在CLI这一标准的规范之上，所以它天生就具有了“跨平台”的基因。在微软发布了第一个针对桌面和服务器平台的.NET Framework之后，它开始 “乐此不疲” 地对这个完整版的.NET Framework进行不同范围和层次的 “阉割” ，进而造就了像Windows Phone、Windows Store、Silverlight和.NET Micro Framework的压缩版的.NET Framework。从这个意义上讲，Mono和它们并没有本质的区别，唯一不同的是Mono真正突破了Windows平台的藩篱。包括Mono在内的这些分支促成了.NET的繁荣，但我们都知道这仅仅是一种虚假的繁荣而已。虽然都是.NET Framework的子集，但是由于它们采用完全独立的运行时和基础类库，这使我们很难开发一个支持多种设备的“可移植（Portable）”应用，这些分支反而成为制约.NET发展的一道道枷锁。至于为什么“可移植（Portable）”.NET应用的开发如此繁琐呢？

所谓由于目标框架的独立性，意味着不仅仅是作为虚拟机的[Runtime](http://site3.yhgj8003.com/)是根据具体平台特性设计的，作为编程基础的BCL也不能跨平台共享，它为开发者带来的一个最大的问题就是：很难编写能够在各个目标框架复用的代码。比较极端的场景就是：当我们需要为一个现有的桌面应用提供针对移动设备的支持时，我们不得不从头到尾开发一个全新的应用，现有的代码难以被新的应用所复用用。 “代码复用”是软件设计一项最为根本的目标，在不考虑跨平台的前提下，我们可以应用相应的设计模式和编程技巧来实现代码的重用，但是平台之间的差异导致了跨平台代码重用确实具有不小的困难。虽然作得不算非常的理想，但是微软在这方面确实做出了很多尝试，我们不妨先来聊聊目前我们都有哪些跨平台代码复用的解决方案。

目录  
一、源代码复用  
    源文件共享  
    文件链接  
    共享项目  
二、程序集复用  
    程序集一致性  
    Retargetable程序集  
    类型的转移  
三、可移植类库（PCL）

**一、源代码复用**

对于包括Mono在内的各个.NET Framework平台的BCL来说，虽然在API定义层面上存在一些共同之处，但是由于它们定义在不同的程序集之中，所以在[PCL（Portal Class Library）](http://h5.yhgj8003.com/)推出之前，针对程序集的共享是不可能实现的，我们只能在源代码层面实现共享。源代码的共享通过在不同项目之间共享源文件的方式来实现，至于具体采用的方式，我们有三种不同的方案供你选择。

**源文件共享**

对于一个能够多个针对不同目标框架的项目共享的源文件，定义其中的代码也有不少是针对具体某个目标框架的。对于这种代码，我们需要按照如下的方式进行编写，相应的项目以添加编译的方式选择与自身平台相匹配的代码编译道生成的程序集中。

1: #if WINDOWS

2:     <<针对Windows Desktop>>

3: #elif SILVERLIGHT

4:     <<针对 Silverlight>>

5: #elif WINDOWS\_PHONE

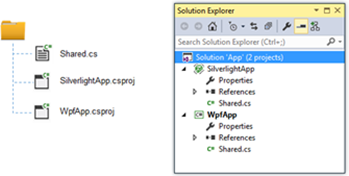
6:     <<针对Windows Phone>>

7: #else

8:     <<针对其他平台>>

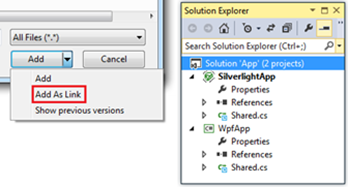
9: #endif

如果多个针对不同.NET Framework平台的项目文件存在于同一个物理目录下，存在于相同目录下的源文件可以同时包含到这些项目中以实现共享的目的。如下图所示，两个分别针对Silverlight和WPF的项目共享相同的目录，与两个项目文件同在一个目录下的C#文件Shared.cs可以同时被包含到这两个项目之中。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171108073639013-1766858859.png)

