持続プログラミング

Forte™ for Java™ プログラミングシリーズ

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900 U.S.A.
650-960-1300

Part No. 816-2850-01
2001 年 10 月 , Revision A

原典 *:* *Programming Persistence*
Part No: 814-1411-10
Revision A

目次

図目次

表目次

はじめに

Forte™ for Java™ プログラミングシリーズの『持続プログラミング』にようこそ。本書は、Forte™ for Java™ プログラミングマニュアルの一冊で、持続データ (アプリケーションの外部のデータベースやデータストアに格納されたデータ) の操作について解説しています。また、Forte for Java でサポートされるさまざまな持続性プログラミングモデルについても説明しています。本書は、Forte for Java 統合開発環境 (IDE) が提供する透過的な持続性 (Transparent Persistence) 技術に焦点を置いています。

本書は、Forte for Java が対応している持続性プログラミングモデルの使用法を習得したいプログラマを対象としています。本書は、Java およびデータベースアクセス技術についての一般的な知識があることを前提としています。本書をお読みになる前に、次のことを理解しておく必要があります。

- Java プログラミング言語
- リレーショナルデータベースの概念 (表、キーなど)
- データベースの使用法

本書に出てくる例は、次のプラットフォームとオペレーティングシステムで作成できます。

- Solaris™ 8 (SPARC™ プラットフォーム版)
- Microsoft Windows 2000, SP2
- Microsoft Windows NT 4.0, SP6
- Red Hat Linux 6.2

本書に掲載されているすべてのスクリーンショットは、Windows 版の Forte for Java ソフトウェアからのものです。他のプラットフォームを使用する場合でも表示上の違いはわずかであるため、内容を理解するのに問題はありません。ほとんどすべての手順において Forte for Java ユーザーインタフェースを使用していますが、コマンド行で

コマンドを入力するよう指示される場合もあります。そのような場合、Microsoft Windows の「コマンドプロンプト」ウィンドウでのプロンプトと構文が例に使用されています。

```
c:¥>cd MyWorkDir¥MyPackage
```

この例を UNIX® または Linux 環境に読み変えるには、プロンプトを変更し、スラッシュ (/) を使用します。

```
% cd MyWorkDir/MyPackage
```

お読みになる前に

本書は、Forte for Java が対応している持続性プログラミングモデルの使用法を習得したいプログラマを対象としています。本書は、Java およびデータベースアクセス技術についての一般的な知識があることを前提としています。本書をお読みになる前に、次のことを理解しておく必要があります。

- Java プログラミング言語
- リレーショナルデータベースの概念 (表、キーなど)
- データベースの使用法

本書の構成

各章の概要を次の表に示します。

第 1 章 では、持続性 (persistence) について、さらに、後続の章で Forte for Java の持続性機能を解説するのに必要な基礎知識について説明しています。Forte for Java IDE が対応している各種の持続性プログラミングモデルも紹介しています。

第 2 章 では、Forte for Java が提供している JDBC™ 生産性向上ツールについて説明しています。このツールを使用すると、データベースを操作するクライアントコンポーネントやアプリケーションを作成するための JDBC プログラミング作業の大部分を自動化することができます。

第 3 章 では、透過的な持続性プログラミングモデルの概要について説明しています。

第 4 章 では、透過的な持続性マッピングツールについて説明しています。さらに、このツールを使用して Java プログラミング言語のクラスとリレーショナルデータベースとをマップする方法についても説明しています。

第 5 章 では、透過的な持続性実行環境と、この環境での持続性オペレーションの実行方法を説明しています。透過的な持続プログラミングについての各種の問題についても取り扱っています。

第 6 章 では、Enterprise Java Beans、J2EE リファレンス実装、および iPlanet アプリケーションサーバーとともに持続可能クラスを使用するプロセスについて説明しています。

付録 A では、Forte for Java IDE で透過的な持続性を使用するのに必要なシステム要件について説明しています。

付録 B では、透過的な持続性機能を実行する 2 つの JSP™ タグについて説明しています。

付録 C では、サポートされない機能、特定のデータベースが透過的な持続性で独自に動作する領域、および旧バージョンの透過的な持続性でクラスを作成した開発者向けのファイル移行情報について詳述しています。

表記上の規則

表 P-1　表記上の規則

字体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 system% You have mail.
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	system% **su** Password:
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには rm *filename* と入力します。
『 』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。
「 」	参照する章、節、ボタンやメニュー名を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。	sun% **grep '^#define ** **XV_VERSION_STRING'**

関連マニュアル

Forte for Java のマニュアルには、Acrobat Reader (PDF) 形式、オンラインヘルプ、サンプルアプリケーションの Readme ファイル、Javadoc™ マニュアルなどがあります。

オンラインで入手可能なマニュアル

次のマニュアルは、Forte for Java のポータルサイト、docs.sun.com の Web サイト、およびインターネットオンラインブックストアの Fatbrain.com から入手することができます。

Forte for Java ポータルサイトでのマニュアルの入手先は http://www.sun.co.jp/forte/ffj/documentation/index.html です。docs.sun.com[SM] web サイトは、http://docs.sun.com です。Fatbrain.com のアドレスは、http://www.fatbrain.com/documentation/sun です。

- リリースノート (HTML 形式)

  Forte for Java の Edition ごとに用意されています。このリリースでの変更情報と技術上の注意事項を説明しています。

- インストールガイド (PDF 形式)

  Forte for Java の Edition ごとに用意されています。対応プラットフォームへの Forte for Java のインストール手順を説明しています。さらに、システム要件、アップグレード方法、Web サーバーやアプリケーションサーバーのインストール、コマンド行での操作、インストールされるサブディレクトリ、Javadoc の設定、データベースの統合、アップデートセンターの使用方法などが含まれます。

- Forte for Java プログラミングシリーズ (PDF 形式)

  さまざまな Forte for Java 機能を使用して優れた J2EE アプリケーションを開発するための詳細な情報が記載されています。

  - 『Web コンポーネントのプログラミング』 - Part No. : 816-2849-01

    JSP ページ、サーブレット、タグライブラリを使用し、クラスとファイルをサポートする J2EE Web モジュールとして Web アプリケーションを構築する方法について説明しています。

  - 『持続プログラミング』 - Part No. : 816-2850-01

    Forte for Java が提供するさまざまな持続性プログラミングモデルのサポート機能について説明しています。特に JDBC と透過的な持続性について詳しく解説します。

  - 『Enterprise JavaBeans コンポーネントのプログラミング』 - Part No. : 816-2845-01

Forte for Java EJB Builder ウィザードとその他のグラフィカルユーザーインタフェースを使用して、Enterprise JavaBeans コンポーネント (コンテナ管理または bean 管理の持続性を伴うセッション bean およびエンティティ bean) を作成する方法について説明しています。

- 『Web サービスのプログラミング』 - Part No. : 816-2844-01

  Web サービスモジュールが提供するツールを使用して Web サービスを構築する方法について説明しています。Web サービスは、Extensible Markup Language (XML) ドキュメントとして発行されるアプリケーションビジネスサービスであり、HTTP 接続を介して配信されます。

- 『XML データサービス用 JSP のプログラミング』 - Part No. : 816-2843-01

  Forte for Java Enterprise Service Presentation Toolkit を使用して動的な XML データを HTML に埋め込む方法について説明しています。

- 『J2EE モジュールおよびアプリケーションのアセンブリと実行』 - Part No. : 816-2846-01

  EJB モジュールと Web モジュールから J2EE アプリケーションを作成する方法、および J2EE アプリケーションを配備して実行する方法について説明しています。

- Forte for Java チュートリアル (PDF 形式)

  チュートリアルアプリケーションは、ユーザー設定ディレクトリの下の `sampledir/tutorial` ディレクトリにあります。

  - 『Forte for Java, Community Edition チュートリアル』 Part No. : 816-2847-01

    Forte for Java, Community Edition のツールを使用し、簡単な J2EE Web アプリケーションを作成する方法を順を追って説明しています。

  - 『Forte for Java, Enterprise Edition チュートリアル』 Part No. : 816-2848-01

    Enterprise JavaBeans コンポーネント、アプリケーションテスト機能、Forte for Java Web サービス技術を使用し、アプリケーションを作成する方法を順を追って説明しています。

オンラインヘルプ

オンラインヘルプは、Forte for Java 開発環境内から参照できます。ヘルプキー (Solaris オペレーティング環境では Help キー、Windows および Linux 環境では F1 キー) を押すか、「ヘルプ」 > 「内容」を選択します。ヘルプの項目と検索機能が表示されます。

プログラム例

Forte for Java の機能を紹介したプログラム例が、関連する `Readme` ファイルとともに、ユーザー設定ディレクトリの `sampledir/examples` ディレクトリに置かれています。また、Forte for Java のポータルサイトから、Enterprise Edition に固有のサンプルファイルをダウンロードし、それらを `sampledir/examples` ディレクトリに置くこともできます。チュートリアルアプリケーション (『Forte for Java, Community Edition チュートリアル』と『Forte for Java, Enterprise Edition チュートリアル』で説明されているアプリケーションを含む) はすべて、`sampledir/tutorial` ディレクトリに置かれています。

Javadoc

Javadoc 形式のマニュアルは、Forte for Java の多くのモジュールに用意されており、IDE の中で参照できます。このマニュアルの使用方法については、リリースノートを参照してください。IDE を起動すると、エクスプローラの Javadoc タブで Javadoc マニュアルを参照できます。

Sun のマニュアルのオンラインでの提供

Sun の各種システムのマニュアルを、次の Web サイトで提供しています。

`http://www.sun.com/products-n-solutions/hardware/docs`

Solaris のマニュアルセットとその他の多くのマニュアルを、次の Web サイトで提供しています。

`http://docs.sun.com`

Sun のマニュアルの注文方法

Sun の製品マニュアルは、Fatbrain.com インターネットブックストアを通じて米国 Sun Microsystems, Inc. に直接注文できます。Fatbrain.com の Sun Documentation Center へは次の URL でアクセスできます。

`http://www.fatbrain.com/documentation/sun`

ご意見の送付先

Sun のマニュアルについてのご意見やご要望をお寄せください。今後のマニュアル作成の参考にさせていただきます。次のアドレスまで電子メールをお送りください。

`docfeedback@sun.com`

電子メールのタイトルに、対象マニュアルの Part No.（このマニュアルの場合は 816-2850-01）を明記してください。

第1章

持続プログラミングの概要

この章では、持続性 (persistence) について、さらに、後続の章で Forte for Java の持続性機能を解説するのに必要な基礎知識について説明しています。Forte for Java が対応している各種の持続性プログラミングモデルも紹介しています。

持続性について

ほとんどのビジネスアプリケーションでは、**持続データ**、すなわちアプリケーションの外部で長期間保存されるデータをプログラムで操作する必要があります。持続データは、一時メモリーに読み込まれ、使用・更新されますが、保存の際にはリレーショナルデータベースやファイルシステムに書き出されます。

持続データの表現

オブジェクト指向のプログラミングシステムでは、メモリーに読み込んだ持続データを、アプリケーションコードから操作するデータオブジェクトとして表現します。一般に、データストアでの持続データと、メモリー上の持続データオブジェクトとの対応付けは、多数のソフトウェア層を介して実現されます。この様子を 図 1-1 に示します。

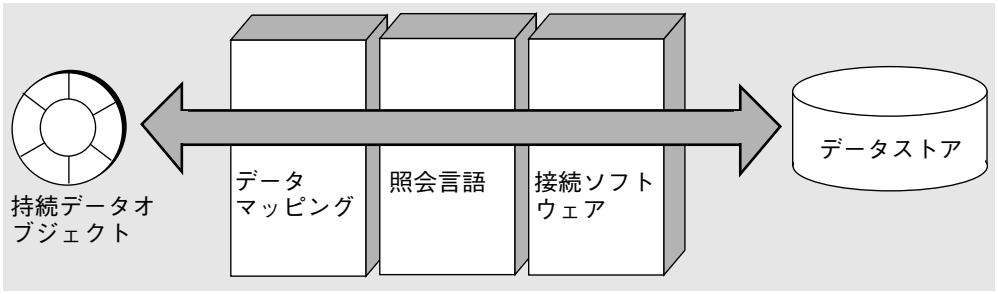

図 1-1　基本的な持続性スキーム

それぞれのデータストアは、ドライバソフトウェアを介して外部とデータをやり取りします。このドライバソフトウェアは、データストアとアプリケーションとの間の接続を設定および維持します。ドライバソフトウェアによって接続が確立されると、照会言語を使用してデータストア内の情報を抽出し、アプリケーションに読み込んだり、アプリケーションからデータストアにデータを書き出したりできます。このほかに、メモリー上のデータオブジェクトとデータストア内の情報をマップするためのソフトウェア層もあります。この一般的なスキームでは、プログラマはアプリケーションから使用および操作する実行時オブジェクトとして持続データを表現することができます。

このスキームは、基本的な持続性オペレーション、すなわち持続データの作成 (データストアへの挿入)、検索 (データストアからの選択)、更新、削除にすべて対応しています。これらの基本操作を、頭文字をとって CRUD と呼びます。

- 作成 (Create)
- 検索 (Retrieve)
- 更新 (Update)
- 削除 (Delete)

アプリケーションの問題

アプリケーションのプログラミングでは、メモリー上のデータオブジェクトとデータストア内の情報との間で、多数の問題を考慮する必要があります。たとえば、**同期化**、**並行性**、**接続資源** などです。

- 同期化

アプリケーションは、(メモリー上とデータストア内の) 2 種類のデータ表現の同期を保つ必要があります。たとえば、持続データオブジェクトを変更できるのは、それに対応する変更をデータストアでも実行できる場合に限られます。データストアへの書

き込みでエラーが発生する可能性もあるため、これらの変更は単一のトランザクションの中で行う必要があります。「トランザクション」とは、一連の操作をまとめたもので、それぞれの操作がすべて成功した場合だけコミットが行われます。エラーが発生した場合は、すべての変更がトランザクションの実行前の状態にロールバックされます。

- 並行性

アプリケーションは、複数のユーザーが持続データに同時にアクセスしても、データが破壊されないようにする必要があります。そのためには、あるユーザーがデータに対して行なった変更を、他のユーザーに適切に認識させる必要があります。

- 接続資源

アプリケーションのユーザー数が増えるにつれて、データストアとの多数の接続を作成および維持するための資源が不足しがちになります。接続管理 / 接続プールスキームを使用して、これらの資源を共有または再利用した方がはるかに効率的です。

同期化、並行性、接続資源の問題は、アプリケーションが大規模になり、複雑になるほど重要になります。少数のクライアントが単一コンピュータの単一データベースにアクセスするアプリケーションでは、同期化、並行性、および接続資源の要件を容易に満たすことができます。しかし、クライアント、データベース、トランザクションの数が増えるにつれて、これらの問題は、解決が困難になっていきます。

Java のデータベースプログラミングモデル

Java 開発環境では、持続データオブジェクトとデータストア間の操作がある程度標準化されています。データベースベンダーのほとんどが、Java 実行環境 (Java Virtual Machine) 用のドライバを提供しています。また、持続性 (CRUD) オペレーションの実行には、標準化された照会言語 (SQL) を使用するのが一般的です。

ただし、持続性オペレーションのプログラミングに対応したモデルはいくつもあり、これらのモデルごとに別々の持続性 API が使用されます。Forte for Java が対応しているプログラミングモデルは次のとおりです。

- JDBC (Java Database Connectivity)
- 透過的な持続性 (Transparent Persistence)

ここからは、これらのプログラミングモデルの概要について解説します。

JDBC (Java Database Connectivity)

持続性オペレーションのコーディングを簡単にするため、Java には標準の持続性プログラミングモデルとして JDBC API が用意されています。JDBC API は、基本的な持続性オペレーションを実行する Java インタフェースの集まりです。Forte for Java には、JDBC API に基づいたツールとプログラミング機能があります。詳細については、第 2 章 を参照してください。

JDBC プログラミングモデル

JDBC プログラミングモデルは、図 1-1 に示したソフトウェア層の構成をほぼそのまま受け継いでいます。プログラマは、まず持続データを表現するクラスを作成し、クラスの個々のフィールドを、1 つ以上のデータベースシステム内の表の列とデータ型にマップするコードを書きます。この作業を行うことで、そのクラスのインスタンス (持続データオブジェクト) を作成し、そのフィールドにデータベースから値を取り込んだり、もしくは新しいインスタンスを作成し、そのフィールドに値を代入し、その値をデータベースに書き込んだりできるようになります。

JDBC の持続性オペレーションで使用する実行時オブジェクトを 図 1-2 に示します。これらのオブジェクトは、JDBC API にインタフェースを実装するクラスのインスタンスです。これらのオブジェクトは、持続性オペレーションを実行する持続性認識コンポーネント (図 1-2) のコードによって参照されます。

たとえば、持続データオブジェクトにデータを読み込むには、次のようにします。

- DriverManager オブジェクトから接続 (データベースとの接続) を取得します。
- この接続オブジェクトから文(照会文) を取得します。
- この文に、選択 (Select) 照会を表す SQL 文字列を渡します。

  この文は接続を介して実行され、その結果としてデータベースから ResultSet (結果セット) が返されます。
- 返された ResultSet からデータ値を抽出し、持続データオブジェクトのフィールドに代入します。

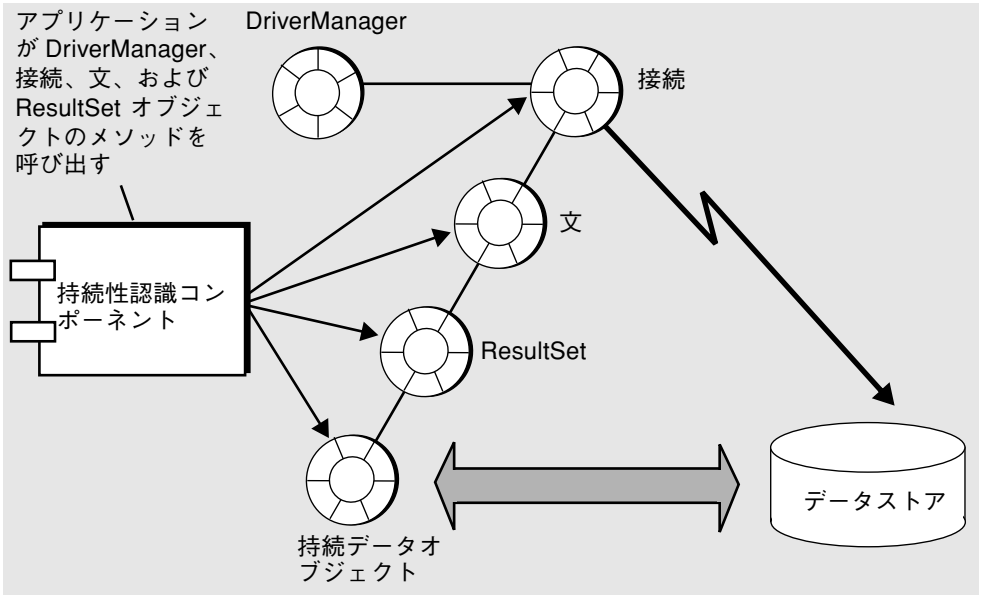

図 1-2　JDBC プログラミングモデル

同様に、SQL の更新文を使用して、持続データオブジェクトの値をデータベースに書き込むこともできます。文や接続の使用を終えたら、JDBC API に含まれているメソッドを使用し、それらを閉じます。

JDBC に準拠したドライバはマルチスレッド型で、複数のスレッドからの同時接続に対応しています。また、それぞれの JDBC 接続は、複数の照会文の同時実行に対応しています。

単純な Java アプリケーションでは、それぞれのクライアントスレッドが接続を明示的に要求し、この接続を介して照会文を実行します。より高度なアプリケーションでは、サーバーコンポーネントが接続を 1 つだけ要求し、この接続を使用して、複数のクライアントスレッドの照会文を同時に実行することができます。(サーバーコンポーネントがスレッドごとに別々の接続を要求することもできます。ただし、個々の接続を初期化するためのオーバーヘッドが増大する可能性があります。)

通常は、それぞれの照会文の実行後に、変更が自動的にコミットされます。ただし、複数のスレッドの照会文を同時に実行している場合は、自動コミット機能を無効にし、接続クラスで定義されたコミット用およびロールバック用のメソッドを使用して、トランザクションを明示的にコミットまたはロールバックすることができます。同じ接続を介して実行される照会文は、すべて同じトランザクション空間に属します。すなわち、これらの照会文はまとめてコミットまたはロールバックされます。た

とえば、論理的に別々の 2 つのトランザクションの照会文を、同じ接続を介して同時に実行した場合を考えてみましょう。その場合は、一方のトランザクションがコミットまたはロールバックされると、もう一方のトランザクションもコミットまたはロールバックされてしまいます。

そのため、マルチスレッド型のデータベースアクセスを安全に実行するには、処理性能を犠牲にして、個々のトランザクションが必要とするたびに接続を開き、不要になったら接続を閉じるか、もしくは JDBC 接続マネージャインタフェースを使用して接続プールを管理し、複数のトランザクションが接続プールを共有できるようにする必要があります。

透過的な持続性

JDBC プログラミングモデルの移植性、同期化、並行性の制約を解消するために、Forte for Java には別のプログラミングモデルが用意されています。このモデルを透過的な持続性と呼びます。透過的な持続性は、JDBC の制約を受けないだけでなく、持続性オペレーションの自動化と管理にも対応し、一般に JDBC を使用する場合よりコード化が簡単です。

- 自動化

  透過的な持続性では、持続データオブジェクトとデータストア内の情報とのマッピングが自動化され、データベースの照会や更新のためのコードも自動的に生成されます。この自動化に使用される透過的な持続性ツールは、さまざまなデータストアに対応しているため、各種のデータベースシステムの間でロジックを移植することができます。アプリケーション内部の持続性ロジックをプログラマが意識する必要もありません。

- 持続性管理

  透過的な持続性には、持続性オペレーションを管理する実行環境も用意されています。この実行環境では、持続性オペレーションが透過的に実行され (マッピングのためのコードを記述したり、照会文や更新文を作成する必要はありません)、さらにトランザクション、並行性、接続プールの管理サービスが提供されます。

ここからは、透過的な持続性プログラミングモデルの概要について説明します。Forte for Java の透過的な持続性機能や、透過的な持続性プログラミングモデルの詳細については、第 3 章～第 6 章を参照してください。

透過的な持続性プログラミングモデル

透過的な持続性プログラミングモデルでは、JDBC とは違って、図 1-1 に示したソフトウェア層のほとんどが自動化されています。以下の操作は必要ありません。

- データストアとの接続を明示的に取得
- SQL 文の作成・実行
- マッピングコードの記述

Forte for Java の透過的な持続性機能を使用すると、SQL、JDBC API、データベースプログラミングの知識がなくても、JDBC に準拠したデータベースに格納された持続データを Java オブジェクトとして参照および操作することができます。透過的な持続性ツールを使用すると、**持続可能クラス**を作成できます。持続可能クラスは、持続データを表現したクラスで、このクラスに対する持続性オペレーションは、透過的な持続性実行時システムによって自動的に実行および管理されます。

持続可能クラスを作成するには、Forte for Java の透過的な持続性ツールを使用し、データベーススキーマからクラス定義を生成するか、既存のクラスをデータベーススキーマにマップします。透過的な持続性ツールは、実行環境がデータストアに固有の文を動的に生成するために、これらのクラスの**拡張**も行います。これにより、透過的な持続性の持続性マネージャが特定のデータストアに合わせて実行時に生成した照会文を使用し、持続可能クラスに対応付けられたデータベースに対する持続性オペレーションを実行することができます。

透過的な持続性の持続性オペレーションで使用する実行時オブジェクトを 図 1-3 に示します。これらのオブジェクトは、透過的な持続性 API のインタフェースを実装するクラスのインスタンスです。これらのオブジェクトは、透過的な持続性実行環境との対話により持続性オペレーションを行う持続性認識コンポーネント (図 1-3) のコードによって参照されます。

たとえば、持続可能クラスのインスタンスにデータを読み込むには、まず持続マネージャファクトリオブジェクトから 持続性マネージャを取得し、次にこの 持続性マネージャから照会を取得し、その照会にパラメータを渡して実行します。この場合は、透過的な持続性実行時システムによって持続可能クラスのインスタンスが作成され、それらのインスタンスに照会の結果が格納されます。

同様に、持続性マネージャの `makePersistent` メソッドを呼び出して、持続可能クラスのインスタンスの値をデータベースに書き込むこともできます。これらの処理に必要な接続は、持続性マネージャが管理します。持続性マネージャは、(持続可能クラスの定義に基づいて)データストア固有の照会文を適切に生成し、それらをデータストアに送出して実行します。

透過的な持続性データベースへのデータの書き込みは、トランザクションのコンテキスト内で実行する必要があります。そのためには、持続性マネージャからトランザクションオブジェクトを取得します。このオブジェクトを使用してトランザクションを開始し、トランザクションをコミットまたはロールバックします。トランザクションを開始してからコミットするまでの間に、持続インスタンスに対して行なったデータ操作すべてが、同じトランザクションの一部になります。このようにすることで、トランザクションの全面的な制御が可能になります。

それぞれの持続性マネージャは、一度に 1 つのトランザクションしか処理できません。そのため、通常はトランザクションを実行するスレッドごとに、持続性マネージャを取得します。ただし、透過的な持続性実行時システムは、後述するように並行性管理と接続プールの両方に対応しているため、スケーラビリティは損なわれません。

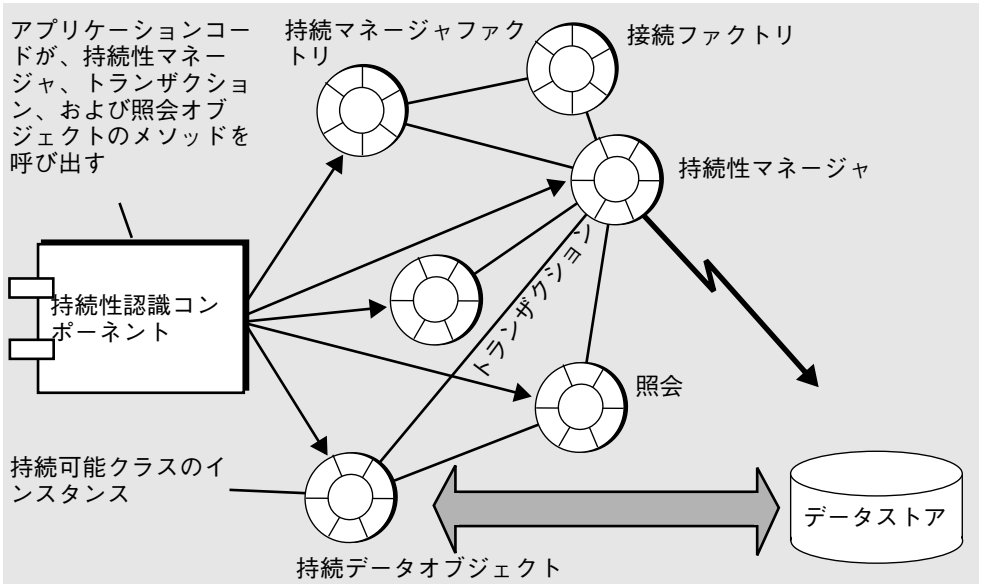

図 1-3　透過的な持続性プログラミングモデル

透過的な持続性プログラミングモデルでは、持続マネージャファクトリと、それに対応する接続ファクトリによって並行性と接続が管理されます。プログラマは、使用するデータストアとそのログイン情報に合わせて持続マネージャファクトリを設定します。さらに、持続性マネージャのインスタンスごとに、透過的な持続性実行時システムでサポートする並行性制御方式や接続管理方式といった、各種のプロパティを設定することができます。

- 並行性

  データストア並行性とオプティミスティック並行性のどちらかを選択することができます。データストア並行性とは、接続先のデータベースに用意されたロック機能を使用し、トランザクションの実行中にデータベースをロックする手法です。また、オプティミスティック並行性とは、複数のスレッドによるデータベースの読み取りを許可し、データベース行に書き込みを行う前に、その行が変更されていないことを確認する手法です。複数のユーザーが同じデータにアクセスし、読み取られたデータがすぐには(ユーザーによる更新操作が完了するまでは)更新されない場合は、一般にオプティミスティック並行性の方が処理性能が高くなります。

- 接続管理

  持続マネージャファクトリを設定することで、接続プールを管理することができます。接続プールでは、それぞれの接続を持続性マネージャの複数のインスタンスで共有し、再使用することができるため、接続資源の使用効率が高まります。多数のスレッドから同じデータベースにアクセスする場合は、接続プールを使用した方が処理性能が高くなります。

第2章

JDBC (Java Data Base Connectivity) の使用

Forte for Java には、JDBC (Java Database Connectivity) モジュールが含まれています。このモジュールを使用すると、データベースを操作するクライアントコンポーネントやアプリケーションを作成するためのプログラミング作業の大部分を自動化することができます。

Forte for Java の JDBC モジュールの目的は、JDBC を使用してデータベース表の検索と更新を行う、Swing (Java Foundation Class) コンポーネントを含む画像フォームのプログラミングの生産性を向上させることです。これにより、2 層アプリケーションアーキテクチャを簡単に作成できます。

この章では、Forte for Java が提供している次の JDBC 生産性向上ツールについて説明しています。その前に、JDBC アプリケーションの大まかな作成手順を紹介します。次のツールがあります。

- データベースエクスプローラ
- JDBC JavaBeans コンポーネント
- JDBC フォームウィザード

JDBC プログラミング

ここでは、JDBC プログラミング作業の手順を簡単に紹介します。これは、4 ページの「JDBC プログラミングモデル」 の内容を補足するものです。

一般的なプログラミング手順

JDBC プログラミングの一般的な手順は次のとおりです。

1. コードにクラスをインポートします。
2. JDBC ドライバをロードします。
3. データベースとの接続を確立します。
4. main メソッドを作成します。
5. try ... catch ブロックを作成し、例外と警告をキャッチします。
6. データベース表を設定し、使用します。
    a. 表を作成します。
    b. JDBC 文を作成します。
    c. 作成した文を実行し、持続性オペレーションを行います。
        i. 表にデータを書き込みます。
        ii. 表からデータを取り出します。
        iii. 更新可能な結果セット (`RowSet`) を作成します。
        iv. 行をプログラムで挿入したり削除します。
    d. トランザクション遮断レベルを管理して、`ResultSet` の変更内容を参照します。

Forte for Java では、これらの手順のほとんどが簡略化されます。Forte for Java の JDBC JavaBeans コンポーネントのプロパティを編集するか、「JDBC フォームウィザード」を使用するだけで、JDBC コードを作成することができます。

JDBC の参考文書

この章では、JDBC プログラミングモデルの基礎知識のある方を対象に、Forte for Java IDE の観点から JDBC プログラミングを説明しています。JDBC の詳しい情報については、次の文書を参照してください。

JDBC プログラミングの習得

「Java Developer Connection」に、JDBC についてのチュートリアルが収録されています。

```
http://developer.java.sun.com/developer/onlineTraining/new2java/programming/learn/jdbc.html
```

「Java Developer Connection」には、JDBC の速習コースも用意されています。

```
http://developer.java.sun.com/developer/onlineTraining/Database/JDBCShortCourse/index.html
```

技術文献

Sun は、次のようなドキュメントを作成しました。

『Duke's Bakery – A JDBC Order Entry Prototype – Part I:』

```
http://developer.java.sun.com/developer/technicalArticles/Database/dukesbakery/
```

JDBC の入門書

JDBC を使用したプログラミングにはじめて着手する際には、次の Web ページが役立ちます。

```
http://developer.java.sun.com/developer/technicalArticles/Interviews/StartJDBC/index.html
```

次の Web ページ「Of Java, Databases, and Really Cool Dead Guys」も参照してください。

```
http://developer.java.sun.com/developer/technicalArticles/Interviews/Databases/index.html
```

JDBC の基礎知識

JDBC の補足情報については、サンのチュートリアルを参照してください。

```
http://java.sun.com/docs/books/tutorial/index.html
```

次のチュートリアルも参考になります。

http://java.sun.com/docs/books/tutorial/jdbc/basics/index.html

データベースエクスプローラの使用

JDBC コードを作成する前に、アプリケーションで使用するデータベースについて理解する必要があります。そのためには、Forte for Java のデータベースエクスプローラを使用します。

Forte for Java のデータベースエクスプローラでは、次の作業を行うことができます。

- データベース構造の表示
- データベース中の表 (列と索引情報を含む) の確認
- データベースに関する SQL ビューの確認
- データベースで定義されているストアドプロシージャーの確認
- データベースデータの表示
- 表の作成
- ビューの作成
- データベース構造のスナップショットの作成
- データベースに送出された SQL コマンドの監視
- データベースへの接続

これらの作業の実行手順については、Forte for Java IDE のデータベースエクスプローラについてのヘルプを参照してください。

JDBC コンポーネントの使用

Forte for Java には、画像フォームや画像コンポーネント用のデータベース接続ツールおよび JDBC コード生成ツールがあります。具体的には、JDBC アプリケーションで使用できる 2 つの基本コンポーネントを対象にしています。

- 画像コンポーネント – 表形式のデータベースデータを表示する Swing コンポーネントです。Forte for Java では、Swing 画像コンポーネントを使用して、ユーザーへデータベースデータを中継するフォームを作成します。すなわち、これらのコンポーネントは行データを操作したり、列を表示したりする手段になります。そのための Swing コードは、Forte for Java が自動的に作成します。これ以外の画像コンポーネントに、 JDBC コンポーネントのデータナビゲーションがあります。このコンポーネントをフォームに追加することで、データの表示方法を制御できます。

- 非画像コンポーネント – データベースからのデータの操作が可能であるが、目に見えない JavaBeans コンポーネントです。非画像コンポーネントに、`RowSet` があります。`RowSet` は、データベースから取り出した情報を含んだ行セットです。
  JDBC JavaBeans コンポーネントを使用するには、次の知識が必要です。

  - 「JDBC」タブ

  - JDBCコンポーネントを使用したアプリケーションのプログラミング

    - Forte for Java での画像フォームの作成

    - Forte for Java のコンポーネントインスペクタでの JDBC JavaBeans コンポーネントの使用方法

「JDBC」タブ

コンポーネントパレットの「JDBC」タブには、JDBC JavaBeans コンポーネントのアイコンが含まれています。これらのコンポーネントを使用すると、Java の Swing コンポーネントとデータベースとのデータのやり取りが簡単になります。これらのコンポーネントのプロパティは、コンポーネントインスペクタのプロパティエディタを使用して設定することができます。

次のコンポーネントがあります。

- ConnectionSource（接続ソース）
- PooledConnectionSource（プールされた接続ソース）
- NBCachedRowSet（`NB Cached RowSet`）
- NBJdbcRowSet（`NB Jdbc RowSet`）
- NBWebRowSet（`NB Web RowSet`）
- StoredProcedure（ストアドプロシージャ）
- DataNavigator（データナビゲータ）

接続ソース

接続ソースは、JDBC に準拠したデータベースとの接続を提供する非画像コンポーネントです。**接続ソース**では、次の情報を設定することができます。

- データベース URL
- JDBC ドライバ名
- ユーザー名
- パスワード

プールされた接続ソース

プールされた接続ソースは**接続ソース**によく似ています。ただし、**プールされた接続ソース**を使用した場合は、実行時に確立されたデータベース接続が、アプリケーションによる使用後も開かれたままになります。

この接続は、同じアプリケーションで再使用できるように、接続プールに保持されます。**プールされた接続ソース**は、アプリケーションが接続先のデータベースを頻繁に開いたり閉じたりする場合に使用することができます。

RowSet の利用法

RowSet は、データベースから取り出した行を表現しています。これらのコンポーネントを使用して、複数の Swing コンポーネントのデータモデルを構成することができます。

`RowSet` の基礎知識

`RowSet` オブジェクトには、JDBC の結果セットや、その他の表形式データソース (ファイルや表計算データ) から取り出した行の集まり (行セット) が収容されます。

`RowSet` は、コードでの実装の仕方に応じて、シリアライズ可能にすることも、表形式以外のデータソースに拡張することもできます。

`RowSet` オブジェクトは、JavaBeans モデルのプロパティやイベント通知方式に従っているため、アプリケーションのその他のコンポーネントと組み合わせて使用できる JavaBeans コンポーネントです。

RowSet は、実装の仕方に応じて、接続したままにしておくことも、接続を切り離しておくこともできます。接続が切り離されている RowSet は、データソースとの接続を取得しない限り、データソースからデータを取り込んだり、更新後のデータをデータソースに伝達したりしません。しかし、ほとんどの時間は接続を閉じたままになります。

接続が切り離されている間は、JDBC ドライバや完全な JDBC API が不要になるため、 RowSet のサイズは非常に小さくなります。したがって、接続が切り離されている RowSet は、ネットワークを介してシン (Thin) クライアントにデータを送出するのに理想的な形式です。

RowSet の種類：

「JDBC」タブでは、次の 3 種類の行セットを使用できます。

- NB Cached RowSet

  NBCachedRowSet は、そのデータをメモリーにキャッシュする、接続が切り離されている RowSet です。この RowSet は小規模なデータに適しています。NBCachedRowSet を使用して、携帯情報端末 (すなわち PDA) など、シン Java クライアントで動作するコードを提供するような JDBC アプリケーションを作成することができます。

  RowSet をデータソースから切り離している場合は、その RowSet に適用された更新がデータベースに伝達されます。

- NB Jdbc RowSet

  NBJdbcRowSet は、Swing コンポーネントのモデルで使用される、接続されている ResultSet オブジェクトの JavaBeans™ ラッピングを表しています。NBJdbcRowSet を使用して、すべてのデータを内部キャッシュに格納するキャッシュされた RowSet よりも効率的に、非常に長い表を読み取ることができます。

- NB Web RowSet

  NBWebRowSet は、Swing コンポーネントのモデルで使用される、キャッシュ内にあるフェッチされた行を表しています。キャッシュされた RowSet のすべての機能を提供し、行を XML 形式でインポートしたり、エクスポートしたりすることもできます。ファイルは、HTTP/XML プロトコルを使用して、インターネット経由で送信することができます。

プロパティエディタで「プロパティ」タブに含まれている次のプロパティを設定すると、JDBC RowSet をカスタマイズすることができます。

表 2-1　RowSet のプロパティ

プロパティ	定義
Command	この RowSet にデータを格納する SQL 照会です。構文が正しい SQL の SELECT 照会であれば何でもかまいません。
Cennection provider	構成済み接続ソースです。ドロップダウンリストから選択します。
Read - only	True の場合、RowSet は読み取り専用になります。RowSet から取り出されたデータをデータベースに書き込むことはできません。
Transaction Isolation	RowSet がトランザクションの中にデータを処理する方法を指定します。詳細については、java.sql.Connection のマニュアルを参照してください。
XML output directory (NBWebRowSet のみ)	NBWebRowSet から取り出されたデータの送信先となるディレクトリを指定します。
XML output file (NBWebRowSet のみ)	NBWebRowSet からの XML 出力を含むファイルの名前を指定します。

RowSetの「他のプロパティ」、「イベント」、および「コード生成」タブ

RowSet の「他のプロパティ」タブを使用すると、追加のプロパティを検査および変更できます。

表 2-2　RowSet の「他のプロパティ」タブのプロパティ

プロパティ	定義
Database URL	レコードが更新されるデータベースの場所。ほとんどの場合、この URL は「接続ソース」の「データベース URL」プロパティにリストされているものと同一です。
Default column values	新しい行に挿入される値です。RowSet の列のリストを取り出すには、「列をフェッチ」を押します。
Execute on load	True の場合、NB RowSet をロード時に実行できます。ロード時実行で使用するパラメータを「フォーム接続」モードを選択すると、指定でき、初期化コードを生成することができます。
Password	この NB RowSet を含む表にアクセスするのに必要なユーザーパスワードです。
Table name	レコードが更新されるデータベース表の名前。
User name	レコードを更新するユーザーの名前。

RowSet の「イベント」タブを使用すると、RowSet に関連付けられているイベントを検査および変更できます。

表 2-3　RowSet 「イベント」タブのプロパティ

プロパティ	定義
cursorMoved	cursorMoved イベントのイベントハンドラを指定します。NBCachedRowSet のカーソルを移動すると、このメソッドが呼び出されます。
rowChanged	rowChanged イベントのイベントハンドラを指定します。RowSet で行を変更すると、このメソッドが呼び出されます。

表 2-3　RowSet 「イベント」タブのプロパティ

プロパティ	定義
rowInserted	rowInserted イベントのイベントハンドラを指定します。RowSet で行を挿入すると、このメソッドが呼び出されます。
rowSetChanged	rowSetChanged イベントのイベントハンドラを指定します。RowSet を変更すると、このメソッドが呼び出されます。
rowCompleted	rowCompleted イベントのイベントハンドラを指定します。挿入した行がデータベースにコミットされた後で、このメソッドが呼び出されます。

