

開発環境は全て
コンテナの中へ



VS Code DevContainer Guidebook

Atsushi Morimoto(@74th)

VS Code Dev Container Guidebook

Atsushi Morimoto (@74th)

目次

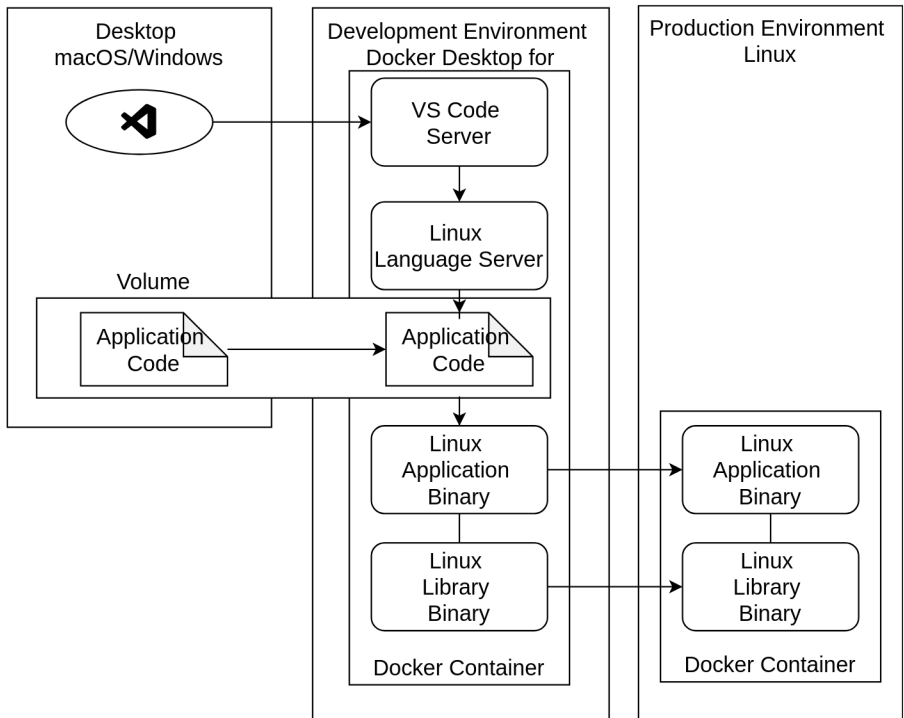
1. コンテナ時代の開発の新常識 Remote Container とは	6
2. 本格的に使い始める	12
2.1 Remote Container 機能の拡張機能をインストールする	
2.2 リポジトリに最初の設定ファイルを追加する	
2.3 フォルダを Dev Container で開き直す	
2.4 リポジトリを Volume にチェックアウトして開く	
2.5 Docker コンテナを別に立ち上げてアタッチする	
2.6 過去に開いた Dev Container を再度開く	
2.7 Dev Container を削除する	
2.8 Visual Studio / Github Codespaces を使う	
3. Dev Container の責務を考える	23
3.1 Dev Container とは何であるか	
3.2 もっと雑な Dev Container を考える	
3.3 より有用な Dev Container の責務	
4. 実践 Dev Container 構築	31
4.1 設定ファイル devcontainer.json でできること	
4.2 Dev Container に必要なパッケージや設定を汎用スクリプトで入れる	
4.3 Multi Staged Build を使ってアプリケーション実行環境と Dev Container を 1 つの Dockerfile で構築する	
4.4 開発に使うサイドカーを docker-compose で同時に立ち上げる	
4.5 開発に特化した Dev Container を運用する	
4.6 GCP などホストの権限を Dev Container に持ち込む	
4.7 パスを追加する	
5. Remote Container 機能を使う環境について	55
5.1 OS が Linux の場合	
5.2 macOS / Windows で Docker Desktop を利用する場合	
5.3 外部ホスト、もしくは仮想マシンの Docker を使う場合	
5.4 Visual Studio / Github Codespaces を使う場合	
5.5 Codespaces の Self Hosted Machine を使う場合	
6. Tips	63
6.1 dotfiles を一緒にインストールする	

6.2 個人的に使う拡張機能をインストールする	
7. Dev Container 応用実例集	65
7.1 Go: 最初はテンプレートから始める	
7.2 Go: GOPATH の外で設定する	
7.3 Go: 依存モジュールの取得を Dev Container のビルドで行う	
7.4 Go: 作成した Dev Container を Github Packages で共有する	
7.5 Go: Dev Container と ビルド時に使うイメージを一致させる	
7.6 Python: GPU を有効にした Jupyter ノートブック環境の構築	
終わりに	84
VS Code 関連著書の紹介	
著者について	89

1. コンテナ時代の開発の新常識 Remote Container とは

昨今の Software の動作環境は、多くが Docker コンテナや関数単位になってきているように感じます。Docker コンテナは、本番運用環境にて Kubernetes や Amazon Elastic Container Service でのアプリケーションの動作単位として使われます。しかし、開発で使うマシンが macOS や Windows であるため、開発環境はまだ Docker コンテナになっていないことが多いのではと思います。そのため、Software の本番運用環境は Linux システム上であるにもかかわらず、開発環境ために macOS や Windows 用のライブラリを用意すること多いと思われます。

VS Code の機能の一つに、「リモート開発機能」があります。VS Code の多くのプログラミング言語の機能は、拡張機能を介して Language Server を使って行われます。リモート開発機能では、その Language Server を含む拡張機能を、VS Code Server と呼ばれるプロセスがリモートマシン上で実行します。VS Code の画面からはこのリモートマシン上の VS Code Server に接続することで、リモートマシンの環境を使った開発がローカルマシンでの開発と同様に行えるようになります。



