

Contents

[.NET Core 文档](#)

[.NET Core 简介](#)

[.NET Core 概述](#)

[安装](#)

[概述](#)

[安装 SDK](#)

[安装运行时](#)

[支持的操作系统和依赖项](#)

[macOS 公证问题](#)

[如何检查 .NET Core 版本](#)

[Linux 包管理器](#)

[Ubuntu 19.10](#)

[Ubuntu 19.04](#)

[Ubuntu 18.04](#)

[Ubuntu 16.04](#)

[CentOS 7](#)

[Debian 10](#)

[Debian 9](#)

[Fedora 31](#)

[Fedora 30](#)

[Fedora 29](#)

[OpenSUSE 15](#)

[RHEL 8](#)

[RHEL 7](#)

[SLES 15](#)

[SLES 12](#)

[安装本地化的 IntelliSense](#)

[入门](#)

[.NET Core 的新增功能](#)

[.NET Core 3.1 的新增功能](#)

[.NET Core 3.0 的新增功能](#)

[.NET Core 2.2 的新增功能](#)

[.NET Core 2.1 的新增功能](#)

[.NET Core 2.0 的新增功能](#)

重大更改

教程

[概述](#)

[通过 Visual Studio 开始使用 .NET Core](#)

[通过 Visual Studio Code 开始使用 .NET Core](#)

[通过 CLI 开始使用 .NET Core](#)

[通过 Visual Studio Code 在 macOS 上开始使用 .NET Core](#)

[通过 Visual Studio for Mac 开始使用 .NET Core](#)

[使用 Visual Studio 调试应用程序](#)

[使用 Visual Studio 发布应用程序](#)

[在 Visual Studio 中创建 .NET Standard 库](#)

[在 Visual Studio 中测试 .NET Standard 库](#)

[在 Visual Studio 中使用 .NET Standard 库](#)

[在 macOS 上构建完整的 .NET Core 解决方案](#)

[使用 .NET Core CLI 组织和测试项目](#)

[使用跨平台工具开发库](#)

[使用插件创建 .NET Core 应用](#)

[开发 ASP.NET Core 应用](#)

[从本机代码承载 .NET Core](#)

[创建 CLI 的模板](#)

[1 - 创建项模板](#)

[2 - 创建项目模板](#)

[3 - 创建模板包](#)

[为 CLI 创建工具](#)

[1 - 创建工具](#)

[2 - 使用全局工具](#)

[3 - 使用本地工具](#)

项目 SDK

[概述](#)

[参考](#)

[Microsoft.NET.Sdk](#)

[Microsoft.NET.Sdk.Web](#)

[Microsoft.NET.Sdk.Razor](#)

.NET Core SDK 概述

.NET Core CLI

[概述](#)

[参考](#)

[dotnet](#)

[dotnet build](#)

[dotnet build-server](#)

[dotnet clean](#)

[dotnet help](#)

[dotnet migrate](#)

[dotnet msbuild](#)

[dotnet new](#)

[dotnet nuget](#)

[dotnet nuget delete](#)

[dotnet nuget locals](#)

[dotnet nuget push](#)

[dotnet nuget add source](#)

[dotnet nuget disable source](#)

[dotnet nuget enable source](#)

[dotnet nuget list source](#)

[dotnet nuget remove source](#)

[dotnet nuget update source](#)

[dotnet pack](#)

[dotnet publish](#)

[dotnet restore](#)

[dotnet run](#)

[dotnet sln](#)
[dotnet store](#)
[dotnet test](#)
[dotnet tool](#)
[dotnet tool install](#)
[dotnet tool list](#)
[dotnet tool restore](#)
[dotnet tool run](#)
[dotnet tool uninstall](#)
[dotnet tool update](#)
[dotnet vstest](#)
[dotnet-install 脚本](#)
项目引用命令
[dotnet add reference](#)
[dotnet list reference](#)
[dotnet remove reference](#)
项目包命令
[dotnet add package](#)
[dotnet list package](#)
[dotnet remove package](#)
全局和本地工具
[管理工具](#)
[排查工具问题](#)
[global.json 概述](#)
[自定义模板](#)
[遥测](#)
[提升的访问权限](#)
[启用 Tab 自动补全](#)
诊断工具
[概述](#)
[托管调试器](#)
[日志记录和跟踪](#)

.NET Core CLI 全局工具

- dotnet-counters
- dotnet-dump
- dotnet-trace

.NET Core 诊断教程

- 调试内存泄露

其他 .NET Core 工具

概述

.NET Core 卸载工具

WCF Web 服务引用提供程序

- dotnet-svcutil

- dotnet-svcutil.xmlserializer

XML 序列化程序生成器

应用程序部署

概述

使用 CLI 发布应用

使用 Visual Studio 部署应用

使用 CLI 创建 NuGet 包

自包含部署运行时前滚

裁剪自包含部署和可执行文件

运行时包存储区

Docker

.NET 和 Docker 简介

使 .NET Core 应用程序容器化

Visual Studio 中的容器工具

运行时配置

设置

编译设置

调试和分析设置

垃圾回收器设置

全球化设置

网络设置

线程设置

迁移

[.NET Core 2.0 - 2.1](#)

[从 project.json 迁移](#)

[project.json 和 csproj 之间的映射](#)

[CLI 变更概述](#)

[csproj 格式的新增内容](#)

[从 DNX 迁移](#)

从 .NET Framework 移植

[概述](#)

[分析第三方依赖项](#)

[移植库](#)

[整理 .NET Core 的项目](#)

[不可用的技术](#)

[工具](#)

[使用 Windows 兼容包](#)

[移植 Windows 窗体项目](#)

[移植 WPF 项目](#)

[端口 C++/CLI 项目](#)

单元测试

[概述](#)

[单元测试最佳做法](#)

[使用 xUnit 进行 C# 单元测试](#)

[使用 NUnit 进行 C# 单元测试](#)

[使用 MSTest 进行 C# 单元测试](#)

[使用 xUnit 进行 F# 单元测试](#)

[使用 NUnit 进行 F# 单元测试](#)

[使用 MSTest 进行 F# 单元测试](#)

[使用 xUnit 进行 VB 单元测试](#)

[使用 NUnit 进行 VB 单元测试](#)

[使用 MSTest 进行 VB 单元测试](#)

[运行选择性单元测试](#)

[对已发布的输出进行单元测试](#)

[使用 Visual Studio 对 .NET Core 项目进行实时单元测试](#)

[版本管理](#)

[概述](#)

[.NET Core 版本选择](#)

[删除过时的运行时和 SDK](#)

[运行时标识符 \(RID\) 目录](#)

[依赖项管理和加载](#)

[依赖项管理](#)

[依赖项加载](#)

[了解 AssemblyLoadContext](#)

[依赖项加载详细信息](#)

[默认依赖项探测](#)

[加载托管程序集](#)

[加载附属程序集](#)

[加载非托管库](#)

[教程](#)

[使用插件创建 .NET Core 应用程序](#)

[如何在 .NET Core 中使用和调试程序集可卸载性](#)

[.NET Core 分发打包](#)

[向 COM 公开 .NET Core 组件](#)

[包、元包和框架](#)

[持续集成](#)

.NET Core 概述

2020/4/9 • [Edit Online](#)

.NET Core 具有以下特性：

- 跨平台：可在 Windows、macOS 和 Linux 操作系统上运行。
- 开放源代码：.NET Core 框架是开放源代码，使用 MIT 和 Apache 2 许可证。.NET Core 是一个 .NET Foundation 项目。
- 现代：它实现了异步编程、使用结构的无复制模式和容器的资源调控等现代范例。
- 性能：通过各种功能（如硬件内部函数、分层编译和跨度 $< T >$ ）来提供高性能。
- 跨环境一致：在多个操作系统和体系结构（包括 x64、x86 和 ARM）上以相同的行为运行代码。
- 命令行工具：包括可用于本地开发和持续集成的易于使用的命令行工具。
- 部署灵活：可以在应用中包含 .NET Core 或并行安装它（用户或系统范围安装）。可搭配 Docker 容器使用。

语言

可以使用 C#、Visual Basic 和 F# 语言编写适用于 .NET Core 的应用程序和库。这些语言可在你喜欢的文本编辑器或集成开发环境（IDE）中使用，包括：

- [Visual Studio](#)
- [Visual Studio Code](#)

编辑器集成部分由 OmniSharp 和 Ionide 项目的参与者提供。

API

.NET Core 公开用于生成任意类型的应用的框架：

- 使用 [ASP.NET Core](#) 的云应用
- 使用 [Xamarin](#) 的移动应用
- 使用 [System.Device.GPIO](#) 的 IoT 应用
- 使用 [WPF](#) 和 Windows 窗体的 Windows 客户端应用
- 机器学习 [ML.NET](#)

包括满足常见需求的多个 API：

- 基元类型，如 [System.Boolean](#) 和 [System.Int32](#)。
- 集合，例如 [System.Collections.Generic.List<T>](#) 和 [System.Collections.Generic.Dictionary< TKey, TValue >](#)。
- 实用程序类型，例如 [System.Net.Http.HttpClient](#) 和 [System.IO.FileStream](#)。
- 数据类型，例如 [System.Data.DataSet](#) 和 [System.Data.Entity.DbSet](#)。
- 高性能类型，例如 [System.Span<T>](#)、[System.Numerics.Vector](#) 和 [Pipelines](#)。

.NET Core 通过实现 [.NET Standard](#) 规范提供 .NET Framework 和 Mono API 的兼容性。

撰写

.NET Core 包括以下部分：

- [.NET Core 运行时](#)：提供类型系统、程序集加载、垃圾回收器、本机互操作和其他基本服务。[.NET Core 框架库](#)：提供基元数据类型、应用编写类型和基本实用程序。

- [ASP.NET Core 运行时](#): 提供一个框架来生成基于云且连接到 Internet 的新式应用程序, 例如 Web 应用、IoT 应用和移动后端。
- [.NET Core SDK 和语言编译器 \(Roslyn 和 F#\)](#): 提供 .NET Core 开发人员体验。
- [dotnet 命令](#): 用于启动 .NET Core 应用和 CLI 命令。它选择并托管运行时, 提供程序集加载策略并启动应用和工具。

开源

.NET Core 是一个通用的[开放源代码](#)开发平台。可以针对 x64、x86、ARM32 和 ARM64 处理器创建适用于 Windows、macOS 和 Linux 的 .NET Core 应用。为[云](#)、[IoT](#)、[客户端 UI](#) 和[机器学习](#)提供了框架和 API。

支持

[Microsoft 支持](#)在 Windows、macOS 和 Linux 上使用 .NET Core。它会定期更新以保证安全和质量(每月的第二个星期二)。

Microsoft 的 .NET Core 二进制发行版在 Azure 中的 Microsoft 维护服务器上进行生成和测试, 并遵循 Microsoft 的工程和安全实践。

[Red Hat 支持在 Red Hat Enterprise Linux \(RHEL\) 上使用 .NET Core](#)。Red Hat 从源中生成 .NET Core, 并在 [Red Hat 软件集合](#)中提供它。Red Hat 和 Microsoft 开展协作, 共同确保 .NET Core 能够在 RHEL 上正常运行。

[Tizen 支持在 Tizen 平台上使用 .NET Core](#)。

下载和安装 .NET Core

2020/3/18 • [Edit Online](#)

本文提供了有关如何下载和安装 .NET Core 的信息。.NET Core 有两部分：用于运行应用的“运行时”和用于创建应用的“SDK”。SDK 包括运行时。

- 如果用户需要 .NET Core 运行应用，请参阅[安装 .NET Core 运行时](#)。
- 如果开发人员需要 .NET Core 创建应用，请参阅[安装 .NET Core SDK](#)。

NOTE

.NET Core SDK 包括运行时。

依赖项

.NET Core 在 Windows、Linux 和 macOS 上受支持。有关要求的完整列表，请参阅[支持的操作系统](#)。

安装 .NET Core SDK

2020/3/18 • [Edit Online](#)

本文介绍如何安装 .NET Core SDK。.NET Core SDK 用于创建 .NET Core 应用和库。.NET Core 运行时始终随 SDK 一起安装。

使用安装程序安装

Windows 具有独立的安装程序，可用于安装 .NET Core 3.1 SDK：

- [x64\(64 位\)CPU](#)
- [x86\(32 位\)CPU](#)

使用安装程序安装

macOS 具有独立的安装程序，可用于安装 .NET Core 3.1 SDK：

- [x64\(64 位\)CPU](#)

下载并手动安装

除了使用适用于 .NET Core 的 macOS 安装程序，还可以下载并手动安装 SDK。

若要提取 SDK 并使 .NET Core CLI 命令可用于终端，请先[下载 .NET Core 二进制版本](#)。然后，打开终端并运行以下命令。假定将运行时下载到 `~/Downloads/dotnet-sdk.pkg` 文件中。

```
mkdir -p $HOME/dotnet
sudo installer -pkg ~/Downloads/dotnet-sdk.pkg -target $HOME/dotnet
export DOTNET_ROOT=$HOME/dotnet
export PATH=$PATH:$HOME/dotnet
```

使用包管理器安装

可使用许多常见的 Linux 包管理器安装 .NET Core SDK。有关详细信息，请参阅[Linux 包管理器 - 安装 .NET Core](#)。

仅在 x64 体系结构上支持使用包管理器安装。如果要使用其他体系结构（如 ARM）安装 .NET Core SDK，请遵循下面的[下载并手动安装](#)说明。有关支持的体系结构的详细信息，请参阅[.NET Core 依赖项和要求](#)。

下载并手动安装

若要提取 SDK 并使 .NET Core CLI 命令可用于终端，请先[下载 .NET Core 二进制版本](#)。然后，打开终端并运行以下命令。

```
mkdir -p $HOME/dotnet && tar zxf dotnet-sdk-3.1.100-linux-x64.tar.gz -C $HOME/dotnet
export DOTNET_ROOT=$HOME/dotnet
export PATH=$PATH:$HOME/dotnet
```

TIP

前面的 `export` 命令只会使 .NET Core CLI 命令对运行它的终端会话可用。

你可以编辑 shell 配置文件，永久地添加这些命令。Linux 提供了许多不同的 shell，每个都有不同的配置文件。例如：

- **Bash Shell**: `~/.bash_profile`、`~/.bashrc`
- **Korn Shell**: `~/.kshrc` 或 `.profile`
- **Z Shell**: `~/.zshrc` 或 `.zprofile`

为 shell 编辑相应的源文件，并将 `:$HOME/dotnet` 添加到现有 `PATH` 语句的末尾。如果不包含 `PATH` 语句，则使用 `export PATH=$PATH:$HOME/dotnet` 添加新行。

另外，将 `export DOTNET_ROOT=$HOME/dotnet` 添加至文件的末尾。

使用 Visual Studio 安装

如果你要使用 Visual Studio 开发 .NET Core 应用，请参阅下表，了解不同目标 .NET Core SDK 版本所需的 Visual Studio 最低版本。

.NET CORE SDK 版本	VISUAL STUDIO 版本
3.1	Visual Studio 2019 版本 16.4 或更高版本。
3.0	Visual Studio 2019 版本 16.3 或更高版本。
2.2	Visual Studio 2017 版本 15.9 或更高版本。
2.1	Visual Studio 2017 版本 15.7 或更高版本。

如果你已安装 Visual Studio，则可以使用以下步骤检查你的版本。

1. 打开 Visual Studio。
2. 选择“帮助” > “Microsoft Visual Studio”。
3. 从“关于”对话框中读取版本号。

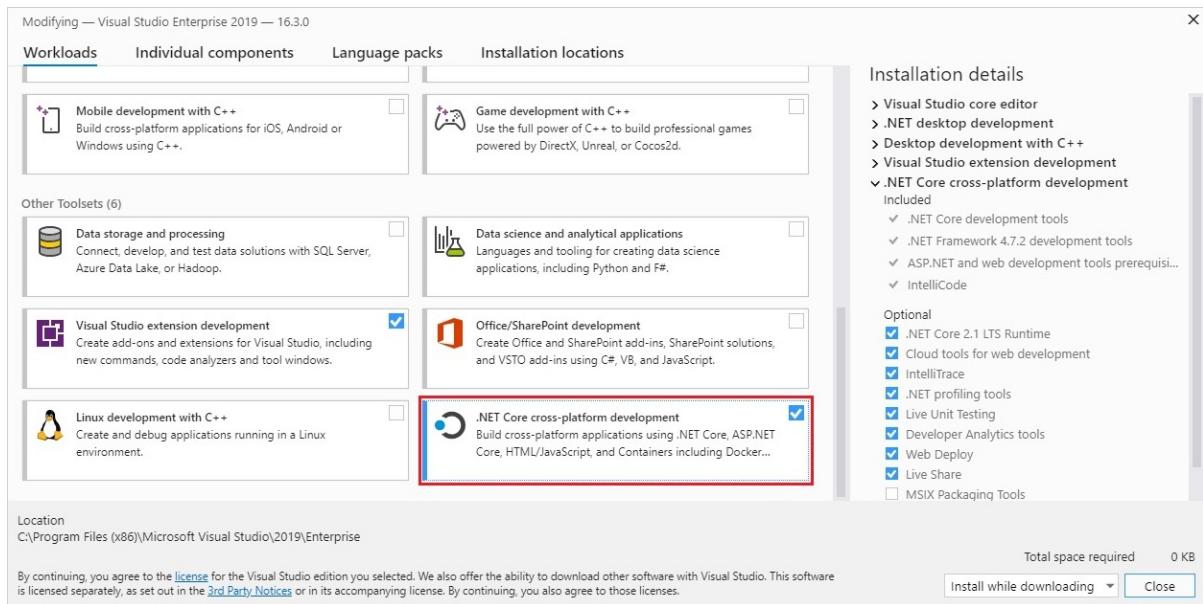
Visual Studio 可安装最新的 .NET Core SDK 和运行时。

- [下载 Visual Studio。](#)

选择工作负载

安装或修改 Visual Studio 时，根据要生成的应用程序的类型，选择以下一个或多个工作负载：

- “其他工具集”部分中的“.NET Core 跨平台开发”工作负载。
- “Web 和云”部分中的“ASP.NET 和 Web 开发”工作负载。
- “Web 和云”部分中的“Azure 开发”工作负载。
- “桌面和移动”部分中的“NET 桌面开发”工作负载。



下载并手动安装

若要提取运行时并使 .NET Core CLI 命令可用于终端, 请先[下载](#) .NET Core 二进制版本。然后, 创建要安装到的目录, 例如 `%USERPROFILE%\dotnet`。最后, 将下载的 zip 文件提取到该目录中。

默认情况下, .NET Core CLI 命令和应用不会使用通过这种方式安装的 .NET Core。必须明确选择使用它。为此, 请更改用于启动应用程序的环境变量:

```
set DOTNET_ROOT=%USERPROFILE%\dotnet
set PATH=%USERPROFILE%\dotnet;%PATH%
```

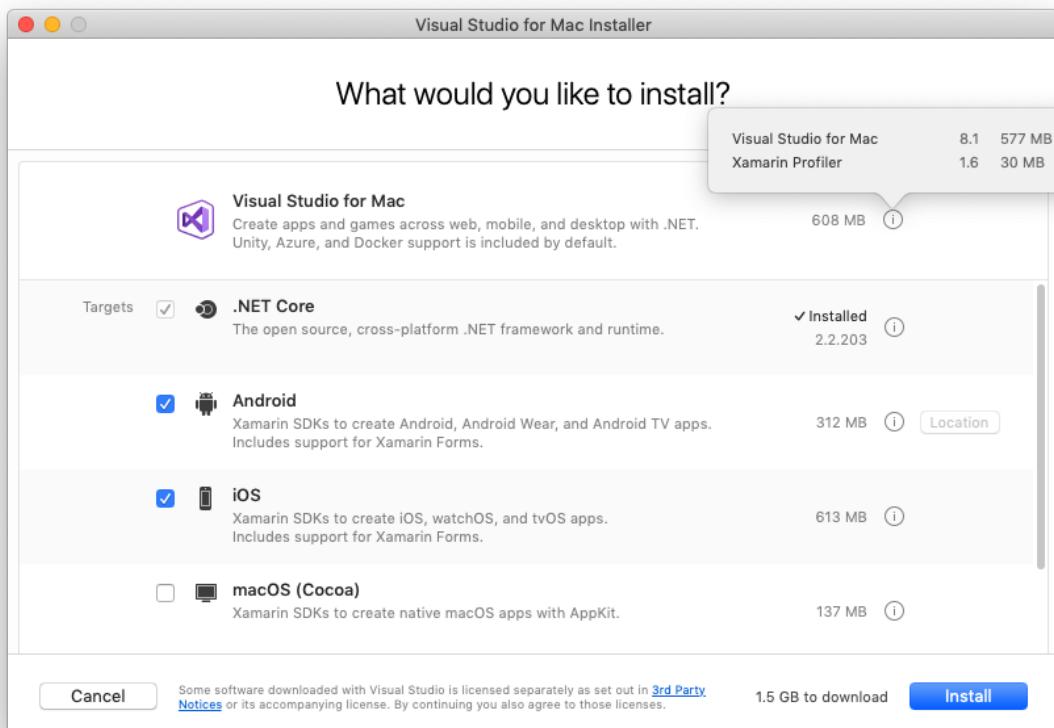
使用此方法可以将多个版本安装到不同的位置, 然后通过使用指向安装位置的环境变量运行应用程序来明确选择应用程序应使用哪个安装位置。

即使已设置这些环境变量, 在选择用于运行应用程序的最佳框架时, .NET Core 仍会考虑默认的全局安装位置。默认位置通常为安装程序使用的 `C:\Program Files\dotnet`。还可以通过设置此环境变量来指示运行时仅使用自定义安装位置:

```
set DOTNET_MULTILEVEL_LOOKUP=0
```

使用 Visual Studio for Mac 安装

在选定“.NET Core”工作负载时, 使用 Visual Studio for Mac 安装 .NET Core SDK。若要开始在 macOS 上进行 .NET Core 开发, 请参阅[安装 Visual Studio 2019 for Mac](#)。对于最新的版本 .NET Core 3.1, 则必须使用 Visual Studio for Mac 8.4 预览版。



随 Visual Studio Code 一起安装

Visual Studio Code 是一个功能强大的轻量级源代码编辑器，可在桌面上运行。Visual Studio Code 适用于 Windows、macOS 和 Linux。

虽然 Visual Studio Code 不像 Visual Studio 一样附带自动的 .NET Core 安装程序，但添加 .NET Core 支持非常简单。

1. [下载并安装 Visual Studio Code。](#)
2. [下载并安装 .NET Core SDK。](#)
3. [从 Visual Studio Code 市场安装 C# 扩展。](#)

使用 PowerShell 自动化安装

[dotnet-install 脚本](#)用于 SDK 的自动化和非管理员安装。可从 [dotnet-install 脚本引用页](#) 下载该脚本。

此脚本默认安装最新的[长期支持 \(LTS\)](#) 版本，即 .NET Core 3.1。若要安装最新版本的 .NET Core，请使用以下开关运行脚本。

```
dotnet-install.ps1 -Channel Current
```

使用 Bash 自动化安装

[dotnet-install 脚本](#)用于 SDK 的自动化和非管理员安装。可从 [dotnet-install 脚本引用页](#) 下载该脚本。

此脚本默认安装最新的[长期支持 \(LTS\)](#) 版本，即 .NET Core 3.1。若要安装最新版本的 .NET Core，请使用以下开关运行脚本。

```
./dotnet-install.sh -c Current
```

所有 .NET Core 下载项

可直接通过以下链接之一下载和安装 .NET Core:

- [.NET Core 3.1 下载](#)
- [.NET Core 3.0 下载](#)
- [.NET Core 2.2 下载](#)
- [.NET Core 2.1 下载](#)

Docker

容器提供了一种将应用程序与主机系统的其余部分隔离的轻量级方法。同一计算机上的容器只共享内核，并使用为应用程序提供的资源。

.NET Core 可在 Docker 容器中运行。官方 .NET Core Docker 映像发布到 Microsoft 容器注册表 (MCR)，用户可以在 [Microsoft.NET Core Docker 中心存储库](#) 中找到这些映像。每个存储库包含 .NET (SDK 或运行时) 和可以使用的操作系统的不同组合的映像。

Microsoft 提供适合特定场景的映像。例如，[ASP.NET Core 存储库](#) 提供针对在生产环境中运行 ASP.NET Core 应用生成的映像。

有关在 Docker 容器中使用 .NET Core 的详细信息，请参阅 [.NET 和 Docker 简介](#) 和 [示例](#)。

后续步骤

- [教程:Hello World 教程。](#)
- [教程:使用 Visual Studio Code 创建一个新应用。](#)
- [教程:使 .NET Core 应用容器化。](#)
- [处理 macOS Catalina 公证。](#)
- [教程:开始使用 macOS。](#)
- [教程:使用 Visual Studio Code 创建一个新应用。](#)
- [教程:使 .NET Core 应用容器化。](#)
- [教程:使用 Visual Studio Code 创建一个新应用。](#)
- [教程:使 .NET Core 应用容器化。](#)

安装 .NET Core 运行时

2020/3/18 • [Edit Online](#)

本文介绍如何下载和安装 .NET Core 运行时。.NET Core 运行时用于运行使用 .NET Core 创建的应用。

使用安装程序安装

Windows 具有独立的安装程序，可用于安装 .NET Core 3.1 运行时：

- x64 (64 位)CPU
- x86 (32 位)CPU

使用安装程序安装

macOS 具有独立的安装程序，可用于安装 .NET Core 3.1 运行时：

- x64 (64 位)CPU

下载并手动安装

除了使用适用于 .NET Core 的 macOS 安装程序，还可以下载并手动安装运行时。

若要安装运行时并使 .NET Core CLI 命令可用于终端，请先[下载](#) .NET Core 二进制版本。然后，打开终端并运行以下命令。假定将运行时下载到 `~/Downloads/dotnet-runtime.pkg` 文件中。

```
mkdir -p $HOME/dotnet
sudo installer -pkg ~/Downloads/dotnet-runtime.pkg -target $HOME/dotnet
export DOTNET_ROOT=$HOME/dotnet
export PATH=$PATH:$HOME/dotnet
```

使用包管理器安装

可使用许多常见的 Linux 包管理器安装 .NET Core 运行时。有关详细信息，请参阅 [Linux 包管理器 - 安装 .NET Core](#)。

仅在 x64 体系结构上支持使用包管理器安装。如果要使用其他体系结构（如 ARM）安装 .NET Core 运行时，请遵循[下载并手动安装](#)部分中的说明。有关支持的体系结构的详细信息，请参阅 [.NET Core 依赖项和要求](#)。

下载并手动安装

若要提取运行时并使 .NET Core CLI 命令可用于终端，请先[下载](#) .NET Core 二进制版本。然后，打开终端并运行以下命令。

```
mkdir -p $HOME/dotnet && tar zxf aspnetcore-runtime-3.1.0-linux-x64.tar.gz -C $HOME/dotnet
export DOTNET_ROOT=$HOME/dotnet
export PATH=$PATH:$HOME/dotnet
```

TIP

前面的 `export` 命令只会使 .NET Core CLI 命令对运行它的终端会话可用。

你可以编辑 shell 配置文件，永久地添加这些命令。Linux 提供了许多不同的 shell，每个都有不同的配置文件。例如：

- **Bash Shell**: `~/.bash_profile`、`~/.bashrc`
- **Korn Shell**: `~/.kshrc` 或 `.profile`
- **Z Shell**: `~/.zshrc` 或 `.zprofile`

为 shell 编辑相应的源文件，并将 `:$HOME/dotnet` 添加到现有 `PATH` 语句的末尾。如果不包含 `PATH` 语句，则使用 `export PATH=$PATH:$HOME/dotnet` 添加新行。

另外，将 `export DOTNET_ROOT=$HOME/dotnet` 添加至文件的末尾。

使用此方法可以将不同的版本安装到不同的位置，并明确选择应用程序要使用的对应版本。

使用 PowerShell 自动化安装

[dotnet-install 脚本](#) 用于运行时的自动化和非管理员安装。可从 [dotnet-install 脚本引用页](#) 下载该脚本。

此脚本默认安装最新的[长期支持 \(LTS\)](#) 版本，即 .NET Core 3.1。可通过指定 `Channel` 开关以选择特定版本。包括 `Runtime` 开关以安装运行时。否则，该脚本安装 [SDK](#)。

```
dotnet-install.ps1 -Channel 3.1 -Runtime aspnetcore
```

NOTE

以上命令安装 ASP.NET Core 运行时，用于实现最大的兼容性。ASP.NET Core 运行时还包括标准 .NET Core 运行时。

下载并手动安装

若要提取运行时并使 .NET Core CLI 命令可用于终端，请先[下载 .NET Core 二进制版本](#)。然后，创建要安装到的目录，例如 `%USERPROFILE%\dotnet`。最后，将下载的 zip 文件提取到该目录中。

默认情况下，.NET Core CLI 命令和应用不会使用通过这种方式安装的 .NET Core。必须明确选择使用它。为此，请更改用于启动应用程序的环境变量：

```
set DOTNET_ROOT=%USERPROFILE%\dotnet
set PATH=%USERPROFILE%\dotnet;%PATH%
```

使用此方法可以将多个版本安装到不同的位置，然后通过使用指向安装位置的环境变量运行应用程序来明确选择应用程序应使用哪个安装位置。

即使已设置这些环境变量，在选择用于运行应用程序的最佳框架时，.NET Core 仍会考虑默认的全局安装位置。默认位置通常为安装程序使用的 `C:\Program Files\dotnet`。还可以通过设置此环境变量来指示运行时仅使用自定义安装位置：

```
set DOTNET_MULTILEVEL_LOOKUP=0
```

使用 Bash 自动化安装

[dotnet-install 脚本](#) 用于运行时的自动化和非管理员安装。可从 [dotnet-install 脚本引用页](#) 下载该脚本。

此脚本默认安装最新的[长期支持 \(LTS\)](#) 版本，即 .NET Core 3.1。可通过指定 `current` 开关以选择特定版本。包括 `runtime` 开关以安装运行时。否则，该脚本安装 [SDK](#)。

```
./dotnet-install.sh --channel 3.1 --runtime aspnetcore
```

NOTE

以上命令安装 ASP.NET Core 运行时，用于实现最大的兼容性。ASP.NET Core 运行时还包括标准 .NET Core 运行时。

所有 .NET Core 下载项

可直接通过以下链接之一下载和安装 .NET Core：

- [.NET Core 3.1 下载](#)
- [.NET Core 2.1 下载](#)

Docker

容器提供了一种将应用程序与主机系统的其余部分隔离的轻量级方法。同一计算机上的容器只共享内核，并使用为应用程序提供的资源。

.NET Core 可在 Docker 容器中运行。官方 .NET Core Docker 映像发布到 Microsoft 容器注册表 (MCR)，用户可以在 [Microsoft.NET Core Docker 中心存储库](#) 中找到这些映像。每个存储库包含 .NET (SDK 或运行时) 和可以使用的操作系统的不同组合的映像。

Microsoft 提供适合特定场景的映像。例如，[ASP.NET Core 存储库](#) 提供针对在生产环境中运行 ASP.NET Core 应用生成的映像。

有关在 Docker 容器中使用 .NET Core 的详细信息，请参阅 [.NET 和 Docker 简介](#) 和 [示例](#)。

后续步骤

- [如何检查是否已安装 .NET Core。](#)

.NET Core 依赖项和要求

2020/4/2 • [Edit Online](#)

本文详细介绍了 .NET Core 支持的操作系统和 CPU 体系结构。

支持的操作系统

- [.NET Core 3.1](#)
- [.NET Core 3.0](#)
- [.NET Core 2.2](#)
- [.NET Core 2.1](#)

.NET Core 3.1 支持下列 Windows 版本：

NOTE

+ 表示最低版本。

(OS)	VERSION	CPU
Windows 客户端	7 SP1+、8.1	x64、x86
Windows 10 客户端	版本 1607+	x64、x86
Windows Server	2012 R2 +	x64、x86
Nano Server	版本 1803+	x64、ARM32

有关 .NET Core 3.1 支持的操作系统、发行版和生命周期策略的详细信息，请参阅 [.NET Core 3.1 支持的 OS 版本](#)。

Windows 7/Vista/8.1/Server 2008 R2

如果要在以下 Windows 版本上安装 .NET SDK 或运行时，则需要其他依赖项：

- Windows 7 SP1
- Windows Vista SP 2
- Windows 8.1
- Windows Server 2008 R2
- Windows Server 2012 R2

安装以下组件：

- [Microsoft Visual C++ 2015 Redistributable 更新 3。](#)
- [KB2533623](#)

如果遇到一个以下错误，也需要满足上述要求：

此程序无法启动，因为计算机上缺少 api-ms-win-crt-runtime-l1-1-0.dll。尝试重新安装该程序以解决此问题。

- 或 -

已找到库 hostfxr.dll，但未能将其从 C:\<path_to_app>\hostfxr.dll 中加载。

- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1

.NET Core 3.1 将 Linux 视为一个操作系统。对于支持的 Linux 发行版，每芯片体系结构都对应有一个 Linux 内部版本。

以下 Linux 发行版/版本支持 .NET Core 3.1：

NOTE

+ 表示最低版本。

(OS)	VERSION	架构
Red Hat Enterprise Linux	6、7、8	X64
CentOS	7+	X64
Oracle Linux	7+	X64
Fedora	30+	X64
Debian	9+	x64、ARM32、ARM64
Ubuntu	16.04+	x64、ARM32、ARM64
Linux Mint	18+	X64
openSUSE	15+	X64
SUSE Enterprise Linux (SLES)	12 SP2+	X64
Alpine Linux	3.10+	x64、ARM64

有关 .NET Core 3.1 支持的操作系统、发行版和生命周期策略的详细信息，请参阅 [.NET Core 3.1 支持的 OS 版本](#)。

有关如何在 ARM64(内核 4.14+)上安装 .NET Core 3.1 的详细信息，请参阅 [在 Linux ARM64 上安装 .NET Core 3.0](#)。

IMPORTANT

ARM64 支持需要 Linux 内核 4.14 或更高版本。某些 linux 发行版满足此要求，而另一些则不满足。例如，支持 Ubuntu 18.04，但不支持 Ubuntu 16.04。

Linux 发行版本依赖项

根据 Linux 发行版, 你可能需要安装其他依赖项。

IMPORTANT

确切的版本和名称可能因所选 Linux 发行版本略有不同。

Ubuntu

Ubuntu 发行版需要安装以下库:

- liblttng-ust0
- libcurl3(针对 14.x 和 16.x)
- libcurl4(针对 18.x)
- libssl1.0.0
- libkrb5-3
- zlib1g
- libicu52(针对 14.x)
- libicu55(针对 16.x)
- libicu57(针对 17.x)
- libicu60(针对 18.x)

对于使用 System.Drawing.Common 程序集的 .NET Core 应用, 还需要以下依赖项:

- libgdiplus(版本 6.0.1 或更高版本)

WARNING

大多数 Ubuntu 版本都包含 libgdiplus 的早期版本。可以通过将 Mono 存储库添加到系统来安装最新版 libgdiplus。有关详细信息, 请参阅 <https://www.mono-project.com/download/stable/>。

CentOS 和 Fedora

CentOS 发行版本需要安装以下库:

- lttng-ust
- libcurl
- openssl-libs
- krb5-libs
- libicu
- zlib

Fedora 用户: 如果 OpenSSL 的版本为 1.1 及更高版本, 则需要安装 compat-openssl10。

对于 .NET Core 2.0, 还需要以下依赖项:

- libunwind
- libuuid

有关依赖项的详细信息, 请参阅[独立式 Linux 应用](#)。

对于使用 System.Drawing.Common 程序集的 .NET Core 应用, 还需要以下依赖项:

- libgdiplus(版本 6.0.1 或更高版本)

WARNING

CentOS 和 Fedora 的大多数版本都包含 libgdiplus 的早期版本。可以通过将 Mono 存储库添加到系统来安装最新版 libgdiplus。有关详细信息, 请参阅 [https://www.mono-project.com/download/stable/。](https://www.mono-project.com/download/stable/)

以下 macOS 版本支持 .NET Core:

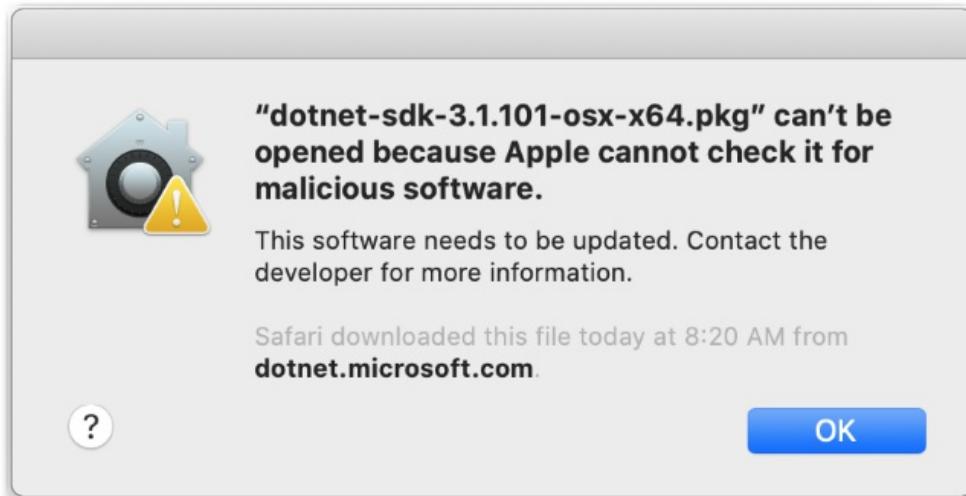
NOTE

 表示最低版本。

.NET CORE 版本	MACOS	架构	
3.1	High Sierra (10.13+)	X64	详细信息
3.0	High Sierra (10.13+)	X64	详细信息
2.2	Sierra (10.12+)	X64	详细信息
2.1	Sierra (10.12+)	X64	详细信息

自 macOS Catalina(版本10.15)开始, 所有在 2019 年 6 月 1 日之后生成并使用开发者 ID 扩散的软件都必须经过公证。此要求应用于 .NET Core 运行时、.NET Core SDK 以及使用 .NET Core 创建的软件。

自 2020 年 2 月 18 日起, .NET Core(运行时和 SDK)版本 3.1、3.0 和 2.1 的安装程序都已经过公证。以前发布的版本没有经过公证。如果运行未经过公证的应用, 将看到类似于下图的错误:



若要详细了解强制执行的公证要求对 .NET Core 和 .NET Core 应用的影响, 请参阅[处理 macOS Catalina 公证](#)。

libgdiplus

使用 System.Drawing.Common 程序集的 .NET Core 应用程序要求安装 libgdiplus。

获取 libgdiplus 的一个简单方法是使用适用于 macOS 的 Homebrew ("brew") 包。在安装 brew 后, 通过在终端(命令)提示符处执行以下命令来安装 libgdiplus :

```
brew update  
brew install mono-libgdiplus
```

后续步骤

- 若要开发并运行应用, 请安装 [.NET Core SDK](#)(包括运行时)。
- 若要运行其他人创建的应用, 请安装 [.NET Core 运行时](#)。

macOS Catalina 公证以及对 .NET Core 下载和项目的影响

2020/3/18 • [Edit Online](#)

自 macOS Catalina(版本 10.15)开始, 所有在 2019 年 6 月 1 日之后生成并使用开发者 ID 扩散的软件都必须经过公证。此要求应用于 .NET Core 运行时、.NET Core SDK 以及使用 .NET Core 创建的软件。本文介绍了 .NET Core 和 macOS 公证可能会遇到的常见情况。

安装 .NET Core

自 2020 年 2 月 18 日起, .NET Core(运行时和 SDK)版本 3.1、3.0 和 2.1 的安装程序都已经过公证。以前发布的版本没有经过公证。通过先下载安装程序, 然后使用 `sudo installer` 命令, 可以手动安装 .NET Core 的未公证版本。有关详细信息, 请参阅[下载并手动安装 macOS](#)。

自以下版本开始, .NET Core 安装程序未经过公证:

- .NET Core 运行时
 - 2.1.16
 - 3.0.3
 - 3.1.2
- .NET Core SDK
 - 2.1.512
 - 3.0.103
 - 3.1.102

默认禁用 appHost

默认情况下, 当项目编译、发布或运行时, .NET Core SDK 3.0 及更高版本的未公证版本会生成一个本机 Mach-O 可执行文件(即 appHost)。此可执行文件是运行应用的一种简便方法。否则必须通过运行 `dotnet <filename.dll>` 启动应用。启用 appHost 后, 会在 appHost 的上下文中调用 `dotnet run` 命令。有关详细信息, 请参阅[appHost 的上下文](#)。

从 .NET Core SDK 3.0 及更高版本的已公证版本开始, 默认情况下不生成 appHost 可执行文件。可以通过项目文件中的 `UseAppHost` 布尔值设置来启用 appHost 生成。还可以在运行的特定 `dotnet` 命令的命令行上通过 `-p:UseAppHost` 参数切换 appHost 的启用状态:

- 项目文件

```
<PropertyGroup>
  <UseAppHost>true</UseAppHost>
</PropertyGroup>
```

- 命令行参数

```
dotnet run -p:UseAppHost=true
```

发布**独立**应用时, 始终会创建 appHost。

有关 `UseAppHost` 设置的详细信息, 请参阅 [Microsoft.NET.Sdk 的 MSBuild 属性](#)。

AppHost 的上下文

在项目中启用了 `appHost` 并使用 `dotnet run` 命令运行应用时, 将在 `appHost` 的上下文中调用应用, 而不是在默认主机(默认主机对应的是 `dotnet` 命令)的上下文中调用。如果在项目中禁用了 `appHost`, 则 `dotnet run` 命令将在默认主机的上下文中运行应用。即使禁用了 `appHost`, 如果发布独立应用, 也会生成一份 `appHost` 可执行文件, 且用户也可以使用该可执行文件运行应用。使用 `dotnet <filename.dll>` 运行应用会通过默认主机(共享运行时)调用应用。

调用使用 `appHost` 的应用时, 应用访问的证书分区与已公证的默认主机不同。如果应用必须访问通过默认主机安装的证书, 请使用 `dotnet run` 命令从应用的项目文件运行它, 或使用 `dotnet <filename.dll>` 命令直接启动该应用。

有关此方案的详细信息, 请参阅 [ASP.NET Core 和 macOS 和证书](#)部分。

ASP.NET Core 和 macOS 和证书

使用 .NET Core, 可以使用 [System.Security.Cryptography.X509Certificates](#) 类在 macOS Keychain 中管理证书。在决定要考虑哪个分区时, 对 macOS Keychain 的访问将使用应用程序标识作为主键。例如, 未签名的应用程序会将机密存储在未签名分区中, 而已签名应用程序会将其机密存储在仅能由它们访问的分区中。要使用的分区取决于调用应用的执行源。

.NET Core 提供三个执行源:`appHost`、默认主机(`dotnet` 命令)和自定义主机。每个执行模型都可能具有不同的标识(已签名或未签名), 并可以访问 Keychain 中的不同分区。通过一种模式导入的证书可能不能通过另一种模式访问。例如, 已公证版本的 .NET Core 具有已签名的默认主机。根据证书的标识将证书导入到安全分区中。由于 `appHost` 是未签名的, 因此无法从生成的 `appHost` 访问这些证书。

再举一例, 默认情况下, ASP.NET Core 通过默认主机导入默认 SSL 证书。使用 `appHost` 的 ASP.NET Core 应用程序将不能访问此证书, 并且当 .NET Core 检测到无法访问该证书时, 将收到错误消息。该错误消息中会提供关于如何解决此问题的说明。

如果证书共享是必需的, macOS 会提供带有 `security` 实用工具的配置选项。

如需详细了解如何解决 ASP.NET Core 证书问题, 请参阅[在 ASP.NET Core 中强制执行 HTTPS](#)。

默认权利

.NET Core 的默认主机(`dotnet` 命令)具有一套默认权利。需要这些权利才能正常运行 .NET Core。你的应用程序可能需要其他权利, 在这种情况下, 需要生成并使用 `appHost`, 然后在本地添加必要的权利。

.NET Core 的默认权利包括:

- `com.apple.security.cs.allow-jit`
- `com.apple.security.cs.allow-unsigned-executable-memory`
- `com.apple.security.cs.allow-dyld-environment-variables`
- `com.apple.security.cs.disable-library-validation`

对 .NET Core 应用进行公证

如果希望应用程序在 macOS Catalina(版本 10.15)或更高版本上运行, 则需要对应用进行公证。为了进行公证而随应用程序提交的 `appHost` 应至少具备与 .NET Core 相同的[默认权利](#)。

后续步骤

- [.NET Core 依赖项和要求](#)。

- 安装 .NET Core SDK。
- 安装 .NET Core 运行时

如何检查是否已安装 .NET Core

2020/3/18 • [Edit Online](#)

本文介绍如何检查计算机上安装的 .NET Core 运行时和 SDK 的版本。如果你拥有一个集成开发环境(如 Visual Studio 或 Visual Studio for Mac)，则可能已安装 .NET core。

安装 SDK 便会安装相应的运行时。

如果本文中的任何命令失败，则未安装运行时或 SDK。有关详细信息，请参阅[下载并安装 .NET Core](#)。

检查 SDK 版本

可使用终端查看当前安装的 .NET Core SDK 版本。打开终端并运行以下命令。

```
dotnet --list-sdks
```

将获得类似于下面的输出。

```
2.1.500 [C:\program files\dotnet\sdk]
2.1.502 [C:\program files\dotnet\sdk]
2.1.504 [C:\program files\dotnet\sdk]
2.1.600 [C:\program files\dotnet\sdk]
2.1.602 [C:\program files\dotnet\sdk]
2.2.101 [C:\program files\dotnet\sdk]
3.0.100 [C:\program files\dotnet\sdk]
3.1.100 [C:\program files\dotnet\sdk]
```

```
2.1.500 [/home/user/dotnet/sdk]
2.1.502 [/home/user/dotnet/sdk]
2.1.504 [/home/user/dotnet/sdk]
2.1.600 [/home/user/dotnet/sdk]
2.1.602 [/home/user/dotnet/sdk]
2.2.101 [/home/user/dotnet/sdk]
3.0.100 [/home/user/dotnet/sdk]
3.1.100 [/home/user/dotnet/sdk]
```

```
2.1.500 [/usr/local/share/dotnet/sdk]
2.1.502 [/usr/local/share/dotnet/sdk]
2.1.504 [/usr/local/share/dotnet/sdk]
2.1.600 [/usr/local/share/dotnet/sdk]
2.1.602 [/usr/local/share/dotnet/sdk]
2.2.101 [/usr/local/share/dotnet/sdk]
3.0.100 [/usr/local/share/dotnet/sdk]
3.1.100 [/usr/local/share/dotnet/sdk]
```

检查运行时版本

可使用以下命令查看当前安装的 .NET Core 运行时版本。

```
dotnet --list-runtimes
```


详细信息

可通过命令 `dotnet --info` 查看 SDK 版本和运行时版本。你还将获得其他环境相关信息，如操作系统版本和运行时标识符 (RID)。

后续步骤

- [安装 .NET Core 运行时。](#)
- [安装 .NET Core SDK。](#)

Ubuntu 19.10 包管理器 - 安装 .NET Core

2020/3/30 • [Edit Online](#)

仅在 x64 体系结构上支持包管理器安装。其他体系结构(如 ARM)必须[手动安装 .NET Core SDK](#)或[手动安装 .NET Core 运行时](#)。有关详细信息, 请参阅[.NET Core 依赖项和要求](#)。

本文介绍如何使用包管理器在 Ubuntu 19.10 上安装 .NET Core。

如果要开发 .NET Core 应用, 请安装 SDK(包括运行时)。或者, 如果只需运行应用程序, 请安装运行时。如果要安装该运行时, 建议安装[ASP.NET Core 运行时](#), 因为它同时包括 .NET Core 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时, 请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息, 请参阅[如何检查是否已安装 .NET Core](#)。

注册 Microsoft 密钥和源

安装 .NET 之前, 需要:

- 注册 Microsoft 密钥。
- 注册产品存储库。
- 安装必需的依赖项。

每台计算机只需要执行一次此操作。

打开终端并运行以下命令。

```
wget https://packages.microsoft.com/config/ubuntu/19.10/packages-microsoft-prod.deb -O packages-microsoft-prod.deb  
sudo dpkg -i packages-microsoft-prod.deb
```

安装 .NET Core SDK

更新可供安装的产品, 然后安装 .NET Core SDK。在终端中, 运行以下命令。

```
sudo apt-get update  
sudo apt-get install apt-transport-https  
sudo apt-get update  
sudo apt-get install dotnet-sdk-3.1
```

IMPORTANT

如果收到类似于“找不到包 `dotnet-sdk-3.1`”的错误消息, 请参阅[包管理器疑难解答](#)部分。

安装 ASP.NET Core 运行时

更新可供安装的产品, 然后安装 ASP.NET Core 运行时。在终端中, 运行以下命令。

```
sudo apt-get update  
sudo apt-get install apt-transport-https  
sudo apt-get update  
sudo apt-get install aspnetcore-runtime-3.1
```

IMPORTANT

如果收到类似于“找不到包 aspnetcore-runtime-3.1”的错误消息，请参阅[包管理器疑难解答](#)部分。

安装 .NET Core 运行时

更新可供安装的产品，然后安装 .NET Core 运行时。在终端中，运行以下命令。

```
sudo apt-get update  
sudo apt-get install apt-transport-https  
sudo apt-get update  
sudo apt-get install dotnet-runtime-3.1
```

IMPORTANT

如果收到类似于“找不到包 dotnet-runtime-3.1”的错误消息，请参阅[包管理器疑难解答](#)部分。

如何安装其他版本

添加到包管理器源的包以可改动的格式命名：`{product}-{type}-{version}`。

- **product**

要安装的 .NET 产品的类型。有效选项是：

- dotnet
- aspnetcore

- **type**

选择 SDK 或运行时。有效选项是：

- SDK
- Runtime — 运行时

- **version**

要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本，例如：

- 3.1
- 3.0
- 2.1

尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版的列表，请参阅[.NET Core 依赖项和要求](#)。

示例

- 安装 ASP.NET Core 3.1 运行时：`aspnetcore-runtime-3.1`
- 安装 .NET Core 2.1 运行时：`dotnet-runtime-2.1`
- 安装 .NET Core 3.1 SDK：`dotnet-sdk-3.1`

缺少包

如果包版本组合无效，则它不可用。例如，未安装 ASP.NET Core SDK，所有 SDK 组件都包含在 .NET Core SDK 中。

`aspnetcore-sdk-2.2` 的值不正确，应为 `dotnet-sdk-2.2`。有关 .NET Core 支持的 Linux 发行版的列表，请参阅

[.NET Core 依赖项和要求](#)。

包管理器疑难解答

本部分提供有关使用程序包管理器安装 .NET Core 时可能会遇到的常见错误的信息。

无法定位

如果收到类似于“找不到包 {.NET Core 包}”的错误消息，请运行以下命令。

```
sudo dpkg --purge packages-microsoft-prod && sudo dpkg -i packages-microsoft-prod.deb  
sudo apt-get update  
sudo apt-get install {the .NET Core package}
```

如果这不起作用，可使用以下命令运行手动安装。

```
sudo apt-get install -y gpg  
wget 0- https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor -o microsoft.asc.gpg  
sudo mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/  
wget https://packages.microsoft.com/config/ubuntu/19.10/prod.list  
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list  
sudo chown root:root /etc/apt/trusted.gpg.d/microsoft.asc.gpg  
sudo chown root:root /etc/apt/sources.list.d/microsoft-prod.list  
sudo apt-get install -y apt-transport-https  
sudo apt-get update  
sudo apt-get install {the .NET Core package}
```

未能提取

安装 .NET Core 包时，可能会看到类似于

`Failed to fetch ... File has unexpected size ... Mirror sync in progress?` 的错误。一般而言，此错误表示 .NET Core 的包源正在通过更新的包版本进行更新，应稍后重试。升级期间，包源的不可用时间不应超过 30 分钟。如果持续收到此错误超过 30 分钟，请在 <https://github.com/dotnet/core/issues> 中提交问题。

Ubuntu 19.04 包管理器 - 安装 .NET Core

2020/3/30 • [Edit Online](#)

仅在 x64 体系结构上支持包管理器安装。其他体系结构(如 ARM)必须[手动安装 .NET Core SDK](#) 或[手动安装 .NET Core 运行时](#)。有关详细信息, 请参阅[.NET Core 依赖项和要求](#)。

本文介绍如何使用包管理器在 Ubuntu 19.04 上安装 .NET Core。

如果要开发 .NET Core 应用, 请安装 SDK(包括运行时)。或者, 如果只需运行应用程序, 请安装运行时。如果要安装该运行时, 建议安装[ASP.NET Core 运行时](#), 因为它同时包括 .NET Core 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时, 请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息, 请参阅[如何检查是否已安装 .NET Core](#)。

注册 Microsoft 密钥和源

安装 .NET 之前, 需要:

- 注册 Microsoft 密钥。
- 注册产品存储库。
- 安装必需的依赖项。

每台计算机只需要执行一次此操作。

打开终端并运行以下命令。

```
wget https://packages.microsoft.com/config/ubuntu/19.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb  
sudo dpkg -i packages-microsoft-prod.deb
```

安装 .NET Core SDK

更新可供安装的产品, 然后安装 .NET Core SDK。在终端中, 运行以下命令。

```
sudo apt-get update  
sudo apt-get install apt-transport-https  
sudo apt-get update  
sudo apt-get install dotnet-sdk-3.1
```

IMPORTANT

如果收到类似于“找不到包 `dotnet-sdk-3.1`”的错误消息, 请参阅[包管理器疑难解答](#)部分。

安装 ASP.NET Core 运行时

更新可供安装的产品, 然后安装 ASP.NET Core 运行时。在终端中, 运行以下命令。

```
sudo apt-get update  
sudo apt-get install apt-transport-https  
sudo apt-get update  
sudo apt-get install aspnetcore-runtime-3.1
```

IMPORTANT

如果收到类似于“找不到包 aspnetcore-runtime-3.1”的错误消息，请参阅[包管理器疑难解答](#)部分。

安装 .NET Core 运行时

更新可供安装的产品，然后安装 .NET Core 运行时。在终端中，运行以下命令。

```
sudo apt-get update  
sudo apt-get install apt-transport-https  
sudo apt-get update  
sudo apt-get install dotnet-runtime-3.1
```

IMPORTANT

如果收到类似于“找不到包 dotnet-runtime-3.1”的错误消息，请参阅[包管理器疑难解答](#)部分。

如何安装其他版本

添加到包管理器源的包以可改动的格式命名：`{product}-{type}-{version}`。

- **product**

要安装的 .NET 产品的类型。有效选项是：

- dotnet
- aspnetcore

- **type**

选择 SDK 或运行时。有效选项是：

- SDK
- Runtime — 运行时

- **version**

要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本，例如：

- 3.1
- 3.0
- 2.1

尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版的列表，请参阅[.NET Core 依赖项和要求](#)。

示例

- 安装 ASP.NET Core 3.1 运行时：`aspnetcore-runtime-3.1`
- 安装 .NET Core 2.1 运行时：`dotnet-runtime-2.1`
- 安装 .NET Core 3.1 SDK：`dotnet-sdk-3.1`

缺少包

如果包版本组合无效，则它不可用。例如，未安装 ASP.NET Core SDK，所有 SDK 组件都包含在 .NET Core SDK 中。`aspnetcore-sdk-2.2` 的值不正确，应为 `dotnet-sdk-2.2`。有关 .NET Core 支持的 Linux 发行版的列表，请参阅 [.NET Core 依赖项和要求](#)。

包管理器疑难解答

本部分提供有关使用程序包管理器安装 .NET Core 时可能会遇到的常见错误的信息。

无法定位

如果收到类似于“找不到包 {.NET Core 包}”的错误消息，请运行以下命令。

```
sudo dpkg --purge packages-microsoft-prod && sudo dpkg -i packages-microsoft-prod.deb  
sudo apt-get update  
sudo apt-get install {the .NET Core package}
```

如果这不起作用，可使用以下命令运行手动安装。

```
sudo apt-get install -y gpg  
wget -O- https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor -o microsoft.asc.gpg  
sudo mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/  
wget https://packages.microsoft.com/config/ubuntu/19.04/prod.list  
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list  
sudo chown root:root /etc/apt/trusted.gpg.d/microsoft.asc.gpg  
sudo chown root:root /etc/apt/sources.list.d/microsoft-prod.list  
sudo apt-get install -y apt-transport-https  
sudo apt-get update  
sudo apt-get install {the .NET Core package}
```

未能提取

安装 .NET Core 包时，可能会看到类似于

`Failed to fetch ... File has unexpected size ... Mirror sync in progress?` 的错误。一般而言，此错误表示 .NET Core 的包源正在通过更新的包版本进行更新，应稍后重试。升级期间，包源的不可用时间不应超过 30 分钟。如果持续收到此错误超过 30 分钟，请在 <https://github.com/dotnet/core/issues> 中提交问题。

Ubuntu 18.04 包管理器 - 安装 .NET Core

2020/3/30 • [Edit Online](#)

仅在 x64 体系结构上支持包管理器安装。其他体系结构(如 ARM)必须[手动安装 .NET Core SDK](#) 或[手动安装 .NET Core 运行时](#)。有关详细信息, 请参阅[.NET Core 依赖项和要求](#)。

本文介绍如何使用包管理器在 Ubuntu 18.04 上安装 .NET Core。

如果要开发 .NET Core 应用, 请安装 SDK(包括运行时)。或者, 如果只需运行应用程序, 请安装运行时。如果要安装该运行时, 建议安装[ASP.NET Core 运行时](#), 因为它同时包括 .NET Core 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时, 请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息, 请参阅[如何检查是否已安装 .NET Core](#)。

注册 Microsoft 密钥和源

安装 .NET 之前, 需要:

- 注册 Microsoft 密钥。
- 注册产品存储库。
- 安装必需的依赖项。

每台计算机只需要执行一次此操作。

打开终端并运行以下命令。

```
wget https://packages.microsoft.com/config/ubuntu/18.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb  
sudo dpkg -i packages-microsoft-prod.deb
```

安装 .NET Core SDK

更新可供安装的产品, 然后安装 .NET Core SDK。在终端中, 运行以下命令。

```
sudo add-apt-repository universe  
sudo apt-get update  
sudo apt-get install apt-transport-https  
sudo apt-get update  
sudo apt-get install dotnet-sdk-3.1
```

IMPORTANT

如果收到类似于“找不到包 dotnet-sdk-3.1”的错误消息, 请参阅[包管理器疑难解答](#)部分。

安装 ASP.NET Core 运行时

更新可供安装的产品, 然后安装 ASP.NET Core 运行时。在终端中, 运行以下命令。

```
sudo add-apt-repository universe  
sudo apt-get update  
sudo apt-get install apt-transport-https  
sudo apt-get update  
sudo apt-get install aspnetcore-runtime-3.1
```

IMPORTANT

如果收到类似于“找不到包 aspnetcore-runtime-3.1”的错误消息，请参阅[包管理器疑难解答部分](#)。

安装 .NET Core 运行时

更新可供安装的产品，然后安装 .NET Core 运行时。在终端中，运行以下命令。

```
sudo add-apt-repository universe  
sudo apt-get update  
sudo apt-get install apt-transport-https  
sudo apt-get update  
sudo apt-get install dotnet-runtime-3.1
```

IMPORTANT

如果收到类似于“找不到包 dotnet-runtime-3.1”的错误消息，请参阅[包管理器疑难解答部分](#)。

如何安装其他版本

添加到包管理器源的包以可改动的格式命名：`{product}-{type}-{version}`。

- **product**

要安装的 .NET 产品的类型。有效选项是：

- dotnet
- aspnetcore

- **type**

选择 SDK 或运行时。有效选项是：

- SDK
- Runtime — 运行时

- **version**

要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本，例如：

- 3.1
- 3.0
- 2.1

尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版的列表，请参阅[.NET Core 依赖项和要求](#)。

示例

- 安装 ASP.NET Core 3.1 运行时：`aspnetcore-runtime-3.1`
- 安装 .NET Core 2.1 运行时：`dotnet-runtime-2.1`
- 安装 .NET Core 3.1 SDK：`dotnet-sdk-3.1`

缺少包

如果包版本组合无效，则它不可用。例如，未安装 ASP.NET Core SDK，所有 SDK 组件都包含在 .NET Core SDK 中。`aspnetcore-sdk-2.2` 的值不正确，应为 `dotnet-sdk-2.2`。有关 .NET Core 支持的 Linux 发行版的列表，请参阅 [.NET Core 依赖项和要求](#)。

包管理器疑难解答

本部分提供有关使用程序包管理器安装 .NET Core 时可能会遇到的常见错误的信息。

无法定位

如果收到类似于“找不到包 {.NET Core 包}”的错误消息，请运行以下命令。

```
sudo dpkg --purge packages-microsoft-prod && sudo dpkg -i packages-microsoft-prod.deb  
sudo apt-get update  
sudo apt-get install {the .NET Core package}
```

如果这不起作用，可使用以下命令运行手动安装。

```
sudo apt-get install -y gpg  
wget -O- https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor -o microsoft.asc.gpg  
sudo mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/  
wget https://packages.microsoft.com/config/ubuntu/18.04/prod.list  
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list  
sudo chown root:root /etc/apt/trusted.gpg.d/microsoft.asc.gpg  
sudo chown root:root /etc/apt/sources.list.d/microsoft-prod.list  
sudo apt-get install -y apt-transport-https  
sudo apt-get update  
sudo apt-get install {the .NET Core package}
```

未能提取

安装 .NET Core 包时，可能会看到类似于

`Failed to fetch ... File has unexpected size ... Mirror sync in progress?` 的错误。一般而言，此错误表示 .NET Core 的包源正在通过更新的包版本进行更新，应稍后重试。升级期间，包源的不可用时间不应超过 30 分钟。如果持续收到此错误超过 30 分钟，请在 <https://github.com/dotnet/core/issues> 中提交问题。

Ubuntu 16.04 包管理器 - 安装 .NET Core

2020/3/30 • [Edit Online](#)

仅在 x64 体系结构上支持包管理器安装。其他体系结构(如 ARM)必须[手动安装 .NET Core SDK](#) 或[手动安装 .NET Core 运行时](#)。有关详细信息, 请参阅[.NET Core 依赖项和要求](#)。

本文介绍如何使用包管理器在 Ubuntu 16.04 上安装 .NET Core。

如果要开发 .NET Core 应用, 请安装 SDK(包括运行时)。或者, 如果只需运行应用程序, 请安装运行时。如果要安装该运行时, 建议安装[ASP.NET Core 运行时](#), 因为它同时包括 .NET Core 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时, 请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息, 请参阅[如何检查是否已安装 .NET Core](#)。

注册 Microsoft 密钥和源

安装 .NET 之前, 需要:

- 注册 Microsoft 密钥。
- 注册产品存储库。
- 安装必需的依赖项。

每台计算机只需要执行一次此操作。

打开终端并运行以下命令。

```
wget https://packages.microsoft.com/config/ubuntu/16.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb  
sudo dpkg -i packages-microsoft-prod.deb
```

安装 .NET Core SDK

更新可供安装的产品, 然后安装 .NET Core SDK。在终端中, 运行以下命令。

```
sudo apt-get update  
sudo apt-get install apt-transport-https  
sudo apt-get update  
sudo apt-get install dotnet-sdk-3.1
```

IMPORTANT

如果收到类似于“找不到包 dotnet-sdk-3.1”的错误消息, 请参阅[包管理器疑难解答](#)部分。

安装 ASP.NET Core 运行时

更新可供安装的产品, 然后安装 ASP.NET Core 运行时。在终端中, 运行以下命令。

```
sudo apt-get update  
sudo apt-get install apt-transport-https  
sudo apt-get update  
sudo apt-get install aspnetcore-runtime-3.1
```

IMPORTANT

如果收到类似于“找不到包 aspnetcore-runtime-3.1”的错误消息，请参阅[包管理器疑难解答部分](#)。

安装 .NET Core 运行时

更新可供安装的产品，然后安装 .NET Core 运行时。在终端中，运行以下命令。

```
sudo apt-get update  
sudo apt-get install apt-transport-https  
sudo apt-get update  
sudo apt-get install dotnet-runtime-3.1
```

IMPORTANT

如果收到类似于“找不到包 dotnet-runtime-3.1”的错误消息，请参阅[包管理器疑难解答部分](#)。

如何安装其他版本

添加到包管理器源的包以可改动的格式命名：`{product}-{type}-{version}`。

- **product**

要安装的 .NET 产品的类型。有效选项是：

- dotnet
- aspnetcore

- **type**

选择 SDK 或运行时。有效选项是：

- SDK
- Runtime — 运行时

- **version**

要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本，例如：

- 3.1
- 3.0
- 2.1

尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版的列表，请参阅[.NET Core 依赖项和要求](#)。

示例

- 安装 ASP.NET Core 3.1 运行时：`aspnetcore-runtime-3.1`
- 安装 .NET Core 2.1 运行时：`dotnet-runtime-2.1`
- 安装 .NET Core 3.1 SDK：`dotnet-sdk-3.1`

缺少包

如果包版本组合无效，则它不可用。例如，未安装 ASP.NET Core SDK，所有 SDK 组件都包含在 .NET Core SDK 中。`aspnetcore-sdk-2.2` 的值不正确，应为 `dotnet-sdk-2.2`。有关 .NET Core 支持的 Linux 发行版的列表，请参阅 [.NET Core 依赖项和要求](#)。

包管理器疑难解答

本部分提供有关使用程序包管理器安装 .NET Core 时可能会遇到的常见错误的信息。

无法定位

如果收到类似于“找不到包 {.NET Core 包}”的错误消息，请运行以下命令。

```
sudo dpkg --purge packages-microsoft-prod && sudo dpkg -i packages-microsoft-prod.deb  
sudo apt-get update  
sudo apt-get install {the .NET Core package}
```

如果这不起作用，可使用以下命令运行手动安装。

```
sudo apt-get install -y gpg  
wget -O- https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor -o microsoft.asc.gpg  
sudo mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/  
wget https://packages.microsoft.com/config/ubuntu/16.04/prod.list  
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list  
sudo chown root:root /etc/apt/trusted.gpg.d/microsoft.asc.gpg  
sudo chown root:root /etc/apt/sources.list.d/microsoft-prod.list  
sudo apt-get install -y apt-transport-https  
sudo apt-get update  
sudo apt-get install {the .NET Core package}
```

未能提取

安装 .NET Core 包时，可能会看到类似于

`Failed to fetch ... File has unexpected size ... Mirror sync in progress?` 的错误。一般而言，此错误表示 .NET Core 的包源正在通过更新的包版本进行更新，应稍后重试。升级期间，包源的不可用时间不应超过 30 分钟。如果持续收到此错误超过 30 分钟，请在 <https://github.com/dotnet/core/issues> 中提交问题。

CentOS 7 包管理器 - 安装 .NET Core

2020/3/30 • [Edit Online](#)

仅在 x64 体系结构上支持包管理器安装。其他体系结构(如 ARM)必须[手动安装 .NET Core SDK](#) 或[手动安装 .NET Core 运行时](#)。有关详细信息, 请参阅[.NET Core 依赖项和要求](#)。

本文介绍如何使用包管理器在 CentOS 7 上安装 .NET Core。

如果要开发 .NET Core 应用, 请安装 SDK(包括运行时)。或者, 如果只需运行应用程序, 请安装运行时。如果要安装该运行时, 建议安装[ASP.NET Core 运行时](#), 因为它同时包括 .NET Core 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时, 请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息, 请参阅[如何检查是否已安装 .NET Core](#)。

注册 Microsoft 密钥和源

安装 .NET 之前, 需要:

- 注册 Microsoft 密钥。
- 注册产品存储库。
- 安装必需的依赖项。

每台计算机只需要执行一次此操作。

打开终端并运行以下命令。

```
sudo rpm -Uvh https://packages.microsoft.com/config/centos/7/packages-microsoft-prod.rpm
```

安装 .NET Core SDK

更新可供安装的产品, 然后安装 .NET Core SDK。在终端中, 运行以下命令。

```
sudo yum install dotnet-sdk-3.1
```

安装 ASP.NET Core 运行时

更新可供安装的产品, 然后安装 ASP.NET 运行时。在终端中, 运行以下命令。

```
sudo yum install aspnetcore-runtime-3.1
```

安装 .NET Core 运行时

更新可供安装的产品, 然后安装 .NET Core 运行时。在终端中, 运行以下命令。

```
sudo yum install dotnet-runtime-3.1
```

如何安装其他版本

添加到包管理器源的包以可改动的格式命名: `{product}-{type}-{version}`。

- **product**

要安装的 .NET 产品的类型。有效选项是:

- dotnet
- aspnetcore

- **type**

选择 SDK 或运行时。有效选项是:

- SDK
- Runtime — 运行时

- **version**

要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本,例如:

- 3.1
- 3.0
- 2.1

尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版的列表,请参阅 [.NET Core 依赖项和要求](#)。

示例

- 安装 ASP.NET Core 3.1 运行时: `aspnetcore-runtime-3.1`
- 安装 .NET Core 2.1 运行时: `dotnet-runtime-2.1`
- 安装 .NET Core 3.1 SDK: `dotnet-sdk-3.1`

缺少包

如果包版本组合无效,则它不可用。例如,未安装 ASP.NET Core SDK,所有 SDK 组件都包含在 .NET Core SDK 中。`aspnetcore-sdk-2.2` 的值不正确,应为 `dotnet-sdk-2.2`。有关 .NET Core 支持的 Linux 发行版的列表,请参阅 [.NET Core 依赖项和要求](#)。

包管理器疑难解答

本部分提供有关使用程序包管理器安装 .NET Core 时可能会遇到的常见错误的信息。

未能提取

安装 .NET Core 包时,可能会看到类似于

`signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'` 的错误。一般而言,此错误表示 .NET Core 的包源正在通过更新的包版本进行更新,应稍后重试。升级期间,包源的不可用时间不应超过 2 小时。如果持续收到此错误超过 2 小时,请在 <https://github.com/dotnet/core/issues> 中提交问题。

Debian 10 包管理器 - 安装 .NET Core

2020/3/30 • [Edit Online](#)

仅在 x64 体系结构上支持包管理器安装。其他体系结构(如 ARM)必须[手动安装 .NET Core SDK](#) 或[手动安装 .NET Core 运行时](#)。有关详细信息, 请参阅[.NET Core 依赖项和要求](#)。

本文介绍如何使用包管理器在 Debian 10 上安装 .NET Core。

如果要开发 .NET Core 应用, 请安装 SDK(包括运行时)。或者, 如果只需运行应用程序, 请安装运行时。如果要安装该运行时, 建议安装[ASP.NET Core 运行时](#), 因为它同时包括 .NET Core 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时, 请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息, 请参阅[如何检查是否已安装 .NET Core](#)。

注册 Microsoft 密钥和源

安装 .NET 之前, 需要:

- 注册 Microsoft 密钥。
- 注册产品存储库。
- 安装必需的依赖项。

每台计算机只需要执行一次此操作。

打开终端并运行以下命令。

```
wget -O- https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor > microsoft.asc.gpg
sudo mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/
wget https://packages.microsoft.com/config/debian/10/prod.list
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list
sudo chown root:root /etc/apt/trusted.gpg.d/microsoft.asc.gpg
sudo chown root:root /etc/apt/sources.list.d/microsoft-prod.list
```

安装 .NET Core SDK

更新可供安装的产品, 然后安装 .NET Core SDK。在终端中, 运行以下命令。

```
sudo apt-get update
sudo apt-get install apt-transport-https
sudo apt-get update
sudo apt-get install dotnet-sdk-3.1
```

安装 ASP.NET Core 运行时

更新可供安装的产品, 然后安装 ASP.NET 运行时。在终端中, 运行以下命令。

```
sudo apt-get update
sudo apt-get install apt-transport-https
sudo apt-get update
sudo apt-get install aspnetcore-runtime-3.1
```

安装 .NET Core 运行时

更新可供安装的产品，然后安装 .NET Core 运行时。在终端中，运行以下命令。

```
sudo apt-get update  
sudo apt-get install apt-transport-https  
sudo apt-get update  
sudo apt-get install dotnet-runtime-3.1
```

如何安装其他版本

添加到包管理器源的包以可改动的格式命名：`{product}-{type}-{version}`。

- **product**

要安装的 .NET 产品的类型。有效选项是：

- dotnet
- aspnetcore

- **type**

选择 SDK 或运行时。有效选项是：

- SDK
- Runtime — 运行时

- **version**

要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本，例如：

- 3.1
- 3.0
- 2.1

尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版的列表，请参阅 [.NET Core 依赖项和要求](#)。

示例

- 安装 ASP.NET Core 3.1 运行时：`aspnetcore-runtime-3.1`
- 安装 .NET Core 2.1 运行时：`dotnet-runtime-2.1`
- 安装 .NET Core 3.1 SDK：`dotnet-sdk-3.1`

缺少包

如果包版本组合无效，则它不可用。例如，未安装 ASP.NET Core SDK，所有 SDK 组件都包含在 .NET Core SDK 中。`aspnetcore-sdk-2.2` 的值不正确，应为 `dotnet-sdk-2.2`。有关 .NET Core 支持的 Linux 发行版的列表，请参阅 [.NET Core 依赖项和要求](#)。

包管理器疑难解答

本部分提供有关使用程序包管理器安装 .NET Core 时可能会遇到的常见错误的信息。

未能提取

安装 .NET Core 包时，可能会看到类似于

`Failed to fetch ... File has unexpected size ... Mirror sync in progress?` 的错误。一般而言，此错误表示 .NET Core 的包源正在通过更新的包版本进行更新，应稍后重试。升级期间，包源的不可用时间不应超过 30 分钟。如果持续收到此错误超过 30 分钟，请在 <https://github.com/dotnet/core/issues> 中提交问题。

Debian 9 包管理器 - 安装 .NET Core

2020/3/30 • [Edit Online](#)

仅在 x64 体系结构上支持包管理器安装。其他体系结构(如 ARM)必须[手动安装 .NET Core SDK](#) 或[手动安装 .NET Core 运行时](#)。有关详细信息, 请参阅[.NET Core 依赖项和要求](#)。

本文介绍如何使用包管理器在 Debian 9 上安装 .NET Core。

如果要开发 .NET Core 应用, 请安装 SDK(包括运行时)。或者, 如果只需运行应用程序, 请安装运行时。如果要安装该运行时, 建议安装[ASP.NET Core 运行时](#), 因为它同时包括 .NET Core 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时, 请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息, 请参阅[如何检查是否已安装 .NET Core](#)。

注册 Microsoft 密钥和源

安装 .NET 之前, 需要:

- 注册 Microsoft 密钥。
- 注册产品存储库。
- 安装必需的依赖项。

每台计算机只需要执行一次此操作。

打开终端并运行以下命令。

```
wget -O- https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor > microsoft.asc.gpg
sudo mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/
wget https://packages.microsoft.com/config/debian/9/prod.list
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list
sudo chown root:root /etc/apt/trusted.gpg.d/microsoft.asc.gpg
sudo chown root:root /etc/apt/sources.list.d/microsoft-prod.list
```

安装 .NET Core SDK

更新可供安装的产品, 然后安装 .NET Core SDK。在终端中, 运行以下命令。

```
sudo apt-get update
sudo apt-get install apt-transport-https
sudo apt-get update
sudo apt-get install dotnet-sdk-3.1
```

安装 ASP.NET Core 运行时

更新可供安装的产品, 然后安装 ASP.NET 运行时。在终端中, 运行以下命令。

```
sudo apt-get update
sudo apt-get install apt-transport-https
sudo apt-get update
sudo apt-get install aspnetcore-runtime-3.1
```

安装 .NET Core 运行时

更新可供安装的产品，然后安装 .NET Core 运行时。在终端中，运行以下命令。

```
sudo apt-get update  
sudo apt-get install apt-transport-https  
sudo apt-get update  
sudo apt-get install dotnet-runtime-3.1
```

如何安装其他版本

添加到包管理器源的包以可改动的格式命名：`{product}-{type}-{version}`。

- **product**

要安装的 .NET 产品的类型。有效选项是：

- dotnet
- aspnetcore

- **type**

选择 SDK 或运行时。有效选项是：

- SDK
- Runtime — 运行时

- **version**

要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本，例如：

- 3.1
- 3.0
- 2.1

尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版的列表，请参阅 [.NET Core 依赖项和要求](#)。

示例

- 安装 ASP.NET Core 3.1 运行时：`aspnetcore-runtime-3.1`
- 安装 .NET Core 2.1 运行时：`dotnet-runtime-2.1`
- 安装 .NET Core 3.1 SDK：`dotnet-sdk-3.1`

缺少包

如果包版本组合无效，则它不可用。例如，未安装 ASP.NET Core SDK，所有 SDK 组件都包含在 .NET Core SDK 中。`aspnetcore-sdk-2.2` 的值不正确，应为 `dotnet-sdk-2.2`。有关 .NET Core 支持的 Linux 发行版的列表，请参阅 [.NET Core 依赖项和要求](#)。

包管理器疑难解答

本部分提供有关使用程序包管理器安装 .NET Core 时可能会遇到的常见错误的信息。

未能提取

安装 .NET Core 包时，可能会看到类似于

`Failed to fetch ... File has unexpected size ... Mirror sync in progress?` 的错误。一般而言，此错误表示 .NET Core 的包源正在通过更新的包版本进行更新，应稍后重试。升级期间，包源的不可用时间不应超过 30 分钟。如果持续收到此错误超过 30 分钟，请在 <https://github.com/dotnet/core/issues> 中提交问题。

Fedora 31 包管理器 - 安装 .NET Core

2020/3/30 • [Edit Online](#)

仅在 x64 体系结构上支持包管理器安装。其他体系结构(如 ARM)必须[手动安装 .NET Core SDK](#) 或[手动安装 .NET Core 运行时](#)。有关详细信息, 请参阅[.NET Core 依赖项和要求](#)。

本文介绍如何使用包管理器在 Fedora 31 上安装 .NET Core。

如果要开发 .NET Core 应用, 请安装 SDK(包括运行时)。或者, 如果只需运行应用程序, 请安装运行时。如果要安装该运行时, 建议安装[ASP.NET Core 运行时](#), 因为它同时包括 .NET Core 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时, 请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息, 请参阅[如何检查是否已安装 .NET Core](#)。

注册 Microsoft 密钥和源

安装 .NET 之前, 需要:

- 注册 Microsoft 密钥。
- 注册产品存储库。
- 安装必需的依赖项。

每台计算机只需要执行一次此操作。

打开终端并运行以下命令。

```
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
sudo wget -O /etc/yum.repos.d/microsoft-prod.repo https://packages.microsoft.com/config/fedora/31/prod.repo
```

安装 .NET Core SDK

更新可供安装的产品, 然后安装 .NET Core SDK。在终端中, 运行以下命令。

```
sudo dnf install dotnet-sdk-3.1
```

安装 ASP.NET Core 运行时

更新可供安装的产品, 然后安装 ASP.NET 运行时。在终端中, 运行以下命令。

```
sudo dnf install aspnetcore-runtime-3.1
```

安装 .NET Core 运行时

更新可供安装的产品, 然后安装 .NET Core 运行时。在终端中, 运行以下命令。

```
sudo dnf install dotnet-runtime-3.1
```

如何安装其他版本

添加到包管理器源的包以可改动的格式命名: `{product}-{type}-{version}`。

- **product**

要安装的 .NET 产品的类型。有效选项是:

- dotnet
- aspnetcore

- **type**

选择 SDK 或运行时。有效选项是:

- SDK
- Runtime — 运行时

- **version**

要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本,

例如:

- 3.1
- 3.0
- 2.1

尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版的列表, 请参阅 [.NET Core 依赖项和要求](#)。

示例

- 安装 ASP.NET Core 3.1 运行时: `aspnetcore-runtime-3.1`
- 安装 .NET Core 2.1 运行时: `dotnet-runtime-2.1`
- 安装 .NET Core 3.1 SDK: `dotnet-sdk-3.1`

缺少包

如果包版本组合无效, 则它不可用。例如, 未安装 ASP.NET Core SDK, 所有 SDK 组件都包含在 .NET Core SDK 中。`aspnetcore-sdk-2.2` 的值不正确, 应为 `dotnet-sdk-2.2`。有关 .NET Core 支持的 Linux 发行版的列表, 请参阅 [.NET Core 依赖项和要求](#)。

包管理器疑难解答

本部分提供有关使用程序包管理器安装 .NET Core 时可能会遇到的常见错误的信息。

未能提取

安装 .NET Core 包时, 可能会看到类似于

`signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'` 的错误。一般而言, 此错误表示 .NET Core 的包源正在通过更新的包版本进行更新, 应稍后重试。升级期间, 包源的不可用时间不应超过 2 小时。如果持续收到此错误超过 2 小时, 请在 <https://github.com/dotnet/core/issues> 中提交问题。

Fedora 30 包管理器 - 安装 .NET Core

2020/3/30 • [Edit Online](#)

仅在 x64 体系结构上支持包管理器安装。其他体系结构(如 ARM)必须[手动安装 .NET Core SDK](#) 或[手动安装 .NET Core 运行时](#)。有关详细信息, 请参阅[.NET Core 依赖项和要求](#)。

本文介绍如何使用包管理器在 Fedora 30 上安装 .NET Core。

如果要开发 .NET Core 应用, 请安装 SDK(包括运行时)。或者, 如果只需运行应用程序, 请安装运行时。如果要安装该运行时, 建议安装[ASP.NET Core 运行时](#), 因为它同时包括 .NET Core 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时, 请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息, 请参阅[如何检查是否已安装 .NET Core](#)。

注册 Microsoft 密钥和源

安装 .NET 之前, 需要:

- 注册 Microsoft 密钥。
- 注册产品存储库。
- 安装必需的依赖项。

每台计算机只需要执行一次此操作。

打开终端并运行以下命令。

```
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
sudo wget -O /etc/yum.repos.d/microsoft-prod.repo https://packages.microsoft.com/config/fedora/30/prod.repo
```

安装 .NET Core SDK

更新可供安装的产品, 然后安装 .NET Core SDK。在终端中, 运行以下命令。

```
sudo dnf install dotnet-sdk-3.1
```

安装 ASP.NET Core 运行时

更新可供安装的产品, 然后安装 ASP.NET 运行时。在终端中, 运行以下命令。

```
sudo dnf install aspnetcore-runtime-3.1
```

安装 .NET Core 运行时

更新可供安装的产品, 然后安装 .NET Core 运行时。在终端中, 运行以下命令。

```
sudo dnf install dotnet-runtime-3.1
```

如何安装其他版本

添加到包管理器源的包以可改动的格式命名: `{product}-{type}-{version}`。

- **product**

要安装的 .NET 产品的类型。有效选项是:

- dotnet
- aspnetcore

- **type**

选择 SDK 或运行时。有效选项是:

- SDK
- Runtime — 运行时

- **version**

要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本,例如:

- 3.1
- 3.0
- 2.1

尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版的列表,请参阅 [.NET Core 依赖项和要求](#)。

示例

- 安装 ASP.NET Core 3.1 运行时: `aspnetcore-runtime-3.1`
- 安装 .NET Core 2.1 运行时: `dotnet-runtime-2.1`
- 安装 .NET Core 3.1 SDK: `dotnet-sdk-3.1`

缺少包

如果包版本组合无效,则它不可用。例如,未安装 ASP.NET Core SDK,所有 SDK 组件都包含在 .NET Core SDK 中。`aspnetcore-sdk-2.2` 的值不正确,应为 `dotnet-sdk-2.2`。有关 .NET Core 支持的 Linux 发行版的列表,请参阅 [.NET Core 依赖项和要求](#)。

包管理器疑难解答

本部分提供有关使用程序包管理器安装 .NET Core 时可能会遇到的常见错误的信息。

未能提取

安装 .NET Core 包时,可能会看到类似于

`signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'` 的错误。一般而言,此错误表示 .NET Core 的包源正在通过更新的包版本进行更新,应稍后重试。升级期间,包源的不可用时间不应超过 2 小时。如果持续收到此错误超过 2 小时,请在 <https://github.com/dotnet/core/issues> 中提交问题。

Fedora 29 包管理器 - 安装 .NET Core

2020/3/30 • [Edit Online](#)

仅在 x64 体系结构上支持包管理器安装。其他体系结构(如 ARM)必须[手动安装 .NET Core SDK](#) 或[手动安装 .NET Core 运行时](#)。有关详细信息, 请参阅[.NET Core 依赖项和要求](#)。

本文介绍如何使用包管理器在 Fedora 29 上安装 .NET Core。

如果要开发 .NET Core 应用, 请安装 SDK(包括运行时)。或者, 如果只需运行应用程序, 请安装运行时。如果要安装该运行时, 建议安装[ASP.NET Core 运行时](#), 因为它同时包括 .NET Core 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时, 请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息, 请参阅[如何检查是否已安装 .NET Core](#)。

注册 Microsoft 密钥和源

安装 .NET 之前, 需要:

- 注册 Microsoft 密钥。
- 注册产品存储库。
- 安装必需的依赖项。

每台计算机只需要执行一次此操作。

打开终端并运行以下命令。

```
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
sudo wget -O /etc/yum.repos.d/microsoft-prod.repo https://packages.microsoft.com/config/fedora/29/prod.repo
```

安装 .NET Core SDK

更新可供安装的产品, 然后安装 .NET Core SDK。在终端中, 运行以下命令。

```
sudo dnf install dotnet-sdk-3.1
```

安装 ASP.NET Core 运行时

更新可供安装的产品, 然后安装 ASP.NET 运行时。在终端中, 运行以下命令。

```
sudo dnf install aspnetcore-runtime-3.1
```

安装 .NET Core 运行时

更新可供安装的产品, 然后安装 .NET Core 运行时。在终端中, 运行以下命令。

```
sudo dnf install dotnet-runtime-3.1
```

如何安装其他版本

添加到包管理器源的包以可改动的格式命名：`{product}-{type}-{version}`。

- **product**

要安装的 .NET 产品的类型。有效选项是：

- dotnet
- aspnetcore

- **type**

选择 SDK 或运行时。有效选项是：

- SDK
- Runtime — 运行时

- **version**

要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本，例如：

- 3.1
- 3.0
- 2.1

尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版的列表，请参阅 [.NET Core 依赖项和要求](#)。

示例

- 安装 ASP.NET Core 3.1 运行时：`aspnetcore-runtime-3.1`
- 安装 .NET Core 2.1 运行时：`dotnet-runtime-2.1`
- 安装 .NET Core 3.1 SDK：`dotnet-sdk-3.1`

缺少包

如果包版本组合无效，则它不可用。例如，未安装 ASP.NET Core SDK，所有 SDK 组件都包含在 .NET Core SDK 中。`aspnetcore-sdk-2.2` 的值不正确，应为 `dotnet-sdk-2.2`。有关 .NET Core 支持的 Linux 发行版的列表，请参阅 [.NET Core 依赖项和要求](#)。

包管理器疑难解答

本部分提供有关使用程序包管理器安装 .NET Core 时可能会遇到的常见错误的信息。

未能提取

安装 .NET Core 包时，可能会看到类似于

`signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'` 的错误。一般而言，此错误表示 .NET Core 的包源正在通过更新的包版本进行更新，应稍后重试。升级期间，包源的不可用时间不应超过 2 小时。如果持续收到此错误超过 2 小时，请在 <https://github.com/dotnet/core/issues> 中提交问题。

openSUSE 15 包管理器 - 安装 .NET Core

2020/3/30 • [Edit Online](#)

仅在 x64 体系结构上支持包管理器安装。其他体系结构(如 ARM)必须[手动安装 .NET Core SDK](#) 或[手动安装 .NET Core 运行时](#)。有关详细信息, 请参阅[.NET Core 依赖项和要求](#)。

本文介绍如何使用包管理器在 openSUSE 15 上安装 .NET Core。

如果要开发 .NET Core 应用, 请安装 SDK(包括运行时)。或者, 如果只需运行应用程序, 请安装运行时。如果要安装该运行时, 建议安装[ASP.NET Core 运行时](#), 因为它同时包括 .NET Core 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时, 请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息, 请参阅[如何检查是否已安装 .NET Core](#)。

注册 Microsoft 密钥和源

安装 .NET 之前, 需要:

- 注册 Microsoft 密钥。
- 注册产品存储库。
- 安装必需的依赖项。

每台计算机只需要执行一次此操作。

打开终端并运行以下命令。

```
sudo zypper install libicu
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
wget https://packages.microsoft.com/config/opensuse/15/prod.repo
sudo mv prod.repo /etc/zypp/repos.d/microsoft-prod.repo
sudo chown root:root /etc/zypp/repos.d/microsoft-prod.repo
```

安装 .NET Core SDK

更新可供安装的产品, 然后安装 .NET Core SDK。在终端中, 运行以下命令。

```
sudo zypper install dotnet-sdk-3.1
```

安装 ASP.NET Core 运行时

更新可供安装的产品, 然后安装 ASP.NET 运行时。在终端中, 运行以下命令。

```
sudo zypper install aspnetcore-runtime-3.1
```

安装 .NET Core 运行时

更新可供安装的产品, 然后安装 .NET Core 运行时。在终端中, 运行以下命令。

```
sudo zypper install dotnet-runtime-3.1
```

如何安装其他版本

添加到包管理器源的包以可改动的格式命名：`{product}-{type}-{version}`。

- **product**

要安装的 .NET 产品的类型。有效选项是：

- dotnet
- aspnetcore

- **type**

选择 SDK 或运行时。有效选项是：

- SDK
- Runtime — 运行时

- **version**

要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本，例如：

- 3.1
- 3.0
- 2.1

尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版的列表，请参阅 [.NET Core 依赖项和要求](#)。

示例

- 安装 ASP.NET Core 3.1 运行时：`aspnetcore-runtime-3.1`
- 安装 .NET Core 2.1 运行时：`dotnet-runtime-2.1`
- 安装 .NET Core 3.1 SDK：`dotnet-sdk-3.1`

缺少包

如果包版本组合无效，则它不可用。例如，未安装 ASP.NET Core SDK，所有 SDK 组件都包含在 .NET Core SDK 中。`aspnetcore-sdk-2.2` 的值不正确，应为 `dotnet-sdk-2.2`。有关 .NET Core 支持的 Linux 发行版的列表，请参阅 [.NET Core 依赖项和要求](#)。

包管理器疑难解答

本部分提供有关使用程序包管理器安装 .NET Core 时可能会遇到的常见错误的信息。

未能提取

安装 .NET Core 包时，可能会看到类似于

`signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'` 的错误。一般而言，此错误表示 .NET Core 的包源正在通过更新的包版本进行更新，应稍后重试。升级期间，包源的不可用时间不应超过 2 小时。如果持续收到此错误超过 2 小时，请在 <https://github.com/dotnet/core/issues> 中提交问题。

RHEL 8 包管理器 - 安装 .NET Core

2020/3/30 • [Edit Online](#)

仅在 x64 体系结构上支持包管理器安装。其他体系结构(如 ARM)必须[手动安装 .NET Core SDK](#) 或[手动安装 .NET Core 运行时](#)。有关详细信息, 请参阅[.NET Core 依赖项和要求](#)。

本文介绍如何使用包管理器在 RHEL 8 上安装 .NET Core。

如果要开发 .NET Core 应用, 请安装 SDK(包括运行时)。或者, 如果只需运行应用程序, 请安装运行时。如果要安装该运行时, 建议安装[ASP.NET Core 运行时](#), 因为它同时包括 .NET Core 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时, 请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息, 请参阅[如何检查是否已安装 .NET Core](#)。

注册 Red Hat 订阅

若要在 RHEL 上从 Red Hat 安装 .NET Core, 首先需要使用 Red Hat 订阅管理器进行注册。如果未在系统上完成此操作, 或者不确定是否完成了此操作, 请参阅[适用于 .NET Core 的 Red Hat 产品文档](#)。

安装 .NET Core SDK

向订阅管理器注册后, 你就可安装并启用 .NET Core SDK。在终端中, 运行以下命令。

```
sudo dnf update  
sudo dnf install dotnet-sdk-3.1
```

安装 ASP.NET Core 运行时

向订阅管理器注册后, 你就可安装并启用 ASP.NET Core 运行时。在终端中, 运行以下命令。

```
sudo dnf update  
sudo dnf install aspnetcore-runtime-3.1
```

安装 .NET Core 运行时

向订阅管理器注册后, 你就可安装并启用 .NET Core 运行时。在终端中, 运行以下命令。

```
sudo dnf update  
sudo dnf install dotnet-runtime-3.1
```

请参阅

- [在 Red Hat Enterprise Linux 8 上使用 .NET Core 3.1](#)

RHEL 7 包管理器 - 安装 .NET Core

2020/3/30 • [Edit Online](#)

仅在 x64 体系结构上支持包管理器安装。其他体系结构(如 ARM)必须[手动安装 .NET Core SDK](#) 或[手动安装 .NET Core 运行时](#)。有关详细信息, 请参阅[.NET Core 依赖项和要求](#)。

本文介绍如何使用包管理器在 RHEL 7 上安装 .NET Core。

如果要开发 .NET Core 应用, 请安装 SDK(包括运行时)。或者, 如果只需运行应用程序, 请安装运行时。如果要安装该运行时, 建议安装[ASP.NET Core 运行时](#), 因为它同时包括 .NET Core 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时, 请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息, 请参阅[如何检查是否已安装 .NET Core](#)。

注册 Red Hat 订阅

若要在 RHEL 上从 Red Hat 安装 .NET Core, 首先需要使用 Red Hat 订阅管理器进行注册。如果未在系统上完成此操作, 或者不确定是否完成了此操作, 请参阅[适用于 .NET Core 的 Red Hat 产品文档](#)。

安装 .NET Core SDK

向订阅管理器注册后, 你就可安装并启用 .NET Core SDK。在终端中, 运行以下命令以启用并安装 RHEL 7 dotnet 通道。

```
subscription-manager repos --enable=rhel-7-server-dotnet-rpms
yum install rh-dotnet31 -y
scl enable rh-dotnet31 bash
```

安装 ASP.NET Core 运行时

向订阅管理器注册后, 你就可安装并启用 ASP.NET Core 运行时。在终端中, 运行以下命令。

```
subscription-manager repos --enable=rhel-7-server-dotnet-rpms
yum install rh-dotnet31-aspnetcore-runtime-3.1 -y
scl enable rh-dotnet31 bash
```

安装 .NET Core 运行时

向订阅管理器注册后, 你就可安装并启用 .NET Core 运行时。在终端中, 运行以下命令。

```
subscription-manager repos --enable=rhel-7-server-dotnet-rpms
yum install rh-dotnet31-dotnet-runtime-3.1 -y
scl enable rh-dotnet31 bash
```

另请参阅

- [在 Red Hat Enterprise Linux 7 上使用 .NET Core 3.1](#)

SLES 15 包管理器 - 安装 .NET Core

2020/3/30 • [Edit Online](#)

仅在 x64 体系结构上支持包管理器安装。其他体系结构(如 ARM)必须[手动安装 .NET Core SDK](#) 或[手动安装 .NET Core 运行时](#)。有关详细信息, 请参阅[.NET Core 依赖项和要求](#)。

本文介绍如何使用包管理器在 SLES 15 上安装 .NET Core。

如果要开发 .NET Core 应用, 请安装 SDK(包括运行时)。或者, 如果只需运行应用程序, 请安装运行时。如果要安装该运行时, 建议安装[ASP.NET Core 运行时](#), 因为它同时包括 .NET Core 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时, 请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息, 请参阅[如何检查是否已安装 .NET Core](#)。

注册 Microsoft 密钥和源

安装 .NET 之前, 需要:

- 注册 Microsoft 密钥。
- 注册产品存储库。
- 安装必需的依赖项。

每台计算机只需要执行一次此操作。

打开终端并运行以下命令。

```
sudo rpm -Uvh https://packages.microsoft.com/config/sles/15/packages-microsoft-prod.rpm
```

安装 .NET Core SDK

更新可供安装的产品, 然后安装 .NET Core SDK。在终端中, 运行以下命令。

```
sudo zypper install dotnet-sdk-3.1
```

安装 ASP.NET Core 运行时

更新可供安装的产品, 然后安装 ASP.NET 运行时。在终端中, 运行以下命令。

```
sudo zypper install aspnetcore-runtime-3.1
```

安装 .NET Core 运行时

更新可供安装的产品, 然后安装 .NET Core 运行时。在终端中, 运行以下命令。

```
sudo zypper install dotnet-runtime-3.1
```

如何安装其他版本

添加到包管理器源的包以可改动的格式命名：`{product}-{type}-{version}`。

- **product**

要安装的 .NET 产品的类型。有效选项是：

- dotnet
- aspnetcore

- **type**

选择 SDK 或运行时。有效选项是：

- SDK
- Runtime — 运行时

- **version**

要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本，例如：

- 3.1
- 3.0
- 2.1

尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版的列表，请参阅 [.NET Core 依赖项和要求](#)。

示例

- 安装 ASP.NET Core 3.1 运行时：`aspnetcore-runtime-3.1`
- 安装 .NET Core 2.1 运行时：`dotnet-runtime-2.1`
- 安装 .NET Core 3.1 SDK：`dotnet-sdk-3.1`

缺少包

如果包版本组合无效，则它不可用。例如，未安装 ASP.NET Core SDK，所有 SDK 组件都包含在 .NET Core SDK 中。`aspnetcore-sdk-2.2` 的值不正确，应为 `dotnet-sdk-2.2`。有关 .NET Core 支持的 Linux 发行版的列表，请参阅 [.NET Core 依赖项和要求](#)。

包管理器疑难解答

本部分提供有关使用程序包管理器安装 .NET Core 时可能会遇到的常见错误的信息。

未能提取

安装 .NET Core 包时，可能会看到类似于

`signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'` 的错误。一般而言，此错误表示 .NET Core 的包源正在通过更新的包版本进行更新，应稍后重试。升级期间，包源的不可用时间不应超过 2 小时。如果持续收到此错误超过 2 小时，请在 <https://github.com/dotnet/core/issues> 中提交问题。

SLES 12 包管理器 - 安装 .NET Core

2020/3/30 • [Edit Online](#)

仅在 x64 体系结构上支持包管理器安装。其他体系结构(如 ARM)必须[手动安装 .NET Core SDK](#) 或[手动安装 .NET Core 运行时](#)。有关详细信息, 请参阅[.NET Core 依赖项和要求](#)。

本文介绍如何使用包管理器在 SLES 12 上安装 .NET Core。

如果要开发 .NET Core 应用, 请安装 SDK(包括运行时)。或者, 如果只需运行应用程序, 请安装运行时。如果要安装该运行时, 建议安装[ASP.NET Core 运行时](#), 因为它同时包括 .NET Core 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时, 请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息, 请参阅[如何检查是否已安装 .NET Core](#)。

注册 Microsoft 密钥和源

安装 .NET 之前, 需要:

- 注册 Microsoft 密钥。
- 注册产品存储库。
- 安装必需的依赖项。

每台计算机只需要执行一次此操作。

打开终端并运行以下命令。

```
sudo rpm -Uvh https://packages.microsoft.com/config/sles/12/packages-microsoft-prod.rpm
```

安装 .NET Core SDK

更新可供安装的产品, 然后安装 .NET Core SDK。在终端中, 运行以下命令。

```
sudo zypper install dotnet-sdk-3.1
```

安装 ASP.NET Core 运行时

更新可供安装的产品, 然后安装 ASP.NET 运行时。在终端中, 运行以下命令。

```
sudo zypper install aspnetcore-runtime-3.1
```

安装 .NET Core 运行时

更新可供安装的产品, 然后安装 .NET Core 运行时。在终端中, 运行以下命令。

```
sudo zypper install dotnet-runtime-3.1
```

如何安装其他版本

添加到包管理器源的包以可改动的格式命名：`{product}-{type}-{version}`。

- **product**

要安装的 .NET 产品的类型。有效选项是：

- dotnet
- aspnetcore

- **type**

选择 SDK 或运行时。有效选项是：

- SDK
- Runtime — 运行时

- **version**

要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本，例如：

- 3.1
- 3.0
- 2.1

尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版的列表，请参阅 [.NET Core 依赖项和要求](#)。

示例

- 安装 ASP.NET Core 3.1 运行时：`aspnetcore-runtime-3.1`
- 安装 .NET Core 2.1 运行时：`dotnet-runtime-2.1`
- 安装 .NET Core 3.1 SDK：`dotnet-sdk-3.1`

缺少包

如果包版本组合无效，则它不可用。例如，未安装 ASP.NET Core SDK，所有 SDK 组件都包含在 .NET Core SDK 中。`aspnetcore-sdk-2.2` 的值不正确，应为 `dotnet-sdk-2.2`。有关 .NET Core 支持的 Linux 发行版的列表，请参阅 [.NET Core 依赖项和要求](#)。

包管理器疑难解答

本部分提供有关使用程序包管理器安装 .NET Core 时可能会遇到的常见错误的信息。

未能提取

安装 .NET Core 包时，可能会看到类似于

`signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'` 的错误。

一般而言，此错误表示 .NET Core 的包源正在通过更新的包版本进行更新，应稍后重试。升级期间，包源的不可用时间不应超过 2 小时。如果持续收到此错误超过 2 小时，请在 <https://github.com/dotnet/core/issues> 中提交问题。

如何为 .NET Core 安装本地化的 IntelliSense 文件

2020/3/18 • [Edit Online](#)

IntelliSense 是一种代码完成辅助工具，可以在不同的集成开发环境 (IDE) 中使用，例如 Visual Studio。默认情况下，在开发 .NET Core 项目时，SDK 仅包含英语版本的 IntelliSense 文件。本文介绍：

- 如何安装这些文件的本地化版本。
- 如何修改 Visual Studio 安装以使用其他语言。

系统必备

- [.NET Core 3.0 SDK 或更高版本。](#)
- [Visual Studio 2019 版本 16.3 或更高版本。](#)

下载并安装本地化的 IntelliSense 文件

IMPORTANT

此过程需具有管理员权限，才能将 IntelliSense 文件复制到 .NET Core 安装文件夹中。

1. 转到[下载 IntelliSense 文件](#)页面。
2. 下载要使用的语言和版本的 IntelliSense 文件。
3. 提取 zip 文件的内容。
4. 导航到 .NET Core Intellisense 文件夹。
 - a. 导航至 .NET Core 安装文件夹。默认情况下，它位于 %ProgramFiles%\dotnet\packs 下。
 - b. 选择要为其安装 IntelliSense 的 SDK，然后导航到关联的路径。有下列选项：

SDK	文件夹
.NET Core	Microsoft.NETCore.App.Ref
Windows 桌面	Microsoft.WindowsDesktop.App.Ref
.NET Standard	NETStandard.Library.Ref

- c. 导航到要为其安装本地化 IntelliSense 的版本。例如，3.1.0。
 - d. 打开 ref 文件夹。
 - e. 打开 moniker 文件夹。例如，netcoreapp3.1。
- 因此，要导航到的完整路径看起来将类似于 C:\Program Files\dotnet\packs\Microsoft.NETCore.App.Ref\3.1.0\ref\netcoreapp3.1。
5. 在刚打开的 moniker 文件夹中创建一个子文件夹。文件夹名称指示要使用的语言。下表指定了不同的选项：

巴西葡萄牙语	pt-br
中文(简体)	zh-hans
中文(繁体)	zh-hant
法语	fr
德语	de
意大利语	it
日语	ja
朝鲜语	ko
俄语	ru
西班牙语	es

6. 将在步骤3中提取的 .xml 文件复制到此新文件夹。.xml 文件按 SDK 文件夹细分，因此，请将它们复制到步骤 4 中选择的相应 SDK。

修改 Visual Studio 语言

要使 Visual Studio 使用其他语言的 IntelliSense，请安装适当的语言包。这可以在[安装过程中](#)完成，也可以之后通过修改 Visual Studio 安装来完成。如果已将 Visual Studio 配置为所需的语言，那么 IntelliSense 安装已准备就绪。

安装语言包

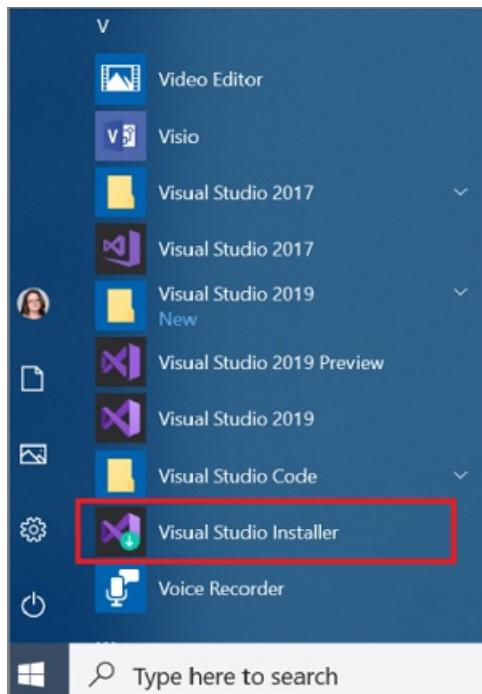
如果在安装过程中未安装所需的语言包，请按以下步骤更新 Visual Studio 来安装语言包：

IMPORTANT

若要安装、更新或修改 Visual Studio，必须使用具有管理员权限的帐户登录。有关详细信息，请参阅[用户权限与 Visual Studio](#)。

1. 在计算机上找到 Visual Studio 安装程序。

例如，在运行 Windows 10 的计算机上，选择“开始”，然后滚动到字母“V”，它作为“Visual Studio 安装程序”在那里列出。



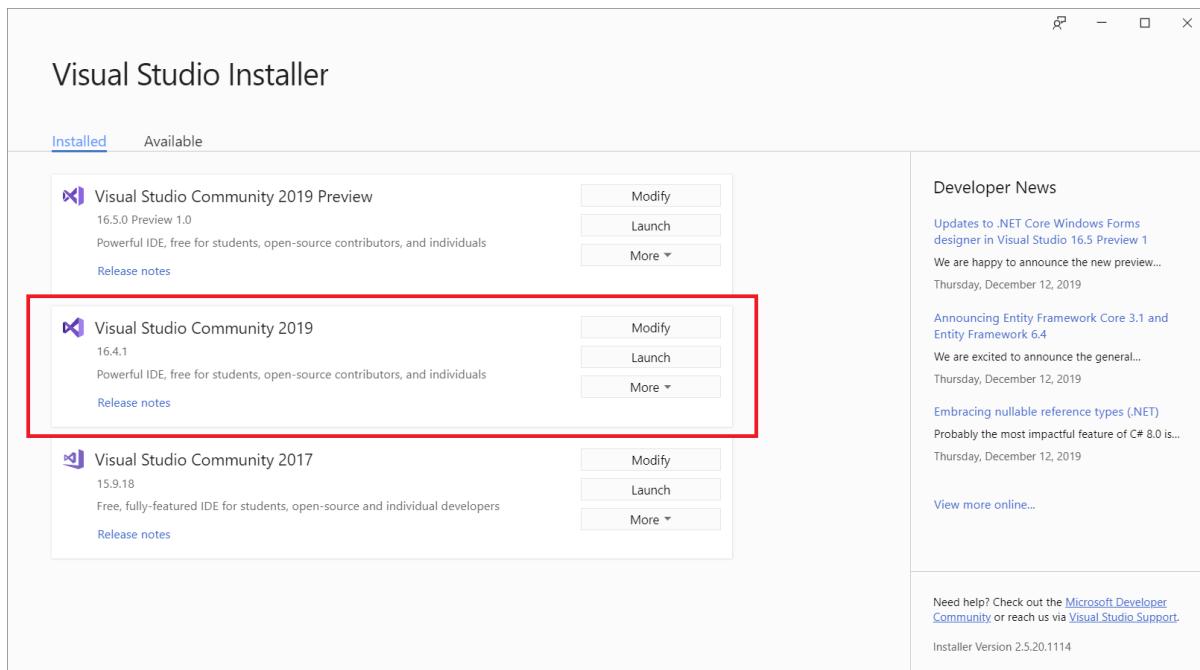
NOTE

还可以在以下位置中找到 Visual Studio 安装程序：

```
C:\Program Files (x86)\Microsoft Visual Studio\Installer\vs_installer.exe
```

可能需要先更新安装程序，然后才能继续操作。如果是这样，请按照提示操作。

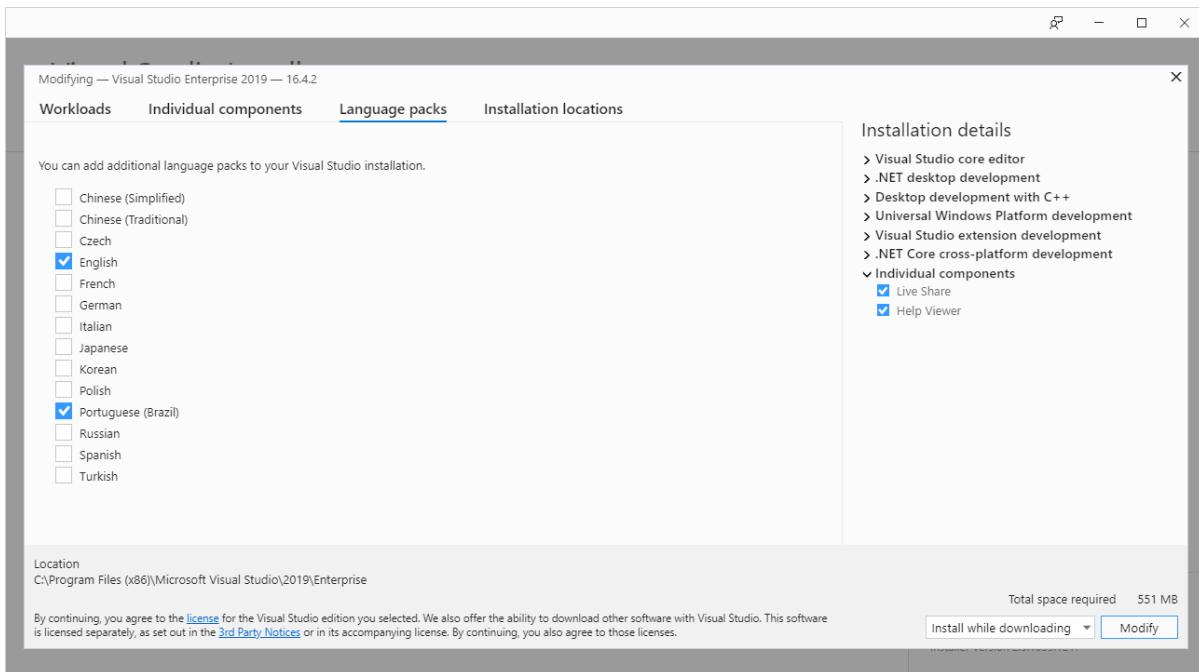
2. 在安装程序中，查找要为其添加语言包的 Visual Studio 版本，然后选择“修改”。



IMPORTANT

如果未看到“修改”按钮，但看到了“更新”按钮，则需要先更新 Visual Studio，才能修改安装。更新完成后，应会显示“修改”按钮。

3. 在“语言包”选项卡中，选择或取消选择要安装或卸载的语言。



4. 选择“修改”。更新开始。

修改 Visual Studio 中的语言设置

安装所需的语言包后，请修改 Visual Studio 设置以使用其他语言：

1. 打开 Visual Studio。
2. 在“启动”窗口中，选择“继续但无需代码”。
3. 在菜单栏上，选择“工具” > “选项”。“选项”对话框随即打开。
4. 在“环境”节点下，选择“国际设置”。
5. 在“语言”下拉列表中，选择所需的语言。选择“确定”。
6. 随即会显示一个对话框，告知必须重启 Visual Studio 才能使所做的更改生效。选择“确定”。
7. 重新启动 Visual Studio。

之后，当打开面向刚安装的 IntelliSense 文件版本的 .NET Core 项目时，IntelliSense 应会按预期方式工作。

另请参阅

- [Visual Studio 中的 IntelliSense](#)

.NET Core 入门

2020/3/18 • [Edit Online](#)

本文提供 .NET Core 入门的相关信息。可在 Windows、Linux 和 macOS 上安装 .NET Core。你可在最喜欢的文本编辑器中编写代码并生成跨平台的库和应用程序。

如果不确定 .NET Core 是什么或其与其他 .NET 技术的关系，请首先参阅 [What is .NET \(.NET 是什么\)](#) 概述。简单地说，.NET Core 是一个跨平台的开放源代码 .NET 实现。

创建应用程序

首先，在计算机上下载并安装 [.NET Core SDK](#)。

然后，打开某一终端，如 PowerShell、命令提示符或 Bash。键入以下 `dotnet` 命令以创建并运行 C# 应用程序：

```
dotnet new console --output sample1  
dotnet run --project sample1
```

您应看到以下输出：

```
Hello World!
```

祝贺你！现已创建了一个简单的 .NET Core 应用程序。还可以使用 [Visual Studio Code](#)、[Visual Studio](#)(仅限 Windows)或 [Visual Studio for Mac](#)(仅限 macOS)来创建 .NET Core 应用程序。

教程

通过以下分步教程着手开发 .NET Core 应用程序：

- [Windows](#)
- [Linux](#)
- [macOS](#)
- 在 [Visual Studio 2019 中创建第一个 .NET Core 控制台应用程序](#)
- 在 [Visual Studio 中使用 .NET Standard 生成类库](#)
- 使用 [.NET Core CLI 实现 .NET Core 入门](#)



观看第 9 频道上的如何安装和使用 [Visual Studio Code](#) 和 [.NET Core](#) 视频。



观看 YouTube 上的 [.NET Core 101](#) 视频。

请参阅 [.NET Core 依赖项和要求](#)一文，以获取支持的 Windows 版本列表。

.NET Core 3.1 的新增功能

2020/3/18 • [Edit Online](#)

本文介绍了 .NET Core 3.1 中的新增功能。此版本包含对 .NET Core 3.0 的细微改进，重点介绍小型但重要的修复。
.NET Core 3.1 中最重要的特性为，它是[长期支持 \(LTS\)](#) 版本。

如果使用的是 Visual Studio 2019，则必须更新到 [Visual Studio 2019 版本 16.4](#) 才能使用 .NET Core 3.1 项目。有关 Visual Studio 中新增功能的详细信息，请参阅 [Visual Studio 2019 版本 16.4 中的新增功能](#)。

Visual Studio for Mac 也支持 .NET Core 3.1，并且 Visual Studio for Mac 8.4 中就包括 .NET Core 3.1。

有关版本的详细信息，请参阅 [.NET Core 3.1 公告](#)。

- 在 Windows、macOS 或 Linux 上[下载并开始使用 .NET Core 3.1](#)。

长期支持

.NET Core 3.1 是未来三年包含来自 Microsoft 的支持的 LTS 版本。强烈建议将应用移到 .NET Core 3.1。其他主要版本的当前生命周期如下所示：

RELEASE	■
.NET Core 3.0	生命周期终结于 2020 年 3 月 3 日。
.NET Core 2.2	生命周期终结于 2019 年 12 月 23 日。
.NET Core 2.1	生命周期终结于 2021 年 8 月 21 日。

有关详细信息，请参阅 [.NET Core 支持策略](#)。

macOS appHost 和公证

仅 macOS

从已公证的适用于 macOS 的 .NET Core SDK 3.1 开始，默认已禁用 appHost 设置。有关详细信息，请参阅 [macOS Catalina 公证以及对 .NET Core 下载和项目的影响](#)。

启用 appHost 设置后，.NET Core 在生成或发布时将生成本机 Mach-O 可执行文件。如果使用 `dotnet run` 命令从源代码中运行应用，或通过启动 Mach-O 可执行文件直接运行应用，则应用会在 appHost 的上下文中运行。

如果没有 appHost，用户就只能使用 `dotnet <filename.dll>` 命令启动[依赖于运行时](#)的应用。发布[独立](#)应用时，始终会创建 appHost。

可以在项目级别配置 appHost，或通过 `-p:UseAppHost` 参数切换特定 `dotnet` 命令的 appHost：

- 项目文件

```
<PropertyGroup>
  <UseAppHost>true</UseAppHost>
</PropertyGroup>
```

- 命令行参数

```
dotnet run -p:UseAppHost=true
```

有关 `UseAppHost` 设置的详细信息, 请参阅 [Microsoft.NETSdk 的 MSBuild 属性](#)。

Windows 窗体

仅限 Windows

WARNING

Windows 窗体中发生重大变更。

旧版控件包含在 Windows 窗体中, 这些窗体在一段时间内无法在 Visual Studio 设计器工具箱中使用。它们已替换为 .NET Framework 2.0 中的新控件。它们已从适用于 .NET Core 3.1 的桌面 SDK 中删除。

		API
DataGrid	DataGridView	DataGridCell DataGridRow DataGridTableCollection DataGridColumnCollection DataGridTableStyle DataGridColumnStyle DataGridLineStyle DataGridParentRowsLabel DataGridParentRowsLabelStyle DataGridBoolColumn DataGridTextBox GridColumnStylesCollection GridTableStylesCollection HitTestType
ToolBar	ToolStrip	ToolBarAppearance
ToolBarButton	ToolStripButton	ToolBarButtonClickEventArgs ToolBarButtonClickEventHandler ToolBarButtonStyle ToolBar.TextAlign
ContextMenu	ContextMenuStrip	
Menu	ToolStripDropDown ToolStripDropDownMenu	MenuItemCollection
MainMenu	MenuStrip	
MenuItem	ToolStripMenuItem	

我们建议你将应用程序更新到 .NET Core 3.1 并移动到替换控件。替换控件是一个简单的过程, 本质上属于“查找和替换”类型。

C++/CLI

仅限 Windows

已添加对创建 C++/CLI(也称为“托管 C++”)项目的支持。从这些项目生成的二进制文件与 .NET Core 3.0 及更高版本兼容。

若要添加对 Visual Studio 2019 版本 16.4 中的 C++/CLI 的支持, 请安装[“使用 C++ 的桌面开发”工作负载](#)。此工作负载将两个模板添加到 Visual Studio:

- CLR 类库(.NET Core)
- CLR 空项目(.NET Core)

后续步骤

- [查看 .NET Core 3.0 和 3.1 之间的重大变更。](#)
- [查看用于 Windows 窗体应用的 .NET Core 3.1 中的中断性变更。](#)

.NET Core 3.0 的新增功能

2020/4/6 • [Edit Online](#)

本文介绍了 .NET Core 3.0 中的新增功能。最大的增强功能之一是对 Windows 桌面应用程序的支持(仅限 Windows)。通过使用 .NET Core 3.0 SDK Windows 桌面组件, 可移植 Windows 窗体和 Windows Presentation Foundation (WPF) 应用程序。明确地说, 只有在 Windows 上才支持和包含 Windows 桌面组件。有关详细信息, 请参阅本文后面的 [Windows 桌面部分](#)。

.NET Core 3.0 添加了对 C#8.0 的支持。强烈建议使用 [Visual Studio 2019 版本 16.3](#) 或更高版本、[Visual Studio for Mac 8.3](#) 或更高版本, 或具有最新 C# 扩展的 [Visual Studio Code](#)。

立即在 Windows、macOS 或 Linux 上[下载并开始使用 .NET Core 3.0](#)。

有关版本的详细信息, 请参阅 [.NET Core 3.0 公告](#)。

Microsoft 认为 .NET Core RC1 可用于生产环境, 且该软件完全受支持。如果使用的是预览版本, 则必须转换为 RTM 版本才能继续获得支持。

语言改进 C# 8.0

C# 8.0 也是该发布的一部分, 包含[可为空引用类型功能](#)、[异步流](#)和[更多模式](#)。有关 C# 8.0 功能的详细信息, 请参阅 [C# 8.0 中的新增功能](#)。

添加了语言增强功能, 以支持下面详细说明的 API 功能:

- [范围和索引](#)
- [异步流](#)

.NET Standard 2.1

.NET Core 3.0 已实现 .NET Standard 2.1。但是, 默认的 `dotnet new classlib` 模板还是会生成一个面向 .NET Standard 2.0 的项目。若要面向 .NET Standard 2.1, 请编辑项目文件并将 `TargetFramework` 属性更改为 `netstandard2.1`:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.1</TargetFramework>
  </PropertyGroup>

</Project>
```

如果使用 Visual Studio, 则需要 [Visual Studio 2019](#), 这是因为 Visual Studio 2017 不支持 .NET Standard 2.1 或 .NET Core 3.0。

编译/部署

默认可执行文件

.NET Core 现在默认生成[依赖于运行时的可执行文件](#)。对于使用全局安装的 .NET Core 版本的应用程序而言, 这是一种新行为。以前, 仅[独立部署](#)会生成可执行文件。

在 `dotnet build` 或 `dotnet publish` 期间, 将创建一个与你使用的 SDK 的环境和平台相匹配的可执行文件(即 `appHost`)。和其他本机可执行文件一样, 可以使用这些可执行文件执行相同操作, 例如:

- 可以双击可执行文件。
- 可以直接从命令提示符启用应用程序，如 Windows 上的 `myapp.exe`，以及 Linux 和 macOS 上的 `./myapp`。

macOS appHost 和公证

仅 macOS

从已公证的适用于 macOS 的 .NET Core SDK 3.0 开始，默认已禁用用于生成默认可执行文件（即 appHost）的设置。有关详细信息，请参阅 [macOS Catalina 公证以及对 .NET Core 下载和项目的影响](#)。

启用 appHost 设置后，.NET Core 在生成或发布时将生成本机 Mach-O 可执行文件。如果使用 `dotnet run` 命令从源代码中运行应用，或通过启动 Mach-O 可执行文件直接运行应用，则应用会在 appHost 的上下文中运行。

如果没有 appHost，用户就只能使用 `dotnet <filename.dll>` 命令启动 [依赖于运行时](#) 的应用。发布 [独立](#) 应用时，始终会创建 appHost。

可以在项目级别配置 appHost，或通过 `-p:UseAppHost` 参数切换特定 `dotnet` 命令的 appHost：

- 项目文件

```
<PropertyGroup>
  <UseAppHost>true</UseAppHost>
</PropertyGroup>
```

- 命令行参数

```
dotnet run -p:UseAppHost=true
```

有关 `UseAppHost` 设置的详细信息，请参阅 [Microsoft.NET.Sdk 的 MSBuild 属性](#)。

单文件可执行文件

`dotnet publish` 命令支持将应用打包为特定于平台的单文件可执行文件。该可执行文件是自解压缩文件，包含运行应用所需的所有依赖项（包括本机依赖项）。首次运行应用时，应用程序将根据应用名称和生成标识符自解压缩到一个目录中。再次运行应用程序时，启动速度将变快。除非使用了新版本，否则应用程序无需再次进行自解压缩。

若要发布单文件可执行文件，请使用 `dotnet publish` 命令在项目或命令行中设置 `PublishSingleFile`：

```
<PropertyGroup>
  <RuntimeIdentifier>win10-x64</RuntimeIdentifier>
  <PublishSingleFile>true</PublishSingleFile>
</PropertyGroup>
```

- 或 -

```
dotnet publish -r win10-x64 -p:PublishSingleFile=true
```

有关单文件发布的详细信息，请参阅 [单文件捆绑程序设计文档](#)。

程序集链接

.NET core 3.0 SDK 随附了一种工具，可以通过分析 IL 并剪裁未使用的程序集来减小应用的大小。

自包含应用包括运行代码所需的所有内容，而无需在主计算机上安装 .NET。但是，很多时候应用只需要一小部分框架即可运行，并且可以删除其他未使用的库。

.NET Core 现在包含一个设置，将使用 [IL 链接器](#) 工具扫描应用的 IL。此工具将检测哪些代码是必需的，然后剪裁未使用的库。此工具可以显著减少某些应用的部署大小。

要启用此工具, 请使用项目中的 `<PublishTrimmed>` 设置并发布自包含应用:

```
<PropertyGroup>
  <PublishTrimmed>true</PublishTrimmed>
</PropertyGroup>
```

```
dotnet publish -r <rid> -c Release
```

例如, 包含的基本“hello world”新控制台项目模板在发布时命中大小约为 70 MB。通过使用 `<PublishTrimmed>`, 其大小将减少到约 30 MB。

请务必考虑到使用反射或相关动态功能的应用程序或框架(包括 ASP.NET Core 和 WPF)通常会在剪裁时损坏。发生此损坏是因为链接器不知道此动态行为, 并且不能确定反射需要哪些框架类型。可配置 IL 链接器工具以发现这种情况。

最重要的是, 剪裁后务必对应用进行测试。

有关 IL 链接器工具的详细信息, 请参阅[文档](#), 或访问 [mono/linker](#) 存储库。

分层编译

.NET Core 3.0 中默认启用了[分层编译](#)(TC)。此功能使运行时能够更适应地使用实时(JIT)编译器来实现更好的性能。

分层编译的主要优势是提供两种实现实时的方法, 可在低质量快速层或高质量慢速层中编译。质量是指方法的优化程度。这有助于提高应用程序在从启动到稳定状态的各个执行阶段的性能。禁用分层编译后, 每种方法都以同一种方式进行编译, 这种方式倾向于牺牲启动性能来保证稳定状态性能。

启用 TC 后, 以下行为适用于应用启动时的方法编译:

- 如果方法具有预先编译的代码([ReadyToRun](#)), 将使用预生成的代码。
- 否则, 将实时编译该方法。一般来说, 这些方法是泛型而不是值类型。
 - 快速 JIT 可以更快地生成较低质量(优化程度较低)的代码。在 .NET Core 3.0 中, 默认认为不包含循环的方法启用了快速 JIT, 并且启动过程中首选快速 JIT。
 - 完全优化的 JIT 可生成更高质量(优化程度更高)的代码, 但速度更慢。对于不使用快速 JIT 的方法(例如, 如果该方法具有 [MethodImplOptions.AggressiveOptimization](#) 特性), 则使用完全优化的 JIT。

对于频繁调用的方法, 实时编译器最终会在后台创建完全优化的代码。然后, 优化后的代码将替换该方法的预编译代码。

通过快速 JIT 生成的代码可能会运行较慢、分配更多内存或使用更多堆栈空间。如果出现问题, 可以在项目文件中使用此 MSBuild 属性禁用快速 JIT:

```
<PropertyGroup>
  <TieredCompilationQuickJit>false</TieredCompilationQuickJit>
</PropertyGroup>
```

若要完全禁用 TC, 请在项目文件中使用此 MSBuild 属性:

```
<PropertyGroup>
  <TieredCompilation>false</TieredCompilation>
</PropertyGroup>
```

TIP

如果在项目文件中更改这些设置，则可能需要执行干净的生成以反映新的设置（删除 `obj` 和 `bin` 目录并重新生成）。

有关在运行时配置编译的详细信息，请参阅[用于编译的运行时配置选项](#)。

ReadyToRun 映像

可以通过将应用程序集编译为 ReadyToRun (R2R) 格式来改进.NET Core 应用程序的启动时间。R2R 是一种预先 (AOT) 编译形式。

R2R 二进制文件通过减少应用程序加载时实时 (JIT) 编译器需要执行的工作量来改进启动性能。二进制文件包含与 JIT 将生成的内容类似的本机代码。但是，R2R 二进制文件更大，因为它们包含中间语言 (IL) 代码（某些情况下仍需要此代码）和相同代码的本机版本。仅当发布面向特定运行时环境 (RID)（如 Linux x64 或 Windows x64）的自包含应用时 R2R 才可用。

若要将项目编译为 ReadyToRun，请执行以下操作：

- 向项目中添加 `<PublishReadyToRun>` 设置：

```
<PropertyGroup>
  <PublishReadyToRun>true</PublishReadyToRun>
</PropertyGroup>
```

- 发布自包含应用。例如，此命令将创建适用于 Windows 64 位版本的自包含应用：

```
dotnet publish -c Release -r win-x64 --self-contained
```

跨平台/体系结构限制

ReadyToRun 编译器当前不支持跨目标。必须在给定的目标上编译。例如，如果想要 Windows x64 R2R 映像，需要在该环境中运行发布命令。

跨目标的例外情况：

- 可以使用 Windows x64 编译 Windows ARM32、ARM64 和 x86 映像。
- 可以使用 Windows x86 编译 Windows ARM32 映像。
- 可以使用 Linux x64 编译 Linux ARM32 和 ARM64 映像。

运行时/SDK

主要版本运行时前滚

.NET Core 3.0 引入了一项选择加入功能，该功能允许应用前滚到 .NET Core 最新的主要版本。此外，还添加了一项新设置来控制如何将前滚应用于应用。这可以通过以下方式配置：

- 项目文件属性：`RollForward`
- 运行时配置文件属性：`rollForward`
- 环境变量：`DOTNET_ROLL_FORWARD`
- 命令行参数：`--roll-forward`

必须指定以下值之一。如果省略该设置，则默认值为“Minor”。

- LatestPatch**
前滚到最高补丁版本。这会禁用次要版本前滚。
- Minor**
如果缺少所请求的次要版本，则前滚到最低的较高次要版本。如果存在所请求的次要版本，则使用 **LatestPatch**

策略。

- **Major**

如果缺少所请求的主要版本，则前滚到最低的较高主要版本和最低的次要版本。如果存在所请求的主要版本，则使用 **Minor** 策略。

- **LatestMinor**

即使存在所请求的次要版本，仍前滚到最高次要版本。适用于组件托管方案。

- **LatestMajor**

即使存在所请求的主要版本，仍前滚到最高主要版本和最高次要版本。适用于组件托管方案。

- **Disable**

不前滚。仅绑定到指定的版本。建议不要将此策略用于一般用途，因为它会禁用前滚到最新补丁的功能。该值仅建议用于测试。

除“Disable”设置外，所有设置都将使用可用的最高补丁版本。

生成会复制依赖项

`dotnet build` 命令现在将应用程序的 NuGet 依赖项从 NuGet 缓存复制到生成输出文件夹。此前，依赖项仅作为 `dotnet publish` 的一部分复制。

有些操作，比如链接和 razor 页面发布仍需要发布。

本地工具

.NET Core 3.0 引入了本地工具。本地工具类似于全局工具，但与磁盘上的特定位置相关联。本地工具在全局范围内不可用，并作为 NuGet 包进行分发。

WARNING

如果尝试使用过 .NET Core 3.0 预览版 1 中的本地工具，例如运行 `dotnet tool restore` 或 `dotnet tool install`，请删除本地工具缓存文件夹。否则，本地工具将无法在任何较新的版本上运行。此文件夹位于：

在 macOS、Linux 上：`rm -r $HOME/.dotnet/toolResolverCache`

在 Windows 上：`rmdir /s %USERPROFILE%\.dotnet\toolResolverCache`

本地工具依赖于当前目录中名为 `dotnet-tools.json` 的清单文件。此清单文件定义在该文件夹和以下文件夹中可用的工具。你可以随代码一起分发清单文件，以确保使用代码的任何人都可以还原和使用相同的工具。

对于全局工具和本地工具，需要一个兼容的运行时版本。目前，NuGet.org 上的许多工具都面向 .NET Core Runtime 2.1。若要在全局范围或本地安装这些工具，仍需要安装 [.NET Core 2.1 运行时](#)。

新 `global.json` 选项

`global.json` 文件包含新选项，当你尝试定义所使用的 .NET Core SDK 版本时，这些选项可提供更大的灵活性。新选项包括：

- `allowPrerelease`：指示在选择要使用的 SDK 版本时，SDK 解析程序是否应考虑预发布版本。
- `rollForward`：指示选择 SDK 版本时要使用的前滚策略，可作为特定 SDK 版本缺失时的回退，或者作为使用更高版本的指令。

有关这些更改的详细信息（包括默认值、支持的值和新的匹配规则），请参阅 [global.json 概述](#)。

垃圾回收堆大小减小

垃圾回收器的默认堆大小已减小，以使 .NET Core 使用更少的内存。此更改更符合具有现代处理器缓存大小的第 0 代分配预算。

垃圾回收大型页面支持

大型页面（也称为 Linux 上的巨型页面）是一项功能，其中操作系统能够建立大于本机页面大小（通常为 4K）的内存

区域，以提高请求这些大型页面的应用程序的性能。

现在可以使用 `GCLargePages` 设置将垃圾回收器配置为一项选择加入功能，以选择在 Windows 上分配大型页面。

Windows 桌面和 COM

.NET Core SDK Windows Installer

用于 Windows 的 MSI 安装程序已从 .NET Core 3.0 开始更改。SDK 安装程序现在将对 SDK 功能区段版本进行就地升级。功能区段在版本号的补丁部分中的百数组中定义。例如，3.0.101 和 3.0.201 是两个不同功能区段中的版本，而 3.0.101 和 3.0.199 则属于同一个功能区段。并且，当安装 .NET Core SDK 3.0.101 时，将从计算机中删除 .NET Core SDK 3.0.100（如果存在）。当 .NET Core SDK 3.0.200 安装在同一台计算机上时，不会删除 .NET Core SDK 3.0.101。

有关版本控制的详细信息，请参阅 [.NET Core 的版本控制方式概述](#)。

Windows 桌面

.NET Core 3.0 支持使用 Windows Presentation Foundation (WPF) 和 Windows 窗体的 Windows 桌面应用程序。这些框架还支持通过 [XAML 岛](#) 从 Windows UI XAML 库 (WinUI) 使用新式控件和 Fluent 样式。

Windows 桌面部件是 Windows .NET Core 3.0 SDK 的一部分。

可以使用以下 `dotnet` 命令创建新的 WPF 或 Windows 窗体应用：

```
dotnet new wpf  
dotnet new winforms
```

Visual Studio 2019 添加了适用于 .NET Core 3.0 Windows 窗体和 WPF 的“新建项目”模板。

有关如何移植现有 .NET Framework 应用程序的详细信息，请参阅 [移植 WPF 项目](#) 和 [移植 Windows 窗体项目](#)。

WinForms 高 DPI

.NET Core Windows 窗体应用程序可以使用 `Application.SetHighDpiMode(HighDpiMode)` 设置高 DPI 模式。

`SetHighDpiMode` 方法可设置相应的高 DPI 模式，除非该设置已通过其他方式（例如使用 `App.Manifest` 或在 `Application.Run` 前面使用 `P/Invoke`）进行设置。

由 `System.Windows.Forms.HighDpiMode` 枚举表示的可能的 `highDpiMode` 值包括：

- `DpiUnaware`
- `SystemAware`
- `PerMonitor`
- `PerMonitorV2`
- `DpiUnawareGdiScaled`

有关高 DPI 模式的详细信息，请参阅 [在 Windows 上开发高 DPI 桌面应用程序](#)。

创建 COM 组件

在 Windows 上，现在可以创建可调用 COM 的托管组件。在将 .NET Core 与 COM 加载项模型结合使用，以及使用 .NET Framework 提供奇偶校验时，此功能至关重要。

与将 `mscoree.dll` 用作 COM 服务器的 .NET Framework 不同，.NET Core 将在生成 COM 组件时向 `bin` 目录添加本机启动程序 `dll`。

有关如何创建 COM 组件并使用它的示例，请参阅 [COM 演示](#)。

Windows 本机互操作

Windows 提供丰富的本机 API，包括平面 C API、COM 和 WinRT 的形式。.NET Core 支持 `P/Invoke`，.NET Core 3.0

则增加了 CoCreate COM API 和 Activate WinRT API 的功能。有关代码示例, 请参阅 [Excel 演示](#)。

MSIX 部署

[MSIX](#) 是新的 Windows 应用程序包格式。可以使用它将 .NET Core 3.0 桌面应用程序部署到 Windows 10。

使用 Visual Studio 2019 中的 [Windows 应用打包项目](#), 可以创建包含[独立式](#) .NET Core 应用的 MSIX 包。

.NET Core 项目文件必须在 `<RuntimeIdentifiers>` 属性中指定支持的运行时:

```
<RuntimeIdentifiers>win-x86;win-x64</RuntimeIdentifiers>
```

Linux 改进

适用于 Linux 的 SerialPort

.NET Core 3.0 提供对 Linux 上 [System.IO.Ports.SerialPort](#) 的基本支持。

以前, .NET Core 只支持在 Windows 上使用 `SerialPort`。

有关对 Linux 上串行端口有限支持的详细信息, 请参阅 [GitHub 问题 #33146](#)。

Docker 和 cgroup 内存限制

在 Linux 上使用 Docker 运行 .NET Core 3.0 时, 可更好地应对 cgroup 内存限制。运行具有内存限制的 Docker 容器(例如使用 `docker run -m`)会更改 .NET Core 的行为方式。

- 默认垃圾回收器 (GC) 堆大小: 最大为 20 MB 或容器内存限制的 75%。
- 可以将显式大小设置为绝对数或 cgroup 限制的百分比。
- 每个 GC 堆的最小保留段大小为 16 MB。此大小可减少在计算机上创建的堆数量。

对 Raspberry Pi 的 GPIO 支持

已向 NuGet 发布了两个可用于 GPIO 编程的包:

- [System.Device.Gpio](#)
- [IoT.Device.Bindings](#)

GPIO 包包括用于 *GPIO*、*SPI*、*I2C* 和 *PWM* 设备的 API。IoT 绑定包括设备绑定。有关详细信息, 请参阅[设备 GitHub 存储库](#)。

ARM64 Linux 支持

.NET Core 3.0 增加了对 ARM64 for Linux 的支持。ARM64 的主要用例是当前的 IoT 场景。有关详细信息, 请参阅[.NET Core ARM64 状态](#)。

ARM64 上适用于 .NET Core 的 Docker 映像可用于 Alpine、Debian 和 Ubuntu。

NOTE

ARM64 尚未提供 Windows 支持。

安全性

Linux 上的 TLS 1.3 和 OpenSSL 1.1.1

.NET Core 现在可以在给定环境中使用 [OpenSSL 1.1.1 中的 TLS 1.3 支持](#)。使用 TLS 1.3:

- 通过减少客户端和服务器之间所需的往返次数, 提高了连接时间。
- 由于删除了各种过时和不安全的加密算法, 提高了安全性。

.NET Core 3.0 在 Linux 系统上使用 OpenSSL 1.1.1、OpenSSL 1.1.0 或 OpenSSL 1.0.2(如果可用)。当 OpenSSL 1.1.1 可用时, [System.Net.Security.SslStream](#) 和 [System.Net.Http.HttpClient](#) 类型都将使用 TLS 1.3(假定客户端和服务器都支持 TLS 1.3)。

IMPORTANT

Windows 和 macOS 尚不支持 TLS 1.3。当支持可用时, .NET Core 3.0 将在这些操作系统上支持 TLS 1.3。

下面的 C# 8.0 示例演示在 Ubuntu 18.10 上 .NET Core 3.0 如何连接到 <https://www.cloudflare.com>:

```
using System;
using System.Net.Security;
using System.Net.Sockets;
using System.Threading.Tasks;

namespace whats_new
{
    public static class TLS
    {
        public static async Task ConnectCloudFlare()
        {
            var targetHost = "www.cloudflare.com";

            using TcpClient tcpClient = new TcpClient();

            await tcpClient.ConnectAsync(targetHost, 443);

            using SslStream sslStream = new SslStream(tcpClient.GetStream());

            await sslStream.AuthenticateAsClientAsync(targetHost);
            await Console.Out.WriteLineAsync($"Connected to {targetHost} with {sslStream.SslProtocol}");
        }
    }
}
```

加密密码

.NET 3.0 增加了对 AES-GCM 和 AES-CCM 密码的支持(分别使用 [System.Security.Cryptography.AesGcm](#) 和 [System.Security.Cryptography.AesCcm](#) 实现)。这些算法都是[用于关联数据的认证加密\(AEAD\)算法](#)。

下面的代码演示了如何使用 [AesGcm](#) 密码加密和解密随机数据。

```

using System;
using System.Linq;
using System.Security.Cryptography;

namespace whats_new
{
    public static class Cipher
    {
        public static void Run()
        {
            // key should be: pre-known, derived, or transported via another channel, such as RSA encryption
            byte[] key = new byte[16];
            RandomNumberGenerator.Fill(key);

            byte[] nonce = new byte[12];
            RandomNumberGenerator.Fill(nonce);

            // normally this would be your data
            byte[] dataToEncrypt = new byte[1234];
            byte[] associatedData = new byte[333];
            RandomNumberGenerator.Fill(dataToEncrypt);
            RandomNumberGenerator.Fill(associatedData);

            // these will be filled during the encryption
            byte[] tag = new byte[16];
            byte[] ciphertext = new byte[dataToEncrypt.Length];

            using (AesGcm aesGcm = new AesGcm(key))
            {
                aesGcm.Encrypt(nonce, dataToEncrypt, ciphertext, tag, associatedData);
            }

            // tag, nonce, ciphertext, associatedData should be sent to the other part

            byte[] decryptedData = new byte[ciphertext.Length];

            using (AesGcm aesGcm = new AesGcm(key))
            {
                aesGcm.Decrypt(nonce, ciphertext, tag, decryptedData, associatedData);
            }

            // do something with the data
            // this should always print that data is the same
            Console.WriteLine($"AES-GCM: Decrypted data is {(dataToEncrypt.SequenceEqual(decryptedData) ? "the same as" : "different than")}{original data."});
        }
    }
}

```

加密密钥导入/导出

.NET Core 3.0 支持从标准格式导入和导出非对称公钥和私钥。你不需要使用 X.509 证书。

所有密钥类型(例如 *RSA*、*DSA*、*ECDsa* 和 *ECDiffieHellman*)都支持以下格式：

- 公钥
 - X.509 SubjectPublicKeyInfo
- 私钥
 - PKCS#8 PrivateKeyInfo
 - PKCS#8 EncryptedPrivateKeyInfo

RSA 密钥还支持：

- 公钥

- PKCS#1 RSA PublicKey

- 私钥

- PKCS#1 RSA Private Key

导出方法生成 DER 编码的二进制数据，导入方法也是如此。如果密钥以文本友好的 PEM 格式存储，调用方需要在调用导入方法之前对内容进行 base64 解码。

```
using System;
using System.Security.Cryptography;

namespace whats_new
{
    public static class RSATest
    {
        public static void Run(string keyFile)
        {
            using var rsa = RSA.Create();

            byte[] keyBytes = System.IO.File.ReadAllBytes(keyFile);
            rsa.ImportRSAPrivateKey(keyBytes, out int bytesRead);

            Console.WriteLine($"Read {bytesRead} bytes, {keyBytes.Length - bytesRead} extra byte(s) in file.");
            RSAParameters rsaParameters = rsa.ExportParameters(true);
            Console.WriteLine(BitConverter.ToString(rsaParameters.D));
        }
    }
}
```

可以使用 [System.Security.Cryptography.Pkcs.Pkcs8PrivateKeyInfo](#) 检查 PKCS#8 文件，使用 [System.Security.Cryptography.Pkcs.Pkcs12Info](#) 检查 PFX/PKCS#12 文件。可以使用 [System.Security.Cryptography.Pkcs.Pkcs12Builder](#) 操作 PFX/PKCS#12 文件。

.NET Core 3.0 API 改动

范围和索引

新 [System.Index](#) 类型可用于编制索引。可从 `int` 创建一个从开头开始计数的索引，也可使用前缀 `^` 运算符 (C#) 创建一个从末尾开始计数的索引：

```
Index i1 = 3; // number 3 from beginning
Index i2 = ^4; // number 4 from end
int[] a = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
Console.WriteLine($"{a[i1]}, {a[i2]}"); // "3, 6"
```

此外，还有 [System.Range](#) 类型，它包含两个 `Index` 值，一个用于开头一个用于结尾，可以使用 `x..y` 范围表达式 (C#) 进行编写。然后可以使用 `Range` 编制索引，以便生成一个切片：

```
var slice = a[i1..i2]; // { 3, 4, 5 }
```

有关详细信息，请参阅[范围和索引教程](#)。

异步流

`IAsyncEnumerable<T>` 类型是 `IEnumerable<T>` 的新异步版本。通过该语言，可通过 `IAsyncEnumerable<T>` 执行 `await foreach` 操作来使用其元素，并对其使用 `yield return` 操作来生成元素。

下面的示例演示如何生成和使用异步流。`foreach` 语句为异步语句，它本身使用 `yield return` 为调用方生成异步流。此模式（使用 `yield return`）是生成异步流的建议模型。

