Oracle9*i*

アプリケーション開発者ガイド - オブジェクト・リレーショナル機能

リリース 2（9.2）

2002 年 7 月

部品番号：J06311-01

ORACLE®

Oracle9*i* アプリケーション開発者ガイド - オブジェクト・リレーショナル機能 , リリース 2（9.2）

部品番号：J06311-01

原本名：Oracle9*i* Application Developer's Guide - Object-Relational Features, Release 2 (9.2)

原本部品番号：A96594-01

原本著者：William Gietz

原本協力者：C. Dupree、G. Arora、C. Iyer、A. Manikutty、A. Yoaz、Q. Yu

目次

3 Oracle プログラム環境のオブジェクト・サポート

4 Oracle オブジェクトの管理

5 オブジェクト・モデルのリレーショナル・データへの適用

6 Oracle オブジェクトの高度なトピック

7 Oracle オブジェクトの使用方法に関する FAQ

8 Oracle オブジェクトの設計上の考慮点

9　オブジェクト・リレーショナル機能を使用したサンプル・アプリケーション

索引

はじめに

このマニュアルでは、Oracle サーバー・リリース 2（9.2）のオブジェクト・リレーショナル機能の使用方法について説明します。このマニュアルの内容は、すべてのプラットフォームで動作する Oracle サーバーのバージョンに適用されますが、システム固有の情報は含みません。

ここでは、次の項目について説明します。

対象読者

このマニュアルは、新しいアプリケーションを開発するプログラマや、既存のアプリケーションを Oracle 環境で実行できるように変換するプログラマを対象としています。オブジェクト・リレーショナル機能は、マルチメディア、地理情報システム（GIS）および複雑なデータを処理するアプリケーションでよく使用されます。オブジェクト・ビュー機能は、既存のリレーショナル・スキーマ上に新しいアプリケーションを作成する場合に役立ちます。

このマニュアルは、アプリケーション・プログラミングの実践的な知識があり、Structured Query Language（SQL）を使用してリレーショナル・データベース・システム内の情報にアクセスする操作を理解していることを前提としています。

このマニュアルの構成

このマニュアルは、次のように構成されています。

第 1 章「Oracle オブジェクトの概要」

オブジェクト・リレーショナル・モデルの主な機能および利点について説明します。

第 2 章「Oracle オブジェクトの基本コンポーネント」

Oracle オブジェクトを扱う場合に必要な、基本的な概念および用語について説明します。

第 3 章「Oracle プログラム環境のオブジェクト・サポート」

SQL、PL/SQL、Oracle Call Interface（OCI）、Pro*C/C++、Oracle Objects For OLE（OO4O）、Java、Java Database Connectivity（JDBC）および Oracle SQLJ におけるオブジェクト・リレーショナル機能について説明します。この章の内容は、教育および計画向けの高度なものです。次の章以降で、オブジェクト・リレーショナル機能の使用方法について詳細に説明します。

第 4 章「Oracle オブジェクトの管理」

オブジェクトおよびオブジェクト型を使用した、基本操作の実行方法について説明します。

第 5 章「オブジェクト・モデルのリレーショナル・データへの適用」

オブジェクト・ビューについて説明します。オブジェクト・ビューを使用すると、基礎となるリレーショナル・スキーマを変更することなくオブジェクト指向アプリケーションを開発できます。

第 6 章「Oracle オブジェクトの高度なトピック」

オブジェクト指向アプリケーションをスケールアップする場合に、記憶域およびパフォーマンスを管理するために必要な機能について説明します。

第 7 章「Oracle オブジェクトの使用方法に関する FAQ」

オブジェクト指向プログラミングを使用したプログラミングを始める場合、または他のデータベース・システムやオブジェクト指向言語などの経験者が Oracle に取り組む場合に役立つヒントを提供します。

第 8 章「Oracle オブジェクトの設計上の考慮点」

Oracle のオブジェクト指向モデルの実装法およびパフォーマンス特性について説明します。

第 9 章「オブジェクト・リレーショナル機能を使用したサンプル・アプリケーション」

リレーショナル・プログラムを、オブジェクト指向プログラム、スキーマなどに書きなおす方法を紹介します。

関連文書

詳細は、次の Oracle ドキュメントを参照してください。

- 『PL/SQL ユーザーズ・ガイドおよびリファレンス』には、オラクル社が開発した SQL の手続き型拡張機能である PL/SQL および高水準プログラミング言語が詳しく説明されています。
- 『Oracle9*i* アプリケーション開発者ガイド - 基礎編』には、アプリケーション開発に関する一般的な情報が記載されています。
- 『Oracle9*i* JDBC 開発者ガイドおよびリファレンス』および『Oracle9*i* Java Stored Procedures Developer's Guide』には、Java を介して Oracle のオブジェクト・リレーショナル機能を使用する方法が説明されています。
- 『Oracle Call Interface プログラマーズ・ガイド』には、Oracle サーバーにアクセスする 3GL アプリケーションを OCI により作成する方法が説明されています。
- 『Pro*C/C++ Precompiler プログラマーズ・ガイド』は、オラクル社のプリコンパイラ Pro* シリーズの説明書です。このシリーズを使用することで、Ada、C、C++、COBOL、FORTRAN のいずれかで書かれた 3GL アプリケーション・プログラムに SQL および PL/SQL を組み込むことができます。
- Oracle Developer/2000 は、フォーム作成プログラム、レポート作成ツール、PL/SQL 用のデバッグ環境などを含む数種類のツールを提供するコオペラティブ開発環境です。Developer/2000 を使用する場合は、該当する Oracle のツール製品のドキュメントを参照してください。
- 『Oracle9*i* SQL リファレンス』および『Oracle9*i* データベース管理者ガイド』には、SQL に関する情報が記載されています。
- 『Oracle9*i* データベース概要』には、Oracle の基本概念が説明されています。

ドキュメント・セットの多くのマニュアルで、Oracle のインストール時にデフォルトでインストールされるシード・データベースのサンプル・スキーマを使用しています。サンプル・スキーマの作成方法および使用方法の詳細は、『Oracle9*i* サンプル・スキーマ』を参照してください。

リリース・ノート、インストール・マニュアル、ホワイト・ペーパーまたはその他の関連文書は、OTN-J（Oracle Technology Network Japan）に接続すれば、無償でダウンロードできます。OTN-J を使用するには、オンラインでの登録が必要です。次の URL で登録できます。

```
http://otn.oracle.co.jp/membership/
```

すでに OTN-J のユーザー名およびパスワードを取得済であれば、次の OTN-J Web サイトの文書セクションに直接接続できます。

```
http://otn.oracle.co.jp/document/
```

表記規則

この項では、このマニュアルの本文およびコード例で使用される表記規則について説明します。この項の内容は次のとおりです。

本文中の表記規則

本文では、特別な用語をより迅速に識別できるように、様々な表記規則を使用します。次の表に、それらの表記規則を説明し、その使用例を示します。

規則	意味	例
太字	太字は、本文中で定義されている用語、または用語集に出現する用語、あるいはその両方を示します。	この句を指定すると、**索引構成表**が作成されます。
固定幅フォントの大文字	固定幅フォントの大文字は、システムが提供する要素を示します。このような要素には、パラメータ、権限、データ型、Recovery Manager キーワード、SQL キーワード、SQL*Plus またはユーティリティ・コマンド、パッケージおよびメソッドが含まれます。また、システムが提供する列名、データベース・オブジェクト、データベース構造、ユーザー名およびロールも含まれます。	`NUMBER` 列に対してのみに、この句を指定できます。 `BACKUP` コマンドを使用して、データベースのバックアップを取ることができます。 `USER_TABLES` データ・ディクショナリ・ビュー内の `TABLE_NAME` 列を問い合せます。 `DBMS_STATS.GENERATE_STATS` プロシージャを使用します。

規則	意味	例
固定幅フォントの小文字	固定幅フォントの小文字は、実行可能ファイル、ファイル名、ディレクトリ名およびユーザーが提供する要素のサンプルを示します。このような要素には、コンピュータ名、データベース名、ネット・サービス名および接続識別子が含まれます。また、ユーザーが提供するデータベース・オブジェクト、データベース構造、列名、パッケージとクラス、ユーザー名とロール、プログラム・ユニットおよびパラメータの値も含まれます。	`sqlplus` と入力して、SQL*Plus をオープンします。
		パスワードは、`orapwd` ファイルで指定します。
		`/disk1/oracle/dbs` ディレクトリ内のデータ・ファイルおよび制御ファイルのバックアップを取ります。
		`hr.departments` 表には、`department_id`、`department_name` および `location_id` 列があります。
	注意：大文字と小文字を組み合せて使用するプログラム要素もあります。これらの要素は、記載されているとおり入力してください。	`QUERY_REWRITE_ENABLED` 初期化パラメータを `true` に設定します。
		`oe` ユーザーとして接続します。
		`JRepUtil` クラスが次のメソッドを実装します。
固定幅フォントの小文字のイタリック	固定幅フォントの小文字のイタリックは、プレースホルダまたは変数を示します。	*`parallel_clause`* を指定できます。
		U*`old_release`*`.SQL` を実行します。ここで、*`old_release`* はアップグレードする前にインストールされているリリースを示しています。

コード例中の表記規則

コード例は、SQL、PL/SQL、SQL*Plus または他のコマンドライン文を説明します。コード例は、固定幅フォントで表示され、この例に示すとおり通常のテキストと区別されます。

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

次の表に、コード例で使用される表記規則を説明し、その使用例を示します。

規則	意味	例
[]	大カッコは、任意に選択する 1 つ以上の項目を囲みます。大カッコは、入力しないでください。	`DECIMAL (`*`digits`* `[ , `*`precision`* `])`
{}	中カッコは、2 つ以上の項目を囲み、そのうちの 1 つの項目は必須です。中カッコは、入力しないでください。	`{ENABLE | DISABLE}`
|	縦線は、大カッコまたは中カッコ内の 2 つ以上のオプションの選択項目を表します。オプションのうちの 1 つを入力します。縦線は、入力しないでください。	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`

規則	意味	例
...	水平の省略記号は、次のいずれかを示します。 ■ 例に直接関連しないコードの一部が省略されている。 ■ コードの一部を繰り返すことができる。	CREATE TABLE ... AS *subquery*; SELECT *col1*, *col2*, ... , *coln* FROM employees;
. . .	垂直の省略記号は、例に直接関連しない複数の行が省略されていることを示します。	SQL> SELECT NAME FROM V$DATAFILE; NAME ---------- /fs1/dbs/tbs_01.dbf /fs1/dbs/tbs_02.dbf . . . /fs1/dbs/tbs_09.dbf 9 rows selected.
その他の句読点	大カッコ、中カッコ、縦線および省略記号以外の句読点は、表示されているとおり入力する必要があります。	acctbal NUMBER(11,2); acct CONSTANT NUMBER(4) := 3;
イタリック体	イタリック体は、特定の値を指定する必要のあるプレースホルダまたは変数を示します。	CONNECT SYSTEM/*system_password* DB_NAME = *database_name*
大文字	大文字は、システムが提供する要素を示します。これらの用語は、ユーザー定義の用語と区別するために大文字で示されます。用語が大カッコ内にないかぎり、表示されているとおりの順序および綴りで入力します。ただし、これらの用語は大 / 小文字が区別されないため、小文字でも入力できます。	SELECT last_name, employee_id FROM employees; SELECT * FROM USER_TABLES; DROP TABLE hr.employees;
小文字	小文字は、ユーザーが提供するプログラム要素を示します。たとえば、表名、列名、ファイル名などです。 **注意 :** 大文字と小文字を組み合せて使用するプログラム要素もあります。これらの要素は、記載されているとおり入力してください。	SELECT last_name, employee_id FROM employees; sqlplus hr/hr CREATE USER mjones IDENTIFIED BY ty3MU9;

オブジェクト・リレーショナルの新機能

ここでは、Oracle9*i* リリース 2（9.2）のオブジェクト・リレーショナルの新機能について説明します。

Oracle9*i* リリース 2（9.2）のオブジェクト・リレーショナルの新機能

- **型のシノニム**

  ユーザー定義型に対してシノニムを定義すると、型が定義されているスキーマの名前で型の名前を修飾することなく、型を使用できます。

  参照： 4-8 ページの「ユーザー定義型のシノニム」を参照してください。

- **ユーザー定義コンストラクタ**

  ユーザー定義コンストラクタ・ファンクションを使用すると、新しく作成したオブジェクト・インスタンスの初期化をカスタマイズすることができます。また、これらのファンクションを使用すると、既存のコード内のコンストラクタへのコールを更新することなく、型を進化させ、新しく追加した属性を保存することができます。

  参照： 6-20 ページの「ユーザー定義コンストラクタ」を参照してください。

Oracle9*i* リリース 1（9.0.1）のオブジェクト・リレーショナルの新機能

- **SQL 型の継承**

  特殊な種類のユーザー定義型は、SQL 型階層内でサブタイプとして定義できます。

  参照： 第 2 章の「型の継承」を参照してください。

- **オブジェクト・ビューの階層**

  型階層内の一部またはすべての型を基にして、オブジェクト・ビューから階層を作成できます。オブジェクト・ビュー階層の存在により、問合せなどの操作で特定のサブタイプを対象にする（これに加え、サブタイプのサブタイプを対象とする場合もあります）処理が簡略化されます。

  参照： 第 5 章の「オブジェクト・ビューの階層」を参照してください。

- **型進化**

  ユーザー定義型は変更可能、つまり進化させることができます。したがって、再作成する必要はありません。

  参照： 第 6 章の「型進化」を参照してください。

- **ユーザー定義集計ファンクション**

  複雑なデータが扱えるように、ユーザー定義集計ファンクションを定義できます。

  参照： 第 6 章の「ユーザー定義集計ファンクション」を参照してください。

- **一般的および一時的なデータ型**

  外部プロシージャに、任意のスカラー型またはユーザー定義型の値を持つ汎用型のフィールドまたはパラメータを指定できます。これにより、同一外部プロシージャの様々なバージョンを実装せずに、複数のデータ型が扱えるようになります。

  参照： 第 6 章の「一時型および汎用型」を参照してください。

- **ファンクション索引**

  タイプ・メソッド・ファンクションを基にしたファンクション索引が作成できます。

  参照： 第 8 章の「タイプ・メソッドの戻り値に基づくファンクション索引」を参照してください。

- **マルチレベル・コレクション**

  コレクション（VARRY およびネストした表）には、それ自身がコレクションである要素またはコレクション属性を持つ要素を入れることができます。

  参照： 第 2 章および第 5 章を参照してください。

- **C++ Interface to Oracle**

  OCI をベースとする C++ Interface（OCCI）to Oracle を使用すると、C++ プログラミング言語のオブジェクト指向機能、ネイティブ・クラスおよびメソッドを使用して、Oracle データベースにアクセスできます。

  参照： 第 3 章の「Oracle C++ Call Interface（OCCI）」を参照してください。

- **Java オブジェクトの記憶域**

  既存の Java クラスにマップする SQL 型を作成して、Java オブジェクトの永続ストレージを確保できます。これらの SQL 型を、Java 言語の SQL 型または SQLJ 型といいます。

  参照： 第 3 章の「Java:JDBC、Oracle SQLJ、JPublisher および SQLJ オブジェクト型」を参照してください。

1

Oracle オブジェクトの概要

この章では、Oracle9*i* オブジェクト・リレーショナル・モデルの主な機能および利点について説明します。内容は次のとおりです。

- Oracle オブジェクトおよびオブジェクト型
- オブジェクトの利点
- オブジェクト・リレーショナル・モデルの主な機能

Oracle オブジェクトおよびオブジェクト型

Oracle オブジェクト型は、複雑な実社会のエンティティをモデル化できるようにするユーザー定義型です。オブジェクト型を利用することで顧客や発注書をデータベース中で 1 つのエンティティ（オブジェクト）として表現できます。

Oracle オブジェクト・テクノロジは、Oracle のリレーショナル・テクノロジを基礎とする抽象化層です。新しいオブジェクト型は、任意の組込みデータベース型、作成済のオブジェクト型、オブジェクト参照およびコレクション型から作成できます。ユーザー定義型のメタデータは、SQL、PL/SQL、Java および他の公開されたインタフェースで使用できるスキーマ内に格納されます。

可変長配列やネストした表などのオブジェクト型および関連オブジェクト指向機能は、データベース内のデータを編成し、アクセスするための高度な手段となります。オブジェクト層の下では、データは従来どおり列や表に格納されますが、このデータは、たとえば顧客や発注書のようにデータに意味を持たせる実社会のエンティティとして扱えます。データベースへ問い合せる場合は、単に顧客を選択するのみです。列と表に置き換えて考える必要はありません。オブジェクトを使用することにより、一部分のみではなく、全体が見えるようになります。

内部的に見ると、オブジェクトに関する文は、従来どおり、基本的にはリレーショナル表および列に関する文です。また、リレーショナル・データ型が扱え、リレーショナル表にデータを格納できます。今回は、これに、オブジェクト指向機能を活用するオプションが加わりました。従来どおり、データの大部分を相関的に扱いながら、オブジェクト指向機能を利用する方向に進むことも、完全にオブジェクト指向アプローチに転向することもできます。たとえば、オブジェクト・データ型をいくつか定義し、リレーショナル表の列にオブジェクトを格納できます。また、既存のリレーショナル・データをもとに**オブジェクト・ビュー**を作成して、オブジェクト・モデルに従ってこのデータを表現しアクセスすることも可能です。または、**オブジェクト表**にオブジェクト・データを格納することも可能です。この場合、それぞれの行が 1 つのオブジェクトになります。

オブジェクトの利点

一般的に、オブジェクト型モデルは、C++ および Java で使用されているクラス・メカニズムに似ています。クラスの場合と同様、オブジェクトにより複雑な実社会のビジネス・エンティティおよび論理が簡単にモデル化できます。また、オブジェクトを再利用できるため、データベース・アプリケーションを短期間で効率的に開発できます。Oracle はデータベースのオブジェクト型をネイティブ・サポートしているため、アプリケーション開発者は自身のアプリケーションで使用するデータ構造に直接アクセスできます。クライアント側オブジェクトと、データが入っているリレーショナル・データベースの列および表との間に、マッピング・レイヤーを記述する必要はありません。また、オブジェクトの抽象化およびカプセル化により、アプリケーションが簡素化されメンテナンスが容易になります。

純粋にリレーショナル・アプローチにより得られるオブジェクトの利点には、次のようなものもあります。

オブジェクトによる操作とデータのカプセル化

データベース表には、データのみが存在します。データで必要になりそうな操作を実行する機能を、オブジェクトに組み込めます。したがって、発注情報オブジェクトに、購入品目すべてのコストを合計するメソッドが組み込まれる場合もあります。または、顧客オブジェクトに、顧客の名前、参照番号、住所、購買履歴および支払パターンを戻すメソッドを持たせることもできます。アプリケーションは、メソッドをコールするのみで、情報を取り出せます。

オブジェクトによる効果

オブジェクト型を使用することで、次のような効果があります。

- オブジェクト型およびオブジェクト・メソッドは、データと一緒にデータベースに格納されるため、どのアプリケーションでも利用できるようになります。すでに完了している開発作業を活用することができ、それぞれのアプリケーションで同様の構造を再作成する必要がありません。
- 一連の関連オブジェクトを1つの単位としてフェッチし、処理できます。サーバーからオブジェクトをフェッチする要求を1回出すのみで、オブジェクト参照により関連付けられた他のオブジェクトが取り出せます。たとえば、クライアントとサーバー間の1回のラウンドトリップで、顧客オブジェクトを選択し、顧客の名前、電話番号および住所情報が取り出せます。

オブジェクトによる部分対全体の関連の表現

リレーショナル・システムでは、複雑な部分対全体の関連が適正に表現できません。たとえば、ピストンとエンジンは、在庫品目の表の中で同じステータスが与えられてしまいます。エンジンのパーツとしてピストンを表現するには、主キーと外部キーの関連を使用して、複数の表で構成されるスキーマを作成する必要があります。これに対し、オブジェクト型には部分対全体の関連を記述する様々な表現方法があります。オブジェクトは他のオブジェクトを**属性**として持ち、属性オブジェクトはオブジェクト属性も持てます。オブジェクト型を結合することにより、この方法でパーツリスト全体の階層を作成できます。

有機体としてのオブジェクト

オブジェクト型を使用すると、エンティティの「客観的実在性」、すなわち一部分が集まって全体を構成するという事実を獲得できます。たとえば、住所には、番地、通り、市、州および郵便番号を入れる必要があります。これらの要素のどれかが欠けると、住所が不完全なものになります。住所オブジェクト型とは異なり、リレーショナル表は表内の列を1つの有機体として表現することはできません。

オブジェクト・リレーショナル・モデルの主な機能

Oracle では、オブジェクト型システムはリレーショナル・モデルの拡張機能として実装されます。オブジェクト型インタフェースでは、問合せ（SELECT…FROM…WHERE）、高速コミット、バックアップおよびリカバリ、スケーラブルな接続性、行レベルのロック、読込み一貫性、パーティション表、パラレル問合せ、クラスタ・データベース、エクスポート / インポート、ローダーなど、標準のリレーショナル・データベース機能が引き続きサポートされます。ただし、SQL および Oracle と接続する各種のプログラム・インタフェース（PL/SQL、Java、OCI、Pro*C/C++、OO4O など）には、オブジェクトをサポートするための拡張機能が追加されています。その結果、オブジェクト・リレーショナル・モデルが誕生しました。このモデルによって、リレーショナル・データベースの優れた並行性とスループットを保つと同時に、オブジェクト・インタフェースの直観性と経済性が実現します。

型の継承

型の継承では、特化されたサブタイプの連続レベルを定義することで、型階層が作成できるようになっています。これにより、オブジェクトの有用性が増大します。これらのサブタイプは、共通する親のオブジェクト型から導出します。導出サブタイプは、親オブジェクト型の機能を継承すると同時に、親の型定義を拡張されます。たとえば、一般的な顧客オブジェクト型から、顧客の特殊な型である企業顧客型または官庁顧客型を導出できます。特殊な型を使用することで、新しい属性の追加や親から継承するメソッドの再定義ができます。その結果発生する型階層により、アプリケーション・モデルの複雑さを管理するために必要な高度な抽象化レベルが得られます。

型進化

ALTER TYPE 文を使用して、既存のユーザー定義型を修正、つまり進化させて、次のような変更を加えることができます。

- 属性の追加および削除
- メソッドの追加および削除
- 長さ、精度または位取りを引き上げるための数値属性の修正
- 文字長を長くするための可変長文字属性の修正
- 型の FINAL および INSTANTIABLE プロパティの変更

変更する型の依存関係は、CREATE TYPE 文に適用するのと同じ妥当性チェックを使用して調べます。型またはその型に依存する型のいずれかが、妥当性チェックで不合格になると、ALTER TYPE 文がロールバックします。その結果、新しい型のバージョンは作成されず、依存スキーマ・オブジェクトは前の状態のままになります。

変更された型を使用するすべての表および列のメタデータは、新しい型定義に対応して更新されます。したがって、データは新しい形式で表や列に格納されます。既存データを新しい形式に変換する処理は、データ更新時に一括または個別に行えます。いずれの場合も、データが古い型定義の形式のまま格納されていても、データは、必ず新しい型定義の形式で表示されます。

オブジェクト・ビュー

Oracle は、ネイティブ機能としてサーバーにオブジェクト・データを格納するのみでなく、オブジェクト・ビュー・メカニズムを介して、既存のリレーショナル・データを対象としたオブジェクトの抽象化を行います。オブジェクト・ビューに属するオブジェクトには、オブジェクト表の行オブジェクトにアクセスする場合と同じ方法でアクセスすることができます。Oracle は、リレーショナル・スキーマおよび表に格納されたデータから、ユーザー定義型のオブジェクトを生成します。オブジェクト・ビューを使用することで、既存のリレーショナル・データベース・スキーマを変更することなく、オブジェクト指向アプリケーションを開発できます。

また、オブジェクト・ビューにより、型階層でもたらされるポリモフィズム（多相性）が活用できるようになります。ポリモフィックな式は、その式の宣言された型またはその型のサブタイプの値をとることができます。ある型階層の構造の一部または全部をミラー化するオブジェクト・ビューの階層を作成すると、その階層内のすべてのビューに対して問合せを行い、関心のある特化レベルでデータにアクセスできるようになります。サブビューを持つオブジェクト・ビューの問合せを行う場合は、ポリモフィック・データ、すなわちビューの型とそのサブタイプの両方の行を取得できます。

SQL のオブジェクトの拡張

新しいオブジェクト関連機能をサポートするために、SQL 拡張機能（新しいデータ定義言語（DDL）を含む）が追加されました。この機能は、オブジェクト型の作成、変更または削除、オブジェクトの表への格納、オブジェクト・ビューの作成、変更または削除を行うためのものです。また、オブジェクト型、参照およびコレクションをサポートするための、SQL データ操作言語（DML）と問合せ拡張機能があります。

PL/SQL のオブジェクト拡張

PL/SQL は、SQL と密接に統合した Oracle のデータベース・プログラミング言語です。Oracle8*i* で導入されているユーザー定義型および他の SQL 型に加えて、ユーザー定義型をシームレスに操作するために、PL/SQL が拡張されています。したがって、アプリケーション開発者は、PL/SQL を使用して、データベース・サーバー内で実行するユーザー定義型に対してロジックや操作を実装できます。

Oracle オブジェクトに対する Java のサポート

Oracle JVM は、RDBMS と密接に統合されており、動的 SQL を提供する JDBC および静的 SQL を提供する SQLJ へのオブジェクト拡張機能を介して行う Oracle オブジェクトへのアクセスをサポートします。したがって、アプリケーション開発者は、Java を使用して、データベース・サーバー内で実行するユーザー定義型に対してロジックや操作を実装できます。Oracle9*i* では、既存の Java クラスにマップする SQL 型を作成して、Java オブジェクトに永続ストレージを持たせることができます。

外部プロシージャ

オブジェクト型のデータベース・ファンクション、プロシージャまたはメンバー・メソッドは、PL/SQL、Java または C で、外部プロシージャとして実装できます。外部プロシージャは、マシン精度の計算においてより効率的な、C などの低レベル言語でより高速で簡単にできる作業に最適です。外部プロシージャは、必ず RDBMS サーバーのアドレス空間外で、セーフ・モードで実行されます。複数のパラメータをシステム定義汎用型として宣言する共通の外部プロシージャが書き込めます。汎用型では、この型を使用して任意の組込み型またはユーザー定義型のデータを扱うプロシージャが認められます。

Object Type Translator（OTT）

OTT は、Oracle データ・ディクショナリからのスキーマ情報を使用して、Java クラス、C 構造体およびインジケータが入っているヘッダー・ファイルを生成することによって、クライアント側でオブジェクト型スキーマへのマッピングを行います。生成されたこれらのヘッダー・ファイルは、ホスト言語アプリケーションで、データベース・オブジェクトへ透過的にアクセスするために使用されます。

クライアント側キャッシュ

Oracle には、データベースに格納された永続オブジェクトに、効率的にアクセスするためのオブジェクト・キャッシュがあります。オブジェクトのコピーは、オブジェクト・キャッシュに置かれます。一度データがクライアントにキャッシュされると、アプリケーションは、これらのデータに対してメモリー・スピードでアクセスできます。キャッシュ内で行われたオブジェクトの変更は、OCI のオブジェクト拡張機能を使用して、データベースに反映できます。

OCI のオブジェクト拡張機能

OCI は、Oracle のオブジェクト機能を使用するアプリケーション開発者およびツール開発者にとって、包括的なアプリケーション・プログラミング・インタフェースです。OCI は、Oracle サーバーに接続する機能と、サーバー内のオブジェクトにアクセスするトランザクションを制御する機能を備えたランタイム環境となります。この OCI により、アプリケーション開発者は、クライアント側のオブジェクト・キャッシュ内のオブジェクトおよびそれらの属性にアクセスして操作できます。このような作業は、相互に関連付けられたオブジェクトをナビゲートしながら行うか、または宣言 SQL DML でデータの性質を指定して、結合的に行います。OCI には、サーバー内で定義されたオブジェクト型に関するメタデータ情報に実行時にアクセスできるようにする様々な機能が用意されています。このような一連の機能によって、データベースに格納されたオブジェクト・メタデータおよび実際のオブジェクト・データへの動的アクセスが容易になります。

Pro*C/C++ のオブジェクト拡張

Oracle Pro*C プリコンパイラは、埋込み SQL アプリケーション・プログラミング・インタフェースを備え、OCI より高レベルの抽象化を行います。OCI と同様、Pro*C プリコンパイラでも、アプリケーション開発で、Oracle のクライアント側のオブジェクト・キャッシュおよび OTT ユーティリティが使用できます。Pro*C は、Oracle9*i* オブジェクト型を対象とした C バインド変数の使用をサポートしています。また、Pro*C には、単純化された新しい構文が組み込まれています。これを使用して SQL 型オブジェクトの割当てや解放を行ったり、DML またはナビゲーショナルなインタフェースを介してこれらのオブジェクトにアクセスできます。このように、アプリケーション開発者は Pro*C の様々な利点を活用できます。たとえば、コンパイル時に（クライアント側の）バインド変数をサーバーのスキーマと照合してタイプ・チェックを行う、Oracle9*i* サーバーのオブジェクト・データを自動マッピングしてクライアント側のバインド変数をプログラムする、クライアント・プロセス内のデータベース・オブジェクトを簡単に管理する、などの操作手段が得られます。

OO4O のオブジェクト拡張

OO4O は、一連の COM オートメーション・インタフェース / オブジェクトで、Oracle9*i* データベース・サーバーへの接続、問合せの実行および結果の管理を行います。OO4O のオートメーション・インタフェースは、Oracle9*i* の機能に簡単かつ効率的にアクセスする手段となります。このインタフェースは、Microsoft COM オートメーション・テクノロジをサポートするほとんどのプログラミング言語またはスクリプティング言語で使用できます。これらの言語としては、Visual Basic、Visual C++、Excel の VBA、VBScript および IIS Active Server Pages の JavaScript があります。

2

Oracle オブジェクトの基本コンポーネント

この章では、オブジェクトを扱ううえでの基本的な情報について説明します。ここでは、オブジェクト型、メソッドおよびコレクションについて説明する他、共有ルート型から導出し、継承によって結合するオブジェクト型の階層を作成する方法およびこの階層を扱う方法についても説明します。

内容は次のとおりです。

オブジェクト・リレーショナル要素

オブジェクト・リレーショナル機能には、新しい概念やリソースが多数導入されています。これらについて、次の項で簡単に説明します。

オブジェクト型

オブジェクト型はある種のデータ型で、NUMBER や VARCHAR2 のような一般的なデータ型の場合と同じ方法で使用できます。たとえば、オブジェクト型をリレーショナル表の列のデータ型として指定し、オブジェクト型の変数を宣言できます。オブジェクト型の値を格納するには、そのオブジェクト型の変数を使用します。オブジェクト型の値は、その型のインスタンスです。オブジェクト・インスタンスは、オブジェクトとも呼ばれます。

リレーショナル・データベースに固有の一般的なデータ型とオブジェクト型の間には、重要な相違点がいくつかあります。

- データベースには一連のオブジェクト型が標準で提供されていないため、使用するオブジェクト型をユーザー自身が定義します。
- オブジェクト型は 1 つの単位ではなく、属性およびメソッドと呼ばれる要素を持っています。

  属性は、対象となるオブジェクトの機能に関するデータを持ちます。たとえば、soldier オブジェクト型に、name、rank および serial number という属性を持たせることができます。属性は宣言されたデータ型を持ち、このデータ型が別のオブジェクト型になることもあります。つまり、オブジェクト・インスタンスの属性は、そのオブジェクトのデータを持ちます。

  メソッドとは、アプリケーションがオブジェクト型の属性に対して有効な操作を実行できるように用意されたプロシージャまたはファンクションです。メソッドはオブジェクト型のオプションの要素で、その型のオブジェクトの動作を定義し、そのオブジェクトの型に何をさせるかを決めます。

- オブジェクト型には、システム固有のデータ型に相当する汎用性はありません。これが、オブジェクト型の重要な長所の 1 つです。オブジェクト型を定義すると、アプリケーション・プログラムで処理する実社会のエンティティ（顧客や発注書など）の実際の構造をモデル化できます。このようにモデル化することにより、エンティティのデータ管理が容易になり、直観的に理解できるようになります。この点で、オブジェクト型は Java や C++ のクラスと似ています。

オブジェクト型は、構造の見取り図またはテンプレートとみなし、オブジェクトはそのテンプレートに基づいて作成された現物と考えることができます。

オブジェクト型はデータベース・スキーマ・オブジェクトで、他のスキーマ・オブジェクトと同様に管理制御されます（第 4 章「Oracle オブジェクトの管理」を参照）。

オブジェクト型は、実社会のオブジェクトの実際の構造をモデル化する場合に使用できます。オブジェクト型を使用することにより、オブジェクトの構造をフラット化して、表と列の 2 次元の純粋なリレーショナル・スキーマに変換せずに、オブジェクトの構造上の内部関連およびオブジェクトの属性を獲得できます。オブジェクト型の場合、関連する個々のデータを、そのデータに定義されている動作と一緒に、1 つの単位として格納できます。これにより、アプリケーション・コードは、これらの単位をオブジェクトとして取り出し、処理します。

型の継承

追加属性や追加メソッドのように特化された追加機能を持つ**サブタイプ**を作成することで、オブジェクト型を**特化**できます。サブタイプは、親オブジェクト型から導出して作成します。この場合、親オブジェクト型は導出サブタイプの**スーパータイプ**と呼ばれます。

サブタイプとスーパータイプは、**継承**により関連付けられます。親の特殊なバージョンとして、サブタイプは、親のすべての属性およびメソッドのみでなく、サブタイプ自体に定義されている特化を持ちます。継承によって関連付けられたサブタイプおよびスーパータイプにより、**型階層**が作成されます。

オブジェクト

オブジェクト型の変数を作成するときに、型のインスタンスを作成します。このインスタンスがオブジェクトです。オブジェクトは、その型に定義された属性およびメソッドを持ちます。オブジェクト・インスタンスは具体的な存在なので、値をその属性に割り当て、そのメソッドをコールできます。

メソッド

メソッドとは、定義された型のオブジェクトに実行させる動作を実装する目的で、オブジェクト型定義に宣言できるファンクションまたはプロシージャのことです。

メソッドは、主にオブジェクトのデータへのアクセスに使用します。アプリケーションがデータに対して実行する可能性のある操作をメソッドとして定義できます。その結果、これらの操作をアプリケーション自身にコーディングする必要がなくなります。アプリケーションは、オブジェクトに対しそれに対応するメソッドをコールして、操作を実行します。

この他、オブジェクト・インスタンスを比較するメソッドや、特定のオブジェクトのデータを使用するかわりに、あるオブジェクト型全体が対象となる操作を実行するメソッドを定義することもできます。

オブジェクト表

オブジェクト表は特別な種類の表であり、各行が 1 つのオブジェクトを表します。

たとえば、次の文は、person オブジェクト型を作成し、person オブジェクトに対するオブジェクト表を定義します。

```
CREATE TYPE person AS OBJECT (
  name        VARCHAR2(30),
  phone       VARCHAR2(20) );

CREATE TABLE person_table OF person;
```

この表は、次の 2 つの方法で表示できます。

- 各行にperson オブジェクトが入っている単一列表として表示します。ここでは、オブジェクト指向操作が実行できます。
- オブジェクト型 person の各属性（name や phone など）が列を占有している複数列表として表示します。ここでは、リレーショナル操作を実行できます。

たとえば、次の指示を実行できます。

```
INSERT INTO person_table VALUES (
       "John Smith",
       "1-800-555-1212" );

SELECT VALUE(p) FROM person_table p
        WHERE p.name = "John Smith";
```

最初の文は、person オブジェクトを person_table に挿入し、person_table を複数列表として扱います。次の文は、person_table から値を単一列表として選択します。ここでは、VALUE ファンクションを使用して、オブジェクト・インスタンスとして行が戻されます。

> **参照：** VALUE ファンクションの詳細は、2-47 ページの「VALUE」を参照してください。

行オブジェクトおよび列オブジェクト

オブジェクト表の中で、完全な行を占めるオブジェクトを行オブジェクトといいます。サイズの大きい行の中の表列を占有するオブジェクト、または他のオブジェクトの属性であるオブジェクトを、**列オブジェクト**といいます。

オブジェクト・ビュー

オブジェクト・ビュー（第 5 章「オブジェクト・モデルのリレーショナル・データへの適用」を参照）は、オブジェクト・リレーショナル機能を使用して関連するデータにアクセスする方法です。オブジェクト・ビューを使用すると、基礎となるリレーショナル・スキーマを変更せずに、オブジェクト指向アプリケーションを開発できます。

REF データ型

REF は、行オブジェクトを指す論理的なポインタで、Oracle の組込みデータ型です。REF および REF のコレクションは、オブジェクト間の対応、特に、多対 1 リレーションシップをモデル化します。したがって、外部キーの必要性が削減されます。REF は、オブジェクト間をナビゲートする簡単なメカニズムを提供します。ポインタに続いてドット表記法を使用することができます。必要に応じて、ユーザーにかわって Oracle が結合します。必要でなければ結合を解除することも可能です。

REF は、REF が参照するオブジェクトの検査または更新に使用できます。REF は、これ自身が参照するオブジェクトのコピーを取得する目的にも使用できます。同じオブジェクト型の異なるオブジェクトを指すように REF を変更したり、REF に NULL 値を割り当てることも可能です。

有効範囲付き REF

列型、コレクション要素またはオブジェクト型属性を REF として宣言する場合、指定されたオブジェクト表の参照のみを含むように、制約を付けることができます。このような REF を、**有効範囲付き** REF といいます。有効範囲付き REF では、有効範囲指定なしの REF に比べて、少ない記憶域でより効率的なアクセスが可能です。

次の例は、有効範囲が address_objtyp のオブジェクト表に制限されている REF 列の address_ref を示しています。

```
CREATE TABLE people (
  id             NUMBER(4)
  name_obj       name_objtyp,
  address_ref    REF address_objtyp SCOPE IS address_objtab,
  phones_ntab    phone_ntabtyp)
  NESTED TABLE   phones_ntab STORE AS phone_store_ntab2 ;
```

REF の有効範囲は、宣言された型のオブジェクト表（この例では、address_objtyp）または宣言された型の任意のサブタイプのオブジェクト表に制限できます。有効範囲をサブタイプのオブジェクト表に制限すると、表内のサブタイプ（およびそのサブタイプ）のインスタンスの参照のみを含むように、REF 列が効果的に制限されます。

サブタイプは型の継承の 1 つの機能です。

参照： 2-31 ページの「型の継承」を参照してください。

参照先がない REF

オブジェクトの削除または権限の変更によって、REF が識別したオブジェクトを使用禁止にすることも可能です。このような REF を、参照先がない REF といいます。Oracle の SQL には、REF がこの条件を満たしているかどうか検証できる述語（IS DANGLING）があります。

REF の参照解除

REF が参照するオブジェクトにアクセスすることを、REF を参照解除するといいます。Oracle には、この参照解除を行う DEREF 演算子があります。

参照先がない REF を参照解除すると、NULL オブジェクトを戻します。

また、Oracle には、REF の暗黙の参照解除を行う機能もあります。次に例を示します。

```
CREATE TYPE person AS OBJECT (
  name    VARCHAR2(30),
  manager REF person );
```

X が PERSON 型のオブジェクトを表すとすると、次の SQL 式は、

```
x.manager.name;
```

ポインタを X という人から X の上司に移し、上司の名前を取り出します（SQL では、このように REF に続けることができますが、PL/SQL ではできません）。

REF の取得

オブジェクト表からオブジェクトを選択して REF 演算子を適用することで、その行オブジェクトに対する REF が取得できます。たとえば、次のように識別番号 1000376 を持つ発注書に対する REF を取得できます。

```
DECLARE OrderRef REF to purchase_order;

SELECT REF(po) INTO OrderRef
               FROM purchase_order_table po
               WHERE po.id = 1000376;
```

問合せは確実に 1 行を戻す必要があります。

参照： 8-9 ページの「REF の記憶域サイズ」を参照してください。

コレクション

1 対多リレーションシップをモデル化するために、Oracle は 2 つのコレクション・データ型、VARRAY およびネストした表をサポートしています。コレクション型は、他のデータ型が使用できるところであれば、どこでも使用できます。また、コレクション型のオブジェクト属性や、コレクション型の列などが使用できます。たとえば、任意の発注書についての明細項目のコレクションを格納するには、発注書オブジェクト型にネストした表属性を持たせることができます。

参照： 2-20 ページの「コレクション」を参照してください。

オブジェクトおよびコレクション型の定義

オブジェクト型およびコレクション型を定義するには、CREATE TYPE 文を使用します。

次に示す CREATE TYPE 文は、オブジェクト型 person、lineitem、lineitem_table および purchase_order を定義しています。lineitem_table は、ネストした表型というコレクション型です。purchase_order オブジェクト型は、この型の lineitems 属性を持ちます。このネストした表の各行は、lineitem 型のオブジェクトです。

CREATE TYPE 文中の字下げされている name、phone、item_name などの要素は属性で、それぞれが宣言されたデータ型を持ちます。

```
CREATE TYPE person AS OBJECT (
  name        VARCHAR2(30),
  phone       VARCHAR2(20) );

CREATE TYPE lineitem AS OBJECT (
  item_name   VARCHAR2(30),
  quantity    NUMBER,
  unit_price  NUMBER(12,2) );

CREATE TYPE lineitem_table AS TABLE OF lineitem;

CREATE TYPE purchase_order AS OBJECT (
  id          NUMBER,
  contact     person,
  lineitems   lineitem_table,

  MEMBER FUNCTION
  get_value   RETURN NUMBER );
```

これは、簡略化された例で、get_value メソッドの本体の宣言は示されていません。get_value メソッドの本体は、CREATE OR REPLACE TYPE BODY 文を使用して指定します。

オブジェクト型を定義しても、記憶域は割り当てられません。

オブジェクト型を型として定義すると、SQL 文の中で lineitem、person および purchase_order が使用できるようになります。これらのオブジェクト型は、NUMBER や VARCHAR2 のような型が使用できる場所のほとんどで使用できます。

たとえば、連絡先を記録するリレーショナル表を定義できます。

```
CREATE TABLE contacts (
  contact     person
  date        DATE );
```

CONTACTS 表は、オブジェクト型を列の 1 つのデータ型として持つリレーショナル表です。リレーショナル表の列を占有するオブジェクトを、列オブジェクトといいます（2-4 ページの「行オブジェクトおよび列オブジェクト」を参照）。

オブジェクト型および参照

この項では、次のオブジェクト型および参照について説明します。

- NULL オブジェクトおよび属性
- オブジェクトおよびコレクションのデフォルト値
- オブジェクト表の制約
- オブジェクト表およびネストした表の索引
- オブジェクト表のトリガー
- REF 型の列および属性のルール
- 名前解決

NULL オブジェクトおよび属性

NULL に初期化されているか、または初期化されていない表列、オブジェクト、オブジェクト属性、コレクションまたはコレクション要素の値は、NULL です。通常、NULL 値は、後で実際の値で置き換えられます。

自分自身の値が NULL であるオブジェクトを、**アトミック NULL** といいます。アトミック NULL オブジェクトは、すべての属性の値が偶然に NULL 値になっているオブジェクトとは異なります。オブジェクトのすべての属性が NULL の場合でも、属性の変更、オブジェクトのメソッドをコールできます。ただし、アトミック NULL オブジェクトの場合は、これらの処理はできません。

たとえば、次のように定義された CONTACTS 表について考えてみます。

```
CREATE TYPE person AS OBJECT (
  name        VARCHAR2(30),
```

```
  phone       VARCHAR2(20) );

CREATE TABLE contacts (
  contact     person
  date        DATE );
```

次の文は、

```
INSERT INTO contacts VALUES (
  person (NULL, NULL),
 '24 Jun 1997' );
```

次の文とは異なる結果を戻します。

```
INSERT INTO contacts VALUES (
  NULL,
 '24 Jun 1997' );
```

どちらの場合も、Oracle は CONTACTS に新しい行用の領域を割り当て、DATE 列を指定された値に設定します。ただし、最初の文は、オブジェクト用の領域を PERSON 列に割り当て、その各属性に NULL を設定します。2 番目の文は、PERSON フィールドそのものを NULL に設定しますが、オブジェクト用の領域は割り当てません。

NULL 値のチェックは省略できる場合があります。リレーショナル表の行または行オブジェクト自体は、NULL に初期化できません。オブジェクトのネストした表に、アトミック NULL の要素を格納することはできません。

ネストした表または配列は NULL になり得るため、その状態を処理する必要があります。NULL コレクションと何も要素を持たない空のコレクションは異なります。

オブジェクトおよびコレクションのデフォルト値

リレーショナル表の列にオブジェクト型またはコレクション型を宣言した場合、DEFAULT 句を含めることができます。明示的に列に値を指定しない場合には、この DEFAULT 句で指定した値が使用されます。DEFAULT 句には、そのオブジェクトまたはコレクションのコンストラクタ・メソッドのリテラル起動を含める必要があります。

コンストラクタ・メソッドのリテラル起動とは、すべての引数がリテラルか、またはさらに続くコンストラクタ・メソッドのリテラル起動のいずれかをとるコンストラクタ・メソッドのコールのことです。変数またはファンクションは使用できません。

次に例を示します。

```
CREATE TYPE person AS OBJECT (
  id          NUMBER
  name        VARCHAR2(30),
  address     VARCHAR2(30) );

CREATE TYPE people AS TABLE OF person;
```

次に、ネストした表型である PEOPLE のコンストラクタ・メソッドのリテラル起動を示します。

```
people ( person(1, 'John Smith', '5 Cherry Lane'),
         person(2, 'Diane Smith', NULL) )
```

次の例は、コンストラクタ・メソッドのリテラル起動を使用してデフォルトを指定します。

```
CREATE TABLE department (
  d_no    CHAR(5) PRIMARY KEY,
  d_name  CHAR(20),
  d_mgr   person DEFAULT person(1,'John Doe',NULL),
  d_emps  people DEFAULT people() )
  NESTED TABLE d_emps STORE AS d_emps_tab;
```

PEOPLE( ) は、空の PEOPLE のコンストラクタ・メソッドのリテラル起動であることに注意してください。

オブジェクト表の制約

他の表と同様に、オブジェクト表にも制約を定義できます。

参照範囲が制限されていない REF 以外の列オブジェクトのリーフ・レベル・スカラー属性に対して、制約を定義できます。

次に例を示します。

最初の例は、オブジェクト表 PERSON_EXTENT の SSNO 列に主キー制約を定義します。

```
CREATE TYPE location (
  building_no NUMBER,
  city        VARCHAR2(40) );

CREATE TYPE person (
  ssno        NUMBER,
  name        VARCHAR2(100),
  address     VARCHAR2(100),
  office      location );

CREATE TABLE person_extent OF person (
  ssno        PRIMARY KEY );
```

次の例の DEPARTMENT 表には、前述の例で定義されたオブジェクト型 LOCATION を持つ列が存在します。この例は、表の DEPT_LOC 列に表示される LOCATION オブジェクトのスカラー属性に制約を定義します。

```
CREATE TABLE department (
  deptno      CHAR(5) PRIMARY KEY,
  dept_name   CHAR(20),
```

```
  dept_mgr    person,
  dept_loc    location,
  CONSTRAINT  dept_loc_cons1
      UNIQUE (dept_loc.building_no, dept_loc.city),
  CONSTRAINT  dept_loc_cons2
       CHECK (dept_loc.city IS NOT NULL) );
```

オブジェクト表およびネストした表の索引

オブジェクト表およびネストした表の記憶表の列や属性についても、他の表と同様に、索引を定義できます。

次の例で示すように、列オブジェクトのリーフ・レベル・スカラー属性に索引を定義できます。有効範囲付き REF の場合、REF 型の列または属性にのみ索引を定義できます。

ここで、DEPT_ADDR は列オブジェクトで、CITY は索引を作成する DEPT_ADDR のリーフ・レベル・スカラー属性です。

```
CREATE TABLE department (
  deptno      CHAR(5) PRIMARY KEY,
  dept_name   CHAR(20),
  dept_addr   address );

CREATE INDEX  i_dept_addr1
          ON  department (dept_addr.city);
```

Oracle が索引定義で列名を指定するところには、オブジェクト列のスカラー属性も指定できます。

オブジェクト表のトリガー

オブジェクト表には、他の表と同様に、トリガーを定義できます。ネストした表の記憶表の列または属性に、トリガーは指定できません。

トリガー本体で LOB 値の変更はできません。トリガーをオブジェクト型とともに使用する場合の制限は他にはありません。

次の例は、前述の項で定義した PERSON_EXTENT 表にトリガーを定義します。

```
CREATE TABLE movement (
     ssno        NUMBER,
     old_office  location,
     new_office  location );

CREATE TRIGGER trig1
  BEFORE UPDATE
            OF  office
            ON  person_extent
```

```
FOR EACH ROW
        WHEN  new.office.city = 'REDWOOD SHORES'
BEGIN
  IF :new.office.building_no = 600 THEN
   INSERT INTO movement (ssno, old_office, new_office)
    VALUES (:old.ssno, :old.office, :new.office);
  END IF;
END;
```

REF 型の列および属性のルール

Oracle では、SCOPE 句または参照制約句を使用して、REF 型の列または属性を制約なしまたは制約付きにすることが可能です。制約なしの REF 列には、対応するオブジェクト型のオブジェクト表に含まれている行オブジェクトに対するオブジェクト参照を格納できます。

Oracle は、そのような列に格納されたオブジェクト参照が、有効な既存の行オブジェクトを指すことを保証しません。したがって、REF 列に、既存の行オブジェクトを指していないオブジェクト参照が含まれることがあります。このような REF 値は、参照先がない参照と呼ばれます。現在、Oracle では、制約なしの REF 列に主キー・ベースのオブジェクト識別子（OID）を含むオブジェクト参照を格納することは認めていません。

REF 列には、参照範囲を特定のオブジェクト表に制限するような制約も指定できます。SCOPE 制約が付いた列に格納されたすべての REF 値は、SCOPE 句で指定された表の行オブジェクトを指します。ただし、REF 値には参照先がない場合があります。

REF 列には、外部キーの指定に類似した参照制約を指定することもできます。参照制約のルールは、そのような列に適用されます。つまり、このような列に格納されたオブジェクト参照は、指定されたオブジェクト表内の有効な既存の行オブジェクトを指す必要があります。

REF 列には、一意制約または主キー制約を指定できません。ただし、NOT NULL 制約は指定できます。

名前解決

Oracle SQL では、いくつかのリレーショナル操作で表名を省略できます。たとえば、ASSIGNMENT が PROJECTS の列で、TASK が DEPTS の列の場合、次のように記述できます。

```
SELECT *
FROM projects
WHERE EXISTS
  (SELECT * FROM  depts
           WHERE assignment = task);
```

Oracle は、各列がどの表に属しているかを判断します。

簡単にメンテナンスできるように、ドット表記法を使用して、表名または表別名で列名を修飾することもできます。

```
SELECT * FROM projects WHERE EXISTS
  (SELECT * FROM  depts WHERE projects.assignment = depts.task);

SELECT * FROM projects pj WHERE EXISTS
  (SELECT * FROM  depts dp WHERE pj.assignment = dp.task);
```

オブジェクト・リレーショナル機能で表別名を指定する必要がある場合もあります。

表別名が必要な場合

修飾されていない名前を使用すると、問題が発生する場合があります。DEPTS に ASSIGNMENT 列を追加した後、問合せの変更をしなかった場合、内側の SELECT が DEPTS 表の ASSIGNMENT 列を使用するように、問合せが自動的に再コンパイルされます。この状況を内部取得といいます。

内部取得および参照解決で発生する同様の問題を回避するために、表別名を使用して、メソッドまたはオブジェクトの属性へのドット表記による参照を修飾してください。ドット表記法を使用せずに、オブジェクト表の最上位属性を参照する場合は、表別名は任意で使用してください。

次の文は、オブジェクト型 PERSON と 2 つの表を定義します。ptab1 は、PERSON 型のオブジェクトのオブジェクト表で、ptab2 はオブジェクト型の列が格納されているリレーショナル表です。

```
CREATE TYPE person AS OBJECT (ssno VARCHAR(20));
CREATE TABLE ptab1 OF person;
CREATE TABLE ptab2 (c1 person);
```

次の問合せには、ssno 属性を参照する正しい方法と、誤った方法が示されています。

```
SELECT            ssno FROM ptab1     ;  --Correct
SELECT         c1.ssno FROM ptab2     ;  --Illegal
SELECT   ptab2.c1.ssno FROM ptab2     ;  --Illegal
SELECT       p.c1.ssno FROM ptab2 p   ;  --Correct
```

- 最初の SELECT 文にある ssno は ptab1 の列の名前で、ドット表記法を使用せずに、最上位属性を直接参照します。したがって、表別名は不要です。
- 2 番目の SELECT 文の ssno は、c1 という名前の列にある PERSON オブジェクトの属性の名前です。この参照はドット表記法を使用しているので、4 番目の SELECT 文に示すように、表別名が必要です。
- 3 番目の SELECT 文は、表名そのものを使用して、この参照を修飾していますが、これは不適切なので、表別名を使用する必要があります。

オブジェクト属性またはメソッドの参照は、表名がスキーマ名によって修飾される場合でも、表名ではなく表別名を使用して修飾する必要があります。

たとえば、次の式は scott スキーマ、projects 表、assignment 列およびこの列の duedate 属性を参照しますが、projects は表名で表別名ではないため、正しい修飾ではありません。

```
scott.projects.assignment.duedate
```

REF を使用する属性参照にも同じ要件が適用されます。

表別名は、問合せ全体を通して同じ表を抽出するように要求されます。また、問合せで表示されるスキーマ名と異なる名前である必要があります。

> **注意：** 列にオブジェクト型が含まれているかどうかにかかわらず、すべての UPDATE 文、DELETE 文、SELECT 文および副問合せに表別名を定義し、それらを使用して列参照を修飾することをお薦めします。

リモート・データベースでユーザー定義型を使用する際の制限事項

現在、ユーザー定義型（特に、PL/SQL パッケージ内部で宣言する型とは対照的な、SQL CREATE TYPE 文を使用して宣言する型）は、1 つのデータベース内部のみで有効です。次のいずれの用途にも、データベース・リンクを使用することはできません。

- リモート表上のユーザー定義型またはオブジェクト REF の問合せ、挿入または更新を行うためにリモート・データベースと接続する。
- リモート・ユーザー定義型のローカル変数を宣言するために、PL/SQL コード内部でデータベース・リンクを使用する。
- PL/SQL リモート・プロシージャ・コールでユーザー定義型引数を送るか、または値を戻す。

メソッド

メソッドとは、定義された型のオブジェクトに実行させる動作を実装する目的で、オブジェクト型定義に宣言できるファンクションまたはプロシージャのことです。アプリケーションは、メソッドをコールし、動作を起動します。

たとえば、発注情報オブジェクトを取得し、その明細項目の合計コストを戻す場合にメソッド get_sum() を宣言できます。次のコード行は、発注書 po のメソッドをコールし、金額を sum_line_items の中に入れて戻します。

```
sum_line_items = po.get_sum();
```

カッコは必ず付けます。PL/SQL ファンクションやプロシージャとは異なり、Oracle では、引数を持たないメソッドのコールを含むすべてのメソッド・コールにカッコを付ける必要があります。

メソッドは、PL/SQL または他のほとんどすべてのプログラミング言語で書くことができます。PL/SQL または Java で書かれたメソッドは、データベースに格納されます。C のような他の言語で書かれたメソッドは、外部に格納されます。

型定義には、2 種類のメソッドを宣言できます。

- メンバー
- スタティック

この他にも、**コンストラクタ・メソッド**と呼ばれるもう 1 種類のメソッドがあります。これは、システムがすべてのオブジェクト型について定義するメソッドです。型のオブジェクト・インスタンスを作成するときに、その型のコンストラクタ・メソッドをコールします。

メンバー・メソッド

メンバー・メソッドは、アプリケーションがオブジェクト・インスタンスのデータにアクセスするための手段となります。オブジェクト型のオブジェクトに実行させるそれぞれの操作について、メンバー・メソッドをオブジェクト型に定義します。たとえば、発注書の明細項目の合計コストを計算する get_sum() メソッドは、特定の発注書のデータを操作するメンバー・メソッドです。

メンバー・メソッドは、現在メソッドが起動されているオブジェクト・インスタンスを示す SELF と呼ばれる組込みパラメータを持っています。メンバー・メソッドは、修飾子の付いていない SELF の属性およびメソッドを参照できます。これにより、メンバー・メソッドの書込みが簡素化されます。次に、SELF を活用して num および den の修飾を省略するメソッド宣言を示します。

```
CREATE TYPE Rational AS OBJECT (
  num INTEGER,
  den INTEGER,
  MEMBER PROCEDURE normalize,
  ...
);

CREATE TYPE BODY Rational AS
  MEMBER PROCEDURE normalize IS
    g INTEGER;
  BEGIN
    g := gcd(SELF.num, SELF.den);
    g := gcd(num, den);           -- equivalent to previous line
    num := num / g;
    den := den / g;
  END normalize;
  ...
END;
```

SELF は明示的に宣言できますが、その必要はありません。常に、このパラメータがメソッドに渡される最初のパラメータになります。メンバー・ファンクションに SELF が宣言されていない場合、そのパラメータ・モードはデフォルトで IN に設定されます。メンバー・プロシージャに SELF が宣言されていない場合、そのパラメータ・モードはデフォルトで IN OUT に設定されます。

メンバー・メソッドは、ドット表記法 object_variable.method() を使用して起動します。この表記法は、メソッドの起動対象となる最初のオブジェクトと、コールするメソッドを指定します。パラメータは必須項目のため、すべてカッコ内に指定します。

オブジェクト比較のメソッド

CHAR または REAL のようなスカラー・データ型の値には、事前定義された順序が存在するため、これらの値は比較できます。ただし、各種のデータ型の属性を多数持つ可能性のある customer_typ のようなオブジェクト型には、事前定義された比較軸は存在しません。オブジェクト型の変数を比較し、順序付けをするには、これらの変数を比較する基準を指定する必要があります。

この操作を行うには、**マップ・メソッド**と**オーダー・メソッド**という 2 種類の特殊なメンバー・メソッドが定義できます。

マップ・メソッド

マップ・メソッドは、任意で使用するメソッドで、オブジェクト・インスタンスをスカラー型（DATE、NUMBER、VARCHAR2）の 1 つ、または ANSI SQL 型（CHARACTER や REAL）にマップすることにより、オブジェクトを比較する際の基準を規定します。マップ・メソッドでは、それぞれのオブジェクトのマップ・メソッドを 1 回コールして、そのオブジェクトを比較に使用する軸（たとえば、数字や日付）上のある位置にマップすることで、オブジェクトの順序付けが行えます。順序付けするオブジェクトの数に制限はありません。

作成する側から見ると、マップ・メソッドは MAP キーワードを使用し、かつリストされたデータ型の 1 つを戻すパラメータを持たないメンバー・ファンクションにすぎません。マップ・メソッドが特殊なメソッドになる理由は、オブジェクト型がこのメソッドを定義すると、メソッドが自動的にコールされ、obj_1 > obj_2 のような比較や、DISTINCT、GROUP BY および ORDER BY 句によって暗に示される比較を評価するためです。obj_1 および obj_2 がマップ・メソッド map() を使用して比較できる 2 つのオブジェクト変数であるとすると、次の比較は、

```
obj_1 > obj_2
```

次の比較と同等になります。

```
obj_1.map() > obj_2.map()
```

「>」以外の他のリレーショナル演算子についても、同じことがいえます。

次の例は、四角形オブジェクトを面積で比較する際の基準を規定するマップ・メソッド area() を定義しています。

```
CREATE TYPE Rectangle_typ AS OBJECT (
  len NUMBER,
  wid NUMBER,
  MAP MEMBER FUNCTION area RETURN NUMBER,
  ...
);

CREATE TYPE BODY Rectangle_typ AS
  MAP MEMBER FUNCTION area RETURN NUMBER IS
  BEGIN
     RETURN len * wid;
  END area;
  ...
END;
```

オブジェクト型は1つのマップ・メソッド（または1つのオーダー・メソッド）のみ宣言できます。サブタイプの場合は、そのルート・スーパータイプでマップ・メソッドを宣言している場合にかぎり、マップ・メソッドを宣言できます。

オーダー・メソッド

オーダー・メソッドは、オブジェクト同士を直接比較します。マップ・メソッドとは異なり、オーダー・メソッドはオブジェクトをいくつでも外部軸にマップすることはできません。オーダー・メソッドは、メソッドに使用されている基準と比較して、現在のオブジェクトが比較対象の他のオブジェクトより小さい、等しいまたは大きいかを知らせるのみです。

オーダー・メソッドは、同じ型の別のオブジェクトについて宣言されたパラメータを持つファンクションの1つです。このメソッドは、負の数、0（ゼロ）または正の数のいずれかを戻すように書く必要があります。戻り値により、SELF パラメータによって選択されたオブジェクトが、他のパラメータのオブジェクトより小さい、等しいまたは大きいかが示されます。

マップ・メソッドと同様、オーダー・メソッドを定義した場合、その型を持つ2つのオブジェクトを比較する必要があれば、自動的にコールされます。

オーダー・メソッドは、比較セマンティクスが複雑すぎてマップ・メソッドが使用できない場合に役立ちます。たとえば、イメージのようなバイナリ・オブジェクトを比較する場合、各イメージの明度またはピクセル数によってイメージを比較するオーダー・メソッドが作成されます。

オブジェクト型は1つのオーダー・メソッド（または1つのマップ・メソッド）のみ宣言できます。導出元が別の型ではない型のみ、オーダー・メソッドが宣言できます。サブタイプはオーダー・メソッドを宣言できません。

次に、顧客 ID で顧客を比較するオーダー・メソッドを示します。

```
CREATE TYPE Customer_typ AS OBJECT (
  id   NUMBER,
  name VARCHAR2(20),
```

```
  addr VARCHAR2(30),
  ORDER MEMBER FUNCTION match (c Customer_typ) RETURN INTEGER
);

CREATE TYPE BODY Customer_typ AS
  ORDER MEMBER FUNCTION match (c Customer_typ) RETURN INTEGER IS
  BEGIN
    IF id < c.id THEN
      RETURN -1;               -- any negative number will do
    ELSIF id > c.id THEN
      RETURN 1;                -- any positive number will do
    ELSE
      RETURN 0;
    END IF;
  END;
END;
```

ガイドライン

マップ・メソッドはオブジェクト値をスカラー値にマップします。このメソッドは、スカラー軸上の位置に従った複数の値の順序付けが行えます。オーダー・メソッドは特定の 2 つのオブジェクトの値を直接比較します。

宣言できるのは、マップ・メソッドかオーダー・メソッドのいずれか一方で、両方を宣言することはできません。いずれか一方のメソッドを宣言すると、SQL 文およびプロシージャ文の中のオブジェクトの比較ができます。ただし、どちらのメソッドも宣言しない場合でも、SQL 文の中にあるオブジェクトについてのみ、等価比較または不等価比較が行えます（同じ型の 2 つのオブジェクトは、対応する属性の値が一致する場合のみ、等価とみなされます）。

大量のオブジェクトをソートまたはマージする場合は、マップ・メソッドを使用します。1 回のコールで、すべてのオブジェクトがスカラー値にマップされ、スカラー値がソートされます。オーダー・メソッドは繰り返してコールする必要があるため、効率的ではありません（一度に比較できるのは 2 つのオブジェクトのみです）。

型の階層内の比較メソッド

一般的な型の定義から特殊な型の定義を導出する型階層では、オーダー・メソッドを定義できるのは、ルート型（他のすべての型の導出元となる最も基本的な型）のみです。ルート型がオーダー・メソッドを定義していない場合は、そのサブタイプもこのメソッドを定義できません。

ルート型がマップ・メソッドを指定した場合、そのサブタイプはルート型のマップ・メソッドをオーバーライドするマップ・メソッドを定義できます。ただし、ルート型がマップ・メソッドを定義していない場合、そのサブタイプはマップ・メソッドを定義できません。

したがって、ルート型がマップ・メソッドとオーダー・メソッドのどちらも指定していない場合、そのサブタイプはマップ・メソッドとオーダー・メソッドのどちらも指定できません。

参照： 2-31 ページの「型の継承」を参照してください。

スタティック・メソッド

スタティック・メソッドはオブジェクトのインスタンスではなく、オブジェクト型に対して起動されます。スタティック・メソッドは、型に対してグローバルに適用され、特定のオブジェクトのインスタンスのデータを参照する必要のない操作に使用します。スタティック・メソッドは、SELF パラメータを持ちません。

スタティック・メソッドは、次に示すように、ドット表記法によりメソッド・コールにオブジェクト型の名前を付けて修飾して起動します。type_name.method()

コンストラクタ・メソッド

すべてのオブジェクト型が、システムによって暗黙的に定義された**コンストラクタ・メソッド**を持っています。コンストラクタ・メソッドは、ユーザー定義型の新しいインスタンスを戻し、その属性の値を設定するファンクションです。独自のコンストラクタを明示的に定義することもできます。この項では、コンストラクタ・メソッド（特にシステム定義コンストラクタ）について説明します。

参照： ユーザー定義コンストラクタおよびその利点の詳細は、6-20 ページの「ユーザー定義コンストラクタ」を参照してください。

コンストラクタ・メソッドはファンクションの一種で、新しいオブジェクトを値として戻します。コンストラクタ・メソッドの名前は、オブジェクト型の名前です。このメソッドのパラメータは、オブジェクト型の属性の名前および型です。

たとえば、Customer_typ という型があるとします。

```
CREATE TYPE Customer_typ AS OBJECT (
  id    NUMBER,
  name  VARCHAR2(20),
  phone VARCHAR2(30),
);
```

次の例は、Customer_typ の新しいオブジェクトのインスタンスを作成し、その属性の値を指定し、このオブジェクトに変数を設定します。

```
cust = Customer_typ(103, "Ravi", "1-800-555-1212")
```

次の例にある INSERT 文は、Address_typ オブジェクト型の属性を持つ顧客オブジェクトを挿入します。コンストラクタ・メソッド Address_typ により、カッコの中に示された属性値を持つこの型のオブジェクトが作成されます。

```
INSERT INTO Customer_objtab
  VALUES (
    1, 'Jean Nance',
    Address_typ('2 Avocet Drive', 'Redwood Shores', 'CA', '95054'),
    ...
    ) ;
```

コレクション

Oracle では、VARRAY とネストした表の 2 つのコレクションがサポートされます。

- VARRAY とは、順序付けられた要素の集合です。それぞれの要素の位置には、索引番号があるため、この番号を使用して個々の要素にアクセスします。VARRAY を定義するときは、ここに格納できる要素の最大数を指定しますが、この値は後で変更できます。VARRAY は不透明なオブジェクト（RAW または BLOB）として格納されます。
- ネストした表には、要素をいくつでも入れることができます。表の定義に、最大値の指定はありません。また、要素の順序は維持されません。通常の表を扱う場合と同様に、ネストした表でも選択、挿入、削除などの操作ができます。ネストした表の要素は、独立した記憶表に格納されます。この記憶表には、各要素が属している親表の行またはオブジェクトを示す列が入っています。

一定数の項目のみ格納する必要がある場合、順序に従って要素をループする必要がある場合、またはコレクション全体を 1 つの値として取り出して操作する必要がある場合は、VARRAY を使用してください。

コレクションについての問合せを効率的に実行する必要がある場合は、任意の数の要素を扱うか、大量の挿入 / 更新 / 削除を行ってから、ネストした表を使用してください。

VARRAY

配列とは、順序付けられたデータ要素の集合です。任意の配列のすべての要素は、同じデータ型の要素です。各要素は、索引を持ちますが、これは配列における要素の位置に対応する番号です。

配列における要素の数は、配列の *サイズ*です。Oracle では、配列を可変サイズにできます。この配列を VARRAY といいます。配列型を宣言する場合は、最大サイズを指定する必要があります。

たとえば、次の文は配列型を宣言しています。

```
CREATE TYPE prices AS VARRAY(10) OF NUMBER(12,2);
```

PRICES 型の VARRAY は、最大 10 個の要素を持ち、それぞれのデータ型は NUMBER(12,2) です。

配列型を作成しても、領域は割り当てられず、データ型が定義されます。このデータ型の用途は、次のとおりです。

- リレーショナル表の列のデータ型
- オブジェクト型属性
- PL/SQL 変数、パラメータまたはファンクション戻り値の型

通常、VARRAY はインラインに、つまり、その行にある他のデータと同じ表領域に格納されます。VARRAY のサイズが十分であれば、BLOB として格納されます。

VARRAY に LOB を入れることはできません。したがって、VARRAY には、LOB 属性を持つユーザー定義型の要素も入れることができません。

参照： 8-14 ページの「VARRAY の記憶域上の考慮点」を参照してください。

ネストした表

ネストした表は順序付けされていないデータ要素の集合で、これらの要素はすべて同じデータ型を持ちます。ネストした表は単一列を持ち、この列の型は組込み型またはオブジェクト型です。ネストした表の列がオブジェクト型であれば、オブジェクト型の各属性に対して列を持つ複数列表としても、表を表示できます。

たとえば、発注書の例で、次の文は、明細項目のネストした表に使用する表の型を宣言しています。

```
CREATE TYPE lineitem_table AS TABLE OF lineitem;
```

ネストした表型の定義では、領域は割り当てられず、型が定義されます。この型の用途は、次のとおりです。

- リレーショナル表の列のデータ型
- オブジェクト型属性
- PL/SQL 変数、パラメータまたはファンクション戻り型

リレーショナル表の列がネストした表型の場合、同一記憶表にあるリレーショナル表のすべての行について、ネストした表データが格納されます。ネストした表属性を持つ型のオブジェクト表の場合も同様に、オブジェクト表と対応付けられた 1 つの記憶表にあるオブジェクト・インスタンスすべてについて、ネストした表データが格納されます。

たとえば、次の文は、オブジェクト型 PURCHASE_ORDER に対してオブジェクト表を定義します。

```
CREATE TABLE purchase_order_table OF purchase_order
```

```
  NESTED TABLE lineitems STORE AS lineitems_table;
```

2 行目は、LINEITEMS_TABLE を、PURCHASE_ORDER_TABLE 内のすべての PURCHASE_ORDER オブジェクトの LINEITEMS 属性の記憶表として指定しています。

ネステッド・カーソルを使用すると、ネストした表の個々の要素に簡単にアクセスできます。

> **参照：** ネステッド・カーソルの詳細は、『Oracle9*i* SQL リファレンス』を参照してください。ネストした表の使用については、8-15 ページの「ネストした表」を参照してください。

マルチレベル・コレクション型

マルチレベル・コレクション型とは、要素自体が直接的または間接的に別のコレクション型であるコレクション型です。使用可能なマルチレベル・コレクション型は、次のとおりです。

- ネストした表型のネストした表
- VARRAY 型のネストした表
- ネストした表型の VARRAY
- VARRAY 型の VARRAY
- ネストした表または VARRAY 型である属性を持つユーザー定義型のネストした表または VARRAY

