PostgreSQL インサイド

PL/SQL を PostgreSQL に移行する

移行概要/移行例

技術を知る

FUJITSU

☐ 導入／環境設定 ☑ 移行 ☐ 性能 ☐ チューニング ☐ バックアップ／リカバリー
☐ 冗長化／負荷分散 ☐ 監視 ☐ データ連携 ☐ 災害対策 ☐ 豆知識

Oracle Database から PostgreSQL への移行を検討する際、膨大な PL/SQL（注 1）資産をどのように移行すればよいか、お困りになったことはないでしょうか。この記事では、これから Oracle Database から PostgreSQL への移行を担うエンジニアの皆さまが、安全かつスムーズに移行を実施できるよう、PL/SQL と PL/pgSQL（注 2）の違い、移行の概要やポイントについてサンプルを交えて解説します。
対象となる Oracle Database のバージョンは 10g 以降、PostgreSQL のバージョンは 11 以降を想定しています。

- 注 1） Oracle Database 独自の手続き型言語の 1 つ
- 注 2） PostgreSQL 独自の手続き型言語の 1 つ

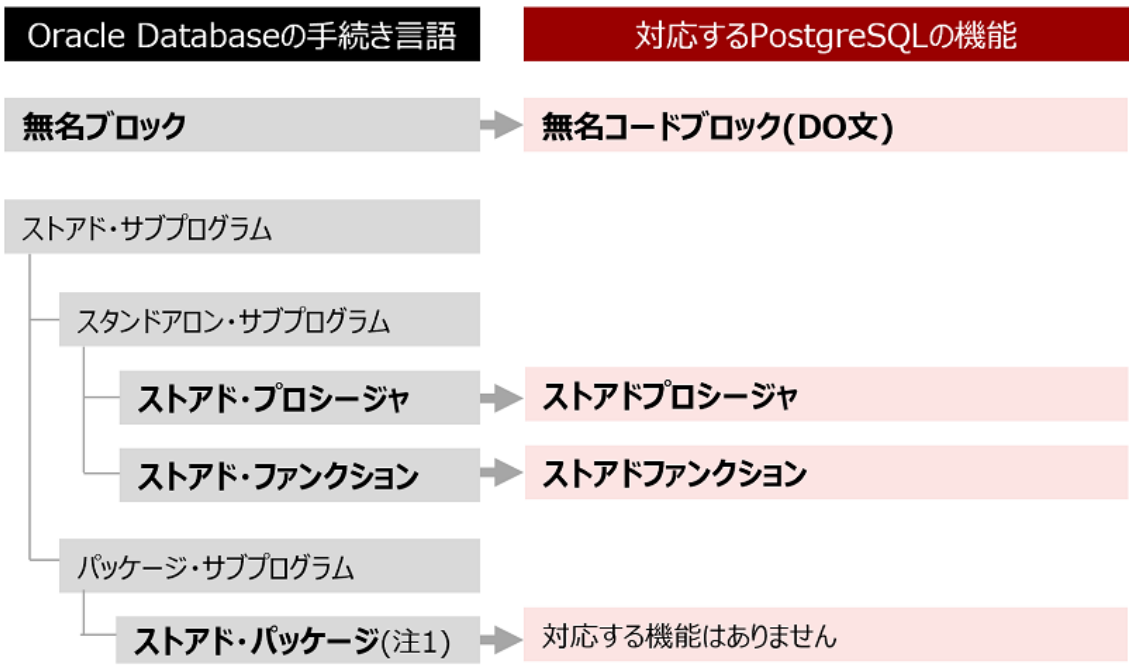
なお、Oracle Database から PostgreSQL への移行にあたっては、まずアーキテクチャーの違いや移行プロセスの概要について押さえておくことが重要です。以下の記事で解説していますので、まだ読まれていない方は、是非先にお読みください。

- データベース移行で押さえておくべきこと ～アーキテクチャーと機能の違い～
- データベース移行で押さえておくべきこと ～移行プロセスについて～

1. 移行対象となる Oracle の手続き言語の種類

ここでは、移行の対象となる Oracle Database の手続き言語の種類と、それに対応する PostgreSQL の機能について説明します。

Oracle Database の PL/SQL は、無名ブロックとストアド・サブプログラムに分類されます。さらに、ストアド・サブプログラムはスタンドアロン・サブプログラムとパッケージ・サブプログラムに分類され、スタンドアロン・サブプログラムにはストアド・プロシージャとストアド・ファンクションがあります。以下の図では、移行の対象となる手続き言語の種類と、対応する PostgreSQL の機能を、太字で示しています。



- 注 1)　ストアド・パッケージには Oracle 社から標準で提供されているパッケージも含まれますが、本記事では対象外とします。

参考

OSS の PostgreSQL をエンタープライズ向けに強化した製品「FUJITSU Software Enterprise Postgres（以降、Enterprise Postgres と略す）」では、PL/SQL パッケージの中でもよく利用されている以下のものを、パッケージとして提供しています。

DBMS_ALERT、DBMS_ASSERT、DBMS_OUTPUT、DBMS_PIPE、DBMS_RANDOM、DBMS_UTILITUY、UTL_FILE、DBMS_SQL

なお、PL/SQL パッケージとは仕様の差異があるため、利用の際にはマニュアルを確認してください。

無名ブロック

PostgreSQL では無名コードブロック（DO 文）に相当します。

- 特徴：スクリプト言語から直接 SQL や PL/SQL 制御ステートメントの処理を記述したものです。
- 仕組み：シェルスクリプトやバッチのようなものです。1 行ずつデータベースと通信しながら実行されます。
- 違い：構文が少し異なります。Oracle Database では無名ブロック内にサブプログラムを書けますが、PostgreSQL では書けません。

