

Oracle9i OLAP

開発者ガイド -Oracle OLAP API

リリース 2 (9.2)

2003 年 5 月

部品番号 : J07216-01

ORACLE®

Oracle9i OLAP 開発者ガイド -Oracle OLAP API, リリース 2 (9.2)

部品番号 : J07216-01

原本名 : Oracle9i OLAP Developer's Guide to the OLAP API, Release 2 (9.2)

原本部品番号 : A95297-01

Copyright © 2000, 2002, Oracle Corporation. All rights reserved.

Printed in Japan.

制限付権利の説明

プログラム（ソフトウェアおよびドキュメントを含む）の使用、複製または開示は、オラクル社との契約に記された制約条件に従うものとします。著作権、特許権およびその他の知的財産権に関する法律により保護されています。

当プログラムのリバース・エンジニアリング等は禁止されています。

このドキュメントの情報は、予告なしに変更されることがあります。オラクル社は本ドキュメントの無謬性を保証しません。

* オラクル社とは、Oracle Corporation（米国オラクル）または日本オラクル株式会社（日本オラクル）を指します。

危険な用途への使用について

オラクル社製品は、原子力、航空産業、大量輸送、医療あるいはその他の危険が伴うアプリケーションを用途として開発されておりません。オラクル社製品を上述のようなアプリケーションに使用することについての安全確保は、顧客各位の責任と費用により行ってください。万一かかる用途での使用によりクレームや損害が発生いたしましても、日本オラクル株式会社と開発元である Oracle Corporation（米国オラクル）およびその関連会社は一切責任を負いかねます。当プログラムを米国国防総省の米国政府機関に提供する際には、『Restricted Rights』と共に提供してください。この場合次の Notice が適用されます。

Restricted Rights Notice

Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

このドキュメントに記載されているその他の会社名および製品名は、あくまでその製品および会社を識別する目的にのみ使用されており、それぞれの所有者の商標または登録商標です。

目次

はじめに	ix
対象読者	x
このマニュアルの構成	x
関連文書	xi
表記規則	xii

1 OLAP API の概要

OLAP API の概要	1-2
多次元の概念および OLAP API	1-2
OLAP API を介してアプリケーションでアクセス可能なデータのタイプ	1-3
OLAP API を介してアプリケーションで実行可能なタスク	1-4
OLAP API 開発のコンテキスト	1-4
OLAP API を介したデータおよびメタデータへのアクセス	1-4
OLAP API の MDM モデル	1-5
OLAP API を介したデータへのアクセス	1-5
ユーザーの接続要求	1-6
OLAP API クライアント・ソフトウェア	1-7
ソフトウェア構成	1-7
OLAP API クライアント・ソフトウェアの使用要件	1-7
OLAP API アプリケーションの開発	1-7
手順 1: 一般的な設計の問題の決定	1-8
手順 2: エンド・ユーザーの問合せに対する要件の決定	1-8
手順 3: エンド・ユーザーの問合せを作成する OLAP API の Template オブジェクトの設計	1-9
手順 4: アプリケーションの Java コードの作成およびテスト	1-10
手順 5: ユーザーへのアプリケーションのデプロイ	1-10

OLAP API アプリケーションが実行するタスク	1-11
タスク 1: データ・ストアへの接続	1-11
タスク 2: 使用可能なメタデータの検出	1-11
タスク 3: 問合せを介したデータの選択および計算	1-12
タスク 4: 問合せ結果の取得	1-12

2 OLAP API メタデータの理解

OLAP API メタデータの概要	2-2
データの準備	2-2
メタデータの準備	2-2
OLAP メタデータ・オブジェクト	2-2
OLAP メタデータ内のディメンション	2-3
OLAP メタデータ内のメジャー	2-3
OLAP メタデータ内のメジャー・フォルダ	2-3
OLAP API の MDM メタデータ・オブジェクトの概要	2-5
MDM オブジェクトへの OLAP メタデータ・オブジェクトのマッピング	2-6
MdmSchema クラス	2-6
MdmSource クラス	2-7
MdmDimension クラス	2-8
MdmDimension の説明	2-8
MdmDimensionDefinition が保持する情報	2-9
MdmDimensionMemberType が保持する情報	2-9
MdmLevel クラス	2-10
MdmLevel の説明	2-10
MdmLevel の要素	2-10
MdmHierarchy クラス	2-11
MdmHierarchy の説明	2-11
レベル MdmHierarchy の要素	2-12
共用体 MdmHierarchy の要素	2-15
MdmListDimension クラス	2-17
MdmListDimension の説明	2-17
MdmListDimension の要素	2-17
MdmMeasure クラス	2-17
MdmMeasure の説明	2-17
MdmMeasure の要素	2-18

MdmAttribute クラス	2-20
MdmAttribute の説明	2-20
MdmAttribute の要素	2-21
MDM メタデータ・オブジェクトのデータ型およびタイプ	2-22
MDM メタデータ・オブジェクトのデータ型	2-22
MdmSource のデータ型の取得	2-23
MDM メタデータ・オブジェクトのタイプ	2-24
MdmSource のタイプの取得	2-26

3 データ・ストアへの接続

接続処理の概要	3-2
接続手順	3-2
接続の前提条件	3-2
接続の確立	3-2
手順 1: JDBC ドライバのロード	3-3
手順 2: DriverManager からの接続の取得	3-3
手順 3: TransactionProvider の作成	3-3
手順 4: DataProvider の作成	3-4
既存の接続の取得	3-4
接続を介した DML コマンドの実行	3-4
接続のクローズ	3-5

4 使用可能なメタデータの検出

メタデータの検出手順の概要	4-2
MDM メタデータ	4-2
メタデータ検出の目的	4-2
メタデータの検出手順	4-2
メタデータの検出および問合せの作成	4-3
MdmMetadataProvider の作成	4-3
ルート MdmSchema の取得	4-3
ルート MdmSchema の機能	4-4
getRootSchema メソッドのコール	4-6
ルート MdmSchema のコンテンツの取得	4-6
MdmSchema の MdmDimension オブジェクトの取得	4-6
MdmSchema のサブスキーマの取得	4-6

サブスキーマのコンテンツの取得	4-6
メジャー MdmDimension およびそのコンテンツの取得	4-6
メタデータ・オブジェクトの特性の取得	4-7
MdmMeasure の MdmDimension オブジェクトの取得	4-7
MdmDimension の関連オブジェクトの取得	4-7
メタデータ・オブジェクトの Source の取得	4-8
メタデータ検出のサンプル・コード	4-9
SampleMetadataDiscoverer プログラムのコード	4-9
SampleMetadataDiscoverer プログラムの出力	4-15

5 問合せの概要

Source オブジェクトの特性	5-2
Source の型	5-2
Source の構造: 入力および出力	5-3
Source オブジェクトの作成	5-4
メタデータ・オブジェクトからの Source オブジェクトの取得	5-5
Source メソッドを使用した新しい Source オブジェクトの作成	5-6
単純な非ディメンション Source オブジェクトの作成	5-7
OLAP API データ型を表す Source オブジェクトの作成	5-8

6 Source メソッドを使用した問合せの作成

Source 値に基づいた選択	6-2
出力値に基づいた選択	6-2
join メソッドを使用した入力から出力への切替え	6-3
Source の構造に対する入出力の順序の影響	6-3
出力値および Source 値に基づいた選択: 例	6-5
ランクに基づいた値の選択	6-5
値の位置の検索	6-5
昇順または降順でランキングされた値	6-8
他の Source の値と同じ順序または逆の順序でランキングされた値	6-8
最低ランキング	6-9
最高ランキング	6-9
平均ランキング	6-9
パック・ランキング	6-10
パーセンタイル・ランキング	6-10

nTile ランキング	6-11
階層での位置付けに基づいた値の選択	6-11
デフォルトの階層を表すプライマリ Source の作成	6-11
親子関係を表すプライマリ Source の作成	6-12
他の関係を表す Source オブジェクトの作成	6-13
階層のドリルダウン: 例	6-13
セルフ・リレーション Source の作成	6-14
数値分析の実行	6-16
数値操作の実行	6-16
数値比較の実行	6-18
標準の数値関数の使用	6-19
集計メソッドの使用	6-20
独自の数値関数の作成	6-23
文字列値の操作	6-25

7 TransactionProvider の使用

Transaction 内での問合せの作成	7-2
Transaction オブジェクトのタイプ	7-2
Transaction の準備およびコミット	7-3
Transaction および Template オブジェクト	7-4
子 Transaction の開始	7-5
Transaction のロールバック	7-7
現行の Transaction の取得および設定	7-8
TransactionProvider オブジェクトの使用	7-8

8 Cursor クラスおよび Cursor の概念

OLAP API の Cursor オブジェクトの概要	8-2
Cursor を作成できない Source	8-3
Cursor オブジェクトおよび Transaction オブジェクト	8-3
Cursor クラス	8-3
Cursor の構造	8-5
Cursor の動作の指定	8-7
CursorManagerSpecification クラス	8-7
CursorSpecification クラス	8-8
CursorInput クラス	8-10

CursorManager クラス	8-10
CursorManager の CursorManagerSpecification の更新	8-10
CursorManager のクラス階層	8-11
CursorManagerUpdateListener クラス	8-12
CursorManagerUpdateEvent クラス	8-13
Cursor の位置およびエクステンツ	8-13
ValueCursor の位置	8-14
CompoundCursor の位置	8-14
Cursor 内での親の開始 / 終了位置	8-19
Cursor のエクステンツの概要	8-22
フェッチ・サイズおよびフェッチ・ブロック	8-23
フェッチ・ブロックの形式の決定	8-24
フェッチ・ブロックの共有	8-25

9 問合せ結果の取得

問合せ結果の取得	9-2
Cursor からの値の取得	9-3
様々なデータ表示のための CompoundCursor のナビゲート	9-8
Cursor の動作の指定	9-16
エクステンツおよび値の開始 / 終了位置の計算	9-18
フェッチ・サイズおよびフェッチ・ブロックの指定	9-21

10 動的問合せの作成

Template オブジェクトの概要	10-2
動的 Source の作成	10-2
ユーザー・インタフェース要素の OLAP API オブジェクトへの変換	10-3
Template および関連クラスの概要	10-3
動的 Source を作成するクラス間の関連	10-4
Template クラス	10-6
MetadataState インタフェース	10-6
SourceGenerator インタフェース	10-6
DynamicDefinition クラス	10-7
Template の設計および実装	10-7
Template のクラスの実装	10-8
Template を使用するアプリケーションの実装	10-13

A 開発環境のセットアップ

概要	A-2
必要なソフトウェア	A-2
アプリケーション開発コンピュータのセットアップ	A-3
jar ファイルのインストール	A-3
OLAP API Javadoc のインストール	A-4
アプリケーションのデプロイに対する考慮点	A-4

索引

はじめに

このマニュアルでは、Java プログラマを対象に Oracle OLAP API (Oracle OLAP 用の Java Application Programming Interface) について説明します。OLAP API は、Oracle OLAP を介して、Oracle データベースに格納されたデータへのアクセスを提供します。OLAP API のデータの問合せ、操作および表示機能は、特にオンライン分析処理を実行するアプリケーションに適しています。

ここでは、次の項目について説明します。

- [対象読者](#)
- [このマニュアルの構成](#)
- [関連文書](#)
- [表記規則](#)

対象読者

このマニュアルは、Oracle OLAP を使用して分析を実行するアプリケーションを作成する Java プログラマを対象としています。

このマニュアルを読む前に、Java、リレーショナル・データベース管理システム、データ・ウェアハウスおよび Oracle OLAP とオンライン分析処理（OLAP）の概念を理解しておく必要があります。

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「OLAP API の概要」

この章では、Java アプリケーションに OLAP API の使用を計画しているアプリケーション開発者を対象に、OLAP API の概要を示します。

第 2 章「OLAP API メタデータの理解」

この章では、OLAP API が提供するメタデータ・オブジェクトについて説明し、これらのオブジェクトと、データベース管理者が OLAP メタデータ API を使用してデータを準備する際に指定するメタデータ・オブジェクトの関係について説明します。

第 3 章「データ・ストアへの接続」

この章では、OLAP API を介してデータ・ストアに接続する方法について説明します。

第 4 章「使用可能なメタデータの検出」

この章では、OLAP API を介してデータ・ストア内のメタデータを検出する手順について説明します。

第 5 章「問合せの概要」

この章では、問合せの作成時に使用するデータ・セットの仕様で、OLAP API オブジェクトである Source オブジェクトについて説明します。

第 6 章「Source メソッドを使用した問合せの作成」

この章では、Source メソッドを使用して問合せを作成する方法について説明します。

第 7 章「TransactionProvider の使用」

この章では、Oracle OLAP API の Transaction および TransactionProvider インタフェースについて説明します。また、アプリケーションにおいてこれらのインタフェースの実装を使用する方法についても説明します。DataProvider を作成する前に、TransactionProvider を作成しておく必要があります。また、導出 Source の Cursor

を作成する前に、`TransactionProvider` のメソッドを使用して、`Transaction` を準備し、コミットしておく必要もあります。

第 8 章「Cursor クラスおよび Cursor の概念」

この章では、問合せの結果を取得してそれにアクセスするために使用する、Oracle OLAP API の `Cursor` クラスおよびその関連クラスについて説明します。また、`Cursor` の位置、フェッチ・サイズおよびエクステントの概念についても説明します。

第 9 章「問合せ結果の取得」

この章では、Oracle OLAP API の `Cursor` を使用して問合せの結果を取得する方法、その結果にアクセスする方法、および結果の表示方法にあわせて `Cursor` の動作をカスタマイズする方法について説明します。

第 10 章「動的問合せの作成」

この章では、動的問合せの作成に使用する Oracle OLAP API の `Template` クラスおよびその関連クラスについて説明します。また、それらのクラスの実装例を示します。

付録 A「開発環境のセットアップ」

この付録では、OLAP API を使用するアプリケーションの開発環境のセットアップ手順について説明します。

関連文書

詳細は、次の Oracle リソースを参照してください。

- Oracle9i OLAP API Javadoc

Oracle OLAP API である Java パッケージのリファレンス情報を示します。

- 『Oracle9i OLAP ユーザーズ・ガイド』

Oracle OLAP を使用方法について説明します。アプリケーション開発およびシステム管理に使用する基本ツールや、ビジネス分析および多次元問合せの基盤となる基本的な概念について説明します。

- 『Oracle9i OLAP 開発者ガイド - Oracle OLAP DML』

アプリケーション開発者を対象に、OLAP DML を使用して複雑なデータ分析タスク（予測、モデル、アロケーション、特定の非加算的な集計など）を実行する方法について説明します。

- 『Oracle9i JDBC 開発者ガイドおよびリファレンス』

Java プログラムからデータにアクセスするための基礎となり、この Java 標準に対する Oracle 固有の拡張機能を提供する、Oracle の Java Database Connectivity (JDBC) 製品に関するタスク指向の情報およびリファレンス情報を示します。

- 『Oracle9i データ・ウェアハウス・ガイド』

OLAP ソリューションをサポートするためのデータ・ウェアハウスの作成時のデータベース構造、概要および問題点について説明します。

このマニュアルの多くの例で、Oracle のインストール時にデフォルトとして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。スキーマの作成方法および使用方法の詳細は、『Oracle9i 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/documentation/>

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す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](#)
- [コード例中の表記規則](#)
- [Windows オペレーティング・システムの表記規則](#)

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記載されている用語（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。 Source クラスおよびそのサブクラスのメソッドは、新しい Source オブジェクト（ 導出 Source オブジェクトともいう）を戻します。

規則	意味	例
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。	getHierarchyType メソッドの戻り値は LEVEL_HIERARCHY です。
固定幅フォントの小文字	固定幅フォントの小文字は、Java プログラム名、ファイル名、パス名および URL を示します。	/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。
固定幅フォントの大文字と小文字の混在	固定幅フォントの大文字と小文字の混在は、クラスおよびインタフェースの名前を示します。また、変数、メソッドおよびパッケージの複合語の名前も示します。クラス名およびインタフェース名の先頭は大文字です。すべての複合語の名前では、2 番目以降の用語の先頭も大文字になります。	アプリケーションは、MdmMetadataProvider の getRootSchema メソッドを使用してメタデータにアクセスします。
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。 Uold_release.SQL を実行します。ここで、old_release とはアップグレード前にインストールしたリリースを示します。

コード例中の表記規則

コード例は、Java、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、次の例に示すとおり通常のテキストと区別されます。

```
Source unitCost = mdmUnitCost.getSource;
```

次の表に、Java コード例で使用される表記規則を説明します。

規則	意味
{ }	中カッコは、文の 1 つのブロックを囲みます。
//	二重スラッシュは、1 行のコメントを示します。コメントは、行の終わりまで続きます。

規則	意味
/* */	スラッシュとアスタリスクは、複数行のコメントを囲みます。コメントは、複数行にまたがることができます。
...	水平の省略記号は、本文に関連しない文または句が省略されていることを示します。

Windows オペレーティング・システムの表記規則

次の表に、Windows オペレーティング・システムの表記規則を説明し、その使用例を示します。

規則	意味	例
「スタート」>	プログラムを起動する方法を示します。	Database Configuration Assistant を起動するには、「スタート」>「プログラム」>「Oracle - HOME_NAME」>「Configuration and Migration Tools」>「Database Configuration Assistant」を選択します。
ファイル名およびディレクトリ名	ファイル名およびディレクトリ名は大 / 小文字が区別されません。特殊文字のうち左山カッコ (<)、右山カッコ (>)、コロン (:)、二重引用符 (")、スラッシュ (/)、縦線 () およびダッシュ (-) は使用できません。円記号 (¥) は、引用符で囲まれている場合でも、要素のセパレータとして処理されます。Windows では、ファイル名が ¥¥ で始まる場合、汎用ネーミング規則が使用されていることになります。	c:¥winnt"¥"system32 は、C:¥WINNT¥SYSTEM32 と同じです。
C:¥>	現在のハードディスク・ドライブの Windows コマンド・プロンプトを表します。コマンド・プロンプトのエスケープ文字はカレット (^) です。プロンプトは作業中のサブディレクトリを示します。このマニュアルでは、コマンド・プロンプトと呼びます。	C:¥oracle¥oradata>
特殊文字	Windows コマンド・プロンプトのうち二重引用符 (") には、エスケープ文字として、特殊文字の円記号 (¥) が必要な場合があります。カッコおよび一重引用符 (') にはエスケープ文字はありません。エスケープ文字および特殊文字の詳細は、Windows オペレーティング・システムのドキュメントを参照してください。	C:¥>exp scott/tiger TABLES=emp QUERY=¥"WHERE job='SALESMAN' and sal<1600¥" C:¥>imp SYSTEM/password FROMUSER=scott TABLES=(emp, dept)

規則	意味	例
<code>HOME_NAME</code>	Oracle ホームの名前を表します。ホーム名には、英数字で 16 文字まで使用できます。ホーム名に使用可能な特殊文字は、アンダースコアのみです。	<code>C:\> net start OracleHOME_NAME\TNSListener</code>
<code>ORACLE_HOME</code> および <code>ORACLE_BASE</code>	<p>Oracle8 リリース 8.0 以下のリリースでは、Oracle コンポーネントをインストールすると、すべてのサブディレクトリが最上位の <code>ORACLE_HOME</code> ディレクトリに配置されます。この <code>ORACLE_HOME</code> ディレクトリには、デフォルトで次のいずれかの名前が使用されます。</p> <ul style="list-style-type: none">■ Windows NT の場合 : <code>C:\orant</code>■ Windows 98 の場合 : <code>C:\orawin98</code> <p>今回のリリースは、Optimal Flexible Architecture (OFA) のガイドラインに準拠しています。最上位の <code>ORACLE_HOME</code> ディレクトリに配置されないサブディレクトリもあります。<code>ORACLE_BASE</code> と呼ばれる最上位ディレクトリがあります。このディレクトリは、デフォルトで <code>C:\oracle</code> となります。他の Oracle ソフトウェアがインストールされていないコンピュータに Oracle9i の最新リリースをインストールする場合、最初の Oracle ホーム・ディレクトリのデフォルト設定は、<code>C:\oracle\orann</code> (nn は最新のリリース番号) となります。Oracle ホーム・ディレクトリは、<code>ORACLE_BASE</code> の直下に配置されます。</p> <p>このマニュアルに示すすべてのディレクトリ・パスの例は、OFA の表記規則に従っています。</p>	<code>ORACLE_BASE\ORACLE_HOME\rdbs\admin</code> ディレクトリへ移動します。

OLAP API の概要

この章では、Java アプリケーションに OLAP API の使用を計画しているアプリケーション開発者を対象に、OLAP API の概要を示します。

この章では、次の項目について説明します。

- [OLAP API の概要](#)
- [OLAP API を介したデータおよびメタデータへのアクセス](#)
- [OLAP API クライアント・ソフトウェア](#)
- [OLAP API アプリケーションの開発](#)
- [OLAP API アプリケーションが実行するタスク](#)

OLAP API の概要

OLAP API は、オンライン分析処理（OLAP）のためにアプリケーションからデータへのアクセスを可能にする Java Application Programming Interface（API）です。OLAP API は、Oracle コンポーネントである Oracle OLAP に付属しています。

OLAP API の用途は、OLAP アプリケーションの開発を容易にすることです。これによって、ユーザーが Graphical User Interface を介してデータに対する選択、集計、計算およびその他の分析タスクを動的に実行できるようになります。通常、OLAP アプリケーションのユーザー・インタフェースでは、データがグラフやクロス集計などの多次元形式で表示されます。

一般に、OLAP アプリケーションはビジネス・インテリジェンス・システムおよびデータ・ウェアハウス・システムのコンテキスト内で開発され、OLAP API の機能は、このタイプのアプリケーション用に最適化されています。OLAP API を使用すると、Java アプリケーションでデータへのアクセス、操作および多次元形式での表示が可能になります。また、OLAP API を使用すると、操作順に従って問合せを定義することもできます。これによって、問合せ全体を再作成しなくても個々の問合せ手順を取り消すことが可能になります。このような複数手順の問合せは、動的な変更および改良を簡単に行うことができます。

多次元概念および OLAP API

データ・ウェアハウスおよび OLAP アプリケーションは、データの多次元ビューに基づき、データの選択を表す問合せを処理します。多次元ビューを反映する概念の定義を次に示します。これらの概念は、データ・ウェアハウス、OLAP および OLAP API の基本となります。

- **ディメンション**: データを分類する構造。顧客、製品および時間のディメンションが一般的に使用されます。通常、ディメンションには 1 つ以上の階層が関連付けられます。複数の異なるディメンションをメジャーと組み合わせて使用すると、エンド・ユーザーがビジネス上の問題を解決できます。たとえば、月ごとにデータを分類する時間ディメンションは、「製品の売上は 1 月と 6 月のどちらが多いか?」のような質問に回答するために役立ちます。
- **メジャー**: 調査および分析可能なデータ（通常は数値であり加算的）。通常、任意のメジャーは 1 つ以上のディメンションで分類され、それらのディメンションによって「ディメンション化された」と表されます。
- **階層**: ディメンション要素を親子関係で編成する方法として、順序付けられたレベルを使用する論理構造。通常、エンド・ユーザーは、階層レベルをドリルダウンまたはドリルアップすることによって、階層を拡張または縮小できます。
- **レベル**: 階層内の位置。たとえば、時間ディメンションには、日、月、四半期および年という階層レベルが含まれる場合があります。
- **属性**: エンド・ユーザーがデータを選択するために指定可能な、ディメンション要素の記述的特性。たとえば、エンド・ユーザーは色の属性を使用して製品を選択する場合があります。

- 問合せ: 特定のデータ・セット（問合せの結果セット）の仕様。仕様には、データの選択、集計、計算または操作が必要な場合があります。このような操作は、問合せ固有のものであります。

これらの概念の他に、キューブおよびエッジという 2 つのデータ・ウェアハウスおよび OLAP の概念があります。これらは OLAP API 固有ではありませんが、多くの場合、OLAP API を使用するアプリケーションの設計に組み込まれます。

- キューブ: 多次元データの論理編成。通常、キューブのエッジにはディメンション値が含まれ、キューブの本体にはメジャー値が含まれます。たとえば、売上データがキューブに編成され、そのエッジに時間、製品および顧客ディメンションの値、本体に売上メジャーの値が含まれる場合があります。
- エッジ: キューブの 1 つの面。各エッジには、1 つ以上のディメンションの値が含まれます。キューブのエッジ数に制限はありませんが、多くの場合、表示の目的でデータが 3 つのエッジ（行エッジ、列エッジおよびページ・エッジ）に沿って編成されます。

これらすべての概念の詳細は、『Oracle9i データ・ウェアハウス・ガイド』を参照してください。

OLAP API を介してアプリケーションでアクセス可能なデータのタイプ

Oracle OLAP の一部である OLAP API によって、Oracle データ・ウェアハウスに格納されたデータに Java アプリケーション（アプレットを含む）でアクセスできるようになります。データ・ウェアハウスは、トランザクション処理ではなく、問合せおよび分析用に設計されたリレーショナル・データベースです。ウェアハウス・データは、多くの場合、多次元データ・モデルを表すスター・スキーマに準拠します。スター・スキーマは、外部キーを介して関連付けられた 1 つ以上のファクト表およびディメンション表で構成されます。通常、データ・ウェアハウスは、Oracle9i Warehouse Builder などの Extraction Transformation Transport (ETT) ツールによってトランザクション処理データベースから作成されます。

OLAP API で任意のデータ・ウェアハウス内のデータにアクセスするには、データベース管理者が、まず Oracle OLAP でサポートされている編成に従ってデータ・ウェアハウスが構成されていることを確認する必要があります。サポートされている編成にはスター・スキーマなどがあります。データがウェアハウス内で編成されると、データベース管理者は OLAP メタデータ API を使用して、必要なメタデータを作成する必要があります。このメタデータは、「データについてのデータ」と定義できます。最後に、メタデータを所定の場所に配置すると、アプリケーションが OLAP API を介してデータおよびメタデータの両方にアクセスできます。

サポートされているデータ・ウェアハウス構成および OLAP メタデータ API の使用方法の詳細は、『Oracle9i OLAP ユーザーズ・ガイド』を参照してください。

データベース管理者が OLAP メタデータ API を使用してメタデータを作成したウェアハウス・データの集合は、OLAP API でアクセス可能なデータ・ストアと考えられます。ただし、OLAP API を介してデータにアクセスする各ユーザーは、セキュリティ上の制限によって、データ・ストア内でアクセス可能なデータの範囲が制限される場合があります。

OLAP API を介してアプリケーションで実行可能なタスク

OLAP API を介してアプリケーションで実行可能なタスクは次のとおりです。

- データ・ストアへの接続の確立。
- メタデータの調査による、表示または分析に使用可能なデータの検出。
- アプリケーション・ユーザーのニーズに応じたデータの操作（データの選択、集計、計算など）を行う問合せの作成。
- 多次元形式での表示用に構成された問合せ結果の取得。
- アプリケーション・ユーザーによる分析の改良時における、（既存の問合せの全面的な再定義ではなく）既存の問合せの変更。

OLAP API 開発のコンテキスト

OLAP API は Java API であるため、Java 環境のすべてのメリットが得られます。OLAP API は、プラットフォーム非依存で、抽象化、カプセル化、ポリモフィズム、継承などのオブジェクト指向 API のメリットを提供します。これらの機能は OLAP API に組み込まれ、クライアント・アプリケーションは Java で作成されているため、そのアプリケーション・コードでもそれらの機能のメリットが得られます。

アプリケーション開発者は、OLAP API を使用するために、Java、オブジェクト指向プログラミング、リレーショナル・データベース、データ・ウェアハウスおよび多次元の OLAP の概念を理解しておく必要があります。

OLAP API を介したデータおよびメタデータへのアクセス

OLAP API メタデータは、任意の接続を介して OLAP API に使用可能なデータを記述します。メタデータには、次の 3 つの項目が記録されます。

- 任意のデータ・セットが存在するという情報。たとえば、データ・ストアに売上メジャーが存在する、という情報です。
- データ・セットの構造。たとえば、売上メジャーが顧客、製品および時間でディメンション化されている、という構造です。
- データ・セットの特性。たとえば、売上メジャーには数値が含まれ、レポートに使用可能な説明的な名前が付いている、という特性です。

一方、1999 年 1 月中旬にアトランタで 3542 ドル分の男子向け上着が販売された、という情報はデータであり、メタデータではありません。

これらの例に示すとおり、売上メジャーのメタデータとデータとは区別されます。OLAP API では、ディメンションのメタデータとデータも同様に区別されます。たとえば、製品ディメンションが存在し、要素としてテキスト値が含まれているという情報はメタデータです。一方、要素のいずれかが「男子向け上着」であるという情報はデータです。

OLAP API の MDM モデル

OLAP API の多次元メタデータ (MDM) ・モデルでは、OLAP およびデータ・ウェアハウスの開発者にとって一般的な多次元形式でデータが記述されます。たとえば、これにはメジャー、ディメンション、階層および属性のオブジェクトが含まれます。

OLAP API によって MDM モデルの実装で提供される Java クラスの一部を次に示します。

- MdmMeasure
- MdmDimension
- MdmHierarchy
- MdmLevel
- MdmAttribute
- MdmSchema
- MdmMetadataProvider

MdmSchema は、MdmMeasure、MdmDimension およびその他の MdmSchema オブジェクトのコンテナです。MdmSchema は、OLAP 管理機能である Oracle Enterprise Manager 内のメジャー・フォルダに対応します。MdmSchema は、リレーショナル・スキーマに対応しない場合がありますことに注意してください。

MdmMetadataProvider は、データベース管理者が Oracle Enterprise Manager の OLAP 管理機能を使用して作成したメタデータ・オブジェクトへのアクセスをアプリケーションに提供します。アプリケーションは、MdmMetadataProvider の `getRootSchema` メソッドを使用してメタデータにアクセスします。このメソッドは、最上位の MdmSchema を戻します。これには、この特定の MdmMetadataProvider を介してアクセス可能なすべての MdmMeasure および MdmDimension オブジェクトが含まれます。MdmDimension および MdmMeasure オブジェクトは階層ツリーで編成され、サブスキーマが最上位のスキーマの下にネストする場合があります。アプリケーションは、すべてのネストした MdmSchema オブジェクトの `getMeasures`、`getDimensions` および `getSubSchemas` メソッドを使用して、メタデータをナビゲートし、使用可能なデータを検出します。また、アプリケーションはメソッドを使用して、関連する MdmHierarchy、MdmLevel および MdmAttribute オブジェクトを取得できます。

OLAP API メタデータの詳細は、[第 2 章「OLAP API メタデータの理解」](#)を参照してください。

OLAP API を介したデータへのアクセス

MdmMeasure または MdmDimension は、データ・ストア内のデータを表します。たとえば、`salesAmount` という MdmMeasure が一連の数値要素 (値は売上高 (ドル)) を表し、`productDim` という MdmDimension が一連のテキスト要素 (値は製品名) を表すと想定します。ただし、アプリケーションは MdmMeasure または MdmDimension を使用してデータに対する問合せを作成できません。MdmMeasure および MdmDimension オブジェクトは、

メタデータとしてデータの定義を提供しますが、そのデータに対する問合せ機能は提供しません。そのため、アプリケーションで、分析用にデータを選択、計算および操作するために問合せを作成する必要があります。

アプリケーションは、MdmMeasure または MdmDimension のデータに対する問合せを作成するために、MdmMeasure または MdmDimension の getSource メソッドをコールします。このメソッドは、問合せ用のデータを表す Source オブジェクトを作成します。Source は、結果セットを定義する問合せの仕様です。この例では、結果セットは MdmMeasure または MdmDimension のデータです。

Source オブジェクトは、メタデータ・オブジェクト用のデータを表す他に、アプリケーションが作成するすべての問合せ用のデータを表すこともできます。たとえば、Source は、MdmDimension 値 (January, February, March) を選択する問合せ、またはある MdmMeasure の値から他の値を引く計算 (salesAmount - unitCost) を行う問合せを指定する場合があります。アプリケーションは、Source およびそのサブクラスの強力なメソッドを使用して、ユーザーの要求どおりにデータを組み合わせることができます。また、新しい各問合せは新しい Source を指定します。

アプリケーションは、任意の Source 用のデータの表示を準備する際、Source の Cursor を作成します。次に、アプリケーションはこの Cursor を使用して OLAP サービスからデータを要求および取得します。アプリケーションがデータを要求する際、一度に要求するデータの量（たとえば、画面上の 40 個のセルを含む表を埋めるために十分な量）を指定できます。次に、OLAP サービスは、効率的な取得に関連する問題を処理します。アプリケーションは、OLAP API を介して取得するデータ・ブロックのタイミング、サイズおよびキャッシュの設定を管理する必要はありません。

多くの OLAP アプリケーションの主な用途はデータ・ストアに対する問合せを作成することであるため、大部分のデータ操作コードは次のクラスを処理します。これらの各クラスには、データを選択、計算および操作を行うためのメソッドが含まれています。

- Source
- BooleanSource
- NumberSource
- StringSource

Source オブジェクトの 1 つのメリットは、ディメンションとメジャーが区別されないことです。すべての Source オブジェクトは同様に動作します。

ユーザーの接続要求

アプリケーション開発者は、データおよびメタデータが適切に準備されていることの他に、アプリケーション・ユーザーが OLAP API を介してデータ・ストアに接続でき、ユーザーにデータへのアクセス権が付与されていることを確認する必要があります。このような接続を設定する方法の詳細は、『Oracle9i OLAP ユーザーズ・ガイド』を参照してください。

OLAP API クライアント・ソフトウェア

OLAP API クライアント・ソフトウェアは、Oracle OLAP へのプログラミング・インタフェースを実装するクラスを含む一連の Java パッケージです。アプリケーションは、これらのクラスのメソッドをコールして、データの検出、問合せ、処理および取得を行います。

Java アプリケーションは、OLAP API Java クラスのメソッドをコールする際、OLAP API クライアント・ソフトウェアを使用して、Oracle データベース・インスタンス内に存在する Oracle OLAP と通信します。OLAP API クライアント・ソフトウェアと Oracle OLAP 間の通信は、リレーショナル・データベースに接続するための標準の Java インタフェースである Java Database Connectivity (JDBC) を介して行われます。JDBC の詳細は、『Oracle9i 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。

ソフトウェア構成

OLAP API クライアント・ソフトウェアを使用する（OLAP API クラスのメソッドをコールする）アプリケーションは、1 つのコンピュータ上に常駐させるか、または 2 つの異なるコンピュータに分割できます。たとえば、エンド・ユーザーの部分を OLAP API のコールを行う部分から分割できます。この場合、3 つのコンピュータ上のソフトウェアが関連する可能性があります。

可能な構成の詳細は、『Oracle9i OLAP ユーザーズ・ガイド』を参照してください。

OLAP API クライアント・ソフトウェアの使用要件

アプリケーションの開発時に OLAP API クラスを使用するには、これらのクラスを標準の方法で Java コードにインポートします。アプリケーションをユーザーにデプロイする際、アプリケーションに OLAP API クラスを含めます。また、ユーザーが JDBC にアクセス可能であることを確認する必要もあります。

OLAP API アプリケーションを開発するには、Sun 社の Java Development Kit (JDK) が必要です。ユーザーには、開発に使用された JDK と互換性のあるバージョン番号の Java Runtime Environment (JRE) が必要です。

Java のバージョン要件および OLAP API クライアント・ソフトウェアの設定方法の詳細は、[付録 A「開発環境のセットアップ」](#)を参照してください。OLAP API クラスおよびメソッドの詳細は、OLAP API Javadoc およびこのマニュアルの後続の章を参照してください。

OLAP API アプリケーションの開発

OLAP API アプリケーションを開発するには、次の手順を実行します。

1. 一般的な設計の問題を決定します。
2. エンド・ユーザーの問合せに対する要件を決定します。

3. エンド・ユーザーの間合せを作成する OLAP API の Template オブジェクトを設計します。これはオプションの手順です。
4. アプリケーションの Java コードを作成し、テストします。
5. ユーザーにアプリケーションをデプロイします。

次に、各手順の概要を示します。

手順 1: 一般的な設計の問題の決定

次のような一般的な問題について考えます。

- アプリケーションをスタンドアロン・アプリケーション（2 層アーキテクチャ）にするか、またはエンド・ユーザー・コードをデータ操作コードと別の層に分割（3 層アーキテクチャ）するか。
- アプリケーションが常に既知の同じメタデータ（たとえば、構造が固定された従業員データ）にアクセスするか、または接続するたびに使用可能なメタデータを検出する必要があるか。

手順 2: エンド・ユーザーの間合せに対する要件の決定

できるだけ詳細に、エンド・ユーザーが作成可能な間合せの特性を指定します。OLAP API を使用すると、操作順に従って間合せを定義することができるため、アプリケーションの間合せの変更機能を決定することも重要です。次のような問題について考えます。

- エンド・ユーザーがアプリケーションのダイアログ・ボックスを介してデータを選択する基準。たとえば、アプリケーションにディメンションのリストが表示されるかどうか。ユーザーがディメンションの階層をドリルアップおよびドリルダウンできるかどうか。ユーザーがデータを選択する際に指定可能なディメンションの属性（色やサイズなど）が存在するかどうか。ユーザーがデータ値に基づいて選択を行う（20,000 人を超える人口など）ことができるかどうか。
- ユーザーが一連の手順で間合せを改良する場合に、プロセスの 1 つの手順を取り消して、間合せを以前の状態に戻すことができるかどうか。
- ユーザーが間合せを改良する場合に、取消し要求の有効範囲を指定できるかどうか。たとえば、取消し要求が、選択ダイアログ・ボックスの多くのフィールドのうち、1 つのフィールドの値にのみ適用される場合があります。

エンド・ユーザーの間合せの計画はアプリケーション設計プロセスの重要な手順であるため、できるだけ慎重に行う必要があります。アプリケーション・ユーザー・インタフェースで処理されるすべての概念的な間合せオブジェクトを識別する、エンド・ユーザーの間合せモデルを作成することが理想です。これによって、オブジェクト指向設計のメリットを利用でき、ユーザー・インタフェース・オブジェクトと OLAP API オブジェクト間の明確な対応付けが可能になります。

アプリケーション・ユーザー・インタフェースの概念的な問合せオブジェクトの例を次に示します。

- **ディメンション**: このオブジェクトは、ユーザーがドリル可能な階層および選択可能な属性を持ちます。
- **ディメンション・セクション**: このオブジェクトは、ディメンション要素のセクションを表します。
- **エッジ**: このオブジェクトは、キューブの1つの面を表し、関連するディメンション・オブジェクトを持ちます。
- **キューブ**: この多次元オブジェクトは、関連するエッジ・オブジェクトを持ちます。また、関連するメジャーも持ちます。

これらの概念的な各問合せオブジェクトは、OLAP API の `Template` オブジェクトによって表すことができます。

手順 3: エンド・ユーザーの問合せを作成する OLAP API の `Template` オブジェクトの設計

`Template` オブジェクトを設計します。この手順は、OLAP API アプリケーションの実装におけるオプションの手順です。`Template` オブジェクトの使用には次のメリットがあるため、この手順を実行することをお勧めします。

- **動的問合せ**。`Template` を使用すると、変更可能な問合せを作成できます。1 つの問合せを作成して、この問合せに類似した別の問合せを実行する必要がある場合、新しい問合せを1 から作成する必要はありません。単に、既存の問合せにわずかな変更を加えるのみです。そのため、問合せは静的ではなく動的になります。
- **問合せの改良およびロールバック**。`Template` を使用すると、問合せの指定時に完了している一連の手順をキャプチャできます。各手順で問合せの追加の改良が行われ、新しい問合せ状態として記録されます。ユーザーが1 つ以上の指定の手順の取消しを決定した場合は、問合せを以前の状態にロールバックすることができます。
- **ユーザー・インタフェースの特性に対するコードの適合**。`Template` を設計する際、ユーザーが実行する操作に直接対応させることができます。たとえば、アプリケーションにバランス・シートが含まれている場合、すべての適切な特性（集計メソッドなど）および動作（自動合計など）を組み込んだバランス・シートの `Template` を作成できます。

`Template` オブジェクトがアプリケーション・ユーザー・インタフェースの問合せ作成をミラー化する方法の詳細な例を次に示します。ユーザーが次の手順を実行してデータの3次元キューブを作成可能なアプリケーションについて考えてみます。

1. キューブに含めるデータを持つメジャーを選択します。
2. キューブの構造を指定する各ディメンションの値を選択します。
3. キューブの3つのエッジに対するディメンションの配置を指定します。

このインタフェースのアプリケーション開発者は、各オブジェクト（ディメンション、ディメンション・セクション、エッジおよびキューブ）の `Template` サブクラスを設計します。設計の 1 つの手順として、必要に応じてオブジェクトを組合せ可能にする `Template` サブクラスのメソッドを指定します。たとえば、エッジの `Template` クラスに `addDimension` メソッド、キューブの `Template` クラスに `addEdge` メソッドを含めることができます。ディメンション、ディメンション・セクション、エッジおよびキューブの `Template` クラスを一度実装すると、アプリケーションでそれらのクラスを繰り返し使用できます。これらのクラスは、データの間合せおよび操作を行うアプリケーション・コードの基本的なビルディング・ブロックです。

アプリケーション設計プロセスのこの段階では、アプリケーションの各 `Template` の詳細な仕様を作成する必要があります。`Template` オブジェクトを設計する方法の詳細は、[第 10 章「動的間合せの作成」](#)を参照してください。

手順 4: アプリケーションの Java コードの作成およびテスト

この時点では Java コードが作成されていません。ここまでは、アプリケーションの設計に関する問題について考え、アプリケーションに含める `Template` オブジェクトの詳細な仕様を作成しました。ここでは、次の手順を実行してアプリケーションを実装します。

1. 開発コンピュータに OLAP API クライアント・ソフトウェアを設定します（[付録 A「開発環境のセットアップ」](#)を参照）。3 層アプリケーションを設計している場合、開発コンピュータは（OLAP API の観点から見て）中間層コンピュータになります。
2. アプリケーションの開発およびテストに使用するデータ・ストアを指定します。データが Oracle データ・ウェアハウス内でスター・スキーマまたはスノーフレーク・スキーマとして構成されており、Oracle Enterprise Manager の OLAP 管理機能でメタデータが提供されていることを確認します。
3. 必要に応じて OLAP API クラスをインポートして、アプリケーションの Java クラスを作成します。作成した Java クラスに、設計した `Template` クラスを含めます。
4. テスト用データ・ストアを使用してアプリケーションをテストします。

OLAP API を使用するアプリケーションをコーディングする方法の詳細は、このマニュアルの後続の章および OLAP API Javadoc を参照してください。アプリケーションで一般的に実行されるタスクの詳細は、1-11 ページの「[OLAP API アプリケーションが実行するタスク](#)」を参照してください。

手順 5: ユーザーへのアプリケーションのデプロイ

アプリケーションをデプロイする際は、次の操作を実行する必要があります。

- OLAP API の Java クラスを、開発した Java クラスとともに含めます。
- ユーザーのコンピュータ（または中間層コンピュータ）に、OLAP オプションを含む Oracle データベース・インスタンスへのアクセス権があることを確認します。

- ユーザーに、OLAP メタデータ API で準備されたメタデータを含む適切な Oracle データ・ウェアハウスへのアクセス権があることを確認します。
- インストール手順および作成したユーザー・インタフェースの説明が記載されたアプリケーションのドキュメントを提供します。

OLAP API アプリケーションが実行するタスク

OLAP API を使用するアプリケーションは、通常、次のタスクを実行します。

1. データ・ストアに接続します。
2. 使用可能なメタデータを検出します。
3. 問合せを介してデータを選択および計算します。
4. 問合せ結果を取得します。

次に、これらのタスクの概要を示します。詳細は、後続の章を参照してください。

タスク 1: データ・ストアへの接続

アプリケーションは、ターゲット Oracle データベースに関するいくつかの情報を識別し、JDBC 接続メソッドにこれらの情報を指定して、データ・ストアに接続します。

接続方法の詳細は、[第 3 章「データ・ストアへの接続」](#)を参照してください。

タスク 2: 使用可能なメタデータの検出

接続の確立後、アプリケーションは `MdmMetadataProvider` を作成します。このオブジェクトは、データ・ストア内のすべてのメタデータ・オブジェクトへのアクセスを提供します。

アプリケーションは、使用可能なメタデータを検出するために、`MdmMetdataProvider` の `getRootSchema` メソッドを使用して、すべてのメタデータ・オブジェクトの最上位のメジャー・フォルダを取得します。次に、ルートの下に存在するディメンション、メジャーおよびサブフォルダを取得します。アプリケーションは、すべてのディメンションおよびメジャーを取得すると、それらに問い合せて、属性、階層、レベルおよびその他の特性を取得できます。

アプリケーションは、処理する必要があるメタデータ・オブジェクトを判別すると、データの選択および操作に使用するためのオブジェクトの関連リストをユーザーに提供できます。

メタデータ・オブジェクトの詳細は、[第 2 章「OLAP API メタデータの理解」](#)を参照してください。アプリケーションが使用可能なメタデータを検出する方法の詳細は、[第 4 章「使用可能なメタデータの検出」](#)を参照してください。

タスク 3: 問合せを介したデータの選択および計算

すべての OLAP アプリケーションの最も重要な部分は、データ・ストアに対する問合せを作成することです。ユーザーは、アプリケーション・ユーザー・インタフェースを使用して、データを選択し、そのデータに実行する処理を指定できます。次に、データ操作コードが、ユーザーによるこれらの指定をデータ・ストアに対する問合せに変換します。問合せには、ディメンション要素の選択のような単純なものから、メジャー値の何通りかの集計および計算のような複雑なものまであります。

問合せを指定する OLAP API オブジェクトは `Source` です。したがって、OLAP API アプリケーションの大部分は、`Source` オブジェクトの処理のみに使用されます。

`select`、`remove`、`appendValues` などのメソッドを使用して `Source` オブジェクトを直接操作することによって、選択を実行できます。また、`plus`、`div`、`total` などのメソッドを使用して値を計算できます。`Source` およびそのサブクラス `NumberSource`、`StringSource`、`BooleanSource` には、データを操作するための様々なメソッドが含まれています。`Source` の最も強力なメソッドは `join` です。このメソッドを使用すると、考えられるほぼすべての方法で `Source` オブジェクトを組み合わせることができます。