通常のシングルレベル・コレクション型と同様に、マルチレベル・コレクション型もリレーショナル表内の列またはオブジェクト表内のオブジェクト属性と一緒に使用できます。

次の例は、ネストした表のネストした表であるマルチレベル・コレクション型を作成します。この例は、それぞれの星にその周りを回転する惑星のネストした表のコレクションが存在し、それぞれの惑星にその衛星のネストした表のコレクションが存在する天体系をモデル化します。

```
CREATE TYPE satellite_t AS OBJECT (
  name        VARCHAR2(20),
  diameter    NUMBER);

CREATE TYPE nt_sat_t AS TABLE OF satellite_t;

CREATE TYPE planet_t AS OBJECT (
  name        VARCHAR2(20),
  mass        NUMBER,
  satellites  nt_sat_t);

CREATE TYPE nt_pl_t AS TABLE OF planet_t;
```

ネストした表の記憶表

ネストした表型の列またはオブジェクト表属性には、列内にあるネストした表すべての行を格納する記憶表が必要になります。ネストした表のマルチレベル・コレクションの場合と同様、ネストした表の内側のセットも外側のセットと同様に記憶表を必要とします。記憶表を指定するには、ネストした表の記憶域句をもう 1 つ追加します。

次のコードは、マルチレベル・コレクション型の列 planets を格納する stars 表（ネストした表属性 satellites を持つオブジェクト型のネストした表）を作成します。外側のネストした表 planets と内側のネストした表 satellites のそれぞれについて、ネストした表の記憶域句が存在します。

```
CREATE TABLE stars (
  name     VARCHAR2(20),
  age      NUMBER,
  planets  nt_pl_t)
NESTED TABLE planets STORE AS planets_tab
  (NESTED TABLE satellites STORE AS satellites_tab);
```

このネストした表は、オブジェクトの名前付き属性になっているため、この例では内側のネストした表 satellite を名前で参照できます。ただし、内側のネストした表が属性でない場合には、名前はありません。このような場合に備え、キーワード COLUMN_VALUE が用意されています。このキーワードは、内側のネストした表の名前のかわりに使用します。次に例を示します。

```
CREATE TYPE inner_table AS TABLE OF NUMBER;

CREATE TYPE outer_table AS TABLE OF inner_table;

CREATE TABLE tab1 (
  col1 NUMBER,
  col2 outer_table)
NESTED TABLE col2 STORE AS col2_ntab
  (NESTED TABLE COLUMN_VALUE STORE AS cv_ntab);
```

ネストした表句には、記憶表の物理属性を指定できます。次に例を示します。

```
CREATE TABLE stars (
  name     VARCHAR2(20),
  age      NUMBER,
  planets  nt_pl_t)
NESTED TABLE planets STORE AS planets_tab
  ( PRIMARY KEY (NESTED_TABLE_ID, name)
    ORGANIZATION INDEX COMPRESS
    NESTED TABLE satellites STORE AS satellites_tab );
```

ネストした表のどの記憶表も、NESTED_TABLE_ID で参照可能な列を含んでいます。この列は、記憶表内の行を、親表内の関連する行と対応付けます。それ自体がネストした表である

親表には、システムから提供された ID 列が 2 つあります。1 つは、NESTED_TABLE_ID で参照可能な ID 列です。この列は、自身の行をもう一度親表内の行と対応付けます。もう 1 つは、ネストした表の子の中にある NESTED_TABLE_ID 列で参照される非表示列です。

前述の例では、ネストした表 planets に ORGANIZATION INDEX 句を追加し、ネストした表に 1 列目が NESTED_TABLE_ID になっている主キーを割り当て、この表を索引構成表（IOT）にしています。この列には、記憶表の行が関連付けられた親表内の行の ID が入っています。NESTED_TABLE_ID を使用して主キーを 1 列目として指定し、表の索引構成を行うと、同じ親である行に属するネストした表の行すべてが、物理的にクラスタ化され、さらに効率的にアクセスできるようになります。

> **参照：** 8-15 ページの「ネストした表の記憶域」および 9-22 ページの「埋込みオブジェクト付きのオブジェクト表」を参照してください。

ネストした表のそれぞれが、専用の表記憶域句を必要とします。したがって、ネストした表の記憶域句はコレクションに存在するネストした表のレベルと同数用意する必要があります。

VARRAY 記憶域

マルチレベル VARRAY の格納方法は、VARRAY が複数個ある VARRAY の 1 つである場合と、ネストした表の 1 つの VARRAY である場合とでは異なります。

- 複数個ある VARRAY の中の 1 つの VARRAY の場合、そのサイズが大きすぎる場合（4000 バイト前後）、または LOB 記憶域が明示的に指定されている場合を除き、VARRRAY 全体がインラインで（行そのものに）格納されます。
- ネストした表の VARRAY の場合、VARRAY 全体が LOB に格納されます。この場合、LOB ロケータのみが行に格納されます。VARRAY のネストした表要素と関連付けられた記憶表は、存在しません。ネストした表コレクション全体が VARRAY 内部に格納されます。

VARRAY については、LOB 記憶域を明示的に指定できます。次の例は、ネストした表の VARRAY 要素に対して、この指定を行います。例に示すとおり、COLUMN_VALUE キーワードはネストした表のみでなく VARRAY の場合も使用できます。

```
CREATE TYPE va1 AS VARRAY(10) OF NUMBER;

CREATE TYPE nt3 AS TABLE OF va1;

CREATE TABLE tab2 (c1 NUMBER, c2 nt3)
NESTED TABLE c2 STORE AS c2_tab2_nt
  ( VARRAY column_value STORE AS LOB tab2_lob );
```

次に、VARRAY 型の VARRAY について指定された明示的な LOB 記憶域を示します。

```
CREATE TYPE t2 AS OBJECT (a NUMBER, b va1);
```

```
CREATE TYPE va2 AS VARRAY(2) OF t2;

CREATE TABLE tab5 (c1 NUMBER, c2 va2)
VARRAY c2 STORE AS tab5_lob;
```

マルチレベル・コレクションの割当ておよび比較

シングルレベル・コレクションと同様、ソースとターゲットのどちらも、マルチレベル・コレクションの割当てで宣言されている同一のデータ型である必要があります。

データ型がコレクション型（マルチレベル・コレクション型など）の項目は、比較できません。

VARRAY またはネストした表の作成

コレクション型のインスタンスは、他のオブジェクト型のインスタンスを作成する場合と同じ方法、つまり型のコンストラクタ・メソッドをコールして作成します。型のコンストラクタ・メソッドの名前は、型の名前です。コレクションの各要素は、カンマで区切ったメソッドの引数のリストとして指定します。

空のリストでコンストラクタ・メソッドをコールすると、その型の空のコレクションが作成されます。空のコレクションとは、偶然に空になっている実際のコレクションで、NULL コレクションとは異なる点に注意してください。

マルチレベル・コレクションのコンストラクタ

シングルレベル・コレクション型と同様に、マルチレベル・コレクション型はそれぞれの型のコンストラクタ・メソッドをコールして作成します。他のユーザー定義型のコンストラクタ・メソッドと同様に、マルチレベル・コレクション型のコンストラクタは、システム定義ファンクションの 1 つで、型と同じ名前を持ち、かつその型の新しいインスタンス（この場合は、新しいマルチレベル・コレクション）を戻します。コンストラクタ・パラメータは、オブジェクト型の属性の名前と型を持ちます。

次の例は、マルチレベル・コレクション型 nt_pl_t のコンストラクタをコールします。この型は、惑星のネストした表で、それぞれに、衛星のネストした表が属性として入っています。外側のネストした表のコンストラクタは、作成する惑星それぞれに対して、planet_t コンストラクタをコールします。各惑星コンストラクタは、衛星のネストした表型に対してコンストラクタをコールして、衛星のネストした表を作成します。衛星のネストした表型コンストラクタは、作成する衛星インスタンスそれぞれに対して satellite_t コンストラクタをコールします。

```
INSERT INTO stars
VALUES('Sun',23,
  nt_pl_t(
    planet_t(
      'Neptune',
```

```
      10,
      nt_sat_t(
        satellite_t('Proteus',67),
        satellite_t('Triton',82)
      )
    ),
    planet_t(
      'Jupiter',
      189,
      nt_sat_t(
        satellite_t('Callisto',97),
        satellite_t('Ganymede', 22)
      )
    )
  )
);
```

コレクションの問合せ

コレクション型の列または属性が入っている表の問合せ方法には、次の 2 つの一般的な方法があります。1 つは、コレクションが入っている結果行に**ネストされた**コレクションを戻す方法で、もう 1 つは、それぞれのコレクション要素が単独で 1 行に現れるように、コレクションを分散または**ネスト解除**する方法です。

コレクション問合せの結果のネスト化

次の問合せにある projects 列は、projects_list_nt 型のネストした表コレクションです。projects コレクション列は、通常のスカラー列として SELECT 構文のリストに現れます。このような SELECT 構文のリスト内のコレクション列を問い合せると、コレクションと関連付けられた結果行の中のコレクションの各要素がネスト化されます。

たとえば、次の問合せでは、各従業員の名前と、従業員のプロジェクトのコレクションが取得されます。プロジェクトのコレクションは、次のようにネストされています。

```
SELECT e.empname, e.projects
  FROM employees e;

  EMPNAME   PROJECTS
  -------   --------
  'Bob'     PROJECTS_LIST_NT(14, 23, 144)
  'Daphne'  PROJECTS_LIST_NT(14, 35)
```

プロジェクト値またはインスタンスが、ユーザー定義型（id と name という 2 つの属性を持つ Proj_t など）の場合、結果行は次のようになります。

```
EMPNAME   PROJECTS
-------   --------
```

```
'Bob'     PROJECTS_LIST_NT(PROJ_T(14, 'White Horse'), PROJ_T(23, 'Excalibur'), ...)
```

SELECT 構文のリスト内のオブジェクト型列にコレクション属性が入っている場合は、結果もネストされます。SELECT 構文のリストに、このコレクションが明示的にリストされていなくても、結果はネストされます。たとえば、SELECT * FROM employees という問合せを行うと、ネストされた結果が作成されます。

コレクション問合せの結果のネスト解除

すべてのツールやアプリケーションが、ネストされた型式の結果を処理できるとはかぎりません。従来のフォーマットを必要とするツールを使用して Oracle コレクション・データを参照するには、1 つの行のコレクション属性をネスト解除またはフラット化して 1 つ以上のリレーショナル行にする必要があります。そのためには、コレクションとともに TABLE 式を使用します。TABLE 式を使用すると、FROM 句にコレクション指定し、表の場合と同様に問い合せることができます。つまり、ネストした表が入っている行とネストした表を結合する操作をします。

TABLE 式は、変数やパラメータなどの一時値を含むコレクション値式の問合せに使用できます。

> **注意：** TABLE 式は、THE を使用した副問合せ式のかわりとなります。THE 副問合せは、将来的には使用されなくなります。

次の問合せでは、前述の例と同様に、各従業員の名前と従業員が担当するプロジェクトのコレクションが取り出されますが、コレクションのネストは解除されます。

```
SELECT e.empname, p.*
  FROM employees e, TABLE(e.projects) p;

  EMPNAME   PROJECTS
  -------   --------
  'Bob'     14
  'Bob'     23
  'Bob'     144
  'Daphne'  14
  'Daphne'  35
```

前述の例に示すとおり、TABLE 式にも、式自身の表別名を持たせることが可能です。この例では、TABLE 式の表別名は、TABLE 式によって戻される列を選択するために、SELECT 構文のリストの中にあります。

TABLE 式は、別の表別名を使用して、TABLE 式が参照するコレクション列が含まれている表を指定します。したがって、TABLE(e.projects) 式は、projects コレクション列が含まれている表として employees 表を指定します。TABLE 式は、FROM 句の式の左側にあ

る任意の表別名を使用して、その表の列を参照できます。このようなコレクション列の参照を、**左相関**といいます。

次の例では、使用する TABLE 式の表別名を指定するために、FROM 句に employees 表がリストされています。結果として生成される employees 表の列は、TABLE 式によって参照される列にかぎられます。

```
SELECT *
  FROM employees e, TABLE(e.projects);

PROJECTS
--------
14
23
144
14
35
```

または

```
SELECT p.*
  FROM employees e, TABLE(e.projects) p
  WHERE e.empid = 100;

PROJECTS
--------
14
23
144
```

次の例は、プロジェクトを担当している従業員の行のみを取り出します。

```
SELECT e.empname, p.*
  FROM employees e, TABLE(e.projects) p;
```

プロジェクトを担当していない従業員の行を取り出すには、外部結合構文を使用します。

```
SELECT e.*, p.*
  FROM employees e, TABLE(e.projects)(+) p;
```

(+) は、employees および e.projects 間の依存結合を、NULL で補強する必要があることを示しています。つまり、e.projects が NULL または空であっても、対応する employees の行が存在すれば、問合せ結果として employees の行を返します。

TABLE 式の副問合せが含まれている問合せのネスト解除

前述の例は、コレクションの名前が入っている TABLE 式を示しています。もう 1 つの方法は、TABLE 式に、コレクションの副問合せを含めるというものです。

次の例では、100 という ID を持つ従業員に関するプロジェクトのコレクションが戻されます。

```
SELECT *
  FROM TABLE(SELECT e.projects
               FROM employees e
               WHERE e.empid = 100);

PROJECTS
--------
14
23
144
```

TABLE 式での副問合せの使用には、次の制限事項があります。

- 副問合せでは、コレクション型を戻す必要があります。
- 副問合せの SELECT 構文のリストには、1 つの項目を入れる必要があります。
- 副問合せでは、1 つのコレクションのみを戻す必要があります。つまり、複数行についてのコレクションを戻すことはできません。たとえば、副問合せ SELECT projects FROM employees は、employees 表に 1 つの行のみ入っている場合にかぎり、TABLE 式で成功します。表に複数行が含まれている場合、副問合せはエラーを戻します。

次の例では、CURSOR 式に埋め込まれている SELECT の FROM 句に使用されている TABLE 式を示します。

```
SELECT e.empid, CURSOR(SELECT * FROM TABLE(e.projects))
  FROM employees e;
```

マルチレベル・コレクションを伴う問合せのネスト解除

問合せのネスト解除は、VARRAY とネストした表のどちらについても、マルチレベル・コレクションで使用できます。次の例は、ネストした表のマルチレベルのネストした表コレクションに対して行う問合せのネスト解除を示しています。この問合せでは、それぞれの星が惑星のネストした表を持ち、それぞれの惑星が衛星のネストした表を持つ stars 表に基づいて、ネストした表の内部セットからすべての衛星の名前が戻されます。

```
SELECT t.name
  FROM stars s, TABLE(s.planets) p, TABLE(p.satellites) t;
```

参照： 8-12 ページの「ネストを解除する問合せを使用したリレーショナル形式でのオブジェクト・データの表示」を参照してください。

コレクションを対象とした DML 操作の実行

Oracle は、ネストした表列を対象とした次の DML 操作をサポートします。

- コレクション全体に対して新しい値を提供する挿入および更新
- ピース単位更新
  - コレクションへの新しい要素の挿入
  - コレクションからの要素の削除
  - コレクションの要素の更新

VARRAY 列を対象としたピース単位の更新はサポートされません。ただし、VARRAY 列を基本単位として挿入または更新することはできます。

ネストした表列をピース単位で更新する場合、DML 文は TABLE 式を使用して、操作対象となるネストした表の値を識別します。

次の DML 文は、ネストした表列を対象としたピース単位操作を示しています。

```
INSERT INTO TABLE(SELECT e.projects
                  FROM       employees e
                  WHERE      e.eno = 100)
   VALUES (1, 'Project Neptune');

UPDATE TABLE(SELECT e.projects
                 FROM       employees e
                 WHERE      e.eno = 100) p
   SET VALUE(p) = project_typ(1, 'Project Pluto')
   WHERE p.pno = 1;

DELETE FROM TABLE(SELECT e.projects
                  FROM       employee e
                  WHERE      e.eno = 100) p
   WHERE p.pno = 1;
```

マルチレベル・コレクションを対象とした DML 操作の実行

ネストした表のマルチレベル・コレクションの場合、DML 操作は、コレクション全体を基本単位とした実行、または特定の要素に対するピース単位の実行ができます。VARRAY のマルチレベル・コレクションの場合、DML 操作はコレクション全体を基本単位として実行されるのみです。

基本データ単位としてのコレクション 前述の「マルチレベル・コレクションのコンストラクタ」では、INSERT 文を使用して、マルチレベル・コレクション全体を挿入する例を示しました。マルチレベル・コレクションの自動更新は、UPDATE 文を使用して行うこともできます。次に、v_planets を惑星のネストした表型 nt_pl_t として宣言された変数だと仮

定してみましょう。次の文は、planets コレクションを 1 つの単位として v_planets の値に設定し、stars を更新します。

```
UPDATE stars  s
SET s.planets = :v_planets
WHERE s.name = 'Aurora Borealis';
```

ネストした表を対象にしたピース単位操作 ピース単位の DML 操作は、ネストした表に対してのみ実行できます。VARRAY に対しては実行できません。

次の例は、ネストした表 planets に対して行うピース単位挿入操作を示しています。この例は、新しい惑星に satellite_t のネストした表を付けて挿入します。

```
INSERT INTO TABLE( SELECT planets FROM stars WHERE name = 'Sun')
VALUES ('Saturn', 56,
  nt_sat_t(
    satellite_t('Rhea', 83)
  )
);
```

次の例は、ピース単位挿入操作を実行して、内側のネストした表を挿入し、惑星用の衛星を追加します。この例も、前述の例と同様、内側のネストした表を選択して、挿入のターゲットを指定する副問合せが含まれている TABLE 式を使用します。

```
INSERT INTO TABLE( SELECT p.satellites
  FROM TABLE( SELECT s.planets
    FROM stars s
    WHERE s.name = 'Sun') p
  WHERE p.name = 'Uranus')
VALUES ('Miranda', 31);
```

型の継承

オブジェクト型を使用することにより、アプリケーションで扱う実社会のエンティティ（顧客や発注書など）をモデル化できます。ただし、これはオブジェクトの機能を利用する最初の手順にすぎません。オブジェクトを使用すると、顧客のようなエンティティをモデル化できるのみでなく、元の型を基にして顧客の特化された様々な型を**型階層**の中で定義することもできます。このように定義した後、階層に対して操作を行い、それぞれの型を実装させ、専用の方法で操作を実行できます。

型階層は、オブジェクト型の系図のようなものです。この階層は、**スーパータイプ**と呼ばれる親ベース型と、親から導出された**サブタイプ**と呼ばれる子オブジェクト型の 1 つ以上のレベルで構成されます。

階層内のサブタイプは、**継承**によりスーパータイプに結び付けられます。したがって、サブタイプはその親型の属性およびメソッドを自動的に取得します。また、サブタイプは親の属

性またはメソッドの変更も、自動的に取得します。スーパータイプの属性またはメソッドが更新されると、サブタイプにおいてもこの更新が行われます。

親から継承したセットに新しい属性およびメソッドを追加するか、またはサブタイプが継承するメソッドを再定義すると、サブタイプは親型の特殊なバージョンになります。継承したメソッドを再定義すると、サブタイプは独自の方法でメソッドを実行できるようになります。これに加え、通常サブタイプのオブジェクト・インスタンスは、コード内に存在するそのスーパータイプのいずれかのオブジェクト・インスタンスと置き換えることができます。そのため、**ポリモフィズム**が得られます。

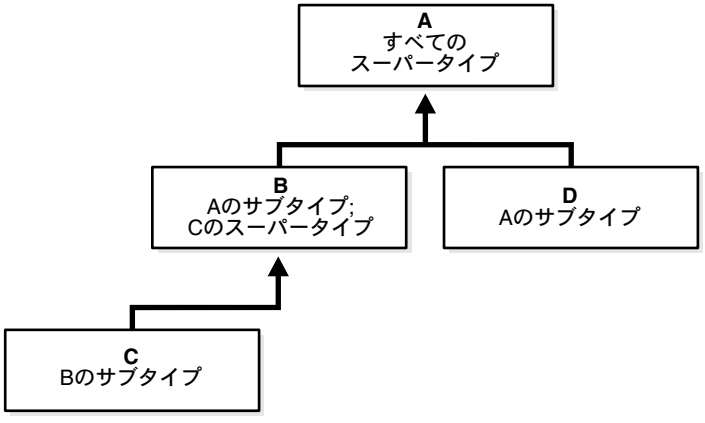
ポリモフィズムとは、コード内の値に、特定の宣言された型または宣言された型のサブタイプの値を格納させるスロットの機能です。様々な型がそれぞれ異なる方法でメソッドを実装しているために、どのような値がスロットを占有していても、この値に対してコールされたメソッドは、その値の型に応じて、それぞれ違った方法で機能します。

型およびサブタイプ

サブタイプは、スーパータイプから直接または他のサブタイプを介して間接的に導出できます。

サブタイプは1つのスーパータイプから直接導出されるのみで、複数のスーパータイプから導出されることはありません。スーパータイプは、兄弟関係にあるサブタイプを複数持つことができますが、サブタイプが持つことができる直系の親スーパータイプは1つのみです。つまり、多重継承はサポートされず、単一継承のみがサポートされます。

サブタイプは、スーパータイプの特殊な異型を定義することによって、スーパータイプから**導出**されます。たとえば、`customer` オブジェクト型からは、特殊な異型である `govt_customer` や `corp_customer` を導出できます。これらのサブタイプは、根本的には `customer` ですが、顧客の特殊な種類です。サブタイプでは、その親から受け取る属性またはメソッドになんらかの変更が加えられますが、これが、サブタイプを特別な存在にすると同時に、親のスーパータイプとの間に違いをもたらします。



オブジェクト型の属性およびメソッドにより、型に性質が与えられます。つまり、これらの属性およびメソッドが、オブジェクト型を決定する重要な特徴になります。customer オブジェクト型が customer_id、name および address という 3 つの属性と、メソッド get_id() を持つ場合、customer から導出されたオブジェクト型は、これらと同じ 3 つの属性とメソッド get_id() を持つことになります。サブタイプはその親型の特殊なケースであり、別な種類の存在ではありません。したがって、サブタイプは一般的な機能を親型と共有します。

サブタイプの属性またはメソッドは、次の方法で特化します。

- 親スーパータイプが持たない**新しい属性を追加**します。

  たとえば、account_mgr_id の属性を定義に追加して、corp_customer を特化し customer の特殊な種類にできます。サブタイプは、親から継承した属性の型の削除や変更はできず、新しい属性を追加するのみです。

- 親スーパータイプにない**新しいメソッドを追加**します。
- サブタイプのバージョンが親のバージョンとは異なるコードを実行できるように、サブタイプが親から継承する**メソッドの一部の実装を変更**します。

  たとえば、shape オブジェクト型でメソッド calculate_area() を定義した場合、shape の 2 つのサブタイプである rectilinear_shape および circular_shape は、それぞれ別の方法でこのメソッドを実装する可能性があります。

サブタイプが親型から属性およびメソッドを引き継ぐことを、**継承する**といいます。つまり、サブタイプを定義した場合、単に親の属性およびメソッドにならってサブタイプの属性およびメソッドが作成されるのみではないということです。オブジェクト型の場合、継承リンクが存続します。親型の属性またはメソッドに後で加えられた変更も継承されるので、サブタイプにもこれらの変更が反映されます。継承したメソッドを再実装しないかぎり、サブタイプには常に親型が持つ属性およびメソッドと同じコア・セットと、自身が追加する属性とメソッドが含まれます。

子型は、その親と異なる型ではなく、その型の特殊な種類だということに注意してください。customer の一般的定義を変更すると、corp_customer の定義も変更されます。

スーパータイプとそのサブタイプの間の継承関係が、オブジェクトの能力の大半をもたらす原因になっていると同時に、オブジェクトを複雑にする原因にもなっています。スーパータイプのメソッドが変更でき、再コンパイルするのみですべての下位サブタイプにこの変更を反映させることができるというのは、非常に強力な機能です。しかし、それと同時に、型を特化する、またはメソッドを再定義することを検討する必要があります。階層内のどの型も、表や列に入れることができるというのも、強力な機能ですが、特別なケースでこれを認めるかどうかを決める必要があります。また、使用する型の範囲のみが型階層から選択されるように、DML 文および問合せに制約を付ける必要がある場合もあります。次の項では、オブジェクトのこのような側面について説明します。

FINAL、NOT FINAL の型およびメソッド

オブジェクト型の定義により、その型からサブタイプが導出可能かどうかが決まります。サブタイプを導出できるようにするには、オブジェクト型を **NOT FINAL** として定義する必要があります。このように定義するには、オブジェクトの型の宣言に NOT FINAL キーワードを入れます。次に例を示します。

```
CREATE TYPE Person_typ AS OBJECT
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100)) NOT FINAL;
```

前述の文は、Person_typ のサブタイプが定義できるように、Person_typ を NOT FINAL 型として宣言しています。デフォルトでは、オブジェクト型は **FINAL** と定義されます。そのため、そのオブジェクト型からサブタイプを導出することはできません。

FINAL 型から NOT FINAL 型への切替え、またその逆の切替えには、ALTER TYPE 文を使用します。次に示す文は、Person_typ を FINAL 型に変更します。

```
ALTER TYPE Person_typ FINAL;
```

NOT FINAL から FINAL へ変更できるのは、ターゲット型がサブタイプを持たない場合のみです。

メソッドも FINAL または NOT FINAL として宣言できます。メソッドが FINAL として宣言されると、サブタイプは自身の実装を組み込むことで、メソッドを**オーバーライド**することはできません。メソッドは、型とは異なり、FINAL がデフォルト設定ではないため、FINAL として明示的に宣言する必要があります。

次の文は、FINAL のメンバー・ファンクションを含んでいる NOT FINAL 型を作成します。

```
CREATE TYPE T AS OBJECT (...,
  MEMBER PROCEDURE Print(),
  FINAL MEMBER FUNCTION foo(x NUMBER)...
) NOT FINAL;
```

参照： 2-37 ページの「メソッドのオーバーライド」を参照してください。

サブタイプの作成

サブタイプは、CREATE TYPE 文を使用して作成します。この文は、UNDER パラメータにより、サブタイプの直系の親を指定します。

```
CREATE TYPE Student_typ UNDER Person_typ
( deptid NUMBER,
   major VARCHAR2(30)) NOT FINAL;
```

この文は、Student_typ を Person_typ のサブタイプとして作成します。Person_typ のサブタイプである Student_typ は、Person_typ に宣言されているか、または Person_typ が継承したすべての属性と、Person_typ が継承したか、または Person_typ で宣言されているすべてのメソッドを継承します。

Student_typ の定義文は、2 つの新しい属性を追加して、Person_typ を特化します。サブタイプの中で宣言する新しい属性には、自身の型階層内の上位に位置するそのスーパータイプのいずれかに宣言されている属性またはメソッドとは別の名前を持たせる必要があります。

型には、複数の子サブタイプを持たせることができます。また、これらの子サブタイプにもサブタイプを持たせることができます。次の文は、もう 1 つのサブタイプである Employee_typ を Person_typ の下に作成します。

```
CREATE TYPE Employee_typ UNDER Person_typ
( empid NUMBER,
  mgr VARCHAR2(30));
```

別のサブタイプの下に、サブタイプを定義できます。この場合も、新しく定義されたサブタイプは、その親型が持つすべての属性およびメソッド（宣言されているもの、および継承されたもの）を継承します。次の文は、新しいサブタイプである PartTimeStudent_typ を Student_typ の下に定義します。この新しいサブタイプは、Student_typ のすべての属性およびメソッドを継承し、新しい属性を追加します。

```
CREATE TYPE PartTimeStudent_typ UNDER Student_typ
( numhours NUMBER);
```

インスタンス化不可の型およびメソッド

オブジェクト型をインスタンス化不可の型として宣言することができます。型がインスタンス化不可（コンストラクタを持たない）の場合は、この型にはコンストラクタ（デフォルトまたはユーザー定義のもの）が存在しないため、その型のインスタンス（オブジェクト）のインスタンス生成はできません。このオプションと組み合せて使用する可能性があるのは、インスタンス生成の対象となる特化サブタイプのスーパータイプとしてのみ使用する型です。次に例を示します。

```
CREATE TYPE Address_typ AS OBJECT(...) NOT INSTANTIABLE NOT FINAL;
CREATE TYPE USAddress_typ UNDER Address_typ(...);
CREATE TYPE IntlAddress_typ UNDER Address_typ(...);
```

メソッドもインスタンス化不可として宣言できます。このオプションは、型にメソッドを実装せずにメソッドを宣言する場合に使用します。インスタンス化不可のメソッドを含む型は、その型自体をインスタンス化不可として定義する必要があります。次に例を示します。

```
CREATE TYPE T AS OBJECT (
  x NUMBER,
  NOT INSTANTIABLE MEMBER FUNCTION func1() RETURN NUMBER
) NOT INSTANTIABLE NOT FINAL;
```

インスタンス化不可のメソッドは、プレースホルダの役目を果たします。すべてのサブタイプにおいてそれぞれ独自の方法でメソッドをオーバーライドする場合に、インスタンス化不可のメソッドを定義します。この場合、スーパータイプにメソッドを定義しても意味がありません。

インスタンス化不可の継承メソッドが、サブタイプにおいても実装されない場合、スーパータイプと同様にサブタイプそのものをインスタンス化不可として定義する必要があります。

インスタンス化不可のサブタイプは、インスタンス化不可のスーパータイプの下に定義できます。

インスタンス化可能な型からインスタンス化不可の型への切替え、またその逆の切替えには、ALTER TYPE 文を使用します。例として、Example_typ をインスタンス化可能な型にする文を示します。

```
ALTER TYPE Example_typ INSTANTIABLE;
```

インスタンス化可能な型からインスタンス化不可の型に変更できる型は、その型を直接または別の型やサブタイプを介して間接的に参照する列、ビュー、表またはインスタンスを持たないものに限定されます。

インスタンス化不可の型を FINAL として宣言することはできません（宣言しても、意味がありません）。

メソッドの継承、オーバーロードおよびオーバーライド

サブタイプは、上位のスーパータイプに宣言されているか、またはこのスーパータイプにより継承されたすべてのメソッド（メンバー・メソッドとスタティック・メソッドの両方）を自動的に継承します。

サブタイプは自身が継承するメソッドを再定義できる他、新しいメソッドを追加することも可能です。サブタイプ自身が継承するメソッドと同じ名前を持つメソッドも追加できるため、結果的にサブタイプに同じ名前を持つメソッドが 2 つ以上入ることになります。

1 つの型に同じ名前の複数のメソッドを持たせることを、メソッドの**オーバーロード**といいます。あるサブタイプの動作をカスタマイズするために、継承されたメソッドを再定義することを、メソッドの**オーバーライド**といいます。

メソッドのオーバーロード

様々な動作方法を用意するには、オーバーロードが役立ちます。たとえば、draw() オブジェクトを持つ shape オブジェクトに、テキスト・ラベルを図面に追加し、ラベルのテキストの引数が入っている別の draw() メソッドをオーバーロードすることができます。

1 つの型が同じ名前のメソッドをいくつか持っている場合、コンパイラはメソッドの**シグネチャ**を使用して、これらを区別します。メソッドのシグネチャは一種の構造プロファイルで、メソッドの名前、数値、型、およびメソッドの仮パラメータ（暗黙的な self パラメータなど）の順序で構成されます。名前は同じでも、別の**シグネチャ**を持つメソッドは、**オーバーロード**（同一型に存在する場合）と呼ばれます。

次の例のサブタイプ MySubType_typ は、foo() のオーバーロードを作成します。

```
CREATE TYPE MyType_typ AS OBJECT (...,
  MEMBER PROCEDURE foo(x NUMBER), ...) NOT FINAL;

CREATE TYPE MySubType_typ UNDER MyType_typ (...,
  MEMBER PROCEDURE foo(x DATE),
  STATIC FUNCTION bar(...)...
  ...);
```

MySubType_typ には、2 つのバージョンの foo( ) が入っています。1 つは、NUMBER パラメータを持つ継承されたバージョンで、もう 1 つは DATE パラメータを持つ新しいバージョンです。

メソッドのオーバーライド

オーバーライドは、サブタイプにおいて継承メソッドにこれまでとは異なる動作をさせるために、継承メソッドを再定義することです。たとえば、shape スーパータイプから導出されたサブタイプ circular_shape は、円の面積を求めるようにメソッド calculate_area() をカスタマイズするために、このメソッドをオーバーライドします。

サブタイプがメソッドをオーバーライドする場合は、そのサブタイプのインスタンスがこのメソッドを起動するごとに、オーバーライドされたメソッドではなく、新しいバージョンのメソッドが実行されます。サブタイプ自身がサブタイプを持つ場合は、元のメソッドではなく、メソッドのオーバーライドを継承します。

スーパータイプが、サブタイプでオーバーライドされるメソッドのオーバーロードを含む場合もあります。メソッドのオーバーロードは、すべて同じ名前を持つため、コンパイラはサブタイプのオーバーライド・メソッドのシグネチャを使用して、オーバーライドするスーパータイプのバージョンを識別します。つまり、メソッドをオーバーライドするには、メソッドのシグネチャを保持する必要があります。

型定義では、メソッド宣言の始めに OVERRIDING キーワードを入れ、メソッドをオーバーライドすることを通知します。たとえば、次のコードでは、サブタイプがメソッド Print() をオーバーライド中であると通知しています。

```
CREATE TYPE MyType_typ AS OBJECT (...,
  MEMBER PROCEDURE Print(),
  FINAL MEMBER FUNCTION foo(x NUMBER)...
) NOT FINAL;

CREATE TYPE MySubType_typ UNDER MyType_typ (...,
  OVERRIDING MEMBER PROCEDURE Print(),
...);
```

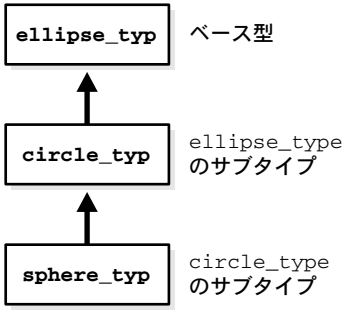
新しいメソッドと同様、オーバーライドするメソッドについての宣言は、CREATE TYPE BODY 文の中に入れます。

メソッドのオーバーライドの制限事項

- オーバーライドできるメソッドは、スーパータイプで FINAL として宣言されていないもののみです。
- オーダー・メソッドが出現する可能性があるのは、型階層のルート型のみで、サブタイプで再定義（オーバーライド）することはできません。
- サブタイプのスタティック・メソッドにより、スーパータイプのメンバー・メソッドを再定義することはできません。
- サブタイプのメンバー・メソッドにより、スーパータイプのスタティック・メソッドを再定義することはできません。
- オーバーライドされているメソッドが、いずれかのパラメータのデフォルト値を持つ場合は、そのパラメータのデフォルト値と同じものを、オーバーライドするメソッドに持たせる必要があります。

動的メソッド・ディスパッチ

メソッドをオーバーライドすると、型階層が同じメソッドの複数の実装を定義できるようになります。たとえば、ellipse_typ 型、circle_typ 型、sphere_typ 型の階層の中で、それぞれの型に、異なる方法でメソッド calculate_area() を定義できます。



このようなメソッドが起動されると、メソッドを起動したオブジェクト・インスタンスの型により、メソッドが使用する実装が決まります。次に、この実装にコールがディスパッチされ、実行されます。メソッド実装を選択するこのプロセスは、コンパイル時ではなく、実行時に処理されるため、「仮想メソッド・ディスパッチ」または「動的メソッド・ディスパッチ」と呼ばれます。

メソッド・コールは最も近い実装にディスパッチされ、現在の型または指定された型に基づいて、継承階層のバックアップを行います。コールによりオブジェクト・インスタンスのメンバー・メソッドが起動される場合は、そのインスタンスの型が現在の型となり、その型で定義されているか、その型が継承した実装が使用されます。コールにより型のスタティック・メソッドが起動された場合は、その型で定義されているか、その型が継承した実装が使用されます。

たとえば、c1 が circle_typ のオブジェクト・インスタンスの場合、c1.foo() により circle_typ に定義されている foo() の実装が最初に検索されます。何も見つからない場合は、実装のスーパータイプ・チェーンを検索して ellipse_typ の実装を見つけます。sphere_typ も実装を定義していますが、型階層の検索は上位に向かって順方向でのみ実行されるため、この検索の対象外となります。現在の型のサブタイプは検索されません。

同様に、スタティック・メソッド circle_typ.bar() へのコールでは、最初に circle_typ の中を調べ、必要に応じて、次に circle_typ のスーパータイプの中を調べます。サブタイプ sphere_typ は検索対象になりません。

型階層内の型の代入

型階層内では、サブタイプはルートのベース型の変種です。たとえば、Student_typ 型および Employee_typ は、Person_typ の 1 つです。ベース型に、これらの型が含まれます。

型階層内で型を扱うときは、最も一般的なレベルで作業する場合（全員の選択や更新など）や、学生のみ、または学生以外の個人のみを選択または更新する場合があります。

全員を選択し、宣言された型が Person_typ のオブジェクトのみでなく、宣言された型（サブタイプ）が Student_typ または Employee_typ のオブジェクトも返す（多様な性質を持つ）機能は、**代入性**と呼ばれます。スロット内にあるスーパータイプ（変数、列など）のかわりに、スーパータイプの下位のサブタイプの 1 つが使用できる場合、スーパータイプは代入可能です。

一般的に、型は代入可能です。サブタイプが上位のスーパータイプの単なる特殊な種類である場合は、型が代入可能であることを期待します。ただし、形式上、サブタイプは元々型の 1 つで、上位のスーパータイプと同じ型ではありません。学生全員や、就業者全員など、個人全員が入っている列には、複数の型のデータが実際に格納されています。

代入性の効果が発揮されるのは、オブジェクト型、オブジェクト型に対する REF またはコレクション型として宣言されているオブジェクト・ビューまたはオブジェクト表の属性、列および行です。

基本的に、オブジェクト属性、コレクション要素および REF は常に代入可能です。型定義のレベルでは、これらの代入性をあるサブタイプに制限する構文はありません。ただし、特定の表や列について、記憶域レベルの代入性を無効にすることも、制約を付けることもできます。

> **参照：** 2-43 ページの「代入性の無効化」および「代入性の制約」を参照してください。

属性の代入性

オブジェクト属性、コレクション要素および REF は代入可能です。MyType がオブジェクト型の場合、次のことができます。

- REF 型属性 : REF MyType として定義された属性には、MyType のインスタンス、または MyType のサブタイプのインスタンスの REF を入れることができます。
- オブジェクト型属性 : MyType として定義された属性には、MyType のインスタンス、または MyType のサブタイプのインスタンスを入れることができます。
- コレクション型要素 : 型 MyType の要素のコレクションには、MyType のインスタンスおよび MyType のサブタイプのインスタンスを入れることができます。

たとえば、次に定義されている Book_typ の author 属性は代入可能です。

```
CREATE TYPE Book_typ AS OBJECT
( title VARCHAR2(30),
  author Person_typ     /* substitutable */);
```

Book_typ のインスタンスを作成するには、タイトル文字列および Person_typ の著者、または Person_typ のサブタイプを指定します。次の例は、型 Employee_typ の著者を指定します。

```
Book_typ('My Oracle Experience',
       Employee_typ(12345, 'Joe', 'SF', 1111, NULL))
```

一般に、属性はドット表記法を使用してアクセスできます。1 つの行または列の宣言された型のサブタイプの属性は、TREAT ファンクションによりアクセスできます。たとえば、Book_typ のオブジェクト・ビュー Books_v では、Employee_typ の著者の従業員 ID を取得するために、TREAT が使用できます（author 列は Person_typ 型です）。

```
SELECT TREAT(author AS Employee_typ).empid FROM Books_v;
```

参照： 2-48 ページの「TREAT」を参照してください。

列および行の代入性

オブジェクト型の列は代入可能です。また、オブジェクト表およびビューの中のオブジェクト型行も代入可能です。つまり、型 T として定義する列または行には、T のインスタンスおよびそのサブタイプを入れることができます。

次に示す文は、前述した Person_typ 型階層の例です。

```
CREATE TYPE Person_typ AS OBJECT
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100)) NOT FINAL;

CREATE TYPE Student_typ UNDER Person_typ
( deptid NUMBER,
   major VARCHAR2(30)) NOT FINAL;
```

```
CREATE TYPE PartTimeStudent_typ UNDER Student_typ
( numhours NUMBER);
```

Person_typ のオブジェクト表には、3 つの型すべての行を入れることができます。特定の型のインスタンスは、INSERT 文の VALUES 句の中でその型のコンストラクタを使用して挿入します。

```
CREATE TABLE persons OF Person_typ;

INSERT INTO persons
  VALUES (Person_typ(1243, 'Bob', '121 Front St'));

INSERT INTO persons
  VALUES (Student_typ(3456, 'Joe', '34 View', 12, 'HISTORY'));

INSERT INTO persons
  VALUES (PartTimeStudent_typ(5678, 'Tim', 13, 'PHYSICS', 20));
```

リレーショナル表またはビューの中の型 Person_typ の代入可能列にも、3 つの型すべてのインスタンスを入れることができます。次の例は、Person_typ 列 author に、個人、学生および定時制の学生を挿入します。

```
CREATE TABLE books (title varchar2(100), author Person_typ);

INSERT INTO books
 VALUES('An Autobiography', Person_typ(1243, 'Bob'));

INSERT INTO books
 VALUES('Business Rules', Student_typ(3456, 'Joe', 12, 'HISTORY'));

INSERT INTO books
 VALUES('Mixing School and Work',
        PartTimeStudent_typ(5678, 'Tim', 13, 'PHYSICS', 20));
```

新しく作成したサブタイプは、代入可能な表およびそのスーパータイプの列のいずれにも（サブタイプを作成する前に存在していた表や列など）格納できます。

スーパータイプの属性を持つサブタイプ

サブタイプには、スーパータイプである属性を持たせることができます。次に例を示します。

```
CREATE TYPE Student_typ UNDER Person_typ (..., advisor Person_typ);
```

ただし、このタイプの列は代入可能ではありません。同様に、サブタイプ ST には、ST のスーパータイプの 1 つが要素の型になっているコレクション属性を入れることができますが、この場合もこのタイプの列は代入可能ではありません。たとえば、Student_typ に

Person_typ のネストした表または VARRAY がある場合、Student_typ 列は代入可能ではありません。

ただし、スーパータイプを参照する REF 属性を持つサブタイプの代入可能な列が定義できます。

参照： 2-43 ページの「代入性の無効化」を参照してください。

REF 列および属性

REF 列および属性は、ビューでも表でも代入可能です。たとえば、ビューまたは表のいずれかで、REF Person_typ として宣言された列に、Person_typ またはそのサブタイプのインスタンスの参照を入れることができます。

コレクション要素

コレクション要素は、ビューでも表でも代入可能です。たとえば、Person_typ のネストした表には、Person_typ またはそのサブタイプのオブジェクト・インスタンスを入れることができます。

代入可能な列の作成後のサブタイプの作成

サブタイプを作成する場合、スーパータイプの代入可能な列を持つ表が自動的に使用可能になり、新しいサブタイプを格納できるようになります。つまり、サブタイプの作成には、このような表の有無が影響します。このような表が存在する場合、代入可能なサブタイプ（Oracle によってその表が格納可能になるサブタイプ）のみ作成できます。

次に、Student_typ サブタイプを作成しようとする例を示します。ただし、Student_typ はスーパータイプの属性を持ち、persons 表に、スーパータイプの代入可能な列 p があるため、この操作は正常に実行されません。

```
CREATE TYPE Person_typ AS OBJECT
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100)) NOT FINAL;

CREATE TYPE Employee_typ UNDER Person_typ
( salary NUMBER) NOT FINAL;

CREATE TABLE persons (p person_typ);
-- Table persons can store Person_typ and Employee_typ

INSERT INTO persons
  VALUES (Person_typ(1243, 'Bob', '121 Front St'));

-- This statement fails because there exists a substitutable
-- column of the supertype.
```

```
CREATE TYPE Student_typ UNDER Person_typ
( advisor Person_typ);
```

次の操作は正常に実行されます。このような Student_typ サブタイプは、代入可能です。Oracle は、persons 表を自動的に使用可能にし、この新しい型のインスタンスを格納します。

```
CREATE TYPE Student_typ UNDER Person_typ
( deptid NUMBER,
  major VARCHAR2(30)) NOT FINAL;

-- Inserts an instance of the subtype in table persons
INSERT INTO persons
  VALUES (Student_typ(3456, 'Joe', '34 View', 12, 'HISTORY'));
```

代入可能な列の作成後のサブタイプの削除

スーパータイプの代入可能な列に、サブタイプのインスタンスが格納されていない場合にのみ、VALIDATE オプションを使用してサブタイプを削除できます。

たとえば、次の文は、persons 表の代入可能な列 p に、Student_typ のインスタンスが格納されているため、正常に実行されません。

```
-- This statement fails:
DROP TYPE Student_typ VALIDATE;
```

型を削除するには、最初にスーパータイプの代入可能な列にあるそのインスタンスを削除します。

```
DELETE FROM persons WHERE p IS OF (Student_typ);

-- Now the DROP statement succeeds
DROP TYPE Student_typ VALIDATE;
```

代入性の無効化

列または属性（任意のレベルにネストされた埋込み属性やコレクションなど）についての代入性は、NOT SUBSTITUTABLE AT ALL LEVELS 句により無効にできます。

次に示す句は、Person_typ インスタンスのみを著者として格納するように、リレーショナル表の book 列に制約を設定し、サブタイプのどのインスタンスも使用できなくします。

```
CREATE TABLE catalog (book Book_typ, price NUMBER)
  COLUMN book NOT SUBSTITUTABLE AT ALL LEVELS;
```

オブジェクト表の場合は、次のように句を表に適用できます。

```
CREATE TABLE Student_books OF Book_typ NOT SUBSTITUTABLE AT ALL LEVELS;
```

次のような構文を使用すると、コレクションの要素型が代入性を持たないように指定できます。

```
CREATE TABLE departments(name VARCHAR2(10), emps emp_set)
  NESTED TABLE (emps)
  NOT SUBSTITUTABLE AT ALL LEVELS STORE AS ...
```

代入性の無効化には、次の注意事項があります。

- REF 列の代入性を無効にするメカニズムはありません。
- 列に適用する NOT SUBSTITUTABLE AT ALL LEVELS 句の場合は、最上位の列である必要があります。この句は、オブジェクト型の属性に適用できません。

代入性の制約

オブジェクト列または属性で許可されているサブタイプの範囲を、宣言された型の階層内の特定のサブタイプに制限するように、制約を加えることができます。これを行うには、IS OF *type* の制約を使用します。

次の文は、著者が学生のみに限定される Book_typ の表を作成します。

```
CREATE TABLE Student_books OF Book_typ
  COLUMN author IS OF (ONLY Student_typ);
```

Book_typ 型が Person_typ 型の著者を許可している場合でも、列宣言により Student_typ のインスタンスのみを格納するように制約されます。

行オブジェクトおよび列オブジェクトを（複数ではなく）1 つのサブタイプに限定する場合、使用できるのは IS OF *type* の演算子のみで、前述の例のように ONLY キーワードを使用する必要があります。

オブジェクト列に制約を付けるには、IS OF *type* か NOT SUBSTITUTABLE AT ALL LEVELS のいずれか一方のみが使用可能で、両方は使用できません。

型をまたがる代入

この項で説明する代入規則は、INSERT/UPDATE 文、RETURNING 句、ファンクションのパラメータ、および PL/SQL 変数に適用されます。

オブジェクトおよびオブジェクトの REF

代入性とは、上位のスーパータイプの 1 つの代理を務めるサブタイプの能力のことです。サブタイプのかわりにスーパータイプを使用するなど、逆の方向で置換を行おうとすると、コンパイル時にエラーとなります。

型 Source_typ のソースを型 Target_typ に代入するパターンは、次のいずれかである必要があります。

- ケース 1:Source_typ および Target_typ が同一型である。
- ケース 2:Source_typ が Target_typ のサブタイプである（ワイドニング）。

ケース 2 は、**ワイドニング**を示しています。ワイドニングとは、ターゲットが宣言された型よりも、ソースが宣言された型の方が範囲が特定される代入のことです。従業員インスタンスを個人タイプの変数に代入する処理は、その例です。

これは、従業員を個人としてみなすという考え方です。個人の範囲を絞って定義し、対象を特化したのが従業員です。したがって、個人をさらに特化して従業員にする存在を無視する場合、個人用のスロットに従業員を入れることができます。すべての従業員が個人のため、ワイドニングは常に機能します。

次の表を使用して、ワイドニングについて説明します。

```
TABLE T(perscol Person_typ, empcol Employee_typ, stucol Student_typ)
```

次に示す代入は、ワイドニングを示しています。perscol が代入不可と定義されていないかぎり、代入は有効です。

```
UPDATE T set perscol = empcol;
```

PL/SQL では、次のとおりです。

```
declare
  var1 Person_typ;
  var2 Employee_typ;
begin
  var1 := var2;
end;
```

ワイドニングに加え、**ナローイング**もあります。ナローイングはワイドニングの逆で、個人など、一般化した広い範囲を対象とする型を、従業員など範囲を限定して定義された型とみなすことを意味します。すべての個人が従業員であるとはかぎらないため、このような特定の代入は、該当する個人が偶然に従業員の場合のみ機能します。

代入のナローイングを行うには、TREAT ファンクションを使用して、ソース値の宣言された型を範囲をさらに限定したターゲット型または階層内に存在する下位のサブタイプの 1 つに明示的に変更する必要があります。実行時、TREAT ファンクションは変更が可能かどうか確認するためのチェックを行います。ソース値（該当する個人）がターゲット型またはサブタイプの 1 つでない場合、TREAT は変更を行うか、または NULL を戻します。

たとえば、次の UPDATE 文は、列 perscol の Person_typ の値を、Employee_typ の列 empcol に設定します。perscol のそれぞれの値についての代入が成功するのは、この個人が従業員でもある場合のみです。George という人が従業員ではない場合、TREAT は NULL を戻し、代入が NULL に戻ります。

```
UPDATE T set empcol = TREAT(perscol AS Employee_typ);
```

次の文は、ソース値の宣言された型を明示的に変更せずに、代入のナローイングを試行しますが、エラーを戻す結果になります。

```
UPDATE T set empcol = perscol;
```

参照： 2-48 ページの「TREAT」を参照してください。

コレクションの代入

コレクション型の式の代入では、ソースの宣言された型がターゲットのそれと一致する必要があります。コレクションの代入では、ワイドニングとナローイングのどちらも使用できませんが、スーパータイプのコレクションにサブタイプ値を代入できます。

たとえば、次のようなコレクション型について考えてみます。

```
CREATE TYPE PersonSet AS TABLE OF Person_typ;
CREATE TYPE StudentSet AS TABLE OF Student_typ;
```

これらのコレクション型の式を、相互に代入することはできませんが、Student_typ のコレクション要素は PersonSet 型のコレクションに代入できます。

```
declare
  var1 PersonSet; var2 StudentSet;
  elem1 Person_typ; elem2 Student_typ;
begin
  var1 := var2;                 /* ILLEGAL - collections not of same type */
  var1 := PersonSet (elem1, elem2);      /* LEGAL : Element is of subtype */
```

比較 : オブジェクト、REF 変数およびコレクション

オブジェクト・インスタンスの比較

2 つのオブジェクト・インスタンスが比較できるのは、これらの宣言された型が一致するか、または一方がもう一方のサブタイプの場合のみです。

マップ・メソッドおよびオーダー・メソッドはオブジェクトを比較するためのメカニズムとなります。いずれか一方のメソッドを任意にオブジェクト型に定義して、この型のオブジェクトを比較する基準を指定します。いずれかのメソッドを定義した場合、その型のオブジェクトまたはそのサブタイプの 1 つを比較する必要があれば、そのメソッドが自動的にコールされます。

型がマップ・メソッドとオーダー・メソッドのどちらも定義していない場合は、SQL 文の中でのみ、その型のオブジェクト変数に対する等価比較または不等価比較が行われます（同じ型の 2 つのオブジェクトは、対応する属性の値が一致する場合のみ、等価とみなされます）。

参照： 2-16 ページの「オブジェクト比較のメソッド」を参照してください。

REF 変数の比較

2 つの REF 変数の比較ができるのは、これらの変数が参照するターゲットに宣言された型が一致するか、または一方がもう一方のサブタイプの場合のみです。

コレクションの比較

コレクションを比較するためのメカニズムはありません。

オブジェクトに便利なファンクションおよび述語

次に、オブジェクトやオブジェクトの参照を扱う際に特に役立つファンクションや述語を示します。

- VALUE
- REF
- DEREF
- TREAT
- IS OF TYPE
- SYS_TYPEID

使用例は、このマニュアルの他の章でも示しています。

PL/SQL の場合、VALUE、REF および DEREF ファンクションが現れるのは、SQL 文の中のみです。

VALUE

SQL 文では、VALUE ファンクションはオブジェクト表またはオブジェクト・ビューの相関変数（表の別名）を引数としてとり、表またはビューの行に対応するオブジェクト・インスタンスを戻します。たとえば、次の文は John Smith という名前の人を全員選択します。

```
SELECT VALUE(p) FROM person_table p
  WHERE p.name = "John Smith";
```

VALUE ファンクションにより、行の宣言された型のインスタンスか、またはその型のサブタイプを戻すことができます。たとえば、次の問合せでは、学生や従業員を含む個人全員が、個人のオブジェクト・ビュー Person_v から戻されます。

```
SELECT VALUE(p) FROM Person_v p;
```

個人のみ、つまり最も狭い意味での型を取り出すには、ONLY キーワードを使用して、問合せ対象のビューまたはサブビューの宣言された型の選択範囲を限定します。

```
SELECT VALUE(p) FROM ONLY(Person_v) p;
```

次の例は、更新を目的とするオブジェクト・インスタンス行を戻すために使用する VALUE です。

```
UPDATE TABLE(SELECT e.projects
               FROM        employees e
               WHERE       e.eno = 100) p
   SET VALUE(p) = project_typ(1, 'Project Pluto')
   WHERE p.pno = 1;
```

REF

SQL 文の REF ファンクションは、オブジェクト表またはビューの相関名を引数としてとり、その表またはビューからオブジェクト・インスタンスの参照（REF）を戻します。REF ファンクションにより、表 / ビューの宣言された型のオブジェクトの参照、またはその型のサブタイプのオブジェクトの参照が戻せます。たとえば、次の文は学生や従業員の参照など個人全員の参照を戻します。

```
SELECT REF(p) FROM Person_v p;
```

次の例は、id 属性が 0001 になっている個人（あるいは学生または従業員）の REF を戻します。

```
SELECT REF(p)
  FROM Person_v p
  WHERE p.id = 0001 ;
```

DEREF

SQL 文の DEREF ファンクションは、REF に対応するオブジェクト・インスタンスを戻します。DEREF により戻されるオブジェクト・インスタンスは、REF の宣言された型か、またはこの型のサブタイプになる可能性があります。

たとえば、次の文はオブジェクト・ビュー Person_v から学生や従業員を含む個人オブジェクトを戻します。

```
SELECT DEREF(REF(p)) FROM Person_v p;
```

TREAT

TREAT ファンクションは、式の宣言された型を指定された型（通常は、式の宣言された型のサブタイプ）に変更しようとします。つまり、このファンクションは、スーパータイプのインスタンスをサブタイプのインスタンスとして扱おうとします（たとえば、個人を学生として扱います）。特定のケースでこれが実現するかどうかは、該当する人が実際に学生（また

は定時制の学生のような学生のサブタイプ）かどうかで決まります。その人が学生であれば、学生が持つ可能性のある別の属性やメソッドと一緒に、学生として戻されます。その人が学生でない場合、TREAT は NULL を戻します。

TREAT は、主に次の 2 つの目的で使用します。

- 代入のナローイングで、階層内のさらに特化された型の変数に式が代入できるように、式のタイプを変更するため（スーパータイプの値をサブタイプに挿入するため）。
- 行または列の宣言された型のサブタイプの属性またはメソッドにアクセスするため。

次の例は、代入に使用されている TREAT を示しています。ここでは、個人型の列が従業員型の列に挿入されます。perscol 内のそれぞれの行について、TREAT より従業員型または NULL が戻されます。どちらが戻されるかは、該当する個人が従業員かどうかで決まります。

```
UPDATE T set empcol = TREAT(perscol AS Employee_typ);
```

次の例では、TREAT により、すべての Student_typ インスタンスが、Person_typ 型（Student_typ のスーパータイプ）のオブジェクト・ビュー Person_v から戻されます。この文は、TREAT を使用して、p の型を Person_typ から Student_typ に変更します。

```
SELECT TREAT(VALUE(p) AS Student_typ)
FROM Person_v p;
```

それぞれの p について TREAT による変更が成功するのは、p の値の最も狭い意味での型または特化された型が Student_typ か、そのサブタイプの 1 つである場合のみです。p が個人で学生ではない場合、または p が NULL の場合、TREAT は SQL の NULL を戻します。

REF 式の宣言された型を変更する場合にも、TREAT が使用できます。次に例を示します。

```
SELECT TREAT(REF(p) AS REF Student_typ)
FROM Person_v p;
```

この例は、すべての Student_typ インスタンスに対して REF を戻します。学生ではない個人インスタンスについては、NULL REF が戻されます。

行または列の宣言された型のサブタイプの属性またはメソッドにアクセスするために使用するのが、おそらく TREAT の最も重要な使用方法です。次の問合せでは、major 属性を持つ個人全員（学生や定時制の学生）のこの属性が取り出されます。学生ではない人については、NULL が戻されます。

```
SELECT name, TREAT(VALUE(p) AS Student_typ).major major
  FROM persons p;

NAME    MAJOR
----    ------
Bob     null
Joe     HISTORY
Tim     PHYSICS
```

major は Student_typ の属性であっても、表 persons の宣言された型である Person_typ ではないため、次の問合せは意図したとおりには機能しません。

```
SELECT name, VALUE(p).major major
  FROM persons p;
```

型 T の代入可能なオブジェクト表または列には、T のすべてのサブタイプのすべての属性についての非表示列があります。非表示列は DESCRIBE 文でリストされませんが、サブタイプの属性データを含んでいます。TREAT を使用すると、これらの列にアクセスできます。

次に、サブタイプのメソッドにアクセスするために使用する TREAT の例を示します。

```
SELECT name, TREAT(VALUE(p) AS Student_typ).major() major
  FROM persons p;
```

> **参照：** 代入で使用する TREAT の詳細は、2-44 ページの「型をまたがる代入」を参照してください。

現在、TREAT は SQL についてのみサポートされ、PL/SQL ではサポートされません。

IS OF *type*

IS OF *type* の述語は、オブジェクト・インスタンスの型の特化レベルを検査します。

たとえば、次の問合せでは、個人表に格納されている学生インスタンス（学生のすべてのサブタイプなど）がすべて取り出されます。

```
SELECT VALUE(p) FROM persons p
WHERE VALUE(p) IS OF (Student_typ);

VALUE(p)
--------
Student_typ('Joe', 3456, 12, 10000)
PartTimeStudent_typ('Tim', 5678, 13, 1000, 20)
```

指定されたサブタイプではないオブジェクト、または指定されたサブタイプのサブタイプに対しては、IS OF より FALSE が戻されます（指定されたサブタイプのサブタイプは、指定されたサブタイプのさらに特化されたバージョンにすぎません）。このようなサブタイプを除外するには、ONLY キーワードが使用できます。このキーワードを使用すると、IS OF は指定された型を除くすべての型に対して FALSE を戻します。

次の問合せでは、学生の著書のみが取り出され、学生のサブタイプ（PartTimeStudent_typ など）の著書は除外されます。

```
SELECT b.title title, b.author author FROM books b
WHERE b.author IS OF (ONLY Student_typ);
```

```
TITLE             AUTHOR
-----             ------
Business Rules    Student_typ('Joe', 3456, 12, 10000)
```

次に示す文は、個人、従業員および学生が入っているオブジェクト・ビュー Person_v のオブジェクトを検証し、指定された 2 つの個人サブタイプである Employee_typ および Student_typ（そのサブタイプが存在する場合は、これらのサブタイプも含まれます）のオブジェクトのみに対する REF を戻します。

```
SELECT REF(p) FROM Person_v p
WHERE VALUE(p) IS OF (Employee_typ, Student_typ);
```

次の文は、最も狭い意味での型または特化された型が Student_typ である学生のみを戻します。ビューに Student_typ のサブタイプ（PartTimeStudent_typ など）のオブジェクトが含まれている場合、これらのオブジェクトは除外されます。この例は、TREAT ファンクションを使用して、学生オブジェクトをビューの宣言された型（つまり Person_typ）から Student_typ に変換します。

```
SELECT TREAT(VALUE(p) AS Student_t)
FROM Person_v p
WHERE VALUE(p) IS OF(ONLY Student_t);
```

REF が示すオブジェクトの型を検証する場合、IS OF *type* の述語を使用して検証する前に、REF の参照を解除するには、DEREF ファンクションが使用できます。

たとえば、PersRefCol を REF Person_typ として宣言すると、次に示すように学生についての行のみが取得できます。

```
SELECT * FROM view
WHERE DEREF(PersRefCol) IS OF (Student_typ);
```

現在、IS OF は SQL についてのみサポートされ、PL/SQL ではサポートされません。

SYS_TYPEID

SYS_TYPEID ファンクションを問合せで使用すると、引数としてファンクションに渡されたオブジェクト・インスタンスの最も狭い意味での型の**型 ID** が戻されます。

オブジェクト・インスタンスの**最も狭い意味での型**とは、インスタンスが属し、ルート型から最も離れたところから取り出される型です。たとえば、Tim が定時制の学生だとすると、彼は学生であると同時に個人ですが、彼の最も狭い意味での型は定時制の学生です。

このファンクションは、代入可能なすべての列と対応付けられた非表示の**型判別式の列**から型 ID を戻し、FINAL のルート型については NULL の型 ID を戻します。

このファンクションの構文は、次のとおりです。

```
SYS_TYPEID( object_type_value )
```

SYS_TYPEID ファンクションを使用するには、オブジェクト型の引数を付ける必要があります。非表示の型判別式の列について索引が作成できるようにすることが、このファンクションの主な目的です。

型階層に属するすべての型に、型階層内で一意な非 NULL 型 ID が割り当てられます。型階層に属さない型には、NULL 型 ID が与えられます。

FINAL のルート型を除くすべての型が、型階層に属します。FINAL のルート型には、継承により関連付けられる型はありません。

- これは FINAL の型のため、そこから導出されるサブタイプを持つことはできません。
- ルート型自身が他の型から導出されることはないため、ルート型のスーパータイプは存在しません。

> **参照：** 型識別式の列の詳細は、6-2 ページの「代入可能な列および表の非表示列」を参照してください。

SYS_TYPEID の例として、Person_typ の代入可能なオブジェクト表 persons を検討してみましょう。Person_typ は、サブタイプとして Student_typ を持ち、Student_typ のサブタイプとして PartTimeStudent_typ を持つ階層のルート型です。

```
CREATE TABLE persons OF Person_typ;

INSERT INTO persons
  VALUES (Person_typ(1243, 'Bob', '121 Front St'));

INSERT INTO persons
  VALUES (Student_typ(3456, 'Joe', '34 View', 12, 'HISTORY'));

INSERT INTO persons
  VALUES (PartTimeStudent_typ(5678, 'Tim', 13, 'PHYSICS', 20));
```

次の問合せでは、SYS_TYPEID を使用します。このファンクションは、persons 表の中にあるオブジェクト・インスタンスの name 属性および型 ID を取得します。インスタンスのそれぞれが、異なる型を持ちます。

```
SELECT name, SYS_TYPEID(VALUE(p)) typeid FROM persons p;

NAME   TYPEID
----   ------
Bob    01
Joe    02
Tim    03
```

次の問合せでは、books 表に格納されている著者の最も狭い意味での型 ID が戻されます。author は、Person_typ の代入可能な列です。

```
SELECT b.title, b.author.name, SYS_TYPEID(author) typeid FROM books b;

TITLE                     AUTHOR     TYPEID
----                      ------     ------
An Autobiography          Bob        01
Business Rules            Joe        02
Mixing School and Work    Tim        03
```

参照： 型判別式の列および他の非表示列の詳細は、第 6 章の「代入可能な列および表の非表示列」を参照してください。

3

Oracle プログラム環境のオブジェクト・サポート

Oracle9*i* では、SQL DDL コマンドを使用してオブジェクト型を作成し、DML コマンドを使用してオブジェクトを操作できます。オブジェクト・サポートは、次の Oracle のアプリケーション・プログラミング環境に組み込まれています。

- SQL
- PL/SQL
- Oracle Call Interface（OCI）
- Oracle C++ Call Interface（OCCI）
- Pro*C/C++
- Object Type Translator（OTT）
- Oracle Objects for OLE （OO4O）
- Java:JDBC、Oracle SQLJ、JPublisher および SQLJ オブジェクト型

SQL

Oracle SQL DDL では、オブジェクト型に対して次のようなサポートが提供されます。

- オブジェクト型、ネストした表および配列の定義
- 権限の指定
- ユーザー定義型の表の列の指定
- オブジェクト表の作成

Oracle SQL DML では、オブジェクト型に対して次のようなサポートが提供されます。

- オブジェクトおよびコレクションの問合せおよび更新
- REF の操作

> **参照：** Oracle SQL 構文の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

PL/SQL

オブジェクト型およびサブタイプは、組込み型が使用可能なほとんどの場所において、PL/SQL プロシージャおよびファンクションで使用できます。

PL/SQL ファンクションおよびプロシージャのパラメータおよび変数には、オブジェクト型を使用できます。

PL/SQL には、オブジェクト型と対応付けられたメソッドを実装できます。これらのメソッド（ファンクションおよびプロシージャ）は、ユーザーのスキーマの一部としてサーバーに格納します。

> **参照：** PL/SQL の詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

Oracle Call Interface（OCI）

OCI とは、アプリケーションが Oracle データベース内でデータおよびスキーマを操作するために使用できる、一連の C ライブラリ関数のことです。OCI では、次の項で説明するように、データベース・アクセスに従来の 3GL およびオブジェクト指向技法の両方がサポートされています。

OCI の重要なコンポーネントは、オブジェクト・キャッシュと呼ばれる作業領域を管理する一連のコールです。オブジェクト・キャッシュは、クライアント側のメモリー・ブロックです。このメモリー・ブロックにより、サーバーに対しさらにラウンドトリップを行わなくても、プログラムはオブジェクト全部を格納し、オブジェクト間をナビゲートできます。

オブジェクト・キャッシュは、それを使用するアプリケーションによって完全に制御および管理されます。Oracle サーバーは、オブジェクト・キャッシュにアクセスできません。オブジェクト・キャッシュを使用するアプリケーション・プログラムは、サーバーと一体となってデータ一貫性を保持し、同時発生する競合アクセスから作業領域を保護する必要があります。

OCI は、次の機能を備えています。

- SQL を使用して、サーバー上のオブジェクトにアクセスします。
- ポインタまたは REF を利用することによって、オブジェクト・キャッシュ内のオブジェクトにアクセスし、操作および管理します。
- Oracle の日付、文字列および数値を C のデータ型に変換します。
- オブジェクト・キャッシュのメモリー・サイズを管理します。

OCI は、オブジェクトを個々にロックできるようにして、同時実行性を改善します。また、複雑なオブジェクト検索をサポートして、パフォーマンスを改善します。

OCI 開発者は、OTT を使用して、Oracle オブジェクト型に対応する C のデータ型を生成できます。

> **参照：** OCI でオブジェクトを使用する場合の詳細は、『Oracle Call Interface プログラマーズ・ガイド』を参照してください。

OCI プログラムにおける連想アクセス

3GL プログラムは、SQL 文および PL/SQL プロシージャを実行し、リレーショナル・データベースに格納されたデータを操作します。データは、通常サーバー上で処理されるので、クライアント側にデータを転送するコストが発生しません。OCI では、オブジェクト・データを操作する SQL 文を実行する Application Program Interface（API）を提供することによって、この結合アクセスをサポートしています。特に、OCI では、次のことができます。

- オブジェクト・データおよびオブジェクト型スキーマ情報を操作する SQL 文を実行します。
- オブジェクト、オブジェクト参照（REF）およびコレクションを、SQL 文の入力変数として渡します。
- オブジェクト、REF およびコレクションを、SQL 文フェッチの出力として戻します。
- オブジェクト、REF およびコレクションを戻す SQL 文のプロパティを記述します。
- オブジェクトのパラメータまたは結果を持つ PL/SQL プロシージャまたはファンクションを記述し実行します。
- 拡張されたコミットおよびロールバック関数を介して、オブジェクト機能とリレーショナル機能を並用できます。

> **参照：** 3-8 ページの「Pro*C/C++ での連想アクセス」を参照してください。

OCI プログラムのナビゲーショナル・アクセス

オブジェクト指向のプログラミング・パラダイムでは、アプリケーションは、実社会のエンティティを、オブジェクト・グラフを形成する相互関連オブジェクトの集合としてモデル化します。オブジェクト間の関連は、参照として実装されます。アプリケーションは、いくつかのイニシャル・オブジェクトの集合から始めて、これらのイニシャル・オブジェクトの参照を使用して残りのオブジェクトにアクセスし、各オブジェクトで計算処理を実行することで、オブジェクトを処理します。OCI では、ナビゲーショナル・アクセスとして知られる、オブジェクト参照を使用してオブジェクト間のアクセスを行うための API が提供されています。特に、OCI では、次のことができます。

- クライアント・マシン上のメモリーにオブジェクトをキャッシュします。
- オブジェクト参照を解除し、オブジェクト・キャッシュ内の対応するオブジェクトを確保します。確保されたオブジェクトは、ホスト言語表現に透過的にマップされます。
- 確保されたオブジェクトが不要になった場合、キャッシュに通知します。
- 1 回のコールで、関連するオブジェクトをまとめてデータベースからクライアント・キャッシュにフェッチします。
- オブジェクトをロックします。
- キャッシュ上でオブジェクトを作成、更新および削除します。
- クライアント・キャッシュ上でオブジェクトに対して行われた変更を、データベースに反映します。

参照： 3-8 ページの「Pro*C/C++ でのナビゲーショナル・アクセス」を参照してください。

オブジェクト・キャッシュ

高パフォーマンスのオブジェクト・ナビゲーショナル・アクセスをサポートするために、OCI ランタイムでは、メモリーにオブジェクトをキャッシュするためのオブジェクト・キャッシュを提供しています。オブジェクト・キャッシュは、オブジェクト・キャッシュ内のデータベース・オブジェクトへの参照（REF）をサポートし、データベース・オブジェクトは、それらの参照を介して識別（確保）されます。オブジェクト・キャッシュは、データベース・オブジェクトに対して透過的で効率的なメモリー管理を提供するため、アプリケーションは、データベース・オブジェクトがキャッシュにロードされたときに、メモリーを割り当てたり解放する必要はありません。

さらに、データベース・オブジェクトは、キャッシュにロードされた際、ホスト言語に透過的にマップされます。たとえば、C 言語では、データベース・オブジェクトは対応する C 構造体へマップされます。オブジェクト・キャッシュは、キャッシュ上のオブジェクトと対応するデータベース・オブジェクトとの間の整合性を保ちます。トランザクションのコミット時にキャッシュ上のオブジェクト・コピーに対して行われた変更は、データベースに自動的に反映されます。