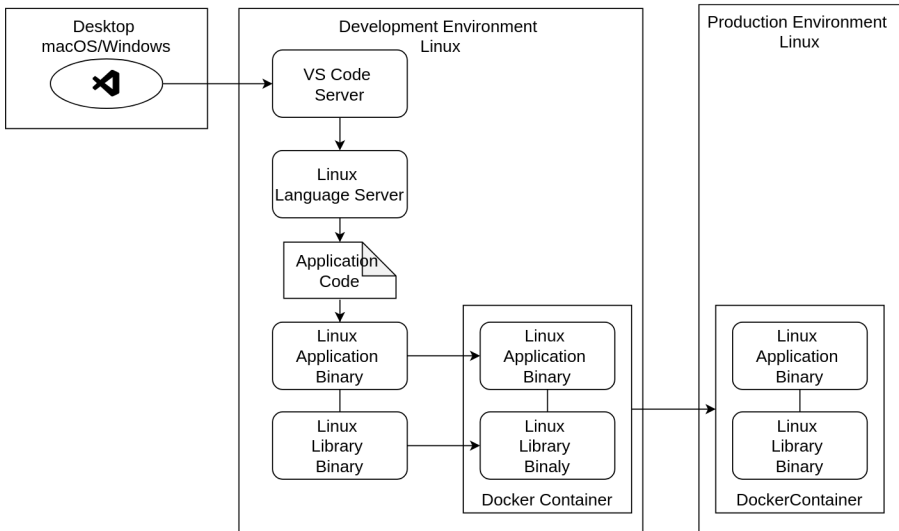
▲リモート開発機能

リモート開発機能は、SSH 接続経由でリモートマシンを使う Remote SSH 機能と、Windows10 上で Linux カーネルを動かす WSL を使った Remote WSL 機能があり、それらでも非常に便利に使えます。筆者は仕事の開発環境は、MacBookPro で仮想マシン実行ソフトウェアである Parallels Desktop を使って Ubuntu の仮想マシンを実行し、リモート開発機能の SSH 機能を使って開発をしています。この環境の詳しい話は、技術書典 7 にて『Visual Studio Code Remote dev & Cloud Code Guide』という本を頒布して紹介しました。Booth.pm *1 でも頒布を続けておりますので、よろしければ購入ください。

本書で取り上げたいトピックはリモート開発機能のうちの、Remote Container 機能です。

*1 <https://74th.booth.pm/items/1575560>

1. コンテナ時代の開発の新常識 Remote Container とは



▲Remote Container機能

Remote Container 機能は、開発も Docker コンテナの中で行うものです。この時に使う Docker コンテナを Dev Container と呼びます。Dev Container の中にソースコードを含むワークスペースをマウントし、さらに VS Code Server を動かして、Docker コンテナの外の VS Code から繋がります。

Remote Container 機能が優れているのは以下の点です。

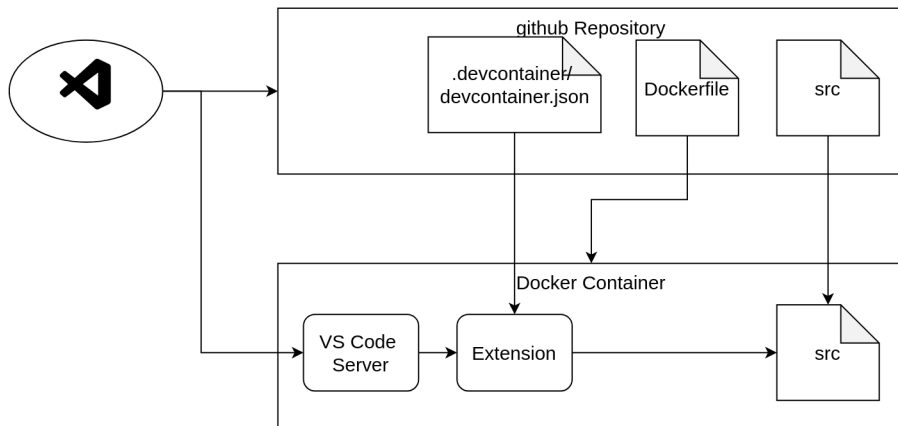
- 環境構築が Dockerfile というソースコードに含まれるシェルスクリプトで完結する。
- 本番環境の Docker コンテナの Dockerfile を流用することで、本番環境に近い環境で開発を進めることができる。
- 開発環境を簡単に準備でき、使い捨てることもできる。

Docker コンテナは"必ず"この Dockerfile という拡張性が高いとはいえないシェルスクリプトを用いてビルドされます。つまり、開発環境を構築するための手順は、手順書といった文章ではなく、コードで記述することを強制されます。この Dockerfile のよしあしについては議論がありますが、筆者はむしろ Dockerfile の拡張性が高くないことが、隠蔽されている部分や機能の副作用が少なく、汎用的な環境構築手法として優れていると考えています。

また、Docker コンテナは本番環境で使われるため、そのための Dockerfile は必ず作成します。Dev Container は、本番環境で使われる Dockerfile が、そのままとは言えませんが近い形で流用することができます。

Docker コンテナは、環境間で容易に運ぶことができ、さらに作成や削除も 1 コマンドで即座に行うことができるので、その速さは仮想マシンより明らかに優れています。

そして Remote Container 機能を使って開発を始めるためには、既にリポジトリに Dev Container の設定や Dockerfile が含まれている場合、必要なことは"リポジトリをコンテナの中で開くための 1 コマンドを実行すること"だけです。なぜならば、このリポジトリの中に必要なものが全て含まれているため、VS Code が自動的に Dev Container を構築し、更に拡張機能をインストールするので、すぐに開発がスタートできる状態になります（ただし設定を作り込む必要があります）。



