Borland Delphi 5
High-productivity development for the Internet

Delphi COM 深入编程

(美) Eric Harmon 著
陈 旭 杨 彬 刘 怀 等译

机械工业出版社
China Machine Press

Delphi COM 深入编程

（美） Eric Harmon 著

陈旭 杨彬 刘怀 等译

机 械 工 业 出 版 社

本书是针对想使用 Delphi 开发 COM/DCOM 应用程序、定制的 COM/DCOM 对象组件或创建可以扩展、易于维护的应用程序的一本独具特色的参考书。它提供了关于接口、类型库、COM 事件/回调及结构化存储的非常好的信息。所有的实例程序都可以直接执行并使用在许多实际的应用程序中。

本书是每一个 Delphi COM 程序员的必备参考书。

图字 01-2000-899

图书在版编目（CIP）数据

Delphi COM 深入编程/（美）哈蒙（Harmon E.）著；陈旭等译.一北京：机械工业出版社，2000.10

ISBN 7-111-08271-0

书名原文：Delphi™ COM Programming

Ⅰ. D… Ⅱ. ①哈… ②陈… Ⅲ. ①Delphi 语言-程序设计②软件-接口 Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2000）第 70900 号

机械工业出版社（北京市百万庄大街 22 号 邮政编码 100037）

责任编辑：张秀恩 封面设计：姚 毅

责任印制：郭景龙

北京京丰印刷厂印刷·新华书店北京发行所发行

2000 年 10 月第 1 版第 1 次印刷

787mm×1092mm 1/16 · 30.5 印张 · 755 千字

0001—5000 册

定价：48.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

本书购书热线电话（010）68993821、68326677-2527

Http://www.machineinfo.gov.cn/book/

前　言

每当我和其他 Delphi 开发人员在 Borland 全球会议或在用户小组的时候，总是有人向我提起那次声名狼籍的有关创建 ActiveX 控件的会议，那是 1995 年在加州的 Anaheim 举办的。我当时用 1 小时 15 分钟向 1400 个开发人员演示了如何在 Delphi 2 中编写一个 ActiveX 控件；用了 5700 行代码，仅实现了一个可以在 Visual Basic 4 窗体中点击的按钮。结果把他们全吓坏了。

那时，Delphi 的前首席设计师 Anders Hejlsberg 邀请我加入 Delphi 开发组，作为“不可完成的任何”小组中的一员，该小组也就是后来的 DAX 小组。这时，工作组成员有 Anders、Conrad Herrmann 和我。后来，来自 Paradox 小组的 Joe“TLB”Bentley 加入我们的小组。最后，没过多久，我们有了最后一名成员，我的好友 Steve“Cookie Monster”Teixeira。

工作是令人难以置信的。我们知道自己正位于技术的最前沿，但同时由于缺少文献、论文和经验，迫使我们没日没夜地工作几个月，目的是创造一个迄今为止最好而且使用最简单的框架。

成为一名 COM 开发人员只需要很短的时间；成为一名好的 COM 开发人员则要花费大量的时间。然而对于我们，情况却完全不一样。我们不仅要创造出一个框架，它能够轻松地创建 COM 服务器和客户应用程序，而且我们还必须在现存的所有支持 COM 的软件上进行测试：Visual C++3、4 和 5；Visual Basic 4 和 5；Internet Explorer 3、4 和 5；Borland C++；Office 产品，等等。考虑到每个容器（COM 容器）里面都有一个“gotcha”（I have got you，英语口语：我得到你了），而每个“gotcha”又包含另一个，我认为我们已经做得相当好了。

Don Box 说：“COM 是爱”。我说：“COM 很难”。这就是为什么我非常高兴得知 Eric 决定编写这本有价值的书，来向 Delphi 开发人员介绍 COM 世界，并且用此重要的、永久的技术来增强他们的能力，使他们更快速地开发应用程序。