オブジェクト・キャッシュでは、REF をオブジェクトにマップするために、高速参照表がメンテナンスされます。アプリケーションが REF を参照解除するときに、対応するオブジェクトがまだキャッシュされていない場合は、オブジェクト・キャッシュからサーバーに自動的に要求が送信され、データベースからオブジェクトがフェッチされてオブジェクト・キャッシュにロードされます。同一の REF に対するその後の参照解除は、ローカル・キャッシュ・アクセスになり、ネットワークのラウンドトリップが発生しないため、より高速になります。キャッシュ内のオブジェクトにアクセス中であることをオブジェクト・キャッシュに通知するために、アプリケーションはオブジェクトを確保し、オブジェクトの処理が終わった時点で確保解除します。オブジェクト・キャッシュは、キャッシュの各オブジェクトの確保カウントをメンテナンスします。確保カウントは、確保コール（ピン・コール）で増加し、確保解除コール（アンピン・コール）で減少します。確保カウントが 0 になると、アプリケーションがそのオブジェクトを必要としなくなったことを意味します。オブジェクト・キャッシュでは最低使用頻度（LRU）アルゴリズムを使用して、キャッシュのサイズが管理されます。キャッシュが最大サイズに達すると、LRU アルゴリズムによって確保カウントが 0 の候補オブジェクトが解放されます。

オブジェクトを操作する OCI プログラムの作成

オブジェクトを操作する OCI プログラムを作成するときは、次の一般的な手順を実行する必要があります。

1. アプリケーション・オブジェクトに対応するオブジェクト型を定義します。
2. SQL DDL 文を実行して、必要なオブジェクト型をデータベースに定義します。
3. オブジェクト型をホスト言語形式で表します。

   たとえば、C プログラムでオブジェクト型のインスタンスを操作するには、それらの型を C 言語形式で記述する必要があります。そのためには、オブジェクト型を C 構造体で表します。OTT というツールを使用すると、オブジェクト型に対応する C の構造体を生成できます。OTT は、等価な C 構造体をヘッダー・ファイル（*.h）に作成します。これらの *.h ファイルを、アプリケーションを実装する C 関数を含む *.c ファイルにインポートします。
4. アプリケーションの *.c ファイルを OCI ライブラリとともにコンパイルおよびリンクすることによって、アプリケーションの実行可能ファイルを作成します。

参照： 6-26 ページの「オブジェクトに対する OCI のヒントおよび技法」を参照してください。

C でのユーザー定義コンストラクタの定義

C でユーザー定義コンストラクタを定義する場合は、PARAMETERS 句に SELF（オプションで、SELF TDO）を指定する必要があります。C 関数を入力すると、オブジェクトがマップされるすべての C 構造体の属性は、NULL に初期化されます。関数により戻される値は、ユーザー定義型のインスタンスにマップされます。

次に例を示します。

```
CREATE OR REPLACE TYPE person AS OBJECT
(
    name VARCHAR2(30),
    CONSTRUCTOR FUNCTION person(name VARCHAR2) RETURN SELF AS RESULT
);

CREATE OR REPLACE TYPE BODY person IS
    CONSTRUCTOR FUNCTION person(name VARCHAR2) RETURN SELF AS RESULT
    IS EXTERNAL NAME "cons_person_typ" LIBRARY person_lib WITH CONTEXT
    PARAMETERS(context, SELF, name OCIString, name INDICATOR sb4);
end;
```

SELF パラメータは、IN パラメータと同様にマップされるため、NOT FINAL 型の場合、(dvoid **) ではなく (dvoid *) にマップされます。

戻り値の TDO は、SELF の TDO と一致する必要があるため暗黙的です。戻り値は NULL になり得ないため、戻りインジケータも暗黙的です。

Oracle C++ Call Interface（OCCI）

OCCI は、C++ プログラミング言語のオブジェクト指向機能、ネイティブ・クラスおよびメソッドを使用して、Oracle データベースにアクセスできるようにする C++ API の 1 つです。

OCCI インタフェースは JDBC インタフェースを基に作成されているため、JDBC インタフェースのように扱いが簡単です。OCCI そのものが OCI の上に構築され、オブジェクト指向パラダイムを使用して OCI の能力およびパフォーマンスを提供します。

OCI は、Oracle データベース用の C API で、Oracle の全機能セットをサポートし、リレーショナル・データとオブジェクト・データへ効率的にアクセスします。ただし、特に複雑なオブジェクト・データ型を扱う場合には、問題が発生する可能性があります。オブジェクト型は C ではサポートされていないため、C でこれらの型をシミュレートするのは容易ではありません。OCCI は、OCI の機能に単純なオブジェクト指向インタフェースを持たせることで、この問題に取り組みます。OCCI では OCI 用のラッパー 1 組を定義して、この機能を提供します。このような高度な抽象化を扱うことにより、開発者は OCI の基盤となる機能を使用して、オブジェクト指向インタフェースを介してサーバーでオブジェクトを操作できます。このインタフェースは、非常にプログラムしやすくなっています。

OCCI は、次の 3 つの機能セットに分類できます。

- 結合リレーショナル・アクセス
- 結合オブジェクト・アクセス
- ナビゲーション・アクセス

OCCI 結合リレーショナルおよびオブジェクト・インタフェース

結合リレーショナル API およびオブジェクト・クラスにより、SQL にデータベースへアクセスする機能が提供されます。これらのインタフェースを介して SQL をサーバー上で実行し、オブジェクトまたはリレーショナル・データを作成および操作し、フェッチします。アプリケーションはサーバー上のどのデータ型にもアクセスできます。これらのデータ型の例は、次のとおりです。

- ラージ・オブジェクト
- オブジェクト / 構造型
- 配列
- 参照

OCCI ナビゲーショナル・インタフェース

ナビゲーショナル・インタフェースとは、SQL を使用せずに、C++ オブジェクトの形式で表現されたオブジェクト・リレーショナル・データにシームレスにアクセスし、修正できるようにする C++ インタフェースです。C++ オブジェクトは透過的にアクセスし、必要に応じてデータベースに格納します。

OCCI ナビゲーショナル・インタフェースを使用すると、オブジェクトを取り出し、参照によりオブジェクト間をナビゲートできます。サーバー・オブジェクトはアプリケーション・キャッシュ内部で C++ クラス・インスタンスとして具体化されます。

アプリケーションは OCCI オブジェクト・ナビゲーショナル・コールを使用して、サーバー・オブジェクトに対し次の機能を実行できます。

- オブジェクトの作成、アクセス、ロック、削除およびフラッシュ
- オブジェクトの参照の取得およびオブジェクトのナビゲート

> **参照：** Oracle C++ API を使用したアプリケーションの作成方法の詳細は、『Oracle C++ Call Interface プログラマーズ・ガイド』を参照してください。

Pro*C/C++

Oracle Pro*C/C++ プリコンパイラを使用すると、プログラマはオブジェクト型を C プログラムおよび C++ プログラムで使用できます。

Pro*C 開発者は、OTT を使用して、Oracle オブジェクト型およびコレクション型を、Pro*C アプリケーションで使用できる C のデータ型にマップできます。

Pro*C では、コンパイル時のオブジェクト型とコレクション型のタイプ・チェック、およびデータベースのデータ型から C のデータ型への自動型変換が行われます。

Pro*C には、オブジェクトを作成および破棄するための EXEC SQL 構文が含まれており、次の 2 つを使用してサーバーにあるオブジェクトにアクセスできます。

- Pro*C プログラム内に埋め込まれた SQL 文および PL/SQL ファンクションまたはプロシージャ。
- オブジェクト・キャッシュへのインタフェース（3-2 ページの「Oracle Call Interface (OCI)」を参照）。この場合、オブジェクトにはポインタを介してアクセスし、サーバー上で変更および更新できます。

> **参照：** Pro*C プリコンパイラの詳細は、『Pro*C/C++ Precompiler プログラマーズ・ガイド』を参照してください。

Pro*C/C++ での連想アクセス

連想アクセスのバックグラウンド情報については、3-3 ページの「OCI プログラムにおける連想アクセス」を参照してください。

Pro*C/C++ では、オブジェクトの連想アクセスに対して次の機能が提供されます。

- オブジェクト・キャッシュに割り当てられたオブジェクトの一時コピーのサポート
- 埋込み SQL の `INSERT`、`UPDATE`、`DELETE` 文、または `SELECT` 文の `WHERE` 句の入力ホスト変数として参照されるオブジェクトの一時コピーのサポート
- 埋込み SQL の `SELECT` および `FETCHS` 文で出力ホスト変数として参照されるオブジェクトの一時コピーのサポート
- オブジェクトの型およびスキーマ情報を取得するために `DESCRIBE` 文を介してオブジェクト型を参照する、ANSI 動的 SQL 文のサポート

Pro*C/C++ でのナビゲーショナル・アクセス

ナビゲーショナル・アクセスに関する基本的情報については、3-4 ページの「OCI プログラムのナビゲーショナル・アクセス」を参照してください。

Pro*C/C++ では、オブジェクトに対してさらにオブジェクト指向のインタフェースをサポートするために、次の機能が提供されます。

- 埋込み SQL の `OBJECT DEREF` 文を使用した、オブジェクト・キャッシュでのオブジェクトの参照解除、確保およびロック（オプション）のサポート
- オブジェクトが更新または削除された場合、または不要になった場合に、Pro*C/C++ ユーザーが埋込み SQL の `OBJECT UPDATE`、`OBJECT DELETE` および `OBJECT RELEASE` 文を使用してオブジェクト・キャッシュに通知できる機能
- 埋込み SQL の `OBJECT CREATE` 文を使用して、オブジェクト・キャッシュに新しい参照可能オブジェクトを作成するための機能
- オブジェクト・キャッシュでの変更を、埋込み SQL の `OBJECT FLUSH` 文を使用してサーバーにフラッシュするための機能

Oracle のオブジェクト型と C 言語のデータ型の間の変換

OTT によって生成されるオブジェクトの C 表現では、スカラー属性に対して OCIString や OCINumber などの、内部詳細が非表示にされた OCI 型が使用されます。コレクション型およびオブジェクト参照も、OCITable、OCIArray および OCIRef 型を使用して同様に表されます。この不透明な型を使用すると、これらの型の内部形式の変更が見えなくなるため、C または C++ アプリケーションでこれらの型を使用することはお薦めしません。
Pro*C/C++ では、C および C++ アプリケーションで OCI 型を簡単に使用できるように、次の拡張が行われています。

- 埋込み SQL の OBJECT GET 文を使用することで、オブジェクト属性を取得し暗黙的に C のデータ型に変換可能。
- 埋込み SQL の OBJECT SET 文を使用することで、オブジェクト属性を C のデータ型で設定可能。
- 埋込み SQL の COLLECTION GET 文を使用することで、コレクションをホスト配列にマップ可能。さらに、コレクションがスカラー型で構成されている場合、OCI 型は、暗黙的に C のデータ型に変換されます。
- 埋込み SQL の COLLECTION SET 文を使用することで、ホスト配列でコレクションの要素を更新可能。COLLECTION GET 文と同様、コレクションがスカラー型で構成されている場合、C のデータ型は、暗黙的に OCI 型に変換されます。

Object Type Translator （OTT）

OTT は、オブジェクト型に対応する C の構造体を自動的に生成するプログラムです。OTT は、Pro*C プリコンパイラおよび OCI サーバー・アクセス・パッケージをさらに使用しやすくします。

> **参照：** OTT の詳細は、『Oracle Call Interface プログラマーズ・ガイド』および『Pro*C/C++ Precompiler プログラマーズ・ガイド』を参照してください。

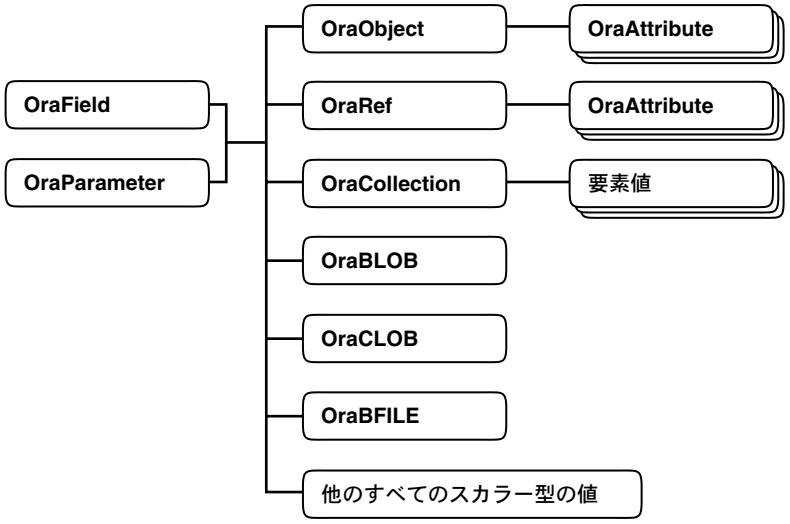
Oracle Objects for OLE （OO4O）

OO4O（Visual Basic、Excel、ActiveX、Active Server Pages 対応）では、Oracle データベース・サーバーにある REF のインスタンス、値インスタンス、可変長配列（VARRAY）およびネストした表へのアクセスおよび操作が完全にサポートされます。

> **参照：** OO4O で Oracle オブジェクトを使用する場合の詳細は、OO4O のオンライン・ヘルプを参照してください。

図 3-1 に、OO4O におけるすべての型の値インスタンスの抑制階層を示します。

図 3-1 サポートされている Oracle データ型



これらの型のインスタンスは、データベースからフェッチできるか、または SQL 文および（ストアド・プロシージャおよびファンクションを含む）PL/SQL ブロックに対する入力変数または出力変数として渡すことができます。すべてのインスタンスは、属性の動的アクセスおよび操作用のメソッドを提供する、COM オートメーション・インタフェースにマップされます。このインタフェースは、次のものから取得できます。

- Dynaset の OraField オブジェクトの値プロパティ
- SQL 文または PL/SQL ブロックで入力または出力パラメータとして使用される OraParameter オブジェクトの値プロパティ
- オブジェクトの属性（REF）
- コレクションの要素（VARRAY またはネストした表）

Visual Basic でのオブジェクトの表現形式（OraObject）

OraObject インタフェースは、Oracle 埋込みオブジェクトまたは値インスタンスの表現形式です。これには、値インスタンスの個々の属性にアクセスおよび操作（更新および挿入）するためのコレクション・インタフェース（OraAttributes）が含まれます。OraAttributes コレクション・インタフェースの個々の属性には、添字または属性の名前を使用して、アクセスできます。

次の Visual Basic の例では、person_tab 表の Address オブジェクトの属性にアクセスする方法を示します。

```
Dim Address OraObject
Set Person = OraDatabase.CreateDynaset("select * from person_tab", 0&)
Set Address = Person.Fields("Addr").Value
msgbox Address.Zip
msgbox.Address.City
```

Visual Basic での REF の表現形式（OraRef）

OraRef インタフェースは、Oracle オブジェクト参照（REF）を表すと同時に、クライアント・アプリケーションの参照可能なオブジェクトを表します。このオブジェクト属性には、OraObject インタフェースによって表されるオブジェクトの属性と同じ方法でアクセスできます。OraRef は、COM での抑制メカニズムを介して、OraObject インタフェースから導出されます。REF オブジェクトは、Dynaset など導出元となるコンテキストからは独立して、更新および削除できます。OraRef インタフェースでは、6-30 ページの「関連オブジェクトのプリフェッチ（複合オブジェクト検索）」で説明する OCI の複合オブジェクト検索機能（COR）を使用して、オブジェクト間の関連をたどる機能のカプセル化も行います。

Visual Basic での VARRAY およびネストした表の表現形式（OraCollection）

OraCollection インタフェースでは、OO4O において Oracle コレクション型、つまり可変長配列（VARRAY）およびネストした表にアクセスおよび操作するためのメソッドが提供されます。コレクションに含まれる要素には、添字でアクセスできます。

次の Visual Basic の例では、department 表の EnameList オブジェクトの属性にアクセスする方法を示します。

```
Dim EnameList OraCollection
Set Person = OraDatabase.CreateDynaset("select * from department", 0&)
set EnameList = Department.Fields("Enames").Value
'access all elements of the EnameList VArray
for I=1 to I=EnameList.Size
   msgbox EnameList(I)
Next I
```

Java:JDBC、Oracle SQLJ、JPublisher および SQLJ オブジェクト型

Java は、開発者に簡単で効率的でポータブルかつ安全なアプリケーション開発プラットフォームを提供する強力な最新のオブジェクト指向言語として登場しました。Oracle では、Oracle オブジェクト機能を Java と統合する 2 つの方法として JDBC および Oracle SQLJ が提供されています。これらのインタフェースにより、Java から SQL データにアクセスできるようになるのみでなく、Java オブジェクトの永続データベース記憶域が得られます。

Oracle オブジェクト・データへの JDBC アクセス

JDBC は、Oracle サーバーに接続するための Java インタフェースの集合です。Oracle では、オブジェクトと JDBC の間の緊密な統合が提供されます。SQL 型は Java クラスにマップでき、このマップ方法について、かなりの柔軟性が提供されています。

Oracle JDBC では、次のことができます。

- 動的 SQL を使用して、Java プログラム内でオブジェクト型およびコレクション型（データベースで定義）にアクセスできます。
- デフォルトまたはカスタマイズ可能なマッピングを使用して、データベースで定義されたデータ型を Java クラスに変換します。

JDBC 2.0 では、ユーザー定義型（オブジェクト型）などのオブジェクト・リレーショナル構造がサポートされています。JDBC は、Oracle オブジェクトを特定の Java クラスのインスタンスとして具体化します。JDBC を使用して Oracle オブジェクトにアクセスするには、Oracle オブジェクトについて Java クラスを作成し、これらのクラスを移入する必要があります。たとえば、次のような処理を行うことができます。

- JDBC にオブジェクトを `STRUCT` として具体化させます。この場合、JDBC は属性のクラスを作成し、作成したクラスを移入します。
- Oracle オブジェクトと Java クラスの間のマッピングを手動で指定します。つまり、Java クラスをオブジェクト・データ用にカスタマイズします。この後、ドライバはカスタマイズされた指定の Java クラスを移入します。これが Java クラスに対する一連の制約となります。これらの制約に従うために、クラスを `SQLData` インタフェースまたは `CustomDatum` インタフェースのどちらに合わせて定義するかを選択できます。

> **参照：** JDBC の詳細は、『Oracle9*i* JDBC 開発者ガイドおよびリファレンス』を参照してください。

Oracle オブジェクト・データへの SQLJ アクセス

SQLJ では、Java コードに埋め込んだ SQL 文を使用して、サーバー・オブジェクトへアクセスできます。

- Java プログラムでオブジェクト型を使用可能です。
- JPublisher を使用して、オブジェクト型およびコレクション型を、アプリケーションで使用できる Java クラスにマップします。
- SQL 文のオブジェクト型およびコレクション型は、コンパイル時にチェックされます。

> **参照：** SQLJ の詳細は、『Oracle9*i* Java Developer's Guide』を参照してください。

データ・マッピング方法の選択

Oracle SQLJ では、イテレータまたはホスト式で使用するオブジェクト型、参照型（REF）およびコレクション型（VARRAY およびネストした表）の強力な型指定または緩い型指定の Java 表現がサポートされます。

強力な型指定の表現では、特定のオブジェクト型、REF 型またはコレクション型に対応するカスタム Java クラスを使用し、インタフェース oracle.sql.CustomDatum を実装する必要があります。Oracle JPublisher ユーティリティでは、そのようなカスタム Java クラスを自動的に生成します。

緩い型指定の表現では、クラス oracle.sql.STRUCT（オブジェクト用）、oracle.sql.REF（参照用）または oracle.sql.ARRAY（コレクション用）を使用します。

JPublisher を使用した JDBC および SQLJ プログラム用 Java クラスの作成

Oracle では、オブジェクト型、参照型およびコレクション型を Java クラスにマップすることで、強力な型指定によるすべての利点を得ることができます。たとえば、次のような処理を行うことができます。

- Oracle JPublisher を使用してカスタム Java クラスを自動的に生成し、それらのクラスを変更なしで使用します。
- JPublisher によって生成されたクラスをサブクラス化し、独自の専用 Java クラスを作成します。
- 『Oracle9*i* SQLJ 開発者ガイドおよびリファレンス』に記述されている要件をクラスが満たす場合は、JPublisher を使用せず、カスタム Java クラスを手動でコーディングできます。

生成されたクラスが十分な機能を果たさない場合は、JPublisher およびサブクラスを使用することをお薦めします。

JPublisher で生成される内容

オブジェクト型に対して JPublisher を実行する場合、次のものが自動的に作成されます。

- ユーザーの Oracle オブジェクト型に対応する型定義として動作するカスタム・オブジェクト・クラス

  このクラスには、各属性に対する取得メソッドおよび設定メソッドが含まれます。メソッド名は、属性 foo の場合は、getFoo() および setFoo() の形式をとります。

  また、オプションで、サーバーで実行される Oracle オブジェクト・メソッドを起動するラッパー・メソッドをクラスに作成するように指定できます。

- ユーザーの Oracle オブジェクト型に対するオブジェクト参照用のカスタム・リファレンス・クラス

このクラスには、ユーザーのカスタム・オブジェクト・クラスのインスタンスを戻す `getValue()` メソッドおよびデータベースのオブジェクト値を更新する `setValue()` メソッドが含まれます。これらのメソッドは、入力としてカスタム・オブジェクト・クラスのインスタンスをとります。

コレクション型に対して JPublisher を実行する場合、次のものが自動的に作成されます。

- ユーザーの Oracle コレクション型に対応する型定義として動作するカスタム・コレクション・クラス

  このクラスには、コレクション全体を取得または更新する、オーバーロード型の `getArray()` メソッドおよび `setArray()` メソッド、コレクションの個々の要素を取得または更新する `getElement()` メソッドおよび `setElement()` メソッド、その他のユーティリティ・メソッドが含まれます。

これらのどのカテゴリでも、JPublisher が生成するカスタム Java クラスでは、`CustomDatum` インタフェース、`CustomDatumFactory` インタフェースおよび `getFactory()` メソッドが実装されます。

> **参照：** JPublisher の使用方法の詳細は、『Oracle9*i* JPublisher ユーザーズ・ガイド』を参照してください。

Java オブジェクトの記憶域

JPublisher を使用すると、既存の SQL 型にマップする Java クラスを作成できます。このように Java クラスを作成した後、JDBC により Java アプリケーションから SQL 型にアクセスできるようになります。

Oracle9*i* では、これとは逆に、既存の Java クラスにマップする SQL 型を作成することができます。これにより、Java オブジェクト用の永続ストレージが用意できます。これらの SQL 型を、Java 言語の SQL 型または SQLJ オブジェクト型といいます。これらの SQL 型は、オブジェクト、属性、列、またはオブジェクト表の行の型として使用できます。オブジェクト参照または外部キーを介して、これらのタイプのオブジェクト（Java オブジェクト）に対して、ナビゲーショナル・アクセスを行ったり、SQL からこれらのオブジェクトの問合せや操作ができます。

他のユーザー定義型と同様に、SQLJ 型も `CREATE TYPE` 文を使用して作成します。SQLJ 型の場合、次の 2 つの特別な要素が `CREATE TYPE` 文に追加されます。

- それぞれの SQLJ 属性とメソッドに対応する Java の属性とメソッド、および SQLJ 型そのものに対応する Java クラスの識別に使用する `EXTERNAL NAME` 句。
- サーバーに対して SQLJ 型をどのように表現するかを指定するための `USING` 句。`USING` 句は、SQLJ 型およびストレージの種類を取得するために使用するインタフェースを指定します。

次に例を示します。

```
CREATE TYPE person_t AS OBJECT
  EXTERNAL NAME 'Person' LANGUAGE JAVA
```

```
  USING SQLData (
    ss_no NUMBER (9) EXTERNAL NAME 'socialSecurityNo',
    name varchar(100) EXTERNAL NAME 'name',
    address full_address EXTERNAL NAME 'addrs',
    birth_date date EXTERNAL NAME 'birthDate',
    MEMBER FUNCTION age () RETURN NUMBER EXTERNAL NAME 'age () return int',
    MEMBER FUNCTION address RETURN full_address EXTERNAL NAME 'get_address ()
      return long_address',
    STATIC create RETURN person_t EXTERNAL NAME 'create () return Person',
    STATIC create (name VARCHAR(100), addrs full_address, bDate DATE)
      RETURN person_t EXTERNAL NAME 'create (java.lang.String, Long_address,
      oracle.sql.date) return Person',
    ORDER FUNCTION compare (in_person person_t) RETURN NUMBER
      EXTERNAL NAME 'isSame (Person) return int'
  )
/
```

SQLJ 型は、対応する Java クラスを型の本体として使用します。ただし、SQL では、通常のオブジェクト型のように、型のメソッドの実装を格納する目的で型の本体は指定しません。

サーバーに対する SQLJ 型の表現

SQLJ 型をサーバーに対してどのように表現し、格納するかは、対応する Java クラスが実装するインタフェースによって決まります。現在、Oracle では、SQLData、CustomDatum または ORAData インタフェースを実装する Java クラスに対してのみ、SQLJ 型の表現がサポートされます。サーバーに対して、これらの Java クラスは表現され、SQL を介してアクセス可能になります。java.io.Serializable インタフェースを実装する Java クラスの表現は、現在サポートされていません。

SQL 表現では、型の属性は、通常のオブジェクト型の属性と同様に、列に格納されます。この表現の場合、オブジェクトは SQL 文を使用してアクセスおよび操作しますが、オブジェクト・データの整合性を確保するために、トリガーおよび制約が使用できるため、属性はすべてパブリックです。

SQL 表現の場合、USING 句により SQLData、CustomDatum、ORAData のいずれかを指定する必要があります。また、これらのインタフェースの 1 つを対応する Java クラスにより実装する必要があります。属性用の EXTERNAL NAME 句は、任意で使用してください。

SQLJ オブジェクト型の作成

SQLJ 型を作成し、そのマッピングを指定する SQL 文は、**ディプロイメント・ディスクリプタ**と呼ばれるファイルの中に収められます。関連する SQL 制約および権限も、このファイルに指定されます。型は、ファイルが実行されるときに作成されます。

次に、Java タイプおよび Java クラスの SQL バージョンを作成するプロセスの概要を示します。

1. Java タイプを設計します。

2. Java クラスを作成します。
3. SQLJ オブジェクト型の文を作成します。
4. JAR ファイルを作成します。これは、必要なクラスがすべて入っている 1 つのファイルです。
5. loadjava ユーティリティを使用して、JAR ファイルに定義されている Java クラスをインストールします。
6. 文を実行して、SQLJ オブジェクト型を作成します。

SQLJ オブジェクト型マッピングのサンプル

次のコードは、2 つの Java クラスを定義しています。その後に続くコードは、ディプロイメント・ディスクリプタに入るような対応する CREATE TYPE 文を示しています。このコードは、SQL 型に対する Java クラスの 1 対 1 マッピングを定義します。この場合、すべての Java フィールドは SQL 型の属性にマップされます。

```
package Examples;

import java.sql.*;
//import oracle.jdbc2.*;

// Java Address class based on SQLJ part 2

public class Address implements SQLData {
  protected String street;
  protected String city;
  protected String state;
  protected int zipCode;
  protected String sql_type;
  public static int recommendedWidth = 250;

  public Address () {
    street = "Unknown";
    city = "somewhere";
    state = "nowhere";
    zipCode = 0;
  }

  public Address (String st, String cit, String stt, int zip) {
    street = st;
    state = stt;
    city = cit;
    zipCode = zip;
  }

  protected static String strip(String in) {
```

```
  int len;
  int i;

  if (in == null) return in;
  if (in.charAt (0) != ' ') return in;

  len = in.length();

  for (i = 0; i < len && in.charAt(i) == ' '; i++) {}

  if (i == len) return null;

  return in.substring (i, len);
}

public String getSQLTypeName() throws SQLException
{
  return sql_type;
}

public void readSQL(SQLInput stream, String typeName)
  throws SQLException
{
  sql_type = typeName;

  street = stream.readString();
  city = stream.readString();
  state = stream.readString();
  zipCode = stream.readInt();
}

public void writeSQL(SQLOutput stream)
  throws SQLException
{
  stream.writeString(street);
  stream.writeString(city);
  stream.writeString(state);
  stream.writeInt(zipCode);
}

public static Address create () {
  return new Address() ;
}

public static Address create (String st, String cit, String stt, int zp) {
  return new Address(st, cit, stt, zp);
```

```
  }

  public String toString() {
    return "Street" + street + "City" + city + "State" + state + zipCode ;
  }

  public Address removeLeadingBlanks () {
    // The definition of the Misc class has been omitted in this example.
    // Misc is fully described in the SQLJ part 2 specification.

    street = strip (street);
    city = strip (city) ;
    state = strip (state);
    return this;
  }
}
// create LongAddress as subclass of Address

public class LongAddress extends Address {

  protected String street2;
  protected String country;
  protected String addrCode ;

  public LongAddress () {

    super();
    street2 = " ";
    country = " ";
    addrCode = " ";
  }

  public LongAddress
  (String st, String st2, String ct, String stt, String cntry, String cd){
    street = st;
    street2 = st2;
    state = stt;
    country = cntry;
    city = ct;
    zipCode = 0;
    addrCode = cd;
  }

  public void readSQL(SQLInput stream, String typeName)
    throws SQLException
  {
    sql_type = typeName;
```

```
  street = stream.readString();
  city = stream.readString();
  state = stream.readString();
  zipCode = stream.readInt();
  street2 = stream.readString();
  country = stream.readString();
  addrCode = stream.readString();

}

public void writeSQL(SQLOutput stream)
  throws SQLException
{
  stream.writeString(street);
  stream.writeString(city);
  stream.writeString(state);
  stream.writeInt(zipCode);
  stream.writeString(street2);
  stream.writeString(country);
  stream.writeString(addrCode);
}

public static Address create () {
  return new LongAddress();
}

public static Address create (
  String st, String st2, String ct, String stt, String cntry, String cd){
  return new LongAddress (st, st2, ct, stt, cntry, cd);
}

public String toString () {
  if (zipCode != 0)
    return "Street " + street + "City " + city + "State " + state +
      "Zip" + zipCode + "USA";
  else
    return "Street " + street + street2 + "City " + city + "State " +
      state + "Country " + country + addrCode ;
}

public Address removeLeadingBlanks () {
  // Misc class is not defined please refer to the SQLJ specs
  street = strip (street);
  street2 = strip (street2);
  city = strip (city) ;
  state = strip (state);
```

```
    country = strip (country);
    addrCode = strip (addrCode);
    return this;
  }
}
```

次のコードは、ディプロイメント・ディスクリプタに入り、SQLJ 型を前述のコードで定義された Java クラスと対応させるために、SQLJ 型を作成します。

```
CREATE TYPE address_t AS OBJECT
  EXTERNAL NAME 'Examples.Address' LANGUAGE JAVA
  USING SQLData(
    street_attr varchar(250) EXTERNAL NAME 'street',
    city_attr varchar(50) EXTERNAL NAME 'city',
    state varchar(50) EXTERNAL NAME 'state',
    zip_code_attr number EXTERNAL NAME 'zipCode',
    STATIC FUNCTION recom_width RETURN NUMBER
      EXTERNAL VARIABLE NAME 'recommendedWidth',
    STATIC FUNCTION create_address RETURN address_t
      EXTERNAL NAME 'create() return Examples.Address',
    STATIC FUNCTION construct RETURN address_t
      EXTERNAL NAME 'create() return Examples.Address',
    STATIC FUNCTION create_address (street VARCHAR, city VARCHAR,
        state VARCHAR, zip NUMBER) RETURN address_t
      EXTERNAL NAME 'create (java.lang.String, java.lang.String,
        java.lang.String, int) return Examples.Address',
    STATIC FUNCTION construct (street VARCHAR, city VARCHAR,
        state VARCHAR, zip NUMBER) RETURN address_t
      EXTERNAL NAME
        'create (java.lang.String, java.lang.String, java.lang.String, int)
        return Examples.Address',
    MEMBER FUNCTION to_string RETURN VARCHAR
      EXTERNAL NAME 'tojava.lang.String() return java.lang.String',
    MEMBER FUNCTION strip RETURN SELF AS RESULT
      EXTERNAL NAME 'removeLeadingBlanks () return Examples.Address'
  ) NOT FINAL;
/
CREATE OR REPLACE TYPE long_address_t
UNDER address_t
EXTERNAL NAME 'Examples.LongAddress' LANGUAGE JAVA
USING SQLData(
    street2_attr VARCHAR(250) EXTERNAL NAME 'street2',
    country_attr VARCHAR (200) EXTERNAL NAME 'country',
    address_code_attr VARCHAR (50) EXTERNAL NAME 'addrCode',
    STATIC FUNCTION create_address RETURN long_address_t
      EXTERNAL NAME 'create() return Examples.LongAddress',
    STATIC FUNCTION  construct (street VARCHAR, city VARCHAR,
```

```
      state VARCHAR, country VARCHAR, addrs_cd VARCHAR)
    RETURN long_address_t
    EXTERNAL NAME
      'create(java.lang.String, java.lang.String, java.lang.String,
      java.lang.String, java.lang.String) return Examples.LongAddress',
  STATIC FUNCTION construct RETURN long_address_t
    EXTERNAL NAME 'Examples.LongAddress() return Examples.LongAddress',
  STATIC FUNCTION create_longaddress (
    street VARCHAR, city VARCHAR, state VARCHAR, country VARCHAR,
    addrs_cd VARCHAR) return long_address_t
    EXTERNAL NAME
      'Examples.LongAddress (java.lang.String, java.lang.String,
      java.lang.String, java.lang.String, java.lang.String)
      return Examples.LongAddress',
  MEMBER FUNCTION get_country RETURN VARCHAR
    EXTERNAL NAME 'country_with_code () return java.lang.String'
);
/
```

マッピングの詳細

- SQLJ 静的ファンクションは、Java クラスのユーザー定義コンストラクタにマップできます。このファンクションの戻り値は、ファンクションをローカルに定義したユーザー定義型をとります。
- Java の静的変数は、対応する静的変数の値を返す SQLJ スタティック・メソッドにマップします。対応する静的変数は、EXTERNAL NAME で識別します。SQLData、CustomDatum または ORAData 表現の場合、属性用の EXTERNAL NAME 句は任意で使用します。
- SQL 表現の SQLJ 型のどの属性も、Java フィールドにマップする必要があります。ただし、必ずしもすべての Java フィールドを、対応する SQLJ 属性にマップする必要はありません。マッピングでは Java フィールドが省略できます。
- クラスは省略できます。SQLJ 型をルート・クラスにマップせず、またスーパークラスを介在させずに、SQLJ 型を Java クラス階層内の非ルート・クラスにマップできます。このようにマップすると、スーパークラスから継承した属性およびメソッドを含んだまま、スーパークラスを非表示にできます。

  ただし、クラス階層内のノード間の構造対応関係および SQLJ 型階層に存在するノード間の構造対応関係はそのまま保つ必要があります。つまり、継承により関連付け、2 つの SQL 型である s_A および s_B にマップする 2 つの Java クラス j_A および j_B の場合、j_A から j_B への継承パスのそれぞれのノードについて、s_A から s_B への継承パスに対応するノードが 1 つ存在する必要があります。

- 前述の制限に違反しないかぎり、Java クラスを複数の SQLJ 型にマップできます。つまり、同一の Java クラスにマップされた 2 つの SQLJ 型は、共通するスーパータイプの祖先クラスを持てないということです。
- すべての Java クラスを SQLJ 型にマップしなければ、SQLJ オブジェクト型の属性を、マップされていない Java クラス（つまり、継承階層において属性がマップされたクラスの上位または下位で発生するクラス）のオブジェクトに設定することが可能です。オブジェクトのクラスが、属性の型 / クラスのスーパークラスの場合は、エラーとなります。オブジェクトのクラスが、属性の型 / クラスのサブクラスであれば、SQL マッピングが存在するオブジェクトの階層の最も狭い意味での型に、オブジェクトがマップされます。

SQLJ 型の進化

`ALTER TYPE` 文を使用すると、属性またはメソッドの追加、削除などを行うことにより、型を進化させられます。

SQLJ 型を進化させる場合は、クラスと型間のマッピングを確認するために、特別な検証を行います。クラスと進化させた型が一致した場合は、有効な型であると表示されます。一致しない場合は、検証待ちと表示されます。

検証待ちという表示は、無効という表示と同じではありません。検証待ちの型は、たとえば `ALTER TYPE` および `GRANT` 文を使用して、操作を継続できます。

SQL 表現を持つ型が、検証待ちと表示されても、メソッド起動を必要としない DML または SELECT 文を使用して、この型の表にアクセスできます。

ただし、シリアル化可能な表現を持ち、検証待ちと表示された型の表に対しては、DML や SELECT 文は実行できません。シリアル化可能な型のデータに対しては、メソッド起動を介したナビゲーショナル・アクセスのみ許可されます。検証待ちの型の場合は、このタイプのアクセスはできません。ただし、検証に合格するまで型をさらに進化させることは可能です。

参照： 6-7 ページの「型進化」を参照してください。

制約

SQL 表現を持つ SQLJ 型の場合、通常のオブジェクト型の場合と同じ制約を定義できます。

制約を定義する対象は、型ではなく表で、列レベルで定義します。SQL 表現を持つ SQLJ 型についてサポートされている制約は、次のとおりです。

- 一意制約
- 主キー
- CHECK 制約
- 属性についての `NOT NULL` 制約
- 参照制約

SQL 表現を持つ SQLJ 型の場合は、列の代入性についての IS OF type 制約もサポートされます。

参照： 2-44 ページの「代入性の制約」を参照してください。

SQLJ オブジェクトの問合せ

SQLJ 型に対しては、通常の SQL オブジェクト型と同様に問合せを行うことができます。

SELECT 文でコールされたメソッドで、属性値を変更することは避けてください。

Java オブジェクトの挿入

SQLJ 型の列が入っている表に、行を挿入するには、この型のコンストラクタ・ファンクションをコールし、この型の Java オブジェクトを作成する必要があります。

システムが生成する暗黙的なコンストラクタを使用するか、または Java クラスのユーザー定義コンストラクタにマップする静的ファンクションを定義してください。

SQLJ オブジェクトの更新

UPDATE 文を使用して、複数の属性の値を変更するか、または属性を更新し SELF を戻すメソッド（つまり、変更を行ったうえでオブジェクトそのものを戻すメソッド）を起動すると、SQLJ オブジェクトを更新できます。

raise() が、指定された分だけ salary フィールド / 属性を増加し、SELF を戻すメンバー・ファンクションである場合について考えてみます。次の文により、オブジェクト表 employee_objtab に含まれているすべての従業員に 1000 分の増額が与えられます。

```
UPDATE employee_objtab SET c=c.raise(1000);
```

SQLJ 型の列は、通常のオブジェクト型と同じ構文を使用して、NULL に設定するか、または別の列に設定できます。たとえば、次の文により、列 c に列 d が割り当てられます。

```
UPDATE employee_reltab SET c=d ;
```

Java でのユーザー定義コンストラクタの定義

Java でユーザー定義コンストラクタを実装する場合、実装ルーチンとして提供される文字列は、静的ファンクションに対応する必要があります。ファンクションの戻り型には、SQL 型にマップされる Java 型を指定します。

次に、Java で実装されたユーザー定義コンストラクタを伴う型宣言の例を示します。

```
CREATE OR REPLACE TYPE Person1_typ AS OBJECT (
  name VARCHAR2(30),
  age NUMBER
  CONSTRUCTOR FUNCTION Person1_typ(name VARCHAR2, age NUMBER)
    RETURN Person1_typ AS RESULT
  AS LANGUAGE JAVA
    NAME 'pkg1.J_Person.J_Person(java.lang.String, int) return J_Person'
);
```

4

Oracle オブジェクトの管理

この章では、Oracle がデータベースの他の部分と関連してどのように機能し、どのように DML 操作および DDL 操作を実行するかについて説明します。内容は次のとおりです。

- オブジェクト型およびそれらのメソッドの権限
- 依存性および不完全な型
- ユーザー定義型のシノニム
- Oracle のツール製品
- ユーティリティ

オブジェクト型およびそれらのメソッドの権限

オブジェクト型に対する権限は、システム・レベルおよびスキーマ・オブジェクト・レベルで存在します。

システム権限

Oracle では、オブジェクト型に次のシステム権限を定義します。

- CREATE TYPE: 自分のスキーマにオブジェクト型を作成できます。
- CREATE ANY TYPE: 任意のスキーマにオブジェクト型を作成できます。
- ALTER ANY TYPE: 任意のスキーマ内のオブジェクト型を変更できます。
- DROP ANY TYPE: 任意のスキーマ内のオブジェクト型を削除できます。
- EXECUTE ANY TYPE: 任意のスキーマ内のオブジェクト型を使用および参照できます。
- UNDER ANY TYPE: NOT FINAL オブジェクト型の下に任意のサブタイプを作成できます。
- UNDER ANY VIEW: 任意のオブジェクト・ビューの下にサブビューを作成できます。

CONNECT ロールおよび RESOURCE ロールには、CREATE TYPE システム権限が含まれます。DBA ロールには、前述のすべての権限が含まれます。

スキーマ・オブジェクト権限

オブジェクト型には、2 つのスキーマ・オブジェクト権限が適用されます。

- オブジェクト型に EXECUTE 権限があると、その型を次のように使用できます。
  - 表の定義
  - リレーショナル表の列の定義
  - オブジェクト型の変数またはパラメータの宣言

  EXECUTE は、コンストラクタなどの型のメソッドを起動します。

  メソッド実行および対応する権限は、ストアド PL/SQL プロシージャのものと同じです。
- UNDER は、権限が与えられた型 / ビューの下にサブタイプ / サブビューを作成します。

  権限付与者が直系のスーパータイプまたはスーパービューについて UNDER 権限を With Grant Option 付きで持っている場合のみ、サブタイプまたはサブビューに対する UNDER 権限が付与されます。

WITH HIERARCHY OPTION 句は、オブジェクトのすべてのサブオブジェクトについて指定されたオブジェクト権限を付与します。このオプションが意味を持つのは、オブジェクト・

ビュー階層内のオブジェクト・ビューに SELECT オブジェクト権限が付与されている場合のみです。この場合、権限が付与されているビューのサブビューすべてに、権限が適用されます。

新しい型または表での型の使用

次の場合は、前述の項で説明した権限の他に、特定の権限が必要です。

- 別のユーザーが作成した型を使用する型または表を作成する場合
- 新しく作成した型または表の使用権限を別のユーザーに付与する場合

新しい型または表を定義するには、EXECUTE ANY TYPE システム権限、または使用する型に対する EXECUTE オブジェクト権限が必要です。これらの権限は、ロールを介してではなく、明示的に付与される必要があります。

新しい型または表へのアクセス権限を他のユーザーに付与するには、Grant Option を含む適切な EXECUTE オブジェクト権限、またはオプション With Admin Option を含む EXECUTE ANY TYPE システム権限が必要です。これらの権限は、ロールを介してではなく、明示的に付与される必要があります。

例

CONNECT ロールおよび RESOURCE ロールを持つ USER1、USER2、USER3 という 3 人のユーザーがいると想定します。

USER1 は、USER1 スキーマ内で次の DDL を実行します。

```
CREATE TYPE type1 AS OBJECT ( attr1 NUMBER );
CREATE TYPE type2 AS OBJECT ( attr2 NUMBER );
GRANT EXECUTE ON type1 TO user2;
GRANT EXECUTE ON type2 TO user2 WITH GRANT OPTION;
```

USER2 は、USER2 スキーマ内で次の DDL を実行します。

```
CREATE TABLE tab1 OF user1.type1;
CREATE TYPE type3 AS OBJECT ( attr3 user1.type2 );
CREATE TABLE tab2 (col1 user1.type2 );
```

USER2 は、Grant Option を含む、USER1 の TYPE2 に対する EXECUTE 権限を持っているため、次の文は正常に実行されます。

```
GRANT EXECUTE ON type3 TO user3;
GRANT SELECT on tab2 TO user3;
```

ただし、USER2 は、Grant Option を含む、USER1.TYPE1 に対する EXECUTE 権限を持っていないため、次の文は正常に実行されません。

```
GRANT SELECT ON tab1 TO user3;
```

USER3 は、次の処理を正常に実行できます。

```
CREATE TYPE type4 AS OBJECT (attr4 user2.type3);
CREATE TABLE tab3 OF type4;
```

オブジェクト型へのアクセスおよびオブジェクト・アクセスの権限

オブジェクト型は EXECUTE 権限のみを使用しますが、オブジェクト表はリレーショナル表と同じ次のすべての権限を使用します。

- SELECT: 表から 1 つのオブジェクトおよびその属性にアクセスできます。
- UPDATE: 表内のオブジェクトの属性を変更できます。
- INSERT: 表に新しいオブジェクトを追加できます。
- DELETE: 表からオブジェクトを削除できます。

同様の表権限および列権限が、オブジェクト型の表列の使用を規制します。

オブジェクト表の列を選択するには、オブジェクト表の型に対する権限は必要ありません。ただし、行全体を選択するには必要です。

次のスキーマについて考えてみます。

```
CREATE TYPE emp_type as object (
  eno    NUMBER,
  ename  CHAR(31),
  eaddr  addr_t );

CREATE TABLE emp OF emp_type;
```

次の 2 つの問合せについても考えてみます。

```
SELECT VALUE(e) FROM emp e;
SELECT eno, ename FROM emp;
```

どちらの問合せの場合も、Oracle は emp 表に対するユーザーの SELECT 権限をチェックします。最初の問合せの場合、データを解析するには、ユーザーは emp_type 型情報を取得する必要があります。問合せが emp_type 型にアクセスすると、Oracle はユーザーの EXECUTE 権限をチェックします。

2 番目の問合せの実行は、指定された型を必要としません。そのため、Oracle は型権限をチェックしません。

さらに、前述の項のスキーマを使用して、USER3 は次の問合せを実行できます。

```
SELECT tab1.col1.attr2 from user2.tab1 tab1;
SELECT t.attr4.attr3.attr2 FROM tab3 t;
```

USER3 による 2 つの選択では、USER3 は基礎となる型の明示的な権限を持っていませんが、型所有者および表所有者は Grant Option を含む必要な権限を持っているため、この文は正常に実行されます。

Oracle は次の要求についての権限をチェックし、ユーザーに各処理に対する権限がない場合はエラーを戻します。

- REF 値を使用してオブジェクト・キャッシュ内に 1 つのオブジェクトを確保すると、Oracle は、そのオブジェクトを含むオブジェクト表に対する SELECT 権限、およびそのオブジェクト型に対する EXECUTE 権限をチェックします（OCI オブジェクト・キャッシュの詳細は、6-26 ページの「オブジェクトに対する OCI のヒントおよび技法」を参照してください）。
- 既存オブジェクトの変更、またはオブジェクト・キャッシュからのオブジェクトのフラッシュを行うと、Oracle は、接続先オブジェクト表に対する UPDATE 権限をチェックします。新しいオブジェクトをフラッシュすると、Oracle は、接続先オブジェクト表に対する INSERT 権限をチェックします。
- オブジェクトを削除すると、Oracle は、接続先の表に対する DELETE 権限をチェックします。
- メソッドを起動すると、Oracle は、対応するオブジェクト型に対する EXECUTE 権限をチェックします。

Oracle は、オブジェクト表に対する列レベルの権限は提供しません。

依存性および不完全な型

型は、互いに依存するように定義できます。たとえば、employee の 1 つの属性を従業員が属する部門にし、department の 1 つの属性を部門を管理する従業員にするように、オブジェクト型 employee および department を定義できます。

このように、直接的または中間の型を介して互いに依存する型を、相互依存といいます。矢印を使用して一連の型の依存関係を示す図の場合、相互依存型同士のつながりがループを形成します。

そのような循環依存を定義するには、サークルの最小限 1 つのセグメントに REF を使用する必要があります。

たとえば、次の型を定義できます。

```
CREATE TYPE department;

CREATE TYPE employee AS OBJECT (
  name    VARCHAR2(30),
  dept    REF department,
  supv    REF employee );

CREATE TYPE emp_list AS TABLE OF employee;
```

```
CREATE TYPE department AS OBJECT (
  name    VARCHAR2(30),
  mgr     REF employee,
  staff   emp_list );
```

これは、適切な相互依存型における SQL DDL 文の適切な順序です。Oracle は、これをエラーなしでコンパイルします。

前述のコードは department 型を 2 回作成することに注意してください。次の文はオプションです。

```
CREATE TYPE department;
```

この文は、employee の REF 属性が指すプレースホルダとして機能する department の不完全な宣言です。AS OBJECT 句が省略され、属性またはメソッドがリストされていない点で、この宣言は不完全です。これらは、後述の型を完成させる完全な宣言で指定します。ここでは、department を不完全なオブジェクト型として作成しておきます。これにより、employee がエラーなしでコンパイルできます。

不完全な型を再定義するには、この例の最後に示すとおり、型の属性およびメソッドを指定する CREATE TYPE 文を実行します。不完全な型が参照するすべての型を作成した後、不完全な型を再定義してください。

不完全な型をプレースホルダとして作成しない場合、その型を参照する型はコンパイルされますが、コンパイル・エラーが発生します。

たとえば、department が存在しない場合、Oracle はそれを不完全な型として作成し、employee はコンパイル・エラーになります。この場合、次回なんらかの操作がこの型にアクセスしようとした際に、employee が再コンパイルされます。この型が依存するすべての型が作成され、その依存性が完全な場合、エラーなしでコンパイルされます。

不完全な型を使用することにより、まだ作成されていないサブタイプに対する REF 属性を格納する型も作成できます。こうしたスーパータイプを作成するには、参照するサブタイプの不完全な型を最初に作成します。完全なサブタイプは、スーパータイプ作成後に作成します。

サブタイプは、直系のスーパータイプの特殊なバージョンであるため、必然的に直系のスーパータイプとの間に明らかな依存関係を持ちます。スーパータイプを削除した後、サブタイプの削除もれが発生しないようにするために、最初にすべてのサブタイプを削除する必要があります。スーパータイプは、そのすべてのサブタイプが削除されるまで、削除できません。

不完全な型の再定義

不完全な型が参照するすべての型が作成された場合は、不完全な型が不完全な状態を維持する必要がなくなるため、型の宣言を再定義してください。型を再定義すると、型が再コンパイルされ、システムが様々なロックを解放できるようになります。

不完全なオブジェクト型は、オブジェクト型として再定義する必要があります。オブジェクト型をコレクション型（ネストした表型または配列型）として再定義することはできません。その型を削除することはできます。

明示的に作成しなかったために Oracle によって作成される不完全な型も、再定義する必要があります。前述の例は、department を明示的に不完全な型として作成しています。department を明示的に不完全な型として作成しなかった場合、employee 型が（エラー付きで）コンパイルできるように、Oracle はこの型を明示的に不完全な型として作成します。department の宣言は、この型を不完全な型として宣言したのがユーザーであるか、Oracle であるかにかかわらず、オブジェクト型として再定義する必要があります。

手動による型の再コンパイル

型の作成でコンパイル・エラーが発生し、その型に対してなんらかの操作（表の作成、行の挿入など）を実行しようとした場合、「この操作を行う前に、タイプ *<typename>* を再コンパイルします。」というエラーが発生することがあります。手動で型を再コンパイルする場合は、ALTER TYPE *typename* RECOMPILE 文を実行します。型のコンパイルが正常に実行された後、操作を再試行してください。

代入可能な表および列の型依存性

代入可能な表または型 T の列は、T に依存するのみでなく、T のスーパータイプにも依存します。それは、T のサブタイプで追加されたそれぞれの属性について、非表示列が表に追加されることからもわかります。代入可能な表または列に、このサブタイプのデータが何も入っていない場合も、非表示列は追加されます。

したがって、たとえば型 Person_typ の個人表は Person_typ のみでなく、Person_typ のサブタイプである Student_typ および PartTimeStudent_typ にも依存します。

依存型、表または列を持つサブタイプを削除しようとすると、DROP TYPE 文はエラーを戻し、異常終了します。たとえば、PartTimeStudent_typ は、persons 表に依存しているので、削除しようとすると、エラーとなります。

依存表または依存列が存在しても、削除する型のデータが何も入っていない場合は、VALIDATE キーワードを使用して、型を削除できます。VALIDATE キーワードが使用されていると、Oracle は指定された型のインスタンスが実際に格納されているかどうかを調べ、何も見つからなければ、型を削除します。非表示列で、型に固有の属性と関連付けられたものも削除されます。

次に示す最初の DROP TYPE 文は正常に実行されません。PartTimeStudent_typ が依存表（persons）になっているためです。ただし、persons に PartTimeStudent_typ のインスタンスが何も入っていなければ（および他の依存表または依存列のいずれも入っていなければ）、VALIDATE キーワードにより 2 番目の DROP TYPE 文は正常に実行されます。

```
DROP TYPE PartTimeStudent_typ; -- Error due to presence of Persons table
DROP TYPE PartTimeStudent_typ VALIDATE; -- Succeeds if there are no stored
                                        -- instances of PartTimeStudent_typ
```

注意： サブタイプの削除中に、VALIDATE オプションを使用することをお薦めします。

FORCE オプション

DROP TYPE 文には、FORCE オプションもあります。このオプションは、型に依存型または依存表がある場合でも、型を削除します。存在するすべての依存型または依存表に無効のマークが付けられ、型が削除されたときにアクセス不能になるので、FORCE オプションは慎重に使用する必要があります。依存する型が削除されたために無効になった表のデータに、再びアクセスすることはできません。こうした表に対して実行できる操作は、削除のみです。

参照： 型の変更方法の詳細は、第 6 章の「型進化」を参照してください。

ユーザー定義型のシノニム

表、ビューおよびその他の様々なスキーマ・オブジェクトについてシノニムが作成できるように、ユーザー定義型についてもシノニムを定義できます。

型のシノニムは、基礎となるスキーマ・オブジェクトを場所に依存せずに参照する手段となるため、他の種類のスキーマ・オブジェクトのシノニムと同じ利点があります。パブリック・タイプのシノニムを使用するアプリケーションは、データベースのスキーマを変更せずに配置できます。型が定義されているスキーマの名前で型の名前を修飾する必要はありません。

参照： シノニムの詳細は、『Oracle9*i* データベース管理者ガイド』を参照してください。

型のシノニムの作成

型のシノニムは、CREATE SYNONYM 文で作成します。この文の構文は、次のとおりです。

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM [<schema>.]<name> FOR
[<schema>.]<type>[@<dblink>];
```

この場合の <type> は、ユーザー定義型です。

たとえば、これらの文は typ1 型を作成し、次にこの型のシノニムを作成します。

```
CREATE TYPE typ1 AS OBJECT (x number);
CREATE SYNONYM syn1 FOR typ1;
```

シノニムは、コレクション型についても作成できます。次の例は、ネストした表型のシノニムを作成します。

```
CREATE TYPE typ2 AS TABLE OF NUMBER;
```

```
CREATE SYNONYM syn2 FOR typ2;
```

パブリック・シノニムは、PUBLIC キーワードを使用して作成します。

```
CREATE TYPE shape AS OBJECT ( name VARCHAR2(10) );
CREATE PUBLIC SYNONYM pub_shape FOR shape;
```

REPLACE オプションを使用すると、シノニムが指す基礎となる型を別の型に変更することができます。たとえば、次の文は、syn1 が指す型を typ2 型に変更します。

```
CREATE OR REPLACE SYNONYM syn1 FOR typ2;
```

型のシノニムの使用

型のシノニムは、型を参照できるすべての場所で使用できます。たとえば、DDL 文の中で型のシノニムを使用して、表列の型または型属性を指定できます。次の例は、シノニム syn1 を使用して、属性の型を typ3 型に指定します。

```
CREATE TYPE typ1 AS OBJECT (x number);
CREATE SYNONYM syn1 FOR typ1;
CREATE TYPE typ3 AS OBJECT ( a syn1 );
```

次の例は、syn1 をシノニムとするユーザー定義型 typ1 のコンストラクタをコールするために使用する型のシノニム syn1 です。この文は、typ1 のオブジェクト・インスタンスを戻します。

```
SELECT syn1(0) FROM dual;
```

次の例は、syn2 がネストした表型のシノニムです。この例は、実際の型の名前のかわりに CAST 式で使用されるシノニムを示しています。

```
SELECT CAST(MULTISET(SELECT sal FROM SCOTT.EMP) AS syn2) FROM dual;
```

表 4-1 に、型のシノニムが使用できる様々な文を示します。

表 4-1

DML 文	DDL 文
SELECT	AUDIT
INSERT	NOAUDIT
UPDATE	GRANT
DELETE	REVOKE
EXPLAIN PLAN	COMMENT
LOCK TABLE	

シノニムを使用するスキーマ・オブジェクトの記述

型のシノニムを使用して型または表が作成されると、DESCRIBE コマンドにより、これらのシノニムが表す型のかわりにシノニムが表示されます。同様に、型の名前を示すカタログ・ビュー（USER_TYPE_ATTRS など）には、対応付けられた型のシノニムの名前が表示されます。

カタログ・ビュー USER_SYNONYMS を問い合せると、型のシノニムの基礎となる型を確認できます。

型のシノニムへの依存

型の宣言でシノニムを直接的または間接的に参照する型は、そのシノニムに依存します。したがって、次に示す型 typ3 は、シノニム syn1 に依存する型です。

```
CREATE TYPE typ3 AS OBJECT ( a syn1 );
```

DDL 文でシノニムを参照する他の種類のスキーマ・オブジェクトも、これらのシノニムに依存します。型のシノニムに依存するオブジェクトは、シノニムとシノニムの基礎になる型のどちらにも依存します。

シノニムの依存関係によって、シノニムが削除できるかどうか、または名前が変更できるかどうかが決まります。また、依存スキーマ・オブジェクトも、シノニムに対する操作の影響を受けます。次の項では、これらの様々な影響について説明します。

型のシノニムを置換する際の制限事項

シノニムに依存表または有効なユーザー定義型が存在しない場合にのみ、シノニムは置換できます。

シノニムを置換することは、シノニムを削除してから、同じ名前で新しいシノニムを再作成することと同等です。

型のシノニムの削除

シノニムは、DROP SYNONYM 文で削除します。

```
DROP SYNONYM pubsyn1;
```

型のシノニムに依存表または有効なユーザー定義型がある場合、型のシノニムを削除するには、FORCE オプションが必要です。FORCE オプションは、列の実際の型が削除された場合と同様に、直接的または間接的にシノニムに依存する列を、未使用とマークします。（たとえば、シノニムを使用して、列の宣言された型の属性の型を指定すると、列は間接的にそのシノニムに依存します。）

削除されたシノニムに依存するすべてのスキーマ・オブジェクトは無効になります。これらのスキーマ・オブジェクトは、削除されたシノニムと同じ名前のローカル・オブジェクトを作成するか、または同じ名前で新しいパブリック・シノニムを作成することで、再検証できます。

型のシノニムの基礎となるベース型の削除は、シノニムの削除と同じ結果を依存オブジェクトにもたらします。

型のシノニムの名前変更

型のシノニムの名前は、RENAME 文で変更します。次の例は、シノニム employee の名前を変更します。

```
RENAME employee TO emp;
```

シノニムの名前を変更することは、シノニムを削除してから、新しい名前でシノニムを再作成することと同等です。

型のシノニムに依存表または有効なユーザー定義型が存在する場合は、型のシノニムの名前は変更できません。

型のパブリック・シノニムおよびローカル・スキーマ・オブジェクト

パブリック・シノニムが、新しいスキーマ・オブジェクトを作成するローカル・スキーマに依存表または有効なユーザー定義型を持っている場合、そのパブリック・シノニムと同じ名前のローカル・スキーマ・オブジェクトは作成できません。また、同じスキーマ内のプライベート・スキーマと同じ名前のローカル・スキーマ・オブジェクトも作成できません。

たとえば、次の例では、表 tab1 はパブリック・シノニム pubsyn1 の依存表です。この表には、型の定義でこのシノニムを使用する列があるためです。したがって、依存表と同じスキーマにパブリック・シノニム pubsyn1 と同じ名前の表を作成しようとすると、正常に実行されません。

```
CREATE TABLE tab1 ( c1 pubsyn1 );    -- Uses public synonym pubsyn1
CREATE TABLE pubsyn1 ( c1 NUMBER );  -- Not allowed
```

Oracle のツール製品

この項では、Oracle オブジェクトのサポート機能を持ついくつかの Oracle のツール製品について説明します。

JDeveloper

JDeveloper は、複数層の Java アプリケーションの作成を目的としたフル機能の統合開発環境です。このツールにより、Java クライアント・アプリケーション、動的 HTML アプリケーション、Web およびアプリケーション・サーバー・コンポーネントおよび業界標準モデルに基づいたデータベース・ストアド・プロシージャの開発、デバッグおよび配置ができます。

JDeveloper の強力な機能が提供されるのは、次の分野です。

- Oracle Business Components for Java
- Web アプリケーションの開発
- クライアント向け Java アプリケーションの開発
- データベースにおける Java
- JavaBeans によるコンポーネント・ベースの開発
- シンプリファイド・データベース・アクセス
- ビジュアル統合開発環境
- Java 言語サポート

JDeveloper は Windows NT で稼働し、Oracle のアプリケーション・サーバーおよびデータベースと緊密に統合された標準の GUI ベースの Java 開発環境となります。

Business Components for Java（BC4J）

標準 EJB および CORBA 開発アーキテクチャである Oracle Business Components for Java がサポートされるので、エンタープライズ向けの Java ビジネス・アプリケーションの開発、配布、カスタマイズが簡素化されます。Oracle Business Components for Java は、アプリケーション・コンポーネントのフレームワークで、次に示す作業に必要な共通ファシリティすべてを管理する再利用可能なソフトウェア・ビルディング・ブロックを提供します。

- リレーショナル・データベースと統合するコンポーネントのビジネス・ロジックの作成およびテスト
- データの複数の SQL ベース・ビューを介したビジネス・ロジックの再利用
- サーブレット、JavaServer Pages（JSP）および Thin-Java Swing クライアントから行うビューへのアクセスおよび更新

- 配布したアプリケーションの修正を必要としないレイヤー単位のアプリケーション機能のカスタマイズ

JPublisher

JPublisher は、完全に Java で書かれたユーティリティで、次のユーザー定義データベース・エンティティをユーザーの Java プログラムで表す Java クラスを生成します。

- データベース・オブジェクト・タイプ
- データベース参照（REF）型
- データベース・コレクション型（VARRAY またはネストした表）
- PL/SQL パッケージ

JPublisher により、強力な型指定で、データベース・オブジェクト型、参照型およびコレクション型（VARRAY またはネストした表）を Java クラスに対してマップする操作を指定し、カスタマイズできます。

参照：『Oracle9*i* JPublisher ユーザーズ・ガイド』を参照してください。

ユーティリティ

オブジェクト型のインポート / エクスポート

エクスポート・ユーティリティおよびインポート・ユーティリティは、Oracle データベースへ、または Oracle データベースから、データを移動します。また、データのバックアップまたはアーカイブ、および Oracle RDBMS の異なるリリースへの移行に役立ちます。

エクスポートおよびインポートは、オブジェクト型をサポートします。エクスポートは、オブジェクト型定義および関連するすべてのデータをダンプ・ファイルに書き込みます。インポートは、ダンプ・ファイルからこれらの項目を再作成します。

型

導出型の定義文はエクスポートされます。インポート時、スーパータイプ定義のインポートが完了する前に、サブタイプを作成できます。この場合、サブタイプを作成するとコンパイル・エラーが発生しますが、無視してください。型のスーパータイプが作成された後、型の再検証が行われます。

オブジェクト・ビューの階層

1 つのビュー階層に属するすべてのビューのビュー定義がエクスポートされます。

SQL*Loader

SQL*Loader ユーティリティは、外部ファイルのデータを、Oracle データベース内の表に移動します。これらのファイルには、INTEGER、CHAR、DATE など基本的なスカラー・データ型で構成されるデータや、行オブジェクトや列オブジェクト（オブジェクト、コレクションまたは REF 属性）、コレクション、LOB など複雑なユーザー定義データ型などが含まれる可能性があります。現在、SQL*Loader はシングルレベル・コレクションのみをサポートしています。マルチレベル・コレクション（他のコレクションが要素になっているか、他のコレクションが入っているコレクション）のロードに、SQL*Loader は使用できません。

SQL*Loader は、SQL*Loader DDL 文が入っている制御ファイルを使用して、データ・ファイルのフォーマット、内容および位置を記述します。

SQL*Loader がデータをロードする方法は、2 通りあります。

- **従来型パス・ロード**: SQL INSERT 文およびバインド配列バッファを使用して、データをデータベース表にロードします。
- **ダイレクト・パス・ロード**: ダイレクト・パス・ロード API を使用して、SQL*Loader クライアントのかわりに、データ・ブロックを直接データベースに書き込みます。

  ダイレクト・パス・ロードは SQL インタフェースを使用しないので、関連する SQL 文の処理時にオーバーヘッドが発生しません。したがって、ダイレクト・パス・ロードからは従来型パス・ロードよりも優れたパフォーマンスが得られます。

どちらの方法も、サポート対象となっているオブジェクトおよびコレクション・データ型のデータのロードに使用できます。

> **参照：** SQL*Loader の使用方法は、『Oracle9*i* データベース・ユーティリティ』を参照してください。

5

オブジェクト・モデルのリレーショナル・データへの適用

この章では、基礎となるリレーショナル・データの構造を変更することなく、オブジェクト指向アプリケーションを作成する方法を示します。

内容は次のとおりです。

オブジェクト・ビュー使用の利点

ビューが仮想表であるように、オブジェクト・ビューは仮想オブジェクト表です。オブジェクト・ビューの各行は、1 つのオブジェクトであるため、ドット表記法を使用してこのオブジェクトのメソッドをコールし属性にアクセスすることができます。また、これを指す REF を作成することもできます。

オブジェクト・ビューは、オブジェクト指向アプリケーションのプロトタイピングやオブジェクト指向アプリケーションへ切り替える場合に役立ちます。ビューの中のデータはリレーショナル表から取り出すことができ、オブジェクト表として定義されているかのようにアクセスすることができるためです。したがって、既存の表を別の物理構造に変換しなくても、オブジェクト指向アプリケーションを実行できます。

オブジェクト・ビューは、リレーショナル・ビューのように使用し、目的のデータのみ画面上に表示できます。たとえば、給料など機密データを除いて、従業員表の特定のデータを表示するオブジェクト・ビューが作成できます。

オブジェクト・ビューの使用によって、パフォーマンスが向上します。オブジェクト・ビューの 1 行を構成するリレーショナル・データは、1 単位としてネットワークを横断するため、ラウンドトリップが大幅に削減されます。

リレーショナル・データを、クライアント側のオブジェクト・キャッシュにフェッチすることができ、C の構造体、C++ または Java クラスにマップすることもできます。そのため、3GL アプリケーションで固有のクラスのように操作することができます。また、複雑なオブジェクト検索のようなオブジェクト機能をリレーショナル・データとともに使用することもできます。

- リレーショナル・データからオブジェクトを合成することによって、新しい方法でデータを問い合せることができます。複数の表との複雑な結合を記述するかわりに、オブジェクトの参照解除を利用することで複数の表からデータを表示することができます。
- このビューのオブジェクトは、クライアント上ではなくサーバー内で処理されるため、SQL 文の数およびネットワーク通信量を大幅に削減できます。
- オブジェクト・ビューのオブジェクト・データは、クライアント側のオブジェクト・キャッシュに確保し使用することができます。オブジェクト・キャッシュ上に確保された、これらの合成されたオブジェクトを、専用のオブジェクト検索メカニズムにより取り出すことで、ネットワーク通信量を削減できます。
- ビュー内部にオブジェクト・モデルを作成しながら、モデル開発が継続できるという、優れた柔軟性が得られます。オブジェクト型を変更する必要がある場合、無効なビューを新しい定義に簡単に置換できます。
- ビューの中のオブジェクトを使用することで、基礎となる記憶域メカニズムの特性が制限されることはありません。同様に、現行のテクノロジによって制限されることもありません。たとえば、パラレル化およびパーティション化されたリレーショナル表からオブジェクトを合成することもできます。
- 基礎となる同じデータから異なる複合データ・モデルを作成できます。

参照：

- SQL 構文および使用方法の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。
- PL/SQL の機能の詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。
- Java の詳細は、『Oracle9*i* Java Stored Procedures Developer's Guide』を参照してください。
- これらの機能の詳細は、『Oracle Call Interface プログラマーズ・ガイド』を参照してください。

オブジェクト・ビューの定義

オブジェクト・ビューを定義する手順は次のとおりです。

1. 既存のリレーショナル表の列にそれぞれ対応するデータ型の属性を持つオブジェクト型を定義します。
2. リレーショナル表からデータを抽出する問合せを作成します。オブジェクト型の属性と同じ順序で、リレーショナル表の列を指定します。
3. 基礎となるデータの属性に基づいて、一意の値を指定します。これは、ビュー内のオブジェクトを指すポインタ（REF）を作成するため OID として動作します。既存の主キーを使用することもできます。

オブジェクト・ビューを更新するには、さらに別の手順が必要です。オブジェクト型の属性が既存の表の列に正確に対応していない場合、次のことを行います。

4. アプリケーション・プログラムがオブジェクト・ビューのデータを更新するときに、Oracle に実行させる INSTEAD OF トリガー・プロシージャ（5-12 ページの「オブジェクト・ビューの更新」を参照）を作成します。

これで、オブジェクト・ビューをオブジェクト表と同様に使用できます。

たとえば、次の SQL 文は、ビューの各行が employee_t 型のオブジェクトであるオブジェクト・ビューを定義します。

```
CREATE TABLE emp_table  (
    empnum   NUMBER (5),
    ename    VARCHAR2 (20),
    salary   NUMBER (9, 2),
    job      VARCHAR2 (20) );

CREATE TYPE employee_t  (
    empno    NUMBER (5),
    ename    VARCHAR2 (20),
    salary   NUMBER (9, 2),
```

```
    job      VARCHAR2 (20) );

