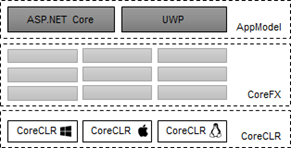
从本质上讲，别无二致。由于采用了统一的中间语言，微软只需要针对不同的平台设计不同的虚拟机（运行时）就能弥合不同操作系统与处理器架构之间的差异，但是“理想很丰满，现实很骨感”。在过去十多年中，微软将.NET引入到了各个不同的应用领域，表面上看起来似乎欣欣向荣，但是由于采用完全独立的多目标框架的设计思路，导致针对多目标框架的代码平台只能通过PCL（参考《[.NET Core跨平台的奥秘[中篇]：复用之殇](http://site3.yhgj8003.com/)》）这种“妥协”的方式来解决。如果依然按照这条道路走下去，.NET的触角延伸得越广，枷锁将越来越多，所以.NET 已经到了不得不做出彻底改变的时刻了。

**一、跨平台的.NET Core**

综上所述，要真正实现.NET 的跨平台伟业，主要需要解决两个问题，一是针对不同的平台设计相应的运行时为中间语言CIL提供一个一致性的执行环境，而是提供统一的BCL以彻底解决代码复用的难题。对于真正跨平台的.NET Core来说，微软不仅为它设计了针对不同平台被成为CoreCLR的运行时，同时还重新设计了一套被称为[CoreFX](http://h5.yhgj8003.com/)的BCL。

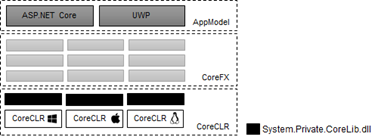
[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171110080710169-2064212392.png)

如上图所示，NET Core目前支持的AppModel主要有两种，其中ASP.NET Core用于开发服务器Web应用和服务，而UWP（Universal Windows Platform）则用于开发能够在各种客户端设备（Mobile、PC、Xbox、Devices + IOT、HoloLens和Surface Hub等）上以自适应方式运行的Windows 10应用。CoreFX是经过完全重写的BCL，除了自身就具有跨平台执行的能力之外，其提供的API也不再是统一定义在少数几个单一的程序集中，而是经过有效分组之后被定义在各自独立的模块中。这些模块对应着一个单一的程序集，并最终由对应的NuGet包来分发。至于底层的虚拟机，微软则为主流的操作系统类型（Windows、Mac OS X和Linux）和处理器架构（x86、x64和ARM）设计了针对性的运行时，被称为CoreCLR。

作为运行时的CoreCLR和提供BCL的CoreFX是.NET Core两根重要的基石，但是就开发成本来看，微软在后者投入的精力是前者无法比拟的。我们知道.NET Core自诞生到现在已经有好些年了，目前的版本还只是到了2.0，从发布进度上显得稍显缓慢，其中一个主要的原因是：重写CoreFX提供的基础API确实是一件繁琐耗时的工程，而且这项工程远未结束。为了对CoreFX提供的BCL有一个大致的了解，我们看看这些常用的基础API究竟定义在哪些命名空间下。

* System.Collections：定义了我们常用的集合类型。
* System.Console：提供API完成基本的控制台操作。
* System.Data：提供用于访问数据库的API，相当于原来的ADO.NET。
* System.Diagnostics：提供基本的诊断、调试和追踪的API。
* System.DirectoryServices：提供基于AD（Active Directory）管理的API。
* System.Drawing：提供GDI相关的API。
* System.Globalization：提供API实现多语言以及全球化支持。
* System.IO：提供针对文件输入输出相关的API。
* System.Net：提供与网络通信相关的API。
* System.Reflection：提供API以实现与反射相关的操作。
* System.Runtime：提供与运行时相关的一些基础类型。
* System.Security：提供与数据签名和加解密相关的API。
* System.Text：提供针对字符串/文本编码与解码相关的API。
* System.Threading：提供用于管理线程的API。
* System.Xml：提供API用以操作XML结构的数据。