ストアド・プロシージャ

- 特徴：SQL や PL/SQL 制御ステートメントを組みわせた処理手順を記述したものです。トランザクション制御が利用できます。
- 仕組み：解析済みの状態でサーバーに格納され、記述した処理はサーバー上で実行されます。復帰値を返しません（ただし、OUT パラメーターを使うことで値を返すことが可能）。
- 違い：構文が少し異なります、Oracle Database ではストアド・プロシージャ内にサブプログラムを書けますが、PostgreSQL では書けません。

ストアド・ファンクション

- 特徴：SQL や PL/SQL 制御ステートメントを組みわせた処理手順を記述したものです。トランザクション制御は利用できません。
- 仕組み：解析済みの状態でサーバーに格納され、記述した処理はサーバー上で実行されます。復帰値を返します。
- 違い：構文が少し異なります、Oracle Database ではストアド・ファンクション内にサブプログラムを書けますが、PostgreSQL では書けません。

ストアド・パッケージ

- 特徴：機能や用途などの単位で、複数のプロシージャやファンクションをパッケージングしたものです。
- 仕組み：解析済みの状態でサーバーに格納され、記述した処理はサーバー上で実行されます。
- 違い：PostgreSQL には同様の機能は無く、移行にはストアドプロシージャやストアドファンクションに分解して置き換えていく必要があります。

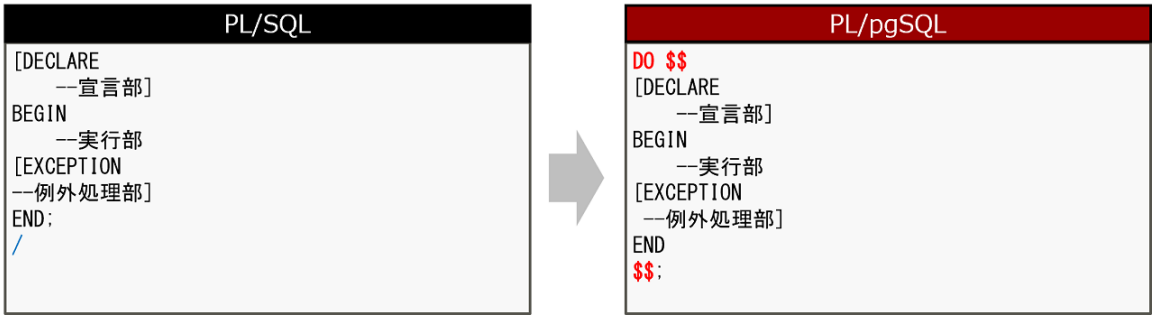
ストアド・パッケージの移行については、本記事の説明に加えてストアド・パッケージならではの手法が必要になります。詳しくは「PL/SQL のストアド・パッケージを PostgreSQL に移行する ～移行概要/移行例～」も併せてお読みください。

2. PL/SQL と PL/pgSQL の基本的な構文と呼び出し方の違い

Oracle Database の PL/SQL と、PostgreSQL の PL/pgSQL での基本的な構文と呼び出し方の違いについて説明します。

無名ブロック

PostgreSQL では、無名コードブロック（DO 文）で記述します（構文の[]は省略可能）。



呼び出し方：

どちらも、プロンプトなどから無名ブロック（無名コードブロック）を記述し、構文が完結した際に、記述した処理が解析されて 1 回実行されます。

ストアド・プロシージャ

PostgreSQL では、PostgreSQL 11 からストアドプロシージャが使えるようになりました。ただし、以下のように構文に違いがあります。

PL/SQL

```
CREATE OR REPLACE PROCEDURE <プロシージャ名>(引数)
IS
    --宣言部
BEGIN
    --実行部
[EXCEPTION
 --例外処理部]
END;
/
```

PL/pgSQL

```
CREATE OR REPLACE PROCEDURE <プロシージャ名>(引数)
AS $$
DECLARE
    --宣言部
BEGIN
    --実行部
[EXCEPTION
 --例外処理部]
END;
$$ LANGUAGE plpgsql;
```

- 補足） Oracle Database の場合、パラメーターが無い場合は引数用の括弧()は不要ですが、PostgreSQL では必要です。

呼び出し方：

PL/SQL

```
EXECUTE <プロシージャ名>(引数);
または、
CALL <プロシージャ名>(引数);

※手続き処理内ではそのまま記述できる
BEGIN
    <プロシージャ名>(引数);
END;
/
```

PL/pgSQL

```
CALL <プロシージャ名>(引数);

※手続き処理内でもCALL文で呼び出す必要がある
DO $$
BEGIN
    CALL <プロシージャ名>(引数);
END $$;
```