```
async IAsyncEnumerable<int> GetBigResultsAsync()
{
    await foreach (var result in GetResultsAsync())
    {
        if (result > 20) yield return result;
    }
}
```

除了能够 `await foreach`，还可以创建异步迭代器，例如，一个返回 `IAsyncEnumerable/IAsyncEnumerator` 的迭代器，可以在其中进行 `await` 和 `yield` 操作。对于需要处理的对象，可以使用各种 BCL 类型（如 `Stream` 和 `Timer`）实现的 `IAsyncDisposable`。

有关详细信息，请参阅[异步流教程](#)。

IEEE 浮点

正在更新浮点 API，以符合 [IEEE 754-2008 修订](#)。这些更改旨在公开所有必需操作并确保这些操作在行为上符合 IEEE 规范。有关浮点改进的详细信息，请参阅[.NET Core 3.0 中的浮点分析和格式化改进](#)博客文章。

分析和格式化修复包括：

- 正确分析并舍入任何输入长度。
- 正确分析并格式化负零。
- 通过执行不区分大小写的检查并允许在前面使用可选的 `+`（如果适用），正确分析 `Infinity` 和 `Nan`。

新的 `System.Math` API 包括：

- `BitIncrement(Double)` 和 `BitDecrement(Double)`

相当于 `nextUp` 和 `nextDown` IEEE 运算。它们将返回最小的浮点数，该数字大于或小于输入值（分别）。例如，
`Math.BitIncrement(0.0)` 将返回 `double.Epsilon`。

- `MaxMagnitude(Double, Double)` 和 `MinMagnitude(Double, Double)`

相当于 `maxNumMag` 和 `minNumMag` IEEE 运算，它们将（分别）返回大于或小于两个输入的量值的值。例如，
`Math.MaxMagnitude(2.0, -3.0)` 将返回 `-3.0`。

- `ILogB(Double)`

相当于返回整数值的 `logB` IEEE 运算，它将返回输入参数的整数对数（以 2 为底）。此方法实际上与
`floor(log2(x))` 相同，但完成后出现最小舍入错误。

- `ScaleB(Double, Int32)`

相当于采用整数值的 `scaleB` IEEE 运算，它实际返回 `x * pow(2, n)`，但完成后出现最小舍入错误。

- `Log2(Double)`

相当于返回（以 2 为底）对数的 `log2` IEEE 运算。它会最小化舍入错误。

- `FusedMultiplyAdd(Double, Double, Double)`

相当于执行乘法加法混合的 `fma` IEEE 运算。也就是说，它以单个运算的形式执行 `(x * y) + z`，从而最小化舍入错误。例如，`FusedMultiplyAdd(1e308, 2.0, -1e308)` 返回 `1e308`。常规 `(1e308 * 2.0) - 1e308` 返回
`double.PositiveInfinity`。

- `CopySign(Double, Double)`

相当于 `copySign` IEEE 运算，它返回 `x` 的值但带有符号 `y`。

.NET 平台相关内部函数

已添加 API，允许访问某些性能导向的 CPU 指令，例如 SIMD 或位操作指令集。这些指令有助于在某些情况下实现

显著的性能改进，例如高效地并行处理数据。

在适当的情况下，.NET 库已开始使用这些指令来改进性能。

有关详细信息，请参阅 [.NET Platform Dependent Intrinsic \(.NET 平台相关内部函数\)](#)。

改进的 .NET Core 版本 API

从 .NET Core 3.0 开始，.NET Core 提供的版本 API 现在可以返回你预期的信息。例如：

```
System.Console.WriteLine($"Environment.Version: {System.Environment.Version}");  
  
// Old result  
//   Environment.Version: 4.0.30319.42000  
//  
// New result  
//   Environment.Version: 3.0.0
```

```
System.Console.WriteLine($"RuntimeInformation.FrameworkDescription:  
{System.Runtime.InteropServices.RuntimeInformation.FrameworkDescription}");  
  
// Old result  
//   RuntimeInformation.FrameworkDescription: .NET Core 4.6.27415.71  
//  
// New result (notice the value includes any preview release information)  
//   RuntimeInformation.FrameworkDescription: .NET Core 3.0.0-preview4-27615-11
```

WARNING

重大变更。这在技术上是一个中断性变更，因为版本控制方案已发生变化。

内置的快速 JSON 支持

.NET 用户在很大程度上依赖于 [Newtonsoft.Json](#) 和其他常用的 JSON 库，它们仍是很好的选择。[Newtonsoft.Json](#) 使用 .NET 字符串作为其基本数据类型，它实际上是 UTF-16。

新的内置 JSON 支持具有高性能、低分配的特点，并且可用于 UTF-8 编码的 JSON 文本。有关 [System.Text.Json](#) 命名空间和类型的详细信息，请参阅以下文章：

- [.NET 中的 JSON 序列化 - 概述](#)
- [如何在 .NET 中对 JSON 数据进行序列化和反序列化。](#)
- [如何从 Newtonsoft.Json 迁移到 System.Text.Json](#)

HTTP/2 支持

[System.Net.Http.HttpClient](#) 类型支持 HTTP/2 协议。如果启用 HTTP/2，则将通过 TLS/ALPN 协商 HTTP 协议版本，并在服务器选择使用 HTTP/2 时使用。

默认协议将保留 HTTP/1.1，但可以在两种不同方法中启用 HTTP/2。首先，可以将 HTTP 请求消息设置为使用 HTTP/2：

```
var client = new HttpClient() { BaseAddress = new Uri("https://localhost:5001") };

// HTTP/1.1 request
using (var response = await client.GetAsync("/"))
    Console.WriteLine(response.Content);

// HTTP/2 request
using (var request = new HttpRequestMessage(HttpMethod.Get, "/") { Version = new Version(2, 0) })
using (var response = await client.SendAsync(request))
    Console.WriteLine(response.Content);
```

其次，可以更改 [HttpClient](#) 以默认使用 HTTP/2：

```
var client = new HttpClient()
{
    BaseAddress = new Uri("https://localhost:5001"),
    DefaultRequestVersion = new Version(2, 0)
};

// HTTP/2 is default
using (var response = await client.GetAsync("/"))
    Console.WriteLine(response.Content);
```

很多时候，在你开发应用程序时要使用未加密的连接。如果你知道目标终结点将使用 HTTP/2，你可以为 HTTP/2 打开未加密的连接。可以通过将 `DOTNET_SYSTEM_NET_HTTP_SOCKETSHANDLER_HTTP2UNENCRYPTEDSUPPORT` 环境变量设置为 1 或通过在应用上下文中启用它来将其打开：

```
AppContext.SetSwitch("System.Net.Http.SocketsHttpHandler.Http2UnencryptedSupport", true);
```

后续步骤

- [查看 .NET Core 2.2 和 3.0 之间的重大变更。](#)
- [查看用于 Windows 窗体应用的 .NET Core 3.0 中的中断性变更。](#)

.NET Core 2.2 的新增功能

2020/3/18 • [Edit Online](#)

.NET Core 2.2 包括在应用程序部署、运行时服务的事件处理、Azure SQL 数据库的身份验证、JIT 编译器性能，以及执行 `Main` 方法之前的代码注入方面的增强功能。

新部署模式

从 .NET Core 2.2 开始，可以部署依赖于框架的可执行文件，这是“.exe”文件而不是“.dll”文件。与依赖框架的部署在功能上类似，依赖框架的可执行文件 (FDE) 仍然依赖于存在的 .NET Core 的共享系统级版本来运行。应用程序只包含代码和任何第三方依赖项。与依赖框架的部署不同，FDE 特定于平台。

这种新的部署模式在构建可执行文件(而不是库)方面具有独特优势，这意味着你可以直接运行应用程序，而无需首先调用 `dotnet`。

核心

在运行时服务中处理事件

你可能经常希望监视应用程序的运行时服务(如 GC、JIT 和 ThreadPool)的使用情况，以了解它们如何影响应用程序。在 Windows 系统上，这通常通过监视当前进程的 ETW 事件来完成。虽然这仍然可以很好地工作，但是如果你在低特权环境中或者在 Linux 或 macOS 上运行，那么并不总是能够使用 ETW。

从 .NET Core 2.2 开始，现在可以使用 `System.Diagnostics.Tracing.EventListener` 类来使用 CoreCLR 事件。这些事件描述了诸如 GC、JIT、ThreadPool 和 interop 等运行时服务的行为。这些事件与作为 CoreCLR ETW 提供程序的一部分公开的事件相同。这允许应用程序使用这些事件或使用传输机制将它们发送到遥测聚合服务。可以在以下代码示例中看到如何订阅事件：

```
internal sealed class SimpleEventListener : EventListener
{
    // Called whenever an EventSource is created.
    protected override void OnEventSourceCreated(EventSource eventSource)
    {
        // Watch for the .NET runtime EventSource and enable all of its events.
        if (eventSource.Name.Equals("Microsoft-Windows-DotNETRuntime"))
        {
            EnableEvents(eventSource, EventLevel.Verbose, (EventKeywords)(-1));
        }
    }

    // Called whenever an event is written.
    protected override void OnEventWritten(EventWrittenEventArgs eventData)
    {
        // Write the contents of the event to the console.
        Console.WriteLine($"ThreadID = {eventData.OSThreadId} ID = {eventData.EventId} Name = {eventData.EventName}");
        for (int i = 0; i < eventData.Payload.Count; i++)
        {
            string payloadString = eventData.Payload[i]?.ToString() ?? string.Empty;
            Console.WriteLine($"\\tName = \"{eventData.PayloadNames[i]}\" Value = \"{payloadString}\"");
        }
        Console.WriteLine("\n");
    }
}
```

此外，.NET Core 2.2 向 `EventWrittenEventArgs` 类添加了以下两个属性来提供有关 ETW 事件的其他信息：

- [EventWrittenEventArgs.OSThreadId](#)
- [EventWrittenEventArgs.TimeStamp](#)

数据

使用 `SqlConnection.AccessToken` 属性对 Azure SQL 数据库进行 AAD 身份验证

从 .NET Core 2.2 开始，由 Azure Active Directory 颁发的访问令牌可用于对 Azure SQL 数据库进行身份验证。若要支持访问令牌，必须将 `AccessToken` 属性添加到 `SqlConnection` 类。若要利用 AAD 身份验证，请下载 `System.Data.SqlClient` NuGet 包的版本 4.6。若要使用功能，可以使用包含在 [Microsoft.IdentityModel.Clients.ActiveDirectory](#) NuGet 包中的[适用于 .NET 的 Active Directory 身份验证库](#)获取访问令牌值。

JIT 编译器改进

分层编译仍然是一项可选功能

在 .NET Core 2.1，JIT 编译器实现了一项新的编译器技术，即“分层编译”，作为可选功能。分层编译旨在提高性能。由 JIT 编译器执行的重要任务之一是优化代码执行。然而，对于很少使用的代码路径，相比执行未优化代码所花费的运行时，编译器可能需要更多的时间来优化代码。分层编译介绍了 JIT 编译中的两个阶段：

- 第一层，将尽可能快地生成代码。
- 第二层，将为那些频繁执行的方法生成优化代码。为了增强性能，第二层编译并行执行。

有关分层编译可能带来的性能改进的信息，请参阅[宣布发布 .NET Core 2.2 预览版 2](#)。

在 .NET Core 2.2 预览版 2 中，默认情况下已启用分层编译。但是，我们仍然没有准备好在默认情况下启用分层编译。因此在 .NET Core 2.2 中，分层编译仍是一项可选功能。有关选择加入分层编译的信息，请参阅[.NET Core 2.1 中的新增功能](#)中的[Jit 编译器改进](#)。

运行时

在执行 Main 方法之前注入代码

从 .NET Core 2.2 开始，可以使用启动挂钩注入代码，然后再运行应用程序的 Main 方法。启动挂钩使主机可以在部署应用程序之后自定义其行为，而不需要重新编译或更改应用程序。

我们希望托管提供商定义自定义配置和策略，包括可能会影响主入口点加载行为的设置，如 `System.Runtime.Loader.AssemblyLoadContext` 行为。挂钩可用于设置跟踪或遥测注入，以设置回调进行处理，或定义其他环境相关的行为。挂钩独立于入口点，因此不需要修改用户代码。

有关详细信息，请参阅[主机启动挂钩](#)。

请参阅

- [.NET Core 的新增功能](#)
- [ASP.NET Core 2.2 的新增功能](#)
- [EF Core 2.2 中的新增功能](#)

.NET Core 2.1 的新增功能

2020/3/19 • [Edit Online](#)

.NET Core 2.1 提供以下几个方面的增强功能和新功能：

- [工具](#)
- [前滚](#)
- [部署](#)
- [Windows 兼容包](#)
- [JIT 编译改进](#)
- [API 更改](#)

工具

.NET Core 2.1 SDK (v 2.1.300)，该工具与 .NET Core 2.1 一起提供，包括以下更改和增强功能：

生成性能改进

.NET Core 2.1 的主要关注点是改进生成时性能，特别是增量生成。这些性能改进适用于使用 `dotnet build` 的两个命令行生成和 Visual Studio 中的生成。一些个别的改进领域包括：

- 对于包资产解决方法，只解决由生成使用的资产而不是所有资产。
- 程序集引用缓存。
- 使用长时间运行的 SDK 生成服务器，这些是跨各个 `dotnet build` 调用的过程。每次 `dotnet build` 运行时不再需要 JIT 编译大量代码块。生成服务器进程可以使用以下命令自动终止：

```
dotnet buildserver shutdown
```

新的 CLI 命令

许多使用 `DotnetCliToolReference` 的仅在每个项目的基础上可用的工具现作为 .NET Core SDK 的一部分提供。这些工具包括：

- `dotnet watch` 提供文件系统观察程序，该程序在执行指定的命令集之前会首先等待文件更改。例如，下面的命令将自动重新生成当前项目，并在其中的文件发生更改时生成详细输出：

```
dotnet watch -- --verbose build
```

请注意 `--verbose` 选项前面的 `--` 选项。它分隔从传递给子 `dotnet` 进程的参数直接传递到 `dotnet watch` 命令的选项。如果没有该选项，`--verbose` 选项将适用于 `dotnet watch` 命令，而非 `dotnet build` 命令。

有关详细信息，请参阅[使用 dotnet watch 开发 ASP.NET Core 应用](#)。

- `dotnet dev-certs` 生成和管理在 ASP.NET Core 应用程序开发期间使用的证书。
- `dotnet user-secrets` 管理 ASP.NET Core 应用程序中用户机密库的机密。
- `dotnet sql-cache` 在 Microsoft SQL Server 数据库中创建表和索引以用于分布式缓存。
- `dotnet ef` 是用于管理 Entity Framework Core 应用程序中数据库、`DbContext` 对象和迁移的工具。有关详细信息，请参阅[EF Core .NET 命令行工具](#)。

全局工具

.NET Core 2.1 支持全局工具，即，可通过命令行在全局范围内使用的自定义工具。以前版本的 .NET Core 中的扩展性模型只能通过使用 `DotnetCliToolReference` 在每个项目的基础上提供自定义工具。

若要安装全局工具，请使用 `dotnet tool install` 命令。例如：

```
dotnet tool install -g dotnetsay
```

完成安装后，可以通过指定工具名称从命令行运行该工具。有关详细信息，请参阅 [.NET Core 工具概述](#)。

使用 `dotnet tool` 命令管理工具

在 .NET Core 2.1 SDK 中，所有工具操作都使用 `dotnet tool` 命令。可用选项如下：

- `dotnet tool install` 安装工具。
- `dotnet tool update` 卸载并重新安装工具，它将高效地对其进行更新。
- `dotnet tool list` 列出当前安装的工具。
- `dotnet tool uninstall` 卸载当前安装的工具。

前滚

从 .NET Core 2.0 开始，所有 .NET Core 应用程序都将自动前滚到系统上安装的最新次要版本。

从 .NET Core 2.0 开始，如果在其中构建应用程序的 .NET Core 版本在运行时不存在，应用程序将针对最新安装的次要版本的 .NET Core 自动运行。换而言之，如果应用程序在 .NET Core 2.0 中生成，而主机系统未安装 .NET Core 2.0 但安装了 .NET Core 2.1，则应用程序将通过 .NET Core 2.1 运行。

IMPORTANT

此前滚行为不适用于预览版本。默认情况下，它也不适用于主要版本，但可以通过以下设置进行更改。

可以通过在没有候选共享框架的情况下更改前滚设置来修改此行为。可用设置如下：

- `0` - 禁用次要版本前滚行为。使用此设置，为 .NET Core 2.0.0 构建的应用程序将前滚到 .NET Core 2.0.1，但不会前滚到 .NET Core 2.2.0 或 .NET Core 3.0.0。
- `1` - 启用次要版本前滚行为。这是设置的默认值。使用此设置，为 .NET Core 2.0.0 构建的应用程序将前滚到 .NET Core 2.0.1 或 .NET Core 2.2.0，具体取决于安装的版本，但它不会前滚到 .NET Core 3.0.0。
- `2` - 启用次要和主要版本前滚行为。即使考虑不同的主要版本，如果这样设置，为 .NET Core 2.0.0 构建的应用程序将前滚到 .NET Core 3.0.0。

可以通过以下三种方式之一修改此设置：

- 将 `DOTNET_ROLL_FORWARD_ON_NO_CANDIDATE_FX` 环境变量设置为所需的值。
- 使用所需的值将下列行添加到 `.runtimeconfig.json` 文件：

```
"rollForwardOnNoCandidateFx" : 0
```

- 使用 [.NET Core CLI](#) 时，请使用所需的值将下列选项添加到 .NET Core 命令，例如 `run`：

```
dotnet run --rollForwardOnNoCandidateFx=0
```

修补程序版本前滚与此设置无关，并且在应用任何潜在的次要或主要版本前滚之后完成。

部署

自包含应用程序服务

`dotnet publish` 现发布服务运行时版本的自包含应用程序。当你使用 .NET Core 2.1 SDK (v 2.1.300) 发布自包含应用程序时，你的应用程序将包括此 SDK 已知的最新服务运行时版本。升级到最新的 SDK 时，你将发布最新的 .NET Core 运行时版本。这适用于 .NET Core 1.0 运行时以及更高版本。

自包含发布依赖于 NuGet.org 上的运行时版本。计算机上不需要有服务运行时。

使用 .NET Core 2.0 SDK，自包含应用程序将通过 .NET Core 2.0.0 运行时发布，除非通过 `RuntimeFrameworkVersion` 属性指定不同版本。借助此新行为，你将不再需要设置此属性便可为自包含的应用程序选择更高版本的运行时。最简单的方法是始终通过 .NET Core 2.1 SDK (v 2.1.300) 发布。

有关详细信息，请参阅[独立部署运行时前滚](#)。

Windows 兼容包

当你将现有代码从 .NET Framework 转移到 .NET Core 时，可以使用 [Windows 兼容包](#)。除了 .NET Core 中提供的 API，它使你还能够访问额外的 20,000 多个 API。这些 API 包括 `System.Drawing` 命名空间中的类型、`EventLog` 类、WMI、性能计数器、Windows 服务以及 Windows 注册表类型和成员。

JIT 编译器改进

.NET Core 包含新的 JIT 编译器技术，称为“分层编译”（也称为“自适应优化”），可以显著提高性能。分层编译是一个可选设置。

由 JIT 编译器执行的重要任务之一是优化代码执行。然而，对于很少使用的代码路径，相比运行未优化代码所花费的运行时，编译器可能需要更多的时间来优化代码。分层编译介绍了 JIT 编译中的两个阶段：

- 第一层，将尽可能快地生成代码。
- 第二层，将为那些频繁执行的方法生成优化代码。为了增强性能，第二层编译并行执行。

可以通过这两种方法之一选择加入分层编译。

- 若要在所有使用 .NET Core 2.1 SDK 的项目中使用分层编译，请设置以下环境变量：

```
COMPlus_TieredCompilation="1"
```

- 若要在每个项目的基础上使用分层编译，将 `<TieredCompilation>` 属性添加到 MSBuild 项目文件的 `<PropertyGroup>` 部分，如以下示例所示：

```
<PropertyGroup>
    <!-- other property definitions -->

    <TieredCompilation>true</TieredCompilation>
</PropertyGroup>
```

API 更改

`Span<T>` 和 `Memory<T>`

.NET Core 2.1 包括一些新类型，使得在使用数组和其他类型内存方面要高效得多。新类型包括：

- [System.Span<T>](#) 和 [System.ReadOnlySpan<T>](#)。
- [System.Memory<T>](#) 和 [System.ReadOnlyMemory<T>](#)。

如果没有这些类型，那么在作为数组的一部分或内存缓冲区的一部分传递此类项时，必须在将数据的某些部分传递给方法之前复制该数据部分。这些类型提供了该数据的虚拟视图，无需额外的内存分配和复制操作。

下面的示例使用 [Span<T>](#) 和 [Memory<T>](#) 实例来提供一个数组 10 个元素的虚拟视图。

```
using System;

class Program
{
    static void Main()
    {
        int[] numbers = new int[100];
        for (int i = 0; i < 100; i++)
        {
            numbers[i] = i * 2;
        }

        var part = new Span<int>(numbers, start: 10, length: 10);
        foreach (var value in part)
            Console.Write($"{value} ");
    }
}

// The example displays the following output:
//      20  22  24  26  28  30  32  34  36  38
```

```
Module Program
Sub Main()
    Dim numbers As Integer() = New Integer(99) {}

    For i As Integer = 0 To 99
        numbers(i) = i * 2
    Next

    Dim part = New Memory(Of Integer)(numbers, start:=10, length:=10)

    For Each value In part.Span
        Console.Write($"{value} ")
    Next
End Sub
End Module

' The example displays the following output:
'      20  22  24  26  28  30  32  34  36  38
```

Brotli 压缩

.NET Core 2.1 添加了对 Brotli 压缩和解压缩的支持。Brotli 是在 [RFC 7932](#) 中定义的通用无损压缩算法，并且大多数 Web 浏览器和主 Web 服务器都提供支持。可以使用基于流的 [System.IO.Compression.BrotliStream](#) 类或基于范围的高性能 [System.IO.Compression.BrotliEncoder](#) 和 [System.IO.Compression.BrotliDecoder](#) 类。下面的示例用 [BrotliStream](#) 类演示压缩：

```

public static Stream DecompressWithBrotli(Stream toDecompress)
{
    MemoryStream decompressedStream = new MemoryStream();
    using (BrotliStream decompressionStream = new BrotliStream(toDecompress, CompressionMode.Decompress))
    {
        decompressionStream.CopyTo(decompressedStream);
    }
    decompressedStream.Position = 0;
    return decompressedStream;
}

```

```

Public Function DecompressWithBrotli(toDecompress As Stream) As Stream
    Dim decompressedStream As New MemoryStream()
    Using decompressionStream As New BrotliStream(toDecompress, CompressionMode.Decompress)
        decompressionStream.CopyTo(decompressedStream)
    End Using
    decompressedStream.Position = 0
    Return decompressedStream
End Function

```

`BrotliStream` 行为等同于 `DeflateStream` 和 `GZipStream`, 这样就可以轻松地将调用这些 API 的代码转换为 `BrotliStream`。

新加密 API 和加密改进

.NET Core 2.1 包括加密 API 的许多增强功能：

- `System.Security.Cryptography.Pkcs.SignedCms` 在 `System.Security.Cryptography.Pkcs` 包中提供。其实现与 .NET Framework 中的 `SignedCms` 类相同。
- `X509Certificate.GetCertHash` 和 `X509Certificate.GetCertHashString` 方法的新重载接受一个哈希算法标识符，使调用方能够使用除 SHA-1 以外的算法获得证书指纹值。
- 新的基于 `Span<T>` 的加密 API 可用于哈希、HMAC、加密随机数生成、非对称签名生成、非对称签名处理和 RSA 加密。
- 通过使用基于 `Span<T>` 的实现，`System.Security.Cryptography.Rfc2898DeriveBytes` 的性能提高了大约 15%。
- 新 `System.Security.Cryptography.CryptographicOperations` 类包括两个新方法：
 - `FixedTimeEquals` 需要固定时间来返回任意两个长度相同的输入，这使得它适用于加密验证，从而避免提供计时旁道信息。
 - `ZeroMemory` 是不能进行优化的内存清理例程。
- 静态 `RandomNumberGenerator.Fill` 方法用随机值填充 `Span<T>`。
- `System.Security.Cryptography.Pkcs.EnvelopedCms` 现在在 Linux 和 macOS 上受支持。
- `System.Security.Cryptography.ECDiffieHellman` 类系列现提供椭圆曲线 Diffie-Hellman (ECDH)。外围应用与 .NET Framework 中相同。
- `RSA.Create` 返回的实例可以使用 SHA-2 摘要对 OAEP 进行加密或解密，并使用 RSA-PSS 生成或验证签名。

套接字改进

.NET Core 包括一个新类型 `System.Net.Http.SocketsHttpHandler` 和重写的 `System.Net.Http.HttpMessageHandler`，两者构成了更高级别网络 API 的基础。例如，`System.Net.Http.SocketsHttpHandler` 是 `HttpClient` 实现的基础。在以前版本的 .NET Core 中，更高级别的 API 基于本机网络实现。

.NET Core 2.1 中引入的套接字实现具有很多优点：

- 对照以前的实现，可以看到显著的性能改进。
- 消除平台依赖项，从而简化部署和维护。
- 在所有 .NET Core 平台之间保持行为一致。

[SocketsHttpHandler](#) 是 .NET Core 2.1 中的默认实现。但是，你可以通过调用 [AppContext.SetSwitch](#) 方法配置应用程序以使用较旧的 [HttpClientHandler](#) 类：

```
AppContext.SetSwitch("System.Net.Http.UseSocketsHttpHandler", false);
```

```
AppContext.SetSwitch("System.Net.Http.UseSocketsHttpHandler", False)
```

还可以使用环境变量选择退出使用基于 [SocketsHttpHandler](#) 的套接字实现。为此，需要将

`DOTNET_SYSTEM_NET_HTTP_USESOCKETSHTTPHANDLER` 设置为 `false` 或 0。

在 Windows 上，还可以选择使用 [System.Net.Http.WinHttpHandler](#)（后者依赖于本机实现），或者通过将类实例传递到 [HttpClient](#) 构造函数来使用 [SocketsHttpHandler](#) 类。

在 Linux 和 macOS 上，可以在每个进程的基础上仅配置 [HttpClient](#)。在 Linux 上，如果想要使用旧的 [HttpClient](#) 实现，则需要部署 [libcurl](#)。（它随 .NET Core 2.0 一起安装。）

重大更改

有关中断性变更的信息，请参阅[从版本 2.0 迁移到 2.1 的中断性变更](#)。

请参阅

- [.NET Core 的新增功能](#)
- [EF Core 2.1 中的新增功能](#)
- [ASP.NET Core 2.1 的新增功能](#)

.NET Core 2.0 的新增功能

2020/3/18 • [Edit Online](#)

.NET Core 2.0 提供以下几个方面的增强功能和新功能：

- [工具](#)
- [语言支持](#)
- [平台改进](#)
- [API 更改](#)
- [Visual Studio 集成](#)
- [文档改进](#)

工具

dotnet restore 隐式运行

在旧版 .NET Core 中，在使用 `dotnet new` 命令新建项目后，以及每当向项目添加新的依赖项时，都必须立即运行 `dotnet restore` 命令，以便下载依赖项。

NOTE

从 .NET Core 2.0 SDK 开始，无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build` 和 `dotnet run`。在执行显式还原有意义的某些情况下，例如 [Azure DevOps Services 中的持续集成生成中](#)，或在需要显式控制还原发生时间的生成系统中，它仍然是有效的命令。

还可以将 `--no-restore` 开关传递到 `new`、`run`、`build`、`publish`、`pack` 和 `test` 命令，从而禁用自动调用 `dotnet restore`。

重定目标到 .NET Core 2.0

如果已安装 .NET Core 2.0 SDK，那么定目标到 .NET Core 1.x 的项目可以重定目标到 .NET Core 2.0。

若要重定目标到 .NET Core 2.0，请将 `<TargetFramework>` 元素（或 `<TargetFrameworks>` 元素，如果项目文件中有多个目标的话）值从 1.x 更改为 2.0，从而编辑项目文件：

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.0</TargetFramework>
</PropertyGroup>
```

还可以用同样的方式，将 .NET Standard 库重定目标到 .NET Standard 2.0：

```
<PropertyGroup>
  <TargetFramework>netstandard2.0</TargetFramework>
</PropertyGroup>
```

若要详细了解如何将项目迁移到 .NET Core 2.0，请参阅[从 ASP.NET Core 1.x 迁移到 ASP.NET Core 2.0](#)。

语言支持

除了支持 C# 和 F# 外，.NET Core 2.0 还支持 Visual Basic。

Visual Basic

在版本 2.0 发布后, .NET Core 现在支持 Visual Basic 2017。可使用 Visual Basic 创建以下类型项目：

- .NET Core 控制台应用程序
- .NET Core 类库
- .NET Standard 类库
- .NET Core 单元测试项目
- .NET Core xUnit 测试项目

例如, 若要创建 Visual Basic“Hello World”应用程序, 请通过命令行按照以下步骤操作:

1. 打开控制台窗口, 创建项目目录, 并将它设为当前目录。

2. 输入命令 `dotnet new console -lang vb`。

此命令创建文件扩展名为 `.vbproj` 的项目文件, 以及名为 `Program.vb` 的 Visual Basic 源代码文件。此文件包含用于将字符串“Hello World!”写入控制台窗口的源代码。

3. 输入命令 `dotnet run`。[.NET Core CLI](#) 自动编译并执行应用程序, 在控制台窗口中 显示文本字符串“Hello World!”。

支持 C# 7.1

.NET Core 2.0 支持 C# 7.1, 其中新增了大量功能, 具体包括:

- 可以使用 `async` 关键字标记 `Main` 方法(应用程序入口点)。
- 推断的元组名称。
- 默认表达式。

平台改进

.NET Core 2.0 包括许多功能, 可便于用户更轻松地在支持的操作系统上安装并使用 .NET Core。

.NET Core for Linux 是一个实现代码

.NET Core 2.0 提供一个 Linux 实现代码, 适用于多个 Linux 发行版本。.NET Core 1.x 要求下载发行版本专属的 Linux 实现代码。

还可以开发定目标到 Linux 一个操作系统的应用程序。.NET Core 1.x 要求分别定目标到每个 Linux 发行版本。

支持 Apple 加密库

macOS 上的 .NET Core 1.x 要求使用 OpenSSL 工具包的加密库。.NET Core 2.0 使用 Apple 加密库, 不要求使用 OpenSSL, 因此不再需要安装它。

API 更改和库支持

支持 .NET Standard 2.0

.NET Standard 定义了一组版本化 API, 这些 API 必须可用于符合相应 Standard 版本要求的 .NET 实现代码。.NET Standard 面向库开发者。旨在保证功能对每个 .NET 实现代码中定目标到 .NET Standard 版本的库可用。.NET Core 1.x 支持 .NET Standard 版本 1.6;.NET Core 2.0 支持最新版 .NET Core 2.0。有关详细信息, 请参阅 [.NET Standard](#)。

.NET Standard 2.0 比 .NET Standard 1.6 多包含 20,000 多个 API。此扩展的外围应用的大部分来自于, 将 .NET Framework 和 Xamarin 的通用 API 合并到 .NET Standard。

.NET Standard 2.0 类库还可以引用 .NET Framework 类库, 但前提是它们调用 .NET Standard 2.0 中的 API。不需要重新编译 .NET Framework 库。

有关自上一版本 (.NET Standard 1.6) 起添加到 .NET Standard 的 API 列表, 请参阅 [.NET Standard 2.0 与 1.6](#)。

扩展的外围应用

与 .NET Core 1.1 相比, .NET Core 2.0 中的可用 API 总数增加了一倍以上。

借助 [Windows 兼容包](#), 从 .NET Framework 移植也简单了很多。

支持 .NET Framework 库

.NET Core 代码可以引用现有的 .NET Framework 库, 包括现有的 NuGet 包。请注意, 库必须使用 .NET Standard 中的 API。

Visual Studio 集成

Visual Studio 2017 版本 15.3 和(在某些情况下)Visual Studio for Mac 提供了大量面向 .NET Core 开发者的重大增强功能。

重定目标 .NET Core 应用程序和 .NET Standard 库

如果已安装 .NET Core 2.0 SDK, 可以将 .NET Core 1.x 项目重定目标到 .NET Core 2.0, 并将 .NET Standard 1.x 库重定目标到 .NET Standard 2.0。

若要在 Visual Studio 中重定目标项目, 可以打开项目属性对话框的“应用程序”选项卡, 再将“目标框架”值更改为“.NET Core 2.0”或“.NET Standard 2.0”。还可以通过右键单击项目并选择“编辑 *.csproj 文件”选项进行更改。有关详细信息, 请参阅本主题前面的[工具](#)部分。

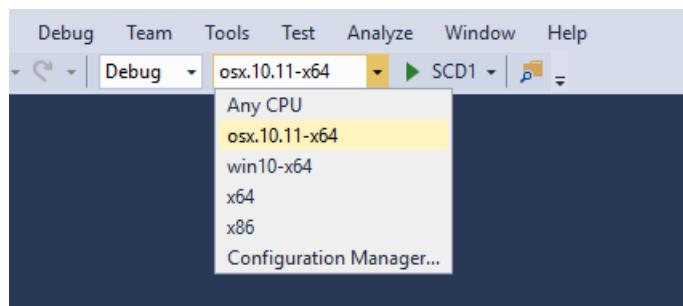
.NET Core 的 Live Unit Testing 支持

修改代码时, Live Unit Testing 在后台自动运行任何受影响的单元测试, 并在 Visual Studio 环境中实时显示结果和代码覆盖率。.NET Core 2.0 现在支持 Live Unit Testing。以前, Live Unit Testing 仅适用于 .NET Framework 应用程序。

有关详细信息, 请参阅[使用 Visual Studio 执行 Live Unit Testing](#) 和 [Live Unit Testing 常见问题解答](#)。

更好地支持多个目标框架

若要为多个目标框架生成项目, 现在可以从顶级菜单中选择目标平台。在下图中, 名为 SCD1 的项目将目标定为 64 位 macOS X 10.11 (`osx.10.11-x64`) 和 64 位 Windows 10/Windows Server 2016 (`win10-x64`)。可以先选择目标框架, 再选择项目按钮(在此示例中是为了要运行调试版本)。



对 .NET Core SDK 的并行支持

现在可以安装 .NET Core SDK, 与 Visual Studio 互不影响。这样, 单版本 Visual Studio 可以生成定目标到不同 .NET Core 版本的项目。以前, Visual Studio 和 .NET Core SDK 紧密结合在一起;特定版本的 SDK 附带了特定版本的 Visual Studio。

文档改进

.NET 应用程序体系结构

通过 [.NET 应用程序体系结构](#), 可以查看一系列电子图书, 其中提供了有关如何使用 .NET 生成内容的指导、最佳做法和示例应用程序:

- [微服务和 Docker 容器](#)
- [使用 ASP.NET 的 Web 应用程序](#)

- [使用 Xamarin 的移动应用](#)
- [使用 Azure 部署到云的应用程序](#)

请参阅

- [ASP.NET Core 2.0 的新增功能](#)

会影响兼容性的变更

2020/3/18 • [Edit Online](#)

在 .NET 的整个历史记录中，它都尝试在版本之间以及 .NET 各个风格之间保持高级别的兼容性。.NET Core 将继续坚守这个准则。尽管可以将 .NET Core 视为独立于 .NET Framework 的新技术，但下面的两个因素使 .NET Core 无法脱离 .NET Framework：

- 有许多最初开发过或在继续开发 .NET Framework 应用程序的开发人员。他们希望各个 .NET 实现中的行为保持一致。
- .NET Standard 库项目允许开发人员创建面向 .NET Core 和 .NET Framework 共享的通用 API 的库。开发人员希望用于 .NET Core 应用程序的库与用于 .NET Framework 应用程序的同一个库的行为相同。

在希望保持各个 .NET 实现之间的兼容性的同时，开发人员还希望在各个 .NET Core 版本之间保持高级别的兼容性。具体而言，为 .NET Core 早期版本编写的代码应在较高版本的 .NET Core 上无缝运行。实际上，许多开发人员都希望新发布的 .NET Core 版本中的新 API 也应该与引入这些 API 的预发布版本兼容。

本文概述了兼容性变更（或中断性变更）的类别，以及 .NET 团队如何评估各个类别中的变更。如果开发人员需打开 [dotnet/runtime](#) GitHub 存储库中要求修改现有 API 的行为的拉取请求，则了解 .NET 团队如何处理可能的中断性变更对他们来说尤其有用。

NOTE

若要查看兼容性类别的定义，如二进制兼容性和向后兼容性，请参阅[中断性变更类别](#)。

以下各个部分说明了 .NET Core API 的变更类别，以及它们对应用程序兼容性的影响。✓ 表示允许更改；✗ 表示不允许更改；? 表示需要评判之前行为的可预测性、显著性和一致性。

NOTE

除了将这些准则用作 .NET Core 库变更评估指南以外，库开发人员还可以使用它们评估他们自己的面向多个 .NET 实现和版本的库更改。

公共协定修改

此类别的变更会修改类型的公共外围应用。禁止此类别中的多数变更，因为它们违反了向后兼容性（使用早期 API 版本生成的应用程序的功能：无需在较高版本上重新编译即可运行）。

类型

- ✓ 允许：基类型已实现接口时，从类型中删除接口实现
- ? 需要评判：向类型添加新的接口实现

此为可接受的变更，因为它不会对现有客户端产生不良影响。对此类型的任何变更必须在此处定义的可接受变更的边界内工作，新实现才能继续成为可接受的实现。添加直接影响设计器或序列化程序功能（生成无法供低级使用的代码或数据的功能）的接口时，需要格外注意。例如 [ISerializable](#) 接口。

- ? 需要评判：引入新的基类

若类型未引入任何新的抽象成员且未更改现有类型的语义或行为，可以将它引入到两个现有类型之间的层次结构。例如，在 .NET Framework 2.0 中，[DbConnection](#) 类成为之前直接派生自 [Component](#) 的 [SqlConnection](#) 的新基类。

- ✓ 允许: 将某个程序集中的类型移动到另一个程序集中

旧程序集必须标有指向新程序集的 [TypeForwardedToAttribute](#)。

- ✓ 允许: 将 `struct` 类型更改为 `readonly struct` 类型

不允许将 `readonly struct` 类型更改为 `struct` 类型。

- ✓ 允许: 没有可访问的(`public` 或 `protected`)构造函数时, 向类型添加 `sealed` 或 `abstract` 关键字

- ✓ 允许: 扩展类型的可见性

- ✗ 不允许: 更改类型的命名空间或名称

- ✗ 不允许: 重命名或删除公共类型

这将中断使用重命名的或删除的类型的所有代码。

- ✗ 不允许: 更改枚举的基础类型

此为编译时和行为中断性变更, 此外它还是可能会导致属性参数不可分析的二进制中断性更改。

- ✗ 不允许: 密封之前未密封的类型

- ✗ 不允许: 向接口的一组基类型添加接口

若接口实现它未曾实现过的接口, 将中断实现此接口的原始版本的所有类型。

- ? 需要评判: 从一组基类删除某个类, 或从一组实现的接口删除某个接口

接口删除规则有一个例外情况: 可以添加派生自删除的接口的接口实现。例如, 如果类型或接口现在实现将实现 `IDisposable` 的 `IComponent`, 则可以删除 `IDisposable`。

- ✗ 不允许: 将 `readonly struct` 类型更改为 `struct` 类型

但允许将 `struct` 类型更改为 `readonly struct` 类型。

- ✗ 不允许: 将 `struct` 类型更改为 `ref struct` 类型, 或将后者改为前者

- ✗ 不允许: 缩小类型的可见性

但允许增大类型的可见性。

成员

- ✓ 允许: 扩展非 `virtual` 成员的可见性

- ✓ 允许: 向不包含任何可访问的(`public` 或 `protected`)构造函数或为 `sealed` 类型的公共类型添加抽象成员

但不允许向包含可访问的(公共或受保护的)构造函数且非 `sealed` 类型的类型添加抽象成员。

- ✓ 允许: 类型不包含任何可访问的(`public` 或 `protected`)构造函数或类型为 `sealed` 类型时, 限制 `protected` 成员的可见性

- ✓ 允许: 将成员移动到层次结构中高于删除的成员所在的类型的类

- ✓ 允许: 添加或删除重写

引入重写可能会导致先前的使用者在调用 `base` 时跳过重写。

- ✓ 允许: 向类添加构造函数及无参数构造函数(若该类过去没有任何构造函数)

但是, 不允许在未对过去不包含任何构造函数的类添加无参数构造函数的情况下向其添加构造函数。

- ✓ 允许: 将成员从 `abstract` 更改为 `virtual`

- ✓ 允许: 从 `ref readonly` 更改为 `ref` 返回值(虚拟方法或接口除外)
- ✓ 允许: 若字段的静态类型为非可变的值类型, 从字段删除 `readonly`
- ✓ 允许: 调用未曾定义的新事件
- ? 需要评判: 向类型添加新实例字段

此变更影响序列化。

- ✗ 不允许: 重命名或删除公共成员或参数

这将中断使用重命名的或删除的成员或参数的所有代码。

这包括删除或重命名属性中的 Getter 或资源库, 以及重命名或删除枚举成员。

- ✗ 不允许: 向接口添加成员
- ✗ 不允许: 更改公共常量或枚举成员的值
- ✗ 不允许: 更改属性类型、字段、参数或返回值
- ✗ 不允许: 添加、删除、或更改参数的顺序
- ✗ 不允许: 向参数添加或从中删除 `in`、`out` 或 `ref` 关键字
- ✗ 不允许: 重命名参数(包括更改其大小写)

鉴于以下两个原因将此视为中断性变更:

- 它将中断后期绑定方案, 如 Visual Basic 中的后期绑定功能和 C# 的 `dynamic`。
- 如果开发人员使用 [命名参数](#), 它将中断源兼容性。

- ✗ 不允许: 从 `ref` 返回值更改为 `ref readonly` 返回值
- ✗ 不允许: 在虚拟方法或接口上从 `ref readonly` 更改为 `ref` 返回值
- ✗ 不允许: 向成员添加或从中删除 `abstract`
- ✗ 不允许: 从成员删除 `virtual` 关键字

通常这不属于中断性变更, 因为 C# 编译器通常会发出 `callvirt` 中间语言 (IL) 指令来调用非虚拟方法 (`callvirt` 执行 null 检查, 而常规调用不会执行此检查), 鉴于下列原因此行为非恒定:

- C# 并非 .NET 面向的唯一语言。
- 目标方法为非虚拟且可能非 null 时(如通过 `? .null` 传播运算符访问的方法), C# 编译器逐渐尝将 `callvirt` 优化为常规调用。

使方法成为虚拟方法意味着使用者代码通常最终要以非虚拟方式调用它。

- ✗ 不允许: 向成员添加 `virtual` 关键字
- ✗ 不允许: 使 `virtual` 成员成为 `abstract` 成员

[抽象成员](#) 提供可以由派生类重写的方法实现。抽象成员不提供任何实现, 且必须重写。

- ✗ 不允许: 向包含可访问的(`public` 或 `protected`)构造函数且非 `sealed` 类型的公共类型添加抽象成员
- ✗ 不允许: 向成员添加或从中删除 `static` 关键字
- ✗ 不允许: 添加排除现有重载并定义其他行为的重载

这将中断已绑定先前重载的现有客户端。例如, 若类包含单个接受 `UInt32` 的方法的版本, 传递 `Int32` 值时,

现有使用者将成功地绑定该重载。但是，如果添加接受 `Int32` 的重载，重新编译或使用晚期绑定时，编译器现在将绑定新的重载。若生成不同的行为，则它属于中断性变更。

- ✗ 不允许：只向过去不包含任何构造函数的类添加构造函数而不添加无参数构造函数
- ✗ 不允许：向字段添加 `readonly`
- ✗ 不允许：降低成员的可见性

这包括在存在可访问的(`public` 或 `protected`)构造函数且类型非 `sealed` 的情况下降低 `protected` 成员的可见性。若不属于上述情况，则允许降低受保护的成员的可见性。

允许增大成员的可见性。

- ✗ 不允许：更改成员的类型

不可修改方法的返回值、属性类型或字段。例如，不可将返回 `Object` 的方法的签名更改为返回 `String`，反之亦然。

- ✗ 不允许：向先前没有任何状态的结构添加字段

有明确的分配规则规定，只要变量类型为无状态结构，即允许使用未初始化的变量。若结构成为有状态结构，代码可能最终成为未初始化的数据。这可能既是源中断性变更，又是二进制中断性变更。

- ✗ 不允许：触发先前从未触发过的现有事件

行为变更

程序集

- ✓ 允许：使程序集成为可移植的程序集并且仍支持同样的平台
- ✗ 不允许：更改程序集的名称
- ✗ 不允许：更改程序集的公钥

属性、字段、参数和返回值

- ✓ 允许：将属性、字段、返回值或 `out` 参数的值更改为派生程度更大的类型

例如，返回 `Object` 的类型的方法可能返回 `String` 实例。(但是不可更改方法签名。)

- ✓ 允许：在成员为非 `virtual` 成员时，扩大属性或参数的可接受值的范围

可以扩展可传递到方法或由成员返回的值范围，但不可扩展参数或成员类型。例如，传递到方法的值可以从 0-124 扩展到 0-255，但参数类型不可从 `Byte` 更改为 `Int32`。

- ✗ 不允许：在成员为 `virtual` 成员时，扩大属性或参数的可接受值的范围

此变更将中断已重写的现有成员，面向扩展的值范围时它们将无法正常运行。

- ✗ 不允许：缩小属性或参数的可接受值的范围
- ✗ 不允许：扩大属性的返回值范围、字段、返回值或 `out` 参数
- ✗ 不允许：更改属性的返回值、字段、方法返回值或 `out` 参数
- ✗ 不允许：更改属性、字段或参数的默认值
- ✗ 不允许：更改数值返回值的精度
- ⚡ 需要评判：关于输入分析和新异常引发的变更(尽管本文档未指定分析行为)

异常

- ✓ 允许: 引发派生程度高于现有异常的异常

由于新异常是现有异常的子类，先前的异常处理代码将继续处理异常。例如，在 .NET Framework 4 中，找不到区域性时，区域性生成和检索方法开始引发 [CultureNotFoundException](#) 而不引发 [ArgumentException](#)。由于 [CultureNotFoundException](#) 派生自 [ArgumentException](#)，因此这是可接受的变更。

- ✓ 允许: 引发比 [NotSupportedException](#)、[NotImplementedException](#)、[NullReferenceException](#) 更加具体的异常

- ✓ 允许: 引发被视为无法恢复的异常

不应捕获无法恢复的异常，而应该由高级别的全部捕获处理程序处理它们。因此，用户不应该拥有捕获这些显式异常的代码。无法恢复的异常包括：

- [AccessViolationException](#)
- [ExecutionEngineException](#)
- [SEHException](#)
- [StackOverflowException](#)

- ✓ 允许: 在新的代码路径中引发新的异常

异常必须仅适用于使用新参数值或状态执行并且无法由面向先前版本的现有代码执行的新代码路径。

- ✓ 允许: 删除异常，以启用更可靠的行为或新方案

例如，可以以其他方式将先前仅处理正值并引发 [ArgumentOutOfRangeException](#) 的 `Divide` 方法更改为同时支持正值和负值并且不引发异常。

- ✓ 允许: 更改错误消息的文本

开发人员不应依赖也会基于用户区域性更改的错误消息文本。

- ✗ 不允许: 在上文未列出的任何其他的情况下引发异常
- ✗ 不允许: 在上文未列出的任何其他的情况下删除异常

特性

- ✓ 允许: 更改不可观测的属性的值
- ✗ 不允许: 更改可观测的属性的值
- ? 需要评判: 删除属性

多数情况下，删除属性（如 [NonSerializedAttribute](#)）为中断性变更。

平台支持

- ✓ 允许: 在平台上支持先前不支持的操作
- ✗ 不允许: 对于平台先前支持的操作，不再支持或者现在需要特定服务包

内部实现变更

- ? 需要评判: 更改内部类型的外围应用

尽管此类更改将中断私有反射，但通常允许这些变更。如果常用的第三方库或大量开发人员依赖内部 API，在这些情况下，可能不允许此类变更。

- ? 需要评判: 更改成员的内部实现

尽管此类更改将中断私有反射，但通常允许这些变更。如果客户代码频繁依赖私有反射，或者变更引入意外

的负面影响，在这些情况下，可能不允许这些变更。

- ✓ 允许: 提高操作的性能

修改操作性能的功能必不可少，但此类变更可能会中断依赖操作的当前速度的代码。这一点尤其适用于对于依赖异步操作计时的代码。性能更改应不影响所说的 API 的其他行为；否则，变更将属于中断性变更。

- ✓ 允许: 间接更改操作性能(通常产生的是负面影响)

若出于某些其他的原因未将所说的变更归类为中断性变更，这是可以接受的。通常需要执行可能包含额外操作或添加新功能的操作。这几乎都会影响性能，但对于使所说的 API 按预期方式运行而言它可能必不可少。

- ✗ 不允许: 将同步 API 更改为异步(反之亦然)

代码更改

- ✓ 允许: 向参数添加 `params`

- ✗ 不允许: 将 `struct` 更改为 `class`, 或将后者改为前者

- ✗ 不允许: 向代码块添加 `checked` 关键字

此变更可能导致先前执行的代码引发 `OverflowException`，此为不可接受的变更。

- ✗ 不允许: 从参数删除 `params`

- ✗ 不允许: 更改事件的触发顺序

开发人员可以合理地期望事件按相同的顺序触发，开发人员代码频繁依赖事件的触发顺序。

- ✗ 不允许: 删除给定操作上的事件引发

- ✗ 不允许: 更改给定事件的调用次数

- ✗ 不允许: 向枚举类型添加 `FlagsAttribute`

通过探讨这些教程来学习 .NET Core 和 .NET Core SDK 工具

2020/3/18 • [Edit Online](#)

以下教程可用于了解 .NET Core。

使用 Visual Studio 创建应用程序

- [创建 Hello World 控制台应用程序](#)
- [调试 Hello World 应用程序](#)
- [发布 Hello World 应用程序](#)
- [生成类库](#)
- [测试类库](#)
- [使用类库](#)
- [Azure Cosmos DB:SQL API 和 .NET Core 入门](#)

使用 Visual Studio Code 生成应用程序

- [C# 和 Visual Studio Code 入门](#)
- [macOS 上的 .NET Core 入门](#)

使用 Visual Studio for Mac 生成应用程序

- [借助 Visual Studio for Mac 在 macOS 上开始使用 .NET Core](#)
- [使用 Visual Studio for Mac 在 macOS 上构建完整的 .NET Core 解决方案](#)

使用 .NET Core CLI 生成应用程序

- [使用 .NET Core CLI 在 Windows/Linux/macOS 上开始使用 .NET Core](#)
- [使用 .NET Core CLI 组织和测试项目](#)
- [F# 入门](#)

其他

- [在 .NET Core 中使用 dotnet 测试的单元测试](#)
- [使用 MSTest 和 .NET Core 执行单元测试](#)
- [使用跨平台工具开发库](#)
- [从本机代码承载 .NET Core](#)
- [创建 CLI 的模板](#)
- [创建和使用适用于 CLI 的工具](#)

有关开发 ASP.NET Core Web 应用程序的教程, 请参阅 [ASP.NET Core 文档](#)。

教程：在 Visual Studio 2019 中创建第一个 .NET Core 控制台应用程序

2020/3/18 • [Edit Online](#)

本文将逐步介绍如何在 Visual Studio 2019 中创建和运行 Hello World .NET Core 控制台应用程序。通常使用 Hello World 应用程序向初学者介绍新的编程语言。此程序只在屏幕上显示短语“Hello World!”。

先决条件

- 安装了具有“.NET Core 跨平台开发”工作负载的 [Visual Studio 2019 版本 16.4 或更高版本](#)。选择此工作负载时，将自动安装 .NET Core 3.1 SDK。

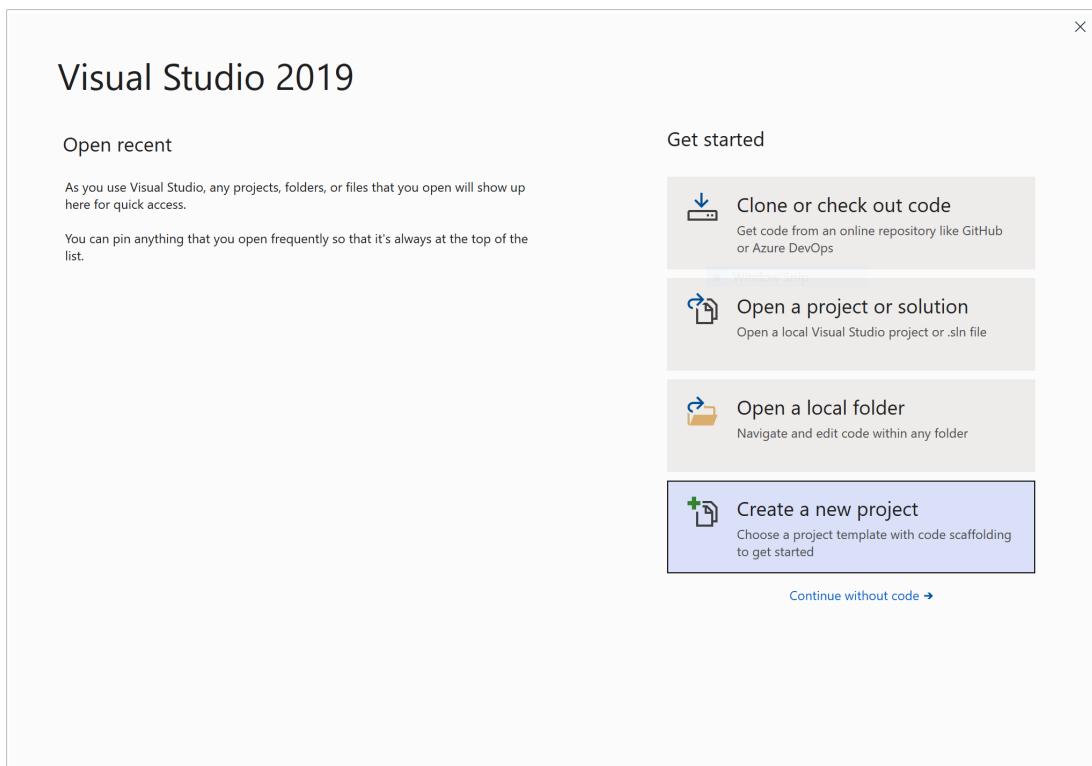
有关详细信息，请参阅[安装 .NET Core SDK](#)一文中的[在 Visual Studio 中安装部分](#)。

创建应用

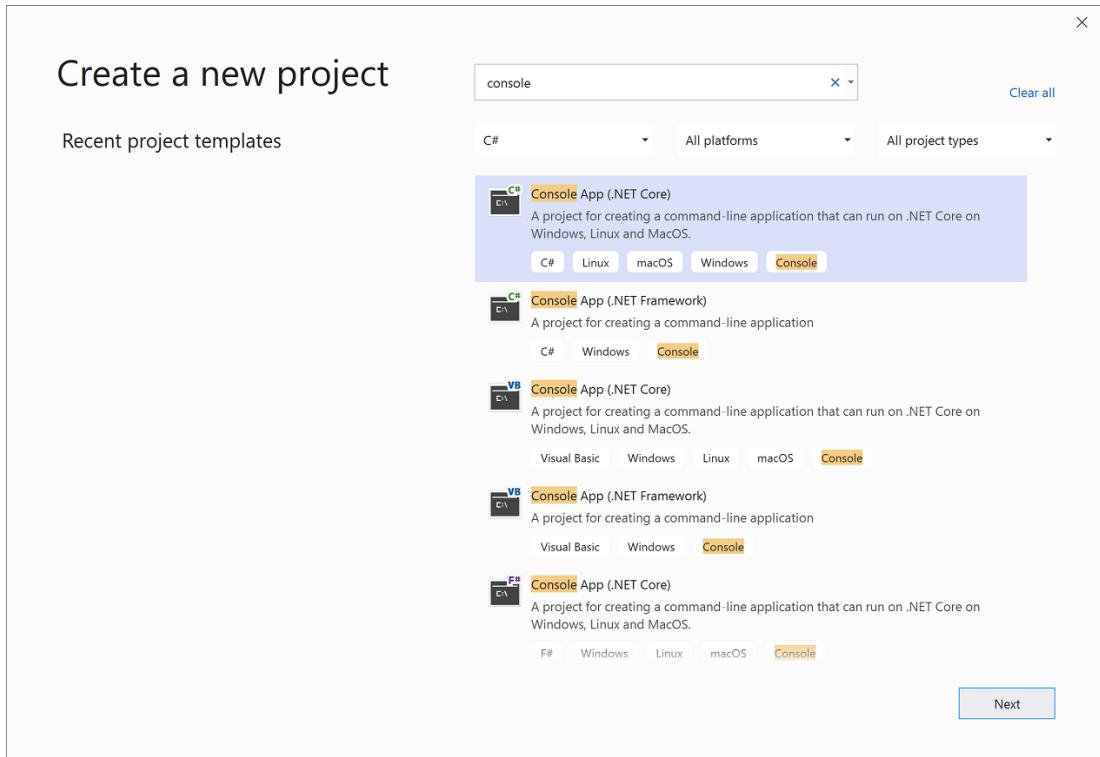
以下说明创建一个简单的 Hello World 控制台应用程序：

- [C#](#)
- [Visual Basic](#)

- 打开 Visual Studio 2019。
- 创建一个名为“HelloWorld”的新 C# .NET Core 控制台应用项目。
 - 在“开始”窗口上，选择“创建新项目”。



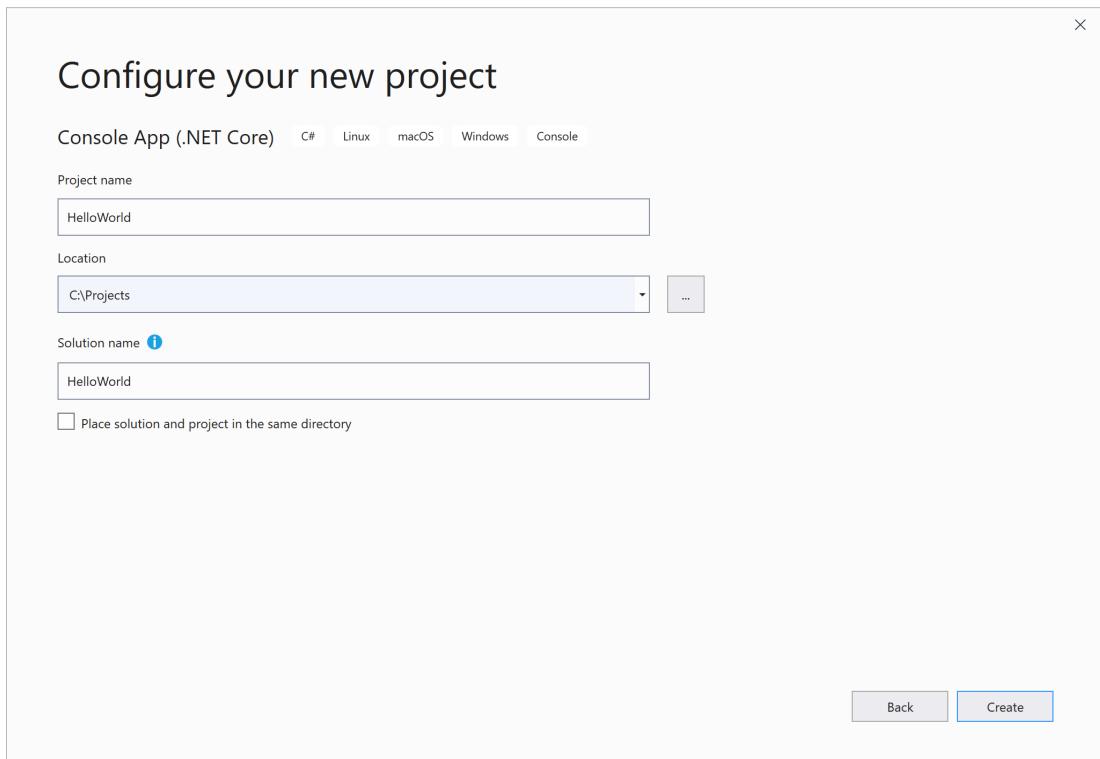
- 在“创建新项目”页面，在搜索框中输入“控制台”。接下来，从“语言”列表中选择“C#”，然后从“平台”列表中选择“所有平台”。选择“控制台应用 (.NET Core)”模板，然后选择“下一步”。



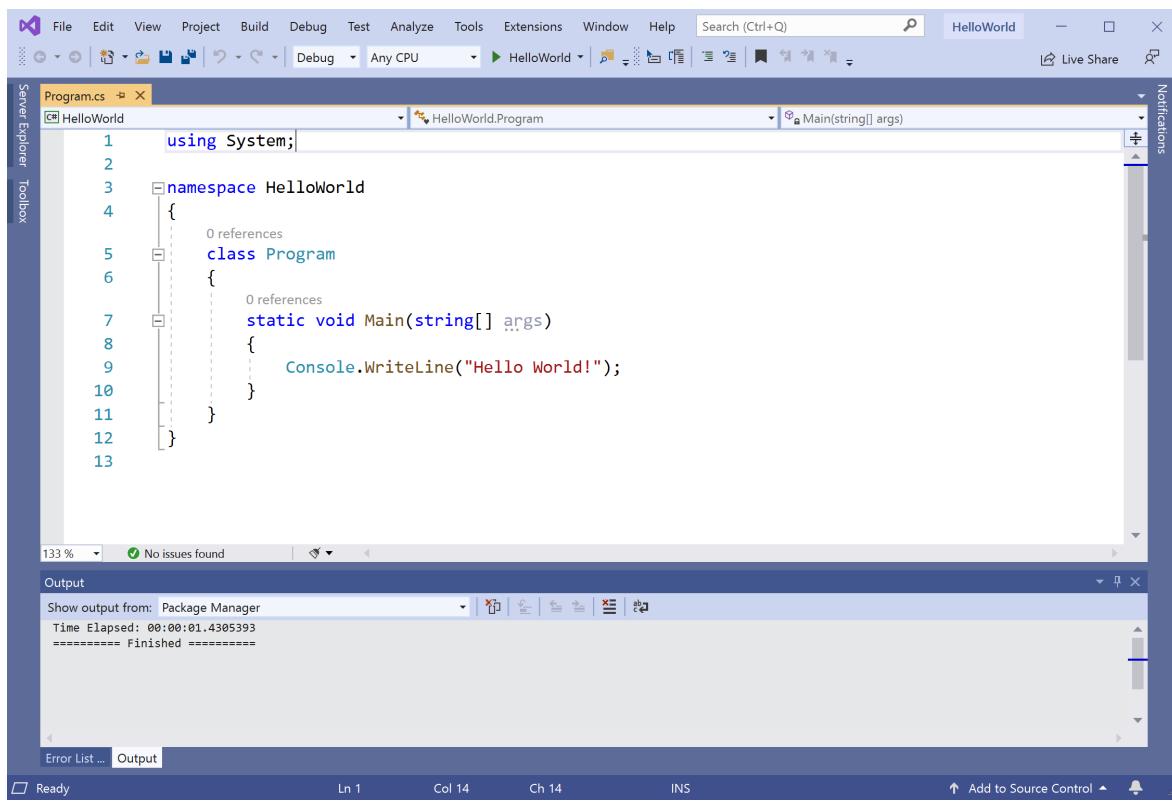
TIP

如果看不到 .NET Core 模板，则可能缺少安装所需的工作负载。在“找不到所需内容？”消息下，选择“安装更多工具和功能”链接。Visual Studio 安装程序随即打开。确保安装了“.NET Core 跨平台开发”工作负载。

- c. 在“配置新项目”页面，在“项目名称”框中输入“HelloWorld”。然后，选择“创建”。



C# .NET Core 控制台应用程序模板会自动定义类 `Program` 和一个需要将 `String` 数组用作自变量的方法 `Main`。`Main` 是应用程序入口点，同时也是在应用程序启动时由运行时自动调用的方法。`args` 数组中包含在应用程序启动时提供的所有命令行自变量。



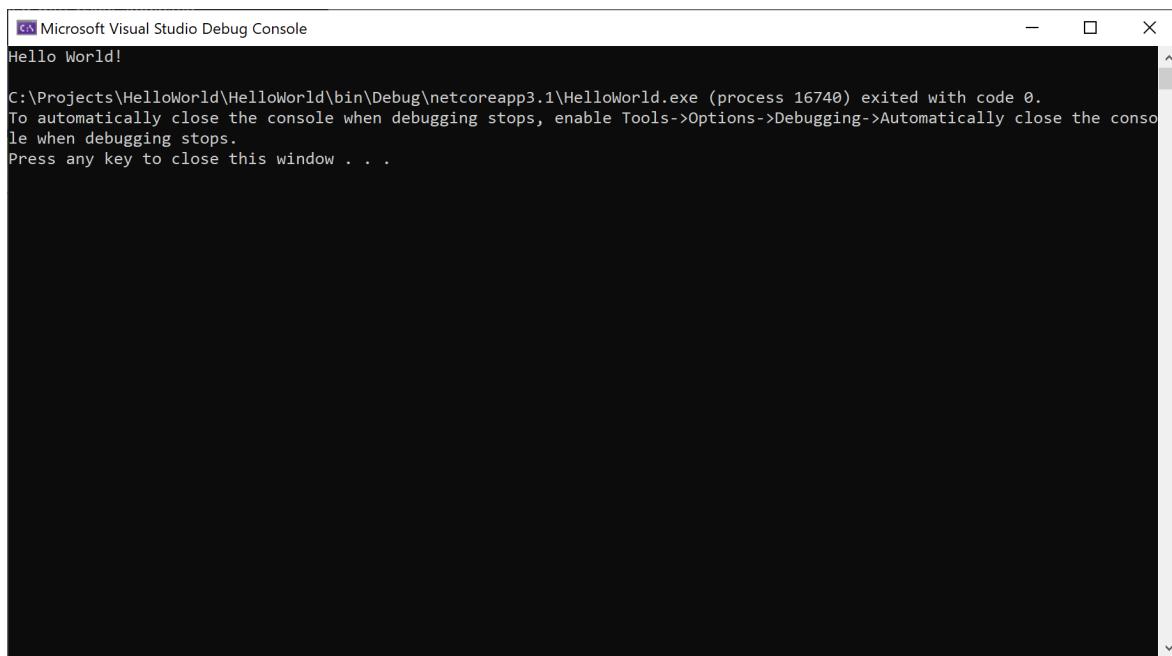
用于创建简单的“Hello World”应用程序的模板。它通过调用 `Console.WriteLine(String)` 方法在控制台窗口中 显示文本字符串“Hello World!”。

运行应用

1. 若要运行程序，请在工具栏上选择“HelloWorld”，或按 F5 。



此时将打开在屏幕上显示文本“Hello World!” 并附带一些 Visual Studio 调试信息的控制台窗口。



2. 按任意键关闭控制台窗口。

增强应用

改进应用程序，提示用户输入名字，并将其与日期和时间一同显示。以下说明再次修改并运行应用：

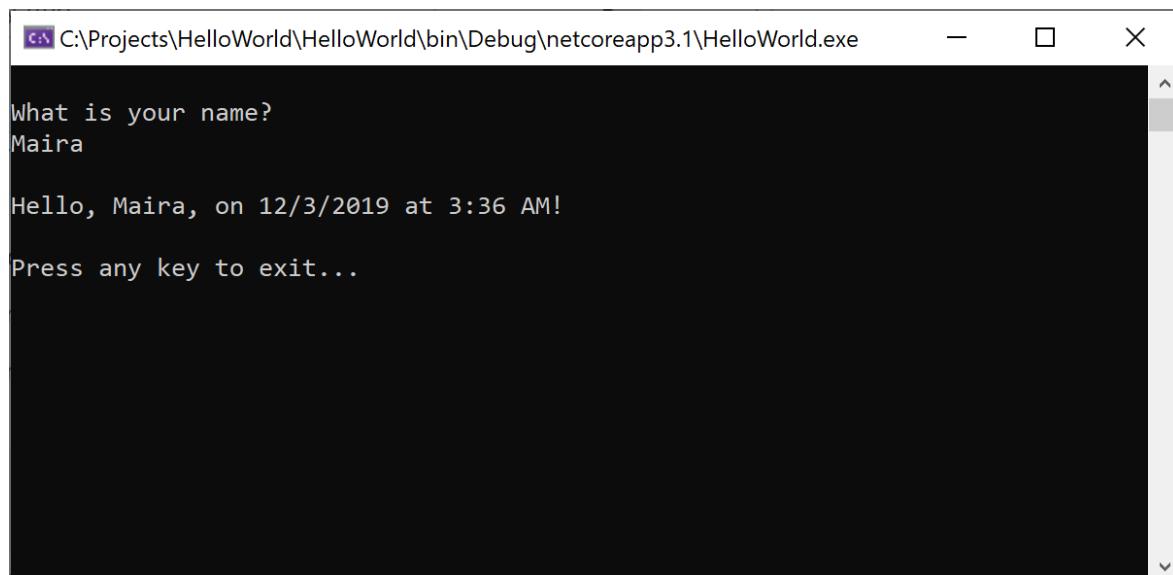
- [C#](#)
- [Visual Basic](#)

1. 将 `Main` 方法的内容(当前只是调用 `Console.WriteLine` 的行)替换为以下代码：

```
Console.WriteLine("\nWhat is your name? ");
var name = Console.ReadLine();
var date = DateTime.Now;
Console.WriteLine($"\\nHello, {name}, on {date:d} at {date:t}!");
Console.Write("\\nPress any key to exit...");
Console.ReadKey(true);
```

此代码在控制台中显示“What is your name?”，然后等待用户输入字符串并按 Enter 键。它将此字符串存储到名为 `name` 的变量中。它还会检索 `DateTime.Now` 属性的值(其中包含当前的本地时间)，并将此值赋给 `date` 变量。最后，使用[内插字符串](#)在控制台窗口中显示这些值。

2. 依次选择“生成”>“生成解决方案”，编译此程序。
3. 若要运行程序，请在工具栏上选择“HelloWorld”，或按 F5。
4. 出现提示时，输入名称并按 Enter 键。



5. 按任意键关闭控制台窗口。

后续步骤

在本文中，你已创建并运行第一个 .NET Core 应用程序。下一步，调试应用。

[在 Visual Studio 中调试 .NET Core Hello World 应用程序](#)

C# 和 Visual Studio Code 入门

2020/4/2 • [Edit Online](#)

.NET Core 提供了快速运行的模块化平台，用于创建在 Windows、Linux 和 macOS 上运行的应用程序。带 C# 扩展的 Visual Studio Code 提供功能强大的编辑体验，完全支持 C# IntelliSense(智能代码填充)和调试。

先决条件

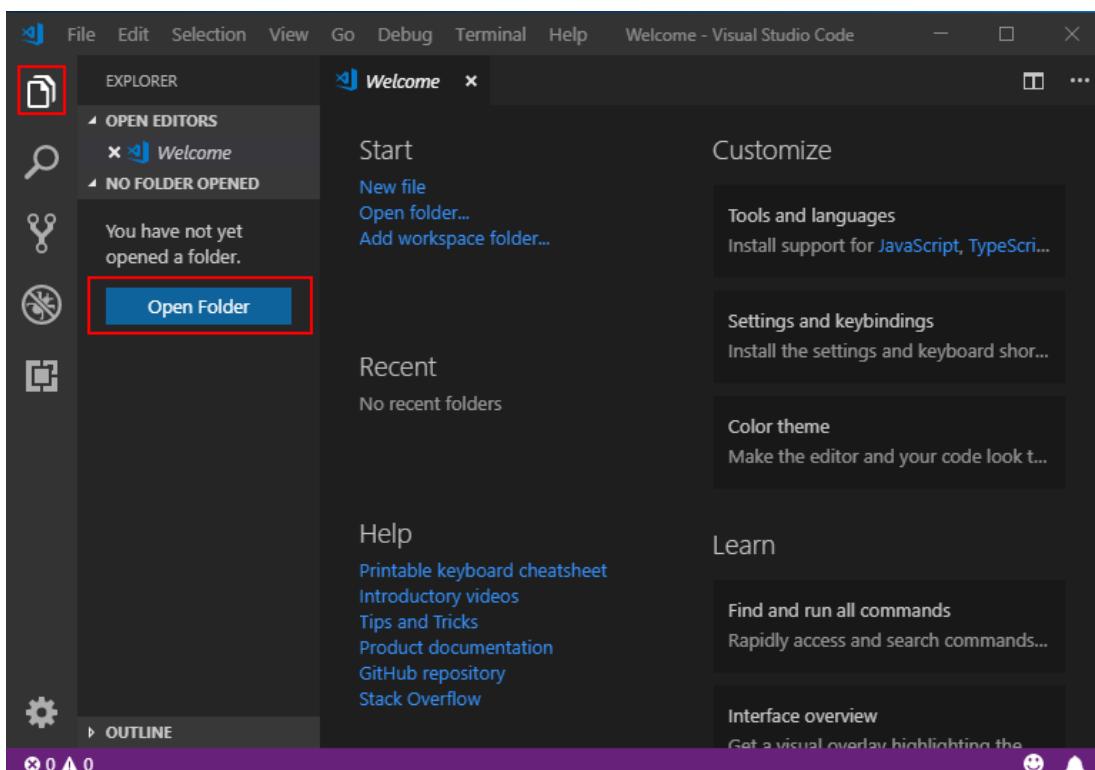
1. 安装 [Visual Studio Code](#)。
2. 获取 [.NET Core SDK](#)。
3. 安装 Visual Studio Code 的 [C# 扩展](#)。若要详细了解如何在 Visual Studio Code 上安装扩展，请访问 [VS Code 扩展市场](#)。

Hello World

让我们从 .NET Core 上的一个简单“Hello World”程序入手：

1. 打开项目：

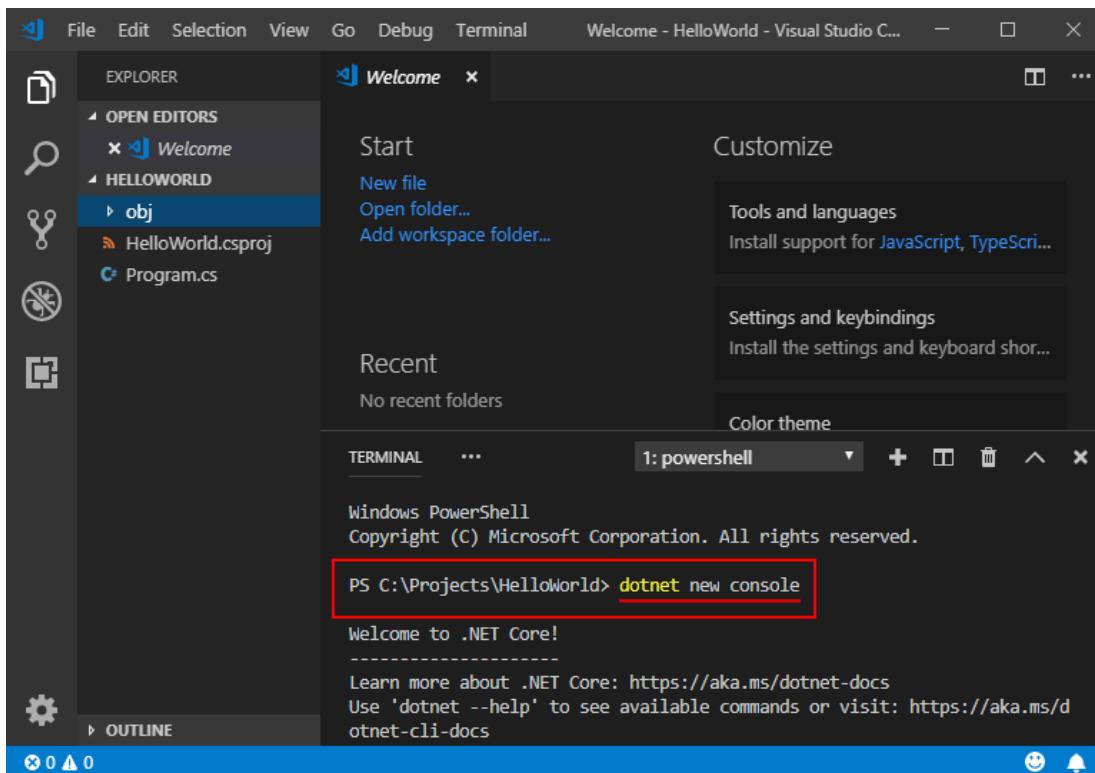
- 打开 Visual Studio Code。
- 依次单击左侧菜单上的“资源管理器”图标和“打开文件夹”。
- 从主菜单中选择“文件”>“打开文件夹”，打开要在其中放置 C# 项目的文件夹，然后单击“选择文件夹”。在我们的示例中，为项目创建名为“HelloWorld”的文件夹。



2. 初始化 C# 项目：

- 通过从主菜单中选择“视图”>“集成终端”，从 Visual Studio Code 中打开集成终端。
- 在终端窗口中，键入 `dotnet new console`。

- 此命令在已编写“Hello World”简单程序的文件夹中创建“Program.cs”文件，以及名为“HelloWorld.csproj”的 C# 项目文件。

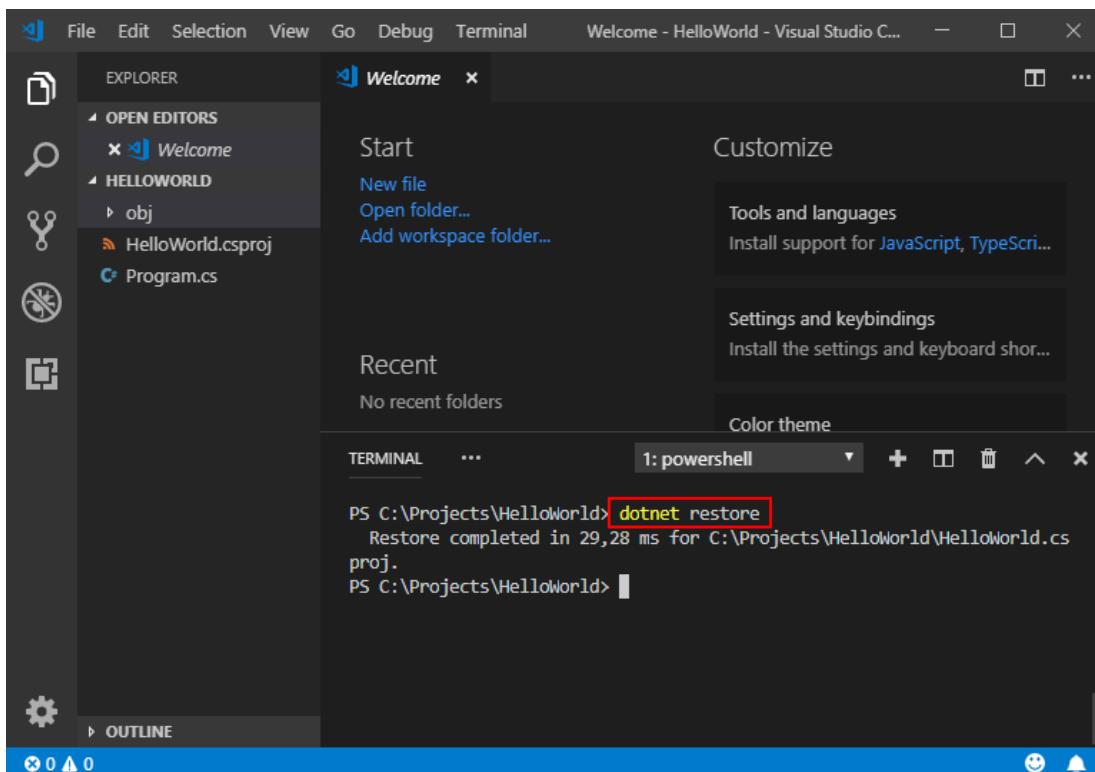


The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left lists a project named 'HELLOWORLD' containing 'obj', 'HelloWorld.csproj', and 'Program.cs'. The Welcome panel is open in the center. In the bottom right terminal window, the command 'dotnet new console' is highlighted with a red box. The output shows the creation of a new .NET Core console application.

```
PS C:\Projects\HelloWorld> dotnet new console
Welcome to .NET Core!
-----
Learn more about .NET Core: https://aka.ms/dotnet-docs
Use 'dotnet --help' to see available commands or visit: https://aka.ms/dotnet-cli-docs
```

3. 解析生成资产：

- 对于 .NET Core 1.x，键入 `dotnet restore`。运行 `dotnet restore` 后，便有权访问生成项目所需的 .NET Core 包。



The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left lists a project named 'HELLOWORLD' containing 'obj', 'HelloWorld.csproj', and 'Program.cs'. The Welcome panel is open in the center. In the bottom right terminal window, the command 'dotnet restore' is highlighted with a red box. The output shows the restoration of NuGet packages for the project.

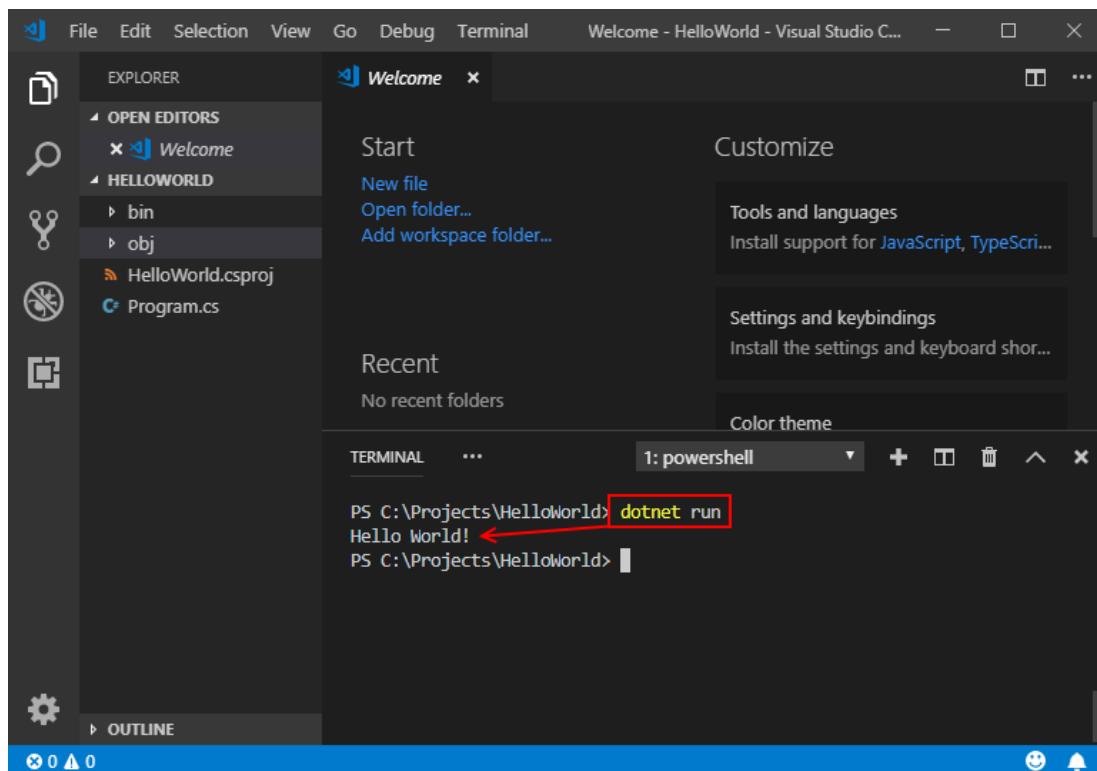
```
PS C:\Projects\HelloWorld> dotnet restore
Restore completed in 29,28 ms for C:\Projects\HelloWorld\HelloWorld.csproj.
PS C:\Projects\HelloWorld>
```

NOTE

从 .NET Core 2.0 SDK 开始，无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build` 和 `dotnet run`。在执行显式还原有意义的某些情况下，例如 Azure DevOps Services 中的持续集成生成中，或在需要显式控制还原发生时间的生成系统中，它仍然是有效的命令。

4. 运行“Hello World”程序：

- 键入 `dotnet run`。



还可以观看简短的视频教程，以获取更多关于在 Windows、macOS 或 Linux 上进行安装的帮助。

调试

1. 单击打开 `Program.cs`。在 Visual Studio Code 中首次打开 C# 文件时，会在编辑器中加载 OmniSharp。

```
1 using System;
2
3 namespace HelloWorld
4 {
5
6     class Program
7     {
8         static void Main(string[] args)
9         {
10             Console.WriteLine("Hello World!");
11         }
12     }
13 }
```

Installing C# dependencies...
Platform: win32, x86_64

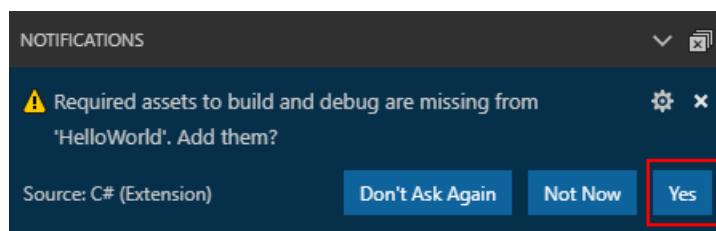
Downloading package 'OmniSharp for Windows (.NET 4.6 / x64)' (31021 KB)..... Done!
Installing package 'OmniSharp for Windows (.NET 4.6 / x64)'

Downloading package '.NET Core Debugger (Windows / x64)' (43046 KB)..... Done!
Installing package '.NET Core Debugger (Windows / x64)'

Downloading package 'Razor Language Server (Windows / x64)' (46894 KB)..... Done!
Installing package 'Razor Language Server (Windows / x64)'

Finished

2. Visual Studio Code 会提示添加缺少的资产，以生成和调试应用。选择“是”。



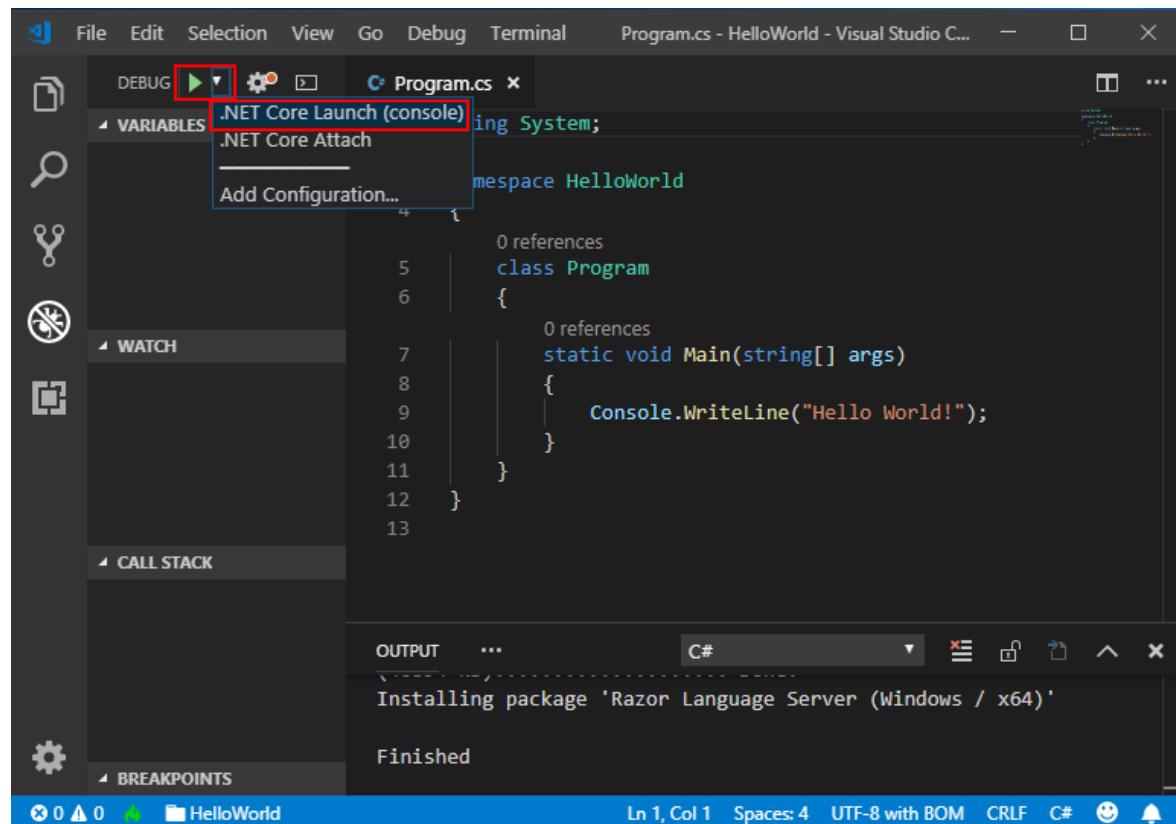
3. 若要打开调试视图，请单击左侧菜单上的“调试”图标。

```
1 using System;
2
3 namespace HelloWorld
4 {
5
6     class Program
7     {
8         static void Main(string[] args)
9         {
10             Console.WriteLine("Hello World!");
11         }
12     }
13 }
```

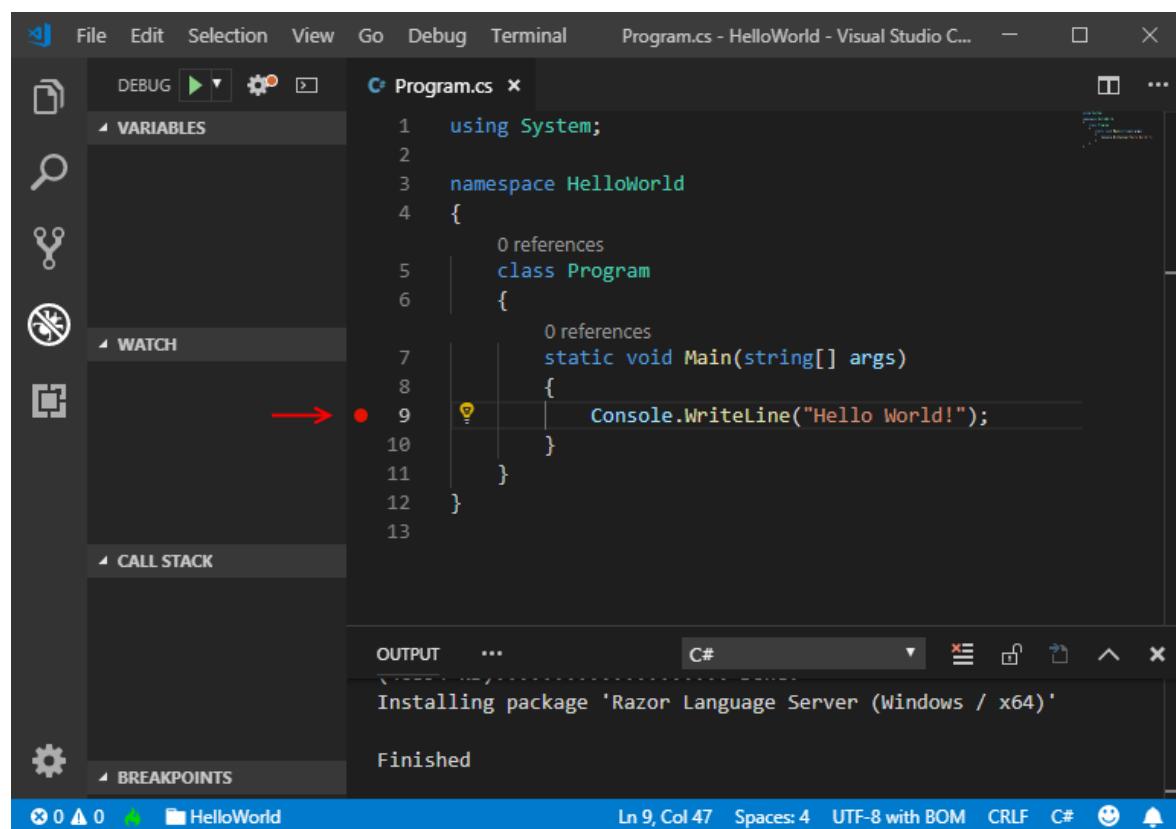
Installing package 'Razor Language Server (Windows / x64)'

Finished

4. 找到窗格最上面的绿色箭头。请确保已选择旁边下拉列表中的“.NET Core Launch (控制台)”。



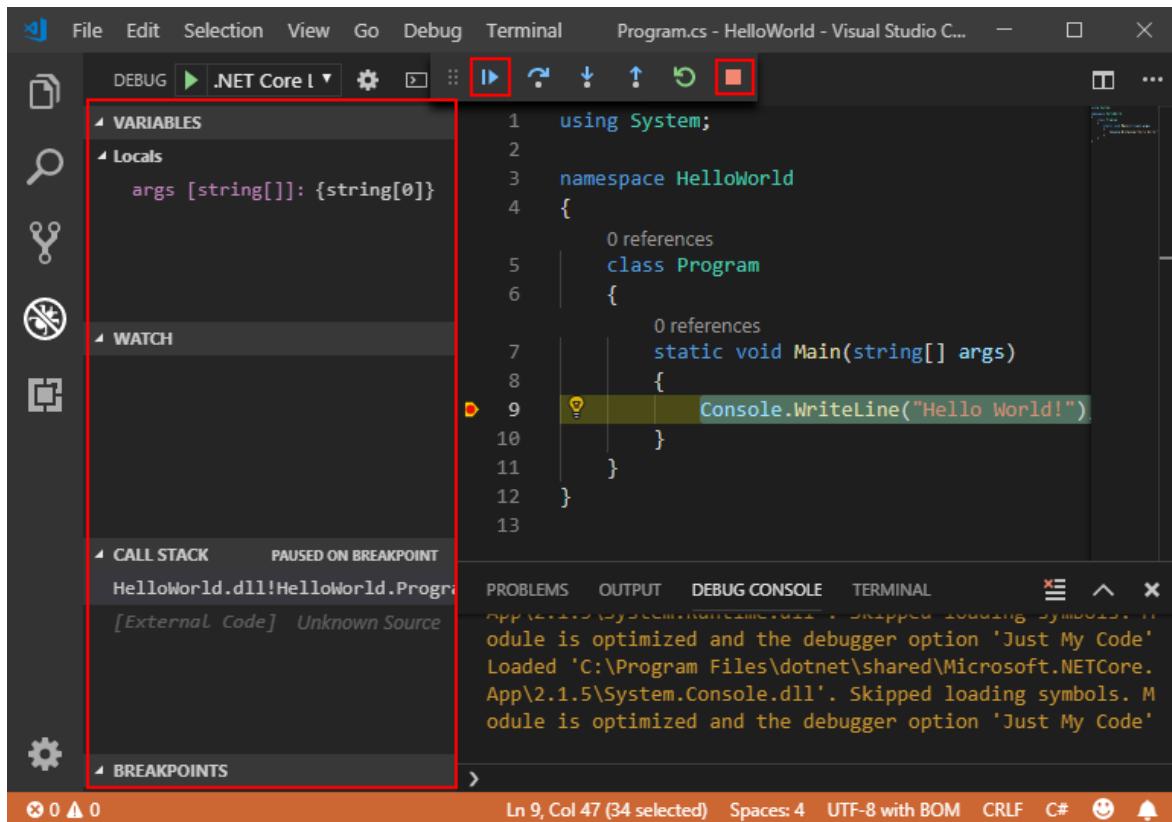
5. 单击第 9 行旁边的编辑器边距 (编辑器中行号左侧的空间)或者将文本光标移动到编辑器中的第 9 行并按 F9, 为项目添加断点。



6. 请按 F5 或选择绿色箭头启动调试。在到达你在上一步中设置的断点时，调试器会停止执行程序。

- 调试时，可以在左上角的窗格中查看局部变量，也可以使用调试控制台进行查看。

7. 选择最上面的蓝色箭头以继续调试，或选择最上面的红色方块以停止调试。



TIP

若要详细了解如何使用 OmniSharp 在 Visual Studio Code 中进行 .NET Core 调试, 以及相关的疑难解答提示, 请参阅[有关设置 .NET Core 调试器的说明](#)。

添加类

1. 若要添加一个新类, 请右键单击 VSCode Explorer 并选择“新文件”。此操作会将新文件添加到在 VSCode 中打开的文件夹中。
2. 将文件命名为 MyClass.cs 。必须在末尾使用 `.cs` 扩展名保存它, 以便将其识别为 csharp 文件。
3. 添加下面的代码, 以创建第一个类。确保包括正确的命名空间, 以便可以从“Program.cs”文件引用它 :

```
using System;

namespace HelloWorld
{
    public class MyClass
    {
        public string ReturnMessage()
        {
            return "Happy coding!";
        }
    }
}
```

4. 通过添加下面的代码, 从“Program.cs”中的主要方法调用新类 :

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            var c1 = new MyClass();
            Console.WriteLine($"Hello World! {c1.ReturnMessage()}");
        }
    }
}
```

5. 保存更改并再次运行程序。新消息应显示追加的字符串。

```
dotnet run
```

将返回以下输出：

```
Hello World! Happy coding!
```

FAQ

缺少在 Visual Studio Code 中生成和调试 C# 所需的资产。调试器显示“无配置”。

Visual Studio Code C# 扩展可生成用于生成和调试的资产。首次打开 C# 项目时，Visual Studio Code 会提示用户生成这些资产。如果当时并未生成这些资产，仍可以通过打开命令面板（“视图”>“命令面板”）并键入“>.NET：生成用于生成和调试的资产”来运行此命令。选择此方法可生成所需的 .vscode、launch.json 和 tasks.json 配置文件。

请参阅

- [设置 Visual Studio Code](#)
- [在 Visual Studio Code 中进行调试](#)

使用 .NET Core CLI 实现 .NET Core 入门

2020/3/18 • [Edit Online](#)

本文介绍如何开始使用 .NET Core CLI 开发在 Windows、Linux 和 macOS 上运行的 .NET Core 应用。

如果不熟悉 .NET Core CLI, 请参阅 [.NET Core 概述](#)。

系统必备

- [.NET Core SDK 3.1](#) 或更高版本。
- 按需选择的文本编辑器或代码编辑器。

Hello, 控制台应用！

若要[查看或下载示例代码](#), 可以访问 dotnet/samples GitHub 存储库。有关下载说明, 请参阅[示例和教程](#)。

打开命令提示符, 创建一个名为“Hello” 的文件夹。导航到创建的文件夹, 键入下列内容。

```
dotnet new console  
dotnet run
```

让我们进行快速演练:

1. `dotnet new console`

`dotnet new` 会创建一个最新的 Hello.csproj 项目文件, 其中包含生成控制台应用所必需的依赖项。此外, 它还会创建一个 Program.cs , 这是包含应用程序入口点的基本文件。

Hello.csproj :

```
<Project Sdk="Microsoft.NET.Sdk">  
  
<PropertyGroup>  
  <OutputType>Exe</OutputType>  
  <TargetFramework>netcoreapp2.2</TargetFramework>  
</PropertyGroup>  
  
</Project>
```

项目文件指定还原依赖项和生成程序所需的一切。

- `<OutputType>` 元素指定我们要生成的可执行文件, 即控制台应用程序。
- `<TargetFramework>` 元素指定要定位的 .NET 实现代码。在高级方案中, 可以指定多个目标框架, 并在单个操作中生成所有目标框架。在本教程中, 我们将仅针对 .NET Core 3.1 进行生成。

Program.cs :

```
using System;

namespace Hello
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

该程序从 `using System` 开始，这意味着“将 `System` 命名空间中的所有内容都纳入此文件的作用域”。`System` 命名空间包括 `Console` 类。

接着定义一个名为 `Hello` 的命名空间。你可以将其更改为任何你喜欢的名称。在该命名空间中定义了一个名为 `Program` 的类，其中 `Main` 方法采用名为 `args` 的字符串数组。此数组包含在运行程序时所传递的参数列表。实际上，不使用此数组，程序只会写入文本“Hello World!” “Hello World!”。稍后将对使用此参数的代码进行更改。

`dotnet new` 隐式调用 `dotnet restore`。`dotnet restore` 调用到 `NuGet`(.NET 包管理器)以还原依赖项树。NuGet 分析 `Hello.csproj` 文件、下载文件中定义的依赖项(或从计算机缓存中获取)并编写 `obj/project.assets.json` 文件，在编译和运行示例时需要使用该文件。

2. `dotnet run`

`dotnet run` 调用 `dotnet build` 来确保已生成要生成的目标，然后调用 `dotnet <assembly.dll>` 运行目标应用程序。

```
dotnet run
```

将获得以下输出。

```
Hello World!
```

或者，还可以运行 `dotnet build` 来编译代码，无需运行已生成的控制台应用程序。这会基于项目的名称将已编译的应用程序作为 DLL 文件生成。在这种情况下，创建的文件命名为 `Hello.dll`。此应用可以使用 Windows 上的 `dotnet bin\Debug\netcoreapp3.1\Hello.dll` 运行(非 Windows 系统使用 `/`)。

```
dotnet bin\Debug\netcoreapp3.1\Hello.dll
```

将获得以下输出。

```
Hello World!
```

在编译应用时，会随 `Hello.dll` 一起创建特定于操作系统的可执行文件。在 Windows 上，这将是 `Hello.exe`；在 Linux 或 macOS 上，这将是 `hello`。在上面的示例中，用 `Hello.exe` 或 `Hello` 命名该文件。可以直接运行该可执行文件。

```
.\bin\Debug\netcoreapp3.1\Hello.exe
```

```
Hello World!
```

修改程序

让我们稍微更改一下程序。Fibonacci 数字很有意思，那么除了使用参数，让我们也来添加 Fibonacci 数字，让运行应用的用户开心一下。

- 将 *Program.cs* 文件的内容替换为以下代码：

```
using System;

namespace Hello
{
    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length > 0)
            {
                Console.WriteLine($"Hello {args[0]}!");
            }
            else
            {
                Console.WriteLine("Hello!");
            }

            Console.WriteLine("Fibonacci Numbers 1-15:");

            for (int i = 0; i < 15; i++)
            {
                Console.WriteLine($"{i + 1}: {FibonacciNumber(i)}");
            }
        }

        static int FibonacciNumber(int n)
        {
            int a = 0;
            int b = 1;
            int tmp;

            for (int i = 0; i < n; i++)
            {
                tmp = a;
                a = b;
                b += tmp;
            }

            return a;
        }
    }
}
```

- 运行 `dotnet build` 以编译更改。
- 运行向应用传递参数的程序。使用 `dotnet` 命令运行应用时，将 `--` 添加到末尾。`--` 右侧的任何内容都将作为参数传递到应用。在下面的示例中，值 `John` 传递到应用。

```
dotnet run -- John
```

将获得以下输出。

```
Hello John!
Fibonacci Numbers 1-15:
1: 0
2: 1
3: 1
4: 2
5: 3
6: 5
7: 8
8: 13
9: 21
10: 34
11: 55
12: 89
13: 144
14: 233
15: 377
```

这就是所有的操作！可以按任意喜欢的方式修改 Program.cs。

使用多个文件

单个文件适用于简单的一次性程序，但如果要构建较为复杂的应用，则项目中可能会有多个代码文件。我们通过缓存一些 Fibonacci 值并添加一些递归特性来基于之前的 Fibonacci 示例进行构建。

1. 使用以下代码将新文件添加到名为 *FibonacciGenerator.cs* 的 *Hello* 目录：

```
using System;
using System.Collections.Generic;

namespace Hello
{
    public class FibonacciGenerator
    {
        private Dictionary<int, int> _cache = new Dictionary<int, int>();

        private int Fib(int n) => n < 2 ? n : FibValue(n - 1) + FibValue(n - 2);

        private int FibValue(int n)
        {
            if (!_cache.ContainsKey(n))
            {
                _cache.Add(n, Fib(n));
            }

            return _cache[n];
        }

        public IEnumerable<int> Generate(int n)
        {
            for (int i = 0; i < n; i++)
            {
                yield return FibValue(i);
            }
        }
    }
}
```

2. 更改 *Main* *Program.cs* 文件中的方法，以实例化新的类并调用其方法，如下例所示：

```
using System;

namespace Hello
{
    class Program
    {
        static void Main(string[] args)
        {
            var generator = new FibonacciGenerator();
            foreach (var digit in generator.Generate(15))
            {
                Console.WriteLine(digit);
            }
        }
    }
}
```

3. 运行 [dotnet build](#) 以编译更改。

4. 通过执行 [dotnet run](#) 来运行应用。

```
dotnet run
```

将获得以下输出。

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
```

发布你的应用

准备好分发应用后，使用 [dotnet publish](#) 命令在 bin_debug_netcoreapp3.1_publish\ (非 Windows 系统使用 \) 处生成 publish 文件夹 \ / 。可以将 publish 文件夹的内容分发到其他平台，只要这些平台安装了 dotnet 运行时即可。

```
dotnet publish
```

将获得类似于下面的输出。

```
Microsoft (R) Build Engine version 16.4.0+e901037fe for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

Restore completed in 20 ms for C:\Code\Temp\Hello\Hello.csproj.
Hello -> C:\Code\Temp\Hello\bin\Debug\netcoreapp3.1>Hello.dll
Hello -> C:\Code\Temp\Hello\bin\Debug\netcoreapp3.1\publish\
```

上面的输出可能会因当前文件夹和操作系统而有所不同，但输出应类似。

可以使用 `dotnet` 命令运行已发布的应用：

```
dotnet bin\Debug\netcoreapp3.1\publish\Hello.dll
```

将获得以下输出。

```
Hello World!
```

如本文开头处所述，会随 `Hello.dll` 一起创建特定于操作系统的可执行文件。在 Windows 上，这将是 `Hello.exe`；在 Linux 或 macOS 上，这将是 `hello`。在上面的示例中，用 `Hello.exe` 或 `Hello` 命名该文件。可以直接运行已发布的可执行文件。

```
.\bin\Debug\netcoreapp3.1\publish\Hello.exe  
Hello World!
```

结束语

这就是所有的操作！现在，可以开始使用此处学到的基本概念来创建自己的程序了。

另请参阅

- [使用 .NET Core CLI 组织和测试项目](#)
- [使用 .NET Core CLI 发布 .NET Core 应用](#)
- [.NET Core 应用程序部署](#)

教程：使用 Visual Studio Code 在 macOS 中创建 .NET Core 解决方案

2020/3/18 • [Edit Online](#)

本文档提供为 macOS 创建 .NET Core 解决方案的步骤和工作流概述。了解到如何通过 NuGet 创建项目、单元测试、使用调试工具和合并第三方库。

NOTE

本文在 macOS 上使用 [Visual Studio Code](#)。

先决条件

获取 [.NET Core SDK](#)。.NET Core SDK 包括最新版本的 .NET Core 框架和运行时。

安装 [Visual Studio Code](#)。在本文中，还将安装可提升 .NET Core 开发体验的 Visual Studio Code 扩展。

打开 Visual Studio Code，并按 Fn+F1 打开 Visual Studio Code 面板，从而安装 Visual Studio Code C# 扩展。键入 ext install，查看扩展列表。选择 C# 扩展。重启 Visual Studio Code 以激活扩展。有关详细信息，请参阅 [Visual Basic Code C# 扩展文档](#)。

入门

在本教程中，将创建三个项目：库项目、对该库项目的测试和使用该库的控制台应用程序。若要[查看或下载本文的源代码](#)，请访问 GitHub 上的 dotnet/samples 存储库。有关下载说明，请参阅[示例和教程](#)。

启动 Visual Studio Code。按 Ctrl` > (反引号) 或在菜单中依次选择“视图”>“终端”，在 Visual Studio Code 中打开嵌入式终端。若要在 Visual Studio Code 外部执行操作，仍可以使用资源管理器的“通过命令提示符打开”（在 macOS 或 Linux 上，为“在终端中打开”）命令打开外部 shell。

首先创建一个解决方案文件，它将用作一个或多个 .NET Core 项目的容器。在终端中，运行 `dotnet new` 命令以在名为 golden 的新文件夹中创建新的解决方案 golden.sln：

```
dotnet new sln -o golden
```

导航到新的 golden 文件夹，执行下列命令来创建库项目，它将在库文件夹中生成 library.csproj 和 Class1.cs 这两个文件：

```
dotnet new classlib -o library
```

执行 `dotnet sln` 命令，将新创建的 library.csproj 添加到解决方案：

```
dotnet sln add library/library.csproj
```

library.csproj 文件包含以下信息：

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
</PropertyGroup>

</Project>
```

库方法以 JSON 格式串行化和反序列化对象。若要支持 JSON 序列化和反序列化，请添加对 `Newtonsoft.Json` NuGet 包的引用。`dotnet add` 命令向项目添加新项。若要添加对 NuGet 包的引用，请使用 `dotnet add package` 命令并指定包的名称：

```
dotnet add library package Newtonsoft.Json
```

这会将 `Newtonsoft.Json` 及其依赖项添加到库项目。或者，可以手动编辑 `library.csproj` 文件，并添加以下节点：

```
<ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="12.0.2" />
</ItemGroup>
```

执行 `dotnet restore` (请参阅注释)，这将还原依赖项，并在库中创建 `obj` 文件夹，该文件夹中包含三个文件，其中一个是 `project.assets.json` 文件：

```
dotnet restore
```

在库文件夹中，将文件 `Class1.cs` 重命名为 `Thing.cs`。将代码替换为以下内容：

```
using static Newtonsoft.Json.JsonConvert;

namespace Library
{
    public class Thing
    {
        public int Get(int left, int right) =>
            DeserializeObject<int>($"{left + right}");
    }
}
```

`Thing` 类包含一个公共方法 `Get`，它返回两个数字的总和，实现方法是将总和转换为字符串，然后反序列化为一个整数。这将使用大量现代 C# 功能，如 `using static` 指令、`expression-bodied 成员` 和 `字符串内插`。

使用 `dotnet build` 命令生成库。这将在 `golden/library/bin/Debug/netstandard1.4` 下生成一个 `library.dll` 文件：

```
dotnet build
```

创建测试项目

生成针对库的测试项目。在 `golden` 文件夹中，创建一个新测试项目：

```
dotnet new xunit -o test-library
```

向解决方案添加测试项目：

```
dotnet sln add test-library/test-library.csproj
```

在上一节创建的库中添加项目引用，这样编译器就可以查找并使用该库项目。使用 `dotnet add reference` 命令：

```
dotnet add test-library/test-library.csproj reference library/library.csproj
```

或者，可以手动编辑 `test-library.csproj` 文件，并添加以下节点：

```
<ItemGroup>
  <ProjectReference Include="..\library\library.csproj" />
</ItemGroup>
```

现已正确配置依赖项，可以开始创建库的测试项目。打开 `UnitTest1.cs`，用以下代码替代其内容：

```
using Library;
using Xunit;

namespace TestApp
{
    public class LibraryTests
    {
        [Fact]
        public void TestThing()
        {
            Assert.NotEqual(42, new Thing().Get(19, 23));
        }
    }
}
```

请注意，在首次创建单元测试（`Assert.NotEqual`）时，已断言值 42 不等于 19+23（或 42），因此测试将失败。生成单元测试的一个重要步骤是：使创建的测试最初失败一次，以便确认其逻辑正确无误。

在 `golden` 文件夹中，执行下列命令：

```
dotnet restore
dotnet test test-library/test-library.csproj
```

这些命令将以递归形式查找所有项目，以还原依赖项、生成依赖性，并激活 xUnit 测试运行程序以运行测试。该测试像预期那样失败。

编辑 `UnitTest1.cs` 文件，将断言从 `Assert.NotEqual` 更改为 `Assert.Equal`。在 `golden` 文件夹中执行下列命令，重新运行测试，此次测试通过：

```
dotnet test test-library/test-library.csproj
```

创建控制台应用

通过以下步骤创建的控制台应用依赖于之前创建的库项目，并在运行时调用其库方法。使用此开发模式，可了解如何创建多个项目的可重用库。

在 `golden` 文件夹中创建新的控制台应用程序：

```
dotnet new console -o app
```

向解决方案添加控制台应用项目：

```
dotnet sln add app/app.csproj
```

运行 `dotnet add reference` 命令，创建库的依赖项：

```
dotnet add app/app.csproj reference library/library.csproj
```

运行 `dotnet restore` ([请参阅注释](#))，在解决方案中还原三个项目的依赖项。打开 `program.cs`，并使用下列行替换 `Main` 方法中的内容：

```
WriteLine($"The answer is {new Thing().Get(19, 23)}");
```

在 `Program.cs` 文件顶部添加两个 `using` 指令：

```
using static System.Console;
using Library;
```

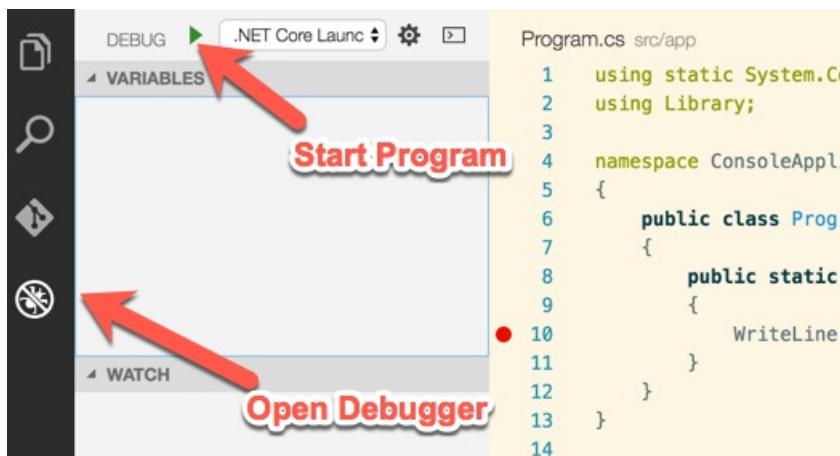
执行下列 `dotnet run` 命令，运行可执行文件，其中，`dotnet run` 后的 `-p` 选项用于指定主应用程序的项目。应用会生成字符串“`The answer is 42`”。

```
dotnet run -p app/app.csproj
```

调试应用程序

在 `Main` 方法中的 `WriteLine` 语句处设置一个断点。要实现此操作，可在光标位于 `WriteLine` 行之上时按 `F9` 键，也可在想要设置断点的行的左侧边缘中单击鼠标。代码行旁边的边缘中将出现一个红色圆圈。到达断点时，将在执行断点行前停止执行代码。

若要打开“调试器”选项卡，请在 Visual Studio Code 工具栏中选择“调试”图标，再从菜单栏中依次选择“视图”>“调试”，或使用键盘快捷方式 `Shift+F5`：



按“开始”按钮，在调试器下启动应用程序。你已在此项目中创建了测试项目和应用程序。调试器会询问你要启动哪个项目。选择“应用”项目。应用开始执行，运行到断点处停止。单步执行 `Get` 方法，确保已传入正确的参数。确认答案是 42。

NOTE

从 .NET Core 2.0 SDK 开始，无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build` 和 `dotnet run`。在执行显式还原有意义的某些情况下，例如 [Azure DevOps Services 中的持续集成生成中](#)，或在需要显式控制还原发生时间的生成系统中，它仍然是有效的命令。

通过 Visual Studio for Mac 开始在 macOS 上使用 .NET Core

2020/3/18 • [Edit Online](#)

Visual Studio for Mac 提供用于开发 .NET Core 应用程序的功能全面的集成开发环境 (IDE)。本文将指导你使用 Visual Studio for Mac 和 .NET Core 来构建简单的控制台应用程序。

NOTE

你的反馈非常有价值。有两种方法可以向开发团队提供有关 Visual Studio for Mac 的反馈：

- 在 Visual Studio for Mac 中，从菜单中选择“帮助”>“报告问题”，或从欢迎屏幕中选择“报告问题”，将打开一个窗口，以供填写 bug 报告。可在[开发人员社区](#)门户中跟踪自己的反馈。
- 若要提出建议，从菜单中选择“帮助”>“提供建议”，或从欢迎屏幕中选择“提供建议”，转到[Visual Studio for Mac 开发人员社区](#)网页。

系统必备

请参阅[.NET Core 依赖项和要求](#)一文。

请查看[.NET Core 支持](#)一文，确保使用的是受支持的 .NET Core 版本。

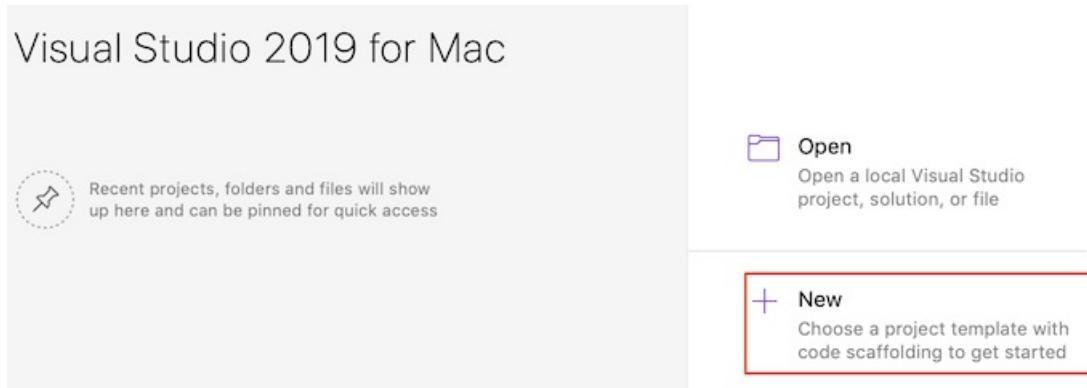
入门

如果已安装先决条件和 Visual Studio for Mac，请跳过此部分，并继续[创建项目](#)。请按照以下步骤安装先决条件和 Visual Studio for Mac：

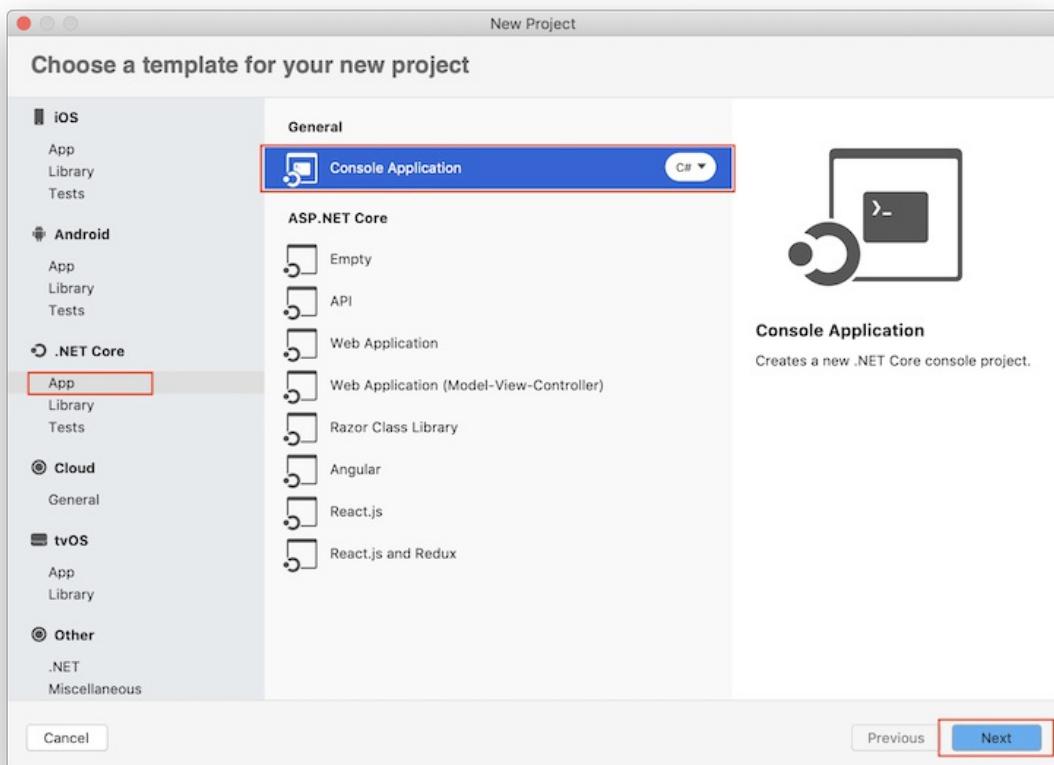
下载[Visual Studio for Mac 安装程序](#)。运行安装程序。阅读并同意许可协议。在安装过程中，选择用于安装 .NET Core 的选项。你有机会安装 Xamarin，这是一项跨平台移动应用开发技术。安装 Xamarin 及其相关组件对于 .NET Core 开发而言是可选项。有关 Visual Studio for Mac 安装过程的分步介绍，请参阅[Visual Studio for Mac 文档](#)。安装完成后，启动 Visual Studio for Mac IDE。

创建项目

1. 在“开始”窗口上，选择“新建”。

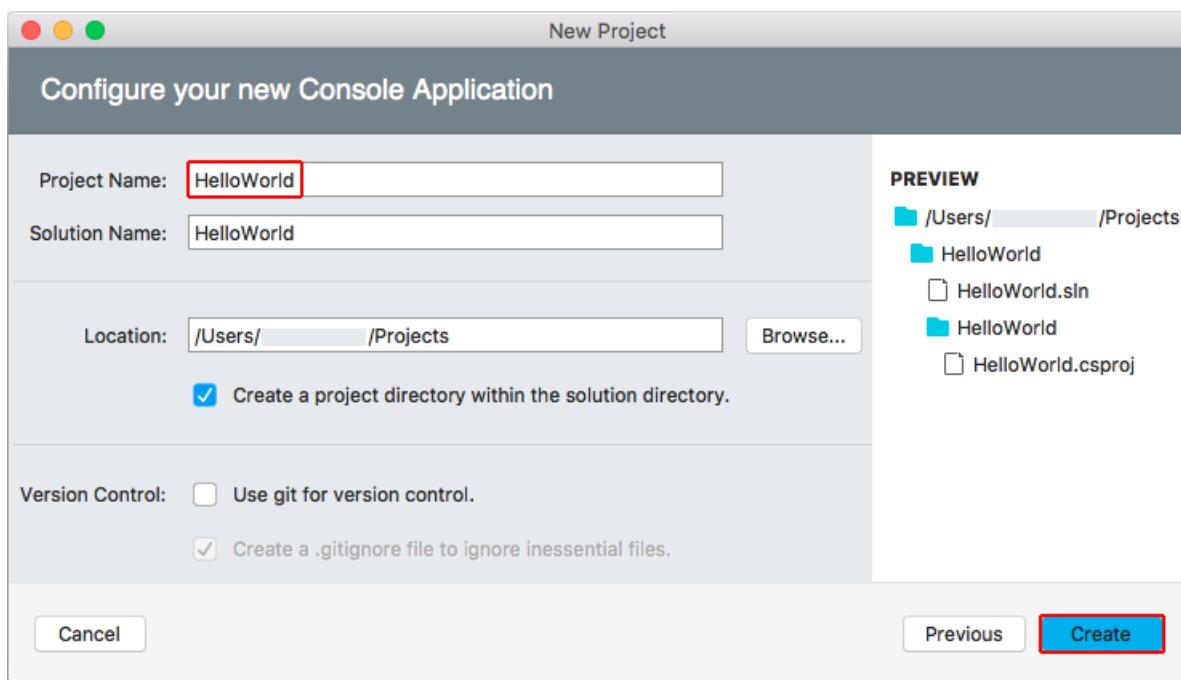


2. 在“新建项目”对话框中，选择“.NET Core”节点下的“应用”。单击“下一步”，然后选择“控制台应用程序”模板。



3. 如果安装了多个版本的 .NET Core, 请选择项目的目标框架。

4. 为“项目名称”键入“HelloWorld”。选择“创建”。



5. 等待还原项目的依赖项。该项目包含一个 C# 文件 `Program.cs`, 其中包含具有 `Program` 方法的 `Main` 类。运行应用时, `Console.WriteLine` 语句将“Hello World!”输出至控制台。

A screenshot of the Visual Studio for Mac interface. The left sidebar shows a 'Solution' tree with a 'HelloWorld' project containing 'Dependencies' and 'Program.cs'. The 'Program.cs' file is selected and highlighted with a red box. The main editor window displays the following C# code:

```
1 using System;
2
3 namespace HelloWorld
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             Console.WriteLine("Hello World!");
10        }
11    }
12 }
13
```

The code block from 'Main' to the closing brace of 'Program' is also highlighted with a red box. The status bar at the bottom shows 'Errors', 'Tasks', and 'Package Console'.

运行此应用程序

使用 ⌘ ↵(command + enter 键)在调试模式下运行应用, 或者使用 ⌘ ⌥ ↵(option + command + enter 键)在发布模式下运行应用。

A screenshot of the 'Application Output' window. It contains the text 'Hello World!' which is highlighted with a red box. The window has standard OS X-style controls (close, minimize, zoom) and a toolbar at the bottom with 'Errors', 'Tasks', and 'Package Console' buttons.

下一步

[使用 Visual Studio for Mac 在 macOS 上构建完整的 .NET Core 解决方案](#)主题为你演示如何构建包含可重用的库和单元测试的完整的 .NET Core 解决方案。

使用 Visual Studio 调试 C# 或 Visual Basic .NET Core Hello World 应用程序

2020/3/18 • [Edit Online](#)

至此，已按照[使用 Visual Studio 2019 创建第一个 .NET Core 控制台应用程序](#)中的步骤操作，创建和运行简单的控制台应用程序。编写和编译应用程序后，可以开始进行测试。Visual Studio 提供一整套调试工具，方便你在排除应用程序故障时使用。

“调试”生成配置

“调试”和“发布”是 Visual Studio 的两种默认生成配置。当前的生成配置显示在工具栏上。下面的工具栏图像显示 Visual Studio 配置为编译应用的“调试”版本：



首先运行应用的“调试”版本。“调试”生成配置会禁用大多数编译器优化，并在生成过程中提供更丰富的信息。

设置断点

运行程序，并尝试一些调试功能：

- [C#](#)
- [Visual Basic](#)

1. 通过在读取的行上单击代码窗口的左边距来设置该行的断点

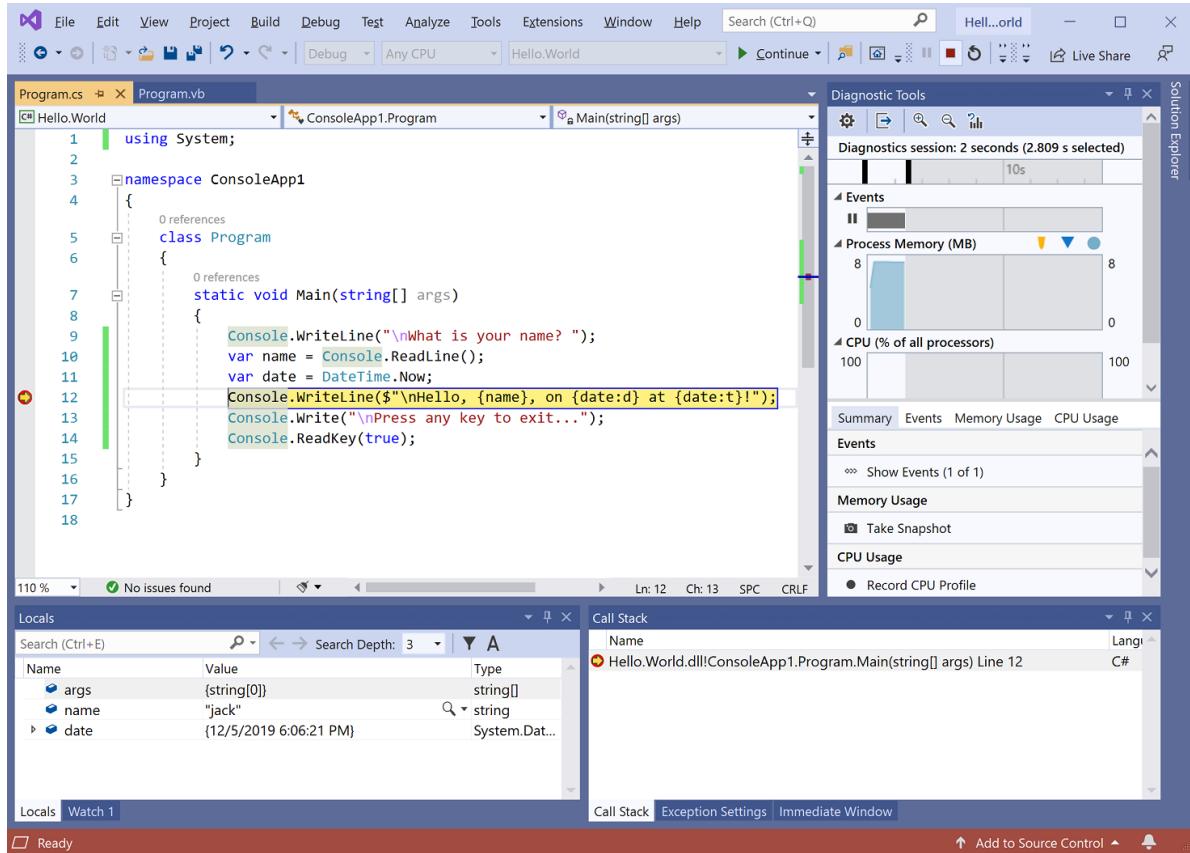
`Console.WriteLine($"\\nHello, {name}, on {date:d} at {date:t}!");`。也可以通过将插入符号置于代码行中，然后按 F9 或从菜单栏中选择“调试”>“切换断点”来设置断点。

断点会在执行包含断点的代码行之前暂时中断执行应用程序。

如下图所示，Visual Studio 通过突出显示此代码行，并在其左侧边缘显示红色圆圈来指明该行设置了断点。

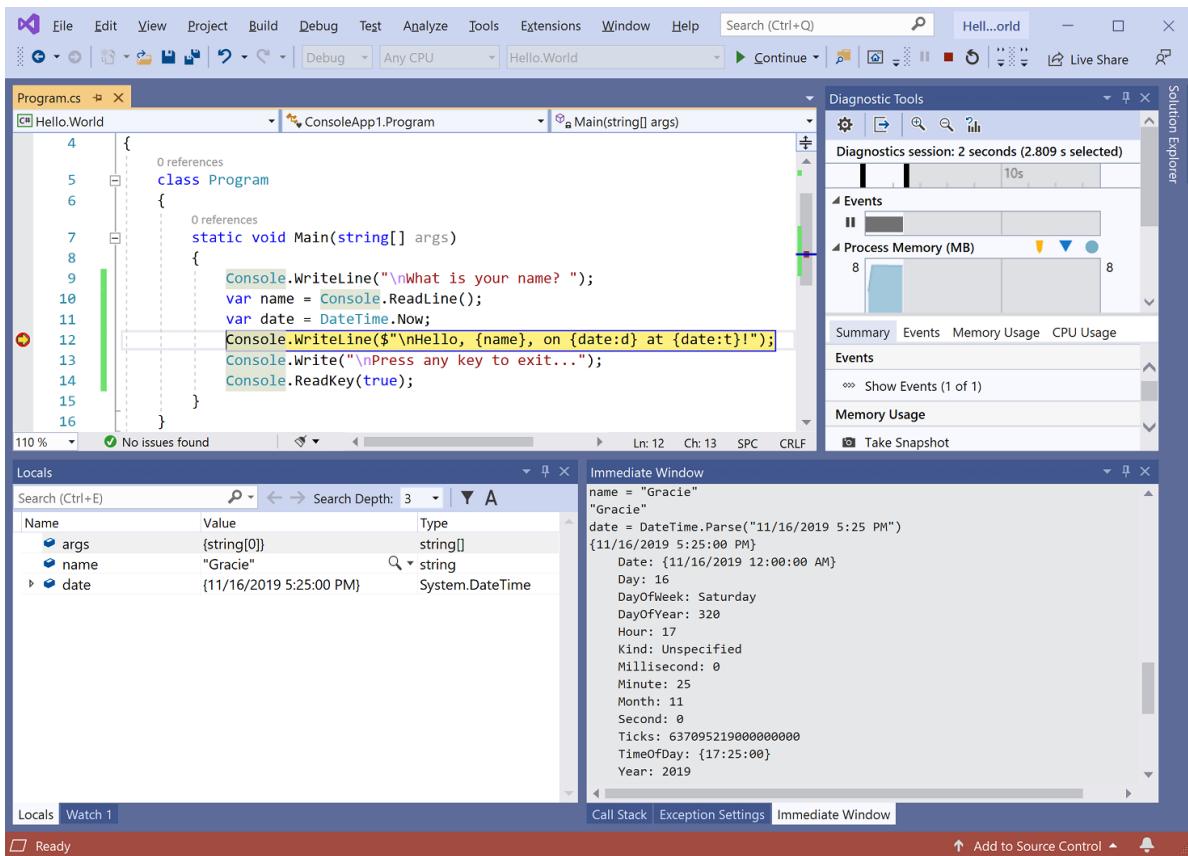
A screenshot of the Visual Studio code editor showing the file 'Program.cs'. The code defines a 'Program' class with a 'Main' method. A red circle marks a breakpoint on line 12, which contains the code `Console.WriteLine($"\\nHello, {name}, on {date:d} at {date:t}!");`. The code editor interface includes tabs for 'Program.cs', 'HelloWorld', and 'Main(string[] args)', and a status bar at the bottom.

