COBOL による Unicode データ処理

Unicode は、世界中の数多くのコンピュータ上で文字列を一貫した方法で符号化し、表現し、扱うための業界標準として、既に 30 年を超える歴史が積み重ねられてきています。グローバル化が進む IT の潮流において、同じアプリケーションシステムで日本国内のみならず海外の業務をも同時に処理することが求められるようになり、このようなケースで Unicode が活用されるようになっています。また、住民基本台帳のように住民票データが国内で統一化されるようになると、多様な人名漢字や地名漢字に対して統一した符号化が必要になります。ここでも Unicode をベースにした規格が採用されています。

COBOL は 60 年以上の歴史を持ちながら現在でも幅広い業種の IT で活用されているプログラミング言語です。特に公共系のシステムでは幅広く利用されているため、Unicode データを取り扱うことに対する要求は高まっています。COBOL は Unicode データを扱うために必要な機能を Java や各種 .NET 言語と同等に提供しています。

本書では、COBOL で書かれたアプリケーションシステムで Unicode データを扱うための背景と技法について解説します。

なお、本書で例示するプログラム例題は Micro Focus Visual COBOL 6.0 を使用して実行された結果を示しています。

White Paper

COBOL による Unicode データ処理

目次

1　COBOL 言語の文字コード対応

COBOL は 1959 年に最初の言語仕様が策定されて以来、国際的な規格化に基づいてその言語仕様が保護されてきました。世界中のあらゆる COBOL コンパイラはこの標準に準拠して作成されています。本書で話題とする文字コードについても言語仕様上の規定があります。本項ではまずこの点について解説します

1-1　COBOL 文字集合

COBOL 言語仕様で規定される構文の中で、プログラムを記述するために使用する文字が規定されており、これを「COBOL 文字集合」と呼びます。COBOL 文字集合は英字（大文字・小文字）、数字、空白、カンマ、ピリオドなどの一連の ASCII 文字に加えて、利用者語や文字定数の中で使用できる各国語文字が含まれています。

他のプログラミング言語と同様に、COBOL もソースファイル内に書かれた文はその文そのものに意味があり、その文がどのように符号化されているかは問いません。例えば "MOVE 入力 TO 出力" という COBOL の文はそのソースファイルが ASCII/Shift_JIS で符号化されていても UTF-8 で符号化されていても全く同じ意味を持ちます。唯一の例外は文字定数です。以下の二つのコード片を比較してみてください。

Java:	String str = “ABC”;
COBOL:	01 ITEM-1 PIC X(3) VALUE “ABC”.

この二つの文は良く似たことを記述しているように見えますが実際には異なります。Java は Unicode が誕生した後でできた言語であり、String 型は全世界の文字を Unicode で内部格納する実装が前提になっています。この意味で Java の文字定数 "ABC" に書かれた文字 A は抽象的なアルファベットの文字 A を意味しています。これに対して COBOL の PIC X(3) は計算機の文字集合のあらゆる文字が格納されうるものとされており、計算機の固有のあらゆるバイナリデータを含みます。従って COBOL の文字定数 "ABC" に書かれた文字 A は、Windows では ASCII/Shift_JIS、 Linux ではコンパイル時のロケールで符号化されたバイナリデータの A を示しています。この意味で COBOL の文字定数は Java の Byte[ ] 型に近く、コンパイル時の環境に依存しています。一方 Java の文字定数はコンパイル時の環境に依存していません。

COBOL の文字定数には X"F1F2" のような 16 進記述も許容されていることに注意してください。

このように、プログラムの実行時に取り扱われるデータ型としての文字は「計算機符号化文字集合」と呼ばれ、ソースコードを記述する「COBOL 文字集合」とは区別されています。

1-2　COBOL における多バイト文字と国際化機能

COBOL 言語が誕生した 1959 年当時は、1 文字 1 バイトで表現することがコンピュータ利用の常識でした。その後日本では漢字を印刷できるプリンタや表示できるディスプレイが登場したことにより、漢字 1 文字を 2 バイトで表現する符号系が登場しました。これに伴い多くの COBOL コンパイラが従来の 1 バイトの PIC X に加えて 2 バイトの PIC N をサポートするようになりましたが、COBOL 言語の国際標準による規格化は 2002 年規格までなされなかったため、各処理系によって言語仕様に若干の相違がある状態となっていました。

このような状況から、COBOL 2002 規格では、日本語のみならず世界での 2 バイト文字圏や、英語ではない 1 バイト文字圏での利用を想定した国際化機能が盛り込まれ、Unicode の存在も視野に入れた言語拡張が策定されています。

COBOL 2002 規格では、英数字型（PIC X / USAGE DISPLAY）と各国語型（PIC N / USAGE NATIONAL）の 2 種類の「計算機符号化文字集合」が用意されています。後者は規格では明示されていないものの 1 文字 2 バイトの符号系が想定されています。Micro Focus Visual COBOL では UCS2 エンコーディングですべての文字を表現する形式が取られています。これによって COBOL も Java などと同等にコンパイル時の環境に依存しない文字の表現を獲得したことになります。これについては本書の 0. で、より詳細に説明します。

1-3　言語外機能の活用

Micro Focus Visual COBOL は、.NET Framework や Java Virtual Machine 上で稼働するバイナリコードにコンパイルすることも可能になっています。そこでは C# や Java 言語と同等に COBOL 言語で .NET Framework や Java VM の提供するクラスライブラリを利用することができます。従って Unicode で内部実装された String オブジェクトを COBOL でも利用することができます。これは PIC X や PIC N のような固定長の文字列型ではなく オブジェクト型としての操作になりますので COBOL 言語の機能で適用できる範囲は狭まりますが、一方で String が装備しているすべてのメソッドを COBOL でも利用することが可能となります。これについては本書の 2-5 にてより詳しく説明します。

2　COBOL による UTF-8 データ処理

UTF-8 エンコーディングは、ASCII 文字を ASCII と同じ 1 バイトで表現し、それ以外の各種文字を 2 バイトから 3 バイトで表現する可変長の符号化方式です。英語のみで書かれた文書の場合 UTF-8 でも ASCII でも全く同じになることから、英語圏では幅広く採用されている形式です。本項では COBOL 言語で UTF-8 形式のデータを処理する方法について説明します。

2-1　バイト列としての処理

1-1 項で述べた通り、COBOL の PIC X(nn) 型データ項目は計算機の文字集合のあらゆる文字が格納されうるものとされており、あらゆるバイナリデータを含み得ます。従って、プログラムを実行するロケールや環境に無関係にここには UTF-8 の文字列を格納することができます。

