White Paper
Micro Focus Visual COBOL

Micro Focus による COBOL と Java の融合

昨今、既存の企業情報システムにおけるモダナイゼーション事例が増加している中で、COBOL ベースの既存 IT 資産を Java ベースの新規開発の中で有効活用することが要求されるようになっています。Java で書かれたモジュールと COBOL で書かれたモジュールを連携させること、特に新規に作成された Java アプリケーションから、既存の COBOL ロジックをコンポーネントとして利用することは、モダナイゼーション後のアプリケーションを強固かつ柔軟に作成するための有効な方法です。
ここでは、Micro Focus が提供する連携技法として、COBOL を部品として利用する方法に関するさまざまな技法を紹介します。

MICRO FOCUS

目次

1. Java 環境における COBOL の活用

Java の普及、Java 人口の増加に伴い、今まで COBOL で書いていたような業務は Java で書けばよく、COBOL は不要になるだろうという見方も出てきています。しかし、プログラミング言語としての Java と COBOL の優劣を比較するのは無意味なことです。

COBOL は、情報システム構築に必要とされる機能を次々に言語に取り込んで発展してきました。これに対して、Java がプログラミング言語として持っている機能の規模は COBOL に比べてほんのわずかです。Java プログラマがシステム資源にアクセスするために豊富な手段を享受できるのは、Java EE が規定する膨大な API があるからです。Java の機能や利点について語るとき、多くの場合は言語としての Java を語っているのではなく、コンピューティング環境としての Java EE のことを語っていることに注意すべきです。そして、Java EE が提供する機能の多くはさまざまな手段によって Java 以外の言語でも利用できるのです。従って、Java の利点を語った後で、「だから C や COBOL より Java が良い」と結論付けるのはたいてい間違った論法です。

分散コンピューティング環境としての Java EE が魅力的で有望なプラットフォームであることに間違いはありません。システム構築基盤として Java EE 準拠のアプリケーションサーバーを選択した場合でも、そこで使用できる言語は Java だけではありません。最近のユーザー事例を見ても、Java だけですべてを構築するのではなく、ビジネスロジックの部分に COBOL を使用しているケースは多くあります。

COBOL で記述されたビジネスロジックを Java から利用可能なようにコンポーネント化してラッピングすることにより、Web ブラウザ、モバイル端末からメインフレームに至るまでさまざまなクライアントから利用可能となり、その資産価値を増大させることができます。

2. 方式の概要

ここで紹介するテクノロジーは次の 3 点です：

1）Micro Focus 提供の RuntimeSystem クラスを使用して COBOL ロジックを Java から呼び出す

Micro Focus Visual COBOL 製品に標準装備している mfcobol.jar パッケージは、Java から使用できるクラスであって、COBOL プログラムをロードして呼び出すことができます。Java から COBOL にパラメーターを渡し、COBOL からの返却値を Java に返すことができ、データ型変換もサポートしています。Java 側に COBOL から例外をスローする方法も提供しています。

2）Enterprise Server 上にディプロイされた COBOL ロジックを Java EE Connector として Java から呼び出す

Micro Focus Visual COBOL の実行環境製品である Micro Focus COBOL Server は COBOL 言語専用のアプリケーションサーバーである Enterprise Server を含んでいます。この上に COBOL プログラムをディプロイすると、Java EE Connector Architecture に準拠する方法で、Java から COBOL を呼び出せます。WebSphere、WebLogic、JBoss などの代表的な Java EE アプリケーションサーバー上で稼動する Java EE アプリケーションから、標準的なプログラミングで COBOL ロジックを使用することができます。

3）JVM COBOL による JVM 内直接連携

Micro Focus Visual COBOL は、COBOL プログラムを JVM 上で稼働する Java バイト コードにコンパイルする JVM COBOL テクノロジーをサポートしています。これを使用して Java アプリケーションと COBOL アプリケーションが同一の JVM 内で相互に呼び出しあうことができます。

利用する Micro Focus 製品

	開発フェーズ	運用フェーズ
RuntimeSystem クラスの利用	Micro Focus Visual COBOL	Micro Focus COBOL Server Micro Focus COBOL Server for SOA
Java EE Connector の利用	Micro Focus Visual COBOL	Micro Focus COBOL Server for SOA
JVM COBOL（注）	Micro Focus Visual COBOL	Micro Focus COBOL Server Micro Focus COBOL Server for SOA

（注）「6-3 マルチユーザーアプリケーションの開発」で紹介する機能は Visual COBOL、COBOL Server、COBOL Server for SOA のいずれもバージョン 2.2 より実装された機能になります。

3. RuntimeSystem クラスの利用

既存の COBOL サブルーチンをそのまま Java から呼び出して利用する最も簡単な方法として、mfcobol.jar パッケージの使用方法を紹介します。

3-1. 最も簡単な例題

Java 言語から Java 以外のネイティブアプリケーションを呼び出すためには、一般には Java Native Interface (JNI) と呼ばれる Java ライブラリを使用します。これを使用すれば、オペレーティングシステム上のネイティブな動的リンクライブラリを Java 仮想マシン上にロードして呼び出すことができます。従ってこの方法を用いて COBOL で書かれたライブラリをロードすることも不可能ではありません。しかし、これには非常に複雑なプログラミングを必要とします。

Visual COBOL 製品に標準装備している mfcobol.jar というパッケージが提供する RuntimeSystem クラスには、一連の COBOL アクセス用関数が提供されています。これを使用すると、複雑な JNI のプログラミングを必要とせずに、Java から COBOL プログラムをロードして呼びだすことができます。

以下に COBOL を呼び出す Java クラスの最も簡単な例題を見ます：

```
import com.microfocus.cobol.*;

class javamain {
      public static void main(String[] args) {
           try{
                 int outVal = 0;
                 Integer inVal = new Integer(args[0]);
                 Object myParms[] = { inVal };
                 outVal = RuntimeSystem.cobcall_int("AddOne", myParms);
                 System.out.println(outVal);
           } catch(Exception e) {
                 System.err.println("COBOL 呼び出し失敗 ¥n");
           }
      }
}
```

この Java クラスはコマンド行起動され、コマンド行引数として渡された数値を AddOne という COBOL プログラムに渡し、そこからの返却値をコンソール表示しています。呼ばれる COBOL プログラムは以下のようなものです。

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  AddOne.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
           COPY JAVATYPES.
       01 outVal jint.
       LINKAGE SECTION.
       01 inVal jint.
       PROCEDURE DIVISION USING inVal.
       1.
           COMPUTE outVal = inVal + 1.
           EXIT PROGRAM
               RETURNING outVal.
```

このプログラムは、パラメーターとして渡された数値に 1 を加算し、その結果を返却値として返しています。
インポートされる mfcobol.jar は、UNIX の場合 $COBDIR/lib 下に、Windows の場合、製品インストール先の ¥bin 及び ¥bin64 の下にあります。

RuntimeSystem を含めた各種クラスは com.microfocus.cobol でパッケージ化されているため、プログラムの先頭に下記のような import 文を記述します :

```
import com.microfocus.cobol.*;
```

Java から COBOL に引き渡すパラメーターは、Java のオブジェクト配列として宣言します。

```
Object myParms[] = { inVal };
```

複数個のパラメーターを渡す場合には、この配列の要素をその個数だけ並べます。COBOL の PROCEDURE DIVISION USING 句に指定した順番通りに指定します。
実際の COBOL プログラムの呼び出しは cobcall_< 型名 > メソッドを使用します。(COBOL から Java に返される返却値のデータ型に応じて、異なる cobcall_ メソッドが用意されており、これらを使い分ける必要があります。例えば、符号付き整数（COBOL では PIC S9(9) COMP-5）を返す COBOL プログラムは、Java に int 型を返却値として返しますので、この場合は cobcall_int メソッドを使用します。) これらはすべてスタティックメソッドなので使用する前に RuntimeSystem をインスタンス化しておく必要はありません。
各々の cobcall_ メソッドは以下の 2 つのパラメーターを受けます :

1) 呼び出す COBOL プログラムの名前を指定する String オブジェクト
2) 渡すパラメーターとしての Java オブジェクト配列

COBOL に渡すパラメーターは Java データ型から COBOL データ型に変換されます。

呼び出される COBOL プログラムは常に動的にロードされます。静的リンクはできません。このため、プログラムは Micro Focus 形式の .int、.gnt または、Linux/UNIX であれば共有ライブラリ (.so、.sl)、Windows であれば DLL にコンパイルする必要があります。ロードされる COBOL プログラムモジュールの探索経路は、Micro Focus のランタイムシステムの CALL 文の規約に従い、カレントディレクトリ、COBPATH 環境変数などが探索されます。

Java と COBOL との間のデータ型変換用に、COPY 登録集 javatypes.cpy が提供されており、COBOL 側ではこれが提供する各種の型定義を利用します。上の例では、Java の int 型に対応する COBOL の jint 型を使用しています。

3-2. ParameterList 型の使用

Java から COBOL に渡すパラメーターを格納するためのクラスとして、com.microfocus.cobol.lang.ParameterList を使用することもできます。このクラスを使用すると多数のパラメーターを使用する呼び出しのプログラミングを簡潔にすることができます。

ParameterList オブジェクトには add() を使用して、順番にパラメーターを追加して作成します。以下は、上述の例題を ParameterList を使用して書き直したものです。

```
import com.microfocus.cobol.*;
import com.microfocus.cobol.lang.*;

class javamain {
      public static void main(String[] args) {
            try {
                  int outVal = 0;
                  ParameterList parms = new ParameterList();
                  parms.add(new Integer(args[0]));
                  outVal = RuntimeSystem.cobcall ("AddOne", parms);
                  System.out.println(outVal);
            } catch(Exception e) {
                  System.err.println("COBOL 呼び出し失敗 ¥n");
            }
      }
}
```

3-3. String 型の受け渡し

Java の String 型は可変長オブジェクトです。一方 COBOL では PICTURE 句で英数字型データ項目の長さを明記する固定長オブジェクトを使用します。このため、String 型の受け渡しでは特別の配慮が必要となります。

mfcobol.jar パッケージでは、この目的のために com.microfocus.lang.Pointer クラスを提供しています。Pointer オブジェクトは、長さと先頭アドレスを情報として持つオブジェクトです。このオブジェクトを Java から cobcall( ) で渡すと、COBOL の PIC X 型データ項目にマップされて渡されます。Java に返却する際は、Pointer オブジェクトから Byte 配列に取り出すことができます。

以下に例を見てみましょう：

```
import com.microfocus.cobol.*;
import com.microfocus.cobol.lang.*;

class javamain {
      public static void main(String[] args) {
            ParameterList Parms = new ParameterList();
            Parms.add(new Integer(args[0]));
            Parms.add(new Pointer("",40));
            try {
                  RuntimeSystem.cobcall("MYQUERY", Parms);
                  Pointer ptr = (Pointer) Parms.getArgument(1);
                  System.out.println(new String(ptr.getBytes()));
            } catch(Exception e) {
                  System.err.println("COBOL 呼び出しエラー ¥n");
            }
      }
}
```

この Java クラスはコマンド行起動され、コマンド行引数として渡された数値を myquery という COBOL プログラムに第一パラメーターとして渡します。この COBOL プログラムは、渡された数値をキーとして顧客データベース照会を行い、顧客名を第二パラメーターとして Java へ返します。呼ばれる COBOL プログラムは以下のようなものです。

