextPopup 事件，而且 Parent 的总是在前触发，因此如果没有在 Parent 中指定 PopupMenu，那么 Parent 的 OnContextPopup 事件可能会多次被触发。

直接使用这个事件的机会也许并不多，但是这个过程我们经常会遇到，比如在一个 Form 中定义了 PopupMenu，其多个 Child 控件并没有指定弹出菜单，那么在 Child 控件上点击右键都会弹出 Parent 所指定的 PopupMenu。

OnDblClick

响应控件的鼠标双击事件，这个事件不象 OnClick 那么丰富，通常只有 Windows 的双击消息才会触发，当然在用户自行设计的控件中也可以将其定义的更为丰富。

OnMouseDown、OnMouseMove、OnMouseUp

这几个事件都是鼠标事件，它们的原型分别如下：

```
typedef void __fastcall (__closure *TMouseEvent)(System::TObject* Sender,
          TMouseButton Button, Classes::TShiftState Shift, int X, int Y);
typedef void __fastcall (__closure *TMouseMoveEvent)(System::TObject
           * Sender, Classes::TShiftState Shift, int X, int Y);
__property TMouseEvent OnMouseDown = {read=FOnMouseDown,
                  write=FOnMouseDown};
__property TMouseMoveEvent OnMouseMove = {read=FOnMouseMove,
                  write=FOnMouseMove};
__property TMouseEvent OnMouseUp = {read=FOnMouseUp,
                  write=FOnMouseUp};
```

分别是在鼠标键被按下、释放和鼠标移动的时候触发，其中 X,Y 分别是鼠标的位置坐标，为 Sender 的 Client 区域的相对坐标，Shift 参数记录着鼠标事件发生的时候键盘的功能键的状态。

OnResize

在控件重新调整大小后触发，这个事件和前面提到的 OnConstrainedResize、OnCanResize 共同负责关于控件大小方面的消息处理。

2.5 TControl 分支

在 VCL 类库中，所有的控件都是从 TControl 中派生而来，在 TControl 的分支中，包含了两大类的控件，一类是从 TWinControl 派生而来，属于一种窗体控件；另一类是从 TGraphicControl 派生而来，属于图形控件。这两种控件的最大区别是窗体控件具有 Windows 窗体的所有特征：具有窗口句柄，可以接收键盘输入，能够获取焦点，也可以作为其它控件的父窗体及可以为其它控件的容器，而图形控件则没有窗口句柄，不能接收焦点，不可以作为其它控件的容器。但是图形控件的消息处理过程要比窗体控件简单的多，因此其响应速度和执行效率要相对高一些。在设计控件的时候，决定是从 TWinControl 派生还是从 TGraphicControl 派生完全取决于控件的需求,只有了解了它们的不同之处才能很好的使用它们。

2.5.1 TGraphicControl

TGraphicControl 是一个相对比较简单的控件，它是由 TControl 派生而来，除了增加了一个 TCanvas 属性和一个 Paint 方法之外，其它的内容和 TControl 完全一样。

属性：

Canvas

原型如下：

```
__property Graphics::TCanvas* Canvas = {read=FCanvas};
```

这是一个只读属性，Canvas 是由控件自身内部创建的，使用者不能自行改变，事实上 Canvas 画布是和控件相对应的固有属性，一旦控件创建，它的画布就是固定的。

在 C++ Builder 中，TCanvas 封装了几乎所有的 Winodws DC 资源和大多数绘图函数，如果用户不使用 OpenGL 或者 DirectX 等绘图函数时，TCanvas 可以完成 Windows 图形显示的所有功能。

对于图形控件，由于本身没有窗口句柄，而 TCanvas 需要一个窗口句柄来创建 Device Context，图形控件都是从其父窗口那里获取 DC 资源的。

方法：

Paint

原型为：`virtual void __fastcall Paint(void);`

提供了一个绘图的接口，所有的派生类都需要重载该方法来实现自身的图形显示。该方法在从 Parent 接收到 WM_PAINT 消息时自动被调用，通常派生类需要重载该方法。

2.5.2 TWinControl

TWinControl 具有标准 Windows 的特征，在 VCL 中还为其定义了许多用于界面显

示的属性、方法和事件，例如 Bevel 相关的属性可以使窗口的显示具有三维效果，Doc 相关的属性使控件可以自由的拖放停靠等。对这些特征我们不多讲解，读者需要翻阅相关资料或者察看联机帮助，我们仅对 TWinControl 中和消息传递，以及可能影响到对象之间相互约束关系的特征作个简单介绍。

属性：

Brush

一个 TBrush 类型的只读属性，用来记录控件的背景显示特征，如：颜色、风格等，由于 TWinControl 没有 Canvas 属性，因此需要用 Brush 来记录这些信息，通常不一定直接使用该属性，而是可以通过 Color 等属性间接的改变之，在某些 TWinControl 的派生类中，也定义了 Canvas 属性，对这些控件可以直接使用 Canvas 进行界面显示的操作。

ControlCount、Controls

这两个属性的原型为：

```
__property int ControlCount = {read=GetControlCount, nodefault};
__property TControl* Controls[int Index] = {read=GetControl};
```

都为只读属性，TWinControl 可以作为其它控件的容器，就是可以作为别的控件的 Parent 属性，Child 控件的 Parent 属性和 TWinControl 控件 Controls 属性共同建立了一种逻辑包含关系，这在 TControl 的 Parent 属性介绍中已经讲过。

ControlCount 记录这 Controls 属性中所包含的 Child 对象的数目，当一个控件被创建的时候，并将 Parent 设置后，该控件会使用 Notification 方法向 Parent 通报，Parent 控件则向 Controls 属性中添加一个控件列表，同时 ControlCount 会自动更新。

DefWndProc

原型如下：

```
__property void * DefWndProc = {read=FDefWndProc, write=FDefWndProc};
```

该属性指定了窗体控件的缺省消息处理函数。在 Windows 程序中，每个窗口都会有一个缺省的消息处理函数，用于标准事件的处理，DefWndProc 在 TWinControl 使用 API 函数 CreateWindow 创建窗体的时候，会调用 DefWndProc 来处理缺省的 Windows 消息，对于一些 WndProc 方法中没有处理的消息，可以通过 DefWndProc 来处理，

Handle

一个 HWND 类型的只读属性，原型如下：

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

记录着窗体控件的窗口句柄。这时和图形控件最大的一个区别，而且在 Windows 程序中，每个窗口都需要一个句柄来在操作系统中定位，大多数的 API 函数也都需要有一个窗口句柄作为参数来表明操作对象，VCL 尽管封装了很多 API 的功能，但在某些特殊操作可能使用 API 更加方便一些，Handle 属性为使用 API 函数提供了可能；另外，对于一般的窗体控件，需要有一个父窗体，包括一个程序的主窗口也是需要指定父窗体（通常为空），如果将一个应用程序或者 ActiveX 控件甚至是其它可执行模块窗体的 Parent 指定为另一个窗口控件的句柄，则可以将可执行模块的窗体嵌入到这个窗体控件中。

ParentWindow

一个和 Parent 非常类似的属性，原型如下：

```
__property HWND ParentWindow = {read=FParentWindow,
                                write=SetParentWindow, nodefault};
```

也是记录着窗体控件的父窗口句柄，和 Parent 不同的是 ParentWindow 通常是用于非 VCL 类型窗口作为控件的父窗体时，ParentWindow 对于 VCL 窗口作为父窗体也是有效的，这两个属性可以同时使用，但是当 Parent 属性非空时，ParentWindow 会被忽略。

如果编制多语言混合程序，需要注意的是，在一些共享的可执行模块中使用 VCL 控件时，不能使用 Parent 来确定控件的父窗体，除非已经明确这个窗口不会使用非 VCL 窗体作为父窗体，取而代之的是使用 ParentWindow 来传递相应的信息。

Showing

一个 bool 类型的只读属性。

Showing 本身是一个供内部使用的属性，用来判断控件是否被显示，通常是用来判断是否为控件分配用于缓冲的内存，比如显示图形资源等，Showing 和一个控件的 Visible 以及父窗体的 Visible 相关，甚至是父窗体的父窗体等等。在编制控件时代码使用 Showing 属性可以提高控件的性能。

WindowHandle

原型如下：

```
__property HWND WindowHandle = {read=FHandle, write=FHandle,
nodefault};
```

这和 Handle 属性非常相似，并且指向的是同一个存储内容，但不同的是 WindowHandle 是可读可写的属性，因此这个属性只是内部使用的受保护的属性，在编写控件时有可能会用到，另一个与 Handle 属性不同之处在于，Handle 对外提供了一个完整的接口，当 Handle 不存在时，读取 Handle 属性会使控件自动的创建窗口句柄，而 WindowHandle 则没有这个过程。

方法：

在 TWinControl 的方法中，有很多是窗口控件的共同特征，这些方法通常已经和相应的 Windows 消息或者经过处理加工的消息相关联，并且允许派生类重载，象 KeyDown、KeyPress、KeyUp 以及和 Mouse 操作相关的方法等，还有一些方法是触发某些事件的内部方法，派生类可以重载，象 DoEnter 等 Do 开头的方法，这和 TControl 中的 Click 等方法比较类似，鉴于篇幅有限，而且 TWinControl 中新增加的方法又特别多，因此对这些方法我们这里不再介绍，而且这些方法也是容易掌握的。

AlignControls

原型为：

```
virtual void __fastcall AlignControls(TControl* AControl,
                                      Windows::TRect &Rect);
```

重新排列窗体控件中容纳的控件。Child 控件的排列方式按照各自的 Align 属性的设置，Rect 为需要重新排列的区域，Acontrol 为指定的 Parent 容器，可以为空。

