BEA WebLogic Server™

WebLogic エンタープライズ JavaBeans プログラマーズ ガイド

BEA WebLogic Server バージョン 7.0
マニュアルの改訂 : 2003 年 12 月 10 日

著作権

限定的権利条項

商標または登録商標

WebLogic エンタープライズ JavaBeans プログラマーズ ガイド

パート番号	マニュアルの日付	ソフトウェアのバージョン
なし	2003 年 12 月 10 日	BEA WebLogic Server バージョン 7.0

目次

このマニュアルの内容

1. WebLogic Server エンタープライズ JavaBean の概要

2. EJB の設計

3. メッセージ駆動型 Bean の設計

4. WebLogic Server EJB コンテナとサポートされるサービス

5. WebLogic Server のコンテナ管理による永続性サービス

6. WebLogic Server コンテナ用の EJB のパッケージ化

7. WebLogic Server への EJB のデプロイ

8. WebLogic Server EJB のユーティリティ

9. weblogic-ejb-jar.xml 文書型定義

10. weblogic-cmp-rdbms-jar.xml 文書型定義

このマニュアルの内容

このマニュアルでは、WebLogic Server 上でエンタープライズ JavaBeans (EJB) を開発およびデプロイする方法を説明します。このマニュアルの内容は以下のとおりです。

- 第 1 章「WebLogic Server エンタープライズ JavaBean の概要」では、WebLogic Server でサポートされる EJB の機能の概要について説明します。
- 第 2 章「EJB の設計」では、開発者が EJB を作成するために使用できる設計手法の概要について説明します。
- 第 3 章「メッセージ駆動型 Bean の設計」では、メッセージ駆動型 Bean を開発して WebLogic Server コンテナにデプロイする方法について説明します。
- 第 4 章「WebLogic Server EJB コンテナとサポートされるサービス」では、WebLogic Server コンテナを使って利用できるサービスについて説明します。
- 第 5 章「WebLogic Server のコンテナ管理による永続性サービス」では、WebLogic Server コンテナ内のエンティティ EJB で利用できる EJB のコンテナ管理による永続性サービスについて説明します。
- 第 6 章「WebLogic Server コンテナ用の EJB のパッケージ化」では、EJB をパッケージ化して WebLogic Server にデプロイするために必要な手順について説明します。
- 第 7 章「WebLogic Server への EJB のデプロイ」では、EJB コンテナに EJB をデプロイする手順について説明します。
- 第 8 章「WebLogic Server EJB のユーティリティ」では、EJB で使用する WebLogic Server に付属のユーティリティについて説明します。
- 第 9 章「weblogic-ejb-jar.xml 文書型定義」では、WebLogic Server に付属の `weblogic-ejb-jar.xml` ファイルにある、WebLogic 固有のデプロイメント記述子について説明します。

- 第 10 章「weblogic-cmp-rdbms-jar.xml 文書型定義」では、WebLogic Server に付属の `weblogic-cmp-rdbms-jar.xml` ファイルにある、WebLogic 固有のデプロイメント記述子について説明します。

対象読者

このマニュアルは、動的な Web ベースのアプリケーションで使用するエンタープライズ JavaBeans (EJB) の開発に関心のあるアプリケーション開発者を主な対象としています。EJB アーキテクチャ、XML コーディング、および Java プログラミングに読者が精通していることを前提として書かれています。

e-docs Web サイト

BEA WebLogic Server 製品のドキュメントは、BEA の Web サイトで入手できます。BEA のホーム ページで [製品のドキュメント] をクリックします。

このマニュアルの印刷方法

Web ブラウザの [ファイル | 印刷] オプションを使用すると、Web ブラウザからこのマニュアルのファイルを一度に 1 つずつ印刷できます。

このマニュアルの PDF 版は、WebLogic Server の Web サイトで入手できます。PDF を Adobe Acrobat Reader で開くと、マニュアルの全体 (または一部分) を書籍の形式で印刷できます。PDF を表示するには、WebLogic Server ドキュメントのホーム ページを開き、[ドキュメントのダウンロード] をクリックして、印刷するマニュアルを選択します。

Adobe Acrobat Reader は Adobe の Web サイト (http://www.adobe.co.jp) で無料で入手できます。

関連情報

BEA の Web サイトでは、WebLogic Server の全マニュアルを提供しています。また、WebLogic Server でエンタープライズ JavaBean を使用する際に役に立つ関連情報を以下に示します。

- Sun Microsystems の EJB 仕様の詳細については、JavaSoft EJB 仕様を参照してください。
- J2EE 仕様の詳細については、JavaSoft J2EE 仕様を参照してください。
- Sun Microsystems の EJB デプロイメント記述子の詳細については、JavaSoft EJB 仕様を参照してください。
- WebLogic Server の weblogic-ejb-jar.xml ファイルにあるデプロイメント記述子の詳細については、「weblogic-ejb-jar.xml 文書型定義」を参照してください。
- WebLogic Server の weblogic-cmp-rdbms-jar.xml ファイルにあるデプロイメント記述子の詳細については、第 10 章「weblogic-cmp-rdbms-jar.xml 文書型定義」を参照してください。
- トランザクションの詳細については、『WebLogic JTA プログラマーズ ガイド』を参照してください。
- WebLogic における JavaSoft Remote Method Invocation (RMI) 仕様の実装の詳細については、以下を参照してください。
  - JavaSoft Remote Method Invocation 仕様
  - 『WebLogic RMI プログラマーズ ガイド』
  - 『WebLogic RMI over IIOP プログラマーズ ガイド』

サポート情報

WebLogic Server のドキュメントに関するユーザからのフィードバックは弊社にとって非常に重要です。質問や意見などがあれば、電子メールで **docsupport-jp@beasys.com** までお送りください。寄せられた意見については、WebLogic Server ドキュメントを作成および改訂する BEA の専門の担当者が直に目を通します。

電子メールのメッセージには、ご使用のソフトウェアの名前とバージョン、およびドキュメントのタイトルと日付をお書き添えください。

本バージョンの BEA WebLogic Server について不明な点がある場合、または BEA WebLogic Server のインストールおよび動作に問題がある場合は、BEA WebSUPPORT (www.bea.com) を通じて BEA カスタマ サポートまでお問い合わせください。カスタマ サポートへの連絡方法については、製品パッケージに同梱されているカスタマ サポート カードにも記載されています。

カスタマ サポートでは以下の情報をお尋ねしますので、お問い合わせの際はあらかじめご用意ください。

- お名前、電子メール アドレス、電話番号、ファクス番号
- 会社の名前と住所
- お使いの機種とコード番号
- 製品の名前とバージョン
- 問題の状況と表示されるエラー メッセージの内容

表記規則

このマニュアルでは、全体を通して以下の表記規則が使用されています。

表記法	適用
〔Ctrl〕+〔Tab〕	複数のキーを同時に押すことを示す。
斜体	強調または書籍のタイトルを示す。

等幅テキスト	コード サンプル、コマンドとそのオプション、データ構造体とそのメンバー、データ型、ディレクトリ、およびファイル名とその拡張子を示す。等幅テキストはキーボードから入力するテキストも示す。
	例：
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

斜体の等幅テキスト	コード内の変数を示す。 例： `String` *`expr`*
すべて大文字のテキスト	デバイス名、環境変数、および論理演算子を示す。 例： LPT1 SIGNON OR
{ }	構文の中で複数の選択肢を示す。実際には、この括弧は入力しない。

表記法	適用
[ ]	構文の中で任意指定の項目を示す。実際には、この括弧は入力しない。 例： `buildobjclient [-v] [-o name ] [-f file-list]...[-l file-list]...`
|	構文の中で相互に排他的な選択肢を区切る。実際には、この記号は入力しない。
...	コマンドラインで以下のいずれかを示す。 ■ 引数を複数回繰り返すことができる。 ■ 任意指定の引数が省略されている。 ■ パラメータや値などの情報を追加入力できる。 実際には、この省略符号は入力しない。 例： `buildobjclient [-v] [-o name ] [-f file-list]...[-l file-list]...`
. . .	コード サンプルまたは構文で項目が省略されていることを示す。 実際には、この省略符号は入力しない。

1 WebLogic Server エンタープライズ JavaBean の概要

WebLogic Server には、Sun Microsystems の仕様で定義されているエンタープライズ JavaBean (EJB) アーキテクチャが実装されています。

注意： WebLogic Server は Sun の J2EE 仕様、EJB 1.1 仕様、および、EJB 2.0 仕様に準拠しています。EJB の機能および動作の説明箇所で、EJB 1.1 または EJB 2.0 向けと明記されている場合を除いては、このマニュアルのすべての情報は両方の実装に関連したものです。既存の EJB 1.1 Bean をこのバージョンの WebLogic Server にデプロイできます。ただし、新しい Bean を開発する場合、EJB 2.0 Bean を開発することをお勧めします。

以下の節では、WebLogic Server のエンタープライズ JavaBean の実装で導入された EJB の機能と変更点について概説します。

- エンタープライズ JavaBean の概要
- Java 仕様の実装
- WebLogic Server による EJB 2.0 のサポート
- EJB ロール
- WebLogic Server 7.0 の EJB 機能の強化
- EJB 開発者向けツール

エンタープライズ JavaBean の概要

エンタープライズ JavaBean は、ビジネス ロジックを実装する再利用可能な Java コンポーネントで、コンポーネントベースの分散ビジネス アプリケーションの開発を可能にします。EJB は EJB コンテナに収められ、永続性、セキュリティ、

トランザクション、同時実行性などの標準セットのサービスを提供します。エンタープライズ JavaBean は、サーバサイド コンポーネントを定義するための標準規格です。WebLogic Server のエンタープライズ JavaBean コンポーネント アーキテクチャの実装は、Sun Microsystems の EJB 仕様に基づいています。

EJB コンポーネント

EJB は、主に次の 3 つのコンポーネントで構成されます。

- **リモート インタフェース**。このインタフェースは、クライアントに対してビジネス ロジックを公開します。
- **ホーム インタフェース**。EJB ファクトリ。クライアントは、このインタフェースを使用して、EJB インスタンスを作成、検索、および削除します。
- **Bean クラス**。このインタフェースは、ビジネス ロジックを実装します。

EJB を作成するには、分散アプリケーションのビジネス ロジックを EJB の実装クラスにコーディングし、デプロイメント記述子ファイルのデプロイメント パラメータを指定し、EJB を JAR ファイルにパッケージ化します。EJB を WebLogic Server にデプロイするには、JAR ファイルから個別にデプロイする方法と、他の EJB および Web アプリケーションと一緒に EAR ファイルにパッケージ化して EAR ファイルをデプロイする方法があります。クライアント アプリケーションは、Bean のホーム インタフェースを使用して EJB を見つけたり、Bean のインスタンスを作成したりすることができます。クライアントは、EJB のリモート インタフェースを使用して EJB のメソッドを呼び出せるようになります。WebLogic Server は、EJB コンテナを管理し、データベース管理、セキュリティ管理、トランザクション サービスなどのシステムレベルのサービスへのアクセスを提供します。

EJB の種類

EJB 仕様では、以下の 4 種類のエンタープライズ JavaBean を定義しています。

- **ステートレス セッション**。この非永続 EJB のインスタンスは、メソッド間の対話または交信ステートを保存しないサービスを提供します。任意のインスタンスを任意のクライアントで使用できます。ステートレス セッション

Bean は、コンテナ管理または Bean 管理のどちらかのトランザクション境界定義を使用できます。

- **ステートフル セッション**。この非永続 EJB のインスタンスは、メソッド間およびトランザクション間で状態を保持します。各セッションは特定のクライアントに関連付けられます。ステートフル セッション Bean は、コンテナ管理または Bean 管理どちらかのトランザクション境界定義を使用できます。
- **エンティティ**。この永続 EJB のインスタンスは、通常はデータベース内の行であるデータのオブジェクト ビューを表します。エンティティ Bean は一意の識別子として主キーを持ちます。エンティティ Bean の永続性は、コンテナ管理でも Bean 管理でもかまいませんが、コンテナ管理によるトランザクションの境界設定のみを使用します。
- **メッセージ駆動型**。この EJB のインスタンスは Java Message Service (JMS) に統合されて、標準の JMS コンシューマとして動作し、サーバと JMS 間の非同期処理を実行するメッセージ駆動型 Bean の機能を提供します。WebLogic Server コンテナは、必要に応じて Bean のインスタンスを作成し、JMS メッセージをインスタンスに渡すことによってメッセージ駆動型 Bean と直接対話します。メッセージ駆動型 Bean は、コンテナ管理または Bean 管理のどちらかのトランザクション境界定義を使用できます。

注意： メッセージ駆動型 Bean は、Sun Microsystems EJB 2.0 仕様の一部です。EJB 1.1 仕様には含まれていません。

Java 仕様の実装

WebLogic Server は、次のような Java 仕様に準拠しています。

J2EE 仕様

WebLogic Server 7.0 は、J2EE 1.3 仕様に準拠しています。

EJB 2.0 仕様

WebLogic Server は、エンタープライズ JavaBeans 2.0 の実装に完全に準拠しており、それをプロダクション段階で使用することができます。

WebLogic Server EJB リソースの保護

大部分のビジネス アプリケーションでは、特定のユーザだけがある条件の下でアプリケーション リソースにアクセスできるように、リソースの保護措置がとられています。WebLogic Server には、このように EJB リソースを保護するための堅牢な機能が含まれています。

『WebLogic リソースのセキュリティ』では、アプリケーションに関する背景情報と、EJB を含む WebLogic Server アプリケーション リソースの保護について説明しています。

WebLogic Server による EJB 2.0 のサポート

WebLogic Server は、Sun Microsystems の EJB 2.0 仕様の実装をサポートしており、Sun Microsystems の EJB 1.1 仕様に準拠しています。ほとんどの場合、このバージョンの WebLogic Server で EJB 1.1 Bean を使用することができます。ただし、既存の EJB デプロイメントを、旧バージョンの WebLogic Server からこのバージョンの EJB コンテナに移行しなければならない場合があります。その場合は、8-26 ページの「DDConverter」で Bean の変換手順を参照してください。

Sun Microsystems の EJB 2.0 仕様では、以下の新機能がサポートされています。

- Java Messaging Service (JMS) コンシューマであるメッセージ起動型 Bean という新しいタイプの EJB。詳細については、第 3 章「メッセージ駆動型 Bean の設計」を参照してください。

- コンテナ管理の永続性を新しい方法で処理する、新しいエンティティ EJB コンテナ管理の永続性モデル。詳細については、第 5 章「WebLogic Server のコンテナ管理による永続性サービス」を参照してください。
- エンティティ EJB 間のコンテナ管理による関係を作成するモデルでは、実装クラスの Bean とデプロイメント記述子の間の関係を定義できます。詳細については、第 5 章「WebLogic Server のコンテナ管理による永続性サービス」を参照してください。
- EJB とそのプロパティをクエリするための EJB-QL という新しい標準クエリ言語。詳細については、第 5 章「WebLogic Server のコンテナ管理による永続性サービス」を参照してください。
- 新しい `ejbSelect` メソッド。このメソッドを使用すると、エンティティ EJB では、EJB-QL クエリを使用して、デプロイメント記述子に定義されているプロパティを内部的にクエリできます。詳細については、第 5 章「WebLogic Server のコンテナ管理による永続性サービス」を参照してください。
- セッションおよびエンティティ Bean 用のローカル インタフェース。EJB の関係は、ローカル インタフェースに基づいています。関係に関わる EJB には、ローカル インタフェースが必要です。詳細については、第 5 章「WebLogic Server のコンテナ管理による永続性サービス」を参照してください。
- エンティティ Bean の特定のインスタンスに固有ではないホーム ビジネス メソッドを実行することを可能にするホーム メソッド。エンティティ Bean に対して 1 つまたは複数のホーム メソッドを定義するには、ホーム インタフェースを使用します。詳細については、第 2 章「EJB の設計」を参照してください。

EJB ロール

EJB の開発プロセスは、以下のロールに明確に分けられます。

アプリケーション ロール

- **エンタープライズ Bean プロバイダ** - エンタープライズ Bean プロバイダは EJB を生成します。生成されるのは、1 つまたは複数の EJB が入った `ejb.jar` ファイルです。プロバイダは、このマニュアルで説明されている設計プロセスを使用して、WebLogic Server 環境にデプロイする EJB を設計します。

  設計プロセスの詳細については、第 2 章「EJB の設計」を参照してください。

- **アプリケーション アセンブラ** - アプリケーション アセンブラは、EJB を JAR、EAR、WAR などのデプロイ可能なユニットにまとめます。EJB とアプリケーション アセンブリに関する指示を含む JAR、EAR、または WAR ファイルが作成されます。これらの指示は、デプロイメント記述子によって設定されます。アセンブラは、設計プロセスと EJB デプロイメント記述子の要素に従って、デプロイメント ユニットをアセンブルします。

  設計プロセスの詳細については、第 2 章「EJB の設計」を参照してください。アセンブリ プロセスの詳細については、第 6 章「WebLogic Server コンテナ用の EJB のパッケージ化」を参照してください。デプロイメント記述子の詳細については、第 9 章「weblogic-ejb-jar.xml 文書型定義」と第 10 章「weblogic-cmp-rdbms-jar.xml 文書型定義」を参照してください。

インフラストラクチャ ロール

- **コンテナ プロバイダ** - コンテナ プロバイダは EJB のデプロイメント ツール、コンテナの監視および管理用ツール、デプロイされた EJB インスタンスの実行時のサポートを提供します。このサポートには、トランザクション管理、セキュリティ管理、クライアントのネットワーク分散、スケーラビリティなどのサービスが含まれます。コンテナ プロバイダは、このマニュアルで説明されているコンテナ管理プロセスを使用して、コンテナを提供します。

  コンテナ管理プロセスの詳細については、第 4 章「WebLogic Server EJB コンテナとサポートされるサービス」を参照してください。

- **永続性マネージャ プロバイダ** - 永続性マネージャ プロバイダは、EJB がコンテナ管理による永続性を利用する場合に、コンテナ内のエンティティ EJB

の永続性サポートを担当します。このサポートは、EJB とデータベース間でデータをやり取りするコードを生成するために、デプロイメント時に提供されます。永続性マネージャ プロバイダは、このマニュアルで説明されているデプロイ プロセスおよびコンテナ管理による永続性 (CMP) 情報を使用して、コンテナ管理による永続性を提供します。

コンテナ管理による永続性の詳細については 第 5 章「WebLogic Server のコンテナ管理による永続性サービス」を、デプロイ プロセスの詳細については 第 6 章「WebLogic Server コンテナ用の EJB のパッケージ化」を参照してください。

デプロイメントおよび管理ロール

- **デプロイヤ** - デプロイヤは、デプロイメント記述子のアプリケーション アセンブリ指示に従って、JAR、EAR、または WAR ファイルに収められている EJB を対象の環境にデプロイします。対象の環境には、WebLogic Server 環境とコンテナが含まれます。デプロイヤによって、EJB が対象の環境に合わせてカスタマイズされ、特定の EJB コンテナにデプロイされます。デプロイヤは、このマニュアルで説明されているデプロイ プロセスを使用して、EJB をデプロイします。

  デプロイ プロセスの詳細については、第 7 章「WebLogic Server への EJB のデプロイ」を参照してください。

- **システム管理者** - システム管理者は、WebLogic Server およびコンテナが含まれるコンピューティングおよびネットワーキング インフラストラクチャのコンフィグレーションと管理を行います。システム管理者は、『管理者ガイド』と WebLogic Server オンライン ヘルプに説明されている管理プロセスを使用して、デプロイ済みアプリケーションを実行時に管理します。

  システム管理者のタスクの詳細については、『管理者ガイド』を参照してください。

WebLogic Server 7.0 の EJB 機能の強化

このリリースの WebLogic Server では、EJB の以下の拡張機能が新しく導入されています。

動的クエリのサポート

動的クエリを使用すると、アプリケーション コードでプログラム的にクエリを作成したり実行できるようになります。そのため、どのファインダ クエリも静的なものにもはや留まらず、EJB のデプロイメント記述子にハードコーディングする必要がありません。新たなクエリを作成し実行するのに EJB を更新し再デプロイしなくて済み、EJB のデプロイメント記述子のサイズが縮小します。動的クエリの詳細については、5-27 ページの「動的クエリの使用」を参照してください。

メッセージ駆動型 Bean の移行サービスのサポート

メッセージ駆動型 Bean 移行サービスを使用すると、メッセージ駆動型 Bean および Java Messaging Service (JMS) サーバを同じクラスタ内の別のサーバに移行でき、その結果、メッセージ駆動型 Bean の回復が促進されます。WebLogic Server の以前のバージョンには、サーバ障害時における JMS サーバとメッセージ駆動型 Bean の回復メカニズムは備わっていませんでした。メッセージ駆動型 Bean 移行サービスの詳細については、3-13 ページの「メッセージ駆動型 Bean の移行サービス」を参照してください。

EJB CMP の複数のテーブル マッピングのサポート

複数のテーブル マッピングを利用することにより、1 つの EJB を 1 つの DBMS データベース内の複数のテーブルにマッピングできるようになります。WebLogic Server 固有の CMP コンテナ XML デプロイメント記述子を使用すると、複数の DBMS テーブルおよび列を EJB および EJB フィールドにマッピングできます。複数のテーブル マッピングの詳細については、5-40 ページの「EJB 2.0 CMP の複数のテーブル マッピング」を参照してください。

EJB WebLogic QL 拡張サポート

WebLogic Server のこのリリースでの EJB WebLogic QL 拡張サポートには、WebLogic QL クエリ言語に対する一連の拡張が含まれます。WebLogic QL は、EJB QL と呼ばれている EJB 2.0 クエリ言語を WebLogic Server 独自に拡張したものです。この拡張では次のような機能をサポートしています。

- サブクエリ
- 集約関数
- ResultSet を返すクエリ

  WebLogic の EJB QL 拡張の詳細については、5-12 ページの「EJB 2.0 用 EJB QL の使い方」を参照してください。

- NO WAIT を伴う SELECT FOR UPDATE

  TRANSACTION_READ_COMMITTED UPDATES の WebLogic の EJB QL 拡張の詳細については、4-45 ページの「Oracle データベースに関する特別な注意」を参照してください。

オプティミスティックな同時実行性のサポート

オプティミスティックな同時実行性のサポートは、WebLogic Server が提供する新たな同時方式です。キャッシング付きまたはキャッシングなしのオプティミスティックなサポートを提供します。WebLogic Server はトランザクションをコミットする前に、データが変更されていないことを確かめることにより、データの一貫性を保証します。この機能の詳細については、4-19 ページの「Optimistic 同時方式」を参照してください。

ReadOnly エンティティの同時実行性のサポート

読み込み専用エンティティ Bean の同時実行性のサポートにより、WebLogic Server では、Bean への同時アクセスを必要とするトランザクションごとに、別々の読み込み専用 Bean インスタンス をアクティブ化できます。これにより、EJB コンテナでの排他的ロックの必要性がなくなり、同時に同じ Bean にアクセ

スするリクエストを並列処理できます。読み込み専用エンティティ Bean のキャッシングについては、4-22 ページの「ReadOnly 同時方式」を参照してください。

組み合わせキャッシングのサポート

組み合わせキャッシングのサポートにより、1 つのキャッシュを複数エンティティ Bean で使用するようにコンフィグレーションできます。これまでは、アプリケーションの一部である各エンティティ Bean ごとに別々のキャッシュをコンフィグレーションする必要がありました。組み合わせキャッシングの詳細については、4-26 ページの「エンティティ Bean の組み合わせキャッシング」を参照してください。

リレーションシップ キャッシングのサポート

リレーションシップ キャッシングのサポートにより、関係する複数のエンティティ Bean を 1 つのキャッシュにロードすることができ、エンティティ Bean のパフォーマンスが向上します。また、関係する複数の Bean に結合クエリを発行することによってクエリの回数を減らせます。リレーションシップ キャッシングの詳細については、4-27 ページの「エンティティ Bean のリレーションシップ キャッシング」を参照してください。

EJB リンクのサポート

EJB リンクのサポートにより、アプリケーションコンポーネント内で宣言した EJB 参照を、同じ J2EE アプリケーションに含まれているエンタープライズ Bean にリンクできるようになります。EJB リンクの詳細については、5-61 ページの「EJB リンクの使用」を参照してください。

一括挿入のサポート

一括挿入のサポートにより、コンテナ管理による永続性 (CMP) Bean の作成における効率が向上します。EJB コンテナが 1 つの SQL 文で CMP Bean での複数回のデータベース挿入を実行できるようになるためです。この機能により、コンテナが何度もデータベースの挿入をしなくて済みます。この機能の詳細については、4-48 ページの「データベースの挿入サポート」を参照してください。

EJB 開発者向けツール

BEA では、EJB の作成とコンフィグレーションを支援するツールを提供しています。

スケルトン デプロイメント記述子を作成する ANT タスク

スケルトン デプロイメント記述子を作成するときに、WebLogic ANT ユーティリティを利用できます。ANT ユーティリティは WebLogic Server 配布キットと共に出荷されている Java クラスです。ANT タスクによって、EJB を含むディレクトリが調べられ、その `ejb.jar` ファイルを基にデプロイメント記述子が作成されます。ANT ユーティリティは、個別の EJB に必要なコンフィグレーションやマッピングに関する情報をすべて備えているわけではないので、ANT ユーティリティによって作成されるスケルトン デプロイメント記述子は不完全なものです。ANT ユーティリティがスケルトン デプロイメント記述子を作成した後で、テキスト エディタ、XML エディタ、または Administration Console の EJB デプロイメント記述子エディタを使ってデプロイメント記述子を編集し、EJB のコンフィグレーションを完全なものにします。

ANT ユーティリティを使ってデプロイメント記述子を作成する方法の詳細については、『WebLogic Server アプリケーションの開発』の「エンタープライズ JavaBeans のパッケージ化」を参照してください。

WebLogic Builder

WebLogic Builder は、ビジュアル環境を提供する開発ツールであり、アプリケーションのデプロイメント記述子 XML ファイルを編集できます。XML ファイルを編集する際、WebLogic Builder のインタフェース上で XML ファイルを見ることができます。この際、XML ファイルをテキスト編集する必要はありません。WebLogic Builder ツールの使い方については、『WebLogic Builder』を参照してください。

EJBGen

`EJBGen` は、エンタープライズ JavaBeans 2.0 のコード ジェネレータです。Bean クラス ファイルに javadoc タグで注釈を付けた後、`EJBGen` を使って EJB アプリケーションのリモート クラス、ホーム クラス、デプロイメント記述子ファイルを生成することができます。EJBGen とサポートされている javadoc タグの詳細については、8-1 ページの「EJBGen」を参照してください。

weblogic.Deployer

`weblogic.Deployer` コマンドライン ユーティリティは、削除された `weblogic.deploy` ユーティリティに置き換わり、WebLogic Server 7.0 で新たに加わったユーティリティです。`weblogic.Deployer` ユーティリティを使うと、コマンドライン、シェル スクリプト、または、Java 以外の自動化されたあらゆる環境からデプロイメントを開始できるようになります。

weblogic.Deployer の使い方とコマンド リストについては、「WebLogic Server デプロイメント」を参照してください。

WebLogic EJB デプロイメント記述子エディタ

WebLogic Server Administration Console には、EJB デプロイメント記述子エディタが統合されています。この統合エディタを使用する前に、少なくとも `ejb.jar` ファイルに追加する次のデプロイメント記述子ファイルのスケルトンを作成する必要があります。

- `ejb-jar.xml`
- `weblogic-ejb-jar.xml`
- `weblogic-cmp-rdbms-jar.xml`

詳細については、6-6 ページの「EJB デプロイメント記述子の指定と編集」を参照してください。

XML エディタ

XML エディタは Ensemble が提供する、XML ファイルの作成と編集のための簡単で使いやすいツールです。このツールを使うと、指定した DTD または XML スキーマに従って XML コードの有効性を検証できます。XML エディタは Windows または Solaris マシンで使用でき、Dev2Dev Online からダウンロードできます。

2　EJB の設計

以下の節では、WebLogic Server エンタープライズ JavaBean (EJB) を設計するためのガイドラインを示します。一部のヒントは、EJB と同じようにリモート オブジェクト モデルと Remote Method Invocation (RMI) にも適用できます。

- セッション Bean の開発
- エンティティ Bean の設計
- メッセージ駆動型 Bean の設計
- EJB での継承の使用
- デプロイされた EJB へのアクセス
- トランザクション リソースの保持

セッション Bean の開発

セッション Bean の設計方法のひとつに、モデル / ビュー設計の使用があります。ビューがグラフィカル ユーザ インタフェース (GUI) フォームであり、モデルは GUI にデータを供給するコードです。典型的なクライアント / サーバ システムでは、モデルはビューと同じサーバに存在し、サーバと通信します。

モデルは、セッション Bean の形態でサーバに配置します。それは、モデル セッション Bean が最終的な表示に影響しないことを除けば、HTML フォームのサポートを提供するサーブレットを配置することに似ています。GUI フォーム インスタンス (サーバでフォームの型として機能する) ごとに 1 つのモデル セッション Bean インスタンスが必要です。たとえば、フォームに表示する 100 個のネットワーク ノードがある場合は、それらのノードに相当する値の配列を返す `getNetworkNodes()` というメソッドを対応する EJB に用意します。

このアプローチでは、全体的なトランザクションの時間が短くなり、ネットワークの帯域幅が最小限で済みます。それに対して、GUI フォームでエンティティ EJB のファインダ メソッドを呼び出し、100 個のネットワーク ノードの参照を個別に取り出すアプローチを考えてみてください。それらの参照の 1 つ 1 つで、クライアントではデータストアに戻ってデータを取り出さなければならないため、かなりのネットワーク帯域幅が消費され、パフォーマンスが許容できないほど低下する場合があります。