```
       $set sql(dbman=odbc)
        IDENTIFICATION DIVISION.
        PROGRAM-ID. MYQUERY.
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        EXEC SQL INCLUDE SQLCA END-EXEC.
        EXEC SQL INCLUDE Customer END-EXEC.
        COPY JAVATYPES.
        LINKAGE SECTION.
        01 CustID jint.
        01 CustName PIC X(40).
        PROCEDURE DIVISION USING CustID, CustName.
        1.
            EXEC SQL
                CONNECT TO 'VCDemo' USER 'admin'
            END-EXEC.
            MOVE CustID TO Customer-CustID.
            EXEC SQL
                SELECT A.CustID, A.Company INTO
                    :Customer-CustID :Customer-CustID-NULL
                   ,:Customer-Company :Customer-Company-NULL
                FROM Customer A
                WHERE ( A.CustID = :Customer-CustID )
            END-EXEC.
            MOVE Customer-Company TO CustName.
            EXEC SQL DISCONNECT CURRENT END-EXEC.
            EXIT PROGRAM.
```

3-4. COBOL サブルーチンのロード

Java から利用する COBOL サブルーチンは、INT、GNT のほか、共有ライブラリ (UNIX/Linux)、DLL(Windows) にリンクして使用します。このとき、もしエントリ名を COBOL プログラムの内部で宣言するならば、Java から呼び出す前にそのモジュールを明示的にロードしておく必要があります。このために runtime.class は cobload( ) メソッドを提供しています。例えば、mycbl.dll の中にリンクされた COBOL サブルーチンを呼び出すためには以下のようにします :

```
{
      if (cobload("mycbl", null) != 0)
           System.out.println("Could not load library¥n");
      else
           System.out.println("Library loaded successfully¥n");
}
```

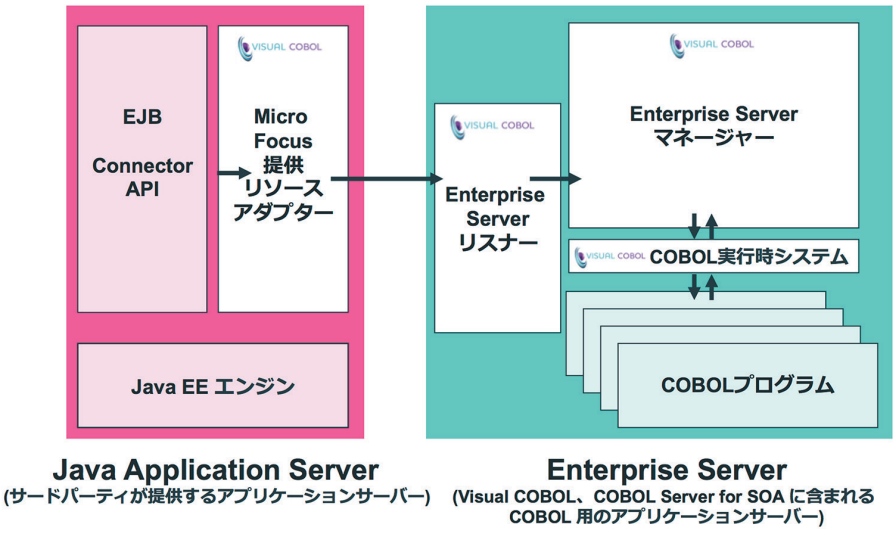
4. Enterprise Server – Java EE Connector としての COBOL

ここまでは、Micro Focus 提供の com.microfocus.cobol.RuntimeSystem クラスを使用する方法について説明しました。この方法を使用してアプリケーションを開発する場合には、COBOL プログラムは「インプロセス」で実行します。COBOL プログラムとの通信は、JNI 呼び出しを通して行います。 つまり、COBOL プログラムは、呼び出し側の Java 仮想マシンと同じプロセスで動作します。インプロセス呼び出しの実行は高速ですが、COBOL プログラムが何らかの理由でクラッシュすると、COBOL プログラムを呼び出した Java アプリケーションを停止させてしまう場合があります。Java EE アプリケーションサーバーは、多くのアプリケーションを実行し、多数のユーザーを処理しますので、呼び出し側の Java アプリケーションを停止させてしまうのは明らかに好ましくありません。COBOL ロジックを Java アプリケーションサーバーの外部に配備して運用し、Java アプリケーション側をピュア Java で構築することでこの点を回避できます。

このように、既存の企業情報システムと Java アプリケーションとを連携させるテクノロジーとしては、Java EE は Java Connector Architecture (JCA) を規定しています。JCA 仕様に準拠して開発された Java アプリケーションサーバーと非 Java アプリケーションは、標準化されたプログラミングによって相互に接続することができます。Java 側では IBM WebSphere、Oracle WebLogic、JBoss などほとんどのアプリケーションサーバー製品がこの仕様に準拠しています。また、非 Java 側もメインフレーム上の CICS、IMS といったミドルウェアや、SAP R/3 などの ERP 製品がこの仕様に準拠した接続を提供しています。

Micro Focus COBOL Server for SOA 製品に含まれる Enterprise Server は、JCA 仕様に準拠した非 Java 側のインターフェイスを提供する COBOL 実行環境です。

JCA 仕様は、非 Java 側が「リソースアダプタ」を提供することを要求しています。リソースアダプタをアプリケーションサーバーに接続し、アプリケーションサーバーと呼び出されるアプリケーションの間の接続を確立して、 接続やトランザクションの処理、 セキュリティ管理を行います。



上に、実行時の構成を図示します。

4-1. なぜ Enterprise Server か？

Enterprise Server を Java EE Connector として使用することによるメリットは以下のとおりです：

1) COBOL ランタイムエラー時の復旧とロギング

Enterprise Server 配下で稼動する COBOL プログラムが異常終了しても、他の実行単位や、呼び出し側の Java EE アプリケーションサーバーは運転を続行します。Java EE 側には例外がスローされます。

2) COBOL ランタイム間のスレッド競合を排除

Enterprise Server 配下のサービスはすべて独立したプロセス内で並行稼動します。このため静的データのスレッド間競合の心配が無く、レガシーな COBOL ロジックを簡単に移行することができます。

3) より高度なデータ型マッピング

mfcobol.jar パッケージでは提供されていなかった、小数部つきの数字項目を BigDecimal に型変換する機能が追加されています。

4) 柔軟なインターフェイスマッピング

開発支援ツール Interface Mapping Toolkit を使用して、既存の COBOL サブルーチンのパラメーター形式を、Java から利用しやすい形にマッピングすることができます。

5) Java EE JTA トランザクションと連携した COBOL トランザクション

COBOL サービスをコンテナ管理トランザクションとしてディプロイすることにより、呼び出し側の Java のトランザクションスコープに、COBOL プログラムのデータベース更新のトランザクション処理を連動させることができます。これによって Enterprise Server をトランザクションモニターとして使用することができます。

以下、これらの機能を説明します。

4-2. Enterprise Server による実行時制御

1) Enterprise Server リポジトリ

Enterprise Server は、きめ細かな構成パラメーターの設定によってチューンアップして運転することができます。その構成の単位としてのサーバーを複数定義することができます。定義情報は Enterprise Server リポジトリに保管されます。

Enterprise Server を最初にインストールすると、出荷時設定で ESDEMO という名前のサーバーがひとつだけ用意されており、これにデフォルトのパラメーターが設定されています。これを使用してテストや検証を行うことができますが、実運用では独自のパラメーター設定に基づいて用途別にサーバー定義を作成して使用します。

2) COBOL コンテナによるセッション管理

Enterprise Server は、個々のサービス要求に対して個別に COBOL アプリケーションを実行するためのコンテナを作成し、この中で COBOL ロジックを実行します。コンテナのための実行プロセスは常駐してプールされその数は構成可能となっています。利用者はハードウェアのスペックとトランザクションのボリウムに応じて、最適なプールサイズを指定することができます。

コンテナは、最初のサービス要求に対して初期化され、COBOL 実行時システムも初期化された状態で開始します。ステートレスなサービス要求に対しては、コンテナは毎回初期化され毎回同じ状態でプログラムを実行します。ステートフルなサービス要求に対しては、セッションが継続する間は同じコンテナが継続して使用されるために、COBOL プログラムの状態は保存されます。

3) トランザクション管理

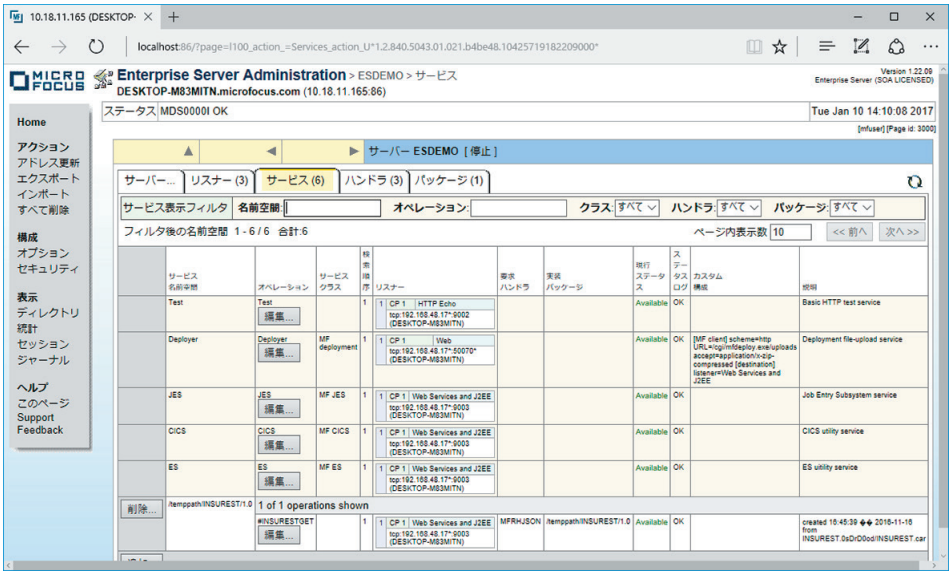
データベースやファイルなどの外部リソースを使用する COBOL アプリケーションは、リソース管理を独自に行うこともできますが、アプリケーションコンテナに管理させることもできます。リソースを独自に管理するサービスは、アプリケーション管理サービスと呼ばれ、COBOL プログラム自身で SQL の COMMIT 文、ROLLBACK 文を発行してトランザクション管理を行うものです。

これに対して、リソースを独自に管理しないサービスを、コンテナ管理サービスと呼びます。これを使用するためには、データベース管理システムが提供するリソースマネージャを、事前に Enterprise Server に定義しておきます。Enterprise Server は XA インターフェイス互換のリソースマネージャに対応しています。サービス要求に対してコンテナが初期化されると、Enterprise Server は、必要なすべてのリソースへの接続を開きます。コンテナ管理サービスを起動すると、Enterprise Server は XA コマンドを使用して、必要なデータベースアクションをすべて実行します。

例えば、COBOL サービスが、10 進データ例外のような非同期のエラーを発生すると、Enterprise Server は自動的にトランザクションをロールバックし、呼び出し側の Java には例外がスローされます。このため Java 側のトランザクションも自然にロールバックされます。

4) 運用管理

Enterprise Server は、Web インターフェイスの管理コンソールを提供しており、管理者はこれを使用してサービスの追加・削除や各種運用オプションの設定・更新、運用状況の監視を行うことができます。一方、コマンドラインのユーティリティも用意されており、スクリプトによる自動運転や、各種運用監視ソリューションとの連携も可能となっています。



Enterprise Server 管理コンソール

5) エラー制御・ジャーナル管理

COBOL プログラムは、コンテナの内部に閉じた環境で実行されます。このため、COBOL プログラムがエラーを発生して異常終了しても他のサービスの実行に影響を与えることはありません。異常終了も含め実行履歴はジャーナルにロギングされ、管理コンソールからモニターすることができます。

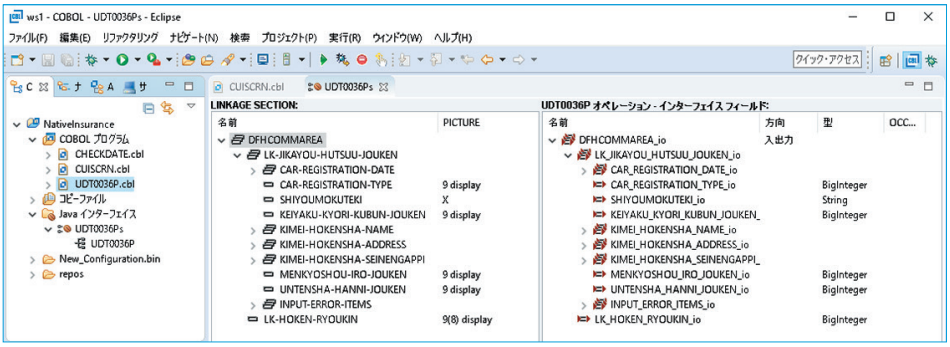
4-3. Interface Mapping Toolkit による開発サポート

COBOL プログラムを Enterprise Server にディプロイするには、別途開発環境で作成しておいた COBOL アーカイブ（.car 拡張子）を手動でディプロイすることもできますが、Visual COBOL が提供する開発環境の中で、自動的に EJB ラッピングしてリモートディプロイすることもできます。これには Interface Mapping Toolkit を使用します。

このツールは以下の 4 つの機能を提供します：

1) 既存 COBOL プログラムのパラメーターマッピング

既存の COBOL サブルーチンの LINKAGE SECTION 中に書かれたパラメーター定義を逆解析し、EJB のセッション Bean メソッドのパラメーターにマッピングします。GUI 操作によって必要なパラメーターを必要な形式でマップすることができます。集団項目をまとめて Java Bean にマップすることも可能です。小数部分を含む数字データ項目は BigDecimal にマップさせます。



2) EJB の自動生成

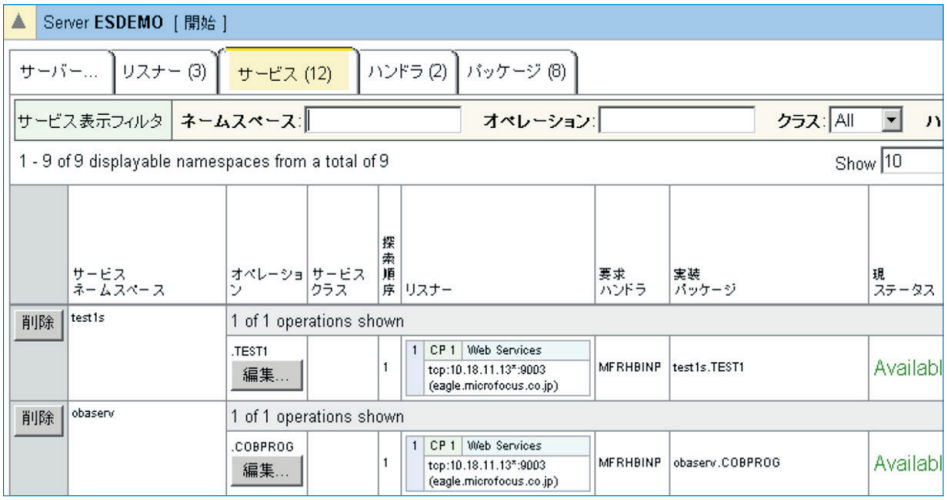
マップ情報に基づいて EJB を自動生成します。生成された EJB は内部的に Enterprise Server に対する Java EE 準拠の Common Client Interface による接続と呼び出しを含んでいますが、Java 側の利用者はその詳細を知る必要は無く、単にセッションビーンのメソッドの呼び出しとして扱うことができます。EJB は Java アーカイブにパッケージされて生成されるので、これをそのまま、または .ear にパッケージした上で、使用する Java アプリケーションサーバーにディプロイします。

オプションとしてディプロイ先の Java アプリケーションサーバーの種別を指定すれば、指定に合致した形式のディプロイメント記述子を含む EJB JAR を生成します。

3) COBOL アーカイブの生成とディプロイ

Interface Mapping Toolkit の中から簡単な GUI 操作で、指定された COBOL プログラムを COBOL アーカイブにパッケージ化し、指定された Enterprise Server にリモートディプロイすることができます。

このときディプロイメントの属性として、トランザクション属性やセッション永続性などを指定します。
ディプロイされたパッケージは、以下のように登録済み COBOL サービスとして、Enterprise Server 配下で利用可能になります。

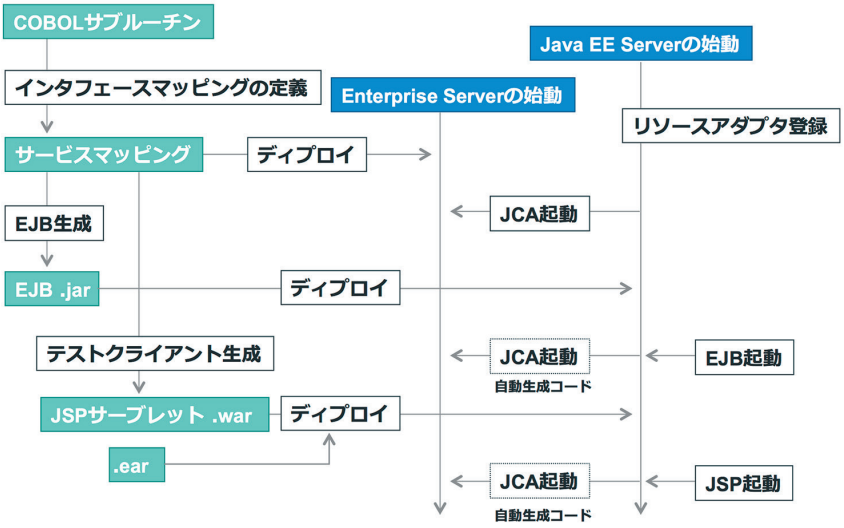


ディプロイされた COBOL サービスは、(2) で自動生成された EJB を使用して呼び出すこともできますし、Connector API を直接使用して 任意の Java EE アプリケーションから呼び出すことができます。

4) テスト用クライアントアプリケーションの生成

対話型でパラメーターの値を受け取り、EJB のメソッドを呼び出して結果を表示するような簡単な Servlet モジュールを含む、.ear パッケージを自動生成します。これをそのままお使いの Java アプリケーションサーバーにディプロイでき、稼動確認を行うことができます

以下に開発の流れを示します：



4-4. Connector API による COBOL サービスの呼び出し

ディプロイされた COBOL サービスを Java EE から利用するためには、Connector Architecture が定める所定の Java API を使用します。アプリケーションサーバーにディプロイされたリソースアダプタは、JNDI に登録されており、最初にこれを Lookup するとことから始まります。以下にそのプログラミング例を示します：

接続ファクトリーの Lookup