Broadcast

原型为：`void __fastcall Broadcast(void *Message);`

向容器中所有的 Child **控件发送一个消息。这个方法在** TWinControl **中非常重要，通常一些重要的信息都是通过** Broadcast **来以广播的方式向** Child **控件发送的。在一个容器控件中，往往会容纳了很多具有一定约束关系的** Child **控件，比如** ToolBar **中的** SpeedButton **或者** ToolButton **等，它们之间是通过** Parent **来建立相互之间的关系，这些关系在表现形式上最终是以属性来实现的，但内部的消息传递是通过一系列的方法来实现的。**

Broadcast **方法很少用在程序设计时，但在编写控件时却可能会经常使用到。**

ContainsControl

原型为：`bool __fastcall ContainsControl(TControl* Control);`

用来判断一个控件是否在容器控件之内，这和 Controls **属性中列表的内容有所不同，**Controls **列表中之包含了自身的** Child **控件，如果** Child **控件也是一个** TWinControl **控件，并且也包含了其它的** Child **控件，那么使用** Controls **属性来判断就和** ContainsControl **方法判断的结构不一致，**ContainsControl **不管是自身的** Child **控件，还是** Child **的** Child **控件，都可以正确的区分是否包含在窗体之内，并且没有层次深度的限制。**

ControlAtPos

原型为：

```
TControl* __fastcall ControlAtPos(const Windows::TPoint &Pos,
                                  bool AllowDisabled, bool AllowWinControls);
```

用来返回指定位置的控件，这个方法只检查自身的 Child **控件，两个参数分别决定是否检查无效的控件和是否过滤** TWinControl **的派生类。**

CreateParams

原型为：`virtual void __fastcall CreateParams(TCreateParams &Params);`

用于创建窗口时所需要的参数。每个 Windows **窗口在使用** API **函数** CreateWindowEx**或者**CreateWindow**是都需要指定一个**Parameter**结构来存储窗口的风格、设置等信息，在** VCL **中** TWinControl **的** Parameter **是设计控件时指定的，或者由其它的一些属性来决定的，该方法可以用来额外的为窗口指定相关参数，本书例子** PickEdit **中就通过了重载** CreatParams **方法为** PickEdit **限定了字符的显示区域。**

CreateParentedControl

原型为：

```
static TWinControl* __fastcall CreateParentedControl(TMetaClass* vmt,
                                                      HWND ParentWindow);
```

用于将 VCL **的** TWinControl **控件插入一个非** VCL **风格的窗口中，这个方法并不是创建一个父窗体，而是重新创建一个** vmt **指定的对象，并将新创建的窗口的父窗口指定为** ParentWindow **参数指定的窗体。**

GetDeviceContext

原型为：`virtual HDC __fastcall GetDeviceContext(HWND &WindowHandle);`

提供了一个获取控件 Device Context 的接口,尽管在 VCL 中可以使用 WindowsAPI 函数 GetDC 获取 DC 资源，但建议使用该方法会更好一些，因为在 VCL 封装的过程已经考虑了获取资源失败的情况，而使用 GetDC 则需要判断函数是否执行成功。

另外这个方法是虚拟的，他允许设计控件的时候重载，这样可以更加灵活、方便的使用 VCL 控件，而不必担心在使用 GetDC 的时候是否传递了正确的参数（在某些控件中，画布并不是自身的 Context DC，而是其中一个 Child 控件的 DC，设计控件时可以通过重载 GetDeviceContext 方法得到纠正）。

GetParentHandle

原型为：`HWND __fastcall GetParentHandle(void);`

返回父窗体的窗口句柄，完全使用 VCL 控件的程序可以使用 Parent 属性来获取父窗体的句柄，但是父窗体不是 VCl 风格的对象时，Parent 属性是空的，这样会得不到正确的结果，使用 GetParentHandle 可以使程序员不必关系父窗体是否为 VCL 对象。

IsControlMouseMsg

原型为：`bool __fastcall IsControlMouseMsg(Messages::TWMMouse &Message);`

在 Windows 系统中，所有的系统消息是发送给回调函数的，更多的是发送给窗口的回调函数，TWinControl 的派生类具有窗口的所有特征，因此可以接收 Windows 的消息，而从 TGraphicControl 派生来的的控件，并不具备这种能力，可以认为图形控件只是 VCL 对象在 Parent 的画布上绘制的一个形状，并不是窗口，所有关于图形控件的鼠标消息都是发送给了 Parent，Parent 则需要判断并正确的将鼠标消息发送给每一个 Child 图形控件，这就不难理解 TWinControl 为什么有一个 ControlAtPos 方法，而且这个方法具有过滤无效控件和窗体控件的能力。

NotifyControls

原型为：`void __fastcall NotifyControls(Word Msg);`

用于向所有 Child 控件发送一个消息，和 Broadcast 类似，但是要简单一些，对于一些参数为空的消息,使用 NotifyControls 要比 Broadcast 方便的多,BroadCast 需要使用一个完整的 Windows 消息结构作为参数，而 NotifyControls 只需要指明消息类型，因此 NotifyControls 不适合用于带参数的消息。

PaintControls

原型为：`void __fastcall PaintControls(HDC DC, TControl* First);`

在指定的 Debice Context 上绘制 Child 控件，DC 指定需要绘制的画布，First 指定从 Controls 列表中的那一个 Child 控件开始画起，当 First 为 NULL，所有的 Child 控件都会被绘制。

这个方法对于控件组合复杂又可能经常刷新的情况非常有用，可以为 Parent 控件创建一个内存的缓冲区，然后重载相关的绘图函数（如 PaintWindow），在绘图函数中先将 Child 控件绘制到缓冲区然后在拷贝到 Parent 控件的画布上,可以有效的缓解显示闪烁的问题，读者可以参考本书的例子 DataPick 控件。

PaintWindow

原型为：`virtual void __fastcall PaintWindow(HDC DC);`

用来绘制控件，类似 TGraphicControl 的 Paint 方法，该方法向

DefaultHandler 传递一个 WM_PAINT 消息，并将消息的 WPara 设置为参数 DC，其它参数和 Result 设置为 0。

事件：

OnEnter

当控件获得了输入焦点的时候触发，原型如下：

```
__property Classes::TNotifyEvent OnEnter = {read=FOnEnter, write
                                    =FOnEnter};
```

在两个 Form 之间切换的时候，不会触发该事件，在两个应用程序之间切换时也不会触发该事件。

当一个容器控件中的 Child 控件获得焦点的时候，它的父窗体也会触发该事件，而且是在 Child 控件的 OnEnter 之前触发。

OnExit

当控件失去焦点的时候触发，和 OnEnter 是相反的，原型如下：

```
__property Classes::TNotifyEvent OnExit = {read=FOnExit, write=FOnExit};
```

这个事件在 Form 之间切换的时候不会触发，在两个应用程序之间切换的时候也不会触发，和 OnEnter 相反，Child 控件总是先一步于 Parent 控件失去焦点。

另一个需要注意的是 Form 中的 ActiveControl 的更新是在 OnExit 事件触发之前进行的。

OnKeyDown、OnKeyPress、OnKeyUp

当一个控件获得输入焦点的时候，所有的键盘消息会被分发到该控件，这三个事件是响应键盘消息的事件，原型分别如下：

```
typedef void __fastcall (__closure *TKeyEvent)(System::TObject* Sender,
                          Word &Key, Classes::TShiftState Shift);
typedef void __fastcall (__closure *TKeyPressEvent)(System::TObject
                          * Sender, char &Key);
__property TKeyEvent OnKeyDown = {read=FOnKeyDown, write=FOnKeyDown};
__property TKeyPressEvent OnKeyPress = {read=FOnKeyPress,
                         write=FOnKeyPress};
__property TKeyEvent OnKeyUp = {read=FOnKeyUp, write=FOnKeyUp};
```

OnKeyDown 事件键盘按下任意键时触发，相当于 WM_KEDOWN 消息触发，OnKeyUP 则相当于 WM_KEYUP 消息触发。这两个事件的触发是在 IME 输入法接收到消息之前产生的，是和键盘消息对应的。输入法不会对其产生任何影响。

OnKeyPress 是在键盘按下一个字符键的时候触发，这个事件不能区分组合键、功能键等。和上面两个事件不同，OnKeyPress 是在 IME 输入法之后收到消息的，因此当输入法打开的时候，OnKeyPress 不会收到键盘的击键内容，取而代之的是输入法的消息。

OnMouseWheel、OnMouseWheelDown、OnMouseWheelUp

这三个事件在鼠标的滚轮被滚动的时候触发，原型分别如下：

```
__property TMouseWheelEvent OnMouseWheel = {read=FOnMouseWheel,
                            write=FOnMouseWheel};
```

```
__property TMouseWheelUpDownEvent OnMouseWheelDown = {read=
                                FOnMouseWheelDown,write=FOnMouseWheelDown};
__property TMouseWheelUpDownEvent OnMouseWheelUp = {read=FOnMouseWheelUp,
                                write=FOnMouseWheelUp};
```

和上面的三个事件相似，OnMouseWheel 在滚轮滚动时触发，OnMouseWheelDown 在滚轮向下滚动的时候触发，OnMouseWheelUp 在滚轮向上滚动的时候触发。

本节所讲的这些类库都属于基类，不会在实际中直接应用，有些甚至是抽象基类，不能被实例化，但这些基类定义了其分支的共同特性，安装在组件面板中的组件都是从这些类中派生而来的。

对于可以直接使用的或者能够被实例化的类，我们这里不再介绍了，它们都具有其祖先的形状和使用特征，只有一些细微的差别，比如 TStream 类不能被实例化，但在组件和程序中往往都是以引用 TStream 的方式是来操作和访问其子孙，这也是面向对象的一个特点，TStream 所提供的属性和方法对大多数的应用都是能够满足的，只有极少数情况，需要引用派生类本身对象的类型才能完成，比如直接访问 TMemoryStream 的内存区就只能通过对 TMemoryStream 类操作来完成。

