COBOL アプリケーションのデータアクセス

企業情報システムはデータとロジックとプレゼンテーションとで成り立っています。事務処理向けのプログラミング言語であるCOBOLは、その50年以上の歴史の中で企業データへのアクセスが最も重要な言語機能の一つであり続けていました。本ホワイトペーパーでは、Micro Focus™ Visual COBOLが提供する各種のデータアクセス機能について、接続方法、プログラミング方法、関連するテクノロジーについて包括的に解説し、新規開発やマイグレーションにおける手法選択の基準を提供することを目的とします。

内容

1 COBOL からのデータアクセス概論

情報システムの設計は、要求分析の結果としての概念データモデリングが基本となります。企業活動を抽象化した結果、ビジネスプロセスは概念データモデル上のイベントやオペレーションとして再定義することができ、これをもとに企業活動を加速する情報システムを開発することができるようになります。
このため、コンピュータによる企業情報処理自動化の 60 年の歴史の中で、この概念データモデルをいかに効果的に効率よく実装するかは常に重要な課題であり続けてきました。
一方この間、企業情報システムの非常に大きな部分は COBOL 言語を使用して開発されています。このため COBOL 言語でモデル化されたデータを操作することがシステム開発の基本的テクニックとして位置づけられてきました。本書は Visual COBOL 製品を使ったデータアクセスの様々な手法を包括的に紹介することを目的とします。本書では「データ」として外部のファイルやデータベースに格納される永続的なものを扱うこととし、COBOL プログラムが処理上一時的に扱うデータ構造は対象外とします。

コンピュータによるデータ処理の最も初期の形態は、構造化されたデータレコードを先頭から順に読んで計算処理をすることです。売上計算を例に取ってみましょう。個々の受注伝票に関する情報は、得意先コード、受注日付、伝票番号、受注明細（品名、数量、単価）といったものから構成されます。これらを構造化レコードとして例えば以下のような物理設計をします：

ヘッダー部						データ部(1)					データ部(8)			
得意先コード	日付			伝票NO	区分	品名コード	数量	単価	金額		品名コード	数量	単価	金額
	年	月	日											
9(6)	99	99	99	9(3)	9	9(6)	S9(7)	9(6)V99	S9(9)		9(6)	S9(7)	S9(6)V99	S9(9)

このようなレコードが全従業員の数だけ順に並んだものが COBOL の順編成ファイルです。COBOL 言語には順編成ファイルを先頭から順に読み書きしたり、特定のキーでソートしたりする構文が用意されています。これについての詳細は本書の 2 章で扱います。
次に、上記のようにコード化された「得意先」「品名」といったエンティティにはそれぞれ詳細情報があります。例えば得意先は 6 桁のコードで表されていますがその実体には、会社名、住所、電話番号など様々な情報があり、それらは処理上必要に応じて得意先コードで突き合わせて参照できなければなりません。このようにファイル内のキー値で指定された固有のレコードにランダムにアクセスできる機能を提供するのが COBOL の索引編成ファイルです。これについての詳細は本書の 3 章で扱います。
COBOL にはこのほかにファイル内のレコード順番号でランダムアクセスすることができる相対編成ファイルもあります。本書ではこれについては触れません。
これら順・索引・相対編成のデータファイルアクセスは COBOL 言語の国際規格で定められた構文によって実装されており、プラットフォーム間でプログラムのソースコード互換性があります。しかしその物理的な実装はコンパイラによって異なっており、データファイル自体の互換性は一般にはありません。Visual COBOL のファイルハンドラは Visual COBOL に固有の実装でこれらの COBOL ファイル編成を処理していますが、その API は公開されています。コンパイラは、COBOL 共通の文法で書かれたソースファイルをコンパイルしてこの公開された API の呼び出しに展開しています。このためこの API を COBOL や C 言語で直接呼び出すことによって Visual COBOL のデータファイルに対して自由度の高いアクセスを行うことができます。さらに Visual COBOL のコンパイラ指令の指定によって、呼び出されるファイルハンドラを Visual COBOL 標準提供のものではなく利用者作成のものに置き換えることも可能です。この機能を使用して COBOL ファイル処理を自由にカスタマイズすることができます。この話題については本書の 4 章で説明します。

概念データモデルが複雑になってゆき、ビジネスエンティティの数が増加してゆくと、その数だけの索引編成ファイルが必要となり、一つのプログラムから数十、数百のファイルを同時に開いて突き合わせ処理を行う必要が出てきます。これは開発生産性を低下させ、実行性能にも悪影響を及ぼします。このような問題を解決するために階層型データベースやネットワーク型データベースが考案され、一つのデータベース内に含まれる多数の構造の異なるレコードを相互の関係で有機的に管理し、アプリケーションから統一的にアクセスできるようになりました。その代表的なものが IBM メインフレームの IMS データベースです。COBOL から IMS データベースへのアクセスは IMS が提供する API の呼び出し、または EXEC DLI コマンドと呼ばれる組み込み構文を使用します。これらはいずれも IMS 固有のプログラミングであり、規格化はされていません。IMS のような階層型データベースやネットワーク型データベースは各社メインフレームでそれぞれ独自に実装されている他、ADABAS や IDMS のような ISV 製品としても普及しました。しかし、UNIX や Windows ではほとんど実装されず、アプリケーションのソースコード互換性もないことから次第にレガシー化しつつあるのが現状です。
Visual COBOL の上位製品である Micro Focus Enterprise Developer は、UNIX, Windows 上で IMS と互換のデータベース管理システムを提供しており、COBOL、PL/I プログラムをそのままでオープンサーバー上にリホストすることができます。この話題については本書の 6.1 で触れます。

70 年代になるとビジネス分野でのコンピュータの利用はバッチ処理だけでなくリアルタイムなオンライン処理にも進出しました。オンライン業務に対して複雑な概念データモデルを実装するためには、データ無矛盾性、トランザクション一貫性、障害回復などの様々な新たな

要求が発生します。これらの要求にこたえるためにリレーショナルデータベースが登場しました。複雑なデータモデルを、エンティティ間の関係やデーター貫性制約まで含めて簡潔に記述でき、その上の操作をクエリーとしてエレガントに記述できる SQL 言語は、当初メインフレームで実装されましたが、その後 UNIX、Windows でも幅広く実装され、普及しました。現在でも Java や .NET アプリケーションの開発で幅広く利用されています。本書の第 5 章では COBOL 言語からリレーショナルデータベースにアクセスする様々な方法について解説します。

2 順ファイル

本章では COBOL による順編成ファイルアクセスに関連する話題を解説します。

2-1 トランザクションファイルとしての順ファイル

企業情報システムを構成するアプリケーションは、その運用形態からバッチとオンラインに大別して語られます。このうちバッチ処理はコンピュータによる事務処理の中でもっとも古くからおこなわれていたものです。
オンライン処理が 1 件のトランザクションをリアルタイムに処理するのに対し、バッチ処理は蓄積された大量件数のデータを一括して処理するものであり、主に夜間や週末・月末などに定期的に実行されます。典型的な例として以下のようなものがあります:

1) 昼の営業時間内に各営業所から挙げられた受注データをデータセンターで取りまとめて夜間に一括して出荷・請求処理を行う
2) 毎月末に、締日までの操業稼働データを整理して全従業員の給与明細発行と銀行振り込みを行う

バッチ処理では大量件数データの処理を行うといっても個々のデータに対して行う処理はすべて共通です。すなわち処理対象が 100 万件あるのであれば異なる 100 万件のデータに対して同じ処理を 100 万回繰り返しています。この意味ではオンライン処理で処理される 1 件のトランザクションと変わりはありません。バッチ処理を構成する個々の繰り返し処理の入力となるデータをトランザクションファイルと呼びます。上記の 1) の場合はその日に集積された受注データがこれに相当し、2) の場合は従業員情報のデータが相当します。
バッチ処理は夜間などの占有環境で実行されるため、コンピュータ資源を占有することができ排他制御などのオーバーヘッドを受けることなく効率的に実行できる点にメリットがあります。一方で、夜間のバッチ処理は翌朝の業務開始時刻までには確実に処理が完了していることが必須であり、オンライン処理のレスポンスタイムと同様に、ある以下それ以上に厳しい性能要求にさらされる処理となっています。たとえ 0.01 秒しかかからない処理でも 100 万回繰り返せば 3 時間以上になりますので、大量バッチ処理を提示に完了させるためには慎重なプログラミングが必要となります。トランザクションファイル 1 件の読み込みでも最も高速な手段が要求されるため、これを COBOL の順編成ファイルとして扱うことは最適な選択です。ディスクファイルをバイトストリームとして固定バイト数入出力する操作はあらゆる入出力処理の中で最も高速です。オペレーティングシステムが提供する read() 関数は、ディスクドライバを効率的に使用してアプリケーションへの入出力を実現しており余分な処理を排除しています。
例として 1. で例示した受注伝票を考えます。この 1 件のデータは、ヘッダー部 16 バイト、データ部 30 バイト × 8 の合計 256 バイトのサイズの中に伝票を構成する情報がパックされています。これを COBOL のレコード宣言で記述すれば以下の通りとなります。

```
01  受注伝票.
    05 ヘッダー部.
      10 得意先コード  PIC 9(6).
      10 日付.
        15 年          PIC 99.
        15 月          PIC 99.
        15 日          PIC 99.
      10 伝票ＮＯ       PIC 9(3).
      10 区分          PIC 9.
    05 データ部.
      05 受注明細       OCCURS 8 TIMES.
        10 品名コード  PIC 9(6).
        10 数量        PIC S9(7).
        10 単価        PIC 9(6)V99.
        10 金額        PIC S9(9).
```

このレコードに対する COBOL の READ 文は、ディスクファイルから次の 256 バイトを機械的に読み込んでこのレコード領域にコピーしているにすぎません。
一方、このデータを正規化されたリレーショナルデータベースとして設計するならばたとえば以下のようになるでしょう。

```
CREATE TABLE  受注伝票 (
    得意先コード  NUMBER(6,0),
    日付          DATE,
    伝票ＮＯ      NUMBER(3,0),
    区分          NUMBER(1,0),
    PRIMARY KEY (得意先コード, 伝票ＮＯ)
)

CREATE TABLE  受注明細 (
    得意先コード  NUMBER(6,0),
    伝票ＮＯ      NUMBER(3,0),
    品名コード    NUMBER(6,0),
    数量          NUMBER(7,0),
    単価          NUMBER(8,2),
    金額          NUMBER(9,0),
    FOREIGN KEY (得意先コード, 伝票ＮＯ)
      REFERENCES 受注伝票 (得意先コード, 伝票ＮＯ)
)
```

リレーショナルデータベースでは、テーブルとカラムの定義はすべて意味を持って格納されていますので、この 1 件の伝票データを読み取るのに一つ一つのカラムに対する読み取りとデータ変換が発生します。これに必要な処理量は COBOL の順編成ファイルからの READ に比較して数 10 倍のリソースを必要とします。
また、このデータを Java の Bean として設計するならばたとえば以下のようになるでしょう。

```
class  受注伝票 {
  public int 得意先コード;
  public java.util.Date 日付;
  public int 伝票ＮＯ;
  public int 区分;
public 受注明細 MeisaiData[8];
}

class 受注明細 {
  public int 数量;
  public java.lang.BigDecimal 単価;
  public int 金額;
}
```

Java ではこのようなクラス化されたオブジェクトを大量に外部記憶に保存するために直列化・永続化という処理を行いいます。上記のクラスオブジェクトを 100 万件並列化してディスクファイルに保存したものを ObjectInputStream から readObject() する処理は、クラスデータの復活のためのオーバーヘッドを考慮すればやはり COBOL の順編成ファイルからの READ に比較して数 10 倍のリソースを必要とします。
バッチ処理のトランザクションファイルとして COBOL の順ファイルが最も適している理由がここにあります。

2-2　固定長ファイルと可変長ファイル

COBOL で順編成ファイルを扱うためには FILE-CONTROL 段落に SELECT 句を書いて宣言する必要があります。以下に例を示します。

```
SELECT  受注伝票ファイル ASSIGN TO DENPYO
     ORGANIZATION IS SEQUENTIAL.
```

ASSIGN TO DENPYO はプログラムの実行時に対象とするファイルの実体を割り当てるのに使用する割り当て名です。メインフレームでは JCL 中の DD 文で物理的な割り当てを記述しますが、オープン環境では環境変数を経由してパス名を割り当てます。
ORGANIZATION IS SEQUENTIAL はこのファイルは順編成ファイルであることを示しています。この句は省略することもでき、ORGANIZATION 句を書かない場合は省略時指定で順編成であることが仮定されます。
順ファイルのレコードサイズは、特に指定しない限り FD 句で定義されるレコード領域の長さになります。ドキュメント目的で FD 句に以下のように明記することもできます。

```
FD  受注伝票ファイル
     RECORD CONTAINS 256 CHARACTERS.
```

ここに指定するレコードサイズは FD 句で定義されるレコード領域の長さと一致していなければなりません。これは固定長レコードのファイルの宣言方法です。
レコードサイズが可変長の順編成ファイルを利用することもできます。1. で例示した受注伝票の場合、明細データが 8 件で固定となっていますが、実際の伝票には明細が 1 件しかないものもあるでしょう。このような場合、残りの 7 件 (210 バイト) の領域は使用されていない無駄なディスク領域となります。そこでこの部分を可変長にして本当に存在する明細データだけを含むファイルとして利用することができます。このための一つの方法は以下のようにレコード領域を可変長の表として定義することです。

```
FD  受注伝票ファイル.
01  受注伝票.
    05 ヘッダー部.
      10 得意先コード  PIC 9(6).
      10 日付.
        15 年          PIC 99.
        15 月          PIC 99.
        15 日          PIC 99.
      10 伝票ＮＯ      PIC 9(3).
      10 区分          PIC 9.
    05 データ部.
      10 受注明細
        OCCURS 1 TO 8 TIMES DEPENDING ON 明細数.
        15 品名コード  PIC 9(6).
        15 数量        PIC S9(7).
        15 単価        PIC 9(6)V99.
        15 金額        PIC S9(9).
```

このファイルを WRITE する前に「明細数」に書き込む明細データの数を明示的に転記しておくことによって可変長順ファイルに指定された個数で書き込むことができます。また、このファイルを READ すると「明細数」に読み込まれたレコード内の明細データの数が設定されています。この時たとえ「明細数」が 1 であっても 受注明細(2) から受注明細(8) は参照できてしまうことに注意して下さい。この部分を参照しても意味のあるデータは存在しません。読み込まれた明細データの範囲内だけを参照してプログラミングすることはプログラマの責任となっています。
可変長の順編成ファイルを利用するもう一つの方法は

```
FD  受注伝票ファイル
    RECORD IS VARYING FROM 46 TO 256 CHARACTERS
     DEPENDING ON レコード長.
```

のように FD 句の中で可変長であることとその最小・最大サイズを宣言してしまうことです。この場合も同様に WRITE する前に「レコード長」に書き出すサイズを設定し、READ した後で「レコード長」を参照して読み込まれたレコードのサイズを確認することができます。また、同様に READ した後で「レコード長」で返された長さを超える部分を参照しないように注意する必要があります。
可変長ファイルの入出力は、レコードサイズのチェックなどの追加処理のため固定長ファイルより若干のオーバーヘッドがありますが、やはりバイトコピーで実装されるために SQL アクセスなどに比べればかなり高速です。

2-3 システム間交換のテキストファイル

COBOL の順編成ファイルは COBOL 言語でアクセスする限りにおいては最も高速で便利な形式ですが、Java や .NET などの他の言語でもアクセスする場合には必ずしも便利ではありません。メインフレーム以外のオープン環境でどのような言語でも共通にアクセスできるファイル形式は改行コードで区切られたテキストファイルです。他システムの他言語で作成されたテキストファイルを COBOL から読む、または COBOL で作成されたテキストファイルを他システムの他言語で読む方法によってシステム間のデータ交換が可能となります。
COBOL ではテキストファイルは「行順編成」という特殊な順編成ファイルとして入出力することができます。その宣言は以下のように記述します。

```
SELECT  テキストファイル ASSIGN TO TEXTFILE
    ORGANIZATION IS LINE SEQUENTIAL.
```

このように宣言されたファイルは WRITE 文でレコードを書きだす毎にその末尾に自動的に改行コードを添付します。また READ 文で読み込む際にはファイル内の次の改行コードが現れるまでを読み込みます。Visual COBOL では、デフォルトでは WRITE 時にレコードの末尾の空白を削除してから改行を送っており、ファイルサイズの削減を意図しています。
行順編成ファイルの改行は Windows では X'0A0D' の 2 バイトであり、UNIX/Linux では X'0A' の 1 バイトです。ASCII ではこの他にタブスキップ (X'09')、改ページ (X'0C')、ファイル終端 (X'1A') などの特殊制御文字が定義されており、行順編成ファイルの READ ではこのような制御文字に対して特殊な処理がなされます。
こういった文字との混同を防ぐために行順編成ファイル中にバイナリデータやパック十進データを含むことはできません。たとえば、READ 文で読むレコード中にパック十進データ X'010C' が含まれていたとするとこの 2 バイト目はパック受信データの一部なのか改ページコー

ドなのかの区別がつきません。Visual COBOL の場合、この X'0C' は改ページとみなされ行の区切りとして扱われますのでパック十進データは正しく読み取ることができません。
通常の順編成ファイルの入出力ではディスクファイルに対してあるバイト数のストリームを物理的に読み書きしていますが、行順編成ファイルの READ では一文字ずつ内容をチェックして改行が現れるまでを読み込みますし WRITE では行末の空白の削除を行っています。このため行順編成ファイルの入出力は通常の順編成ファイルに比べてかなりのオーバーヘッドがあります。このため特別の理由がない限り COBOL プログラム間のデータ交換のために行順編成ファイルを使用するべきではありません。

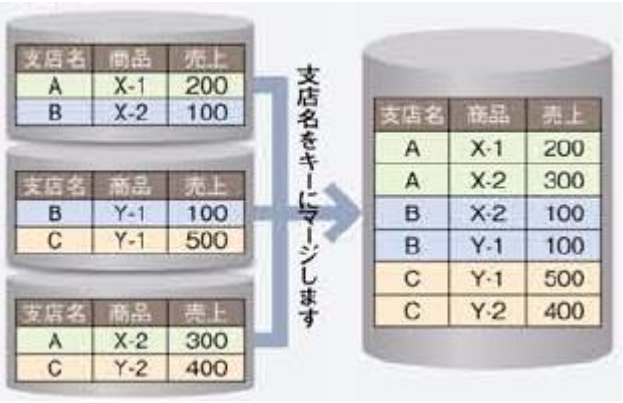
2-4　ソートマージ

個別に集められてきたデータをある順序で並べ替えることは事務処理の中で最も基本的なものです。たとえば多数の契約書データを契約満了日の順番に並べ替えて満了間近なデータから順番に更新処理を行うことなどがこれに相当します。順編成ファイル中のレコードを、それを構成するある項目の値に注目してその順番に並べ替える処理を「ソート」と呼びます。

COBOL では以下のような SORT 文を使用して入力ファイルを指定されたキーでソートして結果を出力ファイルに書き出すことができます。

```
FILE SECTION.
SD  SORT-FILE.
01  SORT-REC.
    05 支店名  PIC X(24).
    05 商品    PIC X(12).
    05 売上    PIC 9(8).

    SORT SORT-FILE ASCENDING KEY IS 支店名 商品
         USING 入力ファイル GIVING 出力ファイル.
```

また、すでにソート済みの同形式の複数ファイルを一つのソート済みファイルにまとめることをマージと呼びます。



COBOL では以下のような MERGE 文を使用して指定されたキーでソート済みの複数の入力ファイルをマージして結果を出力ファイルに書き出すことができます。

```
FILE SECTION.
SD  SORT-FILE.
01  SORT-REC.
    05 支店名  PIC X(24).
    05 商品    PIC X(12).
    05 売上    PIC 9(8).

    MERGE SORT-FILE ASCENDING KEY IS 支店名 商品
          USING 一昨年売上ファイル 昨年売上ファイル
                今年売上ファイル
          GIVING 出力ファイル.
```

ソートやマージのような定型的な処理は COBOL プログラムを書かなくても汎用のユーティリティプログラムで処理することも可能です。COBOL プログラム中の SORT 文を使用して行う処理を「内部ソート」と呼ぶのに対し、ユーティリティによる処理を「外部ソート」と呼びます。このようなユーティリティはメインフレームでは OS の一部として提供されており、ジョブステップの中で利用することができます。Windows や UNIX, Linux の sort コマンドはテキストファイルのソートは可能ですが一般の順編成ファイルは OS 側にその概念がないため対象外となります。これらのオープン OS 上では COBOL ランタイムシステムがユーティリティを提供しています。
Micro Focus が Visual COBOL 製品（およびそのランタイム製品）にバンドルして提供している外部ソートユーティリティは mfsort コマンドです。Windows バッチや UNIX/Linux シェルから以下のようなコマンドでソート処理を実行することができます。

```
mfsort sort fields(1,36,ch,a)
 use 入力ファイル org seq record f,44
 give 出力ファイル
```

コマンド行の fields(1,36,ch,a) は、レコードの先頭から 36 バイト分を文字列型 (ch) で昇順に (a) ソートすることを指示しています。入力ファイルに対する org seq record f,44 はレコード長が固定長 (f) の 44 バイトの順編成 (seq) ファイルであることを指示しています。
ソート処理はコンピュータの黎明期から最も重要なデータ処理の一つでしたので、歴史的に多くの手法が開発されアルゴリズムの改良がなされてきています。現在最も一般的に使用されているアルゴリズムでは N 件のレコードをソートするのに必要な最大計算量は N logN に比例するということがわかっています。これは 10 倍の件数をソートするのに必要な処理時間は 10 倍では済まないということを意味しています。単なる受注登録のような処理であれば N 件のレコードを処理するのに必要な処理時間は N に比例しますので処理対象レコード数が 10 倍に増加すれば処理時間も 10 倍に見積れます。この点でソート処理は業務拡大に伴うバッチ処理の落とし穴になりがちです。夜間のバッチ処理を翌朝のオンライン業務開始までに確実に終了させるためにはソート処理時間退縮のためのチューニングが必要になる場合が多々あります。
ソートのチューニング手法を理解するためにはソートの内部処理を理解する必要があります。10 枚のカードを机上でソートすることを想像してみてください。おそらくテーブルにカードを順に差し込んで並べてゆく方法を取るでしょう。しかしカードが 1 万枚あったらどうするでしょうか。テーブルの上には並び切りません。たとえば以下のような方法があります。

1） 入力デッキから 10 枚ずつ机上でソートする作業を 1000 回行い 1 万枚をデッキ A に置く
2） デッキ A から最初の 100 枚をソート済みの 10 枚ずつの束で取り出し 机上に並べる
3） 10 束のそれぞれ最初の 1 枚を開きその中で最小のカードを取り出して出力デッキに置く
4） カードを取り出した束の次のカードをめくり、新たに開かれている 10 枚から最小のカードを取り出して出力デッキに置く
5） これを繰り返すと最初の 100 枚のカードがソートされて出力デッキに積まれる（マージ処理）
6） 次の 100 枚をソート済みの 10 枚ずつの束で取り出し 机上に並べ同じことを繰り返す
7） これを 100 回繰り返すと出力デッキには 100 枚ずつソートされた 100 個の束が積み重なる
8） 出力デッキから最初の 1000 枚をソート済みの 100 枚ずつの束で取り出し机上に並べる
9） 3), 4), 5), 6), 7) の手順によって最初の 1000 枚をマージしてデッキ A に積む
10）これを 10 回繰り返すとデッキ A には 1000 枚ずつソートされた 10 個の束が積み重なる
11）デッキ A からソート済みの 1000 枚ずつの 10 束を取り出し 机上に並べる
12）3), 4), 5), 6), 7) の手順によって 10000 枚をマージして出力デッキ に積む

この方法であればテーブル上には入出力デッキとデッキ A および 10 個の束を置くスペースがあれば 1 万枚でもいつかはソートすることができます。入出力デッキを除く 11 個のスペースは常に先頭の 1 枚だけが見えていればよいという意味で順編成ファイルです。ソート処理をコンピュータで行う場合もこれと同じことが行えます。もちろんコンピュータで処理する場合には 1 万枚を全件メモリに並べてから

高速ソート処理を行うこともできます。しかしこれには多くのメモリを消費します。1 万枚なら可能でしょうが 100 万枚ならどうでしょうか。大量件数のソートの場合は上記のような作業用のファイルを使用して徐々にソートしてゆくことによって限られたメモリの範囲内でどんなに大量のレコードでもソートすることができるのです。
このような背景のために、大量件数のソート処理の時間短縮はメモリ使用量とディスク入出力回数のトレードオフにかかってきます。机上ソートの例でいえば、もし広い机が使用できるのであれば 10 束ずつではなく 100 束ずつ並べてソートしてゆけば 100 枚ずつソートされた束を 100 個並べてマージしてゆくことでより高速にソートできます。特にコンピュータ上の処理では常にメモリ演算はディスク入出力より高速です。一方あまりにメモリを大量に消費すると OS の仮想記憶域を圧迫しページング回数が増えることによるシステム全体の性能低下につながります。
Visual COBOL ではソート処理に使用するメモリサイズを SORTSPACE 環境変数で自由に設定することができます。もしすべてをメモリ上で処理できる程度の件数であればこの環境変数値をレコードの合計サイズ分に指定すれば一切のディスク入出力なしに処理が完了します。大量件数であればディスク入出力の発生は避けることができません。このようにソート処理で一時的な置き場として使用するディスクファイルをソート処理の「一時ファイル」と呼びます。Visual COBOL ではこのファイルはデフォルトで実行時のカレントディレクトリに作成されますが、一時ファイルを高速アクセスのデバイスに割り付けることによって処理時間を短縮できることがあります。Visual COBOL では TMP 環境変数 (Windows) または TMPDIR 環境変数 (UNIX/Linux) で一時ファイルの使用パスを明示的に指定することができます。これをファイバーチャネルや RAM ディスクファイルに割り当てることも検討に値します。

3 索引ファイル

本章では COBOL による索引編成ファイルアクセスに関連する話題を解説します。

3-1 マスターファイルとしての索引編成ファイル

索引編成ファイルは、キー値で指定されたファイル内の固有のレコードにランダムにアクセスできる機能を持つファイル編成です。順編成ファイルがバッチ処理のトランザクションファイルや中間ファイルとして使用されるのに対し、索引編成ファイルはマスターファイルとして使用されます。たとえば従業員マスターファイルは全従業員に関する様々な情報を保持していますが、従業員番号や従業員名でランダムアクセスできなければなりません。
索引編成ファイルの概念は SQL を使い慣れた方には理解しやすいものです。実際、歴史的には索引編成ファイルはデータベースの概念に先立って登場しており、その後複数の索引編成ファイルの有機的な統合を実現するためにデータベース管理システムが考案された経緯があります。