2. 选择工具栏上含绿色箭头的“HelloWorld”按钮、按 F5 或选择“调试” > “启动调试”，在“调试”模式下运行程序。
3. 当程序提示输入名称时，在控制台窗口中输入字符串，然后按 Enter。
4. 到达断点时，程序停止执行，然后执行 `Console.WriteLine` 方法。“局部变量”窗口显示当前正在执行的方法中定义的变量值。

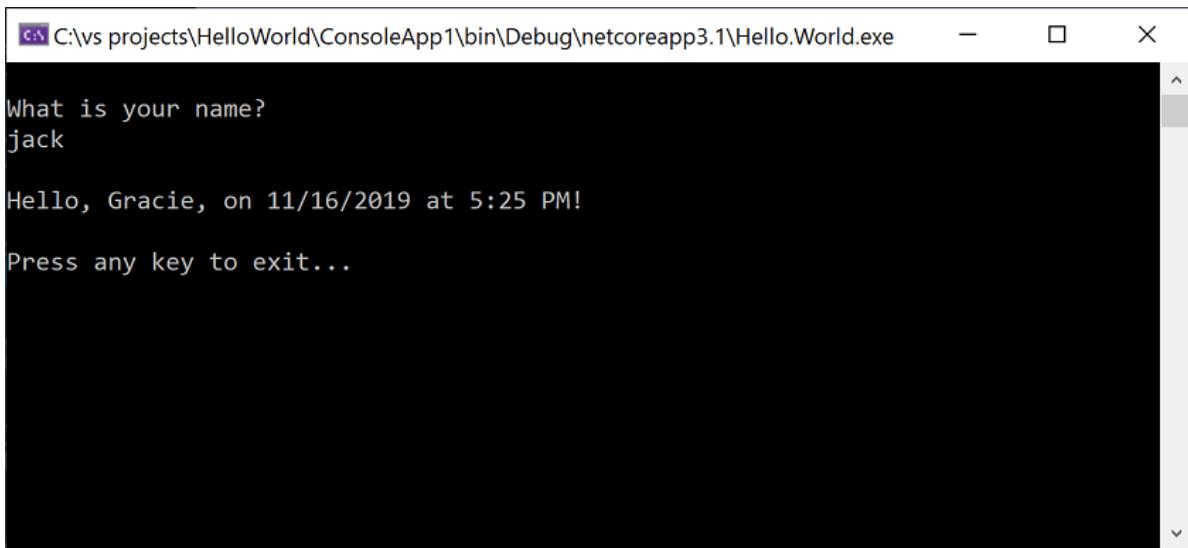


5. 在“即时”窗口中，可以与正在调试的应用程序进行交互。可以通过交互方式更改变量值，看看这样会对程序产生哪些影响。
 - a. 如果“即时”窗口不可见，请选择“调试” > “Windows” > “即时” 来显示它。
 - b. 在“即时” `name = "Gracie"` 窗口中输入，然后按 Enter 键。
 - c. 在“即时” `date = DateTime.Parse("11/16/2019 5:25 PM")` 窗口中输入，然后按 Enter 键。

“即时”窗口显示字符串变量的值和 `DateTime` 值的属性。此外，“局部变量”窗口中也会更新变量值。



- 选择工具栏中的“继续”按钮，或选择“调试” > “继续”，继续执行程序。控制台窗口中显示的值对应于在“即时”窗口中所做的更改。



- 按任意键，退出应用程序并停止调试。

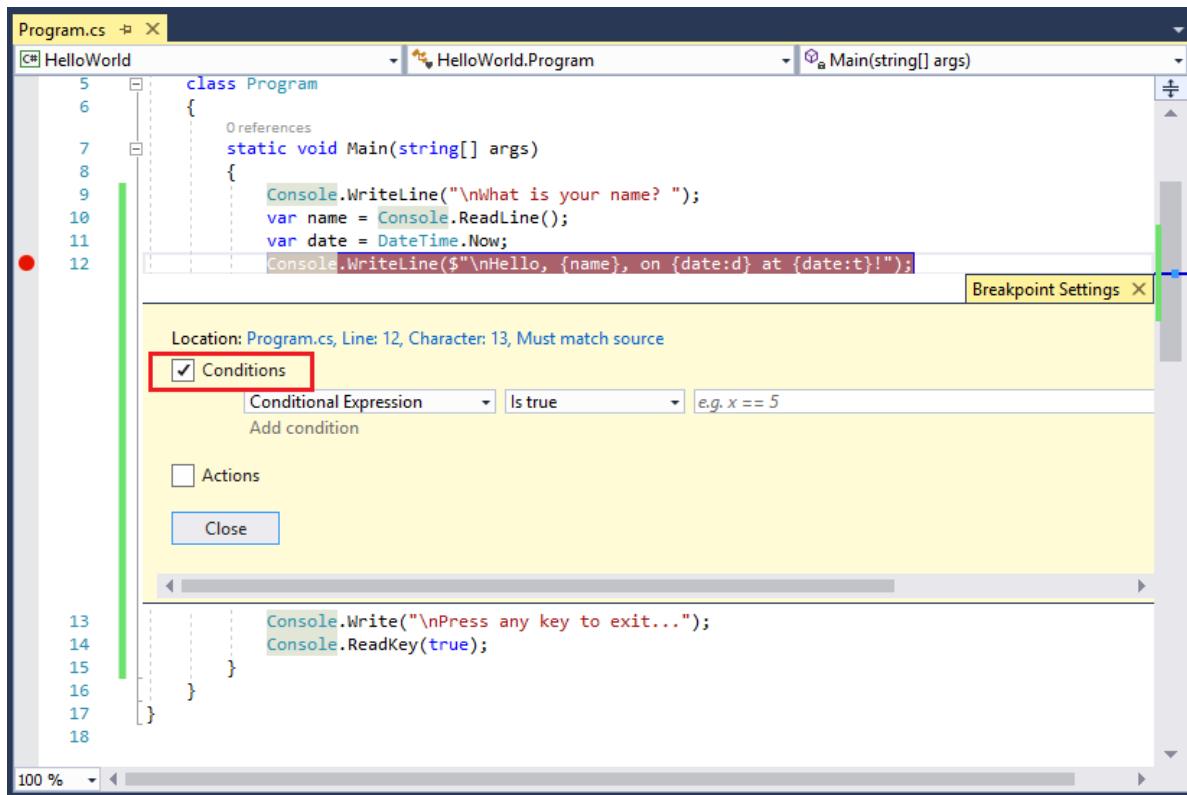
设置条件断点

程序显示用户输入的字符串。如果用户没有输入任何内容，情况又如何呢？可使用名为“条件断点”的实用调试功能来测试这种情况，该功能可在满足一个或多个条件时中断程序执行。

若要设置条件断点，并测试用户无法输入字符串时的情况如何，请执行以下操作：

- C#
- Visual Basic

- 右键单击表示断点的红点。在上下文菜单中，选择“条件”，打开“断点设置”对话框。选择“条件”框（如果尚未选择）。



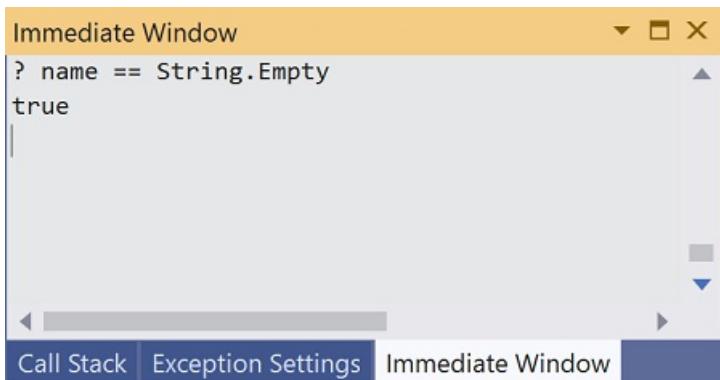
2. 对于“条件表达式”，将“e.g. `x == 5`”替换为以下内容：

```
String.IsNullOrEmpty(name)
```

要测试的代码条件是 `String.IsNullOrEmpty(name)` 方法调用是否为 `true`，因为 `name` 尚未分配有值或值为空字符串（`""`）。与条件表达式不同，你还可以指定命中次数，这样程序就会在语句的执行次数达到指定值时中断执行；也可以指定筛选条件，这样就可以根据诸如线程标识符、进程名称或线程名称之类的属性来中断程序执行。

3. 选择“关闭”以关闭对话框。
4. 通过按 F5 调试来启动程序。
5. 在控制台窗口中，在看到输入名称的提示时按 Enter 键。
6. 由于符合指定的条件（`name` 为 `null` 或 `String.Empty`），因此程序在到达断点时以及在 `Console.WriteLine` 方法执行前停止执行。
7. 选择“局部变量”窗口，其中显示当前正在执行的方法的局部变量值。在这种情况下，`Main` 是当前正在执行的方法。请注意，`name` 变量的值为 `""` 或 `String.Empty`。
8. 在“即时”窗口中输入下面的语句并按 Enter，确认值为空字符串。结果为 `true`。

```
? name == String.Empty
```



9. 选择工具栏上的“继续”按钮，继续执行程序。
10. 按任意键，关闭控制台窗口并停止调试。
11. 单击代码窗口左侧边缘上的点，或选中该代码行，选择“调试”>“切换断点”，从而清除断点。

单步执行程序

使用 Visual Studio，还可以单步执行程序，并监视其执行情况。通常可以设置断点，并使用此功能单步执行程序流，尽管是一小部分程序代码。因为程序很小，因此可以单步执行整个程序：

- C#
- Visual Basic

1. 在菜单栏上，选择“调试”>“单步执行”，或按 F11。Visual Studio 会在要执行的下一行旁边突出显示一个箭头。



此时，“局部变量”窗口显示程序只定义了一个变量，即 `args`。由于尚未向程序传递任何命令行自变量，因此它的值是一个空字符串数组。此外，Visual Studio 还打开了一个空白控制台窗口。

2. 选择“调试”>“单步执行”，或按 F11。Visual Studio 现在突出显示要执行的下一行。如图所示，从上一语句执行到该行代码花费了不到 1 毫秒的时间。`args` 仍然是唯一声明的变量，控制台窗口仍为空白。

```
Program.cs
HelloWorld
HelloWorld.Program
Main(string[] args)
{
    Console.WriteLine("\nWhat is your name? ");
    var name = Console.ReadLine();
    var date = DateTime.Now;
    Console.WriteLine($"\\nHello, {name}, on {date:d} at {date:t}!");
    Console.Write("\\nPress any key to exit...");
    Console.ReadKey(true);
}
```

3. 选择“调试”>“单步执行”，或按 F11。Visual Studio 突出显示包含 `name` 变量赋值的语句。“局部变量”窗口显示 `name` 为 `null`，控制台窗口显示字符串“What is your name?”。
4. 在控制台窗口中输入字符串，然后按 Enter，从而响应提示。控制台无响应，输入的字符串未显示在控制台窗口中，但 `Console.ReadLine` 方法将捕获输入。
5. 选择“调试”>“单步执行”，或按 F11。Visual Studio 突出显示包含 `date` 变量赋值的语句。“局部变量”窗口显示 `Console.ReadLine` 方法调用返回的值。控制台窗口还显示在提示符处输入的字符串。
6. 选择“调试”>“单步执行”，或按 F11。“局部变量”窗口显示通过 `date` 属性赋值后的 `DateTime.Now` 变量值。控制台窗口保持不变。
7. 选择“调试”>“单步执行”，或按 F11。Visual Studio 调用 `Console.WriteLine(String, Object, Object)` 方法。控制台窗口会显示格式化的字符串。
8. 选择“调试”>“跳出”，或按 Shift + F11。这将停止单步执行。控制台窗口会显示一条消息，并等待用户按任意键。
9. 按任意键，关闭控制台窗口并停止调试。

生成“发布”版本

测试应用程序的“调试”版本后，还应该编译并测试“发布”版本。发布版本包含编译器优化，有时可能会对应用程序的行为产生不良影响。例如，旨在提升性能的编译器优化可能会在多线程应用程序中创建争用条件。

若要生成和测试控制台应用程序的发布版本，请将工具栏上的生成配置从“调试”更改为“发布”。



按 F5 或选择“生成”菜单中的“生成解决方案”后，Visual Studio 会编译应用程序的“发布”版本。可像测试“调试”版本一样测试“发布”版本。

后续步骤

调试完应用程序后，下一步是发布应用程序的可部署版本。若要了解如何执行此操作，请参阅[使用 Visual Studio 发布 .NET Core Hello World 应用程序](#)。

使用 Visual Studio 发布 .NET Core Hello World 应用程序

2020/3/18 • [Edit Online](#)

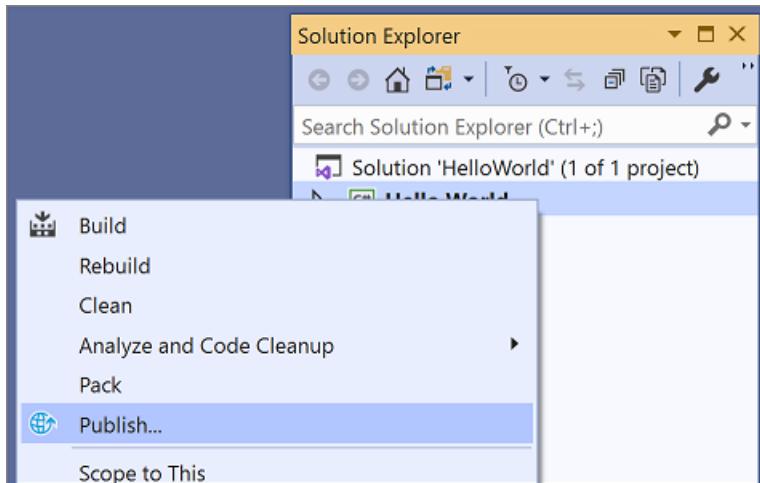
在[在 Visual Studio 中使用 .NET Core 创建 Hello World 应用程序](#)中，生成了 Hello World 控制台应用程序。在[使用 Visual Studio 调试 Hello World 应用程序](#)中，使用 Visual Studio 调试程序测试了应用程序。至此，你已确定应用程序能够按预期运行，可以发布它以供其他用户运行了。发布应用程序会创建运行应用程序所需的一组文件。若要部署文件，请将文件复制到目标计算机。

发布应用

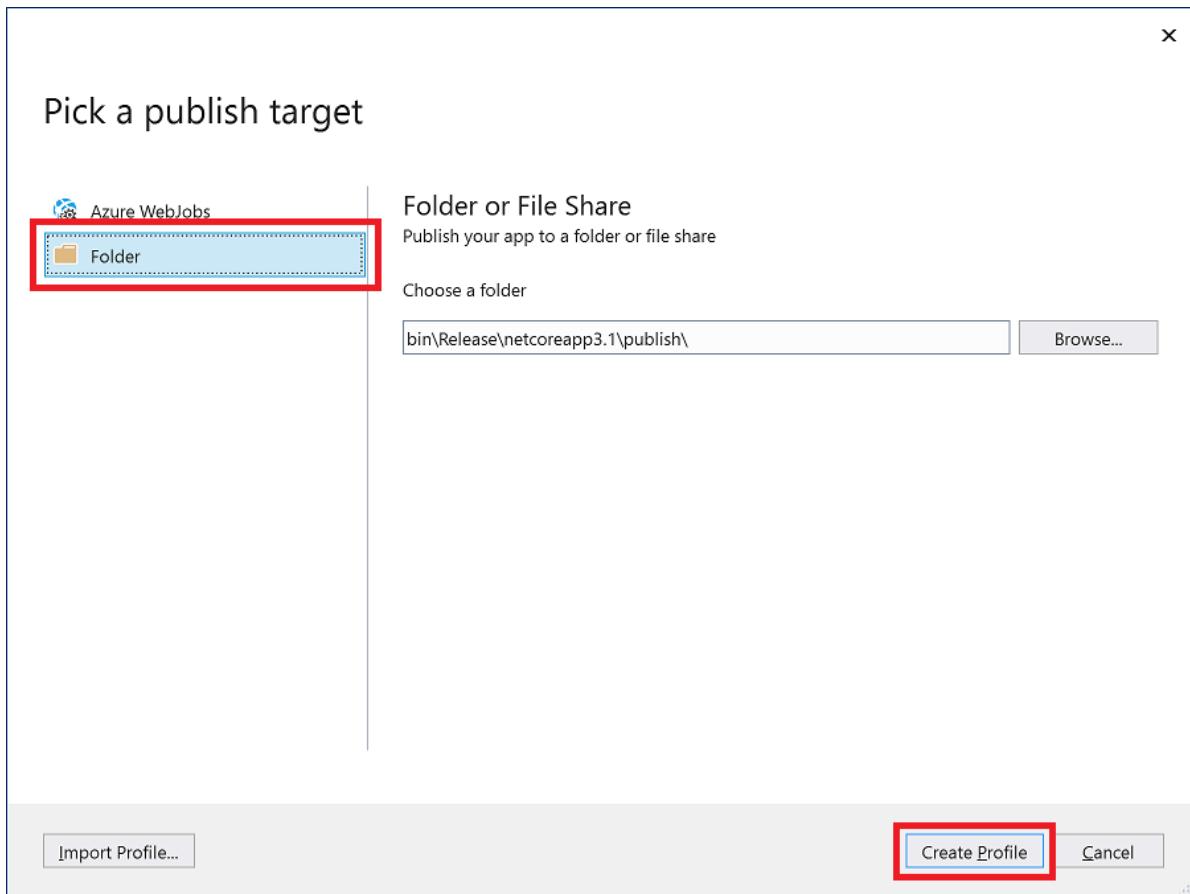
1. 请确保 Visual Studio 生成的是应用程序的发布版本。必要时，将工具栏上的生成配置设置从“调试”更改
为“发布”。



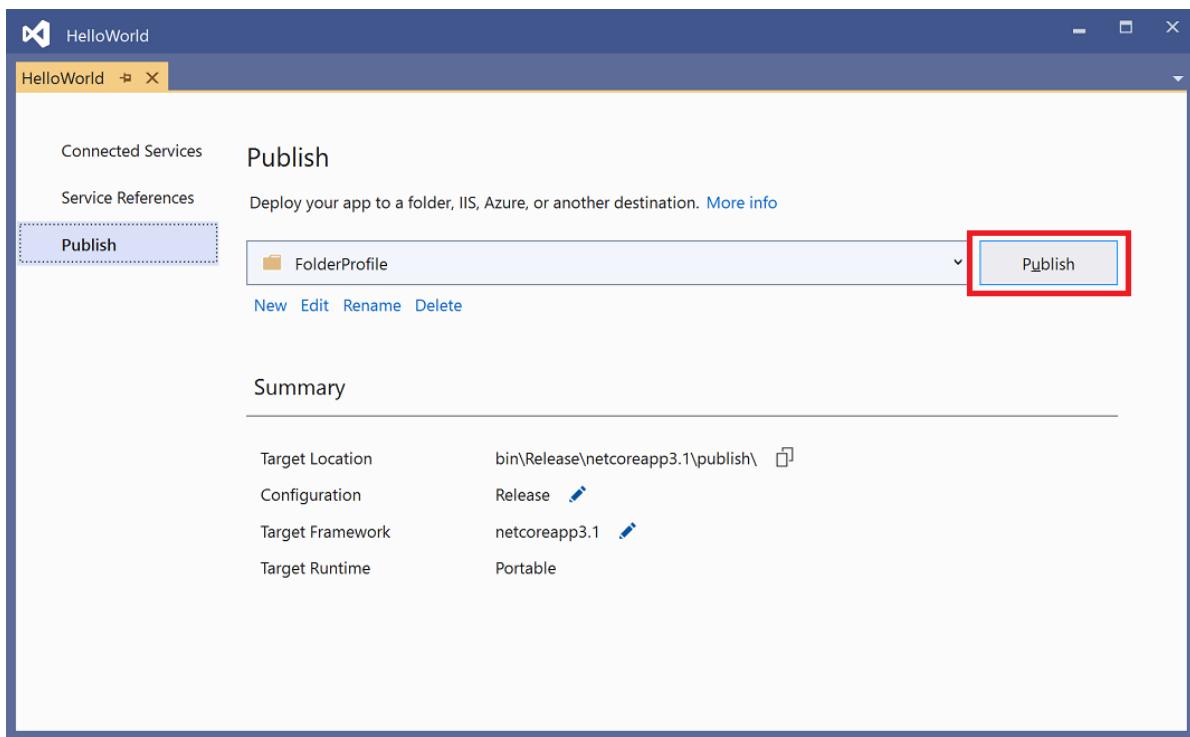
2. 右键单击“HelloWorld”项目（而不是 HelloWorld 解决方案），然后选择菜单中的“发布”。（还可以选择“生成”
主菜单中的“发布 HelloWorld”。）



3. 在“选择发布目标”页上，选择“文件夹”，然后选择“创建配置文件”。



4. 在“发布”页上，选择“发布”。



检查文件

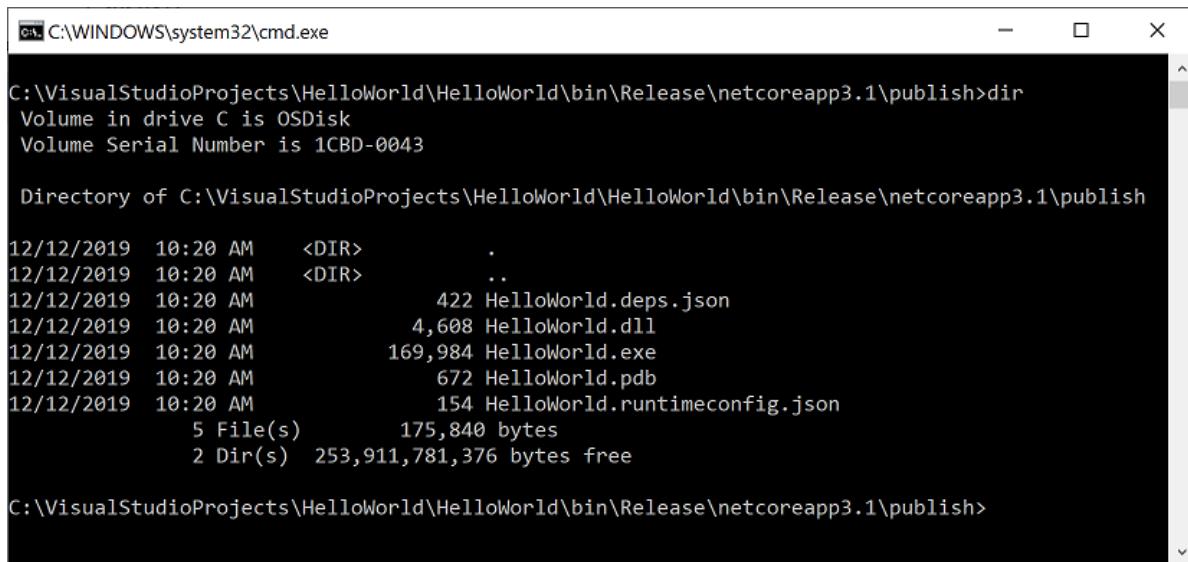
发布过程中会生成依赖于框架的部署，在此类部署中，若系统上安装了 .NET Core，已发布的应用程序可在 .NET Core 支持的任何平台上运行。用户可以通过双击可执行文件或从命令提示符发出 `dotnet HelloWorld.dll` 命令来运行发布的应用。

在下面的步骤中，查看由发布过程创建的文件。

1. 打开命令提示。

打开命令提示符的一种方法是，在 Windows 任务栏上的搜索框中输入“命令提示符”（或简写的“cmd”）。选择“命令提示符”桌面应用或按 Enter（如果已在搜索结果中选择）。

2. 导航到已发布的应用程序，它位于应用程序项目目录的 bin\Release\netcoreapp3.1\publish 子目录中。



```
C:\WINDOWS\system32\cmd.exe

C:\VisualStudioProjects\HelloWorld\HelloWorld\bin\Release\netcoreapp3.1\publish>dir
Volume in drive C is OSDisk
Volume Serial Number is 1CBD-0043

Directory of C:\VisualStudioProjects\HelloWorld\HelloWorld\bin\Release\netcoreapp3.1\publish

12/12/2019  10:20 AM    <DIR>      .
12/12/2019  10:20 AM    <DIR>      ..
12/12/2019  10:20 AM           422 HelloWorld.deps.json
12/12/2019  10:20 AM        4,608 HelloWorld.dll
12/12/2019  10:20 AM       169,984 HelloWorld.exe
12/12/2019  10:20 AM        672 HelloWorld.pdb
12/12/2019  10:20 AM       154 HelloWorld.runtimeconfig.json
               5 File(s)     175,840 bytes
               2 Dir(s)   253,911,781,376 bytes free

C:\VisualStudioProjects\HelloWorld\HelloWorld\bin\Release\netcoreapp3.1\publish>
```

如下图所示，已发布的输出包括以下文件：

- *HelloWorld.deps.json*

这是应用程序的运行时依赖项文件。它定义了运行应用程序所需的 .NET Core 组件和库（包括包含该应用程序的动态链接库）。有关详细信息，请参阅[运行时配置文件](#)。

- *HelloWorld.dll*

这是应用程序的[依赖于框架的部署](#)版本。若要执行此动态链接库，请在命令提示符处输入
`dotnet HelloWorld.dll`。

- *HelloWorld.exe*

这是应用程序的[依赖于框架的可执行文件](#)版本。若要运行该版本，请在命令提示符处输入
`HelloWorld.exe`。

- *HelloWorld.pdb*（对于部署是可选的）

这是调试符号文件。尽管应在需要调试应用程序的已发布版本时保存此文件，但无需将此文件与应用程序一起部署。

- *HelloWorld.runtimeconfig.json*

这是应用程序的运行时配置文件。它标识用于运行应用程序的 .NET Core 版本。还可向其添加配置选项。有关详细信息，请参阅[.NET Core 运行时配置设置](#)。

其他资源

- [.NET Core 应用程序部署](#)

在 Visual Studio 中生成 .NET Standard 库

2020/3/18 • [Edit Online](#)

类库定义的是可以由应用程序调用的类型和方法。借助面向 .NET Standard 2.0 的类库，任何支持相应 .NET Standard 版本的 .NET 实现都可以调用库。完成类库时，可以决定是要将其作为第三方组件进行分布，还是要将其作为与一个或多个应用程序捆绑在一起的组件进行添加。

NOTE

有关 .NET Standard 版本及其支持的平台列表，请参阅 [.NET Standard](#)。

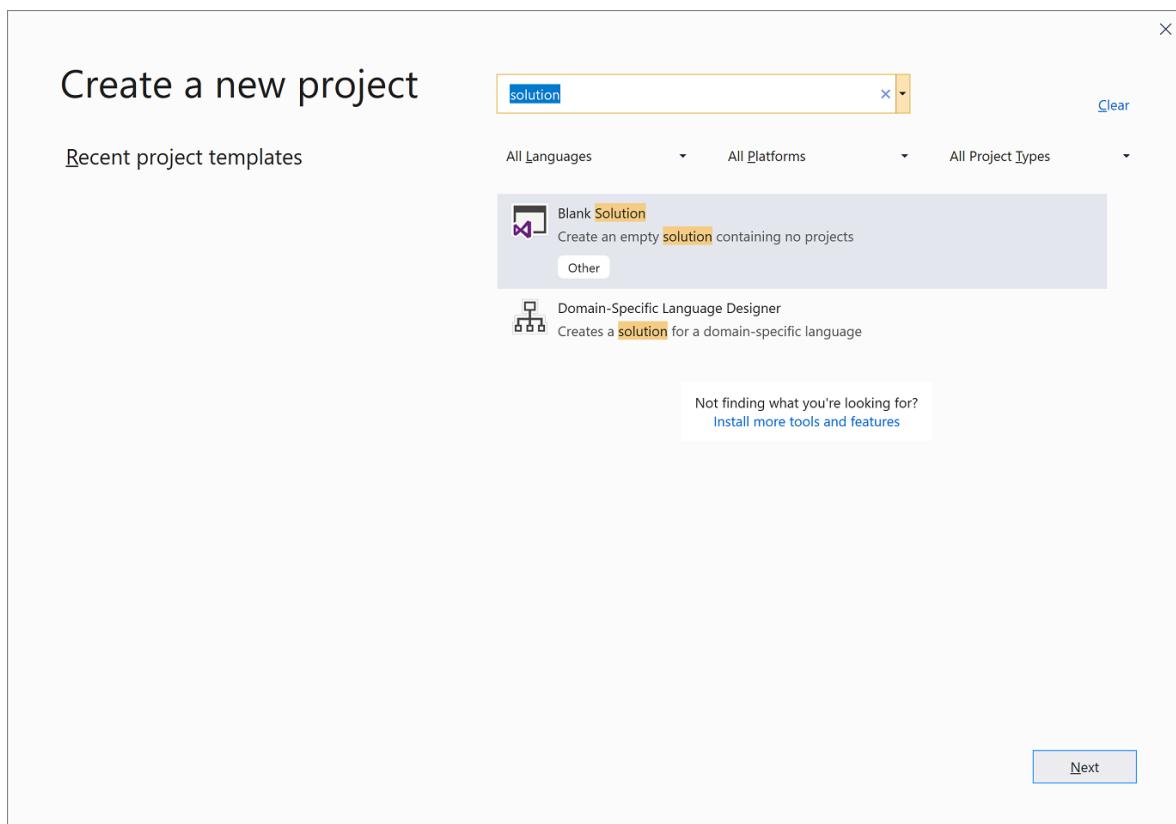
在本主题中，将创建包含一个字符串处理方法的简单实用工具库。我们将把它作为[扩展方法](#)进行实现，这样就可以把它作为 [String](#) 类成员进行调用。

创建 Visual Studio 解决方案

首先，创建一个空白解决方案来放置类库项目。Visual Studio 解决方案用作一个或多个项目的容器。如果继续学习本系列教程，你将向相同的解决方案添加其他相关项目。

创建空白解决方案：

1. 打开 Visual Studio。
2. 在“开始”窗口上，选择“创建新项目”。
3. 在“创建新项目”页面上，在搜索框中输入“解决方案”。选择“空白解决方案”模板，然后选择“下一步”。



4. 在“配置新项目”页面上，在“项目名称”框中输入“ClassLibraryProjects”。然后，选择“创建”。

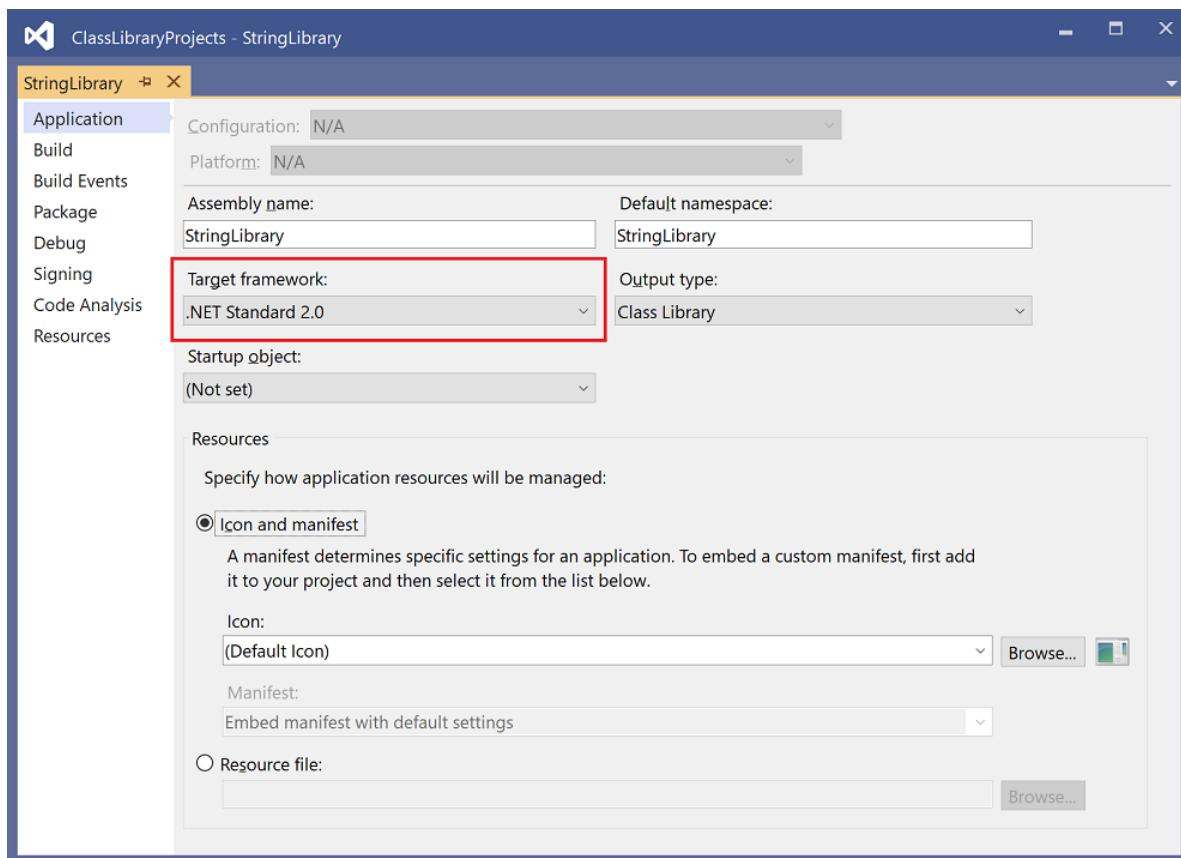
TIP

你还可以跳过此步骤，并让 Visual Studio 在下一步中创建项目时为你创建该解决方案。在“配置新项目”页上查找解决方案选项。

创建类库项目

- [C#](#)
- [Visual Basic](#)

1. 将名为“StringLibrary”的新 C# .NET Standard 类库项目添加到解决方案。
 - a. 在“解决方案资源管理器”中右键单击解决方案并选择“添加”>“新建项目”。
 - b. 在“创建新项目”页面上，在搜索框中输入“库”。从“语言”列表中选择“C#”，然后从“平台”列表中选择“所有平台”。选择“类库 (.NET Standard)”模板，然后选择“下一步”。
 - c. 在“配置新项目”页面，在“项目名称”框中输入“StringLibrary”。然后，选择“创建”。
2. 请检查以确保库面向 .NET Standard 的正确版本。右键单击“解决方案资源管理器”中的库项目，然后选择“属性”。“目标框架”文本框显示项目面向 .NET Standard 2.0。



3. 将代码窗口中的代码替换为以下代码，并保存文件：

```
using System;

namespace UtilityLibraries
{
    public static class StringLibrary
    {
        public static bool StartsWithUpper(this String str)
        {
            if (String.IsNullOrEmpty(str))
                return false;

            Char ch = str[0];
            return Char.IsUpper(ch);
        }
    }
}
```

类库 `UtilityLibraries.StringLibrary` 包含一个名为 `StartsWithUpper` 的方法。此方法会返回 `Boolean` 值，以指明当前字符串实例是否以大写字符开头。Unicode 标准会区分大小写字符。如果为大写字符，`Char.IsUpper(Char)` 方法返回 `true`。

4. 在菜单栏中，选择“生成” > “生成解决方案”。

此项目的编译应该没有错误。

后续步骤

已成功生成库。由于尚未调用库的任何方法，因此还不知道它能否按预期运行。开发库的下一步是测试库。

[创建单元测试项目](#)

在 Visual Studio 中使用 .NET Core 测试 .NET Standard 库

2020/3/18 • [Edit Online](#)

在 [在 Visual Studio 中生成 .NET Standard 库](#) 中，创建了一个简单的类库，用于向 `String` 类添加扩展方法。现在，将创建一个单元测试，用于确保此类库能够按预期运行。向在上一篇文章中创建的解决方案添加单元测试项目。

创建单元测试项目

若要创建单元测试项目，请执行以下操作：

1. 打开在 [ClassLibraryProjects 在 Visual Studio 中生成 .NET Standard 库一文中创建的 解决方案](#)。
2. 将名为“`StringLibraryTest`”的新单元测试项目添加到解决方案。
 - a. 在“解决方案资源管理器”中右键单击解决方案并选择“添加”>“新建项目”。
 - b. 在“添加新项目”页面，在搜索框中输入“`mstest`”。从“语言”列表中选择“C#”或“Visual Basic”，然后从“平台”列表中选择“所有平台”。选择“MsTest 测试项目(.NET Core)”模板，然后选择“下一步”。
 - c. 在“配置新项目”页面，在“项目名称”框中输入“`StringLibraryTest`”。然后选择“创建”。

NOTE

除了 MSTest 之外，还可以在 Visual Studio 中为 .NET Core 创建 xUnit 和 nUnit 测试项目。

3. 此时，Visual Studio 会创建项目，并在具有以下代码的代码窗口中打开类文件：

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace StringLibraryTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

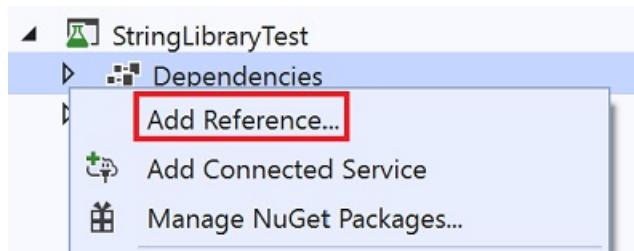
```
Imports Microsoft.VisualStudio.TestTools.UnitTesting

Namespace StringLibraryTest
    <TestClass>
    Public Class UnitTest1
        <TestMethod>
        Sub TestSub()

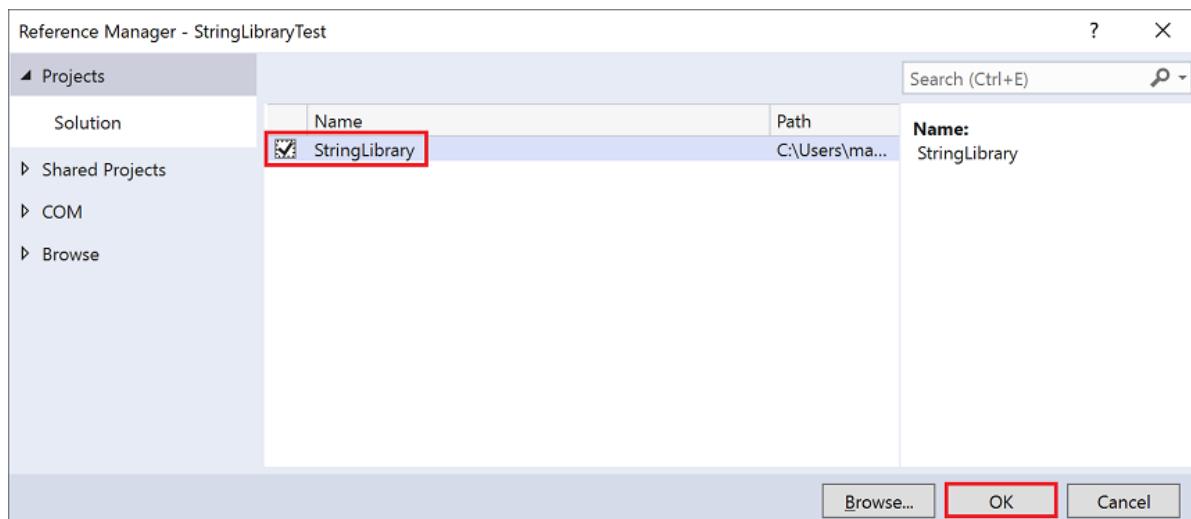
        End Sub
    End Class
End Namespace
```

单元测试模板创建的源代码负责执行以下操作：

- 它会导入 `Microsoft.VisualStudio.TestTools.UnitTesting` 命名空间，其中包含用于单元测试的类型。
 - 向 **类** 应用 `TestClass` `UnitTest1` 特性。测试类中标记有 `TestMethod` 属性的所有测试方法都会在单元测试运行时自动执行。
 - 它应用 `TestMethod` 属性，将 C# 中的 `TestMethod1` 或将 Visual Basic 中的 `TestSub` 定义为在单元测试运行时自动执行的测试方法。
4. 在“解决方案资源管理器”中，右键单击“StringLibraryTest”项目的“依赖项”节点，并从上下文菜单中选择“添加引用”。



5. 在“引用管理器”对话框中，展开“项目”节点，并选中“StringLibrary”旁边的框。添加对 `StringLibrary` 程序集的引用后，编译器可以查找 `StringLibrary` 方法。选择“确定”按钮。这会添加对类库项目 `StringLibrary` 的引用。



添加并运行单元测试方法

运行单元测试时，Visual Studio 执行单元测试类(对其应用了 `TestMethodAttribute` 特性的类)中标记有 `TestClassAttribute` 特性的所有方法。当第一次遇到测试不通过或测试方法中的所有测试均已成功通过时，测试方法终止。

最常见的测试调用 `Assert` 类的成员。许多断言方法至少包含两个参数，其中一个是预期的测试结果，另一个是实际的测试结果。下表显示了 `Assert` 类最常调用的一些方法：

方法	说明
<code>Assert.AreEqual</code>	验证两个值或对象是否相等。如果值或对象不相等，则断言失败。
<code>Assert.AreSame</code>	验证两个对象变量引用的是不是同一个对象。如果这些变量引用不同的对象，则断言失败。

<code>Assert.IsFalse</code>	验证条件是否为 <code>false</code> 。如果条件为 <code>true</code> ，则断言失败。
<code>Assert.IsNotNull</code>	验证对象是否不为 <code>null</code> 。如果对象为 <code>null</code> ，则断言失败。

还可以在测试方法中使用 `ThrowsException` 方法来指示它应引发的异常的类型。如果未引发指定异常，则测试不通过。

测试 `StringLibrary.StartsWithUpper` 方法时，需要提供许多以大写字符开头的字符串。在这种情况下，此方法应返回 `true`，以便可以调用 `IsTrue` 方法。同样，需要提供许多以非大写字符开头的字符串。在这种情况下，此方法应返回 `false`，以便可以调用 `IsFalse` 方法。

由于库方法处理的是字符串，因此还需要确保它能够成功处理空字符串 (`String.Empty`)（不含字符且 `Length` 为 0 的有效字符串）和 `null` 字符串（尚未初始化的字符串）。如果对 `StartsWithUpper` 实例调用 `String` 作为扩展方法，无法向其传递 `null` 字符串。不过，还可以直接将其作为静态方法进行调用，并向其传递一个 `String` 自变量。

将定义三个方法，每个方法都会对字符串数组中的各个元素反复调用它的 `Assert` 方法。由于测试方法在第一次遇到测试不通过时会立即失败，因此将调用方法重载，以便传递字符串来指明方法调用中使用的字符串值。

创建测试方法：

1. 将 `UnitTest1.cs` 或 `UnitTest1.vb` 代码窗口中的代码替换为以下代码：

```

using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using UtilityLibraries;

namespace StringLibraryTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestStartsWithUpper()
        {
            // Tests that we expect to return true.
            string[] words = { "Alphabet", "Zebra", "ABC", "Αθήνα", "Москва" };
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsTrue(result,
                    String.Format("Expected for '{0}': true; Actual: {1}",
                        word, result));
            }
        }

        [TestMethod]
        public void TestDoesNotStartWithUpper()
        {
            // Tests that we expect to return false.
            string[] words = { "alphabet", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                "1234", ".", ";", " " };
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsFalse(result,
                    String.Format("Expected for '{0}': false; Actual: {1}",
                        word, result));
            }
        }

        [TestMethod]
        public void DirectCallWithNullOrEmpty()
        {
            // Tests that we expect to return false.
            string[] words = { string.Empty, null };
            foreach (var word in words)
            {
                bool result = StringLibrary.StartsWithUpper(word);
                Assert.IsFalse(result,
                    String.Format("Expected for '{0}': false; Actual: {1}",
                        word == null ? "<null>" : word, result));
            }
        }
    }
}

```

```

Imports Microsoft.VisualStudio.TestTools.UnitTesting
Imports UtilityLibraries

Namespace StringLibraryTest
    <TestClass>
    Public Class UnitTest1
        <TestMethod>
        Public Sub TestStartsWithUpper()
            ' Tests that we expect to return true.
            Dim words() As String = {"Alphabet", "Zebra", "ABC", "Αθήνα", "Москва"}
            For Each word In words
                Dim result As Boolean = word.StartsWithUpper()
                Assert.IsTrue(result,
                    $"Expected for '{word}': true; Actual: {result}")
            Next
        End Sub

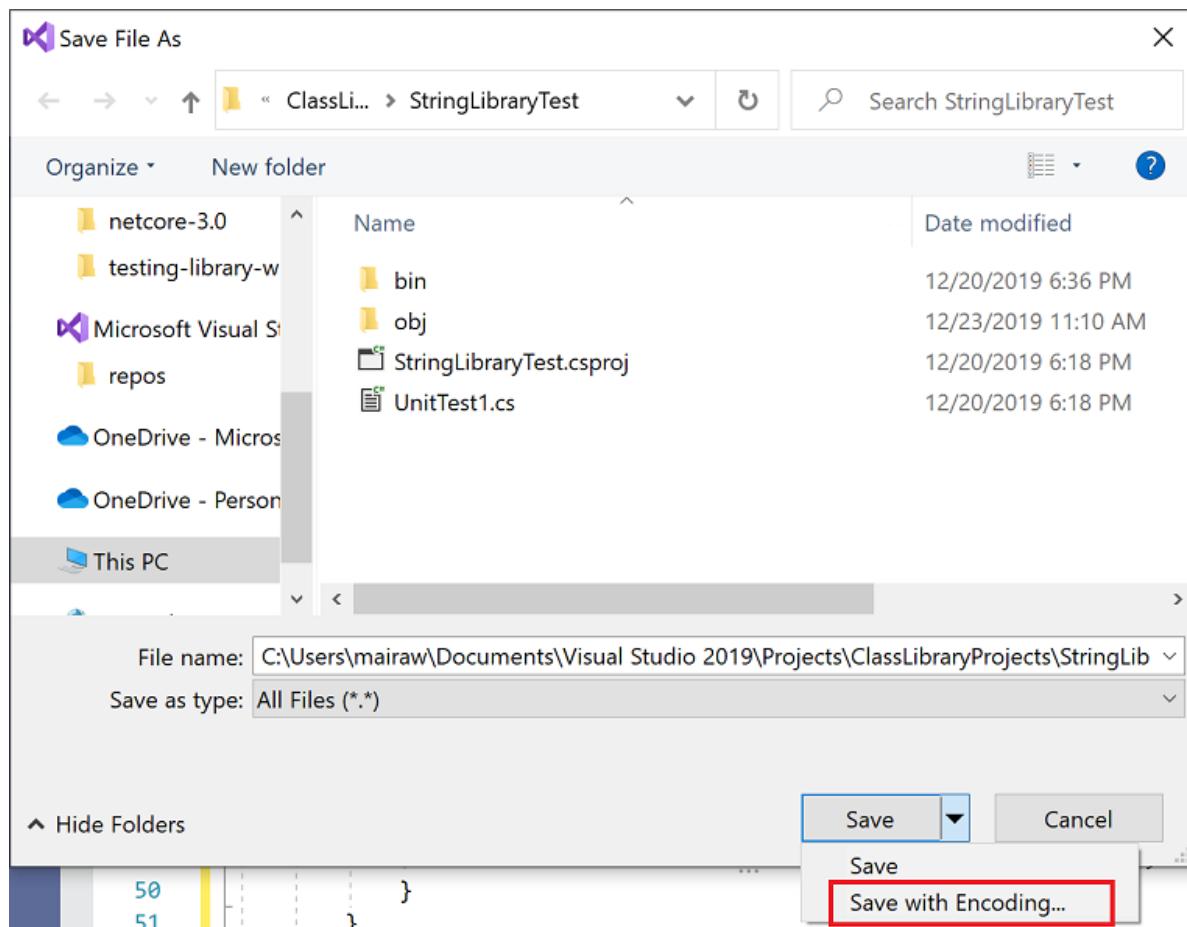
        <TestMethod>
        Public Sub TestDoesNotStartWithUpper()
            ' Tests that we expect to return false.
            Dim words() As String = {"alphabet", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                "1234", ".", ";", " "}
            For Each word In words
                Dim result As Boolean = word.StartsWithUpper()
                Assert.IsFalse(result,
                    $"Expected for '{word}': false; Actual: {result}")
            Next
        End Sub

        <TestMethod>
        Public Sub DirectCallWithNullOrEmpty()
            ' Tests that we expect to return false.
            Dim words() As String = {String.Empty, Nothing}
            For Each word In words
                Dim result As Boolean = StringLibrary.StartsWithUpper(word)
                Assert.IsFalse(result,
                    $"Expected for '{If(word Is Nothing, "<null>", word)}': false; Actual:
{result}")
            Next
        End Sub
    End Class
End Namespace

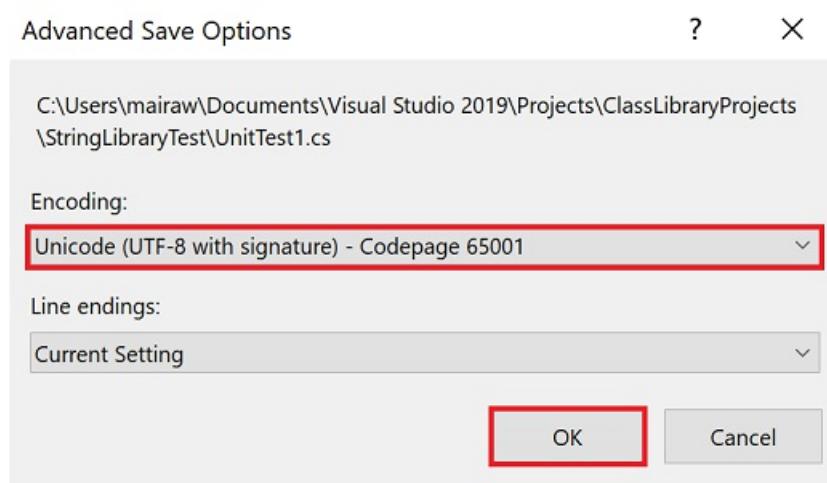
```

`TestStartsWithUpper` 方法中的大写字符的测试包括希腊文大写字母 alpha (U+0391) 和西里尔文大写字母 EM (U+041C)。`TestDoesNotStartWithUpper` 方法中的小写字符的测试包括希腊文小写字母 alpha (U+03B1) 和西里尔文小写字母 Ghe (U+0433)。

- 在菜单栏上，选择“文件”>“将 UnitTest1.cs 另存为”或“文件”>“将 UnitTest1.vb 另存为”。在“文件另存为”对话框中，选择“保存”按钮旁边的箭头，然后选择“保存时使用编码”。

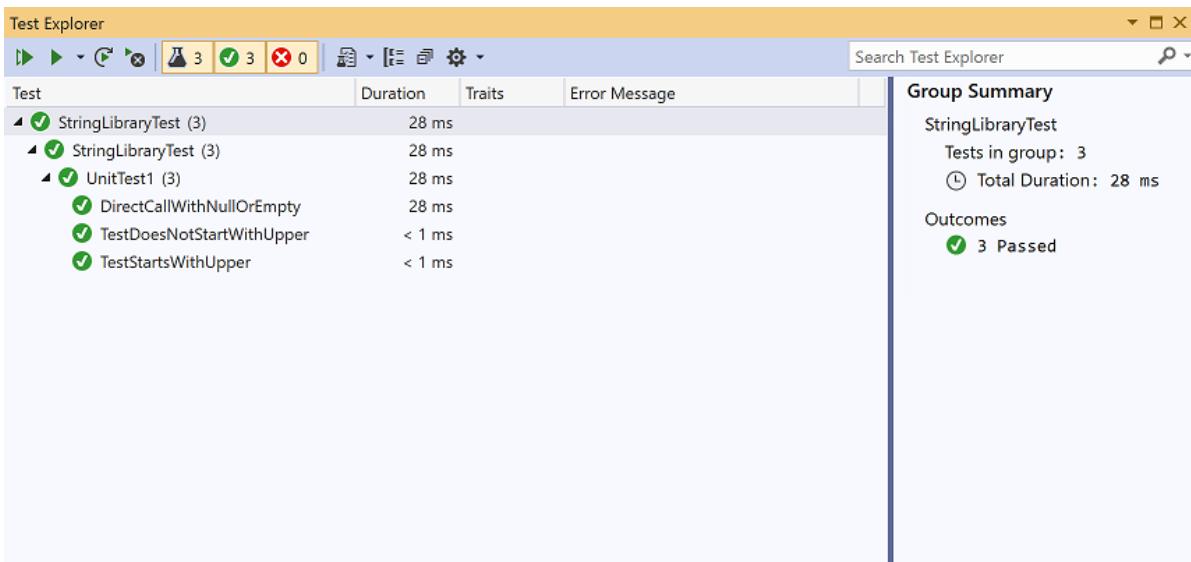


3. 在“确认另存为”对话框中，选择“是”按钮，保存文件。
4. 在“高级保存选项”对话框的“编码”下拉列表中，选择“Unicode (UTF-8 带签名) - 代码页 65001”，然后选择“确定”。



如果无法将源代码保存为 UTF8 编码文件, Visual Studio 可能会将其另存为 ASCII 文件。在这种情况下, 运行时将无法准确解码 ASCII 范围以外的 UTF8 字符, 且测试结果也会不正确。

5. 在菜单栏上, 选择“测试” > “运行” > “所有测试”。此时, “测试资源管理器”窗口打开并显示测试已成功运行。“通过的测试”部分列出了三个测试, “摘要”部分报告了测试运行结果。



处理测试失败

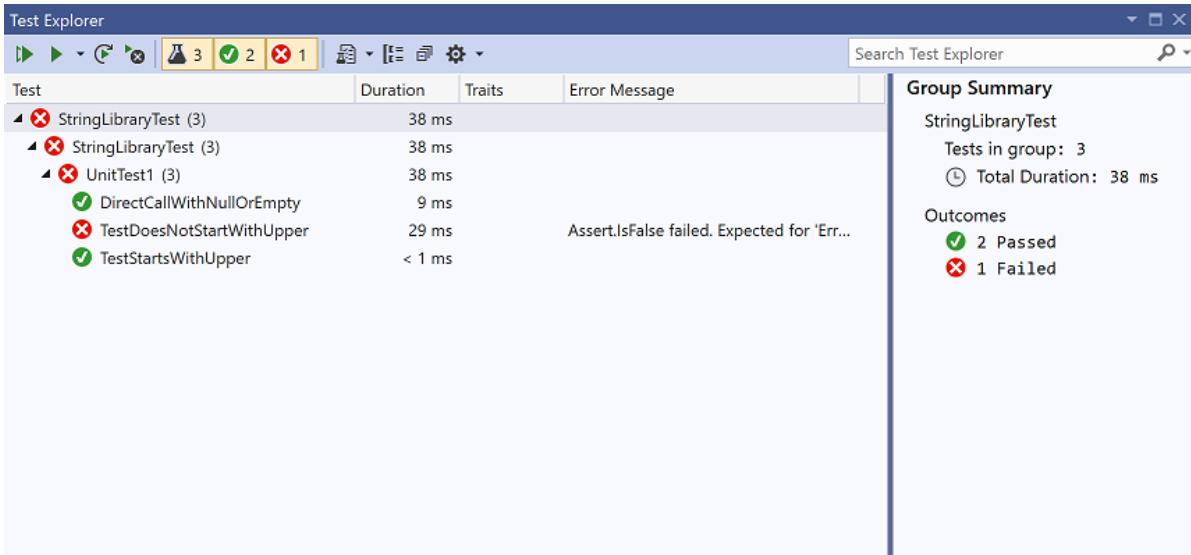
由于运行的测试均通过，因此需进行少量改动，以使其中一个测试方法失败：

1. 通过修改 `words` 方法中的 `TestDoesNotStartWithUpper` 数组来包含字符串“Error”。由于 Visual Studio 将在生成运行测试的解决方案时自动保存打开的文件，因此无需手动保存。

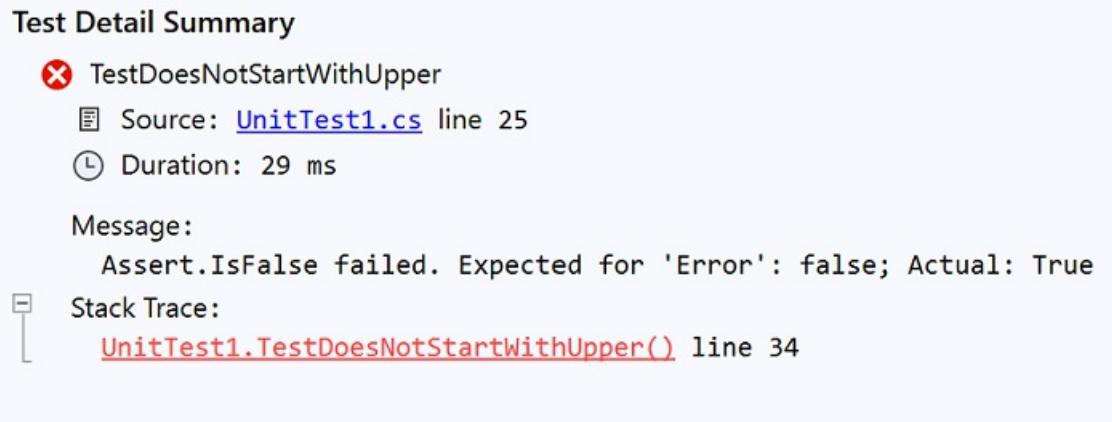
```
string[] words = { "alphabet", "Error", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                   "1234", ".", ";" };
```

```
Dim words() As String = { "alphabet", "Error", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                           "1234", ".", ";" }
```

2. 从菜单栏中选择“测试”>“运行”>“所有测试”，运行测试。“测试资源管理器”窗口指示有两个测试成功，还有一个失败。



3. 选择失败的测试，`TestDoesNotStartWithUpper`。测试资源管理器窗口显示断言生成的消息：“`Assert.IsFalse` 失败。`'Error'` 应返回 `false`; 实际返回 `True`”。由于此次失败，数组中“Error”之后的所有字符串都未进行测试。



4. 撤消在步骤 1 中执行的修改并删除字符串“Error”。重新运行测试，测试将通过。

测试库的发行版本

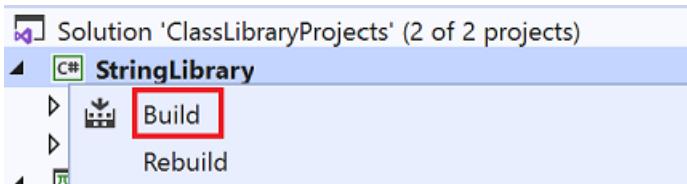
现已测试库的调试版本。至此，测试已全部通过，且已充分测试库，应对库的发布版本再运行一次这些测试。许多因素（包括编译器优化）有时可能会导致调试版本和发行版本出现行为差异。

若要测试发行版本，请执行以下操作：

1. 在 Visual Studio 工具栏中，将生成配置从“调试”更改为“发行”。



2. 在“解决方案资源管理器”中，右键单击“StringLibrary”项目，从上下文菜单中选择“生成”，重新编译库。



3. 从菜单栏中选择“测试”>“运行”>“所有测试”，运行单元测试。测试通过。

至此，已完成对库的测试，下一步就是使其可供调用方使用。可以将类库与一个或多个应用程序捆绑在一起，也可以 NuGet 包的形式分发类库。有关详细信息，请参阅[使用 .NET Standard 类库](#)。

另请参阅

- [单元测试基础知识 - Visual Studio](#)

在 Visual Studio 中使用 .NET Standard 库

2020/3/18 • [Edit Online](#)

创建 .NET Standard 类库、测试并生成库的发行版本后，下一步就是使其可供调用方使用。可以通过以下两种方式执行此操作：

- 如果库将供一个解决方案使用（例如，如果它是一个大型应用程序中的组件），可以将其以项目的形式添加到解决方案中。
- 如果类库将公开发布，你可以以 NuGet 包的形式分发。

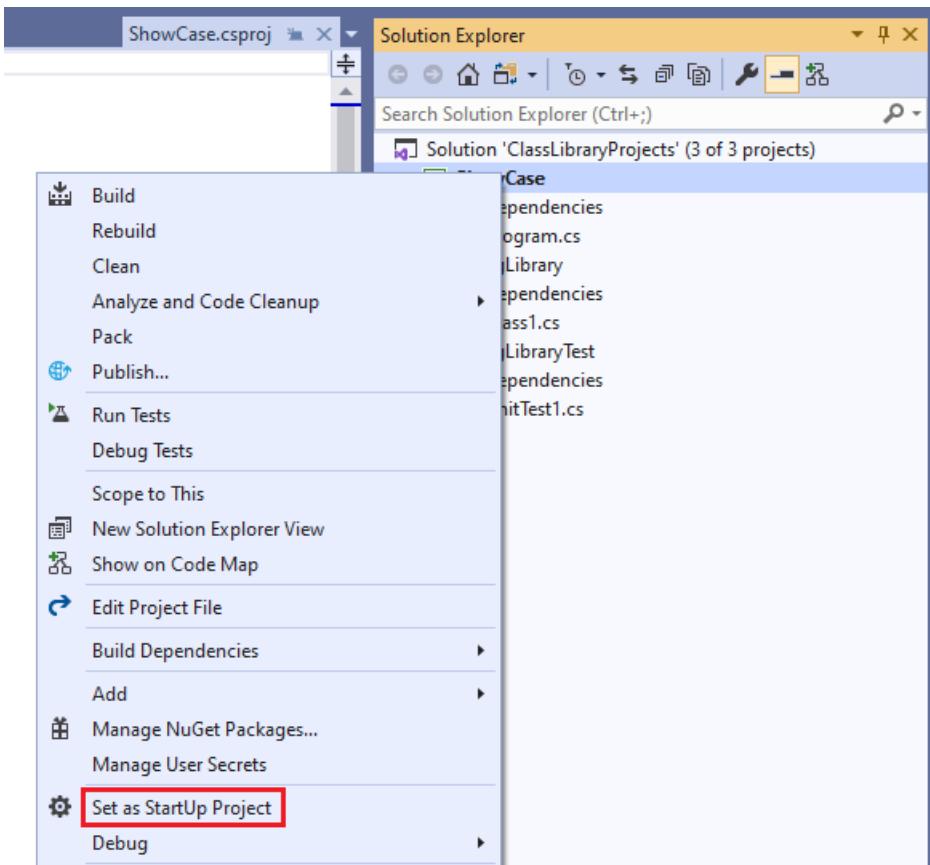
在本教程中，你将了解：

- 向解决方案中添加引用 .NET Standard 库项目的控制台应用。
- 创建包含 .NET Standard 库项目的 NuGet 包。

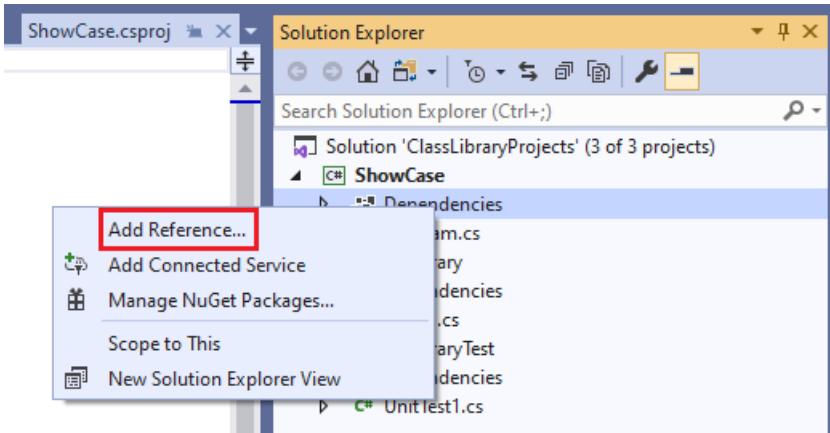
向解决方案添加控制台应用

就像[在 Visual Studio 中测试 .NET Standard 库](#)所述的将单元测试和类库添加到同一解决方案中一样，可以将应用程序添加到同一解决方案中。例如，可在控制台应用程序中使用类库，此应用程序将提示用户输入字符串，并报告第一个字符是否为大写：

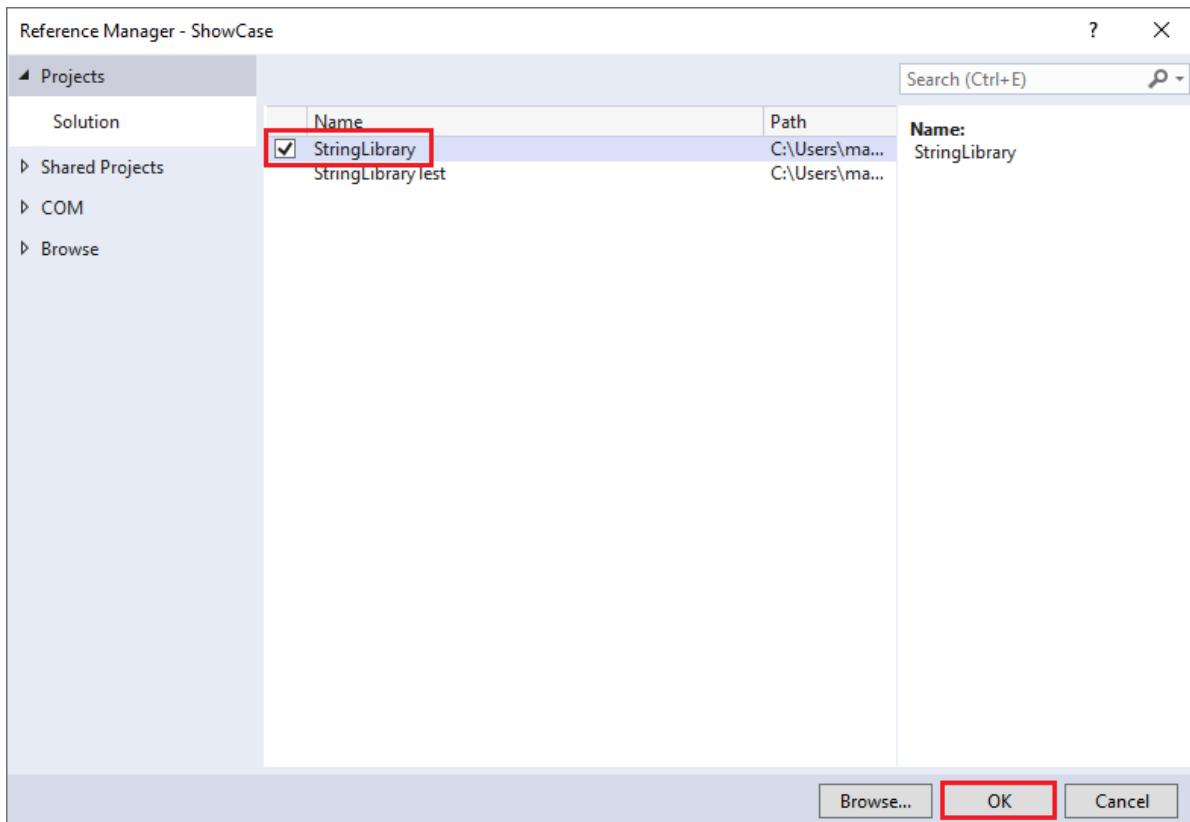
- 打开在 `ClassLibraryProjects` 在 Visual Studio 中生成 .NET Standard 库[一文中创建的](#) 解决方案。
- 将名为“ShowCase”的新 .NET Core 控制台应用程序添加到解决方案。
 - 在“解决方案资源管理器”中右键单击解决方案并选择“添加” > “新建项目”。
 - 在“创建新项目”页面，在搜索框中输入“控制台”。从“语言”列表中选择“C#”或“Visual Basic”，然后从“平台”列表中选择“所有平台”。选择“控制台应用 (.NET Core)”模板，然后选择“下一步”。
 - 在“配置新项目”页面，在“项目名称”框中输入“ShowCase”。然后，选择“创建”。
- 在“解决方案资源管理器”中，右键单击“ShowCase”项目，在上下文菜单中选择“设为启动项目”。



4. 项目一开始无权访问类库。若要允许项目调用类库中的方法，可以创建对该类库的引用。在“解决方案资源管理器”中，右键单击 ShowCase 项目的“依赖项”节点，并选择“添加引用”。



5. 在“引用管理器”对话框中，选择类库项目“StringLibrary”，然后选择“确定”按钮。



6. 在“Program.cs”或“Program.vb”文件的代码窗口中，将所有代码替换为以下代码：

```
using System;
using UtilityLibraries;

class Program
{
    static void Main(string[] args)
    {
        int row = 0;

        do
        {
            if (row == 0 || row >= 25)
                ResetConsole();

            string input = Console.ReadLine();
            if (String.IsNullOrEmpty(input)) break;
            Console.WriteLine($"Input: {input} {"Begins with uppercase? ",30}: " +
                $"{(input.StartsWithUpper() ? "Yes" : "No")}\n");
            row += 3;
        } while (true);
        return;

        // Declare a ResetConsole local method
        void ResetConsole()
        {
            if (row > 0) {
                Console.WriteLine("Press any key to continue...");
                Console.ReadKey();
            }
            Console.Clear();
            Console.WriteLine("\nPress <Enter> only to exit; otherwise, enter a string and press
<Enter>:\n");
            row = 3;
        }
    }
}
```

```

Imports UtilityLibraries

Module Program
    Dim row As Integer = 0

    Sub Main()
        Do
            If row = 0 OrElse row >= 25 Then ResetConsole()

            Dim input As String = Console.ReadLine()
            If String.IsNullOrEmpty(input) Then Return

            Console.WriteLine($"Input: {input} {"Begins with uppercase? ",30}: " +
                $"{If(input.StartsWithUpper(), "Yes", "No")}{vbCrLf}")
            row += 3
        Loop While True
    End Sub

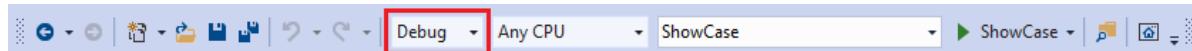
    Private Sub ResetConsole()
        If row > 0 Then
            Console.WriteLine("Press any key to continue...")
            Console.ReadKey()
        End If
        Console.Clear()
        Console.WriteLine("{0}Press <Enter> only to exit; otherwise, enter a string and press <Enter>:
{0}",
                         vbCrLf)
        row = 3
    End Sub
End Module

```

该代码使用 `row` 变量来维护写入到控制台窗口的数据行数计数。如果大于或等于 25，该代码将清除控制台窗口，并向用户显示一条消息。

该程序会提示用户输入字符串。它会指明字符串是否以大写字符开头。如果用户没有输入字符串就按 Enter 键，那么应用程序会终止，控制台窗口会关闭。

- 必要时，将工具栏更改为编译项目的“调试” `ShowCase` 版本。选择“ShowCase”按钮上的绿色箭头，编译并运行程序。



可以按照[使用 Visual Studio 调试 C# 或 Visual Basic .NET Core Hello World 应用程序](#)和[使用 Visual Studio 发布 .NET Core Hello World 应用程序](#)中的步骤操作，调试并发布使用此库的应用程序。

创建 NuGet 包

可采用 NuGet 包的形式发布类库，让类库可供更多调用方使用。Visual Studio 不支持创建 NuGet 包。若要创建一个，需要使用 .NET Core CLI 命令：

- 打开控制台窗口。

例如，在 Windows 任务栏的搜索框中输入“命令提示符”。选择“命令提示符”桌面应用或按 Enter（如果已在搜索结果中选择）。

- 转到类库的项目目录。此目录包含源代码和项目文件 `StringLibrary.csproj` 或 `StringLibrary.vbproj`。
- 运行 `dotnet pack` 命令，以生成带 `.nupkg` 扩展的包：

```
dotnet pack --no-build
```

TIP

如果路径中没有包含 dotnet.exe 的目录，可以通过在控制台窗口中输入 `where dotnet.exe` 来找到它的位置。

若要详细了解如何创建 NuGet 包，请参阅[如何使用 .NET Core CLI 创建 NuGet 包](#)。

使用 Visual Studio for Mac 在 macOS 上构建完整的 .NET Core 解决方案

2020/3/18 • [Edit Online](#)

Visual Studio for Mac 提供用于开发 .NET Core 应用程序的功能全面的集成开发环境 (IDE)。本文演示了构建包含可重用的库和单元测试的 .NET Core 解决方案。

本教程介绍了如何创建接受来自用户的搜索词和文本字符串、使用类库中的方法计算字符串中出现的搜索词的次数，并将结果返回给用户的应用程序。该解决方案还包括类库的单元测试（作为单元测试概念的介绍）。如果希望使用完整的示例学习该教程，请下载[示例解决方案](#)。有关下载说明，请参阅[示例和教程](#)。

NOTE

你的反馈非常有价值。有两种方法可以向开发团队提供有关 Visual Studio for Mac 的反馈：

- 在 Visual Studio for Mac 中，从菜单选择“帮助”>“报告问题”，或从欢迎屏幕中选择“报告问题”，将打开一个窗口，以供填写 bug 报告。可在[开发人员社区](#)门户中跟踪自己的反馈。
- 若要提出建议，从菜单中选择“帮助”>“提供建议”，或从欢迎屏幕中选择“提供建议”，转到[Visual Studio for Mac 开发人员社区网页](#)。

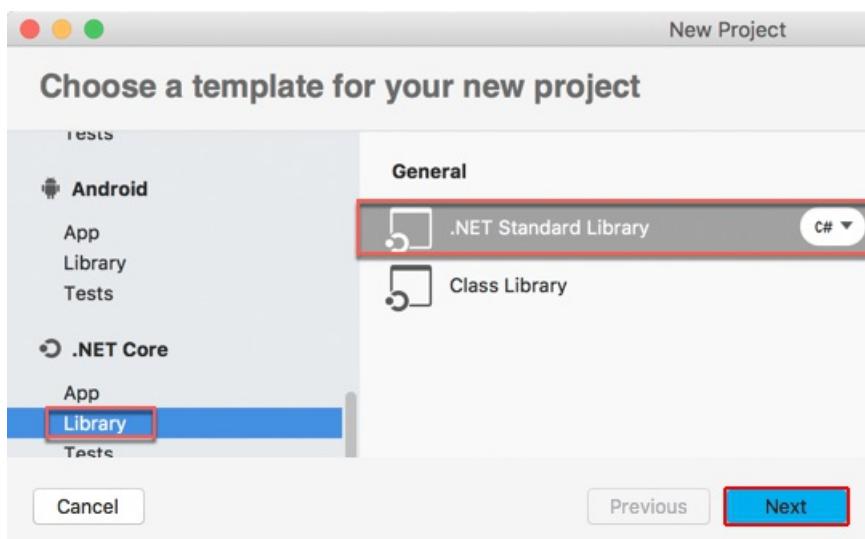
先决条件

- [.NET Core SDK 3.1 或更高版本](#)
- [Visual Studio 2019 for Mac](#)

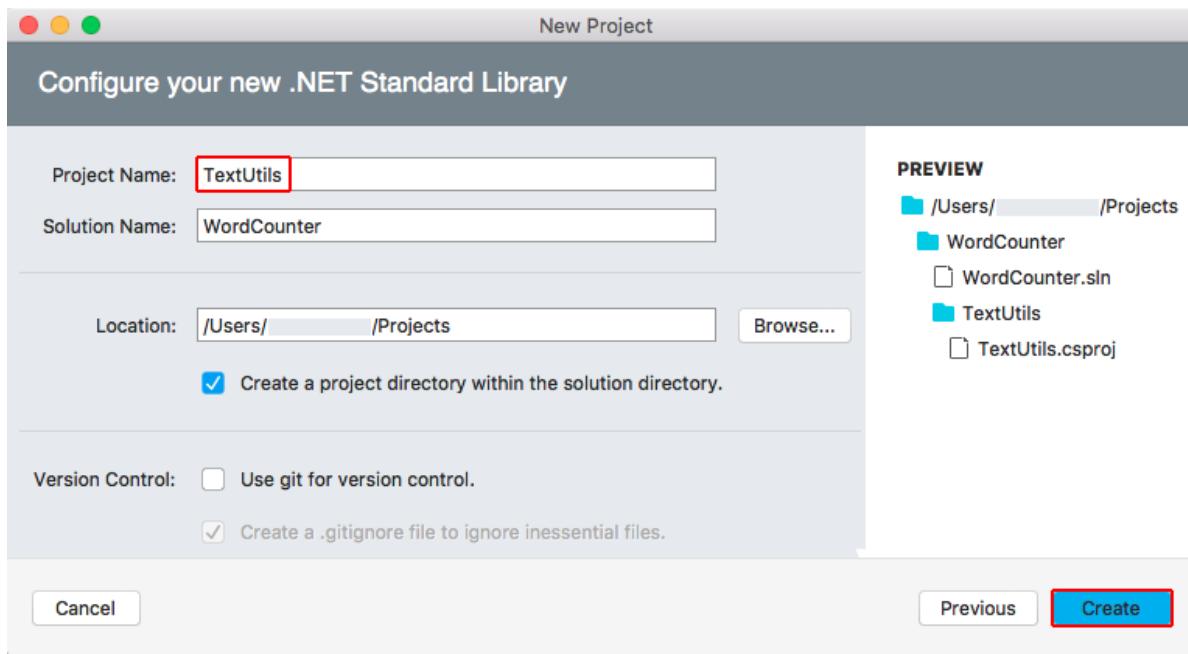
若要了解有关先决条件的详细信息，请参阅[.NET Core 依赖项和要求](#)。有关 Visual Studio 2019 for Mac 完整系统要求的信息，请参阅[Visual Studio 2019 for Mac 产品系列系统要求](#)。

生成库

- 在“开始”窗口，选择“新建项目”。在“.NET Core”节点下的“新建项目”对话框中，选择“.NET 标准库”模板。此命令将创建一个 .NET Standard 库，该库面向 .NET Core 及支持 .NET Standard 版本 2.1 的任何其他 .NET 实现。如果安装了多个版本的 .NET Core SDK，则可以为库选择不同版本的 .NET Standard。选择“.NET Standard 2.1”，然后选择“下一步”。



2. 将项目命名为“TextUtils”（“Text Utilities”的短名称），将解决方案命名为“WordCounter”。使“在解决方案目录中创建项目目录”保持选中状态。选择“创建”。



3. 在“解决方案”边栏中，展开 `TextUtils` 节点以显示模板提供的类文件 `Class1.cs`。按住 Ctrl 并单击该文件，从上下文菜单中选择“重命名”，然后将该文件重命名为 `WordCount.cs`。打开文件并将内容替换为以下代码：

```
using System;
using System.Linq;

namespace TextUtils
{
    public static class WordCount
    {
        public static int GetWordCount(string searchWord, string inputString)
        {
            // Null check these variables and determine if they have values.
            if (string.IsNullOrEmpty(searchWord) || string.IsNullOrEmpty(inputString))
            {
                return 0;
            }

            // Convert the string into an array of words.
            var source = inputString.Split(new char[] { '.', '?', '!', ' ', ';' }, StringSplitOptions.RemoveEmptyEntries);

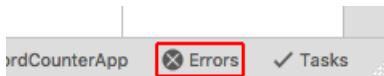
            // Create the query. Use ToLowerInvariant to match uppercase/lowercase strings.
            var matchQuery = from word in source
                            where word.ToLowerInvariant() == searchWord.ToLowerInvariant()
                            select word;

            // Count the matches, which executes the query. Return the result.
            return matchQuery.Count();
        }
    }
}
```

4. 通过使用以下三种不同的方法之一保存文件：使用键盘快捷方式 ⌘+s，从菜单中选择“文件”>“保存”，或按住 Ctrl 并单击文件的选项卡，并从上下文菜单中选择“保存”。下图显示 IDE 窗口：

The screenshot shows the Visual Studio IDE interface. In the center, there is a code editor window titled "WordCount.cs" containing C# code for a class named "WordCount". The code uses LINQ to count words in a string. On the left, there is a "Document Outline" and a "Unit Tests" sidebar. On the right, the "Solution Explorer" shows a project named "WordCounter" with a "TextUtils" folder and a "Dependencies" folder. The file "WordCount.cs" is highlighted with a red border. At the bottom of the screen, there is a navigation bar with tabs for "Test Results", "Errors", "Tasks", and "Package Console".

5. 在 IDE 窗口底部边距处选择“错误”，打开“错误”面板。选择“生成输出”按钮。



6. 从菜单中选择“生成” > “生成所有”。

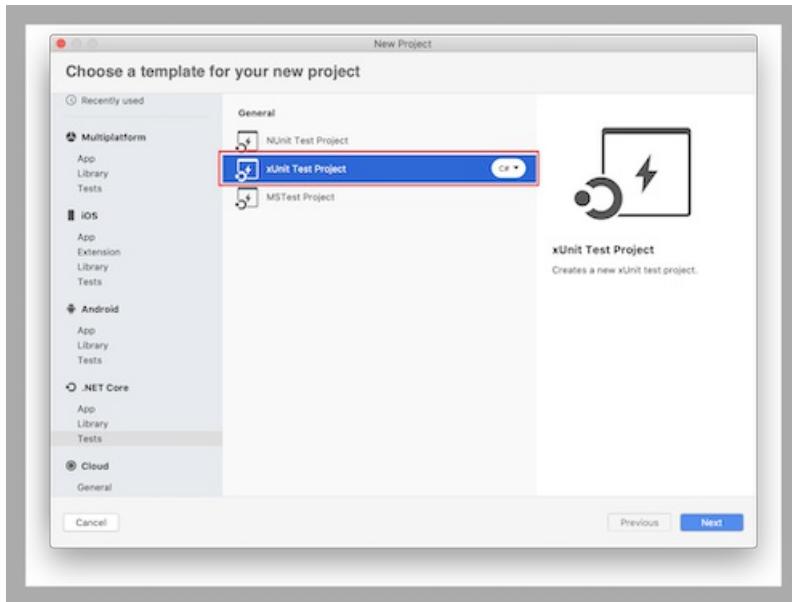
生成解决方案。生成输出面板显示生成成功。

The screenshot shows the "Build" output window. It displays the following text:
Build succeeded.
0 Warning(s)
0 Error(s)
Time Elapsed 00:00:05.13
----- Done -----
Build successful.

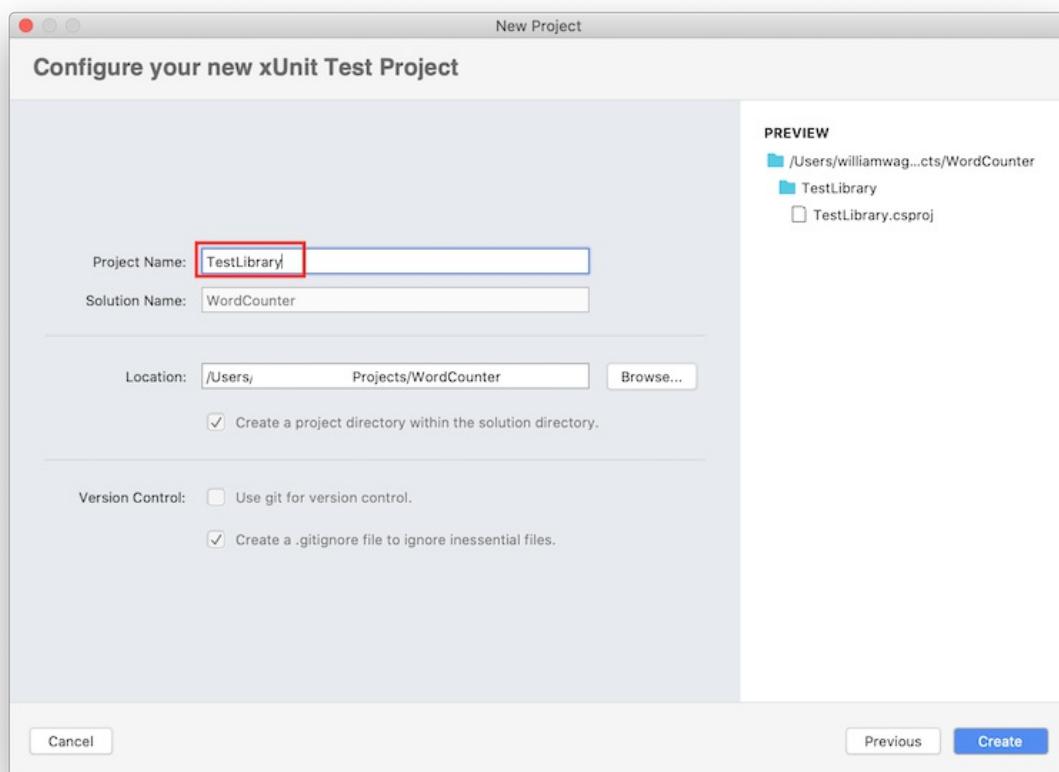
创建测试项目

单元测试在开发和发布期间提供自动化的软件测试。本教程中使用的测试框架是 [xUnit\(版本 2.4.0 或更高版本\)](#)，使用下列步骤将 xUnit 测试项目添加到解决方案时将自动安装此框架：

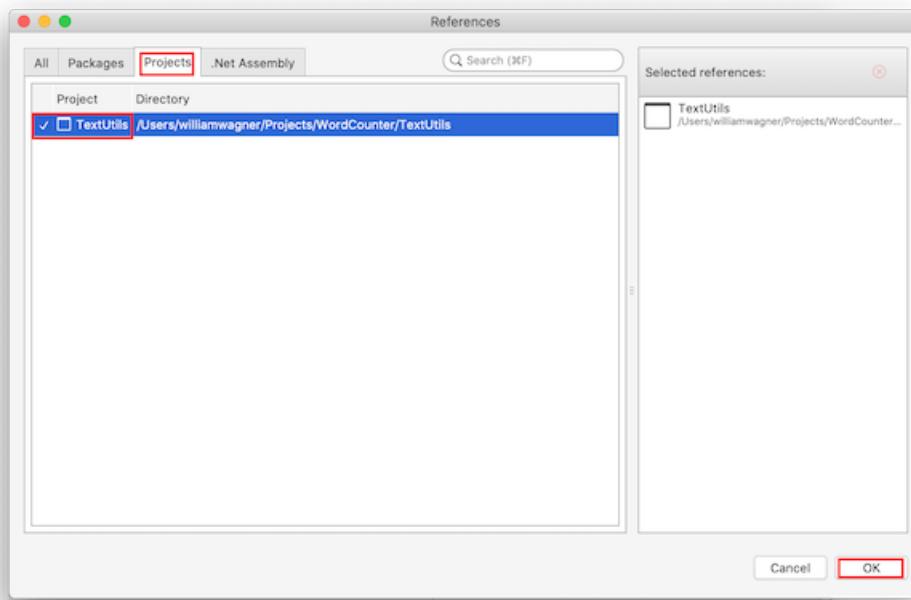
1. 在“解决方案”边栏中，按住 Ctrl 并单击 **WordCounter** 解决方案并选择“添加” > “添加新项目”。
2. 在“新建项目”对话框中，从“.NET Core”节点中选择“测试”。在“下一步”后，选择“xUnit 测试项目”。



3. 如果有多个版本的.NET Core SDK, 则需要选择此项目的版本。选择“.NET Core 3.1”。将新项目命名为“TestLibrary”, 然后选择“创建”。



4. 若要使测试库使用 `WordCount` 类, 请将引用添加到 `TextUtils` 项目中。在“解决方案”边栏中, 按住 Ctrl 并单击“TestLibrary”下的“依赖项”。从上下文菜单中选择“编辑引用”。
5. 在“编辑引用”对话框中, 选择“项目”选项卡上的“TextUtils”项目。选择“确定”。



6. 在 **TestLibrary** 项目中，将 *UnitTest1.cs* 文件重命名为 *TextUtilsTests.cs*。

7. 打开该文件，并将代码替换为以下代码：

```
using Xunit;
using TextUtils;
using System.Diagnostics;

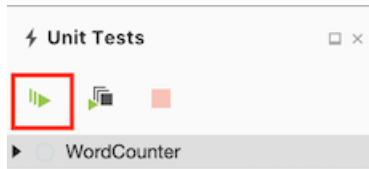
namespace TestLibrary
{
    public class TextUtils_GetWordCountShould
    {
        [Fact]
        public void IgnoreCasing()
        {
            var wordCount = WordCount.GetWordCount("Jack", "Jack jack");

            Assert.NotEqual(2, wordCount);
        }
    }
}
```

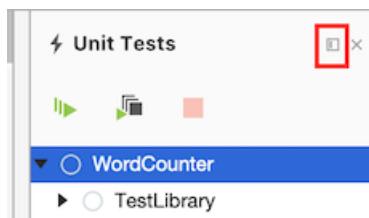
下图显示了就地使用单元测试代码的 IDE。请注意 `Assert.NotEqual` 语句。

请务必使新的测试失败一次，以确定其测试逻辑正确无误。该方法使用“Jack”和“jack”（大写和小写）传递名称“Jack”（大写）和字符串。如果 `GetWordCount` 方法运行正常，则返回搜索词的两个实例的计数。为了有意进行失败测试，首先实现测试断言，即搜索词“Jack”的两个实例不是由 `GetWordCount` 方法返回的。继续执行下一步骤，有意使测试失败。

8. 打开屏幕右侧的“单元测试”面板。从菜单中选择“查看” > “测试”。

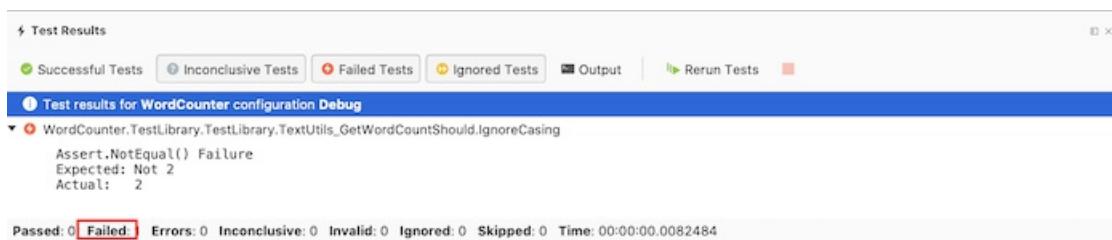


9. 单击“停靠”图标使此面板保持打开状态。（在下图中突出显示。）



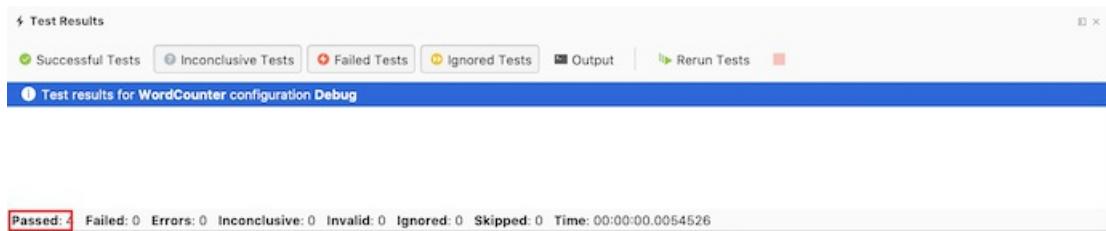
10. 单击“全部运行”按钮。

测试失败，这是正确的结果。测试方法断言不会从提供给 `GetWordCount` 的方法的字符串“Jack jack”中返回 `inputString` “Jack”的两个实例。因为已在 `GetWordCount` 方法中对单词的大小写进行了分解，所以返回了两个实例。2 不等于2 的断言失败。这是正确的结果，且测试的逻辑良好。



11. 通过将 `Assert.NotEqual` 更改为 `Assert.Equal` 来修改 `IgnoreCasing` 测试方法。使用键盘快捷方式 `Ctrl+s` 保存该文件，从菜单选择“文件”>“保存”，或按住 `Ctrl` 并单击文件的选项卡，并从上下文菜单中选择“保存”。

`searchWord` "Jack" 应返回两个实例，并且 `inputString` "Jack jack" 传递到 `GetWordCount`。通过单击“单元测试”面板中的“运行测试”按钮或屏幕底部的“测试结果”面板中的“重新运行测试”按钮重新运行测试。测试通过。在字符串"Jack jack"(忽略大小写)中有"Jack"的两个实例，且测试断言为 `true`。

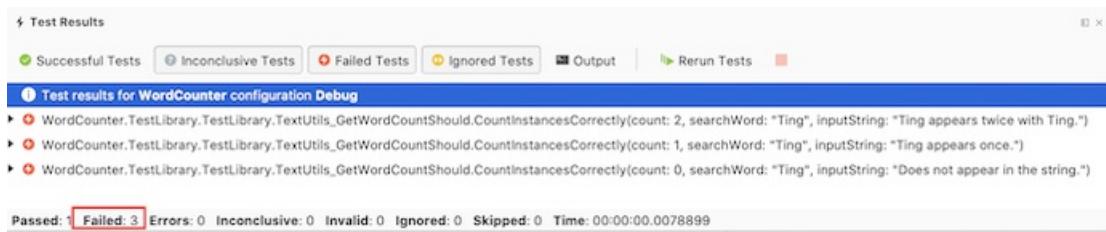


12. 使用 `Fact` 测试单个返回值仅仅是借助单元测试可以实现的功能的开始。另一个功能强大的技术是允许你使用 `Theory` 立即测试多个值。将以下方法添加到你的 `TextUtils_GetWordCountShould` 类。添加该方法后，在类中有两个方法：

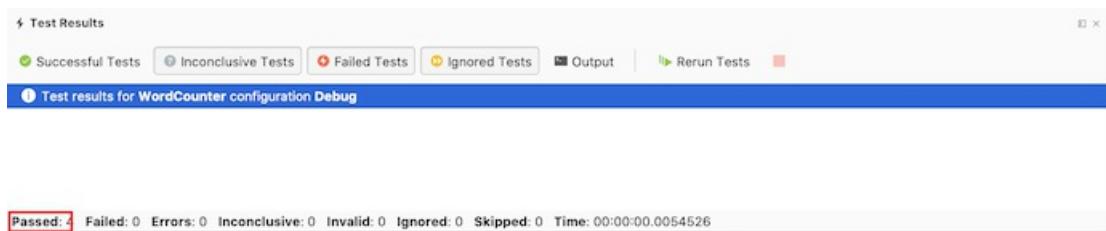
```
[Theory]
[InlineData(0, "Ting", "Does not appear in the string.")]
[InlineData(1, "Ting", "Ting appears once.")]
[InlineData(2, "Ting", "Ting appears twice with Ting.")]
public void CountInstancesCorrectly(int count,
                                     string searchWord,
                                     string inputString)
{
    Assert.NotEqual(count, WordCount.GetWordCount(searchWord,
                                                    inputString));
}
```

`CountInstancesCorrectly` 检查 `GetWordCount` 方法计数是否正确。`InlineData` 提供计数、搜索词和要检查的输入字符串。测试方法为数据的每行运行一次。请再次注意，使用 `Assert.NotEqual` 首先声明失败，即使知道数据中的计数是正确的，且这些值与 `GetWordCount` 方法所返回的计数相匹配。有意执行测试失败的步骤起初看起来像是浪费时间，但是首先通过失败测试检查测试的逻辑对于测试逻辑而言，是一项重要检查。如果在预期失败时找到一种可通过的测试方法，则你已在测试逻辑中找到 bug。每次创建测试方法时，都值得采取此步骤。

13. 保存文件并重新运行测试。大小写测试通过，但三个计数测试失败。这与我们预期的结果完全一致。



14. 通过将 `Assert.NotEqual` 更改为 `Assert.Equal` 来修改 `CountInstancesCorrectly` 测试方法。保存该文件。重新运行测试。所有测试通过。



添加控制台应用

- 在“解决方案”边栏中，按住 Ctrl 并单击 WordCounter 解决方案。通过从“.NET Core” > “应用” 模板中选择模板来添加新的控制台应用程序项目。选择“下一步”。将项目命名为 WordCounterApp。选择“创建”以在解决方案中创建项目。
- 在“解决方案”边栏中，按住 Ctrl 并单击新“WordCounterApp”项目的“依赖项”。在“编辑引用”对话框中，选中“TextUtils”，然后选择“确定”。
- 打开 Program.cs 文件。将代码替换为以下代码：

```
using System;
using TextUtils;

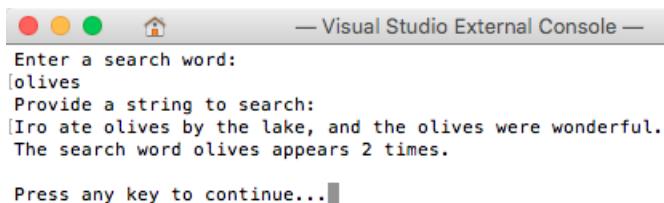
namespace WordCounterApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter a search word:");
            var searchWord = Console.ReadLine();
            Console.WriteLine("Provide a string to search:");
            var inputString = Console.ReadLine();

            var wordCount = WordCount.GetWordCount(searchWord, inputString);

            var pluralChar = "s";
            if (wordCount == 1)
            {
                pluralChar = string.Empty;
            }

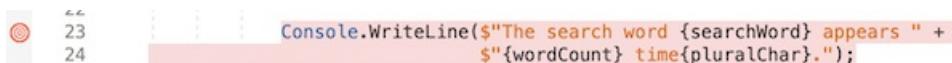
            Console.WriteLine($"The search word {searchWord} appears " +
                $"{wordCount} time{pluralChar}.");
        }
    }
}
```

- 按住 Ctrl 并单击 WordCounterApp 项目，并从上下文菜单中选择“运行项目”。运行应用时，在控制台窗口中的提示符处提供搜索词和输入字符串的值。应用指示搜索词在字符串中出现的次数。



— Visual Studio External Console —
Enter a search word:
[olives
Provide a string to search:
[I ate olives by the lake, and the olives were wonderful.
The search word olives appears 2 times.
Press any key to continue...]

- 要探索的最后一个功能是使用 Visual Studio for Mac 进行调试。在 Console.WriteLine 语句上设置断点：在第 23 行的左边距处选择，可以看到代码行旁边出现红色的圆圈。或者，在代码行上的任意位置进行选择，然后从菜单中选择“运行” > “切换断点”。



- 按住 Ctrl 并单击 WordCounterApp 项目。从上下文菜单中选择“开始调试项目”。应用运行时，输入搜索词“cat”和“The dog chased the cat, but the cat escaped.”以供字符串进行搜索。到达 Console.WriteLine 语句时，将在执行该语句前暂停执行程序。在“本地”选项卡中，可以看到 searchWord、inputString、wordCount 和 pluralChar 值。

```

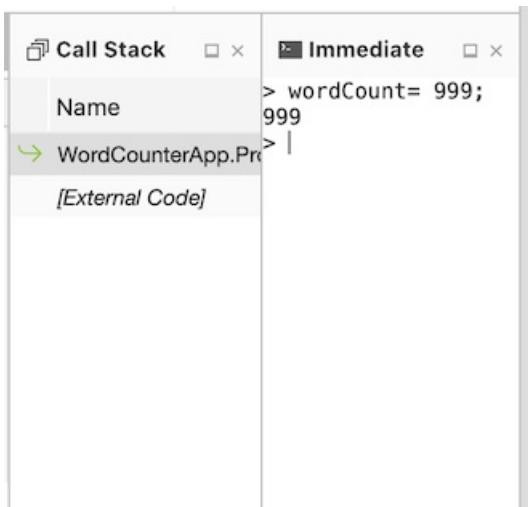
23     Console.WriteLine($"The search word {searchWord} appears " +
24                     $"{wordCount} time{pluralChar}.");
25 }
26 }
27 }

Locals window:
Name      Value      Type
args      {string[0]} string[]
searchWord      "cat"      string
inputString      "The dog chased the cat, but the cat escape" string
wordCount      2      int
pluralChar      "s"      string

Call Stack window:
Name
WordCounterApp.Pr... [External Code]

```

7. 在“即时”面板中，键入“wordCount = 999;”，然后按 Enter 键。此命令将无意义的值 999 分配到 `wordCount` 变量，显示可以在调试时替换变量值。



8. 在工具栏中，单击“继续”。查看控制台窗口中的输出。它报告调试应用时所设置的不正确的值 999。

```

WordCounterApp.dll
Enter a search word:
[cat
Provide a string to search:
[The dog chased the cat, but the cat escaped.
The search word cat appears 999 times.

Press any key to continue...

```

可以使用相同的过程通过单元测试项目来调试代码。按住 Ctrl 并单击“测试库”项目，然后从上下文菜单中选择“启动调试项目”，而不是启动 WordCount 应用项目。Visual Studio for Mac 在附加了调试器的情况下启动测试项目。执行将在添加到测试项目的任何断点或基础库代码处停止。

请参阅

- [Visual Studio 2019 for Mac 发行说明](#)

使用 .NET Core CLI 组织和测试项目

2020/3/18 • [Edit Online](#)

本教程遵循[通过命令行开始在 Windows/Linux/macOS 上使用 .NET Core](#), 不只是介绍了如何创建简单的控制台应用程序, 还介绍了如何开发结构清晰的高级应用程序。在演示如何使用文件夹来组织代码后, 本教程还将说明如何使用 [xUnit](#) 测试框架扩展控制台应用程序。

使用文件夹组织代码

如要要在控制台应用中引入新类型, 可向该应用添加包含该类型的文件。例如, 如果向项目添加包含 `AccountInformation` 和 `MonthlyReportRecords` 类型的文件, 则项目文件结构是平面的, 且易于导航:

```
/MyProject
|__AccountInformation.cs
|__MonthlyReportRecords.cs
|__MyProject.csproj
|__Program.cs
```

但是, 仅在项目规模相对较小时, 此方法才适用。你能否想象在项目中添加 20 个类型时会发生什么? 项目的根目录中会散落许多文件, 这样的项目必然会难以导航和维护。

要组织项目, 请创建一个名为 `Models` 新文件夹, 将其用于保存类型文件。将类型文件放入 `Models` 文件夹中:

```
/MyProject
|__/Models
    |__AccountInformation.cs
    |__MonthlyReportRecords.cs
|__MyProject.csproj
|__Program.cs
```

按逻辑将文件分组到文件夹的项目易于导航和维护。在下一节中, 将创建一个更复杂的示例, 它包含文件夹和单元测试。

使用 NewTypes Pets 示例进行组织和测试

生成示例

对于下列步骤, 可使用 [NewTypes Pets 示例](#) 进行相关操作, 也可以创建自己的文件与文件夹进行操作。各类型按逻辑组织为文件夹结构, 允许日后加入更多类型, 测试也按逻辑放置在文件夹中, 允许日后加入更多测试。

此示例包含两种类型 `Dog` 和 `Cat`, 并使它们实现一个公共接口 `IPet`。对于 `NewTypes` 项目, 目标是将与宠物相关的类型组织到 `Pets` 文件夹中。如果之后添加了另一组类型(例如 `WildAnimals`), 则将其与 `Pets` 文件夹一同放在 `NewTypes` 文件夹中。`WildAnimals` 文件夹可包含不属于宠物的动物类型, 如 `Squirrel` 和 `Rabbit` 类型。按照这种方式添加类型, 不会破坏项目的好组织。

创建以下文件夹结构, 并指明文件内容:

```
/NewTypes
|__/src
|__/NewTypes
|__/Pets
|__Dog.cs
|__Cat.cs
|__IPet.cs
|__Program.cs
|__NewTypes.csproj
```

IPet.cs :

```
using System;

namespace Pets
{
    public interface IPet
    {
        string TalkToOwner();
    }
}
```

Dog.cs :

```
using System;

namespace Pets
{
    public class Dog : IPet
    {
        public string TalkToOwner() => "Woof!";
    }
}
```

Cat.cs :

```
using System;

namespace Pets
{
    public class Cat : IPet
    {
        public string TalkToOwner() => "Meow!";
    }
}
```

Program.cs :

```
using System;
using Pets;
using System.Collections.Generic;

namespace ConsoleApplication
{
    public class Program
    {
        public static void Main(string[] args)
        {
            List<IPet> pets = new List<IPet>
            {
                new Dog(),
                new Cat()
            };

            foreach (var pet in pets)
            {
                Console.WriteLine(pet.TalkToOwner());
            }
        }
    }
}
```

NewTypes.csproj :

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.2</TargetFramework>
</PropertyGroup>

</Project>
```

请执行以下命令：

```
dotnet run
```

获得以下输出：

```
Woof!
Meow!
```

可选练习：可通过扩展此项目添加新的宠物类型，如 `Bird`。使鸟的 `TalkToOwner` 方法向所有者发出 `Tweet!`。再次运行应用。输出将包含 `Tweet!`

测试示例

`NewTypes` 项目已准备就绪，与宠物相关的类型均置于一个文件夹中，因此具有良好的组织。接下来，创建测试项目，并使用 `xUnit` 测试框架开始编写测试。使用单元测试，可自动检查宠物类型的行为，确认其正常运行。

导航回 `src` 文件夹并创建“`test`”文件夹，后者包含 `NewTypesTests` 文件夹。在 `NewTypesTests` 文件夹的命令提示符中，执行 `dotnet new xunit`。这将生成两个文件：`NewTypesTests.csproj` 和 `UnitTest1.cs`。

测试项目当前无法测试 `NewTypes` 中的类型，并且需要对 `NewTypes` 项目的项目引用。要添加项目引用，请使用 `dotnet add reference` 命令：

```
dotnet add reference ../../src/NewTypes/NewTypes.csproj
```

或者，可以选择向 NewTypesTests.csproj 文件添加 `<ItemGroup>` 节点，手动添加项目引用：

```
<ItemGroup>
  <ProjectReference Include="../../src/NewTypes/NewTypes.csproj" />
</ItemGroup>
```

NewTypesTests.csproj：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.4.0" />
    <PackageReference Include="xunit" Version="2.4.1" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.4.1" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="../../src/NewTypes/NewTypes.csproj"/>
  </ItemGroup>

</Project>
```

NewTypesTests.csproj 文件包含下列内容：

- 对 .NET 测试基础结构 `Microsoft.NET.Test.Sdk` 的包引用
- 对 xUnit 测试框架 `xunit` 的包引用
- 对测试运行程序 `xunit.runner.visualstudio` 的包引用
- 对要测试的代码 `NewTypes` 的项目引用

将 UnitTest1.cs 的名称更改为 PetTests.cs，并将文件中的代码替换为下列内容：

```

using System;
using Xunit;
using Pets;

public class PetTests
{
    [Fact]
    public void DogTalkToOwnerReturnsWoof()
    {
        string expected = "Woof!";
        string actual = new Dog().TalkToOwner();

        Assert.NotEqual(expected, actual);
    }

    [Fact]
    public void CatTalkToOwnerReturnsMeow()
    {
        string expected = "Meow!";
        string actual = new Cat().TalkToOwner();

        Assert.NotEqual(expected, actual);
    }
}

```

可选练习：如果先前向所有者添加了生成 `Bird` 的 `Tweet!` 类型，请向 `PetTests.cs` 文件

`BirdTalkToOwnerReturnsTweet` 添加测试方法，检查对于 `TalkToOwner` 类型，`Bird` 方法是否正常工作。

NOTE

尽管期望 `expected` 和 `actual` 值相等，但使用 `Assert.NotEqual` 检查的初始断言表明这些值并不相等。务必最初创建一个失败的测试，以检查测试的逻辑是否正确。确认测试失败后，调整断言，使测试通过。

下面演示了完整的项目结构：

```

/NewTypes
|__/src
|__/NewTypes
|__/Pets
|__Dog.cs
|__Cat.cs
|__IPet.cs
|__Program.cs
|__NewTypes.csproj
|__/test
|__NewTypesTests
|__PetTests.cs
|__NewTypesTests.csproj

```

在 `test/NewTypesTests` 目录中开始。使用 `dotnet restore` 命令还原测试项目。使用 `dotnet test` 命令运行测试。此命令启动项目文件中指定的测试运行程序。

NOTE

从 .NET Core 2.0 SDK 开始，无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build` 和 `dotnet run`。在执行显式还原有意义的某些情况下，例如 [Azure DevOps Services 中的持续集成生成](#) 中，或在需要显式控制还原发生时间的生成系统中，它仍然是有效的命令。

测试按预期失败，控制台显示以下输出：

```
Test run for c:\Users\ronpet\repos\samples\core\Console-
apps\NewTypesMsBuild\test\NewTypesTests\bin\Debug\netcoreapp2.1\NewTypesTests.dll(.NETCoreApp,Version=v2.1)
Microsoft (R) Test Execution Command Line Tool Version 15.8.0
Copyright (c) Microsoft Corporation. All rights reserved.

Starting test execution, please wait...
[xUnit.net 00:00:00.77]      PetTests.DogTalkToOwnerReturnsWoof [FAIL]
[xUnit.net 00:00:00.78]      PetTests.CatTalkToOwnerReturnsMeow [FAIL]
Failed   PetTests.DogTalkToOwnerReturnsWoof
Error Message:
  Assert.NotEqual() Failure
  Expected: Not "Woof!"
  Actual:   "Woof!"
Stack Trace:
  at PetTests.DogTalkToOwnerReturnsWoof() in c:\Users\ronpet\repos\samples\core\Console-
apps\NewTypesMsBuild\test\NewTypesTests\PetTests.cs:line 13
Failed   PetTests.CatTalkToOwnerReturnsMeow
Error Message:
  Assert.NotEqual() Failure
  Expected: Not "Meow!"
  Actual:   "Meow!"
Stack Trace:
  at PetTests.CatTalkToOwnerReturnsMeow() in c:\Users\ronpet\repos\samples\core\Console-
apps\NewTypesMsBuild\test\NewTypesTests\PetTests.cs:line 22

Total tests: 2. Passed: 0. Failed: 2. Skipped: 0.
Test Run Failed.
Test execution time: 1.7000 Seconds
```

将测试的断言从 `Assert.NotEqual` 更改为 `Assert.Equal`：

```
using System;
using Xunit;
using Pets;

public class PetTests
{
    [Fact]
    public void DogTalkToOwnerReturnsWoof()
    {
        string expected = "Woof!";
        string actual = new Dog().TalkToOwner();

        Assert.Equal(expected, actual);
    }

    [Fact]
    public void CatTalkToOwnerReturnsMeow()
    {
        string expected = "Meow!";
        string actual = new Cat().TalkToOwner();

        Assert.Equal(expected, actual);
    }
}
```

使用 `dotnet test` 命令重新运行测试，并获得以下输出：

```
Test run for c:\Users\ronpet\repos\samples\core\console-
apps\NewTypesMsBuild\test\NewTypesTests\bin\Debug\netcoreapp2.1\NewTypesTests.dll(.NETCoreApp,Version=v2.1)
Microsoft (R) Test Execution Command Line Tool Version 15.8.0
Copyright (c) Microsoft Corporation. All rights reserved.

Starting test execution, please wait...

Total tests: 2. Passed: 2. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 1.6029 Seconds
```

测试通过。在与所有者谈话时，宠物类型的方法返回正确的值。

你已了解使用 xUnit 来组织和测试项目的方法。继续使用这些方法，将它们应用于自己的项目中。祝你编码愉快！

使用 .NET Core CLI 开发库

2020/3/18 •• Edit Online

本文介绍如何使用 .NET Core CLI 编写 .NET 的库。CLI 提供可跨任何支持的 OS 工作的高效低级别体验。仍可使用 Visual Studio 生成库，如果你首选这种体验，请[参阅 Visual Studio 指南](#)。

系统必备

需要在计算机上安装 .NET Core SDK 和 CLI。

对于本文档中处理 .NET Framework 版本的部分，需要在 Windows 计算机上安装 .NET Framework。

此外，如果想要支持较旧的 .NET Framework 目标，需要从 [.NET 下载存档页](#) 安装目标包或开发人员工具包。请参阅此表：

.NET FRAMEWORK 版本		说明
4.6.1		.NET Framework 4.6.1 目标包
4.6		.NET Framework 4.6 目标包
4.5.2		.NET Framework 4.5.2 开发人员工具包
4.5.1		.NET Framework 4.5.1 开发人员工具包
4.5		适用于 Windows 8 的 Windows 软件开发工具包
4.0		Windows SDK for Windows 7 和 .NET Framework 4
2.0、3.0 和 3.5		.NET Framework 3.5 SP1 运行时 (或 Windows 8+ 版本)

如何以 .NET Standard 为目标

如果不熟悉 .NET Standard, 请参阅 [.NET Standard](#) 了解详细信息。

在该文中，提供有一个将 .NET Standard 版本映射到各种实现的表格：

.NET STANDARD	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0	2.1
Xamarin. iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14	12.16
Xamarin. Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8	5.16
Xamarin. Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0	10.0
通用 Windows 平台	10.0	10.0	10.0	10.0	10.0	10.0.162 99	10.0.162 99	10.0.162 99	待定
Unity	2018 年 1 月	2018 年 1 月	2018 年 1 月	待定					

1 针对 .NET framework 列出的版本适用于 .NET Core 2.0 SDK 和更高版本的工具。旧版本对 .NET Standard 1.5 及更高版本使用了不同映射。如果无法升级到 Visual Studio 2017 或更高版本, 可[下载适用于 Visual Studio 2015 的 .NET Core 工具](#)。

2 此处所列的版本表示 NuGet 用于确定给定 .NET Standard 库是否适用的规则。虽然 NuGet 将 .NET Framework 4.6.1 视为支持 .NET Standard 1.5 到 2.0, 但使用为从 .NET Framework 4.6.1 项目构建的 .NET Standard 库存在一系列问题。对于需要使用此类库的 .NET Framework 项目, 建议将项目升级到面向 .NET Framework 4.7.2 或更高版本。

3 .NET Framework 不支持 .NET Standard 2.1 或更高版本。有关更多详细信息, 请参阅[.NET Standard 2.1 公告](#)。

- 列表示 .NET Standard 版本。每个标题单元格都是一个文档链接, 其中介绍了相应版本的 .NET Standard 中新增了哪些 API。
- 行表示不同的 .NET 实现。
- 各单元格中的版本号指示要定向到此 .NET Standard 版本所需的最低 实现版本。
- 有关交互式表的信息, 请参阅[.NET Standard 版本](#)。

以下是此表格对于创建库的意义:

选择 .NET Standard 版本时, 需要在能够访问最新 API 与能够定位更多 .NET 实现代码和 .NET Standard 版本之间进行权衡。通过选择 `netstandardX.X` 版本(其中 `X.X` 是版本号)并将其添加到项目文件(`.csproj` 或 `.fsproj`), 控制可面向的平台和版本范围。

面向 .NET Standard 时, 有三种主要选项, 具体取决于你的需求。

1. 可使用 .NET Standard 的默认版本, 该版本由 `netstandard1.4` 模板提供, 可提供对 .NET Standard 上大多数 API 的访问权限, 同时仍与 UWP、.NET Framework 4.6.1 和 .NET Standard 2.0 兼容。

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard1.4</TargetFramework>
  </PropertyGroup>
</Project>
```

2. 可通过修改项目文件 `TargetFramework` 节点中的值来使用更低或更高版本的 .NET Standard。

.NET Standard 版本可后向兼容。这意味着 `netstandard1.0` 库可在 `netstandard1.1` 平台以及更高版本上运行。但是, 没有向前兼容性。低版本的 .NET Standard 平台无法引用较高版本的平台。这意味着 `netstandard1.0` 库不能引用面向 `netstandard1.1` 或更高版本的库。选择适合所需、恰当混合有 API 和平台

支持的 Standard 版本。目前，我们建议 `netstandard1.4`。

3. 如果希望面向 .NET Framework 版本 4.0 或更低版本，或者要使用 .NET Framework 中提供但 .NET Standard 中不提供的 API（例如 `System.Drawing`），请阅读以下部分，了解如何设定多目标。

如何面向 .NET framework

NOTE

这些说明假定计算机上安装有 .NET Framework。请参阅[先决条件](#) 获取安装的依赖项。

请记住，此处使用的某些 .NET Framework 版本不再受支持。有关不受支持的版本信息，请参阅[.NET Framework 支持生命周期策略常见问题](#)。

如果要达到最大数量的开发人员和项目，可将 .NET Framework 4.0 用作基线目标。若要以 .NET Framework 为目标，首先使用与要支持的 .NET Framework 版本相对应的正确目标框架名字对象 (TFM)。

.NET FRAMEWORK	TFM
.NET Framework 2.0	<code>net20</code>
.NET Framework 3.0	<code>net30</code>
.NET Framework 3.5	<code>net35</code>
.NET Framework 4.0	<code>net40</code>
.NET Framework 4.5	<code>net45</code>
.NET Framework 4.5.1	<code>net451</code>
.NET Framework 4.5.2	<code>net452</code>
.NET Framework 4.6	<code>net46</code>
.NET Framework 4.6.1	<code>net461</code>
.NET Framework 4.6.2	<code>net462</code>
.NET Framework 4.7	<code>net47</code>
.NET Framework 4.8	<code>net48</code>

然后将此 TFM 插入项目文件的 `TargetFramework` 部分。例如，下面展示了如何编写面向 .NET Framework 4.0 的库：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net40</TargetFramework>
  </PropertyGroup>
</Project>
```

这就是所有的操作！虽然此库仅针对 .NET Framework 4 编译，但可在较新版本的 .NET Framework 上使用此库。

如何设定多目标

NOTE

以下说明假定计算机上安装有 .NET Framework。请参阅[先决条件](#)部分，了解需要安装哪些依赖项以及在何处下载。

如果项目同时支持 .NET Framework 和 .NET Core，可能需要面向较旧版本的 .NET Framework。在此方案中，如果要为较新目标使用较新的 API 和语言构造，请在代码中使用 `#if` 指令。可能还需要为要面向的每个平台添加不同的包和依赖项，以包含每种情况所需的不同 API。

例如，假设有一个库，它通过 HTTP 执行联网操作。对于 .NET Standard 和 .NET Framework 版本 4.5 或更高版本，可从 `HttpClient` 命名空间使用 `System.Net.Http` 类。但是，.NET Framework 的早期版本没有 `HttpClient` 类，因此可对早期版本使用 `WebClient` 命名空间中的 `System.Net` 类。

项目文件可能如下所示：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFrameworks>netstandard1.4;net40;net45</TargetFrameworks>
  </PropertyGroup>

  <!-- Need to conditionally bring in references for the .NET Framework 4.0 target -->
  <ItemGroup Condition="'$(TargetFramework)' == 'net40'">
    <Reference Include="System.Net" />
  </ItemGroup>

  <!-- Need to conditionally bring in references for the .NET Framework 4.5 target -->
  <ItemGroup Condition="'$(TargetFramework)' == 'net45'">
    <Reference Include="System.Net.Http" />
    <Reference Include="System.Threading.Tasks" />
  </ItemGroup>
</Project>
```

在此处可看到三项主要更改：

1. `TargetFramework` 节点已替换为 `TargetFrameworks`，其中表示了三个 TFM。
2. `<ItemGroup>` 目标有一个 `net40` 节点，拉取一个 .NET Framework 引用。
3. `<ItemGroup>` 目标中有一个 `net45` 节点，拉取两个 .NET Framework 引用。

生成系统可识别以下用在 `#if` 指令中的处理器符号：

.NET Framework	NETFRAMEWORK, NET20, NET35, NET40, NET45, NET451, NET452, NET46, NET461, NET462, NET47, NET471, NET472, NET48
.NET Standard	NETSTANDARD, NETSTANDARD1_0, NETSTANDARD1_1, NETSTANDARD1_2, NETSTANDARD1_3, NETSTANDARD1_4, NETSTANDARD1_5, NETSTANDARD1_6, NETSTANDARD2_0, NETSTANDARD2_1
.NET Core	NETCOREAPP, NETCOREAPP1_0, NETCOREAPP1_1, NETCOREAPP2_0, NETCOREAPP2_1, NETCOREAPP2_2, NETCOREAPP3_0, NETCOREAPP3_1

以下是使用每目标条件编译的示例：

```

using System;
using System.Text.RegularExpressions;
#if NET40
// This only compiles for the .NET Framework 4 targets
using System.Net;
#else
// This compiles for all other targets
using System.Net.Http;
using System.Threading.Tasks;
#endif

namespace MultitargetLib
{
    public class Library
    {
#if NET40
        private readonly WebClient _client = new WebClient();
        private readonly object _locker = new object();
#else
        private readonly HttpClient _client = new HttpClient();
#endif

#if NET40
        // .NET Framework 4.0 does not have async/await
        public string GetDotNetCount()
        {
            string url = "https://www.dotnetfoundation.org/";

            var uri = new Uri(url);

            string result = "";

            // Lock here to provide thread-safety.
            lock(_locker)
            {
                result = _client.DownloadString(uri);
            }

            int dotNetCount = Regex.Matches(result, ".NET").Count;

            return $"Dotnet Foundation mentions .NET {dotNetCount} times!";
        }
#else
        // .NET 4.5+ can use async/await!
        public async Task<string> GetDotNetCountAsync()
        {
            string url = "https://www.dotnetfoundation.org/";

            // HttpClient is thread-safe, so no need to explicitly lock here
            var result = await _client.GetStringAsync(url);

            int dotNetCount = Regex.Matches(result, ".NET").Count;

            return $"dotnetfoundation.org mentions .NET {dotNetCount} times in its HTML!";
        }
#endif
    }
}

```

如果使用 `dotnet build` 生成此项目，则在 `bin/` 文件夹下有三个目录：

```

net40/
net45/
netstandard1.4/

```

其中每个目录都包含每个目标的 `.dll` 文件。

如何在 .NET Core 上测试库

能够跨平台进行测试至关重要。可使用现成的 [xUnit](#) 或 [MSTest](#)。它们都十分适合在 .NET Core 上对库进行单元测试。如何使用测试项目设置解决方案取决于[解决方案的结构](#)。下面的示例假设测试和源目录位于同一顶级目录下。

NOTE

此示例将使用某些 [.NET Core CLI](#) 命令。有关详细信息, 请参阅 [dotnet new](#) 和 [dotnet sln](#)。

1. 设置解决方案。可使用以下命令实现此目的:

```
mkdir SolutionWithSrcAndTest
cd SolutionWithSrcAndTest
dotnet new sln
dotnet new classlib -o MyProject
dotnet new xunit -o MyProject.Test
dotnet sln add MyProject/MyProject.csproj
dotnet sln add MyProject.Test/MyProject.Test.csproj
```

这将创建多个项目，并一个解决方案中将这些项目链接在一起。`SolutionWithSrcAndTest` 的目录应如下所示：

```
/SolutionWithSrcAndTest
|__SolutionWithSrcAndTest.sln
|__MyProject/
|__MyProject.Test/
```

2. 导航到测试项目的目录, 然后添加对 `MyProject.Test` 中的 `MyProject` 的引用。

```
cd MyProject.Test
dotnet add reference ../../MyProject/MyProject.csproj
```

3. 还原包和生成项目：

```
dotnet restore
dotnet build
```

NOTE

从 .NET Core 2.0 SDK 开始, 无需运行 `dotnet restore`, 因为它由所有需要还原的命令隐式运行, 如 `dotnet new`、`dotnet build` 和 `dotnet run`。在执行显式还原有意义的某些情况下, 例如 [Azure DevOps Services 中的持续集成生成](#) 中, 或在需要显式控制还原发生时间的生成系统中, 它仍然是有效的命令。

4. 执行 `dotnet test` 命令, 验证 xUnit 是否在运行。如果选择使用 MSTest, 则应改为运行 MSTest 控制台运行程序。

这就是所有的操作！现在可以使用命令行工具跨所有平台测试库。若要继续测试, 现已设置好了所有内容, 测试库将非常简单：

1. 对库进行更改。
2. 使用 `dotnet test` 命令在测试目录中从命令行运行测试。

调用 `dotnet test` 命令时，将自动重新生成代码。

如何使用多个项目

对于较大的库，通常需要将功能置于不同项目中。

假设要生成一个可以惯用的 C# 和 F# 使用的库。这意味着库的使用者可通过对 C# 或 F# 来说很自然的方式来使用它。例如，在 C# 中，了能会这样使用库：

```
using AwesomeLibrary.CSharp;

public Task DoThings(Data data)
{
    var convertResult = await AwesomeLibrary.ConvertAsync(data);
    var result = AwesomeLibrary.Process(convertResult);
    // do something with result
}
```

在 F# 中可能是这样：

```
open AwesomeLibrary.FSharp

let doWork data = async {
    let! result = AwesomeLibrary.AsyncConvert data // Uses an F# async function rather than C# async method
    // do something with result
}
```

这样的使用方案意味着被访问的 API 必须具有用于 C# 和 F# 的不同结构。通常的方法是将库的所有逻辑因子转化到核心项目中，C# 和 F# 项目定义调用到核心项目的 API 层。该部分的其余部分将使用以下名称：

- `AwesomeLibrary.Core` - 核心项目，其中包含库的所有逻辑
- `AwesomeLibrary.CSharp` - 具有打算在 C# 中使用的公共 API 的项目
- `AwesomeLibrary.FSharp` - 具有打算在 F# 中使用的公共 API 的项目

可在终端运行下列命令，生成与下列指南相同的结构：

```
mkdir AwesomeLibrary && cd AwesomeLibrary
dotnet new sln
mkdir AwesomeLibrary.Core && cd AwesomeLibrary.Core && dotnet new classlib
cd ..
mkdir AwesomeLibrary.CSharp && cd AwesomeLibrary.CSharp && dotnet new classlib
cd ..
mkdir AwesomeLibrary.FSharp && cd AwesomeLibrary.FSharp && dotnet new classlib -lang "F#"
cd ..
dotnet sln add AwesomeLibrary.Core/AwesomeLibrary.Core.csproj
dotnet sln add AwesomeLibrary.CSharp/AwesomeLibrary.CSharp.csproj
dotnet sln add AwesomeLibrary.FSharp/AwesomeLibrary.FSharp.fsproj
```

这将添加上述三个项目和将它们链接在一起的解决方案文件。创建解决方案文件并链接项目后，可从顶级还原和生成项目。

项目到项目的引用

引用项目的最佳方式是使用 .NET Core CLI 添加项目引用。在 `AwesomeLibrary.CSharp` 和 `AwesomeLibrary.FSharp` 项目目录中，可运行下列命令：

```
dotnet add reference ../AwesomeLibrary.Core/AwesomeLibrary.Core.csproj
```

AwesomeLibrary.CSharp 和 AwesomeLibrary.FSharp 的项目文件现在需要将 AwesomeLibrary.Core 作为 `ProjectReference` 目标引用。可通过检查项目文件和查看其中的下列内容来进行验证：

```
<ItemGroup>
  <ProjectReference Include="..\AwesomeLibrary.Core\AwesomeLibrary.Core.csproj" />
</ItemGroup>
```

如果不想使用 .NET Core CLI，可手动将此部分添加到每个项目文件。

结构化解决方案

多项目解决方案的另一个重要方面是建立良好的整体项目结构。可根据自己的喜好随意组织代码，只要使用 `dotnet sln add` 将每个项目链接到解决方案文件，就可在解决方案级别运行 `dotnet restore` 和 `dotnet build`。

使用插件创建 .NET Core 应用程序

2020/3/18 • [Edit Online](#)

本教程展示了如何创建自定义的 `AssemblyLoadContext` 来加载插件。`AssemblyDependencyResolver` 用于解析插件的依赖项。该教程正确地将插件依赖项与主机应用程序隔离开来。将了解如何执行以下操作：

- 构建支持插件的项目。
- 创建自定义 `AssemblyLoadContext` 加载每个插件。
- 使用 `System.Runtime.Loader.AssemblyDependencyResolver` 类型允许插件具有依赖项。
- 只需复制生成项目就可以轻松部署的作者插件。

系统必备

- 安装 [.NET Core 3.0 SDK](#) 或更高版本。

创建应用程序

第一步是创建应用程序：

1. 创建新文件夹，并在该文件夹中运行以下命令：

```
dotnet new console -o AppWithPlugin
```

2. 为了更容易生成项目，请在同一文件夹中创建一个 Visual Studio 解决方案文件。运行以下命令：

```
dotnet new sln
```

3. 运行以下命令，向解决方案添加应用项目：

```
dotnet sln add AppWithPlugin/AppWithPlugin.csproj
```

现在，我们可以填写应用程序的主干。使用下面的代码替换 `AppWithPlugin/Program.cs` 文件中的代码：

```

using PluginBase;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;

namespace AppWithPlugin
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                if (args.Length == 1 && args[0] == "/d")
                {
                    Console.WriteLine("Waiting for any key...");
                    Console.ReadLine();
                }

                // Load commands from plugins.

                if (args.Length == 0)
                {
                    Console.WriteLine("Commands: ");
                    // Output the loaded commands.
                }
                else
                {
                    foreach (string commandName in args)
                    {
                        Console.WriteLine($"-- {commandName} --");

                        // Execute the command with the name passed as an argument.

                        Console.WriteLine();
                    }
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex);
            }
        }
    }
}

```

创建插件接口

使用插件生成应用的下一步是定义插件需要实现的接口。我们建议创建类库，其中包含计划用于在应用和插件之间通信的任何类型。此部分允许将插件接口作为包发布，而无需发布完整的应用程序。

在项目的根文件夹中，运行 `dotnet new classlib -o PluginBase`。并运行

`dotnet sln add PluginBase/PluginBase.csproj` 向解决方案文件添加项目。删除 `PluginBase/Class1.cs` 文件，并使用以下接口定义在名为 `PluginBase` 的 `ICommand.cs` 文件夹中创建新的文件：

```
namespace PluginBase
{
    public interface ICommand
    {
        string Name { get; }
        string Description { get; }

        int Execute();
    }
}
```

此 `ICommand` 接口是所有插件将实现的接口。

由于已定义 `ICommand` 接口，所以应用程序项目可以填写更多内容。使用根文件夹中的 `AppWithPlugin` 命令将引用从 `PluginBase` 项目添加到 `dotnet add AppWithPlugin\AppWithPlugin.csproj reference PluginBase\PluginBase.csproj` 项目。

使用以下代码片段替换 `// Load commands from plugins` 注释，使其能够从给定文件路径加载插件：

```
string[] pluginPaths = new string[]
{
    // Paths to plugins to load.
};

IEnumerable<ICommand> commands = pluginPaths.SelectMany(pluginPath =>
{
    Assembly pluginAssembly = LoadPlugin(pluginPath);
    return CreateCommands(pluginAssembly);
}).ToList();
```

然后用以下代码片段替换 `// Output the loaded commands` 注释：

```
foreach (ICommand command in commands)
{
    Console.WriteLine($"{command.Name}\t - {command.Description}");
}
```

使用以下代码片段替换 `// Execute the command with the name passed as an argument` 注释：

```
ICommand command = commands.FirstOrDefault(c => c.Name == commandName);
if (command == null)
{
    Console.WriteLine("No such command is known.");
    return;
}

command.Execute();
```

最后，将静态方法添加到名为 `Program` 和 `LoadPlugin` 的 `CreateCommands` 类，如下所示：

```
static Assembly LoadPlugin(string relativePath)
{
    throw new NotImplementedException();
}

static IEnumerable<ICommand> CreateCommands(Assembly assembly)
{
    int count = 0;

    foreach (Type type in assembly.GetTypes())
    {
        if (typeof(ICommand).IsAssignableFrom(type))
        {
            ICommand result = Activator.CreateInstance(type) as ICommand;
            if (result != null)
            {
                count++;
                yield return result;
            }
        }
    }

    if (count == 0)
    {
        string availableTypes = string.Join(", ", assembly.GetTypes().Select(t => t.FullName));
        throw new ApplicationException(
            $"Can't find any type which implements ICommand in {assembly} from {assembly.Location}.\n" +
            $"Available types: {availableTypes}");
    }
}
```

加载插件

现在，应用程序可以正确加载和实例化来自已加载的插件程序集的命令，但仍然无法加载插件程序集。使用以下内容在 AppWithPlugin 文件夹中创建名为 PluginLoadContext.cs 的文件：

```

using System;
using System.Reflection;
using System.Runtime.Loader;

namespace AppWithPlugin
{
    class PluginLoadContext : AssemblyLoadContext
    {
        private AssemblyDependencyResolver _resolver;

        public PluginLoadContext(string pluginPath)
        {
            _resolver = new AssemblyDependencyResolver(pluginPath);
        }

        protected override Assembly Load(AssemblyName assemblyName)
        {
            string assemblyPath = _resolver.ResolveAssemblyToPath(assemblyName);
            if (assemblyPath != null)
            {
                return LoadFromAssemblyPath(assemblyPath);
            }

            return null;
        }

        protected override IntPtr LoadUnmanagedDll(string unmanagedDllName)
        {
            string libraryPath = _resolver.ResolveUnmanagedDllToPath(unmanagedDllName);
            if (libraryPath != null)
            {
                return LoadUnmanagedDllFromPath(libraryPath);
            }

            return IntPtr.Zero;
        }
    }
}

```

`PluginLoadContext` 类型派生自 `AssemblyLoadContext`。`AssemblyLoadContext` 类型是运行时中的特殊类型，该类型允许开发人员将已加载的程序集隔离到不同的组中，以确保程序集版本不冲突。此外，自定义 `AssemblyLoadContext` 可以选择不同路径来加载程序集格式并重写默认行为。`PluginLoadContext` 使用 .NET Core 3.0 中引入的 `AssemblyDependencyResolver` 类型的实例将程序集名称解析为路径。`AssemblyDependencyResolver` 对象是使用 .NET 类库的路径构造的。它根据类库的 `.deps.json` 文件（其路径传递给 构造函数）将程序集和本机库解析为它们的相对路径 `AssemblyDependencyResolver`。自定义 `AssemblyLoadContext` 使插件能够拥有自己的依赖项，`AssemblyDependencyResolver` 使正确加载依赖项变得容易。

由于 `AppWithPlugin` 项目具有 `PluginLoadContext` 类型，所以请使用以下正文更新 `Program.LoadPlugin` 方法：

```
static Assembly LoadPlugin(string relativePath)
{
    // Navigate up to the solution root
    string root = Path.GetFullPath(Path.Combine(
        Path.GetDirectoryName(
            Path.GetDirectoryName(
                Path.GetDirectoryName(
                    Path.GetDirectoryName(
                        Path.GetDirectoryName(typeof(Program).Assembly.Location))))));

    string pluginLocation = Path.GetFullPath(Path.Combine(root, relativePath.Replace('\\',
Path.DirectorySeparatorChar)));
    Console.WriteLine($"Loading commands from: {pluginLocation}");
    PluginLoadContext loadContext = new PluginLoadContext(pluginLocation);
    return loadContext.LoadFromAssemblyName(new
AssemblyName(Path.GetFileNameWithoutExtension(pluginLocation)));
}
```

通过为每个插件使用不同的 `PluginLoadContext` 实例，插件可以具有不同的甚至冲突的依赖项，而不会出现问题。

不具有依赖项的简单插件

返回到根文件夹，执行以下步骤：

1. 运行以下命令，新建一个名为 `HelloPlugin` 的类库项目：

```
dotnet new classlib -o HelloPlugin
```

2. 运行以下命令，将项目添加到 `AppWithPlugin` 解决方案中：

```
dotnet sln add HelloPlugin/HelloPlugin.csproj
```

3. 使用以下内容将 `HelloPlugin/Class1.cs` 文件替换为名为 `HelloCommand.cs` 的文件：

```
using PluginBase;
using System;

namespace HelloPlugin
{
    public class HelloCommand : ICommand
    {
        public string Name { get => "hello"; }
        public string Description { get => "Displays hello message." }

        public int Execute()
        {
            Console.WriteLine("Hello !!!");
            return 0;
        }
    }
}
```

现在，打开 `HelloPlugin.csproj` 文件。它应类似于以下内容：

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
  <TargetFramework>netcoreapp3.0</TargetFramework>
</PropertyGroup>

</Project>
```

在 `<Project>` 标记之间添加以下元素：

```
<ItemGroup>
  <ProjectReference Include=".\\PluginBase\\PluginBase.csproj">
    <Private>false</Private>
    <ExcludeAssets>runtime</ExcludeAssets>
  </ProjectReference>
</ItemGroup>
```

`<Private>false</Private>` 元素很重要。它告知 MSBuild 不要将 `PluginBase.dll` 复制到 `HelloPlugin` 的输出目录。如果 `PluginBase.dll` 程序集出现在输出目录中，将在那里查找到该程序集并在加载 `HelloPlugin.dll` 程序集时加载它 `PluginLoadContext`。此时，`HelloPlugin.HelloCommand` 类型将从 `ICommand` 项目的输出目录中的 `PluginBase.dll` 实现接口，而不是加载到默认加载上下文中的接口 `HelloPlugin` ICommand`。因为运行时将这两种类型视为不同程序集的不同类型，所以 `AppWithPlugin.Program.CreateCommand` 方法找不到命令。因此，对包含插件接口的程序集的引用需要 `<Private>false</Private>` 元数据。

同样，如果 `<ExcludeAssets>runtime</ExcludeAssets>` 引用其他包，则 `PluginBase` 元素也很重要。此设置与 `<Private>false</Private>` 的效果相同，但适用于 `PluginBase` 项目或它的某个依赖项可能包括的包引用。

因为 `HelloPlugin` 项目已完成，所以应该更新 `AppWithPlugin` 项目，以确认可以找到 `HelloPlugin` 插件的位置。在 `// Paths to plugins to load` 注释之后，添加 `@"HelloPlugin\\bin\\Debug\\netcoreapp3.0\\HelloPlugin.dll"` 作为 `pluginPaths` 数组的元素。

具有库依赖项的插件

几乎所有插件都比简单的“Hello World”更复杂，而且许多插件都具有其他库上的依赖项。示例中的 `JsonPlugin` 和 `OldJson` 插件项目显示了具有 `Newtonsoft.Json` 上的 NuGet 包依赖项的两个插件示例。项目文件本身没有关于项目引用的任何特殊信息，并且（在将插件路径添加到 `pluginPaths` 数组之后）即使是在 `AppWithPlugin` 应用的同一执行中运行，插件也能完美运行。但是，这些项目不会将引用的程序集复制到它们的输出目录中，因此需要将这些程序集显示在用户的计算机上，以便插件能够正常工作。解决此问题有两种方法。第一种是使用 `dotnet publish` 命令发布类库。或者，如果希望能够将 `dotnet build` 的输出用于插件，可以在插件的项目文件中的 `<CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>` 标记之间添加 `<PropertyGroup>` 属性。有关示例，请参阅 `XcopyablePlugin` 插件项目。

示例中的其他示例

可以在 `dotnet/samples 存储库` 中找到本教程的完整源代码。完成的示例包括 `AssemblyDependencyResolver` 行为的一些其他示例。例如，`AssemblyDependencyResolver` 对象还可以解析本机库和 NuGet 包中所包含的已本地化的附属程序集。示例存储库中的 `UVPlugin` 和 `FrenchPlugin` 演示了这些方案。

从 NuGet 包引用插件接口

假设存在应用 A，它具有 NuGet 包（名为 `A.PluginBase`）中定义的插件接口。如何在插件项目中正确引用包？对于项目引用，使用项目文件的 `<Private>false</Private>` 元素上的 `ProjectReference` 元数据会阻止将 dll 复制到输出。

若要正确引用 `A.PluginBase` 包，应将项目文件中的 `<PackageReference>` 元素更改为以下内容：

```
<PackageReference Include="A.PluginBase" Version="1.0.0">
  <ExcludeAssets>runtime</ExcludeAssets>
</PackageReference>
```

此操作会阻止将 `A.PluginBase` 程序集复制到插件的输出目录，并确保插件将使用 A 版本的 `A.PluginBase`。

插件目标框架建议

因为插件依赖项加载使用 `.deps.json` 文件，所以存在一个与插件的目标框架相关的问题。具体来说，插件应该以运行时为目标，比如 .NET Core 3.0，而不是某一版本的 .NET Standard。`.deps.json` 文件基于项目所针对的框架生成，而且由于许多与 .NET Standard 兼容的包提供了用于针对 .NET Standard 进行生成的引用程序集和用于特定运行时的实现程序集，因此 `.deps.json` 可能无法正确查看实现程序集，或者它可能会获取 .NET Standard 版本的程序集，而不是期望的 .NET Core 版本的程序集。

插件框架引用

插件当前无法向该过程引入新的框架。例如，无法将使用 `Microsoft.AspNetCore.App` 框架的插件加载到只使用根 `Microsoft.NETCore.App` 框架的应用程序中。主机应用程序必须声明对插件所需的全部框架的引用。

ASP.NET Core 入门

2020/3/18 • [Edit Online](#)

有关开发 ASP.NET Core Web 应用程序的教程，请参阅 [ASP.NET Core 教程](#)。

编写自定义 .NET Core 主机以从本机代码控制 .NET 运行时

2020/3/18 • [Edit Online](#)

像所有的托管代码一样, .NET Core 应用程序也由主机执行。主机负责启动运行时(包括 JIT 和垃圾回收器等组件)和调用托管的入口点。

托管 .NET Core 运行时是高级方案, 在大多数情况下, .NET Core 开发人员无需担心托管问题, 因为 .NET Core 生成过程会提供默认主机来运行 .NET Core 应用程序。虽然在某些特殊情况下, 它对显式托管 .NET Core 运行时非常有用 - 无论是作为一种在本机进程中调用托管代码的方式还是为了获得对运行时工作原理更好的控制。

本文概述了从本机代码启动 .NET Core 运行时和在其中执行托管代码的必要步骤。

先决条件

由于主机是本机应用程序, 所以本教程将介绍如何构造 C++ 应用程序以托管 .NET Core。将需要一个 C++ 开发环境(例如, [Visual Studio](#) 提供的环境)。

还将需要一个简单的 .NET Core 应用程序来测试主机, 因此应安装 [.NET Core SDK 并构建一个小型的 .NET Core 测试应用](#)(例如, “Hello World”应用)。使用通过新 .NET Core 控制台项目模板创建的“Hello World”应用就足够了。

承载 API

可以使用三种不同的 API 来托管 .NET Core。本文(及其相关的[示例](#))涵盖所有选项。

- 在 .NET Core 3.0 及更高版本中托管 .NET Core 运行时的首选方法是借助 `nethost` 和 `hostfxr` 库的 API。由这些入口点来处理查找和设置运行时进行初始化所遇到的复杂性;通过它们, 还可启动托管应用程序和调用静态托管方法。
- 托管低于 .NET Core 3.0 的 .NET Core 运行时的首选方法是使用 `CoreClrHost.h` API。此 API 公开一些函数, 用于轻松地启动和停止运行时并调用托管代码(通过启动托管 exe 或通过调用静态托管方法)。
- 也可以使用 `mscoree.h` 中的 `ICLRRuntimeHost4` 接口承载 .NET Core。此 API 比 `CoreClrHost.h` 出现的时间更早, 因此你可能看到过使用此 API 的较旧版本的主机。它仍然可用, 并且比起 `CoreClrHost`, 它可以对主机进程进行更多控制。但是对于大多数方案, `CoreClrHost.h` 现在是首选, 因为它的 API 更简单。

示例主机

有关展示在下面的教程中所述步骤的[示例主机](#), 请访问 `dotnet/samples` GitHub 存储库。该示例中的注释清楚地将这些教程中已编号的步骤与它们在示例中的执行位置关联。有关下载说明, 请参阅[示例和教程](#)。

请记住, 示例主机的用途在于提供学习指导, 在纠错方面不甚严谨, 其重在可读性而非效率。

使用 `NetHost.h` 和 `HostFxr.h` 创建主机

以下步骤详细说明如何使用 `nethost` 和 `hostfxr` 库在本机应用程序中启动 .NET Core 运行时并调用托管静态方法。[示例](#)使用安装了 .NET SDK 的 `nethost` 标头和库, 并从 `dotnet/core-setup` 复制 `coreclr_delegates.h` 和 `hostfxr.h` 文件。

步骤 1 - 加载 `HostFxr` 并获取导出的托管函数

`nethost` 库提供用于查找 `hostfxr` 库的 `get_hostfxr_path` 函数。`hostfxr` 库公开用于托管 .NET Core 运行时的函数。函数的完整列表可在 `hostfxr.h` 和[本机托管设计文档](#)中找到。示例和本教程使用以下函数:

- `hostfxr_initialize_for_runtime_config`: 初始化主机上下文，并使用指定的运行时配置准备初始化 .NET Core 运行时。
- `hostfxr_get_runtime_delegate`: 获取对运行时功能的委托。
- `hostfxr_close`: 关闭主机上下文。

使用 `get_hostfxr_path` 找到了 `hostfxr` 库。随后加载此库并检索其导出。

```
// Using the nethost library, discover the location of hostfxr and get exports
bool load_hostfxr()
{
    // Pre-allocate a large buffer for the path to hostfxr
    char_t buffer[MAX_PATH];
    size_t buffer_size = sizeof(buffer) / sizeof(char_t);
    int rc = get_hostfxr_path(buffer, &buffer_size, nullptr);
    if (rc != 0)
        return false;

    // Load hostfxr and get desired exports
    void *lib = load_library(buffer);
    init_fptr = (hostfxr_initialize_for_runtime_config_fn) get_export(lib,
"hostfxr_initialize_for_runtime_config");
    get_delegate_fptr = (hostfxr_get_runtime_delegate_fn) get_export(lib, "hostfxr_get_runtime_delegate");
    close_fptr = (hostfxr_close_fn) get_export(lib, "hostfxr_close");

    return (init_fptr && get_delegate_fptr && close_fptr);
}
```

步骤 2 - 初始化和启动 .NET Core 运行时

`hostfxr_initialize_for_runtime_config` 和 `hostfxr_get_runtime_delegate` 函数使用将加载的托管组件的运行时配置初始化并启动 .NET Core 运行时。`hostfxr_get_runtime_delegate` 函数用于获取运行时委托，允许加载托管程序集并获取指向该程序集中的静态方法的函数指针。

```
// Load and initialize .NET Core and get desired function pointer for scenario
load_assembly_and_get_function_pointer_fn get_dotnet_load_assembly(const char_t *config_path)
{
    // Load .NET Core
    void *load_assembly_and_get_function_pointer = nullptr;
    hostfxr_handle ctxt = nullptr;
    int rc = init_fptr(config_path, nullptr, &ctxt);
    if (rc != 0 || ctxt == nullptr)
    {
        std::cerr << "Init failed: " << std::hex << std::showbase << rc << std::endl;
        close_fptr(ctxt);
        return nullptr;
    }

    // Get the load assembly function pointer
    rc = get_delegate_fptr(
        ctxt,
        hdt_load_assembly_and_get_function_pointer,
        &load_assembly_and_get_function_pointer);
    if (rc != 0 || load_assembly_and_get_function_pointer == nullptr)
        std::cerr << "Get delegate failed: " << std::hex << std::showbase << rc << std::endl;

    close_fptr(ctxt);
    return (load_assembly_and_get_function_pointer_fn)load_assembly_and_get_function_pointer;
}
```

步骤 3 - 加载托管程序集并获取指向托管方法的函数指针

将调用运行时委托以加载托管程序集并获取指向托管方法的函数指针。委托需要程序集路径、类型名称和方法名称作为输入，并返回可用于调用托管方法的函数指针。

```
// Function pointer to managed delegate
component_entry_point_fn hello = nullptr;
int rc = load_assembly_and_get_function_pointer(
    dotnetlib_path.c_str(),
    dotnet_type,
    dotnet_type_method,
    nullptr /*delegate_type_name*/,
    nullptr,
    (void**)&hello);
```

该示例通过在调用运行时委托时将 `nullptr` 作为委托类型名称传递，对托管方法使用默认签名：

```
public delegate int ComponentEntryPoint(IntPtr args, int sizeBytes);
```

可以通过在调用运行时委托时指定委托类型名称来使用其他签名。

步骤 4 - 运行托管代码！

本机主机现在可以调用托管方法，并向其传递所需的参数。

```
lib_args args
{
    STR("from host!"),
    i
};

hello(&args, sizeof(args));
```

使用 CoreClrHost.h 创建主机

以下步骤详细说明如何使用 `CoreClrHost.h` API 在本机应用程序中启动 .NET Core 运行时并调用托管静态方法。本文档中的代码片段使用一些特定于 Windows 的 API，但是[完整示例主机](#)同时显示 Windows 和 Linux 的代码路径。

[Unix CoreRun 主机](#)显示使用 `coreclrhost.h` 的更为复杂的真实托管示例。

步骤 1 - 查找和加载 CoreCLR

.NET Core 运行时 API 位于 `coreclr.dll`(对于 Windows)、`libcoreclr.so`(对于 Linux)或 `libcoreclr.dylib`(对于 macOS)。承载 .NET Core 的第一步是加载 CoreCLR 库。一些主机探测不同的路径或使用输入参数来查找库，而另一些主机能够从某个路径(例如，紧邻主机的路径，或从计算机范围内的位置)加载库。

找到库之后，系统会使用 `LoadLibraryEx` (对于 Windows) 或 `dlopen` (对于 Linux/macOS) 加载库。

```
HMODULE coreClr = LoadLibraryExA(coreClrPath.c_str(), NULL, 0);
```

步骤 2 - 获取 .NET Core 承载函数

`CoreClrHost` 有几个可用于承载 .NET Core 的重要方法：

- `coreclr_initialize`：启动 .NET Core 运行时并设置默认(且仅设置)AppDomain。
- `coreclr_execute_assembly`：执行托管程序集。
- `coreclr_create_delegate`：创建指向托管方法的函数指针。
- `coreclr_shutdown`：关闭 .NET Core 运行时。
- `coreclr_shutdown_2`：如 `coreclr_shutdown`，但还会检索托管代码的退出代码。

加载 CoreCLR 库之后，下一步是使用 `GetProcAddress` (对于 Windows) 或 `dlsym` (对于 Linux/macOS) 引用这些函

数。

```
coreclr_initialize_ptr initializeCoreClr = (coreclr_initialize_ptr)GetProcAddress(coreClr,
    "coreclr_initialize");
coreclr_create_delegate_ptr createManagedDelegate = (coreclr_create_delegate_ptr)GetProcAddress(coreClr,
    "coreclr_create_delegate");
coreclr_shutdown_ptr shutdownCoreClr = (coreclr_shutdown_ptr)GetProcAddress(coreClr, "coreclr_shutdown");
```

步骤 3 - 准备运行时属性

在启动运行时之前，有必要准备一些属性来指定行为（特别是关于程序集加载器的行为）。

常用属性包括：

- **TRUSTED_PLATFORM_ASSEMBLIES** 这是程序集路径列表（对于 Windows，使用“;”分隔，对于 Linux，使用“:”分隔），运行时在默认情况下能够解析这些路径。一些主机有硬编码清单，其中列出了它们可以加载的程序集。其他主机将把任何库放在这个列表上的特定位置（例如 coreclr.dll 旁边）。
- **APP_PATHS** 这是一个用来探测程序集的路径的列表（如果在受信任的平台程序集（TPA）列表中找不到程序集）。因为主机使用 TPA 列表可以更好地控制加载哪些程序集，所以对于主机来说，确定要加载的程序集并显式列出它们是最佳做法。但是，如果需要探测运行时，则此属性可以支持该方案。
- **APP_NI_PATHS** 此列表与 APP_PATHS 相似，不同之处在于其中的路径用于探测本机映像。
- **NATIVE_DLL_SEARCH_DIRECTORIES** 此属性是一个路径列表，加载程序在查找通过 p/invoke 调用的本机库时应使用这些路径进行探测。
- **PLATFORM_RESOURCE_ROOTS** 此列表包含的路径用于探测资源附属程序集（在区域性特定的子目录中）。

在此示例主机中，TPA 列表是通过简单列出当前目录中的所有库来进行构造的：

```

void BuildTpaList(const char* directory, const char* extension, std::string& tpaList)
{
    // This will add all files with a .dll extension to the TPA list.
    // This will include unmanaged assemblies (coreclr.dll, for example) that don't
    // belong on the TPA list. In a real host, only managed assemblies that the host
    // expects to load should be included. Having extra unmanaged assemblies doesn't
    // cause anything to fail, though, so this function just enumerates all dll's in
    // order to keep this sample concise.
    std::string searchPath(directory);
    searchPath.append(FS_SEPARATOR);
    searchPath.append("*");
    searchPath.append(extension);

    WIN32_FIND_DATAA findData;
    HANDLE fileHandle = FindFirstFileA(searchPath.c_str(), &findData);

    if (fileHandle != INVALID_HANDLE_VALUE)
    {
        do
        {
            // Append the assembly to the list
            tpaList.append(directory);
            tpaList.append(FS_SEPARATOR);
            tpaList.append(findData.cFileName);
            tpaList.append(PATH_DELIMITER);

            // Note that the CLR does not guarantee which assembly will be loaded if an assembly
            // is in the TPA list multiple times (perhaps from different paths or perhaps with different
            NI/NI.dll
            // extensions. Therefore, a real host should probably add items to the list in priority order and
            only
            // add a file if it's not already present on the list.
            //
            // For this simple sample, though, and because we're only loading TPA assemblies from a single
            path,
            // and have no native images, we can ignore that complication.
        }
        while (FindNextFileA(fileHandle, &findData));
        FindClose(fileHandle);
    }
}

```

因为该示例简单，所以只需要 `TRUSTED_PLATFORM_ASSEMBLIES` 属性：

```

// Define CoreCLR properties
// Other properties related to assembly loading are common here,
// but for this simple sample, TRUSTED_PLATFORM_ASSEMBLIES is all
// that is needed. Check hosting documentation for other common properties.
const char* propertyKeys[] = {
    "TRUSTED_PLATFORM_ASSEMBLIES"      // Trusted assemblies
};

const char* propertyValues[] = {
    tpaList.c_str()
};

```

步骤 4 - 启动运行时

与 mscoree.h hosting API(上面所述)不同，CoreCLRHost.h API 使用一个调用启动运行时并创建默认的 AppDomain。`coreclr_initialize` 函数采用基本路径、名称和前面描述的属性，并通过 `hostHandle` 参数将图柄返回到主机。

```
void* hostHandle;
unsigned int domainId;

// This function both starts the .NET Core runtime and creates
// the default (and only) AppDomain
int hr = initializeCoreClr(
    runtimePath,           // App base path
    "SampleHost",          // AppDomain friendly name
    sizeof(propertyKeys) / sizeof(char*), // Property count
    propertyKeys,          // Property names
    propertyValues,        // Property values
    &hostHandle,           // Host handle
    &domainId);           // AppDomain ID
```

步骤 5 - 运行托管代码！

启动运行时之后，主机可以调用托管代码。这可以通过两种不同的方法实现。与本教程相关的示例代码使用 `coreclr_create_delegate` 函数创建静态托管方法的委托。此 API 采用程序集名称、符合命名空间条件的类型名称和方法名称作为输入，并返回可用于调用该方法的委托。

```
dowork_ptr managedDelegate;

// The assembly name passed in the third parameter is a managed assembly name
// as described at https://docs.microsoft.com/dotnet/framework/app-domains/assembly-names
hr = createManagedDelegate(
    hostHandle,
    domainId,
    "ManagedLibrary, Version=1.0.0.0",
    "ManagedLibrary.ManagedWorker",
    "DoWork",
    (void**)&managedDelegate);
```

在此示例中，主机现在可以调用 `managedDelegate` 来运行 `ManagedWorker.DoWork` 方法。

或者，可以使用 `coreclr_execute_assembly` 函数启动托管可执行文件。此 API 采用程序集路径和实参数组作为输入形参。它在该路径加载程序集并调用其主方法。

```
int hr = executeAssembly(
    hostHandle,
    domainId,
    argumentCount,
    arguments,
    "HelloWorld.exe",
    (unsigned int*)&exitCode);
```

步骤 6 - 关闭和清理

最后，主机完成运行托管代码后，使用 `coreclr_shutdown` 或 `coreclr_shutdown_2` 关闭 .NET Core 运行时。

```
hr = shutdownCoreClr(hostHandle, domainId);
```

CoreCLR 不支持重新初始化或卸载。请勿重新调用 `coreclr_initialize` 或卸载 CoreCLR 库。

使用 Mscoree.h 创建主机

如前所述，CoreClrHost.h 现在是承载 .NET Core 运行时的首选方法。但是，如果 CoreClrHost.h 接口不够用（例如，需要非标准的启动标志，或者在默认域上需要 AppDomainManager），仍然可以使用 `ICLRRuntimeHost4` 接口。这些说明将指导你使用 mscoree.h 执行承载 .NET Core 的操作。

CoreRun 主机 显示使用 mscoree.h 的更为复杂的真实托管示例。

有关 mscoree.h 的说明

ICLRRuntimeHost4 .NET Core 承载接口在 [MSCOREE.IDL](#) 中定义。主机需要引用的此文件 (mscoree.h) 的标头版本，该版本是在构建 .NET Core 运行时 时通过 MIDL 生成。如果不想构建 .NET Core 运行时，还可在 dotnet/runtime 存储库中将 mscoree.h 获取为预生成的标头。

步骤 1 - 标识托管的入口点

引用必要的标头后(例如, [mscoree.h](#) 和 stdio.h), .NET Core 主机必须完成的首要任务之一就是找到要使用的托管入口点。在示例主机中，通过将主机的第一个命令行参数作为托管的二进制文件(将执行该文件的 [main](#) 方法)的路径，即可完成此操作。

```
// The managed application to run should be the first command-line parameter.  
// Subsequent command line parameters will be passed to the managed app later in this host.  
wchar_t targetApp[MAX_PATH];  
GetFullPathNameW(argv[1], MAX_PATH, targetApp, NULL);
```

步骤 2 - 查找和加载 CoreCLR

.NET Core 运行时 API 位于 *CoreCLR.dll*(在 Windows 上)中。若要获取托管接口 ([ICLRRuntimeHost4](#))，就必须查找并加载 *CoreCLR.dll*。由主机定义用于规定 *CoreCLR.dll* 查找方式的约定。一些主机会预期文件位于一个常用的计算机范围内的位置(如 %programfiles%\dotnet\shared\Microsoft.NETCore.App\2.1.6)。其他主机会预期 *CoreCLR.dll* 从主机本身或要托管的应用旁的某个位置进行加载。还有一些主机可能会参考环境变量来查找库。

在 Linux 或 macOS 上，核心运行时库分别是 libcoreclr.so 或者 libcoreclr.dylib。

示例主机会为 *CoreCLR.dll* 探测几个常用位置。找到后，必须通过 [LoadLibrary](#) (在 Linux/macOS 上通过 [dlopen](#)) 进行加载。

```
HMODULE ret = LoadLibraryExW(coreDllPath, NULL, 0);
```