以下のコードを見てください。

```
01 ITEM-1 PIC X(20) VALUE "ABC あいう 123".
```

もし、この文字定数が Shift_JIS で符号化されているならこのデータ項目の内容は以下のようになります。

41	42	43	82	A0	82	A2	82	A4	31	32	33	20	20	20	20	20	20	20	20
A	B	C	あ		い		う		1	2	3								

ひらがなの「あいう」は 2 バイトで表現されています。一方、UTF-8 で符号化されているならこのデータ項目の内容は以下のようになります。

41	42	43	E3	81	82	E3	81	84	E3	81	86	31	32	33	20	20	20	20	20
A	B	C	あ			い			う			1	2	3					

ひらがなの「あいう」は 3 バイトで表現されています。同じプログラムでも符号化の相違によってプログラムの実行結果が異なる COBOL の性格を表すものです。

このデータ項目に対して COBOL 言語の様々な文による操作が可能ですが、それらはすべてバイト単位の操作となることに注意が必要です。例えばこの項目に対する部分参照 ITEM-1(4:3) は "あいう" ではなく "あ" 1 文字の 3 バイトを参照します。INSPECT 文による文字数カウントもバイト数単位となります。前述の ITEM-1 データ項目のバイト数を表示する以下のプログラムは、Shift_JIS で符号化されている場合は 12 を表示し、UTF-8 で符号化されている場合は 15 を表示します。

```
WORKING-STORAGE SECTION.
01 X-STR PIC X(20) VALUE "ABCあいう123".
01 CNT PIC 99.
PROCEDURE DIVISION.
    INSPECT X-STR TALLYING CNT
      FOR CHARACTERS BEFORE INITIAL SPACE.
    DISPLAY "[ABCあいう123]のバイト数: " CNT.
```

2-2　UTF-8 型の利用

Micro Focus Visual COBOL では、IBM Enterprise COBOL 製品との互換性強化のため、UTF-8 型をサポートいたしました。この型を利用することで、UTF-8 で符号化された文字定数を設定することができます。

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 UVAR PIC U(10) VALUE U'あいう'.
PROCEDURE DIVISION.
```

上記例では、"あいう" という文字定数が設定されていますが、こちらはコンパイル時の環境に依存せず、常に UTF-8 で符号化されたバイナリデータ X'E38182', X'E38184', X'E38186' として保持されます。

2-3　ファイル入出力

COBOL は言語に固有のデータファイル入出力機能を持っています。これは .NET や Java でストリーム I/O のクラスライブラリを使用するのとは異なり、COBOL の文法で宣言したファイルに、OPEN 文、READ 文、WRITE 文などの文法を用いて入出力する方法です。

COBOL で扱うデータファイルには行順編成 (テキストファイル)、レコード順編成、相対編成、索引編成の 4 種類の形式があります。このうち行順編成を除く形式はバイナリ形式であり、レコードの内容をバイト列として保管しています。このためファイルの READ/WRITE ではデータはバイト単位でそのまま入出力されます。つまり、COBOL プログラム中で取り扱っているデータは、その符号化方式やデータ型に無関係にそのままデータファイル中に書き込まれ、またプログラム内に読み取られます。この時実行時のロケールは無視されますので、UTF-8 のフィールドをプログラムとデータファイルとの間でやり取りする場合は、そこに UTF-8 の文字列が格納されているということはアプリケーション側の約束事項として守らなければなりません。逆にこの約束を守っている限り、COBOL によるファイル入出力はどのように符号化された文字列であってもそのまま入出力することができます。

2-4　データベース入出力

COBOL によるデータベース入出力は埋め込み SQL 文を使用するのが一般的です。埋め込み SQL プログラミングでは、COBOL と SQL という二つの異なる言語を同一のソースプログラム内に同居させ、それらの間の橋渡しを行うためにホスト変数を使用します。

例えば SQL の INSERT 文で COBOL からデータベースのテーブルに行を追加する場合、追加すべき行の各カラムの値は COBOL プログラムがホスト変数に格納したものが使用されます。以下にプログラムの例を見ます。この例は Oracle Pro*COBOL による文法です。

```
WORKING-STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 EMP-NAME PIC X(10) VARYING.
01 EMP-NUMBER PIC S9(4) COMP VALUE ZERO.
     EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
     .........
     MOVE 9000 TO EMP-NUMBER.
     MOVE SPACES TO EMP-NAME-ARR.
     EXEC SQL SELECT ENAME INTO :EMP-NAME
                    FROM EMP WHERE EMPNO = :EMP-NUMBER
     END-EXEC.
     DISPLAY EMP-NAME-ARR.
     EXEC SQL COMMIT WORK RELEASE END-EXEC.
     STOP RUN.
```

Oracle のようなリレーショナルデータベースでは、文字型のカラムに格納されるデータは論理的な「文字」であり、それが物理的にどのようなコードで格納されているかは無意味です。上記のようなプログラムがホスト変数に受け取って格納する際にどのようなコードで格納されるかはクライアント側の設定によって変わります。Oracle の場合、クライアント側で受取る文字コードの設定は NLS_LANG 環境変数で指定します。たとえば Linux 上で以下のように環境変数を設定することによって UTF-8 ロケールで Oracle 上の文字型カラムを UTF-8 エンコードのホスト変数で取り扱うことができるようになります。

```
$ echo $LANG
ja_JP.UTF-8
$ echo $NLS_LANG
Japanese_Japan.AL32UTF8
$
```

上記の例題プログラムは EMP というテーブルの EMPNO = 9000 の行から ENAME というカラムの値をホスト変数 EMP-NAME に受け取っています。この値に以下のような日本語文字が格納されているとします。

```
$ sqlplus
.....
SQL> select ename from emp where empno = 9000;
ENAME
----------
山田
SQL> quit
$
```

このとき UTF-8 環境でこのプログラムを実行した結果は以下のようになります。