エンティティ Bean の設計

エンティティ Bean を使用した RDBMS データの読み書きは、貴重なネットワーク リソースを消費します。ネットワーク トラフィックは、WebLogic Server と基底のデータストアの間だけでなく、クライアントと WebLogic Server の間でも発生する場合があります。以下のアドバイスに従ってエンティティ EJB データを適切にモデル化し、不要なネットワーク トラフィックを防止してください。

エンティティ Bean のホーム インタフェース

コンテナは、コンテナにデプロイされる各エンティティ Bean のホーム インタフェースの実装を提供し、クライアントが JNDI を通じてホーム インタフェースにアクセスできるようにします。エンティティ Bean のホーム インタフェースを実装するオブジェクトは EJBHome オブジェクトと呼ばれます。エンティティ Bean のホーム インタフェースを通じて、クライアントは以下の処理を行うことができます。

- `create()` メソッドを使用して、ホーム内に新しいエンティティ オブジェクトを作成する
- `finder()` メソッドを使用して、ホーム内の既存のエンティティ オブジェクトを検索する
- `remove()` メソッドを使用して、ホームからエンティティ オブジェクトを削除する
- 特定のエンティティ Bean のインスタンスに固有でないホーム メソッドを実行する

エンティティ EJB は大まかにする

システムのすべてのオブジェクトをエンティティ EJB としてモデル化しないでください。特に、ほんの数バイトの小さなデータをエンティティ EJB にすることは避けてください (ネットワーク リソースのトレードオフが成立しないため)。

たとえば、スプレッドシートのセルなどは細かすぎるので、ネットワーク経由で頻繁にアクセスすることは避ける必要があります。ただし、インボイスの項目の論理的なグループやスプレッドシートのセルの集合は、ビジネス ロジックが必要な場合は、エンティティ EJB としてモデル化できます。

追加のビジネス ロジックをエンティティ EJB にカプセル化する

大まかなオブジェクトであっても、ビジネス ロジックが不要であれば、エンティティ EJB としてモデル化するのは適切ではありません。たとえば、エンティティ EJB のメソッドの機能がデータ値の設定または読み込みだけの場合は、モデル化に RDBMS クライアントで JDBC 呼び出しを使用するか、またはセッション EJB を使用する方が適切です。

エンティティ EJB では、モデル化したデータに対してビジネス ロジックをカプセル化します。たとえば、「プラチナ」と「ゴールド」の顧客で別々のビジネス ルールを使用するバンキング アプリケーションでは、すべての顧客の口座がエンティティ EJB としてモデル化されます。その場合、EJB メソッドでは特定の顧客タイプのデータ フィールドを設定するか、読み込むときに適切なビジネス ロジックを適用できます。

エンティティ EJB のデータ アクセスを最適化する

エンティティ EJB では、結局は、データストアのフィールドがモデル化されます。できる限りエンティティ EJB を最適化して、データベース アクセスを合理化し、最小限に抑える必要があります。特に、以下の点に注意します。

- EJB データに対して結合の複雑さを制限します。
- データストアでディスク アクセスが必要となる長時間の処理を避けます。
- クライアントとデータストア間の往復回数を最低限に抑えるために、EJB メソッドでできる限り多くのデータが返されるようにします。たとえば、EJB クライアントでデータ フィールドを読み込む場合は、一括処理の `get/setAttributes()` メソッドを使用してネットワーク トラフィックを最小限に抑えます。

メッセージ駆動型 Bean の設計

メッセージ駆動型 Bean は、WebLogic JMS メッセージング システムでメッセージ コンシューマとして機能します。メッセージ駆動型 Bean の設計の詳細については、第 3 章「メッセージ駆動型 Bean の設計」を参照してください。

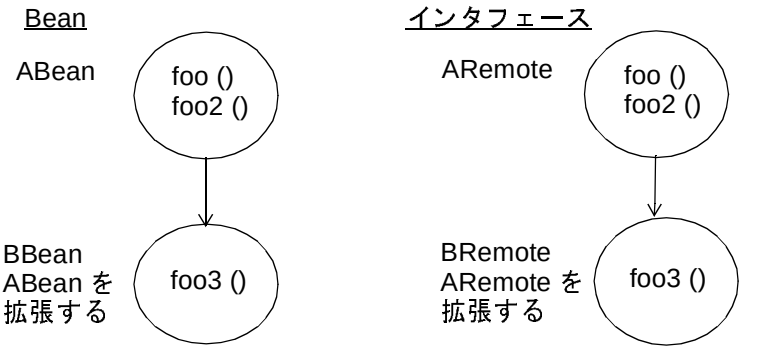
EJB での継承の使用

共通のコードを共有する互いに関連する Bean のグループをビルドする場合は、継承を利用することをお勧めします。ただし、EJB の実装に適用される継承の制限に注意する必要があります。

Bean 管理のエンティティ EJB の場合、`ejbCreate()` メソッドでは主キーが返されなければなりません。Bean 管理の EJB クラスから継承するどのクラスも、Bean 管理の EJB クラスが返すものとは異なる主キー クラスを返す `ejbCreate()` メソッドを実装することはできません。この制限は、新しいクラスが基本の EJB の主キー クラスから派生する場合でも適用されます。また、この制限は、Bean の `ejbFind()` メソッドにも適用されます。

また、その他の EJB 実装から継承する EJB はインタフェースを変更します。たとえば、次の図は、リモートからアクセスできる新しいメソッドが派生 Bean で追加される状況を説明しています。

図 2-1　リモートからアクセス可能なメソッドを追加する派生 Bean (BBean)



さらに、`AHome.create()` と `BHome.create()` では別々のリモート インタフェースが返されるので、`BHome` インタフェースは `AHome` インタフェースから継承することはできない、という制限もあります。特定のクラスに固有のメソッド、スーパークラスから継承するメソッド、またはサブクラスでオーバーライドされるメソッドを Bean で実装するために継承を使用することはできます。継承の例については、WebLogic Server 配布キットにあるクラス内の EJB 1.1 JavaBean のサブクラス Child のサンプルを参照してください。

デプロイされた EJB へのアクセス

WebLogic Server では、EJB のホーム インタフェースとリモートで機能するリモート インタフェースの実装が自動的に作成されます。つまり、EJB と同じサーバにあるクライアントでも、リモート コンピュータ上にあるクライアントでも、デプロイされた EJB に同じようにアクセスできるということです。

EJB では、Java Naming and Directory Interface (JNDI) を使用して環境プロパティを指定する必要があります。さまざまなマシン、アプリケーション サーバ、コンテナなど、ネットワーク上のあらゆる場所にあるホーム EJB が含まれるように EJB クライアントの JNDI ネームスペースをコンフィグレーションできます。

ただし、エンタープライズ アプリケーション システムを設計するときには、EJB とクライアントの間でネットワークを経由してデータを転送することの影響も考慮する必要があります。ネットワークのオーバーヘッドがあるため、Bean へのアクセスは、リモート クライアントからよりも「ローカル」クライアント (サーブレットまたは別の EJB) からの方がはるかに効率的です。リモート クライアントの場合は、データをマーシャリングし、ネットワーク経由で転送して、また復元しなければなりません。

EJB にローカル クライアントからアクセスする場合とリモート クライアントからアクセスする場合の違い

EJB へローカル クライアントからアクセスする場合と、リモート クライアントからアクセスする場合の違いは、Bean の `InitialContext` を取得する方法にあります。リモート クライアントでは、WebLogic Server `InitialContext` ファクトリを使用して `InitialContext` を取得します。通常、WebLogic Server のローカル クライアントでは、次の抜粋部分のように、`getInitialContext` メソッドを使用してこのルックアップを実行します。

図 2-2 ルックアップを実行するローカル クライアントのコード例

```
...
Context ctx = getInitialContext("t3://localhost:7001", "user1", "user1Password");
...
static Context getInitialContext(String url, String user, String password) {

   Properties h = new Properties();
   h.put(Context.INITIAL_CONTEXT_FACTORY,
      "weblogic.jndi.WLInitialContextFactory");
   h.put(Context.PROVIDER_URL, url);
   h.put(Context.SECURITY_PRINCIPAL, user);
h.put(Context.SECURITY_CREDENTIALS, password);

   return new InitialContext(h);

}
```

EJB の内部クライアント (サーブレットなど) では、単純に、次のようなデフォルトのコンストラクタを使用して `InitialContext` を作成できます。

```
Context ctx = new InitialContext();
```

EJB インスタンスの同時アクセスに関する制限

データベース同時実行性は同時アクセス オプションのデフォルトかつ推奨設定ですが、複数のクライアントが排他的同時アクセス オプションを使用して、EJB に順次的にアクセスすることもできます。この排他的オプションを使用する場合、2 つのクライアントでエンティティ EJB の同じインスタンス (主キーが同じインスタンス) へのアクセスが同時に試行されると、2 番目のクライアントは EJB が利用可能になるまでブロックされます。データベースの同時実行性オプションの詳細については、4-18 ページの「Exclusive 同時方式」を参照してください。

ステートフル セッション EJB に同時にアクセスすると、`RemoteException` が発生します。ステートフル セッション EJB に関するこのアクセス制限は、EJB クライアントが WebLogic Server の内部にあるのかそれともリモートにあるのかに関係なく適用されます。ただし、`allow-concurrent-calls` オプションを、ステートフル セッション Bean インスタンスがメソッドの同時呼び出しを許可するように指定できます。

複数のサーブレット クラスがセッション EJB にアクセスする場合は、サーブレット クラスのインスタンスごとではなくサーブレット スレッドごとに独自のセッション EJB インスタンスを使用する必要があります。同時アクセスを避けるために、JSP またはサーブレットはリクエスト スコープでステートフル セッションを使用できます。

EJB 参照のホーム ハンドルへの格納

EJB インスタンスの `EJBHome` オブジェクトを取得した後、クライアントでは `getHomeHandle()` を呼び出してそのホーム オブジェクトへのハンドルを作成できます。`getHomeHandle()` では、後で同じ EJB インスタンスのホーム インタフェースを取得するために使用できる `HomeHandle` オブジェクトが返されます。

クライアントでは、HomeHandle オブジェクトを別のクライアントに引数として渡すことができ、受け取り側のクライアントではそのハンドルを使用して同じ EJBHome オブジェクトへの参照を取得できます。クライアントでは、HomeHandle をシリアライズし、後の使用を目的としてファイルに格納することもできます。

ファイアウォールを介したホーム ハンドルの使用

デフォルトでは、WebLogic Server は、その IP アドレスを EJB の HomeHandle オブジェクトに格納します。これが、特定のファイアウォール システムで問題になる場合があります。ファイアウォールを介して渡されたホーム ハンドルを使用するときに EJBHome オブジェクトが見つからない場合は、次の手順に従います。

1. WebLogic Server を起動します。
2. WebLogic Server Administration Console を起動します。
3. 左ペインで [サーバ] ノードを展開し、サーバを選択します。
4. 右ペインでそのサーバの [コンフィグレーション] タブを選択し、次に [チューニング] タブを選択します。
5. [許可されたリバース DNS] ボックスをチェックし、DNS の逆引き参照を有効にします。

DNS の逆引き参照が有効になると、WebLogic Server では IP アドレスではなくサーバの DNS 名が EJB ホーム ハンドルに格納されます。

トランザクション リソースの保持

通常、データベース トランザクションは、オンライン トランザクション処理システムで最も貴重なリソースの 1 つです。WebLogic Server で EJB を使用する場合、トランザクション リソースはそのデータベース接続との関係によってさらに価値が高まります。

WebLogic Server では、1 つの接続プールを使用して複数の同時データベース要求に対応できます。接続プールの効率は、そのプールを使用するデータベース トランザクションの数と長さによって大きく左右されます。トランザクション非対応のデータベース要求の場合、WebLogic Server では同じ接続を別のクライアントが使用できるように接続の割り当てと割り当て解除を非常に敏速に行うことができます。ただし、トランザクション対応の要求の場合、そのトランザクションの間、接続はクライアントによって「予約状態」になります。

トランザクションをシステム上で最適な方法で使用するには、常に「内部から外部」という方法でトランザクションの境界を設定します。トランザクションの開始と終了ができる限りシステム (データベース) の「内部」になるようにして、必要な場合にのみ「外部」(クライアント アプリケーション方向) に向かうようにします。以降の節では、この規則について詳しく説明します。

トランザクションの管理をデータストアに許可する

多くの RDBMS システムは、オンライン トランザクション処理 (OLTP) トランザクション用の高性能ロック システムを備えています。Tuxedo などのトランザクション処理 (TP) モニタを利用すれば、RDBMS システムでは複数のデータストアにまたがる複雑な意志決定支援のクエリを管理することもできます。基底のデータストアにそのような機能がある場合は、できる限りその機能を使用します。RDBMS による自動的なトランザクションの境界設定を妨げないようにしてください。

EJB に対して Bean 管理のトランザクションの代わりにコンテナ管理のトランザクションを使用する

Bean 管理によるトランザクションの境界設定はなるべく使用しません。特殊な目的のために Bean 管理のトランザクションが必要となる場合以外は、WebLogic Server のコンテナ管理によるトランザクションの境界設定を使用してください。

Bean 管理のトランザクションが必要になるのは、以下のような場合です。

- 1 回のメソッド呼び出しで複数のトランザクションを定義する場合。WebLogic Server では、メソッドごとにトランザクションの境界を設定します。

  注意： ただし、1 回のメソッド呼び出しで複数のトランザクションを使用する代わりに、メソッドを複数のメソッドに分け、それぞれがコンテナ管理のトランザクションを使用する方法をお勧めします。

- 複数の EJB メソッド呼び出しに「またがる」トランザクションを定義する場合。たとえば、あるメソッドを使用してトランザクションを開始し、別のメソッドでトランザクションをコミットまたはロールバックするステートフル セッション EJB を定義する場合です。

  注意： EJB オブジェクトの機能についての詳しい情報が必要になるため、このような動作はできる限り避けてください。ただし、どうしても必要な場合は、Bean 管理によるトランザクションの調整を利用し、個々のメソッドに対するクライアントの呼び出しを調整する必要があります。

アプリケーションからトランザクションの境界を設定しない

通常、クライアント アプリケーションは長期間にわたってアクティブな状態を維持できるとは限りません。クライアントによってトランザクションが開始され、コミットする前にそのクライアントが終了すると、WebLogic Server で貴重なトランザクションと接続のリソースが失われてしまいます。さらに、クライアントがトランザクションの途中で終了しない場合でも、ユーザによるデータのコミットまたはロールバックに依存する場合はトランザクションが途中で終わってしまうこともあります。できる限り、トランザクションの境界は WebLogic Server または RDBMS のレベルで設定してください。

トランザクションの境界設定の詳細については、4-42 ページの「トランザクション管理の責任範囲」を参照してください。

コンテナ管理 EJB ではトランザクション対応データソースを常に使用する

JDBC データソース ファクトリをコンテナ管理 EJB とともに使用できるようにコンディグレーションする場合には、トランザクション非対応のデータソース (`DataSource`) ではなくトランザクション対応のデータソース (`TXDataSource`) を必ずコンフィグレーションします。トランザクション非対応のデータソースの場

合、JDBC 接続はコンテナ管理トランザクションの一部ではなく、挿入操作や更新操作の度にデータベースに即座にコミットする自動コミットモードで動作します。

3　メッセージ駆動型 Bean の設計

以下の節では、メッセージ駆動型 Bean を開発し、WebLogic Server にデプロイする方法について説明します。メッセージ駆動型 Bean では標準の Java Messaging Service (JMS) API を部分的に利用するので、メッセージ駆動型 Bean を実装する前に WebLogic JMS を理解する必要があります。詳細については、『WebLogic JMS プログラマーズ ガイド』を参照してください。

- メッセージ駆動型 Bean とは
- メッセージ駆動型 Bean の開発とコンフィグレーション
- メッセージ駆動型 Bean の呼び出し
- Bean インスタンスの作成と削除
- WebLogic Server でのメッセージ駆動型 Bean のデプロイ
- メッセージ駆動型 Bean でのトランザクション サービスの使用
- メッセージ駆動型 Bean の移行サービス
- 非 BEA JMS プロバイダのメッセージ駆動型 Bean のコンフィグレーション
- JMS サーバまたは非 BEA サービス プロバイダへの再接続
- JMS 分散送り先でリスンするための MDB のコンフィグレーション

メッセージ駆動型 Bean とは

メッセージ駆動型 Bean は、WebLogic JMS メッセージング システムでメッセージ コンシューマとして機能する EJB です。標準の JMS メッセージ コンシューマの場合と同じように、メッセージ駆動型 Bean では JMS キューまたは JMS トピックからメッセージを受信し、そのメッセージの内容に基づいてビジネス ロジックを実行します。

デプロイメント時にキューまたはトピックのリスナを作成すると、WebLogic Server では受信メッセージを処理するために必要に応じてメッセージ駆動型 Bean のインスタンスが自動的に作成および削除されます。

メッセージ駆動型 Bean と標準の JMS コンシューマとの違い

メッセージ駆動型 Bean は EJB として実装されるため、標準の JMS コンシューマでは利用できないサービスからの恩恵を受けます。最も重要なことは、メッセージ駆動型 Bean のインスタンスは、全面的に WebLogic Server EJB コンテナによって管理されるということです。1 つのメッセージ駆動型 Bean クラスを使用することで、WebLogic Server では大量のメッセージを並行して処理するために必要に応じて複数の EJB インスタンスが作成されます。それとは対照的に、標準的な JMS メッセージング システムの場合は、サーバ全体のセッション プールを利用する `MessageListener` クラスを開発者が作成しなければなりません。

WebLogic Server コンテナでは、セキュリティ サービスや自動トランザクション管理など、他の標準的な EJB サービスもメッセージ駆動型 Bean に対して提供されます。それらのサービスの詳細については、4-41 ページの「トランザクション管理」と 3-11 ページの「メッセージ駆動型 Bean でのトランザクション サービスの使用」を参照してください。

また、メッセージ駆動型 Bean は、EJB の「一度書けば、どこにでもデプロイできる」性質からも恩恵を受けます。JMS `MessageListener` は特定のセッション プール、キュー、またはトピックと関連付けられますが、メッセージ駆動型 Bean はサービス リソースにとらわれずに開発できます。メッセージ駆動型 Bean のキューとトピックはデプロイメント時にのみ割り当てられ、WebLogic Server にあるリソースが利用されます。

注意： 標準の JMS リスナにはないメッセージ駆動型 Bean の 1 つの制限は、特定のメッセージ駆動型 Bean のデプロイメントが 1 つのキューまたはトピックとしか関連付けられないことです (3-9 ページの「メッセージ駆動型 Bean の呼び出し」を参照)。アプリケーションにおいて、1 つの JMS コンシューマで複数のキューまたはトピックからのメッセージに対応しなければならない場合は、標準の JMS コンシューマを使用するか、または複数のメッセージ駆動型 Bean クラスをデプロイする必要があります。

メッセージ駆動型 Bean とステートレス セッション EJB との違い

メッセージ駆動型 Bean のインスタンスの動的な作成と割り当ては、ステートレス セッション EJB のインスタンスの動作にいくつかの点で似ています。ただし、メッセージ駆動型 Bean は、以下のような重要な点でステートレス セッション EJB (および他の種類の EJB) とは異なります。

- メッセージ駆動型 Bean では、シリアライズされたメソッド呼び出しのシーケンスではなく、複数の JMS メッセージが非同期で処理される。
- メッセージ駆動型 Bean にはホーム インタフェースやリモート インタフェースがない。したがって、内部または外部のクライアントから直接アクセスできません。クライアントは、JMS キューまたは JMS トピックにメッセージを送信することで間接的にメッセージ駆動型 Bean と対話します。

注意： WebLogic Server コンテナだけが、必要に応じて Bean のインスタンスを作成し、JMS メッセージをインスタンスに渡すことによってメッセージ駆動型 Bean と直接対話します。

- WebLogic Server によってメッセージ駆動型 Bean のライフサイクル全体が管理される。クライアントの要求や API の呼び出しでインスタンスを作成または削除することはできません。

トピックとキューの並行処理

メッセージ駆動型 Bean (MDB) は、トピックおよびキューの並行処理をサポートしています。以前は、キューの並行処理のみがサポートされていました。

同時実行性をサポートするには、コンテナで実行キューのスレッドを使用します。`weblogic-ejb-jar.xml` ファイルの `max-beans-in-free-pool` デプロイメント記述子のデフォルト設定では、ほとんどの並行処理がサポートされます。並行して実行されるコンシューマの数を制限する場合を除き、この値を変更しないでください。

注意： メッセージを一度に受け取るための最大 MDB 数 (`max-beans-in-free-pool` デプロイメント記述子要素でコンフィグレーションする) は、最大実行スレッド数を超過できません。たとえば、max-beans-in-free-pool が 50 に設定されており、最大実行スレッド数が 25 に設定されている場合、実際にメッセージを受け取る MDB は 25 個だけです。

`max-beans-in-free-pool` の詳細については、9-52 ページの「max-beans-in-free-pool」を参照してください。

メッセージ駆動型 Bean の開発とコンフィグレーション

メッセージ駆動型 EJB を作成するには、Bean を適切に動作させるための一般的な慣習に従うだけでなく、JavaSoft EJB 2.0 仕様で説明されている規約にも従う必要があります。メッセージ駆動型 Bean クラスを作成したら、XML 形式の EJB デプロイメント記述子ファイルで Bean のデプロイメント記述子要素を指定することによって、WebLogic Server 用に Bean をコンフィグレーションします。

メッセージ駆動型 Bean を作成するには、次の手順に従います。

1. `javax.ejb.MessageDrivenBean` インタフェースと `javax.jms.MessageListener` インタフェースの両方を実装するソース ファイル (メッセージ駆動型 Bean クラス) を作成します。

   メッセージ駆動型 Bean クラスでは、以下のメソッドを定義する必要があります。

   - コンテナがフリー プールでメッセージ駆動型 Bean のインスタンスを作成するために使用する `ejbCreate()` メソッドを 1 つ。
   - Bean のコンテナがメッセージを受け取ったときに呼び出す `onMessage()` メソッドを 1 つ。このメソッドには、メッセージを処理するビジネス ロジックが格納されます。
   - Bean インスタンスにその環境 (特定のデプロイメント記述子の値) に関する情報を提供する `setMessageDrivenContext{}` メソッドを 1 つ。この

コンテキストは、EJB コンテナによって提供されるサービスに Bean クラスがアクセスする場合にも使用されます。

- メッセージ駆動型 Bean インスタンスをフリー プールから削除する `ejbRemove()` メソッドを 1 つ。

メッセージ駆動型 Bean クラスの出力例について、詳しくは 3-7 ページの「メッセージ駆動型 Bean クラスの必要条件」を参照してください。

2. メッセージ駆動型 Bean に対して、次の XML デプロイメント記述子ファイルを指定します。
   - `ejb-jar.xml`
   - `weblogic-ejb-jar.xml`
   - `weblogic-cmp-rdbms-jar.xml`

   XML ファイルの指定について、詳しくは 6-6 ページの「EJB デプロイメント記述子の指定と編集」を参照してください。
3. Bean の `ejb-jar.xml` ファイルで `message-driven` 要素を設定して、Bean を宣言します。
4. Bean の `ejb-jar.xml` ファイルで `message-driven-destination` 要素を設定して、Bean がトピック用かキュー用かを指定します。
5. 関連付けられたトピックを恒久トピックとするかどうかを指定するときは、Bean の `ejb-jar.xml` ファイルで `subscription-durability` サブ要素を指定します。
6. Bean で独自のトランザクション境界を設定する場合、使用する JMS 確認応答を指定するために `acknowledge-mode` サブ要素を設定します。この要素の値は `AUTO_ACKNOWLEDGE` (デフォルト) または `DUPS_OK_ACKNOWLEDGE` のいずれかです。
7. トランザクション境界をコンテナで管理する場合、Bean の `ejb-jar.xml` ファイルで `transaction-type` 要素を設定して、メソッド呼び出しをエンタープライズ Bean のメソッドに委託するときにコンテナがトランザクション境界を管理する方式を指定します。

次の例は、`ejb-jar.xml` ファイルでのメッセージ駆動型 Bean の指定方法を示したものです。

図 3-1 `ejb-jar.xml` ファイルの XML スタンザの例

```
<enterprise-beans>
```

```
        <message-driven>
                <ejb-name>exampleMessageDriven1</ejb-name>
<ejb-class>examples.ejb20.message.MessageTraderBean</ejb-class>
                <transaction-type>Container</transaction-type>
                <message-driven-destination>
                        <destination-type>
                                javax.jms.Topic
                        </destination-type>
                </message-driven-destination>
                ...
        </message-driven>
        ...
</enterprise-beans>
```

8. Bean の `weblogic-ejb-jar.xml` ファイルで `message-driven-descriptor` 要素を設定して、メッセージ駆動型 Bean を WebLogic Server における JMS 送り先と関連付けます。

次の例は、`weblogic-ejb-jar.xml` ファイルでのメッセージ駆動型 Bean の設定方法を示したものです。

図 3-2　`weblogic-ejb-jar.xml` ファイルの XML スタンザの例

```
<message-driven-descriptor>
        <destination-jndi-name>...</destination-jndi-name>
</message-driven-descriptor>
```

9. 6-11 ページの「デプロイメント ディレクトリへの EJB のパッケージ化」の手順に従って、メッセージ駆動型 Bean クラスをコンパイルして生成します。
10. 7-9 ページの「コンパイル済み EJB ファイルのデプロイ」の手順に従って、Bean を WebLogic Server にデプロイします。

コンテナは、メッセージ駆動型 Bean インスタンスを実行時に管理します。

メッセージ駆動型 Bean クラスの必要条件

EJB 2.0 仕様では、メッセージ駆動型 Bean クラスでメソッドを定義するための詳細なガイドラインが提供されます。次の出力は、メッセージ駆動型 Bean クラスの基本的な構成要素を示しています。クラス、メソッド、およびメソッド宣言は、**太字**で表示されています。

図 3-3 メッセージ駆動型 Bean の基本コンポーネントの出力例

```
public class MessageTraderBean implements MessageDrivenBean,
MessageListener{
        public MessageTraderBean() {...};
                // EJB コンストラクタは必須。パラメータは
                // 受け付けない。コンストラクタは
                // abstract としては宣言しない
        public void ejbCreate() (...)
                // ejbCreate () は必須。パラメータは受け付けない
                // throws 句を使用する場合は、アプリケーション例外を
                // 含めない。ejbCreate() は final または static としては
宣言しない
        public void onMessage(javax.jms.Message MessageName) {...}
                // onMessage() は必須であり、javax.jms.Message 型の
                // パラメータを必ず 1 つとる。throws 句を使用する場合は
                // アプリケーション例外を含めない。onMessage() は final
                // または static としては宣言しない
        public void ejbRemove() {...}
                // ejbRemove() は必須。パラメータは受け付けない。
                // throws 句を使用する場合は、アプリケーション例外を
                //含めない。ejbRemove() は final または static としては宣
言しない
        // EJB クラスでは finalize() メソッドを定義できない
}
```

メッセージ駆動型 Bean コンテキストの使用

WebLogic Server では、`setMessageDrivenContext()` を呼び出して、メッセージ駆動型 Bean インスタンスをコンテナ コンテキストと関連付けます。これは、クライアント コンテキストではありません。クライアント コンテキストは、JMS メッセージでは渡されません。WebLogic Server では、コンテナ コンテキストが EJB に提供されます。そのプロパティには、`MessageDrivenContext` インタフェースの以下のメソッドを使用してインスタンスからアクセスできます。

- `getCallerPrincipal()` - このメソッドは EJB コンテキスト インタフェースから継承されるので、メッセージ駆動型 Bean インスタンスでは呼び出さないでください。
- `isCallerInRole()` - このメソッドは EJB コンテキスト インタフェースから継承されるので、メッセージ駆動型 Bean インスタンスでは呼び出さないでください。
- `setRollbackOnly()` - EJB では、コンテナ管理によるトランザクションの境界設定を利用する場合にのみこのメソッドを使用できます。
- `getRollbackOnly()` - EJB では、コンテナ管理によるトランザクションの境界設定を利用する場合にのみこのメソッドを使用できます。
- `getUserTransaction()` - EJB では、Bean 管理によるトランザクションの境界設定を利用する場合にのみこのメソッドを使用できます。

注意： `getEJBHome()` も `MessageDrivenContext` インタフェースの一部として継承されますが、メッセージ駆動型 Bean にはホーム インタフェースがありません。メッセージ駆動型 EJB のインスタンスから `getEJBHome()` を呼び出すと、`IllegalStateException` が送出されます。

onMessage() によるビジネス ロジックの実装

メッセージ駆動型 Bean の `onMessage()` メソッドでは、その EJB のビジネス ロジックが実装されます。WebLogic Server では、EJB と関連付けられている JMS キューまたは JMS トピックがメッセージを受信したときに、JMS メッセージ オ

ブジェクトをそのまま引数として渡して `onMessage()` を呼び出します。メッセージを解析し、`onMessage()` の必要なビジネス ロジックを実行するのは、メッセージ駆動型 Bean の役割です。

ビジネス ロジックが非同期のメッセージ処理に対応できるようにしておきます。たとえば、EJB では、クライアントから送信された順序でメッセージを受信できるわけではありません。コンテナでのインスタンス プーリングにより、メッセージが順番に受信または処理されることはありません。ただし、メッセージ駆動型 Bean の特定のインスタンスに対する個々の `onMessage()` 呼び出しはシリアライズされます。

