PL/SQL による SQL の実行：ベスト・プラクティスとワースト・プラクティス

Oracle ホワイト・ペーパー

2008 年 9 月

ORACLE

ご注意

本書は、弊社の一般的な製品の方向性に関する概要を説明するものです。また、情報提供を唯一の目的とするものであり、いかなる契約にも組み込むことはできません。下記の事項は、マテリアルやコード、機能の提供を確約するものではなく、また、購買を決定する際の判断材料とはなりえません。オラクルの製品に関して記載されている機能の開発、リリース、および時期については、弊社の裁量により決定いたします。

www.oracle.com/technology/tech/pl_sql/pdf/doing_sql_from_plsql.pdf

PL/SQL による SQL の実行：ベスト・プラクティスとワースト・プラクティス

www.oracle.com/technology/tech/pl_sql/pdf/doing_sql_from_plsql.pdf

PL/SQL による SQL の実行：ベスト・プラクティスとワースト・プラクティス

概要

PL/SQL 開発者には、SQL 文を実行するための構成に多数の選択肢があります。そして、その領域には、埋込み SQL やネイティブ動的 SQL、DBMS_Sql API、あるいはバルクまたは非バルク、暗黙カーソル、パラメータ化された明示カーソル、または参照カーソルといった複数のディメンションが含まれる可能性があります。このため、どれを使用するかを決めるのが困難です。また、新しいバリアントが導入されたことによって、最適な選択をするために古いバリアントを停止する場合もあります。Oracle Database 11g は、動的 SQL の領域を改善することで、従来の方法を維持します。

本書は、PL/SQL により SQL を実行するユースケースを調査して分類し、Oracle Database 11g の観点から、現行の課題に対する最適なアプローチを説明します。

Oracle Corporation 発行「Doing SQL from PL/SQL:Best and Worst Practices」の翻訳版です。

最新版を参照している必要があります。各ページの上部にある URL にアクセスして確認してください。

はじめに

本書は、データベース PL/SQL ユニットのプログラミングに精通し、とくに SQL 文を処理するすべての PL/SQL の方法について多少経験がある Oracle Database 開発者を対象としています。このため、各方法のバリアントの説明や確認はおこないません。むしろ、予備知識があることを前提とすることで、理解していないために説明が必要な部分を認識できます。読者はこのようにして、すべてのベスト・プラクティスの原則の基本となる正しい概念を理解できるようになります。

わかりやすく例をあげましょう。成人が外国語を習得していく場合、伝える内容を徐々に正確に表現できるようになります。しかし、言語の仕組みを深く理解せずに熟語を使用していることがあります。熟語の使い方が不適切なために、内容を誤解されてしまうこともあります。このような問題の解決策は、文法のルールとそのルールに準拠している文の意味を積極的に理解しようとすることです。

5 ページの"*埋込みSQL*、*ネイティブ動的SQL*、*DBMS_Sql API*"の項では、SQL文を処理するPL/SQLの 3 つの方法について、その概要を示しています。

*select*文は、アプリケーション・コード[1]から発行されるSQL文のうちもっとも頻繁に使用されるものです。30 ページの"*select 文のアプローチ*"の項では、次のようにユースケースを分類します。

- 結果セット・サイズが大きい場合の複数行の選択。結果セット・サイズが適度な制限を超えないことを想定できる場合の複数行の選択。単一行の選択。
- コンパイル時に SQL 文の固定が可能であること。コンパイル時にテンプレートの固定が可能であること（ただし、実行時まで表の名前を指定しないこと）。実行時に *select list*、*where* 句、または *order by* 句の構築が必要であること。
- SQL 文の仕様のカプセル化が可能であること。その結果のフェッチおよび単一の PL/SQL ユニットの結果に適用される、後続処理または異なる PL/SQL ユニットの結果の処理へ実装する必要があること。

表データを変更するSQL文は、*select*文に続いてもっとも一般的です。47 ページの"*insert*、*update*、*delete*、*および merge 文のアプローチ*"の項で、これらの文を取り上げます。

埋込み SQL がサポートしていない、*lock table* 文、トランザクション制御文、そのほかすべての SQL 文は、プログラミングへの影響が少ないため、ここでは取り上げません。

59 ページの"*ユースケースの例*"の項では、一般的に発生するシナリオを考察し、要件を実現するための最適なアプローチについて説明します。

本書には、ベスト・プラクティスの原則について、いくつか記載されています。すぐに参照できるように、65 ページの"*付録B：ベスト・プラクティスの原則のまとめ*"で再現[2]します。

本書では、*新たに*コードを記述する場合に使用される最適なアプローチを提供します。コードの刷新プロジェクトを正当化するものではありません。

---

1. ここでのアプリケーション・コードの定義では、インストールおよびアップグレードされるスクリプトを除いています。
2. 本書は、Adobe Framemaker 8.0 で作成しています。相互参照機能によって、ソース・パラグラフのテキストを移動先の参照に挿入できます。このため、まとめに記載されているベスト・プラクティスの原則の記述が本文の記述と同じであることが確認できます（ただし、フォントは維持されません）。

注意事項

3GL プログラミングのベスト・プラクティスの原則を規定することは、非常に困難です。以下のような資質を備えた読者にとって、安全性というテーマは非常に困難な課題の 1 つです。

- 理想の"両親"[3] をもっている
- 非常に発達した言語能力とともに一般的な常識がある
- 機械システムを視覚化する能力がある
- 自身や他人の卓越性を追及する
- 一流の交渉力がある（優れたコードは、悪いコードよりも記述およびテストに時間がかかります。マネージャーは、厳しい時間枠の中でコードの配信を要求します。）
- 一流の教育を受けている
- 優れた技術文書を記述できる（コードの要件の記述、テスト仕様の記述、およびその過程で発生する問題の説明が必要となるためです。）

読者は、次のような知的支援を受けられる環境で作業できます。

- 1 人以上の優れた指導者が身近にいる

本書のテーマに関して、ベスト・プラクティスの原則を取り入れて直感的に使用するには、次の条件を満たす必要があります。

- Oracle Database を十分理解している
- PL/SQL を十分理解している

本書の定期的な改訂

本書は、スペルミスや文法エラーが存在する可能性もあるため、定期的[4] に改訂しています。また、顧客とのユースケースの継続的なディスカッションによって、新しいベスト・プラクティスの原則が作成されることもあります。本書を学習する前に、ページのヘッダーにあるURLにアクセスし、最新版かどうかを確認してください。

URL は変更される場合がありますが、次の Oracle Technical Network にある PL/SQL Technology Center にかならず接続します。

http://www.oracle.com/technology/global/jp/tech/pl_sql/index.html

本書が別の URL へ移動していても、このページで簡単に確認することが可能です。

---

3. 著者の母国語は、イギリス英語です。著者の文化的な背景により、重要な点を強調するためにユーモアをさりげなく用いる傾向があります。イギリス英語を母国語としない読者の方々は、この傾向を念頭において読むことが望まれます。
4. 改訂履歴は本書の最後に記載されています。64 ページの"付録A：変更履歴"を参照してください。

埋込み SQL、ネイティブ動的 SQL、*DBMS_Sql* API

PL/SQL では、SQL を発行する 3 つの方法をサポートします。この項では概要を説明し、各方法をすでに実行した読者が高く評価した点を示しています。特定の要件に合わせて最適な方法を選択するには、3 つのすべての方法を正しく理解する必要があります。

埋込み SQL

PL/SQLの埋込みSQL[5]では、PL/SQL文で直接SQL構文を使用できるので、操作が非常に簡単です[6]。*select*、*insert*、*update*、*delete*、*merge*、*lock table*、*commit*、*rollback*、*savepoint*、*set transaction*といったSQL文のみをサポートします。

通常、PL/SQLの埋込みSQL文の構文は、対応するSQL文の構文と同じです[7]。ただし、*select... into*文、*update... set row...*文、*insert... values Some_Record*文（*Code_3*を参照）には、PL/SQL固有の構文があります[8]。

埋込みSQL文には、その通常のSQL文に対して重要な意味上の利点があります。通常のSQL文でプレースホルダを使用できる場所にPL/SQL識別子を使用できます。6 ページの"*埋込みSQL文の名前解決*"で詳しく説明します。

*Code_1*に、簡単な例を示しています。

```
-- Code_1
for j in 1..10 loop
  v1 := f(j);
  insert into t(PK, v1) values(j, b.v1);
end loop;
commit;
```

ここでの *b* は、変数 *v1* が宣言されたブロックの名前です。

---

5. 本書では、後述する理由に従って、一般的な*静的* SQL ではなく*埋込み* SQL という用語を使用しています。

6. このため、"PL/SQL は SQL に対するオラクルのプロシージャ拡張機能"と呼ばれます。ただし、簡略した表現であり、正確ではありません（過去を考慮して、ストアド PL/SQL ユニットの前に特殊な SQL 文の無名ブロックが導入されました）。次のような表現が適切です。PL/SQL は、SQL コマンドのシームレスな処理を実現するために設計された命令型の 3GL です。そのため、PL/SQL には特別な構文が用意されており、SQL とまったく同じデータ型もサポートされています。

7. SQLには文のシーケンスという概念がないので、SQL文は特殊な終了文字を必要としません。対照的に、PL/SQLユニットは多くの文で構成され、それぞれ最後にセミコロンを使用します。最後にセミコロンを必要とするため、PL/SQLの埋込みSQL文は通常のSQL文と異なります。SQL*Plusスクリプト言語では、SQL*Plusコマンドと組み合わせたSQL文のシーケンスをサポートしているため、初心者は混乱することがあります。そのため、スクリプト言語で各SQL文を終了する特殊な文字が必要になります。デフォルトはセミコロンですが、`SET SQLTERMINATOR`コマンドで上書きされる場合があります。本書のすべてのSQL*Plusスクリプトの例は、`SET SQLTERMINATOR OFF`の発行後に実行されます。初心者は*execute immediate*の引数になるテキストでSQL文の最後にセミコロンを記述してしまう傾向がありますが、これは間違いです。

*Code_2*は、SQL文の処理を簡潔かつ正確にするPL/SQLの言語機能を明確に示しています[9]。これは*暗黙カーソルFORループ*[10]と呼ばれます。

```
-- Code_2
for r in (
  select   a.PK, a.v1
  from    t a
  where    a.PK > Some_Value
  order by a.PK)
loop
  Process_One_Record(r);
end loop;
```

*Code_3*は、PL/SQL固有の構文を含む埋込みSQL文を示しています。

```
-- Code_3
<<b>>declare
  Some_Value t.PK%type := 42;
  The_Result t%rowtype;
Begin
  select   a.*
  into     b.The_Result
  from    t a
where    a.PK = b.Some_Value;

  The_Result.v1 := 'New text';

  update  t a
  set  row = b.The_Result
  where  a.PK = b.The_Result.PK;

  The_Result.PK := -Some_Value;
  insert  into t
  values  The_Result;
end;
```

埋込み SQL 文の名前解決

PL/SQL コンパイラは、埋込み SQL 文を検出すると、次のように処理します。

- 分析するためにSQLサブシステムに渡します[11]。SQLサブシステムは、構文的に正しいことを確認し（正しくない場合はPL/SQLユニットのコンパイルに失敗します）、*from list*項目の名前を検出して、そのスコープでほかの識別子を解決します。
- SQL 文のスコープで識別子を解決できない場合、"エスケープ"し、PL/SQL コンパイラが解決します。最初に、現在の PL/SQL ユニットのスコープで試行されます。失敗した場合、スキーマのスコープで試行されます。これにも失敗した場合は、PL/SQL ユニットのコンパイルは失敗します。

---

8. *where current of Cur* 構文（*Cur* は*明示カーソル*）も通常の SQL にはありません。ただし、本書で推奨されているベスト・プラクティスの原則に従った場合、これは必要ありません。

9. 本書のコードの例では、table tテストを使用します。これを作成するコードは、72 ページの"付録D：テスト・ユーザーUsrおよびテスト表Usr.t(PK number, v1 varchar2(30), ...)の作成"の項に記載されています。

10. この簡単な構造が本番コードとして推奨されていないことが30 ページの"*複数行の選択 - アンバウンド結果セット*"の項で確認できます。

11. *select... into* 文では、SQL サブシステムに文が渡される前に、PL/SQL コンパイラが *into* 句を削除します。*set row* といったそのほかの埋込み SQL 文の PL/SQL 固有の構文でも同様です。

- コンパイル・エラーがない場合、PL/SQL コンパイラは、通常の SQL 文と同じテキストを生成し、生成されたマシン・コードで保存します。PL/SQL ユニットのスコープで解決された識別子を埋込み SQL 文で使用した場合、この文ではプレースホルダを使用します。
- 実行時に、適切な呼出しがおこなわれ、通常の SQL 文をパース、バインド、および実行します。バインド引数は、式に結合されるエスケープ PL/SQL 識別子によって提供されます。*select* 文では、指定した PL/SQL ターゲットに結果がフェッチされます。

*Code_4*は、*Code_1*に対して生成された通常のSQL文[12]を示しています。

```
-- Code_4
INSERT INTO T(PK, V1) VALUES(:B2 , :B1 )
```

*Code_5*は、*Code_3*に対して生成された通常のSQL文を示しています。

```
-- Code_5
SELECT A.* FROM T A WHERE A.PK = :B1

UPDATE T SET "PK" = :B1 ,
             "N1" = :B2 ,
             "N2" = :B3 ,
             "V1" = :B4 ,
             "V2" = :B5 WHERE PK = :B1

INSERT INTO T VALUES (:B1 ,:B2 ,:B3 ,:B4 ,:B5 )
```

PL/SQL コンパイラを使用することで、どの程度作業が軽減されるかを確認します。*The_Result* レコードの構造と表の列形式を検出して、通常の SQL 文のテキストを適宜生成します。呼出し場所の異なる埋込み SQL 文が共有プールで同じ構造を共有する確率を高めるために、大/小文字および空白文字を標準化します。/*+ ... */ ヒント構文を使用している場合のみ、埋込み SQL のコメントが、生成された SQL に保存されます。

生成されたSQL文のバインド引数として機能するPL/SQL識別子は、変数または仮パラメータを表しています。ただし、埋込みSQLを含むPL/SQLユニットにのみ表示されるファンクションは表しません。これは、SQL文のファンクション呼出しに対して定義されたセマンティックから生じます。各行に対し、そのファンクションは、SQL実行サブシステムで評価されなければなりません。このため、埋込みSQL文では、スキーマ・スコープでアクセスできるファンクションのみ使用できます。*Code_1*は、この対処方法を示しています。

---

12. 次のような問合せを使用して、*Code_4*および*Code_5*を検出します。

```
select  Sql_Text
from    v$Sql
where   Lower(Sql_Text) not like '%v$sql%'
and    (Lower(Sql_Text) like 'select%a.*%from%t%' or
        Lower(Sql_Text) like 'update%t%set%' or
        Lower(Sql_Text) like 'insert%into%t%')
```

読みやすくするために、*Code_5*は手動でフォーマットされています。

名前取得、ファイングレインな依存性の追跡、および防衛的プログラミング

*Code_3*の識別子は、修飾子が過剰に付加されているように見えます。同じ機能を実装する*Code_6*が示すように、一般的な方法とは対照的です。

```
-- Code_6
select  v1
into  l_v1
from  t where PK = p_PK;
```

プログラマーによっては、このステータスを示す接頭辞または接尾辞の規則で、ローカル変数や仮パラメータの名前をつけるスタイルをすでに使用しています（*in*、*out*、*in out*などの異なるパラメータ・モードを区別した名前を使用するプログラマーもいます）。通常、名前取得のリスクがないスタイルが要求されます。ここで問題になるシナリオは、*Code_6*の表*t*が変更されて*c1* などの列が追加されることです。SQLコンパイラは最初に*c1* の解決を試み、解決されない場合のみPL/SQLコンパイラへのエスケープを許可するため、新しい*c1* 列の追加によってこのエスケープを停止することで、*Code_6*の意味が変更される可能性があります。これは、エスケープのあとに識別子*c1* が解決されたかどうか（つまり、埋込みSQL文ですでに使用されたかどうか）に依存します。すでに使用されている場合、*c1* という名前はSQLで取得されます。*c1* が識別されない場合、PL/SQLユニットを再コンパイルして正確性を証明する必要があります。

表の列を示す各識別子が表の別名で修飾されている場合、および PL/SQL ユニットのスコープで解決される各識別子が宣言されているブロックの名前で修飾されている場合は、不正確な要素はありません。

以上の説明から、プログラミング・スタイルで意図した目的を実現することが保証できないということは明白です。今回の例では、表 *t* を変更して、*p_PK* という列を追加する場合があります。開発現場が、*p_*または *l_*で始まる名前のスキーマ・レベルの表の列を禁止するといった幅広いネーミング規則を要求する場合のみ、そのプログラミング・スタイルは有効です。ただし、PL/SQL コンパイラでは、このような人為的に規制されたルールを信頼しません。これに関しては、ファイングレインな依存性の追跡を導入している Oracle Database 11g ではさらに重要になります。

以前は、オブジェクト全体の粒度で依存性の情報が記録されました。*Code_6*の例では、現在のPL/SQLユニットが表*t*に依存していることが記録されます。そしてOracle Database 11gでは、PL/SQLユニットが表*t*内の*t.v*および*t.PK*列に依存していることが記録されます。新しいアプローチでは、依存に無関係な方法で参照オブジェクトが変更される場合は、不要な無効化を避け、減少させることを目標としています。今回の例では、経験の浅いプログラマーは、依存するPL/SQLユニットが*t*の特定の名前の列のみを参照する場合に*t*へ新規に列を追加することは無関係であると、まず考えるかもしれません。しかし、名前取得に関しては、常にそうとは限りません。新しい列の名前は、PL/SQLスコープで解決されたエスケープ識別子と競合する場合があります。変更された表の構造で正しい意味をもつPL/SQLを保証する唯一の方法は、再コンパイルして新しく名前の解決が実行されるように、新しい列の追加に応じて無効化することです。

*Code_3*のように修飾名を使用すると、分析内容が変更されます。修飾識別子*b.PK*は、問合せでの別名が*a*である表の列（既存または新規）を意味するものではありません。これは、テストで簡単に確認できます。列*PK*および*v*を使用する表*t*、*Code_3*を含むプロシージャ*p1*、および*Code_6*を含むプロシージャ*p2* を作成します。*User_Objects*問合せで両方が有効なことを確認します。*t*を変更して列（*number*データ型の*c1 など*）を追加し、*User_Objects*問合せを繰り返します。*p1* が有効で*p2* が無効になります[13]。

最後に、*Code_7*で反例を検討してみます。

```
-- Code_7
<<b>>declare
  Some_Value t.PK%type := 42;
  The_Result t%rowtype;

begin
  select   b.*
  into     b.The_Result
  from    t b
  where    b.PK = b.Some_Value;

  DBMS_Output.Put_Line(The_Result.n1);
end;
```

ここでは、表 *t* の別名が PL/SQL ブロックの名前と競合します。*t* に *Some_Value* 列がありませんが、このコードは正しく動作します。ただし、このような列が導入された場合、問合せが意図しなかった意味に変更されます。つまり、コードが名前取得の影響を受けて、ファイングレインな依存性の追跡を活用できなくなるのです。

PL/SQL コンパイラによる埋込み SQL 文の処理（とくに、名前取得のリスクが発生した場合の処理）を理解すると、次のベスト・プラクティスの原則の論理的根拠がわかります。

埋込み SQL では、from list 項目の別名で各列の名前をドット修飾します。宣言するブロックの名前で各 PL/SQL 識別子をドット修飾します。

Principle_1

埋込み SQL 文を記述する場合、常に各 *from list* 項目の別名を確認し、適切な別名で各列を修飾します。現在の PL/SQL ユニットで解決する各識別子の名前は、宣言されているブロックの名前で常に修飾します。これがブロック文の場合、ラベルによる名前が使用されます。別名および PL/SQL ブロックの名前はすべて一意である必要があります。これによって、参照される表が変更される場合に名前取得を回避するため、ファイングレインな依存性の分析で PL/SQL ユニットを無効化する必要がないという結論に達する可能性が高くなります。

PL/SQL プログラムで発行されるすべての SQL は動的 SQL

実行時に、各埋込みSQL文から生成されたSQL文は、Oracle DatabaseのSQLサブシステムでサポートされている方法でのみ実行されます。まず、*セッション・カーソル*[14]が開き、SQL文がテキストとして表示され、パースされます。*select*文では、*select list* 要素のターゲットが定義されます。SQL文にプレースホルダがある場合は、バインド引数がバインドされます。続いて、*セッション・カーソル*が実行されます。

---

13. *p2* は、コードを変更せずに簡単に再度有効にできます。ただし、その有効化によって意味が変わる可能性があります。*p1* の意味は変更できません。

14. この専門用語の定義は、19 ページを参照してください。

*select*文では、結果がフェッチされます。最後に、*セッション・カーソル*が閉じます[15]。

PL/SQL プログラムが動的 SQL を使用する場合も、SQL 文の実行時処理は同じです。違いは、何が実行時コードの生成に必要な処理をおこなうかです。ネイティブ動的 SQL（とくに *DBMS_Sql* API）の場合はプログラマーが、埋込み SQL の場合は PL/SQL コンパイラで、それぞれ処理します。

経験の浅いプログラマー（とくに、動的SQLは初めてで、埋込みSQLの経験は過去にあるプログラマー）がこのことを理解しているとは限りません。たとえば、埋込みSQL文が正しくコンパイルされれば、生成されるSQL文は実行時に失敗[16]するはずがないなどと考えます。これは、少し考えると誤りであることがわかります。実行者権限のPL/SQLユニット[17]が実行される場合、*Current_Schema*が*Session_User*（実行者権限のPL/SQLユニットだけがコール・スタックに存在する場合）または定義者権限のPL/SQLユニット[18]あるいはスタックにもっとも近いビューの*Owner*に設定されます。これは、たとえば、生成されたSQL文内の修飾されていない識別子が、コンパイル時ではなく実行時に個別に解決される場合があることを意味しています[19]。最悪の場合、実行時エラーの*ORA-00942: table or view does not exist*が発生する可能性があります[20]。

プログラマーの認識よりもさらに表現が豊かな埋込み SQL

埋込み SQL を使用していれば SQL の要件に対応できる場合でも、ネイティブ動的SQLまたは*DBMS_Sql* APIを使用しているコードを目にすることがあります(とくに、コードを記述したのが初心者で、経験を積んだ PL/SQL プログラマーがコードを確認していない場合）。コードの表現が十分であれば、埋込み SQL は使用しないという理由は説得力に欠けています。このように埋込み SQL を回避することは、明らかにワースト・プラクティスと言えます（次の項で、この件について触れていきます）。

---

15. 実行時まで *select list* が不明で直接 PL/SQL プログラムで構築されていない場合、追加の手順が必要です。PL/SQL プログラムは、SQL システムに *select list* を要求します。これには、*DBMS_Sql* API の使用が必要です。ただし、このユースケースは、本番 PL/SQL プログラムでは非常にまれです。このアプローチを提案する実装設計は慎重に検討する必要があります。