开始读吧，你会喜欢它，并着手编程。

Alain“Lino”Tadros
技术主管
DeVries 数据系统
www.dvdata.com

第 1 章　在 Delphi 中使用接口

从 Delphi 3 开始，Borland 公司把 **interface** 关键字引入到 Delphi 中。接口形成了 COM 编程的基础，但有趣的是，它们也可以用于非面向 COM 的程序开发中。因此，本章着重讲述作为语言要素的接口。在本章中学到的绝大多数东西都可以 100%地应用于 COM，但是 COM 这个词在本章中不会多次被提到。

除了它们共享同一名称外，在此提到的接口与一个单元的 interface 部分没有任何关系。接口这个词的双重用法不应是混淆的来源，因为它们用在两个完全不相关的上下文中。

1.1　定义接口

也许理解接口的最简单方法是它或多或少等同于一个抽象类。如果不熟悉抽象类，则将其理解为定义行为的类，但是其本身并不能实现功能。相反，它们是依靠派生类来实现的。考虑如下的类声明：

```
type
    TFormattedNumber = class
    public
      function FormattedString: string; virtual; abstract;
    end;
```

这是一个抽象类，读者应该认为它是给一系列派生类建立基础，这些派生类具有一些共性。即，它们全部都可以通过一个叫做 FormattedString 的方法格式化某种数字。

永远不可能创建一个抽象类的实例（或者至少永远不要试图创建一个抽象类的实例！）。创建这个类只是用来“发号施令”的，比如说对它的派生类起作用。

考虑下列代码片断：

```
var
    MyNumber: TFormattedNumber;
begin
    MyNumber := TFormattedNumber.Create;
    MyNumber.Free;
end;
```

这些是合法的代码，编译该代码时从编译器中获得的全部内容为如下的警告：

```
Constructing instance of 'TFormattedNumber' containing abstract methods
```

该代码编译并运行良好。然而，若是尝试使用 MyNumber，Delphi 就会抛出一个 EAbstractError 异常，指出用户要调用抽象方法。

为使 TFormattedNumber 类有用，需要从其中派生类，如下：

```
type
  TFormattedNumber = class
```

```
  public
    function FormattedString: string; virtual; abstract;
  end;

  TFormattedInteger = class(TFormattedNumber)
  private
    FValue: Integer;
  public
    function FormattedString: string; override;
  end;

  TFormattedDouble = class(TFormattedNumber)
  private
    FValue: Double;
  public
    function FormattedString: string; override;
  end;
```

定义类之后，就可以像下面这样使用它：

```
procedure ShowFormattedNumber(F: TFormattedNumber);
begin
  ShowMessage(F.FormattedString);
end;
```

ShowFormattedNumber 过程并不在意是否传递给它 TFormattedInteger 或 TFormattedDouble 类型的参数。它仅仅调用合适类的 FormattedString 方法

对于经验丰富的 Delphi 程序员来说，这并不新鲜。现在我们使用接口来看一下相同的结构。将看到接口可使我们完成同一个结果，并带有一些有趣的（而且是有用的）副作用。

```
type
  IFormattedNumber = interface
    function FormattedString: string;
  end;
```

TFormattedNumber 和 IFormattedNumber 声明之间的相似是很明显的。然而，仍然有一些微妙的并且是重要的差别需要读者了解。以下部分详细探究了这些差别。

1.1.1 作为协议的接口

从概念上讲，接口只不过是接口实现者和接口使用者之间的协议。定义接口，实际上是在说："我看到了这个功能的必要。我承认该功能可以用许多不同方式来实现。我真的不在乎该功能是如何实现的，但是最好坚持这些规定。"接口用户可以往接口规定上编码，而不必担心规定会改变。

在前面的实例中，想要创建一个类（或实际上是一组类），它支持格式化不同类型的数字。派生类可能包括一些类来格式化整数、浮点数、十六进制数等。

接口不是类，而是接口。因此，必须这样声明它。在下面，将更深入地探讨这个声明的重要性。

1.1.2　接口和类的不同

尽管接口与抽象类有众多相同之处，但是它们也有几个重要的不同之处。我们来详细检查一下它们的不同。

为了便于参考，此处重复两个声明：

```
TFormattedNumber = class
protected
  function FormattedString: string virtual; abstract;
end;

IFormattedNumber = interface
  function FormattedString: string;
end;
```

