9. 接口

9.1. Bare (CORBA) 接口

*一个接口*声明一个 API，很像一个类，但是它不定义实施（implementation）。一个类可以实施（implement）很多接口，但是它仅能有一个原型（ancestor）类。
你可以强制转换（cast）一个类到任何它支持的接口，然后*通过这个接口调用方法*。这允许以统一的样式处理类，不从彼此衍生，但是仍然共享一些常见的功能。当一个简单的类继承不够用时是有用的。
在 Object Pascal 中 *CORBA 接口*的工作非常像在 Java 中的接口 (https://docs.oracle.com/javase/tutorial/java/concepts/interface.html)或在 C# 中的接口 (https://msdn.microsoft.com/en-us/library/ms173156.aspx).

```
{$mode objfpc}{$H+}{$J-}
{$interfaces corba}

uses SysUtils, Classes;

type
 IMyInterface = interface
 ['{79352612-668B-4E8C-910A-26975E103CAC}']
   procedure Shoot;
 end;

 TMyClass1 = class(IMyInterface)
   procedure Shoot;
 end;

 TMyClass2 = class(IMyInterface)
   procedure Shoot;
 end;

 TMyClass3 = class
   procedure Shoot;
 end;

procedure TMyClass1.Shoot;
begin
 Writeln('TMyClass1.Shoot');
end;

procedure TMyClass2.Shoot;
begin
 Writeln('TMyClass2.Shoot');
```

```
end;

procedure TMyClass3.Shoot;
begin
  Writeln('TMyClass3.Shoot');
end;

procedure UseThroughInterface(I: IMyInterface);
begin
  Write('Shooting... ');
  I.Shoot;
end;

var
  C1: TMyClass1;
  C2: TMyClass2;
  C3: TMyClass3;
begin
  C1 := TMyClass1.Create;
  C2 := TMyClass2.Create;
  C3 := TMyClass3.Create;
  try
    if C1 is IMyInterface then
      UseThroughInterface(C1 as IMyInterface);
    if C2 is IMyInterface then
      UseThroughInterface(C2 as IMyInterface);
    // The "C3 is IMyInterface" below is false,
    // so "UseThroughInterface(C3 as IMyInterface)" will not execute.
    if C3 is IMyInterface then
      UseThroughInterface(C3 as IMyInterface);
  finally
    FreeAndNil(C1);
    FreeAndNil(C2);
    FreeAndNil(C3);
  end;
end.
```

9.2. CORBA 和接口的 COM 类型

为什么(上面提出的)接口被称为"CORBA"?

> 名称 **CORBA** 是不恰当的。一个更好的名称应该是 **bare 接口**。这些接口是一种*"纯语言特色"*。当你想强制转换各种各样的类为相同的接口时使用它们，因为它们共享一些常见的 API 。

然而这些接口的类型可以与 *CORBA (公共对象请求代理体系结构（Common Object Request Broker Architecture）)技术*一起被使用(查看 https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture) ，它们不以任何方式相关联到这个技术。

需要声明{$interfaces corba}吗？

是的，因为在默认情况下，你创建 *COM 接口*。这可以通过表达{$interfaces com}而被明确地公布，但是它通常不需要，因为它是默认公布。

并且，我不建议使用 *COM 接口*，特别是你正在寻找一些等同于来自其它编程语言的接口。在 Pascal 中 *CORBA 接口*正是你期待的东西，如果你正在寻找一些等同于在 C#和 Java 中的接口。与此同时，*COM 接口*引入你可能不想要的附加特色。

注意，这个{$interfaces xxx}声明仅影响没有任何明确的原型(仅关键字 interface ，不是 interface(ISomeAncestor))的接口。当一个接口有一个原型时，它有像原型一样的相同类，不理会{$interfaces xxx}声明。

COM 接口是什么？

*COM 接口*等同于一个*衍生自一个特殊* IUnknown *接口的接口* 。*衍生自* IUnknown ：

- 你的类需要定义_AddRef 和_ReleaseRef 方法。这些方法的正确实施（implementation）可以使用引用计数（the reference-counting）管理你的对象的生命周期。
- 添加 QueryInterface 方法。
- 允许与 *COM (组件对象模型（Component Object Model）) 技术*相互作用。