CREATE VIEW emp_view1 OF employee_t
    WITH OBJECT IDENTIFIER (empno) AS
        SELECT   e.empnum, e.ename, e.salary, e.job
            FROM     emp_table e
            WHERE    job = 'Developer';
```

リレーショナル表の empnum 列のデータにアクセスするには、オブジェクト型の empno 属性にアクセスすることになります。

アプリケーションにおけるオブジェクト・ビューの使用

オブジェクト・ビューの行にあるデータは、2 つ以上の表から生成されている可能性がありますが、オブジェクトは 1 つの操作のみでネットワークを横断します。クライアント側のオブジェクト・キャッシュの中では、インスタンスは C または C++ の構造体として、または PL/SQL オブジェクト変数として現れます。このインスタンスを、他のすべてのシステム固有の構造体と同様に操作することもできます。

SQL 文の中で、オブジェクト・ビューはオブジェクト表と同じ方法で参照できます。たとえば、オブジェクト・ビューが現れる可能性があるのは、SELECT 構文のリスト、UPDATE-SET 句または WHERE 句の中です。

また、オブジェクト・ビューに、オブジェクト・ビューも定義できます。

オブジェクト表からオブジェクトに対して使用するものと同じ OCI コールを使用して、クライアント側のオブジェクト・ビューのデータにアクセスできます。たとえば、REF を確保するために OCIObjectPin() を、また、オブジェクトをサーバーにフラッシュするために OCIObjectFlush() を使用することができます。オブジェクト・ビューのオブジェクトを更新またはサーバーにフラッシュすると、オブジェクト・ビューが更新されます。

> **参照：** OCI コールの詳細は、『Oracle Call Interface プログラマーズ・ガイド』を参照してください。

オブジェクト・ビューにおけるオブジェクトのネスト

オブジェクト型は、自身にネストした他のオブジェクト型を属性として持つことができます。

オブジェクト・ビューの基になっているオブジェクト型が、それ自身がオブジェクト型である属性を持っている場合は、オブジェクト・ビューを作成する処理の一部として、この属性用の列オブジェクトを用意する必要があります。属性型の列オブジェクトがリレーショナル表にすでに存在する場合は、この列オブジェクトを選択するのみで済みます。存在しない場合は、ビューの主オブジェクト・インスタンスを合成する場合と同様に、基礎となるリレーショナル・データに基づきオブジェクト・インスタンスを合成する必要があります。オブ

ジェクト・インスタンスを作成するための、それぞれのオブジェクト型のコンストラクタ・メソッドをコールして、これらのオブジェクトを合成または作成します。次に、コンストラクタで指定したリレーショナル列からのデータを使用して、これらのオブジェクトの属性を移入します。

たとえば、次の部門表 dept について考えてみます。

```
CREATE TABLE dept
(
    deptno       NUMBER PRIMARY KEY,
    deptname     VARCHAR2(20),
    deptstreet   VARCHAR2(20),
    deptcity     VARCHAR2(10),
    deptstate    CHAR(2),
    deptzip      VARCHAR2(10)
 );
```

住所が部門オブジェクト内部のオブジェクトになっているオブジェクト・ビューを作成します。これにより、住所オブジェクト用の再利用可能なメソッドを定義し、様々なアドレスに使用できるようになります。

1. アドレス・オブジェクト用の型を作成します。

```
CREATE TYPE address_t AS OBJECT
(
   street   VARCHAR2(20),
    city    VARCHAR2(10),
    state   CHAR(2),
    zip     VARCHAR2(10)
);
```

2. 部門オブジェクト用の型を作成します。

```
CREATE TYPE dept_t AS OBJECT
(
   deptno     NUMBER,
   deptname   VARCHAR2(20),
   address    address_t
);
```

3. 部門番号、名前および住所を含むビューを作成します。リレーショナル表の列から、address オブジェクトが作成されます。

```
CREATE VIEW dept_view OF dept_t WITH OBJECT IDENTIFIER (deptno) AS
    SELECT d.deptno, d.deptname,
           address_t(d.deptstreet,d.deptcity,d.deptstate,d.deptzip) AS
      deptaddr
      FROM dept d;
```

オブジェクト・ビューにおける NULL オブジェクトの識別

オブジェクトのコンストラクタから、NULL が返されることはありません。したがって、前述のビューにおいて、リレーショナル表の中の city、street などの列が NULL であっても、アドレス・オブジェクトが NULL になることはありえません。リレーショナル表には、部門の住所が NULL かどうかを指定する列はありません。NULL の deptstreet 列が、すべての住所が NULL であることを示すように規則を定義した場合、DECODE ファンクションまたはその他のファンクションを使用してロジックを取り込み、NULL または構成されたオブジェクトのいずれかを戻すことができます。

```
CREATE VIEW dept_view AS
  SELECT d.deptno, d.deptname,
        DECODE(d.deptstreet, NULL, NULL,
            address_t(d.deptstreet, d.deptcity, d.deptstate, d.deptzip)) AS deptaddr
  FROM dept d;
```

部門の住所は、リレーショナル表の中の列と直接対応していないため、このような方法を使用すると、ビューを介して部門の住所を直接更新できなくなります。この方法ではなく、ビュー上に INSTEAD OF トリガーを定義して、この列の更新を操作します。

オブジェクト・ビューにおけるネストした表および VARRAY の使用

ネストした表および VARRAY のコレクションはどちらも、ビューの列になる場合があります。これらのコレクションは、基礎となるコレクション列から選択するか、または副問合せを使用して合成できます。CAST-MULTISET 演算子によって、これらのコレクションを合成します。

オブジェクト・ビューのシングルレベル・コレクション

ここでも、前述の例を使用し、次の構造を持つ emp リレーショナル表に存在する各従業員について考えてみます。

```
CREATE TABLE emp
(
   empno    NUMBER PRIMARY KEY,
   empname  VARCHAR2(20),
   salary   NUMBER,
   deptno   NUMBER REFERENCES dept(deptno)
);
```

このリレーショナル表を使用すると、部門番号、名前、住所および部門に所属する従業員のコレクションを持つ dept_view を構成できます。

1. 従業員型を定義し、その従業員型用のネストした表を定義します。

```
CREATE TYPE employee_t AS OBJECT
(
  eno NUMBER,
  ename VARCHAR2(20),
  salary  NUMBER
);

CREATE TYPE employee_list_t AS TABLE OF employee_t;
```

2. 部門番号、名前、住所および従業員のネストした表を持つ部門型を定義します。

```
CREATE TYPE dept_t AS OBJECT
(   deptno     NUMBER,
    deptname   VARCHAR2(20),
    address    address_t,
    emp_list   employee_list_t
);
```

3. オブジェクト・ビュー dept_view を定義します。

```
CREATE VIEW dept_view OF dept_t WITH OBJECT IDENTIFIER (deptno) AS
    SELECT d.deptno, d.deptname,
        address_t(d.deptstreet,d.deptcity,d.deptstate,d.deptzip) AS deptaddr,
             CAST( MULTISET (
                          SELECT e.empno, e.empname, e.salary
                          FROM emp e
                          WHERE e.deptno = d.deptno)
                      AS employee_list_t)
                  AS emp_list
   FROM   dept d;
```

CAST-MULTISET ブロック内の SELECT 副問合せにより、現在の部門に所属する従業員のリストが選択されます。MULTISET キーワードは、これが単一の値とは対照的なリストであることを示します。CAST 演算子は、結果セットを適切な型にキャスト（型変換）します。この場合は、ネストした表型 employee_list_t にキャストします。

このビューへの問合せでは、それぞれの部門行に、部門番号、名前、住所オブジェクトおよびその部門に所属する従業員のコレクションが入っている部門リストが取得できます。

オブジェクト・ビューのマルチレベル・コレクション

マルチレベル・コレクションもシングルレベル・コレクションも、同じ方法で作成し、オブジェクト・ビューで使用します。異なるのは、マルチレベル・コレクションで、別のレベルのコレクションを作成する必要がある点のみです。

次の例では、マルチレベル・コレクションが含まれているオブジェクト・ビューを作成します。このビューは、フラットなリレーショナル表（コレクションが何も入っていない表）に基づいています。オブジェクト・ビューを作成する準備作業として、この例で使用するオブ

ジェクト型とコレクション型を作成します。それぞれのリレーショナル表と対応するように、オブジェクト型（たとえば、emp_t）を定義します。このオブジェクト型の属性の型は、それぞれの表列の型と対応します。従業員型はプロジェクトのネストした表（属性）を持ち、部門型は従業員のネストした表（属性）を持ちます。後者のネストした表は、マルチレベル・コレクションです。CREATE VIEW 文の中で CAST-MULTISET 演算子を使用して、コレクションを作成します。

基礎となるリレーショナル表は次のとおりです。

```
CREATE TABLE depts
  ( deptno     NUMBER
  , deptname   VARCHAR2(20));

CREATE TABLE emp
  ( ename VARCHAR2(20),
  , salary     NUMBER
  , deptname   VARCHAR2(20));

CREATE TABLE projects
  ( projname   VARCHAR2(20)
  , mgr        VARCHAR2(20));
```

ビューが使用するオブジェクト型およびコレクション型は、次のとおりです。

```
CREATE TYPE project_t AS OBJECT
  ( projname   VARCHAR2(20)
  , mgr        VARCHAR2(20));

CREATE TYPE nt_project_t AS TABLE OF project_t;

CREATE TYPE emp_t AS OBJECT
  ( ename      VARCHAR2(20)
  , salary     NUMBER
  , deptname   VARCHAR2(20)
  , projects   nt_project_t);

CREATE TYPE nt_emp_t AS TABLE OF emp_t;

CREATE TYPE dept_t AS OBJECT
  ( deptno     NUMBER
  , deptname   VARCHAR2(20)
  , emps       nt_emp_t);
```

次の文により、オブジェクト・ビューが作成されます。

```
CREATE VIEW v_depts OF dept_t AS
  SELECT d.deptno, d.deptname,
    CAST(MULTISET(SELECT e.ename, e.salary, e.deptname,
```

```
      CAST(MULTISET(SELECT p.projname, p.mgr
        FROM projects
        WHERE p.mgr = e.ename)
      AS nt_project_t)
    FROM emp
    WHERE e.deptname = d.deptname)
  AS nt_emp_t)
FROM depts d;
```

オブジェクト・ビューに対する OID の指定

オブジェクト・ビューの中で、行オブジェクトを指すポインタ（REF）を作成できます。ビューのデータは永続的に格納されるわけではないため、OID として使用される値を指定する必要があります。OID によって、オブジェクト・ビューにおけるオブジェクトを、オブジェクト・キャッシュ上で参照し、確保することができます。

このビューがオブジェクト表またはオブジェクト・ビューに基づいている場合には、すでに各行に対応付けられた OID があり、これを再利用できます。WITH OBJECT IDENTIFIER 句を省略するか、または WITH OBJECT IDENTIFIER DEFAULT 句を指定します。

ただし、行オブジェクトをリレーショナル・データから合成する場合は、他の値セットを選択する必要があります。

Oracle では、主キーに基づいた OID を指定できます。行オブジェクトを識別する一意のキーの集合は、そのオブジェクトの識別子になります。値が重複すると、オブジェクト参照時に問題が発生することもあるため、これらの値はビューから選択された行内で一意である必要があります。

WITH OBJECT IDENTIFIER 句を使用して作成されたオブジェクト・ビューは、指定した列の値から導出された OID を持ちます。WITH OBJECT IDENTIFIER DEFAULT 句を指定すると、基礎となる表またはビュー定義によって、OID はシステム生成または主キー・ベースです。

この部門表の例を使用して、部門番号を OID として使用する dept_view オブジェクト・ビューを作成します。

行のオブジェクト型を定義します。この例では、dept_t 部門型を定義します。

```
CREATE TYPE dept_t AS OBJECT
(
  dno        NUMBER,
  dname      VARCHAR2(20),
  deptaddr   address_t,
  emplist    employee_list_t
);
```

基礎となるリレーショナル表に deptno が主キーとして入っているため、各部門行は一意の部門番号を持ちます。このビューでは、deptno 列がオブジェクト型の dno 属性になります。ビュー・オブジェクト内で dno が一意であることがわかれば、これを OID として指定できます。

```
CREATE VIEW dept_view OF dept_t WITH OBJECT IDENTIFIER(dno)
   AS SELECT d.deptno, d.deptname,
                address_t(d.deptstreet,d.deptcity,d.deptstate,d.deptzip),
                CAST( MULTISET (
                          SELECT e.empno, e.empname, e.salary
                          FROM emp e
                          WHERE e.deptno = d.deptno)
                     AS employee_list_t)
   FROM   dept d;
```

参照： 6-6 ページの「オブジェクト識別子（OID）」を参照してください。

オブジェクトを表示するための参照の作成

前述の例にある dept_view ビューから選択された各オブジェクトは、部門番号の値から導出された一意の OID を持っています。リレーショナルの場合、従業員表 emp の外部キー deptno は、部門表 dept の主キー値 deptno と一致します。ここでは、dept_view で OID を作成するために、主キー値を使用しました。これにより、dept_view の主キー値の参照を作成するときに、emp_view の外部キーの値が使用できます。

ここでは、MAKE_REF 演算子により、主キーのオブジェクト参照を合成して、この処理を実現します。この演算子は、参照がポイントするビューまたは表の名前および外部キー値のリストをとり、参照されたビューの中の特定のオブジェクトと一致するような参照の OID を作成します。

従業員の番号、名前、給料および所属部門の参照を持つ emp_view ビューを作成するには、まず従業員型 emp_t を作成し、次にその型に基づいたビューを作成する必要があります。

```
CREATE TYPE emp_t AS OBJECT
(
  eno      NUMBER,
  ename    VARCHAR2(20),
  salary   NUMBER,
  deptref  REF dept_t
);

CREATE VIEW emp_view OF emp_t WITH OBJECT IDENTIFIER(eno)
   AS SELECT e.empno, e.empname, e.salary,
```

```
                    MAKE_REF(dept_view, e.deptno)
        FROM emp e;
```

ビューの中の deptref 列に、部門の参照が格納されます。次の簡単な問合せを書き込み、サンフランシスコ市にある部門のすべての従業員を確認してみます。

```
SELECT e.eno, e.salary, e.deptref.dno
FROM emp_view e
WHERE e.deptref.deptaddr.city = 'San Francisco';
```

REF 修飾子を使用しても、dept_view オブジェクトの参照が取得できる点に注意してください。

```
CREATE VIEW emp_view OF emp_t WITH OBJECT IDENTIFIER(eno)
   AS SELECT e.empno, e.empname, e.salary, REF(d)
         FROM emp e, dept_view d
          WHERE e.deptno = d.dno;
```

この場合、dept_view および emp 表を deptno キーで結合します。MAKE_REF 演算子を使用すると、循環参照を作成できるため、REF 修飾子ではなく、この演算子を使用しています。たとえば、所属部門への参照を持つ従業員のビューを作成することができます。また、部門のビューは、その部門に所属している従業員への参照のリストを持つことができます。

オブジェクト・ビューが主キーに基づく OID を持つ場合、このようなビューへの参照は主キーに基づくということに注意してください。一方、システム生成の OID を持つビューへの参照は、システム生成のオブジェクト参照になります。この違いが関係するのは、OCI オブジェクト・キャッシュでオブジェクト・インスタンスを作成し、新しく作成したオブジェクトの参照を取得する必要がある場合のみです。これについては後の項で説明します。

合成オブジェクトの場合と同様、永続的に格納される参照をビュー列として選択し、これらを問合せで透過的に使用できます。ただし、ビュー・オブジェクトに対するオブジェクト参照を永続的に格納することはできません。

オブジェクト・ビューを利用した逆リレーションシップのモデル化

オブジェクトを持つビューは、逆リレーションシップをモデル化するために使用できます。

1 対 1 リレーションシップ

1 対 1 リレーションシップは、オブジェクトの逆参照を使用してモデル化できます。たとえば、各従業員が専用のコンピュータを持っているとしましょう。リレーショナル・モデルであれば、コンピュータ表から従業員表へ、または逆の方向で外部キーを使用して、1 対 1 リレーションシップを獲得します。ビューを使用すると、従業員からコンピュータ・オブジェクトへのオブジェクト参照や、コンピュータ・オブジェクトから従業員の参照が行えるように、オブジェクトをモデル化できます。

1 対多および多対 1 リレーションシップ

1 対多リレーションシップ（または多対 1 リレーションシップ）は、オブジェクト参照を使用するか、オブジェクトを埋め込むことでモデル化できます。1 対多リレーションシップは、オブジェクトのコレクションまたはオブジェクト参照のコレクションを持つことでモデル化できます。多対 1 リレーションシップは、オブジェクト参照を使用してモデル化できます。

部門対従業員のケースを検討してみましょう。基礎となるリレーショナル・モデルでは、外部キーが従業員表の中にあります。ビューにおいてコレクションを使用すると、部門と従業員のリレーションシップをモデル化できます。部門ビューに従業員のコレクションを持たせ、従業員ビューに部門の参照を持たせることができます（または部門値がインライン化できます）。これにより、前方リレーションシップ（従業員から部門へ）と逆リレーションシップ（部門から従業員リストへ）の両方が得られます。また、部門ビューに、従業員オブジェクトを埋め込むかわりに、従業員オブジェクトの参照のコレクションを持たせることも可能です。

オブジェクト・ビューの更新

オブジェクト表に使用する SQL DML と同じものを使用して、オブジェクト・ビューのデータを更新、挿入および削除できます。あいまいな表現がなければ、オブジェクト・ビューの実表が更新されます。

ビューの問合せに結合、集合演算子、集計ファンクション、GROUP BY または DISTINCT 句が含まれている場合、ビューの直接更新はできません。ビューの問合せで、ビューの列が疑似列または式に基づいている場合も、ビューの個々の列を直接更新することはできません。

ビューが直接更新できなくても、INSTEAD OF トリガーを使用して、間接的に更新できます。この操作を行うには、ビューに対して実行するそれぞれの DML 文に、INSTEAD OF トリガーを定義します。INSTEAD OF トリガーには、目的の変更をビューで行うために、ビューの基礎となる表に対して実行する必要のある操作をプログラムします。この後、INSTEAD OF トリガーが定義された DML 文を発行すると、関連するトリガーが透過的に実行されます。

> **参照：** INSTEAD OF トリガーの例は、5-13 ページの「変更および妥当性チェックを制御する INSTEAD OF トリガーの使用」を参照してください。

ここでは、次の点に注意してください。オブジェクト・ビュー階層では、UPDATE 文および DELETE 文は、SELECT 文と同様に多相性を伴って動作します。つまり、ビューに対して実行された UPDATE または DELETE 文で抽出される行の集合には、指定されたビューのすべてのサブビューにある修飾行も明示的に含まれます。

たとえば、Person_v から全員を削除する次の文により、Student_v から学生全員が削除され、Employee_v ビューから従業員全員が削除されます。

```
DELETE FROM Person_v;
```

サブビューを除外し、影響する行の対象を、実際に指定されたビューに存在する人たちに限定するには、ONLY キーワードを使用します。たとえば、次の文は個人のみ更新し、従業員または学生は更新しません。

```
UPDATE ONLY(Person_v) SET address = ...
```

ビューにおけるネストした表列の更新

ネストした表は、新しい要素の挿入や、既存の要素の更新または削除で変更することができます。ビューの場合と同様に、仮想または合成のネストした表列は、通常は更新できません。この問題に対処するために、Oracle では、この列の上に INSTEAD OF トリガーを作成できます。

（ビューの）ネストした表列に対して定義された INSTEAD OF トリガーは、列を変更するときに起動します。親である行の更新によってコレクション全体が置換される場合は、ネストした表列の INSTEAD OF トリガーは起動されないため注意してください。

変更および妥当性チェックを制御する INSTEAD OF トリガーの使用

INSTEAD OF トリガーを使用することで、他の方法では更新できない複雑なビューを更新できます。また制約の施行、権限のチェック、DML の妥当性チェックにも、このトリガーを使用できます。これらのトリガーを使用すると、挿入、更新および削除が行われた結果、オブジェクト・ビューにより作成されるオブジェクトの変更を制御できます。

たとえば、ある部門の従業員の数を 10 以下とするという条件を施行するケースを考えてみましょう。この条件を強制するために、従業員ビュー用の INSTEAD OF トリガーを作成します。ビューは更新できるため、DML の実行にトリガーは必要ありません。ただし、制約を施行するにはトリガーが必要です。

次のコードを使用して、トリガーを実装します。

```
CREATE TRIGGER emp_instr INSTEAD OF INSERT on emp_view
FOR EACH ROW
DECLARE
  dept_var dept_t;
  emp_count integer;
BEGIN
  -- Enforce the constraint..!
  -- First get the department number from the reference
  UTL_REF.SELECT_OBJECT(:NEW.deptref,dept_var);

  SELECT COUNT(*) INTO emp_count
  FROM emp
  WHERE deptno = dept_var.dno;

  IF emp_count < 9 THEN
```

```
      -- let us do the insert
     INSERT INTO emp VALUES (:NEW.eno,:NEW.ename,:NEW.salary,dept_var.dno);
  END IF;
END;
```

オブジェクト・モデルのリモート表への適用

リモート表は、オブジェクト表のように直接アクセスできませんが、オブジェクト・ビューでは、あたかもオブジェクト表であるかのように、リモート表にアクセスできます。

ワシントン D.C. およびシカゴに支店を持つ会社について考えてみます。各支店に、従業員表があります。ワシントンの本社には、すべての部門のリストを持つ部門表があります。組織全体のビューを作成するには、個々のリモート表の上にビューを作成し、次に組織全体のビューを作成します。

まず、各従業員表のオブジェクト・ビューを作成します。

```
CREATE VIEW emp_washington_view (eno,ename,salary)
   AS SELECT e.empno, e.empname, e.salary
         FROM   emp@washington_link e;

CREATE VIEW emp_chicago_view
   AS SELECT e.eno, e.name, e.salary
         FROM   emp_tab@chicago_link e;
```

ここで、グローバルなビューを作成できます。

```
CREATE VIEW orgnzn_view  OF dept_t WITH OBJECT IDENTIFIER  (dno)
    AS SELECT d.deptno, d.deptname,
              address_t(d.deptstreet,d.deptcity,d.deptstate,d.deptzip),
              CAST( MULTISET (
                      SELECT e.eno, e.ename, e.salary
                      FROM emp_washington_view e)
                   AS employee_list_t)
       FROM   dept d
       WHERE d.deptcity = 'Washington'
   UNION ALL
       SELECT d.deptno, d.deptname,
              address_t(d.deptstreet,d.deptcity,d.deptstate,d.deptzip),
              CAST( MULTISET (
                       SELECT e.eno, e.name, e.salary
                       FROM emp_chicago_view e)
                    AS employee_list_t)
       FROM   dept d
       WHERE d.deptcity = 'Chicago';
```

このビューは、各部門のすべての従業員のリストを持ちます。複数の部門に所属する従業員がいないため、UNION ALL を使用します。複数の部門に所属する従業員がいる場合は、行の UNION が使用できます。ただし、UNION 演算子を使用する場合は、2 つのコレクションの比較が適切に実行されるように、CAST-MULTISET 式内に ORDER BY 演算子を挿入する必要があります。

オブジェクト・ビューにおける複雑なリレーションシップの定義

オブジェクト・ビューには、MAKE_REF 演算子を使用して、循環参照を定義できます。view_A は view_B を参照し、反対に view_A を同様に参照します。これにより、リレーショナル・データで構成されるグラフのように複雑な構造を、オブジェクト・ビューが合成できるようになります。

たとえば、部門および従業員の例の場合、部門オブジェクトに従業員のリストが含まれています。領域を節約するために、部門オブジェクト内のすべての従業員を具体化せずに、従業員オブジェクトの参照を、部門オブジェクト内部に入れてみます。従業員オブジェクトへの参照を構成（確保）し、後でドット表記法を使用してこの参照に従い、従業員情報を抽出できます。

従業員オブジェクトはすでに従業員の所属部門への参照を持っているため、このモデルでのオブジェクト・ビューには、部門ビューと従業員ビューの間の循環参照が含まれます。

オブジェクト・ビュー間の循環参照は、次の 2 通りの方法で作成できます。

方法 1: 2 つ目のビューを作成した後、最初のビューを再作成する

1. ビュー B への参照を持たないビュー A を作成します。
2. ビュー A への参照を含むビュー B を作成します。
3. ビュー A をビュー B の参照を含む新しい定義で置換します。

方法 2: FORCE キーワードを使用して最初のビューを作成する

1. FORCE キーワードを使用して、ビュー B への参照を持つビュー A を作成します。
2. ビュー A の参照を持つビュー B を作成します。ビュー A を使用すると、妥当性チェックが行われ、再コンパイルされます。

方法 2 の方が、少ない手順でできますが、ビューの作成時に、FORCE キーワードによりエラーが隠される可能性があります。ビュー作成中にエラーが発生したかどうかを確認するには、USER_ERRORS カタログ・ビューを問い合せる必要があります。ビュー作成文の中にエラーがないことが保証されている場合にのみ、この方法を使用してください。

また、使用時にエラーによってビューが再コンパイルできない場合は、ALTER VIEW COMPILE コマンドを使用し、ビューを手動で再コンパイルする必要があります。

では、両方の方法の実装について考えてみます。

循環参照を示す表および型

まず、リレーショナル表および対応付けれられたオブジェクト型を設定します。表にオブジェクトが入っていても、この表はオブジェクト表ではありません。データ・オブジェクトにアクセスするために、後でオブジェクト・ビューを作成します。

emp 表には、従業員情報が入っています。

```
CREATE TABLE emp
(
   empno    NUMBER PRIMARY KEY,
   empname  VARCHAR2(20),
   salary   NUMBER,
   deptno   NUMBER
);
```

emp_t 型には、部門の参照が入っています。emp_t 型の作成が正常に実行されるように、ダミーの部門型を用意する必要があります。

```
CREATE TYPE dept_t;
/
```

従業員型は、部門の参照を含んでいます。

```
CREATE TYPE emp_t AS OBJECT
(
  eno NUMBER,
  ename VARCHAR2(20),
  salary  NUMBER,
  deptref REF dept_t
);
/
```

従業員の参照のリストを、ネストした表として表します。

```
CREATE TYPE employee_list_ref_t AS TABLE OF REF emp_t;
/
```

部門表は、典型的なリレーショナル表です。

```
CREATE TABLE dept
(
    deptno        NUMBER PRIMARY KEY,
    deptname      VARCHAR2(20),
    deptstreet    VARCHAR2(20),
    deptcity      VARCHAR2(10),
    deptstate     CHAR(2),
    deptzip       VARCHAR2(10)
 );
```

オブジェクト・ビューを作成するには、リレーショナル表から列にマップするオブジェクト型が必要です。

```
CREATE TYPE address_t AS OBJECT
(
   street        VARCHAR2(20),
    city         VARCHAR2(10),
    state        CHAR(2),
    zip          VARCHAR2(10)
);
/
```

ここで、前に作成した不完全な型を再定義します。

```
CREATE OR REPLACE TYPE dept_t AS OBJECT
(
  dno           NUMBER,
  dname         VARCHAR2(20),
  deptaddr      address_t,
  empreflist    employee_list_ref_t
);
/
```

循環参照を持つオブジェクト・ビューの作成

基礎となるリレーショナル表を定義した後、このリレーショナル表にオブジェクト・ビューを作成します。

方法 1: 2 つ目のビューを作成した後、最初のビューを再作成する

まず、deptref 列に NULL を持つ従業員ビューを作成します。後で、この列を参照に変えます。

```
CREATE VIEW emp_view OF emp_t WITH OBJECT IDENTIFIER(eno)
   AS SELECT e.empno, e.empname, e.salary,
                      NULL
         FROM emp e;
```

次に、従業員オブジェクトへの参照を含む部門ビューを作成します。

```
CREATE VIEW dept_view OF dept_t WITH OBJECT IDENTIFIER(dno)
   AS SELECT d.deptno, d.deptname,
               address_t(d.deptstreet,d.deptcity,d.deptstate,d.deptzip),
               CAST( MULTISET (
                         SELECT MAKE_REF(emp_view, e.empno)
                         FROM emp e
```

```
                        WHERE e.deptno = d.deptno)
                    AS employee_list_ref_t)
  FROM   dept d;
```

従業員全体のオブジェクトのかわりに、従業員オブジェクトへの参照のリストを作成します。ここで、部門ビューへの参照を持つ従業員ビューを再作成します。

```
CREATE OR REPLACE VIEW emp_view OF emp_t WITH OBJECT IDENTIFIER(eno)
  AS SELECT e.empno, e.empname, e.salary,
                    MAKE_REF(dept_view, e.deptno)
       FROM emp e;
```

これでビューが作成されました。

方法 2: FORCE キーワードを使用して最初のビューを作成する

ビューを作成する文に構文エラーがないという確信があれば、FORCE キーワードを使用して、他のビューを存在させずに、最初のビューを強制的に作成できます。

まず、この時点で存在しない部門ビューへの参照を含む従業員ビューを作成します。部門ビューが適切に作成されるまで、このビューの問合せは実行できません。

```
CREATE FORCE VIEW emp_view OF emp_t WITH OBJECT IDENTIFIER(eno)
  AS SELECT e.empno, e.empname, e.salary,
                    MAKE_REF(dept_view, e.deptno)
       FROM emp e;
```

次に、従業員オブジェクトへの参照を含む部門ビューを作成します。emp_view がすでに存在するため、ここで FORCE キーワードを使用する必要はありません。

```
CREATE VIEW dept_view OF dept_t WITH OBJECT IDENTIFIER(dno)
  AS SELECT d.deptno, d.deptname,
            address_t(d.deptstreet,d.deptcity,d.deptstate,d.deptzip),
            CAST( MULTISET (
                      SELECT MAKE_REF(emp_view, e.empno)
                      FROM emp e
                      WHERE e.deptno = d.deptno)
                  AS employee_list_ref_t)
  FROM   dept d;
```

これにより、部門ビューの問合せを実行し、従業員オブジェクトを取得できます。このオブジェクトを取得するには、ネストした表 empreflist からの従業員参照を解除します。

```
SELECT DEREF(e.COLUMN_VALUE)
FROM TABLE( SELECT e.empreflist FROM dept_view e WHERE e.dno = 100) e;
```

COLUMN_VALUE は、スカラーのネストした表の中のスカラー値を表す特別な名前です。この場合、COLUMN_VALUE はネストした表 empreflist の中にある従業員オブジェクトの参照を示しています。

名前が「John」で始まるすべての従業員の従業員番号のみにアクセスすることも可能です。

```
SELECT e.COLUMN_VALUE.eno
FROM TABLE(SELECT e.empreflist FROM dept_view e WHERE e.dno = 100) e
WHERE e.COLUMN_VALUE.ename like 'John%';
```

表形式で出力するには、部門表をネストした表の項目と結合し、参照のリストのネストを解除します。

```
SELECT d.dno, e.COLUMN_VALUE.eno, e.COLUMN_VALUE.ename
FROM dept_view d, TABLE(d.empreflist) e
WHERE e.COLUMN_VALUE.ename like 'John%'
 AND   d.dno = 100;
```

最後に、dept_view のかわりに emp_view を使用するように、前述の問合せを書きなおし、ビュー間をナビゲートする方法を紹介します。

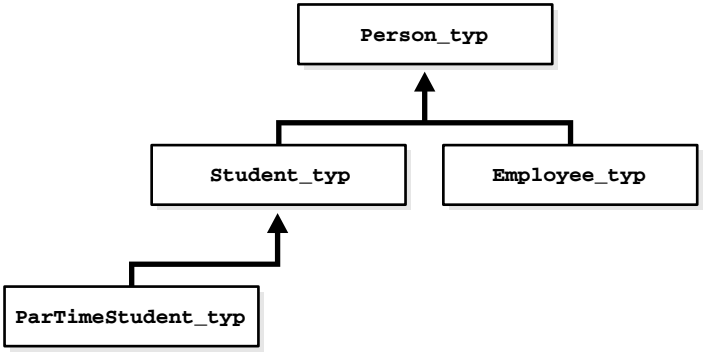
```
SELECT e.deptref.dno, DEREF(f.COLUMN_VALUE)
FROM emp_view e, TABLE(e.deptref.empreflist) f
WHERE e.deptref.dno = 100
AND f.COLUMN_VALUE.ename like 'John%';
```

オブジェクト・ビューの階層

オブジェクト・ビュー階層は、オブジェクト・ビューの集合で、型階層内の様々な型が、それぞれのオブジェクト・ビューの基になっています。ビュー階層内のサブビューは、型階層内のサブタイプがスーパータイプの下に作成される場合と同じ方法で、スーパービューの下に作成されます。

ビュー階層内のそれぞれのオブジェクト・ビューは、1 つの型のオブジェクトと一緒に移入されますが、任意のビューについての問合せでは、そのサブビューも暗黙的に扱います。このように、オブジェクト・ビュー階層により、一定のレベルまたはそれ以上のレベルに特化されたオブジェクトの多相集合を返す問合せを、簡単に組み立てられます。

Person_typ をルートとする、次のような型階層について考えてみます。



オブジェクト・ビューをそれぞれの型に基づいて作成し、この型階層に基づいてオブジェクト・ビュー階層を作成しておくと、目的の特化レベルと対応するオブジェクト・ビューの問合せができます。たとえば、Student_typ のビューについて問い合せ、学生（定時制の学生を含む）のみ入っている結果セットを取得できます。

型階層内のいずれの型も、オブジェクト・ビュー階層のルート・ビューの基として使用できます。したがって、オブジェクト・ビュー階層をルート型から組み立てる必要はありません。また、オブジェクト・ビュー階層を型階層の各リーフまで拡張したり、すべてのブランチを範囲に入れる必要もありません。ただし、下位に介在するサブタイプは省略できません。いずれのサブビューも、直系スーパービューの型の直系サブタイプに基づく必要があります。

型が兄弟関係にあるサブタイプを 2 つ以上持てるように、オブジェクト・ビューも兄弟関係にあるサブビューを 2 つ以上持てます。ただし、任意の型に基づくサブビューが関与できるのは、1 つのオブジェクト・ビュー階層のみです。2 つの異なるオブジェクト・ビュー階層に、同じサブタイプに基づく 1 つのサブビューが存在することはありません。

サブビューはスーパービューから OID を継承します。任意のサブビューに、OID を明示的に指定することは許可されません。

ルート・ビューでは、WITH OBJECT ID 句を使用して、OID を明示的に指定できます。OID がシステムにより生成されない場合、またはルート・ビューに句を指定しない場合は、サブビューを作成できます。ただし、これには、システム生成による OID を使用する表またはビューが、ルート・ビューの基になっていないという条件が付きます。

ビューの基礎となる問合せにより、ビューが更新可能かどうかが決まります。ビューの問合せに、結合、集合演算子、集計ファンクション、GROUP BY、DISTINCT、疑似列または式を何も入れないというのが、ビューを更新可能にするための条件です。サブビューについても同じ条件が適用されます。

ビューが更新可能でなくても、適切な DML 処理を実行するための INSTEAD OF トリガーが定義できます。サブビューは INSTEAD OF トリガーを継承しない点に注意してください。

ビュー階層内のすべてのビューは、同じスキーマに所属する必要があります。

注意： Oracle9*i* では、コンストラクタを持たない型のビューが作成できます。

コンストラクタを持たない型は、インスタンスを持つことはできません。したがって、このような型のオブジェクト・ビューを作成しても意味がありません。ただし、コンストラクタを持たない型は、コンストラクタを持つサブタイプを持つことができます。コンストラクタを持たない型のオブジェクト・ビューが作成できるため、コンストラクタを持たない型が含まれている型階層に基づくオブジェクト・ビュー階層が得られます。

オブジェクト・ビュー階層の作成

オブジェクト・ビュー階層は、ルート・ビューの下にサブビューを作成して、構築します。このサブビューの作成には、CREATE VIEW 文で UNDER キーワードを使用します。

```
CREATE VIEW Student_v OF Student_typ UNDER Person_v
  AS
  SELECT ssn, name, address, deptid, major
  FROM AllPersons
  WHERE typeid = 2;
```

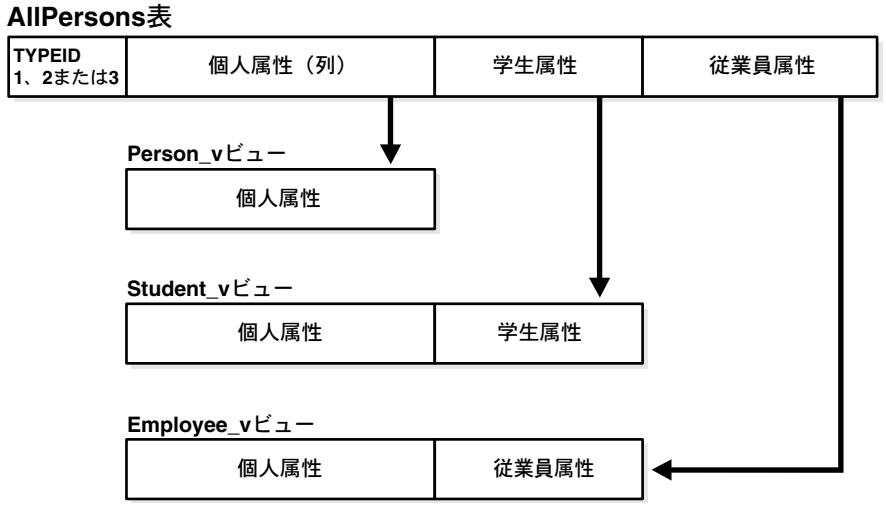
基礎となる様々な記憶域モデルが、同じオブジェクト・ビュー階層の基になることもあります。つまり、基礎となる表の様々なレイアウトや設計により、同じオブジェクト・ビュー階層が作成される可能性があるということです。基礎となる記憶域モデルの設計は、オブジェクト・ビュー階層のパフォーマンスや更新可能性も含んでいます。

次に、考えられる 3 種類の記憶域モデルの例を示します。最初の「フラット」モデルの場合、オブジェクト・ビュー階層内のすべてのビューの基は、同じ表です。2 番目の「水平」モデルの場合、それぞれのビューは、それぞれ別の表と 1 対 1 で対応します。3 番目の「垂直」モデルの場合は、結合を使用してビューが構成されます。

フラット・モデル

「フラット」モデルの場合、オブジェクト・ビュー階層内のすべてのビューの基となるのは、同じ表です。次に示す AllPersons という 1 つの表には、Person_typ、Student_typ または Employee_typ のすべての属性の列が格納されます。

図 5-1 オブジェクト・ビュー階層のフラット記憶域モデル




```
CREATE TABLE AllPersons
( typeid NUMBER(1),
  ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100),
  deptid NUMBER,
  major VARCHAR2(30),
  empid NUMBER,
  mgr VARCHAR2(30));
```

それぞれの行の型は、typeid 列により識別されます。可能な値は次のとおりです。

```
1 = Person_typ
2 = Student_typ
3 = Employee_typ
```

次の文により、オブジェクト・ビュー階層を構築するビューが作成されます。

```
CREATE VIEW Person_v OF Person_typ
  WITH OBJECT OID(ssn) AS
  SELECT ssn, name, address
  FROM AllPersons
  WHERE typeid = 1;

CREATE VIEW Student_v OF Student_typ UNDER Person_v
  AS
```

```
  SELECT ssn, name, address, deptid, major
  FROM AllPersons
  WHERE typeid = 2;

CREATE VIEW Employee_v OF Employee_typ UNDER Person_v
  AS
  SELECT ssn, name, address, empid, mgr
  FROM AllPersons
  WHERE typeid = 3;
```

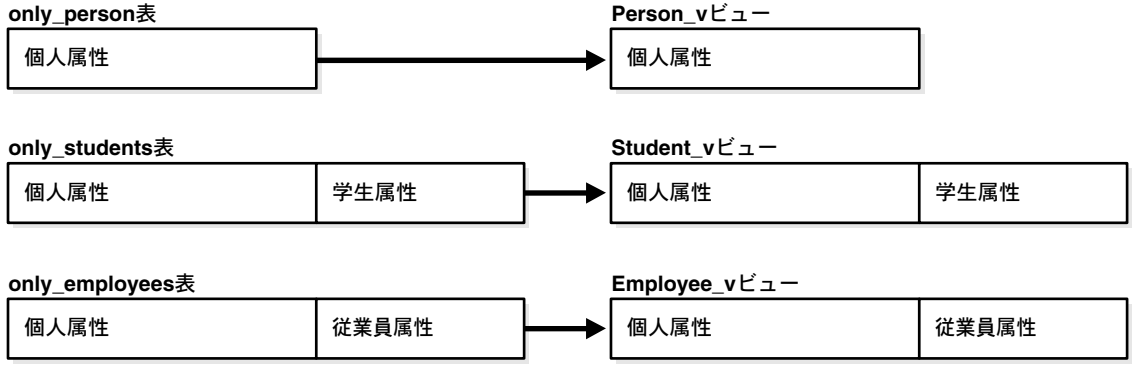
単純である、というのがフラット・モデルの利点です。また、フラット・モデルが、索引や制約への対応に支障をきたす原因になることはありません。このモデルのデメリットは、次のとおりです。

- 1 つの表に格納できる列は 1000 以下のため、フラット・モデルでは、オブジェクト・ビュー階層に格納できる列の合計数が、1000 列に制限されます。
- 表の各列には、オブジェクト・ビューの型に属さないすべての属性のかわりに NULL が入ってしまいます。このように NULL が後続しない状態は、パフォーマンスを低下させる可能性があります。

水平モデル

水平モデルの場合、それぞれのビューまたはサブビューの基になるのは、異なる表です（例に示す表はリレーショナル表ですが、これらの表は、列の代入性が無効になるオブジェクト表になる場合もあります。）

図 5-2 オブジェクト・ビュー階層の水平記憶域モデル




```
CREATE TABLE only_persons
( ssn NUMBER,
  name VARCHAR2(30),
```

```
  address VARCHAR2(100));

CREATE TABLE only_students
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100),
  deptid NUMBER,
  major VARCHAR2(30));

CREATE TABLE only_employees
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100),
  empid NUMBER,
  mgr VARCHAR2(30));
```

次に示すのは、ビューです。

```
CREATE VIEW Person_v OF Person_typ
  WITH OBJECT OID(ssn) AS
  SELECT *
  FROM only_persons

CREATE VIEW Student_v OF Student_typ UNDER Person_v
  AS
  SELECT *
  FROM only_students;

CREATE VIEW Employee_v OF Employee_typ UNDER Person_v
  AS
  SELECT *
  FROM only_employees;
```

水平モデルを使用すると、非常に効率的なのは、次のような構成の問合せです。

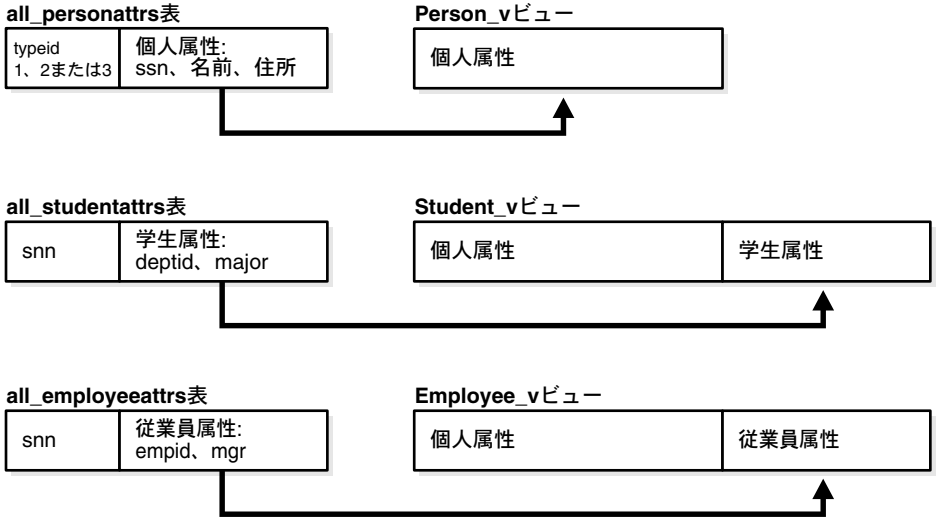
```
SELECT VALUE(p) FROM Person_v p
WHERE VALUE(p) IS OF (ONLY Student_typ);
```

このような問合せでは、特定の型のオブジェクトをすべて取得するために、1 つの物理表にアクセスするのみで済みます。このモデルのデメリットは、`SELECT * FROM view` のような問合せでは、基礎となるすべての表に対して `UNION` を実行し、指定されたビューの列に対してのみ行を投影する必要があることです（5-26 ページの「階層内のビューの問合せ」を参照）。属性（および一意制約）についての索引は複数の表をまたがる必要がありますが、現在この機能はサポートされていません。

垂直モデル

垂直モデルの場合、階層内のそれぞれのビューと対応する物理表が存在しますが、それぞれの物理表に格納されるのは、対応するサブタイプについて固有の属性のみです。

図 5-3 オブジェクト・ビュー階層の垂直記憶域モデル




```
CREATE TABLE all_personattrs
( typeid NUMBER,
  ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100));

CREATE TABLE all_studentattrs
( ssn NUMBER,
  deptid NUMBER,
  major VARCHAR2(30));

CREATE TABLE all_employeeattrs
( ssn NUMBER,
  empid NUMBER,
  mgr VARCHAR2(30));

CREATE VIEW Person_v OF Person_t
WITH OBJECT OID(ssn) AS
  SELECT ssn, name, address
  FROM all_personattrs
```

```
  WHERE typeid = 1;

CREATE VIEW Student_v OF Student_t UNDER Person_v
  AS
  SELECT x.ssn, x.name, x.address, y.deptid, y.major
  FROM all_personattrs x, all_studentattrs y
  WHERE x.typeid = 2 AND x.ssn = y.ssn;

CREATE VIEW Employee_v OF Employee_t UNDER Person_v
  AS
  SELECT x.ssn, x.name, x.address, y.empid, y.mgr
  FROM all_personattrs x, all_studentattrs y
  WHERE x.typeid = 3 AND x.ssn = y.ssn;
```

垂直モデルでは、SELECT * FROM *root_view* のような問合せが効率的に処理されます。また、個々の属性に索引を付けたり、属性に一意制約を付けることができます。ただし、型のインスタンスを再作成するには、それぞれのレベルについて、階層のルートから型を削除する OID の結合を実行する必要があります。

階層内のビューの問合せ

オブジェクト・ビュー階層内の任意のビューまたはサブビューに対し、問合せを実行できます。問合せにより返されるのは、問い合せたビューの宣言された型の行と、その型のサブタイプの行です。たとえば、Person_typ 型階層が基になっているオブジェクト・ビュー階層では、Person_typ のビューを問い合せて、学生や従業員を含む個人全員が格納されている結果セットを取得したり、Student_typ のビューを問い合せて、学生（定時制の学生を含む）のみ格納されている結果セットを取得することができます。

問合せの SELECT 構文のリストには、オブジェクト・インスタンスを戻す REF() や VALUE() のようなファンクションを含めるか、または Person_typ の name や ssn 属性のようなビューの宣言された型のオブジェクト属性を指定できます。

オブジェクト・インスタンスを戻すためにファンクションを指定すると、問合せにより多相結果セットが戻されます。つまり、ビューの宣言された型のインスタンスと、その型のサブタイプのインスタンスのどちらも戻されます。

たとえば、次の問合せでは、あらゆる型の個人、従業員および学生のインスタンスのみでなく、これらのインスタンス REF も戻されます。

```
SELECT REF(p), VALUE(p) FROM Person_v p;
```

ビューの宣言された型の個々の属性を SELECT 構文のリストに指定した場合、または SELECT * を実行した場合も、ビューの宣言された型およびこの型のサブタイプの行が戻されます。ただし、これらの行が投影される先は、ビューの宣言された型の属性の列で、使用されるのは、これらの列のみです。つまり、サブタイプを表現する対象は、ビューの宣言された型からサブタイプが継承した属性、およびサブタイプがビューの宣言された型と共有する属性のみです。

たとえば、次の問合せでは、個人全員の行およびあらゆる型の従業員および学生の行が戻されますが、結果に使用されるのは、Person_typ の属性（name、ssn および address）の列のみです。Student_typ の deptid 属性などサブタイプに追加された属性の行は、何も表示されません。

```
SELECT * FROM Person_v;
```

結果からサブビューを除外するには、ONLY キーワードを使用します。ONLY キーワードにより、選択対象が、問合せ対象のビューの宣言された型に限定されます。

```
SELECT VALUE(p) FROM ONLY(Person_v) p;
```

ビュー階層についての操作の権限

通常、サブビューを持つビューについての問合せで要求されるは、参照対象のビューについての SELECT 権限のみで、サブビューについての明示的な権限は要求されません。次に示す問合せでは、Person_v についての SELECT 権限のみ要求され、このビューのサブビューについての権限は要求されません。

```
SELECT * FROM Person_v;
```

ただし、サブタイプの属性として追加されていても、ルート型では使用しない属性を選択する問合せの場合は、サブビューについての SELECT 権限も要求されます。このようなサブタイプ属性には、別のアクセス権限を適度に必要とする機密情報が格納されている可能性があります。

次に示す問合せでは、オブジェクト・インスタンスが選択され、その結果、サブタイプのすべての属性が取り出されるため、Person_v、Student_v、Employee_v（および Person_v のサブビュー）についての SELECT 権限が要求されます。

```
SELECT VALUE(p) FROM Person_v p;
```

ビュー階層全体についての SELECT 権限を付与する処理を簡素化するには、HIERARCHY オプションが使用できます。ビューについての SELECT 権限をユーザーに付与するときに、HIERARCHY オプションを指定すると、現在および将来存在するビューのすべてのサブビューについての SELECT 権限も暗黙的に付与されます。次に例を示します。

```
GRANT SELECT ON Person_v TO scott WITH HIERARCHY OPTION;
```

サブビューに属する行を除外する問合せには、すべてのサブビューについての SELECT 権限も要求されます。その理由は、インスタンスの最も狭い意味での型にのみ属している行の情報は、機密情報の可能性があり、すべての行をサブビューから除外する問合せ（次に示すような問合せ）の場合は、サブビューについての SELECT 権限が要求されるためです。

```
SELECT * FROM ONLY(Person_v);
```

6

Oracle オブジェクトの高度なトピック

このマニュアルの他の章では、Oracle オブジェクトで作業を開始するために必要なトピックについて説明しています。この章では、オブジェクト・リレーショナル技法を大規模なアプリケーションまたは複雑なスキーマに適用する場合について説明します。

内容は次のとおりです。

- オブジェクトの記憶域
- 型 ID または属性の索引作成
- オブジェクト識別子（OID）
- 型進化
- ユーザー定義コンストラクタ
- オブジェクトに対する OCI のヒントおよび技法
- 一時型および汎用型
- ユーザー定義集計ファンクション
- Oracle オブジェクトを持つ表のパーティション化

オブジェクトの記憶域

Oracle は、オブジェクト型の複合構造を、単純な構造の表に自動的にマップします。

リーフ・レベル属性

オブジェクト型はツリー構造に似ており、ブランチが属性を表します。それ自体がオブジェクトである属性は、それ自体の属性に対してサブブランチを発生させます。

最終的に、各ブランチの終わりは、組込み型（NUMBER、VARCHAR2、REF など）またはコレクション型（VARRAY、ネストした表など）の属性になります。元のオブジェクト型のこのようなリーフ・レベル属性は、それぞれ表の列に格納されます。

コレクション型ではないリーフ・レベル属性は、オブジェクト型のリーフ・レベル・スカラー属性といわれます。

列にまたがって分割される行オブジェクト

オブジェクト表では、すべてのリーフ・レベル・スカラー属性または REF 属性に対するデータが、個々の列に格納されます。各 VARRAY も、大きすぎないかぎり、列に格納されます（6-5 ページの「VARRAY の内部レイアウト」を参照）。ネストした表型のリーフ・レベル属性は、オブジェクト表に対応付けられた個々の表に格納されます。これらの表は、オブジェクト表宣言の一部として宣言する必要があります（6-4 ページの「ネストした表の内部レイアウト」を参照）。

オブジェクト表にあるオブジェクトの属性を取り出したり、変更する場合、対応する操作が表の列で実行されます。オブジェクト自体の値にアクセスすると、オブジェクト表の列を引数として使用し、その型のデフォルトのコンストラクタを起動することによって、オブジェクトのコピーが生成されます。

システムによって生成された OID は、非表示列に格納されます。Oracle は、OID を使用して、そのオブジェクトに対する REF を作成します。

列オブジェクトを持つ表の非表示列

表がオブジェクト型の列とともに定義されている場合、オブジェクト型のリーフ・レベル属性に対する表に、非表示列が追加されます。オブジェクト型の各列には、その列オブジェクトの NULL 情報（トップレベルのオブジェクトおよびネストしたオブジェクトのアトミック NULL）を格納するための非表示列があります。

代入可能な列および表の非表示列

代入可能な列またはオブジェクト表には、その列オブジェクトの各属性に対してのみでなく、そのオブジェクト型のサブタイプに追加された各属性に対しても、非表示列があります。これらの列には、代入可能な列に挿入されたサブタイプのインスタンスに対する属性の値が格納されます。

たとえば、Person_typ の代入可能な列には、Person_typ の各属性、具体的には ssn、name、address に対する非表示列が対応付けられます。また、Person_typ のサブタイプの属性、たとえば、Student_typ に対する deptid と major、および PartTimeStudent_typ に対する numhours の非表示列も作成されます。

サブタイプが作成されると、そのサブタイプに追加された属性の非表示列が、その新しいサブタイプの祖先クラスの代入可能な列を含む表に自動的に追加されます。追加された非表示列は、表にレトロフィットして、新しい型のデータを格納できるようになります。なんらかの理由で、非表示列を追加できない場合は、そのサブタイプの作成はロールバックされます。

サブタイプが DROP TYPE に対する VALIDATE オプションを使用して削除された場合は、各サブタイプの固有の属性に対するこのような非表示列はすべて、自動的に削除されます。

代入可能な列には、非表示の**型判別式の**列が対応付けられます。この列には、**型 ID** と呼ばれる識別子があり、この識別子によって、代入可能な列中の各オブジェクトのうち、最も具体的な型が識別されます。通常、型 ID（RAW）は 1 バイトですが、大きな階層の場合は最大 4 バイトまでとなります。

指定されたオブジェクト・インスタンスの型 ID は、ファンクション SYS_TYPEID を使用して特定できます。たとえば、代入可能なオブジェクト表 persons に次の 3 つの行があるとします。

```
CREATE TABLE persons OF Person_typ;

INSERT INTO persons
  VALUES (Person_typ(1243, 'Bob', '121 Front St'));

INSERT INTO persons
  VALUES (Student_typ(3456, 'Joe', '34 View', 12, 'HISTORY'));

INSERT INTO persons
  VALUES (PartTimeStudent_typ(5678, 'Tim', 13, 'PHYSICS', 20));
```

次の問合せによって、この表に格納されているオブジェクト・インスタンスの型 ID が取得されます。

```
SELECT name, SYS_TYPEID(VALUE(p)) typeid FROM persons p;

NAME   TYPEID
----   ------
Bob    01
Joe    02
Tim    03
```

カタログ・ビューの USER_TYPES、DBA_TYPES および ALL_TYPES には、各型に対する型 ID 値を与える TYPEID 列（非表示ではない）があります。この列に対して結合を実行して、型判別式の列にある型 ID に対応する型名を取得できます。

参照： SYS_TYPEID および型 ID の詳細は、第 2 章の「SYS_TYPEID」を参照してください。

REF

行オブジェクトに対して作成される REF は、OID、オブジェクト表のいくつかのメタデータおよび ROWID（オプション）で構成されます。

REF 型の列にある REF のサイズは、その列に対応付けられた記憶域プロパティに依存します。たとえば、列が REF WITH ROWID として宣言されている場合、ROWID は REF 列に格納されます。ROWID のヒントは、制約付き REF 列にあるオブジェクト参照では無視されます。

列が SCOPE 句を使用して REF として（有効範囲付き REF として）宣言されている場合、オブジェクト表のメタデータおよび ROWID は省略されて、その列は小さくなります。有効範囲付き REF の長さは、16 バイトです。

OID が主キー・ベースの場合、主キーを導出する列の数に基づいて、主キーの値を格納するための 1 つ以上の内部列が作成されます。

注意： OID が主キーから導出される行オブジェクトを REF 列が参照する場合、この REF 列を、主キー・ベース REF または pkREF といいます。pkREF を含む列は、有効範囲付きであるかまたは参照制約が指定されている必要があります。

ネストした表の内部レイアウト

ネストした表の行は、別の記憶表に格納されます。ネストした表には、行ごとに 1 つの記憶表ではなく、列ごとに 1 つの記憶表が対応付けられています。記憶表は、その列にあるすべてのネストした表に対するすべての要素を保持しています。記憶表には、システム生成の値を持つ非表示の NESTED_TABLE_ID 列があります。この値によって、ネストした表の要素が適切な行にマップされます。

記憶表を索引構成表（IOT）にすることによって、コレクション全体を取り出す問合せを高速化できます。ORGANIZATION INDEX 句を STORE AS 句の中に挿入します。

ネストした表型は、オブジェクトまたはスカラーを含むことができます。

- 要素がオブジェクトである場合、記憶表はオブジェクト表のようになります。つまり、そのオブジェクト型のトップレベルの属性が記憶表の列となります。ただし、ネストした表の行には OID の列がないため、ネストした表のオブジェクトに対する REF の作成はできません。
- 要素がスカラーである場合、記憶表には、スカラー値を含む COLUMN_VALUE という単一の列が含まれます。

詳細は、8-15 ページの「ネストした表の記憶域」を参照してください。

VARRAY の内部レイアウト

VARRAY のすべての要素は、単一の列に格納されます。配列のサイズに基づいて、VARRAY はインラインまたは BLOB に格納されます。詳細は、8-14 ページの「VARRAY の記憶域上の考慮点」を参照してください。

型 ID または属性の索引作成

型判別式の列の索引付け

SYS_TYPEID ファンクションを使用して、代入可能な各列にある非表示の型判別式の列に索引を作成できます。型判別式の列には、型 ID があり、この識別子によって、代入可能な列中の各オブジェクト・インスタンスのうち、最も具体的な型が識別されます。この情報は、型に基づいてフィルタにかける IS OF 述語を使用する問合せの評価に使用されるものですが、SYS_TYPEID ファンクションを使用して独自の目的で型 ID にアクセスすることも可能です。

注意： 一般に、型判別式の列には、わずかな数の型 ID のみが格納されています。関連のある型階層に含まれている型の数を超えることはありません。この列はカーディナリティが低いので、ビットマップ索引の作成に適しています。

たとえば、次の文では、表 books の代入可能な author 列の基礎となる型判別式の列のビットマップ索引を作成します。SYS_TYPEID ファンクションを使用して、型判別式の列が参照されます。

```
CREATE BITMAP INDEX typeid_i ON books (SYS_TYPEID(author));
```

代入可能なサブタイプの索引付け

代入可能な列内に格納できる任意の型の属性に対して索引を作成できます。サブタイプの属性は、TREAT ファンクションを使用して求めるサブタイプ（およびそのサブタイプ）以外の型をフィルタにかけて排除することによって CREATE INDEX 文で参照できます。その後、ドット表記によって、希望の属性を指定します。

たとえば、次の文では、books 表にある、すべての学生著者の major 属性に対する索引が作成されます。author 列の宣言された型は Person_typ で、Student_typ がそのサブタイプです。したがって、この列には Person_typ、Student_typ およびいずれかのサブタイプが格納されます。

```
CREATE INDEX major_i ON books
  (TREAT(author AS Student_typ).major);
```

Student_typ は、最初に major 属性を定義した型です。Person_typ スーパータイプにはこの型はありません。したがって、major 属性に対する非表示列のすべての値は、Student_typ または PartTimeStudent_typ の著者（Student_typ サブタイプ）に対する値です。つまり、非表示列の値は、TREAT 式によって戻される値と同一であることを意味しています。この式は、学生のサブタイプも含めて、すべての学生の major の値を戻します。つまり、非表示列も、TREAT 式もともに、学生については専攻を、他の型の著者については NULL をリストします。このことを利用して、非表示列に対する索引 major_i が、通常の B ツリー索引として作成されます。

非表示列の値は、前述のように、TREAT 式によって戻される値と同一です。ただし、TREAT ファンクションのターゲットとして指定された型（前述の例では、Student_typ）が最初にその属性を定義した型である場合にかぎり、そうなります。TREAT ファンクションのターゲットが、次の例のように、単にその属性を継承するサブタイプである場合は、TREAT 式は、サブタイプ（定時制の学生）に対しては NULL 以外の major 値を戻しますが、スーパータイプ（その他の学生）に対しては非 NULL は戻しません。

```
CREATE INDEX major_func_i ON books
  (TREAT(author AS PartTimeStudent_typ).major);
```

この場合、非表示列に major の値として格納されている値は、TREAT 式の結果とは異なっている場合があります。したがって、通常の B ツリー索引を基礎となる列に対して作成することはできません。

このような場合、Oracle では、TREAT 式は他のすべてのファンクション・ベースの式と同様に処理され、索引を、結果に対するファンクション索引として作成する試みが行われます。ただし、ファンクション索引を作成するには、B ツリー索引の作成には不要な、一定の権限およびセッションの設定が必要です。

次の例では、前述の例と同様に、定時制の学生の major 属性に対してファンクション索引を作成しますが、この例では、major の非表示列は、代入可能なオブジェクト表 persons に対応付けられています。

```
CREATE INDEX major_func_i2 ON persons p
  (TREAT(VALUE(p) AS PartTimeStudent_typ).major);
```

オブジェクト識別子（OID）

オブジェクト表にあるすべての行オブジェクトには、対応付けられた論理 OID があります。デフォルトでは、システム生成の一意の OID（16 バイト長）が、各行オブジェクトに割り当てられます。Oracle では、OID のドキュメントも、OID の内部構造へのアクセスも提供されていません。この構造は随時変更される可能性があります。

オブジェクト表の OID 列は非表示列です。オブジェクト表の OID 列が一度設定されると、これを無視できます。かわりに、オブジェクト参照を介してのオブジェクトのフェッチおよびナビゲートに集中できます。

行オブジェクトに対する OID は、オブジェクト表にある行オブジェクトを一意に識別します。オブジェクト表の OID 列に対して索引が暗黙的に作成され、メンテナンスされます。分散環境およびレプリケート環境では、システム生成の一意の識別子によって、オブジェクトが一意に識別されます。

主キー・ベースの OID

ローカルに一意の識別子がグローバルにも一意であると想定できる環境（表が分散化もレプリケートもされない環境）では、行オブジェクトの主キー値をその OID として使用できます。そうすることによって、システム生成の識別子に必要な 16 バイトの記憶域を節約できます。

また、主キー・ベースの識別子を利用することで、オブジェクト表へのデータのロードを高速かつ容易に行うことができます。それに対して、システム生成の OID は、それらの OID の参照も格納される場合は特に、ユーザーが指定したキーを使用して再度マップする必要があります。

型進化

ユーザー定義型の変更を、**型進化**といいます。ユーザー定義型は、次の方法で変更できます。

- 属性の追加と削除
- メソッドの追加と削除
- 数値属性の変更による、その長さ、精度またはスケールの増大
- 可変長文字属性の変更による、その長さの増大
- 型の `FINAL` プロパティおよび `INSTANTIABLE` プロパティの変更

型を変更すると、その型を参照する項目にも影響します。たとえば、新しい属性を型に追加する場合、その新しい属性を追加できるように、その型の列にあるデータを提示する必要があります。

型を直接的または間接的に参照するスキーマ・オブジェクトで、その型に加えられた変更に影響されるものを、その型の**依存オブジェクト**といいます。型は、次の依存オブジェクトを持つことができます。

- 表
- 型またはサブタイプ
- プログラム・ユニット（PL/SQL ブロック）: プロシージャ、ファンクション、パッケージ、トリガー
- 索引タイプ
- ビュー（オブジェクト・ビューも含む）

- ファンクション索引
- 演算子

依存スキーマ・オブジェクトが型の変更によってどのような影響を受けるかは、オブジェクトによって、また、型に加えられた変更の性質によって異なります。

すべての依存プログラム・ユニット、ビュー、演算子および索引タイプは、型が変更されると、無効としてマークされます。これらの無効となったスキーマ・オブジェクトのいずれかが次回参照されると、新しい型の定義を使用して再検証されます。オブジェクトの再コンパイルが成功すると、そのオブジェクトは有効となり、再度使用できるようになります（型に加えられた変更によっては、ファンクション索引が削除または無効化され、再構築が必要です）。

型に依存表がある場合は、型に加えられた各属性に対して、新しい属性の型に依存する表に 1 つ以上の内部列が追加されます。新しい属性が追加される際は、NULL 値が設定されます。削除された各属性に対応付けられている列は削除されます。変更された各属性に対応付けられた長さ、精度またはスケールも、変更内容に対応して変更されます。

これらの変更には、主に表のメタデータ（表の構造に関する情報で、その列および列の型を記述するデータ）の更新がかかわっているので、迅速な処理が必要です。さらに、このような表データは、型の新しいバージョンの形式に合わせた更新も必要です。このデータの更新は、データの量によっては時間がかかる場合があるため、ALTER TYPE コマンドにオプションが用意されています。このオプションによって、すべての依存表データを即時変換するか、元の形式のままにしておいて、更新のたびに少しずつ変換するかを選択できます。

ALTER TYPE の CASCADE オプションは、型の変更を依存型および依存表に伝播します（6-15 ページの「型進化に使用する ALTER TYPE オプション」を参照）。CASCADE 自体に、伝播の一環として表データを新しい型形式に変換するかどうかを選択できるオプションがあります。INCLUDING TABLE DATA オプションはデータを変換し、NOT INCLUDING TABLE DATA オプションはデータを変換しません。デフォルトでは、CASCADE オプションはデータを変換します。いずれの場合も、表データは常に最新バージョンの型の形式で戻されます。表データが以前の型バージョンの形式で保存されている場合は、データが実際に保存されている形式はデータが再書込みされないかぎり変更されないにもかかわらず、そのデータは、戻される前に最新バージョンの形式に変換されます。

最新の型の定義は、システム・ビュー USER_SOURCE から取り出せます。USER_TYPE_VERSIONS ビューで、ある型のすべてのバージョンの定義を参照できます。

次の例では、ある属性を追加し、別の属性を削除することによって Person_typ を変更しています。CASCADE キーワードは、型の変更を依存型および依存表に伝播しますが、NOT INCLUDING TABLE DATA 句によって関連するデータの変換が阻止されています。

```
CREATE TYPE person_typ AS OBJECT (
  first_name   VARCHAR(30),
  last_name    VARCHAR(30),
  age          NUMBER(3));

CREATE TABLE person_tab of person_typ;
```

```
INSERT INTO person_tab VALUES
  (person_typ ('John', 'Doe', 50));

SELECT value(p) FROM person_tab p;

VALUE(P)(FIRST_NAME, LAST_NAME, AGE)
--------------------------------------------
PERSON_TYP('John', 'Doe', 50)

ALTER TYPE person_typ
  ADD ATTRIBUTE (dob DATE),
  DROP ATTRIBUTE age CASCADE NOT INCLUDING TABLE DATA;

-- The data of table person_tab has not been converted yet, but
-- when the data is retrieved, Oracle returns the data based on
-- the latest type version. The new attribute is initialized to NULL.