步骤 3 - 获取 ICLRRuntimeHost4 实例

通过在 [GetCLRRuntimeHost](#) 上调用 [GetProcAddress](#) (或在 Linux/macOS 上调用 [dlsym](#))，然后再调用该函数来检索 [ICLRRuntimeHost4](#) 托管接口。

```
ICLRRuntimeHost4* runtimeHost;  
  
FnGetCLRRuntimeHost pfnGetCLRRuntimeHost =  
    (FnGetCLRRuntimeHost)::GetProcAddress(coreCLRModule, "GetCLRRuntimeHost");  
  
if (!pfnGetCLRRuntimeHost)  
{  
    printf("ERROR - GetCLRRuntimeHost not found");  
    return -1;  
}  
  
// Get the hosting interface  
HRESULT hr = pfnGetCLRRuntimeHost(IID_ICLRRuntimeHost4, (IUnknown**)&runtimeHost);
```

步骤 4 - 设置启动标志并启动运行时

有了 [ICLRRuntimeHost4](#)，现在便可指定运行时范围内的启动标志并启动该运行时。启动标志决定要使用的垃圾回收器 (GC)(并发垃圾回收器或服务器)、是使用单个 AppDomain 还是多个 Appdomain，以及要使用的加载程序优化策略(对于非特定于域的程序集加载)。

```

hr = runtimeHost->SetStartupFlags(
    // These startup flags control runtime-wide behaviors.
    // A complete list of STARTUP_FLAGS can be found in mscoree.h,
    // but some of the more common ones are listed below.
    static_cast<STARTUP_FLAGS>(
        // STARTUP_FLAGS::STARTUP_SERVER_GC |           // Use server GC
        // STARTUP_FLAGS::STARTUP_LOADER_OPTIMIZATION_MULTI_DOMAIN | // Maximize domain-neutral loading
        // STARTUP_FLAGS::STARTUP_LOADER_OPTIMIZATION_MULTI_DOMAIN_HOST | // Domain-neutral loading for
        // strongly-named assemblies
        STARTUP_FLAGS::STARTUP_CONCURRENT_GC |           // Use concurrent GC
        STARTUP_FLAGS::STARTUP_SINGLE_APPDOMAIN |         // All code executes in the default AppDomain
                                                        // (required to use the runtimeHost-
        >ExecuteAssembly helper function)
        STARTUP_FLAGS::STARTUP_LOADER_OPTIMIZATION_SINGLE_DOMAIN // Prevents domain-neutral loading
    )
);

```

通过调用 `Start` 函数启动运行时。

```
hr = runtimeHost->Start();
```

步骤 5 - 准备 AppDomain 设置

启动运行时后，将需要设置 AppDomain。但创建 .NET AppDomain 时必须指定大量选项，因此必须先准备这些选项。

AppDomain 标志指定与安全性和互操作性相关的 AppDomain 行为。早期 Silverlight 主机对沙盒用户代码使用这些设置，但大多数现代 .NET Core 主机以完全信任的方式运行用户代码并启用互操作。

```

int appDomainFlags =
    // APPDOMAIN_FORCE_TRIVIAL_WAIT_OPERATIONS | // Do not pump messages during wait
    // APPDOMAIN_SECURITY_SANDBOXED |      // Causes assemblies not from the TPA list to be loaded as partially
    // trusted
    APPDOMAIN_ENABLE_PLATFORM_SPECIFIC_APPS |   // Enable platform-specific assemblies to run
    APPDOMAIN_ENABLE_PINVOKE_AND_CLASSIC_COMINTEROP | // Allow PInvoking from non-TPA assemblies
    APPDOMAIN_DISABLE_TRANSPARENCY_ENFORCEMENT;    // Entirely disables transparency checks

```

确定要使用的 AppDomain 标志后，必须定义 AppDomain 属性。该属性是字符串的键/值对。这些属性中的许多与 AppDomain 程序集的加载方式相关。

常见 AppDomain 属性包括：

- `TRUSTED_PLATFORM_ASSEMBLIES` 这是程序集路径列表（在 Windows 上，以 `;` 分隔；在 Linux/macOS 上，以 `:` 分隔），AppDomain 应优先加载它们，并完全信任它们（甚至是在部分信任的域中，也不例外）。此列表应包含“框架”程序集和其他受信任的模块，与 .NET Framework 方案中的 GAC 类似。一些主机会将任何库置于此列表上的 `coreclr.dll` 旁，其他主机具有硬编码的清单，其中列出了用于所需用途的受信任程序集。
- `APP_PATHS` 这是一个用来探测程序集的路径的列表（如果在受信任的平台程序集（TPA）列表中找不到程序集）。因为主机使用 TPA 列表可以更好地控制加载哪些程序集，所以对于主机来说，确定要加载的程序集并显式列出它们是最佳做法。但是，如果需要探测运行时，则此属性可以支持该方案。
- `APP_NI_PATHS` 此列表与 `APP_PATHS` 非常相似，不同之处在于其中的路径用于探测本机映像。
- `NATIVE_DLL_SEARCH_DIRECTORIES` 此属性是一个路径列表，加载程序在查找通过 `p/invoke` 调用的本机 DLL 时应使用这些路径进行探测。
- `PLATFORM_RESOURCE_ROOTS` 此列表包含的路径用于探测资源附属程序集（在区域性特定的子目录中）。

在 [简单示例主机](#) 中，这些属性将进行如下设置：

```
// TRUSTED_PLATFORM_ASSEMBLIES
```

```

// "Trusted Platform Assemblies" are prioritized by the loader and always loaded with full trust.
// A common pattern is to include any assemblies next to CoreCLR.dll as platform assemblies.
// More sophisticated hosts may also include their own Framework extensions (such as AppDomain managers)
// in this list.
size_t tpaSize = 100 * MAX_PATH; // Starting size for our TPA (Trusted Platform Assemblies) list
wchar_t* trustedPlatformAssemblies = new wchar_t[tpaSize];
trustedPlatformAssemblies[0] = L'\0';

// Extensions to probe for when finding TPA list files
const wchar_t *tpaExtensions[] =
{
    L"*.dll",
    L"*.exe",
    L"*.wimmd"
};

// Probe next to CoreCLR.dll for any files matching the extensions from tpaExtensions and
// add them to the TPA list. In a real host, this would likely be extracted into a separate function
// and perhaps also run on other directories of interest.
for (int i = 0; i < _countof(tpaExtensions); i++)
{
    // Construct the file name search pattern
    wchar_t searchPath[MAX_PATH];
    wcscpy_s(searchPath, MAX_PATH, coreRoot);
    wcscat_s(searchPath, MAX_PATH, L"\\");

    // Find files matching the search pattern
    WIN32_FIND_DATAW findData;
    HANDLE fileHandle = FindFirstFileW(searchPath, &findData);

    if (fileHandle != INVALID_HANDLE_VALUE)
    {
        do
        {
            // Construct the full path of the trusted assembly
            wchar_t pathToAdd[MAX_PATH];
            wcscpy_s(pathToAdd, MAX_PATH, coreRoot);
            wcscat_s(pathToAdd, MAX_PATH, L"\\");

            wcscat_s(pathToAdd, MAX_PATH, findData.cFileName);

            // Check to see if TPA list needs expanded
            if (wcsnlen(pathToAdd, MAX_PATH) + (3) + wcsnlen(trustedPlatformAssemblies, tpaSize) >= tpaSize)
            {
                // Expand, if needed
                tpaSize *= 2;
                wchar_t* newTPAList = new wchar_t[tpaSize];
                wcscpy_s(newTPAList, tpaSize, trustedPlatformAssemblies);
                trustedPlatformAssemblies = newTPAList;
            }

            // Add the assembly to the list and delimited with a semi-colon
            wcscat_s(trustedPlatformAssemblies, tpaSize, pathToAdd);
            wcscat_s(trustedPlatformAssemblies, tpaSize, L";");

            // Note that the CLR does not guarantee which assembly will be loaded if an assembly
            // is in the TPA list multiple times (perhaps from different paths or perhaps with different
NI/NI.dll
            // extensions. Therefore, a real host should probably add items to the list in priority order and
only
            // add a file if it's not already present on the list.
            //
            // For this simple sample, though, and because we're only loading TPA assemblies from a single
path,
            // we can ignore that complication.
        }
        while (FindNextFileW(fileHandle, &findData));
        FindClose(fileHandle);
    }
}

```

```

// APP_PATHS
// App paths are directories to probe in for assemblies which are not one of the well-known Framework
assemblies
// included in the TPA list.
//
// For this simple sample, we just include the directory the target application is in.
// More complex hosts may want to also check the current working directory or other
// locations known to contain application assets.
wchar_t appPaths[MAX_PATH * 50];

// Just use the targetApp provided by the user and remove the file name
wcscpy_s(appPaths, targetAppPath);

// APP_NI_PATHS
// App (NI) paths are the paths that will be probed for native images not found on the TPA list.
// It will typically be similar to the app paths.
// For this sample, we probe next to the app and in a hypothetical directory of the same name with 'NI'
// suffixed to the end.
wchar_t appNiPaths[MAX_PATH * 50];
wcscpy_s(appNiPaths, targetAppPath);
wcscat_s(appNiPaths, MAX_PATH * 50, L";");
wcscat_s(appNiPaths, MAX_PATH * 50, targetAppPath);
wcscat_s(appNiPaths, MAX_PATH * 50, L"NI");

// NATIVE_DLL_SEARCH_DIRECTORIES
// Native dll search directories are paths that the runtime will probe for native DLLs called via PInvoke
wchar_t nativeDllSearchDirectories[MAX_PATH * 50];
wcscpy_s(nativeDllSearchDirectories, appPaths);
wcscat_s(nativeDllSearchDirectories, MAX_PATH * 50, L";");
wcscat_s(nativeDllSearchDirectories, MAX_PATH * 50, coreRoot);

// PLATFORM_RESOURCE_ROOTS
// Platform resource roots are paths to probe in for resource assemblies (in culture-specific sub-directories)
wchar_t platformResourceRoots[MAX_PATH * 50];
wcscpy_s(platformResourceRoots, appPaths);

```

步骤 6 - 创建 AppDomain

所有 AppDomain 标志和属性都准备就绪后，可使用 `ICLRRuntimeHost4::CreateAppDomainWithManager` 设置 AppDomain。此函数选择性地采用完全限定的程序集名称和类型名称作为域的 AppDomain 管理器。AppDomain 管理器可允许主机控制 AppDomain 行为的某些方面，并且如果主机不打算直接调用用户代码，它可能会提供用于启动托管代码的入口点。

```

DWORD domainId;

// Setup key/value pairs for AppDomain properties
const wchar_t* propertyKeys[] = {
    L"TRUSTED_PLATFORM_ASSEMBLIES",
    L"APP_PATHS",
    L"APP_NI_PATHS",
    L"NATIVE_DLL_SEARCH_DIRECTORIES",
    L"PLATFORM_RESOURCE_ROOTS"
};

// Property values which were constructed in step 5
const wchar_t* propertyValues[] = {
    trustedPlatformAssemblies,
    appPaths,
    appNiPaths,
    nativeDllSearchDirectories,
    platformResourceRoots
};

// Create the AppDomain
hr = runtimeHost->CreateAppDomainWithManager(
    L"Sample Host AppDomain", // Friendly AD name
    appDomainFlags,
    NULL, // Optional AppDomain manager assembly name
    NULL, // Optional AppDomain manager type (including namespace)
    sizeof(propertyKeys) / sizeof(wchar_t*),
    propertyKeys,
    propertyValues,
    &domainId);

```

步骤 7 - 运行托管代码！

现在 AppDomain 启动并运行后，主机可以开始执行托管的代码。执行此操作的最简单方法是使用

`ICLRRuntimeHost4::ExecuteAssembly` 调用托管程序集的入口点方法。请注意，此函数仅适用于单一域方案。

```

DWORD exitCode = -1;
hr = runtimeHost->ExecuteAssembly(domainId, targetApp, argc - 1, (LPCWSTR*)(argc > 1 ? &argv[1] : NULL),
&exitCode);

```

如果 `ExecuteAssembly` 不满足主机的需要，那么另一种方法是使用 `CreateDelegate` 创建指向静态托管方法的函数指针。这要求主机知道要调用的方法的签名（以创建函数指针类型），但允许主机调用代码而不是程序集的入口点。第二个参数中提供的程序集名称是要加载的库的[完全托管程序集名称](#)。

```

void *pfnDelegate = NULL;
hr = runtimeHost->CreateDelegate(
    domainId,
    L"HW, Version=1.0.0.0, Culture=neutral", // Target managed assembly
    L"ConsoleApplication.Program", // Target managed type
    L"Main", // Target entry point (static method)
    (INT_PTR*)&pfnDelegate);

((MainMethodFp*)pfnDelegate)(NULL);

```

步骤 8 - 清理

最后，主机应随后通过卸载 Appdomain、停止运行时并释放 `ICLRRuntimeHost4` 引用来进行清理。

```
runtimeHost->UnloadAppDomain(domainId, true /* Wait until unload complete */);
runtimeHost->Stop();
runtimeHost->Release();
```

CoreCLR 不支持卸载。请勿卸载 CoreCLR 库。

结束语

构建主机后，可以通过从命令行运行主机并传递其所需的任何参数（例如，要运行用于 mscoree 示例主机的托管应用）来对主机进行测试。指定主机要运行的 .NET Core 应用时，请务必使用 `dotnet build` 生成的 .dll。

`dotnet publish` 为独立应用程序生成的可执行文件 (.exe 文件) 实际上是默认的 .NET Core 主机（以便可直接从主流方案中的命令行启动应用）；用户代码被编译为具有相同名称的 dll。

如果开始时操作不起作用，请再次检查 coreclr.dll 是否在主机预期的位置可用、TPA 列表中是否包含了所有必需的框架库以及 CoreCLR 的位数（32 位或 64 位）是否匹配主机的构建方式。

托管 .NET Core 运行时是高级方案，许多开发人员并不需要实施这一方案，但对于那些需要从本机进程启动托管代码的人员，或需要更好地控制 .NET Core 运行时的行为的人员而言，它会非常有用。

教程：创建项模板

2020/3/18 • [Edit Online](#)

使用 .NET Core，可以创建和部署可生成项目、文件甚至资源的模板。本教程是系列教程的第一部分，介绍如何创建、安装和卸载用于 `dotnet new` 命令的模板。

在本系列的这一部分中，你将了解如何：

- 为项模板创建类
- 创建模板配置文件夹和文件
- 从文件路径安装模板
- 测试项模板
- 卸载项模板

先决条件

- [.NET Core 2.2 SDK 或更高版本。](#)
 - 阅读参考文章[为 dotnet new 自定义模板。](#)
- 参考文章介绍了有关模板的基础知识，以及如何将它们组合在一起。其中一些信息将在本文中重复出现。
- 打开终端并导航到 working\templates 文件夹。

创建所需的文件夹

本系列使用包含模板源的“working 文件夹”和用于测试模板的“testing 文件夹”。working 文件夹和 testing 文件夹应位于同一父文件夹下。

首先，创建父文件夹，名称无关紧要。然后，创建一个名为“working”的子文件夹。在 working 文件夹内，创建一个名为“templates”的子文件夹。

接下来，在名为“test”的父文件夹下创建一个文件夹。文件夹结构应如下所示。

```
parent_folder
├── test
└── working
    └── templates
```

创建项模板

项模板是包含一个或多个文件的特定类型的模板。当你想要生成类似于配置、代码或解决方案文件的内容时，这些类型的模板非常有用。在本例中，你将创建一个类，该类将扩展方法添加到字符串类型中。

在终端中，导航到 working\templates 文件夹，并创建一个名为“extensions”的新子文件夹。进入文件夹。

```
working
└── templates
    └── extensions
```

创建一个名为“CommonExtensions.cs”的新文件，并使用你喜爱的文本编辑器打开它。此类将提供一个用于反转字符串内容的名为 `Reverse` 的扩展方法。粘贴以下代码并保存文件：

```
using System;

namespace System
{
    public static class StringExtensions
    {
        public static string Reverse(this string value)
        {
            var tempArray = value.ToCharArray();
            Array.Reverse(tempArray);
            return new string(tempArray);
        }
    }
}
```

现在你已经创建了模板的内容，需要在模板的根文件夹中创建模板配置。

创建模板配置

模板在 .NET Core 中通过模板根目录中的特殊文件夹和配置文件进行识别。在本教程中，你的模板文件夹位于 working\templates\extensions 。

创建模板时，除特殊配置文件夹外，模板文件夹中的所有文件和文件夹都作为模板的一部分包含在内。此配置文件夹名为“template.config”。

首先，创建一个名为“template.config”的新子文件夹，然后进入该文件夹。然后，创建一个名为“template.json”的新文件。文件夹结构应如下所示：

```
working
└── templates
    └── extensions
        └── .template.config
            template.json
```

使用你喜爱的文本编辑器打开 template.json 并粘贴以下 JSON 代码，然后保存。

```
{
    "$schema": "http://json.schemastore.org/template",
    "author": "Me",
    "classifications": [ "Common", "Code" ],
    "identity": "ExampleTemplate.StringExtensions",
    "name": "Example templates: string extensions",
    "shortName": "stringext",
    "tags": {
        "language": "C#",
        "type": "item"
    }
}
```

此配置文件包含模板的所有设置。可以看到基本设置，例如 `name` 和 `shortName`，除此之外，还有一个设置为 `item` 的 `tags/type` 值。这会将你的模板归类为项模板。你创建的模板类型不存在限制。`item` 和 `project` 值是 .NET Core 建议使用的通用名称，便于用户轻松筛选正在搜索的模板类型。

`classifications` 项表示你在运行 `dotnet new` 并获取模板列表时看到的“标记”列。用户还可以根据分类标记进行搜索。不要将 `*.json` 文件中的 `tags` 属性与 `classifications` 标记列表混淆。它们虽然具有类似的名称，但截然不同。`template.json` 文件的完整架构位于 [JSON 架构存储](#)。有关 `template.json` 文件的详细信息，请参阅 [dotnet 创建模板 wiki](#)。

现在你已有一个有效的 `.template.config/template.json` 文件，可以安装模板了。在终端中，导航到 `extensions` 文件

夹，并运行以下命令以安装位于当前文件夹的模板：

- 在 Windows 上: `dotnet new -i .\`
- 在 Linux 或 macOS 上: `dotnet new -i ./`

此命令输出安装的模板列表，其中应包括你的模板。

```
C:\working\templates\extensions> dotnet new -i .\
Usage: new [options]

Options:
-h, --help           Displays help for this command.
-l, --list            Lists templates containing the specified name. If no name is specified, lists all
templates.

... cut to save space ...

Templates          Short Name      Language      Tags
-----
-----
Example templates: string extensions    stringext     [C#]          Common/Code
Console Application                 console       [C#], F#, VB   Common/Console
Class library                      classlib      [C#], F#, VB   Common/Library
WPF Application                    wpf          [C#], VB      Common/WPF
Windows Forms (WinForms) Application winforms     [C#], VB      Common/WinForms
Worker Service                     worker       [C#]          Common/Worker/Web
```

测试项模板

现在你已安装了项模板，可对其进行测试。导航到 test/ 文件夹，使用 `dotnet new console` 创建新的控制台应用程序。这将生成一个可以使用 `dotnet run` 命令轻松测试的工作项目。

```
dotnet new console
```

将获得类似于下面的输出。

```
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on C:\test\test.csproj...
Restore completed in 54.82 ms for C:\test\test.csproj.

Restore succeeded.
```

运行该项目。

```
dotnet run
```

将获得以下输出。

```
Hello World!
```

接下来，运行 `dotnet new stringext` 以从模板生成 CommonExtensions.cs。

```
dotnet new stringext
```

将获得以下输出。

```
The template "Example templates: string extensions" was created successfully.
```

更改 Program.cs 中的代码以使用模板提供的扩展方法反转 "Hello World" 字符串。

```
Console.WriteLine("Hello World!".Reverse());
```

再次运行程序，将看到结果已反转。

```
dotnet run
```

将获得以下输出。

```
!dlroW olleH
```

祝贺你！你已使用 .NET Core 创建并部署了项模板。为准备学习本系列教程的下一部分，必须卸载已创建的模板。确保同时删除 test 文件夹中的所有文件。这将回到干净状态，为本教程的下一个主要部分做好准备。

卸载模板

由于模板是按文件路径安装的，因此，必须使用绝对文件路径将其卸载。可以通过运行 `dotnet new -u` 命令看到已安装的模板列表。你的模板应列在最后。使用列出的路径，通过执行

```
dotnet new -u <ABSOLUTE PATH TO TEMPLATE DIRECTORY> 命令卸载模板。
```

```
dotnet new -u
```

将获得类似于下面的输出。

```
Template Instantiation Commands for .NET Core CLI

Currently installed items:
Microsoft.DotNet.Common.ItemTemplates
Templates:
    dotnet gitignore file (gitignore)
    global.json file (globaljson)
    NuGet Config (nugetconfig)
    Solution File (sln)
    Dotnet local tool manifest file (tool-manifest)
    Web Config (webconfig)

... cut to save space ...

NUnit3.DotNetNew.Template
Templates:
    NUnit 3 Test Project (nunit) C#
    NUnit 3 Test Item (nunit-test) C#
    NUnit 3 Test Project (nunit) F#
    NUnit 3 Test Item (nunit-test) F#
    NUnit 3 Test Project (nunit) VB
    NUnit 3 Test Item (nunit-test) VB

C:\working\templates\extensions
Templates:
    Example templates: string extensions (stringext) C#
```

若要卸载该模板，请运行以下命令。

```
dotnet new -u C:\working\templates\extensions
```

后续步骤

在本教程中，你创建了一个项模板。若要了解如何创建项目模板，请继续学习本系列教程。

[创建项目模板](#)

教程：创建项目模板

2020/3/18 • [Edit Online](#)

使用 .NET Core，可以创建和部署可生成项目、文件甚至资源的模板。本教程是系列教程的第二部分，介绍如何创建、安装和卸载用于 `dotnet new` 命令的模板。

在本系列的这一部分中，你将了解如何：

- 创建项目模板的资源
- 创建模板配置文件夹和文件
- 从文件路径安装模板
- 测试项模板
- 卸载项模板

先决条件

- 完成本系列教程的[第 1 部分](#)。
- 打开终端并导航到 `working\templates` 文件夹。

创建项目模板

项目模板生成可立即运行的项目，使用户可以轻松地使用一组有效的代码。.NET Core 包含一些项目模板，例如控制台应用程序或类库。在本例中，你将创建一个新的控制台项目，该项目启用 C# 8.0 并生成 `async main` 入口点。

在终端中，导航到 `working\templates` 文件夹，并创建一个名为“`consoleasync`”的新子文件夹。进入子文件夹，并运行 `dotnet new console` 以生成标准控制台应用程序。将编辑此模板生成的文件以创建新模板。

```
working
└── templates
    └── consoleasync
        consoleasync.csproj
        Program.cs
```

修改 `Program.cs`

打开 `Program.cs` 文件。控制台项目不使用异步入口点，我们来添加它。将代码更改为以下内容并保存文件。

```
using System;
using System.Threading.Tasks;

namespace consoleasync
{
    class Program
    {
        static async Task Main(string[] args)
        {
            await Console.Out.WriteLineAsync("Hello World with C# 8.0!");
        }
    }
}
```

修改 consoleasync.csproj

将项目使用的 C# 语言版本更新到 8.0 版。编辑 `consoleasync.csproj` 文件并将 `<LangVersion>` 设置添加到 `<PropertyGroup>` 节点。

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>netcoreapp2.2</TargetFramework>

  <LangVersion>8.0</LangVersion>
</PropertyGroup>

</Project>
```

生成项目

在完成项目模板之前，应对其进行测试，确保它能够正确编译和运行。

在终端中，运行以下命令。

```
dotnet run
```

将获得以下输出。

```
Hello World with C# 8.0!
```

可以使用 `dotnet run` 删除已创建的 `obj` 和 `bin` 文件夹。删除这些文件可确保你的模板仅包含与模板相关的文件，而不包含生成操作产生的任何文件。

现在你已经创建了模板的内容，需要在模板的根文件夹中创建模板配置。

创建模板配置

模板在 .NET Core 中通过模板根目录中的特殊文件夹和配置文件进行识别。在本教程中，你的模板文件夹位于 `working\templates\consoleasync`。

创建模板时，除特殊配置文件夹外，模板文件夹中的所有文件和文件夹都作为模板的一部分包含在内。此配置文件夹名为`".template.config"`。

首先，创建一个名为`".template.config"` 的新子文件夹，然后进入该文件夹。然后，创建一个名为`"template.json"` 的新文件。文件夹结构应如下所示。

```
working
└── templates
    └── consoleasync
        └── .template.config
            template.json
```

使用你喜爱的文本编辑器打开 `template.json` 并粘贴以下 json 代码，然后保存。

```
{
    "$schema": "http://json.schemastore.org/template",
    "author": "Me",
    "classifications": [ "Common", "Console", "C#8" ],
    "identity": "ExampleTemplate.AsyncProject",
    "name": "Example templates: async project",
    "shortName": "consoleasync",
    "tags": {
        "language": "C#",
        "type": "project"
    }
}
```

此配置文件包含模板的所有设置。可以看到基本设置，例如 `name` 和 `shortName`，除此之外，还有一个设置为 `project` 的 `tags/type` 值。这会将模板指定为项目模板。你创建的模板类型不存在限制。`item` 和 `project` 值是 .NET Core 建议使用的通用名称，便于用户轻松筛选正在搜索的模板类型。

`classifications` 项表示你在运行 `dotnet new` 并获取模板列表时看到的“标记”列。用户还可以根据分类标记进行搜索。不要将 json 文件中的 `tags` 属性与 `classifications` 标记列表混淆。它们虽然具有类似的名称，但截然不同。`template.json` 文件的完整架构位于 [JSON 架构存储](#)。有关 `template.json` 文件的详细信息，请参阅 [dotnet 创建模板 wiki](#)。

现在你已有一个有效的 `.template.config/template.json` 文件，可以安装模板了。在安装模板之前，请务必删除无需在模板中包含的任何额外文件夹和文件，例如 `bin` 或 `obj` 文件夹。在终端中，导航到 `consoleasync` 文件夹，并运行 `dotnet new -i .\` 以安装位于当前文件夹的模板。如果使用的是 Linux 或 macOS 操作系统，请使用正斜杠：
`dotnet new -i ./`

此命令输出安装的模板列表，其中应包括你的模板。

```
dotnet new -i .\
```

将获得类似于下面的输出。

```
Usage: new [options]

Options:
  -h, --help            Displays help for this command.
  -l, --list             Lists templates containing the specified name. If no name is specified, lists all
                        templates.

... cut to save space ...

Templates          Short Name      Language      Tags
-----
Console Application      console      [C#], F#, VB  Common/Console
Example templates: async project  consoleasync  [C#]       Common/Console/C#8
Class library           classlib     [C#], F#, VB  Common/Library
WPF Application         wpf        [C#], VB    Common/WPF
Windows Forms (WinForms) Application  winforms   [C#], VB    Common/WinForms
Worker Service          worker     [C#]       Common/Worker/Web
```

测试项目模板

现在你已安装了项模板，可对其进行测试。

1. 导航到 `test` 文件夹
2. 使用以下命令创建一个新的控制台应用程序，该命令生成可使用 `dotnet run` 命令轻松测试的工作项目。

```
dotnet new consoleasync
```

将获得以下输出。

```
The template "Example templates: async project" was created successfully.
```

3. 请使用以下命令运行项目。

```
dotnet run
```

将获得以下输出。

```
Hello World with C# 8.0!
```

祝贺你！你已使用 .NET Core 创建并部署了项目模板。为准备学习本系列教程的下一部分，必须卸载已创建的模板。确保同时删除 test 文件夹中的所有文件。这将回到干净状态，为本教程的下一个主要部分做好准备。

卸载模板

由于模板是使用文件路径安装的，因此，必须使用绝对文件路径将其卸载。可以通过运行 `dotnet new -u` 命令看到已安装的模板列表。你的模板应列在最后。使用列出的路径，通过执行

```
dotnet new -u <ABSOLUTE PATH TO TEMPLATE DIRECTORY> 命令卸载模板。
```

```
dotnet new -u
```

将获得类似于下面的输出。

```
Template Instantiation Commands for .NET Core CLI

Currently installed items:
  Microsoft.DotNet.Common.ItemTemplates
    Templates:
      dotnet gitignore file (gitignore)
      global.json file (globaljson)
      NuGet Config (nugetconfig)
      Solution File (sln)
      Dotnet local tool manifest file (tool-manifest)
      Web Config (webconfig)

... cut to save space ...

NUnit3.DotNetNew.Template
  Templates:
    NUnit 3 Test Project (nunit) C#
    NUnit 3 Test Item (nunit-test) C#
    NUnit 3 Test Project (nunit) F#
    NUnit 3 Test Item (nunit-test) F#
    NUnit 3 Test Project (nunit) VB
    NUnit 3 Test Item (nunit-test) VB
  C:\working\templates\consoleasync
    Templates:
      Example templates: async project (consoleasync) C#
```

若要卸载该模板，请运行以下命令。

```
dotnet new -u C:\working\templates\consoleasync
```

后续步骤

在本教程中，你创建了一个项目模板。若要了解如何将项模板和项目模板打包为易于使用的文件，请继续学习本教程系列。

[创建模板包](#)

教程：创建模板包

2020/3/18 • [Edit Online](#)

使用 .NET Core，可以创建和部署可生成项目、文件甚至资源的模板。本教程是系列教程的第三部分，介绍如何创建、安装和卸载用于 `dotnet new` 命令的模板。

在本系列的这一部分中，你将了解如何：

- 创建一个 *.csproj 项目以生成模板包
- 配置项目文件以进行打包
- 从 NuGet 包文件安装模板
- 按包 ID 卸载模板

先决条件

- 完成本系列教程的[第 1 部分](#)和[第 2 部分](#)。

本教程使用本教程前两部分中创建的两个模板。只要将不同的模板作为文件夹复制到 `working\templates\` 文件夹中，就可以使用该模板。

- 打开终端并导航到 `working\` 文件夹。

创建模板包项目

模板包是打包到文件中的一个或多个模板。安装或卸载程序包时，将分别添加或删除包中包含的所有模板。本系列教程的前几部分仅适用于各自的模板。若要共享非打包模板，必须复制模板文件夹并通过该文件夹进行安装。由于模板包中可以包含多个模板，并且是单个文件，因此共享更容易。

模板包由 NuGet 包 (.nupkg) 文件表示。并且，与任何 NuGet 包一样，可以将模板包上传到 NuGet 源。

`dotnet new -i` 命令支持从 NuGet 包源安装模板包。此外，可以直接从 .nupkg 文件安装模板包。

通常情况下，使用 C# 项目文件来编译代码并生成二进制文件。但是，该项目也可用于生成模板包。通过更改 .csproj 的设置，可以阻止它编译任何代码，而是将模板的所有资产都包含在内作为资源。生成此项目后，它会生成模板包 NuGet 包。

将要创建的包将包含先前创建的[项模板](#)和[包模板](#)。由于我们将两个模板分组到 `working\templates\` 文件夹中，因此可以使用 .csproj 文件的 `working` 文件夹。

在终端中，导航到 `working` 文件夹。创建一个新项目，将名称设置为 `templatepack`，并将输出文件夹设置为当前文件夹。

```
dotnet new console -n templatepack -o .
```

`-n` 参数将 .csproj 文件名设置为 `templatepack.csproj`。`-o` 参数将在当前目录中创建文件。应看到类似于以下输出的结果。

```
dotnet new console -n templatepack -o .
```

```
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on .\templatepack.csproj...
Restore completed in 52.38 ms for C:\working\templatepack.csproj.

Restore succeeded.
```

接下来，在你喜爱的编辑器中打开 templatepack.csproj 文件，并将内容替换为以下 XML：

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
  <PackageType>Template</PackageType>
  <PackageVersion>1.0</PackageVersion>
  <PackageId>AdatumCorporation.Utility.Templates</PackageId>
  <Title>AdatumCorporation Templates</Title>
  <Authors>Me</Authors>
  <Description>Templates to use when creating an application for Adatum Corporation.</Description>
  <PackageTags>dotnet-new;templates;contoso</PackageTags>

  <TargetFramework>netstandard2.0</TargetFramework>

  <IncludeContentInPack>true</IncludeContentInPack>
  <IncludeBuildOutput>false</IncludeBuildOutput>
  <ContentTargetFolders>content</ContentTargetFolders>
</PropertyGroup>

<ItemGroup>
  <Content Include="templates\**\*" Exclude="templates\**\bin\**;templates\**\obj\**" />
  <Compile Remove="**\**" />
</ItemGroup>

</Project>
```

上面的 XML 中的 `<PropertyGroup>` 设置分为三组。第一组处理 NuGet 包所需的属性。三个 `<Package>` 设置与 NuGet 包属性相关，用于在 NuGet 源上标识你的包。具体来说，`<PackageId>` 值用于通过单个名称而不是目录路径卸载模板包。它还可用于从 NuGet 源安装模板包。其余的设置，如 `<Title>` 和 `<PackageTags>`，它们与 NuGet 源上显示的元数据相关。有关 NuGet 设置的详细信息，请参阅 [NuGet 和 MSBuild 属性](#)。

必须设置 `<TargetFramework>` 设置，以便在运行 pack 命令编译和打包项目时 MSBuild 正常运行。

最后三个设置与正确配置项目有关，以便在创建 NuGet 包时将模板包含在该包中的相应文件夹中。

`<ItemGroup>` 包含两个设置。首先，`<Content>` 设置的内容包含 templates 文件夹中的所有内容。它还设置为排除任何 bin 文件夹或 obj 文件夹，以防止包含任何已编译的代码（如果已测试和编译模板）。其次，`<Compile>` 设置将所有代码文件排除在编译范围之外，无论它们位于何处都是如此。此设置可阻止用于创建模板包的项目尝试编译 templates 文件夹层次结构中的代码。

生成和安装

保存此文件，然后运行 pack 命令

```
dotnet pack
```

此命令将生成项目并在其中创建一个 NuGet 包，具体应为 working\bin\Debug 文件夹。

```
dotnet pack
```

```
Microsoft (R) Build Engine version 16.2.0-preview-19278-01+d635043bd for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
Restore completed in 123.86 ms for C:\working\templatepack.csproj.
```

```
templatepack -> C:\working\bin\Debug\netstandard2.0\templatepack.dll
Successfully created package 'C:\working\bin\Debug\AdatumCorporation.Utility.Templates.1.0.0.nupkg'.
```

接下来，使用 `dotnet new -i PATH_TO_NUPKG_FILE` 命令安装模板包文件。

```
C:\working> dotnet new -i C:\working\bin\Debug\AdatumCorporation.Utility.Templates.1.0.0.nupkg
Usage: new [options]

Options:
  -h, --help           Displays help for this command.
  -l, --list            Lists templates containing the specified name. If no name is specified, lists all
                        templates.

... cut to save space ...



| Templates                            | Short Name   | Language     | Tags               |
|--------------------------------------|--------------|--------------|--------------------|
| -----                                |              |              |                    |
| Example templates: string extensions | stringext    | [C#]         | Common/Code        |
| Console Application                  | console      | [C#], F#, VB | Common/Console     |
| Example templates: async project     | consoleasync | [C#]         | Common/Console/C#8 |
| Class library                        | classlib     | [C#], F#, VB | Common/Library     |


```

如果将 NuGet 包上传到 NuGet 源，可以使用 `dotnet new -i PACKAGEID` 命令，其中，`PACKAGEID` 与 `.csproj` 文件中的 `<PackageId>` 设置相同。此包 ID 与 NuGet 包标识符相同。

卸载模板包

无论如何安装模板包，即无论是直接使用 `.nupkg` 文件还是通过 NuGet 源安装，删除模板包的操作都是一样的。使用要卸载的模板的 `<PackageId>`。可以通过运行 `dotnet new -u` 命令获取已安装的模板列表。

```
dotnet new -u
```

Template Instantiation Commands for .NET Core CLI

```
Currently installed items:  
Microsoft.DotNet.Common.ItemTemplates  
Templates:  
    dotnet gitignore file (gitignore)  
    global.json file (globaljson)  
    NuGet Config (nugetconfig)  
    Solution File (sln)  
    Dotnet local tool manifest file (tool-manifest)  
    Web Config (webconfig)  
  
... cut to save space ...  
  
NUnit3.DotNetNew.Template  
Templates:  
    NUnit 3 Test Project (nunit) C#  
    NUnit 3 Test Item (nunit-test) C#  
    NUnit 3 Test Project (nunit) F#  
    NUnit 3 Test Item (nunit-test) F#  
    NUnit 3 Test Project (nunit) VB  
    NUnit 3 Test Item (nunit-test) VB  
AdatumCorporation.Utility.Templates  
Templates:  
    Example templates: async project (consoleasync) C#  
    Example templates: string extensions (stringext) C#
```

运行 `dotnet new -u AdatumCorporation.Utility.Templates` 以卸载模板。`dotnet new` 命令将输出应忽略先前安装的模板的帮助信息。

祝贺你！你已安装，并卸载了模板包。

后续步骤

若要了解有关模板的详细信息(你已经了解了大部分相关信息)，请参阅[为 dotnet new 自定义模板](#)一文。

- [dotnettemplating GitHub 存储库 Wiki](#)
- [dotnet/dotnet-template-samples GitHub 存储库](#)
- [JSON 架构存储中的 template.json 架构](#)

教程：使用 .NET Core CLI 创建 .NET Core 工具

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.1 SDK 及更高版本

本教程介绍如何创建和打包 .NET Core 工具。使用 .NET Core CLI，可以创建一个控制台应用程序作为工具，便于其他人安装并运行。.NET Core 工具是从 .NET Core CLI 安装的 NuGet 包。有关工具的详细信息，请参阅 [.NET Core 工具概述](#)。

将创建的工具是一个控制台应用程序，它将消息作为输入，并显示消息以及用于创建机器人图像的文本行。

这是一系列教程(包含三个教程)中的第一个。在本教程中，将创建并打包工具。在接下来的两个教程中，[使用该工具作为全局工具并使用该工具作为本地工具](#)。

先决条件

- [.NET Core SDK 3.1](#) 或更高版本。

本教程和下面的[全局工具教程](#)适用于 .NET Core SDK 2.1 及更高版本，因为全局工具从该版本开始可用。但本教程假定你已安装 3.1 或更高版本，因此你可以选择继续学习[本地工具教程](#)。本地工具从 .NET Core SDK 3.0 开始可用。无论你是将工具用作全局工具还是用作本地工具，创建工具的过程都是相同的。

- 按需选择的文本编辑器或代码编辑器。

创建项目

1. 打开命令提示符，创建一个名为“repository”的文件夹。

2. 导航到“repository”文件夹并输入以下命令：

```
dotnet new console -n microsoft.botsay
```

此命令将在“repository”文件夹下创建一个名为“microsoft.botsay”的新文件夹。

3. 导航到“microsoft.botsay”文件夹。

```
cd microsoft.botsay
```

添加代码

1. 使用代码编辑器打开 `Program.cs` 文件。

2. 将下面的 `using` 指令添加到文件的顶部：

```
using System.Reflection;
```

3. 将 `Main` 方法替换为以下代码，以便处理应用程序的命令行参数。

```
static void Main(string[] args)
{
    if (args.Length == 0)
    {
        var versionString = Assembly.GetEntryAssembly()
            .GetCustomAttribute<AssemblyInformationalVersionAttribute>()
            .InformationalVersion
            .ToString();

        Console.WriteLine($"botsay {versionString}");
        Console.WriteLine("-----");
        Console.WriteLine("\nUsage:");
        Console.WriteLine("  botsay <message>");
        return;
    }

    ShowBot(string.Join(' ', args));
}
```

如果未传递任何参数，将显示简短的帮助消息。否则，所有参数都将连接到单个字符串中，并通过调用在下一步中创建的 `ShowBot` 方法进行打印。

4. 添加一个名为 `ShowBot` 的新方法，该方法采用一个字符串参数。该方法使用文本行打印出消息和机器人图像。

5. 保存更改。

测试应用程序

运行项目并观察输出。尝试使用命令行处的这些变体来查看不同的结果：

```
dotnet run  
dotnet run -- "Hello from the bot"  
dotnet run -- Hello from the bot
```

位于 `--` 分隔符后的所有参数均会传递给应用程序。

打包工具

在将应用程序作为工具打包并分发之前，你需要修改项目文件。

1. 打开“microsoft.botsay.csproj”文件，然后将三个新的 XML 节点添加到 `<PropertyGroup>` 节点的末尾：

```
<PackAsTool>true</PackAsTool>
<ToolCommandName>botsay</ToolCommandName>
<PackageOutputPath>./nupkg</PackageOutputPath>
```

`<ToolCommandName>` 是一个可选元素，用于指定在安装工具后将调用该工具的命令。如果未提供此元素，则该工具的命令名称为没有“.csproj”扩展名的项目文件名。

`<PackageOutputPath>` 是一个可选元素，用于确定将在何处生成 NuGet 包。NuGet 包是.NET Core CLI 用于安装你的工具的包。

项目文件现在类似于以下示例：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>

    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>

    <PackAsTool>true</PackAsTool>
    <ToolCommandName>botsay</ToolCommandName>
    <PackageOutputPath>./nupkg</PackageOutputPath>

  </PropertyGroup>

</Project>
```

2. 通过运行 `dotnet pack` 命令创建 NuGet 包：

```
dotnet pack
```

“microsoft.botsay.1.0.0.nupkg”文件在由 `microsoft.botsay.csproj` 文件的 `<PackageOutputPath>` 值标识的文件夹中创建，在本示例中为“./nupkg”文件夹。

如果想要公开发布一个工具，你可以将其上传到 <https://www.nuget.org>。该工具在 NuGet 上可用后，开发人员就可以使用 `dotnet tool install` 命令安装该工具。在本教程中，你将直接从本地“nupkg”文件夹安装包，因此无需将包上传到 NuGet。

疑难解答

如果在学习本教程时收到错误消息，请参阅[排查 .NET Core 工具使用问题](#)。

后续步骤

在本教程中，你创建了一个控制台应用程序并将其打包为工具。若要了解如何使用该工具作为全局工具，请转到下一教程。

[安装和使用全局工具](#)

如果需要，可以跳过全局工具教程，直接转到本地工具教程。

[安装和使用本地工具](#)

教程：使用 .NET Core CLI 安装和使用 .NET Core 全局工具

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.1 SDK 及更高版本

本教程介绍如何安装和使用全局工具。使用在[本系列的第一个教程](#)中创建的工具。

先决条件

- 完成[本系列的第一个教程](#)。

使用该工具作为全局工具

1. 通过运行 microsoft.botsay 项目文件夹中的 `dotnet tool install` 命令，从包中安装该工具：

```
dotnet tool install --global --add-source ./nupkg microsoft.botsay
```

`--global` 参数指示 .NET Core CLI 将工具二进制文件安装在自动添加到 PATH 环境变量的默认位置中。

`--add-source` 参数指示 .NET Core CLI 临时使用 `./nupkg` 目录作为 NuGet 包的附加源数据源。为包提供了唯一名称，以确保它仅位于 `./nupkg` 目录中，而不是在 Nuget.org 站点上。

输出显示用于调用该工具和已安装的版本的命令：

```
You can invoke the tool using the following command: botsay
Tool 'microsoft.botsay' (version '1.0.0') was successfully installed.
```

2. 调用该工具：

```
botsay hello from the bot
```

NOTE

如果此命令失败，则可能需要打开新终端来刷新 PATH。

3. 通过运行 `dotnet tool uninstall` 命令来删除该工具：

```
dotnet tool uninstall -g microsoft.botsay
```

使用该工具作为自定义位置中安装的全局工具

1. 从包中安装该工具。

在 Windows 上：

```
dotnet tool install --tool-path c:\dotnet-tools --add-source ./nupkg microsoft.botsay
```

在 Linux 或 macOS 上：

```
dotnet tool install --tool-path ~/bin --add-source ./nupkg microsoft.botsay
```

--tool-path 参数指示 .NET Core CLI 将工具二进制文件安装在指定位置中。如果目录不存在，则会创建该目录。此目录不会自动添加到 PATH 环境变量中。

输出显示用于调用该工具和已安装的版本的命令：

```
You can invoke the tool using the following command: botsay  
Tool 'microsoft.botsay' (version '1.0.0') was successfully installed.
```

2. 调用该工具：

在 Windows 上：

```
c:\dotnet-tools\botsay hello from the bot
```

在 Linux 或 macOS 上：

```
~/bin/botsay hello from the bot
```

3. 通过运行 [dotnet tool uninstall](#) 命令来删除该工具：

在 Windows 上：

```
dotnet tool uninstall --tool-path c:\dotnet-tools microsoft.botsay
```

在 Linux 或 macOS 上：

```
dotnet tool uninstall --tool-path ~/bin microsoft.botsay
```

疑难解答

如果在学习本教程时收到错误消息，请参阅[排查 .NET Core 工具使用问题](#)。

后续步骤

在本教程中，已将工具作为全局工具安装和使用。若要安装和使用与本地工具相同的工具，请转到下一教程。

[安装和使用本地工具](#)

教程：使用 .NET Core CLI 安装和使用 .NET Core 本地工具

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 3.0 SDK 及更高版本

本教程介绍如何安装和使用本地工具。使用在[本系列的第一个教程](#)中创建的工具。

先决条件

- 完成[本系列的第一个教程](#)。
- 安装 .NET Core 2.1 运行时。

在本教程中，安装和使用面向 .NET Core 2.1 的工具，因此需要在计算机上安装该运行时。若要安装 2.1 运行时，请转到[.NET Core 2.1 下载页面](#)并在“运行应用 - 运行时”列中查找运行时安装链接。

创建清单文件

若要安装仅用于本地访问的工具（对于当前目录和子目录），必须将其添加到清单文件。

在“microsoft.botsay”文件夹中，向上导航一个级别到“repository”文件夹：

```
cd ..
```

通过运行 `dotnet new` 命令来创建清单文件：

```
dotnet new tool-manifest
```

输出指示文件创建成功。

```
The template "Dotnet local tool manifest file" was created successfully.
```

.config/dotnet-tools.json 文件中尚无工具：

```
{
  "version": 1,
  "isRoot": true,
  "tools": {}
}
```

清单文件中列出的工具可用于当前目录和子目录。当前目录是包含具有清单文件的 .config 目录的目录。

使用引用本地工具的 CLI 命令时，SDK 会在当前目录和父目录中搜索清单文件。如果它找到清单文件，但该文件不包含所引用的工具，则会通过父目录继续向上搜索。搜索在找到所引用的工具或找到将 `isRoot` 设置为 `true` 的清单文件时结束。

将 botsay 作为本地工具安装

从在第一个教程中创建的包中安装该工具：

```
dotnet tool install --add-source ./microsoft.botsay/nupkg microsoft.botsay
```

此命令将该工具添加到在上一步中创建的清单文件。命令输出显示新安装的工具所在的清单文件：

```
You can invoke the tool from this directory using the following command:  
'dotnet tool run botsay' or 'dotnet botsay'  
Tool 'microsoft.botsay' (version '1.0.0') was successfully installed.  
Entry is added to the manifest file /home/name/repository/.config/dotnet-tools.json
```

.config/dotnet-tools.json 文件现在有一个工具：

```
{  
  "version": 1,  
  "isRoot": true,  
  "tools": {  
    "microsoft.botsay": {  
      "version": "1.0.0",  
      "commands": [  
        "botsay"  
      ]  
    }  
  }  
}
```

使用该工具

通过运行“repository”文件夹中的 `dotnet tool run` 命令来调用该工具：

```
dotnet tool run botsay hello from the bot
```

还原其他人安装的本地工具

通常将本地工具安装在存储库的根目录中。将清单文件签入到存储库后，其他开发人员可以获得最新的清单文件。若要安装清单文件中列出的所有工具，他们可以运行单个 `dotnet tool restore` 命令。

1. 打开 .config/dotnet-tools.json 文件并将内容替换为以下 JSON：

```
{  
  "version": 1,  
  "isRoot": true,  
  "tools": {  
    "microsoft.botsay": {  
      "version": "1.0.0",  
      "commands": [  
        "botsay"  
      ]  
    },  
    "dotnetsay": {  
      "version": "2.1.3",  
      "commands": [  
        "dotnetsay"  
      ]  
    }  
  }  
}
```

2. 将 `<name>` 替换为用于创建项目的名称。

3. 保存更改。

进行此更改等同于在其他人安装项目目录的包 `dotnetsay` 后从存储库获取最新版本。

4. 运行 `dotnet tool restore` 命令。

```
dotnet tool restore
```

该命令生成的输出如以下示例所示：

```
Tool 'microsoft.botsay' (version '1.0.0') was restored. Available commands: botsay
Tool 'dotnetsay' (version '2.1.3') was restored. Available commands: dotnetsay
Restore was successful.
```

5. 验证工具是否可用：

```
dotnet tool list
```

输出是包和命令的列表，类似于以下示例：

Package Id	Version	Commands	Manifest
microsoft.botsay	1.0.0	botsay	/home/name/repository/.config/dotnet-tools.json
dotnetsay	2.1.3	dotnetsay	/home/name/repository/.config/dotnet-tools.json

6. 测试工具：

```
dotnet tool run dotnetsay hello from dotnetsay
dotnet tool run botsay hello from botsay
```

更新本地工具

本地工具 `dotnetsay` 的已安装版本为 2.1.3。最新版本是 2.1.4。使用 `dotnet tool update` 命令将工具更新到最新版本。

```
dotnet tool update dotnetsay
```

输出指示新的版本号：

```
Tool 'dotnetsay' was successfully updated from version '2.1.3' to version '2.1.4'
(manifest file /home/name/repository/.config/dotnet-tools.json).
```

`update` 命令查找包含包 ID 的第一个清单文件并对其进行更新。如果搜索范围内的任何清单文件中都没有此类包 ID，SDK 会将新条目添加到最近的清单文件。搜索范围上至父目录，直到找到具有 `isRoot = true` 的清单文件。

删除本地工具

通过运行 `dotnet tool uninstall` 命令来删除已安装的工具：

```
dotnet tool uninstall microsoft.botsay
```

```
dotnet tool uninstall dotnetsay
```

疑难解答

如果在学习本教程时收到错误消息，请参阅[排查 .NET Core 工具使用问题](#)。

请参阅

有关详细信息，请参阅[.NET Core 工具](#)

.NET Core SDK 概述

2020/3/18 • [Edit Online](#)

.NET Core SDK 是一组库和工具，开发人员可用其创建 .NET Core 应用程序和库。它包含以下用于构建和运行应用程序的组件：

- .NET Core CLI。
- .NET Core 库和运行时。
- `dotnet` 驱动程序。

获取 .NET Core SDK

与任何工具一样，首先应将工具安装到计算机上。根据场景，可以使用以下某个方法安装 SDK：

- 使用本机安装程序。
- 使用安装 shell 脚本。

本机安装程序主要用于开发人员的计算机。SDK 通过每个受支持平台的本机安装机制进行分发，例如 Ubuntu 上的 DEB 包或 Windows 上的 MSI 程序包。这些安装程序将根据需要为用户安装并设置环境，以便在安装完成后可立即使用 SDK。但是，这些安装程序也需要对计算机的管理权限。可以在[.NET 下载](#)页面上找到要安装的 SDK。

另一方面，安装脚本不需要使用管理权限。但是，它们也不会在计算机上安装任何系统必备组件；需要手动安装所有系统必备组件。这些脚本主要用于设置生成服务器或希望安装工具但没有管理权限的情况（请务必注意上述系统必备组件注意事项）。可以在[安装脚本引用](#)一文中找到详细信息。如果对如何在 CI 构建服务器上设置 SDK 感兴趣，请参阅[在持续集成 \(CI\) 中使用 .NET Core SDK 和工具](#)一文。

默认情况下，SDK 以“并排”(SxS) 方式安装，这意味着多个版本可以随时在一台计算机上共存。[选择要使用的 .NET Core 版本](#)一文中详细说明了如何在运行 CLI 命令时选择版本。

另请参阅

- [.NET Core CLI 概述](#)
- [.NET Core 版本控制概述](#)
- [如何删除 .NET Core 运行时和 SDK](#)
- [选择要使用的 .NET Core 版本](#)

.NET Core CLI 概述

2020/3/27 • [Edit Online](#)

本文适用于: ✓ .NET Core 2.1 SDK 及更高版本

.NET Core 命令行接口 (CLI) 工具是用于开发、生成、运行和发布 .NET Core 应用程序的跨平台工具链。

.NET Core CLI 包含在 [.NET Core SDK](#) 中。若要了解如何安装 .NET Core SDK, 请参阅[安装 .NET Core SDK](#)。

CLI 命令

默认安装以下命令：

基本命令

- `new`
- `restore`
- `build`
- `publish`
- `run`
- `test`
- `vstest`
- `pack`
- `migrate`
- `clean`
- `sln`
- `help`
- `store`

项目修改命令

- `add package`
- `add reference`
- `remove package`
- `remove reference`
- `list reference`

高级命令

- `nuget delete`
- `nuget locals`
- `nuget push`
- `msbuild`
- `dotnet install script`

工具管理命令

- `tool install`
- `tool list`
- `tool update`
- `tool restore` 自 .NET Core SDK 3.0 起可用。

- `tool run` 自 .NET Core SDK 3.0 起可用。
- `tool uninstall`

工具是控制台应用程序，它们从 NuGet 包中安装并从命令提示符处进行调用。你可自行编写工具，也可安装由第三方编写的工具。工具也称为全局工具、工具路径工具和本地工具。有关详细信息，请参阅 [.NET Core 工具概述](#)。

命令结构

CLI 命令结构包含 [驱动程序 \("dotnet"\)](#) 和 [命令](#)，还可能包含命令 [参数](#) 和 [选项](#)。在大部分 CLI 操作中可看到此模式，例如创建新控制台应用并从命令行运行该应用，因为从名为 `my_app` 的目录中执行时，显示以下命令：

```
dotnet new console  
dotnet build --output /build_output  
dotnet /build_output/my_app.dll
```

驱动程序

驱动程序名为 `dotnet`，并具有两项职责，即运行 [依赖于框架的应用](#) 或执行命令。

若要运行依赖于框架的应用，请在驱动程序后指定应用，例如，`dotnet /path/to/my_app.dll`。从应用的 DLL 驻留的文件夹执行命令时，只需执行 `dotnet my_app.dll` 即可。如果要使用特定版本的 .NET Core 运行时，请使用 `--fx-version <VERSION>` 选项（请参阅 [dotnet 命令](#) 参考）。

为驱动程序提供命令时，`dotnet.exe` 启动 CLI 命令执行过程。例如：

```
dotnet build
```

首先，驱动程序确定要使用的 SDK 版本。如果没有 `global.json` 文件，则使用可用的最新版本 SDK。这有可能是预览版或稳定版，具体取决于计算机上的最新版本。确定 SDK 版本后，它便会执行命令。

命令

由命令执行操作。例如，`dotnet build` 生成代码。`dotnet publish` 发布代码。使用 `dotnet {command}` 约定将命令作为控制台应用程序实现。

自变量

在命令行上传递的参数是被调用的命令的参数。例如，执行 `dotnet publish my_app.csproj` 时，`my_app.csproj` 参数指示要发布的项目，并被传递到 `publish` 命令。

选项

在命令行上传递的选项是被调用的命令的选项。例如，执行 `dotnet publish --output /build_output` 时，`--output` 选项及其值被传递到 `publish` 命令。

请参阅

- [dotnet/sdk GitHub 存储库](#)
- [.NET Core 安装指南](#)

dotnet 命令

2020/4/2 • [Edit Online](#)

本文适用于: ✓ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet` - .NET Core CLI 的通用驱动程序。

摘要

获取有关可用命令和环境的信息：

```
dotnet [-h|--help] [--version] [--info]
        [--list-runtimes] [--list-sdks]
```

运行命令(需要 SDK 安装)：

```
dotnet <COMMAND> [-d|--diagnostics] [-h|--help] [--verbosity]
                    [command-options] [arguments]
```

运行应用程序：

```
dotnet [--additionalprobingpath] [--additional-deps]
        [--fx-version]  [--roll-forward]
        <PATH_TO_APPLICATION> [arguments]

dotnet exec [--additionalprobingpath] [--additional-deps]
            [--fx-version]  [--roll-forward]
            <PATH_TO_APPLICATION> [arguments]
```

`--roll-forward` 自 .NET Core 3.x 起可用。使用 .NET Core 2.x 的 `--roll-forward-on-no-candidate-fx`。

描述

`dotnet` 命令有两个函数：

- 它提供了用于处理 .NET Core 项目的命令。

例如，`dotnet build` 生成项目。每个命令定义自己的选项和参数。所有命令都支持 `--help` 选项，用于打印有关如何使用命令的简短文档。

- 它运行 .NET Core 应用程序。

指定应用程序 `.dll` 文件的路径以运行应用程序。例如，`dotnet myapp.dll` 运行 `myapp` 应用程序。要了解部署选项，请参阅 [.NET Core 应用程序部署](#)。

选项

`dotnet` 本身有不同的选项，可用于运行命令和运行应用程序。

dotnet 本身的选项

以下是 `dotnet` 本身选项。例如 `dotnet --info`。这些选项打印出有关环境的信息。

- `--info`
打印出有关 .NET Core 安装和计算机环境(如当前操作系统)的详细信息，并提交 .NET Core 版本的 SHA。
- `--version`
打印使用中的 .NET Core SDK 版本。
- `--list-runtimes`
打印已安装的 .NET Core 运行时的列表。
- `--list-sdks`
打印已安装的 .NET Core SDK 的列表。
- `-h|--help`
打印可用命令列表。

用于运行命令的 SDK 选项

以下选项适用于使用命令的 `dotnet`。例如 `dotnet build --help`。

- `-d|--diagnostics`
启用诊断输出。
- `-v|--verbosity <LEVEL>`
设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。
并非在每个命令中均受支持。请参阅特定的命令页，确定此选项是否可用。
- `-h|--help`
打印出给定命令的文档，如 `dotnet build --help`。
- `command options`
每个命令定义特定于该命令的选项。有关可用选项的列表，请参阅特定命令页。

运行时选项

`dotnet` 运行应用程序时，可以使用以下选项。例如 `dotnet myapp.dll --fx-version 3.1.1`。

- `--additionalprobingpath <PATH>`
包含要进行探测的探测策略和程序集的路径。
- `--additional-deps <PATH>`
附加 `.deps.json` 文件的路径。`.deps.json` 文件包含依赖项、编译依赖项和用于解决程序集冲突的版本信息列表。有关详细信息，请参阅 GitHub 上的[运行时配置文件](#)。
- `--fx-version <VERSION>`
用于运行应用程序的 .NET Core 运行时版本。
- `--runtimeconfig`
`runtimeconfig.template.json` 文件的路径。`runtimeconfig.template.json` 文件是包含运行时设置的配置文件。有关详细信息，请参阅[.NET Core 运行时配置设置](#)。
- `--roll-forward-on-no-candidate-fx <N>` 在 .NET Core 2.x SDK 中可用。

所需的共享框架不可用时, 请定义行为。`N` 可以是:

- `0` - 禁用次要版本前滚。
- `1` - 前滚次要版本, 但不前滚主版本。这是默认行为。
- `2` - 前滚次要和主版本。

有关详细信息, 请参阅[前滚](#)。

- `--roll-forward <SETTING>` 自 .NET Core SDK 3.0 起可用。

控制将前滚操作应用于应用的方式。`SETTING` 可以为下列值之一。如果未指定, 则 `Minor` 为默认类型。

- `LatestPatch` - 前滚到最高补丁版本。这会禁用次要版本前滚。
- `Minor` - 如果缺少所请求的次要版本, 则前滚到最低的较高次要版本。如果存在所请求的次要版本, 则使用 `LatestPatch` 策略。
- `Major` - 如果缺少所请求的主要版本, 则前滚到最低的较高主要版本和最低的次要版本。如果存在所请求的主要版本, 则使用 `Minor` 策略。
- `LatestMinor` - 即使存在所请求的次要版本, 仍前滚到最高次要版本。适用于组件托管方案。
- `LatestMajor` - 即使存在所请求的主要版本, 仍前滚到最高主要版本和最高次要版本。适用于组件托管方案。
- `Disable` - 不前滚。仅绑定到指定的版本。建议不要将此策略用于一般用途, 因为它会禁用前滚到最新补丁的功能。该值仅建议用于测试。

除 `Disable` 外, 所有设置都将使用可用的最高补丁版本。

前滚行为还可以在项目文件属性、运行时配置文件属性和环境变量中进行配置。有关详细信息, 请参阅[主版本运行时前滚](#)。

dotnet 命令

常规

命令	描述
dotnet build	生成 .NET Core 应用程序。
dotnet build-server	与通过生成启动的服务器进行交互。
dotnet clean	清除生成输出。
dotnet help	显示命令更详细的在线文档。
dotnet migrate	将有效的预览版 2 项目迁移到 .NET Core SDK 1.0 项目。
dotnet msbuild	提供对 MSBuild 命令行的访问权限。
dotnet new	为给定的模板初始化 C# 或 F# 项目。
dotnet pack	创建代码的 NuGet 包。
dotnet publish	发布 .NET 依赖于框架或独立应用程序。
dotnet restore	还原给定应用程序的依赖项。

<code>dotnet run</code>	从源运行应用程序。
<code>dotnet sln</code>	用于添加、删除和列出解决方案文件中项目的选项。
<code>dotnet store</code>	将程序集存储到运行时包存储区。
<code>dotnet test</code>	使用测试运行程序运行测试。

项目引用

<code>dotnet add reference</code>	添加项目引用。
<code>dotnet list reference</code>	列出项目引用。
<code>dotnet remove reference</code>	删除项目引用。

NuGet 包

<code>dotnet add package</code>	添加 NuGet 包。
<code>dotnet remove package</code>	删除 NuGet 包。

NuGet 命令

<code>dotnet nuget delete</code>	从服务器删除或取消列出包。
<code>dotnet nuget push</code>	将包推送到服务器，并将其发布。
<code>dotnet nuget locals</code>	清除或列出本地 NuGet 资源，例如 http 请求缓存、临时缓存或计算机范围的全局包文件夹。
<code>dotnet nuget add source</code>	添加 NuGet 源。
<code>dotnet nuget disable source</code>	禁用 NuGet 源。
<code>dotnet nuget enable source</code>	启用 NuGet 源。
<code>dotnet nuget list source</code>	列出所有已配置的 NuGet 源。
<code>dotnet nuget remove source</code>	删除 NuGet 源。
<code>dotnet nuget update source</code>	更新 NuGet 源。

全局、工具路径和本地工具命令

工具是控制台应用程序，它们从 NuGet 包中安装并从命令提示符处进行调用。你可自行编写工具，也可安装由第

三方编写的工具。工具也称为全局工具、工具路径工具和本地工具。有关详细信息，请参阅 [.NET Core 工具概述](#)。全局和工具路径工具从 .NET Core SDK 2.1 开始可用。本地工具从 .NET Core SDK 3.0 开始可用。

命令	描述
<code>dotnet tool install</code>	在计算机上安装工具。
<code>dotnet tool list</code>	列出计算机上当前安装的所有全局、工具路径或本地工具。
<code>dotnet tool uninstall</code>	从计算机中卸载工具。
<code>dotnet tool update</code>	更新计算机上安装的工具。

其他工具

自 .NET Core SDK 2.1.300 开始，许多使用 `DotnetCliToolReference` 的仅在每个项目的基础上可用的工具现作为 .NET Core SDK 的一部分提供。下表中列出了这些工具：

命令	描述
<code>dev-certs</code>	创建和管理开发证书。
<code>ef</code>	Entity Framework Core 命令行工具。
<code>sql-cache</code>	SQL Server 缓存命令行工具。
<code>user-secrets</code>	管理开发用户机密。
<code>watch</code>	启动文件观察程序，以在更改文件时运行命令。

有关每个工具的详细信息，请键入 `dotnet <tool-name> --help`。

示例

创建新的 .NET Core 控制台应用程序：

```
dotnet new console
```

生成给定目录中的项目及其依赖项：

```
dotnet build
```

运行应用程序：

```
dotnet myapp.dll
```

环境变量

- `DOTNET_ROOT` , `DOTNET_ROOT(x86)`

指定 .NET Core 运行时的位置（如果运行时未安装在默认位置）。Windows 上的默认位置为 `C:\Program Files\dotnet`。Linux 和 macOS 上的默认位置为 `/usr/share/dotnet`。此环境变量仅在通过生

成的可执行文件 (apphosts) 运行应用时使用。在 64 位 OS 上运行 32 位可执行文件时，改用 `DOTNET_ROOT(x86)`。

- `DOTNET_PACKAGES`

全局包文件夹。如果未设置，则默认为 Unix 上的 `~/.nuget/packages` 或 Windows 上的 `%userprofile%\.nuget\packages`。

- `DOTNET_SERVICING`

指定加载运行期间共享主机要使用的服务索引的位置。

- `DOTNET_NOLOGO`

指定是否在首次运行时显示 .NET Core 欢迎消息和遥测消息。设置为 `true` 可将这些消息静音(接受 `true`、`1` 或 `yes` 值)，或者，设置为 `false` 可允许显示消息(接受 `false`、`0` 或 `no` 值)。如果未设置，则默认值为 `false`，表示在首次运行时将显示消息。请注意，此标志对遥测不起作用(请参阅 `DOTNET_CLI_TELEMETRY_OPTOUT` 中关于如何选择不发送遥测数据的信息)。

- `DOTNET_CLI_TELEMETRY_OPTOUT`

指定是否收集并向 Microsoft 发送 .NET Core 工具使用情况的相关数据。设置为 `true` 以选择退出遥测功能(接受的值为 `true`、`1` 或 `yes`)。否则，设置为 `false` 以选择加入遥测功能(接受的值为 `false`、`0` 或 `no`)。如果未设置，则默认为 `false` 且遥测功能为活动状态。

- `DOTNET_MULTILEVEL_LOOKUP`

指定是否从全局位置解析 .NET Core 运行时、共享框架或 SDK。如果未设置，则默认为 1(逻辑 `true`)。设置为 0(逻辑 `false`)，不从全局位置解析，并且具有独立的 .NET Core 安装。有关多级别查找的详细信息，请参阅 [Multi-level SharedFX Lookup](#)(多级别 SharedFX 查找)。

- `DOTNET_ROLL_FORWARD` 自 .NET Core 3.x SDK 起可用。

确定前滚行为。有关详细信息，请参阅本文章前面介绍的 `--roll-forward` 选项。

- `DOTNET_ROLL_FORWARD_ON_NO_CANDIDATE_FX` 在 .NET Core 2.x SDK 中可用。

如果设置为 `0`，则禁用次要版本前滚。有关详细信息，请参阅[前滚](#)。

- `DOTNET_CLI_UI_LANGUAGE`

使用区域设置值(如 `en-us`)设置 CLI UI 的语言。支持的值与 Visual Studio 中的值相同。有关详细信息，请参阅 [Visual Studio 安装文档](#)中有关更改安装程序语言一节。.NET 资源管理器规则适用，因此你无需选取精确匹配项 — 你还可以在 `CultureInfo` 树中选取后代。例如，如果将其设置为 `fr-CA`，CLI 将查找并使用 `fr` 翻译。如果你将其设置为不受支持的语言，CLI 会退回到英语。

- `DOTNET_DISABLE_GUI_ERRORS`

对于启用了 GUI 的已生成可执行文件 - 禁用对话框弹出窗口，此窗口通常显示某些类错误。在这些情况下，它仅写入到 `stderr` 并退出。

- `DOTNET_ADDITIONAL_DEPS`

等效于 CLI 选项 `--additional-deps`。

- `DOTNET_RUNTIME_ID`

替代检测到的 RID。

- `DOTNET_SHARED_STORE`

程序集解析在某些情况下将回退到的“共享存储”的位置。

- `DOTNET_STARTUP_HOOKS`

要从中加载和执行启动挂钩的程序集列表。

- `COREHOST_TRACE` , `COREHOST_TRACEFILE` , `COREHOST_TRACE_VERBOSITY`

控制来自托管组件(例如 `dotnet.exe`、`hostfxr` 和 `hostpolicy`)的诊断跟踪。

请参阅

- [运行时配置文件](#)
- [.NET Core 运行时配置设置](#)

dotnet 生成

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.x SDK 及更高版本

名称

`dotnet build` - 生成项目及其所有依赖项。

摘要

```
dotnet build [<PROJECT>|<SOLUTION>] [-c|--configuration] [-f|--framework] [--force]
[--interactive] [--no-dependencies] [--no-incremental] [--no-restore] [--nologo]
[-o|--output] [-r|--runtime] [-v|--verbosity] [--version-suffix]

dotnet build [-h|--help]
```

说明

`dotnet build` 命令将项目及其依赖项生成为一组二进制文件。二进制文件包括扩展名为 .dll 的中间语言 (IL) 文件中的项目代码。根据项目类型和设置，可能会包含其他文件，例如：

- 可用于运行应用程序的可执行文件(如果项目类型是面向 .NET Core 3.0 或更高版本的可执行文件)。
- 用于调试的扩展名为 .pdb 的符号文件。
- 列出了应用程序或库的依赖项的 .deps.json 文件。
- 用于指定应用程序的共享运行时及其版本的 .runtimeconfig.json 文件。
- 项目通过项目引用或 NuGet 包引用所依赖的其他库。

对于目标版本低于 .NET Core 3.0 的可执行项目，通常不会将 NuGet 中的库依赖项复制到输出文件夹。而是在运行时从 NuGet 全局包文件夹中对其进行解析。考虑到这一点，`dotnet build` 的产品还未准备好转移到另一台计算机进行运行。要创建可部署的应用程序版本，需要发布该应用程序(例如，使用 `dotnet publish` 命令)。有关详细信息，请参阅 [.NET Core 应用程序部署](#)。

对于面向 .NET Core 3.0 及更高版本的可执行项目，库依赖项会被复制到输出文件夹。这意味着如果没有其他任何特定于发布的逻辑(例如，Web 项目具有的逻辑)，则应可部署生成输出。

构建需要 `project.assets.json` 文件，该文件列出了你的应用程序的依赖项。此文件在 `dotnet restore` 执行时创建。如果资产文件未就位，那么工具将无法解析引用程序集，进而导致错误生成。使用 .NET Core 1.x SDK，需要在运行 `dotnet restore` 之前显式运行 `dotnet build`。自 .NET Core 2.0 SDK 起，`dotnet restore` 在 `dotnet build` 运行时隐式运行。若要在运行 build 命令时禁用隐式还原，可以传递 `--no-restore` 选项。

NOTE

从 .NET Core 2.0 开始，无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet build` 和 `dotnet run`。在执行显式还原有意义的某些情况下，例如 [Azure DevOps Services 中的持续集成生成](#) 中，或在需要显式控制还原发生时间的生成系统中，它仍然是有效的命令。

以长格式传递命令时，该命令也支持 `dotnet restore` 选项(例如，`--source`)。不支持缩写选项，例如 `-s`。

项目是否可执行由项目文件中的 `<OutputType>` 属性决定。以下示例显示生成可执行代码的项目：

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
</PropertyGroup>
```

要生成库, 请省略 `<OutputType>` 属性或将其值更改为 `Library`。库的 IL DLL 不包含入口点, 因此无法执行。

MSBuild

`dotnet build` 使用 MSBuild 生成项目, 因此它支持并行生成和增量生成。有关详细信息, 请参阅[增量生成](#)。

除其自己的选项外, `dotnet build` 命令也接受 MSBuild 选项, 如用来设置属性的 `-p` 或用来定义记录器的 `-l`。有关这些选项的详细信息, 请参阅[MSBuild 命令行参考](#)。或者也可以使用 `dotnet msbuild` 命令。

运行 `dotnet build` 等同于运行 `dotnet msbuild -restore`; 但是, 输出的默认详细程度不同。

参数

PROJECT | SOLUTION

要生成的项目或解决方案文件。如果未指定项目或解决方案文件, MSBuild 会在当前工作目录中搜索文件扩展名以 `proj` 或 `sln` 结尾的文件并使用该文件。

选项

- `-c|--configuration <CONFIGURATION>`

定义生成配置。大多数项目的默认配置为 `Debug`, 但你可以覆盖项目中的生成配置设置。

- `-f|--framework <FRAMEWORK>`

编译特定框架。必须在[项目文件](#)中定义该框架。

- `--force`

强制解析所有依赖项, 即使上次还原已成功, 也不例外。指定此标记等同于删除 `project.assets.json` 文件。

- `-h|--help`

打印出有关命令的简短帮助。

- `--interactive`

允许命令停止并等待用户输入或操作。例如, 完成身份验证。自 .NET Core 3.0 SDK 起可用。

- `--no-dependencies`

忽略项目到项目 (P2P) 引用, 并仅生成指定的根项目。

- `--no-incremental`

将生成标记为对[增量生成](#)不安全。此标记关闭增量编译, 并强制完全重新生成项目依赖项关系图。

- `--no-restore`

在生成期间不执行隐式还原。

- `--nologo`

不显示启动版权标志或版权消息。自 .NET Core 3.0 SDK 起可用。

- `-o|--output <OUTPUT_DIRECTORY>`

放置生成二进制文件的目录。如果未指定，则默认路径为 `./bin/<configuration>/<framework>/`。对于具有多个目标框架的项目（通过 `TargetFrameworks` 属性），在指定此选项时还需要定义 `--framework`。

- `-r|--runtime <RUNTIME_IDENTIFIER>`

指定目标运行时。有关运行时标识符 (RID) 的列表，请参阅 [RID 目录](#)。

- `-v|--verbosity <LEVEL>`

设置 MSBuild 详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。

- `--version-suffix <VERSION_SUFFIX>`

设置生成项目时使用的 `$(VersionSuffix)` 属性的值。这仅在未设置 `$(Version)` 属性时有效。然后，`$(Version)` 设置为 `$(VersionPrefix)` 与 `$(VersionSuffix)` 组合，并用短划线分隔。

示例

- 生成项目及其依赖项：

```
dotnet build
```

- 使用“发布”配置生成项目及其依赖项：

```
dotnet build --configuration Release
```

- 针对特定运行时（本例中为 Ubuntu 18.04）生成项目及其依赖项：

```
dotnet build --runtime ubuntu.18.04-x64
```

- 生成项目，并在还原操作过程中使用指定的 NuGet 包源 (.NET Core 2.0 SDK 及更高版本)：

```
dotnet build --source c:\packages\mypackages
```

- 生成项目并设置版本 1.2.3.4 作为使用 `-p` [MSBuild 选项](#) 的生成参数：

```
dotnet build -p:Version=1.2.3.4
```

dotnet build-server

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.1 SDK 及更高版本

名称

`dotnet build-server` -与通过生成启动的服务器进行交互。

摘要

```
dotnet build-server shutdown [--msbuild] [--razor] [--vbcscompiler]
dotnet build-server shutdown [-h|--help]
dotnet build-server [-h|--help]
```

命令

- `shutdown`

关闭从 dotnet 启动的生成服务器。默认情况下会关闭所有服务器。

选项

- `-h|--help`

打印出有关命令的简短帮助。

- `--msbuild`

关闭 MSBuild 生成服务器。

- `--razor`

关闭 Razor 生成服务器。

- `--vbcscompiler`

关闭 VB/C# 编译器生成服务器。

dotnet clean

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.x SDK 及更高版本

名称

`dotnet clean` - 清除项目输出。

摘要

```
dotnet clean [<PROJECT>|<SOLUTION>] [-c|--configuration] [-f|--framework] [--interactive]
    [--nologo] [-o|--output] [-r|--runtime] [-v|--verbosity]
dotnet clean [-h|--help]
```

说明

`dotnet clean` 命令可清除上一个生成的输出。它以 [MSBuild 目标](#) 的形式实现，以便在运行命令时对项目进行评估。只会清除在生成过程中创建的输出。中间 (*obj*) 和最终输出 (*bin*) 文件夹都会被清除。

参数

`PROJECT | SOLUTION`

要清理的 MSBuild 项目或解决方案。如果未指定项目或解决方案文件，MSBuild 会在当前工作目录中搜索文件扩展名以 *proj* 或 *sln* 结尾的文件并使用该文件。

选项

- `-c|--configuration <CONFIGURATION>`

定义生成配置。大多数项目的默认配置为 `Debug`，但你可以覆盖项目中的生成配置设置。只有在生成期间指定了此选项，才必须在清除时使用此选项。

- `-f|--framework <FRAMEWORK>`

在生成时指定的框架。必须在[项目文件](#)中定义该框架。如果在生成时指定了框架，则必须在清除时指定框架。

- `-h|--help`

打印出有关命令的简短帮助。

- `--interactive`

允许命令停止并等待用户输入或操作。例如，完成身份验证。自 .NET Core 3.0 SDK 起可用。

- `--nologo`

不显示启动版权标志或版权消息。自 .NET Core 3.0 SDK 起可用。

- `-o|--output <OUTPUT_DIRECTORY>`

包含要清理的生成项目的目录。如果在生成项目时指定了框架，则使用输出目录开关指定

`-f|--framework <FRAMEWORK>` 开关。

- `-r|--runtime <RUNTIME_IDENTIFIER>`

清除指定运行时的输出文件夹。在创建[独立部署 \(SCD\)](#)时使用此选项。

- `-v|--verbosity <LEVEL>`

设置 MSBuild 详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。

默认值为 `normal`。

示例

- 清除项目的默认生成：

```
dotnet clean
```

- 清除使用版本配置生成的项目：

```
dotnet clean --configuration Release
```

dotnet help 参考

2020/3/18 • [Edit Online](#)

本文适用于: ✓ .NET Core 2.0 SDK 及更高版本

名称

`dotnet help` - 显示指定命令更详细的在线文档。

摘要

`dotnet help <COMMAND_NAME> [-h|--help]`

说明

`dotnet help` 命令打开 docs.microsoft.com 参考页，以提供指定命令的更多详细信息。

参数

- `COMMAND_NAME`

.NET Core CLI 命令名称。有关有效 CLI 命令的列表，请参阅 [CLI 命令](#)。

选项

- `-h|--help`

打印出有关命令的简短帮助。

示例

- 打开 [dotnet new](#) 命令的文档页：

```
dotnet help new
```

dotnet migrate

2020/3/18 • [Edit Online](#)

本文适用于: ✓ .NET Core 2.x SDK

名称

`dotnet migrate` - 将预览版 2 .NET Core 项目迁移到 .NET Core SDK 样式的项目中。

摘要

```
dotnet migrate [<SOLUTION_FILE|PROJECT_DIR>] [--format-report-file-json] [-r|--report-file] [-s|--skip-project-references] [--skip-backup] [-t|--template-file] [-v|--sdk-package-version] [-x|--xproj-file]
dotnet migrate [-h|--help]
```

说明

此命令已弃用。自 .NET Core 3.0 SDK 起, `dotnet migrate` 命令不再可用。它只能将预览版 2 .NET Core 项目迁移到不支持的 1.x .NET Core 项目。

默认情况下, 命令迁移根项目和根项目包含的任何项目引用。在运行时使用 `--skip-project-references` 选项禁用此行为。

可在下列资产上执行迁移:

- 通过指定 `project.json` 文件进行迁移的单个项目。
- 通过将路径传递到 `global.json` 文件在 `global.json` 文件中指定的所有目录。
- 一个 `solution.sln` 文件, 它迁移在解决方案中引用的项目。
- 以递归方式迁移给定目录的所有子目录。

`dotnet migrate` 命令将迁移的 `project.json` 文件保存在 `backup` 目录中, 如果该目录不存在, 将创建一个。使用 `--skip-backup` 选项重写此行为。

默认情况下, 迁移操作会将迁移过程的状态输出到标准输出 (STDOUT)。如果使用 `--report-file <REPORT_FILE>` 选项, 输出将保存到指定的文件中。

`dotnet migrate` 命令仅支持有效的预览版 2 基于 `project.json` 的项目。这意味着, 不能使用它将 DNX 或预览版 1 基于 `project.json` 的项目直接迁移到 MSBuild/csproj 项目。首先需要将项目手动迁移到预览版 2 基于 `project.json` 的项目, 然后使用 `dotnet migrate` 命令迁移该项目。

参数

`PROJECT_JSON/GLOBAL_JSON/SOLUTION_FILE/PROJECT_DIR`

下列路径之一:

- 要迁移的 `project.json` 文件。
- `global.json` 文件: 迁移在 `global.json` 中指定的文件夹。
- `solution.sln` 文件: 迁移该解决方案中引用的项目。
- 要迁移的目录: 在指定的目录中以递归方式搜索要迁移的 `project.json` 文件。

如未指定，则默认为当前目录。

选项

```
--format-report-file-json <REPORT_FILE>
```

将迁移报告文件（而非用户消息）作为 JSON 输出。

```
-h|--help
```

打印出有关命令的简短帮助。

```
-r|--report-file <REPORT_FILE>
```

除控制台外，还将迁移报告输出到文件。

```
-s|--skip-project-references [Debug|Release]
```

跳过迁移项目引用。默认情况下，以递归方式迁移项目引用。

```
--skip-backup
```

在成功迁移后，跳过将 *project.json*、*global.json* 和 **.xproj* 移动到 `backup` 目录的步骤。

```
-t|--template-file <TEMPLATE_FILE>
```

用于迁移的模板 csproj 文件。默认情况下，使用与被 `dotnet new console` 删除的模板相同的模板。

```
-v|--sdk-package-version <VERSION>
```

在已迁移应用中将被引用的 sdk 包的版本。默认为 `dotnet new` 中 SDK 的版本。

```
-x|--xproj-file <FILE>
```

要使用的 xproj 文件的路径。当项目目录中有多个 xproj 时需要。

示例

将当前目录中的项目及其所有项目迁移到项目依赖项：

```
dotnet migrate
```

迁移 *global.json* 文件所包含的所有项目：

```
dotnet migrate path/to/global.json
```

仅迁移当前项目，不迁移项目到项目 (P2P) 的依赖项。此外，使用特定的 SDK 版本：

```
dotnet migrate -s -v 1.0.0-preview4
```

dotnet msbuild

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.x SDK 及更高版本

名称

`dotnet msbuild` - 生成项目及其所有依赖项。

摘要

`dotnet msbuild <msbuild_arguments> [-h]`

说明

`dotnet msbuild` 命令允许访问功能完备的 MSBuild。

该命令与仅适用于 SDK 样式项目的现有 MSBuild 命令行客户端具有完全相同的功能。选项一致。有关可用选项的详细信息，请参阅 [MSBuild 命令行参考](#)。

`dotnet build` 命令相当于 `dotnet msbuild -restore -target:Build`。`dotnet build` 更常用于生成项目，但由于它始终运行生成目标，因此不想生成项目时可以使用 `dotnet msbuild`。例如，如果想要运行特定目标而不生成项目，请使用 `dotnet msbuild` 指定目标。

示例

- 生成项目及其依赖项：

```
dotnet msbuild
```

- 使用“发布”配置生成项目及其依赖项：

```
dotnet msbuild -property:Configuration=Release
```

- 运行发布目标并发布 `osx.10.11-x64` RID：

```
dotnet msbuild -target:Publish -property:RuntimeIdentifiers=osx.10.11-x64
```

- 请参阅包含 SDK 添加的所有目标的整个项目：

```
dotnet msbuild -preprocess
```

dotnet new

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.0 SDK 及更高版本

名称

`dotnet new` - 根据指定的模板，创建新的项目、配置文件或解决方案。

摘要

```
dotnet new <TEMPLATE> [--dry-run] [--force] [-i|--install] [-lang|--language] [-n|--name]
    [--nuget-source] [-o|--output] [-u|--uninstall] [--update-apply] [--update-check] [Template
    options]
dotnet new <TEMPLATE> [-l|--list] [--type]
dotnet new [-h|--help]
```

说明

`dotnet new` 命令基于模板创建 .NET Core 项目或其他项目。

命令调用[模板引擎](#)，以根据指定的模板和选项在磁盘上创建项目。

参数

- `TEMPLATE`

调用命令时要实例化的模板。每个模板可能具有可传递的特定选项。有关详细信息，请参阅[模板选项](#)。

可以运行 `dotnet new --list` 以查看所有已安装模板的列表。如果 `TEMPLATE` 值与返回表中的“模板”或“短名称”列中的文本不完全匹配，则会对这两列执行 substring 匹配。

从 .NET Core 3.0 SDK 开始，当你在以下情况下调用 `dotnet new` 命令时，CLI 将在 NuGet.org 中搜索模板：

- 如果在调用 `dotnet new` 时 CLI 找不到模板匹配项，即使是部分匹配也不行。
- 如果有较新版本的模板可用。在这种情况下，将创建项目或工件，但 CLI 会就模板的更新版本发出警告。

此命令包含默认的模板列表。使用 `dotnet new -l` 获取可用模板的列表。下表显示了随 .NET Core SDK 一起预安装的模板。模板的语言显示在括号内。单击短名称链接可查看特定的模板选项。

名称	短名称	语言	标签	版本
控制台应用程序	控制台	[C#]、F#、VB	常用/控制台	1.0
类库	classlib	[C#]、F#、VB	常用/库	1.0
WPF 应用程序	wpf	[C#]	常用/WPF	3.0

名称	标签	语言	TAGS	版本
WPF 类库	wpflib	[C#]	常用/WPF	3.0
WPF 自定义控件库	wpfcustomcontrollib	[C#]	常用/WPF	3.0
WPF 用户控件库	wpfusercontrollib	[C#]	常用/WPF	3.0
Windows 窗体 (WinForms) 应用程序	winforms	[C#]	常用/WinForms	3.0
Windows 窗体 (WinForms) 类库	winformslib	[C#]	常用/WinForms	3.0
Worker Service	worker	[C#]	常用/Worker/Web	3.0
单元测试项目	mstest	[C#]、F#、VB	测试/MSTest	1.0
NUnit 3 测试项目	nunit	[C#]、F#、VB	测试/NUnit	2.1.400
NUnit 3 测试项	nunit-test	[C#]、F#、VB	测试/NUnit	2.2
xUnit 测试项目	xunit	[C#]、F#、VB	测试/xUnit	1.0
Razor 组件	razorcomponent	[C#]	Web/ASP.NET	3.0
Razor 页	page	[C#]	Web/ASP.NET	2.0
MVC ViewImports	viewimports	[C#]	Web/ASP.NET	2.0
MVC ViewStart	viewstart	[C#]	Web/ASP.NET	2.0
Blazor Server 应用	blazorserver	[C#]	Web/Blazor	3.0
ASP.NET Core 空	web	[C#], F#	Web/空	1.0
ASP.NET Core Web 应用程序 (Model-View-Controller)	mvc	[C#], F#	Web/MVC	1.0
ASP.NET Core Web 应用程序	webapp、razor	[C#]	Web/MVC/Razor Pages	2.2、2.0
含 Angular 的 ASP.NET Core	angular	[C#]	Web/MVC/SPA	2.0
含 React.js 的 ASP.NET Core	react	[C#]	Web/MVC/SPA	2.0
含 React.js 和 Redux 的 ASP.NET Core	reactredux	[C#]	Web/MVC/SPA	2.0

名称	命令	语言	标签	版本
Razor 类库	razorclasslib	[C#]	Web/Razor/ 库/Razor 类库	2.1
ASP.NET Core Web API	webapi	[C#], F#	Web/WebAPI	1.0
ASP.NET Core gRPC 服务	grpc	[C#]	Web/gRPC	3.0
协议缓冲区文件	proto		Web/gRPC	3.0
dotnet gitignore 文件	gitignore		配置	3.0
global.json 文件	globaljson		配置	2.0
NuGet 配置	nugetconfig		配置	1.0
dotnet 本地工具清单文件	tool-manifest		配置	3.0
Web 配置	webconfig		配置	1.0
解决方案文件	sln		解决方案	1.0

选项

- [--dry-run](#)

显示有关以下内容的摘要：给定命令运行时发生的情况。自 .NET Core 2.2 SDK 起可用。

- [--force](#)

强制生成内容，即使会更改现有文件，也不例外。当选择的模板将覆盖输出目录中的现有文件时，需要执行此操作。

- [-h|--help](#)

打印命令帮助。可针对 `dotnet new` 命令本身或任何模板调用它。例如，`dotnet new mvc --help`。

- [-i|--install <PATH|NUGET_ID>](#)

从提供的 `PATH` 或 `NUGET_ID` 安装模板包。若要安装模板包的预发布版本，需要以 `<package-name>::<package-version>` 格式指定该版本。默认情况下，`dotnet new` 为该版本传递 *，它表示最新的稳定包版本。请在[示例](#)部分查看示例。

如果在运行此命令时已经安装了模板的某个版本，则该模板将更新到指定版本，如果没有指定版本，则将更新到最新的稳定版本。

若要了解如何创建自定义模板，请参阅[dotnet new 自定义模板](#)。

- [-l|--list](#)

列出包含指定名称的模板。如果未指定名称，则列出所有模板。

- `-lang|--language {C#|F#|VB}`

要创建的模板的语言。接受的语言因模板而异(请参阅[参数](#)部分中的默认值)。对于某些模板无效。

NOTE

某些 shell 将 `#` 解释为特殊字符。在这些情况下, 请将语言参数值括在引号中。例如,
`dotnet new console -lang "F#"`。

- `-n|--name <OUTPUT_NAME>`

所创建的输出的名称。如果未指定名称, 使用的是当前目录的名称。

- `--nuget-source`

指定在安装期间要使用的 NuGet 源。自 .NET Core 2.1 SDK 起可用。

- `-o|--output <OUTPUT_DIRECTORY>`

用于放置生成的输出的位置。默认为当前目录。

- `--type`

根据可用类型筛选模板。预定义的值为“项目”、“项”或“其他”。

- `-u|--uninstall [PATH|NUGET_ID]`

在提供的 `PATH` 或 `NUGET_ID` 中卸载模板包。如果未指定 `<PATH|NUGET_ID>` 值, 将显示所有当前安装的模板包及其关联的模板。指定 `NUGET_ID` 时, 请勿包含版本号。

如果未指定此选项的参数, 该命令将列出已安装的模板及其详细信息。

NOTE

若要使用 `PATH` 卸载模板, 需要完全限定路径。例如
如, `C:/Users/USER>/Documents/Templates/GarciaSoftware.ConsoleTemplate.CSharp`< 有效, 但是包含文件夹中的 `./GarciaSoftware.ConsoleTemplate.CSharp` 无效。模板路径中不要包含最后的终止目录斜杠。

- `--update-apply`

检查是否有可用于当前安装的模板包的更新并安装这些更新。自 .NET Core 3.0 SDK 起可用。

- `--update-check`

检查是否有可用于当前安装的模板包的更新。自 .NET Core 3.0 SDK 起可用。

模板选项

每个项目模板都可能有附加选项。核心模板有以下附加选项:

控制台

- `-f|--framework <FRAMEWORK>`

指定目标框架。自 .NET Core 3.0 SDK 起可用。

下表根据所使用的 SDK 版本号列出了默认值:

SDK 版本	
3.1	netcoreapp3.1
3.0	netcoreapp3.0

- `--langVersion <VERSION_NUMBER>`

在已创建的项目文件中设置 `LangVersion` 属性。例如，使用 `--langVersion 7.3` 以使用 C# 7.3。不支持 F#。自 .NET Core 2.2 SDK 起可用。

有关默认的 C# 版本列表，请参阅[默认](#)。

- `--no-restore`

如已指定，则在项目创建期间不执行隐式还原。自 .NET Core 2.2 SDK 起可用。

classlib

- `-f|--framework <FRAMEWORK>`

指定目标框架。值：`netcoreapp<version>`（要创建 .NET Core 类库的话）或 `netstandard<version>`（要创建 .NET Standard 类库的话）。默认值为 `netstandard2.0`。

- `--langVersion <VERSION_NUMBER>`

在已创建的项目文件中设置 `LangVersion` 属性。例如，使用 `--langVersion 7.3` 以使用 C# 7.3。不支持 F#。自 .NET Core 2.2 SDK 起可用。

有关默认的 C# 版本列表，请参阅[默认](#)。

- `--no-restore`

在项目创建期间不执行隐式还原。

wpf、wpflib、wpfcustomcontrollib、wpfusercontrollib

- `-f|--framework <FRAMEWORK>`

指定目标框架。默认值为 `netcoreapp3.1`。自 .NET Core 3.1 SDK 起可用。

- `--langVersion <VERSION_NUMBER>`

在已创建的项目文件中设置 `LangVersion` 属性。例如，使用 `--langVersion 7.3` 以使用 C# 7.3。

有关默认的 C# 版本列表，请参阅[默认](#)。

- `--no-restore`

在项目创建期间不执行隐式还原。

winforms、winformslib

- `--langVersion <VERSION_NUMBER>`

在已创建的项目文件中设置 `LangVersion` 属性。例如，使用 `--langVersion 7.3` 以使用 C# 7.3。

有关默认的 C# 版本列表，请参阅[默认](#)。

- `--no-restore`

在项目创建期间不执行隐式还原。

worker、grpc

- `-f|--framework <FRAMEWORK>`

指定目标**框架**。默认值为 `netcoreapp3.1`。自 .NET Core 3.1 SDK 起可用。

- `--exclude-launch-settings`

从生成的模板中排除 `launchSettings.json`。

- `--no-restore`

在项目创建期间不执行隐式还原。

mstest、xunit

- `-f|--framework <FRAMEWORK>`

指定目标**框架**。自 .NET Core 3.0 SDK 起可用的选项。

下表根据所使用的 SDK 版本号列出了默认值：

SDK	
3.1	<code>netcoreapp3.1</code>
3.0	<code>netcoreapp3.0</code>

- `-p|--enable-pack`

允许使用 `dotnet pack` 为项目打包。

- `--no-restore`

在项目创建期间不执行隐式还原。

nunit

- `-f|--framework <FRAMEWORK>`

指定目标**框架**。

下表根据所使用的 SDK 版本号列出了默认值：

SDK	
3.1	<code>netcoreapp3.1</code>
3.0	<code>netcoreapp3.0</code>
2.2	<code>netcoreapp2.2</code>
2.1	<code>netcoreapp2.1</code>

- `-p|--enable-pack`

允许使用 `dotnet pack` 为项目打包。

- `--no-restore`

在项目创建期间不执行隐式还原。

页

- `-na| --namespace <NAMESPACE_NAME>`

已生成代码的命名空间。默认值为 `MyApp.Namespace`。

- `-np| --no-pagemodel`

创建不含 PageModel 的页。

viewimports、proto

- `-na| --namespace <NAMESPACE_NAME>`

已生成代码的命名空间。默认值为 `MyApp.Namespace`。

blazorserver

- `-au| --auth <AUTHENTICATION_TYPE>`

要使用的身份验证类型。可能的值为：

- `None` - 不进行身份验证(默认)。
- `Individual` - 个人身份验证。
- `IndividualB2C` - 使用 Azure AD B2C 进行个人身份验证。
- `SingleOrg` - 对一个租户进行组织身份验证。
- `MultiOrg` - 对多个租户进行组织身份验证。
- `Windows` - Windows 身份验证。

- `--aad-b2c-instance <INSTANCE>`

要连接到的 Azure Active Directory B2C 实例。与 `IndividualB2C` 身份验证结合使用。默认值为 `https://login.microsoftonline.com/tfp/`。

- `-ssp| --suspi-policy-id <ID>`

此项目的登录和注册策略 ID。与 `IndividualB2C` 身份验证结合使用。

- `-rp| --reset-password-policy-id <ID>`

此项目的重置密码策略 ID。与 `IndividualB2C` 身份验证结合使用。

- `-ep| --edit-profile-policy-id <ID>`

此项目的编辑配置文件策略 ID。与 `IndividualB2C` 身份验证结合使用。

- `--aad-instance <INSTANCE>`

要连接到的 Azure Active Directory 实例。与 `SingleOrg` 或 `MultiOrg` 身份验证结合使用。默认值为 `https://login.microsoftonline.com/`。

- `--client-id <ID>`

此项目的客户端 ID。与 `IndividualB2C`、`SingleOrg` 或 `MultiOrg` 身份验证结合使用。默认值为 `11111111-1111-1111-1111-111111111111`。

- `--domain <DOMAIN>`

目录租户的域。与 `SingleOrg` 或 `IndividualB2C` 身份验证结合使用。默认值为 `qualified.domain.name`。

- `--tenant-id <ID>`

要连接到的目录的 TenantId ID。与 `SingleOrg` 身份验证结合使用。默认值为 `22222222-2222-2222-2222-222222222222`。

- `--callback-path <PATH>`

重定向 URI 的应用程序基路径中的请求路径。与 `SingleOrg` 或 `IndividualB2C` 身份验证结合使用。默认值为 `/signin-oidc`。

- `-r|--org-read-access`

允许此应用程序对目录进行读取访问。仅适用于 `SingleOrg` 或 `MultiOrg` 身份验证。

- `--exclude-launch-settings`

从生成的模板中排除 `launchSettings.json`。

- `--no-https`

关闭 HTTPS。此选项仅适用于 `Individual`、`IndividualB2C`、`SingleOrg` 和 `MultiOrg` 未用于 `--auth` 的情况。

- `-uld|--use-local-db`

指定应使用 LocalDB，而不使用 SQLite。仅适用于 `Individual` 或 `IndividualB2C` 身份验证。

- `--no-restore`

在项目创建期间不执行隐式还原。

Web

- `--exclude-launch-settings`

从生成的模板中排除 `launchSettings.json`。

- `-f|--framework <FRAMEWORK>`

指定目标框架。选项在 .NET Core 2.2 SDK 中不可用。

下表根据所使用的 SDK 版本号列出了默认值：

SDK 版本	框架
3.1	<code>netcoreapp3.1</code>
3.0	<code>netcoreapp3.0</code>
2.1	<code>netcoreapp2.1</code>

- `--no-restore`

在项目创建期间不执行隐式还原。

- `--no-https`

关闭 HTTPS。

mvc、webapp

- `-au|--auth <AUTHENTICATION_TYPE>`

要使用的身份验证类型。可能的值为:

- `None` - 不进行身份验证(默认)。
- `Individual` - 个人身份验证。
- `IndividualB2C` - 使用 Azure AD B2C 进行个人身份验证。
- `SingleOrg` - 对一个租户进行组织身份验证。
- `MultiOrg` - 对多个租户进行组织身份验证。
- `Windows` - Windows 身份验证。

- `--aad-b2c-instance <INSTANCE>`

要连接到的 Azure Active Directory B2C 实例。与 `IndividualB2C` 身份验证结合使用。默认值为 `https://login.microsoftonline.com/tfp/`。

- `-ssp|--susi-policy-id <ID>`

此项目的登录和注册策略 ID。与 `IndividualB2C` 身份验证结合使用。

- `-rp|--reset-password-policy-id <ID>`

此项目的重置密码策略 ID。与 `IndividualB2C` 身份验证结合使用。

- `-ep|--edit-profile-policy-id <ID>`

此项目的编辑配置文件策略 ID。与 `IndividualB2C` 身份验证结合使用。

- `--aad-instance <INSTANCE>`

要连接到的 Azure Active Directory 实例。与 `SingleOrg` 或 `MultiOrg` 身份验证结合使用。默认值为 `https://login.microsoftonline.com/`。

- `--client-id <ID>`

此项目的客户端 ID。与 `IndividualB2C`、`SingleOrg` 或 `MultiOrg` 身份验证结合使用。默认值为 `11111111-1111-1111-1111-111111111111`。

- `--domain <DOMAIN>`

目录租户的域。与 `SingleOrg` 或 `IndividualB2C` 身份验证结合使用。默认值为 `qualified.domain.name`。

- `--tenant-id <ID>`

要连接到的目录的 TenantId ID。与 `SingleOrg` 身份验证结合使用。默认值为 `22222222-2222-2222-2222-222222222222`。

- `--callback-path <PATH>`

重定向 URI 的应用程序基路径中的请求路径。与 `SingleOrg` 或 `IndividualB2C` 身份验证结合使用。默认值为 `/signin-oidc`。

- `-r|--org-read-access`

允许此应用程序对目录进行读取访问。仅适用于 `SingleOrg` 或 `MultiOrg` 身份验证。

- `--exclude-launch-settings`

从生成的模板中排除 launchSettings.json。

- `--no-https`

关闭 HTTPS。此选项仅适用于未使用 `Individual`、`IndividualB2C`SingleOrg` 和 `MultiOrg` 的情况。

- `-uld|--use-local-db`

指定应使用 LocalDB，而不使用 SQLite。仅适用于 `Individual` 或 `IndividualB2C` 身份验证。

- `-f|--framework <FRAMEWORK>`

指定目标框架。自 .NET Core 3.0 SDK 起可用的选项。

下表根据所使用的 SDK 版本号列出了默认值：

SDK	值
3.1	<code>netcoreapp3.1</code>
3.0	<code>netcoreapp3.0</code>

- `--no-restore`

在项目创建期间不执行隐式还原。

- `--use-browserlink`

在项目中添加 BrowserLink。选项在 .NET Core 2.2 和 3.1 SDK 中不可用。

angular、react

- `-au|--auth <AUTHENTICATION_TYPE>`

要使用的身份验证类型。自 .NET Core 3.0 SDK 起可用。

可能的值为：

- `None` - 不进行身份验证（默认）。
- `Individual` - 个人身份验证。

- `--exclude-launch-settings`

从生成的模板中排除 launchSettings.json。

- `--no-restore`

在项目创建期间不执行隐式还原。

- `--no-https`

关闭 HTTPS。仅当身份验证为 `None` 时，此选项才适用。

- `-uld|--use-local-db`

指定应使用 LocalDB，而不使用 SQLite。仅适用于 `Individual` 或 `IndividualB2C` 身份验证。自 .NET Core 3.0 SDK 起可用。

- `-f|--framework <FRAMEWORK>`

指定目标框架。选项在 .NET Core 2.2 SDK 中不可用。

下表根据所使用的 SDK 版本号列出了默认值：

SDK 版本	值
3.1	netcoreapp3.1
3.0	netcoreapp3.0
2.1	netcoreapp2.0

reactredux

- `--exclude-launch-settings`

从生成的模板中排除 launchSettings.json。

- `-f|--framework <FRAMEWORK>`

指定目标[框架](#)。选项在 .NET Core 2.2 SDK 中不可用。

下表根据所使用的 SDK 版本号列出了默认值：

SDK 版本	值
3.1	netcoreapp3.1
3.0	netcoreapp3.0
2.1	netcoreapp2.0

- `--no-restore`

在项目创建期间不执行隐式还原。

- `--no-https`

关闭 HTTPS。

razorclasslib

- `--no-restore`

在项目创建期间不执行隐式还原。

- `-s|--support-pages-and-views`

除了将组件添加到此库以外，还支持添加传统的 Razor 页面和视图。自 .NET Core 3.0 SDK 起可用。

webapi

- `-au|--auth <AUTHENTICATION_TYPE>`

要使用身份验证类型。可能的值为：

- `None` - 不进行身份验证（默认）。
- `IndividualB2C` - 使用 Azure AD B2C 进行个人身份验证。
- `SingleOrg` - 对一个租户进行组织身份验证。
- `Windows` - Windows 身份验证。

- `--aad-b2c-instance <INSTANCE>`
要连接到的 Azure Active Directory B2C 实例。与 `IndividualB2C` 身份验证结合使用。默认值为 `https://login.microsoftonline.com/tfp/`。
 - `-ssp|--susi-policy-id <ID>`
此项目的登录和注册策略 ID。与 `IndividualB2C` 身份验证结合使用。
 - `--aad-instance <INSTANCE>`
要连接到的 Azure Active Directory 实例。与 `SingleOrg` 身份验证结合使用。默认值为 `https://login.microsoftonline.com/`。
 - `--client-id <ID>`
此项目的客户端 ID。与 `IndividualB2C` 或 `SingleOrg` 身份验证结合使用。默认值为 `11111111-1111-1111-1111-111111111111`。
 - `--domain <DOMAIN>`
目录租户的域。与 `IndividualB2C` 或 `SingleOrg` 身份验证结合使用。默认值为 `qualified.domain.name`。
 - `--tenant-id <ID>`
要连接到的目录的 TenantId ID。与 `SingleOrg` 身份验证结合使用。默认值为 `22222222-2222-2222-2222-222222222222`。
 - `-r|--org-read-access`
允许此应用程序对目录进行读取访问。仅适用于 `SingleOrg` 身份验证。
 - `--exclude-launch-settings`
从生成的模板中排除 `launchSettings.json`。
 - `--no-https`
关闭 HTTPS。`app.UseHsts` 和 `app.UseHttpsRedirection` 未添加到 `Startup.Configure` 中。此选项仅适用于 `IndividualB2C` 或 `SingleOrg` 未用于身份验证的情况。
 - `-uld|--use-local-db`
指定应使用 LocalDB，而不使用 SQLite。仅适用于 `IndividualB2C` 身份验证。
 - `-f|--framework <FRAMEWORK>`
指定目标框架。选项在 .NET Core 2.2 SDK 中不可用。
- 下表根据所使用的 SDK 版本号列出了默认值：
- | SDK 版本 | |
|--------|----------------------------|
| 3.1 | <code>netcoreapp3.1</code> |
| 3.0 | <code>netcoreapp3.0</code> |
| 2.1 | <code>netcoreapp2.1</code> |
- `--no-restore`

在项目创建期间不执行隐式还原。

global.json

- `--sdk-version <VERSION_NUMBER>`

指定要在 global.json 文件中使用的 .NET Core SDK 版本。

示例

- 通过指定模板名称，创建 C# 控制台应用程序项目：

```
dotnet new "Console Application"
```

- 在当前目录中创建 F# 控制台应用程序项目：

```
dotnet new console -lang F#
```

- 在指定的目录中创建 .NET Standard 类库项目：

```
dotnet new classlib -lang VB -o MyLibrary
```

- 在当前目录中新建没有设置身份验证的 ASP.NET Core C# MVC 项目：

```
dotnet new mvc -au None
```

- 创建新的 xUnit 项目：

```
dotnet new xunit
```

- 列出可用于单页应用程序 (SPA) 模板的所有模板：

```
dotnet new spa -l
```

- 列出与“we”子字符串匹配的所有模板。找不到完全匹配，因此子字符串匹配针对短名称和名称列运行。

```
dotnet new we -l
```

- 尝试调用与 ng 匹配的模板。如果无法确定单个匹配项，请列出部分匹配项的模板。

```
dotnet new ng
```

- 安装 ASP.NET Core 的 SPA 模板 2.0 版：

```
dotnet new -i Microsoft.DotNet.Web.Spa.ProjectTemplates::2.0.0
```

- 列出已安装的模板及其详细信息，包括如何卸载它们：

```
dotnet new -u
```

- 在当前目录中创建 global.json，将 SDK 版本设置为 3.1.101：

```
dotnet new globaljson --sdk-version 3.1.101
```

另请参阅

- [dotnet new 自定义模板](#)
- [创建 dotnet new 自定义模板](#)
- [dotnet/dotnet-template-samples GitHub 存储库](#)
- [dotnet new 可用模板](#)

dotnet nuget delete

2020/3/18 • [Edit Online](#)

本文适用于: ✓ .NET Core 1.x SDK 及更高版本

名称

`dotnet nuget delete` - 从服务器删除或取消列出包。

摘要

```
dotnet nuget delete [<PACKAGE_NAME> <PACKAGE_VERSION>] [--force-english-output] [--interactive] [-k|--api-key]
[--no-service-endpoint]
[--non-interactive] [-s|--source]
dotnet nuget delete [-h|--help]
```

说明

`dotnet nuget delete` 命令从服务器删除或取消列出包。对于 [NuGet.org](#), 该操作将取消列出包。

参数

- `<PACKAGE_NAME>`

要删除的包的名称/ID。

- `<PACKAGE_VERSION>`

要删除的包的版本。

选项

- `--force-english-output`

使用固定的、基于英语的区域性强制运行应用程序。

- `-h|--help`

打印出有关命令的简短帮助。

- `--interactive`

对于身份验证等操作，允许命令阻止并要求手动操作。自 .NET Core 2.2 SDK 起可用的选项。

- `-k|--api-key <API_KEY>`

服务器的 API 密钥。

- `--no-service-endpoint`

不将“api/v2/package”追加至源 URL。自 .NET Core 2.1 SDK 起可用的选项。

- `--non-interactive`

不提示用户输入或确认。

- `-s|--source <SOURCE>`

指定服务器 URL。Nuget.org 的支持 URL 包括 `https://www.nuget.org`、`https://www.nuget.org/api/v3` 和 `https://www.nuget.org/api/v2/package`。对于专用源，请替换主机名(例如, `%hostname%/api/v3`)。

示例

- 删除包 `Microsoft.AspNetCore.Mvc` 的 1.0 版:

```
dotnet nuget delete Microsoft.AspNetCore.Mvc 1.0
```

- 删除包 `Microsoft.AspNetCore.Mvc` 的 1.0 版(不提示用户需要凭据或其他输入):

```
dotnet nuget delete Microsoft.AspNetCore.Mvc 1.0 --non-interactive
```

dotnet nuget locals

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.x SDK 及更高版本

名称

`dotnet nuget locals` -清除或列出本地 NuGet 资源。

摘要

```
dotnet nuget locals <CACHE_LOCATION> [(-c|--clear)|(-l|--list)] [--force-english-output]
dotnet nuget locals [-h|--help]
```

说明

`dotnet nuget locals` 命令清除或列出 http 请求缓存中的本地 NuGet 资源，临时缓存或计算机范围的全局包文件夹。

参数

- `CACHE_LOCATION`

要列出或清除的缓存位置。可以接受以下值之一：

- `all` - 表示指定的操作应用于所有缓存类型，即 http 请求缓存、全局包缓存和临时缓存。
- `http-cache` - 表示指定的操作仅应用于 http 请求缓存。其他缓存位置不受影响。
- `global-packages` - 表示指定的操作仅应用于全局包缓存。其他缓存位置不受影响。
- `temp` - 表示指定的操作仅应用于临时缓存。其他缓存位置不受影响。

选项

- `--force-english-output`

使用固定的、基于英语的区域性强制运行应用程序。

- `-h|--help`

打印出有关命令的简短帮助。

- `-c|--clear`

清除选项对指定的缓存类型执行清除操作。缓存目录的内容被以递归方式删除。正在执行的用户/组必须具有对缓存目录中的文件的相关权限。反之，则显示错误，指示未清除的文件/文件夹。

- `-l|--list`

列表选项用于显示指定缓存类型的位置。

示例

- 显示所有本地缓存目录的路径(http 缓存目录、全局包缓存目录和临时缓存目录)：

```
dotnet nuget locals all -l
```

- 显示本地 http 缓存录的路径：

```
dotnet nuget locals http-cache --list
```

- 清除所有本地缓存目录的文件 (http 缓存目录、全局包缓存目录和临时缓存目录)：

```
dotnet nuget locals all --clear
```

- 清除本地全局包缓存目录中的所有文件：

```
dotnet nuget locals global-packages -c
```

- 清除本地临时缓存目录中的所有文件：

```
dotnet nuget locals temp -c
```

故障排除

有关使用 `dotnet nuget locals` 命令时的常见问题和错误的信息，请参阅[管理 NuGet 缓存](#)。

dotnet nuget push

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.x SDK 及更高版本

名称

`dotnet nuget push` - 将包推送到服务器，并将其发布。

摘要

```
dotnet nuget push [<ROOT>] [-d|--disable-buffering] [--force-english-output] [--interactive] [-k|--api-key] [-n|--no-symbols]
    [--no-service-endpoint] [-s|--source] [--skip-duplicate] [-sk|--symbol-api-key] [-ss|--symbol-source] [-t|--timeout]
dotnet nuget push [-h|--help]
```

说明

`dotnet nuget push` 将包推送到服务器，并将其发布。push 命令使用在系统的 NuGet 配置文件或配置文件链中找到的服务器和凭据详细信息。有关配置文件的详细信息，请参阅 [Configuring NuGet Behavior](#)（配置 NuGet 行为）。通过加载 `%AppData%\NuGet\NuGet.config`（Windows）或 `$HOME/local/share`（Linux/macOS）获得 NuGet 的默认配置，然后加载任意 `nuget.config` 或 `.nuget\nuget.config`，从驱动器的根目录开始，并在当前目录中结束。

参数

- `ROOT`

指定要推送的包的文件路径。

选项

- `-d|--disable-buffering`

当推送到 HTTP(S) 服务器以减少内存使用率时，禁用缓冲。

- `--force-english-output`

使用固定的、基于英语的区域性强制运行应用程序。

- `-h|--help`

打印出有关命令的简短帮助。

- `--interactive`

对于身份验证等操作，允许命令阻止并要求手动操作。自 .NET Core 2.2 SDK 起可用的选项。

- `-k|--api-key <API_KEY>`

服务器的 API 密钥。

- `-n|--no-symbols`

不推送符号(即使存在)。

- `--no-service-endpoint`

不将“api/v2/package”追加至源 URL。自 .NET Core 2.1 SDK 起可用的选项。

- `-s|--source <SOURCE>`

指定服务器 URL。除非在 NuGet 配置文件中设置了 `DefaultPushSource` 配置值，否则此选项是必需的。

- `--skip-duplicate`

将多个包推送到 HTTP(S) 服务器时，将任何 409 冲突响应视为警告，以便可以继续推送。自 .NET Core 3.1 SDK 起可用。

- `-sk|--symbol-api-key <API_KEY>`

符号服务器的 API 密钥。

- `-ss|--symbol-source <SOURCE>`

指定符号服务器 URL。

- `-t|--timeout <TIMEOUT>`

指定推送到服务器的超时(秒)。默认值为 300 秒(5 分钟)。指定为 0(零秒)将应用默认值。

示例

- 将 `foo.nupkg` 推送到默认推送源(指定 API 密钥)：

```
dotnet nuget push foo.nupkg -k 4003d786-cc37-4004-bfdf-c4f3e8ef9b3a
```

- 将 `foo.nupkg` 推送到官方 NuGet 服务器，以指定 API 密钥：

```
dotnet nuget push foo.nupkg -k 4003d786-cc37-4004-bfdf-c4f3e8ef9b3a -s  
https://api.nuget.org/v3/index.json
```

- 将 `foo.nupkg` 推送到自定义推送源 (指定 API 密钥) `https://customsource/`：

```
dotnet nuget push foo.nupkg -k 4003d786-cc37-4004-bfdf-c4f3e8ef9b3a -s https://customsource/
```

- 将 `foo.nupkg` 推送到默认推送源：

```
dotnet nuget push foo.nupkg
```

- 将 `foo.symbols.nupkg` 推送到默认符号源：

```
dotnet nuget push foo.symbols.nupkg
```

- 将 `foo.nupkg` 推送到默认推送源(指定 360 秒超时时间)：

```
dotnet nuget push foo.nupkg --timeout 360
```

- 将当前目录中的所有 `.nupkg` 文件推送到默认推送源：

```
dotnet nuget push *.nupkg
```

NOTE

如果此命令不起作用，则可能是较旧版本的 SDK(.NET Core 2.1 SDK 及更早版本)中的 bug 导致的。要解决此问题，请升级 SDK 版本或改为运行以下命令：dotnet nuget push **/*.nupkg

- 推送所有 .nupkg 文件，即使 HTTP(S) 服务器返回了 409 冲突响应也是如此：

```
dotnet nuget push *.nupkg --skip-duplicate
```

dotnet nuget add source

2020/4/2 • [Edit Online](#)

本文适用于：✓ .NET Core 3.1.200 SDK 及更高版本

“属性”

`dotnet nuget add source` - 添加 NuGet 源。

摘要

```
dotnet nuget add source <PACKAGE_SOURCE_PATH> [--name] [--username]
[--password] [--store-password-in-clear-text] [--valid-authentication-types]
[--configfile]
dotnet nuget add source [-h|--help]
```

描述

`dotnet nuget add source` 命令将新的包源添加到 NuGet 配置文件中。

自变量

- `<PACKAGE_SOURCE_PATH>`

包源的路径。

选项

- `--configfile`

NuGet 配置文件。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅[常见的 NuGet 配置](#)。

- `-n|--name`

源的名称。

- `-p|--password`

连接到已验证源时要使用的密码。

- `--store-password-in-clear-text`

通过禁用密码加密允许存储可移植包源凭据。

- `-u|--username`

连接到已经过身份验证的源时要使用的用户名。

- `--valid-authentication-types`

此源的有效身份验证类型的逗号分隔列表。如果服务器公布 NTLM 或协商，并且你必须使用基本机制发送凭据（例如，在本地 Azure DevOps Server 中使用 PAT 时），则将此项设置为 `basic`。其他有效值包括 `negotiate`、`kerberos`、`ntlm` 和 `digest`，但这些值不太可能有用。

示例

- 将 `nuget.org` 添加为源:

```
dotnet nuget add source https://api.nuget.org/v3/index.json -n nuget.org
```

- 将 `c:\packages` 添加为本地源:

```
dotnet nuget add source c:\packages
```

- 添加需要身份验证的源:

```
dotnet nuget add source https://someServer/myTeam -n myTeam -u myUsername -p myPassword --store-password-in-clear-text
```

- 添加需要身份验证的源(然后继续安装凭据提供程序):

```
dotnet nuget add source https://azureartifacts.microsoft.com/myTeam -n myTeam
```

请参阅

- [NuGet.config 文件中的包源部分](#)
- [sources 命令 \(nuget.exe\)](#)

dotnet nuget disable source

2020/4/2 • [Edit Online](#)

本文适用于：✓ .NET Core 3.1.200 SDK 及更高版本

“属性”

`dotnet nuget disable source` - 禁用 NuGet 源。

摘要

```
dotnet nuget disable source <NAME> [--configfile]
dotnet nuget disable source [-h|--help]
```

描述

`dotnet nuget disable source` 命令在 NuGet 配置文件中禁用现有源。

自变量

- `NAME`

源的名称。

选项

- `--configfile`

NuGet 配置文件。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅[常见的 NuGet 配置](#)。

示例

- 禁用名为 `mySource` 的源：

```
dotnet nuget disable source mySource
```

请参阅

- [NuGet.config 文件中的包源部分](#)
- [sources 命令 \(nuget.exe\)](#)

dotnet nuget enable source

2020/4/2 • [Edit Online](#)

本文适用于：✓ .NET Core 3.1.200 SDK 及更高版本

“属性”

`dotnet nuget enable source` - 启用 NuGet 源。

摘要

```
dotnet nuget enable source <NAME> [--configfile]
dotnet nuget enable source [-h|--help]
```

描述

`dotnet nuget enable source` 命令在 NuGet 配置文件中启用现有源。

自变量

- `NAME`

源的名称。

选项

- `--configfile`

NuGet 配置文件。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅[常见的 NuGet 配置](#)。

示例

- 启用名为 `mySource` 的源：

```
dotnet nuget enable source mySource
```

请参阅

- [NuGet.config 文件中的包源部分](#)
- [sources 命令 \(nuget.exe\)](#)

dotnet nuget list source

2020/4/2 • [Edit Online](#)

本文适用于：✓ .NET Core 3.1.200 SDK 及更高版本

“属性”

`dotnet nuget list source` - 列出所有配置的 NuGet 源。

摘要

```
dotnet nuget list source [--format] [--configfile]
dotnet nuget list source [-h|--help]
```

描述

`dotnet nuget list source` 命令列出 NuGet 配置文件中的所有现有源。

选项

- `--configfile`

NuGet 配置文件。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅[常见的 NuGet 配置](#)。

- `--format`

list 命令输出的格式为：`Detailed`（默认值）和 `Short`。

示例

- 列出当前目录中的已配置源：

```
dotnet nuget list source
```

请参阅

- [NuGet.config 文件中的包源部分](#)
- [sources 命令 \(nuget.exe\)](#)

dotnet nuget remove source

2020/4/2 • [Edit Online](#)

本文适用于：✓ .NET Core 3.1.200 SDK 及更高版本

“属性”

`dotnet nuget remove source` - 删除 NuGet 源。

摘要

```
dotnet nuget remove source <NAME> [--configfile]
dotnet nuget remove source [-h|--help]
```

描述

`dotnet nuget remove source` 命令从 NuGet 配置文件中删除现有源。

自变量

- `NAME`

源的名称。

选项

- `--configfile`

NuGet 配置文件。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅[常见的 NuGet 配置](#)。

示例

- 删除名为 `mySource` 的源：

```
dotnet nuget remove source mySource
```

请参阅

- [NuGet.config 文件中的包源部分](#)
- [sources 命令 \(nuget.exe\)](#)

dotnet nuget update source

2020/4/2 • [Edit Online](#)

本文适用于：✓ .NET Core 3.1.200 SDK 及更高版本

“属性”

`dotnet nuget update source` - 更新 NuGet 源。

摘要

```
dotnet nuget update source <NAME> [--source] [--username]
[--password] [--store-password-in-clear-text] [--valid-authentication-types]
[--configfile]
dotnet nuget update source [-h|--help]
```

描述

`dotnet nuget update source` 命令在 NuGet 配置文件中更新现有源。

自变量

- `<NAME>`

源的名称。

选项

- `--configfile`

NuGet 配置文件。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅[常见的 NuGet 配置](#)。

- `-p|--password`

连接到已验证源时要使用的密码。

- `-s|--source`

包源的路径。

- `--store-password-in-clear-text`

通过禁用密码加密允许存储可移植包源凭据。

- `-u|--username`

连接到已经过身份验证的源时要使用的用户名。

- `--valid-authentication-types`

此源的有效身份验证类型的逗号分隔列表。如果服务器公布 NTLM 或协商，并且你必须使用基本机制发送凭据（例如，在本地 Azure DevOps Server 中使用 PAT 时），则将此项设置为 `basic`。其他有效值包括 `negotiate`、`kerberos`、`ntlm` 和 `digest`，但这些值不太可能有用。

示例

- 更新名为 `mySource` 的源：

```
dotnet nuget update source mySource --source c:\packages
```

请参阅

- [NuGet.config 文件中的包源部分](#)
- [sources 命令 \(nuget.exe\)](#)

dotnet 包

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.x SDK 及更高版本

名称

`dotnet pack` - 将代码打包到 NuGet 包。

摘要

```
dotnet pack [<PROJECT>|<SOLUTION>] [-c|--configuration] [--force] [--include-source] [--include-symbols] [--interactive]
    [--no-build] [--no-dependencies] [--no-restore] [--nologo] [-o|--output] [--runtime] [-s|--serviceable]
    [-v|--verbosity] [--version-suffix]
dotnet pack [-h|--help]
```

说明

`dotnet pack` 命令生成项目并创建 NuGet 包。该命令的结果是一个 NuGet 包，也就是一个 `.nupkg` 文件。

如果要生成包含调试符号的包，可以使用以下两个选项：

- `--include-symbols` : 该选项用于创建符号包。
- `--include-source` : 该选项用于创建带有 `src` 文件夹的符号包，该文件夹包含源文件。

将被打包项目的 NuGet 依赖项添加到 `.nuspec` 文件，以便在安装包时可以进行正确解析。项目到项目的引用不会被打包到项目内。目前，如果具有项目到项目的依赖项，则每个项目均必须包含一个包。

默认情况下，`dotnet pack` 先构建项目。如果希望避免此行为，则传递 `--no-build` 选项。此选项在持续集成 (CI) 生成方案中通常非常有用，你可以知道代码是之前生成的。

可向 `dotnet pack` 命令提供 MSBuild 属性，用于打包进程。有关详细信息，请参阅 [NuGet 元数据属性](#) 和 [MSBuild 命令行引用](#)。示例部分介绍了如何在不同的情况下使用 MSBuild `-p` 开关。

默认情况下，Web 项目不可打包。若要覆盖默认行为，请将以下属性添加到 `.csproj` 文件中：

```
<PropertyGroup>
    <IsPackable>true</IsPackable>
</PropertyGroup>
```

NOTE

从 .NET Core 2.0 开始，无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet build` 和 `dotnet run`。在执行显式还原有意义的某些情况下，例如 [Azure DevOps Services 中的持续集成生成](#) 中，或在需要显式控制还原发生时间的生成系统中，它仍然是有效的命令。

以长格式传递命令时，该命令也支持 `dotnet restore` 选项（例如，`--source`）。不支持缩写选项，例如 `-s`。

参数

要打包的项目或解决方案。它可能是 `csproj` 文件、解决方案文件或目录的路径。如果未指定，此命令会搜索当前目录，以获取项目文件或解决方案文件。

选项

- `-c|--configuration <CONFIGURATION>`

定义生成配置。大多数项目的默认配置为 `Debug`，但你可以覆盖项目中的生成配置设置。

- `--force`

强制解析所有依赖项，即使上次还原已成功，也不例外。指定此标记等同于删除 `project.assets.json` 文件。
。

- `-h|--help`

打印出有关命令的简短帮助。

- `--include-source`

除输出目录中的常规 NuGet 包外，还包括调试符号 NuGet 包。源文件包括在符号包内的 `src` 文件夹中。

- `--include-symbols`

除输出目录中的常规 NuGet 包外，还包括调试符号 NuGet 包。

- `--interactive`

允许命令停止并等待用户输入或操作（例如，完成身份验证）。自 .NET Core 3.0 SDK 起可用。

- `--no-build`

打包前不生成项目。还将隐式设置 `--no-restore` 标记。

- `--no-dependencies`

忽略项目间引用，仅还原根项目。

- `--no-restore`

运行此命令时不执行隐式还原。

- `--nologo`

不显示启动版权标志或版权消息。自 .NET Core 3.0 SDK 起可用。

- `-o|--output <OUTPUT_DIRECTORY>`

将生成的包放置在指定目录。

- `--runtime <RUNTIME_IDENTIFIER>`

指定要为其还原包的目标运行时。有关运行时标识符 (RID) 的列表，请参阅 [RID 目录](#)。

- `-s|--serviceable`

设置包中可用的标志。有关详细信息，请参阅 [.NET 博客：.NET 4.5.1 支持 .NET NuGet 库的 Microsoft 安全更新](#)。

- `--version-suffix <VERSION_SUFFIX>`

定义项目中 `$(VersionSuffix)` MSBuild 属性的值。

- `-v|--verbosity <LEVEL>`

设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。

示例

- 打包当前目录中的项目：

```
dotnet pack
```

- 打包 `app1` 项目：

```
dotnet pack ~/projects/app1/project.csproj
```

- 打包当前目录中的项目并将生成的包放置到 `nupkgs` 文件夹：

```
dotnet pack --output nupkgs
```

- 将当前目录中的项目打包到 `nupkgs` 文件夹并跳过生成步骤：

```
dotnet pack --no-build --output nupkgs
```

- 将项目的版本后缀配置为 `<VersionSuffix>$(VersionSuffix)</VersionSuffix>`.csproj 文件中的，使用给定的后缀打包当前项目，并更新生成的程序包版本：

```
dotnet pack --version-suffix "ci-1234"
```

- 使用 `2.1.0` MSBuild 属性将包版本设置为 `PackageVersion`：

```
dotnet pack -p:PackageVersion=2.1.0
```

- 打包特定目标框架的项目：

```
dotnet pack -p:TargetFrameworks=net45
```

- 打包项目，并使用特定运行时 (Windows 10) 进行还原操作：

```
dotnet pack --runtime win10-x64
```

- 使用 `.nuspec` 文件 打包项目：

```
dotnet pack ~/projects/app1/project.csproj -p:NuspecFile=~/projects/app1/project.nuspec -p:NuspecBasePath=~/projects/app1/nuget
```

dotnet publish

2020/4/9 • [Edit Online](#)

本文适用于：✓ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet publish` - 将应用程序及其依赖项发布到文件夹以部署到托管系统。

摘要

```
dotnet publish [<PROJECT>|<SOLUTION>] [-c|--configuration]
    [-f|--framework] [--force] [--interactive] [--manifest]
    [--no-build] [--no-dependencies] [--no-restore] [--nologo]
    [-o|--output] [-r|--runtime] [--self-contained]
    [--no-self-contained] [-v|--verbosity] [--version-suffix]

dotnet publish [-h|--help]
```

描述

`dotnet publish` 编译应用程序、读取 project 文件中指定的所有依赖项并将生成的文件集发布到目录。输出包括以下资产：

- 扩展名为 dll 的程序集中的中间语言 (IL) 代码。
- 包含项目所有依赖项的 .deps.json 文件。
- .runtimeconfig.json 文件，其中指定了应用程序所需的共享运行时，以及运行时的其他配置选项（例如垃圾回收类型）。
- 应用程序的依赖项，将这些依赖项从 NuGet 缓存复制到输出文件夹。

`dotnet publish` 命令的输出可供部署至托管系统（例如服务器、电脑、Mac、笔记本电脑）以便执行。若要准备用于部署的应用程序，这是唯一正式受支持的方法。根据项目指定的部署的类型，托管系统不一定已在其上安装 .NET Core 共享运行时。有关详细信息，请参阅使用 .NET Core CLI 发布 .NET Core 应用。

MSBuild

`dotnet publish` 命令调用 MSBuild，后者会调用 Publish 目标。任何传递给 `dotnet publish` 的参数都将传递给 MSBuild。`-c` 和 `-o` 参数分别映射到 MSBuild 的 `Configuration` 和 `OutputPath` 属性。

`dotnet publish` 命令接受 MSBuild 选项，如用来设置属性的 `-p` 和用来定义记录器的 `-l`。例如，可以使用以下格式设置 MSBuild 属性：`-p:<NAME>=<VALUE>`。还可以通过引用 .pubxml 文件来设置与发布相关的属性，例如：

```
dotnet publish -p:PublishProfile=Properties\PublishProfiles\FolderProfile.pubxml
```

有关更多信息，请参见以下资源：

- [MSBuild 命令行参考](#)
- [用于 ASP.NET Core 应用部署的 Visual Studio 发布配置文件 \(.pubxml\)](#)
- [dotnet msbuild](#)

自变量

- `PROJECT|SOLUTION`

要发布的项目或解决方案。

- `PROJECT` 是 C#、F# 或 Visual Basic 项目文件的路径和文件名，或包含 C#、F# 或 Visual Basic 项目文件的目录的路径。如果未指定目录，则默认为当前目录。
- `SOLUTION` 是解决方案文件(扩展名为 .sln)的路径和文件名，或包含解决方案文件的目录的路径。如果未指定目录，则默认为当前目录。自 .NET Core 3.0 SDK 起可用。

选项

- `-c|--configuration <CONFIGURATION>`

定义生成配置。大多数项目的默认配置为 `Debug`，但你可以覆盖项目中的生成配置设置。

- `-f|--framework <FRAMEWORK>`

为指定的[目标框架](#)发布应用程序。必须在项目文件中指定目标框架。

- `--force`

强制解析所有依赖项，即使上次还原已成功，也不例外。指定此标记等同于删除 project.assets.json 文件。

- `-h|--help`

打印出有关命令的简短帮助。

- `--interactive`

允许命令停止并等待用户输入或操作。例如，完成身份验证。自 .NET Core 3.0 SDK 起可用。

- `--manifest <PATH_TO_MANIFEST_FILE>`

指定一个或多个[目标清单](#)，用于剪裁与应用程序一同发布的一组包。清单文件是 `dotnet store` 命令输出的一部分。若要指定多个清单，请为每个清单添加一个 `--manifest` 选项。

- `--no-build`

发布前不生成项目。还将隐式设置 `--no-restore` 标记。

- `--no-dependencies`

忽略项目间引用，仅还原根项目。

- `--nologo`

不显示启动版权标志或版权消息。自 .NET Core 3.0 SDK 起可用。

- `--no-restore`

运行此命令时不执行隐式还原。

- `-o|--output <OUTPUT_DIRECTORY>`

指定输出目录的路径。

如果未指定，则默认为依赖于运行时的可执行文件和跨平台二进制文件的路径
[project_file_folder]/bin/[configuration]/[framework]/publish/。默认为独立的可执行文件路径
[project_file_folder]/bin/[configuration]/[framework]/[runtime]/publish/。

- .NET Core 3.x SDK 和更高版本

如果在发布项目时指定了相对路径，则生成的输出目录相对于当前工作目录，而不是项目文件位置。

如果在发布解决方案时指定了相对路径，则所有项目的所有输出都会进入相对于当前工作目录的指定文件夹中。若要使发布输出进入每个项目的单独文件夹，请使用 msbuild `PublishDir` 属性（而不是 `--output` 选项）指定相对路径。例如，`dotnet publish -p:PublishDir=.\publish` 将每个项目的发布输出发送到包含项目文件的文件夹下的 `publish` 文件夹中。

- .NET Core 2.x SDK

如果在发布项目时指定了相对路径，则生成的输出目录相对于项目文件位置，而不是当前工作目录。

如果在发布解决方案时指定了相对路径，则每个项目的输出会进入相对于项目文件位置的单独文件夹中。如果在发布解决方案时指定了绝对路径，则所有项目的所有发布输出都会进入指定文件夹中。

- `--self-contained [true|false]`

与应用程序一同发布 .NET Core 运行时，因此无需在目标计算机上安装运行时。如果指定了运行时标识符，并且项目是可执行项目（而不是库项目），则默认值为 `true`。有关详细信息，请参阅 [.NET Core 应用程序发布和使用 .NET Core CLI 发布 .NET Core 应用](#)。

如果在未指定 `true` 或 `false` 的情况下使用此选项，则默认值为 `true`。在这种情况下，请不要紧接在 `--self-contained` 后放置解决方案或项目参数，因为该位置需要 `true` 或 `false`。

- `--no-self-contained`

等效于 `--self-contained false`。自 .NET Core 3.0 SDK 起可用。

- `-r|--runtime <RUNTIME_IDENTIFIER>`

发布针对给定运行时的应用程序。有关运行时标识符 (RID) 的列表，请参阅 [RID 目录](#)。有关详细信息，请参阅 [.NET Core 应用程序发布和使用 .NET Core CLI 发布 .NET Core 应用](#)。

- `-v|--verbosity <LEVEL>`

设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值是 `minimal`。

- `--version-suffix <VERSION_SUFFIX>`

定义版本后缀来替换项目文件的版本字段中的星号 (*)。

示例

- 为当前目录中的项目创建一个 [依赖于运行时的跨平台二进制文件](#)：

```
dotnet publish
```

自 .NET Core 3.0 SDK 起，此示例还为当前平台创建[依赖于运行时的可执行文件](#)。

- 针对特定运行时，为当前目录中的项目创建[独立可执行文件](#)：

```
dotnet publish --runtime osx.10.11-x64
```

项目文件中必须包含 RID。

- 针对特定平台，为当前目录中的项目创建[依赖于运行时的可执行文件](#)：

```
dotnet publish --runtime osx.10.11-x64 --self-contained false
```

项目文件中必须包含 RID。此示例适用于.NET Core 3.0 SDK 及更高版本。

- 针对特定运行时和目标框架，在当前目录中发布项目：

```
dotnet publish --framework netcoreapp3.1 --runtime osx.10.11-x64
```

- 发布指定的项目文件：

```
dotnet publish ~/projects/app1/app1.csproj
```

- 发布当前应用程序，但在还原操作期间不还原项目到项目 (P2P) 引用，只还原根项目：

```
dotnet publish --no-dependencies
```

请参阅

- [.NET Core 应用程序发布概述](#)
- [使用 .NET Core CLI 发布 .NET Core 应用](#)
- [目标框架](#)
- [运行时标识符 \(RID\) 目录](#)
- [处理 macOS Catalina 公证](#)
- [已发布应用程序的目录结构](#)
- [MSBuild 命令行参考](#)
- [用于 ASP.NET Core 应用部署的 Visual Studio 发布配置文件 \(.pubxml\)](#)
- [dotnet msbuild](#)

dotnet restore

2020/3/18 • [Edit Online](#)

本文适用于: ✓ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet restore` - 恢复项目的依赖项和工具。

摘要

```
dotnet restore [<ROOT>] [--configfile] [--disable-parallel]
    [--force] [--ignore-failed-sources] [--no-cache]
    [--no-dependencies] [--packages] [-r|--runtime]
    [-s|--source] [-v|--verbosity] [--interactive]

dotnet restore [-h|--help]
```

描述

`dotnet restore` 命令使用 NuGet 还原依赖项以及在 project 文件中指定的特定于项目的工具。默认情况下会并行执行对依赖项和工具的还原。

为了还原依赖项, NuGet 需要包所在的源。通常通过“nuGet.config”配置文件提供源。安装 .NET Core SDK 时提供一个默认的配置文件。可以通过在项目目录中创建自己的 nuGet.config 文件来指定其他源。可以使用 `-s` 选项替代 nuget.config 源。

对于依赖项, 使用 `--packages` 参数指定还原操作期间放置还原包的位置。如未指定, 将使用默认的 NuGet 包缓存, 可在所有操作系统上的用户主目录中的 `.nuget/packages` 目录找到它。例如 Linux 上的 `/home/user1` 或 Windows 上的 `C:\Users\user1`。

对于特定于项目的工具, `dotnet restore` 首先还原打包工具所在的包, 然后继续还原 project 文件中指定的工具依赖项。

nuget.config 差异

`dotnet restore` 命令的行为会受 Nuget.Config 文件(如果有)中某些设置的影响。例如, 在 NuGet.Config 中设置 `globalPackagesFolder` 会将还原的 NuGet 包置于指定的文件夹中。这是在 `dotnet restore` 命令中指定 `--packages` 选项的替代方法。有关详细信息, 请参阅 [nuget.config 参考](#)。

有三个 `dotnet restore` 可忽略的特定设置:

- [bindingRedirects](#)

绑定重定向不适用于 `<PackageReference>` 元素, 并且 .NET Core 仅支持 NuGet 包的 `<PackageReference>` 元素。

- [解决方案](#)

此设置特定于 Visual Studio, 不适用于 .NET Core。.Net Core 不使用 `packages.config` 文件, 而是使用 NuGet 包的 `<PackageReference>` 元素。

- [trustedSigners](#)

此设置不适用，如 NuGet 尚不支持跨平台验证受信任包所述。

隐式还原

在运行下列命令时，如有必要，会隐式运行 `dotnet restore` 命令：

- `dotnet new`
- `dotnet build`
- `dotnet build-server`
- `dotnet run`
- `dotnet test`
- `dotnet publish`
- `dotnet pack`

在大多数情况下无需显式使用 `dotnet restore` 命令。

有时，隐式运行 `dotnet restore` 可能不方便。例如，某些自动化系统（如生成系统）需要显式调用 `dotnet restore`，以控制还原发生的时间，以便可以控制网络使用量。要防止隐式运行 `dotnet restore`，可以通过上述任意命令使用 `--no-restore` 标记以禁用隐式还原。

自变量

- `ROOT`

要还原的项目文件的可选路径。

选项

- `--configfile <FILE>`

供还原操作使用的 NuGet 配置文件 (`nuget.config`)。

- `--disable-parallel`

禁用并行还原多个项目。

- `--force`

强制解析所有依赖项，即使上次还原已成功，也不例外。指定此标记等同于删除 `project.assets.json` 文件。

- `-h|--help`

打印出有关命令的简短帮助。

- `--ignore-failed-sources`

如果存在符合版本要求的包，则源失败时警告。

- `--no-cache`

指定不缓存包和 HTTP 请求。

- `--no-dependencies`

当使用项目到项目 (P2P) 引用还原项目时，还原根项目，不还原引用。

- `--packages <PACKAGES_DIRECTORY>`

指定还原包的目录。

- `-r|--runtime <RUNTIME_IDENTIFIER>`

指定程序包还原的运行时。这用于还原 `.csproj` 文件中的 `<RuntimeIdentifiers>` 标记中未显式列出的运行时的程序包。有关运行时标识符 (RID) 的列表, 请参阅 [RID 目录](#)。通过多次指定此选项提供多个 RID。

- `-s|--source <SOURCE>`

指定要在还原操作期间使用的 NuGet 包源。此设置会替代 `nuget.config` 文件中指定的所有源。多次指定此选项可以提供多个源。

- `--verbosity <LEVEL>`

设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值是 `minimal`。

- `--interactive`

允许命令停止并等待用户输入或操作(例如, 完成身份验证)。从 .NET Core 2.1.400 开始。

示例

- 还原当前目录中项目的依赖项和工具:

```
dotnet restore
```

- 还原在给定路径中找到的 `app1` 项目的依赖项和工具:

```
dotnet restore ~/projects/app1/app1.csproj
```

- 通过将提供的文件路径用作源, 在当前目录中还原项目的依赖项和工具:

```
dotnet restore -s c:\packages\mypackages
```

- 通过将提供的两个文件路径用作源, 在当前目录中还原项目的依赖项和工具:

```
dotnet restore -s c:\packages\mypackages -s c:\packages\myotherpackages
```

- 还原当前目录中项目的依赖项和工具, 并显示详细的输出:

```
dotnet restore --verbosity detailed
```

dotnet run

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.x SDK 及更高版本

“属性”

`dotnet run` - 无需任何显式编译或启动命令即可运行源代码。

摘要

```
dotnet run [-c|--configuration] [-f|--framework] [--force] [--interactive] [--launch-profile]
[--no-build] [--no-dependencies] [--no-launch-profile] [--no-restore] [-p|--project]
[-r|--runtime] [-v|--verbosity] [[--] [application arguments]]
dotnet run [-h|--help]
```

描述

`dotnet run` 命令为从源代码使用一个命令运行应用程序提供了一个方便的选项。这对从命令行中进行快速迭代开发很有帮助。命令取决于生成代码的 `dotnet build` 命令。对于此生成的任何要求，例如项目必须首先还原，同样适用于 `dotnet run`。

输出文件会写入到默认位置，即 `bin/<configuration>/<target>`。例如，如果具有 `netcoreapp2.1` 应用程序并且运行 `dotnet run`，则输出置于 `bin/Debug/netcoreapp2.1`。将根据需要覆盖文件。临时文件将置于 `obj` 目录。

如果该项目指定多个框架，在不使用 `-f|--framework <FRAMEWORK>` 选项指定框架时，执行 `dotnet run` 将导致错误。

在项目上下文，而不是生成程序集中使用 `dotnet run` 命令。如果尝试改为运行依赖于框架的应用程序 DLL，则必须在不使用命令的情况下使用 `dotnet`。例如，若要运行 `myapp.dll`，请使用：

```
dotnet myapp.dll
```

有关 `dotnet` 驱动程序的详细信息，请参阅 [.NET Core 命令行工具 \(CLI\)](#) 主题。

若要运行应用程序，`dotnet run` 命令需从 NuGet 缓存解析共享运行时之外的应用程序依赖项。因为它使用缓存的依赖项，因此，不推荐在生产中使用 `dotnet run` 来运行应用程序。相反，使用 [dotnet publish 命令创建部署](#)，并部署已发布的输出。

NOTE

从 .NET Core 2.0 开始，无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet build` 和 `dotnet run`。在执行显式还原有意义的某些情况下，例如 [Azure DevOps Services 中的持续集成生成](#) 中，或在需要显式控制还原发生时间的生成系统中，它仍然是有效的命令。

以长格式传递命令时，该命令也支持 `dotnet restore` 选项（例如，`--source`）。不支持缩写选项，例如 `-s`。

选项

- `--`

将参数分隔到正在运行的应用程序的参数的 `dotnet run`。在此分隔符后的所有参数均传递给已运行的应用程序。

- `-c|--configuration <CONFIGURATION>`

定义生成配置。大多数项目的默认配置为 `Debug`，但你可以覆盖项目中的生成配置设置。

- `-f|--framework <FRAMEWORK>`

使用指定 [框架](#) 生成并运行应用。框架必须在项目文件中进行指定。

- `--force`

强制解析所有依赖项，即使上次还原已成功，也不例外。指定此标记等同于删除 `project.assets.json` 文件。

- `-h|--help`

打印出有关命令的简短帮助。

- `--interactive`

允许命令停止并等待用户输入或操作（例如，完成身份验证）。自 .NET Core 3.0 SDK 起可用。

- `--launch-profile <NAME>`

启动应用程序时要使用的启动配置文件（若有）的名称。启动配置文件在 `launchSettings.json` 文件中进行定义，通常称为 `Development`、`Staging` 和 `Production`。有关详细信息，请参阅[使用多个环境](#)。

- `--no-build`

运行前不生成项目。还隐式设置 `--no-restore` 标记。

- `--no-dependencies`

当使用项目到项目（P2P）引用还原项目时，还原根项目，不还原引用。

- `--no-launch-profile`

不尝试使用 `launchSettings.json` 配置应用程序。

- `--no-restore`

运行此命令时不执行隐式还原。

- `-p|--project <PATH>`

指定要运行的项目文件的路径（文件夹名称或完整路径）。如果未指定，则默认为当前目录。

- `-r|--runtime <RUNTIME_IDENTIFIER>`

指定要为其还原包的目标运行时。有关运行时标识符（RID）的列表，请参阅[RID 目录](#)。自 .NET Core 3.0 SDK 起可用的 `-r` 简短选项。

- `-v|--verbosity <LEVEL>`

设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `m`。自 .NET Core 2.1 SDK 起可用。

示例

- 运行当前目录中的项目：

```
dotnet run
```

- 运行指定的项目：

```
dotnet run --project ./projects/proj1/proj1.csproj
```

- 运行当前目录中的项目(在本例中, `--help` 参数被传递到应用程序, 因为使用了空白的 `--` 选项)：

```
dotnet run --configuration Release -- --help
```