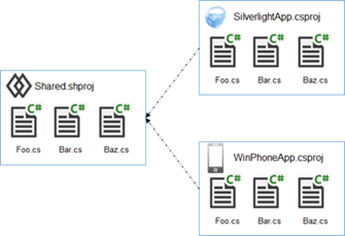
**文件链接**

当我们采用默认的方式将一个现有的文件添加到当前项目之中的时候，Visual Studio会将目标文件拷贝到项目本地的目录下，所以根本起不到共享的目的。但是针对现有文件的添加支持一种叫做“链接”的方式使添加到项目中的文件指向的依然是原来的地址，我们可以为多个项目添加针对同一个文件的链接以实现源文件跨项目共享。同样还是上面演示分别针对Silverlight和WPF的两个项目，不论项目文件和需要被共享的文件存在于哪个目录下面，我们都可以采用如下图所示的添加文件链接的方式分享这个Shared.cs文件。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171108073639341-2088985083.png)

**共享项目（Shared Project）**

普通项目的目的都是组织源文件和其他相关资源并将它们最终编译成一个可被部署的程序集。但是Shared Project这种项目类型则比较特别，它只有对源文件进行组织的功能，却不能通过编译生成程序集，它存在的目的就是为了实现源文件的共享。对于上面我们介绍的两种源代码的共享方式来说，它们都是针对某个单一文件的共享，而Shared Project则可以对多个源文件进行打包以实现批量共享。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171108073639700-189314490.png)

如上图所示，我们可以创建一个Shared Project类型的项目Shared.shproj，并将需要共享的三个C#文件（Foo.cs、Bar.cs和Baz.cs）添加进来。我们将针对这个项目的引用同时添加到一个Silverlight项目（SilverlightApp.csproj）和Windows Phone项目（WinPhoneApp.csproj）之中，当我们对这两个项目实施编译的时候，包含在项目Shared.shproj中的三个C#文件会自动作为当前项目的源文件参与编译。

**二、程序集复用**

我们采用C#、VB.NET这样的编程语言编写的源文件经过编译会生成有IL代码和元数据构成的托管模块，一个或者多个托管模块合并生成一个程序集。程序集的文件名、版本、语言文化和签名的公钥令牌共同组成了它的唯一标识，我们将该标识称为程序集有效名称（Assembly Qualified Name）。除了包含必要的托管模块之外，我们还可以将其他文件作为资源内嵌到程序集中，程序集的文件构成一个“清单（Manifest）”文件来描述，这个清单文件包含在某个托管模块中。

除了作为描述程序集文件构造清单之外，描述程序集的元数据也包含在这个清单文件中。程序集使程序集成为一个自描述性（Self-Describing）的部署单元，除了描述定义在本程序集中所有类型之外，这些元数据还包括对引用自外部程序集的描述。包含在元数据中针对外部程序集的描述是由编译时引用的程序集决定的，引用程序集的名称（包含文件名、版本和签名的公钥令牌）会直接体现在当前程序集的元数据中。针对程序集引用的元数据采用如下的形式（“.assembly extern”）被记录在清单文件中，我们可以看出被记录下来的不仅包含被引用的程序集文件名（“Foo”和“Bar”），还包括程序集的版本，对于签名的程序集（“Foo”）来说，公钥令牌也一并包含其中。

1: .assembly extern Foo

2: {

3:   .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )

4:   .ver 1:0:0:0

5: }

6: .assembly extern Bar

7: {

8:   .ver 1:0:0:0

9: }

包含在当前程序集清单文件中针对引用程序集的元数据是CLR加载目标程序集的依据。在默认的情况下，CLR要求加载与程序集引用元数据完全一致的程序集。具体来说，如果引用的是一个未签名的程序集（“Bar”），那么只要求被加载的程序集具有一致的文件名和版本；如果引用的是一个经过签名的程序集，那么还要求被加载的程序集具有一致的公钥令牌。

在回到《[.NET Core跨平台的奥秘[上篇]：历史的枷锁](http://www.cnblogs.com/artech/p/how-to-cross-platform-01.html)》关于.NET多目标框架独立性的问题。虽然不同的目标框架的BCL在API层面具有很多交集，但是这些API实际上被定义在不同的程序集中，这就导致了在不同的目标框架下共享同一个程序集几乎成了不可能的事情。如果要使跨目标平台程序集复用成为现实，就必须要求CLR在加载程序集时放宽“完全匹配”的限制，因为针对当前程序集清单文件中描述的某个引用程序集来说，在不同的目标框架下可能指向不同的程序集。实际上确实存在这样的一些机制或者策略让CLR加载一个与引用元数据的描述不一致的程序集，我们现在就来聊聊这些策略。

