Visual COBOL チュートリアル

COBOL 開発： ステップバイステップチュートリアル

コンテナを利用した開発

1 目的

コンテナ技術により、Linux カーネルのコンテナ機能を使って実行環境を他のプロセスから隔離し、その中でアプリケーションを動作させることができます。また、コンテナプロセスの起動に必要なシステム資源は、仮想マシンの起動と比較すると非常に軽量です。コンテナ技術の利用により、アプリケーションとライブラリを同一のコンテナ内に固められるため、容易にアプリケーションの移動やディプロイが行えます。

Visual COBOL は、コンテナ技術として Docker、もしくは Podman を利用することができます。

コンテナ技術を利用することにより COBOL 開発に以下の利点を提供します。

- 開発・実行環境をイメージで保持するため、CI ツールとの連携による日々の自動テストや回帰テストの実施や、同一環境の複数立ち上げが非常に容易
- バージョン毎にイメージが作成されるため、パッチアップデートを含めたバージョンアップ検証作業において複数環境構築が不要

本チュートリアルでは、コンテナの基本機能を学びながら、Visual COBOL 製品のコンテナイメージを用いて、単体テストを継続的に実行できる CI 環境の構築を行います。

2 前提

- 本チュートリアルで使用したマシン OS ： Amazon Linux 2023
- Visual COBOL 11.0 Development Hub コンテナ製品をご購入のお客様

