いまさら聞けない
Oracle の基本 中級編

「いまさら聞けない Oracle の基本」が好評につき、続編が登場しました！

前回は Oracle Database のインストール方法から SQL、トランザクションやロック等の基本について解説しましたが、本ドキュメントでは中級編として、ビューやインデックスなどの「データベースオブジェクト」の使い方について解説します。（なお、前回の資料は「いまさら聞けない Oracle の基本（初級編）」という名前に変更しました。）

データベースの勉強には、実際のデータベースを動かしながら学ぶのが一番の近道です。ぜひ、Oracle をインストールの上、本ドキュメントをお読みいただければと思います。Oracle のインストール方法については前回の初級編をご覧ください。

1 データベースオブジェクトを活用する

1-1. データベースオブジェクトとは？

データベースオブジェクトとは、データベースが提供する機能のことです。初級編ではテーブルの作成方法、データの管理方法についてご説明しましたが、テーブルも代表的なデータベースオブジェクトのひとつです。しかし、実際のアプリケーション開発でデータベースを利用すると、テーブルだけでは不便な点が出てきます。例えば、「毎回テーブルを結合するのが面倒」、「大量にデータを格納するテーブルがあるのでパフォーマンスを改善したい」など、個々のシステムに応じていろいろな要望が出てきます。そこで、データベースでは、各開発者が機能を拡張できるような仕組みを提供しています。それが、「データベースオブジェクト」です。ユーザーは SQL 文を使用して自由にデータベースオブジェクトを作成し、機能機能することが可能です。代表的なデータベースオブジェクトは以下の通りです。

目的	データベースオブジェクトの種類
データを格納する	テーブル
データにアクセスしやすく（見やすく）する	ビュー
	シノニム
	データベースリンク
データの整合性を守る	制約
	シーケンス
	トリガー
パフォーマンス（処理速度）を 向上する	インデックス
	ストアドプロシージャ
	ストアドファンクション
	パッケージ

表１－１. 代表的なデータベースオブジェクト

基本的には、データベースオブジェクトを作成したい場合は、「CREATE データベースオブジェクトの種類 データベースオブジェクト名」という SQL で作成することが可能です。また、この SQL の「CREATE」の部分を「DROP」に変更することで、オブジェクトを削除することができます。

オブジェクトを作成するSQLは、DDL(Data Definition Language：データ定義言語)、テーブルのレコードを扱うDML（Data Manipulation Language：データ操作言語）と呼ばれ、区別されています。DMLとDDLの違いは「やり直しができるかどうか」です。DMLでは、ロールバックという命令によりデータの更新をキャンセルすることができますが、DDLはキャンセルができませんので注意が必要です。

次章からは、これら各データベースオブジェクトの概要および作成や利用するための SQL について解説します。

2 データを見やすくする「ビュー」

2-1. ビューとは

ビューはデータをみやすくする目的で作られたオブジェクトです。例として、以下のような２つのテーブルを結合したビューを作成してみましょう。

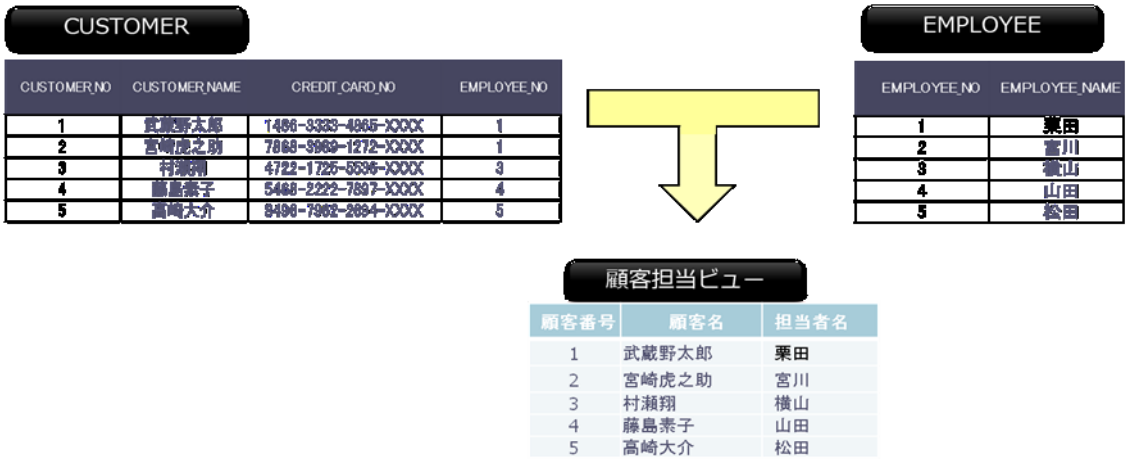

図２－１.ビューの作成イメージ

この例では、顧客情報を持つ「CUSOMER」テーブルと、従業員情報をもつ「EMPLOYEE」テーブルを結合して、顧客ごとの担当者を表示する「顧客担当ビュー」というビューを作成します。ビューの作成の前にはあらかじめ、テーブルの作成が必要です。以下の SQL を実行してください。

```
--CUSTOMER テーブルの作成
CREATE TABLE CUSTOMER
(
    CUSTOMER_NO                 NUMBER,
    CUSTOMER_NAME               VARCHAR2(50),
    CREDIT_CARD_NO              VARCHAR2(19),
    EMPLOYEE_NO                 NUMBER
)
/
--CUSTOMER テーブルのデータを作成
INSERT INTO CUSTOMER values('1','武蔵野太郎','1486-3333-4865-XXXX','1') ;
INSERT INTO CUSTOMER values('2','宮崎虎之助','7868-3989-1272-XXXX','1') ;
INSERT INTO CUSTOMER values('3','村瀬翔','4722-172-5536-XXXX','3') ;
INSERT INTO CUSTOMER values('4','藤島素子','5468-2222-7897-XXXX','4') ;
INSERT INTO CUSTOMER values('5','高崎大介','8496-7962-2694-XXXX','5');
```

```
--EMPLOYEE テーブルの作成
CREATE TABLE EMPLOYEE
(
    EMPLOYEE_NO                    NUMBER,
    EMPLOYEE_NAME                   VARCHAR2(50)
)
/
--EMPLOYEE テーブルのデータを作成
INSERT INTO EMPLOYEE values('1','栗田');
INSERT INTO EMPLOYEE values('2','宮川') ;
INSERT INTO EMPLOYEE values('3','横山') ;
INSERT INTO EMPLOYEE values('4','山田') ;
INSERT INTO EMPLOYEE values('5','松田') ;
```

それでは、これらのテーブルを結合したビューを作成しましょう。ビューを作成するには CREATE VIEW 文を使用します。以下の SQL 文を実行してください。

```
CREATE VIEW V_顧客担当  AS
SELECT    C.CUSTOMER_NO AS  顧客名,
          DECODE (C.AREA_CODE,1,'関東',2,'関西') AS  地域
          E.EMPLOYEE_NAME AS  担当者名
FROM CUTOMER C, EMPLOYEE E
WHERE C.EMPLOYEE_NO = E.EMPLOYEE_NO
/
```