「コード生成」タブを使用すると、行セットに関連のある前処理コードおよび後処理コードを指定することができます。

表 2-4　「コード生成」タブのプロパティ

プロパティ	定義
コード生成	コンポーネントについて、標準コードとシリアライズ (直列化) コードのどちらを生成するかを選択します。
カスタム作成コード	作成したコンポーネント作成コードを、変数名と等号 (=) を含めないで入力します。この作成コードは、initComponents() メソッドで呼び出されます。このプロパティを空白のままにした場合、IDE がデフォルトのコンポーネント作成コードを生成します。
生成後のコード、初期化後コード、生成前のコード、初期化前コード	コンポーネントの作成コードの前後および初期化コードの前後に追加するカスタムコードを作成します。IDE は必ず、initComponents() の初期化コードの前に作成コードを置きます。

表 2-4　「コード生成」タブのプロパティ (続き)

プロパティ	定義
シリアライズ先	コンポーネントをシリアライズ (直列化) する場合、シリアライズ対象ファイルの名前を設定します。
デフォルトの修飾子を使用	デフォルト修飾子を使用してコンポーネントの変数修飾子 (「public」、「private」など) を生成する場合には、`True` に設定します。デフォルト修飾子は、「オプション」ウィンドウの「フォームオブジェクト」ノードの「変数修飾子」プロパティに指定されています。(「ツール」 > 「オプション」を選択するとウィンドウが表示されます。) 「変数修飾子」プロパティをコンポーネントのプロパティシートに表示し、デフォルト修飾子を無効にするには、`False` に設定します。
変数名	コンポーネントの変数名を変更します。

データナビゲータ

データナビゲータは、JDBC モジュールに用意されている画像コンポーネントです。このコンポーネントを使用すると、事前に作成された GUI を使用して `RowSet` を直接ナビゲートすることができます。このコンポーネントは、アプリケーションのプロトタイプやデータ入力アプリケーションを作成する場合に便利です。

データナビゲータのプロパティエディタで、「プロパティ」タブに含まれている次のプロパティを設定すると、データナビゲータをカスタマイズすることができます。

表 2-5　データナビゲータのプロパティ

プロパティ	定義
Auto accept	データベースの変更を自動的に受け付けます。このプロパティの指定により、データナビゲータを介して加えた変更内容をデータベースに即座に伝達されるか、RowSet に追加した後、要求に応じてデータベースに伝達されるかのいずれかです。
Bound RowSet	データナビゲータで制御される `RowSet` を指定します。
Layout of buttons	ボタンを 1 行ないし 2 行のどちらで表示するかを指定します。
Modification buttons visible	修正するボタンの表示を有効または無効にします。

ストアドプロシージャ

ストアドプロシージャは、論理ユニットを形成し固有のタスクを実行する SQL 文のグループです。ストアドプロシージャは、データベースサーバー上で実行するオペレーションまたは照会をカプセル化します。当然ながら、これらのプロシージャーは実行先の DBMS (データベースマネージメントシステム) によって異なります。

Forte for Java IDE では、ストアドプロシージャは、JDBC アプリケーションでデータベースストアドプロシージャを表す非画像コンポーネントです。ストアドプロシージャは、GUI イベント (ボタンのクリックなど) に応じて呼び出すことができます。

ストアドプロシージャの構文は、Forte for Java がサポートする DBMS ごとに異なります。たとえば、ある DBMS では、`begin`、`end` といったキーワードを使用して、プロシージャ定義の始まりと終わりを示しますが、別の DBMS では、このプロシージャ定義を示すのに別のキーワードを使用します。

各種の DBMS で使用可能なストアドプロシージャや、JDBC アプリケーションからストアドプロシージャを呼び出す方法については、『JDBC Tutorial』を参照してください。

ストアドプロシージャのプロパティエディタで、「プロパティ」タブに含まれている次のプロパティを設定すると、ストアドプロシージャをカスタマイズすることができます。プロパティシートにこれらのプロパティを指定すると、ストアドプロシージャをどのユーザーアクションにも接続できます。

表 2-6 ストアドプロシージャのプロパティ

プロパティ	定義
Arguments	アプリケーションから呼び出したときにストアドプロシージャで使用するデータベースのデータを表します。
Bound rowset	ストアドプロシージャを呼び出した後にデータベースから再表示されるドロップダウンリストから RowSet を選択できます。
Call format	ストアドプロシージャの呼び出し書式です。たとえば、「Name」と「Arguments」が含まれる場合、それらはこのプロパティシート上の「Name」プロパティと「Arguments」プロパティの値に置換されます。
Cennection provider	設定した接続ソースで、そのコンテキストにアプリケーションからストアドプロシージャが呼び出されます。
Name	呼び出されるストアドプロシージャの名前です。

JDBC コンポーネントを使用したプログラミング

JDBC モジュールに用意された画像/非画像コンポーネントを Swing コンポーネントと組み合わせて、データベースデータの検索や操作に使用するフォームを作成します。

たとえば、多くの Swing コンポーネント (`JList`、`JTable`、`JComboBox`、`JButton`、`JToggleButton`、`JRadioButton`、`JCheckbox`) は、表示するデータのデータモデルと関連付けられています。IDE では、プロパティエディタやコンポーネントインスペクタを使用して、これらの Swing コンポーネントのデータモデルをカスタマイズし、データベースアクセスで使用する JDBC コンポーネントを指定します。この JDBC コンポーネントを指定すると、Forte for Java によって、それに対応する JDBC コードが自動的に生成されます。

コンポーネントのデータモデルの設定

以下の Swing コンポーネントには、関連付けられたデータモデルがあります 。

- `JList`
- `JTable`
- `JComboBox`
- `JButton`
- `JToggleButton`
- `JRadioButton`
- `JCheckbox`

これらのデータモデルを構成してデータをデータベースから使用してください。

「`JTable`」は、データベースの表を表示する最も一般的なコンポーネントです。このモデルは、各 Swing コンポーネントのプロパティシート (「`model`」プロパティ) で構成できます。

データベースの列の選択

「`JTable`」または「`JList`」など、複数行を表示できるコンポーネントにも「`selectionModel`」プロパティがあります。

「`JList`」と「`JComboBox`」にも特殊な種類のモデルがあります。このモデルでは、SQL の結合を利用して、ある RowSet から 1 列を使用して別の RowSet の別の行を処理してデータを表示します。詳細については、下記を参照してください。

「document」プロパティ (JTextField、JTextArea、JPasswordField、JTextPane、および JEditorPane) を備えているテキストコンポーネントは、このプロパティを設定してデータベースからのデータを使用することができます。

▼ JTable のデータモデルを構成する。

1. JTable のプロパティシートの「model」プロパティで、そのプロパティの値をクリックし、次に「...」ボタンをクリックしてプロパティエディタを開きます。
2. 「TableEditor」モードを選択します。
3. 「行セット」フィールドで、表に表示されている RowSet を選択します。
4. 「列をフェッチ」を使用して、列名をリストに読み込みます。
5. 「追加」、「削除」、「すべて編集可能」、「上へ移動」、「下へ移動」の各ボタンを使用して、名前と表内の列の順序を設定します。
6. 「了解」をクリックして変更内容を保存し、プロパティエディタを閉じます。

▼ JTable と JList の選択モデルを構成する。

1. プロパティシートの「selectionModel」プロパティで、そのプロパティの値をクリックし次に「...」ボタンをクリックして、プロパティエディタを開きます。
2. 「行セット」フィールドで RowSet を選択して、表またはリストに表示します。
3. 「了解」をクリックして変更内容を保存し、プロパティエディタを閉じます。

▼ JList と JComboBox のデータモデルを構成する。

1. プロパティシートの「model」プロパティで、そのプロパティの値をクリックし、次に「...」ボタンをクリックして、プロパティエディタを開きます。
2. 「ListEditor」モード、または「ComboBoxEditor」モードを選択します。「一次行セット」フィールドで、行を取り出すデータモデルの RowSet を選択して、「列」ドロップダウンリストから 1 列選択します。
3. 必要に応じて、「二次行セット」フィールドで (SQL の結合に従って) データの表示元となる RowSet を選択します。一次行セットと二次行セットに指定した対応する列のデータ型は同じでなければなりません。

4. JComboBox の場合は、「結合」チェックボックスをオンにすると、対応するコンポーネントがデータベース結合の結果を表示します。「結合」チェックボックスがオフになっている場合、対応するコンポーネントは、一次行セットの値を設定するためのコードマップとして使用されます。

5. 「データ列」(結合列) と「列を表示」(表示データ) を選択します。「了解」をクリックして変更内容を保存し、プロパティエディタを閉じます。

▼ JCheckbox、JRadioButton および JToggleButton のデータモデルを構成する。

1. プロパティシートの「model」プロパティで、そのプロパティの値をクリックし、次に「...」ボタンをクリックして、カスタムプロパティエディタを開きます。

2. 「行セット」フィールドでデータのフェッチ元となる RowSet を選択します。

3. 列を選択します。この列から取り出されるデータは、コンポーネントを選択するかどうかの判定に使用されます。

4. 選択したコンポーネントに対応するデータベース値を「選択」フィールドに入力し、選択解除したコンポーネントの値を「選択解除」フィールドに入力します。

5. 「了解」をクリックして変更内容を保存し、プロパティエディタを閉じます。

▼ テキストコンポーネントのドキュメントモデルを構成する。

1. プロパティシートの「document」プロパティで、そのプロパティの値をクリックし、次に「...」ボタンをクリックして、プロパティエディタを開きます。

2. 「行セット」フィールドでデータのフェッチ元となる RowSet を選択します。

3. テキストコンポーネントを表示する列を選択します。

4. 「了解」をクリックして変更内容を保存し、プロパティエディタを閉じます。

画像フォームの作成

プロパティエディタを使用して、アプリケーションで使用する Swing コンポーネントをカスタマイズしたら、これらの Swing コンポーネントを使用して、データベース操作用の画像フォームを作成します。

▼ データベースアクセス用の Swing コンポーネントを使用して画像フォームを作成する。

1. Forte for Java IDE では用意されたテンプレートを使用して、Swing コンポーネントのフォームを作成します。
2. コンポーネントパレットから、必要な非画像コンポーネント (接続ソースかプールされた接続ソースのどちらか一方、`RowSet` またはストアドプロシージャ) をフォームに追加します。
3. プロパティエディタを使用して、これらのコンポーネントをアクセス先のデータベースに合わせてカスタマイズします。
4. 必要な画像コンポーネント (データナビゲータなど) を追加します。
5. プロパティエディタを使用して、これらのコンポーネントを適切にカスタマイズし、使用する `RowSet` に関連付けます。

   JDBC アプリケーションで使用する Swing コンポーネントを指定すると、Forte for Java によってアプリケーションに必要な Swing クラスが自動的に作成されます。
6. フォームのプロパティエディタを使用して、実行時にキャッチする例外を指定し、フォームを実行します。

コンポーネントインスペクタでの JDBC コンポーネントの使用方法

Forte for Java のコンポーネントインスペクタを使用すると、JDBC アプリケーションで使用するコンポーネントのプロパティを変更することができます。コンポーネントインスペクタの「非画像コンポーネント」の階層に、次のコンポーネントが表示されます。

- `NBCachedRowSet`
- `NBJdbcRowSet`
- `NBWebRowSet`
- ConnectionSource
- PooledConnectionSource
- StoredProcedure

コンテナ階層の位置に従って、`DataNavigator` や、その他の Swing コンポーネントが表示されます。

「JDBC フォームウィザード」の使用

「JDBC フォームウィザード」を使用すると、画面に表示される指示に従って、データベース表にアクセスするフォームを作成することができます。14 ページの「JDBC コンポーネントの使用」 に記載したプログラミング作業を、プロパティを明示的に編集することなく実行することができます。ウィザードの実行が完了すると、アプリケーション、アプリケーションのファイル名、パッケージが生成されます。

次の節では、Forte for Java IDE に付属しているサンプルの PointBase Server Database を使用して、「JDBC フォームウィザード」の使い方を説明します。

▼ JDBC ウィザードを開く。

- 「ツール」>「JDBC フォームウィザード」を選択します。

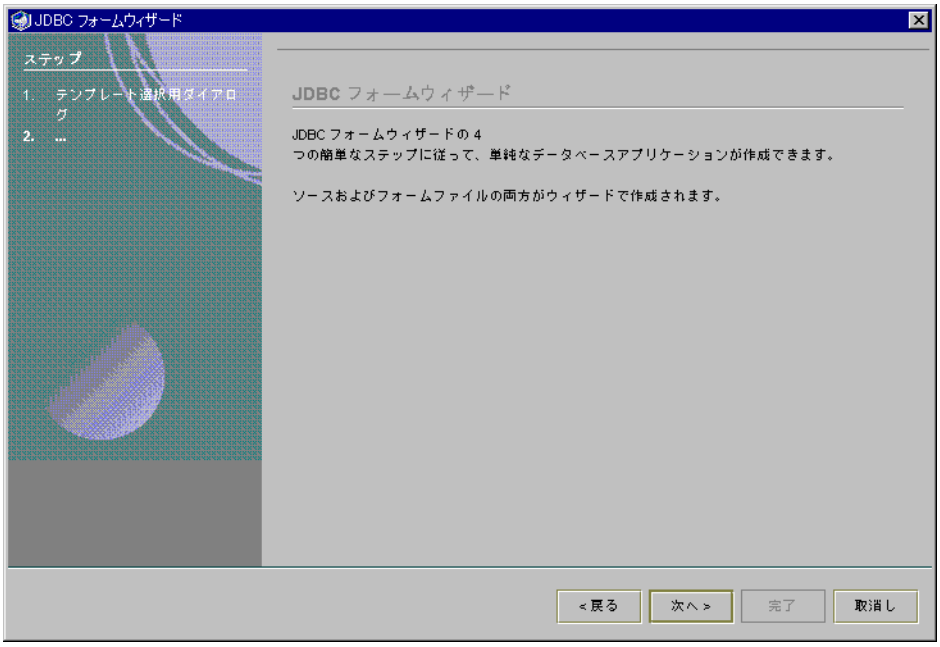

図 2-1　JDBC フォームウィザード

接続の確立

「JDBC フォームウィザード」を使用する場合や、JDBC タブを使用して JDBC クライアントアプリケーションを作成する場合は、最初に使用したい DBMS との接続を確立します。

通常は、フォームエディタや「JDBC フォームウィザード」でフォームを作成すると、JDBC アプリケーションで使用可能なコードが自動的に生成されます。JDBC アプリケーションによって DBMS から取り出された情報は、このフォームに表示されます。

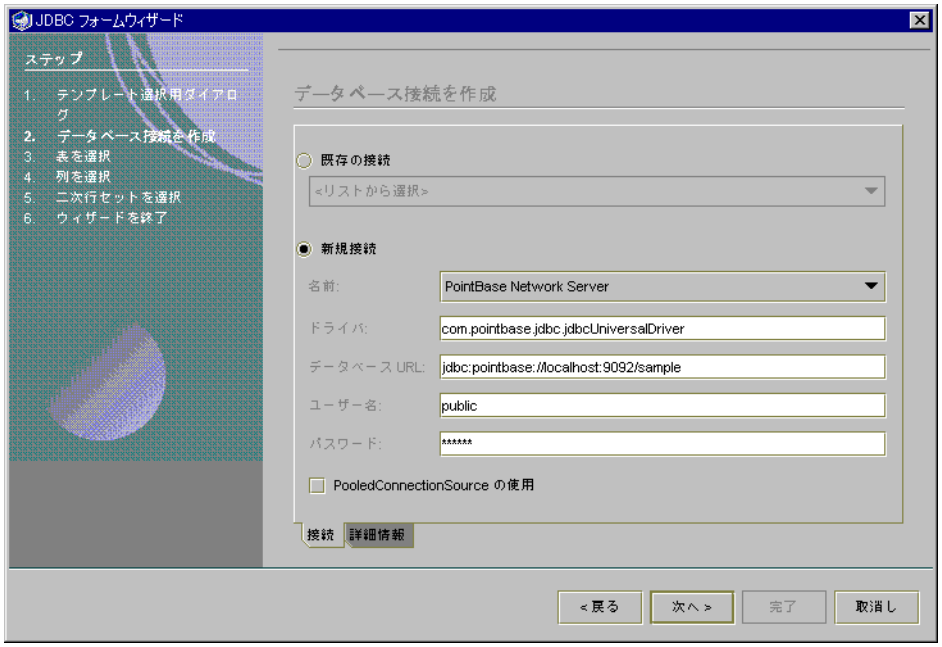

図 2-2　JDBC フォームウィザード、データベース接続を作成

JDBC フォームウィザードの 2 枚目のパネルで、データベースとの接続を確立できます。このパネルでは、データソースに対する接続プールを使用するように指定することができます。

ただし、新しい接続が必要な場合は、次の情報を指定する必要があります。

- データベースの名前。たとえば、PointBase Network Server などです。

- データベースの JDBC ドライバ名。たとえば、`com.pointbase.jdbc.jdbcUniversalDriver` などです。
- データベースの位置を示すデータベース URL。たとえば、`jdbc.pointbase://localhost:9092/sample` などです。
- ユーザー名
- パスワード
- 「PooledConnectionSource の使用」チェックボックスをオンにすると、オプションで接続プールを指定できます。
- オプションで「詳細設定」タブを選択にすると、表を取得するためのスキーマを指定できます。

これらの情報は、生成される JDBC アプリケーションのコードに組み込まれます。

既存の接続を選択するには、「Use Existing Connection」ラジオボタンをクリックし、ドロップダウンリストから接続を選択します。

「次」ボタンをクリックすると、Forte for Java からメソッドが呼び出され、このパネルに入力した情報に従ってデータベース接続が作成されます。このデータベース接続の使い方は、このウィザードで JDBC アプリケーションのコードを作成するときと同じです。

データベースの表またはビューの選択

「JDBC フォームウィザード」の 3 枚目のパネルでは、次の作業を行うことができます。

- 接続先のデータベースの表またはビューを選択します。
- それぞれの表を読み取り専用にするかどうかを指定します。読み取り専用にした場合は、該当する表への書き込みはできません。
- 行挿入のイベントハンドラ (`rowInserted`) を表に追加するかどうかを指定します。このイベントハンドラは、行挿入イベントの発生をチェックします。
- 表のトランザクション遮断レベルを設定します。30 ページの「トランザクション遮断レベル」 を参照してください。
- 表に適用する SQL コマンドを指定します。

「JDBC フォームウィザード」を使用すると、ウィザードで指定した表に対し SQL 文を実行できます。SQLの出力データを利用して、画像フォームを生成します。このウィザードで SQL 文を指定すると、適切な SQL コードが自動的に生成されます。選択した表に適用されるデフォルトの SQL コマンドは、図 2-3 に表示されているコマンドです。

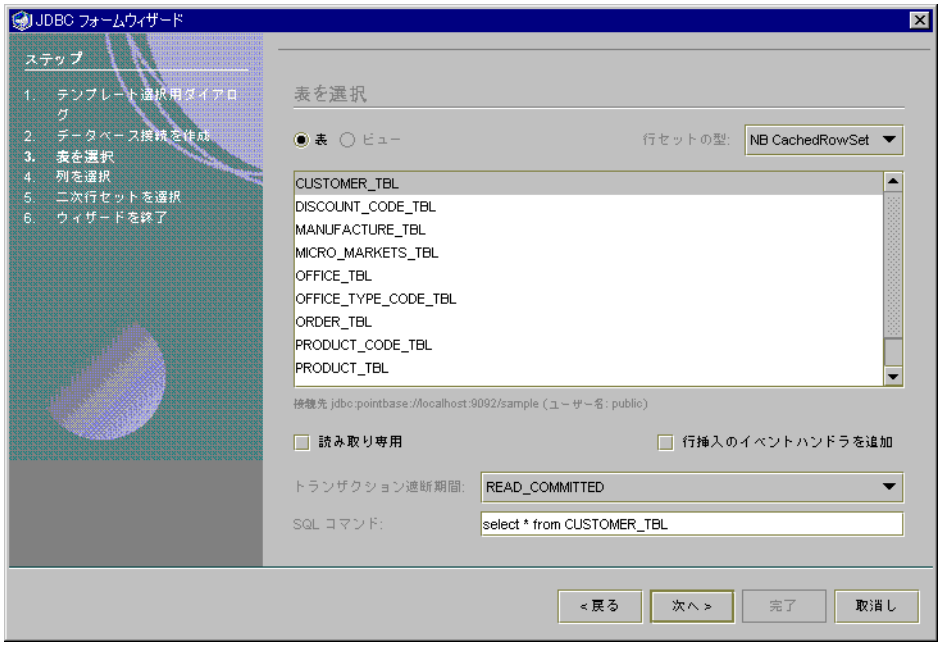

図 2-3　JDBC フォームウィザード、表を選択

トランザクション遮断レベル

DBMS では、トランザクション実行時の衝突を回避するため、**ロック機能**が採用されています。アプリケーションがトランザクションをコミットするかデータベースからロールバックするまで、ロックが有効となります。

ロック機能は、トランザクション遮断レベルに従って設定されます。このロックは、アプリケーションに返される (またはアプリケーションからデータベースにコミットされる) `ResultSet` 全体に適用されます。

デフォルトのトランザクション遮断レベルは、DBMS ごとに異なります。Forte for Java では、JDBC フォームウィザードの 3 枚目のパネルで、次のいずれかのトランザクション遮断レベルを選択することができます。

注 - ドライバと DBMS の両方が使用するトランザクション遮断レベルを、サポートしている必要があります。

表 2-7　トランザクション遮断レベル

プロパティ	定義
TRANSACTION_READ_COMMITTED	まだコミットされていない変更を含んでいる行の読み取りを禁止します。
SERIALIZABLE	TRANSACTION_REPEATABLE_READ の禁止項目を含んでいます。あるトランザクションが WHERE 条件に適合している行をすべて読み取り、別のトランザクションが同じ WHERE 条件に適合した行を挿入し、最初のトランザクションがこの WHERE 条件に適合している行を再び読み取ったときに、2 回目の読み取りで「実体のない行」が抽出される現象を禁止します。
TRANSACTION_NONE	トランザクションに対応しません。
TRANSACTION_REPEATABLE_READ	まだコミットされていない変更を含んでいる行の読み取りを禁止します。また、あるトランザクションが行を読み取り、別のトランザクションがその行を変更し、最初のトランザクションがその行を再び読み取ったときに、2 回目の読み取りで 1 回目とは別の値が読み取られる (再帰的な読み取りが不可能となる) 現象を禁止します。
TRANSACTION_READ_UNCOMMITTED	あるトランザクションによって変更された行を、その変更がデータベースにコミットされる前に別のトランザクションから読み取れるようにします。この変更がロールバックされた場合は、後者のトランザクションが読み取った行は不正になります。

表示する列の選択

JDBC フォームウィザードの 4 枚目のパネルでは、フォームに表示する列を指定することができます。このパネルでは、次の項目を指定できます。

- 生成するアプリケーションに表示する列
- 表示する列の順序
- 列のパラメータ
  - 列のタイトル
  - 列の編集可能性
  - デフォルトの列の値
  - アプリケーションの表を表示するための Swing コンポーネント

次の例では、`JTable` (最も一般的な Swing フォーム) が選択されています。`JTable` フォームは、アプリケーションに複数のデータ列を表示します。

このほかに、次の Swing コンポーネントも選択することができます。

- `Jlist`: リストに列を表示します。
- `JComboBox`: コンボボックスに 1 つの列を表示します。
- `JTextField (s)`: テキストフィールドに 1 つまたは複数の列を表示します。

図 2-4 では、最初の列が選択されています。選択した列は、削除したり、上下に移動したりすることができます。

図 2-4　JDBC フォームウィザード、列を選択

JList または JComboBox を選択する場合、表示できる列は 1 つだけであり、「名前」プロパティから表示する列を選択できます。

1. 「名前」の列に表示されている値に選択します。
2. 組み込まれているコンボボックスから列名を選択します。

▼ 列のタイトルを編集する。

1. 編集する「タイトル」フィールドをダブルクリックします。2 つのタブが付いた「列名」ウィンドウが表示されます。
2. 「文字列の値」タブを選択し、新しい名前を単純な文字列の値として入力します。
3. リソースバンドルを使用して名前を入力するには、「リソースバンドル」タブを選択します。バンドルの名前を「バンドル」フィールドに入力して、「キー」コンボボックスから関連するキーを選択します。
4. 「了解」を選択して、ウィンドウを閉じます。

二次行セット (RowSet) の選択

「二次行セット (RowSet) を選択」このパネルでは、「データベース接続を作成」パネルで作成されたデータベース接続に従って、利用できる表のリストをすべて表示します。また、このパネルは、2 つの RowSet (JList または JCheckbox) が選択された場合に限り使用できます。

このパネルを使用すると、生成されるアプリケーションで二次行セット (RowSet) を生成できます。

▼ 二次行セット (RowSet) を選択する。

1. 「二次行セット (Rowset) を使用」をオンにします。

   この行セットをオンにすると、生成されるアプリケーションで二次行セットが使用されます。
2. 「表」または「ビュー」のラジオボタンを選択します。
3. 「行セットの型」コンボボックスから、行セットの型を選択します。
4. リストから表またはビューを選択します。

5. 対応する行セットを読み取り専用にするには、「読み取り専用」をオンにします。

6. 行挿入のイベントハンドラ (rowInserted) を生成されるアプリケーションのソースコードに追加するには、「行挿入のイベントハンドラを追加」をオンにします。

   新規の行が挿入されるとこのハンドラが呼び出され、デフォルトの列の値が動的に作成できるようになります。

7. 「トランザクション遮断期間」コンボボックスにある値を 1 つ選択して、行セットのトランザクションの遮断期間レベルを設定します。

   デフォルトのトランザクションレベルは、`READ_COMMITTED` です。

8. 「SQL コマンド」テキストフィールドを使用して、行セットを生成するための SQL を記述します。

   デフォルトでは、Forte for Java は、文字列「`select * from table-name` 」を生成します。

9. データベースでは結合で使用するデータ列を選択します。

   この列を選択すると、取り出した主列以外の別のフィールドが表示されます。ただし、このフィールドのデータモデルは、主列と同じデータモデルにする必要があります。

図 2-5　JDBC フォームウィザード、二次行セット (RowSet) を選択

アプリケーションのプレビューと生成

JDBC フォームウィザードの最後のパネルには、作成するアプリケーションのプレビューが表示されます。このパネルを使用して、アプリケーションを完成させます。また、アプリケーションのパッケージ名とファイル名を指定します。

「パッケージ」フィールドでパッケージ名を、「ターゲットファイル」フィールドでターゲットファイル名を指定します。

このパネルで、コンポーネントのレイアウトと、データナビゲータのレイアウトをチェックすることができます。このレイアウトがどのように表示されるかは、アプリケーションで操作するデータの収容先として、どの Swing フォームを選択したかによって異なります。

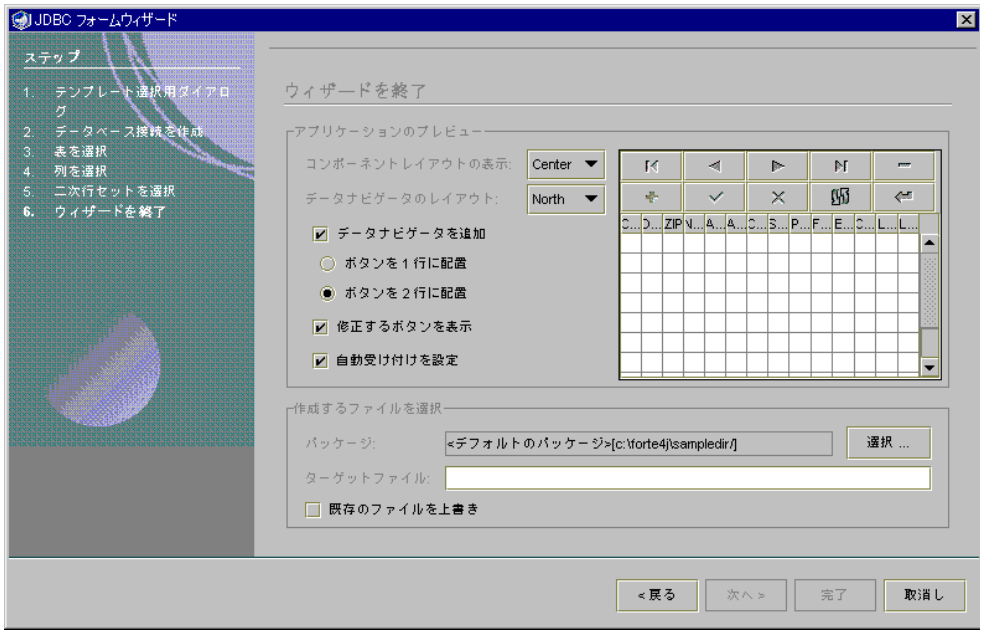

図 2-6　JDBC フォームウィザード、ウィザードを終了

JDBC アプリケーションの実行

JDBC のアプリケーションはその他すべてのフォームと同様に、コンパイル、実行、デバッグを行うことができます。特殊な JDBC ドライバが必要な場合は、JDBC ベースのフォームの外部コンパイル、実行、デバッグがデフォルトで利用できるように、それらのドライバが Forte for Java の CLASSPATH に含まれていることを確認してください。

アプリケーションを IDE 外で実行するには、以下のパッケージへのパスを CLASSPATH に追加します。

- `modules/ext/sql.jar`
- `modules/ext/rowset.jar`
- `lib/ext/jdbc20x.zip`
- 対応する JDBC ドライバ。JDBC ドライバは通常、lib/ext に格納されています。

作成した JDBC アプリケーションで `WebRowSet` を使用する場合、さらに次の 2 つの JAR ファイルが必要です。

- `lib/ext/parser.jar`
- `lib/ext/xerces.jar`

第3章

透過的な持続性の概要

Forte for Java の透過的な持続性 (Transparent Persistence) 機能を使用すると、SQL、JDBC API、データベースプログラミングの知識がなくても、JDBC に準拠したデータベースに格納された持続データを Java オブジェクトとして参照および操作することができます。この章では、透過的な持続性プログラミングモデルの概要について説明しています。

本書で使用している**クラス**、**フィールド**、**オブジェクト**という用語は、すべて Java プラットフォームのクラス、フィールド、オブジェクトを示しています。

透過的な持続性とは

透過的な持続性を利用すると、データストア内の情報に Java オブジェクトとしてアクセスできるために、データベースプログラミングから Java プログラミングを切り離すことができます。この機能は、持続データストアのデータを収容する持続可能 Java クラスによって実現され、特定のデータストアに依存した SQL やコードを作成する必要がなくなります。

まず、透過的な持続性およびそのマッピング機能を使用し、リレーショナルデータベースの表に含まれている列を、自動生成された、またはあらかじめ存在している Java クラスにマップします。これらの Java クラスの関係は、データベースの表の関係を反映したものになります。データベースで外部キーによって結び付けられた表と列は、Java クラスでも参照または集合関係を使用して結び付けられます。

アプリケーションは、Java プログラミング言語によるオブジェクト操作を介してデータストアにアクセスします。専用のデータベースアクセス言語を使用したり、データベーススキーマを意識したりする必要はありません。ビジネスロジックをこれらの Java プログラミング言語クラスに挿入するには、追加のメソッドを定義し、自動生成されるメソッドを拡張するだけです。

透過的な持続性では、データベーススキーマと Java クラスが自動的にマップされます。これには、次の 2 種類の方式があります。

- データベースから Java へのマッピング

データベーススキーマから Java クラスを生成します。スキーマに含まれている任意の表 (またはすべての表) から持続可能クラスを作成することができます。マッピングの対象になるクラスがあらかじめ存在しない場合に最も適した方法です。

- Meet-in-the-middle マッピング

この方法では、既存のスキーマと既存の Java クラスとの間でカスタムマッピングを行います。持続データへのアクセスに使用するクラスをすでに作成している場合は、この方式を使用します。この方式で、データベースから Java へのマッピングで生成したクラスを修正することもできます。

透過的な持続性には実行時ライブラリも含まれています。このライブラリには、透過的な持続性 API を介してアクセスします。透過的な持続性 API は、透過的な持続性オブジェクトの操作やデータベースアクセスを実現する Java ルーチンの集まりで、マップされた Java パッケージの実行環境を提供します。

アプリケーション開発者は、アプリケーションで使用する持続データを表現した Java クラスを操作します。データを取得する必要がある場合は、持続性マネージャまたは照会インスタンスのメソッドを呼び出します (これにより、持続可能クラスのインスタンスが返されます)。データストアからデータを取得する別の方法として、持続性インスタンスの参照または集合関係をナビゲートします。データを変更する必要がある場合は、これらの持続可能インスタンスのメソッドを呼び出します。

Forte for Java の透過的な持続性モジュールには、将来の Java Data Objects (JDO) の仕様が先取りされています。JDO では、持続性マネージャや、JDO 環境のその他の構成要素のスケーラビリティと移植性が実現されています。それぞれの JDO 環境では、持続可能クラスを各種のデータベースソフトウェアや接続管理ソフトウェアと連携させることができます。

透過的な持続性のプログラミング

透過的な持続性は、データストアの知識をもつ開発者と、アプリケーションの知識を持つ開発者の 2 種類のタイプの開発者が、次の異なる作業を分担することを想定しています。

- 持続可能クラスの開発
- 持続性認識アプリケーションの開発

持続可能クラスの開発

持続可能クラスを作成する開発者は、持続性データストア内でデータをモデル化するクラスのセットを作成します。これらのクラスを開発する際に使用できるウィザードについては 第 4 章 に説明されています。

▼ データベーススキーマから Java パッケージを作成する

1. スキーマ収集ツールを使用し、データベーススキーマを収集します。

   データベースに接続していないときでも使用可能な、データベーススキーマのファイルシステム表現を作成します。

2. 次のいずれかの方法で、データベーススキーマに対応する持続可能 Java クラスを作成します。

   - 「Java を生成」ウィザードを使用して、収集済みのデータベーススキーマ表から新しい Java クラスを生成します。この際に、生成クラスからスキーマ表へのマッピングも併せて行います。
   - 「データベースへマップ」ウィザードを使用して、既存の Java クラスを持続可能クラスにし、これらのクラスにデータベーススキーマをマップします。このウィザードを使用して、既存のマッピングをカスタマイズすることもできます。たとえば、フィールドをアンマップしたり、新しく追加したフィールドをマップしたり、クラスを別のスキーマに含まれている表にマップしたり、スキーマを変更し、再収集した後でマッピングを修正したりすることができます。

3. 生成したクラスにビジネスロジックを追加します。

   データベースデータに対応した Java クラスのソースコードを編集します。通常は、これらのクラスにビジネスロジックを追加します。生成されたメソッドや、既存のメソッドにコードを追加したり、これらのクラスに新しいメソッドを追加したりすることができます。

4. ソースコードファイルをコンパイルします。

   コードを作成した後で、Forte for Java IDE を使用し、Java クラスソースファイルをコンパイルします。これらのクラスはデータベース表を表現しています。

5. 持続可能クラスおよび持続性認識クラスをアーカイブまたはパッケージ化します。

   Forte for Java で、コンパイルした Java クラスを `.jar` ファイルにパッケージ化します (配備作業や、持続可能クラスの変更を伴わない別の開発作業で使用できるようにするため)。Forte for Java は、これらのクラスが持続可能クラスまたは持続性認識クラスのいずれであるかを判定し、`.jar` ファイルに追加する前に、透過的な持続性に対応するように拡張します。エンハンサは、すべての必要なサポートを自動的にクラスのバイトコードに追加し、持続性フィールドにアクセスされると同時に透過的な持続性実行環境とクラスが共同で動作するようにします。

注 - 「Persistence 実行」または「Persistence デバッガ」を使用して Forte for Java 内部のアプリケーションを実行またはデバッグする場合、バイトコードの拡張は特殊クラスのローダーによって行われます。この場合、持続可能または持続性認識クラスを `.jar` ファイルにパッケージする必要はありません。

持続性認識アプリケーションの開発

持続性認識アプリケーションの開発者が知っておくべきことは、アプリケーションデータを表現している持続可能クラスと、これらのクラスを操作するために透過的な持続性に用意された標準的な API だけです。これらの API を使用することで、データストアのデータを選択、更新、挿入、削除することができます。これらの呼び出しについては、第 5 章 を参照してください。

データベース表に対応した持続可能 Java クラスを作成すると、これらのマップされた Java クラスを使用するアプリケーションを作成することができます。これらのクラスを使用すると、必要な JDBC 文は自動的に生成されます。開発者が行う必要があるのは、トランザクションを開始、終了し、データベース中のオブジェクトを検出する照

会を指定することだけです。この照会は、Java の式に似たブール型のフィルタで、SQL の SELECT 文に変換されます。照会記述の詳細については、132 ページの「データベースの照会」 を参照してください。

マップされた Java クラスに、Java Server ページ (JSP™) から直接アクセスすることもできます。その場合は、JSP に用意された透過的な持続性タグを使用します。これらのタグについては、付録 B および 『Web コンポーネントのプログラミング』を参照してください。

透過的な持続性および Enterprise JavaBeans

Enterprise Java Beans™ (EJB™) は、分散型のビジネスアプリケーションを開発および配備するためのコンポーネントアーキテクチャです。透過的な持続性は、次の領域において、Enterprise JavaBeans コンポーネントとの統合をサポートします。

- 依存型オブジェクトとして持続可能クラスを直接的に使用する持続認識コンポーネントのステートフルおよびステートレスセッション Bean。
- 持続状態にアクセスし、変更することによって実際にビジネスメソッドを実装する委任オブジェクトとして持続可能インスタンスを使用する持続性コンポーネントの Bean 管理持続性エンティティ Bean。

このリリースでは、コンテナ管理持続性エンティティ Bean はサポートされていません。

Enterprise JavaBean コンポーネントとの統合については、第 6 章 を参照してください。

第4章

持続可能クラスの開発

この章では、透過的な持続性を使用して Java プログラミング言語のクラスとリレーショナルデータベースとをマップする方法について説明しています。

マップ機能

マッピングとは、オブジェクト指向モデルを関係モデル (すなわちリレーショナルデータベースのスキーマ) に結び付ける作業です。透過的な持続性には、相互に関連付けられたクラス群を、関係モデルの相互に関連付けられたメタデータに結び付ける機能があります。これらのクラスには、データと、データを操作するためのメソッドが含まれます。このオブジェクト表現によるデータベースを使用して、Java アプリケーションの基盤を作ることができます。また、このマッピングをカスタマイズすることで、特定のアプリケーションのニーズに合わせてクラスを最適化することもできます。

マッピングの結果として、単一のデータモデルが生成されます。このデータモデルを使用して、持続的なデータベース情報と、通常の一時的なプログラムデータの両方にアクセスすることができます。開発者が理解する必要があるのは、Java プログラミング言語オブジェクトだけです。その元になっているデータベーススキーマについての知識は必要ありません。

マッピングを変更しても、その影響を受けるのは Java クラスだけです。データベーススキーマは現状の定義のまま変わりません。図 4-1 に示すように、データベーススキーマと Java クラスはまったく別のものです。

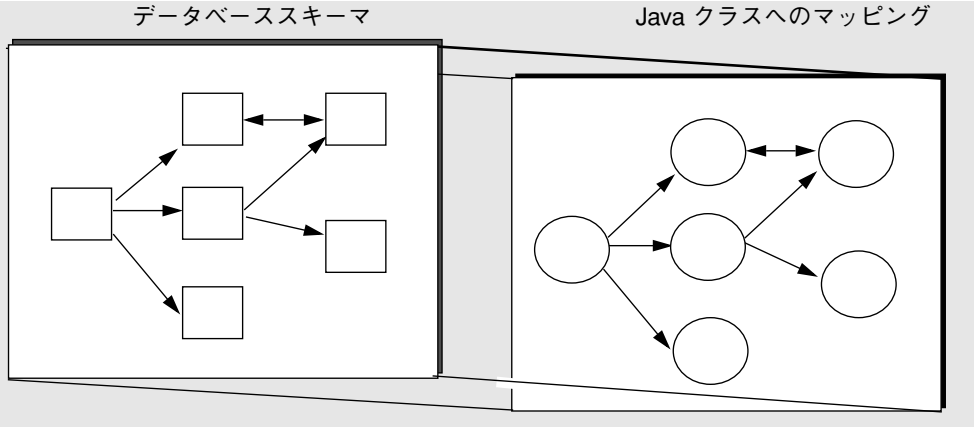

図 4-1　データベースから Java クラスへのマッピング

スキーマからマッピングとクラスモデルの両方を生成することも、既存のクラスを既存のスキーマにマップすることもできます。

注 - 透過的な持続性は、1 つのデータベーススキーマの中で各クラスを表にマップします。また、関連のあるすべてのクラスも、このスキーマにマップされる必要があります。