```
$ rtsora test1.int
山田
$ rtsora test1.int | od -tx1
0000000 e5 b1 b1 e7 94 b0 20 20 20 20 0a
0000013$
```

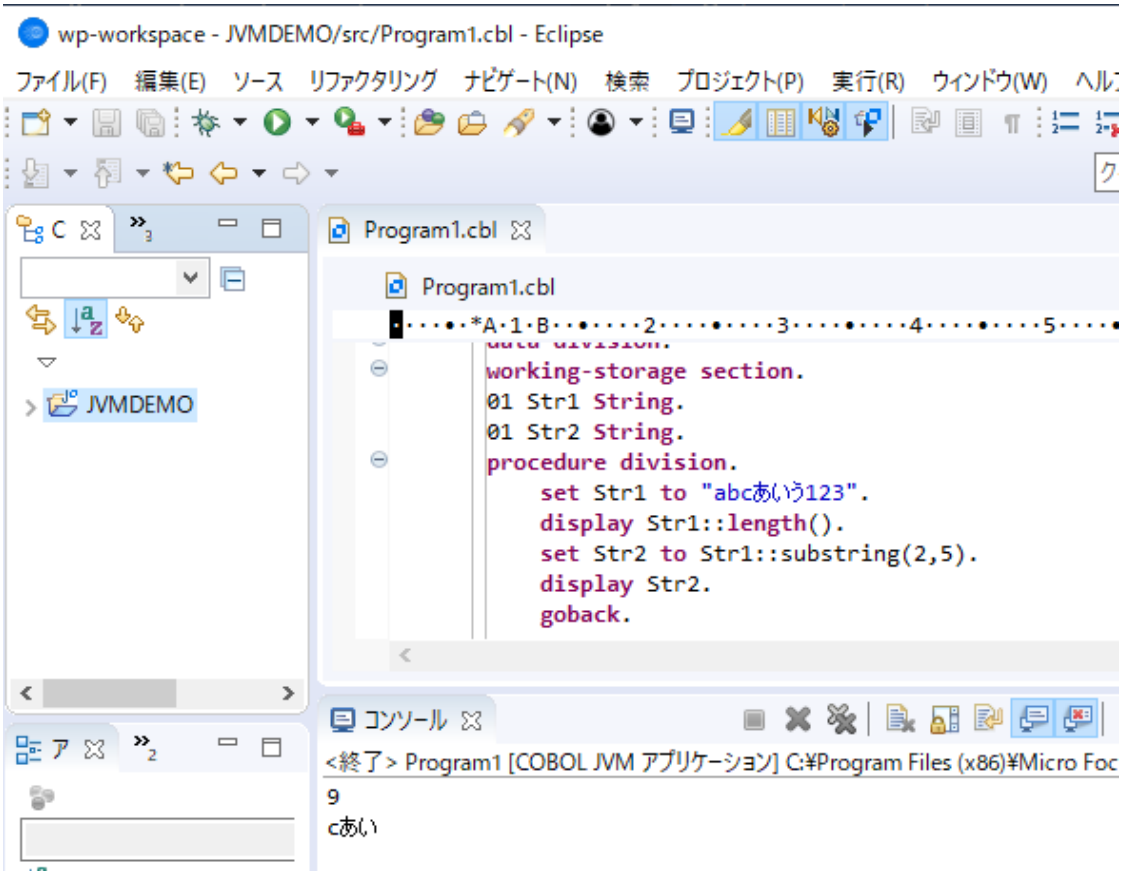
'山田' という文字が UTF-8 で X'E5B1B1E794B0' と 1 文字 3 バイトずつにエンコードされて返されていることがわかります。

2-5 String オブジェクトとしての処理

Micro Focus Visual COBOL は、COBOL プログラムを Java バイトコードにコンパイルする「JVM COBOL」をサポートしています。

ここでは、言語は COBOL であっても実行時の環境は Java VM であり、そこで操作するオブジェクトは完全に Java と同じになり、Java と同様にメソッドを駆使したプログラミングが可能となります。したがって String の操作も Java 言語と同様にバイト単位ではなく文字単位で処理することができます。

開発環境としては Java と同じく Eclipse を使用します。以下のスクリーンショットは、String で宣言した COBOL データ項目に、半角全角混在の文字列を代入し、これに対して文字数や部分文字列を取得する方法の例を示しています。



最初の SET 文は、String 型のデータ項目に文字列 "abc あいう 123" という文字列を代入しています。次の DISPLAY 文はその項目に格納された文字列の文字数を DISPLAY しています。コンソールに 9 と表示されており、全角半角混在でも文字数でカウントしていることがわかります。

次の SET 文は substring メソッドで部分文字列を切り取っています。ここでもコンソールに文字単位で "c あい" という文字列が切り取られていることがわかります。

2-6　UTF-8 および UTF-16 データを使った開発を補助する組み込み関数

Visual COBOL はバージョン 2.2 Update 2 より、IBM Enterprise COBOL 製品との互換性強化のため、UTF-8 データを COBOL プログラムでハンドリングする際に有用な組み込み関数が追加されました。一部の組み込み関数は UTF-16 データにも対応していましたが、バージョン 6.0 より 何れの組み込み関数も UTF-16 データに対応しています。これらをプログラム内に組み入れて活用することで、高度な UTF-8 および UTF-16 データハンドリング処理をシンプルなコード表現できるようになり、可読性の高いソース記述ができます。以下に各関数の特長を簡単にまとめて列挙します。詳細は製品ドキュメントで述べられています。

> ULENGTH 関数

引数に指定された項目中に格納されたデータの文字数を返します。

例えば、UTF-8 で符号化された以下のようにそれぞれ異なるデータ長で表現される文字から成る 10 バイトデータを引数に指定すると文字数「4」が返却されます。プログラム中で指定するデータは下記の文字で構成されています。

「丈」(X'F0A0808B')

「あ」(X'E38182')

「±」(X'C2B1')

「A」(X'41')

利用例：

```
                :
01 WK-CNT PIC 9(02) VALUE ZERO.
01 WK-VAR1 PIC X(10) VALUE X'F0A0808BE38182C2B141'.
PROCEDURE DIVISION.
    MOVE FUNCTION ULENGTH(WK-VAR1) TO WK-CNT.
                :
```

以下は、UTF-16 に符号化された文字データの文字数を返す例となります。

「丈」(NX'D840DC0B')

「あ」(NX'3042')

「±」(NX'00B1')

「A」(NX'0041')

利用例：

```
                :
01 WK-CNT PIC 9(02) VALUE ZERO.
01 WK-VAR1 PIC N(5) USAGE NATIONAL
                        VALUE NX'D840DC0B304200B10041'.
```

```
        PROCEDURE DIVISION.
            MOVE FUNCTION ULENGTH(WK-VAR1) TO WK-CNT.
                    :
```

> UPOS 関数

引数 1 に指定された項目中の n 番目の文字の開始位置を算出します。上の例で使用した UTF-8 データを下記のように全文字について同関数に渡した場合、「1」、「5」、「8」、「10」のようにそれぞれの文字の WK-MIX 内における開始バイト位置が返されます。

```
                    :
        01 WK-CNT PIC 9(02) VALUE ZERO.
        01 WK-IDX PIC 9(05) VALUE ZERO.
        01 WK-MIX PIC X(10) VALUE X'F0A0808BE38182C2B141'.
        PROCEDURE DIVISION.
            PERFORM VARYING WK-CNT FROM 1 BY 1 UNTIL WK-CNT > 4
                MOVE FUNCTION UPOS(WK-MIX WK-CNT) TO WK-IDX
                DISPLAY "RET-VAL: " WK-IDX
            END-PERFORM.
                    :
```

上の UTF-16 データの例では、結果は「1」、「5」、「7」、「9」となります。