ストアド・ファンクション

以下のように構文に違いがあります。

PL/SQL

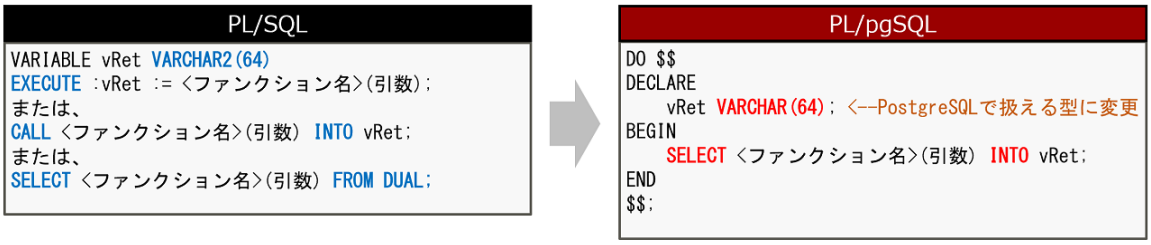
```
CREATE OR REPLACE FUNCTION <ファンクション名>(引数)
RETURN <戻り値の型> IS
    --宣言部
BEGIN
    --実行部
    RETURN ＜戻り値＞;
[EXCEPTION
 --例外処理部]
END;
/
```

PL/pgSQL

```
CREATE OR REPLACE FUNCTION <ファンクション名>(引数)
RETURNS <戻り値の型> AS $$
DECLARE
    --宣言部
BEGIN
    --実行部
    RETURN ＜戻り値＞;
[EXCEPTION
 --例外処理部]
END;
$$ LANGUAGE plpgsql;
```

- 補足）
    - Oracle Database では、パラメーターが無い場合は引数用の括弧()は不要ですが、PostgreSQL は必要です。
    - PostgreSQL では、戻り値が無い場合は、RETURNS VOID を指定します。

呼び出し方：



- 補足）　PostgreSQL では、戻り値を受け取る変数を定義するため、無名コードブロックを使います。データ型は、PostgreSQL で扱える型に変更します。なお、VARCHAR2(64)は 64 バイトの文字列を意味しますが、VARCHAR(64)は 64 文字の文字列を意味します。

3. 移行時によくある非互換の観点と対処方法

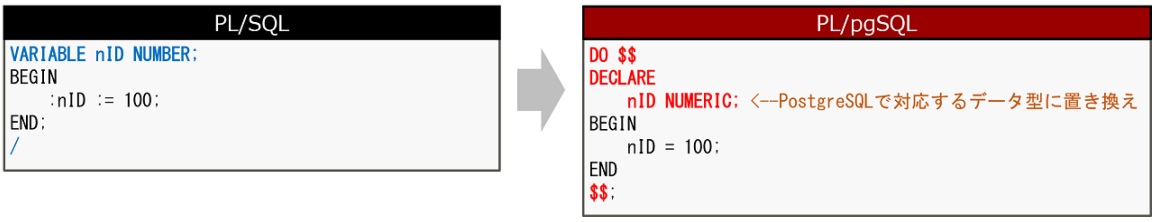
移行に際しては、PL/SQL と PL/pgSQL の構文の違いだけでなく、多くの非互換を考慮する必要があります。そのうち、よく遭遇する非互換の観点を以下に挙げました。ここでは、これらの観点と対処方法について説明します。

- VARIABLE 変数宣言
- コレクション型を戻すストアド・ファンクション
- トランザクション制御
- 制御文（GOTO 文）
- 制御文（FORALL 文）
- 制御文（FOR LOOP 文）
- 例外処理
- %ROWTYPE 属性（ファンクションやプロシージャ引数として利用する場合）
- カーソル属性（%ISOPEN / %NOTFOUND / %FOUND / %ROWCOUNT）
- その他（関連する非互換）
    - 長さ 0 の文字列（空文字列）
    - 結合（||）
    - 暗黙の型宣言
    - テーブルの外部結合（+）

VARIABLE 変数宣言

VARIABLE 変数は、SQL*Plus のプロンプト上で宣言できる変数であり、PL/SQL 内で使用することができます。PostgreSQL の psql のプロンプト上の変数は、「¥set <変数名> <値>」（¥はバックスラッシュ）で定義すれば使えますが、扱えるのは文字列のみです。そのため、移行の手段としては、その変数、および、その変数を利用する処理ロジックを、無名データブロック、あるいは、ストアドファンクションの中で記述することで代替できます。

移行例：



PL/SQL

```
VARIABLE nID NUMBER;
BEGIN
    :nID := 100;
END;
/
```

PL/pgSQL

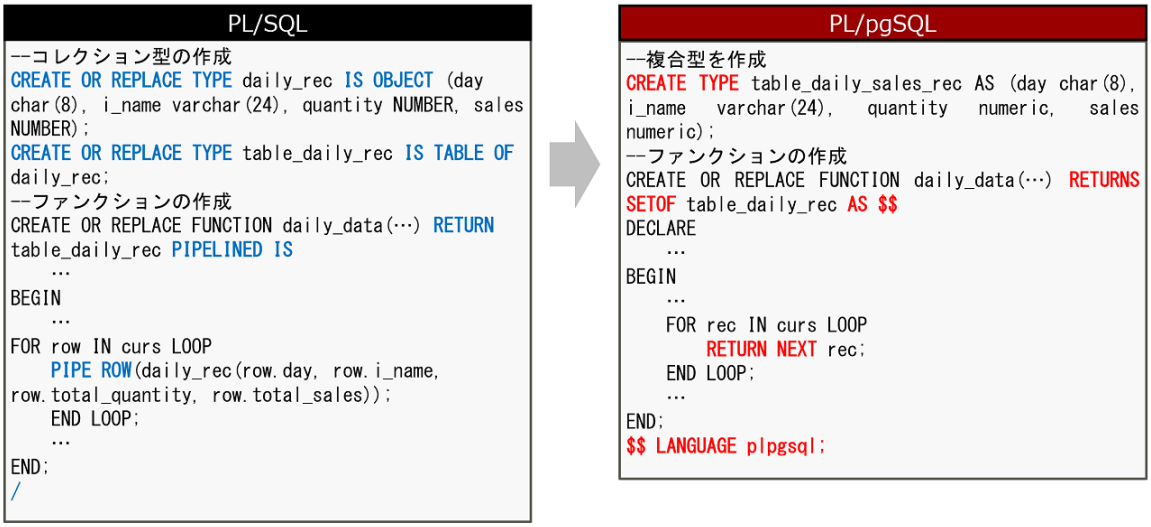
```
DO $$
DECLARE
    nID NUMERIC; <--PostgreSQLで対応するデータ型に置き換え
BEGIN
    nID = 100;
END
$$;
```