- 在当前仅显示最小输出的目录中还原项目的依赖项和工具, 然后运行项目(.NET Core SDK 2.0 及更高版本)：

```
dotnet run --verbosity m
```

dotnet sln

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.x SDK 及更高版本

名称

`dotnet sln` - 在 .NET Core 解决方案文件中列出或修改项目。

摘要

```
dotnet sln [<SOLUTION_FILE>] [command] [-h|--help]
```

说明

使用 `dotnet sln` 命令，可以便捷地在解决方案文件中列出和修改项目。

若要使用 `dotnet sln` 命令，必须存在解决方案文件。如果需要创建一个解决方案文件，请使用 `dotnet new` 命令，如下例所示：

```
dotnet new sln
```

参数

- `SOLUTION_FILE`

要使用的解决方案文件。如果省略此参数，此命令会搜索当前目录来获取一个解决方案文件。如果未找到解决方案文件或找到多个解决方案文件，则该命令将失败。

选项

- `-h|--help`

打印出有关如何使用命令的说明。

命令

`list`

列出解决方案文件中的所有项目。

摘要

```
dotnet sln list [-h|--help]
```

参数

- `SOLUTION_FILE`

要使用的解决方案文件。如果省略此参数，此命令会搜索当前目录来获取一个解决方案文件。如果未找到解决方案文件或找到多个解决方案文件，则该命令将失败。

选项

- `-h|--help`

打印出有关如何使用命令的说明。

add

将一个或多个项目添加到解决方案文件。

摘要

```
dotnet sln [<SOLUTION_FILE>] add [--in-root] [-s|--solution-folder] <PROJECT_PATH> [<PROJECT_PATH>...]  
dotnet sln add [-h|--help]
```

参数

- `SOLUTION_FILE`

要使用的解决方案文件。如果未指定，此命令会搜索当前目录以获取一个解决方案文件，如果找到多个解决方案文件，则该命令将失败。

- `PROJECT_PATH`

要添加到解决方案的一个或多个项目的路径。Unix/Linux shell [glob 模式](#) 扩展由 `dotnet sln` 命令正确处理。

选项

- `-h|--help`

打印出有关如何使用命令的说明。

- `--in-root`

将项目放在解决方案的根目录下，而不是创建解决方案文件夹。自 .NET Core 3.0 SDK 起可用。

- `-s|--solution-folder`

要将项目添加到的目标解决方案文件夹路径。自 .NET Core 3.0 SDK 起可用。

remove

从解决方案文件中删除一个或多个项目。

摘要

```
dotnet sln [<SOLUTION_FILE>] remove <PROJECT_PATH> [<PROJECT_PATH>...]  
dotnet sln [<SOLUTION_FILE>] remove [-h|--help]
```

参数

- `SOLUTION_FILE`

要使用的解决方案文件。如果保留未指定，此命令会搜索当前目录以获取一个解决方案文件，如果找到多个解决方案文件，则该命令将失败。

- `PROJECT_PATH`

要添加到解决方案的一个或多个项目的路径。Unix/Linux shell [glob 模式](#) 扩展由 `dotnet sln` 命令正确处理。

选项

- `-h|--help`

打印出有关如何使用命令的说明。

示例

- 在解决方案中列出项目：

```
dotnet sln todo.sln list
```

- 将一个 C# 项目添加到解决方案中：

```
dotnet sln add todo-app/todo-app.csproj
```

- 从解决方案中删除一个 C# 项目：

```
dotnet sln remove todo-app/todo-app.csproj
```

- 将多个 C# 项目添加到解决方案的根目录中：

```
dotnet sln todo.sln add todo-app/todo-app.csproj back-end/back-end.csproj --in-root
```

- 将多个 C# 项目添加到解决方案中：

```
dotnet sln todo.sln add todo-app/todo-app.csproj back-end/back-end.csproj
```

- 从解决方案中删除多个 C# 项目：

```
dotnet sln todo.sln remove todo-app/todo-app.csproj back-end/back-end.csproj
```

- 使用 glob 模式(仅限 Unix/Linux)将多个 C# 项目添加到解决方案中：

```
dotnet sln todo.sln add **/*.csproj
```

- 使用 glob 模式(仅限 Unix/Linux)将多个 C# 项目从解决方案中删除：

```
dotnet sln todo.sln remove **/*.csproj
```

dotnet store

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.x SDK 及更高版本

名称

`dotnet store` - 将指定的程序集存储到[运行时包存储区](#)。

摘要

```
dotnet store -m|--manifest -f|--framework -r|--runtime [--framework-version] [-h|--help] [--output] [--skip-optimization] [--skip-symbols] [-v|--verbosity] [--working-dir]
```

说明

`dotnet store` 将指定的程序集存储到[运行时包存储区](#)。默认情况下，程序集更适用于目标运行时和框架。有关详细信息，请参阅[运行时包存储区](#)主题。

必需选项

- `-f|--framework <FRAMEWORK>`

指定[目标框架](#)。目标框架必须在项目文件中进行指定。

- `-m|--manifest <PATH_TO_MANIFEST_FILE>`

包存储区清单文件 是包含要存储的包列表的 XML 文件。清单文件的格式与 SDK 样式项目格式兼容。因此，引用所需的包的项目文件能够与 `-m|--manifest` 选项结合使用，以便于将程序集存储到运行时包存储区。若要指定多个清单文件，请为各个文件重复指定选项和路径。例如：

```
--manifest packages1.csproj --manifest packages2.csproj
```

- `-r|--runtime <RUNTIME_IDENTIFIER>`

目标[运行时标识符](#)。

可选选项

- `--framework-version <FRAMEWORK_VERSION>`

指定 .NET Core SDK 版本。使用此选项，可以选择特定的框架版本，不再局限于 `-f|--framework` 选项指定的框架。

- `-h|--help`

显示帮助信息。

- `-o|--output <OUTPUT_DIRECTORY>`

指定运行时包存储区的路径。如果未指定，默认路径为用户配置文件 .NET Core 安装目录的 store 子目录。

- `--skip-optimization`

跳过优化阶段。

- `--skip-symbols`

跳过符号生成。目前，只能在 Windows 和 Linux 上生成符号。

- `-v|--verbosity <LEVEL>`

设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。

- `-w|--working-dir <WORKING_DIRECTORY>`

此命令使用的工作目录。如果未指定，使用当前目录的 `obj` 子目录。

示例

- 存储 `packages.csproj` 项目文件中为 .NET Core 2.0.0 指定的包：

```
dotnet store --manifest packages.csproj --framework-version 2.0.0
```

- 存储 `packages.csproj` 中指定的包，但不进行优化：

```
dotnet store --manifest packages.csproj --skip-optimization
```

另请参阅

- [运行时包存储](#)

dotnet test

2020/3/30 • [Edit Online](#)

本文适用于: ✓ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet test` - 用于执行单元测试的 .NET 测试驱动程序。

摘要

```
dotnet test [<PROJECT> | <SOLUTION>]
  [-a|--test-adapter-path] [--blame] [-c|--configuration]
  [--collect] [-d|--diag] [-f|--framework] [--filter]
  [--interactive] [-l|--logger] [--no-build] [--nologo]
  [--no-restore] [-o|--output] [-r|--results-directory]
  [--runtime] [-s|--settings] [-t|--list-tests]
  [-v|--verbosity] [[--] <RunSettings arguments>]

dotnet test [-h|--help]
```

描述

`dotnet test` 命令用于执行给定项目中的单元测试。`dotnet test` 命令启动为项目指定的测试运行程序控制台应用程序。测试运行程序执行为单元测试框架(例如 MSTest、NUnit 或 xUnit)定义的测试，并报告每个测试是否成功。如果所有测试均成功，测试运行程序将返回 0 作为退出代码；否则将返回 1。测试运行程序和单元测试库打包为 NuGet 包并还原为该项目的普通依赖项。

测试项目使用普通 `<PackageReference>` 元素指定测试运行程序，如下方示例项目文件所示：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.5.0" />
    <PackageReference Include="xunit" Version="2.4.1" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.4.1" />
  </ItemGroup>

</Project>
```

自变量

● `PROJECT | SOLUTION`

测试项目或解决方案的路径。如未指定，则默认为当前目录。

选项

● `a|--test-adapter-path <PATH_TO_ADAPTER>`

在测试运行中使用来自指定路径的自定义测试适配器。

- `--blame`

在意见模式中运行测试。此选项有助于隔离导致测试主机出现故障的有问题的测试。它会在当前目录中创建一个输出文件 (Sequence.xml)，其中捕获了故障前的测试执行顺序。

- `c| --configuration <CONFIGURATION>`

定义生成配置。默认值为 `Debug`，但项目配置可以替代此默认 SDK 设置。

- `-collect <DATA_COLLECTOR_FRIENDLY_NAME>`

为测试运行启用数据收集器。有关详细信息，请参阅[监视和分析测试运行](#)。

- `d| --diag <PATH_TO_DIAGNOSTICS_FILE>`

启用测试平台的诊断模式，并将诊断消息写入指定文件。

- `f| --framework <FRAMEWORK>`

查找特定[框架](#)的测试二进制文件。

- `--filter <EXPRESSION>`

使用给定表达式筛选掉当前项目中的测试。有关详细信息，请参阅[筛选选项详细信息](#)部分。若要获取使用选择性单元测试筛选的其他信息和示例，请参阅[运行选择性单元测试](#)。

- `h| --help`

打印出有关命令的简短帮助。

- `--interactive`

允许命令停止并等待用户输入或操作。例如，完成身份验证。自 .NET Core 3.0 SDK 起可用。

- `l| --logger <LoggerUri/FriendlyName>`

指定测试结果记录器。

- `--no-build`

不在运行测试项目之前生成它。还将隐式设置 `--no-restore` 标记。

- `--nologo`

运行测试，而不显示 Microsoft TestPlatform 横幅。自 .NET Core 3.0 SDK 起可用。

- `--no-restore`

运行此命令时不执行隐式还原。

- `-o| --output <OUTPUT_DIRECTORY>`

查找要运行的二进制文件的目录。

- `-r| --results-directory <PATH>`

用于放置测试结果的目录。如果指定的目录不存在，则会创建该目录。

- `--runtime <RUNTIME_IDENTIFIER>`

要针对其测试的目标运行时。

- `-s| --settings <SETTINGS_FILE>`

`.runsettings` 文件用于运行测试。使用 `.runsettings` 文件配置单元测试。

- `-t|--list-tests`

列出当前项目中发现的所有测试。

- `-v|--verbosity <LEVEL>`

设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。

- `RunSettings` 参数

参数在测试中作为 `RunSettings` 配置传递。参数在“--”(注意 -- 后的空格)后被指定为 `[name]=[value]` 对。空格用于分隔多个 `[name]=[value]` 对。

示例: `dotnet test -- MSTest.DeploymentEnabled=false MSTest.MapInconclusiveToFailed=True`

有关详细信息, 请参阅 [vstest.console.exe: 传递 RunSettings 参数](#)。

示例

- 运行当前目录所含项目中的测试:

```
dotnet test
```

- 运行 `test1` 项目中的测试:

```
dotnet test ~/projects/test1/test1.csproj
```

- 在当前目录运行项目中的测试, 并以 `trx` 格式生成测试结果文件:

```
dotnet test --logger trx
```

筛选选项详细信息

`--filter <EXPRESSION>`

`<Expression>` 格式为 `<property><operator><value>[|&<Expression>]`。

`<property>` 是 `Test Case` 的特性。下面介绍了常用单元测试框架支持的属性:

MSTest	<ul style="list-style-type: none">• FullyQualifiedName• “属性”• ClassName• Priority• TestCategory
xUnit	<ul style="list-style-type: none">• FullyQualifiedName• DisplayName• Traits

`<operator>` 说明了属性和值之间的关系:

III	II
=	完全匹配
!=	非完全匹配
~	包含
!~	不包含

<value> 是字符串。所有查找都不区分大小写。

不含 <operator> 的表达式自动被视为 FullyQualifiedName 属性上的 contains (例如, dotnet test --filter xyz 与 dotnet test --filter FullyQualifiedName~xyz 相同)。

表达式可与条件运算符结合使用:

III	II
	或
&	AND

使用条件运算符时, 可以用括号将表达式括起来(例如, (Name~TestMethod1) | (Name~TestMethod2))。

若要获取使用选择性单元测试筛选的其他信息和示例, 请参阅[运行选择性单元测试](#)。

请参阅

- [框架和目标](#)
- [.NET Core 运行时标识符 \(RID\) 目录](#)

dotnet tool install

2020/3/19 • [Edit Online](#)

本文适用于：✓ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet tool install` - 在计算机上安装指定的 .NET Core 工具。

摘要

```
dotnet tool install <PACKAGE_NAME> <-g|--global>
  [--add-source] [--configfile] [--framework]
  [-v|--verbosity] [--version]

dotnet tool install <PACKAGE_NAME> <--tool-path>
  [--add-source] [--configfile] [--framework]
  [-v|--verbosity] [--version]

dotnet tool install <PACKAGE_NAME>
  [--add-source] [--configfile] [--framework]
  [-v|--verbosity] [--version]

dotnet tool install <-h|--help>
```

描述

`dotnet tool install` 命令为用户提供一种在计算机上安装 .NET Core 工具的方法。若要使用命令，请指定以下安装选项之一：

- 若要在默认位置中安装全局工具，请使用 `--global` 选项。
- 若要在自定义位置中安装全局工具，请使用 `--tool-path` 选项。
- 若要安装本地工具，请省略 `--global` 和 `--tool-path` 选项。

本地工具从 .NET Core SDK 3.0 开始可用。

指定 `-g` 或 `--global` 选项时，全局工具默认安装在以下目录中：

(OS)	IT
Linux/macOS	<code>\$HOME/.dotnet/tools</code>
Windows	<code>%USERPROFILE%\dotnet\tools</code>

本地工具将添加到当前目录下的 `.config` 目录中的 `tool-manifest.json` 文件中。如果清单文件尚不存在，请通过运行以下命令来创建它：

```
dotnet new tool-manifest
```

有关详细信息，请参阅[安装本地工具](#)。

自变量

- `PACKAGE_NAME`

包含要安装的 .NET Core 工具的 NuGet 包的名称/ID。

选项

- `add-source <SOURCE>`

添加安装过程中要使用的其他 NuGet 包源。

- `configfile <FILE>`

要使用的 NuGet 配置 (nuget.config) 文件。

- `framework <FRAMEWORK>`

指定要安装工具的[目标框架](#)。默认情况下，.NET Core SDK 尝试选择最合适的目标框架。

- `-g|--global`

指定安装是用户范围的。不能与 `--tool-path` 选项一起使用。省略 `--global` 和 `--tool-path` 指定本地工具安装。

- `-h|--help`

打印出有关命令的简短帮助。

- `tool-path <PATH>`

指定全局工具的安装位置。路径可以是绝对的，也可以是相对的。如果路径不存在，命令会尝试创建它。省略 `--global` 和 `--tool-path` 指定本地工具安装。

- `-v|--verbosity <LEVEL>`

设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`

。

- `--version <VERSION_NUMBER>`

要安装的工具版本。默认情况下，安装最新的稳定包版本。使用此选项安装工具的预览版或较旧版本。

示例

- `dotnet tool install -g dotnetsay`

在默认位置中安装 `dotnetsay` 全局工具。

- `dotnet tool install dotnetsay --tool-path c:\global-tools`

在特定 Windows 目录中安装 `dotnetsay` 全局工具。

- `dotnet tool install dotnetsay --tool-path ~/bin`

在特定 Linux/macOS 目录中安装 `dotnetsay` 全局工具。

- `dotnet tool install -g dotnetsay --version 2.0.0`

安装 2.0.0 版的 `dotnetsay` 全局工具。

- `dotnet tool install dotnetsay`

在当前目录中安装 [dotnetsay](#) 本地工具。

请参阅

- [.NET Core 工具](#)
- [教程:使用 .NET Core CLI 安装和使用 .NET Core 全局工具](#)
- [教程:使用 .NET Core CLI 安装和使用 .NET Core 本地工具](#)

dotnet tool list

2020/3/19 • [Edit Online](#)

本文适用于：✓ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet tool list` - 列出计算机上当前安装的所有指定类型的.NET Core 工具。

摘要

```
dotnet tool list <-g|--global>  
  
dotnet tool list <--tool-path>  
  
dotnet tool list  
  
dotnet tool list <-h|--help>
```

描述

`dotnet tool list` 命令为用户提供一种在计算机上安装所有.NET Core 全局工具、工具路径工具或本地工具的方法。此命令列出包名称、安装的版本以及工具命令。若要使用命令，请指定以下项之一：

- 在默认位置安装全局工具。使用 `--global` 选项
- 在自定义位置安装全局工具。使用 `--tool-path` 选项。
- 运行本地工具。省略 `--global` 和 `--tool-path` 选项。

本地工具从.NET Core SDK 3.0 开始可用。

选项

- `-g|--global`

列出用户范围的全局工具。不能与 `--tool-path` 选项一起使用。省略 `--global` 和 `--tool-path` 将列出本地工具。

- `-h|--help`

打印出有关命令的简短帮助。

- `--tool-path <PATH>`

指定用于查找全局工具的自定义位置。路径可以是绝对的，也可以是相对的。不能与 `--global` 选项一起使用。省略 `--global` 和 `--tool-path` 将列出本地工具。

示例

- `dotnet tool list -g`

列出计算机上安装的所有用户范围的全局工具(当前用户配置文件)。

- `dotnet tool list --tool-path c:\global-tools`

列出特定 Windows 目录中的全局工具。

- `dotnet tool list --tool-path ~/bin`

列出特定 Linux/macOS 目录中的全局工具。

- `dotnet tool list`

列出当前目录中所有可用的本地工具。

请参阅

- [.NET Core 工具](#)
- [教程:使用 .NET Core CLI 安装和使用 .NET Core 全局工具](#)
- [教程:使用 .NET Core CLI 安装和使用 .NET Core 本地工具](#)

dotnet tool restore

2020/3/19 • [Edit Online](#)

本文适用于: ✓ .NET Core 3.0 SDK 及更高版本

“属性”

`dotnet tool restore` - 在计算机上安装当前目录范围内的 .NET Core 本地工具。

摘要

```
dotnet tool restore <PACKAGE_NAME>
  [--configfile] [--add-source] [tool-manifest]
  [--disable-parallel] [--ignore-failed-sources]
  [--no-cache] [-interactive] [-v|--verbosity]

dotnet tool restore <-h|--help>
```

描述

`dotnet tool restore` 命令查找当前目录范围内的工具清单文件，并安装其中列出的工具。有关清单文件的信息，请参阅[安装本地工具](#)和[调用本地工具](#)。

自变量

- `<PACKAGE_NAME>`

包含要安装的 .NET Core 工具的 NuGet 包的名称/ID。

选项

- `--configfile <FILE>`

要使用的 NuGet 配置 (`nuget.config`) 文件。

- `--add-source <SOURCE>`

添加安装过程中要使用的其他 NuGet 包源。

- `--tool-manifest <PATH>`

清单文件的路径。

- `--disable-parallel`

防止并行还原多个项目。

- `--ignore-failed-sources`

将包源失败视为警告。

- `--no-cache`

不要缓存包和 HTTP 请求。

- `--interactive`

允许命令停止并等待用户输入或操作(例如, 完成身份验证)。

- `-h | --help`

打印出有关命令的简短帮助。

- `-v | --verbosity <LEVEL>`

设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。

示例

- `dotnet tool restore`

还原当前目录的本地工具。

请参阅

- [.NET Core 工具](#)
- [教程:使用 .NET Core CLI 安装和使用 .NET Core 本地工具](#)

dotnet tool run

2020/3/19 • [Edit Online](#)

本文适用于: ✓ .NET Core 3.0 SDK 及更高版本

“属性”

`dotnet tool run` - 调用本地工具。

摘要

```
dotnet tool run <COMMAND NAME>
```

```
dotnet tool run <-h|--help>
```

描述

`dotnet tool run` 命令将搜索当前目录范围内的工具清单文件。当找到对指定工具的引用时，它会运行该工具。有关详细信息，请参阅 [调用本地工具](#)。

自变量

- `COMMAND_NAME`

要运行的工具的命令名称。

选项

- `-h|--help`

打印出有关命令的简短帮助。

示例

- `dotnet tool run dotnetsay`

运行 `dotnetsay` 本地工具。

请参阅

- [.NET Core 工具](#)
- [教程:使用 .NET Core CLI 安装和使用 .NET Core 本地工具](#)

dotnet tool uninstall

2020/3/19 • [Edit Online](#)

本文适用于: ✓ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet tool uninstall` - 从计算机上卸载指定的 .NET Core 工具。

摘要

```
dotnet tool uninstall <PACKAGE_NAME> <-g|--global>  
dotnet tool uninstall <PACKAGE_NAME> <--tool-path>  
dotnet tool uninstall <PACKAGE_NAME>  
dotnet tool uninstall <-h|--help>
```

描述

`dotnet tool uninstall` 提供一种从计算机上卸载 .NET Core 工具的方法。若要使用命令，请指定以下选项之一：

- 若要卸载安装在默认位置的全局工具，请使用 `--global` 选项。
- 若要卸载安装在自定义位置的全局工具，请使用 `--tool-path` 选项。
- 若要卸载本地工具，请省略 `--global` 和 `--tool-path` 选项。

本地工具从 .NET Core SDK 3.0 开始可用。

自变量

- `PACKAGE_NAME`

包含要卸载的 .NET Core 工具的 NuGet 包的名称/ID。你可以使用 `dotnet tool list` 命令查找包名称。

选项

- `-g|--global`

指定要从用户范围的安装中删除工具。不能与 `--tool-path` 选项一起使用。省略 `--global` 和 `--tool-path` 指定要删除的工具是本地工具。

- `-h|--help`

打印出有关命令的简短帮助。

- `--tool-path <PATH>`

指定卸载工具的位置。路径可以是绝对的，也可以是相对的。不能与 `--global` 选项一起使用。省略 `--global` 和 `--tool-path` 指定要删除的工具是本地工具。

示例

- `dotnet tool uninstall -g dotnetsay`

卸载 `dotnetsay` 全局工具。

- `dotnet tool uninstall dotnetsay --tool-path c:\global-tools`

从特定 Windows 目录卸载 `dotnetsay` 全局工具。

- `dotnet tool uninstall dotnetsay --tool-path ~/bin`

从特定 Linux/macOS 目录卸载 `dotnetsay` 全局工具。

- `dotnet tool uninstall dotnetsay`

从当前目录卸载 `dotnetsay` 全局工具。

请参阅

- [.NET Core 工具](#)
- [教程: 使用 .NET Core CLI 安装和使用 .NET Core 全局工具](#)
- [教程: 使用 .NET Core CLI 安装和使用 .NET Core 本地工具](#)

dotnet tool update

2020/3/19 • [Edit Online](#)

本文适用于：✓ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet tool update` - 更新计算机上指定的 .NET Core 工具。

摘要

```
dotnet tool update <PACKAGE_NAME> <-g|--global>
  [--configfile] [--framework] [-v|--verbosity]
  [--add-source]

dotnet tool update <PACKAGE_NAME> <--tool-path>
  [--configfile] [--framework] [-v|--verbosity]
  [--add-source]

dotnet tool update <PACKAGE_NAME>
  [--configfile] [--framework] [-v|--verbosity]
  [--add-source]

dotnet tool update <-h|--help>
```

描述

`dotnet tool update` 命令让你可以将计算机上的 .NET Core 工具更新为包的最新稳定版。此命令卸载并重新安装工具，有效地对工具进行更新。若要使用命令，请指定以下选项之一：

- 若要更新安装在默认位置的全局工具，请使用 `--global` 选项
- 若要更新安装在自定义位置的全局工具，请使用 `--tool-path` 选项。
- 若要更新本地工具，请省略 `--global` 和 `--tool-path` 选项。

本地工具从 .NET Core SDK 3.0 开始可用。

自变量

- `<PACKAGE_NAME>`

包含要更新的 .NET Core 全局工具的 NuGet 包的名称/ID。你可以使用 `dotnet tool list` 命令查找包名称。

选项

- `--add-source <SOURCE>`

添加安装过程中要使用的其他 NuGet 包源。

- `--configfile <FILE>`

要使用的 NuGet 配置 (nuget.config) 文件。

- `--framework <FRAMEWORK>`

指定要更新工具的[目标框架](#)。

- `-g|--global`

指定此更新用于用户范围的工具。不能与 `--tool-path` 选项一起使用。省略 `--global` 和 `--tool-path` 均指定要更新的工具是本地工具。

- `-h|--help`

打印出有关命令的简短帮助。

- `--tool-path <PATH>`

指定全局工具的安装位置。路径可以是绝对的，也可以是相对的。不能与 `--global` 选项一起使用。省略 `--global` 和 `--tool-path` 均指定要更新的工具是本地工具。

- `-v|--verbosity <LEVEL>`

设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。

示例

- `dotnet tool update -g dotnetsay`

更新 `dotnetsay` 全局工具。

- `dotnet tool update dotnetsay --tool-path c:\global-tools`

更新位于特定 Windows 目录的 `dotnetsay` 全局工具。

- `dotnet tool update dotnetsay --tool-path ~/bin`

更新位于特定 Linux/macOS 目录的 `dotnetsay` 全局工具。

- `dotnet tool update dotnetsay`

更新为当前目录安装的 `dotnetsay` 本地工具。

请参阅

- [.NET Core 工具](#)
- [教程: 使用 .NET Core CLI 安装和使用 .NET Core 全局工具](#)
- [教程: 使用 .NET Core CLI 安装和使用 .NET Core 本地工具](#)

dotnet vstest

2020/3/18 • • [Edit Online](#)

本文适用于：✓ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet-vstest` - 从指定文件运行测试。

摘要

```
dotnet vstest [<TEST_FILE_NAMES>] [--Settings] [--Tests]
    [--TestAdapterPath] [--Platform] [--Framework] [--Parallel]
    [--TestCaseFilter] [--logger] [-lt|--ListTests]
    [--ParentProcessId] [--Port] [--Diag] [--Blame]
    [--InIsolation] [[--] <args>...]] [-?|--Help]
```

描述

`dotnet-vstest` 命令运行 `vstest.console` 命令行应用程序以运行自动化单元测试。

自变量

- `TEST_FILE_NAMES`

从指定的程序集运行测试。用空格分隔多个测试程序集名称。支持通配符。

选项

- `--Settings <Settings File>`

运行测试时要使用的设置。

- `--Tests <Test Names>`

运行具有与提供的值匹配的名称的测试。用逗号分隔多个值。

- `--TestAdapterPath`

在测试运行中使用来自给定路径(如果有)的自定义测试适配器。

- `--Platform <Platform type>`

用于执行测试的目标平台体系结构。有效值为 `x86`、`x64` 和 `ARM`。

- `--Framework <Framework Version>`

用于测试执行的目标 .NET Framework 版本。有效值的示例为 `.NETFramework,Version=v4.6` 或 `.NETCoreApp,Version=v1.0`。其他支持的值为 `Framework40`、`Framework45`、`FrameworkCore10` 和 `FrameworkUap10`。

- `--Parallel`

并行运行测试。默认情况下，计算机上的所有可用内核都可供使用。通过在 `runsettings` 文件的

`RunConfiguration` 节点下设置 `MaxCpuCount` 属性来指定显式内核数。

- `--TestCaseFilter <Expression>`

运行与给定表达式匹配的测试。`<Expression>` 的格式为 `<property>Operator<value>[|&<Expression>]`，其中运算符是 `=`、`!=` 或 `~` 之一。运算符 `~` 具有“包含”语义，并适用于字符串属性，如 `DisplayName`。括号 `()` 用于组的子表达式。

- `-?|--Help`

打印出有关命令的简短帮助。

- `--logger <Logger Uri/FriendlyName>`

为测试结果指定一个记录器。

- 若要将测试结果发布到 Team Foundation Server，请使用 `TfsPublisher` 记录器提供程序：

```
/logger:TfsPublisher;
Collection=<team project collection url>;
BuildName=<build name>;
TeamProject=<team project name>
[;Platform=<Defaults to "Any CPU">]
[;Flavor=<Defaults to "Debug">]
[;RunTitle=<title>]
```

- 若要将结果记录到 Visual Studio 测试结果文件 (TRX)，请使用 `trx` 记录器提供程序。此开关使用给定的日志文件名在测试结果目录中创建一个文件。如果未提供 `LogFileName`，将创建唯一的文件名以保留测试结果。

```
/logger:trx [;LogFileName=<Defaults to unique file name>]
```

- `-lt|--ListTests <File Name>`

列出给定测试容器中所有已发现的测试。

- `--ParentProcessId <ParentProcessId>`

父进程的进程 ID 负责启动当前进程。

- `--Port <Port>`

指定套接字连接和接收事件消息的端口。

- `--Diag <Path to log file>`

为测试平台启用详细日志。日志被写入到所提供的文件。

- `--Blame`

在意见模式中运行测试。此选项有助于隔离导致测试主机出现故障的有问题的测试。它会在当前目录中创建一个输出文件 (`Sequence.xml`)，其中捕获了故障前的测试执行顺序。

- `--InIsolation`

在隔离的进程中运行测试。虽然这使得 `vstest.console.exe` 进程不太可能在测试出错时停止，但测试的运行速度会较慢。

- `@<file>`

有关更多选项，请阅读响应文件。

- `args`

指定要传递到适配器的额外参数。参数被指定为 `<n>=<v>` 格式的名称值对，其中 `<n>` 是参数名称，`<v>` 是参数值。使用空格分隔多个参数。

示例

在 mytestproject.dll 中运行测试：

```
dotnet vstest mytestproject.dll
```

在 mytestproject.dll 中运行测试，并使用自定义名称导出到自定义文件夹：

```
dotnet vstest mytestproject.dll --logger:"trx;LogFileName=custom_file_name.trx" --ResultsDirectory:custom/file/path
```

在 mytestproject.dll 和 myothertestproject.exe 中运行测试：

```
dotnet vstest mytestproject.dll myothertestproject.exe
```

运行 `TestMethod1` 测试：

```
dotnet vstest /Tests:TestMethod1
```

运行 `TestMethod1` 和 `TestMethod2` 测试：

```
dotnet vstest /Tests:TestMethod1,TestMethod2
```

dotnet-install 脚本引用

2020/3/18 • [Edit Online](#)

名称

`dotnet-install.ps1` | `dotnet-install.sh` - 用于安装 .NET Core SDK 和共享运行时的脚本。

摘要

Windows:

```
dotnet-install.ps1 [-Channel] [-Version] [-JsonFile] [-InstallDir] [-Architecture]
    [-Runtime] [-DryRun] [-NoPath] [-Verbose] [-AzureFeed] [-UncachedFeed] [-NoCdn] [-FeedCredential]
    [-ProxyAddress] [-ProxyUseDefaultCredentials] [-SkipNonVersionedFiles] [-Help]
```

Linux/macOS:

```
dotnet-install.sh [--channel] [--version] [--jsonfile] [--install-dir] [--architecture]
    [--runtime] [--dry-run] [--no-path] [--verbose] [--azure-feed] [--uncached-feed] [--no-cdn] [--feed-
    credential]
    [--runtime-id] [--skip-non-versioned-files] [--help]
```

说明

`dotnet-install` 脚本用于执行 .NET Core SDK 的非管理员安装，其中包含 .NET Core CLI 和共享运行时。

建议使用脚本的稳定版本：

- Bash (Linux/macOS): <https://dot.net/v1/dotnet-install.sh>
- PowerShell (Windows): <https://dot.net/v1/dotnet-install.ps1>

这些脚本主要用于自动化方案和非管理员安装。有两个脚本：一个是适用于在 Windows 上工作的 PowerShell 脚本，另一个是在 Linux/macOS 上工作的 bash 脚本。这两个脚本的行为相同。bash 脚本也读取 PowerShell 开关。因此，可以在 Linux/macOS 系统上将 PowerShell 开关与脚本结合使用。

安装脚本从 CLI 生成放置下载 ZIP/tarball 文件，并将其安装在默认位置或 `-InstallDir|--install-dir` 所指定的位置。默认情况下，安装脚本下载 SDK 并安装它。如果只想获取共享的运行时，请指定 `-Runtime|--runtime` 参数。

默认情况下，该脚本会将安装位置添加到当前会话的 \$PATH。通过指定 `-NoPath|--no-path` 参数覆盖此默认行为。

运行脚本前，请安装所需的[依赖项](#)。

可以使用 `-Version|--version` 参数安装特定版本。必须将版本指定为由 3 部分构成的版本（例如 `2.1.0`）。如果未提供，则使用 `latest` 版本。

选项

- `-Channel|--channel <CHANNEL>`

指定安装的源通道。可能的值为：

- `Current` - 最新版本。
- `LTS` - 长期支持频道(最新受支持版本)。
- 表示特定版本的由两部分构成的 X.Y 格式的版本(例如 `2.1` 或 `3.0`)。
- 分支名称:例如 `release/3.1.1xx` 或 `master`(适用于每日测试版本)。使用此选项可以从预览通道安装版本。使用[安装程序和二进制文件](#)中列出的通道名称。

默认值为 `LTS`。有关 .NET 支持频道的详细信息,请参阅[.NET 支持策略](#)页。

- `-Version|--version <VERSION>`

表示特定的内部版本。可能的值为:

- `latest` - 频道上的最新内部版本(与 `-Channel` 选项结合使用)。
- `coherent` - 频道上的最新相干内部版本;使用最新的稳定包组合(与分支名称 `-Channel` 选项结合使用)。
- 由三部分组成的版本,采用 X.Y.Z 格式,表示特定的内部版本;取代 `-Channel` 选项。例如:
`2.0.0-preview2-006120`。

如果没有指定, `-Version` 默认值为 `latest`。

- `-JsonFile|--jsonfile <JSONFILE>`

指定将用于确定 SDK 版本的 `global.json` 文件的路径。`global.json` 文件必须具有 `sdk:version` 的值。

- `-InstallDir|--install-dir <DIRECTORY>`

指定安装路径。如果不存在,则会创建该目录。默认值为 `%LocalAppData%\Microsoft\dotnet`。会将二进制文件直接放入目录中。

- `-Architecture|--architecture <ARCHITECTURE>`

要安装的 .NET Core 二进制文件的体系结构。可能值为 `<auto>`、`amd64`、`x64`、`x86`、`arm64` 以及 `arm`。默认值为 `<auto>`, 它表示当前正在运行的操作系统体系结构。

- `-SharedRuntime|--shared-runtime`

NOTE

此参数已过时,可能会在将来版本的脚本中删除。建议的替代项为 `-Runtime|--runtime` 选项。

仅安装共享运行时位,而非整个 SDK。此选项等效于指定 `-Runtime|--runtime dotnet`。

- `-Runtime|--runtime <RUNTIME>`

仅安装共享运行时,而非整个 SDK。可能的值为:

- `dotnet` - `Microsoft.NETCore.App` 共享运行时。
- `aspnetcore` - `Microsoft.AspNetCore.App` 共享运行时。
- `windowsdesktop` - `Microsoft.WindowsDesktop.App` 共享运行时。

- `-DryRun|--dry-run`

如果设置,脚本将不会执行安装。而会显示要使用哪个命令行来持续安装当前请求的 .NET Core CLI 版本。例如,如果指定版本 `latest`,它将显示特定版本的链接,以便可在生成脚本中明确地使用此命令。如果想要自行安装或下载,它还会显示二进制文件位置。

- `-NoPath|--no-path`

如果设定,不会将安装文件夹导出到当前会话的路径。默认情况下,该脚本会修改 PATH,这会使 .NET

Core CLI 在安装后立即可用。

- `-Verbose| --verbose`

显示诊断信息。

- `-AzureFeed| --azure-feed`

指定此安装程序的 Azure 源的 URL。建议不要更改该值。默认值为

`https://dotnetcli.azureedge.net/dotnet`。

- `-UncachedFeed| --uncached-feed`

允许更改此安装程序使用的未缓存源的 URL。建议不要更改该值。

- `-NoCdn| --no-cdn`

禁止从 [Azure 内容分发网络 \(CDN\)](#) 进行下载，并直接使用未缓存源。

- `-FeedCredential| --feed-credential`

用作追加到 Azure 源的查询字符串。这允许更改 URL 以使用非公共 blob 存储帐户。

- `--runtime-id`

指定要为其安装工具的[运行时标识符](#)。使用适用于可移植 Linux 的 `linux-x64`。（仅适用于 Linux/macOS）

- `-ProxyAddress`

如果设置，安装程序发出 Web 请求时将使用该代理。（仅对 Windows 有效）

- `ProxyUseDefaultCredentials`

如果设置，在使用代理地址时，安装程序会使用当前用户的凭据。（仅对 Windows 有效）

- `-SkipNonVersionedFiles| --skip-non-versioned-files`

跳过安装未添加版本的文件，例如 `dotnet.exe`（如果它们已经存在）。

- `-Help| --help`

打印脚本帮助。

示例

- 将最新的长期支持 (LTS) 版本安装到默认位置：

Windows:

```
./dotnet-install.ps1 -Channel LTS
```

macOS/Linux:

```
./dotnet-install.sh --channel LTS
```

- 将 3.1 通道中的最新版本安装到指定位置：

Windows:

```
./dotnet-install.ps1 -Channel 3.1 -InstallDir C:\cli
```

macOS/Linux:

```
./dotnet-install.sh --channel 3.1 --install-dir ~/cli
```

- 安装 3.0.0 版共享运行时：

Windows:

```
./dotnet-install.ps1 -Runtime dotnet -Version 3.0.0
```

macOS/Linux:

```
./dotnet-install.sh --runtime dotnet --version 3.0.0
```

- 获取脚本并在公司代理后面安装 2.1.2 版本(仅限 Windows)：

```
Invoke-WebRequest 'https://dot.net/v1/dotnet-install.ps1' -Proxy $env:HTTP_PROXY -  
ProxyUseDefaultCredentials -OutFile 'dotnet-install.ps1';  
. ./dotnet-install.ps1 -InstallDir '~/.dotnet' -Version '2.1.2' -ProxyAddress $env:HTTP_PROXY -  
ProxyUseDefaultCredentials;
```

- 获取脚本并安装 .NET Core CLI 单行式命令示例：

Windows:

```
# Run a separate PowerShell process because the script calls exit, so it will end the current  
PowerShell session.  
&powershell -NoProfile -ExecutionPolicy unrestricted -Command "  
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12; &  
([scriptblock]::Create((Invoke-WebRequest -UseBasicParsing 'https://dot.net/v1/dotnet-  
install.ps1')))) <additional install-script args>"
```

macOS/Linux:

```
curl -sSL https://dot.net/v1/dotnet-install.sh | bash /dev/stdin <additional install-script args>
```

另请参阅

- [.NET Core 版本](#)
- [.NET Core 运行时和 SDK 下载存档](#)

dotnet add reference

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.x SDK 及更高版本

名称

`dotnet add reference` - 添加项目到项目 (P2P) 引用。

摘要

`dotnet add [<PROJECT>] reference [-f|--framework] <PROJECT_REFERENCES> [-h|--help] [--interactive]`

说明

使用 `dotnet add reference` 命令可方便地向项目添加项目引用。运行该命令后，会将 `<ProjectReference>` 元素添加到项目文件。

```
<ItemGroup>
  <ProjectReference Include="app.csproj" />
  <ProjectReference Include=".\\lib2\\lib2.csproj" />
  <ProjectReference Include=".\\lib1\\lib1.csproj" />
</ItemGroup>
```

参数

- `PROJECT`

指定项目文件。如果未指定，此命令会搜索当前目录来获取一个项目文件。

- `PROJECT_REFERENCES`

要添加的项目到项目 (P2P) 引用。指定一个或多个项目。基于 Unix/Linux 的系统支持 [glob 模式](#)。

选项

- `-h|--help`

打印出有关命令的简短帮助。

- `-f|--framework <FRAMEWORK>`

仅在以特定框架为目标时添加项目引用。

- `--interactive`

允许命令停止并等待用户输入或操作(例如，完成身份验证)。自 .NET Core 3.0 SDK 起可用。

示例

- 添加项目引用：

```
dotnet add app/app.csproj reference lib/lib.csproj
```

- 向当前目录中的项目添加多个项目引用:

```
dotnet add reference lib1/lib1.csproj lib2/lib2.csproj
```

- 使用 glob 模式在 Linux/Unix 上添加多个项目引用:

```
dotnet add app/app.csproj reference **/*.csproj
```

dotnet list reference

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.x SDK 及更高版本

名称

`dotnet list reference` - 列出项目到项目引用。

摘要

`dotnet list [<PROJECT>|<SOLUTION>] reference [-h|--help]`

说明

使用 `dotnet list reference` 命令可方便地列出给定项目或解决方案的项目引用。

参数

- `PROJECT | SOLUTION`

指定用于列出引用的项目或解决方案文件。如果未指定，此命令会搜索当前目录，以获取项目文件。

选项

- `-h|--help`

打印出有关命令的简短帮助。

示例

- 列出指定项目的项目引用：

```
dotnet list app/app.csproj reference
```

- 列出当前目录中的项目的项目引用：

```
dotnet list reference
```

dotnet remove reference

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.x SDK 及更高版本

名称

`dotnet remove reference` - 删除“项目到项目”引用。

摘要

```
dotnet remove [<PROJECT>] reference [-f|--framework] <PROJECT_REFERENCES> [-h|--help]
```

说明

使用 `dotnet remove reference` 命令可方便地从项目删除项目引用。

参数

`PROJECT`

目标项目文件。如果未指定，此命令会搜索当前目录来获取一个项目文件。

`PROJECT_REFERENCES`

要删除的项目到项目 (P2P) 引用。可指定一个或多个项目。基于 Unix/Linux 的终端支持 [Glob 模式](#)。

选项

- `-h|--help`

打印出有关命令的简短帮助。

- `-f|--framework <FRAMEWORK>`

仅在以特定 [框架](#) 为目标时删除引用。

示例

- 从指定项目删除项目引用：

```
dotnet remove app/app.csproj reference lib/lib.csproj
```

- 从当前目录中的项目删除多个项目引用：

```
dotnet remove reference lib1/lib1.csproj lib2/lib2.csproj
```

- 使用 Unix/Linux 的 glob 模式删除多个项目引用：

```
dotnet remove app/app.csproj reference **/*.csproj`
```

dotnet add package

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.x SDK 及更高版本

名称

`dotnet add package` - 向项目文件添加包引用。

摘要

```
dotnet add [<PROJECT>] package <PACKAGE_NAME> [-h|--help] [-f|--framework] [--interactive] [-n|--no-restore] [--package-directory] [-s|--source] [-v|--version]
```

说明

使用 `dotnet add package` 命令可方便地向项目文件添加包引用。运行该命令后，还有一个兼容性检查，确保包与项目中的框架兼容。如果通过了该检查，则将 `<PackageReference>` 元素添加到项目文件并运行 `dotnet` 还原。

NOTE

从 .NET Core 2.0 SDK 开始，无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build` 和 `dotnet run`。在执行显式还原有意义的某些情况下，例如 [Azure DevOps Services 中的持续集成生成](#) 中，或在需要显式控制还原发生时间的生成系统中，它仍然是有效的命令。

例如，将 `Newtonsoft.Json` 添加到 `ToDo.csproj` 后的输出如以下示例所示：

```
Writing C:\Users\me\AppData\Local\Temp\tmp95A8.tmp
info : Adding PackageReference for package 'Newtonsoft.Json' into project 'C:\projects\ToDo\ToDo.csproj'.
log  : Restoring packages for C:\Temp\projects\consoleproj\consoleproj.csproj...
info :   GET https://api.nuget.org/v3-flatcontainer/newtonsoft.json/index.json
info :     OK https://api.nuget.org/v3-flatcontainer/newtonsoft.json/index.json 79ms
info :   GET https://api.nuget.org/v3-flatcontainer/newtonsoft.json/12.0.1/newtonsoft.json.12.0.1.nupkg
info :     OK https://api.nuget.org/v3-flatcontainer/newtonsoft.json/12.0.1/newtonsoft.json.12.0.1.nupkg 232ms
log  : Installing Newtonsoft.Json 12.0.1.
info : Package 'Newtonsoft.Json' is compatible with all the specified frameworks in project
'C:\projects\ToDo\ToDo.csproj'.
info : PackageReference for package 'Newtonsoft.Json' version '12.0.1' added to file
'C:\projects\ToDo\ToDo.csproj'.
```

`ToDo.csproj` 文件现包含用于引用的包的 `<PackageReference>` 元素。

```
<PackageReference Include="Newtonsoft.Json" Version="12.0.1" />
```

参数

- `PROJECT`

指定项目文件。如果未指定，此命令会搜索当前目录来获取一个项目文件。

- `PACKAGE_NAME`

要添加的包引用。

选项

- `-f|--framework <FRAMEWORK>`

仅在以特定[框架](#)为目标时添加包引用。

- `-h|--help`

打印出有关命令的简短帮助。

- `--interactive`

允许命令停止并等待用户输入或操作(例如, 完成身份验证)。从 .NET Core 2.1 SDK, 版本 2.1.400 或更高版本开始可用。

- `-n|--no-restore`

在不执行还原预览和兼容性检查的情况下添加包引用。

- `--package-directory <PACKAGE_DIRECTORY>`

要在其中还原包的目录。Windows 上的默认包还原位置为 `%userprofile%\.nuget\packages`, macOS 和 Linux 上的默认包还原位置为 `~/.nuget/packages`。有关详细信息, 请参阅[在 NuGet 中管理全局包、缓存和临时文件夹](#)。

- `-s|--source <SOURCE>`

要在还原操作期间使用的 NuGet 包源。

- `-v|--version <VERSION>`

包的版本。请参阅[NuGet 包版本控制](#)。

示例

- 将 `Newtonsoft.Json` NuGet 包添加到项目：

```
dotnet add package Newtonsoft.Json
```

- 向项目添加特定版本的包：

```
dotnet add ToDo.csproj package Microsoft.Azure.DocumentDB.Core -v 1.0.0
```

- 使用特定的 NuGet 源添加包：

```
dotnet add package Microsoft.AspNetCore.StaticFiles -s https://dotnet.myget.org/F/dotnet-core/api/v3/index.json
```

另请参阅

- [在 NuGet 中管理全局包、缓存和临时文件夹](#)
- [NuGet 包版本控制](#)

dotnet list package

2020/3/18 • [Edit Online](#)

本文适用于: ✓ .NET Core 2.2 SDK 及更高版本

名称

`dotnet list package` - 列出项目或解决方案的包引用。

摘要

```
dotnet list [<PROJECT>|<SOLUTION>] package [--config] [--framework] [--highest-minor] [--highest-patch]
    [--include-prerelease] [--include-transitive] [--interactive] [--outdated] [--source]
dotnet list package [-h|--help]
```

说明

使用 `dotnet list package` 命令，可以方便地列出特定项目或解决方案的所有 NuGet 包引用。首先，需要生成项目，以提供必需资产以供此命令处理。下面的示例展示了 `dotnet list package SentimentAnalysis` 项目的命令输出：

```
Project 'SentimentAnalysis' has the following package references
[netcoreapp2.1]:
Top-level Package      Requested   Resolved
> Microsoft.ML          1.4.0       1.4.0
> Microsoft.NETCore.App (A) [2.1.0, )  2.1.0

(A) : Auto-referenced package.
```

“已请求”列是指项目文件中指定的包版本，可以是一个范围。“已解析”列列出了项目当前使用的版本，始终都是一个值。紧靠名称旁边显示 (A) 的包表示从项目设置(类型、或 Sdk 属性等)推断出的 <TargetFramework> 隐式包引用 <TargetFrameworks>。

使用 `--outdated` 选项，可以确定项目中正在使用的包是否有更高版本。默认情况下，`--outdated` 列出最新稳定包，除非已解析版本也是预发行版本。若要在列出更高版本时包含预发行版本，还请指定 `--include-prerelease` 选项。下面的示例展示了上一个示例中相同项目的 `dotnet list package --outdated --include-prerelease` 命令输出：

```
The following sources were used:
https://api.nuget.org/v3/index.json
C:\Program Files (x86)\Microsoft SDKs\NuGetPackages\

Project `SentimentAnalysis` has the following updates to its packages
[netcoreapp2.1]:
Top-level Package      Requested   Resolved   Latest
> Microsoft.ML          1.4.0       1.4.0     1.5.0-preview
```

如果需要确定项目是否有可传递依赖关系，请使用 `--include-transitive` 选项。如果在项目中添加包，它转而又依赖另一个包，就会出现可传递依赖关系。下面的示例展示了 `dotnet list package --include-transitive HelloPlugin` 项目的命令运行输出，其中显示顶级包及其依赖的包：

```
Project 'HelloPlugin' has the following package references
[netcoreapp3.0]:
Transitive Package      Resolved
> PluginBase           1.0.0
```

参数

PROJECT | SOLUTION

要对其运行命令的项目或解决方案文件。如果未指定，此命令会搜索当前目录来获取一个项目文件。如果找到多个解决方案或项目，便会抛出错误。

选项

- `--config <SOURCE>`

在搜索版本更高的包时，要使用的 NuGet 源。需要使用 `--outdated` 选项。

- `--framework <FRAMEWORK>`

只显示适用于指定 [目标框架](#) 的包。若要指定多个框架，请多次重复此选项。例如：

```
--framework netcoreapp2.2 --framework netstandard2.0
```

- `-h | --help`

打印出有关命令的简短帮助。

- `--highest-minor`

在搜索版本更高的包时，仅考虑有匹配的主版本号的包。需要使用 `--outdated` 选项。

- `--highest-patch`

在搜索版本更高的包时，仅考虑有匹配的主版本号和次要版本号的包。需要使用 `--outdated` 选项。

- `--include-prerelease`

在搜索版本更高的包时，考虑有预发行版本的包。需要使用 `--outdated` 选项。

- `--include-transitive`

除了顶级包之外，还列出可传递包。如果指定此选项，可以获取顶级包所依赖的包列表。

- `--interactive`

允许命令停止并等待用户输入或操作。例如，完成身份验证。自 .NET Core 3.0 SDK 起可用。

- `--outdated`

列出版本更高的包。

- `-s | --source <SOURCE>`

在搜索版本更高的包时，要使用的 NuGet 源。需要使用 `--outdated` 选项。

示例

- 列出特定项目的包引用：

```
dotnet list SentimentAnalysis.csproj package
```

- 列出有更高版本(包括预发行版本)的包引用:

```
dotnet list package --outdated --include-prerelease
```

- 列出特定目标框架的包引用:

```
dotnet list package --framework netcoreapp3.0
```

dotnet remove package

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.x SDK 及更高版本

名称

`dotnet remove package` - 从项目文件删除包引用。

摘要

```
dotnet remove [<PROJECT>] package <PACKAGE_NAME> [-h|--help]
```

说明

使用 `dotnet remove package` 命令可方便地从项目删除 NuGet 包引用。

参数

`PROJECT`

指定项目文件。如果未指定，此命令会搜索当前目录来获取一个项目文件。

`PACKAGE_NAME`

要删除的包引用。

选项

● `-h|--help`

打印出有关命令的简短帮助。

示例

- 从当前目录中的项目删除 `Newtonsoft.Json` NuGet 包：

```
dotnet remove package Newtonsoft.Json
```

如何管理 .NET Core 工具

2020/3/19 • [Edit Online](#)

本文适用于：✓ .NET Core 2.1 SDK 及更高版本

.NET Core 工具是一种特殊的 NuGet 包，其中包含控制台应用程序。可以通过以下方式在计算机上安装该工具：

- 作为全局工具。

工具二进制文件安装在添加到 PATH 环境变量的默认目录中。无需指定工具位置即可从计算机上的任何目录调用该工具。工具的一个版本用于计算机上的所有目录。

- 作为自定义位置中的全局工具(也称为工具路径工具)。

工具二进制文件安装在你指定的位置中。可以从安装目录调用该工具，也可以通过提供具有命令名称的目录或将目录添加到 PATH 环境变量来调用该工具。工具的一个版本用于计算机上的所有目录。

- 作为本地工具(适用于 .NET Core SDK 3.0 及更高版本)。

工具二进制文件安装在默认目录中。可以从安装目录或其任何子目录调用该工具。不同目录可以使用同一工具的不同版本。

.NET CLI 使用清单文件跟踪哪些工具作为本地工具安装到目录。将清单文件保存到源代码存储库的根目录中后，参与者可以克隆存储库并调用用于安装清单文件中列出的所有工具的单个 .NET Core CLI 命令。

IMPORTANT

.NET Core 工具在完全信任环境中运行。除非你信任工具作者，否则请勿安装 .NET Core 工具。

查找工具

目前，.NET Core 没有工具搜索功能。以下是查找工具的一些方法：

- 请参阅 [natemcmaster/dotnet-tools](#) GitHub 存储库中的工具列表。
- 使用 [ToolGet](#) 搜索 .NET 工具。
- 在 [dotnet/aspnetcore GitHub 存储库的工具目录](#) 中查看 ASP.NET Core 团队创建的工具的源代码。
- 在 [.NET Core dotnet 诊断工具](#) 中了解诊断工具。
- 搜索 [NuGet](#) 网站。但是，NuGet 网站尚无可用于仅搜索工具包的功能。

查看作者和统计信息

由于 .NET Core 工具在完全信任环境中运行，并且全局工具已添加到 PATH 环境变量，因此它们的功能非常强大。请勿下载不信任的人提供的工具。

如果该工具在 NuGet 中托管，可以通过搜索该工具来查看作者和统计信息。

安装全局工具

若要将工具作为全局工具安装，请使用 `dotnet tool install` 的 `-g` 或 `--global` 选项，如以下示例中所示：

```
dotnet tool install -g dotnetsay
```

输出显示用于调用该工具和已安装的版本的命令，类似于以下示例：

```
You can invoke the tool using the following command: dotnetsay  
Tool 'dotnetsay' (version '2.1.4') was successfully installed.
```

工具二进制文件的默认位置取决于操作系统：

(OS)	II
Linux/macOS	\$HOME/.dotnet/tools
Windows	%USERPROFILE%\.dotnet\tools

首次运行 SDK 时，会将此位置添加到用户的路径，因此，无需指定工具位置即可从任何目录调用全局工具。

工具访问特定于用户，而不针对计算机全局。全局工具仅适用于安装了该工具的用户。

安装自定义位置中的全局工具

若要将工具作为自定义位置中的全局工具安装，请使用 [dotnet tool install](#) 的 `--tool-path` 选项，如以下示例中所示。

在 Windows 上：

```
dotnet tool install dotnetsay --tool-path c:\dotnet-tools
```

在 Linux 或 macOS 上：

```
dotnet tool install dotnetsay --tool-path ~/bin
```

.NET Core SDK 不将此位置自动添加至 PATH 环境变量。若要调用工具路径工具，必须确保可使用以下方法之一来调用命令：

- 将安装目录添加到 PATH 环境变量。
- 调用该工具时，指定该工具的完整路径。
- 从安装目录调用该工具。

安装本地工具

适用于 .NET Core 3.0 SDK 及更高版本。

若要安装仅用于本地访问的工具（对于当前目录和子目录），必须将其添加到工具清单文件。若要创建工具清单文件，请运行 [dotnet new tool-manifest](#) 命令：

```
dotnet new tool-manifest
```

此命令在“.config”目录下创建一个名为“dotnet-tools.json”的清单文件。若要将本地工具添加到清单文件，请使用 [dotnet tool install](#) 命令并省略 `--global` 和 `--tool-path` 选项，如以下示例中所示：

```
dotnet tool install dotnetsay
```

命令输出显示新安装的工具所在的清单文件，类似于以下示例：

```
You can invoke the tool from this directory using the following command:  
dotnet tool run dotnetsay  
Tool 'dotnetsay' (version '2.1.4') was successfully installed.  
Entry is added to the manifest file /home/name/botsay/.config/dotnet-tools.json.
```

以下示例显示安装了两个本地工具的清单文件：

```
{  
    "version": 1,  
    "isRoot": true,  
    "tools": {  
        "botsay": {  
            "version": "1.0.0",  
            "commands": [  
                "botsay"  
            ]  
        },  
        "dotnetsay": {  
            "version": "2.1.3",  
            "commands": [  
                "dotnetsay"  
            ]  
        }  
    }  
}
```

通常将本地工具添加到存储库的根目录。将清单文件签入到存储库后，从存储库中签出代码的开发人员会获得最新的清单文件。若要安装清单文件中列出的所有工具，请运行 `dotnet tool restore` 命令：

```
dotnet tool restore
```

输出表明还原了哪些工具：

```
Tool 'botsay' (version '1.0.0') was restored. Available commands: botsay  
Tool 'dotnetsay' (version '2.1.3') was restored. Available commands: dotnetsay  
Restore was successful.
```

安装特定工具版本

若要安装工具的预发布版本或特定版本，请使用 `--version` 选项指定版本号，如以下示例中所示：

```
dotnet tool install dotnetsay --version 2.1.3
```

使用工具

用于调用工具的命令可能不同于安装的包的名称。若要显示计算机上目前安装的所有工具，请使用 [dotnet tool list](#) 命令：

```
dotnet tool list
```

输出显示每个工具的版本和命令，类似于以下示例：

Package Id	Version	Commands	Manifest
botsay	1.0.0	botsay	/home/name/repository/.config/dotnet-tools.json
dotnetsay	2.1.3	dotnetsay	/home/name/repository/.config/dotnet-tools.json

如本示例中所示，列表显示了本地工具。若要查看全局工具，请使用 `--global` 选项，若要查看工具路径工具，请使用 `--tool-path` 选项。

调用全局工具

对于全局工具，请单独使用工具命令。例如，如果命令为 `dotnetsay` 或 `dotnet-doc`，则可以使用以下命令调用该工具：

```
dotnetsay  
dotnet-doc
```

如果命令以前缀 `dotnet-` 开头，则调用该工具的另一种方法是使用 `dotnet` 命令并省略工具命令前缀。例如，如果命令为 `dotnet-doc`，则可以使用以下命令调用该工具：

```
dotnet doc
```

但是，在以下情况下，不能使用 `dotnet` 命令来调用全局工具：

- 全局工具和本地工具具有以 `dotnet-` 为前缀的相同命令。
- 你希望从本地工具范围内的目录调用全局工具。

在这种情况下，`dotnet doc` 和 `dotnet dotnet-doc` 调用本地工具。若要调用全局工具，请单独使用命令：

```
dotnet-doc
```

调用工具路径工具

若要调用使用 `tool-path` 选项安装的全局工具，请确保该命令可用，如[本文前面](#)所述。

调用本地工具

若要调用本地工具，必须从安装目录使用 `dotnet` 命令。可以使用长格式 (`dotnet tool run <COMMAND_NAME>`) 或短格式 (`dotnet <COMMAND_NAME>`)，如以下示例中所示：

```
dotnet tool run dotnetsay  
dotnet dotnetsay
```

如果命令以 `dotnet-` 为前缀，则可以在调用该工具时包括或省略前缀。例如，如果命令为 `dotnet-doc`，则可以使用以下任何示例调用本地工具：

```
dotnet tool run dotnet-doc  
dotnet dotnet-doc  
dotnet doc
```

更新工具

更新工具涉及卸载该工具并重新安装它的最新稳定版。若要更新工具，请使用具有用于安装该工具的相同选项的 [dotnet tool update](#) 命令：

```
dotnet tool update --global <packagename>
dotnet tool update --tool-path <packagename>
dotnet tool update <packagename>
```

对于本地工具，SDK 通过在当前目录和父目录中查找来查找包含包 ID 的第一个清单文件。如果任何清单文件中都没有此类包 ID，SDK 会将新条目添加到最近的清单文件。

卸载工具

使用具有用于安装该工具的相同选项的 [dotnet tool uninstall](#) 命令删除工具：

```
dotnet tool uninstall --global <packagename>
dotnet tool uninstall --tool-path <packagename>
dotnet tool uninstall <packagename>
```

对于本地工具，SDK 通过在当前目录和父目录中查找来查找包含包 ID 的第一个清单文件。

获取帮助和疑难解答

若要获取可用 `dotnet tool` 命令的列表，请输入以下命令：

```
dotnet tool --help
```

若要获取工具使用说明，请输入以下命令之一，或访问工具的网站：

```
<command> --help
dotnet <command> --help
```

如果工具无法安装或运行，请参阅[排查 .NET Core 工具使用问题](#)。

请参阅

- [教程：使用 .NET Core CLI 创建 .NET Core 工具](#)
- [教程：使用 .NET Core CLI 安装和使用 .NET Core 全局工具](#)
- [教程：使用 .NET Core CLI 安装和使用 .NET Core 本地工具](#)

排查 .NET Core 工具使用问题

2020/3/19 • [Edit Online](#)

在尝试安装或运行 .NET Core 工具(可能是全局工具, 也可能是本地工具)时, 可能会遇到问题。本文介绍了常见的根本原因及一些可能的解决方案。

安装的 .NET Core 工具未能运行

当 .NET Core 工具未能运行时, 你最可能遇到下述问题之一:

- 找不到工具的可执行文件。
- 找不到 .NET Core 运行时的正确版本。

找不到可执行文件

如果找不到可执行文件, 则将看到如下所示的消息:

```
Could not execute because the specified command or file was not found.  
Possible reasons for this include:  
* You misspelled a built-in dotnet command.  
* You intended to execute a .NET Core program, but dotnet-xyz does not exist.  
* You intended to run a global tool, but a dotnet-prefixed executable with this name could not be found on  
the PATH.
```

可执行文件的名称决定了调用工具的方式。相关格式请参见下表:

命令	示例
<code>dotnet-<toolName>.exe</code>	<code>dotnet <toolName></code>
<code><toolName>.exe</code>	<code><toolName></code>

- 全局工具

全局工具可安装在默认目录中, 也可安装在特定位置中。默认目录为:

(OS)	示例
Linux/macOS	<code>\$HOME/.dotnet/tools</code>
Windows	<code>%USERPROFILE%\.dotnet\tools</code>

如果要尝试运行全局工具, 请检查计算机上的 `PATH` 环境变量是否包含安装该全局工具的路径且可执行文件是否位于该路径中。

.NET Core CLI 在首次使用时尝试将默认位置添加到 `PATH` 环境变量。但是, 在某些情况下, 位置不会自动添加至 `PATH`:

- 如果要使用 Linux, 并且已使用 `.tar.gz` 文件(而非 `apt-get` 或 `rpm`)安装 .NET Core SDK。
- 如果使用的是 macOS 10.15“Catalina”或更高版本。
- 如果要使用 macOS 10.14“Mojave”或更低版本, 并且已使用 `.tar.gz` 文件(而非 `.pkg`)安装 .NET Core SDK。

- 如果已安装 .NET Core 3.0 SDK，并且已将 `DOTNET_ADD_GLOBAL_TOOLS_TO_PATH` 环境变量设置为 `false`。
- 如果已安装 .NET Core 2.2 SDK 或更低版本，并且已将 `DOTNET_SKIP_FIRST_TIME_EXPERIENCE` 环境变量设置为 `true`。

在这些情况下，或者如果指定了 `--tool-path` 选项，则计算机上的 `PATH` 环境变量不会自动包含安装全局工具的路径。在这种情况下，使用 shell 提供的用于更新环境变量的任何方法，将工具位置（例如 `$HOME/.dotnet/tools`）追加到 `PATH` 环境变量。有关详细信息，请参阅 [.NET Core 工具](#)。

- 本地工具

如果要尝试运行本地工具，请验证当前目录或其任何父目录中是否存在一个名为 `dotnet-tools.json` 的清单文件。此文件还可位于项目文件夹层次结构中任意位置的 `.config` 文件夹下，而不是位于根文件夹中。如果存在 `dotnet-tools.json`，请将其打开，检查是否存在你要尝试运行的工具。如果该文件不包含 `"isRoot": true` 的条目，则还要进一步检查文件层次结构中是否存在其他工具清单文件。

如果要尝试运行已通过指定的路径安装的 .NET Core 工具，则需要在使用该工具时包含该路径。下面是使用安装了工具路径的工具的示例：

```
..\<toolDirectory>\dotnet-<toolName>
```

找不到运行时

.NET Core 工具是[依赖框架的应用程序](#)，也就是说它们依赖于计算机上安装的 .NET Core 运行时。如果找不到所需的运行时，则遵循常规的 .NET Core 运行时前滚规则，例如：

- 应用程序前滚至指定的主要版本和次要版本的最高修补程序版本。
- 如果主要版本号和次要版本号没有匹配的运行时，则使用下一个较高的次要版本。
- 前滚不会发生在运行时的预览版之间，也不会发生在预览版和发行版之间。因此，使用预览版创建的 .NET Core 工具必须由作者重新生成和重新发布，再重新安装。

在下面两种常见场景中，默认不进行前滚：

- 只有运行时的较低版本可用。前滚仅选择运行时的较高版本。
- 只有运行时的较高版本可用。前滚不跨越主要版本边界。

如果应用程序找不到合适的运行时，则它无法运行并报告错误。

要查看计算机上安装了哪些 .NET Core 运行时，可使用下述命令之一：

```
dotnet --list-runtimes  
dotnet --info
```

如果你认为此工具应支持你当前安装的运行时版本，可联系工具作者，询问他们是否可更新版本号或实现多目标。在工具作者编译其工具包并使用更新后的版本号将其重新发布到 NuGet 之后，你可更新你的副本。虽然这种情况不会发生，但最快捷的解决方案是安装适合你要尝试运行的工具的运行时版本。要下载特定的 .NET Core 运行时版本，请访问 [.NET Core 下载页](#)。

如果将 .NET Core SDK 安装到非默认位置，则需要将环境变量 `DOTNET_ROOT` 设置到包含 `dotnet` 可执行文件的目录。

.NET Core 工具安装失败

.NET Core 全局或本地工具安装失败的原因有很多。当工具安装失败时，你将看到如下所示的消息：

```
Tool '{0}' failed to install. This failure may have been caused by:  
  
* You are attempting to install a preview release and did not use the --version option to specify the version.  
* A package by this name was found, but it was not a .NET Core tool.  
* The required NuGet feed cannot be accessed, perhaps because of an Internet connection problem.  
* You mistyped the name of the tool.  
  
For more reasons, including package naming enforcement, visit https://aka.ms/failure-installing-tool
```

为帮助诊断这些失败情况，会随同之前的消息直接向用户显示 NuGet 消息。NuGet 消息可帮助你确定问题。

包命名强制

Microsoft 针对工具的包 ID 更改了相关指导，导致没法用预测出的名称找到很多工具。新的指导是所有 Microsoft 工具均附上“Microsoft”这一前缀。此前缀已预留，仅用于使用 Microsoft 授权证书签名的包。

在过渡期间，一些 Microsoft 工具将采用包 ID 的旧格式，而另外一些将采用新格式：

```
dotnet tool install -g Microsoft.<toolName>  
dotnet tool install -g <toolName>
```

更新包 ID 后，你将需要更改到新的包 ID 以获取最新更新。带简化工具名称的包将遭到弃用。

预览版本

- 你正在尝试安装预览版本，但未使用 `--version` 选项来指定该版本。

必须用名称的一部分指定处于预览版状态的 .NET Core 工具，这样才能指示它们现为预览版。不需要包含整个预览版。假定版本号都采用预期的格式，则你可使用与下例类似的内容：

```
dotnet tool install -g --version 1.1.0-pre <toolName>
```

包不是 .NET Core 工具

- 已按此名称找到 NuGet 包，但它不是 .NET Core 工具。

如果尝试安装是常规 NuGet 包（而非 .NET Core 工具）的 NuGet 包，你将看到如下所示的错误：

```
NU1212: <ToolName> 的项目包组合无效。DotnetToolReference 项目类型仅可包含 DotnetTool 类型的引用。
```

无法访问 NuGet 源

- 无法访问必需的 NuGet 源，这可能是由 Internet 连接问题造成的。

需要访问包含工具包的 NuGet 源才能安装工具。如果源不可用，则安装将失败。可使用 `nuget.config` 修改源、请求特定的 `nuget.config` 文件，也可使用 `--add-source` 开关指定其他源。默认情况下，对于无法连接的任何源，NuGet 都将引发错误。标志 `--ignore-failed-sources` 可跳过这些不可访问的源。

包 ID 错误

- 工具的名称拼写错误。

常见的失败原因是工具名称不正确。原因可能是拼写错误，或者工具已移动或已被弃用。对于 NuGet.org 上的工具，确保名称正确的一种方式是在 NuGet.org 处搜索该工具并复制安装命令。

请参阅

- [.NET Core 工具](#)

global.json 概述

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 2.0 SDK 及更高版本

通过 global.json 文件，可定义在运行 .NET Core CLI 命令时使用的 .NET Core SDK 版本。选择 .NET Core SDK 与指定项目所面向的运行时无关。.NET Core SDK 版本指示使用哪个版本的 .NET Core CLI。

通常会使用最新版 SDK 工具，因此不需要 global.json 文件。在某些高级方案中，你可能需要控制 SDK 工具的版本，本文对如何完成此操作进行了介绍。

有关指定运行时的详细信息，请参阅 [目标框架](#)。

.NET Core SDK 在当前工作目录(不必与项目目录相同)或其某个父目录中查找 global.json 文件。

global.json 架构

SDK

类型: `object`

指定要选择的 .NET Core SDK 的相关信息。

version

- 类型: `string`
- 自 .NET Core 1.0 SDK 起可用。

要使用的 .NET Core SDK 版本。

此字段:

- 不支持通配符，也就是说，必须指定完整版本号。
- 不支持版本范围。

allowPrerelease

- 类型: `boolean`
- 自 .NET Core 3.0 SDK 起可用。

指示在选择要使用的 SDK 版本时，SDK 解析程序是否应考虑预发布版本。

如果未显式设置此值，则默认值将取决于是否从 Visual Studio 运行：

- 如果未使用 Visual Studio，则默认值为 `true`。
- 如果使用 Visual Studio，它将使用请求的预发布状态。也就是说，如果使用 Visual Studio 的预览版本，或者设置了“使用 .NET Core SDK 的预览版”选项(在“工具”>“选项”>“环境”>“预览功能”下方)，则默认值为 `true`，否则为 `false`。

rollForward

- 类型: `string`
- 自 .NET Core 3.0 SDK 起可用。

选择 SDK 版本时要使用的前滚策略，可作为特定 SDK 版本缺失时的回退，或者作为使用更高版本的指令。必须使用 `rollForward` 值指定 [版本](#)，除非将其设置为 `latestMajor`。

要了解可用策略及其行为, 请考虑以下格式为 `x.y.znn` 的 SDK 版本定义:

- `x` 是主版本。
- `y` 是次版本。
- `z` 是功能区段。
- `nn` 是修补程序版本。

下表列出了 `rollForward` 键的可能值:

"I"	"II"
<code>patch</code>	<p>使用指定的版本。 如果找不到, 则前滚到最新的修补程序级别。 如果找不到, 则失败。</p> <p>此值是早期 SDK 版本中的旧行为。</p>
<code>feature</code>	<p>对指定的主版本、次版本和功能区段使用最新的修补程序级别。 如果找不到, 则前滚到同一主/次版本中的下一个较高功能区段, 并使用该功能区段的最新修补程序级别。 如果找不到, 则失败。</p>
<code>minor</code>	<p>对指定的主版本、次版本和功能区段使用最新的修补程序级别。 如果找不到, 则前滚到同一主/次版本中的下一个较高功能区段, 并使用该功能区段的最新修补程序级别。 如果找不到, 则前滚到同一主版本中的下一个较高次版本和功能区段, 并使用该功能区段的最新修补程序级别。 如果找不到, 则失败。</p>
<code>major</code>	<p>对指定的主版本、次版本和功能区段使用最新的修补程序级别。 如果找不到, 则前滚到同一主/次版本中的下一个较高功能区段, 并使用该功能区段的最新修补程序级别。 如果找不到, 则前滚到同一主版本中的下一个较高次版本和功能区段, 并使用该功能区段的最新修补程序级别。 如果找不到, 则前滚到下一个较高主版本、次版本和功能区段, 并使用该功能区段的最新修补程序级别。 如果找不到, 则失败。</p>
<code>latestPatch</code>	<p>使用安装的最新修补程序级别, 以与请求的主版本、次版本和功能区段匹配, 并且该修补程序级别大于或等于指定的值。 如果找不到, 则失败。</p>
<code>latestFeature</code>	<p>使用安装的最高功能区段和修补程序级别, 以与请求的主版本和次版本匹配, 并且该功能区段大于或等于指定的值。 如果找不到, 则失败。</p>
<code>latestMinor</code>	<p>使用安装的最高次版本、功能区段和修补程序级别, 以与请求的主版本匹配, 并且该次版本大于或等于指定的值。 如果找不到, 则失败。</p>
<code>latestMajor</code>	<p>使用安装的最高 .NET Core SDK, 并且该主版本大于或等于指定的值。 如果找不到, 则失败。</p>
<code>disable</code>	<p>不前滚。需要完全匹配。</p>

示例

下面的示例演示如何不使用预发布版本：

```
{  
  "sdk": {  
    "allowPrerelease": false  
  }  
}
```

下面的示例演示如何使用安装的大于或等于指定版本的最高版本：

```
{  
  "sdk": {  
    "version": "3.1.100",  
    "rollForward": "latestMajor"  
  }  
}
```

下面的示例演示如何使用完全指定的版本：

```
{  
  "sdk": {  
    "version": "3.1.100",  
    "rollForward": "disable"  
  }  
}
```

下面的示例演示如何使用安装的特定版本(格式为 3.1.1xx)的最高修补程序版本：

```
{  
  "sdk": {  
    "version": "3.1.100",  
    "rollForward": "latestPatch"  
  }  
}
```

global.json 和 .NET Core CLI

最好能知道计算机上安装了哪些 SDK 版本，以便在 global.json 文件中设置相应版本。有关如何执行此操作的详细信息，请参阅[如何检查是否已安装 .NET Core](#)。

若要在计算机上安装其他 .NET Core SDK 版本，请访问[.NET Core 下载](#)页面。

可执行 `dotnet new` 命令在当前目录中创建一个新的 global.json 文件，类似于以下示例：

```
dotnet new globaljson --sdk-version 3.0.100
```

匹配规则

NOTE

匹配规则由 `dotnet.exe` 入口点控制，该入口点在所有已安装的 .NET Core 运行时中很常见。如果并行安装了多个运行时，则使用已安装的最新版 .NET Core 运行时的匹配规则。

- .NET Core 3.x
- .NET Core 2.x

从 .NET Core 3.0 开始，在确定要使用的 SDK 版本时，适用以下规则：

- 如果未找到 global.json 文件，或者 global.json 未指定 SDK 版本和 `allowPrerelease` 值，则使用安装的最高 SDK 版本(相当于将 `rollForward` 设置为 `latestMajor`)。是否考虑 SDK 预发布版本取决于 `dotnet` 的调用方式。
 - 如果未使用 Visual Studio，则考虑预发布版本。
 - 如果使用 Visual Studio，它将使用请求的预发布状态。也就是说，如果使用 Visual Studio 的预览版本，或者设置了“使用 .NET Core SDK 的预览版”选项(在“工具”>“选项”>“环境”>“预览功能”下方)，则考虑预发布版本；否则仅考虑发布版本。
- 如果找到了未指定 SDK 版本但指定了 `allowPrerelease` 值的 global.json 文件，则使用安装的最高 SDK 版本(相当于将 `rollForward` 设置为 `latestMajor`)。最新 SDK 版本是发布版本还是预发布版本取决于 `allowPrerelease` 的值。`true` 指示考虑预发布版本；`false` 指示仅考虑发布版本。
- 如果找到 global.json 文件，并且该文件指定了 SDK 版本：
 - 如果未设置 `rollForward` 值，它将使用 `latestPatch` 作为默认 `rollForward` 策略。否则，请在 `rollForward` 部分中检查每个值及其行为。
 - 有关是否考虑预发布版本以及未设置 `allowPrerelease` 时的默认行为的信息，请参阅 `allowPrerelease` 部分。

针对生成警告的疑难解答

- 以下警告指示你的项目使用 .NET Core SDK 的预发布版本进行编译：

使用的是 .NET Core SDK 的预览版本，可通过当前项目中的 global.json 文件定义 SDK 版本。有关详细信息，请访问 <https://go.microsoft.com/fwlink/?linkid=869452>。

.NET Core SDK 的各种版本均品质优良稳定，在业内口碑良好。但是，如果不希望使用预发布版本，请在 `allowPrerelease` 部分中查看可用于 .NET Core 3.0 SDK 或更高版本的各种策略。对于从未安装 .NET Core 3.0 或更高版本运行时或 SDK 的计算机，需要创建一个 global.json 文件，并指定要使用的确切版本。

- 以下警告指示项目面向 EF Core 1.0 或 1.1，后者与 .NET Core 2.1 SDK 及更高版本不兼容：

启动项目 '{startupProject}' 面向框架 '.NETCoreApp' 的版本 '{targetFrameworkVersion}'。此版本的 Entity Framework Core .NET 命令行工具仅支持 2.0 或更高版本。有关使用旧版工具的信息，请参阅 <https://go.microsoft.com/fwlink/?linkid=871254>。

从 .NET Core 2.1 SDK(版本 2.1.300)开始，SDK 中包含 `dotnet ef` 命令。若要编译项目，请在计算机上安装 .NET Core 2.0 SDK(版本 2.1.201)或更早版本，并使用 global.json 文件定义所需 SDK 版本。有关 `dotnet ef` 命令的详细信息，请参阅 [EF Core .NET 命令行工具](#)。

请参阅

- [如何解析 SDK 项目](#)

dotnet new 自定义模板

2020/3/18 • [Edit Online](#)

.NET Core SDK 随附了许多已安装并可供你使用的模板。`dotnet new` 命令不仅用于使用模板，还用于说明如何安装和卸载模板。自.NET Core 2.0 起，可以为任何类型的项目（如应用程序、服务、工具或类库）创建自己的自定义模板。甚至可以创建输出一个或多个独立文件（如配置文件）的模板。

可以从任何 NuGet 源上的 NuGet 包安装自定义模板，具体方法是直接引用 NuGet .nupkg 文件，或指定包含模板的文件系统目录。借助模板引擎提供的功能，可以替换值、包括和排除文件，并在使用模板时执行自定义处理操作。

模板引擎是开放源代码，在线代码存储库位于 GitHub 上的 [dotnettemplating](#)。有关模板示例，请访问 [dotnet/dotnet-template-samples](#) 存储库。GitHub 上的 [dotnet new 可用模板](#) 收录了更多模板，包括第三方模板。若要详细了解如何创建和使用自定义模板，请参阅[如何创建自己的 dotnet new 模板](#)和[dotnet/templating GitHub 存储库 Wiki](#)。

若要按照演示步骤操作并创建模板，请参阅[创建 dotnet new 自定义模板](#)教程。

.NET 默认模板

安装 .NET Core SDK 时，将获取十多个用于创建项目和文件的内置模板，包括控制台应用程序、类库、单元测试项目、ASP.NET Core 应用程序（包括 Angular 和 React 项目）和配置文件。若要列出内置模板，请运行带有 `dotnet new` 选项的 `-l|--list` 命令：

```
dotnet new --list
```

Configuration

模板由以下部分组成：

- 源文件和文件夹。
- 配置文件 (`template.json`)。

源文件和文件夹

源文件和文件夹包含运行 `dotnet new <TEMPLATE>` 命令时用户希望模板引擎使用的任何文件和文件夹。模板引擎旨在将可运行项目用作源代码，以生成项目。这样做有以下几个好处：

- 模板引擎不要求用户将特殊令牌注入项目的源代码。
- 代码文件不必是特殊文件，也不必以任何方式进行修改，即可与模板引擎配合使用。因此，处理项目时通常使用的工具也适用于模板内容。
- 生成、运行和调试模板项目，就像生成、运行和调试其他任何项目一样。
- 只需将 `./template.config/template.json` 配置文件添加到项目，即可通过现有项目快速创建模板。

模板中存储的文件和文件夹并不限于正式的 .NET 项目类型。源文件和文件夹可能包含用户希望在使用模板时创建的任何内容，即使模板引擎仅生成一个文件作为输出也不例外。

可以基于在 `template.json` 配置文件中提供的逻辑和设置对模板生成的文件进行修改。用户可以将选项传递到 `dotnet new <TEMPLATE>` 命令以覆盖这些设置。自定义逻辑的一个常见示例为模板部署的代码文件中的类或变量提供名称。

template.json

`template.json` 文件位于模板根目录中的 `.template.config` 文件夹。此文件向模板引擎提供配置信息。最低配置必

须包含下表中列出的成员，这足以创建功能模板。

成员	类型	说明
<code>\$schema</code>	URI	template.json 文件的 JSON 架构。如果指定架构，支持 JSON 架构的编辑器启用 JSON 编辑功能。例如， Visual Studio Code 要求此成员启用 IntelliSense。使用值 <code>http://json.schemastore.org/template</code> 。
<code>author</code>	string	模板创建者。
<code>classifications</code>	array(string)	为了找到模板，用户可能会在搜索模板时使用的 0 个或多个模板特征。如果出现在使用 命令生成的模板列表 中， <code>classifications</code> 还会出现在“Tags” <code>dotnet new -l --list</code> 列中。
<code>identity</code>	string	此模板的唯一名称。
<code>name</code>	string	用户应看到的模板名称。
<code>shortName</code>	string	方便用户选择模板的默认速记名称，适用于模板名称由用户指定（而不是通过 GUI 选择）的环境。例如，通过命令提示符和 CLI 命令使用模板时，短名称非常有用。

template.json 文件的完整架构位于 [JSON 架构存储](#)。有关 template.json 文件的详细信息，请参阅 [dotnet 创建模板](#) [wiki](#)。

示例

例如，下面是包含两个内容文件的模板文件夹：console.cs 和 readme.txt。请注意，其中有包含 template.json 文件的名为 .template.config 的所需文件夹。

```
└── mytemplate
    ├── console.cs
    └── readme.txt
    └── .template.config
        └── template.json
```

template.json 文件如下所示：

```
{
  "$schema": "http://json.schemastore.org/template",
  "author": "Travis Chau",
  "classifications": [ "Common", "Console" ],
  "identity": "AdatumCorporation.ConsoleTemplate.CSharp",
  "name": "Adatum Corporation Console Application",
  "shortName": "adatumconsole"
}
```

mytemplate 文件夹是可安装的模板包。安装此包后，`shortName` 可与 `dotnet new` 命令结合使用。例如，`dotnet new adatumconsole` 会将 `console.cs` 和 `readme.txt` 文件输出到当前文件夹。

将模板打包到 NuGet 包 (.nupkg 文件)

自定义模板与 `dotnet pack` 命令和 .csproj 文件一起打包。或者，NuGet 可与 `nuget pack` 命令以及 .nuspec 文件一起使用。但是，NuGet 在 Windows 上需要 .NET Framework，在 Linux 和 MacOS 上需要 Mono。

该 .csproj 文件与传统代码项目 .csproj 文件略有不同。请注意以下设置：

1. 添加 `<PackageType>` 设置并将其设为 `Template`。
2. 添加 `<PackageVersion>` 设置并将其设为有效的 NuGet 版本号。
3. 添加 `<PackageId>` 设置并将其设为唯一标识符。此标识符用于卸载模板包，NuGet 源用它来注册你的模板包。
4. 应设置泛型元数据设置：`<Title>`、`<Authors>`、`<Description>` 和 `<PackageTags>`。
5. 必须设置 `<TargetFramework>` 设置，即使未使用模板过程生成的二进制文件也必须设置。在下面的示例中，它设置为 `netstandard2.0`。

.nupkg NuGet 包形式的模板包要求所有模板都存储在包中的 `content` 文件夹中。还有几个设置将添加到 .csproj 文件以确保生成的 .nupkg 作为模板包安装：

1. `<IncludeContentInPack>` 设置设为 `true` 以包含项目在 NuGet 包中设为“内容”的任何文件。
2. `<IncludeBuildOutput>` 设置设为 `false` 以从 NuGet 包排除编译器生成的所有二进制文件。
3. `<ContentTargetFolders>` 设置设为 `content`。这可确保设为“内容”的文件存储在 NuGet 包的 `content` 文件夹中。NuGet 包中的此文件夹由 dotnet 模板系统解析。

使所有代码文件不被模板项目编译的一个简单的方法是使用 `<Compile Remove="****" />` 元素内项目文件中的 `<ItemGroup>` 项。

设置模板包结构的一个简单方法是将所有模板放在单独的文件夹中，然后放到位于 .csproj 文件所在目录的 `templates` 文件夹的每个模板文件夹中。这样，你可以使用单个项目项包括 `templates` 中的所有文件和文件夹作为“内容”。在 `<ItemGroup>` 元素中创建

```
<Content Include="templates\**\**" Exclude="templates\**\bin\**;templates\**\obj\**" />
```

下面是一个遵循上述所有准则的示例 .csproj 文件。它将 `templates` 子文件夹打包为“内容”包文件夹，并使所有代码文件都不被编译。

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <PackageType>Template</PackageType>
    <PackageVersion>1.0</PackageVersion>
    <PackageId>AdatumCorporation.Utility.Templates</PackageId>
    <Title>AdatumCorporation Templates</Title>
    <Authors>Me</Authors>
    <Description>Templates to use when creating an application for Adatum Corporation.</Description>
    <PackageTags>dotnet-new;templates;contoso</PackageTags>
    <TargetFramework>netstandard2.0</TargetFramework>

    <IncludeContentInPack>true</IncludeContentInPack>
    <IncludeBuildOutput>false</IncludeBuildOutput>
    <ContentTargetFolders>content</ContentTargetFolders>
  </PropertyGroup>

  <ItemGroup>
    <Content Include="templates\**\**" Exclude="templates\**\bin\**;templates\**\obj\**" />
    <Compile Remove="**\**" />
  </ItemGroup>

</Project>
```

以下示例演示使用 .csproj 创建模板包的文件和文件夹结构。MyDotnetTemplates.csproj 文件和 templates 文件夹都位于名为 project_folder 的根目录中。templates 文件夹包含两个模板 mytemplate1 和 mytemplate2。每个模

板具有内容文件和包含 template.json 配置文件的 .template.config 文件夹。

```
project_folder
|   MyDotnetTemplates.csproj
|
└── templates
    ├── mytemplate1
    |   ├── console.cs
    |   └── readme.txt
    |
    └── .template.config
        └── template.json
    └── mytemplate2
        ├── otherfile.cs
        └── .template.config
            └── template.json
```

安装模板

使用 `dotnet new -i|--install` 命令以安装包。

从 nuget.org 中存储的 NuGet 包安装模板的具体步骤

使用 NuGet 包标识符安装模板包。

```
dotnet new -i <NUGET_PACKAGE_ID>
```

从本地 nupkg 文件安装模板的具体步骤

提供指向 .nupkg NuGet 包文件的路径。

```
dotnet new -i <PATH_TO_NUPKG_FILE>
```

从文件系统目录安装模板的具体步骤

模板可从模板文件夹安装，如以上示例中的 mytemplate1 文件夹。指定 .template.config 文件夹的文件夹路径。

模板目录的路径不需要是绝对路径。但是，卸载从文件夹安装的模板时需要绝对路径。

```
dotnet new -i <FILE_SYSTEM_DIRECTORY>
```

获取已安装模板的列表

在没有任何其他参数的情况下，卸载命令将列出所有已安装的模板。

```
dotnet new -u
```

该命令返回如下所示的输出：

```
Template Instantiation Commands for .NET Core CLI
```

```
Currently installed items:
```

```
Microsoft.DotNet.Common.ItemTemplates
  Templates:
    global.json file (globaljson)
    NuGet Config (nugetconfig)
    Solution File (sln)
    Dotnet local tool manifest file (tool-manifest)
    Web Config (webconfig)
Microsoft.DotNet.Common.ProjectTemplates.3.0
  Templates:
    Class library (classlib) C#
    Class library (classlib) F#
    Class library (classlib) VB
    Console Application (console) C#
    Console Application (console) F#
    Console Application (console) VB
...
...
```

`Currently installed items:` 后面的第一级项是用于卸载模板的标识符。在上述示例中，列出了 `Microsoft.DotNet.Common.ItemTemplates` 和 `Microsoft.DotNet.Common.ProjectTemplates.3.0`。如果使用文件系统路径安装模板，此标识符将是 `.template.config` 文件夹的文件夹路径。

卸载模板

使用 `dotnet new -u|--uninstall` 命令卸载包。

如果通过 NuGet 源或直接通过 `.nupkg` 文件安装包，请提供标识符。

```
dotnet new -u <NUGET_PACKAGE_ID>
```

如果通过指定 `.template.config` 文件夹的路径安装包，请使用该绝对路径卸载包。你可以在 `dotnet new -u` 命令提供的输出中看到模板的绝对路径。有关详细信息，请参阅上文的[获取已安装的模板列表](#)部分。

```
dotnet new -u <ABSOLUTE_FILE_SYSTEM_DIRECTORY>
```

使用自定义模板创建项目

安装模板后，通过执行 `dotnet new <TEMPLATE>` 命令来使用模板，就像使用其他任何预安装模板一样。还可以为命令指定选项 `dotnet new`，包括在模板设置中配置的模板专用选项。直接向命令提供模板的短名称：

```
dotnet new <TEMPLATE>
```

另请参阅

- [创建 dotnet new 自定义模板\(教程\)](#)
- [dotnettemplating GitHub 存储库 Wiki](#)
- [dotnet/dotnet-template-samples GitHub 存储库](#)
- [如何创建自己的 dotnet new 模板](#)
- [JSON 架构存储中的 template.json 架构](#)

.NET Core SDK 遥测

2020/4/2 • [Edit Online](#)

.NET Core SDK 包含遥测功能，可在 .NET Core CLI 故障时收集使用情况数据和异常信息。.NET Core CLI 附带 .NET Core SDK，是一组用于生成、测试和发布 .NET Core 应用的谓词。请务必让 .NET 团队了解到工具使用情况，以便我们对其做出改进。有关故障的信息可帮助团队解决问题并修复 bug。

数据为匿名收集，并根据 [Creative Commons Attribution 许可证](#) 以汇总形式发布。

范围

`dotnet` 具有两个功能：运行应用程序和执行 CLI 命令。按以下格式使用 `dotnet` 来启动应用程序时，不会收集遥测数据：

- `dotnet [path-to-app].dll`

使用任何 .NET Core CLI 命令时，都会收集遥测数据，如：

- `dotnet build`
- `dotnet pack`
- `dotnet run`

如何选择退出

.NET Core SDK 遥测功能默认处于启用状态。要选择退出遥测功能，请将 `DOTNET_CLI_TELEMETRY_OPTOUT` 环境变量设置为 `1` 或 `true`。

如果安装成功，.NET Core SDK 安装程序也会发送一个遥测条目。要选择退出，请在安装 .NET Core SDK 之前设置 `DOTNET_CLI_TELEMETRY_OPTOUT` 环境变量。

公开

首次运行其中一个 .NET Core CLI 命令（例如，`dotnet build`）时，.NET Core SDK 显示以下类似文本。文本可能会因运行的 SDK 版本而略有不同。此“首次运行”体验是 Microsoft 通知用户有关数据收集信息的方式。

```
Telemetry
-----
The .NET Core tools collect usage data in order to help us improve your experience. The data is anonymous. It
is collected by Microsoft and shared with the community. You can opt-out of telemetry by setting the
DOTNET_CLI_TELEMETRY_OPTOUT environment variable to '1' or 'true' using your favorite shell.

Read more about .NET Core CLI Tools telemetry: https://aka.ms/dotnet-cli-telemetry
```

若要禁用此消息和 .NET Core 欢迎消息，请将 `DOTNET_NOLOGO` 环境变量设置为 `true`。请注意，此变量在遥测选择退出时不起作用。

数据点

遥测功能不收集用户名或电子邮件地址等个人数据。也不会扫描代码，更不会提取项目级敏感数据，如名称、存储库或作者。数据通过 [Azure Monitor](#) 技术安全地发送到 Microsoft 服务器，提供对保留数据的受限访问权限，并在严格的安全控制下从安全的 [Azure 存储](#) 系统发布。

保护你的隐私对我们很重要。如果怀疑遥测在收集敏感数据，或认为我们处理数据的方式不安全或不恰当，请在 [dotnet/cli 存储库](#) 中记录问题或发送电子邮件至 dotnet@microsoft.com 进行调查。

遥测功能收集以下数据：

SDK 版本	遥测数据
全部	调用时间戳。
全部	调用的命令(例如，“build”)，从 2.1 开始进行哈希处理。
全部	用于确定地理位置的三个八进制数 IP 地址。
全部	操作系统和版本。
全部	运行 SDK 的运行时 ID (RID)。
全部	.NET Core SDK 版本。
全部	遥测配置文件：一个可选值，仅在用户显式选择加入时可用，并在 Microsoft 内部使用。
>=2.0	命令参数和选项：收集若干参数和选项(非任意字符串)。请参阅 收集的选项 。从 2.1.300 后进行哈希处理。
>=2.0	SDK 是否在容器中运行。
>=2.0	目标框架(来自 <code>TargetFramework</code> 事件)，从 2.1 开始进行哈希处理。
>=2.0	经过哈希处理的媒体访问控制 (MAC) 地址：计算机的加密 (SHA256) 匿名唯一 ID。
>=2.0	经过哈希处理的当前工作目录。
>=2.0	安装成功报告，包含进行了哈希处理的安装程序 exe 文件名。
>=2.1.300	内核版本。
>=2.1.300	Libc 发行/版本。
>=3.0.100	是否已重定向输出(true 或 false)。
>=3.0.100	CLI/SDK 故障时的异常类型及其堆栈跟踪(发送的堆栈跟踪中仅包含 CLI/SDK 代码)。有关详细信息，请参阅 收集的 .NET Core CLI/SDK 故障异常遥测 。

收集的选项

某些命令发送其他数据。小部分命令发送第一个参数：

命令	遥测数据
<code>dotnet help <arg></code>	正在查询命令帮助。

<code>dotnet new <arg></code>	模板名称(进行哈希处理)。
<code>dotnet add <arg></code>	单词 <code>package</code> 或 <code>reference</code> 。
<code>dotnet remove <arg></code>	单词 <code>package</code> 或 <code>reference</code> 。
<code>dotnet list <arg></code>	单词 <code>package</code> 或 <code>reference</code> 。
<code>dotnet sln <arg></code>	单词 <code>add</code> 、 <code>list</code> 或 <code>remove</code> 。
<code>dotnet nuget <arg></code>	单词 <code>delete</code> 、 <code>locals</code> 或 <code>push</code> 。

一小部分命令发送所选项目(如果使用)及其值:

<code>--verbosity</code>	所有命令
<code>--language</code>	<code>dotnet new</code>
<code>--configuration</code>	<code>dotnet build</code> 、 <code>dotnet clean</code> 、 <code>dotnet publish</code> 、 <code>dotnet run</code> 、 <code>dotnet test</code>
<code>--framework</code>	<code>dotnet build</code> 、 <code>dotnet clean</code> 、 <code>dotnet publish</code> 、 <code>dotnet run</code> 、 <code>dotnet test</code> 、 <code>dotnet vstest</code>
<code>--runtime</code>	<code>dotnet build</code> 、 <code>dotnet publish</code>
<code>--platform</code>	<code>dotnet vstest</code>
<code>--logger</code>	<code>dotnet vstest</code>
<code>--sdk-package-version</code>	<code>dotnet migrate</code>

除 `--verbosity` 和 `--sdk-package-version` 外, 从 .NET Core 2.1.100 SDK 开始, 所有其他值都会进行哈希处理。

收集的 .NET Core CLI/SDK 故障异常遥测

如果 .NET Core CLI/SDK 故障, 则会收集 CLI/SDK 代码的异常和跟踪堆栈名称。收集此信息是为了评估问题并改善 .NET Core SDK 和 CLI 的质量。本文提供了所收集数据的信息。本文还提供了有关生成自己的 .NET Core SDK 版本的用户如何避免无意泄露个人或敏感信息的提示。

收集的数据类型

.NET Core CLI 只收集有关 CLI/SDK 异常的信息, 不收集应用程序中的异常信息。收集的数据包含异常和堆栈跟踪的名称。此堆栈跟踪为 CLI/SDK 代码。

下面的示例显示所收集的数据类型:

```
System.IO.IOException
at System.ConsolePal.WindowsConsoleStream.Write(Byte[] buffer, Int32 offset, Int32 count)
at System.IO.StreamWriter.Flush(Boolean flushStream, Boolean flushEncoder)
at System.IO.StreamWriter.Write(Char[] buffer)
at System.IO.TextWriter.WriteLine()
at System.IO.TextWriter.SyncTextWriter.WriteLine()
at Microsoft.DotNet.Cli.Utils.Reporter.WriteLine()
at Microsoft.DotNet.Tools.Run.RunCommand.EnsureProjectIsBuilt()
at Microsoft.DotNet.Tools.Run.RunCommand.Execute()
at Microsoft.DotNet.Tools.Run.RunCommand.Run(String[] args)
at Microsoft.DotNet.Cli.Program.ProcessArgs(String[] args, ITelemetry telemetryClient)
at Microsoft.DotNet.Cli.Program.Main(String[] args)
```

避免意外泄露信息

.NET Core 参与者以及运行自己生成的 .NET Core SDK 版本的任何其他人都应考虑其 SDK 源代码的路径。如果在使用属于自定义调试生成或者使用自定义生成符号文件配置的 .NET Core SDK 时出现故障，则生成计算机的 SDK 源文件路径将作为堆栈跟踪的一部分收集，并且不会进行哈希处理。

因此，.NET Core SDK 的自定义生成不应位于路径名公开个人或敏感信息的目录中。

请参阅

- [.NET Core CLI 遥测 - 2019 第 2 季度数据](#)
- [遥测参考源 \(dotnet/cli 存储库\)](#)

提升的 Dotnet 命令访问权限

2020/4/13 • [Edit Online](#)

开发人员可根据软件开发最佳做法来编写需要最少权限的软件。但是，某些软件（如性能监视工具）由于操作系统规则，需要管理员权限。以下指南介绍使用 .NET Core 编写此类软件的适用方案。

可以运行以下提升的命令：

- `dotnet tool` 命令，如 [dotnet tool install](#)。
- `dotnet run --no-build`
- `dotnet-core-uninstall`

不建议运行其他提升的命令。具体而言，不建议为使用 MSBuild（例如，[dotnet restore](#)、[dotnet build](#) 和 [dotnet run](#)）的命令提升访问权限。主要问题是用户在发出 dotnet 命令后在根帐户和受限帐户之间来回切换时存在权限管理问题。受限用户可能会发现自己无法访问根用户构建的文件。有办法可以解决这种情况，但不一定要使用这些方法。

只要不在根帐户和受限帐户之间来回切换，就可以根帐户的身份运行命令。例如，Docker 容器默认以根帐户身份运行，因此它们具有此特性。

全局工具安装

以下说明展示了执行下述操作的推荐方法：安装、运行和卸载需要提升权限才能执行的 .NET Core 工具。

- [Windows](#)
- [Linux](#)
- [macOS](#)

安装工具

如果文件夹 `%ProgramFiles%\dotnet-tools` 已存在，请执行以下操作以检查“用户”组是否有写入或修改该目录的权限：

- 右键单击 `%ProgramFiles%\dotnet-tools` 文件夹并选择“属性”。随即打开“常用属性”对话框。
- 选择“安全性”选项卡。在“组或用户名”下，检查“用户”组是否具有写入或修改目录的权限。
- 如果“用户”组可以写入或修改目录，则在安装工具时使用其他目录名，而不使用 `dotnet-tools`。

要安装工具，请在提升的提示符下运行以下命令。此操作将在安装期间创建 `dotnet-tools` 文件夹。

```
dotnet tool install PACKAGEID --tool-path "%ProgramFiles%\dotnet-tools".
```

运行全局工具

选项 1 在提升的提示符中使用完整路径：

```
"%ProgramFiles%\dotnet-tools\TOOLCOMMAND"
```

选项 2 将新创建的文件夹添加到 `%Path%`。只需执行此操作一次。

```
setx Path "%Path%;%ProgramFiles%\dotnet-tools\"
```

然后使用以下命令运行：

卸载全局工具

在提升的提示符处，键入下列命令：

```
dotnet tool uninstall PACKAGEID --tool-path "%ProgramFiles%\dotnet-tools"
```

本地工具

本地工具的作用域按用户和个子目录树来限定。执行特权运行后，本地工具将受限的用户环境共享给提升的环境。在 Linux 和 macOS 中，这会导致将文件设置为仅限根用户访问。如果用户切换回受限帐户，则用户无法再访问或写入文件。因此，不建议将必须提升的工具安装为本地工具。建议使用 `--tool-path` 选项和上述全局工具指南。

开发过程中的提升

在开发过程中，可能需要提升访问权限才能测试应用程序。（例如）IoT 应用就常存在这种情况。建议在构建应用程序时不要进行提升，而是在运行时使用提升。有几种模式，如下所示：

- 使用生成的可执行文件（它提供最佳的启动性能）：

```
dotnet build  
sudo ./bin/Debug/netcoreapp3.0/APPLICATIONNAME
```

- 使用 `dotnet run` 命令与 `--no-build` 标志，以避免生成新的二进制文件：

```
dotnet build  
sudo dotnet run --no-build
```

请参阅

- [.NET Core 全局工具概述](#)

如何为 .NET Core CLI 启用 TAB 自动补全

2020/3/18 • [Edit Online](#)

本文适用于: ✓ .NET Core 2.1 SDK 及更高版本

本文介绍如何为三个 shell、PowerShell、Bash 和 zsh 配置 tab 自动补全。对于其他 shell，请参阅相关文档，了解如何配置 tab 自动补全。

设置完成后，通过在 shell 中键入 `dotnet` 命令，然后按下 Tab 即可触发 .NET Core CLI 的 tab 自动补全。当前命令行将发送到 `dotnet complete` 命令，结果将由 shell 处理。可以通过直接向 `dotnet complete` 命令发送内容来测试结果而无需启用 tab 自动补全。例如：

```
> dotnet complete "dotnet a"
add
clean
--diagnostics
migrate
pack
```

如果该命令不起作用，请确保已安装 .NET Core 2.0 SDK 或更高版本。如果已安装，但该命令仍不起作用，请确保 `dotnet` 命令解析为 .NET Core 2.0 SDK 及更高版本。使用 `dotnet --version` 命令查看当前路径解析为的 `dotnet` 的版本。有关详细信息，请参阅[选择要使用的 .NET Core 版本](#)页面。

示例

下面是 tab 自动补全提供的一些示例：

命令	结果	说明
<code>dotnet a*</code>	<code>dotnet add</code>	<code>add</code> 是第一项子命令，按字母排序。
<code>dotnet add p*</code>	<code>dotnet add --help</code>	Tab 自动补全匹配子字符串， <code>--help</code> 首先按字母顺序排列。
<code>dotnet add p**</code>	<code>dotnet add package</code>	第二次按 Tab 将显示下一条建议。
<code>dotnet add package Microsoft*</code>	<code>dotnet add package Microsoft.ApplicationInsights.Web</code>	结果按字母顺序返回。
<code>dotnet remove reference →</code>	<code>dotnet remove reference ...\\src\\OmniSharp.DotNet\\OmniSharp.DotNet.csproj</code>	Tab 自动补全是可识别的项目文件。

PowerShell

要将 tab 自动补全添加到适用于 .NET Core CLI 的 PowerShell，请创建或编辑存储在变量 `$PROFILE` 中的配置文件。有关详细信息，请参阅[如何创建配置文件](#)和[配置文件和执行策略](#)。

将以下代码添加到配置文件中：

```
# PowerShell parameter completion shim for the dotnet CLI
Register-ArgumentCompleter -Native -CommandName dotnet -ScriptBlock {
    param($commandName, $wordToComplete, $cursorPosition)
        dotnet complete --position $cursorPosition "$wordToComplete" | ForEach-Object {
            [System.Management.Automation.CompletionResult]::new($_, $_, 'ParameterValue', $_)
        }
}
```

bash

要将 tab 自动补全添加到适用于 .NET Core CLI 的 bash shell, 请将以下代码添加到 `.bashrc` 文件:

```
# bash parameter completion for the dotnet CLI

_dotnet_bash_complete()
{
    local word=${COMP_WORDS[COMP_CWORD]}

    local completions
    completions=$(dotnet complete --position "${COMP_POINT}" "${COMP_LINE}" 2>/dev/null)
    if [ $? -ne 0 ]; then
        completions=""
    fi

    COMPREPLY=( $(compgen -W "$completions" -- "$word") )
}

complete -f -F _dotnet_bash_complete dotnet
```

zsh

要将 tab 自动补全添加到适用于 .NET Core CLI 的 zsh shell, 请将以下代码添加到 `.zshrc` 文件:

```
# zsh parameter completion for the dotnet CLI

_dotnet_zsh_complete()
{
    local completions=$(dotnet complete "$words")

    reply=( "${(ps:\n:)completions}" )
}

compctl -K _dotnet_zsh_complete dotnet
```

.NET Core 中提供哪些诊断工具？

2020/3/18 • [Edit Online](#)

软件并非始终按预计方式运行，但 .NET Core 具有可帮助用户快速有效地诊断这些问题的工具和 API。

本文可帮助用户查找各种所需的工具。

托管调试器

借助[托管调试器](#)，用户可以与程序进行交互。暂停、增量执行、检查和恢复操作可让用户深入了解代码的行为。调试器是诊断易于重现的功能问题的首选。

日志记录和跟踪

[日志记录和跟踪](#)是相关技术。它们指的是用于创建日志文件的检测代码。这些文件记录了程序操作的详细信息。这些细节可用于诊断最复杂的问题。当与时间戳结合使用时，这些技术在性能调查中也非常有用。

单元测试

[单元测试](#)是持续集成和部署高质量软件的关键组件。单元测试的目的在于，在用户操作导致系统出现问题时提前向其发出警告。

.NET Core dotnet 诊断全局工具

dotnet-counters

[dotnet-counters](#) 是一个性能监视工具，用于初级运行状况监视和性能调查。它通过 [EventCounter API](#) 观察已发布的性能计数器值。例如，可以快速监视 CPU 使用情况或 .NET Core 应用程序中的异常率等指标。

dotnet-dump

通过 [dotnet-dump](#) 工具，可在不使用本机调试器的情况下收集和分析 Windows 和 Linux 核心转储。

dotnet-trace

分析数据通过 .NET Core 中的 [EventPipe](#) 公开。通过 [dotnet-trace](#) 工具，可以使用来自应用的有意思的分析数据，这些数据可帮助你分析应用运行缓慢的根本原因。

.NET Core 诊断教程

调试内存泄露

[教程：调试内存泄漏](#)演示了如何查找内存泄漏。[dotnet-counters](#) 工具用于确认泄露，[dotnet-dump](#) 工具用于诊断泄露。

.NET Core 托管调试器

2020/3/18 • [Edit Online](#)

调试器允许逐步骤暂停或执行程序。暂停后，可以查看进程的当前状态。通过逐步完成关键部分，你可以了解代码以及产生这一结果的原因。

Microsoft 在 Visual Studio 和 Visual Studio Code 中提供托管代码的调试器。

Visual Studio 托管调试器

Visual Studio 是一个集成开发环境，其中提供了最全面的调试器。Visual Studio 是 Windows 开发人员的最佳选择。

- [教程 - 使用 Visual Studio 在 Windows 上调试 .NET Core 应用程序](#)

尽管 Visual Studio 是一个 Windows 应用程序，但仍可用于远程调试 Linux 和 macOS 应用。

- [使用 Visual Studio 在 Linux/OSX 上调试 .NET Core 应用程序](#)

调试 ASP.NET Core 应用需要略微不同的说明。

- [在 Visual Studio 中调试 ASP.NET Core 应用](#)

Visual Studio Code 托管调试器

Visual Studio Code 是轻量型跨平台代码编辑器。它使用与 Visual Studio 相同的 .NET Core 调试器实现，但使用的是简化的用户界面。

- [教程 - 使用 Visual Studio Code 调试 .NET Core 应用程序](#)
- [在 Visual Studio Code 中进行调试](#)

.NET Core 日志记录和跟踪

2020/3/18 • [Edit Online](#)

日志记录和跟踪实际上是同一技术的两个名称。这种简单的技术从计算机早期就开始使用了。它只涉及检测应用程序以写入稍后要使用的输出。

使用日志记录和跟踪的原因

这一简单技术功能非常强大。可以在调试器失败的情况下使用它：

- 很难使用传统调试器来调试在很长一段时间内发生的问题。借助日志，可以跨在较长时间进行详细的事后检查。与此相反，调试器限制为只能进行实时分析。
- 多线程应用程序和分布式应用程序通常难以调试。附加调试器往往会影响行为。可以根据需要分析详细日志，以了解复杂的系统。
- 分布式应用程序中的问题可能源于许多组件之间的复杂交互，将调试器连接到系统的每个部分可能是不合理的。
- 许多服务不应停止。附加调试器往往会导致超时失败。
- 问题并非总是可预见的。日志记录和跟踪旨在降低开销，以便在出现问题的情况下可以始终记录程序。

.NET Core API

打印样式 API

每个 [System.Console](#)、[System.Diagnostics.Trace](#) 和 [System.Diagnostics.Debug](#) 类都提供类似的打印样式 API，以便于日志记录。

选择使用哪种打印样式 API 由用户自己决定。主要区别包括：

- [System.Console](#)
 - 始终启用，并始终写入控制台。
 - 这对于客户可能需要在版本中查看的信息非常有用。
 - 由于这是最简单的方法，所以常常用于临时调试。此调试代码通常不会签入到源代码管理中。
- [System.Diagnostics.Trace](#)
 - 仅在定义 `TRACE` 时启用。
 - 写入到附加 `Listeners`，默认情况下为 `DefaultTraceListener`。
 - 创建将在大多数生成中启用的日志时使用此 API。
- [System.Diagnostics.Debug](#)
 - 仅在定义 `DEBUG` 时启用。
 - 写入附加调试器。
 - 在 `*nix` 写入到 `stderr` 时（如果设置了 `COMPlus_DebugWriteToStdErr`）。
 - 创建将仅在调试生成中启用的日志时使用此 API。

记录事件

以下 API 更面向于事件。它们记录事件对象，而不是记录简单字符串。

- [System.Diagnostics.Tracing.EventSource](#)
 - `EventSource` 是主根 .NET Core 跟踪 API。
 - 在所有 .NET Standard 版本中提供。
 - 仅允许跟踪可序列化的对象。

- 写入到附加的[事件监听器](#)。
- .NET Core 为以下对象提供监听器：
 - 所有平台上的.NET Core EventPipe
 - [Windows 事件跟踪 \(ETW\)](#)
 - [适用于 Linux 的 LTTng 跟踪框架](#)
- [System.Diagnostics.DiagnosticSource](#)
 - 包含在.NET Core 中, 用作.NET Framework 的 [NuGet 包](#)。
 - 允许对非序列化对象进行进程内跟踪。
 - 包括一个桥, 以允许将已记录对象的所选字段写入 [EventSource](#)。
- [System.Diagnostics.Activity](#)
 - 提供了一种明确的方法来标识由特定活动或事务生成的日志消息。此对象可用于关联不同服务中的日志。
- [System.Diagnostics.EventLog](#)
 - 仅限 Windows。
 - 将消息写入 Windows 事件日志。
 - 系统管理员希望严重的应用程序错误消息会在 Windows 事件日志中出现。

ILogger 和日志记录框架

低级别 API 可能不符合你的日志记录需求。可能需要考虑使用日志记录框架。

[ILogger](#) 接口已用于创建一个公共日志记录接口, 可在其中通过依赖项注入插入记录器。

例如, 为了使你能够为应用程序做出最佳选择, [ASP.NET](#) 为选择的内置和第三方框架提供支持：

- [ASP.NET 内置日志记录提供程序](#)
- [ASP.NET 第三方日志记录提供程序](#)

与日志记录相关的引用

- [如何: 使用跟踪和调试进行条件编译](#)
- [如何: 向应用程序代码添加跟踪语句](#)
- [ASP.NET 日志记录](#) 提供它所支持的日志记录技术的概述。
- [C# 字符串内插](#) 可以简化日志记录代码的编写。
- [Exception.Message](#) 属性对日志记录异常很有用。
- [System.Diagnostics.StackTrace](#) 类可用于在日志中提供堆栈信息。

性能注意事项

字符串格式设置可能会占用大量的 CPU 处理时间。

在性能关键应用程序中, 建议执行以下操作:

- 如果未侦听, 则避免进行大量日志记录。通过检查是否首先启用了日志记录, 避免构造开销较大的日志记录消息。
- 仅记录有用的内容。
- 将复杂的格式设置推迟到分析阶段。

dotnet-counters

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 3.0 SDK 及更高版本

安装 dotnet-counters

若要安装最新版 `dotnet-counters` NuGet 包，请使用 `dotnet tool install` 命令：

```
dotnet tool install --global dotnet-counters
```

摘要

```
dotnet-counters [-h|--help] [--version] <command>
```

描述

`dotnet-counters` 是一个性能监视工具，用于临时运行状况监视和初级性能调查。它可以观察通过 [EventCounter API](#) 发布的性能计数器值。例如，可以快速监视 CPU 使用情况或 .NET Core 应用程序中引发的异常率，以了解在使用 `PerfView` 或 `dotnet-trace` 深入调查更严重的性能问题之前是否有任何可疑操作。

选项

- `--version`

显示 `dotnet-counters` 实用工具的版本。

- `-h|--help`

显示命令行帮助。

命令

“

[dotnet-counters collect](#)

[dotnet-counters list](#)

[dotnet-counters monitor](#)

[dotnet-counters ps](#)

dotnet-counters collect

定期收集所选计数器的值，并将它们导出为指定的文件格式以进行后续处理。

摘要

```
dotnet-counters collect [-h|--help] [-p|--process-id] [--refreshInterval] [counter_list] [--format] [-o|--output]
```

选项

- `-p|--process-id <PID>`

要监视的进程的 ID。

- `--refresh-interval <SECONDS>`

更新显示的计数器之间延迟的秒数

- `counter_list <COUNTERS>`

计数器的空格分隔列表。计数器可以指定为 `provider_name[:counter_name]`。如果在没有符合条件的 `counter_name` 的情况下使用 `provider_name`，则会显示所有计数器。若要发现提供程序和计数器名称，请使用 [dotnet-counters list](#) 命令。

- `--format <csv|json>`

要导出的格式。当前可用的格式：csv 和 json。

- `-o|--output <output>`

输出文件的名称。

示例

- 以 3 秒的刷新间隔时间收集所有计数器的值，并生成 csv 输出文件：

```
> dotnet-counters collect --process-id 1902 --refresh-interval 3 --format csv

counter_list is unspecified. Monitoring all counters by default.
Starting a counter session. Press Q to quit.
```

dotnet-counters list

显示按提供程序分组的计数器名称和说明的列表。

摘要

```
dotnet-counters list [-h|--help]
```

示例

```
> dotnet-counters list

Showing well-known counters only. Specific processes may support additional counters.
System.Runtime
    cpu-usage                Amount of time the process has utilized the CPU (ms)
    working-set               Amount of working set used by the process (MB)
    gc-heap-size              Total heap size reported by the GC (MB)
    gen-0-gc-count            Number of Gen 0 GCs / sec
    gen-1-gc-count            Number of Gen 1 GCs / sec
    gen-2-gc-count            Number of Gen 2 GCs / sec
    exception-count          Number of Exceptions / sec
```

dotnet-counters monitor

显示所选计数器的定期刷新值。

摘要

```
dotnet-counters monitor [-h|--help] [-p|--process-id] [--refreshInterval] [counter_list]
```

选项

- `-p|--process-id <PID>`

要监视的进程的 ID。

- `--refresh-interval <SECONDS>`

更新显示的计数器之间延迟的秒数

- `counter_list <COUNTERS>`

计数器的空格分隔列表。计数器可以指定为 `provider_name[:counter_name]`。如果在没有符合条件的 `counter_name` 的情况下使用 `provider_name`，则会显示所有计数器。若要发现提供程序和计数器名称，请使用 [dotnet-counters list](#) 命令。

示例

- 以 3 秒的刷新间隔监视 `System.Runtime` 中的所有计数器：

```
> dotnet-counters monitor --process-id 1902 --refresh-interval 3 System.Runtime

Press p to pause, r to resume, q to quit.

System.Runtime:
  CPU Usage (%)          24
  Working Set (MB)       1982
  GC Heap Size (MB)      811
  Gen 0 GC / second      20
  Gen 1 GC / second      4
  Gen 2 GC / second      1
  Number of Exceptions / sec  4
```

- 仅监视 `System.Runtime` 中的 CPU 使用情况和 GC 堆大小：

```
> dotnet-counters monitor --process-id 1902 System.Runtime[cpu-usage,gc-heap-size]

Press p to pause, r to resume, q to quit.

System.Runtime:
  CPU Usage (%)          24
  GC Heap Size (MB)      811
```

- 监视用户定义的 `EventSource` 中的 `EventCounter` 值。有关详细信息，请参阅[教程：如何使用 EventCounters 度量操作非常频繁事件的性能](#)。

```
> dotnet-counters monitor --process-id 1902 Samples-EventCounterDemos-Minimal

Press p to pause, r to resume, q to quit.

request                  100
```

dotnet-counters ps

显示可监视的 dotnet 进程的列表。

摘要

```
dotnet-counters ps [-h|--help]
```

示例

```
> dotnet-counters ps

15683 WebApi      /home/suwhang/repos/WebApi/WebApi
16324 dotnet      /usr/local/share/dotnet/dotnet
```

转储收集和分析实用工具 (dotnet-dump)

2020/3/18 • [Edit Online](#)

本文适用于: ✓ .NET Core 3.0 SDK 及更高版本

NOTE

macOS 上不支持 `dotnet-dump`。

安装 `dotnet-dump`

若要安装最新版 `dotnet-dump` NuGet 包, 请使用 `dotnet tool install` 命令:

```
dotnet tool install -g dotnet-dump
```

摘要

```
dotnet-dump [-h|--help] [--version] <command>
```

描述

`dotnet-dump` 全局工具是在未涉及任何本机调试器(如 Linux 上的 `lldb`)的情况下收集和分析 Windows 和 Linux 转储的方法。在 `lldb` 无法正常运行的平台(如 Alpine Linux)上, 此工具非常重要。借助 `dotnet-dump` 工具, 可以运行 SOS 命令来分析崩溃和垃圾回收器 (GC), 但它不是本机调试器, 因此不支持显示本机堆栈帧之类的操作。

选项

- `--version`

显示 `dotnet-dump` 实用工具的版本。

- `-h| --help`

显示命令行帮助。

命令

II

[dotnet-dump collect](#)

[dotnet-dump analyze](#)

dotnet-dump collect

从进程捕获转储。

摘要

```
dotnet-dump collect [-h|--help] [-p|--process-id] [--type] [-o|--output] [--diag]
```

选项

- `-h|--help`

显示命令行帮助。

- `-p|--process-id <PID>`

指定从中收集内存转储的进程的 ID 号。

- `--type <Heap|Mini>`

指定转储类型，它确定从进程收集的信息的类型。分为两种类型：

- `Heap` - 大型且相对全面的转储，其中包含模块列表、线程列表、所有堆栈、异常信息、句柄信息和除映射图像以外的所有内存。
- `Mini` - 小型转储，其中包含模块列表、线程列表、异常信息和所有堆栈。

如果未指定，则 `Heap` 为默认类型。

- `-o|--output <output_dump_path>`

应在其中写入收集的转储的完整路径和文件名。

如果未指定：

- 在 Windows 上默认为 `\dump_YYYYMMDD_HHMMSS.dmp`。
 - 在 Linux 上默认为 `/core_YYYYMMDD_HHMMSS`。
- YYYYMMDD 为年/月/日，HHMMSS 为小时/分钟/秒。

- `--diag`

启用转储收集诊断日志记录。

dotnet-dump analyze

启动交互式 shell 以了解转储。shell 接受各种 SOS 命令。

摘要

```
dotnet-dump analyze <dump_path> [-h|--help] [-c|--command]
```

自变量

- `<dump_path>`

指定要分析的转储文件的路径。

选项

- `-c|--command <debug_command>`

指定要在启动时在 shell 中运行的命令。

分析 SOS 命令

<code>soshelp</code>	显示所有可用命令。
<code>soshelp help <command></code>	执行指定的命令。
<code>exit quit</code>	退出交互模式。
<code>clrstack <arguments></code>	仅提供托管代码的堆栈跟踪。
<code>clrthreads <arguments></code>	列出正在运行的托管线程。
<code>dumpasync <arguments></code>	显示有关垃圾回收堆上异步状态机的信息。
<code>dumpassembly <arguments></code>	显示有关程序集的详细信息。
<code>dumpclass <arguments></code>	显示有关指定地址处的 EE 类结构的信息。
<code>dumpdelegate <arguments></code>	显示有关委托的信息。
<code>dumpdomain <arguments></code>	显示所有 AppDomain 和域中的所有程序集的信息。
<code>dumpheap <arguments></code>	显示有关垃圾回收堆的信息和有关对象的收集统计信息。
<code>dumpil <arguments></code>	显示与托管方法关联的 Microsoft 中间语言 (MSIL)。
<code>dumplog <arguments></code>	将内存中压力日志的内容写入到指定文件。
<code>dumpmd <arguments></code>	显示有关指定地址处的 MethodDesc 结构的信息。
<code>dumpmodule <arguments></code>	显示有关指定地址处的 EE 模块结构的信息。
<code>dumpmt <arguments></code>	显示有关指定地址处的方法表的信息。
<code>dumpobj <arguments></code>	显示有关指定地址处的对象的信息。
<code>dso dumpstackobjects <arguments></code>	显示在当前堆栈的边界内找到的所有托管对象。
<code>eeheap <arguments></code>	显示有关内部运行时数据结构所使用的进程内存的信息。
<code>finalizequeue <arguments></code>	显示所有已进行终结注册的对象。
<code>gcroot <arguments></code>	显示有关对指定地址处的对象的引用(或根)的信息。
<code>gcwhere <arguments></code>	显示传入参数在 GC 堆中的位置。
<code>ip2md <arguments></code>	显示 JIT 代码中指定地址处的 MethodDesc 结构。
<code>histclear <arguments></code>	释放由 <code>hist*</code> 命令系列使用的任何资源。

<code>histinit <arguments></code>	从保存在调试对象中的压力日志初始化 SOS 结构。
<code>histobj <arguments></code>	显示与 <code><arguments></code> 相关的垃圾回收压力日志重定位。
<code>histobjfind <arguments></code>	显示在指定地址处引用对象的所有日志项。
<code>histroot <arguments></code>	显示与指定根的提升和重定位相关的信息。
<code>lm modules</code>	显示进程中的本机模块。
<code>name2ee <arguments></code>	显示 <code><argument></code> 的 MethodTable 结构和 EEClass 结构。
<code>pe printexception <arguments></code>	显示从地址 <code><argument></code> 处的 Exception 类派生的任何对象。
<code>setsymbolserver <arguments></code>	启用符号服务器支持
<code>syncblk <arguments></code>	显示 SyncBlock 持有者信息。
<code>threads setthread <threadid></code>	设置或显示 SOS 命令的当前线程 ID。

使用 `dotnet-dump`

第一步是收集转储。如果已生成核心转储，则可以跳过此步骤。操作系统或 .NET Core 运行时的内置[转储生成功能](#)均可以创建核心转储。

```
$ dotnet-dump collect --process-id 1902
Writing minidump to file ./core_20190226_135837
Written 98983936 bytes (24166 pages) to core file
Complete
```

现在，使用 `analyze` 命令分析核心转储：

```
$ dotnet-dump analyze ./core_20190226_135850
Loading core dump: ./core_20190226_135850
Ready to process analysis commands. Type 'help' to list available commands or 'help [command]' to get
detailed help on a command.
Type 'quit' or 'exit' to exit the session.
>
```

此操作会显示一个交互式会话，该会话接受以下类似命令：

```

> clrstack
OS Thread Id: 0x573d (0)
    Child SP          IP Call Site
00007FFD28B42C58 00007fb22c1a8ed9 [HelperMethodFrame_PROTECTOBJ: 00007ffd28b42c58]
System.RuntimeMethodHandle.InvokeMethod(System.Object, System.Object[], System.Signature, Boolean, Boolean)
00007FFD28B42DD0 00007FB1B1334F67 System.Reflection.RuntimeMethodInfo.Invoke(System.Object,
System.Reflection.BindingFlags, System.Reflection.Binder, System.Object[], System.Globalization.CultureInfo)
[/root/coreclr/src/mscorlib/src/System/Reflection/RuntimeMethodInfo.cs @ 472]
00007FFD28B42E20 00007FB1B18D33ED SymbolTestApp.Program.Foo4(System.String)
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 54]
00007FFD28B42ED0 00007FB1B18D2FC4 SymbolTestApp.Program.Foo2(Int32, System.String)
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 29]
00007FFD28B42F00 00007FB1B18D2F5A SymbolTestApp.Program.Foo1(Int32, System.String)
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 24]
00007FFD28B42F30 00007FB1B18D168E SymbolTestApp.Program.Main(System.String[])
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 19]
00007FFD28B43210 00007fb22aa9cedf [GCFrame: 00007ffd28b43210]
00007FFD28B43610 00007fb22aa9cedf [GCFrame: 00007ffd28b43610]

```

查看已终止应用的未经处理的异常：

```

> pe -lines
Exception object: 00007fb18c038590
Exception type:  System.Reflection.TargetInvocationException
Message:         Exception has been thrown by the target of an invocation.
InnerException:   System.Exception, Use !PrintException 00007FB18C038368 to see more.
StackTrace (generated):
SP          IP          Function
00007FFD28B42DD0 0000000000000000
System.Private.CoreLib.dll!System.RuntimeMethodHandle.InvokeMethod(System.Object, System.Object[],
System.Signature, Boolean, Boolean)
00007FFD28B42DD0 00007FB1B1334F67
System.Private.CoreLib.dll!System.Reflection.RuntimeMethodInfo.Invoke(System.Object,
System.Reflection.BindingFlags, System.Reflection.Binder, System.Object[],
System.Globalization.CultureInfo)+0xa7
[/root/coreclr/src/mscorlib/src/System/Reflection/RuntimeMethodInfo.cs @ 472]
00007FFD28B42E20 00007FB1B18D33ED SymbolTestApp.dll!SymbolTestApp.Program.Foo4(System.String)+0x15d
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 54]
00007FFD28B42ED0 00007FB1B18D2FC4 SymbolTestApp.dll!SymbolTestApp.Program.Foo2(Int32, System.String)+0x34
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 29]
00007FFD28B42F00 00007FB1B18D2F5A SymbolTestApp.dll!SymbolTestApp.Program.Foo1(Int32, System.String)+0x3a
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 24]
00007FFD28B42F30 00007FB1B18D168E SymbolTestApp.dll!SymbolTestApp.Program.Main(System.String[])+0x6e
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 19]

StackTraceString: <none>
HRESULT: 80131604

```

Docker 的特殊说明

如果在 Docker 下运行，则转储集合需要 `SYS_PTRACE` 功能 (`--cap-add=SYS_PTRACE` 或 `--privileged`)。

在 Microsoft .NET Core SDK Linux Docker 映像上，某些 `dotnet-dump` 命令可能会引发以下异常：

未经处理的异常: System.DllNotFoundException:无法加载共享库“libdl.so”或其依赖项之一的异常。

若要解决此问题，请安装“libc6-dev”包。

dotnet-trace 性能分析实用工具

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 3.0 SDK 及更高版本

安装 dotnet-trace

使用 `dotnet tool install` 命令安装 `dotnet-trace` NuGet 包：

```
dotnet tool install --global dotnet-trace
```

摘要

```
dotnet-trace [-h, --help] [--version] <command>
```

描述

`dotnet-trace` 工具：

- 是一个跨平台的 .NET Core 工具。
- 在不使用本机探查器的情况下启用正在运行的进程的 .NET Core 跟踪集合。
- 它是围绕 .NET Core 运行时的跨平台 `EventPipe` 技术而构建的。
- 在 Windows、Linux 或 macOS 上提供相同体验。

选项

- `--version`

显示 `dotnet-trace` 实用工具的版本。

- `-h|--help`

显示命令行帮助。

命令

ff

[dotnet-trace collect](#)

[dotnet-trace convert](#)

[dotnet-trace ps](#)

[dotnet-trace list-profiles](#)

dotnet-trace collect

从正在运行的进程中收集诊断跟踪。

摘要

```
dotnet-trace collect [-h|--help] [-p|--process-id] [--buffersize <size>] [-o|--output]
[--providers] [--profile <profile-name>] [--format]
```

选项

- `-p|--process-id <PID>`

从中收集跟踪的进程。

- `--buffersize <size>`

设置内存中循环缓冲区的大小(以 MB 表示)。默认值为 256 MB。

- `-o|--output <trace-file-path>`

收集的跟踪数据的输出路径。如果未指定，则默认为 `trace.nettrace`。

- `--providers <list-of-commma-separated-providers>`

要启用的 `EventPipe` 提供程序的以逗号分隔的列表。这些提供程序会补充 `--profile <profile-name>` 隐含的任何提供程序。如果特定提供程序存在任何不一致的情况，此配置将优先于配置文件中的隐式配置。

此提供程序列表的格式为：

- `Provider[,Provider]`
- `Provider` 的格式为：`KnownProviderName[:Flags[:Level][:KeyValueArgs]]`。
- `KeyValueArgs` 的格式为：`[key1=value1][;key2=value2]`。

- `--profile <profile-name>`

一组命名的预定义提供程序配置，允许简明地指定常见跟踪方案。

- `--format {NetTrace|Speedscope}`

设置跟踪文件转换的输出格式。默认值为 `NetTrace`。

dotnet-trace convert

将 `nettrace` 跟踪转换为备用格式，以便用于备用跟踪分析工具。

摘要

```
dotnet-trace convert [<input-filename>] [-h|--help] [--format] [-o|--output]
```

自变量

- `<input-filename>`

要转换的输入跟踪文件。默认为 `trace.nettrace`。

选项

- `--format <NetTrace|Speedscope>`

设置跟踪文件转换的输出格式。

- `-o|--output <output-filename>`

输出文件名。将添加目标格式的扩展。

dotnet-trace ps

列出可附加到的 dotnet 进程。

摘要

```
dotnet-trace ps [-h|--help]
```

dotnet-trace list-profiles

列出预生成的跟踪配置文件，并描述每个配置文件中包含的提供程序和筛选器。

摘要

```
dotnet-trace list-profiles [-h|--help]
```

使用 dotnet-trace 收集跟踪

若要使用 `dotnet-trace` 收集跟踪，请执行以下操作：

- 需要首先查找要从中收集跟踪的 .NET Core 应用程序的进程标识符 (PID)。
 - 例如，在 Windows 上，可以使用任务管理器或 `tasklist` 命令。
 - 在 Linux 上，使用 `ps` 命令。
 - `dotnet-trace ps`
- 运行下面的命令：

```
dotnet-trace collect --process-id <PID>
```

前面的命令生成类似于以下内容的输出：

```
Press <Enter> to exit...
Connecting to process: <Full-Path-To-Process-Being-Profiled>/dotnet.exe
Collecting to file: <Full-Path-To-Trace>/trace.nettrace
Session Id: <SessionId>
Recording trace 721.025 (KB)
```

- 按 `<Enter>` 键停止收集。`dotnet-trace` 会将完成将事件记录到 `trace.nettrace` 文件中。

查看由 dotnet-trace 捕获的跟踪

在 Windows 上，可以使用 `PerfView` 查看 `.nettrace` 文件以进行分析；对于其他平台上收集的跟踪，可以将跟踪文件移动到 Windows 计算机上，以在 `PerfView` 上进行查看。

在 Linux 上，可以通过将 `dotnet-trace` 的输出格式更改为 `speedscope` 来查看跟踪。可以使用 `-f|--format` 选项更改输出文件格式 - `-f speedscope` 会使 `dotnet-trace` 生成 `speedscope` 文件。可以在 `nettrace` (默认选项) 和 `speedscope` 之间进行选择。可以在 <https://www.speedscope.app> 打开 `Speedscope` 文件。

NOTE

.NET Core 运行时以 `nettrace` 格式生成跟踪。跟踪完成后，跟踪将转换为 `speedscope`(如果指定)。由于某些转换可能会导致数据丢失，因此，原始 `nettrace` 文件将保留在转换后的文件旁边。

使用 dotnet-trace 收集随时间变化的计数器值

`dotnet-trace` 可以：

- 在性能敏感的环境中使用 `EventCounter` 进行基本运行状况监视。例如，在生产环境中。
- 收集跟踪，这样就不需要实时查看。

例如，若要收集运行时性能计数器值，请使用以下命令：

```
dotnet-trace collect --process-id <PID> --providers System.Runtime:0:1:EventCounterIntervalSec=1
```

上述命令通知运行时计数器每秒报告一次，以便进行轻量型运行状况监视。通过使用较大的值(例如 60)替换 `EventCounterIntervalSec=1`，可以收集计数器数据中粒度较小的跟踪。

以下命令比以上命令产生更小的开销和跟踪大小：

```
dotnet-trace collect --process-id <PID> --providers System.Runtime:0:1:EventCounterIntervalSec=1,Microsoft-Windows-DotNETRuntime:0:1,Microsoft-DotNETCore-SampleProfiler:0:1
```

以上命令会禁用运行时事件和托管堆栈探查器。

.NET 提供程序

.NET Core 运行时支持以下 .NET 提供程序：.NET Core 使用相同的关键字来启用 `Event Tracing for Windows (ETW)` 和 `EventPipe` 跟踪。

提供程序	关键字
<code>Microsoft-Windows-DotNETRuntime</code>	<code>运行时提供程序</code> <code>CLR 运行时关键字</code>
<code>Microsoft-Windows-DotNETRuntimeRundown</code>	<code>断开提供程序</code> <code>CLR 断开关键字</code>
<code>Microsoft-DotNETCore-SampleProfiler</code>	启用示例探查器。

教程：调试 .NET Core 中的内存泄漏

2020/3/18 • [Edit Online](#)

本文适用于：✓ .NET Core 3.0 SDK 及更高版本

本教程演示用于分析 .NET Core 内存泄漏的工具。

本教程使用一个示例应用程序，它设计为有意泄漏内存。本示例作为练习提供。还可以分析无意中泄漏内存的应用程序。

在本教程中，你将：

- 使用 [dotnet-counters](#) 检查托管内存的使用情况。
- 生成转储文件。
- 使用转储文件分析内存使用情况。

先决条件

本教程使用：

- [.NET Core 3.0 SDK](#) 或更高版本。
- [dotnet-trace](#) 列出进程。
- [dotnet-counters](#) 检查托管内存的使用情况。
- [dotnet-dump](#) 收集和分析转储文件。
- 要诊断的[示例调试目标](#)应用。

本教程假设已安装示例和工具并可供使用。

检查托管内存的使用情况

在开始收集诊断数据以帮助分析本案例的根本原因时，需要确保实际看到的是内存泄漏（内存增加）。可以使用 [dotnet-counters](#) 工具进行确认。

打开控制台窗口并导航到下载并解压缩[示例调试目标](#)的目录。运行目标：

```
dotnet run
```

在单独的控制台中，使用 [dotnet-trace](#) 工具查找进程 ID：

```
dotnet-trace ps
```

输出应如下所示：

```
4807 DiagnosticScena  
/home/user/git/samples/core/diagnostics/DiagnosticScenarios/bin/Debug/netcoreapp3.0/DiagnosticScenarios
```

现使用 [dotnet-counters](#) 工具检查托管内存的使用情况。`--refresh-interval` 指定两次刷新之间的秒数：

```
dotnet-counters monitor --refresh-interval 1 -p 4807
```

实时输出应如下所示：

```
Press p to pause, r to resume, q to quit.  
Status: Running  
  
[System.Runtime]  
# of Assemblies Loaded 118  
% Time in GC (since last GC) 0  
Allocation Rate (Bytes / sec) 37,896  
CPU Usage (%) 0  
Exceptions / sec 0  
GC Heap Size (MB) 4  
Gen 0 GC / sec 0  
Gen 0 Size (B) 0  
Gen 1 GC / sec 0  
Gen 1 Size (B) 0  
Gen 2 GC / sec 0  
Gen 2 Size (B) 0  
LOH Size (B) 0  
Monitor Lock Contention Count / sec 0  
Number of Active Timers 1  
ThreadPool Completed Work Items / sec 10  
ThreadPool Queue Length 0  
ThreadPool Threads Count 1  
Working Set (MB) 83
```

重点介绍此行：

GC Heap Size (MB)	4
-------------------	---

启动后，可以看到托管堆内存为 4 MB。

现在，点击 URL <http://localhost:5000/api/diagscenario/memleak/20000>。

请注意，内存使用量已增加到 30 MB。

GC Heap Size (MB)	30
-------------------	----

通过监视内存使用情况，可以确定内存正在增长或泄漏。下一步是收集内存分析的适当数据。

生成内存转储

分析可能的内存泄漏时，需要访问应用的内存堆。然后可以分析内存内容。查看对象之间的关系，可以创建理论说明内存未释放的原因。常见的诊断数据源是 Windows 上的内存转储或 Linux 上的等效核心转储。若要生成 .NET Core 应用程序转储，可以使用 [dotnet-dump](#) 工具。

使用之前启动的示例调试目标，运行以下命令以生成 Linux 核心转储：

```
dotnet-dump collect -p 4807
```

结果是位于同一文件夹中的核心转储。

```
Writing minidump with heap to ./core_20190430_185145  
Complete
```

重新启动失败的进程

收集转储后，你应该有足够的信息来诊断失败的进程。如果失败的进程在生产服务器上运行，现在是通过重新启动进程进行短期修正的理想时机。

在本教程中，你已经完成了[示例调试目标](#)，现在可以将其关闭。导航到启动服务器的终端并按 `Control-C`。

分析核心转储

生成核心转储后，请使用 `dotnet-dump` 工具分析转储：

```
dotnet-dump analyze core_20190430_185145
```

其中 `core_20190430_185145` 是要分析的核心转储的名称。

NOTE

如果你看到报错“找不到 libdl.so ”，则可能需要安装 `libc6-dev` 包。有关详细信息，请参阅 [Linux 上 .NET Core 的先决条件](#)。

此时会显示一个提示，可在其中输入 SOS 命令。通常，首先要查看的是托管堆的整体状态：

```
> dumpheap -stat

Statistics:
      MT      Count    TotalSize Class Name
...
00007f6c1eefba8      576        59904 System.Reflection.RuntimeMethodInfo
00007f6c1dc021c8     1749       95696 System.SByte[]
00000000008c9db0    3847      116080     Free
00007f6c1e784a18     175       128640 System.Char[]
00007f6c1dbf5510    217       133504 System.Object[]
00007f6c1dc014c0    467       416464 System.Byte[]
00007f6c21625038      6       4063376 testwebapi.Controllers.Customer[]
00007f6c20a67498   200000      4800000 testwebapi.Controllers.Customer
00007f6c1dc00f90   206770      19494060 System.String
Total 428516 objects
```

可在此处看到大多数对象是 `String` 或 `Customer` 对象。

可以使用方法表 (MT) 再次使用 `dumpheap` 命令来获取所有 `String` 实例的列表：

```
> dumpheap -mt 00007faddaa50f90

      Address          MT      Size
...
00007f6ad09421f8 00007faddaa50f90        94
...
00007f6ad0965b20 00007f6c1dc00f90        80
00007f6ad0965c10 00007f6c1dc00f90        80
00007f6ad0965d00 00007f6c1dc00f90        80
00007f6ad0965d00 00007f6c1dc00f90        80
00007f6ad0965ee0 00007f6c1dc00f90        80

Statistics:
      MT      Count    TotalSize Class Name
00007f6c1dc00f90   206770      19494060 System.String
Total 206770 objects
```

现在可以对 `System.String` 实例使用 `gcroot` 命令，以查看对象的根方式和原因。请耐心等待，因为对于 30 MB 的堆，此命令需要几分钟的时间：

```

> gcroot -all 00007f6ad09421f8

Thread 3f68:
  00007F6795BB58A0 00007F6C1D7D0745 System.Diagnostics.Tracing.CounterGroup.PollForValues()
[/_/src/System.Private.CoreLib/shared/System/Diagnostics/Tracing/CounterGroup.cs @ 260]
    rbx: (interior)
      -> 00007F6BDFFFF038 System.Object[]
      -> 00007F69D0033570 testwebapi.Controllers.Processor
      -> 00007F69D0033588 testwebapi.Controllers.CustomerCache
      -> 00007F69D00335A0 System.Collections.Generic.List`1[[testwebapi.Controllers.Customer,
DiagnosticScenarios]]
        -> 00007F6C000148A0 testwebapi.Controllers.Customer[]
        -> 00007F6AD0942258 testwebapi.Controllers.Customer
        -> 00007F6AD09421F8 System.String

HandleTable:
  00007F6C98BB15F8 (pinned handle)
  -> 00007F6BDFFFF038 System.Object[]
  -> 00007F69D0033570 testwebapi.Controllers.Processor
  -> 00007F69D0033588 testwebapi.Controllers.CustomerCache
  -> 00007F69D00335A0 System.Collections.Generic.List`1[[testwebapi.Controllers.Customer,
DiagnosticScenarios]]
  -> 00007F6C000148A0 testwebapi.Controllers.Customer[]
  -> 00007F6AD0942258 testwebapi.Controllers.Customer
  -> 00007F6AD09421F8 System.String

Found 2 roots.