我们知道对于传统的.NET Framework来说，承载BCL的API几乎都定义在mscorlib.dll这个程序集中，这些API并不是全部都转移到组成CoreFX的众多程序集中，那些与运行时（CoreCLR）具有紧密关系的底层API被定义到一个叫做System.Private.CoreLib.dll的程序集中，所以下图反映了真正的.NET Core层次结构。我们在编程过程中使用的基础数据类型基本上都定义在这个程序集中，所以目前这个程序集的尺寸已经超过了10M。由于该程序集提供的API与运行时关联较为紧密，较之CoreFX提供的API，这些基础API具有较高的稳定性，所以它是随着CoreCLR一起发布的。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171110080710700-787519540.png)

虽然我们编程过程中使用到的绝大部分基础类型都定义在System.Private.CoreLib.dll程序集中，但是这却是一个“私有”的程序集，我们可以从其命名看出这一点。我们将System.Private.CoreLib.dll称为一个私有程序集，并不是说定义其中的都是一些私有类型，而是因为我们在编程的过程不会真正引用这个程序集，这与.NET Framework下的mscorlib.dll是不一样的。不仅如此，当我们编写的.NET Core代码被编译的时候，编译器也不会链接到这个程序集上，也就是说编译后生成的程序集中同样也没有针对该程序集引用的元数据。但是当我们的应用被真正执行的时候，所有引用的基础类型全部会自动 “转移” 到这个程序集中。至于如何实现运行过程中的类型转移，其实就是利用了我们上面介绍的Type Forwarding技术。

**实例演示：针对System.Private.CoreLib.dll程序集的类型转移**

对上面介绍的针对System.Private.CoreLib.dll程序集的类型转移，可能很多人还是难以理解，为了让大家对这个问题具有彻底的认识，我们不妨来做一个简单的实例演示。我们利用Visual Studio创建一个.NET Core控制台应用，并在作为程序入口的Main方法中编写如下几行代码，它们会将我们常用的几个数据类型（System.String、System.Int32和System.Boolean）所在的程序集名称打印在控制台上。

1: class Program

2: {

3:     static void Main()

4:     {

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

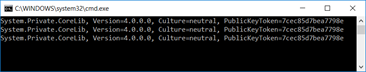
6:         Console.WriteLine(typeof(int).Assembly.FullName);

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

8:     }

9: }

根据我们上面的分析，程序运行过程中使用到的这些基础类型全部来源于System.Private.CoreLib.dll这个程序集中，关于这一点在如下图所示的输出结果中得到了证实。我们通过图2-24所示的输出结果，我们不仅仅知道了这个核心程序集的名称，还知道了该程序集目前的版本（4.0.0.0）；

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171110080711028-454745438.png)

我们说应用编译后生成的程序集并不会具有针对System.Private.CoreLib.dll程序集引用的元数据，为了证明这一点，我们只需要利用Windows SDK（在目录“%ProgramFiles(x86)%Microsoft SDKs\Windows\{version}\Bin”下）提供的反编译工具ildasm.exe就可以了。利用ildasm.exe打开这个控制台应用编译后生成的程序集之后，我们会发现它具有如下这两个程序集的应用。

1: .assembly extern System.Runtime

2: {

3:   .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )

4:   .ver 4:2:0:0

5: }

6: .assembly extern System.Console

7: {

8:   .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )

9:   .ver 4:1:0:0

10: }

实际上我们的程序只涉及到四个类型，即一个Console类型和三个基础数据类型（String、Int32和Boolean），而程序集层面则只有针对System.Runtime和System.Console程序集的引用，那么毫无疑问，后面这三个数据类型肯定与System.Runtime程序集有关，那么该程序集针对这三个数据类型具有怎样的定义呢？为了得到答案，我们先得知道这个程序集究竟被保存在哪里。我们知道“%ProgramFiles%dotnet\”是.NET Core的应用根目录，而这个System.Runtime.dll作为“共享”程序集被保存在子目录“\shared\Microsoft.NETCore.App\2.0.0”下面，这个目录下面还保存着很多其他的共享程序集。

