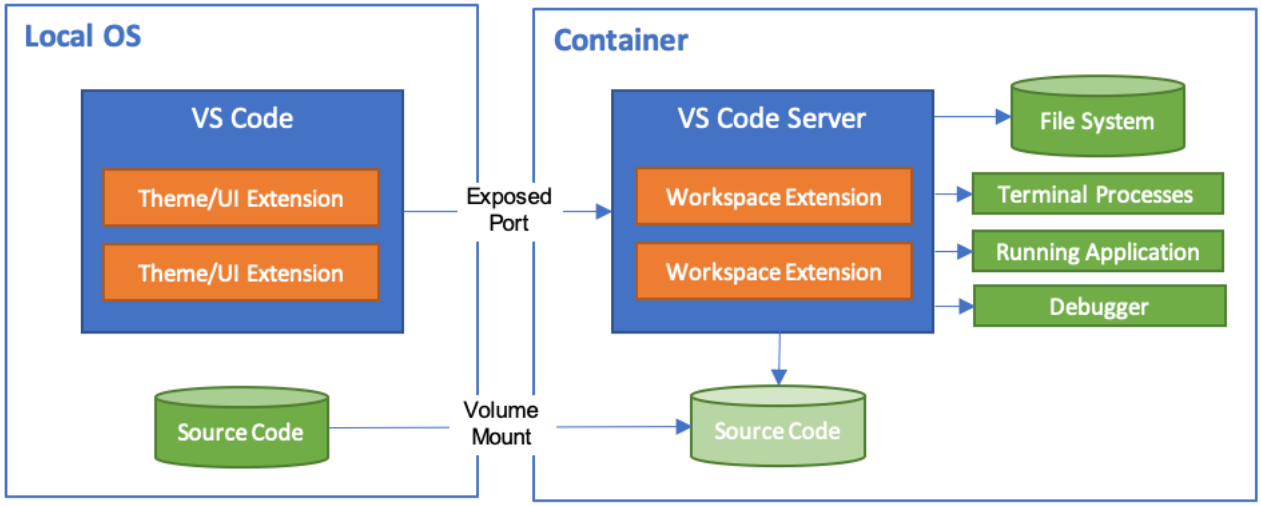
TOPICS 货柜

# 在容器内开发

(https://github.com/Microsoft/vscode-docs/blob/master/docs/remote/containers.md)

在**Visual Studio代码远程-集装箱**扩展可以让你使用一个码头工人的容器 (https://docker.com)作为一个全功能的开发环境。它允许您打开容器内（或装入容器）的任何文件夹，并利用Visual Studio Code的全部功能集。一个devcontainer.json文件在您的项目告诉VS代码是如何访问（或创建）一个**开发容器**有一个定义良好的工具和运行时栈。该容器可用于运行应用程序或用于使用代码库的沙盒工具，库或运行时。

从本地文件系统挂载工作区文件，或者将其复制或克隆到容器中。扩展安装在容器中并在其中运行，可以在其中完全访问工具，平台和文件系统。这意味着您只需连接到另一个容器即可无缝切换整个开发环境。




这样，**无论您的工具（或代码）位于何处**，VS Code都可以提供**本地质量的开发体验** -包括完整的IntelliSense（完成功能），代码导航和调试。

## 入门

### 系统要求

**本地**：

- **Windows**： Windows 10 Pro / Enterprise上的Docker Desktop (https://www.docker.com/products/docker-desktop) 2.0+。Windows 10 Home（2004+）需要Docker Desktop 2.3+和WSL2后端 (https://aka.ms/vscode-remote/containers/docker-wsl2)。（不支持Docker Toolbox。不支持Windows容器映像。）
- **macOS**： Docker Desktop (https://www.docker.com/products/docker-desktop) 2.0+。
- **Linux**：Docker CE / EE (https://docs.docker.com/install/#supported-platforms) 18.06+和Docker Compose (https://docs.docker.com/compose/install) 1.21+。（不支持Ubuntu snap软件包。）

**容器**：

- x86_64 / ARMv7l（AArch32）/ ARMv8l（AArch64）Debian 9以上版本，Ubuntu 16.04以上版本，CentOS / RHEL 7以上版本
- x86_64 Alpine Linux 3.8+

其他 `glibc` 基于Linux的容器如果需要先决条件，（/docs/remote/linux)则可以使用 (/docs/remote/linux)。

虽然提供了ARMv7l（AArch32），ARMv8l（AArch64）和 `musl` 基于Alpine Linux的支持，但由于在扩展中使用 `glibc` 或 `x86` 编译了本机代码，因此这些设备上安装的某些扩展可能无法正常工作。有关详细信息，请参见《使用Linux (/docs/remote/linux)进行远程开发》一 (/docs/remote/linux)文。

虽然需要Docker CLI，但是如果您使用远程Docker主机 (/docs/remote/containers-advanced#_developing-inside-a-container-on-a-remote-docker-host)，则Docker守护程序/服务不需要在本地运行。

### 安装

首先，请按照下列步骤操作：

1. 为您的操作系统安装和配置Docker (https://www.docker.com/get-started)。

   **Windows / macOS**：

   1. 安装适用于Windows / Mac的Docker桌面 (https://www.docker.com/products/docker-desktop)。
   2. 如果未在Windows上使用WSL2，请右键单击Docker任务栏项目，选择“**设置**” /“**首选项**”，然后使用保留源代码的任何位置更新“ **资源**”>“**文件共享**”。请参阅提示和技巧 (/docs/remote/troubleshooting#_container-tips)以进行故障排除。
   3. 启用Windows WSL2后端 (https://aka.ms/vscode-remote/containers/docker-wsl2)：右键单击Docker任务栏项，然后选择**设置**。在“ **资源**”>“ **WSL集成**”下，选中“ **使用基于WSL2的引擎**”并确认您的分发已启用。

   **Linux**：

    1. 请遵循Docker CE / EE (https://docs.docker.com/install/#supported-platforms)的官方安装说明进行分发 (https://docs.docker.com/install/#supported-platforms)。如果您使用的是Docker Compose，请同样遵循Docker Compose的说明 (https://docs.docker.com/compose/install/)。
    2. `docker` 使用终端将用户添加到组中，以运行： `sudo usermod -aG docker $USER`
    3. 退出并重新登录，以使更改生效。
2. 安装Visual Studio Code (https://code.visualstudio.com/)或Visual Studio Code Insiders (https://code.visualstudio.com/insiders/)。
3. 安装远程开发扩展包 (https://aka.ms/vscode-remote/download/extension)。

**使用Git吗？** 这里有两个要考虑的技巧：

- 如果您在Windows本地和容器内都使用相同的存储库，请确保设置一致的行尾。有关详细信息，请参见提示和技巧 (/docs/remote/troubleshooting#_resolving-git-line-ending-issues-in-containers-resulting-in-many-modified-files)。
- 如果您使用Git凭证管理器进行克隆，则您的容器应该已经可以访问您的凭证了！如果您使用SSH密钥，则也可以选择共享它们。有关详细信息，请参见与容器共享Git凭据。

**下一步：** 远程-容器扩展支持两种主要的操作模型：

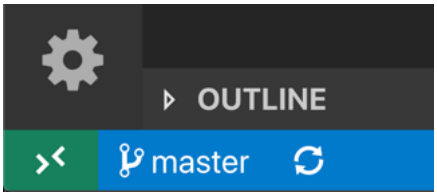
- 您可以将容器用作全职开发环境。
- 您可以附加到正在运行的容器进行检查。

我们将首先介绍使用容器作为您的全职开发环境。

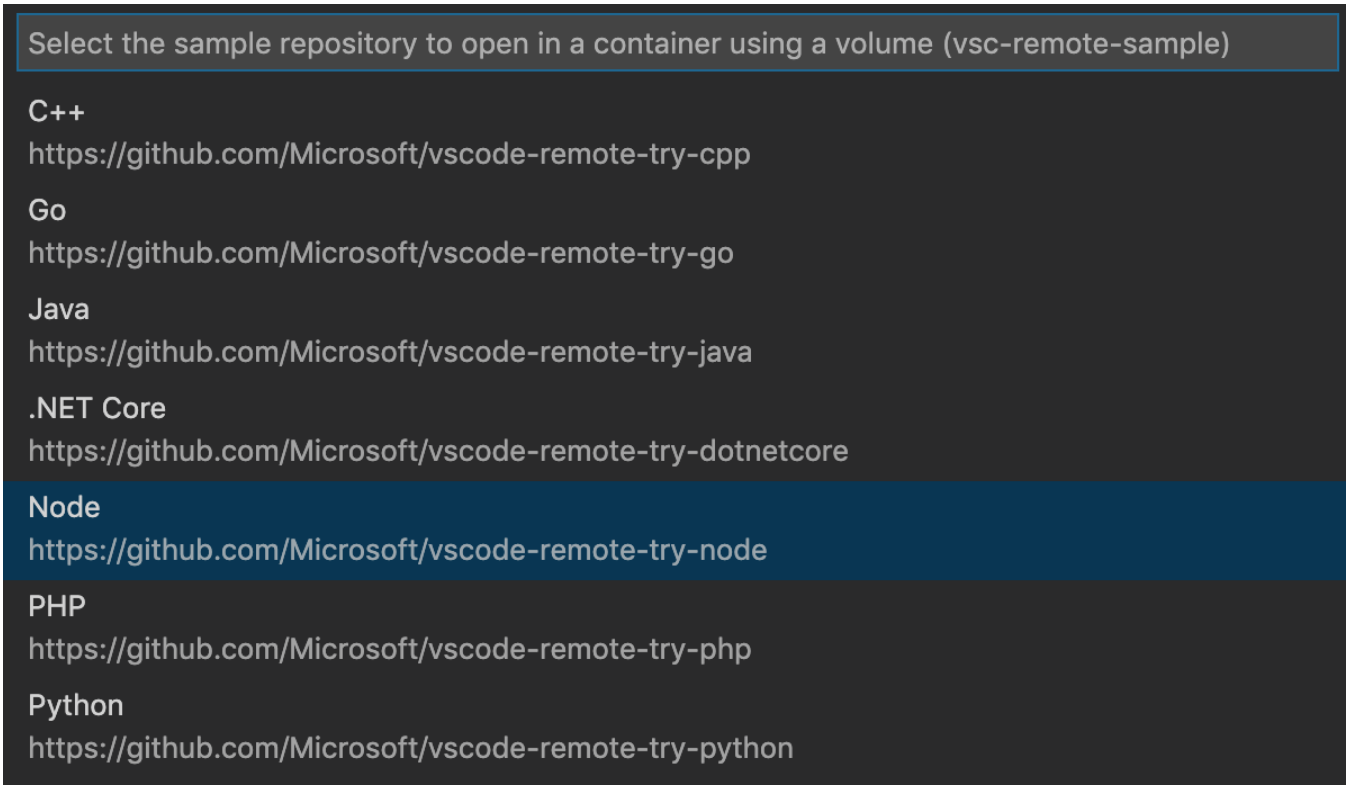
## 快速入门：尝试开发容器

让我们从一个示例项目开始尝试。

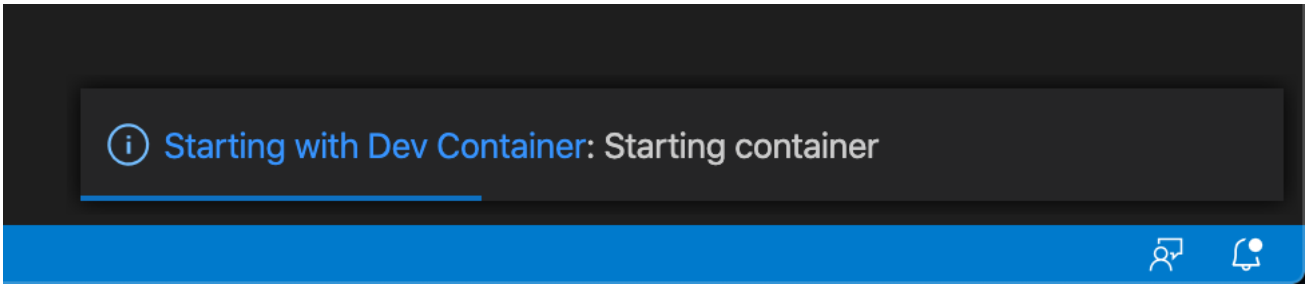
1. 启动VS Code，然后在新窗口中单击左下角的快速操作状态栏项目。




2. 从出现的命令列表中选择“ **远程容器：尝试样品...”，然后**从列表中选择样品。




3. 然后将重新加载该窗口，但是由于该容器尚不存在，因此VS Code将创建一个并将样本存储库克隆到一个隔离的容器卷中 (https://docs.docker.com/storage/volumes/)。这可能需要一些时间，并且进度通知将提供状态更新。幸运的是，下次打开文件夹时无需执行此步骤，因为该容器已经存在。




4. 生成容器后，VS Code会自动连接到该容器，并将项目文件夹从本地文件系统映射到容器中。在克隆的存储库中查看“ **尝试的事情**”部分， `README.md` 以了解下一步的工作。

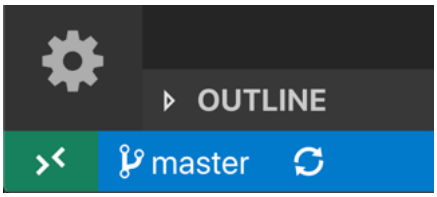
您可能想知道存储库源代码位于何处。在这种情况下，源代码存储在无法从本地操作系统直接访问的容器卷 (https://docs.docker.com/storage/volumes/)中。但是，您也可以通过容器使用本地文件系统中的内容！接下来我们将介绍。

## 快速入门：打开容器中的现有文件夹

接下来，我们将介绍如何使用文件系统上的现有源代码为现有项目设置dev容器，以将其用作全职开发环境。跟着这些步骤：

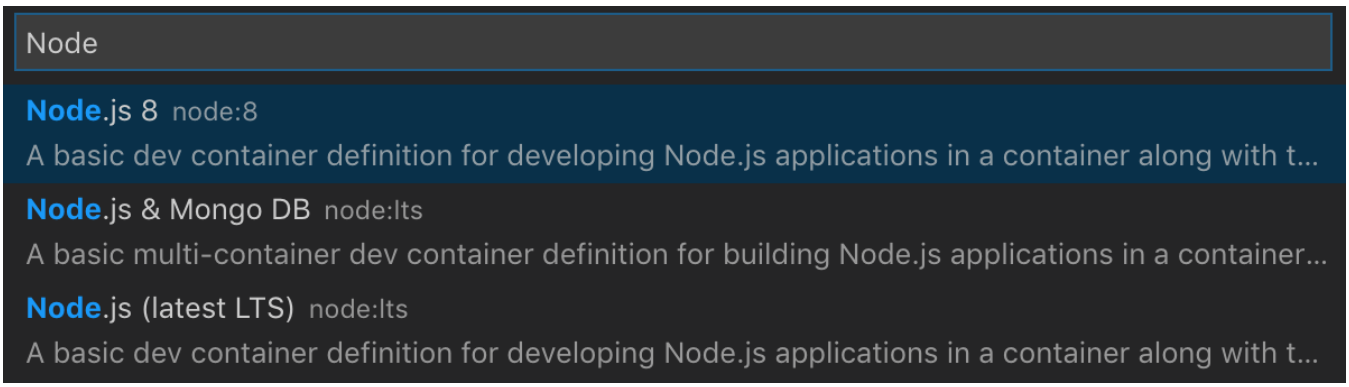
1. 启动VS Code，运行命令面板（ `F1` ）中的“ **远程容器：在容器中打开文件夹**...”命令或快速操作状态栏项目，然后选择要为其设置容器的项目文件夹。