「CREATE VIEW」の後につづけてビュー名を指定します。2 行目以降はビューが表示する列情報を SELECT 文を記述して定義します。今回の例では、SELECT 句に「CUSTOMER_NO」,「AREA_CODE」と、「EMPLOEE_NAME」を記述していますが、各列名の後に「AS」と記述し、続けて対応する日本語名を記述しています。このように AS の後につづけて任意の名前を記述すると、列名を変更してデータを表示することが可能です。この機能をエイリアスと呼ばれます。また、DECODE という関数を使用して、AREA_CODE のデータを変換して表示しています。DECODE 関数の後の括弧の中には、まず、対象の列名（今回の例では AREA_CODE）を記述し、続けて値と対応する文字列を必要な数だけ繰り返して指定します。今回の例では、1 であれば「関東」、2 であれば「関西」という文字列として表示するように指定しています。この SQL を実行すると、ビューが作成されます。以降は「SELECT ＊ FROM ビュー名」だけで、同様のデータが取得できます。

2-2. ビューを作成するメリット

ビューを作成することで、以下のメリットがあります。

・毎回、複雑な SQL 文を記述する必要がなくなる

テーブルの結合してデータを取得することは、ビューを使用しなくても、通常の SELECT 文を使用することで可能です。しかし、毎回このような SQL は複雑ですので、毎回記述するのは手間でしょう。1 度、このようなビューを作成しておけば、「SELECT ＊ FROM ビュー名」のみで同様のデータが取得することが可能となり、楽にデータの取得ができるようになります。

・データを自由に加工することができる

ビューはデータを持たないかわりに、自由にデータを見やすくした状態で表示することができます。開発の現場においては、データベースが登場した当初は、日本語などのマルチバイト文字がサポートされていなかった関係で、テーブル名やテーブルの列名は現在もアルファベット（1 バイト文字）で記述するのが慣習になっています。また、データの容量節約のため、「関東」は 1、「関西」は 2 という数値データで格納しておくことも多々あります。このような生データをそのままみても、意味はわかりづらいでしょう。このような場合にもビューは活躍します。今回の例では、AS 句を使用して列名をわかりやすい日本語に、また、DECODE 関数を利用して、AREA_CODE の数値データを「関東」「関西」に変換した上で表示しています。このように、ビューではテーブルの実データをわかりやすく加工し、表示することができます。

・セキュリティを向上することができる

ビューでは不要なデータを見せなくする目的にも利用できます。今回作成したビューは、各顧客毎の担当者を表示することが目的ですので、CUSTOMER テーブルにある「CREDIT_CARD_NO（クレジットカード番号）」のデータを表示する必要はありません。ビューでは、テーブルがもつ各列から必要な列のみを選択して表示が可能です。もし、従業員にはクレジットカード番号を見せたくないのであれば、今回のようなビューを作成し、このビューのみ SELECT 可能にすることで、個人情報を見せることを防止できます。

テーブルやビューでは、各データベースのユーザー（スキーマ）ごとにデータを参照するかどうかの権限を設定することができます。今回の例では、「GRANT SELECT ON ビュー名 TO 従業員のスキーマ名」というSQL文を実行し、従業員が使用するスキーマに対してビューのみSELECT文を可能にすることで、テーブルを直接みることはできずに、ビューのデータのみ参照できるようにすることが可能です。

注意点として、ビューは読み取り専用（SELECT のみ可能）となりますので、INSERT や UPDATE などの SQL を使用して、データを編集することはできません。もし、データを編集したい場合は、結合する元のテーブルに対して、SQL を実行する必要があります。ただし、単一テーブルをもとに作成したビューであれば、データの編集が可能です。

3 パフォーマンスを向上する「インデックス」

3-1．インデックスとは

ビューと同じぐらいデータベース開発で欠かせないデータベースオブジェクトが「インデックス」です。インデックスは、テーブルの読み取りや書き込みの向上させるためのデータベースオブジェクトです。テーブルの各レコードを本の「ページ」と例えると、インデックスは「目次」のようなものです。インデックスがない場合は、データベース内部では、上から順番に（INSERT 文により格納された順番に）対象のレコードを検索します。そのため、レコード数が１０万件などある場合は、検索が非常に遅くなってしまうことになります。このような場合はインデックスを付与することでパフォーマンス向上ができます。

EC サイトによくある商品検索機能を例に説明しましょう。商品検索機能では「商品名」や商品の「カテゴリ名」で検索が可能ですが、扱う商品が多い場合、つまり商品マスタテーブルが非常に多い場合は、検索のパフォーマンスが悪化してしまうことが予想されます。

```
-- PRODUCT テーブルの作成
CREATE TABLE PRODUCT
(
    PRODUCT_CODE                NUMBER(10,0),
    CATEGORY_NAME               VARCHAR2(50),
    PRODUCT_NAME                VARCHAR2(50),
    PRICE                   NUMBER
)
/
-- PRODUCT テーブルのデータを作成
INSERT INTO PRODUCT values(1,'ミネラルウォーター','富士の天然水','150');
INSERT INTO PRODUCT values(2,'ミネラルウォーター','ビビアン','150');
INSERT INTO PRODUCT values(3,'お茶','白烏龍茶','250');
INSERT INTO PRODUCT values(4,'お茶','ヘルーシア','120');
INSERT INTO PRODUCT values(5,'お茶','六十茶','160');
INSERT INTO PRODUCT values(6,'コーヒー','ファイアー','120');
INSERT INTO PRODUCT values(7,'コーヒー','DOSS','150');
INSERT INTO PRODUCT values(8,'紅茶','オレンジペコ','150');
INSERT INTO PRODUCT values(9,'紅茶','午前の紅茶','150');
INSERT INTO PRODUCT values(10,'紅茶','はちみつ紅茶','200');

--10000 レコードの  PRODUCT テーブルのダミーデータを作成
DECLARE
    I NUMBER;
BEGIN
    DELETE FROM PRODUCT;
    --1000 回繰り返す
```

```
    FOR I IN 1..10000 LOOP
        -- 1 レコードを INSERT する
          INSERT INTO PRODUCT
          VALUES (I
                  ,'カテゴリ' || TO_CHAR(I)
                  ,'商品名' || TO_CHAR(I)
                  ,TO_CHAR(I * 10) ) ;
    END LOOP;
END
/
```

上記の SQL では、まず PRODUCT テーブルを作成し、さらにデータを 10000 レコード作成しています。インデックスが機能が有効になるには相当数のレコード数が必要となります。なお、このデータを作成する構文は SQL を拡張した「PL/SQL ブロック」と呼ばれるものですが、PL/SQL については第 7 章で詳しくご紹介します。今はあまり気にせず読み進めてください。

3-2. インデックスの作成

それではインデックスを作成しましょう。以下の SQL を実行してください。

```
CREATE INDEX　IDX_PRODUCT
ON　PRODUCT（CUTEGORY_NAME、PRODUCT_ NAME）
/
```