SELECT value(p) FROM person_tab p;
VALUE(P)(FIRST_NAME, LAST_NAME, DOB)
--------------------------------------------
PERSON_TYP('John', 'Doe', NULL)
```

SELECT 文では、列データが最新の型バージョンに変換されている可能性があるとしても、変換されたデータが列に再度書き込まれることはありません。ある表の特定のユーザー定義型の列が頻繁に取り出される場合、データ変換が重複して行われないようにするため、そのデータを最新の型バージョンに変換することを検討してください。変換は、列に VARRAY 属性が含まれている場合、特に大きな効果があります。通常、VARRAY の方がオブジェクトやネストした表より変換に必要な時間が長くなります。

データの列は、UPDATE 文を発行して、その列を自らに設定することによって変換できます。次に例を示します。

```
UPDATE dept_tab SET emp_array_col = emp_array_col;
```

ある表内のすべての列を、ALTER TABLE UPGRADE DATA を使用して変換できます。

次に例を示します。

```
ALTER TYPE person_typ ADD ATTRIBUTE (photo BLOB)
  CASCADE NOT INCLUDING TABLE DATA;
ALTER TABLE dept_tab UPGRADE INCLUDING DATA;
```

型を変更する際に必要な変更

型に対する構造的な変更のみが依存データに影響し、データの変換を必要とします。ある型のメソッドの定義や動作（型のメソッドが実装されている場合の型本体の動作）に限定された変更は影響しません。

型に対する次の変更は構造的変更です。

- 属性の追加
- 属性の削除
- 属性の長さ、精度、スケールの変更
- 型のファイナリティ(サブタイプを導出できるかどうかを決定する)の FINAL から NOT FINAL への変更、または NOT FINAL から FINAL への変更

このような変更の結果、変更された型およびそのすべての依存型のバージョンが新しいバージョンになり、新しいバージョンへの変更のプロセスの一環として、依存表の内部列が追加、削除または変更されます。

これらのいずれかの変更を、依存型または依存表を持つ型に対して実行すると、変更を伝播した影響は、メタデータのみでなく、データ記憶域の構成にも影響し、データの変換を必要とします。

データの変換以外にも、変更すべき項目がある場合があります。たとえば、新しい属性が型に追加されて、その型が、その型のコンストラクタをコールする場合、型本体の各コンストラクタは、新しい属性に対する値を指定するように修正する必要があります。同様に、新しいメソッドが追加される場合は、型本体を置換して、新しいメソッドの実装を追加する必要があります。型本体は、CREATE OR REPLACE TYPE BODY 文を使用して変更できます。

型の変更の手順

型を変更する手順は次のとおりです。

次のスキーマについて考えてみます。

```
CREATE TYPE Person_typ AS OBJECT
( name     CHAR(20),
  ssn      CHAR(12),
  address  VARCHAR2(100));

CREATE TYPE Person_nt IS TABLE OF Person_typ;
CREATE TYPE dept_typ AS OBJECT
( mgr    Person_typ,
  emps   Person_nt);

CREATE TABLE dept OF dept_typ;
```

1. この型を変更するため、ALTER TYPE 文を発行します。

   ALTER TYPE 文にオプションが指定されていない場合、デフォルトの動作では、ターゲットの型に依存するオブジェクトが存在しないかを調べます。なんらかの依存オブジェクトが存在する場合は、この文は異常終了します。オプションのキーワードによって、型の変更を依存型および依存表に連鎖的に波及させることができます。

   次のコードでは、NOT INCLUDING TABLE DATA 句を追加することによって、表データの変換が遅延されています。

   ```
   -- Add new attributes to Person_typ and propagate the change
   -- to Person_nt and dept_typ
   ALTER TYPE Person_typ ADD ATTRIBUTE (picture BLOB, dob DATE)
     CASCADE NOT INCLUDING TABLE DATA;
   ```

2. CREATE OR REPLACE TYPE BODY を使用して、対応する型本体を更新して、新しい型定義に更新します。

3. 依存表を最新の型バージョンにアップグレードし、表データを変換します。

   ```
   ALTER TABLE dept UPGRADE INCLUDING DATA;
   ```

4. 必要に応じて、PL/SQL 依存プログラム・ユニットを変更して、型に加えられた変更に対応します。

5. アプリケーションが C 言語で書かれているか、Java で書かれているかによって、OTT または JPublisher（または別のツール）を使用して、新しいヘッダー・ファイルを生成します。

   スーパータイプに新しい属性を追加すると、サブタイプの属性が新しい属性を継承するために、そのすべてのサブタイプの属性数が増大する可能性があります。継承された属性は、常に、宣言された（ローカルに定義された）属性に先行するので、新しい属性をスーパータイプに追加すると、各サブタイプのすべての宣言された属性の順序が 1 つずつ再帰的に増加されます。変更された型のマッピングを更新して、新しい属性を追加する必要があります。OTT と JPublisher がこのタスクを実行します。他のツールを使用する場合は、型のヘッダーが、サーバーでの型の定義と適切に同期されることを確認する必要があります。この同期が確実に行われない場合は、結果として予期しない動作が発生する可能性があります。

6. 必要に応じてアプリケーション・コードを修正し、アプリケーションを再作成します。

型の妥当性チェック

ALTER TYPE 文の実行にあたって、最初に、要求された型の変更について、構文上およびセマンティクス上の妥当性がチェックされます。CREATE TYPE 文についても同様の妥当性チェックおよび追加の妥当性チェックが実行されます。たとえば、削除の対象となっている属性が、パーティション化キーで使用されていないかが確認されます。ターゲットの型の新しい仕様またはその依存型のいずれかについて型の妥当性が確認されない場合は、ALTER

TYPE 文は異常終了します。この場合は、新しい型バージョンは作成されず、すべての依存オブジェクトが変更されないままになります。

依存表が存在する場合は、さらにチェックが行われて、表および索引に関連する制約が遵守されているかが確認されます。ここでも、ALTER TYPE 文について、表に関連する制約のチェックが失敗すると、型の変更は異常終了し、その型の新しいバージョンは作成されません。

複数の属性が単一の ALTER TYPE 文に追加される場合は、指定された順序で追加されます。1 つの ALTER TYPE 文で複数の型の変更を指定することはできますが、属性名およびメソッド・シグネチャは 1 回のみの指定となります。たとえば、単一の文で同じ属性の追加と変更を行うことはできません。

たとえば、次のようになります。

```
CREATE TYPE mytype AS OBJECT (attr1  NUMBER, attr2 NUMBER);
ALTER TYPE mytype ADD ATTRIBUTE (attr3 NUMBER),
  DROP ATTRIBUTE attr2,
  ADD ATTRIBUTE attr4 NUMBER CASCADE;
```

この結果、mytype の定義は次のようになります。

```
  (attr1 NUMBER, attr3 NUMBER, attr4 NUMBER);
```

次の ALTER TYPE 文は、同じ属性（attr5）に対して複数の変更を加えようとするもので、これは無効です。

```
-- invalid ALTER TYPE statement
ALTER TYPE mytype ADD ATTRIBUTE (attr5 NUMBER, attr6 CHAR(10)),
  DROP ATTRIBUTE attr5;
```

次に、妥当性チェックに対する制約、表の制限、および型に対して可能な様々な種類の変更についてのその他の情報に関する注意事項を示します。

属性の削除

- ルート型に由来するすべての属性を削除することはできません。属性ではなく型を削除する必要があります。サブタイプのすべての属性はスーパータイプから継承されるため、サブタイプからすべての属性を削除しても、その属性のカウントは 0（ゼロ）にはなりません。サブタイプでローカルに宣言された属性を削除することは許可されます。
- ターゲットの型でローカルに宣言された型のみを削除できます。継承された属性を、サブタイプから削除することはできません。属性がローカルに宣言された場所でその型からその属性を削除してください。
- 表パーティションの一部である属性または表のサブパーティション・キーの削除はできません。
- オブジェクト表または IOT の主キー OID の属性の削除はできません。

- 属性が削除されると、削除された属性に対応する列は削除されます。
- 索引、統計、制約、および削除された属性を参照する参照整合性制約は削除されます。

属性の型の変更（長さ、精度またはスケールの増大）

- 依存表のファンクション索引、クラスタ・キーまたはドメイン索引で参照される属性の長さは拡張できません。

メソッドの削除

- メソッドは、そのメソッドの定義またはオーバーライドが行われた型からのみ削除できます。継承されたメソッドをサブタイプから削除することはできません。また、スーパータイプからオーバーライドを削除することもできません。
- メソッドがオーバーライドされていない場合、CASCADE オプションを使用してそのメソッドを削除すると、ターゲットの型に由来するメソッドおよびすべてのサブタイプが削除されます。一方、メソッドがオーバーライドされている場合は、CASCADE は実行されないで、ロールバックされます。CASCADE が実行されるためには、サブタイプから、サブタイプを定義する各オーバーライドを削除して、それから、スーパータイプからメソッドを削除します。

  USER_DEPENDENCIES 表に、型も含めて、与えられた型に依存するすべてのスキーマ・オブジェクトがあります。
- INVALIDATE オプションを使用して、オーバーライドのあるメソッドを削除できますが、サブタイプにあるオーバーライドは、依然として手動で削除する必要があります。サブタイプは、オーバーライドを削除するように明示的に変更されないかぎり、無効な状態のままです（それまでは、妥当性チェックのためにサブタイプを再コンパイルしようとすると、「メソッドはオーバーライドしません。」というエラーが発生します）。

  CASCADE とは異なり、INVALIDATE は、型および表のすべてのチェックを迂回して、その型に依存するすべてのスキーマ・オブジェクトを無効とします。それらのオブジェクトは、次回のアクセス時に再検証されます。このオプションは、CASCADE を使用するより高速に処理されますが、依存型および依存表の再検証時に問題が発生しないことが確認されている必要があります。表が無効な場合は、表データにはアクセスできません。表の妥当性が検証できない場合は、そのデータはアクセス不可のままとなります。

  参照： 6-14 ページの「型の変更の妥当性チェックに失敗した場合」を参照してください。

FINAL プロパティまたは INSTANTIABLE プロパティの修正

- その型に表依存オブジェクトがない場合にかぎり、ユーザー定義型を INSTANTIABLE から NOT INSTANTIABLE へ変更できます。
- ユーザー定義型の NOT INSTANTIABLE から INSTANTIABLE への変更は任意の時点で実行できます。このように変更した場合、表には影響しません。

- ターゲットの型にサブタイプがない場合にかぎり、ユーザー定義型を NOT FINAL から FINAL へ変更できます。
- ユーザー定義型を FINAL から NOT FINAL、またはその逆に変更する場合は、CASCADE を使用して依存列内および依存表内のデータを即時変換する必要があります。CASCADE オプション NOT INCLUDING TABLE DATA を使用して、データを遅延させることはできません。

  型を NOT FINAL から FINAL に変換する場合は、CASCADE INCLUDING TABLE DATA を使用する必要があります。型を FINAL から NOT FINAL に変更する場合は、CASCADE INCLUDING TABLE DATA または CASCADE CONVERT TO SUBSTITUTABLE のいずれかを使用できます。

  型を FINAL から NOT FINAL に変更する場合、既存の列および表で変更する型の新しいサブタイプを挿入可能にするかどうかによって、選択する CASCADE オプションがかわります。

  デフォルトでは、型を FINAL から NOT FINAL にすることで、代入可能な新しい表およびその型の列を作成できますが、その型の既存の列（またはオブジェクト表）が自動的に代入可能になることはありません。実際、その逆の動作が行われます。その型の既存の列および表は NOT SUBSTITUTABLE AT ALL LEVELS とマークされます。そのような列の埋込み属性が代入可能な場合は、エラーが生成されます。変更された型の新しいサブタイプは、そのような既存の列および表には挿入できません。

  既存の型の列および表を代入可能にするようユーザー定義型を NOT FINAL に変更する場合（NOT SUBSTITUTABLE とマークされていないことを前提）は、CASCADE オプション CONVERT TO SUBSTITUTABLE を使用します。次に例を示します。

```
ALTER TYPE t NOT FINAL CASCADE CONVERT TO SUBSTITUTABLE;
```

  この CASCADE オプションにより既存の列が SUBSTITUTABLE AT ALL LEVELS とマークされ、列に格納されたインスタンスの型 ID 用に新しい非表示列が追加されます。その後、変更された型のサブタイプ・インスタンスを列に格納することができます。

型の変更の妥当性チェックに失敗した場合

ALTER TYPE 文の INVALIDATE オプションを使用して、依存オブジェクトに加えた型の変更を伝播することなく型の変更ができます。この場合、型の変更によるすべての影響が妥当かどうかを検証する依存型および依存表のチェックは行われません。かわりに、すべての依存スキーマ・オブジェクトは無効とマークされます。型および表も含めて、オブジェクトは、次回の参照時に再検証されます。型の再検証ができない場合、その型は無効のままとなります。また、そのような型を参照する表も、問題が修正されるまではアクセス不可となります。

表が妥当性チェックに失敗するのは、たとえば、次の場合です。型に新しい属性を追加した結果、その表内の一部の列が、最大許容数である 1000 を超えた場合。ある表のパーティション化キーまたはクラスタ化キーとして使用されている属性が型から削除された場合。

型の再検証は、ALTER TYPE COMPILE 文を発行すると強制的に実行できます。無効な表の再検証を強制的に実行するには、ユーザーは、ALTER TABLE UPGRADE 文を発行して、データを最新の型バージョンに変更するかどうかを指定できます。(表の参照時にトリガーされた表の妥当性チェックにおいて、表データは常に最新の型バージョンになるよう更新されます。データの変換を延期するオプションはありません。)

表を最新の型バージョンに変換することができない場合は、その表に対する INSERT 文、UPDATE 文および DELETE 文は使用できません。また、そのデータにはアクセスできなくなります。その表に対して次の DDL は実行できますが、無効な表を参照する他のすべての文は、その表の妥当性が検証されるまで使用できません。

- DROP TABLE
- TRUNCATE TABLE

ある表の %ROWTYPE またはある列の %TYPE、あるいはある表にある属性を使用して定義された変数を含むすべての PL/SQL プログラムは、最新の型バージョンに基づいてコンパイルされます。その表の再検証が失敗すると、その表を参照する任意のプログラム・ユニットのコンパイルも失敗します。

型進化に使用する ALTER TYPE オプション

次に、型の属性またはメソッド定義を変更するために使用する ALTER TYPE 文のオプションについて説明します。

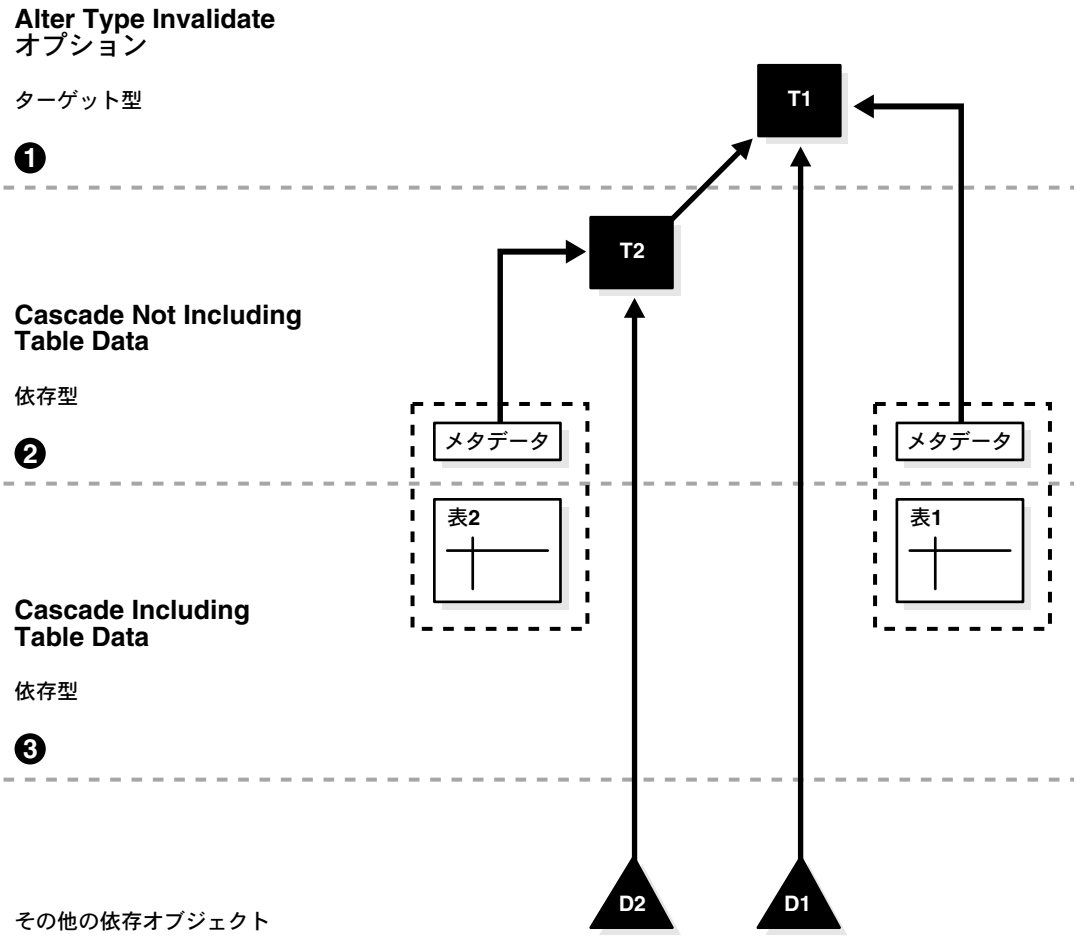
オプション	用途
ADD *method_spec*	指定されたメソッドを型に追加します。
DROP *method_spec*	指定された仕様のメソッドをターゲットの型から削除します。
ADD ATTRIBUTE	指定された属性をターゲットの型に追加します。
DROP ATTRIBUTE	指定された属性をターゲットの型から削除します。
MODIFY ATTRIBUTE	指定された属性の型を変更して、その長さ、精度またはスケールを増大させます。
INVALIDATE	すべての依存オブジェクトを無効とします。このオプションを使用すると、型および表のすべてのチェックが迂回されて、時間が節約できます。 このオプションは、依存型および依存表の妥当性チェックにおいて問題が発生しないことが確実な場合に使用してください。表の妥当性が検証されるまでは表データにはアクセスできません。表の妥当性チェックができない場合は、そのデータはアクセス不可のままになります。

オプション	用途
CASCADE	型の変更を依存型および依存表に伝播します。この文は、FORCE オプションが指定されていないかぎり、依存型または依存表にエラーが検出されると異常終了します。 他のオプションなしに CASCADE が指定されていると、CASCADE に対して INCLUDING TABLE DATA オプションが暗黙的に指定され、すべての表データが最新の型バージョンに変換されます。
INCLUDING TABLE DATA	すべてのユーザー定義列に格納されているデータを、列の型の最新バージョンに変換します。
NOT INCLUDING TABLE DATA	列データを変換しないで、現行の型バージョンに対応付けられたまま維持します。属性が表が参照する型から削除されても、削除された属性の対応する列は表から削除されません。その列のメタデータが未使用とマークされるのみです。削除された属性が列外（たとえば、VARRAY 属性、LOB 属性、またはネストした表属性）に格納されている場合は、その列外のデータは削除されません（使用されない列は、後で、ALTER TABLE DROP UNUSED COLUMNS 文を使用して削除できます）。 このオプションは、数多くの大きな表があり、1 つのトランザクションでそのすべてを変換すると、ロールバック・セグメントが不足する可能性がある場合に使用します。このオプションを使用して、後で、各依存表のデータを、別個のトランザクションで変換できます（ALTER TABLE UPGRADE INCLUDING DATA 文を使用します）。 このオプションを指定すると、表のデータは、以前の型バージョンの形式のまま維持されるので、表の更新が高速化されます。ただし、この表からデータを選択するには、その列に格納されているイメージを最新の型バージョンに変換する必要があります。この操作は、後続の SELECT 文が実行される際のパフォーマンスに影響を及ぼす可能性があります。
FORCE	システムに、依存表および索引からのエラーを無視させます。エラーについては、後で問合せができるように、指定された例外表にログが作成されます。一部の表でエラーが発生した場合、依存表にアクセスできなくなる可能性があるので、このオプションは慎重に使用してください。

オプション	用途
CONVERT TO SUBSTITUTABLE	型を FINAL から NOT FINAL に変更する場合に使用するオプションで、すべてのユーザー定義列に格納されているデータを、最新バージョンの列の型に変換し、作成された型の新しいサブタイプが SUBSTITUTABLE AT ALL LEVELS 型の既存の列およびオブジェクト表に格納できるように、これらの列および表をマークします。 このオプションを指定せずに、型を NOT FINAL に変更した場合、その型の既存の列および表は NOT SUBSTITUTABLE AT ALL LEVELS とマークされ、その型の新しいサブタイプをこれらの列および表に格納できません。このようなサブタイプは、型を変更した後に作成された列および表にのみ格納できます。

図 6-1 に、ALTER TYPE INVALIDATE のオプションとその影響を示します。この図で、T1 は型、T2 は依存型です。図の下の注意も参照してください。

図 6-1 ALTER TYPE オプション



図に関する注意 :

1. **Invalidate**: ①の線の下のすべてのオブジェクトは無効とマークされます。
2. **Cascade Not Including Table Data**: ②の線の下のすべてのオブジェクトは無効とマークされます。すべての依存表のメタデータを最新の型バージョンにアップグレードしますが、表データは変換されません。
3. **Cascade Including Table Data**: ③の線の下のすべてのオブジェクトは無効とマークされます。表データも含め、すべての依存表が最新の型バージョンにアップグレードされます。

型進化に使用する ALTER TABLE オプション

ALTER TABLE を使用して、参照される型の最新バージョンに表データを変換できます。たとえば、次の文は、表 benefits にあるデータを最新の型バージョンに変換します。

```
ALTER TABLE benefits UPGRADE INCLUDING DATA;
```

ALTER TABLE 文には、表データを最新の型バージョンに変換するために使用する次のオプションが含まれています。

オプション	用途
UPGRADE	ターゲット表のメタデータを変換して、参照される型の最新バージョンと一致させます。ターゲット表がすでに有効になっている場合は、表のメタデータは変更されません。 INCLUDING DATA を指定すると、表内のデータは最新の型バージョンの形式に変換されます。デフォルトは、INCLUDING DATA です。以前の型バージョンに基づくデータが格納されている表はどれか、USER_TAB_COLUMNS ビューを参照して確認できます。
INCLUDING DATA	すべてのユーザー定義列に格納されているデータを、列の型の最新バージョンに変換します。列の型に追加された新しい各属性の場合は、新しい属性がデータに追加され、NULL に初期化されます。参照される型から削除される各属性の場合は、対応する属性データが表内の各行から削除されます。表のデータが格納されているすべての表領域は、読込み / 書込みモードである必要があります。読込み / 書込みモードではない場合、この文の処理は失敗します。

オプション	用途
NOT INCLUDING DATA	列データは現状のまま維持され、型バージョンは更新されません。属性がターゲットの表が参照する型から削除されても、削除された属性の対応する列は表から削除されません。その列のメタデータが未使用とマークされるのみです。削除された属性が列外（たとえば、VARRAY 属性、LOB 属性、またはネストした表属性）に格納されている場合は、その列外のデータは削除されません。これらの属性を削除するには、INCLUDING DATA オプションを指定してこの文を再送信できます。 このオプションを指定すると、表のデータは、以前の型バージョンの形式のまま維持されるので、表の更新が高速化されます。この表から選択されたデータは、最新の型バージョンに変換する必要があります。したがって、後続の SELECT 文の実行中、パフォーマンスが影響を受ける可能性があります。 このオプションは、表全体を一度に変換するために必要なロールバック・セグメントが不足している場合に便利です。この場合、まず、表のメタデータを、データを変換することなくアップグレードして、それから、UPDATE 文を発行して、各ユーザー定義列を自らに設定できます。UPDATA 文を指定すると、ターゲットの列内のデータは最新の型バージョンの形式に変換されます。 このオプションは、表のメタデータの更新のみを必要とするため、文を正常に実行するためにすべての表領域の読込み / 書込みモードでのオンライン化が必要なわけではありません。
COLUMN_STORAGE_CLAUSE	表に追加する新しい VARRAY 属性、ネストした表属性、または LOB 属性に対して、記憶域を指定します。

ユーザー定義コンストラクタ

コンストラクタ・ファンクションは、ユーザー定義型のインスタンスの作成に使用します。

システムは、属性を持つすべてのオブジェクト型について、属性値コンストラクタというコンストラクタ・ファンクションを暗黙的に定義します。属性値コンストラクタは、オブジェクトの属性の初期化に使用できます。不透明型は属性を持たないため、属性値コンストラクタを持ちません。ただし、不透明型はコンストラクタ・ファンクションを持つことができ、XMLTYPE など多数のシステム定義の不透明型が、コンストラクタ・ファンクションを持っています。

> **参照：** SYS.ANYTYPE、SYS.ANYDATA、SYS.ANYDATASET の詳細は、6-35 ページの「一時型および汎用型」を参照してください。

属性値コンストラクタはすでに存在するため便利ですが、型進化については、属性値コンストラクタよりユーザー定義コンストラクタの方が優れています。

> **参照：** コンストラクタの詳細は、2-19 ページの「コンストラクタ・メソッド」を参照してください。

属性値コンストラクタ

属性値コンストラクタの場合は、型の各属性値をコンストラクタに渡し、新しいオブジェクト・インスタンスの属性にこれらの値を設定する必要があります。

次に例を示します。

```
CREATE TYPE shape AS OBJECT (
    name VARCHAR2(30),
    area NUMBER
);
-- Attribute value constructor: Sets instance attributes
-- to the specified values

INSERT INTO building_blocks
  VALUES (
    NEW shape('my_shape', 4)
  );
```

NEW は、コンストラクタへのコールの前にオプションで追加するキーワードです。このキーワードを追加することをお薦めします。

コンストラクタと型進化

システムが提供するコンストラクタ・ファンクションを使用することにより、型に対して独自のコンストラクタを定義する必要がなくなります。ただし、インスタンスを作成し初期化するために属性値コンストラクタを使用する場合は、型に宣言されたすべての属性に対して値を指定する必要があります。指定しない場合、コンストラクタへのコールのコンパイルが失敗します。

たとえば、属性を追加して後で型を進化させる場合は、属性値コンストラクタのこの要件によって問題が発生することがあります。型の属性を変更した場合は、その型の属性値コンストラクタも変更されます。属性を追加した場合、更新された属性値コンストラクタは、新しい属性および古い属性に対する値が指定されるものとみなします。その結果、古い属性に対してのみ値が指定された場合は、既存のコード内の属性値コンストラクタへのすべてのコールのコンパイルが失敗します。

参照： 6-7 ページの「型進化」を参照してください。

ユーザー定義コンストラクタの利点

ユーザー定義コンストラクタは、型のすべての属性に対して値を明示的に設定する必要がないため、属性値コンストラクタで発生する問題が回避されます。ユーザー定義コンストラクタは、任意の型の任意の数の引数を取ることができ、これらの引数は型の属性に直接マップする必要がありません。コンストラクタの定義では、属性を任意の適切な値に初期化できます。値が指定されない属性は、システムによって NULL に初期化されます。

たとえば、属性を追加して型を進化させる場合、その型に対するユーザー定義コンストラクタへのコールは変更する必要がありません。通常のメソッドのようなユーザー定義コンストラクタは、型が進化しても自動的に変更されません。そのため、ユーザー定義コンストラクタのコール・シグネチャは同じ状態のまま維持されます。ただし、新しい属性を NULL に初期化しない場合は、コンストラクタの定義の変更が必要になることがあります。

ユーザー定義コンストラクタの定義および実装

ユーザー定義コンストラクタは、通常のメソッド・ファンクションと同様に型本体に定義します。ユーザー定義コンストラクタの宣言および定義は、CONSTRUCTOR FUNCTION 句を使用します。また、RETURN SELF AS RESULT 句も使用する必要があります。

型のコンストラクタは、型と同じ名前を持つ必要があります。次のコードは、shape 型に 2 つのコンストラクタ・ファンクションを定義します。例に示すとおり、異なるシグネチャを持つ複数のバージョンを定義することにより、ユーザー定義コンストラクタをオーバーロードすることができます。

```
CREATE OR REPLACE TYPE shape AS OBJECT
(
    name VARCHAR2(30),
    area NUMBER,
    CONSTRUCTOR FUNCTION shape(name VARCHAR2) RETURN SELF AS RESULT,
    CONSTRUCTOR FUNCTION shape(name VARCHAR2, area NUMBER) RETURN
      SELF AS RESULT
) NOT FINAL;

CREATE OR REPLACE TYPE BODY shape IS
    CONSTRUCTOR FUNCTION shape(name VARCHAR2) RETURN SELF AS RESULT IS
    BEGIN
        SELF.name := name;
        SELF.area := 0;
        return;
    END;
    CONSTRUCTOR FUNCTION shape(name VARCHAR2, area NUMBER) RETURN shape
    SELF AS RESULT IS
    BEGIN
        SELF.name := name;
        SELF.area := area;
        return;
    END;
END;
```

ユーザー定義コンストラクタには、暗黙的な第 1 パラメータ SELF があります。ユーザー定義コンストラクタの宣言では、このパラメータの指定はオプションです。指定する場合、このパラメータのモードは IN OUT として宣言する必要があります。

必須句 RETURN SELF AS RESULT によって、戻されるインスタンスの最も狭い意味での型は、必ず SELF 引数の最も狭い意味での型と同じになります。shape コンストラクタへのコールに対する SELF 引数の最も狭い意味での型が shape の場合、この句によって shape コンストラクタからは shape のサブタイプのインスタンスではなく shape のインスタンスが必ず戻されます。

コンストラクタ・ファンクションがコールされると、システムは SELF 引数の属性を NULL に初期化します。前述の例に示すとおり、ファンクション本体で初期化される後続の属性名は、コンストラクタ・ファンクションの引数の名前と区別するために、SELF で修飾することができます（これらの名前が同一の場合）。引数の名前が異なる場合、このような修飾は不要です。次に例を示します。

```
SELF.name := name;
```

または

```
name := p1;
```

例に示すとおり、ファンクション本体には、明示的な return; を含める必要があります。ただし、**return** キーワードの後に、return 式を続けることはできません。システムは、新しく作成された SELF インスタンスを自動的に戻します。

ユーザー定義コンストラクタは、PL/SQL、C または Java で実装できます。

コンストラクタのオーバーロードおよびオーバーライド

他の型のメソッドと同様、ユーザー定義コンストラクタもオーバーロードできます。

ユーザー定義コンストラクタは継承されないため、オーバーライドできません。ただし、ユーザー定義コンストラクタは隠蔽しているため、ユーザー定義コンストラクタのシグネチャが属性値コンストラクタのシグネチャと完全に一致する場合は、属性値コンストラクタに置き換わります。シグネチャが一致するには、ユーザー定義コンストラクタのパラメータ（暗黙的な SELF パラメータの後）の名前と型が、その型の属性の名前と型と同じである必要があります。ユーザー定義コンストラクタの各パラメータ（暗黙的な SELF パラメータの後）のモードは、必ず IN にしてください。

同じ名前とシグネチャを持つユーザー定義コンストラクタによって属性値コンストラクタが隠蔽されていない場合は、継続して属性値コンストラクタをコールできます。

たとえば、属性の追加により型が進化する場合は、型の属性値コンストラクタのシグネチャもそれに対応して変更することに注意してください。これによって、以前に隠蔽されていた属性値コンストラクタが再度使用可能になります。

ユーザー定義コンストラクタのコール

ユーザー定義コンストラクタは、他のすべてのファンクションと同様にコールします。ユーザー定義コンストラクタは、通常のファンクションが使用できるすべての場所で使用できます。

SELF 引数は、暗黙的に渡され、明示的に渡されません。つまり、次のように使用することはできません。

```
NEW constructor(instance, argument_list)
```

CREATE 文または ALTER TABLE 文の DEFAULT 句には、ユーザー定義コンストラクタは指定できませんが、属性値コンストラクタは指定できます。属性値コンストラクタの引数には、PL/SQL ファンクションまたは他の列（疑似列 LEVEL、PRIOR、ROWNUM など）の参照、あるいは完全に指定されていない日付定数の参照を含めることはできません。CHECK 制約式についても同様です。表の作成中または変更中に、属性値コンストラクタは CHECK 制約式の一部として使用できますが、ユーザー定義コンストラクタは使用できません。

SQL では、引数を持たないコンストラクタ・コールについても、カッコを付ける必要があります。次に例を示します。

```
SELECT NEW type_name() FROM dual;
```

PL/SQL では、引数を持たないコンストラクタを起動する場合、カッコはオプションです。ただし、カッコを付けた方が、コンストラクタ・コールがファンクション・コールであることがより明確になります。

次の PL/SQL 例では、新しい shape を作成するためのコンストラクタ・コールのカッコが省略されています。

```
shape s := NEW my_schema.shape;
```

NEW キーワードおよびスキーマ名はオプションで使用してください。

SQL の例

```
CREATE OR REPLACE TYPE rectangle UNDER shape
(
    length NUMBER,
    breadth NUMBER,
    CONSTRUCTOR FUNCTION rectangle(
        name VARCHAR2, length NUMBER, breadth NUMBER
    ) RETURN SELF as RESULT
);

CREATE OR REPLACE TYPE BODY rectangle IS
    CONSTRUCTOR FUNCTION rectangle(
        name VARCHAR2, length NUMBER, breadth NUMBER
    ) RETURN  SELF AS RESULT IS
    BEGIN
        SELF.name := name;
        SELF.area := length*breadth;
        SELF.length := length;
        SELF.breadth := breadth;
        RETURN ;
```

```
    END;
END;

INSERT INTO rectangle_table
  SELECT NEW rectangle(s.name, s.length, s.breadth) FROM square_table s;

UPDATE rectangle_table SET rec = NEW rectangle('Quad', 12, 5) WHERE rec IS NULL;
```

PL/SQL の例

```
s shape := NEW shape('void');
```

SQLJ オブジェクト型のコンストラクタ

SQLJ オブジェクト型は、Java クラスにマップされる SQL オブジェクト型です。SQLJ オブジェクト型は属性値コンストラクタを持ちます。このオブジェクト型は、参照された Java クラスでコンストラクタにマップされるユーザー定義コンストラクタを持つこともできます。

次に例を示します。

```
CREATE TYPE address AS OBJECT
  EXTERNAL NAME 'university.address'
  LANGUAGE JAVA USING SQLData
(
  street VARCHAR2(100) EXTERNAL NAME 'street',
  city VARCHAR2(50) EXTERNAL NAME 'city',
  state VARCHAR2(50) EXTERNAL NAME 'state',
  zip_code number EXTERNAL NAME 'zipCode',
  CONSTRUCTOR FUNCTION address (full_address VARCHAR)
    EXTERNAL NAME 'address (java.lang.String)',
);
```

シリアル化表現の SQLJ 型は、ユーザー定義コンストラクタのみ持つことができます。SQLJ 型のオブジェクトの内部表現は、SQL にとって不透明であるため、SQLJ 型に属性値コンストラクタは使用できません。

オブジェクトに対する OCI のヒントおよび技法

次の項では、オブジェクトを使用する OCI プログラムで実行される一般的な操作を示し、OCI を効果的に使用するためのヒントおよび技法を紹介します。

オブジェクト・モードでの OCI プログラムの初期化

オブジェクト操作を使用可能にするには、OCI プログラムをオブジェクト・モードで初期化する必要があります。次の OCI コードで、プログラムをオブジェクト・モードで初期化します。

```
err = OCIInitialize(OCI_OBJECT, 0, 0, 0, 0);
```

プログラムがオブジェクト・モードで初期化される場合、オブジェクト・キャッシュが初期化されます。この時点では、キャッシュのメモリーは割り当てられません。キャッシュのメモリーは、必要に応じて割り当てられます。

新規のオブジェクトの作成

`OCIObjectNew()` 関数によって、一時オブジェクトまたは永続オブジェクトが作成されます。一時オブジェクトの存続期間は、それが作成されたセッションの存続期間です。永続オブジェクトとは、データベースのオブジェクト表に格納されるオブジェクトです。`OCIObjectNew()` 関数は、キャッシュ内に作成されたオブジェクトに対するポインタを戻します。そのため、アプリケーションは属性値を直接設定して、新規オブジェクトを初期化する必要があります。オブジェクトは、この時点ではまだデータベースに作成されていません。キャッシュからフラッシュされるときに作成されてデータベースに格納されます。

`OCIObjectNew()` を使用してキャッシュ内にオブジェクトを作成する場合、すべての属性が NULL に設定されます。属性 NULL インジケータ情報は、パラレル NULL インジケータ構造体に記録されます。アプリケーションが属性値を設定したにもかかわらず、パラレル NULL 構造体への NULL インジケータ情報の設定に失敗すると、オブジェクトのフラッシュ時に、オブジェクト属性がデータベース内で NULL に設定されます。

オブジェクト作成時にすべての属性を NOT NULL に設定する場合は、`OCIAttrSet()` 関数を使用して環境ハンドルの `OCI_OBJECT_NEW_NOTNULL` 属性を使用できます。この属性が設定されると、NULL 属性を持たないオブジェクトが作成されます。つまり、すべての属性が、Oracle によって提供されるデフォルト値に設定され、パラレル NULL インジケータ構造体におけるその NULL ステータス情報は、NOT NULL に設定されます。この属性を使用すると、インジケータ構造体を変更する追加手順が不要になります。Oracle によって提供されるデフォルト値は変更できません。かわりに、オブジェクトの作成直後に、オブジェクトに独自のデフォルト値を代入できます。

`OCIObjectNew()` を使用して永続オブジェクトが作成されると、コール側は、新しく作成されたオブジェクトを挿入するデータベース表を識別する必要があります。コール側は、表オブジェクトを使用して表を識別します。スキーマ名および表名が指定されると、`OCIObjectPinTable()` 関数は、表オブジェクトに対するポインタを戻します。`OCIObjectPinTable()` への各コールは、表オブジェクト情報をフェッチするためのサー

バーへのコールになります。必要な表オブジェクトがすでにキャッシュに確保されている場合でも、サーバーへのコールが発生します。アプリケーションが、同一データベース表に挿入するオブジェクトを複数作成している場合は、表オブジェクトを一度確保し、その表オブジェクトへのポインタを将来の使用に備えて保存しておくことをお薦めします。これによって、アプリケーションのパフォーマンスが向上します。

オブジェクトの更新

オブジェクトを更新する前に、そのオブジェクトをキャッシュに確保する必要があります。オブジェクトが確保されると、アプリケーションは必要な属性を直接更新できます。オブジェクトが更新されたことを示すには、`OCIObjectMarkUpdate()` 関数へのコールを行う必要があります。更新済としてマークされたオブジェクトは、使用済リストに置かれ、キャッシュのフラッシュ時またはトランザクションがコミットされるときに、サーバーにフラッシュされます。

オブジェクトの削除

`OCIObjectMarkDelete()` 関数または `OCIObjectMarkDeleteByRef()` 関数をコールすることによって、オブジェクトを削除できます。

オブジェクト・キャッシュ・サイズの制御

次の 2 つの OCI 環境ハンドル属性を使用して、オブジェクト・キャッシュのサイズを制御できます。

- `OCI_ATTR_CACHE_MAX_SIZE`: 最大キャッシュ・サイズを制御します。
- `OCI_ATTR_CACHE_OPT_SIZE`: 最適なキャッシュ・サイズを制御します。

`OCIAttrGet()` または `OCIAttrSet()` 関数を使用して、これらの OCI 属性を取得または設定できます。メモリーがキャッシュに割り当てられるたびに、最大キャッシュ・サイズに達したかどうかを判断するチェックが行われます。最大キャッシュ・サイズに達した場合、キャッシュは、最も前に使用されたオブジェクトを、確保カウントを 0（ゼロ）で自動的に解放（エージ・アウト）します。キャッシュでのメモリー使用量が最適なサイズになるまで、または解放できるオブジェクトがなくなるまで、キャッシュはこのようなオブジェクトの解放を続けます。オブジェクト・キャッシュには、最大キャッシュ・サイズの制限がありません。メモリー割当て要求のサービスによって、キャッシュ・サイズが、指定された最大キャッシュ・サイズを超える可能性があります。前述の 2 つのパラメータを利用することで、キャッシュからのオブジェクトのエージングの頻度をアプリケーションから制御できます。

クライアント・キャッシュへのオブジェクトの取出し（確保）

確保とは、サーバーからクライアント・キャッシュにオブジェクトを取り出し、メモリーにオブジェクトを置き、アプリケーションが操作できるようにオブジェクトにポインタを指定

し、オブジェクトに使用中のマークを付ける処理のことです。OCIObjectPin() 関数は、指定された REF を参照解除して、対応するオブジェクトをキャッシュに確保します。確保されたオブジェクトへのポインタは、コール側に戻されます。また、このポインタは、オブジェクトがキャッシュに確保されているかぎり有効です。オブジェクトが確保解除された後は、そのオブジェクトはエージ・アウトとなって、オブジェクト・キャッシュ内に存在しなくなる場合があるため、このポインタは使用しないでください。

次に、OCIObjectPin() コールおよび OCIObjectUnpin() コールの例を示します。

```
status = OCIObjectPin(envh, errh, empRef,(OCIComplexObject*)0,
                      OCI_PIN_RECENT, OCI_DURATION_TRANSACTION,
                      OCI_LOCK_NONE, (dvoid**)&emp);
/* manipulate emp object */
status = OCIObjectUnpin(envh, errh, emp);
```

OCIObjectPin() の引数である empRef パラメータは、必要な従業員オブジェクトに対する REF を指定します。キャッシュ内の従業員オブジェクトへのポインタは、emp パラメータで戻されます。

OCIObjectPinArray() 関数を使用して、オブジェクトの配列を 1 回のコールで確保できます。この関数は、REF の配列を参照解除して、対応するオブジェクトをキャッシュに確保します。オブジェクトがキャッシュ上にキャッシュされていない場合には、1 回のネットワーク・ラウンドトリップでサーバーから取り出されます。したがって、OCIObjectPinArray() をコールしてオブジェクトの配列を確保すると、アプリケーションのパフォーマンスが向上します。また、確保するオブジェクトの配列の型は異なってもかまいません。

取り出すオブジェクトのバージョンの指定

オブジェクトを確保するときに、PIN オプション引数を使用して、オブジェクトの現在のバージョン、最新バージョンまたは任意のバージョンのどれが必要かを指定できます。有効なオプションの詳細は、次のとおりです。

- PIN オプション OCI_PIN_RECENT。現行のトランザクションでキャッシュにロードされているオブジェクトを戻すように、オブジェクト・キャッシュに指示します。つまり、オブジェクトが現行のトランザクション以前にロードされていた場合、オブジェクト・キャッシュは、このオブジェクトをデータベースからの最新バージョンでリフレッシュする必要があります。同一トランザクション内でオブジェクトの確保に成功すると、キャッシュされているコピーが戻されるので、データベース・アクセスは発生しません。通常は、この PIN オプションを使用してください。
- PIN オプション OCI_PIN_LATEST。常にオブジェクトの最新のコピーを取得するように、オブジェクト・キャッシュに指示します。オブジェクトがすでにキャッシュ内にあり、ロックされていない場合、オブジェクト・コピーは、データベースからの最新のコピーでリフレッシュされます。一方、キャッシュ内のオブジェクトがロックされている場合、これが最新のコピーと想定され、キャッシュされているコピーが戻されます。オブジェクトの最新のコピーを表示する必要があるアプリケーション（株式相場、当座預

金残高などを表示するアプリケーションなど）には、このオプションを使用する必要があります。

- PIN オプション OCI_PIN_ANY。最も効率のよい方法でオブジェクトをフェッチするように、オブジェクト・キャッシュに指示します。戻されるオブジェクトのバージョンは問題ではありません。このオプションは、製品情報、部品情報など、頻繁には変更されないオブジェクトの場合に適しています。また、読込み専用のオブジェクトにも適しています。

オブジェクトの確保時間の指定

オブジェクトを確保するとき、オブジェクトがキャッシュ内に確保される有効期限を指定できます。有効期限がすぎると、オブジェクトは自動的にキャッシュから確保解除されます。オブジェクトの確保有効期限が終了した後は、アプリケーションでオブジェクト・ポインタを使用しないでください。オブジェクトは、OCIObjectUnpin() 関数を明示的にコールすることによって、有効期限切れになる前に確保解除できます。Oracle では、次の 2 つの事前定義済の確保有効期限がサポートされます。

- セッション確保有効期限（OCI_DURATION_SESSION）存続期間 : データベース接続の有効期限です。トランザクションをまたがってキャッシュ内に常に必要なオブジェクトは、セッション有効期限で確保する必要があります。
- トランザクション確保有効期限（OCI_DURATION_TRANS）存続期間 : データベース・トランザクションの有効期限です。つまり、トランザクションがロールバックまたはコミットされた時点で有効期限が終了します。

サーバー上でオブジェクトをロックするかどうかの指定

オブジェクトを確保するとき、コール側は、ロック・オプションを介してオブジェクトをロックするかどうかを指定できます。オブジェクトがロックされると、サーバー側のロックが行われ、他のユーザーがオブジェクトを変更できなくなります。トランザクションがコミットまたはロールバックされた時点で、ロックが解除されます。使用可能なロック・オプションは次のとおりです。

- OCI_LOCK_NONE ロック・オプション : ロックしないでオブジェクトを確保するように、キャッシュに指示します。
- OCI_LOCK_X ロック・オプション : ロックを取得した後にのみオブジェクトを確保するように、キャッシュに指示します。オブジェクトが、現在、他のユーザーによってロックされている場合、このオプションを持つ確保コールは、ロックを取得できるまで待機した後、コール側に戻ります。OCI_LOCK_X ロック・オプションを使用することは、SELECT FOR UPDATE 文を実行することと同等です。
- OCI_LOCK_X_NOWAIT ロック・オプション : ロックを取得した後にのみオブジェクトを確保するように、キャッシュに指示します。OCI_LOCK_X オプションと異なり、OCI_LOCK_X_NOWAIT オプションを持つ確保コールは、オブジェクトが現在別のユーザーによってロックされている場合は待機しません。OCI_LOCK_X_NOWAIT ロック・

オプションを使用することは、SELECT FOR UPDATE WITH NOWAIT 文を実行することと同等です。

ロック技法の選択方法

オブジェクトが更新される頻度に基づいて、前述の項で説明したどのロック・オプションを使用するかを選択できます。

オブジェクトが頻繁に更新される場合は、即時ロック構造を使用できます。この構造は、更新アクセスの競合が頻繁に発生することを想定しています。オブジェクトは、キャッシュにあるオブジェクトが変更される前にロックされ、ロックを所有しているトランザクションがコミットまたはロールバックを実行するまで、他のどのユーザーもそのオブジェクトを変更できないようにします。適切なロック・オプションを選択することによって、確保の時点でオブジェクトをロックできます。確保の時点でロックされていなかったオブジェクトも、OCIObjectLock() 関数によってロックできます。OCIObjectLockNoWait() ロック関数は、別のユーザーがオブジェクトのロックを保持している場合は、ロックの取得を待機しません。

オブジェクトがあまり頻繁に更新されない場合は、コミット時ロック構造を使用できます。この構造は、更新アクセスの競合がほとんどないことを想定しています。オブジェクトは、ロックを取得せずに、キャッシュ内でフェッチおよび変更されます。オブジェクトがサーバーにフラッシュされる場合にのみ、ロックが取得されます。コミット時ロックでは、即時ロックより高度な同時アクセスが可能です。コミット時ロックを効果的に使用するために、Oracle のオブジェクト・キャッシュは、オブジェクトがキャッシュにフェッチされてから、他のユーザーによって変更されたかどうかを検出します。オブジェクト変更検出モードにすると、オブジェクトがキャッシュにフェッチされてから他のユーザーによって変更されていない場合にのみ、そのオブジェクトの変更が永続化されます。このモードは、OCIAttrSet() 関数を使用して環境ハンドルの OCI_OBJECT_DETECTCHANGE 属性を設定することによってアクティブになります。

オブジェクト・キャッシュからのオブジェクトのフラッシュ

オブジェクト・キャッシュ内のオブジェクトに行われた変更は、オブジェクト・キャッシュがフラッシュされるまで、データベースに送信されません。OCICacheFlush() 関数は、クライアントとサーバー間の 1 回のネットワーク・ラウンドトリップで、すべての変更をフラッシュします。変更には、適切なオブジェクト表への新しいオブジェクトの挿入、オブジェクト表でのオブジェクトの更新およびオブジェクト表からのオブジェクトの削除が含まれます。OCITransCommit() 関数をコールすることによってアプリケーションがトランザクションをコミットする場合、オブジェクト・キャッシュは、トランザクションをコミットする前に、キャッシュのフラッシュを自動的に実行します。

関連オブジェクトのプリフェッチ（複合オブジェクト検索）

複合オブジェクト検索（COR）によって、複雑に関連付けられた複数のオブジェクトを操作するアプリケーションのパフォーマンスを大幅に向上します。COR によって、アプリケー

ションは一連の関連オブジェクトを 1 回のネットワーク・ラウンドトリップでプリフェッチできるため、パフォーマンスが向上します。OCIObjectPin() または OCIObjectPinArray() を使用してルート・オブジェクトを確保する場合、ルートとともに関連オブジェクトをプリフェッチするように指定できます。プリフェッチ・オブジェクトはキャッシュには確保されません。かわりに、LRU リストに入れられます。その結果、これらのオブジェクトへの後続の確保コールはキャッシュ・ヒットするため、サーバーへのラウンドトリップが回避されます。

アプリケーションは、次の情報を指定することによって、一連の関連オブジェクトがプリフェッチされるように指定します。

- ルート・オブジェクトへの REF
- プリフェッチされるオブジェクトの内容および境界を指定するための、オブジェクト型と深さ情報の 1 つ以上の組。タイプ情報は、参照解除する必要がある REF 属性およびプリフェッチする必要がある結果オブジェクトを示します。深さは、プリフェッチされるオブジェクトの境界を定義します。深さレベルは、ルート・オブジェクトから関連オブジェクトに最短距離で到達するために横断する必要がある参照数です。

たとえば、次のプロパティを持つ発注書システムを考えてみます。

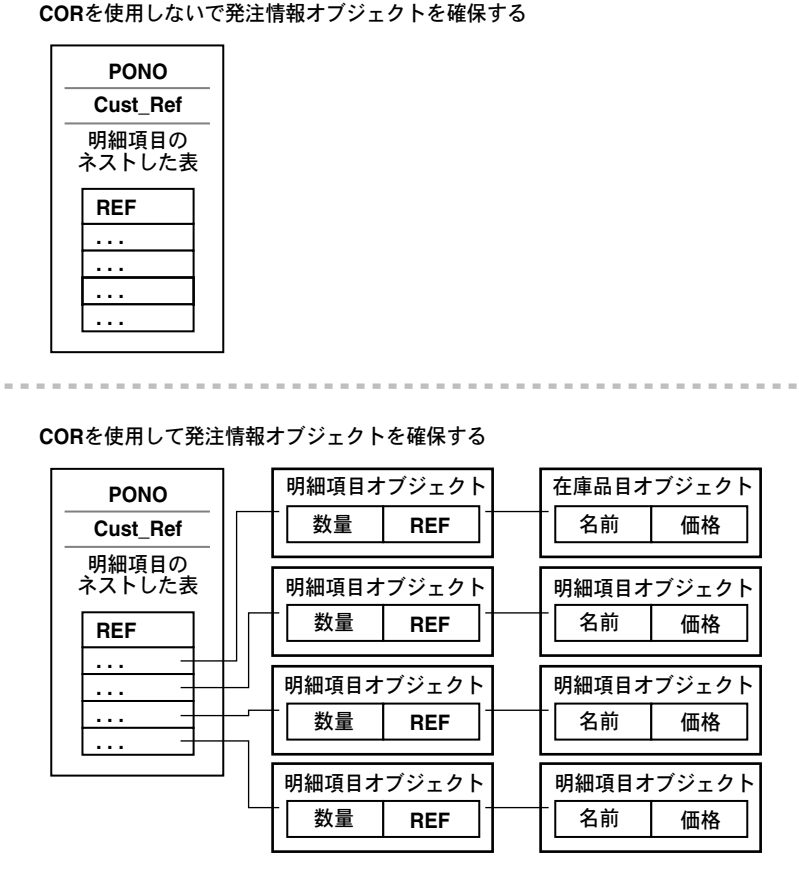
- 各発注情報オブジェクトには、発注書番号、顧客オブジェクトへの REF および明細項目オブジェクトを指す REF のコレクションが含まれています。
- 各顧客オブジェクトには、顧客の名前や住所など、顧客についての情報が含まれています。
- 各明細項目オブジェクトには、在庫品目への参照および発注の数量が含まれています。
- 各在庫品目オブジェクトには、在庫品目の名前、価格およびその品目についてのその他の情報が含まれています。

特定の発注書の合計コストを計算する場合を考えてみます。効率を最大にするには、計算に必要なオブジェクトのみをサーバーからクライアント・キャッシュにフェッチし、サーバーへのコール数をできるだけ少なくして、これらのオブジェクトをフェッチします。

COR を使用しない場合、アプリケーションは、必要なオブジェクトをすべて取り出すためにサーバー・コールを数回行う必要があります。ただし、COR を使用すると、取り出すオブジェクトを指定して、その他の不要なオブジェクトを除外できます。発注書の合計コストを計算するには、発注情報オブジェクト、関連明細項目オブジェクトおよび関連在庫品目オブジェクトが必要ですが、顧客オブジェクトは必要ありません。

したがって、図 6-2 に示すように、COR を使用すると、最も効率のよい方法で計算に必要な情報を取り出すことができます。COR を使用しないで発注情報オブジェクトを確保する場合、そのオブジェクトのみが取り出されます。COR を使用して発注情報オブジェクトを確保すると、発注情報オブジェクト、関連明細項目オブジェクトおよび在庫品目オブジェクトが取り出されます。ただし、関連顧客オブジェクトは計算には不要なため、取り出されません。

図 6-2 COR を使用した場合と COR を使用しない場合のオブジェクトの取出しの違い



OCI および Oracle オブジェクトのデモンストレーション

Oracle オブジェクトで OCI を使用する方法のデモは、`$ORACLE_HOME/rdbms/demo` にある `cdemocor1.c` ファイルを参照してください。

オブジェクト・ビューが提供するオブジェクトでの OCI オブジェクト・キャッシュの使用

オブジェクト表の場合と同様の方法で、オブジェクト・ビューから合成されたオブジェクトを OCI オブジェクト・キャッシュ上で確保およびナビゲートできます。また、そのキャッシュから、その新しいビュー・オブジェクトを作成、更新、削除およびフラッシュすること

もできます。フラッシュによって、ビューに適切な DML（新しく作成されたオブジェクトの挿入、属性の変更に対する更新など）が実行されます。これによって、ビュー上のすべての INSTEAD OF トリガーが起動され、オブジェクトは永続的に格納されます。

オブジェクト・キャッシュに新しく作成されたインスタンスの参照の取得に関して、2 つの方法にはわずかな違いがあります。

主キー・ベースの参照を持つオブジェクト・ビューの場合は、OCIObjectGetObjectRef コールがオブジェクトに対してコールされ、オブジェクト参照が取得される前に、オブジェクトに対する識別子を形成する属性が初期化される必要があります。たとえば、発注情報オブジェクトの OCI オブジェクト・キャッシュに新しいオブジェクトを作成するには、次の手順を行う必要があります。

```
.. /* Initialize all the settings including creating a connection, getting a
      environment handle and so forth. We do not check for error conditions to make
      the example eaiser to read. */
OCIType *purchaseOrder_tdo = (OCIType *) 0; /* This is the type object for the
                                               purchase order */
dvoid * purchaseOrder_viewobj = (dvoid *) 0;   /* This is the view object */

/* The purchaseOrder struct is a structure that is defined to have the same
attributes as that of  PurchaseOrder_objtyp type. This can be created by the user or
generated automatically using  the OTT generator. */
purchaseOrder_struct *purchaseOrder_obj;

/* This is the null structure corresponding to the purchase order object's
attributes */
purchaseOrder_nullstruct *purchaseOrder_nullobj;

/* This is the variable containing the purchase order number that we need to create
*/
int PONo = 1003;

/* This is the reference to the purchase order object */
OCIRef *purchaseOrder_ref = (OCIRef *)0;

/* Pin the object type first */
OCITypeByName( envhp, errhp, svchp,
                              (CONST text *) "",  (ub4) strlen( "") ,
                              (CONST text *) "PURCHASEORDER_OBJTYP" ,
                              (ub4) strlen("PURCHASEORDER_OBJTYP"),
                              (CONST char *) 0, (ub4)0,
                             OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &purchaseOrder_
tdo);

/* Pin the table object - in this case it is the purchase order view */
OCIObjectPinObjectTable(envhp, errhp, svchp, (CONST text *) "",
                              (ub4) strlen( "" ),
```

```
                                    (CONST text *) "PURCHASEORDER_OBJVIEW",
                                    (ub4 ) strlen("PURCHASEORDER_OBJVIEW"),
                                    (CONST OCIRef *) NULL,
                                     OCI_DURATION_SESSION,
                                     &purchaseOrder_viewobj);

/* Now create a new object in the cache. This is a purchase order object */
OCIObjectNew(envhp, errhp, svchp, OCI_TYPECODE_OBJECT, purchaseOrder_tdo,
                     purchaseOrder_viewobj, OCI_DURATION_DEFAULT, FALSE,
                     (dvoid **) *purchaseOrder_obj);

/* Now we can initialize this object, and use it as a regular object. But before
getting the reference to this object we need to initialize the PONo attribute of the
object which makes up its object identifier in the view */

/* Initialize the null identifiers */
OCIObjectGetInd( envhp, errhp, purchaseOrder_obj, purchaseOrder_nullobj);

purchaseOrder_nullobj->purchaseOrder = OCI_IND_NOTNULL;
purchaseOrder_nullobj->PONo = OCI_IND_NOTNULL;

/* This sets the PONo attribute */
OCINumberFromInt( errhp, (CONST dvoid *) &PoNo, sizeof(PoNo), OCI_NUMBER_SIGNED,
                                  &( purchaseOrder_obj->PONo));

/* Create an object reference */
OCIObjectNew( envhp, errhp, svchp, OCI_TYPECODE_REF, (OCIType *) 0,
                           (dvoid *) 0, (dvoid *) 0, OCI_DURATION_DEFAULT, TRUE,
                           (dvoid **) &purchaseOrder_ref);

/* Now get the reference to the newly created object */
OCIObjectGetObjectRef(envhp, errhp, (dvoid *) purchaseOrder_obj, purchaseOrder_ref);

/* This reference may be used in the rest of the program ….. */
…
/* We can flush the changes to the disk and the newly instantiated purchase order
object in the object cache will become permanent. In the case of the purchase order
object, the insert will fire the INSTEAD-OF trigger defined over the purchase order
view to do the actual processing */

OCICacheFlush( envhp, errhp, svchp, (dvoid *) 0, 0, (OCIRef **) 0);
…
```

一時型および汎用型

Oracle には、型の説明、データ・インスタンス、およびオブジェクト型やコレクション型を含むその他の SQL 型のデータ・インスタンスの集合の動的なカプセル化および型の記述の取得を可能にする 3 つの特殊な SQL データ型があります。この 3 つの型は、**匿名**の（名前の付けられていない）コレクション型を含めた匿名型の作成にも使用できます。

この 3 つの SQL 型は、**不透明型**として実装されます。つまり、これらの型の内部構造は、データベースには認識されません。不透明型のデータへの問合せは、目的に合ったファンクション（通常は 3GL ルーチン）を実装することによってのみ実行されます。Oracle は、そのようなファンクションを実装するために OCI と PL/SQL API の両方を提供しています。

この 3 つの汎用 SQL 型は次のとおりです。

型	説明
SYS.ANYTYPE	型記述型。SYS.ANYTYPE には、名前が付いているか付いていないかにかかわらず、オブジェクト型およびコレクション型を含めた、任意の SQL 型の型記述を含めることができます。 ANYTYPE には、永続型の型の記述を含めることができますが、ANYTYPE 自体は**一時**型です。つまり、ANYTYPE 自体の値がデータベースに自動的に格納されることはありません。永続型を作成するには、SQL から CREATE TYPE 文を使用します。
SYS.ANYDATA	**自己記述的**データ・インスタンスの型。SYS.ANYDATA には、与えられた型のインスタンス、データ、およびその型の記述が含まれています。SYS.ANYDATA は、この意味で自己記述的です。ANYDATA は、データベースに永続的に保存できます。
SYS.ANYDATASET	自己記述的なデータ集合の型。SYS.ANYDATASET には、与えられた型の記述、およびその型のデータ・インスタンスの集合が含まれています。ANYDATASET は、データベースに永続的に保存できます。

これらの 3 つの型はいずれも、データベースに対してネイティブな任意の組込み型と併用できます。また、名前が付いているかどうかにかかわらず、オブジェクト型やコレクション型とも併用できます。これらの型は、型の記述、ローン・インスタンスおよび他の型のインスタンスの集合と動的に連携する汎用手段となります。API を使用して、任意の型に対して一時的な ANYTYPE 記述を作成できます。同様に、任意の SQL 型のデータ値を作成または ANYDATA に変換（キャスト）できます。また、ANYDATA を、SQL 型に変換（して戻すことが）できます。また、値の集合と ANYDATASET の場合も同様です。

汎用型は、ストアド・プロシージャを使用した作業を簡単にします。汎用型を使用して、標準型の記述およびデータをカプセル化し、カプセル化された情報を汎用型のパラメータに渡せます。プロシージャ本体では、カプセル化されたデータおよび任意の型の型記述の処理方法を詳細に記述できます。

また、基礎となる様々な型のカプセル化されたデータを、型 ANYDATA または ANYDATASET の単一の表の列に格納することもできます。たとえば、ANYDATA をアドバンスト・キューイングと併用して、異質な型のデータのキューイングをモデル化できます。基礎となるデータ型のデータは、他の任意のデータ同様に、問合せを実行できます。

前述の 3 つの汎用 SQL 型に対応するのが、それらをモデル化する OCI 型です。各 OCI 型は、それぞれの型の作成およびアクセスに使用する関数のセットを備えています。

- OCIType: SYS.ANYTYPE に対応しています。
- OCIAnyData: SYS.ANYDATA に対応しています。
- OCIAnyDataSet: SYS.ANYDATASET に対応しています。

> **参照：** OCIType、OCIAnyData、OCIAnyDataSet API およびそれらの使用方法の詳細は、『Oracle Call Interface プログラマーズ・ガイド』を参照してください。ANYTYPE、ANYDATA、ANYDATASET の各型に対するインタフェース、および ANYTYPE、ANYDATA、ANYDATASET と併用する DBMS_TYPES パッケージ（組込み型およびユーザー定義型の定数を定義）については、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

ユーザー定義集計ファンクション

Oracle は、レコード集合に対する操作に使用する、MAX、MIN、SUM をはじめとする数多くの定義済集計ファンクションを提供しています。これらの定義済集計ファンクションは、スカラー・データに対してのみ使用できます。ただし、これらのファンクションのカスタム実装を独自に作成できます。または、複雑なデータ（たとえば、オブジェクト型、不透明型、LOB 型を使用して格納されているマルチメディア・データなど）に対して使用する新しい集計ファンクションを定義することもできます。

ユーザー定義集計ファンクションは、Oracle 独自の組込み集計ファンクションと同様に、SQL DML 文で使用されます。このようなファンクションがサーバーに登録されていると、ネイティブのルーチンではなく、ユーザーが定義した集計ルーチンがコールされます。

ユーザー定義集計ファンクションは、スカラー・データに対しても使用できます。たとえば、財務アプリケーションまたは科学アプリケーションに対応付けられた複雑な統計データを処理するために特殊な集計ファンクションを実装すると効果的な場合があります。

ユーザー定義集計ファンクションは、拡張フレームワークの機能です。それらの実装には、ODCIAggregate インタフェース・ルーチンを使用します。

> **参照：** ODCIAggregate インタフェース・ルーチンを使用して、ユーザー定義集計ファンクションを実装する方法については、『Oracle9*i* Data Cartridge Developer's Guide』を参照してください。

Oracle オブジェクトを持つ表のパーティション化

パーティション化は、大規模な表や索引を、パーティションと呼ばれる小さく管理が容易な単位に分解することで、大規模な表や索引のサポート上の主要な問題が解決できます。Oracle では、オブジェクト、REF、VARRAY およびネストした表を含む表をパーティション化できるように、パーティション化の機能を拡張しています。LOB に格納された VARRAY は、LOB と同様の方法で同一レベル・パーティション化されます。

次の例では、列オブジェクト ShipToAddr の属性である zip コード（ToZip）に従って、発注書の表をパーティション化します。この例の目的上、パーティション化された VARRAY の記憶域を示すために、ネストした表 LineItemList は VARRAY にされています。

制限事項： ネストした表を含む表をパーティション化することはできますが、ネストした表に対応付けられた記憶表はパーティション化されません。

LineItemList は VARRAY として定義されていると仮定します。

```
CREATE TYPE LineItemList_vartyp as varray(10000) of LineItem_objtyp;

CREATE TYPE PurchaseOrder_typ AS OBJECT (
      PONo                NUMBER,
      Cust_ref            REF Customer_objtyp,
      OrderDate           DATE,
      ShipDate            DATE,
      OrderForm           BLOB,
      LineItemList        LineItemList_vartyp,
      ShipToAddr          Address_objtyp,

   MAP MEMBER FUNCTION
      ret_value RETURN NUMBER,

   MEMBER FUNCTION
      total_value RETURN NUMBER
  );

CREATE TABLE PurchaseOrders_tab of PurchaseOrder_typ
    LOB (OrderForm) store as (nocache logging)
    PARTITION BY RANGE (ShipToAddr.zip)
      (PARTITION PurOrderZone1_part
         VALUES LESS THAN ('59999')
         LOB (OrderForm) store as (
               storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100))
         VARRAY LineItemList store as LOB (
               storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100)),
    PARTITION PurOrderZone6_part
```