为什么不建议你使用 COM 接口？

因为 *COM 接口* "缠住"两个特色，在我看来这应该是不相关的(正交的（orthogonal）)：*多重继承和引用计数*。其它编程语言为这两种特色精确地使用独立的概念。

为清晰易懂：**引用计数（refere nce-counting）**，提供一个自动内存管理器(在简单的情况下，例如，无循环)，**是一个非常有用的概念**。但是，**在我看来，用接口 (instead of making them orthogonal features)缠住这个特色不是清晰易懂的**。它当然不匹配我使用的具体情况。

- 有时，我想强制转换我的(其它不相关的)类到一个常用的接口。
- 有时，我想使用引用计数方法管理内存。
- *可能*，将来我想与 *COM 技术*相互作用。

但是这些全部是独立的，不相关的需要。在我看来，在一个单独的语言特色中缠住它们

是有相反作用。它确实引起实际的问题：

- 如果你需要*强制转换类到一个常见的接口 API* 的特色，但是我不需要引用计数机制(我想手动释放对象)，那么 COM 接口是有问题的。甚至当引用计数被通过一个特殊的_AddRef 和_ReleaseRef 实施（implementation）禁用时，在你已经释放类实例后，你仍然需要注意一个未曾临时挂起的接口引用。关于它的更多细节在下一部分。

- 如果你需要*引用计数*的特色，但是我不需要一个接口层次结构来表示一些不同于类层次结构的东西，那么我不得不在接口中重复（duplicate）我的类。以此方式为每个类创建一个简单的接口。这是适得其反的。我更喜欢有*智能指针*作为一个单独的语言特色，而不是与接口 (并且幸好，它来到了：)缠住在一起.

这是为什么我建议在所有现代代码用接口处理中使用 *CORBA 样式*接口，和{$interfaces corba} 指令的原因。

如果你在同一时间中仅需要"引用计数"和"多重继承"，那么使用 COM 接口。此外，Delphi 现在仅有 COM 接口，所以，如果你的代码必需与 Delphi 兼容，你需要使用 COM 接口。

我们能使用接口持有（have）引用计数吗？

能。仅添加_AddRef / _ReleaseRef 方法。这里不需要从 IUnknown 接口衍生。然而在大多数情况下，如果你想使用你的接口引用计数，你也可以只使用 COM 接口。

9.3.接口 GUIDs（全球唯一的标志符）

GUIDs 是看似随机的字符['{ABCD1234-…}']，你可以看到它们被放置在每个接口的定义处。是的，它们只是随机的。令人失望的是，它们是必需的。
如果你不打算与像 *COM* 或 *CORBA* 一样的通信技术集成，GUIDs 没有意义。但是对于实施（implementation）原因，它们是必需的。不要被编译器糊弄，令人失望的是编译器允许你不带有 GUIDs 来声明接口。
不带有(唯一的) GUIDs，你的接口将被等同是操作符对待。实际上，如果你的类支持任何一个你的接口，它将返回 true 。在这里，神奇的函数 Supports(ObjectInstance, IMyInterface)表现稍微更好一点，因为它拒绝编译不带有一个 GUID 的接口。在 FPC 3.0.0 时，对于 CORBA 和 COM 接口是正确的。
所以，为了慎重起见，你应该总是为你的接口声明一个 GUID。你可以使用 *Lazarus* GUID 生成器 (在编辑器中的快捷方式是 Ctrl + Shift + G)。或你可以使用一个在线服务，像 https://www.guidgenerator.com/ 。
或者，你可以为此使用 RTL 中的 CreateGUID 和 GUIDToString 函数写你自己的工具。查看下面的示例：

```
{$mode objfpc}{$H+}{$J-}
uses SysUtils;
var
  MyGuid: TGUID;
```

```
begin
 Randomize;
 CreateGUID(MyGuid);
 Writeln('[''' + GUIDToString(MyGuid) + ''']');
end.
```