**程序集一致性**

我们都知道.NET Framework是向后兼容的，也就是说原来针对低版本.NET Framework编译生成的程序集是可以直接在高版本CLR下运行的。我们试想一下这么一个问题：就一个针对.NET Framework 2.0编译生成的程序集自身来说，所有引用的基础程序集的版本在元数据描述中都应该是2.0，如果这个程序集在NET Framework 4.0环境下执行，CLR在决定加载它所依赖程序集的时候，应该选择2.0还是4.0呢？

我们不妨通过实验来获得这个问题的答案。我们利用Visual Studio创建一个针对.NET Framework 2.0的控制台应用（命名为App），并在作为程序入口的Main方法上编写如下一段代码。如下面代码片断所示，我们在控制台上输出了三个基本类型（Int32、XmlDocument和DataSet）所在程序集的全名。

1: class Program

2: {

3:     static void Main()

4:     {

5:         Console.WriteLine(typeof(int).Assembly.FullName);

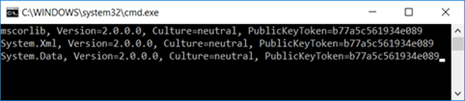
6:         Console.WriteLine(typeof(XmlDocument).Assembly.FullName);

7:         Console.WriteLine(typeof(DataSet).Assembly.FullName);

8:     }

9: }

直接运行这段程序使之在默认版本的CLR（2.0）下运行会在控制台上输出如下的结果，我们会发现上述三个基本类型所在程序集的版本都是2.0.0.0。也就说在这种情况下，运行时加载的程序集和编译时引用的程序集是一致的。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171108073640013-333061392.png)

现在我们在目录“\bin\debug”直接找到以Debug模式编译生成的程序集App.exe，并按照如下的形式修改对应的配置文件（App.exe.config），该配置的目的在于将启动应用时采用的运行时（CLR）版本从默认的2.0切换到4.0。

1: <configuration>

2:   <startup>

3:     <supportedRuntime&nbsp;version="v4.0"/>

4:   </startup>

5: </configuration>

或者：

1: <configuration>

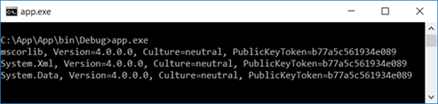
2:   <startup>

3:     <requiredRuntime&nbsp;version="v4.0"/>

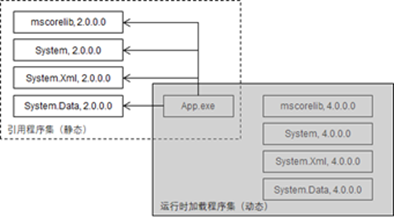
4:   </startup>

5: </configuration>

在无需重新编译（确保运行的依然是同一个程序集）直接运行App.exe，我们会在控制台上得到如下图所示的输出结果，可以看到三个程序集的版本全部变成了4.0.0.0，也就说真正被CLR加载的这些基础程序集是与当前CLR的版本相匹配的。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171108073644653-1301166680.png)

这个简单的实例体现了这么一个特征：运行过程中加载的.NET Framework程序集（承载FCL的程序集）是由当前运行时（CLR）决定的，这些程序集的版本总是与CLR的版本相匹配。包含在元数据中的程序集信息提供目标程序集的名称，而版本则由当前运行的CLR来决定，我们将这个重要的机制称为“程序集一致性（Assembly Unification）”，下图很清晰地揭示了这个特性。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171108073645059-1098109565.png)

**Retargetable程序集**

在默认情况下，如果某个程序集引用了另一个具有强签名的程序集，CLR在执行的时候总是会根据程序集文件名、版本和公钥令牌去定位目标程序集。如果无法找到一个与之完全匹配的程序集，一般情况下会抛出一个FileNotFoundException类型的异常。如果当前引用的是一个Retargetable程序集，则意味着CLR在定位目标程序集的时候可以 “放宽” 匹配的要求，即指要求目标程序集具有相同的文件名即可。

如下图所示，我们的应用程序（App）引用了具有强签名的程序集“Foobar, Version=1.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a”，所以对于编译后生成的程序集App.exe来说，对应的程序集引用将包含目标程序集的文件名、版本和公钥令牌。如果在运行的时候只提供了一个有效名称为“Foobar, Version=2.0.0.0, Culture=neutral, PublicKeyToken=d7fg7asdf7asd7aer”的程序集，除了文件名，后者的版本号和公钥令牌都与程序集引用元数据描述的都不一样。在默认情况下，系统此时总是会抛出一个FileNotFoundException类型的异常，倘若Foobar是一个Retargetable程序集，我们提供的将作为目标程序集被加载并使用。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171108073645309-778761442.png)