```
                    :
        01 WK-CNT PIC 99 VALUE 0.
        01 WK-IDX PIC 99 VALUE 0.
        01 WK-MIX PIC N(5) USAGE NATIONAL
                                VALUE NX'D840DC0B304200B10041'.
        PROCEDURE DIVISION.
            PERFORM VARYING WK-CNT FROM 1 BY 1 UNTIL WK-CNT > 4
                MOVE FUNCTION UPOS(WK-MIX WK-CNT) TO WK-IDX
                DISPLAY WK-IDX
            END-PERFORM.
                    :
```

> USUBSTR 関数

引数 1 中から指定した位置にある文字を取り出します。引数 2 には取り出す文字の開始位置を、引数 3 には取り出す文字の長さを指定します。例えば以下のコードでは「あいうえお」の 2 文字目から 3 文字取り出しています。従いまして戻り値には「いうえ」が格納されます。

```
                    :
        01 WK-VAR PIC X(15) VALUE 'あいうえお'.
        01 WK-RET-VAL PIC X(15) VALUE SPACE.
```

```
        PROCEDURE DIVISION.
            MOVE FUNCTION USUBSTR(WK-VAR 2 3) TO WK-RET-VAL.
            DISPLAY WK-RET-VAL.
                    :
```

> USUPPLEMENTARY 関数

補助文字を使用する Unicode データが最初に出現される位置を算出します。UTF-8 の場合は 4 バイトで表現される文字の開始バイト位置を返します。例えば、以下のようなコードであれば、WK-UTF8 に「丈」(X'F0A0808B') の 4 バイト文字が 2 文字格納されています。このうち最初の「丈」は 6 バイト目から開始しているため、「6」が返されます。

```
                    :
01 WK-CNT8 PIC 9(02) VALUE ZERO.
01 WK-UTF8 PIC X(14) VALUE X'E38182C2B1F0A0808B41F0A0808B'.
PROCEDURE DIVISION.
    MOVE FUNCTION USUPPLEMENTARY(WK-UTF8) TO WK-CNT8.
                    :
```

UTF-16 に符号化された文字を格納する PIC N USAGE NATIONAL の変数が引数に指定された場合はサロゲートペア文字の最初の出現位置を返します。例えば以下のコードであれば、2 文字目にサロゲートペア文字「丈」(X'D840DC0B') 格納されているため、「2」が返されます。(以下はリトルエンディアンのプラットフォームでコーディングする場合の例となります。)

```
                    :
01 WK-CNT16 PIC 9(02) VALUE ZERO.
01 WK-UTF16 PIC N(05) USAGE NATIONAL.
01 WK-UTF16-ELEMENT REDEFINES WK-UTF16.
03 WK-ELEMENT1 PIC N(01) USAGE NATIONAL.
03 WK-ELEMENT2 PIC X(04).
03 WK-ELEMENT3 PIC N(02) USAGE NATIONAL.
PROCEDURE DIVISION.
    MOVE N'あ' TO WK-ELEMENT1.
    MOVE X'40D80BDC' TO WK-ELEMENT2.
    MOVE N'うえ' TO WK-ELEMENT3.
    MOVE FUNCTION USUPPLEMENTARY(WK-UTF16) TO WK-CNT16.
                    :
```

> UVALID 関数

引数に渡されたデータの妥当性を検査します。いずれの場合においても正しい範囲のデータが格納されている場合は、「0」が返されます。UTF-8 で規定されていない範囲のデータが格納されている場合、UTF-8 として成り立たないデータの開始位置が返されます。例えば、以下のコードでは WKUTF-8 を引数として渡した場合は「0」が返ってきますが、「丈」(X'D840DC0B') のうちの４バイト目を「X'0B'」から「X'FF'」に変えたデータを格納する WK-UTF8-W に対する検査では 不正な文字と判断されたデータが開始するバイト位置「6」が返されます。

UTF-16 で正しく符号化できないデータが渡された場合は、その文字位置を返します。

```
                :
       01 WK-RET8 PIC 9(02)     VALUE ZERO.
       01 WK-UTF8 PIC X(14)     VALUE X'E38182C2B1F0A0808B41'.
       01 WK-UTF8-W PIC X(14) VALUE X'E38182C2B1F0A080FF41'.
       PROCEDURE DIVISION.
            MOVE FUNCTION UVALID(WK-UTF8) TO WK-RET8.
            MOVE FUNCTION UVALID(WK-UTF8-W) TO WK-RET8.
                :
```

> UWIDTH 関数

引数 1 に指定されたデータに対して引数 2 で指定した文字位置が示す文字のバイト長を返します。例えば、以下のような上でも使用したデータの各文字を検査するとそれぞれの文字のバイト長「4」、「3」、「2」、「1」が返されます。

```
                :
       01 WK-CNT PIC 9(02) VALUE ZERO.
       01 WK-WIDTH PIC 9(05) VALUE ZERO.
       01 WK-MIX PIC X(10) VALUE X'F0A0808BE38182C2B141'.
       PROCEDURE DIVISION.
            PERFORM VARYING WK-CNT FROM 1 BY 1 UNTIL WK-CNT > 4
                 MOVE FUNCTION UWIDTH(WK-MIX WK-CNT) TO WK-WIDTH
                 DISPLAY "RET-VAL: " WK-WIDTH
            END-PERFORM.
                :
```

3　COBOL による UCS2 データ処理

UCS2 エンコーディングでは、すべての文字を 2 バイトの固定長で表現します。いわゆる BMP(基本多言語面) の文字のみが表現可能でありそれ以外の文字を扱うことができません。固定長であるために文字列処理が高速に実行できるメリットがあるため、OS やミドルウェアの内部表現形式として多用されています。本項では COBOL 言語がどのように UCS2 エンコーディングを使用しているかについて解説します。

3-1　NATIONAL データ型

COBOL の PIC N 型は、もともとメインフレームの EBCDIC 系コード集合で日本語文字を一文字 2 バイトで表現するものに対応する言語機能として仕様化されました。メインフレームでは 1 バイト文字と 2 バイト文字との区切りをシフトコードで区切る、いわゆるロッキングシフト方式でしたのでこのようにフィールド属性として明示的に 2 バイト文字であることを指定する必要がありました。しかし、オープン系の Shift_JIS や Unicode では文字自体が 2 バイト文字であることを示していますので本来このような指定は不要となっていました。たとえば PIC X(10) の項目に 1 バイト文字を 10 文字格納しても、2 バイト文字を 5 文字格納してもよいことになります。

ところが、このような方法では実際の文字の数とフィールドの占める長さが対応つかないことになり、COBOL 言語が持つ文字列操作機能で正しい処理結果が得られなくなります。COBOL には文字列の操作を行う固有の文法があり、部分文字列の取得や文字列内の検索・置換、接続・分解などを行うことができます。しかしこれらは一文字を表現する長さが固定であることが前提になっていました。

たとえば、COBOL の部分参照は固定長項目の部分文字列を取得する構文です。