(1) アクセスモードと COBOL 構文

索引編成ファイルはランダムアクセスをサポートするファイル編成ですが、レコード順のアクセスも可能です。この場合は主キーと呼ばれる項目、たとえば従業員マスターファイルにとっての従業員番号の順番にアクセスします。以下にその宣言例を示します:

```
FILE-CONTROL.
    SELECT 従業員ファイル ASSIGN TO JYUGYOIN
        ORGANIZATION IS INDEXED
        ACCESS MODE IS SEQUENTIAL
        RECORD KEY IS 従業員番号.
FILE SECTION.
FD  従業員ファイル.
01  従業員レコード.
    05 従業員番号  PIC 9(10).
    05 従業員名    PIC X(24).
    05 生年月日    PIC 9(8).
    05 入社年月日  PIC 9(8).
```

ORGANIZATION IS INDEXED はファイルが索引編成であることを示します。
ACCESS MODE IS SEQUENTIAL はプログラムから順アクセスすることを示します。この句は省略可能でその場合は暗黙的に順アクセスとして取り扱われます。順アクセスの場合は順編成ファイルと同様に READ 文、WRITE 文でキーの値の昇順に読み書きします。WRITE の場合は必ずキーの昇順に書いて行かなければならない点に注意が必要です。WRITE 文の実行時にキーの順序が正しくないとファイルステータス "21" が返されて失敗します。
RECORD KEY IS 句では、ファイルのレコード宣言内に書かれているキー項目の名前を指定します。この例ではレコードの先頭 10 バイト分の従業員番号がキーであることを示しています。キーはレコード中のどこにあっても構いません
ランダムアクセスの場合の宣言は ACCESS MODE IS RANDOM (乱アクセス) を指定します。この場合は READ 文の実行前にレコード内の従業員番号にアクセスしたいレコードの従業員番号を設定しておくことによって必要なレコードを読み取ることができます。SQL 文でいえば WHERE 条件でキーの値を指定するのと同じです。以下に例を示します。

```
    MOVE 10000 TO 従業員番号.
    READ 従業員ファイル KEY IS 従業員番号.
```

このときもし該当するキー値のレコードがファイル内に存在しない場合には、READ 文はファイルステータス "23" を返します。あるいは以下のような INVALID KEY 句を追加することでレコードの存在チェックを行うことも可能です。

```
MOVE 10000 TO 従業員番号.
READ 従業員ファイル KEY IS 従業員番号
   INVALID KEY DISPLAY “レコードがありません。”
END-READ.
```

索引編成ファイルではさらにもう一つのアクセスモードとして ACCESS MODE IS DYNAMIC （動的アクセス）があります。これはキーの値の条件で位置づけを行ってそこからキーの値の順に読んで行く方法です。以下に例を示します。

```
MOVE 10000 TO 従業員番号.
START 従業員ファイル KEY IS > 従業員番号.
READ 従業員ファイル NEXT
  AT END SET END-OF-FILE TO TRUE
END-READ.
```

この START 文でファイル内の従業員番号が 10000 を超える最初のレコードに位置づけられ、それ以降 READ 文を繰り返すとそのレコードから順番に読まれてゆきます。
索引編成ファイルの主キーに同じ値を持つ複数のレコードを持たせることは許容されていません。このことはリレーショナルデータベースのテーブルの主キーが UNIQUE 属性を持つのと同じです。既にレコードが存在している主キーの値で新たなレコードを WRITE するとファイルステータス "22" を返します。また READ 文と同様に WRITE 文にも INVALID KEY 句を書いて重複キーのチェックを行うことができます。

(2) 主キーと副キー

索引編成ファイルは必ず主キーを一つ持たなければなりませんが、各アクセスモードはレコード内の主キー以外の項目についてでも利用することができます。この場合、これらの項目は副キーとして定義しておく必要があります。副キーは以下のように ALTERNATE KEY 句で定義します。

```
FILE-CONTROL.
   SELECT 従業員ファイル ASSIGN TO JYUGYOIN
      ORGANIZATION IS INDEXED
      ACCESS MODE IS SEQUENTIAL
      RECORD KEY IS 従業員番号
         ALTERNATE KEY IS 従業員名.
```

この例では主キーである従業員番号の他に副キーとして従業員名でも SEQUENTIAL・RANDOM・DYNAMIC のアクセスモードが利用できます。
デフォルトでは副キーも主キーと同様に同一キー値のレコードを複数持つことは許容されませんが、以下のように WITH DUPLICATES 句を書くことで重複キー値を許容するようにできます。

```
      ALTERNATE KEY IS 従業員番号 WITH DUPLICATES.
```

以上述べてきたように索引編成ファイルには以下のような属性があります。

- レコード長
- 主キーのレコード内オフセットと長さ
- 副キーのレコード内オフセットと長さ（0 個または複数個）
- 各副キーが重複値を許すかどうか

これらの属性は索引編成ファイル自体の中に書き込まれて記録されます。ファイル自体に記録された属性と COBOL プログラム内で宣言された属性は一致していなければなりません。属性が一致しない場合には OPEN 文がファイルステータス "35" を返して失敗します。
アクセスモード（順・乱・動的）はファイル自体の属性ではなく、各プログラムがファイルを取り扱う際の指定であることに注意してください。同じ索引編成ファイルを異なるプログラムから異なるアクセスモードで利用することができます。

(3) レコードの REWRITE/DELETE

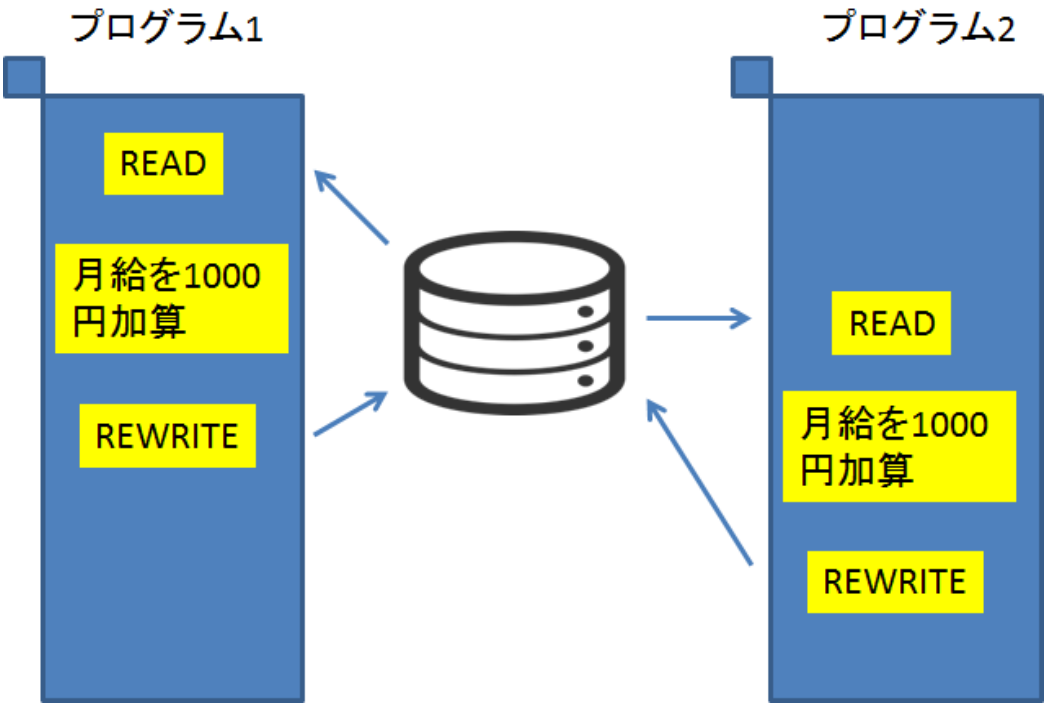
WRITE 文が索引編成ファイルに新たなレコードを追加するのに対して、REWRITE 文は既存のレコードの内容を書き換えます。
順アクセスしているファイルに対する REWRITE 文はその直前に実行された READ 文で読まれたレコードを書き換えます。この時レコードはレコード内の主キーの値で特定されますので、READ してから REWRITE するまでの間に主キーの部分の値を変更してはなりません。もし変更すると REWRITE 文はファイルステータス "21" を返して失敗します。
RANDOM または DYNAMIC のアクセスモードの場合は、事前に READ してある必要はなく、REWRITE する際のレコード内の主キーの値で書き換えられるレコードが決まります。この場合は該当する主キー値のレコードがファイル内に存在していなければなりません。そうでなければ REWRITE 文はファイルステータス "23" を返して失敗します。
DELETE 文はファイルからレコードを削除します。順アクセスしているファイルに対する DELETE 文は、REWRITE 文と同様にその直前に実行された READ 文で読まれたレコードを削除します。
RANDOM または DYNAMIC のアクセスモードの場合は、事前に READ してある必要はなく、DELETE する際のレコード内の主キーの値で書き換えられるレコードが決まります。この場合は該当する主キー値のレコードがファイル内に存在していなければなりません。そうでなければ DELETE 文はファイルステータス "23" を返して失敗します。

3-2　マルチユーザ排他制御

(1)　排他制御の必要性

同じ索引編成ファイルに対して WRITE/REWRITE/DELETE のような更新する操作を同時に複数のユーザから行うと様々な問題を引き起こします。
例えば、従業員ファイルのレコードに「月給」という項目があり、あるプログラムがある社員のレコードを READ して現在の月給に 1000 を加算してから REWRITE するような処理を考えます。この処理を異なる目的の 2 つのプログラムが行うとこの社員の月給は 2000 円アップしなければなりません。しかし、この 2 つのプログラムが同時に実行されると READ 文と REWRITE 文の実行タイミングによっては 1000 円しかアップしない事態が発生します。以下のようにプログラム 1 が REWRITE するのを待たずにプログラム 2 が READ してしまった場合がこれに相当します。

図３－１　同時更新のイメージ



このような問題を回避するためには、READ してから REWRITE するまでの間該当するレコードを他のプログラムからアクセスさせないようにする必要があります。これを排他制御と呼びます。

(2)　ファイルロック

もっとも原始的な排他制御は、プログラムがファイルをオープンした際にファイル自体をロックして他のプログラムからオープンさせないようにすることです。これは以下のように OPEN 文の SHARING 句で指定することができます。

```
OPEN I-O 従業員ファイル SHARING WITH NO OTHER.
```

この文はファイルを CLOSE するまで他のプログラムから一切の OPEN を禁止します。一方、

```
OPEN I-O 従業員ファイル SHARING WITH READ ONLY.
```

は、OPEN INPUT 文による更新しない OPEN だけを許容します。また、ファイルロックをかけないことを明示的に指定するには、

```
OPEN I-O 従業員ファイル SHARING WITH ALL OTHER
```

と書きます。
ファイル入出力に関する排他制御の COBOL 構文は 2002 年の COBOL 国際規格まで標準化されておらず、そのために COBOL システムによってその書き方が異なっていました。Micro Focus では OPEN 文の WITH LOCK 句でロックを指定する構文と、OPEN 文ではなくファイル管理段落の SELECT 句中に LOCK MODE IS EXCLUSIVE 句を書いて占有利用するファイルであることを宣言する構文をサポートしていました。これらの構文は過去に書かれたプログラムの互換性を守るために最新の Visual COBOL でもサポートされています。
SHARING 句を書かない OPEN 文がファイルロックをどのように扱うかは COBOL システムによって異なります。Visual COBOL では以下順序でファイル排他モードを決定する規則となっています。

OPEN 文で WITH LOCK 指定をした場合	SHARING WITH NO OTHER
SELECT 句で LOCK MODE IS EXCLUSIVE 指定をした場合	SHARING WITH NO OTHER
SELECT 句で LOCK MODE IS MANUAL または LOCK MODE IS AUTOMATIC 指定をした場合	SHARING WITH ALL OTHER
OPEN 文のモードが OUTPUT、I-O または EXTEND の場合	SHARING WITH NO OTHER
OPEN 文のモードが INPUT で実行時の構成オプション OPENINPUTSHARED が OFF に設定されている場合	SHARING WITH READ ONLY
OPEN 文のモードが INPUT で実行時の構成オプション OPENINPUTSHARED が ON に設定されている場合	SHARING WITH ALL OTHER

(3) レコードロック

ファイルロックは簡単で確実な排他制御方法ですが、オープンしてからクローズするまでファイルが占有されるため、複数プログラムの並列実行の多重度は制限されます。このため多数のユーザが集中してアクセスするファイルに対しては好ましくありません。このような場合にはファイル全体を複数ユーザで共有してオープンしながら、その中のレコード単位でロックをかける方法が適しています。
レコードロックには、各々の入出力文でレコードのロック管理を意識せずに行う「自動ロック」と、各々の入出力文で意識的にレコードのロック管理を行う「手動ロック」があります。
自動ロックでは、READ されたレコードが自動的にロックされます。これを使用するにはファイル管理段落の SELECT 句中に LOCK MODE IS AUTOMATIC 句を書きます。

```
FILE-CONTROL.
    SELECT 従業員ファイル ASSIGN TO JYUGYOIN
        ORGANIZATION IS INDEXED
        ACCESS MODE IS SEQUENTIAL
        RECORD KEY IS 従業員番号
            LOCK MODE IS AUTOMATIC.
```

自動ロックされたレコードはデフォルトでは次のレコードが READ されてロックされる際に自動的にロック解除されます。この方式を「単一レコードロック」と呼びます。一方

```
            LOCK MODE IS AUTOMATIC
                WITH LOCK ON MULTIPLE RECORDS.
```

と書くと、ファイルは「複数レコードロック」方式となり、自動的なロック解除はされなくなります。この場合には明示的に UNLOCK 文を実行することによってすべてのレコードのロックが解除することができます。
手動ロックでは READ/WRITE/REWRITE の各文に WITH LOCK 句を追加することによって明示的に対象レコードにロックを書けることを指定します。指定しなければロックはかかりません。これを使用するにはファイル管理段落の SELECT 句中に LOCK MODE IS MANUAL 句を書きます。

```
FILE-CONTROL.
    SELECT 従業員ファイル ASSIGN TO JYUGYOIN
        ORGANIZATION IS INDEXED
        ACCESS MODE IS SEQUENTIAL
        RECORD KEY IS 従業員番号
            LOCK MODE IS MANUAL.
```

手動ロックされたレコードについても、明示的に UNLOCK 文を実行することによってすべてのレコードのロックが解除することができます。
以下にこれらの方式の相違をまとめます。

方式	ロック確保	ロック解除
自動ロック・単一レコード	自動	自動
自動ロック・複数レコード	自動	手動: UNLOCK 文
手動ロック	手動: WITH LOCK 句	手動: UNLOCK 文

3-3　索引ファイルユーティリティ

COBOL プログラムから索引編成ファイルにアクセスする方法は以上述べてきたとおりですが、実際のアプリケーション運用に際してはファイル自体のメンテナンス作業が別途必要となります。ファイルの新規作成・削除・コピーやデータ変換も必要ですが、索引編成ファイルの場合は索引再編成という作業が推奨されます。
索引編成ファイルはキーによる乱アクセスを高速に実施するためにファイル内にキー値の索引を持っています。データベース管理システムと異なり管理する常駐のプロセスが存在しないため、ファイルにアクセスする各プログラムが個別にファイルを開いてこの索引を使用した参照・更新を行っています。レコードの追加・削除に伴いこの索引も張り直しが行われますが、これを繰り返しているとバランスの悪い索引になってアクセス速度が低下することがあります。このため定期的に索引を再編成することで健康的な運用を続けることができます。
さらに索引の張り直しを行っている途中で強制的なシステムダウンなどが発生するとファイル内の索引が不整合な状態のまま残されることがあります。このような状態に陥った際にはファイルの修復が必要となります。
Visual COBOL では Rebuild というユーティリティが提供されており、各種のメンテナンス作業をプログラムを書かずに実行することができます。
(1)　ファイルの新規作成

メインフレームやオフコンの OS では索引編成ファイルはプログラムで使用する前に作成を行っておく必要があります。これに対して Windows や UNIX/Linux 上の COBOL 処理系は事前作成の必要がありません。プログラムから OPEN OUTPUT 文を実行すると自動的にファイルが作成され、WRITE 文でレコードを書きだすごとに自動的にファイルが拡大してゆきます。

(2) ファイルのコピー・削除

ファイルのコピー・削除は Windows や UNIX/Linux が OS として標準提供するコマンドを使用することができます。

(3) 索引編成ファイルの属性表示

メインフレームやオフコンの OS では索引編成ファイルのレコード長やキーに関する属性はファイルシステム自体が記録していますので、OS 提供のユーティリティを使用して知ることができます。しかし Windows や UNIX/Linux ではファイルシステム自体は各ファイルをバイトストリームとしてしか認識していません。Windows のエクスプローラでファイルのプロパティを見てもそこには索引編成ファイルとしての情報はありません。
Visual COBOL の Rebuild ユーティリティを使用して対象ファイルの属性を表示することができます。以下に実行例を示します。

```
> rebuild infile.dat –n
      ファイル                  :  infile.dat
      編成                      :  索引
      形式                      :  IDX-8
      記録モード                :  可変
      圧縮                      :  N
      最大レコード長            :        4096
      最小レコード長            :          77
      索引ノードサイズ          :        1024
      データレコード数          :         342

      キー詳細 ('開始'のオフセットを 1 とする):
        キー     開始     長さ     重複    キー圧縮   スパース文字
          0        1       17       N
>
```

(4) ファイルの形式変換

索引編成ファイルに格納されたデータは COBOL プログラムからアクセスすることは簡単ですが、そのほかの言語やアプリケーションからアクセスすることは困難です。業務上索引編成ファイルに格納して運用しているデータを他システム・他アプリケーションと交換するためには一旦順編成ファイルに変換してやり取りする方法が一般的です。メインフレームやオフコンの OS ではこの目的のために索引編成ファイルのアンロードユーティリティが用意されていますが、Windows や UNIX/Linux では COBOL システム側が同等のユーティリティを用意しています。Visual COBOL では Rebuild ユーティリティでこれをサポートしています。以下に実行例を示します。

```
> rebuild infile.dat,outfile.dat –o:ind,seq
```

この例では索引編成ファイルの infile.dat を順編成ファイルの outfile.dat に形式変換しています。「-o:ind,seq」が索引編成から順編成への変換であることを示しています。infile.dat が固定長の索引編成ファイルであれば outfile.dat も入力と同じレコードサイズの固定長順編成ファイルになります。オプションで明示的に出力ファイルのレコードサイズを指定することもできます。
逆に「-o:seq,ind」と指定することで順編成から索引編成変換も可能です。この場合には変換先の索引編成ファイルのキー属性を指定する必要があります。以下に例を示します。

```
> rebuild infile.dat,outfile.dat –o:seq,ind –k:1+2:3+10d
```

この例の「-k:1+2:3+10d」は、出力される索引編成ファイルはレコードの先頭から 2 バイト分は主キーであり、3 バイト目から 10 バイト分が重複キー値を許容する副キーであることを示しています。

(5) 索引編成ファイルの再編成・修復

Rebuild を何のオプションも指定せずに実行すると対象ファイルの索引再編成を行います。

```
> rebuild infile.dat
```

3-4 Fileshare によるトランザクション制御

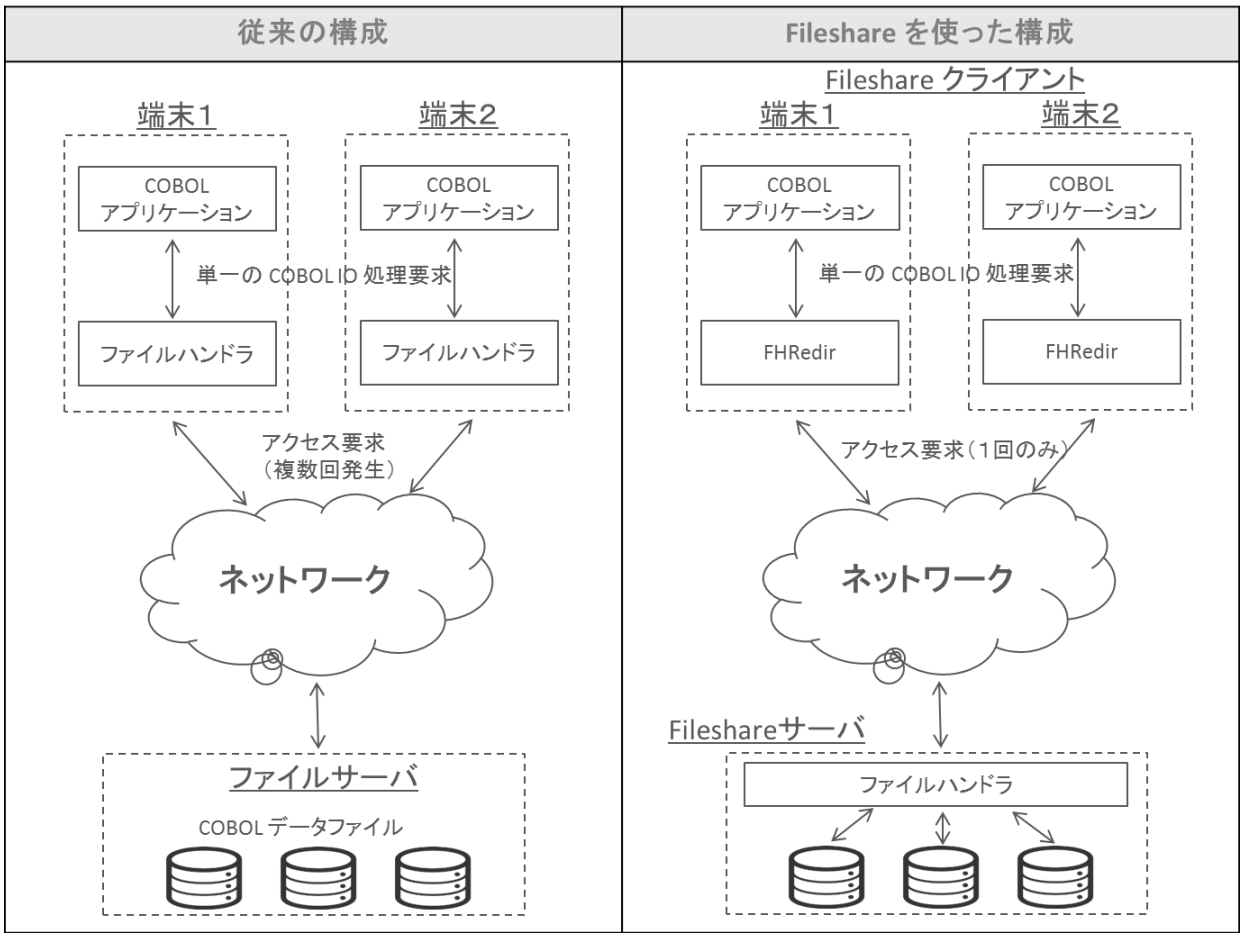
Fileshare はこれまでに紹介したような COBOL の言語規格で定められたデータファイルのハンドリング処理に加えて、主に以下のような機能を提供する Visual COBOL のファイル処理支援機能となります。

> COBOL データファイルのトランザクション処理
> リカバリロギング
> 認証機能等を使った COBOL データファイルのセキュリティ管理
> ネットワークトラフィックの効率化

本項ではこれらの Fileshare の代表的な機能の概要を紹介いたします。特にトランザクション処理機能やリカバリロギング機能を駆使すると、COBOL のデータファイルに対して RDB 等と同様の ACID 特性を満たしたトランザクションを実現させることが可能です。

まずは典型的な Fileshare の構成について見てみます。

図 3 – 2 ネットワーク上で共有されるファイルへの COBOL IO 処理要求



上図のように従来の Fileshare を使わない構成においては、ネットワーク上で共有されているファイルに対して IO 処理要求があるとローカルにある File Handler がその処理をハンドリングします。そのため、1 つの IO 処理要求に対して複数回物理ファイルへアクセスする必要がある場合は、その回数分ネットワークをまたがる必要があります。一方、Fileshare ではローカルにインストールする Visual COBOL のファイルハンドラは IO 処理要求に対する処理はしません。FHRedir(File Handling Redirector) というモジュールがその要求を設定に基づき Fileshare サーバに渡します。Fileshare サーバはその要求に対して Fileshare サーバ上で稼働する File Handler に処理をさせます。従いまして、Fileshare においては 1 つの IO 処理要求に対してラウンドトリップが発生するのは 1 回のみとなります。Fileshare クライアントと Fileshare サーバは単一のマシンで同居することも可能です。この場合ネットワークをまたがったアクセスが発生しないため、ネットワークトラフィックの効率化によるパフォーマンス改善は得られませんが、後述する COBOL データファイルに対するトランザクション管理の導入目的で構築することが可能です。

ここからは Windows 8.1 上で稼働する Fileshare サーバに対して Windows 8.1 及び Red Hat Enterprise Linux 6.5 上で稼働する Fileshare クライアント 2 台からアクセスするシナリオを例にとり構成概要を例示します。Fileshare サーバと Fileshare クライアントの間にユーザ ID、パスワードを使ったセキュリティ管理を導入し堅牢な環境を構築することも可能です。この機能の構成方法も簡単に俯瞰します。続いて簡単なサンプルアプリケーションを使い Fileshare のトランザクション処理やリカバリロギングがどのように処理されるか俯瞰します。

3-3-1 Fileshare サーバの構築

1） Database Reference ファイルの準備
Database Reference ファイルは Fileshare サーバが扱うファイルやログファイルへの参照情報を管理するファイルです。ここでは、下記のコマンドを使って、Visual COBOL コマンドプロンプトのカレントディレクトリ配下に dbrf.ref という Database Reference ファイル及び dbrflog というログファイルを用意します。ログファイルには拡張子を指定しません。「L」(カレントのログファイル)や「B」（バックアップ）から始まる拡張子がシステムにより自動付与されます。

```
C:¥work¥FileShare>fs /d dbrf.ref /l dbrflog
Micro Focus COBOL
Version 2.2.01204 Copyright (C) Micro Focus 1984-2014. All rights reserved.
Fileshare Version 2 Revision 19999

FS101-I d dbrf.ref -- パラメータが受け入れられました
FS101-I l dbrflog -- パラメータが受け入れられました

FS089-I データベース参照ファイルメンテナンスユーティリティ
FS130-I データベース参照ファイルを読み取っています。お待ちください
FS103-I ログファイル名を、データベース参照ファイルに追加しました
FS090-I データベース参照ファイルのインストールが問題なく完了しました

C:¥work¥FileShare>
```