▲Remote Container機能での環境構築

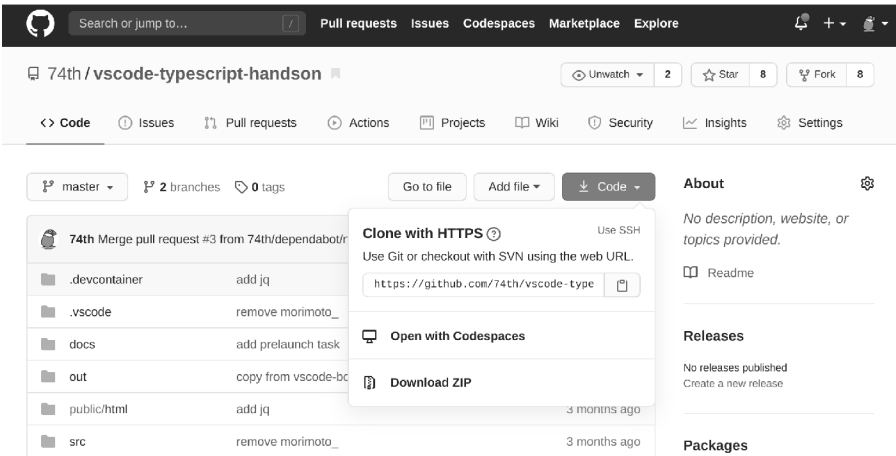
Microsoft は「Visual Studio Codespaces *2 」、「Github Codespaces」という、この Remote Container 機能の環境をクラウドで提供するサービスを開始しました。Web ブラウザで開発が完結することに注目が集まりがちですが、私が注目しているのは必要なスペックの計算資源を時間単位でレンタルし、不必要になればすぐに解約できることです。従来は開発するためには高スペックのマシンを用意したり、開発環境では小さいデータサイズを用意して動かせるようにしていました。クラウド上で計算を実行する場合にも、仮想マシンを使っていたため、環境構築には手数が必要でした。それらがすべて Remote Container 機能と Github Codespaces に集約され、開発環境としてより柔軟にクラウドの計算資源を利用できるようになったといえます。

本書の執筆時現在（2020/09/06）、Github Codespaces は Closed Beta としてサービスが提供されています。Github Codespaces が有効になっている場合、Github のリポジトリのページには "Open with Codespaces" ボタンが追加されます。このボタンを押すと、このリポジトリの Dev Container の設定を使って、Dev Container が立ち上がり、ブ

*2 Visual Studio Codespaces は 2021 年 2 月でクローズし、Github Codespaces に統合されることが発表されています。

1. コンテナ時代の開発の新常識 Remote Container とは

ブラウザ上の Visual Studio Code から開発を行うことができるようになります。環境構築に必要なことはこの 1 ボタンだけであり、それ以外は必要ありません。



▲Github Codespacesの開始ボタン

筆者は、Dev Container から始まるリモート開発機能と、Github のリポジトリ、そして Azuer のクラウドサービスがピタリと結びついたことを知った時、新しい時代が来たなと感動しました。Microsoft が VS Code をオープンソースのまま開発に力を注ぎ続けていること、収益の柱が OS のライセンス販売から Azure のクラウドサービスに変わっていったこと、Github を傘下に収めたこと、リモート開発機能を立ち上げたこと、これらの別々のことが繋がってひとつ世界になったと感じました。Docker コンテナが登場した時以来の感動でした。

Microsoft は現在もかなりの工数をかけて VS Code を開発し続けています。リモート開発機能は残念ながらオープンソースではありません^{*3}が、Visual Studio / Github Codespaces として事業のマネタイズに成功することを、いち利用者として期待しています。

本書は、この Remote Container 機能の解説書になります。Remote Container 機能を解説するだけでなく、開発で使い始めたり、よりよく使うためのノウハウを提供します。それらの情報は VS Code の Remote Container 機能だけではなく、Visual Studio / Github Codespaces でも活用可能だろうと思います。本書によって、Remote Container 機能を使いこなし、自身の開発環境をアップデートしていただければと思います。

筆者は Remote Container 機能の直接の開発者ではなく、いち利用者過ぎません。Remote Container 機能は実は現在でも Preview の拡張機能となっており、Visual Studio

^{*3} オープンソースの Code にはリモート開発機能は含まれていません

/ Github Codespaces のサービスについては Beta サービスが始まったばかりです。そのため本書の内容は、今後の Remote Container 機能、Visual Studio / Github Codespaces で使えることを保証するものではないことをご了承下さい。

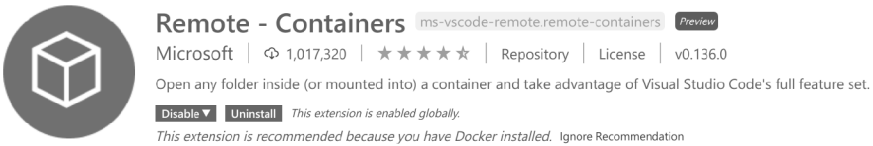
Visual Studio Code (VS Code) は、Microsoft 社の製品であり、オープンソースとして公開されていますが、著作権や製品は Microsoft 社保有のものです。本書は、VS Code 及び Visual Studio / Github Codespaces のファンブックです。本書の内容について、直接 Microsoft 社、Github 社、Github 上の Issue に問い合わせることはおやめ下さい。本書の内容の変更や本書の頒布の中止は、断りなく行われることがあります。

2. 本格的に使い始める

この章では実際に使い方をパターンに分けて解説したいと思います。

2.1 Remote Container 機能の拡張機能をインストールする

VS Code の リモート開発機能は拡張機能として提供されており、拡張機能 "Remote - Containers" をインストールすることで初めて使えるようになります。