```

可以看到 `String` 由 `Customer` 对象直接保存，并由 `CustomerCache` 对象间接保存。

可以继续转储对象，以查看大多数 `String` 对象是否遵循类似的模式。此时，调查会提供足够的信息来确定代码中的根本原因。

可通过此常规过程确定主要内存泄漏源。

清理资源

在本教程中，你已启动一个示例 Web 服务器。此服务器应已关闭，如[重新启动失败的进程](#)部分所述。

还可以删除已创建的转储文件。

后续步骤

恭喜你完成本教程。

我们还在发布更多诊断教程。可以在 [dotnet/diagnostics](#) 存储库上阅读草稿版本。

本教程介绍了重要的 .NET 诊断工具的基础知识。有关高级用法，请参阅以下参考文档：

- [dotnet-trace](#) 列出进程。
- [dotnet-counters](#) 检查托管内存的使用情况。
- [dotnet-dump](#) 收集和分析转储文件。

.NET Core 附加工具概述

2020/3/18 • [Edit Online](#)

本节除了 .NET Core CLI 外，还编译了可支持和扩展 .NET Core 功能的工具列表。

.NET Core 卸载工具

使用 [.NET Core 卸载工具](#) (`dotnet-core-uninstall`)，可清理系统上的 .NET Core SDK 和运行时，以便仅保留指定的版本。可使用选项集合来指定要卸载的版本。

.NET Core 诊断工具

[dotnet-counters](#) 是一个性能监视工具，用于初级运行状况监视和性能调查。

通过 [dotnet-dump](#)，可在不使用本机调试器的情况下收集和分析 Windows 和 Linux 核心转储。

[dotnet-trace](#) 会从你的应用收集分析数据，这些数据可帮助你了解应用运行速度缓慢的原因。

WCF Web Service Reference 工具

WCF (Windows Communication Foundation) [Web service Reference 工具](#) 是一个 Visual Studio 连接服务提供程序，首次推出是在 [Visual Studio 2017 版本 15.5](#) 中。此工具可从网络位置上当前解决方案的 Web 服务中，或从 WSDL 文件中检索元数据。还可生成与 .NET Core 兼容的源文件并使用可用于访问 Web 服务操作的方法定义 WCF 代理类。

WCF dotnet-svcutil 工具

WCF [dotnet-svcutil 工具](#) 是一个 .NET 工具，可从网络位置上的 Web 服务中或从 WSDL 文件中检索元数据。还可生成与 .NET Core 兼容的源文件并使用可用于访问 Web 服务操作的方法定义 WCF 代理类。

dotnet-svcutil 工具是 [WCF Web Service Reference](#) Visual Studio 连接服务提供程序(随 Visual Studio 2017 版本 15.5 首次推出)的替代产品。dotnet-svcutil 工具作为一种 .NET 工具，可用于 Linux、macOS 和 Windows。

WCF dotnet-svcutil.xmlserializer 工具

在 .NET Framework 中，可以使用 svcutil 工具预生成序列化程序集。WCF [dotnet-svcutil.xmlserializer 工具](#) 在 .NET Core 上提供类似的功能。它为客户端应用程序中 WCF 服务协定使用且可由 [XmlSerializer](#) 序列化的类型预生成 C# 序列化代码。当序列化或反序列化这些类型的对象时，这会提高 XML 序列化的启动性能。

XML 序列化程序生成器

正如 [XML 序列化程序生成器工具 \(Sgen.exe\)](#) 适用于 .NET Framework，[Microsoft.XmlSerializer.Generator NuGet 包](#) 是适用于 .NET Core 和 .NET 标准库的解决方案。它为程序集中包含的类型创建 XML 序列化程序集，从而提高使用 [XmlSerializer](#) 序列化或反序列化这些类型对象时，XML 序列化的启动性能。

.NET Core 卸载工具

2020/4/2 • [Edit Online](#)

.NET Core 卸载工具 (`dotnet-core-uninstall`) 使你可以从系统中删除 .NET Core SDK 和运行时。可使用选项集合来指定要卸载的版本。

该工具支持 Windows 和 macOS。目前不支持 Linux。

在 Windows 上，该工具只能卸载使用以下安装程序之一安装的 SDK 和运行时：

- .NET Core SDK 和运行时安装程序。
- Visual studio 安装程序的版本早于 Visual Studio 2019 版本 16.3。

在 macOS 上，该工具只能卸载位于 `/usr/local/share/dotnet` 文件夹中的 SDK 和运行时。

由于这些限制，该工具可能无法卸载计算机上的所有 .NET Core SDK 和运行时。可以使用 `dotnet --info` 命令来查找所有安装的 .NET Core SDK 和运行时，包括此工具无法删除的 SDK 和运行时。`dotnet-core-uninstall list` 命令显示可以通过该工具卸载的 SDK。

安装工具

可以从[此处](#)下载 .NET Core 卸载工具，然后在 [dotnet/cli-lab](#) GitHub 存储库中找到资源代码。

NOTE

此工具需要提升才能卸载 .NET Core SDK 和运行时。因此，应将其安装在写入保护的目录中，如 Windows 上的 C:\Program Files 或 macOS 上的 /usr/local/bin。另请参阅[提升的 Dotnet 命令访问权限](#)。有关详细信息，请参阅[详细安装说明](#)。

运行该工具

以下步骤说明了运行卸载工具的建议方法：

- [步骤 1 - 显示已安装的 .NET Core Sdk 和运行时](#)
- [步骤 2 - 执行试运行](#)
- [步骤 3 - 卸载 .NET Core SDK 和运行时](#)
- [步骤 4 - 删除 NuGet 回退文件夹\(可选\)](#)

步骤 1 - 显示安装的 .NET Core SDK 和运行时

`dotnet-core-uninstall list` 命令列出了已安装的 .NET Core SDK 和运行时，可以通过此工具将其删除。Visual Studio 可能需要某些 SDK 和运行时，它们将显示出来，并说明为何不建议将其卸载。

NOTE

在大多数情况下，`dotnet-core-uninstall list` 命令的输出将与 `dotnet --info` 输出中的已安装版本列表不匹配。具体而言，此工具将不会显示通过 zip 文件安装的版本，也不会显示由 Visual Studio (Visual Studio 2019 16.3 或更高版本) 托管的版本。检查某个版本是否由 Visual Studio 托管的一种方法是在 [Add or Remove Programs](#) 中查看该版本，由 Visual Studio 托管的版本在显示名称中会以这种方式标记。

`dotnet-core-uninstall list`

摘要

```
dotnet-core-uninstall list [options]
```

选项

- [Windows](#)
- [macOS](#)
- [--aspnet-runtime](#)

列出可通过此工具卸载的所有 ASP.NET Core 运行时。

- [--hosting-bundle](#)

列出可通过此工具卸载的所有 .NET Core 运行时和托管捆绑包。

- [--runtime](#)

列出可通过此工具卸载的所有 .NET Core 运行时。

- [--sdk](#)

列出可通过此工具卸载的所有 .NET Core SDK。

- [-v, --verbosity <LEVEL>](#)

设置详细程度。允许使用的值为 [q\[uiet\]](#)、[m\[inimal\]](#)、[n\[ormal\]](#)、[d\[etailed\]](#) 和 [diag\[nostic\]](#)。默认值为 [normal](#)。

- [--x64](#)

列出可通过此工具卸载的所有 x64 .NET Core SDK 和运行时。

- [--x86](#)

列出可通过此工具卸载的所有 x86 .NET Core SDK 和运行时。

示例

- 列出可通过此工具删除的所有 .NET Core SDK 和运行时：

```
dotnet-core-uninstall list
```

- 列出所有 x64 .NET Core SDK 和运行时：

```
dotnet-core-uninstall list --x64
```

- 列出所有 x86 .NET Core SDK：

```
dotnet-core-uninstall list --sdk --x86
```

步骤 2 - 执行试运行

[dotnet-core-uninstall dry-run](#) 和 [dotnet-core-uninstall whatif](#) 命令显示将根据提供的选项删除的 .NET Core SDK 和运行时，而无需执行卸载。这些命令是同义词。

dotnet-core-uninstall dry-run 和 **dotnet-core-uninstall whatif**

摘要

```
dotnet-core-uninstall dry-run [options] [<VERSION>...]
```

```
dotnet-core-uninstall whatif [options] [<VERSION>...]
```

自变量

- `VERSION`

要卸载的指定版本。可以逐一列出多个版本，用空格分隔。此外还支持响应文件。

TIP

响应文件是在命令行上放置所有版本的替代方法。它们是文本文件，通常具有 `*.rsp` 扩展名，每个版本都在单独的行上列出。若要为 `VERSION` 参数指定响应文件，请使用后面紧跟响应文件名的 `@` 字符。

选项

- `Windows`

- `macOS`

- `--all`

删除所有 .NET Core SDK 和运行时。

- `--all-below <VERSION>`

仅删除版本小于指定版本的 .NET Core SDK 和运行时。仍安装指定版本。

- `--all-but <VERSIONS>`

除了那些指定版本外，删除所有 .NET Core SDK 和运行时。

- `--all-but-latest`

删除 .NET Core SDK 和运行时(最高版本除外)。

- `--all-lower-patches`

删除由较高版本的修补程序取代的 .NET Core SDK 和运行时。此选项保护 `global.json`。

- `--all-previews`

删除标记为预览的 .NET Core SDK 和运行时。

- `--all-previews-but-latest`

删除标记为预览的 .NET Core SDK 和运行时(最高预览版除外)。

- `--aspnet-runtime`

仅删除 ASP.NET Core 运行时。

- `--hosting-bundle`

仅删除 .NET Core 运行时和托管绑定。

- `--major-minor <MAJOR_MINOR>`

删除与指定 `major.minor` 版本相匹配的 .NET Core SDK 和运行时。

- `--runtime`

仅删除 .NET Core 运行时。

- `--sdk`

仅删除 .NET Core SDK。

- `-v, --verbosity <LEVEL>`

设置详细程度。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `normal`。

- `--x64`

必须与 `--sdk`、`--runtime` 和 `--aspnet-runtime` 结合使用才能删除 x64 SDK 或运行时。

- `--x86`

必须与 `--sdk`、`--runtime` 和 `--aspnet-runtime` 结合使用才能删除 x86 SDK 或运行时。

- `--force` 强制删除可能由 Visual Studio 使用的版本。

注意：

1. 只需 `--sdk`、`--runtime`、`--aspnet-runtime` 和 `--hosting-bundle` 中的一个。
2. `--all`、`--all-below`、`--all-but`、`--all-but-latest`、`--all-lower-patches`、`--all-previews`、`--all-previews-but-latest`、`--major-minor` 和 `[<VERSION>...]` 除外。
3. 如果未指定 `--x64` 或 `--x86`，则同时删除 x64 和 x86。

示例

NOTE

默认情况下，Visual Studio 或其他 SDK 可能需要的 .NET Core SDK 和运行时不会包含在 `dotnet-core-uninstall dry-run` 输出中。在下面的示例中，某些指定的 SDK 和运行时可能不会包含在输出中，具体取决于计算机的状态。若要包括所有 SDK 和运行时，请将它们显式列出为参数或使用 `--force` 选项。

- 试运行删除已被较高版本的修补程序取代的所有 .NET Core 运行时：

```
dotnet-core-uninstall dry-run --all-lower-patches --runtime
```

- 试运行删除低于版本 `2.2.301` 的所有 .NET Core SDK：

```
dotnet-core-uninstall whatif --all-below 2.2.301 --sdk
```

步骤 3 - 卸载 .NET Core SDK 和运行时

`dotnet-core-uninstall remove` 卸载由选项集合指定的 .NET Core SDK 和运行时。该工具不能用于卸载版本 5.0 或更高版本的 SDK 和运行时。

由于此工具具有破坏性行为，因此强烈建议在运行 `remove` 命令之前执行试运行。使用 `remove` 命令时，试运行将显示要删除的 .NET Core SDK 和运行时。请参阅[是否应删除版本？](#)了解哪些 SDK 和运行时可以安全删除。

Caution

请记住以下注意事项：

- 此工具可以卸载计算机上 `global.json` 文件所需的 .NET Core SDK 版本。可以从[下载 .NET Core](#) 页面重新安装 .NET Core SDK。
- 此工具可以卸载计算机上依赖于框架的应用程序所需的 .NET Core 运行时版本。可以从[下载 .NET Core](#) 页面重新安装 .NET Core 运行时。
- 此工具可以卸载 Visual Studio 所依赖的 .NET Core SDK 和运行时版本。如果中断 Visual Studio 安装，请在

Visual Studio 安装程序中运行“修复”以返回到工作状态。

默认情况下，所有命令都将保留 Visual Studio 或其他 SDK 可能需要的 .NET Core SDK 和运行时。可以通过将这些 SDK 和运行时显式列出为参数或使用 `--force` 选项来卸载这些 SDK 和运行时。

此工具需要提升才能卸载 .NET Core SDK 和运行时。在 Windows 上的管理员命令提示符中运行此工具，在 macOS 上则通过 `sudo` 运行。`dry-run` 和 `whatif` 命令不需要提升。

dotnet-core-uninstall remove

摘要

```
dotnet-core-uninstall remove [options] [<VERSION>...]
```

自变量

- `VERSION`

要卸载的指定版本。可以逐一列出多个版本，用空格分隔。此外还支持响应文件。

TIP

响应文件是在命令行上放置所有版本的替代方法。它们是文本文件，通常具有 *.rsp 扩展名，每个版本都在单独的行上列出。若要为 `VERSION` 参数指定响应文件，请使用后面紧跟响应文件名的 @ 字符。

选项

- [Windows](#)
- [macOS](#)

- `--all`

删除所有 .NET Core SDK 和运行时。

- `--all-below <VERSION>`

仅删除版本小于指定版本的 .NET Core SDK 和运行时。仍安装指定版本。

- `--all-but <VERSIONS>`

除了那些指定版本外，删除所有 .NET Core SDK 和运行时。

- `--all-but-latest`

删除 .NET Core SDK 和运行时(最高版本除外)。

- `--all-lower-patches`

删除由较高版本的修补程序取代的 .NET Core SDK 和运行时。此选项保护 global.json。

- `--all-previews`

删除标记为预览的 .NET Core SDK 和运行时。

- `--all-previews-but-latest`

删除标记为预览的 .NET Core SDK 和运行时(最高预览版除外)。

- `--aspnet-runtime`

仅删除 ASP.NET Core 运行时。

- `--hosting-bundle`

仅删除 .NET Core 运行时和托管绑定。

- `--major-minor <MAJOR_MINOR>`

删除与指定 `major.minor` 版本相匹配的 .NET Core SDK 和运行时。

- `--runtime`

仅删除 .NET Core 运行时。

- `--sdk`

仅删除 .NET Core SDK。

- `-v, --verbosity <LEVEL>`

设置详细程度。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `normal`。

- `--x64`

必须与 `--sdk`、`--runtime` 和 `--aspnet-runtime` 结合使用才能删除 x64 SDK 或运行时。

- `--x86`

必须与 `--sdk`、`--runtime` 和 `--aspnet-runtime` 结合使用才能删除 x86 SDK 或运行时。

- `-y, --yes` 执行命令而不需要进行是或否确认。

- `--force` 强制删除可能由 Visual Studio 使用的版本。

注意：

1. 只需 `--sdk`、`--runtime`、`--aspnet-runtime` 和 `--hosting-bundle` 中的一个。
2. `--all`、`--all-below`、`--all-but`、`--all-but-latest`、`--all-lower-patches`、`--all-previews`、`--all-previews-but-latest`、`--major-minor` 和 `[<VERSION>...]` 除外。
3. 如果未指定 `--x64` 或 `--x86`，则同时删除 x64 和 x86。

示例

NOTE

默认情况下，将保留 Visual Studio 或其他 SDK 可能需要的 .NET Core SDK 和运行时。在下面的示例中，可能保留某些指定的 SDK 和运行时，具体取决于计算机的状态。若要删除所有 SDK 和运行时，请将它们显式列出为参数或使用 `--force` 选项。

- 删除除版本 `3.0.0-preview6-27804-01` 之外的所有 .NET Core 运行时，无需进行 Y/N 确认：

```
dotnet-core-uninstall remove --all-but 3.0.0-preview6-27804-01 --runtime --yes
```

- 删除所有 .NET Core 1.1 SDK，无需进行 Y/N 确认：

```
dotnet-core-uninstall remove --sdk --major-minor 1.1 -y
```

- 删除没有控制台输出的 .NET Core 1.1.11 SDK：

```
dotnet-core-uninstall remove 1.1.11 --sdk --yes --verbosity q
```

- 删除可由此工具安全删除的所有 .NET Core SDK：

```
dotnet-core-uninstall remove --all --sdk
```

- 删除此工具可删除的所有 .NET Core SDK, 包括 Visual Studio 可能需要的 SDK(不推荐)：

```
dotnet-core-uninstall remove --all --sdk --force
```

- 删除响应文件 `versions.rsp` 中指定的所有 .NET Core SDK

```
dotnet-core-uninstall remove --sdk @versions.rsp
```

`versions.rsp` 的内容如下所示：

```
2.2.300  
2.1.700
```

步骤 4 - 删 NuGet 回退文件夹(可选)

在某些情况下, 你不再需要 `NuGetFallbackFolder`, 可能希望将其删除。有关删除此文件夹的详细信息, 请参阅[删除 NuGetFallbackFolder](#)。

卸载工具

- [Windows](#)
- [macOS](#)

1. 打开“添加或删除程序”。
2. 搜索 `Microsoft .NET Core SDK Uninstall Tool`。
3. 选择“卸载”。

使用 WCF Web Service Reference Provider 工具

2020/3/18 • [Edit Online](#)

多年来，许多 Visual Studio 开发者在其 .NET Framework 项目需要访问 Web 服务时，都享受到了[添加服务引用工具所带来的工作效率](#)。WCF Web 服务引用工具是 Visual Studio 连接服务的扩展，提供了类似于 .NET Core 和 ASP.NET Core 项目的“添加服务引用”功能的体验。此工具可从网络位置的当前解决方案的 web 服务中或从 WSDL 文件中检索元数据，并生成包含可用于访问 web 服务的 Windows Communication Foundation (WCF) 客户端代理代码的可兼容 .NET Core 的源文件。

IMPORTANT

应仅从受信任源引用服务。从不受信任的源添加引用可能会危及安全性。

系统必备

- [Visual Studio 2017 版本 15.5 或更高版本](#)

如何使用扩展

NOTE

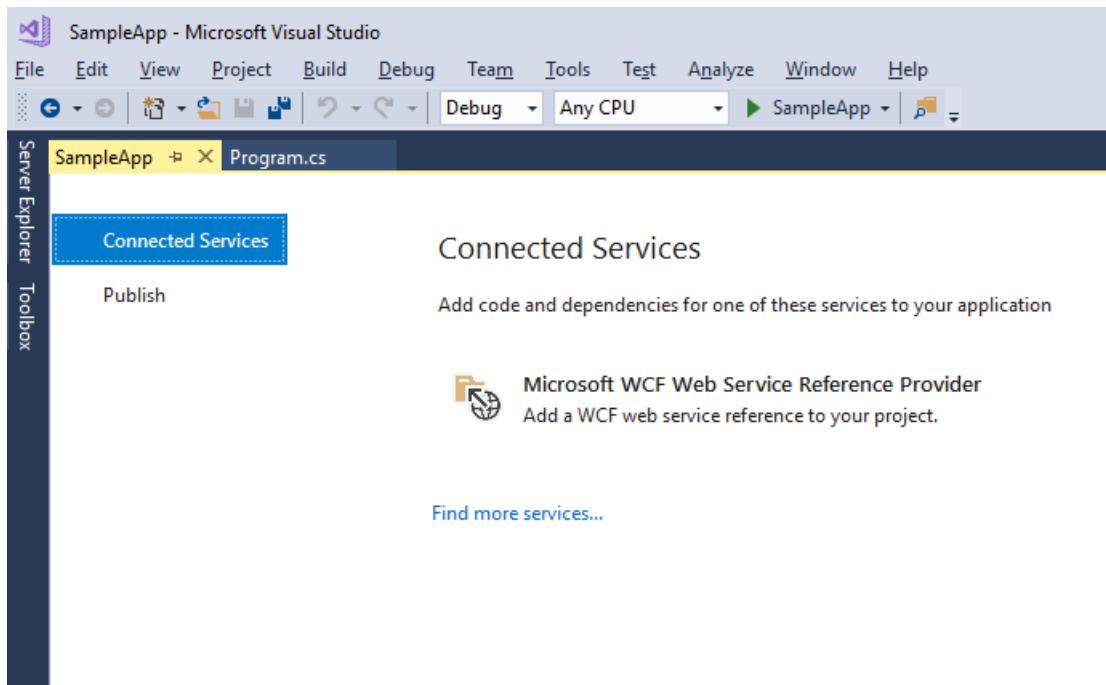
“WCF Web 服务引用”选项适用于使用以下项目模板创建的项目：

- [Visual C# .NET Core >](#)
- [Visual C# .NET Standard >](#)
- [Visual C# Web ASP.NET Core Web 应用程序 > >](#)

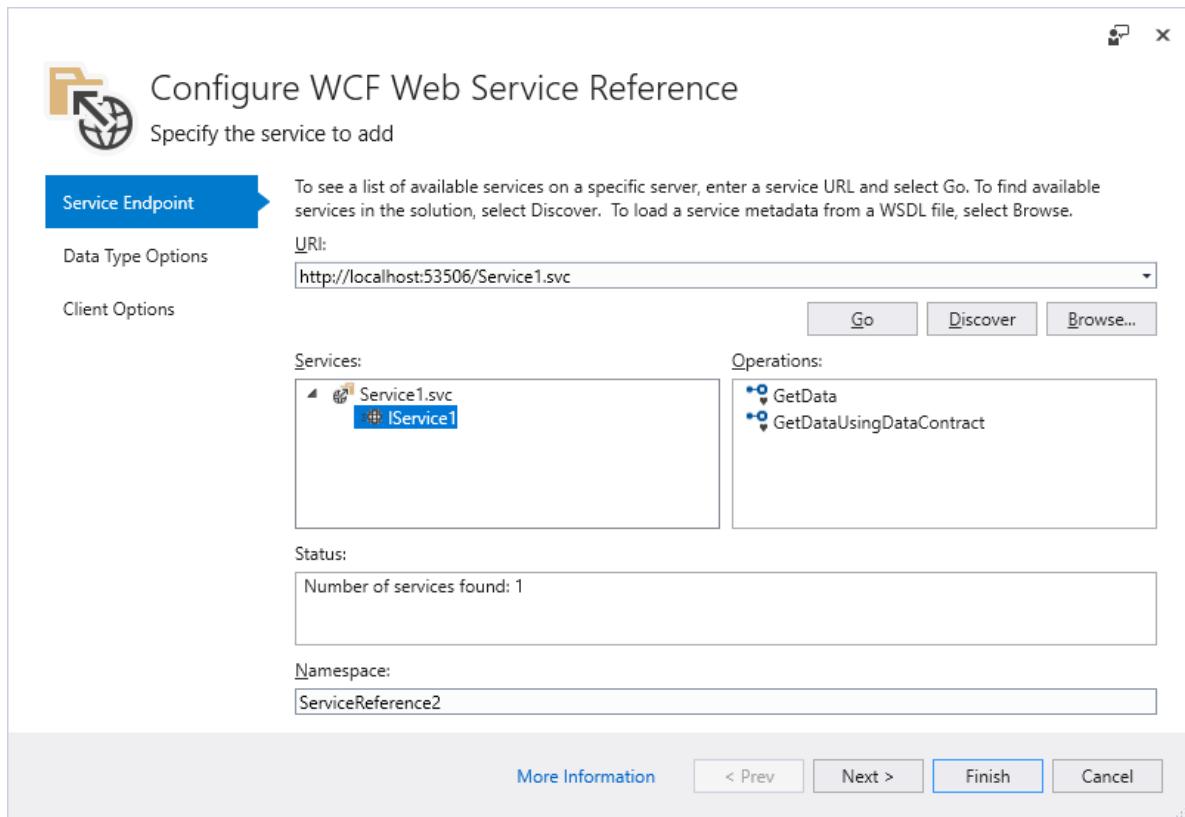
以“ASP.NET Core Web 应用程序”项目模板为例，本文将介绍如何向该项目中添加 WCF 服务引用：

1. 在解决方案资源管理器中，双击项目的“连接的服务”节点（对于 .NET Core 或 .NET Standard 项目，当在解决方案资源管理器中右键单击项目的“依赖项”节点时，该选项可用）。

随即显示“连接的服务”页，如下图所示：



2. 在“连接的服务”页上，单击“Microsoft WCF Web Service Reference Provider”。此操作将显示“配置 WCF Web 服务引用”向导：



3. 选择服务。

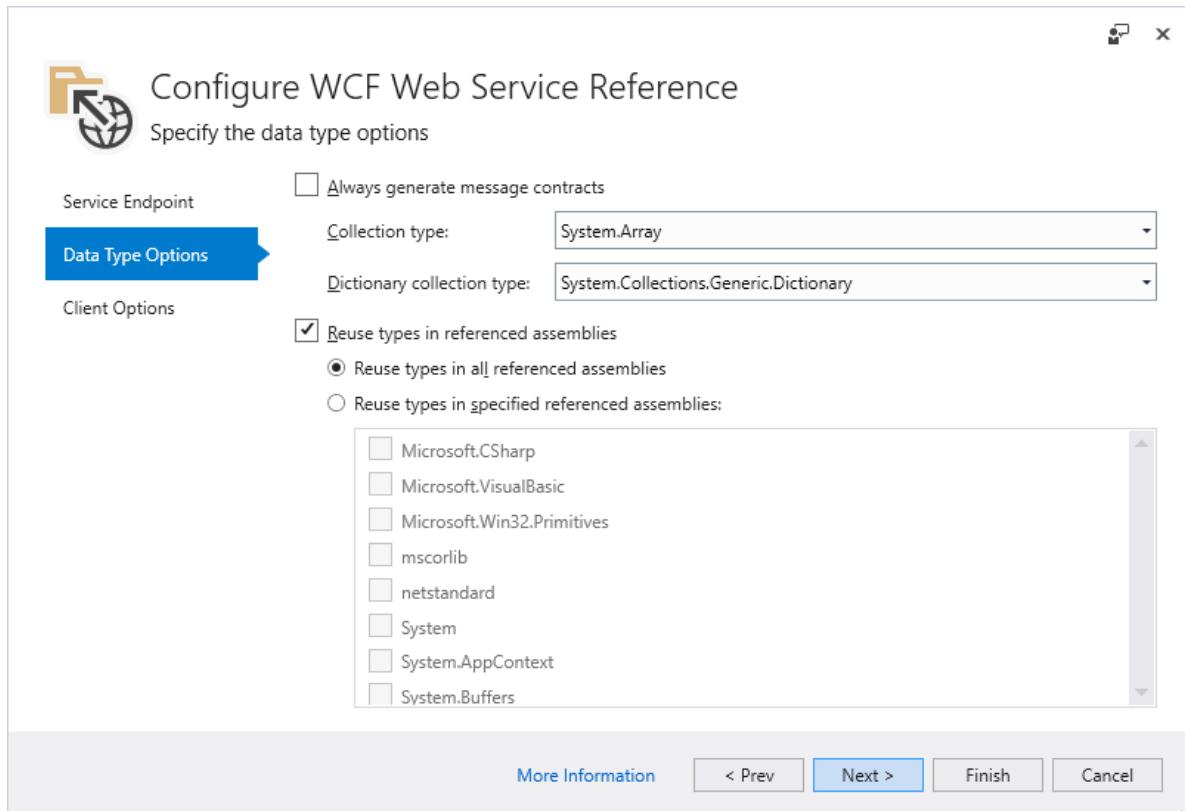
3a. “配置 WCF Web 服务引用”向导中提供了多个服务搜索选项：

- 要搜索当前解决方案中定义的服务，请单击“发现”按钮。
- 要搜索在指定地址托管的服务，请在“地址”框中输入服务 URL，然后单击“转到”按钮。
- 要选择包含 Web 服务元数据信息的 WSDL 文件，请单击“浏览”按钮。

3b. 从“服务”框内的搜索结果列表中选择服务。如果需要，请在相应的“名称空间”文本框中为生成的代码输入命名空间。

3c. 单击“下一步”按钮，打开“数据类型选项”页和“客户端选项”页。或者，单击“完成”按钮，使用默认选项。

4. “数据类型选项”窗体可用于优化生成的服务引用配置设置：



NOTE

如果在项目的引用程序集中定义了服务引用代码生成所需的数据类型，则“重新使用引用程序集中的类型”复选框选项将非常有用。重新使用这些现有数据类型，从而避免编译时类型冲突或运行时问题，这是非常重要的。

加载类型信息时可能会有延迟，具体取决于项目依赖项和其他系统性能因素的数量。加载过程中，“完成”按钮被禁用，除非未选中“重新使用引用程序集中的类型”复选框。

5. 完成后，单击“完成”。

在显示进度的同时，工具：

- 从 WCF 服务下载元数据。
- 在名为“reference.cs”的文件中生成服务引用代码，并将其添加到“连接的服务”节点下的项目。
- 使用在目标平台上编译和运行所需的 NuGet 包引用更新项目文件 (.csproj)。

Adding connected service to project...

Importing web service metadata ...

Number of service endpoints found: 1

Scaffolding service reference code ...

Restoring NuGet packages for bootstrapper ...

[Close](#)

[Abort](#)

进度完成后，可创建生成的 WCF 客户端类型的实例并调用服务操作。

另请参阅

- [Windows Communication Foundation 应用程序入门](#)
- [Visual Studio 中的 Windows Communication Foundation 服务和 WCF 数据服务](#)
- [.NET Core 上 WCF 支持的功能](#)

反馈和问题

如果你有任何问题或反馈，请使用[报告问题](#)工具在[开发者社区](#)进行报告。

发行说明

- 请参阅[发行说明](#)，了解更新的版本信息(包括已知问题)。

.NET Core 的 WCF dotnet-svcutil 工具

2020/3/18 • [Edit Online](#)

Windows Communication Foundation (WCF) dotnet-svcutil 工具是一种 .NET 工具，此工具从网络位置上的 Web 服务中或从 WSDL 文件中检索元数据，并生成包含访问 Web 服务操作的客户端代理方法的 WCF 类。

类似于 .NET Framework 项目的[服务模型元数据 - svcutil](#) 工具，dotnet svcutil 是用于生成 Web 服务引用的命令行工具，与 .NET Core 和 .NET Standard 项目兼容。

dotnet-svcutil 工具是 **WCF Web 服务引用** Visual Studio 连接服务提供程序(随 Visual Studio 2017 版本 15.5 首次推出)的替代选项。dotnet-svcutil 工具作为一种 .NET 工具，可跨平台用于 Linux、macOS 和 Windows。

IMPORTANT

应仅从受信任源引用服务。从不受信任的源添加引用可能会危及安全性。

系统必备

- [dotnet-svcutil 2.x](#)
- [dotnet-svcutil 1.x](#)
- [.NET Core 2.1 SDK 或更高版本](#)
- 你最喜欢的代码编辑器

入门

下面的示例将指导你完成将 Web 服务引用添加到 .NET Core Web 项目并调用该服务所需的步骤。将创建名为“HelloSvcutil”的 .NET Core Web 应用程序，并将引用添加到实现以下协定的 Web 服务：

```
[ServiceContract]
public interface ISayHello
{
    [OperationContract]
    string Hello(string name);
}
```

在此示例中，我们假定 Web 服务托管在以下地址中：<http://contoso.com/SayHello.svc>

从 Windows、macOS 或 Linux 命令窗口执行以下步骤：

1. 为项目创建一个名为“HelloSvcutil”的目录，并将其设置为当前目录，如以下示例所示：

```
mkdir HelloSvcutil
cd HelloSvcutil
```

2. 在该目录中使用 [dotnet new](#) 命令创建新的 C# Web 项目，如下所示：

```
dotnet new web
```

3. 安装 [dotnet-svcutil](#) NuGet 包作为 CLI 工具：

- [dotnet-svcutil 2.x](#)
- [dotnet-svcutil 1.x](#)

```
dotnet tool install --global dotnet-svcutil
```

4. 运行 `dotnet-svcutil` 命令生成 Web 服务引用文件，如下所示：

- [dotnet-svcutil 2.x](#)
- [dotnet-svcutil 1.x](#)

```
dotnet-svcutil http://contoso.com/SayHello.svc
```

生成的文件保存为 `HelloSvcutil/ServiceReference/Reference.cs`。`dotnet-svcutil` 工具还向项目添加代理代码所需的适当 WCF 包作为包引用。

使用服务引用

1. 使用 `dotnet restore` 命令还原 WCF 包，如下所示：

```
dotnet restore
```

2. 找到要使用的客户端类和操作的名称。`Reference.cs` 将包含一个继承自 `System.ServiceModel.ClientBase` 的类，其方法可用于调用服务上的操作。在本例中，想要调用 `SayHello` 服务的 `Hello` 操作。

`ServiceReference.SayHelloClient` 是客户端类的名称，它有一个名为 `HelloAsync` 的方法，可用于调用该操作。

3. 在编辑器中打开 `Startup.cs` 文件，并在顶部为服务引用命名空间添加一个 `using` 语句：

```
using ServiceReference;
```

4. 编辑 `Configure` 方法来调用 Web 服务。为此，可以创建一个继承自 `ClientBase` 的类的实例，并在客户端对象上调用此方法：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.Run(async (context) =>
    {
        var client = new SayHelloClient();
        var response = await client.HelloAsync();
        await context.Response.WriteAsync(response);
    });
}
```

5. 使用 `dotnet run` 命令运行应用程序，如下所示：

```
dotnet run
```

6. 导航到在 Web 浏览器的控制台中列出的 URL(例如, `http://localhost:5000`)。

你将看到以下输出：“Hello dotnet-svcutil!”

有关 `dotnet-svcutil` 工具参数的详细说明, 请调用传递帮助参数的工具, 如下所示:

- [dotnet-svcutil 2.x](#)
- [dotnet-svcutil 1.x](#)

```
dotnet-svcutil --help
```

反馈和问题

如果有任何问题或反馈, 请在 [GitHub 上提问](#)。也可以在 [GitHub 上的 WCF 存储库](#) 中查看任何现有问题。

发行说明

- 请参阅[发行说明](#), 了解更新的版本信息(包括已知问题)。

信息

- [dotnet-svcutil NuGet 包](#)

在 .NET Core 上使用 dotnet-svcutil.xmlserializer

2020/3/18 • [Edit Online](#)

`dotnet-svcutil.xmlserializer` NuGet 包可以为 .NET Core 项目预生成序列化程序集。它为客户端应用程序中由 WCF 服务协定使用的且可由 `XmlSerializer` 序列化的类型预生成 C# 序列化代码。当序列化或反序列化这些类型的对象时，这会提高 XML 序列化的启动性能。

系统必备

- [.NET Core 2.1 SDK 或更高版本](#)
- 你最喜欢的代码编辑器

可以使用命令 `dotnet --info` 检查已安装哪些版本的 .NET Core SDK 和运行时。

入门

在 .NET Core 控制台应用程序中使用 `dotnet-svcutil.xmlserializer`：

1. 在 .NET Framework 中使用默认模板“WCF 服务应用程序”创建一个名为“MyWCFService”的 WCF 服务。在服务方法上添加 `[XmlSerializerFormat]` 属性，如下所示：

```
[ServiceContract]
public interface IService1
{
    [XmlSerializerFormat]
    [OperationContract(Action = "http://tempuri.org/IService1/GetData", ReplyAction =
    "http://tempuri.org/IService1/GetDataResponse")]
    string GetData(int value);
}
```

2. 创建 .NET Core 控制台应用程序作为面向 .NET Core 2.1 或更高版本的 WCF 客户端应用程序。例如，使用以下命令创建名为“MyWCFClient”的应用：

```
dotnet new console --name MyWCFClient
```

要确保项目面向 .NET Core 2.1 或更高版本，请检查项目文件中的 `TargetFramework` XML 元素：

```
<TargetFramework>netcoreapp2.1</TargetFramework>
```

3. 通过运行以下命令将包引用添加到 `System.ServiceModel.Http`：

```
dotnet add package System.ServiceModel.Http
```

4. 添加 WCF 客户端代码：

```

using System.ServiceModel;

class Program
{
    static void Main(string[] args)
    {
        var myBinding = new BasicHttpBinding();
        var myEndpoint = new EndpointAddress("http://localhost:2561/Service1.svc"); //Fill your
service url here
        var myChannelFactory = new ChannelFactory<IService1>(myBinding, myEndpoint);
        IService1 client = myChannelFactory.CreateChannel();
        string s = client.GetData(1);
        ((ICommunicationObject)client).Close();
    }
}

[ServiceContract]
public interface IService1
{
    [XmlSerializerFormat]
    [OperationContract(Action = "http://tempuri.org/IService1/GetData", ReplyAction =
"http://tempuri.org/IService1/GetDataResponse")]
    string GetData(int value);
}

```

5. 通过运行以下命令将引用添加到 `dotnet-svcutil.xmlserializer` 包：

```
dotnet add package dotnet-svcutil.xmlserializer
```

运行该命令应向项目文件中添加一个类似于以下内容的条目：

```

<ItemGroup>
  <DotNetCliToolReference Include="dotnet-svcutil.xmlserializer" Version="1.0.0" />
</ItemGroup>

```

6. 通过运行 `dotnet build` 生成应用程序。如果一切顺利，则会在输出文件夹中生成名为“`MyWCFClient.XmlSerializers.dll`”的程序集。如果该工具无法生成程序集，将在生成输出中看到警告。
7. 例如，通过在浏览器中运行 `http://localhost:2561/Service1.svc` 来启动 WCF 服务。然后启动客户端应用程序，它将在运行时自动加载和使用预生成的序列化程序。

在 .NET Core 上使用 Microsoft XML 序列化程序生成器

2020/3/18 • [Edit Online](#)

本教程介绍如何在 C# .NET Core 应用程序中使用 Microsoft XML 序列化程序生成器。在本教程中可学习：

- 如何创建 .NET Core 应用
- 如何向 Microsoft.XmlSerializer.Generator 包中添加引用
- 如何编辑 MyApp.csproj 以添加依赖项
- 如何添加类和 XmlSerializer
- 如何生成并运行应用程序

正如适用于 .NET Framework 的 [Xml Serializer Generator \(sgen.exe\)](#), [Microsoft.XmlSerializer.Generator NuGet 包](#) 是适用于 .NET Core 和 .NET 标准项目的等效项。它为程序集中包含的类型创建 XML 序列化程序集, 从而提高使用 [XmlSerializer](#) 序列化或反序列化这些类型对象时, XML 序列化的启动性能。

系统必备

完成本教程：

- [.NET Core 2.1 SDK 或更高版本。](#)
- 最喜爱的代码编辑器。

TIP

需要安装代码编辑器？试用 [Visual Studio](#)！

在 .NET Core 控制台应用程序中使用 Microsoft XML 序列化程序生成器

以下说明将展示如何在 .NET Core 控制台应用程序中使用 XML 序列化程序生成器。

创建 .NET Core 控制台应用程序

打开命令提示符, 创建一个名为“MyApp”的文件夹。导航到创建的文件夹, 并键入以下命令：

```
dotnet new console
```

在 MyApp 项目中向 Microsoft.XmlSerializer.Generator 包添加引用

使用 [dotnet add package](#) 命令在项目中添加引用。

类型:

```
dotnet add package Microsoft.XmlSerializer.Generator -v 1.0.0
```

添加包后, 验证对 MyApp.csproj 的更改

打开代码编辑器并开始操作！仍从生成了应用的 MyApp 目录中进行操作。

在文本编辑器中打开 MyApp.csproj。

运行 `dotnet add package` 命令后，会将以下行添加到 MyApp.csproj 项目文件中：

```
<ItemGroup>
  <PackageReference Include="Microsoft.XmlSerializer.Generator" Version="1.0.0" />
</ItemGroup>
```

为 .NET Core CLI 工具支持添加其他 ItemGroup 部分

在已检查的 `ItemGroup` 部分后添加以下行：

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.XmlSerializer.Generator" Version="1.0.0" />
</ItemGroup>
```

在应用程序中添加类

在文本编辑器中打开 Program.cs。在 Program.cs 中添加名为“ MyClass”的类。

```
public class MyClass
{
    public int Value;
}
```

为 `MyClass` 创建 `XmlSerializer`

在 Main 中添加以下行，为 `MyClass` 创建 `XmlSerializer`：

```
var serializer = new System.Xml.Serialization.XmlSerializer(typeof(MyClass));
```

构建并运行应用程序

继续在 MyApp 文件夹中，通过 [运行应用程序，并在运行时自动加载和使用预生成的序列化程序](#) `dotnet run`。

在控制台窗口中键入以下命令：

```
dotnet run
```

NOTE

`dotnet run` 调用 `dotnet build` 来确保已生成要生成的目标，然后调用 `dotnet <assembly.dll>` 运行目标应用程序。

IMPORTANT

本教程中用来运行应用程序的命令和步骤仅用于开发过程。准备好部署应用后，查看适用于 .NET Core 应用的不同部署策略和 `dotnet publish` 命令。

如果一切顺利，则会在输出文件夹中生成名为“ MyApp.XmlSerializers.dll”的程序集。

祝贺你！ 你刚才已完成：

- 创建 .NET Core 应用。
- 向 Microsoft.XmlSerializer.Generator 包中添加引用。
- 编辑 MyApp.csproj 以添加依赖项。
- 添加类和 XmlSerializer。

- 生成和运行应用程序。

相关资源

- [XML 序列化简介](#)
- [如何使用 XmlSerializer 进行序列化 \(C#\)](#)
- [如何:使用 XmlSerializer 进行序列化 \(Visual Basic\)](#)

.NET Core 应用程序发布概述

2020/4/6 • [Edit Online](#)

可在两种模式下发布使用 .NET Core 创建的应用程序，模式会影响用户运行应用的方式。

将应用作为独立应用，生成的应用程序将包含 .NET Core 运行时和库，以及该应用程序及其依赖项。应用程序的用户可以在未安装 .NET Core 运行时的计算机上运行该应用程序。

若将应用发布为依赖于运行时（以前称为“依赖于框架”），则生成的应用程序仅包含该应用程序本身及其依赖项。应用程序的用户必须单独安装 .NET Core 运行时。

默认情况下，这两种发布模式都会生成特定于平台的可执行文件。不使用可执行文件也可以创建依赖于运行时的应用程序，这些应用程序是跨平台的。

生成可执行文件时，可以使用运行时标识符 (RID) 指定目标平台。有关 RID 的详细信息，请参阅 [.NET Core RID 目录](#)。

下表概述了每个 SDK 版本用于将应用发布为依赖于运行时的应用或独立应用的命令：

SDK	SDK 2.1	SDK 3.X	命令
适用于当前平台的 依赖于运行时的可执行文件 。		✓	<code>dotnet publish</code>
适用于特定平台的 依赖于运行时的可执行文件 。		✓	<code>dotnet publish -r <RID> --self-contained false</code>
依赖于运行时的跨平台二进制文件。	✓	✓	<code>dotnet publish</code>
独立可执行文件。	✓	✓	<code>dotnet publish -r <RID></code>

有关详细信息，请参阅 [.NET Core dotnet publish 命令](#)。

生成可执行文件

可执行文件不是跨平台的。它们特定于操作系统和 CPU 体系结构。发布应用并创建可执行文件时，可以将应用发布为[独立应用](#)或[依赖于运行时的应用](#)。将应用发布为独立应用，会在应用中包含 .NET Core 运行时，该应用的用户无需在运行应用前安装 .NET Core。如果将应用发布为依赖于运行时的应用，则该应用不包含 .NET Core 运行时和库，而仅包含该应用和第三方依赖项。

以下命令可生成可执行文件：

SDK	SDK 2.1	SDK 3.X	命令
适用于当前平台的 依赖于运行时的可执行文件 。		✓	<code>dotnet publish</code>
适用于特定平台的 依赖于运行时的可执行文件 。		✓	<code>dotnet publish -r <RID> --self-contained false</code>

	SDK 2.1	SDK 3.X	
独立可执行文件。	✓	✓	<code>dotnet publish -r <RID></code>

生成跨平台二进制文件

以 `dll` 文件的形式将应用发布为 [依赖于运行时的应用](#) 时，将创建跨平台二进制文件。`dll` 文件将与项目同名。例如，如果有名为 `word_reader` 的应用，则会创建名为 `word_reader.dll` 的文件。以这种方式发布的应用可通过 `dotnet <filename.dll>` 命令运行，并且可在任意平台上运行。

只要安装了目标 .NET Core 运行时，就可以在任何操作系统上运行跨平台二进制文件。如果未安装目标 .NET Core 运行时，如果将应用配置为前滚，则它可以使用较新的运行时运行。有关详细信息，请参阅[依赖于运行时的应用前滚](#)。

以下命令可生成跨平台二进制文件：

	SDK 2.1	SDK 3.X	
依赖于运行时的跨平台二进制文件。	✓	✓	<code>dotnet publish</code>

发布依赖于运行时的应用

如果将应用发布为依赖于运行时的应用，则该应用是跨平台的，且不包含 .NET Core 运行时。应用的用户需要安装 .NET Core 运行时。

将应用发布为依赖于运行时的应用，会以 `dll` 文件的形式生成一个 [跨平台二进制文件](#)，还会生成面向当前平台的 [特定于平台的可执行文件](#)。`dll` 是跨平台的，而可执行文件不是。例如，如果发布名为 `word_reader` 的应用且面向 Windows，则将创建 `word_reader.exe` 和 `word_reader.dll`。面向 Linux 或 macOS 时，将创建 `word_reader` 可执行文件和 `word_reader.dll`。有关 RID 的详细信息，请参阅[.NET Core RID 目录](#)。

IMPORTANT

发布依赖于运行时的应用时，.NET Core SDK 2.1 不会生成特定于平台的可执行文件。

可以通过 `dotnet <filename.dll>` 命令运行应用的跨平台二进制文件，并且它可以在任何平台上运行。如果应用使用具有特定于平台的实现的 NuGet 包，则所有平台的依赖项都将连同应用一起复制到发布文件夹。

可以通过将 `-r <RID> --self-contained false` 参数传递到 `dotnet publish` 命令，为特定平台创建可执行文件。省略 `-r` 参数时，将为当前平台创建可执行文件。具有特定于目标平台的依赖项的任何 NuGet 包都将复制到发布文件夹。

优点

- **小型部署**

仅分发应用及其依赖项。.NET Core 运行时和库由用户安装，所有应用共享运行时。

- **跨平台**

应用和任何基于 .NET 的库都可在不同的操作系统上运行。无需为应用定义目标平台。有关 .NET 文件格式的详细信息，请参阅[.NET 程序集文件格式](#)。

- **使用最新修补运行时**

应用会使用目标系统上安装的最新运行时（在 .NET Core 的目标大小系列中）。这意味着应用将自动使用 .NET Core 运行时的最新修补版本。可以重写这一默认行为。有关详细信息，请参阅[依赖于运行时](#)

的应用前滚。

缺点

- 要求预先安装运行时

仅当主机系统上已安装应用设为目标的 .NET Core 版本时，应用才能运行。可以为应用配置前滚行为，要求使用特定版本的 .NET Core 或允许使用较新版本的 .NET Core。有关详细信息，请参阅[依赖于运行时的应用前滚](#)。

- .NET Core 可能更改

可以在运行应用的计算机上更新 .NET Core 运行时和库。在极少数情况下，如果使用 .NET Core 库（大多数应用都会使用），这可能会更改应用的行为。可以配置应用如何使用较新版本的 .NET Core。有关详细信息，请参阅[依赖于运行时的应用前滚](#)。

以下缺陷仅限于.NET Core 2.1 SDK。

- 使用 `dotnet` 命令启动应用

用户必须运行 `dotnet <filename.dll>` 命令来启动你的应用。如果将应用发布为依赖于运行时的应用，则 .NET Core 2.1 SDK 不会生成特定于平台的可执行文件。

示例

发布依赖于运行时的跨平台应用。与 `dll` 文件一起创建面向当前平台的可执行文件。

```
dotnet publish
```

发布依赖于运行时的跨平台应用。与 `dll` 文件一起创建 Linux 64 位可执行文件。此命令对于 .NET Core SDK 2.1 无效。

```
dotnet publish -r linux-x64 --self-contained false
```

发布独立应用

将应用发布为独立应用，将生成特定于平台的可执行文件。输出发布文件夹包含应用的所有组件，包括 .NET Core 库和目标运行时。应用独立于其他 .NET Core 应用，且不使用本地安装的共享运行时。应用的用户无需下载和安装 .NET Core。

针对指定的目标平台生成可执行二进制文件。例如，如果你有一个名为 `word_reader` 的应用，并发布适用于 Windows 的独立可执行文件，则将创建 `word_reader.exe` 文件。针对 Linux 或 macOS 发布时，将创建 `word_reader` 文件。用 `dotnet publish` 命令的 `-r <RID>` 参数指定目标平台和体系结构。有关 RID 的详细信息，请参阅[.NET Core RID 目录](#)。

如果应用具有特定于平台的依赖项（例如包含特定于平台的依赖项的 NuGet 包），这些依赖项将与应用一起复制到发布文件夹。

优点

- 控制 .NET Core 版本

你可以控制随应用部署的 .NET Core 版本。

- 特定于平台的定向

由于你必须为每个平台发布应用，因此你知道应用可运行于哪些平台。如果 .NET Core 引入了新平台，则用户无法在该新平台上运行你的应用，直到你发布面向该平台的版本。在用户在新平台上运行你的应用之前，你可以测试应用以排除兼容性问题。

缺点

- 大型部署

由于你的应用包含 .NET Core 运行时和所有应用依赖项，因此下载大小和所需硬盘空间比依赖于运行时的版本大。

TIP

可以通过使用 .NET Core 全球化固定模式在 Linux 系统上减少大约 28 MB 的部署大小。这会强制应用像处理固定区域性一样处理所有区域性。

- 较难更新 .NET Core 版本

只能通过发布新版本的应用来升级（与应用一起分发的）.NET Core 运行时。你负责向 .NET Core 运行时的安全修补程序提供应用程序的更新版本。

示例

发布独立应用。创建 macOS 64 位可执行文件。

```
dotnet publish -r osx-x64
```

发布独立应用。创建 Windows 64 位可执行文件。

```
dotnet publish -r win-x64
```

请参阅

- [使用 .NET Core CLI 部署 .NET Core 应用。](#)
- [使用 Visual Studio 部署 .NET Core 应用。](#)
- [包、元包和框架。](#)
- [.NET Core 运行时标识符 \(RID\) 目录。](#)
- [选择要使用的 .NET Core 版本。](#)

使用 .NET Core CLI 发布 .NET Core 应用

2020/3/18 • [Edit Online](#)

本文演示了如何使用命令行发布 .NET Core 应用程序。.NET Core 提供了三种发布应用程序的方式。依赖于框架的部署生成一个跨平台 .dll 文件，该文件使用本地安装的 .NET Core 运行时。依赖于框架的可执行文件生成特定于平台的可执行文件，后者使用本地安装的 .NET Core 运行时。独立可执行文件生成特定于平台的可执行文件，并包含 .NET Core 运行时的本地副本。

请参阅 [.NET Core 应用程序部署](#) 了解有关这些发布模式的概述。

正在查找有关 CLI 的快速帮助？下表列出了一些关于如何发布应用的示例。可以使用 `-f <TFM>` 参数或通过编辑项目文件来指定目标框架。有关详细信息，请参阅[发布基本知识](#)。

框架	SDK 版本	命令
依赖框架的部署	2.x	<code>dotnet publish -c Release</code>
依赖于框架的可执行文件	2.2	<code>dotnet publish -c Release -r <RID> --self-contained false</code>
	3.0	<code>dotnet publish -c Release -r <RID> --self-contained false</code>
	3.0*	<code>dotnet publish -c Release</code>
独立部署	2.1	<code>dotnet publish -c Release -r <RID> --self-contained true</code>
	2.2	<code>dotnet publish -c Release -r <RID> --self-contained true</code>
	3.0	<code>dotnet publish -c Release -r <RID> --self-contained true</code>

* 使用 SDK 版本 3.0 时，框架依赖可执行文件是运行基本 `dotnet publish` 命令时的默认发布模式。仅当项目定目标到 .NET Core 2.1 或 .NET Core 3.0 时，这才适用。

发布基本知识

发布应用时，项目文件的 `<TargetFramework>` 设置指定默认目标框架。可以将目标框架更改为任意有效[目标框架名字对象 \(TFM\)](#)。例如，如果项目使用 `<TargetFramework>netcoreapp2.2</TargetFramework>`，则会创建面向 .NET Core 2.2 的二进制文件。此设置中指定的 TFM 是 `dotnet publish` 命令使用的默认目标。

如果要以多个框架为目标，可以将 `<TargetFrameworks>` 设置设置为多个以分号分隔的 TFM 值。可以使用 `dotnet publish -f <TFM>` 命令发布其中一个框架。例如，如果有 `<TargetFrameworks>netcoreapp2.1;netcoreapp2.2</TargetFrameworks>` 并运行 `dotnet publish -f netcoreapp2.1`，则会创建面向 .NET Core 2.1 的二进制文件。

除非另有设置，否则 `dotnet publish` 命令的输出目录为 `./bin/<BUILD-CONFIGURATION>/<TFM>/publish/`。除非使用 `-c` 参数进行更改，否则默认的 BUILD-CONFIGURATION 模式为 Debug。例如，`dotnet publish -c Release -f netcoreapp2.1` 发布到 `myfolder/bin/Release/netcoreapp2.1/publish/`。

如果使用 .NET Core SDK 3.0 或更高版本，则面向 .NET Core 版本 2.1、2.2、3.0 或更高版本的应用的默认发布模式为依赖于框架的可执行文件。

如果使用 .NET Core SDK 2.1，则面向 .NET Core 版本 2.1 和 2.2 的应用的默认发布模式为依赖于框架的部署。

本机依赖项

如果应用具有本机依赖项，则只能在相同操作系统上运行。例如，如果应用使用本机 Windows API，则不能在 macOS 或 Linux 上运行。需要提供特定于平台的代码并为每个平台编译可执行文件。

另外，如果引用的库具有本机依赖项，则应用可能无法在每个平台上运行。但是，引用的 NuGet 包可能包含特定于平台的版本，以便处理所需的本机依赖项。

使用本机依赖项分发应用时，可能需要使用 `dotnet publish -r <RID>` 开关来指定想要发布的目标平台。有关运行时标识符的详细信息，请参阅[运行时标识符 \(RID\) 目录](#)。

有关特定于平台的二进制文件的详细信息，请参阅[依赖于框架的可执行文件](#)和[独立部署](#)部分。

示例应用

可使用以下应用来探索发布命令。通过在终端中运行以下命令来创建应用：

```
mkdir apptest1  
cd apptest1  
dotnet new console  
dotnet add package Figgle
```

控制台模板生成的 `Program.cs` 或 `Program.vb` 文件需要进行以下更改：

```
using System;

namespace apptest1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Figgle.FiggleFonts.Standard.Render("Hello, World!"));
        }
    }
}
```

```
Module Program
    Sub Main(args As String())
        Console.WriteLine(Figgle.FiggleFonts.Standard.Render("Hello, World!"))
    End Sub
End Module
```

运行应用 (`dotnet run`) 时，将显示以下输出：

依赖框架的部署

对于 .NET Core SDK 2.x CLI, 依赖于框架的部署 (FDD) 是基本 `dotnet publish` 命令的默认模式。

将应用作为 FDD 发布时, 会在 `./bin/<BUILD-CONFIGURATION>/<TFM>/publish/` 文件夹中创建 `<PROJECT-NAME>.dll` 文件。若要运行应用, 请导航到输出文件夹并使用 `dotnet <PROJECT-NAME>.dll` 命令。

应用配置为面向 .NET Core 的特定版本。目标 .NET Core 运行时需要位于运行应用程序的任何计算机上。例如, 如果应用面向 .NET Core 2.2, 则运行应用的任何计算机都必须已安装 .NET Core 2.2 运行时。如[发布基础知识](#)部分中所述, 可以编辑项目文件为更改默认目标框架或面向多个框架。

发布 FDD 会创建一个应用, 该应用会自动前滚到运行该应用的系统上可用的最新 .NET Core 安全修补程序。有关编译时的版本绑定的详细信息, 请参阅[选择要使用的 .NET Core 版本](#)。

依赖于框架的可执行文件

对于 .NET Core SDK 3.x CLI, 依赖于框架的可执行文件 (FDE) 是基本 `dotnet publish` 命令的默认模式。只要想要面向当前操作系统, 就不需要指定任何其他参数。

在此模式下, 将创建特定于平台的可执行主机来托管跨平台应用。此模式类似于 FDD, 因为 FDD 需要 `dotnet` 命令形式的主机。每个平台的主机可执行文件名各不相同, 其文件名类似于 `<PROJECT-FILE>.exe`。可以直接运行此可执行文件, 而不是调用 `dotnet <PROJECT-FILE>.dll`, 这仍然是运行应用的可接受方式。

应用配置为面向 .NET Core 的特定版本。目标 .NET Core 运行时需要位于运行应用程序的任何计算机上。例如, 如果应用面向 .NET Core 2.2, 则运行应用的任何计算机都必须已安装 .NET Core 2.2 运行时。如[发布基础知识](#)部分中所述, 可以编辑项目文件为更改默认目标框架或面向多个框架。

发布 FDE 会创建一个应用, 该应用会自动前滚到运行该应用的系统上可用的最新 .NET Core 安全修补程序。有关编译时的版本绑定的详细信息, 请参阅[选择要使用的 .NET Core 版本](#)。

对于 .NET Core 2.2 及早期版本, 必须使用以下开关通过 `dotnet publish` 命令来发布 FDE:

- `-r <RID>` 此开关使用标识符 (RID) 来指定目标平台。有关运行时标识符的详细信息, 请参阅[运行时标识符 \(RID\) 目录](#)。
- `--self-contained false` 此开关告知 .NET Core SDK 创建可执行文件作为 FDE。

每次使用 `-r` 开关时, 输出文件路都将更改为: `./bin/<BUILD-CONFIGURATION>/<TFM>/<RID>/publish/`

如果使用[示例应用](#), 请运行 `dotnet publish -f netcoreapp2.2 -r win10-x64 --self-contained false`。此命令将创建以下可执行文件: `./bin/Debug/netcoreapp2.2/win10-x64/publish/apptest1.exe`

NOTE

可以通过启用全局固定模式来降低部署的总大小。此模式适用于不具有全局意识且可以使用[固定区域性的](#)格式约定、大小写约定以及字符串比较和排序顺序的应用程序。有关全局固定模式及其启用方式的详细信息, 请参阅[.NET Core 全局固定模式](#)。

独立部署

发布独立部署 (SCD) 时, .NET Core SDK 创建特定于平台的可执行文件。发布 SCD 时会包括运行应用所需的所有 .NET Core 文件, 但不包含[.NET Core 的本机依赖项](#)。这些依赖项必须在应用运行前存在于系统中。

发布 SCD 会创建一个不会前滚到最新可用 .NET Core 安全修补程序的应用。有关编译时的版本绑定的详细信息, 请参阅[选择要使用的 .NET Core 版本](#)。

必须通过 `dotnet publish` 命令使用以下开关来发布 SCD:

- `-r <RID>` 此开关使用标识符 (RID) 来指定目标平台。有关运行时标识符的详细信息, 请参阅[运行时标识符 \(RID\) 目录](#)。
- `--self-contained true` 此开关告知 .NET Core SDK 创建可执行文件作为 SCD。

NOTE

可以通过启用全局固定模式来降低部署的总大小。此模式适用于不具有全局意识且可以使用[固定区域性](#)的格式约定、大小写约定以及字符串比较和排序顺序的应用程序。有关全局固定模式及其启用方式的详细信息, 请参阅[.NET Core 全局固定模式](#)。

请参阅

- [.NET Core 应用程序部署概述](#)
- [.NET Core 运行时标识符 \(RID\) 目录](#)

使用 Visual Studio 部署 .NET Core 应用

2020/3/18 • [Edit Online](#)

可将 .NET Core 应用程序部署为依赖框架的部署 或 独立部署，前者包含应用程序二进制文件，但依赖目标系统上存在的 .NET Core，而后者同时包含应用程序和 .NET Core 二进制文件。有关 .NET Core 应用程序部署的概述，请参阅 [.NET Core 应用程序部署](#)。

以下各节演示如何使用 Microsoft Visual Studio 创建下列各类部署：

- 依赖框架的部署
- 包含第三方依赖项的依赖框架的部署
- 独立部署
- 包含第三方依赖项的独立部署

有关使用 Visual Studio 开发 .NET Core 应用程序的信息，请参阅 [.NET Core 依赖项和要求](#)。

依赖框架的部署

如果不使用第三方依赖项，部属依赖框架的部署只包括生成、测试和发布应用。一个用 C# 编写的简单示例可说明此过程。

1. 创建项目。

选择“文件”>“新建”>“项目”。在“新建项目”对话框中，在“已安装”项目类型窗格中展开你的语言的(C# 或 Visual Basic)项目类别，选择“.NET Core”，然后在中心窗格中选择控制台应用 (.NET Core) 模板。在“名称”文本框中输入项目名称，如“FDD”。选择“确定”按钮。

2. 添加应用程序的源代码。

在编辑器中打开 Program.cs 文件或 Program.vb 文件，然后使用下列代码替换自动生成的代码。它会提示用户输入文本，并显示用户输入的个别词。它使用正则表达式 `\w+` 来将输入文本中的词分开。

```
using System;
using System.Text.RegularExpressions;

namespace Applications.ConsoleApps
{
    public class ConsoleParser
    {
        public static void Main()
        {
            Console.WriteLine("Enter any text, followed by <Enter>:\n");
            String s = Console.ReadLine();
            ShowWords(s);
            Console.Write("\nPress any key to continue... ");
            Console.ReadKey();
        }

        private static void ShowWords(String s)
        {
            String pattern = @"\w+";
            var matches = Regex.Matches(s, pattern);
            if (matches.Count == 0)
            {
                Console.WriteLine("\nNo words were identified in your input.");
            }
            else
            {
                Console.WriteLine($"\\nThere are {matches.Count} words in your string:");
                for (int ctr = 0; ctr < matches.Count; ctr++)
                {
                    Console.WriteLine($"    #{ctr,2}: '{matches[ctr].Value}' at position
{matches[ctr].Index}");
                }
            }
        }
    }
}
```

```

Imports System.Text.RegularExpressions

Namespace Applications.ConsoleApps
    Public Module ConsoleParser
        Public Sub Main()
            Console.WriteLine("Enter any text, followed by <Enter>:")
            Console.WriteLine()
            Dim s = Console.ReadLine()
            ShowWords(s)
            Console.Write($"{vbCrLf}Press any key to continue... ")
            Console.ReadKey()
        End Sub

        Private Sub ShowWords(s As String)
            Dim pattern = "\w+"
            Dim matches = Regex.Matches(s, pattern)
            Console.WriteLine()
            If matches.Count = 0 Then
                Console.WriteLine("No words were identified in your input.")
            Else
                Console.WriteLine($"There are {matches.Count} words in your string:")
                For ctr = 0 To matches.Count - 1
                    Console.WriteLine($"#{ctr,2}: '{matches(ctr).Value}' at position
{matches(ctr).Index}")
                Next
            End If
            Console.WriteLine()
        End Sub
    End Module
End Namespace

```

3. 创建应用的调试版本。

选择“生成”>“生成解决方案”。也可通过选择“调试”>“开始调试”来编译和运行应用程序的调试版本。

4. 部署应用。

调试并测试程序后，创建要与应用一起部署的文件。若要从 Visual Studio 发布，请执行以下操作：

- 将工具栏上的解决方案配置从“调试”更改为“发布”，生成应用的发布（而非调试）版本。
- 在“解决方案资源管理器”中右键单击项目（而非解决方案），然后选择“发布”。
- 在“发布”选项卡上，选择“发布”。Visual Studio 将包含应用程序的文件写入本地文件系统。
- “发布”选项卡现在显示单个配置文件 FolderProfile。该配置文件的配置设置显示在选项卡的“摘要”部分。

生成的文件位于 Windows 上名为 `Publish` 的目录（对于 Unix 系统，是名为 `publish` 的目录）中的项目 `\bin\release\netcoreapp2.1` 子目录的子目录中。

与应用程序的文件一起，发布过程将发出包含应用调试信息的程序数据库 (.pdb) 文件。该文件主要用于调试异常。可以选择不使用应用程序文件打包该文件。但是，如果要调试应用的发布版本，则应保存该文件。

可以采用任何喜欢的方式部署完整的应用程序文件集。例如，可以使用简单的 `copy` 命令将其打包为 Zip 文件，或者使用选择的安装包进行部署。安装成功后，用户可通过使用 `dotnet` 命令或提供应用程序文件名（如 `dotnet fdd.dll`）来执行应用程序。

除应用程序二进制文件外，安装程序还应捆绑共享框架安装程序，或在安装应用程序的过程中将其作为先决条件进行检查。安装共享框架需要管理员/根访问权限，因为它属于计算机范围。

包含第三方依赖项的依赖框架的部署

要使用一个或多个第三方依赖项来部署依赖框架的部署，需要任何依赖项都可供项目使用。在生成应用之前，还需执行以下额外步骤：

1. 使用 NuGet 包管理器 向项目添加对 NuGet 包的引用；如果系统上还没有此包，请先安装它。要打开包管理器，请选择“工具” > “NuGet 包管理器” > “管理解决方案的 NuGet 包”。
2. 确认系统上安装了第三方依赖项（例如，`Newtonsoft.Json`）；如果没有，请安装它们。“已安装” 选项卡列出了系统中已安装的 NuGet 包。如果此处未列出 `Newtonsoft.Json`，请选择“浏览” 选项卡，然后在搜索框中输入“`Newtonsoft.Json`”。选择 `Newtonsoft.Json`，在右侧窗格中选择项目，然后选择“安装”。
3. 如果系统中已安装 `Newtonsoft.Json`，请在“管理解决方案包” 选项卡的右侧窗格中选择项目，将其添加到项目。

如果依赖框架的部署具有第三方依赖项，则其可移植性只与第三方依赖项相同。例如，如果某个第三方库只支持 macOS，该应用将无法移植到 Windows 系统。当第三方依赖项本身取决于本机代码时，也可能发生此情况。`Kestrel 服务器`就是一个很好的示例，它需要 `libuv` 的本机依赖项。当为具有此类第三方依赖项的应用程序创建 FDD 时，已发布的输出会针对每个本机依赖项支持（存在于 NuGet 包中）的运行时标识符（RID）包含一个文件夹。

不包含第三方依赖项的独立部署

部署没有第三方依赖项的独立部署包括创建项目、修改 `csproj` 文件、生成、测试以及发布应用。一个用 C# 编写的简单示例可说明此过程。首先，对项目进行创建、编码及测试，就像对依赖框架的部署的操作一样：

1. 创建项目。

选择“文件” > “新建” > “项目”。在“新建项目”对话框中，在“已安装”项目类型窗格中展开你的语言的（C# 或 Visual Basic）项目类别，选择“.NET Core”，然后在中心窗格中选择控制台应用 (.NET Core) 模板。在“名称” 文本框中输入项目名称如“SCD”，然后选择“确定” 按钮。

2. 添加应用程序的源代码。

在编辑器中打开 `Program.cs` 或 `Program.vb` 文件，然后使用下列代码替换自动生成的代码。它会提示用户输入文本，并显示用户输入的个别词。它使用正则表达式 `\w+` 来将输入文本中的词分开。

```
using System;
using System.Text.RegularExpressions;

namespace Applications.ConsoleApps
{
    public class ConsoleParser
    {
        public static void Main()
        {
            Console.WriteLine("Enter any text, followed by <Enter>:\n");
            String s = Console.ReadLine();
            ShowWords(s);
            Console.Write("\nPress any key to continue... ");
            Console.ReadKey();
        }

        private static void ShowWords(String s)
        {
            String pattern = @"\w+";
            var matches = Regex.Matches(s, pattern);
            if (matches.Count == 0)
            {
                Console.WriteLine("\nNo words were identified in your input.");
            }
            else
            {
                Console.WriteLine($"\\nThere are {matches.Count} words in your string:");
                for (int ctr = 0; ctr < matches.Count; ctr++)
                {
                    Console.WriteLine($"    #{ctr,2}: '{matches[ctr].Value}' at position
{matches[ctr].Index}");
                }
            }
        }
    }
}
```

```

Imports System.Text.RegularExpressions

Namespace Applications.ConsoleApps
    Public Module ConsoleParser
        Public Sub Main()
            Console.WriteLine("Enter any text, followed by <Enter>:")
            Console.WriteLine()
            Dim s = Console.ReadLine()
            ShowWords(s)
            Console.Write($"{vbCrLf}Press any key to continue... ")
            Console.ReadKey()
        End Sub

        Private Sub ShowWords(s As String)
            Dim pattern = "\w+"
            Dim matches = Regex.Matches(s, pattern)
            Console.WriteLine()
            If matches.Count = 0 Then
                Console.WriteLine("No words were identified in your input.")
            Else
                Console.WriteLine($"There are {matches.Count} words in your string:")
                For ctr = 0 To matches.Count - 1
                    Console.WriteLine($"    #{ctr,2}: '{matches(ctr).Value}' at position
{matches(ctr).Index}")
                Next
            End If
            Console.WriteLine()
        End Sub
    End Module
End Namespace

```

3. 确定是否要使用全球化固定模式。

特别是如果应用面向 Linux，则可以通过利用[全球化固定模式](#)来减小部署的总规模。全球化固定模式适用于不具有全局意识且可以使用[固定区域性的](#)格式约定、大小写约定以及字符串比较和排序顺序的应用程序。

要启用固定模式，右键单击“解决方案资源管理器”中的项目（不是解决方案），然后选择“编辑 SCD.csproj”或“编辑 SCD.vbproj”。然后将以下突出显示的行添加到文件中：

```

<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>netcoreapp3.0</TargetFramework>
    </PropertyGroup>
    <ItemGroup>
        <RuntimeHostConfigurationOption Include="System.Globalization.Invariant" Value="true" />
    </ItemGroup>
</Project>

```

4. 创建应用程序的调试版本。

选择“生成”>“生成解决方案”。也可通过选择“调试”>“开始调试”来编译和运行应用程序的调试版本。通过此调试步骤，可以识别应用程序在主机平台上运行时出现的问题。仍然必须在每个目标平台上对其进行测试。

如果已启用全球化固定模式，请特别确保测试缺少对文化敏感的数据是否适合应用程序。

完成调试后，可以发布独立部署：

- [Visual Studio 15.6 及更早版本](#)
- [Visual Studio 15.7 及更高版本](#)

调试并测试程序后，为应用的每个目标平台创建要与应用一起部署的文件。

若要从 Visual Studio 发布应用，请执行以下操作：

1. 定义应用的目标平台。

- a. 在“解决方案资源管理器”中右键单击项目（而非解决方案），然后选择“编辑 SCD.csproj”。
- b. 在 csproj 文件（该文件用于定义应用的目标平台）的 `<PropertyGroup>` 部分中创建 `<RuntimeIdentifiers>` 标记，然后指定每个目标平台的运行时标识符 (RID)。还需要添加分号来分隔 RID。请查看[运行时标识符目录](#)，获取运行时标识符列表。

例如，以下示例表明应用在 64 位 Windows 10 操作系统和 64 位 OS X 10.11 版本的操作系统上运行。

```
<PropertyGroup>
  <RuntimeIdentifiers>win10-x64;osx.10.11-x64</RuntimeIdentifiers>
</PropertyGroup>
```

`<RuntimeIdentifiers>` 元素可能会进入 csproj 文件的任何 `<PropertyGroup>` 中。本节后面部分将显示完整的示例 csproj 文件。

2. 发布你的应用。

调试并测试程序后，为应用的每个目标平台创建要与应用一起部署的文件。

若要从 Visual Studio 发布应用，请执行以下操作：

- a. 将工具栏上的解决方案配置从“调试”更改为“发布”，生成应用的发布（而非调试）版本。
- b. 在“解决方案资源管理器”中右键单击项目（而非解决方案），然后选择“发布”。
- c. 在“发布”选项卡上，选择“发布”。Visual Studio 将包含应用程序的文件写入本地文件系统。
- d. “发布”选项卡现在显示单个配置文件 FolderProfile。该配置文件的配置设置显示在选项卡的“摘要”部分。目标运行时用于标识已发布的运行时，目标位置用于标识独立部署文件的写入位置。
- e. 默认情况下，Visual Studio 将所有已发布文件写入单个目录。为了方便起见，最好为每个目标运行时创建单个配置文件，并将已发布文件置于特定于平台的目录中。这包括为每个目标平台创建单独的发布配置文件。现在，请执行下列操作，为每个平台重新生成应用程序：
 - a. 在“发布”对话框中选择“创建新配置文件”。
 - b. 在“选取发布目标”对话框中，将“选择文件夹”位置更改为 bin\Release\PublishOutput\win10-x64。选择“确定”。
 - c. 在配置文件列表中选择新配置文件 (FolderProfile1)，并确保“目标运行时”为 win10-x64。如果不是，请选择“设置”。在“配置文件设置”对话框中，将“目标运行时”更改为 win10-x64，然后选择“保存”。否则，选择“取消”。
 - d. 选择“发布”，发布 64 位 Windows 10 平台的应用。
 - e. 请再次按照上述步骤创建 osx.10.11-x64 平台的配置文件。“目标位置”为 bin\Release\PublishOutput\osx.10.11-x64，“目标运行时”为 osx.10.11-x64。Visual Studio 分配给此配置文件的名称是 FolderProfile2。

每个目标位置中都包含启动应用所需的完整文件集（既包含应用文件，又包含所有 .NET Core 文件）。

与应用程序的文件一起，发布过程将发出包含应用调试信息的程序数据库 (.pdb) 文件。该文件主要用于调试异常。可以选择不使用应用程序文件打包该文件。但是，如果要调试应用的发布版本，则应保存该文件。

可按照任何喜欢的方式部署已发布的文件。例如，可以使用简单的 `copy` 命令将其打包为 Zip 文件，或者使用选择

的安装包进行部署。

下面是此项目完整的 csproj 文件。

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
    <RuntimeIdentifiers>win10-x64;osx.10.11-x64</RuntimeIdentifiers>
  </PropertyGroup>
</Project>
```

包含第三方依赖项的独立部署

部署包含一个或多个第三方依赖项的独立部署包括添加依赖项。在生成应用之前，还需执行以下额外步骤：

1. 使用 NuGet 包管理器 向项目添加对 NuGet 包的引用；如果系统上还没有此包，请先安装它。要打开包管理器，请选择“工具”>“NuGet 包管理器”>“管理解决方案的 NuGet 包”。
2. 确认系统上安装了第三方依赖项（例如，`Newtonsoft.Json`）；如果没有，请安装它们。“已安装”选项卡列出了系统中已安装的 NuGet 包。如果此处未列出 `Newtonsoft.Json`，请选择“浏览”选项卡，然后在搜索框中输入“Newtonsoft.Json”。选择 `Newtonsoft.Json`，在右侧窗格中选择项目，然后选择“安装”。
3. 如果系统中已安装 `Newtonsoft.Json`，请在“管理解决方案包”选项卡的右侧窗格中选择项目，将其添加到项目。

下面是此项目的完整 csproj 文件：

- [Visual Studio 15.6 及更早版本](#)
- [Visual Studio 15.7 及更高版本](#)

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
    <RuntimeIdentifiers>win10-x64;osx.10.11-x64</RuntimeIdentifiers>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="10.0.2" />
  </ItemGroup>
</Project>
```

部署应用程序时，应用中使用的任何第三方依赖项也包含在应用程序文件中。运行应用的系统上不需要第三方库。

可以只将具有一个第三方库的独立部署部署到该库支持的平台。这与依赖框架的部署中具有本机依赖项和第三方依赖项相似，其中的本机依赖项不会存在于目标平台上，除非之前在该平台上安装了该依赖项。

请参阅

- [.NET Core 应用程序部署](#)
- [.NET Core 运行时标识符 \(RID\) 目录](#)

如何使用 .NET Core CLI 创建 NuGet 包

2020/3/18 • [Edit Online](#)

NOTE

下例是关于使用 Unix 的命令行。此处显示的 `dotnet pack` 命令工作方式与在 Windows 上相同。

对于 .NET Standard 和 .NET Core，所有库都应以 NuGet 包方式发布。实际上，这是所有 .NET 标准库的发布和使用方式。可以使用 `dotnet pack` 命令轻松实现此操作。

假设你刚编写了一个很棒的新库，并想通过 NuGet 发布。你就可以使用跨平台工具创建一个 NuGet 包，完全照做就行！下例假定使用一个名为“SuperAwesomeLibrary”的库，该库以 `netstandard1.0` 为目标。

如果存在可传递的依赖项，也就是说，如果一个项目依赖于另一个包，则在创建 NuGet 包前，确保使用 `dotnet restore` 命令还原整个解决方案的包。否则将导致 `dotnet pack` 命令不能正常运行。

NOTE

从 .NET Core 2.0 SDK 开始，无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build` 和 `dotnet run`。在执行显式还原有意义的某些情况下，例如 [Azure DevOps Services 中的持续集成生成中](#)，或在需要显式控制还原发生时间的生成系统中，它仍然是有效的命令。

确认包已还原后，可以导航到库所在的目录：

```
cd src/SuperAwesomeLibrary
```

然后，只需在命令行中输入一个命令：

```
dotnet pack
```

/bin/Debug 文件夹现在如下所示：

```
$ ls bin/Debug  
netstandard1.0/  
SuperAwesomeLibrary.1.0.0.nupkg  
SuperAwesomeLibrary.1.0.0.symbols.nupkg
```

这将生成能够进行调试的包。如果想要生成二进制文件版本的 NuGet 包，只需添加 `--configuration`（或 `-c`）开关并使用 `release` 作为参数。

```
dotnet pack --configuration release
```

/bin 文件夹现在将包含一个 `release` 文件夹，后者包含的 NuGet 包为进制文件版本：

```
$ ls bin/release  
netstandard1.0/  
SuperAwesomeLibrary.1.0.0.nupkg  
SuperAwesomeLibrary.1.0.0.symbols.nupkg
```

因此，现在可以使用此必需的文件发布 NuGet 包了！

不要混淆 `dotnet pack` 和 `dotnet publish`

务必注意，不是任何时候都涉及 `dotnet publish` 命令。`dotnet publish` 命令用于在同一个包中部署具有所有依赖项的应用程序 - 而不是用于生成通过 NuGet 发布和使用的 NuGet 包。

另请参阅

- [快速入门: 创建和发布包](#)

自包含部署运行时前滚

2020/3/18 • [Edit Online](#)

.NET Core [自包含应用程序部署](#)包括 .NET Core 库和 .NET Core 运行时。从 .NET Core 2.1 SDK(版本 2.1.300)开始, [自包含应用程序部署在计算机上发布最高版本的修补程序运行时](#)。默认情况下, [自包含部署的 dotnet publish 选择作为发布计算机上 SDK 一部分而安装的最新版本](#)。这让部署的应用程序在 `publish` 期间能与安全修补程序(以及其他修补程序)配合运行。若要获取新的修补程序, 需要重新发布应用程序。[自包含应用程序是通过在 -r <RID> 命令上指定 dotnet publish 创建的, 或是通过在项目文件 \(csproj / vbproj\) 或命令行中指定运行时标识符 \(RID\) 创建的。](#)

修补程序版本前滚概述

`restore`、`build` 和 `publish` 是可以单独运行的 `dotnet` 命令。运行时选择属于 `restore` 操作, 而不是 `publish` 或 `build` 操作。如果调用 `publish`, 则会选择最新的修补程序版本。如果调用带有 `publish` 参数的 `--no-restore`, 则可能不会获取所需的修补程序版本, 因为没有先使用新的自包含应用程序发布策略执行 `restore`。在这种情况下, 会出现生成错误, 并显示以下类似文本:

"已使用 Microsoft.NETCore.App 版本 2.0.0 还原该项目, 但是按照当前设置, 将改为使用版本 2.0.6。若要解决此问题, 请确保将相同的设置用于 `restore` 和后续操作, 例如 `build` 或 `publish`。如果在 `build` 或 `publish` 期间设置了 `Runtimeldentifier` 属性, 而没有在 `restore` 过程中设置, 通常就会出现此问题。"

NOTE

`restore` 和 `build` 可以作为另一个命令(例如 `publish`)的组成部分隐式运行。当作为另一个命令的组成部分隐式运行时, 它们会带有附加的上下文, 以便生成正确的项目。如果使用某个运行时(例如 `publish`)执行 `dotnet publish -r linux-x64`, 隐式的 `restore` 会还原 `linux-x64` 运行时的包。如果显示调用 `restore`, 则默认情况下它不会还原运行时包, 因为没有上下文。

如何避免在 publish 过程中 restore

你可能在进行 `restore` 操作时不需要运行 `publish`。在创建自包含应用程序时, 若要避免在 `restore` 过程中进行 `publish`, 请执行以下操作:

- 将 `RuntimeIdentifiers` 属性设为一个分号分隔的列表, 其中包含所有要发布的 RID。
- 将 `TargetLatestRuntimePatch` 属性设置为 `true`。

使用 dotnet publish 选项的 no-restore 参数

如果要使用同样的项目文件创建自包含应用程序和[依赖框架的应用程序](#), 并且想通过 `--no-restore` 使用 `dotnet publish` 参数, 请选择以下各项之一:

- 首选依赖框架的行为。如果是依赖框架的应用程序, 则此选项为默认行为。如果是自包含应用程序, 并且能使用未带修补程序的 2.1.0 本地运行时, 请在项目文件中将 `TargetLatestRuntimePatch` 设为 `false`。
- 首选自包含行为。如果是自包含应用程序, 则此选项为默认行为。如果是依赖框架的应用程序, 且需要安装最新版本的修补程序, 请在项目文件中将 `TargetLatestRuntimePatch` 设为 `true`。
- 通过在项目文件中将 `RuntimeFrameworkVersion` 设为特定的修补程序版本, 可对运行时框架版本进行显式控制。

剪裁独立部署和可执行文件

2020/4/9 • [Edit Online](#)

当发布独立应用程序时，.NET Core 运行时会与该应用程序捆绑在一起。此捆绑包向打包的应用程序添加了大量内容。

在涉及到应用程序部署时，大小通常是一个重要因素。使包应用程序的大小尽可能小通常是应用程序开发者的目

根据应用程序的复杂性，运行应用程序只需要运行时的子集。不需要这些未使用的运行时部分，可以从打包的应用程序中进行剪裁。

NOTE

剪裁是 .NET Core 3.1 中的实验性功能，只能用于独立发布的应用程序。

剪裁应用程序

下面的示例演示如何使用 `dotnet publish` 命令剪裁应用程序：

```
dotnet publish -c Release -r win10-x64 --self-contained true /p:PublishSingleFile=false /p:PublishTrimmed=true
```

剪裁功能的工作方式是检查应用程序二进制文件，以发现和生成所需运行时程序集的关系图。未引用的其余运行时程序集会被剪裁。

使用反射时的剪裁问题

在某些情况下，剪裁功能无法检测到引用。例如，使用反射的应用程序可能会遇到此问题，因为只有在运行时才知道程序集的依赖项。

仅当反射使用依赖于未直接引用的运行时程序集时，剪裁才会成为问题。请记住，应用程序代码可能未直接使用反射，但是所引用的第三方程序集可能在使用反射。

当代码通过反射引用某个 API，而你不希望链接器剪裁包含该 API 的程序集时，可以修改项目文件以从剪裁过程中排除该程序集。下面的示例演示如何防止剪裁名为 `System.Security` 的程序集：

```
<ItemGroup>
  <TrimmerRootAssembly Include="System.Security" />
</ItemGroup>
```

请参阅

- [.NET Core 应用程序部署](#)

运行时包存储区

2020/3/30 • [Edit Online](#)

自 .NET Core 2.0 起，可以根据目标环境中已知的一组包来打包和部署应用程序。优点是部署速度更快、磁盘空间使用更少，并可以在某些情况下提升启动性能。

此功能实现为运行时包存储区，这是包在磁盘上的存储目录（通常情况下，在 macOS/Linux 上是 `/usr/local/share/dotnet/store`，在 Windows 上是 `C:/Program Files/dotnet/store`）。此目录下有各个体系结构和目标框架的子目录。文件布局类似于[磁盘上的 NuGet 资产布局](#)：

```
\dotnet
  \store
    \x64
      \netcoreapp2.0
        \microsoft.applicationinsights
        \microsoft.aspnetcore
        ...
    \x86
      \netcoreapp2.0
        \microsoft.applicationinsights
        \microsoft.aspnetcore
        ...
```

目标清单 文件列出了运行时包存储区中的包。开发者可以在发布应用程序时以此清单为目标。目标清单通常是由目标生产环境的所有者提供。

准备运行时环境

运行时环境管理员可以生成运行时包存储区和相应的目标清单，从而优化应用程序，以加快部署速度并释放磁盘空间。

第一步是创建包存储区清单，用于列出运行时包存储区中的包。此文件格式与项目文件格式 (csproj) 兼容。

```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <PackageReference Include="NUGET_PACKAGE" Version="VERSION" />
    <!-- Include additional packages here -->
  </ItemGroup>
</Project>
```

示例

下面的示例包存储区清单 (packages.csproj) 用于将 [Newtonsoft.Json](#) 和 [Moq](#) 添加到运行时包存储区：

```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="10.0.3" />
    <PackageReference Include="Moq" Version="4.7.63" />
  </ItemGroup>
</Project>
```

通过执行 `dotnet store` 并指定包存储区清单、运行时和框架，预配运行时包存储区：

```
dotnet store --manifest <PATH_TO_MANIFEST_FILE> --runtime <RUNTIME_IDENTIFIER> --framework <FRAMEWORK>
```

示例

```
dotnet store --manifest packages.csproj --runtime win10-x64 --framework netcoreapp2.0 --framework-version 2.0.0
```

可以将多个目标包存储区清单路径传递到一个 `dotnet store` 命令，具体方法是在此命令中重复指定选项和路径。

默认情况下，命令输出是在用户配置文件的 `.dotnet/store` 子目录下生成包存储区。可以使用 `--output <OUTPUT_DIRECTORY>` 选项指定其他位置。存储区的根目录包含目标清单 `artifact.xml` 文件。此文件可供下载。如果应用程序创建者在发布时以这个存储区为目标，可以使用此文件。

示例

运行上一示例后生成以下 `artifact.xml` 文件。请注意，由于 `Castle.Core` 是 `Moq` 的依赖项，因此 `artifacts.xml` 清单文件会自动包含并显示它。

```
<StoreArtifacts>
  <Package Id="Newtonsoft.Json" Version="10.0.3" />
  <Package Id="Castle.Core" Version="4.1.0" />
  <Package Id="Moq" Version="4.7.63" />
</StoreArtifacts>
```

根据目标清单发布应用程序

如果磁盘上有目标清单文件，请在使用 `dotnet publish` 命令发布应用程序时指定此文件的路径：

```
dotnet publish --manifest <PATH_TO_MANIFEST_FILE>
```

示例

```
dotnet publish --manifest manifest.xml
```

将生成的已发布应用程序部署到包含目标清单中所述包的环境内。如果不这样做，应用程序便无法启动。

发布应用程序时，通过重复指定选项和路径（例如，`--manifest manifest1.xml --manifest manifest2.xml`），可以指定多个目标清单。这样，应用程序会进行剪裁，依据为提供给命令的目标清单文件中指定的包并集。

在项目文件中指定目标清单

除了使用 `dotnet publish` 命令指定目标清单之外，还可以在项目文件中将目标清单指定为 `TargetManifestFiles><` 标记下的路径分号分隔列表。

```
<PropertyGroup>
  <TargetManifestFiles>manifest1.xml;manifest2.xml</TargetManifestFiles>
</PropertyGroup>
```

仅在应用程序的目标环境已知的情况下（如 .NET Core 项目），才在项目文件中指定目标清单。开放源代码项目的情况有所不同。开放源代码项目的用户通常将项目部署到不同的生产环境。这些生产环境通常都预安装了各组不同的包。在这样的环境中，不能对目标清单作出假设，所以应使用 `--manifest` `dotnet publish` 的选项。

ASP.NET Core 隐式存储区

ASP.NET Core 隐式存储仅适用于 ASP.NET Core 2.0。我们强烈建议应用程序使用 ASP.NET Core 2.1 及更高版本，这些版本不使用隐式存储。ASP.NET Core 2.1 及更高版本使用共享框架。

当 ASP.NET Core 应用程序部署为[从属框架部署 \(FDD\)](#) 应用程序时，应用程序会隐式使用运行时包存储区功能。

`Microsoft.NET.Sdk.Web` 中的目标包括引用目标系统上的隐式包存储区的清单。此外，如果 FDD 应用程序依赖 `Microsoft.AspNetCore.All` 包，则会生成仅包含应用程序及其资产的已发布应用程序，而不是 `Microsoft.AspNetCore.All` 元包中列出的包。假定这些包都位于目标系统上。

安装 .NET Core SDK 后，便会在主机上安装运行时包存储区。其他安装程序可能会提供运行时包存储区，包括 .NET Core SDK 的 Zip/tarball 安装、`apt-get`、Red Hat Yum、.NET Core Windows Server Hosting 捆绑包和手动运行时包存储区安装。

部署[从属框架部署 \(FDD\)](#) 应用程序时，请确保目标环境中已安装 .NET Core SDK。如果应用程序部署环境中未安装 ASP.NET Core，可以在项目文件中指定将 `PublishWithAspNetCoreTargetManifest` < 设置为 `false`，从而选择退出隐式存储区，如以下示例所示：

```
<PropertyGroup>
  <PublishWithAspNetCoreTargetManifest>false</PublishWithAspNetCoreTargetManifest>
</PropertyGroup>
```

NOTE

对于[独立部署 \(SCD\)](#) 应用程序，假定目标系统不一定包含所需的清单包。因此，对于 SCD 应用程序，不能将 `PublishWithAspNetCoreTargetManifest` < 设置为 `true`。

如果使用部署中的清单依赖项(程序集位于 bin 文件夹中)部署应用程序，运行时包存储区不会 在主机上用于相应程序集。将使用 bin 文件夹程序集，无论它是否位于主机上的运行时包存储区中。

清单中指明的依赖项版本必须与运行时包存储区中的依赖项版本一致。如果目标清单与运行时包存储区中的依赖项版本不一致，并且应用程序的部署中没有包的相应版本，那么应用程序将无法启动。例外情况包括，为运行时包存储区程序集调用的目标清单名称，这有助于排查不一致问题。

如果在发布时 部署发生剪裁，只有指明的特定版本清单包，才不会出现在已发布的输出中。主机上必须有指明的包版本，应用程序才能启动。

另请参阅

- [dotnet-publish](#)
- [dotnet-store](#)

.NET 和 Docker 简介

2020/3/18 • [Edit Online](#)

.NET Core 可以在 Docker 容器中轻松运行。容器提供一种轻量级方法，用于将应用程序与主机系统的其他组件分隔，以便仅共享内核，并使用提供给应用程序的资源。如果你不熟悉 Docker，强烈建议通读 Docker 的[概述文档](#)。

若要详细了解如何安装 Docker，请参阅 [Docker 桌面：社区版](#)的下载页面。

Docker 基础知识

首先来熟悉几个概念。Docker 客户端具有可用于管理映像和容器的 CLI。如前文所述，应花时间通读 [Docker 概述](#) 文档。

映像

映像是构成容器基础的文件系统更改的有序集合。映像不具有状态，并且是只读的。大多情况下，一个映像以另一个映像为基础，但具有一些自定义设置。例如，为应用程序创建新映像时，可将其基于已包含 .NET Core 运行时的现有映像。

由于容器是从映像创建的，因此，映像具有一组在容器启动时运行的运行参数（例如启动可执行文件）。

容器

容器是映像的可运行实例。生成映像时，部署应用程序和依赖项。然后可以实例化多个容器，且每个容器相互独立。每个容器实例具有其自己的文件系统、内存和网络接口。

注册表

容器注册表是映像存储库的集合。映像可以基于注册表映像。可以直接从注册表中的映像创建容器。[Docker 容器、映像和注册表之间的关系](#)是构建和生成容器化应用程序或微服务时的一个重要概念。此方法大大缩短了开发和部署之间的时间。

Docker 具有一个托管在 [Docker 中心](#)的公共注册表，可供用户使用。[.NET Core 相关映像](#)均在 Docker 中心列出。

Microsoft 容器注册表 (MCR) 是 Microsoft 提供的容器映像的官方来源。MCR 构建在 Azure CDN 之上，可提供用于全局复制的映像。但是，MCR 没有面向公众的网站，了解有关 Microsoft 提供的容器映像的主要方法是通过 [Microsoft Docker 中心页面](#)。

Dockerfile

Dockerfile 是定义可创建映像的一组指令的文件。Dockerfile 中的每个指令创建映像中的一个层。大多数情况下，在重新生成映像时，只会重新生成已发生更改的层。可以将 Dockerfile 分发给其他人，便于他们采用你创建映像的方式重新创建一个新映像。尽管可以分发有关如何创建映像的指令，但分发映像的主要方式是将其发布到注册表。

.NET Core 映像

官方 .NET Core Docker 映像发布到 Microsoft 容器注册表 (MCR)，用户可以在 [Microsoft.NET Core Docker 中心存储库](#)中找到这些映像。每个存储库包含 .NET (SDK 或运行时) 和可以使用的操作系统的不同组合的映像。

Microsoft 提供适合特定场景的映像。例如，[ASP.NET Core 存储库](#)提供针对在生产环境中运行 ASP.NET Core 应用生成的映像。

Azure 服务

各种 Azure 服务都支持容器。为应用程序创建 Docker 映像并将其部署到以下服务之一：

- [Azure Kubernetes 服务 \(AKS\)](#)
使用 kubernetes 缩放和编排 Linux 容器。
- [Azure 应用服务](#)
在 PaaS 环境中使用 Linux 容器部署 Web 应用或 API。
- [Azure 容器实例](#)
将容器托管在云中，而不使用任何高级管理服务。
- [Azure Batch](#)
使用容器运行重复的计算作业。
- [Azure Service Fabric](#)
使用 Windows Server 容器直接迁移 .NET 应用程序并将其现代化为微服务。
- [Azure 容器注册表](#)
在所有类型的 Azure 部署中存储和管理容器映像。

后续步骤

- [了解如何使 .NET Core 应用程序容器化。](#)
- [了解如何容器化 ASP.NET Core 应用程序。](#)
- [试用“学习 ASP.NET Core 微服务”教程。](#)
- [了解 Visual Studio 中的容器工具](#)

教程：使 .NET Core 应用程序容器化

2020/4/2 • [Edit Online](#)

本教程介绍如何生成包含 .NET Core 应用程序的 Docker 映像。此映像可用于为本地开发环境、私有云或公有云创建容器。

你将了解如何：

- 创建并发布简单的 .NET Core 应用
- 创建并配置用于 .NET Core 的 Dockerfile
- 生成 Docker 映像
- 创建并运行 Docker 容器

你将了解用于 .NET Core 应用的 Docker 容器生成和部署任务。Docker 平台 使用 Docker 引擎 快速生成应用，并将应用打包为 Docker 映像。这些映像采用 Dockerfile 格式编写，以供在分层容器中部署和运行。

WARNING

■ ASP.NET Core ■ 如果使用的是 ASP.NET Core，请参阅教程[了解如何容器化 ASP.NET Core 应用程序](#)。

先决条件

安装以下必备组件：

- [.NET Core 3.1 SDK](#)

如果已安装 .NET Core，请使用 `dotnet --info` 命令来确定使用的是哪个 SDK。

- [Docker 社区版](#)

- Dockerfile 和 .NET Core 示例应用的临时工作文件夹。在本教程中，`docker-working` 用作工作文件夹的名称。

创建 .Net Core 应用

需要有可供 Docker 容器运行的 .NET Core 应用。打开终端、创建工作文件夹（如果尚没有），然后进入该文件夹。

在工作文件夹中，运行下面的命令，在名为“app”的子目录中新建一个项目：

```
dotnet new console -o app -n myapp
```

文件夹树将如下所示：

```
docker-working
|
└── app
    ├── myapp.csproj
    └── Program.cs
|
└── obj
    ├── myapp.csproj.nuget.cache
    ├── myapp.csproj.nuget.dgspec.json
    ├── myapp.csproj.nuget.g.props
    ├── myapp.csproj.nuget.g.targets
    └── project.assets.json
```

`dotnet new` 命令会新建一个名为“应用”的文件夹，并生成一个“Hello World”应用。进入“应用”文件夹并运行命令 `dotnet run`。输出如下：

```
> dotnet run
Hello World!
```

默认模板创建应用，此应用先打印输出到终端，再退出。本教程将使用无限循环的应用。在文本编辑器中，打开“Program.cs”文件。它应如以下代码所示：

```
using System;

namespace myapp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

将此文件替换为以下每秒计数一次的代码：

```
using System;

namespace myapp
{
    class Program
    {
        static void Main(string[] args)
        {
            var counter = 0;
            var max = args.Length != 0 ? Convert.ToInt32(args[0]) : -1;
            while (max == -1 || counter < max)
            {
                counter++;
                Console.WriteLine($"Counter: {counter}");
                System.Threading.Tasks.Task.Delay(1000).Wait();
            }
        }
    }
}
```

保存此文件，并使用 `dotnet run` 再次测试程序。请注意，此应用无限期运行。使用取消命令 `CTRL+C` 可以停止运行。输出如下：

```
> dotnet run  
Counter: 1  
Counter: 2  
Counter: 3  
Counter: 4  
^C
```

如果你在命令行中向应用传递一个数字，它就只会计数到这个数字，然后退出。试一试用 `dotnet run -- 5` 计数到 5。

NOTE

-- 之后的参数都不传递到 `dotnet run` 命令，而是传递到你的应用程序。

发布 .Net Core 应用

请先发布 .NET Core 应用，再将它添加到 Docker 映像。需确保容器在启动时运行应用的发布版本。

在工作文件夹中，进入包含示例源代码的“应用”文件夹，并运行以下命令：

```
dotnet publish -c Release
```

此命令将应用编译到“发布”文件夹中。从工作文件夹到“发布”文件夹的路径应为

```
.\app\bin\Release\netcoreapp3.1\publish\
```

在“应用”文件夹中获取“发布”文件夹的目录清单，以验证 myapp.dll 文件是否已创建。

```
> dir bin\Release\netcoreapp3.1\publish  
  
Directory: C:\docker-working\app\bin\Release\netcoreapp3.1\publish  
  
01/09/2020 11:41 AM <DIR> .  
01/09/2020 11:41 AM <DIR> ..  
01/09/2020 11:41 AM 407 myapp.deps.json  
01/09/2020 12:15 PM 4,608 myapp.dll  
01/09/2020 12:15 PM 169,984 myapp.exe  
01/09/2020 12:15 PM 736 myapp.pdb  
01/09/2020 11:41 AM 154 myapp.runtimeconfig.json
```

如果使用的是 Linux 或 macOS，请使用 `ls` 命令获取目录列表，并验证是否已创建 myapp 文件。

```
me@DESKTOP:/docker-working/app$ ls bin/Release/netcoreapp3.1/publish  
myapp.deps.json myapp.dll myapp.pdb myapp.runtimeconfig.json
```

创建 Dockerfile

`docker build` 命令使用 Dockerfile 文件来创建容器映像。此文件是名为“Dockerfile”的文本文件，它没有扩展名。

在终端中，导航到你在启动时创建的工作文件夹的目录。在工作文件夹中创建名为“Dockerfile”的文件，在文本编辑器中打开它。根据要容器化的应用程序类型，选择 ASP.NET Core 运行时或 .NET Core 运行时。如有疑问，请选择包含 .NET Core 运行时的 ASP.NET Core 运行时。本教程将使用 ASP.NET Core 运行时映像，但在前面部分中创建的应用是 .NET Core 应用。

- ASP.NET Core 运行时

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1
```

- .NET Core 运行时

```
FROM mcr.microsoft.com/dotnet/core/runtime:3.1
```

`FROM` 命令指示 Docker 从指定存储库中拉取标记为“3.1”的映像。请确保拉取的运行时版本与 SDK 面向的运行时一致。例如，在上一节中创建的应用使用的是 .NET Core 3.1 SDK，并且 Dockerfile 中引用的基本映像标记有 3.1。

保存 Dockerfile 文件。工作文件夹的目录结果应如下所示。为节省本文的空间，删掉了一些更高级别的文件和文件夹：

```
docker-working
| Dockerfile
|
└─ app
    | myapp.csproj
    | Program.cs
    |
    └─ bin
        └─ Release
            └─ netcoreapp3.1
                └─ publish
                    myapp.deps.json
                    myapp.exe
                    myapp.dll
                    myapp.pdb
                    myapp.runtimeconfig.json
    └─ obj
```

在终端中运行以下命令：

```
docker build -t myimage -f Dockerfile .
```

Docker 会处理 Dockerfile 中的每一行。`docker build` 命令中的 `.` 指示 Docker 在当前文件夹中查找 Dockerfile。此命令生成映像，并创建指向相应映像的本地存储库“myimage”。在此命令完成后，运行 `docker images` 以列出已安装的映像：

```
> docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
myimage              latest   38db0eb8f648  4 weeks ago  346MB
mcr.microsoft.com/dotnet/core/aspnet  3.1     38db0eb8f648  4 weeks ago  346MB
```

请注意，两个映像共用相同的“IMAGE ID”值。两个映像使用的 ID 值相同是因为，Dockerfile 中的唯一命令是在现有映像的基础之上生成新映像。接下来，在 Dockerfile 中添加两个命令。两个命令都新建映像层，最后一个命令表示 myimage 存储库条目指向的映像。

```
COPY app/bin/Release/netcoreapp3.1/publish/ app/
ENTRYPOINT ["dotnet", "app/myapp.dll"]
```

`COPY` 命令指示 Docker 将计算机上的指定文件夹复制到容器中的文件夹。在此示例中，“发布”文件夹被复制到容器中的“应用”文件夹。

下一个命令 `ENTRYPOINT` 指示 Docker 将容器配置为可执行文件运行。在容器启动时，`ENTRYPOINT` 命令运行。当此命令结束时，容器也会自动停止。

在终端中，运行 `docker build -t myimage -f Dockerfile .`；在此命令完成后，运行 `docker images`。

```
> docker build -t myimage -f Dockerfile .
Sending build context to Docker daemon 1.624MB
Step 1/3 : FROM mcr.microsoft.com/dotnet/core/aspnet:3.1
--> 38db0eb8f648
Step 2/3 : COPY app/bin/Release/netcoreapp3.1/publish/ app/
--> 37873673e468
Step 3/3 : ENTRYPOINT ["dotnet", "app/myapp.dll"]
--> Running in d8deb7b3aa9e
Removing intermediate container d8deb7b3aa9e
--> 0d602ca35c1d
Successfully built 0d602ca35c1d
Successfully tagged myimage:latest

> docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
myimage              latest   0d602ca35c1d  4 seconds ago  346MB
mcr.microsoft.com/dotnet/core/aspnet   3.1     38db0eb8f648  4 weeks ago   346MB
```

Dockerfile 中的每个命令生成了一个层，并创建了“IMAGE ID”。最终“IMAGE ID”是“ddcc6646461b”（你的 ID 会有所不同），接下来在此映像的基础之上创建容器。

创建容器

至此，已有包含应用的映像，可以创建容器了。可以通过两种方式来创建容器。首先，新建已停止的容器。

```
> docker create myimage
ceda87b219a4e55e9ad5d833ee1a7ea4da21b5ea7ce5a7d08f3051152e784944
```

上面的 `docker create` 命令在 `myimage` 映像的基础之上创建容器。此命令的输出显示已创建容器的“CONTAINER ID”（你的 ID 会有所不同）。若要查看所有容器的列表，请使用 `docker ps -a` 命令：

```
> docker ps -a
CONTAINER ID        IMAGE           COMMAND       CREATED        STATUS         PORTS
NAMES
ceda87b219a4        myimage        "dotnet app/myapp.dll"   4 seconds ago   Created
gallant_lehmann
```

管理容器

每个容器都分配有随机名称，可用来引用相应容器实例。例如，自动创建的容器选择了名称“`gallant_lehmann`”（你的名称会有所不同），此名称可用于启动容器。可以使用 `docker create --name` 参数将自动名称替代为特定名称。

下面的示例使用 `docker start` 命令来启动容器，然后使用 `docker ps` 命令仅显示正在运行的容器：

```
> docker start gallant_lehmann
gallant_lehmann

> docker ps
CONTAINER ID        IMAGE           COMMAND       CREATED        STATUS         PORTS
NAMES
ceda87b219a4        myimage        "dotnet app/myapp.dll"   7 minutes ago   Up 8 seconds
gallant_lehmann
```

同样, `docker stop` 命令会停止容器。下面的示例使用 `docker stop` 命令来停止容器, 然后使用 `docker ps` 命令来显示未在运行的容器:

```
> docker stop gallant_lehmann  
gallant_lehmann  
  
> docker ps  
CONTAINER ID        IMAGE               COMMAND       CREATED          STATUS          PORTS     NAMES
```

连接到容器

在容器运行后, 可以连接到它来查看输出。使用 `docker start` 和 `docker attach` 命令, 启动容器并查看输出流。在此示例中, `CTRL + C` 键用于从正在运行的容器中分离出来。此键其实是结束容器中的进程, 进而停止容器。`--sig-proxy=false` 参数可确保 `Ctrl + C` 不停止容器中的进程。

从容器中分离出来后重新连接, 以验证它是否仍在运行和计数。

```
> docker start gallant_lehmann  
gallant_lehmann  
  
> docker attach --sig-proxy=false gallant_lehmann  
Counter: 7  
Counter: 8  
Counter: 9  
^C  
  
> docker attach --sig-proxy=false gallant_lehmann  
Counter: 17  
Counter: 18  
Counter: 19  
^C
```

删除容器

就本文而言, 你不希望存在不执行任何操作的容器。删除前面创建的容器。如果容器正在运行, 停止容器。

```
> docker stop gallant_lehmann
```

下面的示例列出了所有容器。然后, 它使用 `docker rm` 命令来删除容器, 并再次检查是否有任何正在运行的容器。

```
> docker ps -a  
CONTAINER ID        IMAGE               COMMAND       CREATED          STATUS          PORTS     NAMES  
ceda87b219a4        myimage            "dotnet app/myapp.dll"   19 minutes ago   Exited  
gallant_lehmann  
  
> docker rm gallant_lehmann  
gallant_lehmann  
  
> docker ps -a  
CONTAINER ID        IMAGE               COMMAND       CREATED          STATUS          PORTS     NAMES
```

单次运行

Docker 提供了 `docker run` 命令, 用于将容器作为单一命令进行创建和运行。使用此命令, 无需依次运行 `docker create` 和 `docker start`。另外, 还可以将此命令设置为, 在容器停止时自动删除容器。例如, 使用 `docker run -it --rm` 可以执行两项操作, 先自动使用当前终端连接到容器, 再在容器完成时删除容器:

```
> docker run -it --rm myimage
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
^C
```

使用 `docker run -it`，`CTRL + C` 命令会停止在容器中运行的进程，进而停止容器。由于提供了 `--rm` 参数，因此在进程停止时自动删除容器。验证它是否存在：

```
> docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS      NAMES
```

更改 ENTRYPPOINT

使用 `docker run` 命令，还可以修改 Dockerfile 中的 `ENTRYPOINT` 命令，并运行其他内容，但只能针对相应容器。例如，使用以下命令来运行 `bash` 或 `cmd.exe`。根据需要，编辑此命令。

Windows

在本例中，`ENTRYPOINT` 更改为 `cmd.exe`。通过按下 `CTRL+C` 来结束进程并停止容器。

```
> docker run -it --rm --entrypoint "cmd.exe" myimage

Microsoft Windows [Version 10.0.17763.379]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\>dir
Volume in drive C has no label.
Volume Serial Number is 3005-1E84

Directory of C:\

04/09/2019  08:46 AM    <DIR>          app
03/07/2019  10:25 AM           5,510 License.txt
04/02/2019  01:35 PM    <DIR>          Program Files
04/09/2019  01:06 PM    <DIR>          Users
04/02/2019  01:35 PM    <DIR>          Windows
               1 File(s)       5,510 bytes
               4 Dir(s)  21,246,517,248 bytes free

C:\>^C
```

Linux

在本例中，`ENTRYPOINT` 更改为 `bash`。通过运行 `quit` 命令来结束进程并停止容器。

```
root@user:~# docker run -it --rm --entrypoint "bash" myimage
root@8515e897c893:/# ls app
myapp.deps.json  myapp.dll  myapp.pdb  myapp.runtimeconfig.json
root@8515e897c893:/# exit
exit
```

重要命令

Docker 有许多不同的命令，可用于执行你要对容器和映像执行的操作。下面这些 Docker 命令对于管理容器来说至关重要：

- `docker build`

- [docker run](#)
- [docker ps](#)
- [docker stop](#)
- [docker rm](#)
- [docker rmi](#)
- [docker image](#)

清理资源

在本教程中，你创建了容器和映像。如果需要，请删除这些资源。以下命令可用于

1. 列出所有容器

```
> docker ps -a
```

2. 停止正在运行的容器。`CONTAINER_NAME` 表示自动分配给容器的名称。

```
> docker stop CONTAINER_NAME
```

3. 删除容器

```
> docker rm CONTAINER_NAME
```

接下来，删除计算机上不再需要的任何映像。依次删除 Dockerfile 创建的映像，以及 Dockerfile 所依据的 .NET Core 映像。可以使用 IMAGE ID 或 REPOSITORY:TAG 格式字符串。

```
docker rmi myimage:latest  
docker rmi mcr.microsoft.com/dotnet/core/aspnet:3.1
```

使用 `docker images` 命令来列出已安装的映像。

NOTE

映像文件可能很大。通常情况下，需要删除在测试和开发应用期间创建的临时容器。如果计划在相应运行时的基础之上生成其他映像，通常会将基础映像与运行时一同安装。

后续步骤

- [了解如何容器化 ASP.NET Core 应用程序。](#)
- [试学“ASP.NET Core 微服务”教程。](#)
- [查看支持容器的 Azure 服务。](#)
- [了解 Dockerfile 命令。](#)
- [了解用于 Visual Studio 的容器工具](#)

.NET Core 运行时配置设置

2020/3/18 • [Edit Online](#)

.NET Core 支持使用配置文件和环境变量在运行时配置 .NET Core 应用程序的行为。如果出现以下情况，则运行时配置是一个不错的选择：

- 你不拥有或控制应用程序的源代码，因此无法以编程方式对其进行配置。
- 应用程序的多个实例在单个系统上同时运行，并且你想要将每个实例配置为获得最佳性能。

NOTE

本文档正在编写中。如果你注意到此处提供的信息不完整或不准确，可以[创建一个问题](#)告知我们，或[提交拉取请求](#)以解决问题。要了解如何提交 dotnet/docs 存储库的拉取请求，请参阅[参与者指南](#)。

.NET Core 提供以下用于配置运行时应用程序行为的机制：

- [runtimeconfig.json 文件](#)
- [MSBuild 属性](#)
- [环境变量](#)

某些配置值还可以通过调用 [ApplicationContext.SetSwitch](#) 方法以编程方式进行设置。

文档此部分的文章按类别组织，例如[调试](#)和[垃圾回收](#)。如果适用，将显示 runtimeconfig.json 文件、MSBuild 属性、环境变量的配置选项；对于 .NET Framework 项目，还会显示 app.config 文件的配置选项以便交叉引用。

runtimeconfig.json

构建项目时，将在输出目录中生成 `[appname].runtimeconfig.json` 文件。如果项目文件所在的文件夹中存在 `runtimeconfig.template.json` 文件，它包含的任何配置选项都将合并到 `[appname].runtimeconfig.json` 文件中。如果自行构建应用，请将所有配置选项放在 `runtimeconfig.template.json` 文件中。如果只是运行应用，请将其直接插入 `[appname].runtimeconfig.template.json` 文件中。

NOTE

后续生成中将覆盖 `[appname].runtimeconfig.template.json` 文件。

在 runtimeconfig.json 文件的 configProperties 部分指定运行时配置选项。此部分包含窗体：

```
"configProperties": {  
    "config-property-name1": "config-value1",  
    "config-property-name2": "config-value2"  
}
```

示例 `[appname].runtimeconfig.template.json` 文件

如果要将这些选项放在输出 JSON 文件中，请将它们嵌套在 `runtimeOptions` 属性下。

```
{  
    "runtimeOptions": {  
        "tfm": "netcoreapp3.1",  
        "framework": {  
            "name": "Microsoft.NETCore.App",  
            "version": "3.1.0"  
        },  
        "configProperties": {  
            "System.GC.Concurrent": false,  
            "System.Threading.ThreadPool.MinThreads": 4,  
            "System.Threading.ThreadPool.MaxThreads": 25  
        }  
    }  
}
```

示例 `runtimeconfig.template.json` 文件

如果要将这些选项放在模板 JSON 文件中，请省略 `runtimeOptions` 属性。

```
{  
    "configProperties": {  
        "System.GC.Concurrent": false,  
        "System.Threading.ThreadPool.MinThreads": "4",  
        "System.Threading.ThreadPool.MaxThreads": "25"  
    }  
}
```

MSBuild 属性

可以使用 SDK 样式 .NET Core 项目的 `.csproj` 或 `.vbproj` 文件中的 MSBuild 属性设置某些运行时配置选项。MSBuild 属性优先于在 `runtimeconfig.template.json` 文件中设置的选项。它们还会覆盖生成时在 `[appname].runtimeconfig.json` 文件中设置的任何选项。

下面是一个示例 SDK 样式项目文件，其中包含用于配置运行时行为的 MSBuild 属性：

```
<Project Sdk="Microsoft.NET.Sdk">  
  
    <PropertyGroup>  
        <OutputType>Exe</OutputType>  
        <TargetFramework>netcoreapp3.1</TargetFramework>  
    </PropertyGroup>  
  
    <PropertyGroup>  
        <ConcurrentGarbageCollection>false</ConcurrentGarbageCollection>  
        <ThreadPoolMinThreads>4</ThreadPoolMinThreads>  
        <ThreadPoolMaxThreads>25</ThreadPoolMaxThreads>  
    </PropertyGroup>  
  
</Project>
```

用于配置运行时行为的 MSBuild 属性记录在每个区域各自的文章中，例如[垃圾回收](#)。

环境变量

环境变量可用于提供一些运行时配置信息。指定为环境变量的配置旋钮通常有 `COMPlus_` 前缀。

可以使用 Windows 控制面板、命令行或通过在 Windows 和 Unix 系统上调用 `Environment.SetEnvironmentVariable(String, String)` 方法以编程方式定义环境变量。

下面的示例演示如何在命令行中设置环境变量：

```
# Windows
set COMPlus_GCRetainVM=1

# Powershell
$env:COMPlus_GCRetainVM="1"

# Unix
export COMPlus_GCRetainVM=1
```

用于编译的运行时配置选项

2020/4/13 • [Edit Online](#)

分层编译

- 配置实时 (JIT) 编译器是否使用[分层编译](#)。分层编译将方法转换到两个层级：
 - 第一层可以更快速地生成代码([快速 JIT](#))或加载预编译的代码 ([ReadyToRun](#))。
 - 第二层在后台生成优化的代码(“[优化 JIT](#)”)。
- 在 .NET Core 3.0 及更高版本中，默认情况下已启用分层编译。
- 在 .NET Core 2.1 和 2.2 中，默认情况下已禁用分层编译。
- 有关详细信息，请参阅[分层编译指南](#)。

	III	I
runtimesconfig.json	<code>System.Runtime.TieredCompilation</code>	<input checked="" type="checkbox"/> true - 启用 <input type="checkbox"/> false - 禁用
MSBuild 属性	<code>TieredCompilation</code>	<input checked="" type="checkbox"/> true - 启用 <input type="checkbox"/> false - 禁用
■	<code>COMPlus_TieredCompilation</code>	<input checked="" type="checkbox"/> 1 - 启用 <input type="checkbox"/> 0 - 禁用

示例

runtimesconfig.json 文件：

```
{  
    "runtimeOptions": {  
        "configProperties": {  
            "System.Runtime.TieredCompilation": false  
        }  
    }  
}
```

项目文件：

```
<Project Sdk="Microsoft.NET.Sdk">  
  
<PropertyGroup>  
    <TieredCompilation>false</TieredCompilation>  
</PropertyGroup>  
  
</Project>
```

快速 JIT

- 配置 JIT 编译器是否使用[快速 JIT](#)。对于不包含循环且不可使用预编译代码的方法，[快速 JIT](#) 可以更快完成编译，但不会进行优化。
- 启用[快速 JIT](#)会缩短启动时间，但可能会生成性能下降的代码。例如，代码可能会使用更多堆栈空间、分配更多内存并以更慢的速度运行。

- 如果禁用了快速 JIT 但启用了[分层编译](#), 则只有预编译的代码参与分层编译。如果未使用 ReadyToRun 预编译方法, 则 JIT 行为与禁用[分层编译](#)时相同。
- 在 .NET Core 3.0 及更高版本中, 默认启用快速 JIT。
- 在 .NET Core 2.1 和 2.2 中, 默认禁用快速 JIT。

	III	I
runtimeconfig.json	System.Runtime.TieredCompilation.QuickJit <input checked="" type="checkbox"/> true - 启用 <input type="checkbox"/> false - 禁用	
MSBuild 属性	TieredCompilationQuickJit	<input checked="" type="checkbox"/> true - 启用 <input type="checkbox"/> false - 禁用
■	COMPlus_TC_QuickJit	<input checked="" type="checkbox"/> 1 - 启用 <input type="checkbox"/> 0 - 禁用

示例

runtimeconfig.json 文件:

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.Runtime.TieredCompilation.QuickJit": false
    }
  }
}
```

项目文件:

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
  <TieredCompilationQuickJit>false</TieredCompilationQuickJit>
</PropertyGroup>

</Project>
```

适用于循环的快速 JIT

- 配置 JIT 编译器是否对包含循环的方法使用快速 JIT。
- 启用适用于循环的快速 JIT 可以提高启动性能。不过, 在优化程度较低的代码中, 长时间运行的循环可能会停滞较长时间。
- 如果禁用[快速 JIT](#), 则此设置不起作用。
- 默认:禁用 (`false`)。

	III	I
runtimeconfig.json	System.Runtime.TieredCompilation.QuickJit <input checked="" type="checkbox"/> false - 禁用 <input type="checkbox"/> true - 启用	
MSBuild 属性	TieredCompilationQuickJitForLoops	<input checked="" type="checkbox"/> false - 禁用 <input type="checkbox"/> true - 启用

	III	I
■	COMPlus_TC_QuickJitForLoops	<input type="radio"/> 0 - 禁用 <input checked="" type="radio"/> 1 - 启用

示例

runtimeconfig.json 文件：

```
{  
    "runtimeOptions": {  
        "configProperties": {  
            "System.Runtime.TieredCompilation.QuickJitForLoops": false  
        }  
    }  
}
```

项目文件：

```
<Project Sdk="Microsoft.NET.Sdk">  
  
<PropertyGroup>  
    <TieredCompilationQuickJitForLoops>false</TieredCompilationQuickJitForLoops>  
</PropertyGroup>  
  
</Project>
```

ReadyToRun

- 配置 .NET Core 运行时是否要为具有可用 ReadyToRun 数据的映像使用预编译代码。如果禁用此选项，会强制运行时对框架代码进行 JIT 编译。
- 有关详细信息，请参阅 [ReadyToRun](#)。
- 默认：启用 (1)。

	III	I
■	COMPlus_ReadyToRun	<input checked="" type="radio"/> 1 - 启用 <input type="radio"/> 0 - 禁用

用于调试和分析的运行时配置选项

2020/3/18 • [Edit Online](#)

启用诊断

- 配置是否启用调试器、探查器和 EventPipe 诊断。
- 默认: 启用 (1)。

	III	I
runtimeconfig.json	不可用	不可用
■	<code>COMPlus_EnableDiagnostics</code>	<input checked="" type="checkbox"/> - 启用 <input type="checkbox"/> - 禁用

启用分析

- 配置是否为当前正在运行的进程启用分析。
- 默认: 禁用 (0)。

	III	I
runtimeconfig.json	不可用	不可用
■	<code>CORECLR_ENABLE_PROFILING</code>	<input type="checkbox"/> - 禁用 <input checked="" type="checkbox"/> - 启用

探查器 GUID

- 指定要加载到当前正在运行的进程中的探查器 GUID。

	III	I
runtimeconfig.json	不可用	不可用
■	<code>CORECLR_PROFILER</code>	string-guid

探查器位置

- 指定要加载到当前正在运行的进程(或 32 位/64 位进程)的探查器 DLL 路径。
- 如果设置了多个变量，则优先使用指定位数的变量。它们指定要加载的探查器的位数。
- 有关详细信息，请参阅 [Finding the profiler library](#)(查找探查器库)。

	III	I
■	<code>CORECLR_PROFILER_PATH</code>	string-path

	III	I
■	CORECLR_PROFILER_PATH_32	string-path
■	CORECLR_PROFILER_PATH_64	string-path

写入 Perf 映射

- 允许或禁止在 Linux 系统上写入 /tmp/perf-\$pid.map。
- 默认:禁用()。

	III	I
runtimeconfig.json	不可用	不可用
■	COMPlus_PerfMapEnabled	<input type="checkbox"/> 0 - 禁用 <input checked="" type="checkbox"/> 1 - 启用

性能日志标记

- 将 COMPlus_PerfMapEnabled 设置为 1 时, 允许或禁止以性能日志中的标记接受和忽略指定信号。
- 默认:禁用()。

	III	I
runtimeconfig.json	不可用	不可用
■	COMPlus_PerfMapIgnoreSignal	<input type="checkbox"/> 0 - 禁用 <input checked="" type="checkbox"/> 1 - 启用

用于垃圾回收的运行时配置选项

2020/3/18 • [Edit Online](#)

此页包含有关可在运行时更改的垃圾回收器 (GC) 设置的信息。如果你要尝试让正在运行的应用达到最佳性能，请考虑使用这些设置。然而，在特定情况下，默认值为大多数应用程序提供最佳性能。

设置在此页上被排入组中。每个组内的设置通常彼此结合使用以实现特定的结果。

NOTE

- 这些设置也可以在应用运行时由应用动态地进行更改，因此，你设置的任何运行时设置都可能会被覆盖。
- 某些设置（如[延迟级别](#)）通常仅在设计时通过 API 进行设置。此页面省略了此类设置。
- 对于数值，请对 runtimeconfig.json 文件中的设置使用十进制表示法，而对环境变量设置使用十六进制表示法。对于十六进制值，可以使用或不使用“0x”前缀来指定它们。

垃圾回收的风格

垃圾回收的两种主要风格是工作站 GC 和服务器 GC。有关两者之间的差异的详细信息，请参阅[垃圾回收的基础知识](#)。

垃圾回收的次要风格是后台垃圾回收和非并发垃圾回收。

使用以下设置，选择垃圾回收的风格：

System.GC.Server/COMPlus_gcServer

- 配置应用程序是使用工作站垃圾回收还是服务器垃圾回收。
- 默认：工作站垃圾回收 (`false`)。

	III	I	
runtimeconfig.json	<code>System.GC.Server</code>	<code>false</code> - 工作站 <code>true</code> - 服务器	.NET Core 1.0
MSBuild 属性	<code>ServerGarbageCollection</code>	<code>false</code> - 工作站 <code>true</code> - 服务器	.NET Core 1.0
■	<code>COMPlus_gcServer</code>	<code>0</code> - 工作站 <code>1</code> - 服务器	.NET Core 1.0
.NET Framework ■ app.config	<code>GCServer</code>	<code>false</code> - 工作站 <code>true</code> - 服务器	

示例

runtimeconfig.json 文件：

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.Server": true
    }
  }
}
```

项目文件：

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
  <ServerGarbageCollection>true</ServerGarbageCollection>
</PropertyGroup>

</Project>
```

System.GC.Concurrent/COMPlus_gcConcurrent

- 配置是否启用后台(并发)垃圾回收。
- 默认：启用(`true`)。
- 有关详细信息，请参阅[后台垃圾回收](#)和[后台服务器垃圾回收](#)。

	■■■	■	■■■■■
runtimeconfig.json	<code>System.GC.Concurrent</code>	<code>true</code> - 后台 GC <code>false</code> - 非并发 GC	.NET Core 1.0
MSBuild 属性	<code>ConcurrentGarbageCollection</code>	<code>true</code> - 后台 GC <code>false</code> - 非并发 GC	.NET Core 1.0
■ ■ ■ ■ ■	<code>COMPlus_gcConcurrent</code>	<code>true</code> - 后台 GC <code>false</code> - 非并发 GC	.NET Core 1.0
.NET Framework ■■■■■ app.config	<code>gcConcurrent</code>	<code>true</code> - 后台 GC <code>false</code> - 非并发 GC	

示例

runtimeconfig.json 文件：

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.Concurrent": false
    }
  }
}
```

项目文件：

```

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

<PropertyGroup>
    <ConcurrentGarbageCollection>false</ConcurrentGarbageCollection>
</PropertyGroup>

</Project>

```

管理资源使用情况

使用此部分中所述的设置，管理垃圾回收器的内存和处理器使用情况。

有关其中某些设置的详细信息，请参阅 [Middle ground between workstation and server GC](#)(服务器和工作站 GC 之间的中间地带)博客条目。

System.GC.HeapCount/COMPlus_GCHepCount

- 限制通过垃圾回收器创建的堆数。
- 仅适用于服务器垃圾回收。
- 如果启用了默认的 GC 处理器关联，堆计数设置会将 `n` 个 GC 堆/线程关联到前 `n` 个处理器。(使用关联掩码或关联范围设置来精确指定要关联的处理器。)
- 如果禁用了 GC 处理器关联，则此设置会限制 GC 堆的数量。
- 有关详细信息，请参阅 [GCHepCount 备注](#)。

	III	I	
<code>runtimeconfig.json</code>	<code>System.GC.HeapCount</code>	十进制值	.NET Core 3.0
■	<code>COMPlus_GCHepCount</code>	十六进制值	.NET Core 3.0
<code>.NET Framework app.config</code>	<code>GCHepCount</code>	十进制值	.NET Framework 4.6.2

示例：

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.HeapCount": 16
    }
  }
}
```

TIP

如果要在 `runtimeconfig.template.json` 中设置该选项，请指定一个十进制值。如果要将选项设置为一个环境变量，请指定一个十六进制值。例如，若要将堆数限制为 16，则该值对于 JSON 文件为 16，对于环境变量则为 0x10 或 10。

System.GC.HeapAffinitizeMask/COMPlus_GCHepAffinitizeMask

- 指定垃圾回收器线程应使用的的确切处理器数。
- 如果通过将 `System.GC.NoAffinitize` 设置为 `true` 禁用了处理器关联，则忽略此设置。
- 仅适用于服务器垃圾回收。
- 该值是一个位掩码，用于定义可用于该进程的处理器。例如，十进制值 1023 或十六进制值 0x3FF 或 3FF(如果使用环境变量)在二进制记数法中为 0011 1111 1111。这指定将使用前 10 个处理器。若要指定接下来使用的

10 个处理器(即处理器 10-19), 请指定一个十进制值 1047552(或十六进制值 0xFFC00 或 FFC00), 它等效于二进制值 1111 1111 1100 0000 0000。

	III	I	
runtimeconfig.json	System.GC.HeapAffinitizeMask	十进制值	.NET Core 3.0
■	COMPlus_GCHeapAffinitizeMask	十六进制值	.NET Core 3.0
.NET Framework app.config	GCHeapAffinitizeMask	十进制值	.NET Framework 4.6.2

示例：

```
{  
    "runtimeOptions": {  
        "configProperties": {  
            "System.GC.HeapAffinitizeMask": 1023  
        }  
    }  
}
```

System.GC.GCHeapAffinitizeRanges/COMPlus_GCHeapAffinitizeRanges

- 指定用于垃圾回收器线程的处理器列表。
- 此设置与 `System.GC.HeapAffinitizeMask` 类似, 只是它允许你指定超过 64 个的处理器。
- 对于 Windows 操作系统, 请为处理器编号或范围加上相应的 CPU 组作为前缀, 例如“0:1-10,0:12,1:50-52,1:70”。
- 如果通过将 `System.GC.NoAffinitize` 设置为 `true` 禁用了处理器关联, 则忽略此设置。
- 仅适用于服务器垃圾回收。
- 有关详细信息, 请参阅 Maoni Stephens 的博客文章 [Making CPU configuration better for GC on machines with > 64 CPUs](#)(在 CPU 大于 64 个的计算机上, 为 GC 提供更好的 CPU 配置)。

	III	I	
runtimeconfig.json	System.GC.GCHeapAffinitizeRanges	以逗号分隔的处理器编号列表或处理器编号范围。 Unix 示例：“1-10,12,50-52,70” Windows 示例：“0:1-10,0:12,1:50-52,1:70”	.NET Core 3.0
■	COMPlus_GCHeapAffinitizeRanges	以逗号分隔的处理器编号列表或处理器编号范围。 Unix 示例：“1-10,12,50-52,70” Windows 示例：“0:1-10,0:12,1:50-52,1:70”	.NET Core 3.0

示例：

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.GCHHeapAffinizeRanges": "0:1-10,0:12,1:50-52,1:70"
    }
  }
}
```

COMPlus_GCCpuGroup

- 配置垃圾回收器是否使用 CPU 组。

当 64 位 Windows 计算机具有多个 CPU 组(即, 有超过 64 个处理器)时, 通过启用此元素, 可跨所有 CPU 组扩展垃圾回收。垃圾回收器使用所有核心来创建和平衡堆。

- 仅适用于 64 位 Windows 操作系统上的服务器垃圾回收。
- 默认:禁用 (`0`)。
- 有关详细信息, 请参阅 Maoni Stephens 的博客文章 [Making CPU configuration better for GC on machines with > 64 CPUs](#)(在 CPU 大于 64 个的计算机上, 为 GC 提供更好的 CPU 配置)。

	III	I	
<code>runtimeconfig.json</code>	不可用	不可用	不可用
■	<code>COMPlus_GCCpuGroup</code>	<code>0</code> - 禁用 <code>1</code> - 启用	.NET Core 1.0
<code>.NET Framework app.config</code>	<code>GCCpuGroup</code>	<code>false</code> - 禁用 <code>true</code> - 启用	

NOTE

若要配置公共语言运行时 (CLR), 使其也在所有 CPU 组之间分配线程池中的线程, 请启用 `Thread_UseAllCpuGroups` 元素选项。对于 .NET Core 应用, 可以通过将 `COMPlus_Thread_UseAllCpuGroups` 环境变量的值设置为 `1` 以启用此选项。

System.GC.NoAffinize/COMPlus_GCNоАffinize

- 指定是否将垃圾回收线程与处理器关联。若要关联一个 GC 线程, 则意味着它只能在其特定的 CPU 上运行。为每个 GC 线程创建一个堆。
- 仅适用于服务器垃圾回收。
- 默认:将垃圾回收线程与处理器关联 (`false`)。

	III	I	
<code>runtimeconfig.json</code>	<code>System.GC.NoAffinize</code>	<code>false</code> - 关联 <code>true</code> - 不关联	.NET Core 3.0
■	<code>COMPlus_GCNоАffinize</code>	<code>0</code> - 关联 <code>1</code> - 不关联	.NET Core 3.0
<code>.NET Framework app.config</code>	<code>GCNoAffinize</code>	<code>false</code> - 关联 <code>true</code> - 不关联	.NET Framework 4.6.2

示例:

```
{  
    "runtimeOptions": {  
        "configProperties": {  
            "System.GC.NoAffinitize": true  
        }  
    }  
}
```

System.GC.HeapHardLimit/COMPlus_GCHardLimit

- 指定 GC 堆和 GC 簿记的最大提交大小(以字节为单位)。

	III	I	AAAA
runtimeconfig.json	System.GC.HeapHardLimit	十进制值	.NET Core 3.0
■	COMplus_GCHardLimit	十六进制值	.NET Core 3.0

示例：

```
{  
    "runtimeOptions": {  
        "configProperties": {  
            "System.GC.HeapHardLimit": 209715200  
        }  
    }  
}
```

TIP

如果要在 runtimeconfig.template.json 中设置该选项, 请指定一个十进制值。如果要将选项设置为一个环境变量, 请指定一个十六进制值。例如, 若要将堆硬限制指定为 200 个兆字节 (MiB), 则该值对于 JSON 文件为 209715200, 对于环境变量则为 0xC800000 或 C800000。

System.GC.HeapHardLimitPercent/COMPlus_GCHardLimitPercent

- 指定 GC 堆使用量占总内存的百分比。

	III	I	AAAA
runtimeconfig.json	System.GC.HeapHardLimitPercent	十进制值	.NET Core 3.0
■	COMplus_GCHardLimitPercent	十六进制值	.NET Core 3.0

示例：

```
{  
    "runtimeOptions": {  
        "configProperties": {  
            "System.GC.HeapHardLimitPercent": 30  
        }  
    }  
}
```

TIP

如果要在 runtimeconfig.template.json 中设置该选项, 请指定一个十进制值。如果要将选项设置为一个环境变量, 请指定一个十六进制值。例如, 若要将堆使用率限制为 30%, 则该值对于 JSON 文件为 30, 对于环境变量则为 0x1E 或 1E。

System.GC.RetainVM/COMPlus_GCRetainVM

- 配置是将应删除的段置于备用列表上供将来使用, 还是将其释放回操作系统 (OS)。
- 默认: 将段释放回操作系统 (`false`)。

	III	I	
runtimeconfig.json	<code>System.GC.RetainVM</code>	<code>false</code> - 释放到 OS <code>true</code> - 置于备用列表上	.NET Core 1.0
MSBuild 属性	<code>RetainVMGarbageCollection</code>	<code>false</code> - 释放到 OS <code>true</code> - 置于备用列表上	.NET Core 1.0
■	<code>COMPlus_GCRetainVM</code>	<code>0</code> - 释放到 OS <code>1</code> - 置于备用列表上	.NET Core 1.0

示例

runtimeconfig.json 文件:

```
{  
    "runtimeOptions": {  
        "configProperties": {  
            "System.GC.RetainVM": true  
        }  
    }  
}
```

项目文件:

```
<Project Sdk="Microsoft.NET.Sdk">  
  
    <PropertyGroup>  
        <RetainVMGarbageCollection>true</RetainVMGarbageCollection>  
    </PropertyGroup>  
  
</Project>
```

大型页面

COMPlus_GCLargePages

- 指定设置堆硬限制时是否应使用大型页面。
- 默认: 禁用 (`0`)。
- 这是一项实验性设置。

	III	I	
runtimeconfig.json	不可用	不可用	不可用

	III	I	
■	COMPlus_GCLargePages	<input type="checkbox"/> 0 - 禁用 <input checked="" type="checkbox"/> 1 - 启用	.NET Core 3.0

大型对象

COMPlus_gcAllowVeryLargeObjects

- 在 64 位平台上，为总大小大于 2 千兆字节 (GB) 的数组配置垃圾回收器支持。
- 默认：启用 (1)。
- 在未来的 .NET 版本中，此选项可能会过时。

	III	I	
runtimeconfig.json	不可用	不可用	不可用
■	COMPlus_gcAllowVeryLargeObjects	<input checked="" type="checkbox"/> 1 - 启用 <input type="checkbox"/> 0 - 禁用	.NET Core 1.0
.NET Framework ■ app.config	gcAllowVeryLargeObjects	<input checked="" type="checkbox"/> 1 - 启用 <input type="checkbox"/> 0 - 禁用	.NET Framework 4.5

大型对象堆阈值

System.GC.LOHTreshold/COMPlus_GCLoHThreshold

- 指定导致对象进入大型对象堆 (LOH) 的阈值大小(以字节为单位)。
- 默认阈值为 85,000 字节。
- 指定的值必须大于默认阈值。

	III	I	
runtimeconfig.json	System.GC.LOHTreshold	十进制值	.NET Core 1.0
■	COMPlus_GCLoHThreshold	十六进制值	.NET Core 1.0
.NET Framework ■ app.config	GCLOHThreshold	十进制值	.NET Framework 4.8

示例：

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.LOHTreshold": 120000
    }
  }
}
```

TIP

如果要在 runtimeconfig.template.json 中设置该选项，请指定一个十进制值。如果要将选项设置为一个环境变量，请指定一个十六进制值。例如，若要将阙值大小设置为 120,000 个字节，则该值对于 JSON 文件为 120,000，对于环境变量则为 0x1D4C0 或 1D4C0。

独立 GC

COMPlus_GCName

- 指定库的路径，该库包含运行时打算加载的垃圾回收器。
- 有关详细信息，请参阅 [Standalone GC loader design](#)(独立 GC 加载程序设计)。

	III	I	AAAA
runtimeconfig.json	不可用	不可用	不可用
■	COMPlus_GCName	string_path	.NET Core 2.0

用于全球化的运行时配置选项

2020/3/18 • [Edit Online](#)

固定模式

- 确定 .NET Core 应用是否以全球化固定模式运行而无权访问特定区域性的数据和行为, 或者是否有权访问区域性数据。
- 默认: 运行应用并可访问区域性数据 (`false`)。
- 有关详细信息, 请参阅 [.NET Core 全球化固定模式](#)。

	III	I
<code>runtimconfig.json</code>	<code>System.Globalization.Invariant</code>	<code>false</code> - 可访问区域性数据 <code>true</code> - 以固定模式运行
MSBuild 属性	<code>InvariantGlobalization</code>	<code>false</code> - 可访问区域性数据 <code>true</code> - 以固定模式运行
	<code>DOTNET_SYSTEM_GLOBALIZATION_INVARIANT</code>	<code>0</code> - 可访问区域性数据 <code>1</code> - 以固定模式运行

示例

`runtimconfig.json` 文件:

```
{  
  "runtimeOptions": {  
    "configProperties": {  
      "System.Globalization.Invariant": true  
    }  
  }  
}
```

项目文件:

```
<Project Sdk="Microsoft.NET.Sdk">  
  
  <PropertyGroup>  
    <InvariantGlobalization>true</InvariantGlobalization>  
  </PropertyGroup>  
  
</Project>
```

纪元年份范围

- 确定是否放宽对支持多个纪元的日历的范围检查, 或者超出纪元日期范围的日期是否引发 [ArgumentOutOfRangeException](#)。
- 默认: 放宽范围检查 (`false`)。
- 有关详细信息, 请参阅 [日历、纪元和日期范围: 放宽的范围检查](#)。

runtimetimeconfig.json	Switch.System.Globalization.EnforceJapanFormat	false true - 超出范围导致异常
■	不可用	不可用

日语日期分析

- 确定是否成功分析包含“1”或“Gannen”作为年份的字符串，或是否仅支持“1”。
- 默认：分析包含“1”或“Gannen”作为年份的字符串（`false`）。
- 有关详细信息，请参阅[用多个纪元表示日历中的日期](#)。

runtimetimeconfig.json	Switch.System.Globalization.EnforceLegalYearFormat	false true - 仅支持“Gannen”或“1”
■	不可用	不可用

日语年份格式

- 确定是将日本历时代的第1年的格式设置为“Gannen”还是设置为一个数字。
- 默认：将第1年的格式设置为“Gannen”（`false`）。
- 有关详细信息，请参阅[用多个纪元表示日历中的日期](#)。

runtimetimeconfig.json	Switch.System.Globalization.FormatJapaneseYear	false true - 将格式设置为“Gannen”
■	不可用	不可用

用于网络的运行时配置选项

2020/4/9 • [Edit Online](#)

HTTP/2 协议

- 配置是否启用对 HTTP/2 协议的支持。
- 默认:禁用 (`false`)。
- 已在 .NET Core 3.0 中引入。

	runTimeconfig.json	环境变量
	<code>System.Net.Http.SocketsHttpHandler.Http2Support</code> <input checked="" type="checkbox"/> false - 禁用 <input type="checkbox"/> true - 启用	
		<code>DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER</code> <input checked="" type="checkbox"/> 0 - 禁用 SUPPORT <input type="checkbox"/> 1 - 启用

UseSocketsHttpHandler

- 配置高级网络 API(如 `HttpClient`)是使用 `System.Net.Http.SocketsHttpHandler`, 还是使用基于 `libcurl` 的 `System.Net.Http.HttpClientHandler` 实现。
- 默认:使用 `System.Net.Http.SocketsHttpHandler` (`true`)。
- 可通过调用 `HttpContext.SetSwitch` 方法, 以编程方式配置此设置。

	runTimeconfig.json	环境变量
	<code>System.Net.Http.UseSocketsHttpHandler</code> <input checked="" type="checkbox"/> true - 允许使用 SocketsHttpHandler <input type="checkbox"/> false - 允许使用 HttpClientHandler	
		<code>DOTNET_SYSTEM_NET_HTTP_USESOCKETSHTTPHANDLER</code> <input checked="" type="checkbox"/> 1 - 允许使用 SocketsHttpHandler <input type="checkbox"/> 0 - 允许使用 HttpClientHandler

NOTE

从 .NET 5 开始, `System.Net.Http.UseSocketsHttpHandler` 设置不再可用。

用于线程的运行时配置选项

2020/3/18 • [Edit Online](#)

CPU 组

- 配置是否在各 CPU 组之间自动分布线程。
- 默认:禁用()。

	III	I
runtimeconfig.json	不可用	不可用
■	<code>COMPlus_Thread_UseAllCpuGroups</code>	<input type="checkbox"/> - 禁用 <input checked="" type="checkbox"/> - 启用

最小线程数

- 指定工作线程池的最小线程数。
- 对应于 [ThreadPool.SetMinThreads](#) 方法。

	III	I
runtimeconfig.json	<code>System.Threading.ThreadPool.MinThreads</code> 一个表示最小线程数的整数	
MSBuild 属性	<code>ThreadPoolMinThreads</code>	一个表示最小线程数的整数
■	不可用	不可用

示例

runtimeconfig.json 文件:

```
{  
    "runtimeOptions": {  
        "configProperties": {  
            "System.Threading.ThreadPool.MinThreads": 4  
        }  
    }  
}
```

项目文件:

```
<Project Sdk="Microsoft.NET.Sdk">  
  
<PropertyGroup>  
    <ThreadPoolMinThreads>4</ThreadPoolMinThreads>  
</PropertyGroup>  
  
</Project>
```

最大线程数

- 指定工作线程池的最大线程数。
- 对应于 [ThreadPool.SetMaxThreads](#) 方法。

	III	I
runtimeconfig.json	<code>System.Threading.ThreadPool.MaxThreads</code>	一个表示最大线程数的整数
MSBuild 属性	<code>ThreadPoolMaxThreads</code>	一个表示最大线程数的整数
■	不可用	不可用

示例

runtimeconfig.json 文件:

```
{  
    "runtimeOptions": {  
        "configProperties": {  
            "System.Threading.ThreadPool.MaxThreads": 20  
        }  
    }  
}
```

项目文件:

```
<Project Sdk="Microsoft.NET.Sdk">  
  
<PropertyGroup>  
    <ThreadPoolMaxThreads>20</ThreadPoolMaxThreads>  
</PropertyGroup>  
  
</Project>
```

从 .NET Core 2.0 迁移到 2.1

2020/3/18 • [Edit Online](#)

本文介绍从 .NET Core 2.0 迁移到 2.1 的基本步骤。如果希望将 ASP.NET Core 迁移到 2.1，请参阅[从 ASP.NET Core 2.0 迁移到 2.1](#)。

有关 .NET Core 2.1 新增功能的概述，请参阅[.NET Core 2.1 的新增功能](#)。

更新项目文件以使用 2.1 版本

- 打开项目文件 (*.csproj、*.vbproj 或 *.fsproj 文件)。
- 将目标框架值从 `netcoreapp2.0` 更改为 `netcoreapp2.1`。目标框架由 `<TargetFramework>` 或 `<TargetFrameworks>` 元素定义。
例如，将 `<TargetFramework>netcoreapp2.0</TargetFramework>` 更改为
`<TargetFramework>netcoreapp2.1</TargetFramework>`。
- 删除适用于 .NET Core 2.1 SDK(v 2.1.300 或更高版本)中捆绑的工具的 `<DotNetCliToolReference>` 引用。这些引用包括：
 - `dotnet-watch` (`Microsoft.DotNet.Watcher.Tools`)
 - `dotnet-user-secrets` (`Microsoft.Extensions.SecretManager.Tools`)
 - `dotnet-sql-cache` (`Microsoft.Extensions.Caching.SqlConfig.Tools`)
 - `dotnet-ef` (`Microsoft.EntityFrameworkCore.Tools.DotNet`)

在之前的 .NET Core SDK 版本中，对项目文件中这些工具之一的引用类似于以下示例：

```
<DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
```

由于 .NET Core SDK 不再使用此项，因此如果仍在项目中引用了其中一个捆绑工具，则会显示如下警告：

```
The tool 'Microsoft.EntityFrameworkCore.Tools.DotNet' is now included in the .NET Core SDK. Here is information on resolving this warning.
```

从项目文件中删除这些工具的 `<DotNetCliToolReference>` 引用可解决此问题。

另请参阅

- [从 ASP.NET Core 2.0 迁移到 2.1](#)
- [.NET Core 2.1 的新增功能](#)

从 project.json 迁移 .NET Core 项目

2020/3/18 • [Edit Online](#)

本文档介绍了以下三种适用于 .NET Core 项目的迁移方案：

1. [从 project.json 的一个最新有效架构迁移到 csproj](#)
2. [从 DNX 迁移到 csproj](#)
3. [从 RC3 和以前的 .NET Core csproj 项目迁移到最终格式](#)

本文档仅适用于使用 project.json 的较旧的 .NET Core 项目。它不适用于从 .NET Framework 迁移到 .NET Core。

从 project.json 迁移到 csproj

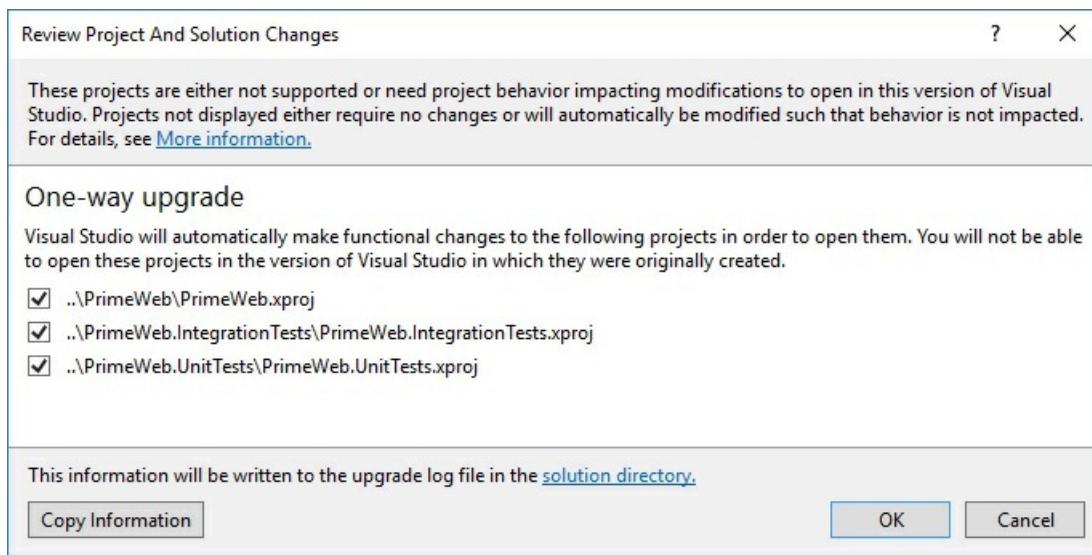
可使用以下任一方法从 project.json 迁移到 .csproj：

- [Visual Studio](#)
- [dotnet migrate 命令行工具](#)

这两种方法使用同一个基础引擎来迁移项目，因此，二者结果相同。在大多数情况下，只需使用这两种方法中的一种将 project.json 迁移到 csproj，而无需进一步对项目文件执行手动编辑。生成的 .csproj 文件的名称与包含目录名称相同。

Visual Studio

在 Visual Studio 2017 或 Visual Studio 2019 版本 16.2 及更早版本中打开 .xproj 文件或引用 .xproj 文件的解决方案文件时，将显示“单向升级”对话框。该对话框将显示要迁移的项目。如果打开解决方案文件，则将列出解决方案文件中指定的所有项目。查看要迁移的项目的列表，然后选择“确定”。



Visual Studio 自动迁移所选的项目。迁移解决方案时，如果不选择所有项目，则会显示相同的对话框，要求升级该解决方案的其余项目。迁移项目后，可通过在“解决方案资源管理器”窗口中右键单击该项目，并选择“编辑 项目名称 > .csproj”来查看和修改其内容 < 。

已迁移的文件（project.json、global.json、.xproj 和解决方案文件）会移动到“备份”文件夹。迁移的解决方案文件会升级到 Visual Studio 2017 或 Visual Studio 2019，并且将无法在 Visual Studio 2015 或更早版本中打开该解决方案文件。还会保存并自动打开名为 UpgradeLog.htm 的文件，该文件包含迁移报告。