単純なユーザー・インタフェースを実装する場合は、`Source` クラスのメソッドのみを使用して、インタフェース内でユーザーが指定したデータを選択および操作できます。ただし、ユーザーが複数手順での選択、および問合せの変更や選択の個々の手順の取消しを行えるようにする場合、`Template` クラスを使用する必要があります (1-7 ページの「[OLAP API アプリケーションの開発](#)」を参照)。各 `Template` のコード内では `Source` クラスのメソッドを使用しますが、`Template` クラス自体でほとんどの複雑な問合せの変更および改良が可能になります。また、汎用の `Template` クラスを作成し、アプリケーションの様々な部分に再利用して、作業を最小限に抑えることができます。

`Source` オブジェクトの使用の詳細は、[第 5 章「問合せの概要」](#)を参照してください。`Template` オブジェクトの使用の詳細は、[第 10 章「動的問合せの作成」](#)を参照してください。

タスク 4: 問合せ結果の取得

OLAP アプリケーションのユーザーは、データの選択、計算、組合せおよび一般的な操作を行う際に、操作結果の表示も要求します。そのため、アプリケーションは、データ・ストアから問合せの結果セットを取得し、データを多次元形式で表示する必要があります。アプリケーションは、OLAP API を介して問合せの結果セットを取得するために、問合せを指定する `Source` に基づいて `Cursor` を作成します。

OLAP API はデータの多次元ビューを処理するように設計されているため、`Source` は多次元の結果セットを持つことができます。たとえば、`Source` は、3 つの `MdmDimension` オブジェクトで構成された `MdmMeasure` を表すことができます。この `Source` の `Cursor` の構造は、`Source` 自体をミラー化したものになります。したがって、`Cursor` の編成は、同じ 3 つの `MdmDimension` オブジェクトに基づいています。

アプリケーションは、多次元の `Cursor` 構造をループして、`Cursor` を介してデータのすべての項目を取得することができます。この設計は、コンピュータ画面に表示するための標準

のユーザー・インタフェース・オブジェクトの要件に対して適切に調整されています。特に、多次元形式でのデータの表示に対して適切に調整されています。

`Cursor` オブジェクトを使用してデータを取得する方法の詳細は、[第 8 章「Cursor クラスおよび Cursor の概念」](#)を参照してください。

OLAP API メタデータの理解

この章では、OLAP API が提供するメタデータ・オブジェクトについて説明し、これらのオブジェクトと、データベース管理者が OLAP メタデータ API を使用して指定する OLAP メタデータ・オブジェクトの関係について説明します。

この章では、次の項目について説明します。

- [OLAP API メタデータの概要](#)
- [OLAP メタデータ・オブジェクト](#)
- [OLAP API の MDM メタデータ・オブジェクトの概要](#)
- [MdmDimension クラス](#)
- [MdmLevel クラス](#)
- [MdmHierarchy クラス](#)
- [MdmListDimension クラス](#)
- [MdmMeasure クラス](#)
- [MdmAttribute クラス](#)
- [MDM メタデータ・オブジェクトのデータ型およびタイプ](#)

OLAP API メタデータの概要

OLAP API は、Oracle データベースに格納されたデータの多次元ビューへのアクセスを可能にする Java アプリケーションを提供します。OLAP API の設計には、そのビューと一貫性があり、データ・ウェアハウスおよび OLAP 開発者にとって一般的なオブジェクト（メジャー、ディメンション、階層、レベル、属性など）が含まれています。OLAP API の設計には、MDM（多次元メタデータ）というオブジェクト指向モデルが組み込まれています。

データベース管理者は、MDM モデルをサポートするために、Oracle データベース内のデータを準備する必要があります。最新の SQL 拡張ではディメンションなどのいくつかの多次元オブジェクトが導入されていますが、その他にも追加する必要があるオブジェクトおよび特性が存在します。

データの準備

データベース管理者は、まず特定の仕様に従って編成されたデータ・ウェアハウスを準備します。たとえば、データ・ウェアハウスはスター・スキーマに準拠する場合があります。この要件の詳細は、『Oracle9i OLAP ユーザーズ・ガイド』を参照してください。

メタデータの準備

管理者は、OLAP メタデータ API を使用して、OLAP メタデータをデータ・ウェアハウスに追加します。この手順で作成された OLAP メタデータ・オブジェクトは、Oracle OLAP がデータにアクセスするために必要なメタデータを提供します。これらの OLAP メタデータ・オブジェクトは、OLAP API の MDM メタデータ・オブジェクトにマップされます。

次の項では、データベース管理者が Oracle OLAP で使用するために準備する OLAP メタデータ・オブジェクトの概要を示します。

OLAP メタデータ・オブジェクト

データベース管理者は、OLAP メタデータ API を使用して、OLAP メタデータをデータ・ウェアハウスに追加します。最終的に、1 つ以上のメジャーを含む 1 つ以上のメジャー・フォルダを作成します。メジャーにはディメンションが含まれ、ディメンションには階層、レベルおよび属性が含まれます。これらの各 OLAP メタデータ・オブジェクトは、OLAP API の MDM オブジェクトに直接マップされます。

OLAP メタデータおよび OLAP メタデータ API の使用方法の詳細は、『Oracle9i OLAP ユーザーズ・ガイド』を参照してください。

OLAP メタデータには、どの MDM オブジェクトにも直接マップされないキューブ・オブジェクトが含まれることに注意してください。データベース管理者は、各メジャーのディメンションを指定する際に、OLAP メタデータ API のキューブを参照します。ディメンションを指定すると、それらはメタデータ内の自身のメジャーと緊密に関連付けられるため、このタイプのキューブ・オブジェクトは MDM モデルでは必要ありません。

この項の後半では、OLAP API の MDM オブジェクトに直接マップされる OLAP メタデータ・オブジェクトの概要を示します。

OLAP メタデータ内のディメンション

データベース管理者がディメンションに対して指定可能な特性の一部を次に示します。

- 一般的な特性。たとえば、ディメンションの名前やそのデータが導出されるスキーマです。
- レベル。ディメンションのレベルを記録します。通常、データベース管理者は OLAP ディメンションごとに 1 つ以上のレベルを指定します。
- 階層。レベル間の親子関係を指定します。通常、データベース管理者は、OLAP ディメンションごとに 1 つ以上の階層を指定します。ディメンションのレベルが 1 つのみである場合、階層は指定されず、そのディメンションは単純な非階層リストになります。
- 属性。ディメンションのレベル要素の特性を記録します。たとえば、属性は顧客ディメンション内の各顧客の性別を記録する場合があります。

通常、データベース管理者はデータベース表の 1 つ以上の列を指定して、OLAP の各レベル、階層および属性の基礎として使用します。

データベース管理者は、ディメンションの作成後にキューブを作成します。キューブは、メジャーの編成構造を提供する一連のディメンションです。

OLAP メタデータ内のメジャー

データベース管理者は、OLAP メタデータ API を使用して、任意のメジャーが任意のキューブに属することを指定できます。キューブはメジャーの編成構造を提供する一連のディメンションであるため、任意のメジャーが任意のキューブに属することを指定すると、そのメジャーのディメンションを指定することになります。これは、メジャーの次元性が最も重要な特性の 1 つである OLAP API では、必須の情報です。

通常、データベース管理者は、メジャーのデータが存在するファクト表の列を指定してそのメジャーのデータを識別します。また、そのデータを生成する計算または変換を指定することもできます。

OLAP メタデータ内のメジャー・フォルダ

データベース管理者は、(最初にディメンションおよびキューブを作成して) メジャーを作成すると、次にメジャー・フォルダという、メジャーの 1 つ以上のグループを作成します。通常、1 つのフォルダに含まれるメジャーの内容は関連しており、すべてのメジャーが同じビジネス・エリアに関連します。たとえば、財務、販売および人事管理用に 3 つの個別フォルダが存在する場合があります。

1 つのメジャー・フォルダ内のメジャーは異なるキューブに属する場合があります、複数のスキーマからのものである場合があります。

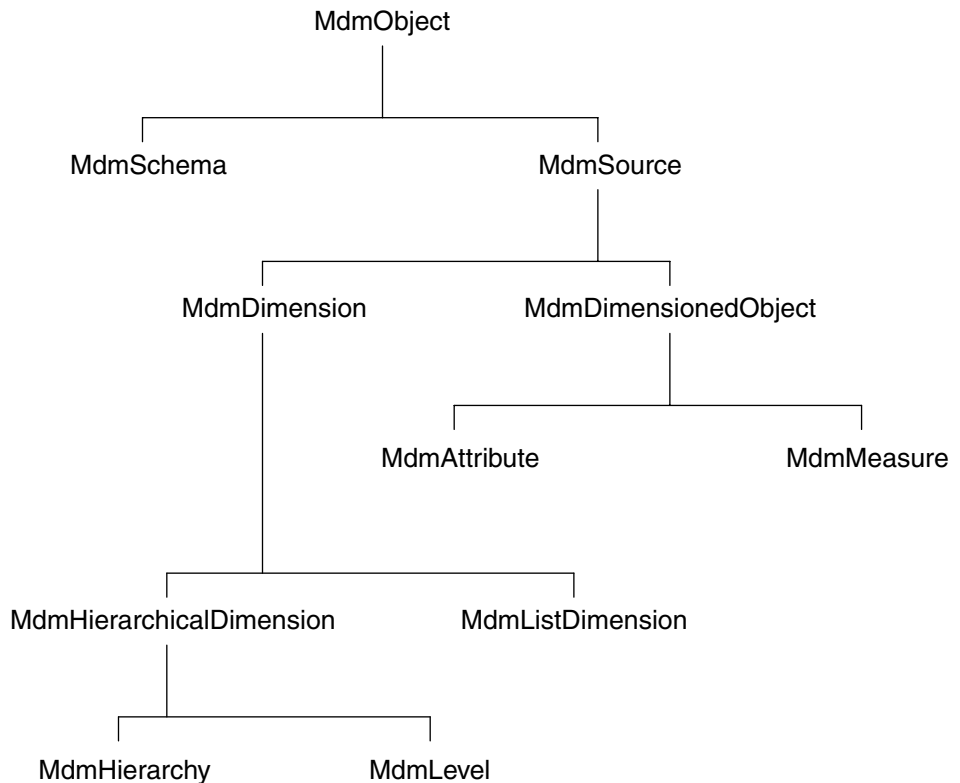
OLAP API アプリケーションがアクセスできるデータの範囲はメジャー・フォルダ単位で定義されるため、データベース管理者は1つ以上のメジャー・フォルダを作成する必要があります。OLAP API `MdmMetadataProvider` は、メジャー・フォルダに含まれるメジャーへのアクセスのみを提供します。それには、各メジャーのディメンション、その階層、レベルおよび属性も含まれます。

この場合、メジャー・フォルダがネスト可能であることを理解しておく必要があります。これは、1つのメジャー・フォルダに、固有のメジャーを持つサブフォルダのみでなく、固有のサブフォルダを持つサブフォルダも含めることができることを意味します。したがって、データベース管理者はフォルダの階層にメジャーを配置でき、OLAP API `MdmMetadataProvider` はすべてのメジャー・フォルダおよびそのサブフォルダへのアクセスを提供できます。

OLAP API の MDM メタデータ・オブジェクトの概要

OLAP API による MDM モデルの実装は、`oracle.express.mdm` パッケージのクラスで表されます。このパッケージのほとんどのクラスは、ディメンションやメジャーなどのメタデータ・オブジェクトを実装します。次の図に、`MdmObject` クラスのサブクラスを示します。

図 2-1 `MdmObject` クラスおよびそのサブクラス



MDM オブジェクトへの OLAP メタデータ・オブジェクトのマッピング

アプリケーションは、OLAP API `MdmMetadataProvider` を作成し、それを使用してデータ・ストア内の使用可能なメタデータ・オブジェクトを検出して、メタデータ・オブジェクトにアクセスします。

データベース管理者が OLAP メタデータ API を使用して指定したメタデータ・オブジェクトは、`MdmMetadataProvider` を介してアクセス可能な MDM メタデータ・オブジェクトに直接マップされます。次の表に、一般的なマッピングを示します。

OLAP メタデータ・オブジェクト	MDM メタデータ・オブジェクト
ディメンション	<code>MdmHierarchy</code> または <code>MdmListDimension</code>
階層	<code>MdmHierarchy</code>
レベル	<code>MdmLevel</code>
メジャー	<code>MdmMeasure</code>
属性	<code>MdmAttribute</code>
メジャー・フォルダ	<code>MdmSchema</code>

この章では、MDM メタデータ・オブジェクトについて説明しています。アプリケーションがデータ・ストア内の使用可能な MDM メタデータ・オブジェクトを検出する方法の詳細は、[第 4 章「使用可能なメタデータの検出」](#)を参照してください。

`MdmSchema` および `MdmSource` は、`MdmObject` の 2 つのサブクラスです。

MdmSchema クラス

`MdmSchema` は、ナビゲーションに使用されるデータ・セットを表します。`MdmSchema` は、`MdmMeasure`、`MdmDimension` およびその他の `MdmSchema` オブジェクトのコンテナです。`MdmSchema` は、関連する項目を含むフォルダまたはディレクトリに相当します。このクラスは、Oracle データベースのリレーショナル・スキーマには対応しません。かわりに、このクラスはメジャー・フォルダに対応します。メジャー・フォルダは、複数のリレーショナル・スキーマからのデータを含むことができ、データベース管理者によって OLAP メタデータ API を使用して作成されます。

OLAP API を介してアクセス可能なデータは、ルート `MdmSchema` という最上位の `MdmSchema` の下に配置されます。ルートの下には、1 つ以上のサブスキーマが存在します。アプリケーションは、この章で説明するとおり、`MdmMetadataProvider` の `getRootSchema` メソッドをコールしてメタデータのナビゲートを開始します。

ルート `MdmSchema` は、データ・ストア内のすべての `MdmDimension` オブジェクトを含みます。ほとんどの `MdmDimension` オブジェクトは、ルート `MdmSchema` の下のサブスキーマに含まれています。ただし、データ・ストアはサブスキーマに含まれていないディメンションを含むことができます。ルート `MdmSchema` は、サブスキーマに含まれていない

MdmDimension オブジェクトのみでなく、サブスキーマに含まれている MdmDimension オブジェクトも含まれます。

ルート MdmSchema は、サブスキーマに含まれていない MdmMeasure オブジェクトのみを含みます。ほとんどの MdmMeasure オブジェクトはサブスキーマに属するため、通常、ルート MdmSchema には MdmMeasure オブジェクトが含まれません。

MdmSchema には、それに含まれるすべての MdmMeasure、MdmDimension および MdmSchema オブジェクトを取得するためのメソッドが存在します。ルート MdmSchema には、メジャー MdmDimension を取得するためのメソッドも存在します。このメジャーの要素は、それがサブスキーマに属するかどうかにかかわらず、データ・ストア内のすべての MdmMeasure オブジェクトです。

MdmSource クラス

MdmSource は、分析に使用されるメジャー、ディメンションまたはその他のデータ・セット（属性など）を表します。この抽象クラスは、MdmMeasure、MdmDimension、MdmAttribute などのいくつかの重要な MDM メタデータ・クラスの基礎となります。

MdmSource オブジェクトはデータを表しますが、そのデータへの問合せを作成する機能は提供しません。このオブジェクトの機能は情報提供用で、データの存在、構造および特性を記録します。データ値へのアクセスは提供しません。

アプリケーションは、MdmSource の getSource メソッドをコールして、任意の MdmSource のデータ値にアクセスします。このメソッドは、アプリケーションが MdmSource で表されるデータへの問合せを作成可能な Source を戻します。次のコードでは、mdmProductsDim という名前の MdmDimension から Source が作成されます。

```
Source productsDim = mdmProductsDim.getSource();
```

MdmSource の getSource メソッドによって戻された Source を、プライマリ Source といいます。アプリケーションは、データを選択、計算または操作する際に、このプライマリ Source から新しい Source オブジェクトを作成します。新しい各 Source は、新しい問合せを指定します。

Source オブジェクトの使用の詳細は、第 5 章「問合せの概要」を参照してください。

この章の後半では、MdmSource のサブクラス、および密接に関連している MdmDimensionDefinition や MdmDimensionMemberType などの他のクラスについて説明します。

MdmDimension クラス

MdmDimension は MdmSource のサブクラスです。

MdmDimension の説明

MdmDimension は、データ・セットの編成が可能な要素のリストを表します。たとえば、ある年の売上データが存在し、それを月ごとに編成する場合、月のリストが売上データのディメンションになります。月ディメンションの値は、売上データ・セットの特定の値を識別するための索引として機能します。

OLAP API では、抽象クラス MdmDimension はデータの編成が可能な要素のリストの概念を表します。MdmDimension には、階層特性を持つリストを表す MdmHierarchicalDimension という名前の抽象サブクラスが含まれています。

MdmDimension の次の具象サブクラスは、分析で使用可能な特定の MdmDimension オブジェクトを指定します。

- MdmLevel。階層構造の 1 つのレベルを提供する要素のリストを表します。各要素は、1 つの親および 1 つ以上の子を持つことができます。1 つの MdmLevel 要素の親および子は、その MdmLevel 内には存在しません。それらは、異なる MdmLevel オブジェクトの要素です。
- MdmHierarchy。親子関係に基づいたレベルを含む階層構造に配置された要素のリストを表します。各要素は 1 つの親および 1 つ以上の子を持つことができ、これらのすべての要素は MdmHierarchy 内に存在します。

親要素および子要素は MdmHierarchy 内に存在しますが、これらの要素は MdmLevel オブジェクトの要素に対応します。そのため、概して MdmHierarchy は MdmLevel オブジェクトで構成されます。MdmHierarchy オブジェクトには、単純に MdmLevel オブジェクトで構成されるものがあります。また、1 つ以上の下位 MdmHierarchy オブジェクトの集合であるものもあります。この下位オブジェクトは、MdmLevel オブジェクトで構成されます。
- MdmListDimension。どの階層構造にも含まれない要素の単純なリストを表します。この要素は親および子を持ちません。

MdmLevel および MdmHierarchy は、どちらも抽象クラス MdmHierarchicalDimension の具象サブクラスです。

MdmDimension は、1 つ以上の MdmAttribute オブジェクトを持つことができます。これらの各オブジェクトは、MdmDimension の要素を要素の特性を表す値にマップします。任意の MdmDimension の MdmAttribute オブジェクトを取得するには、その `getAttributes` メソッドをコールします。

MdmDimension は、基礎となるデータの構造を表す 1 つの MdmDimensionDefinition、および要素の基本特性を表す 1 つの MdmDimensionMemberType を持ちます。これらの 2 つのオブジェクトは、それらが属する MdmDimension の重要な情報を保持します。任意の

MdmDimension についてこれらの関連オブジェクトを取得するには、その `getDefinition` メソッドおよび `getMemberType` メソッドを使用します。

MdmDimensionDefinition が保持する情報

MdmDimensionDefinition は、MdmDimension の基礎となるデータの構造を示します。MdmDimensionDefinition は抽象クラスです。そのため、インスタンスは常に次のサブクラスのいずれかになります。

- **MdmBaseDimensionDefinition**。MdmDimension が単一リストとして構成された基礎となるデータを含んでいることを示します。たとえば、多くの場合、MdmLevel はリレーショナル表の単一列に基づいています。
- **MdmUnionDimensionDefinition**。MdmDimension が 2 つ以上のリストの共用体として構成された基礎となるデータを含んでいることを示します。たとえば、MdmHierarchy はリレーショナル表の 2 つ以上の列（MdmLevel ごとに 1 列）に基づくことができます。
- **MdmAliasDimensionDefinition**。MdmDimension が別の MdmDimension のプロキシ（別名）として機能することを示します。

MdmUnionDimensionDefinition を持つ MdmDimension にはリージョンが存在します。任意の MdmDimension のリージョンは、その MdmDimension の要素のサブセットを表す別の MdmDimension です。たとえば、暦年用の MdmDimension に四半期を表すリージョンおよび月を表すリージョンが 1 つずつ存在する場合があります。MdmDimension のリージョンを取得するには、その MdmUnionDimensionDefinition の `getRegions` メソッドをコールします。

MdmDimensionMemberType が保持する情報

MdmDimensionMemberType は、MdmDimension 内の要素の基本特性を示します。このオブジェクトは各要素の説明を保持し、多くの場合、個々の要素に関するその他の情報を検索するためのメソッドを提供します。MdmDimensionMemberType は抽象クラスです。そのため、インスタンスは常に次のサブクラスのいずれかになります。

- **MdmTimeMemberType**。MdmDimension の要素が期間を表すことを示します。MdmTimeMemberType には、各要素の最終日および期間を検索するためのメソッドが含まれています。
- **MdmMeasureMemberType**。MdmDimension の要素がデータ・ストア内のすべての MdmMeasure オブジェクトであることを示します。MdmMeasureMemberType を持つ MdmDimension は 1 つのみ存在し、これをメジャー MdmDimension といいます。メジャー MdmDimension は、ルート MdmSchema の `getMeasureDimension` メソッドをコールして取得できます。
- **MdmStandardMemberType**。MdmDimension の要素が特定の特性を持たないことを示します。ほとんどの MdmDimension オブジェクトは、MdmStandardMemberType を持ちます。

MdmLevel クラス

MdmLevel は、MdmDimension の抽象サブクラスである MdmHierarchicalDimension のサブクラスです。

MdmLevel の説明

MdmLevel は、その親および子が他の MdmLevel オブジェクトの要素である MdmHierarchicalDimension です。任意の MdmLevel の要素は、MdmHierarchy の要素のサブセットに対応します。

1 つの MdmLevel は、データベース管理者が OLAP メタデータ API を使用して指定したレベルに基づいています。通常、データベース管理者は、データベース表の 1 つの列を指定して、レベルの要素を指定します。

MdmLevel の要素には親子関係がありますが、MdmLevel は単純なリストとして表されます。要素間の親子関係は、親属性および祖先属性に記録されます。これらの属性は、MdmLevel の getParentRelation および getAncestorsRelation メソッドをコールして取得できます。親属性および祖先属性を、親リレーションおよび祖先リレーションともいいます。

通常、基礎となるデータは単一リストとして構成されるため、MdmLevel は MdmBaseDimensionDefinition を持ちます。

MdmLevel の要素

MdmLevel の要素のリストには、そのレベルの要素のみが含まれます。要素の値は一意である必要があります。ただし、一意性はデータベース管理者が 2 つのリレーショナル列を使用してレベルを定義することによって実現できます。たとえば、市を表すレベルは、リレーショナル・データベースで city 列と state 列の両方に基づいて定義できます。これによって、「Springfield」という値を、Illinois 州 Springfield と Massachusetts 州 Springfield を表す市レベルの 2 つの異なる要素に対して表示できるようになります。

次の表に、mdmTimesDimCalHier という名前のレベル MdmHierarchy の四半期（3 か月単位）を記録する、mdmQuarter という名前の MdmLevel の要素を示します。この MdmHierarchy の対象期間は 4 年であるため、mdmQuarter の要素の数は 16 です。

mdmQuarter の要素
1998-Q1
1998-Q2
1998-Q3
1998-Q4
1999-Q1

mdmQuarter の要素
1999-Q2
1999-Q3
1999-Q4
2000-Q1
...
2001-Q4

MdmHierarchy クラス

MdmHierarchy は、MdmDimension の抽象サブクラスである MdmHierarchicalDimension のサブクラスです。

MdmHierarchy の説明

MdmHierarchy は、1 つ以上の階層構造のすべての要素を含む MdmHierarchicalDimension です。すべての親と子は MdmHierarchy 内に存在します。

MdmHierarchy 内には親子関係が存在しますが、その要素は単純なリストとして表されません。要素間の関係は、親属性および祖先属性に記録されます。これらの属性は、MdmHierarchy の getParentRelation メソッドおよび getAncestorsRelation メソッドをコールして取得できます。MdmHierarchy の MdmDimensionDefinition の getRegionAttribute メソッドをコールすると、各要素のリージョンを取得できます。親属性、祖先属性およびリージョン属性を、親リレーション、祖先リレーションおよびリージョン・リレーションともいいます。

通常、MdmHierarchy は次のいずれかのタイプです。

- レベル MdmHierarchy。リージョンが MdmLevel オブジェクトである階層構造を表します。たとえば、暦年用のレベル MdmHierarchy に年、四半期、月および日を表す MdmLevel オブジェクトがそのリージョンとして存在する場合があります。

レベル MdmHierarchy は MdmUnionDimensionDefinition を持ち、そのリージョンは MdmLevel オブジェクトです。getHierarchyType メソッドの戻り値は LEVEL_HIERARCHY です。レベル MdmHierarchy は、データベース管理者が OLAP メタデータ API を使用して定義した階層に基づいています。
- 共用体 MdmHierarchy。1 つ以上の下位階層構造を持つディメンションを表します。これらの構造は、1 つ以上のレベル MdmHierarchy または値 MdmHierarchy オブジェクトで表されます。2 つの構造を持つ MdmHierarchy の例としては、2 つのリージョン（暦年用および会計年度用に 1 つずつ）を持つ、時間を表す共用体 MdmHierarchy があります。各リージョンはレベル MdmHierarchy です。

共用体 MdmHierarchy は MdmUnionDimensionDefinition を持ち、そのリージョンは MdmHierarchy オブジェクトです。getHierarchyType メソッドの戻り値は UNION_HIERARCHY です。共用体 MdmHierarchy は、データベース管理者が OLAP メタデータ API を使用して 1 つ以上の階層を持つように定義したディメンションに基づいています。

- 値 MdmHierarchy。要素が親および子は持っていない、レベルを持たないためにリージョンも持たない階層構造を表します。たとえば、企業の従業員管理構造はレベルを持たない親子関係で表すことができます。

値 MdmHierarchy は MdmBaseDimensionDefinition を持ちます。getHierarchyType メソッドの戻り値は VALUE_HIERARCHY です。値 MdmHierarchy は、データベース管理者が OLAP メタデータ API を使用して値階層としてフラグを立てたディメンションに基づいています。

OLAP API の今回のリリースで MdmHierarchy オブジェクトを使用する場合は、次の点に注意してください。

- getAttributes メソッドは共用体 MdmHierarchy でコールし、その下位のレベル MdmHierarchy や値 MdmHierarchy オブジェクトまたは MdmLevel オブジェクトでコールしないでください。
- 共用体 MdmHierarchy ではなくレベル MdmHierarchy または値 MdmHierarchy に基づく Source オブジェクトへの問合せを作成します。
- getParentRelation メソッドおよび getAncestorsRelation メソッドは、共用体 MdmHierarchy ではなくレベル MdmHierarchy または値 MdmHierarchy でコールします。
- getRegionAttribute メソッドは、共用体 MdmHierarchy ではなくレベル MdmHierarchy の MdmUnionDimensionDefinition でコールします。このメソッドは、各 MdmHierarchy 要素が属する MdmLevel を記録する MdmAttribute を戻します。

レベル MdmHierarchy の要素

レベル MdmHierarchy の要素には、そのすべてのリージョンのすべての要素が含まれます。特定のレベル MdmHierarchy の要素の値は一意である必要があります。次の例では、暦年用および会計年度用の 2 つのレベル MdmHierarchy オブジェクトの要素を示します。

暦年用のレベル MdmHierarchy

次の表に、mdmYear、mdmQuarter、mdmMonth および mdmDay という名前の 4 つの MdmLevel オブジェクトの要素を含む、mdmTimesDimCalHier という名前のレベル

MdmHierarchy の要素の値を示します。要素の数は、年要素が 4 個、四半期要素が 16 個、月要素が 48 個および日要素が 1461 個で合計 1529 個です。

mdmTimesDimCalHier の要素
1998
1998-Q1
1998-01
01-JAN-98
02-JAN-98
03-JAN-98
...
01-FEB-98
02-FEB-98
03-FEB-98
...
1998-Q2
1998-04
01-APR-98
02-APR-98
03-APR-98
...
1999
1999-Q1
1999-01
01-JAN-99
02-JAN-99
03-JAN-99
...

会計年度用のレベル MdmHierarchy

次の表に、mdmFisYear、mdmFisQuarter、mdmFisMonthおよびmdmFisDayという名前の4つのMdmLevelオブジェクトの要素を含む、mdmTimesDimFisHierという名前のレベルMdmHierarchyの要素の値を示します。要素の数は、会計年度要素が4個、会計四半期要素が16個、会計月要素が48個および会計日要素が1461個で合計1529個です。

この例では、MdmLevelオブジェクトmdmFisDayは、MdmLevelオブジェクトmdmDayと同じリレーショナル・データベース列に基づいています（前述の暦年の例を参照）。そのため、これらの2つのMdmLevelオブジェクトに含まれる要素の値は同じです。ただし、これは要素自体が同じであることを意味しません。mdmDayの要素はmdmFisDayの要素とは異なり、2つの要素セットの値のみが同じです。

timesDimFisHier の要素
FIS-1998
FIS-1998-Q1
FIS-1998-01
01-JUL-98
02-JUL-98
03-JUL-98
...
01-AUG-98
02-AUG-98
03-AUG-98
...
FIS-1998-Q2
FIS-1998-04
01-OCT-98
02-OCT-98
03-OCT-98
...
FIS-1999
FIS-1999-Q1
FIS-1999-01
01-JUL-99

timesDimFisHier の要素
02-JUL-99
03-JUL-99
...

用語：ノードおよびリーフ

レベル MdmHierarchy は、親子関係が存在するツリー構造を表します。最下位の MdmLevel の要素をリーフといい、最下位レベルより上位の MdmLevel オブジェクトの要素をノードといいます。ノードは子を持ちますが、リーフは子を持ちません。

共用体 MdmHierarchy の要素

共用体 MdmHierarchy の要素には、そのすべてのリージョンのすべての要素が含まれます。すなわち、共用体 MdmHierarchy には、そのすべての下位 MdmHierarchy オブジェクトに含まれるすべての MdmLevel オブジェクトのすべての要素が含まれます。階層の観点では、この要素セットにはすべての階層のすべてのリーフ（最下位レベルに存在する要素）およびすべてのノード（最下位レベルより上位のレベルに存在する要素）が含まれます。

共用体 MdmHierarchy のリージョン内の個別要素

共用体 MdmHierarchy のリージョン内の要素はすべて異なります。1つの要素が共用体 MdmHierarchy の複数のリージョンに含まれることはありません。これは、データベース管理者がディメンションの2つの異なる階層で同じレベルを指定した場合も同様です。この場合、Oracle OLAP は2つの異なる MdmLevel オブジェクト（レベル MdmHierarchy ごとに1つずつ）を作成します。

共用体 MdmHierarchy の要素は異なりますが、要素の値は一意である必要はありません。そのため、次の例では、共用体 MdmHierarchy の2つのリージョンのリーフ要素は同じ値を持ちます。

時間を表す共用体 MdmHierarchy

2つのリージョンを持つ mdmTimesDim という名前の共用体 MdmHierarchy について考えてみます。1つ目のリージョンは、1529 個の要素を持つ mdmTimesDimCalHier という名前の MdmHierarchy です。2つ目のリージョンは、同様に 1529 個の要素を持つ mdmTimesDimFisHier という名前の MdmHierarchy です。mdmTimesDim の要素セットは、その2つの MdmHierarchy オブジェクトに含まれる要素の共用体です。要素を両方の MdmHierarchy オブジェクトに含めることはできないため、mdmTimesDim の要素は 3058 個になります。暦年は1月1日から始まり、会計年度は7月1日から始まることに注意してください。

次の表に、mdmTimesDim という名前の共用体 MdmHierarchy の要素の値を示します。値が同じである、mdmDay と mdmFisDay の要素を区別するために、mdmFisDay の値の横に「(会計)」と記載しています。mdmDay および mdmFisDay オブジェクトについては、前述のレベル MdmHierarchy の要素の例を参照してください。

mdmTimesDim の要素
1998
1998-Q1
1998-01
01-JAN-98
...
1999
1999-Q1
1999-01
01-JAN-99
...
FIS-1998
FIS-1998-Q1
FIS-1998-01
01-JUL-98 (会計)
...
FIS-1999
FIS-1999-Q1
FIS-1999-01
01-JUL-99 (会計)
...

MdmListDimension クラス

MdmListDimension は、MdmDimension のサブクラスです。

MdmListDimension の説明

MdmListDimension は、階層特性を持たない要素の単純なリストです。親または子を持つという概念は、MdmListDimension の要素には関係ありません。

MdmListDimension の要素

1 つの MdmListDimension は、データベース管理者が OLAP メタデータ API を使用して単一レベルを持ち階層を持たないように指定したディメンションに基づいています。

次の表に、mdmColor という名前の MdmListDimension の要素の値を示します。

mdmColor の要素
Black
Blue
Cyan
Green
Magenta
Red
Yellow
White

MdmMeasure クラス

MdmMeasure は、MdmSource の抽象サブクラスである MdmDimensionedObject のサブクラスです。

MdmMeasure の説明

MdmMeasure は、1 つ以上の MdmDimension オブジェクトで編成されたデータ・セットを表します。データの構造は、多次元配列の構造に類似しています。配列のディメンションと同様に、MdmMeasure を編成する MdmDimension オブジェクトは、個々のセルを識別するための索引を提供します。

たとえば、売上データ用の MdmMeasure が存在し、そのデータが製品、時間、顧客および経路で編成されていると想定します（経路は直接販売や間接販売などの販売方法を表しま

す)。製品ディメンション、時間ディメンション、顧客ディメンションおよび経路ディメンションによって編成構造が提供され、データが 4 次元配列を占有していると考えられます。これらの 4 つのディメンションの値は、配列の特定の各セルを識別するための索引で、1 つの売上値を含みます。配列の中の値を識別するために、各ディメンションに値を指定する必要があります。リレーショナルの観点では、MdmDimension オブジェクトが MdmMeasure の複合（コンポジット）主キーを構成します。

通常、MdmMeasure の値は数値ですが、必須ではありません。

MdmMeasure の要素

1 つの MdmMeasure は、データベース管理者が OLAP メタデータ API を使用して作成した OLAP メジャーに基づいています。ほとんどの場合、データベース管理者は、OLAP メジャーの基礎として機能させるファクト表の列を指定します（または、数学計算またはデータ変換を指定します）。多くの場合、データベース管理者は、集計メソッドとともにメジャーの OLAP ディメンションごとに 1 つ以上の階層も指定します。Oracle OLAP は、これらのすべての情報を使用して MdmMeasure の要素の数および各要素の値を識別します。

MdmDimension の要素によって決定される MdmMeasure の要素

MdmMeasure に含まれる要素のセットは、その MdmDimension オブジェクトの構造によって決定されます。これは、MdmMeasure の各要素が、その MdmDimension オブジェクトの要素を一意に組み合わせて識別されることを意味します。

通常、MdmMeasure の MdmDimension オブジェクトは共用体 MdmHierarchy オブジェクトです。これらのオブジェクトは 1 つ以上の階層構造を持ちます。共用体 MdmHierarchy の要素には、そのリージョンを表すすべてのレベル MdmHierarchy のすべてのリーフおよびノードが含まれることに注意してください。この構造が存在するため、MdmMeasure の要素の値は次の 2 種類になります。

- OLAP メタデータ API を使用して指定した、MdmMeasure の基礎となるファクト表の列（またはファクト表計算）の値。これらの値は、MdmHierarchy のリーフ要素を組み合わせて識別される MdmMeasure の要素に属します。
- Oracle OLAP が提供した集計値。これらの値は、MdmHierarchy の 1 つ以上のノード要素によって識別される MdmMeasure の要素に属します。

たとえば、mdmTimesDim および mdmProductsDim という名前の共用体 MdmHierarchy オブジェクトによってディメンション化された、mdmUnitCost という名前の MdmMeasure が存在すると想定します。各 MdmHierarchy には、リーフ要素（mdmTimesDim の 01-JAN-99 など）およびノード要素（mdmTimesDim の 1999-Q1 など）が含まれています。2 つの要素（各 MdmHierarchy から 1 つずつ）を一意に組み合わせて mdmUnitCost の各要素が識別され、可能なすべての組合せを使用して mdmUnitCost の要素セット全体が指定されます。

mdmUnitCost の要素には、リーフ要素（特定の製品品目や特定の月など）を組み合わせて識別されるものが存在します。また、ノード要素（特定の製品グループや特定の四半期など）を組み合わせて識別されるものも存在します。さらに、リーフ要素とノード要素を組み合わせて

識別されるものも存在します。リーフ要素のみによって識別される `mdmUnitCost` の要素の値は、データベース・ファクト表の列（またはファクト表計算）から直接提供されます。これらの値は、最下位レベルのデータを表します。ただし、1 つ以上のノード要素によって識別される要素の場合は、Oracle OLAP が値を提供します。これらの上位レベルの値は、集計またはロールアップされたデータを表します。

したがって、`MdmMeasure` によって表されるデータは、データ・ストアからのファクト表データと、Oracle OLAP が分析操作用に提供する集計データを組み合わせたものです。

2 つの MdmDimension オブジェクトを持つ MdmMeasure

次の表に、前述の `mdmUnitCost` という名前の `MdmMeasure` に含まれる要素の一部の値を示します。この `MdmMeasure` は、その `MdmDimension` オブジェクトとして `mdmProductsDim` および `mdmTimesDim` を持ちます。これらの各オブジェクトは、レベル `MdmHierarchy` オブジェクトであるリージョンを持つ共用体 `MdmHierarchy` です。たとえば、`mdmTimesDim` のレベル `MdmHierarchy` オブジェクトは `mdmTimesDimCalHier` および `mdmTimesDimFisHier` で、`mdmProductsDim` のレベル `MdmHierarchy` は `mdmProductsDimHier` です。

`MdmMeasure` には多くの要素が存在するため、表にはその一部のみを示しています。たとえば、`mdmTimesDim` の場合、省略記号は `mdmTimesDimFisHier` のリージョン全体とともに、`mdmTimesDimCalHier` のリージョンに含まれるその他の日、月、四半期および年を表していると考えてください。

`mdmProductsDimHier` には、製品カテゴリ（Boys など）、製品のサブカテゴリ（Outerwear - Boys など）および個々の製品品目（#23110 など）を表す 3 つのレベルが存在します。表にはレベルごとに 1 つの要素のみを示し、省略記号は残りのすべての要素を表しています。

表に示すほぼすべての要素は、集計された要素です。非集計要素にはアスタリスクが付けられています。これらの非集計要素は、`mdmProductsDim` と `mdmTimesDim` の両方の最下位レベル要素によって識別される要素です。

mdmProductsDim の要素	mdmTimesDim の要素	mdmUnitCost の要素
Boys	1998	12,800,444.00
Boys	1998-Q1	4,563,150.00
Boys	1998-01	1,837,254.00
Boys	01-JAN-98	185,346.00
Boys	02-JAN-98	232,590.00
Boys	03-JAN-98	155,403.00
...

mdmProductsDim の要素	mdmTimesDim の要素	mdmUnitCost の要素
Outerwear - Boys	1998	6,473,065.00
Outerwear - Boys	1998-Q1	2,000,317.00
Outerwear - Boys	1998-01	637,482.00
Outerwear - Boys	01-JAN-98	27,009.00
Outerwear - Boys	02-JAN-98	20,346.00
Outerwear - Boys	03-JAN-98	12,498.00
...
23110	1998	847,362.00
23110	1998-Q1	200,635.00
23110	1998-01	60,735.00
23110	01-JAN-98	2,226.00 *
23110	02-JAN-98	1,709.00 *
23110	03-JAN-98	2,047.00 *
...

MdmAttribute クラス

MdmAttribute は、MdmSource の抽象サブクラスである MdmDimensionedObject のサブクラスです。

MdmAttribute の説明

MdmAttribute は、MdmDimension の要素の特定の特性を表します。MdmAttribute は、MdmDimension の 1 つの要素を特定の値にマップします。その典型的な例としては、mdmCustomersDim という名前の MdmDimension の各顧客の性別を記録する MdmAttribute があります。この場合、MdmAttribute の要素は「Female」および「Male」という値を持ちます。

MdmAttribute の値には、String 値（「Female」など）、数値（45 など）またはオブジェクト（MdmLevel オブジェクトなど）を指定できます。

MdmMeasure と同様に、MdmAttribute にはその MdmDimension で編成された要素が含まれます。たとえば、性別 MdmAttribute には、mdmCustomersDim という名前の MdmDimension の要素ごとに 1 つの要素（「Female」または「Male」という値を持つ）が含まれます。

通常、MdmDimension のすべての要素が任意の MdmAttribute の値に意味を持ってマップされるわけではありません。たとえば、性別は市や州などの上位レベルでは意味を持たないため、性別 MdmAttribute は mdmCustomersDim の最下位レベルにのみ適用されます。MdmAttribute が MdmDimension の要素の一部に適用されない場合、その MdmAttribute 値は null になります。

MdmAttribute オブジェクトには、1 対 1 ではなく、1 対多のマッピングを提供するものが存在します。そのため、MdmDimension の任意の要素が MdmAttribute の要素セット全体にマップされる場合があります。たとえば、MdmHierarchy の祖先属性として機能する MdmAttribute は、MdmHierarchy の各要素をその祖先 MdmHierarchy 要素のセットにマップします。

MdmAttribute の要素

1 つの MdmAttribute は、データベース管理者が OLAP メタデータ API を使用してディメンションまたはレベル用に指定した属性に基づいています。

次の表に、mdmCustomersDim という名前の MdmDimension に基づく、mdmCustomersDimGender という名前の MdmAttribute の要素を示します。MdmAttribute の値は都市、国および地域レベルでは null であることに注意してください。意味のある値は顧客レベルにのみ存在し、このレベルでは各顧客が番号によって表されます。

mdmCustomersDim の要素	mdmCustomersDimGender の要素
...	...
Africa	null
South Africa	null
Cape Town	null
5420	Female
11650	Female
17880	Male
24120	Female
67720	Male
73960	Male
...	...