我们依然利用反编译工具ildasm.exe查看System.Runtime.dll程序集清单文件的元数据定义。我们会发现整个程序集除了定义少数几个核心类型（比如两个重要的委托类型Action和Func就定义在这个程序集中），它的作用就是将所有基础的类型采用Type Forwarding的方式转移到System.Private.CoreLib.dll程序集中，下面的代码片段为你展示了针对我们程序使用的三个基础数据类型转移的相关定义。

1: .assembly extern System.Private.CoreLib

2: {

3:   .publickeytoken = (7C EC 85 D7 BE A7 79 8E )

4:   .ver 4:0:0:0

5: }

6: .class extern forwarder System.String

7: {

8:   .assembly extern System.Private.CoreLib

9: }

10: .class extern forwarder System.Int32

11: {

12:   .assembly extern System.Private.CoreLib

13: }

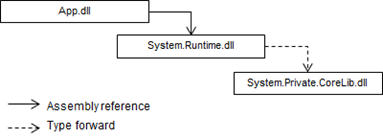
14: .class extern forwarder System.Boolean

15: {

16:   .assembly extern System.Private.CoreLib

17: }

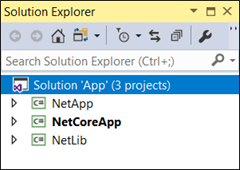
我们演示实例体现的程序集直接的引用关系，以及如上代码片段体现的相关基础类型（System.String、System.Int32和System.Boolean）的转移方向基本体现在如下图所示的关系图中。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171110080711356-1251712273.png)

**复用.NET Framework程序集**

我们将上述这种利用Type Forwarding方式实现跨程序集类型转移的技术成为“垫片（Shim）”，这是实现程序集跨平台复用的重要手段。除了System.Runtime.dll，.NET Core还提供了其他一些其他垫片程序集，正是源于这这些垫片程序集的存在，我们可以将在.NET Framework环境下编译的程序集在.NET Core应用中使用。为了让读者朋友们对此有深刻的认识，我们照例来做一个简单的实例演示。

我们利用Visual Studio创建一个空的解决方案，并添加如下三个项目（NetApp、NetCoreApp、NetLib），其中NetApp和NetCoreApp分别是针对.NET Framework（4.7）和.NET Core（2.0）的控制台程序，而NetLib则是针对.NET Framework的类库项目，该项目定义的API将在NetApp和NetCoreApp被调用。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171110080711622-477062668.png)

我们在NetLib项目中定义了一个Utils工具类，并在其中定义了一个PrintAssemblyNames方法。如下面的代码片段所示，我们在这个方法中打印出三个常用的类型（Task、Uri和XmlWriter）所在的程序集的名称。通过在不同类型（.NET Framework和.NET Core）的应用中调用这个方法，我们就可以确定它们在运行时究竟是从那个程序集中加载的。我们分别在NetApp和NetCoreApp这两个不同类型的控制台程序中调用了这个方法。

**NetLib:**

1: public class Utils

2: {

3:     public static void PrintAssemblyNames()

4:     {

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

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

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

8:     }

9: }

**NetApp:**

1: class Program

2: {

3:     static void Main()

4:     {

5:         Console.WriteLine(".NET Framework 4.7");

6:         Utils.PrintAssemblyNames();

7:     }

8: }

**NetCoreApp:**

1: class Program

2: {

3:     static void Main()

4:     {

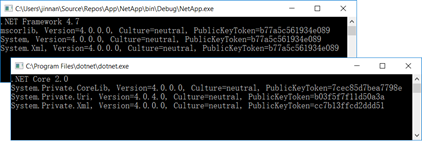
5:         Console.WriteLine(".NET Core 2.0");

6:         Utils.PrintAssemblyNames();

7:     }

8: }

直接运行NetApp和NetCoreApp这两个控制台程序后，我们会发现不同的输出结果。如下图所示，对于我们指定的三个类型（System.Threading.Tasks.Task、System.Uri和System.Xml.XmlWriter），分别在.NET Framework和.NET Core环境下承载它们的程序集是不同的。具体来说，.NET Framework环境下的这三个类型分别定义在mscorlib.dll、System.dll和System.Xml.dll中；当切换到.NET Core环境下后，运行时则会从三个私有的程序集System.Private.CoreLib.dll、System.Private.Uri.dll和System.Private.Xml.dll中加载这三个类型。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171110080711950-1191895934.png)