IMPORTANT

在 Visual Studio 2019 版本 16.3 和更高版本中，无法加载或迁移 .xproj 文件。此外，Visual Studio 2015 也不提供 .xproj 文件迁移的功能。如果使用以上任一 Visual Studio 版本，请安装适当版本的 Visual Studio，或使用下述命令行迁移工具。

dotnet migrate

在该命令行方案中，可以使用 `dotnet migrate` 命令。它会按顺序迁移项目、解决方案或一组文件夹，具体取决于所找到的项。迁移项目时，将迁移项目及其所有依赖项。

已迁移的文件（`project.json`、`global.json` 和 `.xproj`）会移动到“备份”文件夹。

NOTE

如果使用 Visual Studio Code，则 `dotnet migrate` 命令不会修改 `tasks.json` 等 Visual Studio Code 专属文件。需要手动更改这些文件。如果使用 Visual Studio 以外的编辑器或集成开发环境（IDE），也是如此。

请参阅 [project.json 和 csproj 属性之间的映射](#)，了解 `project.json` 和 `csproj` 格式的比较情况。

如果看到错误消息：

找不到匹配命令 `dotnet-migrate` 的可执行文件

请运行 `dotnet --version` 查看所使用的版本。`dotnet migrate` 在 .NET Core SDK 1.0.0 中引入，并在版本 3.0.100 中删除。如果当前目录或父级目录中有 `global.json` 文件，且它指定的 `sdk` 版本在此范围外，则会收到此错误。

从 DNX 迁移到 csproj

如果仍在使用 DNX 进行 .NET Core 开发，则应分两个阶段完成迁移过程：

1. 使用[现有 DNX 迁移指南](#)从 DNX 迁移到启用了 `project.json` 的 CLI。
2. 请按照上一部分中的步骤，从 `project.json` 迁移到 `.csproj`。

NOTE

已于 .NET Core CLI 的预览版 1 发布期间正式弃用 DNX。

从较早的 .NET Core csproj 格式迁移到 RTM csproj

随着工具的每个新的预发布版本的推出，.NET Core csproj 格式也在不断变化发展。没有工具可以将项目文件从早期版本的 csproj 迁移到最新版本，因此需要手动编辑项目文件。实际步骤取决于要迁移的项目文件的版本。根据版本之间的变化，需考虑以下指导信息：

- 从 `<Project>` 元素中删除工具版本属性（如果存在）。
- 从 `xmlns` 元素中删除 XML 命名空间（`<Project>`）。
- 如果不存在，请将 `Sdk` 属性添加到 `<Project>` 元素，并将其设置为 `Microsoft.NET.Sdk` 或 `Microsoft.NET.Sdk.Web`。此属性指定项目使用要使用的 SDK。`Microsoft.NET.Sdk.Web` 用于 Web 应用。
- 从项目的顶部和底部删除
`<Import Project="$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\Microsoft.Common.props" />` 和
`<Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />` 语句。SDK 隐含这些 import 语句，因此项目中不需要这些语句。
- 如果项目中含 `Microsoft.NETCore.App` 或 `NETStandard.Library`<PackageReference>` 项，应将其删除。[SDK 隐含](#)这些包引用。

- 删除 `Microsoft.NET.Sdk` `<PackageReference>` 元素(如果存在)。SDK 引用来自 `Sdk` 元素上的 `<Project>` 属性。
- 删除 **SDK** 隐含的 `glob`。在项目中留下这些 `glob` 会引发生成错误，因为编译项会发生重复。

完成这些步骤后，项目应与 RTM .NET Core csproj 格式完全兼容。

有关从旧的 csproj 格式迁移到新的 csproj 格式之前和之后情况的示例，请参阅 .NET 博客上的 [Updating Visual Studio 2017 RC – .NET Core Tooling improvements](#)(更新 Visual Studio 2017 RC - .NET Core 工具改进)文章。

另请参阅

- [移植、迁移和升级 Visual Studio 项目](#)

project.json 和 csproj 属性之间的映射

2020/3/18 • [Edit Online](#)

作者 [Nate McMaster](#)

.NET Core 工具的开发过程中实施了一项重要的设计更改，即不再支持 *project.json* 文件，而是将 .NET Core 项目转移到 MSBuild/csproj 格式。

本文介绍 *project.json* 中的设置如何以 MSBuild/csproj 格式表示，以便用户可学习如何使用新格式，并了解将项目升级到最新版本的工具时由迁移工具做出的更改。

csproj 格式

新格式 *.csproj 是一种基于 XML 的格式。以下示例演示使用 `Microsoft.NET.Sdk` 的 .NET Core 项目的根节点。对于 Web 项目，所使用的 SDK 是 `Microsoft.NET.Sdk.Web`。

```
<Project Sdk="Microsoft.NET.Sdk">
...
</Project>
```

常见顶级属性

NAME

```
{
  "name": "MyProjectName"
}
```

不再支持。在 csproj 中，这取决于项目文件名（通常与目录名称匹配）。例如，`MyProjectName.csproj`。

默认情况下，项目文件名还指定 `<AssemblyName>` 和 `<PackageId>` 属性的值。

```
<PropertyGroup>
  <AssemblyName>MyProjectName</AssemblyName>
  <PackageId>MyProjectName</PackageId>
</PropertyGroup>
```

如果 `<AssemblyName>` 属性是在 project.json 中定义的，`<PackageId>` 将具有不同于 `buildOptions\outputName` 的其他值。有关详细信息，请参阅[其他常用生成选项](#)。

版本

```
{
  "version": "1.0.0-alpha-*"
}
```

使用 `VersionPrefix` 和 `VersionSuffix` 属性：

```
<PropertyGroup>
  <VersionPrefix>1.0.0</VersionPrefix>
  <VersionSuffix>alpha</VersionSuffix>
</PropertyGroup>
```

还可以使用 `Version` 属性，但这可能会在打包过程中替代版本设置：

```
<PropertyGroup>
  <Version>1.0.0-alpha</Version>
</PropertyGroup>
```

其他常用根级别选项

```
{
  "authors": [ "Anne", "Bob" ],
  "company": "Contoso",
  "language": "en-US",
  "title": "My library",
  "description": "This is my library.\r\nAnd it's really great!",
  "copyright": "Nugetizer 3000",
  "userSecretsId": "xyz123"
}
```

```
<PropertyGroup>
  <Authors>Anne;Bob</Authors>
  <Company>Contoso</Company>
  <NeutralLanguage>en-US</NeutralLanguage>
  <AssemblyTitle>My library</AssemblyTitle>
  <Description>This is my library.
  And it's really great!</Description>
  <Copyright>Nugetizer 3000</Copyright>
  <UserSecretsId>xyz123</UserSecretsId>
</PropertyGroup>
```

框架

一个目标框架

```
{
  "frameworks": {
    "netcoreapp1.0": {}
  }
}
```

```
<PropertyGroup>
  <TargetFramework>netcoreapp1.0</TargetFramework>
</PropertyGroup>
```

多个目标框架

```
{  
  "frameworks": {  
    "netcoreapp1.0": {},  
    "net451": {}  
  }  
}
```

使用 `TargetFrameworks` 属性定义目标框架的列表。使用分号来分隔多个框架值。

```
<PropertyGroup>  
  <TargetFrameworks>netcoreapp1.0;net451</TargetFrameworks>  
</PropertyGroup>
```

依赖项

IMPORTANT

如果依赖项是一个■而不是包，则格式不同。有关详细信息，请参阅[依赖项类型](#)部分。

NETStandard.Library 元包

```
{  
  "dependencies": {  
    "NETStandard.Library": "1.6.0"  
  }  
}
```

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

Microsoft.NETCore.App 元包

```
{  
  "dependencies": {  
    "Microsoft.NETCore.App": "1.0.0"  
  }  
}
```

```
<PropertyGroup>  
  <RuntimeFrameworkVersion>1.0.3</RuntimeFrameworkVersion>  
</PropertyGroup>
```

请注意，迁移项目中的 `<RuntimeFrameworkVersion>` 值由已安装的 SDK 版本确定。

顶级依赖项

```
{  
  "dependencies": {  
    "Microsoft.AspNetCore": "1.1.0"  
  }  
}
```

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore" Version="1.1.0" />
</ItemGroup>
```

依赖项(按框架)

```
{
  "framework": {
    "net451": {
      "dependencies": {
        "System.Collections.Immutable": "1.3.1"
      }
    },
    "netstandard1.5": {
      "dependencies": {
        "Newtonsoft.Json": "9.0.1"
      }
    }
  }
}
```

```
<ItemGroup Condition="$(TargetFramework)=='net451'">
  <PackageReference Include="System.Collections.Immutable" Version="1.3.1" />
</ItemGroup>

<ItemGroup Condition="$(TargetFramework)=='netstandard1.5'">
  <PackageReference Include="Newtonsoft.Json" Version="9.0.1" />
</ItemGroup>
```

导入

```
{
  "dependencies": {
    "YamlDotNet": "4.0.1-pre309"
  },
  "frameworks": {
    "netcoreapp1.0": {
      "imports": [
        "dnxcore50",
        "dotnet"
      ]
    }
  }
}
```

```
<PropertyGroup>
  <PackageTargetFallback>dnxcore50;dotnet</PackageTargetFallback>
</PropertyGroup>
<ItemGroup>
  <PackageReference Include="YamlDotNet" Version="4.0.1-pre309" />
</ItemGroup>
```

依赖项类型

类型: 项目

```
{  
  "dependencies": {  
    "MyOtherProject": "1.0.0-*",  
    "AnotherProject": {  
      "type": "project"  
    }  
  }  
}
```

```
<ItemGroup>  
  <ProjectReference Include="..\MyOtherProject\MyOtherProject.csproj" />  
  <ProjectReference Include="..\AnotherProject\AnotherProject.csproj" />  
</ItemGroup>
```

NOTE

这将打破 `dotnet pack --version-suffix $suffix` 确定项目引用的依赖项版本的方式。

类型:生成

```
{  
  "dependencies": {  
    "Microsoft.EntityFrameworkCore.Design": {  
      "version": "1.1.0",  
      "type": "build"  
    }  
  }  
}
```

```
<ItemGroup>  
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="1.1.0" PrivateAssets="All" />  
</ItemGroup>
```

类型:平台

```
{  
  "dependencies": {  
    "Microsoft.NETCore.App": {  
      "version": "1.1.0",  
      "type": "platform"  
    }  
  }  
}
```

csproj 中没有等效项。

runtimes

```
{  
  "runtimes": {  
    "win7-x64": {},  
    "osx.10.11-x64": {},  
    "ubuntu.16.04-x64": {}  
  }  
}
```

```
<PropertyGroup>
  <RuntimeIdentifiers>win7-x64;osx.10.11-x64;ubuntu.16.04-x64</RuntimeIdentifiers>
</PropertyGroup>
```

独立应用(独立部署)

在 project.json 中, 定义 `runtimes` 部分意味着应用在生成和发布期间独立。在 MSBuild 中, 生成期间所有项目均可移植, 但可发布为独立。

```
dotnet publish --framework netcoreapp1.0 --runtime osx.10.11-x64
```

有关详细信息, 请参阅[独立部署 \(SCD\)](#)。

工具

```
{
  "tools": {
    "Microsoft.EntityFrameworkCore.Tools.DotNet": "1.0.0-*"
  }
}
```

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="1.0.0" />
</ItemGroup>
```

NOTE

csproj 中不支持工具上的 `imports`。需要导入的工具无法用于新的 `Microsoft.NET.Sdk`。

buildOptions

另请参阅[文件](#)。

emitEntryPoint

```
{
  "buildOptions": {
    "emitEntryPoint": true
  }
}
```

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
</PropertyGroup>
```

如果 `emitEntryPoint` 为 `false`, `OutputType` 的值会转换为 `Library` (这是默认值) :

```
{
  "buildOptions": {
    "emitEntryPoint": false
  }
}
```

```
<PropertyGroup>
  <OutputType>Library</OutputType>
  <!-- or, omit altogether. It defaults to 'Library' -->
</PropertyGroup>
```

keyFile

```
{
  "buildOptions": {
    "keyFile": "MyKey.snk"
  }
}
```

`keyFile` 元素在 MSBuild 中扩展为三个属性：

```
<PropertyGroup>
  <AssemblyOriginatorKeyFile>MyKey.snk</AssemblyOriginatorKeyFile>
  <SignAssembly>true</SignAssembly>
  <PublicSign Condition="'$(OS)' != 'Windows_NT'">true</PublicSign>
</PropertyGroup>
```

其他常用生成选项

```
{
  "buildOptions": {
    "warningsAsErrors": true,
    "nowarn": ["CS0168", "CS0219"],
    "xmlDoc": true,
    "preserveCompilationContext": true,
    "outputName": "Different.AssemblyName",
    "debugType": "portable",
    "allowUnsafe": true,
    "define": ["TEST", "OTHERCONDITION"]
  }
}
```

```
<PropertyGroup>
  <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
  <NoWarn>$(NoWarn);CS0168;CS0219</NoWarn>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  <PreserveCompilationContext>true</PreserveCompilationContext>
  <AssemblyName>Different.AssemblyName</AssemblyName>
  <DebugType>portable</DebugType>
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
  <DefineConstants>$(DefineConstants);TEST;OTHERCONDITION</DefineConstants>
</PropertyGroup>
```

packOptions

另请参阅[文件](#)。

常用包选项

```
{
  "packOptions": {
    "summary": "numl is a machine learning library intended to ease the use of using standard modeling techniques for both prediction and clustering.",
    "tags": ["machine learning", "framework"],
    "releaseNotes": "Version 0.9.12-beta",
    "iconUrl": "http://numl.net/images/ico.png",
    "projectUrl": "http://numl.net",
    "licenseUrl": "https://raw.githubusercontent.com/sethjuarez/numl/master/LICENSE.md",
    "requireLicenseAcceptance": false,
    "repository": {
      "type": "git",
      "url": "https://raw.githubusercontent.com/sethjuarez/numl"
    },
    "owners": ["Seth Juarez"]
  }
}
```

```
<PropertyGroup>
  <!-- summary is not migrated from project.json, but you can use the <Description> property for that if needed. -->
  <PackageTags>machine learning;framework</PackageTags>
  <PackageReleaseNotes>Version 0.9.12-beta</PackageReleaseNotes>
  <PackageIconUrl>http://numl.net/images/ico.png</PackageIconUrl>
  <PackageProjectUrl>http://numl.net</PackageProjectUrl>
  <PackageLicenseUrl>https://raw.githubusercontent.com/sethjuarez/numl/master/LICENSE.md</PackageLicenseUrl>
  <PackageRequireLicenseAcceptance>false</PackageRequireLicenseAcceptance>
  <RepositoryType>git</RepositoryType>
  <RepositoryUrl>https://raw.githubusercontent.com/sethjuarez/numl</RepositoryUrl>
  <!-- owners is not supported in MSBuild -->
</PropertyGroup>
```

MSBuild 中没有 `owners` 元素的等效项。对于 `summary`，可使用 MSBuild `<Description>` 属性 - 即使 `summary` 的值未自动迁移到该属性，因为该属性已映射到 `description` 元素。

脚本

```
{
  "scripts": {
    "precompile": "generateCode.cmd",
    "postpublish": [ "obfuscate.cmd", "removeTempFiles.cmd" ]
  }
}
```

它们在 MSBuild 中的等效项是[目标](#)：

```
<Target Name="MyPreCompileTarget" BeforeTargets="Build">
  <Exec Command="generateCode.cmd" />
</Target>

<Target Name="MyPostCompileTarget" AfterTargets="Publish">
  <Exec Command="obfuscate.cmd" />
  <Exec Command="removeTempFiles.cmd" />
</Target>
```

runtimeOptions

```
{  
  "runtimeOptions": {  
    "configProperties": {  
      "System.GC.Server": true,  
      "System.GC.Concurrent": true,  
      "System.GC.RetainVM": true,  
      "System.Threading.ThreadPool.MinThreads": 4,  
      "System.Threading.ThreadPool.MaxThreads": 25  
    }  
  }  
}
```

此组中除“System.GC.Server”属性以外的所有设置与迁移过程中提升为根对象的选项一并被置于项目文件夹中名为 `runtimeconfig.template.json` 的文件中：

```
{  
  "configProperties": {  
    "System.GC.Concurrent": true,  
    "System.GC.RetainVM": true,  
    "System.Threading.ThreadPool.MinThreads": 4,  
    "System.Threading.ThreadPool.MaxThreads": 25  
  }  
}
```

已将“System.GC.Server”属性迁移到 csproj 文件：

```
<PropertyGroup>  
  <ServerGarbageCollection>true</ServerGarbageCollection>  
</PropertyGroup>
```

但可以在 csproj 以及 MSBuild 属性中设置所有这些值：

```
<PropertyGroup>  
  <ServerGarbageCollection>true</ServerGarbageCollection>  
  <ConcurrentGarbageCollection>true</ConcurrentGarbageCollection>  
  <RetainVMGarbageCollection>true</RetainVMGarbageCollection>  
  <ThreadPoolMinThreads>4</ThreadPoolMinThreads>  
  <ThreadPoolMaxThreads>25</ThreadPoolMaxThreads>  
</PropertyGroup>
```

共享

```
{  
  "shared": "shared/**/*.cs"  
}
```

在 csproj 中不支持。而必须在 `.nuspec` 文件中创建要包含的内容文件。有关详细信息，请参阅[包含内容文件](#)。

文件

在 `project.json` 中，可将生成和打包操作扩展为从不同的文件夹进行编译和嵌入。在 MSBuild 中，使用[项](#)实现此操作。以下示例是一个常见转换：

```
{
  "buildOptions": {
    "compile": {
      "copyToOutput": "notes.txt",
      "include": "../Shared/*.cs",
      "exclude": "../Shared/Not/*.cs"
    },
    "embed": {
      "include": "../Shared/*.resx"
    }
  },
  "packOptions": {
    "include": "Views/",
    "mappings": {
      "some/path/in/project.txt": "in/package.txt"
    }
  },
  "publishOptions": {
    "include": [
      "files/",
      "publishnotes.txt"
    ]
  }
}
```

```
<ItemGroup>
  <Compile Include=".\\Shared\\*.cs" Exclude=".\\Shared\\Not\\*.cs" />
  <EmbeddedResource Include=".\\Shared\\*.resx" />
  <Content Include="Views\\**\\*" PackagePath ="%(Identity)" />
  <None Include="some/path/in/project.txt" Pack="true" PackagePath="in/package.txt" />

  <None Include="notes.txt" CopyToOutputDirectory="Always" />
  <!-- CopyToOutputDirectory = { Always, PreserveNewest, Never } -->

  <Content Include="files\\**\\*" CopyToPublishDirectory="PreserveNewest" />
  <None Include="publishnotes.txt" CopyToPublishDirectory="Always" />
  <!-- CopyToPublishDirectory = { Always, PreserveNewest, Never } -->
</ItemGroup>
```

NOTE

许多默认 [glob 模式](#) 由 .NET Core SDK 自动添加。有关更多信息, 请参见[默认编译项值](#)。

所有 MSBuild `ItemGroup` 元素都支持 `Include`、`Exclude` 和 `Remove`。

可使用 `PackagePath="path"` 修改 .nupkg 内的包布局。

除 `Content` 外, 大多数项组需要显式添加要包括在包中的 `Pack="true"`。`Content` 将被置于包中的 *content* 文件夹, 因为 `<IncludeContentInPack>` 属性默认设置为 `true`。有关详细信息, 请参阅[在包中包含内容](#)。

`PackagePath ="%(Identity)"` 是一种将包路径设置为项目相对文件路径的快捷方法。

testRunner

xUnit

```
{  
  "testRunner": "xunit",  
  "dependencies": {  
    "dotnet-test-xunit": "<any>"  
  }  
}
```

```
<ItemGroup>  
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.0.0-*" />  
  <PackageReference Include="xunit" Version="2.2.0-*" />  
  <PackageReference Include="xunit.runner.visualstudio" Version="2.2.0-*" />  
</ItemGroup>
```

MSTest

```
{  
  "testRunner": "mstest",  
  "dependencies": {  
    "dotnet-test-mstest": "<any>"  
  }  
}
```

```
<ItemGroup>  
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.0.0-*" />  
  <PackageReference Include="MSTest.TestAdapter" Version="1.1.12-*" />  
  <PackageReference Include="MSTest.TestFramework" Version="1.1.11-*" />  
</ItemGroup>
```

另请参阅

- [CLI 中更改的简要概述](#)

.NET Core 工具中变更的高级概述

2020/3/18 • [Edit Online](#)

此文档介绍了与从 project.json 移动到 MSBuild 和 csproj 项目系统有关的更改，其中包含关于 .NET Core 工具的分层和 CLI 命令的实现的更改的信息。这些更改与 2017 年 3 月 7 日 .NET Core SDK 1.0 和 Visual Studio 2017 的发布同时发生（请参阅[通知](#)），但是最初是在 .NET Core SDK 预览 3 版本中实现的。

弃用 project.json

.NET Core 工具的最大变更无疑是[弃用 project.json, 改用 csproj](#) 作为项目系统。最新版本的命令行工具不支持 project.json 文件。这意味着它不能用于生成、运行或发布基于 project.json 的应用程序和库。若要使用此版本的工具，请迁移现有项目或启动新的项目。

作为此次移动的一部分，开发用于生成 project.json 项目的自定义生成引擎被替换为一个功能完整的成熟生成引擎，即 [MSBuild](#)。MSBuild 是 .NET 社区中的知名引擎。自从在平台上首次发布，就一直是一项关键技术。由于需要生成 .NET Core 应用程序，所以 MSBuild 已经移植到 .NET Core，并可在 .NET Core 运行的任何平台上使用。.NET Core 的一个主要的好处是跨平台开发堆栈，我们已确保本次迁移不会破坏此好处。

TIP

如果还不熟悉 MSBuild，并且想要了解相关详细信息，可参阅 [MSBuild 概念一文](#)。

工具层

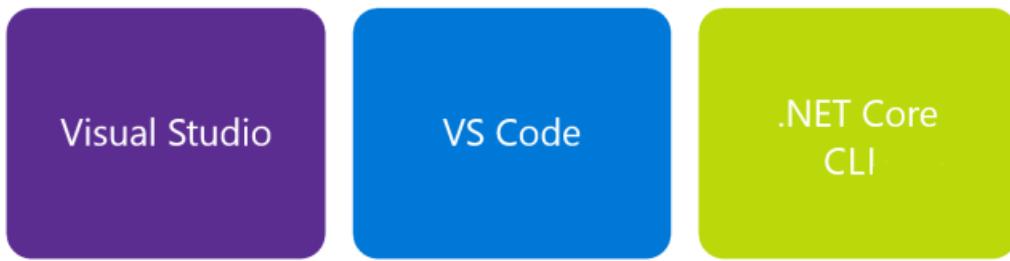
随着生成引擎发生变化，并从现有项目系统中迁移出来，一些问题也会随之而来。这些更改是否会导致 .NET Core 工具生态系统的整体“分层”发生更改？是否有新的位和组件？

首先来简要介绍下预览版 2 分层，如下图所示：



预览版 2 中工具的分层较为简单。底层基底是 .NET Core CLI。其他所有高级工具（如 Visual Studio 或 Visual Studio Code）依靠 CLI 来生成项目、还原依赖项以及完成其他操作。例如，如果 Visual Studio 要执行还原操作，它会在 CLI 中调用 `dotnet restore`（[见备注](#)）命令。

随着迁移到新的项目系统，之前的图表会更改：



主要区别在于 CLI 不再作为基础层;现由“共享 SDK 组件”充当此角色。共享 SDK 组件是一组负责编译代码、发布代码、打包 NuGet 包等操作的目标和关联任务。.NET Core SDK 是一个开源代码, 可在 GitHub 上的 [SDK 存储库](#)中获得。

NOTE

“目标”是一个 MSBuild 术语, 指示 MSBuild 可调用的一个已命名的操作。其通常伴随着执行此目标应执行的某个逻辑的一个或多个任务。MSBuild 支持多个现成的目标, 如 `Copy` 或 `Execute`;它还允许用户使用托管代码编写自己的任务, 并定义要执行这些任务的目标。有关详细信息, 请参阅 [MSBuild 任务](#)。

现在所有工具集使用共享 SDK 组件及其目标, 包括 CLI。例如, Visual Studio 2019 不会调入 `dotnet restore` ([参见说明](#))命令来还原 .NET Core 项目的依赖项。相反, 它会直接使用“还原”目标。由于这些皆是 MSBuild 目标, 因此你也可通过 `dotnet msbuild` 命令使用原始 MSBuild 来执行。

CLI 命令

共享 SDK 组件意味着大部分现有 CLI 命令已重新实现为 MSBuild 任务和目标。这对 CLI 命令和工具集的使用意味着什么?

从使用角度而言, 它不会更改使用 CLI 的方式。CLI 仍具有 .NET Core 1.0 预览 2 版本中存在的核心命令:

- `new`
- `restore`
- `run`
- `build`
- `publish`
- `test`
- `pack`

这些命令的作用没有发生改变(仍可新建项目、生成项目、发布项目、打包项目等等)。可以参阅命令的帮助屏幕(使用 `dotnet <command> --help`)或此站点上的文档, 来熟悉其行为。

从执行角度而言, CLI 命令会采用其参数并构造对“原始”MSBuild 的调用, 从而设置所需属性和运行所需目标。为更好的说明这点, 请参考下面的命令:

```
dotnet publish -o pub -c Release
```

此命令会使用“发布”配置将应用程序发布到 `pub` 文件夹。在内部, 此命令会转换成下面的 MSBuild 调用:

```
dotnet msbuild -t:Publish -p:OutputPath=pub -p:Configuration=Release
```

此规则的两个明显的例外情况是 `new` 和 `run` 命令。它们未实现为 MSBuild 目标。

NOTE

从 .NET Core 2.0 SDK 开始，无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build` 和 `dotnet run`。在执行显式还原有意义的某些情况下，例如 [Azure DevOps Services 中的持续集成生成中](#)，或在需要显式控制还原发生时间的生成系统中，它仍然是有效的命令。

.NET Core 的 csproj 格式的新增内容

2020/3/30 • [Edit Online](#)

本文档概述了作为从 `project.json` 移动到 `csproj` 和 [MSBuild](#) 的一部分，添加到项目文件的更改。有关常规项目文件的语法和引用的详细信息，请参阅 [MSBuild 项目文件](#) 文档。

隐式包引用

基于项目文件的 `<TargetFramework>` 或 `<TargetFrameworks>` 属性中指定的目标框架对元包进行隐式引用。如果指定了 `<TargetFramework>`，则忽略 `<TargetFrameworks>`，而与顺序无关。有关详细信息，请参阅 [包、元包和框架](#)。

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.1</TargetFramework>
</PropertyGroup>
```

```
<PropertyGroup>
  <TargetFrameworks>netcoreapp2.1;net462</TargetFrameworks>
</PropertyGroup>
```

建议

由于隐式引用了 `Microsoft.NETCore.App` 或 `NETStandard.Library` 元包，以下是建议的最佳做法：

- 面向 .NET Core 或 .NET Standard 时，**绝不通过项目文件中的 `<PackageReference>` 项，对 `Microsoft.NETCore.App` 或 `NETStandard.Library` 元包进行显式引用**。
- 面向 .NET Core 时，如果需要特定版本的运行时，应使用项目中的 `<RuntimeFrameworkVersion>` 属性（例如，`1.0.4`），而不是引用元包。
 - 例如，如果使用**独立部署**且需要 1.0.0 LTS 运行时的特定修补程序版本，可能会发生这种情况。
- 面向 .NET Standard 时，如果需要特定版本的 `NETStandard.Library` 元包，可以使用 `<NetStandardImplicitPackageVersion>` 属性并设置所需版本。
- 请勿在 .NET Framework 项目中显式添加或更新对 `Microsoft.NETCore.App` 或 `NETStandard.Library` 元包的引用。使用基于 .NET Standard 的 NuGet 包时，如果需要任意版本的 `NETStandard.Library`，NuGet 可自动安装该版本。

一些包引用的隐式版本

`<PackageReference>` 的大多数用法都要求设置 `Version` 属性，用于指定要使用的 NuGet 包版本。不过，如果使用的是 .NET Core 2.1 或 2.2，且引用 `Microsoft.AspNetCore.App` 或 `Microsoft.AspNetCore.All`，就没有必要设置此属性。.NET Core SDK 可自动选择应使用的包版本。

建议

引用 `Microsoft.AspNetCore.App` 或 `Microsoft.AspNetCore.All` 包时，不要指定包版本。如果你指定版本，SDK 可能会生成警告 NETSDK1071。若要修复此警告，请删除包版本，如下面的示例所示：

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.App" />
</ItemGroup>
```

已知问题：.NET Core 2.1 SDK 只在项目还使用 `Microsoft.NET.Sdk.Web` 时才支持这种语法。.NET Core 2.2 SDK 中解决了此问题。

这些对 ASP.NET Core 元包的引用行为与大多数普通 NuGet 包略有不同。使用这些元包的应用的 [框架依赖部署](#) 自动使

用 ASP.NET Core 共享框架。使用元包时，引用的 ASP.NET Core NuGet 包中的任何资产都不会与应用一起部署。也就是说，ASP.NET Core 共享框架包含这些资产。共享框架中的资产更适合目标平台，旨在缩短应用启动时间。若要详细了解共享框架，请参阅 [.NET Core 分发打包](#)。

如果指定 版本，这会被视为框架依赖部署的 ASP.NET Core 共享框架的最低 版本，并被视为独立式部署的确切 版本。这可能会导致以下后果：

- 如果服务器上安装的 ASP.NET Core 版本低于 PackageReference 中指定的版本，.NET Core 进程便会无法启动。元包更新通常先可用于 NuGet.org，再可用于托管环境（如 Azure）。将 PackageReference 中的版本更新为 ASP.NET Core 可能会导致部署的应用失败。
- 如果应用部署为[独立式部署](#)，应用可能不包含 .NET Core 的最新安全更新程序。如果未指定版本，SDK 可以自动在独立式部署中包含 ASP.NET Core 的最新版本。

.NET Core 项目中默认包含的编译项

已通过移动到最新 SDK 版本中的 *csproj* 格式，将默认的编译项和嵌入资源的包含项和排除项移至 SDK 属性文件。这意味着不需要再在项目文件中指定这些项。

执行此操作的主要目的是减少项目文件中的混杂。SDK 中的默认设置应涵盖最常见的用例，由此便无需在创建的每个项目中重复这些设置。这可使项目文件更小，更易于理解和进行手动编辑（如果需要）。

下表显示同时在 SDK 中包含和排除的元素和 [glob](#)：

编译项	隐式 GLOB	显式 GLOB	不可用 GLOB
Compile	**/*.cs(或其他语言扩展名)	**/*.user; **/*.*proj; **/*.sln; **/*.vsscc	不可用
EmbeddedResource	**/*.resx	**/*.user; **/*.*proj; **/*.sln; **/*.vsscc	不可用
None	**/*	**/*.user; **/*.*proj; **/*.sln; **/*.vsscc	**/*.cs; **/*.*resx

NOTE

排除 glob 始终排除 `./bin` 和 `./obj` 文件夹，它们分别由 MSBuild 属性 `$(BaseOutputPath)` 和 `$(BaseIntermediateOutputPath)` 表示。总体上来说，所有排除都由 `$(DefaultItemExcludes)` 表示。

如果项目中有 glob，却又尝试使用最新的 SDK 生成它，则将收到以下错误：

包含重复的编译项。默认情况下，.NET SDK 包括项目目录中的编译项。可从项目文件中删除这些项，或如果想要在项目文件中显式包括它们，则将“`EnableDefaultCompileItems`”属性设为“`false`”。

要解决此错误，可以删除与前表中所列项匹配的显式 `Compile` 项，也可以将 `<EnableDefaultCompileItems>` 属性设置为 `false`，如下所示：

```
<PropertyGroup>
  <EnableDefaultCompileItems>false</EnableDefaultCompileItems>
</PropertyGroup>
```

将此属性设置为 `false` 将禁用隐式包含，并还原到以前 SDK 的行为，在这种情况下，必须在项目中指定默认 glob。

此更改不会修改其他包含项的主要机制。但是，如果要指定（例如，指定某些文件通过应用发布），仍可以使用 *csproj* 中相应的已知机制来实现（例如，`<Content>` 元素）。

`<EnableDefaultCompileItems>` 仅禁用 `Compile` glob，但不会影响其他 glob（如隐式 `None` glob），这也适用于 `*.cs` 项。

因此，解决方案资源管理器 继续显示在项目中作为 `None` 项的 *.cs 项。以类似的方式，可以将 `<EnableDefaultNoneItems>` 设置为 `false` 以禁用隐式 `None` glob，如下所示：

```
<PropertyGroup>
  <EnableDefaultNoneItems>false</EnableDefaultNoneItems>
</PropertyGroup>
```

要禁用所有隐式 glob，可将 `<EnableDefaultItems>` 属性设置为 `false`，如以下示例所示：

```
<PropertyGroup>
  <EnableDefaultItems>false</EnableDefaultItems>
</PropertyGroup>
```

如何像 MSBuild 一样查看整个项目

虽然这些 csproj 更改极大地简化了项目文件，但建议查看完全展开的项目，就像 MSBuild 查看添加了 SDK 及其目标的项目一样。使用 `dotnet msbuild` 命令的 `/pp` 开关预处理项目，显示导入的文件、文件源及其在生成中的参与情况，而无需实际生成项目：

```
dotnet msbuild -pp:fullproject.xml
```

如果项目有多个目标框架，命令结果应仅侧重于框架之一，具体方法为将相应框架指定为 MSBuild 属性：

```
dotnet msbuild -p:TargetFramework=netcoreapp2.0 -pp:fullproject.xml
```

新增内容

Sdk 特性

.csproj 文件的根 `<Project>` 元素具有名为 `Sdk` 的新特性。`Sdk` 指定项目将使用的 SDK。如[分层文档](#)中所述，SDK 是一组可生成 .NET Core 代码的 MSBuild 任务和目标。.NET Core 可使用以下 SDK：

1. ID 为 `Microsoft.NET.Sdk` 的 .NET Core SDK
2. ID 为 `Microsoft.NET.Sdk.Web` 的 .NET Core Web SDK
3. ID 为 `Microsoft.NET.Sdk.Razor` 的 .NET Core Razor 类库 SDK
4. ID 为 `Microsoft.NET.Sdk.Worker` 的 .NET Core Worker Service(自 .NET Core 3.0 起)
5. ID 为 `Microsoft.NET.Sdk.WindowsDesktop` 的 .NET Core WinForms 和 WPF(自 .NET Core 3.0 起)

需要在 `<Project>` 元素上将 `Sdk` 属性设置为这两个 ID 之一，以使用 .NET Core 工具和生成代码。

PackageReference

`<PackageReference>` 项元素指定[项目中的 NuGet 依赖项](#)。`Include` 属性指定包 ID。

```
<PackageReference Include="package-id" Version="" PrivateAssets="" IncludeAssets="" ExcludeAssets="" />
```

Version

所需的 `Version` 属性指定要还原的包的版本。此属性遵循 [NuGet 版本控制](#) 方案规则。默认行为是最小版本(包含)。例如，指定 `Version="1.2.3"` 等效于 NuGet 表示法 `[1.2.3,)`，表示已解析的包的版本为 1.2.3(如果适用)，如果不适用，则为更高版本。

IncludeAssets、ExcludeAssets 和 PrivateAssets

`IncludeAssets` 属性指定应使用 `<PackageReference>` 指定的包中的哪些资产。默认情况下，包含所有包资产。

`ExcludeAssets` 属性指定不应使用 `<PackageReference>` 指定的包中的哪些资产。

`PrivateAssets` 属性指定应使用 `<PackageReference>` 指定的包中的哪些资产，但不得将这些资产传递到下一个项目。不存在此属性时，`Analyzers`、`Build` 和 `ContentFiles` 资产默认为私有。

NOTE

`PrivateAssets` 等效于 `project.json/xproj` `SuppressParent` 元素。

这些属性可以包含以下一个或多个项，如果列出多个项，则用分号 `;` 字符进行分隔：

- `Compile` - 可对 lib 文件夹的内容进行编译。
- `Runtime` - 分发 runtime 文件夹的内容。
- `ContentFiles` - 使用 *contentfiles* 文件夹的内容。
- `Build` - 使用 build 文件夹中的属性/目标。
- `Native` - 将本机资产内容复制到 output 文件夹 以供运行时使用。
- `Analyzers` - 使用分析器。

此属性也可以包含：

- `None` - 不使用任何资产。
- `All` - 使用所有资产。

DotNetCliToolReference

`<DotNetCliToolReference>` 项元素指定用户想要在项目的上下文中还原的 CLI 工具。在 `project.json` 中，它可以替换 `tools` 节点。

```
<DotNetCliToolReference Include=<package-id> Version="" />
```

请注意，`DotNetCliToolReference` 现已弃用，以支持 .NET Core 本地工具。

Version

`Version` 指定要还原的包的版本。此属性遵循 [NuGet 版本控制](#) 方案规则。默认行为是最小版本(包含)。例如，指定 `Version="1.2.3"` 等效于 NuGet 表示法 `[1.2.3,)`，表示已解析的包的版本为 1.2.3(如果适用)，如果不适用，则为更高版本。

RuntimeIdentifiers

`<RuntimeIdentifiers>` 属性元素可用于指定项目的[运行时标识符 \(RID\)](#) 的列表(以分号分隔)。RID 允许发布独立部署。

```
<RuntimeIdentifiers>win10-x64;osx.10.11-x64;ubuntu.16.04-x64</RuntimeIdentifiers>
```

RuntimeIdentifier

`<RuntimeIdentifier>` 属性元素可用于指定项目的唯一[运行时标识符 \(RID\)](#)。RID 支持发布独立部署。

```
<RuntimeIdentifier>ubuntu.16.04-x64</RuntimeIdentifier>
```

如果需要为多个运行时发布，请使用 `<RuntimeIdentifiers>` (复数)。如果只需要单个运行时，`<RuntimeIdentifier>` 可以进行较快的生成。

PackageTargetFallback

`<PackageTargetFallback>` 属性元素可用于指定要在还原包时使用的一组兼容目标。旨在允许使用 [dotnet TxM \(目标 x 名字对象\)](#) 的包处理未声明 dotnet TxM 的包。如果项目使用 dotnet TxM，那么所依赖的所有包也必须有 dotnet TxM，除非将 `<PackageTargetFallback>` 添加到项目中，以允许非 dotnet 平台与 dotnet 兼容。

以下示例展示了项目中所有目标的回退：

```
<PackageTargetFallback>
    $(PackageTargetFallback);portable-net45+win8+wpa81+wp8
</PackageTargetFallback >
```

以下示例仅指定了 `netcoreapp2.1` 目标的回退：

```
<PackageTargetFallback Condition="'$(TargetFramework)']=='netcoreapp2.1'">
    $(PackageTargetFallback);portable-net45+win8+wpa81+wp8
</PackageTargetFallback >
```

生成事件

在项目文件中指定生成前和生成后事件的方式已更改。建议 `PreBuildEvent` 和 `PostBuildEvent` 属性不要采用 SDK 样式的项目格式，因为不会解析 `$(ProjectDir)` 等宏。例如，不再支持以下代码：

```
<PropertyGroup>
    <PreBuildEvent>"$(ProjectDir)PreBuildEvent.bat" "$(ProjectDir)..\" "$(ProjectDir)" "$(TargetDir)"
    </PreBuildEvent>
</PropertyGroup>
```

在 SDK 样式的项目中，请使用名为 `PreBuild` 或 `PostBuild` 的 MSBuild 目标，并设置 `PreBuild` 的 `BeforeTargets` 属性或 `PostBuild` 的 `AfterTargets` 属性。在上述示例中，使用以下代码：

```
<Target Name="PreBuild" BeforeTargets="PreBuildEvent">
    <Exec Command="$(ProjectDir)PreBuildEvent.bat" $(ProjectDir)..\" $(ProjectDir)" $(TargetDir)
        &quot;$(ProjectDir)&quot; &quot;$(TargetDir)&quot;" />
</Target>

<Target Name="PostBuild" AfterTargets="PostBuildEvent">
    <Exec Command="echo Output written to $(TargetDir)" />
</Target>
```

NOTE

可对 MSBuild 目标使用任何名称，但 Visual Studio IDE 会识别 `PreBuild` 和 `PostBuild` 目标，因此建议使用这些名称，以便可在 Visual Studio IDE 中编辑命令。

NugetMetadataProperties

迁移到 MSBuild 后，我们已将在打包 NuGet 包时使用的输入元数据从 `project.json` 移到 `.csproj` 文件中。输入为 MSBuild 属性，因此它们必须转到 `<PropertyGroup>` 组中。下面列出了在使用 `dotnet pack` 命令或属于 SDK 的 `Pack` MSBuild 目标时，用作打包进程的输入的属性：

IsPackable

一个指定能否打包项目的布尔值。默认值为 `true`。

PackageVersion

指定生成的包所具有的版本。接受所有形式的 NuGet 版本字符串。默认为值 `$(Version)`，即项目中 `Version` 属性的值。

PackageId

指定生成的包的名称。如果未指定，`pack` 操作将默认使用 `AssemblyName` 或目录名称作为包的名称。

Title

明了易用的包标题，通常用在 UI 显示中，如 `nuget.org` 上和 Visual Studio 中包管理器上的那样。如果未指定，则改为

使用包 ID。

Authors

其中名称以分号分隔的包作者列表，其中名称与 nuget.org 上的配置文件名称匹配。这些信息显示在 nuget.org 上的 NuGet 库中，并用于交叉引用同一作者的包。

PackageDescription

用于 UI 显示的包的详细说明。

Description

程序集的详细说明。如果未指定 `PackageDescription`，则此属性还可用作包的说明。

Copyright

包的版权详细信息。

PackageRequireLicenseAcceptance

一个布尔值，指定客户端是否必须提示使用者接受包许可证后才可安装包。默认值为 `false`。

DevelopmentDependency

一个布尔值，用于指定包是否被标记为仅开发依赖项，从而防止包作为依赖项包含到其他包中。利用 `PackageReference` (NuGet 4.8+)，此标志还意味着将从编译中排除编译时资产。有关详细信息，请参阅 [PackageReference 的 DevelopmentDependency 支持](#)。

PackageLicenseExpression

[SPDX 许可证标识符](#)或表达式。例如 `Apache-2.0`。

下面是 [SPDX 许可证标识符](#)的完整列表。NuGet.org 在使用许可证类型表达式时只接受 OSI 或 FSF 批准的许可证。

许可证表达式的准确语法如下面的 [ABNF](#) 所述。

```
license-id          = <short form license identifier from https://spdx.org/spdx-specification-21-web-version#h.luq9dgcle9mo>

license-exception-id = <short form license exception identifier from https://spdx.org/spdx-specification-21-web-version#h.ruv3yl8g6czd>

simple-expression = license-id / license-id"+"

compound-expression = 1*1(simple-expression /
    simple-expression "WITH" license-exception-id /
    compound-expression "AND" compound-expression /
    compound-expression "OR" compound-expression ) /
    "(" compound-expression ")" )

license-expression = 1*1(simple-expression / compound-expression / UNLICENSED)
```

NOTE

一次只能指定 `PackageLicenseExpression`、`PackageLicenseFile` 和 `PackageLicenseUrl` 中的一个。

PackageLicenseFile

如果使用的许可证没有分配 SPDX 标识符，或者它是自定义许可证，则它是包中许可证文件的路径（否则，优先选择 `PackageLicenseExpression`）

替换 `PackageLicenseUrl`，不能与 `PackageLicenseExpression` 结合使用，并且需要 Visual Studio 版本 15.9.4 和 .NET SDK 2.1.502 或 2.2.101 或更高版本。

将需要通过显式地将许可证文件添加到项目中来确保许可证文件已打包，示例用法如下：

```
<PropertyGroup>
  <PackageLicenseFile>LICENSE.txt</PackageLicenseFile>
</PropertyGroup>
<ItemGroup>
  <None Include="licenses\LICENSE.txt" Pack="true" PackagePath="$(PackageLicenseFile)"/>
</ItemGroup>
```

PackageLicenseUrl

适用于包的许可证的 URL。(自 Visual Studio 15.9.4、.NET SDK 2.1.502 和 2.2.101 起已弃用)

PackageIconUrl

64x64 透明背景图像的 URL, 用作 UI 显示中包的图标。

PackageReleaseNotes

包的发行说明。

PackageTags

标记的分号分隔列表, 这些标记用于指定包。

PackageOutputPath

确定用于已打包的包的输出路径。默认值为 `$(OutputPath)`。

IncludeSymbols

此布尔值指示在打包项目时, 包是否应创建一个附加的符号包。符号包的格式由 `SymbolPackageFormat` 属性控制。

SymbolPackageFormat

指定符号包的格式。如果为“symbols.nupkg”, 将使用包含 PDB、DLL 和其他输出文件的 .symbols.nupkg 扩展创建旧符号包。如果为“snupkg”, 将创建包含可移植 PDB 的 snupkg 符号包。默认值为“symbols.nupkg”。

IncludeSource

此布尔值指示包进程是否应创建源包。源包中包含库的源代码以及 PDB 文件。源文件置于生成的包文件中的 `src/ProjectName` 目录下。

IsTool

指定是否将所有输出文件复制到 `tools` 文件夹, 而不是 `lib` 文件夹。这不同于 `DotNetCliTool`, 后者通过在 `.csproj` 文件中设置 `PackageType` 进行指定。

RepositoryUrl

指定存储库的 URL, 该存储库是包的源代码所驻留和/或生成的位置。

RepositoryType

指定存储库的类型。默认值为“git”。

RepositoryBranch

指定存储库中源分支的名称。当项目打包到 NuGet 包时, 它将被添加到包元数据。

RepositoryCommit

可选的存储库提交或更改集, 指示针对其生成包的源。还必须指定 `RepositoryUrl` 才能包含此属性。当项目打包到 NuGet 包中时, 此提交或变更集将添加到包元数据中。

NoPackageAnalysis

指定 pack 不应在生成包后运行包分析。

MinClientVersion

指定可安装此包的最低 NuGet 客户端版本, 并由 nuget.exe 和 Visual Studio 程序包管理器强制实施。

IncludeBuildOutput

此布尔值指定是否应将生成输出程序集打包到 .nupkg 文件中。

IncludeContentInPack

此布尔值指定是否将含有 `Content` 类型的任何项自动包含在生成的包中。默认值为 `true`。

BuildOutputTargetFolder

指定放置输出程序集的文件夹。输出程序集(和其他输出文件)会复制到各自的框架文件夹中。

ContentTargetFolders

此属性指定放置所有内容文件的默认位置(如果未为其指定 `PackagePath`)。默认值为“content;contentFiles”。

NuspecFile

用于打包的 `.nuspec` 文件的相对或绝对路径。

NOTE

如果指定了 `.nuspec` 文件, 则将其用于打包信息, 并且不使用项目中的任何信息。

NuspecbasePath

`.nuspec` 文件的基路径。

NuspecProperties

键=值对的分号分隔列表。

AssemblyInfoProperties

现在, `AssemblyInfo` 文件中通常存在的[程序集特性](#)将自动从属性生成。

PropertiesPerAttribute

如下表所示, 每个特性都有一个可控制其内容的属性, 还有一个可以禁用其生成的属性:

属性	PROPERTY	生成器
<code>AssemblyCompanyAttribute</code>	<code>Company</code>	<code>GenerateAssemblyCompanyAttribute</code>
<code>AssemblyConfigurationAttribute</code>	<code>Configuration</code>	<code>GenerateAssemblyConfigurationAttribute</code>
<code>AssemblyCopyrightAttribute</code>	<code>Copyright</code>	<code>GenerateAssemblyCopyrightAttribute</code>
<code>AssemblyDescriptionAttribute</code>	<code>Description</code>	<code>GenerateAssemblyDescriptionAttribute</code>
<code>AssemblyFileVersionAttribute</code>	<code>FileVersion</code>	<code>GenerateAssemblyFileVersionAttribute</code>
<code>AssemblyInformationalVersionAttribute</code>	<code>InformationalVersion</code>	<code>GenerateAssemblyInformationalVersionAttribute</code>
<code>AssemblyProductAttribute</code>	<code>Product</code>	<code>GenerateAssemblyProductAttribute</code>
<code>AssemblyTitleAttribute</code>	<code>AssemblyTitle</code>	<code>GenerateAssemblyTitleAttribute</code>
<code>AssemblyVersionAttribute</code>	<code>AssemblyVersion</code>	<code>GenerateAssemblyVersionAttribute</code>
<code>NeutralResourcesLanguageAttribute</code>	<code>NeutralLanguage</code>	<code>GenerateNeutralResourcesLanguageAttribute</code>

注意:

- `AssemblyVersion` 和 `FileVersion` 默认采用 `$(Version)` 的值而不带后缀。例如, 如果 `$(Version)` 为

`1.2.3-beta.4`，则值将为 `1.2.3`。

- `InformationalVersion` 默认是 `$(Version)` 的值。
- 如果存在此属性，则 `InformationalVersion` 附加 `$(SourceRevisionId)`。可以使用 `IncludeSourceRevisionInInformationalVersion` 来禁用它。
- `Copyright` 和 `Description` 属性也可用于 NuGet 元数据。
- `Configuration` 与所有生成过程共享，并通过 `dotnet` 命令的 `--configuration` 参数进行设置。

GenerateAssemblyInfo

一个布尔值，用于启用或禁用所有 AssemblyInfo 生成。默认值为 `true`。

GeneratedAssemblyInfoFile

生成的程序集信息文件的路径。默认为 `$(IntermediateOutputPath)` (`obj`) 目录中的某个文件。

从 DNX 迁移到 .NET Core CLI (project.json)

2020/3/18 • [Edit Online](#)

概述

.NET Core 和 ASP.NET Core 1.0 RC1 版本中推出了 DNX 工具。.NET Core 和 ASP.NET Core 1.0 RC2 版本从 DNX 移动到了 .NET Core CLI。

温故知新，我们简单复习下 DNX 是什么。DNX 是用于生成 .NET Core(更具体点，是用于生成 ASP.NET Core 1.0 应用程序)的运行时和工具集。它主要由 3 个部分组成：

1. DNVM -- 用于获取 DNX 的安装脚本
2. DNX(Dotnet 执行运行时) - 执行代码的运行时
3. DNU(Dotnet 开发人员实用程序) - 用于管理依赖项、生成和发布应用程序的工具

引入 CLI 后，上述所有内容现在全部都属于单个工具集。但是，由于 DNX 在 RC1 时间范围内还可用，你可能具有某些使用该 DNX 生成的项目并且想要将其移动到新的 CLI 工具。

本迁移指南就介绍了关于如何将项目从 DNX 迁移到 .NET Core CLI 的基础知识。如果你才刚刚开始在 .NET Core 上从头创建项目，可选择跳过此文档。

工具的主要更改

首先将概述工具中存在的一般性更改。

不再具有 DNVM

DNVM(DotNet 版本管理器 的简称)，是用于在计算机上安装 DNX 的 bash/PowerShell 脚本。它有助于用户从指定的数据源(或默认数据源)获得需要的 DNX，以及将某个 DNX 标记为“活动”，从而将其置于给定会话的 \$PATH 中。这使你能够使用各种工具。

DNVM 现已停用，因为其功能集可能由于 .NET Core CLI 即将推出的更改变得冗余。

可以通过以下两种主要方式打包 CLI：

1. 给定平台的本机安装程序
2. 用于其他情形(如 CI 服务器)的安装脚本

因此，不再需要 DNVM 安装功能。但运行时选择功能又如何呢？

将某个版本的包添加到依赖项，可以引用 `project.json` 中的运行时。正因有此更改，应用程序将能够使用新的运行时数据。将这些数据引入计算机与安装 CLI 方法一样：通过其支持的本机安装程序之一或通过其安装脚本安装运行时。

命令不同

如果使用的是 DNX，则使用某些来自其三个部件(DNX、DNU 或 DNVM)之一的命令。借助 CLI，其中的某些命令发生了改变，有些命令不再适用，有些保持不变，但语义稍有不同。

下表显示了 DNX/DNU 命令及其 CLI 对应项之间的映射。

DNX	CLI	
dnx 运行	<code>dotnet run</code>	从源运行代码。

DNX 命令	CLI 命令	说明
dnu 生成	<code>dotnet build</code>	生成代码的 IL 二进制。
dnu 包	<code>dotnet pack</code>	打包代码的 NuGet 包。
dnx [command](例如, "dnx web")	不适用*	在 DNX 领域中, 按照 project.json 中的定义运行命令。
dnu 安装	不适用*	在 DNX 领域中, 将包安装为一个依赖项。
dnu 还原	<code>dotnet restore</code>	还原 project.json 中指定的依赖项。 (请参阅注释)
dnu 发布	<code>dotnet publish</code>	使用三种形式(可移植、本机可移植和独立形式)之一发布应用程序以进行部署。
dnu 包装	不适用*	在 DNX 领域中, 包装 csproj 中的 project.json。
dnu 命令	不适用*	在 DNX 领域中, 管理全局安装的命令。

(*) - 按照设计, CLI 中不支持这些功能。

不支持 DNX 功能

如上表所示, CLI 中不支持(至少暂时不支持)来自 DNX 领域的某些功能。本部分将介绍最重要的这部分内容, 并概述不支持它们的根本原因, 以及如果确实需要某些功能时相应的解决办法。

全局命令

DNX 附带称为“全局命令”的概念。从本质上来说, 这些都是打包成 NuGet 包的控制台应用程序, 其中 shell 脚本可以调用指定用于运行应用程序的 DNX。

CLI 不支持此概念。但是, 它确实支持添加所有项目命令的这一概念, 这些命令可以使用熟悉的 `dotnet <command>` 语法调用。

安装依赖项

自 v1 起, .NET Core CLI 就没有用于安装依赖项的 `install` 命令。为了从 NuGet 安装包, 需要将其作为依赖项添加到 `project.json` 文件, 然后运行 `dotnet restore` ([请参阅注释](#))。

运行代码

运行代码主要有两种方法。一种是从源中使用 `dotnet run` 运行。与 `dnx run` 不同, 这种方法不能执行任何内存中编译。实际上, 它将调用 `dotnet build` 生成代码, 然后运行生成的二进制文件。

另一种是使用 `dotnet` 自身运行代码。可通过提供程序集路径实现: `dotnet path/to/an/assembly.dll`。

将 DNX 项目迁移到 .NET Core CLI

使用代码时, 除了使用新的命令, 从 DNX 迁移还剩三件主要的事情:

1. 若要使其能够使用 CLI, 请迁移 `global.json` 文件。
2. 将项目文件(`project.json`)本身迁移到 CLI 工具。
3. 将任何 DNX API 迁移到 BCL 对应项。

更改 global.json 文件

对于 RC1 和 RC2(或更高版本)项目, `global.json` 文件充当两者的解决方案文件。为了在 RC1 和更高版本间区分 .NET Core CLI(以及 Visual Studio), 可以使用 `"sdk": { "version" }` 属性来区分哪个项目是 RC1 或更高版本。如果 `global.json` 根本无此节点, 则假定为最新版本。

为了更新 `global.json` 文件, 可以删除此属性或将其设置为想要使用的工具的确切版本, 在本示例中为 `1.0.0-preview2-003121`:

```
{  
  "sdk": {  
    "version": "1.0.0-preview2-003121"  
  }  
}
```

迁移项目文件

CLI 和 DNX 都使用基于 `project.json` 文件的相同基本项目系统。项目文件的语法和语义大致相同, 但根据应用场景, 会略有不同。此外还对架构进行了一些更改, 可在[架构文件](#)中查看这些更改。

如果要生成控制台应用程序, 需要将以下代码段添加到项目文件:

```
"buildOptions": {  
  "emitEntryPoint": true  
}
```

这会指示 `dotnet build` 发出应用程序入口点, 可以有效地使代码具有可运行性。如果要生成类库, 则可以直接省略以上内容。当然, 将上述代码段添加到 `project.json` 文件后, 需要添加静态入口点。从 DNX 迁移后, 它提供的 DI 服务将不再可用, 因此需要属于基本 .NET 入口点: `static void Main()`。

如果 `project.json` 中有“命令”部分, 可将其删除。过去作为 DNU 命令(例如, 实体框架 CLI 命令)存在的某些命令, 将作为每个项目的扩展移植到 CLI。如果生成了自己正在项目中使用的命令, 需要使用 CLI 扩展将其替换。在这种情况下, `commands` 中的 `project.json` 节点需要替换为 `tools` 节点, 并且需要列出工具依赖项。

完成这些操作后, 需要决定希望应用使用的可移植性类型。借助 .NET Core, 我们在提供一系列可从中进行选择的可移植性选项方面投入了大量工作。例如, 可能想要一个完全可移植 的应用程序或者想要一个独立的 应用程序。可移植应用程序选项工作原理更像 .NET Framework 应用程序的工作原理: 它需要共享组件才能在目标计算机 (.NET Core) 上执行。独立应用程序不需要在目标上安装 .NET Core, 但需要为每个要支持的 OS 生成一个应用程序。有关这些可移植性类型等内容, 请参阅[应用程序可移植性类型](#)文档。

调用想要的可移植性类型后, 需要更改目标框架。如果是为 .NET Core 编写应用程序, 很可能要使用 `dnxcore50` 作为目标框架。鉴于 CLI 和全新 [.NET Standard](#) 引入的变化, 框架需要为下列之一:

1. `netcoreapp1.0` - 如果要在 .NET Core 上编写应用程序(包括 ASP.NET Core 应用程序)
2. `netstandard1.6` - 如果要为 .NET Core 编写类库

如果要使用其他 `dnx` 目标, 如 `dnx451`, 则还需要更改这些内容。`dnx451` 应更改为 `net451`。有关详细信息, 请参阅[.NET Standard](#) 主题。

`project.json` 现已差不多准备就绪。需要浏览依赖项列表并将依赖项更新至最新版本, 尤其在使用 ASP.NET Core 依赖项时。如果使用的是 BCL API 的单独包, 可使用运行时包, 如[应用程序可移植性类型](#)文档中所述。

准备就绪后, 就可以尝试使用 `dotnet restore` ([请参阅注释](#))进行还原。具体取决于依赖项版本, 如果 NuGet 无法解析上述目标框架的依赖项, 可能会遇到错误。这是一个“时间点”问题; 随着时间的推移, 支持这些框架的包会越来越多。目前, 如果遇到此类问题, 可以使用 `imports` 节点内的 `framework` 语句指定到 NuGet, 还原“imports”语句内面向该框架的包。此种情况下遇到的还原错误可提供足够的信息, 告知需要导入的框架类型。一般而言, 如果对此感到迷惑或不熟悉, 在 `dnxcore50` 语句中指定 `portable-net45+win8` 和 `imports` 应该有效。下面的 JSON 代码

段显示了这种情况是怎样的：

```
"frameworks": {  
    "netcoreapp1.0": {  
        "imports": ["dnxcore50", "portable-net45+win8"]  
    }  
}
```

运行 `dotnet build` 会显示任何最终的生成错误，但应该不会有很多。生成代码并正常运行后，可以使用运行程序测试它。执行 `dotnet <path-to-your-assembly>` 并查看其运行状况。

NOTE

从 .NET Core 2.0 SDK 开始，无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build` 和 `dotnet run`。在执行显式还原有意义的某些情况下，例如 [Azure DevOps Services 中的持续集成生成中](#)，或在需要显式控制还原发生时间的生成系统中，它仍然是有效的命令。

从 .NET Framework 移植到 .NET Core 的概述

2020/3/18 • [Edit Online](#)

你可能有些代码当前正在 .NET Framework 上运行，但你想将这些代码移植到 .NET Core。本文提供以下内容：

- 移植过程概述。
- 在将代码移植到 .NET Core 时，可能会发现一系列有用的工具。

移植过程概述

建议在将项目移植到 .NET Core 时使用以下过程：

1. 将希望移植的所有项目重定向到目标 .NET Framework 4.7.2 或更高版本。

此步骤可确保在 .NET Core 不支持特殊 API 的情况下，可以为特定于 .NET Framework 的目标使用备用 API。

2. 使用 [.NET 可移植性分析器](#) 来分析程序集，并查看这些程序集是否可移植到 .NET Core。

API 可移植性分析器工具可分析已编译的程序集并生成报表。此报表显示高级别可移植性摘要，以及你所使用的不适用于 .NET Core 的各个 API 细目。

3. 将 [.NET API 分析器](#) 安装到项目中，以识别在某些平台上会引发 `PlatformNotSupportedException` 的 API 以及一些其他潜在的兼容性问题。

此工具与可移植性分析器类似，但它不会分析是否可以在 .NET Core 上构建代码，而是分析你是否正在以会导致在运行时引发 `PlatformNotSupportedException` 的某种方式使用 API。尽管这并不常见，但如果从 .NET Framework 4.7.2 或更高版本进行移动，最好进行检查。如需详细了解会在 .NET Core 上引发异常的 API 信息，请参阅[在 .NET Core 上始终引发异常的 API](#)。

4. 通过 `packages.config` Visual Studio 中的转换工具将所有 依赖项转换为 `PackageReference` 格式。

此步骤涉及到转换旧 `packages.config` 格式的依赖项。`packages.config` 不适用于 .NET Core，因此，如果你有包依赖项，则需要进行此转换。

5. 为 .NET Core 创建新项目并覆盖源文件，或尝试使用工具转换现有的项目文件。

.NET Core 使用不同于 .NET Framework 的更简化的[项目文件格式](#)。需要将项目文件转换为此格式才能继续操作。

6. 移植测试代码。

由于移植到 .NET Core 对代码库来说是很大的更改，因此强烈建议移植测试项目，以便在移植代码后运行测试。MSTest、xUnit 和 NUnit 都适用于 .NET Core。

此外，可以尝试使用 `dotnet try-convert` 工具通过一次操作将较小的解决方案或单个项目移植到 .NET Core 项目文件格式。不能保证 `dotnet try-convert` 适用于所有项目，它可能导致所依赖的行为发生细微变化。使用它作为起点，以自动化可自动执行的基本操作。作为用于迁移项目的一种解决方案，它并不是万无一失的。

后续步骤

[分析依赖项](#)

分析依赖项将代码移植到 .NET Core

2020/3/18 • [Edit Online](#)

若要将代码移植到 .NET Core 或.NET Standard, 必须了解依赖项。外部依赖项是在项目中引用但没有自行构建的 NuGet 包或 `.dll` 文件。

将 NuGet 包迁移到 `PackageReference`

.NET Core 使用 `PackageReference` 来指定包依赖项。如果使用 `packages.config` 在项目中指定包, 则将其转换为 `PackageReference` 格式, 因为 .NET Core 不支持 `packages.config`。

如需了解如何迁移, 请参阅[从 `packages.config` 迁移到 `PackageReference`](#)一文。

升级 NuGet 包

将项目迁移为 `PackageReference` 格式后, 验证包是否与 .NET Core 兼容。

首先, 请将包升级到可以使用的最新版本。可通过 Visual Studio 中的 NuGet 包管理器 UI 完成此操作。包依赖项的较新版本可能已经与 .NET Core 兼容。

分析包依赖项

如果尚未验证转换和升级后的包依赖项是否适用于 .NET Core, 可通过以下几种方法实现此目的:

使用 nuget.org 分析 NuGet 包

可以在包页面“依赖项”部分的 [nuget.org](#) 上查看每个包所支持的目标框架名字对象 (TFM)。

尽管使用站点验证兼容性是一种比较简单的方法, 但站点上并未提供所有包的“依赖项”信息。

使用 NuGet 包资源管理器分析 NuGet 包

NuGet 包本身就是一组包含特定于平台的程序集的文件夹。检查包中是否有包含兼容程序集的文件夹。

检查 NuGet 包文件夹最简单方法是使用 [NuGet 包资源管理器](#) 工具。安装完成后, 使用以下步骤查看文件夹名称:

1. 打开 NuGet 包资源管理器。
2. 单击“从在线源打开包”。
3. 搜索包的名称。
4. 从搜索结果中选择包的名称, 然后点击“打开”。
5. 展开右侧的“lib”文件夹并查看文件夹名称。

使用以下模式之一查找具有名称的文件夹: `netstandardX.Y` 或 `netcoreappX.Y`。

这些值是映射到 [.NET Standard](#) 版本的[目标框架名字对象 \(TFM\)](#)、.NET Core 以及与 .NET Core 兼容的传统可移植类库 (PCL) 配置文件。

IMPORTANT

查看包支持的 TFM 时, 请注意 `netcoreapp*`, 它在兼容时, 仅适用于 .NET Core 项目, 不适用于 .NET Standard 项目。仅面向 `netcoreapp*` 而不是 `netstandard*` 的库只能用于其他 .NET Core 应用。

.NET Framework 兼容性模式

在分析完 NuGet 包之后，你可能会发现它们仅面向 .NET Framework。

从 .NET Standard 2.0 开始，引入了 .NET Framework 兼容性模式。此兼容性模式允许 .NET Standard 和 .NET Core 项目引用 .NET Framework 库。引用 .NET Framework 库不适用于所有项目（如库使用 Windows Presentation Foundation (WPF) API 时），但它的开启了很多移植方案。

在项目中引用面向 .NET Framework 的 NuGet 包时（如 [Huitian.PowerCollections](#)），会收到与以下示例类似的包回退警告（NU1701）：

```
NU1701: Package 'Huitian.PowerCollections 1.0.0' was restored using '.NETFramework,Version=v4.6.1' instead of the project target framework '.NETStandard,Version=v2.0'. This package may not be fully compatible with your project.
```

当添加包以及每次生成时都会显示该警告，以确保在项目中测试该包。如果项目按预期工作，可通过在 Visual Studio 中编辑包属性或在最喜欢的代码编辑器中手动编辑项目文件来取消该警告。

若要通过编辑项目文件来取消警告，请找到要取消警告的包的 `PackageReference` 条目并添加 `NoWarn` 属性。

`NoWarn` 属性接受所有警告 ID 的逗号分隔列表。以下示例显示如何通过手动编辑项目文件来取消 `Huitian.PowerCollections` 包的 NU1701 警告：

```
<ItemGroup>
  <PackageReference Include="Huitian.PowerCollections" Version="1.0.0" NoWarn="NU1701" />
</ItemGroup>
```

有关如何在 Visual Studio 中取消编译器警告的详细信息，请参阅[取消 NuGet 包的警告](#)。

NuGet 包依赖项未在.NET Core 上运行时应执行的操作

如果所依赖的 NuGet 包无法在 .NET Core 上运行，可以执行以下几项操作：

- 如果项目是开放源代码并托管在诸如 GitHub 中，则可以直接与开发人员交流。
- 可直接在 [nuget.org](#) 上联系作者。搜索包并单击包页面左侧的“联系所有者”。
- 可以搜索在 .NET Core 上运行的其他包，它与所使用的包进行的是相同的任务。
- 可以尝试自己编写包所执行的代码。
- 可以通过更改应用的功能来清除对包的依赖性，至少在该包有可用的兼容性版本之前都能这样做。

请注意，开源项目维护人员和 NuGet 包发布者通常是志愿者。他们因为关注某个特定领域而免费提供服务，通常从事另外一份职业。当联系他们请求 .NET Core 支持时请注意这点。

如果使用上述任何选项都无法解决问题，则可能需要移植到最近版本的 .NET Core。

.NET 团队需要了解哪些库最需要使用 .NET Core 支持。你可以发送电子邮件到 dotnet@microsoft.com，谈谈你想使用的库。

分析不是 NuGet 包的依赖项

用户可能拥有不是 NuGet 包的依赖项，如文件系统中的 DLL。确定该依赖项是否可移植的唯一方法是运行 [.NET 可移植性分析器](#) 工具。该工具可以分析面向 .NET Framework 的程序集，并识别不可移植到其他 .NET 平台（如 .NET Core）的 API。可将该工具作为控制台应用程序或作为 [Visual Studio 扩展](#) 来运行。

后续步骤

[移植库](#)

将 .NET Framework 库移植到 .NET Core

2020/3/18 • [Edit Online](#)

了解如何将 .NET Framework 库代码移植到 .NET Core，它在其中跨平台运行并扩展使用它的应用的范围。

系统必备

本文假定你：

- 正在使用 Visual Studio 2017 或更高版本。(Visual Studio 的早期版本不支持 .NET Core。)
- 了解[推荐的移植过程](#)。
- 已解决与[第三方依赖项](#)有关的任何问题。

还应熟悉下列文章的内容：

[.NET Standard](#)

本文介绍了适用于所有 .NET 实现代码的 .NET API 正式规范。

[包、元包和框架](#)

这篇文章介绍了 .NET Core 如何定义和使用包，以及包如何支持多个 .NET 实现代码。

[使用跨平台工具开发库](#)

本文介绍如何使用 .NET Core CLI 编写库。

[.NET Core 的 csproj 格式的新增内容](#)

本文概述了作为从移动到 csproj 和 MSBuild 的一部分，添加到项目文件的更改。

[移植到 .NET Core - 分析第三方依赖项](#)

本文介绍了第三方依赖项的可移植性及 NuGet 包依赖项无法在 .NET Core 上运行时要执行的操作。

重定向到 .NET Framework 4.7.2

如果代码不面向 .NET Framework 4.7.2，建议重定向到 .NET Framework 4.7.2。在 .NET Standard 不支持现有 API 情况下，这可确保最新备用 API 的可用性。

对于每个想要移植的项目，请在 Visual Studio 中执行以下操作：

- 右键单击该项目，然后选择“属性”。
- 在“目标框架”下拉列表中，选择“.NET Framework 4.7.2”。
- 重新编译该项目。

因为项目现在面向 .NET Framework 4.7.2，因此可使用该版本的 .NET Framework 作为移植代码的基准。

确定可移植性

下一步是运行 API 可移植性分析器 (ApiPort) 生成可供分析的可移植性报表。

确保了解[API 可移植性分析器 \(ApiPort\)](#) 及如何生成用于面向 .NET Core 的可移植性报表。执行此操作的方式可能取决于需求和个人偏好。以下部分详细介绍了几种不同的方法。用户可能会发现自己根据生成代码的方式混合使用了这些方法中的步骤。

[主要处理编译器](#)

此方法可能较适合小项目或不会用很多 .NET Framework API 的项目。此方法很简单：

1. 可选择在项目上运行 ApiPort。若运行 ApiPort，则从报告获取有关需要解决的问题的信息。
2. 将所有代码复制到新的 .NET Core 项目。
3. 查看可移植性报表(如果已生成)时，解决编译器错误，直至项目完全得到编译。

尽管它是非结构化的，但这种以代码为中心的方法通常可以快速解决问题。只包含数据模型的项目可能是此方法的理想选择。

可移植性问题得到解决前停留在 .NET Framework 上

如果更希望拥有在整个过程期间编译的代码，此方法可能是最佳选择。该方法如下所示：

1. 在项目上运行 ApiPort。
2. 通过使用可移植的不同 API 解决问题。
3. 记录阻止你使用直接替代方案的所有区域。
4. 对所有要移植的项目重复前面的步骤，直到确信每个项目都做好被复制到新的 .NET Core 项目中的准备。
5. 将代码复制到新的 .NET Core 项目。
6. 解决所有已记录的不存在直接替代方案的问题。

这种谨慎的方法比单纯解决编译器错误更有条理，但相对而言，它仍以代码为中心，且优点是始终拥有编译的代码。解决不能通过只使用另一个 API 解决的某些问题的方法大不相同。你可能会发现对于某些项目，需要制定更全面的计划，这将在下一种方法中介绍。

制定全面的攻击计划

此方法可能最适合大型或更复杂的项目，在这种情况下，为支持 .NET Core，可能必需重构代码或将某些代码区域完全重写。该方法如下所示：

1. 在项目上运行 ApiPort。
2. 了解每个非可移植类型使用的位置以及位置对整体可移植性的影响。
 - 了解这些类型的特性。它们是否数量少，但使用频繁？它们是否数量大，但使用不频繁？它们是串联使用，还是在整个代码中传播？
 - 是否可以轻松隔离不可移植的代码，以便可以更有效地处理它？
 - 是否需要重构代码？
 - 对于这些不可移植的类型，是否存在可完成相同任务的备用 API？例如，如果使用 `WebClient` 类，也许能够改用 `HttpClient` 类。
 - 是否存在其他可用于完成任务的可移植 API，即使它不是直接替代 API？例如，如果使用 `XmlSchema` 来分析 XML，但是无需 XML 架构发现，则可使用 `System.Xml.Linq` API 并自行实现分析，而不依赖于 API。
3. 如果具有难以移植的程序集，是否值得将其暂时留在 .NET Framework 上？以下是一些需要考虑的事项：
 - 库中可能具有某些与 .NET Core 不兼容的功能，因为它太依赖 .NET Framework 或 Windows 特定的功能。是否值得暂时搁置该功能，并直到资源可用于移植这些功能，才发布具有较少功能的库的临时 .NET Core 版本？
 - 重构是否有用？
4. 编写自己对不可用 .NET Framework API 的实现是否合理？可以考虑复制、修改，并使用 [.NET Framework 参考源](#) 中的代码。参考源代码已在 [MIT 许可证](#) 下获得许可，因此可以自由选择将此源作为自己代码的基础。只需确保在代码中正确设置 Microsoft。
5. 根据不同项目的需要，重复此过程。

分析阶段可能需要一些时间，具体取决于代码库的大小。在此阶段花费时间（尤其是在具有复杂的代码库时），全面了解所需的更改范围并制定计划，从长远看通常可节省时间。

计划可能包括对代码库做重大更改，同时面向 .NET Framework 4.7.2，使它成为前一种方法更有条理的版本。着手执行计划的方式具体取决于代码库。

混合方法

在每个项目的基础上，可能会将上述方法进行混合。应该进行对你和代码库最有意义的操作。

移植测试

要确保移植代码后一切正常的最佳方式是在将代码移植到 .NET Core 时进行测试。为此，需要使用将针对 .NET Core 生成和运行测试的测试框架。当前，有三个选择：

- [xUnit](#)
 - [入门](#)
 - [将 MSTest 项目转换为 xUnit 的工具](#)
- [NUnit](#)
 - [入门](#)
 - [关于从 MSTest 迁移到 NUnit 的博客文章](#)
- [MSTest](#)

推荐的方法

从根本上讲，移植工作在很大程度上取决于生成 .NET Framework 代码的方式。移植代码的一个好方法是从库的基项开始，这是代码的基础组件。这可能是数据模型或某些其他内容直接或间接使用的基本类和方法。

1. 移植测试项目，该项目测试目前正在移植的库层。
2. 将库中的基项复制到新的 .NET Core 项目，然后选择想要支持的 .NET Standard 版本。
3. 进行任何所需的更改，使代码进行编译。大部分内容可能会要求将 NuGet 包依赖项添加到 csproj 文件。
4. 运行测试并进行任何所需调整。
5. 选择下一层代码进行移植，并重复前面的步骤。

如果从库的基项开始并从基项向外移动并根据需要测试每一层，移植将是一个系统化的过程，在这种情况下，问题可以一次隔离到一层代码中。

后续步骤

[整理 .NET Core 的项目](#)

组织项目以支持 .NET Framework 和 .NET Core

2020/3/18 • [Edit Online](#)

可以创建一个并行编译 .NET Framework 和 .NET Core 的解决方案。本文介绍可帮助你实现此目标的多个项目组织选项。以下是决定如何使用 .NET Core 设置项目布局时要考虑的一些典型方案。此列表可能无法涵盖所有要求；这些方案的优先级具体取决于项目需求。

- [将现有项目和 .NET Core 项目合并为单个项目](#)

此方案的好处：

- 通过编译单个项目（而非多个项目）简化生成过程，每个项目针对不同的 .NET Framework 版本或平台。
- 由于需要管理单个项目文件，因此简化了多目标项目的源文件管理。添加或删除源文件时，需要手动将备用文件与其他项目进行同步。
- 轻松生成可供使用的 NuGet 包。
- 允许通过使用编译器指令为库中特定的 .NET Framework 版本编写代码。

不支持的方案：

- 要求开发者使用 Visual Studio 2017 或更高版本来打开现有项目。若要支持 Visual Studio 的早期版本，建议[将项目文件保存在不同的文件夹中](#)。

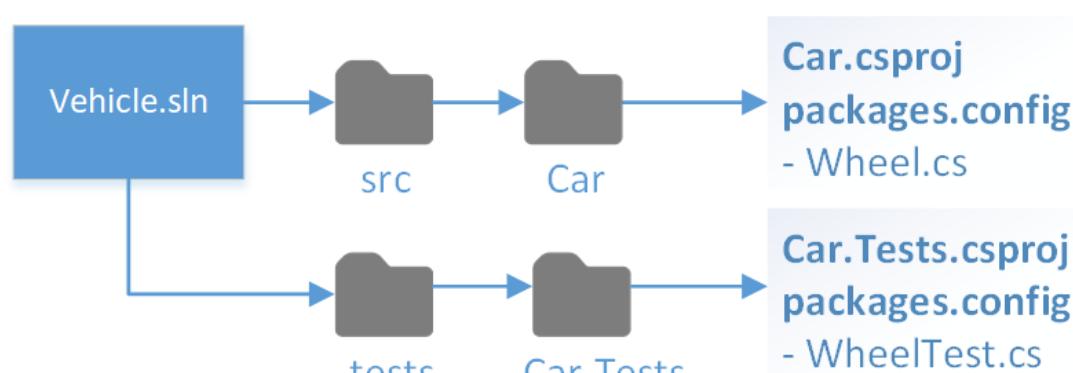
- [将现有项目和新的 .NET Core 项目分离](#)

此方案的好处：

- 支持没有安装 Visual Studio 2017 或更高版本的开发人员和参与者基于现有项目开发。
- 减少现有项目中出现新 bug 的可能性，因为这些项目中不需要进行任何代码改动。

示例

请考虑以下存储库：

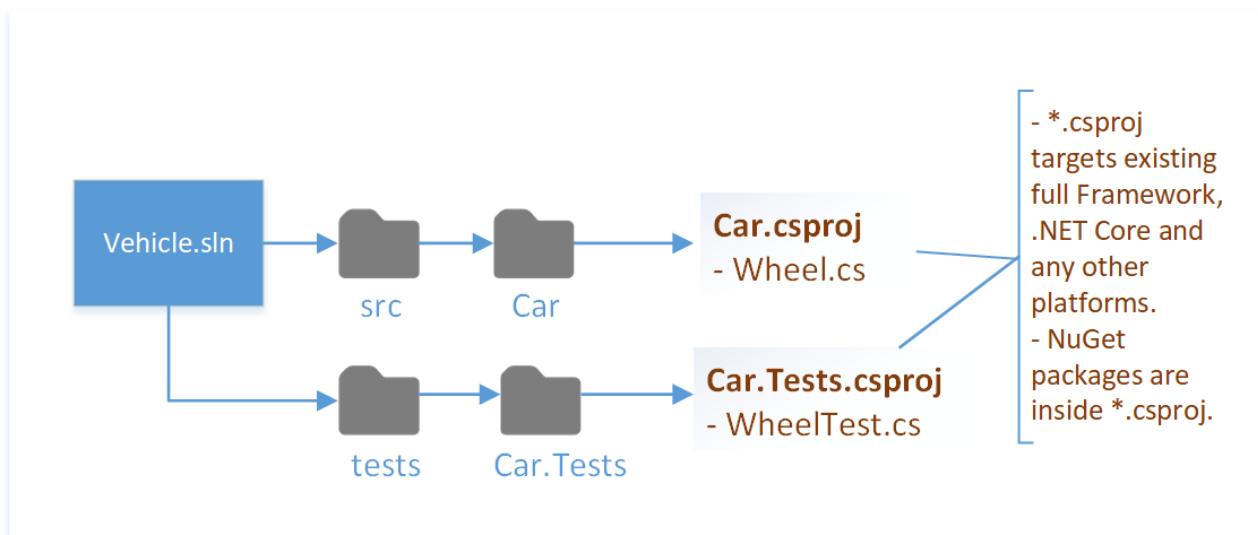


源代码

根据现有项目的约束和复杂性，有几种不同的方法可为此存储库添加对 .NET Core 的支持，下面描述了这些方法。

将现有项目替换为多目标的 .NET Core 项目

重新组织存储库，以便删除任何现有的 `.csproj*` 文件，并创建以多个框架为目标的单一 `.csproj*` 文件。这是一项不错的选择，因为单个项目可以编译不同的框架。它还可以处理每个目标框架的不同编译选项和依赖项。



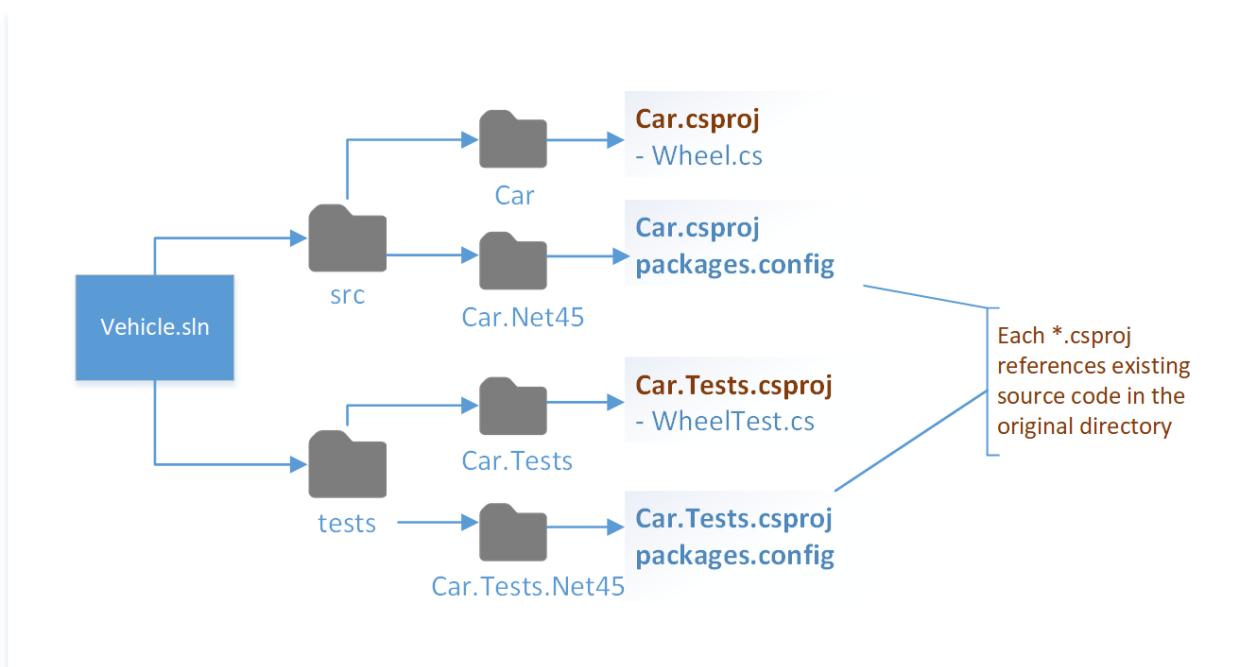
源代码

需注意的更改：

- 用新的 `.NET Core .csproj` 替换 `packages.config` 和 `*.csproj*`。NuGet 包是使用 `<PackageReference>` `ItemGroup` 指定的。

保留现有项目并创建 .NET Core 项目

如果存在以较旧框架为目标的现有项目，可能需要保留这些项目并将 .NET Core 项目用作将来框架的目标。



源代码

将 .NET Core 项目和现有项目保存在不同的文件夹中。将项目保存在不同的文件夹中可以避免强制使用 Visual Studio 2017 或更高版本。可以创建仅打开旧项目的单独解决方案。

另请参阅

- [.NET Core 移植文档](#)

.NET Framework 技术在 .NET Core 上不可用

2020/3/18 • [Edit Online](#)

一些适用于 .NET Framework 库的技术不可用于 .NET Core，例如 AppDomains、远程处理、代码访问安全性 (CAS)、安全透明度和 System.EnterpriseServices。如果库依赖于这些技术中的一个或多个，请考虑使用下面所述的替代方法。有关 API 兼容性的详细信息，请参阅 [.NET Core 中断性变更](#)。

当前未实现某个 API 或技术并不因此意味着有意不对其提供支持。搜索 .NET Core 的 GitHub 存储库，查看所遇到的特定问题是否是特意设计的。如果找不到此类指示器，请在 [dotnet/runtime 存储库](#) 中提出问题，请求提供特定 API 和技术。移植请求的问题会标记为 [端口到核心](#) 标签。

AppDomain

应用程序域 (AppDomain) 可将应用相互隔离。AppDomain 需要运行时支持并且通常价格昂贵。不支持创建其他应用域，也尚未计划在将来添加此功能。对于代码隔离，将流程或容器用作备用。若要动态加载程序集，请使用 [AssemblyLoadContext](#) 类。

.NET Core 公开了一些 [AppDomain](#) API 曲面，以便可以更轻松地从 .NET Framework 进行代码迁移。一些 API 可正常工作（例如 [AppDomain.UnhandledException](#)），一些成员不会执行任何操作（例如 [SetCachePath](#)），也有一些会引发 [PlatformNotSupportedException](#)（例如 [CreateDomain](#)）。对照 [dotnet/runtime GitHub 存储库](#) [System.AppDomain](#) 中的 [引用源](#) 检查所使用的类型。确保选择与已实现的版本相匹配的分支。

远程处理

.NET 远程处理被认为是存在问题的体系结构。它用于进行跨 AppDomain 的通信，该通信现已不再受支持。同样，远程处理也需要运行时支持，进行维护的成本较高。出于这些原因，.NET Core 不支持 .NET 远程处理，并且我们不计划在将来添加对它的支持。

对于跨进程通信，可将进程间通信 (IPC) 机制视为远程处理的备用方案，如 [System.IO.Pipes](#) 类或 [MemoryMappedFile](#) 类。

对于跨计算机的通信，可将基于网络的解决方案用作备用方案。最好使用低开销纯文本协议，例如 HTTP。此外，ASP.NET Core 使用的 Web 服务器 [Kestrel Web 服务器](#) 是一个选择。也可考虑将 [System.Net.Sockets](#) 用于基于网络的跨计算机的方案。请参阅 [.NET 开放源代码开发人员项目:消息传递](#) 了解更多选项。

代码访问安全性 (CAS)

沙盒依赖为托管应用程序或库使用或运行提供资源的运行时或框架进行限制，其在 [.NET Framework 上不受支持](#)，因此在 .NET Core 上也不受支持。.NET Framework 和运行时中存在太多发生特权提升以继续将 CAS 视为安全边界的情况。此外，CAS 使实现更加复杂，通常会对无意使用它的应用程序造成正确性-性能影响。

可使用操作系统提供的安全边界，例如虚拟化、容器或具有最少特权集的用于运行进程的用户帐户。

安全透明度

与 CAS 相似，借助安全透明度可以以声明性方式将沙盒代码与安全关键代码隔离，但是 [不再支持将它作为安全边界](#)。Silverlight 大规模使用了此功能。

可使用操作系统提供的安全边界，例如虚拟化、容器或用于运行进程的用户帐户具有最少的一组特权。

System.EnterpriseServices

.NET Core 不支持 System.EnterpriseServices (COM+)。

另请参阅

- [有关从 .NET Framework 移植到 .NET Core 的概述](#)

用于帮助移植到 .NET Core 的工具

2020/4/13 • [Edit Online](#)

你可能会发现本文中列出的工具在移植时非常有用：

- [.NET 可移植性分析器](#) - 可以生成报告的工具链，该报告说明代码在 .NET Framework 和 .NET Core 之间的可移植性：
 - 作为[命令行工具](#)
 - 作为[Visual Studio 扩展](#)
- [.NET API 分析器](#) - 一个 Roslyn 分析器，可用于发现不同平台上的潜在 C# API 兼容性风险，并检测是否调用了弃用的 API。

此外，可以尝试使用 [CsprojToVs2017](#) 工具将较小的解决方案或单个项目移植到 .NET Core 项目文件格式。

WARNING

CsprojToVs2017 是第三方工具。不能保证它适用于所有项目，而且它可能会导致所依赖的行为发生细微变化。CsprojToVs2017 应作为一个起点，以自动化可自动执行的基本操作。它不是迁移项目文件格式的有保证的解决方案。

使用 Windows 兼容性包将代码移植到 .NET Core

2020/3/30 • [Edit Online](#)

将现有代码移植到 .NET Core 时发现的一些最常见问题依赖于仅在 .NET Framework 中找到的 API 和技术。Windows 兼容性包 提供许多这些技术，因此可以更轻松地生成 .NET Core 应用程序和 .NET Standard 库。

兼容包是 [.NET Standard 2.0 的逻辑扩展](#)，它大幅扩展了 API 集。现有代码几乎不修改即可编译。为了信守 .NET Standard 的承诺（“所有 .NET 实现都提供的一组 API”），.NET Standard 不包括无法跨所有平台工作的技术，如注册表、Windows Management Instrumentation (WMI) 或反射发出 API。Windows 兼容性包位于 .NET Standard 顶部，提供对这些仅限 Windows 的技术的访问权限。它对于想要移动到 .NET Core 但至少第一步仍计划停留在 Windows 上的客户尤其有用。在这种情况下，能够使用仅限 Windows 的技术可消除迁移障碍。

包的内容

Windows 兼容性包通过 [Microsoft.Windows.Compatibility NuGet 包](#) 提供，可从面向 .NET Core 或 .NET Standard 的项目引用。

它提供了约 20,000 个 API，包括仅 Windows API 以及以下技术领域中的跨平台 API：

- 代码页
- CodeDom
- Configuration
- 目录服务
- 绘图
- ODBC
- 权限
- 端口
- Windows 访问控制列表 (ACL)
- Windows Communication Foundation (WCF)
- Windows 加密
- Windows 事件日志
- Windows 管理规范 (WMI)
- Windows 性能计数器
- Windows 注册表
- Windows 运行时缓存
- Windows 服务

有关详细信息，请参阅[兼容包规范](#)。

入门

1. 移植之前，请确保查看[移植过程](#)。
2. 将现有代码移植到 .NET Core 或 .NET Standard 时，请安装 [Microsoft.Windows.Compatibility NuGet 包](#)。

如果要停留在 Windows 上，则已准备完毕。

3. 如果要在 Linux 或 macOS 上运行 .NET Core 应用程序或 .NET Standard 库，请使用 [API 分析器](#)查找不会跨平台工作的 API 的使用情况。

4. 删 除这些 API 的使用情况、将其替换为跨平台替代项，或使用平台检查对其实施保护，例如：

```
private static string GetLoggingPath()
{
    // Verify the code is running on Windows.
    if (RuntimeInformation.OperatingSystem.PlatformType == PlatformType.Windows)
    {
        using (var key = Registry.CurrentUser.OpenSubKey(@"Software\Fabrikam\AssetManagement"))
        {
            if (key?.GetValue("LoggingDirectoryPath") is string configuredPath)
                return configuredPath;
        }
    }

    // This is either not running on Windows or no logging path was configured,
    // so just use the path for non-roaming user-specific data files.
    var appDataPath = Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData);
    return Path.Combine(appDataPath, "Fabrikam", "AssetManagement", "Logging");
}
```

有关演示，请查看 [Windows 兼容性包的第 9 频道视频](#)。

如何将 Windows 窗体桌面应用程序移植到 .NET Core

2020/4/9 • [Edit Online](#)

本文介绍如何将基于 Windows 窗体的桌面应用从 .NET Framework 移植到 .NET Core 3.0 或更高版本。.NET Core 3.0 SDK 支持 Windows 窗体应用程序。Windows 窗体仍是仅适用于 Windows 的框架，并且只能在 Windows 上运行。示例使用 .NET Core SDK CLI 创建和管理项目。

本文中的各种名称用于标识迁移所用的文件类型。迁移项目时，你的文件将以不同的名称命名，因此，请自行在心里将它们与下面列出的文件进行匹配：

文件	描述
MyApps.sln	解决方案文件的名称。
MyForms.csproj	要移植的 .NET Framework Windows 窗体项目的名称。
MyFormsCore.csproj	创建的新 .NET Core 项目的名称。
MyAppCore.exe	.NET Core Windows 窗体应用程序的可执行文件。

先决条件

- 适用于要执行的任何设计器工作的 [Visual Studio 2019 16.5 预览版 1](#) 或更高版本。建议更新到最新的 [Visual Studio 预览版](#)
安装以下 Visual Studio 工作负载：
 - .NET 桌面开发
 - .NET Core 跨平台开发
- 在解决方案中顺利生成和运行的有效 Windows 窗体项目。
- 用 C# 编码的项目。

NOTE

仅在 Visual Studio 2019 或更高版本中支持 .NET Core 3.0 项目。从 Visual Studio 2019 版本 16.5 预览版 1 开始，还支持 .NET Core Windows 窗体设计器。

若要启用该设计器，请转到“工具”>“选项”>“环境”>“预览功能”，然后选择“将预览版 Windows 窗体设计器用于 .NET Core 应用”选项。

考虑

移植 .NET Framework Windows 窗体应用程序时，必须考虑以下几个事项。

- 检查应用程序是否适合迁移。

使用 [.NET 可移植性分析器](#) 确定项目将会迁移到 .NET Core 3.0。如果项目存在 .NET Core 3.0 相关问题，分析器可帮助你识别这些问题。

- 使用的是不同版本的 Windows 窗体。

发布 .NET Core 3.0 预览版 1 时，Windows 窗体在 GitHub 上公布了开放源代码。.NET Core Windows 窗体的代码是 .NET Framework Windows 窗体基本代码的一个分支。可能存在一些差异导致应用程序无法移植。

3. 使用 [Windows 兼容包](#) 可有助于进行迁移。

某些 API 可在 .NET Framework 中使用，但不可用于 .NET Core 3.0。[Windows 兼容包](#)增加了很多这些 API，有助于使 Windows 窗体应用与 .NET Core 兼容。

4. 更新项目使用的 NuGet 包。

在执行任何迁移之前使用最新版 NuGet 包始终是一个不错的做法。如果应用程序引用任何 NuGet 包，请将它们更新到最新版本。确保成功生成应用程序。升级后，如果存在任何包错误，请将包降级到不会破坏代码的最新版本。

创建新的 SDK 项目

创建的新 .NET Core 3.0 项目必须位于不同于 .NET Framework 项目的目录中。如果它们位于同一目录中，可能会与 obj 目录中生成的文件发生冲突。本例中，我们将在 SolutionFolder 目录中创建一个名为 MyFormsAppCore 的目录：

```
SolutionFolder
├──MyApps.sln
├──MyFormsApp
│   └──MyForms.csproj
└──MyFormsAppCore      <--- New folder for core project
```

接下来，需要在 MyFormsAppCore 目录中创建 MyFormsCore.csproj 项目。可以使用所选的文本编辑器手动创建此文件。粘贴以下 XML：

```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">

<PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <UseWindowsForms>true</UseWindowsForms>
</PropertyGroup>

</Project>
```

如果不想手动创建项目文件，可以使用 Visual Studio 或 .NET Core SDK 生成项目。但是，必须删除项目模板生成的所有其他文件（项目文件除外）。若要使用 SDK，请从 SolutionFolder 目录运行以下命令：

```
dotnet new winforms -o MyFormsAppCore -n MyFormsCore
```

创建 MyFormsCore.csproj 后，目录结构应如下所示：

```
SolutionFolder
├──MyApps.sln
├──MyFormsApp
│   └──MyForms.csproj
└──MyFormsAppCore
    └──MyFormsCore.csproj
```

使用 Visual Studio 或 SolutionFolder 目录中的 .NET Core CLI 将 MyFormsCore.csproj 项目添加到 MyApps.sln：

```
dotnet sln add .\MyFormsAppCore\MyFormsCore.csproj
```

修复程序集信息生成

使用.NET Framework 创建的 Windows 窗体项目包含一个 `AssemblyInfo.cs` 文件，该文件包含诸如要生成的程序集的版本等程序集特性。SDK 样式的项目会根据 SDK 项目文件自动生成此信息。同时具有两种类型的“程序集信息”时，会产生冲突。通过禁用自动生成可以解决此问题，这会强制项目使用现有的 `AssemblyInfo.cs` 文件。

若要添加到主 `<PropertyGroup>` 节点，有三个设置项。

- **GenerateAssemblyInfo**

将此属性设置为 `false` 时，它不会生成程序集特性。这可以避免与.NET Framework 项目中的现有 `AssemblyInfo.cs` 文件冲突。

- **AssemblyName**

此属性的值是编译时创建的输出二进制。该名称不需要添加扩展名。例如，使用 `MyCoreApp` 生成 `MyCoreApp.exe`。

- **RootNamespace**

项目使用的默认命名空间。它应该与.NET Framework 项目的默认命名空间匹配。

将这三个元素添加到 `MyFormsCore.csproj` 文件中的 `<PropertyGroup>` 节点：

```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">

<PropertyGroup>
  <OutputType>WinExe</OutputType>
  <TargetFramework>netcoreapp3.0</TargetFramework>
  <UseWindowsForms>true</UseWindowsForms>

  <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
  <AssemblyName>MyCoreApp</AssemblyName>
  <RootNamespace>WindowsFormsApp1</RootNamespace>
</PropertyGroup>

</Project>
```

添加源代码

现在，`MyFormsCore.csproj` 项目不编译任何代码。默认情况下，.NET Core 项目会自动包含当前目录和所有子目录中的所有源代码。必须使用相对路径配置项目以包含.NET Framework 项目中的代码。如果.NET Framework 项目使用了 `.resx` 文件作为窗体的图标和资源，则还需要包含这些文件。

将以下 `<ItemGroup>` 节点添加到项目中。每个语句都包含一个文件 glob 模式，其中包含子目录。

```
<ItemGroup>
  <Compile Include=".\\MyFormsApp\\**\\*.cs" />
  <EmbeddedResource Include=".\\MyFormsApp\\**\\*.resx" />
</ItemGroup>
```

或者，可以为.NET Framework 项目中的每个文件创建一个 `<Compile>` 或 `<EmbeddedResource>` 条目。

添加 NuGet 包

将.NET Framework 项目引用的每个 NuGet 包添加到.NET Core 项目。

很可能你的 .NET Framework Windows 窗体应用程序有一个 packages.config 文件，其中包含项目引用的所有 NuGet 包的列表。可以查看此列表以确定要添加到 .NET Core 项目的 NuGet 包。例如，如果 .NET Framework 项目引用了 MetroFramework、MetroFramework.Design 和 MetroFramework.Fonts NuGet 包，则使用 Visual Studio 或 SolutionFolder 目录中的 .NET Core CLI 将每个包添加到项目中：

```
dotnet add .\MyFormsAppCore\MyFormsCore.csproj package MetroFramework
dotnet add .\MyFormsAppCore\MyFormsCore.csproj package MetroFramework.Design
dotnet add .\MyFormsAppCore\MyFormsCore.csproj package MetroFramework.Fonts
```

之前的命令会将以下 NuGet 引用添加到 MyFormsCore.csproj 项目：

```
<ItemGroup>
  <PackageReference Include="MetroFramework" Version="1.2.0.3" />
  <PackageReference Include="MetroFramework.Design" Version="1.2.0.3" />
  <PackageReference Include="MetroFramework.Fonts" Version="1.2.0.3" />
</ItemGroup>
```

移植控件库

如果要移植 Windows 窗体控件库项目，操作说明与移植 .NET Framework Windows 窗体应用程序项目相同，只不过在一些设置上存在差异。并且此操作中不编译到可执行文件，而是编译到库。除了包含源代码的文件 glob 的路径之外，可执行项目和库项目之间的区别也很小。

借助上一步的示例，详细介绍正在处理的项目和文件。

MyApps.sln	解决方案文件的名称。
MyControls.csproj	要移植的 .NET Framework Windows 窗体控件项目的名称。
MyControlsCore.csproj	创建的新 .NET Core 库项目的名称。
MyCoreControls.dll	.NET Core Windows 窗体控件库。

```
SolutionFolder
├── MyApps.sln
├── MyFormsApp
│   └── MyForms.csproj
└── MyFormsAppCore
    └── MyFormsCore.csproj

└── MyFormsControls
    └── MyControls.csproj
└── MyFormsControlsCore
    └── MyControlsCore.csproj    <--- New project for core controls
```

请考虑 MyControlsCore.csproj 项目与先前创建的 MyFormsCore.csproj 项目之间的差异。

```

<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">

  <PropertyGroup>
    - <OutputType>WinExe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <UseWindowsForms>true</UseWindowsForms>

    <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
    - <AssemblyName>MyCoreApp</AssemblyName>
    - <RootNamespace>WindowsFormsApp1</RootNamespace>
    + <AssemblyName>MyControlsCore</AssemblyName>
    + <RootNamespace>WindowsFormsControlLibrary1</RootNamespace>
  </PropertyGroup>

  <ItemGroup>
    - <Compile Include=".\\MyFormsApp\\**\\*.cs" />
    - <EmbeddedResource Include=".\\MyFormsApp\\**\\*.resx" />
    + <Compile Include=".\\MyFormsControls\\**\\*.cs" />
    + <EmbeddedResource Include=".\\MyFormsControls\\**\\*.resx" />
  </ItemGroup>

</Project>

```

以下是 .NET Core Windows 窗体控件库项目文件的示例：

```

<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <UseWindowsForms>true</UseWindowsForms>

    <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
    <AssemblyName>MyCoreControls</AssemblyName>
    <RootNamespace>WindowsFormsControlLibrary1</RootNamespace>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include=".\\MyFormsControls\\**\\*.cs" />
    <EmbeddedResource Include=".\\MyFormsControls\\**\\*.resx" />
  </ItemGroup>

</Project>

```

如你所见，`<OutputType>` 节点已被删除，默认使编译器生成库而不是可执行文件。`<AssemblyName>` 和 `<RootNamespace>` 已更改。具体来说，`<RootNamespace>` 应匹配正在移植的 Windows 窗体控件库的命名空间。最后，`<Compile>` 和 `<EmbeddedResource>` 节点已调整为指向要移植的 Windows 窗体控件库的文件夹。

接下来，在主要的 .NET Core MyFormsCore.csproj 项目中，添加对新 .NET Core Windows 窗体控件库的引用。使用 Visual Studio 或 .NET Core CLI 从 SolutionFolder 目录添加引用：

```
dotnet add .\\MyFormsAppCore\\MyFormsCore.csproj reference .\\MyFormsControlsCore\\MyControlsCore.csproj
```

上一个命令将以下内容添加到 MyFormsCore.csproj 项目中：

```

<ItemGroup>
  <ProjectReference Include=".\\MyFormsControlsCore\\MyControlsCore.csproj" />
</ItemGroup>

```

编译问题

如果在编译项目时遇到问题，可能是由于正在使用的一些仅适用于 Windows 的 API 在 .NET Framework 中可用，但在 .NET Core 中不可用。可以尝试将 [Windows 兼容包](#) NuGet 包添加到项目中。此包仅在 Windows 上运行，为 .NET Core 和 .NET Standard 项目添加了大约 20,000 个 Windows API。

```
dotnet add .\MyFormsAppCore\MyFormsCore.csproj package Microsoft.Windows.Compatibility
```

上一个命令将以下内容添加到 MyFormsCore.csproj 项目中：

```
<ItemGroup>
  <PackageReference Include="Microsoft.Windows.Compatibility" Version="3.1.0" />
</ItemGroup>
```

Windows Forms Designer — Windows 窗体设计器

如本文所述，Visual Studio 2019 仅支持 .NET Framework 项目中的窗体设计器。通过创建并行 .NET Core 项目，可以在使用 .NET Framework 项目设计窗体时通过 .NET Core 测试项目。解决方案文件包括 .NET Framework 和 .NET Core 项目。在 .NET Framework 项目中添加和设计窗体和控件，并且根据添加到 .NET Core 项目的文件 glob 模式，任何新的或更改的文件将自动包含在 .NET Core 项目中。

一旦 Visual Studio 2019 支持 Windows 窗体设计器，就可以将 .NET Core 项目文件的内容复制/粘贴到 .NET Framework 项目文件中。然后删除使用 `<Source>` 和 `<EmbeddedResource>` 项添加的文件 glob 模式。修复由应用程序使用的任何项目引用的路径。这可以有效地将 .NET Framework 项目升级到 .NET Core 项目。

后续步骤

- 了解从 .NET Framework 到 .NET Core 的中断性变更。
- 详细了解 [Windows 兼容包](#)。
- 观看将 .NET Framework Windows 窗体项目移植到 .NET Core 的视频。



如何将 C++/CLI 项目移植到 .NET Core

2020/3/18 • [Edit Online](#)

从 .NET Core 3.1 和 Visual Studio 2019 16.4 版开始, [C++/CLI 项目](#) 可面向 .NET Core。这种支持使你能够将具有 C++/CLI 互操作层的 Windows 桌面应用程序移植到 .NET Core。本文介绍如何将 C++/CLI 项目从 .NET Framework 移植到 .NET Core 3.1。

C++/CLI .NET Core 限制

与其他语言相比, 将 C++/CLI 项目移植到 .NET Core 具有一些重要限制:

- .NET Core 的 C++/CLI 支持仅适用于 Windows。
- C++/CLI 项目不能面向 .NET Standard, 只能面向 .NET Core(或 .NET Framework)。
- C++/CLI 项目不支持新的 SDK 样式项目文件格式。相反, 即使在面向 .NET Core 时, C++/CLI 项目也使用现有的 vcxproj 文件格式。
- C++/CLI 项目不能同时面向多个 .NET 平台。如果需要同时为 .NET Framework 和 .NET Core 生成 C++/CLI 项目, 请使用单独的项目文件。
- .NET Core 不支持 `-clr:pure` 或 `-clr:safe` 编译, 仅支持新的 `-clr:netcore` 选项(等效于 .NET Framework 的 `-clr`)。

移植 C++/CLI 项目

要将 C++/CLI 项目移植到 .NET Core, 请对 vcxproj 文件进行以下更改。这些迁移步骤不同于其他项目类型所需的步骤, 因为 C++/CLI 项目不使用 SDK 样式的项目文件。

1. 将 `<CLRSupport>true</CLRSupport>` 属性替换为 `<CLRSupport>NetCore</CLRSupport>`。此属性通常位于特定于配置的属性组中, 因此你可能需要在多处替换它。
2. 将 `<TargetFrameworkVersion>` 属性替换为 `<TargetFramework>netcoreapp3.1</TargetFramework>`。
3. 删除任何 .NET Framework 引用(例如 `<Reference Include="System" />`)。使用 `<CLRSupport>NetCore</CLRSupport>` 时, 将自动引用 .NET Core SDK 程序集。
4. 根据需要更新 cpp 文件中的 API 使用情况, 以删除 .NET Core 不可使用的 API。由于 C++/CLI 项目通常是非常精简的互操作层, 因此通常不需要进行诸多更改。[.NET 可移植性分析器](#)可用于识别 C++/CLI 二进制文件使用的不受支持的 .NET API, 就像使用纯托管二进制文件一样。[库移植指南](#)中提供了用于确定代码可移植性和更新项目以使用 .NET Core API 的指南。

WPF 和 Windows 窗体使用情况

.NET Core C++/CLI 项目可以使用 Windows 窗体和 WPF API。要使用这些 Windows 桌面 API, 需要将显式框架引用添加到 UI 库。使用 Windows 桌面 API 的 SDK 样式项目通过使用 `Microsoft.NET.Sdk.WindowsDesktop` SDK 来自动引用必要的框架库。由于 C++/CLI 项目不使用 SDK 样式的项目格式, 因此它们在面向 .NET Core 时需要添加显式框架引用。

要使用 Windows 窗体 API, 请将此引用添加到 vcxproj 文件:

```
<!-- Reference all of Windows Forms -->
<FrameworkReference Include="Microsoft.WindowsDesktop.App.WindowsForms" />
```

要使用 WPF API, 请将此引用添加到 vcxproj 文件:

```
<!-- Reference all of WPF -->
<FrameworkReference Include="Microsoft.WindowsDesktop.App.WPF" />
```

要同时使用 Windows 窗体和 WPF API, 请将此引用添加到 vcxproj 文件:

```
<!-- Reference the entirety of the Windows desktop framework:
Windows Forms, WPF, and the types that provide integration between them -->
<FrameworkReference Include="Microsoft.WindowsDesktop.App" />
```

目前, 不能使用 Visual Studio 的引用管理器来添加这些引用。而是改为手动更新项目文件。可通过在 Visual Studio 中卸载项目, 然后编辑项目文件来完成此更新。还可以使用其他编辑器, 例如 VS Code。

在不使用 MSBuild 的情况下生成

还可以在不使用 MSBuild 的情况下生成 C++/CLI 项目。请按照以下步骤, 使用 cl.exe 和 link.exe 直接生成适用于 .NET Core 的 C++/CLI 项目:

1. 编译时, 将 `-clr:netcore` 传递给 cl.exe。
2. 引用必要的 .NET Core 引用程序集。
3. 链接时, 提供 .NET Core 应用主机目录作为 `LibPath` (以便可以找到 ijwhost.lib)。
4. 将 ijwhost.dll(从 .NET Core 应用主机目录)复制到项目的输出目录。
5. 确保将运行托管代码的应用程序的第一个组件存在 `runtimeconfig.json` 文件。如果应用程序具有托管入口点, 则将自动创建并复制 `runtime.config` 文件。不过, 如果应用程序具有本机入口点, 则需要为第一个 C++/CLI 库创建 `runtimeconfig.json` 文件以使用 .NET Core 运行时。

已知问题

在处理面向 .NET Core 的 C++/CLI 项目时, 需要注意一些已知问题。

- .NET Core C++/CLI 项目中的 WPF 框架引用目前会导致一些有关无法导入符号的无关警告。可以安全忽略这些警告, 并且应该尽快解决它们。
- 如果应用程序具有本机入口点, 则首次执行托管代码的 C++/CLI 库需要 `runtimeconfig.json` 文件。此配置文件在 .NET Core 运行时启动时使用。C++/CLI 项目不会在生成时自动创建 `runtimeconfig.json` 文件, 因此必须手动生成该文件。如果从托管入口点调用 C++/CLI 库, 则 C++/CLI 库不需要 `runtimeconfig.json` 文件(因为入口点程序集将具有一个在启动运行时时使用的该文件)。下面显示了一个简单的示例

`runtimeconfig.json` 文件。有关详细信息, 请参阅 [GitHub 上的规范](#)。

```
{
  "runtimeOptions": {
    "tfm": "netcoreapp3.1",
    "framework": {
      "name": "Microsoft.NETCore.App",
      "version": "3.1.0"
    }
  }
}
```

.NET Core 和 .NET Standard 中的单元测试

2020/3/18 • [Edit Online](#)

通过 .NET Core, 可以轻松创建单元测试。本文介绍了单元测试及其与其他类型的测试的不同之处。页面底部附近的链接资源介绍了如何向解决方案添加测试项目。设置测试项目后, 可使用命令行或 Visual Studio 运行单元测试。

如果要测试 ASP.NET Core 项目, 请参阅 [ASP.NET Core 中的集成测试](#)。

.NET Core 2.0 及更高版本支持 [.NET Standard 2.0](#), 我们将使用它的库来演示单元测试。

可以使用适用于 C#、F# 和 Visual Basic 的内置 .NET Core 2.0 及更高版本单元测试项目模板作为个人项目的起始点。

什么是单元测试？

使用自动测试是确保软件应用程序按作者期望执行操作的一种绝佳方式。软件应用程序有多种类型的测试。其中包括集成测试、Web 测试、负载测试和其他测试。“单元测试”用于测试个人软件组件或方法。单元测试仅应测试开发人员控件内的代码。它们不应测试基础结构问题。基础结构问题包括数据库、文件系统和网络资源。

此外, 请记住还可使用编写测试的最佳做法。例如, [测试驱动开发 \(TDD\)](#) 是指先编写单元测试, 再编写该单元测试要检查的代码。TDD 就像先编写书籍大纲, 再编写该书籍。它旨在帮助开发人员编写更简单、更具可读性的高效代码。

NOTE

ASP.NET 团队遵循[这些约定](#)帮助开发人员为测试类和方法提供合适的名称。

编写单元测试时, 尽量不要引入基础结构依赖项。这些依赖项会降低测试速度, 使测试更加脆弱, 应将其保留供集成测试使用。可以通过遵循 [Explicit Dependencies Principle](#)(显式依赖项原则)和使用 [Dependency Injection](#)(依赖项注入)避免应用程序中的这些依赖项。还可以将单元测试保留在单独的项目中, 与集成测试相分隔。这可确保单元测试项目没有引用或依赖于基础结构包。

后续步骤

有关 .NET Core 项目中的单元测试的详细信息:

.NET Core 单元测试项目支持:

- [C#](#)
- [F#](#)
- [Visual Basic](#)

还可以在以下各项之间进行选择:

- [xUnit](#)
- [NUnit](#)
- [MSTest](#)

可在以下演练中了解详细信息:

- [结合使用 .NET Core CLI 与 xUnit 和 C# 创建单元测试。](#)
- [结合使用 .NET Core CLI 与 NUnit 和 C# 创建单元测试。](#)

- [结合使用 .NET Core CLI 与 MSTest 和 C# 创建单元测试。](#)
- [结合使用 .NET Core CLI 与 xUnit 和 F# 创建单元测试。](#)
- [结合使用 .NET Core CLI 与 NUnit 和 F# 创建单元测试。](#)
- [结合使用 .NET Core CLI 与 MSTest 和 F# 创建单元测试。](#)
- [结合使用 .NET Core CLI 与 xUnit 和 Visual Basic 创建单元测试。](#)
- [结合使用 .NET Core CLI 与 NUnit 和 Visual Basic 创建单元测试。](#)
- [结合使用 .NET Core CLI 与 MSTest 和 Visual Basic 创建单元测试。](#)

可在以下文章中了解详细信息：

- Visual Studio Enterprise 提供用于 .NET Core 的卓越测试工具。有关详细信息，请查看 [Live Unit Testing](#) 或 [代码覆盖率](#)。
- 有关如何运行选择性单元测试的详细信息，请参阅[运行选择性单元测试](#)或[使用 Visual Studio 添加和排除测试](#)。
- [如何通过 .NET Core 和 Visual Studio 使用 xUnit。](#)

.NET Core 和 .NET Standard 中的单元测试

2020/3/18 • [Edit Online](#)

通过 .NET Core，可以轻松创建单元测试。本文介绍了单元测试及其与其他类型的测试的不同之处。页面底部附近的链接资源介绍了如何向解决方案添加测试项目。设置测试项目后，可使用命令行或 Visual Studio 运行单元测试。

如果要测试 ASP.NET Core 项目，请参阅 [ASP.NET Core 中的集成测试](#)。

.NET Core 2.0 及更高版本支持 [.NET Standard 2.0](#)，我们将使用它的库来演示单元测试。

可以使用适用于 C#、F# 和 Visual Basic 的内置 .NET Core 2.0 及更高版本单元测试项目模板作为个人项目的起始点。

什么是单元测试？

使用自动测试是确保软件应用程序按作者期望执行操作的一种绝佳方式。软件应用程序有多种类型的测试。其中包括集成测试、Web 测试、负载测试和其他测试。“单元测试”用于测试个人软件组件或方法。单元测试仅应测试开发人员控件内的代码。它们不应测试基础结构问题。基础结构问题包括数据库、文件系统和网络资源。

此外，请记住还可使用编写测试的最佳做法。例如，[测试驱动开发 \(TDD\)](#) 是指先编写单元测试，再编写该单元测试要检查的代码。TDD 就像先编写书籍大纲，再编写该书籍。它旨在帮助开发人员编写更简单、更具可读性的高效代码。

NOTE

ASP.NET 团队遵循[这些约定](#)帮助开发人员为测试类和方法提供合适的名称。

编写单元测试时，尽量不要引入基础结构依赖项。这些依赖项会降低测试速度，使测试更加脆弱，应将其保留供集成测试使用。可以通过遵循 [Explicit Dependencies Principle](#)(显式依赖项原则)和使用 [Dependency Injection](#)(依赖项注入)避免应用程序中的这些依赖项。还可以将单元测试保留在单独的项目中，与集成测试相分隔。这可确保单元测试项目没有引用或依赖于基础结构包。

后续步骤

有关 .NET Core 项目中的单元测试的详细信息：

.NET Core 单元测试项目支持：

- [C#](#)
- [F#](#)
- [Visual Basic](#)

还可以在以下各项之间进行选择：

- [xUnit](#)
- [NUnit](#)
- [MSTest](#)

可在以下演练中了解详细信息：

- [结合使用 .NET Core CLI 与 xUnit 和 C# 创建单元测试。](#)
- [结合使用 .NET Core CLI 与 NUnit 和 C# 创建单元测试。](#)

- [结合使用 .NET Core CLI 与 MSTest 和 C# 创建单元测试。](#)
- [结合使用 .NET Core CLI 与 xUnit 和 F# 创建单元测试。](#)
- [结合使用 .NET Core CLI 与 NUnit 和 F# 创建单元测试。](#)
- [结合使用 .NET Core CLI 与 MSTest 和 F# 创建单元测试。](#)
- [结合使用 .NET Core CLI 与 xUnit 和 Visual Basic 创建单元测试。](#)
- [结合使用 .NET Core CLI 与 NUnit 和 Visual Basic 创建单元测试。](#)
- [结合使用 .NET Core CLI 与 MSTest 和 Visual Basic 创建单元测试。](#)

可在以下文章中了解详细信息：

- Visual Studio Enterprise 提供用于 .NET Core 的卓越测试工具。有关详细信息，请查看 [Live Unit Testing](#) 或 [代码覆盖率](#)。
- 有关如何运行选择性单元测试的详细信息，请参阅[运行选择性单元测试](#)或[使用 Visual Studio 添加和排除测试](#)。
- [如何通过 .NET Core 和 Visual Studio 使用 xUnit。](#)

.NET Core 和 .NET Standard 单元测试最佳做法

2020/3/18 • [Edit Online](#)

编写单元测试有许多好处；它们有助于回归、提供文档和促进良好的设计。然而，难懂且脆弱的单元测试会对代码库造成严重破坏。本文介绍一些有关 .NET Core 和 .NET Standard 项目的单元测试设计的最佳做法。

本指南将介绍一些在编写单元测试时的最佳做法，使测试具有弹性且易于理解。

作者是 [John Reese](#) 且特别感谢 [Roy Osherove](#)

为什么要执行单元测试？

比执行功能测试节省时间

功能测试费用高。它们通常涉及打开应用程序并执行一系列你（或其他人）必须遵循的步骤，以验证预期的行为。测试人员可能并不总是了解这些步骤，这意味着为了执行测试，他们必须联系更熟悉该领域的人。对于细微更改，测试本身可能需要几秒钟，对于较大更改，可能需要几分钟。最后，在系统中所做的每项更改都必须重复此过程。

而单元测试只需按一下按钮即可运行，只需要几毫秒时间，且无需测试人员了解整个系统。测试通过与否取决于测试运行程序，而非测试人员。

防止回归

回归缺陷是在对应用程序进行更改时引入的缺陷。测试人员通常不仅测试新功能，还要测试预先存在的功能，以验证先前实现的功能是否仍按预期运行。

使用单元测试，可在每次生成后，甚至在更改一行代码后重新运行整套测试。让你确信新代码不会破坏现有功能。

可执行文档

在给定某个输入的情况下，特定方法的作用或行为可能并不总是很明显。你可能会问自己：如果我将空白字符串传递给它，此方法会有怎样的行为？Null？

如果你有一套命名正确的单元测试，每个测试应能够清楚地解释给定输入的预期输出。此外，它应该能够验证其确实有效。

减少耦合代码

当代码紧密耦合时，可能难以进行单元测试。如果不为编写的代码创建单元测试，则耦合可能不太明显。

为代码编写测试会自然地解耦代码，因为采用其他方法测试会更困难。

优质单元测试的特征

- **快速**。对成熟项目进行数千次单元测试，这很常见。应花非常少的时间来运行单元测试。几毫秒。
- **独立**。单元测试是独立的，可以单独运行，并且不依赖文件系统或数据库等任何外部因素。
- **可重复**。运行单元测试的结果应该保持一致，也就是说，如果在运行期间不更改任何内容，总是返回相同的结果。
- **自检查**。测试应该能够在没有任何人工交互的情况下，自动检测测试是否通过。
- **及时**。与要测试的代码相比，编写单元测试不应花费过多不必要的空间。如果发现测试代码与编写代码相比需要花费大量的时间，请考虑一种更易测试的设计。

让我们使用相同的术语

遗憾的是，当谈到测试时，术语“mock”的滥用情况很严重。以下定义了编写单元测试时最常见的 fake 类型：

Fake - Fake 是一个通用术语，可用于描述 stub 或 mock 对象。它是 stub 还是 mock 取决于使用它的上下文。也就是说，Fake 可以是 stub 或 mock。

Mock - Mock 对象是系统中的 fake 对象，用于确定单元测试是否通过。Mock 起初为 Fake，直到对其进行断言。

Stub - Stub 是系统中现有依赖项（或协作者）的可控制替代项。通过使用 Stub，可以在无需使用依赖项的情况下直接测试代码。默认情况下，fake 起初为 Stub。

请思考以下代码片段：

```
var mockOrder = new MockOrder();
var purchase = new Purchase(mockOrder);

purchase.ValidateOrders();

Assert.True(purchase.CanBeShipped);
```

这是 Stub 被引用为 mock 的一个示例。在本例中，它是 Stub。只是将 Order 作为实例化 Purchase（被测系统）的一种方法传递。名称 `MockOrder` 也非常具有误导性，因为同样地 order 不是 mock。

更好的方法是

```
var stubOrder = new FakeOrder();
var purchase = new Purchase(stubOrder);

purchase.ValidateOrders();

Assert.True(purchase.CanBeShipped);
```

通过将类重命名为 `FakeOrder`，使类更通用，类可以用作 mock 或 stub。以更适合测试用例者为准。在上述示例中，`FakeOrder` 用作 stub。在断言期间，没有以任何形状或形式使用 `FakeOrder`。`FakeOrder` 只传递到 `Purchase` 类，以满足构造函数的要求。

要将其用作 Mock，可执行如下操作

```
var mockOrder = new FakeOrder();
var purchase = new Purchase(mockOrder);

purchase.ValidateOrders();

Assert.True(mockOrder.Validated);
```

在这种情况下，检查 Fake 上的属性（对其进行断言），因此在以上代码片段中，`mockOrder` 是 Mock。

IMPORTANT

正确理解此术语至关重要。如果将 stub 称为 mock，其他开发人员会对你的意图做出错误的判断。

关于 mock 与 stub 需要注意的一个重点是，mock 与 stub 很像，但可以针对 mock 对象进行断言，而不针对 stub 进行断言。

最佳做法

为测试命名

测试的名称应包括三个部分：

- 要测试的方法的名称。

- 测试的方案。
- 调用方案时的预期行为。

为什么？

- 命名标准非常重要，因为它们明确地表达了测试的意图。

测试不仅能确保代码有效，还能提供文档。只需查看单元测试套件，就可以在不查看代码本身的情况下推断代码的行为。此外，测试失败时，可以确切地看到哪些方案不符合预期。

不佳：

```
[Fact]
public void Test_Single()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

良好：

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

安排测试

“Arrange、Act、Assert”是单元测试时的常见模式。顾名思义，它包含三个主要操作：

- 安排对象，根据需要对其进行创建和设置。
- 作用于对象。
- 断言某些项按预期进行。

为什么？

- 明确地将要测试的内容从“arrange”和“assert”步骤分开。
- 降低将断言与“Act”代码混杂的可能性。

可读性是编写测试时最重要的方面之一。在测试中分离这些操作会明确地突出显示调用代码所需的依赖项、调用代码的方式以及尝试断言的内容。虽然可以组合一些步骤并减小测试的大小，但主要目标是使测试尽可能可读。

不佳：

```
[Fact]
public void Add_EmptyString_ReturnsZero()
{
    // Arrange
    var stringCalculator = new StringCalculator();

    // Assert
    Assert.Equal(0, stringCalculator.Add(""));
}
```

良好：

```
[Fact]
public void Add_EmptyString_ReturnsZero()
{
    // Arrange
    var stringCalculator = new StringCalculator();

    // Act
    var actual = stringCalculator.Add("");

    // Assert
    Assert.Equal(0, actual);
}
```

以最精简方式编写通过测试

单元测试中使用的输入应为最简单的输入，以便验证当前正在测试的行为。

为什么？

- 测试对代码库的未来更改更具弹性。
- 更接近于测试行为而非实现。

包含比通过测试所需信息更多信息的测试更可能将错误引入测试，并且可能使测试的意图变得不太明确。编写测试时需要将重点放在行为上。在模型上设置额外的属性或在不需要时使用非零值，只会偏离所要证明的内容。

不佳：

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("42");

    Assert.Equal(42, actual);
}
```

良好：

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

避免魔幻字符串

单元测试中的命名变量和生产代码中的命名变量同样重要。单元测试不应包含魔幻字符串。

为什么？

- 测试读者无需检查生产代码即可了解值的特殊之处。
- 明确地显示所要证明的内容，而不是显示要完成的内容。

魔幻字符串可能会让测试读者感到困惑。如果字符串看起来不寻常，他们可能想知道为什么为某个参数或返回值选择了某个值。这可能会使他们仔细查看实现细节，而不是专注于测试。

TIP

编写测试时，应力求表达尽可能多的意图。对于魔幻字符串，一种很好的方法是将这些值赋给常量。

不佳：

```
[Fact]
public void Add_BigNumber_ThrowsException()
{
    var stringCalculator = new StringCalculator();

    Action actual = () => stringCalculator.Add("1001");

    Assert.Throws<OverflowException>(actual);
}
```

良好：

```
[Fact]
void Add_MaximumSumResult_ThrowsOverflowException()
{
    var stringCalculator = new StringCalculator();
    const string MAXIMUM_RESULT = "1001";

    Action actual = () => stringCalculator.Add(MAXIMUM_RESULT);

    Assert.Throws<OverflowException>(actual);
}
```

在测试中应避免逻辑

编写单元测试时，请避免手动字符串串联和逻辑条件，例如 `if`、`while`、`for` 和 `switch` 等等。

为什么？

- 降低在测试中引入 bug 的可能性。
- 专注于最终结果，而不是实现细节。

将逻辑引入测试套件中时，引入 bug 的可能性大幅度增加。你最不希望测试套件中出现 bug。你应该对测试工作充满高度的自信，否则你将不会信任它们。你不信任的测试不会带来任何价值。当测试失败时，你希望意识到代码存在问题且无法忽略该问题。

TIP

如果不可避免地要在测试中使用逻辑，请考虑将测试分成两个或多个不同的测试。

不佳：

```
[Fact]
public void Add_MultipleNumbers_ReturnsCorrectResults()
{
    var stringCalculator = new StringCalculator();
    var expected = 0;
    var testCases = new[]
    {
        "0,0,0",
        "0,1,2",
        "1,2,3"
    };

    foreach (var test in testCases)
    {
        Assert.Equal(expected, stringCalculator.Add(test));
        expected += 3;
    }
}
```

良好:

```
[Theory]
[InlineData("0,0,0", 0)]
[InlineData("0,1,2", 3)]
[InlineData("1,2,3", 6)]
public void Add_MultipleNumbers_ReturnsSumOfNumbers(string input, int expected)
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add(input);

    Assert.Equal(expected, actual);
}
```

更偏好 helper 方法而非 setup 和 teardown

如果测试需要类似的对象或状态, 那么比起使用 Setup 和 Teardown 属性(如果存在), 更偏好使用 helper 方法。

为什么?

- 读者阅读测试时产生的困惑减少, 因为每个测试中都可以看到所有代码。
- 给定测试的设置过多或过少的可能性降低。
- 在测试之间共享状态(这会在测试之间创建不需要的依赖项)的可能性降低。

在单元测试框架中, 在测试套件的每个单元测试之前调用 `Setup`。虽然有些人可能会将其视为有用的工具, 但它通常最终导致庞大且难懂的测试。每个测试通常有不同的要求, 以使测试启动并运行。遗憾的是, `Setup` 迫使你对每个测试使用完全相同的要求。

NOTE

自版本 2.x 起, xUnit 已删除 `SetUp` 和 `TearDown`

不佳:

```
private readonly StringCalculator stringCalculator;
public StringCalculatorTests()
{
    stringCalculator = new StringCalculator();
}
```

```
// more tests...
```

```
[Fact]
public void Add_TwoNumbers_ReturnsSumOfNumbers()
{
    var result = stringCalculator.Add("0,1");

    Assert.Equal(1, result);
}
```

良好：

```
[Fact]
public void Add_TwoNumbers_ReturnsSumOfNumbers()
{
    var stringCalculator = CreateDefaultStringCalculator();

    var actual = stringCalculator.Add("0,1");

    Assert.Equal(1, actual);
}
```

```
// more tests...
```

```
private StringCalculator CreateDefaultStringCalculator()
{
    return new StringCalculator();
}
```

避免多个断言

在编写测试时，请尝试每次测试只包含一个 Assert。仅使用一个 assert 的常用方法包括：

- 为每个 assert 创建单独的测试。
- 使用参数化测试。

为什么？

- 如果一个 Assert 失败，将不计算后续 Assert。
- 确保在测试中没有断言多个事例。
- 让你从整体上了解测试失败原因。

将多个断言引入测试用例时，不能保证所有断言都会执行。在大多数单元测试框架中，一旦断言在单元测试中失败，则进行中的测试会自动被视为失败。这可能会令人困惑，因为正在运行的功能将显示为失败。

NOTE

此规则的一个常见例外是对对象进行断言。在这种情况下，通常可以对每个属性进行多次断言，以确保对象处于所预期的状态。

不佳：

```
[Fact]
public void Add_EdgeCases_ThrowsArgumentExceptions()
{
    Assert.Throws<ArgumentException>(() => stringCalculator.Add(null));
    Assert.Throws<ArgumentException>(() => stringCalculator.Add("a"));
}
```

良好：

```
[Theory]
[InlineData(null)]
[InlineData("a")]
public void Add_InputNullOrEmptyAlphabetic_ThrowsArgumentException(string input)
{
    var stringCalculator = new StringCalculator();

    Action actual = () => stringCalculator.Add(input);

    Assert.Throws<ArgumentException>(actual);
}
```

通过单元测试公共方法验证专有方法

在大多数情况下，不需要测试专用方法。专用方法是实现细节。可以这样认为：专用方法永远不会孤立存在。在某些时候，存在调用专用方法作为其实现的一部分的面向公共的方法。你应关心的是调用到专用方法的公共方法的最终结果。

请考虑下列情形

```
public string ParseLogLine(string input)
{
    var sanitizedInput = TrimInput(input);
    return sanitizedInput;
}

private string TrimInput(string input)
{
    return input.Trim();
}
```

你的第一反应可能是开始为 `TrimInput` 编写测试，因为想要确保该方法按预期工作。但是，`ParseLogLine` 完全有可能以一种你所不期望的方式操纵 `sanitizedInput`，使得对 `TrimInput` 的测试变得毫无用处。

真正的测试应该针对面向公共的方法 `ParseLogLine` 进行，因为这是你最终应该关心的。

```
public void ParseLogLine_ByDefault_ReturnsTrimmedResult()
{
    var parser = new Parser();

    var result = parser.ParseLogLine(" a ");

    Assert.Equals("a", result);
}
```

由此，如果看到一个专用方法，可以找到公共方法并针对该方法编写测试。不能仅仅因为专用方法返回预期结果就认为最终调用专用方法的系统正确地使用结果。

Stub 静态引用

单元测试的原则之一是其必须完全控制被测试的系统。当生产代码包含对静态引用（例如 `DateTime.Now`）的调用

时，这可能会存在问题。考虑下列代码

```
public int GetDiscountedPrice(int price)
{
    if(DateTime.Now.DayOfWeek == DayOfWeek.Tuesday)
    {
        return price / 2;
    }
    else
    {
        return price;
    }
}
```

如何对此代码进行单元测试？可以尝试一种方法，例如

```
public void GetDiscountedPrice_ByDefault_ReturnsFullPrice()
{
    var priceCalculator = new PriceCalculator();

    var actual = priceCalculator.GetDiscountedPrice(2);

    Assert.Equals(2, actual)
}

public void GetDiscountedPrice_OnTuesday_ReturnsHalfPrice()
{
    var priceCalculator = new PriceCalculator();

    var actual = priceCalculator.GetDiscountedPrice(2);

    Assert.Equals(1, actual);
}
```

遗憾的是，你会很快意识到你的测试存在一些问题。

- 如果在星期二运行测试套件，则第二个测试将通过，但第一个测试将失败。
- 如果在任何其他日期运行测试套件，则第一个测试将通过，但第二个测试将失败。

要解决这些问题，需要将“seam”引入生产代码中。一种方法是在接口中包装需要控制的代码，并使生产代码依赖于该接口。

```
public interface IDateTimeProvider
{
    DayOfWeek DayOfWeek();
}

public int GetDiscountedPrice(int price, IDateTimeProvider dateTimeProvider)
{
    if(dateTimeProvider.DayOfWeek() == DayOfWeek.Tuesday)
    {
        return price / 2;
    }
    else
    {
        return price;
    }
}
```

你的测试套件现在变得

```
public void GetDiscountedPrice_ByDefault_ReturnsFullPrice()
{
    var priceCalculator = new PriceCalculator();
    var dateTimeProviderStub = new Mock<IDateTimeProvider>();
    dateTimeProviderStub.Setup(dtp => dtp.DayOfWeek()).Returns(DayOfWeek.Monday);

    var actual = priceCalculator.GetDiscountedPrice(2, dateTimeProviderStub);

    Assert.Equals(2, actual);
}

public void GetDiscountedPrice_OnTuesday_ReturnsHalfPrice()
{
    var priceCalculator = new PriceCalculator();
    var dateTimeProviderStub = new Mock<IDateTimeProvider>();
    dateTimeProviderStub.Setup(dtp => dtp.DayOfWeek()).Returns(DayOfWeek.Tuesday);

    var actual = priceCalculator.GetDiscountedPrice(2, dateTimeProviderStub);

    Assert.Equals(1, actual);
}
```

现在，测试套件可以完全控制 `DateTime.Now`，并且在调用方法时可以存根任何值。

使用 dotnet test 和 xUnit 在 .NET Core 中进行 C# 单元测试

2020/3/18 • [Edit Online](#)

本教程演示如何生成包含单元测试项目和源代码项目的解决方案。若要使用预构建解决方案学习本教程，请[查看或下载示例代码](#)。有关下载说明，请参阅[示例和教程](#)。

创建解决方案

在本部分中，将创建包含源和测试项目的解决方案。已完成的解决方案具有以下目录结构：

```
/unit-testing-using-dotnet-test
  unit-testing-using-dotnet-test.sln
  /PrimeService
    PrimeService.cs
    PrimeService.csproj
  /PrimeService.Tests
    PrimeService_IsPrimeShould.cs
    PrimeServiceTests.csproj
```

以下说明提供了创建测试解决方案的步骤。有关通过一个步骤创建测试解决方案的说明，请参阅[用于创建测试解决方案的命令](#)。

- 打开 shell 窗口。
- 运行下面的命令：

```
dotnet new sln -o unit-testing-using-dotnet-test
```

`dotnet new sln` 命令用于在 unit-testing-using-dotnet-test 目录中创建新的解决方案。

- 将目录更改为 unit-testing-using-dotnet-test 文件夹。
- 运行下面的命令：

```
dotnet new classlib -o PrimeService
```

`dotnet new classlib` 命令用于在 PrimeService 文件夹中创建新的类库项目。新的类库将包含要测试的代码。

- 将 *Class1.cs* 重命名为 *PrimeService.cs*。
- 将 PrimeService.cs 中的代码替换为以下代码：

```
using System;

namespace Prime.Services
{
    public class PrimeService
    {
        public bool IsPrime(int candidate)
        {
            throw new NotImplementedException("Not implemented.");
        }
    }
}
```

- 前面的代码：
 - 引发 `NotImplementedException`, 其中包含一条消息, 指示未实现。
 - 稍后在教程中更新。
- 在 unit-testing-using-dotnet-test 目录下运行以下命令, 向解决方案添加类库项目：

```
dotnet sln add ./PrimeService/PrimeService.csproj
```

- 运行以下命令创建 PrimeService.Tests 项目：

```
dotnet new xunit -o PrimeService.Tests
```

- 上面的命令：
 - 在 PrimeService.Tests 目录中创建 PrimeService.Tests 项目。测试项目将 `xUnit` 用作测试库。
 - 通过将以下 `<PackageReference />` 元素添加到项目文件来配置测试运行程序：
 - “Microsoft.NET.Test.Sdk”
 - “xunit”
 - “xunit.runner.visualstudio”
- 运行以下命令将测试项目添加到解决方案文件：

```
dotnet sln add ./PrimeService.Tests/PrimeService.Tests.csproj
```

- 将 `PrimeService` 类库作为一个依赖项添加到 PrimeService.Tests 项目中：

```
dotnet add ./PrimeService.Tests/PrimeService.Tests.csproj reference ./PrimeService/PrimeService.csproj
```

用于创建解决方案的命令

本部分汇总了上一部分中的所有命令。如果已完成上一部分中的步骤, 请跳过本部分。

以下命令用于在 Windows 计算机上创建测试解决方案。对于 macOS 和 Unix, 请将 `ren` 命令更新为 OS 版本的 `mv` 以重命名文件：

```
dotnet new sln -o unit-testing-using-dotnet-test
cd unit-testing-using-dotnet-test
dotnet new classlib -o PrimeService
ren .\PrimeService\Class1.cs PrimeService.cs
dotnet sln add ./PrimeService/PrimeService.csproj
dotnet new xunit -o PrimeService.Tests
dotnet add ./PrimeService.Tests/PrimeService.Tests.csproj reference ./PrimeService/PrimeService.csproj
dotnet sln add ./PrimeService.Tests/PrimeService.Tests.csproj
```

请按照上一部分中的“将 PrimeService.cs 中的代码替换为以下代码”的说明进行操作。

创建测试

测试驱动开发 (TDD) 中的一种常用方法是在实现目标代码之前编写测试。本教程使用 TDD 方法。`IsPrime` 方法可调用，但未实现。对 `IsPrime` 的测试调用失败。对于 TDD，会编写已知失败的测试。更新目标代码使测试通过。你可以重复使用此方法，编写失败的测试，然后更新目标代码使测试通过。

更新 PrimeService.Tests 项目：

- 删除 PrimeService.Tests/UnitTest1.cs。
- 创建 PrimeService.Tests/PrimeService_IsPrimeShould.cs 文件。
- 将 PrimeService_IsPrimeShould.cs 中的代码替换为以下代码：

```
using Xunit;
using Prime.Services;

namespace Prime.UnitTests.Services
{
    public class PrimeService_IsPrimeShould
    {
        private readonly PrimeService _primeService;

        public PrimeService_IsPrimeShould()
        {
            _primeService = new PrimeService();
        }

        [Fact]
        public void IsPrime_InputIs1_ReturnFalse()
        {
            var result = _primeService.IsPrime(1);

            Assert.False(result, "1 should not be prime");
        }
    }
}
```

[Fact] 属性声明由测试运行程序运行的测试方法。从 PrimeService.Tests 文件夹运行 `dotnet test`。`dotnet test` 命令生成两个项目并运行测试。xUnit 测试运行程序包含要运行测试的程序入口点。`dotnet test` 使用单元测试项目启动测试运行程序。

测试失败，因为尚未实现 `IsPrime`。使用 TDD 方法，只需编写足够的代码即可使此测试通过。使用以下代码更新 `IsPrime`：

```
public bool IsPrime(int candidate)
{
    if (candidate == 1)
    {
        return false;
    }
    throw new NotImplementedException("Not fully implemented.");
}
```

运行 `dotnet test`。测试通过。

添加更多测试

为 0 和 -1 添加素数测试。你可以复制上述测试并将以下代码更改为使用 0 和 -1：

```
var result = _primeService.IsPrime(1);

Assert.False(result, "1 should not be prime");
```

仅当参数更改代码重复和测试膨胀中的结果时复制测试代码。以下 xUnit 属性允许编写类似测试套件：

- `[Theory]` 表示执行相同代码，但具有不同输入参数的测试套件。
- `[InlineData]` 属性指定这些输入的值。

可以不使用上述 xUnit 属性创建新测试，而是用来创建单个索引。替换以下代码：

```
[Fact]
public void IsPrime_InputIs1_ReturnFalse()
{
    var result = _primeService.IsPrime(1);

    Assert.False(result, "1 should not be prime");
}
```

替换为以下代码：

```
[Theory]
[InlineData(-1)]
[InlineData(0)]
[InlineData(1)]
public void IsPrime_ValuesLessThan2_ReturnFalse(int value)
{
    var result = _primeService.IsPrime(value);

    Assert.False(result, $"{value} should not be prime");
}
```

在前面的代码中，`[Theory]` 和 `[InlineData]` 允许测试多个小于 2 的值。2 是最小的素数。

运行 `dotnet test`，其中两个测试失败。若要使所有测试通过，请使用以下代码更新 `IsPrime` 方法：

```
public bool IsPrime(int candidate)
{
    if (candidate < 2)
    {
        return false;
    }
    throw new NotImplementedException("Not fully implemented.");
}
```

遵循 TDD 方法，添加更多失败的测试，然后更新目标代码。请参阅[已完成的测试版本](#)和[库的完整实现](#)。

已完成的 `IsPrime` 方法不是用于测试素性的有效算法。

其他资源

- [xUnit.net 官方网站](#)
- [ASP.NET Core 中的测试控制器逻辑](#)
- [dotnet add reference](#)

使用 NUnit 和 .NET Core 进行 C# 单元测试

2020/3/18 • [Edit Online](#)

本教程介绍分步构建示例解决方案的交互式体验，以了解单元测试概念。如果希望使用预构建解决方案学习本教程，请在开始前[查看或下载示例代码](#)。有关下载说明，请参阅[示例和教程](#)。

本文介绍如何测试 .NET Core 项目。如果要测试 ASP.NET Core 项目，请参阅[ASP.NET Core 中的集成测试](#)。

系统必备

- [.NET Core 2.1 SDK 或更高版本。](#)
- 按需选择的文本编辑器或代码编辑器。

创建源项目

打开 shell 窗口。创建一个名为 unit-testing-using-nunit 的目录，以保留该解决方案。在此新目录中，运行以下命令，为类库和测试项目创建新的解决方案文件：

```
dotnet new sln
```

接下来，创建 PrimeService 目录。下图显示了当前的目录和文件结构：

```
/unit-testing-using-nunit
  unit-testing-using-nunit.sln
  /PrimeService
```

将 PrimeService 作为当前目录，并运行以下命令以创建源项目：

```
dotnet new classlib
```

将 Class1.cs 重命名为 PrimeService.cs。创建 `PrimeService` 类的失败实现：

```
using System;

namespace Prime.Services
{
    public class PrimeService
    {
        public bool IsPrime(int candidate)
        {
            throw new NotImplementedException("Please create a test first.");
        }
    }
}
```

将目录更改回 unit-testing-using-nunit 目录。运行以下命令，向解决方案添加类库项目：

```
dotnet sln add PrimeService/PrimeService.csproj
```

创建测试项目

接下来，创建 PrimeService.Tests 目录。下图显示了它的目录结构：

```
/unit-testing-using-nunit
  unit-testing-using-nunit.sln
  /PrimeService
    Source Files
    PrimeService.csproj
  /PrimeService.Tests
```

将 PrimeService.Tests 目录作为当前目录，并使用以下命令创建一个新项目：

```
dotnet new nunit
```

`dotnet new` 命令可创建一个将 NUnit 用作测试库的测试项目。生成的模板在 PrimeService.Tests.csproj 文件中配置测试运行程序：

```
<ItemGroup>
  <PackageReference Include="nunit" Version="3.12.0" />
  <PackageReference Include="NUnit3TestAdapter" Version="3.16.0" />
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.4.0" />
</ItemGroup>
```

测试项目需要其他包创建和运行单元测试。在上一步中，`dotnet new` 已添加 Microsoft 测试 SDK、NUnit 测试框架和 NUnit 测试适配器。现在，将 `PrimeService` 类库作为另一个依赖项添加到项目中。使用 `dotnet add reference` 命令：

```
dotnet add reference ../PrimeService/PrimeService.csproj
```

可以在 GitHub 上的[示例存储库](#)中看到整个文件。

下图显示了最终的解决方案布局：

```
/unit-testing-using-nunit
  unit-testing-using-nunit.sln
  /PrimeService
    Source Files
    PrimeService.csproj
  /PrimeService.Tests
    Test Source Files
    PrimeService.Tests.csproj
```

在 unit-testing-using-nunit 目录中执行以下命令：

```
dotnet sln add ./PrimeService.Tests/PrimeService.Tests.csproj
```

创建第一个测试

编写一个失败测试，使其通过，然后重复此过程。在 PrimeService.Tests 目录中，将 `UnitTest1.cs` 文件重命名为 `PrimeService_IsPrimeShould.cs`，并将其整个内容替换为以下代码：

```

using NUnit.Framework;
using Prime.Services;

namespace Prime.UnitTests.Services
{
    [TestFixture]
    public class PrimeService_IsPrimeShould
    {
        private PrimeService _primeService;

        [SetUp]
        public void SetUp()
        {
            _primeService = new PrimeService();
        }

        [Test]
        public void IsPrime_InputIs1_ReturnFalse()
        {
            var result = _primeService.IsPrime(1);