▲Extension Remote Containers

コマンドで行う場合には、以下を実行します。

```
code --install-extension ms-vscode-remote.remote-containers
```

2.2 リポジトリに最初の設定ファイルを追加する

Remote Container 機能の設定ファイルは .devcontainer/devcontainer.json ファイルです。Git リポジトリにこのファイルを追加すれば Remote Container 機能を使うことができます。

Remote Container 機能の設定ファイルを最初にする時には、Microsoft 社の提供するテンプレートを使うことができます。既に Dockerfile を作っている場合を除いては、このテンプレートを使うと良いでしょう。設定ファイルを追加するには、コマンド "Remote-Containers: Add Development Container File ..." を実行します。

これを実行すると、以下の 2 つのファイルが作られます。

```
- .devcontainer
  |- devcontainer.json
  `-- Dockerfile
```

2.3 フォルダーを Dev Container で開き直す

.devcontainer/devcontainer.json のあるリポジトリを既にチェックアウトしている場合、このリポジトリを Dev Container で開き直すコマンドから、Remote Container 機能を使い始めることができます。 コマンド "Remote-Containers: Reopen Folder in Container" を実行します。

すると、VS Code が開き直され、以下のシーケンスでコンテナが起動します。

1. Dockerfile を指定している場合、Dockerfile のビルド
2. コンテナの起動
3. .gitconfig など設定のインストール
4. VS Code Server のインストール
5. 拡張機能のインストール
6. postCreateCommand の実行
7. コンテナ内のフォルダーを VS Code で開く

このコマンドで開いたフォルダーは、Docker の Volume を使って、ホストのディレクトリをマウントしています。 そのためコンテナの中、もしくは外で行った変更は、コンテナの外とコンテナの中の 2 つが同時に書き換えられたように見えます。

なお、VS Code を閉じた場合は、一度ホストのフォルダーを開いて"Reopen Folder in Container"を実行するか、Remote タブを開き、その中から目的のコンテナを選択します。一度ビルドすると、再起動時にはビルドを行いません。

devcontainer.json や Dockerfile を書き換えた場合は、コマンド "Remote-Containers: Rebuild Container" を実行します。 Dockerfile だけではなく、devcontainer.json を書き換えた場合にもこのコマンド実行する必要があります。

コマンド "Remote-Container: Reopen Folder in Container" では、利用する docker サービスでローカルフォルダーをマウント可能なことが必要です。 Linux を使用してローカルの docker サービスを利用している場合、もしくは docker Desktop for mac/Windows を利用している場合は特に問題はありません。 いっぽう、外部ホストの docker サービスを使っている場合には、ローカルのディレクトリをマウントすることはできないため、この機能を使って使用を開始することはできません。 詳細は 5.2. 節で解説しますが、docker Desktop for mac を使っている場合にはストレージの性能問題が発生しやすいため、この使い方が最適とはいえないものになっています。

2.4 リポジトリを Volume にチェックアウトして開く

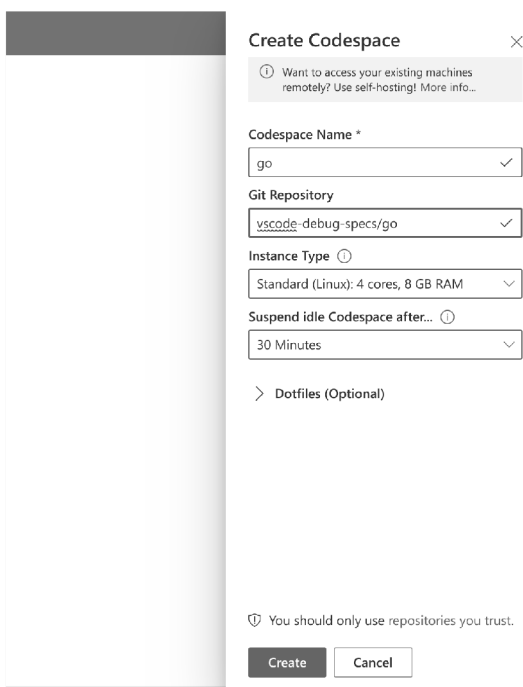
前節では既にチェックアウトされたフォルダーで使う方法を説明しましたが、チェックアウトし直しても良い場合コマンド"Remote-Containers: Open Repository in Container"が使えます。 この機能を使うとリポジトリは docker サービスの docker

2.8 Visual Studio / Github Codespaces を使う

Remote Container 機能で用意した devcontainer.json は Web 版の VS Code である Visual Studio / Github Codespaces でも使うことができます。

Codespaces の利用のやり方は 2.3 節で解説した、コマンド"Remote-Container: Open Repository in Container" と同じように使うことができます。Codespaces では、立ち上げているインスタンスのことを "Codespace" と呼びます。

筆者は Remote Container 機能のクラウドサービスと認識しています。必要な CPU、メモリのインスタンスをオンデマンドで利用することができます。Visual Studio Codespaces の利用には、2020/09/06 現在は Azure のアカウントが必要になります。Github Codespaces は、2020/09/06 現在は Beta サービスであり、招待が必要です。



Create Codespace

① Want to access your existing machines remotely? Use self-hosting! More info...

Codespace Name *

go ✓

Git Repository

vscode-debug-specs/go ✓

Instance Type ①

Standard (Linux): 4 cores, 8 GB RAM ✓

Suspend idle Codespace after... ①

30 Minutes ✓

> Dotfiles (Optional)

⚠ You should only use repositories you trust.