本チュートリアルでは、一部の手順において、下記リンク先のサンプルファイルを使用します。事前にダウンロードをお願いします。

サンプルプログラムのダウンロード

内容

1 目的

2 前提

3 コンテナ技術の紹介

3.1 Visual COBOL 製品コンテナイメージについて

3.2 パッチアップデートについて

4 チュートリアルの流れ

4.1 コンテナの起動方法

4.2 コンテナを利用した COBOL 開発

4.3 ホスト OS とコンテナ環境間のリソース共有

4.3.1 バインドマウントの利用

4.3.2 ボリュームマウントの利用

4.4 JVM COBOL 開発

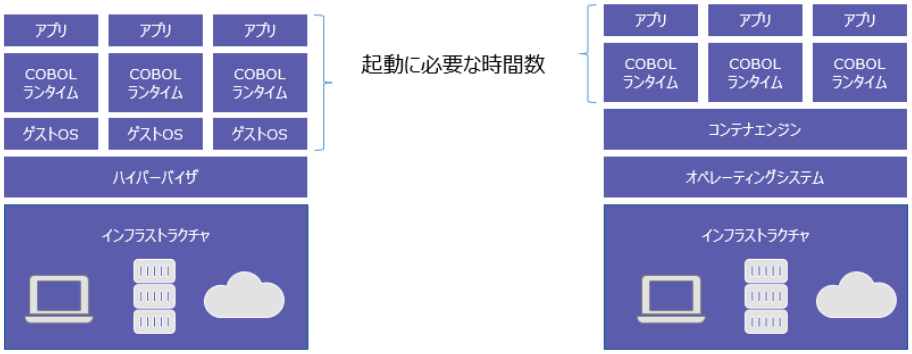
4.5 コンテナを利用した単体テストの実行

5 補足

5.1 サンプルスクリプトの紹介

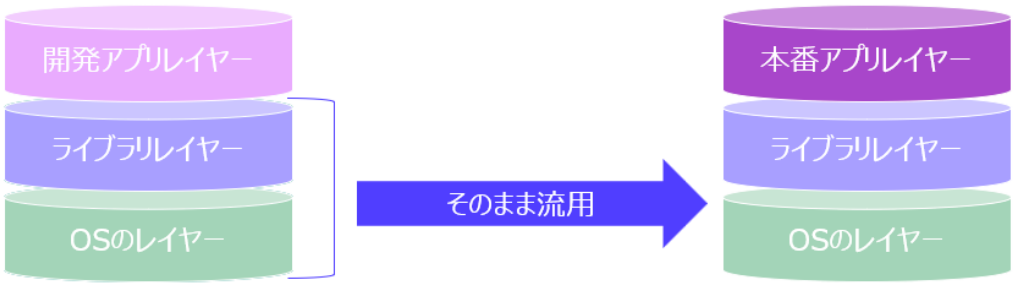
3　コンテナ技術の紹介

一般的な仮想化技術を用いて複数環境を構築した場合、ハイパーバイザ上の各仮想ハードウェア環境にそれぞれ OS がインストールされるので、起動を必要とするばかりでなく必要なハードウェアリソースが多くなります。一方、Docker や Podman などのコンテナ技術を使用した場合、OS の上位にコンテナエンジンが入り、そこで分離されたアプリケーションがそれぞれ実行されるので CPU やメモリーなどのリソースが有効活用でき、簡単に環境の複製や削除をおこなうことができます。



一般的なサーバー仮想化による運用　　コンテナ技術を利用した運用

コンテナ環境を構成するイメージファイルは、レイヤーと呼ばれる論理的な層を重ねて作成されます。この特長を利用することで、下の図のように 開発環境で作成した OS やライブラリのレイヤーを本番機に持ち込むことで、本番環境用のアプリケーションを追加するだけで本番環境向けのイメージの構築が可能となります。OS やライブラリレイヤーといったプラットフォーム部を固定化することで、開発・本番環境差異による動作不具合といった問題を防止することができます。



なお、コンテナ環境の利用にあたっては、以下の点についてご注意ください。

- 一般的にコンテナイメージは 64-bit 版のみサポートされています。
- 文字コード Shift_JIS を利用する際は OS のサポート情報をご確認ください。

3.1 Visual COBOL 製品コンテナイメージについて

Visual COBOL 製品のコンテナイメージは、以下の構成となっています。

以下は、Amazon Linux 2023 環境のイメージ名です。

レイヤー名	イメージ名
Visual COBOL コンテナ製品対話操作用	amzn2023_11.0_x64_login
Visual COBOL コンテナ製品	amzn2023_11.0_x64
OS	Amazonlinux

Visual COBOL コンテナ製品は、目的により以下の 2 つのイメージを使い分けます。

- Visual COBOL コンテナ製品

  アプリケーションを含むコンテナを作成する場合にベースイメージとして使用

- Visual COBOL コンテナ製品対話操作用

  対話型によるコンテナ環境内での操作や、スクリプトによる実行など、開発環境をオンデマンドで利用

Visual COBOL コンテナ製品対話操作用は、通常インストールでは自動的に作成されますが、インストール時に作成を抑止する指示もできます。

3.2 パッチアップデートについて

コンテナの特長の 1 つとして、同一イメージに対して各バージョン用のタグを設定することで、容易に複数環境を管理できる点があげられます。

Visual COBOL 製品も、このコンテナの特長を活かすため、パッチアップデートイメージは別なタグ名が設定される形で別イメージとなります。

例えば、Amazon Linux 2023 環境に Patch Update 1 を適用した場合、以下の製品イメージが新規に作成されます。

rocketsoftware/vcdevhub:amzn2023_11.0_x64_login_pu01

rocketsoftware/vcdevhub: amzn2023_11.0_x64_pu01

4　チュートリアルの流れ

本章では、製品コンテナイメージを利用した開発に必要な操作や機能を以下の順に紹介します。

1. コンテナの起動方法
2. コンテナを利用した COBOL 開発
3. ホスト OS とコンテナ環境間のリソース共有
4. JVM COBOL 開発
5. コンテナを利用した単体テストの実行

以降の手順では、3.1 にて紹介した対話操作用のイメージを使用します。本イメージをお持ちでないお客様は、再インストール作業を行ってください。また、一部の手順において、本文書 1 ページ目に記載したリンクよりダウンロードできるサンプルファイルを使用します。事前にダウンロードの上、ホスト OS 上の任意のディレクトリに転送・解凍を行ってください。手順では /tmp 配下に解凍しているものとして説明を行います。