マッピング方式

持続可能クラスは、従業員、部署といったデータエンティティを表現したものです。データエンティティをモデル化するには、持続可能クラスに、データストア内の列に対応する持続フィールドを追加します。

最も単純なモデル化では、1 つの持続可能クラスでデータストア内の 1 つの表を表現し、その表のすべての列について持続フィールドを作成します。たとえば、`Employee` クラスの場合は、データストアの `EMPLOYEE` 表に含まれている、`lastname`、`firstname`、`department`、`salary` といった列ごとに、持続フィールドを 1 つずつ作成します。

データストア内の一部の列だけを、持続フィールドとして使用することもできます。

透過的な持続性で Java クラスをデータベーススキーマにマップするには、次の 2 通りの方法があります。

- データベースから Java へのマッピング

「Java を生成」ウィザードを使用し、データベーススキーマから Java クラスを生成します。このウィザードは、該当スキーマの任意の表またはすべての表にマップされる持続可能クラスを作成します。マッピングの対象になるクラスがあらかじめ存在しない場合に最も適した方法です。

このマッピング方式で開発者が行う必要があるのは、スキーマの中のどの表をマップするかを指定することだけです。モデル化プロセスの間に、データ間の関係を示す主キーフィールドや外部キーフィールドといったスキーマが分析され、その Java 表現が作成されます。その結果作成されるオブジェクト群は、データベース中のメタデータの構造を反映したものになります。Java コードは自動的に生成されます。

- Meet-in-the-middle マッピング

「データベースへマップ」ウィザードと「プロパティ」ウィンドウを使用し、既存のスキーマと既存の Java クラスとの間にカスタムマッピングを作成します。持続データのアクセスに使用するクラスがすでに存在する場合には、この方法を使用してください。この方式で生成したクラスを修正することもできます。

関係のマッピング

関係には 1 対 1、1 対多、多対多の種類があり、このうちのどれであるかは、関係に含まれる各クラスのインスタンス数によって決まります。関係を利用することで、あるオブジェクトからその関連オブジェクトを操作できます。データベースにおける関係は外部キー列によって、また多対多関係の場合には結合表によって表されます。Java コードでは、関係はその多重度に応じて、オブジェクト参照 (コレクション型や持続可能型のフィールド)で表現されます。

透過的な持続性が Java コードを生成する場合、コレクションフィールドは 1 対多関係の「多」の側を表します。透過的な持続性では、実際の持続可能クラス型の変数を使用して 1 対多関係の「1」の側を表します。

たとえば、従業員のコレクションに対して関係を持つ部署オブジェクトについて取りあげてみます。部署オブジェクトから関係をナビゲートすることで、この部署に関する従業員全員を見ることができます。同様に、1 名の従業員を表示させ、その所属先

部署を見ることもできます。1 つの部署には複数の従業員を配属することができますが、それぞれの従業員の配属先は 1 つしかありません。データベースでは、外部キーを使用してこの関係を実現します。

Department クラスには、HashSet 型の employees フィールドが含まれることが考えられます。この HashSet フィールドは、多数の従業員を表す能力を部署オブジェクトに与えます。また、Employee クラスには、Department 型の department フィールドが含まれています。Department 参照フィールドにより、1 名の従業員に 1 つの部署を持たせることができます。

Department クラスに組み込まれるコードは次のようになります。

```
private java.util.HashSet employees;
```

Employee クラスに組み込まれるコードは次のようになります。

```
private Department department;
```

クラスのフィールドノードには、関係フィールドが表示されます。このフィールドには、関係するクラス、上限、下限などのプロパティもあります。Meet-in-the-middle マッピングでは、次のプロパティが設定されません。プロパティウィンドウでそれらのプロパティを設定する必要があります。詳細については、77 ページの「オプションとプロパティの設定」 を参照してください。

関係の作成は「Java を生成」ウィザードで自動的に行なったり、Java コードで正しい型のフィールドを作成することによって行うこともできます。

注 - Java 生成時、透過的な持続性は、アンマップされているクラスを参照する関係フィールドを無視します。この場合、透過的な持続性は関係フィールドを通常フィールドとして扱います。

「Java を生成」ウィザードは、データベースの表から取り出した外部キーを使用して関係を確認します。同ウィザードは、さまざまな表を参照する外部キーを持つ表として結合表を解釈します。

たとえば、DEPARTMENT 表と EMPLOYEE 表があり、DEPARTMENT と EMPLOYEE との間に 1 対多関係があるとします。いずれの表にも主キーがあります。また、EMPLOYEE の表には、DEPARTMENT 主キーである DEPID に対応する値を持つ外部

キー列があります。このスキーマを元に、透過的な持続性は Department クラスと Employee クラスを生成します。Department クラスには多数の従業員を収容できるフィールドがあり、Employee クラスには 1 つの部署だけを参照できるフィールドがあります。図 4-2 に、この様子が示されています。

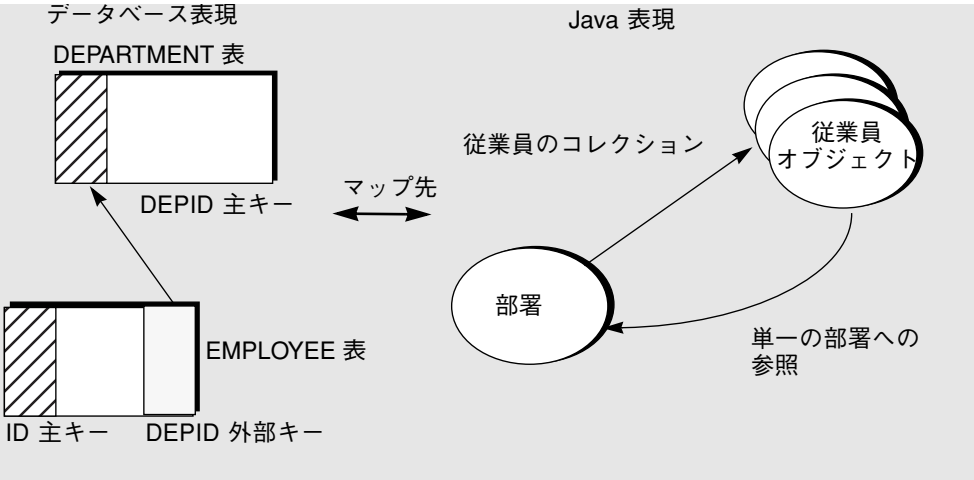

図 4-2　外部キーと 1 対多の関係

データベースで多対多関係にある表を表すには、結合表を使用します。Java 側では、関係の両側にあるクラスが他方のオブジェクトに対する参照を複数個収容できるフィールドを使用します。図 4-3 に、多対多関係の様子が示されています。

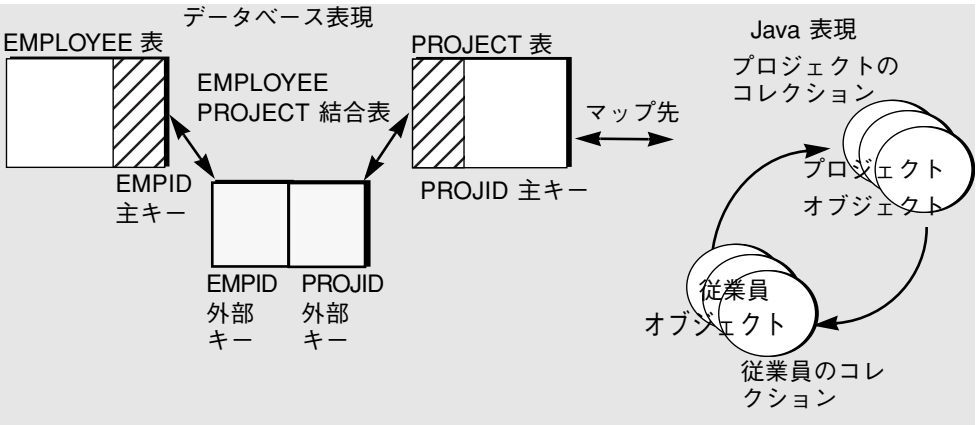

図 4-3　外部キーと多対多の関係

注 - 透過的な持続性は、結合表における重複エントリをサポートしません。関係の「多」の側は `HashSet` を使用して実装されるため、オブジェクトの重複は認められません。

管理されている関係

一対のクラスのフィールド間で管理されている関係では、関係の片側の動作がもう一方の側に影響します。

実行時、あるインスタンスのフィールドが別のインスタンスを参照するように修正された場合、参照されるインスタンスの関係フィールドは関係の変更を反映するように修正されます。

透過的な持続性は、次の関係に対応します。

- 1 対 1 の関係
- 1 対多の関係
- 多対多の関係

1 対 1 の関係

1 対 1 の関係では、各クラスに単一の値が入るフィールドがあり、値の型は相手側のクラスに対応します。関係の片側のフィールドに加えられた変更は、関係の変更として処理されます。一方のフィールドがヌル以外の値からヌルに変更された場合、もう一方のフィールドもヌル以外の値からヌルに変更されます。一方のフィールドがヌルからヌル以外の値に変更された場合、もう一方のフィールドは相手側のインスタンスを参照するように変更されます。もう一方のフィールドがヌル以外の値であった場合、関係が変更される前にそのフィールドの値がヌルになります。

1 対多の関係

1 対多の関係では、「多」の側に単一値のフィールドがあり、「1」の側に複数値のフィールド (コレクション) があります。

インスタンスがコレクションフィールドに追加された場合、新しいインスタンスのフィールドは、コレクションフィールドを含むインスタンスを参照するように更新されます。インスタンスがコレクションから削除された場合、そのインスタンスのフィールドはヌルになります。

「多」の側のフィールドの変更、追加、減算は、関係の変更として処理されます。「多」の側のフィールドがヌルからヌル以外の値に変更された場合、「1」の側のコレクション値のフィールドにこのインスタンスが追加されます。「多」の側のフィールドがヌル以外の値からヌルに変更された場合、「1」の側のコレクション値のフィールドからこのインスタンスが削除されます。

多対多の関係

多対多の関係では、関係の両側に複数値 (**コレクション**) のフィールドがあります。片側のコレクションの内容に加えられた変更は、関係の変更として処理されます。インスタンスが片側のコレクションに追加された場合、そのインスタンスはもう一方のコレクションにも追加されます。インスタンスが片側のコレクションから削除された場合、そのインスタンスはもう一方のコレクションからも削除されます。

注 - 管理されている関係でオブジェクトを削除する場合、警告メッセージは表示されません。透過的な持続性は、外部キー側の関係を自動的に取り消し、確定メッセージを表示することなくオブジェクトを削除します。

透過的な持続性の Java 生成 オプションを設定すると、管理されている関係は自動的に生成されます。これらのオプションを設定するには、「Java を生成」ウィザードの「オプションをカスタマイズ」区画を利用するか (57 ページの「スキーマからの持続可能クラスの生成」 を参照してください)、「ツール」>「オプション」を選択した後、「透過的な持続性」の下にある「Java 生成オプション」を選択します (78 ページの「Java 生成オプション」 を参照してください)。

次の手順は、既存の 2 つのクラスのに対し管理されている関係を作成する方法を説明するものであり、Meet-in-the-middle マッピングを採用しています。

▼ 管理されている関係を作成する

1. 2 つのクラスに関係フィールドを 1 つずつ作成します。
2. フィールドが持続としてマークされていることを確認します。65 ページの「フィールドを持続フィールドにする。」 を参照してください。
3. エクスプローラウィンドウで、クラスの一方を展開して関係フィールドを選択します。

4. フィールドのプロパティウィンドウを開きます。

   もう一方のクラスの名前が、「関連クラス」プロパティの値として表示されます。値が表示されない場合、プロパティ値をクリックし、「...」ボタンをクリックして、関連クラスを選択します。クラスが持続可能でない場合、クラスを変換する必要があります (64 ページの「クラスを持続可能クラスにする」 を参照してください)。

5. もう一方のクラスを選択し、「了解」をクリックします。

6. プロパティウィンドウに戻り、「関連フィールド」をクリックします。もう一方のクラスから関係フィールドを選択します。

   ドロップダウンメニューに希望のフィールドが表示されない場合、そのフィールドが持続としてマークされているかどうか確認してください。フィールドが既にマップされている場合、「マッピング」プロパティのドロップダウンメニューを使用してフィールドをアンマップします。

7. エクスプローラウィンドウで、もう一方のクラスを展開して関係フィールドを選択します。

8. フィールドのプロパティウィンドウを開きます。

   「関連クラス」プロパティおよび「関連フィールド」プロパティが、指定したように設定されています。

   2 つの関係フィールドは、管理されている関係を表しています。

   この関係をデータベースにマップするには、47 ページの「関係のマッピング」 を参照してください。

持続可能クラスの開発

スキーマを収集する

Java クラスをデータベーススキーマにマップする前に、スキーマを収集する必要があります。スキーマを収集すると、ファイルシステムに作業コピーが作成されます。これで、データベースに影響を与えずに、マップ作業を進めることができます。

注 - 収集したスキーマはパッケージに格納することをお奨めします。スキーマを入れるパッケージがない場合は、ファイルシステムをマウスの右ボタンでクリックし、「新規パッケージを作成」を選択して、パッケージを作成してください。

▼ スキーマを収集する

1. 次のいずれかの方法で、「データベーススキーマ」ウィザードを表示します。
   - ファイルシステムをマウスの右ボタンでクリックし、「新規」>「Databases」>「Databases Schema」を選択します。
   - 「ファイル」メニューから「新規」を選択し、「テンプレートウィザードから新規作成」ウィザードで「Databases」をダブルクリックし、「Databases Schema」を選択します。
   - 「ツール」メニューから「データベーススキーマの収集」を選択します。
2. 「作成場所」区画 (図 4-4) で、スキーマの作業用コピーのファイル名を入力し、収集したスキーマのパッケージを選択します。

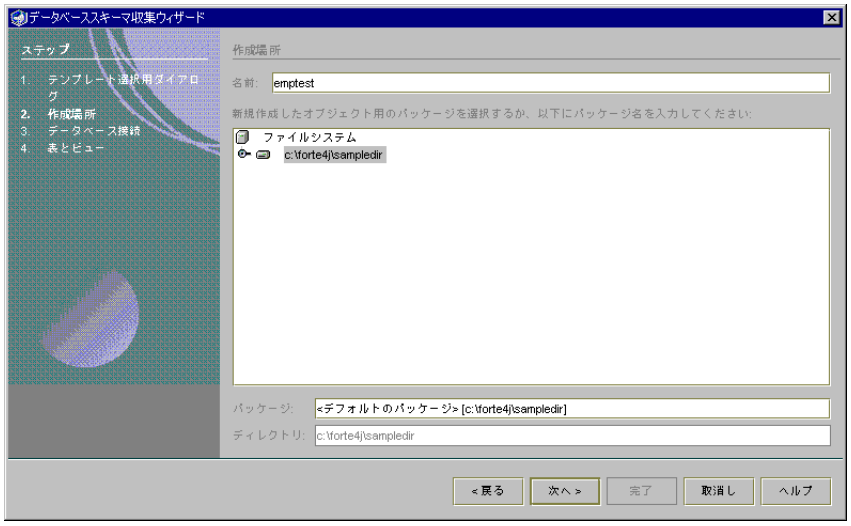

図 4-4　「データベーススキーマ収集ウィザード」、作成場所

3. 「データベース接続」区画 (図 4-5) において、既に確立した接続がある場合には、「既存の接続」メニューからその接続を選択できます。まだ接続を確立していない場合は、「新規接続」の各フィールドに次の情報を入力します。

   - 接続先のデータベースの名前。(データベースがドロップダウンメニューにリストされていない場合、ウィザードを終了し、IDE にドライバをインストールする必要があります。)
   - システムの JDBC ドライバ。
   - データベースの JDBC URL (ドライバ識別子、サーバー、ポート、データベース名)。たとえば、`jdbc:pointbase://localhost:9092/sample` などです。

     JDBC URL の構文は、DBMS (Oracle、Microsoft SQL Server、PointBase のいずれか) と、そのバージョンによって異なります。使用する DBMS の正しい構文については、システム管理者に問い合わせてください。

     図 4-5 の例では、ドライバが PointBase Network Server 、サーバーが `localhost`、ポートが `9092`、データベース名が `sample` に設定されています。実際に指定する URL は、使用するデータソースによって異なります。
   - データベースのユーザー名
   - そのユーザーのパスワード

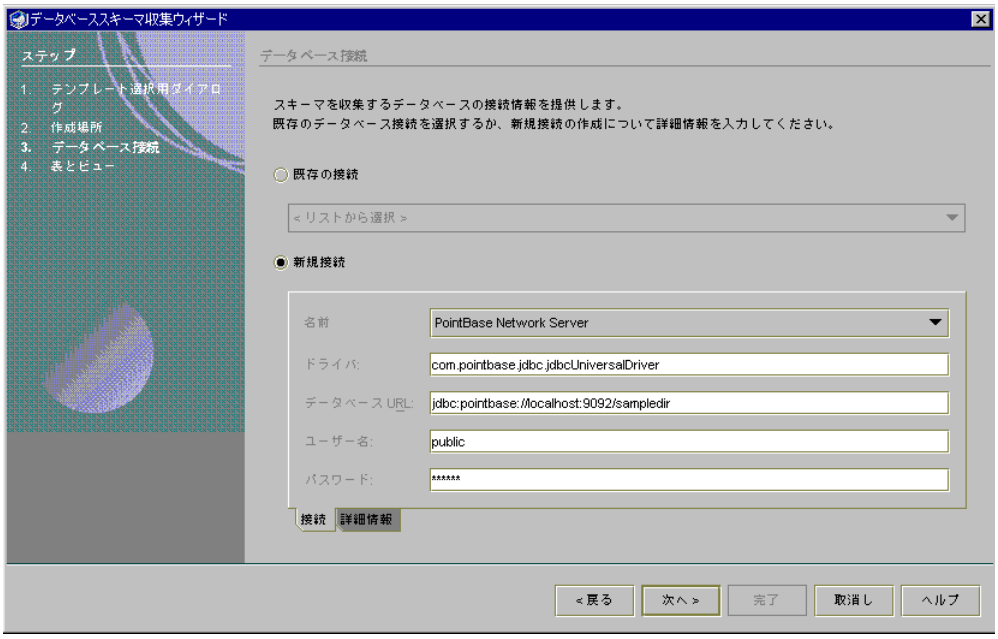

図 4-5　「データベーススキーマ収集ウィザード」、データベース接続

4. 「表とビュー」区画 (図 4-6) において、収集する表とビューを選択し、「完了」をクリックします。

注 - 1 つの表を選択し、その表を外部キーで参照する別の表を選択しなかった場合、選択した表が 1 つであっても両方の表が収集されます。

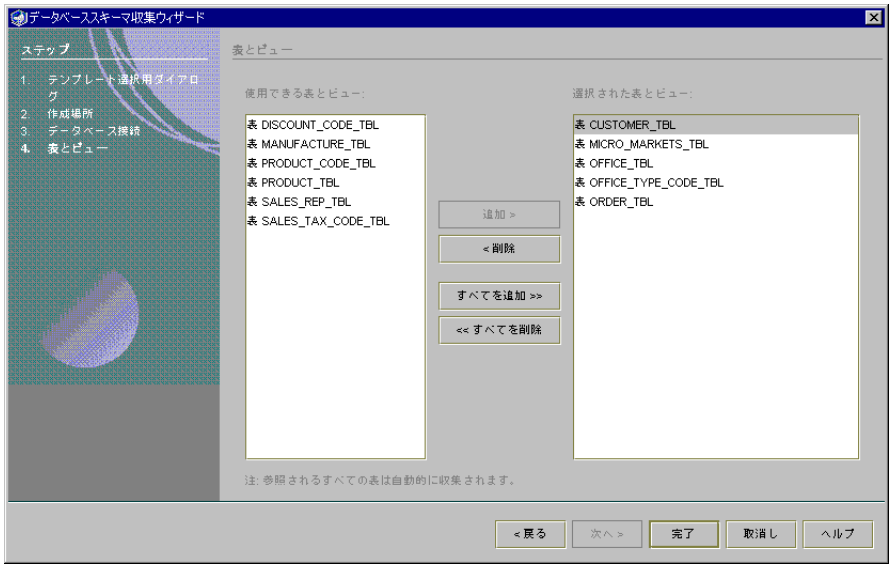

図 4-6　「データベーススキーマ収集ウィザード」、表とビュー

図 4-7 が示すように、データベースとそのスキーマはエクスプローラウィンドウに表示されます。

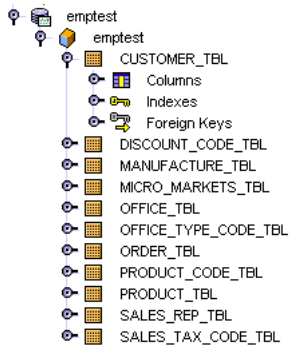

図 4-7　エクスプローラウィンドウのデータベーススキーマ

持続可能クラスの作成

透過的な持続性で Java クラスをデータベーススキーマにマップするには、次の 2 通りの方法があります。

- データベースから Java へのマッピング

データベーススキーマから Java クラスを生成するには、57 ページの「スキーマからの持続可能クラスの生成」 を参照してください。

- Meet-in-the-middle マッピング

既存のスキーマと既存の Java クラスの間にカスタムマッピングを作成するには、63 ページの「既存のクラスをスキーマにマップ」 を参照してください。

スキーマからの持続可能クラスの生成

1. スキーマノードを右クリックし、「Java を生成」を選択します。「Java を生成」ウィザードが表示されます (図 4-10)。このウィザードでは、次の作業を行うことができます。

   - 生成した Java クラスを入れるターゲットパッケージを選択できます。
   - 生成するクラスのオプションをカスタマイズできます。
   - 対応する Java クラスに生成されるデータベース表を選択できます。

2. 「作成場所を選択」区画 (図 4-8) において、リストされているパッケージからパッケージを選択するか、「パッケージ」フィールドに新しいパッケージ名を入力します。

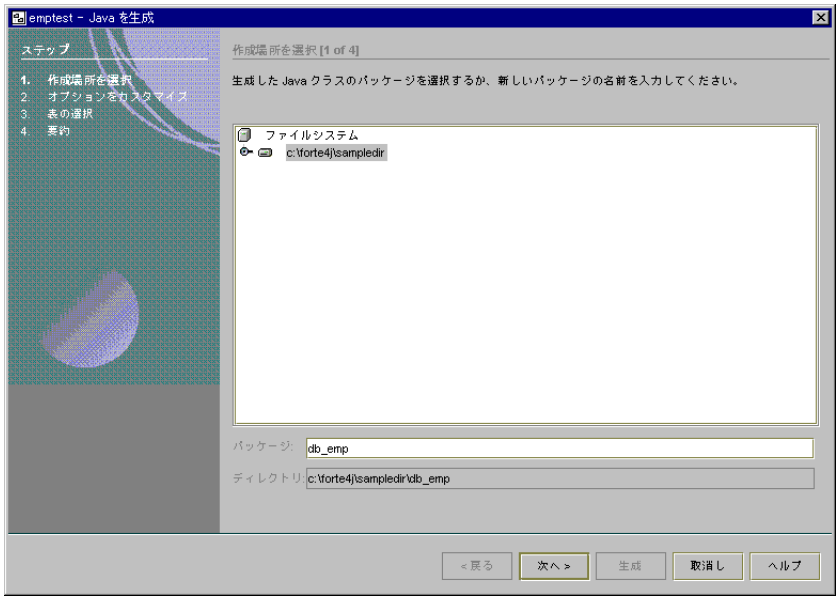

図 4-8　「Java を生成」ウィザード、作成場所を選択

3. 「オプションをカスタマイズ」区画 (図 4-9) において、生成する Java クラスのオプションを選択します。

   単一セッションでオプションを変更することも、将来の Java 生成セッションのためにデフォルトプロパティとして保存することも可能です。設定できるオプションについては、表 4-2 を参照してください。

注 - これらのデフォルトオプションは、「ツール」>「オプション」を選択し、「透過的な持続性」の下の「Java 生成オプション」を選択して表示されるプロパティシートで変更できます。78 ページの「Java 生成オプション」 を参照してください。

関係フィールドの命名規則を設定するには、「関係ネーミングポリシー」の「...」フィールドをクリックします。80 ページの「関係ネーミングポリシー」 を参照してください。

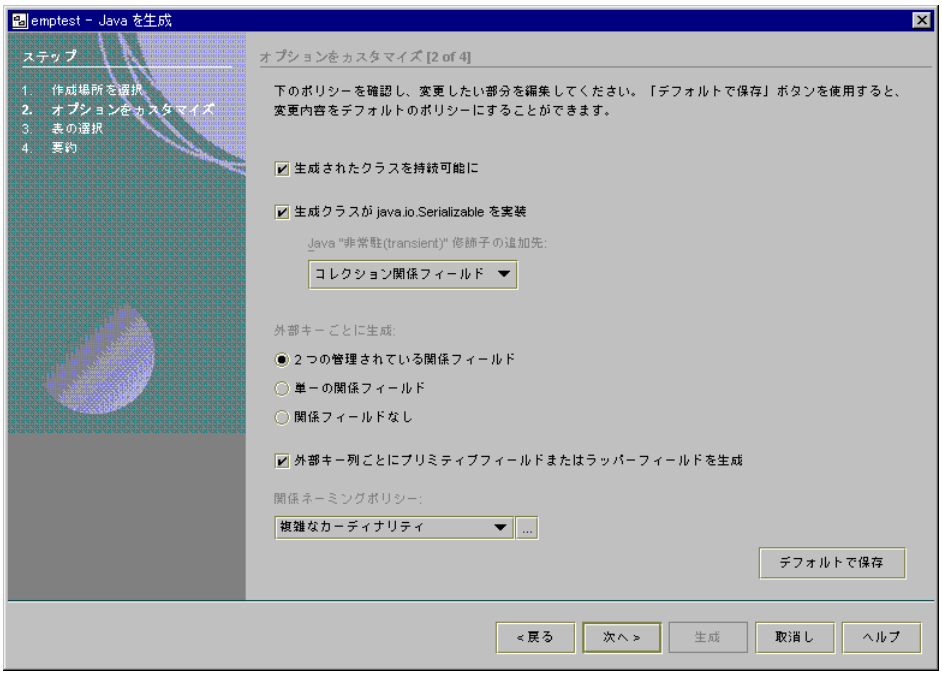

図 4-9　「Java を生成」ウィザード、オプションをカスタマイズ

4. 「表の選択」区画 (図 4-10) において、対応する Java クラスを生成する表とビューを選択します。

   表とビューを個々に選択するか、「すべての表を追加」を選択して一度にすべての表を選択するか、「すべてのビューを追加」を選択して一度にすべてのビューを選択できます。

   「選択できる表とビュー」に、利用できる表がリストされます。「追加」ボタンを使用して、マップする表を指定します。クラス名はクリックして編集することができます。

   「Java クラス」列の <**結合表**> という表示は、結合表で結合された 2 つのクラスの間に多対多の関係を作成し、結合表自身のクラスは作成しないことを示しています。結合表のクラスを作成したい場合は、<**結合表**> をクリックし、ドロップダウンメニューからクラスを選択するか、クラス名を入力します (ただし、結合表に主キーがあることが条件になります)。この操作で、他の 2 つのクラスと、結合表にマッピングされたクラスとの間に 1 対 n の関係が作成されます。関係を作成せずに 2 つの表をマッピングするには、リストから結合表を削除します。

透過的な持続性で生成されるクラスは、主キーのある表にマップされたものだけです。主キーのない表は、「選択できる表とビュー」には表示されません。例外として、主キーのない結合表は表示されますが、この表は主キーのある 2 つの表を結び付ける機能しかないため、クラスを直接マップすることはできません。

さまざまな場所にあるクラスを保存する場合、ある場所でクラスを生成し、ウィザードを再実行し、「作業場所を選択」パネルで新しい場所を選択します。

複数のクラスを同一の表またはビューにマップするには、ウィザードを複数回実行し、名前をカスタマイズするか、ファイルを異なる場所に保存します。

図 4-10 「Java を生成」ウィザード、表の選択

関係クラスの生成

表とビューは、さまざまな組み合わせで選択できます。それぞれの組み合わせの結果を 表 4-1 に示します。このリストでは、2 つの表を 「A」、「 B 」、結合表を 「 AB 」 と仮定しています。

表 4-1　関係クラスの生成

A	B	AB	結果
追加	追加	結合表	クラス A、B が生成され、2 つのコレクション関係、A 対 B、および B 対 A がそれに伴います。
追加	追加	Java クラス名	クラス A、B、AB が生成され、4 つの関係フィールド (A 対 AB、AB 対 A、B 対 AB、AB 対 B) がそれに伴います。
追加	追加	追加なし	A と B は生成されますが、A または B の関係フィールドは生成されず、表 AB は実行時に使用されません。
追加	追加なし	結合表	B を追加するか、AB の名前を変更して、結合表のクラスを生成可能にしない限り、ウィザードを終了できません。
追加	追加なし	Java クラス名	AB とのコレクション関係および B のプリミティブフィールドとともにクラス A が生成されます。
追加	追加なし	追加なし	B との関係を持たないクラス A が生成されます。関係フィールドは生成されません。
追加なし	追加なし	結合表	結合表 AB は、A および B との不完全な関係を保持します。A と B の両方を追加するか、結合表名を変更してクラスにするまで、Java クラスを生成できません。
追加なし	追加なし	Java クラス名	外部キー列のプリミティブ型の持続フィールドとして生成された外部キーとともに AB が生成されます。

5. 「生成」をクリックして、選択した各表の持続可能クラスを作成し、すべてのフィールドと関係をマップします。

   「オプションをカスタマイズ」区画において、「生成されたクラスを持続可能に」チェックボックスをオフにした場合、持続可能ではない Java ファイルが生成されます。

   「表の選択」区画で表とビューが選択された後、IDE は、不完全な関係をチェックします。生成されるすべてのクラスが「要約」区画にリストされます。リストには、不完全な関係を含むクラスも表示されます。

   関係で結び付けられている 2 つの表のうち、どちらか一方の表をマップの対象から除外している場合は、その関係がマップされないことが通知されます。「戻る」ボタンをクリックして、前のパネルに戻り、マッピングを修正することができます。代わりに「生成」ボタンをクリックし、関係を無視してクラスを生成することもできます。

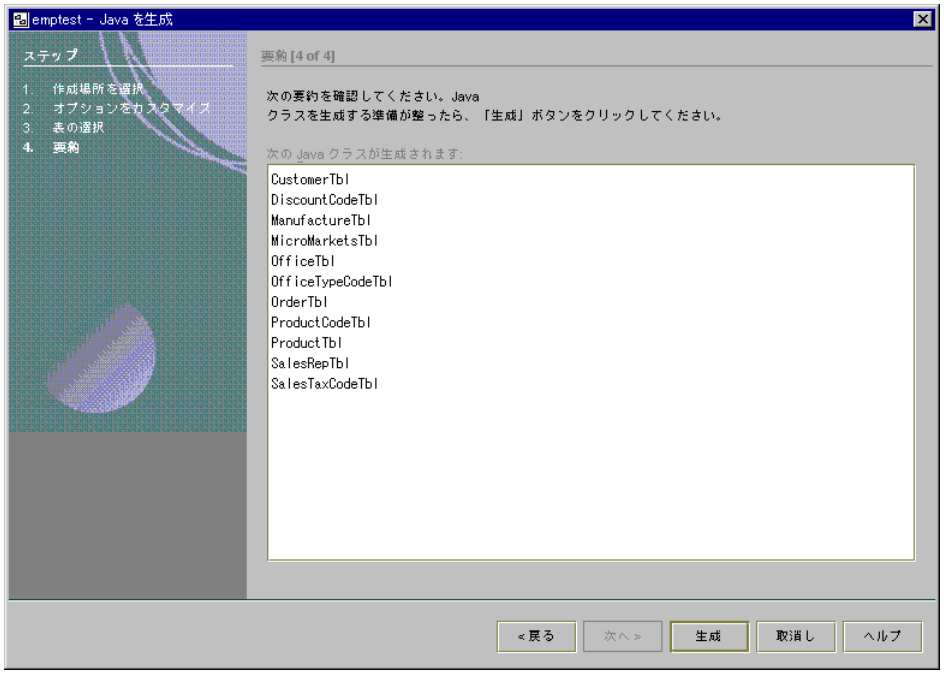

図 4-11 「Java を生成」ウィザード、要約

マップしたクラスをカスタマイズする手順については、65 ページの「持続可能クラスをマップする」 を参照してください。

注 - 2 つのクラスを同じ主表にマップしたい場合は、「Java を生成」ウィザードを 2 回使用し、別々の名前でこれらのクラスを生成します。このようにすることで、名前が別々の 2 つのクラスを同じ表にマップすることができます。

既存のクラスをスキーマにマップ

ここでは、透過的な持続性を使用してマッピングをカスタマイズしたり、既存のオブジェクトモデルのマッピングを作成する方法について説明します。

Java クラスをデータベーススキーマにマップするには、事前に次のことを確認する必要があります。

- データベーススキーマがエクスプローラのファイルシステムに収集され、マウントされている。

この作業の手順については、52 ページの「スキーマを収集する」 を参照してください。

- マッピングしているクラスと関係するどのクラスも持続可能クラスである。(クラスそのものはウィザードが起動されると自動的に持続可能になります。)

この作業の手順については、64 ページの「クラスを持続可能クラスにする」 を参照してください。

- マップしたいフィールドが、すべて持続フィールドとして設定されている。

この作業の手順については、64 ページの「フィールドを持続フィールドにする」 を参照してください。

「データベースにマップ」コマンドを再実行すると、既存のマッピングを編集することができます。それまでの設定内容で「データベースへマップ」ウィザードが再表示されます。

代わりに、「プロパティ」ウィンドウで個々のプロパティを編集して、マッピングを部分的に設定したり編集することもできます。「プロパティ」ウィンドウでは、すべてのマッピング情報と持続情報にアクセスできますが、「データベースにマップ」ウィザードを使用すると、クラスとフィールドのグループを一度に参照したり編集できるため、マッピングモデルの全体像を把握しやすくなります。

クラスを持続可能クラスにする

クラスをデータベース表にマップするには、そのクラスに加えて、そのクラスに関係しているすべてのクラスを、事前に持続可能クラスにしておく必要があります。「データベースへマップ」ウィザードでは、マップするために選択したクラスが自動的に持続可能クラスになりますが、それ以外のクラスについては、手作業による変換が必要です。

複数のクラスを選択し、まとめて変換することができます。互いに関連のあるクラスを変換する際には、必ずこの方法で行なってください。この方法によってすべての関係フィールドを自動的に持続にします。

そのためには、変換したいクラスをそれぞれマウスの右ボタンでクリックし、「持続可能に変換」を選択します。一度にクラスをまとめて変換するには、「Ctrl」キーを押しながらクラスを選択することにより、複数のクラスを選択します。マウスの右ボタンでクリックし、「持続可能に変換」を選択します。

クラスを持続可能から元に戻す

逆に、持続可能クラスを非持続可能にするには、クラスをマウスの右ボタンでクリックし、「持続可能プロパティを解除」を選択します。この操作により、クラスからすべてのスキーママッピングと持続プロパティが削除されます。

「持続可能に変換」を使用してクラスを再変換すると、持続プロパティがデフォルト値に復元されます。47 ページの「Meet-in-the-middle マッピング」 で説明されているように、クラスをデータベーススキーマにマップする必要があります。

フィールドを持続フィールドにする

クラスを持続可能にすると、持続と解釈できるすべてのフィールドは自動的に持続になります。任意のフィールドを追加して、そのフィールドから持続データにアクセスする場合は、追加したフィールドを個別に持続にする必要があります。

▼ フィールドを持続フィールドにする。

1. エクスプローラウィンドウでクラスを展開し、その中の「フィールド」ノードをさらに展開し、持続フィールドにしたいフィールドを選択します。

   3 角形の印が付いているものは持続フィールド、3 角形の印と矢印が付いているものは関係フィールド、丸印が付いているものは非持続フィールドです。図 4-12 を参照してください。

2. プロパティウィンドウで、「持続」プロパティをクリックしてドロップダウンメニューを表示し、「True」を選択します。

   ドロップダウンメニューで「False」を選択すると、非持続フィールドに戻すことができます。

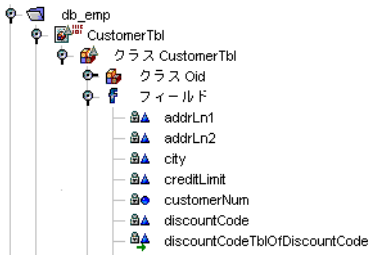

図 4-12 「持続」フィールド

持続可能クラスをマップする

▼ 「データベースへマップ」ウィザードを使用してクラスを表にマップする。

1. マップするクラスをマウスの右ボタンでクリックし、「データベースにマップ」コマンドを選択します。「データベースへマップ」ウィザードが表示されます (図 4-13)。

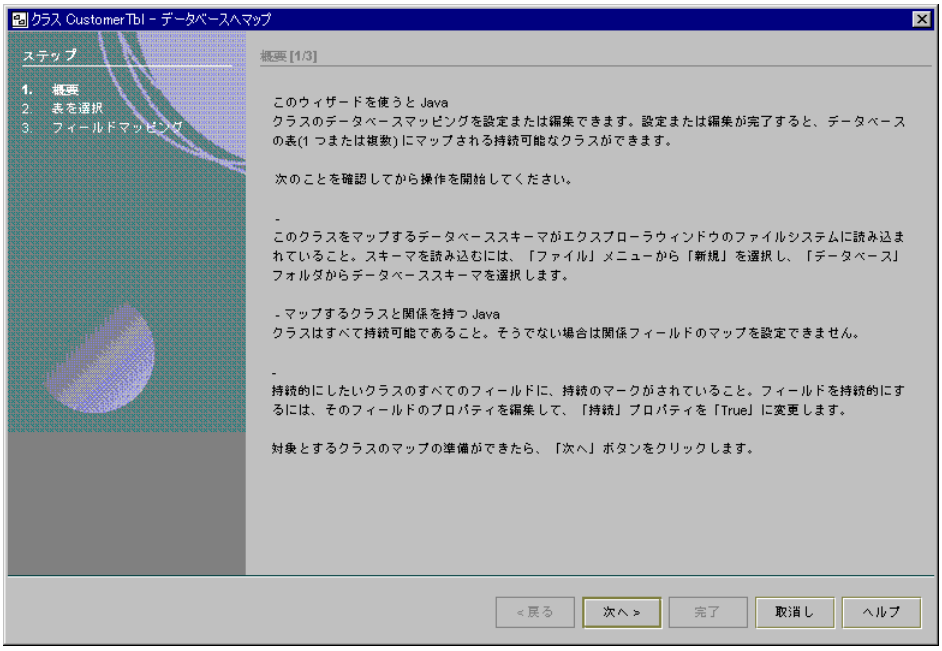

図 4-13 「データベースへマップ」ウィザード、概要

2. 概要に記述されている確認作業を終えている場合は、「次へ」をクリックします。「表を選択」区画が表示されます (図 4-14)。まだ確認を行なっていない場合は、「取消し」をクリックし、確認作業を終えてからウィザードを再起動します。

3. 「主表」コンボボックスから主表を選択するか、「ブラウズ」をクリックして「主表を選択」ダイアログを開きます。

図 4-14 「データベースへマップ」ウィザード、表を選択

4. 「主表を選択」ダイアログ (図 4-15) を開いたら、スキーマを見つけて展開し、表を見つけます。表を選択し、「了解」をクリックします。

   主表として選択する表は、クラスに最も適合する表にしてください。

   主表として選択できるのは、主キーのある表だけで、マップするクラスに最も適合する表でなければなりません。

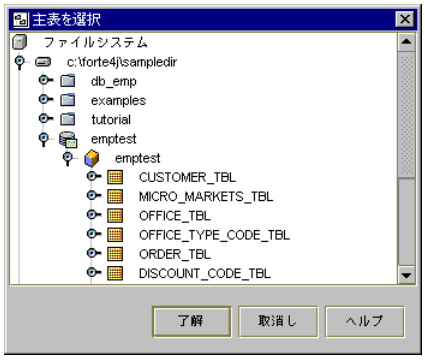

図 4-15 「主表を選択」ダイアログ

5. 主表を設定した後は、「追加」をクリックして「マップされた二次表の設定」ダイアログ (図 4-16) を表示し、複数の二次表をマッピングします。

   二次表によって、主表にはない列に、持続可能クラスのフィールドをマッピングすることができます。たとえば、二次表として DEPARTMENT (部署) 表を追加し、Employee (従業員) クラスに従業員の配属先の部署名を組み込むことができます。二次表は、あるクラスが関係フィールドによって別のクラスと関連付けられている関係とは異なります。二次表マッピングでは、同じクラス内のフィールドは 2 つの異なる表にマッピングされます。二次表を使用すると、主表にはない列をクラスのフィールドに直接マップすることができます。このダイアログを使用すると、二次表を選択し、それらが主表にリンクされている様子を表示できます。