> **提示：**如果要在打开文件夹之前编辑容器的内容或设置，则可以运行“ **远程容器：添加开发容器配置文件**...”。




2. 现在为您的开发容器选择一个起点。您可以从可过滤列表中选择基本**dev容器定义**，或者如果所选文件夹中存在Dockerfile (https://docs.docker.com/engine/reference/builder/)或Docker Compose文件 (https://docs.docker.com/compose/compose-file/#compose-file-structure-and-examples)，则可以使用现有文件 (https://docs.docker.com/compose/compose-file/#compose-file-structure-and-examples)。

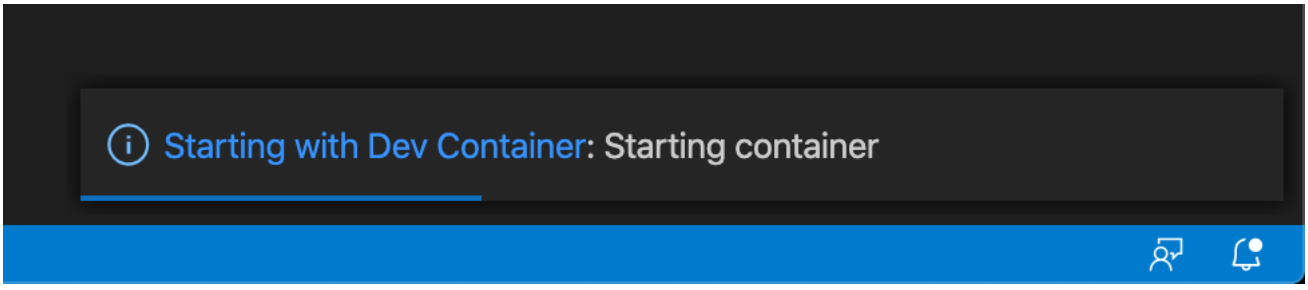
> **注意：**使用Alpine Linux容器时，由于 `glibc` 扩展内部代码的依赖性，某些扩展可能无法正常工作。




该列表将根据您打开的文件夹的内容自动排序。请注意，显示的dev容器定义来自vscode-dev-containers存储库 (https://aka.ms/vscode-dev-containers)。您可以浏览该 `containers` 存储库的文件夹以查看每个定义的内容。

3. 选择了容器的起点之后，VS Code会将dev容器配置文件添加到您的项目（ `.devcontainer/devcontainer.json` ）。

4. VS Code窗口将重新加载并开始构建dev容器。进度通知提供状态更新。您只需要在首次打开容器时就构建一个容器。首次成功构建后打开文件夹将更快。




5. 构建完成后，VS Code将自动连接到容器。

现在，您可以像在本地打开项目时一样，在VS Code中与项目进行交互。从现在开始，当您打开项目文件夹时，VS Code将自动选择并重用您的dev容器配置。

> **提示：是否**要使用远程Docker主机？有关设置的详细信息，请参见“ 高级容器”文章 (/docs/remote/containers-advanced#_developing-inside-a-container-on-a-remote-docker-host)。

虽然使用此方法将 (https://docs.docker.com/storage/bind-mounts/)本地文件系统绑定安装 (https://docs.docker.com/storage/bind-mounts/)到容器很方便，但在Windows和macOS上确实有一些性能开销。有一些技巧 (/docs/remote/containers-advanced#_improving-container-disk-performance)，你可以申请，以提高磁盘性能，也可以使用分离容器容积打开容器存储库来代替。

### 在Windows上的容器中打开WSL2文件夹

如果您正在使用Windows Subsystem for Linux v2（WSL2），(https://docs.microsoft.com/en-us/windows/wsl/wsl2-about)并且已启用Docker Desktop的WSL2后端 (https://aka.ms/vscode-remote/containers/docker-wsl2)，则可以使用WSL中存储的源代码！

启用WSL2引擎后，您可以：

- 从命令面板中选择“ **远程容器：在容器中打开文件夹**...”，(kbstyle(F1)) 然后使用本地 `\\wsl$` 共享选择WSL文件夹。
- 从已使用Remote-WSL (https://aka.ms/vscode-remote/download/wsl)扩展名**打开的文件夹中**使用“ **远程容器：在容器中重新打开文件夹**”命令。 (https://aka.ms/vscode-remote/download/wsl)

快速入门的其余部分照原样适用！您可以在文档中 (/docs/remote/wsl)了解有关Remote-WSL扩展的 (/docs/remote/wsl)更多信息。

### 打开容器中的现有工作区

如果工作空间仅**引用文件所在文件夹（或文件夹本身）的子文件夹的相对路径**，则还可以按照类似的过程在**单个容器中**打开VS Code多根工作空间 (/docs/editor/multi-root-workspaces)。 **`.code-workspace`**

您可以：

- 使用“ **远程容器：在容器中打开工作区**...”命令。
- **打开**包含 `.code-workspace` 容器中文件的文件夹后，使用**文件>打开工作区**...。

连接后，您可能希望**将 `.devcontainer` 文件夹添加**到工作区，以便在尚不可见的情况下轻松编辑其内容。

还要注意，虽然您不能在同一VS Code窗口中的同一工作空间中使用多个容器，但可以从单独的窗口一次 (/docs/remote/containers-advanced#_connecting-to-multiple-containers-at-once)使用多个Docker Compose托管容器 (/docs/remote/containers-advanced#_connecting-to-multiple-containers-at-once)。

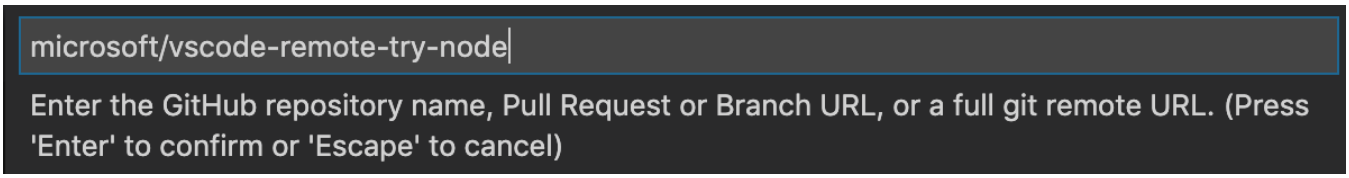
## 快速入门：在隔离的容器卷中打开Git存储库或GitHub PR

虽然可以在容器中打开本地克隆的存储库，但您可能希望使用存储库的隔离副本进行PR审查，或者调查另一个分支而不影响您的工作。

存储库容器使用隔离的本地Docker卷，而不是绑定到本地文件系统。除了不污染文件树之外，本地卷还具有改善Windows和macOS上的性能的好处。（有关如何在其他情况下使用这些类型的卷的信息，请参阅高级配置 (/docs/remote/containers-advanced#_improving-container-disk-performance)。）

例如，请按照以下步骤在存储库容器中打开“尝试”存储库之一：

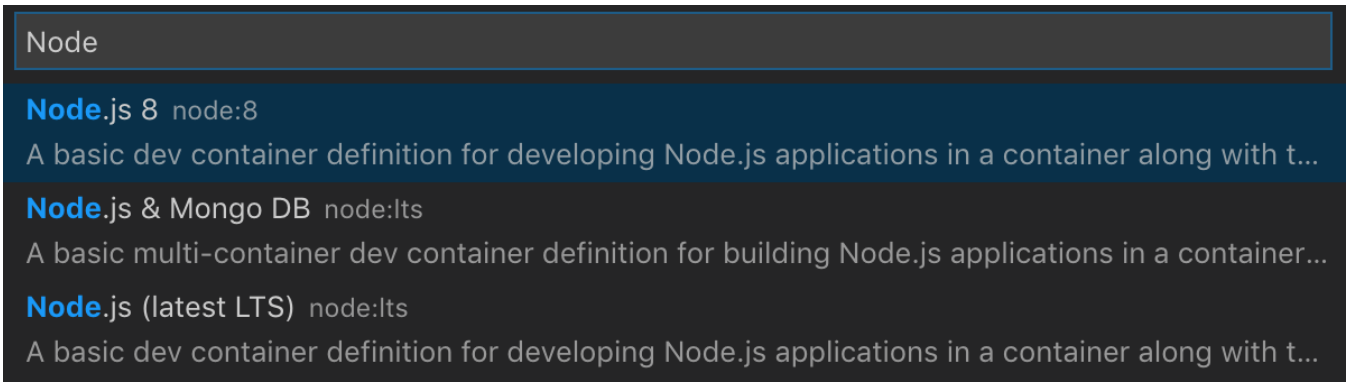
1. 启动VS Code并运行**Remote-Containers：从命令面板（ F1 ）打开容器中的存储库...。**
2. 在出现的输入框中输入 `microsoft/vscode-remote-try-node` （或其他“尝试”存储库之一），Git URI，GitHub分支URL或GitHub PR URL，然后按 Enter 。




> **提示：**如果选择私有存储库，则可能要设置凭证管理器或将SSH密钥添加到SSH代理。请参阅与容器共享Git凭证。

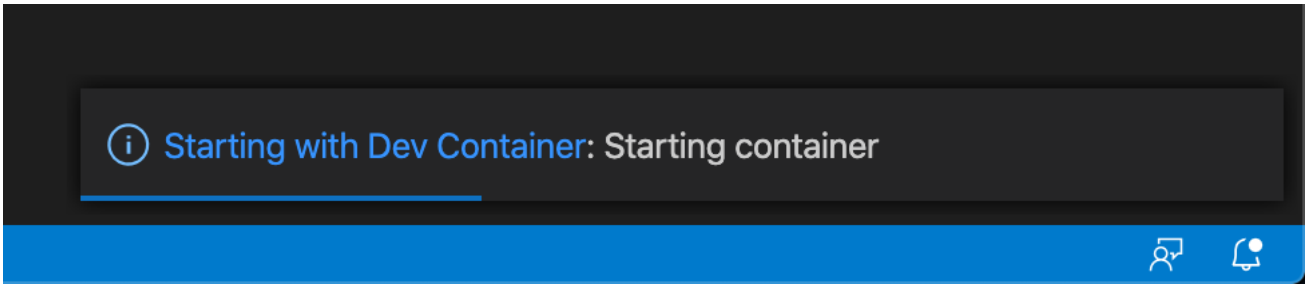
3. 如果您的存储库中没有 `.devcontainer/devcontainer.json` 文件，系统将要求您从可过滤列表或现有Dockerfile (https://docs.docker.com/engine/reference/builder/)或Docker Compose文件 (https://docs.docker.com/compose/compose-file/#compose-file-structure-and-examples)（如果存在）中选择一个起点。

> **注意：**使用Alpine Linux容器时，由于 `glibc` 扩展内部代码的依赖性，某些扩展可能无法正常工作。

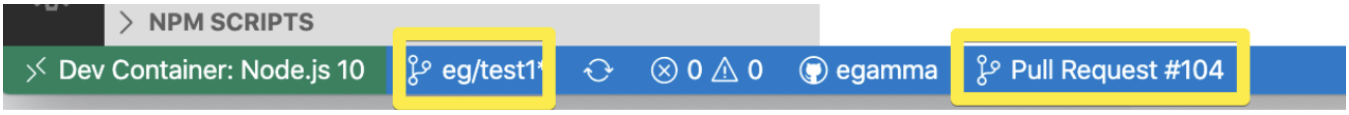



该列表将根据您打开的文件夹的内容自动排序。请注意，显示的dev容器定义来自vscode-dev-containers存储库 (https://aka.ms/vscode-dev-containers)。您可以浏览该 `containers` 存储库的文件夹以查看每个定义的内容。

4. VS Code窗口（实例）将重新加载，克隆源代码并开始构建dev容器。进度通知提供状态更新。




如果您在步骤2中粘贴了GitHub Pull Request URL，则PR将自动检出，并且GitHub Pull Requests (https://marketplace.visualstudio.com/items?itemName=GitHub.vscode-pull-request-github)扩展将安装在容器中。该扩展程序提供了其他与PR相关的功能，例如PR资源管理器，与PR注释在线交互以及状态栏可见性。




5. 构建完成后，VS Code将自动连接到容器。现在，您可以在此隔离的环境中使用存储库源代码，就像您在本地克隆代码一样。

请注意，如果容器由于类似Docker构建错误之类的原因而无法启动，则可以在对话框中选择“ **在恢复容器中打开** ”，该对话框似乎进入“恢复容器”，允许您编辑Dockerfile或其他内容。完成后，使用“ **文件** ”>“ **最近打开** ”以选择存储库容器并重试。

> **提示：是否**要使用远程Docker主机？有关设置的详细信息，请参见“ 高级容器”文章 (/docs/remote/containers-advanced#_developing-inside-a-container-on-a-remote-docker-host)。

## 快速入门：为多个项目或文件夹配置沙箱

虽然开发容器通常绑定到单个文件夹，存储库或项目，但它们也可以与多个文件夹一起使用，以简化工具的设置或沙箱。想象一下，对于给定的工具集，您的源代码跨多个存储库放在一个文件夹中。

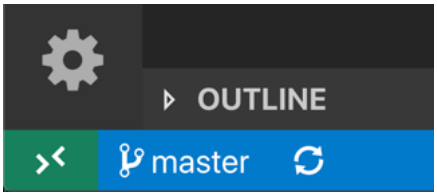
例如：

```
Repos
  node
  python
    starter-snake-python
    vscode-remote-try-python
    your-python-project-here
  go
  dotnet
```