インデックスを作成するには、CREATE INDEX の後にインデックス名を指定します。ON のあとには対象のテーブル名を指定し、さらに、括弧につづけて対象となる列を指定します。複数ある場合はそれぞれの列を「,」で区切る必要があります。この例では PRODUCT テーブルの「CUTEGORY_NAME(カテゴリ名)」「PRODUCT_ NAME(商品名)」に対して、インデックスを作成しています。このインデックスを作成後、

```
SELECT * FROM PRODUCT
WHERE CATEGORY_NAME = 'コーヒー' AND PRODUCT_NAME = 'ファイアー'
/
```

という SQL を発行 します。すると、以下のようにインデックスが使用され、検索の動作がはやくなります。

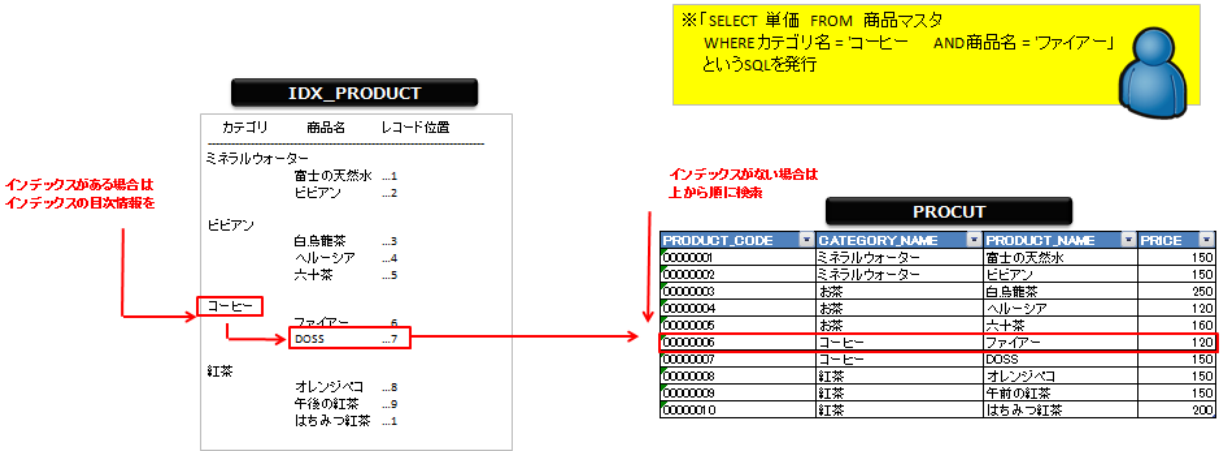

図３－１.インデックスの動作イメージ

データベース内部では、そのインデックス（目次）をみて目的のレコード（ページ）を探すように動作が変更され、目的のレコードまでの到達がはやくなります。この仕組みによりデータを取得するまでのパフォーマンスが向上します。

3-2．インデックスの注意点

ただし、インデックスは以下のような場合には使われないので注意が必要です。

・インデックスの対象列に、SQL で使用される列が含まれていない場合

SQL 文中の SELECT 句や WHERE 句、GROUP BY 句、ORDER BY 句で記述されているテーブル列に対して、インデックスの対象列が含まれている場合に使用されます。今回のインデックスの場合は、商品名やカテゴリ名が含まれる場合に使用されますが、まったく関係のない列を WHERE 句にして SELECT している場合などはインデックスは使わません。

・インデックス数や対象列が多すぎる

かといって、インデックスの対象列を増やしてしまうと、目次のサイズが肥大化してしまいます。1000 ページもある目次なんて意味がないとの同じように、データベース内部でも、サイズの大きいインデックスは使わないという判断がなされます。

また、インデックスはひとつのテーブルにいくつも作成することができますが、インデックス数を増やしすぎてもいけません。テーブルの更新と同時に、インデックスの更新も行われるため、インデックス数が多いと更新のパフォーマンスが悪化してしまうためです。

これらにあてはまる場合、インデックスは使用されませんので、アプリケーションが発行している SQL の中から、対象テーブルレコード数が多く、パフォーマンスが悪化している SQL を特定し、インデックスを作成する必要があります。作成する際は、どの列に対して作成するべきか、また、いくつ作成するかなど必ず検討するようにしましょう。

なお、インデックスを使用するかどうかは、データベースが自動で判断しますが、最適なインデックスにしているにもかかわらず、使われないケースもあります。その場合はヒント句を使用します。SELECT 文を以下のように変更してください。

```
SELECT /*+ INDEX (PRODUCT.IDX_PRODUCT)*/* FROM PRODUCT
WHERE CATEGORY_NAME = 'コーヒー' AND PRODUCT_NAME = 'ファイアー'
/
```

「/*+ ～ */」の中がヒント句と呼ばれるものです。SELECT の直後に挿入します。このように記述した

場合、「この SELECT 文では必ず IDX_PRODUCT というインデックスを使ってね」という命令になり、必ずインデックスを使用してデータ検索されるようになります。ただし、インデックスを使わない方がいい場合でも、必ずインデックスが使われることになりますで、ヒント句は「どうしても」という場合に使用するようにしてください。

4 データの整合性を保つ「制約」

4-1. 制約とは

データの整合性とは、「アプリケーションの欠陥が起きないようなデータの状態となっているか」ということです。図４－１ のように、注文情報を管理する「注文」テーブルと、商品情報を格納する「商品マスタ」テーブルがあったとします。「商品マスタ」には「商品名」や「単価」「備考」という列がありますが、「備考」という列は補足情報を入れる列ですので、データは入っていても、空のデータ（NULL）になっていても構いません。それに対し「商品名」や「単価」という列は、重要な情報ですので、必ずデータが入っている必要があります。そのためには、アプリケーションの商品登録画面にて、「商品名」「価格」の情報が空で灯篭しようとした場合はエラーにするなどの入力チェックを設けなければいけません。この入力チェックがないと、商品名や単価がわからない商品データができてしまうことになります。また、整合性を守るためには、複数のテーブル間による関連も考慮しなければなりません。例えば、商品の削除する機能を設けた場合、削除する前には、あらかじめその商品で注文がないかを確認する必要があります。もし、削除してしまうと、どの商品の注文なのかわからないデータができることになってしまいます。このような状態は「データの不整合が起きている」と呼ばれます。