MDM メタデータ・オブジェクトのデータ型およびタイプ

すべての MdmSource オブジェクトには、次の 2 つの基本特性があります。

- データ型
- タイプ

MDM メタデータ・オブジェクトのデータ型

データ型は、コンピュータ言語およびデータベース・テクノロジーでは一般的な概念です。一般に、データは INTEGER、BOOLEAN、STRING などの型に分類されます。

OLAP API は、FundamentalMetadataObject クラスおよび FundamentalMetadataProvider クラスを介してデータ型の概念を実装します。OLAP API が認識するすべてのデータ型は、FundamentalMetadataObject によって表されます。このオブジェクトを取得するには、FundamentalMetadataProvider のメソッドをコールします。

次の表に、最も一般的な OLAP API データ型を示します。この表には、データ型ごとに、そのデータ型を表す FundamentalMetadataObject の説明およびそのオブジェクトを戻す FundamentalMetadataProvider のメソッドの名前を示しています。

OLAP API データ型	FundamentalMetadataObject の説明	FundamentalMetadataProvider のメソッド
Boolean	Java の boolean データ型に対応するデータ型を表します。	getBooleanDataType
Date	Java の Date クラスに対応するデータ型を表します。	getDateDataType
Double	Java の double データ型に対応するデータ型を表します。	getDoubleDataType
Float	Java の float データ型に対応するデータ型を表します。	getFloatDataType
Integer	Java の int データ型に対応するデータ型を表します。	getIntegerDataType
Short	Java の short データ型に対応するデータ型を表します。	getShortDataType
String	Java の String クラスに対応するデータ型を表します。	getStringDataType

これらの一般的なデータ型の他に、OLAP API には（一般的なデータ型のグループを表す）2 つの汎用データ型、および値が存在しないことを表す 2 つのデータ型が含まれています。次の表に、これらの追加データ型を示します。

OLAP API データ型	FundamentalMetadataObject の説明	FundamentalMetadataProvider のメソッド
Number	OLAP API 数値データ型（Double、Float、Integer および Short）の一部またはすべてを含む汎用データ型を表します。	getNumberDataType
Value	OLAP API データ型の一部またはすべてを含む汎用データ型を表します。	getValueDataType
Empty	MdmSource に対して定義された要素が存在しない場合など、データが欠落していることを表します。	getEmptyDataType
Void	MdmSource が null 値を持つ単一の要素を含む場合など、データが null であることを表します。	getVoidDataType

MdmMeasure などの MDM メタデータ・オブジェクトが任意のデータ型である場合、その各要素がそのデータ型に準拠することを意味します。データ型が数値データ型である場合、要素は特定のデータ型（Double、Float、Integer または Short）のみでなく、汎用データ型である Number にも準拠します。すべての MDM メタデータ・オブジェクトの要素は、Integer や String などのより特殊なデータ型のみでなく、Value データ型にも準拠します。

オブジェクトに数値データ型を表す要素と非数値データ型を表す要素が混在している場合、そのデータ型は Value のみになります。そのオブジェクトは、Value より特殊なデータ型を持ちません。

データ型が関連する MDM メタデータ・オブジェクトは、MdmMeasure、MdmHierarchy、MdmLevel などの MdmSource オブジェクトです。MdmMeasure の一般的なデータ型は数値データ型のいずれかで、MdmHierarchy または MdmLevel の一般的なデータ型は String です。

MdmSource のデータ型の取得

データ・ストアから MdmSource を取得済で、その要素のデータ型を確認する場合は、その `getDataType` メソッドをコールします。このメソッドは、`FundamentalMetadataObject` を戻します。

戻された `FundamentalMetadataObject` によって表される OLAP API データ型を確認するには、それを各 OLAP API データ型の `FundamentalMetadataObject` と比較します。すなわち、それを `FundamentalMetadataProvider` の各データ型メソッドの戻り値と比較します。

次のメソッドの例では、パラメータとして渡された `MdmSource` のデータ型を示す定数が戻されます。このコードは、`DataProvider` (`dp`) のメソッドをコールして `FundamentalMetadataProvider` を作成していることに注意してください。`DataProvider` の取得方法の詳細は、第4章「使用可能なメタデータの検出」を参照してください。また、このメソッドで参照されている定数は、このメソッドが属するクラスの他の箇所で定義されていることにも注意してください。これらの定数は、OLAP API によって提供されたものではありません。

例 2-1 `MdmSource` のデータ型の取得

```
public int getDataType(MdmSource metaSource) {

    int theDataType = 0;
    FundamentalMetadataProvider fmp =
        dp.getFundamentalMetadataProvider();

    if (fmp.getBooleanDataType() == metaSource.getDataType())
        theDataType = BOOLEAN_TYPE;
    else if (fmp.getDateDataType() == metaSource.getDataType())
        theDataType = DATE_TYPE;
    else if (fmp.getDoubleDataType() == metaSource.getDataType())
        theDataType = DOUBLE_TYPE;
    else if (fmp.getFloatDataType() == metaSource.getDataType())
        theDataType = FLOAT_TYPE;
    else if (fmp.getIntegerDataType() == metaSource.getDataType())
        theDataType = INTEGER_TYPE;
    else if (fmp.getShortDataType() == metaSource.getDataType())
        theDataType = SHORT_TYPE;
    else if (fmp.getStringDataType() == metaSource.getDataType())
        theDataType = STRING_TYPE;
    else if (fmp.getNumberDataType() == metaSource.getDataType())
        theDataType = NUMBER_TYPE;
    else if (fmp.getValueDataType() == metaSource.getDataType())
        theDataType = VALUE_TYPE;

    return theDataType;
}
```

MDM メタデータ・オブジェクトのタイプ

`MdmSource` などの MDM メタデータ・オブジェクトは、要素の集合です。MDM メタデータ・オブジェクトのタイプ（データ型とは異なる）とは、そのメタデータ・オブジェクトが自身の要素を導出した別のメタデータ・オブジェクトのことを示します。1つのメタデータ・オブジェクトの要素は、そのタイプの要素のサブセットに対応します。メタデータ・オブジェクトには、そのタイプの要素と一致しない要素は存在できません。

次に示す、OLAP API データ型が `String` である `mdmCustomersDim` という共用体 `MdmHierarchy` の例について考えてみます。`mdmCustomersDim` には 1 つのリージョン (`mdmCustomersDimGeogHier` という名前のレベル `MdmHierarchy`) が存在し、そのリージョンにはそれ自体のリージョン (`MdmLevel` オブジェクト) が存在します。いずれの場合も、リージョンは要素のサブセットを表します。次に、リージョンをそれが属する `MdmHierarchy` の下にインデントして示します。

```
mdmCustomersDim
    mdmCustomersDimGeogHier
        mdmGeogTotal
        mdmRegion
        mdmSubregion
        mdmCountry
        mdmState
        mdmCity
        mdmCustomer
```

階層構造が存在するため、たとえば、`mdmCountry` はその要素を `mdmCustomersDimGeogHier` の要素から導出します。この場合、`mdmCountry` の要素のセットは `mdmCustomersDimGeogHier` の要素のサブセットに対応し、`mdmCountry` のタイプは `mdmCustomersDimGeogHier` になります。

同様に、`mdmCustomersDimGeogHier` は `mdmCustomersDim` のリージョンです。そのため、`mdmCustomersDimGeogHier` はそのタイプである `mdmCustomersDim` からそれ自体の要素を導出しています。

ただし、`mdmCustomersDim` はどのオブジェクトのリージョンでもなく、階層の最上位です。`mdmCustomersDim` がその要素を導出する要素のプールは、可能な `String` 値のセット全体です。そのため、`mdmCustomersDim` のタイプは、OLAP API `String` データ型を表す `FundamentalMetadataObject` になります。`mdmCustomersDim` の場合、タイプとデータ型は同じです。

最も一般的な `MdmSource` オブジェクトの一般的なタイプを次に示します。

- `MdmLevel` のタイプは、それが属するレベル `MdmHierarchy` です。
- レベル `MdmHierarchy` のタイプは、それが属する共用体 `MdmHierarchy` です。
- 共用体 `MdmHierarchy` のタイプは、その OLAP API データ型を表す `FundamentalMetadataObject` です。通常、これは `String` データ型です。
- `MdmMeasure` のタイプは、その OLAP API データ型を表す `FundamentalMetadataObject` です。通常、これは OLAP API 数値データ型のいずれかです。

MdmSource のタイプの取得

データ・ストアから MdmSource を取得済で、そのタイプを確認する場合は、その `getType` メソッドをコールします。このメソッドは、MdmSource オブジェクトのタイプであるオブジェクトを戻します。

たとえば、次の Java 文は、`mdmCountry` という名前の MdmLevel のタイプを取得します。

例 2-2 MdmSource のタイプの取得

```
MetadataObject mdmCountryType = ((MdmSource) mdmCountry).getType();
```

データ・ストアへの接続

この章では、OLAP API を介してデータ・ストアに接続する方法について説明します。

この章では、次の項目について説明します。

- [接続処理の概要](#)
- [接続の確立](#)
- [既存の接続の取得](#)
- [接続を介した DML コマンドの実行](#)
- [接続のクローズ](#)

接続処理の概要

アプリケーションが OLAP API を介してデータにアクセスする場合、Sun 社の Java Database Connectivity (JDBC) の Oracle 実装によって提供される接続が使用されます。この JDBC 実装の使用の詳細は、『Oracle9i 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。

接続手順

接続手順では、Oracle JDBC ドライバのロード、そのドライバを介した接続の取得、およびトランザクションとデータ転送を処理する 2 つの OLAP API オブジェクトの作成を行います。

これらの手順の詳細は、3-2 ページの「[接続の確立](#)」を参照してください。

接続の前提条件

Oracle データベースへの OLAP API 接続を行う前に、次の要件を満たしていることを確認します。

- Oracle データベース・インスタンスが OLAP オプションを指定してインストールされ、実行されている。
- Oracle データベースのユーザー ID が、データ・ストアの基礎となるリレーショナル・スキーマへのアクセス権を持つ。
- Oracle クライアントの JDBC ドライバのインストールが完了している。JDBC ドライバをインストールする方法の詳細は、『Oracle9i 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。
- OLAP API jar ファイルがアプリケーション開発コンピュータ上にデプロイされており、アプリケーション・コードが使用可能である。OLAP API jar ファイルを設定する方法の詳細は、[付録 A「開発環境のセットアップ」](#)を参照してください。

接続の確立

接続を確立するには、次の手順を実行します。

1. 使用する JDBC ドライバをロードします。
2. DriverManager から Connection を取得します。
3. TransactionProvider を作成します。
4. DataProvider を作成します。

この項では、これらの手順の詳細を示します。

これらの手順で作成する `TransactionProvider` および `DataProvider` オブジェクトは、データ・ストアの処理で使われます。たとえば、特定の `Source` オブジェクトを作成する際に、この `DataProvider` オブジェクトのメソッドを使います。

手順 1: JDBC ドライバのロード

次のコードでは、JDBC ドライバがロードされ、JDBC `DriverManager` に登録されます。

例 3-1 接続用の JDBC ドライバのロード

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

ドライバのロード後、`DriverManager` オブジェクトを使用して接続できます。Oracle JDBC ドライバをロードする方法の詳細は、『Oracle9i 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。

手順 2: DriverManager からの接続の取得

次のコードでは、`DriverManager` から JDBC `Connection` オブジェクトが取得されます。

例 3-2 JDBC 接続の取得

```
String url = "jdbc:oracle:thin:@lab1:1521:orcl";
String user = "hepburn";
String password = "tracey";
oracle.jdbc.OracleConnection conn = (oracle.jdbc.OracleConnection)
    java.sql.DriverManager.getConnection(url, user, password);
```

この例では、`tracey` というパスワードを持つユーザー `hepburn` が、`orcl` というシステム識別子 (SID) を持つデータベースに接続されます。接続は、ホスト `lab1` の TCP/IP リスナー・ポート 1521 を介して行われます。この接続では、Oracle JDBC Thin ドライバが使用されます。

様々な方法で `getConnection` メソッドを使用して接続の特性を指定できます。詳細は、『Oracle9i 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。

`Connection` オブジェクトの取得後、必要な OLAP API オブジェクトである `TransactionProvider` および `DataProvider` を作成できます。

手順 3: TransactionProvider の作成

`TransactionProvider` は、OLAP API インタフェースです。したがって、コードでは、具象クラスの `ExpressTransactionProvider` という名前のインスタンスを使います。次のコードでは、`TransactionProvider` が作成されます。

例 3-3 TransactionProvider の作成

```
ExpressTransactionProvider tp = new ExpressTransactionProvider();
```

TransactionProvider は、DataProvider の作成に必要です。

手順 4: DataProvider の作成

DataProvider は、OLAP API の抽象クラスです。したがって、コードでは、具象サブクラスの ExpressDataProvider という名前のインスタンスを使用します。次のコードでは、DataProvider が作成および初期化されます。

例 3-4 DataProvider の作成

```
ExpressDataProvider dp = new ExpressDataProvider(conn, tp);  
dp.initialize();
```

DataProvider は、MetadataProvider の作成（[第 4 章「使用可能なメタデータの検出」](#)を参照）に必要です。

既存の接続の取得

接続の確立後に JDBC Connection オブジェクトにアクセスする必要がある場合、DataProvider の getConnection メソッドをコールします。次のコードでは、dp という名前の DataProvider の getConnection メソッドがコールされます。

例 3-5 既存の接続の取得

```
oracle.jdbc.OracleConnection currentConn = dp.getConnection();
```

接続を介した DML コマンドの実行

一部のアプリケーションは、Oracle OLAP のデータ操作言語（DML）コマンドまたはプログラムのランタイム実行に依存します。DML コマンドおよびプログラムは、OLAP API に固有の、MDM メタデータのコンテキスト外に存在する分析作業領域で実行されます。したがって、これらのコマンドおよびプログラムは、MdmMeasure や MdmDimension などの MDM オブジェクトでは動作しません。かわりに、これらのコマンドおよびプログラムは、変数やディメンションなどの DML オブジェクトで動作します。MDM コンテキストと DML コンテキストは関連していますが、個別のものです。

分析作業領域で DML コマンドまたはプログラムを実行するには、使用する JDBC Connection オブジェクトを指定して、OLAP API SLPExecutor オブジェクトを作成します。データ操作言語は、ストアド・プロシージャ言語（SPL）ともいいます。

次のコードでは、conn という名前の JDBC Connection オブジェクトで、SPLExecutor オブジェクトが作成および初期化されます。

例 3-6 DML コマンドの実行

```
SPLExecutor dmlExec = new SPLExecutor(conn);  
dmlExec.initialize();
```

DML コマンドを実行する分析作業領域を指定するには、DML コマンド **AW** を使用して、該当する作業領域を指定します。たとえば、次のコマンドは、mysales という名前の作業領域を指定する **AW** コマンドを実行します。

```
string returnVal = dmlExec.execute('aw attach mysales');
```

DML の使用方法の詳細は、『Oracle9i OLAP 開発者ガイド - Oracle OLAP DML』および Oracle9i OLAP DML Reference ヘルプを参照してください。SPLExecutor の使用方法の詳細は、OLAP API Javadoc を参照してください。

接続のクローズ

データ・ストアの処理の完了後、JDBC Connection オブジェクトの **close** メソッドを使用します。次のサンプル・コードでは、Connection オブジェクトは conn と示されています。

例 3-7 接続のクローズ

```
conn.close();
```

OLAP API の使用を終了した後でデータベースへの JDBC 接続での作業を続行する場合は、DataProvider の **close** メソッドを使用して OLAP API リソースを解放します。次のサンプル・コードでは、DataProvider は dp と示されています。

```
dp.close();
```

使用可能なメタデータの検出

この章では、OLAP API を介してデータ・ストア内のメタデータを検出する手順について説明します。

この章では、次の項目について説明します。

- [メタデータの検出手順の概要](#)
- [MdmMetadataProvider の作成](#)
- [ルート MdmSchema の取得](#)
- [ルート MdmSchema のコンテンツの取得](#)
- [メタデータ・オブジェクトの特性の取得](#)
- [メタデータ・オブジェクトの Source の取得](#)
- [メタデータ検出のサンプル・コード](#)

メタデータの検出手順の概要

OLAP API は、データベース管理者が OLAP メタデータ API を使用して OLAP メタデータを作成した Oracle データの集合へのアクセスを提供します。このデータの集合は、アプリケーションのデータ・ストアです。

データ・ストアには、データベース管理者が OLAP メタデータ API を使用して作成したすべてのメジャー・フォルダが含まれます。ただし、任意のアプリケーションの実行中に参照可能なデータ・ストアの範囲は、接続を確立した際に使用したユーザー ID に付与されるデータベース権限によって異なります。ユーザーは、データベース管理者が作成したすべてのメジャー・フォルダ (MdmSchema オブジェクトなど) を参照できますが、それらのメジャー・フォルダ内のメジャーおよびディメンションを参照できるのは、そのメジャーおよびディメンションに基づくリレーショナル表へのアクセス権を所有している場合にかぎられます。

MDM メタデータ

データベース管理者がメタデータを作成すると、OLAP メタデータ API は、メジャー、ディメンションおよびその他の OLAP メタデータ・オブジェクトを作成します。OLAP API では、これらのオブジェクトは多次元メタデータ (MDM)・オブジェクトとしてアクセスされます (第2章「[OLAP API メタデータの理解](#)」を参照)。OLAP メタデータ・オブジェクトと MDM オブジェクトの間のマッピングは、Oracle OLAP によって自動的に実行されます。

メタデータ検出の目的

アプリケーションは、データ・ストア内のメタデータ・オブジェクトを使用して、データを理解します。これらのオブジェクトを使用して、使用可能なデータ、データの構造および特性を調べることができます。

したがって、接続後最初に実行する手順は使用可能なメタデータを検出することです。この情報を使用して、選択または計算するデータおよび表示方法に関する選択肢をエンド・ユーザーに提示することができます。

メタデータの検出手順

メタデータを調べる前に、アプリケーションが Oracle OLAP への接続を確立する必要があります (第3章「[データ・ストアへの接続](#)」を参照)。その後、アプリケーションは次の手順を実行します。

1. MdmMetadataProvider を作成します。
2. MdmMetadataProvider からルート MdmSchema を取得します。
3. MdmMeasure、MdmDimension、MdmMeasureDimension および MdmSchema オブジェクトを含むルート MdmSchema のコンテンツを取得します。さらに、サブスキーマのコンテンツを取得します。

4. MdmMeasure および MdmDimension のそれぞれの特性を取得します。たとえば、各 MdmMeasure には、その MdmDimension オブジェクトを取得します。また、各 MdmDimension には、共用体 MdmHierarchy、レベル MdmHierarchy、MdmLevel または MdmListDimension を検出します。

次に、これらの手順の詳細を示します。

メタデータの検出および問合せの作成

メタデータの検出後は、通常、データの選択、計算および操作を行う問合せを作成します。これらの方法でデータを処理するには、Oracle OLAP が問合せ用データを表すために作成した Source オブジェクトを取得する必要があります。これらの Source オブジェクトを、プライマリ Source オブジェクトといいます。

この章では、使用可能なメタデータの検出の最初の手順について詳しく説明します。また、メタデータ・オブジェクトからのプライマリ Source の取得手順についても簡単に説明します。プライマリ Source オブジェクトを処理する方法およびそれらに基づく問合せを作成する方法の詳細は、後続の章を参照してください。

MdmMetadataProvider の作成

MdmMetadataProvider は、データ・ストア内のメタデータへのアクセスを提供します。これは、メジャー、ディメンション、メジャー・フォルダなどの OLAP メタデータ・オブジェクトを、MdmMeasure、MdmDimension、MdmSchema などの対応する MDM オブジェクトにマップします。

MdmMetadataProvider を作成する前に、DataProvider を作成する必要があります（[第3章「データ・ストアへの接続」](#)を参照）。

次のコードでは、dp という名前の DataProvider を使用して MdmMetadataProvider が作成されます。

例 4-1 MdmMetadataProvider の作成

```
MdmMetadataProvider mp = null;  
mp = (MdmMetadataProvider) dp.getDefaultMetadataProvider();
```

ルート MdmSchema の取得

データ・ストア内のメタデータの調査では、最初にルート MdmSchema を取得します。

ルート MdmSchema の機能

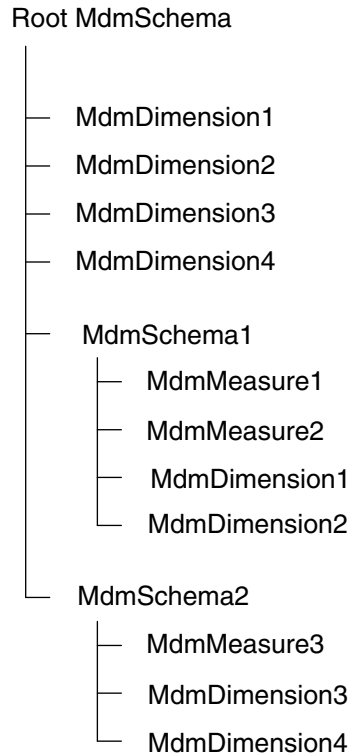
任意の MdmMetadataProvider を介してアクセス可能なメタデータ・オブジェクトは、最上位にルート MdmSchema を持つツリーのような構造で構成されます。ルート MdmSchema の下に、サブスキーマと呼ばれる MdmDimension オブジェクトおよび 1 つ以上の MdmSchema オブジェクトが存在します。また、サブスキーマに属さない MdmMeasure オブジェクトは、ルートの下に含まれます。

サブスキーマは、独自の MdmMeasure および MdmDimension オブジェクトを持ちます。また、独自のサブスキーマを持つこともできます。

ルート MdmSchema は、サブスキーマ内のすべての MdmDimension オブジェクトを含みます。したがって、通常、1 つの MdmDimension はツリーで 2 回（ルート MdmSchema の下で 1 回、サブスキーマの下で 1 回）表示されます。MdmDimension がサブスキーマに属さない場合、ルートの下にのみ表示されます。

使用可能なメタデータ・オブジェクトは、ルート MdmSchema でツリーの最上位から順に検索されます。次の図に、2 つのサブスキーマおよび 4 つの MdmDimension オブジェクトを持つ MdmSchema を示します。

図 4-1 ルート MdmSchema およびサブスキーマ



データベース管理者は、OLAP メタデータ API を使用して、1 つ以上の最上位レベルのメジャー・フォルダの下にディメンションおよびメジャーを配置します。Oracle OLAP は、メジャー・フォルダを MdmSchema オブジェクトにマップする際、常に、ルート MdmSchema を MdmSchema オブジェクトの上に最上位レベルのメジャー・フォルダ用に作成します。したがって、データベース管理者が 1 つのメジャー・フォルダのみを作成した場合でも、対応する MdmSchema は、ルートの下でサブスキーマとなります。

MDM メタデータ・オブジェクトおよび OLAP メタデータ・オブジェクトへのマップの詳細は、[第 2 章「OLAP API メタデータの理解」](#)を参照してください。

getRootSchema メソッドのコール

次のコードでは、mp という名前の MdmMetadataProvider のルート MdmSchema が取得されます。

例 4-2 ルート MdmSchema の取得

```
MdmSchema root = mp.getRootSchema();
```

ルート MdmSchema のコンテンツの取得

ルート MdmSchema は、MdmDimension および MdmSchema オブジェクトを含み、MdmMeasure オブジェクトを含む場合もあります。さらに、ルート MdmSchema は、すべての MdmMeasure オブジェクトを示すメジャー MdmDimension を持ちます。

MdmSchema の MdmDimension オブジェクトの取得

次のコードでは、schema という名前の MdmSchema に含まれる、MdmDimension オブジェクトの List が取得されます。

例 4-3 MdmDimension オブジェクトの取得

```
List dims = schema.getDimensions();
```

MdmSchema のサブスキーマの取得

次のコードでは、schema という名前の MdmSchema に含まれる、MdmSchema オブジェクトの List が取得されます。

例 4-4 サブスキーマの取得

```
List subSchemas = schema.getSubSchemas();
```

サブスキーマのコンテンツの取得

ルート MdmSchema の下の各 MdmSchema では、getMeasures、getDimensions および getSubSchemas メソッドをコールできます。手順は、ルート MdmSchema のコンテンツの取得手順と同じです。

メジャー MdmDimension およびそのコンテンツの取得

次のコードでは、ルート MdmSchema のメジャー MdmDimension が取得されます。このメソッドは、ルート MdmSchema のみで使用してください。メジャー MdmDimension を持つのはルート MdmSchema のみであるため、このメソッドをサブスキーマで使用しても意味がありません。

例 4-5 MdmMeasureDimension およびそのコンテンツの取得

```
MdmMeasureDimension mdmMeasureDim = root.getMeasureDimension();
```

次のコードでは、メジャー MdmDimension の要素である MdmMeasure オブジェクトの名前が出力されます。

```
MdmMeasureMemberType mdmMemberType =
    (MdmMeasureMemberType) mdmMeasureDim.getMemberType();
List mdList = mdmMemberType.getMeasures();
Iterator mdIter = mdList.iterator();
while (mdIter.hasNext())
    System.out.println("*****Contains Measure: " +
        ((MdmMeasure) mdIter.next()).getName());
```

メタデータ・オブジェクトの特性の取得

MdmMeasure および MdmDimension オブジェクトのリストを検出した後、それらのオブジェクトの特性を調べます。

MdmMeasure の MdmDimension オブジェクトの取得

MdmMeasure の主な特性は、MdmDimension オブジェクトを関連付けていることです。次のコードでは、sales という名前の MdmMeasure 用に MdmDimension オブジェクトの List が取得されます。

```
List dimsOfSales = mdmSalesAmount.getDimensions();
```

この章の後半に示すサンプル・コードの getMeasureInfo メソッドは、任意の MdmMeasure に属する MdmDimension オブジェクトを反復する 1 つの方法を示しています。

MdmDimension の関連オブジェクトの取得

MdmDimension は、getDefinition および getMemberType メソッドをコールして取得可能な MdmDimensionDefinition および MdmDimensionMemberType オブジェクトを関連付けています。MdmHierarchy である場合、それには自身の MdmUnionDimensionDefinition の getRegions メソッドをコールして取得可能なリージョンも存在します。

共用体 MdmHierarchy のレベル MdmHierarchy オブジェクトを取得する方法の例を次に示します。次のコードでは、レベル MdmHierarchy オブジェクトの名前が出力されます。

```
MdmUnionDimensionDefinition unionDef =
    (MdmUnionDimensionDefinition) mdmDimObj.getDefinition();
List hierarchies = unionDef.getRegions();
```

```
for (Iterator iterator = hierarchies.iterator();
    iterator.hasNext();)
{
    MdmHierarchy hier = (MdmHierarchy) iterator.next();
    System.out.println("Hierarchy: " + hier.getName());
}
```

この章の後半に示すサンプル・コードの `getDimInfo` メソッドは、任意の `MdmDimension` の次のメタデータ・オブジェクトを取得する 1 つの方法を示しています。

- `MdmDimensionMemberType`。
- `MdmAttribute` オブジェクト。
- 具象クラスおよび階層タイプ。
- 親、祖先およびリージョン属性。
- `MdmDimensionDefinition`。
- リージョン。共用体 `MdmHierarchy` の場合、コードではコンポーネント `MdmHierarchy` オブジェクトが取得されます。レベル `MdmHierarchy` の場合、コードではコンポーネント `MdmLevel` オブジェクトが取得されます。
- デフォルトのレベル `MdmHierarchy` (共用体 `MdmHierarchy` の場合)。

メソッドは、その他の `MdmDimension` 特性の取得にも使用できます。MDM クラスのすべてのメソッドの詳細は、OLAP API Javadoc を参照してください。

メタデータ・オブジェクトの Source の取得

メタデータ・オブジェクトはデータ・セットを表しますが、そのデータへの問合せを作成する機能は提供しません。この機能は情報提供用で、データの存在、構造および特性を記録します。データ値へのアクセスは提供しません。

アプリケーションは、任意のメタデータ・オブジェクトのデータ値にアクセスするために、そのデータを表す `Source` オブジェクトを取得します。メタデータ・オブジェクトのデータを表す `Source` を、プライマリ `Source` といいいます。

アプリケーションは、メタデータ・オブジェクトのプライマリ `Source` を取得するために、そのメタデータ・オブジェクトの `getSource` メソッドをコールします。たとえば、アプリケーションが 1999 年の売上高を表示する必要がある場合、まず `mdmSalesAmount` という名前の `MdmMeasure` の `getSource` メソッドを使用します。

例 4-6 メタデータ・オブジェクトのプライマリ Source の取得

```
Source salesAmount = mdmSalesAmount.getSource();
```


アプリケーションは、MdmSource の具象サブクラスのインスタンスであるすべてのオブジェクトの getSource メソッドをコールできます。具象サブクラスのリストを次に示します。

- MdmHierarchy
- MdmLevel
- MdmListDimension
- MdmAttribute
- MdmMeasure

プライマリ Source オブジェクトを取得および処理する方法の詳細は、[第 5 章「問合せの概要」](#)を参照してください。

メタデータ検出のサンプル・コード

次に示すサンプル・コードは、SampleMetadataDiscoverer という名前の簡単な Java プログラムです。このプログラムでは、すべてのデータ・ストアのルート MdmSchema の下に存在するメタデータ・オブジェクトが検出されます。このプログラムでは、ルート MdmSchema およびそのサブスキーマの MdmMeasure および MdmDimension オブジェクトの名前および関連オブジェクトのリストが出力されます。

プログラム・コードの後に、売上履歴のリレーショナル・スキーマで構成されるデータ・ストアに対して実行された場合のプログラムの出力を示します。このデータ・ストアは Oracle のインストール時に提供されます。OLAP メタデータでは、売上履歴スキーマはメジャー・フォルダ SH_CAT として表されています。OLAP API 接続を介して、メジャー・フォルダ SH_CAT が SH_CAT という名前の MdmSchema にマップされます。

SampleMetadataDiscoverer プログラムには、SH_CAT および MdmSchema に固有のコードのフラグメントが含まれます。このコードでは、getName メソッドへの戻り値が PRODUCTS_DIM である、MdmDimension のプライマリ Source が取得されます。

ほとんどの場合、アプリケーションは、メタデータ・オブジェクトの検索にその内部名 (PRODUCTS_DIM など) を使用しません。また、出力の生成に System.out.println メソッドを使用しません。ただし、このサンプル・コードでは、簡略化のためこれらの方法を使用します。

SampleMetadataDiscoverer プログラムのコード

このプログラムは、MyConnection という名前の仮想クラスの connectOnLab1 という名前の仮想メソッドをコールして接続を確立します。また、MyConnection.closeConnection という名前のメソッドをコールして接続をクローズします。これらのメソッドのコードはここでは示していません。接続の手順は、[第 3 章「データ・ストアへの接続」](#)を参照してください。

例 4-7 使用可能なメタデータの検出

```
package mytestpackage;

import oracle.express.mdm.*;
import oracle.olapi.metadata.MetadataObject;

import oracle.olapi.data.source.Source;
import oracle.express.olapi.data.full.ExpressDataProvider;

public class SampleMetadataDiscoverer {

    static final int TERSE = 0;
    static final int VERBOSE = 1;

    public SampleMetadataDiscoverer(){
    }

    public static void main(String[] args) {

        // Connect through JDBC to a database on Lab1
        //      and get a DataProvider (see Chapter 3)
        ExpressDataProvider dp = MyConnection.connectOnLab1();

        // Create an MdmMetadataProvider
        MdmMetadataProvider mp = null;
        mp = (MdmMetadataProvider) dp.getDefaultMetadataProvider();

        // Get metadata info about the root MdmSchema and its subschemas
        MdmSchema root = null;
        try {
            root = mp.getRootSchema();
            System.out.println("***Root MdmSchema: " + root.getName());
            MdmDimension measureDim = root.getMeasureDimension();
            System.out.println("*****Measure MdmDimension: " +
                measureDim.getName());
            getSchemaInfo(root, TERSE);
        } catch (Exception e) {
            System.out.println("***Exception encountered : " + e.toString());
        }

        // Make a Source object out of the PRODUCTS_DIM MdmDimension
        System.out.println("***Making a Source object for PRODUCTS_DIM");

        MdmDimension mdmProductDim = null;
        try {
            List rootDims = root.getDimensions();
            Iterator rootDimIter = rootDims.iterator();
```

```

        while (mdmProductDim == null && rootDimIter.hasNext()) {
            MdmDimension aDim = (MdmDimension) rootDimIter.next();
            if (aDim.getName().equals("PRODUCTS_DIM"))
                mdmProductDim = aDim;
        }
        Source product = mdmProductDim.getSource();
        System.out.println("*****Made the Source");
    } catch (Exception e) {
        System.out.println("*****Exception encountered : " + e.toString());
    }

    // Close the connection
    MyConnection.closeConnection(conn);
}

// *****

// Method for getting info about an MdmSchema
public static void getSchemaInfo(MdmSchema schema, int outputStyle) {

    System.out.println("***Schema: " + schema.getName());
    // Get the MdmSchema's dimension info
    MdmDimension oneDim = null;
    try {
        List dims = schema.getDimensions();
        Iterator dimIter = dims.iterator();
        System.out.println(" ");
        System.out.println("*****");
        System.out.println(" ");
        while (dimIter.hasNext()) {
            oneDim = (MdmDimension) dimIter.next();
            getDimInfo(oneDim, outputStyle);
            System.out.println(" ");
            System.out.println("*****");
            System.out.println(" ");
        }
    } catch (Exception e) {
        System.out.println("*****Exception encountered : " + e.toString());
    }

    // Get the MdmSchema's measure info
    MdmMeasure oneMeasure = null;
    try {
        List measures = schema.getMeasures();
        Iterator measIter = measures.iterator();
        while (measIter.hasNext()) {
            oneMeasure = (MdmMeasure) measIter.next();

```

```
        getMeasureInfo(oneMeasure, outputStyle);
        System.out.println("  ");
        System.out.println("  ");
    }
} catch (Exception e) {
    System.out.println("*****Exception encountered : " + e.toString());
}

// Get the MdmSchema's subschema info
MdmSchema oneSchema = null;
try {
    List subSchemas = schema.getSubSchemas();
    Iterator subSchemaIter = subSchemas.iterator();
    while (subSchemaIter.hasNext()) {
        oneSchema = (MdmSchema) subSchemaIter.next();
        getSchemaInfo(oneSchema, VERBOSE);
    }
} catch (Exception e) {
    System.out.println("***Exception encountered : " + e.toString());
}
}

// *****

// Method for getting info about an MdmDimension
public static void getDimInfo(MdmDimension dim, int outputStyle) {

    System.out.println("*****MdmDimension Name: " + dim.getName());
    System.out.println("*****Description: " + dim.getDescription());

    if (outputStyle == VERBOSE) {

        // Get MdmDimensionMemberType for the MdmDimension
        try {
            MdmDimensionMemberType dimMemberType = dim.getMemberType();
            if (dimMemberType instanceof MdmStandardMemberType)
                System.out.println("*****Member Type: MdmStandardMemberType");
            if (dimMemberType instanceof MdmTimeMemberType)
                System.out.println("*****Member Type: MdmTimeMemberType");
            if (dimMemberType instanceof MdmMeasureMemberType)
                System.out.println("*****Member Type: MdmMeasureMemberType");
        } catch (Exception e) {
            System.out.println("***Exception encountered : " + e.toString());
        }

        // Get attributes of the MdmDimension
        try {
```

```

        List attributes = dim.getAttributes();
        Iterator attrIter = attributes.iterator();
        while (attrIter.hasNext())
            System.out.println("*****Attribute: " +
                ((MdmAttribute) attrIter.next()).getName());
    } catch (Exception e) {
        System.out.println("***Exception encountered : " + e.toString());
    }

    // Get concrete class and hierarchy type of the MdmDimension
    String kindOfDim = null;
    try {
        if (dim instanceof MdmListDimension) {
            kindOfDim = "ListDim";
            System.out.println("*****" + dim.getName() +
                " is an MdmListDimension");
        }
        else if (dim instanceof MdmHierarchy)
            switch(((MdmHierarchy) dim).getHierarchyType()) {
                case (MdmHierarchy.UNION_HIERARCHY):
                    kindOfDim = "UnionHier";
                    System.out.println("*****" + dim.getName() +
                        " is a union MdmHierarchy");
                    break;
                case (MdmHierarchy.LEVEL_HIERARCHY):
                    kindOfDim = "LevelHier";
                    System.out.println("*****" + dim.getName() +
                        " is a level MdmHierarchy");
                    break;
                case (MdmHierarchy.VALUE_HIERARCHY):
                    kindOfDim = "ValueHier";
                    System.out.println("*****" + dim.getName() +
                        " is a value MdmHierarchy");
                    break;
            }
        else {
            kindOfDim = "Level";
            System.out.println("*****" + dim.getName() + " is an MdmLevel");
        }
    } catch (Exception e) {
        System.out.println("***Exception encountered : " + e.toString());
    }

    // For level MdmHierarchy, get parent, ancestors, and region attributes
    if (kindOfDim.equals("LevelHier"))
    {
        System.out.println("*****Parent attribute: " +

```

```
(MdmHierarchicalDimension) dim).getParentRelation().getName();
System.out.println("*****Ancestors attribute: " +
    (MdmHierarchicalDimension) dim).getAncestorsRelation().getName());
System.out.println("*****Region attribute: " +
    (MdmUnionDimensionDefinition) dim.getDefinition()
    .getRegionAttribute().getName());
}

// Get the MdmDimensionDefinition for the MdmDimension
MdmDimensionDefinition dimDef = dim.getDefinition();
// For union or level MdmHierarchy, list the regions and default hierarchy
if ((kindOfDim.equals("UnionHier")) || (kindOfDim.equals("LevelHier")))
{
    try {
        System.out.println("    ");
        System.out.println("*****The following are the regions of " +
            dim.getName());
        List regions = ((MdmUnionDimensionDefinition)dimDef).getRegions();
        Iterator regIter = regions.iterator();
        while (regIter.hasNext()) {
            MdmDimension oneRegion = (MdmDimension) regIter.next();
            System.out.println("*****" + oneRegion.getName());
            if (oneRegion.hasMdmTag(MdmMetadataProvider.DEFAULT_HIERARCHY_TAG))
                System.out.println("***** (The " + oneRegion.getName() +
                    " region is the default MdmHierarchy)");
        }
    } catch (Exception e) {
        System.out.println("***Exception encountered : " + e.toString());
    }
}

// For union or level MdmHierarchy, get region info
if ((kindOfDim.equals("UnionHier")) || (kindOfDim.equals("LevelHier")))
{
    try {
        System.out.println("    ");
        System.out.println("*****Information about the regions of " +
            dim.getName() + ":");
        List regions = ((MdmUnionDimensionDefinition)dimDef).getRegions();
        Iterator regIter = regions.iterator();
        while (regIter.hasNext()) {
            MdmDimension oneRegion = (MdmDimension) regIter.next();
            getDimInfo(oneRegion, VERBOSE);
        }
    } catch (Exception e) {
        System.out.println("***Exception encountered : " + e.toString());
    }
}
```

```

    }
    }
    System.out.println("    ");
}

// *****

// Method for getting info about an MdmMeasure
public static void getMeasureInfo(MdmMeasure measure, int outputStyle) {
    System.out.println("*****Measure: " + measure.getName());
    if (outputStyle == VERBOSE) {

        // Get the dimensions of the MdmMeasure
        try {
            List mDims = measure.getDimensions();
            Iterator mDimIter = mDims.iterator();
            while (mDimIter.hasNext())
                System.out.println("*****Dimension of the Measure: " +
                    ((MdmDimension) mDimIter.next()).getName());
        } catch (Exception e) {
            System.out.println("*****Exception encountered : " + e.toString());
        }
    }
}
}

```

SampleMetadataDiscoverer プログラムの出力

このサンプル・プログラムの出力は、次のような Java 文で生成されたテキスト行で構成されています。

```
System.out.println("***Root MdmSchema: " + root.getName());
```

このコードは、戻り値が簡単のため `getName` メソッドを使用します。もう 1 つの方法として `getDescription` メソッドを使用できますが、出力は冗長になります。

プログラムが売上履歴スキーマで実行されている場合、出力には次の項目が含まれます。

- ルート MdmSchema の名前 ROOT
- ルート MdmSchema のメジャー MdmDimension の名前 MEASUREDIMENSION
- ルート MdmSchema の MdmDimension オブジェクトの名前と説明
- SH_CAT MdmSchema の MdmDimension および MdmMeasure オブジェクトの名前、説明および詳細

SH_CAT MdmSchema は、ルート MdmSchema の唯一のサブスキーマであるため、その MdmDimension オブジェクトはルート内のものと同じです。

- コードが PRODUCTS_DIM という名前の MdmDimension のプライマリ Source を取得したことを示す 2 行

次のとおり出力されます。一部の空白行は省略されています。

```
***Root MdmSchema: ROOT
*****Measure MdmDimension: MEASUREDIMENSION
***Schema: ROOT

*****
*****MdmDimension Name: CHANNELS_DIM
*****Description: Channel Values

*****
*****MdmDimension Name: CUSTOMERS_DIM
*****Description: Customer Dimension Values

*****
*****MdmDimension Name: PRODUCTS_DIM
*****Description: Product Dimension Values

*****
*****MdmDimension Name: PROMOTIONS_DIM
*****Description: Promotion Values

*****
*****MdmDimension Name: TIMES_DIM
*****Description: Time Dimension Values

*****

***Subschema: SH_CAT
***Schema: SH_CAT

*****

*****MdmDimension Name: CHANNELS_DIM
*****Description: Channel Values
*****Member Type: MdmStandardMemberType
*****Attribute: Long Description
*****Attribute: Short Description
*****CHANNELS_DIM is a union MdmHierarchy

*****The following are the regions of CHANNELS_DIM
*****CHANNEL_ROLLUP
***** (The CHANNEL_ROLLUP region is the default MdmHierarchy)

*****Information about the regions of CHANNELS_DIM:
```



```
*****MdmDimension Name: CHANNEL_ROLLUP
*****Description: Standard Channels
*****Member Type: MdmStandardMemberType
*****CHANNEL_ROLLUP is a level MdmHierarchy
*****Parent attribute: PARENTRELATION
*****Ancestors attribute: ANCESTORSRELATION
*****Region attribute: LEVELRELATION

*****The following are the regions of CHANNEL_ROLLUP
*****CHANNEL_TOTAL
*****CHANNEL_CLASS
*****CHANNEL

*****Information about the regions of CHANNEL_ROLLUP:
*****MdmDimension Name: CHANNEL_TOTAL
*****Description: Channel Total for the standard hierarchy
*****Member Type: MdmStandardMemberType
*****CHANNEL_TOTAL is an MdmLevel

*****MdmDimension Name: CHANNEL_CLASS
*****Description: Channel Class level of the standard hierarchy
*****Member Type: MdmStandardMemberType
*****CHANNEL_CLASS is an MdmLevel

*****MdmDimension Name: CHANNEL
*****Description: Channel level of the standard hierarchy
*****Member Type: MdmStandardMemberType
*****CHANNEL is an MdmLevel

*****

*****MdmDimension Name: CUSTOMERS_DIM
*****Description: Customer Dimension Values
*****Member Type: MdmStandardMemberType
*****Attribute: Long Description
*****Attribute: Short Description
*****Attribute: First Name
*****Attribute: Last Name
*****Attribute: Gender
*****Attribute: Marital Status
*****Attribute: Year of Birth
*****Attribute: Income Level
*****Attribute: Credit Limit
*****Attribute: Street Address
*****Attribute: Postal Code
*****Attribute: Phone Number
*****Attribute: E-mail
```

```
*****CUSTOMERS_DIM is a union MdmHierarchy

*****The following are the regions of CUSTOMERS_DIM
*****GEOG_ROLLUP
***** (The GEOG_ROLLUP region is the default MdmHierarchy)
*****CUST_ROLLUP

*****Information about the regions of CUSTOMERS_DIM:
*****MdmDimension Name: GEOG_ROLLUP
*****Description: Standard
*****Member Type: MdmStandardMemberType
*****GEOG_ROLLUP is a level MdmHierarchy
*****Parent attribute: PARENTRELATION
*****Ancestors attribute: ANCESTORSRELATION
*****Region attribute: LEVELRELATION

*****The following are the regions of GEOG_ROLLUP
*****GEOG_TOTAL
*****REGION
*****SUBREGION
*****COUNTRY
*****STATE
*****CITY
*****CUSTOMER

*****Information about the regions of GEOG_ROLLUP:
*****MdmDimension Name: GEOG_TOTAL
*****Description: Geography Total for the standard CUSTOMER hierarchy
*****Member Type: MdmStandardMemberType
*****GEOG_TOTAL is an MdmLevel

*****MdmDimension Name: REGION
*****Description: Region level of the standard CUSTOMER hierarchy
*****Member Type: MdmStandardMemberType
*****REGION is an MdmLevel

*****MdmDimension Name: SUBREGION
*****Description: Subregion level of the standard CUSTOMER hierarchy
*****Member Type: MdmStandardMemberType
*****SUBREGION is an MdmLevel

*****MdmDimension Name: COUNTRY
*****Description: Country level of the standard CUSTOMER hierarchy
*****Member Type: MdmStandardMemberType
*****COUNTRY is an MdmLevel

*****MdmDimension Name: STATE
```

```
*****Description: State level of the standard CUSTOMER hierarchy
*****Member Type: MdmStandardMemberType
*****STATE is an MdmLevel

*****MdmDimension Name: CITY
*****Description: City level of the standard CUSTOMER hierarchy
*****Member Type: MdmStandardMemberType
*****CITY is an MdmLevel

*****MdmDimension Name: CUSTOMER
*****Description: Customer level of standard CUSTOMER hierarchy
*****Member Type: MdmStandardMemberType
*****CUSTOMER is an MdmLevel

*****MdmDimension Name: CUST_ROLLUP
*****Description: Standard
*****Member Type: MdmStandardMemberType
*****CUST_ROLLUP is a level MdmHierarchy
*****Parent attribute: PARENTRELATION
*****Ancestors attribute: ANCESTORSRELATION
*****Region attribute: LEVELRELATION

*****The following are the regions of CUST_ROLLUP
*****CUST_TOTAL
*****STATE
*****CITY
*****CUSTOMER

*****Information about the regions of CUST_ROLLUP:
*****MdmDimension Name: CUST_TOTAL
*****Description: Customer Total for the standard CUSTOMER hierarchy
*****Member Type: MdmStandardMemberType
*****CUST_TOTAL is an MdmLevel

*****MdmDimension Name: STATE
*****Description: State level of the standard CUSTOMER hierarchy
*****Member Type: MdmStandardMemberType
*****STATE is an MdmLevel

*****MdmDimension Name: CITY
*****Description: City level of the standard CUSTOMER hierarchy
*****Member Type: MdmStandardMemberType
*****CITY is an MdmLevel

*****MdmDimension Name: CUSTOMER
*****Description: Customer level of standard CUSTOMER hierarchy
```

```
*****Member Type: MdmStandardMemberType
*****CUSTOMER is an MdmLevel

*****

*****MdmDimension Name: PRODUCTS_DIM
*****Description: Product Dimension Values
*****Member Type: MdmStandardMemberType
*****Attribute: Long Description
*****Attribute: Short Description
*****PRODUCTS_DIM is a union MdmHierarchy

*****The following are the regions of PRODUCTS_DIM
*****PROD_ROLLUP
***** (The PROD_ROLLUP region is the default MdmHierarchy)

*****Information about the regions of PRODUCTS_DIM:
*****MdmDimension Name: PROD_ROLLUP
*****Description: Standard
*****Member Type: MdmStandardMemberType
*****PROD_ROLLUP is a level MdmHierarchy
*****Parent attribute: PARENTRELATION
*****Ancestors attribute: ANCESTORSRELATION
*****Region attribute: LEVELRELATION

*****The following are the regions of PROD_ROLLUP
*****PROD_TOTAL
*****CATEGORY
*****SUBCATEGORY
*****PRODUCT

*****Information about the regions of PROD_ROLLUP:
*****MdmDimension Name: PROD_TOTAL
*****Description: Product Total for the standard PRODUCT hierarchy
*****Member Type: MdmStandardMemberType
*****PROD_TOTAL is an MdmLevel

*****MdmDimension Name: CATEGORY
*****Description: Category level of standard PRODUCT hierarchy
*****Member Type: MdmStandardMemberType
*****CATEGORY is an MdmLevel

*****MdmDimension Name: SUBCATEGORY
*****Description: Sub-category level of standard PRODUCT hierarchy
*****Member Type: MdmStandardMemberType
*****SUBCATEGORY is an MdmLevel
```

```
*****MdmDimension Name: PRODUCT
*****Description: Product level of standard PRODUCT hierarchy
*****Member Type: MdmStandardMemberType
*****PRODUCT is an MdmLevel

*****

*****MdmDimension Name: PROMOTIONS_DIM
*****Description: Promotion Values
*****Member Type: MdmStandardMemberType
*****Attribute: Long Description
*****Attribute: Short Description
*****PROMOTIONS_DIM is a union MdmHierarchy

*****The following are the regions of PROMOTIONS_DIM
*****PROMO_ROLLUP
***** (The PROMO_ROLLUP region is the default MdmHierarchy)

*****Information about the regions of PROMOTIONS_DIM:
*****MdmDimension Name: PROMO_ROLLUP
*****Description: Standard Promotions
*****Member Type: MdmStandardMemberType
*****PROMO_ROLLUP is a level MdmHierarchy
*****Parent attribute: PARENTRELATION
*****Ancestors attribute: ANCESTORSRELATION
*****Region attribute: LEVELRELATION

*****The following are the regions of PROMO_ROLLUP
*****PROMO_TOTAL
*****CATEGORY
*****SUBCATEGORY
*****PROMO

*****Information about the regions of PROMO_ROLLUP:
*****MdmDimension Name: PROMO_TOTAL
*****Description: Promotions Total for the standard PROMOTION hierarchy
*****Member Type: MdmStandardMemberType
*****PROMO_TOTAL is an MdmLevel

*****MdmDimension Name: CATEGORY
*****Description: Category level of the standard PROMOTION hierarchy
*****Member Type: MdmStandardMemberType
*****CATEGORY is an MdmLevel

*****MdmDimension Name: SUBCATEGORY
```

```

*****Description: Sub-category level of the standard PROMOTION hierarchy
*****Member Type: MdmStandardMemberType
*****SUBCATEGORY is an MdmLevel

*****MdmDimension Name: PROMO
*****Description: Promotion level of the standard PROMOTION hierarchy
*****Member Type: MdmStandardMemberType
*****PROMO is an MdmLevel

*****

*****MdmDimension Name: TIMES_DIM
*****Description: Time Dimension Values
*****Member Type: MdmStandardMemberType
*****Attribute: Long Description
*****Attribute: Short Description
*****Attribute: Period Number
*****Attribute: Period Number of Days
*****Attribute: Period End Date
*****TIMES_DIM is a union MdmHierarchy

*****The following are the regions of TIMES_DIM
*****CAL_ROLLUP
***** (The CAL_ROLLUP region is the default MdmHierarchy)
*****FIS_ROLLUP

*****Information about the regions of TIMES_DIM:
*****MdmDimension Name: CAL_ROLLUP
*****Description: Calendar
*****Member Type: MdmStandardMemberType
*****CAL_ROLLUP is a level MdmHierarchy
*****Parent attribute: PARENTRELATION
*****Ancestors attribute: ANCESTORSRELATION
*****Region attribute: LEVELRELATION

*****The following are the regions of CAL_ROLLUP
*****YEAR
*****QUARTER
*****MONTH
*****DAY

*****Information about the regions of CAL_ROLLUP:
*****MdmDimension Name: YEAR
*****Description: Year level of the Calendar hierarchy
*****Member Type: MdmStandardMemberType
*****YEAR is an MdmLevel