2） Database Reference ファイルに COBOL アプリケーションから IO 処理されるファイルの情報を登録
先ほど用意した Database Reference ファイル dbfm.ref にカレントディレクトリ配下の FSDEMOIDX.DAT を認識させます。

```
C:¥work¥FileShare>fs /d dbrf.ref /f FSDEMOIDX.DAT
Micro Focus COBOL
Version 2.2.01204 Copyright (C) Micro Focus 1984-2014. All rights reserved.
Fileshare Version 2 Revision 19999

FS101-I d dbrf.ref -- パラメータが受け入れられました
FS101-I f FSDEMOIDX.DAT -- パラメータが受け入れられました

FS089-I データベース参照ファイルメンテナンスユーティリティ
FS130-I データベース参照ファイルを読み取っています。お待ちください
FS109-W ファイル FSDEMOIDX.DAT がありません
FS106-I FSDEMOIDX.DAT - データベース参照ファイルへの追加
FS090-I データベース参照ファイルのインストールが問題なく完了しました

C:¥work¥FileShare>
```

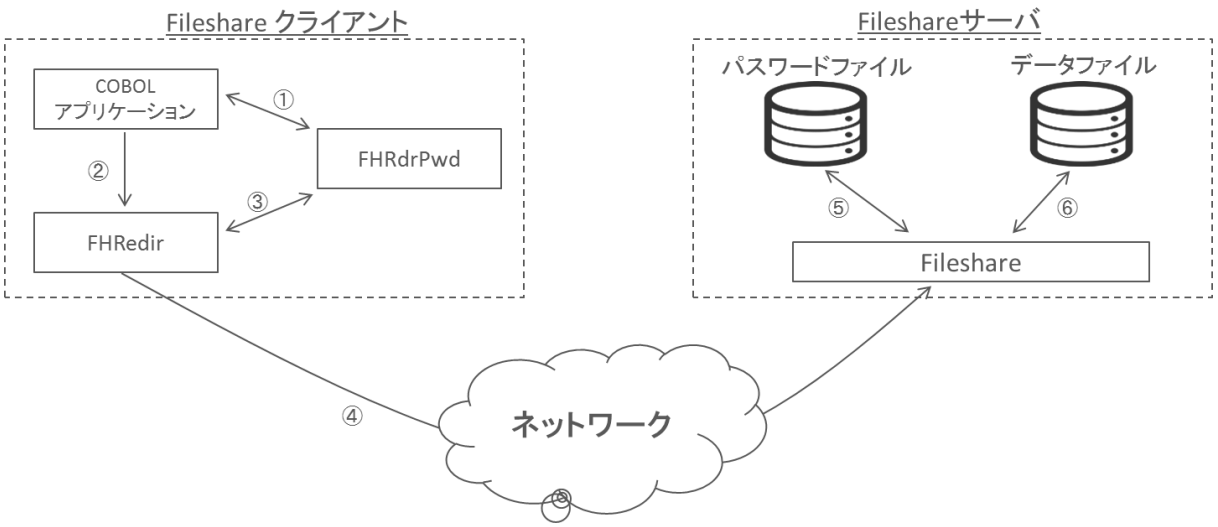
3） Password ファイルの作成及び認証情報を 1 件登録
下記コマンドにより passwd.fil という Password ファイルが作成されます。また、同時に /u、/pw オプションを指定しユーザ ID「ym」、パスワード「password」という認証情報を 1 件登録しています。

```
C:¥work¥FileShare>fs /pf passwd.fil /u ym /pw password
Micro Focus COBOL
Version 2.2.01204 Copyright (C) Micro Focus 1984-2014. All rights reserved.
```

```
Fileshare Version 2 Revision 19999

FS101-I pf passwd.fil -- パラメータが受け入れられました
FS101-I u ym -- パラメータが受け入れられました
FS101-I pw password -- パラメータが受け入れられました

FS094-I パスワードファイルメンテナンスユーティリティ
FS134-I パスワードファイルが見つかりません - 新しいパスワードファイルの作成中
FS096-I パスワードファイルのインストールが完了しました

C:¥work¥FileShare>
```

Fileshare は下図のようなフローでプログラム中の一番最初の IO 処理前に Fileshare サーバに対して認証処理を行います。

図３－３　Fileshare のパスワード管理システム



① Fileshare クライアント上の COBOL アプリケーションは FHRdrPwd(File Handling Redirector password) モジュールを呼び出し、ユーザ ID　及びパスワードを登録します。
② Fileshare サーバ上の COBOL データファイルに対して COBOL の IO 処理要求を投げます。
③ FHRedir は FHRdrPwd を呼び出し、①で登録された認証情報を取得した上で、要求及びその認証情報を Fileshare サーバに渡します。
④ 認証情報がネットワークをまたいでパスされます。
⑤ Fileshare は渡された認証情報を使ってパスワードファイルと突合します。
⑥ 認証情報が正しければ、 Fileshare クライアントと Fileshare サーバの接続を確立し以降の COBOL IO 処理要求を受け付けるようにします。もし、認証に失敗したら COBOL データファイルにアクセスできない旨のエラーが返されます。

エラー出力例：

```
$ cobrun PWDDEMO.gnt

入出力 エラー: ファイル 'FSDEMOIDX.DAT'
エラーコード: 9/037 (ANS74), pc=0, call=1, seg=0
 37  ファイルアクセスが拒否されている

$
```

4） Fileshare サーバの構成ファイルを用意
Fileshare サーバは fs コマンドを使って起動します。この fs コマンドには多くのオプションが用意されておりコマンド実行時にそれらのオプションを指定することも可能ですが、構成ファイルを用意しそこに書かれたオプションを読み込ませることも可能です。本例では下記のような構成ファイルを用意しました。

```
/s win81fssrv
/cm ccitcp
/d dbrf.ref
/t 120
/pf passwd.fil
```

各オプションの指定内容を下表に記します：

/s win81fssrv	ネットワーク上に登録する Fileshare サーバ名を指定します。 ここでは win81fssrv という名前でネットワーク上に登録しています。
/cm ccitcp	Fileshare サーバとクライアントの間で使う通信プロトコルを指定します。Fileshare では TCP/IP、CCISMEM のいずれかの CCI(Common Communications Interface) を通信プロトコルとして利用できます。ここでは TCP/IP を選択しました。TCP/IP を使う場合、ccitcp2 のプロセスを起動し Registration サーバとして機能させます。Visual COBOL 及び COBOL Server for SOA に付属するコンポーネント Micro Focus Directory Server でこの機能を代用することも可能です。
/d dbrf.ref	Fileshare サーバが利用する Database Reference ファイルを指定します。ここでは１）で用意した dbrf.ref を指定しています。
/t 120	レコードロックのタイムアウト時間を秒数で指定します。デフォルトは 60 秒となります。
/pf passwd.fil	パスワード認証によるセキュリティ機能を有効にする場合は、Fileshare サーバが参照する Password ファイルを指定します。ここでは３）で用意した passwd.fil を指定しています。

5） 環境変数 FS に Fileshare サーバの構成ファイルまでのパスを指定

6） Fileshare サーバの起動

```
C:¥work¥FileShare>fs
Micro Focus COBOL
Version 2.2.01204 Copyright (C) Micro Focus 1984-2014. All rights reserved.
Fileshare Version 2 Revision 19999

FS101-I s win81fssrv -- パラメータが受け入れられました
FS101-I cm ccitcp -- パラメータが受け入れられました
FS101-I d dbrf.ref -- パラメータが受け入れられました
FS101-I t 120 -- パラメータが受け入れられました
FS101-I pf passwd.fil -- パラメータが受け入れられました

FS084-I Fileshare サーバ を 開始します (改訂 19999)

FS133-I パスワードファイルを処理しています。お待ちください
FS135-I Fileshare サーバセキュリティはアクティブです
FS130-I データベース参照ファイルを読み取っています。お待ちください
FS104-I ログファイルが見つかりません。新しいリカバリログファイルを作成します
FS186-I Fileshare サーバ がログファイル 'dbrflog.L01' を使用しています
FS108-I データベース参照ファイルに指定されたファイルを処理しています
FS109-W ファイル C:¥WORK¥FILESHARE¥FSDEMOIDX.DAT がありません
FS132-I Fileshare サーバ は、以下の通信プロトコルで実行されています：
        CCITCP
FS085-I Fileshare サーバ は 動作します
```

3-3-2 Fileshare クライアントの構築

1） Fileshare クライアントの構成ファイルを用意
Fileshare クライアントの接続先のサーバ名、通信プロトコル、利用する COBOL データファイルの情報等を構成ファイルに指定します。本例では下記のようなエントリを持った構成ファイルを用意しました。

```
/s win81fssrv
```

```
/cm ccitcp
/f FSDEMOIDX.DAT /s win81fssrv
```

各オプションの指定内容を下表に記します：

/s win81fssrv	このクライアントがアクセスするデフォルトの Fileshare サーバを上で構成した win81fssrv に指定しています。
/cm ccitcp	Fileshare サーバとクライアントの間で使う通信プロトコルを TCP/IP に指定しています。
/f FSDEMOIDX.DAT /s win81fssrv	FSDEMOIDX.DAT に対する COBOL IO 処理要求は win81fssrv にリダイレクトするよう指定しています。 本例では Fileshare サーバをこの構成ファイルで指定していますが、プログラム中の SELECT 文にて下記のように指定することも可能です。 [Windows 場合] SELECT AFILE ASSIGN TO "$$<サーバ名>¥<Fileshare サーバ上のファイルパス>" [UNIX/Linux 場合] SELECT AFILE ASSIGN TO "$$<サーバ名>//<Fileshare サーバ上のファイルパス>"

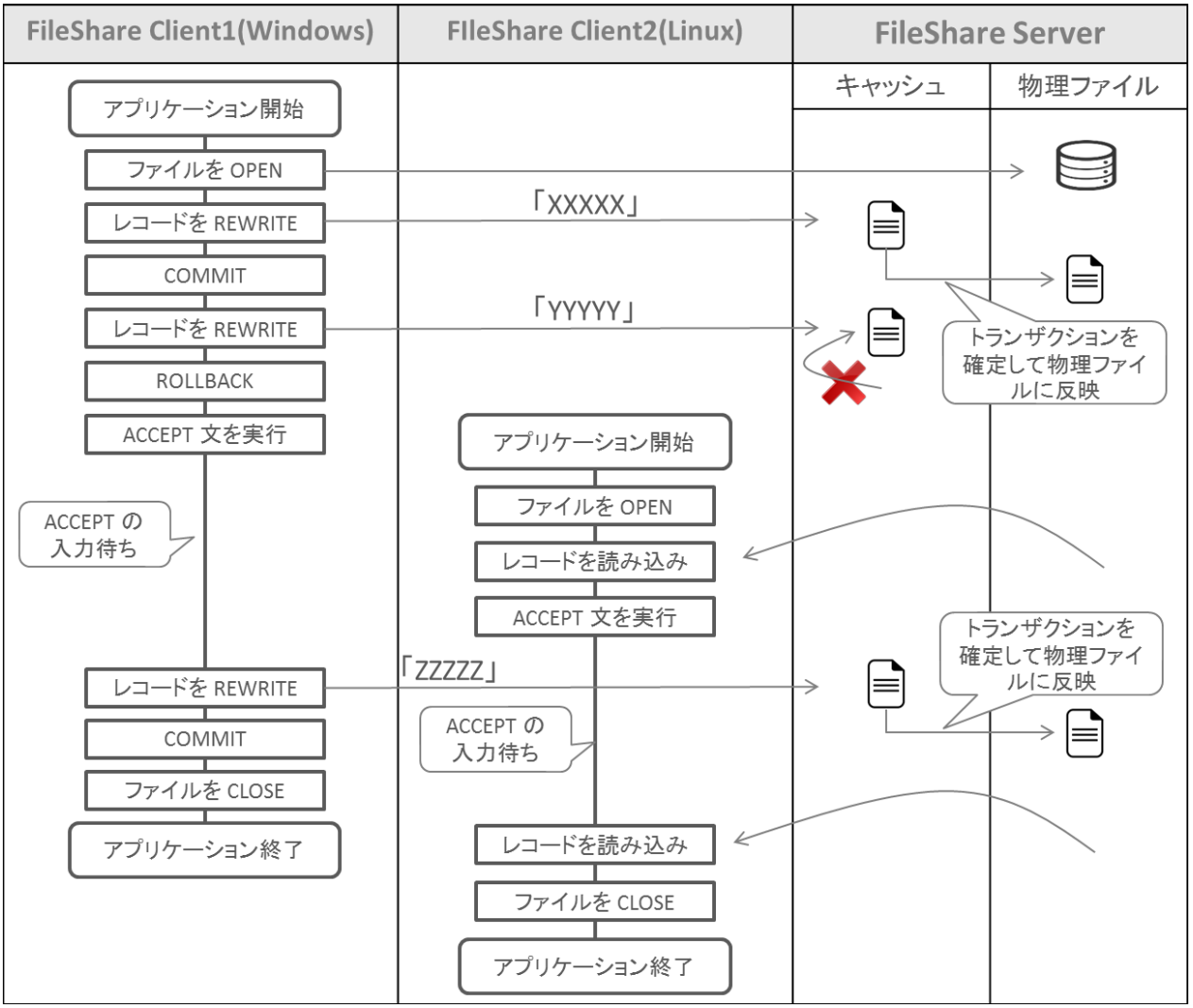
2) 1) で用意した構成ファイルを環境変数 FHREDIR でポイント
3) CCITCP2 を Registration サーバとして利用する場合は、Fileshare サーバの IP アドレスもしくはホスト名を CCITCP2 環境変数でポイント

3-3-3 Fileshare を通じてデータファイルにアクセスする COBOL アプリケーションの開発

以上の設定で Fileshare クライアントと Fileshare サーバが通信するための環境が整いました。ここからは実際に Fileshare を通じて Fileshare サーバ上にホストされている COBOL データファイルにアクセスするアプリケーションを開発してみます。

ここでは下図のようなフローのアプリケーションを考えてみます：

図３－４　サンプルアプリケーション処理フロー



これに伴い、Windows 8.1 及び RedHat Enterprise Linux 6.5 上で稼働する Fileshare クライアントを２台用意しました。これらは Windows 8.1 上で稼働する Fileshare サーバへアクセスします。
Windows の方のクライアントに用意するアプリケーションは Fileshare サーバ上にある索引ファイルに対してレコードの更新(REWRITE)要求を３回投げます。１回目の更新後は COMMIT 文を発行し更新を確定させます。これにより、Fileshare サーバ上で Fileshare サーバのキャッシュ、OS のキャッシュ、もしくはログファイル上で保留されていたトランザクション情報を実際に物理ファイルへ反映させます。２回目の更新後は ROLLBACK 文を発行しトランザクションを取り消します。そのため、２回目の更新データ「YYYYY」はデータファイルには反映されず、１回目の更新で書きだされた「XXXXX」が物理ファイルに記録されている状態となります。３回目の更新後はトランザクションを COMMIT 文で確定させます。これにより最終的には「ZZZZZ」が物理ファイルに記録されていることとなります。
Linux 側に用意するアプリケーションは Windows 側のアプリケーションが更新するレコードを２回読み込みします。１回目の読み込みは２回目の更新に続き ROLLBACK された後に読み込み要求をかけます。この時点では上述のように１回目の更新データがファイルに反映されているため、「XXXXX」が返ってくるはずです。２回目の読み込は Windows 側のアプリケーションの IO 処理が完了した後に処理します。従いまして「ZZZZZ」が読み出されるはずです。

1)　Windows 側のアプリケーションの準備

Windows 側には下記のようなプログラム FSDEMO1.cbl を用意しました。コード下で解説している箇所を除けばデフォルトのファイルハンドラを利用する場合と記法は何ら変わりありません。

```
01      $SET CALLFH"FHREDIR"
02       ENVIRONMENT DIVISION.
03       INPUT-OUTPUT SECTION.
04       FILE-CONTROL.
05          SELECT DEMOIDX ASSIGN TO "FSDEMOIDX.DAT"
06                ORGANIZATION INDEXED RECORD KEY NUMVAL
07                ACCESS MODE RANDOM
08                LOCK MODE IS AUTOMATIC WITH ROLLBACK.
09       DATA            DIVISION.
10       FILE           SECTION.
11       FD  DEMOIDX.
12       01  DEMOIDX-REC.
13          05  NUMVAL       PIC 9(3) COMP-3.
14          05  TEXTVAL      PIC X(5).
15       WORKING-STORAGE SECTION.
16       01 FUNCTION-CODE     PIC X COMP-X VALUE 1.
17       01 USER-NAME         PIC X(20) VALUE "ym".
18       01 USER-PASSWORD     PIC X(20) VALUE "password".
19       01 WK-ITEM           PIC X(10) VALUE SPACE.
20       PROCEDURE        DIVISION.
21       1.
22          CALL "fhrdrpwd" USING FUNCTION-CODE,
23                           USER-NAME,
24                           USER-PASSWORD.
25          OPEN  I-O DEMOIDX.
26          MOVE  333     TO NUMVAL.
27          MOVE  ALL "X" TO TEXTVAL.
28          REWRITE DEMOIDX-REC.
29          COMMIT.
30          MOVE  333     TO NUMVAL.
31          MOVE  ALL "Y" TO TEXTVAL.
32          ROLLBACK.
33          REWRITE DEMOIDX-REC.
34          DISPLAY "ENTER SOMETHING: " WITH NO ADVANCING.
35          ACCEPT WK-ITEM.
36          MOVE  333     TO NUMVAL.
37          MOVE  ALL "Z" TO TEXTVAL.
38          REWRITE DEMOIDX-REC.
39          COMMIT.
40          CLOSE        DEMOIDX.
41          GOBACK.
```

01 行目	COBOL の IO 処理要求はデフォルトで指定されるローカルの Visual COBOL のファイルハンドラに処理させるのではなく、File Handler Redirector を通じて Fileshare でファイルハンドリングさせる旨をコンパイラに認識させるための指令です。これ以外にも 4 章で紹介するような方法で EXTFH の代わりに CALL"FHREDIR" を直接呼び出して処理させたり、ファイルハンドラ構成オプション FHREDIR を ON にして構成することも可能です。
05 行目	3-3-2 で構成したファイルに割り当てています。
08 行目	Fileshare のトランザクション処理機能を有効にするには SELECT 文にて WITH ROLLBACK 句を指定します。これによりトランザクションがない状態で WRITE、REWRITE、DELETE のような更新要求が投げられるとトランザクションが開始されます。開始されたトランザクションは COMMIT 文、ROLLBACK 文のいずれかで確定、取り消しを行う必要があります。トランザクションがアクティブな状態でファイルを CLOSE しますとエラーが返ってきます。
22 行目	COBOL の IO 処理を開始する前に fhrdrpwd を呼び出し、Fileshare サーバとパスワード認証させます。

COMMIT 文、もしくは ROLLBACK 文を使ってトランザクションを確定もしくは取り消します。

29, 32, 39 行目

2） Linux 側のアプリケーションの準備

Linux 側には下記のようなプログラム FSDEMO2.cbl を用意しました。

```
01      $SET CALLFH"FHREDIR"
02       ENVIRONMENT DIVISION.
03       INPUT-OUTPUT SECTION.
04       FILE-CONTROL.
05          SELECT DEMOIDX ASSIGN TO "FSDEMOIDX.DAT"
06                 ORGANIZATION INDEXED RECORD KEY NUMVAL
07                 ACCESS MODE RANDOM.
08       DATA            DIVISION.
09       FILE           SECTION.
10       FD  DEMOIDX.
11       01  DEMOIDX-REC.
12          05  NUMVAL       PIC 9(3) COMP-3.
13          05  TEXTVAL      PIC X(5).
14       WORKING-STORAGE SECTION.
15       01 FUNCTION-CODE     PIC X COMP-X VALUE 1.
16       01 USER-NAME         PIC X(20) VALUE "ym".
17       01 USER-PASSWORD     PIC X(20) VALUE "password".
18       01 WK-ITEM           PIC X(10) VALUE SPACE.
19       PROCEDURE        DIVISION.
20       1.
21          CALL "fhrdrpwd" USING FUNCTION-CODE,
22                          USER-NAME,
23                          USER-PASSWORD.
24          OPEN  INPUT DEMOIDX.
25          MOVE  333 TO NUMVAL.
26          READ DEMOIDX WITH IGNORE LOCK.
27          DISPLAY "TEXTVAL: " TEXTVAL.
28          DISPLAY "ENTER SOMETHING: " WITH NO ADVANCING.
29          ACCEPT WK-ITEM.
30          MOVE  333 TO NUMVAL.
31          READ DEMOIDX WITH IGNORE LOCK.
32          DISPLAY "TEXTVAL: " TEXTVAL.
33          CLOSE  DEMOIDX.
34          GOBACK.
```

01 行目　Windows 側に用意したプログラムと同様に COBOL の IO 処理要求を File Handler Redirector を通じて Fileshare でファイルハンドリングさせる旨をコンパイラに認識させます。

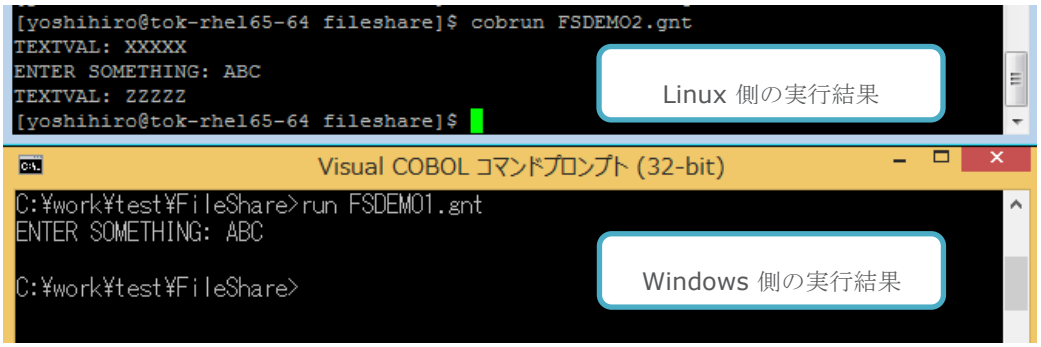
05 行目　3-3-2 で構成したファイルに割り当てています。このプログラムはレコードの参照のみ行い、トランザクション管理をする必要がないため、WITH ROLLBACK 句は指定していません。

21 行目　ターゲットの Fileshare サーバはパスワード認証によるセキュリティ管理が有効なため、Windows 側のアプリケーションと同様に認証処理をさせます。

3) アプリケーションの実行

下図は上で用意したアプリケーションを gnt にコンパイルして実行した結果です。1 回目の READ 文は 2 回目の更新後に実行されていますが、2 回目の更新は ROLLBACK されているので返ってきたのは 1 回目に更新された値となっています。2 回目の READ 文では 3 回目の更新が確定した後に処理しているため、3 回目の更新値「ZZZZZ」が戻されています。

図 3 – 5 サンプルアプリケーション実行結果



3-3-4 Fileshare による Rollforward リカバリ

Fileshare サーバ構築時に指定したログファイルはリカバリ目的で利用することが可能です。3-3-3 でアプリケーションを実行し、データファイルを更新していますが、ここではその更新前の状態にリストアするようすを紹介いたします。

1) Fileshare サーバを停止

Fileshare サーバを起動した Visual COBOL プロンプトにて escape キーを打鍵します。続いて表示されるプロンプトに対して「Y」を返します。

```
                                                    :
FS085-I Fileshare サーバ は 動作します
FS097-I Fileshare サーバ を 終了しますか ? (Y/N)
Y
FS086-I Fileshare サーバ を 終了します
FS087-I Fileshare サーバ は 終了しました

C:¥work¥FileShare>
```

2) アプリケーション実行前の状態にリストア
下記のコマンドを実行して Database Reference ファイル dbrf.ref で管理するデータファイルを 2015 年 1 月 8 日 16 時 00 分 00 秒の状態にリストアします。

```
C:¥work¥FileShare>fs /r dbrf.ref /dt 150108160000
Micro Focus COBOL
Version 2.2.01204 Copyright (C) Micro Focus 1984-2014. All rights reserved.
Fileshare Version 2 Revision 19999

FS101-I r dbrf.ref -- パラメータが受け入れられました
FS101-I dt 150108160000 -- パラメータが受け入れられました

FS091-I ロールフォワードリカバリユーティリティ
FS130-I データベース参照ファイルを読み取っています。お待ちください
FS108-I データベース参照ファイルに指定されたファイルを処理しています
FS037-I ファイル C:¥WORK¥TEST¥FILESHARE¥FSDEMOIDX.DAT
        はファイル見出しレコードを持っていません
FS130-I データベース参照ファイルを読み取っています。お待ちください
FS212-I ロールフォワードリカバリユーティリティ が ログファイル 'dbrflog.L01'
        を処理しています
FS126-I リカバリを行っています。お待ちください
FS139-I ログファイル中に、日付と時間の一致する項目が見つかりました
FS092-I ロールフォワードリカバリ が完了しました

C:¥work¥FileShare>
```

3) Fileshare サーバを起動
4) 3-3-2 で使ったデータファイルを読み込むアプリケーションを再度実行
3-3-2 の時点では Windows 側のアプリケーションにより「ZZZZZ」にレコードが更新されていましたが、そのアプリケーションを実行する前の時点にリカバリしたため、更新前の「CCCCC」という値が戻ってきています。

```
$ cobrun FSDEMO2.gnt
TEXTVAL: CCCCC
ENTER SOMETHING: ABC
TEXTVAL: CCCCC
$
```

4 ファイルハンドラのカスタマイズ

Visual COBOL には前章までに述べてきたような順編成、相対編成、索引編成といった COBOL のファイルの入出力処理をハンドリングする Micro Focus File Handler が備わっています。ファイルの入出力に関連するコードが記述された COBOL プログラムをコンパイルして実行すると、デフォルトではこのファイルハンドラが使用され、ロジックに応じた適切な命令を OS に投げて処理します。このファイルハンドラの挙動は様々な方法でカスタマイズすることが可能です。本章ではこれらそれぞれのカスタマイズ技術を概説します。

4-1 ファイルハンドラ構成オプション

Visual COBOL には 60 以上にも及ぶファイルハンドラの挙動を構成するためのオプションが用意されています。これらを構成することでパフォーマンスに影響し得る動作要素やハンドリング処理に関する各種制限等を調整できます。オプションのリスト及び各オプションのデフォルト値等については製品ドキュメントを参照してください。ファイルハンドラにデフォルト値と異なる動作を指示する場合は、ファイルハンドラ構成ファイルにその任意の値を記述して構成します。このファイルハンドラ構成ファイルは以下のいずれかの方法で Visual COBOL に認識させます。

- 環境変数 EXTFH で構成ファイルをポイント
- $COBDIR/etc に extfh.cfg を配置
- カレントディレクトリに extfh.cfg を配置
- CBL_CFGREAD_EXTFH ライブラリルーチンでプログラムより動的に指定