```
Context ic = new InitialContext();
ConnectionFactory cf = ConnectionFactory
ic.lookup("java:/eis/MFCobol_v1.5");
```

接続の取得

```
javax.resource.cci.Connection = null;
con = cf.getCobolConnection(con);
```

インタラクションの作成

```
Interaction ix = con.createInteraction();
CobolInteractionSpec iSpec = new CobolInteractionSpec();
iSpec.setFunctionName("ReadCustS.READCUST");
iSpec.setArgument(0, RuntimeProperties.BY_REFERENCE);
iSpec.setArgument(1, RuntimeProperties.BY_REFERENCE);
```

COBOL に渡すパラメーターの作成

```
RecordFactory rf = cf.getRecordFactory();
IndexedRecord iRec = rf.createIndexedRecord("…");
IndexedRecord oRec = rf.createIndexedRecord("…");
iRec.add(new Integer(custid1));
iRec.add(custname);
```

COBOL サービスの呼び出し

```
ix.execute(iSpec, iRec, oRec);
```

接続の切断

```
ix.close();
```

4-5. 自動生成された EJB の呼び出し

Interface Mapping Toolkit が自動生成した EJB には、マッピング時に付けたオペレーション名と同名のメソッドが含まれています。このメソッドは、マッピング時に「入力」または「入出力」と定義したパラメーターを引数として受け取り、「出力」または「入出力」と定義したパラメーターを、ArrayList オブジェクトとして返します。それ以外は標準的な EJB の呼び出し方で呼び出すことができます。以下にその例を示します：

```
EJB インタフェース変数の宣言
private com.mypackage.ReadCustS.ReadCustS ri = null;
private com.mypackage.ReadCustS.ReadCustSHome hi = null;

EJB の lookup
InitialContext initCtx = new InitialContext();
Object objRef =
initCtx.lookup("java:comp/env/ejb/ReadCustSEJB");

コンポーネントインタフェースの作成
hi = (com.mypackage.ReadCustS.ReadCustSHome)
javax.rmi.PortableRemoteObject.narrow(objRef,
com.mypackage.ReadCustS.ReadCustSHome.class);
ri = hi.create();

