Visual COBOL
White Paper

Rocket® software

データベース切り替えによるプログラムへの影響

Last updated: October 31, 2024

1. はじめに

企業の知的財産である COBOL アプリケーションは、ビジネスにおいて変わらず重要な役割を担っています。しかし、そのアプリケーションが稼働する環境は、ハードウェア機器や製品の保守サポート終了、新しい技術の導入などによって日々変化を続けています。稼働環境が変わったとしても、国際規格に則った COBOL 構文 で構築されたアプリケーションは従来と同様に動作しますが、言語仕様外の範囲では、その限りではありません。例えば、クラウドへの移行に合わせてクラウド プロバイダーが提供するマネージドのデータベース サービスを利用することになり、現在利用しているデータベースからの変更が発生するケースを考えます。データベース製品が独自に拡張した機能や SQL 方言といわれるデータベース製品固有の機能を利用している場合は、プログラムの変更が必要です。また、ネットワーク環境の差異による性能遅延対策が必要となる場合があります.

本ドキュメントではデータベースの切り替えによって発生しうる一般的なアプリケーションへの影響とその対策方法について説明いたします。

なお、データベースには様々な種類がありますが、本ドキュメントでは COBOL アプリケーションが多く利用しているリレーショナル データベース (RDBMS) を前提とします。

2. データベースへの接続方法

データベース連携アプリケーションの開発には、データベースとの通信仕様に基づいた手順をプログラムとして実装する必要があります。しかし、実際のアプリケーション開発では、これらの通信仕様に基づいた手順を自ら実装することはなく、以下のような方法を利用します。

- データベース製品が提供するコンポーネントの利用
- ODBC (Open Database Connectivity) の利用
- JDBC (Java Database Connectivity) の利用
- ADO.NET の利用

2.1. データベース製品が提供するコンポーネントの利用

データベース製品は、それぞれのデータベースが定める通信仕様に基づいた連携機能、すなわち API を提供しています。ほかの方法と比較しますと、データベース製品が提供することから、一般的にその機能を最も多く利用することができます。一方、データベース製品を変更すると、プログラムの修正が必要になることがあります。

2.2. ODBC (Open Database Connectivity) の利用

ODBC とは、複数のデータベースで実装すべきデータベース API を定めた仕様で、アプリケーションは ODBC 経由でデータベースにアクセスできるようになります。ODBC 準拠のデータベースと同じ方法で連携できることが特徴で、多くのデータベース製品が ODBC に準拠しています。しかし、ODBC が標準機能を定めた仕様であることから、データベース固有の機能など、すべての機能を利用できないこともあります。

2.3. JDBC (Java Database Connectivity) の利用

ODBC の仕様をもとに、Java アプリケーションから様々なデータベースと連携できるように定めた規格が JDBC です。一般的には、Java アプリケーションは JDBC を利用することになりますが、JDBC から ODBC ドライバーを利用する JDBC-ODBC ブリッジを採用することで、ODBC ドライバーを介したデータベースアクセスも行えるようです。ODBC と同様、JDBC も標準機能を定めた仕様であるため、データベース固有の機能など、すべての機能を利用できないこともあります。

2.4. ADO.NET の利用

.NET Framework, .NET 上で実行されるプログラムがデータベースや XML などに標準的な方法で連携できるように定めた規格が ADO.NET です。.NET テクノロジを構成する 1 つのクラスとして提供されており、C# などの .NET 言語でのアプリケーション開発では一般的に ADO.NET を利用してデータベースアクセスを行います。

3. データベースと連携する COBOL アプリケーション開発

データベースへの問合せは 、一般的には SQL と呼ばれるクエリ言語を用いて行われますが、この SQL 構文は COBOL の言語規格に含まれていません。しかし、COBOL は、ビジネス ロジックと同様に、データベースへの問合せをそのプログラム内に記述できる、埋込み SQL という機能を提供しています。

```
        EXEC SQL
          INSERT INTO book_info VALUES (
               :B-STOCKNO,
               :B-TITLE,
               :B-TYPE,
               :B-AUTHOR,
               :B-RETAIL,
               :B-ONHAND,
               :B-SOLD
           )
        END-EXEC.
```