16. ここでの失敗の概念は、insert の Dup_Val_On_Index 例外や、select... into の No_Data_Found および Too_Many_Rows 例外などのデータ駆動型の条件を除外して、定義しています。

17. 実行者権限の PL/SQL ユニットは、*authid* プロパティと *Current_User* が同じです。

18. 実行者権限の PL/SQL ユニットは、*authid* プロパティと *Definer* が同じです。

19. このため、実行者権限の PL/SQL ユニットを正しくコンパイルするには、PL/SQL ユニットの *Owner* が所有するスキーマのテンプレート・オブジェクトが必要になる場合があります。名前解決のルールおよび PL/SQL のコンパイル時に使用される権限確認のルールは、実行者権限および定義者権限の PL/SQL ユニットで同じです。

20. 定義者権限の PL/SQL ユニットは、*public* に明示的に付与される権限とともに、*Owner* に明示的に付与される権限を常に確認します。実行者権限の PL/SQL ユニットは、コール・スタックの状態に依存する権限を確認します。定義者権限の PL/SQL ユニットがスタックに存在する場合、実行者権限の PL/SQL ユニットは、定義者権限の PL/SQL ユニットまたはスタックにもっとも近いビューと同じ権限を確認します。実行者権限の PL/SQL ユニットだけがコール・スタックに存在する場合、*Current_User* の権限を直接確認するか、現在使用できるすべてのロールを介した *Current_User* の権限とともに *public* を介した *Current_User* の権限を確認します。このため、実行者権限の PL/SQL ユニットがスキーマ修飾名を使用してオブジェクトを識別する場合でも、実行時に *ORA-00942* が発生する可能性があります。

ネイティブ動的 SQL

Code_8 では、ネイティブ動的SQLの簡単な例を示しています。名前に*ネイティブ*という用語を使用しているのは、PL/SQL言語機能として実装されているためです。

```
-- Code_8
procedure p authid Current_User is
  SQL_Statement constant varchar2(80) := q'[
   alter session
    set NLS_Date_Format = 'dd-Mon-yyyy hh24:mi:ss'
  ]';
begin
  execute immediate SQL_Statement;
  DBMS_Output.Put_Line(Sysdate());
end p;
```

*alter session*のSQL文は埋込みSQLでサポートされていないので、*Code_9*に示されているようにPL/SQLコンパイラでSQL文を分析しない方法を使用する必要があります。*動的SQL*という用語は、このような方法を表すために広範に使用されますが、"動的"という言葉が誤解を招く可能性があります。この方法で実行されるSQL文のテキストが実行時に構築される*可能性がある*のでこの用語が選択されたのですが、実際はかならずしもそうではありません。*Code_8*では、コンパイル時にSQL文が固定されます。これは、*constant*キーワードを使用することで強調されます。

汎用的な PL/SQL ベスト・プラクティスの原則は、次のとおりです。

ブロックによって意図的に変更されない限り、constant キーワードで各 PL/SQL 変数を宣言します。

Principle_2

初期化のあとに変更されない変数の宣言で*constant*キーワードを使用します[21]。最悪のケースでも*constant*の変更を試みるコードがコンパイルに失敗するため（このエラーによってプログラマーの考え方が鋭くなります）、この原則に従ってもペナルティはありません。この原則は、可読性と正確性という点で明白な利点が得られます[22]。

動的SQLのコンテキストでは、可能であれば、*constant*としてSQL文のテキストを宣言すると有用です。このようにすることで、SQLインジェクション攻撃[23]の領域が削減されます。

このコネクションで、*p* の *authid* プロパティが *Current_User* に明示的に設定されます。*authid* 句が省略された場合、プロパティのデフォルト値の *Definer* が使用されます。

21. PL/SQL コンパイラがこの事例を検出する場合があります（ただし、パッケージ仕様のグローバル・レベルで変数が宣言される場合を除く）。エンハンスメント・リクエスト 6621216 では、この事例にコンパイラの警告を要求します。

22. この情報がないと安全ではないと判断されてしまう場合に、ある状況下で *constant* キーワードを使用することによって、特定の最適化は安全であるということが PL/SQL コンパイラに通知されます。

この動作は、履歴によって決定します。互換性を理由に変更することはできません。ただし、プログラマーは、プロパティのデフォルト値を無視して、次のベスト・プラクティスの原則を採用できます。

常に *auth*id プロパティを明示的に指定します。Current_User または Definer を慎重に選択します。

Principle_3

各PL/SQLユニットの*authid*プロパティを常に明示的に指定します。ユニットの目的を慎重に分析したあと、定義者権限または実行者権限を選択します[24]。

Code_9 では、ネイティブ動的SQLの反例を示しています。

```
-- Code_9
procedure p(Input in varchar2) authid Current_User is
  SQL_Statement constant varchar2(80) := 'Mary had... ';
begin
  execute immediate SQL_Statement||Input;
end p;
```

このバージョンのプロシージャ *p* は、エラーなくコンパイルされます。ただし、実行時に *ORA-00900: invalid SQL statement* エラーで失敗します（この例で使用されている実引数は関係ありません）。これは、明白な結果を抽出して、明白ではない結果を議論するコンテキストを提供するための極端な方法です。議論によって、ベスト・プラクティスの原則のコメントが動機づけられます。

実行時までSQL文は分析されません。これは、埋込みSQLが対応していないSQL文のサポートを可能にする方法の特性です。通常、SQL文のテキストは、実行時までわかりません。そのため、実行されるSQL文がコンパイル時に存在せずに、コンパイルとSQL文の実行の間に作成されるオブジェクトにアクセスする場合があります。グローバル一時表[25]を導入する前に、PL/SQLプログラムで大量の一時データのオーバーフローに対応するスクラッチ表を使用している場合があります。単純な実装では、動的SQLでスクラッチ表を作成し、あとで削除します。高度な実装では、管理プールからこのような表の名前を保存し、あとで削除します。いずれの場合も、プログラムが実行時まで表の名前を識別できないので、埋込みSQLでサポートされる通常のSQL文でも動的SQL文が必要になります。*Code_10*は、定型化された一般的な例を示しています。

---

23. SQLインジェクションの詳細については、このホワイト・ペーパーでは取り上げていません。ここでは、コンパイル時にテキストが固定されるSQL文のみを発行するPL/SQLプログラムで脅威を避けることができるということ、そして、*constant* SQL文テキストを使用する埋込みSQLおよび動的SQLにこの特性があるということを指摘するに留めておきます。SQLインジェクションに対する脆弱性がないことを確認するのは、SQLを発行するコードを記述したPL/SQLプログラマーの責任です。参考資料のホワイト・ペーパー『*How to write injection-proof PL/SQL*』は、Oracle Technology Network の次の URL で確認してください。
www.oracle.com/technology/tech/pl_sql/how_to_write_injection_proof_plsql.pdf
詳細まで確認することを強く推奨します。

24. エンハンスメント・リクエスト 6522196 は、*authid* プロパティが明示的に指定されていない場合にコンパイラの警告を要求します。これが Oracle Database 11g Release 2 に実装されています。

25. サポートされているすべてのバージョンの Oracle Database でグローバル一時表がサポートされます。

```
-- Code_10
procedure b(The_Table in varchar2, PK t.PK%type)
  authid Current_User
is
  Template constant varchar2(200) := '
 select a.v1 from &&t a where a.PK = :b1';

  Stmt constant varchar2(200) := Replace(
 Template, '&&t',
 Sys.DBMS_Assert.Simple_Sql_Name(The_Table));

  v1 t.v1%type;
begin
  execute immediate Stmt into v1 using PK;
  ...

end b;
```

*constant*テンプレート[26]および派生した*constant* SQL文の使用によって、コードの正確性が増します。

本書の目的はベスト・プラクティスとワースト・プラクティスであるため、例以外のコード（ただし、反例を除く）を記載する余裕がありません。そのため、説明には *Sys.DBMS_Assert.Simple_Sql_Name()* を使用します。名前は、汎用的な SQL および PL/SQL のベスト・プラクティスの原則に従って修飾されます。

Owner を使用して、Oracle Database に付属しているオブジェクトの名前をドット修飾します。

Principle_4

Oracle Database に付属するオブジェクトの参照は *Owner* でドット修飾されます（*Sys* になる場合が多いですが、かならずそうなるわけではありません）。これによって、目的のオブジェクトと名前が競合するローカル・オブジェクトが、名前の解決時に現行のスキーマに作成される場合でも、意図したとおりに保存されます。

実引数が適切な修飾されていないSQL識別子の場合、SQL文のとおりに記述され、ファンクションで入力が返されます。返されない場合は、*ORA-44003: invalid SQL name*エラーが発生します。*Code_11*の*b()*の呼出しがエラーなく実行され、期待された結果が生成されます。

```
-- Code_11
b('t',  42);
b('T',  42);
b('"T"', 42);
```

*Code_12*の呼出しがORA-44003 で失敗します。

```
-- Code_12
b('"USR"."T"', 42);
```

Code_13 では、要求するSQL文を示しています。

```
-- Code_13
select a.v1 from "USR"."T" a where a.PK = :b1
```

この文は適切ですが識別子が修飾されます。そのため、修飾されないとアサーションは失敗します。

---

26. SQL 文テンプレートの&&t 表記に形式的な意味はありません。テンプレートの概念を詳細に説明している参考資料のホワイト・ペーパー『*How to write injection-proof PL/SQL*』で使用されます。SQL を実行する Oracle Database のスキームを使用して、値をプレースホルダにバインドできます。ただし、文の識別子に対応する機能はありません。

*Code_14*の呼出しもORA-44003 で失敗します。

```
-- Code_14
b(  't a
 where 1=0
 union
 select Username v1
 from All_Users where User_ID = :b1 --',
  42);
```

識別子は適切ですが、アサーションは失敗します。*Code_15* では、要求するSQL文を示しています。

```
-- Code_15
select a.v1 from t a
where 1=0
union
select Username v1
from  All_Users where User_ID = :b1 -- a where a.PK = :b1
```

これは、簡潔なSQLインジェクションの例で、要求されたSQL文は適切です。SQL文は、プログラマーの意図と異なる構文テンプレートの例です。動的に構築されたSQL文は、リスクの高い方法で大幅に変更されています[27]。*DBMS_Assert*ファンクションを慎重に使用すれば、このような脆弱性からコードを保護することができます。

グローバル一時表を使用して一部のユースケースで簡潔かつ適切なアプローチを実現できますが、実行時まで 1 つ以上のSQL文の識別子がわからないユースケースもあります。これは、*select*、*insert*、*update*、*delete*、*merge*、または*lock table*文[28]がネイティブ動的SQLで最適にサポートされる（ほぼ）唯一のユースケースです[29]。

埋込み SQL が機能的に適切な状況において、動的 SQL の使用を正当化する理由をときどき耳にすることがあります。それは、動的 SQL を使用することで依存性をもたずに構築できるため、PL/SQL ユニットを無効化しなくても、PL/SQL ユニットが依存する表とビューの構造を変更することができるといったものです。ここでの提言は、PL/SQL コンパイラが実行できない処理をプログラマーが認識するということと、このような重要な表やビューを変更しても、PL/SQL ユニットからのそれらへのアクセスの有効性には影響を与えません。新しいファイングレイン依存性の追跡モデルが導入されたため、Oracle Database 11g ではこのリスクの高い分析の必要性がなくなります。埋込み SQL で参照されるオブジェクトにおこなわれた変更に関する安全性が立証されます。変更が安全な場合、参照している PL/SQL ユニットは有効なままです。安全ではない可能性があれば、PL/SQL ユニットは無効になります。

---

27. 攻撃者がソース・コードを参照していることが想定されます。また、テスト実装で脆弱性を把握している可能性があります。ただし、SQL インジェクションの脆弱性の多くは、ブラック・ボックス・テストによって検出できます。

28. Oracle Database の『SQL Language Reference Guide』の正式な定義と一般的な使用方法が異なるため、このホワイト・ペーパーでは、*DML* という用語を使用していません。一般的な使用方法は、*select* を除外し、*insert*、*update*、*delete*、および *merge* と対比してこれを設定します。つまり、4 つの DML のみを呼び出します（*lock table* は考慮しません）。また、『SQL Language Reference Guide』には、DML の定義に *call* と *explain plan* が含まれます。ただし、これらは PL/SQL の埋込み SQL ではサポートされません。

29. もう 1 つ、難解なユースケースがあります。たとえば、大きい表を使用したデータウェアハウス・アプリケーションの場合、最適な実行計画は、制約条件であるリテラル値の使用に依存します。参照列への実際の値の分散を記録する統計から適切に使用されます。つまり、その条件が、動的 SQL を順番に指示する SQL 文に直接エンコードされることを意味しています。
このユースケースでは、Sys.DBMS_Assert.Enquote_Literal()を使用して、SQL インジェクションの脅威から保護する必要があります。

この項については、以下のベスト・プラクティスの原則にまとめられます。

コンパイル時にテキストが固定される SQL 文を使用してください。使用できない場合は、固定したテンプレートを使用してください。プレースホルダにバインドします。DBMS_Assert を使用して、連結した SQL 識別子を保護します。

Principle_5

コンパイル時にテキストが固定されるSQL文のみを常に使用してください。*select*、*insert*、*update*、*delete*、*merge*、および*lock table*文には、埋込みSQLを使用します。ほかの文には、ネイティブ動的SQLを使用します。コンパイル時にSQL文を固定できない場合、固定した構文テンプレートを使用し、名前のプロビジョニングへの実行時のバリエーションを制限してください(これは、プレースホルダの使用とバインディングの負担の軽減を意味します)。スキーマ・オブジェクトの名前と列名などのオブジェクト内の識別子には、*Sys.DBMS_Assert.Simple_Sql_Name()*を使用します。例外的に、プレースホルダではなくリテラル値の使用が条件の場合は、*Sys.DBMS_Assert.Enquote_Literal()*を使用します。ほかの値(*Code_8*の*NLS_Date_Format*の値など)の場合は、パラメータ化されたユーザー入力に応じてプログラムで構築します。

最後に、*execute immediate* はネイティブ動的 SQL を実装する唯一の構成メンバーではありません。*Cur* が *カーソル変数である open Cur for* 文については後述します。

DBMS_Sql API

DBMS_Sql API によって、動的 SQL を実行する手順がサポートされます。以前のバージョンの Oracle Database では、*DBMS_Sql* API が動的 SQL を実行する*唯一*の手段でしたが、さまざまな出来事を経て、現在では、サポートされているすべてのバージョンでネイティブ動的 SQL がサポートされています。

ネイティブ動的SQLは、*DBMS_Sql* APIの拡張機能として導入されました(簡潔な記述と迅速な実行が可能です)。*Code_17*および*Code_18*でこの点を確認できます。一意に識別される単一行を大きい表から正しく選択するテストで、*Code_16*で設定したSQL文を実行します。

```
-- Code_16
Stmt constant varchar2(80) := '  select t.n1 from t where t.PK
= :b1';
```

*Code_17*では、*DBMS_Sql* APIを使用します。

```
-- Code_17
declare
  Cur  integer := DBMS_Sql.Open_Cursor(Security_Level=>2);
  Dummy  integer;
begin
  DBMS_Sql.Parse(Cur, Stmt, DBMS_Sql.Native);
  DBMS_Sql.Define_Column(Cur, 1, n1);

  for j in 1..No_Of_Rows loop
    DBMS_Sql.Bind_Variable(Cur, ':b1', j);
    Dummy := DBMS_Sql.Execute_And_Fetch(Cur, true);
    DBMS_Sql.Column_Value(Cur, 1, n1);
 ...

  end loop;
  DBMS_Sql.Close_Cursor(Cur);
end;
```

Open_Cursor()、Parse()、Define_Column()、およびClose_Cursor()の呼出しはループの外で、Bind_Variable()およびExecute_And_Fetch()の呼出しはループの中で実行されます。これによって、同じSQL文を繰り返してパースする必要がなくなります[30]。

*Code_18*では、ネイティブ動的SQLを使用します。

```
-- Code_18
for j in 1..No_Of_Rows loop
  execute immediate Stmt
 into n1 using j;
  ...
end loop;
```

*Code_18*は、*Code_17*よりも短く、わかりやすくなっています。このほうが、プログラマーの意図を正確に表現する可能性が増します。また、*Code_18*は、11,000行のテスト表において*Code_17*の実行速度の約 2 倍です。これは、同等の埋込みSQLアプローチとほぼ同じ速度です。

ただし、*DBMS_Sql* APIは、ネイティブ動的SQLでは対応できないSQL文を実行する要件をサポートしています[31]。その要件は以下のとおりです。

---

30. *Parse()*呼出しは、同じ意味をもつ同一の文のテキストで、共有プールの共有可能な*SQL構造*を検出します。何も検出されない場合、ハード・パースが実行されます。これは周知のとおり負荷が高くなります。ただし、すでに存在している共有可能な構造を使用するタスク（ソフト・パース）でも負荷はかかります。*Code_17*の*Open_Cursor()*、*Parse()*、*Define_Column()*、および*Close_Cursor()*を移動して、ループ内で呼出すように変更すると、11,000 行のテスト表での*Code_17*の実行結果よりも約 3 倍の時間がかかります（適切な統計的調査では、繰り返し実行されるパースは、実際にソフトである（速度が遅い）という結果を示しています）。

31. 逆に、ネイティブ動的 SQL に対応するほとんどの SQL 文の実行要件は、*DBMS_Sql* API にも対応します。Oracle Database 11g は、多くの拡張機能を *DBMS_Sql* API に導入しています。*Parse()*には、SQL 文に対して新しい CLOB 形式のオーバーロードが備わっています。*select list* には、ユーザー定義型の列を含めることができます。ユーザー定義型のバインド引数がサポートされます。*DBMS_Sql 数値カーソル*を*参照カーソル*に変換することができます（また、参照カーソルを*DBMS_Sql 数値カーソル*に変換することもできます）。

    1 つ例外があります。ユーザー定義型のコレクションに *select list* をバルク・フェッチできません。*DBMS_Sql* パッケージ仕様に定義されているコレクション・タイプのいずれかを使用する必要があります。

- プレースホルダへのバインドの要件は、実行時までわかりません。*Code_18*を参照すると、理由が簡単にわかります。ネイティブ動的SQLは、*using*句でバインディングをサポートします。コンパイル時にこの句が固定されます。対照的に、*DBMS_Sql* APIは、実行時のテストに応じて*Bind_Variable()*へ必要なだけの呼出しを実行できます。
- 戻り値の要件は、実行時までわかりません。これは、実行時まで*select list*がわからない*select*文で顕著に発生します。*Code_18*では、ネイティブ動的SQLで*into*句を使用し、*select list*のPL/SQLターゲットを指定する方法を示しています。この句もコンパイル時に固定されます。対照的に、*DBMS_Sql* APIは、実行時のテストに応じて*Define_Column()*へ必要なだけの呼出しを実行できます。

insert、*update*、*delete*、または*merge*文に*returning*句が含まれる場合、*Code_19*で示されているように*using*句[32]でこれらの値のPL/SQLターゲットを指定します。

```
-- Code_19
  ...
  Stmt constant varchar2(200) := q'[
 update t
   set t.v1 = 'New '||t.v1   where t.PK = :i1
   returning t.v1 into :o1]';
begin
  ...
execute immediate Stmt using in PK, out v1;
  ...
```

次に、動的 SQL を実行するための適切な方法とそれにかかる時間に関して疑問が生じます。これには、次のベスト・プラクティスの原則で解決できます。

動的 SQL には、ネイティブ動的 SQL を使用します。使用できない場合のみ、DBMS_Sql API を使用します。

Principle_6

動的SQLには、機能が不十分な場合を除いて常にネイティブ動的SQLを使用します。機能が不十分な場合にのみ、*DBMS_Sql* APIを使用します。*select*、*insert*、*update*、*delete*、および*merge*文の場合、コンパイル時にわからないプレースホルダまたは*select list*項目がSQL文に含まれると、ネイティブ動的SQLでは対応できなくなります[33]。ほかのSQL文の場合、操作がリモート・データベースで実行されるとネイティブ動的SQLでは対応できません。

カーソル・タクソノミー

さまざまなカーソルの特性を表すために、慎重に定義された専門用語[34]は、SQLを発行するための 3 つの方法を使用した経験のあるPL/SQLプログラマーだけが、その定義を理解できるようになります。

カーソル・タクソノミーに関する質問

主な質問は、以下のとおりです。

---

32. この反対は、戻り値のターゲットがプレースホルダとして提供される SQL の *returning* 句の構文です。

33. DBMS_Sql.To_Refcursor()ファンクションを使用して、実行されている DBMS_Sql 数値カーソルを変換できます。また、DBMS_Sql.To_Cursor_Number()ファンクションを使用して、カーソル変数を DBMS_Sql 数値カーソルに変換できます。

*質問 0：カーソル*という用語はどの論理ドメインで使用されますか。また、どのドメインにおいても同じことを指しますか。

*質問 1：*誰がカーソルを管理しますか（カーソルのオープン、SQL 文のパース、カーソルのクローズなど）。プログラマーですか。それとも PL/SQL システムですか。

*質問 2：*カーソルにはプログラマーが定義する識別子がありますか。ある場合、その識別子はどのように使用できますか。

*質問 3：*埋込み SQL、ネイティブ動的 SQL、または *DBMS_Sql* API を使用してカーソルは開きますか。

質問 0 については、2 つの異なるドメイン（公開および非公開）があります。公開ドメインは、PL/SQLソース・テキストおよびPL/SQLの構文とセマンティックの定義を考慮して指定される動作の説明です。非公開ドメインは、PL/SQLシステムの実装です。理論的に望ましくありませんが、プロフェッショナルなOracle Database開発者は理解しておく必要があります。PL/SQLランタイム・システムは、OCI[35]と同等のサーバー側の機能を正しく呼び出して、ソース・テキストで指定するSQL文の処理を管理します。Oracle Netプロトコルの受信側で使い慣れたクライアント側のOCIを実装するAPIとして、このAPIを表現できます。このため、同等の操作がサポートされます。SQLを発行する 3 つの方法のいずれかを使用するPL/SQLコードは、サーバー側のOCIに実行時に呼び出される方法と同じように実装されます。つまり、クライアント側のOCIを使用するプログラマーが、慣れている*カーソル*の概念でデータベースPL/SQLで発行されるSQL文の実行時処理の説明をサポートします。とくに、*Open_Cursors*や*Session_Cached_Cursors*などの初期化パラメータは、データベースPL/SQLで発行されるSQL文と（直接）OCI、ODBC、またはJDBCドライバ（thickドライバあるいはthinドライバ）を使用したクライアント・プログラムで発行されるSQLで同じ意味をもちます。

非公開ドメインの詳細は、共有プールに関連します。*v$SqlArea*および*v$Sql*ビューで公開されるセッションに依存しない構造は、正しくありませんがカーソル[36]と呼ばれます。

専門用語

専門用語[37]を以下に示します。

- *共有可能な SQL 構造*

  共有プールのオブジェクトです。メタデータが *v$SqlArea* および *v$Sql* ビューに公開されます。共有可能な *SQL 構造*は、これを作成したセッションのライフタイムを超えて存在し、同時にほかのセッションでも使用できます。

