MICRO FOCUS

Micro Focus 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 ： Red Hat Enterprise Linux 7.6 / Red Hat Enterprise Linux 8.2
- Linux 環境に Red Hat のサブスクリプション情報が登録済みであること
- Visual COBOL 7.0 Development Hub 製品をご購入のお客様

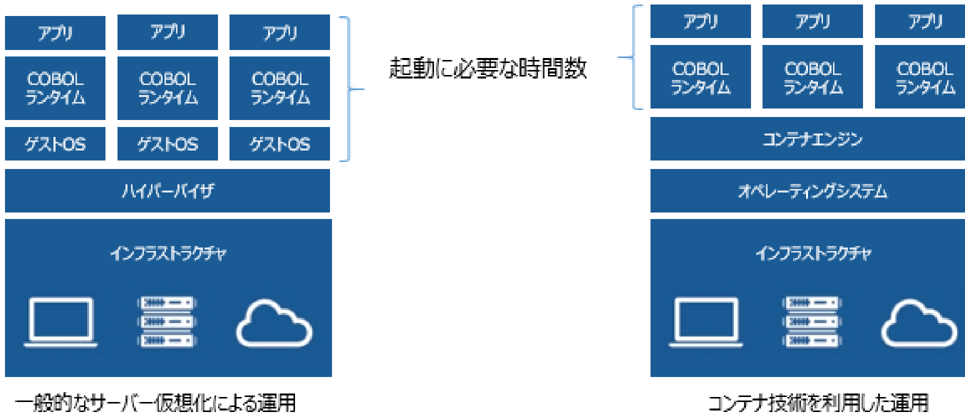
本文書では、製品コンテナイメージをインストールするホスト OS 環境を Red Hat Enterprise Linux 7.x, 8.x 、もしくは、単に 7.x, 8.x と記載していますが、製品コンテナイメージがサポートする環境は Red Hat Enterprise Linux 7.4 以降, もしくは Red Hat Enterprise Linux 8.0 以降となります。また、各 OS 環境とコンテナサポートバージョンについては、Red Hat 社サイトにてご確認ください。

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。

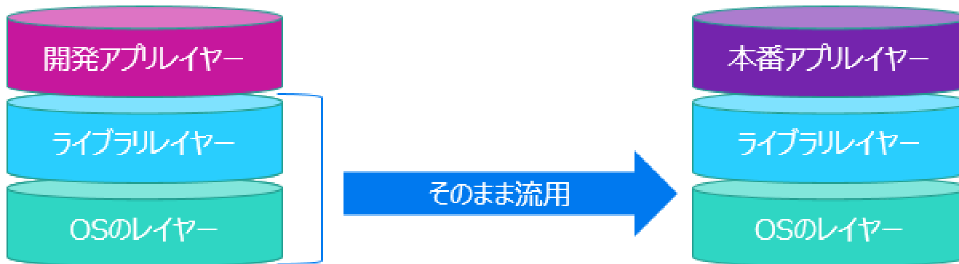
サンプルプログラムのダウンロード

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。



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。

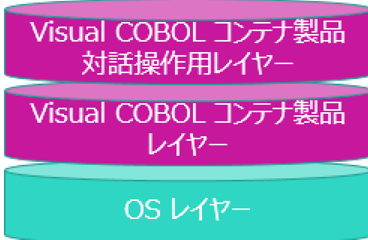


なお、コンテナ環境の利用にあたっては、以下の点についてご注意ください。

- コンテナイメージは 64-bit 版のみ Red Hat 社から提供されています。
- 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、Red Hat Enterprise Linux 7.6, 8.2 環境での各イメージ名です。

レイヤー名	Red Hat Enterprise Linux 7.6	Red Hat Enterprise Linux 8.2
Visual COBOL 製品コンテナイメージ対話操作用	microfocus/vcdevhub: rhel7_7.0_x64_login	microfocus/vcdevhub: rhel8.2_7.0_x64_login
Visual COBOL 製品コンテナイメージ	microfocus/vcdevhub: rhel7_7.0_x64	microfocus/vcdevhub: rhel8.2_7.0_x64
OS レイヤー	rhel7/rhel:latest	ubi8/ubi-minimal:8.2