```
      VALUES LESS THAN ('79999')
      LOB (OrderForm) store as (
            storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100))
      VARRAY LineItemList store as LOB (
            storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100)),
PARTITION PurOrderZone0_part
   VALUES LESS THAN ('99999')
    LOB (OrderForm) store as (
            storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100))
    VARRAY LineItemList store as LOB (
            storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100)));
```

オブジェクト・ビューでのパラレル問合せ

ビューから合成されたオブジェクトに対しては、パラレル問合せがサポートされています。

（ORDER BY、GROUP BY および SET の操作を使用して）結合およびソートがパラレルに含まれる問合せを実行するには、MAP ファンクションが必要です。MAP ファンクションがない場合、問合せは自動的にシリアルになります。

ネストした表の列でのパラレル問合せは、サポートされていません。ビューに対してパラレル・ヒントまたはパラレル属性が存在しても、問合せにネストした表の列が含まれる場合は、シリアルになります。

パラレル DML は、INSTEAD OF トリガーを持つビューではサポートされません。ただし、トリガー内の個々の文は、パラレル化できます。

ロケータでネストした表のパフォーマンスを向上させる方法

コレクション型は、C++ や Java などの言語でのシステム固有の型または構造へ直接マップすることはありません。これらの言語を使用しているアプリケーションは、OCI などの Oracle インタフェースを介して、コレクションの内容にアクセスする必要があります。

一般に、クライアントが（オブジェクトをフェッチすることによって）ネストした表に明示的または暗黙的にアクセスすると、コレクション値全体がクライアント・プロセスに戻されます。パフォーマンス上の理由から、クライアントは、コレクションの内容全体を取り出すのを遅延または回避する場合があります。Oracle では、ネストした表の実際の値ではなくロケータを使用することによって、これに対処します。実際にコレクションの内容にアクセスが行われたときは、その内容がクライアントに自動的に転送されます。

ネストした表のロケータは、コレクション値へのハンドルに似ています。このロケータは、検索実行時のデータベース・スナップショットを確保することで、ネストした表の値またはコピー・セマンティクスを保とうとします。スナップショットによって、コレクション要素がロケータを使用してフェッチされたときに、データベースがネストした表の値の正しいインスタンスを取り出せるようになります。ロケータの有効範囲は 1 つのセッションにかぎられ、複数のセッションにまたがって使用することはできません。データベース・スナップショットが使用されているため、ネストした表の更新率が高い場合は、「snapshot too old」というエラーが発生する場合があります。LOB ロケータとは異なり、ネストした表のロケータは純粋なロケータであり、コレクション・インスタンスを変更するためには使用できません。

7

Oracle オブジェクトの使用方法に関する FAQ

この章では、Oracle のオブジェクト・リレーショナル機能に関して、ユーザーからよくある質問およびその答えについて説明します。内容は次のとおりです。

この章は、入門情報およびこのマニュアルの他の章を読んだ後にまだ疑問が残っている場合の参照用の情報を記載しています。

Oracle オブジェクトに関する一般的な質問

オブジェクト・リレーショナル機能は、別のオプションですか？

現在では、そうではありません。Oracle9*i* では、オブジェクト・リレーショナル機能は、サーバー製品の基本機能の一部です。

Oracle9*i* のオブジェクト・リレーショナルおよび拡張テクノロジの設計目標は何ですか？

Oracle9*i* のオブジェクトおよび拡張テクノロジの設計目標は、次のとおりです。

- 型システムを拡張してユーザー定義型をサポートすることによって、ユーザーがビジネス・オブジェクトをデータベース内にモデル化できるようにします。これらの型はアプリケーション・オブジェクトを綿密にモデル化し、データベース・サーバーにおいて、数値や文字などの組込み型と同様に扱えるようにするためのものです。
- Oracle データベース内に格納されたデータへのオブジェクト・ベースのアクセスを容易にするための構造基盤を提供します。また、この構造基盤によって、アプリケーションで使用されるデータ・モデルとデータベースがサポートするデータ・モデルとの不一致を最小にします。
- マルチメディア、財務および空間的アプリケーションに必要な新しいデータ型に対する組込みサポートを提供します。
- データベースの拡張性のフレームワークを提供し、新しいマルチメディア・データ型および複合データ型をサポートし、データベース内でシステム固有の管理ができるようにします。このフレームワークでは、データ・カートリッジを使用してサード・パーティが行う、データ・サーバーの拡張に必要な構造基盤を提供します。

このマニュアルでは、オブジェクト・リレーショナル・テクノロジについて説明しています。拡張性の詳細は、『Oracle9*i* Data Cartridge Developer's Guide』を参照してください。

オブジェクト型

構造化データとは何ですか？

SQL 92 標準では、ほとんどのデータベース・プログラミングで使用される 19 のアトミック・データ型が定義されています。これらの種類のデータは、単純構造化データといわれます。

Oracle オブジェクトでは、`REF` およびコレクションの概念が導入されています。これらの種類のデータは、複合構造化データといわれます。

LOB には、情報を格納するための別の方法があります。それらは、非構造化データといわれます。

ユーザー定義型、ユーザー定義ファンクションおよび抽象データ型はどこにありますか？

Oracle において、ユーザー定義型や抽象データ型は、オブジェクト型に相当します。

また、Oracle において、ユーザー定義ファンクションは、オブジェクト型メソッドに相当します。

このような用語を使用するのは、これらの意味が業界共通の使用方法とは異なるためです。たとえば、Oracle オブジェクトには NULL を指定できますが、抽象データ型には NULL を指定できません。

オブジェクト型とは何ですか？

Oracle9*i* では、オブジェクト型といわれるユーザー定義データ型の形式がサポートされています。オブジェクト型は、実社会のエンティティを抽象化したものです。オブジェクト型は、次のコンポーネントを持つスキーマ・オブジェクトです。

- スキーマ内でオブジェクト型を一意に識別する名前
- 実社会のエンティティの構造および状態をモデル化する属性
- 実社会のエンティティの動作を実装するメソッド

オブジェクト型はなぜ便利なのですか？

オブジェクト型は、C++ および Java でサポートされているクラス・メカニズムに似ています。オブジェクトを再利用できるため、より速く、効率的にデータベース・アプリケーションを開発できます。オブジェクトのサポートによって、複雑な実社会のビジネス・エンティティおよび論理のモデル化が簡単になります。オブジェクト型をデータベースで固有にサポートするので、アプリケーション開発者がクライアント側オブジェクトとデータベース・オブジェクトとの間のマッピング・レイヤーを記述する必要がなくなりました。また、オブジェクトを抽象化およびカプセル化することで、アプリケーションを簡単に理解およびメンテナンスできます。

Oracle9*i* では、オブジェクト・データはどのように格納および管理されますか？

オブジェクトは、データ・サーバーによってシステム固有に管理されます。オブジェクト型は、列の型（列オブジェクト）、またはオブジェクト表内の各行の型（行オブジェクト）として使用できます。列オブジェクトとして使用された場合、オブジェクト型はリレーショナ

ル表のフィールドとして動作します。各行オブジェクトには、OID といわれる一意の識別情報が付きます。

オブジェクトは、データベース・コンポーネントと完全に統合されます。オブジェクトには、索引付けおよびパーティション化を行うことができます。たとえば、オブジェクトを含む問合せはパラレル化でき、統計情報を使用したコストベースのオプティマイザによって最適化されます。

定評ある Oracle データ・サーバーを基盤として作成されているため、オブジェクトは、リレーショナル・データと同様の信頼性、可用性およびスケーラビリティで管理されます。

Oracle9*i* では、継承はサポートされていますか？

ユーザー定義の SQL 型の単一継承がサポートされます。単一のスーパータイプから複数のサブタイプを導出できます。サブタイプ自体をさらに分割することもできるので、希望の数のレベルを持つ型階層を作成できます。その型をサブタイプ化するか、またはインスタンス化するかを制御するためのキーワードが提供されています。

また、C++ および Java マッピングを介してのクライアント側の継承がサポートされています。C++ の場合、Oracle Designer に付属の Object Database Designer を使用して、統一モデリング言語（UML）の図に基づいた DDL および C++ コードを生成します。Java の場合、Oracle JDBC ドライバの CustomDatum 機能を使用します。

サーバー側メソッド継承は、Oracle9*i* JVM によって Java に提供されます。

オブジェクト・メソッド

オブジェクト・メソッドはどの言語を使用して記述できますか？

メソッドは、PL/SQL、Java、C または C++ で実装できます。C および C++ は、外部プロシージャ機能を介してサポートされます。一方、PL/SQL メソッドおよび Java メソッドは、サーバーのアドレス空間内で実行されます。オブジェクト型が様々なプログラミング言語で実装可能であっても、SQL のメソッド仕様部をその実装から分離すると、オブジェクト型にメソッドを起動する方法が 1 つになります。Oracle では、PL/SQL メソッド、Java メソッドおよび外部 C プロシージャをサーバーから起動するための、安全で保護された環境を提供しています。ユーザー・メソッドにプログラミング・エラーがあっても、サーバーが失敗したりデータベースが破損することはありません。そのため、ミッション・クリティカルな環境におけるサーバーの信頼性および可用性が保証されます。

オブジェクト・メソッドに PL/SQL と Java のどちらを使用するかは、どのように判断しますか？

Oracle では、PL/SQL および Java はサーバー・プログラミング言語として相互変換して使用できます。PL/SQL は、SQL のシームレスな拡張で、データベース内で SQL 集約操作を

行うために適した言語です。Java は、データベース・サーバーを含む、複数層で実行する移植可能なアプリケーションを記述するために適した発展段階の言語です。

外部プロシージャはいつ使用する必要がありますか？

外部プロシージャは、通常、C などの低レベル言語で記述されるのが最適な計算集中型操作に使用されます。また、データ・サーバー内で実行するために Java または PL/SQL で簡単に書きなおすことができない既存のライブラリにあるルーチンを起動することにも役立ちます。

外部プロシージャをコールするプロセス間通信（IPC）オーバーヘッドは、PL/SQL プロシージャまたは Java プロシージャをコールする IPC オーバーヘッドより大きくなります。ただし、外部プロシージャで行われる計算が、複雑で何万もの指示からなる場合、外部プロシージャをコールするオーバーヘッドが重要でなくなります。

定義者権限および実行者権限とは何ですか？

定義者権限と実行者権限の区別は、オブジェクト以外にも適用されます。オブジェクト指向のプログラムには、通常、再利用可能なモジュールが含まれるため、実行者権限は特に役立つ場合があります。

オブジェクト・メソッドは、メソッド定義に基づいて、所有者の権限（定義者権限）または現行のユーザーの権限（実行者権限）で実行されます。実行者権限は、再利用可能なオブジェクトを記述する場合に役立ちます。これは、これらのオブジェクトのユーザーは、オブジェクトの作成者に対して表へのアクセス権限を付与する必要がないためです。定義者権限は、オブジェクトの実装の一部として、オブジェクト・メソッドがオブジェクトの作成者によってメンテナンスされるメタデータにアクセスする必要がある場合に役立ちます。メタデータにアクセスするメソッドは、オブジェクトの作成者が所有のメタデータをユーザーに公開する必要がないように、定義者権限を使用して実行されます。

オブジェクト参照

オブジェクト参照とは何ですか？

オブジェクト参照（REF）は、オブジェクト表に格納された行オブジェクト、またはオブジェクト・ビューから作成されたオブジェクトを、一意に識別します。通常、REF 値は、オブジェクトの一意の識別子、オブジェクト表に対応付けられた一意の識別子、およびそのオブジェクトが格納されているオブジェクト表の行の ROWID から導出されます。オプションの ROWID は、オブジェクトへの高速アクセスのヒントとして使用されます。

オブジェクト参照はいつ使用する必要がありますか？外部キーとの違いは何ですか？

外部キーと同様に、オブジェクト参照は、複雑な関連をモデル化する場合に役立ちます。複雑な関連をモデル化する場合、次の理由で、オブジェクト参照の方が外部キーより柔軟性があります。

- オブジェクト参照は、強力に型指定されており、コンパイル時のタイプ・チェックが向上します。
- オブジェクト参照のコレクションを使用して、1 対多リレーションシップをモデル化できます。
- アプリケーションにおいて、SQL 文を作成しなくても、オブジェクト参照を使用して簡単にオブジェクトをナビゲートし取り出すことができます。
- SQL の REF ナビゲーションによって、複雑な結合を行う必要がなくなります。
- オブジェクト参照によって、アプリケーションは、サーバーへの 1 回の要求で、REF によって関連付けられたオブジェクトを取り出せます。

主キーに基づいてオブジェクト参照を作成できますか？

できます。外部キーに基づいてオブジェクト参照を作成し、次のオブジェクトを参照できます。

- オブジェクト・ビュー : オブジェクト・ビューを使用してリレーショナル表からオブジェクトを作成する場合、作成されたオブジェクトの OID は、通常、基礎となるリレーショナル表上の主キーに基づきます。
- 主キー・ベースの OID があるオブジェクト表 : オブジェクト表を定義する場合、システムによって生成される OID ではなく、行オブジェクトの OID として主キーを指定するオプションが提供されています。

有効範囲付き REF とは何ですか？また、いつ使用する必要がありますか？

一般的に、列には、オブジェクトが格納されているオブジェクト表に関係なく、特定の宣言された型へのオブジェクト参照が含まれる場合があります。ただし、REF 型の列は、指定したオブジェクト表からのオブジェクトへの参照のみを含むよう有効範囲（制約）を指定できます。有効範囲付き REF の表識別子はシステムに格納されないため、有効範囲付き REF は標準の REF よりディスクの占有サイズが小さくて済みます。そのため、できるだけ有効範囲付き REF を使用してください。また、有効範囲付き REF のナビゲーションを含む問合せは、適切な場合、結合に最適化されます。

PL/SQL および Java でオブジェクト参照を使用してオブジェクトを操作できますか？

できます。PL/SQL および Java の両方ともオブジェクト参照をサポートしています。PL/SQL では、UTL_REF パッケージにオブジェクト参照を与えることで、オブジェクトの取出しおよび更新ができます。Java では、get メソッドおよび set メソッドを使用してオブジェクト参照を参照クラスにマップし、オブジェクトの取出しおよび更新ができます。

コレクション

Oracle9*i* ではどのような種類のコレクションがサポートされていますか？

Oracle9*i* では、可変配列（VARRAY）とネストした表という 2 種類のコレクションがサポートされています。表のオブジェクト型および列の属性は、コレクション型に指定できます。また、コレクション自体もコレクション型に指定できます。VARRAY とネストした表を使用することによって、アプリケーションは、1 対多リレーションシップおよび多対多リレーションシップをデータベース・スキーマ内でシステム固有にモデル化できます。

Oracle オブジェクトは、コレクション内でのコレクションをサポートしますか？

サポートします。VARRAY は別の VARRAY やネストした表を持つことができ、ネストした表は別のネストした表や VARRAY を持つことができます。同様に、コレクション型の属性を持つオブジェクト型のコレクションを持つことができます。このようなマルチレベル・コレクションは、何レベルにでもネストすることができます。

コレクションのモデル化に VARRAY とネストした表のどちらを使用するかは、どのように判断すればよいですか？

コレクション要素の順序を保持する必要がある場合には、VARRAY が役立ちます。VARRAY は、常にコレクション全体を 1 つの単位として操作し、コレクションの個々の要素に問合せを要求しない場合に、非常に効率的です。VARRAY のサイズが小さい場合は、含んでいる行とともにインラインで格納されます。また、VARRAY のサイズが一定のしきい値より大きい場合は、自動的に LOB として格納されます。

コレクション要素間に順序がなく、個々の要素へ効率的に問合せを行うことが重要である場合には、ネストした表が役立ちます。コレクション型列の要素は、リレーション・スキーマ内の親子表に類似した、個々の表に格納されます。

コレクション・ロケータとは何ですか？

コレクション・ロケータによって、アプリケーションは、メモリー内にコレクションを具体化することなく、大きなコレクションを取り扱うことができます。これによって、大きいコレクションをインタフェース間で効率的に転送できます。コレクションは、アプリケーションが最初にその要素にアクセスしたときに透過的に具体化されます。また、アプリケーションは、ロケータを使用してコレクションのサブセットに対する問合せおよび取出しができます。

コレクション・ロケータを扱うためには、CREATE および ALTER TABLE DDL で指定します。コレクションへのアクセスはカプセル化されているため、アプリケーションは、値として戻されるように指定されたネストした表の場合と同じインタフェースを使用して、ロケータとして戻されるように指定されたネストした表を取り出します。

コレクション・ネスト解除とは何ですか？

コレクション・ネスト解除によって、アプリケーションは、指定した行の一連のコレクションに対して効率的に問合せができます。この問合せは、リレーショナル・スキーマ内における指定した親である行の子である行に対する問合せに類似しています。コレクション・ネスト解除を利用することで、アプリケーションは、コレクション・フォーム内またはフラット親子フォーム内の 1 対多リレーションシップを柔軟に表示できます。

オブジェクト・ビュー

オブジェクト・ビューとオブジェクト表の違いは何ですか？

リレーショナル・ビューとリレーショナル表に類似点があるように、オブジェクト・ビューには、次のようなオブジェクト表に類似したプロパティがあります。

- オブジェクトを行に含みます。ビューの列は、オブジェクト型の最上位の属性にマップします。
- 各オブジェクトには、対応付けられた識別子があります。識別子は、ビュー定義者によって指定されます。ほとんどの場合、ベースとなる表の主キーが識別子として動作します。

オブジェクト・ビューは更新可能ですか？

すべての属性が表の実列にマップされている場合、オブジェクト・ビューは簡単に更新できます。CAST-MULTISET などのより複雑な技法を使用して属性を導出するビューには、INSTEAD OF トリガーを使用して更新できます。そのようなビューを更新すると（あるいは挿入または削除すると）、システムは、ベース表を暗黙的に変更するのではなく、ビューに指定された INSTEAD OF トリガーを起動します。トリガー本体に含める必要があるすべての更新セマンティクスはカプセル化できます。

オブジェクト・キャッシュ

なぜオブジェクト・キャッシュが必要なのですか？

オブジェクト・キャッシュを使用することで、アプリケーションには次のような利点があります。

- データベース・オブジェクトをメモリー内のホスト言語オブジェクトに透過的にマップできます。
- 永続オブジェクトに対して透過的で効率的なメモリー管理ができます。アプリケーションがデータベース・オブジェクトにアクセスするためのメモリーの割当てを意識する必要はありません。
- クライアント側オブジェクトに対してトランザクション・セマンティクスが使用できます。オブジェクト・キャッシュで変更された永続オブジェクトは、クライアントとサーバー間の 1 回のラウンドトリップで、データベースに反映されます。
- オブジェクトのナビゲーショナル・アクセスが可能です。オブジェクト・キャッシュによって、ナビゲーション・スタイルでオブジェクトにアクセスできます。OCI のオブジェクト機能を使用すると、REF を確保することによって、オブジェクトがオブジェクト・キャッシュにフェッチされます。オブジェクトへのナビゲーショナル・アクセスは、REF を介して相互接続されているオブジェクトを操作する場合により適しています。
- 複合オブジェクトの取出しが可能です。つまり、オブジェクトをサーバーからフェッチする 1 回の要求で、REF を介してそのオブジェクトに関連付けられている他のオブジェクトを含めて、クライアント・サーバー間の 1 回のラウンドトリップで取り出すことができます。

オブジェクト・ロックはオブジェクト・キャッシュでサポートされますか？

オブジェクト・キャッシュでは、即時ロック構造およびコミット時ロック構造の両方がサポートされます。

- 即時ロック構造では、オブジェクトは、キャッシュにあるオブジェクトが変更される前にロックされ、ロックを所有しているトランザクションがコミットまたはロールバックを実行するまで、他のどのユーザーもそのオブジェクトを変更できないようにします。
- コミット時ロック構造では、オブジェクトは、ロックを取得せずに、キャッシュ内でフェッチおよび変更されます。オブジェクトがサーバーにフラッシュされる場合にのみ、ロックが取得されます。コミット時ロックでは、即時ロックより高度な同時アクセスが可能です。コミット時ロックを効果的に使用するために、オブジェクト・キャッシュは、オブジェクトがキャッシュにフェッチされてから、他のユーザーによって変更されたかどうかを検出する機能を提供します。オブジェクト変更検出モードにすると、オブジェクトがキャッシュにフェッチされてから他のユーザーによって変更されていない場合にのみ、そのオブジェクトの変更が持続的に行われます。

ラージ・オブジェクト（LOB）

Oracle を使用して LOB を管理するにはどのようにすればいいですか？

テキスト、イメージ、オーディオ、ビデオなどのマルチメディア・データ型をサポートするには、バイナリおよび文字データに対する強力なサポートが必要です。このような種類のデータは大きくなる傾向があり、様々なバイナリ・データに直接アクセスする必要があります。そのため、Oracle では、大規模のバイナリ・データおよび文字データのサポートを大幅に改善しました。ここで LOB 型が導入され、イメージ、オーディオ・ファイル、テキストおよび空間データを含む、様々なドメインからのドメイン固有の大きなデータを格納するために使用されます。

Oracle では、バイナリ、文字ベースおよびファイル・ベースという 3 種類のラージ・データ型をサポートしています。LOB を作成する機能に加えて、Oracle サーバーでは、バイナリ・データを管理するために改善された点もいくつかあります。これらの改善点は、次のとおりです。

- 表に 1 つ以上の LOB 列を定義するためのサポート
- LOB データへのランダムでピース単位のアクセス
- LOB データを単一のストリームとして転送するためのサポート
- LOB データに対するロギングおよびキャッシュを禁止するためのサポート
- LOB をインライン行記憶域から別のセグメントまたは別の表領域内の行外記憶域に透過的に移動するためのサポート

LOB の詳細は、『Oracle9*i* アプリケーション開発者ガイド - ラージ・オブジェクト』を参照してください。

ユーザー定義演算子

ユーザー定義演算子とは何ですか？

Oracle では、オブジェクト指向アプリケーションの開発者は、ユーザー定義演算子と呼ばれるドメイン固有の演算子（たとえば、Contains、Within_Distance、Similar）を持つ組込み関係演算子（たとえば、+、-、/、*、LIKE）のリストを拡張できます。ユーザー定義演算子は、SELECT 構文のリストまたは where 句など、組込み演算子が使用できるところではどこでも使用できます。組込み演算子と同様に、ユーザー定義演算子は別の型の引数をサポートできます。また、索引を使用して評価できます。

参照： ユーザー定義演算子の詳細は、『Oracle9*i* SQL リファレンス』の「CREATE OPERATOR」を参照してください。

- 『Oracle9*i* Data Cartridge Developer's Guide』

ユーザー定義演算子はなぜ役に立つのですか？

組込み演算子と同様に、ユーザー定義演算子では、オブジェクト・データに対して効果的な内容ベースの問合せおよびソートができます。たとえば、一定の資格を含む履歴書を検索するには、Contains 演算子を SQL の where 句の一部として指定できます。オプティマイザは、関係演算子を評価するために B ツリー索引を使用するのと同様に、履歴書の列にテキスト索引を使用するように選択して、問合せを効率的に実行します。

8

Oracle オブジェクトの設計上の考慮点

この章では、Oracle のオブジェクト・リレーショナル・モデルの実装およびパフォーマンス特性について説明します。ここで説明する内容は、論理データ・モデルを Oracle の物理的な実装にマップしたり、オブジェクト指向機能を使用するアプリケーションを開発する場合に役立ちます。

内容は次のとおりです。

- オブジェクト表およびオブジェクト列のレプリケーション
- オブジェクトに対する制約
- 型進化
- パフォーマンス・チューニング
- Oracle オブジェクトでのパラレル問合せ
- ヒントおよび技法

この章を読む前に、Oracle オブジェクトの背景となる基本概念について理解しておいてください。

参照：

- Oracle オブジェクトの概念的な説明については、『Oracle9*i* データベース概要』を参照してください。
- Oracle オブジェクトを使用するための SQL 文については、『Oracle9*i* SQL リファレンス』を参照してください。

列または行としてのオブジェクトの表現

オブジェクトは、列オブジェクトとしてリレーショナル表の列に格納するか、または行オブジェクトとしてオブジェクト表に格納できます。そのオブジェクトが含まれているリレーショナル・データベース・オブジェクトの外で意味を持つオブジェクト、または複数のリレーショナル・データベース・オブジェクトの間で共有されるオブジェクトは、行オブジェクトとして参照できるようにする必要があります。したがって、そのようなオブジェクトは、リレーショナル表の列に格納するのではなく、オブジェクト表に格納する必要があります。

たとえば、オブジェクト型 CUSTOMER のオブジェクトは、特定の発注書の外で意味を持ち、他から参照できるようにする必要があります。したがって、CUSTOMER オブジェクトは、オブジェクト表の行オブジェクトとして格納する必要があります。ところが、オブジェクト型 address のオブジェクトは、特定の発注書の外ではあまり意味を持たず、発注書内の 1 つの属性です。したがって、address オブジェクトは、リレーショナル表またはオブジェクト表の列に列オブジェクトとして格納する必要があります。そのため、address は、customer 行オブジェクトの中の列オブジェクトになる場合もあります。

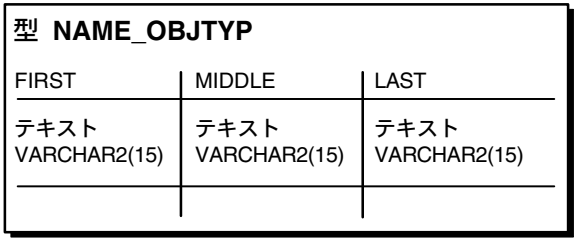
列オブジェクトの記憶域

列オブジェクトの記憶域は、1 つの集合としてオブジェクトを形成する、同等のスカラー列集合の記憶域と同じです。違いは、オブジェクトのアトミック NULL 値およびその埋込みオブジェクト属性をメンテナンスするオーバーヘッドが加わることのみです。これらの値は、それぞれの列オブジェクトが NULL かどうか、およびその埋込みオブジェクト属性が NULL かどうかを指定するため、NULL インジケータといわれます。ただし、NULL インジケータは、列オブジェクトのスカラー属性が NULL かどうかは指定しません。Oracle では、別の方法でスカラー属性が NULL かどうかを判断します。

ある組織の構成員の識別番号、名前、住所および電話番号を持つ表を考えてみます。名前、住所および電話番号を保持するために、異なる 3 つのオブジェクト型を作成できます。まず、次の SQL 文を入力して、name_objtyp オブジェクト型を作成します。

```
CREATE TYPE name_objtyp AS OBJECT (
  first       VARCHAR2(15),
  middle      VARCHAR2(15),
  last        VARCHAR2(15));
```

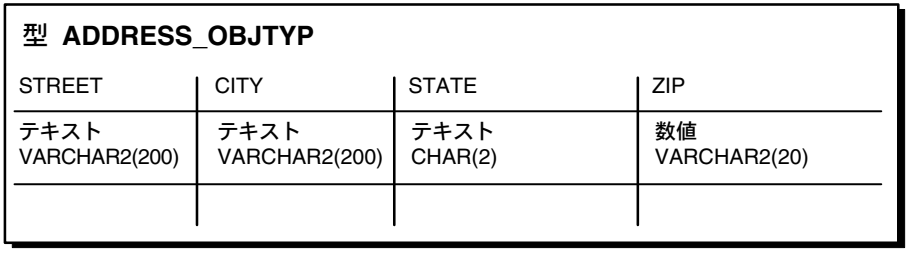
図 8-1 name_objtyp 型のオブジェクト・リレーショナル表現



次に、次の SQL 文を入力して、address_objtyp オブジェクト型を作成します。

```
CREATE TYPE address_objtyp AS OBJECT (
  street       VARCHAR2(200),
  city         VARCHAR2(200),
  state        CHAR(2),
  zipcode      VARCHAR2(20));
```

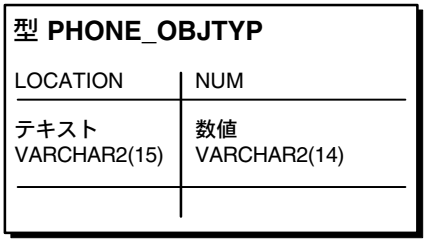
図 8-2 address_objtyp 型のオブジェクト・リレーショナル表現



最後に、次の SQL 文を入力して、phone_objtyp オブジェクト型を作成します。

```
CREATE TYPE phone_objtyp AS OBJECT (
  location     VARCHAR2(15),
  num          VARCHAR2(14));
```

図 8-3 phone_objtyp 型のオブジェクト・リレーショナル表現



各構成員が複数の電話番号を持っている場合があるため、phone_objtyp オブジェクト型に基づいてネストした表型 phone_ntabtyp を作成します。

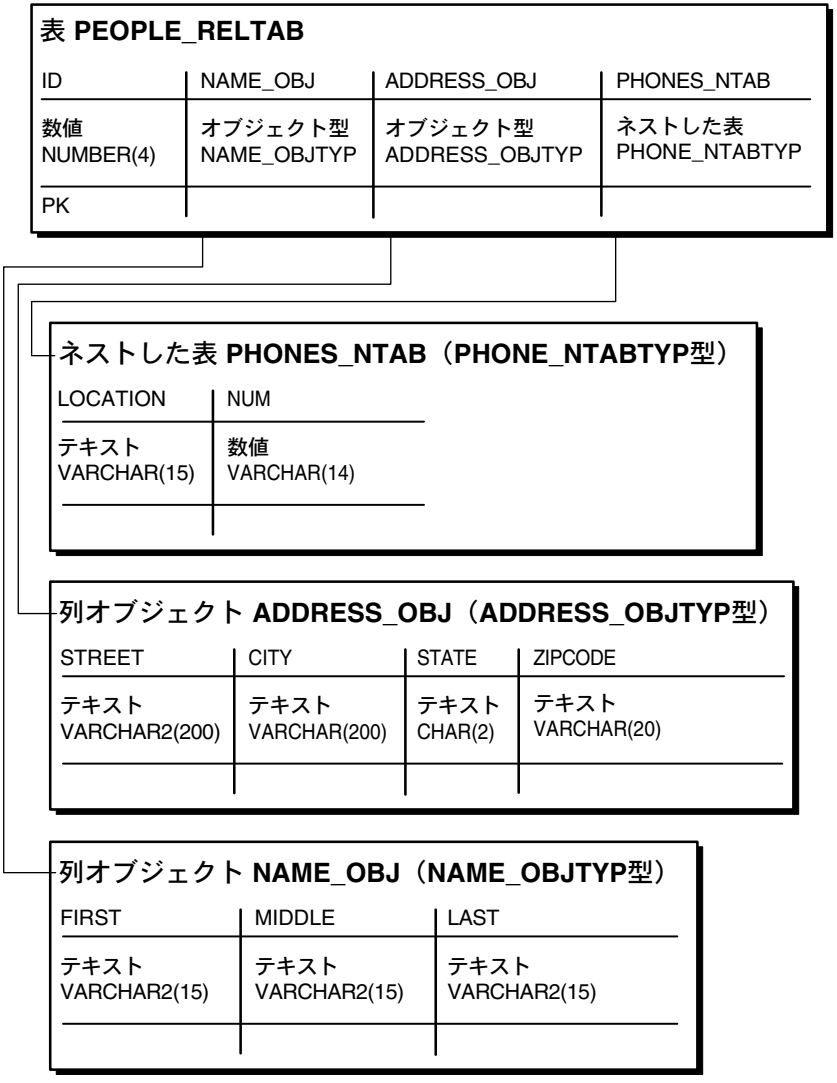
```
CREATE TYPE phone_ntabtyp AS TABLE OF phone_objtyp;
```

> **参照：** ネストした表の詳細は、8-15 ページの「ネストした表」を参照してください。

これらのオブジェクト型を準備し、次の SQL 文を使用して、組織の構成員についての情報を保持する表を作成できます。

```
CREATE TABLE people_reltab (
  id            NUMBER(4)   CONSTRAINT pk_people_reltab PRIMARY KEY,
  name_obj      name_objtyp,
  address_obj   address_objtyp,
  phones_ntab   phone_ntabtyp)
  NESTED TABLE  phones_ntab STORE AS phone_store_ntab;
```

図 8-4 people_reltab リレーショナル表の表現



people_reltab 表には、name_obj、address_obj および phones_ntab の 3 つの列オブジェクトがあります。phones_ntab 列オブジェクトは、ネストした表でもあります。

注意： people_reltab 表とその列および対応付けられている型は、この章の他の例でも使用します。

people_reltab 表に格納される各オブジェクトの記憶域は、オブジェクトの属性の記憶域と同じです。たとえば、name_obj オブジェクトに必要な記憶域は、NULL インジケータのオーバーヘッドを除いて、first 属性、middle 属性、last 属性の組合せに対する記憶域と同じです。

COMPATIBLE パラメータが 8.1.0 以上に設定された場合、オブジェクトの NULL インジケータおよびその埋込みオブジェクト属性には、それぞれ 1 ビットが割り当てられます。したがって、(すべてのネスト・レベルのオブジェクトを含めて) n 個の埋込みオブジェクト属性を持つオブジェクトには、記憶域のオーバーヘッドが CEIL(n/8) バイトあります。たとえば、people_reltab 表では、各行に対する NULL 情報のオーバーヘッドは、CEIL(3/8) つまり CEIL(.37) になり、これは 1 バイトに丸められます。この場合、各行には、name_obj、address_obj および phones_ntab の 3 つのオブジェクトがあります。

ただし、COMPATIBLE パラメータが 8.1.0 未満、たとえば 8.0.0 に設定された場合、記憶域は、次の計算によって決定されます。

```
CEIL(n/8) + 6
```

ここで、n はオブジェクト内のすべての属性（スカラーおよびオブジェクト）の合計数です。したがって、たとえば people_reltab 表では、各行に対する NULL 情報のオーバーヘッドは次の計算から 7 バイトになります。

```
CEIL(4/8) + 6 = 7
```

CEIL(4/8) は CEIL(.5) で、これは 1 バイトに丸められます。この場合、各行にはオブジェクトが 3 つとスカラーが 1 つあります。

したがって、列オブジェクトを操作する際の記憶域のオーバーヘッドおよびパフォーマンスは、同等のスカラー列集合の場合と類似しています。コレクション属性の記憶域については、8-12 ページの「ネストを解除する問合せを使用したリレーショナル形式でのオブジェクト・データの表示」を参照してください。

参照： CEIL の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

オブジェクト表の行オブジェクトの記憶域

行オブジェクトは、オブジェクト表に格納されます。オブジェクト表は、オブジェクトを格納し、このオブジェクト属性に対してリレーショナル表と同様にアクセスできるリレーショナル・ビューを提供する特殊な表です。オブジェクト表は、オブジェクト表に格納された、オブジェクトの最上位の属性に対応するデータ型の列を持つリレーショナル表と、論理的、

物理的に類似しています。主な違いは、オブジェクト表には、追加の OID の列および索引をオプションで含めることができることです。

OID 記憶域および OID 索引 Oracle では、デフォルトで、すべての行オブジェクトに OID と呼ばれる一意で不変の OID が割り当てられます。OID を利用することで、対応する行オブジェクトを、他のオブジェクトまたはリレーショナル表から参照できます。このような参照は、REF と呼ばれる組込みデータ型によって表されます。REF によって、指定されたオブジェクト型の行オブジェクトへの参照がカプセル化されます。

デフォルトでは、オブジェクト表にはシステムによって生成される OID 列が含まれるため、各行オブジェクトには、グローバルに一意な OID が割り当てられます。この OID 列は、効率的な OID ベースの検索ができるように自動的に索引付けされます。OID 列は、16 バイトの主キー列を別に持つのと同じ効果があります。

主キー・ベースの OID 主キー列が使用可能な場合、16 バイトの OID 列およびその索引をメンテナンスする際の記憶域およびパフォーマンスのオーバーヘッドを回避できます。システムによって生成される OID を使用するかわりに、CREATE TABLE 文を使用して、システムが主キー列を表のオブジェクトの OID に使用するように指定できます。したがって、既存の列をオブジェクトの OID として使用することもできるし、Oracle によって生成されたグローバルに一意な 16 バイトの OID よりも小さい、アプリケーションによって生成された OID を使用することもできます。

オブジェクト比較のパフォーマンス

オブジェクト型に定義されたマップ・メソッドまたはオーダー・メソッドを起動することによって、オブジェクトを比較できます。マップ・メソッドは、オブジェクトの順序付けを行う際にオブジェクトをスカラー値に変換します。オブジェクトは、スカラー値にマップされると効率的に順序付けできるため、可能であればオブジェクトをスカラー値に変換することをお薦めします。

ORDER BY または GROUP BY の処理のためにオブジェクトのソートが必要な場合、オブジェクトのマップ方法がパフォーマンスに大きく影響します。オブジェクトを他のオブジェクトと何度も比較する必要がある場合があり、最初にオブジェクトをスカラー値にマップできると、効率が大幅に向上するからです。比較セマンティクスが非常に複雑な場合、またはオブジェクトを比較用のスカラー値にマップできない場合、指定された 2 つのオブジェクトに対して、オブジェクト作成者によって決定された順序を戻すオーダー・メソッドを定義できます。オーダー・メソッドは、マップ・メソッドほど効率的でないため、オーダー・メソッドを使用するとパフォーマンスが低下する場合があります。どのオブジェクト型でも、マップ・メソッドまたはオーダー・メソッドのどちらか一方を実装できますが、両方は実装できません。

ここでもう一度、4 つの文字属性、street、city、state および zipcode で構成されるオブジェクト型 address について考えてみます。この場合は、各オブジェクトは簡単にスカラー値に変換できるため、最も効率的な比較メソッドはマップ・メソッドです。たとえば、州によってすべてのオブジェクトを順序付けるマップ・メソッドを定義できます。

一方、イメージなどのバイナリ・オブジェクトを比較する場合を考えてみます。この場合、比較セマンティクスが複雑すぎてマップ・メソッドは使用できない可能性があります。その場合は、オーダー・メソッドを使用して比較を実行できます。たとえば、各イメージの明度またはピクセル数によってイメージを比較するオーダー・メソッドを作成できます。

マップ・メソッドもオーダー・メソッドもないオブジェクト型のオブジェクトに対しては、等価比較のみが可能です。この場合、Oracle では、対応するオブジェクト属性のフィールド同士が定義順に比較されます。いずれかのフィールドの比較結果が不一致となると、その時点で、FALSE 値が戻されます。すべてのフィールドの比較において一致が確認されると、TRUE 値が戻されます。ただし、オブジェクトが LOB 属性のコレクションを持つ場合、フィールド・ベースのオブジェクトの比較は行われません。LOB 属性を持つオブジェクトの比較を実行するためにはマップ・メソッドまたはオーダー・メソッドが必要です。

オブジェクト識別子（OID）の記憶域上の考慮点

REF は、OID を使用してオブジェクトを指します。システムによって生成される OID または主キー・ベースの OID のどちらかを使用できます。この 2 種類の OID の違いについては、8-7 ページの「オブジェクト表の行オブジェクトの記憶域」を参照してください。オブジェクト表に対して、システムが生成する OID を使用する場合、Oracle がこれらの OID を格納する列の索引をメンテナンスします。索引には記憶域が必要で、個々の行オブジェクトには、システムが生成した OID があります。このような OID には、行当たり 16 バイトの記憶域が必要です。

システムによって生成される OID のかわりに、OID として主キーを使用することで、このような記憶域要件の増加を回避できます。リレーショナル表の外部キーと同様の方法で、主キー・ベースの OID を持つ行オブジェクトの参照を格納する列に対して、参照整合性を確保できます。

ただし、各主キーのサイズが 16 バイトを超え、数多くの REF がある場合は、各 REF のサイズは主キーと同じであるため、主キー・ベースの OID を使用すると、システムによって生成される OID より多くの領域が必要になる場合があります。また、主キー・ベースの各 OID は、ローカルに一意です（グローバルに一意である必要はありません）。グローバルに一意の識別子が必要な場合は、主キーがグローバルに一意であることを保証するか、またはシステムによって生成される OID を使用する必要があります。

REF の記憶域サイズ

REF には、次の 3 つの論理コンポーネントが含まれています。

- 参照されるオブジェクトの OID。システムによって生成される OID の長さは 16 バイトです。主キー・ベースの OID のサイズは、主キー列のサイズによって決まります。
- 参照されるオブジェクトが含まれている表またはビューの OID。この長さは 16 バイトです。
- ROWID ヒント。この長さは 10 バイトです。

REF 列に対する整合性制約

REF 列に対する参照整合性制約によって、その REF に対応する行オブジェクトが存在することが保証されます。REF に対する参照整合性制約によって、リレーショナル・データに対して主キー / 外部キー関連を指定した場合と同じ関連が生成されます。参照整合性制約は、REF に対応する行オブジェクトが存在することを保証する唯一の方法であるため、通常はできるだけ参照整合性制約を使用してください。ただし、ネストした表にある REF に対しては、参照整合性制約を指定できません。

有効範囲付き REF のパフォーマンスおよび記憶域に関する考慮点

有効範囲付き REF には、指定したオブジェクト表の参照のみを含む、という制約があります。有効範囲付き REF は、REF となる列型、コレクション要素、またはオブジェクト型属性を宣言するときに指定できます。

有効範囲付き REF の方が格納する際に効率的なため、通常は、有効範囲なしの REF ではなく、有効範囲付き REF を使用してください。有効範囲なしの REF を格納するには、36 バイト以上（ROWID を使用する場合は 37 バイト以上）必要ですが、有効範囲付き REF の格納には、ターゲット・オブジェクトの OID と同じだけの領域のみを必要とするため、参照される OID がシステムによって生成された OID か、主キー・ベースの OID かによりますが、16 バイト未満で格納できる場合もあります。システムによって生成された OID の場合は、16 バイト必要です。主キー・ベースの OID の場合は、主キー値を格納できるだけの領域が必要ですが、これは 16 バイト未満の場合もあります（ただし、主キー・ベースの OID に対する REF は、選択時に動的な構成が必要なので、システムによって生成された OID に対する REF の場合と比較して、多くのメモリー領域が必要となる場合があります）。

有効範囲付き REF は記憶域が少なくて済むうえに、オプティマイザによって、有効範囲付き REF を参照解除する問合せを最適化して、さらに効率的な結合にすることができます。有効範囲なしの REF に対しては、この最適化はできません。問合せの最適化時に、有効範囲なしの REF をどの表に含めるべきか、オプティマイザによる判断が不可能なためです。

ただし、参照整合性制約とは異なり、有効範囲付き REF によっては、参照される行オブジェクトの存在は保証されません。保証されるのは、参照されるオブジェクト表の存在のみです。したがって、行オブジェクトに対して有効範囲付き REF を指定した後でその行オブジェクトを削除すると、参照対象となるオブジェクトがなくなるため、その有効範囲付き REF は、参照先がない REF になります。

注意： 参照整合性制約には、暗黙的に有効範囲が付きます。

アプリケーション設計上、参照されるオブジェクトが複数の表に分散している場合は、有効範囲なしの REF が便利です。ROWID ヒントは有効範囲付き REF に対しては無視されるため、「WITH ROWID オプションを使用したオブジェクト・アクセスの高速化」で説明するとおり、ROWID ヒントによるパフォーマンスの向上の方が、有効範囲付き REF を使用した

場合の記憶域の節約および問合せの最適化よりも重要である場合は、有効範囲なしの REF を使用してください。

有効範囲付き REF の索引付け

CREATE INDEX コマンドを使用して、有効範囲付き REF 列に対する索引を作成できます。作成した索引を使用して、有効範囲付き REF を参照解除する問合せを効率的に評価できます。このような問合せは暗黙的に結合に変換されます。Oracle では、ある種の問合せに対しては、有効範囲付き REF 列の索引を使用して結合を効率的に評価できます。

たとえば、オブジェクト型 address_objtyp を使用して address_objtab という名前のオブジェクト表を作成するとします。

```
CREATE TABLE address_objtab OF address_objtyp ;
```

次に、住所に対して REF が使用されること以外は、8-3 ページの「列オブジェクトの記憶域」で説明した people_reltab 表と同じ定義を持つ people_reltab2 表を作成できます。

```
CREATE TABLE people_reltab2 (
  id            NUMBER(4)   CONSTRAINT pk_people_reltab2 PRIMARY KEY,
  name_obj      name_objtyp,
  address_ref   REF address_objtyp SCOPE IS address_objtab,
  phones_ntab   phone_ntabtyp)
  NESTED TABLE  phones_ntab STORE AS phone_store_ntab2 ;
```

これで address_ref 列に対して索引を作成できます。

```
CREATE INDEX address_ref_idx ON people_reltab2 (address_ref) ;
```

次の問合せで address_ref が参照解除されます。

```
SELECT id FROM people_reltab2 p
   WHERE p.address_ref.state = 'CA' ;
```

この問合せの実行時には、効率的に評価を行うために address_ref_idx 索引が使用されます。ここで、address_ref は、address_objtab オブジェクト表に格納される住所の参照を格納した有効範囲付き REF 列です。前述の問合せは、結合を持つ問合せに暗黙的に変換されます。

```
SELECT p.id FROM people_reltab2 p, address_objtab a
   WHERE p.address_ref = ref(a) AND a.state = 'CA' ;
```

Oracle のオプティマイザによって、address_objtab を外部表としてネステッド・ループ結合を実行し、有効範囲付き REF 列 address_ref の索引を使用して、一致する住所を検索する計画が作成される場合があります。

WITH ROWID オプションを使用したオブジェクト・アクセスの高速化

REF 列に対して WITH ROWID オプションが指定されていると、REF で参照されるオブジェクトの ROWID がメンテナンスされます。こうしておくと、REF に含まれている ROWID を直接使用して参照されるオブジェクトの検索が実行できるため、OID 索引から ROWID をフェッチする手間が省けます。したがって、ROWID ヒントを指定するには、WITH ROWID オプションを使用してください。ROWID を含めることによって、REF の記憶域要件が 10 バイト増加するため、これをメンテナンスするにはそれだけ多くの記憶域が必要です。

OID 索引検索を迂回すると、アプリケーションでの REF を使用した操作（ナビゲーショナル・アクセス）のパフォーマンスが向上します。実際のパフォーマンス向上率は、次の要因によって、アプリケーションごとに異なります。

- OID 索引の大きさ
- OID 索引がバッファ・キャッシュにキャッシュされているかどうか
- アプリケーションが実行する REF を使用した操作数

WITH ROWID オプションを使用する場合、REF 内の OID で行オブジェクトの OID がチェックされるため、このオプションは単なるヒントとなります。2 つの OID が一致しない場合は、REF 内 OID ではなく OID 索引が使用されます。ROWID ヒントは、有効範囲付き REF、参照整合性制約付き REF および主キー・ベースの REF に対しては利用できません。

ネストを解除する問合せを使用したリレーショナル形式でのオブジェクト・データの表示

コレクションに対してネストを解除する問合せを実行することで、データをフラットな（リレーショナルな）形式で表示できます。ネストを解除する問合せは、ネストした表および VARRAY の両方で実行できます。この項では、ネストを解除する問合せの例を示します。

ネストした表は、次の例のように、TABLE 構文を使用して問合せに対してネストを解除できます。

```
SELECT p.name_obj, n.num
   FROM people_reltab p, TABLE(p.phones_ntab) n ;
```

ここで、phones_ntab は、ネストした表 phones_ntab の属性を指定します。子である行（この例では電話番号）を持たない親である行も確実に取り出されるようにするには、「+」を使用して外部結合構文を使用します。たとえば次のようになります。

```
SELECT p.name_obj, n.num
   FROM people_reltab p, TABLE(p.phones_ntab) (+) n ;
```

問合せの SELECT 構文のリストが、親表のネストした表の列以外の列を参照しない場合は、ネストした表の記憶表に対してのみ実行されるように、問合せの最適化が実行されます。

ネストを解除する問合せの構文は、VARRAY およびネストした表のどちらの場合も同じです。たとえば、ネストした表 phones_ntab ではなく、phones_var という名前の VARRAY の場合を考えてみます。次の例は、TABLE 構文を使用して VARRAY を問い合せます。

```
SELECT p.name_obj, n.num
   FROM people_reltab p, TABLE(p.phones_var) n ;
```

次の例は、ネストした表のマルチレベル・コレクションに対してネストを解除する問合せです。それぞれの星に対して惑星のネストした表があり、各惑星に対して衛星のネストした表がある場合、問合せは、すべての衛星の名前を戻します。

```
CREATE TYPE satellite_t AS OBJECT (
  name        VARCHAR2(20),
  diameter    NUMBER);

CREATE TYPE nt_sat_t AS TABLE OF satellite_t;

CREATE TYPE planet_t AS OBJECT (
  name        VARCHAR2(20),
  mass        NUMBER,
  satellites  nt_sat_t);

CREATE TYPE nt_pl_t AS TABLE OF planet_t;

CREATE TABLE stars (
  name     VARCHAR2(20),
  age      NUMBER,
  planets  nt_pl_t)
NESTED TABLE planets STORE AS planets_tab
  (NESTED TABLE satellites STORE AS satellites_tab);

SELECT t.name FROM stars s, TABLE(s.planets) p, TABLE(p.satellites) t;
```

実表 stars の列は SELECT 構文のリストには表示されないため、問合せは、直接 satellites 記憶表に対して実行されるように最適化されます。

マルチレベル・コレクションの問合せでは、外部結合構文も使用できます。

ネストを解除する問合せでのプロシージャおよびファンクションの使用

ネストを解除する問合せを実行するためのプロシージャおよびファンクションを作成できます。たとえば、location が「home」である電話番号のみを戻す、home_phones() というファンクションを作成できます。home_phones() ファンクションを作成するには、次の例のようなコードを入力します。

```
CREATE OR REPLACE FUNCTION home_phones(allphones IN phone_ntabtyp)
        RETURN phone_ntabtyp IS
```

```
   homephones phone_ntabtyp := phone_ntabtyp();
   indx1      number;
   indx2      number := 0;
BEGIN
   FOR indx1 IN 1..allphones.count LOOP
      IF
         allphones(indx1).location = 'home'
      THEN
         homephones.extend;    -- extend the local collection
         indx2 := indx2 + 1;
         homephones(indx2) := allphones(indx1);
      END IF;
   END LOOP;

   RETURN homephones;
END;
/
```

ここで、人のリストおよびその自宅の電話番号を問い合せるには、次のように入力します。

```
SELECT p.name_obj, n.num
   FROM people_reltab p, TABLE(
      CAST(home_phones(p.phones_ntab) AS phone_ntabtyp)) n ;
```

自宅の電話番号がリストされていない人も含めて、人のリストおよびその自宅の電話番号を問い合せるには、次のように入力します。

```
SELECT p.name_obj, n.num
   FROM people_reltab p,
       TABLE(CAST(home_phones(p.phones_ntab) AS phone_ntabtyp))(+) n ;
```

> **参照：** TABLE 構文の使用方法の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

VARRAY の記憶域上の考慮点

格納される VARRAY のサイズは、VARRAY 内の現行の要素数にのみ依存し、保持できる最大要素数には関係しません。VARRAY の記憶域には、NULL 情報など、わずかなオーバーヘッドがかかります。したがって、格納される VARRAY のサイズは、（要素のサイズ）×（要素数）より少し大きくなります。

VARRAY は、列値または BLOB として格納されます。VARRAY の定義時に、宣言された VARRAY の LIMIT（要素の上限値）を使用して計算される VARRAY の最大許容サイズに基づいて、VARRAY の格納方法が決定されます。サイズが約 4000 バイトを超える場合、VARRAY は BLOB に格納されます。その他の場合は、その列自体に列値として格納されます。さらに、Oracle ではインライン LOB がサポートされています。したがって、（LOB ロ

ケータ用に何バイトか予約された）大きな VARRAY の最初の 4000 バイトに相当する要素は、その行自体の列に格納されます。

VARRAY およびネストした表のパフォーマンス

アプリケーションでコレクション全体が 1 つの単位として操作される場合、ネストした表 VARRAY はまとまって格納され、ネストした表とは異なり、データを取り出すための結合は必要ありません。

VARRAY の問合せ

ネストを解除する構文を、ネストした表へのアクセスの場合と同様の方法で、VARRAY 列へのアクセスにも使用できます。

> **参照：** 詳細は、8-12 ページの「ネストを解除する問合せを使用したリレーショナル形式でのオブジェクト・データの表示」を参照してください。

VARRAY の更新

VARRAY の要素単位の更新は、サポートされていません。このため、VARRAY が更新される場合、古いコレクション全体が新しいコレクションで置き換えられます。

ネストした表

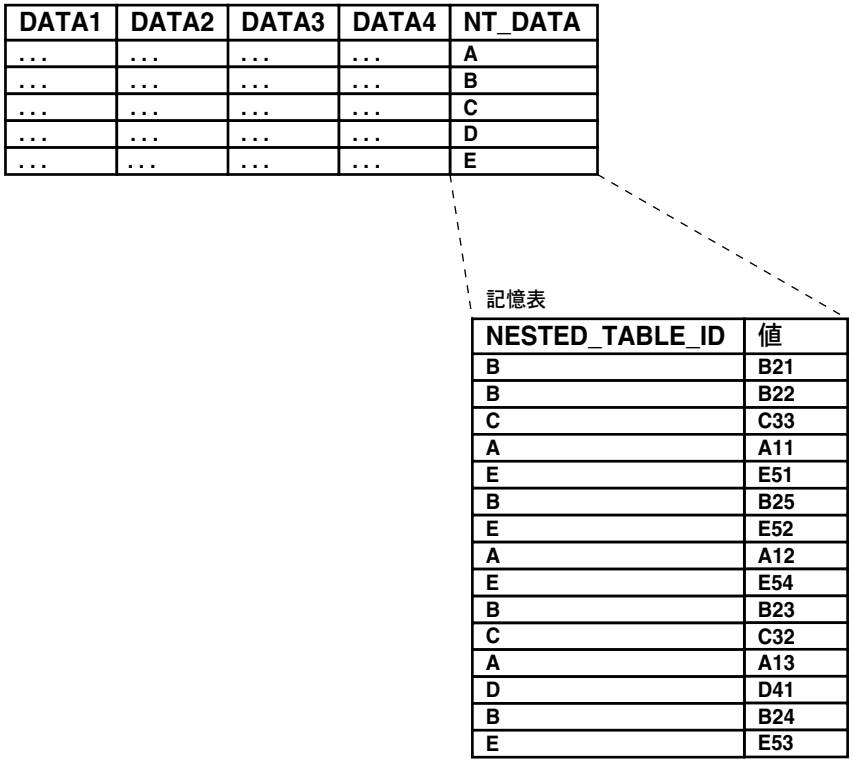
次の項では、ネストした表を使用するための設計上の考慮点について説明します。

ネストした表の記憶域

Oracle では、ネストした表の行は別の記憶表に格納されます。システムが生成する NESTED_TABLE_ID（長さ 16 バイト）によって、親である行およびそれに対応する記憶表の行が関連付けられます。

図 8-5 に、記憶表の動作を示します。記憶表には、ネストした表列内のネストした表ごとの値が含まれています。この個々の値が、記憶表の 1 行に相当します。記憶表は、NESTED_TABLE_ID を使用して、個々の値に対するネストした表を追跡します。したがって、図 8-5 では、ネストした表 A に属するすべての値が識別され、それに続いてネストした表 B に属するすべての値も同様に識別されます。

図 8-5 ネストした表の記憶域

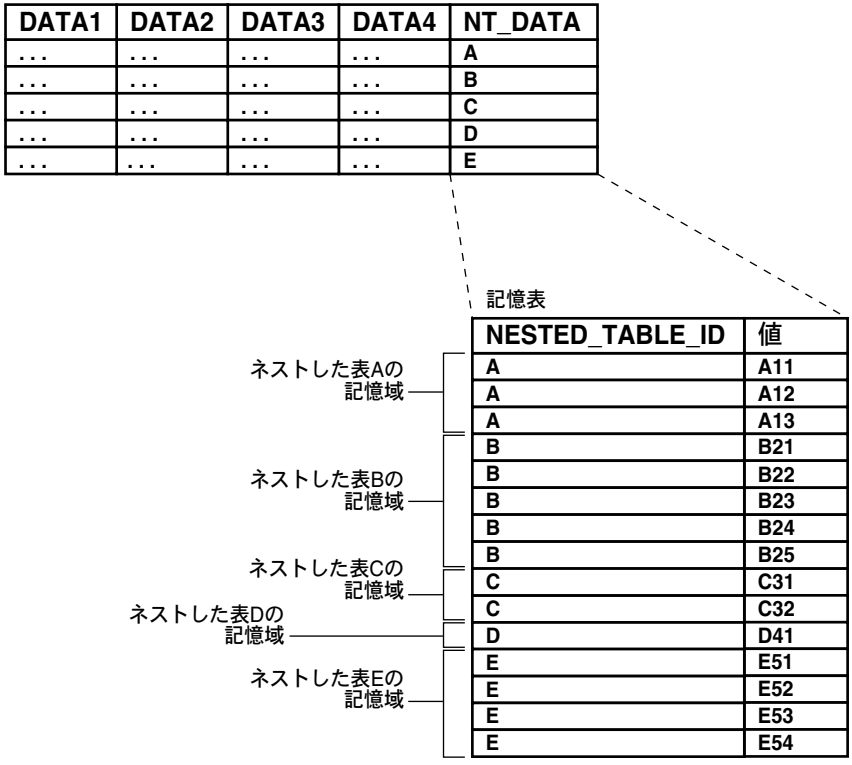


索引構成表（IOT）に格納されたネストした表

ネストした表が主キーを持つ場合、この表を IOT として構成できます。NESTED_TABLE_ID 列が、指定された親である行に対する主キーの第一要素（プレフィックス）である場合、その子である行は物理的に 1 つにクラスタ化されます。このため、親である行がアクセスされるときに、そのすべての子である行を効率的に取り出せます。親である行のみがアクセスされる場合も、子である行が親である行に混入することはないため、同等の効率が維持されます。

図 8-6 に、ネストした表が IOT である場合の記憶表の動作を示します。記憶表では、ネストした表の列内にあるネストした表の値は、NESTED_TABLE_ID ごとにグループ化されます。図 8-6 では、記憶表のデータは、親表の NT_DATA 列にあるネストした表ごとにグループ化されています。つまり、ネストした表 A のすべての値がグループ化され、ネストした表 B のすべての値も同様にグループ化され、他のネストした表についても同様の動作となります。

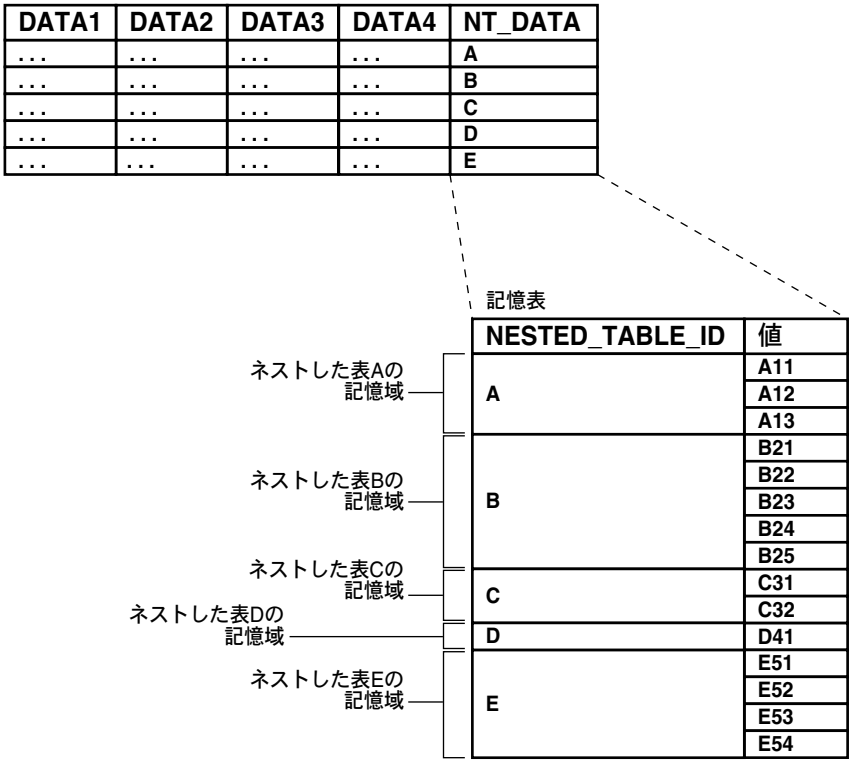
図 8-6 IOT 記憶域のネストした表



さらに、COMPRESS 句によって、IOT 列に対してプレフィックスを圧縮することが可能です。この圧縮では、それぞれの子である行にある親のキーの重複部分を取り除きます。つまり、親キーが個々の子である行で繰り返されることがないため、記憶域を大幅に節約することができます。

つまり、COMPRESS 句を使用してネストした表の圧縮を指定すると、記憶表に必要な領域の合計サイズが削減されます。これは、同じグループの各値に対して同一の NESTED_TABLE_ID が繰り返されないためです。かわりに、図 8-7 に示すように、NESTED_TABLE_ID はグループにつき 1 回のみ格納されます。

図 8-7 IOT 記憶域に圧縮して格納されたネストした表



通常は、一般に、NESTED_TABLE_ID 列を主キーのプレフィックスとして使用して、ネストした表を IOT に格納することをお薦めします。さらに、IOT でプレフィックス圧縮を有効にする必要があります。ただし、通常の業務としてはネストした表を 1 つの単位として取り出すことがなく、子である行をクラスタ化することもない場合は、ネストした表を IOT に格納しないでください。また、圧縮も指定しないでください。

ネストした表の索引

（IOT でない）表に格納されたネストした表に対しては、記憶表の NESTED_TABLE_ID 列に索引を作成する必要があります。親表の対応する ID 列の索引は、表の作成時に自動的に作成されます。NESTED_TABLE_ID 列に索引を作成することによって、ネストした表の子である行にさらに効率的にアクセスできるようになります。これは、親表とネストした表の結合が実行される際は、必ず NESTED_TABLE_ID 列が使用されるためです。

ネストした表のロケータ

大規模な子集団の場合は、必要に応じて子である行にアクセスできるように、親である行、および子集団に対するロケータを戻せます。子集団のフィルタもできます。ネストした表のロケータを使用すると、各親に対して子である行が不必要に転送されることを回避できます。

次のいずれかのアクションを実行して、ネストした表のロケータを使用して子である行にアクセスできます。

- OCI コレクション関数をコールします。このアクションは、OCIColl* 関数などのクライアント側コードでコレクション要素にアクセスするときに暗黙的に発生します。最初のアクセスで、コレクション全体が暗黙的に取り出されます。

  参照： OCI コレクション関数の詳細は、『Oracle Call Interface プログラマーズ・ガイド』を参照してください。

- SQL を使用して、ネストした表に対応する行を取り出します。このアクションについては、9-23 ページの「オブジェクト表 PurchaseOrder_objtab」を参照してください。

マルチレベル・コレクションでは、任意のネスト・レベルで指定されたコレクションを持つロケータを使用できます。コレクションをロケータとして取り出すかどうか指定する方法は、次の 2 通りです。

表作成時に指定する方法

コレクション型が列型として使用されていて、NESTED TABLE 記憶域句が使用されている場合は、RETURN LOCATOR 句を使用して、特定のコレクションをロケータとして取り出すように指定できます。

たとえば、third_level が 3 つのレベルのネストした表で構成されたコレクション型だとします。次の例では、RETURN LOCATOR 句によって、常に、第 2、すなわち中間のレベルのネストした表がロケータとして取り出されることが指定されています。

```
CREATE TABLE tab1 (
  a  NUMBER,
  b  third_level)
NESTED TABLE b STORE AS b_ntab
  (NESTED TABLE COLUMN_VALUE STORE AS cv1_ntab RETURN LOCATOR
    (NESTED TABLE COLUMN_VALUE STORE AS cv2_ntab ));
```

取出し処理中にヒントとして指定する方法

問合せによって、ヒント NESTED_TABLE_GET_REFS を使用してロケータとして取り出せます。次の例では、表 tab1 から列 b をロケータとして取り出します。

```
SELECT /*+ NESTED_TABLE_GET_REFS +*/ b
FROM tab1
WHERE a = 2;
```

ただし、RETURN LOCATOR 句の場合とは異なり、ヒントを使用する場合は、特定の内部コレクションをロケータとして戻す指定はできません。

セット・メンバーシップ問合せの最適化

ネストした表にある特定の項目を検索する場合、セット・メンバーシップ問合せが役立ちます。たとえば、次の問合せでは、home というロケーションが、子集団のメンバーシップ、具体的には、ネストした表 phones_ntab（親表 people_reltab にあります）にあるかどうかをテストします。

```
SELECT * FROM people_reltab p
   WHERE 'home' IN (SELECT location FROM TABLE(p.phones_ntab)) ;
```

Oracle では、子集団のメンバーシップを内部的にセミ結合に変換することによって、そのメンバーシップをテストする問合せをより効率的に実行しています。ただし、この最適化は、ALWAYS_SEMI_JOIN 初期化パラメータが設定されている場合にのみ実行されます。セミ結合を実行する場合、このパラメータの有効な値は MERGE および HASH です。これらのパラメータ値によって、使用すべき結合メソッドが示されます。

注意： 前述の例では、home および location は子集団要素です。子集団要素がオブジェクト型である場合は、セット・メンバーシップ問合せを実行するマップ・メソッドまたはオーダー・メソッドが必要です。

ネストした表での DML 操作

ネストした表に対しては DML 操作を実行できます。適切な SQL コマンドを使用することによって、ネストした表に対する行の挿入、行の削除、および既存の行の更新が可能になります。これらの操作では、ネストした表は TABLE 副問合せによって識別されます。次の例では、ネストした表 phones_ntab への電話番号の挿入も含めて、表 people_reltab に新しい人物についての値を挿入します。

```
INSERT INTO people_reltab values (
   0001,
   name_objtyp(
      'john', 'william', 'foster'),
   address_objtyp(
      '111 Maple Road', 'Fairfax', 'VA', '22033'),
   phone_ntabtyp(
      phone_objtyp('home', '650.331.1222'),
      phone_objtyp('work', '650.945.4389'))) ;
```

次の例では、表 people_reltab にある、識別番号が 0001 の既存の人物に対して、ネストした表 phones_ntab に電話番号を挿入します。

```
INSERT INTO TABLE(SELECT p.phones_ntab FROM people_reltab p WHERE p.id = '0001')
   VALUES ('cell', '650.331.9337') ;
```

特定のネストした表を削除するには、次の例のように、親である行でそのネストした表列を NULL に設定します。

```
UPDATE people_reltab SET phones_ntab = NULL WHERE id = '0001' ;
```

削除したネストした表には、再作成するまで値を挿入できません。識別番号が 0001 である人物に対して、ネストした表の列オブジェクト phones_ntab にネストした表を再作成するには、次の SQL 文を入力します。

```
UPDATE people_reltab SET phones_ntab = phone_ntabtyp() WHERE id = '0001' ;
```

ネストした表を再作成しながら値を挿入することもできます。

```
UPDATE people_reltab
   SET phones_ntab = phone_ntabtyp(phone_objtyp('home', '650.331.1222'))
   WHERE id = '0001' ;
```

ネストした表に対して DML 操作を実行すると親である行はロックされます。したがって、特定のネストした表のデータへの変更は、一度に 1 つのみです。ネストした表のある行に対する変更と他の行に対する変更を同時に処理することはできません。ただし、同時変更をサポートする必要があるのはネストした表の一部のデータのみで、ネストした表の他のデータではこのサポートが必要でない場合は、同時変更の必要なデータへの REF の使用を検討する必要があります。

たとえば、発注書を処理するアプリケーションを使用している場合、発注書に顧客情報および明細項目を含める場合があります。この場合、顧客情報が頻繁に変更されることはないため、このデータに対して同時変更をサポートする必要はありません。一方、明細項目は頻繁に変更される可能性があります。同じ発注書にある明細項目について同時更新をサポートするには、この明細項目を別のオブジェクト表に格納して、これをネストした表に格納した REF で参照します。

マルチレベル・コレクション

第 2 章では、コレクション型をネストして、真のマルチレベル・コレクション（ネストした表のネストした表、VARRAY のネストした表、ネストした表の VARRAY、コレクション型の属性を持つオブジェクト型の VARRAY またはネストした表）を作成する方法について説明しています。

REF を使用して間接的にコレクションをネストすることもできます。たとえば、ネストした表属性または VARRAY の属性を持つオブジェクトを参照する属性を持つオブジェクト型のネストした表を作成できます。実際にはマルチレベル・コレクションのすべての要素にアクセスする必要はない、という場合は、REF を使用してコレクションをネストすると、パフォーマンスが向上する可能性があります。ロードする必要があるのは、要素そのものではなく REF のみです。

真のマルチレベル・コレクション（具体的にはマルチレベルのネストした表）を使用すると、コレクションの個々の要素にアクセスする問合せのパフォーマンスが向上します。すべての要素にアクセスする必要はない場合、ネストした表ロケータを使用すると、プログラムで規定されたアクセスのパフォーマンスが向上する可能性があります。

REF を使用して別のコレクションをネストする例として、8-3 ページの「列オブジェクトの記憶域」で説明されている name_objtyp、address_objtyp および phone_ntabtyp の各オブジェクト型を使用して、person_objtyp と呼ばれる新しいオブジェクト型を作成するとします。1 人が複数の電話番号を持つことがあるため、phone_ntabtyp オブジェクト型がネストした表になっていることに注意してください。

オブジェクト型 person_objtyp を作成するには、次の SQL 文を実行します。

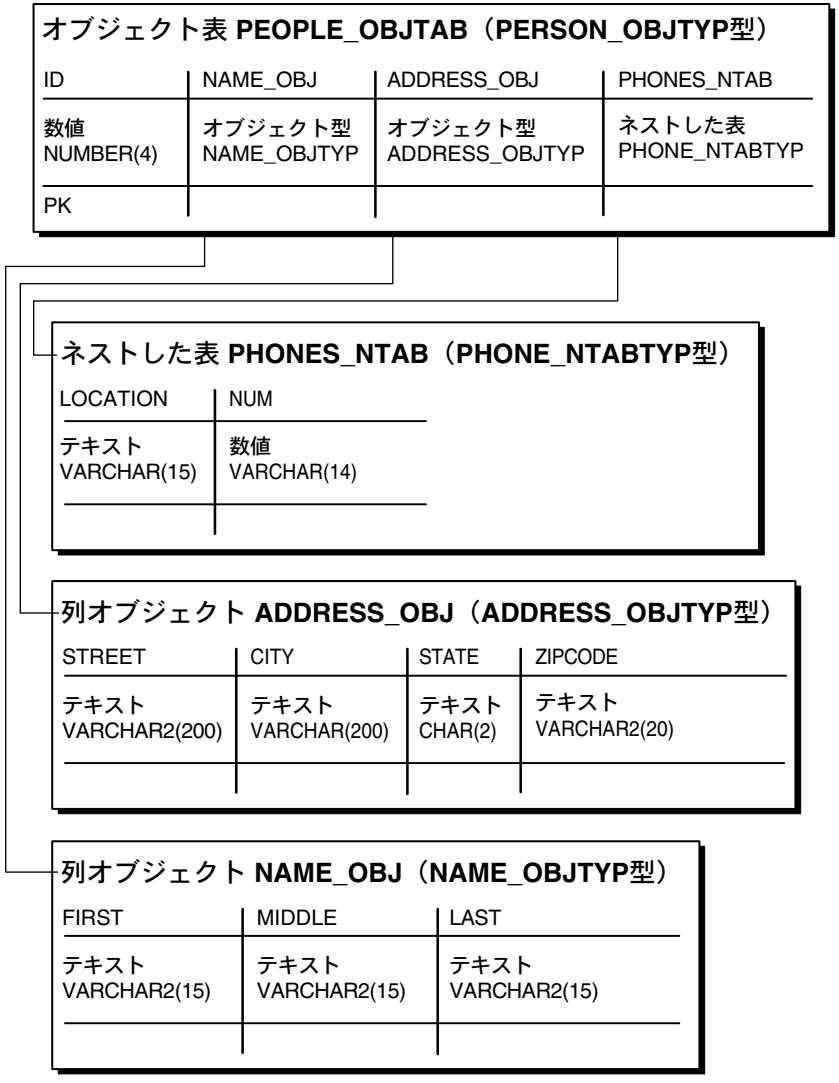
```
CREATE TYPE person_objtyp AS OBJECT (
   id            NUMBER(4),
   name_obj      name_objtyp,
   address_obj   address_objtyp,
   phones_ntab   phone_ntabtyp);
```

person_objtyp オブジェクト型の people_objtab というオブジェクト表を作成するには、次の SQL 文を実行します。

```
CREATE TABLE people_objtab OF person_objtyp (id PRIMARY KEY)
   NESTED TABLE phones_ntab STORE AS phones_store_ntab ;
```

people_objtab 表は、8-3 ページの「列オブジェクトの記憶域」で説明している表 people_reltab と同じ属性を持ちます。違いは、people_objtab は行オブジェクトを持つオブジェクト表であり、people_reltab 表は 3 つの列オブジェクトを持つリレーショナル表であるという点です。

図 8-8 people_objtab オブジェクト表のオブジェクト・リレーショナル表現



ここで、people_objtab オブジェクト表にある行オブジェクトを他の表から参照できます。たとえば、次の項目が含まれる projects_objtab 表を作成するとします。

- 各プロジェクトのプロジェクト識別番号
- 各プロジェクトのタイトル
- 各プロジェクトのプロジェクト・リード
- 各プロジェクトの説明
- 各プロジェクトに割り当てられたチームのメンバーを格納するネストした表コレクション

プロジェクト・リードに対しては people_objtab への REF を使用でき、チームに対しては REF のネストした表コレクションを使用できます。まず、person_objtyp オブジェクト型に基づいて、personref_ntabtyp というネストした表オブジェクト型を作成します。

```
CREATE TYPE personref_ntabtyp AS TABLE OF REF person_objtyp;
```

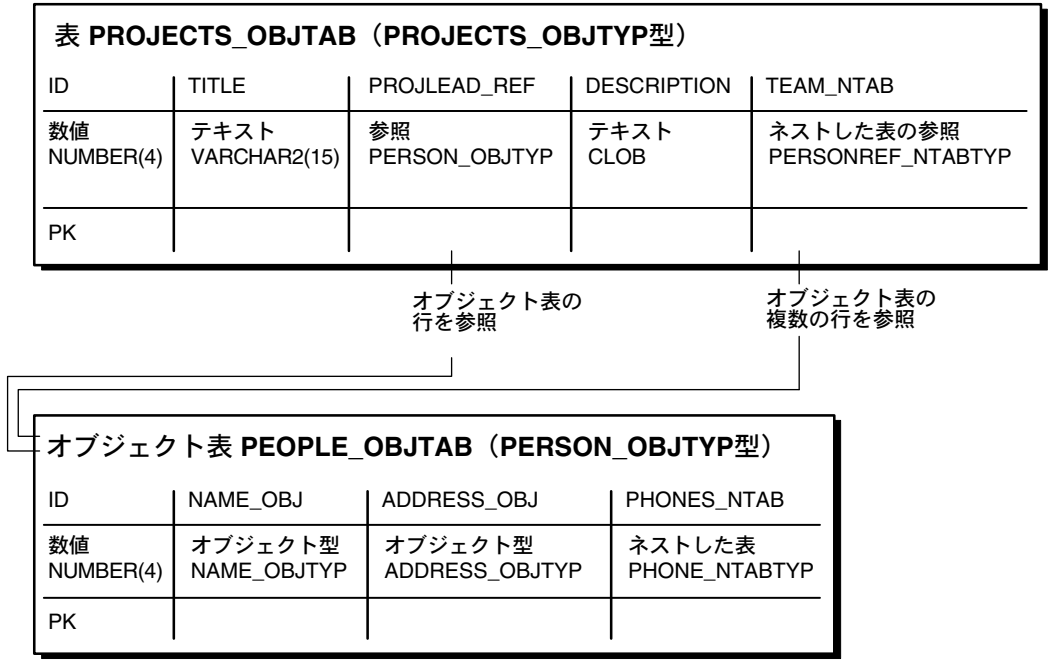
これで、オブジェクト表 projects_objtab を作成する準備ができました。まず、次の SQL 文を実行して、オブジェクト型 projects_objtyp を作成します。

```
CREATE TYPE projects_objtyp AS OBJECT (
   id              NUMBER(4),
   title           VARCHAR2(15),
   projlead_ref    REF person_objtyp,
   description     CLOB,
   team_ntab       personref_ntabtyp);
```

次に、projects_objtyp に基づいて、オブジェクト表 projects_objtab を作成します。

```
CREATE TABLE projects_objtab OF projects_objtyp (id PRIMARY KEY)
   NESTED TABLE team_ntab STORE AS team_store_ntab ;
```

図 8-9 projects_objtab オブジェクト表のオブジェクト・リレーショナル表現



people_objtab オブジェクト表および projects_objtab オブジェクト表が作成されると、間接的にネストしたコレクションを持つことになります。つまり、projects_objtab 表には、people_objtab 表にある人物を指す REF のネストした表コレクションが含まれ、people_objtab 表にある人物には電話番号のネストした表コレクションがあります。

次のようにして、people_objtab 表に値を挿入できます。

```
INSERT INTO people_objtab VALUES (
   0001,
   name_objtyp('JOHN', 'JACOB', 'SCHMIDT'),
   address_objtyp('1252 Maple Road', 'Fairfax', 'VA', '22033'),
   phone_ntabtyp(
      phone_objtyp('home', '650.339.9922'),
      phone_objtyp('work', '510.563.8792'))) ;

INSERT INTO people_objtab VALUES (
   0002,
   name_objtyp('MARY', 'ELLEN', 'MILLER'),
   address_objtyp('33 Spruce Street', 'McKees Rocks', 'PA', '15136'),
   phone_ntabtyp(
```

```
      phone_objtyp('home', '415.642.6722'),
      phone_objtyp('work', '650.891.7766'))) ;

INSERT INTO people_objtab VALUES (
   0003,
   name_objtyp('SARAH', 'MARIE', 'SINGER'),
   address_objtyp('525 Pine Avenue', 'San Mateo', 'CA', '94403'),
   phone_ntabtyp(
      phone_objtyp('home', '510.804.4378'),
      phone_objtyp('work', '650.345.9232'),
      phone_objtyp('cell', '650.854.9233'))) ;
```

さらに、次の例のように、REF 演算子を使用して people_objtab オブジェクト表から選択することによって、projects_objtab リレーショナル表に挿入できます。

```
INSERT INTO projects_objtab VALUES (
   1101,
   'Demo Product',
   (SELECT REF(p) FROM people_objtab p WHERE id = 0001),
   'Demo the product, show all the great features.',
   personref_ntabtyp(
      (SELECT REF(p) FROM people_objtab p WHERE id = 0001),
      (SELECT REF(p) FROM people_objtab p WHERE id = 0002),
      (SELECT REF(p) FROM people_objtab p WHERE id = 0003))) ;

INSERT INTO projects_objtab VALUES (
   1102,
   'Create PRODDB',
   (SELECT REF(p) FROM people_objtab p WHERE id = 0002),
   'Create a database of our products.',
   personref_ntabtyp(
      (SELECT REF(p) FROM people_objtab p WHERE id = 0002),
      (SELECT REF(p) FROM people_objtab p WHERE id = 0003))) ;
```

注意： この例では、REF を格納するためにネストした表を使用していますが、REF を VARRAY に格納することもできます。つまり、REF の VARRAY を作成できます。

メソッド・ファンクションに対する言語の選択

メソッド・ファンクションは、Oracle でサポートされる任意の言語（PL/SQL、Java、C など）で実装できます。特定のアプリケーション用に言語を選択する場合は、次の要因を考慮してください。

- 使用しやすさ
- SQL コール
- 実行速度
- 同一 / 異なるアドレス空間

一般に、アプリケーションで主に計算を実行する場合には、C が適していますが、比較的多くのデータベース・コールを実行する場合は、PL/SQL または Java が適しています。

C で実装されるメソッドは、外部プロシージャを使用して、サーバーとは別のプロセスで実行されます。それに対して、Java または PL/SQL で実装されるメソッドは、サーバーと同じプロセスで実行されます。

メソッドの実装例

この項で説明する例では、異なる言語で実装されたメソッドを持つオブジェクト型が関係しています。この例では、オブジェクト型 `ImageType` は、`ID` 属性（一意に識別される `NUMBER`）および `IMG` 属性（イメージを格納する `BLOB`）を持ちます。オブジェクト型 `ImageType` は、次のメソッドを持ちます。

- `get_name()` メソッド。データベース内でイメージの名前を検索してフェッチします。このメソッドは、PL/SQL で実装されています。
- `rotate()` メソッド。イメージを回転します。このメソッドは、C で実装されています。
- `clear()` メソッド。指定された色の新しいイメージを戻します。このメソッドは、Java で実装されています。

C でメソッドを実装する場合は、外部 C ルーチンが含まれるライブラリを指す、`LIBRARY` オブジェクトを定義する必要があります。Java でメソッドを実装する場合を想定して、この例では、メソッドを持つ Java クラスがコンパイルされ、Oracle にアップロードされていると仮定しています。

オブジェクト型の定義およびそのメソッドは次のとおりです。

```
CREATE TYPE ImageType AS OBJECT (
   id   NUMBER,
   img  BLOB,
   MEMBER FUNCTION get_name() return VARCHAR2,
   MEMBER FUNCTION rotate() return BLOB,
   STATIC FUNCTION clear(color NUMBER) return BLOB
   );
```