让我们在该 `./Repos/python` 文件夹中设置一个用于所有Python项目的容器。

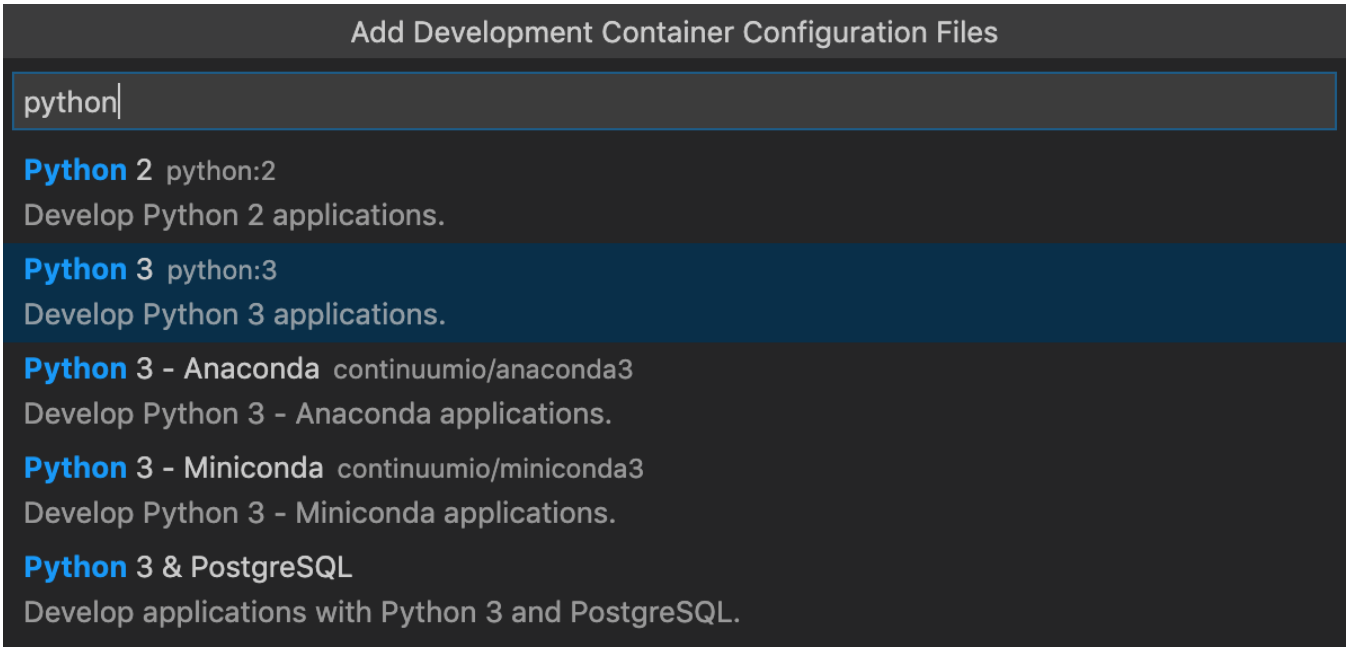
1. 启动VS Code，从“命令面板”（ `F1` ）中选择“ **远程容器：在容器中打开文件夹...** ”或快速操作“状态栏”项，然后选择文件夹。 `./Repos/python`

> **提示：**如果要在打开文件夹之前编辑容器的内容或设置，则可以运行“ **远程容器：添加开发容器配置文件...** ”。



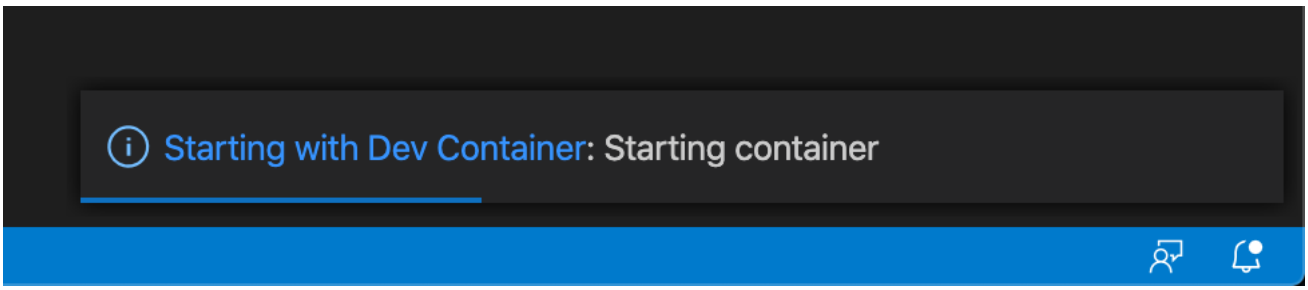

2. 现在为您的开发容器选择一个起点。您可以从可过滤列表中选择基本 **dev容器定义** ，或者如果所选文件夹中存在Dockerfile (https://docs.docker.com/engine/reference/builder/)或Docker Compose文件 (https://docs.docker.com/compose/compose-file/#compose-file-structure-and-examples)，则可以使用现有文件 (https://docs.docker.com/compose/compose-file/#compose-file-structure-and-examples)。

> **注意：**使用Alpine Linux容器时，由于 `glibc` 扩展内部代码的依赖性，某些扩展可能无法正常工作。

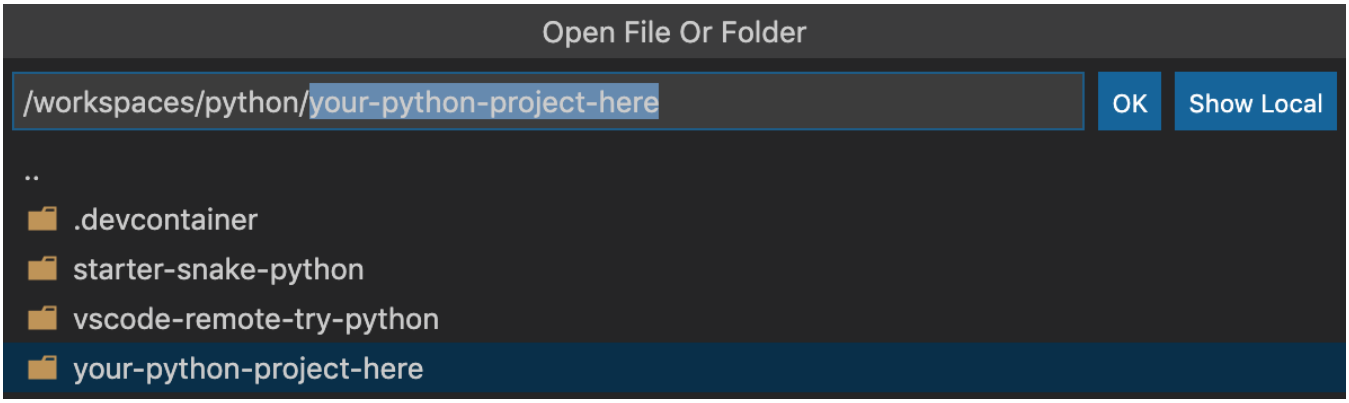



该列表将根据您打开的文件夹的内容自动排序。请注意，显示的dev容器定义来自vscode-dev-containers存储库 (https://aka.ms/vscode-dev-containers)。您可以浏览该 `containers` 存储库的文件夹以查看每个定义的内容。

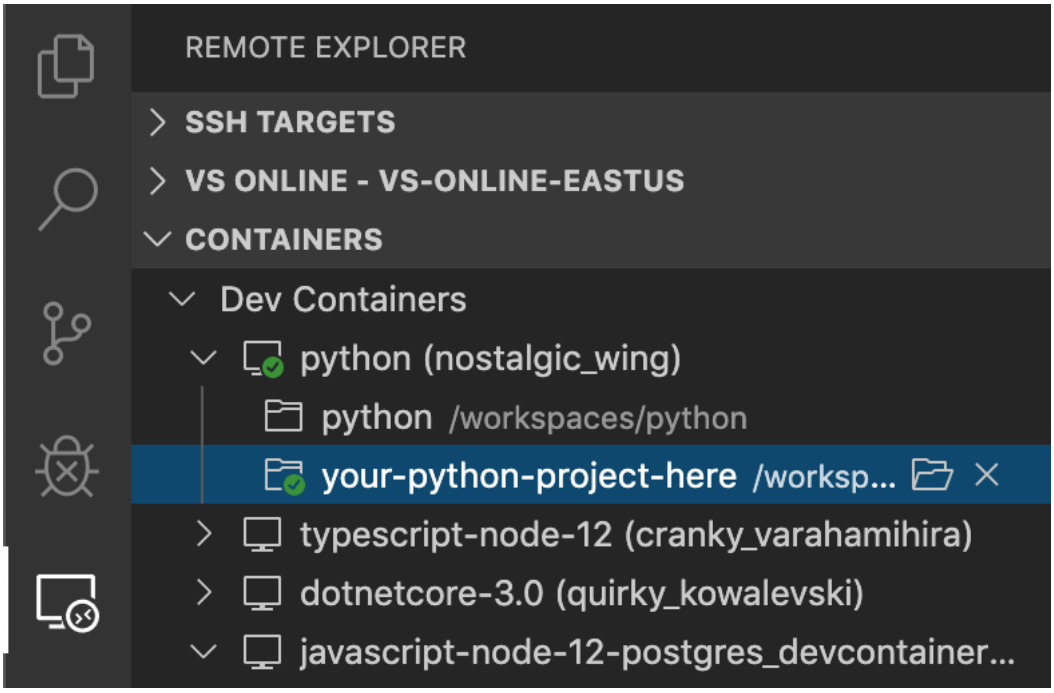
3. 选择容器的起点后，VS Code会将dev容器配置文件添加到该 `./Repos/python/.devcontainer` 文件夹中。
4. VS Code窗口将重新加载并开始构建dev容器。进度通知提供状态更新。您只需要在首次打开容器时就构建一个容器。首次成功构建后打开文件夹将更快。




5. 构建完成后，VS Code将自动连接到容器。连接后，使用**文件**>**打开**... **/打开文件夹**...在下选择一个文件夹 `./Repos/python` 。




6. 稍后，VS Code将打开同一容器内的文件夹。将来，您可以使用活动栏中的**远程资源管理器**直接在容器中打开此子文件夹。




> **提示：**无需挂载本地文件系统，您可以使用类似的流程来设置具有隔离的，性能更高的卷的容器，将您的源代码克隆到该卷中。有关详细信息，请参见“ 高级容器”(/docs/remote/containers-advanced#_use-a-named-volume-for-your-entire-source-tree)文章。

## 创建一个devcontainer.json文件

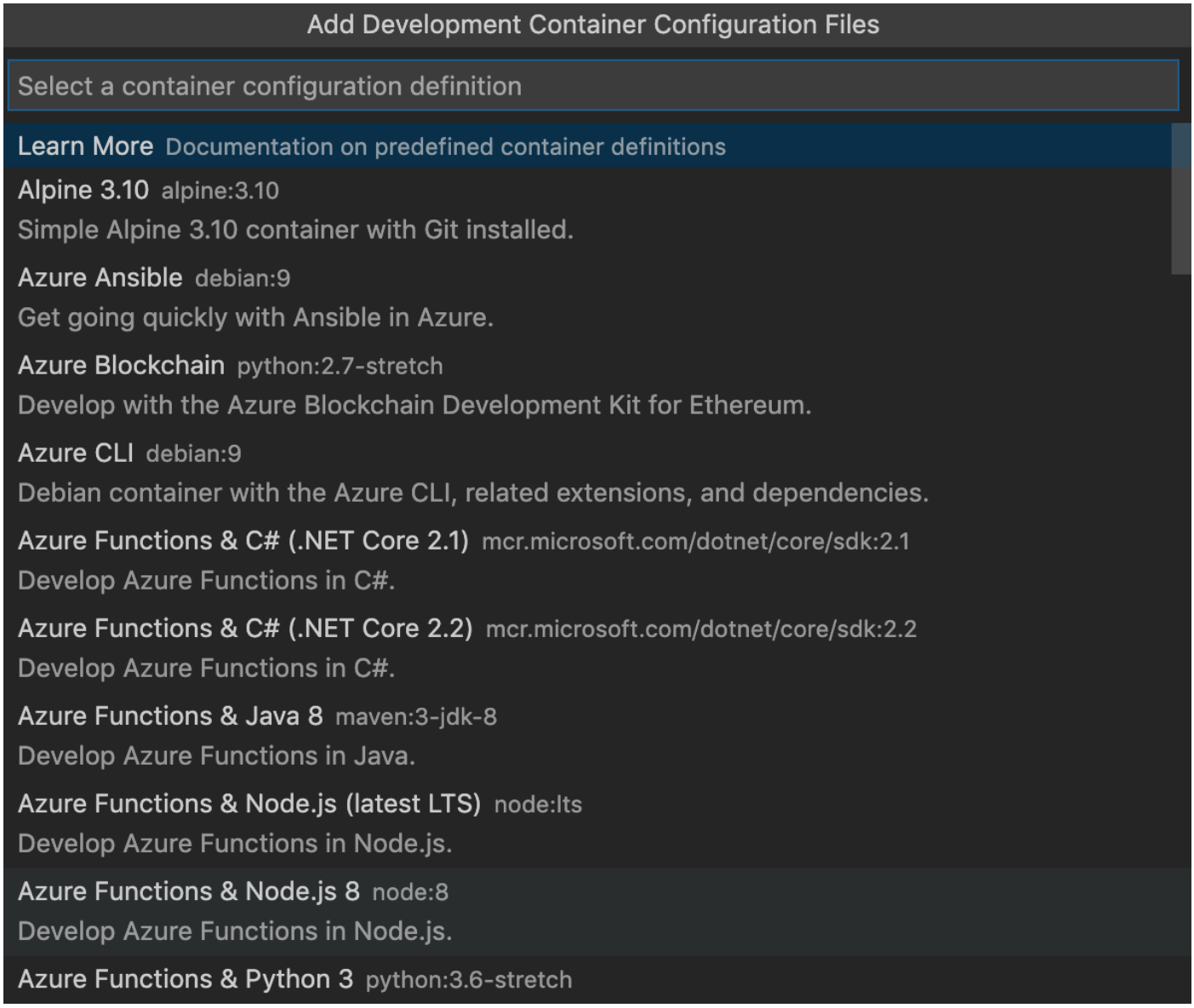
VS Code的容器配置存储在devcontainer.json文件中。该文件类似于 `launch.json` 用于调试配置的文件，但是用于启动（或附加到）开发容器。您还可以指定在容器运行后要安装的任何扩展，或指定创建环境后的命令。开发容器配置位于项目根目录下 `.devcontainer/devcontainer.json` 或作为 `.devcontainer.json` 文件（请注意点缀）存储在项目根目录中。

您可以使用任何映像，Dockerfile或一组Docker Compose文件作为起点。这是一个简单的示例，该示例使用预构建的VS Code开发容器映像之一(https://hub.docker.com/_/microsoft-vscode-devcontainers)：

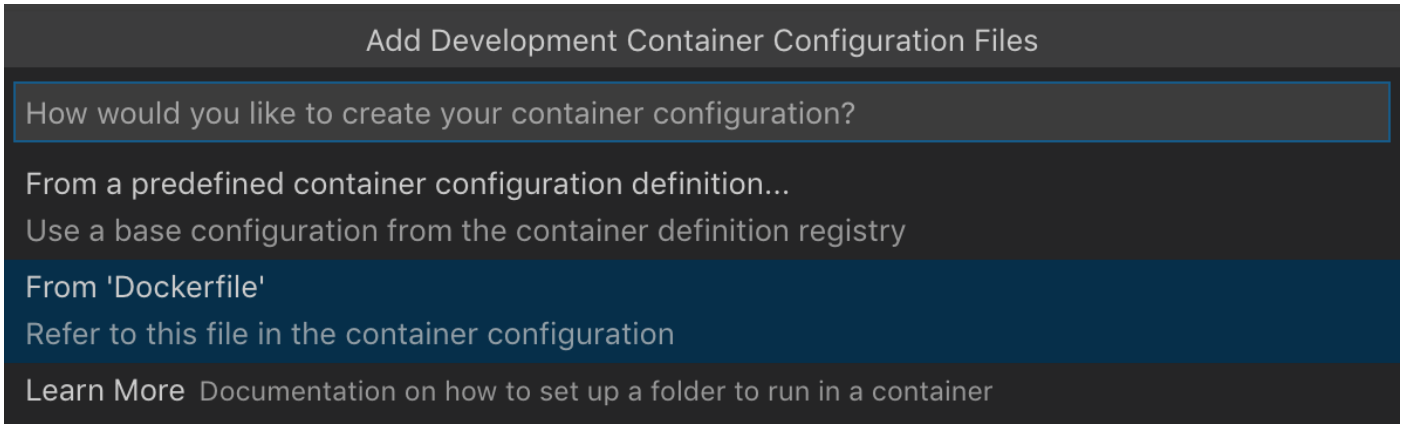
```
{
  "image": "mcr.microsoft.com/vscode/devcontainers/typescript-node:0-12",
  "forwardPorts": [3000],
  "extensions": ["dbaeumer.vscode-eslint"]
}
```