VCL 的结构我们只作一个简单介绍，需要进一步了解的读者请参考相关书籍。

3 C++ Builder 的资源

在 Windows 程序设计中，资源是用户界面不可缺少的，所谓资源就是一种数据，供程序运行时使用的数据。资源通常存在于可执行文件的内部，是程序的一部分，当程序需要向用户界面输出，程序会调用相应的 API 函数将提取资源并显示在屏幕或者输出在相应的设备上。用户可以为应用程序设计任何类型的资源，但除了标准的 Windows 资源以外，其他类型的资源都可能需要使用用户自行设计的资源处理代码。

在使用 Windows SDK 设计程序时，资源和代码是相对独立的，使用和设计都很不方便，因此在 C++语言中，大多数的 FrameWorks 都将对象和资源进行了不同程度的封装，包括 MFC、OWL 以及 C++ Builder 中 VCL。在这些 FrameWorks 中 VCL 是将资源最彻底的封装的一种类库，也是使用最为方便的一种，也是最不遵循 SDK 标准的资源类型，实际上用户使用 C++ Builder 设计的程序，大部分资源是不能使用 Windows SDK 进行初始化的，但这些工作都是由 VCL 的代码自行完成的，这是对资源的更高层次的应用，同时也带来了相应的副作用，一般情况 C++ Builder 编制的可执行程序体积都比使用其他 FrameWorks 编制的相同功能的程序大得多。这并不是一个主要的问题，因为目前的硬件发展速度远远大于软件的发展速度，特别注重程序运行性能的时代已经成为过去，大多数应用程序都将更多的注重于程序的稳定性、可靠性、易维护性以及可扩展性，C++Builder 的 VCL 正是为了适应这种需求而产生的。

尽管对于大多数程序而言，已经不需要单独的使用资源编辑器进行资源设计，因为 C++ Builder 中会自动根据界面的设计而生成其专用的资源，但如果你是一个组件开发

者、如果你希望编制功能更加强大的程序或者你希望更多的了解 C++ Builder，那么还是有必要了解 Windows 的历史，下面将对 Windows 的资源进行简单描述，需要更深的了解请参阅相关书籍！

3.1 Windows 资源

这里所说的 Windows 资源，是针对 C 语言的 SDK 而言的，是没有涉及到面向对象的设计方法，MFC、OWL 中资源的使用已经有所不同，但不是本书要讲的内容，假如你想深入学习并使用 C++ Builder，也就没有必要对 MFC 和 OWL 进行深入的了解。

首先我们来看一下 Windows 的资源类型，确切的将是 Win32 的资源类型，在 Windows 95 和 WindwosNT 4.0 之前的操作系统，使用的是 16 位代码，资源稍有区别，而且 16 位的操作系统已经步入历史的红尘，有兴趣的读者可以查阅早期的 Windows 参考书籍。

3.1.1 Windows 资源的类型

在 Win32 SDK 中，提供了以下类型的 Windows 资源类型：

Accelerator:加速键，俗称热键，通常在程序中使用组合键，如经常使用的 Ctrl+C 为拷贝、Ctrl+V 为粘贴等都属于加速键，使用加速键的目的是将一些常用的操作和相对简单的按键组合对应起来，这里所指的加速键都是应用程序内部的，与 Windows 加速键有所区别，也就是只有一个应用程序处于激活状态时，这些加速键才在应用程序中起作用，而 Windows 热键需要使用 Windows API 向系统注册，无论什么情况下只要系统热键注册成功，都是有效的。加速键通常可以使用资源编辑器或者文本编辑器来编辑。

Bitmap:位图，这是 Windows 中应用比较多的一种资源，实际上就是 bmp 图片，大小没有限制，可以是 4 色、16 色、256 色以及 16 位、真彩等，但在程序中为了减少资源占用，通常使用 256 色以下的图像，对于 16 位色、真彩等高质量的图像可以采用工具软件进行抖动处理转换成 256 色位图。位图资源可以使用资源编辑器、图像工具、文本编辑器等来编辑。

Cursor：光标，是一种特殊的位图，大小限制为 32 x 32 像素，用于屏幕上显示鼠标的位置，可以使用图像编辑器或者资源编辑器来编辑光标资源。

Dialog：对话框，是一种用于建立特定对话框窗体的资源通常需要使用资源编辑器进行创建和编辑。

DialogEx：扩展对话框，同 Dialog 资源，但是属于 Windows 95 及以后版本的 Widnows 资源，支持 Win95 扩展界面，使用对话框编辑器创建和编辑。

Icon：图标，是一种较小的位图文件通常也使用简单颜色，大小限制为 16 x 16、32 x 32、48 x 48、64 x 64 像素，使用图像编辑器创建和编辑图标。

Menu：菜单资源，Windows 中菜单资源也是应用最为广泛的资源，基本上是不可缺少的资源类型，可以使用文本编辑器或者资源编辑器创建和编辑一个菜单资源。

MenuEx：扩展菜单资源，同 Menu 资源，也是用于 Win32 的资源类型，使用资源编辑器或者文本编辑器创建和编辑。

RCDATA：RCDATA 为 Windows 程序提供了一种扩展资源类型，RCDATA 可以是任何类型的资源，比如声音、动画等都可以作为 RCDATA 类型的资源被加入可执行程序或者动态链接库等。通常需要使用文本编辑器或资源编辑器编制资源脚本（RC 文件），用以包含最终的资源。

Stringtable:字符串表，在一个程序中，通常会有一些特定的字符串用于显示错误信息、警告、提示等，这些信息的显示模式是类似的，但是内容随不同的要求在变，这时候可以使用 StringTable 资源，将常用的字符串保存在可执行文件内部。字符串资源使用文本编辑器或者资源编辑器创建和编辑。

VersionInfo:版本信息，版本信息对于 Windows 程序而言，不像 Linux 中要求的那样严谨，但是良好的习惯可以为程序开发以及维护带来方便，有其在共享代码的程序中，版本信息可以提供给调用模块和被调用模块之间一个良好的确认关系，可以让程序有能力处理不同版本之间的代码差异。

其中RCDATA是Windows不提供处理代码的，Bitmap通常也需要用户干预，Dialog是一个交互式的界面元素，Windows 负责将其按照用户预先设计的形状显示于桌面，但其各个元素的操作需要和程序的代码对应，StringTable，需要代码根据实际情况自行提取。其他的资源类型基本都是由 Windows 完成处理的，程序只要告诉系统，使用什么资源就行了。

除了以上的类型以外，实际上 Windows 支持用户自定义的资源类型，尽管 RCDATA 可以支持任意类型的资源，但是 RCDATA 类型并没有一个明确的含义，根据其名称并不能判断是属于什么样的资源，比如程序需要加载一个 Jpg 文件作为资源，可以使用以下脚本（存在与.RC 文件）

```
BackGround RCDATA ".\BackGround.jpg"
```

在这个脚本中定义了一个名称为 BackGround 的资源，程序可以通过其名称对其进行访问，类型为 RCDATA（用户的未知类型），该资源的实体为当前目录的“BackGround.jpg”文件，这是正确的，但是不明确。如果使用以下脚本，则会更加明确：

```
BackGround JPEG ".\BackGround.jpg"
```

在这个脚本中，与前一个不同的是资源的类型为 JPEG 类型，这是 Windows 没有提供的类型，但在程序中是允许的，类型为 JPEG，可以明确的指定该资源的数据类型是 Jpeg 压缩图像。这对于程序设计的规范化而言是非常有利的。

3.1.2 Windows 资源的初始化

前面将 Windows 的资源类型进行了简单的说明，加下来将就 Windows 资源的初始化，即 Windows 如何使用这些资源进行简单的介绍。我们先看一个 Windows 标准的资

源文件。

下面是一个典型的资源文件，这是使用 Borland C++的资源编辑器创建的资源脚本文件。名称为 sample.RC，其中包含了三个资源，Menu、Dialog 和 BitMap，如下：

```
#define IDB_BITMAP1 1
#define IDM_MENU1   1
#define IDD_DIALOG2 2
#define IDC_EDIT1   102
//以下是 Dialog 的内容
IDD_DIALOG2 DIALOG 0, 0, 240, 150
STYLE DS_MODALFRAME | DS_3DLOOK | DS_CONTEXTHELP | WS_POPUP | WS_VISIBLE
                    | WS_CAPTION | WS_SYSMENU
CAPTION ""
FONT 8, "MS Sans Serif"
{
 CONTROL "OK", IDOK, "BUTTON", BS_PUSHBUTTON | BS_CENTER | WS_CHILD
                     | WS_VISIBLE | WS_TABSTOP, 184, 48, 50, 15
 CONTROL "Cancel", IDCANCEL, "BUTTON", BS_PUSHBUTTON | BS_CENTER | WS_CHILD
                     | WS_VISIBLE | WS_TABSTOP, 184, 80, 50, 14
CONTROL "Edit1", IDC_EDIT1, "edit", ES_LEFT | WS_CHILD | WS_VISIBLE | WS_BORDER
                     | WS_TABSTOP, 8, 4, 60, 16
 CONTROL "TabControl1", IDC_TABCONTROL1, "SysTabControl32", WS_CHILD
                     | WS_VISIBLE, 8, 24, 168, 116
}
//以下是 Menu 的内容
IDM_MENU1 MENU
{
  POPUP "&File"
  {
    MENUITEM "&New", 101
  // ......   ......   ......
    MENUITEM SEPARATOR
    MENUITEM "E&xit", 108
  }
  POPUP "&Help"
  {
    MENUITEM "&Help Topics", 901
    MENUITEM "&What's This?", 902
    MENUITEM SEPARATOR
```

```
    MENUITEM "&About", 903
  }
}
//以下是 BitMap 的内容
IDB_BITMAP1 BITMAP
{
 '42 4D 76 06 00 00 00 00 00 00 36 04 00 00 28 00'
 '00 00 18 00 00 00 18 00 00 00 01 00 08 00 00 00'
  // ……  ……  …… 位图数据
 'FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF'
 'FF FF FF FF FF FF'
}
```