接口有如下特征：

- 接口被声明为 interface 类型，不是 class 类型。惯例是：接口名从字母 I 开始，正如类类型名从字母 T 开始。
- 所有的接口都从 IUnknown 直接或间接继承。就像 TObject 是 Delphi 中所有类的基类一样，IUnknown 是 COM 中所有接口的基础。将在本章稍后部分详细探究 IUnknown 接口。
- 不能创建接口实例。下面的代码是非法的：

```
var
 MyNumber: IFormattedNumber;
begin
 MyNumber := IFormattedNumber.Create; // 非法
end;
```

当试图编译代码时编译器产生如下错误信息：

```
Object or class type required
```

此特征是受欢迎的安全网。尽管用抽象类可以明确地创建那个类的实例，但是这个特性使编译器增强了不能创建接口实例的概念。

- 不能在接口中指定范围指示。接口定义的所有方法都是公有型（public），不能在接口声明中包括范围指示（包括公有型）。
- 接口不能声明变量。接口只能决定提供什么样功能。对于如何完成该功能没有限制。如果允许在接口声明中放一个成员变量，就会在某种程度上命令用何种方法来完成想要的功能。
- 在接口中声明的所有函数和过程，概念上讲都是虚（virtual）抽象函数和过程。声明时不必带 virtual 关键字；事实上，这样做是非法的。

1.1.3　接口是不变的

定义并公布接口之后，就固定成型，不能修改那个接口了。显然，在开发接口过程中，

会经历许多接口及相关代码的修正，但是公布接口给他人使用后，就不能改变接口了。如果需要增强接口，可以发表新“版本”的接口。例如假设想要把一个称为 SetCaption 的方法添加到 IFormattedNumber 接口中。创建一个新的接口，叫做 IFormattedNumber2，就像这样：

```
type
  IFormattedNumber2 = interface
    function FormattedString: string;
    procedure SetCaption(ACaption: string);
  end;
```

任何仍在使用 IFormattedNumber 接口的现有代码会继续不变地工作。新代码可以利用新的 IFormattedNumber2 接口。

注意

声明了新的 IFormattedNumber2 接口并不意味着必须完全重新实现接口。如果 IFormattedNumber2 接口所做的就是添加一个新的 SetCaption 方法，那么可以从 IFormattedNumber 派生 IFormattedNumber2。

```
type
  IFormattedNumber2 = interface(IFormattedNumber)
    procedure SetCaption(ACaption: string);
  end;
```

现在知道了什么是接口以及它和抽象类是如何关联的（并且不同的）。在下面一节中，将演示如何声明一个接口。

1.2 声明一个接口

读者在上面已经看到了简单接口声明的实例。下面又是一个实例：

```
IFormattedNumber = interface
  ['{2DE825C1-EADF-11D2-B39F-0040F67455FE}']
  function FormattedString: string;
end;
```

马上就应注意到出现在接口名下面的新行。该行是指全球唯一标识符（Globally Unique Identifier），或缩写为 GUID。所有的 COM 接口以及某些只在自己的应用程序内部使用的接口，需要唯一的 GUID 才能正常运行。

定义 GUID

GUID 是 16 字节的唯一数字。创建 GUID 的运算法则非常复杂，由开放软件基金会（Open Software Foundation）定义。运算法则考虑了当前日期和时间、当前用于产生 GUID 的程序过程号、存在网卡中的唯一的 ID（如果有的话）以及其他东西。幸运的是，不必理解运算法就可以给自己的接口创建 GUID。

如果有网卡，所有在计算机中产生的 GUID 都能保证是唯一的。所有的网卡都包括一个唯一的序列号，即 MAC 地址，已把它作为 GUID 的部分包括在内。如果没有网卡，仍然在统计上保障 GUID 是唯一的。这归因于 GUID 中大量的位数和用于产生 GUID 的运算法则。