从命令面板（ `F1` ）中选择“ **远程容器：添加开发容器配置文件**...”命令会将所需的文件添加到您的项目中作为起点，您可以根据需要进一步对其进行自定义。

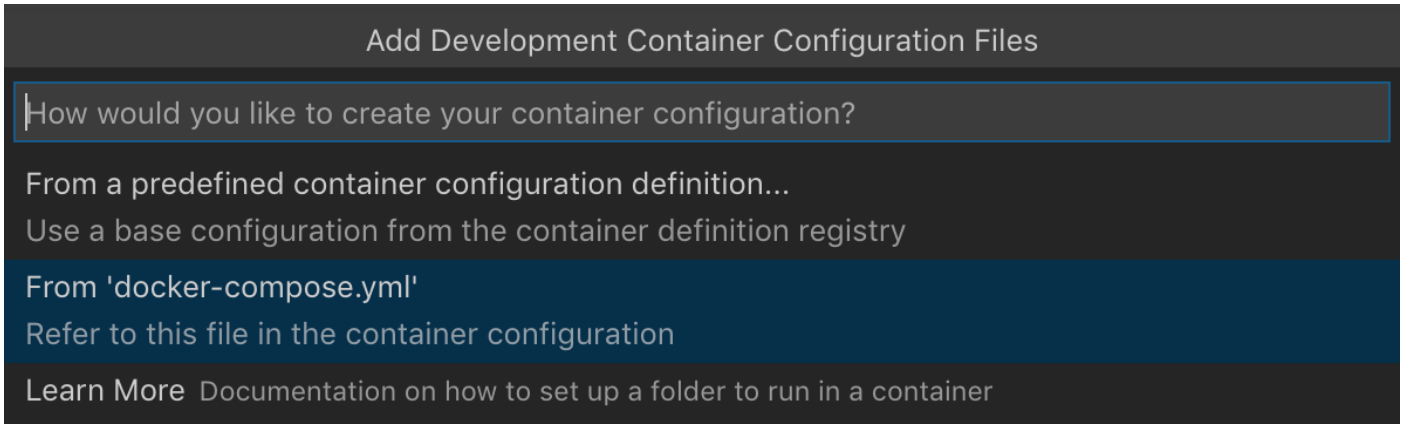
该命令使您可以基于文件夹的内容从列表中选择预定义的容器配置：




或重用现有的Dockerfile：




或重用现有的Docker Compose文件：




例如，通过 `devcontainer.json` 文件，您可以：

- 旋转独立的"沙盒"容器以隔离工具链或加快设置。
- 使用由映像，Dockerfile或Docker Compose定义的容器部署的应用程序。
- (/docs/remote/containers-advanced#_using-docker-or-kubernetes-from-a-container)在开发容器中使用Docker或Kubernetes (/docs/remote/containers-advanced#_using-docker-or-kubernetes-from-a-container)构建和部署您的应用程序。

您可以从vscode-dev-containers存储库中 (https://aka.ms/vscode-dev-containers)选择所有预定义的容器配置，该存储库 (https://aka.ms/vscode-dev-containers)提供了 `devcontainer.json` 针对不同场景的示例。您可以将配置更改为：

- 在容器中安装其他工具，例如Git。
- 自动安装扩展。
- 转发或发布其他端口。
- 设置运行时参数。
- 重用或扩展您现有的Docker Compose设置 (https://aka.ms/vscode-remote/containers/docker-compose/extend)。
- 以及更高级的容器配置 (/docs/remote/containers-advanced)。

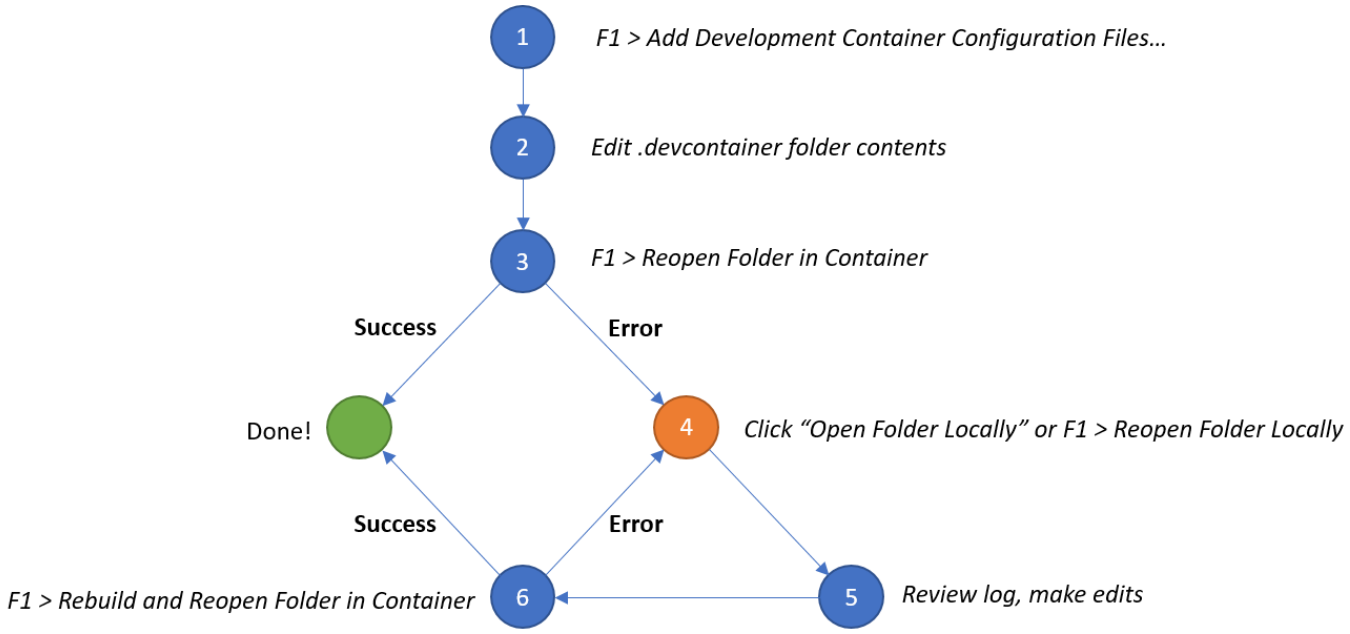
如果 `devcontainer.json` 支持的工作流程不能满足您的需求，您也可以附加到已经运行的容器。

> **提示：是否**要使用远程Docker主机？有关设置的详细信息，请参见" 高级容器"文章 (/docs/remote/containers-advanced#_developing-inside-a-container-on-a-remote-docker-host)。

### 配置编辑循环

编辑容器配置很容易。由于重建容器将容器"重置"为其开始的内容（与当地的源代码除外），VS代码不会自动如果编辑容器配置文件重建（ `devcontainer.json` ， `Dockerfile` ， `docker-compose.yml` ）。相反，有几个命令可用于简化配置的编辑。

这是使用这些命令的典型编辑循环：




1. 从**远程容器**开始：在命令面板（ F1 ）中**添加开发容器配置文件...**。
2. `.devcontainer` 根据需要编辑文件夹的内容。
3. 与**Remote-Containers**一起尝试：**重新打开Container中的Folder**。
4. 如果看到错误，请在出现的对话框中单击" **本地打开文件夹** "。
5. 重新加载窗口后，**构建日志**的副本**将出现**在中，以便您调查问题。 `.devcontainer` 根据需要编辑文件夹的内容。（如果关闭了日志，也可以使用" **远程容器：打开日志文件**..."命令再次查看该日志。）
6. 运行**远程容器：重建并重新打开容器中的文件夹，**如果需要，请跳至步骤4。

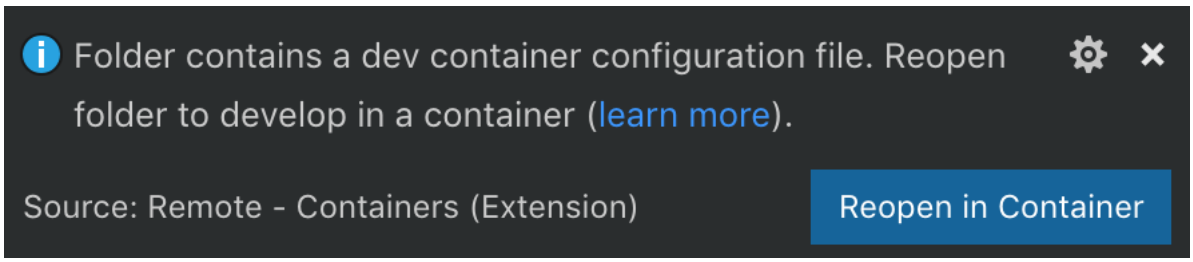
如果已经成功构建，则 `.devcontainer` 在连接到容器时仍可以根据需要编辑文件夹的内容，然后在命令面板（ F1 ）中选择" **远程容器：重新**构建**容器**"，以使更改生效。

使用“ **远程容器：在容器中打开存储库**”命令时，也可以在容器上进行迭代。

1. 从**远程容器**开始：在命令面板（ F1 ）**中的“容器”中打开存储库**。如果输入的存储库中没有 `devcontainer.json` ，则会要求您选择一个起点。
2. `.devcontainer` 根据需要编辑文件夹的内容。
3. 与**Remote-Containers**一起尝试：**Rebuild Container**。
4. 如果看到错误，请在出现的对话框中单击“ **在恢复容器中打开** ”。
5. `.devcontainer` 根据需要在此“恢复容器”中编辑文件夹的内容。
6. 使用**远程容器：**如果仍然遇到问题，请**在“容器”中重新打开**并跳至步骤4。

### 将配置文件添加到公共或私有存储库

通过将 `devcontainer.json` 文件添加到源代码管理，您可以轻松共享项目的自定义dev容器定义。通过将这些文件包含在存储库中，只要安装了“远程-容器”扩展名，任何在VS Code中打开仓库的本地副本的人都将被自动提示以重新打开容器中的文件夹。




除了让您的团队使用一致的环境和工具链带来的好处外，这还使新的贡献者或团队成员更容易快速地提高生产力。初次贡献者将需要较少的指导，并且会减少与环境设置有关的问题。

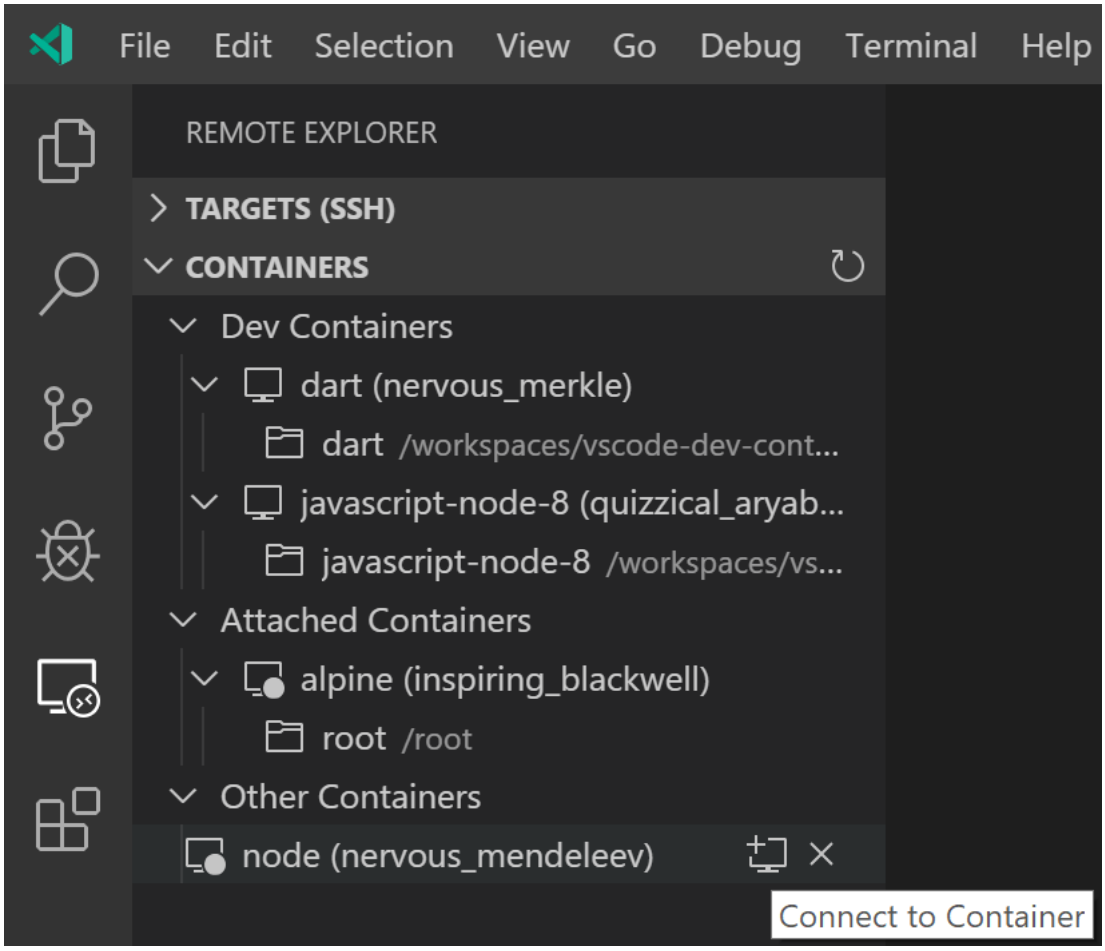
## 附加到正在运行的容器

尽管在许多情况下使用VS Code启动新容器可能很有用，但它可能与您的工作流程不匹配，并且您可能更喜欢将VS Code“附加”到已经运行的Docker容器上，而不管它是如何启动的。附加后，您可以像使用一样在容器中打开文件夹时安装扩展，编辑和调试 `devcontainer.json` 。

> **注意：**使用Alpine Linux容器时，由于 `glibc` 扩展内部代码的依赖性，某些扩展可能无法正常工作。

### 附加到Docker容器

要附加到Docker容器，请从命令面板（ F1 ）中选择“ **远程容器：附加到正在运行的容器**...”命令，或者在活动栏中使用“ **远程资源管理器**”，然后在您**要使用的容器**上选择“ **连接到容器**内联”操作想要连接。




### 附加的容器配置文件

反复连接到给定的Docker容器时，VS Code支持映像级配置文件以加快设置速度。附加后，无论何时打开文件夹，安装扩展程序或转发端口，本地映像特定的配置文件都会自动更新以记住您的设置，以便在再次附加时，一切都回到正确的位置。

