COBOL 利用技術のご紹介

COBOL コンソーシアム利用技術分科会
小林 純一(マイクロフォーカス株式会社 技術部)

第 6 回　COBOL か Java か？

1. Java は COBOL に代われるか

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。

COBOLもJavaもプログラミング言語に過ぎません。どちらも人間が計算機にさせたい仕事を記述するための言語です。従って、向き不向きや好き嫌いはあるものの、言語の機能に関して比較することはあまり意味がありません。

特に Javaはクラスライブラリの整備によっていくらでも守備範囲を広げてゆけるものです。また、COBOLもオブジェクト指向機能や、利用者定義関数の装備によって拡張性をもつようになります。従って、ある機能を取り上げて、「これがJavaでできるか、COBOLでできるか」と問うてもしかたのないことです。

COBOLもJavaもプログラミング言語に過ぎないとはいえ、それが登場する背景には言語論にとどまらない要求があり、その要求はそれぞれ異なっています。40年前にCOBOLが登場したときの要求は、一般的なデータ処理を高い生産性と保守性で記述でき、しかもプラットフォーム間のソース互換性を保証することでした。一方Javaは、最初はプラットフォームに依存しないオブジェクト指向言語として設計されましたが、その後のJava仕様開発の中では、インターネットで共有される分散コンピューティング環境を提供することを目的とするようになってきています。

COBOLは、その要求にこたえるためにデータ処理に必要な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、J2EE 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 J2EEのことを語っていることにまず注意すべきです。そして、J2EE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 J2EE 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。しかしその中でのシステム構築において使用できるプログラミング言語は Javaだけではありません。

本稿では、J2EE環境でのシステム構築において COBOLを活用することのメリットについて解説します。

2. 一般的な比較

Javaは、言語のトークンや分離符に C言語と共通するものが少なくないので、一見 C や C++ に類似した言語であるような印象を受けます。しかし、Javaには C より COBOL に近い面もあります。どちらもハードウェアアーキテクチャに依存しないプログラミングができる言語であり、プラットフォーム間の互換性が高いという点があげられます。

また、JavaもCOBOLもリンク時の静的解決を必要としない動的ローディングによるモジュール化を許しています。現在ほとんどの Cまたは C++言語の実装では、コンパイルされたコードはオペレーティングシステムのリンカーによって実行形式や共有ライブラリにリンクされることを必要としています。これに対して Javaはバイトコードを実行時に動的ローディングします。このようなプラットフォームに依存しない中間言語形式は、COBOLでは古くから実装されてきています。Javaが登場する10年以上も前から、Micro Focus COBOLや AcuCOBOL のようなオープン系COBOLには、DOS や UNIX を含む多くのプラットフォーム間でバイナリ互換性を持つ動的ロードの中間コードを生成し、それを各プラットフォーム上の仮想計算機が実行する方式をサポートしていました。もちろん、これを規格化して流通させたことは Javaがはじめて達成した功績です。

次に、COBOLとJavaの一般的な相違について検討します。

Javaは、その最大の利点の一つであるプラットフォーム間のバイナリ互換性を達成するために、中間言語の動的ローディングしか許していません。この実行形態は Servletのような動的なインスタンス化で実行される動作環境では大きな利点となります。一方、ほとんどのCOBOLの実装では、機械語にコンパイルされたコードの静的な実行をサポートしています。プログラムのロード時にメモリ領域が静的に確保され、その上での演算をひとつのプロセスに専念させることができます。このような実行形態は大量のバッチ処理を高速に実行するのに有利です。

技術的な見地から離れてみますと、COBOLの既存プログラム資産の量は決して無視できません。最新のテクノロジーを駆使したシステム構築であっても、COBOLで書かれた既存コードを再利用したいという要望は少なくありません。プログラム資産のみならず、スキル資産もCOBOLは膨大です。最近 Java人口は増大しておりますが、企業のIT部門の業務知識を持った人材に絞ってみれば、当分の間は COBOL人口が凌駕しているでしょう。

Javaの仕様が未だに開発途上であることも、長期に渡って使用されることが前提の企業の基幹システムで敬遠される理由の一つになっています。COBOLの規格は ANSI85 以来17年に及んで安定しており、今年規格化される予定の COBOL2002でも、ANSI85 からの高い

上位互換性があります。Javaも、1.3 以来旧版からの互換性は高くなりましたが、新規格の出る頻度が COBOLに比べて非常に高いことを心配するユーザもあります。