上記のように、EXEC-SQL から END-EXEC によって囲まれた範囲にホスト変数と呼ばれるデータ項目の定義や SQL を記述することで、データベースからデータの取得や挿入といった操作を COBOL プログラム内で記述できるようになります。なお、埋込み SQL をそのまま COBOL 言語としてコンパイルすることはできませんので、プリプロセッサ (Preprocessor) と呼ばれる機能がコンパイル処理の前に実行されます。プリプロセッサによって、埋込み SQL 部が COBOL 言語に変換され、コンパイルできるようになります。Visual COBOL 製品では、市販されているデータベース製品が提供するプリプロセッサを利用した開発だけではなく、ODBC / JDBC 仕様準拠の様々なデータベースと通信可能な OpenESQL プリプロセッサを利用した開発が行えます。

なお、データベース製品が提供するプリプロセッサを利用した開発では、プログラム内の埋め込み SQL 句が各データベースの適切な呼び出し方法に変換されます。この変換後のコードをもとにデバッグ作業を行うことになりますが、オリジナルと大きく異なるため、デバッグ作業は容易ではありません。しかし、Visual COBOL 製品搭載の統合プリプロセッサ COBSQL を利用することで、変換前のプログラム上でデバッグ作業を効率よく実施することができます。

4. データベース変更による影響

本節からは、Visual COBOL 製品を使用したアプリケーションの移行に伴う、データベース変更時の考慮点について紹介します。

4.1. データベース接続情報

COBOL を用いてデータベース連携アプリケーションを開発する場合、データベースとの接続をアプリケーション自身で行う方法と XA 準拠リソースを利用する方法があります。

4.1.1. アプリケーション自身で行う方法

この方法では、データベースの認証情報やサーバー アドレス、ポート番号といった情報、もしくは、それらを定義したデータソース名をプログラムで指定します。これまでデータベース製品が提供するプリプロセッサを利用したデータベース連携アプリケーションを OpenESQL に切り替える場合、プログラムの修正が必要になる場合があります。

以下は、Db2 製品のプリプロセッサを利用するプログラムですが、接続先のデータベース名のみを指定しています。データベース接続時の認証はログイン情報が使用されます。

```
・・・
        MOVE “SAMPLE” TO WS-DB
        EXEC SQL
           CONNECT TO :WS-DB
        END-EXEC
・・・
```

Linux 環境を例にとると、プログラムは以下のコマンドでコンパイルします。

```
$ cob -u db2prep.cbl -C"DB2(DB==sample)"
```

しかし、PostgreSQL に移行する場合などでは、このような接続はできず、データソースへ認証情報を設定するか、プログラムを以下のように変更する必要があります[1]。

```
・・・
        EXEC SQL
          CONNECT :DB-USER IDENTIFIED BY :DB-PASSWORD
                        USING         :ODBC-NAME
        END-EXEC
・・・
```

また、コンパイルコマンドもコンパイル指令が変更になります。

[1] プログラムの記述方法はこちらのマニュアルに従います。

$ cob -u postgresqlopenesql.cbl -C”SQL(DBMAN==ODBC)”

4.1.2. XA 準拠リソースを利用する方法

データベース別にリソースマネージャーとの連携機能を提供するスイッチモジュールをビルドし、このモジュールを介して接続しますが、データベースによって接続文字列の設定方法が異なります。

表 1 XA リソースで設定する接続文字列仕様

データベース	設定仕様
Oracle	XA+SesTm=**time**+SqlNet=**name**+Acc=P/**user**/**password**
Db2	DB=**name**,UID=**identifier**,PWD=**password**,AXLIB=**casaxlib**
Microsoft SQL Server	DSN=**odbc_dsn**
EDB PostgreSQL	DSN=**odbc_dsn**

表内の太字個所が編集すべき個所となります。上記は弊社マニュアル作成時点での最小設定項目であり、各項目の詳細やその他項目については、各データベースマニュアルをご参照ください。

また、Visual COBOL 製品は、マネージ COBOL と呼ばれる Java バイトコードや .NET の CIL コードを COBOL プログラムから直接生成する機能を提供しています。本機能を利用する場合は、コンパイラ指令 DBMAN[2] 値の変更や CONNECT 構文の変更が必要になることがあります。