```
01 ITEM-1 PIC X(10) VALUE "ABCDEFGHIJ".
```

に対して ITEM-1(3:4) は 10 バイトの項目 ITEM-1 の 3 バイト目から始まる 4 バイト分の部分文字列を取り出します。すなわちこの結果は "CDEF" になります。この処理がバイト単位に行われるものである点に注意してください。例えば、Shift_JIS のソースコードで

```
01 ITEM-1 PIC X(10) VALUE "ABC あいう J".
```

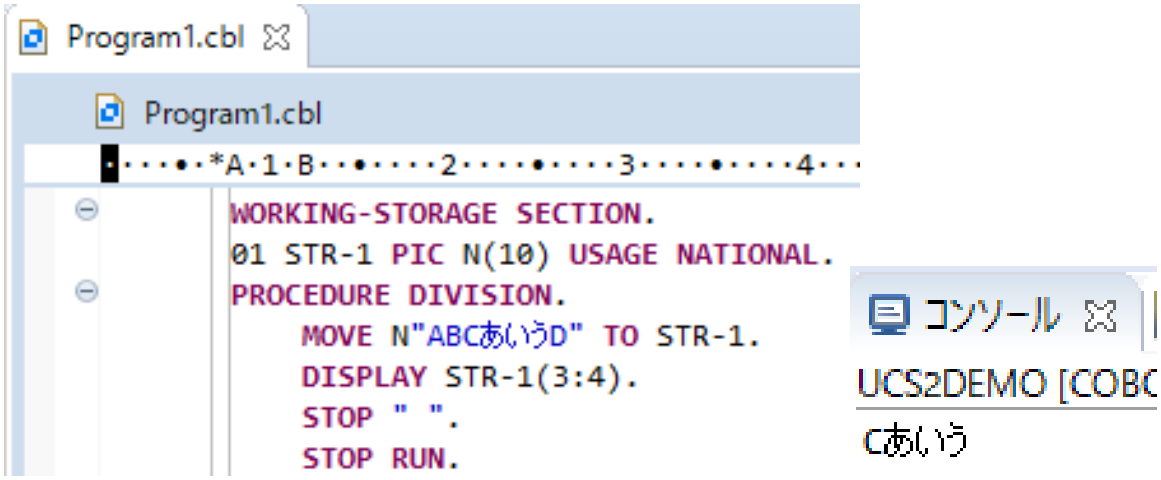
に対する ITEM-1(3:4) は "C あいう" にはなりません。C から始まる 4 バイト分のデータになりますので末尾は"い"の 1 バイト目になります。

このような半角全角混在の文字列に対しバイト単位ではなく文字単位で文字列処理を行う方法が UCS2 エンコーディングによる PIC N 項目の利用です。

COBOL 2002 規格で新たに導入された各国語型（PIC N / USAGE NATIONAL）のデータ型の実装として、Micro Focus Visual COBOL では UCS2 エンコーディングですべての文字を表現する形式が取られています。ここではアルファベットの "ABC" も日本語の "あいう" もすべて 1 文字 2 バイトで格納されこの文字

数が PICTURE 句の N の個数に一致します。

以下の例は Windows 版の Visual COBOL でこのようなプログラミングを行っている様子を示しています。文字定数 N"ABC あいう D" は、UCS2 型の文字定数であることを明示する記法です。この記法はソースコードがどんなエンコーディングで書かれていても無関係に同じ結果となります。Java の String 定数と同様にそこに書かれている抽象的な文字そのものが定数として割りつけられるものです。



3-2　ファイル入出力

2-3 で述べた通り、COBOL のレコード順編成、相対編成、索引編成はバイナリ形式であり、レコードの内容をバイト列として保管しているためファイルの READ/WRITE ではデータはバイト単位でそのまま入出力されます。PIC N USAGE NATIONAL で宣言され UCS2 のデータを格納しているフィールドについてもその通りです。

ここでも UCS2 のフィールドをプログラムとデータファイルとの間でやり取りする場合は、そこに UCS2 のデータが格納されているということはアプリケーション側の約束事項として守らなければなりません。

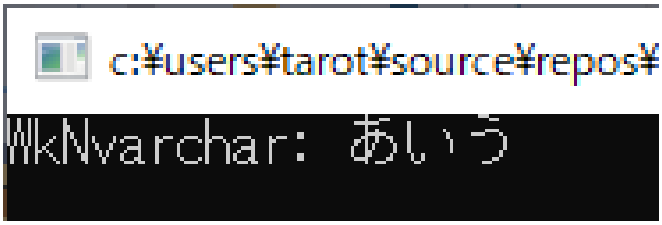
3-3　データベース入出力

2-4 で述べたとおり、COBOL から埋め込み SQL プログラミングでデータベースにアクセスする場合にはホスト変数を使用してデータベースのテーブルと COBOL プログラムとがデータを交換します。埋め込み SQL の方式によってはこのホスト変数に UCS2 エンコーディングの PIC N USAGE NATIONAL の型を使用することができます。

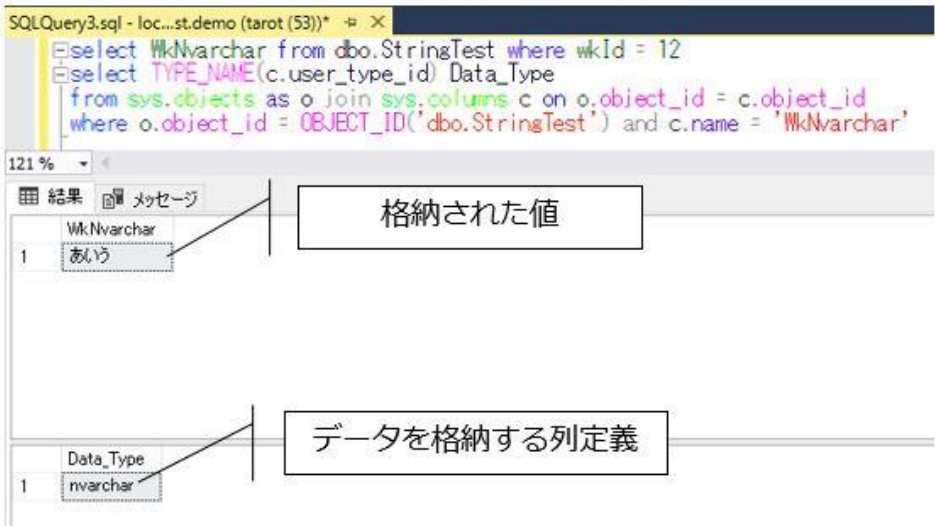
たとえば以下のコードは Visual COBOL が装備する OpenESQL を使用して ADO.NET データプロバイダ経由で SQL Server にアクセスする場合を示しています。このコードをお試しになる場合は、CONNECT 文に渡される変数や SELECT 文等を適宜環境に合わせて編集してください。