资源脚本文件（RC 文件）同其他源文件一样，可以在其中定义一些标识符，因为 Windows 需要对每一个资源分配一个唯一的标识号，用以区分各个资源，上面例子中的：

```
#define IDB_BITMAP1     1
#define IDM_MENU1       1
#define IDD_DIALOG2     2
#define IDC_EDIT1       102
```

就是用来定义资源标识的，尽管 IDB_BITMAP1 和 IDM_MENU1 都是 1，但它们属于不同的资源类型，因此上在系统中也是唯一标识的。这实际上就是为资源定义了一个名称，也可以不这样去做，像前面的例子：

```
BackGround JPEG ".\BackGround.jpg"
```

就没有使用定义的标识，而直接使用了资源的名称 BackGround，这对于程序的执行没有任何影响，唯一的区别是，编译后的资源名称有所不同，使用定义标识符，编译后的资源名称将会用较为简短的文字取代有固定含义的长字符串，而直接使用名称的资源编译后其名称将会原封不动的被保存在可执行文件中。在早期的 Windows 程序中对一个可执行文件的大小还比较关注，所以通常采用预先定义标识符，而在 C++ Builder 中，一般不提倡预定义的名称，因为资源往往是和对象联系在一起的，一个具有特定含义的名称不仅方便程序设计，有时候还必须的，比如注册一个自行设计的组件，并且在组件面板中要为其指定一个相应的图标，这时候就需要一个位图资源，而且这个资源的名称必须和组件的名称一致，编译后才可能被 IDE 正确的加载到组件面板。

对于资源的定义一般是：

资源名称 ＋ 资源类型 ＋ 资源的数据

关于资源的名称和类型前面已经说过，资源的数据根据资源类型的不同也有所差异，通常可以使用 {} 来表示数据块的起始和结束。在 Sample.RC 中：

```
IDD_DIALOG2 DIALOG 0, 0, 240, 150
```

定义了一个对话框资源，名称为 IDD_DIALOG2 实际编译后的名称为 2，后面为这个对话框的基本数据，左上角为（0，0），右下角为（240，150）；

```
STYLE DS_MODALFRAME | DS_3DLOOK | DS_CONTEXTHELP | WS_POPUP | WS_VISIBLE
                    | WS_CAPTION | WS_SYSMENU
CAPTION ""
FONT 8, "MS Sans Serif"
```

这一部分定义了对话框的风格；从这里可以看出，这个对话框中的基本数据其实就是使用 Windows API 函数 CreatWindow 创建一个窗体时的基本数据，当然创建窗体的基本数据并不止这些，没有设置的数据均会按照缺省值进行创建。

到这里读者也许会明白资源和程序的关系，资源就是提供给程序使用的数据。一个简单窗体，可以使用对话框资源来创建，也完全可以在程序中使用代码进行创建，但使用资源显然要灵活的多，修改、增加也相对容易直观的多。

Dialog 后面{}中的数据为扩展的数据块，它们不属于窗体本身的数据，而是在基本窗体中增加的控件，这个例子中包含了两个按钮（Button）、一个编辑框（Edit）和一个 TabControl1 控件，对话框创建后如图 4.2 所示：

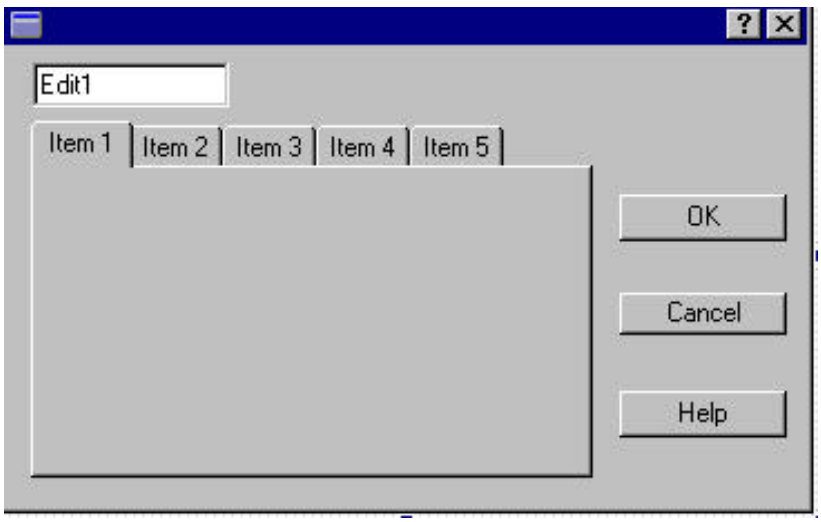


图 4.2 对话框资源的显示结果

当程序中需要创建这个对话框时，会调用相应的系统函数创建这个窗体，并将{}中包含的控件使用相应的API函数创建于这个窗体中，同时系统需要作的还有将这个窗体或者控件的回调函数向系统注册，以确保该窗体的 Windows 消息会被正确的处理函数接收到。

在 Sample.RC 中 IDM_MENU1 MENU 表示定义一个 Menu 资源，名称为 IDM_MENU1，{}的数据表示菜单项的内容，和 Dialog 类似，当程序需要创建或者使用该资源的时侯，会自动的调用相应的 API 函数去创建菜单，并将菜单发生的消息发送给相应的处理函数，与 Dialog 不同的是每一个菜单项都会有一个标识符：

```
MENUITEM "&New", 101
```

中的 101 就是一个唯一的标识，用于在回调函数中区分属于那个菜单项的消息。关于 Windows 消息的详细内容参阅相关书籍，这里不多赘述。这个例子中的菜单显示结果如图 4.3：

在 Sample.RC 中另一个资源为 BitMap 类型：

```
IDB_BITMAP1 BITMAP
```

表示定义了一个位图资源，其名称为：IDB_BITMAP1，实际编译后的名称为：1，Bitmap 的所有数据都在其后{}中，包括位图的大小、颜色深度以及像素的各种信息等。在这里纯文本方式来观察 Sample.RC 很不直观，也可以使用另一种方法，将资源数据指向到另外的文件中，比如：

```
IDB_BITMAP1 BITMAP "Bmp1.bmp"
```



图4.3 菜单资源的显示效果

这种文件指向可以是其他的 RC 脚本文件，其他的二进制资源文件(Res 文件)或者和资源相对应类型的文件，上面的 IDB_BITMAP1 就对应了一个 Bitmap 文件 Bmp1.bmp。

对于位图资源的使用往往不是由系统自行完成的，可能会和用户的代码有关，这里不作例子，读者可以参考 SDK 或者后面关于 C++ Builder 中使用资源的例子。

简单的总结一下，Windows 对资源的初始化可以使用下图 4.4 说明：

对于大部分资源，如同图中显示的，是由 Windows API 来完成资源的载入、创建、显示等等，应用程序仅仅是告诉系统下来该使用那个资源，甚至有些过程是在 Windows API 和 User Interface 之间就完成的，而应用程序并不参与，比如在 Windows 95 以后弹出菜单会跟随鼠标移动，而应用程序并不知道在什么时候某个菜单项会处于什么状态，这些都是 Windows 自行完成的（实际上是 Windows 的缺省消息处理函数进行处理的），应用程序只需要接收菜单所发出的消息并处理就行了。

对于不是 Windows 标准的资源，使用时并没有相应的 API 进行处理，这时候，通常需要应用程序自行处理，而需要 Windows API 所作的仅仅是从程序中提取资源，并交给应用程序。C++ Builder 中使用的资源大多数属于这种情况。

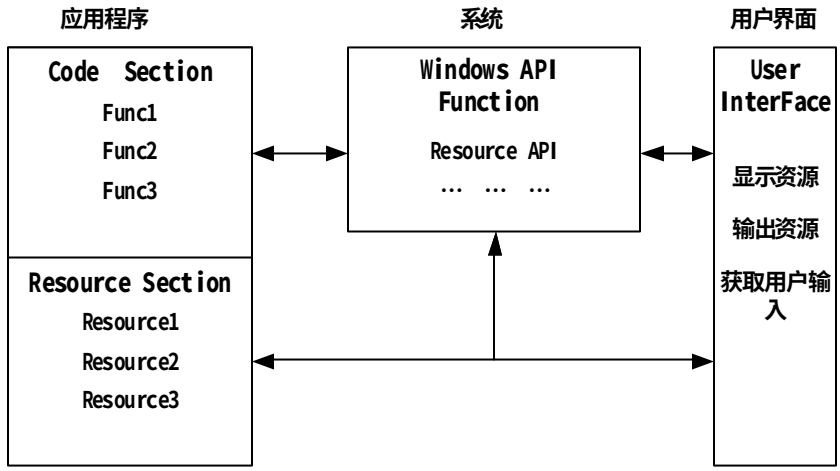


图 4.4 Windows 中对资源的使用

3.2 C++ Builder 资源

在 C++ Builder 设计的 Windows 应用程序中，资源仍然是不可缺少的部分，也正如前面所说的，VCL 是不同于其它以前的 OWL、FMC 等 FrameWorks 的，在资源的使用上也有很大的不同，同时 C++ Builder 也是完全兼容 MFC 和 OWL 的，也完全支持 C 语言 SDK 中对资源定义的规范。

这一节我们简单的讲述一下 C++ Builder 中资源的一些特点，有助于读者更好的理解 VCL 的运行机制，也能更好的使用 C++ Builder。

3.2.1 解读 dfm 文件

首先我们来看一个 C++ Builder 中的资源文件，这个文件通常不需要程序员手工创建，而是在 C++ Builder 的 IDE 中自动生成的，这就是 dfm 文件。