要查看或编辑配置，请从命令面板（ F1 ）中选择“ **远程容器：打开附加的容器配置文件**...”命令。打开的文件支持属性的子集： `devcontainer.json`

```
{
  // Default path to open when attaching to a new container.
  "workspaceFolder": "/path/to/code/in/container/here",

  // An array of extension IDs that specify the extensions to
  // install inside the container when you first attach to it.
  "extensions": ["dbaeumer.vscode-eslint"],

  // Any *default* container specific VS Code settings
  "settings": {
    "terminal.integrated.shell.linux": "/bin/bash"
  },

  // An array port numbers to forward
  "forwardPorts": [8000],

  // Container user VS Code should use when connecting
  "remoteUser": "vscode",

  // Set environment variables for VS Code and sub-processes
  "remoteEnv": { "MY_VARIABLE": "some-value" }
}
```

有关属性及其用途的完整列表，请参见随附的容器配置参考。

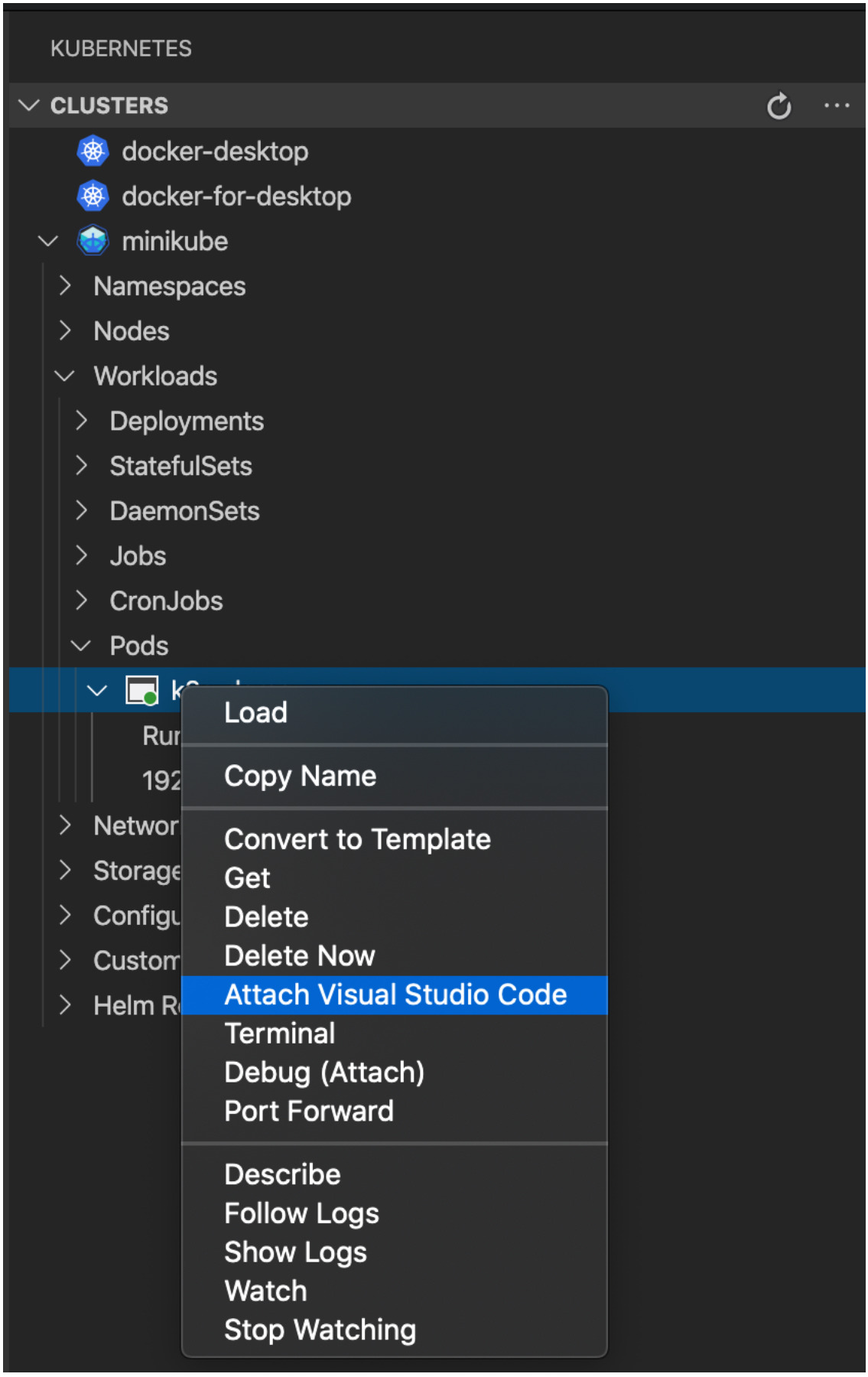
保存后，每当您第一次打开具有相同图像名称的容器时，这些属性将用于配置环境。

最后，如果您有要安装的扩展，而无论附加到哪个容器，则可以进行更新 `settings.json` 以指定应始终安装的扩展的列表。我们将在下一部分中介绍此选项。

### 附加到Kubernetes集群中的容器

要连接到Kubernetes集群中的容器，请首先安装Kubernetes扩展 (https://marketplace.visualstudio.com/items?itemName=ms-kubernetes-tools.vscode-kubernetes-tools) 以及 `kubectl` 远程-容器扩展。然后从活动栏中选择Kubernetes资源管理器，并展开集群和Pod，您要附加到的容器将驻留在该Pod中。最后，右键单击容器，然后从上下文菜单中选择“ **附加Visual Studio代码**”。

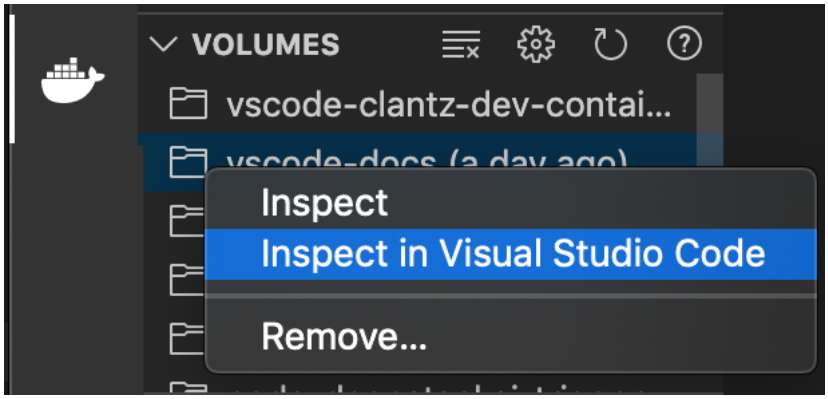
> **注意：** Kubernetes集群中的容器尚不支持附加的容器配置文件。




### 检查卷

有时，您可能会遇到要检查或进行更改的使用Docker命名卷的情况。您可以使用VS Code处理这些内容，而无需 `devcontainer.json` 通过选择**Remote-Containers：检查卷中的**内容来创建或修改文件。**容器**...来自命令面板（ F1 ）。

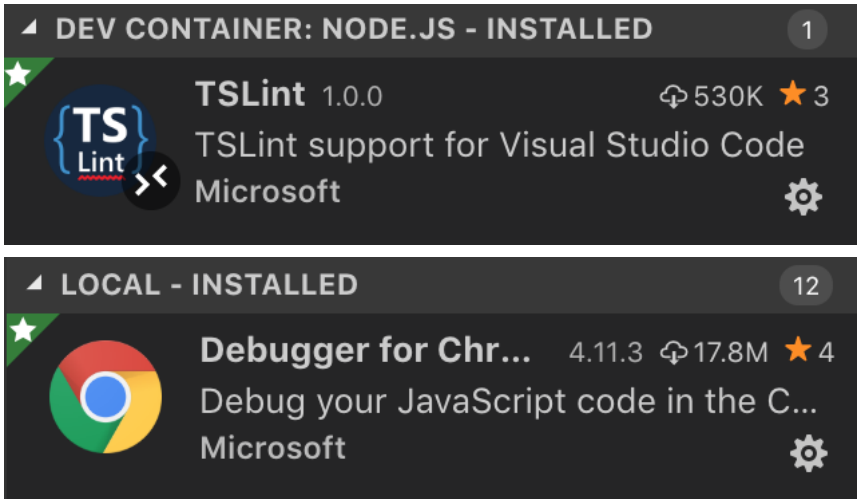
如果您安装了Docker扩展 (https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-docker)，还可以右键单击**Docker Explorer**的**Volumes**部分中的**卷**，然后选择**Inspect in Visual Studio Code**。




## 管理扩展

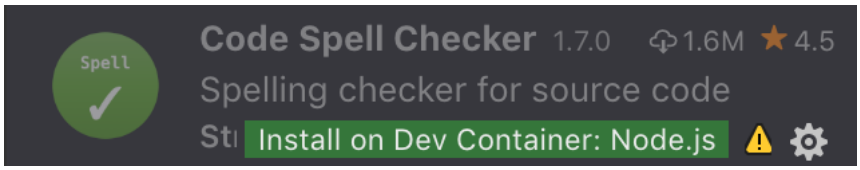
VS Code在以下两个位置之一运行扩展：在UI /客户端本地或在容器中。虽然会在本地安装影响VS Code UI的扩展（例如主题和代码片段），但大多数扩展将驻留在特定的容器内。这样，您就可以在容器中仅安装给定任务所需的扩展，并仅通过连接到新容器即可无缝切换整个工具链。

如果您从"扩展"视图安装扩展，它将自动安装在正确的位置。您可以根据类别分组确定扩展的安装位置。将有一个**Local-Installed**类别，也有一个用于您的容器的类别。

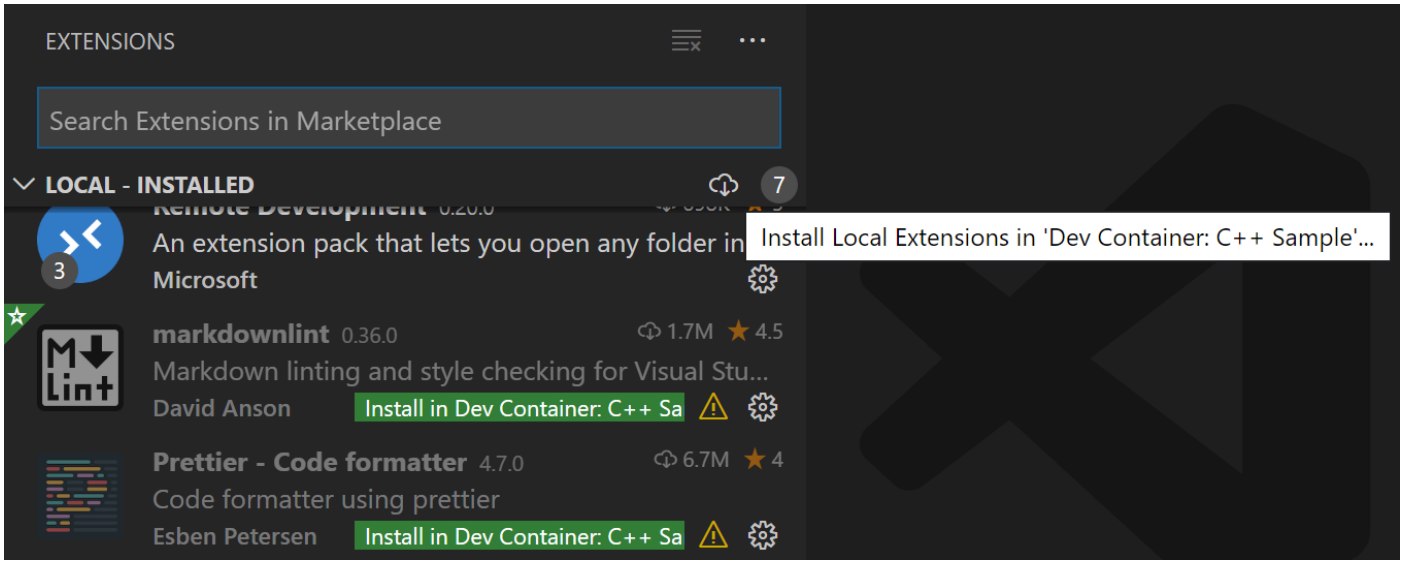



> **注意：**如果您是扩展作者，并且扩展无法正常工作或安装在错误的位置，请参阅支持远程开发 (/api/advanced-topics/remote-extensions)以获取详细信息。

实际上需要远程运行的本地扩展将在" **本地-已安装**"类别中显示" **已禁用** " 。选择**安装**在您的远程主机上安装扩展。



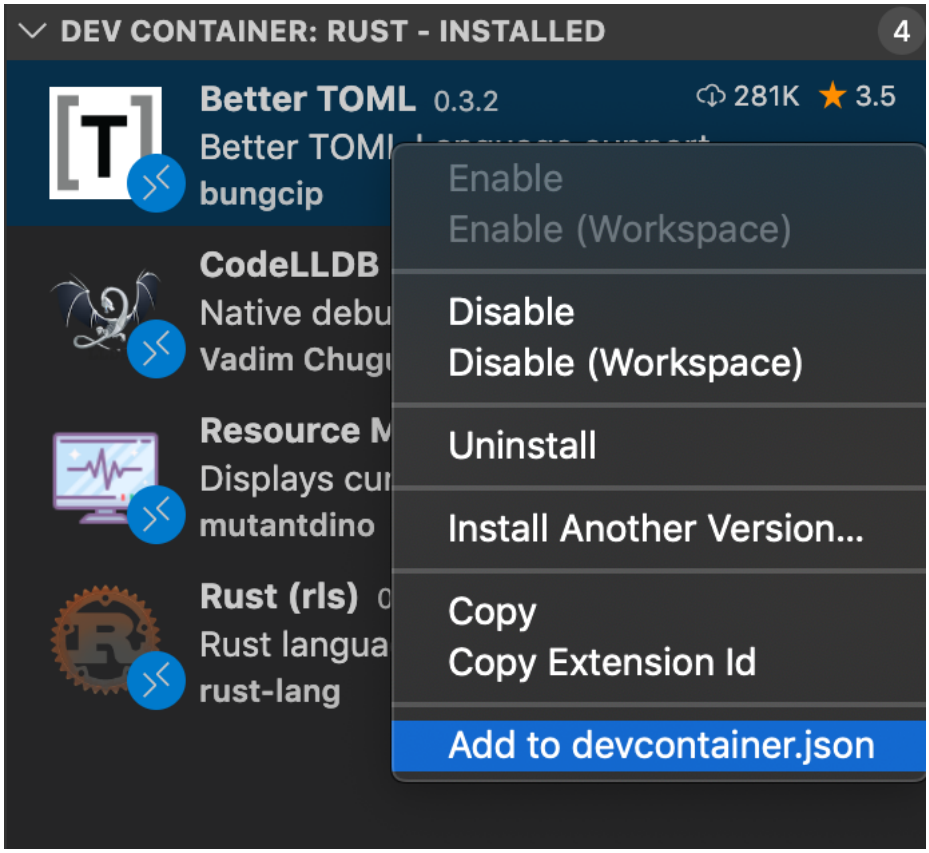

您还可以通过以下方式将所有本地安装的扩展安装在Dev Container中：进入"扩展"视图，然后选择" **本地-已安装**"标题栏右侧的云按钮，选择" **在Dev Container中安装本地扩展：[名称]**"。这将显示一个下拉列表，您可以在其中选择要在容器中安装的本地安装扩展。




