包、元包和框架

2016/06/20 • +1

本文内容

.NET Core 是一种由 NuGet 包组成的平台。 有些产品体验受益于包的细粒度定义，而另一些受益于粗粒度的定义。 为了适应这种二元定义，一款好的产品应该作为一组细粒度的包发布，并在更粗的粒度组块中进行分发，单个包的正式的名字叫做元包。

每个 .Net Core 包都支持以框架形式通过多个 .Net 实现代码运行。 其中有些框架是传统框架，例如表示 .NET Framework 的 `net46`。 而另一些则是新框架，可视为是“基于包的框架”，这种是框架的另外一种新的定义模型。 这些基于包的框架整个都是由包组成的，它们自身也被定义成包，这就在包与框架之间形成了一种比较密切的关系。

package

.NET Core 被分成一组包，它们提供基元类型、更高级的数据类型、应用组合类型和通用实用工具。 每一个包都代表着单独的同名程序集。 例如，System.Runtime 包包含 System.Runtime.dll。

以细粒度方式定义这些包具有以下好处：

- 细粒度的包可以在它自己的计划内交付，只需完成仅对相关的其他有限的包进行测试即可。
- 细粒度的包可以提供不同的 OS 和 CPU 支持。
- 细粒度的包可以单独依赖于某一个库。
- 应用可以变得更小，因为没有引用的包不会变成应用发行的一部分。

上述某些好处只适用于某些特定场合。 例如，.NET Core 的所有包通常都会在同一计划内提供对同一平台的支持。 在这种情况下，补丁与更新会以小的单独包的形式发布和安装。 由于这种小范围的变化，补丁的验证与时间花费，都可以限制到单个库的需求范围中。

以下是 .NET Core 重要的 NuGet 包列表：

- System.Runtime - 最基础的 .NET Core 包，包括 Object、String、Array、Action 和 IList<T>。

- System.Collections - 一组（主要）泛型集合，包括 List<T> 和 Dictionary<TKey,TValue>。
- System.Net.Http - 一组用于 HTTP 网络通信的类型，包括 HttpClient 和 HttpResponseMessage。
- System.IO.FileSystem - 一组用于读写到本地或网络磁盘存储的类型，包括 File 和 Directory。
- System.Linq - 一组用于查询对象的类型，包括 `Enumerable` 和 ILookup<TKey,TElement>。
- System.Reflection - 一组用于加载、检查和激活类型的类型，包括 Assembly、TypeInfo 和 MethodInfo。