ラッパーメソッドの呼び出し
java.util.ArrayList rv = ri.READCUST(
     ((Integer)jspBean.getREADCUST_Custid1AsObject()).intValue(),
       (String)jspBean.getREADCUST_CustnameAsObject(),
       (String)jspBean.getREADCUST_CustcompanyAsObject(),
       (String)jspBean.getREADCUST_CustemailAsObject());
```

5. Enterprise Server 開発上の留意点

ここでは、Enterprise Server の Java EE 連携を使用した現実のシステム開発で予測される問題のいくつかについて補足します。

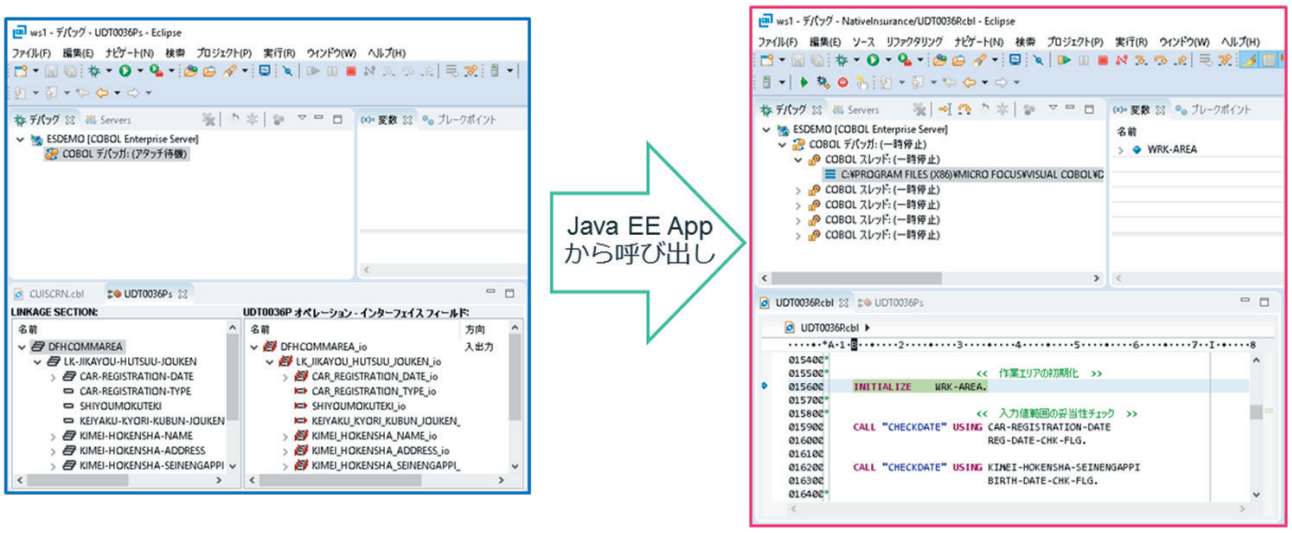
5-1. コマンドラインの運用

Enterprise Serverリポジトリに定義済みのサーバーの運転は、Enterprise Server AdministrationのWeb画面で行うことができます。しかし自動運転を行うためにコマンド行ユーティリティも提供されています。

casstart /r< サーバー名 >
指定されたサーバーを開始します
casstop /r< サーバー名 >
指定されたサーバーを停止します

5-2. COBOL のデバッグ

Enterprise Server 配下にディプロイされた COBOL サービスをデバッグするには、Visual COBOL Eclipse IDE のデバッグ機能が利用できます。COBOL サービスをディプロイした Eclipse 開発プロジェクトの中で、デバッグ構成として対象の Enterprise Server インスタンスを指定してデバッグセッションを開始すると該当の COBOL サービスの実行を待機します。この状態で Java 側からサービス要求を発行すれば、Eclipse デバッガーによるデバッグが可能となります。



5-3. 効果的なインターフェイスマッピング

Interface Mapping Toolkitを使用して既存のCOBOLサブルーチンのLINKAGEパラメーターを、Javaから呼び出しやすい形式のマッピングに定義できることは既に見ました。ここではデフォルトマッピングとして、自動的に基本データ項目単位で決まったデータ型対応規則に基づいてマッピングを作成することができます。しかし、通常のCOBOLサブルーチンでは、パラメーターはCOBOLのレコード形式になっている場合が多く、しかも一般に項目数が非常に多くなります。その中には、文字型とパック十進・バイナリとが混在している場合もあります。また、REDEFINES や OCCURS もかかれています。これらを、一つ一つの基本データ項目に分割して、パラメーターとして Java とやりとりすることは効率的ではありません。

通常、Oracle JDK や各種サードベンダーから提供されている Java API のメソッドのパラメーターは、最大でも 20 を超えることは無く、それ以上の数のパラメーターを持つメソッドを、Java プログラマに提供することは、呼び出し側の開発生産性や品質に悪影響を与えます。従って、ほとんどの場合、既存の COBOL サブルーチンに対してデフォルトマッピングは使用するべきではありません。

1) 再利用マッピング

Interface Mapping Toolkit の再利用マッピングは、COBOL のレコードを Java の Bean にマップさせる機能です。これを使用すると、例えば 100 件の基本項目からなる COBOL のレコードを、100 個のメンバーを持つ Java Bean にマップさせることができます。Java から COBOL を呼び出す際には、この Java Bean のインスタンスをパラメーターで渡すことができますので、Java 側のコーディングの負担を軽減することができます。

2) 入出力の区分

マッピングの定義において個々のパラメーターについて、入力パラメーター・出力パラメーター・入出力パラメーターの区分を指定することができます。COBOL 言語には、LINKAGE パラメーターの入出力区分を指定する機能はありませんので、Interface Mapping Toolkit はデフォルトで「入出力」の区分を与えています。しかし、出力パラメーターとしてしか使用していないパラメーターを入出力でマップすると、呼び出し側の Java で必要も無いパラメーターのオブジェクトを用意しなければならなくなりますので、開発生産性のみならず実行性能にも悪影響があります。

個々のパラメーターの入出力区分は、既存アプリケーションについての知識からのみ知ることができます。これをすべてデフォルトで運用してしまうことは望ましくありません。

3) 複数マッピングの定義

Interface Mapping Toolkit では、ひとつの COBOL サブルーチンに対して、複数の異なるマッピングを、名前を変えて定義することができます。これを使用すると、同じプログラムを用途に応じて異なる呼び出し方をすることができます。

この機能は LINKAGE パラメーター中に REDEFINES がある場合には特に有効です。また、あるフィールドの値によって呼び出しの機能が変わってくるようなプログラムの場合にも、それぞれに対して別名をつけてマッピングすることができます。

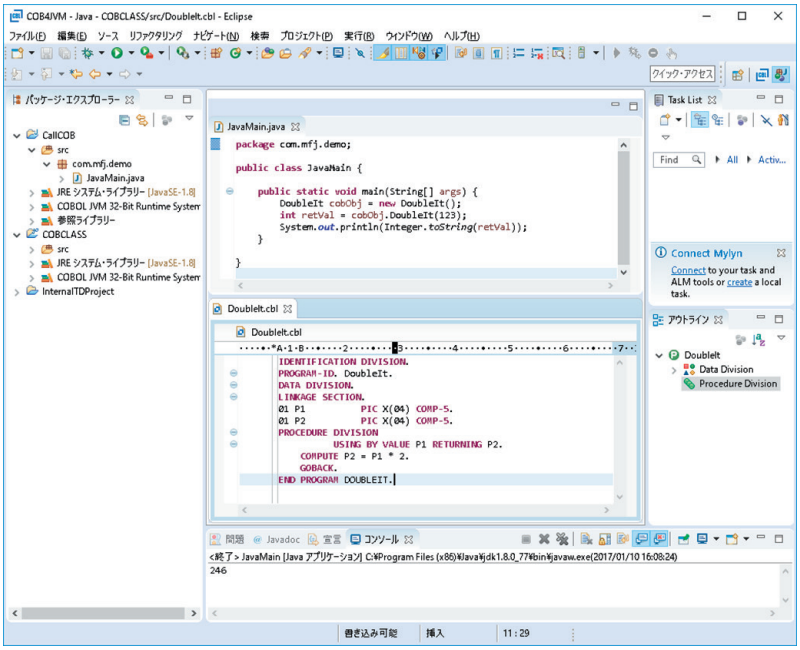
4) 使用しないパラメーター、固定値を渡すパラメーター

既存の COBOL プログラムを流用するときに、必ずしもすべての機能が流用の対象にはなりません。このためパラメーターのうちの一部は使用しないかも知れません。このようなパラメーターはマッピングの対象外にしたり、固定の規定値を与えておくことができます。これによって呼び出し側の Java では意識しなくても良くなります。

6. JVM COBOL

これまでに見てきた方法は、いずれも JVM 内で稼働する Java アプリケーションから JVM 外で稼働する COBOL アプリケーションにアクセスするものでした。一方、Micro Focus Visual COBOL は、COBOL プログラムを JVM 上で稼働する Java バイトコードにコンパイルする JVM COBOL テクノロジーをサポートしています。これを使用して Java アプリケーションと COBOL アプリケーションが同一の JVM 内で相互に呼び出しあうことができます。

Visual COBOL は Eclipse ベースの IDE を持っており、Java 言語の開発と COBOL 言語の開発を一つの開発環境の中で共存させることができます。以下に、同じワークスペース内で Java と COBOL を混在させている様子を示します。



この例で示すように、COBOL プログラムはクラス構造でプログラミングされる必要はありません。従来型の手続き型の COBOL プログラムをコンパイルすると、それはその名前の JVM クラス内のインスタンスメソッドとしてコンパイルされますので、Java 側からインスタンス化してメソッドとして呼び出すことができます。

6-1. COBOL – Java 間のパラメーター授受

上の例で見ましたように JVM COBOL で生成された手続き型の COBOL プログラムは COBOL であるということを Java 側で意識することなく、呼び出しができます。しかし、COBOL – Java の連携を考えた場合、COBOL 固定長文字列や十進数値データなど相互で対応しないデータ型があります。JVM 環境で再利用したい COBOL にこのようなパラメーターが定義されている場合、特別な考慮が必要となりますが、再利用する COBOL プログラムに手を入れずに Java 側から呼び出す手段が Visual COBOL には備えられています。本項ではその代表的な2つの方法を紹介します。

ここでは、以下のようなパラメーターを持つ COBOL を例にとってそれぞれのパラメーター変換技法を紹介します。