Visual COBOL 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 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 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 製品コンテナイメージ対応操作用は、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。例えば、Red Hat の OS イメージでは、前述の 8.2 環境例である OS レイヤーではタグ名に "8.2" が設定されていますが、Red Hat 社より 8.0, 8.1 といったタグ名のイメージも提供しています。

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、Red Hat Enterprise Linux 7.6, 8.2 環境に patchupdate02 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。

- Red Hat Enterprise Linux 7.6
  microfocus/vcdevhub:rhel7_7.0_x64_pu02
  microfocus/vcdevhub:rhel7_7.0_x64_pu02_login
- Red Hat Enterprise Linux 8.2
  microfocus/vcdevhub:rhel8.2_7.0_x64_pu02
  microfocus/vcdevhub:rhel8.2_7.0_x64_pu02_login

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。

なお、Red Hat Enterprise Linux 7.x, 8.x いずれの環境をご利用のお客様も本チュートリアルは実施できますが、説明時にコンテナコマンド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>: 7.x 環境のお客様は docker, 8.x 環境のお客様は podman に読み替えてください。

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) 管理者権限のあ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>

   下記は、8.2 環境で patchupdate02 を適用した場合のコマンドになります。

   podman run --rm -it microfocus/vcdevhub:rhel8.2_7.0_x64_pu02_login

```
#  podman run --rm -it microfocus/vcdevhub:rhel8.2_7.0_x64_pu02_login
COBDIR set to /opt/microfocus/VisualCOBOL
[root@1e14d04815c9 ~]#
```

補足）
上記実行画面イメージの中で、COBDIR set to /opt/microfocus/VisualCOBOL の文言は、Red Hat Enterprise Linux 7.x をご利用のお客様は表示されません。次の手順にて、この環境変数の設定を行いますので、ここでは表示されていなくても問題ありません。

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。

3) Red Hat Enterprise Linux 7.x 環境をお使いのお客様のみ、以下のコマンドを実行します。

. /opt/microfocus/VisualCOBOL/bin/cobsetenv

```
# . /opt/microfocus/VisualCOBOL/bin/cobsetenv
COBDIR set to /opt/microfocus/VisualCOBOL
```

補足）
Red Hat Enterprise Linux 8.x 環境のコンテナイメージでは、本設定が自動的に行われています。

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。

```
# 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。

```
# 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
[root@1e14d04815c9 ~]#
```

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

```
# cob -x Program1.cbl
[root@9009377b3096 ~]# ./Program1
7
[root@9009377b3096 ~]#
```

計算結果である "7" が表示され、正しくコンパイル、および、実行ができることを確認してください。

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

```
# exit
exit
[root@localhost 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>

下記は、8.2 環境で patchupdate02 を適用した場合のコマンドになります。

podman run --rm -it microfocus/vcdevhub:rhel8.2_7.0_x64_pu02_login

```
#  podman run --rm -it microfocus/vcdevhub:rhel8.2_7.0_x64_pu02_login
COBDIR set to /opt/microfocus/VisualCOBOL
[root@28d12ff40a85 ~]#
```

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。

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。

exit

```
# exit
exit
[root@localhost tmp]#
```

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。

```
# mkdir /tmp/share
#
```

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>

以下は、8.2 環境で patchupdate02 を適用した場合のコマンドになりますが、mount オプションにより、ホスト OS 上の /tmp/share をコンテナ環境下の /tmp/hostshare にマウントしています。

podman run --rm -it -v /tmp/share:/tmp/hostshare:z microfocus/vcdevhub:rhel8.2_7.0_x64_pu02_login

```
#  podman run --rm -it -v /tmp/share:/tmp/hostshare:z
microfocus/vcdevhub:rhel8.2_7.0_x64_pu02_login
```