由于NetApp和NetCoreApp这两个控制台应用使用的都是同一个针对.NET Framework编译的程序集NetLib.dll，所以我们先利用反编译工具ildasm.exe查看一下它具有怎样的程序集引用。如下面的代码片段所示，程序集NetLib.dll引用的程序集与控制台应用NetApp的输出结果是一致的。

1: .assembly extern mscorlib

2: {

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

4:   .ver 4:0:0:0

5: }

6: .assembly extern System

7: {

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

9:   .ver 4:0:0:0

10: }

11: .assembly extern System.Xml

12: {

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

14:   .ver 4:0:0:0

15: }

那么我们的核心问题变成了：Task、Uri和XmlWriter这三个类型在.NET Core的运行环境下是如何转移到其他程序集中的。要回答这个问题，我们只需要利用ildasm.exe查看mscorlib.dll、System.dll和System.Xml.dll反编译这三个程序集就可以了。这三个程序集同样存在于“%ProgramFiles%dotnet\\shared\Microsoft.NETCore.App\2.0.0”目录下，通过反编译与它们相关的程序集，我们得到如下所示的相关元数据。

**mscorlib.dll**

1: .assembly extern System.Private.CoreLib

2: {

3:   .publickeytoken = (7C EC 85 D7 BE A7 79 8E )

4:   .ver 4:0:0:0

5: }

6: .class extern forwarder System.Threading.Tasks.Task

7: {

8:   .assembly extern System.Private.CoreLib

9: }

**System.dll**

1: .assembly extern System.Private.Uri

2: {

3:   .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )

4:   .ver 4:0:4:0

5: }

6: .class extern forwarder System.Uri

7: {

8:   .assembly extern System.Private.Uri

9: }

**System.Xml.dll**

1: .assembly extern System.Xml.ReaderWriter

2: {

3:   .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )

4:   .ver 0:0:0:0

5: }

6: .class extern forwarder System.Xml.XmlWriter

7: {

8:   .assembly extern System.Xml.ReaderWriter

9: }

**System.Xml.ReaderWriter.dll**

1: .assembly extern System.Private.Xml

2: {

3:   .publickeytoken = (CC 7B 13 FF CD 2D DD 51 )

4:   .ver 4:0:0:0

5: }

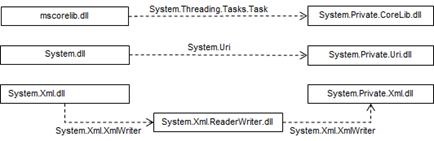
6: .class extern forwarder System.Xml.XmlWriter

7: {

8:   .assembly extern System.Private.Xml

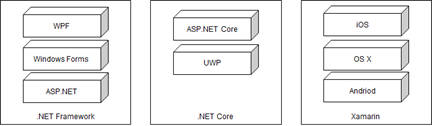
9: }

如上面的代码片段所示，针对Task、Uri和XmlWriter这三个类型的转移一共涉及到七个程序集，其中mscorlib.dll、System.dll和System.Xml.dll是NetLib.dll直接引用的三个程序集，而System.Private.CoreLib.dll、System.Private.Uri.dll和System.Private.Xml.dll则是最终承载这三个类型的程序集。对于Task和Uri类型来说，它们只经历一次转移，而XmlWriter则经历了两次类型转移，它转移到程序集System.Xml.ReaderWriter.dll中，再借助后者转移到目标程序集System.Private.Xml.dll，程序集引用和类型转移关系体现在下图中。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171110080712278-1133595246.png)