---

34. "term of art"（専門用語）をGoogleで検索すると、Everything2.comでは次のように定義され議論されています。*専門的な活動分野で専門家によって使用される語またはフレーズ。活動分野において、厳密で、一般的にかなり専門的な意味を持ちます。専門家らは、その分野の専門用語を使用して簡潔かつ明確に意思の疎通を図ります。一般的に、専門用語として適切な新語を作成するのは非常に困難と言われています。専門用語となる新語の意味は、分野に固有ではないことも少なくありません。これによって、分野の専門家とそれ以外の人との間に、情報伝達の壁ができるという認識が強くなります。*

35. OCI は、Oracle Call Interface の略です。

36. たとえば、"カーソル共有"や"子カーソル"などのフレーズでこれを耳にします。

SQL文のテキストが同じ場合およびほかの共有基準（SQL文の識別子が同じオブジェクトを示す場合など）が満たされる場合のみ、*共有可能なSQL構造*を再利用できます[38]。このため、この用語は、非公開領域のPL/SQLの実装（実際は、SQL文の処理をサポートする環境の実装）に属します。*共有可能なSQL構造*を再利用すると、パフォーマンスが向上します。このため、すべての環境からSQLを処理するもっとも有名な次のベスト・プラクティスの原則があります。

動的 SQL を使用する場合、SQL 文のリテラルを回避してください。代わりに、適切な値をプレースホルダにバインドしてください。

Principle_7

動的に作成した SQL 文で連結されたリテラルを使用しないでください。リテラルではなくプレースホルダを使用してください。実行時にリテラルだった値をバインドします。これによって、*共有可能な SQL 構造*を最大限に再利用できます。

- *セッション・カーソル*

  セッションのメモリ[39]のオブジェクトです。このため、セッションとともに停止します。また、メタデータが*v$Open_Cursor*ビューに公開されます。個別のセッションのSQL処理がサポートされます。

  この用語も非公開領域に属します。SQL文を発行するクライアント（たとえば、PL/SQL）は、*セッション・カーソル*を使用します。*セッション・カーソル*は、単一の*共有可能なSQL構造*と関連づけられます。ただし、*共有可能なSQL構造*には、関連づけられたいくつかの*セッション・カーソル*が存在する場合があります。また、*セッション・カーソル*は、再利用される可能性のあるオブジェクトです。クライアントが特定のSQL文の処理を完了した場合、この処理をサポートした*セッション・カーソル*は破棄されません。*ソフト・クローズ*としてマークされ、最近使用したキャッシュに保存されます[40]。最初に*ソフト・クローズしたセッション・カーソル*のキャッシュを検索して、SQL文をパースするクライアントの呼出しが実装されます。検索には、共有プールの*共有可能なSQL構造*の再利用候補に関する検索と同じ基準（SQL文のテキストおよび意味の識別）を使用します[41]。検出されない場合のみ、*新しいセッション・カーソル*の基礎として使用される一致した*共有可能なSQL構造*が共有プールで検索されます。共有プールの検索は、*ソフト・パース*と呼ばれます。*ソフト・クローズしたセッション・カーソル*の再利用は、*ソフト・パース*の実行を避ける最適化です。最悪の場合、共有プールでも検出されません。この場合、*ハード・パース*で新しい*セッション・カーソル*の基礎になる適切で新しい*共有可能なSQL構造*が作成されます。

---

37. 外国語の単語として *implicit*（暗黙）や *explicit*（明示）、*cursor*（カーソル）などを扱う際に役立ちます。また、英単語本来の意味との関連性を意識する必要がありません。とくに、*implicit cursor*（暗黙カーソル）や *explicit cursor*（明示カーソル）といった英語に直訳しない外国語の熟語のフレーズを検討する際に有効です。問題は、これらのフレーズの意味と正しい使用方法だけです。原則はやはり通例となります。アメリカ英語の *freeway*（高速道路）は通行料がかからないことを示す場合が多いですが、イギリスでは通行料のかかる *freeway* は珍しくありません。

38. 再利用は、*select*、*insert*、*update*、*delete*、*merge*、*無名 PL/SQL ブロック*といった SQL 文に制限されます。プレースホルダが使用できるのは、これらの SQL 文だけです。

39. これは一般的に PGA とも呼ばれますが、正しくは UGA です。

40. Cursor_Space_For_Time が true に設定される場合のみ、セッション・カーソルがキャッシュされます。

41. PL/SQL は、検索領域を単一の項目に絞り込んだ解析呼出しをおこなう PL/SQL 文のソース・テキストの場所に基づいて、このアプローチの最適化されたバージョンで使用します。

*Open_Cursors*初期化パラメータは、単一のセッションで同時にオープン状態で存在するセッション・カーソルの最大数を設定します。*Session_Cached_Cursors*初期化パラメータは、単一のセッションで同時にソフト・クローズ状態で存在するセッション・カーソルの最大数を設定します[42]。新しくソフト・クローズしたカーソルまたは新しく開いたセッション・カーソルのために、ソフト・クローズしたセッション・カーソルが破棄される場合があります。

- *暗黙カーソル*

  暗黙カーソルは、明らかな PL/SQL 言語の構造と概念の観点で表示されるカーソルがない場合に、埋込み SQL 構造およびネイティブ動的 SQL 構造のファミリーを実装する SQL 処理をサポートしたセッション・カーソルを表します。PL/SQL コンパイラで実行される分析を反映した PL/SQL ランタイム・システムで、*open*、*parse*、*bind*、*execute*、*fetch*、*close* などの操作を指定する明示的な言語構造を必要としないセッション・カーソルを管理します。このため、この用語は非公開領域に属します。

  SQLを実行するカーソルのないPL/SQL構造のファミリーの例を示す場合に、この用語が非公式に使用されることがあります。このような使用には注意してください。逆説的に、類似した用語の*暗黙カーソル属性*は、PL/SQLの構文およびセマンティックの公開領域に属します。同類の*明示カーソル属性*を定義するまで、この用語は定義しません。*Code_1*、*Code_2*、*Code_3*は、埋込みSQLのカーソルのないPL/SQL構造の例を示しています。*Code_8*、*Code_10*、*Code_18*、*Code_19*は、ネイティブ動的SQLのカーソルのないPL/SQL構造の例です。これには、常に*execute immediate*が使用されます。

- *明示カーソル*

  用語から*暗黙カーソル*の反意語という印象がありますが、そうではありません[43]。*明示カーソル*は、固有のPL/SQL言語機能です。このため、この用語は公開領域に属します。*Code_20*の*Pkg1* パッケージの仕様で宣言されている*Cur_Proc*識別子は、*明示カーソル*を示します。

```
-- Code_20
package Pkg1 is
  type Result_t is record(PK t.PK%type, v1 t.v1%type);
  cursor Cur_Proc(PK in t.PK%type) return Result_t;
  ...
end Pkg1;
```

42. *Open_Cursors* は、機能の制限を設定します。アプリケーションが同時にアクティブにするセッション・カーソルの数を確認する必要があります。対照的に、*Session_Cached_Cursors* は、パフォーマンスのトレードオフに対して標準的な領域を管理します。

43. これは以前に警告されています。

Cur_Proc は、*Code_21* の *Pkg1* パッケージ本体で定義されます。

```
-- Code_21
package body Pkg1 is
 cursor Cur_Proc(PK in t.PK%type) return Result_t is
  select   a.PK, a.v1
  from     t a
  where    a.PK > Cur_Proc.PK
  order by a.PK;
  ...
end Pkg1;
```

*Code_22*は、*Cur_Proc明示カーソル*の使用方法を示しています。この構造は、*明示カーソルFORループ*と呼ばれます。

```
-- Code_22
for r in Pkg1.Cur_Proc(PK=>Some_Value)loop
  Process_Record(r);
end loop;
```

*明示カーソル*は、動的 SQL を使用して定義することはできません。埋込み SQL が唯一の手段です。

プログラマーが*明示カーソル*という用語を採用したのですが、可変要素ではありません。つまり、サブプログラムの呼出しで実際の引数として使用することはできないのです。また、ファンクションで返すこともできません。この点で、プロシージャに非常に似ています[44]。また、前方宣言、パッケージと本体への宣言と定義の分割、仮パラメータの使用などの類似性もあります。ただし、この可能性を活用するコードを記述してもメリットはありません（代わりに、*参照カーソル*を返すファンクションを常に使用でき、メリットを得られます）。

- *参照カーソル*

  宣言されたPL/SQLのみのデータ型[45]です。たとえば、*Code_23*または*Code_24*の*Cur_t*などです。参照カーソルを使用して、変数、サブプログラムの仮パラメータ、およびファンクションの戻り値を宣言できます。コレクションの要素または*レコード*のフィールドのデータ型の宣言には使用できません。2 種類の参照カーソル（*弱い参照カーソル*および*強い参照カーソル*）があります。

- *カーソル変数*

  データ型が参照カーソルに基づいた変数です。参照カーソル、*弱い*参照カーソル、*強い*参照カーソル、および*カーソル変数*は、PL/SQL言語の機能です。このため、公開領域に属します。*Cur*が*カーソル変数*の場合、*select*文と*カーソル変数*を関連づける*open Cur for* PL/SQL文で使用できます。埋込みSQLまたはネイティブ動的SQLを使用して関連づけることができます。*fetch*文のソースとしても*Cur*を使用できます[46]。

---

44. *カーソル・サブプログラム*という名前が適切である可能性もあります。

45. 厳密にいうと、*record(...)*や *table of boolean index by pls_integer* のように、*カーソル*というキーワードはデータ型構造を示します。*REF* キーワードは別で、データ型が参照カーソルの場合に参照セマンティックに従うことを示します。これは異例です。通常、PL/SQL の場合に値セマンティックに従います。参照セマンティックに従う場合も少しあります。たとえば、独自のルールである永続 LOB ロケータなどです。

パッケージの仕様または本体のグローバル・レベルで*カーソル変数*が宣言されない場合があります。

- *強い参照カーソル*

  宣言されたデータ型です。*Code_23*の*Strong_Cur_t*などが該当します。

```
-- Code_23
  type Result_t is record(PK t.PK%type, v1 t.v1%type);
  type Strong_Cur_t is ref cursor return Result_t;
```

  *強い*参照カーソルは、*select* 文で定義する *select list* 項目の数値およびデータ型を厳密に指定します。埋込み SQL でのみ、データ型が*強い*参照カーソルの*カーソル変数*を開くことができます。

- *弱い参照カーソル*

  宣言されたデータ型です。*Code_24*の*Weak_Cur_t*などが該当します。

```
-- Code_24
  type Weak_Cur_t is ref cursor;
```

  *弱い*参照カーソルは、*select*文で定義する*select list*項目の数値およびデータ型には依存しません。データ型が*弱い*参照カーソルの*カーソル変数*は、埋込みSQLまたはネイティブ動的SQLを使用することで開くことができます[47]。

  *Code_25*の*Pkg2* パッケージの仕様では、*New_Cursor()*ファンクションを宣言します。このファンクションは、*Code_20*の*Pkg1* で宣言された*明示カーソル*と同じようにパラメータ化され、値が*カーソル変数*に設定されるように設計されています。

```
-- Code_25
 package Pkg2 is
  type Result_t is record(PK t.PK%type, v1 t.v1%type);

  type Cur_t is ref cursor
    $if $$Embedded $then return Result_t;
    $else ;
    $end

  function New_Cursor(
    PK in t.PK%type)
    return Cur_t;
  ...
end Pkg2;
```

---

46. 制限を克服するために*明示カーソル*よりもあとに*参照カーソル*と*カーソル変数*が PL/SQL に導入されました。*参照カーソル*と*カーソル変数*が最初に導入されていた（最初から*バッチ・バルク・フェッチ*構造がサポートされていた）場合、*明示カーソル*を導入したプロジェクトが正しく評価されていた可能性は非常に低いと考えられます。

47. データ型が*強い参照カーソル*の*カーソル変数*を使用すると、レコードまたは *select list* の構造に一致しない一連のスカラーにフェッチする場合に、ランタイム・エラーではなくコンパイル時にエラーを取得できる利点があります。それ以外の場合、利点はありません。また、ソース・テキストの保守に少し負担がかかります。Oracle9*i* Database から、*Standard* パッケージで*弱い参照カーソル*型の *Sys_RefCursor* が宣言されます。これを使用すると入力が省略されて、読者に直接意味が伝えられます。

*Code_26*の*Pkg2* パッケージの本体では、ファンクションを定義します[48]。

```
-- Code_26
package body Pkg2 is
  function New_Cursor(
   PK in t.PK%type)
   return Cur_t

  is
    Cur_Var Cur_t;
  begin

 open Cur_Var for
   $if $$Embedded $then
     select  a.PK, a.v1
     from  t a
     where  a.PK > New_Cursor.PK
     order by  a.PK;

$else
 '
      select    a.PK, a.v1
      from      t a
      where     a.PK > :b1
      order by a.PK'
     using in New_Cursor.PK;
$end
   return Cur_Var;
  end New_Cursor;
  ...
end Pkg2;
```

*Code_25*および*Code_26*では、条件付きコンパイル[49]を使用して、*カーソル変数*を開く 2 つの方法における小さな差異と大きな類似点を強調しています。

---

48. 文の種類が *select*、*insert*、*update*、*delete*、*merge*、または*無名 PL/SQL ブロック*の場合、*constant* を使用して宣言されたテキストの SQL 文でネイティブ動的 SQL を使用する理由は、実際のコードではほとんどありません。前述のとおり、使用する理由があるとすれば、参照対象の表または PL/SQL ユニットがコンパイル時には存在していないが、コード実行前には作成されるという場合です。コードがインストール・スクリプトではない限り、このアプローチを提案するユースケースは、慎重に検討する必要があります。

49. Oracle Database 10g Release 2 で条件付きコンパイルが導入されました。*Embedded* 識別子は、CC フラグと呼ばれます。*$$Embedded*（問合せディレクティブと呼ばれます）を記述して、ソース・テキストに値を取得します。次のようなコマンド（create or replace の前）で値が設定されます。

```
alter session set Plsql_CCflags = 'Embedded:true'
```

既存のパッケージ本体 Pkg には、次のようなコマンドを使用します。

```
alter package Pkg compile
  Plsql_CCflags = 'Embedded:false'
  reuse settings
```

*Code_27*は、*New_Cursor()*で初期化された*Cur_Var*カーソル変数の使用を示しています。ループ構造は、*無限カーソル・フェッチ・ループ*[50]と呼ばれます。

```
-- Code_27
declare
  Cur_Var Pkg2.Cur_t :=
    Pkg2.New_Cursor(PK=>Some_Value);
  r Pkg2.Result_t;begin
  loop
    fetch Cur_Var into r;
    exit when Cur_Var%NotFound;
    Process_Record(r);
  end loop;
  close Cur_Var;
end;
```

無限カーソル・フェッチ・ループ（*Code_27*）は機能的に*明示カーソルFOR*ループ（*Code_22*）と同等ですが、より冗長になります。*無限カーソル・フェッチ・ループ*は*カーソル変数*または*明示カーソル*で使用できますが、*明示カーソルFOR*ループは*明示カーソル*でのみ使用できます。また、Oracle Database 10g以降、*明示カーソルFOR*ループは、*無限カーソル・フェッチ・ループ*よりも高速になりました。これは、最適化されたコンパイラが配列フェッチ[51]を使用して、前者を内部的かつ安全に実装できるようになったためです。このような最適化は、後者にとって安全ではありません。コード内の同じ*明示カーソル*または*カーソル変数*からインターリーブ・フェッチが発生しないことがオプティマイザで保証されないためです[52]。ただし、これらのアプローチよりも*バッチ・バルク・フェッチ*（*Code_29*を参照）または*全体バルク・フェッチ*（*Code_32*および*Code_33*を参照）の方が望ましいため、これらのアプローチには現実的な利点がありません。

- *識別カーソル*

  *無限カーソル・フェッチ・ループ*のソース・テキストは、*明示カーソル*および*カーソル変数*の両方で同一であり、（*Code_29*および*Code_34*に示されているように）このプロパティをもつほかの構造が存在するため、*明示カーソル*および*カーソル変数*のスーパークラスに対して、この専門用語を使用すると便利です。実際、このような用語は存在しませんが、本書では名前の意図するとおりに*識別カーソル*という用語を採用しています。これによって、*暗黙カーソル*で内部的にサポートされるカーソルのないPL/SQL構造と、プログラマーが*明示カーソル*または*カーソル変数*の識別子を作成する場合に*識別カーソル*を使用する構造を、それぞれ正しく区別できます。また、サポートしている*セッション・カーソル*の管理方法がPL/SQLシステムに少なくともある程度は通知されます。

---

50. 明示カーソル FOR ループは明示カーソルにのみ有効ですが、無限カーソル・フェッチ・ループは明示カーソルおよびカーソル変数の両方に有効です。ただし、現実的な利点はありません。

51. この"配列フェッチ"は、サーバー側の OCI のプログラミング技術を示します。

52. コード内の同じ*明示カーソル*または*カーソル変数*からインターリーブ・フェッチが発生しないことがオプティマイザで保証されない制限は、現在使用されている（Oracle Database 11g の）オプティマイザ技術を反映しています。

- *DBMS_Sql 数値カーソル*

  *DBMS_Sql.Open_Cursor()*ファンクションの戻り値です。*number*データ型（または*integer*などの*number*のサブタイプ）の通常の変数（*Cur*など）に割り当てることができます。SQL文の処理が完了した場合、*in out*仮パラメータの実効値*c*として、*Cur*を使用して*DBMS_Sql.Close_Cursor()*を呼び出します。呼出し時に*Cur*が既存の開いている*DBMS_Sql数値カーソル*を示す場合、*Cur*が*null*に設定されます。*DBMS_Sql.Is_Open()*を呼び出して、*Cur*の現在の値が開いている*DBMS_Sql数値カーソル*を示しているかどうかを確認できます。*DBMS_Sql* APIの各サブプログラムには、値を返す*Open_Cursor()*およびパラメータ・モードが*in out*の*Close_Cursor()*を除く既存の開いている*DBMS_Sql数値カーソル*に実効値を設定する*in*仮パラメータがあります。既存の開いている*DBMS_Sql数値カーソル*を示さない*Cur*の値でこれらを呼び出すと、*ORA-29471: DBMS_SQL access denied*エラーが発生します。セッションでこのエラーが発生すると、*DBMS_Sql* APIの後続のすべてのサブプログラムの呼出しで同じエラーが発生します[53]。

  *Open_Cursor()*ファンクションには、2 つのオーバーロードがあります。1 つには仮パラメータがありません。もう 1 つ（Oracle Database 11gの新機能）には、仮パラメータ（値 *1* または *2* を使用できる*Security_Level*）があります。*Security_Level = 2* の場合、*Parse()*への最近の呼出しと同様に*DBMS_Sql* APIへのすべての呼出しで、*Current_User*および有効なロールを同じにする必要があります[54]。*Security_Level = 1* の場合、*Parse()*への最近の呼出しと同様に*Bind_Variable()*、*Execute()*、*Execute_And_Fetch()*への呼出しで、*Current_User*および有効なロールを同じにする必要があります。ただし、*Define_Column()*、*Define_Array()*、*Fetch_Rows()*などの呼出しは制限されていません。本書では、プロデューサPL/SQLユニット（41 ページの"*プロデューサ/コンシューマのモジュール化のアプローチ*"を参照）の*DBMS_Sql* APIへの呼出しのカプセル化を推奨しています。次のベスト・プラクティスの原則は、この推奨に従っています。

常に DBMS_Sql.Parse (Security_Level=>2)で DBMS_Sql 数値カーソルを開きます。

Principle_8

*Security_Level*仮パラメータを使用する*DBMS_Sql.Parse()*のオーバーロードを常に使用してください[55]。また、*DBMS_Sql数値カーソル*のすべての操作が同じ*Current_User*および有効なロールで実行される実効値*2*で常に呼び出してください。

---

53. これは、Oracle Database 11g の新しい動作です。開いている *DBMS_Sql 数値カーソル*へのスキャン攻撃から保護して、制限されたデータを表示するためにリバインドや再実行などをおこなう目的があります。

54. 最近の *Parse()*呼出し時と同じ有効なロール（またはそのスーパーセット）を使用する必要があります。

55. エンハンスメント・リクエスト 6620451 は、仮パラメータのない *DBMS_Sql.Parse()*のオーバーロードが使用される場合にコンパイラの警告を要求します。

- *明示カーソル属性*

  *Cur*が*明示カーソル*または*カーソル変数*の場合、*Cur%IsOpen*（戻り値は*boolean*）、*Cur%NotFound*（戻り値は*boolean*）[57]、および*Cur%RowCount*（戻り値は*integer*）といったリフレクタ[56]を使用できます。*Cur%IsOpen*がtrueではない場合、ほかの*明示カーソル属性*の参照に失敗します[58]。*select*文にのみ*明示カーソル*および*カーソル変数*が開かれるので、*Cur%RowCount*は、カーソルのライフタイムで今までにフェッチした行の総数を提供します。*Cur%NotFound*は、すべての行がフェッチされるまで*true*のままです。本書で推奨しているSQL文を処理するためのアプローチを採用した場合、すべての一般的なユースケースで*Cur%RowCount*および*Cur%NotFound*を使用することで利点を得ることはできません。*Code_28*に示されているように、例外ハンドラに*Cur%IsOpen*が役立つ可能性があります[59]。

  ```
  -- Code_28
  if Cur_Var%IsOpen then
    close Cur_Var;

  end if;
  ```

- *暗黙カーソル属性*

  これらのリフレクタは、*暗黙カーソル*が使用される現在のSQL文（現在実行されていない場合は最近完了したSQL文）の実行情報を通知します。*暗黙カーソル*がとくに*execute immediate*文をサポートするので、すべての種類の文が使用される可能性があります。スカラー・リフレクタは、*Sql%IsOpen*、*Sql%NotFound*[60]、*Sql%RowCount*です。

  これらは、同じデータ型を返し、対応する名前の*明示カーソル属性*と同じ意味があります（*SQL* は PL/SQL の予約語なので、*暗黙カーソル属性*と*明示カーソル属性*を混同する危険はありません）。ただし、*暗黙カーソル*が PL/SQL システムで管理されるので、*Sql%IsOpen* を確認する利点はありません（単なる調査対象です）。驚くことに（しかし、*暗黙カーソル*が PL/SQL システムで管理されることと同じ理由で）、*Sql%IsOpen* が false の場合に *Sql%NotFound* および *Sql%RowCount* を参照できます。*Sql%NotFound* は、*Sql%RowCount = 0* と常に同じなので無視できます。*Sql%RowCount* は、最新の *select*、*insert*、*update*、*delete*、または *merge* 文の影響を受ける行の数を通知します。ただし、より直接的なほかの方法では同じ結果になるので、*select* 文は無視できます。ほかの種類の文のあとに *Sql%RowCount* が常にゼロになるため、無視できます。

---

56. ファンクションのように機能します。式に使用できますが、割当て対象としては使用できません。
57. Cur%Found もありますがここでは取り上げません。値が常に not Cur%NotFound と同じになり、ほとんど使用されないためです（Cur%NotFound が null の場合、Cur%Found も null です）。
58. エラーは ORA-01001: *invalid cursor* です。
59. *カーソル変数*が範囲外の場合、PL/SQL ランタイム・システムで閉じます。ただし、例外ハンドラで明示的にこの安全策に対応しても問題ありません。対策が必要かどうかを推測するよりもこれを実行する方が常に簡単です。
60. 値が常に not Sql%NotFound と同じになる Sql%Found もあります。