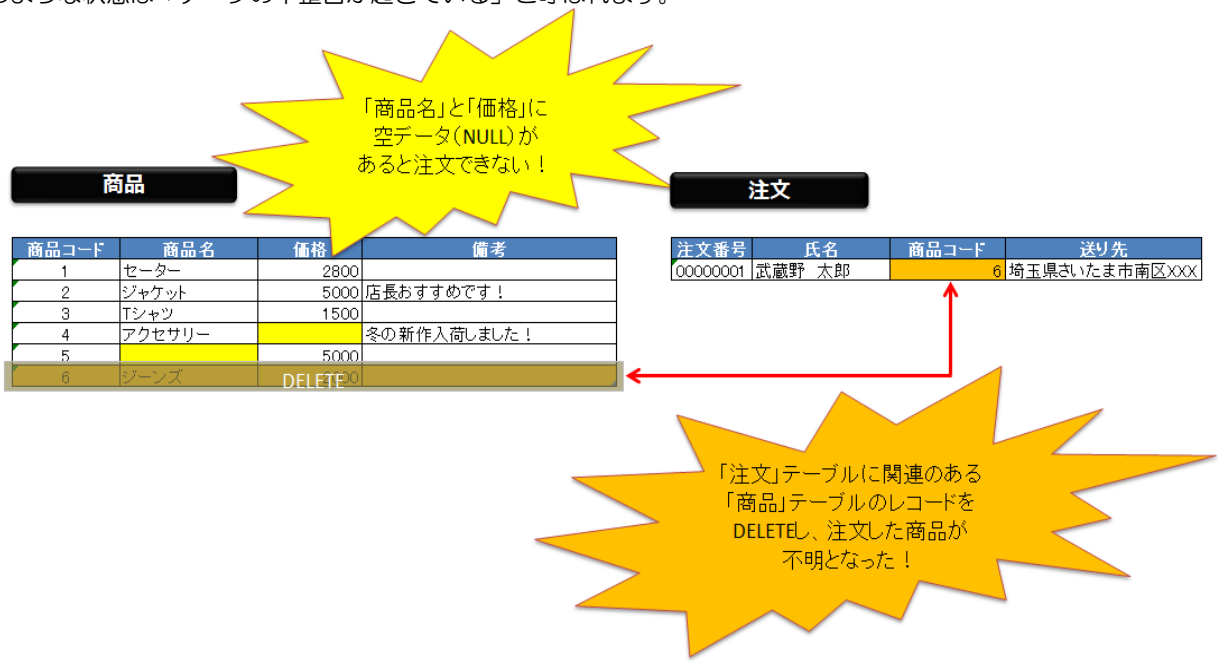

図４－１.データの不整合が起きた状態

データに不整合が起きると、アプリケーション上の重大な欠陥となってしまいますので、のような問題点は、本番環境で稼働する前のテスト段階で取り除き、データの不整合が起きないようにしなければなりません。しかし、開発するアプリケーションが大規模になればなるほど、膨大なテストパターンが必要ですので、すべてのテストを網羅できず、問題点を残したままリリースしてしまうことも充分ありえます。では、データの不整合が絶対に起きないようにするにはどうすればよいのでしょうか？そこで登場するのが「制約」です。制約はテーブルに対してチェックを追加し、データの不整合を防止するデータベースオブジェクトです。制約には、以下のような種類があります。

制約の種類	説明
主キー制約 (PRIMAY KEY 制約)	「値が一意であること」をチェックます。例えば、「商品マスタ」であれば、商品を識別するための「商品コード」など重複してはいけない列に対して指定します。主キー制約はテーブル１つにつき１つになります。
一意制約 (UNIQUE KEY 制約)	「値が一意であること」という保証します。主キー制約と同様ですが、NULL（空のデータ）は許される点と、１つのテーブルにつき、いくつでも作成できる点が異なります。
必須制約 （NOT　NULL 制約）	「値が必ず格納されること」をチェックします。NULL データが格納されないことができなくなります。必須制約は列ごとに指定します。
外部キー制約 （FOREIGN KEY 制約）	「格納する列の値が、関連するテーブルのデータの値と一致しているかどうか」をチェックします。
チェック制約 (CHECK 制約)	上記以外に任意の制限を設けたい場合に使用します。チェックするコードをSQL で指定します。例えば「性別　IN（‘男性’,’女性’）」と記述しておくと、「性別」列には、「男性」か「女性」とどちらかのデータしか格納できなくなります。

表４－１.制約の種類

さきほどの例のように、データの整合性を守るためにチェックする項目は、データが必須であるかどうかだけでなく、複数テーブル間の関連性や、一意な番号になっていることなど、様々な内容があります。また、アプリケーション独自のチェックロジックを設けたいこともあるでしょう。制約ではこれらをすべてチェックできるようになっています。

4-2. 制約の作成

実際にこれらの制約を作成し、動作を確認してみましょう。以下の SQL を実行し、サンプルテーブルと制約を作成してください。

```
--(a)CUSTOMER」テーブルの作成
CREATE TABLE CUSTOMER
(
    CUSTOMER_CODE          NUMBER(7,0)    NOT NULL,
    CUSTOMER_NAME          VARCHAR2(50)   NOT NULL,
    AREA_CODE              NUMBER(7,0)    NOT NULL,
    EMAIL                  VARCHAR2(256),
    SEX                    VARCHAR2(4)
)
/

--(b)CUSTOMER」テーブルのデータを作成
INSERT INTO CUSTOMER VALUES(1,'川口 柊斗',1,'kawaguchi@XXX.co.jp','男性');
INSERT INTO CUSTOMER VALUES (2,'内村 勤',2,'uchimura@YYY.co.jp','男性');
INSERT INTO CUSTOMER VALUES (3,'脇田 彩乃',4,'wakita@ZZZ.co.jp','女性');
INSERT INTO CUSTOMER VALUES (4,'大前 香美',7,'oomae@ABC.co.jp','女性');
INSERT INTO CUSTOMER VALUES (5,'北沢 亮',8,'kitazawa@XYZ.co.jp','男性');
/
```

```
--(c)主キー制約の追加
ALTER TABLE CUSTOMER ADD CONSTRAINT PK_CUSTOMER
PRIMARY KEY (CUSTOMER_CODE)
/

--(d)一意制約の追加
ALTER TABLE CUSTOMER ADD CONSTRAINT UNQ_CUSTOMER
UNIQUE (EMAIL)
/

--(e)チェック制約の追加
ALTER TABLE CUSTOMER ADD CONSTRAINT CHK_CUSTOMER
CHECK (SEX IN ('男性','女性'))
/

--(f)「AREA」テーブルの追加
CREATE TABLE AREA
(
    AREA_CODE                   NUMBER(1,0) PRIMARY KEY,
    AREA_NAME                   VARCHAR2(50) NOT NULL
)
/

--(g)「AREA」テーブルのデータを作成
INSERT INTO AREA VALUES (1,'北海道');
INSERT INTO AREA VALUES(2,'東北');
INSERT INTO AREA VALUES(3,'関東');
INSERT INTO AREA VALUES(4,'関西');
INSERT INTO AREA VALUES(5,'四国');
INSERT INTO AREA VALUES(7,'九州');
INSERT INTO AREA VALUES(8,'沖縄');

--(h)外部キー制約の追加
ALTER TABLE CUSTOMER ADD CONSTRAINT "FK_CUSTOMER_AREA"
FOREIGN KEY (AREA_CODE) REFERENCES AREA (AREA_CODE)
/
```

このSQLではサンプルのテーブルとして「CUSTOMER」「AREA」を作成し、さらに「CUSTOMER」に対して主キー制約、一意制約、チェック制約を作成しています。また、「CUSTOMER」の「エリアコード」と、「エリアマスタ」の「エリアコード」の間に外部キー制約を作成しています。制約はテーブルに追加するオブジェクトとなりますので、作成するSQLにはALTER TABLE文を使用します。「ALTER TABLE 追加する対象のテーブル名 ADD CONSTRAINT 制約名」に続き、各制約の種類ごとに対象の列などの情報を記述します。例えば主キー制約であれば、「PRIMARY KEY(対象の列名)」、外部キー制