稼働環境によって異なるコンテナの違いを吸収するため、以下の用語を使用します。

<コンテナコマンド>: docker または、podman

また、本チュートリアルでは、コンテナイメージは Amazon Linux 2023 上で Visual COBOL DevelopmentHub 11.0 を使用しています。異なるバージョンをご利用のお客様は、適宜、イメージ名を変更してください。

4.1　コンテナの起動方法

コンテナを起動し、簡単な COBOL テストプログラムを実行します。

1) docker を起動できるユーザーでログインします。
2) コンテナを起動します。

コンテナの起動コマンドは以下の構成で実行されます。

<コンテナコマンド> run --rm -it <コンテナイメージ名>:<コンテナイメージタグ名>

docker run --rm -it rocketsoftware/vcdevhub:amzn2023_11.0_x64_login

```
[ec2-user@al2023-v-na tmp]$ docker run --rm -it rocketsoftware/vcdevhub:amzn2023_11.0
_x64_login
[ec2-user@06deb2407b9f ~]$
```

補足）

コンテナイメージの一覧は、以下のコマンドを実行することで確認できます。

<コンテナコマンド> images

オプションについて）

ここで使用するオプションは以下の通りです。

--rm:　　コンテナプロセスが終了した時点で、今回使用した環境を破棄します

-it:　　i と t オプション双方で、対話モードでコンテナ環境内での操作が行えるようになります。

4.2 コンテナを利用した COBOL 開発

ここでは、4.1 にて起動したコンテナ内で 製品付属のサンプルプログラムをコンパイル・実行を行います。コンテナ環境であっても、開発手順は、通常のコマンドライン開発と同様です。

1) 以下のコマンドを実行し、簡単な計算プログラムを作成します。

注意）

プログラムのインデントにご注意ください。

```
[ec2-user@06deb2407b9f ~]$ echo "
          program-id. Program1.
          data division.
          working-storage section.
          01 a pic 9.
          procedure division.
          compute a = 1 + 2 * 3.
          display a.
          end program." > Program1.cbl
[ec2-user@06deb2407b9f ~]$
```

補足）

コンテナ環境の特長の１つである軽量化の実現のため、一般的にコンテナ環境内に ”vi” コマンドはインストールされていません。このため、上記では echo コマンドを使い、標準出力をファイルにリダイレクトして、プログラムファイルを作成しています。必要に応じて、手動でインストールすることも可能です。また、ホスト OS のディレクトリをコンテナ環境にマウントする手法もあります。こちらは、4.3 で紹介します。

2) コンパイルを行った後、プログラムを実行します。

cob -x Program1.cbl

./Program1

```
[ec2-user@06deb2407b9f ~]$ cob -x Program1.cbl
[ec2-user@06deb2407b9f ~]$ ./Program1
7
[ec2-user@06deb2407b9f ~]$
```

計算結果である “7” が表示され、正しくコンパイル、および、プログラムの実行が行われています。

3) コンテナ環境からホスト OS に戻ります。コンテナ環境は “--rm” オプションにより破棄されます。

exit

```
[ec2-user@06deb2407b9f ~]$ exit
exit
[ec2-user@al2023-v-na tmp]$
```

4) さきほど同様のコマンドを用いて、コンテナプロセスを起動、プロセスへのアタッチを行います。

<コンテナコマンド> run --rm -it <コンテナイメージ名>:<コンテナイメージタグ名>

docker run --rm -it rocketsoftware/vcdevhub:amzn2023_11.0_x64_login

```
[ec2-user@al2023-v-na tmp]$ docker run --rm -it  rocketsoftware/vcdevhub:amzn2023_11.
0_x64_login
[ec2-user@fa4a19c17099 ~]$
```