```
       PROGRAM-ID. COBOLPGM.
       DATA DIVISION.
       LINKAGE SECTION.
       01 LNK-SNGL PIC 9(3) COMP-3.
       01 LNK-GRP.
          03 LNK-GRP-ONE          PIC X(10).
          03 LNK-GRP-TWO          PIC 9(5) COMP-5.
       PROCEDURE DIVISION USING LNK-SNGL
                                LNK-GRP.
           DISPLAY "LNK-SNGL    : " LNK-SNGL.
           DISPLAY "LNK-GRP-ONE: " LNK-GRP-ONE.
           DISPLAY "LNK-GRP-TWO: " LNK-GRP-TWO.
           GOBACK.
```

1） COBOL ラッパープログラムによる変換

JVM COBOL は JVM 内で稼働するクラスを記述できるという意味では Java 言語と変わることはありません。同機能は ISO 2002 国際標準を独自に JVM アプリケーション開発向けに拡張した Micro Focus 方言を利用できます。この方言では下表のような Java の基本型や java.lang.String にマッピングした特別な COBOL のデータ型を用意しています。

Java	COBOL
byte	binary-char
short	binary-short
int	binary-long
long	binary-double
float	float-short
double	float-long
boolean	condition-value
char	character
java.lang.String	string

1 つ目の変換技法は、この JVM の基本型に対応した型を引数に持つ COBOL プログラムを用意し、それを Java と再利用したい COBOL の間のインターフェイスとして利用する方法です。Java が直接呼び出すのはそのインターフェイスであり、その中で Java から受け取ったデータを再利用する COBOL プログラムのパラメーターに合わせて変換します。そのインターフェースプログラムは変換した COBOL 型のデータをパラメーターとして利用し COBOL を通常の CALL 文にて呼び出します。以下はこの技法による実装例です。

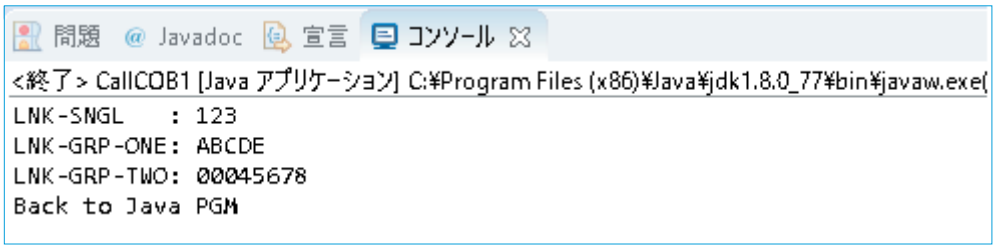
インターフェイスとして利用する COBOL プログラム：

```
       PROGRAM-ID. COBWRAPPER.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01 LNK-SNGL PIC 9(3) COMP-3.
       01 LNK-GRP.
          03 LNK-GRP-ONE          PIC X(10).
          03 LNK-GRP-TWO          PIC 9(5) COMP-5.
       LINKAGE SECTION.
       01 LNK_SNGL                binary-short.
       01 LNK_GRP_ONE             string.
       01 LNK_GRP_TWO             binary-long.
       PROCEDURE DIVISION USING    BY VALUE LNK_SNGL
                                   BY VALUE LNK_GRP_ONE
                                   BY VALUE LNK_GRP_TWO.
           MOVE LNK_SNGL    TO LNK-SNGL.
           MOVE LNK_GRP_ONE TO LNK-GRP-ONE.
           MOVE LNK_GRP_TWO TO LNK-GRP-TWO
           CALL "COBOLPGM"  USING LNK-SNGL
                                  LNK-GRP.
           GOBACK.
```

呼び出し側の Java プログラム：

```
public class CallCOB1 {
      public static void main(String[] args) {
           short  p1  = (short)123;
           String p2 = "ABCDE";
           int    p3 = 45678;
           COBWRAPPER cobpgm = new COBWRAPPER();
           cobpgm.COBWRAPPER(p1,  p2,  p3);
           System.out.println("Back to Java PGM");
      }
}
```

実行結果：

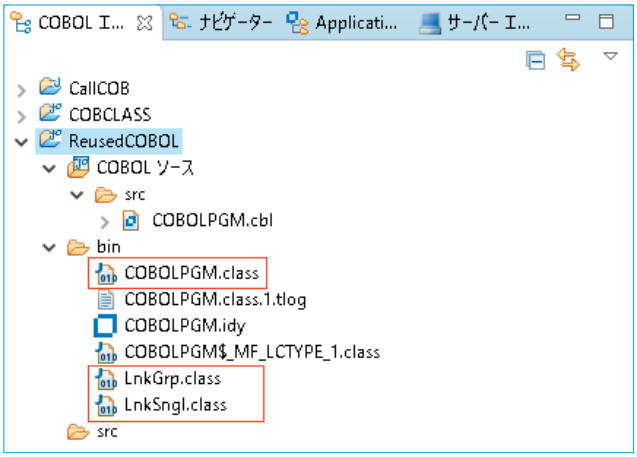


2）SMARTLINKAGE による変換モジュールの自動生成

上で紹介した変換技法では再利用する COBOL プログラムに手を加える必要はありませんでしたが、別途新たなプログラムを用意する必要がありました。SMARTLINKAGE とはこのインターフェイス目的で利用される変換プログラムをも Visual COBOL に自動生成させてしまう機能になります。この技法を使えば、再利用したい COBOL プログラムに COBOL 固有のデータ型を持つパラメーターが定義されていようとも Java 側、COBOL 側ともに特別な対応をすることなくシームレスな連携が可能になります。

この SMARTLINKAGE 機能を利用するには、まず COBOL プログラムを ILSMARTLINKAGE 指令を指定してコンパイルします。これにより、LINKAGE SECTION 中に定義された 01 レベルの変数単位でクラスが自動生成されます。このクラスこそが Java 、COBOL 間のデータ型の差分を吸収する仕組みが実装されたクラスとなります。このクラスには各 COBOL の変数単位に Java の基本型及び String の引数／戻り値を持つ setter 及び getter が実装されています。Java はこのクラスの setter、getter を通じてパラメーターの授受をします。COBOL プログラムを呼び出す際はこの自動生成されたクラスの参照を渡します。

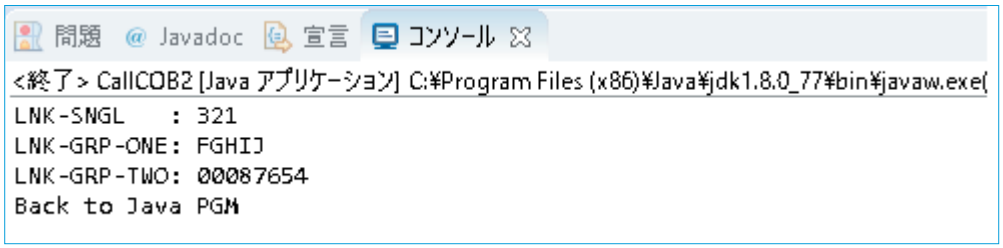
先ほどのプログラムを ILSMARTLINKAGE 指令を指定してコンパイルすると下図のように COBOL プログラムに対応するクラスファイルに加え、LINKAGE SECTION の 01 レベルに定義された LNK-SNGL 及び LNK-GRP に対応するクラスも併せて生成されています。



Java では例えば以下のようにして SMARTLINKAGE を使った COBOL プログラムを呼び出します。

```
public class CallCOB2 {
      public static void main(String[] args) {
           LnkSngl p1 = new LnkSngl();
          p1.setLnkSngl((short)321);
          LnkGrp p2 = new LnkGrp();
          p2.setLnkGrpOne("FGHIJ");
          p2.setLnkGrpTwo(87654);
           COBOLPGM cobpgm = new COBOLPGM();
           cobpgm.COBOLPGM(p1, p2);
        System.out.println("Back to Java PGM");
      }
}
```

実行結果



SMARTLINKAGE 機能により自動で生成されるクラスは下表のマッピングルールに基づいて変換機能を提供します。このマッピングルールと異なる変換をしたい場合は、1つ目の技法として紹介した作法で対応する必要があります。

COBOL のデータ型 Native	Java のデータ型
PIC X(n)	binary-char
PIC S9(n) … n <= 2	binary-short
PIC 9(n) … n <= 2	binary-long
PIC S9(n) … 2 < n <= 4	binary-double
PIC 9(n) … 2 < n <= 4	float-short
PIC S9(n) … 4 < n <= 9	float-long
PIC 9(n) … 4 < n <= 9	condition-value
PIC S9(n) … 9 < n <= 19	character
PIC 9(n) … 9 < n <= 19	string
PIC 9(n)V9(m)	java.math.BigDecimal もしくは com.microfocus.cobol.program.ScaledInteger
COMP-1	float
COMP-2	double
数値編集項目	String
PIC N(n)	String
PIC A(n)	String
集団項目	String

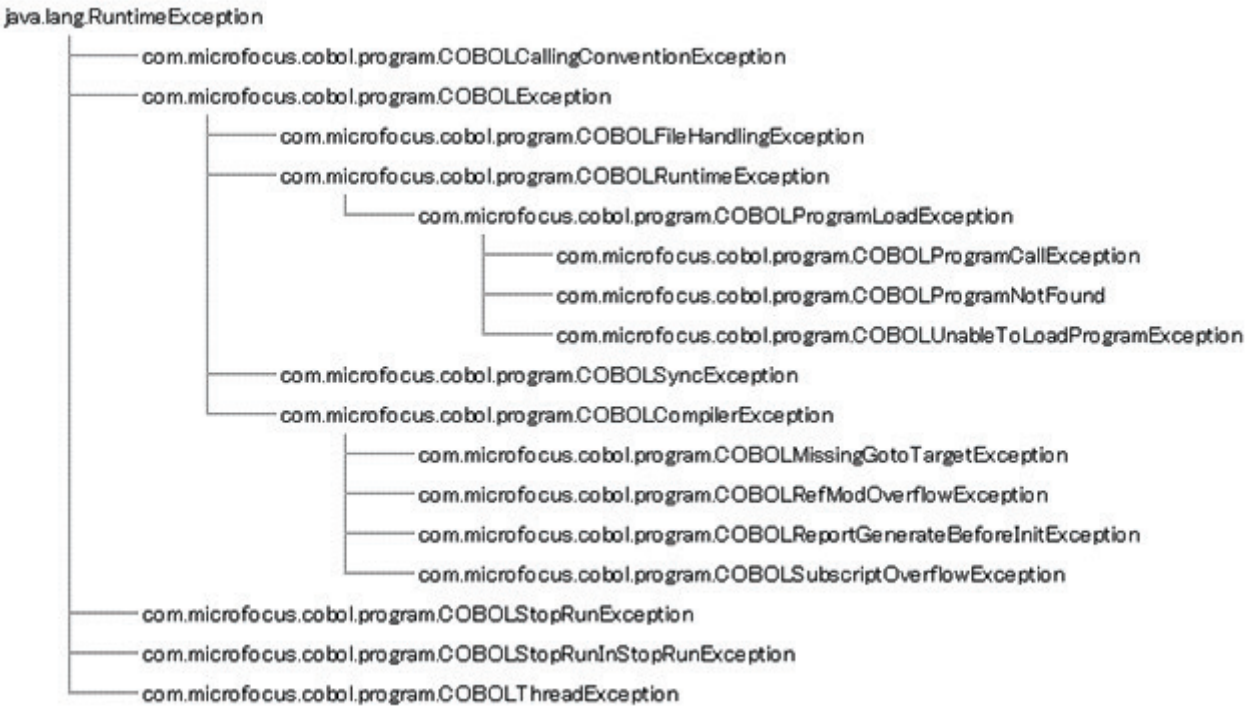
（注）PIC 9 および PIC S9 の項目は USAGE 句に関わらず共通です。

6-2 COBOL ロジック中で発生する例外のハンドリング

Java では実行時エラーが発生した場合、Java 実行環境が例外を投げます。プログラム側でこの例外に対し何も対策をしていないと、プログラムはそこで強制終了となります。最悪の場合システム自体が停止することも想定されます。そのため、一般的な Java アプリケーションはアプリケーション全体のプロセスに影響が及ばないよう例外処理を実装します。

JVM COBOL で開発された COBOL アプリケーションは JVM 上で稼働する JVM クラスであり、COBOL 内で実行時エラーが発生する際は、Java と同様に例外が投げられます。下図は COBOL より投げられる主な例外の一覧になります。

Visual COBOL から投げられる主な例外の一覧



COBOL アプリケーションの実行時エラーは例外として扱われるため、COBOL を呼び出す Java 側では下記のように Java の一般的なコーディングで COBOL の例外ハンドリングをプログラムに実装できます。

添え字範囲外のエラーを発生させるロジックを含んだ COBOL プログラム：