約であれば「FOREIGN KEY(対象の列名) REFERENCES 関連するテーブル名(対象の列名)」という記述になります。また、CREATE TABLE 文と同時に制約を追加することも可能です。(e)の SQL 文では「エリアコード」に対して、主キー制約を同時に追加しています。このように記述することでテーブルの作成と同時に主キー制約を作成することができます。例外として、必須制約のみは CREATE TABLE 文内でしか記述できません。必須制約を追加するには列名につづけて「NOT NULL」と記述します。

CREATE TABLE文で同時に制約を作成する場合、その制約のオブジェクト名はデータベース側で自動に決められます。例えば、[SYS_C0012345]といった名前になります。制約のオブジェクト名を自分で設定したい場合はALTER TABLE文を使用してください。

制約の作成後は、動作を確認してみましょう。以下の SQL を実行してください。

```
--(a)顧客コードがすでに格納されているレコードと同じ番号にして INSERT する
INSERT INTO CUSTOMER VALUES(1,'テスト 太郎',1,'test@AAA.co.jp','男性') ;

--(b)メールアドレスがすでに格納されているレコードと同じ値にして INSERT する
INSERT INTO CUSTOMER VALUES(6, 'テスト郎',1,'kawaguchi@XXX.co.jp','男性') ;

--(c)顧客名を NULL で INSERT する
INSERT INTO CUSTOMER VALUES (6,NULL,1,'test@AAA.co.jp','男性') ;

--(d)性別が「男性」「女性」以外の値にして INSERT する
INSERT INTO CUSTOMER VALUES (6,'テスト太郎',1,'test@AAA.co.jp','不明');

(e)エリアコードを「エリアマスタ」にない値にして INSERT する
INSERT INTO CUSTOMER VALUES (6, 'テスト太郎',10,'test@AAA.co.jp','男性') ;
```

上記の INSERT 文は、いずれもデータの不整合を起こす SQL ですが、これらの INSERT 文を実行しても制約のチェックによってすべてエラーとなり、データが格納されることはありません。もし、アプリケーションの不具合によりこのような INSERT 文が実行されてしまっても、即座にエラーを返してくれますので、テスト中にアプリケーションの不具合が発見しやすく、品質向上につなげることができます。

4-3. 制約を一時的に無効にする

このように役立つ制約ですが、開発中にはこの制約がわずらわしい場合があります。例えば筆者の場合は、プログラミングが一通り終わっていざテストに入る前は、いったん作成中のテーブルデータをすべて削除するようにしていますが、上記のエリアマスタのように外部キー制約が設定されたレコードをすべて削除しようとすると、外部キー制約違反となり、削除することができません。いったんデータをすべてクリアするような場合など、テーブルデータを一時的に不整合な状態にする場合は、以下のような SQL 文で制約をいったん無効（制約がないのと同じ状態）にすることにより可能です。

```
ALTER TABLE CUSTOMER DISABLE CONSTRAINT FK_CUSTOMER_AREA;
```

制約を再び有効にする場合は上記の「DISABLE」の箇所を「ENABLE」に変更し、再実行してください。

5 順序を保証するシーケンス

5-1.シーケンスとは

前章でも整合性を保つ制約についてご説明しましたが、データの順序整合性を保つのに役立つオブジェクトとして「シーケンス」があります。シーケンスは正しい「連番」を生成するためのデータベースオブジェクトです。さきほど作成した「CUSTOMER」テーブルには「顧客コード」という列がありましたが、このように、テーブルには一意に識別するための列を設けるのが慣習となっています。レコードごとに一意なデータを設定しておくと、「エリアマスタ」などの関連テーブルと結合しやすくなる、インデックスの対象列として使用することで検索が速くなるなどのメリットがあるからです。

なお、このような一意に識別するコード列に対しては、通常、主キー制約を設けますが、主キー制約を設けると内部では自動的にインデックスが作成されます。

しかし、このような列をテーブルに設けると、アプリケーション側からレコードを挿入する際に、一意な連番を生成する必要が出てきます。このプログラミングはけっこう面倒なものです。単純な方法としては、INSERT する直前に以下のような SELECT 文で次に格納すべき連番を求める方法があります。

```
SELECT MAX(CUSTOMER_CODE)+1 FROM CUSTOMER;
```

MAX 関数は取得したすべてのレコードの中から、最も値が大きい値を取得する関数です。この例では、「CUSTOMER」テーブルに現在格納されているレコードの中から、最大の顧客コードの値を取得し、さらにその番号を＋１した値を取得ています。この SQL で、次に格納するべき顧客コードの連番が求められますので、この値をもとに INSERT 文を実行することが可能です。「なんだ。そんなに難しくないじゃないか」と思われた読者の方もいらっしゃるかもしれません。しかし、実はこの方法では、複数のユーザーが同時に INSERT を実行しようとした場合、同じ顧客コードが生成されてしまうという問題があります。

データベースには、SELECTする際にテーブルをロックする機能があります。テーブルをいったんロックすると、そのロックを解除するまで、他のユーザーはテーブルデータを参照することができませんので、ロックする方法で複数のユーザーが同時実行しても、同じ番号が取得される問題を回避することが可能です。ただし、この方法においても、ロック待ちによるパフォーマンスの無駄が発生してしまう問題があります。

頻繁に更新されるテーブルでない場合はあまり問題ありませんが、もっと安全に連番を生成する方法があれば、それを使用するにこしたことはありません。そこで登場するのが「シーケンス」です。このデータベースオブジェクトを使用することにより、安全に番号を取得することができます。

5-2.シーケンスの作成

さきほどの「CUSTOMER」テーブルの「顧客コード」用にシーケンスを作成してみましょう。以下の SQL を実行してください。

```
CREATE SEQUENCE SEQ_CUSTOMER_CODE
INCREMENT BY 1
START WITH 6
```

```
MAXVALUE 9999999
/
```

シーケンスを作成するには「CREATE SEQUENCE」に続き、シーケンス名をします。ここでは「SEQ_CUSTOMER_CODE」という名前にしています。「INCREMENT BY」の後に増分値、「START WITH」の後に初期値を記述します。さきほどのサンプル SQL にて、すでに５レコード格納しているため、今回は６から始まるようにしています。MAXVALUE には最大値を指定します。「顧客コード」の長さは「NUMBER (7,0)、」つまり７桁としていますので、「9999999」が最大値です。

なお、「シーケンス」オブジェクトはOracleデータベースにはありますが、その他のデータベースには搭載されていないことがあります。Microsoftの「SQL Server」ではIDENTITYプロパティで同等の処理ができます。

シーケンスの作成後は、INSERT 文にて以下のようにシーケンスを使用して連番を生成します。

```
INSERT INTO CUSTOMER VALUES (SEQ_CUSTOMER_CODE.NEXTVAL, 'テスト太郎',1,'test@AAA.co.jp','男性');
```

「顧客コード」の箇所に「SEQ_顧客コード.NEXTVAL」と記述しています。これはシーケンスの NEXTVAL 関数を使用するというものです。NEXTVAL 関数を実行するごとに「7」、「8」「9」...と順に値が生成され、一意な顧客コードを割り当てることができます。このようにシーケンスを使用して連番を生成すると、複数のユーザーから同時に INSERT が走っても同じ番号が割り振られることはありません。