但是，某些扩展程序可能需要您在容器中安装其他软件。如果遇到问题，请查阅扩展文档以获取详细信息。

向devcontainer.json添加扩展

您可以手动编辑devcontainer.json文件以添加扩展名ID列表，也可以在Extensions视图中右键单击任何扩展名，然后选择**Add to devcontainer.json**。




"始终安装"扩展

如果您希望始终将扩展安装在任何容器中，则可以更新 `remote.containers.defaultExtensions` 用户设置 (/docs/getstarted/settings)。例如，如果您想安装GitLens (https://marketplace.visualstudio.com/items?itemName=eamodio.gitlens)和Resource Monitor (https://marketplace.visualstudio.com/items?itemName=mutantdino.resourcemonitor)扩展，则可以如下指定其扩展ID：

```
"remote.containers.defaultExtensions": [
    "eamodio.gitlens",
    "mutantdino.resourcemonitor"
]
```

高级：强制扩展在本地或远程运行

扩展通常经过设计和测试，可以在本地或远程运行，不能同时运行。但是，如果扩展支持，则可以强制其在 `settings.json` 文件中的特定位置运行。

例如，以下设置将强制Docker扩展在本地运行，而Debugger for Chrome扩展将在远程运行而不是默认设置：

```
"remote.extensionKind": {
    "ms-azuretools.vscode-docker": [ "ui" ],
    "msjsdiag.debugger-for-chrome": [ "workspace" ]
}
```

的价值 `"ui"` ，而不是 `"workspace"` 将迫使扩展到在本地UI /客户端，而不是运行。通常，除非扩展文档中另有说明，否则仅应将其用于测试，因为它**可能会破坏扩展**。有关详细信息，请参见支持远程开发 (/api/advanced-topics/remote-extensions)的文章。

## 转发或发布端口

容器是隔离的环境，因此，如果要访问容器中的服务器，服务或其他资源，则需要将端口"转发"或" 发布 (https://stackoverflow.com/a/22150099) "到主机。您可以将容器配置为始终公开这些端口，也可以只是临时转发它们。

始终转发端口

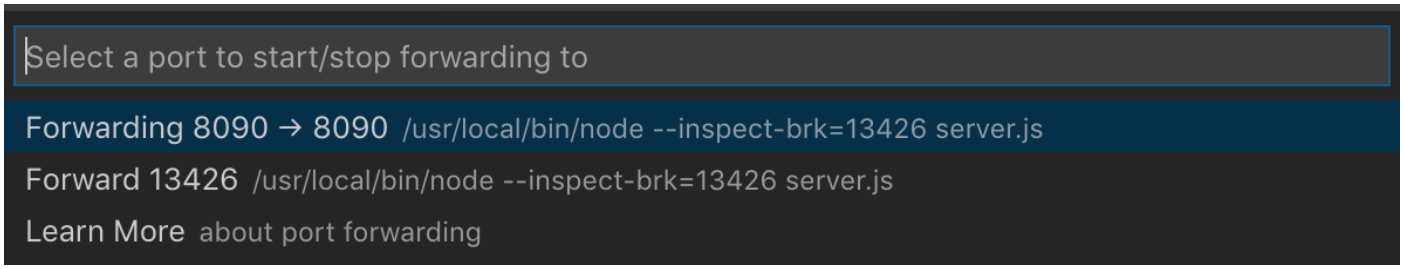
您可以使用中的属性指定在附加或打开容器中的文件夹时**始终**要转发的端口列表。 `forwardPorts devcontainer.json`

```
"forwardPorts": [3000, 3001]
```

只需重新加载/重新打开窗口，当VS Code连接到容器时，该设置就会应用。

临时转发端口

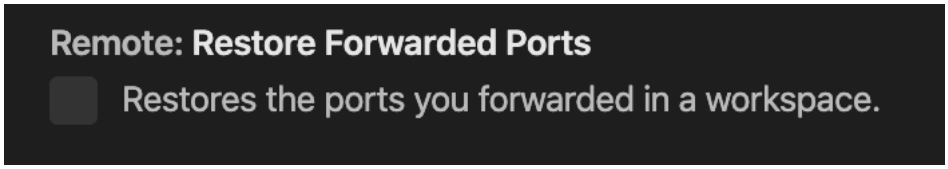
如果您需要访问未添加 `devcontainer.json` 或未在Docker Compose文件中发布的端口，则可以通过在命令面板（ F1 ）中运行“ **转发端口**”命令在会话持续时间内**临时转发**新端口。




选择端口后，通知将告诉您应使用localhost端口访问容器中的端口。例如，如果转发了侦听端口3000的HTTP服务器，则通知可能会告诉您它已映射到本地主机上的端口4123。然后，您可以使用来连接到该远程HTTP服务器 `http://localhost:4123` 。

如果以后需要访问此信息，可在Remote Explorer 的**转发端口**部分中找到。

如果您希望VS Code记住已转发的任何端口，请在“设置”编辑器（ Ctrl +， ）中选中“ **远程：还原转发的端口** ”或在中设置。 `"remote.restoreForwardedPorts": true settings.json`




### 发布端口

创建容器时，Docker的概念是“发布”端口。发布的端口的行为非常类似于您可用于本地网络的端口。如果您的应用程序仅接受来自的呼叫 `localhost` ，则它将拒绝来自已发布端口的连接，就像本地计算机进行网络呼叫一样。另一方面，转发的端口实际上看起来像 `localhost` 应用程序。每种方法在不同情况下都很有用。

要发布端口，您可以：

1. **使用appPort属性：**如果在中引用映像或Dockerfile `devcontainer.json` ，则可以使用该 `appPort` 属性将端口发布到主机。

   ```
   "appPort": [ 3000, "8921:5000" ]
   ```

2. **使用泊坞窗撰写端口映射：**该端口映射 (https://docs.docker.com/compose/compose-file#ports)可以很容易地添加你的 `docker-compose.yml` 文件发布额外的端口。

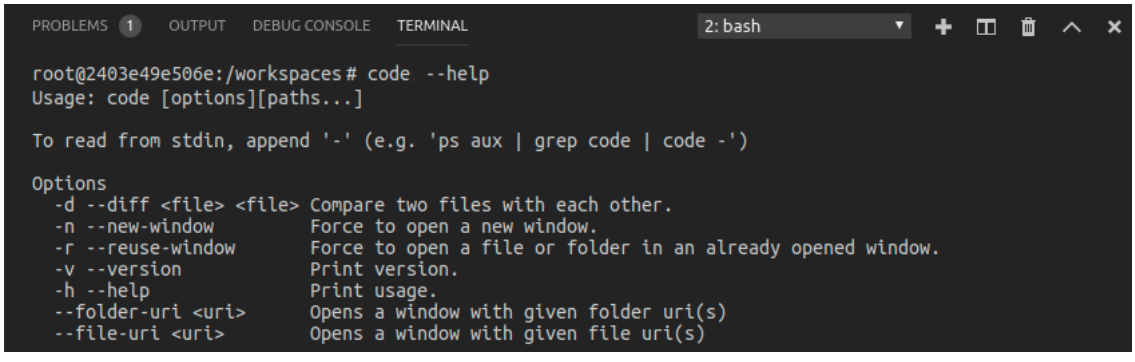
   ```
   ports:
   - "3000"
   - "8921:5000"
   ```

在每种情况下，您都需要重建容器以使设置生效。当您连接到容器时，可以通过在命令面板（ F1 ）中运行“ **远程容器：重建容器**”命令来执行此操作。

## 打开终端

通过VS Code在容器中打开终端很简单。在容器中打开文件夹后，您在VS Code中打开的**任何终端窗口**（**Terminal> New Terminal**）将自动在容器中运行，而不是在本地运行。

您也可以在 `code` 同一终端窗口中使用命令行来执行许多操作，例如在容器中打开新文件或文件夹。键入 `code --help` 以从命令行了解可用的选项。



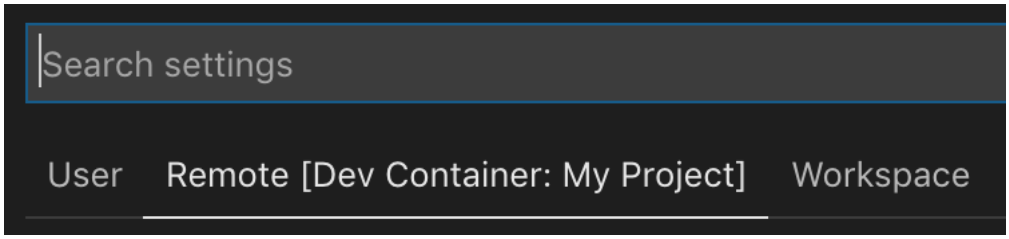

## 在容器中调试

在容器中打开文件夹后，就可以像在本地运行应用程序时一样的方式使用VS Code的调试器。例如，如果您在中选择启动配置 `launch.json` 并开始调试（ F5 ），则应用程序将在远程主机上启动并将调试器附加到该主机上。

有关在中配置VS Code的调试功能的详细信息，请参见调试 (/docs/editor/debugging)文档 `.vscode/launch.json` 。

## 容器特定的设置

当您连接到开发容器时，也会重复使用VS Code的本地用户设置。虽然这可以使您的用户体验保持一致，但是您可能需要在本地计算机和每个容器之间更改某些设置。幸运的是，一旦连接到容器，您还可以通过运行“ **偏好**设置”来设置特定于容器的设置：从命令面板（ F1 ）**打开“远程设置”**命令，或者在“设置”编辑器中选择“ **远程**”选项卡。每当您连接到容器时，这些设置将覆盖您已有的所有本地设置。




默认容器特定设置

您可以在 `devcontainer.json` 使用 `settings` 属性中包括容器特定设置的默认值。创建容器后，这些值将自动放置在容器内特定于容器的设置文件中。

例如，将其添加到 `.devcontainer/devcontainer.json` 将设置Java主目录路径：

```
"settings": {
    "java.home": "/docker-java-home"
}
```

由于这只是建立默认值，因此一旦创建容器，您仍然可以根据需要更改设置。

## 与您的容器共享Git凭据

Remote-Containers扩展为从容器内部使用本地Git凭据提供了开箱即用的支持。在本节中，我们将逐步介绍两个受支持的选项。

如果您没有在本地设置用户名或电子邮件地址，则系统可能会提示您设置用户名或电子邮件地址。您可以通过运行以下命令在**本地**计算机上执行此操作：

```
git config --global user.name "Your Name"
git config --global user.email "your.email@address"
```

该扩展名将 `.gitconfig` 在启动时自动将本地文件复制到容器中，因此您无需在容器本身中执行此操作。

### 使用凭证助手

如果使用HTTPS克隆存储库并**在本地OS中配置** (https://help.github.com/en/articles/caching-your-github-password-in-git)**了凭据帮助** (https://help.github.com/en/articles/caching-your-github-password-in-git)**程序，则无需进一步设置。**您在本地输入的凭据将在容器中重复使用，反之亦然。

### 使用SSH密钥

在某些情况下，您可能会使用SSH密钥而不是凭据帮助程序来克隆存储库。要启用此方案，**如果正在运行**，扩展将自动转发您的**本地SSH代理** (https://www.ssh.com/ssh/agent)。

您可以使用 `ssh-add` 命令将本地SSH密钥添加到代理（如果正在运行）。例如，从终端或PowerShell运行以下命令：

```
ssh-add $HOME/.ssh/github_rsa
```

在Windows和Linux上，您可能会收到错误消息，因为该代理未运行（macOS通常默认情况下正在运行它）。请按照以下步骤解决问题：

**Windows**：

启动**本地Administrator PowerShell**并运行以下命令：

```
# Make sure you're running as an Administrator
Set-Service ssh-agent -StartupType Automatic
Start-Service ssh-agent
Get-Service ssh-agent
```

**Linux:**

首先，通过在终端中运行以下命令在后台启动SSH代理：

```
eval "$(ssh-agent -s)"
```

然后将这些行添加到您的 `~/.bash_profile` 或 `~/.zprofile` （对于Zsh）中，以便从登录时开始：

```
if [ ! -z "$SSH_AUTH_SOCK" ]; then
   # Check for a currently running instance of the agent
   RUNNING_AGENT="`ps -ax | grep 'ssh-agent -s' | grep -v grep | wc -l | tr -d '[:space:]'`"
   if [ "$RUNNING_AGENT" = "0" ]; then
        # Launch a new instance of the agent
        ssh-agent -s &> .ssh/ssh-agent
   fi
   eval `cat .ssh/ssh-agent`
fi
```

### 共享GPG密钥

如果要使用GPG (https://www.gnupg.org/)签署您的提交，则也可以与容器共享本地密钥。您可以在GitHub文档中 (https://help.github.com/en/github/authenticating-to-github/managing-commit-signature-verification)找到有关使用GPG密钥签名的信息 (https://help.github.com/en/github/authenticating-to-github/managing-commit-signature-verification)。

如果您没有设置GPG，则可以在**Windows**上安装Gpg4win (https://www.gpg4win.org/)或在**macOS上**可以安装GPG工具 (https://gpgtools.org/)。在**Linux上**，使用系统的软件包管理器**在本地**安装 `gnupg2` 软件包。

接下来， `gnupg2` 通过更新Dockerfile 将其安装在容器中。例如：

```
RUN apt-get update && apt-get install gnupg2 -y
```

或者，如果以非root用户 (/docs/remote/containers-advanced#_adding-a-nonroot-user-to-your-dev-container)身份运行：

```
RUN sudo apt-get update && sudo apt-get install gnupg2 -y
```

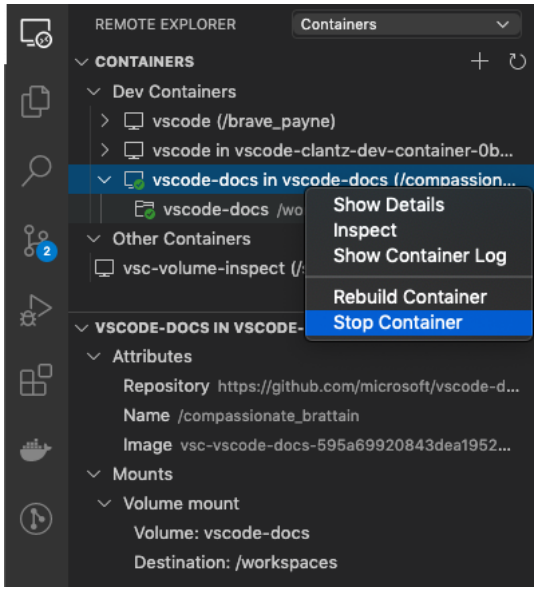
容器下次启动时，您的GPG密钥也应该在容器内部也可以访问。

> **注意：**如果 `gpg` 以前在容器中使用过，则可能需要运行“ **远程容器：重建容器**”以使更新生效。

## 管理容器

默认情况下，“远程-容器”扩展程序会自动启动 `devcontainer.json` 打开文件夹时提到的容器。当您关闭VS Code时，扩展程序会自动关闭您连接的容器。您可以通过添加更改此行为 `"shutdownAction": "none"` 来 `devcontainer.json` 。