3. J2EE 環境での COBOL の利用

システム構築基盤として J2EE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。この章では、Java環境下でCOBOLを活用する手段について説明します。

3.1. バッチプログラムでの COBOL 利用

すべての情報システムが 24時間オンライン稼動を要請されているわけではありません。システムによっては、昼間がオンライン稼動で夜間がバッチ稼動という形態もありえます。このような場合 バッチ処理だけでも COBOLを使用することができます。ほとんどのデータベース管理システムは Javaからの JDBCアクセスと、COBOLからの埋め込みSQLアクセスの両方をサポートしています。

3.2. CORBA 配下での COBOL トランザクションの利用

J2EE が提供する RMI-IIOP を使用して、外部のCORBAコンポーネントと対話することができます。ほとんどのCORBA製品はCOBOLによるコンポーネント化をサポートしていますので、この方法でCOBOLロジックを利用できます。CORBA以外でも、ベンダー独自のトランザクションモニター製品で Javaからのアクセス部品を提供しているものも多くあります。

3.3. JNI ベースの言語連携機能の利用

Java Native Interface を使用して、非JavaアプリケーションをJava VM中にロードして呼び出すことができます。しかし、これを直接使用して COBOLプログラムを呼び出すことはお薦めできません。データ型変換などの問題によってプログラミングの負担が大きくなります。最近の COBOLの実装では、独自のJava連携機能をサポートするものが増えてきています。例として、Micro Focusが提供する mfcobol.runtime 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にパラメタを渡し、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。

4. COBOL の得意分野

プログラミング言語としての性格の相違から、COBOLにはCOBOLの得意な分野があります。この章では、Javaでもできないことは無いがどちらかと言えば COBOLの方が向いているとされている機能を紹介します。

4.1. 10 進演算

明示的な10進演算はCOBOLの持つユニークな言語機能です。プログラマはそれぞれのデータ項目に対して、小数点上・下の10進桁数を明示的に定義することができ、これらの間での四則演算およびべき乗の計算を自由に記述できます。例として以下のような計算を考えます：

```
01  A   PIC 9(5)V9(3)   VALUE 123.456.
01  B   PIC V9(5)       VALUE 0.00234.
01  C   PIC V9(5)       VALUE 0.00123.
01  D   PIC 9(5)V9(3).
    COMPUTE D = A * (1 + B) / (1 - C).
```

A、B、C という３つの値から D を計算しています。それぞれの値には独立に桁数が定義されています。

Javaには言語としてサポートしている10進データ型はありませんが、java.math パッケージの BigDecimalクラスを使用して10進演算が可能です。上の計算式を BigDecimalクラスで書きなおすと以下のようになります：

```
import java.math.*;
BigDecimal A = new BigDecimal(“123.456”);
BigDecimal B = new BigDecimal(“0.00234”);
BigDecimal C = new BigDecimal(“0.00123”);
BigDecimal D = (A.multiply((new BigDecimal(“1.00000”)).add(B)))
               .divide((new BigDecimal(“1.00000”)).subtract(C));
```

このように **Java**でも10進演算は可能ですが、かなり読みにくいものになります。プログラムが新規作製される場合にはそれほどの生産性の差は出ないかもしれません。しかし、これを将来に渡って保守して行く人のためには決して良いプログラミングではないでしょう。

言語としてサポートされていないことによる性能問題も無視できません。**Java**の **BigDecimal**クラスのようにライブラリとして提供されている場合、一般にはコンパイラによる最適化が効きません。**COBOL**のように、算術式を**COBOL**コンパイラが機械語に展開する場合、定数式の畳み込み、共通式の再利用と言った最適化技法が使用され、効率的なコードで計算が実行されます。

BigDecimal は変更不可能なオブジェクトですので、計算が必要となる場合にその都度オブジェクトを作製しなければなりません。**COBOL**で通常行われるバッチ処理では、同じ計算式が異なる値で繰り返し計算されますが、静的に割り付けられたメモリ領域を繰り返し使用して計算がなされます。**100**万件のレコードを処理するバッチを想定すれば、**Java**で必要となるオーバーヘッドが無視できなくなることもありえます。

4.2. PICTURE 編集

COBOLでは PICTURE文字列で数字編集を定義します。これによって、カンマ編集、通貨記号の浮動挿入、先頭のゼロ抑止など、通常要求されるすべての編集が可能です。