极其重要的是，当创建一个新的接口时，就生成了一个新的、唯一的 GUID 用于那个接口。永远不要把 GUID 从一个接口复制到另一个接口。

当需要 GUID 时，Delphi 使生成 GUID 变得十分简单。只要把光标放在想要插入 GUID 的地方，并按 Ctrl+Shift+G 就可以了。在内部，Delphi 调用 Windows API 函数 CoCreateGuid 来产生 GUID。

程序清单 1-1 演示了一个编程实例，可直接调用 CoCreateGuid 来产生 GUID 程序清单。

程序清单 1-1　GUIDDemo 应用程序的主窗体

```
unit MainForm;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls, ComObj,
ActiveX;

type
  TForm1 = class(TForm)
    btnGenerate: TButton;
    Memo1: TMemo;
    procedure btnGenerateClick(Sender: TObject);
private
  { Private declarations }
public
  { Public declarations }
end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.btnGenerateClick(Sender: TObject);

var
  Guid: TGUID;
begin
  CoGreateGuid(Guid);
```

```
    Memo1.Lines.Add(GuidToString(Guid));
  end;

  end.
```

图 1-1 显示了运行中的 GUIDDemo。

请注意：GUID 看起来好像是连续编号的。但是，如果关闭程序并再一次运行它，就会产生一个完全不同的 GUID 集。

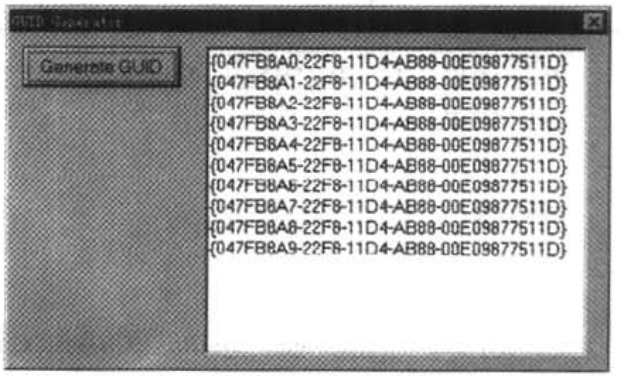

图 1-1　GUIDDemo 演示了如何调用 CoCreateGuid

Delphi 定义 TGUID 结构如下：

```
TGUID = record
  D1: LongWord;
  D2: Word;
  D3: Word;
  D4: array[0..7] of Byte;
end;
```

TGUID 中的四个域对应于 GUID 结构的四个部分。GUID 在 Delphi 中可以用两种方法表示。例如以 IFormattedNumber 的 GUID 为例，下面的两个声明是完全一样的：

```
MyGuid := ['{2DE825C1-EADF-11D2-B39F-0040F67455FE}'];
MyGUID: TGUID = (D1: $2DE825C1; D2: $EADF; D3: $11D2; D4: ($B3, $9F, $00, $40, $F6, $74, $55, $FE));
```

显然，第一行更容易读。

声明接口是创建一个有用的接口的前半部分。声明接口之后，仍然需要给出接口的实现。在下一部分，将演示如何实现接口。

1.3　实现接口

正如上一部分提到的那样，不能创建接口的实例。那么，如何使用 IFormattedNumber 接口？

答案是创建实现接口的类，如：

```
type
TFormattedInteger = class(TObject, IFormattedNumber)
private:
 FValue: Integer;
public:
 constructor Create(AValue: Integer);
 function FormattedString: string; procedure SetValue(AValue: Integer);
end;
```

现在我们可以按通常的方法创建类的实例了，如下：

```
constructor TFormattedInteger.Create(AValue: Integer);
begin
 inherited Create;

 FValue := AValue;
end;

function TFormattedInteger.FormattedString: string;
begin
 Result := 'The value is ' + IntToStr(FValue);
    end;

procedure TFormattedInteger.SetValue(AValue: Integer);
begin
   FValue := Value;
end;
```