除了定义程序集的元数据多了如下一个retargetable标记之外，Retargetable程序集与普通程序集并没有本质区别。

普通程序集：

.assembly Foobar

Retargetable程序集：

.assembly retargetable Foobar

这样一个retargetable标记可以通过按照如下所示的方式在程序集上应用AssemblyFlagsAttribute特性来添加。不过这样的重定向仅仅是针对.NET Framework自身提供的基础程序集有效，虽然我们也可以通过使用AssemblyFlagsAttribute特性为自定义的程序集添加这样一个retargetable标记，但是CLR并不会赋予它重定向的能力。

[assembly:AssemblyFlags(AssemblyNameFlags.Retargetable)]

如果某个程序集引用了一个Retargetable程序集，自身清单文件针对该程序集的引用元数据同样具有如下所示的retargetable标记。CLR正式利用这个标记确定它引用的是否是一个Retargetable程序集，进而确定针对该程序集的加载策略，即采用针对文件名、版本和公钥令牌的完全匹配策略，还是采用只针对文件名的降级匹配策略。

针对普通程序集的引用：

1: 针对普通程序集的引用

2: .assembly extern Foobar

3: {

4:   .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )

5:   .ver 1:0:0:0

6: }

针对Retargetable程序集的引用：

1: .assembly extern retargetable Foobar

2: {

3:   .publickeytoken = (B7 7A 5C 56 19 34 E0 89)

4:   .ver 1:0:0:0

5: }

**类型的转移**

在进行框架或者产品升级过程，我们经常会遇到针对程序集的合并和拆分的场景，比如在新版本中需要对现有的API进行从新规划，可能会将定义在程序集A中定义的类型转移到程序集B中。但是即使发生了这样的情况，我们依然需要为新框架或者产品提供向后兼容的能力，这就需要使用到所谓“类型转移（Type Forwarding）”的特性。

为了让读者朋友们对类型转移这个重要的特性具有一个大体的认识，我们来作一个简单的实例演示。我们利用Visual Studio创建一个针对.NET Framework 3.5的控制台应用App，并在作为程序入口的Main方法中编写了如下两行代码将两个常用的类型（String和Func<>）所在的程序集名打印出来。程序编译之后会在 “\bin\Debug” 目录下生成可执行文件App.exe和对应的配置文件App.exe.config。从如下给出的配置文件内容可以看出.NET Framework 3.5采用的运行时（CLR）版本为 “v2.0.50727” 。

1: class Program

2: {

3:     static void Main()

4:     {

5:         Console.WriteLine(typeof(string).Assembly.FullName);

6:         Console.WriteLine(typeof(Func<>).Assembly.FullName);

7:     }

8: }

App.exe.config

1: <configuration>

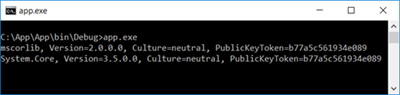
2:   <startup>

3:     <supportedRuntime&nbsp;version="v2.0.50727"/></startup>

4:   </startup>

5: </configuration>

现在我们直接以命令行的执行执行编译生成的App.exe后会在控制台上得到如下图所示的输出结果。可以看出对于我们给出的这两个基础类型（String和Func<>），只有String类型被定义在程序集mscorlib.dll之中，而类型Func<>其实被定义在另一个叫做System.Core.dll的程序集之中。其实Framework 2.0、3.0和3.5不仅仅共享相同的运行时（CLR 2.0），对于提供基础类型的核心程序集mscorlib.dll也是共享的，下图输出的版本信息已经说明了这一点。也就是说，.NET Framework 2.0发布时提供的程序集mscorlib.dll在.NET Framework 3.x时代就没有升级过。Func<>类型是在.NET Framework 3.5发布时提供的一个基础类型，所以不得不将它定义在一个另一个程序集中，微软将这个程序集命令为System.Core.dll。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171108073645606-1613010339.png)

现在我们看看.NET Framework 4.0（CLR 4.0）环境下运行同一个应用程序（App.exe）是否会有不同的输出结果。为此我们在不对项目做重新编译情况下直接修改配置文件App.exe.config，并按照如下所示的方式将运行时版本设置为4.0。