補足）
公式ドキュメントでは、v オプションより、mount オプションの使用が望ましいと記載されています。この場合、selinux 対応が必要となります。前述したように、本機能は、Visual COBOL 製品機能範囲ではなく、コンテナ技術側となりますので、本情報についての詳細は、docker, podman などのマニュアルをご参照ください。

コンテナ環境内に /tmp/hostshare に移動します。

```
# cd /tmp/hostshare
[root@d17e7e15f2a1 hostshare]#
```

MICRO FOCUS

3) Red Hat Enterprise Linux 7.x 環境をお使いのお客様のみ、以下のコマンドを実行します。

. /opt/microfocus/VisualCOBOL/bin/cobsetenv

```
# . /opt/microfocus/VisualCOBOL/bin/cobsetenv
COBDIR set to /opt/microfocus/VisualCOBOL
```

4) 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。

```
# 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
[root@d17e7e15f2a1 hostshare]#
```

5) コンパイルを行います。

cob -x Program1.cbl

```
# cob -x Program1.cbl
[root@d17e7e15f2a1 hostshare]#
```

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

```
# exit
exit
[root@localhost tmp]#
```

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

```
# cd /tmp/share/
[root@localhost share]# ls
Program1   Program1.cbl   Program1.idy   Program1.int   Program1.o
[root@localhost share]#
```

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

下記は、8.2 環境で実行する場合です。

podman volume create testvol

```
# podman volume create testvol
testvol
[root@localhost 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>

以下は、8.2 環境で patchupdate02 を適用した場合のコマンドになりますが、testvol ボリュームをコンテナ環境下の /tmp/hostshare にマウントしています。

podman run --rm -it -v testvol:/tmp/hostshare microfocus/vcdevhub:rhel8.2_7.0_x64_pu02_login

```
#   podman run --rm -it -v testvol:/tmp/hostshare
microfocus/vcdevhub:rhel8.2_7.0_x64_pu02_login
COBDIR set to /opt/microfocus/VisualCOBOL
[root@975786ea585a ~]#
```

3) Red Hat Enterprise Linux 7.x 環境をお使いのお客様のみ、以下のコマンドを実行します。

. /opt/microfocus/VisualCOBOL/bin/cobsetenv

```
# . /opt/microfocus/VisualCOBOL/bin/cobsetenv
COBDIR set to /opt/microfocus/VisualCOBOL
```

4) /tmp/hostshare 配下でプログラムの作成、コンパイルを行います。

```
# 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。

```
# pwd
/tmp/hostshare
[root@975786ea585a 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
[root@975786ea585a hostshare]#
```

5) コンパイルを行います。

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

```
# cob -x Program1.cbl
[root@975786ea585a hostshare]#
```

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
```

```
# exit
exit
[root@localhost 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

以下は 8.2 環境でのコマンドとなります。

podman volume inspect testvol

```
# podman volume inspect testvol
[
    {
        "Name": "testvol",
        "Driver": "local",
        "Mountpoint": "/var/lib/containers/storage/volumes/testvol/_data",
        "CreatedAt": "2020-12-07T14:11:08.978079133+09:00",
        (中略)
    }
]
[root@localhost 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。

```
# ls /var/lib/containers/storage/volumes/testvol/_data
Program1   Program1.cbl   Program1.idy   Program1.int   Program1.o
[root@localhost 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

以下は 8.2 環境でのコマンドとなります。

podman volume remove testvol

```
# podman volume remove testvol
testvol
```

さきほど確認したディレクトリも削除されます。

```
# ls /var/lib/containers/storage/volumes/testvol/_data
ls: '/var/lib/containers/storage/volumes/testvol/_data' にアクセスできません: そのようなファイルやディレクトリはありません
```

MICRO FOCUS

4.4. JVM COBOL 開発

JVM COBOL の開発を行うためには、コンテナイメージ内に AdoptOpenJDK がインストールされている必要があります。通常、AdoptOpenJDK はコンテナイメージにインストールされていますが、インストーラーオプションにて AdoptOpenJDK をインストールしない指示を行ったお客様は、再インストールを行ってください。