   二次表は、主表と二次表の両方における値が同じである行を持つ 1 個以上の列によって主表と関係していなければなりません。通常、この関係は、表の間の外部キーとして定義されます。ドロップダウンメニューから二次表を選択すると、主表と二次表を結び付ける外部キーの存在がチェックされます。外部キーが存在する場合は、デフォルト時に参照キーとして表示されます。

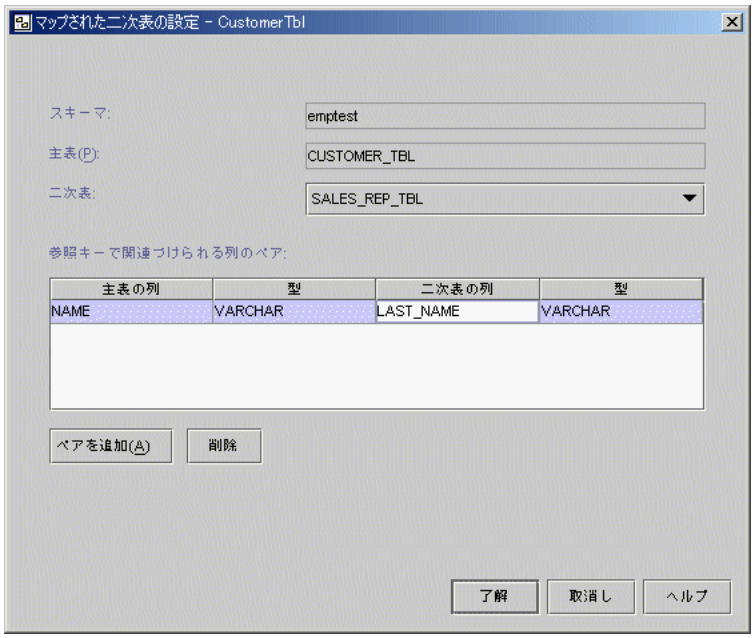

図 4-16 マップされた二次表の設定

a. コンボボックスから二次表を選択します。

二次表が選択されると、透過的な持続性は主表と二次表の間に外部キーが存在するかどうかを確認します。外部キーが存在する場合、デフォルトの参照キーとして表示されます。外部キーが存在しない場合、<列を選択> という文字列が表示されるため、参照キーを設定する必要がありません。

b. 参照キーを設定するには、<列を選択> をクリックしてドロップダウンメニューから列を選択します。

主列を選択すると、二次列の選択肢は主列と互換のある型の列のみに限定されます。互換性のある列が存在しない場合、二次列の選択肢に「<互換列なし>」 がという文字列が含まれます。また、選択した二次表の列と互換性のない主表の列を選択しなおすと、二次表の列の表示が「<列を選択>」に戻ります。

論理的に関係している列の組み合わせがない場合、つまり、論理的な参照キーがない場合は、二次表選択が適切であるかどうか再考してください。

「ペアを追加」を選択すると、キーと列のペアが追加されるため複数のペアから複合キーを設定できます。

6. 「了解」をクリックし、設定した内容を保存します。

7. 「データベースへマップ」ウィザードで「次へ」をクリックして、「フィールドマッピング」区画を表示します (図 4-17 を参照してください)。

「フィールドマッピング」区画は、クラスのすべての持続フィールドおよびそれらのマッピング状態を表示します。フィールドを個別にマップする場合は、該当するフィールドのドロップダウンメニューから、そのフィールドにマップしたい列を選択します。また、「自動マップ」を選択すると、すべてのフィールドを一括してマップすることができます。この機能では、それぞれのフィールドに、論理的に最も適切と思われる列が自動的にマップされます。したがって、それまでのマッピング内容は影響を受けません。

クラスのフィールドがリストされていない場合は、そのクラスが持続可能でない可能性があります。クラスが持続可能に設定された後に追加されたフィールドや、「プロパティ」ウィンドウで「持続」が「False」に設定されている場合に、この状況になる

ことがあります。このようなフィールドを持続可能にするには、「完了」をクリックしてウィザードを終了してからフィールドの「持続」プロパティを「True」に変更してください。

ドロップダウンメニューに表示されない列をフィールドにマップしたい場合は、「戻る」をクリックして前の区画に戻り、その列を含んでいる表を二次表として追加します。

マップされているフィールドに対してのみ「<アンマップ>」を選択できます。「Shift」キーや「Ctrl」キーを押しながらフィールドを選択すると、複数のフィールドを一度にアンマップすることができます。1 つのフィールドだけをアンマップしたい場合は、そのフィールドのドロップダウンメニューから「<アンマップ>」を選択します。

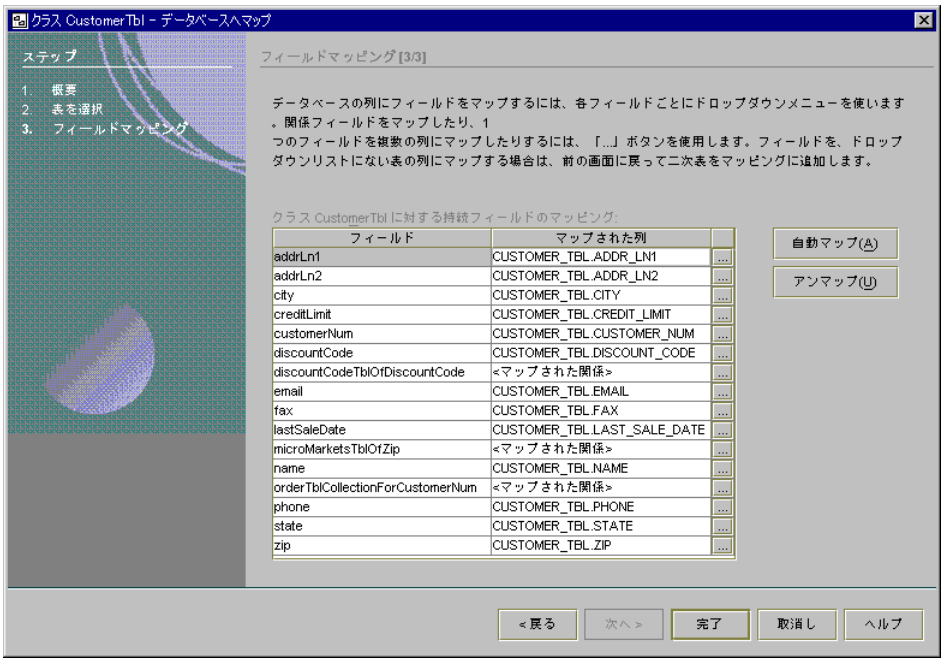

図 4-17 「データベースへマップ」ウィザード、フィールドマッピング

a. フィールドを複数の列にマップするには、「フィールドマッピング」区画の適切なフィールドの「...」ボタンをクリックし、「フィールドを複数列にマップ」ダイアログを表示します (図 4-18 を参照してください)。

このダイアログで、「使用可能な列」のリストから列を追加します。リストの列は、このクラスにマップした表からのものです。列の順序を変更するには、「上へ移動」または「下へ移動」をクリックします。

マッピングしたい列が見つからない場合は、マッピングに二次表を追加するか、選択している主表を変更する必要があるかもしれません。列がまったく列挙されていない場合は、主表がまだマッピングされていないか、列のない表がマップされています。

フィールドを複数の列にマップした場合、すべての列がリストの先頭列の値で更新されます。したがって、列の中のいずれかの値が透過的な持続性アプリケーション外で変更された場合、その値は最初の列が変更された場合にのみ読み取られます。データベースに値を書き込むと、これらの列の値の不一致が解消されます。

また、これらの列に複数のフィールドをマップした場合、マッピングが部分的に重複しないことを確認する必要もあります。

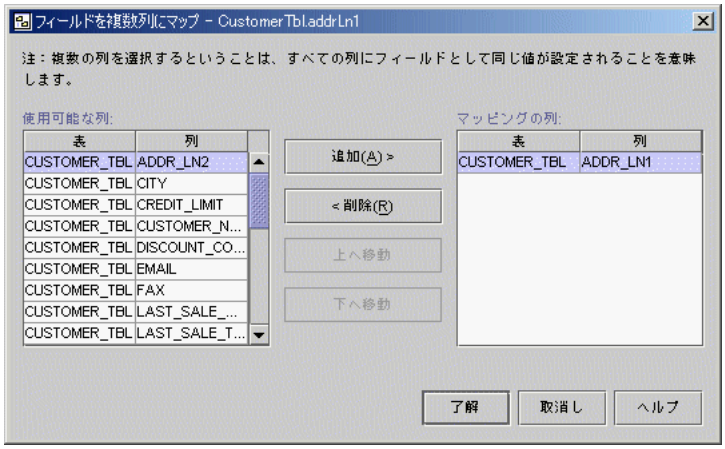

図 4-18 「フィールドを複数列にマップ」ダイアログ

次の 3 つの例で重複を考えてみましょう。

- 列 A および B にマップされたフィールド Aと、列 B にマップされたフィールド B。マッピングが部分的に重複しているため、この例では、コンパイル時に妥当性検査エラーが発生します。
- 列 A にマップされたフィールド A と、列 B にマップされたフィールド B。重複がないため、このマッピングは許可されます。

- 列 A にマップされたフィールド A と、列 A および B にマップされたフィールド B。マッピングが完全に重複しているため、このマッピングは許可されます。

b. 「了解」をクリックし、設定した内容を保存します。

関係フィールドをマップする

データベース表同士が外部キーで結び付けられている場合は、Java クラスの参照機能にも同じ関係を反映させたほうが好都合です。「関係フィールドのマップ」ダイアログを使用すると、外部キーの関係を Java クラスの参照フィールドに対応付けることができます。

c. 関係フィールドをマップするには、「フィールドマッピング」関係フィールドのドロップダウンメニューの隣りにある区画で「...」ボタンをクリックして、「関係フィールドのマップ」ウィザード (図 4-19) を表示します。

「データベースへマップ」ウィザード以外で「関係フィールドのマップ」ウィザードを使用するには、エクスプローラで関係フィールドをクリックし、そのプロパティを表示して、「マッピング」プロパティを編集します。

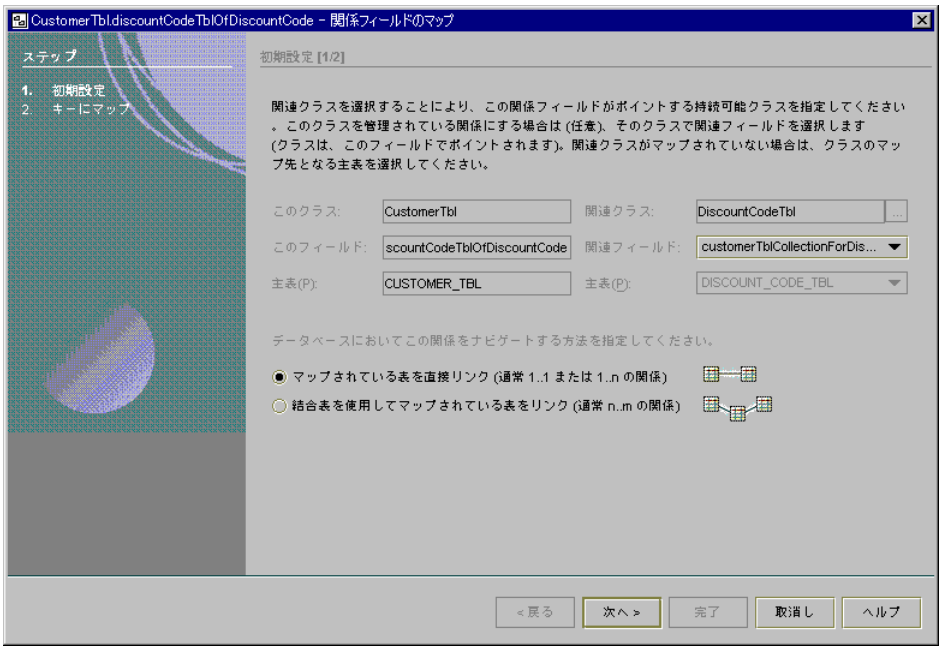

図 4-19 関係フィールドのマップ、初期設定

- この区画で、関連クラスが設定されていることを確認します。関連クラスが設定されていない場合は、設定を行なってください。選択したいクラスが持続可能でない場合、「取消し」をクリックしてウィザードを終了し、クラスを持続可能に変換し、ウィザードを再起動する必要があります。
- 関連フィールド (存在する場合) が適切であること、および関連クラスに主表が設定されていることを確認します。

注 - 論理関連フィールドがある場合、主表を選択することをお奨めします。そうすることで、管理されている関係が作成されます。

- 表を直接リンクするか、結合表を介してリンクします。

d. 関係が 1 対 1 または 1 対多である場合、表を直接リンクします。「次へ」をクリックして、「関係フィールドのマップ」ウィザードの「キーにマップ」区画を表示します (図 4-20 を参照してください)。

この区画は、次の要素を表示します。

- 既存のマッピング (1 つだけ存在し、初期設定で変更がなかった場合)
- デフォルトマッピング (既存のマッピングが存在しないか、マッピングが既に有効でない場合)

  ウィザードは、既存の外部キーに基づいて、2 つの関連クラス間で最も論理的なキーと列のペアを判定します。外部キーが存在しない場合、ローカル列と外部列を選択することにより、キーと列のペアを作成する必要があります。各ペアの列には、同じ値が含まれているものとみなされます。

  複合キーを作成するには、「ペアを追加」ボタンを使用して、キーと列のペアを追加します。

  「完了」ボタンが使用できない場合、キーと列のペアを選択する必要があります。

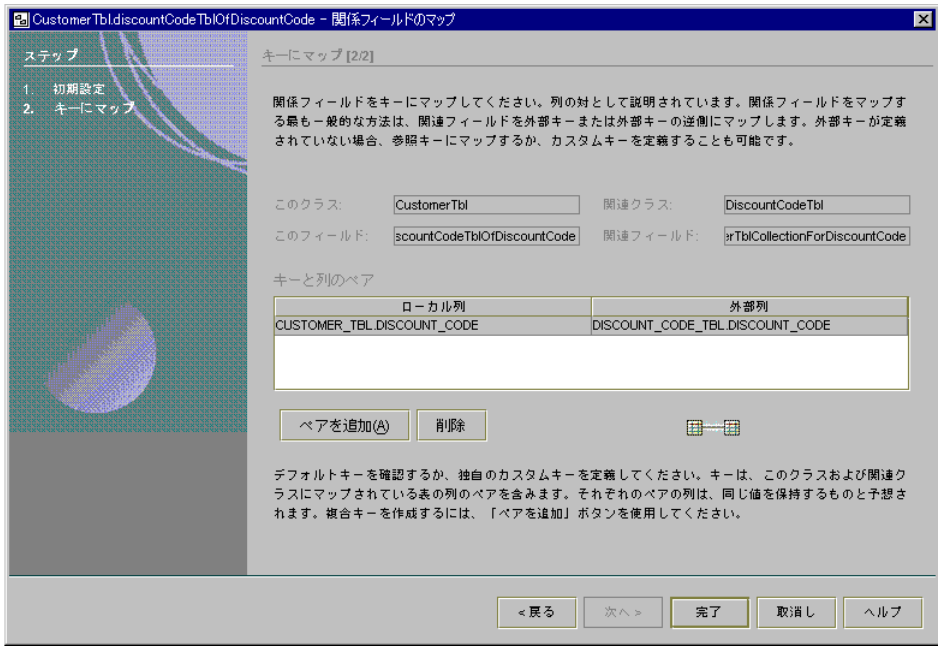

図 4-20 関係フィールドのマップ、キーにマップ

e. 使用している関係が多対多の場合、結合表を介して表をリンクします。「次へ」をクリックして、「キーにマップ: ローカル列から結合表へ」区画が表示します (図 4-21 を参照してください)。

この区画は、次の要素を表示します。

- 関係の最初のクラスとフィールド
- フィールド間の関係を作成するための結合表

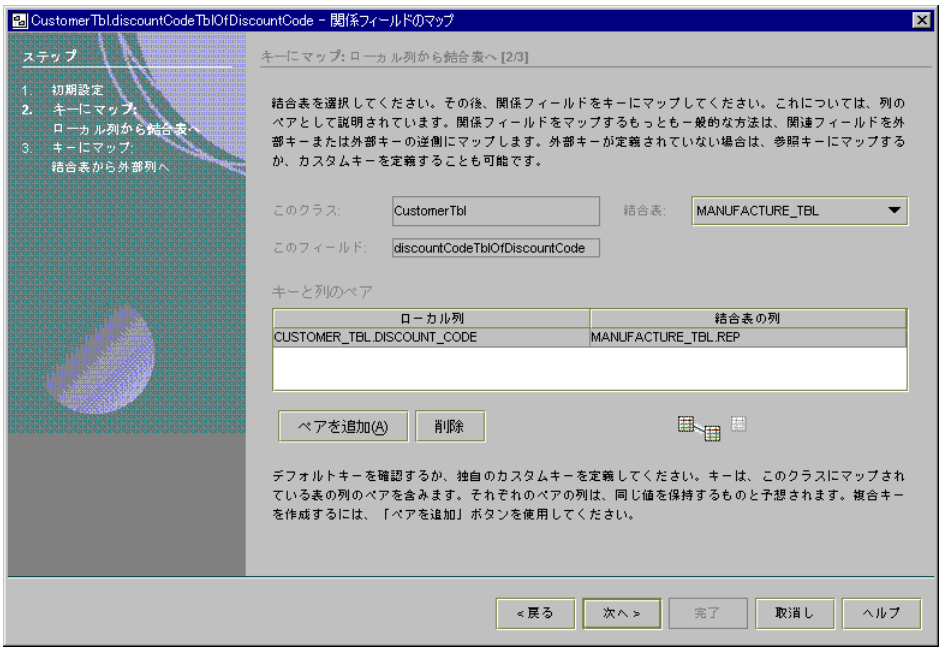

図 4-21 関係フィールドのマップ、キーにマップ: ローカル列から結合表へ

- フィールド結合表と関連クラスのマップ先となる表の間のキーと列のペア

  この区画では、結合表を選択した後、関係フィールドをキーにマップします。このマップは、「このクラス」のマップ先となる表と結合表の間の関係のみです。結合表が存在しない場合、直前の区画に戻り、「マップされている表を直接リンク」を選択します。

  クラスのマップ先となる 2 つの表の間に配置する結合表を選択します。ウィザードは、結合表と「このクラス」のマップ先となる表の間の最も論理的なキーと列のペアを決定します。

  表の間に外部キーが存在する場合、ウィザードはその外部キーをデフォルトキーと列のペアとして使用します。外部キーが存在しない場合、結合表を「このクラス」のマップ先となる表にナビゲートする列のペアを選択することにより、キーを作成する必要があります。各ペアの列には、同じ値が含まれているものとみなされます。

  複合キーを作成するには、「ペアを追加」ボタンを使用して、キーと列のペアを追加します。

  「次へ」ボタンが使用できない場合、結合表を選択するか、最低 1 つのキーと列のペアを選択する必要があります。

f. 「次へ」をクリックして、「キーにマップ: 結合表から外部列へ」区画を表示します。

この区画では、直前の区画で選択した結合表に二次表を関連付けます。

ウィザードは、結合表と関連クラスのマップ先となる表の間の最も論理的なキーと列のペアを決定します。

表の間に外部キーが存在する場合、ウィザードはその外部キーをデフォルトキーと列のペアとして使用します。外部キーが存在しない場合、結合表を関連クラスのマップ先となる表にナビゲートする列のペアを選択することにより、キーを作成する必要があります。各ペアの列には、同じ値が含まれているものとみなされます。

複合キーを作成するには、「ペアを追加」ボタンを使用して、キーと列のペアを追加します。

「完了」ボタンが使用できない場合、有効なキーと列のペアを選択する必要があります。

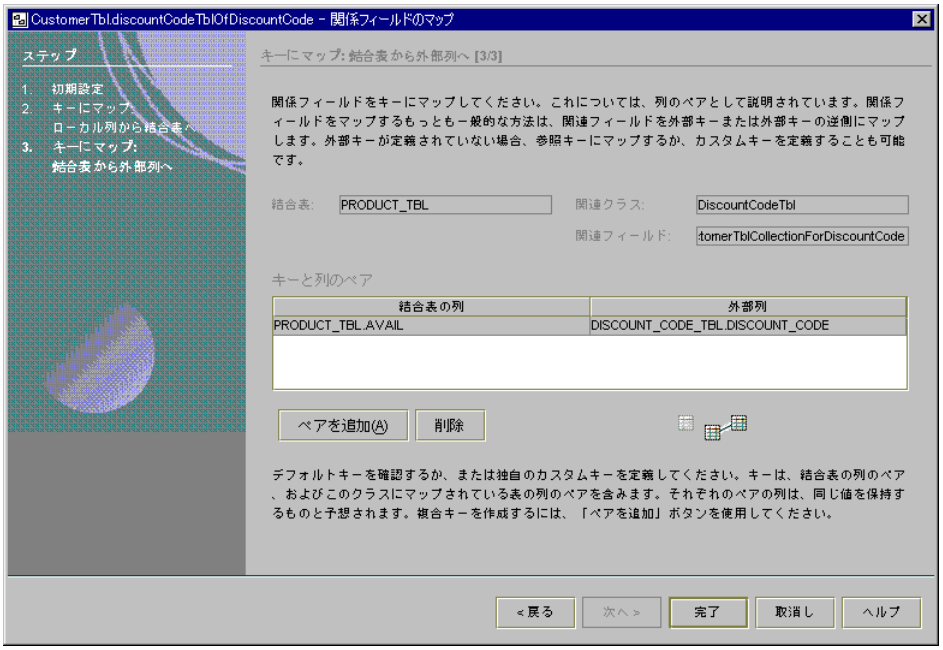

図 4-22 関係フィールドのマップ、キーにマップ: 結合表から外部列へ

g. 「完了」をクリックして、「データベースへマップ」ウィザードの「フィールドマッピング」区画に戻ります。

8. 「完了」をクリックして、「フィールドマッピング」区画を閉じ、Java クラスをデータベーススキーマにマップします。

オプションとプロパティの設定

透過的な持続性のウィザードではなくノードのプロパティシートを使用すると、次の項目を設定できます。

- 持続クラスの継続的な検査
- Java 生成オプション
- 関連フィールドの命名規則
- 持続可能なクラスとフィールドのプロパティ

持続クラスの継続的な検査

このプロパティシートを開くには、「ツール」>「オプション」を選択し、「透過的な持続性」ノードを選択します。

「有効な Java の変更内容」プロパティを True に設定すると、持続可能クラスのソースコードの変更内容が検査されるため、コンパイルエラーを回避できます。クラスが変更されてまだその変更が有効でない場合、「質問」ダイアログが表示され、次の 3 つの選択肢が提示されます。

- 「保存」。透過的な持続性は変更内容を維持し、コンパイルエラーが生じないように、ファイルにその他の変更を加えます。
- 「破棄」。変更内容を破棄します。
- 「無視」。なにも実行しません。「無視」を選択した場合、コンパイル時に問題が生じることがあります。

False に設定すると、「無視」を選択した場合と同様の結果が生じます。

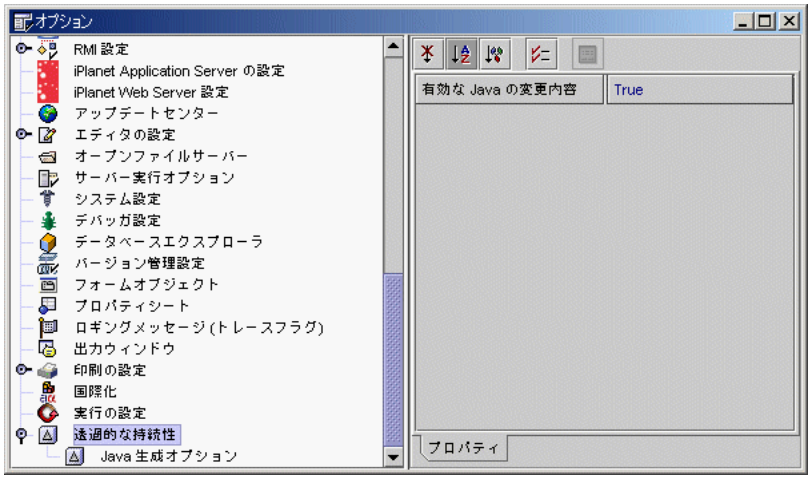

図 4-23 「有効な Java の変更内容」プロパティ

Java 生成オプション

Java 生成オプションは、「Java を生成」ウィザードが Java クラスとマッピング情報を生成する際に使用するプロパティを指定します。これらのプロパティを上書きするには、「Java を生成」ウィザードの 2 枚目のパネルを使用するか、「ツール」>「オプション」を選択した後、「透過的な持続性」の下の「Java 生成オプション」を選択します。Java 生成プロパティについては、表 4-2 を参照してください。プロパティ

シートについては、図 4-24 を参照してください。

表 4-2　Java 生成プロパティ

プロパティ	説明
持続可能に	生成された Java クラスはデータベースにマップされます。False に設定した場合、使用している表のプレーンなオブジェクトラッパーが取得されます。これらのラッパーには、透過的な持続性の機能がありません。
直列化できるように実装	True に設定した場合、生成された Java クラスは、`java.io.Serializable` を実装します。その結果、クラスの直列化が可能となるため、クライアントとサーバーなど異なる階層間でストリームに書き込むことができます。
Java 非常駐修飾子	クラスが `java.io.Serializable` を実装している場合、非常駐修飾子を特定のフィールドに追加できます。このプロパティを使用すると、次の非常駐修飾子を追加できます。 • コレクション関係フィールド。単一の参照は、所有オブジェクトとともに直列化されます。 • すべての関係フィールド。単一の参照かコレクションかに関係なく、関連オブジェクトは所有で直列化されません。 • フィールドなし。オブジェクトグラフの完全な closure が直列化されます。
FK のプリミティブ	各外部キー (FK) 列のプリミティブまたはラッパーフィールドを生成するかどうかを指定します。関係とともにプリミティブフィールドを生成する場合、実行時に暗黙で指定できます。
関係ネーミング	関係フィールドの名前の作成に使用するポリシー。「単純なカーディナリティ」を指定すると、関係フィールドのカーディナリティに基づいて 2 つの規則が提供されます。「複雑なカーディナリティ」を指定すると、フィールドのカーディナリティ、およびフィールドが示す外部キーの側に基づいて、5 つの規則が提供されます。個々の規則は編集可能です。「...」ボタンをクリックすると、「関係ネーミングプロパティエディタ」が開きます。
関係の型	外部キーごとに生成される関係の種類 • 「管理されている」は、関係双方からナビゲートおよび更新が可能です。 • 「単一フィールド」は、外部キーを含む表に対応するクラスからのみナビゲートと更新が可能です。 • 「なし」を選択した場合、関係フィールドは生成されません。

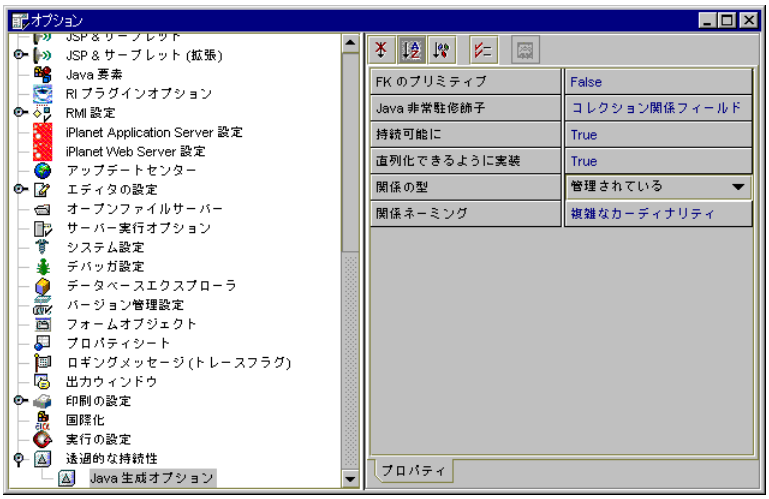

図 4-24 Java 生成オプション

関係ネーミングポリシー

外部キーを保持する表の Java クラスを生成する場合、特殊な関係フィールドを作成します。これらのフィールドは外部キーの列のペアにマップされるため、フィールドの名前は、各フィールドの名前を結合することによって作成されます。デフォルト設定のままにしておくことをお勧めしますが、これらのフィールドのネーミングポリシーをカスタマイズすることも可能です。

「関係ネーミングエディタ」を使用すると、ポリシーの個々の規則を編集できます。

▼ エディタを開く

1. 「ツール」 > 「オプション」を選択します。
2. 「透過的な持続性」ノードを展開し、「Java 生成オプション」を選択します。
3. 「関係ネーミング」プロパティを選択し、ドロップダウンメニューから「単純なカーディナリティ」または「複雑なカーディナリティ」を選択します。
4. 「...」ボタンをクリックします。

   この操作により、「関係ネーミングプロパティエディタ」が開きます (図 4-25 を参照してください)。

注 - また、「Java を生成」ウィザードの 2 枚目のパネルからエディタを開くには、「関係ネーミングポリシー」をドロップダウンメニューから選択し、「...」ボタンをクリックします。

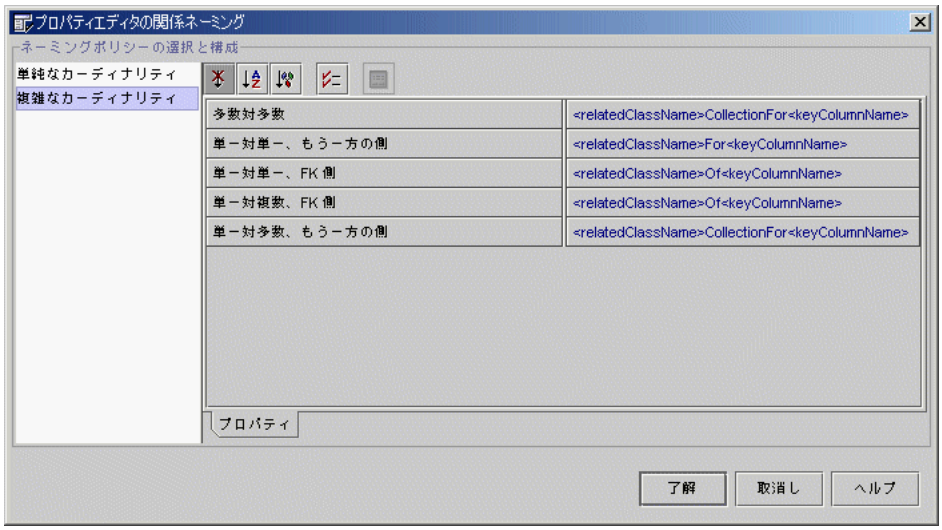

図 4-25 関係ネーミングエディタ

「単純なカーディナリティ」を選択した場合、2 つの規則が表示されます (表 4-3 を参照してください)。

表 4-3 単純なカーディナリティのネーミングポリシー

規則	説明
多数側	コレクション関係を表すフィールドのデフォルト名です。
単一側	非コレクション関係を表すフィールドのデフォルト名です。

「複雑なカーディナリティ」を選択すると、5 つの規則が表示されます (表 4-4 を参照してください)。

表 4-4　複雑なカーディナリティのネーミングポリシー

規則	説明
単一対複数、FK 側	1:n 関係の非コレクションフィールドのデフォルト名です。
単一対複数、もう一方の側	1:n 関係のコレクションフィールドのデフォルト名です。
単一対単一、FK 側	1:1 関係の外部キー側のフィールドのデフォルト名です。
単一対単一、もう一方の側	1:1 関係の非外部キー側のフィールドのデフォルト名です。
多数対多数	n:m 関係のフィールドのデフォルト名です。

名前を編集するには、プロパティの値をクリックし、フィールドに入力します。編集を簡単に行うには、フィールドの「...」ボタンをクリックします。この操作により、ネーミングポリシー規則のプロパティエディタが開きます (図 4-26 を参照してください)。

「了解」をクリックすると、編集内容が保存されます。

図 4-26　ネーミングポリシー規則のプロパティエディタ

プロパティ規則エディタでネーミングポリシーを編集するには、「使用可能なタグ」ドロップダウンメニューからタグを選択し、「タグを挿入」をクリックするか、「規則」テキストフィールドに手動で文字列を入力します。

ドロップダウンメニューに表示されるタグについては、表 4-5 を参照してください。

表 4-5　関係ネーミングタグ

タグ	説明
<thisClassName>	このフィールドが属するクラスの名前を使用します。
<relatedClassName>	「もう一方の側」のクラス、つまりこの関係がポイントするクラスの名前を使用します。
<keyColumnName>	外部キーの列の名前を使用します。
<thisTableName>	このクラスがマップする表の名前を使用します。
<relatedTableName>	関連クラスのマップ先となる表の名前を使用します。

角括弧 (<>) の外側に入力されたテキストは、文字列として扱われます。

エディタは、閉じる前に文字列を検査します。文字列が有効でない場合、エラーメッセージが表示されます。

持続可能クラスのプロパティ

持続可能クラスおよび持続フィールドには、「データベースへのマップ」ウィザードを使用しないで指定できる独自のプロパティがあります。表 4-6 は、持続可能クラスに固有のプロパティについて説明します。

表 4-6　持続可能クラスのプロパティ

プロパティ	説明
キークラス	持続可能インスタンスを識別するためのキーフィールドを含んでいる関連クラスです。ただし、Meet-in-the-middle マッピングでは、キークラスを手作業で設定する必要があります。キークラスの設定の詳細については、90 ページの「キーフィールドとキークラス」 を参照してください。
マップされた主表	スキーマの中で、持続可能クラスに最も適合している表を、そのクラスの主表として選択します。持続可能クラスをマップするには、主表を指定する必要があります。この作業の手順については、63 ページの「既存のクラスをスキーマにマップ」 を参照してください。
マップされたスキーマ	持続可能クラスのマップ先となる表が入っているスキーマ。主表と二次表は、このスキーマに存在していなければなりません。52 ページの「スキーマを収集する」 で説明しているように、スキーマを収集するまで、この設定を行うことはできません。
マップされた二次表	二次表では、主表にはない列をクラスのフィールドにマップできます。たとえば、`Employee` クラスに部署名を含めるために、二次表として `DEPARTMENT` 表を追加することができます。複数の二次表を追加できますが、二次表をまったく使用しなくてもかまいません。このプロパティは、「マップされた主表プロパティ」を設定しているときだけ有効になります。二次表を追加する詳細については、69 ページ を参照してください。
持続可能	持続可能クラスかどうかを示します。このプロパティは、True に設定されているときだけ表示されます。クラスを持続可能に変換する手順については、64 ページの「クラスを持続可能クラスにする」 を参照してください。クラスを持続可能から元に戻す手順については、64 ページの「クラスを持続可能から元に戻す」 を参照してください。

持続可能クラスのプロパティは、図 4-27 のようになります。

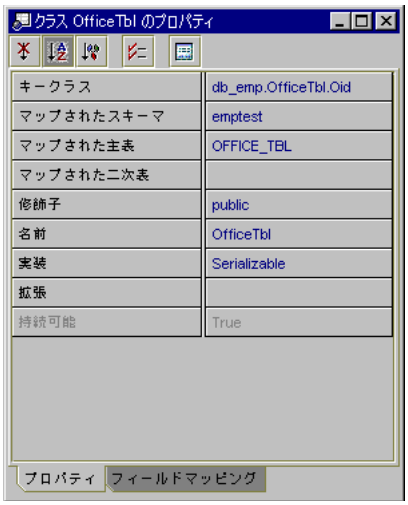

図 4-27 持続可能クラスのプロパティ

クラスをアンマップするには、「マップされた主表」プロパティのドロップダウンメニューから「<アンマップ>」を選択します。フィールドマッピングや二次表を設定している状態で、現在マップしているクラスをアンマップすると、「質問」ダイアログが表示されます。クラスをアンマップしてもかまわない場合は「了解」をクリックします。「取消し」をクリックすると、マッピング状態の変更が中止され、クラスがマップされたままになります。

「プロパティ」ウィンドウの下にある「フィールドマッピング」タブをクリックすると、持続可能クラスのフィールドマッピングプロパティが表示されます (図 4-28)。

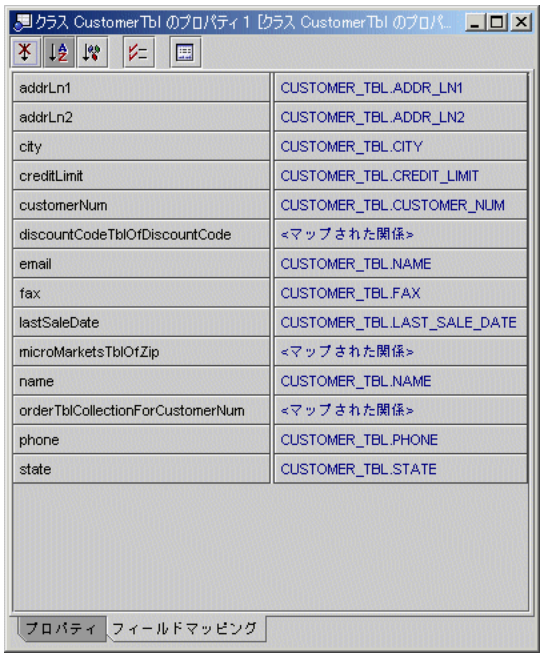

図 4-28　フィールドマッピングプロパティ

持続フィールドのプロパティ

フィールドのプロパティを表示するには、フィールドノードを右クリックし、「プロパティ」を選択します。持続フィールドのプロパティシートについては、図 4-29 を参照してください。

図 4-29 持続フィールドのプロパティ

持続フィールドをマップするには、「マッピング」プロパティを選択し、そのドロップダウンメニューから列を選択します。マップ先の列をさらに追加したい場合は、「...」ボタンをクリックし、マッピングのプロパティエディタを表示します。このプロパティエディタの詳細については、図 4-18 を参照してください。

関係フィールドをマップするには、「マッピング」プロパティを選択し、「...」ボタンをクリックし、「関係フィールドのマップ」ウィザードを表示します。このウィザードの詳細については、72 ページの「関係フィールドをマップする」 を参照してください。

フィールドをアンマップするには、ドロップダウンメニューから「<アンマップ>」または「<アンマップされた関係>」を選択します。

持続フィールドに固有のプロパティを 表 4-7 に示します。

表 4-7　持続フィールドのプロパティ

プロパティ	説明
削除アクション (関係フィールドのみ)	「階層」または「なし」に設定します。「階層」は、このフィールドを削除すると関連フィールドもすべて削除されることを意味します。「なし」は、このフィールドに表示されているオブジェクトだけが削除されることを意味します。
関連クラス (関係フィールドのみ)	関連クラスとは、関係フィールドがポイントするクラスのことです。関連クラスは、コレクションの要素を構成するオブジェクトの種類を識別します。フィールドがコレクションでない場合、このプロパティは使用できません。
関連フィールド (関係フィールドのみ)	関連フィールドは、関係クラス内の関係フィールドに設定できます。このプロパティを設定すると、管理されている関係に関係フィールドがロックされます。
フェッチグループ	「レベル」、「独立」、「デフォルト」、または「なし」を指定できます。階層型と独立型の 2 種類のフェッチグループがあります。あるフィールドを「デフォルト」に設定することは、「デフォルト」に設定されている他のすべてのフィールドとともに、そのフィールドが取り出されることを意味します。「レベル 1」グループのフィールドが取り出されると、「レベル 1」グループと「デフォルト」グループのすべてのフィールドが同様に取り出されます。 関連フィールドは、フェッチグループに入ることができません。 階層グループには「デフォルト」 設定と「レベル」設定があり、相互に階層を形成しています (たとえば、「レベル 2」も「レベル 1」を含みます)。 独立型のグループには、「デフォルト」グループと、指定された「独立」グループだけが含まれます (「独立 2 」は、「独立 1 」を含みません)。 「フェッチグループ」プロパティが無効になっている場合、フィールドは持続でないか、マップされないか、またはキーフィールドが「True」に設定されていて常に取り出される、のいずれかです。
キーフィールド (持続フィールドのみ)	True に設定した場合、フィールドは、持続可能クラスの主表の主キーの列にマップされます。