4.2. SQL

[2] Java バイトコード向けはこちら、.NET の CIL コード向けはこちらです

データベースへの接続方法と同様、データベースへの問合せを行う SQL にも、データベース特有の SQL 方言と呼ばれるものがあります。一例として、以下の Oracle や Db2 の DECODE 関数があげられます。

```
SELECT DECODE(deptno,
            10, 'ACCOUNTING',
            20, 'RESEARCH',
            30,'SALES',
            40, 'OPERATIONS')
FROM emp
```

上記は、emp テーブル の deptno カラムの値をもとに部署名を戻すクエリです。しかし、PostgreSQL など他のデータベースに移行した際、DECODE 句は符号化された値をデコードする関数として機能するため、以下のような CASE 句を用いたクエリに書き直す必要があります。

```
SELECT CASE deptno
   WHEN 10 THEN 'ACCOUNTING'
   WHEN 20 THEN 'RESEARCH'
   WHEN 30 THEN 'SALES'
   WHEN 40 THEN 'OPERATIONS'
END
FROM emp
WHERE empno=7934
```

このような SQL 方言を使用している場合は、移行先データベースのマニュアルなどを参照の上、同等の機能を持つクエリへの書き換えや、プログラム ロジックでの対応が必要になります。

4.3. エラー判定

データベースへの問合せの成否は、従来では SQLCODE、現在は SQLSTATE の値を確認することが一般的ですが、データベースによって異なる値を戻すことがあります。例として、レコード挿入時の一意制約違反で戻される値は、Microsoft SQL Server では SQLCODE=2627, SQLSTATE=23000 、PostgreSQL では SQLCODE=1, SQLSTATE=23505 です。このようなデータベース変更が行われる場合、一般的にはプログラム内の一意制約違反の判定個所全ての修正が必要です。しかし、Visual COBOL 製品が提供するプリプロセッサ OpenESQL を利用する場合は、設定ファイルによって戻される値を変更

可能な SQL エラー マッピング機能を利用することで、プログラム修正作業を削減することができます。例えば、以下の設定ファイルでは、前述した一意制約違反について Microsoft SQL Server から戻される値を PostgreSQL が戻す値に変更します。

```
$ cat emap/ms2pg.emap
-2627, 23000, , -1, 23505,~
```

4.4. 性能

基幹システムで多く利用される COBOL アプリケーションにとって、性能はシステムの最重要評価項目の１つです。もし、移行先の環境で期待した性能に達しない場合は、環境の違いに注目しながら原因を究明していく必要があります。データベース連携アプリケーションについて考えますと、性能遅延の大きな原因は以下の３つに分類されます。

1. 環境のリソース不足によるもの
2. 索引が正しく設定されていないなど、データベース側のチューニング不足によるもの
3. アプリケーションとデータベースの稼働環境構成の相違によるもの

１）、２）については、アプリケーションの問題ではないため、本ドキュメントでは説明の対象外とします。３）については、アプリケーションとデータベースが同一環境や同一ネットワーク内で動作していた移行元環境から、クラウド環境へ移行する際にマネージド データベース サービスを利用するような場合に発生することが多く、主な原因はデータベースへの問合せ時間が長くなったことに起因します。データベースへの問合せはミリ秒単位で行えているから問題ないと思いがちですが、大量のデータを扱うバッチ処理を担う COBOL アプリケーションでは大きな遅延になりえます。例として、あるバッチ処理が M 件のデータを処理することを考えます。１つのデータを処理するにあたり、N 回のデータベースへの問合せを行っている場合は、バッチ処理全体のデータベースへの問い合わせ件数は N x M 回で計算されます。M=50,000, N=10 を例とすると 500,000 回となり、移行先の環境でのデータベースへの問

合せに 1 ミリ秒だけ遅延が発生しているケースでも、全体では 500 秒の遅延になります。この問題は COBOL だけではなく他のプログラミング言語においても発生しうる問題であり、「