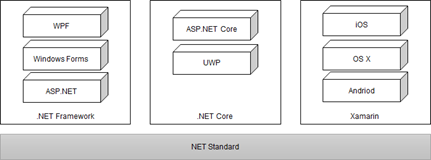
**二、多平台复用的BCL**

虽然.NET Core借助于CoreCLR和CoreFX实现了真正的跨平台，但是目前的.NET Core仅仅提供ASP.NET Core和UWP这两种编程模型，虽然后者旨在实现多种设备的统一编程，但依然还是关注于Windows平台。对于传统.NET Framework下面向桌面应用的WPF和Windows Forms，它们并没有跨平台的意义，所以依然是今后.NET的一大分支。除此之外，虽然我们有了跨平台的ASP.NET Core，传统的ASP.NET依然被保留了下来，并且在今后一段时间内还将继续升级。除了.NET Framework和.NET Core，.NET还具有另一个重要的分支，那就是Xamarin，它可以帮助我们为iOS、OS X和Android编写统一的应用。在.NET诞生十多年后，微软开始对.NET进行了全新的布局，建立了 “大一统” 的.NET平台。总的来说，这个所谓的大一统.NET平台由如下图所示的.NET Framework、.NET Core和Xamarin这三个分支组成。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171110080712809-415015493.png)

虽然被微软重新布局的.NET平台只包含了三个分支，但是之前遇到的一个重要的问题依然存在，那就是代码的复用，说的更加具体的是应该是程序集的复用而不是源代码的复用。我们知道之前解决程序集服务的方案就是PCL，但这并不是一种理想的解决方案，由于各个目标框架具有各种独立的BCL，所以我们创建的PCL项目只能建立在指定的几种兼容目标框架的BCL交集之上。对于全新的.NET平台来说，这个问题通过提供统一的BCL得到根本的解决，这个统一的BCL被称为.NET Standard。

我们可以将.NET Standard称为新一代的PCL，PCL提供的可移植能力仅仅限于创建时就确定下来的几种目标平台，但是.NET Standard做得更加彻底，因为它在设计的时候就已经考虑针对三大分支的复用。如下图所示，.NET Standard为.NET Framework、.NET Core和Xamarin提供了统一的API，那么我们在这组标准API基础上编写的代码自然就能被所有类型的.NET应用复用。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171110080713247-1120334375.png)

.NET Standard提供的API主要是根据现有.NET Framework来定义的，它的版本升级反映了其提供的API不断丰富的过程，目前最新版本（.NET Standard 2.0）提供的API数量在前一版本基础上几乎翻了一番。Visual Studio提供相应的项目模板帮助我们创建基于.NET Standard的类库项目，这样的项目会采用专门的目标框架别名netstandard{version}。一个针对.NET Standard 2.0的类库项目具有如下的定义，我们可以看到它采用的目标框架别名为 “.NET Standard 2.0” 。

1: <Project Sdk="Microsoft.NET.Sdk">

2:   <PropertyGroup>

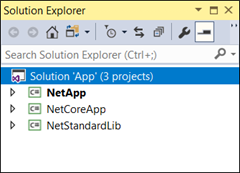
3:     <TargetFramework>netstandard2.0</TargetFramework>

4:   </PropertyGroup>

5: </Project>

顾名思义，.NET Standard仅仅是一个标准，而不提供具体的实现。我们可以简单理解为.NET Standard为我们定义了一整套标准的接口，各个分支需要针对自身的执行环境对这套接口提供实现。对于.NET Core来说，它的基础API主要由CoreFX和System.Private.CoreLib.dll这个核心程序集来承载，这些API基本上就是根据.NET Standard来设计的。但是对.NET Framework来说，它的BCL提供的API与.NET Standard存在着很大的交集，实际上.NET Standard基本上就是根据.NET Framework现有的API来设计的，所以微软不可能在.NET Framework上重写一套类型于CoreFX的实现，只需要采用某个技术 “链接” 到现有的程序集上就可以了。

一个针对.NET Standard编译生成的程序集在不同的执行环境中针对真正提供实现的程序集的所谓“链接”依然是通过上面我们介绍的“垫片”技术来实现的，为了彻底搞清楚这个问题，我们还是先来作一个简单的实例演示。如下图所示，我们创建了与上面演示实例具有类似结构的解决方案，与之不同的是，分别针对.NET Framework和.NET Core的控制台应用NetApp和NetCoreApp共同引用的类库NetStandardLib是一个.NET Standard 2.0类库项目。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171110080714309-1378018006.png)