非スカラー・リフレクタは、*Sql%Bulk_RowCount*および*Sql%Bulk_Exceptions*で、それぞれ*forall*文に関連した場合のみ有効です。*forall*文は、*insert*、*update*、*delete*、および*merge*のSQL文のみをサポートします。*Sql%Bulk_RowCount*は、numericデータ型で索引づけされたコレクションです。要素のデータ型は*pls_integer*です。*forall*文の繰返しで影響を受ける行の数を通知します。索引は繰返しの数で、*1* から文の起動に使用されたコレクションの*Count()*までの範囲で順番につけられます。*Sql%Bulk_Exceptions*は、*forall*文で*save exceptions*が使用される場合のみ有効です。*Bulk_Errors*例外の*ORA-24381* の例外ハンドラでのみアクセスできます。*pls_integer*で索引づけされたコレクションです。要素のデータ型は*レコード*に基づきます。最初のフィールドは*Error_Index*で、2 つ目のフィールドは*Error_Number*です。両方とも*pls_integer*です。索引は *1* から順番につけられます。ただし、*forall*文の繰返しが多いと例外が発生します。*Error_Index* は、繰返しの数です。範囲は *1* から文の起動に使用されたコレクションの*Count()*までです。*Error_Number*は、*Pragma Exception_Init*文で使用されるような例外に対応するOracleエラー番号と*-1* を乗算した値と同じです[61]。

*暗黙カーソル*を使用する方法で SQL 文を実行するたびに、*暗黙カーソル属性*で前述のような SQL 文に通知した値を上書きします。このため、次のベスト・プラクティスの原則があります。

使用する必要がある唯一の明示カーソル属性は、Cur%IsOpen です。必要な暗黙カーソル属性は、Sql%RowCount、Sql%Bulk_RowCount、および Sql%Bulk_Exceptions のみです。

Principle_9

本書が推奨するアプローチに従う場合、唯一の便利な*明示カーソル属性*は*Cur%IsOpen* です。ほかの*明示カーソル属性*を使用する必要はありません。唯一影響のあるスカラー型の*暗黙カーソル属性*は、*Sql%RowCount* です。*暗黙カーソル*を使用して対象の SQL 文を実行する文に準拠する PL/SQL 文で、これを常に確認してください。同じ論理が *Sql%Bulk_RowCount* コレクションに当てはまります。*Bulk_Errors* 例外の例外ハンドラでのみ、*Sql%Bulk_Exceptions* を使用する必要があります。実行可能なセクションの唯一の文として、*forall* 文を含む*ブロック文*にこれを配置してください。

まとめ

慎重に定義した専門用語により、しっかりとした基盤が確立されたので、この項の前半に記載した質問に回答できます。

- *"質問 0：カーソルという用語はどの領域で使用されますか。また、各領域で同じ意味で使用されていますか"*では、公開領域（PL/SQLの構文およびセマンティック）と非公開領域（PL/SQLのランタイム実装）の 2 つの異なる領域でこの用語が使用されます。大局的には、SQL文の処理をサポートするほかの環境の構文およびセマンティックの説明にもこの用語を使用します。ただし、本書では取り上げません[62]。

61. *Error_Number* が正数であることに混乱するユーザーもいます。これは、このような動作の変更で破損する可能性のある既存のプログラムのために修正されないバグとして考慮してください。

- "*質問 1：誰がカーソルを管理しますか（カーソルのオープン、SQL文のパース、カーソルのクローズなど）。プログラマーですか。それともPL/SQLシステムですか*"では、実行を命令する言語構造を使用しないでSQL文をサポートするセッション・カーソルの管理方法をPL/SQLシステムで決定する場合、一連のカーソルのないPL/SQL構造があります。このような場合、セッション・カーソルは*暗黙カーソル*と呼ばれます（既存のアプリケーション・アーキテクチャに準拠したコードを記述しているジュニア・プログラマーは、*カーソル*という用語を何年も耳にしていない可能性があります）。ほかの場合（埋込みSQLおよびネイティブ動的SQLでは常に*select*文）、セッション・カーソルの管理を決定する 2 つの明示的な言語構造のいずれかを使用します。構造は*明示カーソル*と*カーソル変数*です。*DBMS_Sql* APIを使用する場合、プログラマーは、OCIで公開されるカーソル管理プリミティブに、通常 1 対 1 でマップされるサブプログラムを使用して、セッション・カーソルを詳細に管理します。
- "*質問 2：カーソルにはプログラマーが定義する識別子がありますか。ある場合、どうすれば使用できますか*"では、カーソルのないPL/SQL構造では反復してプログラマーが定義する識別子を使用できません。ただし、*明示カーソル*、*カーソル変数*、および*DBMS_Sql数値カーソル*では、識別子を作成できます。*明示カーソル*の識別子は、サブプログラムの識別子に似ています。割当てには使用できません（サブプログラムの仮パラメータとして使用できないことを示します）。対照的に、*カーソル変数*および*DBMS_Sql数値カーソル*の識別子をサブプログラムの仮パラメータとして割当てに使用できます。
- "*質問 3：埋込みSQL、ネイティブ動的SQL、またはDBMS_Sql APIを使用してカーソルが開きますか*"では、特定の埋込みSQL文およびネイティブ動的SQL文に応じて*暗黙カーソル*が管理されますが、*DBMS_Sql* APIでは管理されません。埋込みSQLの*select*文にのみ、*明示カーソル*が関連づけられます。埋込みSQLの*select*文またはネイティブ動的SQLの*select*文で、*カーソル変数*を開くことができます。すべての種類のSQL文で*DBMS_Sql数値カーソル*を使用できます。

カーソル[63]自体が便利な専門用語というわけではありません。それどころか、修飾されていないと意味がないのです。つまり、直接関連する文において適切な専門用語を使用して、修飾されていないカーソルを削除することが、あいまいな文を回避するための有効な手段となります。これは、次のベスト・プラクティスの原則につながります。

---

62. SQL 文コマンドの*カーソル変数*のプレースホルダとして*:Cur* を記述できるように、SQL*Plus スクリプト言語で `VARIABLE Cur REFCURSOR` コマンドをサポートします。

63. *カーソル*という用語を選択したのは、結果セットに沿って実行され、現在の位置を保持するからです。この用語が使われるようになった時点（本書の執筆時点でオラクルには 30 年の歴史があります）で、ビジュアル・ディスプレイ・ユニット（80 文字のスクロール可能なテレタイプ・ロールを緑色の画面に表示）は、現在の文字位置を表すために類似した概念が必要になり、同じ*カーソル*という用語が採用されました。この用語は *select* 文の処理の説明に属しますが、厳密にいうとすべての種類の SQL 文の処理を含むすべての構造を示します。

専門用語のセッション・カーソル、暗黙カーソル、明示カーソル、カーソル変数、および DBMS_Sql 数値カーソルを学習します。省略せず、慎重に使用してください。

Principle_10

PL/SQL プログラムを説明する場合、PL/SQL プログラム自体の説明、そのコメント、および外部ドキュメントの記述が、修飾されていない"カーソル"の使用を回避する目的で含まれています。*セッション・カーソル*、*暗黙カーソル*、*明示カーソル*、*カーソル変数*、または *DBMS_Sql 数値カーソルといった適切な専門用語*を使用してください。この原則により思考力が高まってプログラムの品質が向上します。

SELECT 文のアプローチ

SELECT 文の概念が難解であると思われているようですが、この項は、読者が主題の内容に精通していることを前提としています。ここでは、最適なアプローチの詳細について説明し、さらに、これまでの経緯から簡単だと思われていた、最適ではないアプローチに簡単に触れることにします。

複数行の選択 - アンバウンド結果セット

データベース PL/SQL プログラムの一般的な使用の 1 つに、レポートの一括準備があります。これは、非常に大きい表の各行（および関連する詳細）の処理をおこなう際に使用します。このようなシナリオには、各行を個別に処理できます。

Oracle Databaseを使用してプログラミングするすべてのPL/SQLアプリケーション開発者は、SQLとPL/SQLは異なる仮想マシンで実行されるということを、じきに理解するようになります。つまり、PL/SQLサブプログラムがSQL文を実行する際に、PL/SQL仮想マシンからSQL仮想マシンに切り替えられ、再度PL/SQL仮想マシンに切り替える、いわゆるコンテキスト・スイッチが少なくとも 1 回、あるいは何回もおこなわれます。PL/SQLでは、メモリ内での使用のために最適化される形式が使用され、SQLでは、ディスク上での使用のために最適化される形式が使用されるため、コンテキスト・スイッチではデータの表示変換も同時におこなわれます。コンテキスト・スイッチには必然的に負担が生じます。SQL文の実行中に発生するコンテキスト・スイッチの数を最小限に抑えることで、パフォーマンスは改善できます[64]。

Oracle8i Database（今ではほとんど使用されていません[65]）で、バルクSQLのPL/SQL構造が導入されました。以降、Oracle Databaseのすべてのサポート・バージョンでこの構造がサポートされています。*select*文の場合、limit allで、1 回のコンテキスト・スイッチによって多数の結果行をフェッチできます。本番使用のコードで、複数の行を返す*select*文を実行する際に、PL/SQLの非バルク構造が適しているというわけではありません[66]。

64. コンテキスト・スイッチの説明が不十分と感じるかもしれませんが、重要なことは、別の手段で同じ効果を得ることは困難であるということを覚えておくことです。たとえば、C や Java で記述されたクライアント側のプログラムで SQL を実行する方法を考えてください。ここでもコンテキスト・スイッチが必要です。この場合のコンテキスト・スイッチは、より劇的です。単一の実行可能なプログラムのアドレス空間内でおこなう PL/SQL と SQL のコンテキスト・スイッチのラウンドトリップは、ここでは異なるマシンで実行される個別のプログラム間でおこなわれます。ただし、ネットワーク通信を実装するほかのプログラムが介在します。そのため、データ表現が 2 回変換される場合があります（クライアント・プログラムの表現および通信の表現の変換と、通信の表現および SQL のディスク上の表現の変換）。

65. Oracle Technology Network の Web サイトから『PL/SQL ユーザーズ・ガイドおよびリファレンス』のリリース 8.1.6 をオンラインで入手できます。公開日は 1999 年です。

66. PL/SQLに精通しているユーザーは、一時的な*非定型*レポートの生成に頻繁に使用します。このように一度書き込んで数回使用する場合、*Code_29*のバルク・アプローチよりもプログラミングの負担が若干少ないため、*Code_2*の*暗黙カーソルFOR*ループが適切なアプローチになります。

フェッチ・ループのプログラミング

多くの行をフェッチするターゲットには、コレクションを使用する必要があります。これをモデル化するもっとも一般的な方法は、フィールドが*select list*項目に対応した*レコード*のコレクションを使用することです。*Code_29*は、例を示しています。

```
-- Code_29
-- Cur is already open here.
-- The fetch syntax is the same for
-- an explicit cursor and a cursor variable.
loop
  fetch Cur bulk collect into Results limit Batchsize;
  -- The for loop doesn't run when Results.Count() = 0
  for j in 1..Results.Count() loop
   Process_One_Record(Results(j));
  end loop;
  exit when Results.Count() < Batchsize;
end loop;
close Cur;
```

Cur カーソルを開くために使用される問合せの結果セットが大きいことが要件ドキュメントに記載されている場合、プログラマーは、1 回のコンテキスト・スイッチですべての結果を安全にフェッチできません。正確に言えば、すべての結果はバッチでフェッチする必要があります。コンパイル時にバッチの最大サイズを安全に設定できます。*limit* 句を使用してこれを要求します。

プログラマーによっては、終了基準の指定方法がわからずに、*Cur%NotFound*のテストを試みるプログラマーもいます。この処理は、*fetch*文が行の取得中に、フェッチされた行数がバッチサイズよりも少ないにもかかわらず*false*になるため不適切です。最後の部分的なバッチのサイズは、たいていゼロ以外になる可能性が高く、このため、通常*Cur%NotFound*が*false*の場合に終了すると、異常な動作が起こります[67]。テストに必要な数は、*Results.Count()*でフェッチした行数です。*fetch*文の直後にある*exit when Results.Count() < 1;*の配置は適切ですが、これは通常、必要なフェッチよりも 1 つ多いフェッチが試行されることを意味します。*Code_29*は、このわずかな負担を回避する適切なアプローチを示しています。当然ですが、行の総数がバッチサイズの整数倍になる場合、正しい動作を保証するための負担が生じます。ただし、通常は、*1..Results.Count()*ループで結果が処理され、正確性が保証されます。

結果として、*バッチ・バルク・フェッチ*の実装において*明示カーソル属性*は役立ちません。

*Batchsize*の値は、数百程度が適切です[68]。これを実行時に変更する理由はありません。このため、*constant*として*Batchsize*が適切に宣言されます。また、*Batchsize*と同等の最大サイズの*varray*として、コレクションが適切に宣言されます。*Code_30*は、この宣言を示しています[69]。

---

67. *Cur%RowCount* も役立ちません。連続した *fetch* 文でインクリメントされるので、すべてのバッチでフェッチした行の総数と同じままになります。

68. バッチサイズがわずかな値（3 や 5）の場合に少しずつ増やすことで、すべての結果の処理に必要な時間が大幅に削減されたというテスト結果があります。ただし、バッチサイズが数百程度の場合、数百から数千に増やしても効果は小さいという結果がでています。この点に関しては、ご自身でテストすることをお奨めします。

```
-- Code_30
Batchsize constant pls_integer := 1000;

type Result_t is record(PK t.PK%type, v1 t.v1%type);
type Results_t is varray(1000) of Result_t;
Results Results_t;
```

カーソルのオープン

カーソルのないPL/SQL構造で*バッチ・バルク・フェッチ*を表す構文がないという点は重要です。このアプローチでは、*識別カーソル*を使用する必要があります。*Code_31*は、*Code_29*を実行可能にするために*Cur識別カーソル*を設置する 3 つの方法を示しています。

```
-- Code_31
 ...
 $if $$Approach = 1 $then
  cursor Cur is
   select    a.PK, a.v1
   from      t a
   where     a.PK > b.Some_Value
   order by  a.PK;
$elsif $$Approach = 2 or $$Approach = 3 $then
  Cur Sys_Refcursor;
  $if $$Approach = 3 $then
   Stmt constant varchar2(200) := '
     select  a.PK, a.v1
     from  t a
     where  a.PK > :b1
     order by  a.PK';
    $end
   $end
begin
  $if $$Approach = 1 $then
 open Cur;
   $elsif $$Approach = 2 $then
 open Cur for
   select    a.PK, a.v1
   from      t a
   where     a.PK > b.Some_Value
   order by  a.PK;
$elsif $$Approach = 3 $then
 open Cur for Stmt using Some_Value;
$end
...
```

CC フラグの *Approach* が *1* の場合、*Cur* は*明示カーソル*として設置されます。*Approach* が *2* の場合、*カーソル変数*として設置され、埋込み SQL で開かれます。*Approach* が *3* の場合、*カーソル変数*として設置され、ネイティブ動的 SQL で開かれます。
ただし、*Code_29*のコメントに示されているように、これらの 2 つの種類のカーソルのフェッチ・コードは同じです。このため、"*明示カーソルとカーソル変数*の使用が望ましい状況をどうやって判断すればいいか"という疑問が生じます。ユースケースで動的SQLが必要な場合は、*カーソル変数*だけが使用可能です。したがって、この疑問に関連するのは、埋込みSQLでこの要件を満たすことができる場合だけです。

---

69. Oracle Database 11gのPL/SQLでは、*Code_30*の特別な意味をもった数値である 1000 のテキストの繰返しを回避するための便利な方法がありません（*$$Batch_Size*などの条件付きコンパイルの問合せディレクティブを使用できますが、*Code_30*を含むPL/SQLユニットのカプセル化が破損するという大きな欠点があります）。

これについては簡単に回答できます。このような場合は、宣言と対象の *select* 文への関連付けがよりコンパクトなコードで達成されるので、*明示カーソル*を使用してください。
ほかにも検討事項はありますが、それらは具体的なユースケースで検討するのが最善の選択です。この点については、41 ページの*"プロデューサ/コンシューマのモジュール化のアプローチ"*で説明します。

この項の議論は、以下のベスト・プラクティスの原則にまとめられます。

問合せで取得される行数がわからない場合、無限ループ内で limit 句とともに fetch... bulk collect into を使用してください。

Principle_11

多くの行が選択されて結果セットが大きい場合は、*無限ループ*内で *limit* 句とともに *fetch... bulk collect into* を使用してバッチで処理します。*limit* 句の値として *constant* を使用し、バッチサイズを定義します。適切な値は 1000 です。同じサイズで宣言された *varray* にフェッチします。ループを終了するために*%NotFound* カーソル属性をテストしないでください。代わりに、ループの最後の文として *exit when Results.Count() < Batchsize;*を使用します。これによって、最後のフェッチがゼロ行になる処理が正しく実行されます。埋込み SQL で十分な場合は、*明示カーソル*を使用します。ネイティブ動的 SQL が必要な場合は、*カーソル変数*を使用します。

複数行の選択 - バウンド結果セット

データベースPL/SQLプログラムの一般的な使用には、マスター行および関連するすべての詳細行のフェッチもあります。正規化されたマスターの例は、*Orders*表です。正規化された詳細の例は、*Order_Line_Items*表です。これらの表がインターネット・ショッピング・サイトのバックエンドをサポートしている場合、注文の明細項目の最大数（たとえば、1000）を設定して、これを超えないようにビジネス・ルールをまとめると安全です[70]。

推奨されているバッチサイズで*バッチ・バルク・フェッチ*・アプローチを使用する場合、複数のバッチをフェッチしません。そのため、このアプローチは必要ありません。代わりに、簡単な*全体バルク・フェッチ*を使用できます。

*Code_32*は、埋込みSQLを使用した例を示しています[71]。

```
-- Code_32
declare
  Target_Varray_Too_Small exception;
  pragma Exception_Init(Target_Varray_Too_Small, -22165);
begin
  select            a.PK, a.v1
  bulk collect into  x.Results
  from              t a
  where             a.PK > x.Some_Value
  order by           a.PK;
exception when Target_Varray_Too_Small then
  Raise_Application_Error(-20000,
 'Fatal: Business Rule 12345 violated.');
end;
```

70. 書籍または DVD をオンラインで注文する利用者のショッピング・カートが 1,000 個のアイテムでいっぱいになることはまずありません。もしこのようなことがあった場合でも、"ショッピング・カートのアイテムが 1,000 を超えました。確認して別のカートを利用してください。本日注文すると、新しいカートのアイテムには 10%の割引が適用されます。"というように顧客側にメリットのあるメッセージであれば、顧客の満足度が低下したりビジネスの損失につながったりする可能性は低いでしょう。

71. 本書では、現実的な例を作成しようとはしていません。本書は、そうした例を必要としない十分な経験をもっている読者を対象にしています。ここでは、PL/SQL により SQL を実行するさまざまな技術に焦点を当てています。余計な細かい情報で説明が複雑になることを避け、一般的な例でわかりやすく説明しています。

*1..<varray size>*の範囲ではない索引で *varray* の要素にアクセスした場合、*ORA-22165* エラーが発生します。事前のバグ診断技術として、このエラーのハンドラを提供する厳密に囲んだ*ブロック文*の*全体バルク・フェッチ*文を記述すると便利です。

*Code_33*は、ネイティブ動的SQLを使用した例を示しています。

```
-- Code_33
declare
  Stmt constant varchar2(200) := '
    select    a.PK, a.v1
    from      t a
    where     a.PK > :b1
    order by  a.PK';
  Target_Varray_Too_Small ...
begin
  execute immediate Stmt
    bulk collect into Results
    using Some_Value;
exception when Target_Varray_Too_Small then
  Raise_Application_Error(-20000,
  'Fatal: Business Rule 12345 violated.');
end;
```

各例でカーソルのないPL/SQL構造が使用されていることを確認します。比較のため、*Code_34*に*Cur識別カーソル*を使用する*全体バルク・フェッチ*方式を示します。

```
-- Code_34
-- Cur is already open here.
-- The fetch syntax is the same for
-- an explicit cursor and a cursor variable.
fetch Cur bulk collect into Results;
close Cur;
```

*Code_31*と同じコードを使用して、*Cur*を設置します。

*Code_34*のアプローチに必要なコード量の合計は、機能上同等の*Code_32*や*Code_33*よりも大幅に多くなります。理論上の違いは、*識別カーソル*の使用によって*select*文を定義するコードとそこからフェッチされるコードを異なるモジュール間に分割できることです。ただし、このようなモジュール化スキームが一般的に適切ではないことが、41 ページの"*プロデューサ/コンシューマのモジュール化のアプローチ*"に記載されています。

値を超えると致命的なエラーが発生するハードコードされた制限を禁止している読者は、常に*バッチ・バルク・フェッチ*を使用する必要があります[72]。ただし、*Code_35*で示されているように、とくにアンバウンド結果セットのN番目のスライスを取得するために*select*文が設計される事例を忘れないでください。少なくともここでは、*全体バルク・フェッチ*を安全に使用できます。

72. エンハンスメント・リクエスト 6616605 は、全体バルク・バッチ（select... bulk collect into、execute immediate... bulk collect into、または fetch... bulk collect into の 3 つのうちのいずれか）が使用される場合に、新しい PL/SQL のコンパイラの警告を要求します。

```
-- Code_35
-- Set the lower and upper bound of the slice.
lb := Slice_Size*(Slice_No - 1) + 1;
ub := lb + Slice_Size - 1;

with Unbounded as (
  select    a.PK, a.v1, Rownum r
  from      t a
  order by  a.PK
  )
select                  Unbounded.PK, Unbounded.v1
bulk collect into       b.Results
from                    Unbounded
where                   Unbounded.r between b.lb and b.ub;
```

41 ページの"*プロデューサ/コンシューマのモジュール化のアプローチ*"のアンバウンド結果セットのスライスの説明に戻ります。

この項の議論は、以下のベスト・プラクティスの原則にまとめられます。

問合せでどのように行の最大数が取得されるかがわかる場合は、select... bulk collect into または execute immediate... bulk collect into を使用して、単一の手順ですべての行をフェッチします。

Principle_12

多くの行が選択されており、結果セットの管理可能な最大サイズが安全に設定された場合、単一の手順ですべての行をフェッチします。埋込み SQL を使用できる場合、カーソルのない PL/SQL 構造 *select... bulk collect into* を使用します。動的 SQL が必要な場合、*execute immediate... bulk collect into* を使用します。処理できる最大サイズで宣言された *varray* にフェッチします。*ORA-22165* の例外ハンドラを実装して、バグ診断をサポートしてください。