コレクション型を戻すストアドファンクション

上記の「2.基本的な構文と呼び出し方の違い」の「ストアド・ファンクション」のところで示したように、単純なデータ型を戻り値として返す場合は、データ型を変更するだけで移行できます。
しかし、ストアド・ファンクションの利用方法として、例えば、ある商品のその日の注文個数や売上額などをレコードとして抽出するような用途で使われることもよくあります。そのようなストアド・ファンクションの戻り値には、必要情報を含む複数のレコードを返すために、コレクション型などが使われます。そこで、ここではコレクション型を戻すストアド・ファンクションを例に挙げて説明します。

Oracle Database では、コレクション型を定義する IS TABLE OF や、その型を定義する OBJECT 型などが使われますが、PostgreSQL には同様の機能が存在しません。そのため、PostgreSQL では別の手段で実現します。IS TABLE OF については、PostgreSQL のファンクションがテーブルを返却する機構として SETOF があるため、これを代用します。OBJECT 型については CREATE TYPE で代用します。

移行例：



PL/SQL

```
--コレクション型の作成
CREATE OR REPLACE TYPE daily_rec IS OBJECT (day
char(8), i_name varchar(24), quantity NUMBER, sales
NUMBER);
CREATE OR REPLACE TYPE table_daily_rec IS TABLE OF
daily_rec;
--ファンクションの作成
CREATE OR REPLACE FUNCTION daily_data(…) RETURN
table_daily_rec PIPELINED IS
    …
BEGIN
    …
FOR row IN curs LOOP
    PIPE ROW(daily_rec(row.day, row.i_name,
row.total_quantity, row.total_sales));
    END LOOP;
    …
END;
/
```

PL/pgSQL

```
--複合型を作成
CREATE TYPE table_daily_sales_rec AS (day char(8),
i_name   varchar(24),   quantity   numeric,   sales
numeric);
--ファンクションの作成
CREATE OR REPLACE FUNCTION daily_data(…) RETURNS
SETOF table_daily_rec AS $$
DECLARE
    …
BEGIN
    …
    FOR rec IN curs LOOP
        RETURN NEXT rec;
    END LOOP;
    …
END;
$$ LANGUAGE plpgsql;
```

トランザクション制御

ストアド・プロシージャを使用している場合は、トランザクション制御を利用できます。
ただし、Oracle Database と PostgreSQL とでは、トランザクション制御自体に非互換があります。まず、その例を説明します。

Oracle Database は、SQL 文が実行されると暗黙的にトランザクションを開始し、COMMIT/ROLLBACK でトランザクションを終了します。トランザクション内の特定の SQL 文がエラーになったとしても、成功した SQL 文はコミットされます。以下の例では、SQL 文 1 および SQL 文 3 がコミットされることを示しています。
なお、コミットしなかった場合、データベースとの接続が切断されたタイミングで暗黙的にコミットされます。

PL/SQL

```
SQL文1;     --暗黙的にトランザクション開始
SQL文2;   ←ここでエラーが発生
SQL文3;     --このSQL文は実行される
COMMIT;     --成功したSQL文1、SQL文3はコミットされる
```

PostgreSQL は、BEGIN 文でトランザクションを開始し、END/COMMIT/ROLLBACK でトランザクションを終了します。なお、BEGIN 文が無い場合は、SQL 文の 1 行ごとに暗黙的にトランザクションをコミットします。トランザクション内で 1 つでも SQL 文がエラーになると、成功した SQL 文も含めてすべての SQL 文がコミットされません。以下の例では、SQL 文 1 がロールバックされることを示しています。
なお、トランザクション終了時やデータベースとの接続が切断されたタイミングで暗黙的にロールバックされます。

PL/pgSQL

```
BEGIN;     --トランザクション開始
SQL文1;
SQL文2;   ←ここでエラーが発生（この後、以降のSQL文が無条件にエラーになる）
SQL文3;     --無条件にエラーになる
END;      --成功したSQL文1がロールバックされる
```

以上のような非互換があるため、ストアドプロシージャを移行する際にもその影響を受けます。以下に注意すべき非互換について説明します。

1) エラー発生時のロールバックについて

Oracle Database は、エラーが発生して EXCEPTION に移動した際に、トランザクション制御を行う必要があります。また、カーソルなどの資源が残っている場合には回収処理が必要です。

PostgreSQL は、エラーが発生すると、トランザクション内で行った処理がすべてロールバックされます。これにより、トランザクション制御やカーソルの資源回収などはシステム側で行われるため、記述する必要がありません。なお、プロシージャの呼び出し元において、エラーが発生した際のリトライ処理があれば、どの SQL 文から再実行するかなども別途見直す必要があります。