```
CREATE TYPE BODY ImageType AS
   MEMBER FUNCTION get_name() RETURN VARCHAR2
   AS
   imgname VARCHAR2(100);
   BEGIN
      SELECT name INTO imgname FROM imgtab WHERE imgid = id;
      RETURN imgname;
   END;

   MEMBER FUNCTION rotate() RETURN BLOB
   AS LANGUAGE C
   NAME "Crotate"
   LIBRARY myCfuncs;

   STATIC FUNCTION clear(color NUMBER) RETURN BLOB
   AS LANGUAGE JAVA
   NAME 'myJavaClass.clear(color oracle.sql.NUMBER) RETURN oracle.sql.BLOB';

END;
/
```

制限事項： タイプ・メソッドは、Java のスタティック・メソッドのみにマップできます。

参照：

- 詳細は、『Oracle9*i* Java Stored Procedures Developer's Guide』を参照してください。
- 言語の選択の詳細は、第 3 章「Oracle プログラム環境のオブジェクト・サポート」を参照してください。

スタティック・メソッド

スタティック・メソッドは、SELF 値が第 1 パラメータとして渡されない点で、メンバー・メソッドと異なります。SELF の値が問題ではないメソッドは、スタティック・メソッドとして実装する必要があります。スタティック・メソッドは、ユーザー定義コンストラクタに使用できます。

次の例では、コンストラクタに類似したメソッドで、明示的な入力パラメータに基づいてその型のインスタンスを作成し、指定された表にそのインスタンスを挿入します。

```
CREATE OR REPLACE TYPE atype AS OBJECT(a1 NUMBER,
   STATIC PROCEDURE newa (
```

```
      p1        NUMBER,
      tabname   VARCHAR2,
      schname   VARCHAR2));

CREATE OR REPLACE TYPE BODY atype AS
    STATIC PROCEDURE newa (p1 NUMBER, tabname VARCHAR2, schname VARCHAR2)
      IS
      sqlstmt VARCHAR2(100);
    BEGIN
      sqlstmt := 'INSERT INTO '||schname||'.'||tabname|| ' VALUES (atype(:1))';
      EXECUTE IMMEDIATE sqlstmt USING p1;
    END;
END;
/

CREATE TABLE atab OF atype;
   BEGIN
     atype.newa(1, 'atab', 'scott');
   END;
```

実行者権限を使用した再利用コードの作成

任意のスキーマで使用できる汎用オブジェクト型を作成するには、CREATE OR REPLACE TYPE の AUTHID CURRENT_USER オプションを介して、実行者権限を使用する型を定義する必要があります。一般に、次の条件がいずれも真である場合に実行者権限を使用します。

- データにアクセスし操作するタイプ・メソッドがある。
- これらのタイプ・メソッドを定義していないユーザーによって使用される必要がある。

たとえば、SCOTT によって 8-28 ページの「スタティック・メソッド」で作成された型 atype に対する実行権限を、ユーザー SARA に付与し、その後この型に基づいて表 atab を作成できます。

```
GRANT EXECUTE ON atype TO SARA ;
CONNECT SARA/TPK101 ;
CREATE TABLE atab OF scott.atype ;
```

ここで、ユーザー SARA が、次の文で atype を使用するとします。

```
BEGIN
  scott.atype.newa(1, 'atab', 'SARA'); -- raises an error
END;
/
```

型定義者（SCOTT）には、newa プロシージャで挿入を実行するために必要な権限がないため、この文を実行するとエラーが戻されます。このエラーは、実行者権限を使用して atype を定義することによって回避できます。ここで、まず両方のスキーマの atab 表を削除し、実行者権限を使用して atype を再作成します。

```
DROP TABLE atab ;
CONNECT SCOTT/TIGER ;
DROP TABLE atab ;

CREATE OR REPLACE TYPE atype AUTHID CURRENT_USER AS OBJECT(a1 NUMBER,
   STATIC PROCEDURE newa(p1 NUMBER, tabname VARCHAR2, schname VARCHAR2));

CREATE OR REPLACE TYPE BODY atype AS
  STATIC PROCEDURE newa(p1 NUMBER, tabname VARCHAR2, schname VARCHAR2)
   IS
     sqlstmt VARCHAR2(100);
   BEGIN
      sqlstmt := 'INSERT INTO '||schname||'.'||tabname|| ' VALUES
         (scott.atype(:1))';
      EXECUTE IMMEDIATE sqlstmt USING p1;
   END;
END;
/
```

これで、ユーザー SARA が再度 atype を使用しようとした場合、文は正常に実行されます。

```
GRANT EXECUTE ON atype TO SARA ;
CONNECT SARA/TPK101 ;
CREATE TABLE atab OF scott.atype;

BEGIN
  scott.atype.newa(1, 'atab', 'SARA'); -- executes successfully
END;
/
```

このとき文が正常に実行されるのは、プロシージャは定義者（SCOTT）権限でなく実行者（SARA）権限で実行されるためです。

型階層内では、サブタイプには、そのすぐ上のスーパータイプと同じ権限モデルが存在します。つまり、サブタイプはスーパータイプの権限を暗黙的に継承するため、明示的な権限の指定はできません。さらに、スーパータイプが定義者権限によって宣言された場合は、サブタイプはスーパータイプと同じスキーマ内に配置されている必要があります。これらの規則によって、実行者権限型の階層がスキーマを超えて展開されることになります。ただし、定義者権限モデルを使用する型の階層は、単一のスキーマ内に存在する必要があります。

次に例を示します。

```
CREATE TYPE deftype1 AS OBJECT (...); -- Definer rights type
CREATE TYPE subtype1 UNDER deftype1(...); -- subtype in same schema as supertype
```

```
CREATE TYPE schema2.subtype2 UNDER deftype1(...); -- ERROR

CREATE TYPE invtype1 AUTHID CURRENT_USER AS OBJECT (...); -- Invoker rights type
CREATE TYPE schema2.subtype2 UNDER invtype1 (...); -- LEGAL
```

タイプ・メソッドの戻り値に基づくファンクション索引

ファンクション索引は、式またはファンクションの戻り値に基づいた索引です。このファンクションは、オブジェクト型のメソッド・ファンクションの場合もあります。

メソッド・ファンクションに基づくファンクション索引は、索引付けの対象となっている列または表の各オブジェクト・インスタンスに対するファンクションの戻り値をあらかじめ計算しておいて、それらの値を索引に格納します。この索引では、ファンクションを評価しなおすことなく列や表が参照されます。

ファンクション索引は、WHERE 句にファンクションを含む問合せのパフォーマンスを向上させる場合に役立ちます。たとえば、次のコードにはオブジェクト表 emps の問合せが含まれています。

```
CREATE TYPE emp_t AS OBJECT
(
  name   VARCHAR2
  salary NUMBER,
  MEMBER FUNCTION bonus RETURN NUMBER DETERMINISTIC
);

CREATE OR REPLACE TYPE BODY emp_t IS
 MEMBER FUNCTION bonus RETURN NUMBER IS
 BEGIN
  RETURN self.salary * .1;
 END;
END;

CREATE TABLE emps OF emp_t ;

SELECT e
  FROM emps
  WHERE e.bonus() > 2000 ;
```

この問合せを評価するためには、この表の各行オブジェクトに対する bonus() を評価する必要があります。bonus() の戻り値に対してファンクション索引がある場合は、この評価の作業はすでに実行済なので、Oracle の動作としては、索引内にある問合せ結果を検索するのみで済みます。これによって、Oracle が問合せを実行した結果を戻す速度が向上します。

ファンクションの戻り値の索引付けが効果的に実行できるのは、戻り値が一定である場合、つまり、ファンクションが各オブジェクト・インスタンスに対して常に同じ値を戻す場合のみです。このため、ファンクション索引でユーザー定義ファンクションを使用する場合は、

前述の例のように、DETERMINISTIC キーワードを使用してそのファンクションを宣言しておく必要があります。このキーワードによって、各オブジェクト・インスタンスの入力引数値の集合に対して、このファンクションから常に一定の値が戻されることが保証されます。

次の例では、表 emps のメソッド bonus() に対するファンクション索引を作成します。

```
CREATE INDEX emps_bonus_idx ON emps x (x.bonus()) ;
```

参照： ファンクション索引の詳細は、『Oracle9*i* データベース概要』および『Oracle9*i* SQL リファレンス』を参照してください。

現行のオブジェクト形式への変換

リリース 8.1 以上で作成された表では、オブジェクトは、リリース 8.0 の形式と比較して記憶域が小さく、パフォーマンス特性の優れた新しい形式で格納されます。また、より効率的な転送プロトコルが使用されています。COMPATIBLE パラメータが 8.1.0 以上に設定された場合、新しく作成する表および列にあるすべてのオブジェクトは、自動的にリリース 8.1 の形式で格納されます。また、すべてのオブジェクト（新規または以前のもの）がリリース 8.1 の形式で転送されます。リリース 8.0 で作成された表では、オブジェクトは、明示的に変換されないかぎり、今後も、リリース 8.0 の形式で格納されます。

リリース 8.0 のデータベースで作成されたオブジェクトをリリース 8.1 で導入された形式に変換できます。手順は次のとおりです。

1. CREATE TABLE...AS SELECT... 文を使用して表を再作成します。
2. 表のデータをエクスポート / インポートします。

参照： 互換性および COMPATIBLE 初期化パラメータの詳細は、『Oracle9*i* データベース移行ガイド』を参照してください。

注意： リリース 8.0 の形式は今後のリリースでは使用できません。

オブジェクト表およびオブジェクト列のレプリケーション

オブジェクト表およびオブジェクト・ビューは、マテリアライズド・ビューとしてレプリケートできます。また、あるオブジェクト型、コレクション型または REF 型の列を含むリレーショナル表もレプリケートできます。このようなマテリアライズド・ビューは、オブジェクト・リレーショナル・マテリアライズド・ビューと呼ばれます。

オブジェクト・リレーショナル・マテリアライズト・ビューが必要とするすべてのユーザー定義型は、マスター・サイトおよびマテリアライズド・ビュー・サイトに存在する必要があります。そのようなユーザー定義型の ID およびバージョンは両方のサイトで一致している必要があります。

オブジェクト型、コレクション型または REF 型の列のレプリケーション

更新可能とするには、オブジェクト列を含む表に基づくマテリアライズド・ビューでは、そのビューを定義する問合せにある列をオブジェクトとして選択する必要があります。問合せが、その列のオブジェクト型の一定の属性のみを選択する場合は、マテリアライズド・ビューは読込み専用となります。

ビュー定義問合せは、コレクション型または REF 型の列も選択できます。REF は、主キー・ベースのキーか、システム生成のキーのいずれかで、有効範囲付きまたは有効範囲なしの場合があります。有効範囲付き REF の列は、元のリモート表ではなくマテリアライズド・ビュー・サイトの別の表（たとえば、マスター表のローカル・マテリアライズド・ビューなど）に、有効範囲を付けなおすことができます。

オブジェクト表のレプリケーション

オブジェクト表に基づくマテリアライズド・ビューは**オブジェクト・マテリアライズド・ビュー**と呼ばれます。このようなマテリアライズド・ビューは、それ自体がオブジェクト表です。オブジェクト・マテリアライズド・ビューは、OF <*type*> のキーワードを CREATE MATERIALIZED VIEW 文に追加することによって作成されます。たとえば次のようになります。

```
CREATE MATERIALIZED VIEW customer OF cust_objtyp
AS SELECT * FROM Scott.Customer_objtab@dbs1;
```

通常のオブジェクト表の場合と同様、オブジェクト・マテリアライズド・ビューの各行もオブジェクト・インスタンスです。したがって、マテリアライズド・ビューを作成するビュー定義問合せは、マスター表にあるオブジェクト全体を選択する必要があります。この問合せは、サブセットのみの選択はできません。たとえば、次のようなマテリアライズド・ビューは使用できません。

```
CREATE MATERIALIZED VIEW customer OF cust_objtyp
AS SELECT CustNo FROM Scott.Customer_objtab@dbs1;
```

OF <*type*> のキーワードを省略することによってオブジェクト表からオブジェクト・リレーショナル・マテリアライズド・ビューを作成できますが、そのようなビューは読込み専用です。オブジェクト表から、更新可能なオブジェクト・リレーショナル・マテリアライズド・ビューを作成できます。

たとえば、次の CREATE MATERIALIZED VIEW 文は、オブジェクト表の読込み専用オブジェクト・リレーショナル・マテリアライズド・ビューを作成します。ビュー定義問合せは、そのオブジェクト型のすべての列 / 属性を選択しますが、オブジェクトの属性として選択するわけではありません。したがって、作成されたビューは、オブジェクト・リレーショナル・ビューで、読込み専用です。

```
CREATE MATERIALIZED VIEW customer
AS SELECT * FROM Scott.Customer_objtab@dbs1;
```

オブジェクト表に基づいたオブジェクト・リレーショナル・ビューとマテリアライズド・ビューのいずれの場合も、マスターのオブジェクト表の型が FINAL でない場合は、マテリアライズド・ビューの定義問合せにある FROM 句に、ONLY キーワードが含まれている必要があります。たとえば次のようになります。

```
CREATE MATERIALIZED VIEW customer OF cust_objtyp
AS SELECT CustNo FROM ONLY Scott.Customer_objtab@dbs1;
```

そうでない場合は、FROM 句から ONLY キーワードを省略してください。

> **参照：** オブジェクト表とオブジェクト列のレプリケーションの詳細は、『Oracle9*i* アドバンスト・レプリケーション』を参照してください。

オブジェクトに対する制約

Oracle では、型指定内での制約およびデフォルトはサポートされません。ただし、表を作成するときに、制約およびデフォルトを指定できます。

```
CREATE OR REPLACE TYPE customer_type AS OBJECT(
   cust_id INTEGER);

CREATE OR REPLACE TYPE department_type AS OBJECT(
   deptno INTEGER);

CREATE TABLE customer_tab OF customer_type (
   cust_id default 1 NOT NULL);

CREATE TABLE department_tab OF department_type (
   deptno PRIMARY KEY);

CREATE TABLE customer_tab1 (
   cust customer_type DEFAULT customer_type(1)
   CHECK (cust.cust_id IS NOT NULL),
   some_other_column VARCHAR2(32));
```

型進化

次の項では、型進化に関連する設計上の考慮点について説明します。

型の変更のクライアントへの送信

ある型がサーバー側で進化すると、この型を使用するすべてのクライアント・アプリケーションで、この型に対応付けられている構造体の変更が必要になります。この変更は、OTT または JPublisher を使用して実行できます。また、構造体の変更に伴ってプログラムの変更が必要な場合もあります。このような変更を行った後、アプリケーションをコンパイルしなおし、再リンクする必要があります。

サード・パーティ・アプリケーションのリリース間で型が変更される場合があります。サード・パーティ・アプリケーションの最新リリースとの互換性を確立するために再コンパイルする必要があることをクライアント・アプリケーションに通知するには、compatibility_init ファンクションをクライアントからコールします。このファンクションは、クライアント・アプリケーションのどのリリースを使用しているかを示す文字列を入力として受け取ることができます。リリース文字列が最新バージョンと一致しない場合は、エラーが発生します。その場合、クライアント・アプリケーションは、最新リリースとの互換性を確立するための変更の一環として、リリース文字列を変更します。

たとえば次のようになります。

```
FUNCTION compatibility_init(rel IN VARCHAR2, errmsg OUT VARCHAR2)
RETURN NUMBER;
```

ここで、

rel は、その製品、たとえば、リリース 9.0.1 によって選択されたリリース文字列です。

errmsg は、戻す必要のある任意のエラー・メッセージです。

このファンクションは、成功すると 0（ゼロ）を戻し、エラーの場合は 0（ゼロ）以外の値を戻します。

デフォルトのコンストラクタの変更

型が変更されると、新しく追加された属性をパラメータ・リストに含めるためなどに、デフォルトでシステム定義されたコンストラクタを変更する必要があります。デフォルトのコンストラクタを使用している場合は、コールをコンパイルするためにプログラム内の起動を変更する必要があります。

システム定義されたデフォルトのコンストラクタではなく、独自でコンストラクタ・ファンクションを定義して使用する場合は、コンストラクタのコールを変更する必要はありません。

> **参照：** 6-20 ページの「ユーザー定義コンストラクタ」を参照してください。

型の FINAL プロパティの変更

型 T1 を、FINAL から NOT FINAL に変更した場合、クライアント・プログラムにある型 T1 の属性はすべて、インライン構造体から T1 へのポインタに変更されます。したがって、この属性がアクセスされた場合に参照解除を使用できるようにするには、プログラムを変更する必要があります。

逆に、ある型を、NOT FINAL から FINAL に変更するには、その型の属性をポインタからインライン構造体に変更する必要があります。

たとえば、型 T1 (a int) と T2 (b T1) があるとします。ここで、T1 のプロパティは FINAL です。T2 に対応する C/JAVA 構造体は T2 (T1 b) です。ところが、T1 のプロパティを NOT FINAL に変更すると、T2 の構造体は T2 (T1 *b) になります。

パフォーマンス・チューニング

アプリケーションのパフォーマンスの測定およびチューニングの詳細は、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。特に重要なパフォーマンス要因は、次のとおりです。

- 統計を収集する ANALYZE コマンド
- SQL コマンドの実行をプロファイルする tkprof
- 問合せ計画を生成する実行計画

Oracle オブジェクトでのパラレル問合せ

Oracle では、オブジェクトを使用してパラレル問合せを実行できます。ただし、次の制限があります。

- （ORDER BY、GROUP BY および SET 操作を使用して）結合およびソートをパラレルに実行する問合せを行うには、MAP ファンクションが必要です。MAP ファンクションがない場合、問合せは自動的にシリアルになります。
- ネストした表でのパラレル問合せはサポートされません。表に対するパラレル・ヒントまたはパラレル属性がある場合でも、問合せはシリアルになります。
- パラレル DML およびパラレル DDL は、オブジェクトではサポートされません。DML および DDL は、常にシリアルに実行されます。

ヒントおよび技法

次の項では、Oracle のオブジェクト型を使用した作業の様々な側面に関するヒントを示します。

型進化かサブタイプの作成かの決定

アプリケーションがライフ・サイクルを終了するまでには、既存のユーザー定義型を変更すべきか、それとも、新しい要件を満たす特化されたサブタイプを作成すべきか、という問題が頻繁に発生します。その答えは、新しい要件の性質およびアプリケーションのセマンティクス全体でのコンテキストによって異なります。次に 2 つの例をあげて説明します。

広範囲に使用されているベース型の変更

属性 Street、State、ZIP を持つユーザー定義型 address があるとします。

```
CREATE TYPE address AS OBJECT
(
Street  VARCHAR2(80),
State   VARCHAR2(20),
ZIP     VARCHAR2(10)
);
```

後になって、世界各地の住所をサポートするためには、Country 属性を追加して address 型を拡張する必要性に気がつきます。address のサブタイプの作成と address 型自体の進化とでは、どちらが得策でしょうか。

アプリケーション全体にわたって広範囲に使用されている一般的なベース型の場合は、型進化を通じて変更を実行することをお薦めします。

特化の追加

図形型（たとえば、曲線、円、正方形、テキスト）の既存の型階層に、追加のバリエーション、具体的にはベジエ曲線を格納する必要があるとします。ベース型の不足を反映しないこのような新しい特化をサポートするために、継承を使用して、Curve 型の下に、新しいサブタイプ BezierCurve を作成する必要があります。

つまり、必要とされる変更のセマンティクスによって、型進化を使用すべきか、それとも継承を使用すべきかが決まります。より全般的な変更で、ベース型に影響が及ぶ場合は、型進化を使用してください。より特化された変更に対しては、継承を使用して変更を実行してください。

ANYDATA とユーザー定義型の相違点

ANYDATA は Oracle によって提供される型で、任意の Oracle データ型のインスタンスを、組込み型かユーザー定義かにかかわらず保持することができます。ANYDATA は自己記述的な型で、インスタンスの形式を決定するために使用できるリフレクションに類似した API です。

代入性機能を通して、継承および ANYDATA は、ともにプレースホルダ内に考えられるインスタンスの任意の集合を格納するポリモフィックな能力を備えていますが、この 2 つのモデルがその機能を提供する形式は異なります。

継承モデルの場合は、考えられるインスタンスのポリモフィックな集合は、単一の型の階層の一構成部分である必要があります。変数には、定義された型またはそのサブタイプのインスタンスのみを保持する潜在的な能力があります。スーパータイプの属性にアクセスして、そのサブタイプで定義されている（またサブタイプによってオーバーライドされる可能性のある）メソッドをコールすることができます。IS OF 演算子および TREAT 演算子を使用して、あるインスタンスの具体的な型をテストすることもできます。

ただし、ANYDATA 変数は、異質なインスタンスを格納する可能性があります。インスタンスを抽出してしまわないかぎり、ANYDATA 変数に格納されている実際のインスタンスの属性にアクセスすることも、メソッドをコールすることもできません。該当するインスタンスの型を抽出するには、ANYDATA メソッドを使用します。ANYDATA は、ファンクション / プ

ロシージャがパラメータの具体的な型には関心がない場合にパラメータの受渡しを行う場合に便利です。

継承は優れたモデリング、厳密な型指定、特化などを提供します。共通するものがあるとはかぎらない、任意の数の考えられるインスタンスのうちの 1 つを保持するのみの場合は、ANYDATA を使用してください。

ポリモフィック・ビュー : オブジェクト・ビュー階層の代入

第 5 章では、それぞれが単一の型のオブジェクトを含んでいるオブジェクト・ビューの集合からビュー階層を構築する方法について説明しました。このようなビュー階層では、その階層内のビューに対する問合せによる、ビューまたはそのサブビューによって格納されたオブジェクトのポリモフィックな集合の参照が可能です。

このようなポリモフィックな問合せをサポートする代入の方法として、オブジェクトのポリモフィックな集合を戻す問合せに基づいてオブジェクト・ビューを定義できます。このアプローチは、あるビューを既存の表またはビューの集合にまたがって定義するときに特に便利です。

たとえば、Person_t のオブジェクト・ビューを、Employee_t インスタンスも含めた Person_t インスタンスを戻す問合せに対して定義できます。次の文からは、persons 表から個人を、employees 表から従業員を選択する問合せに基づいてビューが作成されます。

```
CREATE VIEW Persons_view OF Person_t AS
  SELECT Person_t(...) FROM persons
  UNION ALL
  SELECT TREAT(Employee_t(...) AS Person_t) FROM employees;
```

このビューに対して定義されている INSTEAD OF トリガーのかわりに、VALUE ファンクションを使用して、現在のオブジェクトにアクセスでき、また、オブジェクトの最も具体的な型に基づいて適切なアクションをとることができます。

ポリモフィック・ビューとオブジェクト・ビュー階層には、次のような重要な相違点があります。

- **アドレス指定可能度** : ビュー階層では、各サブビューは、問合せおよび DML 文の中で別々に参照できます。したがって、特定の型のオブジェクトの各集合には、論理名があります。一方、ポリモフィック・ビューは単一のビューなので、特定の型のオブジェクトの集合を取得するには、述語を使用する必要があります。
- **進化** : 新しいサブタイプを追加する場合は、既存のビュー定義を変更することなくビュー階層にサブビューを追加できます。ポリモフィック・ビューの場合は、別の UNION ブランチを追加することによって、単一のビュー定義を変更する必要があります。
- **DML 文** : ビュー階層では、各サブビューは、本質的に更新可能であるか、INSTEAD OF トリガーを持っているかのいずれかです。ポリモフィック・ビューの場合は、ビュー上の与えられた操作に対して INSTEAD OF トリガーを 1 つのみ定義することができます。

SQLJ オブジェクト型

SQLJ オブジェクト型の使用目的

『Information Technology - SQLJ - Part 2』ドキュメント（SQLJ 規格）によると、SQLJ オブジェクト型は、Java 用に設計されたデータベース・オブジェクト型です。SQLJ オブジェクト型は Java クラスにマップされます。マッピングが拡張 SQL の `CREATE TYPE` コマンド（DDL 文）によって登録されると、Java アプリケーションで Oracle9*i* の JDBC ドライバを通じて、Java オブジェクトを直接またはデータベースから、挿入または選択できるようになります。したがって、ユーザーは、同じクラスを、クライアントでは JDBC を使用して、サーバーでは SQL メソッドのディスパッチによって、配布できます。

SQLJ オブジェクト型の作成に必要な手順

拡張 SQL の `CREATE TYPE` コマンドは、次のことを行います。

- データベース・カタログに属性、ファンクションおよび Java クラスの外部名を移入します。また、Java クラスとそれに対応する SQLJ オブジェクト型間の依存関係は維持されます。
- Java クラスが存在することを検証し、また、それによって `USING` 句の値に対応するインタフェースが実装されることも検証します。
- Java フィールドが（`EXTERNAL NAME` 句で指定されているとおりに）存在すること、また、これらのフィールドが対応する SQL 属性に対する互換性を検証します。
- コンストラクタ、外部変数名、結果として `self` を返す外部ファンクションをサポートする内部クラスを生成します。

SQLJ オブジェクト型の使用に適した場面

SQLJ オブジェクト型は、SQL オブジェクト型の特殊なケースで、すべてのメソッドが Java クラスで実装されています。Java クラスと対応する SQL 型間のマッピングは、SQLJ オブジェクト型仕様によって管理されます。つまり、SQLJ オブジェクト型仕様には、対応する型本体の仕様はありません。

また、SQLJ オブジェクト型間の継承ルールによって、Java クラス階層と対応する SQLJ オブジェクト型階層の正しいマッピングが指定されます。これらのルールによって、SQLJ 型階層に有効なマッピングが含まれていることが保証されます。したがって、SQLJ オブジェクト型のスーパータイプまたはサブタイプは別の SQLJ オブジェクト型である必要があります。

カスタム SQLJ オブジェクト型の使用に適した場面

カスタム・オブジェクト型は、SQL オブジェクト型にアクセスするための Java インタフェースです。SQL オブジェクト型には、PL/SQL、Java、C などの言語で実装されたメソッドが含まれている場合があります。与えられた SQL オブジェクト型内の Java で実装さ

れたメソッドは、関連のない様々なクラスに属することができます。つまり、SQLJ オブジェクト型は特定の Java クラスにマップされるわけではありません。

クライアントがこれらのオブジェクトにアクセスできるようにするため、JPublisher を使用して対応する Java クラスを生成できます。さらに、ユーザーは、生成されたクラスを、対応するメソッドのコードで補強する必要があります。別の方法として、SQL オブジェクト型に対応するクラスを作成する方法もあります。

JDBC ユーザーは、マップ内の SQL 型名とそれに対応する Java クラス間の対応関係を実行時に登録する必要があります。

SQLJ オブジェクト型と JDBC を使用したカスタム・オブジェクト型の相違点

次の表は、SQLJ オブジェクト型とカスタム・オブジェクト型の相違点の要約です。

特性	SQLJ オブジェクト型の動作	カスタム・オブジェクト型の動作
型コード	`OracleTypes.JAVA_STRUCT` 型コードを使用して、SQLJ オブジェクト型を SQL の `OUT` パラメータとして登録します。`OracleTypes.JAVA_STRUCT` 型コードは、`ORAData` インタフェースまたは `SQLData` インタフェースを実装しているクラスの `_SQL_TYPECODE` フィールドでも使用されます。	`OracleTypes.STRUCT` 型コードを使用して、カスタム・オブジェクト型を SQL の `OUT` パラメータとして登録します。`OracleTypes.STRUCT` 型コードは、`ORAData` インタフェースまたは `SQLData` インタフェースを実装しているクラスの `_SQL_TYPECODE` フィールドでも使用されます。
作成	まず、`SQLData` または `ORAData` インタフェースおよび `ORADataFactory` インタフェースを実装する Java クラスを作成してから、Java クラスをデータベースにロードします。次に、SQLJ オブジェクト型に対して拡張 SQL の `CREATE TYPE` コマンドを発行します。	カスタム・オブジェクト型に対して拡張 SQL の `CREATE TYPE` コマンドを発行してから、JPublisher を使用して、または手動で `SQLData` または `ORAData` の Java ラッパー・クラスを作成します。
メソッド・サポート	外部名、コンストラクタ・コール、副作用のあるメンバー・ファンクションへのコールをサポートします。	型メソッドを Java メソッドとして実装するデフォルト・クラスはありません。一部のメソッドは SQL でも実装できます。
型マッピング	型マッピングは、拡張 SQL の `CREATE TYPE` コマンドによって自動的に行われます。ただし、SQLJ オブジェクト型には、それを定義する Java クラスがクライアント上にある必要があります。	型マップでの SQL と Java の対応関係を登録します。対応関係にない場合は、その型は `oracle.sql.STRUCT` として実現されます。
継承	SQL 階層を Java クラス階層にマップするためのルールがあります。これらのルールの詳細は『Oracle9*i* SQL リファレンス』を参照してください。	マッピング・ルールはありません。

その他のヒント

階層内の列の置換え可能性と属性の数

属性データを格納するため、列または表が型 T である場合は、Oracle によって、T の各属性に対して、列または表が置換え可能な場合は、T の各サブタイプの各属性に対して、非表示列が追加されます。同様に、行の中のオブジェクト・インスタンスの型を追跡するため、非表示の typeid 列も追加されます。

1 つの表内の列の数は、1,000 に制限されています。属性の合計数が 1,000 に迫っている型階層では、その階層内の型の置換え可能な列を使用しているときにこの制限を超えてしまうというリスクがあります。この結果発生する問題を回避するため、属性の合計数が大きい階層については、次のオプションのいずれかを検討してください。

- ビューの使用
- REF の使用
- 階層の分割

型間の循環的依存関係

型間に循環的な依存関係を発生させることは避けてください。つまり、T 型のメソッドが、型 T1（T 型を戻すメソッドを持つ）を戻すような状況を発生させないでください。

PL/SQL と TREAT と IS OF

PL/SQL は現在、TREAT 演算子と IS OF 演算子（第 2 章を参照）をサポートしていませんが、SQL はサポートしています。これらの演算子を使用するには、SQL を使用してください。

9

オブジェクト・リレーショナル機能を使用したサンプル・アプリケーション

この章では、ユーザー定義データ型（Oracle オブジェクト）の作成および使用方法の概要を説明する応用例を示します。

例では、顧客の発注書を管理するアプリケーションについて、様々なバージョンのデータベース・スキーマを作成します。まず、純粋なリレーショナル・バージョンを示し、次に同等のオブジェクト・リレーショナル・バージョンを示します。いずれのバージョンでも、顧客、発注書、明細項目などの同じ基本エンティティが使用されています。ただし、オブジェクト・リレーショナル・バージョンでは、これらのエンティティに対してユーザー定義型を作成し、それぞれのユーザー定義型のインスタンスをインスタンス化することによって、特定の顧客および発注書のデータを管理します。

内容は次のとおりです。

概要

ユーザー定義型とは、アプリケーション内のデータ構造および操作を形式化したスキーマ・オブジェクトのことです。

この章の例では、ユーザー定義型の定義、使用および進化における最も重要な点を説明します。ユーザー定義型の作業で重要な点の 1 つは、オブジェクトに対して操作を実行するメソッドを作成することです。この例では、オブジェクト型メソッドの定義で PL/SQL 言語を使用します。型の定義など、ユーザー定義型の使用にかかわるそれ以外の部分では SQL を使用します。

PL/SQL および Java を使用すると、特にコレクション要素のアクセスおよび操作の点で、この章に示す以上の機能が実現できます。

OCI、Pro*C/C++ または OO4O を使用するクライアント・アプリケーションでは、その広範な機能を利用してオブジェクトおよびコレクションにアクセスし、それらをクライアント上で操作できます。

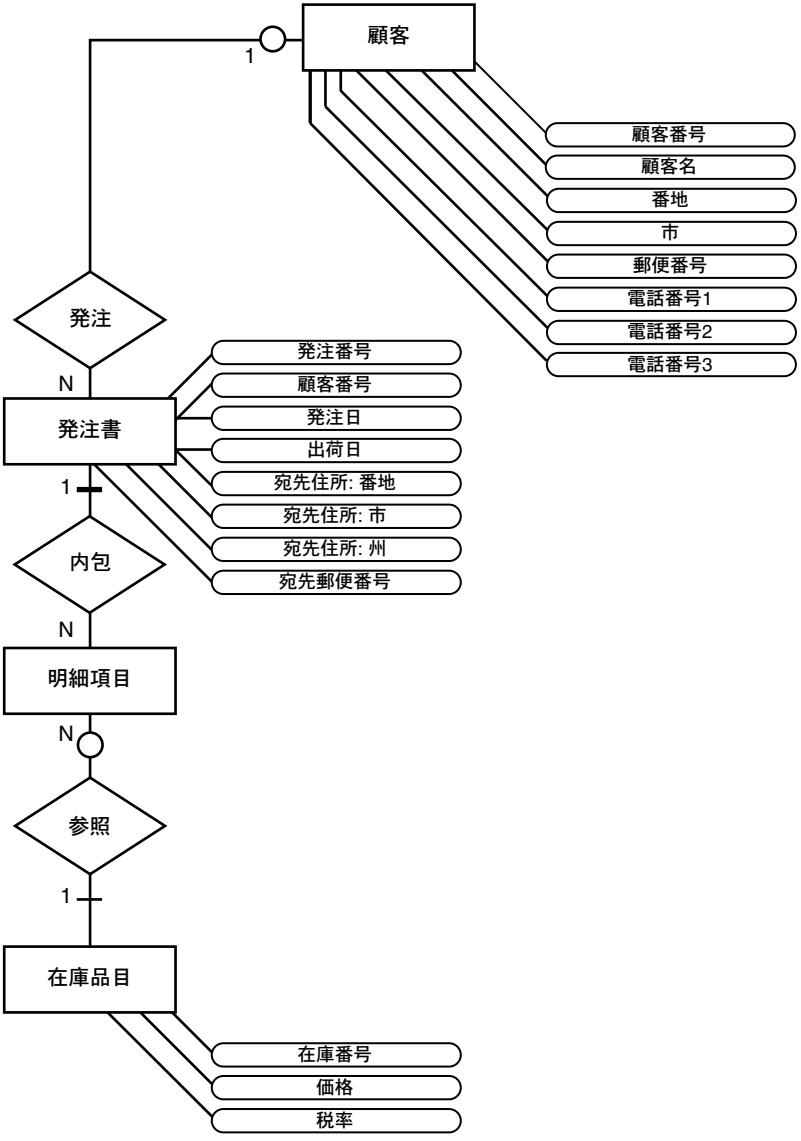
参照：

- ユーザー定義型の SQL 構文および使用方法の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。
- PL/SQL の機能の詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。
- Java の詳細は、『Oracle9*i* Java Stored Procedures Developer's Guide』を参照してください。
- 『Oracle Call Interface プログラマーズ・ガイド』を参照してください。
- 『Pro*C/C++ Precompiler プログラマーズ・ガイド』を参照してください。

リレーショナル・モデルでのスキーマの実装

この項では、図 9-1 に示す発注書スキーマのリレーショナル・バージョンを実装します。

図 9-1 発注アプリケーションのエンティティ・リレーションシップ（E-R）図



エンティティおよびリレーションシップ

この例の基本エンティティは、次のとおりです。

- 顧客
- 販売製品の在庫
- 発注書

図 9-1 からわかるように、顧客には連絡用の情報があり、住所および一連の電話番号はその顧客専用です。このアプリケーションでは、異なる顧客が同一の住所または電話番号と対応付けられることはありません。顧客が住所を変更した場合、以前の住所はなくなります。顧客でなくなった場合は、対応付けられた住所は削除されます。

顧客と発注書は 1 対多リレーションシップになります。1 人の顧客が複数の発注を行うことはできますが、ある 1 つの発注書を発行できる顧客は 1 人のみです。顧客はその顧客が発注を行う前に定義できるため、1 対多リレーションシップは必須でなく任意のものです。

同様に、発注書と在庫品目は多対多リレーションシップになります。多対多リレーションシップでは、どの在庫品目がどの発注書に記載されているかが示されないため、エンティティ・リレーションシップ（E-R）には明細項目という概念があります。発注書には、明細項目が 1 つ以上含まれる必要があります。各明細項目は、1 つの発注書のみに対応付けられています。

明細項目と在庫品目のリレーションシップは、在庫品目の記載がない明細項目があってもよく、1 つの在庫品目を 1 つ以上の明細項目に記載することもできますが、各明細項目が参照する在庫品目は 1 つのみです。

リレーショナル・モデルでの表の作成

リレーショナル・アプローチでは、すべてを正規化して表にします。表名は、`Customer_reltab`、`PurchaseOrder_reltab` および `Stock_reltab` です。

住所の各部は、`Customer_reltab` 表の列になります。

電話番号を列として構造化することによって、顧客の電話番号の数に任意の制限を設定します。

リレーショナル・アプローチでは、発注書と明細項目を切り離し、`PurchaseOrder_reltab` および `LineItems_reltab` という名前の独自の表に組み込みます。図 9-1 で示されるように、明細項目は、発注および在庫品目の両方に対してリレーションシップを持ちます。これらは、`PurchaseOrder_reltab` および `Stock_reltab` への外部キーを持つ、`LineItems_reltab` 表の列として実装されます。

注意： この項では、リレーショナル表の名前には接尾辞 _reltab を付ける規則に従っています。このような記述法を定めることによって、コードのメンテナンスが簡単になります。

これが表（_tab）と型（_typ）の区別に役立ちます。ただし、名前は任意に選択できます。オブジェクト・リレーショナル構造の主なメリットの 1 つは、対応する実社会のオブジェクトを忠実にモデル化する名前を付けることができる点です。

次に、リレーショナル・アプローチの表の例を示します。

Customer_reltab

Customer_reltab 表の定義は、次のとおりです。

```
CREATE TABLE Customer_reltab (
  CustNo                NUMBER NOT NULL,
  CustName              VARCHAR2(200) NOT NULL,
  Street                VARCHAR2(200) NOT NULL,
  City                  VARCHAR2(200) NOT NULL,
  State                 CHAR(2) NOT NULL,
  Zip                   VARCHAR2(20) NOT NULL,
  Phone1                VARCHAR2(20),
  Phone2                VARCHAR2(20),
  Phone3                VARCHAR2(20),
  PRIMARY KEY (CustNo)
  ) ;
```

この Customer_reltab 表には、顧客に関するすべての情報が格納されます。これは、（NOT NULL 制約で定義される）顧客に固有の情報、および必須ではないすべての情報が含まれることを意味します。この表定義によると、このアプリケーションではすべての顧客が出荷先住所を持つことが必須です。

エンティティ・リレーションシップ（E-R）図では発注を行う顧客が示されていましたが、表では、顧客と発注書間のリレーションシップは考慮されていません。このリレーションシップは発注書によって管理する必要があります。

PurchaseOrder_reltab

PurchaseOrder_reltab 表の定義は、次のとおりです。

```
CREATE TABLE PurchaseOrder_reltab (
   PONo        NUMBER, /* purchase order no */
   Custno      NUMBER references Customer_reltab, /*  Foreign KEY referencing
                                                      customer */
   OrderDate   DATE, /*  date of order */
```

```
  ShipDate    DATE, /* date to be shipped */
  ToStreet    VARCHAR2(200), /* shipto address */
  ToCity      VARCHAR2(200),
  ToState     CHAR(2),
  ToZip       VARCHAR2(20),
  PRIMARY KEY(PONo)
  ) ;
```

このように、PurchaseOrder_reltab は外部キー（FK）列 CustNo によって、顧客と発注書間のリレーションシップを管理します。CustNo は、Customer_reltab の CustNo キーを参照します。PurchaseOrder_reltab 表には、関連する明細項目の情報は含まれていません。明細項目表（次の項を参照）は、発注番号を使用して親発注書に明細項目を関連付けます。

LineItems_reltab

LineItems_reltab 表の定義は、次のとおりです。

```
CREATE TABLE LineItems_reltab (
  LineItemNo          NUMBER,
  PONo                NUMBER REFERENCES PurchaseOrder_reltab,
  StockNo             NUMBER REFERENCES Stock_reltab,
  Quantity            NUMBER,
  Discount            NUMBER,
  PRIMARY KEY (PONo, LineItemNo)
  ) ;
```

> **注意：** LineItems_reltab 表を作成する前に、9-6 ページの「Stock_reltab」で説明される Stock_reltab 表を作成してください。

表名には複数形（LineItems_reltab）が使用され、コードを読む他のユーザーに対して、この表が明細項目のコレクションを保持することが強調されています。

E-R 図で示すように、明細項目のリストは発注書および在庫品目の両方にリレーションシップを持ちます。これらのリレーションシップは、LineItems_reltab で次の 2 つの外部キー列によって管理されます。

- PONo: PurchaseOrder_reltab の PONo 列を参照します。
- StockNo: Stock_reltab の StockNo 列を参照します。

Stock_reltab

Stock_reltab 表の定義は、次のとおりです。

```
CREATE TABLE Stock_reltab (
  StockNo      NUMBER PRIMARY KEY,
  Price        NUMBER,
```

```
  TaxRate      NUMBER
  ) ;
```

リレーショナル・モデルでの値の挿入

このアプリケーションでは、次の文で表にデータを挿入します。

在庫品目録を作成する

```
INSERT INTO Stock_reltab VALUES(1004, 6750.00, 2) ;
INSERT INTO Stock_reltab VALUES(1011, 4500.23, 2) ;
INSERT INTO Stock_reltab VALUES(1534, 2234.00, 2) ;
INSERT INTO Stock_reltab VALUES(1535, 3456.23, 2) ;
```

顧客を登録する

```
INSERT INTO Customer_reltab
  VALUES (1, 'Jean Nance', '2 Avocet Drive',
         'Redwood Shores', 'CA', '95054',
         '415-555-1212', NULL, NULL) ;

INSERT INTO Customer_reltab
  VALUES (2, 'John Nike', '323 College Drive',
         'Edison', 'NJ', '08820',
         '609-555-1212', '201-555-1212', NULL) ;
```

発注する

```
INSERT INTO PurchaseOrder_reltab
  VALUES (1001, 1, SYSDATE, '10-MAY-1997',
          NULL, NULL, NULL, NULL) ;

INSERT INTO PurchaseOrder_reltab
  VALUES (2001, 2, SYSDATE, '20-MAY-1997',
         '55 Madison Ave', 'Madison', 'WI', '53715') ;
```

明細項目の細目

```
INSERT INTO LineItems_reltab VALUES(01, 1001, 1534, 12,  0) ;
INSERT INTO LineItems_reltab VALUES(02, 1001, 1535, 10, 10) ;
INSERT INTO LineItems_reltab VALUES(01, 2001, 1004,  1,  0) ;
INSERT INTO LineItems_reltab VALUES(02, 2001, 1011,  2,  1) ;
```

リレーショナル・モデルでのデータの問合せ

このアプリケーションでは、次の問合せを実行できます。

特定の発注書に対する顧客データおよび明細項目データを取得する

```
SELECT   C.CustNo, C.CustName, C.Street, C.City, C.State,
         C.Zip, C.phone1, C.phone2, C.phone3,
         P.PONo, P.OrderDate,
         L.StockNo, L.LineItemNo, L.Quantity, L.Discount
 FROM    Customer_reltab C,
         PurchaseOrder_reltab P,
         LineItems_reltab L
 WHERE   C.CustNo = P.CustNo
  AND    P.PONo = L.PONo
  AND    P.PONo = 1001 ;
```

発注書の合計値を取得する

```
SELECT     P.PONo, SUM(S.Price * L.Quantity)
 FROM      PurchaseOrder_reltab P,
           LineItems_reltab L,
           Stock_reltab S
 WHERE     P.PONo = L.PONo
  AND      L.StockNo = S.StockNo
 GROUP BY P.PONo ;
```

特定の在庫番号で識別される在庫品目を使用する明細項目に対する発注書データおよび明細項目データを取得する

```
SELECT    P.PONo, P.CustNo,
          L.StockNo, L.LineItemNo, L.Quantity, L.Discount
 FROM     PurchaseOrder_reltab P,
          LineItems_reltab     L
 WHERE    P.PONo = L.PONo
   AND    L.StockNo = 1004 ;
```

リレーショナル・モデルでのデータの更新

このアプリケーションでは、次の文を実行してデータを更新できます。

発注書 1001 の在庫品目 1534 の数量を更新する

```
UPDATE LineItems_reltab
   SET      Quantity = 20
   WHERE    PONo     = 1001
   AND      StockNo  = 1534 ;
```

リレーショナル・モデルでのデータの削除

このアプリケーションでは、次の文を実行してデータを削除できます。

発注書 1001 の削除

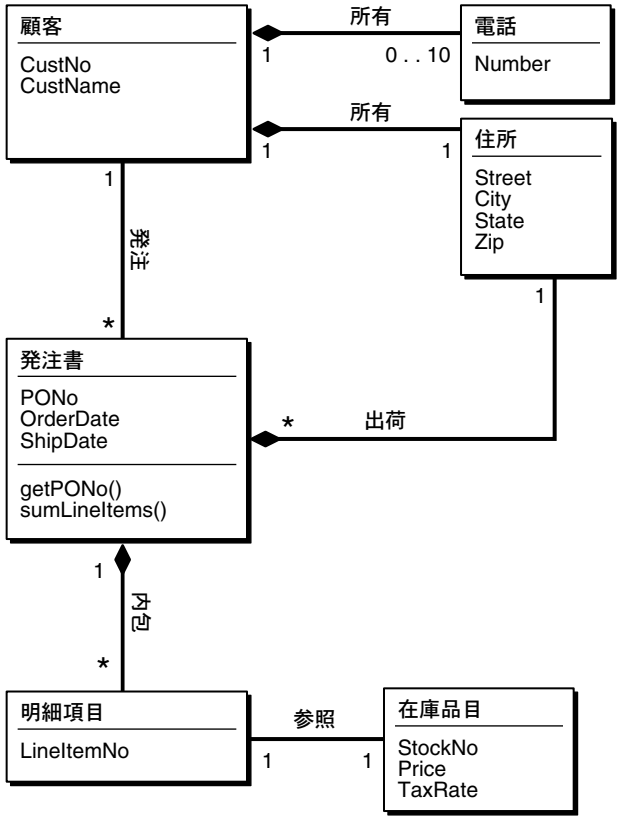
```
DELETE
   FROM   LineItems_reltab
   WHERE  PONo = 1001 ;

DELETE
   FROM   PurchaseOrder_reltab
   WHERE  PONo = 1001 ;
```

オブジェクト・リレーショナル・モデルでのスキーマの実装

オブジェクト・リレーショナル（O-R）・アプローチは、9-4 ページの「エンティティおよびリレーションシップ」と同じ E-R から始まります。これを次のクラス図のようにオブジェクト指向の観点で見ることによって、実社会の構造をより忠実にデータベース・スキーマに変換できます。

図 9-2 発注アプリケーションのクラス図



O-R アプローチでは、住所または複数の電話番号をリレーショナル表の関連のない列に分けるのではなく、住所全体および電話番号のリスト全体を表す型を定義します。また、O-R アプローチでは、明細項目および発注書を別々に格納するかわりにネストした表を使用します。

メイン・エンティティである顧客、在庫および発注書は、オブジェクト型となります。オブジェクト参照はそれらのリレーションシップの一部を示すために使用されます。コレクション型である VARRAY およびネストした表は、複数値属性をモデル化します。

注意： この章では、最初からオブジェクト・リレーショナル・スキーマを構築することにより、オブジェクト・リレーショナル・インタフェースを実装します。このアプローチでは、データを格納するためにオブジェクト表を作成します。また、オブジェクト表のかわりにオブジェクト・ビューを使用して、リレーショナル表に格納された既存のデータにオブジェクト・リレーショナル・インタフェースを実装する方法もあります。オブジェクト・ビューについては、第 5 章を参照してください。

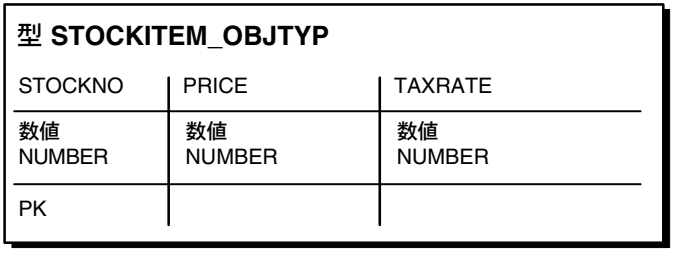
型の定義

ユーザー定義型は、CREATE TYPE 文で作成します。たとえば、次の文は、型 StockItem_objtyp を作成します。

```
CREATE TYPE StockItem_objtyp AS OBJECT (
  StockNo    NUMBER,
  Price      NUMBER,
  TaxRate    NUMBER
  );
```

型 StockItem_objtyp のインスタンスは、顧客が発注する在庫品目を表すオブジェクトです。これには、3 つの数値属性があります。

図 9-3 StockItem_objtyp のオブジェクト・リレーショナル表現



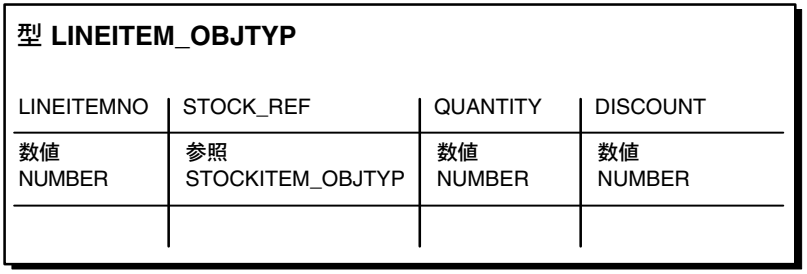
型を定義する順序によって、違いが発生します。理想的には、他の型を参照する型を定義する前に、参照される他の型を定義します。

たとえば、型 LineItem_objtyp は、StockItem_objtyp のオブジェクトへの REF である属性を含めることによって StockItem_objtyp を参照（前提）とします。これは、LineItem_objtyp 型を作成する文でもわかります。

```
CREATE TYPE LineItem_objtyp AS OBJECT (
  LineItemNo   NUMBER,
  Stock_ref    REF StockItem_objtyp,
  Quantity     NUMBER,
```

```
  Discount      NUMBER
  );
```

図 9-4 LineItem_objtyp 型のオブジェクト・リレーショナル表現



型 LineItem_objtyp のインスタンスは、明細項目を表すオブジェクトです。3 つの数値属性および 1 つの REF 属性で構成されます。LineItem_objtyp では、明細項目エンティティをモデル化しており、対応する在庫オブジェクトへのオブジェクト参照が含まれます。

型の循環参照によっては、前提とされるすべての型を作成する前に、型を作成することが困難または不可能である場合もあります。このような場合は、**不完全な型**と呼ばれるものを作成して、参照先として作成する他の型のプレースホルダとして使用します。他の型を作成した後、不完全な型を完全な型に置換します。

たとえば、StockItem_objtyp を作成する前に LineItem_objtyp を作成する必要がある場合、次のような文を使用して、不完全な型として LineItem_objtyp を作成します。

```
CREATE TYPE LineItem_objtyp;
```

不完全な型を作成するために使用する CREATE TYPE 文の構成には、AS OBJECT 句および属性が指定されていません。

不完全な型を完全な定義に置換するには、次に示すとおり、OR REPLACE 句を使用します。

```
CREATE OR REPLACE TYPE LineItem_objtyp AS OBJECT (
  LineItemNo   NUMBER,
  Stock_ref    REF StockItem_objtyp,
  Quantity     NUMBER,
  Discount     NUMBER
  );
```

置換する不完全な型が存在しない場合に、OR REPLACE を使用してもかまいません。

スキーマで必要な残りの型を作成します。次の文は、電話番号のリストの配列型を定義します。

```
CREATE TYPE PhoneList_vartyp AS VARRAY(10) OF VARCHAR2(20);
```

PhoneList_vartyp のすべてのデータ単位またはインスタンスは、最大 10 件の電話番号の VARRAY であり、各番号は VARCHAR2 型のデータ項目で表されます。

電話番号のリストを含めるために VARRAY またはネストした表を使用できます。この場合、リストは顧客ごとの連絡用電話番号の集合です。次の理由から、ネストした表より VARRAY の方をお薦めします。

- 番号の順序が重要な場合があります。ネストした表は順序付けられませんが、VARRAY は順序付けられます。
- 特定の顧客の電話番号の数は多くありません。VARRAY では、要素の最大数（この場合は 10）を事前に指定する必要があります。そのため、サイズ制限のないネストした表よりも、記憶域を効率的に使用できます。
- ネストした表は問合せ可能ですが、VARRAY は問合せできません。ただし、電話番号リストに対して問合せを行う必要はないため、ネストした表を使用することはありません。

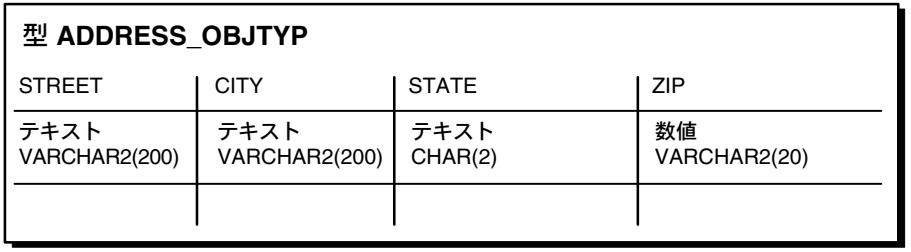
通常、順序付けおよび記憶域制限が設計上重要な問題でない場合は、次のような経験則を適用して VARRAY を使用するかまたはネストした表を使用するかを決定できます。コレクションを問い合せる必要がある場合はネストした表を使用し、コレクション全体を 1 つとして取り出す場合は VARRAY を使用します。

参照： VARRAY およびネストした表の設計上の考慮点の詳細は、第 8 章「Oracle オブジェクトの設計上の考慮点」を参照してください。

次の文で、住所を表すオブジェクト型 Address_objtyp を定義します。

```
CREATE TYPE Address_objtyp AS OBJECT (
  Street          VARCHAR2(200),
  City            VARCHAR2(200),
  State           CHAR(2),
  Zip             VARCHAR2(20)
  )
/
```

図 9-5 Address_objtyp 型のオブジェクト・リレーショナル表現



住所の各属性は文字列で、標準的な住所表記の各構成部分を表します。

次の文で、オブジェクト型 Customer_objtyp を定義します。この型は、他のユーザー定義型を組込みブロックとして使用します。

```
CREATE TYPE Customer_objtyp AS OBJECT (
  CustNo           NUMBER,
  CustName         VARCHAR2(200),
  Address_obj      Address_objtyp,
  PhoneList_var    PhoneList_vartyp,

  ORDER MEMBER FUNCTION
    compareCustOrders(x IN Customer_objtyp) RETURN INTEGER
) NOT FINAL;
```

型 Customer_objtyp のインスタンスは、特定の顧客についての情報ブロックを表すオブジェクトです。Customer_objtyp オブジェクトの属性は、型 PhoneList_vartyp の数値、文字列、Address_objtyp オブジェクトおよび VARRAY です。

NOT FINAL 句を使用すると、必要に応じて後で顧客型のサブタイプを作成できます。デフォルトでは、型は FINAL として作成されます。そこからサブタイプを導出し、さらに特化することはできません。この章の後半で、より特化された顧客に対する Customer_objtyp のサブタイプを定義します。

すべての Customer_objtyp オブジェクトには、2 種類の比較メソッドのうちの 1 つ、オーダー・メソッドが対応付けられています。Oracle では、2 つの Customer_objtyp オブジェクトの比較が必要になると、compareCustOrders メソッドが暗黙的に起動されます。

> **注意：** 比較メソッドを実装する PL/SQL は、9-18 ページの「compareCustOrders メソッド」にあります。

2 種類の比較メソッドとは、マップ・メソッドおよびオーダー・メソッドです。このアプリケーションでは、説明のため、両方のメソッドを 1 つずつ使用します。

オーダー・メソッドは、比較する 2 つのオブジェクトのペアごとにコールする必要があるのに対して、マップ・メソッドは、オブジェクトごとに 1 回ずつコールされます。一般に、オブジェクトの集合をソートする場合、オーダー・メソッドのコール回数は、マップ・メソッドの場合より多くなります。

参照：

- マップ・メソッドとオーダー・メソッドの詳細は、第 2 章を参照してください。
- プラグマ宣言の使用方法の詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

次の文は、明細項目のネストした表の型を定義します。各発注書では、発注書の明細項目を含めるために、このネストした表型のインスタンスを使用します。

```
CREATE TYPE LineItemList_ntabtyp AS TABLE OF LineItem_objtyp;
```

この型のインスタンスは、ネストした表オブジェクト（つまりネストした表）であり、各行には、型 LineItem_objtyp のオブジェクトが含まれます。複数値明細項目リストを表すには、次の理由から、LineItem_objtyp オブジェクトの VARRAY より、明細項目のネストした表の方が適しています。

- 明細項目の内容の問合せは、ほとんどのアプリケーションで必要です。明細項目がネストした表に格納されている場合、SQL を使用してこれを行うことができますが、VARRAY に格納されている場合にはできません。
- アプリケーションで明細項目データの索引付けが必要な場合、ネストした表では索引付けできますが、VARRAY ではできません。
- 明細項目の格納順序は重要ではなく、問合せによる発注は、明細項目番号によって実行できます。
- 発注書の明細項目の数には、実質的な上限はありません。VARRAY を使用すると、要素数の任意の上限を指定する必要があります。

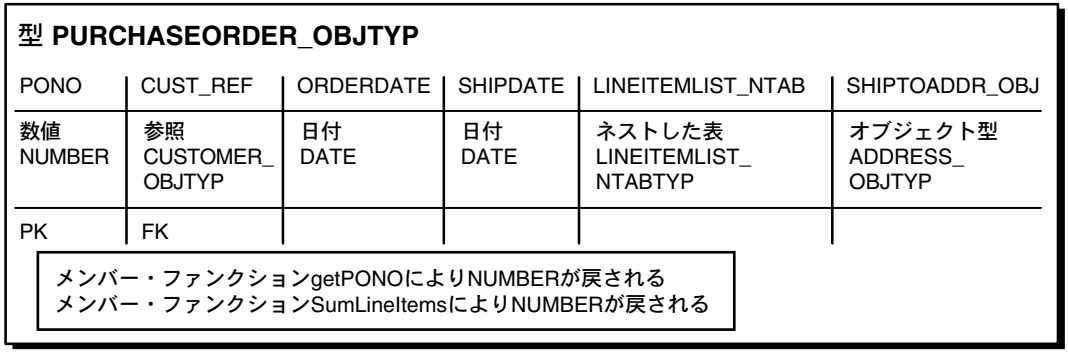
次の文は、オブジェクト型 PurchaseOrder_objtyp を定義します。

```
CREATE TYPE PurchaseOrder_objtyp AUTHID CURRENT_USER AS OBJECT (
  PONo                  NUMBER,
  Cust_ref              REF Customer_objtyp,
  OrderDate             DATE,
  ShipDate              DATE,
  LineItemList_ntab     LineItemList_ntabtyp,
  ShipToAddr_obj        Address_objtyp,

  MAP MEMBER FUNCTION
    getPONo RETURN NUMBER,
```

```
  MEMBER FUNCTION
    sumLineItems RETURN NUMBER
  );
```

図 9-6 PuchaseOrder_objtyp のオブジェクト・リレーショナル表現



型 PurchaseOrder_objtyp のインスタンスは、発注書を表すオブジェクトです。これには、Customer_objtyp への REF、Address_objtyp オブジェクトおよび型 LineItem_objtyp を基にしている型 LineItemList_ntabtyp のネストした表を含む 6 つの属性があります。

型 PurchaseOrder_objtyp のオブジェクトには、getPONo および sumLineItems の 2 つのメソッドがあります。そのうちの 1 つ、getPONo はマップ・メソッドで、2 種類の比較メソッドの 1 つです。マップ・メソッドは、オブジェクトのレコード順序内における指定レコードの相対位置を戻します。そのため、Oracle では、2 つの PurchaseOrder_objtyp オブジェクトの比較が必要になると、getPONo メソッドが暗黙的にコールされます。

2 つのプラグマ宣言で、2 つのメソッドでデータベースに対してどのようなアクセスが必要かが PL/SQL に通知されます。

前述の文には、メソッド getPONo および sumLineItems を実装する、実際の PL/SQL プログラムは含まれていません。実際のプログラムは、9-16 ページの「メソッドの定義」にあります。

メソッドの定義

型にメソッドが存在しない場合、定義には CREATE TYPE 文のみ含まれます。ただし、メソッドが存在する型の場合、型の定義を完全にするために型本体も定義する必要があります。これは CREATE TYPE BODY 文を使用します。CREATE TYPE と同様、OR REPLACE を

使用できます。メソッドを変更するために既存の型本体を新規のものと置き換える場合は、この句を追加する必要があります。

次の文は、型 PurchaseOrder_objtyp の本体を定義します。文は、型のメソッドを実装する PL/SQL プログラムを提供します。

```
CREATE OR REPLACE TYPE BODY PurchaseOrder_objtyp AS

MAP MEMBER FUNCTION getPONo RETURN NUMBER is
   BEGIN
      RETURN PONo;
   END;

MEMBER FUNCTION sumLineItems RETURN NUMBER is
      i             INTEGER;
      StockVal      StockItem_objtyp;
      Total         NUMBER := 0;

   BEGIN
      FOR i in 1..SELF.LineItemList_ntab.COUNT LOOP
         UTL_REF.SELECT_OBJECT(LineItemList_ntab(i).Stock_ref,StockVal);
         Total := Total + SELF.LineItemList_ntab(i).Quantity * StockVal.Price;
      END LOOP;
      RETURN Total;
   END;
END;
/
```

getPONo メソッド

getPONo メソッドは、メソッドをコールする型 PurchaseOrder_objtyp のいずれかのインスタンスの PONo 属性の値（発注番号）を単純に戻します。このような取得メソッドを使用すると、オブジェクトの内部表現が変更された場合に、そのオブジェクトを使用するコードを書きなおさずに済みます。

sumLineItems メソッド

sumLineItems メソッドでは、多くのオブジェクト・リレーショナル機能を使用します。

- 前述のとおり、sumLineItems メソッドの基本的な機能は、それに対応付けられた PurchaseOrder_objtyp オブジェクトの明細項目の請求額の合計を戻すことです。キーワード SELF は、すべてのファンクションのパラメータとして暗黙的に作成され、そのオブジェクトを参照します。
- キーワード COUNT は、PL/SQL 表または配列にある要素の総数を表します。ここで LOOP を組み合せると、アプリケーションは、コレクションのすべての要素（この場合、発注書の項目）で処理を繰り返します。このように、SELF.LineItemList_ntab.COUNT はネストした表内の要素の数を表し、SELF で表さ

れる PurchaseOrder_objtyp オブジェクトの LineItemList_ntab 属性と一致します。

- 実装にあたって、UTL_REF パッケージのメソッドが使用されます。Oracle では、PL/SQL プログラム内の REF の暗黙的な参照解除がサポートされていないため、UTL_REF メソッドが必要です。UTL_REF パッケージによって、オブジェクト参照を操作するメソッドが提供されます。ここで、Stock_ref に対応する StockItem_objtyp オブジェクトを取得するために、SELECT_OBJECT メソッドがコールされます。
- AUTHID CURRENT_USER 構文は、実行者権限を使用して PurchaseOrder_objtyp が定義されていることを指定します。したがって、このメソッドは、型を定義したユーザーの権限でなく、現行のユーザーの権限で実行されます。
- PL/SQL 変数 StockVal は、StockItem_objtyp 型です。UTL_REF.SELECT_OBJECT は、これを次のような参照を持つオブジェクトに設定します。

  ```
  (LineItemList_ntab(i).Stock_ref)
  ```

  このオブジェクトは、現在選択されている明細項目で参照される、実際の在庫品目です。
- 対象の在庫品目を取り出すと、次にそのコストが計算されます。このプログラムでは、在庫品目のコストが、StockVal.Price という、StockItem_objtyp オブジェクトの Price 属性として参照されます。ただし、項目のコストを計算するには、品目の発注量を知っておく必要があります。このアプリケーションでは、LineItemList_ntab(i).Quantity という項目で、現在選択されている LineItem_objtyp オブジェクトの Quantity 属性が表されます。

残りのメソッド・プログラムは明細項目の値を合計するループです。メソッドは、合計を戻します。

compareCustOrders メソッド

次の文は、Customer_objtyp オブジェクト型の型本体にある compareCustOrders メソッドを定義します。

```
CREATE OR REPLACE TYPE BODY Customer_objtyp AS
  ORDER MEMBER FUNCTION
  compareCustOrders (x IN Customer_objtyp) RETURN INTEGER IS
  BEGIN
    RETURN CustNo - x.CustNo;
  END;
END;
/
```

前述のとおり、オーダー・メソッド compareCustOrders を実行すると、2 つの顧客発注書が比較されます。このメソッドは、別の Customer_objtyp オブジェクトを入力の引数として受け取り、2 つの CustNo 数値の差を戻します。戻り値は次のとおりです。

- 負数 : 内部にあるオブジェクトの CustNo の値が小さい場合
- 正数 : 内部にあるオブジェクトの CustNo の値が大きい場合
- 0（ゼロ）: 2 つのオブジェクトが持つ CustNo の値が同じ場合（この場合、自分自身を参照します）

戻り値が正、負または 0（ゼロ）であるかによって、顧客番号の相対順序が示されます。たとえば、小さい番号が大きい番号より先に作成されます。オーダー・メソッドの入力引数（SELF および明示的引数）のいずれかが NULL である場合、Oracle によってオーダー・メソッドがコールされることはなく、結果は NULL として扱われます。

これで、発注書スキーマのオブジェクト・リレーショナル・バージョンのすべてのユーザー定義型を定義しました。ただし、実際の発注書データを追加するこれらの型のインスタンス、およびこれらのデータを格納する表は作成していません。これらの方法については、次の項を参照してください。

オブジェクト表の作成

オブジェクト型の作成は、表の作成とは異なります。型の作成は、論理構造を定義するのみで、記憶域は作成しません。オブジェクト・リレーショナル・インタフェースを使用するには、オブジェクト表にデータを格納する場合でも、リレーショナル表にデータを残してオブジェクト・ビューからアクセスする場合でも、オブジェクト型を作成する必要があります。オブジェクト・ビューおよびオブジェクト表は、オブジェクト型を前提としています。オブジェクト表またはオブジェクト・ビューは、常に特定のオブジェクト型の表またはビューです。この点で、指定したデータ型を常に持つリレーショナル列に類似しています。

> **参照 :** オブジェクト・ビューの詳細は、第 5 章「オブジェクト・モデルのリレーショナル・データへの適用」を参照してください。

リレーショナル列と同様に、オブジェクト表には、1 種類の行（つまり、表と同じ宣言済の型のオブジェクト・インスタンス）を追加できます。（また、表が代入可能な場合、宣言済の型のサブタイプのインスタンスも追加できます）。

オブジェクト表の各行は、単一のオブジェクト・インスタンスです。つまり、オブジェクト表には、宣言済のオブジェクト型の単一の列のみが含まれます。ただし、リレーショナル表と大きな違いはありません。リレーショナル表の各行は、リレーショナル表 Customers の顧客など、理論的には単一のエンティティを表します。リレーショナル表の列には、このエンティティの属性のデータが格納されます。

同様に、オブジェクト表では、オブジェクト型の属性は、オブジェクト表への挿入やオブジェクト表から選択が可能な列にマップされます。大きな違いは、オブジェクト表では、データが表の型によって定義された構造に格納および取得されるため、ごく単純な問合せでデータの完全なマルチレベル構造を取得できることです。

オブジェクト表 Customer_objtab

次の文で、Customer_objtyp 型のオブジェクトを保持するオブジェクト表 Customer_objtab を定義します。

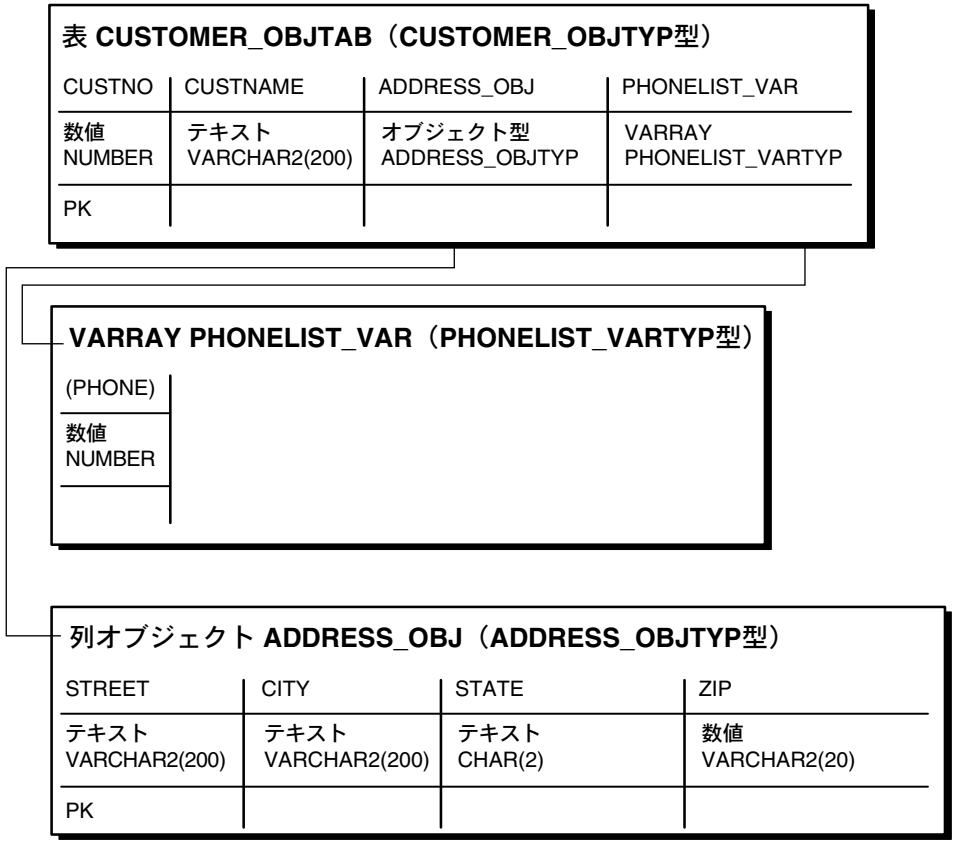
```
CREATE TABLE Customer_objtab OF Customer_objtyp (CustNo PRIMARY KEY)
   OBJECT IDENTIFIER IS PRIMARY KEY ;
```

リレーショナル表と異なり、オブジェクト表を作成する場合は、データ型（オブジェクト表に追加するオブジェクトの型）を指定します。

表には、Customer_objtyp の各属性の列が存在します。