Javaでは同様の機能が DecimalFormatクラスで提供されます。PICTURE文字列に相当する「パターン」という Stringで編集形式を指示します。パターンで指定できる編集機能はむしろCOBOLより汎用的です。COBOLと同様に逆編集も可能です。国際化にも対応しており、通貨記号や編集形式を各国の独自形式に動的に対応させることができます。

例えば、

```
01  A   PIC 9(5)V9(3)   VALUE 1234.567.
01  B   PIC \\\,\\9.999.
    MOVE   A  TO   B.
```

は、DecimalFormatクラスで書き表すと以下のようになります：

```
BigDecimal A = new BigDecimal("01234.567");
String B;
DecimalFormat decform = new DecimalFormat("\\##,##0.###;-\\##,##0.###");
B = decform.format(A);
```

プログラミング方法で比較してみますと、ここでも言語としてサポートしているCOBOLの方が、Javaによるライブラリ使用より便利であることがわかります。COBOLでは受け側の Bという項目の属性として PICTURE編集を記述し、手続き部の MOVE文では特別な記述

が不要であるのに対し、DecimalFormatクラスでは、編集の発生する転記に対して編集パターンごとの DecimalFormatオブジェクトを作成しなければなりません。

また、現状では DecimalFormatクラスは、数値を Double または Int でしか格納できません。したがって BigDecimalオブジェクトをこれに適用しても、18桁までの精度しか得られませんし、暗黙的な丸めが発生してしまいます。したがって、厳密には COBOLのPICTUREの代替機能を提供していないことになります。

BigDecimalクラス自身が用意している format()メソッドを使用するとある程度の編集は行えますが、浮動挿入やカンマ編集まではできません。

5. データベース入出力

データベースアクセスは、企業情報システム構築においてもっとも普遍的なプログラミングです。この章では、COBOLとJavaの比較をデータベースアクセスの観点から見てみます。

COBOLによる関係データベース入出力の標準は埋め込みSQL文です。COBOLソース中に EXEC SQL と END-EXEC で囲まれた SQL文を埋め込み、ホスト変数と呼ばれる特殊なCOBOLデータ項目によって、SQL文とCOBOL文との間のデータ交換を行います。この方法は、SQL言語の国際標準によって規格化されており、ほとんどのCOBOL処理系・SQL処理系によってサポートされています。歴史も長く、関係データベースの黎明期からメインフレームのCOBOLプログラミングで利用されてきた実績があります。

Javaによるデータベースアクセスの標準APIは JDBCです。JDBC API は、Javaで記述されたクラスおよびインタフェースのセットで構成され、SQL 文を関係データベースシステムへ簡単に送信し、結果を受け取ることができます。

以下の例は、テーブルから全行を受け取るプログラミングを COBOL と Java で書き比べてみたものです。方法は何通りかありますが、ここではもっとも普通に使用されているプログラミング手段を選んでみました。まず、COBOLでは以下のようにカーソルを宣言して、順次 FETCH する方式です：

```
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  A   PIC X(30).
01  B   PIC 9(5)V9(3).
01  C   PIC X(5).
    EXEC SQL END DECLARE SECTION END-EXEC.

    EXEC SQL CONNECT <接続の詳細> END-EXEC.
    EXEC SQL DECLARE C1 CURSOR FOR
```

```
    SELECT A, B, C FROM TABLE1
  END-EXEC.
  EXEC SQL OPEN C1 END-EXEC.
  PERFORM UNTIL SQLCODE = +100
    EXEC SQL FETCH C1 INTO :A :B :C END-EXEC
  END-PERFORM.
```

一方 Javaで JDBC を使用すると、以下のように ResultSet オブジェクトを使用してカーソルの内容を取得するプログラミングになります：

```
Connection con = DriverManager.getConnection(<接続の詳細>);
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT A, B, C FROM TABLE1");
while (rs.next()) {
      String a = rs.getString("A");
      String b = rs.getNumber("B");
      String c = rs.getString("C");
}
```

COBOLでもJavaでも、SQL文を使用することには変わりありませんし、どちらも直感的にわかりやすいので、これは慣れや好き嫌いの問題であると思われます。筆者の個人的な感想としては、JDBC や ADO のように SQL文をプログラム中の文字列定数として記述するのには抵抗があります。SQLはプログラミング言語ですので埋め込みSQLのようにソースプログラム内にコードとして書くほうが自然に感じます。

6. おわりに

J2EE準拠のアプリケーションサーバーをインフラとしたシステム構築は、今後ますます普及して行くものとおもわれます。CORBAのような、プログラミング言語に依存しないコンポーネント化技術が成熟した現在、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には大きなメリットがあります。

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。