通常，包含元包要比包含各个包更加简单可靠。 但是当需要单个包时，可以按以下示例所示的那样来包含它，此示例引用 System.Runtime 包。

XML 复制

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard1.6</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="System.Runtime" Version="4.3.0" />
  </ItemGroup>
</Project>
```

元包

元包就是一个 NuGet 包约定，描述了一组意义相关的包。 开发团队利用依赖项来描述这一组包。 他们通过这一组包来描述一个框架，然后有选择地发布出去。

默认情况下，早期版本的 .NET Core 工具（同时基于 project.json 和 csproj 的工具）指定一个框架和一个元包。 但目前，由目标框架隐式引用元包，以便将每个元包绑定到一个目标框架。 例如，`netstandard1.6` 框架引用 NetStandard.Library 1.6.0 版元包。 同样，`netcoreapp2.1` 框架引用 Microsoft.NETCore.App 2.1.0 版元包。 有关详细信息，请参阅 .NET Core SDK 中的隐式元包引用。

以某个框架为目标以及隐式引用元包，这实际上是添加了对元包中每一个独立包的引用依赖。 这使这些包中的所有库都可用于 IntelliSense（或类似体验），同时也可用于发布应用。

使用元包具有以下好处：

- 在引用大量细粒度包方面，提供了一种方便的用户体验。
- 定义了一组经过充分测试且运行良好的包（包括指定的各种版本）。

.NET Standard 元包为：

- NETStandard.Library - 描述了属于".NET Standard"一部分的各种库。 适用于所有支持 .NET Standard 的 .NET 实现（例如，.NET Framework、.NET Core 和 Mono）。 也就是"netstandard"框架。

重要的 .NET Core 元包有：

- Microsoft.NETCore.App - 描述了属于 .NET Core 发行版的部分库。 也就是 .NETCoreApp 框架。 它依赖于更小的 `NETStandard.Library`。
- Microsoft.AspNetCore.App - 包含来自 ASP.NET Core 和 Entity Framework Core 的所有受支持的包（包含第三方依赖项的包除外）。 有关详细信息，请参阅 ASP.NET Core 的 Microsoft.AspNetCore.App 元包。
- Microsoft.AspNetCore.All - 包含来自 ASP.NET Core、Entity Framework Core 以及 ASP.NET Core 和 Entity Framework Core 使用的内部和第三方依赖项的所有受支持包。 有关详细信息，请参阅 ASP.NET Core 2.x 的 Microsoft.AspNetCore.All 元包。
- Microsoft.NETCore.Portable.Compatibility - 一组兼容外观，使基于 mscorlib 的可移植类库(PCL) 得以在 .Net Core上运行。

框架

每个 .NET Core 包支持一组运行时框架。 框架描述了一组可用的 API（以及潜在的其他特性），所以你可以在指定一个目标框架时使用这些功能。 添加新的 API 时，它们就会进入版本控制流程。

例如，System.IO.FileSystem 支持以下框架：

- .NETFramework,Version=4.6
- .NETStandard,Version=1.3
- 6 种 Xamarin 平台（例如，xamarinios10）

将前两个框架进行对比很有帮助，因为它们各自代表了一种不同的框架定义方式。

`.NETFramework,Version=4.6` 框架表示 .NET Framework 4.6 中可用的 API。 你可以生成使用 .NET Framework 4.6 引用程序集编译的库，并以NuGet 包的方式在 net46 lib 文件夹中发布这些库。 这样，你的库就会被那些基于或者兼容 .Net Framework 4.6 的应用所使用。 这是所有框架的传统工作原理。

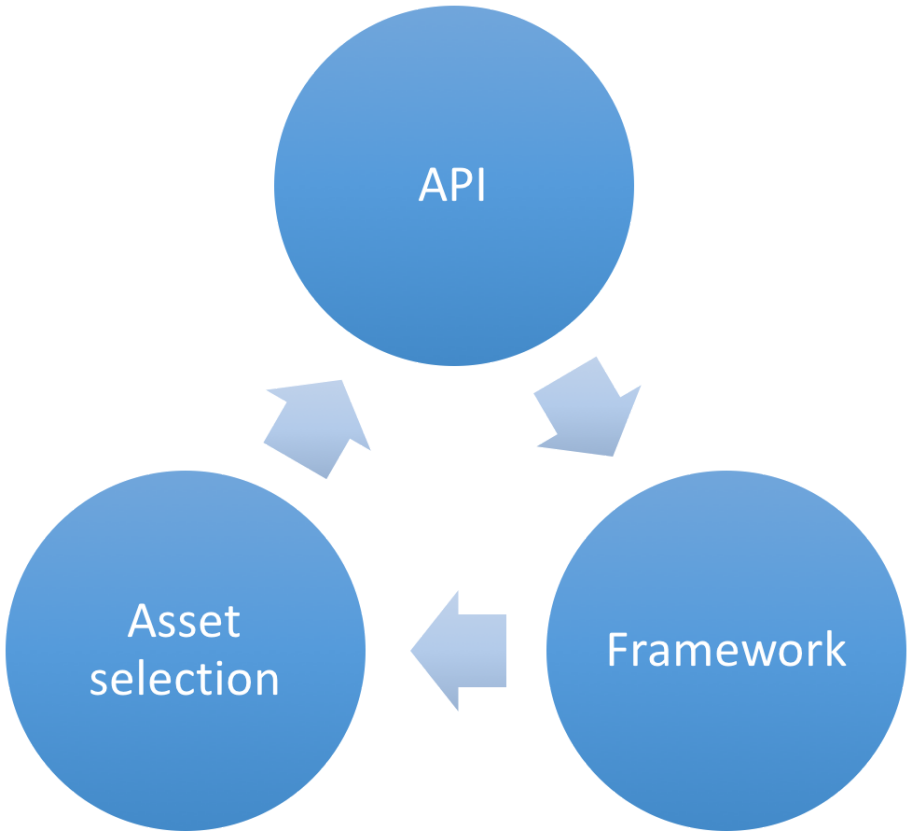
`.NETStandard,Version=1.3` 框架是一个基于包的框架。 它依赖基于框架的包，来定义和公开与框架有关的 API。

基于包的框架

框架和包之间是一种双向关系。 首先是为一个给定的框架定义了 API，例如 `netstandard1.3`。 以 `netstandard1.3` 为目标的包（或兼容的框架，如 `netstandard1.0`）定义了适用于 `netstandard1.3` 的 API。 听起来像是循环定义，然而并不是。 从"基于包的"这个词本身的角度来讲，框架的 API 定义是来自于包的。 框架本身并不定义任何 API。

其次，是这个双向关系中的资产选择。 包可以包含多个框架的资产。 对于一组包和/或元包的引用，框架需要决定它应选择哪些资产，例如，是 `net46` 还是 `netstandard1.3`。 选择正确的资产很重要。 例如，`net46` 资产可能并不与 .NET Framework 4.0 或 .NET Core 1.0 兼容。

可以在下图中看到这种关系。 *API* 选择框架 作为目标并定义了框架。 而*框架*用于*资产选择*。 *资产*实现了 API。



在 .Net Core 基础之上，基于包的框架主要有两个：

- `netstandard`
- `netcoreapp`

.NET Standard

.NET Standard（目标框架名字对象：`netstandard`）框架表示在 .NET Standard 基础之上生成并由其定义的 API。 如果构建的库将用于在多个运行时上运行，就应将此框架作为目标。 这样便可在任何一种兼容 .NET Standard 的运行时上受支持，例如 .NET

Core、.NET Framework 和 Mono/Xamarin。 每个运行时都支持一组 .NET Standard 版本，具体取决于实现的 API。

`netstandard` 框架隐式引用 NETStandard.Library 元包。 例如，以下 MSBuild 项目文件指示项目以 `netstandard1.6` 为目标，其引用 NETStandard.Library 1.6 版元包。

XML 复制