複数行の選択 - 実行時までわからない *select list* またはバインディング要件

組織内の職務をサポートする情報システムを実装した場合、多くの列を使用した単一のデータベース表として主要な情報エンティティを表すことが一般的です（さまざまなディテール表の可能性もあります）。たとえば、有名な*HR.Employees*のように各従業員のファクトを記録する人事システムです。このようなシステムの一般的な要件は、表の列にマップされるエンティティ属性に一致条件を入力できる場合および任意で条件を空白のままにできる場合に、エンドユーザーの問合せインタフェースをサポートすることです。オラクルは、このような内部システムを用意しています。この機能の問合せ画面には、約 12 個のフィールドがあります。各属性がテキスト値（氏名や役職など）なので、"greater than"や"between"の条件テストをおこなう必要がありません。ただし、給与などの数値属性を使用する類似したシステムでは、このようなテストが役立つ場合があります。最終的な*select*文の*where*句で*like*述語に各テストが対応する場合でも、問合せ画面に必要なwhere句の数が非常に多くなり[73]、コンパイル時にテキストが固定されてバインディング要件が識別されるSQL文では、個別のwhere句はサポートされません。

73. *N* 列では、where 句の数は、*N* の 1 や *N* の 2 といった選択の合計と同じになります。*N* の（*N-1*）および *N* の *N* が最大になります。これは *2N-1* です。12 列では 4095 になります。

問合せ画面で対象となる条件を選択する際に柔軟性を求める一方で、結果の表示が機能上の仕様で固定される大規模クラスのアプリケーションがあります。実装において、これはコンパイル時に*select list*が識別されることを意味します。このようなユースケースの場合、バインディングのサポートに*DBMS_Sql* APIが必要です。また、ネイティブ動的SQLを使用して結果を取得できます（簡潔にプログラミングをおこない迅速に実行できます）。*Code_36*は、このアプローチを示しています[74]。

```
-- Code_36
DBMS_Sql_Cur := DBMS_Sql.Open_Cursor(Security_Level=>2);

-- Build the select statement
...
DBMS_Sql.Parse(DBMS_Sql_Cur, Stmt, DBMS_Sql.Native);

-- More eleborate logic is needed when the values to be bound
-- are not all the same datatype.
for j in 1..No_Of_Placeholders loop
  DBMS_Sql.Bind_Variable(
   DBMS_Sql_Cur, ':b'||To_Char(j), Bind_Values(j));
end loop;
Dummy := DBMS_Sql.Execute(DBMS_Sql_Cur);

declare
  Cur_Var Sys_Refcursor :=
   DBMS_Sql.To_Refcursor(DBMS_Sql_Cur);
begin
  loop
    fetch Cur_Var bulk collect into Results limit Batchsize;
    for j in 1..Results.Count() loop
     Process_One_Record(Results(j));
    end loop;
    exit when Results.Count() < Batchsize;
  end loop;
  close Cur_Var;
end;
```

このユースケースで*DBMS_Sql* APIを使用した場合の重要な特徴は、実行時に識別されるファクトで制御フローおよび*Bind_Variable()*の呼出しの実際の引数を決定できることです。ネイティブ動的SQLではこのようなスキームを使用できません。*using*句（*Cur*が*カーソル変数*の場合の*execute immediate... into*文または*open Cur for*文）がコンパイル時に固定されるためです[75]。

*バッチ・バルク・フェッチ*の使用には注意してください。このユースケースは、アンバウンド結果セットを示す傾向があります。

SQL 文を構築するロジックは、アプリケーション固有です。一般的に、ユーザーが入力した基準を表す *in* 仮パラメータをテストします。null ではない場合、テキストが SQL 文に追加され、示される条件が表されます。追加されるテキストは *and cn = :bm* 形式で、*cn* は表示される列です。*m* は現在の条件での通常の値を示す runner になります。*Code_37* は、このアプローチを示しています。

---

74. このアプローチは、*DBMS_Sql.To_Refcursor()*ファンクションの使用に依存します。これは、Oracle Database 11gの新機能です。SQL文の情報やセキュリティがデータベースに隠されていても、データベース・クライアントでフェッチが実装されることをシステム・アーキテクチャ全体で推奨していれば、この機能は有効となります。クライアントが呼び出すデータベースPL/SQLサブプログラムで*参照カーソル*に基づく*return*データ型を使用する必要があります。バインディングに*DBMS_Sql* APIが必要な場合でも、これが可能です。ただし、ほかのアプローチが優れていることが41 ページの"*プロデューサ/コンシューマのモジュール化のアプローチ*"に記載されています。

```
-- Code_37
for j in 1..User_Criteria.Count() loop
  if User_Criteria(j) is not null then
   No_Of_Placeholders := No_Of_Placeholders + 1;
   Stmt := Stmt||' and '||
    Column_Names(j)||' = :b'||To_Char(No_Of_Placeholders);
   Bind_Values(No_Of_Placeholders) := User_Criteria(j);
  end if;
end loop;
```

本書で説明されているように、*constant*でSQL文の開始文字列を表すことができます[76]。基準の入力がない場合にSQL文で=を使用し、ワイルドカード文字（%および_）を含む場合に*like*を使用すると、実際のロジックはさらに複雑になります。対象の列に*varchar2*、*number*、*date*などのデータ型が混在し、ユーザーが非等価演算子を指定できる場合（ポップアップ・リストからの選択など）、複雑さが増します。

一般的に組織内の職務をサポートする情報システムに関連して、ユーザーがレポートに含める属性を構成できる状態にしておくことが要件仕様として提示されています。つまり、実装においてコンパイル時に*select list*がわからないため、*DBMS_Sql* APIも使用して結果を取得する必要があることを意味します。*Code_38*は、このアプローチを示しています。

---

75. ネイティブ動的 SQL を順に使用する*無名 PL/SQL ブロック*をプログラムで生成および実行して、コンパイル時に *using* 句が固定されるという現象を顧客が解決しようとする事例を確認しました。通常は、*using* 句をプログラムで構築することで、仮定の制限を解決するというソリューションが主流です。このため、欠点が隠れてしまう巧妙なアプローチを確認できません。このアプローチでは、*DBMS_Sql.Bind_Variable()*と動的にバインドされる値を*無名 PL/SQL ブロック*のテキストのリテラルとして代わりにエンコードする必要があります（これを回避するには、*DBMS_Sql* API を使用して無名 PL/SQL ブロックを実行する必要があり、これによってポイントが変わります）。つまり、生成されたそれぞれの*無名 PL/SQL ブロック*のテキストが以前と異なることを意味します。結果として、実行ごとに*ハード・パース*がおこなわれます。

    ベスト・プラクティスとしてこのアプローチが提案されていることがありますが、このアプローチは典型的なワースト・プラクティスの例であり、注意が必要です。

76. 一般的に、ループを開始する前に *constant* テキストに *where 1=1* を追加します。これによって、連結ロジックがわかりやすくなります。

```
-- Code_38
declare
  ...
  type Results_t is table of DBMS_Sql.Varchar2_Table
    index by pls_integer;
  Results Results_t;
begin
  -- Open the DBMS_Sql_Cur,
  -- build and parse the select statement,
  -- bind to the placeholders, and execute as in Code_36.
  -- This will set No_Of_Select_List_Items.

  ...
  loop
 -- Tell it to fill the target arrays
 -- from element #1 each time.
 for j in 1..No_Of_Select_List_Items loop
    DBMS_Sql.Define_Array(     DBMS_Sql_Cur, j, Results(j),
Batchsize, 1); end loop;

 Dummy := DBMS_Sql.Fetch_Rows(DBMS_Sql_Cur);

 for j in 1..No_Of_Select_List_Items loop
   -- Have to delete explicitly. NDS does it for you.
   Results(j).Delete();
   DBMS_Sql.Column_Value(DBMS_Sql_Cur, j, Results(j));
 end loop;

 for j in 1..Results(1).Count() loop
   -- Process the results.
 ...
 end loop;

  exit when Results(1).Count() < Batchsize;
 end loop;
 DBMS_Sql.Close_Cursor(DBMS_Sql_Cur);
end;
```

完全なソリューションは膨大なので、*Code_38*は概要程度に留めています。*select list*項目のデータ型は一般的に異なるので、対応するデータ型のターゲットが使用される場合は、非常に複雑なアプローチが必要になります。ただし、処理の最終的な目的が、人間によって読み取れるレポートを準備することであれば、各*select list*項目を*varchar2* に変換することで（*select list*で適切な*To_Char()*ファンクションを呼び出すことを推奨します）、アプローチを大幅に簡素化できます。フェッチ・ターゲットとして、要素のデータ型が提供された*DBMS_Sql.Varchar2_Table*コレクションの*index by pls_integer*表を使用できます。つまり、*select list*の構築の結果として*No_Of_Select_List_Items*が識別されたあと、同じ操作を各*select list*項目に適用するループで*Define_Array()*および*Column_Value()*を呼び出すことができます。"Process the results"（結果の処理）のコメントは、プログラミングが複雑になることを示しています。ただし、ロジックは一般的なものであり、SQL処理との関連はなくなります。

ここで *DBMS_Sql* API を使用すると、実行時に初めて識別されるファクトで、制御フローおよび *Define_Array()*と *Column_Value()*の呼出しの実際の引数を決定できるということに注意してください。ネイティブ動的 SQL ではこのようなスキームを使用できません。（*execute immediate... into* 文または *fetch... bulk collect into* 文の）*into* 句がコンパイル時に固定されるためです。

コンパイル時にバインディング要件が識別されて実行時まで*select list*の構成がわからない場合、アプリケーションに指定される要件が実装設計につながる可能性

はほとんどありません。結果のフェッチを処理するためにネイティブ動的SQLを使用し、必要に応じて*カーソル変数*および*DBMS_Sql* APIを開くことができます。*Code_39*は、このアプローチを示しています。

```
-- Code_39
open Cur_Var for Stmt using Some_Value;
DBMS_Sql_Cur := DBMS_Sql.To_Cursor_Number(Cur_Var);

-- The fetch loop is identical to that shown in Code_38.
loop
  for ... loop
    DBMS_Sql.Define_Array(..., Results(j), ...);
  end loop;

  Dummy := DBMS_Sql.Fetch_Rows(DBMS_Sql_Cur);

  for ... loop
 ...
 DBMS_Sql.Column_Value(..., Results(j));
  end loop;

  for j in 1..Results(1).Count() loop
    -- Process the results.
    ...
  end loop;

  exit when Results(1).Count() < Batchsize;
end loop;
```

結果の順序の基準指定を可能にするユーザー・インタフェースが必要な場合、バインディングの実装（*order by* 句でプレースホルダを使用しない）やフェッチの実装に影響しないのでさらに複雑になることはありません。

この項の議論は、以下のベスト・プラクティスの原則にまとめられます。

実行時まで select list のバインディング要件がわからない場合は、DBMS_Sql API を使用します。少なくとも select list が識別されている場合、To_Refcursor()を使用し、次にバッチ・バルク・フェッチを使用します。

Principle_13

リテラルで *where* 句を構成しないでください。とくに、ユーザーが明示的に入力した *where* 句を連結しないでください。プレースホルダをバインドするアプローチよりもパフォーマンスが大幅に低下します。直接 *where* 句を入力した場合、*Sys.DBMS_Assert.Enquote_Literal()*で SQL インジェクションから保護することはできません。実行時までバインディング要件がわからない場合、*DBMS_Sql* API を使用して、SQL 文をパース、バインド、および実行します。コンパイル時に *select list* が識別される場合、*To_Refcursor()*を使用して *DBMS_Sql 数値カーソル*を*カーソル変数*に変換し、*バッチ・バルク・フェッチ*を使用します。実行時まで *select list* がわからない場合、*DBMS_Sql* API を使用して結果もフェッチします。コンパイル時にバインディング要件が識別され、実行時まで *select list* が識別されない珍しい事例の場合、ネイティブ動的 SQL を使用して*カーソル変数*を開き、*To_Cursor_Number()*を使用して*カーソル変数*を *DBMS_Sql 数値カーソル*に変換します。次に、*DBMS_Sql* API を使用して、結果をフェッチします。

単一行の選択

このユースケースの明白な例は、主キーで識別される行の取得です。ただし、最初に想定したよりも一般的ではない可能性があります。マスター表の行の場合、関連する詳細が同時に必要になります。ディテール表の行の場合、特定のマスターのすべての行が同時に必要になります。ただし、単一の行だけを必要とするユースケースもあります。

*Code_40*は、埋込みSQLを使用した例を示しています。

```
-- Code_40
select  a.PK, a.v1
into    b.The_Result
from    t a
where   a.PK = Some_Value;
```

*Code_41*は、ネイティブ動的SQLを使用した例を示しています。

```
-- Code_41
declare
  Stmt constant varchar2(200) := '
    select      a.PK, a.v1
    from        t a
    where       a.PK = :b1';
begin
  execute immediate Stmt
   into The_Result
   using Some_Value;
```

*Code_40*と*Code_32*、*Code_41*と*Code_33*を比較してください。各例では、カーソルのないPL/SQL構造を使用しています。*Code_34*と比較するため、*Code_42*では*Cur識別カーソル*を使用する*全体バルク・フェッチ*方式を示します。

```
-- Code_42
-- Cur is already open here.
-- The fetch syntax is the same for
-- an explicit cursor and a cursor variable.
fetch Cur into The_Result;
close Cur;
```

*Code_43*は、*Code_42*を実行可能にするために*Cur識別カーソル*を設置する 3 つの方法を示しています。複数行の事例である*Code_31*と非常に似ています。

```
-- Code_43
    $if $$Approach = 1 $then
     cursor Cur is
     select      a.PK, a.v1
     from        t a
     where       a.PK = b.Some_Value;
  $elsif $$Approach = 2 or $$Approach = 3 $then
   Cur Sys_Refcursor;
   $if $$Approach = 3 $then
    Stmt constant varchar2(200) := '
     select      a.PK, a.v1
     from        t a
     where       a.PK = :b1';
    $end
   $end
begin
   $if $$Approach = 1 $then
    open Cur;
   $elsif $$Approach = 2 $then
    open Cur for
     select      a.PK, a.v1
     from        t a
     where       a.PK = b.Some_Value;
   $elsif $$Approach = 3 $then
    open Cur for Stmt using Some_Value;
   $end
```

複数行の事例と同様に、*Code_42*のアプローチに必要なコード量の合計は、機能的に同等の*Code_40*や*Code_41*よりも大幅に多くなります。対応する引数が適用されます[77]。

実行時までバインディング要件がわからないシナリオの性質から複数行を返す問合せの可能性が高くなります。このため、望ましくない *No_Data_Found* 例外および予期しない *Too_Many_Rows* 例外が発生する場合、*DBMS_Sql* API の使用は適切ではありません。単一の行だけが要求されているが、実行時まで *select list* がわからないという事例は珍しいので、*DBMS_Sql* API は必要ありません。

この項の議論は、以下のベスト・プラクティスの原則にまとめられます。

単一の行だけを取得するには、select... into または execute immediate... into を使用します。No_Data_Found および Too_Many_Rows を活用してください。

Principle_14

単一の行だけを選択するときは、単一の手順で行をフェッチします。埋込み SQL が使用できれば、カーソルのない PL/SQL 構造 *select... into* を使用します。動的 SQL が必要な場合は、*execute immediate... into* を使用します。望ましくない *No_Data_Found* 例外および予期しない *Too_Many_Rows* 例外を活用してください。

プロデューサ/コンシューマのモジュール化のアプローチ

データベース PL/SQL プログラムのモジュール化を計画している場合、異なる PL/SQL ユニットで次の 2 つの異なる処理を実行することが一般的です。

- SQL 文（*select*、*insert*、*update*、*delete*、*merge*、*lock table*、*commit*、または *rollback*）の実行
- 表を変更するために取得または使用されるデータの処理

説明が（この項で焦点をあてている）*select* 操作だけに制限される場合、プロデューサ/コンシューマ・メタファは正しく適用されます。プロデューサがすべての SQL を管理し、コンシューマは関与しないのが理想的です。ただし、このような明確な区別は現実的に多くの欠点（少なくとも Oracle Database 11g を使用する場合）をもつため、コンシューマがデータベースの外側の場合、推奨されるアプローチにはなりません。

職務の分離の明白な理由は、アクセス制御構造を実施するためです。表データへのすべてのアクセスを専用データ・アクセス PL/SQL ユニットに制限して、多くのビジネス・ルール（どのような種類のデータに依存する変更が許可されるか、どのように非正規化が維持されるかなど）は安全かつ効果的に実装されます。表と同じスキーマの定義者権限のデータ・アクセス PL/SQL ユニットを使用し、ほかのスキーマでこのデータを使用または準備する PL/SQL ユニットを実装して、これを簡単に実施できます。データ・アクセス PL/SQL ユニットの *Execute* 権限は、ほかのスキーマに付与されます。ただし、表の *Select*、*Insert*、*Update*、および *Delete* 権限は付与されません。データベースのクライアントによるアクセスには、同じスキームが便利です。このベスト・プラクティスの原則には、明白な利点があります。

API を定義するオブジェクトのプライベート・シノニムだけを含む専用スキーマを使用して、データベース・アプリケーションを公開します。

Principle_15

アプリケーションの機能にアクセスするために、データベース・クライアントが接続するアプリケーションの専用スキーマ（*Some_App_API* など）を作成します。厳密に保護されたパスワードを使用して、*Some_App_API* 以外のスキーマにすべてのアプリケーションのデータベース・オブジェクトを実装します。

---

77. パフォーマンスおよび機能の観点から、*select... into* または *execute immediate... into* よりも *fetch... into* のほうが望ましいと思っているユーザーもいますが、これらに根拠はありません。fetch... into は、*select... into* および *execute immediate... into* よりもパフォーマンスが低下します。一意キーを使用して単一の行だけを取得する要件の場合、*Too_Many_Rows* 例外だけが役立ちます。通常、*No_Data_Found* 例外は、望ましくありませんが予想外ではありません。リカバリが可能であり、厳密に囲まれた例外ハンドラでプログラムする必要があります。

また、*Some_App_API* のオブジェクトをプライベート・シノニムだけに制限します。クライアント API を公開するアプリケーションのデータベース・オブジェクトだけにこのようなシノニムを作成します。これらのオブジェクトのアクセスに必要な権限だけを *Some_App_API* に付与します。

オラクルの主要な顧客の多くは、次のベスト・プラクティスの原則を採用して、体制を強化しています。

PL/SQL API でデータベース・アプリケーションを公開します。データベースのクライアントがアクセスできないスキーマのすべての表を非表示にします。

Principle_16

*Some_App_API*のプライベート・シノニムがPL/SQLユニットだけに公開されるオブジェクト型を制限します。ほかのスキーマのすべての表を非表示にします。これらの権限を*Some_App_API*[78] に付与しないでください。

表データを取得するタスクの場合、どのようにプロデューサ/コンシューマ API を設計すればよいのでしょうか。データベース内のモジュール化スキームには、次の 3 つの可能性があります。

- パッケージ仕様のパラメータ化された*明示カーソル*の宣言を公開して、本体の定義を非表示にします。カーソルのオープン、フェッチ、およびクローズをクライアントで管理します。
- return データ型が参照カーソルに基づくパッケージ仕様のパラメータ化されたファンクションの宣言を公開して、本体の定義を非表示にします。ファンクションの return が割り当てられる*カーソル変数*のフェッチおよびクローズをクライアントで管理します。
- return データ型が取得データを表現するように設計されるパッケージ仕様のファンクションの宣言を公開して、本体の定義を非表示にします。通常、return データ型は、*レコード*（パラメータ化が常に単一の行を示す場合）、*レコードのコレクション*、または XML ドキュメント（パラメータ化が複数行を示す場合）として実装されます。

データベースの外側のコンシューマにデータベースを公開する API を定義するには、クライアントが PL/SQL で実装されている場合（つまり、Oracle Forms の場合）のみ、最初のアプローチを使用できます。埋込み SQL で満たすことができる問合せ要件のみサポートされていることも障害になります。コンシューマがデータベースの内側の場合でも推奨されているわけではないので、これ以上取り上げません。

データベースの外側のコンシューマにデータベースを公開するAPIを定義する場合、常に 2 つ目のアプローチを使用できます。Oracle DatabaseのSQL文の処理をサポートするすべてのクライアント環境には、*無名PL/SQLブロック*の実行をサポートするAPIが含まれます。とくに、ブロックの変数として記述された場合、*カーソル変数*として宣言されるプレースホルダに適切なクライアント・データ構造をバインドできます。このアプローチを使用すると、理想的なモジュール化の概念が変わってしまいますが（*参照カーソル*からのフェッチを実装するために、SQLの仕組み[79] をクライアントが把握する必要があるため）、多くの顧客がこのアプローチを採用し、ミッション・クリティカルな本番コードで使用しています。*ADT*のコレクションを使用する必要がある理想的な概念は非常に複雑なことがわかります[80]。

78. これは、ソフトウェア・エンジニアリングの一般的なベスト・プラクティスの原則を Oracle Database 用に特殊化したものです。わかりやすい API を使用して慎重に設計された抽象化レベルでシステムをモジュールに分割し、各モジュールの機能を公開してこの API のモジュールの実装を非表示にします。Oracle Database の PL/SQL サブプログラムは、API を定義する手段を提供します。モジュール実装の一部である表およびコンテンツを操作する SQL 文を、クライアントからデータベースでは見えないようにする必要があります。

3 つ目のアプローチでクライアントにデータベースを公開するAPIを定義するには、少し修正が必要です。こうしたすべてのAPIがプレースホルダへの適切なクライアント・データ構造のバインディングをサポートしているわけではありません。プレースホルダは、対応する仮パラメータがレコードまたはレコードのコレクションであるサブプログラムの呼出しにおいて、実際の引数としての役割を果たします。ただし、ターゲットの仮パラメータが*ADT*[81]または*ADT*のコレクションの場合、すべてのAPIがバインディングをサポートします。71 ページの*"付録C：レコードのコレクションにselect文の結果を移入する代替のアプローチ"*は、この目的のいくつかの構造を示しています。人間が読み取れるレポートを準備することがクライアントの目的である場合、returnデータ型としてXMLドキュメントを選択することがとくに有用です。

3 つ目が理論的にもっともわかりやすい API 設計です。プロデューサは、データ取得のすべての処理をカプセル化します。コンシューマは、正しく指定された表現のデータのみを確認できます。

この 3 つ目のアプローチに関連して、コンシューマとプロデューサのステートフルな関係下でアンバウンド結果セットをフェッチする方法を確認してから、ステートレスな関係の事例を確認すると有用です[82]。

プロデューサ/コンシューマのステートフルな関係

コンシューマおよびプロデューサが、異なるスキーマのパッケージのPL/SQLサブプログラムとして実装される場合などのステートフルな事例[83]で、状態はどのようになり、誰が保持するのでしょうか。状態は、次にフェッチされて*カーソル変数*[84]に保存される結果セットのメンバーです。ただし、Oracle Database 11gでは、パッケージの仕様または本体のグローバル・レベルで*カーソル変数*が宣言されない場合があります[85]。解決策として、すべてのバッチが処理されるまで、コンシューマが保持できる状態にプロデューサがハンドルを戻す方法があります[86]。*Code_44*は、*プロデューサ・パッケージ*の仕様を示しています。