下面的文件为一个典型的 Form 的资源（dfm 文件），在这个 Form 中，包含了一个 ToolBar、三个 ToolButton、一个 StatusBar、一个 ImageList 和一个 MainMenu，dfm 文件的字符形式如下：

```
object Form1: TForm1
  Left = 351
  Top = 131
  Width = 396
  Height = 260
  Caption = 'Form1'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET     //属性为对象 Font，直接包含的
```

```
Font.Color = clWindowText
Font.Height = -11
Font.Name = 'MS Sans Serif'
Font.Style = []
Menu = MainMenu1
OldCreateOrder = False
PixelsPerInch = 96
TextHeight = 13
object ToolBar1: TToolBar     //包含对象，使用 Parent 建立逻辑包含，
                              //并且头文件中有对象声明
  Left = 0
  Top = 0
  Width = 388
  Height = 29
  Caption = 'ToolBar1'
  EdgeBorders = [ebTop, ebBottom]
  Flat = True
  Images = ImageList1
  TabOrder = 0
  object ToolButton1: TToolButton  //逻辑包含，Parent 为 ToolBar
    Left = 0
    Top = 0
    Caption = 'ToolButton1'
    ImageIndex = 0
  end
  object ToolButton2: TToolButton
    Left = 23
    Top = 0
    Caption = 'ToolButton2'
    ImageIndex = 1
  end
  object ToolButton3: TToolButton
    Left = 46
    Top = 0
    Caption = 'ToolButton3'
    ImageIndex = 2
  end
end
object StatusBar1: TStatusBar
```

```
  Left = 0
  Top = 195
  Width = 388
  Height = 19
  Panels = <>
  SimplePanel = False
end
object ImageList1: TImageList
  Left = 240
  Top = 32
  Bitmap = {
    494C010103000400040010001000FFFFFFFFFF00FFFFFFFFFFFFFFFF424D3600
    0000000000000000000000000FFFFFF00FFFF882380000000F83FC00755550000
      //…….
        //位图文件
      //…….
    3C190001E6F200009C0B0001E6F200008C438003FEFE0000C467C6C700000000
    E00FFC7F55550000F83FFFFF00000000}
end
object MainMenu1: TMainMenu
  Left = 208                //Left 和 Top 属性只在设计时使用
  Top = 32
  object File1: TMenuItem   //MainMenu 的主菜单项
    Caption = 'File'
    object New1: TMenuItem  //File 菜单项的子菜单项，其 Parent 为 File 菜单项
      Caption = 'New'
    end
    object N2: TMenuItem
      Caption = '-'
    end
    object Open1: TMenuItem
      Caption = 'Open'
    end
    object Close1: TMenuItem
      Caption = 'Save'
    end
    object SaveAs1: TMenuItem
      Caption = 'Save As'
    end
```

```
        object Close2: TMenuItem
          Caption = 'Close'
        end
        object N1: TMenuItem
          Caption = '-'
        end
        object Exit1: TMenuItem
          Caption = 'Exit'
        end
      end
      object Edit1: TMenuItem
        Caption = 'Edit'
        object Cut1: TMenuItem
          Caption = 'Cut'
        end
        object Delete1: TMenuItem
          Caption = 'Delete'
        end
        object Copy1: TMenuItem
          Caption = 'Copy'
        end
        object Pastl1: TMenuItem
          Caption = 'Paste'
        end
      end
    end
  end
```

从这个文件可以看到 dfm 和 Windows 资源脚本文件非常相似，但也有很大的区别，看起来 dfm 文件倒更象一个 ini 文件的格式，在 C++ Builder 中正是依靠这种格式来记录一个 Form 的所有信息。

面前也说过，C++ Builder 中，资源都是以对象为单位的，一个资源实际上就是一个对象的信息，上面的 dfm 文件在 C++ Builder 的 IDE 中的显示形状如图 4.5：

运行后的形状和图 4.5 中的 Form 是一致的，除了 TmainMenu 和 TimageList 是组件不在 Form 中显示以外，其它的控件和设计时是一样的。我们观察一下 dfm 文件，发现对一个 Form 来说基本结构是这样的：

```
object ObjectName : ObjectClass
  ObjectProperty = PropertyValue
```

```
    //.....
    object ObjectName：ObjectClass
      //子对象，允许多重嵌套
      //.....
    end
end
```

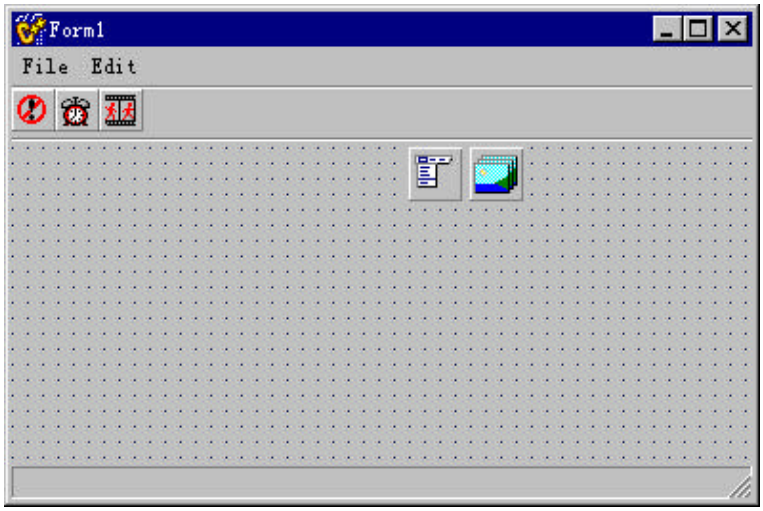


图 4.5 C++ Builder 中的资源显示效果

在上面的例子中，顶层的 ObjectName 为 Form1，ObjectClass 为 TForm1，每一个对象总是以 object 开始，以 end 结束，中间的内容是这个对象的属性值，包括事件和其它的对象，嵌套在一个对象内部的对象称作子对象，其 Parent 属性就是上一层对象，比如在上面的例子的 Form1 中，包含了 ToolBar1、StatusBar1、ImageList1 和 MainMenu1 等四个子对象，而 TollBar1 中包含了三个子对象，分别是 ToolButton1、TollBuitton2 和 TollButton3，ToolBar 就作为三个 ToolButton 的容器控件 Parent 在程序中出现，三个 ToolButton 都是 ToolBar 的 Child 控件。MainMenu 的情况和 ToolBar 相似。

在 dfm 文件中，对象属性的值和在 IDE 中设置的值是一致的，实际上 C++Builder 正是通过 dfm 文件将设计时的属性记录下来的，在设计一个控件的时候，可以定义某个属性是否需要存储，将属性的 stored 部分设置为 false 的时候，属性在 IDE 中的设置不会存储在 dfm 文件中，只有定义成需要存储的属性，并且在 IDE 中设置的值和其缺省值不相同的时候才会被存储在 dfm 文件中，这在前面讲述__property 关键字的时候已经讲过，不过事件在 dfm 文件中总是被存储下来的，而且事件这个特殊的属性没有缺省值，也不能定义是否存储，而且通常必须是可读可写的属性。

3.2.2 C++ Builder 资源的初始化

Dfm 文件在 C++ Builder 中编译以后会生成一个 res 的二进制的资源文件，而每

一个 Form（最顶层的对象，也可能是 DataMoudel 等对象）会被当成一个资源来处理，其类型是 RCDATA 用户自定义类型，这种类型的资源 Windows API 是没有对应的 API 函数来处理的，只能由应用程序来进行处理。

在所有的 VCL 类库中，从 TPersistent 类开始，包括它的派生类中都已经包含了对流式数据的存储和读取，在 TComponent 类之后，增强了从流（TStream）存储和读取组件的能力，而在 dfm 文件中的对象大多数都是从 TComponent 派生的组件，也有一些是从 TPersistent 派生的类，资源文件中所有的组件在单元的头文件中都有声明的原型，而从 TPersistent 派生的类一般在头文件中没有相应的声明原型。一个使用 C++ Builder 编制的应用程序和其它语言设计的程序一样，都包含代码部分和资源部分，但它们的初始化过程有所不同，采用 C++ Builder 设计的程序其资源的初始化过程可以用下图 4.6 来表示：

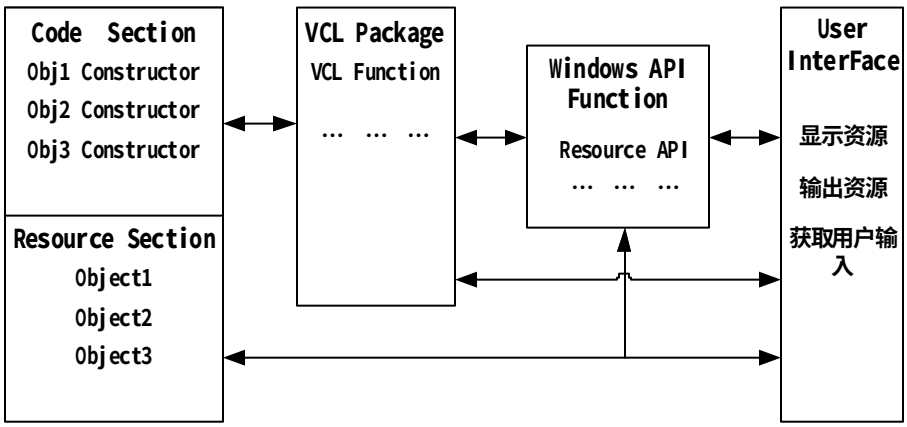


图 4.6 C++ Builder 中的资源的使用

图中显示，应用程序在获取资源的时候，首先需要调用 VCL 资源的处理函数，诸如 LoadFromStream 等方法，将经过编译而生成的二进制资源解释为每个对象所需要的数据，然后每个对象调用相应的构造函数，来生成对象，并最终由 Windows API 函数完成窗体的创建、显示图形等资源的输出和显示。

在 VCL 中有一些控件并不是 Windows 的 API 所提供的，对于这一些对象在创建的时候，很可能是由 VCL 直接向用户输出而不是通过一定的 Resource API 来完成的。除此之外，在 C++ Builder 中，同样可以使用 Windows 的标准资源，用户可以象 SDK 编程那样来使用资源。