与上面演示的实例一样，我们在NetStandardLib中定义了如下一个Utils类，并利用定义其中的静态方法PrintAssemblyNames数据两个数据类型（Dictionary<,>和SortedDictionary<,>）所在的程序集名称，该方法分别在NetApp和NetCoreApp的入口Main方法中被调用。

**NetStandardLib:**

1: public class Utils

2: {

3:     public static void PrintAssemblyNames()

4:     {

5:         Console.WriteLine(typeof(Dictionary<,>).Assembly.FullName);

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

7:     }

8: }

**NetApp:**

1: class Program

2: {

3:     static void Main()

4:     {

5:         Console.WriteLine(".NET Framework 4.7");

6:         Utils.PrintAssemblyNames();

7:     }

8: }

**NetCoreApp:**

1: class Program

2: {

3:     static void Main()

4:     {

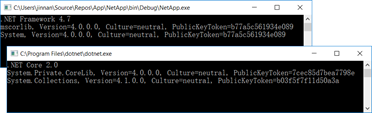
5:         Console.WriteLine(".NET Core 2.0");

6:         Utils.PrintAssemblyNames();

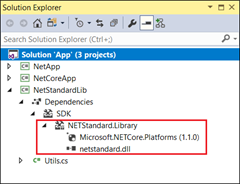
7:     }

8: }

直接运行这两个分别针对.NET Framework和.NET Core的控制台应用NetApp和NetCoreApp，我们会发现它们会生成不同的输出结果。如下图所示，在.NET Framework和.NET Core 执行环境下，Dictionary<,>和SortedDictionary<,>这另个泛型字典类型其实来源于不同的程序集。具体来说，我们常用的Dictionary<,>类型在.NET Framework 4.7和.NET Core 2.0环境下分别定义在程序集mscorlib.dll和System.Private.CoreLib.dll中，而SortedDictionary<,>所在的程序集则分别是System.dll和System.Collection.dll。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171110080714763-364773834.png)

对于演示的这个实例来说，这个NetStandardLib类库项目针对的目标框架为.NET Standard 2.0，后者最终体现为一个名为NetStandard.Library.nupkg的NuGet包，这一点其实可以从Visual Studio针对该项目的依赖节点可以看出来。如下图所示，这个名为NetStandard.Library的NuGet包具有一个核心的程序集netstandard.dll，上面我们所说的.NET Standard API就定义在该程序集中。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171110080715138-227409411.png)

也就是说，所有.NET Standard 2.0项目都具有针对程序集netstandard.dll的依赖，这个依赖自然也会体现在编译后生成的程序集上。对于我们演示实例中的这个类库项目NetStandardLib编译生成的同名程序集来说，它针对程序集netstandard.dll的依赖体现在如下所示的元数据中。

1: .assembly extern netstandard

2: {

3:   .publickeytoken = (CC 7B 13 FF CD 2D DD 51 )

4:   .ver 2:0:0:0

5: }

6: .assembly NetStandardLib

7: {

8:   ...

9: }

10: ...

按照我们即有的知识，原本定义在netstandard.dll的两个类型（Dictionary<,>和SortedDictionary<,>）在不同过的执行环境中需要被转移到另一个程序集中，我们完全可以在相应的环境中提供一个同名的垫片程序集并借助类型的跨程序集转移机制来实现，实际上微软也就是这么做的。我们先来看看针对.NET Framework的垫片程序集netstandard.dll的相关定义，我们可以直接在NetApp编译的目标目录中找到这个程序集。借助于反编译工具ildasm.exe，我们可以很容易地得到与Dictionary<,>和SortedDictionary<,>这两个泛型字典类型转移的相关元数据，具体的内容下面的代码片段所示。