詳細については、javax.jms.MessageListener.onMessage() を参照してください。

例外の処理

メッセージ駆動型 Bean のメソッドは、`onMessage()` であっても、アプリケーション例外または `RemoteException` を送出してはなりません。メソッドでそのような例外が送出されると、WebLogic Server では `ejbRemove()` を呼び出すことなく即座に EJB のインスタンスが削除されます。ただし、クライアントの観点からすれば、その EJB は依然として存在していることになります。なぜなら、以降のメッセージは WebLogic Server によって作成される新しい Bean インスタンスに転送されるからです。

メッセージ駆動型 Bean の呼び出し

JMS キューまたは JMS トピックがメッセージを受信すると、WebLogic Server では次のようにして関連するメッセージ駆動型 Bean を呼び出します。

1. WebLogic Server が新しい Bean インスタンスを取得します。

WebLogic Server では、`weblogic-ejb-jar.xml` ファイルで設定される `max-beans-in-free-pool` 属性を使用して、新しい Bean インスタンスがフリー プールで使用可能かどうかを判定します。

2. Bean インスタンスがフリー プールで使用可能な場合、WebLogic Server はそのインスタンスを使用します。

   フリー プールに使用可能な Bean インスタンスがなく、`max-beans-in-free-pool` で指定された制限に達している場合、WebLogic Server は Bean インスタンスが解放されるまで待機します。この属性の詳細については、9-60 ページの「max-beans-in-free-pool」を参照してください。

   Bean インスタンスがフリー プールになく、`max-beans-in-free-pool` で指定された制限に達していない場合、WebLogic Server は Bean の `ejbCreate()` メソッドを呼び出して新しい Bean インスタンスを作成し、続けて Bean の `setMessageDrivenContext()` メソッドを呼び出してそのインスタンスをコンテナ コンテキストに関連付けます。3-8 ページの「メッセージ駆動型 Bean コンテキストの使用」で説明されているように、Bean ではこのコンテキストの要素を利用できます。

3. WebLogic Server では、Bean に関連付けられている JMS キューまたはトピックでメッセージを受け取ると、Bean の `onMessage()` メソッドを呼び出して、ビジネス ロジックを実装します。

   3-8 ページの「onMessage() によるビジネス ロジックの実装」を参照してください。

注意： これらのインスタンスはプールに配置できます。

Bean インスタンスの作成と削除

WebLogic Server コンテナでは、メッセージ駆動型 Bean の `ejbCreate()` および `ejbRemove()` メソッドを呼び出して、Bean クラスのインスタンスを作成または削除します。各メッセージ駆動型 Bean には、少なくとも 1 つの `ejbCreate()` および `ejbRemove()` メソッドが必要です。WebLogic Server コンテナでは、これらのメソッドを使用して、JMS キューまたはトピックからメッセージを受信して Bean インスタンスが作成されたときに作成関数を、トランザクションがコミッ

トされて Bean インスタンスが削除されたときに削除関数を処理します。WebLogic Server は、JMS キューまたはトピックからメッセージを受け取ります。

他の EJB タイプの場合と同じように、`ejbCreate()` メソッドでは Bean の活動に必要なあらゆるリソースを用意しなければなりません。`ejbRemove()` メソッドでは、WebLogic Server によってインスタンスが削除される前にリソースを解放しなければなりません。

メッセージ駆動型 Bean では、`ejbRemove()` メソッドの外側においても何らかのかたちで通常のクリーンアップ ルーチンを実行する必要があります。なぜなら、実行時例外が送出されるなどして、`ejbRemove()` が呼び出されないこともあり得るからです。

WebLogic Server でのメッセージ駆動型 Bean のデプロイ

メッセージ駆動型 Bean のデプロイ先は、初めて起動した場合の WebLogic Server、または実行中の WebLogic Server です。Bean のデプロイの詳細については、7-2 ページの「WebLogic Server 起動時の EJB のデプロイメント」または 7-3 ページの「動作中の WebLogic Server への EJB のデプロイ」を参照してください。

メッセージ駆動型 Bean でのトランザクション サービスの使用

その他の型の EJB と同じく、メッセージ駆動型 Bean では Bean 管理のトランザクションを使用して独自のトランザクション境界を設定することも、WebLogic Server のコンテナにトランザクションを管理させる (コンテナ管理のトランザクション) こともできます。どちらの場合でも、メッセージ駆動型 Bean が、メッセージを送信するクライアントからトランザクション コンテキストを受け取る

ことはありません。WebLogic Server では常に、Bean のデプロイメント記述子ファイルで指定されたトランザクション コンテキストを使用して Bean の `onMessage()` メソッドを呼び出します。

どのクライアントでもメッセージ駆動型 Bean に対する呼び出しのためのトランザクション コンテキストは提供されないので、コンテナ管理のトランザクションを使用する Bean は `ejb-jar.xml` ファイルの `container-transaction` 要素で `Required` または `NotSupported` `trans-attribute` を指定してデプロイする必要があります。

`ejb-jar.xml` ファイルの次の例は、メッセージ駆動型 Bean のトランザクション コンテキストの指定方法を示しています。

図 3-4　ejb-jar.xml ファイルの XML スタンザの例

```
<assembly-descriptor>
       <container-transaction>
              <method>

<ejb-name>MyMessageDrivenBeanQueueTx</ejb-name>
                     <method-name>*</method-name>
              </method>
       <trans-attribute>NotSupported</trans-attribute>
       </container-transaction>
</assembly-descriptor>
```

メッセージの受信

EJB の `onMessage()` メソッドを呼び出すきっかけとなる JMS メッセージの受信は、通常はトランザクションのスコープには含まれません。ただし、Bean 管理のトランザクションとコンテナ管理のトランザクションに関しては別の方法で処理されます。

- Bean 管理のトランザクションを使用する EJB の場合、メッセージの受信は常に Bean のトランザクションのスコープ外になる。
- コンテナ管理によるトランザクションの境界設定を使用する EJB については、`ejb-jar.xml` ファイルで Bean の `transaction-type` 要素が `Required`

に設定されている場合に限り、WebLogic Server ではメッセージの受信が Bean のトランザクションの一部となる。

メッセージの確認応答

コンテナ管理によるトランザクションの境界設定を使用するメッセージ駆動型 Bean の場合は、EJB トランザクションがコミットされたときに WebLogic Server で自動的にメッセージの確認応答が行われます。EJB で Bean 管理のトランザクションが使用される場合、メッセージの受信と確認応答は両方とも EJB トランザクション コンテキストの外側で行われます。Bean 管理のトランザクションを使用する EJB では WebLogic Server によって自動的にメッセージの確認応答が行われますが、`ejb-jar.xml` ファイルで定義される `acknowledge-mode` デプロイメント記述子要素を使用して確認応答のセマンティクスをコンフィグレーションできます。

メッセージ駆動型 Bean の移行サービス

WebLogic Server では、メッセージ駆動型 Bean の移行と回復のサービスをサポートします。移行と回復のサービスを提供するため、WebLogic JMS では WebLogic Server が提供する移行フレームワークに則って移行の要求に応え、障害発生後に JMS サーバをオンラインに戻します。JMS サーバを使用可能なサーバに移行する際には、関連付けられているメッセージ駆動型 Bean を障害サーバから同じ WebLogic Server クラスタ内の使用可能なサーバに手動で移行する必要があります。メッセージ駆動型 Bean はクラスタ化されたサーバ間でのみ移行サービスを利用できます。移行サービスは複数のクラスタ間をまたいで利用できません。

WebLogic Server が JMS サーバとともにメッセージ駆動型 Bean をクラスタ内の使用可能なサーバに移行しないとすれば、JMS 送り先にメッセージが溢れることになります。元のサーバが回復するまでの間、メッセージ駆動型 Bean を円滑に回復するために、メッセージ駆動型 Bean ではそれ自身が移行可能なことをマークし、WebLogic Server は移行サービス プロセスを実装します。Bean を他のサーバに移行した後、そのサーバは JMS サーバに接続し、障害サーバの代わりに JMS 送り先からメッセージのプルを再開します。

メッセージ駆動型 Bean の移行サービスの有効化

メッセージ駆動型 Bean の移行サービスを有効化するには、次の手順に従います。

1. 3-4 ページの「メッセージ駆動型 Bean の開発とコンフィグレーション」の手順に従って、メッセージ駆動型 Bean をコンフィグレーションします。
2. `ejb-jar.xml` ファイルの `destination-type` 要素を設定することによって、メッセージ駆動型 Bean の JMS 送り先のタイプを指定します。これについては、「分散送り先のコンフィグレーション」を参照してください
3. JMS 送り先のデプロイメント スキーマを次のいずれかに指定します。
   - 単純な送り先 - JMS 送り先は分散しておらず、EJB コンテナはメッセージ駆動型 Bean を JMS 送り先とともにデプロイします。
   - 分散送り先 - EJB コンテナは、メッセージ駆動型 Bean を JMS 送り先が分散しているすべてのサーバ上に JMS 送り先とともにデプロイします。

   この手順については、「分散送り先のコンフィグレーション」を参照してください。
4. WebLogic Server Administration Console を使用し、JMS サーバをコンフィグレーションします。手順については、「WebLogic JMS の管理」を参照してください。

   JMS サーバが WebLogic Server クラスタ内の あるサーバ上にデプロイされ、一連の JMS 送り先に対する要求を処理します。
5. その JMS サーバの移行できる対象をコンフィグレーションします。この手順については、「JMS の移行できる対象のコンフィグレーション」を参照してください。

メッセージ駆動型 Bean の移行

WebLogic Server クラスタ内において、障害サーバから使用可能なサーバにメッセージ駆動型 Bean を移行するには、次の手順に従います。

1. WebLogic Server Administration Console を起動します。

2. JMS 送り先のデプロイメント スキーマを次のいずれかに指定します。
   - 単純な送り先 - JMS 送り先は分散しておらず、EJB コンテナはメッセージ駆動型 Bean を JMS 送り先とともにデプロイします。
   - 分散送り先 - EJB コンテナは、メッセージ駆動型 Bean を JMS 送り先が分散しているすべてのサーバ上に JMS 送り先とともにデプロイします。

メッセージ駆動型 Bean は JMS サーバの移行先を調べることができるので、メッセージ駆動型 Bean に対して移行先を変更する必要はありません。

しかし、メッセージ駆動型 Bean はクラスタ内、あるいは、JMS サーバの移行先リストにあるすべてのサーバ上にデプロイされる必要があります。メッセージ駆動型 Bean は移行の間、有効でないためです。メッセージ駆動型 Bean は移行先リストに含まれるすべてのサーバ上に JMS 送り先とともにデプロイされ、JMS 送り先が非アクティブな間は非アクティブなままです。

WebLogic Server は、メッセージ駆動型 Bean をアクティブにした時点で JMS サーバを探索し、その Bean に指定された JMS 送り先からメッセージのプルを開始します。

非 BEA JMS プロバイダのメッセージ駆動型 Bean のコンフィグレーション

IBM MQSeries などの非 BEA JMS プロバイダで機能するように、メッセージ駆動型 Bean をコンフィグレーションすることができます。WebLogic Server 7.0 以降、Bean 管理によるトランザクションをサポートする MDB (トランザクション非対応 MDB) に加えて、コンテナ管理によるトランザクションをサポートする MDB (トランザクション対応 MDB) に関してこのコンフィグレーションが可能です。

つまり、トランザクション MDB のあるアプリケーションは、MDB によって処理されるメッセージに関して非 BEA JMS プロバイダとの間で「かならず 1 回」のセマンティクスを実現できます。WebLogic Server は XA を使用して、非 BEA JMS プロバイダをトランザクションの中に自動的に取得します。

トランザクション非対応 MDB のあるアプリケーションの場合、MDB は少なくとも 1 回のメッセージ処理セマンティクスを提供し、XA は必要ありません。

非 BEA JMS プロバイダが XA をサポートしていない場合、そのプロバイダではコンテナ管理によるトランザクションをサポートする MDB をデプロイすることはできません。また、JMS プロバイダが XA をサポートしている場合は、`weblogic-ejb-jar.xml` ファイル内で指定する JMS 接続ファクトリが XA をサポートするということを保証する必要があります。この指定方法は、各 JMS プロバイダでさまざまです。

トランザクション対応 MDB の指定

MDB をトランザクション対応として指定するには、次の手順に従います。

- `ejb-jar.xml` ファイルの `message-driven` 要素内の `transaction-type` 要素を `Container` に設定する。
- `ejb-jar.xml` ファイルの `container-transaction` 要素内の `trans-attribute` 要素を `Required` に設定する。

また、非 BEA JMS プロバイダに合わせて `weblogic-ejb-jar.xml` ファイルの destination-jndi-name、initial-context-factory、provider-url、および connection-factory-jndi-name 要素を適切に設定します。

トランザクション対応 MDB の場合、connection-factory-jndi-name で指定された JMS 接続ファクトリが JMS に対するオプションの XA 拡張をサポートしている必要があります。

トランザクション非対応 MDB の指定

MDB をトランザクション非対応として指定するには、次の手順に従います。

- `ejb-jar.xml` ファイルの `message-driven` 要素内の `transaction-type` 要素を `Bean` に設定する。
- `ejb-jar.xml` ファイルの `container-transaction` 要素内の `trans-attribute` 要素を `NotRequired` に設定する。

非 BEA プロバイダを使用するための MDB のコンフィグレーションの方法の例については、ホワイト ペーパー「Using Foreign JMS Providers with WLS Message Driven Beans」(`jmsmdb.pdf`) を参照してください。

JMS サーバまたは非 BEA サービス プロバイダへの再接続

メッセージ駆動型 Bean は、非クラスタ WebLogic Server インスタンスまたは非 BEA サービス プロバイダにデプロイされている JMS サーバの JMS 送り先をリスンします。サーバがダウンしたためにその送り先との接続が失われた場合、メッセージ駆動型 Bean は一定の間隔で送り先との再接続を試みます。この間隔は、Bean の `weblogic-ejb-jar.xml` ファイルの `jms-polling-interval-seconds` 要素を設定して秒単位で指定します。

JMS 分散送り先でリスンするための MDB のコンフィグレーション

WebLogic JMS では、1 つの分散送り先セットのメンバーとして複数の物理的送り先 (キューおよびトピック) をコンフィグレーションできるため、クラスタ内の WebLogic Server インスタンスで障害が発生してもサービスを継続することができます。このようにコンフィグレーションすると、プロデューサおよびコンシューマは、1 つの送り先のように見える送り先を介してメッセージを送受信します。

しかし実際には、メッセージングの負荷が分散送り先内のすべての送り先メンバーに分散されます。サーバの障害などによって使用できない送り先メンバーがある場合は、セット内の他の送り先メンバーにトラフィックがリダイレクトされます。

このリリースから、MDB がクラスタ内のサーバにデプロイされると、WebLogic Server は分散送り先のメンバーを自動的に列挙し、MDB が各メンバーで必ずリスンするようにします。

MDB がクラスタ内のサーバにデプロイされると、WebLogic Server は分散送り先のメンバーを自動的に列挙し、MDB が各メンバーで必ずリスンするようにします。

次の手順に従って、分散送り先に対してメッセージ駆動型 Bean をコンフィグレーションします。

1. 「分散送り先のコンフィグレーション」の説明に従って、メッセージ駆動型 Bean をコンフィグレーションします。
2. `weblogic-ejb-jar.xml` で MDB の destination-jndi-name を、分散トピックまたはキューを JNDI ネームスペースにバインドするための名前に設定します。
3. MDB の対象を分散送り先の対象と同じものにします。分散送り先が存在する場合は、MDB をデプロイする必要があります。
4. MDB をデプロイします。

デプロイ時に、MDB は WebLogic Server インスタンス上に存在する分散送り先のメンバーを検出し、自身をそのメンバーに固定し、メッセージの処理を開始します。

メッセージ駆動型 Bean のセキュリティ ID のコンフィグレーション

メッセージ駆動型 Bean (MDB) が JMS キューまたはトピックからメッセージを受信する場合、EJB コンテナは資格マッピング プロバイダおよび資格マップを使用して、JMS 接続を確立するときに使用するセキュリティ ID (ユーザ名とパスワード) を取得します。資格マッピングは、MDB 開始時の 1 回だけ行われます。EJB コンテナが接続すると、JMS プロバイダはセキュリティ ID を使用して、すべてのメッセージを取得します。セキュリティ ID は、MDB を使用して非

BEA JMS プロバイダ (別のベンダの JMS プロバイダまたは別の WebLogic Server ドメイン内で動作している WebLogic Server JMS プロバイダ) からメッセージを受信するときに特に重要です。

MDB のセキュリティ ID をコンフィグレーションするには、次の手順に従います。

1. MDB 用の WebLogic ユーザを作成します。詳細については、『WebLogic リソースのセキュリティ』の「ユーザとグループ」を参照してください。この WebLogic ユーザには、非 BEA JMS プロバイダが JMS 接続を確立するために必要なユーザ名およびパスワードを指定します。
2. WebLogic Server Administration Console の左ペインで、[デプロイメント] を展開します。
3. [デプロイメント] ノードを展開すると、デプロイ可能な WebLogic リソースのタイプが表示されます。
4. 資格マップを作成する EJB リソース (この場合は MDB) を右クリックします。
5. [個別の Bean のポリシーとロールを定義 ...] オプションを選択します。
6. 資格マップを作成する MDB に対応する [資格マップを定義] リンクをクリックします。
7. 手順 1 で MDB 用として定義した WebLogic Server ユーザ名およびパスワードを [WLS ユーザ] フィールドに入力します。
8. [適用] をクリックして変更を保存します。

4 WebLogic Server EJB コンテナとサポートされるサービス

以下の節では、WebLogic Server の EJB コンテナについて説明します。また、EJB の動作のさまざまな側面について、コンテナが提供する機能およびサービスとの関連から説明します。コンテナ管理による永続性 (CMP) の詳細については、第 5 章「WebLogic Server のコンテナ管理による永続性サービス」を参照してください。

- EJB コンテナ
- EJB のライフサイクル
- エンティティ EJB に対する ejbLoad() と ejbStore() の動作
- EJB の同時方式
- エンティティ Bean の組み合わせキャッシング
- トランザクション間のキャッシング
- WebLogic Server クラスタにおける EJB
- トランザクション管理
- データベースの挿入サポート
- リソース ファクトリ

EJB コンテナ

EJB コンテナは、デプロイ済み EJB を収容する実行時コンテナです。WebLogic Server の起動時に自動的に作成されます。EJB オブジェクトは、作成から削除まで、ライフサイクル全体に渡ってこのコンテナ内で動作します。EJB コンテナ

は、キャッシュ、同時実行性、永続性、セキュリティ、トランザクション管理、ロック、環境、メモリ レプリケーション、コンテナ内の EJB オブジェクトのクラスタ化などの標準のサービス集合を提供します。

1 つのコンテナに複数の Bean をデプロイできます。コンテナにデプロイされる個々のセッション Bean およびエンティティ Bean に対して、コンテナはホーム インタフェースを提供します。クライアントはホーム インタフェースを通じて、エンティティ Bean に属するエンティティ オブジェクトを作成、検索、および削除したり、特定のエンティティ Bean オブジェクトに固有でないホームのビジネス メソッドを実行することができます。クライアントは Java Naming and Directory Interface (JNDI) を使用して、または EJB 参照に従って (こちらのほうがより一般的)、エンティティ Bean のホーム インタフェースをルックアップできます。JNDI のネームスペース内でエンティティ Bean のホーム インタフェースを見つけられるようにするのはコンテナの役割です。JNDI を通じたホーム インタフェースのルックアップについて、詳しくは『WebLogic JNDI プログラマーズ ガイド』を参照してください。

EJB のライフサイクル

以降の節では、コンテナによるキャッシュ サービスのサポートに関する情報を示します。また、WebLogic Server での EJB インスタンスのライフサイクルを、WebLogic Server の視点から説明します。これらの節では EJB インスタンスという用語を使用しますが、これは EJB Bean クラスの実際のインスタンスという意味です。EJB インスタンスは、クライアント側から見た EJB の論理インスタンスを意味するものではありません。

エンティティ Bean のライフサイクルとキャッシュおよびプール

WebLogic Server は、エンティティ EJB のパフォーマンスとスループットを向上させるために、次の機能を提供します。

- フリー プール — ファインダやホーム メソッドを呼び出す場合や、エンティティ Bean を作成する場合に使用される匿名エンティティ Bean を格納します。
- キャッシュ — ID (主キー) を持つインスタンス、または現在トランザクションに関与しているインスタンス (READY および ACTIVE エンティティ EJB インスタンス) を格納します。

以降の節では、エンティティ Bean インスタンスのライフサイクル、コンテナがフリー プールおよびキャッシュを取得および管理する方法について説明します。図 4-2 を参照してください。

エンティティ EJB インスタンスの初期化 (フリー プール)

`initial-beans-in-free-pool` にゼロ以外の値を指定すると、WebLogic Server では、起動時に、指定した数の Bean インスタンスがプール内に生成されます。

`initial-beans-in-free-pool` のデフォルト値はゼロです。起動時に Bean インスタンスをフリー プールに格納しておくと、Bean に対する最初のリクエストが来たときに新しいインスタンスを生成せずに処理できるため、EJB の初期応答時間が短縮されます。

フリー プールからの Bean インスタンスの取得は、プールが空の場合でも常に成功します。プールが空の場合は、新しい Bean インスタンスが作成されて返されます。

POOLED Bean は匿名のインスタンスで、ファインダやホーム メソッドに使用されます。プールに格納できるインスタンスの最大数は、`weblogic-ejb-jar.xml` の `max-beans-in-free-pool` 要素の値で指定されます。

READY および ACTIVE エンティティ EJB インスタンス (キャッシュ)

Bean のビジネス メソッドが呼び出されると、コンテナはプールからインスタンスを取得して `ejbActivate` を呼び出し、インスタンスはメソッド呼び出しを処理します。

READY インスタンスはキャッシュ内にあり、ID (関連付けられた主キー) を持っていますが、現在トランザクションに関与していません。WebLogic では、READY エンティティ EJB インスタンスを、最長時間未使用 (LRU) の順序で保持します。モニタ タブの [Current Beans in Cache] フィールドには、アクティブなまたは準備が整っている Bean の数が表示されます。

ACTIVE インスタンスは現在トランザクションに関与しています。トランザクションが完了すると、インスタンスは READY になり、他の Bean でスペースが必要になるまでキャッシュに保持されます。

`max-beans-in-cache` の影響と、キャッシュで許可される主キーが同じインスタンスの数は、次節「キャッシュのルールは同時方式によって異なる」で説明されているように同時方式によって異なります。

キャッシュのルールは同時方式によって異なる

表 4-1 では、同時方式ごとに以下のことを示しています。

- `weblogic-ejb-jar.xml` の `max-beans-in-cache` 要素の値が、キャッシュ内のエンティティ Bean インスタンスの数をどのように制限するのか
- 主キーが同じエンティティ Bean インスタンスがどのくらいキャッシュで許可されるのか

図 4-1　同時方式別のエンティティ EJB のキャッシング動作

同時方式オプション	キャッシュ内の Bean インスタンスの数に対する max-beans-in-cache の影響	キャッシュ内に同時に存在できる、主キーが同じインスタンスの数
Exclusive	max-beans-in-cache = ACTIVE Bean の数 + READY インスタンスの数	1
Database	キャッシュでは max-beans-in-cache までの ACTIVE Bean インスタンスと、max-beans-in-cache までの READY Bean インスタンスを格納できる	複数
ReadOnly	max-beans-in-cache = ACTIVE Bean の数 + READY インスタンスの数	1

キャッシュからの Bean の削除

READY エンティティ EJB インスタンスは、他の Bean でスペースが必要になるとキャッシュから削除されます。READY インスタンスがキャッシュから削除される場合、Bean の ejbPassivate が呼び出されて、コンテナは Bean をフリープールに戻そうとします。

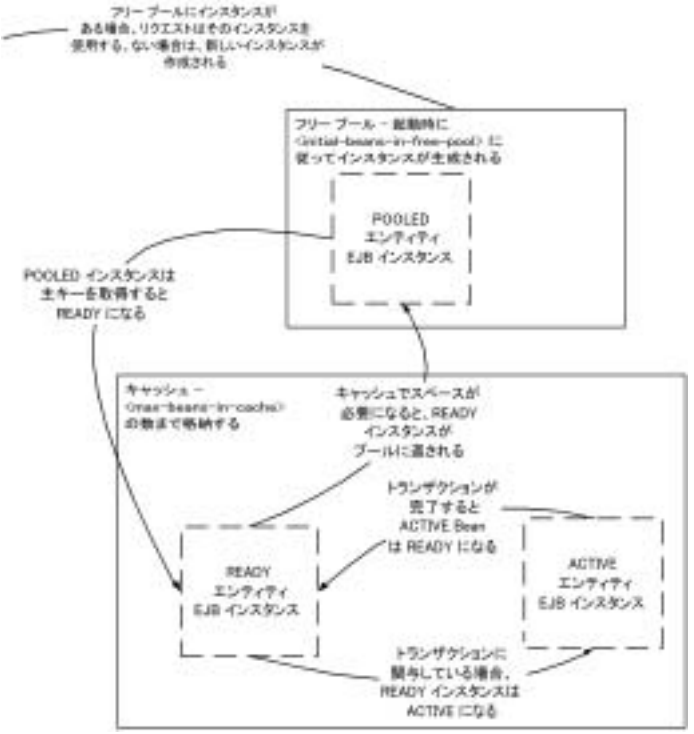
コンテナがインスタンスをフリー プールに返そうとするときに、プール内に max-beans-in-free-pool の数のインスタンスが既にある場合、インスタンスは破棄されます。

ACTIVE エンティティ EJB インスタンスは、参加しているトランザクションがコミットまたはロールバックするまで、キャッシュから削除されません。トランザクションがコミットまたはロールバックすると、インスタンスは READY になり、キャッシュから削除できるようになります。

エンティティ EJB のライフサイクルの遷移

図 4-2 では、EJB のフリー プールとキャッシュ、およびエンティティ Bean インスタンスのライフサイクルを通じて発生する遷移について示します。

図 4-2 エンティティ Bean のライフサイクル

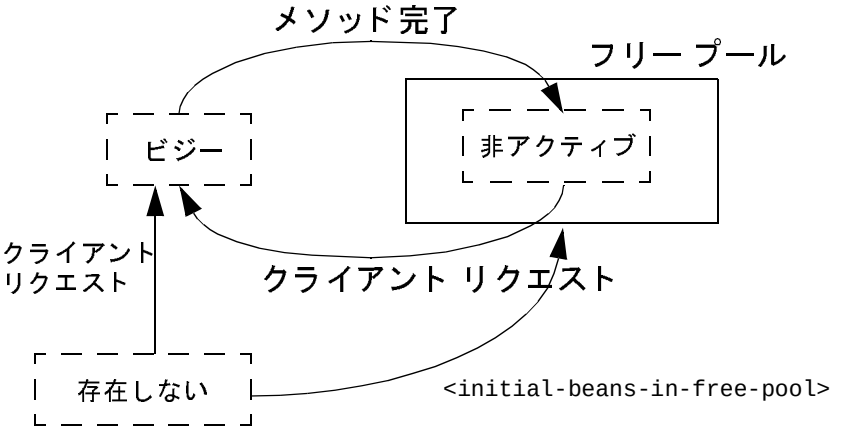


ステートレス セッション EJB のライフサイクル

WebLogic Server は、フリー プールを使用してステートレス セッション EJB のパフォーマンスとスループットを高めます。フリー プールには、非バインド ステートレス セッション EJB が格納されます。非バインド EJB インスタンスはステートレス セッション EJB クラスのインスタンスで、メソッド呼び出しを処理しません。

次の図に、WebLogic Server のフリー プールと、ステートレス EJB がフリー プールに出入りするプロセスを示します。点線は、WebLogic Server 側から見た EJB の「状態」を表しています。

図 4-3 ステートレス セッション EJB のライフサイクルを示す WebLogic Server フリー プール



ステートレス セッション EJB インスタンスを初期化する

デフォルトでは、WebLogic Server には起動時にステートレス セッション EJB インスタンスは存在しません。クライアントが個々の Bean にアクセスすると、WebLogic Server は EJB の新しいインスタンスを初期化します。ただし、WebLogic Server の起動時に非アクティブな EJB インスタンスを作成する場合、`initial-beans-in-free-pool` デプロイメント記述子要素を `weblogic-ejb-jar.xml` ファイルで指定します。

このようにしておけば、クライアントが EJB にアクセスしたときの初期応答時間を短縮できます。これは、Bean を初期化してからアクティブ化するのではなく、フリー プールから Bean をアクティブ化することによって、最初のクライアント リクエストを満足させることができるからです。デフォルトでは、`initial-beans-in-free-pool` は 0 に設定されています。

注意： フリー プールの最大サイズは、`max-beans-in-free-pool` デプロイメント要素の値、使用可能メモリ、または実行スレッドの数によって制限されます。

ステートレス セッション EJB をアクティブ化およびプールする

ステートレス セッション EJB に対するメソッドを呼び出すと、WebLogic Server はフリー プールからインスタンスを取得します。EJB は、クライアントのメソッド呼び出しの間アクティブ状態になります。メソッドが完了すると、EJB のインスタンスはフリー プールに戻されます。WebLogic Server は各メソッドの呼び出し後にクライアントからステートレス セッション Bean を非バインドするので、クライアントが使用する実際の Bean クラスは呼び出しごとに異なります。