```

```
*****MdmDimension Name: QUARTER
*****Description: Quarter level of the Calendar hierarchy
*****Member Type: MdmStandardMemberType
*****QUARTER is an MdmLevel

*****MdmDimension Name: MONTH
*****Description: Month level of the Calendar hierarchy
*****Member Type: MdmStandardMemberType
*****MONTH is an MdmLevel

*****MdmDimension Name: DAY
*****Description: Day level of the Calendar hierarchy
*****Member Type: MdmStandardMemberType
*****DAY is an MdmLevel

*****MdmDimension Name: FIS_ROLLUP
*****Description: Fiscal
*****Member Type: MdmStandardMemberType
*****FIS_ROLLUP is a level MdmHierarchy
*****Parent attribute: PARENTRELATION
*****Ancestors attribute: ANCESTORSRELATION
*****Region attribute: LEVELRELATION

*****The following are the regions of FIS_ROLLUP
*****FIS_YEAR
*****FIS_QUARTER
*****FIS_MONTH
*****FIS_WEEK
*****DAY

*****Information about the regions of FIS_ROLLUP:
*****MdmDimension Name: FIS_YEAR
*****Description: Year level of the Fiscal hierarchy
*****Member Type: MdmStandardMemberType
*****FIS_YEAR is an MdmLevel

*****MdmDimension Name: FIS_QUARTER
*****Description: Quarter level of the Fiscal hierarchy
*****Member Type: MdmStandardMemberType
*****FIS_QUARTER is an MdmLevel

*****MdmDimension Name: FIS_MONTH
*****Description: Month level of the Fiscal hierarchy
*****Member Type: MdmStandardMemberType
*****FIS_MONTH is an MdmLevel
```

```
*****MdmDimension Name: FIS_WEEK
*****Description: Week level of the Fiscal hierarchy
*****Member Type: MdmStandardMemberType
*****FIS_WEEK is an MdmLevel

*****MdmDimension Name: DAY
*****Description: Day level of the Calendar hierarchy
*****Member Type: MdmStandardMemberType
*****DAY is an MdmLevel

*****

*****Measure: SALES_QUANTITY
*****Dimension of the Measure: CHANNELS_DIM
*****Dimension of the Measure: CUSTOMERS_DIM
*****Dimension of the Measure: PRODUCTS_DIM
*****Dimension of the Measure: PROMOTIONS_DIM
*****Dimension of the Measure: TIMES_DIM

*****Measure: SALES_AMOUNT
*****Dimension of the Measure: CHANNELS_DIM
*****Dimension of the Measure: CUSTOMERS_DIM
*****Dimension of the Measure: PRODUCTS_DIM
*****Dimension of the Measure: PROMOTIONS_DIM
*****Dimension of the Measure: TIMES_DIM

*****Measure: UNIT_PRICE
*****Dimension of the Measure: PRODUCTS_DIM
*****Dimension of the Measure: TIMES_DIM

*****Measure: UNIT_COST
*****Dimension of the Measure: PRODUCTS_DIM
*****Dimension of the Measure: TIMES_DIM

***Making a Source object for PRODUCTS_DIM
*****Made the Source
```

問合せの概要

この章では、問合せの結果を表すデータ・セットの仕様である Source オブジェクトについて説明します。Source メソッドを使用して問合せを作成する方法の詳細および例は、[第 6 章「Source メソッドを使用した問合せの作成」](#)を参照してください。Template オブジェクトを使用して問合せを作成する方法の詳細は、[第 10 章「動的問合せの作成」](#)を参照してください。

この章では、次の項目について説明します。

- [Source オブジェクトの特性](#)
- [Source オブジェクトの作成](#)

Source オブジェクトの特性

OLAP API のデータ・モデルは一意です。リレーショナル・モデルまたは多次元モデルとは異なります。OLAP API では、問合せの結果を表すデータ・セットの仕様は、Source クラスまたはそのサブクラス（表 5-1 を参照）のインスタンスによって表されます。各 Source は、問合せを作成した操作を定義する SourceDefinition と対になっています。

表 5-1 Source クラスのサブクラス

サブクラス	値の Java 型	OLAP API データ型
BooleanSource	boolean 値	Boolean
DateSource	Java Date オブジェクト	Date
NumberSource	double、float、int または short 型の値、またはこれらの数値の組合せ	Double、Float、Integer、Short または Number
StringSource	Java String オブジェクト	String

Source オブジェクトは不変です。Source オブジェクトは、一度作成すると変更できません。Source オブジェクトをユーザーに変更可能で提供する場合は、Template オブジェクトによって定義された Source オブジェクトを使用します。Template オブジェクトは状態を持ち、いつでも変更できます。Template オブジェクトを使用して問合せを作成する方法の詳細は、第 10 章「動的問合せの作成」を参照してください。

Source オブジェクトは、単なるデータ・セットの仕様であり、このオブジェクト自体には実際のデータは含まれません。ただし、このオブジェクトを、このオブジェクトが定義する結果セットとみなすことはできます。この観点から見ると、Source は型および構造（入力および出力）を持ちます。

Source の型

すべての Source オブジェクトは型を持ちます。OLAP API では、Source の型は他の Source オブジェクトとして表され、Source はここからその値を取得します。Source の型を取得するには、getType メソッドを使用します。

OLAP API は、FundamentalMetadataObject を提供して、各基本 Java データ型、Java String オブジェクトおよび Java Date オブジェクトを表します。これらのオブジェクトを、OLAP API データ型といいます。表 5-2 に、Source オブジェクトの OLAP API データ型および相互関係を示します。OLAP API データ型を表す Source オブジェクトを作成するには、5-8 ページの「OLAP API データ型を表す Source オブジェクトの作成」に示す手順を実行します。Source の OLAP API 型を取得するには、getDataType メソッドを使用します。

多くの場合、新しい Source を作成する操作によって、その Source の型が決定します。たとえば、値が各顧客を示す一意の数値識別子である customer という名前の Source オブ

ジェクトが存在すると想定します。customer の OLAP API 型は Integer です。さらに、customer の select メソッドを使用して、customerSelection という名前の別の Source オブジェクトを作成すると想定します。この場合、customerSelection の OLAP API 型は customer です。

表 5-2 Source オブジェクトの OLAP API データ型

OLAP API データ型	説明
Value	OLAP API のいずれかのデータ型の Source オブジェクト
Boolean	値が Java boolean データ型の Source オブジェクト
Date	値が Java Date オブジェクトの Source オブジェクト
Number	OLAP API のいずれかの数値データ型の Source オブジェクト
Double	値が Java double データ型の Source オブジェクト
Float	値が Java float データ型の Source オブジェクト
Integer	値が Java int データ型の Source オブジェクト
Short	値が Java short データ型の Source オブジェクト
String	値が Java String オブジェクトの Source オブジェクト
Empty	定義された値を持たない Source オブジェクト
Null	1 つの null 値を持つ Source オブジェクト

Source の構造 : 入力および出力

(Empty 型の Source 以外の) すべての Source オブジェクトは値を持ちます。場合によっては、Source の値が一意のデータ項目で、それ自体で意味を持つ場合があります。リレーショナルの概念では、このタイプの Source は、Source の値を含む 1 列の表と考えることができます。多次元の概念では、このタイプの Source は、ディメンションと考えることができます。

Source の値が一意のデータ項目ではないため、それ自体では意味を持たない場合があります。Source の値は、他の Source の値との関係でのみ意味を持ちます。この場合、Source の構造は、入力および出力という他の Source オブジェクトによって決定されます。これらの他の Source オブジェクトが入力または出力のどちらであるかは、値が指定されているかどうかによって決定します。

- **出力:** 値が指定されている場合、他の Source オブジェクトは出力と呼ばれます。Source の値は、一連の出力値によって識別されます。Source の出力を取得するには、getOutputs メソッドを使用します。
- **入力:** 値が指定されていない場合、他の Source オブジェクトは入力と呼ばれます。入力を持つ Source は、値を決定できない結果セットです。リレーショナルの概念では、

入力、Source の値に対するキーとして機能する、SQL 文の GROUP BY リスト内の列と考えることができます。多次元概念では、入力は、ディメンション・リストに含まれていない Source のディメンションと考えることができます。Source の入力を取得するには、getInputs メソッドを使用します。

Source の入力および出力は、Oracle OLAP による Source の処理方法を決定します。たとえば、Cursor が Source 上でオープンされている場合、Oracle OLAP は、データを作成するためにその出力全体でループを行いますが、これによってすべての入力が（任意に）排除されます。入力を含む Source の 1 つ以上の値を取得するには、Source の必要な値を一意に識別する入力の値を指定する必要があります。入力の値を指定する順序によって、Source の構造および処理が決定します。値を最初に指定する入力は、最も遅く変化する出力になります。入力値を指定する方法の詳細は、6-2 ページの「[出力値に基づいた選択](#)」および 6-3 ページの「[Source の構造に対する入出力の順序の影響](#)」を参照してください。

さらに、Source が入力と出力の両方を持つ場合、Source の値はその出力値のセットによって識別され、通常、出力される値の各セットは特定の数の値（データのサブセット）を識別します。一部の Source メソッドは、これらのデータのサブセットを処理します。たとえば、Oracle OLAP は、位置（各サブセットの最初の値が位置 1）に基づいて値を選択するメソッドを処理する場合、または average や total などの集計メソッドを処理する場合、Source の出力全体でループを行います。位置指定メソッドおよび集計メソッドの詳細は、6-5 ページの「[値の位置の検索](#)」および 6-20 ページの「[集計メソッドの使用](#)」を参照してください。

Source オブジェクトの作成

OLAP API を使用して問合せを作成するプロセスでは、多くの異なる Source オブジェクトが作成されます。このプロセスの概要を次に示します。

1. メタデータ・オブジェクトに対応する Source オブジェクトを作成します（5-5 ページの「[メタデータ・オブジェクトからの Source オブジェクトの取得](#)」を参照）。これらの Source オブジェクト（**プライマリ Source オブジェクト**ともいう）は、作成元のメタデータ・オブジェクトと類似した構造をしています。
2. プライマリ Source オブジェクトのメソッドをコールして問合せを開始します。Source クラスおよびそのサブクラスのメソッドは、新しい Source オブジェクト（**導出 Source オブジェクト**ともいう）を戻します。必要な結果が得られるまで追加の Source オブジェクトを導出して分析を続行し、プログラムにデータを取得します。導出 Source オブジェクトおよびそれらのオブジェクトを作成するメソッドの概要は、5-6 ページの「[Source メソッドを使用した新しい Source オブジェクトの作成](#)」を参照してください。詳細は、OLAP API Javadoc を参照してください。Source メソッドを使用して選択および一般的な分析操作を実行する方法の詳細は、第 6 章「[Source メソッドを使用した問合せの作成](#)」を参照してください。

問合せプロセスの一部として、単純な非ディメンション Source オブジェクトを作成して選択および計算を行う際にオペランドとして使用する場合、または OLAP API データ型を表す Source オブジェクトを作成する場合があります。これらのタイプの Source オブジェクトを作成する方法の詳細は、5-7 ページの「[単純な非ディメンション Source](#)」

[オブジェクトの作成](#) および 5-8 ページの「[OLAP API データ型を表す Source オブジェクトの作成](#)」を参照してください。

- 3. Source オブジェクトで表されるデータ・セットを取得する際、Cursor を作成します ([第 9 章「問合せ結果の取得」](#)を参照)。

メタデータ・オブジェクトからの Source オブジェクトの取得

メタデータ・オブジェクトから Source オブジェクトを取得するには、次の手順を実行します。

- 1. 対応する Source オブジェクトを作成するメタデータ・オブジェクトを作成します ([第 2 章](#)を参照)。
- 2. getSource メソッドを使用して、メタデータ・オブジェクトから Source オブジェクトを作成します。

MdmDimension、MdmHierarchy または MdmLevel オブジェクトからの Source の作成

MdmDimension、MdmHierarchy または MdmLevel の getSource メソッドをコールして作成した Source には、入力または出力が存在しません。この Source は、値の単純なリストの仕様です。

2-12 ページの「[暦年用のレベル MdmHierarchy](#)」では、mdmTimesDimCalHier という名前の MdmHierarchy を作成する方法を説明しています。timesDimCalHier という名前の Source を mdmTimesDimCalHier から作成するには、[例 5-1](#) のコードを使用します。

例 5-1 MdmHierarchy の Source の取得

Source timesDimCalHier = mdmTimesDimCalHier.getSource;

timesDimCalHier という名前の Source は、1529 個の値（年を表す 4 つの値、四半期を表す 16 個の値、月を表す 48 個の値および日付を表す 1461 個の値）を含む単純な非索引付けリストで構成されています。

timesDimCalHier の値
1998
1998-Q1
1998-01
01-JAN-98
02-JAN-98
03-JAN-98

timesDimCalHier の値
...
1998-Q2
1998-04
01-APR-98
...
1999
...

MdmMeasure または MdmAttribute オブジェクトからの Source の作成

MdmMeasure または MdmAttribute の getSource メソッドをコールして作成した Source は、1 つ以上の入力を持つデータ・セットの仕様です。各入力は、MdmDimension から作成されたプライマリ Source です。そのため、MdmMeasure または MdmAttribute から作成した Source によって表されるデータ・セットの仕様は、完全ではありません。結果的に、これらの Source オブジェクトの Cursor を作成できないため、アプリケーションに値を取得できません。MdmMeasure または MdmAttribute から作成された Source の値を取得するには、ディメンションの機能を果たす Source オブジェクトの値に対して値を指定して、そこから新しい Source を導出する必要があります（6-2 ページの「[出力値に基づいた選択](#)」を参照）。

2-19 ページの「[2 つの MdmDimension オブジェクトを持つ MdmMeasure](#)」では、mdmUnitCost という名前の MdmMeasure を作成する方法を説明しています。unitCost という名前の Source を mdmUnitCost から作成するには、[例 5-2](#) のコードを使用します。

例 5-2 MdmMeasure の Source の取得

```
Source unitCost = mdmUnitCost.getSource;
```

mdmUnitCost には MdmDimension オブジェクトとして mdmProductsDim および mdmTimesDim が存在するため、unitCost は 2 つの入力（productsDim および timesDim）を持ちます。unitCost の 1 つ以上の値を取得するには、unitCost の値を一意に識別する入力（productsDim および timesDim）の値を指定する必要があります。入力値を指定する方法の詳細は、6-2 ページの「[出力値に基づいた選択](#)」を参照してください。

Source メソッドを使用した新しい Source オブジェクトの作成

ほとんどの OLAP 問合せでは、Source クラスのメソッドを使用して、既存の Source オブジェクトから新しい Source オブジェクトが導出されます。

[表 5-1](#) に、OLAP API で最も重要な Source メソッドの概要を示します。

表 5-3 主な Source メソッド

メソッド	説明
join	OLAP API で最も重要な Source 作成メソッドです。join メソッドは、ベース Source と他の Source（結合 Source という）の値を組み合わせ、3 つ目の Source（比較 Source という）を使用してこのデータ・セットを指定された方法でフィルタ処理することによって、新しい Source を作成します。join メソッドでオプションのパラメータを使用して、結合 Source を新しい Source への出力として追加することもできます。
alias	ベース Source オブジェクトと同じであるが、型としてベース Source を持つ新しい Source オブジェクトを作成します。
distinct	ベース Source オブジェクトと同じであるが、すべてのコピー行（タプル）が削除された新しい Source オブジェクトを作成します。
position	ベース Source と同じ構造で、値がベース Source の値の位置である新しい Source オブジェクトを作成します。
value	ベース Source の値、および入力としてベース Source を持つ新しい Source オブジェクトを作成します。

OLAP API は、表 5-3「主な Source メソッド」に示すメソッドのかわりに使用可能な様々なメソッドを提供します。これらのメソッドは、join メソッドのバリエーションの他に、appendValue、at、cumulativeInterval、first、ge、interval、selectValues、sortAscending などのメソッドを含みます。これらすべてのメソッドの詳細は、OLAP API Javadoc を参照してください。これらのメソッドを使用してデータを分析する方法の詳細は、第 6 章「Source メソッドを使用した問合せの作成」を参照してください。

単純な非ディメンション Source オブジェクトの作成

DataProvider クラスの createConstantSource、createListSource および createRangeSource メソッドを使用して、メタデータ・オブジェクトまたは他の Source オブジェクトに基づかない、オペランドとして使用可能な単純な非ディメンション Source オブジェクトを作成できます。これらの Source オブジェクトは、定数 Source オブジェクト、リスト Source オブジェクトおよび範囲 Source オブジェクトともいいます。

アプリケーションで使用される myDataProvider という名前の DataProvider を表すオブジェクトが存在すると想定します。また、演算目的用に 1 つの値 4 を含む Source が必要であると想定します。この Source を作成するには、例 5-3 に示すコードを発行します。

例 5-3 定数 Source の作成

```
Number Javadoc myConstantFour = myDataProvider.createConstantSource(4);
```

OLAP API データ型を表す Source オブジェクトの作成

FundamentalMetadataProvider のメソッドを使用して、OLAP API データ型を表すオブジェクトを取得できます。各メソッドは、FundamentalMetadataObject を戻します。[表 5-4](#) に、OLAP API データ型およびそれらの取得に使用するメソッドを示します。

表 5-4 OLAP API データ型を表すオブジェクトを取得するメソッド

OLAP API データ型	このデータ型を取得するメソッド
Value	getValueDataType
Boolean	getBooleanDataType
Date	getDateDataType
Number	getNumberDataType
Double	getDoubleDataType
Float	getFloatDataType
Int	getIntegerDataType
Short	getShortDataType
String	getStringDataType
Empty	getEmptyDataType 空の Source を取得するには、 DataProvider.getEmptySource メソッドを使用します。
Null	getVoidDataType null の Source を取得するには、 DataProvider.getVoidSource メソッドを使用します。

OLAP API データ型を表す Source オブジェクトを作成するには、次の手順を実行します。

1. DataProvider クラスの getFundamentalMetadataProvider メソッドを使用して、FundamentalMetadataProvider を取得します。
2. FundamentalMetadataProvider クラスの適切なメソッドを使用して、OLAP API データ型を表す FundamentalMetadataObject オブジェクトを作成します。
3. FundamentalMetadataObject.getSource メソッドを使用して、手順 1 で戻されたオブジェクトから Source を作成します。

[例 5-4](#) では、OLAP API Boolean データ型を表す、olapBooleanDataType という名前の Source オブジェクトが作成されます。olapBooleanDataType を使用して、他の Source の OLAP API データ型が Boolean かどうかを確認できます。

例 5-4 OLAP API Boolean データ型の Source の作成

```
FundamentalMetadataObject myFundamentalMetadataProvider =  
    myDataProvider.getFundamentalMetadataProvider();  
FundamentalMetadataObject olapBooleanFundObj =  
    myFundamentalMetadataProvider.getBooleanType();  
Source olapBooleanDataType = olapBooleanFundObj.getSource();
```

Source メソッドを使用した問合せの作成

多くの問合せは、Source メソッドの Source オブジェクトをコールして新しい Source オブジェクトを作成することによって作成されます。Source メソッドの概要は、5-6 ページの「[Source メソッドを使用した新しい Source オブジェクトの作成](#)」を参照してください。詳細は、OLAP API Javadoc を参照してください。この章では、Source メソッドを使用して問合せを作成する方法について説明します。

この章では、次の項目について説明します。

- [Source 値に基づいた選択](#)
- [出力値に基づいた選択](#)
- [ランクに基づいた値の選択](#)
- [階層での位置付けに基づいた値の選択](#)
- [セルフ・リレーション Source の作成](#)
- [数値分析の実行](#)
- [文字列値の操作](#)

Source 値に基づいた選択

ベース Source の特定の値のみを選択することによって、既存の Source から新しい Source を作成できます。通常、次のいずれかのメソッドを使用して、ベース値の値に基づいて選択を行います。

```
Source.select(BooleanSource)
Source.selectValue(Source)
Source.selectValues(Source)
Source.selectValues(Source[])
BooleanSource.selectValue(boolean)
BooleanSource.selectValues(boolean[])
NumberSource.selectValue(double)
NumberSource.selectValue(int)
NumberSource.selectValue(float)
NumberSource.selectValue(short)
NumberSource.selectValues(double[])
NumberSource.selectValues(float[])
NumberSource.selectValues(int[])
NumberSource.selectValues(short[])
StringSource.selectValue(String)
StringSource.selectValues(String[])
```

次の構文で、join メソッドを使用して値を選択することもできます。

```
Source Source::join (Source joined,
    Source comparison,
    Source.COMPARISON_RULE_SELECT,
    boolean visible);
```

mdmTimesDim という名前の MdmDimension オブジェクトから作成し、値がカレンダー値である timesDim という名前のプライマリ Source オブジェクトが存在すると想定します。1996 に対応する値のみを選択するには、[例 6-1](#) に示すコードを発行します。

例 6-1 Source 値に基づいた選択

```
Source timesSel = timesDim.selectValue("1996");
```

出力値に基づいた選択

Source オブジェクト用の Cursor を作成する場合、そのオブジェクトは入力を持つことができません。MdmMeasure または MdmAttribute から作成されたすべての Source は入力を持つため、入力の値を指定する必要があります。OLAP API には、この非常に一般的な操作をサポートする特別な join メソッドが含まれています。

Source の入力値を指定することを、入力から出力への切替えといいます。この意味では、Source を (getInputs メソッドによって戻される) 入力のリストから (getOutputs メソッドによって戻される) 出力のリストに移動することは、SQL の GROUP BY リストから列を移動することと似ています。

join メソッドを使用した入力から出力への切替え

Source の入力値を指定して入力を出力に切り替えるには、次の join メソッドを使用します。ここで、元の Source は、出力に切り替える入力を持つ Source オブジェクトであり、結合された Source は、切り替える入力です。

```
Source newSource = base.join (Source joined);
```

これは、次の join メソッドのショートカットです。

```
Source newSource = base.join (joined, emptySource, Source.COMPARISON_RULE_REMOVE, true);
```

比較 Source は、値を持たない空の Source です。このため、COMPARISON_RULE_REMOVE 定数が指定されている場合でも、比較の結果、削除される値は存在しません。また、visible フラグが true に設定されているため、結合 Source は、新しい Source の出力になります。

Source クラスおよびそのサブクラスの多くのメソッドは、join メソッドを暗黙的にコールするメソッドであるため、これらのメソッドの一部を使用して、入力を出力に切り替えることもできます。

Source の構造に対する入出力の順序の影響

Source の構造は、Source の入力を出力に切り替える順序によって決定されます。出力を持つ Source の場合、最初に作成された出力は、最も速く変化する出力になります。最後に作成された出力は、最も遅く変化する出力になります。

2 つの join メソッドを単一の文に並列する場合、左端から最初の join が、最初に処理されます。そのため、複数の join メソッドを含む単一の文を作成する場合、新しい Source の最も速く変化する入力を、文中最初の join で結合する Source にする必要があります。

mdmUnitCost という名前の MdmMeasure オブジェクトから作成した unitCost という名前のプライマリ Source が存在すると想定します。unitCost という名前の Source は、timesDim および productsDim の入力を持ちますが、出力は持ちません。timesDim および productsDim Source オブジェクトは、入力も出力も持ちません。unitCost の入力を出力に切り替える順序によって、Cursor を作成できる Source の構造が決定されます。例 6-2 に、最初に timesDim に結合した場合の結果を示します。例 6-3 に、最初に productsDim に結合した場合の結果を示します。

最初に作成される出力が timesDim の場合の入力から出力への切替え

例 6-2 のコードを発行して、unitCost という名前のプライマリ Source の入力を出力に切り替えると想定します。

例 6-2 最初に作成される出力が timesDim の場合の入力から出力への切替え

```
Source newSource = unitCost.join(timesDim).join(productsDim);
```

このコードには、2 つの join メソッドが並列されています。
unitCost.join(timesDim) が最初に処理されるため、timesDim の出力値が最初に指定されます。timesDim は、新しい Source に対して最初に定義される出力です。最初の join が処理された後、結果の Source（名前なし）によって表されるデータ・セットは、次の構造を持ちます。

timesDim（出力 1）	unitCost の値
1998	4,000 500
31-DEC-01	9 500

2 番目の join が処理された後、newSource によって表されるデータ・セットは、両方の出力（timesDim および productsDim）の名前と値で構成されます。timesDim は、値が指定された最初の出力であるため、最も速く変化する出力となり、新しい Source の構造は、次のようになります。

productsDim（出力 2）	timesDim（出力 1）	unitCost の値
Boys	1998	4,000
Boys	31-DEC-01	10
49780	1998	500
49780	31-DEC-01	9

最初に作成される出力が productsDim の場合の入力から出力への切替え

例 6-3 のコードを発行して、unitCost の入力を出力に切り替えると想定します。

例 6-3 最初に作成される出力が productsDim の場合の入力から出力への切替え

```
Source newSource = unitCost.join(productsDim).join(timesDim);
```

例 6-3 のコードには、2 つの join メソッドが並列されています。unitCost.join(productsDim) が最初に処理されるため、productsDim は、新しい Source に対して最初に定義される出力です。そのため、productsDim は、最も速く変化する出力であり、新しい Source は、次に示す構造を持ちます。

timesDim (出力 2)	productsDim (出力 1)	unitCost の値
1998	Boys	4,000
1998	49780	500
31-DEC-01	Boys	10
31-DEC-01	48780	9

出力値および Source 値に基づいた選択 : 例

MdmDimension オブジェクトから作成した productsDim、promotionsDim、channelsDim および timesDim という名前の 4 つのプライマリ Source オブジェクトが存在すると想定します。また、MdmMeasure オブジェクトから作成した sales という名前のプライマリ Source オブジェクトが存在すると想定します。productsDim、promotionsDim、channelsDim および timesDim オブジェクトは、出力を持ちません。sales オブジェクトは、入力として productsDim、promotionsDim、channelsDim および timesDim を持ちます。

1996 年の売上が 10,000,000 ドルを超えたすべての製品を値として持つ、bigSeller という名前の新しい Source を作成するには、例 6-4 のコードを発行します。

例 6-4 出力値および Source 値に基づいた選択

```
Source promotionSel = promotionsDim.selectValue("Promo total");
Source channelSel = channelsDim.selectValue("Channel total");
Source timeSel = timesDim.selectValue("1996");
Source bigSellers = productsDim.select(sales.gt(10000000)).
    join(promotionSel).join(timeSel).join(channelSel);
```

ランクに基づいた値の選択

Source が 1 つまたは複数の属性に従ってソートされている場合、この Source の値の位置は、ある種類のランキング（一意ランキング）を表します。この他に、一意でない様々なランキングがありますが、それらは可変ランキングと呼ばれます。

値の位置の検索

表 6-1 に示すメソッドを使用すると、Source 内の位置に基づいて値を検索できます。また、指定した 1 つまたは複数の値によって値の位置を検索することもできます。OLAP API で

は、位置は1から始まる値です。Source が入力を持たない場合は、位置は Source 値のセット全体に対応し、1つの値のみが位置1を持ちます (6-7 ページの「[入力または出力を持たない場合の値の位置の検索](#)」を参照)。Source が入力を持つ場合は、位置は出力値の一意の各セットによって識別される Source の値のサブセットに対応し、各サブセットの最初の値が位置1を持ちます (6-7 ページの「[出力および入力を持たない場合の値の位置の検索](#)」を参照)。

表 6-1 位置に基づいて値を検索するメソッド

メソッド	説明
<code>position()</code>	ベース Source を入力として持つ、Integer 型の新しい Source を作成します。値は、ベース Source の値の位置で、1から始まります。ベース Source が1つまたは複数の属性に従ってソートされている場合、その位置は、ある種類のランキング（一意ランキング）を表します。
<code>at(pos)</code>	ベース Source と同じ構造で、値がベース Source 内の指定された位置に存在する値のみである新しい Source を作成します。このメソッドには2つのバージョンが存在します。一方のバージョンを使用すると、Source オブジェクトを使用して値の位置を指定できます。他方のバージョンを使用すると、int 値を使用して値の位置を指定できます。
<code>first()</code>	ベース Source と同じ構造で、値がベース Source 内の位置1に存在する値のみである新しい Source を作成します。
<code>last()</code>	ベース Source と同じ構造で、値がベース Source 内の最後の位置に存在する値のみである新しい Source を作成します。
<code>positionOfValue(value)</code>	ベース Source と同じ構造で、値がベース Source 内の指定された値の位置である新しい Source を作成します。このメソッドには2つのバージョンが存在します。一方のバージョンを使用すると、Source を使用して値を指定できます。他方のバージョンを使用すると、String を使用して値を指定できます。
<code>positionOfValues(values)</code>	ベース Source と同じ構造で、値がベース Source 内の指定された複数の値の位置である新しい Source を作成します。このメソッドには2つのバージョンが存在します。一方のバージョンを使用すると、Source を使用して値を指定できます。他方のバージョンを使用すると、String オブジェクトの配列を使用して値を指定できます。

入力または出力を持たない場合の値の位置の検索

次に示す、入力または出力を持たない `products` という名前の `Source` が存在し、その値は製品の一意の識別子であると想定します。

products の値
395
49780

`products` の値の位置を値として持つ、`productsPosition` という名前の新しい `Source` を作成するには、例 6-5 のコードを発行します。

例 6-5 入力または出力を持たない場合の値の位置の検索

```
Source productsPosition = products.position();
```

次の表に、`products` の値の位置を示す `productsPosition` を示します。`position()` メソッドは、1 から始まります。

products の値	値の位置
395	1
49780	2

出力および入力を持たない場合の値の位置の検索

次に示す、`products` の出力および `countries` の入力を持つ `unitsSoldByCountry` という名前の `Source` が存在し、その値は各国で販売された各製品の合計数であると想定します。

products (出力)	unitsSoldByCountry の値
395	500 800
49780	10000 50

値が `unitsSoldByCountry` の値の位置である、`positionUnitsSoldByCountry` という名前の新しい `Source` を作成するには、例 6-6 のコードを発行します。

例 6-6 出力および入力を持つ場合の値の位置の検索

```
Source positionUnitsSoldByCountry = unitsSoldByCountry.position();
```

次の表に、unitsSoldByCountry の値の位置を示す positionUnitsSoldbyCountry を示します。

products（出力）	positionUnitsSoldbyCountry の値
395	1
	2
49780	1
	2

昇順または降順でランキングされた値

最も単純なランキングの 1 つは、Source の値を昇順または降順でソートすることです。

例 6-7 では、base という名前の Source と同じ値が昇順でソートされている、sortedTuples という名前の新しい Source が作成されます。例 6-8 では、base という名前の Source の値が降順でランキングされます。

例 6-7 昇順での値のランキング

```
Source sortedTuples = base.sortAscending();
```

例 6-8 降順での値のランキング

```
Source sortedTuples = base.sortDescending();
```

他の Source の値と同じ順序または逆の順序でランキングされた値

Source の値を、他の Source の値と同じ順序または逆の順序でソートしてランキングできます。

例 6-9 に、base という名前の Source の値が sortValue という名前の Source と同じ順序でランキングされます。例 6-10 では、base という名前の Source の値が sortValue という名前の Source と逆の順序でランキングされます。

例 6-9 他の Source の値と同じ順序での値のランキング

```
Source sortedTuples = base.sortAscending(Source sortValue);
```

例 6-10 他の Source の値と逆の順序での値のランキング

```
Source sortedTuples = base.sortDescending(Source sortValue);
```

最低ランキング

最低ランキングは、同順（属性に対する 1 つの値を共有する Source 内の複数の値）の処理方法が一意ランキング（一意の位置）とは異なります。すべての同順は、同じランク（最低ランク）に指定されます。

例 6-11 に、最低ランキングを使用したサンプル・コードを示します。ここでは、値をランキングする base という名前の Source が、input1 および input2 という 2 つの入力を持つことを前提としています。

例 6-11 最低ランキング

```
Source sortedTuples = base.join(input1).sortDescending(input2);
Source equivalentRankedTuples =
    sortedTuples.join(input2, input2);
NumberSource minRank = sortedTuples.
    positionOfValues(equivalentRankedTuples).minimum();
```

最高ランキング

最高ランキングは、同順（属性に対する 1 つの値を共有する Source 内の複数の値）の処理方法が一意ランキング（一意の位置）とは異なります。すべての同順は、同じランク（最高ランク）に指定されます。

例 6-12 に、最高ランキングを使用したサンプル・コードを示します。ここでは、値をランキングする base という名前の Source が、input1 および input2 という 2 つの入力を持つことを前提としています。

例 6-12 最高ランキング

```
Source sortedTuples = base.join(input1).sortDescending(input2);
Source equivalentRankedTuples =
    sortedTuples.join(input2, input2);
NumberSource maxRank = sortedTuples.positionOfValues
    (equivalentRankedTuples).maximum();
```

平均ランキング

平均ランキングは、同順（属性に対する 1 つの値を共有する Source 内の複数の値）の処理方法が一意ランキングとは異なります。すべての同順は、同じランク（同順の値の一意の平均ランク）に指定されます。

例 6-13 に、平均ランキングを使用したサンプル・コードを示します。ここでは、値をランキングする base という名前の Source が、input1 および input2 という 2 つの入力を持つことを前提としています。

例 6-13 平均ランキング

```
Source sortedTuples = base.join(input1).sortDescending(input2);
Source equivalentRankedTuples =
    sortedTuples.join(input2, input2);
NumberSource averageRank = sortedTuples.positionOfValues
    (equivalentRankedTuples).average();
```

パック・ランキング

パック・ランキングは、稠密ランキングとも呼ばれ、ランクを連続した整数にまとめるという点で、最低ランキングと区別されます。

例 6-14 に、パック・ランキングを使用したサンプル・コードを示します。ここでは、値をランキングする `base` という名前の `Source` が、`input1` および `input2` という 2 つの入力を持つことを前提としています。

例 6-14 パック・ランキング

```
Source tuples = base.join(output1);
Source firstEquivalentTuple = tuples.join(input2, input2.first());
Source packedRank = firstEquivalentTuple.join(tuples).
    sortDescending(input2).positionOfValues(base.value()).
    join(time.value());
```

パーセンタイル・ランキング

次のフォーミュラを使用して、`N` 個の値を持つ `Source S` の属性 `A` のパーセンタイルを計算すると想定します。

$$\text{Percentile}(x) = \text{number of values} \\ \text{(for which the A differs from A}(x)\text{)} \\ \text{that come before x in the ordering} * 100 / N$$

算出されるパーセンタイルは、 $\text{minimum rank} - 1 * 100 / N$ と等しくなります。

例 6-15 に、パーセンタイル・ランキングを使用したサンプル・コードを示します。ここでは、値をランキングする `base` という名前の `Source` が、`input1` および `input2` という 2 つの入力を持つことを前提としています。

例 6-15 パーセンタイル・ランキング

```
Source sortedTuples = base.join(input1).sortDescending(input2);
Source equivalentRankedTuples =
    sortedTuples.join(input2, input2);
NumberSource minRank = sortedTuples.
    positionOfValues(equivalentRankedTuples).minimum();
NumberSource percentile = minRank.minus(1).times(100).
    div(sortedTuples.count());
```

nTile ランキング

任意の n の nTile ランキングは、順序付けされたサイズ件数の Source を n バケットに分割することによって定義されます。この場合、ランク k のバケットはサイズを示します。nTile ランクは、フォーミュラ $\text{ceiling}*((\text{uniqueRank}*n)/\text{count})$ と等しくなります。

例 6-16 に、nTile ランキングを使用したサンプル・コードを示します。ここでは、値をランキングする base という名前の Source が、input1 および input2 という 2 つの入力を持つことを前提としています。

例 6-16 nTile ランキング

```
NumberSource n = ...;
Source sortedTuples = base.join(input1).sortDescending(input2);
NumberSource uniqueRank = sortedTuple.
    positionOfValues(base.value().join(input1.value()));
NumberSource ntile = uniqueRank.times(n).
    div(sortedTuples.count()).ceiling();
```

階層での位置付けに基づいた値の選択

階層での位置付けに基づいて値を選択するには、階層をナビゲートする必要があります。階層内をナビゲートするには、2 つのプライマリ Source オブジェクト（階層に対応するプライマリ Source およびこの階層内の親子関係を表すプライマリ Source）を作成する必要があります。

デフォルトの階層を表すプライマリ Source の作成

デフォルトの階層を表す Source を作成するには、次の手順を実行します。

1. 次の手順を実行して、MdmDimension のデフォルトの階層を取得します。
 - a. MdmUnionDimensionDefinition が存在するかどうかを確認して、MdmDimension が共用体ディメンションであるかどうかを確認します。
 - b. MdmDimension に MdmUnionDimensionDefinition が存在する場合、MdmHierarchy オブジェクトであるリージョンが存在するかどうかを確認します。
 - c. MdmDimension に MdmHierarchy オブジェクトであるリージョンが存在する場合、デフォルトの階層である MdmHierarchy を選択します。
2. このデフォルトの階層の getSource メソッドをコールして、この階層を Source オブジェクトにします。

getMyDefaultHierarchy は、次に示すとおり、MdmDimension のデフォルトの階層を取得します。このメソッドは、MdmDimension のリージョンを取得する getMyRegions メソッドをコールします。次に、getMyRegions メソッドは、MdmDimension が共用体ディ

メンションかどうかを確認する `getMyMdmUnionDimensionDefinition` メソッドをコールします。

例 6-17 デフォルトの階層の取得

```
// method that gets all of the Regions of an MdmDimension
private MdmHierarchy getMyDefaultHierarchy(MdmDimension mdmDim) {
    List hierarchies = getMyRegions(mdmDim);
    if ( hierarchies == null )
        return null;
    for (Iterator iterator = hierarchies.iterator(); iterator.hasNext();) {
        MdmHierarchy hier = (MdmHierarchy) iterator.next();
        if (hier.hasMdmTag(MdmMetadataProvider.DEFAULT_HIERARCHY_TAG))
            return hier;
    }
    return null;
}

// method that gets all of the Regions of an MdmDimension
private List getMyRegions(MdmDimension mdmDimension) {
    MdmUnionDimensionDefinition unionDimDef =
        getMyMdmUnionDimensionDefinition ( mdmDimension );
    if ( unionDimDef != null )
        return unionDimDef.getMyRegions();
    return null;
}

// method that checks to see if MdmDimension is a UnionDimension
private MdmUnionDimensionDefinition getMyMdmUnionDimensionDefinition( MdmDimension
    mdmDimension ) {
    MdmDimensionDefinition dimDef = mdmDimension.getDefinition();
    if((dimDef == null) || (!(dimDef instanceof MdmUnionDimensionDefinition)))
        return null;
    return (MdmUnionDimensionDefinition) dimDef;
    return null;
}
```

親子関係を表すプライマリ Source の作成

`MdmHierarchy` がレベル階層である場合、その値は互いに親子関係にあります。階層内の親子関係を表す Source オブジェクトを作成するには、次の手順を実行します。

1. `MdmHierarchy` の `getParentRelation` メソッドを使用して、親子関係を表す `MdmAttribute` を作成します。
2. `getSource` メソッドを使用して、手順 1 で作成した `MdmAttribute` から Source を作成します。

他の関係を表す Source オブジェクトの作成

OLAP API の（親子関係などの）関係表現には、方向性があるという特性があります。親子関係を表す Source オブジェクトでは、子が親にマップされます。子に親はマップされません。これに対して、SQL の場合は、親子関係を表す表には方向性はありません。これは、OLAP API の場合、SQL とは異なり、Source オブジェクトの構造を使用して Source オブジェクトの join 実行方法が自動的に決定されるためです。OLAP API では、関係に方向性があるため、方向を逆にする場合は、関係を反転する必要があります。

levelHierarchy という名前の階層上に、parentChild という名前の Source が存在すると想定します。他の関係を表す Source オブジェクトを作成するには、この 2 つの Source オブジェクトを異なる方法で join します。すなわち、parentChild 関係を表す Source に join メソッドを使用して、階層内の子、兄弟および祖父母を表す新しい Source オブジェクトを作成できます（例 6-18、例 6-19 および例 6-20 を参照）。また、階層をドリルダウンすることもできます（6-13 ページの「階層のドリルダウン: 例」を参照）。

例 6-18 子の選択

```
Source childParent = levelHierarchy.join(parentChild,
    levelHierarchy.value());
```

例 6-19 兄弟の選択

```
Source siblingParent = levelHierarchy.join(parentChild, parent);
```

例 6-20 祖父母の選択

```
Source grandParent = parentChild.join(levelHierarchy, parentChild);
```

階層のドリルダウン: 例

MdmDimension オブジェクトが存在し、これに対して productsDim という名前の Source を作成したと想定します。また、MdmDimension オブジェクトにデフォルトの階層が存在し、これに対して prodStdHierObj という名前の MdmHierarchy および prodHeir という Source を作成したと想定します。例 6-21 では、この階層の「Trousers - Women」部がドリルダウンされます。

例 6-21 階層のドリルダウン

```
// Get the parent relation from the hierarchy
MdmAttribute prodHierParentObj = prodStdHierObj.getParentRelation();
StringSource prodHierParent = prodHierParentObj.getSource();
// Select children of Trousers - Women
// - Reverse the parent relation to get a children relation
Source prodHierChildren = prodHier.join(prodHierParent, prodHier.value());
// - Note the join is hidden because we only want the children of
// - Trousers - Women, and not Trousers - Women itself
```

```

Source trousersChildren = prodHierChildren.join(prodHier,
    context.getDataProvider().createConstantSource("Trousers - Women"), false);
// Select Shirts - Boys, Trousers - Women, and Shorts - Men
Source prodHierSel = prodHier.selectValues(new String[]
    {"Shirts - Boys", "Trousers - Women", "Shorts - Men"});
// Insert the children of Trousers - Women after Trousers - Women
// (which is 2nd value)
Source drilledProdHierSel = prodHierSel.appendValues(trousersChildren);
// This selection has the effect of sorting the result in hierarchical order.
Source result = prodHier.selectValues(drilledProdHierSel);

```

セルフ・リレーション Source の作成

地域間で比較を行うと想定します。具体的には、行と列の両方に地域名が表示されるデータのビューを作成すると想定します。OLAP API では、この作成に、`alias()` および `value()` メソッドを使用します。`alias()` メソッドを使用すると、データ、入力および出力に関して元の Source を正確にミラー化する新しい Source が作成されます。元の Source のタイプが別名 Source になるという点のみが異なります。`value()` メソッドを使用すると、元の Source をタイプおよび入力とする新しい Source が作成されます。

元の Source (base) の入力 A が、次の join で結合された Source の出力 B と一致すると想定します。

```
Source result = base.join(joined, comparison);
```

この入出力の一致を回避して、A を結果の入力としておくには、[例 6-22](#) のコードを使用します。

入出力を持たず、地域名が値である region という名前の Source が存在すると想定します。また、行と列の両方に地域名が表示されるデータのビューを作成すると想定します。この表の各セルに、地域間の面積（平方マイル単位）の差をパーセントで表示すると想定します。すなわち、region という名前の 2 つの Source を入力として持つ regionComparison という名前の Source を作成します。[例 6-23](#) では、この作成方法を示します。

例 6-22 セルフ・リレーションを作成するプロシージャ

```

//Create an alias Source named B2 for a Source named B;
Source B2 = B.alias();
//Create a variant of the original called base2
//We know that input A will match to B
Source base2 = base.join(B, B2.value());
//Now join base2 and joined
//We know that input B2 will not match to B in joined
Source preResult = base2.join(joined, comparison);
//Finally, join to the B2 and regain the input A
Source result = preResult.join(B2, A.value());

```


例 6-23 セルフ・リレーション Source の作成

```
//Create an alias for region that is for the row
Source rowRegion = region.alias();

//Create an alias for region that is for the column
Source columnRegion = region.alias();

//Create rowRegionArea which has an input of rowRegion,
//    an output of area,
//    and values whose values are the same as those of region
Source rowRegionArea = area.join(rowRegion.value());

//Create columnRegionArea which has an input of columnRegion,
//    an output of area,
//    and values whose values are the same as those of region
Source columnRegionArea = area.join(columnRegion.value());

//Compute the values of the cells
Source areaComparison = rowRegionArea.div(columnRegionArea).times(100);

//Create a new Source with outputs rather than inputs
Source regionComparison = areaComparison.join(rowRegion.join(columnRegion));
```

コードの最初の 2 行で、`region` という名前の Source の別名である新しい 2 つの Source オブジェクトが作成されます。これらの Source オブジェクトの名前は、`rowRegion` および `columnRegion` です。

次の 2 行では、`rowRegionArea` と `columnRegionArea` という名前の Source オブジェクトが作成されます。この 2 つは、それぞれ `rowRegion` と `columnRegion` の面積を表します。この例では、`rowRegionArea` を作成するために、`region` の入力を持つ `area` を、`rowRegion` の入力および `region` と同じ値を持つ `rowRegion.value()` に `join` しています。`rowRegionArea` Source は、`rowRegion` の入力、`area` の出力および `region` と同じ値を持ちます。`columnRegionArea` を作成するには、`region` の入力を持つ `area` を、`columnRegion` の入力および `region` と同じ値を持つ `columnRegion.value()` に `join` します。`columnRegionArea` という名前の Source は、`columnRegion` の入力、`area` の出力および `region` と同じ値を持ちます。これらの `join` コールは、`region` 入力と、`rowRegion` または `columnRegion` を置き換えます。この場合、両方ともデータとして地域名を持つため、`area` の値は事実上変更されません。

コードの次の行では、必要な計算が実行されます。`rowRegionArea` は、入力として `rowRegion` を持ち、`columnRegionArea` は面積として `columnRegion` を持つため、`areaComparison` という名前の新しい Source は、`rowRegion` および `columnRegion` という名前の 2 つの入力を持ちます。この両方の入力の値は、地域名です。これで、コピー入力を持つ Source オブジェクトが作成されました。

最後に、入力 (`rowRegion` および `columnRegion`) に `areaComparison` を `join` することによって、入力を出力に切り替えることができます。

数値分析の実行

NumberSource クラスおよびそのサブクラスは、様々な Source メソッドのうち、数値固有のメソッドを定義します。このメソッドを使用すると、数値を追加、挿入、選択および削除できます。また、NumberSource クラスおよびそのサブクラスのメソッドを使用して、減算や除算などの単純な数値操作、数値の比較、絶対値の計算などの標準の数値関数の操作、および値の合計による集計操作を実行できます。また、独自の関数を作成して、プログラム固有の数値分析を実行することもできます。

数値操作の実行

OLAP API を使用する場合、minus などの NumberSource メソッドを使用して、基本的な数値操作を実行します。各メソッドには様々なバージョンが存在し、これらを使用してリテラル値の double、float、int または short を指定できます。また、NumberSource を引数とするバージョンも、各メソッドに 1 つ存在します。