PL/SQL

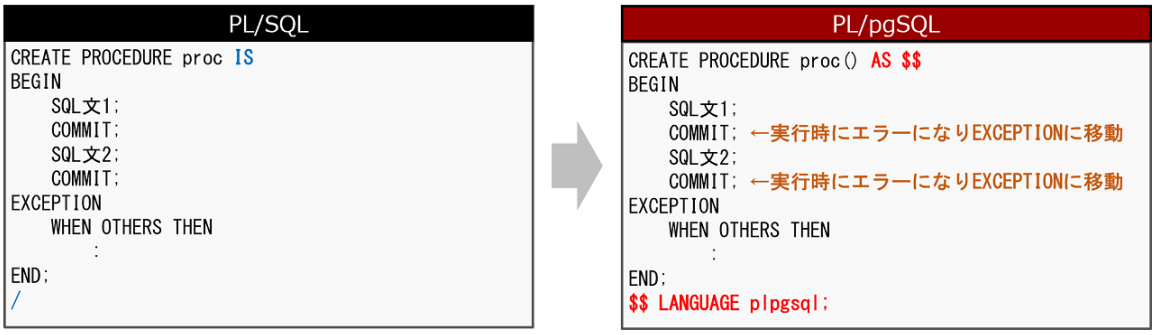
```
CREATE PROCEDURE proc IS
    CURSOR cur IS SELECT ～;
BEGIN
    SQL文1;
    SQL文2; ←エラーが発生するとEXCEPTIONに飛ぶ
    SQL文3;
EXCEPTION
    WHEN OTHERS THEN
    :
    ROLLBACK;  ←明にトランザクション制御が必要
    IF(cur%ISOPEN) THEN  ←資源回収も必要
        CLOSE cur;
    END IF;
    エラー処理;
END;
/
```

PL/pgSQL

```
CREATE PROCEDURE proc() AS $$
DECLARE
    cur CURSOR FOR SELECT ～;
BEGIN
    SQL文1;
    SQL文2; ←エラーが発生するとEXCEPTIONに飛ぶ
    SQL文3;
EXCEPTION
 WHEN OTHERS THEN
    : ←トランザクション制御も資源回収も不要
    エラー処理;
END;
$$ LANGUAGE plpgsql;
```

2) EXCEPTION 句を含むプロシージャ内にトランザクション制御がある場合について

EXCEPTION 句を含むプロシージャ内にトランザクション制御（COMMIT/ROLLBACK）がある場合、Oracle Database と PostgreSQL とでは動作が異なります。Oracle Database では、定義時 / 実行時にエラーになりません。PostgreSQL では、定義時に正常終了しますが、実行時に COMMIT や ROLLBACK でエラーになります。なお、EXCEPTION 句の中には COMMIT/ROLLBACK を記述することができます。



PL/SQL

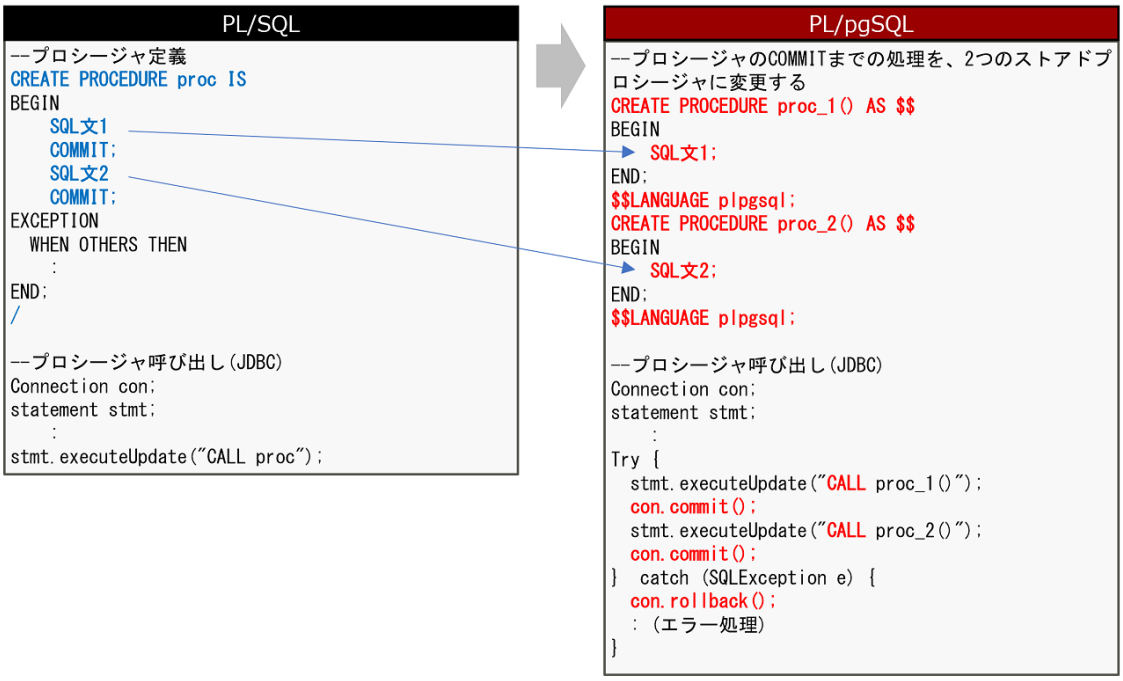
```
CREATE PROCEDURE proc IS
BEGIN
    SQL文1;
    COMMIT;
    SQL文2;
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        :
END;
/
```