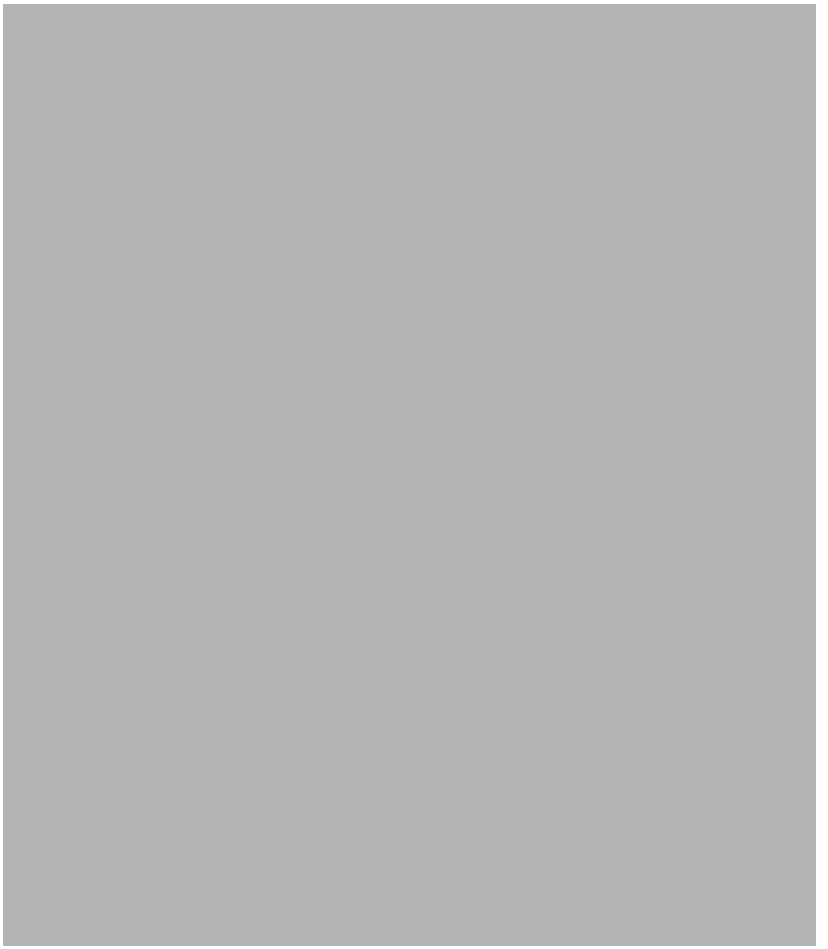
EJB クラスのすべてのインスタンスがアクティブで、`max-beans-in-free-pool` に達した場合、EJB クラスを要求する新しいクライアントは、アクティブ EJB がメソッド呼び出しを完了するまでブロックされます。トランザクションがタイムアウトになった場合 (トランザクション非対応の呼び出しでは、5 分経過した場合)、WebLogic Server は、リモート クライアントに対しては `RemoteException` を、ローカル クライアントに対しては `EJBException` を送出します。

ステートフル セッション EJB のライフサイクル

WebLogic Server は、Bean インスタンスのキャッシュを使用してステートフル セッション EJB のパフォーマンスを高めます。キャッシュはメモリ内にアクティブな EJB インスタンスを格納することで、それらをクライアント リクエストに即座に使用できるようにしています。キャッシュには、クライアントによって現在使用されている EJB と最近使用されたインスタンスが格納されます。キャッシュ内のステートフル セッション Bean は、特定のクライアントにバインドされます。

次の図に、WebLogic Server のキャッシュと、ステートフル EJB がキャッシュに出入りするプロセスを示します。

図 4-4　ステートフル セッション EJB のライフサイクル



ステートフル セッション EJB の作成

WebLogic Server には起動時にステートフル セッション EJB インスタンスは存在しません。クライアントがステートフル セッション Bean へのアクセスを開始する前に、Bean とのセッションで使用する新しい Bean インスタンスが作成されます。セッションが終了すると、インスタンスは破棄されます。セッションが進行中の間は、インスタンスはメモリにキャッシュされます。

ステートフル セッション EJB のパッシベーション

パッシベーションは、EJB の状態をディスク上で保持しつつキャッシュからその EJB インスタンスを削除するために WebLogic Server が使用するプロセスです。パッシブ化されている EJB はメモリには存在せず、キャッシュに格納されているときのようにクライアント リクエストに応じて即座に使用することはできません。

EJB 開発者は、`ejbPassivate()` メソッドが呼び出されたときに、ステートフル セッション Bean が、WebLogic Server によってデータがシリアライズされてインスタンスのパッシベーションが行われるような状態に置かれるようにしなければなりません。パッシベーションの間、WebLogic Server は `transient` であると宣言されていないフィールドをシリアライズしようとします。このため、すべての非 `transient` フィールドがシリアライズ可能オブジェクト (Bean のリモートおよびホーム インタフェースなど) を表すようにしなければなりません。EJB 2.1 では、許可されるフィールド タイプが指定されています。

パッシベーションの制御

ステートフル セッション Bean のパッシベーションを制御するルールは、Bean の `cache-type` 要素の値によって決まります。この要素の値は以下のいずれかです。

- `LRU` ー 最長時間未使用 (積極的なパッシベーション)
- `NRU` ー 最近未使用 (怠惰なパッシベーション)

`idle-timeout-seconds` 要素と `max-beans-in-cache` 要素も、`cache-type` の値に基づいてパッシベーションと削除の動作に影響を与えます。

積極的なパッシベーション (LRU)

cache-type を LRU に設定してステートフル セッション Bean の積極的なパッシベーションをコンフィグレーションすると、コンテナは以下の動作を行います。

- 以下の状況で、インスタンスをディスクにパッシベーションする。
  - max-beans-in-cache の値に関係なく、インスタンスの非アクティブな状態が idle-timeout-seconds の期間を経過するとすぐ
  - idle-timeout-seconds が経過していなくても、max-beans-in-cache に達したとき
- パッシベーションされたインスタンスを、そのパッシベーション後、idle-timeout-seconds の期間非アクティブな状態が続いた後にディスクから削除する。

怠惰なパッシベーション (NRU)

cache-type を NRU に設定して怠惰なパッシベーションがコンフィグレーションされた場合、関連するシステムのオーバーヘッドが原因でコンテナは Bean のパッシベーションを回避します。コンテナは以下のように動作します。

- idle-timeout-seconds が経過したときにキャッシュから Bean のインスタンスを削除し、それをディスクにパッシベーションすることはしない。この動作は、積極的な削除と呼ばれます。積極的な削除を行うと、アクティブでないインスタンスによってメモリまたはディスクのリソースが消費されなくなります。
- max-beans-in-cache に達したときに、idle-timeout-seconds が経過していなくてもインスタンスをディスクにパッシベーションする。cache-type が NRU の場合は、max-beans-in-cache に達するのがパッシベーションの原因となる唯一のイベントです。

アイドル状態にある EJB の削除の防止

idle-timeout-seconds を 0 に設定すると、WebLogic Server は、ある期間、アイドル状態になっている EJB をもはや削除しません。しかし、キャッシュ リソースが不足状態になった場合、EJB のパッシベーションが行われる可能性は依然としてあります。

EJB のキャッシュ サイズの管理

キャッシュ サイズを管理してプロダクション環境のパフォーマンスを最適化する方法については、『WebLogic Server パフォーマンス チューニング ガイド』の「EJB キャッシュ サイズの設定」を参照してください。

パッシベーションされた Bean の永続ストア ディレクトリの指定

ステートフル セッション Bean のパッシベーションが行われると、その状態はファイル システム ディレクトリに格納されます。各サーバ インスタンスは、パッシベーションされたステートフル セッション Bean の状態を格納するためのディレクトリを持ちます。このディレクトリは、永続ストア ディレクトリと呼ばれます。永続ストア ディレクトリには、パッシベーションされた Bean ごとに 1 つのサブディレクトリが格納されます。

永続ストア ディレクトリは、たとえば次のようにサーバ インスタンス ディレクトリにデフォルトで作成されます。

```
D:\releases\700\bea\user_domains\mydomain\myserver\pstore\
```

永続ストアのパスは次のとおりです。

*RootDirectory**ServerName**persistent-store-dir*

各要素の説明は次のとおりです。

- *RootDirectory* − WebLogic Server が実行されるディレクトリ。次に例を示します。

  ```
  D:\releases\700\bea\user_domains\mydomain
  ```

  RootDirectory は、サーバの起動時に `-Dweblogic.RootDirectory` プロパティで指定できます。

- *ServerName* − サーバ インスタンスの名前。

- *persistent-store-dir* − `weblogic-ejb-jar.xml` の `<stateful-session-descriptor>` スタンザの `persistent-store-dir` 要素の値。`<persistent-store-dir>` で値を指定しないと、ディレクトリの名前はデフォルトで `pstore` になります。

永続ストア ディレクトリには、パッシベーションされた Bean ごとに、ハッシュコードで名前が付けられたサブディレクトリが格納されます。たとえば、上の例のパッシベーションされた Bean のサブディレクトリは次のようになります。

```
D:\releases\700\bea\user_domains\mydomain\myserver\pstore\14t89ge
x0m2fr
```

ステートフル セッション Bean への同時アクセス

EJB 2.0 仕様によると、ステートフル セッション EJB に同時アクセスすると `RemoteException` が送出されます。ステートフル セッション EJB に関するこのアクセス制限は、EJB クライアントが WebLogic Server の内部にあるのかそれともリモートにあるのかに関係なく適用されます。この制限を無効にして、同時呼び出しが可能なようにステートフル セッション Bean をコンフィグレーションするには、`allow-concurrent-calls` デプロイメント要素を設定します。

複数のサーブレット クラスがステートフル セッション EJB にアクセスする場合は、サーブレット クラスのインスタンスごとではなくサーブレット スレッドごとに独自のセッション EJB インスタンスを使用する必要があります。同時アクセスを避けるために、JSP またはサーブレットはリクエスト スコープでステートフル セッションを使用できます。

エンティティ EJB に対する ejbLoad() と ejbStore() の動作

WebLogic Server は、`ejbLoad()` および `ejbStore()` への呼び出しを使用して、エンティティ EJB の永続フィールドを読み書きします。デフォルトでは、WebLogic Server は `ejbLoad()` と `ejbStore()` を次の手順で呼び出します。

1. エンティティ EJB のトランザクションが開始されます。クライアントが明示的に新しいトランザクションを開始して Bean を呼び出すか、または WebLogic Server が Bean のメソッド トランザクション属性に従って新しいトランザクションを開始します。
2. WebLogic Server は `ejbLoad()` を呼び出して、Bean の永続データの最新バージョンを基盤となるデータストアから読み出します。
3. トランザクションがコミットすると、WebLogic Server は `ejbStore()` を呼び出して、永続フィールドを基盤となるデータベースに書き込みます。

こうした ejbLoad() と ejbStore() の呼び出しという単純なプロセスにより、新しいトランザクションは常に EJB の最新の永続的データを使用し、コミット時には常にそのデータをデータベースに書き込むようになります。しかし、環境によっては、パフォーマンス上の理由から ejbLoad() と ejbStore() の呼び出しを制限することができます。この場合、代わりに ejbStore() の呼び出しをより頻繁に行って、コミットされていないトランザクションの中間結果を参照することができます。

WebLogic Server には、ejbLoad() と ejbStore() の動作をコンフィグレーションするためのデプロイメント記述子要素が weblogic-ejb-jar.xml および weblogic-cmp-rdbms-jar.xml ファイルに用意されています。

is-modified-method-name を使用した ejbStore() の呼び出しの制限 (EJB 1.1 のみ)

is-modified-method-name デプロイメント記述子要素は、EJB 1.1 のコンテナ管理永続性 (CMP) Bean だけに適用されます。この要素は、weblogic-ejb-jar.xml ファイルに入っています。WebLogic Server の CMP 実装では、CMP フィールドの変更が自動的に検出され、それらの変更だけが基盤のデータストアに書き込まれます。Bean 管理の永続性 (BMP) では is-modified-method-name を使用しないことをお勧めします。なぜなら、is-modified-method-name 要素と ejbstore メソッドの両方を作成しなければならないからです。

デフォルトでは、WebLogic Server は各トランザクションが正常に完了 (コミット) したときに ejbStore() を呼び出します。ejbStore() は、EJB の永続フィールドが実際に更新されたかどうかに関係なく、コミット時に呼び出され、その結果、DBMS が更新されます。WebLogic Server の is-modified-method-name 要素は、ejbStore() の不必要な呼び出しによってパフォーマンスが低下するような場合に使用します。

is-modified-method-name を使用するには、EJB プロバイダは最初に、永続データが更新されたときに WebLogic Server に「合図」を送る EJB メソッドを開発する必要があります。このメソッドは、EJB フィールドが 1 つも更新されなかった場合は「false」を、フィールドが更新された場合は「true」を返さなければなりません。

EJB プロバイダまたは EJB デプロイメント記述子は、次に `is-modified-method-name` 要素の値を使用して、このメソッドの名前を識別します。WebLogic Server は、トランザクションがコミットされると指定されたメソッド名を呼び出し、そのメソッドが「true」を返した場合にだけ `ejbStore()` を呼び出します。この要素の詳細については、9-45 ページの「is-modified-method-name」を参照してください。

is-modified-method-name に関する警告

`is-modified-method-name` 要素を使用すると、`ejbStore()` の不必要な呼び出しを避けることによってパフォーマンスを高めることができます。しかし、いつ更新が行われたかを正確に識別することは、EJB 開発者にとっては負担が大きい作業です。指定された `is-modified-method-name` が不正確なフラグを WebLogic Server に返した場合、データの完全性に関する問題が起こりかねず、多くの場合、こうした問題を追跡するのは困難です。

エンティティ EJB の更新内容がシステム内で失われたと思われる場合、まずどのような状況でもすべての `is-modified-method-name` 要素の値が「true」を返すように設定してください。これにより、WebLogic Server のデフォルト `ejbStore()` 動作に戻すことができ、問題が解決する場合があります。

delay-updates-until-end-of-tx を使用した ejbStore() 動作の変更

デフォルトでは、WebLogic Server はトランザクションのすべての Bean の永続ストレージをトランザクションの完了 (コミット) 時にだけ格納します。これにより、不必要な更新と `ejbStore()` の呼び出しの反復が回避され、一般にパフォーマンスが向上します。

データベースがアイソレーション レベルの `READ_UNCOMMITTED` を使用する場合、他のデータベース ユーザが継続中のトランザクションの中間結果を参照できるようにすることができます。こうしたケースでは、トランザクションの完了時にだけデータストアを更新するという WebLogic Server のデフォルト動作は不適切な場合があります。

デフォルト動作を無効にするには、delay-updates-until-end-of-tx デプロイメント記述子要素を使用します。この要素は、weblogic-ejb-jar.xml ファイルで設定します。この要素を「false」に設定すると、WebLogic Server はトランザクションの完了時ではなく各メソッド呼び出しの後に ejbStore() を呼び出します。

注意： delay-updates-until-end-of-tx を false に設定しても、各メソッド呼び出しの後にデータベース更新が「コミットされた」状態になるわけではありません。更新はデータベースに送信されるだけです。更新は、トランザクションの完了時にのみデータベースにコミットまたはロールバックされます。

EJB の同時方式

同時方式は、EJB コンテナがエンティティ Bean への同時アクセスをどのように管理するかを指定するものです。Database オプションが WebLogic Server のデフォルトの同時方式ですが、その Bean でどのような同時アクセスが必要になるかによって、別のオプションを指定したくなるでしょう。WebLogic Server では次の同時方式オプションを提供しています。

同時方式オプション	説明
Exclusive	Bean がトランザクションに関連付けられている場合、キャッシュされたエンティティ EJB インスタンスを排他的にロックする。EJB インスタンスに対するリクエストは、トランザクションが完了するまでブロックされる。このオプションは、WebLogic Server バージョン 3.1 ～ 5.1 までのデフォルトのロック動作。
Database	エンティティ EJB に対するリクエストのロックを基盤のデータストアに委ねる。WebLogic Server は、独立したエンティティ Bean を割り当て、ロックとキャッシングをデータベースが処理できるようにする。現在のデフォルト オプション。

同時方式オプション	説明
Optimistic	トランザクションの実行中、EJB コンテナあるいはデータベースでロックを行わない。EJB コンテナはトランザクションをコミットする前に、そのトランザクションで更新したデータがどれも変化していないことを確認する。更新データのどれかが変わっていた場合、EJB コンテナはトランザクションをロールバックする。
ReadOnly	読み込み専用エンティティ Bean のみで使用される。トランザクションごとに新たなインスタンスをアクティブ化する。これにより、リクエストは並列処理される。WebLogic Server は、read-timeout-seconds パラメータを基に、ReadOnly Bean に対して ejbLoad() を呼び出す。

読み書き対応 EJB の同時方式

read-write EJB の場合、Exclusive、Database、および Optimistic 同時方式を使えます。WebLogic Server では、各トランザクションの最初に EJB データをキャッシュにロードするか、または、4-31 ページの「トランザクション間のキャッシュを使用した ejbStore() の呼び出しの制限」のようになります。トランザクションが正常にコミットしたとき、WebLogic Server は ejbStore() を呼び出すか、または、4-14 ページの「is-modified-method-name を使用した ejbStore() の呼び出しの制限 (EJB 1.1 のみ)」のようになります。

同時方式の指定

EJB で使用するロック メカニズムを指定するには、weblogic-ejb-jar.xml の concurrency-strategy デプロイメント パラメータを設定します。concurrency-strategy は個々の EJB レベルで設定するので、EJB コンテナ内でロック メカニズムが混在することがあります。

次のものは weblogic-ejb-jar.xml からの抜粋ですが、ある EJB に同時方式を設定する手順を示しています。この XML コード例では、デフォルトのロック メカニズム、Database を指定しています。

図 4-5 同時方式を指定する XML の例

```
<entity-descriptor>
       <entity-cache>
       ...
       <concurrency-strategy>Database</concurrency-strategy>
       </entity-cache>
       ...
</entity-descriptor>
```

concurrency-strategy を指定しない場合、WebLogic Server はエンティティ EJB インスタンスに対してデータベース ロックを実行します。

それぞれの同時方式について、以降の節で説明します。

Exclusive 同時方式

WebLogic Server 5.1 および 4.5.1 では、Exclusive 同時方式がデフォルトでした。このロック方式は、EJB データへのアクセスの信頼性を高め、ejbLoad() を不必要に呼び出して EJB インスタンスの永続フィールドをリフレッシュするのを防ぎます。しかし、排他的ロック メカニズムは、EJB のデータへの同時アクセスに対しては最良のモデルとはなりません。いったんクライアントが EJB インスタンスをロックすると、他のクライアントは、永続フィールドを読もうとしているだけであっても、EJB のデータからブロックされてしまうからです。

WebLogic Server の EJB コンテナは、エンティティ EJB インスタンスに対して排他的ロック メカニズムを使用します。クライアントが EJB または EJB メソッドをトランザクションに関与させると、WebLogic Server はそのトランザクションの間そのインスタンスを排他的にロックします。同じ EJB またはメソッドを要求する他のクライアントは、現在のトランザクションが完了するまでブロックされます。

Database 同時方式

Database 同時方式は WebLogic Server におけるデフォルトであり、EJB 1.1 と EJB 2.0 で推奨されているメカニズムです。データベース ロックによって、エンティティ EJB の同時アクセスの処理速度が向上します。WebLogic Server コンテナでは、ロック サービスを基盤となるデータベースに任せます。排他的ロックとは異なり、基盤データ ストアはより高い粒度で EJB データをロックでき、またデッドロックを検出することができます。

データベース ロック メカニズムでは、EJB コンテナが引き続きエンティティ EJB クラスのインスタンスをキャッシュします。しかし、コンテナはトランザクション間の EJB インスタンスの中間状態をキャッシュしません。代わりに、WebLogic Server は、トランザクションの開始時に各インスタンスに対して `ejbLoad()` を呼び出して、最新の EJB データを取得します。続いて、データのコミット リクエストがデータベースに送られます。このためデータベースは、EJB データのロック管理とデッドロック検出をすべて処理します。

基盤となるデータベースにロックを任せることで、エンティティ EJB データへの同時アクセスのスループットを上げつつ、デッドロックの検出も実行できます。しかし、データベース ロックを使用するには、基盤となるデータベースのロック方式に関するより詳細な知識が必要となります。この結果、EJB の異なるシステム間での移植性が低下する可能性があります。

`cache-between-transactions` 要素を「`true`」に設定し、Optimistic ではなく Database 同時方式を使う場合、コンパイラから `cache-between-transactions` を無効にすべきであることを伝える警告メッセージが返されます。この条件が揃っている場合、WebLogic Server は `cache-between-transactions` を自動的に無効にします。

Optimistic 同時方式

`Optimistic` 同時方式は、トランザクションの実行中、EJB コンテナまたはデータベースでどのようなロックも行いません。このオプションを指定した場合、EJB コンテナはあるトランザクションが更新したデータに変更がないことを確かめます。トランザクションのコミット前にフィールドをチェックするという手段によって、「効率的な更新」を実行します。

注意： Optimistic 同時方式の場合、EJB コンテナは Blob/Clob フィールドをチェックしません。これを回避するには、バージョン チェックまたはタイムスタンプ チェックを行います。

Optimistic 同時方式の制限

Optimistic 同時方式を使用する場合は、`weblogic-cmp-jar.xml` の `include-updates` 要素を `false` に設定することをお勧めします。`include-updates` を `true` に設定して Optimistic 同時方式を使用すると、ペシミ

スティックな同時方式を使用するのと同じことになり、効果がありません。include-updates を true に設定する必要がある場合は、Database 同時方式を使用してください。

include-updates を true に設定して Optimistic 同時方式を使用するのは、トランザクション中にロックするデータベース (Oracle 以外のデータベース) ではサポートされません。オプティミスティックなトランザクションでは、ローカルトランザクションを使用して読み込みを行い、トランザクションの終了までロックすることを避けるためです。ただし、オプティミスティックなトランザクションでは、必要に応じて更新をロールバックできるように、現在の JTA トランザクションを使用して書き込みを行います。一般に、JTA トランザクションで行われる更新は、その JTA トランザクションがコミットされるまで、読み込みトランザクションからは見えません。

Optimistic 同時方式におけるデータの有効性のチェック

トランザクションによるデータの読み込みや更新が、別のトランザクションによって変更されていないことを確認するために、トランザクションをコミットする前に Optimistic 同時方式の Bean のトランザクション データを検証するように、EJB コンテナをコンフィグレーションできます。変更されたデータを検出した場合、EJB コンテナはトランザクションをロールバックします。

注意： EJB コンテナは、Optimistic 同時方式の Bean では、Blob または Clob フィールドを検証しません。これを回避するには、バージョン チェックまたはタイムスタンプ チェックを行います。

オプティミスティックなチェックのコンフィグレーション

Optimistic 同時方式の Bean の妥当性チェックは、weblogic-cmp-jar.xml の table-name スタンザの verify-columns 要素を使用してコンフィグレーションします。

Optimistic 同時方式を使用する場合は、verify-columns 要素によって、テーブル カラムの妥当性がどのようにチェックされるのかが指定されます。

1. verify-columns 要素の値を以下のいずれかの値に指定します。

- Read— トランザクション中に読み込まれたテーブル内のすべてのカラムをチェックする場合。これには、トランザクションによって読み込まれるだけの行と、トランザクションによって読み込まれてから更新または削除される行の両方が含まれます。

- `Modified`— 現在のトランザクションで更新または削除されたカラムだけをチェックする場合。
- `Version`— テーブルにバージョン カラムが含まれ、このカラムが Optimistic 同時方式を実装するために使用されていることをチェックする場合。

  Version カラムは、初期値 0 で作成し、行が変更される度に 1 ずつ加算されるようにする。
- `Timestamp`— テーブルにタイムスタンプ カラムが含まれ、このカラムが Optimistic 同時方式を実装するために使用されていることをチェックする場合。タイムスタンプ ベースの Optimistic 同時方式では、データベース カラムに対する粒度を 1 秒にする必要があります。

EJB コンテナはバージョンまたはタイムスタンプ カラムを管理して、トランザクションの完了時にその値を適切に更新します。

注意： バージョン カラムまたはタイムスタンプ カラムは、トランザクションで通常の CMP または CMR フィールドが変更されなかった場合は更新されません。トランザクションで変更された唯一のデータがバージョンまたはタイムスタンプ カラムの値だった場合 (トランザクションの開始の結果として)、トランザクションの最後にオプティミスティックなチェックで使用するカラムは更新されません。

2. `verify-columns` が `Version` または `Timestamp` に設定されている場合は、`weblogic-cmp-jar.xml` ファイルの `table-map` スタンザの `optimistic-column` を使用してバージョンまたはタイムスタンプ カラムを指定します。このカラムを `cmp-field` にマップするかは任意です。

   `optimistic-column` 要素では、Optimistic 同時方式を実装するためのバージョン値またはタイムスタンプ値を格納するデータベース カラムを指定します。すべてのデータベースが大文字と小文字を区別するわけではありませんが、この要素では大文字と小文字を区別します。この要素の値は、`verify-columns` が `Version` または `Timestamp` に設定されている場合を除いて無視されます。

EJB が複数のテーブルにマップされる場合、トランザクション中に更新したテーブルに対してのみオプティミスティックなチェックが実行されます。

注意： デフォルトでは、オプティミスティックな Bean ではトランザクション間のキャッシングは有効でありません。明示的に有効にする必要があります。4-31 ページの「トランザクション間のキャッシュを使用した

ejbStore() の呼び出しの制限」を参照してください。ただし、複数のクライアントが EJB に同時にアクセスする場合、Optimistic 同時方式でのトランザクション間のキャッシングは危険です。あるクライアントが EJB を変更する場合に、他のクライアントによって使用されているコピーを無効にする方法がありません。

ReadOnly 同時方式

WebLogic Server では、読み込み専用エンティティ Bean に対する同時アクセスをサポートしています。この同時方式では、トランザクションごとに 1 つの読み込み専用エンティティ Bean インスタンスをアクティブ化することでリクエストを並行に処理できるようにします。

WebLogic Server 7.0 より前は、読み込み専用エンティティ Bean には排他的ロック同時方式が用いられていました。この方式では、Bean がトランザクションに関連付けられている場合、キャッシュされたエンティティ Bean インスタンスを排他的にロックします。このエンティティ Bean に対する他のリクエストは、トランザクションが完了するまでブロックされます。

データベースからの読み込みを避け、WebLogic Server では キャッシュ中の既存インスタンスから EJB 2.0 CMP Bean の状態をコピーします。このリリースでは、`ReadOnly` オプションが読み込み専用エンティティ Bean のデフォルトの同時方式です。

アプリケーションレベルまたはコンポーネントレベルにおいて、読み込み専用エンティティ Bean のキャッシングを指定することができます。

読み込み専用エンティティ Bean のキャッシングを有効にするには、次の手順に従います。

1. JAR ファイルまたは EAR ファイルに対し、`concurrency-strategy` デプロイメント記述子要素で `ReadOnly` オプションを指定します。
   - `weblogic-application.xml` ファイルで、(一連の EAR ファイルに相当する) アプリケーションレベルのキャッシュに対して concurrency-strategy 要素を指定します。
   - `weblogic-ejb-jar.xml` ファイルで、(一連の JAR ファイルに相当する) コンポーネントレベルのキャッシュに対して concurrency-strategy 要素を指定します。z

2. デプロイメント記述子を指定する手順については、6-6 ページの「EJB デプロイメント記述子の指定と編集」を参照してください。

読み込み専用エンティティ Bean と ReadOnly 同時方式

以前のバージョンの読み込み専用エンティティ Bean は、このバージョンの WebLogic Server で機能します。以前のバージョンと同じように、`weblogic-ejb-jar.xml` で `read-timeout-seconds` 要素を設定できます。EJB の同時方式が `ReadOnly` で、`read-timeout-seconds` が設定されている場合、読み込み専用 Bean が呼び出されると、キャッシュのデータが `read-timeout-seconds` の設定より古いかどうかが WebLogic Server によってチェックされます。古い場合は、Bean の `ejbLoad` が呼び出されます。それ以外の場合は、キャッシュされているデータが使用されます。

ReadOnly 同時方式の制限

`read-only` 同時方式を使用するエンティティ EJB は、以下の制限に従わなければなりません。

- WebLogic Server は読み込み専用エンティティ EJB に対しては `ejbStore()` を呼び出さないため、EJB データの更新を要求できない。
- EJB のメソッド呼び出しは、多重呼び出し不変でなければならない。詳細については、4-34 ページの「クラスタ内のステートレス セッション EJB」を参照してください。
- この Bean の基盤データは外部ソースによって更新されるため、`ejbLoad()` の呼び出しはデプロイメント パラメータの `read-timeout-seconds` によって制御される。

読み込み専用マルチキャストの無効化

読み込み専用マルチキャストを無効化すると、キャッシュされたデータを効率的に無効にできます。

読み込み専用エンティティ Bean を無効にするには、`CachingHome` または `CachingLocalHome` インタフェースの次の `invalidate()` メソッドを呼び出します。

図 4-6 CachingHome インタフェースと CachingLocalHome インタフェースを示すコード例

```
package weblogic.ejb;

public interface CachingHome {

    public void invalidate(Object pk) throws RemoteException;
    public void invalidate (Collection pks) throws RemoteException;
    public void invalidateAll() throws RemoteException;

public interface CachingLocalHome {

    public void invalidate(Object pk) throws RemoteException;
    public void invalidate (Collection pks) throws RemoteException;
    public void invalidateAll() throws RemoteException

}
```