表 6-2 に、基本的な数値操作に使用する OLAP API のメソッドの概要を示します。

表 6-2 基本的な数値操作を実行する OLAP API のメソッド

メソッド	説明
div(rhs)	ベース NumberSource と同じ構造で、指定した値でベース NumberSource の値を除算した値を持つ新しい NumberSource を作成します。
intpart()	ベース NumberSource と同じ構造で、値がベース NumberSource の値の整数部分である新しい NumberSource を作成します。
minus(rhs)	ベース NumberSource と同じ構造で、ベース NumberSource の値から指定した値を減算した値を持つ新しい NumberSource を作成します。
negate()	ベース NumberSource と同じ構造で、ベース NumberSource の値の正負を反転した値を持つ新しい NumberSource を作成します。
plus(rhs)	ベース NumberSource と同じ構造で、ベース NumberSource の値に指定した値を加算した値を持つ新しい NumberSource を作成します。
rem(rhs)	ベース NumberSource と同じ構造で、指定した値でベース NumberSource の値を除算した際の余りを値として持つ新しい NumberSource を作成します。
times (rhs)	ベース NumberSource と同じ構造で、指定した値でベース NumberSource の値を乗算した値を持つ新しい NumberSource を作成します。

すべての値から同じ値の減算：例

次に示すとおり、出力が productsDim および timesDim の、unit_Cost という名前の NumberSource が存在し、その型が Integer であると想定します。

productsDim	timesDim	unit_Cost
Boys	1998	4000
Boys	31-DEC-01	10
49780	1998	500
49780	31-DEC-01	9

次の例では、unit_Cost の各値から売上上の 10% を減算して、示されている各製品からの調整済の収入を計算し、percentAdjustment という名前の新しい Source が作成されます。

例 6-24 すべての値から同じ値の減算

NumberSource percentAdjustment = unit_Cost.minus(unit_Cost.times(.10));

percentAdjustment という名前の新しい NumberSource の構造および値は次のとおりです。

productsDim	timesDim	percentAdjustment
Boys	1998	3600
Boys	31-DEC-01	9
49780	1998	450
49780
49780	31-DEC-01	8

ある NumberSource の値から別の NumberSource の値の減算：例

前述の例で説明した unitCost という名前の NumberSource と、次に示す unitManufacturingCost という名前の NumberSource が存在すると想定します。

productsDim	timesDim	unitCost の値
Boys	1998	600
Boys	31-DEC-01	3
49780	1998	250
49780	31-DEC-01	2

各製品の非製造コストを計算すると想定します。このコストを計算するには、総コストから製造コストを減算する必要があります。これを行うには、次のコードを使用して unit_Cost に対して操作を行い、nonManufacturingCost という名前の新しい Source を作成します。

例 6-25 ある NumberSource の値から別の NumberSource の値の減算

```
NumberSource nonManufacturingCost = unitCost.minus(unitManufacturingCost);
```

nonManufacturingCost の構造および値は次のとおりです。

productsDim	timesDim	値
Boys	1998	3400
Boys	31-DEC-01	7
49780	1998	250
49780	31-DEC-01	7

これらのメソッドの詳細は、OLAP API Javadoc を参照してください。

数値比較の実行

NumberSource クラスには、数値比較を実行する多数のメソッドが含まれています。これらのメソッドは、NumberSource の各値と指定した値を比較します。これらのメソッドは、元の NumberSource と同じ構造の BooleanSource を戻します。この BooleanSource の値は、元の NumberSource の任意の値に対する比較が真の場合は true、偽の場合は false になります。各メソッドには様々なバージョンが存在し、これらを使用してリテラル値の double、float、int または short を指定できます。

表 6-3 に、数値比較に使用する OLAP API のメソッドを示します。これらのメソッドの詳細は、OLAP API Javadoc を参照してください。

表 6-3 数値比較に使用するメソッド

メソッド	説明
eq	ベース NumberSource と同じ出力および入力で、指定した値と等しい NumberSource の各値に対しては true、指定した値と等しくない NumberSource の各値に対しては false の値を持つ新しい BooleanSource を作成します。
ge	ベース NumberSource と同じ出力および入力で、指定した値以上の NumberSource の各値に対しては true、指定した値未満の NumberSource の各値に対しては false の値を持つ新しい BooleanSource を作成します。
gt	ベース NumberSource と同じ出力および入力で、指定した値を超える NumberSource の各値に対しては true、指定した値以下の NumberSource の各値に対しては false の値を持つ新しい BooleanSource を作成します。
le	ベース NumberSource と同じ出力および入力で、指定した値以下の NumberSource の各値に対しては true、指定した値を超える NumberSource の各値に対しては false の値を持つ新しい BooleanSource を作成します。
lt	ベース NumberSource と同じ出力および入力で、指定した値未満の NumberSource の各値に対しては true、指定した値以上の NumberSource の各値に対しては false の値を持つ新しい BooleanSource を作成します。
ne	ベース NumberSource と同じ出力および入力で、指定した値と等しくない NumberSource の各値に対しては true、指定した値と等しい NumberSource の各値に対しては false の値を持つ新しい BooleanSource を作成します。

標準の数値関数の使用

OLAP API には、標準の数値関数を表す多くのメソッドが含まれています。表 6-4 に、これらのメソッドを示します。独自の関数を作成することもできます（6-23 ページの「[独自の数値関数の作成](#)」を参照）。

NumberSource に対してこれらの関数を使用すると、元の NumberSource と同じ構造で、元の NumberSource の値を関数に従って変更した値を持つ新しい NumberSource が戻されます。たとえば、abs() メソッドは、個々の値が元の NumberSource 内の対応する値の絶対値である新しい NumberSource を戻します。

表 6-4 標準の数値関数を表すメソッド

メソッド	説明
abs()	ベース NumberSource と同じ出力および入力で、ベース NumberSource の各値の絶対値を値として持つ新しい NumberSource を作成します。
arccos()	ベース NumberSource と同じ出力および入力で、NumberSource の各値（コサインとして解釈）の角度（ラジアン）値を値として持つ新しい NumberSource を作成します。
arcsin()	ベース NumberSource と同じ出力および入力で、NumberSource の各値（サインとして解釈）の角度（ラジアン）値を値として持つ新しい NumberSource を作成します。
arctan()	ベース NumberSource と同じ出力および入力で、NumberSource の各値（タンジェントとして解釈）の角度（ラジアン）値を値として持つ新しい NumberSource を作成します。
cos()	ベース NumberSource と同じ出力および入力で、NumberSource の各値（ラジアンで表された角度値として解釈）のコサインを値として持つ新しい NumberSource を作成します。
cosh()	ベース NumberSource と同じ出力および入力で、NumberSource の各値（ラジアンで表された角度値として解釈）の双曲線コサインを値として持つ新しい NumberSource を作成します。
log()	ベース NumberSource と同じ出力および入力で、NumberSource の各値の自然対数を値として持つ新しい NumberSource を作成します。
pow(rhs)	ベース NumberSource と同じ出力および入力で、NumberSource の各値を指定した値で累乗した値を持つ新しい NumberSource を作成します。
round(multiple)	ベース NumberSource と同じ出力および入力で、NumberSource の各値を指定した値に最も近い倍数に丸めた値を持つ新しい NumberSource を作成します。
sin()	ベース NumberSource と同じ出力および入力で、NumberSource の各値（角度として解釈）のサインを値として持つ新しい NumberSource を作成します。

集計メソッドの使用

stdev() などの標準の数値メソッドは、NumberSource の各値に対して機能します。集計メソッドとは、Source の一連の値を使用して計算を実行する total() のようなメソッドです。Oracle OLAP による集計関数の処理方法は、ベース NumberSource が入力を持つかどうかによって異なります。

- ベース NumberSource が入力を持たない場合、集計関数は、入力および出力を持たない新しい NumberSource を作成します。値は、ベース NumberSource 内のすべての値を使用して計算された 1 つの値です（6-21 ページの「Source が出力のみを持つ場合の合計の計算：例」の例を参照）。
- ベース NumberSource が入力を持つ場合、出力値の各セットは、値（タプル）のサブセットを識別します。この場合、集計メソッドは、データの各サブセットに対して機能します。集計関数は、ベース NumberSource と同じ出力を持たず、出力値の各セットに対して 1 つの値を持つ新しい NumberSource を作成します。これらの値は、出力値のそのセットによって識別される、ベース NumberSource 内のすべての値を使用して計算されます（6-21 ページの「Source が出力のみを持つ場合の合計の計算：例」の例を参照）。

表 6-5 に、数値の集計操作に使用する OLAP API のメソッドを示します。独自の集計関数を作成することもできます（6-23 ページの「独自の数値関数の作成」を参照）。

表 6-5 集計メソッド

メソッド	NumberSource が入力を持たない場合の説明
average	入力および出力を持たず、値が NumberSource の値の平均である新しい NumberSource を作成します。
maximum	入力および出力を持たず、値が NumberSource の最大値である新しい NumberSource を作成します。
minimum	入力および出力を持たず、値が NumberSource の最小値である新しい NumberSource を作成します。
total	入力および出力を持たず、値が NumberSource の値の合計である新しい NumberSource を作成します。

各数値集計メソッドには、2 つのバージョンが存在します。一方のバージョンでは、計算時にすべての null 値が排除されます。他方のバージョンでは、null 値を計算に含めるかどうかを指定できます。

OLAP API のメソッドが値の位置を判断する方法、および集計メソッドの値を計算するための値を判断する方法の詳細は、6-5 ページの「値の位置の検索」を参照してください。

Source が出力のみを持つ場合の合計の計算：例

2 つの出力（products および countries）を持つ unitsSoldByCountry という名前の Source が存在し、その値は各国で販売された各製品の合計数であると想定します。

products（出力 2）	countries（出力 1）	unitsSoldByCountry の値
395	Australia	1300
395	United States	800

products（出力 2）	countries（出力 1）	unitsSoldByCountry の値
49780	Australia	10050
49780	United States	50

これらの値を合計すると想定します。products と countries は両方とも出力であるため、次のコードを発行すると、新しい NumberSource は、すべての国で販売されたすべての製品の合計数を計算します。

例 6-26 Source が出力のみを持つ場合の合計値の計算

```
NumberSource totalUnitsSold = unitsSoldByCountry.total();
```

totalUnitsSold という名前の新しい NumberSource には、unitsSoldByCountry の値の合計値である単一の値のみが含まれます。

totalUnitsSold の値
11350

Source が出力および入力を持つ場合の合計の計算 : 例

countries の出力および products の入力を持つ unitsSoldByCountry という名前の Source が存在し、その値は各国で販売された各製品の合計数であると想定します。

countries（出力）	unitsSoldByCountry の値
Australia	1300
	10050
United States	50
	800

これらの値を合計すると想定します。入力が products であるため、次のコードを発行すると、新しい NumberSource は、各国で販売されたすべての製品の合計数を計算します。すべての国でのすべての製品の合計が計算されるわけではありません。

例 6-27 Source が出力および入力を持つ場合の合計値の計算

```
NumberSource totalUnitsSoldByCountry = unitsSoldByCountry.total();
```

totalUnitsSoldByCountry という名前の新しい NumberSource の出力は countries で、値は次のとおりです。

countries（出力）	unitsSoldByCountry の値
Australia	11350
United States	850

独自の数値関数の作成

alias メソッドを使用すると、パラメータを作成できます。alias メソッドを使用して新しい関数を作成する方法の詳細は、例 6-28「標準関数の作成」を参照してください。この方法では、セルまたは行の計算関数のみを作成できます。クライアント集計関数または位置ベースの関数を作成するには、extract メソッドを使用します。

独自の標準関数の作成：例

例 6-28 では、ある数字を取ってそれを 1.05 倍する関数が作成されます。この関数には、param という名前の 1 つのパラメータが存在します。このパラメータは、すべての数値のセットである OLAP API Number データ型を表す基本 Source の alias メソッドをコールすることによって作成されます。(value メソッドを使用して、パラメータを関数の入力にする方法に注意してください。) 例 6-28 で作成された関数は、事実上、OLAP API の組み込み関数と同じです。この関数は、パラメータおよび必要なパラメータ式に結合して使用できます。

unitsSold のメジャーがパラメータで指定する値より大きいすべての製品のセットになるように定義された、製品の選択を作成すると想定します。パラメータは、この Source からデータをフェッチする前に指定しておく必要があります。例 6-29 のコードを使用して、このパラメータを作成できます。パラメータ値を 100 に設定するには、例 6-30 のコードを使用します。その後、例 6-31 に示すとおり、sales という名前の Source に、例 6-28 で作成する関数を適用できます。

例 6-28 標準関数の作成

```
//Get the Source that represents the number data type
NumberSource number = (NumberSource)dataProvider
    .getFundamentalDefinitionProvider()
    .getNumberDataType()
    .getSource();
//Create a parameter
NumberSource param = (NumberSource)number.alias();
//Create a function
NumberSource function = ((NumberSource)param.value()).times(1.05);
```

例 6-29 パラメータ化された選択の作成

```
//Get the Source that represents the number data type
NumberSource number = dataProvider
```

```
.getFundamentalDefinitionProvider()
.getNumberDataType()
.getSource();
//Create a parameter
NumberSource param = (NumberSource)number.alias();
//Create a parameterized selection
Source products = ...;
NumberSource unitsSold = ...;
Source productSelection = products.select(unitsSold.gt(param.value()));
```

例 6-30 パラメータの値の設定

```
Source unitsSoldGT100 = productSelection.join(param, 100);
```

例 6-31 作成した標準関数の使用

```
//Use the function
NumberSource sales = ...;
NumberSource fsales = function.join(param, sales);
```

独自の集計関数の作成 : 例

重み付けされた平均関数を作成すると想定します。この平均関数を作成するには、[例 6-32 「重み付けされた平均関数の作成」](#) のコードを作成します。[例 6-28 「標準関数の作成」](#) の標準関数の例と同様に、このコードでは、関数が使用する `param` という名前のパラメータが最初に作成されます。ただし、ここでは集計関数を作成するため、コードは、`param` 付きの `extract()` メソッドを使用して最終結果を計算します。

[例 6-32](#) で作成される重み付けされた平均関数を使用するには、[例 6-33](#) に示すコードを発行します。

例 6-32 重み付けされた平均関数の作成

```
//Define an aggregation function
NumberSource weight = ...;
//Create a parameter
NumberSource param = (NumberSource) number.alias();
//Create a function
NumberSource weightedAverage = param.extract().times(weight).average();
```

例 6-33 [例 6-32](#) で作成される重み付けされた平均関数の使用

```
//Use the aggregation function
NumberSource sales = ...;
NumberSource paramSales = dp.createConstantSource(param.selectValues(sales));
Source weightedSales = weightedAverage.join(paramSales);
```

文字列値の操作

StringSource クラスは、様々な Source メソッドのうち、文字列固有のメソッドを定義します。このメソッドを使用すると、Java の String オブジェクトである値を追加、挿入、選択および削除できます。

StringSource クラスには、StringSource オブジェクトの値を操作するためのメソッドも含まれています (表 6-6 を参照)。また、OLAP API には、StringSource の値に含まれる部分文字列を操作するためのメソッドも含まれています (表 6-7 を参照)。

表 6-6 StringSource オブジェクトの値を操作するためのメソッド

メソッド	説明
length()	ベース StringSource と同じ構造で、値が StringSource の各値の長さである新しい NumberSource を作成します。
textFill(width)	ベース StringSource と同じ構造で、空白を追加することによってベース StringSource の各値を指定した幅に再フォーマットした値を持つ新しい StringSource を作成します。
toLowerCase()	ベース StringSource と同じ構造で、ベース StringSource の値のすべてのアルファベット文字を小文字にした値を持つ新しい StringSource を作成します。
toUpperCase()	ベース StringSource と同じ構造で、ベース StringSource の値のすべてのアルファベット文字を大文字にした値を持つ新しい StringSource を作成します。
trim()	ベース StringSource と同じ構造で、ベース StringSource の値の先頭および後続の空白を削除した値を持つ新しい StringSource を作成します。
trimLeading()	ベース StringSource と同じ構造で、ベース StringSource の値の先頭の空白を削除した値を持つ新しい StringSource を作成します。
trimTrailing	ベース StringSource と同じ構造で、ベース StringSource の値の後続の空白を削除した値を持つ新しい StringSource を作成します。

表 6-7 StringSource オブジェクトの部分文字列を操作するためのメソッド

メソッド	説明
indexOf (substring, fromIndex)	ベース StringSource と同じ構造で、指定した文字位置から始まる、ベース StringSource の値に含まれる指定した部分文字列を値として持つ新しい StringSource を作成します。
remove (index, length)	ベース StringSource と同じ構造で、ベース StringSource の値から指定した文字位置間の文字を削除した値を持つ新しい StringSource を作成します。
replace (oldString, newString)	ベース StringSource と同じ構造で、ベース StringSource の値に含まれる指定した部分文字列を別の部分文字列で置換した値を持つ新しい StringSource を作成します。
substring (index, length)	ベース StringSource と同じ構造で、ベース StringSource の値に含まれる指定した部分文字列を値として持つ新しい StringSource を作成します。

各メソッドには、2つのバージョンが存在します。一方のバージョンでは、Source オブジェクトを使用して値を指定し、他方のバージョンでは、リテラル値を使用して値を指定します。

TransactionProvider の使用

この章では、Oracle OLAP API の Transaction および TransactionProvider インタフェースについて説明します。また、アプリケーションにおいてこれらのインタフェースの実装を使用する方法についても説明します。DataProvider を作成する前に、TransactionProvider を作成しておく必要があります。また、導出 Source の Cursor を作成する前に、TransactionProvider のメソッドを使用して、Transaction を準備し、コミットしておく必要もあります。

この章では、次の項目について説明します。

- [Transaction 内での問合せの作成](#)
- [TransactionProvider オブジェクトの使用](#)

Transaction 内での問合せの作成

Oracle OLAP API は、トランザクション処理を行います。問合せ作成の各手順は、Transaction のコンテキストで実行されます。OLAP API アプリケーションの最初の段階で、TransactionProvider が作成されます。TransactionProvider によって、アプリケーションに Transaction オブジェクトが提供されます。

TransactionProvider によって、次のことが保証されます。

- Transaction が、その他の Transaction オブジェクトから分離されます。Transaction で実行中の操作は、別の Transaction オブジェクトでは参照できません（影響ありません）。
- Transaction での操作に失敗した場合、その結果は、取り消されます（その Transaction がロールバックされます）。
- 完了した Transaction の結果は存続します。

別の Source のメソッドをコールして導出 Source を作成する場合、その Source は、現行の Transaction のコンテキストに作成されます。Source は、それを作成した Transaction の中またはその Transaction の子 Transaction の中でアクティブです。

TransactionProvider のメソッドをコールして、現行の Transaction を取得または設定します。または、子 Transaction を開始します。子 Transaction では、親 Transaction で作成した Template の状態を変更できます。親 Transaction および子 Transaction で Template によって作成された Source が指定するデータを表示すると、アプリケーションのエンド・ユーザーに what-if 分析の実行手段を提供できます。

Transaction オブジェクトのタイプ

OLAP API には、次の 2 つのタイプの Transaction オブジェクトが存在します。

- 読み込み Transaction。現行の Transaction は、最初は読み込み Transaction です。読み込み Transaction は、Cursor を作成して Oracle OLAP からデータをフェッチする場合に必要です。Cursor オブジェクトの詳細は、[第 9 章](#)を参照してください。
- 書き込み Transaction。書き込み Transaction は、導出 Source の作成または Template の状態の変更を行う場合に必要です。導出 Source を作成する方法の詳細は、[第 5 章](#)を参照してください。Template オブジェクトの詳細は、[第 10 章](#)を参照してください。

導出 Source を作成した場合、または Template オブジェクトの状態を変更した場合、最初の読み込み Transaction に、子の書き込み Transaction が自動的に生成されます。この子 Transaction が、現行の Transaction になります。

その後、導出 Source をもう 1 つ作成するか、または Template の状態を再度変更した場合、その操作は同じ書き込み Transaction で実行されます。同じ書き込み Transaction で、導出 Source オブジェクトをいくつでも作成でき、Template 状態も何回でも変更できま

す。これらの Source オブジェクト、または Template によって作成された Source を使用すると、複雑な問合せを定義できます。

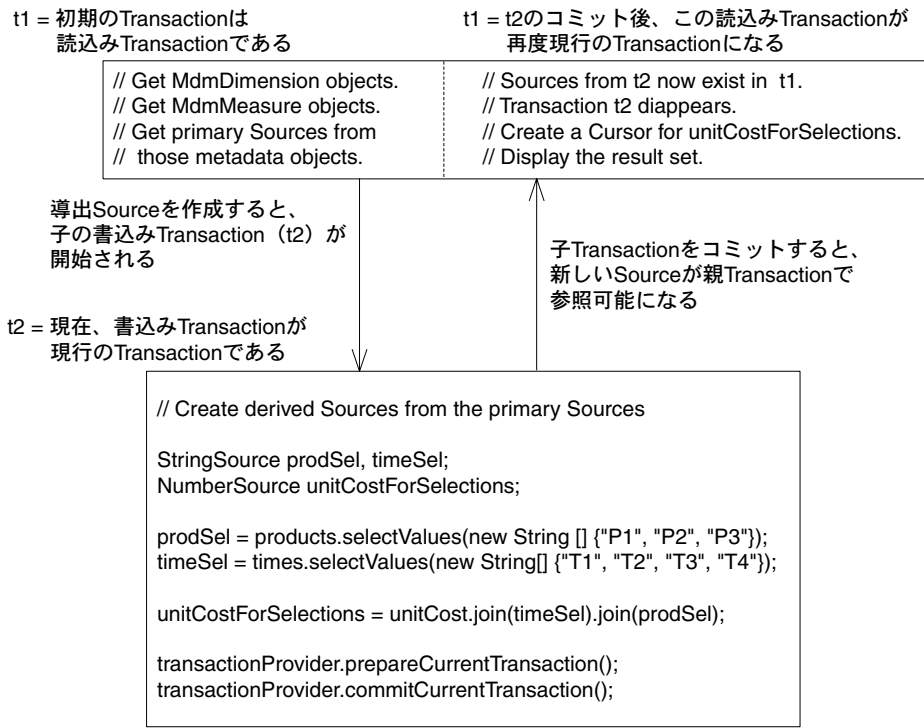
Cursor を作成して、導出 Source によって指定される結果セットをフェッチする前に、子の書込み Transaction から親の読込み Transaction に Source を移動する必要があります。この移動を行うには、Transaction を準備し、コミットする必要があります。

Transaction の準備およびコミット

子 Transaction で作成した Source を親の読込み Transaction に移動するには、TransactionProvider の `prepareCurrentTransaction` および `commitCurrentTransaction` メソッドをコールします。子の書込み Transaction をコミットした場合、子 Transaction で作成した Source は、親の読込み Transaction に移動します。子 Transaction は削除され、親 Transaction が現行の Transaction になります。Source は、現行の読込み Transaction でアクティブであるため、その Source の Cursor を作成できます。

次の図に、子の書込み Transaction で作成された Source を、その親の読込み Transaction に移動するプロセスを示します。

図 7-1 書き込み Transaction から親の読み込み Transaction へのコミット



Transaction および Template オブジェクト

現行の Transaction の取得および設定、子 Transaction の開始および Transaction のロールバックを実行すると、エンド・ユーザーは、動的問合せの任意の状態から様々な選択を実行できます。このように最初の状態に基づいて代替を作成することを、**what-if** 分析といいます。

エンド・ユーザーに同じ最初の間合せに基づいた代替を提供するには、次の手順を実行します。

1. 親 Transaction に Template を作成し、Template の初期状態を設定します。
2. Template によって作成された Source を取得します。Cursor を作成して結果セットを検索し、Cursor から値を取得します。その後、エンド・ユーザーに結果を表示します。
3. 子 Transaction を開始して、Template の状態を変更します。

4. 子 Transaction で Template によって作成された Source を取得します。Cursor を作成して、取得した値を表示します。

その後、最初の Template の状態を 2 番目のものと置換できます。2 番目の状態を廃棄して、1 番目のものを保持することもできます。

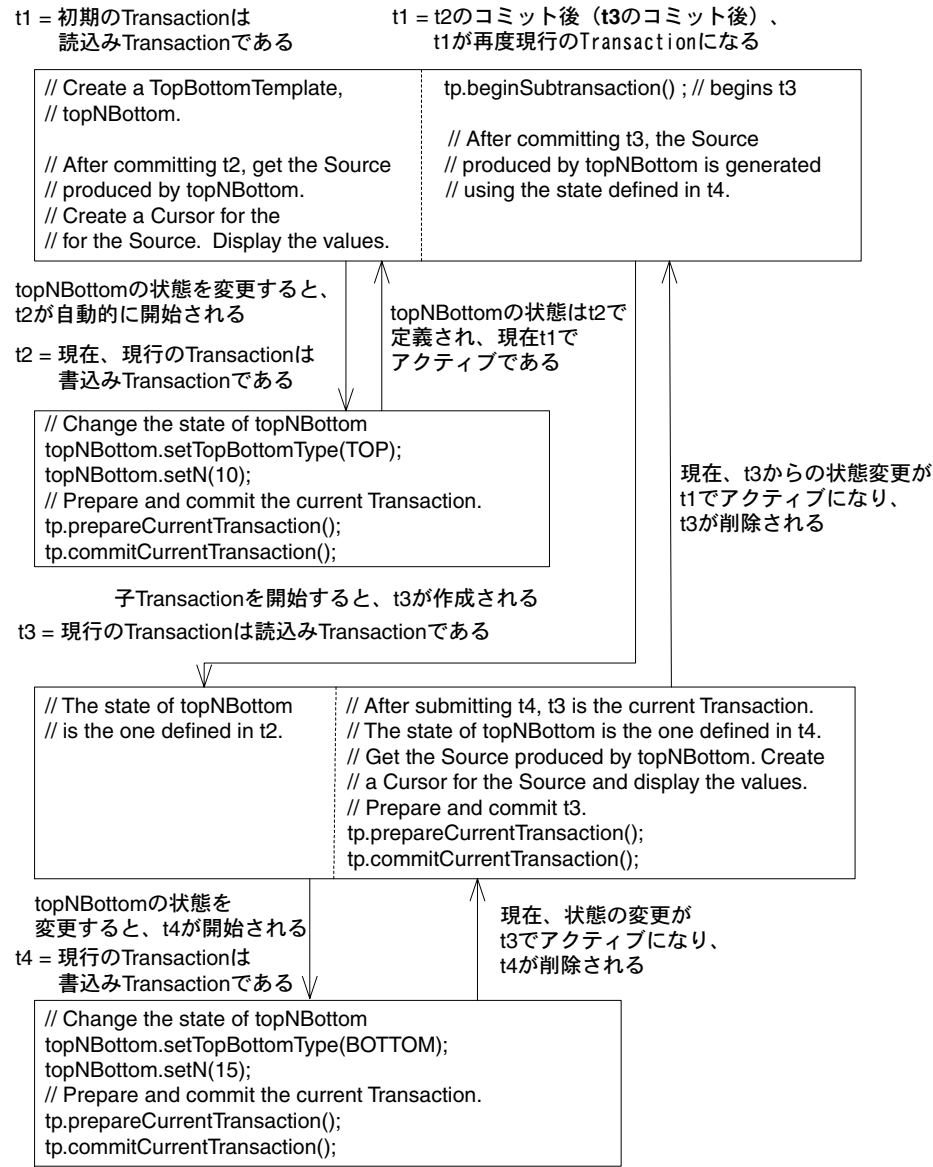
子 Transaction の開始

子の読み込み Transaction を開始するには、使用中の TransactionProvider の `beginSubtransaction` メソッドをコールします。その後、Template の状態を変更すると、子の書き込み Transaction が自動的に開始されます。書き込み Transaction は、子の読み込み Transaction の子です。

Template によって作成された Source が指定するデータを取得するには、親の読み込み Transaction への書き込み Transaction を準備してコミットします。その後、Cursor を作成してデータをフェッチできます。Template の変更後の状態は、元の親では参照できません。親の読み込み Transaction への子の読み込み Transaction を準備してコミットするまでは、変更後の状態は、親では参照できません。

次の図に、子の読み込み Transaction の開始、書き込み Transaction での Source オブジェクトの作成および親の読み込み Transaction への書き込み Transaction のコミットを示します。また、親の読み込み Transaction への子の読み込み Transaction のコミットを示します。次の図の `tp` は、TransactionProvider を示します。

図 7-2 親 Transaction への子の読み込み Transaction のコミット



子の読み込み Transaction を開始した後、その子の子の読み込み Transaction（最初の親 Transaction の孫）を開始できます。子および孫 Transaction オブジェクトの作成方法の例については、[例 7-2](#) を参照してください。

Transaction のロールバック

使用中の TransactionProvider の rollbackCurrentTransaction メソッドをコールすることによって、Transaction をロールバック（取消し）できます。Transaction をロールバックすることによって、Transaction 中に実行されたすべての変更が廃棄され、Transaction が削除されます。

Transaction をロールバックする前に、その Transaction で作成したすべての CursorManager オブジェクトをクローズする必要があります。Transaction をロールバックした後、作成したすべての Source オブジェクト、または Transaction で実行した Template 状態の変更が無効になります。その Source オブジェクト用に作成したすべての Cursor も無効になります。

Transaction をロールバックすると、その Transaction を準備およびコミットすることはできません。同様に、Transaction をコミットした後は、それをロールバックすることはできません。

例 7-1 Transaction のロールバック

次の例では、TopBottomTemplate が作成され、その状態が設定されます。この例では、子 Transaction が開始されて、TopBottomTemplate が異なる状態に設定されます。その後、子 Transaction がロールバックされます。TransactionProvider は tp です。

```
// The current Transaction is a read Transaction, t1.
// Create a TopBottomTemplate using product as the base
// and dp as the DataProvider.
TopBottomTemplate topNBottom = new TopBottomTemplate(product, dp);

// Changing the state of a Template requires a write Transaction, so a
// write child Transaction, t2, is automatically started.
topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_TOP);
topNBottom.setN(10);
topNBottom.setCriterion(singleSelections.getSource());

// Prepare and commit the Transaction t2.
tp.prepareCurrentTransaction();
tp.commitCurrentTransaction();           //t2 disappears

// The current Transaction is now t1.
// Create a Cursor and display the results (operations not shown).

// Start a child Transaction, t3. It is a read Transaction.
tp.beginSubtransaction();                // t3 is the current Transaction
```

```
// Change the state of topNBottom. Changing the state requires a
// write Transaction so Transaction t4 starts automatically,
topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_BOTTOM);
topNBottom.setN(15);

// Prepare and commit the Transaction.
tp.prepareCurrentTransaction();
tp.commitCurrentTransaction();           // t4 disappears

// Create a Cursor and display the results. // t3 is the current Transaction
// Close the CursorManager for the Cursor created in t3.
// Undo t3, which discards the state of topNBottom that was set in t4.
tp.rollbackCurrentTransaction()         // t3 disappears

// Transaction t1 is now the current Transaction and the state of
// topNBottom is the one defined in t2.
```

現行の Transaction の取得および設定

次の例のように、使用中の TransactionProvider の `getCurrentTransaction` メソッドをコールすることによって、現行の Transaction を取得します。

```
Transaction t1 = getCurrentTransaction();
```

以前に保存した Transaction を現行の Transaction にするには、次の例のように、TransactionProvider の `setCurrentTransaction` メソッドをコールします。

```
setCurrentTransaction(t1);
```

TransactionProvider オブジェクトの使用

Oracle OLAP API では、TransactionProvider インタフェースは、具象クラスの `ExpressTransactionProvider` によって実装されます。DataProvider を作成する前に、ExpressTransactionProvider の新しいインスタンスを作成する必要があります。その後、TransactionProvider を DataProvider コンストラクタに渡します。TransactionProvider によって、アプリケーションに Transaction オブジェクトが提供されます。

7-3 ページの「[Transaction の準備およびコミット](#)」に示すとおり、親の読み込み Transaction において、子の書き込み Transaction で作成した導出 Source を参照できるようにするには、`prepareCurrentTransaction` および `commitCurrentTransaction` メソッドを使用します。その後、その Source 用の Cursor を作成できます。

アプリケーションで Template オブジェクトを使用している場合は、TransactionProvider の他のメソッドを使用して次の操作を行うこともできます。

- 子 Transaction を開始します。
- 保存するために、現行の Transaction を取得します。
- 現行の Transaction を以前に保存したものに設定します。
- 現行の Transaction をロールバック（取消し）します。これによって、Transaction で行われたすべての変更が廃棄されます。Transaction は、ロールバックされた後は無効となりコミットできません。Transaction は、コミットされた後はロールバックできません。Transaction で Source の Cursor を作成した場合、Transaction をロールバックする前に CursorManager をクローズする必要があります。

例 7-2 に、動的問合せの変更に Transaction オブジェクトを使用する方法を示します。この例は、第 10 章で定義している TopBottomTest アプリケーションに基づきます。例では、Transaction オブジェクトを追跡できるように、getCurrentTransaction メソッドをコールすることによって別の Transaction オブジェクトが保存されています。

TopBottomTest クラスのコードの最後の 5 行を例 7-2 のコードと置換してください。

例 7-2 子 Transaction オブジェクトの使用

```
// The parent Transaction is the current Transaction at this point.
// Save the parent read Transaction as parentT1.
Transaction parentT1 = tp.getCurrentTransaction();

// Begin a child Transaction of parentT1.
tp.beginSubtransaction(); // This is a read Transaction.

// Save the child read Transaction as childT2.
Transaction childT2 = tp.getCurrentTransaction();

// Change the state of the TopBottomTemplate. This starts a
// write Transaction, a child of the read Transaction childT2.
topNBottom.setN(15);
topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_BOTTOM);

// Save the child write Transaction as writeT3.
Transaction writeT3 = tp.getCurrentTransaction();

// Prepare and commit the write Transaction writeT3.
try{
    context.getTransactionProvider().prepareCurrentTransaction();
}
catch (NotCommittableException e){
```

```
        System.out.println("Caught exception " + e + ".");
    }
    context.getTransactionProvider().commitCurrentTransaction();

    // The commit moves the changes made in writeT3 into its parent,
    // the read Transaction childT2. The writeT3 Transaction
    // disappears. The current Transaction is now childT2
    // again but the state of the TopBottomTemplate has changed.

    // Create a Cursor and display the results of the changes to the
    // TopBottomTemplate that are visible in childT2.
    createCursor(topNBottom.getSource());

    // Begin a grandchild Transaction of the initial parent.
    tp.beginSubtransaction(); // This is a read Transaction.

    // Save the grandchild read Transaction as grandchildT4.
    Transaction grandchildT4 = tp.getCurrentTransaction();

    // Change the state of the TopBottomTemplate. This starts another
    // write Transaction, a child of grandchildT4.
    topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_TOP);

    // Save the write Transaction as writeT5.
    Transaction writeT5 = tp.getCurrentTransaction();

    // Prepare and commit writeT5.
    try{
        context.getTransactionProvider().prepareCurrentTransaction();
    }
    catch(NotCommittableException e){
        System.out.println("Caught exception " + e + ".");
    }
    context.getTransactionProvider().commitCurrentTransaction();

    // Transaction grandchildT4 is now the current Transaction and the
    // changes made to the TopBottomTemplate state are visible.

    // Create a Cursor and display the results visible in grandchildT4.
    createCursor(topNBottom.getSource());

    // Commit the grandchild into the child.
    try{
        context.getTransactionProvider().prepareCurrentTransaction();
    }
    catch(NotCommittableException e){
        System.out.println("Caught exception " + e + ".");
    }
```

```
}
context.getTransactionProvider().commitCurrentTransaction();

// Transaction childT2 is now the current Transaction.
// Instead of preparing and committing the grandchild Transaction,
// you could rollback the Transaction, as in the following
// method call:
// rollbackCurrentTransaction();
// If you roll back the grandchild Transaction, then the changes
// you made to the TopBottomTemplate state in the grandchild
// are discarded and childT2 is the current Transaction.

// Commit the child into the parent.
try{
    context.getTransactionProvider().prepareCurrentTransaction();
}
catch(NotCommittableException e){
    System.out.println("Caught exception " + e + ".");
}
context.getTransactionProvider().commitCurrentTransaction();

// Transaction parentT1 is now the current Transaction. Again,
// you could roll back the childT2 Transaction instead of
// preparing and committing it. If you did so, then the changes
// you made in childT2 are discarded. The current Transaction
// would be parentT1, which would have the original state of
// the TopBottomTemplate, without any of the changes made in
// the grandchild or the child transactions.

} // end of main() method
} // end of TopBottomTest class
```

Cursor クラスおよび Cursor の概念

この章では、問合せの結果を取得してそれにアクセスするために使用する、Oracle OLAP API の `Cursor` クラスおよびその関連クラスについて説明します。また、`Cursor` の位置、フェッチ・サイズおよびエクステンツの概念についても説明します。`Cursor` およびその関連オブジェクトの作成と使用の例については、[第 9 章](#)を参照してください。

この章では、次の項目について説明します。

- [OLAP API の `Cursor` オブジェクトの概要](#)
- [Cursor の位置およびエクステンツ](#)
- [フェッチ・サイズおよびフェッチ・ブロック](#)

OLAP API の Cursor オブジェクトの概要

Cursor を使用すると、Source によって定義される結果セットを取得できます。Source 用の Cursor を作成する場合、複数の中間手順を実行する必要があります。データ・ストアから取得するデータを定義した Source を作成した後に、次の手順に従って、その Source 用の Cursor を作成します。

1. 使用中の `DataProvider` の `createCursorManagerSpecification` メソッドに Source を渡すことによって、`CursorManagerSpecification` を作成します。`CursorManagerSpecification` は、Source の構造をミラー化した構造で、`CursorSpecification` オブジェクトを含んでいます。
2. `DataProvider` の `createCursorManager` メソッドをコールして、それに `CursorManagerSpecification` を渡すことによって、`CursorManager` を作成します。`CursorManager` によって、Cursor オブジェクトが作成され、その Cursor オブジェクト用のローカル・データ・キャッシュも管理されます。動的問合せのための、Source に対する変更も認識されます。`CursorManagerSpecification` の Source が入力を持つ場合、それらの入力用の Source オブジェクトの配列を `createCursorManager` メソッドに渡す必要もあります。
3. `CursorManager` の `createCursor` メソッドをコールすることによって、Cursor を作成します。Cursor の構造には、`CursorManagerSpecification` および Source の構造がミラー化されます。`CursorManagerSpecification` の `CursorSpecification` オブジェクトによって、対応する Cursor オブジェクトの動作が指定されます。`CursorManagerSpecification` の Source が入力を持つ場合、その入力 Source オブジェクトの値を指定する `CursorInput` オブジェクトの配列を `createCursor` メソッドに渡す必要もあります。

Cursor の作成例については、[第9章](#)を参照してください。

このアーキテクチャによって、結果セットからデータをフェッチしたり、表示するデータを選択するときに、柔軟性が大幅に向上します。次の操作を実行できます。

- 1つの Source に対して複数の `CursorManagerSpecification` オブジェクトを作成できます。1つの結果セットから様々な値の集合を取得したり表示するために、様々な `CursorManagerSpecification` オブジェクトの `CursorSpecification` コンポーネントに異なる動作を指定できます。Source からのデータを異なる形式（表およびクロス集計など）で表示するときに、この操作を実行する場合があります。
- Template によって作成された Source が変更されたという通知を受け取ることができます。`CursorManagerUpdateListener` を Source の `CursorManager` に追加すると、動的問合せの Source が変更されたために `CursorManager` の `CursorManagerSpecification` を更新する必要がある場合に、`CursorManager` から `CursorManagerUpdateListener` に通知されます。
- `CursorManager` の `CursorManagerSpecification` を更新することができます。Template オブジェクトを使用して動的問合せを生成しているときに Template の状態が変更された場合、Template によって生成された Source が変更されます。Template によって生成された Source 用の Cursor を作成していた場合、変更された

Source 用に更新された `CursorManagerSpecification` で、`CursorManager` の `CursorManagerSpecification` を置き換える必要があります。その後、`CursorManager` から新しい `Cursor` を作成できます。

- 同じ `CursorManager` から異なる `Cursor` オブジェクトを作成し、それらの `Cursor` オブジェクトに異なるフェッチ・サイズを設定することができます。これは、同じデータを表およびグラフとして表示する必要がある場合に行います。

Cursor を作成できない Source

Source オブジェクトには、`Cursor` がデータ・ストアから取得できるデータが指定されないものがあります。`Cursor` を作成できない Source オブジェクトは次のとおりです。

- 計算不可能な操作を指定する Source。たとえば、無限再帰を指定する Source です。
- 無限結果セットを定義する Source。たとえば、すべての `String` オブジェクトのセットを表す基本 Source です。
- 要素を持たない Source か、または要素を持たない他の Source を含む Source。たとえば、`DataProvider` の `getEmptySource` メソッドによって戻された Source や、空の Source から導出された別の Source です。`MdmDimension` から取得したプライマリ Source から値を選択することによって作成された導出 Source で、選択された値がディメンションに存在しない Source も該当します。

Cursor オブジェクトおよび Transaction オブジェクト

導出 Source を作成するか、または `Template` の状態を変更する場合、Source は、現在の `Transaction` のコンテキストに作成されます。Source は、それを作成した `Transaction` の中またはその `Transaction` の子 `Transaction` の中でアクティブです。Source の `Cursor` を作成するには、その Source が現行の `Transaction` でアクティブである必要があります。

導出 Source の作成は、書込み `Transaction` で実行されます。`Cursor` の作成は、読み込み `Transaction` で実行されます。導出 Source の作成後、その Source 用の `Cursor` を作成するには、アプリケーションで使用されている `TransactionProvider` の `prepareCurrentTransaction` および `commitCurrentTransaction` メソッドをコールして、書込み `Transaction` を読み込み `Transaction` に変更する必要があります。`Transaction` オブジェクトおよび `TransactionProvider` オブジェクトの詳細は、[第7章](#)を参照してください。

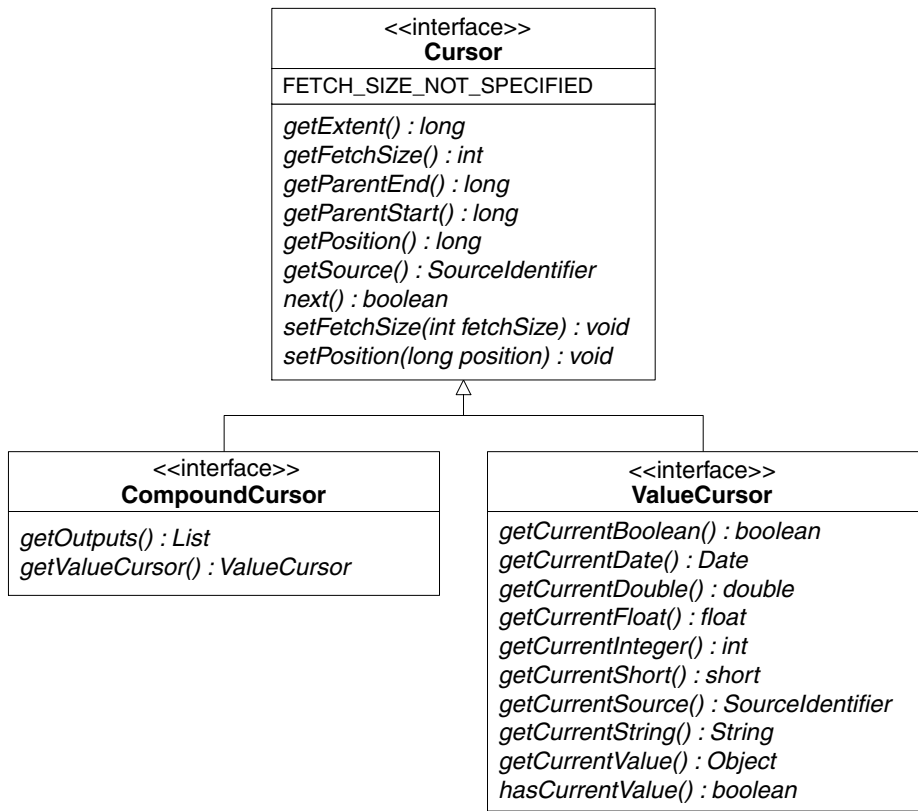
Cursor クラス

`oracle.olapi.data.cursor` パッケージでは、次の表に示すインタフェースが Oracle OLAP API によって定義されます。

インタフェース	説明
Cursor	現在の位置の概念をカプセル化する抽象スーパークラス。
ValueCursor	現在の位置に 1 つの値を持つ Cursor。ValueCursor には、子 Cursor オブジェクトは存在しません。
CompoundCursor	子 Cursor オブジェクトを持つ Cursor。含まれる Cursor オブジェクトは、その Source の値に対する子 ValueCursor と、その Source の各出力用の出力子 Cursor です。

図 8-1 に、Cursor クラスのクラス階層を示します。CompoundCursor インタフェースおよび ValueCursor インタフェースは、Cursor インタフェースを拡張します。

図 8-1 Cursor の階層



Cursor の構造

Cursor の構造には、Source の構造がミラー化されます。Source が出力を持たない場合、その Source 用の Cursor は ValueCursor です。Source が 1 つ以上の出力を持つ場合、その Source 用の Cursor は CompoundCursor です。CompoundCursor には、子として、その CompoundCursor の Source のベースの値を持つベース ValueCursor、および 1 つ以上の出力 Cursor オブジェクトが存在します。

Source の出力は、別の Source です。出力 Source 自体も出力を持つことができます。Source の出力用の子 Cursor は、出力 Source が出力を持たない場合は ValueCursor、出力を持つ場合は CompoundCursor です。

たとえば、製品のディメンションの値を表すプライマリ Source から選択した、製品の識別値を示す productSel という導出 Source を作成したと想定します。productSel の値に 815、1050 および 2055 を選択したと想定します。productSel 用の Cursor を作成した場合、productSel は出力を持たないため、その Cursor は ValueCursor になります。

また、時間値のディメンションを表すプライマリ Source から選択した日付値を示す、timeSel という導出 Source も作成したと想定します。timeSel の値は、1-JAN-00、1-APR-00、1-JUL-00 および 1-OCT-00 です。

製品の単価の値を表す MdmMeasure が存在すると想定します。MdmMeasure には、入力として、製品および時間を表す MdmDimension オブジェクトが存在します。このメジャーから、unitPrice という Source を取得します。この Source には、入力として、製品および時間が存在します。