以下にファイルハンドラ構成オプションの構成例を記します：

1) 10 バイトのレコードを持つ行順ファイルに末尾に 5 バイトの空白値を含んだデータ書き込む COBOL プログラムを用意

```
$ cat CREATELS.cbl
       FILE-CONTROL.
          SELECT  LSFILE      ASSIGN TO  "TESTLS.dat"
                 ORGANIZATION IS LINE SEQUENTIAL.
       DATA DIVISION.
       FILE SECTION.
       FD  LSFILE.
       01  LSFILE-REC              PIC X(10).
       PROCEDURE DIVISION.
          OPEN  OUTPUT  LSFILE.
          WRITE LSFILE-REC FROM "ABCDE     ".
          CLOSE LSFILE.
          GOBACK.
$
```

2) デフォルトの状態でこのプログラムを実行し、プログラムにより生成されたファイルを確認

```
$ cobrun CREATELS.gnt
$ od -x TESTLS.dat
0000000 4241 4443 0a45
0000006
$
```

データ末尾の空白値がトリムされてファイルに書き込まれています。これはファイルハンドラ構成オプション STRIPSPACE のデフォルト動作(ON)によるものです。

3) STRIPSPACE の動作をデフォルトの ON から OFF に変更する構成ファイルを用意

カレントディレクトリに extfh.cfg を用意：

```
$ cat extfh.cfg
[XFH-DEFAULT]
STRIPSPACE=OFF
$
```

XFH-DEFAULT タグは全ファイルに対してそのタグ以下の構成を反映させるための指定です。特定のファイルにのみ構成を反映させたい場合は、対象のファイル名のタグを用意します。

4) プログラムを再度実行し、プログラムにより生成されたファイルを確認

```
$ cobrun CREATELS.gnt
$ od -x TESTLS.dat
0000000 4241 4443 2045 2020 2020 000a
0000013
$
```

空白値をトリムせずにファイル書き込みしています。

上述のように Visual COBOL ではこの STRIPSPACE の他に 60 以上にも及ぶ豊富なオプションを用意しファイルハンドラの柔軟なカスタマイズを可能としています。以下に上記例の他に Visual COBOL がサポートする主なファイルハンドラ構成オプションを列記します：

> 行順編成ファイル入出力時にタブ文字の展開可否を指定する **EXPANDTAB**, **INSERTTAB**
> I/O バッファサイズを指定する **SEQDATBUF**, **IDXDATBUF**, **RELDATBUF**
> OPEN INPUT したファイルの READ 時にレコードロックを無視させる **IGNORELOCK**
> ロックされているファイルの OPEN でのリトライ有無を指定する **RETRYOPEN**
> ロックされているレコードの READ でのリトライ有無を指定する **RETRYLOCK**

4-2 FCD パラメータ値のカスタマイズ

何らかの I/O 処理を伴う COBOL アプリケーションが実行されると Visual COBOL は、FCD(File Control Description) と呼ばれる領域を各ファイルに対して確保します。この領域にはファイル編成、アクセスモード、レコード長をはじめとした約 80 にも及ぶファイルの各種属性情報が保持されます。Micro Focus File Handler はこの情報に基づきファイルのハンドリング処理を行います。
Visual COBOL は FCD2 と FCD3 という 2 つの FCD フォーマットを用意しています。デフォルトでは FCD3 が使用されますが、32-bit 版の旧製品と互換性を持たせる等の目的で FCD3 コンパイラ指令を無効にして FCD2 を使用するよう調整することも可能です。実際のフォーマットについては $COBDIR/cpylib 配下にある xfhfcd3.cpy(FCD3 フォーマット) 及び xfhfcd2.cpy(FCD2 フォーマット)で確認が可能です。
ファイルハンドラは索引ファイルを処理する場合、この領域に加えてキー定義ブロックと呼ばれる索引キーに関する情報を保持するエリアも確保します。このキー定義ブロックは以下の 3 つの領域で構成されます。

- グローバル情報領域
- キー定義領域
- 構成要素定義領域

COBOL プログラムはこの FCD へアクセスしてファイル情報を参照したり、FCD のパラメータ値を変更することも可能です。FCD へは以下のようにしてアクセスします。

- FCDREG コンパイラ指令を指定
- LINKAGE SECTION に FCD 定義を用意し、それを SET 文で実際の FCD へマップ
- 索引ファイルの場合は、LINKAGE SECTION にキー定義ブロック定義を用意し同様に SET 文でマップすることでアクセスが可能

ここでは下記のようなプログラムを使って FCD へアクセス及び FCD パラメータ値のカスタマイズをしてアプリケーションの挙動を調整するようすを見てみます。

```
01        $SET FCDREG
02         ENVIRONMENT DIVISION.
03         INPUT-OUTPUT SECTION.
04         SELECT DEMOSEQFILE1 ASSIGN TO 'DEMOSEQ1.dat'.
05         SELECT DEMOSEQFILE2 ASSIGN TO 'DEMOSEQ2.dat'.
06         DATA DIVISION.
07         FILE SECTION.
08         FD DEMOSEQFILE1.
09         01 DEMOSEQ1-REC            PIC X(10).
10         FD DEMOSEQFILE2.
11         01 DEMOSEQ2-REC            PIC X(10).
12         LINKAGE SECTION.
13         01 WK-FCD.
14           COPY 'XFHFCD.CPY'.
15         PROCEDURE DIVISION.
16            SET ADDRESS OF WK-FCD TO ADDRESS OF FH--FCD OF DEMOSEQFILE2.
17            MOVE H"02" TO fcd-flags-1.
18            OPEN OUTPUT DEMOSEQFILE1 DEMOSEQFILE2
19            WRITE DEMOSEQ1-REC FROM "AAAAAAAAAA".
20            WRITE DEMOSEQ2-REC FROM "AAAAAAAAAA".
21            WRITE DEMOSEQ1-REC FROM "BBBBBBBBBB".
22            WRITE DEMOSEQ2-REC FROM "BBBBBBBBBB".
23            CLOSE DEMOSEQFILE1 DEMOSEQFILE2.
24            GOBACK.
```

01 行目	FCDREG コンパイラ指令を指定します。本指令を指定することにより特殊レジスタ FH—FCD が作成されます。
14 行目	$COBDIR/cpylib 以下にある xfhfcd.cpy を利用して FCD の構造を定義します。
16 行目	DEMOSECFILE2 の FCD に LINKAGE SECTION で定義した FCD をマップします。
17 行目	fcd-flags-1 のビット 1 を立ててファイル操作のトレースを有効にします。

このプログラムでは、2 つの順ファイルを生成し、それぞれに 2 件ずつレコードを書き込みます。このうち DEMOSEQFILE2 に対しては、このファイルの FCD へプログラムからアクセスさせます。FCD 中の fcd-flags-1 のビット 1 はファイル操作のトレースの有効可否を制御する役割を担います。ここではこのフラグを有効にして DEMOSEQFILE2 のみファイル操作のトレースを生成させるよう制御しています[1]。実際、コンパイル、実行して生成されたトレースを参照すると下記のように DEMOSEQFILE2 に対するファイル操作のみがトレースされます。これにより、COBOL は正しく FCD にアクセスし、Micro Focus File Handler は COBOL プログラムが変更したパラメータを正しく解釈して処理していることがわかります。

1) コンパイル・実行

```
$ cob -u fcdparmdemo.cbl
$ cobrun fcdparmdemo.gnt
$
```

2) 生成されたトレースを確認

```
$ ls XFHTRACE.XFH
XFHTRACE.XFH
$ head -1 XFHTRACE.XFH
Nb▒*▒13322448150213▒N▒▒▒▒ ▒▒▒▒▒▒▒▒ ▒▒▒▒▒▒
▒▒▒▒▒
$
```

トレースはデフォルトでは XFHTRACE.XFH という名前でバイナリ形式で生成されます。

3) トレースをテキスト形式にダンプ

```
$ cobfhrepro3 /d
Session is 0000000000
 I/O     Sess  Op  Opcode   Act   Exp   Filename                    O
Number    Id  Code Desc     Stat  Stat                            K
-------- ---- ---- -------- ----- ----- ------------------------------ -
00000001 0001 FA01 Open      /    0/0   DEMOSEQ2.dat
00000002 0001 FAF3 Write     /    0/0   DEMOSEQ2.dat
00000003 0001 FAF3 Write     /    0/0   DEMOSEQ2.dat
00000004 0001 FA80 Close     /    0/0   DEMOSEQ2.dat
Session 000 finished
$
```

➔ FCD パラメータを変更した DEMOSEQFILE2 に対するファイル処理のみがトレースされていることが確認できます。

[1] ファイル操作のトレースは本例のようにプログラム中で動的に制御しなくとも 4-1 で紹介したファイルハンドラ構成オプションを使って制御することも可能です。

4-3　ファイルハンドラ API の呼出し

Visual COBOL が提供する Micro Focus File Handler には呼び出し可能なインターフェースが用意されています。
プログラマは任意の操作コードをこのインターフェースを通じてファイルハンドラに渡し、ファイル処理関連の COBOL 文を介さずに直接 Micro Focus Filer Handler を呼び出すことが可能です。この操作コードに関しては、COBOL プログラムで発行可能なファイル入出力命令をベースとした標準操作コード、並びに索引の再構成のように COBOL の言語仕様の範囲で表現が難しいような要求を処理する特殊操作コードが用意されています。操作コードの一覧については製品マニュアルを参照してください。
COBOL 文によるファイル入出力要求が発行されると COBOL の言語仕様、コンパイラ指令の指定内容、ファイルハンドラ構成オプションの設定値等に基づき適切な操作コード及び FCD のパラメータがセットされ Micro Focus File Handler が呼び出されます。そのため、プログラムからもこれと同じように Micro Focus File Handler を直接呼出してあげれば COBOL 文を使わなくとも同等の処理を実現することができます。また、操作コードや FCD の組み合わせを COBOL 文ではできないようなかたちに調整してファイルの入出力動作等を独自にカスタマイズさせることも可能です。

ここでは Micro Focus File Handler を直接呼び出す簡単な例を見てみます。

(1) 標準操作コードを使った例 – 相対バイトアドレス指定によるレコード読み込み

```
01        $SET FCDREG
02         ENVIRONMENT DIVISION.
03         INPUT-OUTPUT SECTION.
04         SELECT DEMOFILE ASSIGN TO 'stddemo.dat'.
05         DATA DIVISION.
06         FILE SECTION.
07         FD DEMOFILE.
08         01 DEMOF-REC             PIC  X(10).
09         WORKING-STORAGE SECTION.
10         01 REL-ADDR              OCCURS 3 PIC X(4) COMP-X.
11         01 OP-CODE               PIC X(2) COMP-X.
12         LINKAGE SECTION.
13         01 WK-FCD.
14            COPY 'XFHFCD.CPY'.
15         PROCEDURE DIVISION.
16            SET ADDRESS OF WK-FCD TO ADDRESS OF FH--FCD OF DEMOFILE.
17            OPEN OUTPUT DEMOFILE.
18            MOVE ALL 'A' TO DEMOF-REC.
19            WRITE DEMOF-REC.
20            MOVE FCD-RELADDR-OFFSET TO REL-ADDR(1).
21            MOVE ALL 'B' TO DEMOF-REC.
22            WRITE DEMOF-REC.
23            MOVE FCD-RELADDR-OFFSET TO REL-ADDR(2).
24            MOVE ALL 'C' TO DEMOF-REC.
25            WRITE DEMOF-REC.
26            MOVE FCD-RELADDR-OFFSET TO REL-ADDR(3).
27            CLOSE DEMOFILE.
28
29            OPEN INPUT DEMOFILE.
30            MOVE REL-ADDR(2) TO FCD-RELADDR-OFFSET
31            MOVE H'FAC9' TO OP-CODE
32            CALL 'EXTFH' USING OP-CODE, WK-FCD.
33            DISPLAY "DEMOF-REC: " DEMOF-REC.
34            CLOSE DEMOFILE
35            GOBACK.
```

20 23 26 行目　レコードの相対バイトアドレスを格納する FCD パラメータを WORKING-STORAGE で定義した変数に退避します。各レコードの相対バイトアドレスを退避しています。
32 行目　操作コード「X'FAC9'(Direct READ)」で Micro Focus File Handler を呼び出します。

このサンプルプログラムではまずプログラムが使用する順ファイルの FCD に LINKAGE SECTION で定義した FCD 構造をマップさせます。その後、順ファイルを作成しレコードを 3 件追加します。各レコードを挿入した後にはそのレコードの相対バイトアドレスを WORKING-STORAGE で定義した変数にコピー退避させます。続いて、作成したファイルを OPEN INPUT で開きます。ここで READ 文を実行しますと最初のレコード「AAAAAAAAAA」が返ってきます。しかし、レコードの相対バイトアドレスを格納する FCD パラメータ FCD-RELADDR-OFFSET に 2 番目のレコードの相対バイトアドレスを指定した上で、Direct READ の操作コードを Micro Focus File Handler を呼び出しています。これにより 2 番目のレコードが返ってきます。下記はこの実行結果となります。

```
$ cob -u stdcodedemo.cbl
$ cobrun stdcodedemo.gnt
DEMOF-REC: BBBBBBBBBB
$
```

1 回目の READ 処理で 2 番目のレコードが返ってきていることがわかります。

(2) 特殊操作コードを使った例 – File Information

索引キーの再構成、データのフラッシュ、ファイル情報の取得等 COBOL 文として用意されていないような操作も処理するための機能が Micro Focus File Handler には備わっています。この機能を利用するには特殊操作コードを使います。ここではファイル情報取得機能を使って実態が不明なファイルを解析させるようすを例にとり機能を確認します。

```
01      $SET DEFAULTBYTE"00"
02       WORKING-STORAGE            SECTION.
03       01  WK-FCD.
04           COPY 'XFHFCD.CPY'.
05       01  WK-FILENAME         PIC X(20)  VALUE "UNKNOWN.dat".
06       01  OPCODE             PIC X(2)   VALUE X"0006".
07       PROCEDURE              DIVISION.
08           MOVE fcd--version-number TO fcd-version.
09           MOVE H'FF'               TO FCD-ORGANIZATION.
10           MOVE 11                  TO FCD-NAME-LENGTH.
11           SET fcd-filename-address TO ADDRESS OF WK-FILENAME.
12           CALL "EXTFH" USING OPCODE WK-FCD.
13           DISPLAY "ORGANIZATION    = " fcd-organization.
14           DISPLAY "RECORDING-MODE = " fcd-recording-mode.
15           DISPLAY "MAX REC LENGTH = " fcd-max-rec-length.
16           GOBACK.
```

01 行目　　FCD を全て X'00' で初期化します。
08 - 11 行目　　ファイル情報取得処理に必要なインプットパラメータをセットします。ここではカレントディレクトリにある UNKNOWN.dat というファイルを解析します。
12 行目　　ファイル情報取得処理の操作コード X'0006' を使って Micro Focus File Handler を呼び出します。

実行結果：

```
$ cob -u spcodedemo.cbl
$ cobrun spcodedemo.gnt
ORGANIZATION    = 003
RECORDING-MODE = 001
MAX REC LENGTH = 0000000010
$
```

解析対象の UNKNOWN.dat は可変長で最大 10 バイトのレコードを持つ相対編成ファイルであることがわかりました。

4-4　カスタム化されたファイルハンドラの利用

Visual COBOL には Micro Focus File Handler がビルドインされており、COBOL アプリケーションから発行される入出力要求はこのファイルハンドラが言語仕様や構成に基づいて適切な処理をします。この処理動作は上述のように柔軟にカスタマイズするオプションが用意されています。Visual COBOL では更にこのビルドインされたファイルハンドラではなくファイルハンドラ自体を個別に用意して利用する手段も提供しています。このカスタマイズ化機能を使えば、ファイル入出力処理を言語仕様とは異なる特別な要件に基づいた挙動となるよう処理させたり、固有のフォーマットでファイル処理のトレースを取得させる等といったことが可能となります。この個別で用意したファイルハンドラにファイル入出力処理をさせる場合であっても、COBOL 文自体は特にそれを意識してコーディングする必要はありません。コンパイル時などに使用するファイルハンドラを構成できます。また、ファイルハンドラについてもインターフェースが正しく実装されていれば C 等 COBOL 以外の言語を用いてコーディングすることも可能です。

ここではファイル入出力処理は Micro Focus File Handler に引き続き委譲しますが、カスタマイズされたトレース取得処理を間に挟む簡単な例題を用いてその構成方法を確認してみます。

(1) ファイルハンドラを用意

```
01      $SET FCDREG NOCANCEL
02       ENVIRONMENT DIVISION.
03       INPUT-OUTPUT SECTION.
04       SELECT FHTRACE ASSIGN TO 'FHTRACE.trc'
05       ORGANIZATION IS LINE SEQUENTIAL.
06       DATA DIVISION.
07       FILE SECTION.
08       FD FHTRACE.
09       01 TRACE-REC.
10          03 F-DATE                 PIC 9(8) USAGE DISPLAY.
11          03 FILLER                 PIC X.
12          03 F-TIME                 PIC 9(8) USAGE DISPLAY.
13          03 FILLER                 PIC X.
14          03 F-OP-NAME                 PIC X(10).
15          03 FILLER                 PIC X.
16          03 F-FNAME                  PIC X(51).
17       WORKING-STORAGE SECTION.
18       LINKAGE SECTION.
19       01 OP-CODE                     PIC X(2).
20       01 WK-FCD.
21          COPY 'XFHFCD.CPY'.
22       01 WFNAME                      PIC X(256).
23       PROCEDURE DIVISION USING OP-CODE WK-FCD.
24           SET RETURN-CODE TO 1.
25           OPEN EXTEND FHTRACE.
26           ACCEPT F-DATE FROM DATE YYYYMMDD.
27           ACCEPT F-TIME FROM TIME.
28           SET ADDRESS OF WFNAME TO fcd-filename-address.
29           MOVE WFNAME(1:fcd-name-length) TO F-FNAME.
30
31           MOVE SPACE TO F-OP-NAME.
32           IF OP-CODE >= X'FA00' AND OP-CODE <= X'FA08' THEN
33                 MOVE "OPEN" TO F-OP-NAME
34           ELSE
35             IF (OP-CODE >= X'FAE1' AND OP-CODE <= X'FAE6') OR
36                OP-CODE = X'FAF3' THEN
37                MOVE "WRITE" TO F-OP-NAME
38             ELSE
39               IF OP-CODE >= X'FA80' AND OP-CODE <= X'FA86' THEN
40                  MOVE "CLOSE" TO F-OP-NAME
41               END-IF
42             END-IF
```

```
43              END-IF.
44
45              IF F-OP-NAME <> SPACE THEN
46                 WRITE TRACE-REC
47              END-IF.
48              CLOSE FHTRACE.
49              CALL 'EXTFH' USING OP-CODE, WK-FCD.
50              GOBACK.
```

01 行目	カスタム化したファイルハンドラはユーザプログラムが CANCEL される前に CANCEL されるのを防ぐべく NOCANCEL 指令を指定してコンパイルする必要があります。
24 行目	ファイルハンドラから入出力処理が発生する場合は RETURN-CODE に 1 を指定します。
31 - 47 行目	OPEN 文、WRITE 文、CLOSE 文から発生する操作コードが渡ってきた場合に限り COBOL 文及び処理日時をトレースに記録します。
49 行目	トレースへの書き込み処理が完了したら実際の処理を Micro Focus File Handler に委譲します。

(2) 用意したファイルハンドラを呼び出し可能な共有オブジェクト形式にコンパイル

```
$ cob -z FHDEMO.cbl
$
```

(3) 4-2 で使用した例題プログラムに CALLFH(WKFH) 指令を指定して入出力に関する COBOL 文の処理は本例で用意したファイルハンドラを使うよう構成してコンパイル

```
$ head -2 fcdparmdemo.cbl
      $SET FCDREG CALLFH(FHDEMO)
       ENVIRONMENT DIVISION.
$ cob -u fcdparmdemo.cbl
$
```

(4) (3) で再コンパイルしたプログラムを実行

```
$ cobrun fcdparmdemo.gnt
$ ls -l DEMOSEQ*
-rw-rw-r-- 1 yoshihiro yoshihiro 20  2 月 16 18:33 2015 DEMOSEQ1.dat
-rw-rw-r-- 1 yoshihiro yoshihiro 20  2 月 16 18:33 2015 DEMOSEQ2.dat
$
```

➔ ファイルの入出力要求はファイルハンドラを変更する前と同様に処理されていることがわかります。

(5) 生成されたトレースファイルを確認

```
$ cat FHTRACE.trc
20150217 14095672 OPEN       DEMOSEQ1.dat
20150217 14095672 OPEN       DEMOSEQ2.dat
20150217 14095672 WRITE      DEMOSEQ1.dat
20150217 14095672 WRITE      DEMOSEQ2.dat
20150217 14095672 WRITE      DEMOSEQ1.dat
20150217 14095672 WRITE      DEMOSEQ2.dat
20150217 14095672 CLOSE      DEMOSEQ1.dat
20150217 14095672 CLOSE      DEMOSEQ2.dat
$
```

➔ ファイル入出力に関する各 COBOL 文が発行される際に (1) で用意したファイルハンドラが呼ばれていたことがわかります。

5 リレーショナルデータベース

5-1 RDB アクセス手法の概要

リレーショナルデータベース管理システム（RDBMS）は COBOL コンパイラから見てサードパーティが提供するミドルウェアとなります。そのため、データベースを操作するには前章までに見てきたファイルアクセスと異なり、RDBMS や関連するソフトウェアが提供するアクセスメソッドをアプリケーション内から利用する必要があります。これらが提供するライブラリを直接 COBOL プログラムより呼び出してデータベース操作することも可能ですが、複雑なコーディングを要するため一般的ではありません。COBOL には従来より埋め込み SQL 文というという抽象的なコーディングでデータベース操作するための技術が提供されてきました。この技術を使えば、複雑なライブラリコール等のロジックをコーディングすることなく高水準言語の特長を活かしたまま開発が可能です。埋め込み SQL 文を使う場合、COBOL プログラムソース中には EXEC SQL と END-EXEC で囲った中に直接 SQL 文を記述します。それら埋め込み SQL 文はコンパイルの前にプリコンパイルというステップにて、実際のライブラリやルーチンの CALL 等に置き換えられます。続いてその展開されたソースがコンパイラにかけられます。Visual COBOL はこのプリコンパイルのフェーズを内部的に処理しシングルステップでコンパイルする手段も提供しています。

本章の以降の各項では、この埋め込み SQL 文を使った RDB アクセスを中心に代表的なアクセス手段を紹介いたします。下表はこれらの手段と主な RDBMS との対応付けをまとめた表になります。

	Oracle Database	Microsoft SQL Server	IBM DB2	PostgreSQL	MySQL
Pro*COBOL (※1)	○	n/a	n/a	n/a	n/a
DB2 ECM (※2)	n/a	n/a	○	n/a	n/a
OpenESQL ODBC (※3)	○	○	○	○	○
OpenESQL ADO.NET (※3)	○	○	○	○	○
OpenESQL JDBC (※3)	○	○	○	○	○

（※１） Pro*COBOL は Oracle 社の製品です。製品バージョン等との具体的な対応付けは Oracle 社のドキュメント等をご参照してください。

（※２） EB2 ECM は DB2 のプリプロセッサの機能を内部的に利用します。保証対象の製品とバージョン等との関連付けは IBM 社側の Certification に依存します。

（※３） OpenESQL が指定するそれぞれの規約に準拠したドライバを利用する必要があります。また、その規約に準拠したドライバを使うのであれば、表中にない RDBMS とも連携できる可能性があります。詳しくは Visual COBOL 製品マニュアルをご参照してください。

5-2 Oracle Pro*COBOL

Oracle データベースクライアントは、COBOL アプリケーションから Oracle データベースにアクセスするためのプログラミングツールとして Pro*COBOL というプリコンパイラを搭載しています。このプリコンパイラは Micro Focus 製品も意識して設計されており、埋め込み SQL 文を含んだソースを Micro Focus 製品向けの COBOL ソースへプリコンパイルします。Pro*COBOL では以下のような順でモジュールを生成します。

① 埋め込み SQL 文を含んだプログラムをコーディング
② コーディングしたプログラムを Oracle が提供するプリコンパイラでプリコンパイル
③ Visual COBOL でコンパイル、リンク

Visual COBOL は COBSQL という上の②と③をシングルステップで処理する機能を備えます。埋め込み SQL 文が入った状態のソースコードを同機能を有効にしてコンパイルすると、内部的に Oracle のプリコンパイラを呼び出します。更に、Visual COBOL でデバッグをする際は、プリコンパイルされたソースではなく、埋め込み SQL 文が入った状態のロジカルなソースをベースにデバッガを走らすことができます。また Pro*COBOL で開発する際は、プリコンパイル前後のソースを管理する必要がありますが、COBSQL を使えば、埋め込み SQL 文が入ったソースで一元管理ができます。

Pro*COBOL を使った開発例は以下の別文書にて公開しています。Oracle のドキュメント等と併せて参考にしてください。

https://www.amc.rocketsoftware.co.jp/pdf/products/COBOL/verification-report/visualcobol/white_paper_067.pdf (Windows 編)

https://www.amc.rocketsoftware.co.jp/pdf/products/COBOL/verification-report/visualcobol/white_paper_066.pdf（UNIX/Linux 編）

5-3 DB2 ECM

IBM DB2 のクライアント製品は、DB2 サーバ上のデータベースを操作する COBOL アプリケーションを開発するためのプリコンパイラを搭載しています。このプリコンパイラは Micro Focus 製品も意識して設計されており、埋め込み SQL 文を含んだソースを Micro Focus 製品向けの COBOL ソースへプリコンパイルします。同機能を利用する場合、以下のような順でモジュールを生成します。

① 埋め込み SQL 文を含んだプログラムをコーディング
② コーディングしたプログラムを DB2 が提供するプリコンパイラでプリコンパイル
③ SQL 文をターゲットのデータベースへバインド(②と同時に処理することも可能)
④ Visual COBOL でコンパイル、リンク

Visual COBOL は上の②、③、④と同等の処理をシングルステップで処理する DB2 ECM(External Compiler Module)という機能を備えます。DB2 ECM 機能を有効にして埋め込み SQL を含むソースをコンパイルすると、内部的に埋め込み SQL 文を適切な DB2 の機能への呼び出し処理に変換した上でコンパイラにかけます。DB2 のプリコンパラで利用できるオプションの多くが DB2 ECM でも利用可能となっており、データベース連携部分の処理を要件に合わせてカスタマイズすることが可能です。

当該機能を利用すれば、プリコンパイル後の複雑なソースではなく、埋め込み SQL 文を含んだロジカルなソースを使ってデバッガを走らせることができます。またプリコンパイラを使う場合は、プリコンパイル前後のソースを管理する必要がありますが、DB2 ECM を使えば、プログラマが実際にメンテナンスする埋め込み SQL 文が入ったソースで一元管理ができます。