6 アクセス効率を向上する「シノニム」

6-1.シノニムとは

シノニムは「スキーマの違いを意識しなくすためのオブジェクト」です。
テーブルやビューなど、データベースオブジェクトを作成した場合は、SQL を実行したユーザー、つまりログイン中のユーザーがそのデータベースオブジェクトの所有者となります。スキーマとデータベースオブジェクトは親子関係となりますので、あるひとつのデータベースオブジェクトは、必ずどれかひとつのデータベースが所有する関係となります。
データベースユーザーのことは、データベースオブジェクトの所有者という意味で、「スキーマ」と呼ばれます。

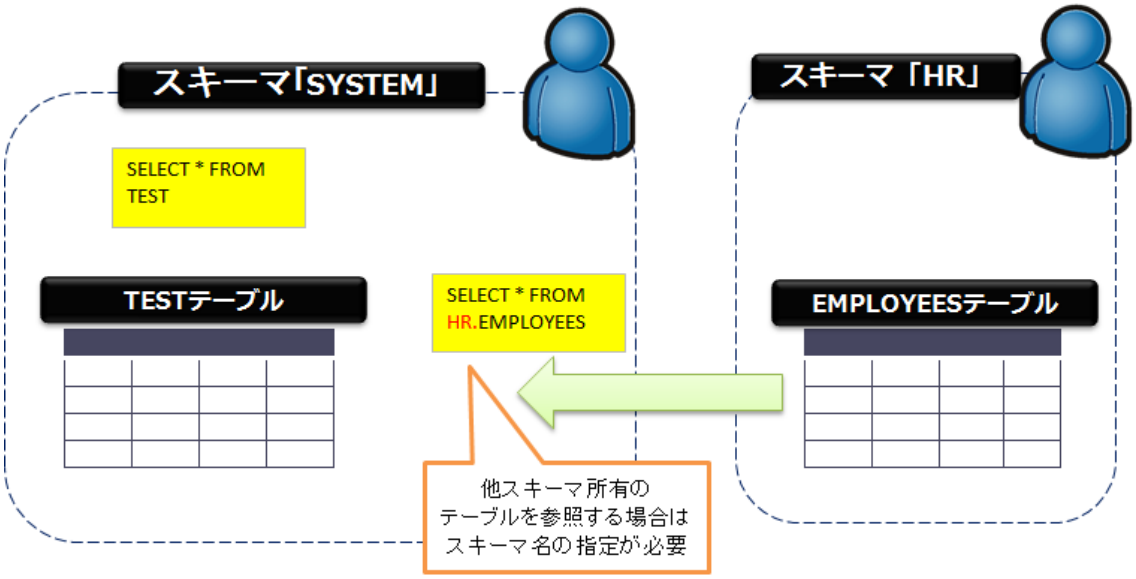

図６－１.スキーマとデータベースオブジェクトの関係

もし、他のスキーマが持っているテーブルに対して SELECT 文などの SQL を実行したい場合は、必ず SQL 文に「スキーマ名」の指定が必要となります。Oracle をインストールすると、「HR」というスキーマが既定で存在しますが、もし HR ユーザー以外のユーザーでデータベースにログインし、HR ユーザーが所有している EMPLOYEES というテーブルを参照したい場合は、「SELECT ＊ FROM HR.EMPLOYEE」と記述する必要があります。実際に動作を確認してみましょう。データベースにログイン後、それぞれ以下の SQL を実行してください。

```
SELECT * FROM EMPLOYEES;        --エラー
SELECT * FROM HR. EMPLOYEES;    --OK
```

上の SQL では FROM 句にテーブル名のみ記述しています。この場合ログインユーザーが所有しているテーブルの中から、EMPLOYEES テーブルを検索するため、「表またはビューがみつかりません。」というエラーになってしまいます。一方、下の SQL では FROM 句にスキーマ名も指定しているため、HR ユーザーの EMPLOYEES データが正常に取得されます。このようにスキーマが別の場合でも、スキーマ名を指定することで正常に SQL を実行することができますが、スキーマの指定を毎回行うのは、けっこう面倒なものです。
実際の開発では複数のシステムと連携することがよくあります。例えば、商品の注文情報を管理する「販

売管理システム」と、社員の個人情報を管理する「人事システム」、会計情報を管理する「会計システム」というシステムがあった場合、会計システム上では、販売管理の注文データをもとに売上データの作成や、人事システムから社員の口座情報を調べて給与の支払データの作成などの処理が必要でしょう。通常は、各システムごとに、専用のスキーマを作成し、個々にテーブルデータを管理しますので、会計システム上からその他のシステムのデータを参照する場合は、スキーマ名の指定が必要となってしまいます。

個々の SQL ごとに会計システムのスキーマ名、販売管理システムのスキーマ名、スキーマ名を指定しない３パターンが必要となり、使い分けるのはかなり手間です。そのような場合に使えるのが今回ご紹介するシノニムです。

6-2.シノニムの作成

シノニムを作成する SQL は以下の通りです。実際に作ってみましょう。

```
CREATE SYNONYM EMPLOYEES FOR HR.EMPLOYEES;
```

「CREATE SYNONYM」に引き続き、シノニム名を指定します。シノニム名は「別名」の意味の通り、別の名前にすることができます。今回は、同じ「EMPLOYEES」としています。「FOR」に続き、参照元のオブジェクト名を指定します。ここでは「HR.EMPLOYEES
」となります。シノニムの作成時は、必ず「スキーマ名.オブジェクト名」の形式にしなければなりませんのでご注意ください。
また、「CREATE PUBLIC SYNONYM」に変更すると、パブリックシノニムとして作成されます。SYSTEM ユーザーだけでなく、すべてのスキーマから共通でシノニムを使用したい場合は、パブリックシノニムにします。今回の例では PUBLIC の指定はありませんので、シノニムを作成したユーザー（ログインユーザー）のみ使用可能なシノニムとなります。では、このシノニムを作成後、もう一度さきほどの SQL 文を実行してみましょう。

```
SELECT * FROM EMPLOYEES; --OK
```

今度は FROM 句にスキーマ名はありませんが、正常にテーブルデータが取得されます。この SQL では、まず、シノニムである「EMPLOYEES」を参照し、シノニムで設定されている「HR.EMPLOYEES」経由でデータを参照しています。今回の例ではシノニムをテーブル名と同じ名前にしましたが、このようにしておければ、あたかもログインユーザーが EMPLOYEES テーブルを所有しているかのように利用することができます。
シノニムはちょうど Windows の「ショートカットファイル」と同じようなものです。1 度ショートカットファイルを作ってしまったあとは、実際のファイルがどこのフォルダ格納されているのかを意識することなく、ショートカットをダブルクリックすることでそのファイルにアクセス可能になります。また、デスクトップなどに作ればすばやいアクセスも可能です。シノニムも同じように、必要なテーブルが複数のスキーマでバラバラに所有されている場合でも、簡易にアクセスが可能となります。