PL/pgSQL

```
CREATE PROCEDURE proc() AS $$
BEGIN
    SQL文1;
    COMMIT; ←実行時にエラーになりEXCEPTIONに移動
    SQL文2;
    COMMIT; ←実行時にエラーになりEXCEPTIONに移動
EXCEPTION
    WHEN OTHERS THEN
        :
END;
$$ LANGUAGE plpgsql;
```

対処方法としては、以下の 2 つの方法があります。

- A）COMMIT/ROLLBACK の単位でストアドプロシージャを分割し、上位の呼び元(JDBC など)で COMMIT/ROLLBACK および EXCEPTION を処理します。
  以下の例では、ストアドプロシージャを COMMIT/ROLLBACK の単位で分割し、トランザクション制御（COMMIT/ROLLBACK）を上位の呼び出し元で行っています。



PL/SQL

```
--プロシージャ定義
CREATE PROCEDURE proc IS
BEGIN
    SQL文1
    COMMIT;
    SQL文2
    COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    :
END;
/

--プロシージャ呼び出し(JDBC)
Connection con;
statement stmt;
    :
stmt.executeUpdate("CALL proc");
```

PL/pgSQL

```
--プロシージャのCOMMITまでの処理を、2つのストアドプロシージャに変更する
CREATE PROCEDURE proc_1() AS $$
BEGIN
    SQL文1;
END;
$$LANGUAGE plpgsql;
CREATE PROCEDURE proc_2() AS $$
BEGIN
    SQL文2;
END;
$$LANGUAGE plpgsql;

--プロシージャ呼び出し(JDBC)
Connection con;
statement stmt;
    :
Try {
  stmt.executeUpdate("CALL proc_1()");
  con.commit();
  stmt.executeUpdate("CALL proc_2()");
  con.commit();
}  catch (SQLException e) {
  con.rollback();
  : (エラー処理)
}
```

- B）EXCEPTION でキャッチすべき対象の処理だけを 1 階層深いサブブロックで記述し、その上位で COMMIT/ROLLBACK を処理します。
  詳細は、PostgreSQL 技術インデックス「PostgreSQL でストアドプロシージャを使用する」の「3.1 トランザクション制御に関する制約」において、サブブロック化の説明を参照してください。

3) ストアドプロシージャ内のトランザクション制御がエラーになるケースについて

PostgreSQL について、トランザクションブロック、あるいは、ストアドファンクションから呼び出したストアドプロシージャの中にトランザクション制御（COMMIT/ROLLBACK）を記述することができません。実行時エラーになります。

対処方法の一例として、呼び出し元のトランザクションブロックを無名コードブロックやストアドプロシージャで記述する方法を示しますので、対応を検討してください。

PL/SQL

```
BEGIN;
  CALL proc; ←トランザクションブロックから呼び出す
END;

CREATE PROCEDURE proc IS
BEGIN
    SQL文;
    COMMIT;
END;
/
```

PL/pgSQL

```
DO $$
BEGIN
  CALL proc(); ←無名コードブロックから呼び出す
END
$$;

CREATE PROCEDURE proc() AS $$
BEGIN
    SQL文;
    COMMIT; ←トランザクションブロックから呼び出すと実行時にエラーになるが、無名コードブロックからの呼び出しは実行時エラーにならない
END;
$$ LANGUAGE plpgsql;
```

制御文（GOTO 文）

PostgreSQL では GOTO 文が利用できないため、GOTO 文を利用しないよう、構造化された処理に書き換える必要があります。複数のルートから呼ばれる処理への GOTO 文であれば、その処理をプロシージャ化して呼び出したり、ループ内の処理を抜ける GOTO 文であれば、GOTO 文を CONTINUE 文や EXIT 文に置き換えたりすることで対処します。

移行例：

PL/SQL

```
--条件1,2の処理の後にのみ処理4を実行
--どのルートの処理も最後に処理5を実行
 :
IF <条件1> THEN
    (処理1)
    GOTO ADD_PROC;
ELSIF <条件2> THEN
    (処理2)
    GOTO ADD_PROC;
ELSE
    (処理3)
    GOTO END_PROC;
END IF;
<<ADD_PROC>>
(処理4)
<<END_PROC>>
(処理5)
 :
```

PL/pgSQL

```
--複数ルートから呼ばれる処理4をプロシージャ化
--GOTO文とラベルを削除する
CREATE PROCEDURE add_proc() AS $$
BEGIN
    (処理4)
END;
$$ LANGUAGE plpgsql;

 :
IF <条件1> THEN
    (処理1)
    call add_proc();
ELSIF <条件2> THEN
    (処理2)
    call add_proc();
ELSE
    (処理3)
END IF;
(処理5)
 :
```

制御文（FORALL 文）

PostgreSQL には、FORALL 文はありません。FOR LOOP 文に変更してください。なお、パフォーマンス劣化に注意が必要です。

移行例：

PL/SQL

```
DECLARE
    TYPE nList IS VARRAY(10) OF NUMBER;
    depts nList;
BEGIN
    depts := nList(10,20,30);
    FORALL i IN depts.FIRST..depts.LAST
        INSERT INTO tbl VALUES(depts(i));
END;
/
```

PL/pgSQL

```
DO $$
DECLARE
    depts NUMERIC[10];
BEGIN
    depts := array[10,20,30];
    FOR i IN 1..array_length(depts, 1)
    LOOP
        INSERT INTO tbl VALUES(depts[i]);
    END LOOP;
END $$;
```