Create Cancel

▲Visual Studio Codespaces の Codespace 作成画面

拡張機能 Codespaces をインストールすると、VS Code のコマンドから Codespace を作成したり、立ち上げた Codespace に接続することができます。

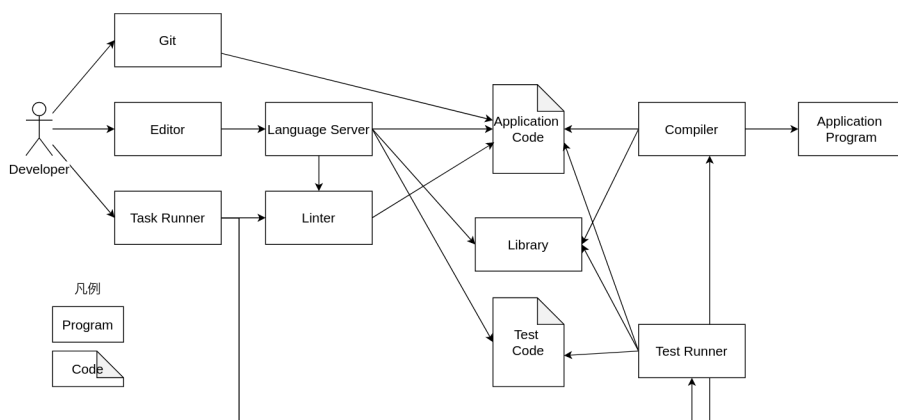
3. Dev Container の責務を考える

前章では Remote Container 機能の概要を説明し、Dev Container 内で VS Code Server を起動して接続して開発する環境であることを説明しました。本章では、この中心である Dev Container とは何なのかをみていきたいと思います。

3.1 Dev Container とは何であるか

Remote Container 機能では、開発に必要なすべての操作を Dev Container 内で行います。この節では一般的な VS Code を用いた開発時で行いたいことを列挙し、それは Dev Container で担わせるにはどうすべきかを考えてみたいと思います。

まず、コードを書く時に使うものを図示すると以下ようになります。



▲コードを書く時に使うもの

この中で開発者（以下、Developer）は次のことをしています。

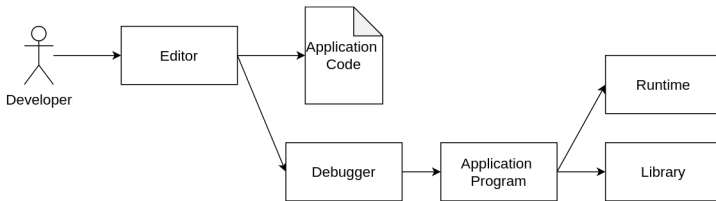
1. Editor を通じて、Language Server の支援を受けながら、Application Code を記述すること。
2. Task Runner を通じて、Compiler を実行し、Application Code と Library から、Application Program をビルドすること。
3. Task Runner を通じて、Linters など静的解析ツールを実行し、Application Code を分析すること。
4. Task Runner から実行される Test Runner を通じて、Test Code を実行すること。

3. Dev Container の責務を考える

最近の VS Code をはじめとするエディターでは、言語に依存した解析機能は Language Server を利用することが多くなっています。開発時に実行するビルドやテスト、Linter 等解析ツールの実行は、その言語に特化したタスクランナーを使うことが主流です（JavaScript における npm script、Java における Gradle など）。Linter の機能は Language Server を介して実行されるケースも多くなりました。

最近の Application はその Application Code からなる Application Program 単体で動作することは少なく、なんらかの Library を用いてビルド、および実行されます。

一方、ビルドされた Application Program をデバッグ実行する環境を図示すると以下のようになります。



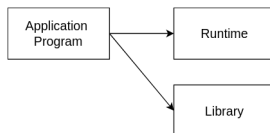
▲デバッグ時に使うもの

この中で Developer は次のことをしています。

1. Editor から Debugger を通じて、Application Program を解析しながら実行すること

デバッグ作業の中でも、デバッグ実行（Application Program を一時停止させながらステップ実行させること）の際に使うものを図示しました。今まで IDE の専売特許だったデバッグ実行は、VS Code も備えており、そのことが VS Code の大きな特徴のひとつとなっています。

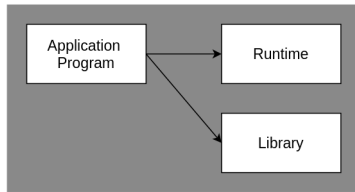
では、最後に Application Program を実行する時の動作を図示します。



▲実行時に使うもの

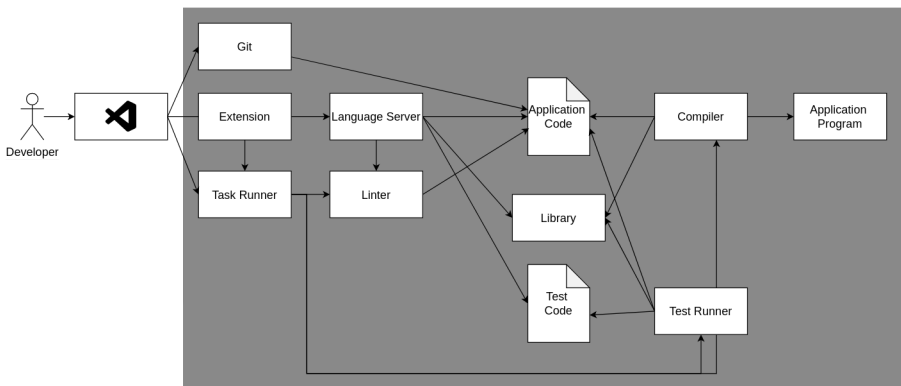
先にも述べたように、多くの Application Program は単体で動作することは少なく、なんらかの Library と一緒に動作します。