表 4-7　持続フィールドのプロパティ (続き)

プロパティ	説明
下限 (関係フィールドのみ)	関係フィールドに収容できる最小オブジェクト数。0 (デフォルト値) に設定すると、このフィールドをヌルにすることができます。関係の「多」の側では、この値に、「上限」を超えない任意の整数値を設定できます。関係の「1」の側では、この値に、1 または 0 を設定します。
マッピング	フィールドのマッピング状態を示します。
持続	True に設定した場合、このフィールドの値はデータベースに格納されます。
読み取り専用	True に設定した場合、このフィールドの値をデータベースで更新することはできません。
上限 (関係フィールドのみ)	関係フィールドに収容できる最大オブジェクト数。関係の「多」の側では、このプロパティは任意の整数値に設定できます。デフォルトは「*」(java.lang.integer.MAX_VALUE) です。関係の「1」の側での「上限」は 1 で、変更はできません。

エクスプローラのフィールドアイコンは、クラスまたはフィールドが持続可能であるかどうかで異なります。持続可能のクラスとフィールドは三角形でマークされます (図 4-30 および 図 4-31 を参照してください)。

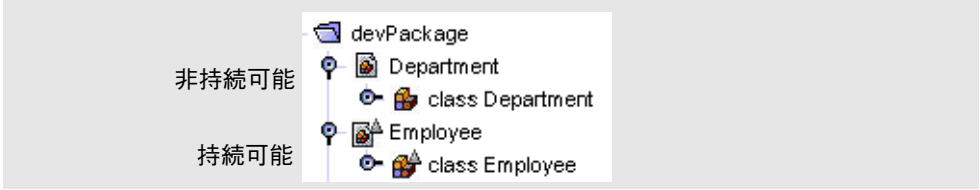

図 4-30　クラスのアイコン

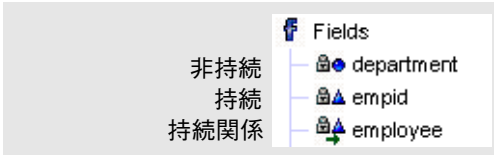

図 4-31　フィールドのアイコン

キーフィールドとキークラス

キークラスとは、個々の透過的な持続性インスタンスの一意の識別子情報を収容するために、持続可能クラスごとに用意されるクラスです。キークラスは、Java ジェネレータによって自動的に作成され、キーフィールドも自動的に設定されます。ただし、Meet-in-the-middle マッピングでは、これらのプロパティを明示的に設定し、キークラスを書き込む必要があります。

キークラスとキーフィールドを生成した後でフィールドを変更する場合、Oid クラスを更新する必要があります。テンプレートを使用して新しい持続可能クラスを作成する場合、更新可能なスケルトン Oid クラスが生成されます。

キークラスには次の 2 種類があります。

- `Oid` と命名された静的な内部クラス
- 接尾辞 `Key` が付いた単独のクラス

図 4-27 では、キークラスが `db_emp.OfficeTbl.Oid` に設定されています。これは、Java ジェネレータによって自動的に設定された `Oid` クラスです。

▼ キークラスとキーフィールドを設定する

1. クラスのノードのプロパティで、「キークラス」プロパティを設定します。キークラス名として、有効なクラス名を指定してください。
2. キークラスを作成し、キーフィールドをすべて取り込みます。

   持続可能クラスのフィールドのうち、主キーとして設定されているものを、すべてキークラスで宣言します。キークラスの個々のフィールドの名前と型は、持続可能クラスの対応するフィールドと同じにする必要があります。キークラスのすべてのフィールドは、公的に宣言する必要があります。キークラスは、`java.io.Serializable` を実装し、`equals` および `hashCode` メソッドを上書きする必要があります。
3. 持続可能クラスのフィールドのうち、主キーにマップされたすべてのフィールドについて、「キーフィールド」プロパティを True に設定します。持続可能クラスに含まれていないフィールドについては、このプロパティをすべて False に設定します。

Employee クラスに対し定義されている内部 Oid クラスの例を次に示します。

```
public static class Oid {

public long empid;
    public Oid() {
    }

    public boolean equals(java.lang.Object obj) {
        if( obj==null ||
        !this.getClass().equals(obj.getClass())) return(false);
        Oid o=(Oid) obj;
        if( this.empid!=o.empid ) return( false);
        return(true);
    }

    public int hashCode() {
        int hashCode=0;
        hashCode += empid;
        return(hashCode);
    }
```

Employees クラスに対して定義されるキークラスの例を次に示します。

```
public static class EmployeeKey implements java.io.Serializable{

public long empid;
    public EmployeeKey() {
    }

    public boolean equals(java.lang.Object obj) {
        if( obj==null ||
        !this.getClass().equals(obj.getClass())) return(false);
        EmployeeKey = (EmployeeKey) obj;
        if( this.empid!=o.empid ) return(false);
        return(true);
    }

    public int hashCode() {
        int hashCode=0;
        hashCode += empid;
```

```
        return(hashCode);
    }
}
```

アプリケーションの実行

Forte for Java でアプリケーションをコンパイルした後は、パッケージを .jar ファイルに追加したり、アプリケーションを Forte for Java 上で実行することができます。

JAR ファイルの作成

IDE の JAR パッケージャを使用すると、ファイル階層から単一の JAR (Java ARchive) ファイルを作成できます。このファイルは、IDE 外部のアプリケーションで使用できます。透過的な持続性を使用できるアプリケーションの場合、持続可能クラスと、持続可能クラスの持続フィールドにアクセスするクラス (持続性認識クラス) は、IDE の JAR パッケージツール (JAR、WAR、EAR パッケージャなど) でアーカイブし、クラスファイルのバイトコードを拡張する必要があります。

持続可能クラスの JAR ファイルを作成する場合、次の事柄も考慮する必要があります。

- Javac コンパイラからの予期しないコンパイルエラーが発生する恐れがあるため、Java ファイルを JAR ファイルに追加しないでください。そのためには、jarContent ノードの「ファイルフィルタ」プロパティを「<*.java および *.jar 以外のすべてのファイル>」に設定します。このプロパティを「<クラスのみ>」に設定しないように注意してください。「<クラスのみ>」に設定すると、マッピングファイル (*.mapping) が除外されてしまいます。マッピングファイルは、持続可能クラスを識別するために透過的な持続性実行時環境で必要とされます。
- また、スキーマファイル (*.dbschema) がインクルードされていることを確認してください。スキーマファイルは、指定のパッケージに含まれていれば、自動的に取り込まれます。それ以外の場合は、 CLASSPATH でスキーマファイルの位置を指定する必要があります。
- 持続性認識アプリケーションを IDE の外側で使用する場合、次の JAR ファイルが CLASSPATH に含まれていることを確認してください。
  - .../modules/ext/persistence-rt.jar
  - .../modules/dbschema.jar
  - .../lib/ext/xerces.jar
  - <パッケージ>.jar (持続クラスのパッケージを含む JAR ファイル)
  - JDBC ドライバ

▼ JAR ファイルを作成する

1. 「テンプレートウィザードから新規作成」を使用し、「JAR パッケージャ」テンプレートを開きます。
2. JAR ファイルのコンテンツを指定します。
3. JAR ファイルをコンパイルします。

注 - JAR パッケージャを使用して、Forte for Java の外部で使用する `.jar` ファイルを作成する場合は、デフォルトのファイルフィルタ「`<*.java` および `*.jar` 以外のすべてのファイル`>`」を使用しないと、コンパイルエラーが発生することがあります。

JAR ファイルの作成の詳細については、IDE ヘルプのトピック 「JAR パッケージャの使用方法」を参照してください。

Forte for Java でのアプリケーションの実行

使用しているアプリケーションの `main()` メソッドを含むクラスを選択し、そのプロパティシートを開き、「実行」タブの「実行方法」プロパティの値として「Persistence 実行」を選択します。

この操作により、クラスのロード時に透過的な持続性の拡張機能 (Enhancer) が呼び出され、生成されたクラスに `com.sun.forte4j.persistence.PersistenceCapable` インタフェースが実装されます。この結果、持続可能クラスは実行時環境と交信できるようになります。

`com.sun.forte4j.persistence.PersistenceCapable` インタフェースには、一連のメソッドが宣言されています。持続可能クラスを使用するアプリケーション開発者は、これらのメソッドを使用して、持続可能クラスのインスタンスの状態を参照および再設定することができます。

クラスの開発者と、クラスを使用するアプリケーション開発者のどちらも、生成されたバイトコードの内容を知る必要はありません。クラスの開発者は、持続データの正確なモデル化に専念することができます。

「Persistence 実行」を使用して Forte for Java IDE 外部で持続可能クラスの実行などを行わない場合、IDE の JAR パッケージャを使用して、持続可能クラスおよび持続性認識クラスをアーカイブする必要があります。それらのファイルがアーカイブされることにより、各クラスに拡張機能が適用されます。この作業では、透過的な持続性モジュールが Forte for Java で使用可能になっている必要があります。

注 - JSP/サーブレットアプリケーションの Web モジュールで持続可能クラスを使用する場合、持続可能クラスを JAR ファイルとしてパッケージし、JAR ファイルを Web モジュールの `WEB-INF/lib` ディレクトリに配置します。WAR ファイルを作成して Web アプリケーションを配備する必要のない限り、持続可能クラスを Web モジュールの `WEB-INF/classes` (または `WEB-INF/`クラス) ディレクトリに直接配置しないでください。IDE のアーカイブツール (JAR または WAR パッケージャなど) または「Persistence 実行」を使用する場合に限り、透過的な持続性クラスファイルが拡張されます。

対応しているデータ型

透過的な持続性は、Java データフィールドから SQL 型へのマッピングに使用される、JDBC 1.0 の SQL データ型をサポートします。表 4-8 に、サポートしているデータ型を示します。

表 4-8　サポートしているデータ型

JDBC SQL データ型
`BIGINT`
`BIT`
`CHAR`
`DATE`
`DECIMAL`
`DOUBLE`
`FLOAT`
`INTEGER`
`LONGVARCHAR`

表 4-8　サポートしているデータ型 (続き)

JDBC SQL データ型
NUMERIC
REAL
SMALLINT

サポートしているデータ型のヌル可能性を 表 4-9 に示します。

表 4-9　マッピングでのデータ型変換

Java データ型	JDBC データ型	ヌル可能性
boolean	BIT	ヌル可能でない
java.lang.Boolean	BIT	ヌル可能
byte	TINYINT	ヌル可能でない
java.lang.Byte	TINYINT	ヌル可能
double	FLOAT	ヌル可能でない
java.lang.Double	FLOAT	ヌル可能
double	DOUBLE	ヌル可能でない
java.lang.Double	DOUBLE	ヌル可能
float	REAL	ヌル可能でない
java.lang.Float	REAL	ヌル可能
int	INTEGER	ヌル可能でない
java.lang.Integer	INTEGER	ヌル可能
long	BIGINT	ヌル可能でない
java.lang.Long	BIGINT	ヌル可能
long	DECIMAL (scale==0)	ヌル可能でない
java.lang.Long	DECIMAL (scale==0)	ヌル可能
long	NUMERIC (scale==0)	ヌル可能でない
java.lang.Long	NUMERIC (scale==0)	ヌル可能
short	SMALLINT	ヌル可能でない

表 4-9　マッピングでのデータ型変換 (続き)

Java データ型	JDBC データ型	ヌル可能性
java.lang.Short	SMALLINT	ヌル可能
java.math.BigDecimal	DECIMAL (scale!=0)	ヌル可能でない
java.math.BigDecimal	DECIMAL (scale!=0)	ヌル可能
java.math.BigDecimal	NUMERIC	ヌル可能
java.math.BigDecimal	NUMERIC	ヌル可能でない
java.lang.String	CHAR	ヌル可能でない
java.lang.String	CHAR	ヌル可能
java.lang.String	VARCHAR	ヌル可能でない

注 - 透過的な持続性は、BLOB 型のマップ列に対応していません。BLOB 型を参照または更新するには、専用の JDBC トランザクションを使用する必要があります。

第5章

持続性認識アプリケーションの開発

この章では、透過的な持続性実行環境と、この環境での持続性オペレーションの実行方法を説明しています。透過的な持続性プログラミングについての各種の問題についても取り扱っています。

透過的な持続性 API は、データベースの操作を制御します。アプリケーションは、この API を使用してデータベースとの接続を確立し、トランザクションを作成します。挿入と削除は、すべてトランザクションのコンテキスト内で行う必要があります。

概要

透過的な持続性実行環境は、持続可能クラスのインスタンスと持続性マネージャ のメソッドを、アプリケーションで使用するデータベースで解釈可能な命令に翻訳し、Java 開発者に統一的なインタフェースを提供します。

この実行環境は、いくつかの Java インタフェースを介して参照することができます。これらのインタフェースは、メソッド呼び出しをデータベース固有の命令に翻訳する機能を備えた、持続データ用のメソッドの集まりです。

スキーマにマップした持続可能 (Persistence-Capable、PC) クラスがある場合には、開発者は PC クラスのメソッドおよび持続性認識実行時サポートクラスを呼び出して持続データにアクセスします。開発者は、Forte for Java の通常の編集、コンパイル、テストラン、および実装環境を使用して、PC クラスを使用するコードを記述します。

実行時クラスの透過的な持続性の実装は、`com.sun.forte4j.persistence` インタフェースによって決定されます。透過的な持続性には、これらのインタフェースが実装されている`persistence-rt.jar` というファイルが含まれます。

アプリケーションは、Java メソッドを呼び出してデータベースを操作します。一般的な手順は以下のとおりです。データベースへの接続、トランザクションの開始、持続データの選択・挿入・更新・削除、トランザクションのコミット (またはロールバック) という標準的な手順に従うだけでよく、照会言語を使用したり、データベース固有の Java コードを作成したりする必要はありません。

アプリケーションがデータストアからデータをロードする場合、データをモデル化する PC クラスのインスタンスを使用します。アプリケーションが持続フィールドの値を変更すると、透過的な持続性の実行環境がその変更を追跡し、アプリケーションがそのトランザクションをコミットする時に新しい値をデータストアに保存します。データを獲得する必要がある時には、開発者は持続性マネージャ (このマネージャは持続可能クラスのインスタンスを返します) のメソッドを呼び出します。また、データを変更する必要がある時には、持続可能インスタンスのメソッドを呼び出します。

ここからは、アプリケーションで持続可能クラスのインスタンスを作成・使用する方法について説明します。

持続性認識クラスの開発

アプリケーションは Java プログラミング言語で作成します。既存のクラスを使用することも、必要に応じて新しいクラスを作成することもできます。透過的な持続性のオブジェクトやクラスは、Java のその他のオブジェクトやクラスと同じように使用することができます。唯一の違いは、透過的な持続性の持続オブジェクトの値が、データベースに保存されることだけです。そのため、開発者はデータソース (データベース、ローカル変数など) の違いを意識しないですみます。

持続性認識ロジック

実際のアプリケーションでの透過的な持続性の代表的なアーキテクチャを 図 5-1 に示します。このアプリケーションは、標準の J2EE アーキテクチャに基づいていて、JSP やサーブレットがリモートサイトにいるエンドユーザーと情報をやり取りします。JSP やサーブレットは、エンドユーザーからの入力を受け付け、それらの入力に対してどのような処理が必要かを特定し、中間層のサービスにその処理を実行させます。たとえば、エンドユーザーが従業員のレコードを参照したい場合は、JSP やサーブレットは中間層のサービスに要求を送出し、該当する従業員のレコードを返させます。JSP

やサーブレットは、このレコードの取り出し方法を知る必要はありません。すなわち、JSP やサーブレットは、持続性認識ロジックの組み込み先として適切ではありません。

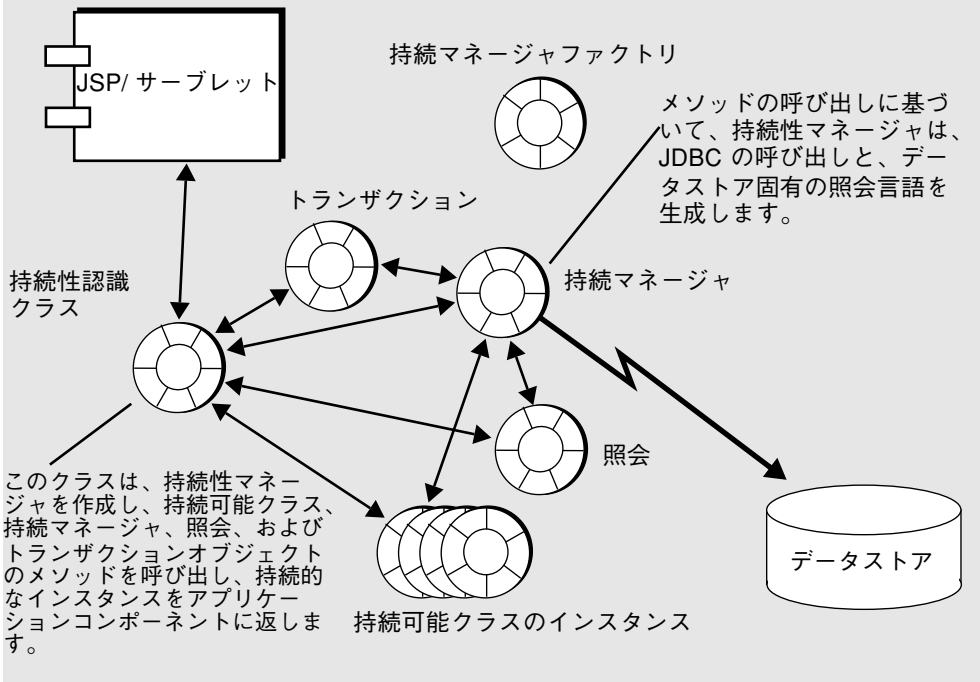

図 5-1　持続性認識ロジックの単独クラスへの移動

この処理を実現するため、持続性認識ロジックを単独のクラスに移します。JSP やサーブレットは、次のようにして持続性認識クラスのメソッドを呼び出し、従業員のレコードを要求します。

```
Employee requestedEmployee =
PersistentAwareInstance.getEmployeeData("485843");
```

持続性認識インスタンスは、要求された従業員レコードの `Employee` インスタンスを取得し、JSP やサーブレットに返します。この持続インスタンスは、JSP やサーブレットに返された後も、持続性マネージャとそのトランザクションに関連付けられたままになります。これは、JSP やサーブレットがフィールドの値を更新したときに、持続性マネージャがデータストアの更新処理を自動的に生成できるようにするためです。持続性マネージャが生成した更新処理は、その時点でのトランザクションと並行性の制御方式に従って管理されます。

エンドユーザーが新しい従業員のレコードを入力した場合は、JSP や サーブレットは新しいインスタンスを作成し、持続性認識クラスに渡します。

```
Employee newEmployee = new Employee(<data>);
PersistentAwareInstance.addEmployeeData(newEmployee);
```

持続性認識クラスは、これを次のように処理します。

```
PersistenceManager.makePersistent(newEmployee);
```

図 5-1 のアーキテクチャでは、JSP やサーブレットは複数のエンドユーザーを同時に処理します。エンドユーザーごとにセッションを作成し、それぞれのセッションで、エンドユーザーの Web ブラウザと JSP (またはサーブレット) との間で受け渡される一連の HTML ページを処理することができます。JSP やサーブレットは、データストアサービスの持続性認識インスタンスに要求を送出しますが、要求を受け取った持続性認識インスタンスは、JSP やサーブレットのどのセッションから要求が送出されたかを特定し、あるセッションからの要求を他のセッションからの要求と区別する必要があります。

持続性マネージャは、通常、複数のデータストア操作で作成またはフェッチされた TP インスタンスのセットを管理するため、会話的なセッションで生成された持続的インスタンスを管理できます。

開発手順

アプリケーション開発者は、透過的な持続性クラスと実行環境オブジェクトのメソッドを使用してデータを操作します。ここでは、メソッド呼び出しの基本的な手順を説明します。

1. **持続マネージャファクトリを作成または入手します。**

   持続マネージャファクトリは設定可能なコンポーネントで、データベース接続情報を記録したプロパティを備えています。すでに使用している環境で設定され、JNDI 検索機能でアクセス可能な持続マネージャファクトリが用意されている場合もあります。詳細については、105 ページの「持続マネージャファクトリを作成する」 を参照してください。

2. **必要に応じて接続ファクトリを作成します。**

この作業が必要になるのは、接続プールを実装する場合だけです。このアプローチの詳細については、110 ページの「プール接続」 を参照してください。

3. **持続性マネージャを作成します。**

   通常は、セッションごとに持続性マネージャを作成します。持続性マネージャは、持続マネージャファクトリのプロパティで定義された接続を使用します。ただし、この設定をアプリケーションでオーバーライドすることもできます。詳細については、105 ページの「持続マネージャファクトリを作成する」 を参照してください。

4. **`currentTransaction()` を呼び出して、持続性マネージャからトランザクションオブジェクトを取得します。**

   通常は、アプリケーションからトランザクションを開始します。トランザクションオブジェクトを持続性マネージャから取得し、持続性マネージャが管理するインスタンスに適用します。詳細については、117 ページの「トランザクション」 を参照してください。

5. **照会インタフェースを使用して、データベースから持続可能クラスのインスタンスを取り出します。**

   これらのインスタンスのメソッドを呼び出して、インスタンスの内容を更新します。インスタンスを挿入・削除するには、持続性マネージャインタフェースのメソッドを使用します。

   アプリケーションは、データベースの照会、レコードの更新、レコードの追加といった処理を行うときに、必要なデータを表現した持続インスタンスを作成します。これらのインスタンスのためのデータベース操作は、持続性マネージャがすべて管理します。したがって、個々の持続性マネージャが管理する持続インスタンスが、それぞれのセッションでのデータ表現になります。

6. **トランザクションをコミットするか、またはロールバックします。**

   トランザクションをコミットすると、データベースに更新内容が保存されます。トランザクションをロールバックすると、データベースがトランザクション開始前の状態のままになります。

   アプリケーションがトランザクションをコミットすると、それぞれの持続インスタンスのその時点の状態に応じたデータベース操作が行われます。たとえば、トランザクションの実行中に、持続インスタンスがある場合は挿入処理が生成され、削除されたインスタンスがある場合は削除処理が生成され、更新されたインスタンスがある場合は更新処理が生成されます。

7. 他のトランザクションを実行します。

   同じ持続性マネージャインスタンスを他のトランザクションで再使用することができます。新しい持続性マネージャインスタンスを使用することもできます。

8. 持続性マネージャを閉じて、アプリケーションを終了します

この開発手順を、図 5-2 にフローチャートで示します。

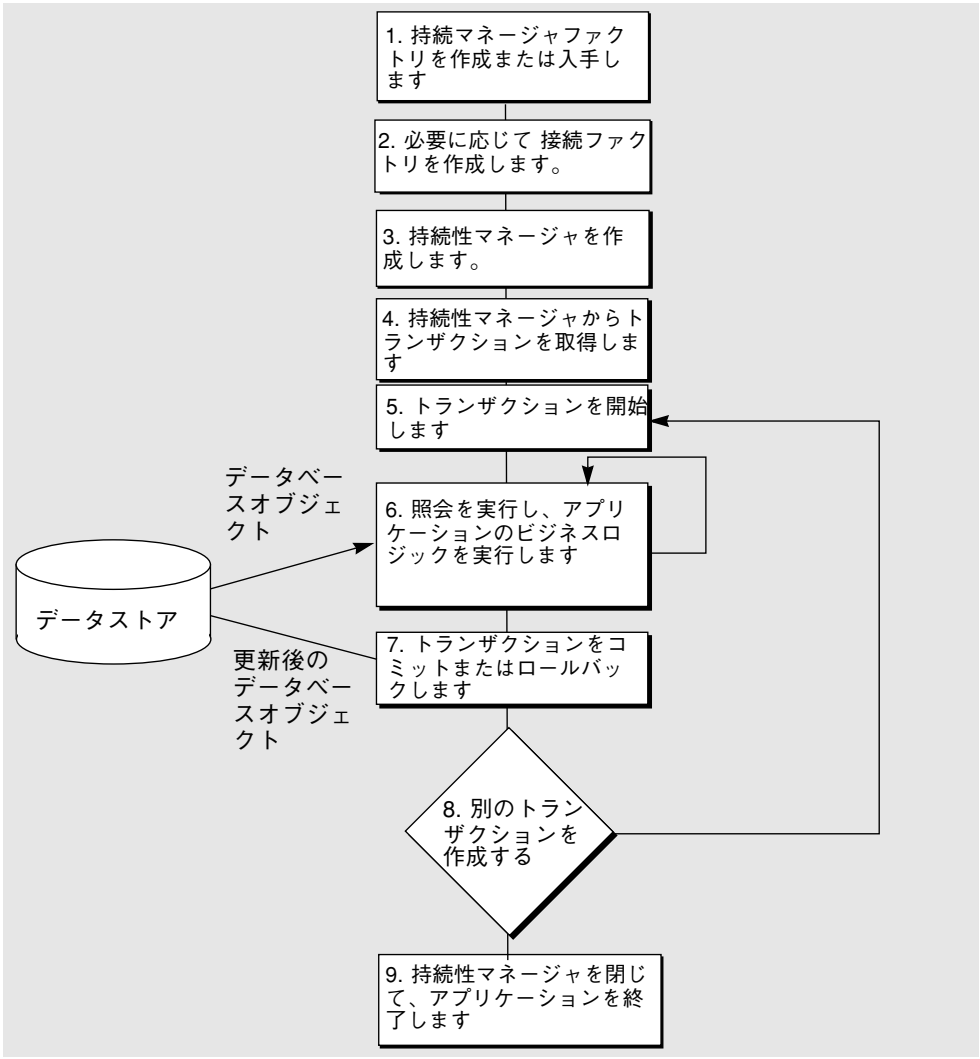

図 5-2　透過的な持続性アプリケーションのロジック

持続マネージャファクトリを作成する

持続マネージャファクトリは、持続性認識アプリケーションの基盤になります。持続マネージャファクトリはクラスとして実装され、開発者が直接インスタンス化することができます。それ以外のオブジェクトを取得する場合は、持続マネージャファクトリや持続性マネージャのメソッドを呼び出します。通常は、すでに使用している環境で設定され、JNDI のメソッドでアクセス可能な持続マネージャファクトリが用意されています。この場合、開発者は、111 ページの「持続性マネージャを作成する」 までスキップできます。

一般に、アプリケーションは持続マネージャファクトリを介して接続を取得します。持続マネージャファクトリには、データベースへの接続に必要な情報を記録するプロパティがあります。アプリケーションは、持続マネージャファクトリをインスタンス化し、そのプロパティを設定し、その後で持続性マネージャを作成します。このようにすることで、持続マネージャファクトリで設定した接続情報を持続性マネージャで使用できるようになります。

持続性認識クラスを作成するには、「ファイル」>「新規」>「Classes」>「Classes」を選択します。クラスに名前を付けて「完了」をクリックしてください。

それぞれのメソッドの詳細を 表 5-1 に示します。

表 5-1　PersistenceManagerFactory のメソッド

メソッド	説明
`setOptimistic` `getOptimistic`	トランザクションの並行性制御方式を示すフラグを操作します。デフォルト値は true です。
`setRetainValues` `getRetainValues`	トランザクションをコミットした後の持続インスタンスの取り扱い方法を示すフラグを操作します。デフォルト値は true です。
`setIgnoreCache` `getIgnoreCache`	照会でフィルタ式を評価するときに、キャッシュされているインスタンスを考慮に入れるかどうかを示すフラグを操作します。このフラグの値は常に true です。false に変更すると、`JDOUnsupportedOptionException` がスローされます。
`setNontransactionalRead` `getNontransactionalRead`	トランザクションの外部から非トランザクションインスタンスの読み取りを可能にするかどうかを示すフラグを操作します。デフォルト値は true です。
`setConnectionFactory` `getConnectionFactory`	データストア接続の取得先として使用する接続ファクトリを指定します。
`setConnectionMinPool` `getConnectionMinPool`	接続プール内の最小接続数を指定します。
`setConnectionMaxPool` `getConnectionMaxPool`	接続プール内の最大接続数を指定します。
`setConnectionFactoryName` `getConnectionFactoryName`	データストア接続の取得先として使用する接続ファクトリの名前を指定します。この名前が JNDI で検索されることにより、接続ファクトリが見つけ出されます。
`setConnectionTransactionIsolation` `getConnectionTransactionIsolation`	トランザクション遮断レベルをデフォルト設定値から変更します。 引数 `level` は、接続先のデータベースが対応している `java.sql.Connection.TRANSACTION_*` オプションであれば何でもかまいません。

表 5-1　PersistenceManagerFactory のメソッド (続き)

メソッド	説明
`getPersistenceManager`	持続性マネージャのインスタンスを返します。このインスタンスのプロパティは、持続マネージャファクトリで指定されている値に設定されます。 いったん getPersistenceManager を使用すると、それ以降はどの set メソッドも使用できなくなります。
`getProperties`	透過的な持続性では、動作とは関係のないプロパティを記録し、プロパティインスタンスを介してアプリケーションから参照可能にすることができます。このメソッドは、これらのプロパティインスタンスを取得します。それぞれのキーと値は文字列型です。必要なキーは次のとおりです。 VendorName: ベンダー名 VersionNumber: バージョン番号を示す文字列 持続マネージャファクトリのプロパティ設定値は、その持続マネージャファクトリから作成される持続性マネージャのデフォルト設定値になります。いったん持続性マネージャを作成すると、それ以降は持続マネージャファクトリのプロパティを変更できません。

表 5-1　PersistenceManagerFactory のメソッド (続き)

メソッド	説明
`QueryTimeout` `UpdateTimeout`	データベースのデッドロックを防止するため、トランザクションインスタンスの照会や更新を開始してから、タイムアウトになるまでの時間を指定します。 タイムアウト時間は秒数で指定します。ゼロは無制限を意味します。ここで指定した値は、該当するデータベースに対するすべてのトランザクションのデフォルト値になります。 いったん持続性マネージャを作成すると、それ以降は 持続マネージャファクトリのプロパティを変更できません。トランザクションのタイムアウト時間は、必要に応じて変更することができます。 PointBaseは、現時点では `PreparedStatement.setQueryTimeout()` に対応していません。タイムアウト時間をデフォルト値 (現行リリースでは 60 秒) から変更したい場合は、URL または `pointbase.ini` ファイルに、`locks.timeout=value` という文字列を追加します。ただし、`locks.timeout=0` を指定すると、タイムアウト時間が 0 秒に設定されることに注意してください。`setQueryTimeout(0)` の場合は、タイムアウト時間が無制限になります)。 `Locks.timout` は、クライアント側ではなくサーバー側で設定します。このことは、すべての接続に対し指定の値がホールドすることを意味します。

データベースに接続する

接続は、透過的な持続性実行環境によって開かれ、管理されます。持続マネージャファクトリは設定可能なコンポーネントで、データベースの URL、そのデータベースの有効なユーザー名とパスワードといった、データベース接続に必要な情報を記録するプロパティを備えています。持続性マネージャは、これらの持続マネージャファクトリで設定された接続情報を使用します。アプリケーションが、照会の実行要求の送出といった、接続を必要とする操作をはじめて実行すると、持続性マネージャによって接続が開かれます。

接続の管理方式には次の 4 種類があります。

- 単純接続
- プール接続

- 分散トランザクション
- 管理接続

管理接続以外の方式 (単純接続とプール接続) では、トランザクションが内部的に管理する接続がトランザクションの終了を処理します。管理接続では、接続に関連付けられた XAResource がトランザクションを終了させます。どちらの場合も、持続性マネージャでは接続基盤とのインタフェースを適切に設定する必要があります。

接続ファクトリ

標準の接続機能を使用する場合は、通常は接続ファクトリの該当するプロパティをすべてサポートする必要があります。接続ファクトリは、直接設定することも、持続マネージャファクトリを介して設定することもできます。

それぞれのメソッドの詳細を 表 5-2 に示します。

表 5-2　ConnectionFactory のメソッド

メソッド	説明
URL	データソースの URL です。
UserName	接続に使用するユーザー名です。
Password	ユーザーのパスワードです。
DriverName	接続ドライバ名です。
ServerName	データソースのサーバー名です。
PortNumber	データソースとの接続に使用するポート番号です。
MaxPool	接続プール内の最大接続数を指定します。
MinPool	接続プール内の最小接続数を指定します。
MsWait	接続を取得するまでの待機時間 (ミリ秒単位) です。この時間が過ぎても接続プールから接続が与えられない場合は、例外がスローされます。
LogWriter	メッセージの送出先の PrintWriter です。
LoginTimeout	データソースとの新しい接続を確立するまでの待機時間 (秒数) です。
TransactionIsolation	接続のトランザクション遮断レベルです。この設定はすべての接続に適用されます。

単純接続

最も簡単なのは、持続性マネージャからデータベースに直接接続し、トランザクションデータを管理する方法です。その場合は、ユーザーやデータストアを識別するための情報を除いて、接続のプロパティを外部から参照する必要はありません。トランザクション処理の間、持続性マネージャからのデータ読み取り / 書き込み要求や、トランザクションの終了要求は接続によって処理されます。

アプリケーションで接続プールを使用する必要がない場合は、`PersistenceManagerFactory` の次のプロパティしか設定する必要はありません。

- `ConnectionUserName` – 接続に使用するユーザー名です。
- `ConnectionPassword` – ユーザーのパスワードです。
- `ConnectionURL` – データソースの URL です。
- `ConnectionDriverName` – 接続のドライバ名です。

持続マネージャファクトリで設定したこれらのプロパティ値は、その持続マネージャファクトリから作成したすべての持続性マネージャインスタンスのデフォルト値になります。

たとえば、持続マネージャファクトリを初期化するコンストラクタは、次のようになります。

```
public DataSource() {
PersistenceManagerFactory pmf = new
PersistenceManagerFactoryImpl();
    pmf.setConnectionUserName("scott");
    pmf.setConnectionPassword("tiger");

pmf.setConnectionDriverName("oracle.jdbc.driver.OracleDriver");
    pmf.setConnectionURL("jdbc:oracle:thin:@DIESEL:1521:ORCL");
    setOptimistic(false); // デフォルト値は true
}
```

プール接続

この方式は、単純接続より少し複雑です。この方式では、持続マネージャファクトリから複数の持続性マネージャインスタンスを作成し、資源の消費を抑えるため、それらのインスタンスから接続プールを使用します。それぞれの持続性マネージャは、単一のデータベースのトランザクションを処理します。この場合、プールしている接続

ファクトリは、持続性マネージャインスタンスで使用される別個のコンポーネントです。持続マネージャファクトリは、接続プールコンポーネントへの参照を、JNDI 名またはオブジェクト参照として含んでいます。接続プールコンポーネントは単独で設定されるため、持続マネージャファクトリでは、このコンポーネントを使用するように設定するだけですみます。

その他の接続プロパティが必要な場合は、持続マネージャファクトリで `setConnectionMinPool` と `setConnectionMaxPool` を設定する必要があります。

セッションのビジネスメソッドで、長時間に及ぶオプティミスティックトランザクションを実行する場合は、その途中で接続を要求し、データベースからデータを取り出したい場合があります。その場合は、持続性マネージャから要求を創出し、接続プールから接続を取得します。処理が終了すると、この接続は接続プールに返されます。

データベーストランザクションでは、**トランザクション**が取得した接続がセッションが終了するまで保持されます。セッションの終了 (コミットまたはロールバック) 後は、この接続が接続プールに返され、後続のトランザクションで再使用されます。

持続性マネージャを作成する

持続性マネージャは、アプリケーションと透過的な持続性実行時環境との対話の開始地点となります。特定のデータストアに関する情報をカプセル化し、接続を開始して、照会とトランザクションの管理を行います。持続性マネージャを宣言する前に、持続マネージャファクトリを設定しておく必要があります。

持続性認識クラスで、持続性マネージャを宣言して持続性マネージャインスタンスを作成します。

```
private PersistenceManager pm;this.pm =
pmf.getPersistenceManager();
```

それぞれの持続性マネージャは、一度に 1 つのトランザクションしかサポートしませんが、このトランザクションは、トランザクション自身が作成する持続可能 (persistent-capable) クラスのすべてのトランザクションインスタンスに適用します。トランザクションで使用するため、アプリケーションは持続性マネージャからトランザクションオブジェクトを獲得します。

```
Transaction myTx = myPersistenceManager.currentTransaction();
```

ほとんどの場合、アプリケーションは単一のデータベースから複数のローカルトランザクションを実行します。これらのトランザクションは、トランザクションオブジェクトメソッドの呼び出しによって開始され終了します。

```
myTx.begin();myTx.commit(); // or myTx.rollback();
```

持続性マネージャは通常、データベースとのすべての対話 (持続データのキャッシュコピーの再表示なども含む) を管理します。アプリケーションでは、トランザクション範囲の識別だけが必要になります。

それぞれのメソッドの詳細を 表 5-3 に示します。

表 5-3　`PersistenceManager` のメソッド

メソッド	説明
`isClosed`	持続性マネージャインスタンスが作成されている場合は false を返します。 `close` メソッドが正常終了した後は true を返します。
`close`	トランザクションがアクティブになっていないことを確認します。トランザクションがアクティブの場合、例外をスローします。 資源 (**トランザクション**など) をすべて解放します。 `close` メソッドの終了後、`isClosed()` 以外のすべての持続性マネージャのメソッドが例外をスローします。
`currentTransaction`	持続性マネージャに関連付けられているトランザクションインスタンスを返します このメソッドからアクティブではないトランザクションインスタンスが返された場合は、そのインスタンスを使用して、トランザクションを終了できません。ただし、フラグの設定には使用できます。

表 5-3　PersistenceManager のメソッド (続き)

メソッド	説明
newQuery	持続性マネージャインスタンスは、照会インスタンスのファクトリ (作成元) になり、照会は持続性マネージャのコンテキスト内で実行されます。実際の照会は、持続性マネージャ自身が実行する場合もあれば、持続性マネージャがデータストアに命じて実行させる場合もあります。
getExtent	指定されたクラスのすべてのインスタンスを、読み取り専用の コレクションの形式で返します。subclasses フラグが true の場合は、該当するクラスのサブクラスのインスタンスもすべて返します。このメソッドから返されたコレクションは、主に照会インスタンスのパラメータとして使用します。そのため、このコレクションは、要素の重複がないかぎり、JVM ではインスタンス化されません。通常、このコレクションを使用する必要があるのは、データベースインスタンスの候補を特定したい場合だけです。 PersistenceManager.getExtent メソッドを、引数 subclasses=true を指定して呼び出すことはできません。 PersistenceManager.getExtent メソッドから返されたコレクションは、照会の中だけで使用することができます。また、このコレクションに対応しているメソッドは、iterator メソッドしかありません。それ以外のコレクションメソッド (追加、サイズ設定など) を適用すると、 UnsupportedOperationException 例外や JDOUnsupportedOptionException 例外がスローされます。
getObjectById	キャッシュ内の持続インスタンスのうち、オブジェクト ID が引数で指定された値と一致しているものを返します。キャッシュ内にアクティブなインスタンスがない場合は、空のインスタンスを作成し、その主キーフィールドに引数で指定された ObjectID を格納し、戻り値として返します。 インスタンスがデータベースに存在しない場合、このメソッドは異常終了します。ただし、インスタンスのフィールドへの後続アクセスは、例外をスローします。また、そのインスタンスで関係を設定すると、トランザクションが異常終了します。

表 5-3　PersistenceManager のメソッド (続き)

メソッド	説明
getObjectId	指定されたインスタンスのオブジェクト ID を返します。この ID は、その ID の作成元の持続性マネージャのコンテキスト内だけで一意性が保証されています。また、一意性が保証されるのは、最初の 2 種類の ID、すなわちアプリケーションで管理する ID と、データベースで管理される ID (現行リリースでは未対応) だけです。 持続性マネージャインスタンスの内部では、このメソッドから返された ObjectID は、その種類にかかわらず、該当する持続性マネージャに関連付けられたすべてのインスタンスの間で一意になります。 このメソッドから返された ObjectID をアプリケーションから変更しても、その ObjectID を持つインスタンスは影響を受けません。すなわち、返された ObjectID はローカルインスタンスのコピー (クローン) です。
getTransactionalInstance	持続性マネージャインスタンスの有効な持続インスタンスを返します。現在のインスタンスが特定の持続性マネージャに関連付けられている場合に、それとは別の持続性マネージャのインスタンスを取得したいときに使用します。 aPersistenceManager.getTransactionalInstance(pc) は、次のコードと同じです。 aPersistenceManager.getObjectById(pc.getStateManager() .getPersistenceManager() .getObjectId(pc))