---

79. とくに、コンシューマは、プログラミング環境に*全体バルク・フェッチ*または*バッチ・バルク・フェッチ*と同等の機能を実装する必要があります。

80. 複雑ですが、これを実装する方法を説明します。

81. create type... as object(...)の結果の便利なシノニムとして、本書では ADT を使用します。ADT を Object_Type が type のオブジェクトとして参照すると非常に複雑になります。また、これはコレクション型というよりもオブジェクト型だといえます。

82. コンシューマがデータベースの内側の場合、関係が自動的にステートフルになります。コンシューマがデータベースの外側の場合、関係が一般的にステートレスになります。

83. 近年、*新規*プロジェクトでユーザー向けアプリケーションを構築し、ユーザー・インタフェースにデータベースへのステートフルな接続が装備されているアーキテクチャを選択することは比較的まれです。ほぼ例外なく、ステートレスな HTML ブラウザのユーザー・インタフェースを実装することが一般的です。注目すべき例外として、データベース開発の IDE（Oracle SQL Developer など）があります。

84. ここでは、*カーソル*という用語の隠喩的な意味を理解します。

85. この制限に根拠はありません。エンハンスメント・リクエスト 6619359 は、解除を要求します。

```
-- Code_44
package Producer is
  type Result_t is record(PK t.PK%type, v1 t.v1%type);
  type Results_t is table of Result_t index by pls_integer;
  function The_Results(
    Some_Value in t.PK%type,
    Cur_Var in out Sys_Refcursor)
    return Results_t;
end Producer;
```

*Code_45*は、本体を示しています。

```
-- Code_45
package body Producer is
  function The_Results(
    Some_Value in t.PK%type,
    Cur_Var in out Sys_Refcursor)
    return Results_t
  is
   Stmt constant varchar2(200) := '
    select        a.PK, a.v1
    from  t a
    where         a.PK > :b1
    order by      a.PK';
   Batchsize constant pls_integer := 1000;
   Results Results_t;
begin
  if Cur_Var is null then
    open Cur_Var for Stmt using Some_Value;
  end if;

  fetch Cur_Var bulk collect into Results limit Batchsize;

  if Results.Count() < Batchsize then
    close Cur_Var;
    Cur_Var := null;
  end if;

 return Results;

  exception when others then
  if Cur_Var%IsOpen then
    close Cur_Var;
  end if;
  raise;
 end The_Results;
end Producer;
```

*Code_46*は、コンシューマ・プロシージャを示しています。

```
-- Code_46
procedure Consumer is
  Some_Value constant t.PK%type := 0;
  Cur_Var Sys_Refcursor := null;
  Results Producer.Results_t;
begin
  loop
    Results := Producer.The_Results(Some_Value, Cur_Var);
    for j in 1..Results.Count() loop
      ...
    end loop;
    exit when Cur_Var is null;
  end loop;
end Consumer;
```

ネイティブ動的SQLを使用する必要がない場合、パッケージ本体の最上位レベルで宣言されている*明示カーソル*を代わりに使用できます[87]。コンシューマはプロデューサへの呼出しの間に*参照カーソル*を"保持"する必要がないので、設計が簡素化されます。

---

86. これは非常に一般的なパラダイムです。たとえば、*DBMS_Sql* および *Utl_File* で使用されます。

コンシューマはプロデューサへの呼出しの間に*参照カーソル*を"保持"する必要がないので、設計が簡素化されます。埋込みSQLとネイティブ動的SQLの両方でこのアプローチを使用できるので、ここでは*参照カーソル*を使用しました。*DBMS_Sql* APIを使用する要件の場合、プロデューサの設計が大幅に変更されることがあります。状態は、*Open_Cursor()*から返された数値です。プロデューサ・パッケージの本体の最上位レベルで宣言された変数へこの値を簡単に保存できます。コンシューマが呼出しの間に*参照カーソル*を"保持"する必要がないことを除いて、コンシューマはプロデューサの実装におけるこのような違いを確認しません。防衛的設計では、3 つのすべての方法に対してAPIで*参照カーソル*を公開し、プロデューサを実装できます。実装に*明示カーソル*または*DBMS_Sql* APIを使用した場合も違いはありません[88]。

プロデューサ/コンシューマのステートレスな関係

ステートレスなHTMLブラウザでユーザー・インタフェースを実装するOracle Databaseアプリケーションでは、"次のページ"および"前のページ"ボタンまたはN番目のページに直接移動できるボタンを使用して、バッチの問合せ結果を表示することが非常に一般的です。このアーキテクチャは、各ページ・ビュー・リクエストがミドルウェアからデータベースへの呼出しで満たされることを示します。このデータベースは、前のページ・ビューの呼出しではなく異なるセッションから呼び出されます[89]。つまり、これを実行するためには、必要なページを取得する問合せをパラメータ化する必要があります。ステートフルな構造とは異なり、最後にアクセスした場所からは続行できません。*Code_35*は、このような問合せを示しています。これは、*全体バルク・フェッチ*・アプローチに完全に対応しています。

この項の議論は、以下のベスト・プラクティスの原則にまとめられます。

return データ型が任意のデータを表すファンクションとして、プロデューサ/コンシューマ API を定義します。プロデューサ・モジュールのすべての SQL 処理を非表示にします。これによって、コンシューマは、要件の変更が原因の実装の変更による影響を受けなくなります。問合せのパラメータ化のようにプロデューサ・ファンクションをパラメータ化してください。このアプローチは、バッチでの行の取得や単一の呼出しでのすべての行の取得に対応します。これがスライスになる場合もあります。

Principle_17

return データ型が作成されたデータを表す機能として、プロデューサ/コンシューマ API を定義します。プロデューサ・モジュールで、SQL 処理に関連するすべての処理（フェッチを含む）を非表示にします。問合せのパラメータ化で単一の行のみを指定する場合、*select list* と同じ構造で *レコード*または *ADT* を使用します。この場合、動的 SQL の要件に応じてカーソルのない PL/SQL 構造の *select... into* または *execute immediate... into* を使用します。問合せのパラメータ化で複数の行を指定する場合、*レコード*のコレクションまたは *ADT* のコレクションを使用します。ここで安全であることが確認できれば、*全体バルク・フェッチ*を使用します。これによって、カーソルのない PL/SQL 構造の *select... bulk collect into* または *execute immediate... bulk collect into* を使用できます。*全体バルク・フェッチ*が安全ではない場合、プロデューサ/コンシューマの関係がステートフルなときに*バッチ・バルク・フェッチ*を使用します。これには*識別カーソル*が必要です。埋込み SQL で十分な場合、プロデューサ・パッケージの本体のグローバル・レベルで宣言された*明示カーソル*を使用します。これによって、各バッチを取得するコンシューマからの呼出しの状態が保持されます。動的 SQL が必要な場合、*カーソル変数*を使用します。コンシューマが保持できるように、各バッチの結果とともにこれをコンシューマに戻します。プロデューサ/コンシューマの関係がステートレスな場合、結果セットのスライスを使用します。

87. 本書のアプローチでは、パッケージ仕様の*明示カーソル*の宣言とパッケージ本体の定義を分割する機能は活用していません。
88. Oracle Database 11g の *DBMS_Sql.To_Refcursor()*ファンクションも確認してください。*select list* の構成がコンパイル時に識別される場合、ネイティブ動的 SQL を使用してわかりやすいフェッチ・コードを記述できます。
89. このパラダイムは、読取り一貫性の問合せ結果を表示する典型的なプリファレンスを意図的に放棄します。

*全体バルク・フェッチ*で各スライスの配信を実装します。要件に示されている場合、*DBMS_Sql* API を使用して、プロデューサ・モジュールでこれを使用するすべてのコードを非表示にします。

Oracle Database 11gには、PL/SQLファンクション結果キャッシュが導入されています。プログラマーは、*authid*が使用される構文で*result_cache*キーワードを使用します。ファンクションの起動に使用する実際の引数の組合せごとに戻り値をキャッシュして、同じ引数を使用した後続の呼出しの再計算を少なくします。すべてのセッションでキャッシュにアクセスできます。*return*データ型は、レコードまたはADTあるいはそのコレクションに基づいています。つまり、ファンクションの戻り値の計算が表から取得されるデータに依存する場合に適したPL/SQLの*メモ化*[90]です。プログラマーは、宣言的な*relies_on*句を使用して、コンテンツがファンクションの結果に影響する表を示すことができます。変更がこれらの表にコミットされる場合、ファンクションのキャッシュ結果がパージされます。プロデューサ・ファンクションが単一の行または少数の行を返し、ファンクションが依存する表データが頻繁に変更されない場合、*result_cache*でファンクションをマークすることで、パフォーマンスを大幅に向上できます。この明白な例は、代理の主キーと対応する人間が読み取れる一意キー（ユーザー・インタフェースで制御する値リストを移入するキー）のマッピングを返すファンクションです。

90. この技術は、ソフトウェア・エンジニアリングでよく知られています。Wikipediaに次の説明があります。*コンピュータ分野におけるメモ化とは、おもにコンピュータ・プログラムを高速化するために使用される最適化技法です。Donald Michieが1968 年に構築しました。メモ化された関数は、一連の特定の入力に対応する結果を"記憶"します。記憶された入力を使用した後続の呼出しでは、再計算するのではなく、記憶されている結果を返します。*

INSERT、UPDATE、DELETE、および MERGE 文のアプローチ

ユーザー・インタラクションでは、単一の行の *insert*、*update*、*delete*、または *merge* を使用する場合が一般的なので、このユースケースを最初に扱います。*select* とは異なり、複数の行を使用する場合にこれらの操作を無制限の多くの行でサポートする必要はありません。任意の操作を表すデータは PL/SQL データ構造で最初に発生するので、複数行のバルク操作に合わせてバッチ処理を実行できます。

この項のコードでは、埋込みSQLを使用します。常にカーソルのないシングルトンPL/SQL構造の文を使用します。*識別カーソル*を使用する可能性はないため、すべての操作で*暗黙カーソル*を使用します。つまり、ユースケースで必要になる場合、ネイティブ動的SQLへの移行が機械的になります。57 ページの*"insert、update、delete、およびmergeのネイティブ動的SQLの使用"*で、これを簡単に説明します。これらのSQL文に対する*DBMS_Sql* APIの使用は説明しません。最初に必要だと判断したユースケースでこれを回避する方法を示します[91]。

単一の行の操作

単一の行のシナリオの課題は、*insert* および *update* で一部の列だけに値を指定する処理です。

単一の行の *insert*

*Code_47*は、*insert*の 2 つの事例を示しています。

```
-- Code_47
insert into t(PK, n1) values (b.PK, b.n1);
...
insert into t(PK, v1) values (b.PK, b.v1);
```

*Code_48*の宣言[92]で*Insert_Row_Into_T()*プロシージャの設計および実装を検討してください。

```
-- Code_48
procedure Insert_Row_Into_T(
  PK in t.PK%type,
  n1 in t.n1%type := null,
  n1_Specified in boolean := false,
  ...
  v1 in t.v1%type := null,
  v1_Specified in boolean := false,
  ...)
  authid Current_User;
```

35 ページの*"複数の行の選択 - 実行時までわからない select list またはバインディング要件"*で、SQL文にすべてが記述されない可能性のある 12 個の列で組合せ的爆発が発生することを説明しました。可能性のあるすべての事例に対応するには、1,000 以上の個別の埋込みSQL文が必要になります。コンパイル時までバインディング要件がわからない*select*文と同じように、*DBMS_Sql* APIを使用できます。

---

91. バインディング要件または *select list* の構成が実行時までわからない *select* 文に対する *DBMS_Sql* API の使用を習得したユーザーは、*insert*、*update*、*delete*、または *merge* 文に対しても簡単に使用できます。

92. オプションの値を設定した列に対応する各形式に指定されたかどうかを示す *boolean* が設定されています。プロシージャで *null* を判別するテストを実行してパラメータがデフォルトかどうかを確認しないのは、一般的に *null* が任意の値になる可能性があるためです。

ただし、適切な代替手段がある場合は、これを回避するというベスト・プラクティスの原則（17 ページを参照）を確認してください。この場合の代替手段は、各列の値を指定して、指定されない列にはデフォルト値を設定する文を使用することです。

Oracle9i Database Release 2 で、埋込みSQLのバインド引数としての *レコード*の使用に対応しました。これによって、この課題に簡単に対処できます。*Code_49*は、デフォルト値にアクセスする方法を示しています[93,94]。

```
-- Code_49
insert into     t(PK)
values          (PK)
returning       PK, n1, n2, v1, v2
into            New_Row;

if n1_Specified then
  New_Row.n1 := n1;
end if;
...
update  t
set     row = New_Row
where   t.PK = New_Row.PK;
```

ただし、このアプローチには、1 つで十分であるにもかかわらず 2 つのPL/SQL→SQL→PL/SQLコンテキスト・スイッチ[95]が必要であるという欠点があります。デフォルト値を取得する最適な方法は、デフォルト値を定義する *レコード*型を使用することです。プログラマーは、無名の*t%rowtype*や対応する*t.PK%type*などの項目が列の制約やデフォルト値を表から継承しないことを忘れてしまう場合があります[96]。ただし、アプリケーションのインストール・スクリプトおよびパッチ/アップグレード・スクリプトを構築する場合は、原則に従えば問題になりません。73 ページの*Code_81*は、本書の例に使用されるテスト表*t*のアプローチを示しています。これは、次のベスト・プラクティスの原則につながります。

各アプリケーション表で、同じ制約およびデフォルト値を定義するテンプレートのレコード型を保存します。

Principle_18

各アプリケーション表の *レコード*型の宣言を公開するパッケージを保存します。宣言では、表の特徴を示す列名、データ型、制約、およびデフォルト値の指定を繰り返す必要があります。

93. *Rowid* を返すと効率的かもしれませんが、これは *レコード*の使用を妨げます。

94. *returning row into A_Record* の使用についてですが、この構文はサポートされていません。エンハンスメント・リクエスト 6621878 でこれを要求します。

95. 30 ページの*"複数の行の選択 - アンバウンド結果セット"*を参照してください。

96. この理由は非常に複雑です。たとえば、スキーマ・レベルの表には、デフォルト値のない *not null* 制約の列が存在する場合があります（値を提供しないと、*insert* のエラーが発生します）。しかし、PL/SQL は、*not null* 変数または *レコード*・フィールドにはデフォルト値があると主張します。また、表の列にチェック制約がある場合、ほかの複雑さが発生します。

テンプレートの *レコード*型を配置して、*Code_50*として*Code_49*を書き換えることができます。

```
-- Code_50
  New_Row Tmplt.T_Rowtype;begin
  New_Row.PK := PK;
  if n1_Specified then
 New_Row.n1 := n1;
  end if;
  ...
insert into t values New_Row;
```

単一の行の *update*

*Code_51*は、*update*の 2 つの事例を示しています。

```
-- Code_51
update  t a
set     a.n1 = b.n1
where   a.PK = b.PK;
...
update  t a
set     a.v1 = b.v1
where   a.PK = b.PK;
```

*Code_52*の宣言[97]で*Insert_Row_Into_T()*プロシージャの設計および実装を検討してください。

```
-- Code_52
procedure Update_T_Row(
  PK in t.PK%type,
  n1 in t.n1%type := null,
  n1_Specified in integer := 0,
  ...  v1 in t.v1%type := null,
  v1_Specified in integer := 0
  ...)
  authid Current_User;
```

*Code_51*に示されているような埋込みSQL文を使用する実装では、組合せ的爆発が発生します。*Code_49*と同じ目的のアプローチでは、任意の新しい行を *レコード*に取得して指定されたフィールドだけを変更し、*update... set row...*を使用します。*Code_53*はこれを示します[98]。

```
-- Code_53
select  * into The_Row from t a
where   a.PK = Update_T_Row.PK
for update;

if n1_Specified then
  The_Row.n1 := n1;
end if;
...
update t a set row = The_Row where a.PK = Update_T_Row.PK;
```

97. *integer* として *n1_Specified* などを宣言する理由は、すぐに明らかになります。

98. *Code_53*は、*n1_Specified*などが*boolean*のバージョンの*Update_T_Row()*を使用します。

*Code_54*は、2 つのPL/SQL→SQL→PL/SQLコンテキスト・スイッチを回避するアプローチ[99]を示しています。

```
-- Code_54
update  t a
set     a.n1 = case n1_Specified
                 when 0 then a.n1
                 else       Update_T_Row.n1
               end,
        a.v1 = case v1_Specified
                 when 0 then a.v1
                 else       Update_T_Row.v1
               end
where  a.PK = Update_T_Row.PK;
```

これで、*integer* として *n1_Specified* などが宣言される理由が明らかになりました。SQL は、*boolean* データ型を識別しません。

単一の行の *delete*

単一の行を一意なキーで識別する必要があるので、実行時までわからないバインディング要件のジレンマは生じません。*Code_55*に例を示します（場合によっては、削除された行のすべての値の監査を保持する必要があります）。

```
-- Code_55
  Old_Row t%rowtype;
begin
  delete    from t a
  where     a.PK = b.PK
  returning  a.PK, a.n1, a.n2, a.v1, a.v2
  into      Old_Row;
```

単一の行の *merge*

*merge*文のサポートがOracle9i DatabaseのSQLに追加されました。また、埋込みSQLにはこのサポートが自動的に継承されます[100]。その目的は、*update*または*insert*用にソース表から互換性のある構造の宛先表に移動する行を*選択*することです。ソース表と宛先表の名前のついた列のペアの値に従って、選択がおこなわれます。このため、この機能は、非公式ではありますが、"upsert"と呼ばれることもあります。この項では、PL/SQLの*merge*文を使用してPL/SQL変数で表される行のupsertを実行します。

単純なSQLの例から開始します。表*t*と完全に同じ列定義をもち、表*t*で値が*PK*として表される行を含む表*t1* の場合とそれ以外を想定します。*Code_56*は、*t1* で一致する行がもつ値を利用して、一致する*PK*値で*t*の行を*更新*し、*PK*と一致しない場所に*t1* の残りの行を*挿入*するSQL文[101]を示しています。

---

99. パフォーマンスの調査は、まだおこなわれていません。理論上の欠点は、*Code_53*と*Code_54*のアプローチで、不要な場合でも各フィールドにアクセスしてしまうことです。ただし、*DBMS_Sql* API を使用してこの欠点を回避する代替手段にも、パフォーマンスの欠点があります。

100. Oracle9i Database では、PL/SQL コンパイラに"一般的な SQL パーサー"が導入されました。これ以前のリリースでは、埋込み SQL がサポートするクラスの文により PL/SQL コンパイル・エラーが発生する場合がありました。現在は不要ですが、回避方法は、このような文に動的 SQL を使用することです。

101. 空の行を交互に配置してこのような長い文を読みやすくするには、`SET SQLBLANKLINES ON` `SQL*Plus` コマンドを使用できます。

```
-- Code_56
merge into t Dest
using      t1 Source
on        (Dest.PK = Source.PK)

when matched then update set
  Dest.n1 = Source.n1,
  Dest.n2 = Source.n2,
  Dest.v1 = Source.v1,
  Dest.v2 = Source.v2

when not matched then insert values (
  Source.PK,
  Source.n1,
  Source.n2,
  Source.v1,
  Source.v2)
```

構文が冗長なように見えます。ただし、ソース表と宛先表で同じ列名を使用する必要はありません。*Code_56*が示す一般的な事例は、省略できません。

末尾にセミコロンを追加することで*Code_56*のSQLから適切なPL/SQL埋込みSQL文を作成できます。ただし、便宜上、プレースホルダを使用するSQL文に対応させる必要があります。また、周知のように、プレースホルダは識別子の代わりとしては不適切です。*Code_57*[102]の標準的なSQL文がヒントになります。

```
-- Code_57
merge into t Dest
using     (select
                1 PK,
                51 n1,
                101 n2,
                'new v1' v1,
                'new v2' v2
                from Dual) Source
on  (Dest.PK = Source.PK)

when matched then update set
  Dest.n1 = Source.n1,
  Dest.n2 = Source.n2,
  Dest.v1 = Source.v1,
  Dest.v2 = Source.v2

when not matched then insert values (
  Source.PK,
  Source.n1,
  Source.n2,
  Source.v1,
  Source.v2)
```

102. この文は、SQL文の適切な形式を保証するルールの作成が不可能であることを示しています。*Code_35*のように*with*句で改善しようとしましたが、適切な構文を作成できませんでした。

ここでは、処理したあとにソース行のあるPL/SQLユニットでこの手段を使用する方法を、*Result レコード*で簡単に確認できます[103]。*Code_58*にこの方法を示します。

```
-- Code_58
  Result t%rowtype;
begin
  ...
  merge into    t Dest
  using         (select
                    Result.PK PK,
                    Result.n1 n1,
                    ...,
                    Result.v1 v1,
                    ...
                from Dual d) Source
on              (Dest.PK = Source.PK)

when matched then update set
  Dest.n1 = Source.n1,
  ...,
  Dest.v1 = Source.v1,
  ...

when not matched then insert values (
  Source.PK,
  Source.n1,
  ...,
  Source.v1,
  ...);
```

比較のため、*Code_59*には*merge*文を使用しないで"upsert"要件を満たす方法を示します。

```
-- Code_59
  Result t%rowtype;
begin
  ...
  begin
    insert into t values Result;
  exception when Dup_Val_On_Index then
    update      t a
    set         row = b.Result
    where       a.PK = b.Result.PK;
  end;
```

表形式の変更によるコード・メンテナンスに*レコード*を使用する利点があるので、*Code_58*よりも*Code_59*の方が魅力的に見えます。ただし、パフォーマンス・テストでは、"正式"な*merge*アプローチのほうが大幅に高速化されています。多くの人は、記述しやすいコードよりも、維持がたいへんであっても優れたパフォーマンスを発揮するコードを使用します。これは、次のベスト・プラクティスの原則につながります。

"upsert"要件には、merge を使用します。例外ハンドラの insert と一緒に update... set row...を使用しないでください。

Principle_19

"upsert"が必要な場合は、*merge* を使用してください。*update... set row...*を実装して、対応する *insert* を実装した *Dup_Val_On_Index* の例外ハンドラを用意するよりも有効です。

---

103. 確認しやすいですが、構文の記述は簡単ではありません。ただし、（パフォーマンスの向上につながるため）提供されるセマンティックには利用する価値があります。

複数行の操作

連続して何度も特定の*insert*、*update*、*delete*、または*merge*文を実行するために使用される一連の値は、PL/SQLユニットで算出するのが一般的です。初心者は、*Code_60*のようなコードを記述します。

```
-- Code_60
for ... loop
  ...
  PK := ...
  n1 := ...
  ...
  update  t a
  set     a.n1 = b.n1, ...
  where   a.PK = b.PK;
end loop;
```

Oracle8i Databaseでは、*forall*文の導入により、*insert*、*update*、*delete*、および*merge*[104]文を繰り返し使用する状況における効率が改善されました。*Code_61*は、*forall*文を使用して書き換えられた*Code_60*を示しています。

```
-- Code_61
for ... loop
  ...
  PKs(j) := ...
  n1s(j) := ...
  ...
end loop;
forall j in 1..PKs.Count() loop
  update    t a
  set       a.n1 = b.n1s(j), ...
  where     a.PK = b.PKs(j);
```