以下に Visual COBOL が DB2 ECM 用に提供するサンプルを本書用に簡略化加工したものをコンパイル、デバッグ実行する様子を記します。

1) 利用するプログラムを用意

```
01        $SET DB2(DB=SAMPLE PASS=xxx.yyy)
02         WORKING-STORAGE SECTION.
03         EXEC SQL INCLUDE SQLCA END-EXEC
04         01  DCLSTAFF.
05            10 STAFF-ID                    PIC S9(04)  COMP.
06            10 STAFF-NAME.
07              49 STAFF-NAME-LEN             PIC S9(04)  COMP.
08              49 STAFF-NAME-DAT             PIC X(9).
09            10 STAFF-DEPT                  PIC S9(04)  COMP.
10         01  DCLSTAFF-NULL.
11            10 STAFF-NAME-NULL              PIC S9(04)  COMP.
12         PROCEDURE DIVISION.
13         MAIN-PARA.
14            EXEC SQL
15                 CONNECT TO SAMPLE USER xxx USING yyy
16            END-EXEC.
17            EXEC SQL
18               DECLARE CSR1 CURSOR FOR SELECT
19                 A.ID ,A.NAME
20               FROM STAFF A
21               WHERE ( A.DEPT = :STAFF-DEPT )
22               ORDER BY A.NAME
23             END-EXEC.
24             MOVE 51        TO  STAFF-DEPT.
25             EXEC SQL OPEN CSR1 END-EXEC.
26
27             DISPLAY "EMPLOYEES IN DEPARTMENT# 51".
28             PERFORM UNTIL SQLCODE < 0 OR SQLCODE = +100
29               MOVE SPACES            TO STAFF-NAME-DAT
30               EXEC SQL
31                  FETCH CSR1 INTO
32                       :STAFF-ID ,:STAFF-NAME:STAFF-NAME-NULL
33               END-EXEC
34               DISPLAY  " "
```

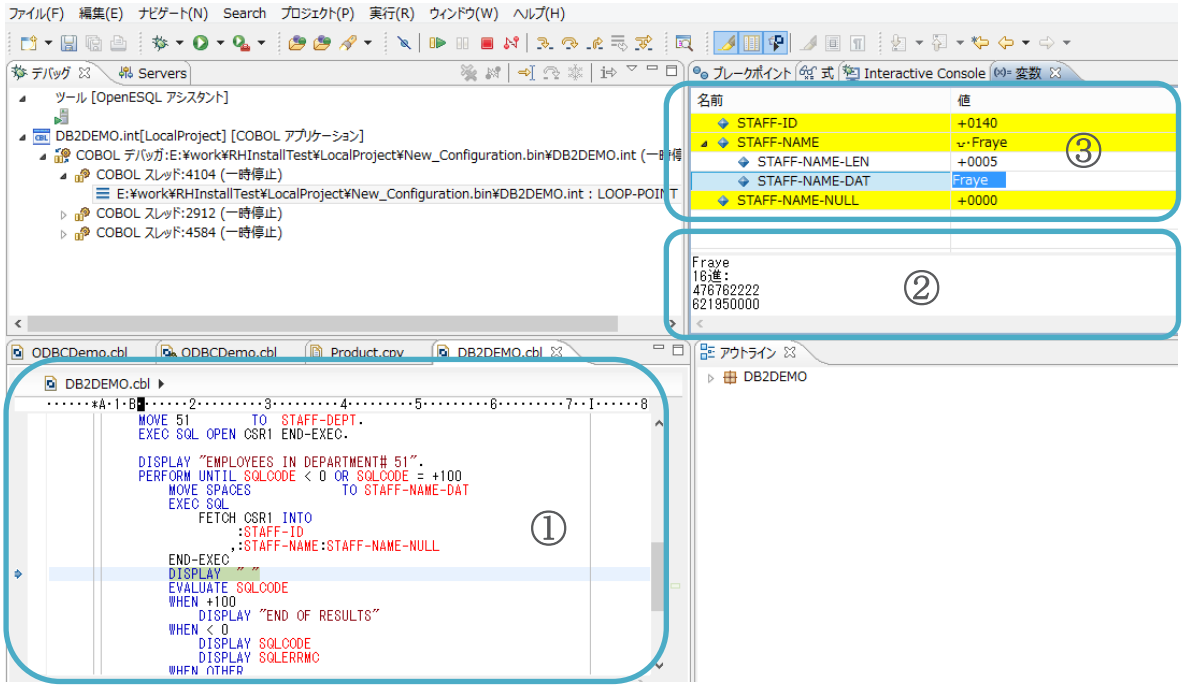
```
35              EVALUATE SQLCODE
36              WHEN +100
37                 DISPLAY "END OF RESULTS"
38              WHEN < 0
39                 DISPLAY SQLCODE
40                 DISPLAY SQLERRMC
41              WHEN OTHER
42                 DISPLAY "ID    " STAFF-ID       WITH NO ADVANCING
43                 DISPLAY " NAME " STAFF-NAME-DAT
44              END-EVALUATE
45           END-PERFORM.
46           EXEC SQL CLOSE CSR1 END-EXEC.
47
48           EXEC SQL CONNECT RESET END-EXEC.
49           DISPLAY "STATIC SELECT SAMPLE COMPLETED".
50           STOP RUN.
```

01 行目　　DB2 コンパイラ指令を指定することで DB2 ECM の機能が有効になります。
01, 15 行目　　このコードをお試しになる場合は、接続の部分をご自身の環境に合った内容に修正してご利用してください。

2)　コンパイル

3)　デバッグ実行(ここでは Eclipse IDE を用いた例を示します。)

デバッグ画面：

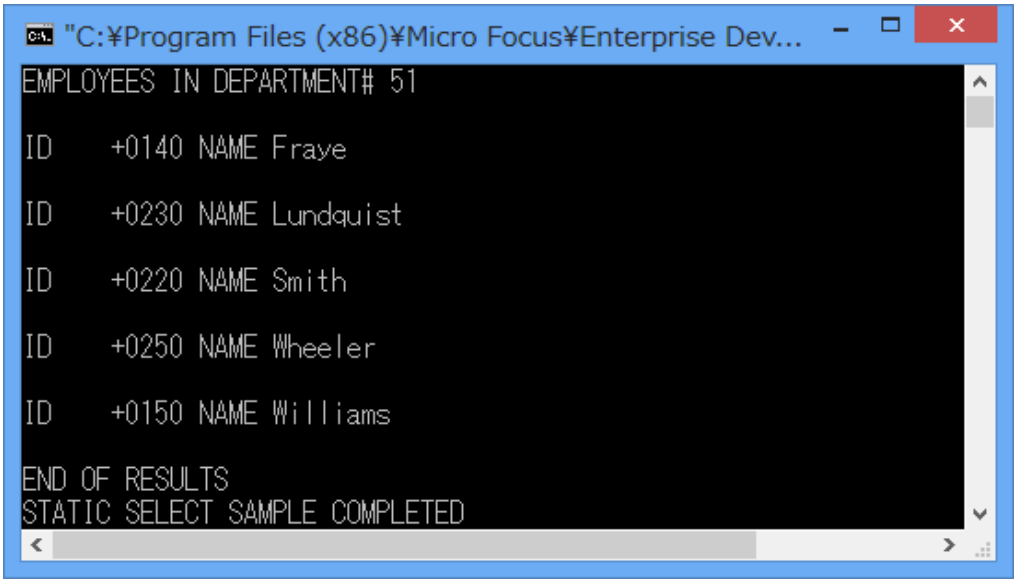


① ステップ単位で処理を進めることができます。ブレークポイントを指定して任意のステップまで処理を自動で進めることもできます。また、そのブレークポイントも例えばループカウンタが「５」になった時に止めるようにする等、条件を指定することも可能です。

② COBOL の変数はもちろんのことホスト変数についても格納されている値をモニタすることが可能です。テキスト表示だけではなく、バイナリ表示、16 進表示にすることも可能です。

③ 任意の変数をポイントすると変数ビューを 16 進表示に設定しなくてもその変数を 16 進表示で確認することができます。

実行結果：



5-4 OpenESQL ODBC

OpenESQL はユビキタスなデータベースアクセスを提供することをコンセプトに開発された Visual COBOL が提供するプリプロセッサです。5-2 及び 5-3 の技法はいずれも Micro Focus 製品を意識したプリコンパイラを使用できる環境上で作業することを前提としていましたが、本機能はそのような制約を伴いません。

Pro*COBOL や DB2 のプリコンパイルと同様に OpenESQL を使ってプログラムを構築する際も EXEC SQL と END-EXEC で囲った中に SQL 文を記述します。OpenESQL は ANSI 標準で定められた SQL 文を解釈するよう設計されています。そのため、他環境からの移行を考えた場合、移行元の COBOL プログラムが連携する RDBMS 固有の表現を使わない埋め込み SQL 文でコーディングされていれば、接続部分等環境に依存した部分への対処をするだけでスムーズに移行できる可能性があります。

OpenESQL はパフォーマンスの調整、トレース採取、OpenESQL に解釈させる SQL 文のレベル、コンパイル時の接続検査等、多岐に及ぶオプションを用意しています。これらについては最も汎用的な値が規定値として指定されていますが、これらを環境や要件に合わせて構成することで更に目的に沿ったモジュールを生成することができます。

OpenESQL は、ODBC、ADO.NET、JDBC を経由して RDBMS へ接続することが可能です。本項ではこのうち ODBC を経由した接続パターンについて解説します。

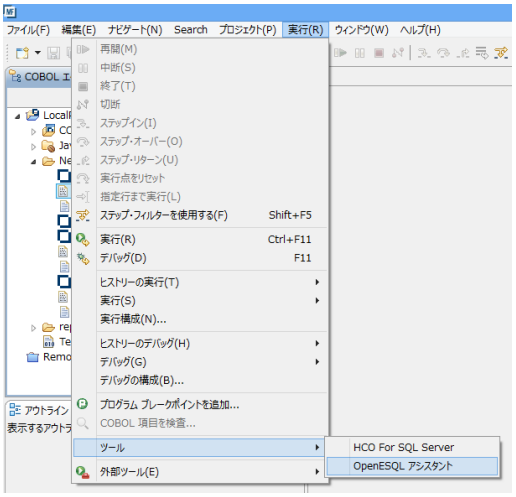
ODBC(**O**pen **D**ata**B**ase **C**onnectivity) は 1992 年、RDBMS に依存しないデータベースアクセスメソッドを提供することを目的に SQL Access Group により開発された技術です。ODBC では、アプリケーションと RDBMS の間にドライバを介在させ、アプリケーションから渡される命令を RDBMS が解釈可能なかたちに変換して RDBMS に届けます。即ち、アプリケーションは ODBC 命令を ODBC ドライバへ発行し、RDBMS は応答をアプリケーションではなく ODBC ドライバに返答します。そのため、この仕組みを利用するには RDBMS、アプリケーションともに ODBC の規約に準ずる必要があります。本稿執筆時点で最新の Visual COBOL 2.2J に搭載された OpenESQL は ODBC 3.0 に準じた ODBC ドライバとの連携をサポートしています。

Visual COBOL には OpenESQL Assistant という OpenESQL を使った開発をサポートするためのツールが付属されています。これを利用すれば、データベースと連携する部分はほぼコーディングレスで構築することができます。

以下は OpenESQL を使って ODBC ドライバ経由で Microsoft SQL Server 2012 に接続させる様子を示しています。前述した OpenESQL Assistant を使ってほぼコーディングすることなく構築したプログラムを埋め込み SQL 文が入った状態でデバッグ実行しています。

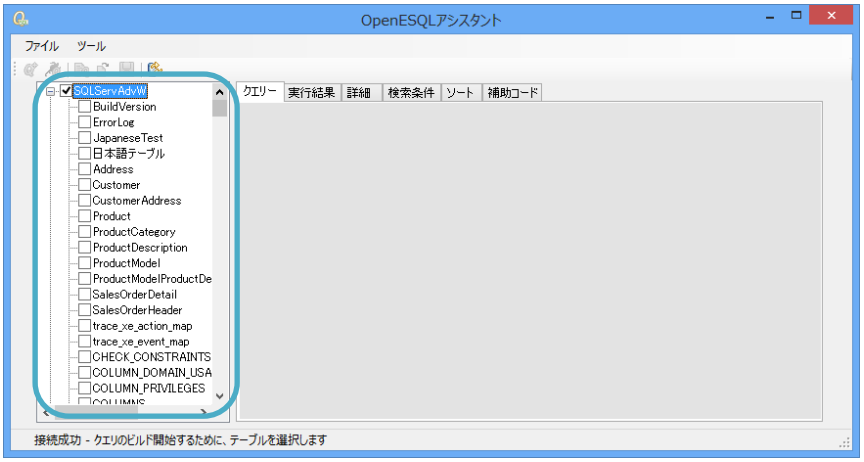
1) Microsoft SQL Server に接続する ODBC データソースを用意
2) Eclipse を起動し、「COBOL プロジェクト」を作成
3) プログラムを追加
4) OpenESQL Assistant を起動

   [実行]メニュー > [ツール] > [OpenESQL アシスタント]



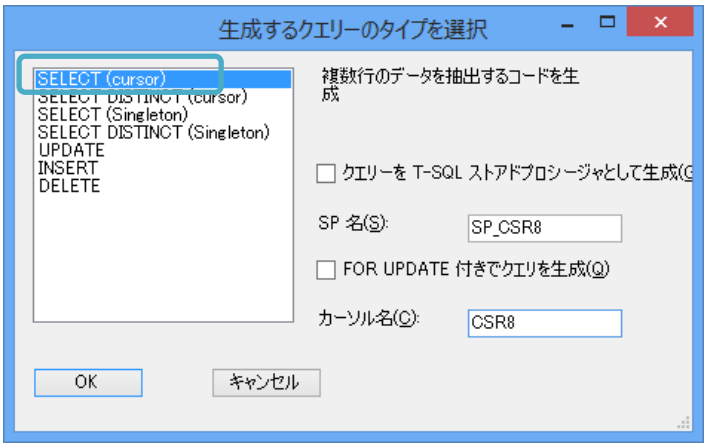
5) 1) で用意した ODBC データソースを指定

   接続先のデータベース中にあるテーブルがリストされます。



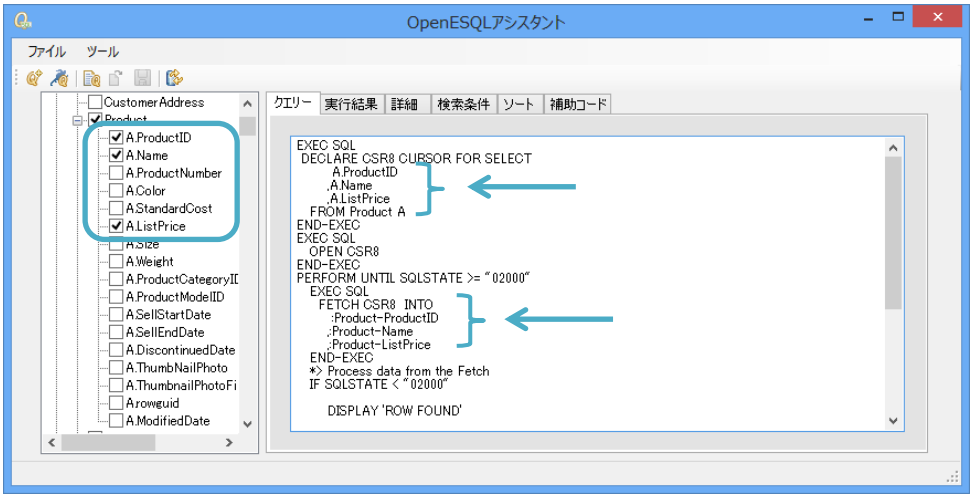
6) 5) で表示されたリストにて連携したいテーブルにチェック

チェック後ポップアップされる画面では「SELECT(cursor)」を選択してみます。



7) 6) で選択したテーブルに定義された列がリストされますので、クエリしたい列にチェック

※ チェックした内容と自動生成された埋め込み SQL が連動し、選択に応じた列がカーソル定義や FETCH 文に自動で追加されます。



8) 汎用 SQL 文を自動で生成

[補助コード] タブにおける [コードの種類] 欄にて「汎用 SQL プログラム」を選択します。

9) 「生成された補助コード」欄に生成されるコードをプログラムに貼り付け
10) ホスト変数を含んだ COPY ファイルを生成

[補助コード] タブにおける [コードの種類] 欄にて「ホスト変数の宣言」を選択します。

11) 10) で生成された COPY ファイルをプロジェクトへインポート
12) COBOL プログラムに自動生成された COPY ファイルに関する記述を追加

編集前：

```
                :
       EXEC SQL BEGIN DECLARE SECTION  END-EXEC.
      *> Put your host variables here if you need to port
      *> to other COBOL compilers
       EXEC SQL END DECLARE SECTION END-EXEC.
                :
```

編集後：

```
                :
       EXEC SQL BEGIN DECLARE SECTION  END-EXEC.
       EXEC SQL INCLUDE PRODUCT END-EXEC.
       EXEC SQL END DECLARE SECTION END-EXEC.
                :
```

※ 本例では連携するテーブル名が「PRODUCT」のため、COPY ファイルもそのテーブル名となっていますが、この部分は実際に利用されるテーブル名に合わせて適宜変更してください。

13) 7) で生成させた [クエリ] タブ中のコードをプログラムに貼り付け

プログラム中の下記コメント部分と差し替えます。

```
                :
      *> Put your program logic/SQL statements here
                :
```

14) 必要に応じてプログラムソースを調整

本例では以下の 3 か所を変更しました。

① 接続部分
下記のようにユーザ名とパスワードを追加しました。「xxx」の部分がユーザ名、「yyy」がパスワードとなります。

```
                    :
        EXEC SQL
            CONNECT TO 'SQLServAdvW' USER 'xxx.yyy'
        END-EXEC
                    :
```

② テーブル名にスキーマ名を追加

本例ではカーソル定義内における SELECT 文中のテーブル名にスキーマ名を付加しました。

```
                    :
        EXEC SQL
          DECLARE CSR8 CURSOR FOR SELECT
              A.ProductID
             ,A.Name
             ,A.ListPrice
          FROM SalesLT.Product A
        END-EXEC
                    :
```

③ FETCH 後、データをコンソール出力させるよう DISPLAY 文を追加

編集前：

```
                    :
          IF SQLSTATE < "02000"

             DISPLAY 'ROW FOUND'

          END-IF
                    :
```

編集後：

```
                    :
    01 Product-Name-DISP              PIC X(50).
    PROCEDURE DIVISION.
                    :
          IF SQLSTATE < "02000"
             MOVE FUNCTION DISPLAY-OF(Product-Name) TO Product-Name-DISP
             DISPLAY 'PRODUCT-NAME     : " Product-Name-DISP
             DISPLAY 'Product-ListPrice: " Product-ListPrice
             DISPLAY ' '
          END-IF
                    :
```

本例で用意したプログラムは最終的に下記のようになります[2]。

```
01      $SET SQL(DBMAN=ODBC)  SOURCEFORMAT(VARIABLE)
02       WORKING-STORAGE SECTION.
03          EXEC SQL INCLUDE SQLCA  END-EXEC.
04       01 MFSQLMESSAGETEXT  PIC X(250).
05       01 IDX              PIC X(04)  COMP-5.
06          EXEC SQL BEGIN DECLARE SECTION  END-EXEC.
07          EXEC SQL INCLUDE PRODUCT END-EXEC.
08          EXEC SQL END DECLARE SECTION END-EXEC.
```

[2] 表示の都合上、OpenESQL Assistant が生成したコードから多少体裁は整えています。

```
09        01 Product-Name-DISP                PIC X(50).
10        PROCEDURE DIVISION.
11           EXEC SQL
12              WHENEVER SQLERROR perform OpenESQL-Error
13           END-EXEC
14           EXEC SQL
15              CONNECT TO 'SQLServAdvW' USER 'xxx.yyy'
16           END-EXEC
17           EXEC SQL
18              DECLARE CSR8 CURSOR FOR SELECT
19                   A.ProductID
20                  ,A.Name
21                  ,A.ListPrice
22              FROM SalesLT.Product A
23           END-EXEC
24           EXEC SQL  OPEN CSR8 END-EXEC
25           PERFORM UNTIL SQLSTATE >= "02000"
26              EXEC SQL
27                   FETCH CSR8  INTO
28                   :Product-ProductID
29                  ,:Product-Name
30                  ,:Product-ListPrice
31              END-EXEC
32              IF SQLSTATE < "02000"
33                 MOVE FUNCTION DISPLAY-OF(Product-Name) TO Product-Name-DISP
34                 DISPLAY 'PRODUCT-NAME     : ' Product-Name-DISP
35                 DISPLAY 'Product-ListPrice: ' Product-ListPrice
36                 DISPLAY ' '
37              END-IF
38           END-PERFORM
39           EXEC SQL  CLOSE CSR8 END-EXEC
40           EXEC SQL DISCONNECT CURRENT END-EXEC
41           GOBACK.
42        OpenESQL-Error Section.
43           display "SQL Error = " sqlstate " " sqlcode
44           display MFSQLMESSAGETEXT
45           EXIT.
```

01 行目　　OpenESQL の機能を有効にするには SQL 指令を指定します。本例では ODBC 経由で接続させたいため、OpenESQL のオプション DBMAN には ODBC を指定します。

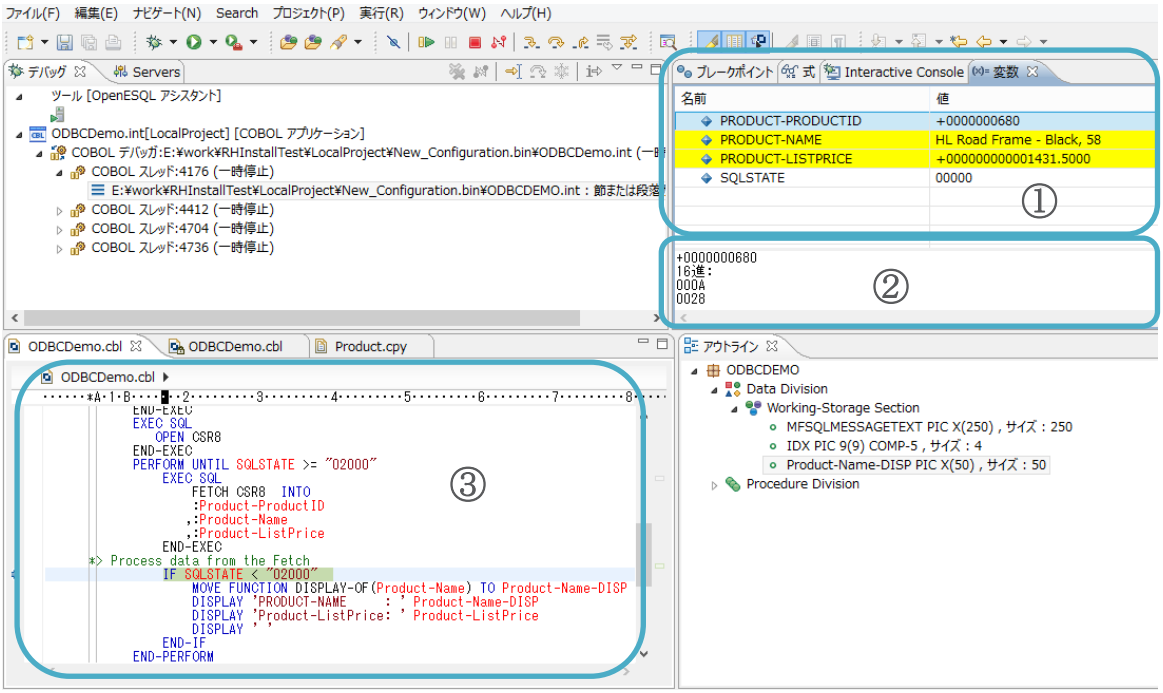
12 行目　　アプリケーションが何等かのデータベース操作にて SQLCODE がマイナス値になるような状態に陥った場合、「SQLERROR」を指定した WHENEVER 文に処理が移ります。

33 行目　　UCS-2 の形式で受け取ったホスト変数中のデータをその環境のロケールに合ったコードに変換しています。（本例は Windows 環境下で処理しているため、SJIS に変換されます。）

15）コンパイル

16）Eclipse IDE でデバッグ実行

デバッグ画面：

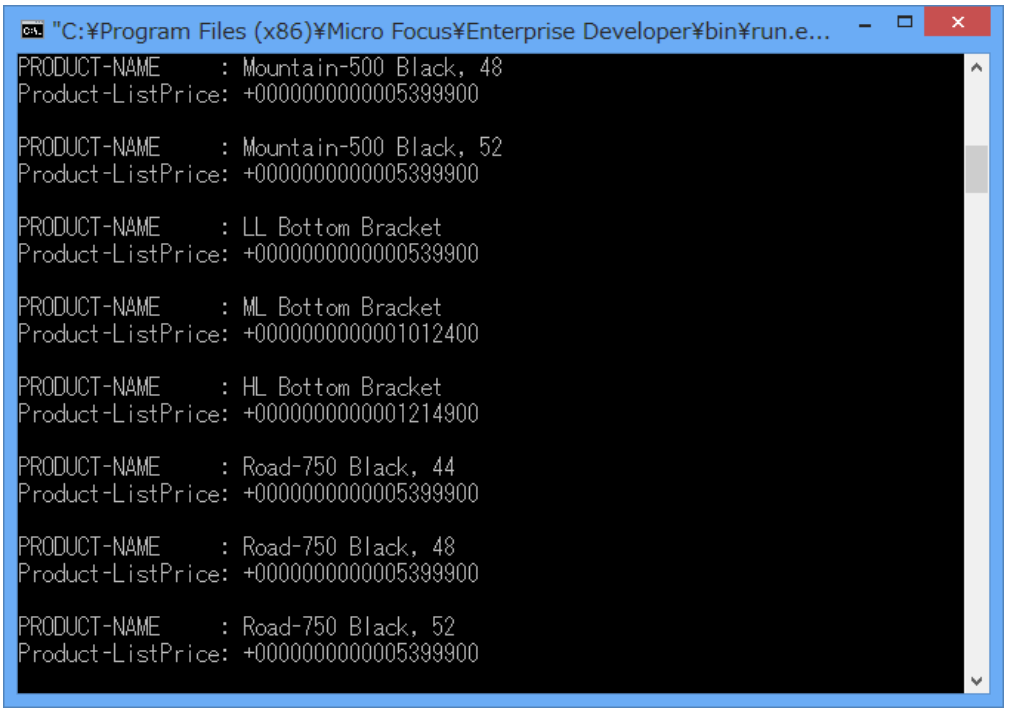


① ステップ単位で処理を進めることができます。ブレークポイントを指定して任意のステップまで処理を自動で進めることもできます。また、そのブレークポイントも例えばループカウンタが「 5 」になった時に止めるようにする等、条件を指定することも可能です。