今まで Docker コンテナが担ってきたのはこの"実行時"のみであり、Docker コンテナの中身は以下のように、実行時に必要な Library と Runtime（Java における JVM や、Python の Python プログラムなど）のみが Application Program と一緒に格納されていました。



▲今までDockerコンテナが担ってきたこと

VS Code の Remote Container 機能では、開発の殆どのプロセスを Docker コンテナの中で行います。Docker コンテナの中で実行する範囲を図示すると以下のようになります。



▲Remote Container機能でDockerコンテナの実行時に中にあるもの

エディタ以外のほとんどの機能が Docker コンテナの中で行われます。

最終的にこの状態にいたるために、Dev Container として Docker コンテナを作る時に必要なことを考えています。

Dev Container 自体には、ソースコードを含む全てを格納したコンテナを作成しなければいけないわけではありません。Remote Container 機能では、Dev Container を Docker コンテナとして起動する時に以下を行います。

1. Code を Volume としてマウントする。
2. VS Code Server をインストールする。
3. VS Code Extension をインストールする。

さらに先に図示したもののうち、Application Program は成果物にあたります。よって、残りのものが Dev Container に格納すべきものにあたります。これを図示すると以下のようにになります。

4. 実践 Dev Container 構築

この章では、実践的な Dev Container の Dockerfile や設定ファイル devcontainer.json の書き方について解説します。

4.1 設定ファイル devcontainer.json でできること

Dev Container の設定ファイルである devcontainer.json を使ってできることを解説します。

Dockerfile、イメージの指定

Dev Container として使う Docker イメージには、Dockerfile を同梱してその場でビルドする方法と、Docker イメージのタグを指定してレジストリから pull する方法の 2 つがあります。

レジストリに Dockerfile を同梱してある場合には、以下のように build プロパティに記述します。ここで記述する Dockerfile のパスは、.devcontainer ディレクトリからの相対パスであることに注意して下さい。

```
// .devcontainer/devcontainer.json
{
  "build": {
    // Dockerfileのパス (.devcontainer/ディレクトリからの相対パス)
    "dockerfile": "Dockerfile",
    "context": "..",
    // ビルド時引数
    "args": { "VARIANT": "3.8" }
  },
  ...
}
```

この場合、"Remote-Container: Reopen in Container"など Remote Container 機能で開くコマンドを実行した際に初めてビルドが走ります。build オブジェクトには、target プロパティで multi staged build の対象を指定することができます (4.2 節参照)。

既にビルドされているイメージをコンテナレジストリからダウンロードするようにも設定できます。その場合には、build プロパティの代わりに、image プロパティを指定します。

4.2 Dev Container に必要なパッケージや設定を汎用スクリプトで入れる

Dev Container に必要なパッケージのインストールや、ユーザの追加などの便利な処理がまとまったあスクリプトが Microsoft から提供されています。このスクリプトは Dev Container のサンプルと同じリポジトリに格納されています。

- **Development Container Scripts**

<https://github.com/microsoft/vscode-dev-containers/tree/master/script-library>

`common-(distribution).sh` というスクリプトでは、以下のことを行ってくれます。

- 必要なパッケージ (curl や git など) のインストール
- ユーザ、グループの作成
- sudo を使えるようにする
- (option) zsh のインストール

このスクリプトはディストリビューションごとに用意されているため、もとにするコンテナに合わせて選択します。

- `common-debian.sh`: Debian/Ubuntu をもとにした Docker イメージ用
- `common-alpine.sh`: Alpine をもとにした Docker イメージ用
- `common-redhat.sh`: CentOS/Redhat/OracleLinux をもとにした Docker イメージ用

DockerHub で公開されている公式のイメージは、Debian か Alpine で提供されています。`python:3.8-buster` と、Debian のバージョンの名前(10:buster、11:bullseye、12:bookworm)が付いているものには `common-debian.sh` を使用し、`python:3.8-alpine` と alpine 付いているものには `common-alpine.sh` を使用します。

また、npm などのパッケージマネージャを使うためのスクリプトも提供されています。

- `git-lfs-debian.sh`: Git LFS を使う
- `{java,go,rust,node}-debian.sh`: {Java,Go,Rust,NodeJS} を使う
- `docker-{debian,redhat}.sh`: Docker in Docker など docker コマンド、docker-compose コマンドを使う
- `kubecthl-helm-debian.sh`: kubectl、helm を使う

これを使うには、curl でダウンロードして実行する、もしくはリポジトリにこのスクリプトをコピーして使います*3。コピーする場合、最初の curl のインストールも省くことができます。

5. Remote Container 機能を使う環境について

本章では、Remote Container 機能を実行する環境について解説します。Remote Container 機能では Docker 上で Dev Container を動かすことになるため、なんらかの Docker サービスが必要になります。

以下に OS ごとに紹介していきます。

5.1 OS が Linux の場合

Linux の場合、Docker をそのまま使うことができます。Docker がサービスとして起動すれば、追加の設定は特に必要ありません。

Ubuntu、Debian を使用している場合、パッケージマネージャで容易にインストールが可能です。

```
sudo apt update
sudo apt install -y docker.io
```

Docker にはユーザ権限で Docker エンジンを実動かす Rootless Docker ^{*1} がありますが、確認した限りではこれも Remote Container 機能での利用には問題が発生することがわかりました ^{*2}。

REPL 8、CentOS 8 の場合、パッケージマネージャでは Docker はインストールできませんが、Docker 互換のコンテナエンジン Podman をインストールすることができます。

```
sudo yum update
sudo yum -y install podman
```