1: <configuration>

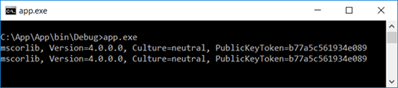
2:   <startup>

3:     <supportedRuntime&nbsp;version="v4.0"/>

4:   </startup>

5: </configuration>

下图是同一个App.exe在.NET Framework 4.0环境下的输出结果，可以看出我们提供的两个基础类型所在的程序集都是mscorlib.dll。也就是当.NET Framework升级到4.0之后，不仅仅运行时升级到了全新的CLR 4.0，微软同时也对承载基础类型的mscorelib.dll程序集进行了重新规划，所以定义在System.Core.dll程序集中的基础类型也基本上又重新回到了mscorlib.dll这个本应该属于它的程序集中。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171108073645934-1372936452.png)

我们来继续分析上面演示的这个程序。由于App.exe这个程序集最初是针对目标框架.NET Framework 3.5编译生成的，所以它的清单文件将包含针对mscorlib.dll（2.0.0.0）和System.Core.dll（3.5.0.0）的程序集引用。下面的代码片段展示了针对这两个程序集引用的元数据的定义。

1: .assembly extern mscorlib

2: {

3:   .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )

4:   .ver 2:0:0:0

5: }

6: .assembly extern System.Core

7: {

8:   .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )

9:   .ver 3:5:0:0

10: }

当App.exe在.NET Framework 4.0环境中运行时，由于它的元数据提供的是针对System.Core.dll程序集的引用，所以CLR总是试图加载该程序集并从中定位目标类型（比如我们演示实例中的类型Func<>）。如果当前运行环境无法提供这个程序集，那么毫无疑问，一个FileNotFoundException类型的异常会被抛出来。也就是，虽然类型Func<>在.NET Framework 4.0中已经转移到了新的程序集mscorlib.dll中，当前环境依然会提供一个文件名为System.Core.dll的程序集。

System.Core.dll存在的目的是告诉CLR它需要加载的类型已经发生转移，并将该类型所在的新的程序集名称告诉它，那么.NET Framework 4.0环境中的System.Core.dll是如何描述类型Func<>已经转移到程序集mscorelib.dll之中了呢？如果分析程序集System.Core.dll中的元数据，我们可以看到如下一段于此相关的代码。在程序集的清单文件中，每一个被转移的类型都对应这个这么一个 “.class extern forwarder” 指令。

1: .class extern forwarder System.Func`1

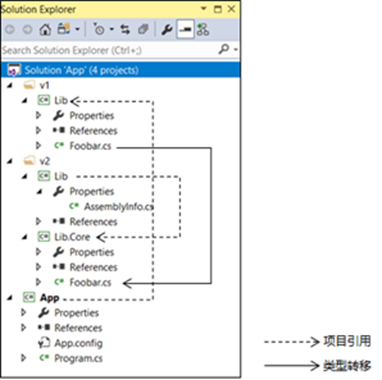
2: {

3:   .assembly extern mscorlib

4: }

不同于上面介绍的Retargetable程序集，类型的转移并不是只针对.NET Framework提供的基础程序集，如果我们自己开发的项目也需要提供类似的向后兼容性，也可以使用这个特性。针对类型转移类型的编程只涉及到一个类型为TypeForwardedToAttribute的特性，接下来我们通过一个简单的实例来演示一下如何利用这个特性将某个类型转移到一个新的程序集中。

我们利用Visual Studio创建了如下图所示的解决方案，它演示了这样一个场景：控制台应用使用到了V1版本的类库Lib（v1\Lib），其中涉及到一个核心类型Foobar。该类库升级到V2版本时，我们选择将所有的核心类型统一定义在新的程序集Lib.Core中，所以类型Foobar需要转移到Lib.Core中。作为类库的发布者，我们希望使用到V1版本的应用能够直接升级到V2版本，也就是升级的应用不需要在引用新的Lib.Core程序集情况下对源代码进行重新编译，而是直接部署V2版本的两个程序集（Lib.dll和Lib.Core）就可以了。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171108073646528-1073742375.png)

上图中的虚线箭头和实线箭头分别代表项目之间的引用关系，我们从中可以看出v2目录下的Lib项目具有对Lib.Core项目的引用，因为它需要引用转移到Lib.Core项目中的类型。为了完成针对类型Foobar的转移，我们只需要在v2\Lib中定义如下一行简单的代码就可以了，我们将这行代码定义在AssemblyInfo.cs文件中。