如果想在此点编译该代码，就会得到如下错误：

```
Undeclared identifier: 'QueryInterface'
Undeclared identifier: '_ AddRef'
Undeclared identifier: '_Release'
```

这些标识符是从哪儿来的？我们当然不能在我们的代码中定义它们。这些是在 IUnknown 接口中定义的函数。重新回顾一下前面讲述的，即所有的接口都是最终从 IUnknown 中派生出来的，但没有确切解释什么是 IUnknown。

1.3.1　实现 IUnknown

如果在 system.pas 的 VCL 源代码中看一下，将会看到下列声明：

```
IUnknown = interface
['{00000000-0000-0000-C000-000000000046}']
function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
function _AddRef: Integer; stdcall;
 function _Release: Integer; stdcall;
end;
```

IUnknown 是所有接口的基类，并由 Microsoft 定义。QueryInterface、_AddRef 和 _Release 都是函数，必须由 IUnknown 的所有派生来实现。也就是说，必须由实现接口的所有类来实现。

我们花一些时间来详细探究 IUnknown 接口。

1. QueryInterface

QueryInterface 是请求指向一个接口指针的函数。如果接口是由问题中的对象实现的，QueryInterface 返回在 Obj 参数中的接口，且返回数值为 0。如果接口不是由对象实现的，QueryInterface 返回 Microsoft 定义的常量 E_NOINTERFACE。

2. _AddRef

接口是引用计数。因此在对象中当获得接口指针时，对象被引用的次数就增加了。当结束使用接口时，对象的引用次数就下降了。当引用次数到达零时，就自动销毁对象。_AddRef是负责增大引用计数的函数。如何物理上存储引用计数由读者决定，但是典型情况下将建立整型(Integer)变量去保存计数。

3. _Release

_Release 是个负责降低引用计数的函数。当引用计数达到零时，就自动销毁对象。

引用计数在 1.3.2 节中“接口是被计数的引用”部分更详细地讨论。现在，仅仅了解接口就是被计数的引用，以及_AddRef 增加了引用计数并且_Release 降低了引用计数就可以了。

4. 人工实现 IUnknown

我们还不能编译 TFormattedNumber，因为它并不实现 IUnknown 接口。添加一些必要的函数是很容易的事，如程序清单 1-2 所示的新的 TFormattedInteger 声明。

程序清单 1-2 实现 IUnknown

```
type
    TFormattedInteger = class(TObject, IFormattedNumber)
  private:
    FRefCount: Integer;
    FValue: Integer;
    public:
    Constructor Create(AValue: Integer);

    // IUnknown 接口
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;

    // IFormattedNumber 接口
    Function FormattedString: string; virtual;
    Procedure SetValue(AValue: Integer);
end;

constructor TFormattedInteger.Create(AValue: Integer);
begin
 Inherited Create;

 FValue := AValue;
end;

// IUnknown 接口
function TFormattedInteger.QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
```

```
const
  E_NOINTERFACE = $80004002;
begin
  if GetInterface(IID, Obj) then
    Result :=0
  else
    Result :=E_NOINTERFACE;
end;

function TFormattedInteger._AddRef: Integer; stdcall;
begin
  Inc(FRefCount);
  Result := FrefCount;
end;

function TFormattedInteger._Release: Integer; stdcall;
begin
 Dec(FrefCount);
 Result := FrefCount;
 If FRefCount = 0 then
   Destroy;
end;

// IFormattedNumber 接口
function TFormattedInteger.FormattedString: string;
begin
  Result := 'The value is ' + IntToStr(FValue);
end;

procedure TFormattedInteger.SetValue(AValue: Integer);
begin
  FValue := Value;
end;
```