Podman を使う場合、ユーザ設定で docker の代わりに podman を使うように設定します。

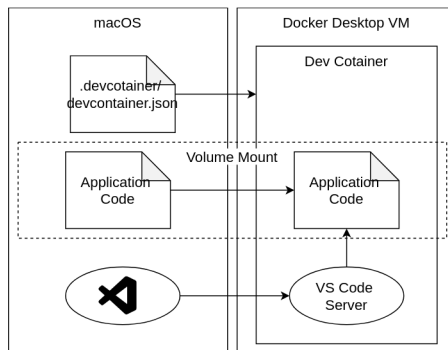
^{*1} <https://docs.docker.com/engine/security/rootless/>

^{*2} 通常 Docker コンテナ内のユーザ ID と、マウントしたホストディレクトリのユーザ ID は一致しますが、Rootless では一致しません。一致することを前提に作られている機能があるようです。

```
// User Settings
{
  "remote.containers.dockerPath": "podman"
}
```

5.2 macOS / Windows で Docker Desktop を利用する場合

Docker Desktop は macOS / Windows で docker の環境を容易に構築するソフトウェアです。Docker Desktop を macOS / Windows で使う場合、構成は以下の図ようになります。



▲macOSでDocker Desktop for Macを使う場合

Docker Desktop for Windows の場合、通常は Hyper-V 上に構築された VM を使います。アーキテクチャーとしては上記の macOS の場合と同様です。

Docker Desktop を使う場合、Docker Desktop をインストールする以外の作業は不要です。

Windows では Hyper-V の代わりに WSL を使うことができます。WSL を使う場合の利点は以下のとおりです。

- VM 用に先に CPU、メモリを確保しておく必要がない *3
- ファイルシステムが少し速い

Docker for Desktop で WSL を使うには、設定画面で "Use the WSL based engine" にチェックを付けます。

*3 執筆時点では、Linux の File Cache が際限なくメモリを使用する問題があるようです

7. Dev Container 応用実例集

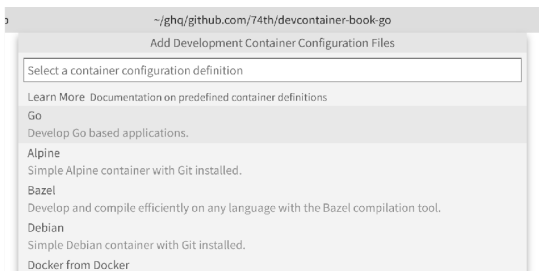
この章では具体的な用途に対して Dev Container を作っていく過程を解説します。

7.1 Go: 最初はテンプレートから始める

新しく Go で Web アプリケーションを作り始めることを想定した、Dev Container の環境作りを紹介します。新しいアプリケーションを作り始めるため、github.com 上にリポジトリを作成し空のリポジトリをチェックアウトした状態からスタートします。

```
$ git clone https://github.com/74th/devcontainer-book-go
```

Go の Dev Container のテンプレートは Microsoft によって用意されています*1。コマンド"Remote-Container: Add Development Container Configuration File..."を実行し、Go のテンプレートを選択します。



▲Goのテンプレート

すると、ワークスペースに Dev Container の設定ファイルと、Dockerfile が作られます。この Dockerfile は以下のように Microsoft が提供するコンテナがもとにするコンテナ (FROM 句) として設定されています。

```
# .devcontainer/Dockerfile
ARG VARIANT="1"
FROM mcr.microsoft.com/vscode/devcontainers/go:0-${VARIANT}
```

*1 <https://github.com/microsoft/vscode-dev-containers/tree/master/containers/go>

このコンテナ*2は、Docker Hub の`go`コンテナに、拡張機能 Go で使うライブラリを足したものになっています。

先の Dockerfile を見ると、VARIANT というコンテナビルド時のパラメータが設定されています。これは Go のランタイムのバージョンです。この値は Dev Container の設定ファイルで指定することができます。

```
// .devcontainer/devcontainer.json
{
  "name": "Go",
  "build": {
    "dockerfile": "Dockerfile",
    "args": {
      "VARIANT": "1",
      "INSTALL_NODE": "false",
      "NODE_VERSION": "lts/*"
    }
  }
  //...
}
```

`"VARIANT": "1"`で、現在の 1 系の最新バージョンが使われるようになります。Go ではバージョンの差分で困ることは少ないですが、チームでバージョンを揃えておく場合には、指定すると良いでしょう。ここでは現在 (2020/09/06) の最新バージョンの`"VARIANT": "1.15"`を指定しておきます。

ユーザ Root でファイルが作られるのを防ぐため、ユーザの追加をしておきます。

```
// .devcontainer/devcontainer.json
{
  // ...
  "remoteUser": "vscode"
}
```

コマンド"Remote-Container: Rebuild and Reopen in Container"を実行して、Dev Container を起動します。

ターミナルで Go のバージョンを確認すると、1.15 が使われています。

```
$ go version
go version go1.15 linux/amd64
```

*2 <https://github.com/microsoft/vscode-dev-containers/blob/master/containers/go/.devcontainer/base.Dockerfile>

この環境では環境変数 GOPATH が /go に設定され、書き込み可能なパーミッションが指定されています。

```
$ echo $GOPATH
/go

$ ls -al /go
total 16
drwxrwxrwx 4 root root 4096 Aug 12 00:22 .
drwxr-xr-x 1 root root 4096 Aug 30 03:49 ..
drwxrwxrwx 2 root root 4096 Aug 12 00:22 bin
drwxrwxrwx 2 root root 4096 Aug 12 00:22 src
```