次のコード例は、ホームを CachingHome にキャストしてからメソッドを呼び出す方法を示したものです。

図 4-7 ホームをキャストしてからメソッドを呼び出す方法を示すコード例

```
import javax.naming.InitialContext;
import weblogic.ejb.CachingHome;

Context initial = new InitialContext();
Object o = initial.lookup("CustomerEJB_CustomerHome");
CustomerHome customerHome = (CustomerHome)o;

CachingHome customerCaching = (CachingHome)customerHome;
customerCaching.invalidateAll();
```

`invalidate()` メソッドを呼び出すと、読み込み専用エンティティ Bean はローカル サーバで無効化され、マルチキャスト メッセージがクラスタ内の他のサーバに送信されてキャッシュされているその Bean のコピーが無効化されます。無効化された読み込み専用エンティティ Bean を次に呼び出すと、`ejbLoad` が呼び出されます。`ejbLoad()` は、最新の永続的データを基盤となるデータストアから読み出します。

WebLogic Server は、トランザクションの更新が完了してから `invalidate()` メソッドを呼び出します。トランザクションの更新中に無効化処理が行われた場合、アイソレーション レベルでコミットされていないデータの読み出しが許可されていないと、更新前のデータが読み出される可能性があります。

read-mostly パターン

WebLogic Server は、weblogic-ejb-jar.xml に設定される read-mostly キャッシュ方式をサポートしていません。しかし、頻繁に更新されない EJB データがある場合、read-only EJB と read-write EJB を組み合わせることによって、「read-mostly パターン」を作成できます。

詳細については、WebLogic Server の配布キットの次の場所にある read-mostly のサンプルを参照してください。

%SAMPLES_HOME%/server/config/examples/ejb/extensions/readMostly

WebLogic Server は、read-mostly パターンの自動 invalidate() メソッドを提供します。このパターンでは、読み込み専用エンティティ Bean と読み書き対応エンティティ Bean が同じデータにマップされます。データを読み出すには読み込み専用エンティティ Bean を使用し、データを更新するには読み書き対応エンティティ Bean を使用します。

read-mostly パターンでは、read-only エンティティ EJB は、weblogic-ejb-jar.xml ファイルの read-timeout-seconds デプロイメント記述子要素によって指定されている間隔で Bean データを取得します。別の read-write エンティティ EJB は、その read-only EJB と同じデータをモデル化し、必要な間隔でそのデータを更新します。

read-mostly パターンを作成する場合、以下の注意に従って、データの一貫性に関する問題が発生しないようにしてください。

- すべての read-only EJB について、同じトランザクションで更新されるすべての Bean の read-timeout-seconds を同じ値に設定する。
- すべての read-only EJB について、read-timeout-seconds を、許容できるパフォーマンス レベルを生み出す範囲で最も小さい値に設定する。
- システム内のすべての read-write EJB がデータの必要最小限の部分だけを更新するようにして、ejbStore() の呼び出しごとに Bean が多数の未変更フィールドをデータストアに書き込むことを回避する。
- すべての read-write EJB がデータをタイムリーに更新するようにする。これにより、read-write Bean が対応する read-only Bean の

read-timeout-seconds に及ぶ長時間のトランザクションに関与しなくなります。

EJB 2.0 を実行している場合は、Optimistic 同時方式を使用する単一の Bean を使用すると read-mostly パターンに近くなります。オプティミスティックな Bean は読み込みを行うときに read-only Bean のように動作します (キャッシュから読み込み、古いデータを返すことができます)。ただし、オプティミスティックな Bean が書き込みを行う場合、コンテナは、更新されるデータが変更されていないことを保証します。つまり、Database 同時方式を使用する Bean と同じレベルの一貫性を書き込みに対して提供します。4-19 ページの「Optimistic 同時方式」を参照してください。

注意： WebLogic Server クラスタでは、read-only EJB のクライアントは、キャッシュされた EJB データを使用することで恩恵を受けることができます。read-write EJB のクライアントは、真のトランザクション動作から恩恵を受けることができます。これは、read-write EJB の状態が常に基盤データベース上のそのデータの状態と一致するからです。詳細については、4-39 ページの「クラスタ内のエンティティ EJB」を参照してください。

エンティティ Bean の組み合わせキャッシング

組み合わせキャッシングを使用すると、同じ J2EE アプリケーション内の複数エンティティ Bean 間で 1 つの実行時キャッシュを共有できます。これまでは、アプリケーションの一部であるエンティティ Bean それぞれに対し、別々のキャッシュをコンフィグレーションする必要がありました。このことは、エンティティ Bean ごとにキャッシュをコンフィグレーションするのに時間がかかること、アプリケーションの実行により多くのメモリが必要になるなど、利便性やパフォーマンスの点で若干の問題を招いていました。この機能は、それらの問題解決に役立ちます。

アプリケーションレベルのキャッシュをコンフィグレーションするには、次の手順に従います。

1. weblogic-application.xml ファイルが EAR ファイルの META-INF ディレクトリ内にあるか確めます。

2. weblogic-application.xml ファイルに次のように入力します。

```
<weblogic-application>
 <ejb>
    <entity-cache>
        <entity-cache-name>large_account</entity-cache-name>
        <max-cache-size>
            <megabytes>1</megabytes>
        </max-cache-size>
    </entity-cache>
 </ejb>
</weblogic_application>
```

   entity-cache 要素を使用して、実行時にエンティティ Bean インスタンスをキャッシュする、名前付きのアプリケーションレベル キャッシュを定義します。各キャッシュを何個のエンティティ Bean が参照するかについて制限はありません。

   entity-cache のサブ要素は、weblogic-ejb-jar.xml デプロイメント記述子ファイルで使われるのと同じ基本的意味を持っています。

3. weblogic-ejb-jar.xml ファイルの entity-descriptor 要素を指定します。

   entity-descriptor 要素を使用し、エンティティ Bean がアプリケーションレベルのキャッシュを使用するようにコンフィグレーションします。

デプロイメント記述子を指定する手順については、6-6 ページの「EJB デプロイメント記述子の指定と編集」を参照してください。

weblogic-application.xml デプロイメント記述子については、『WebLogic Server アプリケーションの開発』の「application.xml デプロイメント記述子の要素」に詳細な説明があります。

トランザクション間のキャッシング

トランザクション間のキャッシング、または、長期キャッシングを使用すると、EJB コンテナはエンティティ Bean の永続データをトランザクション間でキャッシュできるようになります。エンティティ Bean に関してトランザクション間のキャッシングを設定できるかどうかは、以下の 3 つの表にまとめてあるように、同時方式によって決まります。

表 4-1 BMP Bean に関して各同時方式で許可される cache-between-transactions の値

BMP Bean が使用する同時方式	設定可能な cache-between-transactions の値
Database	False のみ
Exclusive	True または False
Read-Only	True または False
Optimistic	該当なし。Optimistic 同時方式は BMP Bean では使用できない。

表 4-2 CMP 2.0 Bean に関して各同時方式で許可される cache-between-transactions の値

CMP 2.0 Bean が使用する同時方式	設定可能な cache-between-transactions の値
Database	True のみ
Exclusive	True または False
Read-Only	True または False
Optimistic	True または False

表 4-3 CMP 1.1 Bean に関して各同時方式で許可される cache-between-transactions の値

CMP 1.1 Bean が使用する同時方式	設定可能な cache-between-transactions の値
Database	True のみ
Exclusive	True または False
Read-Only	True または False
Optimistic	該当なし。Optimistic 同時方式は CMP 1.1 Bean では使用できない。

Exclusive 同時方式でのトランザクション間のキャッシング

Exclusive 同時方式のエンティティ Bean で長期キャッシングを有効にする場合、EJB コンテナは基盤データへの更新アクセスを排他的にする必要があります。これはつまり、その EJB コンテナ以外の別のアプリケーションがデータを更新してはならないということです。あるクラスタ内に Exclusive 同時方式の EJB をデプロイする場合、長期キャッシングは自動的に無効になります。クラスタ内のどのノードからもデータを更新できるからです。これにより長期キャッシングは実行不能になります。

WebLogic Server の旧バージョンでは、この機能は weblogic-ejb-jar.xml の db-is-shared 要素によって制御されていました。

注意： Exclusive 同時方式は単一サーバでの機能です。クラスタ化されたサーバでは使用しないでください。

ReadOnly 同時方式でのトランザクション間のキャッシング

ReadOnly 同時方式のエンティティ Bean で長期キャッシングを無効にした場合、EJB コンテナでは読み込み専用データの長期キャッシングを常に実行するので、cache-between-transactions の設定値は無視されます。

Optimistic 同時方式でのトランザクション間のキャッシング

Optimistic 同時方式のエンティティ Bean で長期キャッシングを有効にした場合、EJB コンテナは前回のトランザクションでキャッシュされた値を再利用します。コンテナは、トランザクションの最後にオプティミスティックに競合をチェックすることによって、更新におけるトランザクションの一貫性を保証します。オプティミスティックなチェックを設定する手順については、4-19 ページの「Optimistic 同時方式」を参照してください。

また、オプティミスティックな競合を回避できるように、オプティミスティックなデータ更新に関する通知が他のクラスタ メンバーにブロードキャストされます。

トランザクション間のキャッシングの有効化

トランザクション間のキャッシングを有効にするには、次の手順に従います。

1. 次のオプションの 1 つを選択し、weblogic-ejb-jar.xml ファイルの cache-between-transactions 要素を設定します。
   - EJB コンテナによるデータの長期キャッシングの実行を有効にする場合、True を指定する。
   - EJB コンテナによるデータの短期キャッシングの実行を有効にする場合、False を指定します。これはデフォルト設定です。
2. デプロイメント記述子を指定する手順については、6-6 ページの「EJB デプロイメント記述子の指定と編集」を参照してください。

トランザクション間のキャッシュを使用した ejbStore() の呼び出しの制限

デフォルトによって、WebLogic Server は各トランザクションの開始時に `ejbLoad()` を呼び出します。この動作は、複数のソースがデータベースを更新するような環境に適しています。複数のクライアント (WebLogic Server を含む) が EJB の基盤データを変更する可能性があるため、`ejbLoad()` の最初の呼び出しは、キャッシュ済みデータを更新する必要があることを Bean に通知し、確実に最新のデータが処理対象になるようにします。

クラスタではなく単一のサーバで Exclusive 同時方式を使用する場合のように、単一の WebLogic Server トランザクションのみが特定の EJB に同時アクセスするような特殊な環境では、`ejbLoad()` をデフォルトで呼び出す必要はありません。EJB の基盤データを更新するクライアントまたはシステムは他に存在しないので、WebLogic Server の EJB データのキャッシュは常に最新のものとなります。こうしたケースでは、`ejbLoad()` を呼び出しても、Bean にアクセスする WebLogic Server クライアントに対して余計なオーバーヘッドが発生するだけです。

単一の WebLogic Server トランザクションが特定の EJB にアクセスする場合、`ejbLoad()` の不要な呼び出しを避けるために、WebLogic Server には `cache-between-transactions` デプロイメント パラメータが用意されています。デフォルトでは、`cache-between-transactions` は各 EJB に対して Bean の `weblogic-ejb-jar.xml` ファイルで「false」に設定されています。このため、`ejbLoad()` は各トランザクションの開始時に必ず呼び出されます。単一の WebLogic Server トランザクションだけが EJB の基盤データにアクセスするようなケースでは、その Bean の `weblogic-ejb-jar.xml` ファイルの `d` を「true」に設定できます。`cache-between-transactions` を「true」に設定して EJB をデプロイすると、WebLogic Server の単一のインスタンスが以下の場合に限り、その Bean の `ejbLoad()` を呼び出します。

- クライアントが最初に EJB を参照する場合
- EJB のトランザクションがロールバックされた場合

cache-between-transactions に関する制限

cache-between-transactions には以下の制限があります。

単一サーバ デプロイメントでは、cache-between-transactions は、Exclusive、Optimistic、および Read-Only 同時方式でのみ有効にできます。Database 同時方式では cache-between-transactions を使用できません。

クラスタ デプロイメントでは、`cache-between-transactions` は、Optimistic および Read-Only 同時方式でのみ有効にできます。Exclusive または Database 同時方式では cache-between-transactions を使用できません。

WebLogic Server クラスタにおける EJB

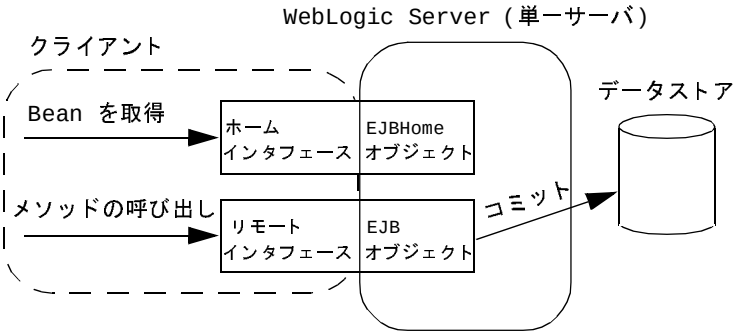
この節では EJB のクラスタ化のサポートについて説明します。

クラスタ化されたホームおよび EJBObject

WebLogic Server クラスタ内の EJB は、ホーム オブジェクトと EJB オブジェクトという 2 つの主要構造を修正したものを使用します。単一サーバ (非クラスタ化) 環境では、クライアントは EJB のホーム インタフェースを通じて EJB をルックアップします。ホーム インタフェースの背後には、対応するホーム オブジェクトがサーバ上に存在します。Bean の参照後、クライアントはリモート インタフェースを通じてその Bean のメソッドと対話します。リモート インタフェースの背後には、EJB オブジェクトがサーバ上に存在します。

次の図は、単一サーバ環境での EJB の動作を示しています。

図 4-8 単一サーバの動作



注意： EJB のフェイルオーバは、リモートクライアントと EJB との間でのみ機能します。

クラスタ化された EJB ホーム オブジェクト

すべての EJB タイプ (ステートレス セッション EJB、ステートフル セッション EJB、およびエンティティ EJB) は、クラスタ対応のホーム スタブを持つことができます。クラスタ対応のホーム スタブを作成するかどうかは、`weblogic-ejb-jar.xml` の `home-is-clusterable` デプロイメント記述子要素によって決定されます。

EJB がクラスタにデプロイされると、そのホームはクラスタワイドのネーミング サービスにバインドされます。各サーバではそのホームのインスタンスを同じ名前でバインドできます。クライアントがこのホームをルックアップすると、そのクライアントでは Bean がデプロイされた各サーバ上のホームへの参照を持つレプリカ対応スタブが取得されます。`create()` または `find()` が呼び出されると、その呼び出しはレプリカ対応スタブによってレプリカの 1 つに転送されます。ホームのレプリカは、`find()` の結果を受信するか、またはそのサーバで Bean のインスタンスを作成します。

クラスタ化されたホーム スタブは、EJB ルックアップ リクエストを使用可能なサーバに分散することでロード バランシングを実現します。また、他のサーバに障害が発生したときにルックアップ リクエストを使用可能なサーバに転送することで、それらのリクエストのフェイルオーバもサポートします。

クラスタ化された EJBObject

WebLogic Server クラスタでは、EJBObject のサーバサイド表現は、レプリカ対応の EJBObject スタブによって置き換えることができます。このスタブは、クラスタ内のサーバに存在する EJBObject のすべてのコピーに関する知識を保持します。EJBObject スタブは、EJB メソッド呼び出しに対するロード バランシングおよびフェイルオーバを提供します。たとえば、クライアントが特定の WebLogic Server に対して EJB メソッドを呼び出したが、そのサーバがダウンしている場合、EJBObject スタブはそのメソッド呼び出しを稼働中の別のサーバにフェイルオーバします。

EJB がレプリカ対応 EJBObject スタブを利用するかどうかは、デプロイされている EJB のタイプと、エンティティ EJB の場合、デプロイメント時に選択した同時方式によって決まります。詳細については、4-34 ページの「さまざまなタイプの EJB に対するクラスタ化のサポート」を参照してください。

さまざまなタイプの EJB に対するクラスタ化のサポート

次の節では、セッション EJB とエンティティ EJB に対するクラスタ化のサポートについて説明します。

クラスタ内のステートレス セッション EJB

ステートレス セッション EJB は、クラスタ対応のホーム スタブとレプリカ対応の EJBObject スタブの両方を持つことができます。デフォルトによって、WebLogic Server は EJB メソッド呼び出しに対するフェイルオーバ サービスを提供しますが、これは障害がメソッド呼び出しの合間に発生した場合に限ります。たとえば、メソッドの完了後に障害が発生した場合や、メソッドがサーバに接続

されなかった場合などには、フェイルオーバが自動的にサポートされます。しかし、EJB メソッドの実行中に障害が発生した場合、WebLogic Server は別のサーバにそのメソッドを自動的にフェイルオーバしません。

このデフォルト動作では、EJB メソッド内のデータベース更新がフェイルオーバによって「重複」することはありません。たとえば、データストア内のある値をインクリメントするメソッドをクライアントが呼び出し、そのメソッドが完了する前に WebLogic Server が別のサーバにフェイルオーバした場合、クライアントの 1 度のメソッド呼び出しによってデータベースが 2 度更新されることになります。

繰り返し呼び出しても更新が重複しないように記述されたメソッドを、「多重呼び出し不変」と呼びます。WebLogic Server には、多重呼び出し不変のメソッド用に 2 種類の `weblogic-ejb-jar.xml` デプロイメント プロパティ (一方は Bean レベル、もう一方はメソッド レベル) が用意されています。

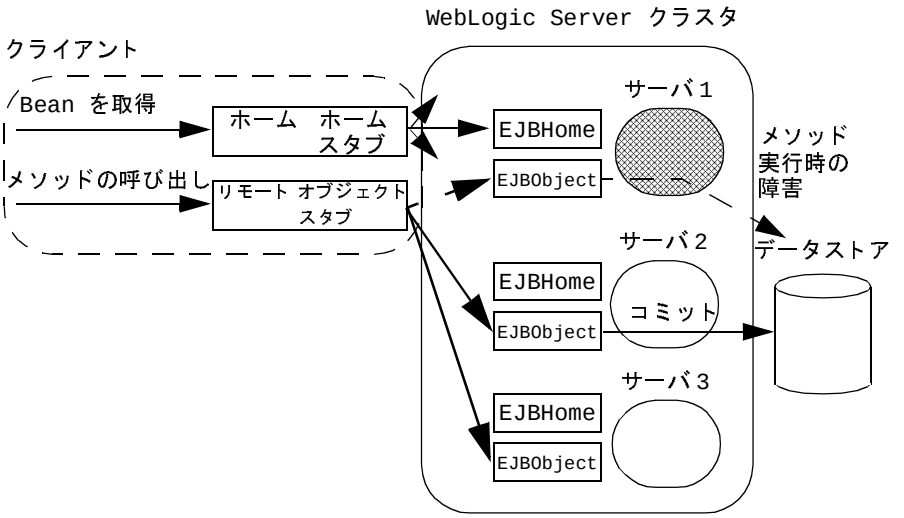
Bean レベルでは、`stateless-bean-methods-are-idempotent` を「true」に設定した場合、WebLogic Server はメソッドが多重呼び出し不変であると見なして、メソッド呼び出し中に障害が発生した場合でも、EJB メソッドに対するフェイルオーバ サービスを提供します。

メソッド レベルでは、`idempotent-methods` デプロイメント プロパティを使用して同じことを実行できます。

```
<idempotent-methods>
       <method>
              <description>...</description>
              <ejb-name>...</ejb-name>
              <method-intf>...</method-intf>
              <method-name>...</method-name>
              <method-params>...</method-params>
       </method>
</idempotent-methods>
```

次の図は、WebLogic Server クラスタ環境でのステートレス セッション EJB の動作を示しています。

図 4-9 クラスタ化されたサーバ環境でのステートレス セッション EJB



クラスタ内のステートフル セッション EJB

ステートフル セッション EJB でクラスタ対応のホーム スタブを利用できるようにするには、home-is-clusterable を「true」に設定します。これにより、ステートフル EJB ルックアップに対するフェイルオーバとロード バランシングが提供されます。このようにコンフィグレーションされたステートフル セッション EJB は、レプリカ対応の EJBObject スタブを利用できます。ステートフル セッション EJB のインメモリ レプリケーションの詳細については、4-37 ページの「ステートフル セッション EJB のインメモリ レプリケーション」を参照してください。

注意： ロード バランシングとフェイルオーバについては、『WebLogic Server クラスタ ユーザーズ ガイド』で詳しく説明されています。次の 3 つの節を参照してください。「EJB と RMI オブジェクト」、「EJB と RMI オブジェクトのロード バランシング」、「EJB と RMI のレプリケーションとフェイルオーバ」。

ステートフル セッション EJB のインメモリ レプリケーション

WebLogic Server EJB コンテナは、ステートフル セッション EJB 向けのクラスタ化をサポートします。WebLogic Server 5.1 では、`EJBHome` オブジェクトだけがステートフル セッション EJB 用にクラスタ化されますが、EJB コンテナは、EJB の状態をクラスタ化された WebLogic Server インスタンスの間でレプリケートできます。

ステートフル セッション EJB のレプリケーション サポートは、EJB クライアントには見えません。ステートフル セッション EJB がデプロイされると、WebLogic Server はそのステートフル セッション EJB 用にクラスタ対応の `EJBHome` スタブとレプリカ対応の `EJBObject` スタブを作成します。`EJBObject` スタブは、EJB インスタンスが動作するプライマリ WebLogic Server インスタンスと、その Bean の状態をレプリケートするためのセカンダリ WebLogic Server のリストを保持します。

EJB のクライアントがその EJB の状態を変更するトランザクションをコミットするたびに、WebLogic Server はその EJB の状態をセカンダリ サーバ インスタンスにレプリケートします。Bean の状態のレプリケーションは直接メモリ内で行われるため、クラスタ化された環境で最高のパフォーマンスを得られます。

プライマリ サーバ インスタンスがダウンした場合、クライアントの次のメソッド呼び出しはセカンダリ サーバの EJB インスタンスに自動的に転送されます。セカンダリ サーバは、その EJB インスタンスのプライマリ WebLogic Server となり、新しいセカンダリ サーバが別のフェイルオーバに対応します。EJB のセカンダリ サーバがダウンした場合、WebLogic Server はクラスタから新しいセカンダリ サーバ インスタンスを使用します。

このため、ステートフル セッション EJB のクライアントは EJB の最後にコミットされた状態に迅速にアクセスできます。ただし、後述の 4-38 ページの「インメモリ レプリケーションの制限事項」で挙げる特殊な環境は除きます。レプリケーション グループの使い方の詳細については、「レプリケーション グループを使用する」を参照してください。

インメモリ レプリケーションの要件とコンフィグレーション

WebLogic Server クラスタでステートフル セッション EJB の状態をレプリケートするには、クラスタの EJB クラスを同種にする必要があります。つまり、同じデプロイメント プロパティを使用して、クラスタ内のすべての WebLogic Server インスタンスに同じ EJB クラスをデプロイしなければなりません。異種クラスタに対するインメモリ レプリケーションはサポートされていません。

デフォルトでは、WebLogic Server はクラスタ内でステートフル セッション EJB をレプリケートしません。これは、WebLogic Server バージョン 6.0 でリリースされた動作をモデル化したものです。レプリケーションを有効にするには、`weblogic-ejb-jar.xml` デプロイメント ファイルの `replication-type` デプロイメント パラメータを `InMemory` に設定します。

図 4-10 レプリケーションを有効にする XML の例

```
<stateful-session-clustering>
        ...
       <replication-type>InMemory</replication-type>
</stateful-session-clustering>
```

インメモリ レプリケーションの制限事項

ステートフル セッション EJB の状態をレプリケートすることによって、プライマリ WebLogic Server インスタンスがダウンした場合でも、一般にクライアントは EJB の最後にコミットされた状態を取得できます。しかし、次のフェイルオーバのシナリオでは、最後にコミットされた状態を取得できないこともまれにあります。

- ステートフル EJB が関与するトランザクションをクライアントがコミットしたが、その EJB の状態がレプリケートされる前にプライマリ WebLogic Server がダウンした場合。この場合、クライアントの次のメソッド呼び出しは前回コミットされた状態に対して実行されます。
- クライアントがステートフル セッション EJB のインスタンスを作成し、最初のトランザクションをコミットしたが、その EJB の初期状態がレプリケートされる前にプライマリ WebLogic Server がダウンした場合。初期状態をレプリケートできなかったため、クライアントの次のメソッド呼び出しは EJB インスタンスを見つけることができません。クライアントは、クラスタ化された `EJBHome` スタブを使って EJB インスタンスを再作成し、トランザクションをやり直す必要があります。

- プライマリ サーバとセカンダリ サーバがどちらもダウンした場合。クライアントは EJB インスタンスを再作成し、トランザクションをやり直す必要があります。

クラスタ内のエンティティ EJB

すべての EJB と同様に、エンティティ EJB は、`home-is-clusterable` を「true」に設定することによって、クラスタ対応のホーム スタブを利用できます。

`EJBObject` スタブの動作は、`weblogic-ejb-jar.xml` の `concurrency-strategy` デプロイメント要素によって決まります。`concurrency-strategy` は `Read-Write` または `Read-Only` に設定できます。デフォルト値は `Read-Write` です。

詳細については、以下の節を参照してください。

- 4-39 ページの「クラスタ内の読み込み専用エンティティ EJB」
- 4-39 ページの「クラスタ内の読み書き対応エンティティ EJB」

クラスタ内の読み込み専用エンティティ EJB

ホームで読み込み専用エンティティ Bean が検索または作成される場合は、レプリカ対応の `EJBObject` スタブが返されます。このスタブでは、すべての呼び出しでロード バランシングが行われますが、回復可能な呼び出しエラーが発生したときに自動的にフェイルオーバは行われません。読み込み専用 Bean は、データベース読み込みを防止するためにすべてのサーバでキャッシュされます。

クラスタ内の読み書き対応エンティティ EJB

ホームで読み書き対応エンティティ Bean が検索または作成される場合は、ローカル サーバでインスタンスが取得され、そのサーバに固定された `EJBObject` スタブが返されます。ロード バランシングとフェイルオーバはホームのレベルでのみ行われます。クラスタにはエンティティ Bean の複数のインスタンスが存在する可能性があるため、各インスタンスは各トランザクションの前にデータベースから読み込まれ、各コミットで書き込まれなければなりません。

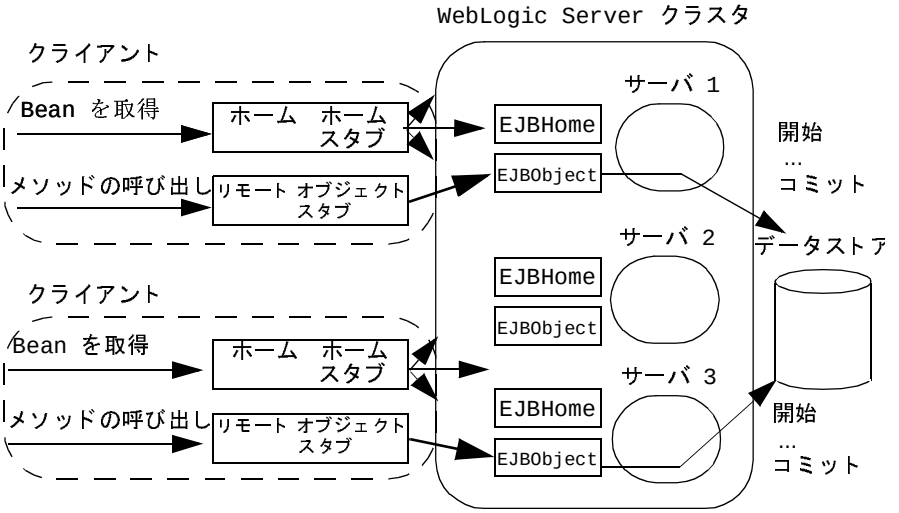
クラスタ内の `read-write` エンティティ EJB は、以下のとおり、非クラスタ化システム内のエンティティ EJB と同じように動作します。

- 複数のクライアントがトランザクション内の Bean を使用できる。

- `ejbLoad()` は、`cache-between-transactions` が `false` に設定されている場合、常に各トランザクションの最初に呼び出される。
- `ejbStore()` の動作は、4-13 ページの「エンティティ EJB に対する ejbLoad() と ejbStore() の動作」で説明したルールに従います。

図 4-11 では、WebLogic Server クラスタ環境での読み書き対応エンティティ EJB の動作を示しています。ホーム スタブ上の 3 つの矢印は 3 つのサーバすべてを指し、複数のクライアント アクセスを示しています。

図 4-11 クラスタ化されたサーバ環境の読み書き対応エンティティ EJB



注意： 上の図では、ホーム スタブを起点とする矢印はいずれも各サーバの EJBHome を指しています。

読み書き対応エンティティ EJB は、`home-is-clusterable` が true に設定されている場合に、安全な例外に関して自動フェイルオーバをサポートします。たとえば、メソッドの完了後に障害が発生した場合や、メソッドがサーバに接続されなかった場合などには、フェイルオーバが自動的にサポートされます。

クラスタ アドレス

クラスタをコンフィグレーションする際に、クラスタ内の管理対象サーバを識別するクラスタ アドレスを入力します。クラスタ アドレスは、URL のホスト名部分を構築するためにエンティティ Bean およびステートレス Bean で使用されます。クラスタ アドレスが設定されていない場合、EJB ハンドルは正しく動作しません。クラスタ アドレスの詳細については、『WebLogic Server クラスタ ユーザーズ ガイド』を参照してください。

トランザクション管理

以降の節では、EJB コンテナによるトランザクション管理サービスのサポートについての情報を示します。いくつかのトランザクション シナリオを通して EJB を説明していきます。分散トランザクション (複数のデータストアで更新を行うトランザクション) に関与する EJB は、トランザクションのすべてのブランチが論理単位としてコミットまたはロールバックされることを保証します。