现在暂时不考虑 QueryInterface 的实现。目前仅把它接受为是此函数非常标准的实现。在本章稍后部分，将详细探究 QueryInterface。

正如读者所见，_AddRef 和_ Release 简单增加和降低了 FRefCount 变量的值。当变量值降到零时，就销毁对象。

5. 使用 TInterfacedObject 来自动实现 IUnknown

幸运的是，不必在每次创建实现接口的类时都编写代码。Delphi 为读者提供了名为

TInterfacedObject 的类来实现 IUnknown 接口。程序清单 1-3 演示了 TInterfacedObject 部分的源代码，这些部分对本段的讨论来说很有趣。TInterfacedObject 在 system.pas 中定义。

程序清单 1-3 TInterfacedObject 的源代码

```
  TInterfacedObject = class(TObject, IUnknown)
  protected
  FRefCount: Integer;
  function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
  public
  procedure BeforeDestruction; override;
  property RefCount: Integer read FRefCount;
  end;

procedure TInterfacedObject.BeforeDestruction;
begin
  if RefCount <> 0 then Error(reInvalidPtr);
end;

function TInterfacedObject.QueryInterface(const IID: TGUID; out Obj): HResult;
const
  E_NOINTERFACE = $80004002;
Begin
  if GetInterface(IID, Obj) then Result := 0 else Result := E_NOINTERFACE;
end;

function TInterfacedObject._AddRef: Integer;

begin
 Result := InterlockedIncrement(FRefCount);
end;

function TInterfacedObject._Release: Integer;
begin
 Result := InterlockedDecrement(FRefCount);
 if Result = 0 then
   Destroy;
end;
```

除了在 InterlockedIncrement 和 InterlockedDecrement 针对线程安全的使用上以及 RefCount 特性上不同外，IUnknown 的类实现几乎与我们刚刚开发的完全一样。然而，这个类的存在意味着可以仅仅从 TInterfacedObject 派生，而不是从 TObject 派生及重复实现 IUnknown。

但是有时可能读者自己想要实现 IUnknown 接口。在本章 1.3.3 节 3.“绕过自动引用计数”中探讨那样做的原因。

1.3.2　创建、使用及销毁接口

编写完类的代码之后，就可以使用代码了，就像使用其它别的类一样。例如：

```
var
  MyNumber: TFormattedInteger;
begin
  MyNumber := TFormattedInteger.Create(12);
  ShowMessage(MyNumber.FormattedString);
  MyNumber.Free;
end;
```

用户可能不明白接口的声明有什么好处，因为它看上去没有提供任何益处。前面的代码也可以写成另外一种相似的形式。下面的代码在功能上和前面的相同。

```
var
  MyNumber: IFormattedNumber;
begin
  MyNumber := TFormattedInteger.Create(12);
  ShowMessage(MyNumber.FormattedString);
end;
```

这段代码中有很多有趣的地方。首先，MyNumber 被定义为 IFormattedNumber 类型而不是 TFormattedInteger 类型。在这个例子中，MyNumber 为指向一个接口的指针，而不是指向一个对象实例的指针。这里又一个非常重要的区别，MyNumber 不访问任何信息，除非在 IFormattedNumber 接口中定义。例如：下面的代码将无法编译通过：

```
MyNumber.SetValue(10);
```

编译器将会显示如下出错信息：

```
Undeclared identified: 'SetValue'
```

任何 IFormattedNumber 类型的变量仅仅能“看到”IFormattedNumber 的声明，即使它被创建为 TFormattedInteger 类型，如前面的代码所示。

另一个值得注意的问题是，这段代码并没有显式地释放 MyNumber，看上去似乎有一个内存漏洞，但实际上，MyNumber 是自动释放的，这需要进一步解释。

接口是被计数的引用

前面提到过，接口是被计数的引用。但作者并没有解释引用计数的工作原理。Delphi 在后台做了许多艰苦的工作，以便让引用计数尽可能地不可见。应当指出的是，对大多数语言，情况并不是这样的。例如，使用 C++（并非 C++ Builder），用户需要手工调用 AddRef 及 Release 来增加或减少引用计数。