[assembly: System.Runtime.CompilerServices.TypeForwardedTo(typeof(Lib.Foobar))]

为了检验针对Foobar类型的转移是否成功，我们在控制台应用App中定义了如下一段程序，它负责将Foobar类型当前所在程序集的名称输出到控制台上。接下来我们只需要编译（以Debug模式）整个解决方案，那么V2版本的两个程序集（Lib.dll和Lib.Core.dll）将保存到\v2\lib\bin\debug\目录下。

1: class Program

2: {

3:     static void Main()

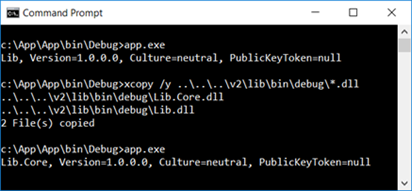
4:     {

5:         Console.WriteLine(typeof(Foobar).Assembly.FullName);

6:     }

7: }

接下来我们采用命令行的形式来运行控制台程序App.exe。如下图所示，我们将当前目录切换到App.exe所在的目录（\app\bin\debug）下并执行App.exe，输出的结果表明Foobar类型当前所在的程序集为Lib.dll。接下来我们将针对V2版本的两个程序集拷贝进来后再次执行App.exe，我们发现此时的Foobar类型已经是从新的程序集Lib.Core.dll中加载的了。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171108073646825-805899003.png)

我们顺便来查看一下V2版本程序集Lib.dll的清单文件的内容。如下面的代码片段所示，在源代码中通过使用TypeForwardedToAttribute特性定义的类型转移在编译之后被转换成了一个“.class extern forwarder”指令。

1: .assembly extern Lib.Core

2: {

3:   .ver 1:0:0:0

4: }

5: .class extern forwarder Lib.Foobar

6: {

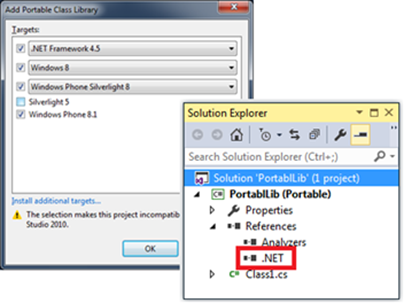
7:   .assembly extern Lib.Core

8: }

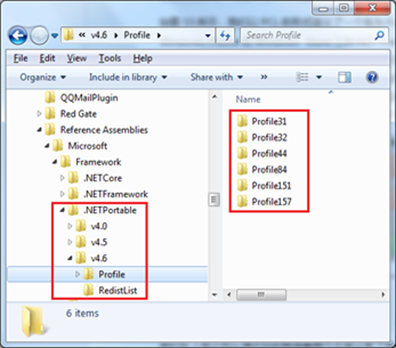
9: …

**三、可移植类库（PCL）**

在.NET Framework的时代，创建可移植类库（PCL：Portable Class Library）是实现跨多个目标框架程序集共享的唯一途径。上面介绍的内容都是在为PCL做铺垫，只有充分理解了Retargetable程序集和类型转移的前提下才可能了解PCL的实现原理有正确的理解。考虑到很多读者朋友并没有使用PCL的经历，所以我们先来介绍一下如何创建一个PCL项目。 当我们采用Visualization Studio的Class Library（Portal）项目模板创建一个PCL项目的时候，需要在如下图所示的对话框中选择支持的目标框架及其版本。Visual Studio会为新建的项目添加一个名为 “.NET” 的引用，这个引用指向一个由选定目标框架决定的程序集列表。由于这些程序集提供的API能够兼容所有选择的平台，我们在此基础编写的程序自然也具有平台兼容性。

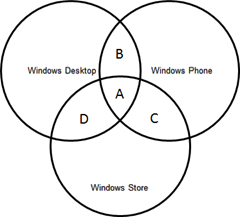
[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171108073647169-148271719.png)

如果查看这个特殊的.NET引用所在的地址，我们会发现它指向目录“%ProgramFiles%\Reference Assemblies\Microsoft\Framework\.NETPortable\{version}\Profile\ProfileX”。如果查看 “%ProgramFiles%\Reference Assemblies\Microsoft\Framework\.NETPortable” 目录，我们会发现它具有如下图所示的结构。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171108073647497-1893348574.png)