5) Program1.cbl が存在しないことの確認を終えましたら、exit コマンドを実行し、ホスト OS に戻ります。

ls Program1.cbl

exit

```
[ec2-user@fa4a19c17099 ~]$ ls Program1.cbl
ls: cannot access 'Program1.cbl': No such file or directory
[ec2-user@fa4a19c17099 ~]$ exit
exit
[ec2-user@al2023-v-na tmp]$
```

補足）

同じようにコンテナプロセスの起動・アタッチを行いましたが、さきほどのコンパイル結果の実行モジュール “Program1” は存在しません。これは、コンテナ起動時は常にイメージに保存された環境で初期化されるためです。

4.3 では、コンテナプロセス内で作成したファイル、もしくは、ホスト OS からコンテナ環境へのリソース共有方法を紹介します。

4.3 ホスト OS とコンテナ環境間のリソース共有

コンテナ環境は仮想環境であり、ホスト OS とは別環境のため、ファイルなどのリソースが個別になります。

本節では、ホスト OS ・コンテナ環境間のリソース共有方法について紹介します。なお、これはコンテナ技術がサポートする機能範囲であり、弊社 Visual COBOL 製品がサポートする機能ではないため、詳細については各コンテナ技術のマニュアルなどをご参照ください。

4.3.1 バインドマウントの利用

本手法は、事前準備なく、コンテナ環境起動時に、mount オプションを使用することで、コンテナ環境内に指定したホスト OS へのディレクトリ・ファイルをマウントすることができます。ここでは、コンテナ環境内で作成したリソースをホスト OS でも参照できることを確認します。

1) バインドマウント対象となるディレクトリを任意のディレクトリに作成します。

ここでは、対象を /tmp/share とします。

```
[ec2-user@al2023-v-na tmp]$ mkdir /tmp/share
[ec2-user@al2023-v-na tmp]$
```

2) 4.1, 4.2 と同様の手順で、コンテナ内に Program1. cbl を作成します。

今回のコンテナ環境起動時には、v オプションを利用して、バインドマウント対象ディレクトリ（/tmp/share）を指定します。

<コンテナコマンド> run --rm -it –v<ホスト OS のバインドマウント対象ディレクトリ>:<コンテナ環境のマウント先ディレクトリ>:z <コンテナイメージ名>:<コンテナイメージタグ名>

docker run --rm -ti -v /tmp/share:/tmp/hostshare:z ¥

rocketsoftware/vcdevhub:amzn2023_11.0_x64_login

```
[ec2-user@al2023-v-na tmp]$ docker run --rm -ti -v /tmp/share:/tmp/hostshare:z ro
cketsoftware/vcdevhub:amzn2023_11.0_x64_login
[ec2-user@d8c0ef064f69 ~]$
```

コンテナ環境内に /tmp/hostshare に移動します。

```
[ec2-user@d8c0ef064f69 ~]$ cd /tmp/hostshare/
[ec2-user@d8c0ef064f69 hostshare]$
```

3) テストプログラムを作成します。

```
echo "
        program-id. Program1.
        data division.
        working-storage section.
        01 a pic 9.
        procedure division.
        compute a = 1 + 2 * 3.
        display a.
        end program." > Program1.cbl
```

注意）
プログラムのインデントにご注意ください。

```
[ec2-user@d8c0ef064f69 hostshare]$ echo "
        program-id. Program1.
        data division.
        working-storage section.
        01 a pic 9.
        procedure division.
        compute a = 1 + 2 * 3.
        display a.
        end program." > Program1.cbl
[ec2-user@d8c0ef064f69 hostshare]$
```

4) コンパイルを行います。

```
cob -x Program1.cbl
```

```
[ec2-user@d8c0ef064f69 hostshare]$ cob -x Program1.cbl
[ec2-user@d8c0ef064f69 hostshare]$
```