```
       PROGRAM-ID. COBLib1.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01 A                PIC X    OCCURS 2.
       01 I                PIC 9    VALUE 3.
       PROCEDURE DIVISION.
           MOVE SPACE TO A(I).
           GOBACK.
```

例外ハンドリングを実装した COBOL を呼び出す Java プログラム：

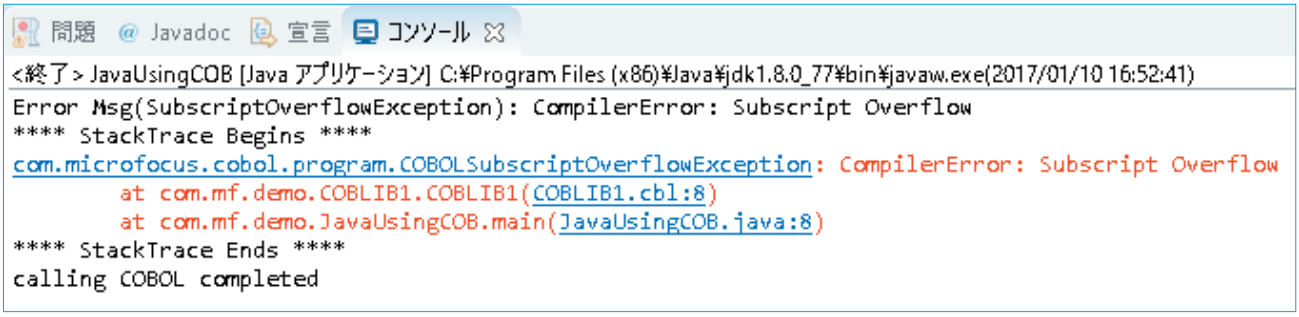
```
import com.microfocus.cobol.program.COBOLSubscriptOverflowException;
import com.microfocus.cobol.program.COBOLException

public class JavaUsingCOB {

      public static void main(String[] args) {
           try{
                 COBLib1 cobclass = new COBLib1();
                 long res = cobclass.COBLib1();
           } catch (COBOLSubscriptOverflowException SOE) {
                 System.out.println("Error Msg(SubscriptOverflowException): "+ SOE.getMessage());
                 System.out.println("**** StackTrace Begins ****");
                 SOE.printStackTrace();
                 System.out.println("**** StackTrace Ends ****");
           } catch (COBOLException CE) {
                 System.out.println("Error Msg: " + CE.getMessage());
                 System.out.println("**** StackTrace Begins ****");
                 CE.printStackTrace();
                 System.out.println("**** StackTrace Ends ****");
          }
           System.out.println("calling COBOL completed");
      }

}
```

実行結果：



COBOL プログラム側で実行時エラーが発生していても例外ハンドリングを実装しているため、Java 側の処理を継続できていることが確認できます。

6-3. マルチユーザーアプリケーションの開発

JSP/Servlet のような Web Application を開発する場合、マルチユーザーからの同時アクセスを考慮する必要があります。このようなアプリケーションはそれぞれのユーザーに対してセッションを持たせる等して処理の一貫性を管理しますが、COBOL アプリケーションの多くはそのような点は考慮されていません。そのため、各スレッドがアドレス空間を共有するデフォルトの状態で COBOL アプリケーションを Web Application 内で運用するとセッション毎によるメモリの分離が実現できません。Visual COBOL はこのような条件下での COBOL アプリケーションの運用を想定して RunUnit という機能を提供しています。この RunUnit 機能を活用することで、メモリやファイルロックテーブルを単一プロセス内で分離することが可能です。これにより、シングルユーザーによる利用を想定して開発された COBOL プログラムであっても Web Application のようなマルチユーザーアプリケーション上に容易にマイグレーションできるようになります。尚、RunUnit を使ったマルチユーザーアクセス対応は、COBOL アプリケーションの書き換えを伴いません。

以下は、データベースへ接続を確立し5秒スリープさせる COBOL プログラムを各スレッド単位に RunUnit を使って処理を分離させる例題プログラムになります。Web Application にて各セッション毎に RunUnit で COBOL の処理を分離させるよう作りこむのであれば、例えば com.microfocus.cobol.runtimeservices.RunUnit をセッション変数として登録し、そのセッション内で使いまわす等といったかたちで応用します。

COBOL プログラム：

```
$SET SQL(DBMAN=JDBC)
$SET CONSTANT driverClass "com.microsoft.sqlserver.jdbc.SQLServerDriver"
$SET CONSTANT databaseURL1 "jdbc:sqlserver://localhost;"
$SET CONSTANT databaseURL2 "databaseName=XXX;user=YYY;password=ZZZ"
 PROGRAM-ID. COBDBPROC.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
   EXEC SQL include sqlca END-EXEC.
 01 CONNECTIONSTRING PIC X(300) VALUE SPACES.
 01 LOOP-F           PIC X(10).
 01 TIME-WK.
    03  TIME1.
        05  TIME1-9  PIC 9(6).
    03  TIME2.
        05  TIME2-9  PIC 9(6).
    03  TIME3        PIC 9(6).
 PROCEDURE DIVISION.
 1.
     MOVE "Driver=" & driverClass & ";URL=" & databaseURL1 & databaseURL2
          TO CONNECTIONSTRING.
     EXEC SQL CONNECT USING :CONNECTIONSTRING END-EXEC.
     DISPLAY "CONNECTED".
     ACCEPT TIME1  FROM TIME.
     MOVE SPACE TO LOOP-F.
     PERFORM UNTIL LOOP-F = 'END'
          ACCEPT TIME2  FROM TIME
          COMPUTE TIME3 = TIME2-9 - TIME1-9
          IF TIME3 > 5
             MOVE "END" TO LOOP-F
          END-IF
     END-PERFORM.
     EXEC SQL DISCONNECT CURRENT END-EXEC.
     DISPLAY "DISCONNECTED".
     GOBACK.
```

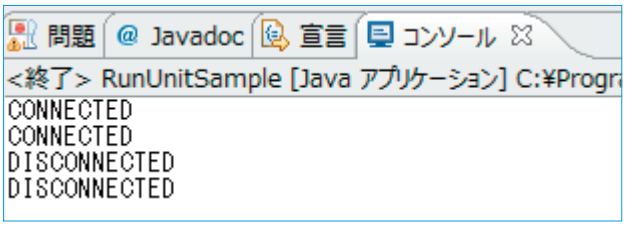
呼び出し側の Java プログラム：

```
import com.microfocus.cobol.runtimeservices.RunUnit;

public class RunUnitSample {
	public static void main(String[] args) {
		SmplThread1 thread1 = new SmplThread1();
		SmplThread1 thread2 = new SmplThread1();
		thread1.start();
		thread2.start();
	}
}

class SmplThread1 extends Thread{
	public void run() {
		 RunUnit ru = new RunUnit();
	COBDBPROC cobproc = new COBDBPROC();
		ru.Add(cobproc);
		try{
			cobproc.COBDBPROC();
		} catch(RuntimeException RE){
			System.out.println(RE.getMessage());
		}finally{
			ru.StopRun();
		}
	}
}
```

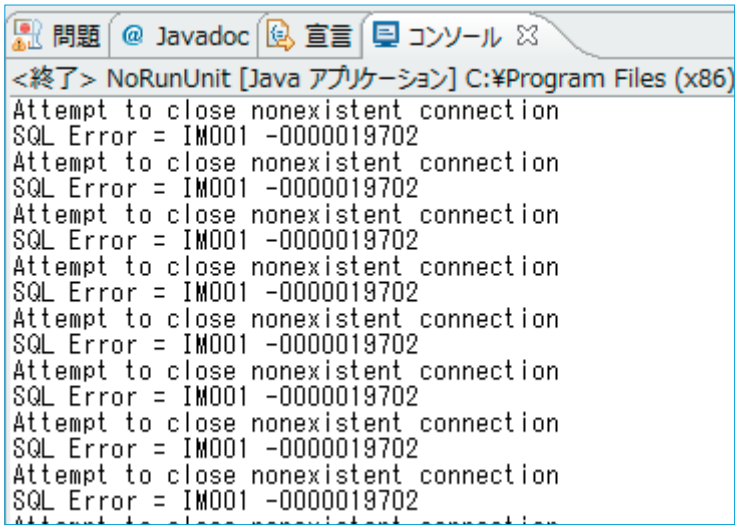
実行結果：



SmplThread1 クラス中のメソッドを以下のように RunUnit で分離しないように作り変えてみます。本稿執筆の際に使用した環境では最初の DISCONNECT にて他方のスレッドで使用している接続も切断され、2つ目のスレッド中の DISCONNECT 文にて切断対象の接続がない旨のエラーとなりました。

```
	public void run() {
		COBDBPROC cobproc = new COBDBPROC();
		cobproc.COBDBPROC();
		try{
			:
	}
```

実行結果：



6-4. COBOL ロジックの Java 向けラッピング

既存 COBOL ロジックをなるべくそのまま再利用したいという要望は当然あります。しかし、単独のサブルーチンを単純にメソッドとして呼び出すのは、COBOL 側から見ればもっとも簡単な方法に見えますが、呼び出し側の Java アプリケーションから見れば必ずしも最適な利用方法ではありません。それは、この方法が Java が提供する豊かなオブジェクトモデルを何も使用していないことによります。例えば Java らしいプログラミングでは以下のような設計がなされています。

● オブジェクトプロパティとしての外部参照
一般に COBOL の CALL 文による副プログラム呼び出しでは、プログラムにパラメーターとして値を渡し処理結果をパラメーターとして受け取ります。これらは、呼び出しの際にのみローカルに使用されるパラメーター領域ではなく、オブジェクトのライフサイクルの間永続的に存在し、外部から参照・更新が可能なデータを、プロパティとして持つことができます。

● コンストラクタによるオブジェクト初期化
オブジェクトがインスタンス化される際に行われる初期化処理（ファイルの OPEN、領域の初期値設定など）は、コンストラクタと呼ばれる特殊なメソッドに集約させることができます。こうしておくことによって呼び出し側 Java 言語は new 操作を行うことで、COBOL 側の初期化処理を自動的に起動できるようになります。

● 例外スローによるエラー通知
COBOL でのサブルーチン呼び出しでは、呼び出し側にエラーが発生したことを通知するためには、パラメーターにエラーコードを設定して返す方法が一般的です。一方 Java では、呼び出したメソッドがスローする例外をキャッチして処理する方法が一般的です。Java の世界で COBOL ロジックを活用する際に、COBOL プログラミングの常識を押し通すことは、呼び出し側の Java プログラマにとっては負担になります。

手続き型のプログラム構造のまま COBOL を維持することももちろん可能ですが、上記のような理由から段階的に Java の世界と親和性の高い構造に作り変えていくことも1つのストラテジです。以下、上述のような Java らしいプログラミングを、COBOL ロジックの再利用で活用する方法について、チュートリアルで見てみます。

6-5. JVM COBOL ラッピングのチュートリアル

一例として以下のような簡単な例題を考えます。