バインドされるコレクションの要素の数が*N*の場合に*forall*文を実装することによって、*Code_60*を*N*回切り替えるのではなく、1 回のPL/SQL→SQL→PL/SQLコンテキスト・スイッチ[105]でSQL文の*N*回の実行を管理して、効率性を改善します。パフォーマンスが何倍にも改善されることが小規模なテストで示されています。

*forall*文を使用することがもっとも重要です。使用しない理由がありません。*Code_61*は、*Code_60*よりも記述や理解が困難ではありません。表現も同様です。たとえば、*Code_61*を少し書き換えることで、*レコード・バインド*で*set row*構文を使用できます。これにより、パフォーマンスの利点も大きくなります。

104. Oracle9i Database まで *merge* 文は導入されていませんでした。導入後は、*forall* 文でもサポートされています。

105. 30 ページの*"複数の行の選択 - アンバウンド結果セット"*を参照してください。

forall 文の実行時に発生する例外の処理

*Code_62*は、*Code_60*の単一の行のアプローチを変更する方法を示しています。特定の繰返しが望ましくないが予想外でもない例外を引き起こした場合に、その実行を継続させます。

```
-- Code_62
for ... loop
  ...
  PK := j;
  n1 := j;
  ...
  begin
    update   t a
    set     a.n1 = b.n1, ...
    where    a.PK = b.PK;
  exception
  when Dup_Val_On_Index then
   n := n + 1;
   The_Exceptions(n).Error_Index := j;
   The_Exceptions(n).Error_Code := SqlErrm();
  when ... then
  ...
  end;
end loop;
```

The_Exceptions は、*index by pls_integer* 表です。この要素は、例外を引き起こした繰返しの回数を保持するフィールドと、エラー・コードを保持するフィールドをそれぞれもった*レコード*です。*Dup_Val_On_Index* 以外の例外（容量不足エラーなど）は、大局的にこれだけがリカバリ可能だと見なされるので、設計上適切に処理するために上位層にバブルアップされます。

*Code_63*は、特定の繰返しで例外が発生した場合に、*Code_61*のバルク・アプローチを変更して実行を継続する方法を示しています。

```
-- Code_63
for ... loop
  ...
  PKs(j) := ...
  n1s(j) := ...
  ...
end loop;

declare
  Bulk_Errors exception;
  pragma Exception_Init(Bulk_Errors, -24381);
begin
  forall j in 1..PKs.Count() save exceptions
    update   t a
    set      a.n1 = b.n1s(j), ...
    where    a.PK = b.PKs(j);
exception
  when Bulk_Errors then
 for j in 1..Sql%Bulk_Exceptions.Count() loop
   The_Exceptions(j).Error_Index :=
     Sql%Bulk_Exceptions(j).Error_Index;
   The_Exceptions(j).Error_Code :=
     -Sql%Bulk_Exceptions(j).Error_Code;
 end loop;
end;
```

実際のコードでは、事前に定義された*Sql%Bulk_Exceptions*コレクションから、ローカルの*The_Exceptions*コレクションにコピーするループが理解できない場合があ

ります。*Code_63* をこのように記述したのは、バルク・アプローチの *Sql%Bulk_Exceptions* の情報が単一の行のアプローチの手動で移入した *The_Exceptions*の情報と同じであることを強調するためです。

若干のセマンティックの違いを確認してください。単一の行のアプローチでは、*Dup_Val_On_Index* だけを取得するハンドラを使用できました。バルク・アプローチの *save exceptions* は、*ORA-24381* を取得して実装される *when others* ハンドラのように機能します。*Dup_Val_On_Index* だけがリカバリ可能な設計では、*ORA-24381* のハンドラの *Sql%Bulk_Exceptions* を横断して、*Error_Code* の値が *Dup_Val_On_Index* に対応する値ではない場合に新しい例外を意図的に発生させる必要があります。

補足：DML エラー・ロギング

DML エラー・ロギングは、Oracle Database 10g Release 2 で導入されました。*insert* 文などの特殊な構文を使用すると、特定の行でエラー（*varchar2* 値は、ターゲット・フィールドの許容値よりも大きくなる可能性があります）が発生した場合に、その行をスキップして次の行の操作を継続するように要求できます。また、スキップした行の情報が（自律型トランザクションで）この目的のために指定した表へ書き込まれます。

この機能は、特定のシナリオを考慮して導入されました。つまり、既知のダーティ・データ[106]（検出後に手動で破棄または修正し、2 回目のロードで使用するデータ）をもつソースから表をバルク・ロードするために*insert... select...*を使用する場合です。このアプローチは、PL/SQLを使用して*暗黙カーソルFORループ*でソース行をステップ実行するハンド・コーディング・アプローチ（各行をターゲット表に挿入し、例外が発生するごとに対処するアプローチ）としては、大きなパフォーマンス上の利点があります。また、このハンド・コーディング・アプローチは、*バッチ・バルク・フェッチ*および*insert*の*forall*文を使用することで最適化できます（このアプローチの例は、59 ページの*Code_68*に示されています）。このような最適化であっても、DMLエラー・ロギングを使用して不良データをスキップするアプローチの方が、PL/SQLアプローチよりも大幅に高速になります。また、単一のSQL文としてプログラミングすることが非常に容易になります。

この結果から、そのほかの理由からすでに PL/SQL を使用して *forall* 文で記述されているコードにおいても、（*save exceptions* 句を使用して *ORA-24381* のハンドラを実装する代わりに）汎用的な代替手段として DML エラー・ロギングを推奨する開発者もいます。*forall* 文の本来の目的をよく考え、*insert*、*update*、*delete*、または *merge* を使用する場合の選択肢を慎重に検討することを推奨します。

2 つのアプローチにおけるセマンティックの違いに注意してください。通常、*insert* よりも *update* および *delete* の違いが大きくなります。このため、*insert* の *forall* 文の一般的な使用では、1 回の繰返しでソース PL/SQL コレクションから単一の行を挿入します。ただし、*update* および *delete* の場合は、1 回の繰返しで多くの行に影響を与えます。*forall* 文の失敗の粒度は、繰返しです。ただし、DML エラー・ロギングの失敗の粒度は、単一の行です。一方のパラダイムが優れているということはありません。つまり、特定の要件によって適切な選択が決定します。DML エラー・ロギングでアプリケーションの通常のオブジェクトとして考慮する必要がある表に、失敗したデータをコミットすることにも注意してください。このスキームには、セッション固有の概念がありません。

---

106. 簡潔なユースケースは、外部表としてファイル・システムに外部システムのデータを表示します。

このため、（管理者によるバルク・ロードとは対照的に）マルチユーザー・アプリケーションでは、適切な特注のタグ付けメカニズムでこの概念を導入する必要があります。つまり、エラー表のコンテンツに特注の状態監視が必要になることを意味します。DML エラー・ロギングでは、すべての種類のエラーを処理できないことにも注意する必要があります。たとえば、遅延制約違反、容量不足エラー、または一意な制約あるいは索引違反を発生させる *update* や *merge* 操作は処理できません。

forall 文の *レコード*のフィールドの参照

Oracle Database 11gでは、プログラマーから強い不満の出ていた制限がなくなりました。*Code_64*を参照してください。

```
-- Code_64
loop
  fetch Cur bulk collect into Results limit Batchsize;

  for j in 1..Results.Count() loop
    ...
  end loop;

  forall j in 1..Results.Count()
    update t a
    set    a.v1 = b.Results(j).v1
    where   Rowid = b.Results(j).Rowid;

  exit when Results.Count() < Batchsize;
end loop;
```

以前のバージョンでは、このコードのコンパイルは失敗し、*PLS-00436: implementation restriction: cannot reference fields of BULK In-BIND table of records*エラーが発生します。これを回避するには、表の列ごとにスカラーの個別のコレクションを使用する必要がありました。*Code_68*（59 ページを参照）は、以前のバージョンのOracle Databaseで*Code_64*と同等の内容を記述した事例です。

バルク・マージ

単一の行の*merge*文の"バルク化"には、*forall*文を使用した単一の*insert*、*update*、および*delete*の変換と同じ論理を適用したくなります。*Code_65*は、この場合のコードを示しています。

```
-- Code_65
  type Results_t is table of t%rowtype index by pls_integer;
  Results Results_t;
begin
  ...
  forall j in 1..Results.Count()
     merge into  t Dest
     using        (select
                       Results(j).PK PK,
                       Results(j).n1 n1,
                       ...,
                       Results(j).v1 v1, ...
                    from Dual d) Source
     on           (Dest.PK = Source.PK)
     when matched then update set
        Dest.n1 = Source.n1,
        ...,
        Dest.v1 = Source.v1, ...
     when not matched then insert values (
        Source.PK,
        Source.n1,
        ...,
        Source.v1,
        ...);
```

どのように*merge*文が記述されているかを検討してください。その目的は、*update*または*insert*のためにソース表から異なる宛先表に移動する行を選択することです。オブジェクト・データ型やコレクション・データ型はスキーマ・レベルで定義された条件下で、データ型がオブジェクトのコレクションであるPL/SQL変数とともにtable演算子を使用できるということを覚えていれば、これを活かすことができます。*Code_66*は、これを実行するSQL*Plusスクリプトを示しています。

```
-- Code_66
create type Result_t is object(
  PK number,
  n1 number,
  ...
  v1 varchar2(30),
  ...)
/
create type Results_t is table of Result_t
/
```

これを理解することによって、*Code_67*に*Code_65*とは大幅に異なる目的を表現できます。

```
-- Code_67
  Results Results_t;
begin
  ...
  merge into  t Dest
  using      (select * from table(Results)) Source
  on          (Dest.PK = Source.PK)

  when matched then update set
    Dest.n1 = Source.n1,
    ...,
    Dest.v1 = Source.v1,
    ...

  when not matched then insert values (
    Source.PK,
    Source.n1,
    ...,
    Source.v1,
    ...);
```

*Code_65*は、再バインドして何回も再実行するSQL文を要求します。*Code_67*は、SQL文に対してバインドと実行を 1 回だけ要求します。当然、このコードは高速になります。また、*オブジェクト*の表を移入するプログラミングには、*レコード*の表を移入するプログラミングとは多少異なる記述が求められます。ただし、この違いは根本的ではなく表面的なものです。

この項の議論は、以下のベスト・プラクティスの原則にまとめられます。

単一の行の文を繰り返し記述するのではなく forall 文を使用してください。失敗した繰返しをスキップしても安全な場合に ORA-24381 を処理します。バルク・マージでは、オブジェクトのコレクションとともに table 演算子を使用します。

Principle_20

特定の *insert*、*update*、または *delete* 文の繰返しには、同等の単一の行によるアプローチではなく、常に *forall* 文を使用して実行するようにしてください。*save exceptions* キーワードを使用して、特定の繰返しに失敗したあとに安全に続行できる場合に *ORA-24381* のハンドラを提供します。バルク・マージでは、*オブジェクト*のコレクションとともに *table* 演算子を使用して、マージされる行を表します。

insert、*update*、*delete*、および *merge* のネイティブ動的 SQL の使用

埋込みSQLを使用する作業コードを、ネイティブ動的SQLを使用するように変換することは、ほかと比べて機械的な作業です。変換には常に*execute immediate*文を使用します。埋込みSQLテキストがPL/SQL変数ではなくプレースホルダをもつ同

等のテキストを格納する文字列変数[107]に置き換えられます。これは*execute immediate*とともに使用されます。*using*句でバインディングが実現します。*out*バインドを使用してSQL文の*returning*句の出力が取得されます（*into*句は一切使用しません）。*set row*構文を使用する埋込みSQLは、ネイティブ動的SQLでは表現できない点にとくに注意してください。各*レコード・フィールド*は、むしろ明示的に記述する必要があります。

107. 多くの場合、*execute immediate* のオペランドは *varchar2* です。ただし、Oracle Database 11g では、*clob* を使用できるようになりました。これは、*varchar2* の 32K の制限を超えるソース・テキストをもつ PL/SQL ユニットのプログラム生成に有用です。以前のリリースでこの制限を超えた場合は、コードを書き換えて *DBMS_Sql* API を使用していました。

ユースケースの例

この項では、一般的に発生するシナリオを確認して、要件を実装する最適なアプローチを説明します。

問合せ結果に応じた表データの変更

同じ場所で、または新しい表へ、表データを一括変換する必要がある場合、さまざまなシナリオが発生します。場合によっては、ソース表のデータを複数の宛先表に分散させる必要があります[108]。*update*文または*insert into... select ... from...*文のSQL式だけを使用する場合、必要なルールを表現できないことがあります。*Code_68*は、このような処理における*バッチ・バルク・フェッチ*と*forall*文を併用する方法を示しています。

```
-- Code_68
  cursor Cur is
    select Rowid, a.v1 from t a for update;

  type Rowids_t is varray(1000) of Rowid;
  Rowids Rowids_t;

  type vs_t is varray(1000) of t.v1%type;
  vs vs_t;

  Batchsize constant pls_integer := 1000;
begin
  ...
  loop
    fetch Cur bulk collect into Rowids, vs limit Batchsize;
    for j in 1..Rowids.Count() loop
     -- This is a trivial example.
     vs(j) := f(vs(j));
    end loop;
    forall j in 1..Rowids.Count()
     update  t a
     set    a.v1 = b.vs(j)
     where   Rowid = b.Rowids(j);
  exit when Rowids.Count() < Batchsize;
end loop;
```

変換が少なくPL/SQLファンクションを使用して表現できると、簡単なアプローチが可能になります。*Code_69*は、*Code_68*と同じ機能を提供します。

```
-- Code_69
...
update t set v1 = f(v1);
```

*Code_69*では、*update*文を発行するためにPL/SQLサブプログラムを使用する必要はありませんが、サブプログラムが表へのダイレクトSQLアクセスを隠すAPIの一部である場合には役立つことがあります。ただし、*Code_68*はこの手法の例に過ぎません。*Code_69*のように、表現できない変換もあります。

---

108. このシナリオは、とくに特定のバージョンから次のバージョンへのアプリケーションのアップグレードに関連して発生します。ベンダーは機能の追加やパフォーマンスの向上を図るために、アプリケーションの表の設計を変更する場合があります。たとえば、アップグレードで非正規化を維持する列が追加される場合です。アップグレード・スクリプトは、このような列を一括して移入する必要があります。

*Code_68*および*Code_69*のパフォーマンスを比較すると有用です。*Code_68*のアプローチを使用することで、パフォーマンスが低下するのでしょうか。そうではないと思われます[109]。

最後に、単純な単一の行によるアプローチがどのように記述され、どのように作用するかについて検討します。*Code_70*にこれを示します。

```
-- Code_70
for r in (select Rowid, a.v1 from t a for update) loop
  r.v1 := f(r.v1);
  update t a set a.v1 = r.v1
  where Rowid = r.Rowid;
end loop;
```

明らかに、*Code_68*よりも*Code_70*のほうが簡潔です。ただし、実行速度は大幅に低速になります[110]。

*Code_70*のバリエーションとして*Cur明示カーソル*および*where current of Cur*構造を使用する場合があります。*Code_71*にこれを示します。

```
-- Code_71
  cursor Cur is
    select a.v1 from t a for update;
  v1 t.v1%type;
begin
  open Cur;
  loop
    fetch Cur into v1;
    exit when Cur%NotFound;
    v1 := f(v1);
    update  t a
    set  a. v1 = v1
    where   current of Cur;
  end loop;
  close Cur;
```

*Code_72*は、PL/SQLコンパイラが*Code_71*のソース・コードから生成するSQL文[111]を示しています。

```
-- Code_72
SELECT A.V1 FROM T A FOR UPDATE
UPDATE T A SET A.V1 = V1 WHERE ROWID = :B1
```

これは、*where current of Cur*構造が、*Rowid*を明示的に選択することで達成される*Code_70*の糖衣構文にすぎないことを示しています。バルク構造に相当するものはありませんが、欠点は少しもありません[112]。

ベスト・プラクティスの原則は、以下のとおりです。

---

109. 固有のデータで独自のテストを実行できます。

110. 2,000,000 行を含む表を使用したテストで、*Code_68*と*Code_69*は、*DBMS_Utility.Get_CPU_Time()*を使用して測定したCPU時間が数パーセントの測定精度内で同じでした。ただし、*Code_70*は、*1.5x.*倍低速になりました。

111. 次のような問合せを使用することで、*Code_72*は生成されます。

```
select  Sql_Text
from    v$Sql
where   Lower(Sql_Text) not like '%v$sql%'
and     (Lower(Sql_Text) like 'select%a.v1%from%t%' or
         Lower(Sql_Text) like 'update%t%a%set%')
```

読みやすくなるように、Code_5が手動で整えられています。

バッチ・バルク・フェッチを使用した行の取得、PL/SQL による行の処理、および forall 文を使用した各バッチの送信の実行を躊躇する必要はありません。SQL 文で直接 PL/SQL ファンクションを使用する場合と比較しても、このアプローチを使用することでパフォーマンスが大幅に低下することはないためです。

Principle_21

PL/SQL でのみ表現できるアプローチを使用して多くの行を変換する必要がある場合、*バッチ・バルク・フェッチ*で取得して処理し、*forall* 文で各バッチの結果を使用して、*Rowid* でソース行を*更新*するか異なる表に*挿入*します。必要に応じて、このアプローチは *merge* と組み合わせることができます。適切な SQL 文で直接 PL/SQL ファンクションを使用する場合と比較しても、このアプローチを使用することでパフォーマンスが大幅に低下することはありません。

実行時までわからない *in list* 項目の数

可能性は低いものの、*where*句で 5 つの項目とともに*in list*を使用する問合せが必要な場合、要件をあとで変更して現実的なユースケースを反映する課題に左右されず埋込みSQL文を記述できます。*Code_73*は、この仮定の文を示しています。

```
-- Code_73
select    a.PK,          a.v1
bulk collect into        b.Results
from                     t a where
                         a.v1 in (b.p1, b.p2, b.p3, b.p4, b.p5);
```

*Code_74*は、より現実的な文を表しています。

```
-- Code_74
select                   a.PK, a.v1
bulk collect into        b.Results
from                     t a
where                    a.v1 in (b.ps(1), b.ps(2), b.ps(3),
                                  b.ps(4), b.ps(5), b.ps(6),
                                  b.ps(7), b.ps(8), b.ps(9),
                                  ...                      );
```

問題は、索引値のリテラルを使用してコレクションの各要素に明示的な参照を記述するのが困難なことです。記述できたとしてもテキストが膨大になります[113]。むしろ、"数が多い場合でもこのコレクションのすべての要素"を表す構文が必要です。このような構文は、埋込みSQLでサポートされています。*Code_75*にこれを示します。

```
-- Code_75
  ps Strings_t;
begin
  select                 a.PK, a.v1
  bulk collect into      b.Results
  from                   t a
  where                  a.v1 in (select  Column_Value
                                  from   table(b.ps));
```

112. 一般的に、*Rowid* の不変性に依存することは危険です。たとえば、*alter table... shrink* コマンドに従う場合、特定の主キーを使用した行の *Rowid* は、行の移動の結果として変更されることがあります。ただし、別のセッションの *alter table... shrink* は現在のトランザクションが完了するまで待機するため、*select... for update* の発行とこのトランザクションを完了する *commit* または *rollback* の間にリスクはありません。

113. このような明示的に索引付けされた 1,000 個の要素をハードコードできます。そのあとの実行時に、任意の値に応じて多くの要素を設定し、残りの値を *null* にできます。*null* の等価テストは失敗した等価テストと同じ効果をもつため、これは正しいセマンティックです。

ただし、この内容はほとんどわかりません。これは、おそらく*table*演算子[114]を使用しているためです。*ps*のデータ型は、スキーマ・レベルで宣言する必要があります。*Code_76* は、これを作成するSQL*Plusスクリプトを示しています。

```
-- Code_76
create type Strings_t is table of varchar2(30)
/
```

*table*演算子を知らないプログラマーが実行時にSQL文のテキストを構築して機能要件を満たすことは珍しくありません。この場合、リテラル値を使用してネイティブ動的SQLで文を実行するか、プレースホルダを使用して*DBMS_Sql* APIで文を強制的に実行し、実行時までバインディング要件がわからない状況に対応します。こうした回避策は、典型的なワースト・プラクティスといえます[115]（また、対象の問合せを結合として表現できるように、最初に*in list*値をグローバル一時表に挿入するプログラマーもいるようです）。ベスト・プラクティスの原則は、以下のとおりです。

要素の数がわからない in list の機能には、"where x in (select Column_Value from table(The_Values))"を使用します。

Principle_22

要素の数が実行時までわからない *in list* の機能が必要な場合、データ型をスキーマ・レベルで定義する必要があるコレクションに値を移入して、*"where x in (select Column_Value from table(The_Values))"*を使用します。別のアプローチは検討しないでください。

114. *table* 演算子の使用方法は、『Oracle Database オブジェクト・リレーショナル開発者ガイド』の"*SQL による個々のコレクション要素の操作*"の項で説明されています。『Oracle Database PL/SQL 言語リファレンス』でも、説明はありませんが"*PL/SQL の言語要素*"の項で言及しています。

115. ネイティブ動的 SQL のアプローチは個別の文のテキストの拡散につながるので、余分なハード・パースが発生します（きちんと考慮されていない場合は、SQL インジェクションのリスクも発生します）。*DBMS_Sql* API を使用するとハード・パースの回数が削減されますが、1 回で十分な場合に *N* 回のハード・パースを実行すると、*N-1* 回余分になります。

結論

本書の5 ページにある"*埋込みSQL、ネイティブ動的SQL、およびDBMS_Sql API*"の項では、PL/SQLからSQLを実行するさまざまな構造を簡単に説明しました。また、新しい専門用語を紹介することで、ユーザーが混乱する可能性の高い概念[116]を明確にしました。

ほかにも32 ページの"*select文のアプローチ*"の項と50 ページの"*insert、update、delete、およびmerge文のアプローチ*"の項では、異なる観点で説明しました。これは、一貫性のあるメンタル・モデルで保持できるほど小規模で、ほとんどの実用的な目的に対して十分な機能をもつ、利用可能な機能のサブセットを確認し、推奨するためです。これには妥協が必要になります。PL/SQLによりSQLを実行する方法（および機能が導入された経緯）を完全に把握しており、実際のアプリケーションですべての技術を活用した経験をもったユーザーは、新しいプロジェクトで特定の課題に対し、最適な手法を常に選択できます。ただし、そのようなユーザーは、本書の対象読者ではありません。本書は、簡単に理解および保守できるように、一貫性のあるアプローチを使用し、容認できるパフォーマンスをもったコードを記述する必要がある読者を対象としています。正確性も求められます。使用されるプログラミング技術が比較的少ない場合や統一された方法でこれらの技術要件を満たすことができる場合は、ほとんど同じ目的をもった一連のプログラムよりも正確性が重要となります。

本書が対象読者の手助けとなれば幸いです。

Bryn Llewellyn
オラクル、PL/SQL Product Manager
bryn.llewellyn@oracle.com

2008 年 9 月 21 日

116. この混乱の理由は、PL/SQL の長い歴史を通じてオラクルが安易に使用してきた専門用語によるものです。

付録A：

変更履歴

2008 年 9 月 21 日

- 初版

2008 年 9 月 21 日

- *authid* の選択に関するベスト・プラクティスの原則の改訂。同僚からのフィードバックによるマイナー・エラーの修正

付録B：ベスト・プラクティスの原則のまとめ

この付録には、本書で推奨しているベスト・プラクティスの原則を記載しています。あるプログラマーの格言に、"ルールは服従するためのものではなく、役立てるものである"という言葉がありますが、[117] 第一（メタ）原則としてさらに適切な提案を以下に示します。