3.2.3 在 C++ Builder 中使用资源文件

C++ Builder 可以支持所有的 Windows 标准资源类型，但通常是没有必要这么作的，因为 VCL 所提供的 IDE 已经非常的强大，在某些时候，也许会用到一些特殊的资源，比如声音、压缩图像等等，这些都不是 Windows 的标准资源，而且 VCL 也没有对这些资源也没有进行相应的封装，这时就需要使用资源文件来完成这些功能。

在下面的这个例子中，使用了 Windows 的标准资源类型 Menu、Icon、Cursor、Bitmap 和一个用户类型的资源 JPG 类型，其中 Menu、Icon、Cursor 和 Bitmap 仅仅

是为了演示一下如何在 C++ Builder 中使用标准资源，而这个 JPG 示范才是很可能是在实际中用到的。

在这个例子中有一个主程序和一个动态连接库，主程序中包含了一个资源文件，里面有一个 Windows 的 Menu、Icon 和 Cursor 资源，动态连接库中包含了一个 JPG 的资源,程序运行后会载入 Icon、Menu 和 Cursor,当用户点击 Button1 是程序载入 Bitmap 资源并显示在主窗体中，当用户点击 Button2 时，程序会自动加载动态连接库，并将其中的资源载入主窗体中的 Image 控件。

程序的部分代码如下：

```
TForm1 *Form1;
const crMyCursor = 5;      //载入光标资源使用
//---------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
       : TForm(Owner)
{   /* 空构造函数 */ }
//---------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    HMENU hMenu;        //定义一个 Menu 句柄
    Screen->Cursors[crMyCursor] = LoadCursor(HInstance, "IDC_CURSOR1");
    if(Screen->Cursors[crMyCursor])
       Cursor = crMyCursor;       //通过 Screen 对象将光标载入并供程序使用
    Icon->ReleaseHandle();        //Form 加载 Icon 前首先要释放，VCL 的需要
    Icon->Handle = LoadIcon(HInstance,"IDI_ICON1");  //载入图标 IDI_ICON1
    hMenu = LoadMenu(HInstance,"IDM_MENU1");         //载入 Menu，IDM_MENU1
    if( hMenu )
       if(!::SetMenu( Handle,hMenu ))         //将 Form 的菜单设置为 hMenu
          Application->MessageBox("error","Error",0);
}
//---------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{    //点击 Button1 的事件处理函数
    Graphics::TBitmap *Bitmap1 = new Graphics::TBitmap();
    try
    {
       Bitmap1->LoadFromResourceName((unsigned int)HInstance, "BITMAP1");
       Canvas->Draw(12,12,Bitmap1);   //将位图显示在窗体中
    }  //使用 LoadFromResourceName 方法载入位图，比 API 有更好的适用性
    catch (...)
```

```
    {
      Application->MessageBox("error","Error",0);
    }
    delete Bitmap1;
}
//---------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{      //点击 Button2 的事件处理函数
    TMemoryStream *Stream = new TMemoryStream();
    TJPEGImage * Jpg = new TJPEGImage();
    void *Buffer;
    int Length;
    HMODULE Module;
    HRSRC ReSource;
    HGLOBAL hBuffer;
      //以上为临时变量
    Module = LoadLibrary("Res.dll");              //加载动态连接库 Res.dll
    ReSource = FindResourceEx(Module,"JPG","DllRes",
                  MAKELANGID(LANG_NEUTRAL,SUBLANG_NEUTRAL));
     //查找资源 DllRes，类型为 JPG，JPG 为用户自定义的类型
    Length = SizeofResource(Module,ReSource);     //确定资源的长度
    hBuffer = LoadResource(Module,ReSource);      //载入资源数据
    Stream->WriteBuffer(hBuffer,Length);          //将资源数据导入 Stream 对象
    Stream->Position = 0;
    Jpg->LoadFromStream(Stream);                  //将数据导入 JPG 图形对象
    Image1->AutoSize = true;
    Image1->Picture->Assign(Jpg);                 //显示 Jpeg 图形
    Image1->AutoSize = false;
    FreeLibrary(Module);                          //释放动态连接库
    delete Stream;
    delete Jpg;
}
```

应用程序可以通过动态连接库来使用特定的资源，这样的好处是可以最大限度的节省应用程序占的系统资源，比如程序的启动画面可能会包含一个图形，而这个图形在程序启动以后就不再需要，这时就可以通过动态连接库的资源来加载。

使用动态连接库来加载资源的另一个用处在于隐藏了实际资源的数据，且方便程序更换不同的资源而不需要重新编译整个应用程序，只需要将相应的模块重新编译就可以了。这个例子可以按照一下步骤来实现：

一．创建一个新的 Project，在 Form 中加入一个 Image 控件，两个 Button 控件。
二．在 Project 的 Form 单元文件中加入 Jpeg 单元的包含信息。
三．创建一个资源脚本文件 res.rc 并将其加入到 Priject 中。
四．创建一个任意的 dll 工程，并命名为 Res.dll。
五．创建一个资源脚本文件 dllres.rc，并加入到 Res.dll 工程中。
六．将上面的几个事件处理函数加入单元文件。
这个例子中有两个资源脚本文件，可以按照下面的例子编写：
Res.rc：

```
IDC_CURSOR1 CURSOR
{
 '00 00 02 00 01 00 20 20 00 00 00 00 00 00 30 01'
       //光标资源的数据部分，也可以指定一个特定的光标文件
 'FF FF FF FF FF FF'
}

BITMAP1 BITMAP
{
 '42 4D 76 08 00 00 00 00 00 00 76 00 00 00 28 00'
       //位图资源的数据部分，也可以指定一个特定的位图文件
 '99 99 99 99 99 99'
}

IDI_ICON1 ICON
{
 '00 00 01 00 01 00 20 20 10 00 00 00 00 00 E8 02'
       //图标资源的数据部分，也可以指定一个特定的图标文件
 'FC A1 FF F9 FE BC FF F9 FE 3E FF F9 FF FF'
}

IDM_MENU1 MENU
{
 POPUP "&MenuTest"
 {
  MENUITEM "&Item1", 101
  MENUITEM "&Item2", 102
  MENUITEM "&Item3", 103
 }
}
```

这个资源脚本可以使用 Resource Workshop，或者 Borland C++等工具编写。另一个资源文件 dllres.rc 如下：

```
DllRes JPG "Test.jpg"
```

其中 Test.jpg 可以是任意的 Jpeg 类型文件。关于这个例子可以在本书所附的例程中找到完整的代码。

4 C++ Builder 的异常处理

在上一章讲面向状态的思想时，我们已经提到过，C++ Builder 使用了异常处理机制来保证一段代码可以可靠、完整的执行，而不必要一步一步小心翼翼的，使程序的设计更加符合人类的习惯。异常处理并不是 C++ Builder 所特有的，在标准的 C++语言中本身就支持异常处理，而且在 Win32 中，对于异常处理又加入了更多的内容，这一节将简单的就 C++ Builder 中的异常处理作以介绍。

C++Builder 支持 C++异常处理、基于 C 的结构异常处理以及 VCL 异常处理。本节中所用的关于 C++异常处理及结构异常处理的例子通过命令行使用 bcc32.exe 可成功地编译并运行，而不是使用 IDE。在调用标准 C++例程及对象时可用 C++异常处理。VCL 异常处理可在 IDE 内进行。实际上，C++Builder 同样支持 C++异常处理和基于 C 的结构异常处理。但使用 C++ Builder 和 VCL 可开发包含内嵌异常处理的例程的应用程序，这些例程可以在出现错误时自动发送异常。

4.1 C++异常处理

异常是指需要特殊处理的例外情况，包括运行时发生的错误，如除数为零，存储空间不足等。异常处理提供了一种标准的方法以处理错误，发现可预知及不可预知的问题，及允许开发者识别、查出和修改错漏之处（bugs）。到了 C++Builder 的 VCL 中异常处理并不只是提供了调试、测试等基本功能，而且成为程序正常运行不可缺少的部分。

4.1.1 异常处理的 ANSI 规定

C++ Builder 异常处理支持 ANSI/ISO 提议的 C++工作标准。发送异常可允许你收集发送点的信息，这将有助于诊断异常发生的原因。可使用异常处理程序来确定程序终止之前的操作。只有同步异常（错误都由程序内部引起）可被处理。外部事件（例如按下 Ctrl+C 组合键）不被认为是一个异常。C++语言规定所有的异常都应在一个 try-block 中被发送。这个块之后紧接着是一个或多个 catch 块，用于识别和处理 try-block 中发生的错误。

4.1.2 异常处理语法

异常处理要求使用三个关键字：try、catch 及 throw。程序通过可能产生特殊状况的 trying 语句以准备捕捉异常。当 C++程序发送一个异常时，可将控制转移或 throw 到程序另外的被称为异常处理程序的、用于处理该类异常的部分中去。这种处理程序被称为 catch 异常。程序通过执行 throw 语句来发送异常。throw 语句通常在一个函数内发生，如：

```
throw "overflow";
```

在这个例子中，语句发送一个描述异常类型的对象，这是一个算术运算溢出。从而程序的另一部分可捕捉并处理这个异常对象。要使用异常处理，需将代码封闭在一个 try/catch 结构中，try/catch 结构的语法如下：

```
try-block:
try compound-statement handler-list
handler-list:
handler handler-list    //handler-list 是可选的
handler:
catch (exception-declaration) compound-statement
exception declaration:
type-specifier-list declarator
type-specifier-list abstract-declarator
type-specifier-list
...
throw-expression:
throw assignment-expression  //assignment-expression 是可选的
```

注意 C 程序中不支持 try、catch 及 throw 关键字。一个由 try 确定的 try-block 后必须紧接着一个由 catch 确定的处理程序。try-block 是一段用于确定程序执行时的控制流的语句。若一个异常在 try-block 中被发送，程序控制被转移到适当的异常处理程序。处理程序是一段被设计来用于处理异常的代码。C++语言要求至少有一个处理程序紧接着 try-block。程序应包括一个可处理程序中可能发生的各个异常的处理程序。