対策例」で紹介しているような手法を用いて改善を行います。

4.5. サービス指向アーキテクチャ(SOA) のアプリケーション運用

Visual COBOL 製品には、COBOL 資産を REST, SOAP 形式の Web サービスとして提供する機能やフロントの Java アプリケーション サーバーと JCA (Java Connector Architecture) 仕様に基づいた連携機能を提供する SOA (Service-Oriented Architecture) 機能[3]があり、本機能を用いて COBOL アプリケーションをサービスとして運用できます。また、製品に搭載されるマネージ COBOL 機能を用いて生成した COBOL プログラムの Java バイトコードや .NET の CIL コードを Java や .NET 技術で構築したシステム上でサービスとして運用することで、同様の構成を実現できます。いずれの構成でも、他システムやアプリケーションから容易に COBOL アプリケーションを呼び出すことができますが、以下の点に注意が必要です。

- 非同期リクエストの考慮
- XA リソースを用いる際のデータベース セッション管理

4.5.1. 非同期リクエストの考慮

一般的に、サービスクライアントプログラムは非同期のリクエストを送信します。クライアント側のタイムアウトは数秒であるため、長時間の処理の実行要求はクライアント側でタイムアウトになります。クライアント側で実行結果を得るため再実行を行うと、アプリケーション

[3] これらの機能については、マニュアルやチュートリアルにて紹介しております。

サーバーのみならず、データベースサーバーへの多重リクエストとなり、性能に影響を及ぼすだけではなく、多重処理が行われる可能性があります。

4.5.2. XA リソースを用いる際のデータベース セッション管理

4.1 で紹介したように、アプリケーションでデータベース接続を管理していたシステムを XA 準拠リソースを利用する方法に切り替える場合は、プログラムの修正が必要になります。

5. トラブルシューティング

データベース連携アプリケーションでは ODBC トレースを確認することでボトルネックとなる個所を特定できる場合があります。

5.1. ODBC トレースの取得方法

本項では、Windows/Linux に一般的な ODBC 経由のデータベース連携アプリケーションにおける ODBC トレース取得方法を紹介します。なお、トレースログの出力内容については、本ドキュメントの対象外となるため、別途資料をご参照ください。

Windows

Windows に標準搭載されている [ODBC Data Sources (xx-bit)[4]] ダイアログにて ODBC データソースを定義します。トレース設定についても、同ダイアログを開いたうえで [トレース] タブより [トレースの開始] をクリックすることで、[ログ ファイルのパス] に指定したファイルに ODBC ログが出力されます。