② COBOL の変数はもちろんのことホスト変数についても格納されている値をモニタすることが可能です。PRODUCT-NAME には UCS-2 でエンコードされたデータが格納されていますが、この変数ビューではテキスト形式に変換して表示します。また、本ビューはテキスト表示だけではなく、バイナリ表示、16 進表示にすることも可能です。

③ 任意の変数をポイントすると変数ビューを 16 進表示に設定しなくてもその変数を 16 進表示で確認することができます。

実行結果：



データベース連携アプリケーションをゼロスクラッチで開発しているにも関わらずほぼ手を入れることなく開発できることを確認しました。これを利用し、例えばファイルアクセスをするアプリケーションをデータベース連携するよう書き換えるようなケースでも新規開発部分はほぼ自動でロジックを構築し入れ替えることができます。また、開発したアプリケーションは Micro Focus が COBOL 用に作りこんだデバッガを使って効率的にデバッギング作業ができます。

5-5 OpenESQL ADO.NET

5-5-1 OpenESQL ADO.NET の基本機能

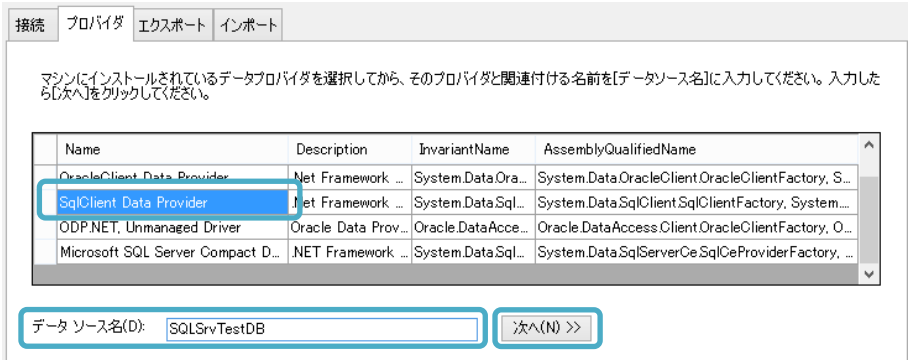
OpenESQL は、ADO.NET を経由して RDBMS へ接続する手段も用意しています。

ADO.NET は .NET Framework プログラマ向けにデータアクセスサービスを提供するソフトウェアコンポーネントです。同コンポーネントは .NET のクラスライブラリ形式で提供されます。Visual COBOL が備える .NET マネージド機能は .NET のクラスライブラリを C#、VB.NET のような他の .NET 言語が利用するのと同様に COBOL においても利用を可能とします。この場合、.NET アプリケーション開発シーンにおける利便性を意識して Micro Focus が拡張した .NET 用の COBOL 方言を使って表現します。C# 等の開発に慣れ親しんだ開発者であればこの技法を使えば言語間の壁をあまり感じることなく、COBOL で ADO.NET を介したデータアクセスを表現できるはずです。しかし、アプリケーションの移行を考えた場合、特に移行元のアプリケーションが埋め込み SQL 文を使ってデータベースアクセスを表現している場合等では、埋め込み SQL 文を維持したまま .NET 環境に移行したいところです。
OpenESQL の ADO.NET 連携機能を利用すれば埋め込み SQL 文が入ったソースを ADO.NET 経由で RDBMS にアクセスする .NET マネージドコードにコンパイルできます。

本稿執筆時点で最新の Visual COBOL 2.2J に搭載された OpenESQL は CLR2 及び CLR 4 ADO.NET data providers との連携をサポートとしています。プログラム記述する埋め込み SQL 文や OpenESQL のオプションは OpenESQL として基本的に共通です[3]。従いまして、Native 環境で OpenESQL ODBC を使って開発していたモジュールを 接続部分等環境に依存した部分のみ変更した上で .NET 環境でリコンパイルするとそのまま動く可能性があります。

5-4 で紹介した OpenESQL Assistant は ADO.NET との連携パターンでもご利用いただけます。しかし、ここでは 5-4 で利用したプログラムの接続部分のみを変更して同プログラムより .NET アプリケーションを生成し SQL Server 2012 上のデータを操作するようすを示します。以下は、開発/デバッグに Visual Studio 2013 IDE を利用した例を示しています。

1) Visual COBOL インストール時に同時にインストールされる付属ツール ADO.NET Connection Editor を起動
2) SQL Server 接続用のデータソースを作成
   ① [追加] ボタンを押下
   ② プロバイダに「SqlClient Data Provider」を選択、データソース名を指定し、[次へ] ボタンを押下

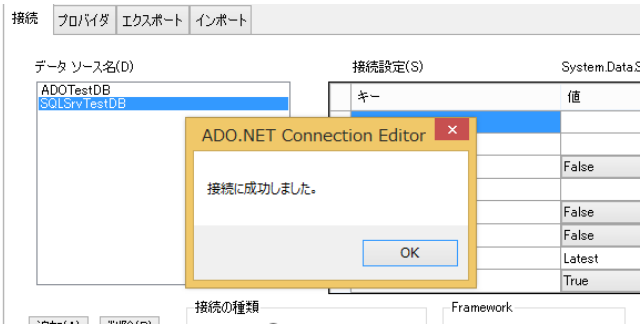


   ③ Data Source, User ID, Password 等接続に必要な情報を指定

[3]データアクセス仕法の違いから、ADO.NET でのみ利用可能等、一部共通でないものもあります。詳細については製品マニュアルをご参照ください。

④ [テスト接続] ボタンを押下し、正しく設定できていることを確認

正しく設定できていれば下図のようなポップアップが出力されます：



3) Visual Studio を起動し、Managed カテゴリ下の「コンソールアプリケーション」プロジェクトを作成
4) 前項で使用したプログラム及び COPY ファイルをプロジェクトへ追加

ソリューションエクスプローラにてプロジェクトを右クリックし、
[追加] > [既存の項目]
を選択し、対象のファイルを追加します。

5) OpenESQL のオプションを ADO.NET 用に変更

OpenESQL オプション DBMAN で「ADO」を指定します。

編集前：

```
$SET SQL(DBMAN=ODBC)  SOURCEFORMAT(VARIABLE)
 WORKING-STORAGE SECTION.
                    :
```

編集後：

```
$SET SQL(DBMAN=ADO)  SOURCEFORMAT(VARIABLE)
 WORKING-STORAGE SECTION.
                    :
```

6) 接続文字列を 2)で作成したものに差し替え

本例ではデータソース名を「AdventureWorksLT2012」と命名し、データソース中にユーザ名及びパスワードを登録しているため、下記のように指定します[4]。

編集後：

```
                    :
      EXEC SQL
         CONNECT TO 'SQLSrvTestDB'  <-----
      END-EXEC
                    :
```

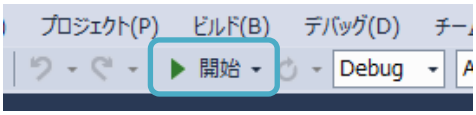
7) コンパイル

[ビルド]メニュー > [ソリューションのリビルド]
を選択します。

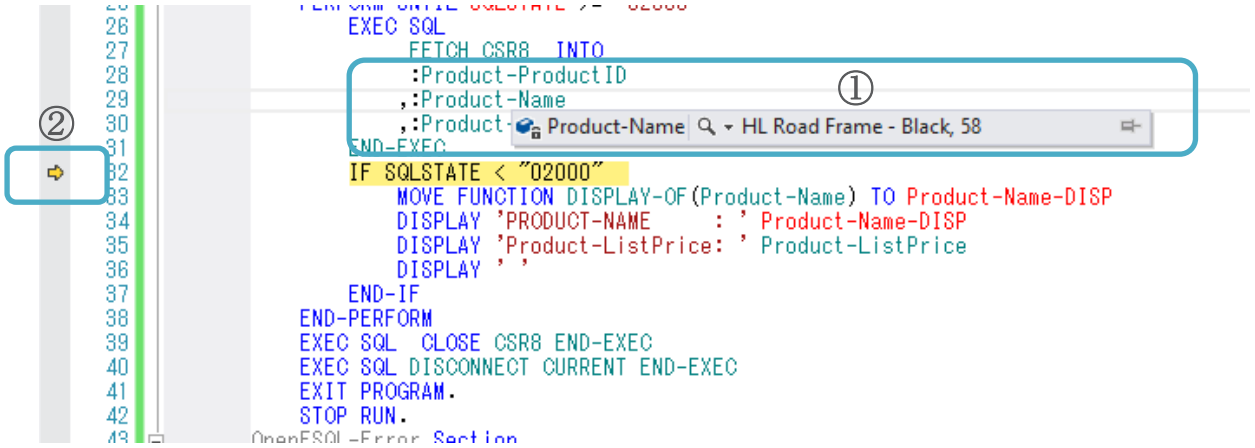
[4] 接続文字列の指定は例示した記法に限らず、様々なコーディングが可能です。コーディングする際、適宜製品マニュアルをご参照してください。

8) デバッグ実行

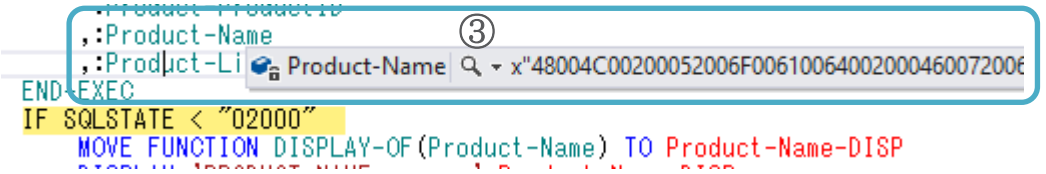
ツールバーより [開始] アイコンをクリックします。



デバッグ画面 1 ：



デバッグ画面 2 ：
16 進数による表示も可能です。

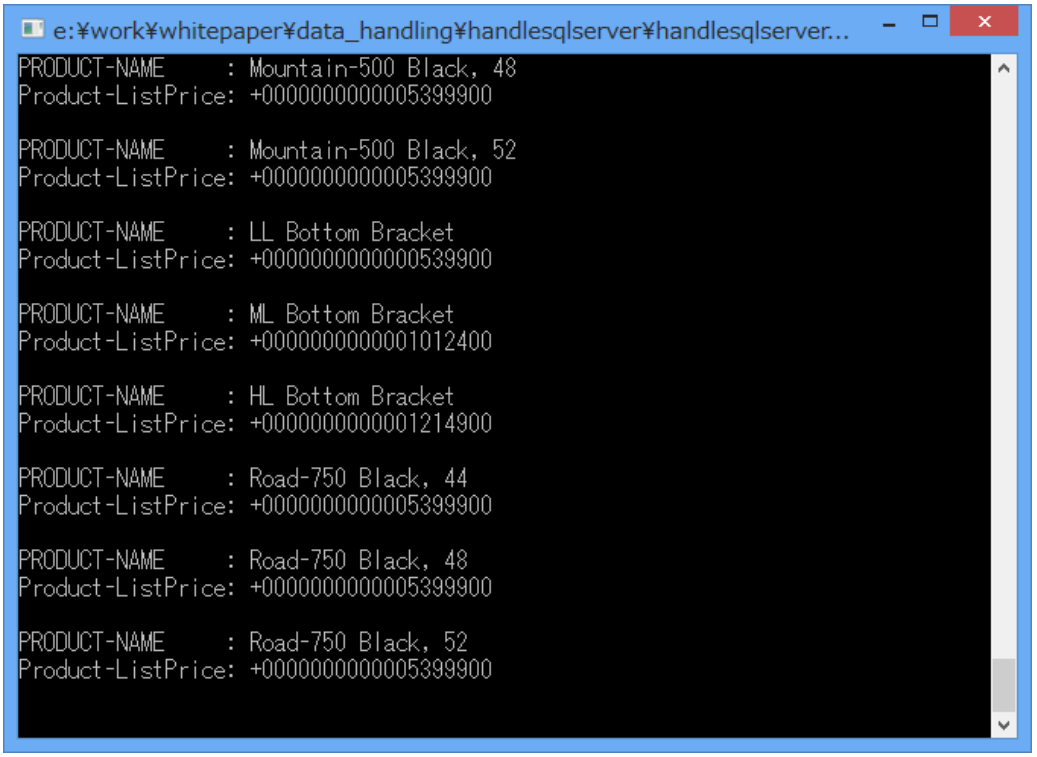


① ステップ単位で処理を進めることができます。ブレークポイントを指定して任意のステップまで処理を自動で進めることもできます。また、そのブレークポイントも例えばループカウンタが「 5 」になった時に止めるようにする等、条件を指定することも可能です。

② ホスト変数を含めた任意の変数にカーソルを hover して変数に格納された値を確認することができます。PRODUCT-NAME には UCS-2 でエンコードされたデータが格納されていますが、テキスト形式での表示が可能です。モニタしたい変数をウォッチウィンドウに固定表示することも可能です。

③ Hover による表示画面及びウォッチウィンドウ内の値は 16 進表示で確認することもできます。

実行結果：



前項で使用したプログラムに対して接続部分を書き換えるだけで .NET 環境で同じように動作させられることを確認しました。これは即ち、前項で解説したように埋め込み SQL 文を使った Open レガシー環境等から .NET への移行作業が驚くほど省力化できることを示唆しています。また、Visual COBOL の Visual Studio 版はデバッガを COBOL 用に作りこんでいるため、C# の開発等で一般的に利用されているデバッガを COBOL の開発でも同様に利用し更なる開発生産性の向上が見込めます。

5-5-2 埋め込み ADO 文を使った非接続型データアクセス

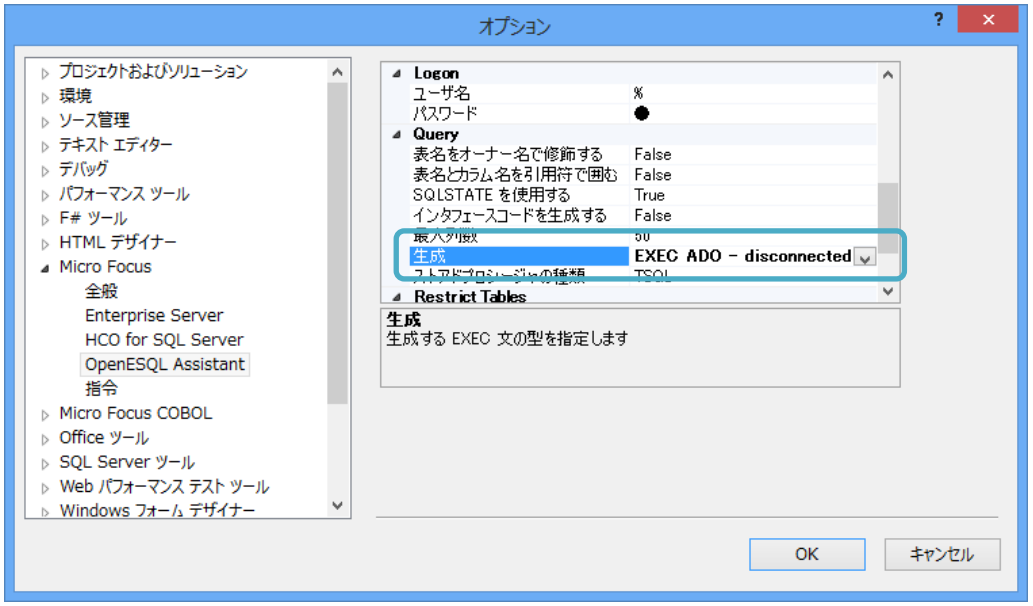
ADO.NET フレームワークは 5-5-1 で紹介したような接続指向型のデータアクセスだけでなく非接続型のデータアクセスもサポートします。接続型のアーキテクチャでは SQL 文によるデータ操作する際は、データベースとの接続が確立されている必要があります。一方、非接続型では、この部分は必須ではありません。この技法で最も一般的に利用される Data Set を例にとりますと、まずはデータベースからデータを取得し、メモリ上にデータをストアします。アプリケーションは以後、そのメモリにストアしたデータに対して SQL 文を発行しデータ操作をします。この間、データベースと接続が確立されている必要はありません。データの更新要求に対しては、まずはメモリ上のデータを更新します。続いて、別のタイミングでメモリ上で更新された内容をデータベースへ反映させます。

Visual COBOL で .NET アプリケーションを開発する場合、前述のように .NET アプリケーション開発シーンにおける利便性を意識して Micro Focus が拡張した .NET 用の COBOL 方言を使い、ADO.NET の技術を利用できます。これを使えば、C# 等でコーディングするのと同じロジックで ADO.NET の非接続型データアクセスも表現できます。C# 等に慣れ親しんだ開発者であればこの方法は受け入れやすい選択肢となるはずです。一方、OpenESQL は埋め込み SQL 文による従来からの COBOL らしいコーディングを維持したいというケースにも対応できるよう埋め込み ADO 文という SQL 文以外の ADO 技術を扱うための拡張機能も利用できます。更にこの埋め込み ADO 文にも OpenESQL Assistant は対応しています。埋め込み SQL 文の例と同様に実際にデータベースを参照して生成されるスニペットを活用して構築することが可能です。

OpenESQL Assistant は以下のようにして非接続の埋め込み ADO 文モードに変更することが可能です。

1) Visual Studio を起動
2) OpenESQL Assistant の生成モードを「EXEC ADO - disconnected」に変更

   [ツール]メニュー > [オプション]
   を選択しポップアップされる画面にて
   [Micro Focus] > [OpenESQL Assistant]
   へナビゲートし表示される画面の「生成」欄を「EXEC ADO - disconnected」に変更します。

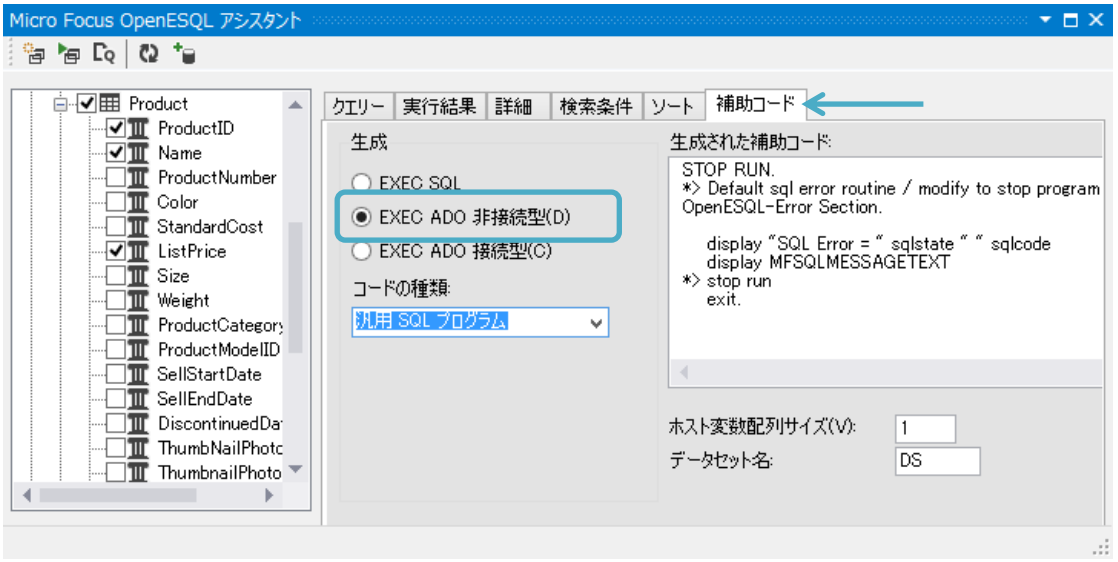


3) OpenESQL Assistant を起動

   [ツール]メニュー > [Micro Focus COBOL] > [OpenESQL アシスタント]
   とナビゲートします。

   EXEC SQL モードと同様に任意のテーブル及び列を選択して、クエリロジックを構築することが可能です。

4) 関連する各種ロジックを補助コードタブで自動生成



5) 必要なロジックを補完しプログラムをコーディング

下記は接続型の例で処理した内容と同等のことを非接続型で実現するよう書き直した例です[5]。ロジックの多くは OpenESQL Assistant により自動生成されたものです。

```
01      $SET SQL(DBMAN=ADO)  SOURCEFORMAT(VARIABLE)
02       WORKING-STORAGE SECTION.
03       EXEC SQL INCLUDE SQLCA END-EXEC.
04       EXEC SQL BEGIN DECLARE SECTION END-EXEC.
05       EXEC SQL INCLUDE Product END-EXEC.
06       EXEC ADO DECLARE Product DATATABLE
07           ( ProductID                INT32       NOT NULL
08            ,Name                     STRING(50)  NOT NULL
09            ,ListPrice                DECIMAL     NOT NULL
10           ) END-EXEC
11       EXEC ADO DECLARE DS DATASET FOR Product END-EXEC.
12       EXEC SQL END DECLARE SECTION END-EXEC .
13       01 Product-Name-DISP                PIC X(50).
14       PROCEDURE DIVISION.
15           EXEC SQL
16               CONNECT TO 'AdventureWorksLT2012'
17           END-EXEC.
18           EXEC ADO INITIALIZE DATASET END-EXEC
19           EXEC ADO PREPARE TO FILL product END-EXEC.
20           EXEC ADO
21               TO FILL product USE
22               SELECT ProductID, Name, ListPrice
23               FROM SalesLT.Product
24           END-EXEC.
25           EXEC ADO FILL DATASET END-EXEC.
26           EXEC SQL DISCONNECT CURRENT END-EXEC.
27           EXEC ADO
28               DECLARE CSR14 DATAROWS FOR
```

[5] 接続型の例で指定したエラーハンドリング等の技法は非接続型のロジックでも利用できますが、サンプルコードではページの都合上敢えて省いています。

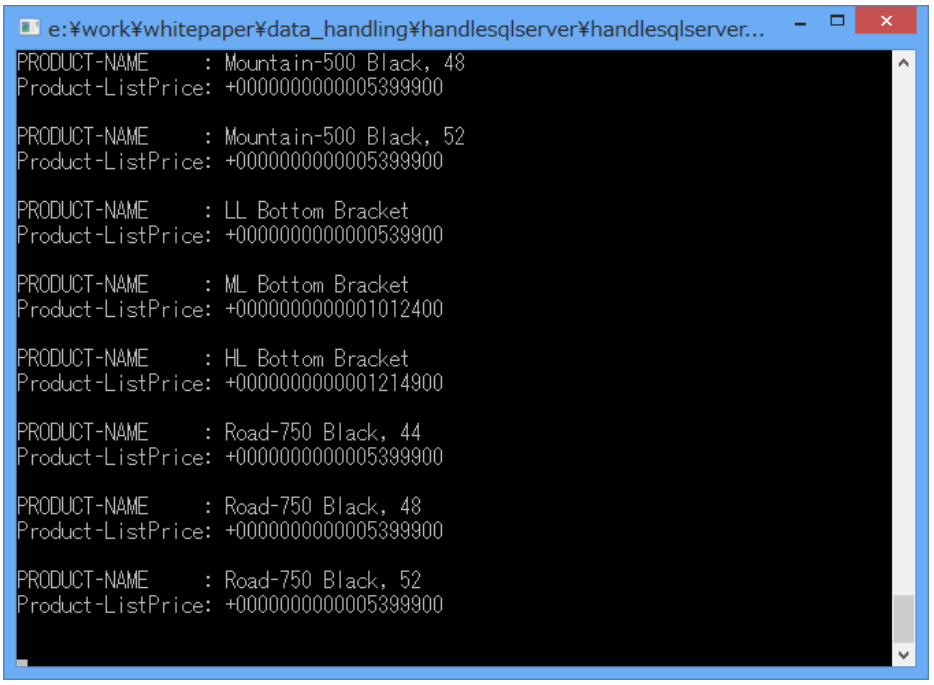
```
29              SELECT ProductID ,Name ,ListPrice
30              FROM Product
31          END-EXEC.
32          EXEC ADO OPEN CSR14 END-EXEC
33          PERFORM UNTIL SQLSTATE >= "02000"
34            EXEC ADO
35                FETCH CSR14  INTO
36                    :Product-ProductID
37                   ,:Product-Name
38                   ,:Product-ListPrice
39            END-EXEC
40            IF SQLSTATE < "02000"
41               MOVE FUNCTION DISPLAY-OF(Product-Name) TO Product-Name-DISP
42               DISPLAY 'PRODUCT-NAME     : ' Product-Name-DISP
43               DISPLAY 'Product-ListPrice: ' Product-ListPrice
44               DISPLAY ' '
45            END-IF
46          END-PERFORM.
47          EXEC ADO CLOSE CSR14 END-EXEC.
48          GOBACK.
```

06 行目	System.Data.DataTable クラスオブジェクトを生成し、列を定義させるのと同等の命令です。 System.Data.DataSet クラスオブジェクトを生成し、Table 要素を追加するのと同等の命令です。
11, 18, 19 行目	System.Data.SqlClient.SqlDataAdapter クラスの Fill メソッドを使って DataSet に充填するのと同等の命令です。
20, 25 行目	メモリ上にデータが転送され、以後はこのデータを操作するため、この時点でデータベースとの接続を切断しています。
26 行名	DataSet に充填されたデータを System.Data.DataRow に転記するのと同等の命令です。ここでは、接続型で記述していた CURSOR – FETCH と近い文法で表現できています。
27, 34 行目	

6) デバッグ実行

ツールバーより [開始] アイコンをクリックします。

サンプルの実行結果：



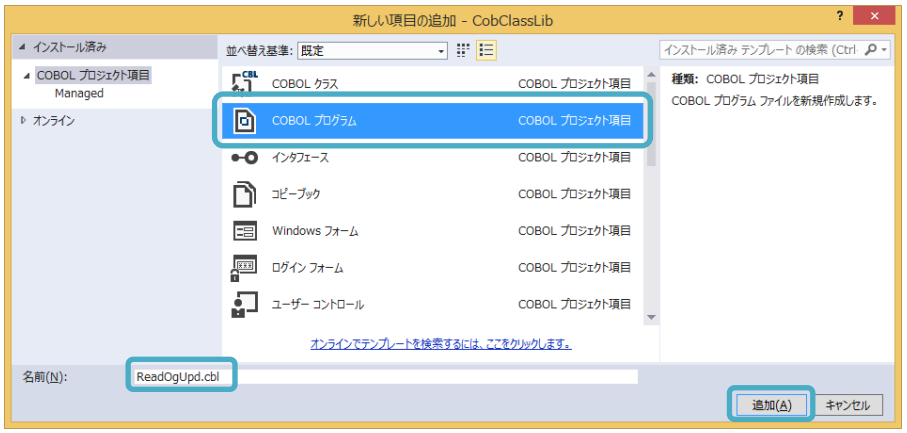
非接続型データアクセスの技術を使う場合でも、C# 等と親和性のある文法ではなく、従来から COBOL で利用されている文法に近い記法で表現できることを確認できました。