5) コンテナ環境からホスト OS に戻ります。コンテナ環境は “--rm” オプションにより破棄されます。

```
exit
```

```
[ec2-user@d8c0ef064f69 hostshare]$ exit
exit
[ec2-user@al2023-v-na tmp]$
```

6) ホスト OS のバインドマウント対象である /tmp/share 内には、コンテナ内で作成したファイルが確認できます。

```
[ec2-user@al2023-v-na tmp]$ ls /tmp/share/
Program1  Program1.cbl  Program1.idy  Program1.int  Program1.o
[ec2-user@al2023-v-na tmp]$
```

4.3.2 ボリュームマウントの利用

バインドマウントは、直接、ホスト OS のリソースをコンテナプロセス内にマッピングする仕組みでした。こちらで紹介するボリュームマウントとは、コンテナ技術がサポートするコマンドを利用して、ホスト OS 環境内の特定ディレクトリ先に物理保存領域を作成し、ボリュームという論理名称を用いて、ホスト OS とコンテナ環境間のリソース共有を実現します。

1) 下記コマンドにて、ボリュームを作成します。

<コンテナコマンド> volume create testvol

docker volume create testvol

```
[ec2-user@al2023-v-na tmp]$ docker volume create testvol
testvol
[ec2-user@al2023-v-na tmp]$
```

2) 作成したボリュームをマウントしながら、コンテナプロセスを起動、アタッチを行います。

<コンテナコマンド> run --rm -it –v<ボリューム名>:<コンテナ環境のマウント先ディレクトリ> <コンテナイメージ名>:<コンテナイメージタグ名>

```
docker run --rm -it -v testvol:/tmp/hostshare ¥
            --user $(id -u root):$(id -g root) ¥
            rocketsoftware/vcdevhub:amzn2023_11.0_x64_login
```

補足）

ec2-user を wheel グループに登録することで、ec2-user で docker を利用できます。しかし、docker デーモンが root で動作しているため、docker volume によって作成された testvol の権限は root になります。その結果、ボリュームマウントされたディレクトリの権限不足が発生します。このため、--user オプションを使用して root 情報を使用してログインしています。

管理者権限をもつユーザーで実行している場合は、このような権限不一致の問題は起こりません。

```
[ec2-user@al2023-v-na tmp]$ docker run --rm -it -v testvol:/tmp/hostshare ¥
            --user $(id -u root):$(id -g root) ¥
            rocketsoftware/vcdevhub:amzn2023_11.0_x64_login
bash-5.2#
```

3) 以下のコマンドを実行し、COBOL 開発環境を設定します。

. /opt/rocketsoftware/VisualCOBOL/bin/cobsetenv

```
bash-5.2# . /opt/rocketsoftware/VisualCOBOL/bin/cobsetenv
COBDIR set to /opt/rocketsoftware/VisualCOBOL
bash-5.2#
```

4) /tmp/hostshare に移動したうえで、プログラムの作成、コンパイルを行います。

```
cd /tmp/hostshare
echo "
            program-id. Program1.
            data division.
            working-storage section.
            01 a pic 9.
            procedure division.
```

```
compute a = 1 + 2 * 3.
display a.
end program." > Program1.cbl
```

注意）
プログラムのインデントにご注意ください。

```
bash-5.2# cd /tmp/hostshare
echo "
        program-id. Program1.
        data division.
        working-storage section.
        01 a pic 9.
        procedure division.
        compute a = 1 + 2 * 3.
        display a.
        end program." > Program1.cbl
bash-5.2#
```