次のベスト・プラクティスの原則に違反する前に、経験のある同僚から承認を得てください。

✓ Principle_0：経験のある PL/SQL プログラマーと最初に議論することなく、以下の原則から逸脱するようなユースケースを採用しないでください。

埋込み SQL では、from list 項目の別名で各列の名前をドット修飾します。宣言するブロックの名前で各 PL/SQL 識別子をドット修飾します。

✓ *Principle_1：*埋込みSQL文を記述する場合、常に各*from list*項目の別名を確認し、適切な別名で各列を修飾します。現在のPL/SQLユニットで解決する各識別子の名前は、宣言されているブロックの名前で常に修飾します。これがブロック文の場合、ラベルによる名前が使用されます。別名およびPL/SQLブロックの名前はすべて一意である必要があります。これによって、参照される表が変更される場合に名前取得を回避するため、ファイングレインな依存性の分析でPL/SQLユニットを無効化する必要がないという結論に達する可能性が高くなります。 （9 ページ）

ブロックが変更しない限り、constant キーワードで各 PL/SQL 変数を宣言します。

✓ *Principle_2：*初期化のあとに変更されない変数の宣言でconstantキーワードを使用します 21。最悪のケースでもconstantの変更を試みるコードがコンパイルに失敗するため（このエラーによってプログラマーの考え方が鋭くなります）、この原則に従ってもペナルティはありません。この原則は、可読性と正確性という点で明白な利点が得られます。（11 ページ）

常に authid プロパティを明示的に指定します。Current_User または Definer を慎重に選択します。

✓ *Principle_3：*各PL/SQLユニットのauthidプロパティを常に明示的に指定します。ユニットの目的を慎重に分析したあと、定義者権限または実行者権限を選択します。 （12 ページ）

117. このテーマのバリエーションで、"*この場合も含めて、すべてのルールは破られるものである*"という矛盾も存在します。

Owner を使用して、Oracle Database に付属しているオブジェクトの名前をドット修飾します。

✓ *Principle_4：*Oracle Databaseに付属するオブジェクトの参照はOwnerでドット修飾されます（Sysになる場合が多いですが、かならずそうなるわけではありません）。これによって、目的のオブジェクトと名前が競合するローカル・オブジェクトが、名前の解決時に現行のスキーマに作成される場合でも、意図したとおりに保存されます。（13 ページ）

コンパイル時にテキストが固定される SQL 文を使用してください。使用できない場合は、固定したテンプレートを使用してください。プレースホルダにバインドします。DBMS_Assert を使用して、連結した SQL 識別子を保護します。

✓ *Principle_5：*コンパイル時にテキストが固定されるSQL文のみを常に使用してください。select、insert、update、delete、merge、およびlock table文には、埋込みSQLを使用します。ほかの文には、ネイティブ動的SQLを使用します。コンパイル時にSQL文を固定できない場合、固定した構文テンプレートを使用し、名前のプロビジョニングへの実行時のバリエーションを制限してください（これは、プレースホルダの使用とバインディングの負担の軽減を意味します）。スキーマ・オブジェクトの名前と列名などのオブジェクト内の識別子には、Sys.DBMS_Assert.Simple_Sql_Name()を使用します。例外的に、プレースホルダではなくリテラル値の使用が条件の場合は、Sys.DBMS_Assert.Enquote_Literal()を使用します。ほかの値（Code_8のNLS_Date_Formatの値など）の場合は、パラメータ化されたユーザー入力に応じてプログラムで構築します。（15 ページ）

動的 SQL には、ネイティブ動的 SQL を使用します。使用できない場合のみ、DBMS_Sql API を使用します。

✓ *Principle_6：*動的SQLには、機能が不十分な場合を除いて常にネイティブ動的SQLを使用します。機能が不十分な場合にのみ、*DBMS_Sql* APIを使用します。*select*、*insert*、*update*、*delete*、および*merge*文の場合、コンパイル時にわからないプレースホルダまたは*select list*項目がSQL文に含まれると、ネイティブ動的SQLでは対応できなくなります。ほかのSQL文の場合、操作がリモート・データベースで実行されるとネイティブ動的SQLでは対応できません。（17 ページ）

動的 SQL を使用する場合、SQL 文のリテラルを回避してください。代わりに、適切な値をプレースホルダにバインドしてください。

✓ *Principle_7：*動的に作成したSQL文で連結されたリテラルを使用しないでください。リテラルではなくプレースホルダを使用してください。実行時にリテラルだった値をバインドします。これによって、*共有可能なSQL構造*を最大限に再利用できます。（19 ページ）

常に DBMS_Sql.Parse (Security_Level=>2)で DBMS_Sql 数値カーソルを開きます。

✓ *Principle_8：Security_Level*仮パラメータを使用する*DBMS_Sql.Parse()*のオーバーロードを常に使用してください。また、*DBMS_Sql数値カーソル*のすべての操作が同じ*Current_User*および有効なロールで実行される実効値 *2* で常に呼び出してください。（25 ページ）

使用する必要がある唯一の明示カーソル属性は、Cur%IsOpen です。必要な暗黙カーソル属性は、Sql%RowCount、Sql%Bulk_RowCount、および Sql%Bulk_Exceptions のみです。

✓ *Principle_9 :* 本書が推奨するアプローチに従う場合、唯一の便利な*明示カーソル属性*は*Cur%IsOpen*です。ほかの*明示カーソル属性*を使用する必要はありません。唯一影響のあるスカラー型の*暗黙カーソル属性*は、*Sql%RowCount*です。*暗黙カーソル*を使用して対象のSQL文を実行する文に準拠するPL/SQL文で、これを常に確認してください。同じ論理が*Sql%Bulk_RowCount*コレクションに当てはまります。*Bulk_Errors*例外の例外ハンドラでのみ、*Sql%Bulk_Exceptions*を使用する必要があります。実行可能なセクションの唯一の文として、*forall*文を含む*ブロック文*にこれを配置してください。 （27 ページ）

専門用語のセッション・カーソル、暗黙カーソル、明示カーソル、カーソル変数、および DBMS_Sql 数値カーソルを学習します。省略せず、慎重に使用してください。

✓ *Principle_10 :* PL/SQLプログラムを説明する場合、PL/SQLプログラム自体の説明、そのコメント、および外部ドキュメントの記述が、修飾されていない"カーソル"の使用を回避する目的で含まれています。*セッション・カーソル*、*暗黙カーソル*、*明示カーソル*、*カーソル変数*、または*DBMS_Sql数値カーソルといった適切な専門用語*を使用してください。この原則により思考力が高まってプログラムの品質が向上します。 （29 ページ）

問合せで取得される行数がわからない場合、無限ループ内で limit 句とともに fetch... bulk collect into を使用してください。

✓ *Principle_11 :* 多くの行が選択されて結果セットが大きい場合は、*無限ループ*内で*limit*句とともに*fetch... bulk collect into*を使用してバッチで処理します。*limit*句の値として*constant*を使用し、バッチサイズを定義します。適切な値は 1000 です。同じサイズで宣言された*varray*にフェッチします。ループを終了するために*%NotFound*カーソル属性をテストしないでください。代わりに、ループの最後の文として*exit when Results.Count() < Batchsize;*を使用します。これによって、最後のフェッチがゼロ行になる処理が正しく実行されます。埋込みSQLで十分な場合は、*明示カーソル*を使用します。ネイティブ動的SQLが必要な場合は、*カーソル変数*を使用します。 （33 ページ）

問合せでどのように行の最大数が取得されるがかわかる場合は、select... bulk collect into または execute immediate... bulk collect into を使用して、単一の手順ですべての行をフェッチします。

✓ *Principle_12 :* 多くの行が選択されており、結果セットの管理可能な最大サイズが安全に設定された場合、単一の手順ですべての行をフェッチします。埋込みSQLを使用できる場合、カーソルのないPL/SQL構造*select... bulk collect into*を使用します。動的SQLが必要な場合、*execute immediate... bulk collect into*を使用します。処理できる最大サイズで宣言された*varray*にフェッチします。*ORA-22165* の例外ハンドラを実装して、バグ診断をサポートしてください。 （35 ページ）

実行時まで select list のバインディング要件がわからない場合は、DBMS_Sql API を使用します。少なくとも select list が識別されている場合、To_Refcursor() を使用し、次にバッチ・バルク・フェッチを使用します。

✓ *Principle_13 :* リテラルで*where*句を構成しないでください。とくに、ユーザーが明示的に入力した*where*句を連結しないでください。プレースホルダをバインドするアプローチよりもパフォーマンスが大幅に低下します。直接*where*句を入力した場合、*Sys.DBMS_Assert.Enquote_Literal()*でSQLインジェクションから保護することはできません。実行時までバインディング要件がわからない場合、*DBMS_Sql* APIを使用して、SQL文をパース、バインド、および実行します。コンパイル時に*select list*が識別される場合、*To_Refcursor()*を使用して*DBMS_Sql数値カーソル*を*カーソル変数*に変換し、*バッチ・バルク・フェッチ*を使用します。実行時まで*select list*がわからない場合、*DBMS_Sql* APIを使用して結果もフェッチします。コンパイル時にバインディング要件が識別され、実行時まで*select list*が識別されない珍しい事例の場合、ネイティブ動的SQLを使用して*カーソル変数*を開き、*To_Cursor_Number()*を使用して*カーソル変数*を*DBMS_Sql数値カーソル*に変換します。次に、*DBMS_Sql* APIを使用して、結果をフェッチします。（39 ページ）

単一の行だけを取得するには、select... into または execute immediate... into を使用します。No_Data_Found および Too_Many_Rows を活用してください。

✓ *Principle_14 :* 単一の行だけを選択するときは、単一の手順で行をフェッチします。埋込みSQLが使用できれば、カーソルのないPL/SQL構造*select... into*を使用します。動的SQLが必要な場合は、*execute immediate... into*を使用します。望ましくない*No_Data_Found*例外および予期しない*Too_Many_Rows*例外を活用してください。（41 ページ）

API を定義するオブジェクトのプライベート・シノニムだけを含む専用スキーマを使用して、データベース・アプリケーションを公開します。

✓ *Principle_15 :* アプリケーションの機能にアクセスするために、データベース・クライアントが接続するアプリケーションの専用スキーマ（Some_App_APIなど）を作成します。厳密に保護されたパスワードを使用して、Some_App_API以外のスキーマにすべてのアプリケーションのデータベース・オブジェクトを実装します。また、Some_App_APIのオブジェクトをプライベート・シノニムだけに制限します。クライアントAPIを公開するアプリケーションのデータベース・オブジェクトだけにこのようなシノニムを作成します。これらのオブジェクトのアクセスに必要な権限だけをSome_App_APIに付与します。（41 ページ）

PL/SQL API でデータベース・アプリケーションを公開します。データベースのクライアントがアクセスできないスキーマのすべての表を非表示にします。

✓ *Principle_16 :* Some_App_APIのプライベート・シノニムがPL/SQLユニットだけに公開されるオブジェクト型を制限します。ほかのスキーマのすべての表を非表示にします。これらの権限をSome_App_APIに付与しないでください。（42 ページ）

return データ型が任意のデータを表すファンクションとして、プロデューサ/コンシューマ API を定義します。プロデューサ・モジュールのすべての SQL 処理を非表示にします。これによって、コンシューマは、要件の変更が原因の実装の変更による影響を受けなくなります。問合せのパラメータ化のようにプロデューサ・ファンクションをパラメータ化してください。このアプローチは、バッチでの行の取得や単一の呼出しでのすべての行の取得に対応します。これがスライスになる場合もあります。

✓ *Principle_17 :* returnデータ型が作成されたデータを表す機能として、プロデューサ/コンシューマAPIを定義します。プロデューサ・モジュールで、SQL処理に関連するすべての処理（フェッチを含む）を非表示にします。問合せのパラメータ化で単一の行のみを指定する場合、select listと同じ構造でレコードまたはADTを使用します。この場合、動的SQLの要件に応じてカーソルのないPL/SQL構造のselect... intoまたはexecute immediate... intoを使用します。問合せのパラメータ化で複数の行を指定する場合、レコードのコレクションまたはADTのコレクションを使用します。ここで安全であることが確認できれば、全体バルク・フェッチを使用します。これによって、カーソルのないPL/SQL構造のselect... bulk collect intoまたはexecute immediate... bulk collect intoを使用できます。全体バルク・フェッチが安全ではない場合、プロデューサ/コンシューマの関係がステートフルなときにバッチ・バルク・フェッチを使用します。これには識別カーソルが必要です。埋込みSQLで十分な場合、プロデューサ・パッケージの本体のグローバル・レベルで宣言された明示カーソルを使用します。これによって、各バッチを取得するコンシューマからの呼出しの状態が保持されます。動的SQLが必要な場合、カーソル変数を使用します。コンシューマが保持できるように、各バッチの結果とともにこれをコンシューマに戻します。プロデューサ/コンシューマの関係がステートレスな場合、結果セットのスライスを使用します。全体バルク・フェッチで各スライスの配信を実装します。要件に示されている場合、DBMS_Sql APIを使用して、プロデューサ・モジュールでこれを使用するすべてのコードを非表示にします。 （45 ページ）

各アプリケーション表で、同じ制約およびデフォルト値を定義するテンプレートのレコード型を保存します。

✓ *Principle_18 :* 各アプリケーション表のレコード型の宣言を公開するパッケージを保存します。宣言では、表の特徴を示す列名、データ型、制約、およびデフォルト値の指定を繰り返す必要があります。 （48 ページ）

"upsert"要件には、merge を使用します。例外ハンドラの insert と一緒に update... set row...を使用しないでください。

✓ *Principle_19 :* "upsert"が必要な場合は、mergeを使用してください。update... set row...を実装して、対応するinsertを実装したDup_Val_On_Indexの例外ハンドラを用意するよりも有効です。 （52 ページ）

単一の行の文を繰り返し記述するのではなく forall 文を使用してください。失敗した繰返しをスキップしても安全な場合に ORA-24381 を処理します。バルク・マージでは、オブジェクトのコレクションとともに table 演算子を使用します。

✓ *Principle_20 :* 特定のinsert、update、またはdelete文の繰返しには、同等の単一の行によるアプローチではなく、常にforall文を使用して実行するようにしてください。save exceptionsキーワードを使用して、特定の繰返しに失敗したあとに安全に続行できる場合にORA-24381 のハンドラを提供します。バルク・マージでは、オブジェクトのコレクションとともにtable演算子を使用して、マージされる行を表します。 （57 ページ）

バッチ・バルク・フェッチを使用した行の取得、PL/SQL による行の処理、および forall 文を使用した各バッチの送信の実行を躊躇する必要はありません。SQL 文で直接 PL/SQL ファンクションを使用する場合と比較しても、このアプローチを使用することでパフォーマンスが大幅に低下することはないためです。

✓ *Principle_21 :* PL/SQLでのみ表現できるアプローチを使用して多くの行を変換する必要がある場合、バッチ・バルク・フェッチで取得して処理し、forall文で各バッチの結果を使用して、Rowidでソース行を更新するか異なる表に挿入します。必要に応じて、このアプローチはmergeと組み合わせることができます。適切なSQL文で直接PL/SQLファンクションを使用する場合と比較しても、このアプローチを使用することでパフォーマンスが大幅に低下することはありません。（60 ページ）

要素の数がわからない in list の機能には、"where x in (select Column_Value from table(The_Values))"を使用します。

✓ *Principle_22 :* 要素の数が実行時までわからないin listの機能が必要な場合、データ型をスキーマ・レベルで定義する必要があるコレクションに値を移入して、"where x in (select Column_Value from table(The_Values))"を使用します。別のアプローチは検討しないでください。（62 ページ）

付録C：

*レコード*のコレクションに *select* 文の結果を移入する アプローチの代案

*Code_77*は、もっともわかりやすいアプローチを示しています。結果が *レコード*のコレクションにバルク・フェッチされ、値が同じ構造の*ADT*のコレクションに明示的なループによりコピーされます。

```
-- Code_77
select PK, n1, n2, v1, v2
bulk collect into Records
from t
order by PK;

Objects.Extend(Records.Count());
for j in 1..Records.Count() loop
  Objects(j) := Object_t(
    Records(j).PK,
    Records(j).n1,
    Records(j).n2,
    Records(j).v1,
    Records(j).v2);
end loop;
```

*Code_78*は、*ADT*のコレクションに直接バルク・フェッチできる*select list*要素として必要な*ADT*を構築し、*レコード*の一時的なコレクションを使用しないコンパクトなアプローチを示しています。ただし、このアプローチは、*Code_77*よりも大幅に低速になります。

```
-- Code_78
select Object_t(PK, n1, n2, v1, v2)
bulk collect into Objects
from t
order by PK;
```

Code_79 は、直接フェッチできる単一の行の *select list* 要素として SQL で直接必要な *ADT* のコレクションを構築するコンパクトなアプローチを示しています。このアプローチは、*Code_77* とほぼ同じ速度です。

```
-- Code_79
select cast(multiset(
    select Object_t(PK, n1, n2, v1, v2)
    from t
    order by PK)
  as Objects_t)
  into Objects
  from Dual;
```

もっとも速い 2 つのアプローチも、*レコード*のコレクションにフェッチしてその結果を処理する場合に比べると大幅に低速になります。ただし、Oracle Database 11g を使用したこのアプローチでは、PL/SQL API のみを通じてデータベースを公開するため、任意のクライアントが使用できるようになります。

付録D：

テスト・ユーザーUsr およびテスト表 Usr.t(PK number, v1 varchar2(30), ...)の作成

*Code_80*は、固有のテスト・データベースの*非定型*テストを開始する便利なSQL*Plusスクリプトを示しています。

```
-- Code_80
CONNECT Sys/p@111 AS SYSDBA
declare
  User_Does_Not_Exist exception;
  pragma Exception_Init(User_Does_Not_Exist, -01918);
begin
  begin
 execute immediate 'drop user Usr cascade';
  exception when User_Does_Not_Exist then null; end;
  execute immediate '
     grant Create Session, Resource to Usr identified by p';
end;
/
alter session set Current_Schema = Usr
/
```

*Sys*で接続すると、*DBA_Objects*、*v$Sql*、*v$Parameter*などのビューおよび*alter system*などのコマンドへの*非定型*の問合せを正しく使用できます。一時的なデータベースでのみこれを実行する必要があります[118]。ただし、この手段を使用する場合は注意してください。SQL*Plusスクリプトにより提供されるテストの目的によっては、すべてのオブジェクトにアクセスできるので結果がわかりづらくなる場合があります。これは、実行者権限のPL/SQLユニットの動作を調査または説明するテストで顕著です。

118. このために、新しく作成したデータベースのコールド・バックアップを作成し、スクリプトを記述してデータベースを停止してからリストアして、データベースを起動しています。開発者向けのIntel/Linux マシンで実行すると、数分の処理で済みます。仕組みをテストまたは確認するために潜在的にリスクを伴うテストを検討する場合、正常なテスト環境をリストアする迅速で信頼性の高い機能は非常に有効です。

*Code_81*は、パラメータ化された行数を使用した便利なテスト表*t*を作成するプロシージャを示しています。パフォーマンス・テストにおいて、小規模なデータセットでコードを迅速にテストしてから大規模なデータセットでテストできるため非常に便利です。

```
-- Code_81
procedure Usr.Create_Table_T(No_Of_Batches in pls_integer)
  authid Current_User
is
  Batchsize constant pls_integer := 1000;
  No_Of_Rows constant pls_integer := Batchsize*No_Of_Batches;
  n integer := 0;

  type PKs_t is table of number      index by pls_integer; PKs Pks_t;
  type n1s_t is table of number      index by pls_integer; n1s n1s_t;
  type n2s_t is table of number      index by pls_integer; n2s n2s_t;
  type v1s_t is table of varchar2(30) index by pls_integer; v1s v1s_t;
  type v2s_t is table of varchar2(30) index by pls_integer; v2s v2s_t;
begin
  declare
     Table_Does_Not_Exist exception;
     pragma Exception_Init(Table_Does_Not_Exist, -00942);
  begin
     execute immediate 'drop table Usr.t';
  exception when Table_Does_Not_Exist then null; end;

 execute immediate '
  create table Usr.t(
   PK number                          not null,
   n1 number        default 11        not null,
   n2 number        default 12        not null,
   v1 varchar2(30)  default ''v1''    not null,
   v2 varchar2(30)  default ''v2''    not null)';

  execute immediate '
   create or replace package Usr.Tmplt is
    type T_Rowtype is record(
     PK number              not null := 0,
     n1 number              not null := 11,
     n2 number              not null := 12,
     v1 varchar2(30)        not null := ''v1'',
     v2 varchar2(30)        not null := ''v2'');
   end Tmplt;';

 for j in 1..No_Of_Batches loop
  for j in 1..Batchsize loop
   n := n + 1;
   PKs(j) := n;
   n1s(j) := n*n;
   n2s(j) := n1s(j)*n;
   v1s(j) := n1s(j);
   v2s(j) := n2s(j);
  end loop;
  forall j in 1..Batchsize
   execute immediate '
     insert into           Usr.t(PK, n1, n2, v1, v2)
     values                (:PK,  :n1, :n2, :v1, :v2)'
     using                 PKs(j), n1s(j), n2s(j), v1s(j), v2s(j);
  end loop;

  execute immediate
     'alter table Usr.t add constraint t_PK primary key(PK)';
end Create_Table_T;
```

このプロシージャは、表*t*と同じ構造、および同じ制約とデフォルト値を実行して、*T_Rowtype レコード*型を公開するパッケージも作成します。これは、一部の値だけが指定される場合に、埋込みSQLで新しい行を*t*に挿入する事例などで役立ちます（*t%rowtype*で宣言された変数は、表から制約およびデフォルト値を取得しません）。これは、47 ページの*"単一の行のinsert"*のコードで活用されています。

"*PK number*"などの繰り返される文字列や *varchar2 定数*のデフォルト値を宣言すれば、*Create_Table_T()*プロシージャが改善されると主張する人もいますが、読みやすさを考慮して、本書では現状のままにしています。また、そのほかのアプローチも存在します。たとえば、PL/SQL サブプログラムは、*All_Tab_Cols* カタログ・ビューから *Column_Name*、*Data_Type*、*Data_Default* などの列にアクセスして、表のリストごとに*レコード・テンプレート・パッケージ*を作成できます。すべてを同時に実行することは、アプリケーションのインストールやパッチ/アップグレード・スクリプト上の問題になります。

このプロシージャでは、文のテキストがコンパイル時に固定されても、*insert* 文の動的 SQL を使用する必要があります。*insert* のターゲット表がコンパイル時に存在しないためです。

PL/SQL による SQL の実行：ベスト・プラクティスとワースト・プラクティス
2008 年 9 月
オラクル PL/SQL Product Manager、Bryn Llewellyn