9.4. 引用计数(COM)接口

*COM 接口*带来两个附加特色：

1. 与 COM (来自 Windows 的一个技术，也在 Unix 上通过 *XPCOM* 可用，由 Mozilla 使用)集成，
2. 引用计数(当所有的接口引用超出范围时，为你提供自动销毁)。

当使用 *COM 接口*时，你需要知道自动销毁机制和与 COM 技术的关系。
在实践中，这意味着：

- 你的类需要实施（implement）一个神奇的_AddRef ， _Release ， 和 QueryInterface 方法。或衍生自一些已经实施（implements）它们的东西。这些方法的一个特别的实施（implementation）可能事实上启用或禁用 COM 接口(尽管禁用它是有点危险—查看下一个重点)的*引用计数*特色。
  - 标准的类 TInterfacedObject 实施（implements）这些方法来*启用*引用计数。
  - 标准的类 TComponent 实施（implements）这些方法来*禁用*引用计数。**在 Castle Game Engine 中**，为了这个目的，我们给你附加的有用的原型 TNonRefCountedInterfacedObject 和 TNonRefCountedInterfacedPersistent ， 查看 https://github.com/castle-engine/castle-engine/blob/master/src/base/castleinterfaces.pas 。
- 当它可能被一些接口变量引用时，你需要注意 释放类。因为接口是使用虚拟方法(因为它*能被引用计数，即使你非法侵入_AddRef 方法，而不非引用计数...*)发布的，你不能释放底层的对象实例，只要一些借口变量可能指向它。查看， FPC 手册 (http://freepascal.org/docs-html/ref/refse47.html)中的*"7.7 引用计数"*。

使用 *COM 接口*的最安全的方法是

- 接受它们被引用计数的事实，
- 从 acedObject 衍生得到合适的类，
- 并避免使用类实例，反而总是通过接口访问实例，让引用计数管理存储单元分配。

这是这样的接口实例的一个示例：

```
{$mode objfpc}{$H+}{$J-}
{$interfaces com}

uses SysUtils, Classes;

type
 IMyInterface = interface
 ['{3075FFCD-8EFB-4E98-B157-261448B8D92E}']
   procedure Shoot;
 end;

 TMyClass1 = class(TInterfacedObject, IMyInterface)
```

```
    procedure Shoot;
  end;

  TMyClass2 = class(TInterfacedObject, IMyInterface)
    procedure Shoot;
  end;

  TMyClass3 = class(TInterfacedObject)
    procedure Shoot;
  end;

procedure TMyClass1.Shoot;
begin
  Writeln('TMyClass1.Shoot');
end;

procedure TMyClass2.Shoot;
begin
  Writeln('TMyClass2.Shoot');
end;

procedure TMyClass3.Shoot;
begin
  Writeln('TMyClass3.Shoot');
end;

procedure UseThroughInterface(I: IMyInterface);
begin
  Write('Shooting... ');
  I.Shoot;
end;

var
  C1: IMyInterface;  // COM takes care of destruction
  C2: IMyInterface;  // COM takes care of destruction
  C3: TMyClass3;     // YOU have to take care of destruction
begin
  C1 := TMyClass1.Create as IMyInterface;
  C2 := TMyClass2.Create as IMyInterface;
  C3 := TMyClass3.Create;
  try
    UseThroughInterface(C1); // no need to use "as" operator
    UseThroughInterface(C2);
    if C3 is IMyInterface then
      UseThroughInterface(C3 as IMyInterface); // this will not execute
```

```
 finally
   { C1 and C2 variables go out of scope and will be auto-destroyed now.

     In contrast, C3 is a class instance, not managed by an interface,
     and it has to be destroyed manually. }
   FreeAndNil(C3);
 end;
end.
```