5) コンパイルを行います。

cob -x Program1.cbl

```
bash-5.2# cob -x Program1.cbl
bash-5.2#
```

6) コンテナ環境からホスト OS に戻ります。コンテナ環境は “--rm” オプションにより破棄されます。

exit

```
bash-5.2# exit
exit
[ec2-user@al2023-v-na tmp]$
```

7) ホスト OS 上で、コンテナ環境内で作成したファイルが参照できることを確認します。

以下のコマンドにて、testvol ボリュームの物理領域を確認します。

<コンテナコマンド> volume inspect testvol

docker volume inspect testvol

```
[ec2-user@al2023-v-na tmp]$ docker volume inspect testvol
[
    {
        "CreatedAt": "2024-08-30T06:37:25Z",
        "Driver": "local",
        "Labels": null,
        "Mountpoint": "/var/lib/docker/volumes/testvol/_data",
        "Name": "testvol",
        "Options": null,
        "Scope": "local"
    }
]
[ec2-user@al2023-v-na tmp]$
```

オレンジ色で設定した Mountpoint のパスが物理パスとなります。

実際に、上記パスを確認すると、以下のようにコンテナ環境で作成したファイルが確認できます。

```
$ sudo ls /var/lib/docker/volumes/testvol/_data
Program1  Program1.cbl  Program1.idy  Program1.int  Program1.o
[ec2-user@al2023-v-na tmp]$
```

8) 不要となった testvol ボリュームを削除します。

<コンテナコマンド> volume remove testvol

docker volume rm testvol

```
[ec2-user@al2023-v-na tmp]$ docker volume rm testvol
testvol
[ec2-user@al2023-v-na tmp]$
```

さきほど確認したディレクトリも削除されます。

```
[ec2-user@al2023-v-na tmp]$ sudo ls /var/lib/docker/volumes/testvol/_data
ls: cannot access '/var/lib/docker/volumes/testvol/_data': No such file or directory
[ec2-user@al2023-v-na tmp]$
```

4.4 JVM COBOL 開発

本節で使用するサンプルは、Java クラスより指定された２つの数値を COBOL プログラムで加算し、Java 側でその結果を表示する簡単なプログラムです。

1) サンプルディレクトリをマウントしつつ、コンテナ環境を起動します。

<コンテナコマンド> run --rm -it -v /tmp/vc-containertutorial01/jvm:/tmp/sample:z ¥
rocketsoftware/vcdevhub:rhel8.6_9.0_x64_pu02_login

docker run --rm -it -v /tmp/vc-containertutorial01/jvm:/tmp/sample:z ¥
rocketsoftware/vcdevhub:amzn2023_11.0_x64_login

```
[ec2-user@al2023-v-na tmp]$ docker run --rm -it -v /tmp/vc-containertutorial01/jvm:/tmp/sample:z ¥
>         rocketsoftware/vcdevhub:amzn2023_11.0_x64_login
[ec2-user@d56bbb6f7d17 ~]$
```

補足)

ホスト OS とコンテナ環境間のリソース共有を行っています。4.3.1 にて紹介しておりますので、ご参照ください。

2) サンプルディレクトリに移動したうえで、以下のコマンドを用いて、JVM COBOL のコンパイルを行います。

cd /tmp/sample

cob -j Add.cbl -C"ilnamespace(sample.jvm) ilsmartlinkage"

javac JavaMain.java

```
[ec2-user@d56bbb6f7d17 tmp]$ cd /tmp/sample
[ec2-user@d56bbb6f7d17 sample]$ cob -j Add.cbl -C"ilnamespace(sample.jvm) ilsmartlinkage"
[ec2-user@d56bbb6f7d17 sample]$ javac JavaMain.java
[ec2-user@d56bbb6f7d17 sample]$
```

補足)

cob コマンドに指定しているコンパイラー指令 ilsmartlinkage を使用することで、Java 言語から容易に COBOL プログラムを呼び出せるようになります。Ilnamespace は、コンパイルによって作成される JVM COBOL クラスの名前空間

を指定しています。各指令の詳細については、製品マニュアルをご参照ください。

3) プログラムを実行します。

java JavaMain

```
[ec2-user@d56bbb6f7d17 sample]$ java JavaMain
Var1(58731) + Var2(73959) = Rst(132690)
[ec2-user@d56bbb6f7d17 sample]$
```