5-5-3 OpenESQL ADO を使った他 .NET 言語アプリケーションとの接続の共有

アプリケーションの将来性や使い勝手といったことを考えた場合、新しい技術を取り入れてアプリケーションを進化させたいという要求が挙がるのは必然です。しかし、長年コストをかけてメンテナンスを重ね、既に品質担保ができているビジネスロジック部分に手を入れることは非常に大きなリスクを伴います。Visual COBOL の .NET マネージド機能を利用すれば、COBOL プログラムで書かれたビジネスロジック部分を .NET MSIL コードにリコンパイルし、.NET 環境上で高価値な COBOL 資産を継続して利用することができます。一方、グラフィックな処理部分等、COBOL の優位性が比較的低い部分については C# のようなこれらの分野で一般的に利用されている別言語でリエンジニアリングできればコスト面やメンテナンス性等の観点から得られるメリットは大きいです。このようなかたちで .NET 化してモダナイズするのであれば C# 等と COBOL との連携は必須要件となります。MSIL コードにコンパイルされた COBOL プログラムは C# 等からは COBOL であるということを意識することなく一般的な .NET クラスライブラリを呼び出すのと同じ記法で呼び出せます。.NET 開発シーンを意識して拡張された COBOL 方言ではクラスオブジェクトの参照渡しを C# 等と同じように表現できるため、C# 等と接続を共有したり COBOL も含めたトランザクション管理を C# で行うこともできます。それだけでなく、埋め込み SQL 文や埋め込み ADO 文を使った従来からの COBOL らしい表現で書かれたプログラムについても他の .NET 言語と接続やトランザクションを 共有することが可能です。この機能を活用すれば、RDB 連携を含む COBOL プログラムの .NET 化を一層効率化できる可能性があります。例えば、C# 等で RDB 接続のためのメソッド等を一元化したいといった要件に対しても、既に埋め込み SQL 文を使ってコーディングされ運用中の COBOL プログラムに軽微な修正を加えるだけで応えることができます。

以下では埋め込み SQL 文を使って、他の .NET 言語（本例では C# ）と接続及びトランザクションを共有する様子を記します。ここでは、接続を C# 側から開始し、その接続を使って COBOL 側で SELECT 文及び UPDATE 文を実行します。その UPDATE 文による変更を C# 側で COMMIT/ROLLBACK をして COBOL のトランザクションを管理していること示しています。

1） Visual Studio を起動し [COBOL プロジェクト] > [Managed] 配下の「クラスライブラリ」プロジェクトを作成
2） COBOL プログラムを追加
   ① ソリューションエクスプローラにてプロジェクトを右クリックから
   [追加] > [新しい項目]
   を選択
   ② [COBOL プログラム] を選択し任意のファイル名を指定の上、[追加] ボタンを押下



3） レコードを 1 件読み込み、その後にそのレコードに対して更新をかける埋め込み SQL 文を含む COBOL ロジックを 2) で追加したプログラムへコーディング

本例で用意したプログラムソース：

```
01        $SET SQL(DBMAN=ADO)  SOURCEFORMAT(VARIABLE)
02         program-id. ReadOgUpd as "CobClassLib.ReadOgUpd".
03         DATA DIVISION.
04         WORKING-STORAGE SECTION.
05            EXEC SQL BEGIN DECLARE SECTION END-EXEC.
06         01  DCLtesttbl.
07            03 HV-INTCOL                PIC S9(09)  COMP-5.
08            03 HV-SMALLINTCOL           PIC S9(04)  COMP-5.
09         01 HV-CONN        TYPE System.Data.SqlClient.SqlConnection.
10         01 HV-TRAN        TYPE System.Data.SqlClient.SqlTransaction.
11            EXEC SQL END DECLARE SECTION END-EXEC.
12            EXEC SQL INCLUDE SQLCA  END-EXEC.
```

```
13      01 SMALLINTCOL-DISP             PIC S9(04).
14      PROCEDURE DIVISION USING
15              mySQLConnection AS TYPE System.Data.SqlClient.SqlConnection
16              mySQLTransaction AS TYPE System.Data.SqlClient.SqlTransaction.
17      MAIN-PROC SECTION.
18      MAIN-RTN.
19         SET HV-CONN TO mySQLConnection.
20         SET HV-TRAN TO mySQLTransaction.
21
22         EXEC ADO
23            BIND CONNECTION SQLConnName TO :HV-CONN WITH transaction :HV-TRAN
24         END-EXEC.
25
26         MOVE 2 TO HV-INTCOL.
27         PERFORM SELECT-RTN.
28         DISPLAY "SMALLINTCOL AT COBOL(BEFORE UPDATE): " SMALLINTCOL-DISP.
29
30         COMPUTE HV-SMALLINTCOL = HV-SMALLINTCOL + 1.
31         PERFORM UPDATE-RTN.
32
33         PERFORM SELECT-RTN.
34         DISPLAY "SMALLINTCOL AT COBOL(AFTER UPDATE) : " SMALLINTCOL-DISP.
35
36         EXEC ADO UNBIND CONNECTION SQLConnName END-EXEC.
37         GOBACK.
38
39      SELECT-RTN.
40         EXEC SQL
41            SELECT SMALLINTCOL INTO :HV-SMALLINTCOL
42            FROM dbo.testtbl WHERE INTCOL = :HV-INTCOL
43         END-EXEC.
44         MOVE HV-SMALLINTCOL TO SMALLINTCOL-DISP.
45         EXIT.
46
47      UPDATE-RTN.
48         EXEC SQL
49            UPDATE dbo.testtbl
50               SET SMALLINTCOL = :HV-SMALLINTCOL
51            WHERE INTCOL = :HV-INTCOL
52         END-EXEC.
53         EXIT.
```

09 行目	C# より受け取った接続オブジェクトを保持するためのホスト変数定義です。
10 行目	C# より受け取ったトランザクションオブジェクトを保持するためのホスト変数定義です。
19 行目	C# より受け取った接続オブジェクトを COBOL 側で定義した変数にセットします。
20 行目	C# より受け取ったトランザクションオブジェクトを COBOL 側で定義した変数にセットします。
23 行目	この命令を発行することで OpenESQL にて C# で開始した接続及びトランザクションを利用できるようになります。
36 行目	OpenESQL から接続を解放し再び C# 側でこの接続オブジェクト及びトランザクションオブジェクトを利用できるようにします。

4) COBOL と連携する C# プロジェクトを追加

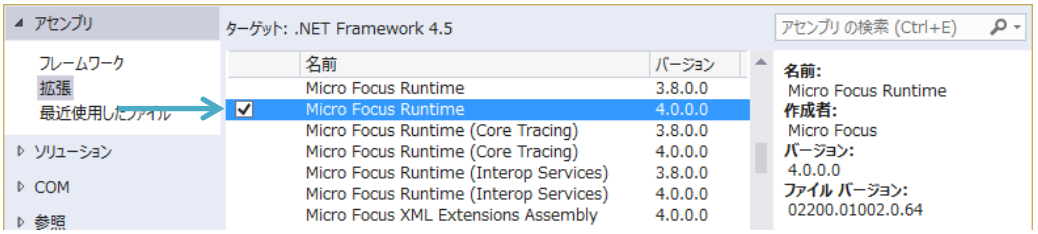
① ソリューションエクスプローラにてソリューションを右クリックし、
[追加] > [新しいプロジェクト]
を選択

② 「Visual C#」ページ中の「コンソールアプリケーション」プロジェクトを選択し、追加

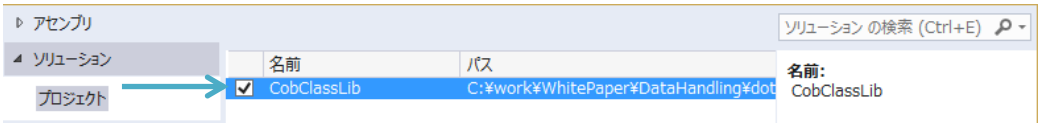
5) Visual COBOL のランタイムライブラリ及び COBOL のプロジェクトを C# プロジェクトからの参照先へ追加

① ソリューションエクスプローラにて [参照設定] を右クリックし、[サービス参照の追加] を選択

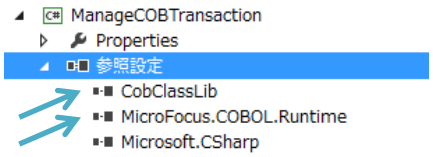
② [アセンブリ] > [拡張]
ページにて、「Micro Focus Runtime」をチェック



③ [ソリューション] > [プロジェクト]
ページにて 1) で作成した COBOL のプロジェクトをチェック



④ [OK] ボタンを押下
チェック入れた項目が参照項目に追加されます：



6) 上で用意した COBOL から生成される .NET クラスライブラリを 2 回呼び、1 回目はトランザクションを取り消し、2 回目はトランザクションを確定させるロジックを C# プロジェクト中のプログラムへコーディング

本例で用意したプログラムソース：

```
01  using System;
02  using System.Collections.Generic;
03  using System.Linq;
04  using System.Text;
05  using System.Threading.Tasks;
06  using System.Data.SqlClient;
07
08  namespace ManageCOBTransaction
09  {
10      class ManageCOBOLTransaction{
11          SqlConnection myConn;
12          SqlTransaction myTran;
13          SqlCommand myComm;
14          SqlDataReader myReader;
15          CobClassLib.ReadOgUpd cobclass;
16
17          static void Main(string[] args){
18              ManageCOBOLTransaction myClass = new ManageCOBOLTransaction();
19                  myClass.handleCOBogSQL();
20          }
21
22          private void handleCOBogSQL(){
23              string constr =
24                  @"Data Source=localhost;Integrated Security=no;" +
25                  @"Initial Catalog=TestDB;User Id=xxx;Password=yyy";
26
27              try {
28                  using (myConn = new SqlConnection(constr)) {
29                      myConn.Open();
30                      using (myTran = myConn.BeginTransaction()) {
31                          myComm = new SqlCommand
32                          ("SELECT SMALLINTCOL FROM dbo.testtbl WHERE INTCOL = 2",
33                                  myConn, myTran);
34                          cobclass = new CobClassLib.ReadOgUpd();
35                          cobclass.ReadOgUpd(ref myConn, ref myTran);
36                          myTran.Rollback();
37                          printRecord();
38                              }
39                      using (myTran = myConn.BeginTransaction()) {
40                          cobclass.ReadOgUpd(ref myConn, ref myTran);
41                          myTran.Commit();
42                          printRecord();
43                      }
44                  }
45              } catch (Exception e) {
```

```
46              Console.WriteLine("Message    : " + e.Message);
47              Console.WriteLine("Stack Trace: " + e.StackTrace);
48          }
49      }
50
51      private void printRecord(){
52          using (myReader = myComm.ExecuteReader()) {
53              while (myReader.Read()) {
54                  Console.WriteLine("SMALLINTCOL AT C#: " + myReader["SMALLINTCOL"]);
55              }
56          }
57      }
58  }
59  }
```

28 行目	RDB との接続を確立させます。
34 - 35 行目	一回目の COBOL の呼び出しです。通常の .NET クラス中のインスタンスメソッドを呼び出すように
	COBOL から生成されたクラスをインスタンス化して、メソッド呼び出しをします。
	COBOL 内で処理された更新を Rollback して取り消します。
36 行目	COBOL が更新したレコードを読み込む SELECT 文を含むメソッドを呼び出し、更新が Rollback された
37 行目	ことを確認します。
	再び、COBOL を呼び出します。
40 行目	COBOL で処理された更新を COMMIT して確定させます。
41 行目	更新確定後に対象のレコードを読み込み正しく反映されていることを確認します。
42 行目	

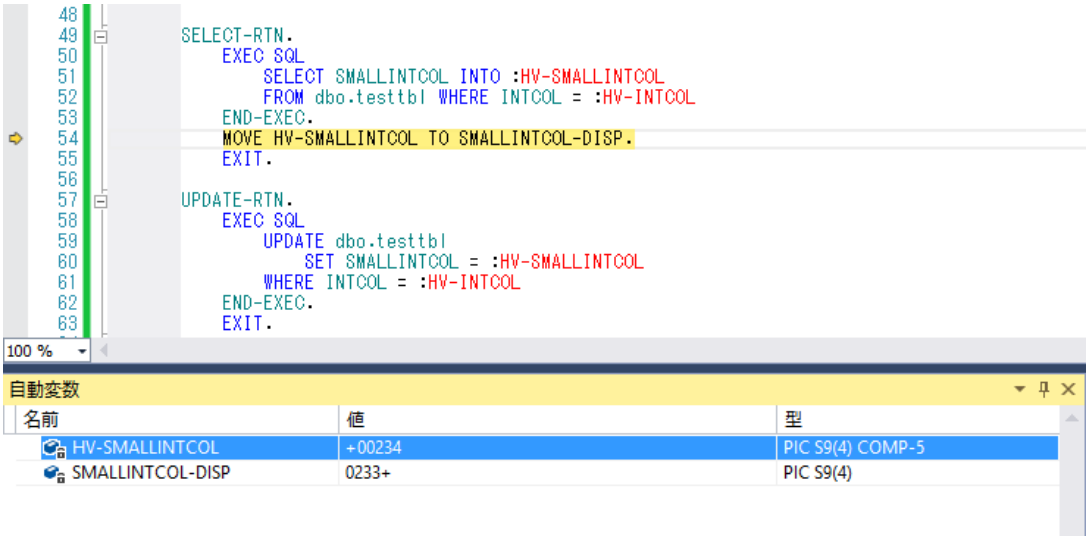
7) C# のプロジェクトをスタートアッププロジェクトに設定

ソリューションエクスプローラにて C# のプロジェクトを右クリックして
[スタートアッププロジェクトに設定]
を選択

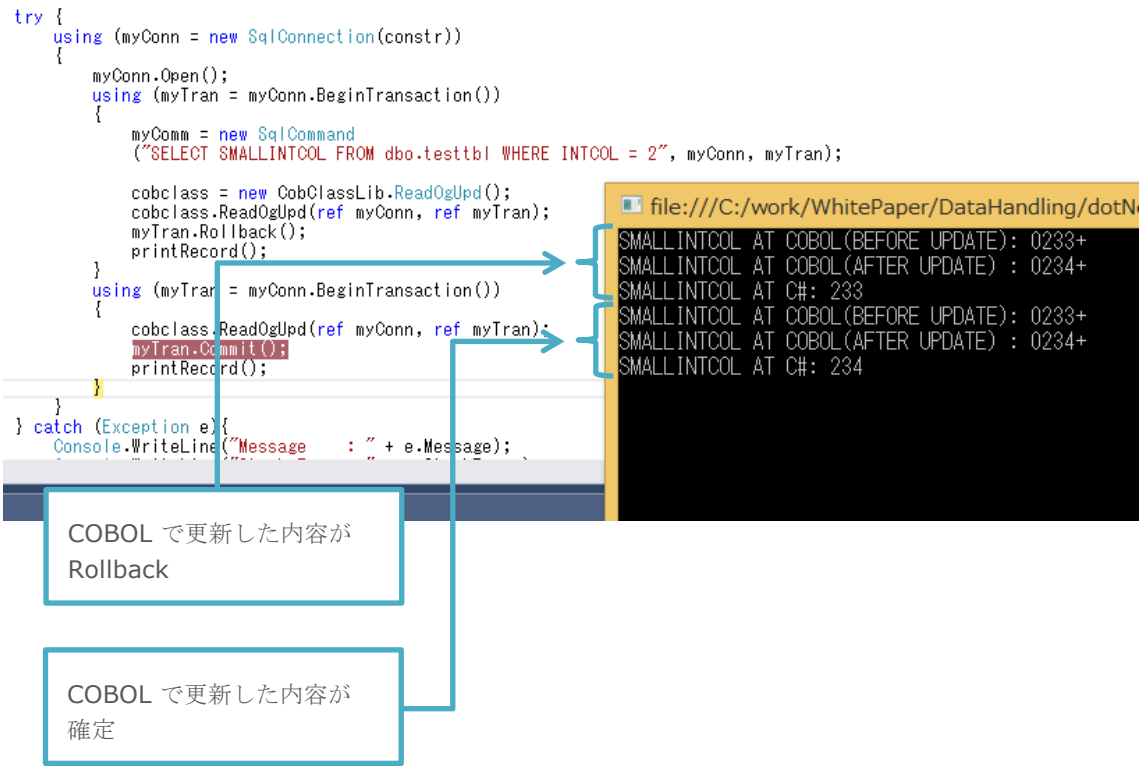
8) 用意したアプリケーションをデバッグ実行

F11 キーを打鍵し、ステップイン実行を開始

デバッグイメージ：



処理結果：


try {
using (myConn = new SqlConnection(constr))
{
myConn.Open();
using (myTran = myConn.BeginTransaction())
{
myComm = new SqlCommand
("SELECT SMALLINTCOL FROM dbo.testtbl WHERE INTCOL = 2", myConn, myTran);
cobclass = new CobClassLib.ReadOgUpd();
cobclass.ReadOgUpd(ref myConn, ref myTran);
myTran.Rollback();
printRecord();
}
using (myTran = myConn.BeginTransaction())
{
cobclass.ReadOgUpd(ref myConn, ref myTran);
myTran.Commit();
printRecord();
}
}
} catch (Exception e){
Console.WriteLine("Message : " + e.Message);
file:///C:/work/WhitePaper/DataHandling/dotN
SMALLINTCOL AT COBOL(BEFORE UPDATE): 0233+
SMALLINTCOL AT COBOL(AFTER UPDATE) : 0234+
SMALLINTCOL AT C#: 233
SMALLINTCOL AT COBOL(BEFORE UPDATE): 0233+
SMALLINTCOL AT COBOL(AFTER UPDATE) : 0234+
SMALLINTCOL AT C#: 234
COBOL で更新した内容が Rollback
COBOL で更新した内容が確定

5-6 OpenESQL JDBC

5-6-1 OpenESQL JDBC の基本機能

OpenESQL は、JDBC を経由して RDBMS へ接続する手段も用意しています。

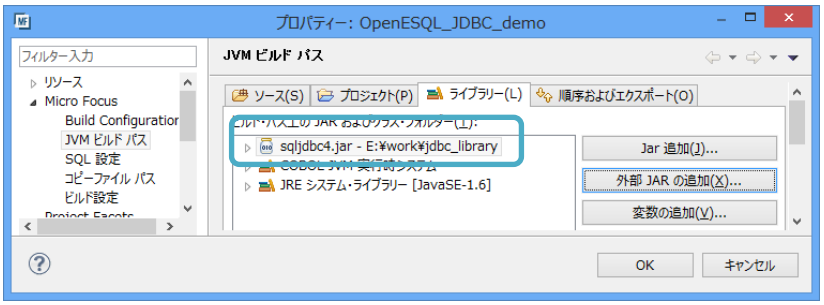
JDBC(**J**ava **D**ata**B**ase **C**onnectivity) は Java の「Write Once, Run Anywhere」のコンセプトをデータアクセスにも拡げた特定の RDBMS に依存しない技術です。Java プログラムは Java SE や Java EE に含まれる JDBC API を通じて SQL 文を発行しデータの取得や操作をします。Visual COBOL が備える COBOL for JVM を使うと COBOL プログラムを Java と同様に javabyte コードにコンパイルし JVM クラスが生成されます。Native コードにコンパイルするのと異なり、プログラムは javabyte コード同士でやりとりが可能なため、JNI のような複雑な連携を考慮することなく Java API を利用できます。Visual COBOL では Micro Focus が JVM アプリケーション開発の簡便性を意識して拡張した COBOL 方言を利用できます。この COBOL 方言を活用すれば Java の開発に慣れ親しんだ開発者であれば抵抗なく COBOL においても JDBC を介したデータアクセスを表現できるはずです。
しかし、アプリケーションの移行を考えた場合、特に移行元のアプリケーションが埋め込み SQL 文を使ってデータベースアクセスを表現している場合等では、埋め込み SQL 文を維持したまま JVM 環境に移行したいところです。OpenESQL の JDBC 連携機能を利用すれば埋め込み SQL 文が入ったソースを JDBC 経由で RDBMS にアクセスする javabyte コードにコンパイルできます。

本稿執筆時点で最新の Visual COBOL 2.2J に搭載された OpenESQL は JDBC version 4 のドライバとの連携をサポートとしています。プログラムに記述する埋め込み SQL 文や OpenESQL のオプションは OpenESQL として基本的に共通です[6]。従いまして、Native 環境で OpenESQL ODBC を使って開発していたモジュールを 接続部分等環境に依存した部分のみ変更した上で JVM 環境でリコンパイルするとそのまま動く可能性があります。

本稿では 5-4 で利用したプログラムの接続部分のみを変更して同プログラムより JVM アプリケーションを生成し SQL Server 2012 上のデータを操作するようすを示します。開発/デバッグには Eclipse IDE を利用しています。

1) Eclipse を起動し、「COBOL JVM プロジェクト」を作成
2) JDBC Driver をビルドパスに追加

   ① COBOL エクスプローラにてプロジェクトを右クリックし「プロパティー」を選択
   ② [Micro Focus] > [JVM ビルドパス] を選択
   ③ [ライブラリー] タブを選択
   ④ [外部 JAR の追加] ボタンを押下



   ここでは「sqljdbc4.jar」を追加しました。

3) 5-4 で使用したプログラム及び COPY ファイルをプロジェクトへインポート

   ① COBOL エクスプローラにて「src」フォルダを右クリックし
   [インポート] > [インポート] を選択
   ② [一般] > [ファイルシステム] を選択
   ③ [参照] ボタンを押下し、エクスプローラから対象のファイルを選択

[6]データアクセス仕法の違いから、JDBC でのみ利用可能等、一部共通でないものもあります。詳細については製品マニュアルをご参照ください。

4） OpenESQL のオプションを JDBC 用に変更
OpenESQL オプション DBMAN で「JDBC」を指定します。

編集前：

```
$SET SQL(DBMAN=ODBC)  SOURCEFORMAT(VARIABLE)
 WORKING-STORAGE SECTION.
                    :
```

編集後：

```
$SET SQL(DBMAN=JDBC)  SOURCEFORMAT(VARIABLE)
 WORKING-STORAGE SECTION.
                    :
```

5） 接続文字列を JDBC 用に編集[7]

編集後：

```
$SET SQL(DBMAN=JDBC)  SOURCEFORMAT(VARIABLE)
$SET CONSTANT driverClass "com.microsoft.sqlserver.jdbc.SQLServerDriver"
$SET CONSTANT databaseURL1 "jdbc:sqlserver://localhost;"
$SET CONSTANT databaseURL2 "databaseName=AdventureWorksLT2012;"
$SET CONSTANT databaseURL3 "user=coboltest;password=password"
                    :
 01 CONNECTIONSTRING PIC X(300) VALUE SPACES.
    EXEC SQL END DECLARE SECTION END-EXEC.
                    :
    MOVE "Driver=" & driverClass &
      ";URL=" & databaseURL1 & databaseURL2 & databaseURL3
      TO CONNECTIONSTRING.
    EXEC SQL
       CONNECT USING :CONNECTIONSTRING
    END-EXEC
                    :
```

① ② ③

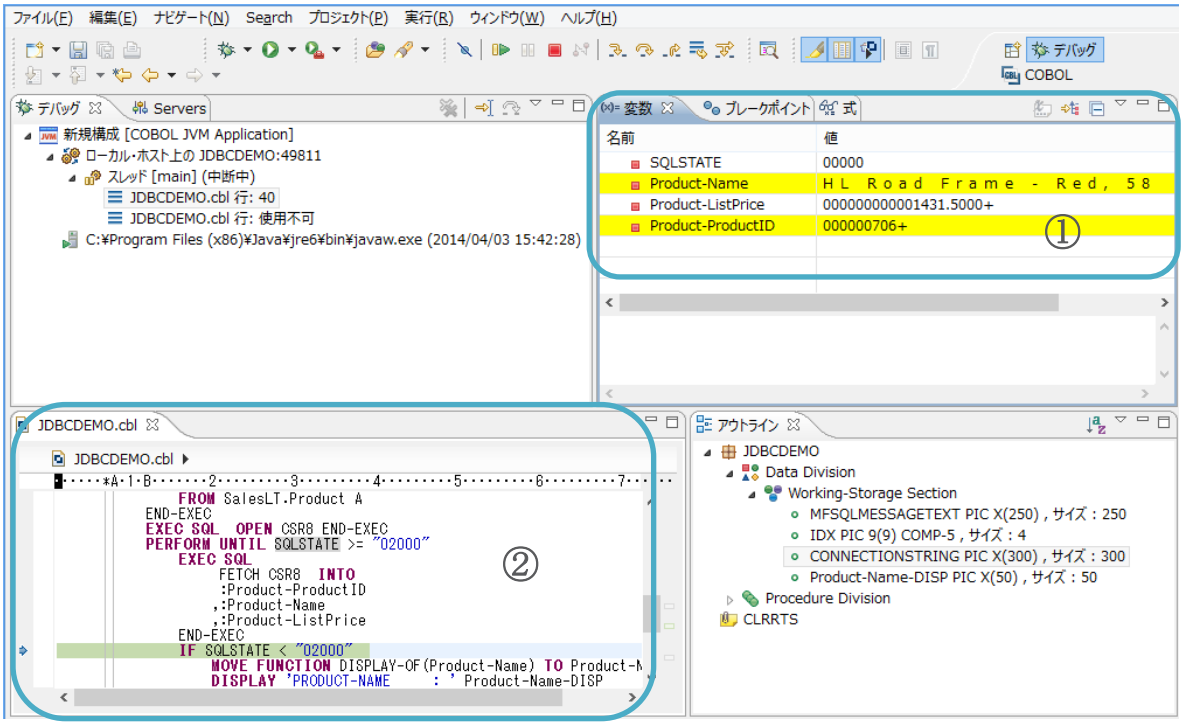
① JDBC ドライバ、DB のホスト名、DB 名、ユーザ/パスワードを定数として定義します。本例ではソース中に表示上の理由から敢えてハードコードしていますが、これを外部ファイルに定義することで、ソースの修正を伴わずに接続先を変更することができます。
② ①で定義した定数を利用して接続文字列を構築します。
③ CONNECT 文を発行して接続を確立させます。

6） Eclipse IDE でデバッグ実行
COBOL エクスプローラにて対象の COBOL ファイルを選択し、右クリックから
[デバッグ] > [COBOL JVM アプリケーション] を選択しデバッグを開始[8]

[7] 接続文字列の指定は例示した記法に限らず、様々なコーディングが可能です。コーディングする際、適宜製品マニュアルをご参照してください。

[8] プログラムは COBOL でコーディングされていますが、生成されているのは JVM クラスとなるため、本オペレーションにて「Java アプリケーション」を選択してもデバッグ実行は可能です。