```
       FILE-CONTROL.
           SELECT CUST-MASTER ASSIGN TO "CUST.dat"
           ORGANIZATION INDEXED RECORD KEY FS-CUSTID
           ACCESS MODE RANDOM.
       DATA DIVISION.
       FILE SECTION.
       FD CUST-MASTER.
       01 CUST-REC.
           05 FS-CustId        PIC X(4) COMP-5.
           05 FS-CustName      PIC X(30).
           05 FS-CustCompany   PIC X(30).
           05 FS-CustEmail     PIC X(30).
       LINKAGE SECTION.
       01 CustId               PIC X(4) COMP-5.
       01 CustName             PIC X(30).
       01 CustCompany          PIC X(30).
       01 CustEmail            PIC X(30).
       PROCEDURE DIVISION
                 USING CustId CustName CustCompany CustEmail.
       1.
           OPEN I-O CUST- MASTER.
           MOVE CustId TO FS-CustId.
           READ CUST-MASTER INVALID CONTINUE.
           CLOSE CUST-MASTER.
           MOVE FS-CustName    TO CustName.
           MOVE FS-CustCompany TO CustCompany.
           MOVE FS-CustEmail   TO CustEmail.
           EXIT PROGRAM.
```

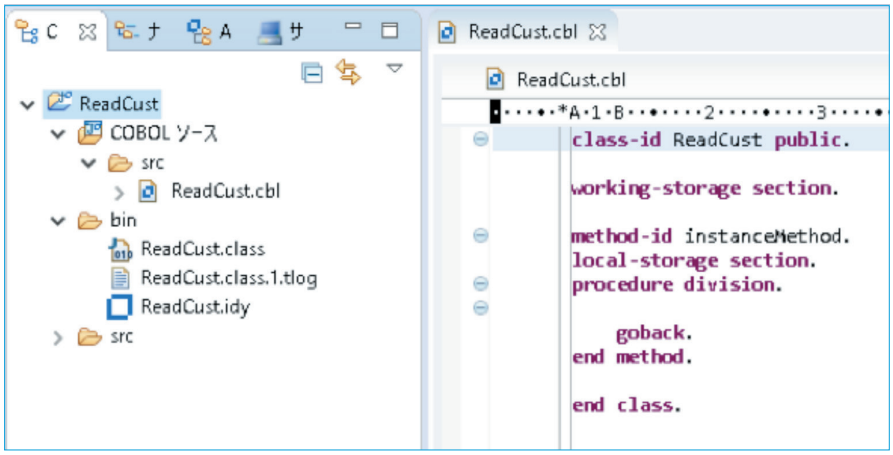
この簡単な COBOL プログラムは、第一パラメーターとして顧客番号を受け取り、顧客マスターの索引ファイルから該当するレコードを読んでその顧客名、会社名、メールアドレスを返すというものです。このプログラムを Java クラス化するにあたって以下の方針は妥当なものといえます。

- ファイル定義はメソッドの外に出し、クラスの内部プロパティとする。
- 顧客番号、顧客名、会社名、Email は、それぞれクラスの外部プロパティとしてエクスポートする。
- ファイルの OPEN はクラスのコンストラクタの中で行う。
- ReadCust メソッドを用意し、プロパティに設定された顧客番号で顧客マスターファイルを読み、顧客名、会社名、Email のプロパティに値を設定する。
- READ で失敗した場合は呼び出し側に例外をスローする。

以下、この方針に従って JVM のクラスを COBOL 言語で作成する手順を見てゆきます。

1) Visual COBOL Eclipse IDE を起動し、[ファイル] > [新規] > [COBOL JVM プロジェクト] でプロジェクトを新規作成します。

2) プロジェクトを右クリックし、[新規作成] > [COBOL JVM クラス] を選択し、適切な名前を付けてクラスのテンプレートを作成します。以下の通りテンプレートが作成されます：



3) 再利用する COBOL プログラムからファイル定義部分を抽出し、メソッドの外側にコピーします。そして、コンストラクタを表す「new」というメソッド名でファイルの OPEN を実装します。これでプログラムは以下のようになります。

```
Class-id ReadCust public.
FILE-CONTROL.
  SELECT CUST-MASTER ASSIGN TO "CUST.dat"
         ORGANIZATION INDEXED RECORD KEY FS-CUSTID
ACCESS MODE RANDOM.
data division.
FILE             SECTION.
FD CUST-MASTER.
01 CUST-REC.
    05 FS-CustId           PIC X(4)    COMP-5.
    05 FS-CustName         PIC X(30).
    05 FS-CustCompany      PIC X(30).
    05 FS-CustEmail        PIC X(30).
method-id new.
procedure division.
    OPEN INPUT CUST-MASTER.
    goback.
end method.
end class ReadCust.
```

4) クラスの外部プロパティはメソッドの外側に WORKING-STORAGE SECTION として記述します。外部プロパティの名前は、COBOL の PROPERTY AS 句で明示的に指定します。

```
WORKING-STORAGE SECTION.
01 CustId       binary-long Property As "CustId".
01 CustName     String      Property As "CustName".
01 CustCompany  String      Property As "CustCompany".
01 CustEmail    String      Property As "CustEmail".
```

5) ReadCust メソッドを実装します。READ 文を実行し、レコードが存在すればレコード中の各データ項目を外部プロパティに設定します。INVALID KEY 条件が発生したら例外をスローします。

COBOL では System.Exception の例外オブジェクトを、RAISE 文を使用してスローすることができます。RAISE 文は ISO2002 国際標準で COBOL 言語に追加された COBOL 文法です。メソッドは以下の通りとなります。

```
 method-id. ReadCust.
procedure division.
    MOVE CustId TO FS-CustId.
    READ CUST-MASTER INVALID KEY
        CLOSE CUST-MASTER
        RAISE type java.lang.Exception::"new"(" レコードなし ")
    NOT INVALID KEY
        MOVE FS-CustName TO CustName
        MOVE FS-CustCompany TO CustCompany
        MOVE FS-CustEmail TO CustEmail
    END-READ.
    CLOSE CUST-MASTER
    goback.
end method.
```

プログラムは以下の通りになりました。

```
Class-id ReadCust public.
FILE-CONTROL.
   SELECT CUST-MASTER ASSIGN TO "CUST.dat"
           ORGANIZATION INDEXED RECORD KEY FS-CUSTID
           ACCESS MODE RANDOM.
data division.
FILE             SECTION.
FD CUST-MASTER.
01 CUST-REC.
    05 FS-CustId            PIC X(4)   COMP-5.
    05 FS-CustName          PIC X(30).
    05 FS-CustCompany       PIC X(30).
    05 FS-CustEmail         PIC X(30).
WORKING-STORAGE SECTION.
01 CustId       binary-long Property As "CustId".
01 CustName     String      Property As "CustName".
01 CustCompany  String      Property As "CustCompany".
01 CustEmail    String      Property As "CustEmail".

method-id new.
procedure division.
    OPEN INPUT CUST-MASTER.
    goback.
end method.

method-id. ReadCust.
procedure division.
    MOVE CustId TO FS-CustId.
    READ CUST-MASTER INVALID KEY
        CLOSE CUST-MASTER
        RAISE type java.lang.Exception::"new"(" レコードなし ")
```

```
            NOT INVALID KEY
                MOVE FS-CustName TO CustName
                MOVE FS-CustCompany TO CustCompany
                MOVE FS-CustEmail TO CustEmail
            END-READ.
            CLOSE CUST-MASTER
            goback.
        end method.
        end class ReadCust.
```

6) 以上で再利用する COBOL ロジックのクラス化は完了です。次にこのクラスを外部から利用してみます。上記で作成された ReadCust.class を CLASSPATH に載せて、以下の Java プログラムを実行してみます。

```
public class javamain {
      public static void main(String[] args) {
            ReadCust aRC = new ReadCust();
            aRC.setCustId(12345);
            aRC.ReadNext();
            System.out.println(aRC.getCustName());
      }
}
```

7) このプログラムは、もしデータファイル内にキー値 12345 のレコードが存在すれば、そのレコードに相当する顧客名を表示しますが、存在しない場合は以下のような例外を引き起こします。

```
Exception in thread "main" java.lang.Exception: レコードなし
at ReadCust.ReadNext(ReadCust.cbl:32)
at javamain.main(javamain.java:10)
```

COBOL で作成したクラスが正しく例外をスローしていることが確認できます。

6-6. JVM 向けリソースの活用

冒頭でも述べましたが Java が採用／検討される理由として標準化された技術や数多に及ぶ周辺技術が挙げられます。Java EE はエンタープライズアプリケーションの構築に必要な API 仕様を公式の標準化プロセスとして定めています。この仕様に従ってアプリケーションを開発することで Java EE 仕様を実装した様々な基盤上で運用が可能となります。また、Java EE はバージョンアップを重ねる度に機能拡充を遂げており、アプリケーションの機能の幅も拡げていくことが可能です。また、Java EE といった標準仕様だけでなく多くの団体やコミュニティーが Java や JVM を意識したフレームワーク等を提供しています。例えば、Apache Software Foundation は Hadoop や Spark といった JVM 向けの分散処理技術を提供しています。今や SNS 業界のみならずエンタープライズの分野でもこれらの技術を活用し解析処理等の高速化に取り組んでいます。

本章で紹介しています JVM COBOL は COBOL 言語を JVM 言語として扱うことを可能とする技術です。つまり Java EE や Hadoop を始めとした JVM 向けのフレームワーク上で COBOL 資産を運用することを可能とします。第4章で紹介した技術は Java EE が提供する API 仕様の1つである JCA を利用し JVM の外にある COBOL へ JVM からアクセスするための技術です。一方、JVM COBOL の場合は JVM 上で JVM クラスとして COBOL へアクセスが可能となるため、Java EE の環境を用意せずとも COBOL の利用が可能となります。そのため、例えば Hadoop を活用したバッチアプリケーションを開発するという場合でも Java EE の環境を使わずに COBOL 資産を活用した並列分散アプリケーションの構築が可能となります。この際 Hadoop のフレームワークを実装した Mapper や Reducer から COBOL 資産を直接 6 - 1 で紹介した要領で呼ぶことも可能ですが、前項で紹介したようなオブジェクト指向型 プログラムを挟みフレームワークを実装する Java 側の負担より軽減しつつも再利用する COBOL になるべく影響を与えない開発も可能です。本項ではこれを実現する Micro Focus が独自に拡張した JVM アプリケーション開発と親和性の高い方言・機能を更に掘り下げて紹介します。

1) ライブラリのインポート

Java 言語では import キーワードを利用することで、プログラム中で完全修飾せずにパッケージに属するクラスやインターフェイス等を利用できるようになります。例えば、Java にて