[4] xx は使用されている ODBC データソースのビット数 (32/64) に読み替えてください。

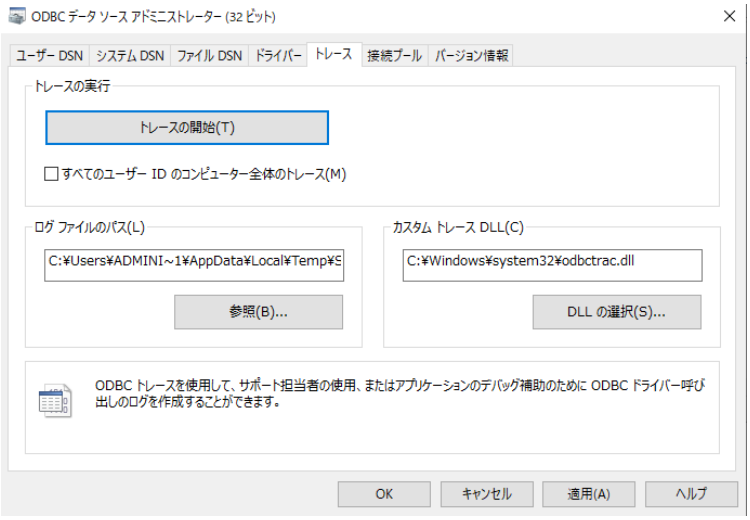

図 1 ODBC Data Sources ダイアログ

以下のログは、Microsoft SQL Server 接続時のトレースを一部抜粋したものです。

```
SQL32       32c-174c    ENTER SQLAllocEnv
             HENV *          0x012FF988
SQL32       32c-174c    EXIT  SQLAllocEnv  with return code 0 (SQL_SUCCESS)

SQL32       32c-174c    ENTER SQLAllocEnv
             HENV *          0x012FF988
SQL32       32c-174c    EXIT  SQLAllocEnv  with return code 0 (SQL_SUCCESS)
             HENV *          0x012FF988 ( 0x01847AF0)

SQL32       32c-174c    ENTER SQLSetEnvAttr
             SQLHENV           0x01847AF0
             SQLINTEGER                200 <SQL_ATTR_ODBC_VERSION>
             SQLPOINTER                2 <SQL_OV_ODBC2>
             SQLINTEGER                  0

SQL32       32c-174c    EXIT  SQLSetEnvAttr  with return code 0 (SQL_SUCCESS)
             SQLHENV           0x01847AF0
             SQLINTEGER                200 <SQL_ATTR_ODBC_VERSION>
             SQLPOINTER                2 <SQL_OV_ODBC2>
             SQLINTEGER                  0HENV *           0x012FF988 ( 0x01847AF0)

SQL32       32c-174c    ENTER SQLSetEnvAttr
             SQLHENV           0x01847AF0
             SQLINTEGER                200 <SQL_ATTR_ODBC_VERSION>
             SQLPOINTER                2 <SQL_OV_ODBC2>
             SQLINTEGER                  0
```

Linux

Linux では、unixODBC と呼ばれるオープンソースの ODBC ドライバー マネージャーを利用するケースを紹介します。unixODBC で ODBC トレースを行うためには、odbcinist.ini に以下の定義を追加します。

```
[ODBC]
Trace=1
TraceFile=/tmp/sqlodbc.log
```

TraceFile で定義したパスに ODBC トレースログが出力されます。以下のログは、PostgreSQL に ODBC 経由で接続したトレースの抜粋です。

```
[ODBC][7271][1716255341.745713][__handles.c][493]
            Exit:[SQL_SUCCESS]
                  Environment = 0x1e6d760
[ODBC][7271][1716255341.745838][SQLSetEnvAttr.c][192]
            Entry:
                  Environment = 0x1e6d760
                  Attribute = SQL_ATTR_ODBC_VERSION
                  Value = 0x2
                  StrLen = 0
[ODBC][7271][1716255341.745871][SQLSetEnvAttr.c][384]
            Exit:[SQL_SUCCESS]
[ODBC][7271][1716255341.748288][SQLAllocHandle.c][469]
            Entry:
                  Handle Type = 2
                  Input Handle = 0x1e6d760
[ODBC][7271][1716255341.748363][SQLAllocHandle.c][585]
            Exit:[SQL_SUCCESS]
                  Output Handle = 0x1e93110
[ODBC][7271][1716255341.748491][SQLDataSources.c][203]
            Entry:
                  Environment = 0x1e6d760
[ODBC][7271][1716255341.750484][SQLDataSources.c][404]
            Exit:[SQL_SUCCESS]
```

5.2. 対策例

本章記載の対策例は、COBOL 言語固有ではなく Java, .NET など他のプログラミング言語で開発されたアプリケーションでも広く採用されている手法です。ボトルネック要因を正しく判断したうえで対策を実施することが、どのようなアプリケーションにおいても重要になります。

5.2.1. ネットワーク構成の見直し

アプリケーションが稼働する環境からデータベースが稼働する環境までのネットワーク構成を見直すことで、環境間の通信応答時間を改善します。本手法は、アプリケーション観点の対策ではないため、環境を構築するプラットフォームやネットワーク関連資料などを参照ください。

5.2.2. 接続プール / コネクションプール

アプリケーションからデータベースへ接続・切断を行う場合、それぞれの処理でリソースに負荷がかかります。この回数が多いことにより性能遅延が発生している場合は、接続を保持して再利用する、接続プールという手法を用いることで改善が望めます。

接続プール機能は、データベース サーバー、ODBC ドライバー マネージャー などから提供されています。さらに、この機能を COBOL や他の開発言語を用いて、アプリケーション機能として開発することもできます。