GOPATH の下でフォルダーを展開する必要がある場合、"workspaceMount"と"workspaceFolder"を GOPATH の下である /go/src にマウントされるように設定します。

```
//.devcontainer/devcontainer.json
{
  // ...
  "workspaceMount":
    "source=${localWorkspaceFolder},target=/go/src/github.com/74th/devcontainer-book-go,type=bind",
  "workspaceFolder": "/go/src/github.com/74th/devcontainer-book-go"
}
```

7.2 Go: GOPATH の外で設定する

以前の Go は環境変数 GOPATH のディレクトリで、リポジトリのパスのディレクトリ作って、その中で開発する必要がありました。Go Modules が登場してからは、開発するアプリケーションのパスと依存するライブラリを go.mod ファイルに記述することで、GOPATH の下である必要はなくなりました。

Go Modules を使い始めるには以下のコマンドを実行します。

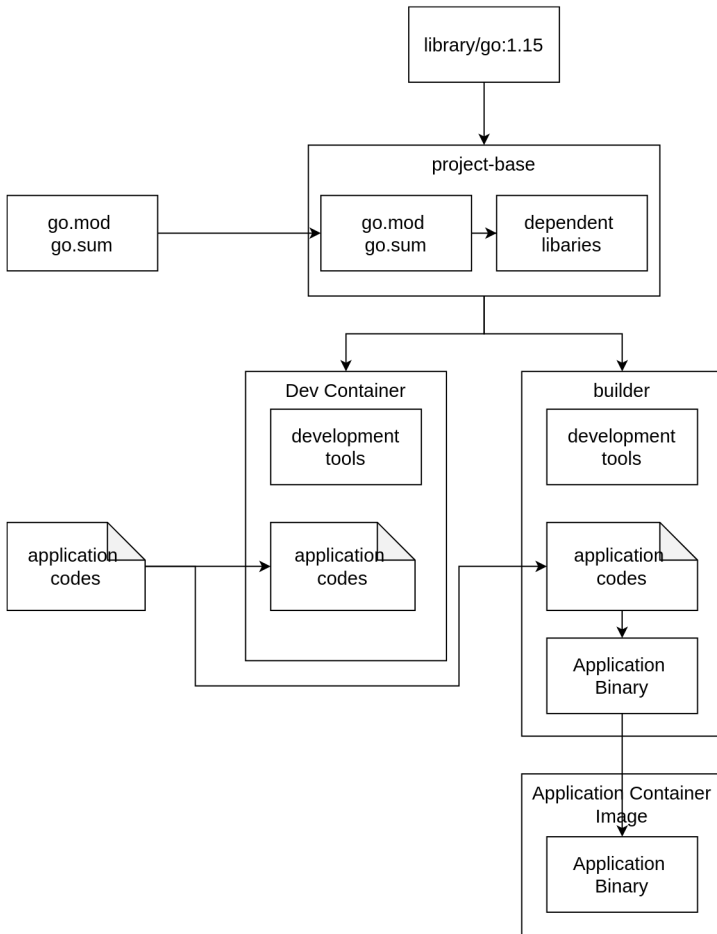
```
$ go mod init github.com/74th/devcontainer-book-go
go: creating new go.mod: module github.com/74th/devcontainer-book-go
```

すると、以下の内容の go.mod ファイルが作られます。

```
// go.mod
module github.com/74th/devcontainer-book-go
```

ここでは以下のような運用を考えてみました。

- Dev Container とビルド時に使うコンテナイメージは、同じコンテナイメージをもとにする
- Dev Container と CI 用のコンテナイメージを使う
- アプリケーションを実際動かすコンテナには、ソースコードやビルド環境を入れず、ビルド時に使うコンテナからアプリケーションバイナリだけを入れる。



▲Goのコンテナイメージ運用

これらのコンテナイメージを、4.3 節で紹介した Multi Staged Build という機能を使って、コンテナイメージのビルドを分岐させて、それぞれのコンテナを作ります。

終わりに

本書では、Dev Container での開発の魅力と、そのノウハウについて解説してきました。まだ、コンテナの中で開発する手法は、一般化していないながらも、VS Code の Remote Container 機能で一通りのことが行えることが理解いただけたのではと思います。

本文中でも紹介しましたが筆者の開発環境は Parallels Desktop、もしくは Ubuntu Desktop を使って、Linux 環境に移行しており、Dev Container で開発環境を構築できることは知りながらも、大きく Dev Container に移行することはしていませんでした（特に Docker Desktop のストレージ遅い問題は聞いていたので）。リモートパッケージマネージャが提供する環境分離機能はたいいてい十分強力で、Dev Container にしなくとも、リモート SSH 機能を使って Linux 環境で開発できていれば十分でした。しかし、筆者は、Visual Studio Codespaces が発表され、それが Remote Container 機能と結びついていると分かった時に、次の時代の開発環境のスタンダードが今来たのだと、本当に感動しました。Remote Container 機能として、リポジトリ内の `devcontainer.json` で環境設定が完結していることは知っていましたが、これがクラウド、そして Github と結びついた時に、開発環境が一つに繋がって、これで十分な状態になったと思いました。その時のパッションで Dev Container について考えたことをまとめたのが、本書となります。

Dev Container を使った開発は、今は VS Code での開発に限定されています。しかし、開発ツールの多くが Language Server で提供されるようになると、Dev Container の中には Language Server のツールをインストールした状態にできます。すると、LSP をサポートしたエディタであれば Dev Container 内の Language Server を利用できます。これにより、将来的には Dev Container は VS Code に限らず一般化し、開発環境のスタンダードになるのではないかと思います。

本書によって、読者の開発環境を Dev Container を使ってより充実したものになればと思います。

VS Code Dev Container Guidebook

2020年9月12日 初版発行 (@技術書典9)

Atsushi Morimoto (@74th) 著