WebLogic Server の現行バージョンは、Java Transaction API (JTA) をサポートしています。JTA は、分散トランザクション対応アプリケーションの実装に使用できます。

また、1.1 と 2.0 のどちらの EJB の場合でも 2 フェーズ コミットがサポートされています。2 フェーズ コミット プロトコルは、複数のリソース マネージャにまたがって 1 つのトランザクションを調整する手段です。これにより、トランザクションによる更新を関連データベースのすべてにコミットするか、またはすべてのデータベースから完全にロールバックし、トランザクションによる状態の前の状態に戻すことで、データの完全性が保証されます。

トランザクション管理の責任範囲

セッション EJB は、自身のコード、そのクライアントのコード、または WebLogic Server コンテナに基づいてトランザクション境界を定義できます。EJB は、コンテナまたはクライアントによって定められたトランザクション境界を使用できます。しかし、一定の制限に従わない限り、独自のトランザクション境界を定義することはできません (11.2.5)。

- **Bean 管理のトランザクション**では、EJB のコードがトランザクションの境界定義を管理します。Bean 管理またはクライアント管理のトランザクションが必要な場合、Java コードを記述して `javax.transaction.UserTransaction` インタフェースを使用しなければなりません。これにより、EJB またはクライアントは、JNDI を通じて `UserTransaction` オブジェクトにアクセスし、`tx.begin()`、`tx.commit()`、`tx.rollback()` を明示的に呼び出してトランザクション境界を指定できます (11.2.3)。トランザクション境界の定義の詳細については、4-43 ページの「javax.transaction.UserTransaction の使い方」を参照してください。
- **コンテナ管理のトランザクション**では、WebLogic Server EJB コンテナによってトランザクションの境界設定が管理されます。コンテナ管理トランザクションを使用する EJB については、個々の EJB メソッドのトランザクション要件を指定する各種のデプロイメント要素を使用できます。デプロイメント記述子の詳細については、『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』を参照してください。

注意： EJB プロバイダが `ejb-jar.xml` ファイルにメソッドのトランザクション属性を指定しない場合、WebLogic Server はデフォルトで supports 属性を使用します。

トランザクション イベントのシーケンスは、コンテナ管理のトランザクションと Bean 管理のトランザクションで異なります。

javax.transaction.UserTransaction の使い方

EJB またはクライアント コードにトランザクション境界を定義するには、Java Transaction Service (JTS) または JDBC データベース接続を取得する前に、UserTransaction オブジェクトを取得してトランザクションを開始しなければなりません。UserTransaction オブジェクトを取得するには、次のコマンドを使用します。

```
ctx.lookup("javax.transaction.UserTransaction");
```

データベース接続を取得してからトランザクションを開始した場合、その接続は新しいトランザクションと何の関連性も持たず、以後のトランザクション コンテキストにその接続を「入れる」ためのセマンティクスも存在しません。JTS 接続がトランザクション コンテキストに関連付けられていない場合、JTS 接続は自動コミットを true に設定した標準の JDBC 接続と同じように動作し、更新はデータストアに自動的にコミットされます。

いったんトランザクション コンテキスト内にデータベース接続を作成すれば、その接続は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まで「予約」状態になります。アプリケーションのパフォーマンスとスループットを維持するには、常にトランザクションを迅速に完了させることによって、データベース接続を解放し、他のクライアント リクエストが使用できるようにする必要があります。詳細については、2-8 ページの「トランザクション リソースの保持」を参照してください。

注意： アクティブなトランザクション コンテキストには単一のデータベース接続しか関連付けることができません。

コンテナ管理 EJB に対する制限

コンテナ管理によるトランザクションを使用する EJB 内で javax.transaction.UserTransaction メソッドを使用することはできません。

トランザクションのアイソレーション レベル

トランザクションのアイソレーション レベルを設定する方法は、アプリケーションが Bean 管理による境界設定を使用するか、コンテナ管理による境界設定

を使用するかによって異なります。以下の節では、これらのシナリオの各々について検討します。

ユーザ トランザクションのアイソレーション レベルの設定

ユーザ トランザクションのアイソレーション レベルは、Bean の Java コードで設定します。アプリケーションが実行されると、トランザクションが明示的に開始されます。レベルの設定方法を示すコードの例については、図 4-12 を参照してください。

図 4-12 ユーザ トランザクションのアイソレーション レベルを設定する Java コード例

```
import javax.transaction.Transaction;
import java.sql.Connection
import weblogic.transaction.TxHelper:
import weblogic.transaction.Transaction;
import weblogic.transaction.TxConstants;

User Transaction tx = (UserTransaction)

ctx.lookup("javax.transaction.UserTransaction");

// ユーザ トランザクションを開始する

       tx.begin();

// トランザクションのアイソレーション レベルを TRANSACTION_READ_COMMITED に設定する

Transaction tx = TxHelper.getTransaction();
       tx.setProperty (TxConstants.ISOLATION_LEVEL, new Integer
        (Connection.TRANSACTION_READ_COMMITED));

// トランザクションの処理を実行する

       tx.commit();
```

コンテナ管理トランザクションのアイソレーション レベルの設定

コンテナ管理トランザクションのアイソレーション レベルは、`weblogic-ejb-jar.xml` デプロイメント ファイルの `transaction-isolation` 要素の `isolation-level` サブ要素で設定します。WebLogic Server は、この値を基盤データベースに渡します。トランザクションの動作は、EJB のアイソレー

ション レベル設定と、基盤となる永続ストレージの同時実行性の制御に依存します。コンテナ管理トランザクションのアイソレーション レベルの設定に関する詳細については、『WebLogic JTA プログラマーズ ガイド』を参照してください。

TransactionSerializable の制限

多くのデータストアでは、シリアライゼーションに関する問題の検出は、単一ユーザ接続の場合でさえ制限されています。したがって、`transaction-level` を EJB に対して `TransactionSerializable` 設定にした場合でも、同じ行に対してクライアント間で競合が起きると、その EJB クライアントで例外またはロールバックを受け取ることがあります。こうした問題を避けるには、クライアント アプリケーションのコードが SQL 例外を検出および調査して、例外を解決するためにトランザクションをやり直すなどの適切なアクションを取るようにしなければなりません。

WebLogic Server では、4-45 ページの「Oracle データベースに関する特別な注意」で説明しているように、Oracle データベースでこの問題を回避するために設計された特別な `isolation-level` 設定を用意しています。

その他のデータベース ベンダについては、使用しているデータベースのマニュアルでアイソレーション レベルのサポートに関する詳細を参照してください。

Oracle データベースに関する特別な注意

`isolation-level` の設定を `TransactionSerializable` にしたとしても、Oracle ではコミット時までシリアライゼーションの問題が検出されません。返されるエラー メッセージは次のとおりです。

```
java.sql.SQLException: ORA-08177: can't serialize access for this
transaction
```

WebLogic Server では、これを回避するために特別な `isolation-level` 設定を提供しています。

- このオプションを定義するメソッドに対して `TransactionReadCommittedForUpdate` を設定した場合。設定されると、その時点から各 `SELECT` クエリには、選択した行で取得されるロックに `ForUpdate` が追加されます。その結果、クエリによって変更された行を即座にロックできなければ、Oracle はその行が解放されるまで待ちます。この状

態は、トランザクションが COMMIT または ROLLBACK を行うまで有効となります。

- このオプションを定義するメソッドに対して TransactionReadCommittedForUpdateNoWait を設定した場合。設定されると、その時点から各 SELECT クエリには、選択した行のロックを取得する ForUpdateNoWait が追加されます。その結果、クエリによって変更された行を即座にロックできなければ、Oracle はそのクエリを途中で打ち切ります。トランザクションがコミットまたはロールバックするまで、この条件が有効になります。

  注意： ForUpdateNoWait は、コンテナ管理 Bean に対してのみ適用されます。

複数の EJB 間でのトランザクションの分散

WebLogic Server は、複数のデータストアに分散されるトランザクションをサポートしています。このため、単一のデータベース トランザクションを複数のサーバの複数の EJB に分散できます。これらのタイプのトランザクションを明示的にサポートするには、トランザクションを開始し、複数の EJB を呼び出します。また、単一の EJB が、同じトランザクション コンテキスト内で暗黙的に動作する他の EJB を呼び出す場合もあります。以降の節では、これらのシナリオについて説明します。

単一トランザクション コンテキストから複数の EJB を呼び出す

次のコードでは、クライアント アプリケーションは UserTransaction オブジェクトを取得し、それを使ってトランザクションを開始してコミットします。クライアントは、トランザクションのコンテキストの中で 2 つの EJB を呼び出します。各 EJB のトランザクション属性は、Required に設定されています。

図 4-13 トランザクションの開始とコミット

```
import javax.transaction.*;
...
u = (UserTransaction)
jndiContext.lookup("javax.transaction.UserTransaction");
u.begin();
```

```
account1.withdraw(100);
account2.deposit(100);
u.commit();
...
```

上のコードでは、「account1」および「account2」EJB によって行われる更新は、単一の `UserTransaction` のコンテキスト内で発生します。EJB は、1 つの論理単位としてコミットまたはロールバックされます。このことは、「account1」と「account2」が同じ WebLogic Server、複数の WebLogic Server、または WebLogic Server クラスタのどれに存在している場合にも当てはまります。

EJB 呼び出しをこのようにラップするための条件は、「account1」と「account2」のどちらもクライアント トランザクションをサポートしなければならないということだけです。これを満たすには、Bean の `trans-attribute` 要素を `Required`、`Supports`、または `Mandatory` に設定します。

複数操作トランザクションをカプセル化する

また、トランザクションをカプセル化する「ラッパー」EJB を使用することもできます。クライアントは、ラッパー EJB を呼び出して銀行振替などのアクションを実行します。ラッパー EJB は、新しいトランザクションを開始し、1 つまたは複数の EJB を呼び出してトランザクションの作業を実行することで、それに応答します。

「ラッパー」EJB は、他の EJB を呼び出す前にトランザクション コンテキストを明示的に取得することもできます。また、EJB の `trans-attribute` 要素が `Required` または `RequiresNew` に設定されている場合、WebLogic Server は新しいトランザクション コンテキストを自動的に作成できます。`trans-attribute` 要素は、`weblogic-ejb-jar.xml` ファイルで設定します。ラッパー EJB によって呼び出されるすべての EJB は、トランザクション コンテキストをサポートできなければなりません (その `trans-attribute` 要素が `Required`、`Supports`、または `Mandatory` に設定されていなければなりません)。

WebLogic Server クラスタ内の複数の EJB 間でトランザクションを分散する

WebLogic Server では、WebLogic Server クラスタ内の EJB に対してはトランザクションのパフォーマンスがさらに向上します。単一のトランザクションが複数の EJB を使用する場合、WebLogic Server は異なるサーバの EJB ではなく、単一

の WebLogic Server インスタンスに存在する EJB インスタンスを使おうとします。この方法を使用すると、トランザクションのネットワーク トラフィックを最小限に抑えることができます。

場合によっては、トランザクションはクラスタ内の複数の WebLogic Server インスタンスに存在する EJB を使用します。これは、すべての EJB がすべての WebLogic Server インスタンスにデプロイされていない異種クラスタで起こる場合があります。このような場合、WebLogic Server は複数の直接接続ではなく 1 つの多層接続を使用してデータベースにアクセスします。この方法を使うと、リソースの使用量が減り、トランザクションのパフォーマンスが向上します。

しかし、最高のパフォーマンスを得るには、クラスタを同種にする必要があります。つまり、すべての EJB が使用可能なすべての WebLogic Server インスタンスに存在する必要があります。

データベースの挿入サポート

WebLogic Server では、EJB コンテナが新規作成された Bean をどの時点で、どのようにデータベースに挿入するかを管理できます。`weblogic-cmp-rdbms-jar.xml` ファイルの `delay-database-insert-until` デプロイメント記述子要素を設定することによって各自の希望を指定します。次のものから選択できます。

- 次節で説明するように、EJB コンテナが `ejbCreate` か `ejbPostCreate` を実行した後まで、データベースの挿入を遅延します。
- 4-49 ページの「一括挿入」で説明するように、1 つの SQL 文でデータベースに複数のエントリを挿入する。

`delay-database-insert-until` 要素には、以下の値を指定できます。

- `ejbCreate` ー このメソッドは、`ejbCreate` の直後にデータベースの挿入を実行します。
- `ejbPostCreate` ー このメソッドは、`ejbPostCreate` の直後に挿入を実行します (デフォルト)。
- `commit` ー トランザクションがコミットしたとき一括挿入を実行します。

図 4-14 delay-database-insert-until を指定する XML の例

```
<delay-database-insert-until>ejbPostCreate</delay-database-insert
-until> -->
```

Delay-Database-Insert-Until

デフォルトでは、データベースへの挿入はクライアントが ejbPostCreate メソッドを呼び出した後に行われます。weblogic-cmp-rdbms-jar.xml ファイルの delay-database-insert-until 要素に ejbCreate または ejbPostCreate を指定した場合、EJB コンテナは新規 Bean の挿入を遅延させます。このどちらかのオプションを指定することによって、EJB コンテナが RDBMS CMP を使用する新しい Bean をいつデータベースに挿入するかを正確に指定します。

cmr-field が null 値を許可しない foreign-key column にマップされている場合、EJB コンテナがデータベースの挿入を ejbPostCreate の後まで遅らせるように指定する必要があります。この場合、cmr-field を ejbPostCreate で null でない値に設定してから Bean をデータベースに挿入します。

注意： Bean の主キーが不明な段階で、cmr-fields を ejbCreate メソッド呼び出しの中で設定することはできません。

BEA では、ejbPostCreate メソッドが Bean の永続フィールドを変更した場合には、データベースの挿入を ejbPostCreate の後に遅らせるよう指定することを推奨しています。これにより、不要な保存操作を行わずに済むので、パフォーマンスが向上します。

最大限の柔軟性を実現するため、関連 Bean を ejbPostCreate メソッドで作成することは避けてください。こうしたインスタンスを追加作成すると、データベースの制約によって関連 Bean が未作成の Bean を参照できない場合、データベースの挿入を遅らせることができなくなる可能性があります。

一括挿入

一括挿入のサポートにより、コンテナ管理による永続性 (CMP) Bean の作成における効率が向上します。EJB コンテナが 1 つの SQL 文で CMP Bean での複数回のデータベース挿入を実行できるようになるためです。この機能により、コンテナは何度もデータベースを挿入しなくて済むようになります。

weblogic-cmp-rdbms-jar.xml ファイルの delay-database-insert-until 要素に commit オプションを指定した場合、EJB コンテナは一括データベース挿入を実行します。

一括挿入更新を使用する場合、トランザクションの開始からコミットまでの間に一括挿入だけが挿入に適用されるように、トランザクションの境界を設定しなければなりません。

注意： 一括挿入は、addBatch() と executeBatch() メソッドをサポートしているドライバのみに機能します。たとえば、Oracle Thin ドライバはこれらのメソッドをサポートしていますが、WebLogic Oracle JDBC ドライバはサポートしていません。

一括挿入の使用に際しては、次の 2 つの制限があります。

1 回の一括挿入で作成するエントリの総数が、weblogic-ejb-jar.xml ファイルで指定した max-beans-in-cache の設定値を超えることはできません。この要素の詳細については、9-51 ページの「max-beans-in-cache」を参照してください。

weblogic-cmp-rdbms-jar.xml ファイルの dbms-column-type 要素で OracleBlob か OracleClob を設定した場合、一括挿入は自動的に無効になります。Blob または Clob カラムがデータベース テーブルに含まれている状況では、時間短縮があまりできないからです。この場合、デフォルトでは WebLogic Server は Bean ごとに挿入を実行します。

リソース ファクトリ

以降の節では、EJB コンテナによるリソース サービスのサポートについての情報を示します。WebLogic Server では、EJB は JDBC ドライバを直接インスタンス化することによって JDBC 接続プールにアクセスできます。しかし、その代わりに、JDBC データソース リソースをリソース ファクトリとして WebLogic Server JNDI ツリーにバインドすることをお勧めします。

リソース ファクトリを使用すると、EJB は EJB デプロイメント記述子内のリソース ファクトリ参照を稼働中の WebLogic Server で使用可能なリソース ファクトリにマップできます。リソース ファクトリ参照は使用するリソース ファクトリ タイプを定義しなければなりませんが、リソースの実際の名前は、Bean がデプロイされるまで指定されません。

以降の節では、JDBC データソース および URL リソースを WebLogic Server の JDNI 名にマップする方法について説明します。

注意： WebLogic Server は、JMS 接続ファクトリもサポートしています。

JDBC データソース ファクトリの設定

次の手順に従って、`javax.sql.DataSource` リソース ファクトリを WebLogic Server 内の JNDI 名にバインドします。必要に応じて、トランザクション対応か非対応の JDBC データソースを設定できます。

トランザクション非対応のデータソースの場合、JDBC 接続は自動コミットモードで動作し、コンテナ管理トランザクションの一部にはならず、挿入操作や更新操作のつど、データベースに即座にコミットします。

トランザクション対応のデータソースの場合、あるメソッド上の複数回の挿入および更新操作を 1 つのコンテナ管理トランザクションとして送信することができ、1 つの論理単位としてコミットまたはロールバックされます。

注意： コンテナ管理による永続性を使用するエンティティ Bean では、データの一貫性を保つため、トランザクション非対応のデータソースではなくトランザクション対応のデータソースを必ず使用します。

JDBC データソース ファクトリを作成するには、以下の手順に従います。

1. Administration Console で JDBC 接続プールを設定します。詳細については、『管理者ガイド』の「JDBC 接続の管理」を参照してください。
2. WebLogic Server を起動します。
3. WebLogic Server Administration Console を起動します。
4. Administration Console の左ペインで、[サービス] ノードをクリックして JDBC を展開します。

5. [JDBC データ ソース ファクトリ] を選択し、右ペインで [新しい JDBCDataSource のコンフィグレーション] オプションをクリックします。
6. [名前]、[ユーザ名]、[URL]、[ドライバ クラス名]、[ファクトリ名] 属性フィールドに値を入力します。
7. [プロパティ] 属性フィールドにすべての接続プロパティを入力します。
   a. トランザクション非対応の JDBC データソースの場合は、次のとおり入力します。

   `weblogic.jdbc.DataSource.`*`jndi_name`*`=`*`pool_name`*

   ここで *`jndi_name`* は、データソースにバインドする完全な WebLogic Server JNDI 名、*`pool_name`* は、手順 1 で作成した WebLogic Server 接続プールの名前です。

   b. トランザクション対応の JDBC データベースの場合は、Administration Console の左ペインで [トランザクション データソース] を選択し、右ペインで [新しい JDBCTxDataSource のコンフィグレーション] をクリックし、次のように入力します。

   `weblogic.jdbc.TXDataSource.`*`jndi_name`*`=`*`pool_name`*

   ここで *`jndi_name`* はトランザクション対応データソースにバインドする完全な WebLogic Server JNDI 名、*`pool_name`* は手順 1 で作成した WebLogic Server 接続プールの名前です。

   トランザクション データ ソースと非トランザクション データ ソースの詳細については、「JDBC データ ソースのコンフィグレーション」を参照してください。
8. [作成] をクリックして JDBC データ ソース ファクトリを作成します。左ペインの JDBC データソース ノードの下に新しいデータソース ファクトリが追加されます。
9. [適用] をクリックして変更を保存します。
10. 次のいずれかの方法で、データソースの JNDI 名を EJB のローカル JNDI 環境にバインドします。
    - 既存の EJB リソース ファクトリ参照を JNDI 名にマップする。
    - テキスト エディタを使用して、`weblogic.ejb-jar.xml` デプロイメント ファイルの `resource-description` 要素を直接編集する。デプロイメン

ト記述子を編集する手順については、6-6 ページの「EJB デプロイメント記述子の指定と編集」を参照してください。

URL 接続ファクトリの設定

WebLogic Server で URL 接続ファクトリを設定するには、次の手順で URL 文字列を JNDI 名にバインドします。

1. 使用している WebLogic Server のインスタンス用の `config.xml` ファイルをテキスト エディタで開き、以下の `config.xml` 要素の `URLResource` 属性を設定します。
   - `WebServer`
   - `VirtualHost:`
2. 次の構文に従って、`WebServer` 要素の `URLResource` 属性を設定します。

   ```
   <WebServer URLResource="weblogic.httpd.url.testURL=http://
   localhost:7701/testfile.txt” DefaultWebApp=”default-tests”/>
   ```

3. 仮想ホストが必要な場合は、次の構文に従って `VirtualHost` 要素の `URLResource` 属性を設定します。

   ```
   <VirtualHostName=guestserver" targets="myserver,test_web_server
   "URLResource="weblogic.httpd.url.testURL=http://
   localhost:7701/testfile.txt” VirtualHostNames=”guest.com”/>
   ```

4. 変更を `config.xml` に保存し、WebLogic Server を再起動します。

5 WebLogic Server のコンテナ管理による永続性サービス

以下の節では、WebLogic Server EJB コンテナでサポートされているコンテナ管理による永続性（CMP）機能について説明します。

- コンテナ管理による永続性サービスの概要
- EJB 1.1 CMP の RDBMS 永続性用の記述
- EJB 1.1 CMP 用の WebLogic クエリ言語 (WLQL) の使用
- CMP 1.1 ファインダ クエリとしての SQL の使用
- EJB 2.0 用 EJB QL の使い方
- 動的クエリの使用
- Oracle の SELECT HINT の使用
- 「get」および「set」メソッドの制限
- Oracle DBMS の BLOB および CLOB DBMS カラムのサポート
- WebLogic Server での EJB 1.1 CMP の調整更新
- WebLogic Server での EJB 1.1 CMP の調整更新
- CMP キャッシュのフラッシュ
- 主キーの使用
- EJB 2.0 CMP に対する自動主キー生成
- EJB 2.0 CMP の複数のテーブル マッピング
- 自動テーブル作成
- コンテナ管理による関係
- グループ

- EJB リンクの使用
- CMP フィールドの Java データ型

コンテナ管理による永続性サービスの概要

EJB コンテナは、EJB と WebLogic Server 間の統一的なインタフェースを提供します。コンテナは、EJB の新しいインスタンスを作成し、これらの Bean リソースを管理し、トランザクション、セキュリティ、同時実行性、ネーミングなどの永続性サービスを実行時に提供します。ほとんどの場合、WebLogic Server の旧バージョンの EJB はコンテナで動作します。ただし、Bean のコードの移行が必要な場合については、『Migration Guide』を、変換ツールの使い方については、8-26 ページの「DDConverter」を参照してください。

WebLogic Server のコンテナ管理による永続性 (CMP) モデルでは、EJB のインスタンス フィールドをデータベースのデータと同期することで、CMP エンティティ Bean の永続性を実行時に自動処理します。

エンティティ Bean では、コンテナ管理による永続性を利用して、エンティティ Bean インスタンスで永続データ アクセスを実行するメソッドが生成されます。生成されたメソッドでは、エンティティ Bean インスタンスと基盤のリソース マネージャの間でデータが転送されます。永続性は実行時にコンテナによって処理されます。コンテナ管理の永続性を利用する利点は、エンティティが格納されるデータ ストアからエンティティ Bean が論理的に独立することです。コンテナでは、論理的な関係と物理的な関係のマッピングが実行時に管理されると同時に、それらの参照整合性が管理されます。

永続フィールドと関係によって、エンティティ Bean の抽象永続性スキーマが構成されます。デプロイメント記述子は、エンティティ Bean でコンテナ管理による永続性が使用されることを示します。また、デプロイメント記述子は、データにアクセスするコンテナへの入力としても使用されます。

EJB の永続性サービス

WebLogic Server は、エンティティ Bean に永続性サービスを提供します。エンティティ EJB が任意のトランザクション対応またはトランザクション非対応の永続ストレージにその状態を保存する (「Bean 管理による永続性」) ことも、コンテナが EJB の非 transient インスタンス変数を自動的に保存する (「コンテナ管理による永続性」) こともできます。WebLogic Server では、このどちらも選択可能であり、両者を併用することもできます。

EJB がコンテナ管理の永続性を使用する場合、EJB が使用する永続性サービスのタイプを `weblogic-ejb-jar.xml` デプロイメント ファイルに指定します。自動永続性サービスのハイレベル定義は、persistence-use 要素に格納されます。`persistence-use` には、EJB がデプロイ時に使用するサービスが定義されます。

自動永続性サービスは、さらに別のデプロイメント ファイルを使用してそのデプロイメント記述子を指定し、エンティティ EJB ファインダ メソッドを定義します。たとえば、WebLogic Server RDBMS ベースの永続性サービスは、特定の Bean からデプロイメント記述子とファインダ定義を取得するときに、その Bean の `weblogic-cmp-rdbms-jar.xml` ファイルを使用します。詳細については、5-3 ページの「WebLogic Server RDBMS 永続性の使い方」で説明します。

サードパーティの永続性サービスでは、他のファイル フォーマットを使用してデプロイメント記述子をコンフィグレーションします。しかし、ファイルのタイプに関係なく、コンフィグレーション ファイルは `weblogic-ejb-jar.xml` の persistence-use 要素で参照しなければなりません。

注意： コンテナ管理による永続性 Bean では、接続プールの最大接続数を 1 より大きい値にコンフィグレーションします。WebLogic Server のコンテナ管理による永続性サービスでは、同時に 2 つの接続を取得しなければならない場合があるからです。

WebLogic Server RDBMS 永続性の使い方

EJB で WebLogic Server RDBMS ベースの永続性サービスを使用するには、次の手順に従います。

1. 専用の XML デプロイメント ファイルを作成します。
2. コンテナ管理による永続性を使用する各 EJB に対して永続性の要素を定義します。
3. デプロイメント記述子ファイルの作成手順については、6-6 ページの「EJB デプロイメント記述子の指定と編集」を参照してください。

WebLogic Server のユーティリティ、DDConverter を使用してこのファイルを作成した場合、ファイルの名前は `weblogic-cmp-rdbms-jar.xml` となります。このファイルをまったく新しく作成する場合は、異なる名前でファイルを保存できます。ただし、`weblogic-ejb-jar.xml` の `persistence-type` および `persistence-use` 要素が適切なファイルを参照していることを確認する必要があります。

`weblogic-cmp-rdbms-jar.xml` は、WebLogic Server RDBMS ベースの永続性サービスを使用して EJB の永続性デプロイメント記述子を定義します。

各 `weblogic-cmp-rdbms-jar.xml` ファイルでは、以下の永続性オプションを定義します。

- EJB 2.0 CMP の EJB 接続プールまたはデータ ソース
- EJB フィールドとデータベース要素のマッピング
- クエリ言語
  - EJB 1.1 CMP 用の WebLogic クエリ言語 (WLQL)
  - EJB 2.0 CMP 用の WebLogic QL 拡張機能付き WebLogic EJB-QL (省略可能)
- ファインダ メソッド定義 (CMP 1.1)
- 関係についての外部キーのマッピング
- クエリ用の WebLogic Server 固有のデプロイメント記述子

EJB 1.1 CMP の RDBMS 永続性用の記述

クライアントはファインダ メソッドを使用して、クエリを実行し、クエリ条件を満たすエンティティ Bean の参照を受け取ることができます。この節では、RDBMS 永続性を使用する WebLogic 固有の 1.1 EJB 用のファインダを作成する方法について説明します。コンテナ管理による永続性を利用する 2.0 EJB のファインダ クエリを定義するには、ポータブルなクエリ言語である EJB QL を使用します。EJB QL の詳細については、5-12 ページの「EJB 2.0 用 EJB QL の使い方」を参照してください。

WebLogic Server では、ファインダを簡単に作成できます。

1. `EJBHome` インタフェースでファインダのメソッド シグネチャを記述します。
2. `ejb-jar.xml` デプロイメント ファイルでファインダのクエリ式を定義します。

`ejbc` は、`ejb-jar.xml` のクエリを使用して、デプロイメント時にファインダ メソッドの実装を作成します。

RDBMS 永続性用のファインダの主要コンポーネントは以下のとおりです。

- `EJBHome` 内のファインダ メソッド シグネチャ
- `ejb-jar.xml` 内で定義される `query` スタンザ
- `weblogic-cmp-rdbms-jar.xml` 内の省略可能な `finder-query` スタンザ

以降の節では、WebLogic Server デプロイメント ファイルの XML 要素を使用して EJB ファインダを記述する方法について説明します。

ファインダ シグネチャ

`find`*`MethodName`*`()` という形式で、ファインダ メソッドのシグネチャを指定します。`weblogic-cmp-rdbms-jar.xml` に定義されるファインダ メソッドは、EJB オブジェクトの Java コレクションまたは単一オブジェクトを返す必要があります。

注意： 関連付けられている EJB クラスの単一オブジェクトを返す `findByPrimaryKey(primkey)` メソッドを定義することもできます。

finder-list スタンザ

`finder-list` スタンザは、`EJBHome` 内の 1 つまたは複数のファインダ メソッド シグネチャを EJB オブジェクトを検索するためのクエリに関連付けます。次に、WebLogic Server RDBMS ベースの永続性を使用した単純な `finder-list` スタンザの例を示します。

```
<finder-list>
     <finder>
          <method-name>findBigAccounts</method-name>
          <method-params>
               <method-param>double</method-param>
          </method-params>
          <finder-query><![CDATA[(> balance $0)]]></finder-query>
     </finder>
</finder-list>
```