```
CustNo           NUMBER
CustName         VARCHAR2(200)
Address_obj      Address_objtyp
PhoneList_var    PhoneList_vartyp
```

図 9-7 Customer_objtab 表のオブジェクト・リレーショナル表現



オブジェクト表のテンプレートとしてのオブジェクトのデータ型

Customer_objtyp という型があるため、同じ型のオブジェクト表を多数作成できます。たとえば、同じく Customer_objtyp 型のオブジェクト表 Customer_objtab2 を作成することもできます。

複数の表を作成する場合は、バリエーションを導入できます。Customer_objtab を作成した文によって、CustNo 列に主キー制約が定義されました。この制約は、このオブジェクト表のみに適用されます。同じ型の別のオブジェクト表には、この制約がない場合があります。

オブジェクト識別子（OID）および参照

Customer_objtab には、行オブジェクトとして表される顧客オブジェクトが含まれています。Oracle では、行オブジェクトが参照できます。これは、他の行オブジェクトまたはリレーショナル行が、行オブジェクトの OID を使用して、行オブジェクトを参照できることを意味します。たとえば、発注書の行オブジェクトは、顧客行オブジェクトに対するオブジェクト参照を使用して、顧客行オブジェクトを参照できます。オブジェクト参照は、REF 型で表されるシステム生成の値で、行オブジェクトの一意の OID に基づきます。

Oracle では、すべての行オブジェクトには一意の OID があることが必須です。一意の OID 値がシステムによって生成されるように指定することも、行オブジェクトの主キーが一意の OID として機能するように指定することもできます。CREATE TABLE 文を実行する際に、OBJECT IDENTIFIER IS PRIMARY KEY または OBJECT IDENTIFIER IS SYSTEM GENERATED（デフォルト）を指定することによって、これを指定します。後者の指定がデフォルトです。主キーの値が 16 バイト（システム生成の識別子のデフォルト値）より小さい場合、OID として主キーを使用すると、効率が向上する場合があります。ここでの例では、行 OID として主キーが使用されています。

埋込みオブジェクト付きのオブジェクト表

Customer_objtab の Address_obj 列には、Address_objtyp オブジェクトが含まれることに注意してください。このように、オブジェクト型は、それ自体がオブジェクト型である属性を持つことができます。オブジェクト表の宣言された型のオブジェクト・インスタンスは、1 つのオブジェクト・インスタンスが表の行全体を占有するため、**行オブジェクト**と呼ばれます。Address_obj 列のような埋込みオブジェクトは、**列オブジェクト**と呼ばれます。行オブジェクトとの違いは、行全体を占有しないことです。したがって、参照はできず、REF の対象にはなりません。NULL の場合もあります。

Address_objtyp オブジェクトの属性は、組込み型です。これらは複合型（つまり、個々の属性を持つオブジェクト型）ではなく、スカラー型であるため、分岐の終端を表すことから**リーフ・レベル**属性と呼ばれます。Address_objtyp オブジェクトの列およびそれらの属性が、オブジェクト表 Customer_objtab に作成されます。これらの列は、ドット表記法を使用して参照またはナビゲートできます。たとえば、Zip 列に索引を作成する場合、これを Address.Zip として参照できます。

PhoneList_var 列には、PhoneList_vartyp 型の VARRAY が含まれます。PhoneList_vartyp 型の各オブジェクトは、最大 10 件の電話番号の VARRAY として定義されており、各番号は VARCHAR2 型のデータ項目として表されます。

```
CREATE TYPE PhoneList_vartyp AS VARRAY(10) OF VARCHAR2(20);
```

PhoneList_vartyp 型の各 VARRAY は、最大 200 文字（10 × 20）と多少のオーバーヘッドのみ含むことができるため、Oracle は、VARRAY を単一のデータ単位として PhoneList_var 列に格納します。Oracle では、4000 バイト以下の VARRAY はインライン BLOB に格納されます。つまり、VARRAY 値の一部は、表外に格納される可能性があります。

オブジェクト表 Stock_objtab

次の文は、StockItem_objtyp オブジェクトのオブジェクト表を作成します。

```
CREATE TABLE Stock_objtab OF StockItem_objtyp (StockNo PRIMARY KEY)
   OBJECT IDENTIFIER IS PRIMARY KEY ;
```

表の各行は、StockItem_objtyp オブジェクトで、それぞれ次の 3 つの数値属性があります。

```
StockNo     NUMBER
Price       NUMBER
TaxRate     NUMBER
```

Oracle によって、属性ごとに列が作成されます。Oracle では、各属性に列が割り当てられ、CREATE TABLE 文によって、StockNo 列に主キー制約が設定されます。また、主キーが行オブジェクトの識別子として使用されるよう指定されます。

オブジェクト表 PurchaseOrder_objtab

次の文は、PurchaseOrder_objtyp オブジェクトのオブジェクト表を定義します。

```
CREATE TABLE PurchaseOrder_objtab OF PurchaseOrder_objtyp (  /* Line 1 */
   PRIMARY KEY (PONo),                                       /* Line 2 */
   FOREIGN KEY (Cust_ref) REFERENCES Customer_objtab)        /* Line 3 */
   OBJECT IDENTIFIER IS PRIMARY KEY                          /* Line 4 */
   NESTED TABLE LineItemList_ntab STORE AS PoLine_ntab (     /* Line 5 */
     (PRIMARY KEY(NESTED_TABLE_ID, LineItemNo))              /* Line 6 */
     ORGANIZATION INDEX COMPRESS)                            /* Line 7 */
   RETURN AS LOCATOR                                         /* Line 8 */
/
```

前述の CREATE TABLE 文では、PurchaseOrder_objtab オブジェクト表が作成されます。各行の重要点は、次のとおりです。

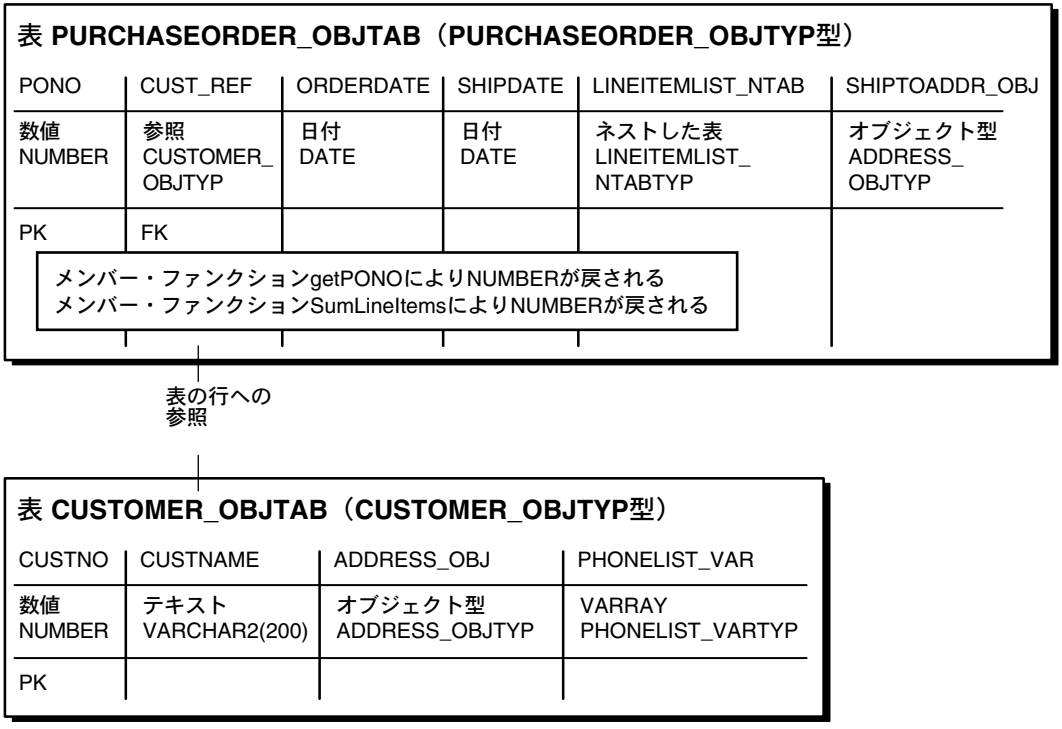
行 1:

```
CREATE TABLE PurchaseOrder_objtab OF PurchaseOrder_objtyp (
```

この行は、表の各行が PurchaseOrder_objtyp オブジェクトであることを示します。PurchaseOrder_objtyp オブジェクトの属性は、次のとおりです。

```
  PONo                NUMBER
  Cust_ref            REF Customer_objtyp
  OrderDate           DATE
  ShipDate            DATE
  LineItemList_ntab   LineItemList_ntabtyp
  ShipToAddr_obj      Address_objtyp
```

図 9-8 PurchaseOrder_objtab 表のオブジェクト・リレーショナル表現



行 2:

```
PRIMARY KEY (PONo),
```

この行は、PONo 属性が表の主キーであることを指定します。

行 3:

```
FOREIGN KEY (Cust_ref) REFERENCES Customer_objtab)
```

この行は、Cust_ref 列についての参照制約を指定します。この参照制約は、リレーショナル表に対して指定されたものと似ています。制約がない場合、REF 列によって任意の行オブジェクトを参照できます。ただし、この場合、Cust_ref REF では Customer_objtab オブジェクト表の行オブジェクトのみ参照できます。

行 4:

```
OBJECT IDENTIFIER IS PRIMARY KEY
```

この行は、PurchaseOrder_objtab オブジェクト表の主キーが行の OID として使用されることを示します。

行 5 〜 8:

```
NESTED TABLE LineItemList_ntab STORE AS PoLine_ntab (
     (PRIMARY KEY(NESTED_TABLE_ID, LineItemNo))
     ORGANIZATION INDEX COMPRESS)
   RETURN AS LOCATOR
```

これらの行は、ネストした表の LineItemList_ntab 列の記憶域仕様およびプロパティを示します。ネストした表の行は、別の記憶表に格納されます。この記憶表に対して、ユーザーが直接問合せを実行することはできませんが、メンテナンスの目的で DDL 文で参照できます。記憶表には、NESTED_TABLE_ID という非表示列があり、対応する親である行の行と一致します。特定の親に属するネストした表のすべての要素に、同一の NESTED_TABLE_ID 値があります。たとえば、PurchaseOrder_objtab の任意の行にある、ネストした表のすべての要素は、NESTED_TABLW_ID の値が同じです。ネストした表の要素のうち、PurchaseOrder_objtab の別の行に属する要素は、NESTED_TABLE_ID の値が異なっています。

前述の CREATETABLE の例で、行 5 は、ネストした表 LineItemList_ntab の行が PoLine_ntab という名前の（記憶表として参照される）別の表に格納されることを示します。STOREAS 句によって、記憶表に対する制約および記憶域仕様を指定することもできます。この例で、行 7 は、IOT であることを示します。一般に、1 つの IOT にネストした表の行を格納すると効果的です。これによって、同じ親に属している行がクラスタ化されます。IOT で COMPRESS を指定すると、記憶域が節約されます。これは、COMPRESS を指定しない場合、IOT のキーである NESTED_TABLE_ID 部分が、親の行オブジェクトのすべての行に対して繰り返されるためです。一方、COMPRESS を指定すると、NESTED_TABLE_ID は親である各行のオブジェクトに 1 回のみ格納されます。

CREATE TABLE 文では、REF への SCOPEFOR 制約は指定できません。したがって、Stock_ref がオブジェクト表 Stock_objtab のみを参照できるように指定するには、PoLine_ntab 記憶表に次の ALTER TABLE 文を発行します。

```
ALTER TABLE PoLine_ntab
   ADD (SCOPE FOR (Stock_ref) IS stock_objtab) ;
```

この文は、親表でなく記憶表に対して実行されることに注意してください。

> **参照：** ネストした表を IOT として構成してネストした表の圧縮を指定する利点、および一般的なネストした表の記憶域の詳細は、8-15 ページの「ネストした表の記憶域」を参照してください。

行 6 で、NESTED_TABLE_ID および LineItemNo 属性を記憶表の主キーとして指定することには、目的が 2 つあります。第 1 に、これは IOT のキーとして機能します。第 2 に、親表の各行の中にあるネストした表の列（LineItemNo）の一意性を施行します。キーに LineItemNo 列を組み込むことによって、この文で、各発注書の LineItemNo 列の値が確実に一意になります。

行 8 は、ネストした表 LineItemList_ntab が取り出される際に、ロケータ形式で戻されることを示します。LOCATOR を指定しないとデフォルトは VALUE となり、ネストした表に対するロケータのみが戻されるかわりに、ネストした表全体が戻されることを示します。ネストした表コレクションに多くの要素が含まれる場合、含まれている行オブジェクトまたは列が選択されるたびにネストした表全体を戻すのは、効率的ではありません。

ネストした表のロケータが戻されるように指定すると、実際のコレクション値に対するロケータのみがクライアントに戻されます。アプリケーションは、OCICollIsLocator インタフェースまたは UTL_COLL.IS_LOCATOR インタフェースをコールすることで、フェッチされたネストした表がロケータ形式か、値形式であるかがわかります。ロケータが戻されていることがわかると、アプリケーションはロケータを使用して問い合せ、ネストした表内の必要な行要素のサブセットのみをフェッチできます。このようなネストした表の行の、ロケータベースの取出しは、元の文のスナップショットに基づいており、ネストした表の値またはコピー・セマンティクスを保ちます。つまり、ネストした表の行要素のサブセットをフェッチするためにロケータが使用される場合、ネストした表のスナップショットは、ロケータが最初に取り出された時点のネストした表を反映しています。

9-16 ページの「メソッドの定義」で説明した、PurchaseOrder_objtyp の sumLineItems メソッドの実装についてもう一度考えてみます。この実装では、ネストした表 LineItemList_ntab が VALUE として戻されることを想定しています。大きいネストした表をより効率的に処理し、PurchaseOrder_objtab 内のネストした表がロケータとして戻されることを活用するには、sumLineItems メソッドを次のように書きなおす必要があります。

```
CREATE OR REPLACE TYPE BODY PurchaseOrder_objtyp AS

   MAP MEMBER FUNCTION getPONo RETURN NUMBER is
      BEGIN
         RETURN PONo;
      END;

   MEMBER FUNCTION sumLineItems RETURN NUMBER IS
      i          INTEGER;
      StockVal   StockItem_objtyp;
      Total      NUMBER := 0;
```

```
   BEGIN
      IF (UTL_COLL.IS_LOCATOR(LineItemList_ntab)) -- check for locator
         THEN
            SELECT SUM(L.Quantity * L.Stock_ref.Price) INTO Total
            FROM   TABLE(CAST(LineItemList_ntab AS LineItemList_ntabtyp)) L;
      ELSE
         FOR i in 1..SELF.LineItemList_ntab.COUNT LOOP
            UTL_REF.SELECT_OBJECT(LineItemList_ntab(i).Stock_ref,StockVal);
            Total := Total + SELF.LineItemList_ntab(i).Quantity *
                                                              StockVal.Price;
         END LOOP;
      END IF;
   RETURN Total;
   END;
END;
/
```

書きなおされた sumLineItems メソッドでは、UTL_COLL.IS_LOCATOR ファンクションを使用して、ネストした表の属性 LineItemList_ntab がロケータとして戻されているかどうかがチェックされます。条件が TRUE と評価された場合、TABLE 式を使用して、ネストした表のロケータが問い合せられます。

注意： 現在このような TABLE 式では、CAST 式が必要です。これは、SQL コンパイル・エンジンが問合せをコンパイルできるように、SQL コンパイル・エンジンにコレクション属性（またはパラメータ、変数）の実際の型を知らせることを目的としています。

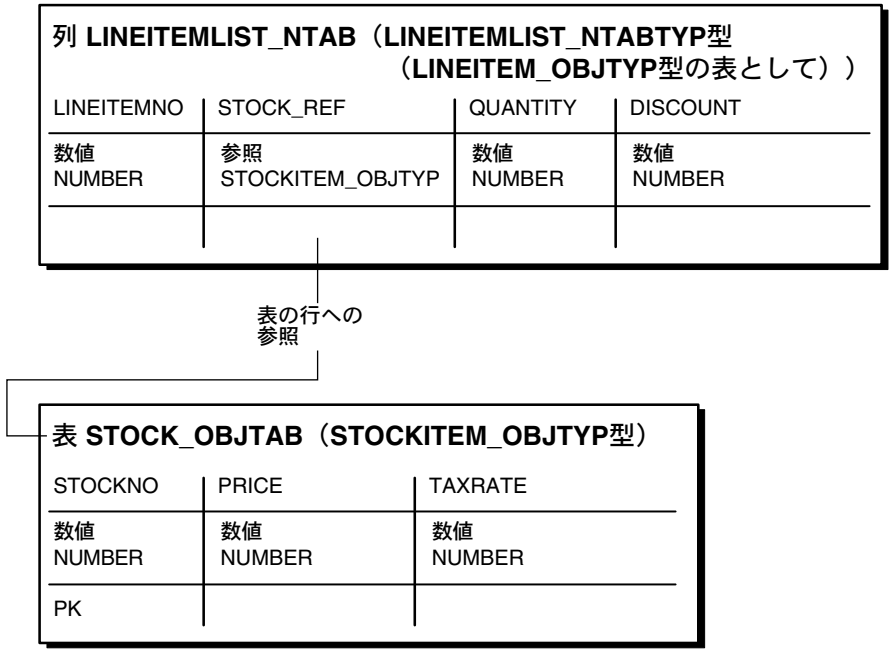
ネストした表のロケータを問い合せることで、発注書の膨大な明細項目リストの処理がより効率的になります。LineItemList_ntab で繰返しを行う前述のコードは、ネストした表が VALUE として戻される場合を処理するために保持されています。

表が作成された後、次の ALTER TABLE 文が発行されます。

```
ALTER TABLE PoLine_ntab
   ADD (SCOPE FOR (Stock_ref) IS stock_objtab);
```

この文は、ネストした表の Stock_ref 列の有効範囲が Stock_objtab にかぎられることを指定します。これは、この列に格納される値が、Stock_objtab の行オブジェクトへの参照である必要があることを示します。SCOPE 制約と参照制約の相違は、SCOPE 制約の方は、参照されるオブジェクトに依存しない点にあります。たとえば、Stock_objtab 内の参照される行オブジェクトは、ネストした表 Stock_ref 列で参照されている場合でも削除できます。このような削除を行うと、ネストの対応する参照は、DANGLING REF になります。

図 9-9 ネストした表 LineItemList_ntab のオブジェクト・リレーショナル表現

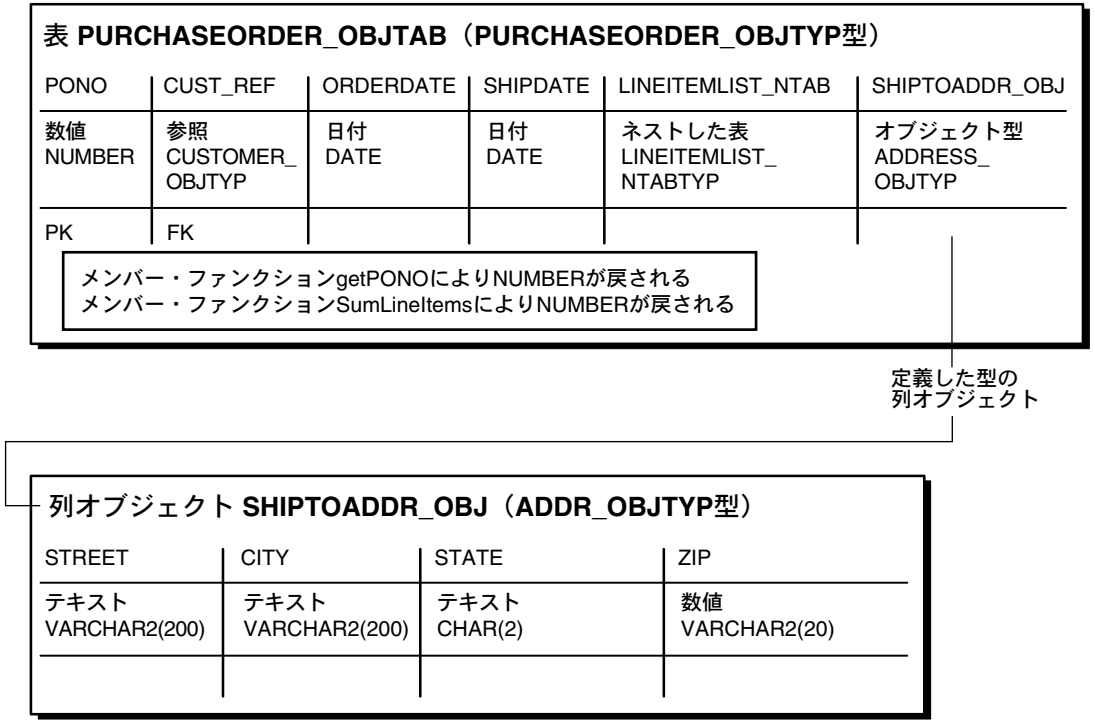


Oracle では、記憶表に対して参照制約は指定できません。ネストした表の場合は、REF 列に対して SCOPE 句を指定すると便利です。一般に、REF 列に有効範囲または参照制約を指定すると、いくつか利点があります。

- Oracle によって、行オブジェクトの一意識別子のみが列の REF 値として格納されるようになるため、記憶域が節約されます。
- 記憶表の REF 列に索引が作成できるようになります。
- Oracle によって、これらの REF の参照解除を含む問合せが、参照表を含む結合として再作成できるようになります。

ここまでで、発注書アプリケーションのすべての表が設定されました。次の項では、これらの表の操作方法を示します。

図 9-10 PurchaseOrder_objtab 表のオブジェクト・リレーショナル表現



値の挿入

この項では、事前にリレーショナル表で行ったように、オブジェクト表に同じデータを挿入する方法を説明します。値の一部が、型のインスタンスを作成するためにオブジェクト型のコンストラクタへのコールを取り込むことに注意してください。

Stock_objtab

```
INSERT INTO Stock_objtab VALUES(1004, 6750.00, 2) ;
INSERT INTO Stock_objtab VALUES(1011, 4500.23, 2) ;
INSERT INTO Stock_objtab VALUES(1534, 2234.00, 2) ;
INSERT INTO Stock_objtab VALUES(1535, 3456.23, 2) ;
```

Customer_objtab

```
INSERT INTO Customer_objtab
  VALUES (
    1, 'Jean Nance',
```

```
    Address_objtyp('2 Avocet Drive', 'Redwood Shores', 'CA', '95054'),
    PhoneList_vartyp('415-555-1212')
    ) ;

INSERT INTO Customer_objtab
  VALUES (
    2, 'John Nike',
    Address_objtyp('323 College Drive', 'Edison', 'NJ', '08820'),
    PhoneList_vartyp('609-555-1212','201-555-1212')
    ) ;
```

PurchaseOrder_objtab

```
INSERT INTO PurchaseOrder_objtab
  SELECT  1001, REF(C),
          SYSDATE, '10-MAY-1999',
          LineItemList_ntabtyp(),
          NULL
   FROM   Customer_objtab C
   WHERE  C.CustNo = 1 ;
```

前述の文で、次の属性を持つ PurchaseOrder_objtyp オブジェクトが構成されます。

```
  PONo                1001
  Cust_ref            REF to customer number 1
  OrderDate           SYSDATE
  ShipDate            10-MAY-1999
  LineItemList_ntab   an empty LineItem_ntabtyp
  ShipToAddr_obj      NULL
```

この文は、問合せを使用して、CustNo 値が 1 である Customer_objtab オブジェクト表の行オブジェクトに対する REF を構成します。

次の文は、TABLE 式を使用して、ネストした表（PONo 値が 1001 である PurchaseOrder_objtab 表の行オブジェクトの LineItemList_ntab 列にあるネストした表）を挿入先として識別します。

> **注意：** Oracle8 リリース 8.0 でサポートされていた、ネストした表を識別するためのフラット化した副問合せまたは THE（副問合せ）式は使用されなくなり、かわりに、このリリースでは、次の例に示す TABLE 式構文がサポートされています。

```
INSERT INTO TABLE (
  SELECT  P.LineItemList_ntab
   FROM   PurchaseOrder_objtab P
   WHERE  P.PONo = 1001
```

```
  )
  SELECT  01, REF(S), 12, 0
   FROM   Stock_objtab S
   WHERE  S.StockNo = 1534 ;
```

前述の文で、TABLE 式で識別されたネストした表に明細項目が挿入されます。挿入される明細項目には、StockNo 値が 1534 であるオブジェクト表 Stock_objtab の行オブジェクトに対する REF があります。

次の文は、前述の文と同じパターンです。

```
INSERT INTO PurchaseOrder_objtab
  SELECT  2001, REF(C),
          SYSDATE, '20-MAY-1997',
          LineItemList_ntabtyp(),
          Address_objtyp('55 Madison Ave','Madison','WI','53715')
   FROM   Customer_objtab C
   WHERE  C.CustNo = 2 ;

INSERT INTO TABLE (
  SELECT  P.LineItemList_ntab
   FROM   PurchaseOrder_objtab P
   WHERE  P.PONo = 1001
  )
  SELECT  02, REF(S), 10, 10
   FROM   Stock_objtab S
   WHERE  S.StockNo = 1535 ;

INSERT INTO TABLE (
  SELECT  P.LineItemList_ntab
   FROM   PurchaseOrder_objtab P
   WHERE  P.PONo = 2001
  )
  SELECT  10, REF(S), 1, 0
   FROM   Stock_objtab S
   WHERE  S.StockNo = 1004 ;

INSERT INTO TABLE (
  SELECT  P.LineItemList_ntab
   FROM   PurchaseOrder_objtab P
   WHERE  P.PONo = 2001
  )
  VALUES(11, (SELECT REF(S)
    FROM  Stock_objtab S
    WHERE S.StockNo = 1011), 2, 1) ;
```

問合せ

次の問合せ文では、比較メソッドが暗黙的にコールされます。ここでは、PurchaseOrder_objtyp 型のオブジェクトを、その型の比較メソッドを使用して Oracle でどのように順序付けるかを示します。

```
SELECT  p.PONo
 FROM   PurchaseOrder_objtab p
 ORDER BY VALUE(p) ;
```

選択内のそれぞれの PurchaseOrder_objtyp オブジェクトに対し、マップ・メソッド getPONo が起動します。このメソッドは、オブジェクトの PONo 属性を戻すため、選択は発注書番号が昇順に並んだリストを生成します。

次の問合せは、リレーショナル・モデルで実行された問合せに対応しています。

発注書 1001 の顧客および明細項目データ

```
SELECT  DEREF(p.Cust_ref), p.ShipToAddr_obj, p.PONo,
        p.OrderDate, LineItemList_ntab
 FROM   PurchaseOrder_objtab p
 WHERE  p.PONo = 1001 ;
```

各発注書の合計金額

```
SELECT   p.PONo, p.sumLineItems()
 FROM    PurchaseOrder_objtab p ;
```

在庫品目 1004 の発注書および明細項目

```
SELECT   po.PONo, po.Cust_ref.CustNo,
         CURSOR (
           SELECT  *
            FROM   TABLE (po.LineItemList_ntab) L
            WHERE  L.Stock_ref.StockNo = 1004
           )
 FROM    PurchaseOrder_objtab po ;
```

前述の問合せでは、ネストした表から選択された一連の LineItem_obj オブジェクトの集合に対するネステッド・カーソルが戻されます。アプリケーションは、ネステッド・カーソルからフェッチして、個々の LineItem_obj オブジェクトを取得できます。前述の問合せは、出力結果に関して、前述のネストした集合をネスト解除することによって、次のように表すこともできます。

```
SELECT   po.PONo, po.Cust_ref.CustNo, L.*
 FROM    PurchaseOrder_objtab po, TABLE (po.LineItemList_ntab) L
 WHERE   L.Stock_ref.StockNo = 1004 ;
```

前述の問合せは、結果セットをフラット化したフォーム（または第 1 正規形）として戻します。このような問合せは、ODBC などのリレーショナル・ツールおよび API から Oracle コ

レクション列にアクセスする場合に役立ちます。前述のネストを解除する例では、LineItemList_ntab 行を持つ PurchaseOrder_objtab オブジェクト表の行のみが戻されます。対応する LineItemList_ntab に行があるかどうかにかかわらず、PurchaseOrder_objtab 表のすべての行をフェッチするには、(+) 演算子が必要です。

```
SELECT   po.PONo, po.Cust_ref.CustNo, L.*
 FROM    PurchaseOrder_objtab po, TABLE (po.LineItemList_ntab) (+) L
 WHERE   L.Stock_ref.StockNo = 1004 ;
```

発注書のすべての明細項目にわたる値引きの平均

この要求には、すべての PurchaseOrder_objtab 行のすべてのネストした表 LineItemList_ntab にある行の問合せが必要です。さらに、ネストの解除が必要です。

```
SELECT   AVG(L.DISCOUNT)
 FROM    PurchaseOrder_objtab po, TABLE (po.LineItemList_ntab) L ;
```

削除

次の例は、リレーショナルの例で必要とされた 2 つの削除文と同じ結果になります（9-8 ページの「リレーショナル・モデルでのデータの削除」を参照）。ここでは、明細項目を含む発注情報オブジェクト全体が単一の SQL 操作で削除されます。リレーショナルの場合、発注書の明細項目を明細項目表から削除する必要があり、発注書は発注表から別々に削除する必要があります。

発注書 1001 の削除

```
DELETE
 FROM   PurchaseOrder_objtab
 WHERE  PONo = 1001 ;
```

ユーザー定義型の進化

完全に構築され、完成したアプリケーションでも修正される場合があります。要件が変更され、新しい環境に適用させるために、基盤となるオブジェクト・モデルまたはスキーマを変更したり、予定した作業をより効率的に行うためにオブジェクト・モデルを改良する必要がある場合もあります。

オブジェクト・リレーショナル・アプリケーションの使用中に、設計の改良方法を発見したとします。また、ほとんどの場合、顧客はレコードを起動するたびに購入履歴を参照するとします。現在のオブジェクト・モデルでこれを行うには、顧客および発注書の情報を持つ Customer_objtab および PurchaseOrder_objtab という 2 つの表の結合が必要となります。設計の改良によって、顧客表から直接関連する発注書のデータへのアクセスが可能になります。

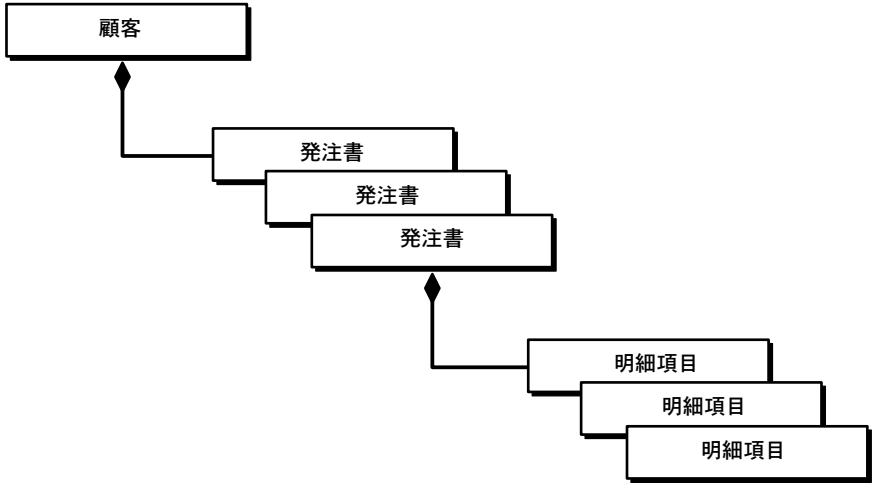
これを行うには、顧客の発注書の情報が、顧客を表すオブジェクト・インスタンスに追加されるように Customer_objtyp を変更する方法があります。つまり、Customer_objtyp に発注書の情報の属性を追加します。複数の発注書の情報を追加するには、属性がコレクション型（ネストした表）である必要があります。

属性の追加は、ユーザー定義型を変更または進化させる方法の 1 つです。型を進化させると、Oracle では、型自体、型のサブタイプ、属性として変更された型を持つ他のオブジェクト型、および変更された型の表や列など、それに依存するすべてのスキーマ・オブジェクトに変更が適用されます。

Customer_objtyp を変更して発注書のネストした表に属性を追加するには、次の手順に従います。

1. 発注書のネストした表に新しい型を作成します。
2. Customer_objtyp を変更し、新しい型に新しい属性を追加します。
3. Customer_objtab オブジェクト表で、新しく追加されたネストした表の記憶表に名前を付け、有効範囲を指定します。

   新しい属性用に Customer_objtab オブジェクト表をアップグレードをすると、発注書自体に明細項目のネストした表が存在するため、2 つのレベルのネストした表（一方がもう一方の内部にある）が追加されます。



発注書のネストした表および明細項目のネストした表の両方に、主キー・ベースの REF が含まれるように有効範囲を指定する必要があります。詳細は、次の項を参照してください。

前述の手順を完了すると、顧客および発注書の情報は、モデルにより論理的に関連付けられ、顧客表に対して、顧客、発注書および明細項目のすべての情報を問い合せることが可能になります。また、顧客表に対して単一の INSERT 文を使用して、新しい顧客の新しい発注書を挿入できます。

顧客型への属性の追加

Customer_objtyp の属性として発注書のネストした表を追加する前に、この種類のネストした表の型を定義する必要があります。次の文で定義します。

```
CREATE TYPE PurchaseOrderList_ntabtyp AS TABLE OF PurchaseOrder_objtyp;
```

これで、ALTER TYPE 文を使用して、この型の属性を Customer_objtyp に追加できます。

```
ALTER TYPE Customer_objtyp
  ADD ATTRIBUTE (PurchaseOrderList_ntab PurchaseOrderList_ntabtyp)
  INVALIDATE;
```

この ALTER TYPE 文は、最後の INVALIDATE オプション以外は単純です。このオプションは、Customer_objtyp を参照する型および表のアップデートと関連します。

変更される型に依存する型または表がある場合、その型に対する ALTER TYPE 文には、CASCADE または INVALIDATE を指定して、依存する型または表への変更の適用方法を指定する必要があります。

- CASCADE は、型の変更を適用する前に依存する型または表の妥当性チェックを行います。チェックは、表のパーティション・キーとして使用されている属性の削除など、変更に無効な操作が伴わないことを確認します。依存する型または表の妥当性チェックに失敗した場合、型の変更は強制終了します。依存するすべての型または表の妥当性チェックが正常に行われた場合、システムは型の変更を伝播するために必要なメタデータおよびデータの変更を遂行します。これには、列の自動追加および削除、ネストした表の記憶表の作成などが含まれます。
- INVALIDATE オプションは、事前の妥当チェックは行わず、依存する型または表の型の変更を直接適用します。これらは、次回のアクセス時に検証されます。この方法で型を変更すると、妥当性チェックにかかる時間が削減されますが、依存表にアクセスしようとしたときに依存表の検証が行えない場合、妥当性チェックに合格するまでデータにアクセスできません。

ALTER TYPE 文で、よりリスクの高い INVALIDATE オプションを使用する理由は、時間の削減ではなく、Customer_objtab 表に追加する必要のある発注書および明細項目の新しいネストした表の記憶表が、システムにより自動的に作成および名前付けされることを防ぐためです。オブジェクト表は、Customer_objtyp 型の依存表です。CASCADE を使用すると、記憶表が自動的に作成され、システム生成された名前が付けられます。REF 列ごとに有効範囲を追加するためにこれらの表を変更する必要があるため、CASCADE は使用しないでください。そのためには、手動で記憶表を設定（Customer_objtab 表に ALTER TABLE 文を使用）し、名前を付けます。この名前を使用すると、複数の ALTER TABLE 文を使用して記憶表を変更し、REF 列に有効範囲を追加できます。

主キーに基づくオブジェクト識別子を持つ表のオブジェクトへの REF をある列に格納するには、それ列の有効範囲にその表を指定する必要があるため、これらの操作が必要です。列の有効範囲に特定の表を指定することは、列のすべての REF がその表のオブジェクトの REF であることを宣言することです。主キー・ベースのオブジェクト識別子は、特定の表で使用する場合のみ一意であることが保証されるため、この宣言が必要となります。必ずしも、すべての表に対して一意ではありません。主キー・ベースの REF（またはユーザー定義 REF）を有効範囲の指定されていない列に挿入しようとすると、「オブジェクト・ビュー REF またはユーザー定義 REF を挿入できません。」というエラーが発生します。

明細項目には、オブジェクト識別子が表の主キーを使用する Stock_objtab 表のオブジェクトへの REF が含まれます。そのため、PurchaseOrder_objtab 表を作成した後、この表の明細項目のネストした表にある記憶表の REF 列に有効範囲を追加する必要があります。また、表 Customer_objtab の明細項目の新しいネストした表に対して同様の操作を繰り返します。

表 Customer_objtab に追加する発注書の新しいネストした表にも同じ操作を行います。発注書は、表 Customer_objtab の顧客を参照します。また、この表のオブジェクト識別子は、主キー・ベースでもあります。

次に示す文は、表 Customer_objtab をアップグレードして、表の宣言された型の変更に対応し、新しいネストした表の記憶表を設定する ALTER TABLE 文です。

> **注意：** ALTER TABLE の前にある 2 つの ALTER TYPE 文は、リリース 2（9.2）の不具合に対する処置です（リリース前に解決する可能性もあります）。これらの文によって、Customer_objtyp の参照依存型が強制的に再コンパイルされます。通常、再コンパイルは自動的に行われます。

```
ALTER TYPE PurchaseOrder_objtyp COMPILE;

ALTER TYPE PurchaseOrderList_ntabtyp COMPILE;

ALTER TABLE Customer_objtab UPGRADE
NESTED TABLE PurchaseOrderList_ntab STORE AS PO_List_nt
  (NESTED TABLE LineItemList_ntab STORE AS Items_List_nt);
```

新しい記憶表の名前は、PO_List_nt および Items_List_nt です。次の文は、これらの表の REF 列の有効範囲に特定の表を指定します。

```
ALTER TABLE PO_List_nt ADD (SCOPE FOR (Cust_Ref) IS Customer_objtab);

ALTER TABLE Items_List_nt ADD (SCOPE FOR (Stock_ref) IS Stock_objtab);
```

ここで、Customer_objtab の顧客に発注書を挿入する前に、次の作業を行います。表の顧客ごとに、PurchaseOrderList_ntabtyp の実際のネストした表をインスタンス化する必要があります。

新しい属性用に表に列を追加すると、既存の行の列値は NULL に初期化されます。つまり、既存の顧客ごとの発注書のネストした表は NULL になり、実際のネストした表は存在せず、空のネストした表も存在しません。顧客ごとのネストした表をインスタンス化する前に、発注書を挿入しようとすると、「NULL の表値を参照しています。」というエラーが発生します。

次の文は、各行を更新して実際のネストした表のインスタンスを含めることで、列が発注書を保持する準備をします。

```
UPDATE Customer_objtab c
  SET c.PurchaseOrderList_ntab = PurchaseOrderList_ntabtyp();
```

この文の PurchaseOrderList_ntabtyp() は、ネストした表の型のコンストラクタ・メソッドへのコールです。発注書が指定されていないこのコールは、空のネストした表を作成します。

マルチレベル・コレクションでの作業

ここまでで、発注書のネストした表を追加するために Customer_objtyp 型を進化させ、ネストした表に発注書を格納できるように表 Customer_objtab を設定しました。次に Customer_objtab に発注書を挿入します。

表 PurchaseOrder_objtab には、すでに 2 つの発注書が存在します。次の 2 つの文は、これらの 2 つの発注書を Customer_objtab にコピーします。

```
INSERT INTO TABLE (
  SELECT   c.PurchaseOrderList_ntab
    FROM   Customer_objtab c
    WHERE  c.CustNo = 1
  )
  SELECT VALUE(p)
    FROM PurchaseOrder_objtab p
    WHERE p.Cust_Ref.CustNo = 1;

INSERT INTO TABLE (
  SELECT   c.PurchaseOrderList_ntab
    FROM   Customer_objtab c
    WHERE  c.CustNo = 2
  )
  SELECT VALUE(p)
    FROM PurchaseOrder_objtab p
    WHERE p.Cust_Ref.CustNo = 2;
```

ネストした表への挿入

前述の INSERT 文には、挿入対象となる表を指定する TABLE 式と、挿入するデータを取得する SELECT の 2 つの主な部分があります。各部分の WHERE 句は、発注書を受け取る顧客オブジェクト（TABLE 式内）、および発注書が選択される顧客（発注書を取得する副問合せ）を指定します。

副問合せの WHERE 句は、ドット表記法を使用して CustNo 属性 p.Cust_Ref.CustNo にナビゲートします。ドット表記法を使用する際は、表別名（この場合は p）が必要となります。表別名を指定せずに Cust_Ref.CustNo を使用するとエラーが発生します。

WHERE 句でドット表記を使用する際は、発注書の REF 属性 Cust_Ref から顧客の CustNo 属性にナビゲートできることにも注意してください。SQL（PL/SQL ではない）は、暗黙的にこのようにドット表記法で使用され REF への参照が解除されます。

INSERT 文の前半の TABLE 式は、式から戻されるコレクションを表として処理します。ここで使用する式は、特定の顧客の発注書のネストした表を挿入先として選択します。

INSERT 文の後半では、VALUE() ファンクションが選択した行をオブジェクトとして戻します。この場合、各行は発注情報オブジェクトであり、独自の明細項目で完結しています。発注書の行は、PurchaseOrder_objtyp 型の 1 つの表からその型の別の表への挿入のために選択されます。

前述の INSERT 文は、PurchaseOrder_objtyp の顧客参照属性を使用して、既存の各発注書がどの顧客のものであるかを識別します。ただし、以前の発注書がすべて発注書表からアップグレード済の Customer_objtab にコピーされたため、発注書のこの顧客参照属性は廃棄されます。これで、発注書は顧客オブジェクト自体に格納されます。

次の ALTER TYPE 文は、顧客参照属性を削除するように PurchaseOrder_objtyp を進化させます。また、この文は、ShipToAddr_obj 属性を冗長（出荷先の住所が、顧客の住所と常に同じであると想定）であるため削除します。

```
ALTER TYPE PurchaseOrder_objtyp
  DROP ATTRIBUTE Cust_ref,
  DROP ATTRIBUTE ShipToAddr_obj
  CASCADE;
```

この例では、CASCADE オプションを使用して、システムで妥当性チェックを実行し、依存する型および表に対し、すべての必要な変更を加えることができました。

新しい発注書への明細項目の挿入

前述の INSERT の例では、VALUE() ファンクションを使用して、明細項目独自のネストした表を完了させるために発注書のネストした表に既存の発注情報オブジェクトを選択および挿入しました。次の例では、発注情報オブジェクトとしてインスタンス化されていない新しい発注書の挿入方法を説明します。この場合、発注書の明細項目のネストした表および各明細項目オブジェクトのデータをインスタンス化する必要があります。（左側の行番号は、説明用に記載されています。）

```
SQL> INSERT INTO TABLE (
  2    SELECT c.PurchaseOrderList_ntab
  3      FROM Customer_objtab c
  4      WHERE c.CustName = 'John Nike'
  5     )
  6    VALUES (1020, SYSDATE, SYSDATE + 1,
  7      LineItemList_ntabtyp(
  8        LineItem_objtyp(1, MAKE_REF(Stock_objtab, 1004), 1, 0),
  9        LineItem_objtyp(2, MAKE_REF(Stock_objtab, 1011), 3, 5),
 10        LineItem_objtyp(3, MAKE_REF(Stock_objtab, 1535), 2, 10)
 11       )
 12     );
```

行 1 ～ 5 は、TABLE 式を使用して、挿入先のネストした表（顧客 John Nike の発注書のネストした表）を選択します。

VALUES 句（行 6 ～ 12）には、新しい発注書の各属性値が含まれます。各属性は次のとおりです。

```
PONo
OrderDate
ShipDate
LineItemList_ntab
```

INSERT 文の行 6 は、PONo、OrderDate および ShipDate という 3 つの発注書属性の値を指定します。

属性値のみが指定されています。発注書コンストラクタは指定されていません。明示的に発注書コンストラクタを指定して、ネストした表の発注書インスタンスをインスタンス化する必要はありません。これは、ネストした表は、発注書のネストした表として宣言されているためです。発注書コンストラクタの指定を省略すると、システムは自動的に発注書をインスタンス化します。ただし、必要に応じてコンストラクタを指定することもできます。この場合、VALUES 句は次のようになります。

```
VALUES (
  PurchaseOrder_objtyp(1020, SYSDATE, SYSDATE + 1,
    LineItemList_ntabtyp(
      LineItem_objtyp(1, MAKE_REF(Stock_objtab, 1004), 1, 0),
      LineItem_objtyp(2, MAKE_REF(Stock_objtab, 1011), 3, 5),
      LineItem_objtyp(3, MAKE_REF(Stock_objtab, 1535), 2, 10)
     )
  )
)
```

行 7 ～ 11 は、明細項目のネストした表にデータをインスタンス化し、指定します。コンストラクタ・メソッド LineItemList_ntabtyp(…) は、3 つの明細項目を含むネストした表のインスタンスを作成します。

明細項目コンストラクタ LineItem_objtyp() は、各明細項目のオブジェクト・インスタンスを作成します。明細項目属性の値は、コンストラクタへの引数として提供されます。

MAKE_REF ファンクションは、明細項目の Stock_ref 属性への REF を作成します。MAKE_REF への引数は、参照する在庫表の名前および在庫品目の主キー値です。ここで MAKE_REF を使用できるのは、在庫表のオブジェクト識別子が主キー・ベースであるためです。そうでない場合、在庫表の行への REF を取得するために副問合せで REF ファンクションを使用する必要があります。

マルチレベルのネストした表の問合せ

埋込みとは逆に、他の最上位列や属性のように SELECT 構文のリストで指定することによって、最上位のネストした表を問い合せることができますが、結果は読みやすくはありません。たとえば、次の問合せは、John Nike の発注書のネストした表を選択します。

```
SELECT c.PurchaseOrderList_ntab
  FROM Customer_objtab c
  WHERE CustName = 'John Nike';
```

この問合せは、次のような結果を戻します。

```
PURCHASEORDERLIST_NTAB(PONO, ORDERDATE, SHIPDATE, LINEITEMLIST_NTAB(LINEITEMNO,
--------------------------------------------------------------------------------
PURCHASEORDERLIST_NTABTYP(PURCHASEORDER_OBJTYP(2001, '25-SEP-01', '20-MAY-97', L
INEITEMLIST_NTABTYP(LINEITEM_OBJTYP(10, 00004A038A00468ED552CE6A5803ACE034080020
B8C834000000142601000100010029000000000090600812A00078401FE0000000B03C20B050000
00000000000000000000000000000000000, 1, 0), LINEITEM_OBJTYP(11, 00004A038A00468ED
552CE6A5803ACE034080020B8C834000000142601000100010029000000000090600812A0007840
1FE0000000B03C20B0C00000000000000000000000000000000000, 2, 1))), PURCHASEORDE
R_OBJTYP(1020, '25-SEP-01', '26-SEP-01', LINEITEMLIST_NTABTYP(LINEITEM_OBJTYP(1,
 00004A038A00468ED552CE6A5803ACE034080020B8C834000000142601000100010029000000000
0090600812A00078401FE0000000B03C20B0500000000000000000000000000000000000000, 1,
0), LINEITEM_OBJTYP(2, 00004A038A00468ED552CE6A5803ACE034080020B8C83400000014260
1000100010029000000000090600812A00078401FE0000000B03C20B0C0000000000000000000000
00000000000000000, 3, 5), LINEITEM_OBJTYP(3, 00004A038A00468ED552CE6A5803ACE0340
80020B8C834000000142601000100010029000000000090600812A00078401FE0000000B03C2102
40000000000000000000000000000000000000, 2, 10))))
```

通常は、REF を表示するのではなく、ネスト解除された形式でインスタンス・データを表示する必要があり、TABLE 式（今回は、問合せの FROM 句）で行うことができます。

たとえば、次の問合せは、John Nike のすべての発注書の PO 番号、発注日および出荷日を選択します。

```
SELECT p.PONo, p.OrderDate, p.Shipdate
FROM Customer_objtab c, TABLE(c.PurchaseOrderList_ntab) p
WHERE c.CustName = 'John Nike';
```

```
      PONO ORDERDATE SHIPDATE
---------- --------- ---------
      2001 25-SEP-01 26-SEP-01
      1020 25-SEP-01 26-SEP-01
```

TABLE 式は、引数としてコレクションを取り、SQL 文で SQL 表のように使用できます。前述の問合せでは、FROM 句の TABLE 式で発注書のネストした表をリストすることにより、一般的な表の列と同様にネストした表の列を選択できます。列は、使用した p という表別名により、ネストした表に属することが識別されます。例に示すとおり、FROM 句の TABLE 式は、独自の表別名を持つことができます。

TABLE 式内では、ネストした表は顧客表の独自の表別名 c により、顧客表 Customer_objtab の列として識別されます。FROM 句では、表 Customer_objtab はそれを参照する TABLE 式の前にあることに注意してください。FROM 句で TABLE 式の左側に表別名を使用することを、左相関といいます。これにより、表および TABLE 式を関連付けることができます。これには、別の TABLE 式の表別名を使用する TABLE 式も含まれます。また、この方法で他のネストした表に埋め込まれているネストした表の列を選択できます。

たとえば、次の問合せは、PO 番号 1020 のすべての明細項目の情報を選択します。

```
SELECT p.PONo, i.LineItemNo, i.Stock_ref.StockNo, i.Quantity, i.Discount
  FROM Customer_objtab c, TABLE(c.PurchaseOrderList_ntab) p,
    TABLE(p.LineItemList_ntab) i
  WHERE p.PONo = 1020;

      PONO LINEITEMNO STOCK_REF.STOCKNO   QUANTITY   DISCOUNT
---------- ---------- ----------------- ---------- ----------
      1020          1              1004          1          0
      1020          2              1011          3          5
      1020          3              1535          2         10
```

この問合せでは、2 つの TABLE 式を使用します。後者は前者を参照しています。明細項目の情報は、外側のネストした表の発注書番号 1020 に属する内側のネストした表から選択されます。

顧客表の列は、SELECT リストでも WHERE 句でも存在しないことに注意してください。顧客表は、ネストした表へのアクセスの開始点を指定するために FROM 句にリストされています。

次に、前述の問合せのバリエーションを示します。このバージョンは、「*」ワイルド・カードを使用して、TABLE 式コレクションのすべての列を指定できることを示します。

```
SELECT p.PONo, i.*
  FROM Customer_objtab c, TABLE(c.PurchaseOrderList_ntab) p,
    TABLE(p.LineItemList_ntab) i
  WHERE p.PONo = 1020;
```

型の継承および代入可能列

アカウント・マネージャを使用して、多数の大規模な通常の顧客を扱うとします。これらの顧客の顧客レコードに対して、アカウント・マネージャの ID のフィールドを追加するとします。

前述の例では、発注書のネストした表の属性を追加する際に、顧客型自体を進化させました。アカウント・マネージャ ID の属性を追加するために、顧客型自体を再度進化させることもできますが、顧客型のサブタイプを作成して、サブタイプのみに属性を追加することもできます。どちらがよいでしょうか。

そのような判断を行うには、新しい属性をベース型のすべてのインスタンス（すべての顧客）に対して適用するか、またはベース型の識別可能なサブクラスのみに適用するかを考慮する必要があります。

すべての顧客には発注書があるため、型自体を変更し、それらに属性を追加する方が適切です。ただし、すべての顧客に対してアカウント・マネージャが存在するわけではありません。実際、アカウント・マネージャは企業顧客にのみ存在します。一般に、顧客型を進化させて顧客に対して無意味な属性を追加するより、識別された特定の顧客に対して新しいサブタイプを作成して、新しい属性を追加する方が適切です。

サブタイプの作成

ベース型がサブタイプを許可する場合のみ、ベース型にサブタイプを作成できます。型のサブタイプ化が可能であるかどうかは、型の FINAL プロパティに依存します。デフォルトでは、新しい型は FINAL として作成されます。これは、この型が最終形であり、この下にサブタイプを作成できないことを表します。サブタイプを作成できる型を作成するには、顧客型を作成した場合と同様に、CREATE TYPE 文で NOT FINAL を指定する必要があります。

サブタイプは、CREATE TYPE 文に UNDER キーワードを使用することによって定義できます。次の文は、新しいサブタイプ Corp_Customer_objtyp を Customer_objtyp の下に作成します。型は、NOT FINAL として作成されているため、必要に応じて後でサブタイプを追加することもできます。

```
CREATE TYPE Corp_Customer_objtyp UNDER Customer_objtyp
  (account_mgr_id     NUMBER(6) ) NOT FINAL;
```

新しいサブタイプを作成するために CREATE TYPE 文を使用する場合、追加する新しい属性およびメソッドのみリストします。サブタイプは既存のすべての属性およびメソッドをベース型から継承するため、指定する必要はありません。新しい属性およびメソッドは、継承された属性およびメソッドの後に追加されます。たとえば、新しい Corp_Customer_objtyp サブタイプの完全なリストは次のようになります。

```
CustNo
CustName
Address_obj
Phonelist_var
PurchaseOrderList_ntab
```

```
Account_mgr_id
```

デフォルトでは、サブタイプのインスタンスは、任意のサブタイプのベース型である任意の列またはオブジェクト表に格納できます。ベース型スロットにサブタイプ・インスタンスを格納する機能を、**代入性**といいます。NOT SUBSTITUTABLE として明示的に宣言されている場合を除いて、列および表は代入可能です。システムは、サブタイプ属性や各行に格納されたインスタンスの型 ID の非表示列に新しい列を自動的に追加します。

実際には、FINAL 型のサブタイプを作成することは可能ですが、まず ALTER TYPE 文を使用して型を FINAL 型から NOT FINAL に進化させる必要があります。変更された型の既存の列および表で新しいサブタイプのインスタンスを格納する場合は、ALTER TYPE 文で CASCADE オプション CONVERT TO SUBSTITUTABLE を指定します。

> **参照：** 第 6 章の「型進化」を参照してください。

サブタイプの挿入

列またはオブジェクト表が代入可能の場合、列または表の宣言された型のインスタンスのみでなく、宣言された型のどのサブタイプのインスタンスにも挿入できます。表 Customer_objtab の場合、これは一般および企業のすべての種類の顧客情報を格納するために表を使用できることを表します。ただし、サブタイプへの情報の挿入方法には重要な違いが 1 つあります。それは、サブタイプのコンストラクタを明示的に指定する必要があるということです。コンストラクタの使用は、列または表の宣言された型のインスタンスのみのオプションです。

たとえば、次の文は、新しい一般顧客 William Kidd を挿入します。

```
INSERT INTO Customer_objtab
  VALUES (
    3, 'William Kidd',
    Address_objtyp('43 Harbor Drive', 'Redwood Shores', 'CA', '95054'),
    PhoneList_vartyp('415-555-1212'),
    PurchaseOrderList_ntabtyp()
  );
```

VALUES 句には、各 Customer_objtyp 属性のデータが含まれますが、Customer_objtyp コンストラクタは省略されています。ここでは、表の宣言された型は Customer_objtyp であるため、コンストラクタはオプションとなります。ネストした表の属性の場合、コンストラクタ PurchaseOrderList_ntabtyp() は空のネストした表を作成しますが、どの発注書にもデータは指定されません。

次の文は、同じ表に新しい企業顧客を挿入します。コンストラクタ Corp_Customer_objtyp() およびアカウント・マネージャ ID に追加のデータ値 531 が使用されていることに注意してください。

```
INSERT INTO Customer_objtab
  VALUES (
    Corp_Customer_objtyp(        -- Subtype requires a constructor
```

```
      4, 'Edward Teach',
      Address_objtyp('65 Marina Blvd', 'San Francisco', 'CA', '94777'),
      PhoneList_vartyp('415-555-1212', '416-555-1212'),
      PurchaseOrderList_ntabtyp(), 531
    )
  );
```

次の文は、これらの 2 種類の新しい顧客の発注書を挿入します。新しい顧客を挿入する文とは異なり、発注書を挿入するための 2 つの文は構造的に同じ（ただし、発注書の明細項目数を除く）です。

```
            -- Insert PO for ordinary customer

INSERT INTO TABLE (
  SELECT c.PurchaseOrderList_ntab
    FROM Customer_objtab c
    WHERE c.CustName = 'William Kidd'
   )
  VALUES (1021, SYSDATE, SYSDATE + 1,
    LineItemList_ntabtyp(
      LineItem_objtyp(1, MAKE_REF(Stock_objtab, 1535), 2, 10),
      LineItem_objtyp(2, MAKE_REF(Stock_objtab, 1534), 1, 0)
     )
   );

            -- Insert PO for corporate customer

INSERT INTO TABLE (
  SELECT c.PurchaseOrderList_ntab
    FROM Customer_objtab c
    WHERE c.CustName = 'Edward Teach'
   )
  VALUES (1022, SYSDATE, SYSDATE + 1,
    LineItemList_ntabtyp(
      LineItem_objtyp(1, MAKE_REF(Stock_objtab, 1011), 1, 0),
      LineItem_objtyp(2, MAKE_REF(Stock_objtab, 1004), 3, 0),
      LineItem_objtyp(3, MAKE_REF(Stock_objtab, 1534), 2, 0)
     )
   );
```

代入可能な列の問合せ

代入可能な列または表には、複数のデータ型を含めることができます。たとえば、顧客表の単一の問合せで、すべての種類の顧客情報を取得できます。また、特定の種類の顧客または特定の種類の顧客の特定の属性の情報のみを取得することもできます。

次の例に、代入可能な表または列から必要な情報を取得するための手法を示します。

すべての顧客の選択

次の問合せでは、VALUE() ファンクションを使用して、表のすべての種類の顧客のインスタンスを選択します。

```
SELECT VALUE(c)
  FROM Customer_objtab c;
```

すべての企業顧客（およびそのサブタイプ）の選択

次の問合せでは、IS OF 述語を含む WHERE 句を使用して、企業顧客以外の顧客にフィルタ処理を行います。つまり、問合せは、すべての種類の企業顧客を戻しますが、その他の種類の顧客のインスタンスは戻しません。

```
SELECT VALUE(c)
  FROM Customer_objtab c
  WHERE VALUE(c) IS OF (Corp_Customer_objtyp);
```

すべての企業顧客（サブタイプなし）の選択

この問合せは、Corp_Customer_objtyp のサブタイプにフィルタ処理を行うために IS OF 述語に ONLY キーワードを追加することを除いて前述の例と同じです。特定の型が Corp_Customer_objtyp であるインスタンスのみの行が戻されます。

```
SELECT p.PONo
  FROM Customer_objtab c, TABLE(c.PurchaseOrderList_ntab) p
  WHERE VALUE(c) IS OF (ONLY Corp_Customer_objtyp);
```

企業顧客のみの PONo の選択

次の問合せは、TABLE 式を使用して（発注書のネストした表から）発注書番号を取得します。この属性は、すべての種類の顧客にありますが、WHERE 句で、企業顧客のみに検索を制限します。

```
SELECT p.PONo
  FROM Customer_objtab c, TABLE(c.PurchaseOrderList_ntab) p
  WHERE VALUE(c) IS OF (Corp_Customer_objtyp);
```

サブタイプ属性の選択

次の問合せはアカウント・マネージャ ID のデータを戻します。この属性は、企業顧客サブタイプのみが所有します。表の宣言された型には存在しません。

この問合せでは、サブタイプ属性 Account_mgr_id にアクセスするために、TREAT() ファンクションを使用して、システムが各顧客を企業顧客として扱うようにします。

```
SELECT CustName, TREAT(VALUE(c) AS Corp_Customer_objtyp).Account_mgr_id
```

```
  FROM Customer_objtab c
  WHERE VALUE(c) IS OF (ONLY Corp_Customer_objtyp);
```

ここでは、Account_mgr_id が表の宣言された型 Customer_objtyp の属性ではないため、TREAT() が必要です。SELECT 構文のリストのように属性をリストすると、次のような問合せは、「列名が無効です。」というエラーを戻します。これは Corp_Customer_objtyp のインスタンス以外を除外する WHERE 句でも発生します。WHERE 句では、結果から行を除外するのみであるため、これでは不十分です。

```
    -- Returns error, "invalid column name" for Account_mgr_id

SELECT CustName, Account_mgr_id
  FROM Customer_objtab
  WHERE VALUE(c) IS OF (ONLY Corp_Customer_objtyp);
```

各インスタンスの（サブ）タイプの検出

代入可能な各列またはオブジェクトには、各行のインスタンスの型を識別する関連付けられた非表示型 ID 列が存在します。USER_TYPES カタログ・ビューを使用すると、型 ID を参照できます。

SYS_TYPEID() ファンクションは、特定のインスタンスの型 ID を戻します。次の問合せは SYS_TYPEID() および USER_TYPES カタログ・ビューの結合を使用して、表 Customer_objtab の各顧客のインスタンスの型名を戻します。

```
SELECT c.CustName, u.TYPE_NAME
  FROM Customer_objtab c, USER_TYPES u
  WHERE SYS_TYPEID(VALUE(c)) = u.TYPEID;

CUSTNAME
--------------------------------------------------------------------------------
TYPE_NAME
------------------------------
Jean Nance
CUSTOMER_OBJTYP

John Nike
CUSTOMER_OBJTYP

William Kidd
CUSTOMER_OBJTYP

Edward Teach
CORP_CUSTOMER_OBJTYP
```

参照： SYS_TYPEID()、VALUE() および TREAT() の詳細は、第 2 章の「オブジェクトに便利なファンクションおよび述語」を参照してください。

Oracle Objects for OLE でのオブジェクトの操作

Windows システム上では、OO4O を使用して、Visual Basic、または Excel などの COM プロトコルをサポートする他の環境で、オブジェクト指向データベース・プログラムを作成できます。

次の各例では、冒頭に、データベースに接続するための類似のヘッダーがあります。その後、それぞれの例で、オブジェクト・データに対して行われる操作ごとに異なる内容が続きます。

データの選択

次に、SELECT 操作を実行するボタンのイベント・ハンドラを示します。

- データベースから行集合を取得します。各行には、いくつかのリレーショナル列およびオブジェクトであるいくつかの列が含まれています。
- CUSTREF 列の名前を使用して、その値（オブジェクト）を取り出します。
- こうすると、ドット表記法を使用して、オブジェクトの属性にアクセスできます。汎用オブジェクト型 OraObject として変数を定義します。OraObject が実際のオブジェクトでインスタンス化されると、対応するオブジェクト型のプロパティを獲得します。

```
Private Sub obj_select_Click()
 Dim OO4OSession As OraSession
 Dim InvDB As OraDatabase
 Dim PurchaseOrder As OraDynaset
 Dim CustomerInfo As OraRef
 Dim LineItemsList As OraCollection
 Dim LineItem As OraObject
 Dim ShipToAddr As OraObject
 Dim StockInfo As OraRef
 Dim CustomerAddr As OraObject

 'Create the OraSession Object.
 Set OO4OSession = CreateObject("OracleInProcServer.XOraSession")

 'Create the OraDatabase Object by opening a connection to Oracle.
 Set InvDB = OO4OSession.OpenDatabase("exampledb", "scott/tiger", 0&)

 'Select from purchase_tab
 Set PurchaseOrder = InvDB.CreateDynaset("select * from purchase_tab", 0&)

 'Get the custref attribute from PurchaseOrder
```

```
Set CustomerInfo = PurchaseOrder.Fields("custref").Value

' Accessing attributes CustomerInfo object

'Display custno,custname,phonelist attibutes of CustomerInfo
MsgBox CustomerInfo.custno
MsgBox CustomerInfo.custname

'Get address and phonelist attibutes of CustomerInfo
Set CustomerAddr = CustomerInfo.Address

'Display all the atributes of CustomerAddr
MsgBox CustomerAddr.Street
MsgBox CustomerAddr.State
MsgBox CustomerAddr.Zip

'  Accessing elements of LineItemsList Object

'Get line_item_list attribute from PurchaseOrder
Set LineItemsList = PurchaseOrder.Fields("line_item_list").Value

'Get LineItem object element from LineItemList collection
 Set LineItem = LineItemsList(1)

'Display lineitemno,quantity,discount attibutes
MsgBox LineItem.lineitemno
MsgBox LineItem.quantity
MsgBox LineItem.discount

'Access stockref attribute of LineItem
Set StockInfo = LineItem.Stockref

'Display stockno,cost,tax_code of StockInfo
MsgBox StockInfo.stockno
MsgBox StockInfo.cost
MsgBox StockInfo.tax_code

End Sub
```

データの挿入

次に、データベースから行集合を取り出して、新しい行を追加するプログラムを示します。

- 適切なオブジェクト型のオブジェクトをいくつか作成します。
- オブジェクトにサンプル値を移入します。

- 発注書表に新しい行を作成し、その列に値を指定します。オブジェクトではない列は、直接設定できます。オブジェクトである列は、VALUE フィールドを使用して設定する必要があります。

```
Dim OO4OSession As OraSession
Dim InvDB As OraDatabase
Dim PurchaseOrder As OraDynaset
Dim CustomerInfo As OraRef
Dim LineItemsList As OraCollection
Dim LineItem As OraObject
Dim ShipToAddr As OraObject
Dim StockInfo As OraRef
Dim CustomerAddr As OraObject

 'Create the OraSession Object.
 Set OO4OSession = CreateObject("OracleInProcServer.XOraSession")

 'Create the OraDatabase Object by opening a connection to Oracle.
 Set InvDB = OO4OSession.OpenDatabase("exampledb", "scott/tiger", 0&)

 'Select from purchase_tab
 Set PurchaseOrder = InvDB.CreateDynaset("select * from purchase_tab", 0&)

 '  Step 1  - Creating CustomerInfo ref object

 'select a ref from customer_tab for custono 2
 Set CustomerDyn = InvDB.CreateDynaset("select REF(C) from customer_tab c
where c.custno  = 2", 0&)

 'get the CustomerInfo ref object
 Set CustomerInfo = CustomerDyn.Fields(0).Value

 ' Step 2  - Creating LineItemsList object

 ' Create a new line_items_list object
 Set LineItemsList = InvDB.CreateOraObject("line_item_list_t")

 ' Create a new line_items object
 Set LineItem = InvDB.CreateOraObject("line_item_t")

 'set attributes of LineItem object
 LineItem.lineitemno = 2
 LineItem.quantity = 15
 LineItem.discount = 30
 LineItem.Stockref = Null

 'set the LineItem to first element of LineItemList
```

```
LineItemsList(1) = LineItem

' Step 3 - Creating ShipToAddr object

' create a shiptoaddr object
Set ShipToAddr = InvDB.CreateOraObject("address_t")

'set the attributes of ShipToAddr Object
ShipToAddr.city = "Belmont"
ShipToAddr.Street = "Continentals way"
ShipToAddr.Zip = "94002"
ShipToAddr.State = "CA"

' Start the AddNew operation on PurchaseOrder dynaset

PurchaseOrder.AddNew

 PurchaseOrder.Fields("pono").Value = 1002
 PurchaseOrder.Fields("orderdate").Value = "5/15/99"
 PurchaseOrder.Fields("shipdate").Value = "6/15/99"

 'set the custref field to CustomerInfo object created in step1
 PurchaseOrder.Fields("custref").Value = CustomerInfo

 'set the line_item_list  field to LineItemslist object created in step2
 PurchaseOrder.Fields("line_item_list").Value = LineItemsList

 'set the shiptoaddr  field to ShipToAddr  object created in step3
 PurchaseOrder.Fields("shiptoaddr").Value = ShipToAddr

' Call the update method on Purchaseorder Dynaset which inserts a new row
' in purchase_tab table

PurchaseOrder.Update
```

データの更新

次に、データベースから行をいくつか取り出して、特定のデータを更新するプログラムを示します。

- 単一行を戻す問合せを使用して、発注書を選択します。
- 個々のデータ項目を取得して、その他の表および元の発注書から操作します。
- 更新のために発注書の行をロックし、新しい値を入れます。

```
Dim OO4OSession As OraSession
Dim InvDB As OraDatabase
Dim PurchaseOrder As OraDynaset
Dim CustomerInfo As OraRef
Dim LineItemsList As OraCollection
Dim LineItem As OraObject
Dim ShipToAddr As OraObject
Dim StockInfo As OraRef
Dim CustomerAddr As OraObject

'Create the OraSession Object.
Set OO4OSession = CreateObject("OracleInProcServer.XOraSession")

'Create the OraDatabase Object by opening a connection to Oracle.
Set InvDB = OO4OSession.OpenDatabase("exampledb", "scott/tiger", 0&)

'Select from purchase_tab for pono 1002
Set PurchaseOrder = InvDB.CreateDynaset("select * from purchase_tab where
pono = 1002", 0&)

'Create a StockInfo from stock_tab for stockno 1535
Set StockDyn = InvDB.CreateDynaset("select REF(s) from stock_tab s where
s.stockno = 1535", 0&)
Set StockInfo = StockDyn.Fields(0).Value

'Get line_item_list attribute from PurchaseOrder
Set LineItemsList = PurchaseOrder.Fields("line_item_list").Value

'Get LineItem object element from LineItemList collection
Set LineItem = LineItemsList(1)

'Start the edit operation on PurchaseOrder dynaset
PurchaseOrder.Edit

' Set the StockInfo object created in Step1 to stockref attribute
' of LineItem
LineItem.Stockref = StockInfo
PurchaseOrder.Update
```

メソッド・ファンクションのコール

次に、発注書を取り出してメンバー・ファンクション TOTAL_VALUE をコールし、発注書の一部である明細項目のコストを合計するプログラムを示します。

- 発注書表から 1 行を選択します。結果がオブジェクトとして戻るように、VALUE を選択していることに注意してください。

- 発注書オブジェクト（結果行の 0（ゼロ）列目）へのポインタを取得します。このポインタは、後で PL/SQL ストアド・プロシージャに渡され、Java メソッドまたは C++ メソッドの SELF ポインタをシミュレートします。
- 暗黙的な SELF パラメータおよびメソッド・ファンクションの戻り値に対応するパラメータのリストを作成します。各パラメータに、バインド変数、値、モードおよび型を指定します。
- メソッド・ファンクションに対応するストアド・プロシージャをコールして、結果を TOTALVALUE バインド変数に格納します。
- 結果を使用するには、パラメータ・リストから戻り値を取り出します。

```
Dim OO4OSession As OraSession
Dim InvDB As OraDatabase
Dim PurchaseOrderObj As OraDynaset

'Create the OraSession Object.
Set OO4OSession = CreateObject("OracleInProcServer.XOraSession")

'Create the OraDatabase Object by opening a connection to Oracle.
Set InvDB = OO4OSession.OpenDatabase("exampledb", "scott/tiger", 0&)

'Select from purchase_tab
Set PurchaseOrderDyn = InvDB.CreateDynaset("select VALUE(p)  from
purchase_tab p where p.pono = 1001", 0&)

'Get the PurchaseOrderObj
Set PurchaseOrderObj = PurchaseOrderDyn.Fields(0).Value

'Create a OraParameter object for purchase_order_t object and set it to
PurchaseOrder
InvDB.Parameters.Add "PURCHASEORDER", PurchaseOrderObj, ORAPARM_BOTH,
ORATYPE_OBJECT, "PURCHASE_ORDER_T"

'Create a parameter for total_value return
InvDB.Parameters.Add "TOTALVALUE", "", ORAPARM_OUTPUT

'Execute a member method
InvDB.ExecuteSQL ("BEGIN :TOTALVALUE :=
PURCHASE_ORDER_T.TOTAL_VALUE(:PURCHASEORDER); END;")

'Display the totalvalue
MsgBox InvDB.Parameters("TOTALVALUE").Value
```

索引

A

C

D

E

F

I

J

N

O

P

R

S

T

U

V

W

あ

い

え

お

か

き

け

こ

さ

し

す

せ

そ

た

て

と

な

ね

は

ひ

ふ

へ

ほ

ま

め

ゆ

り

れ

ろ

わ