4.1.3 声明异常

虽然 C++允许异常可为任意类型，但异常对象还是很有用的。异常对象可当作其他对象一样看待。异常对象将异常发送点的信息携带到异常对象被捕捉的地方。应用程序的用户在程序运行时遇到不正常的情况时会很需要这些信息。由 C++语言预定义的异常，在前面结束异常类可以找到，或在联机帮助中的“Library Reference”中找到更为详细的

解释。

4.1.4 发送异常

可能发生异常的代码块需以 try 为前缀同时封闭在花括号内。这表明程序准备检查异常。若异常发生，则程序流中断，然后会发生：

- 程序搜索匹配的处理程序。
- 若找到处理程序，则栈在该点被打开。
- 程序控制转到处理程序。
- 若没有找到处理程序，可调用 set_terminate（）以提供终止处理程序，否则，程序会调用 terminate 函数。若无异常被发送，程序正常执行。

当异常发生时，throw 语句初始化一个 T 类型（匹配 arg 参数的类型）的临时对象在 throw（Targ）中使用。对象的其他拷贝可在编译器需要时生成。因此，对于包含子对象的类来说，定义一个拷贝结构很有用（对于按位拷贝无须定义拷贝结构）。

下列例子说明了几种发送异常的方法。这些例子通过命令行使用 bcc32.exe 可成功地编译并运行，而不需使用 IDE，例子中在没有处理程序存在是调用 terminate

```
/*程序执行结果:
*** Nobody threw an exception ***
*/
#include <except.h>
#include <process.h>
#include <stdio.h>
bool pass;
class Out{};
void my_terminate(){ puts("*** Nobody threw an exception ***"); exit(1); }
void festival(bool firsttime){ if(firsttime) throw Out(); }
void test()
{
   try        { festival(false); }
   catch(Out& e){ pass = false; }
   throw;  //这里不会继续抛出异常
}
int main()
{
   set_terminate(my_terminate);
   try        { test();     }
   catch(Out& e){ pass = true; }
   return pass ? (puts("got out of test"),0) : (puts("still have test"),1);
}
```

4.1.5 异常规范

C++提供一种称为议程规范（exception specification）的特性可将一个函数可能发送的异常在声明中列出。

exception specification 使用时类似于函数声明的后缀，句法如下：

```
exception-specification:

throw (type-id-list)            //type-id-list 为可选的
type-id-list:
type-id
type-id-list, type-id
```

这样的函数后缀不是函数类型的一部分。因此，指向函数的指针不会受 exception specification 的影响。像这样的指针只检查函数返回值和参数类型。因此，下列声明是合法的：

```
void f2(void) throw();          //没有异常被抛出
void f3(void) throw (BETA);     // 只能抛出 BETA 异常类
void (* fptr)();                // 返回为 void 的函数指针
fptr = f2;
fptr = f3;
```

当重载虚拟函数时需要小心。因为 exception specification 不是函数类型的一部分，所以可能会破坏程序设计。

例子：在下例中，定义的派生类 BETA::vfunc 不发送任何异常，这偏离了原始的函数声明。

```
class ALPHA
{
public:
   struct ALPHA_ERR {};
   virtual void vfunc(void) throw (ALPHA_ERR) {} // Exception specification
};

class BETA : public ALPHA
{
   void vfunc(void) throw() {} // Exception specification 被改变
};
```

下例是带有异常规范的函数。

```
void f1();                  // 该函数可以抛出任意类型的异常
void f2() throw();          // 不能抛出任何类型的异常
void f3() throw( A, B* );   // 可以抛出一个 A 或其派生类的异常，或者 B 类或其派生
                            // 类的指针异常
```

这样的函数的定义和所有的声明必须有一个包含相同的 type-ids 集的异常规范。若函数发送一个规范中未列出的异常，程序会调用 unexpected。

4.1.6 异常处理的构造和析构

若不能成功地构造一个对象，类构造函数会发送异常。当按值发送对象且异常被引发时，拷贝构造函数被调用。拷贝构造函数在发送点初始化一个临时对象。应用程序也可产生另外的拷贝。如果构造函数发送了异常，对象的析构函数就不需要被调用。析构函数仅为基类和那些在 try-block 中构造成功的对象调用。为 try-block 中构造的 throw 语句后的对象调用析构函数的进程称为 stack unwinding 。若析构函数在这个进程中引发一个异常且没有处理，则 terminate 被调用。析构函数是缺省调用的，不过可通过使用 -xd- 编译器选项关闭缺省设置。

4.2 Win32 下的结构异常

Win32 支持与 C++异常类似的基于 C 的结构异常处理。不过还是有一些关键性的差别，当与 C++代码混用时需要小心地使用。在 C++ Builder 应用程序中使用结构异常处理时需注意：

- C 结构异常可在 C++程序中使用。
- C++异常不能在 C 程序中使用，因为 C++异常要求处理程序由 catch 关键字指定，而 catch 关键字在 C 程序中是不允许的。
- 一个由调用 RaiseException 函数生成的异常应在一个 try/__except(C++) 块中或 __try/__except(C) 块中处理（也可使用 try/__finally 或 __try/__finally 块。当 Raise Exception 函数调用时所有 try/catch 块中的处理程序都被忽略。
- 没有被应用程序处理的异常不会导致调用 terminate()，而是被传递给操作系统（通常来说，操作系统最终处理的结果也是结束进程）。
- 异常处理程序不接受异常对象的拷贝，除非它们请求如此。

可在 C 或 C++程序中使用下列异常函数：

- GetExceptionCode 。
- GetExceptionInformation 。
- SetUnhandledExceptionFilter 。
- UnhandledExceptionFilter 。

C++ Builder 没有限制在__try/__except 过滤器或 try/__except 块之外使用 UnhandledExceptionFilter 函数。但是，当这个函数在__try/__except 过滤器或 try/__except 块之外调用时，程序行为不确定。

4.2.1 结构异常的语法

在 C 程序中,实现结构异常的 ANSI 兼容关键字为__except、__finally 和__try。注意__try 关键字仅能在 C 程序中使用。若想要编写可移植的代码，就不要在 C++程序中使用结构异常处理。

try-except 异常处理语法如下：

```
try-block:
__try compound-statement (in a C module)
try compound-statement (in a C++ module)
handler:
__except (expression) compound-statement
try-finally termination syntax
```

try-finally 终止语法如下：

```
try-block:
__try compound-statement (in a C module)
try compound-statement (in a C++ module)
termination:
__finally compound-statement
```

结构异常可使用扩展的 C++ 异常处理：

```
try
{
   foo();
}
__except(__expr__)
{
   //异常处理
}
```

4.2.2 异常过滤器

一个过滤器表达式可调用过滤器函数，但是过滤器函数不能调用 GetExceptionInformation。可以把 GetExceptionInformation 的返回值作为一个参数传递给过滤器函数。为传递 EXCEPTION_POINTERS 信息到一个异常处理程序，过滤器表达式

或过滤器函数必须从 GetExceptionInformation 拷贝指针或数据到处理器以后可访问它的地方。在嵌套的 try-except 语句中，每个语句的过滤器表达式都会被计算，直到它定位到 EXCEPTION_EXECUTE_HANDLER 或 EXCEPTION_CONTINUE_EXECUTION。过滤器表达式能调用 GetExceptionInformation 以得到异常信息。只要 GetExceptionInformation 或 GetExceptionCode 直接在提供给__except 的表达式中被调用，你可使用这个函数决定如何处理异常而非创建一个复杂的 C++表达式。几乎所有处理异常需要的信息都能从 GetExceptionInformation()的结果中提取。GetExceptionInformation()返回一指针到 EXCEPTION_POINTERS 结构：

```
struct EXCEPTION_POINTERS
{
    EXCEPTION_RECORD *ExceptionRecord;
    CONTEXT *Context;
};
```

EXCEPTION_RECORD 包含与硬件无关的状态：

```
struct EXCEPTION_RECORD
{
    DWORD ExceptionCode;
    DWORD ExceptionFlags;
    struct EXCEPTION_RECORD *ExceptionRecord;
    void *ExceptionAddress;
    DWORD NumberParameters;
    DWORD ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
};
```

GetExceptionInformation()返回一指针到 EXCEPTION_POINTERS 结构：

EXCEPTION_RECORD 包含与硬件无关的状态：通常，过滤器函数在 ExceptionRecord 中搜索信息以决定如何响应。有时需要一些更特殊的信息（特别是操作为 EXCEPTION_CONTINUE_EXECUTION：若没做任何事，引起异常的代码将再被执行）。对于这种状况，EXCEPTION_POINTERS 结构中另外的字段提供异常发生时的处理器状态。如果这个结构被修改并且过滤器返回 EXCEPTION_CONTINUE_EXCEPTION，它将被用来在执行继续前设置线程的状态。例如：

```
static int xfilter(EXCEPTION_POINTERS *xp)
{
    int rc;
    EXCEPTION_RECORD *xr = xp->ExceptionRecord;
```

```
    CONTEXT *xc = xp->Context;
    switch (xr->ExceptionCode) {
        case EXCEPTION_BREAKPOINT:
            //内嵌的断点，步进！
            ++xc->Eip;
            rc = EXCEPTION_CONTINUE_EXECUTION;
            break;
        case EXCEPTION_ACCESS_VIOLATION:
            rc = EXCEPTION_EXECUTE_HANDLER;
            break;
        default:
            //放弃
            rc = EXCEPTION_CONTINUE_SEARCH;
            break;
    };
    return rc;
}
...
EXCEPTION_POINTERS *xp;
try
{
    func();
}
__except(xfilter(xp = GetExceptionInformation()))
{
    abort();
}
```