            Assert.IsFalse(result, "1 should not be prime");
        }
    }
}

```

[TestFixture] 属性表示包含单元测试的类。[Test] 属性指示方法是测试方法。

保存此文件并执行 `dotnet test` 以构建测试和类库，然后运行测试。NUnit 测试运行程序包含要运行测试的程序入口点。`dotnet test` 使用已创建的单元测试项目启动测试运行程序。

测试失败。尚未创建实现。在起作用的 `PrimeService` 类中编写最简单的代码，使此测试通过：

```

public bool IsPrime(int candidate)
{
    if (candidate == 1)
    {
        return false;
    }
    throw new NotImplementedException("Please create a test first.");
}

```

在 `unit-testing-using-nunit` 目录中再次运行 `dotnet test`。`dotnet test` 命令构建 `PrimeService` 项目，然后构建 `PrimeService.Tests` 项目。构建这两个项目后，该命令将运行此单项测试。测试通过。

添加更多功能

你已经通过了一个测试，现在可以编写更多测试。质数有其他几种简单情况：0, -1。可以添加具有 [Test] 属性的新测试，但这很快就会变得枯燥乏味。还有其他 NUnit 属性可用于编写一套类似的测试。`[TestCase]` 属性用于创建一套可执行相同代码但具有不同输入参数的测试。可以使用 `[TestCase]` 属性来指定这些输入的值。

无需创建新的测试，而是应用此属性来创建数据驱动的单个测试。数据驱动的测试方法用于测试多个小于 2 (即最小质数) 的值：

```
[TestCase(-1)]
[TestCase(0)]
[TestCase(1)]
public void IsPrime_ValuesLessThan2_ReturnFalse(int value)
{
    var result = _primeService.IsPrime(value);

    Assert.IsFalse(result, $"{value} should not be prime");
}
```

运行 `dotnet test`，两项测试均失败。若要使所有测试通过，可以在 `PrimeService.cs` 文件中更改 `if` 方法开头的 `Main` 子句：

```
if (candidate < 2)
```

通过在主库中添加更多测试、理论和代码继续循环访问。你将拥有[已完成的测试版本](#)和[库的完整实现](#)。

你已生成一个小型库和该库的一组单元测试。你已将解决方案结构化，使添加新包和新测试成为了正常工作流的一部分。你已将多数的时间和精力集中在解决应用程序的目标上。

使用 MSTest 和 .NET Core 进行 C# 单元测试

2020/3/18 • [Edit Online](#)

本教程介绍分步构建示例解决方案的交互式体验，以了解单元测试概念。如果希望使用预构建解决方案学习本教程，请在开始前[查看或下载示例代码](#)。有关下载说明，请参阅[示例和教程](#)。

本文介绍如何测试 .NET Core 项目。如果要测试 ASP.NET Core 项目，请参阅[ASP.NET Core 中的集成测试](#)。

创建源项目

打开 shell 窗口。创建一个名为 unit-testing-using-mstest 的目录，用以保存解决方案。在此新目录中，运行 `dotnet new sln` 为类库和测试项目创建新的解决方案文件。接下来，创建 PrimeService 目录。下图显示了当前的目录和文件结构：

```
/unit-testing-using-mstest
    unit-testing-using-mstest.sln
    /PrimeService
```

将 `PrimeService` 作为当前目录，然后运行 `dotnet new classlib` 以创建源项目。将 `Class1.cs` 重命名为 `PrimeService.cs`。创建 `PrimeService` 类的失败实现：

```
using System;

namespace Prime.Services
{
    public class PrimeService
    {
        public bool IsPrime(int candidate)
        {
            throw new NotImplementedException("Please create a test first.");
        }
    }
}
```

将目录更改回 `unit-testing-using-mstest` 目录。运行 `dotnet sln add PrimeService/PrimeService.csproj` 向解决方案添加类库项目。

创建测试项目

接下来，创建 `PrimeService.Tests` 目录。下图显示了它的目录结构：

```
/unit-testing-using-mstest
    unit-testing-using-mstest.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
```

将 `PrimeService.Tests` 目录作为当前目录，并使用 `dotnet new mstest` 创建一个新项目。dotnet 新命令会创建一个将 MSTest 用作测试库的测试项目。生成的模板在 `PrimeServiceTests.csproj` 文件中配置测试运行程序：

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.3.0" />
  <PackageReference Include="MSTest.TestAdapter" Version="1.1.18" />
  <PackageReference Include="MSTest.TestFramework" Version="1.1.18" />
</ItemGroup>
```

测试项目需要其他包创建和运行单元测试。上一步中的 `dotnet new` 添加了 MSTest SDK、MSTest 测试框架和 MSTest 运行程序。现在，将 `PrimeService` 类库作为另一个依赖项添加到项目中。使用 `dotnet add reference` 命令：

```
dotnet add reference ../PrimeService/PrimeService.csproj
```

可以在 GitHub 上的[示例存储库](#)中看到整个文件。

下图显示了最终的解决方案布局：

```
/unit-testing-using-mstest
  unit-testing-using-mstest.sln
  /PrimeService
    Source Files
    PrimeService.csproj
  /PrimeService.Tests
    Test Source Files
    PrimeServiceTests.csproj
```

在 `unit-testing-using-mstest` `dotnet sln add .\PrimeService.Tests\PrimeService.Tests.csproj` 目录中执行。

创建第一个测试

编写一个失败测试，使其通过，然后重复此过程。从 `PrimeService.Tests` 目录删除 `UnitTest1.cs`，并创建一个名为 `PrimeService_IsPrimeShould.cs` 且包含以下内容的新 C# 文件：

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Prime.Services;

namespace Prime.UnitTests.Services
{
    [TestClass]
    public class PrimeService_IsPrimeShould
    {
        private readonly PrimeService _primeService;

        public PrimeService_IsPrimeShould()
        {
            _primeService = new PrimeService();
        }

        [TestMethod]
        public void IsPrime_InputIs1_ReturnFalse()
        {
            var result = _primeService.IsPrime(1);

            Assert.IsFalse(result, "1 should not be prime");
        }
    }
}
```

`TestClass` 属性表示包含单元测试的类。`TestMethod` 属性指示方法是测试方法。

保存此文件并执行 `dotnet test` 以构建测试和类库，然后运行测试。MSTest 测试运行程序包含要运行测试的程序入口点。`dotnet test` 使用已创建的单元测试项目启动测试运行程序。

测试失败。尚未创建实现。在起作用的 `PrimeService` 类中编写最简单的代码，使此测试通过：

```
public bool IsPrime(int candidate)
{
    if (candidate == 1)
    {
        return false;
    }
    throw new NotImplementedException("Please create a test first.");
}
```

在 `unit-testing-using-mstest` 目录中，再次运行 `dotnet test`。`dotnet test` 命令构建 `PrimeService` 项目，然后构建 `PrimeService.Tests` 项目。构建这两个项目后，该命令将运行此单项测试。测试通过。

添加更多功能

你已经通过了一个测试，现在可以编写更多测试。质数有其他几种简单情况：0, -1。可使用 `TestMethod 属性` 添加新测试，但这很快就会变得枯燥乏味。还有其他 MSTest 属性，使用这些属性可编写类似测试的套件。

`DataTestMethod 属性` 表示执行相同代码但具有不同输入参数的测试套件。可以使用 `DataRow 属性` 来指定这些输入的值。

可以不使用这两个属性创建新测试，而用来创建单个数据驱动的测试。数据驱动的测试方法用于测试多个小于 2（即最小质数）的值：

```
[DataTestMethod]
[DataRow(-1)]
[DataRow(0)]
[DataRow(1)]
public void IsPrime_ValuesLessThan2_ReturnFalse(int value)
{
    var result = _primeService.IsPrime(value);

    Assert.IsFalse(result, $"{value} should not be prime");
}
```

运行 `dotnet test`，两项测试均失败。若要使所有测试通过，可以更改方法开头的 `if` 子句：

```
if (candidate < 2)
```

通过在主库中添加更多测试、理论和代码继续循环访问。你将拥有已完成的测试版本和库的完整实现。

你已生成一个小型库和该库的一组单元测试。你已将解决方案结构化，使添加新包和新测试成为了正常工作流的一部分。你已将多数的时间和精力集中在解决应用程序的目标上。

另请参阅

- [Microsoft.VisualStudio.TestTools.UnitTesting](#)
- [在单元测试中使用 MSTest 框架](#)
- [MSTest V2 测试框架文档](#)

使用 dotnet test 和 xUnit 在 .NET Core 中进行 F# 库单元测试

2020/3/18 • [Edit Online](#)

本教程介绍分步构建示例解决方案的交互式体验，以了解单元测试概念。如果希望使用预构建解决方案学习本教程，请在开始前[查看或下载示例代码](#)。有关下载说明，请参阅[示例和教程](#)。

本文介绍如何测试 .NET Core 项目。如果要测试 ASP.NET Core 项目，请参阅[ASP.NET Core 中的集成测试](#)。

创建源项目

打开 shell 窗口。创建一个名为 unit-testing-with-fsharp 的目录，以保留该解决方案。在此新目录中，运行 `dotnet new sln` 创建新的解决方案。这样便于管理类库和单元测试项目。在解决方案库中，创建 MathService 目录。目录和文件结构目前如下所示：

```
/unit-testing-with-fsharp
  unit-testing-with-fsharp.sln
  /MathService
```

将 MathService 作为当前目录，然后运行 `dotnet new classlib -lang "F#"` 以创建源项目。创建数学服务的失败实现：

```
module MyMath =
    let squaresOfOdds xs = raise (System.NotImplementedException("You haven't written a test yet!"))
```

将目录更改回 unit-testing-with-fsharp 目录。运行 `dotnet sln add .\MathService\MathService.fsproj` 向解决方案添加类库项目。

创建测试项目

接下来，创建 MathService.Tests 目录。下图显示了它的目录结构：

```
/unit-testing-with-fsharp
  unit-testing-with-fsharp.sln
  /MathService
    Source Files
    MathService.fsproj
  /MathService.Tests
```

将 MathService.Tests 目录作为当前目录，并使用 `dotnet new xunit -lang "F#"` 创建一个新项目。这会创建将 xUnit 用作测试库的测试项目。生成的模板在 MathServiceTests.fsproj 中配置测试运行程序：

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.3.0-preview-20170628-02" />
  <PackageReference Include="xunit" Version="2.2.0" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.2.0" />
</ItemGroup>
```

测试项目需要其他包创建和运行单元测试。`dotnet new` 在以前的步骤中已添加 xUnit 和 xUnit 运行程序。现在，

将 `MathService` 类库作为另一个依赖项添加到项目中。使用 `dotnet add reference` 命令：

```
dotnet add reference ../MathService/MathService.fsproj
```

可以在 GitHub 上的[示例存储库](#)中看到整个文件。

最终的解决方案布局将如下所示：

```
/unit-testing-with-fsharp
  unit-testing-with-fsharp.sln
  /MathService
    Source Files
    MathService.fsproj
  /MathService.Tests
    Test Source Files
    MathServiceTests.fsproj
```

在 `unit-testing-with-fsharp` `dotnet sln add .\MathService.Tests\MathService.Tests.fsproj` 目录中执行。

创建第一个测试

编写一个失败测试，使其通过，然后重复此过程。打开 `Tests.fs` 并添加以下代码：

```
[<Fact>]
let ``My test`` () =
    Assert.True(true)

[<Fact>]
let ``Fail every time`` () = Assert.True(false)
```

`[<Fact>]` 属性表示由测试运行程序运行的测试方法。在 `unit-testing-with-fsharp` 中，执行 `dotnet test` 以构建测试和类库，然后运行测试。xUnit 测试运行程序包含要运行测试的程序入口点。`dotnet test` 使用已创建的单元测试项目启动测试运行程序。

这两个测试演示了最基本的已通过测试和未通过测试。`My test` 通过，而 `Fail every time` 未通过。现在创建针对 `squaresOfOdds` 方法的测试。`squaresOfOdds` 方法返回输入序列中所有奇整数值的平方序列。可以以迭代的方式创建可验证此功能的测试，而非尝试同时写入所有的函数。若要让每个测试都通过，意味着要针对此方法创建必要的功能。

可以编写的最简单的测试是调用包含所有偶数的 `squaresOfOdds`，它的结果应该是一个空整数序列。此测试如下所示：

```
[<Fact>]
let ``Sequence of Evens returns empty collection`` () =
    let expected = Seq.empty<int>
    let actual = MyMath.squaresOfOdds [2; 4; 6; 8; 10]
    Assert.Equal<Collections.Generic.IEnumerable<int>>(expected, actual)
```

测试失败。尚未创建实现。在起作用的 `MathService` 类中编写最简单的代码，使此测试通过：

```
let squaresOfOdds xs =
    Seq.empty<int>
```

在 `unit-testing-with-fsharp` 目录中，再次运行 `dotnet test`。`dotnet test` 命令构建 `MathService` 项目，然后构建 `MathService.Tests` 项目。构建这两个项目后，该命令将运行此单项测试。测试通过。

完成要求

你已经通过了一个测试，现在可以编写更多测试。下一个简单示例使用的序列包含的唯一奇数为 1。数值 1 较为简单，因为 1 的平方是 1。下一个测试如下所示：

```
[<Fact>]
let ``Sequences of Ones and Evens returns Ones`` () =
    let expected = [1; 1; 1; 1]
    let actual = MyMath.squaresOfOdds [2; 1; 4; 1; 6; 1; 8; 1; 10]
    Assert.Equal<Collections.Generic.IEnumerable<int>>(expected, actual)
```

执行 `dotnet test` 以运行测试，并显示新测试失败。现在更新 `squaresOfOdds` 方法，以处理此新测试。筛选出序列中的所有偶数值，以使此测试通过。可以编写一个小筛选器函数并使用 `Seq.filter` 来实现此目的：

```
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
```

还要执行一个步骤：计算每个奇数的平方值。从编写新测试开始：

```
[<Fact>]
let ``SquaresOfOdds works`` () =
    let expected = [1; 9; 25; 49; 81]
    let actual = MyMath.squaresOfOdds [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
    Assert.Equal(expected, actual)
```

可以通过映射操作传递经过筛选的序列来计算每个奇数的平方，以此方式来修复测试的缺陷：

```
let private square x = x * x
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
    |> Seq.map square
```

你已生成一个小型库和该库的一组单元测试。你已将解决方案结构化，使添加新包和新测试成为了正常工作流的一部分。你已将多数的时间和精力集中在解决应用程序的目标上。

另请参阅

- [dotnet new](#)
- [dotnet sln](#)
- [dotnet add reference](#)
- [dotnet test](#)

使用 dotnet test 和 NUnit 在 .NET Core 中进行 F# 库的单元测试

2020/3/18 • [Edit Online](#)

本教程介绍分步构建示例解决方案的交互式体验，以了解单元测试概念。如果希望使用预构建解决方案学习本教程，请在开始前[查看或下载示例代码](#)。有关下载说明，请参阅[示例和教程](#)。

本文介绍如何测试 .NET Core 项目。如果要测试 ASP.NET Core 项目，请参阅[ASP.NET Core 中的集成测试](#)。

系统必备

- [.NET Core 2.1 SDK 或更高版本](#)。
- 按需选择的文本编辑器或代码编辑器。

创建源项目

打开 shell 窗口。创建一个名为 unit-testing-with-fsharp 的目录，以保留该解决方案。在此新目录中，运行以下命令，为类库和测试项目创建新的解决方案文件：

```
dotnet new sln
```

接下来，创建 MathService 目录。下图显示了当前的目录和文件结构：

```
/unit-testing-with-fsharp
  unit-testing-with-fsharp.sln
  /MathService
```

将 MathService 作为当前目录，并运行以下命令以创建源项目：

```
dotnet new classlib -lang "F#"
```

创建数学服务的失败实现：

```
module MyMath =
    let squaresOfOdds xs = raise (System.NotImplementedException("You haven't written a test yet!"))
```

将目录更改回 unit-testing-with-fsharp 目录。运行以下命令，向解决方案添加类库项目：

```
dotnet sln add .\MathService\MathService.fsproj
```

创建测试项目

接下来，创建 MathService.Tests 目录。下图显示了它的目录结构：

```
/unit-testing-with-fsharp
  unit-testing-with-fsharp.sln
  /MathService
    Source Files
      MathService.fsproj
  /MathService.Tests
```

将 MathService.Tests 目录作为当前目录，并使用以下命令创建一个新项目：

```
dotnet new nunit -lang "F#"
```

这会创建一个将 NUnit 用作测试框架的测试项目。生成的模板在 MathServiceTests.fsproj 中配置测试运行程序：

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.5.0" />
  <PackageReference Include="NUnit" Version="3.9.0" />
  <PackageReference Include="NUnit3TestAdapter" Version="3.9.0" />
</ItemGroup>
```

测试项目需要其他包创建和运行单元测试。在上一步中，`dotnet new` 已添加 NUnit 和 NUnit 测试适配器。现在，将 `MathService` 类库作为另一个依赖项添加到项目中。使用 `dotnet add reference` 命令：

```
dotnet add reference ../MathService/MathService.fsproj
```

可以在 GitHub 上的[示例存储库](#)中看到整个文件。

最终的解决方案布局将如下所示：

```
/unit-testing-with-fsharp
  unit-testing-with-fsharp.sln
  /MathService
    Source Files
      MathService.fsproj
  /MathService.Tests
    Test Source Files
      MathService.Tests.fsproj
```

在 `unit-testing-with-fsharp` 目录中执行以下命令：

```
dotnet sln add .\MathService.Tests\MathService.Tests.fsproj
```

创建第一个测试

编写一个失败测试，使其通过，然后重复此过程。打开 `UnitTest1.fs` 并添加以下代码：

```

namespace MathService.Tests

open System
open NUnit.Framework
open MathService

[<TestFixture>]
type TestClass () =

    [<Test>]
    member this.TestMethodPassing() =
        Assert.True(true)

    [<Test>]
    member this.FailEveryTime() = Assert.True(false)

```

[**<TestFixture>**] 属性表示包含测试的类。[**<Test>**] 属性表示由测试运行程序运行的测试方法。在 unit-testing-with-fsharp 目录中，执行 `dotnet test` 以构建测试和类库，然后运行测试。NUnit 测试运行程序包含要运行测试的程序入口点。`dotnet test` 使用已创建的单元测试项目启动测试运行程序。

这两个测试演示了最基本的已通过测试和未通过测试。`My test` 通过，而 `Fail every time` 未通过。现在创建针对 `squaresOfOdds` 方法的测试。`squaresOfOdds` 方法返回输入序列中所有奇整数值的平方序列。可以以迭代的方式创建可验证此功能的测试，而非尝试同时写入所有的函数。若要让每个测试都通过，意味着要针对此方法创建必要的功能。

可以编写的最简单的测试是调用包含所有偶数的 `squaresOfOdds`，它的结果应该是一个空整数序列。此测试如下所示：

```

[<Test>]
member this.TestEvenSequence() =
    let expected = Seq.empty<int>
    let actual = MyMath.squaresOfOdds [2; 4; 6; 8; 10]
    Assert.That(actual, Is.EqualTo(expected))

```

请注意已将 `expected` 序列转换为列表。NUnit 框架依赖于许多标准 .NET 类型。此依赖关系表示公共接口和预期结果支持 [ICollection](#)，而非 [IEnumerable](#)。

运行此测试时，会看到测试失败。尚未创建实现。在起作用的 MathService 项目的 Library.fs 类中编写最简单的代码，使此测试通过：

```

let squaresOfOdds xs =
    Seq.empty<int>

```

在 unit-testing-with-fsharp 目录中，再次运行 `dotnet test`。`dotnet test` 命令构建 `MathService` 项目，然后构建 `MathService.Tests` 项目。构建这两个项目后，该命令将运行测试。现在两个测试通过。

完成要求

你已经通过了一个测试，现在可以编写更多测试。下一个简单示例使用的序列包含的唯一奇数为 `1`。数值 `1` 较为简单，因为 `1` 的平方是 `1`。下一个测试如下所示：

```

[<Test>]
member public this.TestOnesAndEvens() =
    let expected = [1; 1; 1; 1]
    let actual = MyMath.squaresOfOdds [2; 1; 4; 1; 6; 1; 8; 1; 10]
    Assert.That(actual, Is.EqualTo(expected))

```

如果执行 `dotnet test`，新测试将失败。必须更新 `squaresOfOdds` 方法才能处理此新测试。必须筛选出序列中的所有偶数值，以使此测试通过。可以编写一个小筛选器函数并使用 `Seq.filter` 来实现此目的：

```
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
```

注意对 `Seq.toList` 的调用。它会创建一个列表，此列表将实现 `ICollection` 接口。

还要执行一个步骤：计算每个奇数的平方值。从编写新测试开始：

```
[<Test>]
member public this.TestSquaresOfOdds() =
    let expected = [1; 9; 25; 49; 81]
    let actual = MyMath.squaresOfOdds [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
    Assert.That(actual, Is.EqualTo(expected))
```

可以通过映射操作传递经过筛选的序列来计算每个奇数的平方，以此方式来修复测试的缺陷：

```
let private square x = x * x
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
    |> Seq.map square
```

你已生成一个小型库和该库的一组单元测试。你已将解决方案结构化，使添加新包和新测试成为了正常工作流的一部分。你已将多数的时间和精力集中在解决应用程序的目标上。

另请参阅

- [dotnet add reference](#)
- [dotnet test](#)

使用 dotnet test 和 MSTest 在 .NET Core 中进行 F# 库单元测试

2020/3/18 • [Edit Online](#)

本教程介绍分步构建示例解决方案的交互式体验，以了解单元测试概念。如果希望使用预构建解决方案学习本教程，请在开始前[查看或下载示例代码](#)。有关下载说明，请参阅[示例和教程](#)。

本文介绍如何测试 .NET Core 项目。如果要测试 ASP.NET Core 项目，请参阅[ASP.NET Core 中的集成测试](#)。

创建源项目

打开 shell 窗口。创建一个名为 unit-testing-with-fsharp 的目录，以保留该解决方案。在此新目录中，运行 `dotnet new sln` 创建新的解决方案。这样便于管理类库和单元测试项目。在解决方案库中，创建 MathService 目录。目录和文件结构目前如下所示：

```
/unit-testing-with-fsharp
  unit-testing-with-fsharp.sln
  /MathService
```

将 MathService 作为当前目录，然后运行 `dotnet new classlib -lang "F#"` 以创建源项目。创建数学服务的失败实现：

```
module MyMath =
    let squaresOfOdds xs = raise (System.NotImplementedException("You haven't written a test yet!"))
```

将目录更改回 unit-testing-with-fsharp 目录。运行 `dotnet sln add .\MathService\MathService.fsproj` 向解决方案添加类库项目。

创建测试项目

接下来，创建 MathService.Tests 目录。下图显示了它的目录结构：

```
/unit-testing-with-fsharp
  unit-testing-with-fsharp.sln
  /MathService
    Source Files
    MathService.fsproj
  /MathService.Tests
```

将 MathService.Tests 目录作为当前目录，并使用 `dotnet new mstest -lang "F#"` 创建一个新项目。这会创建一个将 MSTest 用作测试框架的测试项目。生成的模板在 MathServiceTests.fsproj 中配置测试运行程序：

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.3.0-preview-20170628-02" />
  <PackageReference Include="MSTest.TestAdapter" Version="1.1.18" />
  <PackageReference Include="MSTest.TestFramework" Version="1.1.18" />
</ItemGroup>
```

测试项目需要其他包创建和运行单元测试。`dotnet new` 在前面的步骤中已添加 MSTest 和 MSTest 运行程序。现

在，将 `MathService` 类库作为另一个依赖项添加到项目中。使用 `dotnet add reference` 命令：

```
dotnet add reference ../MathService/MathService.fsproj
```

可以在 GitHub 上的[示例存储库](#)中看到整个文件。

最终的解决方案布局将如下所示：

```
/unit-testing-with-fsharp
  unit-testing-with-fsharp.sln
  /MathService
    Source Files
    MathService.fsproj
  /MathService.Tests
    Test Source Files
    MathServiceTests.fsproj
```

在 `unit-testing-with-fsharp` `dotnet sln add .\MathService.Tests\MathService.Tests.fsproj` 目录中执行。

创建第一个测试

编写一个失败测试，使其通过，然后重复此过程。打开 `Tests.fs` 并添加以下代码：

```
namespace MathService.Tests

open System
open Microsoft.VisualStudio.TestTools.UnitTesting
open MathService

[<TestClass>]
type TestClass () =

    [<TestMethod>]
    member this.TestMethodPassing() =
        Assert.IsTrue(true)

    [<TestMethod>]
    member this.FailEveryTime() = Assert.IsTrue(false)
```

`[<TestClass>]` 属性表示包含测试的类。`[<TestMethod>]` 属性表示由测试运行程序运行的测试方法。在 `unit-testing-with-fsharp` 目录中，执行 `dotnet test` 以构建测试和类库，然后运行测试。MSTest 测试运行程序包含要运行测试的程序入口点。`dotnet test` 使用已创建的单元测试项目启动测试运行程序。

这两个测试演示了最基本的已通过测试和未通过测试。`My test` 通过，而 `Fail every time` 未通过。现在创建针对 `squaresOfOdds` 方法的测试。`squaresOfOdds` 方法返回输入序列中所有奇整数值的平方列表。可以以迭代的方式创建可验证此功能的测试，而非尝试同时写入所有的函数。若要让每个测试都通过，意味着要针对此方法创建必要的功能。

可以编写的最简单的测试是调用包含所有偶数的 `squaresOfOdds`，它的结果应该是一个空整数序列。此测试如下所示：

```
[<TestMethod>]
member this.TestEvenSequence() =
    let expected = Seq.empty<int> |> Seq.toList
    let actual = MyMath.squaresOfOdds [2; 4; 6; 8; 10]
    Assert.AreEqual(expected, actual)
```

请注意已将 `expected` 序列转换为列表。MSTest 库依赖于许多标准 .NET 类型。此依赖关系表示公共接口和预期结果支持 `ICollection`, 而非 `IEnumerable`。

运行此测试时, 会看到测试失败。尚未创建实现。在起作用的 `Mathservice` 类中编写最简单的代码, 使此测试通过:

```
let squaresOfOdds xs =
    Seq.empty<int> |> Seq.toList
```

在 `unit-testing-with-fsharp` 目录中, 再次运行 `dotnet test`。`dotnet test` 命令构建 `MathService` 项目, 然后构建 `MathService.Tests` 项目。构建这两个项目后, 该命令将运行此单项测试。测试通过。

完成要求

你已经通过了一个测试, 现在可以编写更多测试。下一个简单示例使用的序列包含的唯一奇数为 `1`。数值 1 较为简单, 因为 1 的平方是 1。下一个测试如下所示:

```
[<TestMethod>]
member public this.TestOnesAndEvens() =
    let expected = [1; 1; 1; 1]
    let actual = MyMath.squaresOfOdds [2; 1; 4; 1; 6; 1; 8; 1; 10]
    Assert.AreEqual(expected, actual)
```

如果执行 `dotnet test`, 新测试将失败。必须更新 `squaresOfOdds` 方法才能处理此新测试。必须筛选出序列中的所有偶数值, 以使此测试通过。可以编写一个小筛选器函数并使用 `Seq.filter` 来实现此目的:

```
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd |> Seq.toList
```

注意对 `Seq.toList` 的调用。它会创建一个列表, 此列表将实现 `ICollection` 接口。

还要执行一个步骤: 计算每个奇数的平方值。从编写新测试开始:

```
[<TestMethod>]
member public this.TestSquaresOfOdds() =
    let expected = [1; 9; 25; 49; 81]
    let actual = MyMath.squaresOfOdds [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
    Assert.AreEqual(expected, actual)
```

可以通过映射操作传递经过筛选的序列来计算每个奇数的平方, 以此方式来修复测试的缺陷:

```
let private square x = x * x
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
    |> Seq.map square
    |> Seq.toList
```

你已生成一个小型库和该库的一组单元测试。你已将解决方案结构化, 使添加新包和新测试成为了正常工作流的一部分。你已将多数的时间和精力集中在解决应用程序的目标上。

另请参阅

- [dotnet new](#)
- [dotnet sln](#)
- [dotnet add reference](#)
- [dotnet test](#)

使用 dotnet test 和 xUnit 进行 Visual Basic .NET Core 库单元测试

2020/3/18 • [Edit Online](#)

本教程介绍分步构建示例解决方案的交互式体验，以了解单元测试概念。如果希望使用预构建解决方案学习本教程，请在开始前[查看或下载示例代码](#)。有关下载说明，请参阅[示例和教程](#)。

本文介绍如何测试 .NET Core 项目。如果要测试 ASP.NET Core 项目，请参阅[ASP.NET Core 中的集成测试](#)。

创建源项目

打开 shell 窗口。创建一个名为 unit-testing-vb-using-dotnet-test 的目录，以保留该解决方案。在此新目录中，运行 `dotnet new sln` 创建新的解决方案。此做法便于管理类库和单元测试项目。在解决方案目录中，创建 PrimeService 目录。目前目录和文件结构如下所示：

```
/unit-testing-using-dotnet-test
  unit-testing-using-dotnet-test.sln
  /PrimeService
```

将 PrimeService 作为当前目录，然后运行 `dotnet new classlib -lang VB` 以创建源项目。将 Class1.VB 重命名为 PrimeService.VB。创建 `PrimeService` 类的失败实现：

```
Namespace Prime.Services
    Public Class PrimeService
        Public Function IsPrime(candidate As Integer) As Boolean
            Throw New NotImplementedException("Please create a test first")
        End Function
    End Class
End Namespace
```

将目录更改回 unit-testing-vb-using-dotnet-test 目录。运行 `dotnet sln add .\PrimeService\PrimeService.vbproj` 向解决方案添加类库项目。

创建测试项目

接下来，创建 PrimeService.Tests 目录。下图显示了它的目录结构：

```
/unit-testing-vb-using-dotnet-test
  unit-testing-vb-using-dotnet-test.sln
  /PrimeService
    Source Files
      PrimeService.vbproj
    /PrimeService.Tests
```

将 PrimeService.Tests 目录作为当前目录，并使用 `dotnet new xunit -lang VB` 创建一个新项目。此命令会创建将 xUnit 用作测试库的测试项目。生成的模板在 PrimeServiceTests.vbproj 中配置了测试运行程序：

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.3.0-preview-20170628-02" />
  <PackageReference Include="xunit" Version="2.2.0" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.2.0" />
</ItemGroup>
```

测试项目需要其他包创建和运行单元测试。`dotnet new` 在以前的步骤中已添加 xUnit 和 xUnit 运行程序。现在，将 `PrimeService` 类库作为另一个依赖项添加到项目中。使用 `dotnet add reference` 命令：

```
dotnet add reference ..\PrimeService\PrimeService.vbproj
```

可以在 GitHub 上的[示例存储库](#)中看到整个文件。

最终的文件夹布局将如下所示：

```
/unit-testing-using-dotnet-test
  unit-testing-using-dotnet-test.sln
  /PrimeService
    Source Files
    PrimeService.vbproj
  /PrimeService.Tests
    Test Source Files
    PrimeServiceTests.vbproj
```

在 `unit-testing-vb-using-dotnet-test` 目录中执行 `dotnet sln add .\PrimeService.Tests\PrimeService.Tests.vbproj`。

创建第一个测试

编写一个失败测试，使其通过，然后重复此过程。从 `PrimeService.Tests` 目录删除 `UnitTest1.vb`，并创建一个名为 `PrimeService_IsPrimeShould.VB` 的新 Visual Basic 文件。添加以下代码：

```
Imports Xunit

Namespace PrimeService.Tests
  Public Class PrimeService_IsPrimeShould
    Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

    <Fact>
    Sub IsPrime_InputIs1_ReturnFalse()
      Dim result As Boolean = _primeService.IsPrime(1)

      Assert.False(result, "1 should not be prime")
    End Sub

  End Class
End Namespace
```

`<Fact>` 属性表示由测试运行程序运行的测试方法。在 `unit-testing-using-dotnet-test` 中，执行 `dotnet test` 以构建测试和类库，然后运行测试。xUnit 测试运行程序包含要运行测试的程序入口点。`dotnet test` 使用已创建的单元测试项目启动测试运行程序。

测试失败。尚未创建实现。在起作用的 `PrimeService` 类中编写最简单的代码，使此测试通过：

```

Public Function IsPrime(candidate As Integer) As Boolean
    If candidate = 1 Then
        Return False
    End If
    Throw New NotImplementedException("Please create a test first.")
End Function

```

在 unit-testing-vb-using-dotnet-test 目录中，再次运行 `dotnet test`。`dotnet test` 命令构建 `PrimeService` 项目，然后构建 `PrimeService.Tests` 项目。构建这两个项目后，该命令将运行此单项测试。测试通过。

添加更多功能

你已经通过了一个测试，现在可以编写更多测试。质数有其他几种简单情况：0, -1。可以将这些情况添加为具有 `<Fact>` 属性的新测试，但这很快就会变得枯燥乏味。还有其他 xUnit 属性，可使你编写类似测试套件。`<Theory>` 属性表示执行相同代码，但具有不同输入参数一系列测试。可以使用 `<InlineData>` 属性来指定这些输入的值。

可以不使用这两个属性创建新测试，而用来创建单个索引。此索引是测试多个小于 2（即最小的质数）的值的方法：

```

Public Class PrimeService_IsPrimeShould
    Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

    <Theory>
    <InlineData(-1)>
    <InlineData(0)>
    <InlineData(1)>
    Sub IsPrime_ValueLessThan2_ReturnFalse(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.False(result, $"{value} should not be prime")
    End Sub

    <Theory>
    <InlineData(2)>
    <InlineData(3)>
    <InlineData(5)>
    <InlineData(7)>
    Public Sub IsPrime_PrimesLessThan10_ReturnTrue(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.True(result, $"{value} should be prime")
    End Sub

    <Theory>
    <InlineData(4)>
    <InlineData(6)>
    <InlineData(8)>
    <InlineData(9)>
    Public Sub IsPrime_NonPrimesLessThan10_ReturnFalse(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.False(result, $"{value} should not be prime")
    End Sub
End Class

```

运行 `dotnet test`，两项测试均失败。若要使所有测试通过，可以更改方法开头的 `if` 子句：

```
if candidate < 2
```

通过在主库中添加更多测试、理论和代码继续循环访问。你将拥有已完成的测试版本和库的完整实现。

你已生成一个小型库和该库的一组单元测试。你已将解决方案结构化，使添加新包和新测试成为了正常工作流的

一部分。你已将多数的时间和精力集中在解决应用程序的目标上。

使用 dotnet test 和 NUnit 进行 Visual Basic .NET Core 库的单元测试

2020/3/18 • [Edit Online](#)

本教程介绍分步构建示例解决方案的交互式体验，以了解单元测试概念。如果希望使用预构建解决方案学习本教程，请在开始前[查看或下载示例代码](#)。有关下载说明，请参阅[示例和教程](#)。

本文介绍如何测试 .NET Core 项目。如果要测试 ASP.NET Core 项目，请参阅[ASP.NET Core 中的集成测试](#)。

系统必备

- [.NET Core 2.1 SDK 或更高版本](#)。
- 按需选择的文本编辑器或代码编辑器。

创建源项目

打开 shell 窗口。创建一个名为 unit-testing-vb-nunit 的目录，以保留该解决方案。在此新目录中，运行以下命令，为类库和测试项目创建新的解决方案文件：

```
dotnet new sln
```

接下来，创建 PrimeService 目录。下图显示了当前的文件结构：

```
/unit-testing-vb-nunit
  unit-testing-vb-nunit.sln
    /PrimeService
```

将 PrimeService 作为当前目录，并运行以下命令以创建源项目：

```
dotnet new classlib -lang VB
```

将 Class1.VB 重命名为 PrimeService.VB。创建 `PrimeService` 类的失败实现：

```
Namespace Prime.Services
    Public Class PrimeService
        Public Function IsPrime(candidate As Integer) As Boolean
            Throw New NotImplementedException("Please create a test first.")
        End Function
    End Class
End Namespace
```

将目录更改回 unit-testing-vb-using-mstest 目录。运行以下命令，向解决方案添加类库项目：

```
dotnet sln add .\PrimeService\PrimeService.vbproj
```

创建测试项目

接下来，创建 PrimeService.Tests 目录。下图显示了它的目录结构：

```
/unit-testing-vb-nunit  
  unit-testing-vb-nunit.sln  
  /PrimeService  
    Source Files  
      PrimeService.vbproj  
    /PrimeService.Tests
```

将 PrimeService.Tests 目录作为当前目录，并使用以下命令创建一个新项目：

```
dotnet new nunit -lang VB
```

[dotnet new](#) 命令可创建一个将 NUnit 用作测试库的测试项目。生成的模板在 PrimeServiceTests.vbproj 文件中配置了测试运行程序：

```
<ItemGroup>  
  <PackageReference Include="nunit" Version="3.12.0" />  
  <PackageReference Include="NUnit3TestAdapter" Version="3.16.0" />  
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.4.0" />  
</ItemGroup>
```

测试项目需要其他包创建和运行单元测试。在上一步中，[dotnet new](#) 已添加 NUnit 和 NUnit 测试适配器。现在，将 [PrimeService](#) 类库作为另一个依赖项添加到项目中。使用 [dotnet add reference](#) 命令：

```
dotnet add reference ..\PrimeService\PrimeService.vbproj
```

可以在 GitHub 上的[示例存储库](#)中看到整个文件。

最终的解决方案布局将如下所示：

```
/unit-testing-vb-nunit  
  unit-testing-vb-nunit.sln  
  /PrimeService  
    Source Files  
      PrimeService.vbproj  
    /PrimeService.Tests  
      Test Source Files  
      PrimeService.Tests.vbproj
```

在 unit-testing-vb-nunit 目录中执行以下命令：

```
dotnet sln add .\PrimeService.Tests\PrimeService.Tests.vbproj
```

创建第一个测试

编写一个失败测试，使其通过，然后重复此过程。在 PrimeService.Tests 目录中，将 UnitTest1.vb 文件重命名为 PrimeService_IsPrimeShould.VB，并将其整个内容替换为以下代码：

```

Imports NUnit.Framework

Namespace PrimeService.Tests
    <TestFixture>
    Public Class PrimeService_IsPrimeShould
        Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

        <Test>
        Sub IsPrime_InputIs1_ReturnFalse()
            Dim result As Boolean = _primeService.IsPrime(1)

            Assert.False(result, "1 should not be prime")
        End Sub

    End Class
End Namespace

```

<**TestFixture**> 属性指示包含测试的类。<**Test**> 属性表示由测试运行程序运行的方法。在 unit-testing-vb-nunit 中，执行 **dotnet test** 以构建测试和类库，然后运行测试。NUnit 测试运行程序包含要运行测试的程序入口点。**dotnet test** 使用已创建的单元测试项目启动测试运行程序。

测试失败。尚未创建实现。在起作用的 **PrimeService** 类中编写最简单的代码，使此测试通过：

```

Public Function IsPrime(candidate As Integer) As Boolean
    If candidate = 1 Then
        Return False
    End If
    Throw New NotImplementedException("Please create a test first.")
End Function

```

在 unit-testing-vb-nunit 目录中，再次运行 **dotnet test**。**dotnet test** 命令构建 **PrimeService** 项目，然后构建 **PrimeService.Tests** 项目。构建这两个项目后，该命令将运行此单项测试。测试通过。

添加更多功能

你已经通过了一个测试，现在可以编写更多测试。质数有其他几种简单情况：0, -1。可以将这些情况添加为具有 <**Test**> 属性的新测试，但这很快就会变得枯燥乏味。还有其他 xUnit 属性，可使你编写类似测试套件。
<TestCase> 属性表示执行相同代码，但具有不同输入参数一系列测试。可以使用 **<TestCase>** 属性来指定这些输入的值。

无需创建新测试，而是应用这两个属性来创建一系列测试，用于测试小于 2(最小质数)的几个值：

```

<TestFixture>
Public Class PrimeService_IsPrimeShould
    Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

    <TestCase(-1)>
    <TestCase(0)>
    <TestCase(1)>
    Sub IsPrime_ValuesLessThan2_ReturnFalse(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsFalse(result, $"{value} should not be prime")
    End Sub

    <TestCase(2)>
    <TestCase(3)>
    <TestCase(5)>
    <TestCase(7)>
    Public Sub IsPrime_PrimesLessThan10_ReturnTrue(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsTrue(result, $"{value} should be prime")
    End Sub

    <TestCase(4)>
    <TestCase(6)>
    <TestCase(8)>
    <TestCase(9)>
    Public Sub IsPrime_NonPrimesLessThan10_ReturnFalse(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsFalse(result, $"{value} should not be prime")
    End Sub
End Class

```

运行 `dotnet test`，两项测试均失败。若要使所有测试通过，可以在 `PrimeServices.cs` 文件中更改 `if` 方法开头的 `Main` 子句：

```
if candidate < 2
```

通过在主库中添加更多测试、理论和代码继续循环访问。你将拥有[已完成的测试版本](#)和[库的完整实现](#)。

你已生成一个小型库和该库的一组单元测试。你已将解决方案结构化，使添加新包和新测试成为了正常工作流的一部分。你已将多数的时间和精力集中在解决应用程序的目标上。

使用 dotnet test 和 MSTest 进行 Visual Basic .NET Core 库单元测试

2020/3/18 • [Edit Online](#)

本教程介绍分步构建示例解决方案的交互式体验，以了解单元测试概念。如果希望使用预构建解决方案学习本教程，请在开始前[查看或下载示例代码](#)。有关下载说明，请参阅[示例和教程](#)。

本文介绍如何测试.NET Core 项目。如果要测试 ASP.NET Core 项目，请参阅[ASP.NET Core 中的集成测试](#)。

创建源项目

打开 shell 窗口。创建一个名为 unit-testing-vb-mstest 的目录，以保留该解决方案。在此新目录中，运行 `dotnet new sln` 创建新的解决方案。此做法便于管理类库和单元测试项目。在解决方案目录中，创建 PrimeService 目录。目前目录和文件结构如下所示：

```
/unit-testing-vb-mstest
  unit-testing-vb-mstest.sln
  /PrimeService
```

将 PrimeService 作为当前目录，然后运行 `dotnet new classlib -lang VB` 以创建源项目。将 Class1.VB 重命名为 PrimeService.VB。创建 `PrimeService` 类的失败实现：

```
Namespace Prime.Services
    Public Class PrimeService
        Public Function IsPrime(candidate As Integer) As Boolean
            Throw New NotImplementedException("Please create a test first")
        End Function
    End Class
End Namespace
```

将目录更改回 unit-testing-vb-using-mstest 目录。运行 `dotnet sln add .\PrimeService\PrimeService.vbproj` 向解决方案添加类库项目。

创建测试项目

接下来，创建 PrimeService.Tests 目录。下图显示了它的目录结构：

```
/unit-testing-vb-mstest
  unit-testing-vb-mstest.sln
  /PrimeService
    Source Files
      PrimeService.vbproj
    /PrimeService.Tests
```

将 PrimeService.Tests 目录作为当前目录，并使用 `dotnet new mstest -lang VB` 创建一个新项目。此命令会创建一个将 MSTest 用作测试库的测试项目。生成的模板在 PrimeServiceTests.vbproj 中配置了测试运行程序：

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.5.0" />
  <PackageReference Include="MSTest.TestAdapter" Version="1.1.18" />
  <PackageReference Include="MSTest.TestFramework" Version="1.1.18" />
</ItemGroup>
```

测试项目需要其他包创建和运行单元测试。`dotnet new` 在前面的步骤中已添加 MSTest 和 MSTest 运行程序。现在，将 `PrimeService` 类库作为另一个依赖项添加到项目中。使用 `dotnet add reference` 命令：

```
dotnet add reference ..\PrimeService\PrimeService.vbproj
```

可以在 GitHub 上的[示例存储库](#)中看到整个文件。

最终的解决方案布局将如下所示：

```
/unit-testing-vb-mstest
  unit-testing-vb-mstest.sln
  /PrimeService
    Source Files
    PrimeService.vbproj
  /PrimeService.Tests
    Test Source Files
    PrimeServiceTests.vbproj
```

在 `unit-testing-vb-mstest` `dotnet sln add .\PrimeService.Tests\PrimeService.Tests.vbproj` 目录中执行。

创建第一个测试

编写一个失败测试，使其通过，然后重复此过程。从 `PrimeService.Tests` 目录删除 `UnitTest1.vb`，并创建一个名为 `PrimeService_IsPrimeShould.VB` 的新 Visual Basic 文件。添加以下代码：

```
Imports Microsoft.VisualStudio.TestTools.UnitTesting

Namespace PrimeService.Tests
  <TestClass>
  Public Class PrimeService_IsPrimeShould
    Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

    <TestMethod>
    Sub IsPrime_InputIs1_ReturnFalse()
      Dim result As Boolean = _primeService.IsPrime(1)

      Assert.IsFalse(result, "1 should not be prime")
    End Sub

  End Class
End Namespace
```

`<TestClass>` 属性指示包含测试的类。`<TestMethod>` 属性表示由测试运行程序运行的方法。在 `unit-testing-vb-mstest` 中，执行 `dotnet test` 以构建测试和类库，然后运行测试。MSTest 测试运行程序包含要运行测试的程序入口点。`dotnet test` 使用已创建的单元测试项目启动测试运行程序。

测试失败。尚未创建实现。在起作用的 `PrimeService` 类中编写最简单的代码，使此测试通过：

```

Public Function IsPrime(candidate As Integer) As Boolean
    If candidate = 1 Then
        Return False
    End If
    Throw New NotImplementedException("Please create a test first.")
End Function

```

在 unit-testing-vb-mstest 目录中，再次运行 `dotnet test`。`dotnet test` 命令构建 `PrimeService` 项目，然后构建 `PrimeService.Tests` 项目。构建这两个项目后，该命令将运行此单项测试。测试通过。

添加更多功能

你已经通过了一个测试，现在可以编写更多测试。质数有其他几种简单情况：0, -1。可以将这些情况添加为具有 `<TestMethod>` 属性的新测试，但这很快就会变得枯燥乏味。还有其他 MSTest 属性，使用这些属性可编写类似测试的套件。`<DataTestMethod>` 属性表示执行相同代码，但具有不同输入参数一系列测试。可以使用 `<DataRow>` 属性来指定这些输入的值。

可以不使用这两个属性创建新测试，而用来创建单个索引。此索引是测试多个小于 2（即最小的质数）的值的方法：

```

<TestClass>
Public Class PrimeService_IsPrimeShould
    Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

    <DataTestMethod>
    <DataRow(-1)>
    <DataRow(0)>
    <DataRow(1)>
    Sub IsPrime_ValuesLessThan2_ReturnFalse(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsFalse(result, $"{value} should not be prime")
    End Sub

    <DataTestMethod>
    <DataRow(2)>
    <DataRow(3)>
    <DataRow(5)>
    <DataRow(7)>
    Public Sub IsPrime_PrimesLessThan10_ReturnTrue(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsTrue(result, $"{value} should be prime")
    End Sub

    <DataTestMethod>
    <DataRow(4)>
    <DataRow(6)>
    <DataRow(8)>
    <DataRow(9)>
    Public Sub IsPrime_NonPrimesLessThan10_ReturnFalse(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsFalse(result, $"{value} should not be prime")
    End Sub
End Class

```

运行 `dotnet test`，两项测试均失败。若要使所有测试通过，可以更改方法开头的 `if` 子句：

```
if candidate < 2
```

通过在主库中添加更多测试、理论和代码继续循环访问。你将拥有[已完成的测试版本](#)和[库的完整实现](#)。

你已生成一个小型库和该库的一组单元测试。你已将解决方案结构化，使添加新包和新测试成为了正常工作流的一部分。你已将多数的时间和精力集中在解决应用程序的目标上。

运行选择性单元测试

2020/3/18 • [Edit Online](#)

借助 .NET Core 中的 `dotnet test` 命令，可以使用筛选表达式来运行选择性测试。本文演示如何筛选运行哪些测试。下面的示例使用 `dotnet test`。如果使用的是 `vstest.console.exe`，请将 `--filter` 替换成 `--testcasefilter:`。

NOTE

在 *nix 上使用包含感叹号 (!) 的筛选器需要转义，因为保留了 !。例如，如果命名空间包含 `IntegrationTests` `dotnet test --filter FullyQualifiedName\!~IntegrationTests`，则此筛选器将跳过所有测试。请注意感叹号前面的反斜杠。

MSTest

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace MSTestNamespace
{
    [TestClass]
    public class UnitTest1
    {
        [TestCategory("CategoryA")]
        [Priority(1)]
        [TestMethod]
        public void TestMethod1()
        {
        }

        [Priority(2)]
        [TestMethod]
        public void TestMethod2()
        {
        }
    }
}
```

EXPRESSION

IF

`dotnet test --filter Method`

运行 `FullyQualifiedName` 包含 `Method` 的测试。在 `vstest 15.1+` 中可用。

`dotnet test --filter Name~TestMethod1`

运行名称包含 `TestMethod1` 的测试。

`dotnet test --filter ClassName=MSTestNamespace.UnitTest1`

运行属于类 `MSTestNamespace.UnitTest1` 的测试。
■：由于 `ClassName` 值应有命名空间，因此 `ClassName=UnitTest1` 无效。

`dotnet test --filter FullyQualifiedName!=MSTestNamespace.UnitTest1.TestMethod1`

运行除 `MSTestNamespace.UnitTest1.TestMethod1` 之外的其他所有测试。

`dotnet test --filter TestCategory=CategoryA`

运行含 `[TestCategory("CategoryA")]` 批注的测试。

EXPRESSION	运行
<code>dotnet test --filter Priority=2</code>	运行含 <code>[Priority(2)]</code> 批注的测试。

使用条件运算符 | 和 &

EXPRESSION	运行
<code>dotnet test --filter "FullyQualifiedName~UnitTest1 TestCategory=CategoryA"</code>	运行 <code>UnitTest1</code> 包含 <code>FullyQualifiedName</code> 或 <code> </code> 的测试 <code>TestCategory `` CategoryA</code> 。
<code>dotnet test --filter "FullyQualifiedName~UnitTest1&TestCategory=CategoryA"</code>	运行 <code>UnitTest1</code> 包含 <code>FullyQualifiedName</code> 且 <code>&</code> 的测试 <code>TestCategory `` CategoryA</code> 。
<code>dotnet test --filter "(FullyQualifiedName~UnitTest1&TestCategory=CategoryA) Priority=1"</code>	运行 <code>FullyQualifiedName</code> 包含 <code>UnitTest1</code> 且 <code>&</code> 或 <code> </code> <code>Priority</code> 是 1 的测试 <code>CategoryA</code> <code>Priority</code> 。

xUnit

```
using Xunit;

namespace XUnitNamespace
{
    public class TestClass1
    {
        [Trait("Category", "CategoryA")]
        [Trait("Priority", "1")]
        [Fact]
        public void Test1()
        {
        }

        [Trait("Priority", "2")]
        [Fact]
        public void Test2()
        {
        }
    }
}
```

EXPRESSION	运行
<code>dotnet test --filter DisplayName=XUnitNamespace.TestClass1.Test1</code>	仅运行一个测试，即 <code>XUnitNamespace.TestClass1.Test1</code> 。
<code>dotnet test --filter FullyQualifiedName!=XUnitNamespace.TestClass1.Test1</code>	运行除 <code>XUnitNamespace.TestClass1.Test1</code> 之外的其他所有测试。
<code>dotnet test --filter DisplayName~TestClass1</code>	运行显示名称包含 <code>TestClass1</code> 的测试。

在代码示例中，包含键 `Category` 和 `Priority` 的已定义特征可用于筛选。

EXPRESSION	运行
------------	----

EXPRESSION	运行
<code>dotnet test --filter XUnit</code>	运行 <code>FullyQualifiedName</code> 包含 <code>XUnit</code> 的测试。在 <code>vstest 15.1+</code> 中可用。
<code>dotnet test --filter Category=CategoryA</code>	运行包含 <code>[Trait("Category", "CategoryA")]</code> 的测试。

使用条件运算符 | 和 &

EXPRESSION	运行
<code>dotnet test --filter "FullyQualifiedName~TestClass1 Category=CategoryA"</code>	运行 <code>TestClass1</code> 包含 <code>FullyQualifiedName</code> 或 <code>Category`CategoryA</code> 。
<code>dotnet test --filter "FullyQualifiedName~TestClass1&Category=CategoryA"</code>	运行 <code>TestClass1</code> 包含 <code>FullyQualifiedName</code> 且 <code>Category`CategoryA</code> 。
<code>dotnet test --filter "(FullyQualifiedName~TestClass1&Category=CategoryA) Priority=1"</code>	运行 <code>FullyQualifiedName</code> 包含 <code>TestClass1</code> 且 <code>Category`CategoryA</code> 或 <code>Priority`1</code> 。

NUnit

```
using NUnit.Framework;

namespace NUnitNamespace
{
    public class UnitTest1
    {
        [Category("CategoryA")]
        [Property("Priority", 1)]
        [Test]
        public void TestMethod1()
        {
        }

        [Property("Priority", 2)]
        [Test]
        public void TestMethod2()
        {
        }
    }
}
```

EXPRESSION	运行
<code>dotnet test --filter Method</code>	运行 <code>FullyQualifiedName</code> 包含 <code>Method</code> 的测试。在 <code>vstest 15.1+</code> 中可用。
<code>dotnet test --filter Name~TestMethod1</code>	运行名称包含 <code>TestMethod1</code> 的测试。
<code>dotnet test --filter FullyQualifiedName~NUnitNamespace.UnitTest1</code>	运行属于类 <code>NUnitNamespace.UnitTest1</code> 的测试。
<code>dotnet test --filter FullyQualifiedName!=NUnitNamespace.UnitTest1.TestMethod1</code>	运行除 <code>NUnitNamespace.UnitTest1.TestMethod1</code> 之外的其他所有测试。

EXPRESSION	运行
<code>dotnet test --filter TestCategory=CategoryA</code>	运行含 <code>[Category("CategoryA")]</code> 批注的测试。
<code>dotnet test --filter Priority=2</code>	运行含 <code>[Priority(2)]</code> 批注的测试。

使用条件运算符 | 和 &

EXPRESSION	运行
<code>dotnet test --filter "FullyQualifiedName~UnitTest1 TestCategory=CategoryA"</code>	运行 <code>UnitTest1</code> 包含 <code>FullyQualifiedName</code> 或 <code> </code> 的测试 <code>TestCategory``CategoryA</code> 。
<code>dotnet test --filter "FullyQualifiedName~UnitTest1&TestCategory=CategoryA"</code>	运行 <code>UnitTest1</code> 包含 <code>FullyQualifiedName</code> 且 <code>&</code> 的测试 <code>TestCategory``CategoryA</code> 。
<code>dotnet test --filter "(FullyQualifiedName~UnitTest1&TestCategory=CategoryA) Priority=1"</code>	运行 <code>FullyQualifiedName</code> 包含 <code>UnitTest1</code> 且 <code>&</code> 或 <code>TestCategory</code> 是 1 的测试 <code>CategoryA</code> <code>Priority</code> 。

通过 dotnet vstest 测试已发布的输出

2020/3/18 • [Edit Online](#)

可以使用 `dotnet vstest` 命令测试已发布的输出。这将适用于 xUnit、MSTest 和 NUnit 测试。只需找到属于已发布输出的 DLL 文件，然后运行：

```
dotnet vstest <MyPublishedTests>.dll
```

其中，`<MyPublishedTests>` 是已发布的测试项目的名称。

示例

下面的命令演示在已发布的 DLL 上运行测试。

```
dotnet new mstest -o MyProject.Tests
cd MyProject.Tests
dotnet publish -o out
dotnet vstest out/MyProject.Tests.dll
```

NOTE

注意：如果你的应用面向 `netcoreapp` 之外的框架，仍可通过传入带有框架标志的目标框架来运行 `dotnet vstest` 命令。例如 `dotnet vstest <MyPublishedTests>.dll --Framework:".NETFramework,Version=v4.6"`。在 Visual Studio 2017 Update 5 及更高版本中，自动检测所需的框架。

请参阅

- [使用 dotnet 测试和 xUnit 进行单元测试](#)
- [使用 dotnet 测试和 NUnit 进行单元测试](#)
- [使用 dotnet 测试和 MSTest 进行单元测试](#)

.NET Core 的版本控制方式概述

2020/3/18 • [Edit Online](#)

.NET Core 是指 .NET Core 运行时和 .NET Core SDK，它包含开发应用程序所需的工具。.NET Core SDK 可与任何以前版本的 .NET Core 运行时一起使用。本文介绍运行时和 SDK 版本策略。有关 .NET Standard 版本号的说明，请参阅介绍 [.NET Standard](#) 的文章。

.NET Core 运行时和 .NET Core SDK 以不同的速率添加新功能，通常情况下，.NET Core SDK 提供更新工具的速度比 .NET Core 运行时更改生产中所用运行时的速度快。

版本控制详细信息

“.NET Core 2.1”是指 .NET Core 运行时版本号。.NET Core 运行时用于版本控制的主要/次要/补丁方法需遵循语义版本控制。

.NET Core SDK 不遵循语义版本控制。.NET Core SDK 发布速度更快，其版本必须传达相应的运行时和 SDK 自己的次要版本和修补程序版本。.NET Core SDK 版本的前两个位置被锁定到与之一起发布的 .NET Core 运行时。每个版本的 SDK 都可以为此版本或任何更低版本的运行时创建应用程序。

SDK 版本号的第三个位置同时传达次要编号和修补程序编号。次要版本乘以 100。次要版本 1，修补程序版本 2 将表示为 102。最后两位数代表修补程序号。例如，.NET Core 2.2 可能会创建如下表所示的版本：

II	.NET CORE III	.NET CORE SDK (*)
初始版本	2.2.0	2.2.100
SDK 修补程序	2.2.0	2.2.101
运行时和 SDK 修补程序	2.2.1	2.2.102
SDK 功能更改	2.2.1	2.2.200

(*) 这个图表以 2.2 .NET Core 运行时为例，因为一个历史项目表明 .NET Core 2.1 的第一个 SDK 是 2.1.300。有关详细信息，请参阅 [.NET Core 版本选择页](#)。

注意：

- 如果在运行时功能更新之前，SDK 有 10 个功能更新，则版本号将滚动到 1000 系列，2.2.1000 等编号为 2.2.900 之后的功能版本。应该不会出现这种情况。
- 不会出现为发布功能的 99 修补程序版本。如果某版本接近此数字，则会强制发布功能。

可在 [dotnet/设计](#) 存储库中查看初始建议的更多详细信息。

语义化版本控制

.NET Core 运行时大致遵循语义版本控制 (*SemVer*)，采用 版本控制，通过版本号的各部分来描述更改程度和类型
MAJOR.MINOR.PATCH。

MAJOR.MINOR.PATCH[-PRERELEASE-BUILDDATE]

可选的 `PRERELEASE` 和 `BUILDDATE` 部分永远不会成为受支持版本的一部分，并且将仅存在于夜间版本、来自源目

标的本地版本，以及不受支持的预览版本中。

了解运行时版本号更改

MAJOR 在下列情况时递增：

- 产品或新产品方向发生重大更改。
- 发生了中断性变更。接受中断性变更存在较大障碍。
- 旧版本不再受支持。
- 采用了现有依赖项的较新 **MAJOR** 版本。

MINOR 在下列情况时递增：

- 添加了公共 API 外围应用。
- 添加了新行为。
- 采用了现有依赖项的较新 **MINOR** 版本。
- 引入了新依赖项。

PATCH 在下列情况时递增：

- 进行了 Bug 修复。
- 添加了对较新平台的支持。
- 采用了现有依赖项的较新 **PATCH** 版本。
- 任何其他不符合上述情况的更改。

存在多处更改时，单个更改影响的最高级别元素会递增，并将随后的元素重置为零。例如，当 **MAJOR** 递增时，**MINOR** 和 **PATCH** 将重置为零。当 **MINOR** 递增时，**PATCH** 将重置为零，而 **MAJOR** 保持不变。

文件名中的版本号

为 .NET Core 下载的文件带有版本，例如 `dotnet-sdk-2.1.300-win10-x64.exe`。

预览版

预览版向版本中追加了 `-preview[number]-([build]|"final")`。例如，`2.0.0-preview1-final`。

服务版本

在版本发布后，版本分支通常停止生成日常版本，而开始生成服务版本。服务版本向版本追加了 `-servicing-[number]`。例如，`2.0.1-servicing-006924`。

与 .NET Standard 版本的关系

.NET Standard 由 .NET 引用程序集组成。每个平台都有多个特定的实现。引用程序集包含 .NET API 的定义，后者是给定 .NET Standard 版本的一部分。每个实现均满足特定平台上的 .NET Standard 协定。可参阅 .NET 指南中有关 [.NET Standard](#) 的文章，深入了解 .NET Standard。

.NET Standard 引用程序集使用 **MAJOR.MINOR** 版本控制方案。**PATCH** 级别对 .NET Standard 并无用处，因为它只公开 API 规范（没有实现），并且根据定义，对 API 的任何更改都将作为功能集的更改，从而产生新的 **MINOR** 版本。

每个平台上的实现通常可作为平台版本的一部分更新，因此对于在该平台上使用 .NET Standard 的程序员来说并不明显。

每个版本的 .NET Core 都实现了某一版本的 .NET Standard。实现某一版本的 .NET Standard 意味着支持以前版本的 .NET Standard。.NET Standard 和 .NET Core 版本独立。.NET Core 2.0 实现 .NET Standard 2.0 是巧合。.NET Core 2.1 也可实现 .NET Standard 2.0。.NET Standard 的未来版本出现后，.NET Core 将支持这些版本。

.NET CORE	.NET STANDARD
1.0	达 1.6
2.0	达 2.0
2.1	达 2.0
2.2	达 2.0
3.0	至 2.1

另请参阅

- [目标框架](#)
- [.NET Core 分发打包](#)
- [.NET Core 支持生命周期简报](#)
- [.NET Core 2 和版本绑定](#)
- [.NET Core 的 Docker 映像](#)

选择要使用的 .NET Core 版本

2020/4/2 • [Edit Online](#)

本文介绍 .NET Core 工具、SDK 和运行时在选择版本时所使用的策略。这些策略可通过使用指定版本，使正在运行的应用程序之间达到平衡，同时实现开发人员和最终用户计算机的轻松升级。这些策略执行以下操作：

- 实现 .NET Core 的轻松高效部署，包括安全性和可靠性更新。
- 使用独立于目标运行时的最新工具和命令。

需要选择版本的情况如下：

- 运行 SDK 命令时，[SDK 使用最新安装的版本](#)。
- 生成程序集时，[目标框架名字对象定义生成时 API](#)。
- 运行 .NET Core 应用程序时，[依赖于目标框架的应用程序将前滚](#)。
- 发布独立应用程序时，[独立部署包括所选的运行时](#)。

本文档其余部分将介绍这四种方案。

SDK 使用最新安装的版本

SDK 命令包括 `dotnet new` 和 `dotnet run`。`.NET Core CLI` 必须为每个 `dotnet` 命令选择 SDK 版本。即使在以下情况下，它也会默认使用计算机上安装的最新 SDK：

- 项目面向早期 .NET Core 运行时版本。
- .NET Core SDK 的最新版本是预览版本。

面向较旧的 .NET Core 运行时版本时，可利用最新的 SDK 功能和功能改进。可在不同项目上面向 .NET Core 的多个运行时版本，同时对所有项目使用相同的 SDK 工具。

在少数情况下，可能需要使用版本较旧的 SDK。在 [global.json 文件](#) 中指定该版本。“使用最新”策略表示仅使用 global.json 指定早于最新安装版本的一个 .NET Core SDK 版本。

可将 global.json 放置在文件层次结构中的任何位置。CLI 从项目目录中向上搜索其找到的第一个 global.json。由用户控制对哪些项目应用给定的 global.json（按其在文件系统中的位置）。.NET CLI 从当前工作目录路径向上导航，以迭代方式搜索 global.json 文件。找到的第一个 global.json 文件指定要使用的版本。如果已安装该 SDK 版本，则使用该版本。如果找不到 global.json 中指定的 SDK，则 .NET CLI 将使用[匹配规则](#)来选择兼容的 SDK，如果找不到，则会失败。

下面的示例演示 global.json 语法：

```
{  
  "sdk": {  
    "version": "3.0.0"  
  }  
}
```

选择 SDK 版本的过程如下：

1. `dotnet` 从当前工作目录向下导航路径，以迭代方式搜索 global.json 文件。
2. `dotnet` 使用所找到的第一个 global.json 中指定的 SDK。
3. 如果未找到 global.json，`dotnet` 使用最新安装的 SDK。

有关选择 SDK 版本的详细信息，可参阅 global.json 相关文章的[匹配规则](#)部分。

目标框架名字对象用于定义生成时 API

针对在“目标框架名字对象”(TFM) 中定义的 API 构建项目。在项目文件中指定[目标框架](#)。按如下示例所示，设置项目文件中的 `TargetFramework` 元素：

```
<TargetFramework>netcoreapp3.0</TargetFramework>
```

可能会针对多个 TFM 构建项目。对库设置多个目标框架更为常见，但也可对应用程序执行此操作。指定

`TargetFrameworks` 属性 (`TargetFramework` 的复数形式)。目标框架以分号分隔，如下例所示：

```
<TargetFrameworks>netcoreapp3.0;net47</TargetFrameworks>
```

给定的 SDK 支持固定的一组框架，其中的上限框架为 SDK 附带的运行时的目标框架。例如，.NET Core 3.0 SDK 包含 .NET Core 3.0 运行时，该运行时是 `netcoreapp3.0` 目标框架的一个实现。.NET Core 3.0 SDK 支持 `netcoreapp2.1`、`netcoreapp2.2` 和 `netcoreapp3.0`，但不支持 `netcoreapp3.1` (或更高版本)。安装 .NET Core 3.1 SDK 以针对 `netcoreapp3.1` 进行构建。

.NET Standard 目标框架中的上限框架同样是 SDK 附带的运行时的目标框架。.NET Core 3.1 SDK 的上限为 `netstandard2.1`。有关详细信息，请参阅 [.NET Standard](#)。

依赖于框架的应用会前滚

在使用 `dotnet run` 从源运行应用程序时，在使用 `dotnet myapp.dll` 从[框架相关部署](#)运行应用程序时，或使用 `myapp.exe` 从[框架相关可执行文件](#)运行应用程序时，`dotnet` 可执行文件是应用程序的主机。

该主机选择计算机上安装的最新修补程序版本。例如，如果在项目文件中指定 `netcoreapp3.0`，并且 `3.0.4` 是安装的最新 .NET 运行时，则使用 `3.0.4` 运行时。

如果未找到可接受的 `3.0.*` 版本，则使用新的 `3.*` 版本。例如，如果指定了 `netcoreapp3.0` 并且仅安装了 `3.1.0`，则应用程序在运行时使用 `3.1.0` 运行时。此行为称为“次要版本前滚”。此外，不会考虑较低版本。如果未安装可接受的运行时，应用程序将不会运行。

如果你面向版本 3.0，则下面几个使用示例展示了此行为：

- ✓ 指定 3.0。3.0.5 是安装的最高修补程序版本。使用 3.0.5。
- ✗ 指定 3.0。未安装 3.0.* 版本。2.1.1 是安装的最高运行时版本。会显示一条错误消息。
- ✓ 指定 3.0。未安装 3.0.* 版本。3.1.0 是安装的最高运行时版本。使用 3.1.0。
- ✗ 指定 2.0。未安装 2.x 版本。3.0.0 是安装的最高运行时版本。会显示一条错误消息。

次要版本回滚会产生一个可能影响最终用户的副作用。请参考以下方案：

1. 应用程序指定需要版本 3.0。
2. 运行时，未安装 3.0.*，安装的是 3.1.0。将使用版本 3.1.0。
3. 稍后，用户重新安装 3.0.5 和运行应用程序，而现将使用版本 3.0.5。

3.0.5 和 3.1.0 可能具有不同行为，序列化二进制数据等方案中尤其如此。

独立部署包括所选的运行时

可以将应用程序作为[独立分发](#)进行发布。此方法可将 .NET Core 运行时和库与应用程序进行捆绑。独立部署不具有对运行时环境的依赖关系。在发布时(而不是运行时)选择运行时版本。

发布进程将选择给定运行时系列中的最新修补程序版本。例如，`dotnet publish` 将选择 .NET Core 3.0.4 (如果它是 .NET Core 3.0 运行时系列中的最新修补程序版本)。目标框架(包括最新安装的安全修补程序)与应用程序捆

绑打包。

如果为应用程序指定的最新版本不满足要求，会出现错误。`dotnet publish` 绑定到最新的运行时修补程序版本（在给定的主要及次要版本系列内）。`dotnet publish` 不支持 `dotnet run` 的前滚语义。有关修补程序和独立部署的详细信息，请参阅有关 .NET Core 应用程序部署中[运行时修补程序选择](#)的文章。

独立部署可能需要特定修补程序版本。可以重写项目文件中的最低运行时修补程序版本（重写为更高或更低版本），如下例所示：

```
<RuntimeFrameworkVersion>3.0.4</RuntimeFrameworkVersion>
```

`RuntimeFrameworkVersion` 元素重写默认版本策略。对于独立部署，`RuntimeFrameworkVersion` 指定确切的运行时框架版本。对于依赖于框架的应用程序，`RuntimeFrameworkVersion` 指定所需的最低运行时框架版本。

请参阅

- [下载和安装 .NET Core。](#)
- [如何删除 .NET Core 运行时和 SDK。](#)

如何删除 .NET Core 运行时和 SDK

2020/3/18 • [Edit Online](#)

经过一段时间后，在安装 .NET Core 运行时和 SDK 的更新版本时，用户可能需要从计算机中删除过时的 .NET Core 版本。如有关 [.NET Core 版本选择](#) 的文章中详述，删除旧版运行时可能会更改为运行共享框架应用程序所选择的运行时。

是否应删除某个版本？

借助 [.NET Core 版本选择](#) 行为和 .NET Core 各个更新之间的运行时兼容性，可安全地删除以前的版本。.NET Core 运行时更新在主版本“区段”（如 1.x 和 2.x）中兼容。此外，较新版本的 .NET Core SDK 通常能够以兼容的方式生成以运行时的早期版本为目标的应用程序。

通常，只需要应用程序所需的最新 SDK 和运行时的最新补丁版本。保留旧版 SDK 或运行时版本的实例包括维护基于 project.json 的应用程序。除非应用程序有需保留早期 SDK 或运行时的特定原因，否则可以安全地删除旧版本。

确定安装内容

从 .NET Core 2.1 开始，.NET CLI 提供一些可用于列出计算机上安装的 SDK 和运行时版本的选项。使用 `dotnet --list-sdks` 查看计算机上安装的 SDK 列表。使用 `dotnet --list-runtimes` 查看计算机上安装的运行时列表。下文显示了 Windows、macOS 或 Linux 的典型输出：

- [Windows](#)
- [Linux](#)
- [macOS](#)

通过运行以下命令：

```
dotnet --list-sdks
```

将获得类似于下面的输出：

```
2.1.200-preview-007474 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007480 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007509 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007570 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007576 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007587 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007589 [C:\Program Files\dotnet\sdk]
2.1.200 [C:\Program Files\dotnet\sdk]
2.1.201 [C:\Program Files\dotnet\sdk]
2.1.202 [C:\Program Files\dotnet\sdk]
2.1.300-preview2-008533 [C:\Program Files\dotnet\sdk]
2.1.300 [C:\Program Files\dotnet\sdk]
2.1.400-preview-009063 [C:\Program Files\dotnet\sdk]
2.1.400-preview-009088 [C:\Program Files\dotnet\sdk]
2.1.400-preview-009171 [C:\Program Files\dotnet\sdk]
```

并通过运行以下命令：

```
dotnet --list-runtimes
```

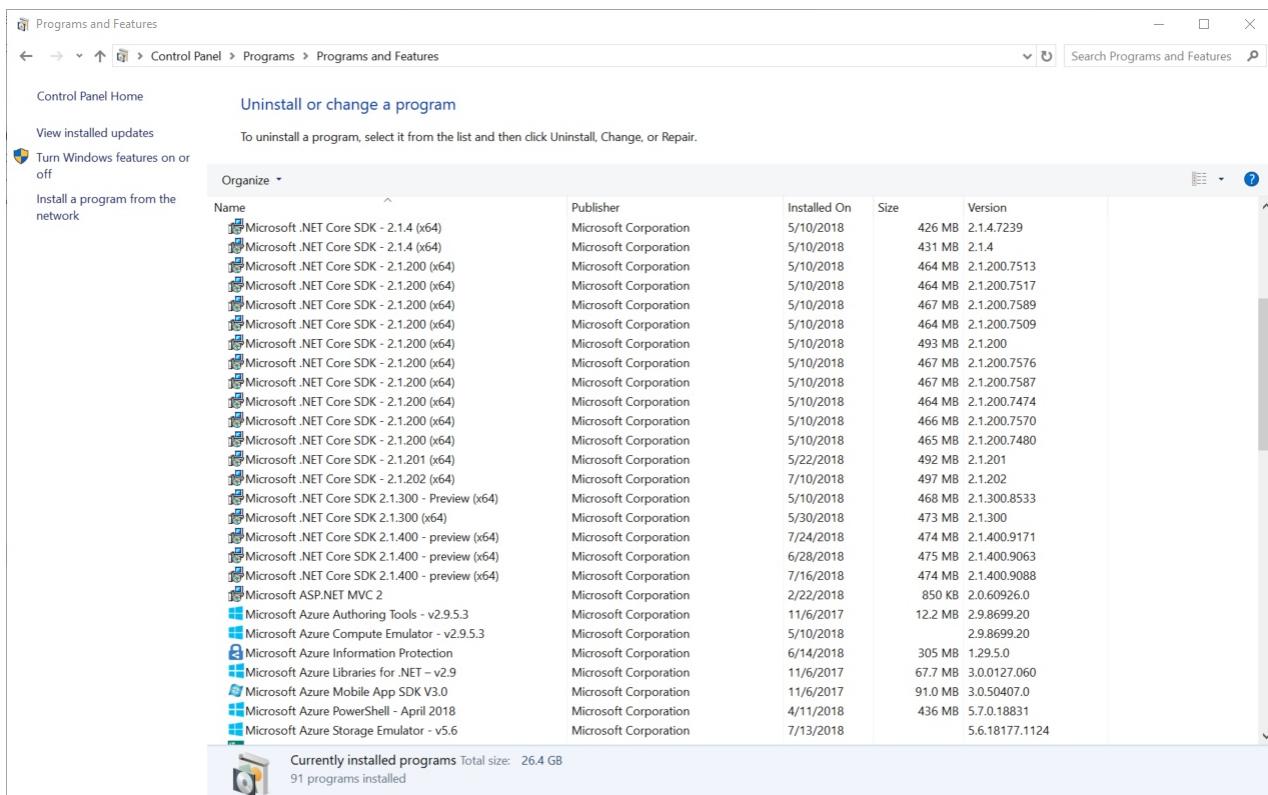
将获得类似于下面的输出：

```
Microsoft.AspNetCore.All 2.1.0-preview2-final [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.All]
Microsoft.AspNetCore.All 2.1.0 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.All]
Microsoft.AspNetCore.All 2.1.1 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.All]
Microsoft.AspNetCore.All 2.1.2 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.All]
Microsoft.AspNetCore.App 2.1.0-preview2-final [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 2.1.0 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 2.1.1 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 2.1.2 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.NETCore.App 2.0.6 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.0.7 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.0.9 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.1.0-preview2-26406-04 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.1.0 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.1.1 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.1.2 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
```

卸载 .NET Core

- [Windows](#)
- [Linux](#)
- [macOS](#)

.NET Core 使用 Windows“添加/删除程序”对话框来删除 .NET Core 运行时和 SDK 的版本。下图显示了“添加/删除程序”对话框，其中包含已安装的多个版本的 .NET 运行时和 SDK。



选择要从计算机中删除的任何版本，然后单击“卸载”。

.NET Core 卸载工具

[.NET Core 卸载工具](#) (`dotnet-core-uninstall`) 使你可以从系统中删除 .NET Core SDK 和运行时。可使用选项集合来指定应卸载的版本。

.NET Core SDK 版本的 Visual Studio 依赖项

在 Visual Studio 2019 版本 16.3 之前, Visual Studio 安装程序称为独立的 .NET Core SDK 安装程序。因此, SDK 版本显示在 Windows“添加/删除程序”对话框中。使用独立安装程序删除 Visual Studio 安装的 .NET Core SDK 可能会破坏 Visual Studio。如果 Visual Studio 在卸载 SDK 之后出现问题, 请在该特定版本的 Visual Studio 上运行修复。下表显示了 .NET Core SDK 版本的一些 Visual Studio 依赖项:

VISUAL STUDIO	.NET CORE SDK
Visual Studio 2019 版本 16.2	.NET Core SDK 2.2.4xx, 2.1.8xx
Visual Studio 2019 版本 16.1	.NET Core SDK 2.2.3xx, 2.1.7xx
Visual Studio 2019 版本 16.0	.NET Core SDK 2.2.2xx, 2.1.6xx
Visual Studio 2017 版本 15.9	.NET Core SDK 2.2.1xx, 2.1.5xx
Visual Studio 2017 版本 15.8	.NET Core SDK 2.1.4xx

从 Visual Studio 2019 版本 16.3 开始, Visual Studio 负责其自己的 .NET Core SDK 副本。为此, 在“添加/删除程序”对话框中将不再显示这些 SDK 版本。

删除 NuGet 回退文件夹

在 .NET Core 3.0 SDK 之前, .NET Core SDK 安装程序使用 NuGetFallbackFolder 存储 NuGet 包的缓存。此缓存在操作期间(如 `dotnet restore` 或 `dotnet build /t:Restore`)使用。`NuGetFallbackFolder` 在 Windows 位于 C:\Program Files\dotnet\sdk, 在 macOS 上位于 /usr/local/share/dotnet/sdk。

如果是以下情况, 则可能需要删除此文件夹:

- 仅使用 .NET Core 3.0 SDK 或更高版本进行开发。
- 你使用早于 3.0 的 .NET Core SDK 版本进行开发, 但可以联机工作, 并且操作速度可能会慢一些。

如果要删除 NuGet 回退文件夹, 可以将其删除, 但需要管理员权限才能执行此操作。

建议不要删除 dotnet 文件夹。这样做会删除以前安装的所有全局工具。此外, 在 Windows 上:

- 你将中断 Visual Studio 2019 版本 16.3 及更高版本。可以运行“修复”来恢复。
- 如果“添加/删除程序”对话框中存在 .NET Core SDK 条目, 它们将是孤立的。

.NET Core RID 目录

2020/3/18 • [Edit Online](#)

RID 是运行时标识符的缩写。RID 值用于标识应用程序运行所在的目标平台。.NET 包使用它们来表示 NuGet 包中特定于平台的资产。以下值是 RID 的示例：`linux-x64`、`ubuntu.14.04-x64`、`win7-x64` 或 `osx.10.12-x64`。对于具有本机依赖项的包，RID 将指定在其中可以还原包的平台。

可以在项目文件的 `<RuntimeIdentifier>` 元素中设置一个 RID。可以将多个 RID 定义为项目文件的 `<RuntimeIdentifiers>` 元素中的列表（以分号分隔）。也可使用以下 `--runtime` .NET Core CLI 命令 通过 选项使用它们：

- [dotnet build](#)
- [dotnet clean](#)
- [dotnet pack](#)
- [dotnet publish](#)
- [dotnet restore](#)
- [dotnet run](#)
- [dotnet store](#)

表示具体操作系统的 RID 通常遵循以下模式：`[os].[version]-[architecture]-[additional qualifiers]`，其中：

- `[os]` 是操作系统/平台系统名字对象。例如，`ubuntu`。
- `[version]` 是操作系统版本，使用的格式是以点（`.`）分隔的版本号。例如，`15.10`。
 - 版本不应为营销版本，因为它们通常代表该操作系统的多个离散版本，且具有不同的平台 API 外围应用。
- `[architecture]` 是处理器体系结构。例如：`x86`、`x64`、`arm` 或 `arm64`。
- `[additional qualifiers]` 进一步区分了不同的平台。例如：`aot`。

RID 图表

RID 图表或运行时回退图表是互相兼容的 RID 列表。[Microsoft.NETCore.Platforms](#) 包中定义了 RID。可以在 [存储库的 runtime.json](#) `dotnet/runtime` 文件中查看支持的 RID 列表和 RID 图表。在此文件中，可以看到除基 RID 以外的所有 RID 均包含 `#import` 语句。这些语句指示的是兼容的 RID。

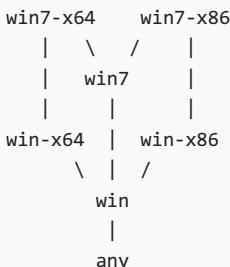
NuGet 还原包时，它将尝试找到指定运行时的完全匹配项。如果未找到完全匹配项，NuGet 将返回此图表，直至它根据 RID 图表找到最相近的兼容系统。

以下示例是 `osx.10.12-x64` RID 的实际条目：

```
"osx.10.12-x64": {  
    "#import": [ "osx.10.12", "osx.10.11-x64" ]  
}
```

上述 RID 指定 `osx.10.12-x64` 导入 `osx.10.11-x64`。因此，当 NuGet 还原包时，它将尝试找到包中的 `osx.10.12-x64` 的完全匹配项。例如，如果 NuGet 无法找到特定的运行时，可以还原指定 `osx.10.11-x64` 运行时的包。

以下示例演示了 `runtime.json` 文件中定义的另一个略大的 RID 图表：



所有 RID 最终都会映射回根 `any` RID。

使用 RID 时，必须牢记以下几个注意事项：

- RID 是不透明字符串，应将其视为黑盒。
- 请勿以编程方式生成 RID。
- 使用已为平台定义的 RID。
- RID 必须具有特定性，因此请勿通过实际的 RID 值假定任何情况。

使用 RID

若要使用 RID，必须知道有哪些 RID。新值将定期添加到该平台。若要获取最新的完整版，请参阅 [存储库上的 runtime.json](#) `dotnet/runtime` 文件。

.NET Core 2.0 SDK 引入了可移植 RID 的概念。它们是添加到 RID 图表的新值，并且未与特定版本或 OS 发行版本关联，是使用 .NET Core 2.0 及更高版本的首选值。它们在处理多个 Linux 发行版时特别有用，因为大多数发行版 RID 都映射到可移植的 RID。

以下列表显示了一小部分用于每个 OS 的最常见 RID。

Windows RID

仅列出了公共值。若要获取最新的完整版，请参阅 [存储库上的 runtime.json](#) `dotnet/runtime` 文件。

- 可移植(.NET Core 2.0 或更高版本)
 - `win-x64`
 - `win-x86`
 - `win-arm`
 - `win-arm64`
- Windows 7 / Windows Server 2008 R2
 - `win7-x64`
 - `win7-x86`
- Windows 8.1 / Windows Server 2012 R2
 - `win81-x64`
 - `win81-x86`
 - `win81-arm`
- Windows 10 / Windows Server 2016
 - `win10-x64`
 - `win10-x86`
 - `win10-arm`
 - `win10-arm64`

有关详细信息，请参阅 [.NET Core 依赖项和要求](#)。

Linux RID

仅列出了公共值。若要获取最新的完整版，请参阅 [存储库上的 runtime.json](#) `dotnet/runtime` 文件。运行以下未列出的发行版的设备可能适用于其中一个可移植 RID。例如，可以将运行未列出的 Linux 发行版的 Raspberry Pi 设备定向为使用 `linux-arm`。

- 可移植(.NET Core 2.0 或更高版本)
 - `linux-x64` (大多数桌面发行版，如 CentOS、Debian、Fedora、Ubuntu 及派生版本)
 - `linux-musl-x64` (使用 `musl` 的轻量级发行版，如 Alpine Linux)
 - `linux-arm` (在 ARM 上运行的 Linux 分发版，如 Raspberry Pi)
- Red Hat Enterprise Linux
 - `rhel-x64` (被 `linux-x64` 取代，适用于 RHEL 6 以上版本)
 - `rhel.6-x64` (.NET Core 2.0 或更高版本)
- Tizen(.NET Core 2.0 或更高版本)
 - `tizen`
 - `tizen.4.0.0`
 - `tizen.5.0.0`

有关详细信息，请参阅 [.NET Core 依赖项和要求](#)。

macOS RID

macOS RID 使用较早的“OSX”品牌。仅列出了公共值。若要获取最新的完整版，请参阅 [存储库上的 runtime.json](#) `dotnet/runtime` 文件。

- 可移植(.NET Core 2.0 或更高版本)
 - `osx-x64` (最低 OS 版本为 macOS 10.12 Sierra)
- macOS 10.10 Yosemite
 - `osx.10.10-x64`
- macOS 10.11 El Capitan
 - `osx.10.11-x64`
- macOS 10.12 Sierra(.NET Core 1.1 或更高版本)
 - `osx.10.12-x64`
- macOS 10.13 High Sierra(.NET Core 1.1 或更高版本)
 - `osx.10.13-x64`
- macOS 10.14 Mojave(.NET Core 1.1 或更高版本)
 - `osx.10.14-x64`

有关详细信息，请参阅 [.NET Core 依赖项和要求](#)。

另请参阅

- [运行时 ID](#)

管理 .NET Core 应用程序中的依赖项

2020/3/18 • [Edit Online](#)

本文介绍如何通过编辑项目文件或使用 CLI 来添加和删除依赖项。

<PackageReference> 元素

<PackageReference> 项目文件元素具有以下结构：

```
<PackageReference Include="PACKAGE_ID" Version="PACKAGE_VERSION" />
```

`Include` 特性指定要添加到项目的包的 ID。 `Version` 特性指定要获取的版本。 版本根据 [NuGet 版本规则](#) 进行指定。

NOTE

如果不熟悉“项目-文件”语法，可参阅 [MSBuild 项目参考](#) 文档了解详细信息。

使用条件来添加仅在特定目标中可用的依赖项，如以下示例所示：

```
<PackageReference Include="PACKAGE_ID" Version="PACKAGE_VERSION" Condition="$(TargetFramework) == 'netcoreapp2.1'" />
```

上述示例中的依赖项只有在对给定目标生成时才有效。 条件中的 `$(TargetFramework)` 是将在项目中设置的 MSBuild 属性。 对于大多数常见的 .NET Core 应用程序，无需这样做。

通过编辑项目文件添加依赖项

若要添加依赖项，请在 `<ItemGroup>` 元素内添加 `<PackageReference>` 元素。 可以添加到现有 `<ItemGroup>`，也可以新建一个元素。 下面的示例使用 `dotnet new console` 创建的默认控制台应用程序项目：

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="3.1.2" />
  </ItemGroup>

</Project>
```

使用 CLI 添加依赖项

若要添加依赖项，请运行 `dotnet add package` 命令，如以下示例中所示：

```
dotnet add package Microsoft.EntityFrameworkCore
```

通过编辑项目文件删除依赖项

若要删除依赖项，请从项目文件中删除其 `<PackageReference>` 元素。

使用 CLI 删除依赖项

若要删除依赖项，请运行 `dotnet remove package` 命令，如以下示例中所示：

```
dotnet remove package Microsoft.EntityFrameworkCore
```

请参阅

- [项目文件中的 NuGet 包](#)
- [dotnet list package 命令](#)

.NET Core 中的依赖项加载

2020/3/18 • [Edit Online](#)

每个 .NET Core 应用程序都有依赖项。即使是简单的 `hello world` 应用程序也在 .NET Core 类库的各个部分中有依赖项。

了解 .NET Core 默认程序集加载逻辑有助于理解和调试典型的部署问题。

在某些应用程序中，在运行时动态确定依赖项。在这些情况下，了解托管程序集和非托管依赖项的加载方式至关重要。

了解 AssemblyLoadContext

[AssemblyLoadContext API](#) 是 .NET Core 加载设计的核心。了解 [AssemblyLoadContext](#) 一文提供了有关设计的概念性概述。

加载详细信息

以下几篇文章简要介绍了加载算法的详细信息：

- 托管程序集加载算法
- 附属程序集加载算法
- 非托管(本机)库加载算法
- 默认探测

使用插件创建 .NET Core 应用程序

[创建包含插件的 .NET Core 应用程序](#) 教程介绍了如何创建自定义 AssemblyLoadContext。它使用 [AssemblyDependencyResolver](#) 来解析插件的依赖项。该教程正确地将插件依赖项与主机应用程序隔离开来。

如何在 .NET Core 中使用和调试程序集可卸载性

[如何在 .NET Core 中使用和调试程序集可卸载性](#) 一文是逐步骤教程。其中显示了如何加载 .NET Core 应用程序、执行以及将其卸载。该文章还提供了调试提示。

了解 System.Runtime.Loader.AssemblyLoadContext

2020/4/6 • [Edit Online](#)

[AssemblyLoadContext](#) 类对于 .NET Core 是唯一的。本文尝试使用概念性信息来补充 [AssemblyLoadContext API 文档](#)。

本文与实现动态加载的开发人员有关，尤其是动态加载框架的开发人员。

什么是 AssemblyLoadContext？

每个 .NET Core 应用程序均隐式使用 [AssemblyLoadContext](#)。它是运行时的提供程序，用于定位和加载依赖项。只要加载了依赖项，就会调用 [AssemblyLoadContext](#) 实例来定位该依赖项。

- 它提供定位、加载和缓存托管程序集和其他依赖项的服务。
- 为了支持动态代码加载和卸载，它创建了一个独立上下文，用于在其自己的 [AssemblyLoadContext](#) 实例中加载代码及其依赖项。

何时需要多个 AssemblyLoadContext 实例？

单个 [AssemblyLoadContext](#) 实例限制为每个简单程序集名称 [AssemblyName.Name](#) 只加载 [Assembly](#) 的一个版本。

当动态加载代码模块时，此限制可能会成为一个问题。每个模块都是独立编译的，并且可能依赖于不同版本的 [Assembly](#)。当不同的模块依赖于常用库的不同版本时，通常会出现此问题。

为了支持动态加载代码，[AssemblyLoadContext API](#) 提供在同一个应用程序中加载 [Assembly](#) 冲突版本的功能。每个 [AssemblyLoadContext](#) 实例提供一个唯一字典，该字典将每个 [AssemblyName.Name](#) 映射到特定的 [Assembly](#) 实例。

它还提供了一种方便的机制，将与代码模块相关的依赖项分组，以便以后进行卸载。

AssemblyLoadContext.Default 实例有什么特别之处？

启动时，运行时将自动填充 [AssemblyLoadContext.Default](#) 实例。它使用 [默认探测](#) 来定位和查找所有静态依赖项。

它解决了最常见的依赖项加载方案。

AssemblyLoadContext 如何支持动态依赖项？

[AssemblyLoadContext](#) 具有可替代的各种事件和虚函数。

[AssemblyLoadContext.Default](#) 实例仅支持替代事件。

[托管程序集加载算法](#)、[附属程序集加载算法](#) 和 [非托管\(本机\)库加载算法](#) 文章引用了所有可用事件和虚拟函数。这些文章显示加载算法中的每个事件和函数的相对位置。本文不会重现该信息。

本部分介绍相关事件和函数的一般原则。

- **可重复**。针对特定依赖项的查询必须始终产生相同的响应。必须返回同一个已加载的依赖项实例。此要求是保持缓存一致性的基础。特别是对于托管程序集，我们要创建 [Assembly](#) 缓存。缓存键是一个简单的程序集名称 [AssemblyName.Name](#)。
- **通常不引发**。当找不到请求的依赖项时，这些函数应返回 `null` 而不是引发。引发将提前结束搜索，并将异常传至调用方。应将引发限制为针对意外错误，如程序集损坏或内存不足等情况。

- **避免递归。**请注意，这些函数和处理程序实现了用于定位依赖项的加载规则。实现不应调用触发递归的 API。代码通常应调用 AssemblyLoadContext 加载函数，这些函数需要特定路径或内存引用参数。
- **加载到正确的 AssemblyLoadContext。**选择加载依赖项的位置是应用程序特定的。选择是通过这些事件和函数实现的。当代码调用 AssemblyLoadContext 时，按路径加载函数在你要加载代码的实例上调用它们。有时返回 null，并让 AssemblyLoadContext.Default 处理加载可能是最简单的选项。
- **注意线程争用。**加载可由多个线程触发。AssemblyLoadContext 通过以原子方式将程序集添加到其缓存来处理线程争用。将丢弃争用失败方的实例。在实现逻辑中，不要添加未正确处理多个线程的额外逻辑。

如何隔离动态依赖项？

每个 AssemblyLoadContext 实例都表示 Assembly 实例和 Type 定义的唯一范围。

这些依赖项之间没有任何二进制隔离。它们仅通过不按名称查找彼此来进行隔离。

在每个 AssemblyLoadContext 中：

- AssemblyName.Name 可以引用不同的 Assembly 实例。
- Type.GetType 可能会为同一类型 name 返回不同类型的实例。

如何共享依赖项？

可以在 AssemblyLoadContext 实例之间轻松共享依赖项。常规模型用于一个 AssemblyLoadContext 来加载依赖项。另一个通过使用对已加载程序集的引用来共享依赖项。

此共享是运行时程序集所必需的。这些程序集只能加载到 AssemblyLoadContext.Default。ASP.NET、WPF 或 WinForms 等框架也是如此。

建议将共享依赖项加载到 AssemblyLoadContext.Default。此共享是常见的设计模式。

共享是通过自定义 AssemblyLoadContext 实例编码实现的。AssemblyLoadContext 具有可替代的各种事件和虚函数。当这些函数中的任何函数返回对在另一个 AssemblyLoadContext 实例中加载的 Assembly 实例的引用时，将共享 Assembly 实例。标准加载算法会延迟 AssemblyLoadContext.Default 加载，以简化通用共享模式。请参阅托管程序集加载算法。

复杂情况

类型转换问题

当两个 AssemblyLoadContext 实例包含具有相同 name 的类型定义时，它们不是同一类型。当且仅当它们来自同一个 Assembly 实例时，它们的类型相同。

使事情复杂化的是，这些不匹配类型的异常消息可能会令人困惑。在异常消息中，按简单类型名称来引用这些类型。在这种情况下，常见异常消息的格式如下所示：

无法将类型为“IsolatedType”的对象转换为类型“IsolatedType”。

调试类型转换问题

如果给定一对不匹配的类型，还必须了解：

- 每种类型的 Type.Assembly
- 每种类型的 AssemblyLoadContext，这可以通过 AssemblyLoadContext.GetLoadContext(Assembly) 函数获得。

如果给定两个对象 a 和 b，在调试器中评估以下内容将非常有用：

```
// In debugger look at each assembly's instance, Location, and FullName
a.GetType().Assembly
b.GetType().Assembly
// In debugger look at each AssemblyLoadContext's instance and name
System.Runtime.Loader.AssemblyLoadContext.GetLoadContext(a.GetType().Assembly)
System.Runtime.Loader.AssemblyLoadContext.GetLoadContext(b.GetType().Assembly)
```

解决类型转换问题

可以通过两种设计模式来解决这些类型转换问题。

1. 使用常见的共享类型。此共享类型可以是基元运行时类型，也可以涉及在共享程序集中创建新的共享类型。
共享类型通常是在应用程序的程序集中定义的[接口](#)。另请参阅：[如何共享依赖项？](#)。
2. 使用封送处理技术从一种类型转换为另一种类型。

默认探测

2020/3/18 • [Edit Online](#)

`AssemblyLoadContext.Default` 实例负责定位程序集的依赖项。本文介绍 `AssemblyLoadContext.Default` 实例的探测逻辑。

主机配置的探测属性

运行时启动时，运行时主机提供一组命名的探测属性，这些属性可配置 `AssemblyLoadContext.Default` 探测路径。

每个探测属性均可选。如果存在，则每个属性都是一个字符串值，其中包含绝对路径的分隔列表。在 Windows 上，分隔符为“;”，在所有其他平台上，分隔符为“:”。

属性	描述
<code>TRUSTED_PLATFORM_ASSEMBLIES</code>	平台和应用程序程序集文件路径的列表。
<code>PLATFORM_RESOURCE_ROOTS</code>	用于搜索附属资源程序集的目录路径的列表。
<code>NATIVE_DLL_SEARCH_DIRECTORIES</code>	用于搜索非托管(本机)库的目录路径的列表。
<code>APP_PATHS</code>	用于搜索托管程序集的目录路径的列表。
<code>APP_NI_PATHS</code>	用于搜索托管程序集的本机映像的目录路径的列表。

如何填充属性？

填充属性有两个主要方案，具体取决于 `myapp>.deps.json` 文件是否存在`<`。

- 当 `.deps.json` 文件存在时，将对其进行分析以填充探测属性*。
- 如果 `.deps.json*` 文件不存在，则假定应用程序的目录以包含所有依赖项。目录的内容用于填充探测属性。

此外，也会对任何引用框架的 `.deps.json*` 文件进行类似分析。

最后，可使用环境变量 `ADDITIONAL_DEPS` 添加其他依赖项。

如何查看托管代码中的探测属性？

通过使用上表中的属性名称调用 `AppContext.GetData(String)` 函数，可查看每个属性。

如何调试探测属性的构造？

启用某些环境变量后，.NET Core 运行时主机将输出有用的跟踪消息：

环境变量	描述
<code>COREHOST_TRACE=1</code>	启用跟踪。
<code>COREHOST_TRACEFILE=<path></code>	跟踪文件路径而不是默认 <code>stderr</code> 。
<code>COREHOST_TRACE_Verbosity</code>	将详细程度设置为从 1(最低)到 4(最高)。

托管程序集默认探测

当探测以定位托管程序集时, `AssemblyLoadContext.Default` 会按以下顺序查找:

- 与 `AssemblyName.Name` 中的 `TRUSTED_PLATFORM_ASSEMBLIES` 匹配的文件(在删除文件扩展名之后)。
- 包含公共文件扩展名的 `APP_NI_PATHS` 中的本机映像程序集文件。
- 包含公共文件扩展名的 `APP_PATHS` 中的程序集文件。

附属(资源)程序集探测

若要查找特定区域性的附属程序集, 请构造一组文件路径。

对于 `PLATFORM_RESOURCE_ROOTS` 和 `APP_PATHS` 中的每个路径, 附加 `CultureInfo.Name` 字符串、目录分隔符、`AssemblyName.Name` 字符串和扩展名".dll"。

如果存在任何匹配的文件, 请尝试加载并返回该文件。

非托管(本机)库探测

当探测以查找非托管库时, 将搜索 `NATIVE_DLL_SEARCH_DIRECTORIES` 以查找匹配库。

托管程序集加载算法

2020/3/19 • [Edit Online](#)

托管程序集与涉及不同阶段的算法一起定位并加载。

除附属程序集和 [WinRT](#) 程序集之外的所有托管程序集都使用相同算法。

何时加载托管程序集？

触发托管程序集加载的最常见机制是静态程序集引用。每当代码使用在另一个程序集中定义的类型时，编译器都会插入这些引用。根据运行时的需要加载这些程序集 ([load-by-name](#))。

直接使用特定的 API 也将触发加载：

API	如果	ACTIVE ASSEMBLYLOADCONTEXT
<code>AssemblyLoadContext.LoadFromAssemblyName</code>	<code>Load-by-name</code>	<code>this</code> 实例。
<code>AssemblyLoadContext.LoadFromAssemblyPath</code> <code>AssemblyLoadContext.LoadFromNativeImagePath</code>	从路径加载。	<code>this</code> 实例。
<code>AssemblyLoadContext.LoadFromStream</code>	从对象加载。	<code>this</code> 实例。
<code>Assembly.LoadFile</code>	在新的 <code>AssemblyLoadContext</code> 实例中从路径加载	新的 <code>AssemblyLoadContext</code> 实例。
<code>Assembly.LoadFrom</code>	在 <code>AssemblyLoadContext.Default</code> 实例中从路径加载。 向 <code>AssemblyLoadContext.Default</code> 添加 <code>Resolving</code> 处理程序。处理程序将从其目录加载程序集的依赖项。	<code>AssemblyLoadContext.Default</code> 实例。
<code>Assembly.Load(AssemblyName)</code> <code>Assembly.Load(String)</code> <code>Assembly.LoadWithPartialName</code>	<code>Load-by-name</code> 。	从调用方推断。 首选 <code>AssemblyLoadContext</code> 方法。
<code>Assembly.Load(Byte[])</code> <code>Assembly.Load(Byte[], Byte[])</code>	从新 <code>AssemblyLoadContext</code> 实例的对象中加载。	新的 <code>AssemblyLoadContext</code> 实例。
<code>Type.GetType(String)</code> <code>Type.GetType(String, Boolean)</code> <code>Type.GetType(String, Boolean, Boolean)</code>	<code>Load-by-name</code> 。	从调用方推断。 首选使用 <code>assemblyResolver</code> 参数的 <code>Type.GetType</code> 方法。

API		ACTIVE ASSEMBLYLOADCONTEXT
Assembly.GetType	如果类型 <code>name</code> 描述程序集限定的泛型类型，则触发 <code>Load-by-name</code> 。	从调用方推断。 使用程序集限定的类型名称时，首选 <code>Type.GetType</code> 。
<code>Activator.CreateInstance(String, String)</code> <code>Activator.CreateInstance(String, String, Object[])</code> <code>Activator.CreateInstance(String, String, Boolean, BindingFlags, Binder, Object[], CultureInfo, Object[])</code>	<code>Load-by-name</code> 。	从调用方推断。 首选采用 <code>Type</code> 参数的 <code>Activator.CreateInstance</code> 方法。

算法

以下算法描述运行时如何加载托管程序集。

1. 确定 `active AssemblyLoadContext`。

- 对于静态程序集引用，`active AssemblyLoadContext` 是已加载引用程序集的实例。
- 首选 API 使 `active AssemblyLoadContext` 显式。
- 其他 API 推断 `active AssemblyLoadContext`。对于这些 API，将使用 `AssemblyLoadContext.CurrentContextualReflectionContext` 属性。如果其值为 `null`，则使用推断的 `AssemblyLoadContext` 实例。
- 请见上表。

2. 对于 `Load-by-name` 方法，活动 `AssemblyLoadContext` 将加载程序集。通过以下方式按优先级排序：

- 检查其 `cache-by-name`。
- 调用 `AssemblyLoadContext.Load` 函数。
- 检查 `AssemblyLoadContext.Default` 实例的缓存并运行 **托管程序集默认探测** 逻辑。
- 引发 `AssemblyLoadContext` 活动的 `AssemblyLoadContext.Resolving` 事件。
- 引发 `AppDomain.AssemblyResolve` 事件。

3. 对于其他类型的加载，`active AssemblyLoadContext` 加载程序集。通过以下方式按优先级排序：

- 检查其 `cache-by-name`。
- 从指定的路径或原始程序集对象加载。

4. 在任一情况下，如果新加载了一个程序集，则：

- 引发 `AppDomain.AssemblyLoad` 事件。
- 将引用添加到程序集的 `AssemblyLoadContext` 实例的 `cache-by-name`。

5. 如果找到了程序集，则会根据需要将引用添加到 `active AssemblyLoadContext` 实例的 `cache-by-name` 中。

附属程序集加载算法

2020/3/19 • [Edit Online](#)

使用附属程序集来存储为语言和区域性自定义的本地化资源。

附属程序集使用不同于常规托管程序集的加载算法。

何时加载附属程序集？

加载本地化资源时加载附属程序集。

加载本地化资源的基本 API 是 `System.Resources.ResourceManager` 类。最后, `ResourceManager` 类将为每个 `CultureInfo.Name` 调用 `GetSatelliteAssembly` 方法。

较高级别的 API 可能会提取低级别 API。

算法

.NET Core 资源回退进程包含以下步骤:

1. 确定 `active AssemblyLoadContext` 实例。在所有情况下, `active` 实例都是执行程序集的 `AssemblyLoadContext`。
 - 检查其缓存。
 - 检查当前正在执行程序集的目录, 查找与请求的 `CultureInfo.Name`(例如 `es-MX`)匹配的子目录。
2. `active` 实例尝试通过以下方式按优先级排序来加载请求的区域性的附属程序集:
 - 检查其缓存。
 - 检查当前正在执行程序集的目录, 查找与请求的 `CultureInfo.Name`(例如 `es-MX`)匹配的子目录。

NOTE

3.0 版之前的 .NET Core 中未实现此功能。

NOTE

在 Linux 和 macOS 上, 子目录区分大小写, 并且必须是以下两种情况之一:

- 完全匹配大小写。
- 为小写。

- 如果 `active` 是 `AssemblyLoadContext.Default` 实例, 则通过运行默认附属(资源)程序集探测逻辑。

- 调用 `AssemblyLoadContext.Load` 函数。

- 引发 `AssemblyLoadContext.Resolving` 事件。

- 引发 `AppDomain.AssemblyResolve` 事件。

3. 如果加载附属程序集:

- 引发 `AppDomain.AssemblyLoad` 事件。

- 将搜索程序集以查找请求的资源。如果运行时在程序集中找到该资源, 则使用它。如果找不到该资源, 将继续搜索。

NOTE

要在附属程序集中查找资源，运行时将搜索 `ResourceManager` 为当前 `CultureInfo.Name` 请求的资源文件。在资源文件中，它搜索请求的资源名称。如果找不到上述任何一个，则资源将被视为未找到。

4. 接下来，运行时通过许多潜在级别搜索父区域性程序集，每次均重复步骤 2 和 3。

每个区域性只有一个父级，由 `CultureInfo.Parent` 属性定义。

当区域性的 `Parent` 属性为 `CultureInfo.InvariantCulture` 时，父区域性搜索将停止。

对于 `InvariantCulture`，我们不会返回到步骤 2 和 3，而是继续执行步骤 5。

5. 如果仍未找到资源，则使用默认(回退)区域性的资源。

通常，默认区域性的资源包含在主应用程序集中。不过，可以为

`NeutralResourcesLanguageAttribute.Location` 属性指定 `UltimateResourceFallbackLocation.Satellite`。此值指示资源的最终回退位置是附属程序集，而不是主程序集。

NOTE

默认区域性为最终回退。因此，建议在默认资源文件中始终包含一组详尽的资源。这有助于防止引发异常。通过拥有详尽资源集，可为所有资源提供回退，并确保始终向用户呈现至少一种资源，即使该资源不是特定于区域性的。

6. 最后，

- 如果运行时找不到默认(回退)区域性的资源文件，将引发 `MissingManifestResourceException` 或 `MissingSatelliteAssemblyException` 异常。
- 如果找到资源文件但是请求的资源不存在，则请求返回 `null`。

非托管（本机）库加载算法

2020/3/19 • [Edit Online](#)

非托管库与涉及不同阶段的算法一起定位并加载。

以下算法描述如何通过 `PInvoke` 加载本机库。

`PInvoke` 加载库算法

`PInvoke` 在尝试加载非托管程序集时使用以下算法：

1. 确定 `active AssemblyLoadContext`。对于非托管加载库，`active AssemblyLoadContext` 是具有定义 `PInvoke` 的程序集的算法。
2. 对于 `active AssemblyLoadContext`，尝试通过以下方式按优先级排序来查找程序集：
 - 检查其缓存。
 - 调用由 `NativeLibrary.SetDllImportResolver(Assembly, DllImportResolver)` 函数设置的当前 `System.Runtime.InteropServices.DllImportResolver` 委托。
 - 对 `active AssemblyLoadContext` 调用 `AssemblyLoadContext.LoadUnmanagedDll` 函数。
 - 检查 `AppDomain` 实例的缓存并运行 **非托管（本机）库探测** 逻辑。
 - 引发 `active AssemblyLoadContext` 的 `AssemblyLoadContext.ResolvingUnmanagedDll` 事件。

使用插件创建 .NET Core 应用程序

2020/3/18 • [Edit Online](#)

本教程展示了如何创建自定义的 `AssemblyLoadContext` 来加载插件。`AssemblyDependencyResolver` 用于解析插件的依赖项。该教程正确地将插件依赖项与主机应用程序隔离开来。将了解如何执行以下操作：

- 构建支持插件的项目。
- 创建自定义 `AssemblyLoadContext` 加载每个插件。
- 使用 `System.Runtime.Loader.AssemblyDependencyResolver` 类型允许插件具有依赖项。
- 只需复制生成项目就可以轻松部署的作者插件。

系统必备

- 安装 `.NET Core 3.0 SDK` 或更高版本。

创建应用程序

第一步是创建应用程序：

1. 创建新文件夹，并在该文件夹中运行以下命令：

```
dotnet new console -o AppWithPlugin
```

2. 为了更容易生成项目，请在同一文件夹中创建一个 Visual Studio 解决方案文件。运行以下命令：

```
dotnet new sln
```

3. 运行以下命令，向解决方案添加应用项目：

```
dotnet sln add AppWithPlugin/AppWithPlugin.csproj
```

现在，我们可以填写应用程序的主干。使用下面的代码替换 `AppWithPlugin/Program.cs` 文件中的代码：

```
using PluginBase;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;

namespace AppWithPlugin
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                if (args.Length == 1 && args[0] == "/d")
                {
                    Console.WriteLine("Waiting for any key...");
                    Console.ReadLine();
                }

                // Load commands from plugins.

                if (args.Length == 0)
                {
                    Console.WriteLine("Commands: ");
                    // Output the loaded commands.
                }
                else
                {
                    foreach (string commandName in args)
                    {
                        Console.WriteLine($"-- {commandName} --");

                        // Execute the command with the name passed as an argument.

                        Console.WriteLine();
                    }
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex);
            }
        }
    }
}
```

创建插件接口

使用插件生成应用的下一步是定义插件需要实现的接口。我们建议创建类库，其中包含计划用于在应用和插件之间通信的任何类型。此部分允许将插件接口作为包发布，而无需发布完整的应用程序。

在项目的根文件夹中，运行 `dotnet new classlib -o PluginBase`。并运行

`dotnet sln add PluginBase/PluginBase.csproj` 向解决方案文件添加项目。删除 `PluginBase/Class1.cs` 文件，并使用以下接口定义在名为 `PluginBase` 的 `ICommand.cs` 文件夹中创建新的文件：

```
namespace PluginBase
{
    public interface ICommand
    {
        string Name { get; }
        string Description { get; }

        int Execute();
    }
}
```

此 `ICommand` 接口是所有插件将实现的接口。

由于已定义 `ICommand` 接口，所以应用程序项目可以填写更多内容。使用根文件夹中的 `AppWithPlugin` 命令将引用从 `PluginBase` 项目添加到 `dotnet add AppWithPlugin\AppWithPlugin.csproj reference PluginBase\PluginBase.csproj` 项目。

使用以下代码片段替换 `// Load commands from plugins` 注释，使其能够从给定文件路径加载插件：

```
string[] pluginPaths = new string[]
{
    // Paths to plugins to load.
};

IEnumerable<ICommand> commands = pluginPaths.SelectMany(pluginPath =>
{
    Assembly pluginAssembly = LoadPlugin(pluginPath);
    return CreateCommands(pluginAssembly);
}).ToList();
```

然后用以下代码片段替换 `// Output the loaded commands` 注释：

```
foreach (ICommand command in commands)
{
    Console.WriteLine($"{command.Name}\t - {command.Description}");
}
```

使用以下代码片段替换 `// Execute the command with the name passed as an argument` 注释：

```
ICommand command = commands.FirstOrDefault(c => c.Name == commandName);
if (command == null)
{
    Console.WriteLine("No such command is known.");
    return;
}

command.Execute();
```

最后，将静态方法添加到名为 `Program` 和 `LoadPlugin` 的 `CreateCommands` 类，如下所示：

```
static Assembly LoadPlugin(string relativePath)
{
    throw new NotImplementedException();
}

static IEnumerable< ICommand> CreateCommands(Assembly assembly)
{
    int count = 0;

    foreach (Type type in assembly.GetTypes())
    {
        if (typeof(ICommand).IsAssignableFrom(type))
        {
            ICommand result = Activator.CreateInstance(type) as ICommand;
            if (result != null)
            {
                count++;
                yield return result;
            }
        }
    }

    if (count == 0)
    {
        string availableTypes = string.Join(", ", assembly.GetTypes().Select(t => t.FullName));
        throw new ApplicationException(
            $"Can't find any type which implements ICommand in {assembly} from {assembly.Location}.\n" +
            $"Available types: {availableTypes}");
    }
}
```

加载插件

现在，应用程序可以正确加载和实例化来自已加载的插件程序集的命令，但仍然无法加载插件程序集。使用以下内容在 AppWithPlugin 文件夹中创建名为 PluginLoadContext.cs 的文件：

```

using System;
using System.Reflection;
using System.Runtime.Loader;

namespace AppWithPlugin
{
    class PluginLoadContext : AssemblyLoadContext
    {
        private AssemblyDependencyResolver _resolver;

        public PluginLoadContext(string pluginPath)
        {
            _resolver = new AssemblyDependencyResolver(pluginPath);
        }

        protected override Assembly Load(AssemblyName assemblyName)
        {
            string assemblyPath = _resolver.ResolveAssemblyToPath(assemblyName);
            if (assemblyPath != null)
            {
                return LoadFromAssemblyPath(assemblyPath);
            }

            return null;
        }

        protected override IntPtr LoadUnmanagedDll(string unmanagedDllName)
        {
            string libraryPath = _resolver.ResolveUnmanagedDllToPath(unmanagedDllName);
            if (libraryPath != null)
            {
                return LoadUnmanagedDllFromPath(libraryPath);
            }

            return IntPtr.Zero;
        }
    }
}

```

`PluginLoadContext` 类型派生自 `AssemblyLoadContext`。`AssemblyLoadContext` 类型是运行时中的特殊类型，该类型允许开发人员将已加载的程序集隔离到不同的组中，以确保程序集版本不冲突。此外，自定义 `AssemblyLoadContext` 可以选择不同路径来加载程序集格式并重写默认行为。`PluginLoadContext` 使用 .NET Core 3.0 中引入的 `AssemblyDependencyResolver` 类型的实例将程序集名称解析为路径。`AssemblyDependencyResolver` 对象是使用 .NET 类库的路径构造的。它根据类库的 `.deps.json` 文件（其路径传递给 构造函数）将程序集和本机库解析为它们的相对路径 `AssemblyDependencyResolver`。自定义 `AssemblyLoadContext` 使插件能够拥有自己的依赖项，`AssemblyDependencyResolver` 使正确加载依赖项变得容易。

由于 `AppWithPlugin` 项目具有 `PluginLoadContext` 类型，所以请使用以下正文更新 `Program.LoadPlugin` 方法：

```

static Assembly LoadPlugin(string relativePath)
{
    // Navigate up to the solution root
    string root = Path.GetFullPath(Path.Combine(
        Path.GetDirectoryName(
            Path.GetDirectoryName(
                Path.GetDirectoryName(
                    Path.GetDirectoryName(
                        Path.GetDirectoryName(typeof(Program).Assembly.Location))))));

    string pluginLocation = Path.GetFullPath(Path.Combine(root, relativePath.Replace('\\',
    Path.DirectorySeparatorChar)));
    Console.WriteLine($"Loading commands from: {pluginLocation}");
    PluginLoadContext loadContext = new PluginLoadContext(pluginLocation);
    return loadContext.LoadFromAssemblyName(new
    AssemblyName(Path.GetFileNameWithoutExtension(pluginLocation)));
}

```

通过为每个插件使用不同的 `PluginLoadContext` 实例，插件可以具有不同的甚至冲突的依赖项，而不会出现问题。

不具有依赖项的简单插件

返回到根文件夹，执行以下步骤：

1. 运行以下命令，新建一个名为 `HelloPlugin` 的类库项目：

```
dotnet new classlib -o HelloPlugin
```

2. 运行以下命令，将项目添加到 `AppWithPlugin` 解决方案中：

```
dotnet sln add HelloPlugin/HelloPlugin.csproj
```

3. 使用以下内容将 `HelloPlugin/Class1.cs` 文件替换为名为 `HelloCommand.cs` 的文件：

```

using PluginBase;
using System;

namespace HelloPlugin
{
    public class HelloCommand : ICommand
    {
        public string Name { get => "hello"; }
        public string Description { get => "Displays hello message." }

        public int Execute()
        {
            Console.WriteLine("Hello !!!");
            return 0;
        }
    }
}

```

现在，打开 `HelloPlugin.csproj` 文件。它应类似于以下内容：

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
</PropertyGroup>

</Project>
```

在 `<Project>` 标记之间添加以下元素：

```
<ItemGroup>
    <ProjectReference Include="..\PluginBase\PluginBase.csproj">
        <Private>false</Private>
        <ExcludeAssets>runtime</ExcludeAssets>
    </ProjectReference>
</ItemGroup>
```

`<Private>false</Private>` 元素很重要。它告知 MSBuild 不要将 `PluginBase.dll` 复制到 `HelloPlugin` 的输出目录。如果 `PluginBase.dll` 程序集出现在输出目录中，将在那里查找到该程序集并在加载 `HelloPlugin.dll` 程序集时加载它 `PluginLoadContext`。此时，`HelloPlugin.HelloCommand` 类型将从 `ICommand` 项目的输出目录中的 `PluginBase.dll` 实现 接口，而不是加载到默认加载上下文中的接口 `HelloPlugin`ICommand`。因为运行时将这两种类型视为不同程序集的不同类型，所以 `AppWithPlugin.Program.CreateCommands` 方法找不到命令。因此，对包含插件接口的程序集的引用需要 `<Private>false</Private>` 元数据。

同样，如果 `<ExcludeAssets>runtime</ExcludeAssets>` 引用其他包，则 `PluginBase` 元素也很重要。此设置与 `<Private>false</Private>` 的效果相同，但适用于 `PluginBase` 项目或它的某个依赖项可能包括的包引用。

因为 `HelloPlugin` 项目已完成，所以应该更新 `AppWithPlugin` 项目，以确认可以找到 `HelloPlugin` 插件的位置。在 `// Paths to plugins to load` 注释之后，添加 `@"HelloPlugin\bin\Debug\netcoreapp3.0\HelloPlugin.dll"` 作为 `pluginPaths` 数组的元素。

具有库依赖项的插件

几乎所有插件都比简单的“Hello World”更复杂，而且许多插件都具有其他库上的依赖项。示例中的 `JsonPlugin` 和 `OldJson` 插件项目显示了具有 `Newtonsoft.Json` 上的 NuGet 包依赖项的两个插件示例。项目文件本身没有关于项目引用的任何特殊信息，并且（在将插件路径添加到 `pluginPaths` 数组之后）即使是在 `AppWithPlugin` 应用的同一执行中运行，插件也能完美运行。但是，这些项目不会将引用的程序集复制到它们的输出目录中，因此需要将这些程序集显示在用户的计算机上，以便插件能够正常工作。解决此问题有两种方法。第一种是使用 `dotnet publish` 命令发布类库。或者，如果希望能够将 `dotnet build` 的输出用于插件，可以在插件的项目文件中的 `<CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>` 标记之间添加 `<PropertyGroup>` 属性。有关示例，请参阅 `XcopyablePlugin` 插件项目。

示例中的其他示例

可以在 `dotnet/samples` 存储库中找到本教程的完整源代码。完成的示例包括 `AssemblyDependencyResolver` 行为的一些其他示例。例如，`AssemblyDependencyResolver` 对象还可以解析本机库和 NuGet 包中所包含的已本地化的附属程序集。示例存储库中的 `UVPlugin` 和 `FrenchPlugin` 演示了这些方案。

从 NuGet 包引用插件接口

假设存在应用 A，它具有 NuGet 包（名为 `A.PluginBase`）中定义的插件接口。如何在插件项目中正确引用包？对于项目引用，使用项目文件的 `<Private>false</Private>` 元素上的 `ProjectReference` 元数据会阻止将 dll 复制到输出。

若要正确引用 `A.PluginBase` 包，应将项目文件中的 `<PackageReference>` 元素更改为以下内容：

```
<PackageReference Include="A.PluginBase" Version="1.0.0">
<ExcludeAssets>runtime</ExcludeAssets>
</PackageReference>
```

此操作会阻止将 `A.PluginBase` 程序集复制到插件的输出目录，并确保插件将使用 A 版本的 `A.PluginBase`。

插件目标框架建议

因为插件依赖项加载使用 `.deps.json` 文件，所以存在一个与插件的目标框架相关的问题。具体来说，插件应该以运行时为目标，比如 .NET Core 3.0，而不是某一版本的 .NET Standard。`.deps.json` 文件基于项目所针对的框架生成，而且由于许多与 .NET Standard 兼容的包提供了用于针对 .NET Standard 进行生成的引用程序集和用于特定运行时的实现程序集，因此 `.deps.json` 可能无法正确查看实现程序集，或者它可能会获取 .NET Standard 版本的程序集，而不是期望的 .NET Core 版本的程序集。

插件框架引用

插件当前无法向该过程引入新的框架。例如，无法将使用 `Microsoft.AspNetCore.App` 框架的插件加载到只使用根 `Microsoft.NETCore.App` 框架的应用程序中。主机应用程序必须声明对插件所需的全部框架的引用。



向 COM 公开 .NET Core 组件

2020/3/19 • [Edit Online](#)

在 .NET Core 中，与 .NET Framework 相比，向 COM 公开 .NET 对象的过程已明显简化。下面的过程将引导你如何向 COM 公开类。本教程介绍了如何：

- 从 .NET Core 向 COM 公开类。
- 生成 COM 服务器作为构建 .NET Core 库的一部分。
- 自动为无注册表 COM 生成并行服务器清单。

先决条件

- 安装 [.NET Core 3.0 SDK](#) 或更高版本。

创建库

第一步是创建库。

- 创建新文件夹，并在该文件夹中运行以下命令：

```
dotnet new classlib
```

- 打开 `Class1.cs`。

- 将 `using System.Runtime.InteropServices;` 添加到文件顶部。

- 创建名为 `IIServer` 的接口。例如：

```
using System;
using System.Runtime.InteropServices;

[ComVisible(true)]
[Guid(ContractGuids.ServerInterface)]
[InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
public interface IIServer
{
    /// <summary>
    /// Compute the value of the constant Pi.
    /// </summary>
    double ComputePi();
}
```

- 将 `[Guid("<IID>")]` 属性添加到接口，包含要实现的 COM 接口的接口 GUID。例如

`[Guid("fe103d6e-e71b-414c-80bf-982f18f6c1c7")]`。请注意，此 GUID 必须唯一，因为它是 COM 的此接口的唯一标识符。在 Visual Studio 中，可通过转到“工具”>“创建 GUID”以打开“创建 GUID”工具来生成 GUID。

- 将 `[InterfaceType]` 属性添加到接口，并指定接口应实现的基本 COM 接口。

- 创建用于实现 `IIServer` 的名为 `Server` 的类。

- 将 `[Guid("<CLSID>")]` 属性添加到类，包含要实现的 COM 类的类标识符 GUID。例如 `[Guid("9f35b6f5-2c05-4e7f-93aa-ee087f6e7ab6")]`。与接口 GUID 一样，此 GUID 必须唯一，因为它是 COM 的此接口的唯一标识符。

9. 将 `[ComVisible(true)]` 属性添加到接口和类。

IMPORTANT

与 .NET Framework 不同, .NET Core 要求指定想要通过 COM 激活的任何类的 CLSID。

生成 COM 主机

1. 打开 `.csproj` 项目文件并在 `<PropertyGroup></PropertyGroup>` 标记中添加
`<EnableComHosting>true</EnableComHosting>`。
2. 生成项目。

生成的输出将具有 `ProjectName.dll`、`ProjectName.deps.json`、`ProjectName.runtimeconfig.json` 和
`ProjectName.comhost.dll` 文件。

为 COM 注册 COM 主机

打开提升的命令提示符, 然后运行 `regsvr32 ProjectName.comhost.dll`。这将使用 COM 注册所有公开的 .NET 对象。

启用 RegFree COM

1. 打开 `.csproj` 项目文件并在 `<PropertyGroup></PropertyGroup>` 标记中添加
`<EnableRegFreeCom>true</EnableRegFreeCom>`。
2. 生成项目。

生成的输出现在还将具有 `ProjectName.X.manifest` 文件。此文件是用于无注册表的 COM 的并行清单。

示例

GitHub 上的 `dotnet/samples` 存储库中有一个正常运行的 [COM 服务器示例](#)。

附加说明

与 .NET Framework 不同, .NET Core 不支持从 .NET Core 程序集生成 COM 类型库 (TLB)。本指南旨在说明如何为 COM 接口的本机声明手动编写 IDL 文件或 C/C++ 标头。

此外, 将 .NET Framework 和 .NET Core 同时加载到同一进程具有诊断限制。主要限制是调试托管组件, 因为不能同时调试 .NET Framework 和 .NET Core。此外, 这两个运行时实例不共享托管程序集。这意味着无法在两个运行时之间共享实际的 .NET 类型, 所有交互必须仅限于公开的 COM 接口协定。

包、元包和框架

2020/3/18 • [Edit Online](#)

.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，.NET Core 应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是作为包进行创建的，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。

包

.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内。

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。

```
<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。元包通过使这组包成为依赖项来表示这组包。通过指定一个框架，元包能够有选择地为这组包发布一个框架。

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)。建立 `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。建立 `.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 mscorelib 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。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。

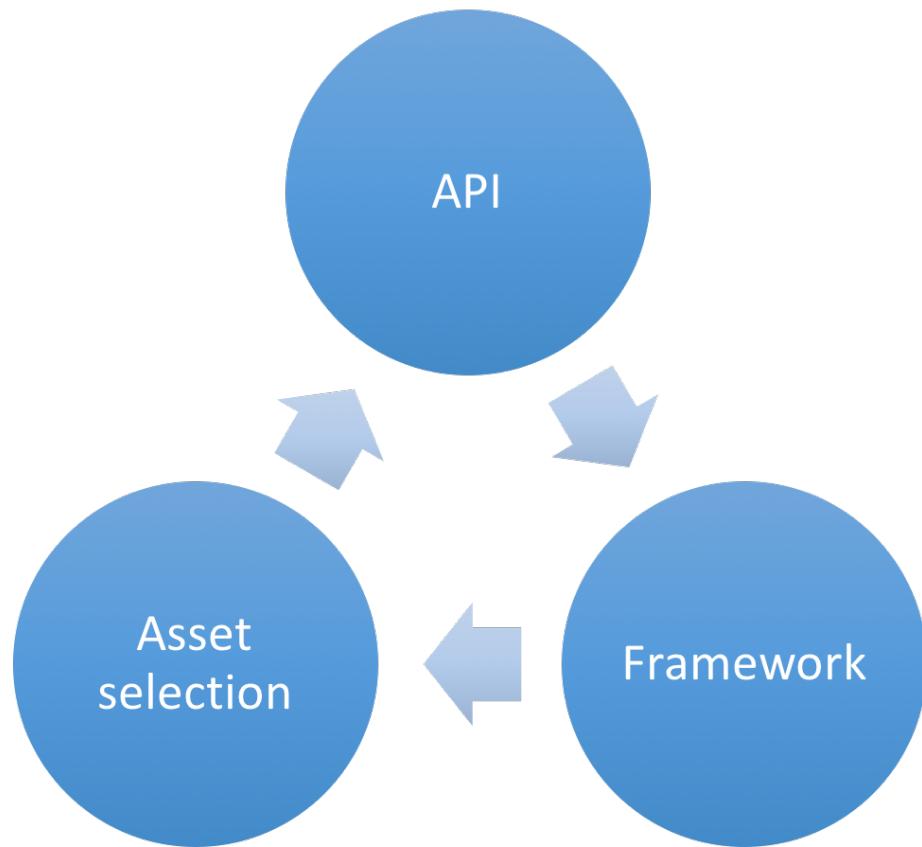
基于包的框架

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。如果构建的库将在多个运行时中运行，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`。

```
<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。

```
<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。（为与前面的示例对称，这里使用 v1.3.0，但实际上它并不存在。）

.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。

由 `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。

在持续集成 (CI) 中使用 .NET Core SDK 和工具

2020/3/18 • [Edit Online](#)

本文档概述了如何在生成服务器上使用 .NET Core SDK 及其工具。.NET Core 工具集既可以交互方式运行(当开发者在命令提示符处键入命令时),也可以自动运行(当持续集成 (CI) 服务器运行生成脚本时)。命令、选项、输入和输出都相同,可通过提供的唯一内容来获取用于生成应用的工具和系统。本文档重点介绍了 CI 工具获取方案,并提供了有关如何设计和构建生成脚本的建议。

CI 生成服务器的安装选项

使用本机安装程序

本机安装程序适用于 macOS、Linux 和 Windows。安装程序需要拥有对生成服务器的管理员 (sudo) 访问权限。使用本机安装程序的优势在于,可以安装运行工具所需的全部本机依赖项。本机安装程序还可以在整个系统内安装 SDK。

macOS 用户应使用 PKG 安装程序。在 Linux 上,可选择使用基于源的包管理器(如用于 Ubuntu 的 apt-get 或用于 CentOS 的 yum),也可以选择使用包本身(即 DEB 或 RPM)。在 Windows 上,使用 MSI 安装程序。

有关最新的稳定二进制文件,请参阅 [.NET 下载](#)。若要使用最新(但可能不稳定)的预览版工具,请使用 [dotnet/core-sdk GitHub 存储库](#) 中提供的链接。对于 Linux 发行版本,可以使用 `tar.gz` 存档(亦称为 `tarballs`) ;使用存档中的安装脚本来安装 .NET Core。

使用安装程序脚本

使用安装程序脚本,可以在生成服务器上执行非管理员安装,并能轻松实现自动化,以便获取工具。安装程序脚本负责下载并将工具提取到默认或指定位置,以供使用。还可以指定要安装的工具版本,以及是要安装整个 SDK,还是仅安装共享运行时。

安装程序脚本在开始生成时自动运行,以提取和安装相应版本的 SDK。相应版本是指生成项目所需的任意 SDK 版本。使用安装程序脚本,可以在服务器的本地目录中安装 SDK,并能从安装位置运行工具,还可以在生成后进行清理(或让 CI 服务进行清理)。这样,可以封装和隔离整个生成进程。有关安装脚本参考,请参阅 [dotnet-install](#) 一文。

NOTE

Azure DevOps Services

使用安装程序脚本时,不会自动安装本机依赖项。如果操作系统没有本机依赖项,必须手动安装。有关详细信息,请参阅 [.NET Core 依赖项和要求](#)。

CI 安装示例

此部分介绍了如何使用 PowerShell 或 bash 脚本进行手动安装,同时还介绍了多个服务型软件 (SaaS) CI 解决方案。涵盖的 SaaS CI 解决方案包括 [Travis CI](#)、[AppVeyor](#) 和 [Azure Pipelines](#)。

手动安装

每个 SaaS 服务都有自己的生成进程创建和配置方法。如果使用与所列不同的 SaaS 解决方案,或需要超越预封装支持范围的自定义设置,至少必须执行一些手动配置。

一般来说,手动安装需要获取一个版本的工具(或最新每日版工具),再运行生成脚本。可以使用 PowerShell 或 bash 脚本安排 .NET Core 命令,也可以使用概述生成进程的项目文件。[业务流程部分](#) 详细介绍了这些选项。

创建执行手动 CI 生成服务器安装的脚本后,在开发计算机上使用它来生成本地代码以供测试。确认此脚本可以在

本地正常运行后，将它部署到 CI 生成服务器。下面展示了相对简单的 PowerShell 脚本，以说明如何获取 .NET Core SDK，并将它安装到 Windows 生成服务器上：

```
$ErrorActionPreference="Stop"
$ProgressPreference="SilentlyContinue"

# $LocalDotnet is the path to the locally-installed SDK to ensure the
#   correct version of the tools are executed.
$LocalDotnet=""
# $InstallDir and $CliVersion variables can come from options to the
#   script.
$InstallDir = "./cli-tools"
$CliVersion = "1.0.1"

# Test the path provided by $InstallDir to confirm it exists. If it
#   does, it's removed. This is not strictly required, but it's a
#   good way to reset the environment.
if (Test-Path $InstallDir)
{
    rm -Recurse $InstallDir
}
New-Item -Type "directory" -Path $InstallDir

Write-Host "Downloading the CLI installer..."

# Use the Invoke-WebRequest PowerShell cmdlet to obtain the
#   installation script and save it into the installation directory.
Invoke-WebRequest `

    -Uri "https://dot.net/v1/dotnet-install.ps1" `

    -OutFile "$InstallDir/dotnet-install.ps1"

Write-Host "Installing the CLI requested version ($CliVersion) ..."

# Install the SDK of the version specified in $CliVersion into the
#   specified location ($InstallDir).
& $InstallDir/dotnet-install.ps1 -Version $CliVersion `

    -InstallDir $InstallDir

Write-Host "Downloading and installation of the SDK is complete."

# $LocalDotnet holds the path to dotnet.exe for future use by the
#   script.
$LocalDotnet = "$InstallDir/dotnet"

# Run the build process now. Implement your build script here.
```

在此脚本末尾提供生成进程的实现代码。此脚本先获取工具，再执行生成进程。对于 UNIX 计算机，下面的 bash 脚本以类似方式执行 PowerShell 脚本中所述的操作：

```

#!/bin/bash
INSTALLDIR="cli-tools"
CLI_VERSION=1.0.1
DOWNLOADER=$(which curl)
if [ -d "$INSTALLDIR" ]
then
    rm -rf "$INSTALLDIR"
fi
mkdir -p "$INSTALLDIR"
echo Downloading the CLI installer.
$DOWNLOADER https://dot.net/v1/dotnet-install.sh > "$INSTALLDIR/dotnet-install.sh"
chmod +x "$INSTALLDIR/dotnet-install.sh"
echo Installing the CLI requested version $CLI_VERSION. Please wait, installation may take a few minutes.
"$INSTALLDIR/dotnet-install.sh" --install-dir "$INSTALLDIR" --version $CLI_VERSION
if [ $? -ne 0 ]
then
    echo Download of $CLI_VERSION version of the CLI failed. Exiting now.
    exit 0
fi
echo The CLI has been installed.
LOCALDOTNET="$INSTALLDIR/dotnet"
# Run the build process now. Implement your build script here.

```

Travis CI

可以将 [Travis CI](#) 配置为使用 `csharp` 语言和 `dotnet` 键安装 .NET Core SDK。有关详细信息，请参阅 [Travis CI 官方文档](#)[生成 C#、F# 或 Visual Basic 项目](#)。请注意，访问 Travis CI 信息时，社区维护的 `language: csharp` 语言标识符适用于所有 .NET 语言，包括 F# 和 Mono。

Travis CI 可同时在生成矩阵中运行 macOS 和 Linux 作业。在生成矩阵中，可以指定运行时、环境和排除项/包含项的组合，从而涵盖应用的生成组合。有关详细信息，请参阅 [Travis CI 文档中的自定义生成一文](#)。基于 MSBuild 的工具在包中添加 LTS (1.0.x) 和最新 (1.1.x) 运行时；因此，通过安装 SDK，可以收到执行生成所需的一切。

AppVeyor

[AppVeyor](#) 使用 `visual studio 2017` 生成辅助角色映像安装 .NET Core 1.0.1 SDK。提供具有不同版本的 .NET Core SDK 的其他生成映像。有关详细信息，请参阅 [AppVeyor 文档中的 appveyor.yml 示例](#)和[生成辅助角色映像一文](#)。

.NET Core SDK 二进制文件通过安装脚本下载并解压缩到子目录，再添加到 `PATH` 环境变量。添加生成矩阵可以运行包含多个版本 .NET Core SDK 的集成测试：

```

environment:
matrix:
  - CLI_VERSION: 1.0.1
  - CLI_VERSION: Latest

install:
  # See appveyor.yml example for install script

```

Azure DevOps Services

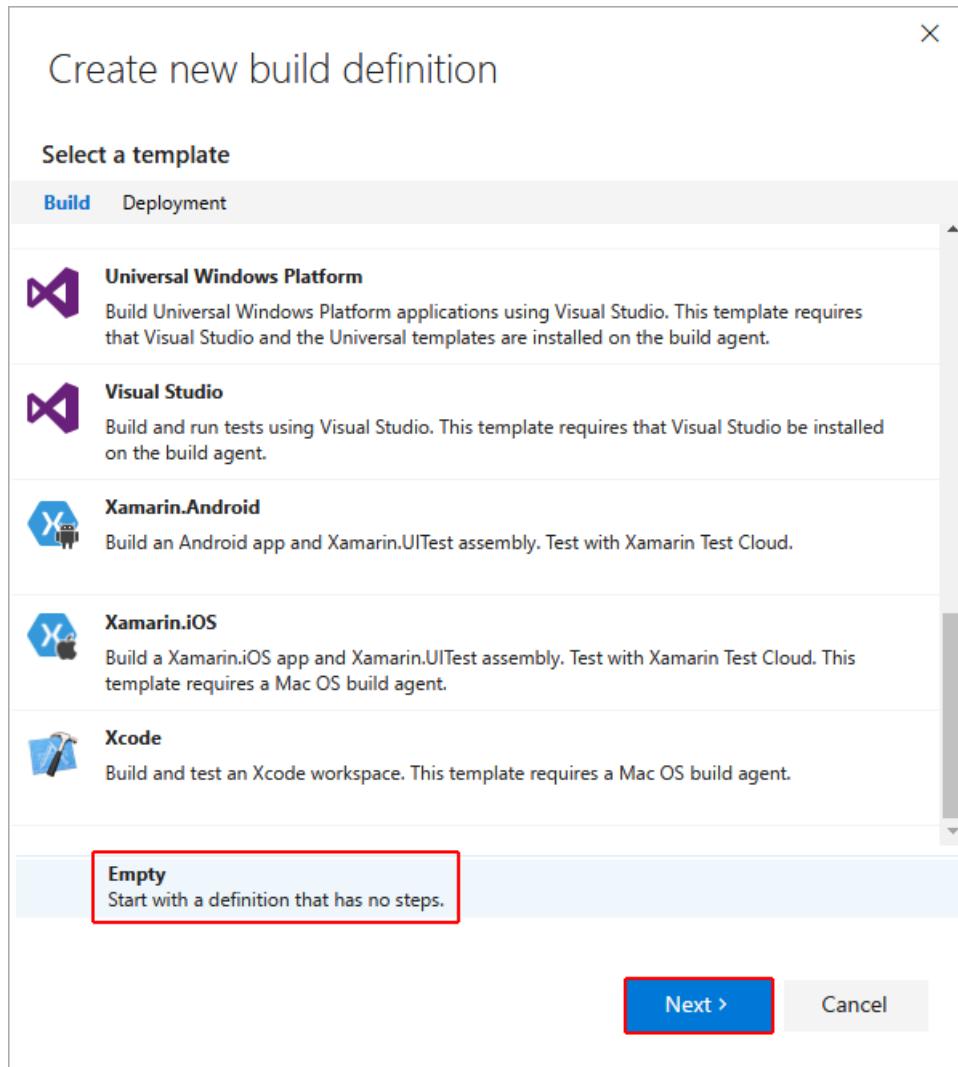
将 Azure DevOps Services 配置为使用以下方法之一生成 .NET Core 项目：

1. 使用命令运行[手动安装步骤](#)中的脚本。
2. 创建包含多个 Azure DevOps Services 内置生成任务(这些任务被配置为使用 .NET Core 工具)的生成。

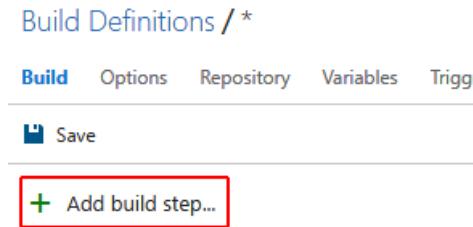
这两种均为有效解决方案。使用手动安装脚本，可以控制收到的工具版本，因为工具是作为生成的一部分进行下载。此生成是通过必须创建的脚本进行运行。本文仅涉及手动选项。有关使用 Azure DevOps Services 生成任务撰写生成的详细信息，请参阅 [Azure Pipelines](#) 一文。

若要在 Azure DevOps Services 中使用手动安装脚本，请新建生成定义，并指定要对生成步骤运行的脚本。为此，请使用 Azure DevOps Services 用户界面：

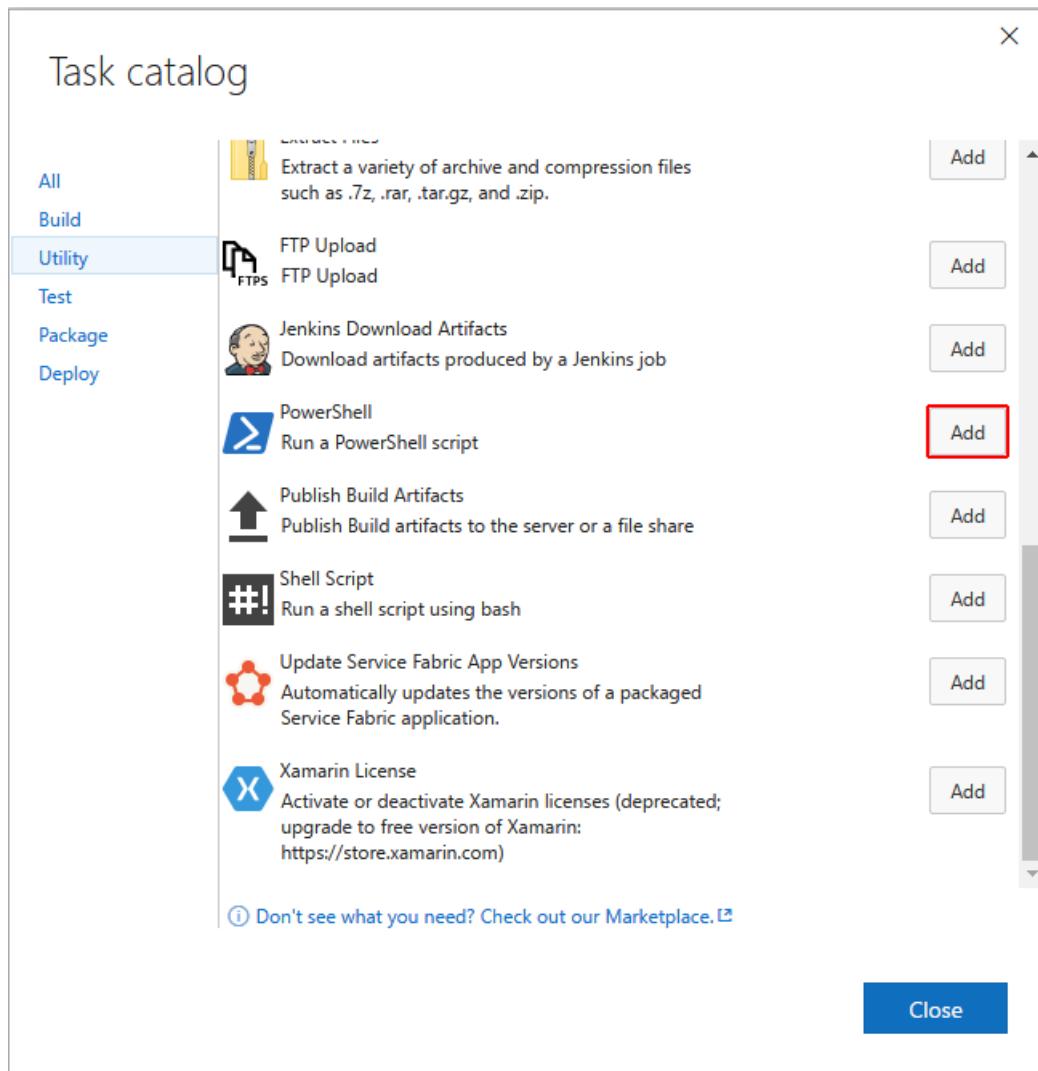
- 首先，新建生成定义。到达可以定义要创建的生成类型的屏幕后，选择“空”选项。



- 配置要生成的存储库后，将转到生成定义。选择“添加生成步骤”：



- 此时，系统会显示“任务目录”。此目录包含在生成中使用的任务。由于已有脚本，因此选择“PowerShell:运行 PowerShell 脚本”旁边的“添加”按钮。



4. 配置生成步骤。从要生成的存储库中添加脚本：

The screenshot shows the 'PowerShell Script' build step configuration. The 'File Path' field is highlighted with a red box.

Configuration details:

- Type: PowerShell Script
- Version: 1.*
- File Path: (highlighted with a red box)
- Arguments:
- Advanced Options:
 - Enabled: checked
 - Continue on error: unchecked
 - Always run: unchecked
 - Timeout: 0

安排生成

本文档的大部分内容介绍了如何获取 .NET Core 工具和配置各种 CI 服务，并未介绍如何安排或实际生成 .NET Core 代码。具体如何构建生成进程取决于许多因素，我们无法在本文中笼统概述。有关使用每种技术安排生成的详细信息，请浏览 [Travis CI](#)、[AppVeyor](#)、和 [Azure Pipelines](#) 文档集中提供的资源和示例。

使用 .NET Core 工具构建 .NET Core 代码生成进程的两种常规方法是，直接使用 MSBuild 或使用 .NET Core 命令行命令。应采用哪种方法取决于对方法的熟悉程度和复杂性取舍。使用 MSBuild，可以将生成进程表达为任务和目标，但需要学习 MSBuild 项目文件语法，这增加了复杂性。使用 .NET Core 命令行工具可能更为简单，但需要在 `bash` 或 `PowerShell` 等脚本语言中编写业务流程逻辑。

另请参阅

- [.NET 下载 - Linux](#)