注意： `method-param` 要素内に非プリミティブな型を使用する場合には、完全修飾名を指定する必要があります。たとえば、`Timestamp` ではなく `java.sql.Timestamp` を使用します。完全修飾名を使用しないと、デプロイメント ユニットのコンパイル時に `ejbc` でエラー メッセージが生成されます。

finder-query 要素

`finder-query` 要素は、RDBMS から EJB オブジェクトをクエリするための WebLogic クエリ言語 (WLQL) 式を定義します。WLQL は、ファインダ パラメータ、EJB 属性、および Java 言語の式に対して標準の演算子セットを使用します。WLQL の詳細については、5-7 ページの「EJB 1.1 CMP 用の WebLogic クエリ言語 (WLQL) の使用」を参照してください。

注意： `finder-query` 値のテキストは、常に、XML `CDATA` 属性を使用して定義してください。`CDATA` を使用すると、WLQL 文字列中に特殊文字が入っていても、ファインダをコンパイルしたときにエラーが発生しないようになります。

CMP ファインダでは、1 回のデータベース クエリですべての Bean をロードできます。そのため、100 の Bean もデータベースに一度アクセスするだけでロードできます。Bean 管理による永続性 (BMP) ファインダは、データベースに一度アクセスして、ファインダで選択した Bean の主キー値を取得する必要があります。各 Bean にアクセスする場合、通常は Bean がキャッシュされていないことを想定して、もう一度データベースにアクセスする必要があります。したがって、100 の Bean にアクセスするために、BMP はデータベースに 101 回アクセスします。

EJB 1.1 CMP 用の WebLogic クエリ言語 (WLQL) の使用

EJB 1.1 CMP 用の WebLogic クエリ言語 (WLQL) を使用すると、コンテナ管理による永続性を利用する 1.1 エンティティ EJB をクエリできます。`weblogic-cmp-rdbms-jar.xml` ファイルでは、各 `finder-query` スタンザには EJB を返すためのクエリを定義する WLQL 文字列が指定されていなければなりません。EJB 向けの WLQL とそれに対応するデプロイメント ファイル (EJB 1.1 仕様に基づいたもの) を使用します。

注意： 2.0 EJB のクエリについては、5-12 ページの「EJB 2.0 用 EJB QL の使い方」を参照してください。weblogic-ql クエリを使用すると、elb-ql クエリが完全にオーバライドされます。

WLQL 構文

WLQL 文字列では、比較演算子用に次のプレフィックス表記法を使用します。

```
(operator operand1 operand2)
```

追加の WLQL 演算子は、単一のオペランド、テキスト文字列、またはキーワードを受け付けます。

WLQL 演算子

次の表に、有効な WLQL 演算子を示します。

演算子	説明	サンプル構文
=	等しい	(= operand1 operand2)
<	より小さい	(< operand1 operand2)
>	より大きい	(> operand1 operand2)
<=	以下	(<= operand1 operand2)
>=	以上	(>= operand1 operand2)
!	Boolean not	(! operand)
&	Boolean and	(& operand)
|	Boolean or	(| operand)
like	指定された *text_string*、または入力パラメータ中の % 記号に基づいたワイルドカード検索	(like *text_string*%)
isNull	単一オペランドの値が NULL	(isNull operand)
isNotNull	単一オペランドの値が NULL 以外	(isNotNull operand)
orderBy	指定されたデータベース カラムを基準に結果を並び替える **注意：** orderBy 句には永続ファイル名ではなく常にデータベース カラム名を指定する。WebLogic Server は orderBy に指定されたファイル名を変換しない	(orderBy '*column_name*')

演算子	説明	サンプル構文
desc	結果を降順に並び替える。orderBy と組み合わせた場合にのみ使用	(orderBy '*column_name* desc')

WLQL オペランド

有効な WLQL オペランドは以下のとおりです。

- 別の WLQL 式
- weblogic-cmp-rdbms-jar.xml ファイルの別の場所で定義されたコンテナ管理フィールド

  注意： RDBMS カラム名は WLQL のオペランドとして使用できません。代わりに、weblogic-cmp-rdbms-jar.xml の attribute-map に定義されているとおり、RDBMS カラムにマップされる EJB 属性 (フィールド) を使用します。

- $*n* によって識別されるファインダ パラメータまたは Java 式。*n* はパラメータまたは式の数です。デフォルトでは、$*n* は、ファインダ メソッド シグネチャの *n* 番目のパラメータにマップされます。Java 式が組み込まれたより高度な WLQL 式を記述するには、$*n* を Java 式にマップします。

  注意： *$n* という表記法は、1 ではなく 0 で始まる配列に基づいています。たとえば、ファインダの最初の 3 つのパラメータは、$0、$1、および $2 に対応しています。式は個々のパラメータにマップされる必要はありません。高度なファインダは、パラメータより多くの式を定義できます。

WLQL 式の例

次のコード例は、weblogic-cmp-rdbms-jar.xml ファイルから基本的な WLQL 式を使用する部分を抜粋したものです。

- この例では、ファインダに指定された balanceGreaterThan パラメータより大きい balance 属性を持つすべての EJB が返されます。EJBHome 内のファインダ メソッド シグネチャは次のとおりです。

  ```
  public Enumeration findBigAccounts(double balanceGreaterThan)
       throws FinderException, RemoteException;
  ```

  <finder> スタンザの例は次のとおりです。

  ```
  <finder>
       <method-name>findBigAccounts</method-name>
       <method-params>
            <method-param>double</method-param>
       </method-params>
       <finder-query><![CDATA[(> balance $0)]]></finder-query>
  </finder>
  ```

  balance フィールドは、EJB の永続デプロイメント ファイルの中の属性マップに定義する必要があります。

注意： finder-query 値のテキストは、常に、XML CDATA 属性を使用して定義してください。CDATA を使用すると、WLQL 文字列中に特殊文字が入っていても、ファインダをコンパイルしたときにエラーが発生しないようになります。

- 次に、複雑な WLQL 式の使用例を示します。文字列を区切る引用符 (') の使い方にも注意してください。

  ```
  <finder-query><![CDATA[(& (> balance $0) (! (= accountType
  'checking')))]]></finder-query>
  ```

- 次の例では、テーブル内のすべての EJB が検索されます。この例では、次のファインダ メソッド シグネチャを使用しています。

  ```
  public Enumeration findAllAccounts()
       throws FinderException, RemoteException
  ```

  <finder> スタンザの例では、空の WLQL 文字列が使用されています。

  ```
  <finder>
       <method-name>findAllAccounts</method-name>
       <finder-query></finder-query>
  </finder>
  ```

- 次のクエリでは、lastName フィールドが「M」で始まるすべての EJB が検索されます。

  ```
  <finder-query><![CDATA[(like lastName M%)]]></finder-query>
  ```

- 次のクエリでは、null の firstName フィールドを持つすべての EJB が返されます。

  ```
  <finder-query><![CDATA[(isNull firstName)]]></finder-query>
  ```

- 次のクエリでは、balance フィールドが 5000 より大きいすべての EJB が返され、それらの Bean がデータベース カラム id によってソートされます。

  ```
  <finder-query><![CDATA[WHERE >5000 (orderBy 'id' (> balance
  5000))]]></finder-query>
  ```

- 次のクエリは前の例とほぼ同じですが、EJB が降順で返されます。

  ```
  <finder-query><![CDATA[(orderBy 'id desc' (>
  ))]]></finder-query>
  ```

CMP 1.1 ファインダ クエリとしての SQL の使用

WebLogic Server では、標準 WLQL クエリの代わりに SQL 文字列を使用し、CMP 1.1 ファインダ クエリ用の SQL を記述することができます。SQL 文が CMP 1.1 ファインダ クエリとしてデータベースから値を取り出します。複雑なファインダ クエリが必要とされ WLQL を使えない場合に、SQL を使って CMP 1.1 ファインダ クエリを記述します。

WLQL の詳細については、5-7 ページの「EJB 1.1 CMP 用の WebLogic クエリ言語 (WLQL) の使用」を参照してください。

SQL ファインダ クエリを指定するには、次の手順に従います。

1. weblogic-cmp-rdbms-jar.xml ファイルで、finder-sql 要素を使って次のように SQL クエリを記述します。

   findBigAccounts(double cutoff) は次のようになります。

   ```
   <finder-sql><![CDATA{balance >$0]]></finder-sql>
   ```

   SQL 文字列に含まれている「$0」や「$1」といった値は、ファインダ メソッドへのパラメータを参照しています。WebLogic Server コンテナ は「$」をパラメータに置き換えますが、SQL クエリを解釈しようとはしません。

2. コンテナは次のような SQL を発行します。

   ```
   SELECT <columns> FROM table WHERE balance > ?
   ```

   この SQL は、WHERE 句の SQL 文でなければなりません。コンテナが SELECT 句と FROM 句を付加します。WHERE 句の中に任意の SQL が含まれます。

「より大きい (**>**)」や「より小さい (**<**)」のシンボルなど、SQL クエリ内で XML パーサを混乱させる可能性のある文字を使用する場合には、必ず、前述の SQL 文の例のように CDATA 形式を使用して SQL クエリを宣言してください。

注意： SQL クエリ内ではベンダ固有の SQL を際限なく使用できます。

EJB 2.0 用 EJB QL の使い方

EJB クエリ言語 (QL) は、コンテナ管理による永続性を利用する 2.0 エンティティ EJB のファインダ メソッドを定義する移植可能なクエリ言語です。このクエリ言語は SQL に似ており、クエリ内の 1 つまたは複数のエンティティ EJB オブジェクトまたはフィールドを選択する場合に使用します。CMP フィールドはデプロイメント記述子で宣言するので、`findByPrimaryKey()` 以外のすべてのファインダ メソッドのクエリをデプロイメント記述子で作成できます。`findByPrimaryKey` は、コンテナによって自動的に処理されます。EJB QL クエリの検索スペースは、`ejb-jar.xml` (コンテナ管理によるフィールドとその関連データベース カラムの Bean のコレクション) で定義された EJB のスキーマからなります。

EJB 2.0 Bean についての EJB QL の要件

デプロイメント記述子では、EJB QL クエリ文字列を使用して、EJB 2.0 のエンティティ Bean の各ファインダ クエリを定義する必要があります。WebLogic Query Language (WLQL) を EJB 2.0 エンティティ Bean で使用することはできません。WLQL は、EJB 1.1 CMP で使用することを想定しています。

WLQL から EJB QL への移行

以前のバージョンの WebLogic Server を使用したことがあれば、コンテナ管理によるエンティティ EJB ではファインダ メソッド用に WLQL を使用できます。この節では、WLQL の一般的な処理についてのクイック リファレンスを提供します。WLQL の構文と EJB QL の構文の対応については、次の表を参考にしてください。

WLQL のサンプル構文	対応する EJB QL の構文
`(= operand1 operand2)`	WHERE operand1 = operand2
`(< operand1 operand2)`	WHERE operand1 < operand2
`(> operand1 operand2)`	WHERE operand1 > operand2
`(<= operand1 operand2)`	WHERE operand1 <= operand2
`(>= operand1 operand2)`	WHERE operand1 >= operand2
`(! operand)`	WHERE NOT operand
`(& expression1 expression2)`	WHERE expression1 AND expression2
`(| expression1 expression2)`	WHERE expression1 OR expression2
`(like` *`text_string`*`%)`	WHERE operand LIKE '*`text_string`*`%'`
`(isNull operand)`	WHERE operand IS NULL
`(isNotNull operand)`	WHERE operand IS NOT NULL

EJB QL の EJB 2.0 WebLogic QL 拡張機能の使い方

WebLogic Server には、標準の EJB QL の拡張であり、SQL に似た WebLogic QL という言語が用意されています。この言語はファインダ式と連携し、RDBMS の EJB オブジェクトのクエリ用に使用されます。query は、weblogic-ql 要素を使用して、weblogic-cmp-rdbms-jar.xml デプロイメント記述子に定義します。

ejb-jar ファイルには、weblogic-cmp-rdbms-jar.xml ファイルの weblogic-ql 要素に対応するクエリ要素が必要です。ただし、weblogic-cmp-rdbms-jar.xml のクエリ要素は、ejb-jar.xml のクエリ要素をオーバライドします。

upper 関数と lower 関数

EJB WebLogic QL upper および lower 拡張機能は、大文字と小文字の違いがある以外は検索式の文字と一致する結果をファインダ メソッドが返せるように、引数の大文字と小文字を変換します。大文字と小文字の変換は文字列を照合するための一時的なものなので、データベース内に永続しません。基底のデータベースも、upper および lower 関数をサポートしている必要があります。

upper 関数は、文字列の照合の前に、引数内の文字をすべて大文字に変換します。クエリ内で大文字で表された式で upper 関数を使用すると、大文字であるか小文字であるかに関係なく、式に一致するすべての項目が返されます。たとえば、次のように入力します。

```
select name from products where upper(name)='DETERGENT';
```

lower 関数は、文字列の照合の前に、引数内の文字をすべて小文字に変換します。クエリ内で小文字で表された式で lower 関数を使用すると、大文字であるか小文字であるかに関係なく、式に一致するすべての項目が返されます。

```
select type from products where lower(name)='domestic';
```

注意： upper および lower 拡張機能は、WebLogic Server 7.0 SP03 で追加されました。

SELECT DISTINCT の使用

EJB WebLogic QL 拡張機能の SELECT DISTINCT では、重複したクエリをフィルタ処理するようデータベースに指示します。SELECT DISTINCT を EJB QL クエリで指定すると、重複した結果をソートするために EJB コンテナのリソースが使用されません。

EJB 2.0 CMP Bean の `weblogic-ql` 要素の XML スタンザで値を `TRUE` に設定して `sql-select-distinct` 要素を指定すると、作成されるデータベース クエリの SQL STATEMENT には DISTINCT 句が含まれます。

`sql-select-distinct` 要素は `weblogic-cmp-rdbms-jar.xml` ファイルで指定します。ただし、Oracle データベースでアイソレーション レベルを `READ_C0MMITED_FOR_UPDATE` に指定してある場合、`sql-select-distinct` を指定することはできません。Oracle 上では、クエリに `sql-select-distinct` と `READ_C0MMITED_FOR_UPDATE` の両方を指定することができないからです。このアイソレーション レベルをセッション Bean などで使用する可能性がある場合、`sql-select-distinct` 要素を使用しないでください。

ORDERBY の使用

WebLogic クエリ言語 (WL QL) の拡張機能である ORDERBY は、ファインダ メソッドと連携して、選択における CMP フィールドの選択順序を指定するキーワードです。

図 5-1 id による順序付けを指定する WebLogic QL ORDERBY 拡張機能

```
ORDERBY
       SELECT OBJECT(A) from A for Account.Bean
              ORDERBY A.id
```

注意： ORDERBY は、すべてのソート処理を DBMS に委ねます。このため、取得される結果の順序は、実行中の Bean の基盤となる特定の DBMS によって異なります。

また、次のように ORDERBY に昇順 [ASC] か降順 [DESC] か指定することもできます。

図 5-2 id による順序付けを指定する WebLogic QL ORDERBY 拡張機能 (昇順 / 降順指定)

```
ORDERBY <field> [ASC|DESC], <field> [ASC|DESC]
     SELECT OBJECT(A) from A for Account.Bean, OBJECT(B) from B
     for Account.Bean
          ORDERBY A.id ASC; B.salary DESC
```

サブクエリの使用

WebLogic Server では、EJB QL のサブクエリの次のような機能をサポートしています。

- サブクエリの戻り値の型
  - 単一の `cmp-field`
  - 集約関数
  - 単純主キーを持つ Bean
- 比較オペランドとしてのサブクエリ
- 相関サブクエリ
- 非相関サブクエリ
- サブクエリでの DISTINCT 句

WebLogic QL とサブクエリの関係は、SQL クエリとサブクエリの関係に似ています。WebLogic QL のサブクエリは、外部 WebLogic QL クエリの WHERE 句内で使用してください。若干の例外はありますが、サブクエリの構文は WebLogic QL クエリのものとほぼ同じです。

WebLogic QL を指定する手順については、5-14 ページの「EJB QL の EJB 2.0 WebLogic QL 拡張機能の使い方」を参照してください。この手順に従って、SELECT 文で次の例のようにサブクエリを指定します。

次のクエリは、成績順位をもとに平均以上の生徒を選択しています。

```
SELECT OBJECT(s) FROM studentBean AS s WHERE s.grade > (SELECT
AVG(s2.grade) FROM StudentBean AS s2)
```

上記クエリの例の中で、サブクエリ、「`SELECT AVG(s2.grade) FROM StudentBean AS s2`」が EJB QL クエリと同じ構文を備えている点に注意してください。

サブクエリをネストすることもできます。この深さは、基盤データベースのネストの許容範囲によって制限されます。

WebLogic QL クエリでは、メイン クエリとそのすべてのサブクエリの FROM 句で宣言される識別子がユニークでなければなりません。これはつまり、サブクエリの内側で前にローカルに宣言した識別子をそのサブクエリで再び宣言することはできないということです。

たとえば、次の例は無効です。Employee Bean がクエリとサブクエリの双方で emp として宣言されています。

```
SELECT OBJECT(emp)
       FROM EmployeeBean As emp
              WHERE emp.salary=(SELECT MAX(emp.salary) FROM
                     EmployeeBean AS emp WHERE employee.state=MA)
```

このクエリは次のように記述すべきです。

```
SELECT OBJECT(emp)
       FROM EmployeeBean As emp
              WHERE emp.salary=(SELECT MAX(emp2.salary) FROM
                     EmployeeBean AS emp2 WHERE emp2.state=MA)
```

この例では、サブクエリの Employee Bean がメインクエリの Employee Bean と異なる識別子になるように正しく宣言されています。

サブクエリの戻り値の型

WebLogic QL サブクエリの戻り値の型は、次のような各種の型のいずれかになります。

単一の cmp-field 型のサブクエリ

WebLogic Server は、`cmp-field` で構成されている戻り値の型をサポートしています。サブクエリから返される結果は、1 つの値または値の集合から構成されている可能性があります。`cmp-field` 型の値を返すサブクエリの例を次に示します。

```
SELECT emp.salary FROM EmployeeBean AS emp WHERE emp.dept =
'finance'
```

このサブクエリは、財務部門の従業員の給料すべてを選択しています。

集約関数

WebLogic Server は、ある cmp-field に対する集約から構成されている戻り値型をサポートしています。集約は必ず 1 つの値から構成されるので、ここで返される値も常に 1 つの値になります。cmp-field の集約 (MAX) の型の値を返すサブクエリの例を次に示します。

```
SELECT MAX(emp.salary) FROM EmployeeBean AS emp WHERE emp.state=MA
```

このサブクエリは、マサチューセッツで最高額の給料を 1 つだけ選択しています。

集約関数の詳細については、5-22 ページの「集約関数の使用」を参照してください。

単純主キーを持つ Bean

WebLogic Server は、単純主キーを持つ 1 つの cmp-bean で構成されている戻り値の型をサポートしています。

注意： 複合主キーを持つ Bean はサポートされていません。複合主キーを持つ Bean をサブクエリの戻り値の型として指定しようとしても、クエリをコンパイルする時点で失敗に終わります。

単純主キーを持つ Bean 型の値を返すサブクエリの例を次に示します。

```
SELECT OBJECT(emp) FROM EMployeeBean As emp WHERE
emp.department.budget>1,000,000
```

このサブクエリは、$1,000,000 以上の予算がある部門の全従業員のリストを返します。

比較オペランドとしてのサブクエリ

サブクエリを比較演算子のオペランドとして使用します。WebLogic QL では、比較演算子 ([NOT]IN、[NOT]EXISTS) および算術演算子 (<、>、<=、>=、=、および ANY や ALL と使用する <>) のオペランドをサブクエリとしてサポートしています。

[NOT] IN

[NOT] IN 比較演算子は、左側のオペランドが右側のサブクエリ オペランドのメンバーかどうか検査します。

サブクエリが NOT IN オペレータの右側のオペランドとなっている例を次に示します。

```
SELECT OBJECT(item)
       FROM ItemBean AS item
              WHERE item.itemId NOT IN
                     (SELECT oItem2.item.itemID
                                  FROM OrderBean AS orders2,
IN(orders2.orderItems)oIttem2
```

サブクエリは、すべての注文から品目すべてを選択しています。

メイン クエリの NOT IN 演算子では、サブクエリによって返された集合の中に含まれていない品目すべてを選択しています。したがって、最終的に、メインクエリが未注文の品目すべてを選択します。

[NOT] EXISTS

[NOT] EXISTS 比較演算子は、サブクエリ オペランドによって返された集合が空かどうか検査します。

サブクエリが NOT EXISTS オペランドのオペランドとなっている例を次に示します。

```
SELECT (cust) FROM CustomerBean AS cust
       WHERE NOT EXISTS
              (SELECT order.cust_num FROM OrderBean AS order
                     WHERE cust.num=order_num)
```

これは、相関サブクエリを用いたクエリの一例となっています。詳細については、5-21 ページの「相関サブクエリと非相関サブクエリ」を参照してください。このクエリは、注文していない顧客をすべて返します。

```
SELECT (cust) FROM CustomerBean AS cust
       WHERE cust.num NOT IN
              (SELECT order.cust_um FROM OrderBean AS order
                     WHERE cust.num=order_num)
```

算術演算子

右側のサブクエリ オペランドが 1 つの値を返す場合、比較の算術演算子を使用できます。右側のサブクエリが複数の値を返す場合には、サブクエリの前に ANY または ALL 修飾子が必要です。

「=」演算子を使用したサブクエリの例を次に示します。

```
SELECT OBJECT (order)
       FROM OrderBean AS order, IN(order.orderItems)oItem
              WHERE oItem.quantityOrdered =
                     (SELECT MAX (subOItem.quantityOrdered)
                            FROM Order ItemBean AS subOItem
                                   WHERE subOItem,item itemID = ?1)
AND oItem.item.itemId = ?1
```

特定の品目 ID に対し、サブクエリはその品目の注文の最高数を返します。サブクエリが返している集合が、「=」演算子に必要な 1 つの値となっている点に注意してください。

メイン クエリの「=」演算子では、これと同じ品目 ID に対し、どの注文の注文品目の注文量がサブクエリから返された注文の最高数に等しいかを調べています。最終的に、特定の品目について注文が最高数である OrderBean をクエリが返します。

右側のサブクエリ オペラントが複数の値を返す場合には、算術演算子と ANY または ALL と組み合わせて使用します。

ANY または ALL を使用したサブクエリの例を次に示します。

```
SELECT OBJECT (order)
       FROM OrderBean AS order, IN(order.orderItems)oItem
              WHERE oItem.quantityOrdered > ALL
                     (SELECT subOItem.quantityOrdered
                            FROM OrderBean AS suborder IN
(subOrder.orderItems)subOItem
                                   WHERE subOrder,orderId = ?1)
```

サブクエリは、特定の注文 ID に対し、その注文 ID で注文された各品目の注文数の集合を返します。メイン クエリの「>」ALL 演算子は、各品目の注文数がサブクエリから返された集合内のすべての値を上回っている注文すべてを探しています。最終的にメインクエリは、入力された注文に対して、全品目で注文数を上回っている注文すべてを返します。

サブクエリが複数の値を持つ結果を返す可能性があるため、「>」演算子ではなく、「>」ALL 演算子が使用されている点に注意してください。

<、>、<=、>=、=、<> といったすべての算術演算子がこのように使用されます。

相関サブクエリと非相関サブクエリ

WebLogic Server は、相関サブクエリと非相関サブクエリの双方をサポートしています。

非相関サブクエリ

非相関サブクエリは、その外部クエリとは独立に評価されます。非相関サブクエリの例を次に示します。

```
SELECT OBJECT(emp) FROM EmployeeBean AS emp
       WHERE emp.salary>
       (SELECT AVG(emp2.salary) FROM EmployeeBean AS emp2)
```

この非相関サブクエリの例では、平均以上の給与の従業員を選択しています。この例では、算術演算子「>」を使用します。

相関サブクエリ

相関サブクエリは、その外部クエリの値がサブクエリでの評価に関与するサブクエリです。相関サブクエリの例を次に示します。

```
SELECT OBJECT (mainOrder) FROM OrderBean AS mainOrder
       WHERE 10>
             (SELECT COUNT (DISTINCT subOrder.ship_date)
       FROM OrderBean AS subOrder
       WHERE subOrder.ship_date>mainOrder.ship_date
       AND mainOrder.ship_date IS NOT NULL
```

この相関サブクエリの例では、最後に出荷された 10 個の注文を選択しています。NOT IN 演算子を使用しています。

注意： 相関クエリでは、非相関クエリより処理オーバーヘッドが大きくなる可能性があることに注意してください。

サブクエリでの DISTINCT 句

サブクエリ内で DISTINCT 句を使用すると、そのサブクエリで生成される SQL 内で SQL SELECT DISTINCT を使うことができます。サブクエリでの DISTINCT 句の使い方は、メインクエリでの使い方とは異なります。メインクエリ内の DISTINCT 句は EJB コンテナによって施されますが、サブクエリ内の DISTINCT 句は生成された SQL の SQL SELECT DISTINCT によって施されます。サブクエリ内に DISTINCT 句が含まれる例を次に示します。

```
SELECT OBJECT (mainOrder) FROM OrderBean AS mainOrder
       WHERE 10>
              (SELECT COUNT (DISTINCT subOrder.ship_date)
       FROM OrderBean AS subOrder
       WHERE subOrder.ship_date>mainOrder.ship_date
       AND mainOrder.ship_date IS NOT NULL
```

この例では、最後に出荷された 10 個の注文を選択しています。

集約関数の使用

WebLogic Server では、WebLogic QL の集約関数をサポートしています。これらは、WHERE 句のようなクエリの一部ではなく、SELECT 句の対象として使われるのみです。集約関数の振る舞いは SQL 関数と似ています。これらの関数は、クエリの WHERE 条件によって返される Bean の外側で評価されます。

WebLogic QL を指定する手順については、5-14 ページの「EJB QL の EJB 2.0 WebLogic QL 拡張機能の使い方」を参照してください。これに従って、次の表のサンプルのように、集約関数を指定した SELECT 文を記述します。

サポートされている関数とサンプル文のリストを次に示します。

集約関数	説明	サンプル文
MIN(x)	このフィールドの最小値を返す。	SELECT MIN(t.price) FROM TireBean AS t WHERE t.size=?1 この文では、入力された特定のサイズの最安値を選択している。

集約関数	説明	サンプル文
MAX(x)	このフィールドの最大値を返す。	SELECT MAX(s.customer_count) FROM SalesRepBean AS s WHERE s.city='Los Angeles' この文では、ロサンゼルス市内の販売代理人それぞれが担当する顧客数の中の最大値を選択している。
AVG([DISTINCT] x)	このフィールドの平均値を返す。	SELECT AVG(b.price) FROM BookBean AS b WHERE b.category='computer_science' この文では、コンピュータ科学分野の書籍の平均価格を選択している。
SUM([DISTINCT] x)	このフィールドの合計を返す。	SELECT SUM(s.customer_count) FROM SalesRepBean AS s WHERE s.city='Los Angeles' この文では、ロスアンゼルス市内の販売代理人が担当している顧客の合計数を取得している。
COUNT([DISTINCT] x)	フィールドの発生数を返す。	SELECT COUNT(s.deal.amount) FROM SalesRepBean AS s, IN(deal)s WHERE s.deal.status='closed' AND s.deal.amount>=1000000 この文では、100 万ドル以上の非公開取引の数を取得している。

次の手順で、ResultSet として集約関数を返すことができます。

ResultSet を返すクエリの使用

WebLogic Server は、複数カラム クエリの結果を `java.sql.ResultSet` の形式で返す、`ejbSelect()` クエリをサポートしています。この機能を支援するために、WebLogic Server では、次のように SELECT 句の対象フィールドをカンマで区切って指定できるようになりました。

```
SELECT emmp.name, emp.zip FROM EmployeeBean AS emp
```

このクエリは、従業員の名前と郵便番号の値をカラムとしその数行を含んだ java.sqlResultSet を返します。

WebLogic QL を指定する手順については、5-14 ページの「EJB QL の EJB 2.0 WebLogic QL 拡張機能の使い方」を参照してください。5-14 ページの「EJB QL の EJB 2.0 WebLogic QL 拡張機能の使い方」の手順に従って、上記のクエリの例で WebLogic QL を指定したように、ResultSet を指定するクエリを記述します。また、同様にこれに従って、下記の表のサンプルで示すように、集約クエリを指定した SELECT 文を記述します。

EJB QL で作成される ResultSet は、`cmp-field` の値、または、`cmp-field` 値の集合だけを返します。Bean を返すことはできません。

さらに、`cmp-field` と集約関数を組み合わせた場合、下記のサンプルに示すような強力なクエリを作成できます。

次の行 (Bean) は、各地区の従業員の給与を示しています。

カリフォルニア市内の従業員の給与を示す CMP フィールド

名前	地区	給与
Matt	CA	110,000
Rob	CA	100,000

アリゾナ市内の従業員の給与を示す CMP フィールド

名前	地区	給与
Dan	AZ	120,000
Dave	AZ	80,000

テキサス市内の従業員の給与を示す CMP フィールド

名前	地区	給与
Curly	TX	70,000
Larry	TX	180,000
Moe	TX	80,00

注意： 各行が 1 つの Bean を表します。

次の SELECT 文のクエリでは、ResultSet、集約関数 (AVG) とともに、GROUP BY 文と ORDER BY 文を使用し、複数カラム クエリの結果を降順ソートを使って取り出します。

```
SELECT e.location, AVG(e.salary)
       FROM Finder EmployeeBean AS e
              GROUP BY e.location
                     ORDER BY 2 DESC
```

このクエリは、各地区の従業員の平均給与を降順で示します。2 という数字は、ORDERBY ソートを SELECT 文の 2 番目の項目で実行することを意味しています。GROUP BY 句は、e.location 属性に一致する従業員の平均給与を指定しています。