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になります。

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 microfocus/vcdevhub:rhel8.2_7.0_x64_pu02_login

以下は、8.2 環境で patchupdate02 を適用した場合のコマンドとなります。

podman run --rm -it -v /tmp/vc-containertutorial01/jvm:/tmp/sample:z microfocus/vcdevhub:rhel8.2_7.0_x64_pu02_login

```
#   podman run --rm -it -v /tmp/vc-containertutorial01/jvm:/tmp/sample:z
microfocus/vcdevhub:rhel8.2_7.0_x64_pu02_login
COBDIR set to /opt/microfocus/VisualCOBOL
```

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） Red Hat Enterprise Linux 7.x 環境をお使いのお客様のみ、以下のコマンドを実行します。

. /opt/microfocus/VisualCOBOL/bin/cobsetenv

export PATH=/usr/java/default/bin:$PATH

```
# . /opt/microfocus/VisualCOBOL/bin/cobsetenv
COBDIR set to /opt/microfocus/VisualCOBOL
[root@a39a5df2aa60 usr]# export PATH=/usr/java/default/bin:$PATH
[root@a39a5df2aa60 usr]#cd
```

3） 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

```
# cd /tmp/sample
[root@8cc57d2004da sample]# cob -j Add.cbl -C"ilnamespace(sample.jvm) ilsmartlinkage"
[root@8cc57d2004da sample]# javac JavaMain.java
[root@8cc57d2004da 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。

4) プログラムを実行します。

java JavaMain

```
# java JavaMain
Var1(58731) + Var2(73959) = Rst(132690)
```

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。

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

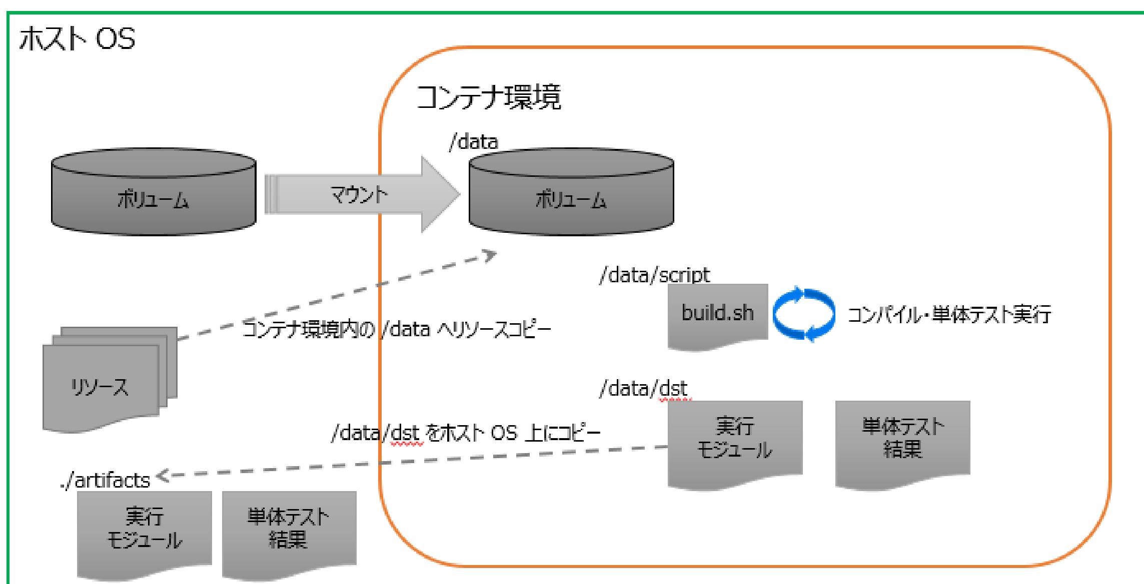
```
# exit
exit
[root@localhost tmp]#
```

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 の修正箇所
VCIMAGE	Visual COBOL 製品コンテナイメージ名とタグ名 例として、8.2 環境で 7.0 patchupdate02 のイメージは以下になります。 microfocus/vcdevhub:rhel8.2_7.0_x64_pu02 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>

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。