前面的部分不是有了排版错误吧？通过查看前面的 TFormattedInteger，发现我们实现了名为_AddRef 及_Release 的函数。现在，正要讨论 AddRef 和 Release。Microsoft 最初将这两个函数命名为 AddRef 和 Release。Borland 公司将函数名前面加上了下划线，以给用户加深这样的印象：不要自己调用这些函数。

在欣赏了 Delphi 为我们所作的一切之后，让我们考虑一下如何在一种特定语言中编写使用接口的代码。在这里，作者将使用 Delphi 语法，因为大家可能对其他的语言并不熟悉，例如：C 或 C++。

```
procedure DoSomethingWithInterfaces;
var
  MyIntf: IFormattedNumber;
begin
  MyIntf := TFormattedInteger.Create(12);
  MyIntf.AddRef;
  ShowMessage(MyIntf.FormattedString);
  MyIntf.Release;

  MyIntf := TFormattedHexInteger.Create(1024);
  MyIntf.AddRef;
  Showmessage(MyIntf.FormattedString);
  MyIntf.Release;
end;
```

一旦得到了一个接口的引用（在代码的第一行），就有责任显式地调用 AddRef 来增加对该接口的一个引用。一旦有人拥有了一个对象接口的引用，该对象就不能被销毁。因此，需要在可能的时候增加引用计数。

在调用了 AddRef 之后，就可以使用该接口来调用接口中的方法。在使用完了接口之后，调用 Release 来减少引用计数。在 Release 函数中，若引用计数降到零，对象就被自动地销毁。

现在看看在 Delphi 中需要做什么来完成同样的工作。

```
procedure DoSomethingWithInterfaces;
var
  MyIntf: IFormattedNumber;
begin
  MyIntf := TFormattedInteger.Create(12);
  ShowMessage(MyIntf.FormattedString);
  MyIntf := TFormattedHexInteger.Create(1024);
  ShowMessage(MyIntf.FormattedString);
end;
```

这就对了，我们不需要_AddRef 和_Release 函数。那么，谁调用了_AddRef 和_ Release 函数呢？更确切地说，它们是如何（及何时）被调用的呢？Delphi 将会在合适的时候自动调用它们。让我们看一下到底 Delphi 在后台做了一些什么事，Delphi 自动加入的代码都被加

上了注释，注意：Delphi 并没有把这些代码加到用户的源代码文件中，这只是一个假想的表达方式，演示了在编译时把什么加入到执行程序中。

```
procedure DoSomethingWithInterfaces;
var
  MyIntf: IFormattedNumber;
begin
  MyIntf := TFormattedInteger.Create(12);
  MyIntf._AddRef; // Delphi 所加
  ShowMessage(MyIntf.FormattedString);
  MyIntf._Release; // Delphi 所加
  MyIntf := TFormattedHexInteger.Create(1024);

  MyIntf._AddRef; // Delphi 所加
  ShowMessage(MyIntf.FormattedString);
  MyIntf._Release; // Delphi 所加
end;
```

顺便说一下，若想强迫销毁一个接口，只需简单地将接口变量赋值为 nil，如下面的代码所示：

```
MyIntf := nil;
```

1.3.3　获取接口的指针

Delphi 提供了获取接口指针的一些方法。可以获得一个接口的指针，该指针给出了实现接口的一个对象，或给出了另一个接口的指针。在本章 1.5“高级接口问题”中，将讨论多个接口。

1. 直接分配

从对象获取接口的最简单的方法是通过直接分配。类是与它们实现的接口类型兼容的，因此可以编写代码如下：

```
var
  MyInteger: TFormattedInteger;
  MyNumber:   IFormattedNumber;
    begin
  MyInteger：= TFormattedInteger.Create(12);
  MyNumber：=MyInteger;
end;
```

因为编译时检查代码，所以如果问题中的代码不支持接口，那么就会产生编译期间错误。尽管有时并不知道对象是否支持一个接口，用户却希望在运行时发现。GetInterface 函数使用户可以这样做。

GetInterface

GetInterface 定义如下：

```
function TObject.GetInterface(const IID: TGUID; out Obj): Boolean;
```