

教程：使 .NET Core 应用程序容器化

2020/01/09 • 👤 👤

本文内容

[先决条件](#)

[创建 .Net Core 应用](#)

[发布 .Net Core 应用](#)

[创建 Dockerfile](#)

[创建容器](#)

[重要命令](#)

[清理资源](#)

[后续步骤](#)

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。

💡 提示

如果要使用现有的 ASP.NET Core 应用，请参阅教程[了解如何容器化 ASP.NET Core 应用](#)。

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：

.NET Core CLI	复制
<pre>dotnet new console -o app -n myapp</pre>	

文件夹树将如下所示：


	复制
<pre>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</pre>	

`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：

console	复制
<pre>> dotnet run Hello World!</pre>	

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：

C#


 复制

```
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：

C#


 复制

```
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：

console

 复制

```
> 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。

① 备注

-- 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：

.NET Core CLI

复制

```
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。

console

复制

```
> 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。

bash

复制

```
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

Dockerfile	复制
<code>FROM mcr.microsoft.com/dotnet/core/aspnet:3.1</code>	

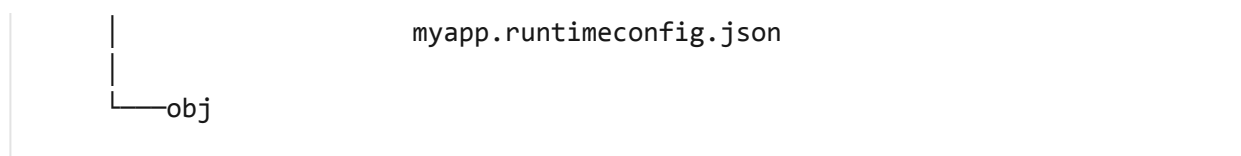
- .NET Core 运行时

Dockerfile	复制
<code>FROM mcr.microsoft.com/dotnet/core/runtime:3.1</code>	

`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，删掉了一些更深级别的文件和文件夹：

	复制
<pre>docker-working ├── Dockerfile └── app ├── myapp.csproj ├── Program.cs └── bin ├── Release │ ├── netcoreapp3.1 │ │ └── publish │ │ ├── myapp.deps.json │ │ ├── myapp.exe │ │ ├── myapp.dll │ │ └── myapp.pdb</pre>	



在终端中运行以下命令：

console	复制
<pre>docker build -t myimage -f Dockerfile .</pre>	

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：

console	复制																				
<pre>> docker images</pre> <table border="1"><thead><tr><th>REPOSITORY</th><th>SIZE</th><th>TAG</th><th>IMAGE ID</th></tr></thead><tbody><tr><td>myimage</td><td></td><td>latest</td><td>38db0eb8f648</td></tr><tr><td>4 weeks ago</td><td>346MB</td><td></td><td></td></tr><tr><td>mcr.microsoft.com/dotnet/core/aspnet</td><td></td><td>3.1</td><td>38db0eb8f648</td></tr><tr><td>4 weeks ago</td><td>346MB</td><td></td><td></td></tr></tbody></table>		REPOSITORY	SIZE	TAG	IMAGE ID	myimage		latest	38db0eb8f648	4 weeks ago	346MB			mcr.microsoft.com/dotnet/core/aspnet		3.1	38db0eb8f648	4 weeks ago	346MB		
REPOSITORY	SIZE	TAG	IMAGE ID																		
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。

Dockerfile	复制
<pre>COPY app/bin/Release/netcoreapp3.1/publish/ app/ ENTRYPOINT ["dotnet", "app/myapp.dll"]</pre>	

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 。

console	复制
---------	----

```
> 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。

console	复制
<pre>> docker create myimage ceda87b219a4e55e9ad5d833ee1a7ea4da21b5ea7ce5a7d08f3051152e784944</pre>	

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：

console	复制
<pre>> docker ps -a CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES ceda87b219a4 myimage "dotnet app/myapp.dll" 4 seconds ago Created gallant_lehmann</pre>	

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：

console

复制

```
> 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：

console

复制

```
> 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` 击键用于从正在运行的容器中分离出来。 此击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。

console

复制

```
> 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。

console

 复制

```
> 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。

console

 复制

```
> 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：

console

 复制

```
> 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。验证它是否不存在：

console				复制
<pre>> docker ps -a</pre>				
CONTAINER ID	IMAGE	COMMAND	CREATED	
STATUS	PORTS	NAMES		

更改 ENTRYPOINT

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。

console				复制
<pre>> docker run -it --rm --entrypoint "cmd.exe" myimage</pre>				
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。

```
bash
```

[复制](#)

```
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

```
console
```

[复制](#)

```
> 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。

```
console
```


[复制](#)

```
> docker stop CONTAINER_NAME
```

3. 删除容器

console	 复制
<pre>> docker rm CONTAINER_NAME</pre>	

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。

console	 复制
<pre>docker rmi myimage:latest docker rmi mcr.microsoft.com/dotnet/core/aspnet:3.1</pre>	

使用 `docker images` 命令来列出已安装的映像。

ⓘ 备注

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](#)

此页面有帮助吗？

 是  否