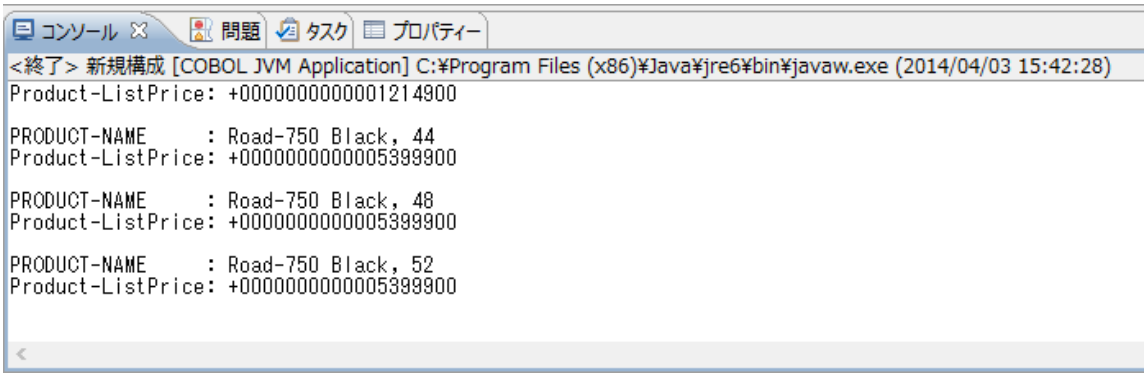
デバッグ画面：



① ステップ単位で処理を進めることができます。ブレークポイントを指定して任意のステップまで処理を自動で進めることもできます。

② COBOL の変数はもちろんのことホスト変数についても格納されている値をモニタすることが可能です。

実行結果：



5-4 で使用したプログラムに対して接続部分を書き換えるだけで JVM 環境で同じように動作させられることを確認しました。この技術を駆使すれば、Open レガシー環境等で稼働する埋め込み SQL 文を使う COBOL プログラムを JVM 環境へのほとんど書き換えをせずに移行できる可能性があることが確認できました。前項にて見たように OpenESQL は ADO.NET においても同じように埋め込み SQL 文を使った開発をサポートしているため、.NET から JVM への移行、またその逆についても容易に実現ができます。Visual COBOL の Eclipse 版はデバッガを COBOL 用に作りこんでいるため、Java の開発で一般的に利用されているデバッガを COBOL の開発でも同様に利用し更なる開発生産性の向上が見込めます。

5-6-2 OpenESQL JDBC を使った Java アプリケーションとの接続の共有

アプリケーションの将来性や使い勝手といったことを考えた場合、新しい技術を取り入れてアプリケーションを進化させたいという要求が挙がるのは必然です。しかし、長年コストをかけてメンテナンスを重ね、既に品質担保ができているビジネスロジック部分に手を入れることは非常に大きなリスクを伴います。Visual COBOL の COBOL for JVM の機能を利用すれば、COBOL プログラムで書かれたビジネスロジック部分を javabyte コードにリコンパイルし、JVM 環境上で JVM クラスとして継続して利用することができます。一方、グラフィックな処理部分等、COBOL の優位性が比較的低い部分については JSP/Servlet のようなこの分野で一般的に利用されている技術でリエンジニアリングできればコスト面、メンテナンス性等の観点から得られるメリットは大きいです。このようなかたちで JVM 化してモダナイズするのであれば Java と COBOL との連携は必須要件となります。javabyte コードにコンパイルされた COBOL プログラムは Java からは COBOL であるということを意識することなく一般的な Java で書かれた JVM クラスを呼び出すのと同じ記法で呼び出せます。JVM アプリケーション開発における簡便性を意識して拡張された COBOL 方言ではクラスオブジェクトの参照渡しを Java と同じように扱えるため、Java と接続を共有したり COBOL も含めたトランザクション管理を Java で行うこともできます。それだけでなく、埋め込み SQL 文を使った従来からの COBOL らしい表現で書かれたプログラムについても Java と接続やトランザクションを 共有することが可能です。この機能を活用すれば、RDB 連携を含む COBOL プログラムの JVM 化を一層効率化できる可能性があります。例えば、Java で RDB 接続のためのメソッド等を一元化したいといった要件に対しても、既に埋め込み SQL 文を使ってコーディングされ運用中の COBOL プログラムに軽微な修正を加えるだけで応えることができます。

以下では埋め込み SQL 文を使って、他の JVM 言語（本例では Java ）と接続及びトランザクションを共有する様子を記します。ここでは、接続を Java 側から開始し、その接続を使って COBOL 側で SELECT 文及び UPDATE 文を実行します。その UPDATE 文による変更を Java 側で COMMIT/ROLLBACK をして COBOL のトランザクションを管理していること示しています。

1） 前項と同じ要領で「COBOL JVM プロジェクト」を作成し、JDBC Driver をビルドパスに追加

2） COBOL プログラムを新規に追加
   ① COBOL エクスプローラにて「src」フォルダを右クリックし
   [新規] > [COBOL プログラム]
   を選択
   ② パッケージ名及びプログラム名に任意の値を指定して [完了] ボタンを押下

3） レコードを 1 件読み込み、その後にそのレコードに対して更新をかける埋め込み SQL 文を含む COBOL ロジックを 2) で追加したプログラムへコーディング

本例で用意したプログラムソース：

```
01        $SET SQL(DBMAN=JDBC)  SOURCEFORMAT(VARIABLE)
02         PROGRAM-ID. ReadOgUpd as "com.mfj.demo.ReadOgUpd".
03         DATA DIVISION.
04         WORKING-STORAGE SECTION.
05            EXEC SQL BEGIN DECLARE SECTION END-EXEC.
06         01  DCLtesttbl.
07            03 HV-INTCOL                PIC S9(09)  COMP-5.
08            03 HV-SMALLINTCOL           PIC S9(04)  COMP-5.
09         01 HV-CONN        TYPE java.sql.Connection.
10            EXEC SQL END DECLARE SECTION END-EXEC.
11            EXEC SQL INCLUDE SQLCA  END-EXEC.
12         01 SMALLINTCOL-DISP             PIC S9(04).
13         PROCEDURE DIVISION USING
14                   BY VALUE mySQLConnection AS TYPE java.sql.Connection.
15         MAIN-PROC SECTION.
16         MAIN-RTN.
17            SET HV-CONN TO mySQLConnection.
18            EXEC SQL
19               BIND CONNECTION SQLConnName TO :HV-CONN
20            END-EXEC.
21
22            MOVE 3 TO HV-INTCOL.
23            PERFORM SELECT-RTN.
24            DISPLAY "SMALLINTCOL AT COBOL(BEFORE UPDATE): " SMALLINTCOL-DISP.
25
```

```
26          COMPUTE HV-SMALLINTCOL = HV-SMALLINTCOL + 1.
27          PERFORM UPDATE-RTN.
28
29          PERFORM SELECT-RTN.
30          DISPLAY "SMALLINTCOL AT COBOL(AFTER UPDATE) : " SMALLINTCOL-DISP.
31
32          EXEC SQL UNBIND CONNECTION END-EXEC.
33          GOBACK.
34
35      SELECT-RTN.
36          EXEC SQL
37              SELECT SMALLINTCOL INTO :HV-SMALLINTCOL
38              FROM dbo.testtbl WHERE INTCOL = :HV-INTCOL
39          END-EXEC.
40          MOVE HV-SMALLINTCOL TO SMALLINTCOL-DISP.
41          EXIT.
42
43      UPDATE-RTN.
44          EXEC SQL
45              UPDATE dbo.testtbl
46                  SET SMALLINTCOL = :HV-SMALLINTCOL
47              WHERE INTCOL = :HV-INTCOL
48          END-EXEC.
49          EXIT.
```

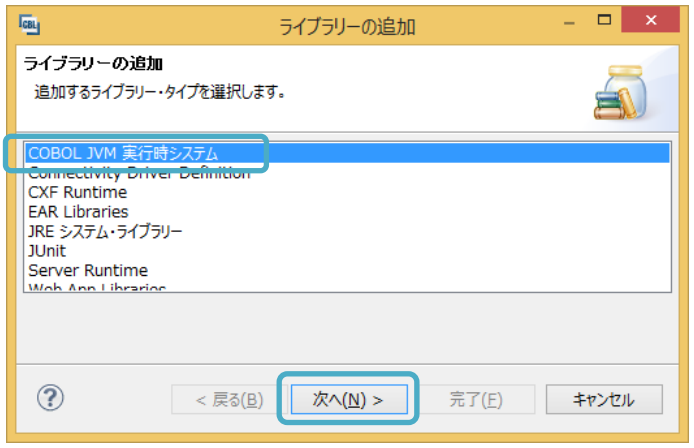
09 行目　　Java より受け取った接続オブジェクトを保持するためのホスト変数定義です。
17 行目　　Java より受け取った接続オブジェクトを COBOL 側で定義した変数にセットします。
19 行目　　この命令により OpenESQL にて Java で開始した接続を利用できるようになります。
32 行目　　OpenESQL から接続を解放し再び Java 側でこの接続オブジェクトを利用できるようにします。

4) Java プロジェクトを追加
[ファイル] メニュー > [新規] > [その他] > [Java プロジェクト]
を選択し、任意のプロジェクト名の Java プロジェクトを追加

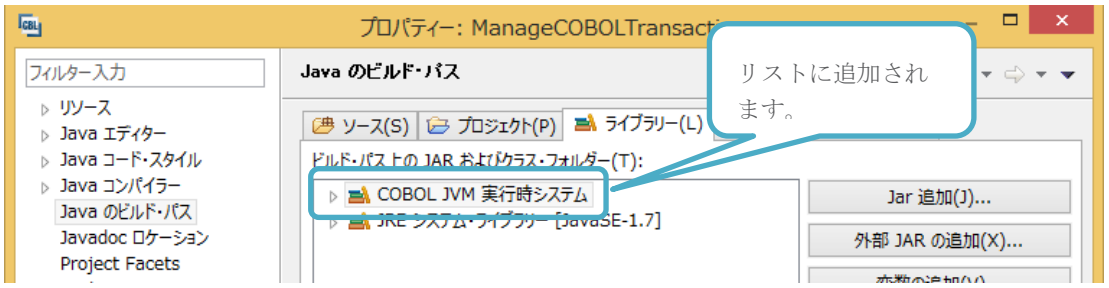
5) Java クラスを追加
① COBOL エクスプローラにて Java プロジェクトを右クリックし
[新規] > [クラス]
を選択
② 3) で指定したパッケージ名を「パッケージ」欄に指定
③ 任意のクラス名を「名前」欄に指定
④ 「public static void main(String[] args)」をチェック
⑤ [完了] ボタンを押下

6) Visual COBOL のランタイムライブラリを Java プロジェクトのビルドパスへ追加
① COBOL エクスプローラにて Java プロジェクトを右クリックし
[ビルド・パス] > [ビルド・パスの構成]
を選択
② [ライブラリーの追加] ボタンを押下

③ 「COBOL JVM 実行時システム」を選択の上、[次へ] ボタンを押下

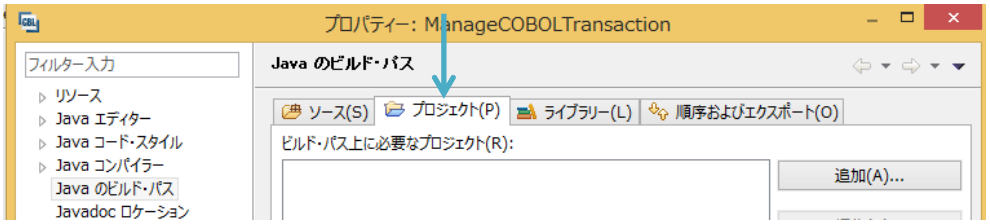


④ [完了] ボタンを押下



7) 上で用意した COBOL から生成される JVM クラスをビルドパスへ追加

① 7) で起動した「プロパティー」ウィンドウにおける「Java のビルド・パス」ページにて「プロジェクト」タブをクリック



② [追加] ボタンを押下
③ 用意した COBOL JVM プロジェクトにチェックを入れ、[OK] ボタンを押下
④ [OK] ボタンを押下し、「プロパティー」ウィンドウを終了

8) 上で用意した COBOL から生成される JVM クラスを 2 回呼び、1 回目はトランザクションを取り消し、2 回目はトランザクションを確定させるロジックを 7) で追加した Java クラスへコーディング

本例で用意したプログラムソース：

```
01  package com.mfj.demo;
02  import java.sql.*;
03
04  public class ManageCOBOLTransaction {
05      Connection con;
06
07      public static void main(String[] args) {
08          ManageCOBOLTransaction myClass = new ManageCOBOLTransaction();
09          myClass.handleCOBogSQL();
```

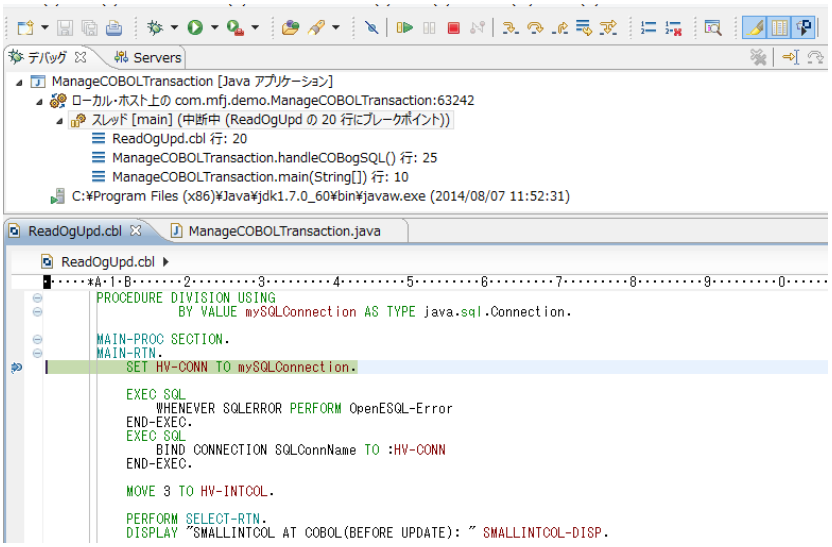
```
10      }
11
12      private void handleCOBogSQL(){
13          String jdbc_url = "jdbc:sqlserver://localhost;databaseName=TestDB";
14        String user = "coboltest";
15        String password = "password";
16
17        try {
18          Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
19          con = DriverManager.getConnection(jdbc_url, user, password);
20          con.setAutoCommit(false);
21
22          ReadOgUpd cobclass = new ReadOgUpd();
23          cobclass.ReadOgUpd(con);
24          con.rollback();
25          printRecord();
26
27          cobclass.ReadOgUpd(con);
28          con.commit();
29          printRecord();
30
31          con.close();
32
33        } catch (Exception e){
34          e.printStackTrace();
35        }
36      }
37
38      private void printRecord() throws SQLException{
39          Statement stmt = con.createStatement();
40        ResultSet rset = stmt.executeQuery(
41                    "SELECT SMALLINTCOL FROM dbo.testtbl WHERE INTCOL = 3");
42        while (rset.next()) {
43          System.out.println("SMALLINTCOL AT Java: " + rset.getString(1));
44        }
45        rset.close();
46        stmt.close();
47      }
48  }
```

19 行目　　RDB との接続を確立させます。
22 - 23 行目　一回目の COBOL の呼び出しです。通常の JVM クラス中のインスタンスメソッドを呼び出すように
COBOL から生成されたクラスをインスタンス化して、メソッド呼び出しをします。
COBOL 内で処理された更新を Rollback して取り消します。
24 行目　　COBOL が更新したレコードを読み込む SELECT 文を含むメソッドを呼び出し、更新が Rollback された
25 行目　　ことを確認します。
再び、COBOL を呼び出します。
27 行目　　COBOL で処理された更新を COMMMIT して確定させます。
28 行目　　更新確定後に対象のレコードを読み込み正しく反映されていることを確認します。
29 行目

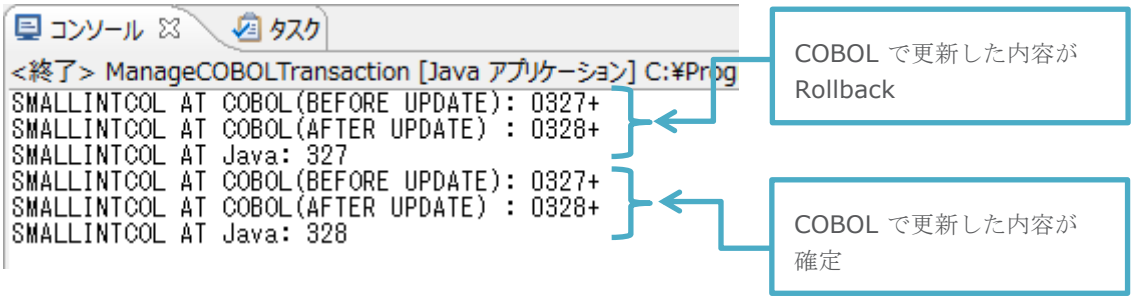
9) 用意したアプリケーションをデバッグ実行
① Java プログラム中の最初の命令にブレークポイントを指定
② COBOL プログラム中の最初の命令にブレークポイントを指定
③ パッケージ・エクスプローラにて Java プログラムを右クリックから
[デバッグ] > [Java アプリケーション]
を選択

④ 前項の要領で用意したアプリケーションをデバッグ

デバッグイメージ：



実行結果：



5-7 API 呼び出し

Visual COBOL で開発する COBOL アプリケーションは OS のシステムコールを含むあらゆる API を CALL 文を使って呼び出すことができます。本章でこれまで紹介してきた埋め込み SQL 文を使った各技法はプログラマが抽象的な表現でプログラムを組み上げるための機能を提供していますが、内部的には RDBMS や ODBC ドライバマネージャ等が提供する API を呼んでこれらとやりとりをしています。埋め込み SQL 文を使ってロジカルなプログラムを組み上げる手法は保守性や利便性等の観点で見ても優れてはいますが、何か特別な要件により直接 API にアクセスする必要があれば CALL 文を使ってアクセスすることも可能です。

ここではまず下記のような単純に ODBC 経由で接続を確立する C のプログラムを考えてみます：

```
#include <stdio.h>
#include <sql.h>
#include <sqlext.h>

main() {
  SQLHENV env;
  SQLHDBC dbc;
  SQLHSTMT stmt;
  SQLRETURN ret;
  SQLCHAR outstr[1024];
  SQLSMALLINT outstrlen;

  SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env);
  SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void *) SQL_OV_ODBC3, 0);
  SQLAllocHandle(SQL_HANDLE_DBC, env, &dbc);
  ret = SQLDriverConnect(dbc, NULL, "DSN=postgres;", SQL_NTS,
            outstr, sizeof(outstr), &outstrlen,
            SQL_DRIVER_COMPLETE);
  printf("Connected¥n");
  printf("Returned connection string was:¥n¥t%s¥n", outstr);

  SQLDisconnect(dbc);
  SQLFreeHandle(SQL_HANDLE_DBC, dbc);
  SQLFreeHandle(SQL_HANDLE_ENV, env);
}
```

これをコンパイルして実行すると「postgres」というデータソースを使って RDB との接続を確立し返ってきた接続先に関する情報をコンソール出力します。

```
$ gcc -lodbc -o simpconnc simpconn.c
$ ./simpconnc
Connected
Returned connection string was:

DSN=postgres;DATABASE=testdb;SERVER=localhost;PORT=5432;UID=postgres;PWD=;SSLmode=disable;ReadOnly=0;Protocol=7.4;FakeOidIndex=0;ShowOidColumn=0;RowVersioning=0;ShowSystemTables=0;ConnSettings=;Fetch=100;Socket=4096;UnknownSizes=0;MaxVarcharSize=255;MaxLongVarcharSize=8190;Debug=0;CommLog=0;Optimizer=0;Ksqo=1;UseDeclareFetch=0;TextAsLongVarchar=1;UnknownsAsLongVarchar=0;BoolsAsChar=1;Parse=0;CancelAsFreeStmt=0;ExtraSysTablePrefixes=dd_;;LFConversion=0;UpdatableCursors=1;DisallowPremature=0;TrueIsMinus1=0;BI=0;ByteaAsLongVarBinary=0;UseServerSidePrepare=1;LowerCaseIdentifier=0;
$
```

これに相当するロジックを COBOL で組み上げると下記のようになります。

```
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01 SQL-HANDLE-ENV           PIC S9(04) COMP-5 VALUE 1.
       01 SQL-NULL-HANDLE          PIC S9(18) COMP-5 VALUE 0.
       01 SQLHENV                  USAGE POINTER.
       01 SQL-ATTR-ODBC-VERSION    PIC S9(09) COMP-5 VALUE 200.
       01 SQLPOINTER               POINTER.
       01 SQL-OV-ODBC3             PIC  9(18) COMP-5 VALUE 3.
       01 ZERO-PARM                PIC  9(09) COMP-5 VALUE 0.
       01 SQL-HANDLE-DBC           PIC S9(04) COMP-5 VALUE 2.
       01 SQLHDBC                  USAGE POINTER.
       01 SQL-RETURN               PIC S9(04) COMP-5 VALUE 0.
       01 SQLHWND                  USAGE POINTER VALUE NULL.
       01 IN-CONNECTION-STRING-VAL  PIC  X(13) VALUE "DSN=postgres;".
       01 IN-CONNECTION-STRING      USAGE POINTER.
       01 SQL-NTS                  PIC S9(04) COMP-5 VALUE -3.
       01 OUT-CONNECTION-STRING    PIC  X(1024).
       01 BUFFER-LENGTH            PIC S9(04) COMP-5 VALUE 1024.
       01 STRING-LENGTH2PTR        USAGE POINTER.
       01 SQL-DRIVER-COMPLETE      PIC  9(04) COMP-5 VALUE 1.
       PROCEDURE DIVISION.
          CALL "SQLAllocHandle" USING BY VALUE SQL-HANDLE-ENV
                                BY VALUE SQL-NULL-HANDLE
                                BY REFERENCE SQLHENV..
          CALL "SQLSetEnvAttr" USING BY VALUE SQLHENV
                               BY VALUE SQL-ATTR-ODBC-VERSION
                               BY VALUE SQL-OV-ODBC3
                               BY VALUE ZERO-PARM.
          CALL "SQLAllocHandle" USING BY VALUE SQL-HANDLE-DBC
                                BY VALUE SQLHENV
                                BY REFERENCE SQLHDBC.

          SET IN-CONNECTION-STRING TO
             ADDRESS OF IN-CONNECTION-STRING-VAL.
          SET STRING-LENGTH2PTR TO ADDRESS OF BUFFER-LENGTH.
          CALL "SQLDriverConnect" USING
                   BY VALUE     SQLHDBC
                   BY VALUE     SQLHWND
                   BY VALUE     IN-CONNECTION-STRING
                   BY VALUE     SQL-NTS
                   BY REFERENCE OUT-CONNECTION-STRING
                   BY VALUE     BUFFER-LENGTH
                   BY REFERENCE STRING-LENGTH2PTR
                   BY VALUE     SQL-DRIVER-COMPLETE
                            RETURNING SQL-RETURN.

          DISPLAY "Connected: "
          DISPLAY "Returned connection string was: "
                 OUT-CONNECTION-STRING
          CALL "SQLDisconnect" USING BY VALUE SQLHDBC
          CALL "SQLFreeHandle" USING BY VALUE SQL-HANDLE-DBC
                               BY VALUE SQLHDBC.
          CALL "SQLFreeHandle" USING BY VALUE SQL-HANDLE-ENV
                               BY VALUE SQLHENV.
          GOBACK.
```

このプログラムをコンパイルして実行すると C でコーディングしたプログラムと同等の結果が得られることがわかります。

```
$ cob -x -lodbc -o simpconncob simpconn.cbl
$ ./simpconncob
Connected:
Returned connection string was:
DSN=postgres;DATABASE=testdb;SERVER=localhost;PORT=5432;UID=postgres;PWD=;SSLmode=disa
ble;ReadOnly=0;Protocol=7.4;FakeOidIndex=0;ShowOidColumn=0;RowVersioning=0;ShowSystemTable
s=0;ConnSettings=;Fetch=100;Socket=4096;UnknownSizes=0;MaxVarcharSize=255;MaxLongVarcharSi
ze=8190;Debug=0;CommLog=0;Optimizer=0;Ksqo=1;UseDeclareFetch=0;TextAsLongVarchar=1;Unkn
ownsAsLongVarchar=0;BoolsAsChar=1;Parse=0;CancelAsFreeStmt=0;ExtraSysTablePrefixes=dd_;;LFCo
nversion=0;UpdatableCursors=1;DisallowPremature=0;TrueIsMinus1=0;BI=0;ByteaAsLongVarBinary=0
;UseServerSidePrepare=1;LowerCaseIdentifier=0;
$
```

今回は接続処理に絞って API コールを使った技法を紹介しましたが、接続処理のみならずデータ取得、データ更新等についても同様に複雑な処理を要します。埋め込み SQL 文を使ってコーディングする場合はこのような複雑な API コール等の処理を隠蔽しプログラマに意識させません。OpenESQL をはじめとした埋め込み SQL 文を使った技術を利用する場合、接続処理であれば CONNECT 文 1 つで表現しロジカルなプログラム構築を可能としています。

おわりに

以上、Micro Focus™ Visual COBOL が提供する各種データアクセス機能をそれぞれ例を交えて概説してきました。企業活動において迅速且つ正確な情報管理は肝要であり、それを実現するための仕組みが企業システムには求められます。Visual COBOL は上述のように 50 年以上にも及ぶ COBOL の歴史とともに実績を重ねるデータファイル管理や現代の企業データ管理で一般的に利用される RDBMS へのアクセスを様々なかたちから実現する機能を装備しています。いずれの機能を用いても各利用者のデータモデルや設計に応じた柔軟なカスタマイズが可能です。加えて、Visual COBOL にはデータアクセスプログラム開発を補助するためのユーティリティも整備されており、俊敏な開発を支援します。以上に述べた各機能についてはオンラインマニュアル及び製品付属のマニュアルでより詳述しています。また製品には各種テーマに応じたサンプルも付属されており、これらを使って実際の動きを確認することも可能です。
昨今の企業が扱うデータ量は急速なスピードで増加しており、IT 技術の進歩にも相まってプロビジョニングの迅速性、高可用性、堅牢性、パフォーマンス等を向上させるための様々な技術が生まれてきました。この中には RDBMS のような既存技術をベースとした追加機能や機能向上だけではなく、NoSQL のような新しい技術も含まれます。このような先端技術と COBOL アプリケーションは組み合わせられないと考えられがちですが、Visual COBOL の柔軟性はこのような技術との連携をも可能にします。この技法を含めた技術を解説する動画も Micro Focus のホームページで公開しています。
Visual COBOL には COBOL からデータアクセスさせるための機能が豊富に搭載されています。この豊富な選択肢の中から皆様が現実に即して技術を検討する上で本書がその出発点となり、検討の一助を担えれば幸いです。

マイクロフォーカス合同会社
https://www.amc.rocketsoftware.co.jp/