表 5-3　PersistenceManager のメソッド (続き)

メソッド	説明
makePersistent	持続インスタンスをデータベースに挿入します。このメソッドは、アクティブなトランザクションのコンテキスト内で呼び出す必要があります。このメソッドは、該当するインスタンスにオブジェクト ID を割り当て、その状態を persistent-new に切り換えます。(コミットまたはペシミスティックトランザクションでのユーザー照会により) このインスタンスがフラッシュされる場合は、このインスタンスから (インスタンス内の持続フィールドを介して参照されている) すべての一時インスタンスが、持続インスタンスであるかのように取り扱われます。 持続性マネージャに、同じ ObjectId を持つ別のオブジェクトがすでに関連付けられている場合は、このメソッドは JDOUserException をスローします。 持続性マネージャによって管理されている持続インスタンスは、このメソッドの影響を受けません。また、該当するインスタンスが他の持続性マネージャによって管理されている場合は、このメソッドは JDOUserException をスローします。
deletePersistent	データベースから持続インスタンスを削除します。このメソッドは、アクティブなトランザクションのコンテキスト内で呼び出す必要があります。(コミットまたはペシミスティックトランザクションでのユーザー照会により) このメソッドを適用したインスタンスがデータベースにコミットされると、データベース上の該当するデータ表現が削除されます。 このメソッドの機能は、正確には makePersistent の逆ではありません (makePersistent には状態を変更する機能があるためです)。システム固有のポリシーオプション (カスケード削除など) によっては、従属データベースオブジェクトが削除される場合があります。 該当するインスタンスが他の持続性マネージャによって管理されている場合や、該当するインスタンスが一時インスタンスの場合は、このメソッドは例外をスローします。 トランザクションの中ですでに削除されているインスタンスは、このメソッドの影響を受けません。
getPersistenceManagerFactory	持続性マネージャの作成元の 持続マネージャファクトリを返します。

表 5-3　PersistenceManager のメソッド (続き)

メソッド	説明
setUserObject / getUserObject	持続インスタンスを管理するために、データ保持用のユーザーオブジェクトを使用したい場合があります。この 2 つのメソッドを使用して、これらのオブジェクトを管理することができます。これらのオブジェクトのパラメータは、システムからは参照・使用されません。
getProperties	透過的な持続性では、動作とは関係のないプロパティを記録し、プロパティインスタンスを介してアプリケーションから参照可能にすることができます。このメソッドは、これらのプロパティインスタンスを取得します。それぞれのキーと値は文字列型です。必要なキーは次のとおりです。 • VendorName: ベンダー名 • VersionNumber: バージョン番号を示す文字列
getObjectIdClass	アプリケーションで ObjectID クラスのインスタンスを作成したい場合があります。このメソッドは、持続可能クラスを引数として受け取り、ObjectID クラスを返します。
newSCOInstance	第 2 クラスオブジェクトの新しいインスタンスを返します。引数としてこのインスタンスの型、およびこのインスタンスの内容が変更された場合に通知する所有者オブジェクトとそのフィールド名を指定します。コレクション型のクラスを作成した場合は、その中の要素の型は制限されません。要素として null も追加することができます。また、そのクラスの初期サイズはゼロになります。
newCollectionInstance	新しいコレクションインスタンスを返します。引数としてこのインスタンスの型 (またはインタフェース)、およびこのインスタンスの内容が変更された場合に通知する所有者オブジェクトとそのフィールド名を指定します。このコレクションクラスの要素の型は、引数 elementType で指定したもの (またはその型に代入可能なインスタンス) に制限されます。また、引数 allowNulls を設定すると、要素として null を追加可能になります。このクラスの初期サイズは、引数 initialSize で指定した値になります。

トランザクション

挿入処理と削除処理は、すべてトランザクションのコンテキスト内で実行する必要があります。トランザクションは、データベースの読み取り操作と更新操作の一貫性を保証します。トランザクションは、ディスクのクラッシュなど、データベースの一貫性を損なうシステム問題を回避します。さらに、トランザクションを使用すると、データベース中の同じデータを、別々のアプリケーションから同時に参照・更新できるようになります。トランザクションのコンテキスト内でこれらの操作を行えば、複数のアプリケーションによる同時更新によって整合性が失われる場合に、それらの更新をすべて中止することができます。

それぞれの持続性マネージャは、一度に 1 つのトランザクションしかサポートしませんが、このトランザクションは、トランザクション自体が保有する持続可能 (persistent-capable) クラスのすべてのトランザクションインスタンスに適用します。このトランザクションを操作するには、次のようにして持続性マネージャからトランザクションオブジェクトを取得します。

```
Transaction myTrans = myPersistenceManager.currentTransaction();
```

通常、アプリケーションは単一のデータベースに対するローカルトランザクションを実行します。その場合は、トランザクションオブジェクトのメソッドを次のように呼び出して、トランザクションを開始・終了することができます。

```
Transaction txn=pm.currentTransaction();
txn.begin();
...処理...
txn.commit();
}
catch (Exception e) {
txn.rollback();
}
```

持続性マネージャは、持続データのデータベースからの再取得を含め、データベースとのやり取りをすべて管理します。そのため、アプリケーションで行う必要があるのは、個々のトランザクションの開始と終了を指示することだけです。

それぞれのメソッドの詳細を 表 5-4 に示します。

表 5-4　トランザクションのメソッド

メソッド	説明
`begin`	新しいトランザクションを開始します。トランザクションがすでにアクティブになっている場合は JDOUserException をスローします。
`commit`	commit メソッドは次の処理を行います。 • 削除されたインスタンスを一時インスタンスにします。 • retainValues = false の場合は、持続インスタンスを hollow 状態に切り換え、主キーを除くフィールド値をすべて消去します。 • retainValues = true の場合は、持続インスタンスを persistent-nontransactional 状態に切り換え、現在のフィールド値をそのまま保ちます。
`rollback`	rollback メソッドは次の処理を行います。 • persistent-new 状態のインスタンスを一時インスタンスにし、フィールド値を持続化される前の状態に戻します。 • retainValues = false の場合は、持続インスタンスを hollow 状態に切り換え、主キーを除くフィールド値をすべて消去します。 • retainValues = true の場合は、持続インスタンスを persistent-nontransactional 状態に切り換え、フィールド値を変更前の状態に戻します。
`getPersistenceManager`	トランザクションインスタンスに関連付けられた持続性マネージャを返します。
`isActive`	アクティブなトランザクションがあるかどうかを通知します。

表 5-4　トランザクションのメソッド (続き)

メソッド	説明
`getRetainValues` `setRetainValues`	このフラグを true に設定すると、commit メソッドと rollback メソッドの処理が次のようになります。 • `commit` メソッドを呼び出したときに、持続インスタンスが persistent-nontransactional 状態に切り換わり、現在のフィールド値がそのまま保たれます。 • `rollback` メソッドを呼び出したときに、持続インスタンスが persistent-nontransactional 状態に切り換わり、フィールド値が変更前の状態に戻されます。 このフラグを false に設定すると、次のような処理が行われます。 • `commit` メソッドを呼び出したときに、持続インスタンスが hollow 状態に切り換わり、主キーを除くフィールド値がすべて消去されます。 • `rollback` メソッドを呼び出したときに、持続インスタンスが hollow 状態に切り換わり、主キーを除くフィールド値がすべて消去されます。
`getOptimistic` `setOptimistic`	Optimistic フラグを true に設定すると、トランザクションの管理方式としてオプティミスティック並行性制御が使用されます。getOptimistic メソッドを呼び出すと、現在の Optimistic フラグが指定した値に置き換えられます。true に設定した場合は、NontransactionalRead フラグも true に設定されます。デフォルト値は true です。

表 5-4　トランザクションのメソッド (続き)

メソッド	説明
`getNontransactionalRead` `setNontransactionalRead`	これらのメソッドは、トランザクションの外部から非トランザクションインスタンスの読み取りを可能にするかどうかを示すフラグを操作します。 NontransactionalRead フラグを true に設定すると、アクティブなトランザクションを使わずに照会やナビゲートを行うことができます。このフラグを false に設定すると、アクティブなトランザクションの外部で照会やナビゲートを行なった場合に例外がスローされます。デフォルト値は true です。
`getSynchronization` `setSynchronization`	管理接続方式とそれ以外の接続方式の両方で同期処理を行うことができます。トランザクションに登録した同期インスタンスは、別の setSynchronization メソッドで明示的に変更しないかぎり、そのトランザクションに登録されたままになります。 同期インスタンスは、トランザクションごとに 1 つしか登録できません。アプリケーションで、複数のインスタンスに同期コールバックを受け取らせる必要がある場合は、アプリケーションインスタンスでそれらのインスタンスを管理し、コールバックを渡します。すでに登録されている同期インスタンスがある場合は、新しく登録した同期インスタンスに置き換えられます。 トランザクションの commit メソッドを呼び出すと、コミット処理の前に beforeCompletion メソッドが呼び出されます。`beforeCompletion` メソッドは、ロールバック処理の前には呼び出されません。また、トランザクションの commit メソッドや rollback メソッドを呼び出すと、コミット処理やロールバック処理の後で、`afterCompletion`メソッドが呼び出されます。`afterCompletion (int status)` メソッドの引数は、`Status.STATUS_COMMITTED` と `Status.STATUS_ROLLEDBACK`のどちらかになります。

表 5-4　トランザクションのメソッド (続き)

メソッド	説明

`QueryTimeout`
`UpdateTimeout`
	データベースのデッドロックを防止するため、トランザクションインスタンスの照会や更新を開始するまでのタイムアウト時間を指定します。
	タイムアウト時間は秒数で指定します。ゼロは無制限を意味します。例:
	`tx.setQueryTimeout(6);`
	`tx.setUpdateTimeout(10);`
	PointBase は、現時点では `PreparedStatement.setQueryTimeout()` に対応していません。タイムアウト時間をデフォルト値 (現行リリースでは 60 秒) から変更したい場合は、URL または `pointbase.ini` ファイルに、`locks.timeout=value` という文字列を追加します。ただし、`locks.timeout=0` を指定すると、タイムアウト時間が 0 秒に設定されることに注意してください。`setQueryTimeout(0)` の場合は、タイムアウト時間が無制限になります)。
	`Locks.timout` は、クライアント側ではなくサーバー側で設定します。このことは、すべての接続に対し指定の値がホールドすることを意味します。

トランザクション遮断期間レベル

トランザクション遮断期間レベルは、トランザクションの並行処理の程度を指定したものです。複数のユーザーが同じデータベースにアクセスする場合は、ユーザーから見たデータの信頼性と、処理性能とのバランスを定める必要があります。データベースにアクセスするときに、次の不整合が発生する可能性があります。

- ダーティー読み取り

  まだコミットされていないデータを読み取ることです。トランザクション A がデータを更新し、そのデータがコミットされる前に、更新後のデータをトランザクション B が読み取り、更新内容がコミットされる代わりにロールバックされると、トランザクション B が読み取ったデータは不正になります。

- 反復不能読み取り

同じトランザクションの中で、同じ照会を繰り返し実行した場合に、異なる結果が返されることです。あるトランザクションが読み取った行を、他のトランザクションが更新 (または削除) してコミットすると、最初のトランザクションのそれ以降の読み取りで、今までとは別のデータが返されます。この現象は、同じデータに対して、別々のユーザーが読み取りと更新を同時に行なった場合に発生する可能性があります。

- 幽霊行の挿入

  読み取りの対象になる行が、他のユーザーのトランザクションによって挿入されることです。たとえば、あるユーザーが同じ SELECT 文を 2 回実行したときに、1 回目の実行では 4 つの行が、2 回目の実行では 5 つの行が選択される場合があります。これは、1 回目の実行と 2 回目の実行の間に、この照会の条件に適合する行を別のユーザーが挿入したためです。

遮断期間レベルを高くするほど、これらの不整合が発生しにくくなりますが、オーバーヘッドが増大するためにアプリケーションの処理性能が低下し、システムの並行性に悪影響が現れます。

透過的な持続性では、データベースのデフォルトの遮断期間レベル (Oracle および MSSQL では TRANSACTION_READ_COMMITTED、PointBase では TRANSACTION_SERIALIZABLE) が使用されます。

java.sql.Connection では、次の SQL 名を使用しています。

```
int TRANSACTION_NONE = 0;
int TRANSACTION_READ_UNCOMMITTED = 1;
int TRANSACTION_READ_COMMITTED   = 2;
int TRANSACTION_REPEATABLE_READ  = 4;
int TRANSACTION_SERIALIZABLE     = 8;
```

それぞれの遮断期間レベルの設定で、どの不整合が発生する可能性があるかを 表 5-5 に示します。

表 5-5　遮断期間レベル

レベル	ダーティー読み取り	反復不能読み取り	幽霊行の挿入
TRANSACTION_READ_UNCOMMITTED	可能性あり	可能性あり	可能性あり
TRANSACTION_READ_COMMITTED	可能性なし	可能性あり	可能性あり
TRANSACTION_REPEATABLE_READ	可能性なし	可能性なし	可能性あり
TRANSACTION_SERIALIZABLE	可能性なし	可能性なし	可能性なし

遮断期間レベルを TRANSACTION_NONE に設定した場合は、トランザクションの使用がサポートされなくなります。

注 - Oracle は、TRANSACTION_READ_UNCOMMITTED と TRANSACTION_REPEATABLE_READ に対応していません。透過的な持続性では、設定した遮断期間レベルはチェックされません。未対応の遮断期間レベルを設定すると、データベースの制約に違反してしまいます。

並行性制御

データベース環境でのプログラミングは、トランザクションが基本になります。トランザクションを使用することで、複数のユーザーによるデータベースの同時アクセスを正しく処理し、データベースの整合性を保つことができます。これは、操作の挿入や削除はすべてトランザクションのコンテキスト内で行われなければならないということを意味します。

透過的な持続性では、次の 2 通りの方法で並行トランザクションを処理することができます。

- オプティミスティックトランザクション制御 (デフォルト)

オプティミスティック並行性制御では、別のトランザクションが同じデータを変更する前にトランザクションが完了するようになっています。システム側ではトランザクションがコミットすることを想定しています。しかしシステムが衝突を検出した場合、つまり最初のトランザクションの処理中に、別のトランザクションが同じデータに対して変更を行った場合、トランザクションをロールバックします。

持続性マネージャは、アプリケーションがトランザクションを開始したときに、使用されているデータベースレコードの状態をすべて記録します。さらに、トランザクションをコミットする前に、データベースレコードのその時点での状態をトランザクション開始時の状態と比較し、トランザクションの実行中に、他のユーザーが同じデータストアを更新しなかったかどうかを確認します。

- データストアトランザクション制御

  データストアトランザクション制御では、トランザクションは、データベースおよび指定されたトランザクションの遮断期間レベルによって処理されます。詳細については、121 ページの「トランザクション遮断期間レベル」 を参照してください。

  アプリケーションがトランザクションを開始すると、持続性マネージャはデータベースそのものにトランザクションを開始するように指示します。つまり、データに初めてアクセスしてからコミットするまでの間、データストアトランザクションがアクティブであることを意味します。

持続マネージャファクトリには、デフォルトの並行性管理方式を設定するメソッドが用意されています。トランザクションオブジェクトにはトランザクションを開始する前に並行性制御ストラテジーを設定できるメソッドが用意されています。

オプティミスティックトランザクションの実行時間は、データベーストランザクションより長くなります。これは、それぞれのオプティミスティックトランザクションが 2 つのデータベーストランザクションで構成されていることが原因です。照会のための読み取りトランザクション (照会の完了とともに終了します) と、コミットのための書き込みトランザクションです。さらに、更新をコミットする際に、使用されているオブジェクトに対応する行をチェックする必要があるため、データベースに余分な負担がかかります。しかし、オプティミスティックトランザクションでは、データベースレコードのロック時間を最小限に抑えることができるため、並行処理性能は最適化されます。

オプティミスティックトランザクションでエラーが発生すると、エラーが発生したオブジェクトの配列を含んだ例外がスローされます。

データベーストランザクションの回復は、データベースが処理します。たとえば、データベースはデッドロックやタイムアウトの発生を検出し、トランザクションを適切に中止またはロールバックします。

オプティミスティックトランザクションは、Web アプリケーションのように、ユーザーからの入力を待って処理を行うアプリケーションに適しています。サーバー上で即座に実行するアプリケーション (バッチアプリケーションや、状態を持たないセッション Bean やサーブレット) では、データベーストランザクションを使用してください。

値の保持

持続性マネージャを設定すると、トランザクションの外部で値を保持することができます。この機能は、オプティミスティックトランザクションを使用する場合や、トランザクションの外部でデータを選択したい場合に便利です。このことは、トランザクションのコンテキストの外側であっても、データがローカルでキャッシュされることを意味します。そのため、データアクセスは高速化しますが、データベースが IDE の外側で更新された場合にローカルキャッシュに陳腐なデータを保有する危険性があります。

`retainValues` フラグを false に設定した場合は (値の保持機能を無効にした場合は)、デフォルトフェッチグループ中のいずれかのフィールドにはじめてアクセスしたときに、そのグループに含まれているすべてのフィールドがまとめて読み取られます。デフォルトフェッチグループに含まれていないフィールドについては、それぞれのフィールドに初めてアクセスしたときに個別に読み取られます。

注 - `retainValues()` が true に設定されている場合、次の状態になることがあります。

```
tx.begin();
Object o1 = c.get(i);
c.add(o);       //再ロードが発生し、既存の重複要素がすべて削除されます。
o1 == c.get(i); // 新しいコレクションの内容に応じて、true または false
が
                // 返されます。
```

オプティミスティック並行性制御によるコーディング

Optimistic フラグをtrue に設定すると、NontransactionalRead フラグも true に設定されます。

オプティミスティック並行性制御では、トランザクションの実行時間が短いほど、正常にコミットされる可能性が高くなります。トランザクションの実行時間が長引くと、他のトランザクションによって更新対象のデータが先に更新されてしまうことになりかねません。その場合は、システムがそのことを検出し、フラッシュ時やコミット時に JDODataStoreException 例外をスローします。その場合は、現在のトランザクションをロールバックする必要があります。

オプティミスティックトランザクションが適しているのは、実行時間が長い複数のトランザクションによる、同じインスタンスの更新がほとんど発生しない場合です。その場合は、コミットを行うまでは、データベースに排他アクセスを適用する必要がないため、処理性能が高くなります。

オプティミスティックトランザクションでは、データベースから読み取ったり、照会によって抽出したインスタンスは、修正するか、削除するか、アプリケーションで明示的に指定しないかぎり、トランザクションインスタンスになりません。

コミット時には、トランザクションインスタンスとデータベースの現在の内容が比較され、インスタンスのトランザクション開始前の「事前イメージ」が、データベースの現在の内容と同じかどうかがチェックされます。

データベースの現在の内容が事前イメージと一致していないインスタンス、すなわち他のトランザクションによって更新されたインスタンスが見つかった場合は、それらのインスタンスのリストを含んだ例外がスローされます。ただし、オプティミスティックトランザクションはまだアクティブのままで、このトランザクションをロールバックする必要があります。

オプティミスティックモードで動作する 透過的な持続性 アプリケーションは、同時更新が発生した場合に、JDODataStoreException をスローします。

オプティミスティックトランザクションを使用する場合は、**トランザクション**の Optimistic フラグをセットします。

フラッシュ時やコミット時は、同じグループに属しているフィールドだけが、同時更新のチェックの対象になります。

アプリケーションがデータの更新をデータベースにコミットする準備ができると、システムは該当するデータが他のトランザクションによって更新されていないかどうかを確認します。該当するデータが他のトランザクションによって更新されていない場合は、トランザクションを正常終了することができます。他のトランザクションによって更新をロールバックする必要があります。

注 - 並行トランザクションの衝突によってトランザクションがロールバックされた場合は、通常は他のトランザクションによってデータベースの値が変更されています。

データストア並行性制御によるコーディング

データストア並行性制御では、使用しているデータベースと、設定した遮断期間レベルに応じた処理が行われます。

データストア並行性制御では、オブジェクトの更新後にトランザクションを確実に続行し、コミットすることができます。ただし、デッドロックやエラーが発生しないことが条件になります。

デッドロックが発生するのは、複数のトランザクションが同じレコード群を更新しようとした場合です。たとえば、1 つのトランザクションがレコード A をロックし、レコード B のロックを取得できるまで待機します。それと同時に、もう 1 つのトランザクションがレコード B をロックし、レコード A のロックを取得できるまで待機します。いずれのトランザクションも保有しているロックを手放さないため、互いに取得が不可能なロックを待っていることから双方でデッドロックが発生します。デッドロックが発生した場合の処理は、データベース管理システムによって異なります。

たとえば透過的な持続性を使用しているアプリケーションで、トランザクション A が持続オブジェクト O1 を更新し、現在は持続オブジェクト O2 を更新しようとしています。このときに、別のトランザクション B が持続オブジェクト O2 を更新していて、現在は持続オブジェクト O1 を更新しようとしています。これはデータベースのデッドロック状態です。一方のトランザクションがオブジェクト O1 の読み取りを終え、オブジェクト O2 の更新を開始しようとし、同時にもう一方のトランザクションがオブジェクト O2 の読み取りを終え、オブジェクト O1 の更新を開始しようとした場合も、同様のデッドロックが発生します。これらのデッドロックの結果は、使用している DBMS によって異なります。

Microsoft SQL Server では、デッドロックは検出されません。そのため、トランザクションの `setQueryTimeout()` メソッドや `setUpdateTimeout()` メソッドを呼び出して、照会や更新がタイムアウトになるまでの時間を指定しておく必要があります。デフォルト設定のままでは、永久にタイムアウトになりません。

これに対して、Oracle では、ユーザーが該当するトランザクションをコミットしたときに、並行トランザクションのデッドロックが検出されます。その場合は、先にコミットしたトランザクションが正常終了し、もう一方のトランザクションがロールバックされます。

一般に、データストア並行性制御では、トランザクションの実行時間を短くしないと、他のトランザクションの処理が妨害されてしまいます。ただし、排他制御モードで実行されるアプリケーション (支払伝票発行アプリケーションのように、他のアプリケーションを排除して、データベースやその一部を制御するアプリケーション) では、このことはほとんど問題になりません。

データベースにアクセスする

ここでは、持続可能クラスのインスタンスのライフサイクルについて説明します。持続可能クラスには、クラス (Bean) の開発者が指定した動作に加えて、参照拡張機能や透過的な持続性から提供される動作が組み込まれます。このような持続可能クラスの拡張により、透過的な持続性での持続状態の自動取得機能が有効になり、透過的な持続性インスタンスと通常のインスタンスとの違いを意識しないアプリケーション開発が可能になります。

持続可能クラスには、データベース中のデータをモデル化した持続フィールドと関係フィールドが含まれています。アプリケーションがデータベース中の特定のエンティティを操作するには、そのデータをモデル化した持続可能クラスのインスタンスを作成し、操作する必要があります。たとえば、アプリケーションで従業員のデータベース表をモデル化した `Employee` クラスを使用している場合は、`Employee` クラスのインスタンスが必要になります。

データを表現した持続インスタンスを作成すると、それぞれのインスタンスの動作が、モデル化の元になったトランザクションストアにリンクされます。透過的な持続性は、インスタンスの値の変更を自動的に追跡し、トランザクションのデータ整合性を保ちながら、データベースから値を再取得したり、データベースに値を保存したりします。そのため、アプリケーションコードでは、持続インスタンスを Java インスタンスとして操作するだけですみます。その操作に対応するデータベース操作は、すべて透過的な持続性実行環境が行います。

持続インスタンスは、ライフサイクル全体に渡って状態を次々と変え、最終的には JVM によってガベージコレクトされます。状態がどのように切り替わるかは、そのインスタンスに対する直接的な操作に加えて、アプリケーションや実行環境が持続性マネージャ上で実行した処理によって決まります。

トランザクションを開始した後で、インスタンスとデータベースとの整合性が失われた状態になることがあります。この状態を「dirty」状態と呼びます。たとえば、トランザクションの中でインスタンスを削除・変更したり、新しく持続インスタンスを作成すると、それらのインスタンスの状態が dirty になります。

透過的な持続性は、持続インスタンスの現在の状態をデータベースに保存します。この処理を「フラッシュ」と呼びます。フラッシュが行われても、インスタンスの dirty 状態は変化しません。

ここからは、アプリケーションで持続可能クラスのインスタンスを作成および使用する方法について説明します。また、インスタンスの操作や状態について透過的な持続性で使用する用語も、いくつか紹介します。インスタンス状態は、基本的には実行環境が使用する管理情報ですが、場合によってはアプリケーションによる参照・再設定が必要になります。

オーバーフロー保護

データベースの書き込み保護は、データベースのドライバが行います。透過的な持続性自身は、書き込みチェックを行いません。

データベースからの読み取りについては、データベースから返された値が数値で、その値が該当するフィールド型の `MIN_VALUE` 値より小さいか、または `MAX_VALUE` 値より大きい場合に、JDOUserException がスローされます。たとえば、`short` 型の値の範囲を -32768 以上、かつ 32768 以下に設定するには、次のようにします。

```
java.lang.Short:
public static final short MIN_VALUE = -32768;
public static final short MAX_VALUE = 32767;
```

読み取り時のオーバーフローチェックは、`short`、`int`、`long`、`byte`、`Short`、`Integer`、`Long`、`Byte`の各データ型に適用されます。

持続データの挿入

クライアントから新しいレコードが渡された場合は、それに対応する新しい持続インスタンスを作成します。

```
Employee newEmployee = new Employee(<data>);
// インスタンスは "transient" 状態
pMgr.makePersistent(newEmployee);
// インスタンスは "persistent-new" 状態
```

トランザクションをコミットすると、透過的な持続性実行環境によって、このインスタンスにカプセル化されたデータ用の SQL 挿入処理 (またはそれに相当する処理) が生成されます。

この処理は 2 段階の手順になります。newEmployee インスタンスを作成した段階では、そのインスタンスは持続性マネージャには関連付けられず、トランザクションが終了しても自動保存は行われません。makePersistent() を呼び出すことで、このインスタンスが持続性マネージャに関連付けられ、アプリケーションのために、その値が管理されるようになります。

持続データの更新

持続インスタンスのデータを変更するには、そのインスタンスを直接操作します。

```
selectedEmployee.setVacationHours(132);
// インスタンスは "dirty" 状態
```

トランザクションをコミットすると、透過的な持続性実行環境によって、このインスタンスにカプセル化されたデータ用の SQL 更新処理 (またはそれに相当する処理) が生成されます。トランザクションをコミットした後は、このインスタンスの状態がリセットされます。

透過的な持続性は、索引によって要素を削除する SCO コレクションに対応していません。その元になっているコレクションが、更新処理の間にデータベースからの再取得によって変更される可能性があるからです。

持続データの削除

持続インスタンスで表現されたデータを削除するには、持続性マネージャのメソッドを呼び出します。

```
persistenceManager.deletePersistent(selectedEmployee);
// インスタンスに削除マークが付きます
```

トランザクションをコミットすると、透過的な持続性実行環境によって、このインスタンスで表現されたデータ用の SQL 削除処理 (またはそれに相当する処理) が生成されます。

透過的な持続性は、次の 2 種類の削除方式に対応しています。

- なし (デフォルト)

  オブジェクトは削除しますが、そのオブジェクトに片方向で関係しているオブジェクトは削除しません。

  関係性の管理において、削除されるオブジェクトと関連オブジェクトの関係は無効となります。

- カスケード

  フラッシュ時やコミット時に、オブジェクトを削除するとともに、そのオブジェクトに関係しているオブジェクトもすべて削除します。

  たとえば、Department (部署) と Employee (従業員) という 2 つのクラスがあり、Department には Employee のコレクションが、 Employee には Department の参照が含まれている場合を考えてみましょう。

  Employee 関係に対してカスケード削除を指定しておくと、 Department インスタンスを削除したときに、その Department に関連付けられている Employee インスタンスもすべて削除されます。

  Employee 関係に対してカスケード削除を指定しておくと、 Department インスタンスを削除したときに、その Department に関連付けられている Employee インスタンスもすべて削除されます。このときに、Employee 関係に対してもカスケード削除を指定している場合は、その Department に関連付けられているその他の Employee インスタンスもすべて削除されます。

削除方式は、持続可能クラスの「プロパティ」ウィンドウ の「削除アクション」フィールドで指定することができます。77 ページの「オプションとプロパティの設定」 を参照してください。

注 - 1 対多や多対多の関係の「多」の側でカスケード削除を指定すると、削除の結果が不正になることがあります。カスケード削除は、1 対 1 の関係か、または 1 対多の関係の「1」の側だけで設定してください。

多対多の関係の一方の側のオブジェクトをすべて削除する例として、プロジェクトと従業員の間の関係から、プロジェクトをすべて削除する場合を考えてみましょう。このコードは次のようになります。

```
Collection p = e.getProjects();
Object[] a = p.toArray();
p.clear();
pm.deletePersistent(a);
```

データベースの照会

照会を使用すると、SQL 文を個別に作成しなくても、持続データにアクセスすることができます。任意の数の別々のデータベースに対して、同じコードを適用することができます。また、コードを変更することなく、別のデータベースに持続可能クラスを再マップすることも可能です。この処理は、スキーマの異なるデータベースにも適用することができます。

データベースのデータが必要な場合は、`newQuery()` メソッドを使用して持続性マネージャから照会オブジェクトを取得し、照会インタフェースのメソッドを使用して照会を定義し、その照会を実行します。この例を次に示します。

```
Class empClass = Employee.class;
Collection empExtent = pMgr.getExtent(empClass, false);
String empFilter = "id == 59439";
Query q = pMgr.newQuery(empClass, empExtent, empFilter);
Collection result = (Collection) q.execute();
```

照会を定義するための要素を 表 5-6 に示します。

表 5-6　照会の要素

要素	必須 / 任意	説明
候補クラス	必須	照会に対して考慮される候補コレクション内のインスタンスのクラスを定義します。照会フィルタの中で指定する名前は、このクラスの構成要素名として解釈されます。このクラスは持続可能クラスでなければなりません。このクラスは、newQuery メソッドの引数か、Query インタフェースの setClass メソッドで定義します。
候補コレクション	必須	候補クラスの範囲コレクションです (範囲コレクションについては、PersistenceManager.getExtent メソッドを参照してください)。このコレクションが、照会の入力コレクションになります。このクラスは、newQuery メソッドの引数か、照会インタフェースの　setCandidates　メソッドで定義します。メモリーコレクションを対象にした照会は実行できません。照会の候補コレクションとして指定できるのは、範囲コレクションだけです。
照会フィルタ	必須	候補コレクションの中の、どのオブジェクトを照会の結果として返すかを指定した文字列です。このクラスは、newQuery メソッドの引数か、照会インタフェースの setFilter メソッドで定義します。省略時の照会フィルタは true になります。その場合は、すべてのインスタンスが返されます。
照会パラメータ	任意	照会では、1 つ以上のパラメータを指定しておき、照会の実行時に実際の値に置き換えることができます。照会パラメータの定義には、Java 言語の仮引数の構文を使用します。照会パラメータは、照会インタフェースの declareParameters メソッドで定義します。

表 5-6　照会の要素 (続き)

要素	必須 / 任意	説明
照会変数	任意	照会フィルタでは変数を使用することができます。この変数は、コレクション型の関係をナビゲート (追跡) するときに必要になります。照会変数の定義には、Java 言語のローカル変数の構文を使用します。照会パラメータは、Query インタフェースの `declareVariables` メソッドで定義します。
インポート文	任意	照会パラメータや照会変数で、候補クラス以外のクラスの構成要素を指定したい場合は、曖昧さを防ぐため、そのクラス名を `import` 文で宣言する必要があります。この文の構文は、Java 言語の `import` 文と同じです。照会パラメータは、Query インタフェースの `declareImports` メソッドで定義します。
順序付け	任意	結果クラスのフィールドを基準にして、照会の結果セットを並べ替えることができます。そのためには、並べ替えの基準にするフィールドのリストと、昇順 / 降順を示す情報を指定します。照会パラメータは、Query インタフェースの `setOrdering` メソッドで定義します。

持続性マネージャは、照会インスタンスのファクトリ (作成元) になり、照会は持続性マネージャのコンテキスト内で実行されます。照会によって返されるすべての持続インスタンスは、持続性マネージャとそのトランザクションに関連付けられます。この持続性マネージャの自動更新/再表示 (Update/Refresh) 処理には、これらのインスタンスが含まれることになります。同じ持続性マネージャの中で、複数の照会インスタンスを同時にアクティブにすることができます。

作成する照会ごとに、持続性マネージャの newQuery() メソッドを呼び出します。先ほどの例では、候補クラス、候補コレクション、およびフィルタを指定して、照会インスタンスを作成しています。その他のメソッドを 表 5-7 に示します。

表 5-7　各種の newQuery メソッド

メソッド	説明
`Query newQuery()`	空の照会インスタンスを作成します。
`Query newQuery (Object query)`	別の照会から新しい照会インスタンスを作成します。引数として指定する照会インスタンスは、別の実行環境からシリアライズ / 復元可能なものか、現在の持続性マネージャに割り当てられているものでなければなりません。引数として指定した照会インスタンスから、結果クラス、フィルタ、インポート宣言、変数宣言、パラメータ宣言、順序付けの設定が取り出され、新しい照会インスタンスにコピーされます。ただし、候補コレクションの設定は無視されます。
`Query newQuery (Class cls)`	候補クラスを指定して照会インスタンスを作成します。
`Query newQuery (Class cls, Collection cln)`	候補クラスと候補コレクションを指定して、照会インスタンスを作成します。
`Query newQuery (Class cls, String filter)`	候補クラスとフィルタを指定して、照会インスタンスを作成します。
`Query newQuery (Class cls, Collection cln, String filter)`	候補クラス、候補コレクション、およびフィルタを指定して、照会インスタンスを作成します。

照会インタフェースのメソッドを 表 5-8 に示します。

表 5-8　照会インタフェースのメソッド

メソッド	説明
`void setClass (Class resultClass)`	照会インスタンスの候補クラスを設定します。
`void setCandidates (Collection candidateCollection)`	照会インスタンスの候補コレクションを設定します。
`void setFilter (String filter)`	照会インスタンスの照会フィルタを設定します。
`void declareParameters (String parameters)`	照会インスタンスのパラメータ宣言を設定します。このメソッドを使用して、後続の execute メソッドで使用するパラメータの型と名前を定義します。
`void declareVariables (String variables)`	照会インスタンスの変数宣言を設定します。このメソッドを使用して、フィルタで使用する変数の型と名前を定義します。パラメータと異なり、変数の値は execute メソッドでは指定しません。
`void declareImports (String imports)`	照会インスタンスのインポート文を設定します。
`void setOrdering (String ordering)`	照会インスタンスの順序付け文を設定します。
`void setIgnoreCache (boolean flag); boolean getIgnoreCache ()`	キャッシュの中の変更された値を無視することで、照会から適切な結果が返されるようにします。この機能が役立つのは、オプティミスティックトランザクションだけです。この機能を使用すると、キャッシュの中の更新されたインスタンスを考慮に入れずに、データベースから照会結果が返されます。`setIgnoreCache (false)` はサポートされていません。
`void compile ()`	照会インスタンスの設定内容をチェックし、不整合が見つかった場合は例外をスローします。

照会インタフェースには、指定された引数に基づいて照会を実行するメソッドが用意されています。`Query.execute` は必ずオブジェクトのコレクションを戻します。先ほどの例では、照会で選択されるオブジェクトは 1 つしかありませんが、`q.execute` の結果の動的な型は `Collection` です。そのため、結果コレクションの反復処理が必要になります。この場合は、`Iterator.next` によって `Employee` が返されます。

照会フィルタ

照会フィルタは、コレクションの中のインスタンスごとに評価される Java の論理式です。照会フィルタを指定しなかった場合は、`true` というデフォルトの照会フィルタが使用されます。その場合は、入力コレクションの中の該当するクラスのインスタンスがすべて抽出されます。

単純なフィルタ式

最も単純なフィルタ式は、候補クラスのフィールドを定数と比較する関係式です。

```
q.setFilter("id == 59439");
```

また、論理演算子 `&`、`&&`、`|`、`||`、`!`を、算術演算子 `+`、`-`、`*`、`/` と同様に使用できます。たとえば、次の 1 行目のコードでは、ファーストネームが John で、かつラストネームが Jones の要素が抽出されます。また、2 行目のコードでは、ファーストネームが John か、または給与が 200,000 を超えている要素が抽出されます。

```
q.setFilter("firstname == \"John\" & lastname == \"Jones\"");
q.setFilter("firstname == \"John\" | salary > 200000.0");
```

フィルタ式の中の識別名は、パラメータまたは変数として定義したものや、クラス名としてインポートしたものを除いて、すべて候補クラスのフィールド名として取り扱われます。たとえば、上のコードの `firstname`、`lastname`、`salary` は、いずれも `Employee` クラスのフィールド名です。また、Java 言語と同様に、予約語 `this` は、候補コレクションの要素のうち、現在評価しているものを示しています。

次のフィルタ式は同じ意味になります。

```
q.setFilter("firstname == \"John\"");
q.setFilter("this.firstname == \"John\"");
```

代入演算子、前置 / 後置インクリメント演算子、前置 / 後置デクリメント演算子は使用できません。そのため、フィルタ式を使用しても、抽出されたオブジェクトは影響を受けません。使用できるメソッドは、`Collection.contains`、Collection.isEmpty、`String.startsWith`、および `String.endsWith` です。Java 言語とは違って、基本型とラッパークラスのインスタンスとの一致比較や順序比較

は、Date フィールドと Date パラメータとの一致比較や順序比較と同様に有効になります。また、その他の比較演算子 (<、<=、>、>=、!=) や、&、&&といった論理演算子も使用することができます。

照会パラメータ

照会パラメータは、照会宣言の中で固定されていない唯一の定義要素です。照会パラメータの実際の値は、execute メソッドに渡します。次の照会は、execute メソッドで指定されたファーストネームを持つ従業員を返します。

```
Class empClass = Employee.class;
String filter = "firstname == name";
Collection empExtent = pMgr.getExtent(empClass, false);
String param = "String name";
Query q = pMgr.newQuery(empClass, empExtent, filter);
q.declareParameters(param);
Collection result = (Collection) q.execute("John");
```

この例では、firstname が持続可能クラス Employee のフィールドで、name が照会パラメータの名前です。このパラメータの実際の値は、execute メソッドの引数として指定します。したがって、この例の q.execute("John") は、firstname フィールドの値が John になっている Employee インスタンスのコレクションを返します。q.execute("Sue") のように、この execute メソッドを再び呼び出して別のパラメータ値を渡すと、同じ照会インスタンスを再使用して、別の名前の Employee インスタンスを抽出することができます。

照会パラメータの宣言では、パラメータの型と名前を定義します。execute メソッドに渡すパラメータ値は、ここで定義した型と互換性のあるものでなければなりません。1 つの照会で複数のパラメータを定義することもできます。その場合は、それらのパラメータの値を、宣言した順に execute メソッドに渡します。

execute メソッドの個々の引数は、該当するパラメータの値と、基本型のパラメータのラップ値のどちらかの値を持つオブジェクトです。

注 - execute メソッドに渡したパラメータ値は、現在の実行だけで使用されます。それ以降の実行では、この値を使用することはできません。

照会 API から呼び出されて照会要素を定義するメソッド setClass、setCandidates、setFilter、declareImports、declareParameters、declareVariables、setOrdering は追加ではなく置換されます。このことは、照会実行前にこれらのメソッドが 2 回呼び出された場合、2 回目の呼び出しが 1 回目のメソッドによる設定を上書きすることを意味します。次のサンプルコードでは、単一のパラメータlastname を使用する照会が定義されています。

```
Query query = pm.newQuery(Employee.class);
query.declareParameters("String firstname");
query.declareParameters("String lastname");
...
```

2 つのパラメータを使用する照会を作成する場合、単一の declareParameters で定義する必要があります。

```
Query query = pm.newQuery(Employee.class);
query.declareParameters("String firstname, String lastname");
```

関係のナビゲート

照会フィルタでは、Java 言語と同様に関係をナビゲート (追跡) することができます。次の照会は、Employee インスタンスのうち、関係の相手側の Department インスタンスの name フィールド値が、パラメータとして渡された値と同じものを返します。

```
Class empClass = Employee.class;
String filter = "department.name == depName";
Collection empExtent = pm.getExtent (empClass, false);
String param = "String depName";
Query q = pm.newQuery (empClass, empExtent, filter);
q.declareParameters(param);
Collection emps = (Collection) q.execute ("R&D");
```