- 補足） depts.FIRST や depts.LAST は、Oracle Database のコレクション・メソッド・コールであり、PostgreSQL ではサポートされていません。移行方法としては、先頭のデータは 1 に固定し、最後のデータは array_length 関数を使うことで代替しています。

制御文（FOR LOOP 文）

PostgreSQL でも FOR LOOP 文は利用できます。ただし、FOR LOOP 文で使用する変数がレコード型の場合は、使用するレコード型変数を PostgreSQL の定義に変更する必要があります。また、REVERSE 指定がある場合は、下限値と上限値を入れ替える必要があります。

移行例：

PL/SQL

```
DECLARE
    TYPE tType IS RECORD (<変数> <型>, …);
    <レコード型変数> tType;
BEGIN
    FOR <レコード型変数> IN (SQL文)
    LOOP
        SQL文;
    END LOOP;

    --REVERSE指定がある場合
    FOR <変数> IN REVERSE <下限値>..<上限値>
    LOOP
        SQL文;
    END LOOP;
END;
/
```

PL/pgSQL

```
DO $$
DECLARE
    <レコード型変数> RECORD;
BEGIN
    FOR <レコード型変数> IN (SQL文)
    LOOP
        SQL文;
    END LOOP;

    --REVERSE指定がある場合
    FOR <変数> IN REVERSE <上限値>..<下限値>
    LOOP
        SQL文;
    END LOOP;
END $$;
```

例外処理

EXCEPTION の記述方法に違いはありませんが、対応する PostgreSQL の例外名やエラーコードへの書き換えが必要となる場合があります。

移行例：

PL/SQL

```
EXCEPTION
    WHEN CASE_NOT_FOUND THEN エラー処理1
    WHEN ZERO_DIVIDE THEN エラー処理2
    WHEN OTHERS THEN エラー処理3
END;
```

PL/pgSQL

```
EXCEPTION
    WHEN CASE_NOT_FOUND THEN エラー処理1
    WHEN DIVISION_BY_ZERO THEN エラー処理2
    WHEN OTHERS THEN エラー処理3
END;
```


%ROWTYPE 属性（ファンクションやプロシージャ引数として利用する場合）

PostgreSQL では、ストアドプロシージャやストアドファンクションの引数に、「テーブル名%ROWTYPE」の指定はできません。そのため、複合型で別途定義する必要があります。また、呼び出し元の修正も必要になります。

移行例：

PL/SQL

```
CREATE FUNCTION <ファンクション名>(引数名1 IN/OUT
テーブル名%ROWTYPE, …) RETURN <戻り値の型> IS …;

--呼び出し方法
SELECT <ファンクション名>(引数名1, ・・・) FROM DUAL;
※引数名1がOUTモードのときにはSQL実行時に値が代入される
```

PL/pgSQL

```
--複合型の定義追加
CREATE TYPE <テーブル名>_ROWTYPE AS (カラム名1 データ型1,
…);

CREATE FUNCTION <ファンクション名>(引数名1 IN/OUT <テーブル名>_ROWTYPE, …) RETURNS <戻り値の型> AS $$ …;

--呼び出し方法(引数がINモードのとき)
fRet <戻り値の型>;
SELECT <ファンクション名>(引数名1 IN 型, …) INTO fRet;

--呼び出し方法(引数がOUT(IN OUT)モードのとき)
SELECT * INTO fRet FROM <ファンクション名>(引数名1 OUT 型,
…);
※SQL実行時に、変数fRetに値が代入される
※複合型のカラムデータを参照する際は、変数の前後を括弧で囲む必要がある
```

カーソル属性（%ISOPEN / %NOTFOUND / %FOUND / %ROWCOUNT）

カーソル属性は PostgreSQL にはありません。以下のような代替方法を使って移行を行います。なお、GET DIAGNOSTICS 構文は、直前の実行 SQL 文の処理件数を取得する機構です。

移行例：

PL/SQL

```
--%ISOPEN
OPEN cur;
FETCH cur INTO cur_rec;
IF cur%ISOPEN THEN
    CLOSE cur;
END IF;

--%NOTFOUND
FETCH cur INTO cur_rec;
EXIT WHEN cur%NOTFOUND;

--%FOUND
FETCH cur INTO cur_rec;
IF cur%FOUND THEN
    SQL文;
END IF;

--%ROWCOUNT
FETCH cur INTO cur_rec;
IF cur%ROWCOUNT > 5 THEN
    :
END IF;
```

PL/pgSQL

```
--%ISOPENの代替方法
※エラーハンドリングする例
BEGIN
    CLOSE cur;
EXCEPTION
    WHEN invalid_cursor_name THEN NULL;
END;

--%NOTFOUNDの代替方法
FETCH cur INTO cur_rec;
IF NOT FOUND THEN
    EXIT;
END IF;

--%FOUNDの代替方法
FETCH cur INTO cur_rec;
IF FOUND THEN
    SQL文;
END IF;

--%ROWCOUNTの代替方法
cnt integer := 0;
FETCH cur INTO cur_rec;
GET DIAGNOSTICS int_var = ROW_COUNT;
cnt = cnt + int_var;
IF cnt > 5 THEN
    :
END IF;
```