Oracle上で現在日付を求めるSQLとして「SELECT SYSDATE FROM DUAL」という有名なSQLがありますが、実はこの「DUAL」もパブリックシノニムです。このようにシノニムがあれば、ユーザーはどのユーザーが所有しているかを意識する必要なく、お使いいただいていることがわかると思います。

また、関連するデータベースオブジェクトして、「データベースリンク」があります。データベースリンクの詳細な説明は本ドキュメントでは割愛しますが、データベースリンクは他のデータベース上にあるデータベースオブジェクトにアクセス可能にするものです。参照したいテーブルが同一データベース上の他のスキーマにあるのであれば「シノニム」、他のデータベースであれば「データベースリンク」と使い分けることで、テーブルの所有者やデータベースが異なっていても容易にアクセス可能となります。

7 パフォーマンスを向上するストアドプログラム

7-1.ストアドプログラムとは

ストアドプログラムは、データベース内に蓄積（ストア）し、データベース上で処理を向上するための特殊なデータベースオブジェクトです。データベースと連携したアプリケーションでは、アプリケーションのソースコード内に、SQL の処理を記述するのが普通です。この場合、データベースとアプリケーションで SQL の命令や SQL の結果データの送受信が行われます。繰り返し処理などで、何度も SQL を発行する場合や、クライアントサーバーシステムなど、データベースと、プログラムを実行するマシンが別々になるような環境においては、かなりパフォーマンスの低下を招いてしまいます。このような場合に役立つのがストアドプログラムです。ストアドプログラムはデータベースの内部で蓄積（ストア）され、実行もデータベース上で直接行われるため、パフォーマンスアップを見込むことができます。

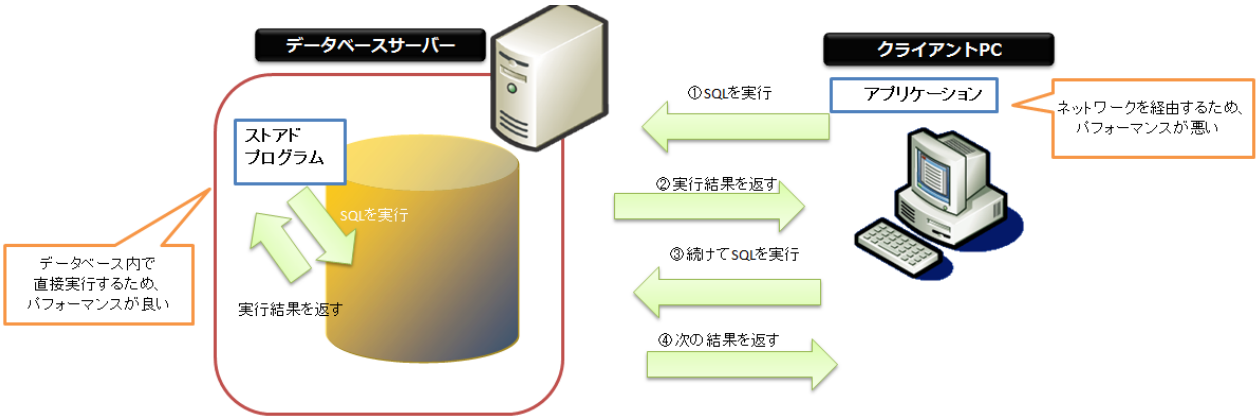

図７－１.通常のプログラムとストアドプログラムの違い

ストアドプログラムは、「プロシージャ」と「ファンクション」の2種類があります。両者の違いは、「戻り値を返すか、返さないか」です。プロシージャが処理を実行するだけであるのに対して、ファンクションには戻り値を返すことができます。ファンクションとして作成する場合は、「CREATE FUNCTION」文、プロシージャの場合は「CREATE PROCEDURE」文で作成します。

7-1.ストアドプログラムの作成

実際に、第６章で使った HR.EMPLOYEES テーブルのデータを CSV（カンマ区切りのデータ）として表示するサンプルを作成してみましょう。以下の SQL を実行してください。

```
CREATE FUNCTION OUTPUT_EMPLOYEES_CSV(P_DEPARTMENT_ID NUMBER)
RETURN VARCHAR2 IS
/****************************** 宣言部 ******************************/
    --カーソルの宣言
    CURSOR EMPLOYEES_CUR IS
    SELECT FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER
    FROM HR.EMPLOYEES
    WHERE DEPARTMENT_ID = P_DEPARTMENT_ID;
    --カーソル変数の宣言
    EMPLOYEES_REC EMPLOYEES_CUR%ROWTYPE;
    --文字列変数の宣言
    COLUMN_VALUES VARCHAR2(255);
/****************************** 処理部 ******************************/
BEGIN
    --対象レコードがなくなるまで繰り返す
    FOR EMPLOYEES_REC IN EMPLOYEES_CUR LOOP
        -- 1 行分のデータを文字列に代入し、画面に表示
        COLUMN_VALUES := EMPLOYEES_REC.FIRST_NAME      || ','
                       || EMPLOYEES_REC.LAST_NAME      || ','
                       || EMPLOYEES_REC.EMAIL          || ','
                       || EMPLOYEES_REC.PHONE_NUMBER;
        DBMS_OUTPUT.PUT_LINE(COLUMN_VALUES);
    END LOOP;
    RETURN 'SUCCESS!';
/****************************** 例外処理部 ******************************/
EXCEPTION
    WHEN OTHERS THEN
        --エラーメッセージを画面に表示
        DBMS_OUTPUT.PUT_LINE('ERROR! :' || SQLERRM);
        --エラーメッセージを返す
        RETURN 'ERROR!' || SQLERRM;
END;
/
```

ストアドプログラムの作成においては、PL/SQL（Programming Language for SQL）という言語を使用します。PL/SQL は、通常の SQL を拡張し、変数や IF 文、繰り返し命令、エラー処理などのプログラムに必要な処理を記述可能にした Oracle 専用のプログラム言語ものです。

PL/SQLはOracle専用の言語ですが、その他のデータベースでもストアドプログラムはサポートされています。例えばSQL Serverでも同様にSQLを拡張した「Transact-SQL」という言語があり、その機能もPL/SQLと同等になります。

PL/SQL のコードを大きく分けると定義部、宣言部、処理部、例外処理部に分かれます。それぞれごとにコードをみていきましょう。

・定義部
プログラムの名前と引数の定義です。「CREATE FUNCTION」に引き続きストアドプログラム名を指定しています。ここでは「OUTPUT_EMPLOYEES_CSV」がプログラム名です。引数を指定する場合は、そのあとの括弧を記載し、括弧の中に引数名、データ型を記述します。ここでは、「P_DEPARTMENT_ID」という NUMBER 型の引数を記述しています。また、ストアドファンクションの場合は戻り値のデータ型を「RETURNS」につづけて指定します。当サンプルでは、最後に成功したか失敗したかの結果を文字列として返しますので、戻り値のデータ型として「VARCHAR2」を指定しています。