コレクション型の関係をナビゲートするには、照会変数を使用します。このフィルタ式には、変数の範囲を指定する Collection.contains メソッドが含まれています。メソッドの呼び出しの後に、コレクション関係内のインスタンスごとに条件を定義する論理式が続きます。次の照会は、Department インスタンスのうち、パラメータとして渡された値より給与が高い Employee インスタンスが 1 つ以上関連付けられているものをすべて返します。式 emps.contains (emp) は、コレクション型の関

係の相手側の Employee インスタンスを、変数 emp の有効範囲として定義しています。また、式 emp.salary > sal は、これらの Employee インスタンスの選択条件です。

```
Class depClass = Department.class;
Collection deptExtent = pm.getExtent (depClass, false);
String imports = "import mypackage.Employee";
String vars = "Employee emp";
String filter = "emps.contains (emp) & emp.salary > sal";
Query q = pm.newQuery (depClass, deptExtent, filter);
q.declareParameters(param);
q.declareVariables (vars);
Collection deps = (Collection) q.execute (new Float (30000.));
```

透過的な持続性では、関係フィールドと持続インスタンスを比較することができます。たとえば、フィルタ式で、Employee の departmentdepartment フィールドと Department の照会パラメータを、次のような形で比較することが可能です："department == dept"。

注 - 透過的な持続性では、同じ変数に複数の contains 節を適用することはできません。また、宣言した変数はフィルタの中だけでしか使用できません。

順序付けの指定

次の照会は、給与が 30000 を超えている Employee インスタンスを、給与の昇順に選択します。

```
Class empClass = Employee.class;
Collection empExtent = pMgr.getExtent(empClass, false);
String empFilter = "salary > 30000.0";
Query q = pMgr.newQuery(empClass, empExtent, empFilter);
q.setOrdering("salary ascending");
Collection result = (Collection) q.execute();
```

setOrdering メソッドに渡す引数では、複数の順序付けを宣言することができます。その場合は、それらをコンマ (,) で区切って指定します。結果セットは、1 回目の順序式によって順序付けされます。最初の順序式の結果が同じ値となったエントリは、2 回目の式で順序付けされ、それでも同じ値となる場合は、さらに順序式が行われます。関係性のナビゲーションを含む順序式を指定することも可能です。

次の順序付け宣言により、上記の照会は関連部署の名前の昇順で従業員を返します。同じ部署の従業員は、給与の高いものから順序付けされます。

```
q.setOrdering("department.name ascending, salary ascending");
```

文字列の処理

フィルタ式では、== 演算子や != 演算子を使用して、String 型のフィールドや値を比較することができます。また、透過的な持続性は、ワイルドカードを使用した照会にも対応しています。その場合は、String メソッド、`startsWith` や `endsWith` を使用します。次のフィルタ式は、ファーストネームが `M` で始まる `Employee` インスタンスをすべて選択します。

```
String empFilter = firstname.startsWith("M");
```

オプティミスティックトランザクションとデータストアトランザクションでの照会の違い

データストアトランザクションで照会を実行すると、最初にそのトランザクションによる更新内容がフラッシュされ、その後でデータストアで照会が評価されます。そのため、照会の結果には、照会の実行前にトランザクションで行なった更新内容がすべて反映されます。これに対して、オプティミスティックトランザクションでは、フラッシュが行われないため、照会の結果には現在の更新内容は反映されず、現在のトランザクションで更新されているために、実際には照会の条件を満たしていないインスタンスが抽出される可能性があります。なお、非トランザクション読み取りが許可されている場合は、トランザクションの外部で照会を実行することができます。

式の表現

透過的な持続性では、次の要素を組み合わせて式を指定することができます。

■ 演算子 − Java 言語で定義されているすべてのデータ型に適用することができます。演算子の一覧を 表 5-9 に示します。

表 5-9　式の演算子

演算子	説明
==	等しい
!=	等しくない
>	より大きい
<	より小さい
>=	～以上
<=	～以下
&	論理 AND (ビットごとの AND ではありません)
&&	条件 AND
|	論理 OR (ビットごとの OR ではありません)
||	条件 OR
~	論理値または整数値のビットごとの反転
+	2 項または単項の追加、または String の連結
-	2 項の減算または数値符号の反転
*	乗算
/	除算
!	論理反転

■ ( ) − 演算子の優先順位を明示的に指定するときに使用します。

■ キャスト演算子 (クラスの型変換)

■ 比較のための数値演算項の値の格上げ

■ ラッパー型 (`Boolean`、`Byte`、`Short`、`Integer`、`Long`、`Float`、および `Double`) ならびに `BigDeci`

■ `mal` および `BigInteger` のオブジェクト値フィールドに対する一致比較、順序比較、および算術演算

この処理では、比較項や演算項としてラップ値が使用されます。

- PersistenceCapable 型のオブジェクト値フィールドの一致比較

  この処理では、参照の透過的な持続性ID が使用されます。すなわち、2 つのオブジェクトの透過的な持続性ID が等しい場合は、それらが一致していると見なされます。

- PersistenceCapable 以外の型のオブジェクト値フィールドの一致比較

  この処理では、該当するフィールド型の equals メソッドが使用されます。

- 文字列の連結

  文字列の連結だけが使用可能です。たとえば、String + primitive は使用できません。

例

ここからは、よく使われる照会の例をいくつか取り上げます。それぞれの例には、説明に加えて、同じ機能を持つ ANSI SQL 文を付け加えてあります。

これらの例では、次の持続可能クラスを使用しています。

```
package com.xyz.hr;
class Employee {
String name;
Float salary;
Department dept;
Employee boss;
}
package com.xyz.hr;
class Department {
String name;
Collection emps;
}
```

単一の表の選択

次の照会は、範囲コレクションに含まれている Employee インスタンスをすべて選択します。

```
同等の ANSI SQL 文: SELECT * FROM EMPLOYEE

Class empClass = Class.forName("com.xyz.hr.Employee");
Collection clnEmployee = pm.getExtent (empClass, false);
String filter = "true";
Query q = pm.newQuery (empClass, clnEmployee, filter);
Collection emps = (Collection) q.execute ();
```

制約条件を使用した単一の表の選択

次の照会は、Employee インスタンスのうち、フィールド値が判定条件を満たしているもの (この場合は給与が 30000 を超えているもの) をすべて選択します。

```
同等の ANSI SQL 文: SELECT * FROM EMPLOYEE WHERE SALARY > 30000
```

salary フィールドの Float 値はアンラップされ、定数値と比較されます。候補インスタンスの salary フィールド値が null の場合は、比較のための値のアンラップを行えないため、そのインスタンスは拒絶されます。

```
Class empClass = Class.forName("com.xyz.hr.Employee");
Collection clnEmployee = pm.getExtent (empClass, false);
String filter = "salary > 30000.00";
Query q = pm.newQuery (empClass, clnEmployee, filter);
Collection emps = (Collection) q.execute ();
```

パラメータによる制約条件を使用した単一の表の選択

次の照会は、Employee インスタンスのうち、フィールド値がパラメータを使用した判定条件を満たしているもの(この場合は給与がパラメータとして渡された値を超えているもの) をすべて選択します。

```
同等の ANSI SQL 文: SELECT * FROM EMPLOYEE WHERE SALARY > ?
```

パラメータの宣言は、1 つ以上のパラメータの型宣言をコンマ (,) で区切って並べた String になります。その構文は、Java メソッドのシグニチャの構文と同じです。

候補インスタンスの salary フィールド値が null の場合は、比較のための値のアンラップを行えないため、そのインスタンスは拒絶されます。

```
Class empClass = Class.forName("com.xyz.hr.Employee");
Collection clnEmployee = pm.getExtent (empClass, false);
String filter = "salary > sal";
String param = "Float sal";
Query q = pm.newQuery (empClass, clnEmployee, filter);
q.declareParameters(param);
Collection emps = (Collection) q.execute (new Float (30000.));
```

順序付けを使用した単一の表の選択

次の照会は、オブジェクトのリストを、オブジェクト内の 1 つ以上のフィールドの値を基準にして並べ替えた結果を返します。

順序付けの文は、1 つ以上の順序付け宣言をコンマ (,) で区切って並べた String になります。それぞれの順序付け宣言は、ターゲットクラスのフィールド名に、ascending (昇順) または descending (降順) を付け加えたものになります。

```
同等の ANSI SQL 文: SELECT * FROM EMPLOYEE ORDER BY LASTNAME
ASCENDING, FIRSTNAME ASCENDING

Class empClass = Class.forName("com.xyz.hr.Employee");
Collection clnEmployee = pm.getExtent (empClass, false);
String filter = "true";
Query q = pm.newQuery (empClass, clnEmployee, filter);
query.setOrdering("lastname ascending, firstname ascending")
Collection emps = q.execute ();
```

関係の「1」の側との結合

次の照会は、オブジェクトの中から、参照先のオブジェクトが判定条件を満たしているもの (この場合は、Employee インスタンスのうち、関係の相手側の Department インスタンスの name フィールド値が、パラメータとして渡された値と一致しているもの) を選択します。

```
同等の ANSI SQL 文: SELECT EMPLOYEE.* FROM EMPLOYEE, DEPARTMENT
WHERE DEPARTMENT.DEPTNAME = ? AND EMPLOYEE.DEPTID =
DEPARTMENT.DEPTID
```

候補インスタンスの dept フィールド値が null の場合は、比較のための値のナビゲートを行えないため、そのインスタンスは拒絶されます。

```
Class empClass = Class.forName("com.xyz.hr.Employee");
Collection clnEmployee = pm.getExtent (empClass, false);
String filter = "dept.name == name";
String param = "String Engineering";
Query q = pm.newQuery (empClass, clnEmployee, filter);
q.declareParameters ("String name");
Collection emps = (Collection) q.execute ("Engineering");
```

関係の「多」の側との結合

次の照会は、オブジェクトの中から、参照先のコレクションに判定条件を満たしているオブジェクトが含まれているもの (この例では、Department インスタンスのうち、関係の相手側の Employee インスタンスのコレクションに、パラメータとして渡された値より給与が高い Employee インスタンスが 1 つ以上含まれているもの) を選択します。

```
同等の ANSI SQL 文: SELECT DEPARTMENT.* FROM DEPARTMENT, EMPLOYEE
WHERE EMPLOYEE.SALARY > 30000 AND DEPARTMENT.DEPTID =
EMPLOYEE.DEPTID

Class depClass = Class.forName("com.sun.xyz.Department");
Collection clnDepartment = pm.getExtent (depClass, false);
String vars = "Employee emp";
String filter = "emps.contains (emp) & emp.salary > sal";
String param = "float sal";
Query q = pm.newQuery (depClass, clnDepartment, filter);
q.declareParameters(param);
q.declareVariables (vars);
Collection deps = (Collection) q.execute (new Float (30000.));
```

主キーと外部キーの重複

透過的な持続性は、主キーと外部キーの重複に対応していますが、いくつかの注意事項があります。次のスキーマを例に、このことを考えてみましょう。

```
CREATE TABLE Order
(
        orderNumber INT PRIMARY KEY,
        customerName VARCHAR2(32) NULL,
        requestedDate DATE NULL
)
CREATE TABLE LineItem
(
        lineItemNumber INT NOT NULL,
        orderNumber INT NOT NULL,
        price FLOAT NOT NULL,
        description VARCHAR2(100) NULL,
        PRIMARY KEY (lineItemNumber, orderNumber),
        FOREIGN KEY (orderNumber) REFERENCES Order(orderNumber)
)
```

これに対応する持続可能クラスは次のようになります。

```
public class Order
{
        int ordernumber;
        String customername;
        Date requesteddate;
        HashSet lineitems;
}
public class Lineitem
{
        int lineitemnumber;
        int ordernumber;
        float price;
        String description;
        Order order;
}
```

透過的な持続性は主キーの変更に対応していないため、Order と Lineitem との関係は変更できません。たとえば、Order に Lineitem を追加するには、Lineitem.ordernumber を変更する必要がありますが、このフィールドは主キーの一部です。同様に、Order から Lineitem を削除する場合も、Lineitem.ordernumber をゼロに設定する必要がありますが、そうするとデータベースの制約に違反してしまいます。どちらの場合も、透過的な持続性では Oid は更新されず、キャッシュ中のインスタンスの再ハッシュも行われません。

この問題に対処するには、次の手法を使用します。

Order/Lineitem 関係の作成

Order/Lineitem 関係を作成するには、次のようにします。

```
tx.begin();
Order o = new Order();
o.setOrdernumber(1);
o.setCustomername("peter");
HashSet items = new HashSet();
o.setLineitems(ltems);
Lineitem lt = new Lineitem();
lt.setLineitemnumber(1);
lt.setOrdernumber(1);
```

ordernumber を、既存の Order の ordernumber に明示的に設定する必要があります。この Order は、すでにデータベースに含まれている持続インスタンスでも、まだデータベースに収容していない持続インスタンスでもかまいません。

```
items.add(lt);
```

注 - いったん Lineitem.ordernumber を 1 に設定すると、その Lineitem は Order 1 の lineitems コレクションにしか追加できません。

```
pm.makePersistent(o);
tx.commit();
```

Order/Lineitem 関係の削除

次のコードを使用すると、Order/Lineitem 関係を適切に削除することができます。

```
tx.begin();
Order o = ....              // Order を取得
Lineitem lt = .....         // 削除したいLineitem を取得
```

同じトランザクションの中で明示的に削除する場合に限り、Order から Lineitem を削除することができます。次の 2 つの行のどちらか一方を使用することができます。

```
pm.deletePersistent(lt);
o.getLineitems().remove(lt);
```

次のようにして、Order からすべての Lineitems を削除することもできます。この場合も、同じトランザクションの中で明示的に削除することが条件になります。

```
pm.deletePersistent(o.getLineitems());
o.getLineitems().clear();

tx.commit();
```

制約

次の制約があります。

- ある Order の Lineitem を、別の Order に移すことはできません。この処理を行うには、Order から該当する Lineitem を取り除き、新しい Lineitem を作成して、別の Order に追加する必要があります。
- Lineitem.setOrder() は使用できません。例:

```
Lineitem lt = o.getLineitems() .get(0);
```

この行を実行すると、JDOUnsupportedOptionException 例外がスローされます。

```
lt.setOrder(null);
```

また、次の 2 行のコードを実行すると、コミット時に JDOUserException がスローされます。

```
o.getLineitems.add(lt);
lt.setOrder(o);
```

フェッチグループ

フェッチグループとは、まとめて取り出す持続フィールドの集まりです。アプリケーションが、フェッチグループに属しているいずれかのフィールドの値を要求すると、そのグループに属しているすべてのフィールドの値がまとめて取り出されます。この機能を使用すると、従業員の住所を構成しているフィールド群など、同時に使用することの多い値を効率的に転送することができます。クラスの開発者は、データベースレコードのフィールドを分析し、クラス定義にフェッチグループを含めるかどうかを決定することができます。

「レベル」、「独立」、「デフォルト」、または 「なし」 を指定できます。設定には、階層型と独立型の 2 種類があります。

階層グループには「デフォルト」設定と「レベル」設定があり、相互に階層を形成しています。あるフィールドを「デフォルト」に設定することは、「デフォルト」に設定されている他のすべてのフィールドとともに、そのフィールドが取り出されることを意味します。レベル 1 グループのフィールドが取り出されると、レベル 1 グループとデフォルトグループのすべてのフィールドが同様に取り出されます。

デフォルトでは、透過的な持続性は関係フィールド以外の持続フィールドすべてを「デフォルト」フェッチグループに含めます。フェッチグループプロパティが無効になっている場合、フィールドは持続的でないか、マップされないか、またはキーフィールドであり、常に取り出されます。関係フィールドの設定値は、「なし」でなければなりません。

インスタンス状態のチェック

先ほどの照会、更新といった基本操作についての説明では、持続インスタンスの状態について紹介し、持続性マネージャがこれらの状態をどのように設定し、トランザクションの終了時に、その状態に応じてどのような処理を行うかについて説明しました。必要であれば、アプリケーションからこれらの状態を参照・設定することができます。

アプリケーションからインスタンスの状態をチェックするには、`JDOHelper` クラスを使用するとよいでしょう。このクラスに用意された static メソッドを使用すると、`PersistenceCapable` を実装したインスタンスに現在の状態を報告させることができます。インスタンスが `PersistenceCapable` を実装していない場合は、これらのメソッドは一時インスタンスから返される値と同じ値を返します。

たとえば、次のようなメソッドがあります。

```
isDirty()
makeDirty()
```

透過的な持続性の識別性

Java では、2 つのインスタンスが同一のインスタンスであるか、同一のデータを表しているものかを確認するため、次の 2 つの概念を定義しています。

- Java オブジェクト等価性は、完全に JVM によって管理されます。2 つのインスタンスが同じであるのは、これらのインスタンスが JVM 内の同じ記憶場所に存在する場合であり、この場合に限定されます。
- Java オブジェクトの等価性は、クラスによって決まります。整数の場合には同じ値、ビット配列の場合には等価ビットというように、同じ値を表している場合、これらのインスタンスは同等です。

Java オブジェクトの識別情報と等価性との間のやりとりは、透過的な持続性開発者にとっては重要な要素です。Java オブジェクトの等価性はアプリケーションごとに固有で、アプリケーションでの等価性の実装は、透過的な持続性では変更されません。また、特定のデータベースオブジェクトの持続状態を表現したインスタンスは、持続性マネージャごとに 1 つしかありません。したがって、透過的な持続性は、JVM オブジェクトの識別情報とアプリケーションの等価性とは別にオブジェクト識別情報を定義します。

アプリケーションでは、持続可能クラスの等価性を、JVM での識別情報を使用する、デフォルトの等価性とは別に実装する必要があります。これは、持続インスタンスの JVM での識別情報は、ごく一部の場合を除いて、同じ持続性マネージャの中や、同じ時間や空間の中でしか保証されないからです。

持続インスタンスをデータベースに保存し、それらのインスタンスを照会演算子 == を使用した照会で抽出したり、識別情報が適用される持続コレクション (Set、Map) から参照する場合は、主キーや Oid をキーとして使用し、等価性の実装を Transparent Persistence での等価性の実装と完全に一致させる必要があります。このことは義務付けられているわけではありませんが、正しく実装しないと、コレクションの意味合いが違ってしまう可能性があります。

本書では、Java オブジェクトの識別情報との混同を避けるため、透過的な持続性での識別情報の概念を、「透過的な持続性での ID」と呼びます。透過的な持続性での ID はデータベース用に使用されます。データベースでは、インスタンスの中の値によってデータベース中のオブジェクトの識別情報が決まります。透過的な持続性での ID は、アプリケーションで管理し、データベースによって適用されます。

持続性マネージャは、開発者に代わってインスタンスの ID を管理しますが、== 演算子などを使用して持続インスタンスを比較する際には、比較の対象になるのは 透過的な持続性での Oid になります。

Oid クラス

Oid クラス (オブジェクト ID) は、持続可能クラスごとに作成されます。Oid クラスは持続可能クラスのプロパティの一つで、マッピング時に作成する必要があります。

それぞれの持続性マネージャは、透過的な持続性インスタンスのキャッシュを管理し、特定のデータベースオブジェクトをカプセル化した透過的な持続性インスタンスが、持続性マネージャごとに 1 つしか存在しないようにする必要があります。

そのために、それぞれの透過的な持続性クラスに Oid クラスを関連付けて、そのクラスの中に、透過的な持続性インスタンスを一意に識別する値を収容するフィールドを組み込みます。その結果として、透過的な持続性クラスのインスタンスごとに、Oid クラスのインスタンスが 1 つずつ対応付けられ、そのフィールドに、該当する透過的な持続性インスタンスの ID が収容されます。これで、実行環境が Oid を比較し、透過的な持続性インスタンスの識別情報と等価性を管理できるようになります。

多くのデータベースでは、エンティティの ID はデータの中の値によって決まります。その代表的な例がリレーショナルデータベースシステムです。なぜなら、それぞれの行やオブジェクトに、その行やオブジェクトを識別するキー値が含まれているからです。このようなデータベースでは、Java 生成機能によって生成される Oid クラスは「主キークラス」になり、そのフィールドに主キー値が収容されます。

Oid クラスには次の 2 種類があります。

- 接尾辞 `Oid` が付いた静的な入れ子クラス (デフォルト)
- 接尾辞 `Key` が付いた単独のクラス

これらの接尾辞は、両方とも大文字 / 小文字が区別されます。

持続可能クラスの名前は `mypackage.Employee` などとなり、有効な Oid クラスの名前は `mypackage.Employee.Oid` または `mypackage.EmployeeKey` などとなります。

注 - 「プロパティ」ウィンドウには、持続可能クラスのフィールドごとに Boolean 型の「キーフィールド」オプションがあります。しかし、Oid クラスでのキーフィールドは、持続可能クラスのフィールドのうち、Oid クラスの対応するフィールドと名前と型が同一の (public な) フィールドです。

混乱を避けるため、持続可能クラスの「キーフィールド」オプションの設定を、Oid クラスの構造に合わせる必要があります。そのためには次のようにします。

- 持続可能クラスで主キーとして設定したフィールドを Oid クラスで宣言します。
- Oid クラスのフィールドを主キーとして設定し、そのフィールドが持続可能クラスに存在していることを確認します。
- 持続可能クラスと Oid クラスの同名のフィールドを同じ型にします。

一意性

持続インスタンスの透過的な持続性での ID は、システムによって管理されます。このように、ID (データベースキーや主キー) によって同一性が管理されるため、次のどの手段で持続インスタンスを取得したとしても、特定のデータベースオブジェクトを表現した持続インスタンスは、持続性マネージャインスタンスごとに 1 つしかありません。

- `PersistenceManager.getObjectById(Object oid);`
- `Query` インスタンスを使用した照会
- 持続インスタンスからのナビゲート
- `PersistenceManager.makePersistent(Object pc);`
- `PersistenceManager.getTransactionalInstance(Object pc);`

主キーは特定のフィールドの集まりです。主キーを構成するフィールドは持続可能クラスのプロパティで、そのクラスを拡張し、実行時に使用できるようにした後は変更できません。一時インスタンスを持続インスタンスにすると、システムは主キーを構成しているフィールドの値から、透過的な持続性での ID を作成します。

マッピング

マッピングを行うと持続可能クラスごとに static な (かつ public の) 入れ子クラスを生成します。このクラスを `Oid` クラスと呼びます。このクラスには、`<className>.Oid` という形式でアクセスすることができます。生成時には、個々のクラスを持続可能クラスにするかどうかを指定します。それぞれの持続可能クラスの主キーフィールドと、`<className>.Oid` のフィールドは、GUI では変更が禁止されません。持続可能クラスの主キーフィールドの名前や型と、`<className>.Oid` のフィールドの名前や型の不一致が発生しないように注意してください。

クラス Employee の Oid クラスの作成・使用例を次に示します。

```
Employee.Oid eieio = new Employee.Oid();
eieio.id = 142857;
Employee emp = (Employee) myPM.getObjectById (eieio);
String name = emp.getName();
```

持続オブジェクトモデル

Java 実行環境は、開発者が使用する各種のクラスに対応しています。通常は、アプリケーションの各クラスは密接に結び付いていて、それらのクラスのインスタンスにデータベースの内容全体が取り込まれます。

しかし、アプリケーションが一度に使用する持続インスタンスの数は、通常はそれほど多くはありません。透過的な持続性は、アプリケーションにはインスタンスの全体像にアクセスできるように見せかけ、実際にはそれらのインスタンスのごく一部のサブセットを JVM でインスタンス化すればよいようにします。

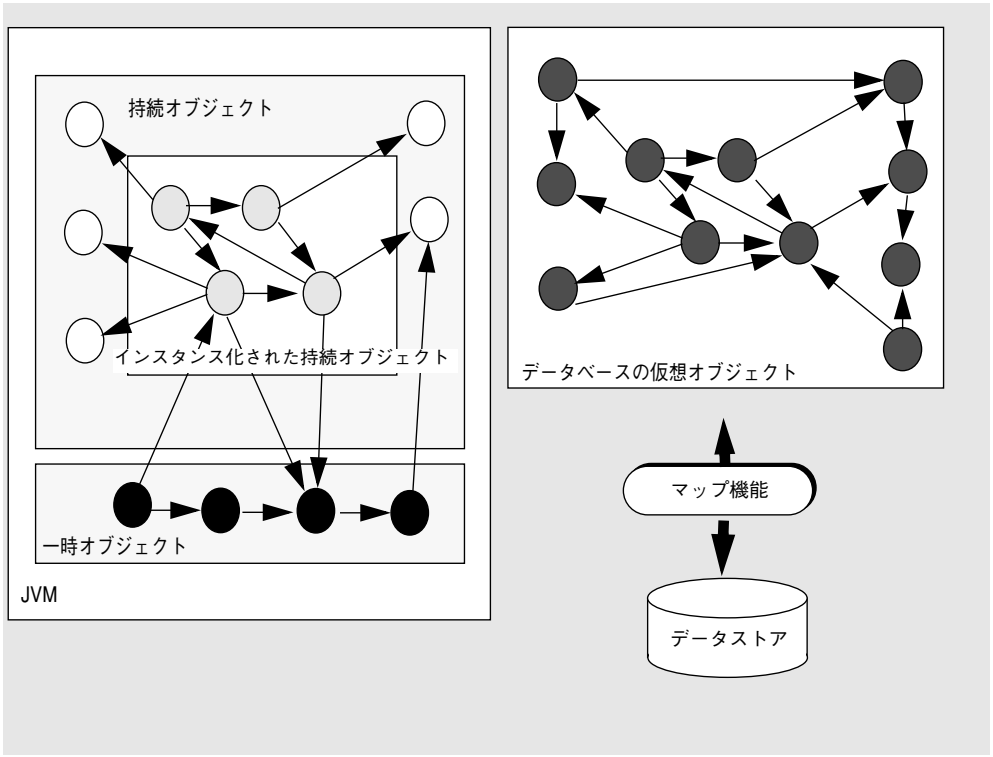

図 5-3　持続オブジェクトのインスタンス化

JVM の中に、独立した複数の作業単位があり、それらを別々に処理しなければならない場合があります。透過的な持続性では、同じデータベースオブジェクトを複数の Java インスタンスにインスタンス化することができます。ある持続オブジェクトから、参照を介して別の持続オブジェクトがアクセスされると、そのたびに該当するインスタンスが JVM にインスタンス化されます。

データベースでのオブジェクトの記憶形式と、JVM でのオブジェクトの記憶形式は異なります。そのため、透過的な持続性は実行時に使用可能なメタデータを使用し、Java インスタンスとデータベース中のオブジェクトとのマッピング (対応付け) を作成します。

持続可能クラスの非持続フィールドの型には制限はありません。これらのフィールドは、Java 言語で定義されているとおりに動作します。これに対して、持続可能クラスの持続フィールドは、クラス定義でのフィールドの型の特性に応じて制限が適用されます。

アーキテクチャ

Java では、変数 (クラスのフィールドを含みます) に型があります。変数の型は、プリミティブ型か参照型のいずれかです。参照型は、クラスまたはインタフェースです。配列はクラスとして取り扱われます。

インスタンスは特定のクラスに属します。インスタンスの所属先のクラスは、そのインスタンスが作成されたときに決定されます。インスタンスは変数に代入することができます。ただし、その変数の型に対して代入互換性があることが条件になります。

透過的な持続性オブジェクトモデルは、次の 2 種類のクラスを区別します： 持続可能なクラスと、そうでないクラス。ユーザー定義クラスは持続可能クラスになります。ただし、その状態がアクセス不能なオブジェクトやリモートオブジェクトの状態に依存しているもの (たとえば、`java.net.SocketImpl` のサブクラスとして定義したユーザー定義クラスや、ネイティブコールを使用して動作を実装したユーザー定義クラス) は、持続可能クラスにはなりません。

システム定義クラス (`java.lang`、`java.io`、`java.net` などで定義されているクラス) は、持続可能クラスにはなりません。また、次の持続フィールドも持続可能クラスに指定することはできません。

- すべての基本型 (boolean、byte、short、int、long、char、float、double)
- すべての不変オブジェクトクラス型 (第 2 クラスオブジェクトとしての Boolean、Character、Integer、Long、Float、Double、String)
- `java.util` パッケージの可変オブジェクトクラス型 (Date、ArrayList、Vector)、および `java.sql` パッケージの可変第 2 クラスオブジェクトとしての可変オブジェクトクラス型 (Date、Time、Timestamp)

持続オブジェクトと一時オブジェクト

データベースに関連付けられたクラスを「持続可能クラス」と呼びます。持続可能クラスを表現したオブジェクトは、持続オブジェクトにも、一時オブジェクトにもすることができます。このうち、持続オブジェクトはデータベースに保存されます。これに対して、一時オブジェクトは、そのオブジェクトをインスタンス化したプログラムを実行している間しか存在しません。

データベースにインスタンスを保存できるようにするには、そのインスタンスのクラスに PersistenceCapable インタフェースを実装する必要があります。透過的な持続性では、Java コードの生成時に、このインタフェースを実装するコードが自動的に追加されます。

持続可能クラスのフィールドの型

持続可能クラスのフィールドには、持続フィールド、トランザクション非持続フィールド、非トランザクション非持続フィールドの 3 種類があります。

持続フィールド

持続フィールドの型を 表 5-10 に示します。

表 5-10 持続フィールドの型

フィールドの型	説明
基本型	透過的な持続性は、基本型 boolean、byte、short、int、long、char、float、および double のフィールドに対応します。基本型の値は、所有元の第 1 クラスオブジェクトに関連付けられているデータベースに保存されます。基本型には透過的な持続性での ID はありません。
不変オブジェクトクラス	透過的な持続性は、不変オブジェクトクラス型のフィールドに対応し、これらのフィールドを第 2 クラスオブジェクトや 第 1 クラスオブジェクトとして使用可能にすることができます。 package java.lang: Boolean、Character、Integer、Long、Float、Double、および String 透過的な持続性のアプリケーションは、これらのフィールドが第 2 クラスオブジェクトあるいは第 1 クラスオブジェクトのどちらとして処理されていても使用可能にできます。

表 5-10 持続フィールドの型 (続き)

フィールドの型	説明

可変オブジェクトクラス	透過的な持続性は、可変オブジェクトクラス型のフィールドに対応し、これらのフィールドを第 2 クラスオブジェクトや 第 1 クラスオブジェクトとして使用可能にすることができます。
	`package java.util`: `Date` および `HashSet`
	`package java.sql`: `Date`、`Time`、および `Timestamp`
	これらのフィールドは第 2 クラスオブジェクトとして取り扱われる場合もあるため、これらの可変オブジェクトクラスの持続インスタンスでの動作は、一時インスタンスでの動作と同じではありません。

`PersistenceCapable` クラス	透過的な持続性は、`PersistenceCapable` クラス型のフィールドの第 1 クラスオブジェクトとしての使用に対応します。
Collection インタフェース	透過的な持続性は、Collection インタフェース型のフィールドに対応します。 `package java.util`: `Collection` および `Set`

持続フィールドと非持続フィールド

持続可能クラスには、持続フィールドと非持続フィールドの両方を組み込むことができます。

- 持続フィールドは、持続データを表現するのに使用します。このフィールドは、透過的な持続性実行環境がクラスの使用者に代わって管理します。透過的な持続性実行環境は、現在のトランザクションの状態や、使用されている並行性制御方式に従って、持続フィールドの値とデータベースとの同期化、オブジェクト値のデータベースへのフラッシュといった処理を自動的に実行します。
- 非持続フィールドは、アプリケーションロジックで管理します。このフィールドは、透過的な持続性による処理の対象になりません。アプリケーションでは、このフィールドを使用して、作業用の値を一時保存したり、トランザクションで使用する値のうち、データベースに保存する必要のないものを記録したりすることができます。

JDOインタフェース

com.sun.forte4j.persistence パッケージには、次の JDO インタフェースが含まれています。

- PersistenceManagerFactory – 透過的な持続性クラスの使用者 (アプリケーション開発者) が、持続性マネージャを作成できるようにします。

  開発者は、持続性マネージャのコンストラクタを使用できません。持続性マネージャを作成するには、持続マネージャファクトリを使用します。透過的な持続性 API には、このインタフェースを実装したクラスが含まれています。アプリケーションでは、まず持続マネージャファクトリをインスタンス化し、そのプロパティを設定し、その後で持続性マネージャを作成します。持続マネージャファクトリのプロパティ設定値は、その持続マネージャファクトリから作成される持続性マネージャのデフォルト設定値になります。接続プールを使用する場合も、持続マネージャファクトリと接続ファクトリを使用して、そのためのプロパティを設定することができます。

- PersistenceManager – 持続可能クラスをトランザクションモードで管理および操作できるようにします (その結果として、データベースで選択、挿入、更新、削除処理が行われます)。

  持続性マネージャは、持続データのデータベースからの再取得を含め、データベースとのやり取りをすべて管理します。そのため、アプリケーションで行う必要があるのは、個々のトランザクションの開始と終了を指示することだけです。

  持続性マネージャは、アプリケーションが作成した持続可能クラスのインスタンスや、アプリケーションが作成した照会に応じて持続性マネージャ自身が取得した持続可能クラスのインスタンスを管理します。それぞれの持続性マネージャは、1 つのトランザクションだけを処理することができます。すなわち、それぞれの持続性マネージャは、通常は単一のクライアントセッションで作成または取得された持続インスタンスを管理します。そのため、通常はクライアントセッションごとに、持続性マネージャが 1 つずつ必要です。それぞれの持続性マネージャからは、1 つのデータベースにしか接続できません (ただし、そのデータベースの複数の表を使用することができます)。したがって、クライアントセッションによっては、複数の持続マネージャファクトリから複数の持続性マネージャを取得する必要があります。

- Transaction – 持続可能クラスの使用者がトランザクションを開始および終了 (コミットまたはロールバック) できるようにします。

持続性マネージャから、このインタフェースを実装したオブジェクト (トランザクションオブジェクト) を取得します。これで、その持続性マネージャが管理する持続インスタンスを、トランザクションで使用できるようになります。複数データベーストランザクションを実行する場合は、複数の持続性マネージャを使用する必要があります。

- `Query` – 持続可能クラスの使用者が照会を作成できるようにします。

  持続性マネージャから、このインタフェースを実装したオブジェクトを取得します。これで、Query メソッドを使用して、JDO 照会の構文で照会を作成できるようになります。作成した照会を実行するには、`execute()` メソッドを呼び出します。照会の結果は、透過的な持続性クラスのインスタンスのコレクションの形式でアプリケーションに返されます。

- JDO 例外 – JDO では、`JDOException` と、この例外から派生した多数の例外が定義されています。これらはチェックされない実行時例外ですアプリケーション開発者は、アプリケーションからスローする JDO 例外をキャッチするコードを作成する必要があります。

これらのインタフェースのコードは、透過的な持続性の `.jar` ファイルに含まれています。持続マネージャファクトリはクラスとして実装され、開発者が直接インスタンス化することができます。それ以外のオブジェクトを取得する場合は、該当する Factory のメソッドを呼び出します。

持続可能クラスは、定義上 `PersistenceCapable` インタフェースを実装したクラスです。透過的な持続性クラスの使用者 (アプリケーション開発者) は、このインタフェースに用意されたメソッドを使用し、透過的な持続性インスタンスの状態を参照および再設定することができます。

透過的な持続性クラスには、このインタフェースを実装する必要がありますが、クラスの開発者がそのためのコードを作成する必要はありません。これらのコードは、クラスを拡張したときに、透過的な持続性によって自動的に生成されます。これで、拡張後のクラスが透過的な持続性実行環境と連携できるようになります。クラスの開発者と、クラスを使用するアプリケーション開発者のどちらも、生成されたコードの内容を知る必要はありません。

透過的な持続性クラスは移植可能で、別の JDO 環境に移すことができます。新しい環境で再び拡張作業を行うと、これらのクラスが正しく機能するようになります。

JDO 例外

規則違反についての例外を 表 5-11 に示します。

表 5-11 JDO ユーザー例外

例外	説明
JDOException (「オブジェクトが PersistenceCapableではありません」)	PersistenceCapable を実装していないクラスのオブジェクトを持続化することはできません。
JDOUserException (「同じ主キーを使用するインスタンスが既にこの PM キャッシュに存在します」)	makePersistent を適用しようとした Java オブジェクトと、データベース ID が同一の Java オブジェクトがすでに存在しています。
JDOFatalUserException (「PM が閉じています」)	使用しようとした持続性マネージャがすでに閉じられています。
JDOFatalInternalException	マッピング時または実行時に予期しないエラーが発生しました。
JDOUnsupportedOptionException	対応していないオプション (setIgnoreCache(false) など) は使用できません。
JDODataStoreException	データベースで競合が発生しているか、整合性の制約に違反しました。
JDOQueryException(「候補クラスの指定がありません」)	候補コレクションが指定されていません。Query インタフェースの setClass メソッドを確認してください。
JDOQueryException(「候補コレクションの指定がありません」)	候補コレクションが指定されていません。Query インタフェースの setCandidates メソッドを確認してください。
JDOQueryException(「候補コレクションが候補クラス <クラス> と一致しません」)	候補コレクションが候補クラスの範囲コレクションではありません。
JDOQueryException(「引数の数が間違っています」)	declareParameters で定義されたものより多くの実引数が渡されました。

表 5-11 JDO ユーザー例外 (続き)

例外	説明
JDOQueryException(「バインドされていない照会パラメータ "パラメータ"」)	Query インタフェースの execute メソッドに、照会パラメータの値が与えられていません。
JDOQueryException(「実パラメータの型に互換性がありません」) " java.lang.String" から " long" に変換できません。;	実引数の型と、パラメータ宣言で指定された型の間に互換性がありません。
JDOQueryException(「<メソッド> 列(<nr>): <問題の説明>」)	このフォームは、Query 定義に関する問題を示しています。<メソッド> は、Query メソッド (setFilter、declareParameters、setOrdering など) の 1 つです。<nr> は、エラーの発生した列の番号です。<問題の説明> は、エラーの説明であり、構文エラーや '<' の間違った引数などを示します。たとえば、this.michael == 0 というフィルタ式を指定した場合に、クラス Employee でフィールド michael が定義されていないと、その結果が JDOQueryException("setFilter column(6): Field 'michael' not defined for class 'com.xyz.hr.Employee'.") になります。

持続性認識アプリケーションのデバッグ

Persistence デバッガを使用すると、持続可能クラスを JAR ファイルとしてパッケージすることなく、持続認識アプリケーションをデバッグできます。Persistence 実行と同様に、Persistence デバッガは特殊なクラスローダーを使用して、透過的な持続性クラスファイルのロード時に、それらのクラスの拡張機能を適用します。

▼ アプリケーションをデバッグする

1. JDBC ドライバがマウントされているか CLASSPATH に含まれていることを確認します。
2. IDE デバッグ環境でアプリケーションを開きます。
3. 「プロジェクト」>「設定」を選択します。
4. 「デバッガの種類」を選択し、「Persistence デバッガ」を選択します。
5. ほかの Forte for Java デバッガと同様に Persistence デバッガを使用してデバックします。

   Forte for Java デバッグ環境の詳細については、IDE オンラインヘルプの 「プログラミングのデバック」を参照してください。

第6章

透過的な持続性と Enterprise Java Beans の併用

この章では、Enterprise Java Beans コンポーネントと透過的な持続性を併用する方法について説明するほか、J2EE™ Reference Implementation (J2EE RI) および iPlanet™ Application Server (iAS) アプリケーションとともに持続可能クラスを使用するためのサンプルコードを紹介します。

注 - Forte for Java は、Forte for Java IDE 外部で開発された透過的な持続性を使用する Enterprise Java Beans に対応していません。

Enterprise Beans 環境における 透過的な持続性の機能

透過的な持続性では、リレーショナルデータベースに格納されている持続データのオブジェクトを表示できます。持続インスタンスは、セッション bean またはエンティティ bean を伴うヘルパーオブジェクトとして、Enterprise Beans 環境で使用できます。透過的な持続性を使用して Enterprise Bean 環境のパフォーマンスを向上すると、データベースに頻繁にアクセスする必要がなくなります。個別の `get` および `set` メソッドを作成する代わりに、複数のフィールドを表示および更新する値のオブジェクトとして、シリアライズされた持続可能クラスを使用できます。

Enterprise Beans 環境で透過的な持続性を使用するには、その他のあらゆる持続アプリケーションと同様に、まず持続可能クラスを作成します。作成されたこれらのクラスは、Enterprise Beans とともに使用できます。

Enterprise Beans とともに持続可能クラスを使用する環境は、2 層アプリケーションの場合と多少異なります。これらの違いは、持続性マネージャが取得される方法と、トランザクションの管理方法に関係しています。