その他（関連する非互換）

PL/SQL のコード内に記載されるその他の非互換として、よく遭遇するものについて以下に列挙します。

- 長さ 0 の文字列（空文字列）
- 結合（||）
- 暗黙の型宣言
- テーブルの外部結合（+）

長さ 0 の文字列（空文字列）

''（シングルクォーテーションが 2 つ連続）で表現される「長さ 0 の文字列」について、Oracle Database と PostgreSQL では扱いが異なります。Oracle Database では、変数などに長さ 0 の文字列を代入すると、実際には NULL が格納されるため、その変数を判定する場合には、"IS NULL"や、"IS NOT NULL"で判定します。それに対して、PostgreSQL では長さ 0 の文字列と NULL を区別して格納されるため、ここに非互換があります。

そのため、Oracle Database から PostgreSQL に移行する際には、基本的に長さ 0 の文字列('')を設定している箇所において、長さの 0 の文字列を NULL に変更します（なお、判定文の方を書き換える手段もありますが、ロジックを考慮する必要があるため工数が増える可能性があります）。

書き換え例：

参考までに、長さ 0 の文字列が設定される主なケースを以下に示します。

- 代入文 name := '' ;
- ファンクションパラメタ func('abc', 3, '', 10)
- ファンクションパラメタ定義する際のデフォルト値 DEFAULT ''
- ファンクションの復帰値 RETURN ''
- SELECT 文の選択リスト SELECT '' as text,・・・
- INSERT 文の VALUE 句 INSERT ・・・ VALUE('', ・・・)
- UPDATE 文の SET 句 UPDATE ・・・ SET name = '' ・・・

なお、以下の場合はこの非互換とは関係ありません。

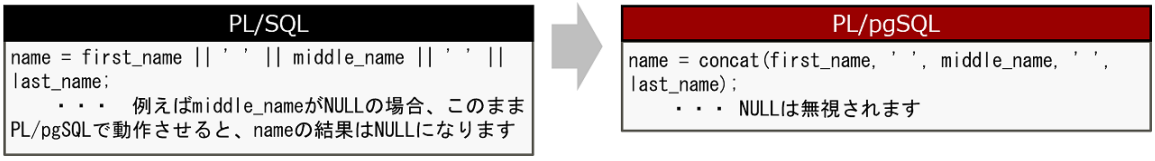
- replace 関数の第 3 引数にある長さ 0 の文字列
- 長さ 0 の文字列の||による文字列連結

結合（||）

処理結果メッセージのテキストを作成する際に、結合（||）を利用してテキストを結合することはよく見られます。
文字列連結の中に NULL が含まれている場合、Oracle Database では NULL を無視しますが、PostgreSQL では 1 つでも NULL が含まれていると結合文字全体が NULL になります。
そのため、結合するテキストが NULL になる箇所の見直しが必要です。

該当箇所が見つかった場合、CONCAT 関数(NULL は無視されます）の利用や、NVL 関数や COALESCE 関数で NULL を抑止することを検討してください。また、結合する数が多い場合などは、連結（||）のオーバーロードを作成し、その中で NULL を抑止する処理を入れる手段もあります。

書き換え例：

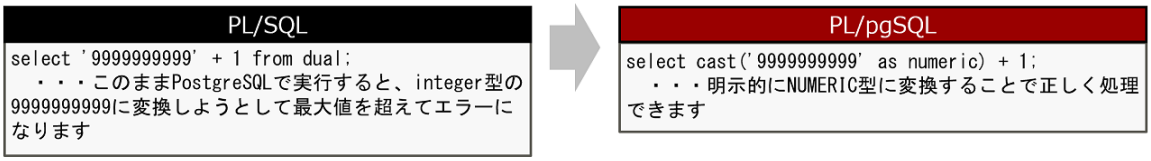


暗黙の型宣言

異なる型どうしの比較や演算などにおいて、Oracle Database では暗黙に、強力な型変換が行われます。
PostgreSQL にも暗黙の型変換の機構がありますが、基本的な型変換のみ行われます。

Oracle Database から PostgreSQL に移行する際に、データ型が合っていない旨のエラーが多くでることが良くあります。
そのため、異なる型どうしの比較や演算を行っている箇所について、明示的な型変換、あるいは、CREATE CAST を利用して新たな型変換を定義するなどの対処が必要になります。

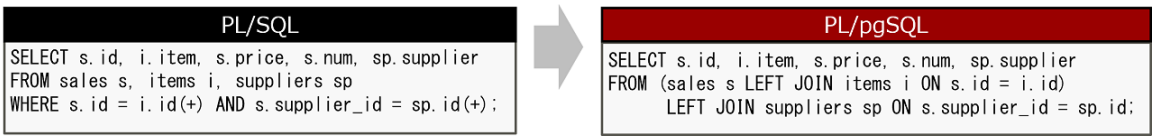
書き換え例：



テーブルの外部結合（+）

PostgreSQL は、外部結合（+）に対応していません。
LEFT OUTER JOIN/RIGHT OUTER JOIN 句を用いた結合に書き換えてください。

書き換え例：



参考

PostgreSQL は、他のデータベースからの移行を考慮した改良が行われています。PL/pgSQL の関連において、以下の互換性対応があります。

- PostgreSQL のパラメーター定義は「モード/パラメタ名/データ型」だが、互換性対応のため「パラメタ名/モード/データ型」も可能
- PostgreSQL 14 からプロシージャやファンクションのパラメーター定義に OUT モードをサポート

Oracle Database の PL/SQL から PostgreSQL の PL/pgSQL への移行について、概要と移行例を説明しました。本記事を活用して、安全かつスムーズに移行作業を進めて頂ければ幸いです。

2022 年 11 月 18 日