```xml
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard1.6</TargetFramework>
  </PropertyGroup>
</Project>
```

但项目文件中的框架和元包引用不需要匹配，并且可使用项目文件中的 `<NetStandardImplicitPackageVersion>` 元素指定低于元包版本的框架版本。 例如，以下项目文件有效。

XML 复制

```xml
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard1.3</TargetFramework>

<NetStandardImplicitPackageVersion>1.6.0</NetStandardImplicitPackageVersion>
  </PropertyGroup>
</Project>
```

面向 `netstandard1.3` 却使用 `NETStandard.Library` 1.6.0 版本，这一点很奇怪。 然而，这是一个有效的用例，因为元包支持更旧的 `netstandard` 版本。 可能恰好你已将 1.6.0 版的元包进行了标准化，然后将其用于所有库，而这些库可以面向各种 `netstandard` 版本。 使用此方法，只需还原 `NETStandard.Library` 1.6.0，无需加载早期版本。

反之，将 `netstandard1.6` 设为目标，却使用 1.3.0 版的 `NETStandard.Library` ，这是无效的。 不能将高版本的框架设为目标，却使用低版本的元包，因为低版本的元包不会对高版本的框架公开任何资产。 元包的版本控制方案断言元包匹配它们所描述的框架的最高版本。 凭借版本控制方案， `NETStandard.Library` 的第一个版本是 v1.6.0，因为它包含 `netstandard1.6` 资产。 而上例中使用 v1.3.0 版本只是为了保持对称，实际上它并不存在。

.NET Core 应用程序

.NET Core 应用程序（目标框架名字对象： `netcoreapp` ）框架表示 .NET Core 发行版及其提供的控制台应用程序模型附带的包和相关 API。 .NET Core 必须使用此框架，因为必须

要使用其中的控制台应用程序模型。同时只运行于 .Net Core 平台的库也应使用此模型。使用此框架后，所有应用和库将只能够在 .Net Core 上运行。

`Microsoft.NETCore.App` 元包的目标框架是 `netcoreapp`。 它提供了约 60 个库的访问权限，其中约 40 个由 `NETStandard.Library` 包提供，还有另外 20 个库。 可以引用目标框架为 `netcoreapp` 或与框架（如 `netstandard`）兼容的库获得对其他 API 的访问权限。

由 `Microsoft.NETCore.App` 提供的大部分其他库还可以使用 `netstandard` 作为目标，如果其他 `netstandard` 库满足这些框架的依赖项的话。 这意味着，`netstandard` 库也可以引用这些包作为依赖项。