- bean インスタンスがアクティブの間、JNDI 検索を実行することにより、持続マネージャファクトリが検出されます。
- EJB コンテナと透過的な持続性は、独自のトランザクションを管理しない持続性認識 enterprise bean のトランザクション管理と同期をとります。

透過的な持続性を使用する Enterprise Bean も、2 層アプリケーションの環境と多少異なります。主な違いは、ビジネスメソッドの実装方法、同期の処理方法、およびトランザクションの管理方法です。

- bean のビジネスメソッドを実装する場合、必要に応じて bean にアクセスして状態を変更する持続可能クラスのインスタンスへの参照が使用されます。
- トランザクションの同期は、持続性マネージャによって処理されます。このとき、コンテナは、bean のライフサイクルの適当な時点でトランザクション実行のコールバックを作成します。
- 各ビジネスメソッドは、持続マネージャファクトリから独自の持続性マネージャインスタンスを取得する必要があります。ビジネスメソッドが終了した時点で、持続性マネージャインスタンスは閉じなければなりません。その結果、透過的な持続性実行時環境とコンテナの間でトランザクションが同期化されます。
- bean は、トランザクション補完のために、コンテナ管理トランザクションを使用できます。この場合、コードを追加する必要はありません。独自のトランザクションを管理している bean は、ユーザートランザクション (`javax.transaction.UserTransaction` インタフェースのインスタンス) または持続性マネージャから取得する透過的な持続性トランザクションを使用できます。

セッション bean およびエンティティ bean は持続性マネージャを取得し、その持続性マネージャで定義されたインタフェースを使用する持続インスタンスで CRUD (作成、読み取り、更新、削除) オペレーションを実行できます。その行程は、2 層環境でアプリケーションが動作する場合と何ら変わりありません。

ビジネスメソッドで使用する持続インスタンスを検出するには、`com.sun.forte4j.persistence.Persistence Manager` インタフェースの `getObjectByID(Object oid)` メソッドを呼び出すか、`com.sun.forte4j.persistence.Query` インタフェースから照会を実行します。ここでは、2 層アプリケーションとの違いはありません。

Enterprise Java Beans 環境で透過的な持続性を使用する通常の手順を次に示します。

- 透過的な持続性環境で持続可能クラスを作成またはマップします。
- IDE の EJB テンプレートを使用して enterprise bean を生成します。この時、次の処理を行うコードが生成されます。
  - 動的な JNDI 検索を実行して、持続マネージャファクトリを検出します。
  - 持続マネージャファクトリに対して `getPersistenceManager()` を実行し、持続性マネージャを取得します。
  - 持続データを使用して、希望のオペレーションを実行します。
  - 持続性マネージャを閉じます。

接続プールを使用する場合、持続マネージャファクトリの外側でデータソースプロパティを構成する必要があります。

Enterprise Java Beans、J2EE RI、および iAS の詳細については、それぞれのモジュールに含まれているドキュメントを参照してください。また、Forte for Java ポータルサイトで Enterprise JavaBeans と透過的な持続性を併用する例が多数盛り込まれたホワイトペーパーも用意されています。

シリアライズへの対応

EJB メソッドのパラメータまたは戻り値の型として持続可能クラスを引き渡す場合、クラスをシリアライズ可能にする必要があります。持続可能クラスを生成すると、それらのクラスをシリアライズ可能として定義するオプションが提供されます (つまり `java.io.Serializable` インタフェースが実装されます)。

持続性マネージャをホストする仮想マシンの外側にオブジェクトを引き渡すと、持続性マネージャはオブジェクトの状態を追跡できなくなります。そのため、リモートインタフェースから受け取ったオブジェクトを enterprise bean が更新するためには、変更されたオブジェクトを受け入れて変更内容を持続インスタンスに適用する bean メソッドを用意する必要があります。または、オブジェクトの変更内容についてクライアントが判定し、別の bean メソッドが情報を更新することも可能です。

後者のケースを次に示します。ビジネスメソッドにおいて、持続可能インスタンスは、トランザクションの一部であっても関連する enterprise bean コンポーネントを保持しない別の持続可能インスタンスを参照することがあります。そのような参照をクライアントに返す場合、インスタンスのクラスはシリアライズ可能でなくてはなりません。

たとえば、エンティティ bean `OrderBean` はヘルパーインスタンスとして持続可能クラス `Order` を使用し、`Order` は持続可能クラス `LineItem` を使用するとします。持続可能インスタンス `LineItem` の配列を返すには、`LineItem` をシリアライズ可能にし、次のメゾットと `OrderBean` 上のリモートメソッドを記述します。

```
public Collection getLineItems()
```

シリアライズ可能な持続インスタンスのコピーを作成するには、持続性マネージャを閉じる前に、JDO ヘルパー メソッド `createSerializedCopy` を呼び出します。この場合のコードの例を次に示します。

```
persistenceManager = persistenceManagerFactory.getPersistenceManager();
//照会またはナビゲーションを実行します
//Collection items = ...
Collection result = (Collection)JDOHelper.createSerializedCopy(items);
persistenceManager.close();
return result;
```

トランザクションと Enterprise Beans

トランザクション管理は、トランザクションの開始時と終了時、およびトランザクションがコミットされたかロールバックされたかをコンテナに伝えるプロセスです。enterprise bean では、トランザクション管理は、bean の種類に依存する標準的な方法で処理されます。

コンテナ管理トランザクションの補完も含めて エンティティ bean とセッション bean をプログラミングすると、アプリケーションコンポーネントはトランザクションを補完しません。bean 管理トランザクションの補完も含めて セッション bean をプログラミングすると、bean がトランザクションを補完します。

使用する bean の種類に関係なく、透過的な持続性は、コンテナのトランザクション補完意味論に調整されます。

持続性マネージャはトランザクション指向のオブジェクトです。つまり、特定のトランザクションに固有の情報を含んでいます。持続マネージャファクトリは持続性マネージャのプールを管理し、それぞれの持続性マネージャが個別のトランザクション

と関連付けられます。bean は、トランザクションに適した持続性マネージャを取得しなければなりません。そのため、持続性マネージャは、実行スレッドがトランザクションと関連付けられた時点で取得します。

各ビジネスメソッドは、持続マネージャファクトリから持続性マネージャを取得し、メソッドの終了時に持続性マネージャを閉じます。

bean 管理トランザクションをともなうステートフルなセッション bean の場合、持続性マネージャの取得時期は bean が決定します。これは、持続性マネージャが対話状態で管理されるためです。

透過的な持続性を使用する Enterprise Bean の作成

ここからは、持続可能クラスを使用する enterprise bean を作成するための一般的なプロセスを説明します。各節は、持続可能クラスが作成済みであることを前提に記述されています。

JNDI 検索の設定

Enterprise JavaBeans 環境において、リソースを使用するすべてのコンポーネントは、配備記述子に埋め込まれたリソースを識別し、実行時に JNDI 検索でそれらのリソースを動的に取得する必要があります。JDBC 接続は、コンテナによって管理され、bean コンポーネントによって検索されるリソースの一例です。透過的な持続性環境において、持続マネージャファクトリは、配備記述子として構成され、実行時に検索されるリソースです。

好ましいアプローチとしては、`java:comp/env/jdo/`*persistencemanagerfactoryname* において持続マネージャファクトリの参照を宣言します。

対応する名前が実行時に `InitialContext` に与えられると、コンテナは、適切な持続マネージャファクトリを見つけて bean に返します。

持続マネージャファクトリは、多くの bean に共有されるリソースであり、JDBC データソースと関連付けられます。enterprise bean では、同じデータソースを使用するすべての bean が同じ持続マネージャファクトリを共有すべきです。そうすることで、同一トランザクション内の異なる bean が同じ持続性マネージャを検出できます。

bean の開発には、使用される持続マネージャファクトリは名前で識別されます。bean の中には、その名前は、特定 持続マネージャファクトリに関連付けられます。実行時、名前のついた持続マネージャファクトリは、JNDI の名前検索によって検出されます。

▼ JNDI 検索を実行する

bean の中に次の変数を配置します。

```
String persistenceManagerFactoryResourceName = "java:comp/env/jdo/pmfname";
PersistenceManagerFactory persistenceManagerFactory;
```

`setSessionContext` メソッドに次のコードを追加します。

```
InitialContext initialContext = new InitialContext();
persistenceManagerFactory= (PersistenceManagerFactory)
initialContext.lookup(persistenceManagerFactoryResourceName);
```

セッション bean テンプレートを使用すると、既に追加されている JNDI 検索とともに新しいセッション bean を作成できます。その後は、JNDI 名の可変部分を実際の名前に置き換え、必要な参照を bean のプロパティシートに追加するだけです。

▼ IDE を使用して透過的な持続性認識セッション bean を作成する

1. 「ファイル」>「新規」を選択し、「EJB」>「Session bean」を選択します。

   EJB テンプレートが表示されます。

2. bean の型を選択します。「コンテナ管理によるトランザクション 」(CMT bean) または「bean 管理によるトランザクション」(BMT bean) のいずれかを選択します。

3. 「Transparent Persistence と共に使用」チェックボックスをオンにし、「次へ」をクリックします。

注 - ステートフル CMT bean を作成する場合、`SessionSynchronization` インタフェースを実装するオプションが用意されます。このオプションは使用しないでください。

「EJB コンポーネント」区画が表示されます。

4. bean が完成するまでテンプレートに従って継続します。

   図 6-1 が示すように、エクスプローラウィンドウに bean が表示されます。

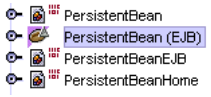

   図 6-1　持続的な Enterprise Bean

5. エクスプローラウィンドウ内の bean をマウスの右ボタンでクリックし、「プロパティ」を選択します。
6. 「J2EE RI」タブを選択します。
7. 「JNDI 名」プロパティの値を入力します。
8. 生成された持続マネージャファクトリの JNDI 名を実際の名前に置き換えます。

リソース参照の設定

Enterprise JavaBean を J2EE RI または iAS とともに設定する場合、持続マネージャファクトリをリソースファクトリ参照として識別する必要があります。この作業には、Enterprise JavaBean プロパティシートを使用します。

▼ 持続マネージャファクトリをリソース参照として設定する

1. エクスプローラウィンドウで、「Enterprise JavaBeans」ノードをマウスの右ボタンでクリックします。
2. 「参照」タブの「リソースファクトリ参照」の値をクリックし、「...」ボタンをクリックします。プロパティエディタが開きます。
3. 「追加」ボタンをクリックします。「追加　リリース参照」ウィンドウが開きます。
4. 持続マネージャファクトリの名前を追加します。
5. 「型」のドロップダウンメニューから com.sun.forte4j.persistence.PersistenceManagerFactory を選択します。
6. J2EE RI サーバーにアプリケーションを配備する予定がある場合、「J2EE RI」タブを選択し、手順 4 で指定したのと同じ名前を「JNDI 名」に入力します。
7. 「了解」をクリックして終了します。

注 - Enterprise JavaBean を iAS とともに作成する場合、上記の手順を繰り返して、プロパティシートにデータソースへの参照を追加します。

bean 管理トランザクションの使用

トランザクションの補完にあたり、コンテナが提供する `javax.transaction.UserTransaction` を使用するか、持続性マネージャが提供する `com.sun.forte4j.persistence.Transaction` を使用するか決定する必要があります。

複数のトランザクションに対し同一の持続性マネージャを使用する場合、`com.sun.forte4j.persistence.Transaction` を使用してトランザクションを補完する必要があります。トランザクションごとに持続性マネージャを取得する場合、どちらのテクニックを使うかはユーザーの判断に任されます。

トランザクションの補完に `com.sun.forte4j.persistence.Transaction` を使用するには、次のようなコードを使用します。

```
// 同一の持続性マネージャで複数のトランザクションを使用するビジネスメソッド
persistenceManager = persistenceManagerFactory.getPersistenceManager();
persistenceManager.currentTransaction().begin();
// 最初のトランザクションで持続的オペレーションを実行します
persistenceManager.currentTransaction().commit();
PersistenceManager.currentTransaction().begin();
// 第 2 番目のトランザクションで持続的オペレーションを実行します。
persistenceManager.currentTransaction().commit();
persistenceManager.close();
```

トランザクションの補完に `javax.transaction.UserTransaction` を使用する場合、持続マネージャファクトリから持続性マネージャを取得する前にトランザクションを開始し、トランザクションをコミットする前に持続性マネージャを閉じます。

トランザクションの補完に javax.transaction.UserTransaction を使用するには、次のようなコードを使用します。

```
sessionContext.getUserTransaction().begin();
persistenceManager = persistenceManagerFactory.getPersistenceManager();
// 最初のトランザクションで持続的なオペレーションを実行します。
persistenceManager.close();
sessionContext.getUserTransaction().commit();
```

コンテナ管理によるトランザクションの使用

コンテナ管理トランザクションの補完も含めてエンティティ bean とセッション bean をプログラミングすると、アプリケーションコンポーネントはトランザクションを実行しません。透過的な持続性は、コンテナのトランザクション補完意味論に調整されます。

▼ コンテナ管理によるトランザクションを使用する。

1. bean の中に次の変数を配置します。

```
PersistenceManager persistenceManager;
```

2. ビジネスメソッドごとに、持続インスタンスのオペレーション周辺で次のコードをラップします。

```
persistenceManager = persistenceManagerFactory.getPersistenceManager();
// 持続的オペレーションを実行します
persistenceManager.close();
```

コンテナ管理トランザクションの例を次に示します。

```
public java.lang.String addEmployee(long empid,
       java.lang.String lastName,
       java.lang.String firstName,
       double salary) {
    Employee emp = new Employee();
    emp.setEmpid(empid);
    emp.setLastname(lastName);
    emp.setFirstname(firstName);
    emp.setSalary(salary);

    try {
        persistenceManager =
persistenceManagerFactory.getPersistenceManager(dbuser, dbpasswd);
    persistenceManager.makePersistent(emp);
    return "Created Employee: " + emp.getEmpid();
    } catch (Exception e) {
        e.printStackTrace();
        return "Failed Create Employee: " + empid + ": " + e.toString();
    } finally {
        persistenceManager.close();
    }
}
```

J2EE 参照への透過的な持続性の統合

前述したように、透過的な持続性は J2EE RI (version 1.2.2 のみ) と併用できます。次の手順に従ってください。

1. モジュールがインストールされたディレクトリから *J2EE_HOME*/lib/system JDBC ドライバと次の 3 つのファイルをコピーします。

   - *IDE_Install*/modules/dbschema.jar
   - *IDE_Install*/modules/ext/persistence-rt.jar
   - *IDE_Install*/lib/ext/xerces.jar

   注 - *IDE_Install* は、IDE インストールディレクトリか、または Forte for Java (FFJ) ユーザーディレクトリです (アップデートセンターから FFJ モジュールをダウンロードした場合)。

   ---

2. *J2EE_HOME*/bin/userconfig.sh スクリプトを編集して、JDBC ドライバと上記の 3 つの JAR ファイルを J2EE_CLASSPATH に追加します。

3. 持続マネージャファクトリとデータソースに対する JNDI 検索を設定します。

   a. J2EE_HOME/config/default.properties を編集します。

      - サーバー起動時にロードされるドライバのリストに com.sun.forte4j.persistence.internal.EJB.j2sdkee121Helper を追加して、統合を有効にします。

```
jdbc.drivers=...:com.sun.forte4j.persistence.internal.ejb.j2sdkee121Helper
```

- データソースとして持続マネージャファクトリを登録し、jdo/*empPMF* および jdbc/*datasource* を持続マネージャファクトリおよびデータソースに対し独自に設定した JNDI 名と置き換えます。

```
jdbc20.datasources=jdo/empPMF|
xadatasource.0.jndiname=jdo/empPMF
xadatasource.0.classname =
  com.sun.forte4j.persistence.PersistenceManagerFactoryImpl
xadatasource.0.prop.ConnectionFactoryName=jdbc/datasource
xadatasource.0.prop.Optimistic=false
```

注 - 上記の各行の後ろにスペースや非表示文字がないことを確認します。J2EE サーバーはそれらの要素を認識できません。また、transaction.timeout の値が「0」に設定されていることを確認します。

---

b. J2EE RI 起動スクリプトを編集して、システムプロパティを追加します。

デフォルトでは、透過的な持続性は、各持続可能クラスを単一のクラスローダーでロードする必要があります。J2EE RI でこの標準操作が行われると、持続可能クラスを使用できるJ2EE アプリケーションが 1 つだけになり、アプリケーションの再配備が不可能になります。J2EE RI 起動スクリプトに次のシステムプロパティを追加すると、このデフォルトの動作が変更されます。

```
-Dcom.sun.forte4j.persistence.model.multipleClassLoaders
```

c. システムプロパティに次の値のいずれかを設定します。

- 「ignore」: 単一の持続可能クラス定義を使用します。この設定は、複数のアプリケーションで同じ持続可能クラスを使用し、それぞれのクラス定義が同一である場合か、または単独のアプリケーションで持続可能クラスを使用し、開発/配備/テストサイクルの期間にアプリケーションを変更する場合に適しています。
- 「reload」: 既存の持続可能クラスの定義を置き換えます。この設定は、単独のアプリケーションで持続可能クラスを使用し、開発/配備/テストサイクルの期間にアプリケーションを変更する場合に適しています。

Solraris 環境では、$J2EE_HOME/bin/j2ee の PROPS 変数を次のように設定します。

```
PROPS="-Dcom.sun.enterprise.home=$J2EE_HOME
-Djava.security.policy==$J2EE_HOME/lib/security/server.policy
-Dcom.sun.forte4j.persistence.model.multipleClassLoaders=reload"
```

Windows 環境では、j2ee.bat の %JAVACMD% コマンドを直接的に次のように変更します。

```
%JAVACMD%
-Djava.security.policy==%J2EE_HOME%\lib\security\server.policy
-Dcom.sun.enterprise.home=%J2EE_HOME%
-Dcom.sun.forte4j.persistence.model.multipleClassLoaders=reload
-classpath "%CPATH%" com.sun.enterprise.server.J2EEServer %1 %2
```

4. Enterprise JavaBean および J2EE RI のマニュアルの指示に従って、J2EE RI サーバーを起動し、Enterprise JavaBean を作成します。

   また、171 ページの「リソース参照の設定」 の説明にあるように、リソース参照として持続マネージャファクトリを設定する必要があります。

注 - トランザクションの境界設定のためにコンテナ管理トランザクションを使用する J2EE RI ビジネスメソッドでロールバックを実行する場合、ctx.setRollbackOnly() を呼び出す前に、持続インスタンスのシリアライズコピーを作成する必要があります。167 ページの「シリアライズへの対応」 を参照してください。

透過的な持続性と iPlanet Application Server の統合

iPlanet Application Server (iAS) 6.0 SP3 プラグインモジュールは、iPlanet アプリケーションと Web サーバーのプラグインモジュールに対し、アプリケーションプログラムインタフェース (API) を提供します。「透過的な持続性と Enterprise JavaBeans の併用」の個所で説明しているように、透過的な持続性は iAS と併用できます。次の手順に従ってください。

1. persistenceManagerFactory リソース参照を登録して JNDI 検索を実行できるように登録パラメータを変更します。

   a. kregedit (Solaris の IAS_Install_dir/ias/bin/kregedit または Windows の kregedit.bat) を実行します。

   b. 「SOFTWARE\iPlanet」>「Application Server」>「 6.0」>「jndiConfig」をクリックします。

   c. 「編集」>「キーを追加」を選択し、jdo を入力します。

   d. jdo をクリックします。

   e. 「編集」>「値を追加」を選択します。次の値を追加します。

      - contextClassName
      - com.netscape.server.jdo.PMFContext

   f. 「編集」>「値を追加」を再び選択します。次の値を追加します。

      - factoryClassName
      - com.netscape.server.jdo.PMFContextFactory

2. 必要な JAR ファイルを CLASSPATH に追加します。

   a. Solaris では、次の手順に従います。

      - IAS_Install_dir/ias/env/iasenv.ksh の THIRD_PARTY_JDBC_CLASSPATH の行の直前に次のコードを挿入します。

```
FFJ_IDE=IDE_Install
TP_PATH=$FFJ_IDE/modules/dbschema.jar:
$FFJ_IDE/lib/ext/xerces.jar:$FFJ_IDE/modules/ext/persistence-rt.jar:$FFJ_IDE/i
Planet/jdoias/iaspmf.jar:$FFJ_IDE/iPlanet/jdoias
```

      - CLASSPATH の直前に透過的な持続性のパス $TP_PATH を追加します。
        CLASSPATH=$TP_PATH:既存のコード

   b. Windows 環境では、CLASSPATH を編集する必要があります。

- 「SOFTWARE\iPlanet」>「 Application Server」 >「6.0」> 「Java」 f c > 「ClassPass」をクリックします。ClassPass のパスの直前に次の行を追加します。

```
IDE_Install/modules/dbschema.jar:IDE_Install/lib/ext/xerces.jar:IDE_Install/mo
dules/ext/persistence-rt.jar:IDE_Install/iPlanet/jdoias/iaspmf.jar:
IDE_Install/iPlanet/jdoias
```

3. iPlanet Application Server を再起動します。

   マニュアルに記載されている手順に従って、iPlanet プラグインを使用可能にします。

4. 持続マネージャファクトリを追加および登録します。

   a. エクスプローラーの「実行時」タブの「サーバーレジストリ」ノードを展開します。「JDO(TP) 持続マネージャファクトリ」をクリックし、「持続マネージャファクトリの追加」を選択した後、次のようにプロパティの値を入力します。

      接続ファクトリ：jdbc/ データソース名
      (例えば jdbc/PointBase)
      持続マネージャファクトリ名：empPMF
      (その他のプロパティは任意に設定)

   b. 作成された持続マネージャファクトリ(empPMF) をマウスの右ボタンでクリックし、「登録」を選択します。「登録のためのサーバー選択」ウィンドウで自分のサーバーを選択します。「登録」ボタンをクリックします。

   c. Enterprise JavaBean プロパティシートを使用して、持続マネージャファクトリをリソースファクトリに設定します。

      詳細については、171 ページの「リソース参照の設定」 を参照してください。また、持続マネージャファクトリを使用するすべての Enterprise JavaBeans のデータソース参照を設定します。

5. プラグインマニュアルに記載された手順に従って、自分の bean を iAS に配備できるようにリソース参照を修正します。

付録 A

システム要件

透過的な持続性は、DB2 Universal Database、Oracle 8*i*、PointBase、および Microsoft SQL Server 用の持続可能クラスの開発と使用に対応しています。

透過的な持続性を使用するには、Forte for Java で 透過的な持続性モジュールを実行することに加えて、Forte for Java のインストールディレクトリの `lib/ext` サブディレクトリに、次のいずれかの JDBC ドライバをインストールする必要があります。

- WebLogic for SQL server 2000 driver
- IDE に PointBase データベースと共にバンドルされている PointBase Embedded 3.5 driver
- ORACLE 8*i* 8.1.6 Thin
- DB2 Universal Database, Version 7.1

注 - 透過的な持続性では、照会文の構文解析用に ANTLR 2.7.0 が必要です。ANTLR 2.7.0 は自動的に組み込まれ、有効になりますが、実行時の JVM に他のバージョンの ANTLR が含まれていると、競合が発生してしまいます。透過的な持続性を実行する前に、他のバージョンの ANTLR を必ず無効にしてください。

`CLASSPATH` 変数に、次のソフトウェアのパスを挿入する必要があります。

- 使用する JDBC ドライバ
- 透過的な持続性実行時パッケージ、`persistence-rt.jar`
- Forte for Java インストールの `modules` ディレクトリの `dbschema.jar`。このファイルへのパスは、<*FFJ install root*>`/modules/dbschema.jar` になります。
- XML SAXParser。このソフトウェアへのパスは、<*FFJ install root*>`/lib/ext/xerces.jar` となります。

persistence-rt.jar および dbschema.jar ファイルの位置は、透過的な持続性および DBSchema モジュールのインストール方法によって異なります。

- IDE と同時にモジュールをインストールする場合、ファイルは、<*install root*>/modules/ext に配置されます。
- マルチユーザーモード (デフォルト) で動作しているアップデートセンターからモジュールをインストールする場合、ファイルは、<*ffjuser*>/modules/ext に配置されます。
- シングルユーザーモードで動作しているアップデートセンターからモジュールをインストールする場合、ファイルは、<*install root*>/modules/ext に配置されます。

注 - 透過的な持続性の実行中に CLASSPATH 変数を変更した場合は、Forte for Java を再起動する必要がありません。

付録B

透過的な持続性の JSP タグ

透過的な持続性は、JSP タグの `PersistenceManager` と `jdoQuery` をサポートしています。JSP タグの一般情報については、Forte for Javaプログラミングシリーズの『Web コンポーネントのプログラミング』 を参照してください。

`PersistenceManager` タグ

`PersistenceManager` タグは、`jdoQuery` タグが `database.jdbc` 接続を介してオブジェクトを取り出すのに使用する `PersistenceManager` を作成します。持続性マネージャの保存範囲は、application、session、request、page の 4 種類の中から選択することができます。デフォルトでは、application です。

`PersistenceManager` の属性:

- `id` (必須)

  `PersistenceManager` 情報が格納される ID です。JDO Query タグは ID を使用してオブジェクトを取り出します。この属性は静的に設定することも、JSP の式で設定することもできます。

- スコープ

  `PersistenceManager` の保存範囲です。指定する値は、application、session、request、または page です。この属性は静的に設定することも、JSP の式で設定することもできます。

- `connection` (必須)

接続情報を取り出すために使われる接続 ID を指定します。この属性は静的に設定することも、JSP の式で設定することもできます。

- connectionScope

  接続 ID の検索対象範囲。指定する値は、application、session、request、または page です。この属性を指定しなかった場合は、page、request、session、application の順に、すべての範囲が検索の対象になります。この属性は静的に設定することも、JSP の式で設定することもできます。

PersistenceManager タグの例:

```
<%@taglib uri="/WEB-INF/lib/dbtags.jar" prefix="jdbc" %>
<%@taglib uri="/WEB-INF/lib/tptags.jar" prefix="jdo" %>
<jdbc:connection id="conn"
 driver="weblogic.jdbc.mssqlserver4.Driver"
 url="jdbc:weblogic:mssqlserver4:marina@bete:1433"
 user="mv" password="mv" />
<jdo:persistenceManager id="empPM" connection="conn" />
```

jdoQuery タグ

データベースを照会して検索結果を獲得します。照会の結果を反復子タグに渡すと、その内容を表示することができます。

jdoQuery タグは、標準 SQL 文、Insert、Update、Delete、および Select をサポートします。SQL 文は属性としてではなく、本体で指定します。そのため、照会をどのように作成するかは JSP のスクリプトで制御することができます。

jdoQuery の属性:

- ID (必須)

  照会インスタンスの ID です。このインスタンスが queryscpoe の範囲内に存在している場合は、照会の本体は実行されません。queryid は、ResultsId と異なるので注意してください。ResultsId は、結果が格納される ID です。

- className (必須)

データベースから抽出するオブジェクトの正式なクラス名です。`package.subpackage.ClassName` の形式で指定します。

- フィルタ

  データベースから抽出したオブジェクトを絞り込むのに使用するフィルタです。`emp.salary < 10000` のように指定します。

- imports

  照会を作成するときに使用するクラス名と変数を指定した import 文字列です。

- 変数

  データベースからオブジェクトを取り出すための照会を作成するときに使用する変数です。

- `persistenceManager` (必須)

  照会の作成と実行に使用する `PersistenceManager` の ID です。

- `persistenceManagerScope`

  `PersistenceManager` の ID の検索範囲です。application、session、request、page の 4 種類の中から選択することができます。この値を設定しないと、次の順序でシステムによりスコープ全体が検索されます： page、request、session、application。この属性は静的に設定することも、JSP の式で設定することもできます。

- `resultsId` (必須)

  照会で得られた結果は、この属性で指定された値によって格納されます。この属性は静的に設定することも、JSP の式で設定することもできます。

- `resultsScope`

  結果データの格納先範囲。application、session、request、page の 4 種類の中から選択することができます。

jdoQuery タグの例:

```
<%@taglib uri="/WEB-INF/lib/dbtags.jar" prefix="jdbc" %>
<%@taglib uri="/WEB-INF/lib/tptags.jar" prefix="jdo" %>
<jdbc:connection id="conn"
 driver="weblogic.jdbc.mssqlserver4.Driver"
 url="jdbc:weblogic:mssqlserver4:marina@bete:1433"
 user="mv" password="mv" />
<jdo:persistenceManager id="empPM" connection="conn" />
<jdo:jdoQuery id="employeeQuery"
 persistenceManager="empPM"
 className="empdept.post.Employee"
 resultsid="employeeDS" resultsScope="session" />
<% printJDOQueryResults(pageContext,out,"employeeDS"); %>
<jdbc:cleanup scope="session" status="ok" />
```

付録C

制約と制限

この付録では、サポートされていないか制約のある機能、データベース固有の動作と制限事項が透過的な持続性に与える影響、および旧バージョンの透過的な持続性でクラスを作成した開発者のためのファイル移行情報について説明します。

ここでは、次の項目を扱います。

- サポートされていない機能と制約事項
- 次のデータベースとともに透過的な持続性を使用する場合に課される制約と制限
  - PointBase 3.5 Network (Multi-User) Server。IDE とともにバンドルされています。
  - Oracle 8.1.6 Thin Driver
  - WebLogic JDBC driver 5.1.0 for Microsoft SQL Server 2000
  - DB2 Universal Database, Version 7.1
  - Microsoft JDBC-ODBC Bridge
- 旧バージョンの透過的な持続性で作成されたクラスの移行操作

サポートされていない機能

透過的な持続性は、現時点で次の機能をサポートしていません。

- 主キーのないテーブル
- 主キーの値の更新機能
- 予備の列のある Join テーブル

- ユーザー定義の並行性グループ
- ユーザー定義、大型オブジェクト、および国際文字セットのデータ型。Blobs、Clobs、text、nChar、nVarchar、ntextなど
- 継承機能： 持続可能クラスは、別のクラスから直接的または間接的に拡張できません。
- 複数のデータベーススキーマをまたぐクラス間の関係性
- 循環する依存関係を含むオブジェクトグラフの挿入と削除
- テーブルのすべての主キー列を含まないビュー (単純主キーと合成主キー)。 透過的な持続性は、すべての主キー列を含まないビューをサポートしません。
- ビューにマップされたクラスの実行時の動作は、ビューの更新と削除について元になるデータベースの制限を受けます。その制限事項に違反した場合、データベースは例外をスローします。制限事項の一部を次に示します。
  - 集合関数 (`SUM`、`AVG`、`max`、`min`、`count`、および `count(*)`) を定義に含むビュー
  - ユーザー定義の関数を保持するビュー
  - `WITH CHECK OPTION` を定義に含むビュー
  - `group by` 節を定義に含むビュー
  - `order by` 節を定義に含むビュー

制約

次の機能はサポートされていますが、一部に制約があります。

アプリケーションクラスローダー

透過的な持続性は、関係する 2 つの持続可能クラスは同一のクラスローダーでロードされるものと仮定します。透過的な持続性では、同じクラス名を有する 2 つのクラスを異なるクラスローダーでロードできません。ロードを強行すると、`JDOFatalUserException`,「class *class.Name* loaded by multiple class loaders」という例外メッセージが表示されます。

アプリケーションサーバー環境では、この制約は、
com.sun.forte4j.persistence.model.multipleClassLoaders option を
使用することで解決されます。第 6 章 を参照してください。

コレクション関係の比較

コレクション関係とヌル以外の値は比較できません。照会を強行すると、
JDOUnsupportedOptionException 例外が発生します。

ユーザー定義の Clone() メソッド

透過的な持続性では、持続性認識クラスの持続インスタンスの新作クローンを一時的なものとする必要があります。ほとんどすべての場合において、この制約事項は、透過的な持続性の拡張機能 (enhancer) を使用することによって遵守できます。enhancer は、適切な clone() メソッドを使用するか (ユーザーが何も定義していない場合)、ユーザー定義の clone() メソッドにコードを追加します。

作成されたクローンは、生成またはユーザー定義されたクローンメソッドがスーパークラス (super.clone()) のクローンメソッドの呼び出しから戻ると同時に、一時的としてマークされます。そのため、持続性認識クラスのすべてのスーパークラスに含まれるユーザー定義クローンメソッドは、直接的または間接的に新作クローンの持続フィールドにアクセスするコードを呼び出します。そうした呼び出しが行われることから、透過的な持続性実行環境との間に対話が発生し、その後で、持続可能サブクラスにおいてクローンが一時的としてマークされます。

ユーザー定義のコンストラクタ

透過的な持続性実行環境は、拡張機能 (enhancer) によって追加された特殊なコンストラクタを使用して、持続可能クラスのインスタンスを作成します。このコンストラクタは、持続可能クラスのその他のコンストラクタ (ユーザー定義のコンストラクタなど) を呼び出さない代わりに、持続可能クラスのスーパークラスから引数のない (デフォルトとも呼ばれる) コンストラクタを呼び出します。そのため、持続可能クラスに次のような制約が課せられます。

- スーパークラスは、持続可能サブクラスにアクセス可能なデフォルトコンストラクタを提供しなければなりません。

- 持続可能クラスでは、ユーザー定義でないコンストラクタまたは非静的インスタンスフィールドの初期化が、照会または関係性のナビゲーションの結果として 透過的な持続性実行環境で作成されたインスタンスで実行されます。

データベースの制限と制約

特定のデータベースに次のような制限と制約が課されます。

PointBase 3.5 Network (Multi-User) Server

ここでは、IDE にバンドルされている PointBase Network Server 3.5 が特定の環境において透過的な持続性に及ぼす影響について説明します。

エラーメッセージ：「java.net.SocketException: Socket closed」

PersistenceManagerFactory が接続プールなしで構成され、一部の PersistenceManagerFactory のインスタンスが不要である場合、不要なコレクションのプロセスは、接続が終了する際に、次のメッセージをSystem.out に出力します。

```
java.net.SocketException: Socket closed
```

この例外は内部で無視されます。実行時への影響はありません。

PointBase Database version 3.4

PointBase Database version 3.4 は、引用符で囲まれた通常の識別子をサポートしないことから、透過的な持続性でサポートされません。

このバージョンでアプリケーションを実行するには、次の 2 行を使用してファイル .tpersistence.properties を作成することにより、3.5 の設定を上書きします。

```
database.pointbase.QUOTE_CHAR_END=
database.pointbase.QUOTE_CHAR_START=
```

データベースを呼び出すアプリケーションのルートディレクトリに、このファイルを配置します。

isEmpty() メソッド

フィルタで isEmpty() メソッドを使用すると、JDBC SQLException 例外がスローされます。このような照会の例を次に示します。

```
query.setFilter(;employees.isEmpty(););
```

PointBase Network Server の位置

「データベーススキーマ」ウィザードは、PointBase Network Server がデータベースを開始したディレクトリに配置されていると仮定します。たとえば、IDE からデータベースを開始した場合、ウィザードはデータベースが次の位置にあると仮定します。

Forte_Home\pointbase\network\databases

フェッチグループの複数の関係フィールド

PointBase Network Server を使用している場合、フェッチグループに複数の関係フィールドを配置できません。

解決策：それぞれの関連フィールドにおいて、「フェッチグループ」プロパティを「none」に設定します。

Oracle 8.1.6 Thin Driver

ここでは、特定の環境において Oracle 8.1.6 データベースが透過的な持続性 に及ぼす影響について説明します。

並行トランザクション

遮断レベルが SERIALIZABLE に設定されているデータストアのトランザクションは、Oracle において他のデータベースと異なる動作をします。たとえば、次の 2 つのトランザクションに注意してください。

トランザクション 1: オブジェクトをキャッシュに取り込み、キャッシュ内でオブジェクトフィールドを変更します。

トランザクション 2: 主キーの値が同じであるオブジェクトを別のキャッシュに取り込み、キャッシュ内でオブジェクトフィールドを変更します。

ほとんどのデータベースは各行に読み取りロックを配置するため、トランザクション 1 をコミットする前にトランザクション 2 をコミットすることが不可能になります。トランザクション 1 の前にトランザクション 2 のコミットを試行すると、Oracle データベースはトランザクション 2 をコミットした後に例外をスローし、メッセージ `cannot serialize access for this transaction` を表示します。

並行更新操作

Oracle データベースとともにマルチスレッド環境で並行更新操作を試行すると、プロセスがハングすることがあります。

接続の確立

Oracle Thin Driver では、接続の確立時にユーザー名とパスワードを指定する必要があります。ユーザー名とパスワードは、非管理環境で持続マネージャファクトリの初期化時に指定するか、管理環境でデータソースのプロパティの構成時に指定するか、メソッド `PersistenceManagerFactory.getPersistenceManager(`*user, password*`)` にヌル以外の引数を渡して指定します。

WebLogic JDBC Driver 5.1.0 for Microsoft SQL Server 2000

ここでは、特定の環境において、WebLogic JDBC driver 5.1.0 for Microsoft SQL Server 2000 が透過的な持続性に及ぼす影響について説明します。

1 対 1 の関係

外部キーの列の 1 つが独自の制約を課す場合、1 対 1 の関係に属するインスタンスを削除する際に、例外がスローされます。解決策：1 つのトランザクションで関係を取り消し、新しいトランザクションでインスタンスを削除します。

J2EE Reference Implementation Application Server

J2EE RI Application Server を WebLogic ドライバとともに使用し、ドライバファイルがライセンスファイルと分離されている場合、ライセンスファイルをドライバファイルにパッケージして、このドライバのすべてのコンポーネントに `java.security.AllPermission` を適用する必要があります。

DB2 Universal Database, Version 7.1

ここでは、特定の環境において DB2 Universal データベースが透過的な持続性に及ぼす影響について説明します。

1 対 1 の関係

DB2 が「外部キー」列に独自の制約を追加することから、1 対 1 の関係を削除したり「null」に設定することはできません。

独自の制約を有する列

独自の制約を有する列は、複数のヌルの値を含むことができません。

DT_VARCHAR2_2000 データ型

透過的な持続性が DB2 DT_VARCHAR2_2000 データ型をサポートする場合、次の制約が生じます。

- デフォルトのフェッチグループに DT_VARCHAR2_2000 フィールドを配置できません。照会が異常終了します。この事態を回避するには、テーブルへのマッピング期間中、デフォルトのフェッチグループから JDBC Type DT_VARCHAR2_2000 フィールドを明示的に除外します。
- DT_VARCHAR2_2000 フィールドの値がヌルと比較される照会に限り、DT_VARCHAR2_2000 フィールドを使用できます。たとえば、`DT_VARCHAR2_2000Field == null`、`DT_VARCHAR2_2000Field != null` などです。
- オプティミスティックなトランザクションをコミットしている間は、更新と削除を実行できません。代わりに、データストアのトランザクションを使用します。

- エントリのサイズが 4000 バイト以下の場合に限り、DT_VARCHAR2_2000 フィールドを更新できます。

Select 文

DB2 は、? <op> ? を伴う SELECT 文を生成する照会をサポートしません。この問題は、リテラルまたは照会パラメータを比較する場合に発生します。

また、更新ロックされているレコードを照会が選択した場合、アプリケーションがハングします。この事態は、更新されるインスタンスが照会実行より先にデータベースにフラッシュされた場合、データストアで発生します。

Microsoft JDBC-ODBC Bridge

ここでは、Microsoft JDBC-ODBC Bridge (SQLSRV32.DLL) version 2.0001 (03.70.0623) が透過的な持続性に及ぼす影響について説明します。

連結

文字列を連結する照会 (たとえば、フィルタ `startsWith`、`endsWith` を使用したり、`"Engi" + "neering"` のように + 記号を使用する照会) は 0 行を返しますが、例外はスローしません。

日付

2079-06-07 00:00:00.0 以降の日付で更新を行うと操作が異常終了し、例外メッセージ SQLException: `Datetime field overflow` が表示されます。

ファイルの移行

持続クラスのファイル形式は、本バージョンの透過的な持続性モジュールで変更されています。古いファイルは、本バージョンでも表示/実行が可能であり、新旧両方のファイルを使用するアプリケーションで使うこともできます。ただし、新しい形式のファイルは、旧バージョンの透過的な持続性モジュールで使用できません。

過去に作成した持続ファイルを開いて変更する場合、ユーザーのクラスを最新の形式に移行するかどうかを尋ねるメッセージが表示されます。

次の中から選択できます。

- 「いますぐ保存」。変更をコミットし、ファイルを即座に新形式に移行します。
- 「後で保存」。初めて保存された時点でファイルを新形式に移行します。
- 「取消し」。ファイルの変更は取り消され、ファイルの形式は当初のままとなります。

索引

N

O

P

R

V

W

X

あ