次のように指定して、productSel および timeSel を unitPrice に結合し、productSel および timeSel を出力として持つ unitPriceByDay という Source を作成します。

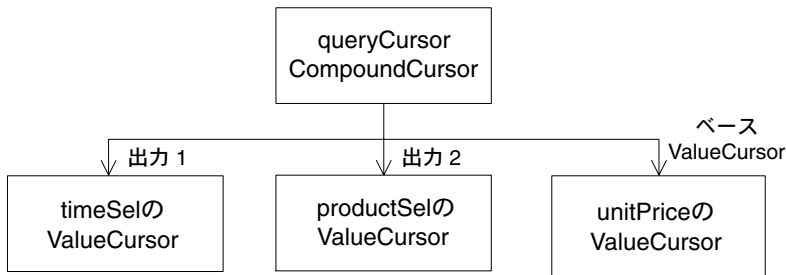
```
unitPriceByDay = unitPrice.join(productSel).join(timeSel);
```

unitPriceByDay によって定義された結果セットは、出力によって編成された単価の値です。timeSel が unitPrice.join(productSel) の結果に結合されているため、timeSel は遅く変化する出力です。これは、結果セットによって、選択された各時間値に対して選択された一連の製品が指定されることを意味します。各時間値に対して、結果セットは 3 つの製品値を持つため、製品値は時間値より速く変化します。unitPriceByDay のベース ValueCursor の値は、各日付の各製品に対して 1 つの値があるため、すべての値のうちで最も速く変化します。

その後、unitPriceByDay 用の Cursor (queryCursor) を作成します。unitPriceByDay は出力を持つため、queryCursor は CompoundCursor です。queryCursor のベース ValueCursor は unitPrice からの値を持ちます。これは、unitPriceByDay を作成した操作のベース Source です。queryCursor の出力は、productSel からの値を持つ ValueCursor と、timeSel からの値を持つ ValueCursor です。

図 8-2 に、queryCursor の構造を示します。ベース ValueCursor および 2 つの出力 ValueCursor オブジェクトは、親 CompoundCursor である queryCursor の子です。

図 8-2 CompoundCursor である queryCursor の構造



次の表は、queryCursor の値を表で示したものです。左の列には時間値、中央の列には製品値、右の列には指定した日における指定した製品の単価を示します。

日付	製品	単価
01-JAN-00	815	58
01-JAN-00	1050	24
01-JAN-00	2055	24
01-APR-00	815	59
01-APR-00	1050	24
01-APR-00	2055	25
01-JUL-00	815	59
01-JUL-00	1050	25
01-JUL-00	2055	25
01-OCT-00	815	61
01-OCT-00	1050	25
01-OCT-00	2055	26

ValueCursor から値を取得する例については、第 9 章を参照してください。

Cursor の動作の指定

CursorManagerSpecification の CursorSpecification オブジェクトによって、対応する Cursor オブジェクトの動作のいくつかの側面が指定されます。対応する Cursor オブジェクトを作成する前に、その動作を CursorSpecification に指定する必要があります。動作を指定するには、次の CursorSpecification メソッドを使用します。

- `setDefaultFetchSize`
- `setExtentCalculationSpecified`
- `setParentEndCalculationSpecified`
- `setParentStartCalculationSpecified`
- `specifyDefaultFetchSizeOnChildren`
(CompoundCursorSpecification の場合のみ)

CursorSpecification には、動作が指定されているかどうかを確認するためのメソッドも含まれています。それらのメソッドを次に示します。

- `isExtentCalculationSpecified`
- `isParentEndCalculationSpecified`
- `isParentStartCalculationSpecified`

CursorSpecification メソッドを使用して、デフォルトのフェッチ・サイズの設定、エクステンツの計算、あるいは親の中での値の開始位置または終了位置の計算を実行した場合、次の Cursor メソッドを正常に実行できます。

- `getExtent`
- `getFetchSize`
- `getParentEnd`
- `getParentStart`
- `setFetchSize`

Cursor の動作を指定する例については、[第 9 章](#)を参照してください。フェッチ・サイズの詳細は、[8-23 ページの「フェッチ・サイズおよびフェッチ・ブロック」](#)を参照してください。Cursor のエクステンツの詳細は、[8-22 ページの「Cursor のエクステンツの概要」](#)を参照してください。親 Cursor の中での、Cursor の現在の値の開始位置および終了位置の詳細は、[8-19 ページの「Cursor 内での親の開始 / 終了位置」](#)を参照してください。

CursorManagerSpecification クラス

Source の CursorManagerSpecification には、1 つ以上の CursorSpecification オブジェクトが存在します。これらのオブジェクトの構造には、その Source の構造が反映されます。たとえば、出力を持つ Source には、その Source の最上位（ルート）の

CursorSpecification、その Source の値用の、子 CursorSpecification、およびその Source の各出力用の、子 CursorSpecification が存在します。

出力を持たない Source には、1 セットの値のみが存在します。したがって、その Source の CursorManagerSpecification は、1 つの CursorSpecification のみを持ちます。その CursorSpecification は、CursorManagerSpecification のルート・レベルの CursorSpecification です。

1 つ以上の入力を持つ多次元 Source 用の CursorManagerSpecification を作成できます。これを作成する場合、CursorManagerSpecification の CursorManager を作成するときに、各入力に Source を指定する必要があります。また、CursorManager から Cursor を作成するときに、各入力 Source に CursorInput を指定する必要があります。CursorManager を使用して一連の Cursor オブジェクトを作成し、各 Cursor が、入力 Source オブジェクトの 1 つの値の様々なセットによって指定されるデータを取得するようにする場合、入力を持つ Source の CursorManagerSpecification を作成します。

Cursor の構造には、その CursorManagerSpecification の構造が反映されます。Cursor は、出力を持たない Source の場合は 1 つの ValueCursor、出力を持つ Source の場合は子 Cursor オブジェクトを持つ CompoundCursor になります。各 Cursor は、CursorManagerSpecification 内の CursorSpecification に対応しています。対応する Cursor の動作の側面を指定するには、CursorSpecification メソッドを使用します。

アプリケーションで Template オブジェクトが使用されており、Template の状態の変更に従って Template によって生成された Source の構造が変更された場合、アプリケーションによってその Source 用に生成されたすべての CursorManagerSpecification オブジェクトが期限切れになります。CursorManagerSpecification が期限切れになった場合、新しい CursorManagerSpecification を作成する必要があります。その後、CursorManager の以前の CursorManagerSpecification を新しい CursorManagerSpecification に置き換えるか、または新しい CursorManagerSpecification を使用して新しい CursorManager を作成します。CursorManagerSpecification が期限切れになっているかどうかは、CursorManagerSpecification の isExpired メソッドをコールして確認できます。

CursorSpecification クラス

CursorSpecification では、対応する Cursor の動作の特定の側面を指定できます。CursorSpecification は、直接作成しません。Source を DataProvider の createCursorManagerSpecification に渡すと、その Source のルート・レベルの CursorSpecification を含んだ CursorManagerSpecification が戻されます。Source が出力を持つ場合、CursorManagerSpecification には、その Source の値用およびその Source の各出力用の、子 CursorSpecification が含まれます。

CursorSpecification メソッドを使用すると、次の操作を実行できます。

- CursorSpecification に対応する Source を取得できます。
- 対応する Cursor のデフォルトのフェッチ・サイズを取得または設定できます。

- CompoundCursorSpecification に、デフォルトのフェッチ・サイズが、対応する Cursor の子に設定されたことを指定できます。
- Oracle OLAP で Cursor のエクステントを計算する必要があることを指定できます。
- エクステントの計算が指定されているかどうかを判断できます。
- 対応する Cursor の、親 Cursor の中での現在の値の開始位置または終了位置を、Oracle OLAP で計算する必要があることを指定できます。親の中での値の開始位置および終了位置がわかっている場合、親 Cursor がその値に対して持っている、速く変化する要素の数を判断できます。
- 対応する Cursor の、親の中での現在の値の開始位置または終了位置の計算が指定されているかどうかを判断できます。
- CursorSpecificationVisitor を受け入れることができます。

詳細は、8-13 ページの「[Cursor の位置およびエクステント](#)」および 8-23 ページの「[フェッチ・サイズおよびフェッチ・ブロック](#)」を参照してください。

oracle.olapi.data.source パッケージでは、次の表に示すクラスが Oracle OLAP API によって定義されます。

クラス	説明
CursorSpecification	サブクラスによって継承されるメソッドを実装する抽象スーパークラス。
ValueCursorSpecification	値を持ち、出力を持たない Source の CursorSpecification。
CompoundCursorSpecification	1 つ以上の出力を持つ Source の CursorSpecification。 CompoundCursorSpecification には、子コンポーネントである CursorSpecification オブジェクトが存在します。

Cursor の構造は、その CursorManagerSpecification の構造と同じです。CursorManagerSpecification の各 ValueCursorSpecification または CompoundCursorSpecification に対して、Cursor は対応する ValueCursor または CompoundCursor を持ちます。Cursor から特定の情報または動作を取得できるようにするには、アプリケーションによって Cursor が作成される前に、対応する CursorSpecification のメソッドをコールすることによって、そのアプリケーションでその情報または動作が必要であることを指定する必要があります。

CursorInput クラス

CursorInput は、DataProvider の createCursorManager メソッドへの inputSources 引数である Source オブジェクトの配列に含める Source の値を指定します。1 つ以上の入力を持つ Source 用の CursorManagerSpecification を作成する場合、その CursorManagerSpecification の CursorManager を作成するときに inputSources 引数を指定する必要があります。createCursorManagerSpecification メソッドに渡す Source の各入力用の inputSources 配列に、Source を含めます。

CursorInput オブジェクトを作成する場合、1 つの値または ValueCursor を指定できます。ValueCursor を指定した場合、CursorInput の synchronize メソッドをコールして、CursorInput の値を ValueCursor の現在の値にすることができます。

CursorManager クラス

CursorManager は、CursorManager によって作成された Cursor オブジェクトのデータのバッファリングを管理します。CursorManager を作成するには、DataProvider の createCursorManager メソッドをコールして、そのメソッドに CursorManagerSpecification を渡します。その CursorManagerSpecification の Source が 1 つ以上の入力を持つ場合、Source オブジェクトの配列も createCursorManager メソッドに渡します。この配列に、各入力の Source を含めます。

同じ CursorManager から複数の Cursor を作成できます。これは、1 つの結果セットのデータを、表やグラフなどの異なる形式で表示する場合に有効です。1 つの CursorManager によって作成されたすべての Cursor オブジェクトの仕様（デフォルトのフェッチ・サイズやフェッチ・サイズが設定されたレベルなど）は同じです。仕様が同じである Cursor オブジェクト間では、CursorManager によって管理されるデータを共有できます。

CursorManager には、Cursor の作成、CursorManager の CursorManagerSpecification を更新する必要があるかどうかの確認、および CursorManagerUpdateListener の追加または削除を行うためのメソッドが含まれています。SpecifiedCursorManager インタフェースによって、CursorManagerSpecification の更新、SpecifiedCursorManager がオープンしているかどうかの確認、および SpecifiedCursorManager のクローズを行うためのメソッドが追加されます。DataProvider の createCursorManager メソッドによって、SpecifiedCursorManager インタフェースの実装が戻されます。

アプリケーションで SpecifiedCursorManager が不要になった場合、それをクローズしてアプリケーションおよび Oracle OLAP のリソースを解放する必要があります。SpecifiedCursorManager をクローズするには、close メソッドをコールします。

CursorManager の CursorManagerSpecification の更新

アプリケーションで OLAP API Template オブジェクトが使用されており、Template の状態の変更に従ってその Template によって生成された Source の構造が変更された場合、

その Source 用のすべての CursorManagerSpecification オブジェクトが無効になります。変更された Source に対して新しい CursorManagerSpecification オブジェクトを作成する必要があります。

新しい CursorManagerSpecification を作成した後、その Source の新しい CursorManager を作成できます。ただし、新しい CursorManager は、必ずしも作成する必要はありません。既存の CursorManager の updateSpecification メソッドをコールして、以前の CursorManagerSpecification を新しい CursorManagerSpecification に置き換えることができます。その後、CursorManager から新しい Cursor を作成できます。

CursorManager の CursorManagerSpecification を更新する必要があるかどうかを判断するには、CursorManager の isSpecificationUpdateNeeded メソッドをコールします。CursorManagerUpdateListener を使用して、Source の変更によって生成されたイベントをリスニングすることもできます。詳細は、8-12 ページの「CursorManagerUpdateListener クラス」を参照してください。

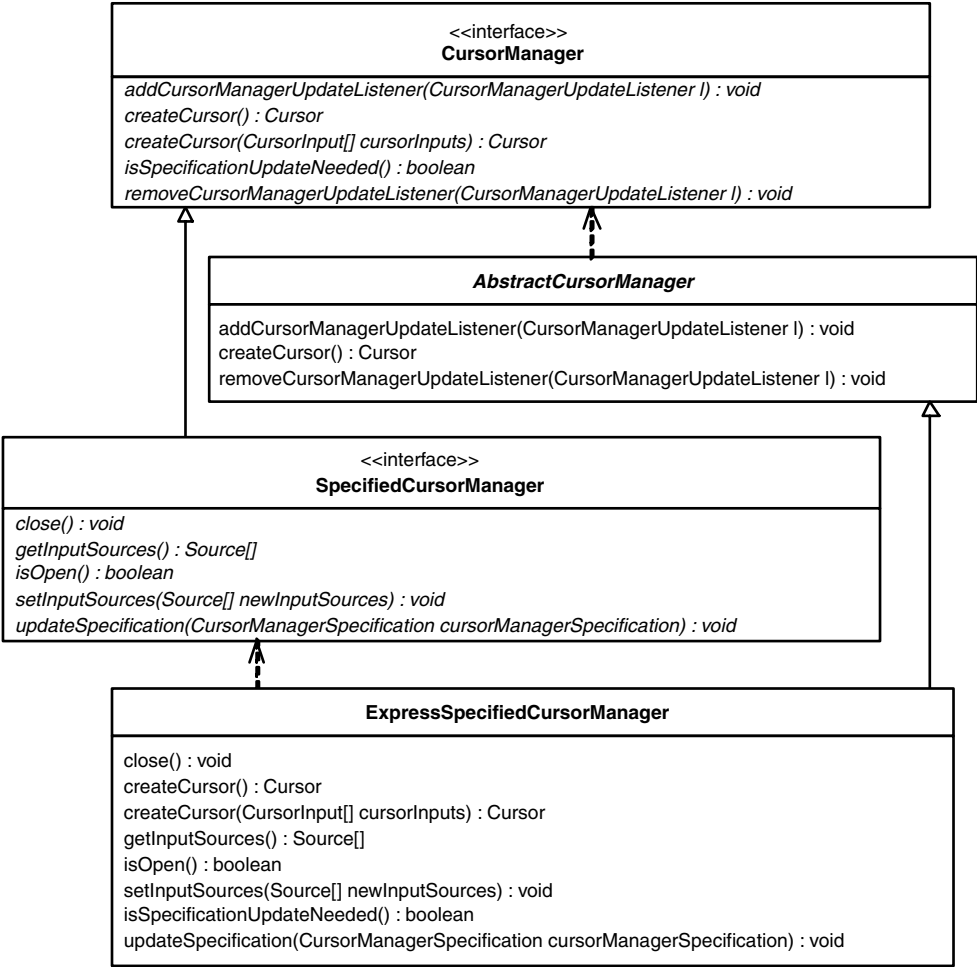
CursorManager のクラス階層

次の表に、CursorManager のインタフェースおよびクラスの大部分を示します。

インタフェース / クラス	説明
CursorManager	すべての CursorManager オブジェクトのメソッドを定義するインタフェース。
AbstractCursorManager	CursorManagerUpdateListener オブジェクトを追加および削除するメソッドを実装する CursorManager。詳細は、8-12 ページの「CursorManagerUpdateListener クラス」を参照してください。
SpecifiedCursorManager	CursorManager の追加のメソッドを定義するインタフェース。
ExpressSpecifiedCursorManager	SpecifiedCursorManager インタフェースを実装し、AbstractCursorManager を拡張するクラス。Oracle OLAP API では、DataProvider の createCursorManager メソッドによってこのクラスのインスタンスが戻されます。

図 8-3 に、前述の表に示す CursorManager クラスの関係を示します。実線の矢印は、あるクラスが、矢印が指すクラスを拡張することを示しています。破線の矢印は、そのクラスが、矢印が指すインタフェースを実装することを示しています。

図 8-3 CursorManager の階層



CursorManagerUpdateListener クラス

CursorManagerUpdateListener は、CursorManagerUpdateEvent オブジェクトを受け取るメソッドを持つインタフェースです。Oracle OLAP では、Template によって生成された Source が変更された場合、または CursorManager によって CursorManagerSpecification が更新された場合に、CursorManagerUpdateEvent オブジェクトが生成されます。アプリケーションで CursorManagerUpdateListener を

使用して、CursorManager から新しい Cursor オブジェクトを作成する必要があるか、または Cursor からのデータ表示を更新する必要があることを示すイベントをリスニングできます。

CursorManagerUpdateListener を使用するには、そのインタフェースを実装し、そのクラスのインスタンスを作成し、その CursorManagerUpdateListener を任意の Source の CursorManager に追加します。Source が変更されると、CursorManager によって CursorManagerUpdateListener の適切なメソッドがコールされ、そのメソッドに CursorManagerUpdateEvent が渡されます。その後、アプリケーションで新しい Cursor オブジェクトの生成に必要なタスクを実行し、Source によって定義された結果セットの値の表示を更新できます。

CursorManagerUpdateListener インタフェースは、複数のバージョンを実装できます。それらのインタフェースのインスタンスを同じ CursorManager に追加できます。

CursorManagerUpdateEvent クラス

Oracle OLAP では、Template によって生成された Source が変更された場合、または CursorManager によって CursorManagerSpecification が更新された場合に、CursorManagerUpdateEvent オブジェクトが生成されます。

このクラスのインスタンスは直接作成しません。Oracle OLAP によって CursorManagerUpdateEvent オブジェクトが生成され、CursorManager に追加した CursorManagerUpdateListener オブジェクトの適切なメソッドに渡されます。CursorManagerUpdateEvent には、発生したイベントのタイプを示すフィールドがあります。CursorManagerUpdateEvent のメソッドを使用して、それに関する情報を取得できます。

Cursor の位置およびエクステント

Cursor には、1 つ以上の位置があります。Cursor の現在の位置は、Cursor 内で現在アクティブな位置です。Cursor の現在の位置を移動するには、その Cursor の setPosition または next メソッドをコールします。

Oracle OLAP では、Cursor に操作 (getCurrentValue メソッドのコールなど) を試行しないかぎり、Cursor に設定した位置が検証されません。現在の位置を、負の値か、または Cursor 内の位置の数より大きい値に設定して Cursor 操作を試行すると、Cursor によって PositionOutOfBoundsException が発生します。

Cursor のエクステントの詳細は、8-22 ページの「[Cursor のエクステントの概要](#)」を参照してください。

ValueCursor の位置

ValueCursor の現在の位置には、取得可能な 1 つの値が指定されています。たとえば、productSel (8-5 ページの「[Cursor の構造](#)」に示す導出 Source) は、製品のディメンションおよび階層的なグループを示すプライマリ Source から 3 つの製品を選択したものです。productSel の ValueCursor には、3 つの要素が存在します。次の例では、ValueCursor の各要素の位置を取得し、その位置の値を表示します。output オブジェクトは、PrintWriter です。

```
// productSelValCursor is the ValueCursor for productSel
do {
    output.print(productSelValCursor.getPosition + " : ");
    output.println(productSelValCursor.getCurrentValue);
}
while(productSelValCursor.next());
```

前述の例によって表示される値は、次のとおりです。

```
1 : 815
2 : 1050
3 : 2055
```

次の例では、productSelValCursor の現在の位置を 2 に設定し、その位置の値を取得します。

```
productSelValCursor.setPosition(2);
output.println(productSelValCursor.getCurrentValue);
```

前述の例によって表示される値は、次のとおりです。

```
1050
```

ValueCursor から現在の値を取得するその他の例については、[第 9 章](#)を参照してください。

CompoundCursor の位置

CompoundCursor では、その子 ValueCursor オブジェクトの要素の各セットに対して 1 つの位置が存在します。CompoundCursor の現在の位置によって、それらのセットのいずれかが指定されます。

たとえば、unitPriceByDay (8-5 ページの「[Cursor の構造](#)」に示す Source) には、メジャー unitPrice からの値が含まれます。これらの値は、異なる時点での製品の単価です。unitPriceByDay の出力は、時間ディメンションから選択された 4 つの日付値および製品ディメンションから選択された 3 つの製品値を表す Source オブジェクトです。

unitPriceByDay の結果セットでは、各タプル（出力値の各セット）に 1 つのメジャー値が存在するため、値の合計数は 12 になります（4 つの日付のそれぞれに対して 3 つの製品の

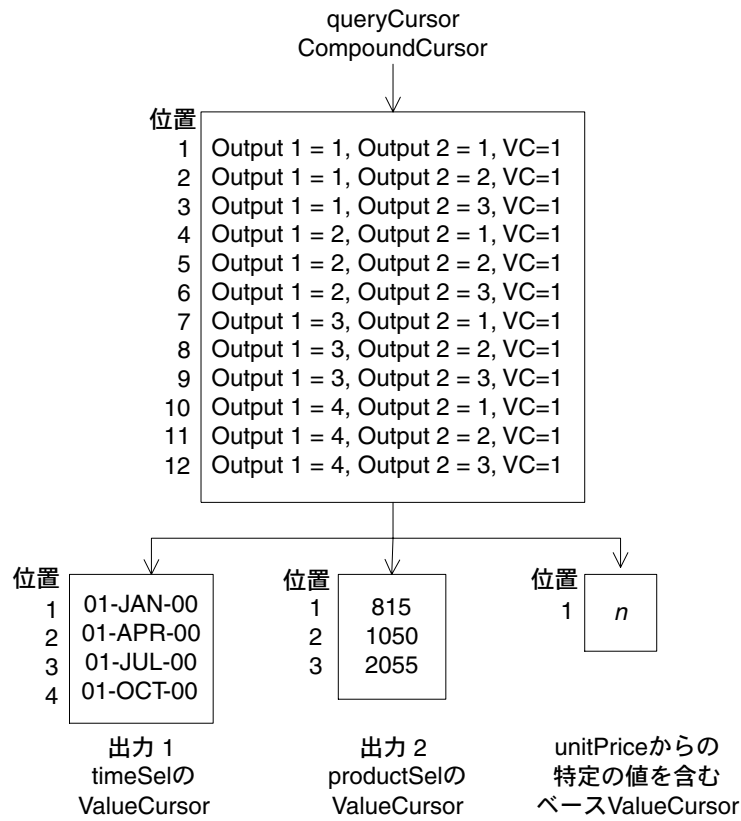
それぞれ 1 つの値)。したがって、unitPriceByDay 用に作成された CompoundCursor である queryCursor には 12 の位置が存在します。

queryCursor の各位置によって、その出力およびベース ValueCursor の 1 つの位置セットが指定されます。たとえば、queryCursor の位置 1 によって、その出力およびベース ValueCursor の次の位置セットが定義されます。

- 出力 1 の位置 1 (timeSel の ValueCursor)
- 出力 2 の位置 1 (productSel の ValueCursor)
- queryCursor のベース ValueCursor の位置 1 (この位置には、出力の値によって指定される unitPrice メジャーの値が存在します。)

図 8-4 に、CompoundCursor である queryCursor の位置、ベース ValueCursor および出力を示します。

図 8-4 queryCursor 内の Cursor の位置



任意の 1 セットの出力値によって `unitPrice` の値が 1 つのみ指定されるため、`queryCursor` の `ValueCursor` には位置が 1 つのみ存在します。`unitPriceByDay` のような問合せの場合、その `Cursor` の `ValueCursor` の値と位置は、ルート・レベルの `CompoundCursor` の任意の 1 つの位置に対して、一度にそれぞれ 1 つのみになります。

次の表に、`queryCursor` からのデータ表示の 1 例を示します。これは、4 つの列および 5 つの行で構成されたクロス集計ビューです。左の列は日付値です。一番上の行は製品値です。クロス集計の交差している各セルは、その日の製品の価格です。

日付	製品		
	815	1050	2055
01-JAN-00	58	24	24
01-APR-00	59	24	25
01-JUL-00	59	25	25
01-OCT-00	61	25	26

CompoundCursor では、その ValueCursor オブジェクトの位置が互いに相対的に調整されます。CompoundCursor の現在の位置によって、その子 ValueCursor オブジェクトの現在の位置が指定されます。[例 8-1](#) では、queryCursor の位置を設定し、子 Cursor オブジェクトの現在の値と位置を取得します。

例 8-1 CompoundCursor の位置の設定および現在の値の取得

```
CompoundCursor rootCursor = (CompoundCursor) queryCursor;
ValueCursor baseValueCursor = rootCursor.getValueCursor();
List outputs = rootCursor.getOutputs();
ValueCursor output1 = (ValueCursor) outputs.get(0);
ValueCursor output2 = (ValueCursor) outputs.get(1);
int pos = 5;
rootCursor.setPosition(pos);
System.out.println("CompoundCursor position set to " + pos + ".");
System.out.println("CC position = " + rootCursor.getPosition() + ".");
System.out.println("Output 1 position = " + output1.getPosition() +
    ", value = " + output1.getCurrentValue());
System.out.println("Output 2 position = " + output2.getPosition() +
    ", value = " + output2.getCurrentValue());
System.out.println("VC position = " + baseValueCursor.getPosition() +
    ", value = " + baseValueCursor.getCurrentValue());
```

[例 8-1](#) によって表示される値は、次のとおりです。

```
CompoundCursor position set to 5.
CC position = 5.
Output 1 position = 2, value = 01-APR-00
Output 2 position = 2, value = 1050
VC position = 1, value = 24
```

queryCursor の位置は、unitPriceByDay の結果セットに常に各時間値に対する 3 つの製品値が含まれるという点で対称的です。したがって、productSel の ValueCursor に

は、timeSel の ValueCursor の各値の 3 つの位置が常に含まれています。timeSel の出力 ValueCursor は、productSel の ValueCursor より遅く変化します。

ただし、非対称の場合、ValueCursor 内の位置の数は、遅く変化する出力に対して常に同じではありません。たとえば、2000 年 10 月 1 日の製品 2055 の単価が null で（その日にその製品が購入されなかったため）、問合せで null 値が抑制されている場合、queryCursor に含まれる位置は 11 のみです。timeSel の ValueCursor の位置が 4 の場合、productSel の ValueCursor に含まれる位置は 2 つのみです。

例 8-2 では、8-5 ページの「**Cursor の構造**」の問合せを修正したものを使用して、非対称の結果セットを生成します。修正された問合せの結果セットでは、任意の日の価格によって製品を指定します。productByPriceOnDay のベース値は、unitPrice および timeSel の値によって指定される、productSel の値です。

productByPriceOnDay は導出 Source であるため、この例では現行の Transaction を準備してコミットします。この例の TransactionProvider は tp です。Transaction オブジェクトの詳細は、[第 7 章](#)を参照してください。

この例では、productByPriceOnDay の Cursor を作成し、CompoundCursor の位置を通してループし、各子 ValueCursor オブジェクトの位置と現在の値を取得して、位置と値を表示します。

例 8-2 非対称問合せでの位置

```
// Create the query
productByPriceOnDay = productSel.join(unitPrice).join(timeSel);

//Prepare and commit the current Transaction.
try{
    tp.prepareCurrentTransaction();
}
catch (NotCommittableException e){
    output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create the Cursor. The DataProvider is dp.
CursorManagerSpecification cursorMgrSpec =
    dp.createCursorManagerSpecification(productByPriceOnDay);
CursorManager cursorManager = dp.createCursorManager(cursorMgrSpec);
Cursor queryCursor2 = cursorManager.createCursor();

// Get the ValueCursor and the outputs
CompoundCursor rootCursor = (CompoundCursor) queryCursor2;
ValueCursor baseValueCursor = rootCursor.getValueCursor();
List outputs = rootCursor.getOutputs();
ValueCursor output1 = (ValueCursor) outputs.get(0);
ValueCursor output2 = (ValueCursor) outputs.get(1);
```

```
// Get the positions and values and display them.
System.out.println("    CC %t%tOutput 1 %tOutput 2 %tVC");
System.out.println("position %tposition:value " +
    "%tposition:value %tposition:value");

do {
    System.out.println("        " + root.getPosition() +
        "%t%t    " + output1.getPosition() +
        "    : " + output1.getCurrentValue() +
        "%t    " + output2.getPosition() +
        "    : " + output2.getCurrentValue() +
        "%t    " + baseValueCursor.getPosition() +
        "    : " + baseValueCursor.getCurrentValue());
}
while(queryCursor2.next());
```

例 8-2 によって表示される値は、次のとおりです。

CC	Output 1	Output 2	VC
position	position:value	position:value	position:value
1	1 : 01-JAN-00	1 : 58	1 : 815
2	1 : 01-JAN-00	2 : 24	1 : 1050
3	1 : 01-JAN-00	2 : 24	2 : 2055
4	2 : 01-APR-00	1 : 59	1 : 815
5	2 : 01-APR-00	2 : 24	1 : 1050
6	2 : 01-APR-00	3 : 25	1 : 2055
7	3 : 01-JUL-00	1 : 59	1 : 815
8	3 : 01-JUL-00	2 : 25	1 : 1050
9	3 : 01-JUL-00	2 : 25	2 : 2055
10	4 : 01-OCT-00	1 : 61	1 : 815
11	4 : 01-OCT-00	2 : 25	1 : 1050
12	4 : 01-OCT-00	3 : 26	1 : 2055

unitPrice 値（出力 2）を持つ ValueCursor には、01-JAN-00 および 01-JUL-00 に対して 2 つの異なる値しかないため、これらの値に対して 2 つの位置しか存在しません。2 つの製品の価格が、これら 2 つの日付で同じです。製品 1050 および 2055 は、2000 年 1 月 1 日では 24、2000 年 7 月 1 日では 25 です。timeSel 値が 01-JAN-00 または 01-JUL-00 の場合、queryCursor2 のベース ValueCursor には 2 つの位置が存在します。これは、これらの日付の各 unitPrice 値が一意ではないためです。

Cursor 内での親の開始 / 終了位置

CompoundCursor から取得するデータの表示を効率的に管理するために、現在の遅く変化する値に対して存在する速く変化する値の数を把握することが必要な場合があります。たとえば、クロス集計で、キューブの 1 つのエッジの値を 1 つの行に表示する場合、その行に対して表示する列の数を把握する必要がある場合があります。

子 Cursor の現在の値に対して存在する速く変化する値の数を判断するには、その現在の値の、親 Cursor の中での開始位置および終了位置を検出します。次に示すとおり、終了位置から開始位置を引き、1 を足します。

```
long span = (cursor.getParentEnd() - cursor.getParentStart()) + 1;
```

これによって、親 Cursor の中での、子 Cursor の現在の値のスパンが計算されます。このスパンから、現在の値に対して存在する子 Cursor の最も速く変化する値の数を判断できます。開始位置と終了位置の計算は、多くの時間およびコンピューティング資源が必要であるため、アプリケーションでその情報が必要な場合にのみこれらの計算を実行するように指定します。

Oracle OLAP API の Cursor を使用すると、アプリケーションで、現在表示しているデータのみをクライアント・コンピュータに実際に存在させることができます。Cursor のデータ量を指定する方法の詳細は、8-23 ページの「[フェッチ・サイズおよびフェッチ・ブロック](#)」を参照してください。

ただし、クライアント・コンピュータ上のデータからは、親 Cursor の中での、子 Cursor の現在の値が開始または終了する位置を判断することはできません。その情報を取得するには、Cursor の getParentStart および getParentEnd メソッドを使用します。

たとえば、アプリケーションに、非対称のエッジを持つキューブを表す cube という名前の Source が存在すると想定します。このキューブは、4 つの出力を持ちます。この cube Source では、2000 年の第 1 四半期に特定の都市で顧客への販売額が 5000 ドルより大きかった製品を定義しています。これらの製品は、テレビによる宣伝期間中（TV）に、直接経路（S）を介して販売されました。

その Source の Cursor を作成し、cubeCursor という名前を付けます。CompoundCursor である cubeCursor には、次の子 Cursor オブジェクトが存在します。

- 出力 1。宣伝の値の ValueCursor です。
- 出力 2。経路の値の ValueCursor です。
- 出力 3。時間の値の ValueCursor です。
- 出力 4。顧客の値の ValueCursor です。
- ベース ValueCursor。販売額が 5000 ドルより大きい製品を値として持ちます。

図 8-5 に、子 Cursor オブジェクトの値を階層的に表した親 cubeCursor を示します。一番上の行は、最も遅く変化する出力である宣伝の値、一番下の行は、最も速く変化する子である製品の値です。ユーザー・インタフェースに現在表示されているエッジ部分は、太線内に示されている、cubeCursor の位置 7～9 の間のブロックのみです。cubeCursor の位置 1～10 は、一番上の行の上に示しています。

図 8-5 cubeCursor の子 ValueCursor の値

1	2	3	4	5	6	7	8	9	10
TV									
S									
2000-01			2000-02				2000-03		
Bonn		London	Bonn		London	Paris	Bonn	London	
1050	2055	815	1050	1555	935	1050	935	1050	3690

時間 Source の出力 ValueCursor の現在の値は、2000-02 です。ブロック内のデータからは、親 cubeCursor の中での現在の値 2000-02 の開始位置が 4、終了位置が 7 であることは判断できません。

前述の図の cubeCursor を、再度、図 8-6 に示します。ここでは、子 Cursor オブジェクトの各値を、親 cubeCursor の位置の範囲で示します。大きい値から小さい値を引いて 1 を足すことによって、各値のスパンを計算できます。たとえば、時間の値 2000-02 のスパンは、 $(7 - 4 + 1) = 4$ です。

図 8-6 cubeCursor の子 Cursor オブジェクトの位置の範囲

1	2	3	4	5	6	7	8	9	10
1-10									
1-10									
1 to 3			4 to 7				8 to 10		
1 to 2		3 to 3	4 to 5		6 to 6	7 to 7	8 to 8	9 to 10	
1 to 1	2 to 2	3 to 3	4 to 4	5 to 5	6 to 6	7 to 7	8 to 8	9 to 9	10 to 10

親 Cursor の中での子 Cursor の値の開始位置および終了位置を計算するように Oracle OLAP に指定するには、その Cursor に対応する CursorSpecification の `setParentStartCalculationSpecified` および `setParentEndCalculationSpecified` メソッドをコールします。開始位置と終了位置の計算が指定されているかどうかは、その CursorSpecification の `isParentStartCalculationSpecified` または `isParentEndCalculationSpecified` メソッドをコールすることによって判断できます。これらの計算を指定する例については、第 9 章を参照してください。

Cursor のエクステントの概要

Cursor のエクステントは、遅く変化する出力に対してその Cursor が含む要素の合計数です。図 8-7 に、遅く変化する出力の値に対する、cubeCursor の子 Cursor ごとの位置数を示します。子 Cursor オブジェクトを階層的に示します。一番上の行が最も速く変化する出力です。

cubeCursor 内の要素の合計数は 10 であるため、cubeCursor のエクステントは 10 です。この数は、一番上の行の上に示します。一番上の行は宣伝の値の ValueCursor で、その下の行は経路の値の ValueCursor です。これらの各 ValueCursor オブジェクトが持つ値は 1 つのみであるため、これらのエクステントは 1 です。

上から 3 番目の行は時間の値を表します。3 か月分の値があるため、このエクステントは 3 です。その次の行は都市ごとの顧客の ValueCursor です。この要素のエクステントは、遅く変化する出力（時間）の値によって異なります。最初の月の顧客 ValueCursor のエクステントは 2、2 か月目は 3、3 か月目は 2 です。

一番下の行は、CompoundCursor である cubeCursor のベース ValueCursor です。この値は製品です。製品 ValueCursor の要素のエクステントは、顧客 ValueCursor および時間 ValueCursor の値によって異なります。たとえば、2 つの製品の値が最初の月と都市の値のセットによって指定されるため（2000-01 の Bonn の 1050 および 2055）、そのセットの製品 ValueCursor のエクステントは 2 です。2 番目の顧客と時間の値のセット（2000-10、London）の場合、製品 ValueCursor のエクステントは 1 です（以後、同様に続きます）。

図 8-7 cubeCursor の子 Cursor オブジェクトの要素数

10									
1									
1									
1			2				3		
1		2	1		2	3	1	2	
1	2	1	1	2	1	1	1	1	2

エクステントの情報は、たとえば、正確な列数や正確なサイズのスクロールバーを表示するために使用できます。ただし、エクステントの計算は、コストが高くなる場合があります。たとえば、キューブを表す Source が 4 つの出力を持ち、各出力には数百の値が含まれる場合があります。出力セットのメジャーのすべての null 値と 0（ゼロ）値が結果セットから排除された場合、Source の CompoundCursor のエクステントを計算するには、Oracle OLAP で、CompoundCursor を作成する前に結果領域全体を横断検索する必要があります。エクステントの計算を指定しない場合、Oracle OLAP では、Cursor のフェッチ・サイズによって指定されるとおり、アプリケーションの必要に応じて、キューブの出力によって定義された要素のセットを横断検索する必要があるのみです。

Oracle OLAP で Cursor のエクステントを計算するように指定するには、その Cursor に対応した CursorSpecification の `setExtentCalculationSpecified` メソッドをコールします。エクステントの計算が指定されているかどうかは、CursorSpecification の `isExtentCalculationSpecified` メソッドをコールすることによって判断できます。Cursor のエクステントの計算を指定する例については、[第9章](#)を参照してください。

フェッチ・サイズおよびフェッチ・ブロック

OLAP API の Cursor は、Source の結果セット全体を表します。ただし、その Cursor が Oracle OLAP から一度に取得するのは結果セットの一部のみであるため、それは仮想的な Cursor です。CursorManager は、仮想 Cursor を管理し、アプリケーションの必要に応じて Oracle OLAP から結果を取得します。CursorManager で仮想 Cursor が管理されることによって、アプリケーションの負荷が大幅に軽減されます。

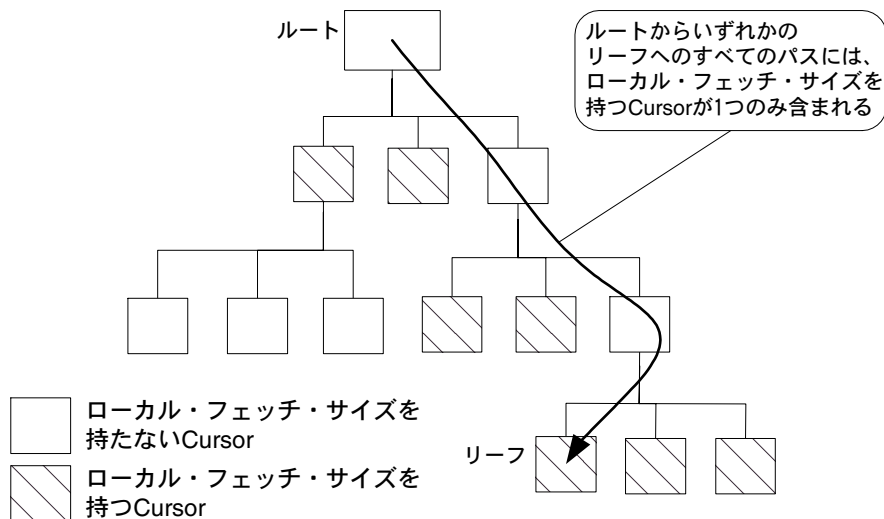
1 回のフェッチ操作で Cursor が取得するデータの量は、その Cursor に指定されたフェッチ・サイズによって決定されます。CompoundCursor の場合、1 回の操作でフェッチされるデータの量は、その子のすべての ValueCursor オブジェクトのフェッチ・サイズの積です。1 回のフェッチで取得される値の合計セットが、その Cursor のフェッチ・ブロックです。フェッチ・サイズは、アプリケーションがローカル・コンピュータにキャッシュする必要があるデータの量を制限し、データの表示方法の要件を満たすようにキャッシュをカスタマイズすることによってフェッチの効率を最大限に向上させるために指定します。

Source の CursorManagerSpecification を作成する際、Cursor 作成の最初の手順として、Oracle OLAP によって CursorManagerSpecification のルート・レベルの CursorSpecification にデフォルトのフェッチ・サイズが指定されます。CursorManagerSpecification の CursorSpecification オブジェクトのメソッドをコールすることによって、デフォルトのフェッチ・サイズを指定するか、または CompoundCursor の他のレベルでフェッチ・サイズを設定することを指定できます。

フェッチ・サイズが CursorSpecification に指定された場合、その Cursor の `getFetchSize` または `setFetchSize` をコールすることによって、対応する Cursor のフェッチ・サイズを取得または設定できます。CompoundCursor の場合、出力内の様々なレベルで子 Cursor オブジェクトに異なるフェッチ・サイズを設定できます。

フェッチ・ブロックのサイズが指定されている Cursor は、ローカル・フェッチ・サイズを持ちます。CompoundCursor 内のすべての Cursor オブジェクトがローカル・フェッチ・サイズを持つわけではありません。CompoundCursor の構造はツリーのようなものであり、Cursor オブジェクトの階層が最上位の（ルート）Cursor から始まり、すべての子 Cursor オブジェクトがその下に続きます。ルートから始まりリーフ ValueCursor まで続く階層内のすべてのパスは、ローカル・フェッチ・サイズを持つ Cursor を 1 つのみ含むことができます。親 Cursor にフェッチ・サイズを指定すると、その親の子のすべての Cursor オブジェクトが影響を受けます。これは、フェッチ・ブロックには、フェッチ・サイズによって指定された、各子 Cursor の要素数以下しか含めることができないことを意味します。[図 8-8](#) に、Cursor ツリーの階層内のパスの例を示します。ローカル・フェッチ・サイズを持つ Cursor は斜線で示します。

図 8-8 Cursor 階層内のローカル・フェッチ・サイズのパス



フェッチ・ブロックの形式の決定

CompoundCursor では、フェッチ・サイズを設定するレベルによって、CompoundCursor のフェッチ・ブロックの形式が決定されます。CompoundCursor の最適なフェッチ・ブロックは、Cursor のナビゲート方法およびデータの表示方法によって異なります。データの表示方法を決定した後、次の操作を実行します。

- ユーザー・インタフェースに表示する結果セットの部分に必要なすべてのデータを含めるために十分大きいフェッチ・ブロックを指定します。たとえば、データを表に表示し、ウィンドウに一度に表示できる行数が 25 である場合、25 行以上を含めることができるフェッチ・ブロックを指定します。これより小さい値を指定すると、Cursor は表示のために、Oracle OLAP に複数回アクセスする必要があります。
- 結果セットをループするために使用する Cursor オブジェクトのフェッチ・サイズを指定します。たとえば、表ビューの場合はルート Cursor にフェッチ・サイズを設定し、クロス集計ビューの場合は子 Cursor オブジェクトにフェッチ・サイズを設定します。
- すべてのフェッチ・サイズの積によってフェッチ・ブロック内のセルの合計数が決まるため、この積は比較的小さくしておきます。すべてのフェッチ・サイズの積が大きすぎる場合、仮想 Cursor のメリットが得られなくなります。

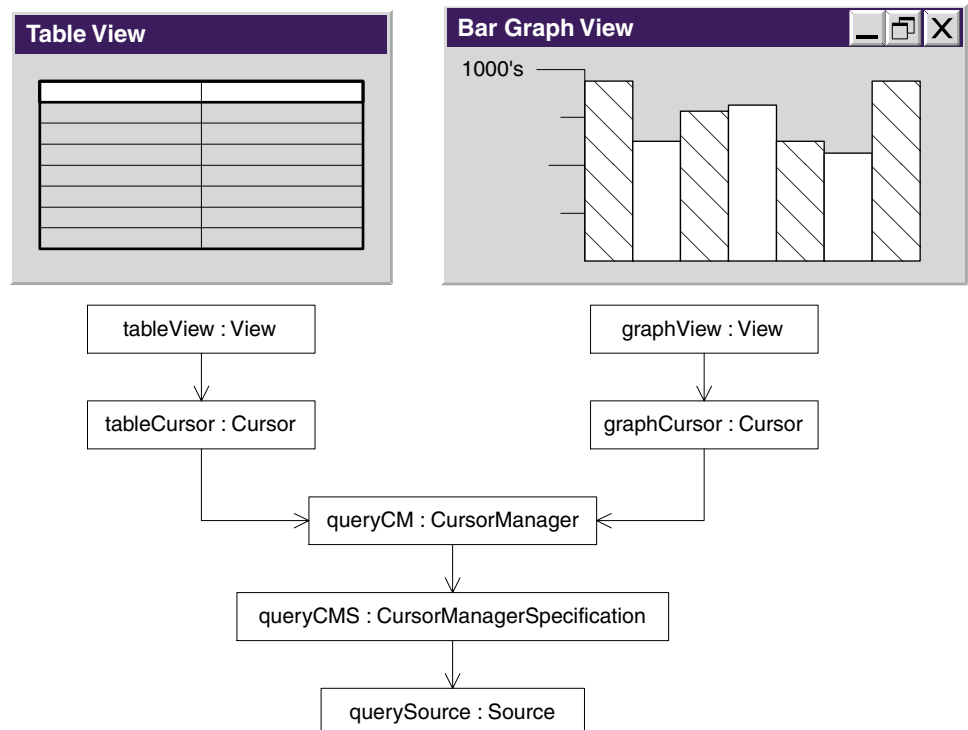
様々な表示用のフェッチ・サイズおよびフェッチ・ブロックの指定の例については、[第 9 章](#)を参照してください。

フェッチ・ブロックの共有

同じ `CursorManager` から複数の `Cursor` オブジェクトを作成し、それらの `Cursor` オブジェクトを同時に使用できます。これらの `Cursor` オブジェクトは、個別のデータ・キャッシュを持つのではなく、`CursorManager` によって管理されるデータを共有できます。これは、これらの `Cursor` オブジェクトのフェッチ・ブロックの形式が同じであるためです。フェッチ・ブロックの形式は、フェッチ・サイズが指定された `CursorManagerSpecification` のレベルによって決定されます。

この例は、問合せの結果を表とグラフの両方で表示するアプリケーションです。このアプリケーションは、`Source` の `CursorManagerSpecification` を作成し、その後 `CursorManagerSpecification` の `CursorManager` を作成します。このアプリケーションは、同じ `CursorManager` から、表ビューとグラフ・ビュー用に、2 つの個別の `Cursor` オブジェクトを作成します。これらの 2 つのビューは、同じ問合せを共有し、同じデータを表示しますが、その表示形式は異なります。図 8-9 に、`Source`、`Cursor` オブジェクトおよびビューの関係を示します。