```
$SET SQL(DBMAN=ADO) UNICODE(NATIVE)
 DATA DIVISION.
 WORKING-STORAGE SECTION.
      EXEC SQL INCLUDE SQLCA END-EXEC.
 01                   USR PIC X(20) VALUE 'coboltest/password'.
 01                   SRV PIC X(10) VALUE 'SQLSrvrADO'.
      EXEC SQL BEGIN DECLARE SECTION END-EXEC.
*> ホスト変数
 01 StringTest-WkNvarchar PIC N(50) USAGE NATIONAL.
*> インディケータ変数
 01 StringTest-WkNvarchar-NULL PIC S9(04) COMP-5.
      EXEC SQL END DECLARE SECTION END-EXEC.
 PROCEDURE DIVISION.
*> データベースへ接続
      EXEC SQL CONNECT :USR using :SRV END-EXEC.
*> 'あいう' が格納されたレコードを取得
      EXEC SQL
           SELECT WkNvarchar
           INTO :StringTest-WkNvarchar:StringTest-WkNvarchar-NULL
           FROM StringTest WHERE WkID = 12
      END-EXEC.
*> 取得したデータを表示
      DISPLAY "WkNvarchar: "
              FUNCTION DISPLAY-OF(StringTest-WkNvarchar)
*> データベースから切断
      EXEC SQL DISCONNECT CURRENT END-EXEC.
      EXIT PROGRAM.
      STOP RUN.
```

本例で取得したデータを格納するテーブル列は Unicode 形式で文字列を格納する nvarchar で定義されています。これを PIC N USAGE NATIONAL で定義されたホスト変数で受け取っています。



実行結果



例中で使用したデータ

3-4　ASCII や UTF-8 との相互変換

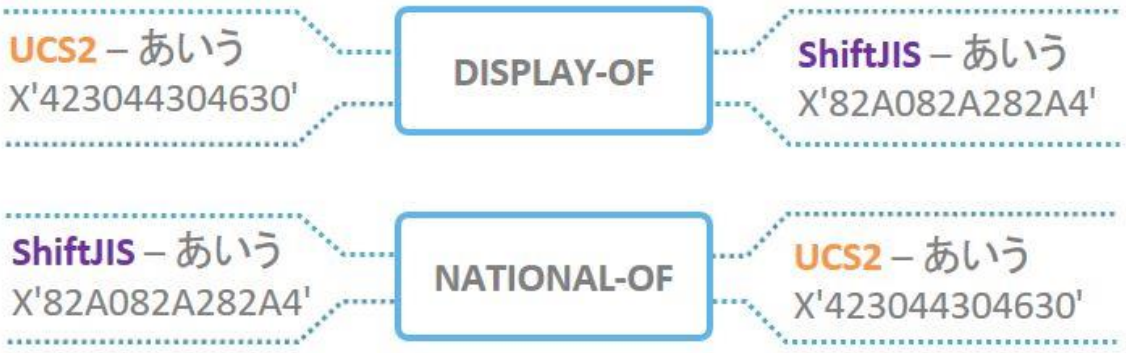
前項までで紹介していますように文字定数はソースファイルの符号化に依存します。一方、各国語型（PIC N / USAGE NATIONAL）で表現した場合は UCS2 として扱います。このような差分を相互変換して吸収する機能を提供する組み込み関数が Visual COBOL には用意されています。

- DISPLAY-OF　・・・　各国語文字等を引数で受け取り、対応する英数字を返す
- NATIONAL-OF　・・・　英数字等を引数で受け取り、対応する英数字を各国語型で返す

次に例示するプログラムソースを Shift_JIS のエンコーディングで保存した場合を考えます。これを Shift_JIS 配下で実行すると、まずは DISPLAY-OF 関数にて UCS2 で符号化された文字「あいう」を実行時ロケールの Shift_JIS で符号化したデータに変換します。続く NATIONAL-OF 関数では、引数に渡された ShiftJ_IS データ「あいう」を UCS2 で符号化された各国語文字型のデータに変換します。

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 UCS2-DATA PIC N(3) USAGE NATIONAL VALUE N'あいう'.
01 SJIS-DATA PIC X(6) VALUE 'あいう'.
01 WK-NVAR PIC N(3) USAGE NATIONAL.
01 WK-XVAR PIC X(6).
PROCEDURE DIVISION.
    MOVE FUNCTION DISPLAY-OF(UCS2-DATA) TO WK-XVAR.
    MOVE FUNCTION NATIONAL-OF(SJIS-DATA) TO WK-NVAR.
    GOBACK.
```

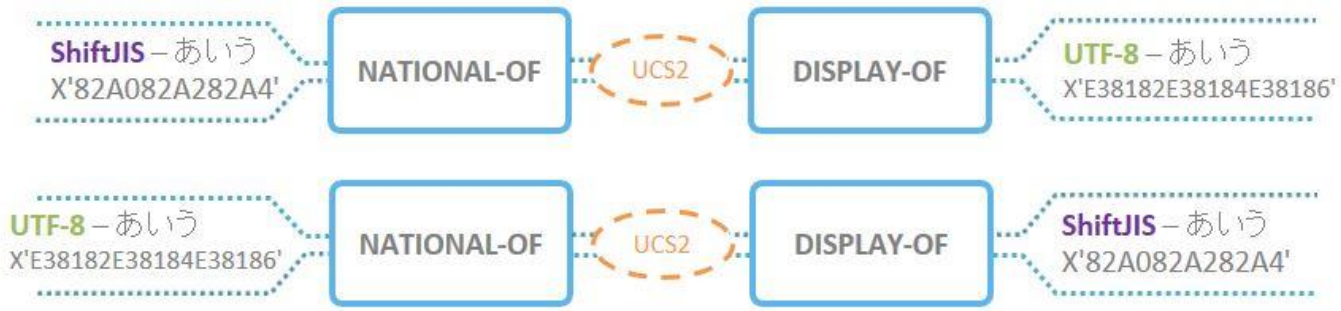
つまり、これらの関数の実行により下図に示すような変換処理が得られたことがわかります。



DISPLAY-OF 及び NATIONAL-OF はともに ISO 2002 の国際規格で規定された関数となります。Micro Focus Visual COBOL はこれを更に拡張し、指定したコードページに対応した文字コードへの変換機能を提供します。(コードページ 1208 (UTF-8) 以外を指定する場合は、CCSID 変換テーブルを参照する必要があります。手順につきましてはマニュアルを参照してください。) これを利用すれば、実行時ロケールに応じたコードページで符号化されたデータと UTF-8 で符号化されたデータを相互変換できるようになります。下記のコードを Shift_JIS ロケール配下でコンパイルし実行すると最初の MOVE 文で Shift_SJIS データを UTF-8 のデータに変換し、続く MOVE 文ではその逆の変換を処理しています。