1: .assembly extern mscorlib

2: {

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

4:   .ver 0:0:0:0

5: }

6: .assembly extern System

7: {

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

9:   .ver 0:0:0:0

10: }

11: .class extern forwarder System.Collections.Concurrent.ConcurrentDictionary`2

12: {

13:   .assembly extern mscorlib

14: }

15: .class extern forwarder System.Collections.Generic.SortedDictionary`2

16: {

17:   .assembly extern System

18: }

针对.NET Core的垫片程序集netstandard.dll被保存在我们前面提到的共享目录“%ProgramFiles%dotnet\shared\Microsoft.NETCore.App\2.0.0”下，我们采用同样的方式提取出与Dictionary<,>和SortedDictionary<,>这两个泛型字典类型转移的元数据。从如下的代码片段我们可以清晰地看出，Dictionary<,>和SortedDictionary<,>这两个类型都被转移到程序集System.Collections.dll之中。

1: .assembly extern System.Collections

2: {

3:   .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )

4:   .ver 0:0:0:0

5: }

6: .class extern forwarder System.Collections.Generic.Dictionary`2

7: {

8:   .assembly extern System.Collections

9: }

10: .class extern forwarder System.Collections.Generic.SortedDictionary`2

11: {

12:   .assembly extern System.Collections

13: }

从演示实例的执行结果我们知道，SortedDictionary<,>确实是定义在程序集System.Collections.dll中，但是我们常用的Dictionary<,>类型则出自核心程序集System.Private.CoreLib.dll，那么我们可以断定Dictionary<,>类型在System.Collections.dll中必然出现了二次转移。为了确认我们的断言，我们只需要采用相同的方式反编译程序集System.Collections.dll，该程序集也被存储在共享目录 “%ProgramFiles%dotnet\shared\Microsoft.NETCore.App\2.0.0” 中，该程序集中针对Dictionary<,>类型的转移体现在如下所示的元数据中。

1: .assembly extern System.Private.CoreLib

2: {

3:   .publickeytoken = (7C EC 85 D7 BE A7 79 8E )

4:   .ver 4:0:0:0

5: }

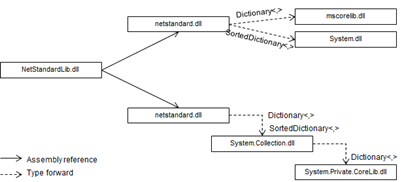
6: .class extern forwarder System.Collections.Generic.Dictionary`2

7: {

8:   .assembly extern System.Private.CoreLib

9: }

上面针对Dictionary<,>和SortedDictionary<,>这两个类型分别在.NET Framework 4.7和.NET Core环境下的跨程序集转移路径基本上体现在下图之中。简单来说，.NET Framework环境下的垫片程序集netstandard.dll将这两个类型分别转移到了程序集mscorlib.dll和System.dll之中。如果执行环境切换到了.NET Core，这两个类型先被转移到System.Collection.dll中，但是Dictionary<,>这个常用类型最终是由System.Private.CoreLib.dll这个基础程序集承载的，所有System.Collection.dll中针对该类型作了二次转移。

[](http://images2017.cnblogs.com/blog/19327/201711/19327-20171110080715622-565495374.png)

上面这个简单的类型基本上揭示了.NET Standard为什么能够提供全平台的可移植性，我们现在来对此做一个简单的总结。.NET Standard API由NetStandard.Library这个NuGet包来承载，后者提供了一个名为netstandard.dll的程序集，保留在这个程序集中的仅仅是. NET Standard API的存根（Stub），而不提供具体的实现。所有对于一个目标框架为.NET Standard的类库项目编译生成的程序集来说，它们保留了针对程序集netstandard.dll的引用。

.NET平台的三大分支（.NET Framework、.NET Core和Xamarin）按照自己的方式各自实现了.NET Standard规定的这套标准的API。由于在运行时真正承载.NET Standard API的类型被分布到多个程序集中，所以. NET Standard程序集能够被复用的前提是运行时能够将这些基础类型链接到对应的程序集上。由于. NET Standard程序集是针对netstandard.dll进行编译的，所以我们只需要在各自环境中提供这个同名的程序集来完成类型的转移即可。

下载地址：<http://h5.yhgj8003.com/>

文章来源：<http://site3.yhgj8003.com/>