4.2.3 在 C++中混用结构异常

在 C++程序中混用结构异常时需要了解几项内容。首先，尽管 C++ Builder 用 Win32 结构异常实现 C++异常，C++异常对__except 块来说仍是透明的。一个 try 块后可跟随一个 except 块或至少一个 catch 块。若试图混用二者会产生一个编译错误。需要处理两种类型异常的代码应该简单地在两个 try 块内嵌套：

```
try
{
    EXCEPTION_POINTERS *xp;
    try
    {
```

```
        func();
    }
    __except(xfilter(xp = GetExceptionInformation()))
    {
       //...
    }
}
catch (...)
{
   //...
}
```

函数 throw()规范不会影响程序关于 Win32 异常的行为。另外，未处理的异常最后由操作系统处理（若调试器不首先处理它），不像 C++程序那样调用 terminate()。任何使用-xd 编译器选项（缺省打开）编译的程序块将调用所有“自动”存储的对象的析构函数。打开栈的操作从异常发生点到异常被捕捉点间发生。

C++程序中基于 C 的异常示例

```
/*程序执行结果:
Another exception:
Caught a C-based exception.
Caught C++ exception[Hardware error: Divide by 0]
C++ allows __finally too!  */
#include <stdio.h>
#include <string.h>
#include <windows.h>
class Exception
{
public:
    Exception(char* s = "Unknown"){ what = strdup(s);      }
    Exception(const Exception& e ){ what = strdup(e.what); }
    ~Exception()                 { delete[] what;        }
    char* msg() const            { return what;          }
private:
    char* what;
};
int main()
{
    float e, f, g;
```

```
    try
    {
      try
      {
         f = 1.0;
         g = 0.0;
         try
         {
            puts("Another exception:");
            e = f / g;
         }
         __except(EXCEPTION_EXECUTE_HANDLER)
         {
            puts("Caught a C-based exception.");
            throw(Exception("Hardware error: Divide by 0"));
         }
      }
      catch(const Exception& e)
      {
         printf("Caught C++ Exception: %s :\n", e.msg());
      }
    }
    __finally
    {
       puts("C++ allows __finally too!");
    }
    return e;
}
```

4.2.4 __finally 终止块

结构异常处理模型支持“终止块”，在被保护块正常退出后执行或经由异常。C++ Builder 编译器在 C 中以下列语法支持它：

```
__try
{
   func();
}
__finally
{
```

```
    //不管 func 是否抛出异常，该段代码都会被执行
}
```

下例说明如何使用终止块：

```
/*程序执行结果:
An exception:
Caught an exception.
The __finally is executed too!
No exception:
No exception happened, but __finally still executes!  */

#include <stdio.h>
#include <windows.h>
int main()
{
    float e, f, g;
    try
    {
       f = 1.0;
       g = 0.0;
       try
       {
          puts("An exception:");
          e = f / g;
       }
       __except(EXCEPTION_EXECUTE_HANDLER)
       {
          puts("Caught an exception.");
       }
    }
    __finally
    {
       puts("The __finally is executed too!");
    }
    try
    {
       f = 1.0;
       g = 2.0;
```

```
        try
        {
           puts("No exception:");
           e = f / g;
        }
        __except(EXCEPTION_EXECUTE_HANDLER)
       {
           puts("Caught an exception.");
       }
    }
    __finally
    {
       puts("No exception happened, but __finally still executes!");
    }
    return e;
}
```

C++代码也能通过创建局部对象处理终止块，这些对象在范围退出时调用析构函数。由于 C++ Builder 结构异常支持析构清理，这使得程序可以不用考虑异常的类型。注意有一个需要担心的情况，当异常被引发并没有处理程序时。对于 C++异常，C++Builder 编译器为局部对象调用析构函数（不要求由语言定义），而未处理的 Win32 异常，析构清理则不会发生。

4.3 VCL 异常处理

在应用程序中使用 VCL 组件时，需要理解 VCL 异常处理机制。因为异常被内置于许多类中，并且当意外事件发生时自动地被发送。若不处理异常，VCL 将以缺省方式处理。通常，是显示描述发生错误类型的消息。当编程时遇见显示的一条指示被发送异常类型的消息时，可在 VCL Reference 中查一下异常类。提供的信息经常能够帮助你决定错误在何处发生及其原因。另外，前面介绍了可能引起异常的语言差异，以及在对象构造期间发送的异常。

4.3.1 C++和 VCL 异常处理之间的差别

下面是 C++和 VCL 异常处理之间一些需注意的差别。

由构造函数发送的异常：

- C++析构函数应为被完整构造的基类及成员调用。
- VCL 基类析构函数甚至在对象或基类未被完整构造时也调用。

捕捉并发送异常：

- C++异常可以通过引用、指针或值捕捉。VCL 异常是从 TObject 派生出的异常

类，仅可通过引用或指针捕捉。试图通过值捕捉 TObject 异常会导致编译错误。硬件或操作系统异常，例如 EAccessViolation，应通过引用捕捉。

- VCL 异常通过引用捕捉。
- 不能使用 throw 重新发送在 VCL 代码内被捕捉的异常。

4.3.2 处理操作系统异常

C++ Builder 允许处理操作系统引发的异常。操作系统异常包括越权访问、整数运算错、浮点运算错、栈溢出以及 Ctrl+C 中断。这些在 C++ RTL 中处理，并且在分发到应用程序前被转换为 VCL 异常类。可以编写如下的 C++代码：

```
try
{
    char *p = 0;
    *p = 0;
}

//必须始终以引用的方式 catch

catch (const EAccessViolation &e)
{
    printf("You can't do that!\n");
}
```

C++ Builder 使用的类与 Delphi 的相同，并仅存在于 C++ Builder VCL 应用程序中。从 TObject 派生并要求 VCL 支持。下面是 C++ Builder 异常处理的一些特征：

- 不负责释放异常对象。
- 操作系统异常应通过引用捕捉。
- 在操作系统异常被相关的 VCL catch 块捕捉，并且 catch 块结束后不能重新发送操作系统异常。
- 在操作系统异常被相关的操作系统 catch 块捕捉，并且 catch 块结束后不能重新发送操作系统异常。

最后两点可以粗略地说：一旦操作系统异常作为 C++异常被捕捉，就不能作为操作系统异常或 VCL 异常被重新发送，除非在 catch 栈框架之内。

4.3.3 处理 VCL 异常

C++ Builder 为处理由 VCL 引发的软件异常或由 C++引发的 TObject 派生的异常类而拓宽语义。在这种情况下，由于 VCL 风格类仅能在堆中被分配，从而有下面两个规则。

- VCL 风格异常类只能通过指针捕捉，若其为软件异常也可以通过引用（通常推荐

使用引用）捕捉。

- VCL 风格异常应该用“by value”语法发送。

C++ Builder 包括一大组内置异常类用以自动处理诸如除数为零错误、文件 I/O 错误、无效类型转换及许多其他的异常状况。正如前面章节说讲述的，所有的 VCL 异常类都从一个根对象 Exception 派生而来。Exception 封装了所有异常类的基本属性和方法，并且为应用程序处理异常提供一致的接口。可以用一个 Exception 类型的参数把异常传递给 catch 块。可以使用下列语法捕捉 VCL 异常：

```
catch (const exception_class &exception_variable)
```

确定想要捕捉的异常类并提供一个变量以引用它。下面是如何发送 VCL 异常的例子：

```
void __fastcall TForm1::ThrowException(TObject *Sender)
{
    try
    {
        throw Exception("VCL component");
    }
    catch(const Exception &E)
    {
        ShowMessage(AnsiString(E.ClassName())+ E.Message);
    }
}
```

在前例中 throw 语句创建异常类的一个实例并且调用其构造函数。所有由 Exception 派生而来的异常都有可被显示的消息，通过构造函数传递并通过消息属性检索。下面给出了由 Exception 派生的一些 VCL 内置的异常类及其用途，关于 Exception 类及其派生类的详细内容参见前面相关内容和 C++ Builder 联机帮助。

异常类	类描述：
EAbort	不显示错误消息对话框，终止事件序列
EAccessViolation	检查内存访问错误
EBitsError	阻止对布尔型数组无效访问
EComponentError	表示试图无效注册或重命名组件
EConvertError	指出字符串或对象转换错误
EDatabaseError	指出一个数据库访问错误
EDBEditError	捕捉与指定模式不兼容的数据
EDivByZero	捕捉整数零除错误

EExternalException	代表未被识别的异常代码
EInOutError	代表一个文件 I/O 错误
EIntOverflow	指出其结果对分配的寄存器太大的整数计算
EInvalidCast	检查不合法的类型转换
EInvalidGraphic	指出欲使用未被识别的图形文件格式
EInvalidOperation	试图对组件非法操作时发生
EInvalidPointer	源于非法指针操作
EMenuError	菜单项包含的问题
EOleCtrlError	检测 ActiveX 控件的问题
EOleError	指定 OLE 自动化错误
EPrinterError	表明打印错误
EPropertyError	设置属性值失败时发生
ERangeError	指出整数值对其声明类型来说太大
ERegistryException	指出注册表错误
EStackOverflow	当栈进入最后的保护页时发生
EZeroDivide	捕捉浮点零除错误

从上面的表可看到，内置 VCL 异常类为你处理了大部分异常并能简化代码。这样可方便你创建自己的异常类处理唯一的状况。可声明一个新异常类为 Exception 的派生类，和 Exception 有同样多的构造函数(或从 SYSUTILS.HPP 拷贝一个已有的类构造函数)。

5 小结

本节主要讲述了 C++ Builder 中的一些关键字及语法的扩充；C++ Builder 的 VCL 基本类库的结构即主要属性等；C++ Builder 中资源的使用以及 C++ Builder 中异常处理的基本特征，通过本节内容，相信读者会对 C++ Builder 的一些运行机制有个更加深入的了解，这些内容有些并不是使用 C++ Builder 所必须的，但是了解这些内容可以更好的、更自由的驾驭 C++ Builder。