如上图所示，本机所在目录“%ProgramFiles%\Reference Assemblies\Microsoft\Framework\.NETPortable”下具有三个代表.NET Framework版本的子目录（v4.0、v4.5和v4.6）。具体到针对某个.NET Framework版本的目录（比如v4.6），其子目录Profile下具有一系列以 “Profile” + “数字” （比如Profile31、Profile32和Profile44等）命名的子目录，实际上PCL项目引用的就是存储在这些目录下的程序集。

对于两个不同平台的.NET Framework来说，它们的BCL在API的定义上存在交集，从理论上来说，建立在这个交集基础上的程序是可以被这两个平台中共享的。如下图所示，如果我们编写的代码需要分别对Windows Desktop/Phone、Windows Phone/Store和Windows Store/Desktop平台提供支持，那么这样的代码依赖的部分仅限于两两的交集A+B、A+C和A+D。如果要求这部分代码能够运行在Windows Desktop/Phone/Store三个平台上，那么它们只能建立在三者之间的交集A上。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171108073647903-1187963400.png)

针对所有可能的目标框架（包括版本）的组合，微软会将作为两者交集的API提取出来并定义在相应的程序集中。比如说所有的目标框架都包含一个核心的程序集mscorlib.dll，虽然定义其中的类型及其成员在各个目标框架不尽相同，但是它们之间肯定存在交集，微软针对不同的目标框架组合将这些交集提取出来并定义在一系列同名程序集中，并同样命名为mscorlib.dll。 微软按照这样的方式创建了其他针对不同.NET Framework平台组合的基础程序集，这些针对某个组合的所有程序集构成一系列的Profile，并定义在上面我们提到过的目录下。值得一提的是，所有这些针对某个Profile的程序集均为Retargetable程序集。

当我们创建一个PCL项目的时候，第一个必需的步骤是选择兼容的目标框架（和版本），Visual Studio会根据我们的选择确定一个具体的Profile，并为创建的项目添加针对该Profile的程序集引用。由于所有引用的程序集是根据我们选择的目标框架组合 “度身定制” 的，所以定义在PCL项目的代码才具有可移植的能力。

上面我们仅仅从开发的角度解释了定义在PCL项目的代码本身为什么能够确保是与目标.NET Framework平台兼容的，但是在运行的角度来看这个问题，却存在额外两个问题：

* 元数据描述的引用程序集与真实加载的程序集不一致，比如我们创建一个兼容.NET Framework 4.5和Silverlight 5.0的PCL项目，被引用的程序集mscorlib.dll的版本为2.0.5.0，但是Silverlight 5.0运行时环境中的程序集mscorlib.dll的版本则为5.0.5.0。
* 元数据描述的引用程序集的类型定义与运行时加载程序集类型定义不一致，比如引用程序集中的某个类型被转移到了另一个程序集中。

由于PCL项目在编译时引用的均为Retargetable程序集，所以程序集的重定向机制帮助我们解决了第一个问题。因为在CLR在加载某个Retargetable程序集的时候，如果找不到一个与引用程序集在文件名、版本、语言文化和公钥令牌完全匹配的程序集，则会只考虑文件名的一致性。至于第二个问题，自然可以通过上面我们介绍的类型转移机制来解决。

综上所述，虽然微软在针对多个目标框架的代码复用上面为我们提供了一些解决方案。在源代码共享方面，我们可以采用共享项目，虽然共享项目能够做到将一组源文件进行打包复用，但是我个人基本上不怎么用它，因为如果我们在其中定义一些公有类型，那么引用该共享项目的项目之间会造成命名冲突。从另一方面讲，我们真正需要的是程序集层面的复用，但是在这方面微软只为我们提供了PCL。PCL这种采用提取目标框架API交集的方式注定了只能是一种临时的解决方案，试着想一下：如果目标框架由10种，每种有3个版本，我们需要为多少种组合创建相应的Profile。对于开发者来说，如果目标框架（包括版本），我们在创建PCL项目进行兼容框架的选择都会成问题。所以我们针对希望的是能够提供给全平台支持的BCL，你可以已经知道了，这就是Net Standard，那么Net Standard是如何能够在多个目标框架中复用的呢？请求关注本系列终结篇《[.NET Core跨平台的奥秘[下篇]：全新的布局](http://www.cnblogs.com/artech/p/how-to-cross-platform-03.html)》。

下载地址：<http://h5.yhgj8003.com/>

文章来源：<http://site3.yhgj8003.com/>