・宣言部
定義部の後の「IS」に続けて、ストアドプログラム内で使用する変数の定義を行います。今回のサンプルでは、カーソルと呼ばれる変数を定義しています。ストアドプログラムには、「カーソル」と呼ばれる、SQL の実行結果を格納する独自の変数があります。ここでは、「EMPLOYEES_CUR」という、カーソルを作成しています。また、カーソル名の後には続けて SELECT 文を記述しています。EMPLOYEES テーブルから、氏名、EMAIL、電話番号を取得する SQL にしています。また、WHERE 条件にて、DEPARTMENT_ID（部門 ID）が引数の P_DEPARTMENT_ID と一致することを条件としています。このようにしておくと、実行する際に引数で渡された DEPARTMENT_ID に一致するレコードだけがカーソルに格納されます。また、カーソルを使用する場合は、その実行結果を 1 レコード単位で受け取るための「カーソルレコード変数」もセットで宣言します。「EMPLOYEES_REC」がカーソル変数です。

・処理部
ストアドプログラムのメインのブロックとなります。「BEGIN」に続けて、プログラムの実際の処理を記述します。今回のサンプルでは、カーソル FOR ループを実行して 1 行づつ CSV に出力しています。カーソルの結果を 1 レコードずつ取得しながら、繰り返し処理を行う命令です。1 行分のカンマ区切りで連結した結果を文字列変数に代入し、画面に出力しています。DBMS_OUTPUT.PUT_LINE という関数で出力できます。レコードをなくなった場合は FOR ループの外に脱出し、最後に「SUCCESS！」という文字列を返して終了しています。

・例外処理部
ストアドプログラムの実行中、不正な処理が行われた場合は、例外が発生し、処理が中断されます。もし、例外が発生したときに、何らかの後処理を行い場合は、「EXCEPTION」に続けて、例外発生時の処理を記述します。例外処理部では、まず「WHEN」の後につづけて例外の種類を記述します。例外と一言にいっても、データが見つからない場合、0 で割り残した場合などの様々な例外の種類があります。もし、例外の種類によって処理を分岐したい場合は WHEN 句を複数記述することで条件分岐をすることができます。今回のサンプルでは、「WHEN OTHER」と 1 つだけ記述していますが、これは、どのような例外が発生した場合も同一の処理をする場合の記述方法になります。続けて実際の処理を記述しますが、今回は、sqlerrm という変数を画面に表示しています。ストアドプログラムの実行途中で例外エラーが発生した場合は、sqlerrm という変数に自動的にエラーメッセージが代入されますので、このメッセージを画面に表示しています。このようにしておくと、例外発生時にそのエラー内容が画面に表示されますので、ユーザーはエラーの原因をつかむことができます。例外処理部は必須ではありませんが、もし、プログラム内に INSERT や UPDATE などのトランザクションを行う場合は、例外処理部を記載し、例外発生時は、ROLLBACK を発行するようにするとよいでしょう。

なお、もし、ストアドプログラムのコードに誤りがあると、実行した際にエラーとなりますので、その際は、適宜修正後、再度実行してください。なお、一度ストア度プログラムを作成後、再作成する場合は、「DROP FUNCTION(PROCEUDRE) ストアドプログラム名」→「FUNCTION(PROCEUDRE) ストアドプログラム名」の手順で再作成する必要がありますが、ストアドプログラムの場合は他のデータベースオブジェクトと異なり、「CREATE OR REPLACE FUNCTION(PROCEUDRE) ストアドプログラム名」と記述することもできます。これは、すでに同じ名前のファンクションを作成していた場合は再作成するという SQL になります。ストアドプログラム作成においては、コードの記述ミスで何度も作成しなおすことがよくあります。その場合は、CREATE OR REPLACE の SQL で簡単に再作成することができます。

ストアドプログラム作成した後は、実際に実行してみましょう。ストアドプログラムを実行する SQL は以下の通りです。

```
DECLARE
  RET VARCHAR2(10);
BEGIN
  RET := OUTPUT_EMPLOYEES_CSV(50);
  DBMS_OUTPUT.PUT_LINE(RET);
END
/
```

まず、大きく DECLARE～BEGIN～END ブロックで囲み、DECLARE の後には戻り値を受け取る変数を宣言し、BEGIN～END の中で、ストアドプログラム名を記載します。続く括弧の中には引数として「50」を渡します。実は、ストアドプログラムを呼び出す SQL は、PL/SQL のコードとして記述する必要がありますが、このように DECLARE～BEGIN～END ブロックで囲むと、データベース側では、PL/SQL のコードとして認識され、ストアドプログラムの呼び出しが可能となります。なお、プロシージャの場合は、戻り値を受け取る必要がありませんので、DECLARE ブロックはなくても構いません。

Oracle標準ツールである「SQL*Plus」では「EXECUTE ストアドプログラム名（引数）」でプロシージャ、ファンクションの両方を実行することが可能です。ただし、「EXECUTE」はSQL*Plus専用のコマンドとなり、その他のアプリケーションからは実行できませんのでご注意ください。

この SQL を実行すると、DEPARTMENT_ID が 50 となっているレコードが、CSV データとして表示され、最後にファンクションから受け取った戻り値（「SUCCESS」か、エラーメッセージ）が画面に表示されます。

今回、PL/SQL については基本的なものしか解説できませんでしたが、その他にも一般のプログラミング言語でサポートしているような処理が可能です。例えば、今回は CSV データを画面に表示するサンプルでしたが、これを、ファイルに出力することも可能です。PL/SQL の専用の解説本も多くありますので、興味があれば、ぜひそちらなどで理解を深めていただければと思います。

また、Oracle では「パッケージ」というデータベースオブジェクトもあります。詳細な説明は本ドキュメントでは割愛させていただきますが、複数のストアドプログラム/ストアドファンクションをひとまとめにしたものです。関連するプロシージャ、ファンクションをモジュール化して管理できる他、一度パッケージを実行した時点で、パッケージ全体の情報がメモリにロードされるため、個々のストアドプログラムを別々に実行するよりも、はやく処理できるなどのメリットがあります。

最後に

本ドキュメントは以上になります。

その他にも「タイプ」や「トリガー」「マテリアライズドビュー」など、ご紹介できなかったデータベースオブジェクトもありますが、基本的にはこれまでご紹介した「アクセスをしやすくする」「パフォーマンスを向上する」「整合性を保つ」のいずれかの目的で作られた機能です。各データベースオブジェクトを使用するメリットをおさえていただいた上で、ぜひ、実際の開発現場に活用してください！

また、弊社の Oracle 開発支援ツールとして「SI Object Browser」という製品があります。

GUI でもデータベースオブジェクトが作成でき、開発効率を高めることができるツールです。

（ＳＱＬの勉強には向いていないかもしれませんが…）

ブログでも画面入りでご紹介していますので、ご興味があればぜひご覧ください。

<製品ホームページ>

https://products.sint.co.jp/siob

<ブログ>

https://products.sint.co.jp/siob/blog

お問い合わせ先

本ドキュメントや弊社製品に関しましてご不明点がございましたらお知らせく ださい。

株式会社システムインテグレータ Object Browser 事業部 営業担当

TEL ： 03-5768-7979 東京営業所 後迫（ウシロザコ）、横島

06-4706-5471 大阪支社 永田

E-Mail： oob@sint.co.jp

URL ： https://products.sint.co.jp/siob/inquiry