9.5. 禁用引用计数使用 COM 接口

如同在前面部分提到，你的类可以从 TComponent (或一个类似的类，像 TNonRefCountedInterfacedObject 和 TNonRefCountedInterfacedPersistent) 衍生，为 COM 界面禁用引用计算。这允许你使用 COM 接口，并且仍然可以手动释放类。
当一些接口变量可能引用它时，在这种情况下，你需要注意不释放类实例。记住，每个类型强制转换 Cx 为 IMyInterface ，也创建一个临时的接口变量，甚至可能到当前的 procedure（过程）结尾时出现。因为这个原因，下面的示例使用一个 UseInterfaces procedure（过程），并且它释放在这个 procedure（过程）(当我们可以确保这个临时接口变量在范围外时)*外部*的类实例。
为避免这种混乱，通常，使用 CORBA 接口更好，如果你不想带有你的接口引用计数 。

```
{$mode objfpc}{$H+}{$J-}
{$interfaces com}

uses SysUtils, Classes;

type
 IMyInterface = interface
 ['{3075FFCD-8EFB-4E98-B157-261448B8D92E}']
   procedure Shoot;
 end;

 TMyClass1 = class(TComponent, IMyInterface)
   procedure Shoot;
 end;

 TMyClass2 = class(TComponent, IMyInterface)
   procedure Shoot;
 end;

 TMyClass3 = class(TComponent)
   procedure Shoot;
 end;

procedure TMyClass1.Shoot;
begin
```

```
  Writeln('TMyClass1.Shoot');
end;

procedure TMyClass2.Shoot;
begin
  Writeln('TMyClass2.Shoot');
end;

procedure TMyClass3.Shoot;
begin
  Writeln('TMyClass3.Shoot');
end;

procedure UseThroughInterface(I: IMyInterface);
begin
  Write('Shooting... ');
  I.Shoot;
end;

var
  C1: TMyClass1;
  C2: TMyClass2;
  C3: TMyClass3;

procedure UseInterfaces;
begin
  if C1 is IMyInterface then
  //if Supports(C1, IMyInterface) then // equivalent to "is" check above
    UseThroughInterface(C1 as IMyInterface);
  if C2 is IMyInterface then
    UseThroughInterface(C2 as IMyInterface);
  if C3 is IMyInterface then
    UseThroughInterface(C3 as IMyInterface);
end;

begin
  C1 := TMyClass1.Create(nil);
  C2 := TMyClass2.Create(nil);
  C3 := TMyClass3.Create(nil);
  try
    UseInterfaces;
  finally
    FreeAndNil(C1);
    FreeAndNil(C2);
    FreeAndNil(C3);
```

```
  end;
end.
```

9.6.不带有"as"操作符类型强制转换接口

这部分适用于 *CORBA* 和 *COM* 接口。
在运行时使用操作符强制转换到一个接口类型进行一次检查。考虑这行代码：

```
UseThroughInterface(Cx as IMyInterface);
```

它适用于先前部分示例中的 C1 ， C2 ， C3 实例。如果执行，在 C3 的情况下可能发生一个运行时错误，它不实施（implement）IMyInterface (但是我们在做强制转换前，通过控制 Cx 是 IMyInterface，避免该错误)。
你可以代替强制转换实例像一个隐式接口一样：

```
UseThroughInterface(Cx);
```

在这种情况下，在编译时，类型强制转换必需是有效的。所以这将对 C1 和 C2 (它们被声明为实施（implement）IMyInterface 的类)编译。但是它将不对 C3 编译。
本质上，这种类型强制转换的样子和工作方式正像常规的类。究竟在哪里要求一个类 TMyClass 的一个实例，你总是可以在一个变量哪里使用，变量使用 TMyClass 的一个类**或** TMyClass **衍生物**声明。相同的规则适用于接口。在这样的情况下，不需要任何显式类型强制转换。
同样，一个相等的东西是

```
UseThroughInterface(IMyInterface(Cx));
```

这也是一种在编译时必需是有效的类型强制转换。注意，这种语法与类类型强制转换不一致。在类的情况下，写 TMyClass(C)是一个*不安全的，不受限制的*类型强制转换。在接口的情况下，写 IMyInterface(C)是一种安全的，敏捷的 (在编译时检查)类型强制转换。

山东世联环保科技开发有限公司

Shandong World United Environmental Protection Technology Development Co.,Ltd.

世联环保官方网站：http://www.cnwut.com

感谢 **山东世联环保科技开发有限公司** 资助 Lazarus 书籍的中文化翻译。

希望大家帮助与支持 **山东世联环保科技开发有限公司** 的发展，

为 **世联环保** 提供业务信息，进而支持 Lazarus 中文化发展！