Java プログラムから JVM COBOL プログラムが呼び出されていることが確認できます。

4) コンテナ環境からホスト OS に戻ります。コンテナ環境は “--rm” オプションにより破棄されます。

exit

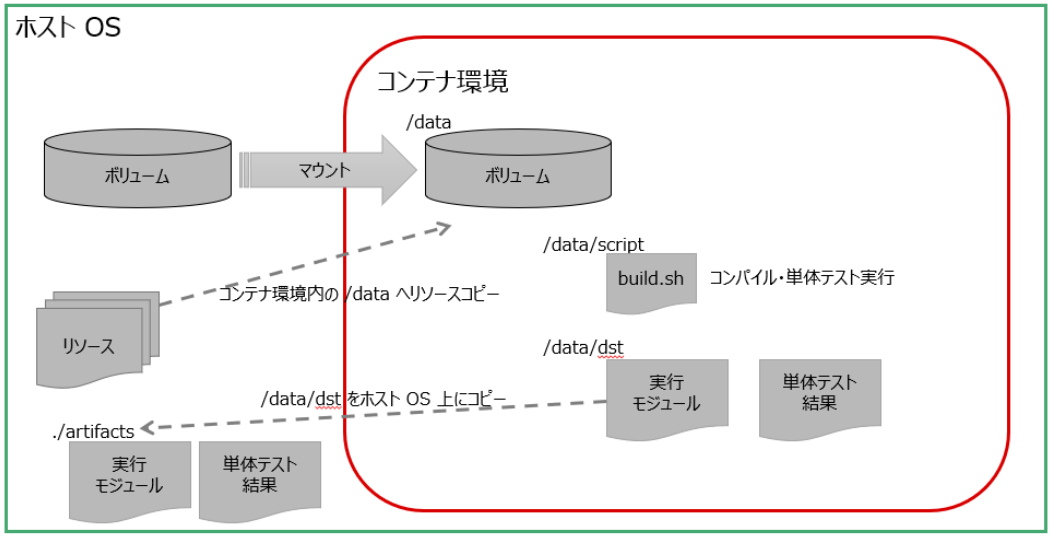
```
[ec2-user@d56bbb6f7d17 sample]$ exit
exit
[ec2-user@al2023-v-na tmp]$
```

4.5 コンテナを利用した単体テストの実行

これまでに、コンテナ環境内での COBOL 開発は、通常のコマンドライン開発と同様に行えること、コンテナ内で作成したリソースをホスト OS に共有する方法を確認しました。

コンテナ仮想化技術は、コンテナ環境の起動が高速であり、起動毎にイメージで保存された環境が構築されるといった特長から、繰り返し同じ処理を実行し続ける場面において非常に役立ちます。例えば、データファイルなどへの更新処理が行われるようなケースでも、コンテナ環境を起動するたびにファイル状態が初期状態に戻るため、データの初期化といった前処理を行う必要がありません。また、スケールアウト方式による負荷分散、性能アップへの展開もできます。

本節では、製品付属の開発支援ユーティリティの 1 つである COBOL 専用の単体テストフレームワークである MFUnit をコンテナ環境で実行する構築例を紹介します。なお、本例の全体像は、以下のようになっています。



本節で使用するサンプルは、従来から製品に付属している AirportDemo で、使用して空港情報や空港間の距離を計算するネイティブ COBOL アプリケーションとなります。今回は、空港間距離の計算処理に対して単体テストを実行します。ファイル構造は以下の通りです。

リソース名	リソース説明
build.sh	コンテナ環境の起動・破棄やリソースコピーなど、全体制御を実施
dat/airports.dat	空港情報が定義された索引ファイル
dst	コンテナ環境内で成果物を保存するディレクトリ
script/build.sh	コンテナ環境内で実行され、プログラムのコンパイルと単体テストを実行するスクリプト
src/*	COBOL プログラムを保持するディレクトリ テスト対象： aircode.cbl 単体テストプログラム： TestProgram1.cbl