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 SJIS-DATA PIC X(6) VALUE 'あいう'.
01 WK-XVAR PIC X(6).
01 WK-UVAR PIC X(9).
PROCEDURE DIVISION.
    MOVE FUNCTION
    DISPLAY-OF(FUNCTION NATIONAL-OF(SJIS-DATA) 1208)
    TO WK-UVAR.
    MOVE FUNCTION
    DISPLAY-OF(FUNCTION NATIONAL-OF(WK-UVAR 1208))
    TO WK-XVAR.
    GOBACK.
```

具体的には下図のように最初の MOVE 文では、まず Shift_JIS のデータを UCS2 に変換し、その変換結果を使って UTF-8 に変換します。 続く MOVE 文では UTF-8 のデータを UCS2 に変換し、その変換結果を使って最初の例と同様に Shift_JIS へ変換します。 つまり、いずれの変換も UCS2 への変換を経て目的のコードページへ変換します。



4　COBOL による UTF-16 データ処理

UCS2 のような 16 ビット符号化方式では表現できる文字数が制限され、全世界のすべての文字を共通の符号化によって表現すると言う Unicode の目的を達成するには不足することが早い時期で判明しました。このためにサロゲートペア（代用対）による表現が開発され UTF-16 として標準化されました。ここでは連続する 2 つの 16 ビット文字によって BMP に収録できない文字を表現します。これも文字の種類によって 2 バイトまたは 4 バイトとなる可変長の符号化であるための問題点はありますが、BMP の範囲内であれば 16 ビット文字のみで作成された固定長文字によるテキストと同じになるというメリットがあります。本項では、このような可変長符号化されたデータを COBOL 言語で取り扱う方式について解説します。

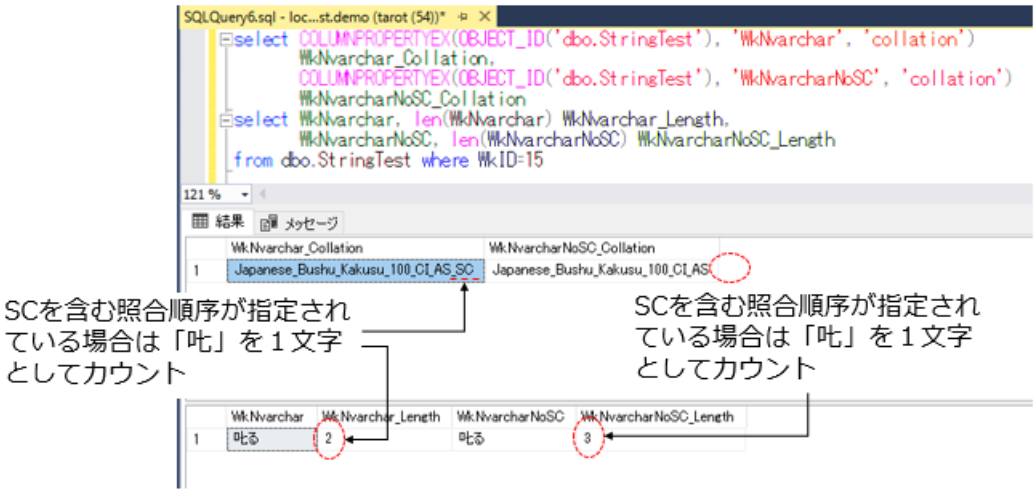
4-1　一般的なサロゲートペア文字操作

Windows では Vista 以降 OS 内部の文字コードとしてサロゲートペア文字がサポートされました。これによって Windows アプリケーションでこれらの文字を正しく表示できるようになっています。SQL Server でもサロゲート文字に対応した機能が装備されており、特に SQL Server 2012 より導入された SC(Supplementary Character) 照合順序を使用することでサロゲート文字を 1 つのコードポイントとしてカウントすることができます。

例えば C# で、

```
// 文字列「叱る」
string moji = "¥uD842¥uDF9F る";
```

と宣言した文字列は、サロゲートペア文字を含む 2 文字の文字列であり、これを SC を含んだ照合順序を持つ nvarchar 型のカラムに格納すると、SQL Server は 2 文字として認識します。下図は SC を含んだ照合順序を持つ nvarchar カラムに格納された「叱る」をカウントした場合、2 文字とカウントされる様子を示しています。



ところが、この文字列を .NET Framework の String オブジェクトとして受け取ると、その length プロパティは3となり、3文字と認識されます。従って、C# などの .NET プログラミング言語でこのような文字を扱う場合には、これまで通りの String 操作では正しいハンドリングができなくなります。このため StringInfo クラスや Char.IsSurrogate メソッドのようなサロゲートペア文字の処理に対応したプログラミングが追加で必要となります。以下のサンプルは C# でサロゲートペア文字を操作する例となります。

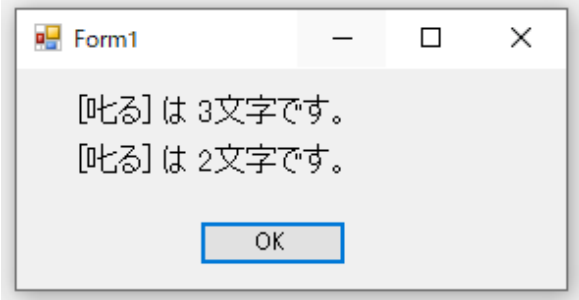
```
using System.Globalization;

namespace CSDemo
{
    2 個の参照
    public partial class Form1 : Form
    {
        0 個の参照
        public Form1()
        {
            InitializeComponent();
        }

        1 個の参照
        private void button1_Click(object sender, EventArgs e)
        {
            // 文字列「叱る」
            string moji = "¥uD842¥uDF9F る";
            this.label1.Text = "[" + moji + "] は "
                + moji.Length + "文字です。";
            StringInfo si = new StringInfo(moji);
            this.label2.Text = "[" + moji + "] は "
                + si.LengthInTextElements + "文字です。";
        }
    }
}
```

3文字としてカウントされる。

2文字としてカウントされる。



実行結果

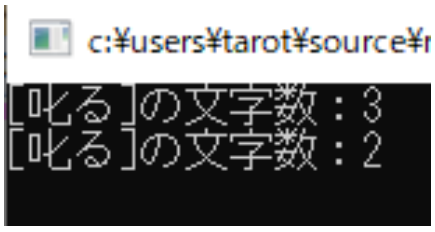
4-2　COBOL によるサロゲートペア文字操作

3 章で述べた通り、COBOL で PIC N NATIONAL と宣言された項目は、１文字２バイト固定で UCS2 符号化の Unicode 文字を格納します。これは BMP の範囲内で UTF-16 符号化と等価です。従って、この宣言でサロゲートペア文字以外の UTF-16 文字列を操作することができます。

もしここにサロゲートペア文字が含まれていると、それは C# のような言語で String 操作を行うのと同様に COBOL でも PIC N ２文字分の領域を必要とします。即ち COBOL でも C# などと同様に１文字のサロゲートペア文字は２文字として扱われてしまいます。このため COBOL 言語の部分参照や STRING 文などによる文字数カウントでは正しい文字数による操作ができないことになります。

この問題を解決する方法は 4-1 項で述べた C# による解決方法と変わりありません。以下の例題は COBOL でも C# と同様にサロゲートペア文字に対応するコーディングを行っているものです。