尽管可以使用命令行来管理容器，但是也可以使用 **Remote Explorer**。要停止容器，请从下拉菜单（如果有）中选择“**容器**”，右键单击正在运行的容器，然后选择“ **停止容器**”。您还可以启动退出的容器，删除容器和删除最近的文件夹。在“详细信息”视图中，您可以转发端口并在浏览器中打开已转发的端口。



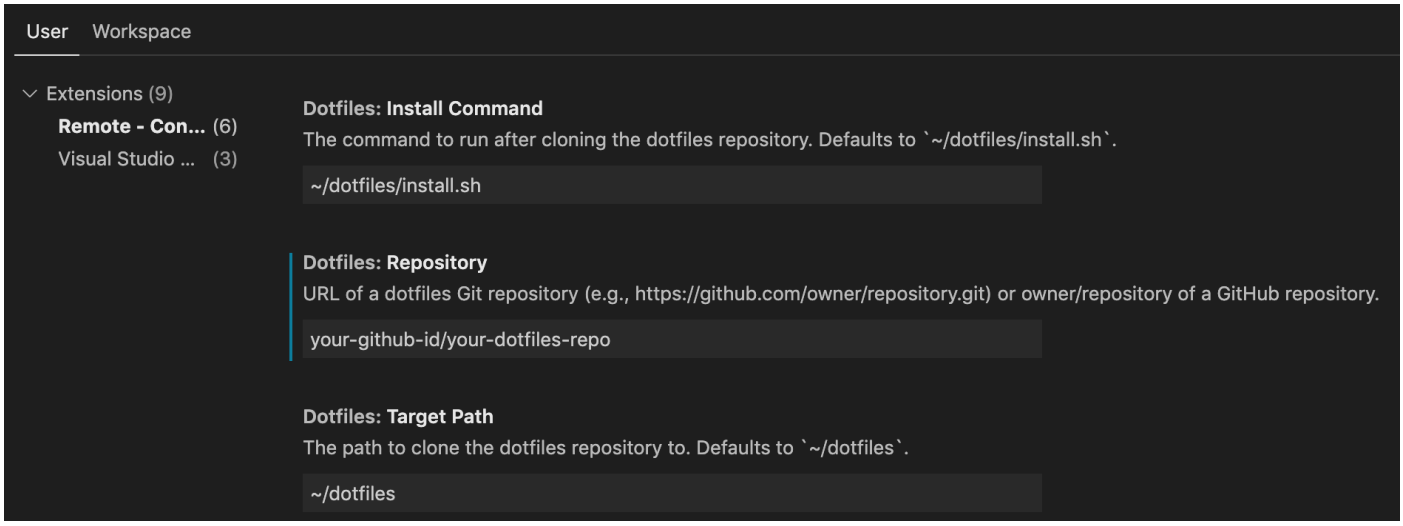

如果要清除图像或批量删除的容器，请参阅清除未使用的容器和图像 (/docs/remote/troubleshooting#_cleaning-out-unused-containers-and-images)以获取其他选项。

## 使用dotfile存储库进行个性化

点文件是文件名以点（ `.` ）开头的文件，通常包含各种应用程序的配置信息。由于开发容器可以涵盖广泛的应用程序类型，因此将这些文件存储在某个位置可能很有用，以便一旦启动并运行它们就可以轻松将它们复制到容器中。

一种常见的方法是将这些点文件存储在GitHub存储库中，然后使用实用程序来克隆和应用它们。Remote-Containers扩展内置支持将它们与您自己的容器一起使用。如果您不熟悉 (https://dotfiles.github.io/)此概念，请查看存在的其他dotfiles引导存储库 (https://dotfiles.github.io/)。

要使用它，请将您的dotfiles GitHub存储库添加到VS Code的用户设置（ `Ctrl +,` ），如下所示：




或在 `settings.json` ：

```
{
  "dotfiles.repository": "your-github-id/your-dotfiles-repo",
  "dotfiles.targetPath": "~/dotfiles",
  "dotfiles.installCommand": "~/dotfiles/install.sh"
}
```

从现在开始，无论何时创建容器，都将使用dotfiles存储库。

## 深入了解：设置文件夹以在容器中运行

VS Code Remote-有几种不同的方法可以使用容器在完全容器化的环境中开发应用程序。通常，有两种主要的场景引起了人们对这种开发风格的兴趣：

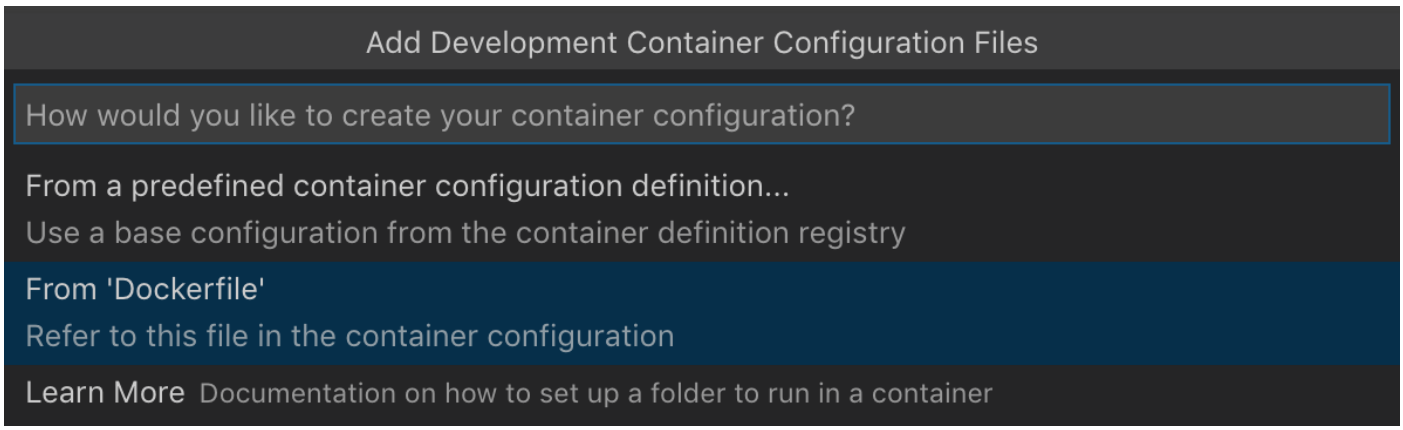
- **独立开发人员沙箱**：您可能不会将应用程序部署到容器化环境中，但仍希望将构建和运行时环境与本地OS隔离开来，加快设置速度，或在更具生产代表性的环境中进行开发。例如，您可能正在本地macOS或Windows计算机上运行代码，这些代码最终将部署到Linux VM或服务器上，对多个项目有不同的工具链要求，或者希望能够使用可能影响您本地计算机的工具/软件包。机器意外或不希望的方式。为此，您可以引用容器映像或Dockerfile。
- **容器部署的应用程序**：您将应用程序部署到一个或多个容器中，并且希望在容器化环境中本地工作。VS Code当前支持使用以多种方式定义的基于容器的应用程序：
  - Dockerfile：您正在使用单个Dockerfile描述单个容器/服务。
  - Docker Compose：您正在使用使用 `docker-compose.yml` 文件描述的多个协调服务。
  - 在每种情况下，您可能还需要构建容器映像并 (/docs/remote/containers-advanced#_using-docker-or-kubernetes-from-a-container)从容器内部部署到Docker或Kubernetes (/docs/remote/containers-advanced#_using-docker-or-kubernetes-from-a-container)。
  - 仅附加： `devcontainer.json` 如果本节中描述的任何工作流程都不满足您的需要，则可以附加到已经运行的容器，尽管没有附加支持。

本节将引导您针对每种情况配置项目。该vscode-DEV-容器GitHub的仓库 (https://aka.ms/vscode-dev-containers)中还含有开发容器定义，让您快速启动和运行。

### 使用映像或Dockerfile

您可以通过向项目中添加（或）配置文件，将VS Code配置为将DockerHub (https://hub.docker.com)或Azure容器注册表 (https://azure.microsoft.com/services/container-registry/)等来源的现有映像重 (https://hub.docker.com)用于您的开发容器。此外，如果您无法找到满足您需要的映像，只有一个基于容器的项目，或者只是想自动化安装多个其他依赖项，则可以使用Dockerfile (https://docs.docker.com/engine/reference/builder/)生成映像。 `.devcontainer/devcontainer.json` `.devcontainer.json` (https://docs.docker.com/engine/reference/builder/)

为了快速上手，请在VS Code中**打开要使用的文件夹**，然后在命令面板（ `F1` ）中运行" **远程容器：添加开发容器配置文件**..."命令。




系统会要求您选择一个现有的Dockerfile（如果存在），或者从vscode-dev-containers存储库 (https://github.com/Microsoft/vscode-dev-containers)中选择一个预定义的容器配置，该存储库会 (https://github.com/Microsoft/vscode-dev-containers)根据文件夹的内容自动排序。然后，VS Code会将 `devcontainer.json` 所有其他必需的文件添加到该文件夹中。尽管大多数这些预定义的"开发容器定义"都包含一个Dockerfile，但是您可以根据需要将它们用作映像的起点。

> **注意：**使用Alpine Linux容器时，由于 `glibc` 扩展内部代码的依赖性，某些扩展可能无法正常工作。

您也可以手动创建配置。配置VS Code以使用Dockerfile构建容器映像或仅重用现有映像之间的区别是 `devcontainer.json`：

- **要使用图像：** 设置 `image` 属性。例如，这将使用JavaScript和Node.js 12预先构建的VS Code Development Container映像 (https://hub.docker.com/_/microsoft-vscode-devcontainers)，转发端口3000，安装ES Lint扩展名，并 `npm install` 在完成后运行：

  ```
  {
    "name": "My Project",
    "image": "mcr.microsoft.com/vscode/devcontainers/javascript-node:0-12",
    "forwardPorts": [3000],
    "extensions": ["dbaeumer.vscode-eslint"],
    "postCreateCommand": "npm install"
  }
  ```

- **要使用Dockerfile：** 设置 `dockerFile` 属性。例如，这将导致VS Code使用指定的 `Dockerfile` 正向端口5000 构建dev容器映像，并在容器中安装C＃扩展名：

  ```
  {
    "name": "My Node.js App",
    "dockerFile": "Dockerfile",
    "forwardPorts": [5000],
    "extensions": ["ms-dotnettools.csharp"]
  }
  ```

见devcontainer.json参考有关其他可用的属性，如信息 `forwardPorts` ， `postCreateCommand` 以及 `extensions` 列表。

将 `.devcontainer/devcontainer.json` 文件添加到文件夹后，从命令面板（ `F1` ）运行“ **远程容器：在容器中重新打开文件夹**”命令（或，如果尚未使用VS Code ，则运行“ **远程容器：在容器中打开文件夹**... ”） 。创建**容器后**，除非更改此行为 (/docs/remote/containers-advanced#_changing-the-default-source-code-mount)，否则**本地文件系统会自动“绑定”安装到容器中**，并且可以从VS Code开始使用它。 (/docs/remote/containers-advanced#_changing-the-default-source-code-mount)

但是，在Linux上，使用绑定安装时，您可能需要设置并**指定非root用户**，否则您创建的任何文件都是root。扩展附带的所有配置文件和映像都包括您可以指定的非root用户。有关详细信息，请参见将非root用户添加到您的dev容器 (/docs/remote/containers-advanced#_adding-a-nonroot-user-to-your-dev-container)。

```
# Change user for VS Code and sub-processes (terminals, tasks, debugging)
"remoteUser": "your-user-name-here",
# Or change the user for all container processes
"containerUser": "your-user-name-here"
```

您还可以使用该属性添加**其他本地安装点，**以使容器可以访问其他位置 `mounts` 。

例如，您可以装载家庭/用户个人资料文件夹：

```
"mounts": [
    "source=${localEnv:HOME}${localEnv:USERPROFILE},target=/host-home-folder,type=bind,consistency=cached"
]
```

`"${localWorkspaceFolder}` 如果需要将某些文件从本地文件系统挂载到容器中，也可以参考。

该 `runArgs` 属性与docker run (https://docs.docker.com/engine/reference/commandline/run/)命令支持相同的参数列表，并且可用于包括设置环境变量 (/docs/remote/containers-advanced#_adding-environment-variables)在内的多种场景。

如果您的应用程序是使用C ++，Go或Rust或使用基于**ptrace的调试器的**另一种语言构建的，则该 `runArgs` 属性还可用于配置所需的运行时安全性和功能设置。

例如：

```
{
  "name": "My Go App",
  "dockerFile": "Dockerfile",
  "extensions": ["golang.go"],
  "runArgs": ["--cap-add=SYS_PTRACE", "--security-opt", "seccomp=unconfined"]
}
```

尽管效率不如自定义Dockerfile，但是您也可以使用该 `postCreateCommand` 属性来**安装其他软件**，这些**软件可能不在基础映像中，**或者在某些情况下您**不希望修改**正在重复使用的**Dockerfile。**

例如，以下是在基本Ubuntu 18.04容器映像 `devcontainer.json` 中添加编译器工具和C ++ (https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools)扩展的：

```
{
  "image": "ubuntu:18.04",
  "extensions": ["ms-vscode.cpptools"],
  "postCreateCommand": "apt-get update && apt-get install -y build-essential cmake cppcheck valgrind"
}
```

有关使用安装软件的更多信息，请参阅安装其他 `apt-get` 软件。

一旦挂载了源代码，该命令就会运行，因此您也可以使用该属性 `npm install` 在源代码树中运行命令或执行Shell脚本之类的命令。

```
"postCreateCommand": "bash scripts/install-dev-tools.sh"
```

默认情况下，当VS代码开头的的容器，它会**覆盖容器的默认命令**是 `/bin/sh -c "while sleep 1000; do :; done"` 。这样做是因为如果默认命令失败或退出，容器将停止。但是，这可能不适用于某些图像。如果您使用的映像要求运行默认命令才能正常工作，请在 `devcontainer.json` 文件中添加以下内容。

```
"overrideCommand": false
```

首次创建容器后，您将需要运行**Remote-Containers：Rebuild Container**命令以使 `devcontainer.json` Dockerfile的更新或使其生效。

### 安装其他软件

将VS Code连接到容器后，您可以打开VS Code终端并针对容器内的OS执行任何命令。这使您可以安装新的命令行实用程序，并从Linux容器内部启动数据库或应用程序服务。

大多数容器映像基于Debian或Ubuntu，其中 `apt` or `apt-get` 命令用于安装新软件包。您可以在Ubuntu文档中 (https://help.ubuntu.com/lts/serverguide/apt.html)了解有关该命令的更多信息。高山图像包括类似 `apk` 命令 (https://wiki.alpinelinux.org/wiki/Alpine_Linux_package_management)而的CentOS / RHEL / Oracle的SE / Fedora的图像使用 `yum` (https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/deployment_guide/ch-yum)或最近 `dnf` (https://fedoraproject.org/wiki/DNF?rd=Dnf)。