1) サンプルファイルを解凍したディレクトリ配下の native ディレクトリに移動したうえで build.sh を開き、コンパイル・テストで使用する以下の変数を環境に合わせ変更し保存します。

VCIMAGE 変数の値を更新します。

環境毎の修正箇所、および、スクリプトの初期値は、以下のようになっています。

変数	Build.sh の変更箇所
VCIMAGE	Visual COBOL 製品コンテナイメージ名とタグ名 Amazon Linux 2023 の場合は、 rocketsoftware/vcdevhub:amzn2023_11.0_x64 イメージ名が分からない場合は、以下のコマンドで確認できます。 <コンテナコマンド> images
CONTAINER_CMD	<コンテナコマンド>
DOCKER_OPTION	podman を使用する場合はコメントアウトしてください。

2) build.sh を実行します。

sh build.sh

build.sh のスクリプト内容については 5.1 をご参照ください。

```
[ec2-user@al2023-v-na native]$ sh build.sh
COBDIR set to /opt/rocketsoftware/VisualCOBOL
Rocket (R) COBOL - cobmfurun64 Utility
Unit Testing Framework for Unix/Native/64

Fixture : TestProgram1

Test Run Summary
 Overall Result             Passed
 Tests run                   2
 Tests passed                2
 Tests failed               0
 Total execution time       0
[ec2-user@al2023-v-na native]$
```

テスト結果が表示され、2 件中 2 件成功と表示されていることが分かります。

また、直下に artifacts ディレクトリが作成され、コンテナ環境内で作成された成果物が保存されています。

```
[ec2-user@al2023-v-na native]$ ls -R artifact/
artifact/:
TestProgram1.idy  TestProgram1.int  aircode.idy  aircode.int  result

artifact/result:
TEST-MFUT_TestAIRCODE.xml  TEST-MFUT_testDistance.xml  TestProgram1-report.txt
[ec2-user@al2023-v-na native]$
```

5 補足

5.1 サンプルスクリプトの紹介

説明のため、スクリプト内には存在しない行番号を設定しています。

```
 1  ####################################
 2  # 変数定義 #
 3  ####################################
 4  VCIMAGE=rocketsoftware/vcdevhub:amzn2023_11.0_x64
 5  CONTAINER_CMD=docker
 6  DOCKER_OPTION="-u $(id -u):$(id -g)"
 7  MOUNT_DIR=./mount_dir
 8  ARTIFACT_DIR=artifact
 9  ####################################
10  #   成果物保存ディレクトリの削除   #
11  ####################################
12  if [ -d $ARTIFACT_DIR ]; then
13     rm -fr $ARTIFACT_DIR
14  fi
15  ####################################
16  # コンテナを利用した処理   #
17  ####################################
18  mkdir -p $MOUNT_DIR/$ARTIFACT_DIR
19  cp -pr src $MOUNT_DIR
20  cp -pr dat $MOUNT_DIR
21  cp -pr script $MOUNT_DIR
22        $CONTAINER_CMD  run  --rm  $DOCKER_OPTION  -v  $MOUNT_DIR:/data:z  --workdir
/data/$ARTIFACT_DIR $VCIMAGE sh ../script/build.sh
23  mv $MOUNT_DIR/$ARTIFACT_DIR ./
24  rm -fr $MOUNT_DIR
```

免責事項

ここで紹介したソースコードは、機能説明のためのサンプルであり、製品の一部ではございません。ソースコードが実際に動作するか、御社業務に適合するかなどに関しまして、一切の保証はございません。 ソースコード、説明、その他すべてについて、無謬性は保障されません。

ここで紹介するソースコードの一部、もしくは全部について、弊社に断りなく、御社の内部に組み込み、そのままご利用頂いても構いません。

本ソースコードの一部もしくは全部を二次的著作物に対して引用する場合、著作権法の精神に基づき、適切な扱いを行ってください。