図 8-9 1 つの `Source` および異なる値ビュー用の 2 つの `Cursor`



問合せ結果の取得

この章では、Oracle OLAP API の `Cursor` を使用して問合せの結果を取得する方法およびその結果にアクセスする方法について説明します。また、結果の表示方法にあわせて `Cursor` の動作をカスタマイズする方法についても説明します。`Cursor` のクラス階層や関連するクラス、および `Cursor` の位置、フェッチ・サイズおよびエクステンツの概念については、[第 8 章](#)を参照してください。

この章では、次の項目について説明します。

- [問合せ結果の取得](#)
- [様々なデータ表示のための `CompoundCursor` のナビゲート](#)
- [`Cursor` の動作の指定](#)
- [エクステンツおよび値の開始 / 終了位置の計算](#)
- [フェッチ・サイズおよびフェッチ・ブロックの指定](#)

問合せ結果の取得

問合せは、Oracle OLAP から取得するデータや、そのデータに対して Oracle OLAP を使用して実行する任意の計算を指定する OLAP API の Source です。Cursor は、Source によって指定された結果セットを取得（フェッチ）するオブジェクトです。Source 用の Cursor を作成するには、次の手順を実行します。

1. MdmObject からプライマリ Source を取得します。または、DataProvider または Source を操作して導出 Source を作成します。Source オブジェクトを取得または作成する方法の詳細は、[第 5 章](#)を参照してください。
2. Source が導出 Source の場合、Source を作成した Transaction を準備してコミットします。Transaction を準備してコミットするには、TransactionProvider の prepareCurrentTransaction および commitCurrentTransaction メソッドをコールします。Transaction を準備してコミットする方法の詳細は、[第 7 章](#)を参照してください。Source がプライマリ Source の場合、Transaction を準備してコミットする必要はありません。
3. DataProvider の createCursorManagerSpecification メソッドをコールして、そのメソッドに Source を渡すことによって、CursorManagerSpecification を作成します。
4. DataProvider の createCursorManager メソッドをコールして、そのメソッドに CursorManagerSpecification を渡すことによって、SpecifiedCursorManager を作成します。CursorManagerSpecification の Source が 1 つ以上の入力を持つ場合、各入力に Source を提供する Source オブジェクトの配列も渡す必要があります。
5. CursorManager の createCursor メソッドをコールすることによって、Cursor を作成します。入力 Source オブジェクトの配列を使用して CursorManager を作成している場合、各入力 Source に値を提供する CursorInput オブジェクトの配列も渡す必要があります。

例 9-1 では、querySource という名前の導出 Source 用の Cursor が作成されます。この例では、tp という名前の TransactionProvider および dp という名前の DataProvider が使用されます。例では、cursorMngrSpec という名前の CursorManagerSpecification、cursorMngr という名前の SpecifiedCursorManager および queryCursor という名前の Cursor が作成されます。

例では、最後に SpecifiedCursorManager がクローズされています。Cursor を使用した後は、SpecifiedCursorManager をクローズしてリソースを解放する必要があります。

例 9-1 Cursor の作成

```
try{
    tp.prepareCurrentTransaction();
}
catch (NotCommittableException e) {
```

```

        System.out.println("Caught exception " + e + ".");
    }
    tp.commitCurrentTransaction();
    CursorManagerSpecification cursorMgrSpec =
        dp.createCursorManagerSpecification(querySource);
    SpecifiedCursorManager cursorMgr =
        dp.createCursorManager(cursorMgrSpec);
    Cursor queryCursor = cursorMgr.createCursor();

    // ... Use the Cursor in some way, such as to display its values.

    cursorMgr.close();

```

Cursor からの値の取得

現在の位置の概念は、Cursor インタフェースによってカプセル化されます。このインタフェースには、現行の位置を移動するためのメソッドがあります。ValueCursor および CompoundCursor インタフェースは、Cursor インタフェースを拡張します。Oracle OLAP API には、ValueCursor および CompoundCursor インタフェースの実装が存在します。CursorManager の createCursor メソッドをコールすると、Cursor を作成中の Source に応じて、ValueCursor 実装または CompoundCursor 実装が戻されます。

ValueCursor は、単一セットの値が存在する Source の場合に戻されます。ValueCursor は、現在の位置に値を持ち、現在の位置の値を取得するためのメソッドを含みます。

CompoundCursor は、2 セット以上の値が存在する Source (1 つ以上の出力を持つ Source) 用に作成されます。Source の値の各セットは、CompoundCursor の子 ValueCursor によって表されます。CompoundCursor には、その子 Cursor オブジェクトを取得するためのメソッドが存在します。

Source の構造によって Cursor の構造が決定します。Source は、ネストした出力を持つ場合があります。これは、Source の 1 つ以上の出力自体が、出力を持つ Source である場合に発生します。Source がネストした出力を持つ場合、Source 用の CompoundCursor には、ネストした出力用の子 CompoundCursor が存在します。

CompoundCursor によって、その子 Cursor オブジェクトの位置が調整されます。CompoundCursor の現在の位置によって、その子 Cursor オブジェクトの 1 セットの位置が指定されます。

出力値のレベルが 1 つのみの Source の例については、[例 9-4](#) を参照してください。ネストした出力値を持つ Source の例については、[例 9-5](#) を参照してください。

値の単一セットを表す Source の例には、MdmDimension の getSource メソッドによって戻されるもの（製品の値の階層リストを表す MdmDimension など）があります。その Source の Cursor を作成することによって、ValueCursor が戻されます。getCurrentValue メソッドをコールすることによって、ValueCursor の現在の位置の製品の値が戻されます。

例 9-2 では、製品の値を表す `MdmDimension` の `mdmProductHier` から `Source` が取得されます。また、その `Source` 用の `Cursor` が作成されます。例では、現在の位置が、`ValueCursor` の第 5 要素に設定されて、`Cursor` から製品の値が取得されます。その後、`CursorManager` がクローズされます。例の中の `dp` は、`DataProvider` です。

例 9-2 ValueCursor からの単一の値の取得

```
Source productSource = mdmProductHier.getSource();
// Because productSource is a primary Source, you do not need to
// prepare and commit the current Transaction.
CursorManagerSpecification cursorMgrSpec =
    dp.createCursorManagerSpecification(productSource);
SpecifiedCursorManager cursorMgr =
    dp.createCursorManager(cursorMgrSpec);
Cursor productCursor = cursorMgr.createCursor();
// Cast the Cursor to a ValueCursor.
ValueCursor productValues = (ValueCursor) productCursor;
// Set the position to the fifth element of the ValueCursor.
productValues.setPosition(5);

// Product values are strings. Get the String value at the current
// position.
String value = productValues.getCurrentString();

// Do something with the value, such as display it...

// Close the SpecifiedCursorManager.
cursorMgr.close();
```

例 9-3 では、例 9-2 と同じ `Cursor` が使用されています。例 9-3 では、`do...while` ループが使用され、`ValueCursor` の `next` メソッドを使用して `ValueCursor` の位置が移動されます。`next` メソッドは、有効な位置で開始され、`Cursor` にその他の位置がある場合、`true` が戻されます。また、このメソッドによって、現在の位置から次の位置に進められます。

例では、位置が、`ValueCursor` の最初の位置に設定されています。例では、位置をループして、`getCurrentValue` メソッドが使用され、その時点での現在の位置で値が取得されます。

例 9-3 ValueCursor からのすべての値の取得

```
// productValues is the ValueCursor for productSource
productValues.setPosition(1);
do {
    System.out.println(productValues.getCurrentValue());
}
while (productValues.next());
```

CompoundCursor によって表された結果セットの値は、CompoundCursor の子 ValueCursor オブジェクトに存在します。これらの値を取得するには、CompoundCursor から子 ValueCursor オブジェクトを取得する必要があります。

CompoundCursor の例としては、(メジャーのディメンションから選択した値によって指定される) メジャーの値を表す Source 用の CursorManager の createCursor メソッドをコールすることによって戻されるものがあります。

例 9-4 では、salesAmount という名前の Source が使用されます。この Source は、売上高を表す MdmMeasure の getSource メソッドをコールすることによって生成されます。メジャーのディメンションは、製品、顧客、時間、経路および宣伝を表す MdmDimension オブジェクトです。この例では、これらのディメンションから選択した値を表す Source オブジェクトが使用されます。該当する Source オブジェクトの名前は、prodSel、custSel、timeSel、chanSel および promoSel です。メジャーおよびディメンションの選択を表す Source オブジェクトの作成については、ここでは示しません。

例 9-4 では、ディメンションの選択がメジャーに結合されます。この結果、salesForSelections という名前の Source が生成されます。salesForSelections 用の salesForSelCursor という名前の Cursor が作成されます。また、Cursor が salesCompndCrshr という名前の CompoundCursor にキャストされ、ベース ValueCursor および CompoundCursor からの出力が取得されます。この場合、各出力は ValueCursor です。出力は、List に戻されます。List 内での出力の順序は、ディメンションがメジャーに結合される場合と逆の順序です。例の中の dp は DataProvider で、tp は TransactionProvider です。

例 9-4 CompoundCursor からの ValueCursor オブジェクトの取得

```
Source salesForSelections = salesAmount.join(prodSel)
                                         .join(custSel)
                                         .join(timeSel)
                                         .join(chanSel)
                                         .join(promoSel);

// Prepare and commit the current Transaction
try{
    tp.prepareCurrentTransaction();
}
catch (NotCommittableException e){
    output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for salesForSelections
CursorManagerSpecification cursorMgrSpec =
    dp.createCursorManagerSpecification(salesForSelections);
SpecifiedCursorManager cursorMgr =
    dp.createCursorManager(cursorMgrSpec);
Cursor salesForSelCursor = cursorMgr.createCursor();
```

```
// Cast salesForSelCursor to a CompoundCursor
CompoundCursor salesCompndCrsr = (CompoundCursor) salesValues;

// Get the base ValueCursor
ValueCursor specifiedSalesVals = salesCompndCrsr.getValueCursor();

// Get the outputs
List outputs = salesCompndCrsr.getOutputs();
ValueCursor promoSelVals = (ValueCursor) outputs.get(0);
ValueCursor chanSelVals = (ValueCursor) outputs.get(1);
ValueCursor timeSelVals = (ValueCursor) outputs.get(2);
ValueCursor custSelVals = (ValueCursor) outputs.get(3);
ValueCursor prodSelVals = (ValueCursor) outputs.get(4);

// You can now get the values from the ValueCursor objects.
// When you have finished using the Cursor objects, close the
// SpecifiedCursorManager.
cursorMgr.close();
```

例 9-5 で使用される売上高のメジャーは、例 9-4 と同じものですが、ディメンションの選択をメジャーに結合する方法は異なります。例 9-5 では、2 つのディメンションの選択が結合されます。その後、結果は、単一のディメンションの選択をメジャーに結合することによって生成された Source に結合されます。その結果生成された Source の salesForSelections では、問合せにネストした出力が存在することが表されます。これは、問合せに 2 つ以上のレベルの出力があることを意味します。

したがって、この例で salesForSelections 用に作成される CompoundCursor にも、ネストした出力が存在します。CompoundCursor は、子ベース ValueCursor を持ちます。また、その出力は、3 つの子 ValueCursor オブジェクトおよび 1 つの子 CompoundCursor を持ちます。

例 9-5 では、宣伝ディメンション値の選択 (promoSel) が経路ディメンション値の選択 (chanSel) に結合されます。その結果は、ベース値として経路の値、出力値として宣伝の値を持つ Source (chanByPromoSel) です。製品、顧客および時間の値の選択は、salesAmount に結合され、その後 chanByPromoSel に結合されます。これによって生成される問合せは、salesForSelections によって表されます。

例では、現行の Transaction が準備およびコミットされ、salesForSelections 用の salesForSelCursor という名前の Cursor が作成されます。

Cursor は salesCompndCrsr という名前の CompoundCursor にキャストされ、ベース ValueCursor およびその出力が取得されます。例の中の dp は DataProvider で、tp は TransactionProvider です。

例 9-5 ネストした出力を持つ CompoundCursor からの値の取得

```
// ...in someMethod...
```



```
Source chanByPromoSel = chanSel.join(promoSel);
Source salesForSelections = salesAmount.join(prodSel)
                                .join(custSel)
                                .join(timeSel)
                                .join(chanByPromoSel);

// Prepare and commit the current Transaction
try{
    tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
    output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for salesForSelections
CursorManagerSpecification cursorMgrSpec =
    dp.createCursorManagerSpecification(salesForSelections);
SpecifiedCursorManager cursorMgr =
    dp.createCursorManager(cursorMgrSpec);
Cursor salesForSelCursor = cursorMgr.createCursor();

// Send the Cursor to a method that does different operations
// depending on whether the Cursor is a CompoundCursor or a
// ValueCursor.
printCursor(salesForSelCursor);
cursorMgr.close();
// ...the remaining code of someMethod...

// The printCursor method has a do...while loop that moves through the positions
// of the Cursor passed to it. At each position, the method prints the number of
// the iteration through the loop and then a colon and a space. The output
// object is a PrintWriter. The method calls the private _printTuple method and
// then prints a new line. A "tuple" is the set of output ValueCursor values
// specified by one position of the parent CompoundCursor. The method prints one
// line for each position of the parent CompoundCursor.
public void printCursor(Cursor rootCursor) {
    int i = 1;
    do {
        output.print(i++ + ": ");
        _printTuple(rootCursor);
        output.print("\n");
        output.flush();
    }
    while(rootCursor.next());
}
```

```
// If the Cursor passed to the _printTuple method is a ValueCursor,
// the method prints the value at the current position of the ValueCursor.
// If the Cursor passed in is a CompoundCursor, the method gets the
// outputs of the CompoundCursor and iterates through the outputs,
// recursively calling itself for each output. The method then gets the
// base ValueCursor of the CompoundCursor and calls itself again.
private void _printTuple(Cursor cursor) {
    if(cursor instanceof CompoundCursor) {
        CompoundCursor compoundCursor = (CompoundCursor)cursor;
        // Put an open parenthesis before the value of each output
        output.print("(");
        Iterator iterOutputs = compoundCursor.getOutputs().iterator();
        Cursor output = (Cursor)iterOutputs.next();
        _printTuple(output);
        while(iterOutputs.hasNext()) {
            // Put a comma after the value of each output
            output.print(",");
            _printTuple((Cursor)iterOutputs.next());
        }
        // Put a comma after the value of the last output
        output.print(",");
        // Get the base ValueCursor
        _printTuple(compoundCursor.getValueCursor());

        // Put a close parenthesis after the base value to indicate
        // the end of the tuple.
        output.print(")");
    }
    else if(cursor instanceof ValueCursor) {
        ValueCursor valueCursor = (ValueCursor) cursor;
        if (valueCursor.hasCurrentValue())
            print(valueCursor.getCurrentValue());
        else
            // If this position has a null value
            print("NA");
    }
}
```

様々なデータ表示のための CompoundCursor のナビゲート

CompoundCursor のメソッドを使用すると、簡単にその構造全体を移動（ナビゲート）して、子 ValueCursor から値を取得できます。OLAP の多次元問合せからのデータは、通常、クロス集計形式または表やグラフとして表示されます。

複数の行および列にデータを表示するには、表示の要件に応じて、様々なレベルの CompoundCursor の位置をループします。表などの一部の表示方法の場合、親

CompoundCursor の位置をループします。クロス集計などのその他の表示方法の場合、子 Cursor オブジェクトの位置をループします。

問合せの結果を表ビューで表示する場合、各行が出力の各 ValueCursor およびベース ValueCursor からの値で構成されるため、最上位（ルート・レベル）の CompoundCursor の位置を決定して、その位置を反復します。例 9-6 に、ある時点での結果セットの一部のみを表示します。製造コストの値を持つメジャーに基づいた問合せを表す Source 用の Cursor が作成されます。そのメジャーのディメンションは、製品ディメンションおよび時間ディメンションです。プライマリ Source オブジェクトの作成、およびディメンションの導出された選択については、ここでは示しません。

例では、ディメンション値の選択を表す Source オブジェクトが、メジャーを表す Source に結合されます。現行の Transaction は準備およびコミットされ、Cursor が作成されます。Cursor は CompoundCursor にキャストされます。この例では、CompoundCursor の位置が設定され、CompoundCursor の 12 の位置を反復し、それらの位置で指定された値が出力されます。TransactionProvider は tp で、DataProvider は dp です。output オブジェクトは PrintWriter です。

例 9-6 表ビュー用のナビゲート

```
Source unitPriceByDay = unitPrice.join(productSel)
                                   .join(timeSel);

try{
    tp.prepareCurrentTransaction();
}
catch (NotCommitableException e) {
    output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for unitPriceByDay
CursorManagerSpecification cursorMgrSpec =
    dp.createCursorManagerSpecification(unitPriceByDay);
SpecifiedCursorManager cursorMgr =
    dp.createCursorManager(cursorMgrSpec);
Cursor unitPriceByDayCursor = cursorMgr.createCursor();

// Cast the Cursor to a CompoundCursor
CompoundCursor rootCursor = (CompoundCursor) unitPriceByDayCursor;

// Determine a starting position and the number of rows to display
int start = 7;
int numRows = 12;

// Iterate through the specified positions of the root CompoundCursor.
// Assume that the Cursor contains at least (start + numRows) positions.
for(int pos = start; pos < start + numRows; pos++) {
```

```
// Set the position of the root CompoundCursor
rootCursor.setPosition(pos);
// Print the values of the output ValueCursors
output.print(rootCursor.getOutputs().get(0).getCurrentValue() + "¥t");
output.print(rootCursor.getOutputs().get(1).getCurrentValue() + "¥t");
// Print the value of the base ValueCursor and a new line
output.print(rootCursor.getValueCursor().getCurrentValue() + "¥n");
output.flush();
};
cursorMgr.close();
```

問合せの時間の選択に 8 つの値（1999 年および 2000 年の各四半期の初日など）が存在し、製品の選択に 3 つの値が存在する場合、unitPriceByDay 問合せの結果セットには、24 の位置が存在します。例 9-6 では、次のような表が表示されます。この表には、CompoundCursor の 7 ～ 18 の位置によって指定された値が含まれます。

01-JUL-99	815	57
01-JUL-99	1050	23
01-JUL-99	2055	22
01-OCT-99	815	56
01-OCT-99	1050	24
01-OCT-99	2055	21
01-JAN-00	815	58
01-JAN-00	1050	24
01-JAN-00	2055	24
01-APR-00	815	59
01-APR-00	1050	24
01-APR-00	2055	25

例 9-7 では、例 9-6 と同じ問合せが使用されます。クロス集計ビューでは、第 1 行が列ヘッダーです。この例でのヘッダーは、timeSel からの値です。timeSel ディメンションの選択は最初にメジャーに結合されているため、timeSel の出力は、速く変化する出力です。残りの行は、行ヘッダーで始まります。行ヘッダーは、遅く変化する出力の productSel からの値です。列ヘッダーの下に表示される残りの行の位置には、ディメンション値のセットによって指定された unitPrice の値が含まれます。

クロス集計ビューで問合せの結果を表示するには、最上位の CompoundCursor の子の位置を指定して、その位置を反復します。例 9-7 では、結果の値が取得されます。ただし、クロス集計表示の適切なセルに値を入れるためのコードは含まれていません。

例 9-7 単一ページのクロス集計ビュー用のナビゲート

```
Source unitPriceByDay = unitPrice.join(productSel)
                                     .join(timeSel);

try{
    tp.prepareCurrentTransaction();
}
```

```
catch(NotCommittableException e){
    output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for unitPriceByDay
CursorManagerSpecification cursorMgrSpec =
    dp.createCursorManagerSpecification(unitPriceByDay);
SpecifiedCursorManager cursorMgr =
    dp.createCursorManager(cursorMgrSpec);
Cursor unitPriceByDayCursor = cursorMgr.createCursor();

// Cast the Cursor to a CompoundCursor
CompoundCursor rootCursor = (CompoundCursor) unitPriceByDayCursor;

// Determine a starting position and the number of rows to display.
// colStart is the position in columnCursor at which the current
// display starts and rowStart is the position in rowCursor at
// which the current display starts.
int colStart = 1;
int rowStart = 1;
String productValue;
String timeValue;
double price;
int numProducts = 3;
int numDays = 12;

// Get the outputs and the ValueCursor
CompoundCursor rootCursor = (CompoundCursor) unitPriceByDayCursor;
List outputs = rootCursor.getOutputs();
// The first output has the values of timeSel, the slower varying output
ValueCursor rowCursor = (ValueCursor) outputs.get(0);
// The second output has the faster varying values of productSel
ValueCursor columnCursor = (ValueCursor) outputs.get(1);
ValueCursor unitPriceValues = rootCursor.getValueCursor();// Prices

// Loop through positions of the faster varying output Cursor
for(int pPos = colStart; pPos < colStart + numProducts; pPos++) {
    columnCursor.setPosition(pPos);
    // Loop through positions of the slower varying output Cursor
    for(int tPos = rowStart; tPos < rowStart + numDays; tPos++) {
        rowCursor.setPosition(tPos);
        // Get the values. Sending the values to the appropriate
        // display mechanism is not shown.
        productValue = columnCursor.getCurrentString();
        timeValue = rowCursor.getCurrentString();
        price = unitPriceValues.getCurrentDouble();
```

```
    }  
  }  
  cursorMngr.close();
```

図 9-1 のクロス集計ビューは、unitPriceByDay 問合せによって指定された結果セットからの値を表示したものです。

図 9-1 unitPriceByDay によって指定された結果セットのクロス集計ビュー

	815	1050	2055
01-JAN-99	56	22	21
01-APR-99	57	22	21
01-JUL-99	57	23	22
01-OCT-99	56	24	21
01-JAN-00	58	24	24
01-APR-00	59	24	25
01-JUL-00	59	25	25
01-OCT-00	61	25	26

例 9-8 では、売上高のメジャーに基づく Source が作成されます。そのメジャーのディメンションは、顧客ディメンション、製品ディメンション、時間ディメンション、経路ディメンションおよび宣伝ディメンションです。それらのディメンション用の Source オブジェクトは、ディメンション値の選択を表します。これらの Source オブジェクトの作成については、ここでは示しません。

ディメンションの選択をメジャーの Source に結合することによって作成される問合せは、その出力値によって指定される総売上高の値を表します。

この例では、その問合せの Cursor が作成され、その Cursor が printAsCrosstab メソッドに送られます。このメソッドは、Cursor からの値をクロス集計に出力します。このメソッドは、ページ値、列値および行値を出力する別のメソッドをコールします。

Cursor の最も速く変化する出力は、顧客の選択で、フランス、イギリスおよびアメリカ合衆国のすべての顧客を指定する 3 つの値が存在します。顧客の値はクロス集計の列ヘッダーです。第 2 位の速く変化する出力は、製品の選択で、製品の種類を指定する 4 つの値が存在します。ページ・ディメンションは、時間の 2 つの値（2000 年の第 1 および第 2 四半期）、経路の 1 つの値（直接経路）および宣伝の 1 つの値（すべての宣伝）の選択です。

TransactionProvider は tp で、DataProvider は dp です。output オブジェクトは、PrintWriter です。

例 9-8 複数ページのクロス集計ビュー用のナビゲート

```
// ...in someMethod...
Source salesAmountsForSelections = salesAmount.join(customerSel)
                                                .join(productSel);
                                                .join(timeSel);
                                                .join(channelSel);
                                                .join(promotionSel);

try{
    tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
    output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for salesAmountsForSelections
CursorManagerSpecification cursorMgrSpec =
    dp.createCursorManagerSpecification(salesAmountsForSelections);
SpecifiedCursorManager cursorMgr =
    dp.createCursorManager(cursorMgrSpec);
Cursor salesForSelCursor = cursorMgr.createCursor();

// Send the Cursor to the printAsCrosstab method
printAsCrosstab(salesForSelCursor);

cursorMgr.close();
// ...the remainder of the code of someMethod...

// This method expects a CompoundCursor.
private void printAsCrosstab(Cursor cursor) {
    // Cast the Cursor to a CompoundCursor
    CompoundCursor rootCursor = (CompoundCursor) cursor;
    List outputs = rootCursor.getOutputs();
    int nOutputs = outputs.size();

    // Set the initial positions of all outputs
    Iterator outputIter = outputs.iterator();
    while (outputIter.hasNext())
        ((Cursor) outputIter.next()).setPosition(1);

    // The last output is fastest-varying; it represents columns.
    // The next to last output represents rows.
    // All other outputs are on the page.
    Cursor colCursor = (Cursor) outputs.get(nOutputs - 1);
    Cursor rowCursor = (Cursor) outputs.get(nOutputs - 2);
    ArrayList pageCursors = new ArrayList();
    for (int i = 0 ; i < nOutputs - 2 ; i++) {
```

```
        pageCursors.add(outputs.get(i));
    }

    // Get the base ValueCursor, which has the data values
    ValueCursor dataCursor = rootCursor.getValueCursor();

    // Print the pages of the crosstab
    printPages(pageCursors, 0, rowCursor, colCursor, dataCursor);
}

// Prints the pages of a crosstab
private void printPages(List pageCursors, int pageIndex, Cursor rowCursor,
                        Cursor colCursor, ValueCursor dataCursor) {
    // Get a Cursor for this page
    Cursor pageCursor = (Cursor) pageCursors.get(pageIndex);

    // Loop over the values of this page dimension
    do {
        // If this is the fastest-varying page dimension, print a page
        if (pageIndex == pageCursors.size() - 1) {
            // Print the values of the page dimensions
            printPageHeadings(pageCursors);

            // Print the column headings
            printColumnHeadings(colCursor);

            // Print the rows
            printRows(rowCursor, colCursor, dataCursor);

            // Print a couple of blank lines to delimit pages
            output.println();
            output.println();
        }

        // If this is not the fastest-varying page, recurse to the
        // next fastest varying dimension.
        else {
            printPages(pageCursors, pageIndex + 1, rowCursor, colCursor,
                      dataCursor);
        }
    } while (pageCursor.next());

    // Reset this page dimension Cursor to its first element.
    pageCursor.setPosition(1);
}

// Prints the values of the page dimensions on each page
```



```
private void printPageHeadings(List pageCursors) {
    // Print the values of the page dimensions
    Iterator pageIter = pageCursors.iterator();
    while (pageIter.hasNext())
        output.println(((ValueCursor) pageIter.next()).getCurrentValue());
    output.println();
}

// Prints the column headings on each page
private void printColumnHeadings(Cursor colCursor) {
    do {
        output.print("¥t");
        output.print(((ValueCursor) colCursor).getCurrentValue());
    } while (colCursor.next());
    output.println();
    colCursor.setPosition(1);
}

// Prints the rows of each page
private void printRows(Cursor rowCursor, Cursor colCursor,
                       ValueCursor dataCursor) {
    // Loop over rows
    do {
        // Print row dimension value
        output.print(((ValueCursor) rowCursor).getCurrentValue());
        output.print("¥t");
        // Loop over columns
        do {
            // Print data value
            output.print(dataCursor.getCurrentValue());
            output.print("¥t");
        } while (colCursor.next());
        output.println();

        // Reset the column Cursor to its first element
        colCursor.setPosition(1);
    } while (rowCursor.next());

    // Reset the row Cursor to its first element
    rowCursor.setPosition(1);
}
```

例 9-8 のクロス集計の出力は、次のようになります。

```
Promotion total
Direct
2000-Q1
```

	FR	UK	US
Outerwear - Men	750563.50	938014.00	12773925.50
Outerwear - Women	984461.00	1388755.50	15421979.00
Outerwear - Boys	693382.00	799452.00	9183052.00
Outerwear - Girls	926520.50	977291.50	11854203.00
Promotion total			
Direct			
2000-Q2			

	FR	UK	US
Outerwear - Men	683521.00	711945.00	9947221.50
Outerwear - Women	840024.50	893587.50	12484221.00
Outerwear - Boys	600382.50	755031.00	8791240.00
Outerwear - Girls	901558.00	909421.50	9975927.00

Cursor の動作の指定

指定可能な Cursor の動作は、次のとおりです。

- Cursor のフェッチ・サイズ。これは、1 回のフェッチ操作中に Cursor によって取得される結果セットの要素数です。
- Cursor のフェッチ・ブロックの形式。フェッチ・ブロックは、親 CompoundCursor によって取得される各子 ValueCursor の要素のセットです。フェッチ・ブロックの形式は、フェッチ・サイズを設定する CompoundCursor のレベルです。
- Oracle OLAP によって、Cursor のエクステントを計算するかどうか。エクステントは、Cursor の位置の合計数です。Cursor が、CompoundCursor の子 Cursor である場合、そのエクステントは、任意の遅く変化する出力に対して相対的です。
- Oracle OLAP によって、子 Cursor の値が開始または終了する、親 Cursor の中での位置が計算されるかどうか。

Cursor の動作を指定するには、その Cursor 用の CursorSpecification のメソッドを使用します。Cursor 用の CursorSpecification を取得するには、Source 用に作成した CursorManagerSpecification のメソッドを使用します。

注意： エクステントあるいは（子 Cursor の現在の値に対する）親 Cursor の中での開始位置または終了位置を計算するように指定すると、非常にコストが高くなる場合があります。この計算には、多くの時間と計算リソースが使用される場合があります。アプリケーションに必要な場合にのみ、これらの計算を指定してください。

Source、Cursor、CursorSpecification および CursorManagerSpecification オブジェクトの関係、あるいはフェッチ・サイズ、エクステンツまたは Cursor の位置の概念の詳細は、[第 8 章](#)を参照してください。

[例 9-9](#) では、Source およびその CursorManagerSpecification が作成された後に、CursorManagerSpecification から CursorSpecification オブジェクトが取得されます。ルート CursorSpecification は、最上位の CompoundCursor の CursorSpecification です。

例 9-9 CursorManagerSpecification からの CursorSpecification オブジェクトの取得

```
Source salesAmountsForSelections = salesAmount.join(customerSel)
                                                .join(productSel);
                                                .join(timeSel);
                                                .join(channelSel);
                                                .join(promotionSel);

try{
    tp.prepareCurrentTransaction();
}
catch (NotCommittableException e){
    output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for salesAmountsForSelections
CursorManagerSpecification cursorMgrSpec =
    dp.createCursorManagerSpecification(salesAmountsForSelections);

// Get the root CursorSpecification of the CursorManagerSpecification.
CompoundCursorSpecification rootCursorSpec =
    (CompoundCursorSpecification) cursorMgrSpec.getRootCursorSpecification();

// Get the CursorSpecification for the base values
ValueCursorSpecification baseValueSpec =
    rootCursorSpec.getValueCursorSpecification();

// Get the CursorSpecification objects for the outputs
List outputSpecs = rootCursorSpec.getOutputs();
ValueCursorSpecification promoSelValCSpec =
    (ValueCursorSpecification) outputSpecs.get(0);
ValueCursorSpecification chanSelValCSpec =
    (ValueCursorSpecification) outputSpecs.get(1);
ValueCursorSpecification timeSelValCSpec =
    (ValueCursorSpecification) outputSpecs.get(2);
ValueCursorSpecification prodSelValCSpec =
    (ValueCursorSpecification) outputSpecs.get(3);
ValueCursorSpecification custSelValCSpec =
    (ValueCursorSpecification) outputSpecs.get(4);
```

CursorSpecification オブジェクトを取得した後は、そのメソッドを使用して、それに対応している Cursor オブジェクトの動作を指定できます。

エクステントおよび値の開始 / 終了位置の計算

CompoundCursor によって取得された結果セットの表示を管理するために、その子 Cursor コンポーネントのエクステントを知っておくことが必要な場合があります。親 CompoundCursor の中で子 Cursor の現在の値が開始される位置も知っておくことが必要な場合があります。子 Cursor の現在の値のスパンを知っておくことが必要な場合もあります。スパンは、子 Cursor の現在の値が占有する、親 Cursor の位置の数です。値の終了位置から開始位置を引き、さらに 1 を引くことによって、スパンを計算できます。

Cursor のエクステント、あるいは親 Cursor の中での値の開始位置または終了位置を取得する前に、Oracle OLAP でエクステントまたはそれらの位置の計算を行うことを指定する必要があります。これらの計算の実行を指定するには、Cursor 用の CursorSpecification のメソッドを使用します。

例 9-10 では、Cursor のエクステントの計算を指定しています。この例では、例 9-9 の CursorManagerSpecification が使用されています。

例 9-10 Cursor のエクステントの計算の指定

```
CompoundCursorSpecification rootCursorSpec =  
(CompoundCursorSpecification) cursorMgrSpec.getRootCursorSpecification();  
rootCursorSpec.setExtentCalculationSpecified(true);
```

CursorSpecification のメソッドを使用すると、次の例に示すように、CursorSpecification で、Cursor のエクステントを計算するように指定されているかどうかを判断できます。

```
boolean isSet = rootCursorSpec.isExtentCalculationSpecified();
```

例 9-11 では、子 Cursor の現在の値に対する、親 Cursor の中での開始位置および終了位置の計算が指定されています。この例では、例 9-9 の CursorManagerSpecification が使用されています。

例 9-11 親の中での開始 / 終了位置の計算の指定

```
CompoundCursorSpecification rootCursorSpec =  
(CompoundCursorSpecification) cursorMgrSpec.getRootCursorSpecification();  
  
// Get the List of CursorSpecification objects for the outputs.  
// Iterate through the list, specifying the calculation of the extent  
// for each output CursorSpecification.  
Iterator iterOutputSpecs = rootCursorSpec.getOutputs().iterator();  
ValueCursorSpecification valCursorSpec = (ValueCursorSpecification)  
    iterOutputSpecs.next();
```

```

while(iterOutputSpecs.hasNext()) {
    valCursorSpec.setParentStartCalculationSpecified(true);
    valCursorSpec.setParentEndCalculationSpecified(true);
    valCursorSpec = (ValueCursorSpecification) iterOutputSpecs.next();
}

```

CursorSpecification のメソッドを使用すると、次の例に示すように、CursorSpecification で、子 Cursor の現在の値に対する、親 Cursor の中での開始 / 終了位置を計算するように指定されているかどうかを判断できます。

```

boolean isSet;
Iterator iterOutputSpecs = rootCursorSpec.getOutputs().iterator();
ValueCursorSpecification valCursorSpec = (ValueCursorSpecification)
    iterOutputSpecs.next();

while(iterOutputSpecs.hasNext()) {
    isSet = valCursorSpec.isParentStartCalculationSpecified();
    isSet = valCursorSpec.isParentEndCalculationSpecified();
    valCursorSpec = (ValueCursorSpecification) iterOutputSpecs.next();
}

```

例 9-12 では、CompoundCursor の 2 つの出力に関して、(子 Cursor の現在の値に対する) 親 CompoundCursor の中での位置のスパンが決定されます。この例では、例 9-8 の salesAmountsForSelections Source が使用されています。

例では、時間および製品の選択の現在の値の開始および終了位置が取得され、親 Cursor の中でのこれらの値のスパンが計算されます。親は、ルート CompoundCursor です。TransactionProvider は tp で、DataProvider は dp で、output は PrintWriter です。

例 9-12 値の親の中での位置のスパンの計算

```

Source salesAmountsForSelections = salesAmount.join(customerSel)
                                                .join(productSel);
                                                .join(timeSel);
                                                .join(channelSel);
                                                .join(promotionSel);

try{
    tp.prepareCurrentTransaction();
}
catch(NotCommitableException e){
    output.println("Caught exception " + e + ".");
}

tp.commitCurrentTransaction();

// Create a CursorManagerSpecification for salesAmountsForSelections
CursorManagerSpecification cursorMgrSpec =
    dp.createCursorManagerSpecification(salesAmountsForSelections);

```

```
// Get the root CursorSpecification from the CursorManagerSpecification.
CompoundCursorSpecification rootCursorSpec =
(CompoundCursorSpecification) cursorMgrSpec.getRootCursorSpecification();
// Get the CursorSpecification objects for the outputs
List outputSpecs = rootCursorSpec.getOutputs();
ValueCursorSpecification timeSelValCSpec =
(ValueCursorSpecification) outputSpecs.get(2); ¥¥ output for time
ValueCursorSpecification prodSelValCSpec =
(ValueCursorSpecification) outputSpecs.get(3) ¥¥ output for product

// Specify the calculation of the starting and ending positions
timeSelValCSpec.setParentStartCalculationSpecified(true);
timeSelValCSpec.setParentEndCalculationSpecified(true);
prodSelValCSpec.setParentStartCalculationSpecified(true);
prodSelValCSpec.setParentEndCalculationSpecified(true);

// Create the CursorManager and the Cursor
SpecifiedCursorManager cursorMgr = dp.createCursorManager(cursorMgrSpec);
CompoundCursor cursor = (CompoundCursor) cursorMgr.createCursor();

// Get the child Cursor objects
ValueCursor baseValCursor = cursor.getValueCursor();
List outputs = cursor.getOutputs();
ValueCursor promoSelVals = (ValueCursor) outputs.get(0);
ValueCursor chanSelVals = (ValueCursor) outputs.get(1);
ValueCursor timeSelVals = (ValueCursor) outputs.get(2);
ValueCursor custSelVals = (ValueCursor) outputs.get(3);
ValueCursor prodSelVals = (ValueCursor) outputs.get(4);

// Set the position of the root CompoundCursor
cursor.setPosition(15);
/*
 * Get the values at the current position and determine the span
 * of the values of the time and product outputs.
 */
output.print(promoSelVals.getCurrentValue() + ", ");
output.print(chanSelVals.getCurrentValue() + ", ");
output.print(timeSelVals.getCurrentValue() + ", ");
output.print(custSelVals.getCurrentValue() + ", ");
output.print(prodSelVals.getCurrentValue() + ", ");
output.println(baseValCursor.getCurrentValue());

// Determine the span of the values of the two fastest varying outputs
int span;
span = (prodSelVals.getParentEnd() - prodSelVals.getParentStart()) - 1;
output.println("The span of " + prodSelVals.getCurrentValue() +
```

```
" at the current position is " + span + ".")
span = (timeSelVals.getParentEnd() - timeSelVals.getParentStart()) -1;
output.println("The span of " + timeSelVals.getCurrentValue() +
" at the current position is " + span + ".")
cursorMgr.close();
```

この例によって生成される出力は、次のとおりです。

```
Promotion total, Direct, 2000-Q1, Outerwear - Men, US, 9947221.50
The span of Outerwear - Men at the current position is 3.
The span of 2000-Q2 at the current position is 12.
```

フェッチ・サイズおよびフェッチ・ブロックの指定

1 回のフェッチ操作中に Oracle OLAP がクライアント・アプリケーションに送る `Cursor` の要素数は、その `Cursor` 用に指定されたフェッチ・サイズによって異なります。`CompoundCursor` の場合、`CompoundCursor` 自体、またはその子 `Cursor` コンポーネントの 1 つ以上のレベルにフェッチ・サイズを設定できます。`CompoundCursor` にフェッチ・サイズを設定することによって、その子 `Cursor` コンポーネント用にそのフェッチ・サイズを指定できます。

1 回のフェッチで `Cursor` が取得する要素のセットが、フェッチ・ブロックです。フェッチ・ブロックの形式は、フェッチ・サイズを設定する `Cursor` コンポーネントのセットによって決定します。フェッチ・サイズおよびフェッチ・ブロックの詳細は、[第 8 章](#)を参照してください。

フェッチ・ブロックの形式および具体的なフェッチ・サイズは、データ表示の要件に従って指定します。表ビューに問合せの結果を表示するには、最上位の `CompoundCursor` にフェッチ・サイズを指定します。

クロス集計ビューに結果を表示するには、最上位の `CompoundCursor` の子にフェッチ・サイズを指定します。複数レベルのネストした出力を持つ問合せの結果を表示するクロス集計の場合、様々なレベルのコンポーネント `CompoundCursor` オブジェクトの子にフェッチ・サイズを指定する場合があります。

`CursorSpecification` のメソッドを使用すると、その `Cursor` のデフォルトのフェッチ・サイズを設定できます。`CompoundCursorSpecification` の場合、その子にフェッチ・サイズを設定してフェッチ・ブロックの形式を指定および決定できます。

デフォルトのフェッチ・サイズが `CursorSpecification` に設定されている場合、`CursorSpecification` 用の `Cursor` の `setFetchSize` メソッドを使用して、`Cursor` のフェッチ・サイズを変更できます。デフォルトでは、`CursorManagerSpecification` のルート `CursorSpecification` のフェッチ・サイズは、100 に設定されています。

例 9-13 では、メジャーのディメンションからの値の選択によって指定される売上高のメジャー値を表す `Source` が作成されます。製品および顧客の選択には、それぞれ 10 の値が存在し、時間の選択には、4 つの値が存在します。宣伝および経路の選択には、それぞれ 1

つの値が存在します。売上高がディメンション値の各セットに対して存在すると想定すると、問合せの結果セットには、300 個 ($10 \times 10 \times 3 \times 1 \times 1$) の要素が含まれています。

20 行のみの要素の表示にあわせて、例では、最上位の `CompoundCursor` に 20 個の要素のフェッチ・サイズが設定されています。デフォルトのフェッチ・サイズは、ルート `CursorSpecification` (この例では、最上位の `CompoundCursor` である `CompoundCursorSpecification`) に自動的に設定されるため、この例では、フェッチ・サイズの変更に `CompoundCursor` の `setFetchSize` メソッドが使用されています。フェッチ・ブロックは、最上位の `CompoundCursor` の 20 個の位置によって指定される出力値とベース値のセットです。TransactionProvider は `tp` で、DataProvider は `dp` です。

例 9-13 表ビュー用のフェッチ・サイズおよびフェッチ・ブロックの指定

```
Source salesAmountsForSelections = salesAmount.join(customerSel)
                                              .join(productSel);
                                              .join(timeSel);
                                              .join(channelSel);
                                              .join(promotionSel);

try{
    tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
    output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for salesAmountsForSelections
CursorManagerSpecification cursorMgrSpec =
    dp.createCursorManagerSpecification(salesAmountsForSelections);
SpecifiedCursorManager cursorMgr = dp.createCursorManager(cursorMgrSpec);
Cursor cursor = cursorMgr.createCursor();

// Set the fetch size of the top-level CompoundCursor to 20
cursor.setFetchSize(20);
```

例 9-14 は、例 9-7 を変更したものです。例 9-14 では、for ループが繰り返される回数は、Cursor のエクステンツによって異なります。この例では、for ループの条件文として Cursor に存在する位置の数を指定するかわりに、Cursor のエクステンツが取得され、それが条件として使用されています。クロス集計に表示するうえで最も効果的なフェッチ・ブロックは、CompoundCursor の各位置に対してその位置での子 Cursor の要素のエクステンツを含むものです。

この例では、CursorManagerSpecification が作成され、ルート CursorSpecification が取得されます。ルート CursorSpecification は、CompoundCursorSpecification としてキャストされます。例では、ルート

CompoundCursorSpecification の子にデフォルトのフェッチ・サイズを設定することが指定されています。また、そのエクステントの計算についても指定されています。

例では、出力の各 ValueCursor に対するフェッチ・サイズが、その ValueCursor のエクステントと等しく設定されています。その後、子 ValueCursor オブジェクトの位置をループすることによって、クロス集計に表示可能な部分が取得されています。

例 9-14 エクステントを使用したクロス集計ビュー用のフェッチ・サイズの指定

```
Source unitPriceByDay = unitPrice.join(productSel)
                                   .join(timeSel);

try{
    tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
    output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a CursorManagerSpecification for unitPriceByDay
CursorManagerSpecification cursorMgrSpec =
    dp.createCursorManagerSpecification(unitPriceByDay);

// Get the root CursorSpecification and cast it to a
// CompoundCursorSpecification
CompoundCursorSpecification rootSpec =
    (CompoundCursorSpecification) cursorMgrSpec.getRootCursorSpecification();

// Specify setting the fetch size on the child Cursor objects
// and calculating the extent of the positions in the Cursor
rootSpec.specifyDefaultFetchSizeOnChildren();
rootSpec.setExtentCalculationSpecified(true);

// Create the CursorManager and the Cursor
SpecifiedCursorManager cursorMgr =
    dp.createCursorManager(cursorMgrSpec);
Cursor unitPriceByDayCursor = cursorMgr.createCursor();

// Cast the Cursor to a CompoundCursor
CompoundCursor rootCursor = (CompoundCursor) unitPriceByDayCursor;

// Determine a starting position and the number of rows to display.
// The position in columnCursor at which the current display starts
// is colStart and rowStart is the position in rowCursor at which
// the current display starts.
int colStart = 1;
int rowStart = 1;
```

```

String productValue;
String timeValue;
double price;

// The number of values from the ValueCursor objects for products and
// days are now initialized as 1 because the ValueCursor objects have
// at least one element.
int numProducts = 1;
int numDays = 1;

// Get the ValueCursor and the outputs
CompoundCursor rootCursor = (CompoundCursor) unitPriceByDayCursor;
List outputs = rootCursor.getOutputs();
// The first output has the values of timeSel, the slower varying output
ValueCursor rowCursor = (ValueCursor) outputs.get(0);
// The second output has the faster varying values of productSel
ValueCursor columnCursor = (ValueCursor) outputs.get(1);
ValueCursor unitPriceValues = rootCursor.getValueCursor();// Prices

// Loop through the positions of the faster varying output Cursor
for(int pPos = colStart; pPos < colStart + numProducts; pPos++) {
    columnCursor.setPosition(pPos);
    // Get the extents of the output ValueCursor objects
    numProducts = columnCursor.getExtent();
    numDays = rowCursor.getExtent();
    // Set the fetch sizes
    columnCursor.setFetchSize(numProducts);
    rowCursor.setFetchSize(numMonths);
    // Loop through the positions of the slower varying output Cursor
    for(int tPos = rowStart; tPos < rowStart + numDays; tPos++) {
        rowCursor.setPosition(tPos);

        // Get the values. Sending the values to the appropriate
        // display mechanism is not shown.
        productValue = columnCursor.getCurrentString();
        timeValue = rowCursor.getCurrentString();
        price = unitPriceValues.getCurrentDouble();
    }
}

```

動的問合せの作成

この章では、動的問合せの作成に使用する Oracle OLAP API の `Template` クラスおよびその関連クラスについて説明します。また、それらのクラスの実装例を示します。

この章では、次の項目について説明します。

- [Template オブジェクトの概要](#)
- [Template および関連クラスの概要](#)
- [Template の設計および実装](#)