5.2.3. バルク処理

バルク処理とは、データベースへ同じ問合せを行う際に、複数の要求を一回の問合せで行う手法です。Java や .NET などのプログラミング言語においては一般的に利用されており、COBOL でも利用することができます。COBOL プログラム内の埋込み SQL を用いたデータベースへの問合せは、COBOL プログラムとのデータ受渡しにホスト変数と呼ばれるデータ項目を使用します。バルク処理では、このホスト変数を OCCURS 句によって配列定義したホスト配列を利用することで、配列の要素数分のデータを一括取得できるようになります。

以下は、book_info テーブルの stock_no カラム値の取得を一回の問合せで 10 件取得するコードです。

```
・・・
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    EXEC   SQL BEGIN   DECLARE SECTION    END-EXEC
    01 B-DETAILS.
     03 B-STOCKNO    OCCURS 10 TIMES PIC X(4).
     03 B-TITLE      OCCURS 10 TIMES PIC X(20).
```

```
・・・
       EXEC SQL
         SELECT
                stock_no, title
           INTO :B-STOCKNO, :B-TITLE
           FROM
             book_info
           ORDER BY
             stock_no
       END-EXEC.
・・・
```

ホスト配列を使用しない場合は、データ取得件数分の SELECT 処理が発行されます。一方、ホスト配列を用いることで、SELECT 処理の発行回数を CEILING(データ取得件数 / ホスト配列の要素数) に抑えることができます。

INSERT, UPDATE 処理については、FOR 句を用いてデータ件数を指定することにより、対象となるデータのみを適切に指定できます。以下のコードでは、埋込み SQL で使用しているホスト配列の要素数が 10 となっていますが、MAX-INSERT-NUM 項目に 2 を設定することで INSERT 処理は 2 件になります。

```
     WORKING-STORAGE SECTION.
     EXEC    SQL BEGIN   DECLARE SECTION    END-EXEC
     01 B-DETAILS.
      03 B-STOCKNO    OCCURS 10 TIMES PIC X(4).
      03 B-TITLE      OCCURS 10 TIMES PIC X(20).
     01 MAX-INSERT-NUM  PIC S9(4) COMP-5.
・・・
       MOVE 2 TO MAX-INSERT-NUM.
        EXEC SQL
          FOR :MAX-INSERT-NUM
          INSERT INTO book_info VALUES (
               :B-STOCKNO,
               :B-TITLE
           )
        END-EXEC
・・・
```

大量のデータを扱うバッチ処理を担う COBOL アプリケーションへのバルク処理の導入は、データベースへの問合せ数を大きく削減することができます。

5.2.4. データのプリフェッチ

データベースによっては、SELECT 句の実行時に内部的に複数データを取得しキャッシュすることで、ロジック内のデータ取得処理のパフォーマンスを向上させる、プリフェッチ機能

が提供されていることがあります。こちらの機能を利用することで、さきほどのバルク処理より少ないプログラム修正量で性能を改善できる可能性があります。

5.2.5. CURSOR 宣言時の DEF_CURSOR 値

CURSOR を利用したデータの取得処理では、DEF_CURSOR 指令の値を確認してください。FOR UPDATE, FOR SHARE などを指定しない CURSOR では DEF_CURSOR 指令の値を “RO”（読み取り専用）とすることで性能を改善できる可能性があります。

5.2.6. ロジックの見直し

ロジックのストアドプロシージャ化、問合せで参照するデータを事前にキャッシュ化、もしくは、データをミラーリングするなどの方法を利用することで、問合せ速度の改善を図る手法です。

6. おわりに

長い間ビジネスの基幹を支え続けている COBOL アプリケーションは、現在もそのステップ数が増加していると言われています。一方で、現行環境のサポート終了などの外部要因により、COBOL アプリケーションが稼働している環境の最新化やデータベースの変更を求められています。データベースや環境構成の変化に起因するアプリケーションの修正や性能遅延については、COBOL 言語に限らず、どのようなプログラミング言語で開発されたアプリケーションでも発生しますが、その調査や対策もまた、プログラミング言語特有のものではありません。正しい調査と対策を実施することで、国際規格によって保護された COBOL アプリケーションは、従来と同じように最新環境上でもビジネスを支え続けます。

本ホワイトペーパーは 2024 年 11 月に作成したものです。
MFWPV3-2411-00MFA ¦ © 2024 Rocket Software. All rights reserved.