```
# sh build.sh
vol-vctutorial--14432
00b9bfb300f694e12821f6c76b3c2bbdba0c1255b87e055d40e7e2e713c4c495
00b9bfb300f694e12821f6c76b3c2bbdba0c1255b87e055d40e7e2e713c4c495
COBDIR set to /opt/microfocus/VisualCOBOL
Micro Focus 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
6129d7e919cd1fe19af8dc5677a1a2800d578269f1a2b117085a972f6d5669dd
vol-vctutorial--14432
[root@localhost 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。

```
]# ls artifact/
TestProgram1.idy   TestProgram1.int   aircode.idy   aircode.int   result
```

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。

```
01: ##################################
02: #                    変数定義                         #
03: ##################################
04: BASEVOLNAME=vol-vctutorial-
05: DATAVOLUME=${BASEVOLNAME}-$$
06: VOLIMAGENAME=devhub-image
07: VCIMAGE=microfocus/vcdevhub:rhel8.2_7.0_x64_pu02
08: CONTAINER_CMD=podman
09:
10:
11: ##################################
12: #     成果物保存ディレクトリの削除                  #
13: ##################################
14: if [ -d artifact ]; then
15:     rm -fr artifact
16: fi
17:
18: ##################################
19: #        コンテナを利用した処理                      #
20: ##################################
21: $CONTAINER_CMD volume create ${DATAVOLUME}
22: $CONTAINER_CMD run -v ${DATAVOLUME}:/data --name ${VOLIMAGENAME} ${VCIMAGE}
23: $CONTAINER_CMD cp src      ${VOLIMAGENAME}:/data
24: $CONTAINER_CMD cp dat      ${VOLIMAGENAME}:/data
25: $CONTAINER_CMD cp script ${VOLIMAGENAME}:/data
26: $CONTAINER_CMD cp dst      ${VOLIMAGENAME}:/data
27: $CONTAINER_CMD stop ${VOLIMAGENAME}
28: $CONTAINER_CMD rm ${VOLIMAGENAME}
29: $CONTAINER_CMD run -v ${DATAVOLUME}:/data --name ${VOLIMAGENAME} --workdir
/data/dst ${VCIMAGE} sh ../script/build.sh
31: $CONTAINER_CMD cp ${VOLIMAGENAME}:/data/dst artifact
32: $CONTAINER_CMD rm ${VOLIMAGENAME}
33: $CONTAINER_CMD volume rm ${DATAVOLUME}
```

基本的な流れは、4.5 で紹介した通りとなりますが、ここでは 21 行目以降のコンテナコマンドについて処理概要を紹介します。

行数	説明
21～28	リソース共有のためボリュームを作成し、ボリュームへのリソースをコピーしています。 23～26 行目まで、$CONTAINER_CMD cp コマンドが実行されていますが、これは稼働中のコンテナとホスト OS 間でリソースをコピーすることができます。例えば、23 行目では、ホスト OS 上の src ディレクトリをコンテナ環境内の /data ディレクトリにコピーしています。 ボリュームへのリソースコピーが終了した時点で、コンテナを停止・破棄します。
29～33	29 行目で、さきほどのボリュームを再度マウントして、コンテナ環境内でコンパイルと単体テストを実行します。 workdir を指定することで作業ディレクトリを /data/dst としています。 31 行目では、さきほどとは逆に、コンテナ環境で作成した成果物をホスト OS 環境の artifacts というディレクトリ名でコピーしています。 その後、コンテナの停止・破棄と、ボリュームの削除を行っています。

WHAT'S NEXT

- 本チュートリアルで学習した技術の詳細については製品マニュアルをご参照ください。
- 単体テスト機能を含め、製品付属の開発支援ユーティリティについて、別途チュートリアルを用意しておりますので、マニュアルとともにご参照ください。

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。