```
WORKING-STORAGE SECTION.
01 N-Str PIC N(10) USAGE NATIONAL.
01 S-Str String.
01 CNT PIC 99.
01 SI TYPE System.Globalization.StringInfo.
PROCEDURE DIVISION.
    MOVE NX"D842DF9F" TO N-Str.
    MOVE "る" TO N-Str(3:1).
    INSPECT N-Str TALLYING CNT
      FOR CHARACTERS BEFORE INITIAL SPACE.
    SET S-Str TO N-Str(1:CNT).
    DISPLAY "[叱る]の文字数：" S-Str::Length.
    SET SI TO TYPE System.Globalization.StringInfo::New(S-Str).
    DISPLAY "[叱る]の文字数：" SI::LengthInTextElements.
    STOP " ".
```



実行結果

なお、SQL Server の nvarchar 型カラムにサロゲートペア文字が含まれている場合も、PIC N NATIONAL で宣言された COBOL のホスト変数には２文字分（４バイト）で格納されるため nvarchar の文字数と COBOL 側の文字数が構成によっては合わなくなることにも注意が必要です。（前述のように SQL Server 2012 では使い方によってはサロゲートペア文字を１文字として扱います。）これもまた COBOL に固有の問題ではなく C# のプログラマと同様に注意すべき問題となっています。

5　Unicode を使用したシステム開発設計

ここまでに述べてきた事柄に留意して COBOL 言語で開発するシステムにおいて Unicode を使用する場合の留意点を以下に整理します。

5-1　バッチアプリケーション

多くのバッチアプリケーションは概括的に「入力」 → 「データの加工処理」 → 「出力」といったフローから構成されています。この「入力」、「出力」先の選択肢としてはデータファイルやデータベースが挙げられます。それぞれとデータ授受する手法についてはこれまでに紹介してきましたが、ここでは改めて COBOL プログラムでこれらと Unicode データを連携させる際の留意点を整理します。

システム内部で使用するデータファイルでは、2-3 で述べたようにレコード順編成、相対編成、索引編成といったファイルはレコード内容をバイト列として保管します。COBOL プログラムから READ/WRITE されたレコードは符号化やデータ型に関わらずそのままバイト単位で処理されます。そのため、正しいレコード設計をすることで Unicode データであっても Shift_JIS のような単一言語文字符号化形式で符号化されたデータと同じように入出力できます。どのフィールドにどの形式で Unicode データを格納するのかを設計上明確にし、格納されるデータの長さを考慮して十分なバイト数をレコードに確保します。

リレーショナルデータベースの種類によってはデータベースで Unicode に符号化されたデータを扱う手法が異なる場合があります。例えば、Oracle データベースでは、nchar や nvarchar2 のような Unicode に対応したデータ型を使用しなくとも Unicode をデータベースキャラクタセットに指定することで char や varchar2 のようなデータベースキャラクタセットに応じた文字列を格納するデータ型でも Unicode を扱うことができます。データベースと Unicode データを授受する場合は、このようなデータベース設計も考慮する必要があります。そして COBOL プログラム側ではデータベースと授受するためのホスト変数のタイプを上で紹介しましたように正しく設計します。

バッチアプリケーションがデータ加工処理するためにデータファイルやデータベースと各国語データを交換する場合、UTF-8 符号化方式が広く使われています。処理の設計によっては上記のとおり COBOL 言語で扱う方式に合わせて適宜フィールドを型変換する必要となるケースが想定されます。

COBOL 言語のプリンタ順編成ファイルには Unicode データを直接出力することはできないため、バッチアプリケーションによる帳票出力設計には注意が必要です。実現方法の選択肢としては、Unicode データに対応した帳票サーバ製品に UTF-8 に符号化されたデータを引き渡す方法が挙げられます。

5-2　オンラインアプリケーション

COBOL を使ったオンラインアプリケーションを考えた場合、主に 2 つの構成に大別できます。 1 点目は、Java Application Server や .NET ベースの Web アプリケーション、リッチクライアントアプリケーションをフロントエンドとして、COBOL のビジネスロジックを利用するようモダナイズされたパターンです。こちらの場合は、上で述べたような COBOL 言語を扱う際の一般的な留意点を意識し設計してさえいれば、特段注意は必要ありません。一方、SCREEN SECTION 構文を使い COBOL 言語でフロントエンドを記述する旧来のパターンでは Unicode の入出力が不可能なため、注意が必要です。

5-3　J2EE 連携アプリケーション

Visual COBOL は、COBOL プログラムを Enterprise Server と呼ばれる Application Server に配備し、J2EE アプリケーションから JCA 経由で利用する仕組みを提供しています。J2EE アプリケーションと COBOL プログラムの間でデータ授受するには、データ型が異なるためこれをマッピングする仕組みが必要となります。Visual COBOL が提供する IMTK(Interface Mapping ToolKit) というユーテリティを使用しますと COBOL サーバ側のロケールに合わせるかたちでマッピングを生成します。
つまり、

String <=> PIC X(nn)

のようなマッピングであれば、PIC X(nn) に入る文字は COBOL サーバ側のロケールに対応した方式に符号化されたバイトとなります。
従いまして、このパターンであっても上で述べたような一般的な留意点さえ勘案して設計していれば特段注意は必要ありません。

5-4　マネージ COBOL アプリケーション

Visual COBOL は、COBOL プログラムから JVM バイトコード、もしくは MSIL コードを直接生成する機能を提供しています。このパターンでは、COBOL プログラムは Java、もしくは .NET 言語として扱われるため、Unicode の使用においても 通常の Java や .NET 言語で開発する際に注意すべき点を踏まえた設計をしていれば特段注意は必要ありません。

6　まとめ

ここまで本書では COBOL 言語で Unicode データを取り扱う方法を見てきました。Unicode ハンドリングに関してプログラミング言語の特殊性はなく、Java や .NET 言語でできることが COBOL でもそのままできることが理解いただけたことと存じます。

記載の会社名、製品名は各社の商標または登録商標です。
本ホワイトペーパーは 2020 年 12 月に作成したものです。
MFWP05-2012-00MFC | © 2020 Micro Focus. All rights