```
import java.io.*;
```

のように記述すると java.io パッケージのメンバーへプログラム中にて完全修飾せずにアクセスすることが可能になります。COBOL でもこの Java の import キーワードと同等の機能が用意されています。この場合 **ILUSING** コンパイラ指令を指定します。上の例であれば ILUSING (java.io) と指定します。もちろん、この指令を指定せずに完全修飾してパッケージ化されたメンバーへアクセスすることも可能です。

2) パッケージの宣言

コンパイルするクラスが所属するパッケージの指定は、Java 言語では package ディレクティブを使用します。
COBOL では PROGRAM-ID 段落及び CLASS - ID 段落の AS 指定がこれと同等の機能を提供します。Java の

```
package com.microfocus.demo;
public class ReadCust {
        :
```

は COBOL では

```
       PROGRAM-ID ReadCust as "com.microfocus.demo.ReadCust" public.
              :
```

と記述します。

手続き型で書かれたレガシー環境で利用されるプログラムソースに全く手を入れずプログラムをパッケージ化することも可能です。この場合は ILNAMESPACE コンパイラ指令を利用します。例えば、

```
       PROGRAM-ID ReadCust.
              :
```

のように JVM 連携を意識せずに書かれた PROGRAM-ID 段落を含むプログラムに対して ILNAMESPACE(com.microfocus.demo) を指定してコンパイルすると上の例と同じ名前でパッケージ化されたクラスが生成されます。

3) 変数の宣言

Java クラスのインスタンスを格納する変数の宣言は、COBOL では TYPE 句を使用します。Java の

```
java.net.Socket  mySocket = null;
```

に相当する宣言を COBOL では

```
       01  mySocket TYPE java.net.Socket  VALUE NULL.
```

と記述します。

従来の COBOL では変数は DATA DIVISION 内で宣言しますが、Java などではこのように明確にプログラムの処理と変数の宣言で記述エリアを分ける必要はありません。Micro Focus の JVM 拡張方言では DECLARE 文が実装され、PROCEDURE DIVISION 内でも Java と同じように局所変数宣言ができるようになりました。上の例を PROCEDURE DIVISION 内で宣言するには以下のように記述します。

```
       PROCEDURE DIVISION.
              :
       DECLARE  mySocket AS TYPE java.net.Socket = NULL.
```

4) メソッド、プロパティの参照

クラスに従属するメソッドやプロパティの参照は、Java 言語ではピリオド "." を使用して修飾します。COBOL ではこれに相当する分離符は "::" (二つ引き続くコロン) です。Java の

```
String myStr = "ABCDEFGH" ;
System.out.println(myStr.substring(4));
```

に相当する記述は COBOL では

```
       01  myStr  String VALUE  "ABCDEFGH".
              :
           DISPLAY myStr::substring(4).
```

と記述します。上記のコードは「EFGH」を表示します。

5) クラスのインスタンス化

Java では new キーワードを使いオブジェクトをインスタンス化します。

```
Aaa objA = new Aaa();
```

COBOL でも同様に new キーワードを使ってオブジェクトをインスタンスすることが可能です。

[SET 文を使う場合]

```
       WORKING-STORAGE SECTION.
       01 objA   TYPE Aaa.
       PROCEDURE DIVISION.
           SET objA TO new Aaa().
```

[DECLARE 文を使う場合]

```
           DECLARE objA = new Aaa().
```

6) 継承関係・実装関係の宣言

Java ではスーパークラスを継承する際には **extends** キーワードを、インターフェイスを実装する際は **implements** キーワードを指定します。COBOL においても INHERITS FROM 句、IMPLEMENTS 句を指定することでそれぞれと同等の宣言が可能です。例えば、Java では継承並びに実装宣言を下記のように宣言します。

```
public class JavaAAA extends MySuper implements MyInterface{
       :
```

これに相当する宣言を COBOL では下記のように記述可能です。

```
       CLASS-ID. CobAAA INHERITS FROM TYPE MySuper
                        PUBLIC
                        IMPLEMENTS TYPE MyInterface.
           :
       END CLASS.
```

7) アノテーションの利用

Java のアノテーションは Java のソースコードに付与するメタデータの一種です。コンパイル時に class ファイルに埋め込まれるアノテーションを基にした付加情報はツール、デバッガー、アプリケーション等から活用されます。

最も典型的な例としては、継承するクラス中のメソッドをオーバーライドしていることを示す @Override が挙げられます。例えば下記のように @Override アノテーションを記載せず更にメソッド名にスーパークラスと異なる名前を誤って記述していますと正しくオーバーライドされず誤記されたメソッドが子クラスのメンバーとしてコンパイルされてしまいます。

```
public class JavaAAA extends MySuper {
    @Override
    public void methodB(){
        :
    }
}
```

COBOL では @Override に限っては相当するキーワード OVERRIDE 句を用意しています。上記に相当する宣言は下記のように指定します。

```
       CLASS-ID. CobAAA INHERITS FROM TYPE MySuper PUBLIC.
       METHOD-ID. methodB PUBLIC OVERRIDE.
       PROCEDURE DIVISION.
           :
       END METHOD.
       END CLASS
```

OVERRIDE 句を指定しているにも関わらずスーパークラスにないメソッド名を記述しますと下記のようにコンパイル時に不整合を検出することが可能です。

```
C:¥work¥java_COBOL¥JVM>cobol CobAAA.cbl JVMGEN;
Micro Focus COBOL
Version 2.3.02161 Copyright (C) Micro Focus 1984-2016. All rights reserved.
* 956-S******************                                              **
**     メソッドまたはプロパティ 'methodBB' が継承クラスに見つからない - OVERRIDE
または REDEFINE 句を指定できない
*1712-S*************************                                      **
**     'type CobAAA' には 0 個のパラメーターのある可視インスタンスメソッド 'methodB'
 がない
* チェック終了 - エラーを発見しました

C:¥work¥java_COBOL¥JVM>
```

この他のアノテーションに対しては ATTRIBUTE 句を用いてアノテーションを指定します。例えば下記のようなアノテーションの利用を考えます。

```
@interface MyCustomAnnotation {
    int      count();
    String   wkstr();
}
```

これを Java で利用する場合は下記のように記述します。

```
public class JavaBBB {
    @MyCustomAnnotation(
        count=3,
        wkstr="ABC"
    )
    public void annoDemoJava(){
    ・・・
    }
}
```

同様に COBOL でも下記のように記述することで利用が可能です。

```
       CLASS-ID. CobBBB PUBLIC.
       METHOD-ID. annoDemoCOB PUBLIC
           ATTRIBUTE MyCustomAnnotation(NAME count=3, NAME wkstr="ABC").
       PROCEDURE DIVISION.
              :
           GOBACK.
       END METHOD.
       END CLASS.
```

本項では Micro Focus が独自に拡張した JVM アプリケーションを開発する際に有用な 同環境と親和性の高い記法の代表例を紹介しました。JVM 向けに用意されたフレームワーク等を享受するという方針を立てたからといって全て Java に書き換える必要はありません。この拡張記法を活用して既存の COBOL 資産を JVM 向けにテイラードする、もしくは Java と既存 COBOL 資産の間のインターフェイスを用意することで実績のある COBOL 資産を JVM 上で継続運用できます。

6-7. JVM COBOL のデータベースアクセス

JVM COBOL は埋め込み SQL 文によるデータベースアクセスをサポートしています。JVM COBOL では、埋め込み SQL 文を JDBC 経由のアクセスに変換する機能をサポートしています。以下にその例を見ます：

```
       WORKING-STORAGE SECTION.
       01  EMPNO  PIC 9(9) COMP-5.
       01  ENAME  PIC X(20).
       01  JOB    PIC X(20).
           EXEC SQL INCLUDE SQLCA END-EXEC.
       PROCEDURE DIVISION.
           EXEC SQL CONNECT USING
               "DRIVER=org.postgresql.Driver;" &
               "URL=jdbc:postgresql://localhost:5444/edb?" &
               "user=XXX&password=YYY"
           END-EXEC.
           EXEC SQL DECLARE CUR1 CURSOR FOR
               SELECT EMPNO, ENAME, JOB FROM EMP
           END-EXEC.
           EXEC SQL OPEN CUR1 END-EXEC.
           PERFORM UNTIL EXIT
               EXEC SQL FETCH CUR1 INTO :EMPNO, :ENAME, :JOB END-EXEC
               IF SQLCODE NOT = 0
                   EXIT PERFORM
               END-IF
               DISPLAY EMPNO "," ENAME "," JOB
           END-PERFORM.
           EXEC SQL CLOSE CUR1 END-EXEC.
           EXEC SQL DISCONNECT CURRENT END-EXEC.
           STOP RUN.
```

この例題は PostgreSQL の JDBC ドライバを使用して接続していますが、その他の JDBC ドライバでも接続文字列を対象の RDB 向けへ変更するだけで同様に利用することができます。

上記プログラムを見れば、接続方法の相違こそあれ、従来の埋め込み SQL プログラムと同様に、
1) 接続
2) カーソルの宣言
3) カーソルのオープン
4) 結果セットの FETCH
5) カーソルのクローズ
といった標準的なロジックでデータ取得のためのロジックを構築していることが判ります。

7. まとめ

企業情報システムにおける COBOL への過去の投資を最大限に活用しつつ、Java アプリケーションサーバー上でアプリケーションを開発してゆくための、具体的な手法を見てきました。

COBOL で開発されたアプリケーションは多くの企業の企業活動を長年強力に支えています。エンタープライズアプリケーション開発の目的として広く利用されるようになった Java EE ではありますが、全てのエンタープライズアプリケーションを COBOL から Java に書き換えることは甚大なコストやリスクを伴います。実際、Java EE 等の技術に注目はしていても、現在も COBOL アプリケーションを基幹システムとして利用する企業は非常に多く存在します。本書で紹介した技術を使って COBOL で記述されたビジネスロジックをコンポーネント化して Java 環境のシステム構築で活用することは、Java EE の技術を活用したプラットフォームで COBOL からの書き換えによるコストやリスクを回避できる非常な有効なソリューションと成り得ます。

プログラミング言語は道具に過ぎませんので、適材適所の使い分けが重要です。今後も COBOL は Java と共存して使われつづけて行くと考えます。