Template オブジェクトの概要

Template クラスは、Oracle OLAP API の強力な機能の基礎となります。Template オブジェクトを使用して、変更可能な Source オブジェクトを作成します。これらの Source オブジェクトを使用すると、エンド・ユーザーの選択に応じて変更可能な動的問合せを作成できます。Template オブジェクトを使用すると、ユーザー・インタフェース要素を OLAP API の操作およびオブジェクトに簡単に変換できます。

次の項では、これらの機能の概要を示します。この章の後半では、動的 Source オブジェクトの作成に使用する Template クラスおよびその他のクラスについて説明します。動的 Source への変更、およびこれらの変更の保存や廃棄に使用する Transaction オブジェクトの詳細は、[第 7 章](#)を参照してください。

動的 Source の作成

Template の主な機能は、動的 Source の作成です。この機能は、Template によって使用される他のオブジェクトのうち、DynamicDefinition クラスおよび MetadataState クラスのインスタンスに基づいています。

Source を作成すると、SourceDefinition が自動的に作成されます。SourceDefinition には、Source の作成方法に関する情報が含まれています。作成された Source は、その SourceDefinition と永続的に対になります。SourceDefinition の getSource メソッドは、対になる Source を取得します。

DynamicDefinition は、SourceDefinition のサブクラスです。Template は、作成した Source の SourceDefinition のプロキシとして機能する DynamicDefinition を作成します。これは、DynamicDefinition の getSource メソッドが、永続的に対になる同じ Source を常に取得するのではなく、Template によって現在作成されている Source を取得することを意味します。取得された Source が異なる場合でも、DynamicDefinition のインスタンスは変更されません。

Template によって作成された Source は変更可能です。これは、Template で Source の作成に使用された値（他の Source オブジェクトを含む）が変更可能であるためです。Template は、これらの値を MetadataState に格納します。Template には、MetadataState の現在の状態の取得、値の取得または設定、および状態の設定を行うメソッドが含まれています。これらのメソッドを使用して、MetadataState に格納されたデータ値を変更します。

DynamicDefinition を使用して、Template によって作成された Source を取得します。アプリケーションが Template で Source の作成に使用された値の状態を（たとえばエンド・ユーザーの選択に応じて）変更した場合、新しい Source で以前の Source とは異なる結果セットが定義されている場合でも、同じ DynamicDefinition を使用して Source を再度取得します。

Template によって作成された Source は、他の Source オブジェクトを作成する一連の Source の操作（たとえば、選択、ソート、計算および結合の一連の操作）の結果になる可能性があります。これらの操作のコードを、Template 用の SourceGenerator の generateSource メソッドに格納します。このメソッドは、Template によって作成され

た Source を戻します。この操作では、MetadataState に格納されたデータが使用されません。

様々な Template オブジェクトによって作成された動的 Source オブジェクトが相互に作用する複雑な問合せを作成する場合があります。この問合せを作成すると、複雑な問合せ全体を定義する Source が作成されます。最終の Source の作成に使用した Template オブジェクトのいずれかの状態を変更する場合、最終の Source は以前の Source とは異なる結果セットを表します。これによって、問合せを定義するために行ったすべての操作を繰り返さずに、最終の問合せを変更できます。

ユーザー・インタフェース要素の OLAP API オブジェクトへの変換

Template オブジェクトを設計して、アプリケーションのユーザー・インタフェースの要素を表します。この Template オブジェクトによって、エンド・ユーザーの選択が、Source を作成する OLAP API の問合せ作成の操作に変換されます。その後、Cursor を作成して、Source で定義される結果セットを Oracle OLAP からフェッチします。Cursor から値を取得し、それらの値をエンド・ユーザーに表示します。エンド・ユーザーによって選択が変更された場合は、Template の状態を変更します。その後、Template によって作成された Source を取得して、新しい Cursor を作成し、取得した新しい値を表示します。

Template および関連クラスの概要

OLAP API では、いくつかのクラスが連携して動作して、動的 Source を作成します。Template の設計では、次のクラスおよびインタフェースを実装または拡張する必要があります。

- Template 抽象クラス
- MetadataState インタフェース
- SourceGenerator インタフェース

これらの3つのクラスのインスタンスと、Oracle OLAP によって作成される他のクラスのインスタンスが連携して動作して、Template によって定義された Source を作成します。Oracle OLAP で提供されるクラスは次のとおりです。これらのクラスは、ファクトリ・メソッドをコールして作成します。

- DataProvider
- DynamicDefinition

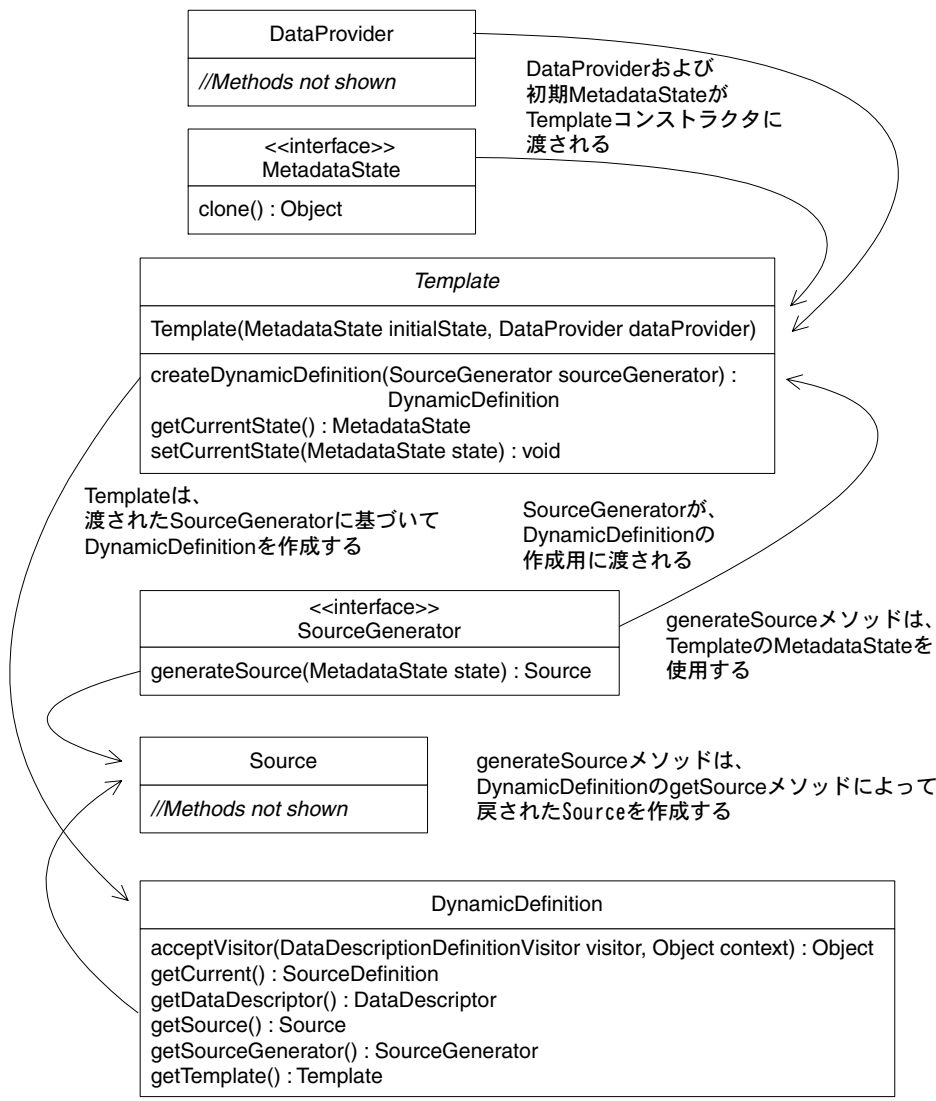
動的 Source を作成するクラス間の関連

動的 Source を作成するクラスは、次のように連携して動作します。

- Template には、DynamicDefinition を作成し、MetadataState の現在の状態を取得および設定するメソッドが含まれています。Template 抽象クラスの拡張では、MetadataState のフィールドの値を取得および設定するメソッドを追加します。
- MetadataState の実装には、Template 用の Source の作成に使用されるデータを格納するためのフィールドが存在します。新しい Template を作成する場合、MetadataState を Template のコンストラクタに渡します。DynamicDefinition の getSource メソッドをコールすると、MetadataState は SourceGenerator の generateSource メソッドに渡されます。
- DataProvider は、Template の作成に使用され、SourceGenerator によって新しい Source オブジェクトの作成に使用されます。
- SourceGenerator の実装には、MetadataState のデータの現在の状態を使用して Template 用の Source を作成する generateSource メソッドが含まれています。SourceGenerator を Template の createDynamicDefinition メソッドに渡して、DynamicDefinition を作成します。
- DynamicDefinition には、SourceGenerator によって作成された Source を取得する getSource メソッドが含まれています。DynamicDefinition は、取得された Source と永続的に対になる SourceDefinition のプロキシとして機能します。

図 10-1 に、これらのクラス間の関連を示します。右側の矢印は、DataProvider および MetadataState オブジェクトが Template コンストラクタに渡され、SourceGenerator が Template の createDynamicDefinition メソッドに渡されることを示します。左側の矢印は、createDynamicDefinition メソッドによって DynamicDefinition が戻され、SourceGenerator の generateSource メソッドおよび DynamicDefinition の getSource メソッドによって同じ Source が戻されることを示します。

図 10-1 動的 Source を作成するクラス間の関連



Template クラス

Template を使用して、変更可能な Source を作成します。Template には、DynamicDefinition を作成し、Template の現在の状態を取得および設定するメソッドが含まれています。Template クラスの拡張では、Template 用の MetadataState のフィールドにアクセスするためのメソッドを追加します。Template によって DynamicDefinition が作成され、これを使用して Template 用の SourceGenerator によって作成された Source を取得します。

Template の実装例については、10-8 ページの[例 10-1](#)を参照してください。

MetadataState インタフェース

MetadataState インタフェースの実装は、Template の値の現在の状態を格納します。MetadataState には、現在の状態のコピーを作成する clone メソッドが含まれる必要があります。

新しい Template をインスタンス化する場合、MetadataState を Template コンストラクタに渡します。Template には、MetadataState によって格納された値を取得および設定するためのメソッドが含まれています。SourceGenerator の generateSource メソッドは、Template 用の Source を作成する際に、MetadataState を使用します。

MetadataState の実装例については、10-11 ページの[例 10-2](#)を参照してください。

SourceGenerator インタフェース

SourceGenerator の実装には、Template 用の Source を作成する generateSource メソッドが含まれている必要があります。SourceGenerator が作成する必要がある Source は 1 種類のみです (BooleanSource、NumberSource、StringSource など)。generateSource メソッドは、Source を作成する際に、Template 用の MetadataState によって表されるデータの現在の状態を使用します。

generateSource メソッドによって作成された Source を取得するには、Template の createDynamicDefinition メソッドに SourceGenerator を渡して、DynamicDefinition を作成します。その後、DynamicDefinition の getSource メソッドをコールして、Source を取得します。

Template は、それぞれ異なる SourceGenerator を実装した複数の DynamicDefinition を作成できます。異なる SourceGenerator オブジェクトの generateSource メソッドで、Template 用の MetadataState の現在の状態によって定義された同じデータが使用され、異なる問合せを定義する Source オブジェクトが作成されます。

SourceGenerator の実装例については、10-12 ページの[例 10-3](#)を参照してください。

DynamicDefinition クラス

DynamicDefinition は、SourceDefinition のサブクラスです。Template の createDynamicDefinition メソッドをコールして、それに SourceGenerator を渡すことによって、DynamicDefinition を作成します。DynamicDefinition の getSource メソッドをコールして、SourceGenerator によって作成された Source を取得します。

Template によって作成された DynamicDefinition は、SourceGenerator によって作成された Source の SourceDefinition のプロキシです。SourceDefinition は、その Source と永続的に対になります。Template の状態が変更された場合、SourceGenerator によって作成される Source は異なります。DynamicDefinition はプロキシであるため、Source の SourceDefinition が異なる場合でも、同じ DynamicDefinition を使用して新しい Source を取得できます。

DynamicDefinition の getCurrent メソッドは、generateSource メソッドによって現在戻されている Source と永続的に対になる SourceDefinition を戻します。DynamicDefinition の使用例については、10-14 ページの例 10-4 を参照してください。

Template の設計および実装

Template の設計には、アプリケーションのユーザー・インタフェースの問合せ作成要素が反映されます。たとえば、エンド・ユーザーが値リストから多くの値を要求する問合せを作成可能なアプリケーションを開発すると想定します。値は、メジャーの 1 つのディメンションの値です。そのメジャーの他のディメンションは、単一の値に制限されます。

アプリケーションのユーザー・インタフェースでは、エンド・ユーザーがダイアログ・ボックスを使用して次の操作を実行できます。

- データ値が特定の範囲内にあるかどうかを指定するラジオ・ボタンの選択。
- ドロップダウン・リストからのメジャーの選択。
- フィールドからの数値の選択。この数値によって、表示するデータ値の数を指定します。
- 表示するデータ値のベースとしての、メジャーのディメンションの選択。たとえば、ユーザーが製品ディメンションを選択した場合、問合せによって、製品リストからいくつかの製品が指定されます。リストは、メジャー、および他のディメンションの選択値によって決定されます。
- 「Single Selections」ダイアログ・ボックスの表示。エンド・ユーザーは、このダイアログ・ボックスを介して、選択されているメジャーの他のディメンションに対する単一の値を選択します。ディメンションの値の選択後、エンド・ユーザーはこのダイアログ・ボックスの「OK」ボタンをクリックし、最初のダイアログ・ボックスに戻ります。
- 問合せの作成。「OK」ボタンをクリックすると問合せが作成され、問合せ結果が表示されます。

最初のダイアログ・ボックスでエンド・ユーザーが作成する問合せを表す Source を作成するには、TopBottomTemplate という名前の Template を設計します。また、SingleSelectionTemplate という名前の Template も設計し、エンド・ユーザーによるベース・ディメンション以外のディメンションに対する単一の値の選択を表す Source を作成します。Template オブジェクトの設計には、ダイアログ・ボックスのユーザー・インタフェース要素が反映されます。

TopBottomTemplate およびその MetadataState と SourceGenerator を設計するには、次の操作を実行します。

- Template を拡張する TopBottomTemplate クラスの作成。このクラスに、Template の現在の状態の取得、ユーザーによって指定された値の設定、および Template の現在の状態の設定を行うメソッドを追加します。
- MetadataState を実装する TopBottomTemplateState クラスの作成。このクラスに、Template によって作成された Source の作成に使用する SourceGenerator の値を格納するためのフィールドを提供します。値は、TopBottomTemplate のメソッドによって設定されます。
- SourceGenerator を実装する TopBottomTemplateGenerator クラスの作成。このクラスの generateSource メソッドでは、エンド・ユーザーの選択によって指定される Source を作成する操作を提供します。

エンド・ユーザーは、このアプリケーションを使用して、最初のダイアログ・ボックスで売上高をメジャーとして選択し、製品をベース・ディメンションとして選択します。エンド・ユーザーは、「Single Selections」ダイアログ・ボックスで、サンフランシスコの顧客、2000 年の第 1 四半期、直接経路および広告による宣伝を、他の各ディメンションの単一の値として選択します。

エンド・ユーザーが作成した問合せは、広告による宣伝が行われている間、2000 年の第 1 四半期にサンフランシスコの顧客に対して直接経路で販売された製品のうち、総売上高の値が最も高い 10 の製品を要求します。

TopBottomTemplate、TopBottomTemplateState および TopBottomTemplateGenerator オブジェクトの実装例、およびこれらのオブジェクトを使用するアプリケーションの例については、[例 10-1](#)、[例 10-2](#)、[例 10-3](#) および [例 10-4](#) を参照してください。

Template のクラスの実装

[例 10-1](#) に、TopBottomTemplate クラスの実装例を示します。

例 10-1 Template の実装

```
package myTestPackage;

import oracle.olapi.data.source.DataProvider;
import oracle.olapi.data.source.DynamicDefinition;
```

```
import oracle.olapi.data.source.Source;
import oracle.olapi.data.source.Template;
import oracle.olapi.transaction.metadataStateManager.MetadataState;

/**
 * Creates a TopBottomTemplateState, a TopBottomTemplateGenerator,
 * and a DynamicDefinition. Gets the current state of the
 * TopBottomTemplateState and the values it stores. Sets the data values
 * stored by the TopBottomTemplateState and sets the changed state as
 * the current state.
 */
public class TopBottomTemplate extends Template {
    public static final int TOP_BOTTOM_TYPE_TOP = 0;
    public static final int TOP_BOTTOM_TYPE_BOTTOM = 1;

    // Variable to store the DynamicDefinition.
    private DynamicDefinition _definition;

    /**
     * Creates a TopBottomTemplate with default type and number values
     * and a specified base dimension.
     */
    public TopBottomTemplate(Source base, DataProvider dataProvider) {
        super(new TopBottomTemplateState(base, TOP_BOTTOM_TYPE_TOP, 0),
              dataProvider);

        // Create the DynamicDefinition for this Template. Create the
        // TopBottomTemplateGenerator that the DynamicDefinition uses.
        _definition =
            createDynamicDefinition(new TopBottomTemplateGenerator(dataProvider));
    }

    /**
     * Gets the Source produced by the TopBottomTemplateGenerator
     * from the DynamicDefinition.
     */
    public final Source getSource() {
        return _definition.getSource();
    }

    /**
     * Gets the Source that is the base of the values in the result set.
     * Returns null if the state has no base.
     */
    public Source getBase() {
        TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
        return state.base;
    }
}
```

```

/**
 * Sets a Source as the base.
 */
public void setBase(Source base) {
    TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
    state.base = base;
    setCurrentState(state);
}

/**
 * Gets the Source that specifies the measure and the single
 * selections from the dimensions other than the base.
 */
public Source getCriterion() {
    TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
    return state.criterion;
}

/**
 * Specifies a Source that defines the measure and the single values
 * selected from the dimensions other than the base.
 * The SingleSelectionTemplate produces such a Source.
 */
public void setCriterion(Source criterion) {
    TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
    state.criterion = criterion;
    setCurrentState(state);
}

/**
 * Gets the type, which is either TOP_BOTTOM_TYPE_TOP or
 * TOP_BOTTOM_TYPE_BOTTOM.
 */
public int getTopBottomType() {
    TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
    return state.topBottomType;
}

/**
 * Sets the type.
 */
public void setTopBottomType(int topBottomType) {
    if ((topBottomType < TOP_BOTTOM_TYPE_TOP) ||
        (topBottomType > TOP_BOTTOM_TYPE_BOTTOM))
        throw new IllegalArgumentException("InvalidTopBottomType");
    TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();

```

```

        state.topBottomType = topBottomType;
        setCurrentState(state);
    }

    /**
     * Gets the number of values selected.
     */
    public float getN() {
        TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
        return state.N;
    }

    /**
     * Sets the number of values to select.
     */
    public void setN(float N) {
        TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
        state.N = N;
        setCurrentState(state);
    }
}

```

例 10-2 に、前述の TopBottomTemplateState クラスの実装例を示します。

例 10-2 MetadataState の実装

```

package myTestPackage;

import oracle.olapi.data.source.Source;
import oracle.olapi.transaction.metadataStateManager.MetadataState;

/**
 * Stores data that can be changed by its TopBottomTemplate.
 * The data is used by a TopBottomTemplateGenerator in producing
 * a Source for the TopBottomTemplate.
 */
public final class TopBottomTemplateState
    implements Cloneable, MetadataState {
    public int topBottomType;
    public float N;
    public Source criterion;
    public Source base;

    /**
     * Creates a TopBottomTemplateState.
     */
    public TopBottomTemplateState(Source base, int topBottomType, float N) {

```

```
        this.base = base;
        this.topBottomType = topBottomType;
        this.N = N;
    }

    /**
     * Creates a copy of this TopBottomTemplateState.
     */
    public final Object clone() {
        try {
            return super.clone();
        }
        catch(CloneNotSupportedException e) {
            return null;
        }
    }
}
```

例 10-3 に、前述の TopBottomTemplateGenerator クラスの実装例を示します。

例 10-3 SourceGenerator の実装

```
package myTestPackage;

import oracle.olapi.data.source.DataProvider;
import oracle.olapi.data.source.Source;
import oracle.olapi.data.source.SourceGenerator;
import java.lang.Math;

/**
 * Produces a Source for a TopBottomTemplate based on the data
 * values of a TopBottomTemplateState.
 */
public final class TopBottomTemplateGenerator
    implements SourceGenerator {
    // Store the DataProvider.
    private DataProvider _dataProvider;

    /**
     * Creates a TopBottomTemplateGenerator.
     */
    public TopBottomTemplateGenerator(DataProvider dataProvider) {
        _dataProvider = dataProvider;
    }

    /**
     * Generates a Source for a TopBottomTemplate using the current
```

```

    * state of the data values stored by the TopBottomTemplateState.
    */
    public Source generateSource(MetadataState state) {
        TopBottomTemplateState castState = (TopBottomTemplateState)state;
        if (castState.criterion == null)
            throw new NullPointerException("CriterionParameterMissing");
        Source sortedBase = null;
        if (castState.topBottomType == TOP_BOTTOM_TYPE_TOP)
            sortedBase = castState.base.sortDescending(castState.criterion);
        else
            sortedBase = castState.base.sortAscending(castState.criterion);
        return sortedBase.interval(1, Math.round(castState.N));
    }
}

```

Template を使用するアプリケーションの実装

エンド・ユーザーが行った選択を Template 用の MetadataState に格納した後、DynamicDefinition の getSource メソッドを使用して、Template によって作成された Source を取得します。この項では、例 10-1 で説明した TopBottomTemplate を使用するアプリケーションの例を示します。便宜上、このコードでは多くの例外処理を省略しています。

この例で使用されている Context クラスには、次の操作を行うメソッドが含まれています。

- Oracle OLAP への接続
- データベースのオープン
- エンド・ユーザーが選択したメジャーおよびディメンションのメタデータ・オブジェクトの取得
- メタデータ・オブジェクトからのプライマリ Source オブジェクトの取得

この例では、次の操作が行われます。

- Context からのプライマリ Source オブジェクトの取得
- メジャーのいくつかのディメンションから単一の値を選択するための SingleSelectionTemplate の作成
- TopBottomTemplate の作成およびエンド・ユーザーが行った選択の格納
- TopBottomTemplate によって作成された Source の取得
- その Source 用の Cursor の作成
- Cursor からの値の取得および表示

例 10-4 には、エンド・ユーザーと対話するためのコード、SingleSelectionTemplate、または SingleSelectionTemplate 用の MetadataState および SourceGenerator オ

プロジェクトを実装するためのコードは含まれていません。この例のクラスには、`Cursor` を作成するメソッド、および `Cursor` の値を出力するメソッドが含まれています。他のすべての操作は、`main` メソッドで実行されます。`Context` オブジェクトは、データベースへの接続、`DataProvider` と `TransactionProvider`、およびプライマリ `Source` オブジェクトを提供します。

例 10-4 Template によって作成された Source の取得

```
package myTestPackage;

import oracle.olapi.data.source.Source;
import oracle.olapi.data.source.StringSource;
import oracle.olapi.data.source.DataProvider;
import oracle.olapi.data.source.CursorManagerSpecification;
import oracle.olapi.data.cursor.CursorManager;
import oracle.olapi.data.source.SpecifiedCursorManager;
import oracle.olapi.data.cursor.Cursor;
import oracle.olapi.data.cursor.ValueCursor;
import oracle.olapi.transaction.NotCommittableException;
import myTestPackage.Context;
import myTestPackage.TopBottomTemplate;
import myTestPackage.SingleSelectionTemplate;

/**
 * Creates a query that specifies a number of values from the top or
 * bottom of a list of values from one of the dimensions of a measure.
 * The list is determined by the measure and by single values from
 * the other dimensions of the measure. Displays the results of the
 * query.
 */
public class TopBottomTest {
    /**
     * Prints the values of the Cursor.
     */
    public static void printCursor(Cursor cursor) {

        // Because the result is a single set of values with no outputs,
        // cast the Cursor to a ValueCursor and print out the values.
        ValueCursor valueCursor = (ValueCursor) cursor;
        int i = 1;
        do {
            System.out.println(i + ". " + valueCursor.getCurrentValue());
            i++;
        } while (valueCursor.next());
    }

    /**
```



```

    * Creates a Cursor.
    */
    public static void createCursor(Source choice, DataProvider dp) {
        CursorManagerSpecification cursorMgrSpec =
            dp.createCursorManagerSpecification(choice);
        SpecifiedCursorManager cursorManager =
            dp.createCursorManager(cursorMgrSpec);
        Cursor cursor = cursorManager.createCursor();
        // Print the values of the Cursor.
        printCursor(cursor);
        // Close the CursorManager.
        cursorManager.close();
    }

    public static void main(String[] args) {

        // Create a Context object and from it get the DataProvider and
        // the primary Source objects for the measure and the dimensions.
        Context context = new Context();
        DataProvider dp = context.getDataProvider();
        Source[] sources = context.getPrimarySourcesByName(
            new String[]{"SALES_AMOUNT", "PRODUCTS_DIM", "CUSTOMERS_DIM",
                "CHANNELS_DIM", "TIMES_DIM", "PROMOTIONS_DIM"});
        Source salesAmount = sources[0];
        StringSource product = (StringSource)sources[1];
        StringSource customer = (StringSource)sources[2];
        StringSource channel = (StringSource)sources[3];
        StringSource time = (StringSource)sources[4];
        StringSource promo = (StringSource)sources[5];

        // Create a SingleSelectionTemplate to produce a Source that
        // specifies a single value for each of the dimensions other
        // than the base for the selected measure.
        SingleSelectionTemplate singleSelections =
            new SingleSelectionTemplate(salesAmount, dp);
        singleSelections.addSelection((StringSource) customer,
            "San Francisco");
        singleSelections.addSelection((StringSource) time, "2000-Q1");
        // S is the direct sales channel
        singleSelections.addSelection((StringSource) channel, "S");
        singleSelections.addSelection((StringSource) promo, "billboard");

        // Create a TopBottomTemplate and set the parameters selected by
        // the end user, including a dimension as the base and the
        // Source produced by the SingleSelectionTemplate as the
        // criterion.
        TopBottomTemplate topNBottom = new TopBottomTemplate(product, dp);
    }

```

```
topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_TOP);
topNBottom.setN(15);
topNBottom.setCriterion(singleSelections.getSource());

// With methods on the TransactionProvider, prepare and commit
// the transaction.
try{
    context.getTransactionProvider().prepareCurrentTransaction();
}
catch(NotCommittableException e){
    System.out.println("Cannot prepare current Transaction. " +
        "Caught exception " + e + ".");
}
context.getTransactionProvider().commitCurrentTransaction();

// Get the Source produced by the TopBottomTemplate,
// create a Cursor for it and display the results.
createCursor(topNBottom.getSource(), dp);
}
}
```

開発環境のセットアップ

この付録では、OLAP API を使用するアプリケーションの開発環境について説明します。

この付録では、次の項目について説明します。

- 概要
- 必要なソフトウェア
- アプリケーション開発コンピュータのセットアップ
- アプリケーションのデプロイに対する考慮点

概要

OLAP オプションを指定して Oracle をインストールすると、データベースおよびそのホスト・コンピュータに必要なすべての Oracle OLAP ソフトウェアが提供されます。また、クライアント・インストール時に、アプリケーション開発コンピュータで OLAP API クライアント・アプリケーションを開発するために必要な jar ファイルが提供されます。

アプリケーション開発者は、Administrator オプションを指定してクライアント・インストールを完了し、Java アプリケーションを開発するコンピュータにこれらの jar ファイルをコピーする必要があります。また、サポートしている JDBC および Java のファイルが開発コンピュータで使用可能であることを確認する必要があります。

必要なソフトウェア

アプリケーション開発コンピュータには、次のファイルが必要です。

- OLAP API クライアント・ソフトウェアを表す OLAP API jar ファイル。Administrator オプションを指定した Oracle のクライアント・インストール時に、OLAP API Javadoc とともにこれらのファイルが提供されます。
- アプリケーションと Oracle データベース間の通信を提供する Oracle JDBC (Java Database Connectivity) jar ファイル。Administrator オプションを指定した Oracle のクライアント・インストール時に、JDBC が提供されます。JDBC の Oracle 実装の使用方法の詳細は、OTN の Web サイトを参照してください。

<http://otn.oracle.co.jp/>

他のベンダーの製品ではなく、Oracle の実装を使用する必要があります。

- Java Development Kit (JDK) バージョン 1.2。JDK は、Oracle のインストール時には提供されません。JDK の取得および使用方法の詳細は、Sun 社の Java Web サイトを参照してください。

<http://java.sun.com>

Oracle JDeveloper を開発環境として使用している場合は、JDBC および JDK はコンピュータにすでにインストールされています。ただし、JDeveloper で正しいバージョンの JDK を使用していることを確認してください。

次に OLAP API プログラムを開発する際に必要となる jar ファイル のリストおよび場所を記述します。
(OS が UNIX または Linux の場合を記述しています。MS Windows の場合は、/ を ¥ に読み替えてください。)

OLAP API および JDBC の Jar ファイル	ファイルの存在する場所
express_common.jar	ORACLE_HOME/olap/olapi/lib
express_connection.jar	ORACLE_HOME/olap/olapi/lib
express_mdm.jar	ORACLE_HOME/olap/olapi/lib
express_olapi_common.jar	ORACLE_HOME/olap/olapi/lib
express_olapi_data.jar	ORACLE_HOME/olap/olapi/lib
express_olapi_data_full.jar	ORACLE_HOME/olap/olapi/lib
express_spl.jar	ORACLE_HOME/olap/olapi/lib
collections.jar	ORACLE_HOME/BC4J/lib
classes12.jar	ORACLE_HOME/jdbc/lib
nls_charset12.jar	ORACLE_HOME/jdbc/lib

express*.jar ファイルは、OLAP API のクラスを含んでいます。collections.jar ファイルは、Java 1.2 のコレクション・フレームワークのクラスを含んでおり、OLAP API クラスから使用されることがあります。

classes12.jar ファイルは、JDK 1.2.x および 1.3.x を使用する際に必要となる JDBC クラスを含んでいます。nls_charset12.jar ファイルはオプションです。nls_charset12.jar ファイルは、オブジェクトやコレクション・タイプにおける NLS サポートを提供しています。詳細は、ORACLE_HOME/jdbc ディレクトリの Readme.txt ファイルおよび『Oracle9i 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。

アプリケーション開発コンピュータのセットアップ

jar ファイルのインストール

開発環境で jar ファイルをアクセス可能にするには、次の手順を実行します。

1. アプリケーション開発コンピュータで、ご使用のプラットフォーム用の Oracle Client Software CD を起動します。
2. Administrator のインストール・タイプを選択し、指示に従ってインストールを完了します。

3. インストール時に OLAP API jar ファイルがコピーされたコンピュータ上で、これらのファイルを検索します。Oracle ホーム・ディレクトリの `olap/olapi/lib` サブディレクトリ内を検索します。
4. OLAP API jar ファイルを、Oracle JDeveloper などの使用している Java 統合開発環境 (IDE) にアクセス可能にします。
5. Java 環境変数 `CLASSPATH` を編集して、ファイルのパスを含めます。
6. IDE では、ファイルをプログラムへのクラスのインポート用にアクセス可能にするために必要なすべての指定を行います。

OLAP API Javadoc のインストール

アプリケーション開発コンピュータの OLAP API Javadoc にアクセスする場合は、それを含む jar ファイルを、インストール時にそのファイルがコピーされたコンピュータ上で検索します。Oracle ホーム・ディレクトリの `olap/olapi/doc` サブディレクトリ内を検索します。ファイルのインストール方法および Web ブラウザでのアクセス方法の詳細は、ディレクトリ内の `readme.txt` ファイルを参照してください。

アプリケーションのデプロイに対する考慮点

アプリケーションをデプロイする際は、OLAP API を実行する各コンピュータに次のものがインストールされていることを確認します。

- OLAP API jar ファイル
- Oracle JDBC
- Java Runtime Environment (JRE)

JDBC および JRE については、インストールされているバージョンが、アプリケーションの開発時に使用したバージョンと互換性があることを確認してください。

A

alias メソッド
説明, 5-6
例, 6-14

C

CompoundCursor オブジェクト
位置, 8-14
クロス集計ビュー用のナビゲート、例, 9-10, 9-12
子の取得、例, 9-5
表ビュー用のナビゲート、例, 9-9
Connection オブジェクト
既存の Connection オブジェクトの取得例, 3-4
クローズ例, 3-5
作成例, 3-2
CursorInput クラス, 8-8, 8-10
CursorManagerSpecification クラス, 8-8
オブジェクトの作成、例, 9-2
CursorManagerUpdateEvent クラス, 8-13
CursorManagerUpdateListener クラス, 8-13
CursorManager オブジェクト
CursorManagerSpecification の更新, 8-11
Transaction のロールバック前のクローズ, 7-9
作成、例, 9-2
CursorManager クラス, 8-10
階層, 8-11
CursorSpecification オブジェクト
CursorManagerSpecification からの取得、例, 9-17
CursorSpecification クラス, 8-8
Cursor オブジェクト
Cursor を作成できない Source オブジェクト, 8-3
値の開始位置および終了位置、計算例, 9-18
値の取得、例, 9-3

位置, 8-13
エクステンツの計算、例, 9-18
エクステンツの定義, 8-22
親の開始 / 終了位置, 8-19
クロス集計ビュー用のフェッチ・サイズの指定、
例, 9-22
現行の Transaction での作成, 8-3
現在の位置、定義, 8-13
構造, 8-5
子の取得、例, 9-5
作成、例, 9-2
スパン、定義, 8-20
動作の指定, 8-7, 9-16
速く変化するコンポーネントと遅く変化するコン
ポーネント, 8-5
表ビュー用のフェッチ・サイズの指定、例, 9-22
フェッチ・サイズの定義, 8-23
フェッチ・ブロックの定義, 8-23
Cursor クラス
アーキテクチャ、メリット, 8-2
階層, 8-4
Cursor 内の現在の位置、定義, 8-13
Cursor の値のスパン
定義, 8-20, 9-18
Cursor のエクステンツ
計算例, 9-18
使用, 8-22
定義, 8-22
Cursor のフェッチ・サイズ
指定, 8-23
指定する理由, 8-23
指定例, 9-22
定義, 8-23
Cursor のフェッチ・ブロック
共有, 8-25

形式の決定, 8-24
定義, 8-23

D

DataProvider オブジェクト
 MdmMetadataProvider の作成, 4-3
 作成, 3-4
distinct メソッド
 説明, 5-6
DriverManager オブジェクト, 3-3
DynamicDefinition クラス, 10-7

E

ExpressTransactionProvider クラス, 7-8
extract メソッド、説明, 5-6

F

FundamentalMetadataObject クラス, 2-22
FundamentalMetadataProvider クラス, 2-22

G

getSource メソッド
 DynamicDefinition クラス, 10-2, 10-7
 MdmSource クラス, 2-7
 Template によって作成された Source の取得例,
 10-13
 単純事例, 4-8
 プライマリ Source オブジェクトの作成用, 5-5, 5-6

J

Java Development Kit、必要なバージョン, A-2
Javadoc, A-4
JDBC
 Connection オブジェクト, 3-3
 DriverManager オブジェクト, 3-3
 インストール, A-2
 ドライバのロード, 3-3
join メソッド
 入力から出力への切替え, 6-3
 例, 6-3 ~ 6-5, 6-14

M

MDM、「多次元メタデータ・モデル」を参照

MdmAttribute オブジェクト

 Source オブジェクトの作成, 5-6
 取得例, 4-8
 説明, 2-20
 要素, 2-21

MdmDimensionDefinition オブジェクト

 取得例, 4-8
 説明, 2-9

MdmDimensionMemberType オブジェクト

 取得例, 4-8
 説明, 2-9

MdmDimension オブジェクト

 概要, 1-6
 関連 MdmAttribute オブジェクト, 2-8
 関連 MdmDimensionDefinition オブジェクト, 2-9
 関連 MdmDimensionMemberType オブジェクト,
 2-9
 関連オブジェクトの取得例, 4-7
 説明, 2-8
 リージョン, 2-9

MdmHierarchy オブジェクト

 Source オブジェクトの作成, 6-11
 値タイプの説明, 2-11
 共用体 MdmHierarchy の要素, 2-15
 共用体タイプの説明, 2-11
 説明, 2-11
 レベル MdmHierarchy の要素, 2-12
 レベル・タイプの説明, 2-11

MdmLevel オブジェクト

 説明, 2-10
 要素, 2-10

MdmListDimension オブジェクト

 説明, 2-17
 要素, 2-17

MdmMeasure オブジェクト

 値の種類, 2-18
 概要, 1-6
 説明, 2-17
 ディメンションの取得例, 4-7
 要素, 2-18

MdmMetadataProvider オブジェクト

 概要, 1-5
 作成, 4-3
 説明, 4-3

MdmObject クラス, 2-5
MdmSchema オブジェクト

概要, 1-5
コンテンツの取得, 4-6
説明, 2-6
ルート, 2-6, 4-4
ルートの取得, 4-6

MdmSource オブジェクト, 2-7

MDM オブジェクトのタイプ

取得, 2-26
定義, 2-24

MetadataState クラス, 10-6

実装例, 10-11

O

OLAP API

アプリケーション開発用のインストール, A-2
ソフトウェア・コンポーネント, 1-7
定義, 1-2

OLAP API Boolean データ型, 2-22, 5-8

OLAP API Date データ型, 2-22, 5-8

OLAP API Double データ型, 2-22, 5-8

OLAP API Empty データ型, 2-23, 5-8

OLAP API Float データ型, 2-22, 5-8

OLAP API Integer データ型, 2-22, 5-8

OLAP API Null データ型, 5-8

OLAP API Number データ型, 2-23, 5-8

OLAP API Short データ型, 2-22, 5-8

OLAP API String データ型, 2-22, 5-8

OLAP API Value データ型, 2-23, 5-8

OLAP API Void データ型, 2-23

OLAP API データ型

MDM メタデータ・オブジェクト用, 2-22
表すオブジェクト, 5-8

OLAP メタデータ API, 2-2

OLAP メタデータ・オブジェクト, 2-2

P

position メソッド

説明, 5-6
例, 6-7

S

selectValue メソッド

例, 6-5

SourceGenerator クラス, 10-6

実装例, 10-12

Source オブジェクト

DynamicDefinition からの変更可能な Source の取得, 10-7

Transaction オブジェクトでアクティブ, 7-2, 8-3

関係, 6-13

基本, 5-4

構造, 5-5, 5-6

取得, 5-5, 5-6

セルフ・リレーション, 6-14

属性用, 5-6

定数, 5-4

導出, 5-4

範囲, 5-4

プライマリ, 5-4

変更可能, 10-2

メジャー用, 5-6

リスト, 5-4

Source クラス

基本形メソッド, 5-6, 5-7

ショートカット・メソッド, 5-7

有効な方法, 5-7

Source メソッド

alias, 5-6

distinct, 5-6

extract, 5-6

position, 5-6

value, 5-6

文字列, 6-25 ~ 6-26

SpecifiedCursorManager オブジェクト

createCursorManager メソッドによって戻される,
8-10

クローズ, 8-10

T

Template オブジェクト

概要の例, 1-9

作成に使用するクラス, 10-3

使用されている Transaction オブジェクト, 7-4

使用のメリット, 1-9

動的 Source を作成するクラス間の関連, 10-4

- 変更可能な Source オブジェクトの作成用, 10-2
- Template クラス, 10-6
 - 実装例, 10-8
 - 設計, 10-7
- TransactionProvider インタフェース, 7-8
- TransactionProvider オブジェクト
 - MdmMetadataProvider の作成, 4-3
 - 作成, 3-3
- Transaction オブジェクト
 - Template クラスでの使用, 7-4
 - 書込み, 7-2
 - 現行, 7-2
 - 現行の Transaction での Cursor の作成, 8-3
 - 現行の取得, 7-8
 - 現行の設定, 7-8
 - 子の使用例, 7-9
 - 子の読込みおよび書込み, 7-2
 - コミット, 7-3
 - 準備, 7-3
 - 読込み, 7-2
 - ロールバック, 7-7

V

- ValueCursor オブジェクト
 - 値の取得、例, 9-3, 9-4
 - 位置, 8-14
 - 親 CompoundCursor からの取得、例, 9-5
- value メソッド, 5-6

あ

- 値 MdmHierarchy, 2-11
- アプリケーション
 - 開発手順, 1-7
 - 実行されるタスク, 1-11
 - デプロイ, 1-10
- アプリケーション開発用のインストール, A-2

い

- 位置
 - CompoundCursor, 8-14
 - Cursor, 8-13
 - ValueCursor, 8-14
 - 親の開始 / 終了, 8-19
 - 要素, 6-7

う

- 売上履歴スキーマ
 - メタデータ・オブジェクトのリスト, 4-15
 - メタデータ検出プログラム, 4-9

お

- 遅く変化する Cursor コンポーネント, 8-5, 8-18
- 親子関係
 - Source オブジェクトの作成, 6-12
 - 階層, 2-3, 2-8, 2-10, 2-11, 2-15
- 親属性
 - MdmHierarchy オブジェクト, 2-11
 - MdmLevel オブジェクト, 2-10
 - 取得例, 4-8

か

- 階層
 - MdmHierarchy オブジェクト, 2-11
 - OLAP メタデータ, 2-3
 - Source オブジェクトの作成, 6-11
 - 定義, 1-2
 - デフォルトの取得, 4-8, 6-12
 - ドリルダウン, 6-13
 - ノードおよびリーフ, 2-15
- 階層内のノード, 2-15
- 階層内のリーフ, 2-15
- 階層のドリルダウン, 6-13
- 書込み Transaction オブジェクト, 7-2
- 仮想 Cursor
 - 定義, 8-23
- 関係
 - Source オブジェクト, 6-13

き

- 基本 Source オブジェクト
 - 作成方法, 5-8
 - 定義, 5-4
- 基本形メソッド, 5-6, 5-7
- 共用体 MdmHierarchy, 2-11

く

クロス集計ビュー

Cursor のナビゲート、例、9-10、9-12

さ

サブスキーマ

コンテンツの取得、4-6

説明、4-4

し

集計関数、作成、6-24

集計メソッド

使用、6-21、6-22

説明、6-20

リスト、6-21

出力

CompoundCursor、8-5、8-20、8-22

位置、8-15

CompoundCursorSpecification からの取得、例、9-17

CompoundCursor からの取得、例、9-5

入力からの切替え、6-3 ～ 6-5

ネスト、例、9-6

す

数値関数

作成、6-23

数値操作

実行、6-16 ～ 6-18

メソッドのリスト、6-16、6-18

例、6-17

数値比較

実行、6-18 ～ 6-19

数値メソッド

使用、6-19 ～ 6-24

せ

接続

確立の手順、3-2

既存の接続の取得、3-4

クローズ、3-5

前提条件、3-2

セルフ・リレーション

Source オブジェクト、6-14

そ

属性

MdmAttribute オブジェクト、2-20

OLAP メタデータ、2-3

Source オブジェクト、5-6

親、2-10、2-11

祖先、2-10、2-11

定義、1-2

リージョン、2-11

祖先属性

MdmHierarchy オブジェクト、2-11

MdmLevel オブジェクト、2-10

取得例、4-8

た

多次元メタデータ・モデル (MDM)

概要、1-5

説明、2-2

タプル

Cursor、例、9-7

定義、8-15

て

提供されるソフトウェア、A-2

定数 Source オブジェクト

定義、5-4

例、5-7

ディメンション

MdmDimension オブジェクト、2-8

OLAP メタデータ、2-3

Source オブジェクト、5-5

定義、1-2

データ・ウェアハウス、1-3

データ型

MdmSource オブジェクト、2-23

MDM メタデータ・オブジェクト、2-22

データ・ストア

調査、4-2

定義、1-3

有効範囲、4-2

デフォルトの階層

取得, 6-12

取得例, 4-8

と

問合せ

結果の取得手順, 9-2

問合せではない Source オブジェクト, 8-3

動的, 10-2

導出 Source オブジェクト

概要, 5-7

説明, 5-7

定義, 5-4

動的問合せ, 10-2

ドキュメント, A-4

に

入出力の順序

Source の構造に対する影響, 6-3, 6-4

決定, 6-3, 6-4

入力

出力への切替え, 6-3 ~ 6-5

ね

ネストした出力

Cursor からの値の取得、例, 9-6

Source、定義, 9-3

は

速く変化する Cursor コンポーネント, 8-5

パラメータ

作成, 6-23

パラメータ化された選択

作成, 6-23

範囲 Source オブジェクト, 5-4

ひ

非対称の結果セット、Cursor の位置, 8-18

表ビュー

Cursor のナビゲート、例, 9-9

ふ

プライマリ Source オブジェクト

getSource メソッドの結果, 4-8

MdmHierarchy オブジェクト, 6-11

MdmSource オブジェクト, 2-7

親子関係, 6-12

構造, 5-5, 5-6

取得, 5-5 ~ 5-6

定義, 5-4

め

メジャー

MdmMeasure オブジェクト, 2-17

OLAP メタデータ, 2-3

Source オブジェクト, 5-6

定義, 1-2

メジャー MdmDimension オブジェクト, 4-6

メジャー・フォルダ

MdmSchema オブジェクトへのマッピング, 2-6

OLAP メタデータ, 2-3

メタデータ

MDM メタデータへの OLAP メタデータのマッピング, 2-6

OLAP API に対する準備, 1-3, 2-2

検出, 4-2

検出用のサンプル・コード, 4-9 ~ 4-24

定義, 1-3

データとの区別, 1-4

も

文字列メソッド, 6-25 ~ 6-26

よ

要素

MdmAttribute, 2-21

MdmLevel, 2-10

MdmListDimension, 2-17

MdmMeasure, 2-18

値を使用した選択, 6-2

共用体 MdmHierarchy, 2-15

ソート, 6-8, 6-11

ランキング, 6-8 ~ 6-11

レベル MdmHierarchy, 2-12

要素の選択

階層での位置付け, 6-11 ~ 6-13

要素値, 6-2 ~ 6-5

ランク, 6-5 ~ 6-11

要素のソート, 6-8, 6-11

要素のランキング, 6-8 ~ 6-11

読み込み Transaction オブジェクト, 7-2

り

リージョン

MdmDimension, 2-9

取得例, 4-8

リージョン属性

MdmHierarchy オブジェクト, 2-11

取得例, 4-8

リスト Source オブジェクト, 5-4

る

ルート MdmSchema

機能, 4-4

取得, 4-6

説明, 2-6

れ

レベル

MdmLevel オブジェクト, 2-10

OLAP メタデータ, 2-3

定義, 1-2

レベル MdmHierarchy, 2-11