要安装的软件的文档通常会提供特定的说明，但是 `sudo` 如果您以root用户身份在容器中运行，则可能不需要在命令前加上前缀。

例如：

```
# If running as root
apt-get update
apt-get install <package>
```

如果您以root用户身份运行，则只要 `sudo` 您在容器中配置，就可以安装软件。已经 `sudo` 设置了所有预定义的容器，但是“ 高级容器配置”一 (/docs/remote/containers-advanced#_adding-a-nonroot-user-to-your-dev-container)文可以帮助您为自己的容器进行设置。无论如何，如果您进行安装和配置，则 `sudo` 可以以包括root用户在内的任何用户身份运行时使用它。

```
# If sudo is installed and configured
sudo apt-get update
sudo apt-get install <package>
```

但是，如果您**重建**容器，则必须**重新**安装手动**安装的**所有内容。为避免此问题，可以在中的 `postCreateCommand` 属性中使用一系列命令，也可以在自定义Dockerfile中 `devcontainer.json` 使用 `RUN` 指令。您可以用来将多个命令串在一起。 `&&`

使用Dockerfile：

```
RUN apt-get update && apt-get install <packaging>
```

使用 `devcontainer.json`：

```
"postCreateCommand": "apt-get update && apt-get install <package>"
```

或者，如果以非root用户 (/docs/remote/containers-advanced#_adding-a-nonroot-user-to-your-dev-container)身份运行：

```
"postCreateCommand": "sudo apt-get update && sudo apt-get install <package>"
```

在 `postCreateCommand` 一旦容器运行的运行，所以你也可以使用属性像运行命令 `npm install` 或在你的源代码树来执行shell脚本（如果你已经安装的话）。

```
"postCreateCommand": "bash scripts/install-dev-tools.sh"
```

### 使用Docker Compose

在某些情况下，仅一个容器环境是不够的。幸运的是，远程-容器支持Docker Compose (https://docs.docker.com/compose/)管理的多容器配置。

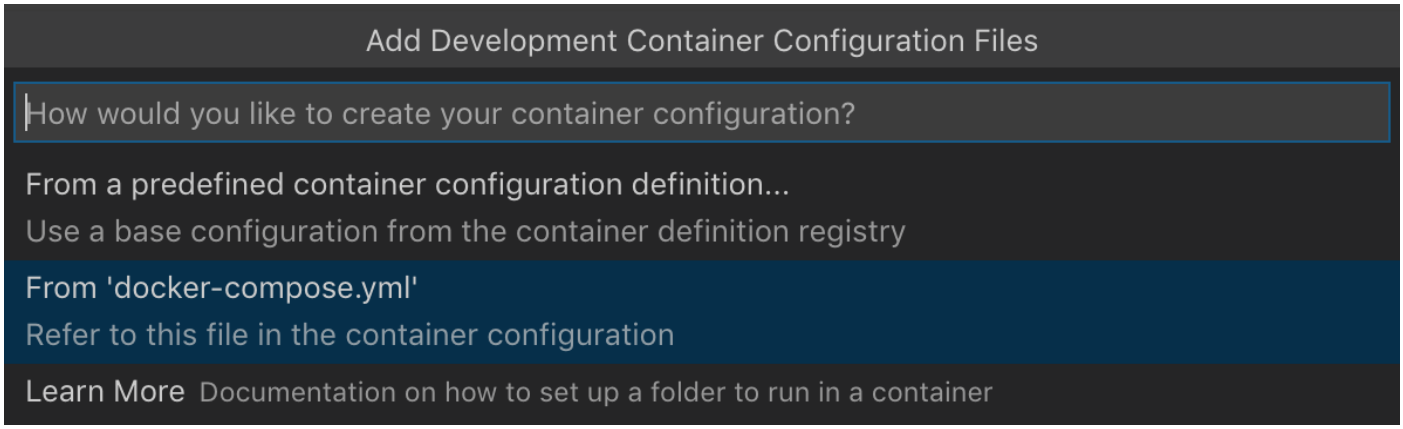
您可以：

1. 使用现有的未修改中定义的服务 `docker-compose.yml` 。
2. 创建 `docker-compose.yml` 用于开发服务的新文件（或复制现有文件）。
3. 扩展您现有的Docker Compose配置以开发服务。
4. 使用单独的VS Code窗口一次使用多个Docker Compose定义的服务 (/docs/remote/containers-advanced#_connecting-to-multiple-containers-at-once)。

> **注意：**使用Alpine Linux容器时，由于 `glibc` 扩展内部代码的依赖性，某些扩展可能无法正常工作。

可以将VS Code配置为**自动启动** Docker Compose文件中特定服务的**任何所需容器**。如果您已经使用命令行启动了已配置的容器，则VS Code将**附加到**您指定**的正在运行的服务上**。这为您的多容器工作流提供了上面针对Docker映像和Dockerfile流所述的相同的快速设置优势，同时仍允许您根据需要使用命令行。

为了快速上手，请在VS Code中**打开要使用的文件夹**，然后在命令面板（ `F1` ）中运行“ **远程容器：添加开发容器配置文件**...”命令。




系统会要求您选择一个现有的Docker Compose文件（如果存在），或者从vscode-dev-containers存储库 (https://github.com/Microsoft/vscode-dev-containers)中选择一个预定义的容器配置，该存储库 (https://github.com/Microsoft/vscode-dev-containers)基于文件夹的内容进行排序。这些“开发容器定义”中的许多都使用Dockerfile，因此请选择以下定义之一作为Docker Compose的起点：现有Docker Compose (https://aka.ms/vscode-remote/samples/existing-docker-compose)，Node.js和MongoDB (https://aka.ms/vscode-remote/samples/node-mongo)，Python＆PostgreSQL (https://aka.ms/vscode-remote/samples/python-postgres)或Docker-from-Docker Compose (https://aka.ms/vscode-remote/samples/docker-from-docker-compose)。选择后，VS Code将适当的文件 `.devcontainer/devcontainer.json` （或 `.devcontainer.json` ）添加到该文件夹中。

您也可以手动创建配置。要重复使用未经修改的Docker Compose文件，可以使用中的 `dockerComposeFile` 和 `service` 属性 `.devcontainer/devcontainer.json` 。

例如：

```
{
  "name": "[Optional] Your project name here",
  "dockerComposeFile": "../docker-compose.yml",
  "service": "the-name-of-the-service-you-want-to-work-with-in-vscode",
  "workspaceFolder": "/default/workspace/path/in/container/to/open",
  "shutdownAction": "stopCompose"
}
```

看到devcontainer.json参考信息可用的其他属性，如 `workspaceFolder` 和 `shutdownAction` 。

将 `.devcontainer/devcontainer.json` 文件添加到文件夹后，从命令面板（ `F1` ）运行“ **远程容器：在容器中重新打开文件夹**”命令（或，如果尚未使用VS Code ，则运行“ **远程容器：在容器中打开文件夹**... ”） 。

如果容器尚未运行，则 `docker-compose -f ../docker-compose.yml up` 在此示例中将调用VS Code 。该 `service` 属性指示Docker Compose文件VS Code中的哪个服务应连接，而不是启动哪个服务。如果您手动启动它们，则VS Code将附加到您指定的服务。

您还可以创建Docker Compose文件的开发副本。例如，如果您有 `.devcontainer/docker-compose.devcontainer.yml` ，则只需在中更改以下行 `devcontainer.json` ：

```
"dockerComposeFile": "docker-compose.devcontainer.yml"
```

但是，一种更好的方法通常是通过**将** Docker Compose文件**扩展为另一个**来避免复制该文件。在下一部分中，我们将扩展Docker Compose文件。

为了避免在默认容器命令失败或退出时关闭容器，您可以为您在其中指定的服务修改Docker Compose文件， `devcontainer.json` 如下所示：

```
# Overrides default command so things don't shut down after the process ends.
command: /bin/sh -c "while sleep 1000; do :; done"
```

如果尚未这样做，则可以使用Docker Compose文件中 (https://docs.docker.com/compose/compose-file/#volumes)的卷列表 (https://docs.docker.com/compose/compose-file/#volumes)**将本地源代码“绑定”安装**到容器中 (https://docs.docker.com/compose/compose-file/#volumes)。

例如：

```
volumes:
  # Mounts the project folder to '/workspace'. The target path inside the container
  # should match what your application expects. In this case, the compose file is
  # in a sub-folder, so we will mount '..'. You would then reference this path as the
  # 'workspaceFolder' in '.devcontainer/devcontainer.json' so VS Code starts here.
  - ..:/workspace:cached
```

但是，在Linux上，使用绑定安装时，您可能需要设置并**指定非root用户**，否则您创建的任何文件都是root。有关详细信息，请参见将非root用户添加到您的dev容器 (/docs/remote/containers-advanced#_adding-a-nonroot-user-to-your-dev-container)。要以其他用户身份运行VS Code，请将其添加到 `devcontainer.json` ：

```
"remoteUser": "your-user-name-here"
```

如果您希望所有进程以不同的用户身份运行，请将其添加到Docker Compose文件中的相应服务中：

```
user: your-user-name-here
```

如果您没有创建用于开发的自定义Dockerfile，则可能需要安装其他开发人员工具，例如 `curl` 在服务容器内。尽管效率不如将这些工具添加到容器映像中，但您也可以将 `postCreateCommand` 属性用于此目的。

```
"postCreateCommand": "apt-get update && apt-get install -y curl"
```

或者，如果以非root用户身份运行并且 `sudo` 安装在容器中：

```
"postCreateCommand": "sudo apt-get update && sudo apt-get install -y curl"
```

有关使用安装软件的更多信息，请参阅安装其他 `apt-get` 软件。

如果您的应用程序是使用C ++，Go或Rust或使用基于ptrace的调试器的另一种语言构建的，则还需要将以下设置添加到Docker Compose文件中：

```
# Required for ptrace-based debuggers like C++, Go, and Rust
cap_add:
- SYS_PTRACE
security_opt:
- seccomp:unconfined
```

首次创建容器后，需要运行**Remote-Containers：Rebuild Container**命令以使 `devcontainer.json` ， Docker Compose文件或相关Dockerfile的更新生效。

### 扩展Docker Compose文件以进行开发

引用现有的部署/非开发重点 `docker-compose.yml` 存在一些潜在的缺点。

例如：

- 如果容器的入口点关闭，则Docker Compose将关闭容器。这对于您正在调试并且需要反复重新启动应用程序的情况来说是有问题的。
- 您也可能没有将本地文件系统映射到容器中，也可能没有将端口暴露于要访问的其他资源（如数据库）中。
- 您可能需要将本地 `.ssh` 文件夹的内容复制到容器中，或者设置上面使用Docker Compose中所述的ptrace选项。

您可以通过使用多个 `docker-compose.yml` (https://docs.docker.com/compose/extends/#multiple-compose-files)覆盖或补充主要文件的文件 (https://docs.docker.com/compose/extends/#multiple-compose-files)扩展整个Docker Compose配置，从而解决这些问题以及类似的问题。

例如，考虑以下附加 `.devcontainer/docker-compose.extend.yml` 文件：

```
version: '3'
services:
  your-service-name-here:
    volumes:
      # Mounts the project folder to '/workspace'. While this file is in .devcontainer,
      # mounts are relative to the first file in the list, which is a level up.
      - .:/workspace:cached

    # [Optional] Required for ptrace-based debuggers like C++, Go, and Rust
    cap_add:
      - SYS_PTRACE
    security_opt:
      - seccomp:unconfined

    # Overrides default command so things don't shut down after the process ends.
    command: /bin/sh -c "while sleep 1000; do :; done"
```

该文件可以根据需要提供其他设置，例如端口映射。要使用它，请按特定顺序引用您的原始 `docker-compose.yml` 文件 `.devcontainer/devcontainer.extend.yml`：

```
{
  "name": "[Optional] Your project name here",

  // The order of the files is important since later files override previous ones
  "dockerComposeFile": ["../docker-compose.yml", "docker-compose.extend.yml"],

  "service": "your-service-name-here",
  "workspaceFolder": "/workspace",
  "shutdownAction": "stopCompose"
}
```

然后，在启动任何容器时，VS Code将**自动使用这两个文件**。您还可以从命令行自己启动它们，如下所示：

```
docker-compose -f docker-compose.yml -f .devcontainer/docker-compose.extend.yml up
```

尽管该 `postCreateCommand` 属性允许您在容器内安装其他工具，但在某些情况下，您可能希望具有特定的Dockerfile进行开发。您也可以使用相同的方法来引用 `Dockerfile` 专门用于开发的自定义，而无需修改现有的Docker Compose文件。例如，您可以更新 `.devcontainer/devcontainer.extend.yml` 如下：

```
version: '3'
services:
  your-service-name-here:
      # Note that the path of the Dockerfile and context is relative to the *primary*
      # docker-compose.yml file (the first in the devcontainer.json "dockerComposeFile"
      # array). The sample below assumes your primary file is in the root of your project.
      build:
        context: .
        dockerfile: .devcontainer/Dockerfile
      volumes:
        - .:/workspace:cached
      command: /bin/sh -c "while sleep 1000; do :; done"
```

#### Docker Compose开发容器定义

以下是使用Docker Compose的开发容器定义：

- 现有Docker Compose- (https://aka.ms/vscode-remote/samples/existing-docker-compose)包括一组文件，您可以将它们放到现有项目中，这些 `docker-compose.yml` 文件将在项目根目录中重用该文件。
- Node.js和MongoDB- (https://aka.ms/vscode-remote/samples/node-mongo)一个Node.js容器，它连接到另一个容器中的Mongo DB。
- Python＆PostgreSQL- (https://aka.ms/vscode-remote/samples/python-postgres)在另一个容器中连接到PostGreSQL的Python容器。
- Docker-from-Docker Compose- (https://aka.ms/vscode-remote/samples/docker-from-docker-compose)包括Docker CLI，并说明了如何通过批量安装Docker Unix套接字，从开发容器内部使用它来访问本地Docker安装。

### 先进的容器配置

请参阅“ 高级容器配置” (/docs/remote/containers-advanced)文章，以获取有关以下主题的信息：

- 添加环境变量 (/docs/remote/containers-advanced#_adding-environment-variables)
- 添加另一个本地文件挂载 (/docs/remote/containers-advanced#_adding-another-local-file-mount)
- 更改或删除默认源代码安装 (/docs/remote/containers-advanced#_changing-the-default-source-code-mount)
- 改善容器磁盘性能 (/docs/remote/containers-advanced#_improving-container-disk-performance)
- 向您的开发容器添加非root用户 (/docs/remote/containers-advanced#_adding-a-nonroot-user-to-your-dev-container)
- 避免在容器重建时扩展安装 (/docs/remote/containers-advanced#_avoiding-extension-reinstalls-on-container-rebuild)
- 设置Docker Compose的项目名称 (/docs/remote/containers-advanced#_setting-the-project-name-for-docker-compose)
- 从容器内部使用Docker或Kubernetes (/docs/remote/containers-advanced#_using-docker-or-kubernetes-from-a-container)
- 一次连接到多个容器 (/docs/remote/containers-advanced#_connecting-to-multiple-containers-at-once)
- 在远程Docker Machine或SSH主机上的容器内部进行开发 (/docs/remote/containers-advanced#_developing-inside-a-container-on-a-remote-docker-host)
- 减少Dockerfile构建警告 (/docs/remote/containers-advanced#_reducing-dockerfile-build-warnings)