ResultSet は、次のように降順で並んでいます。

地区	平均
TX	110,000
AZ	100,000
CA	105,000

注意： ResultSet を返すクエリ内で ORDERBY の引数として使用できるのは整数だけです。WebLogic Server では、Bean を返すファインダ、または `ejbselect()` 内で ORDERBY の引数として整数を使用することはできません。

Query インタフェースのプロパティベース メソッド

Query インタフェースには、検索メソッドと実行メソッドがあります。検索メソッドは、EJBObject を返すという点で標準の EJB メソッドと似ています。実行メソッドは、個々のフィールドを選択できるという点で、Select 文に似ています。

Query インタフェースの戻り値の型は、切断された ResultSet です。つまり、ResultSet がオープンなデータベース接続を保持していない点を除き、ResultSet の情報にアクセスするのと同じように、返されたオブジェクトの情報にアクセスするということです。

Query インタフェースのプロパティベース メソッドを使用すると、クエリに特有の設定を別の方法で指定できます。QueryProperties インタフェースは標準の EJB クエリ設定を保持し、WLQueryProperties インタフェースは WebLogic 固有のクエリ設定を保持します。

Query インタフェースは QueryProperties を拡張しますが、実際の Query 実装は WLQueryProperties を拡張したものなので、フィールド グループを設定する図 5-3 の例のように、安全にキャストできます。

図 5-3 WLQueryProperties によるフィールド グループの設定

```
Query query=qh.createQuery(); ((WLQueryProperties)
query).setFieldGroupName("myGroup"); Collection
results=query.find(ejbql);
```

または

```
Query query=qh.createQuery(); Properties props = new Properties();
props.setProperty(WLQueryProperties.GROUP_NAME, "myGroup");
Collection results=query.find(ejbql, props);
```

動的クエリの使用

動的クエリを使用すると、各自のアプリケーション コードでプログラム的にクエリを作成したり実行できるようになります。たとえば、クエリを使って RDBMS に対して EJB オブジェクトの情報を要求できます。この機能は、EJB 2.0 CMP Bean でのみ利用できます。動的クエリの使用には、次のような利点があります。

- EJB を更新してデプロイせずに、新しいクエリを作成して実行できます。
- EJB デプロイメント記述子ファイルのサイズを縮小できます。ファインダ クエリがデプロイメント記述子内で静的に定義される代わりに、動的に作成されるようになるためです。

動的クエリの有効化

動的クエリを有効化するには、次の手順に従います。

1. その EJB の `weblogic-ejb-jar.xml` デプロイメント記述子ファイルの `enable-dynamic-queries` 要素を次のように指定します。

   ```
   <enable-dynamic-queries>True</enable-dynamic-queries>
   ```

2. `enable-dynamic-queries` 要素の追加や作成の手順については、6-6 ページの「EJB デプロイメント記述子の指定と編集」を参照してください。

3. `ejb-jar.xml` デプロイメント記述子ファイルの `method-permission` 要素を指定することによって、標準メソッド権限を設定し、動的クエリへのアクセスを制御します。

   `weblogic.ejb.QueryHome` インタフェースの `createQuery()` メソッドの `method-permission` を設定することによって、動的クエリを実行する `weblogic.ejb.Query` オブジェクトへのアクセスを制御します。

   `createQuery()` メソッドの `method-permission` を指定した場合、`method-permission` の設定が `Query` クラスの実行と検索メソッドに適用されます。

動的クエリの実行

次のコード例では、動的クエリをどのように実行するかを示します。

```
InitialContext ic=new InitialContext();
FooHome fh=(FooHome)ic.lookup("fooHome");
QueryHome qh=(QueryHome)fh;
Sring ejbql="SELECT OBJECT(e)FROM EmployeeBean e WHERE
e.name='rob'"
Query query=qh.createQuery();
query.setMaxElements(10)
Collection results=query.find(ejbql);
```

Oracle の SELECT HINT の使用

WebLogic Server は、INDEX の使い方に関するヒントを Oracle Query オプティマイザに渡すことを可能にする EJB QL 拡張機能をサポートしています。この拡張機能を使用すると、データベース エンジンにヒントを提供できます。たとえば、検索先のデータベースが ORACLE_SELECT_HINT によって恩恵を受けることがわかっている場合は、ANY 文字列値を取り、その文字列値をデータベースに対するヒントとして SQL SELECT 文の後に挿入する ORACLE_SELECT_HINT 句を定義します。

このオプションを使用するには、この機能を使用するクエリを `weblogic-ql` 要素で宣言します。この要素は、`weblogic-cmp-rdbms-jar.xml` ファイルに入っています。`weblogic-ql` 要素では、EJB-QL に対する WebLogic 固有の拡張機能を含むクエリを指定します。

WebLogic QL のキーワードおよび使い方は次のとおりです。

```
SELECT OBJECT(a) FROM BeanA AS a WHERE a.field > 2 ORDERBY a.field
SELECT_HINT '/*+ INDEX_ASC(myindex) */'
```

この文は、Oracle のオプティマイザ ヒントを使用して次の SQL を生成します。

```
SELECT /*+ INDEX_ASC(myindex) */ column1 FROM .... (etc)
```

WebLogic QL ORACLE_SELECT_HINT 句では、単一引用符で囲まれた部分 (' ') が SQL SELECT の後に挿入されます。クエリ作成者は、引用符内のデータを Oracle データベースが確実に認識できるものにする必要があります。

「get」および「set」メソッドの制限

WebLogic Server では、コンテナ管理によるフィールドの読み出しおよび修正にコンテナ生成のアクセサ メソッドを使用します。それらのメソッドの名前は `get` または `set` で始まり、`ejb-jar.xml` で定義されている永続フィールドの実際の名前を使用します。これらのメソッドは、`public`、`protected`、および `abstract` として宣言されます。

Oracle DBMS の BLOB および CLOB DBMS カラムのサポート

WebLogic Server は、Oracle Binary Large Object (BLOB) および Character Large Object (CLOB) DBMS カラムを EJB CMP でサポートしています。BLOB および CLOB は、大きなオブジェクトを効率的に保存したり、検索したりするためのデータ型です。CLOB は文字オブジェクトで、BLOB は大きなバイト配列に変換される画像などのバイナリまたはシリアライズ可能オブジェクトです。

BLOB および CLOB は、文字列変数である `OracleBlob` または `OracleClob` の値を BLOB または CLOB カラムにマップします。WebLogic Server では、BLOB をバイト配列またはシリアライズ可能なオブジェクトにマップし、CLOB をデータ型 `java.lang.string` にマップします。現時点では、`char` 配列を CLOB カラムにマップすることはできません。

BLOB/CLOB サポートを有効にするには次の手順に従います。

1. Bean クラスで変数を宣言します。

2. weblogic-cmp-rdbms jar.xml ファイルで dbms-column-type デプロイメント記述子を宣言して XML を編集します。
3. Oracle データベースに BLOB または CLOB を作成します。

BLOB/CLOB オブジェクトのサイズが大きいため、BLOB または CLOB を使用すると、パフォーマンスが低下する場合があります。

デプロイメント記述子による BLOB の指定

次の XML コードは、weblogic-cmp-rdbms-jar-xml ファイルの dbms-column 要素を使用して BLOB オブジェクトを指定する方法を示しています。

図 5-4 BLOB オブジェクトの指定

```
<field-map>
       <cmp-field>photo</cmp-field>
       <dbms-column>PICTURE</dbms-column>
       <dbms_column-type>OracleBlob</dbms-column-type>
</field-map>
```

デプロイメント記述子による CLOB の指定

次の XML コードは、weblogic-cmp-rdbms-jar-xml ファイルの dbms-column 要素を使用して CLOB オブジェクトを指定する方法を示しています。

図 5-5 CLOB オブジェクトの指定

```
<field-map>
       <cmp-field>description</cmp-field>
       <dbms-column>product_description</dbms-column>
       <dbms_column-type>OracleClob</dbms-column-type>
</field-map>
```

WebLogic Server での EJB 1.1 CMP の調整更新

コンテナ管理 EJB が読み書きされるときに、コンテナは get および set コールバックを受け取るので、EJB のコンテナ管理による永続性 (CMP) は、自動的に調整更新をサポートします。EJB 1.1 CMP Bean を調整すると、パフォーマンスの向上に役立ちます。

WebLogic Server は、EJB 1.1 CMP の調整更新をサポートするようになりました。ejbStore が呼び出されると、EJB コンテナはコンテナ管理フィールドがトランザクションで変更されたかどうかを自動的に判定します。変更されたフィールドだけがデータベースに書き込まれます。変更されたフィールドがない場合、データベースは更新されません。

以前のバージョンの WebLogic Server では、CMP 1.1 Bean が変更されたかどうかをコンテナに通知する isModified メソッドを記述することができました。isModified は現在も WebLogic Server でサポートされていますが、isModified メソッドを使用しないで、更新されたフィールドをコンテナに判定させることをお勧めします。

この機能は EJB 2.0 CMP に対してデフォルトで有効です。EJB CMP 1.1 の調整更新を有効にするには、weblogic-cmp-rdbms-jar.xml ファイルの次のデプロイメント記述子要素を true に設定します。

```
<enable-tuned-updates>true</enable-tuned-updates>
```

CMP の調整更新を無効にするには、このデプロイメント記述子要素を次のように設定します。

```
<enable-tuned-updates>false</enable-tuned-updates>
```

この場合、ejbStore は常にすべてのフィールドをデータベースに書き込みます。

CMP 2.0 エンティティ Bean 向けに最適化されたデータベース更新

CMP 2.0 エンティティ Bean の場合、setXXX() メソッドでは、変更されていないプリミティブの値と、変更不能なフィールドの値をデータベースに書き込みません。この最適化により、特に大量のデータベース トランザクションを伴うアプリケーションにおいて、パフォーマンスが向上します。

CMP キャッシュのフラッシュ

トランザクションによる更新内容は、トランザクションで発行されたクエリ、ファインダ、および ejbSelect の結果に反映させる必要があります。この要件に従うとパフォーマンスが低下する場合があるので、新しいオプションにより、Bean に関するクエリを実行する前にキャッシュをフラッシュするように指定することができます。

このオプションが無効の場合 (デフォルト設定)、現在のトランザクションの結果はクエリに反映されません。このオプションを有効にした場合、コンテナはキャッシュされているトランザクションの変更をすべてデータベースに書き込んでから新しいクエリを実行します。この方法により、変更が結果に表示されます。

このオプションを有効にするには、weblogic-cmp-rdbms-jar.xml ファイルで include-updates 要素を true に設定します。

図 5-6 トランザクションの結果をクエリに反映するための指定

```
<weblogic-query>
       <query-method>
       <method-name>findBigAccounts</method_name>
       <method-params>
              <method-param>double</method-param>
       </method-params>
       </query-method>
       <weblogic-ql>WHERE BALANCE>10000 ORDERBY NAME</weblogic-ql>
       <include-updates>true</include-updates>
</weblogic-query>
```

デフォルトは `false` で、この設定は最大限のパフォーマンスを実現します。キャッシュされているトランザクションに対して行われる更新はクエリの結果に反映され、変更はデータベースには書き込まれず、クエリの結果に変更は見られません。

この機能を使用するかどうかは、データを最新かつ一貫性のあるものにしておくことよりもパフォーマンスを重視するかどうかで判断します。

主キーの使用

主キーとは、エンティティ Bean をそのホーム内でユニークに識別するオブジェクトです。コンテナでは、エンティティ Bean の主キーを操作できなければなりません。個々のエンティティ Bean クラスは、その主キーに対して別のクラスを定義できますが、複数のエンティティ Bean クラスが同じ主キー クラスを使用できます。主キーは、エンティティ Bean のデプロイメント記述子で指定されます。コンテナ管理の永続性を利用するエンティティ Bean の主キー クラスを指定するには、エンティティ Bean クラスの 1 つまたは複数のフィールドに主キーをマップします。

すべてのエンティティ オブジェクトはそのホーム内で一意な ID を持ちます。2 つのエンティティ オブジェクトのホームと主キーが共通する場合、両者は同一のものと見なされます。クライアントはエンティティ オブジェクトのリモート インタフェースへの参照に対して `getPrimaryKey()` メソッドを呼び出して、そのホーム内でのエンティティ オブジェクトの ID を調べることができます。参照と関連付けられたオブジェクト ID は、参照が有効な間は変化しません。したがって `getPrimaryKey()` メソッドは、同じエンティティ オブジェクトの参照に対して呼び出されたときは常に同じ値を返します。エンティティ オブジェクトの主キーを知っているクライアントは、Bean のホーム インタフェースの `findByPrimaryKey(key)` メソッドを呼び出すことによって、エンティティ オブジェクトへの参照を取得することができます。

1 つの CMP フィールドにマップされた主キー

エンティティ Bean クラスでは、主キーを 1 つの CMP フィールドにマップできます。ejb-jar.xml ファイルのデプロイメント記述子、primkey-field 要素を使用して、主キーであるコンテナ管理フィールドを指定します。
prim-key-class 要素は、主キー フィールドのクラスでなければなりません。

1 つまたは複数の CMP フィールドをラップする主キー クラス

主キー クラスを 1 つまたは複数のフィールドにマップすることができます。主キー クラスは public でなければならず、パラメータを付けない public コンストラクタを持たなければなりません。ejb-jar.xml ファイルのデプロイメント記述子、prim-key-class 要素を使用して、エンティティ Bean の主キー クラスの名前を指定します。このデプロイメント記述子要素にはクラス名だけを指定できます。主キー クラス内のすべてのフィールドは public として宣言する必要があります。クラス内のフィールドは、ejb-jar.xml ファイルの主キー フィールドと同じ名前を持たなければなりません。

無名主キー クラス

エンティティ Bean が無名主キークラスを使用する場合、EJB をサブクラス化し java.lang.Integer 型の cmp-field をそのサブクラスに追加する必要があります。そのフィールドの自動主キー生成を有効にし、コンテナがフィールドの値を自動的に埋め、そのフィールドを weblogic-cmp-rdbms-jar.xml デプロイメント記述子のデータベース カラムにマップできるようにします。

最終的に、ejb-jar.xml ファイルを更新して元の EJB クラスではなく EJB サブクラスを指定するようにし、その Bean を WebLogic Server にデプロイします。

元の EJB を無名主キー クラスとともに使用した場合、デプロイメント時に WebLogic Server から次のエラー メッセージが表示されます。

EJB の *ejb_name* では、「不明な主キー クラス (<prim-key-class> == java.lang.Object)」をデプロイメント時に (java.lang.Object 以外のオブジェクトとして) 指定する必要があります。

主キーの使用に関するヒント

WebLogic Server で主キーを使用する場合のヒントをいくつか挙げておきます。

- 主キー クラスをコンテナ管理フィールドにしないでください。

  ejbCreate は主キー クラスを戻り値の型として指定していますが、以下の制約があります。

- ejbCreate を使用して新しい主キー クラスを作成しない。代わりに、コンテナが主キー クラスを内部的に作成できるようにします。

- ejbCreate メソッド内で setXXX メソッドを使用して、主キーの cmp-field の値を設定します。

- BigDecimal 型の CMP フィールドは、CMP Bean の主キー フィールドとして使用しないでください。boolean BigDecimal.equals (object x) メソッドでは、値とスケールが同じ場合に限り 2 つの BigDecimal が等しいと判断されます。なぜなら、Java 言語とさまざまなデータベースでは精度に違いがあるからです。たとえば、このメソッドでは 7.1 と 7.10 は等しいとは判断されません。したがって、このメソッドを使用すると、ほとんどの場合で false が返されるか、CMP Bean でエラーが発生します。

  BigDecimal を主キーとして使用する必要がある場合は、次の操作を行ってください。

  a. 主キー クラスを実装します。

  b. この主キー クラスで、boolean equal (Object x) メソッドを実装します。

  c. equal メソッドで、boolean BigDecimal.compareTo(BigDecimal val) を使用します。

データベース カラムへのマッピング

WebLogic Server では、データベース カラムを cmp-field と cmr-field に同時にマップすることができます。その場合、cmp-field は読み込み専用となります。cmp-field が主キー フィールドの場合、cmp-field に対して setXXX メソッドを使うことによって create() メソッドが呼び出されたときにフィールドの値が設定されるように指定します。

EJB 2.0 CMP に対する自動主キー生成

WebLogic Server は、コンテナ管理による永続性 (CMP) 用の自動主キー生成機能をサポートしています。

注意： この機能は EJB CMP 2.0 コンテナに対してのみサポートされており、EJB CMP 1.1 に対する自動主キー生成機能はサポートされていません。1.1 Bean の場合は、Bean 管理による永続性 (BMP) を使用する必要があります。

生成されるキーのサポートは以下の 2 つの方法で提供されます。

- **DBMS 主キー生成の使用。**コンパイル時に指定された一連のデプロイメント記述子を使用して、サポートされているデータベースと連携してキー生成をサポートするためのコンテナ コードを生成します。

  このオプションでは、コンテナはすべてのキー生成を基底のデータベースに委ねます。この機能を有効にするには、サポートされている DBMS の名前と、データベースで必要な場合にはジェネレータ名を指定します。CMP コードは、この機能を実装するすべての細部を処理します。

  この機能の詳細については、5-37 ページの「Oracle 用主キー サポートの指定」と 5-38 ページの「Microsoft SQL Server 用主キー サポートの指定」を参照してください。

- **Bean プロバイダが指定した命名済シーケンス テーブルの使用。**WebLogic Server で指定されたスキーマを持ち、ユーザが作成して名前を付けたデータベース テーブルを使用します。コンテナは、このテーブルを使用してキーを生成します。

このオプションでは、現在の主キー値を保持するテーブルに名前を付けます。テーブルは、次の文で定義するように、1 行 1 カラムで構成されます。

```
CREATE table_name (SEQUENCE int)
INSERT into table_name VALUES (0)
```

注意： Oracle のテーブルの作成手順については、Oracle データベースのマニュアルを参照してください。

`weblogic-cmp-rdbms-jar.xml` ファイルで `key_cache_size` 要素を設定して、データベースの SELECT および UPDATE によって一度に取得する主キー値の数を指定します。`key_cache_size` のデフォルト値は `1` です。データベース アクセスを最小限に抑えてパフォーマンスを向上するために、BEA ではこの要素には値 `>1` を設定することを推奨しています。この機能の詳細については、5-39 ページの「主キーの命名済シーケンス テーブル サポートの指定」を参照してください。

現時点では、WebLogic Server は、Oracle および Microsoft SQL Server 向けの DBMS 主キー生成サポートだけを提供します。ただし、命名済シーケンス テーブルは、サポートされている他のデータベースで使用できます。また、この機能は単純 (非複合) 主キーで使用することを想定したものです。

有効なキー フィールド型

Bean の抽象「get」および「set」メソッドでは、以下の 2 つの型のいずれかとしてフィールドを宣言できます。

- `java.lang.Integer`
- `java.lang.Long`

Oracle 用主キー サポートの指定

Oracle データベース用の主キー生成サポートでは、Oracle の `SEQUENCE` 機能が使用されます。この機能は、Oracle データベース内の `Sequence` エンティティと連携して一意の主キーを生成します。Oracle `SEQUENCE` は、新しい数値が必要な場合に呼び出されます。

SEQUENCE がデータベース内に作成されたら、XML デプロイメント記述子で自動キー生成を指定します。weblogic-cmp-rdbms-jar.xml ファイルで、次のように自動キー生成を指定します。

図 5-7 Oracle 用自動キー生成の指定

```
<automatic-key-generation>
       <generator-type>ORACLE</generator-type>
       <generator_name>test_sequence</generator-name>
       <key-cache-size>10</key-cache-size>
</automatic-key-generation>
```

generator-name 要素で、使用する ORACLE SEQUENCE の名前を指定します。ORACLE SEQUENCE が SEQUENCE INCREMENT 値を付けて作成した場合は、key-cache-size を指定しなければなりません。この値は、Oracle SEQUENCE INCREMENT 値と一致する必要があります。これら 2 つの値が一致しない場合、重複キーの問題が発生する可能性が高くなります。

警告： Oracle では、ジェネレータ タイプ USER_DESIGNATED_TABLE を使用しないでください。使用すると、次の例外が発生する可能性があります。

```
javax.ejb.EJBException: nested exception
is:java.sql.SQLException: Automatic Key Generation Error:
attempted to UPDATE or QUERY NAMED SEQUENCE TABLE
NAMED_SEQUENCE_TABLE, but encountered SQLException
java.sql.SQLException:ORA-08177: can't serialize access for
this transaction.
```

USER_DESIGNATED_TABLE モードは、TX ISOLATION LEVEL を SERIALIZABLE に設定します。Oracle では、これによって問題が発生する可能性があります。

代わりに、AutoKey オプション ORACLE を使用します。

Microsoft SQL Server 用主キー サポートの指定

Microsoft SQL Server データベース用の主キー生成サポートでは、SQL Server の IDENTITY カラムが使用されます。Bean が作成され、新しい行がデータベース テーブルに挿入されると、SQL Server は、IDENTITY カラムとして指定されたカラムに、次の主キー値を自動的に挿入します。

注意： Microsoft SQL Server のテーブルの作成手順については、Microsoft SQL Server データベースのマニュアルを参照してください。

IDENTITY がデータベース テーブル内に作成されたら、XML デプロイメント記述子で自動キー生成を指定します。weblogic-cmp-rdbms-jar.xml ファイルで、次のように自動キー生成を指定します。

図 5-8 Microsoft SQL 用自動キー生成の指定

```
<automatic-key-generation>
       <generator-type>SQL_SERVER</generator-type>
</automatic-key-generation>
```

generator-type 要素では、主キーの生成方法を指定します。

主キーの命名済シーケンス テーブル サポートの指定

サポートされていないデータベース向けの主キー生成サポートでは、Named SEQUENCE TABLE を使用してキー値を保持します。テーブルには、整数の SEQUENCE INT である単一カラムを持つ単一行を含める必要があります。このカラムは、現在のシーケンス値を保持します。

注意： テーブルの作成手順については、各データベース製品のマニュアルを参照してください。

命名済シーケンス テーブル サポートを利用する場合、その基盤データベースがトランザクション アイソレーション レベル、TRANSACTION_SERIALIZABLE をサポートしているか確かめます。weblogic-ejb.xml ファイルの isolation-level 要素にこのオプションを指定します。TRANSACTION_SERIALIZABLE オプションは、あるトランザクションを複数回同時に実行することが、そのトランザクションを順番に複数回実行した場合と同じ結果になることを仕様とします。データベースで トランザクション アイソレーション レベル、TRANSACTION_SERIALIZABLE がサポートされていなければ、命名済シーケンス テーブル サポートを使用できません。

注意： 基盤データベースでどのような型のアイソレーション レベルがサポートされているかについては、そのデータベースのドキュメントを参照してください。アイソレーション レベルの設定の詳細については、6-6 ページの「EJB デプロイメント記述子の指定と編集」を参照してください。

NAMED_SEQUENCE_TABLE がデータベース内に作成されたら、次の例のように、weblogic-cmp-rdbms-jar.xml ファイル内の XML デプロイメント記述子を使用して自動キー生成を指定します。

図 5-9 命名済シーケンス テーブル用の自動キー生成サポートの指定

```
<automatic-key-generation>
       <generator-type>NAMED_SEQUENCE_TABLE</generator-type>
       <generator_name>MY_SEQUENCE_TABLE_NAME</generator-name>
       <key-cache-size>100</key-cache-size>
</automatic-key-generation>
```

generator-name 要素によって、使用する SEQUENCE TABLE の名前を指定します。key-cache-size を使用すると、1 回の DBMS 呼び出しでコンテナが取得するキーの数を示すキー キャッシュのサイズをオプションで指定することもできます。

パフォーマンスを向上するために、BEA ではこの値を >1 (1 より大きい) に設定することを推奨しています。この設定により、次のキー値を取得するためのデータベースの呼び出し回数を減らすことができます。

また、NAMED SEQUENCE テーブルは Bean のタイプごとに作成することをお勧めします。異なるタイプの Bean が NAMED SEQUENCE テーブルを共有しないようにしてください。こうすることで、キー テーブルの競合の発生を防ぎます。

EJB 2.0 CMP の複数のテーブル マッピング

EJB 2.0 CMP Bean における複数のテーブル マッピングを使用すると、1 つの EJB をあるデータベース内の複数の DBMS テーブルにマップできるようになります。その EJB の weblogic-cmp-rdbms-xml ファイルで、複数の DBMS テーブルとカラムを、EJB とこれのフィールドにマップすることによって、この機能をコンフィグレーションします。マッピングには次のタイプが含まれます。

- EJB コンテナ管理による永続性 (cmp) フィールド ー これらのフィールドに、EJB のどの cmp-field をどの DBMS テーブルにマップするかを記述します。
- EJB コンテナ管理による関係 (cmr) フィールド ー これらのフィールドに、DBMS 内の関係をマップするのに必要な外部キー カラムがどこの EJB DBMS テーブルにあるかを記述します。

複数のテーブル マッピングを有効にするには、次のことが必要になります。

- その EJB があるコンテナ管理による関係に関与しており、その関係が DBMS テーブルに複数の外部キーを持つ必要がある場合、これらの外部キーはその EJB の複数のテーブルの中の 1 つのテーブルのみに置かれます。

これまでは、1 つの EJB を 1 つのテーブル、あるいは、フィールドやカラムの 1 リストに関連付けていました。現在は、EJB がマップするテーブルの数の分だけ、フィールドやカラムの一連のセットを 1 つの EJB にマップさせることができます。

1 つの Bean での複数のテーブル マッピングには次のような制限があります。

ある 1 つのエンティティ Bean にマップする複数のテーブルの主キー間に、参照一貫性制約を課してはなりません。この場合、Bean 削除時に実行時エラーになる可能性があります。

関係の中の Bean の 1 つが複数のテーブルにマップする場合の CMR の指定については、5-51 ページの「複数のテーブルにマップされる EJB の CMR の指定」を参照してください。

自動テーブル作成

テーブルがまだ作成されていない場合には、XML デプロイメント記述子ファイルおよび Bean クラスの記述に基づいて、WebLogic Server がテーブルを自動的に作成するよう指定できます。JAR ファイル内の関係に結合が含まれている場合、テーブルはすべての Bean および関係の結合テーブルに対して作成されます。この機能を明示的に有効にするには、JAR ファイルのすべての Bean に対して、RDBMS のデプロイメントごとのデプロイメント記述子でこの機能を定義します。

自動テーブル生成を有効にした場合、WebLogic Server は `weblogic-cmp-rdbms-jar.xml` 内の `database-type` 要素の値を調べ、そのデータベースのテーブルを作成するのに必要になる正確な構文とデータ型変換について判断します。WebLogic Server バージョン 7.0 では、以下のデータベースおよびベンダにおけるベンダ固有の `CREATE TABLE` 構文とデータ型変換を使用します。

- Informix
- Oracle
- PointBase
- SQL Server
- Sybase

これら以外のすべてのデータベース システムの場合、WebLogic Server では基本的構文と次の表に示すデータ型変換を使用し、最適な方法で新しいテーブルを作成します。

表 5-1 Java の汎用的フィールドと DBMS カラムの型の変換

Java の型	DBMS カラムの型
boolean	INTEGER
byte	INTEGER
char	CHAR
double	DOUBLE PRECISION
float	FLOAT
int	INTEGER
long	INTEGER
short	INTEGER
java.lang.string	VARCHAR (150)
java.lang.BigDecimal	DECIMAL (38, 19)

Java の型	DBMS カラムの型
java.lang.Boolean	INTEGER
java.lang.Byte	INTEGER
java.lang.Character	CHAR (1)
java.lang.Double	DOUBLE PRECISION
java.lang.Float	FLOAT
java.lang.Integer	INTEGER
java.lang.Long	INTEGER
java.lang.Short	INTEGER
java.sql.Date	DATE
java.sql.Time	DATE
java.sql.Timestamp	DATETIME
byte[]	RAW (1000)
有効な SQL 型でないすべてのシリアライズ可能なクラス	RAW (1000)

デプロイメント ファイルの記述に基づいてフィールドをデータベース内の適切なカラムの型にマップできない場合、CREATE TABLE は失敗し、エラーが送出されるので、テーブルを手動で作成する必要があります。

プロダクション環境では自動テーブル作成を使用しないことをお勧めします。この機能は、設計および試作品の開発段階での使用が適しています。プロダクション環境では、外部キーの制約の宣言など、より正確なテーブル スキーマ定義を使用する必要があります。

自動テーブル作成を定義するには、次の手順に従います。

1. weblogic-cmp-rdbms-jar.xml ファイルで create-default-dbms-table 要素を True に設定して、JAR ファイルのすべての Bean に対して自動テーブル作成を明示的に有効にします。次の構文で指定します。

```
<create-default-dbms-tables>True</create-default-dbms-tables>
```

2. weblogic-cmp-rdbms-jar.xml の database-type 要素でデータベース システムまたはデータベース ベンダ名を正しく指定します。database-type の値が INFORMIX、ORACLE、POINTBASE、SQL_SERVER、SYBASE であるものについては、CREATE TABLE 文とデータ型マッピングが提供されています。他のすべての DBMS システムでは、基本構文と上記のテーブルに示したデータ型変換を用います。

コンテナ管理による関係

コンテナ管理による関係 (CMR) は 2 つのエンティティ EJB 間で定義する関係であり、データベースのテーブル間の関係に似ています。同じ処理タスクに関与する 2 つの EJB 間で CMR を定義すると、アプリケーションは以下の機能を利用することができます。

- 関連する複数の Bean を一緒にキャッシュして、処理タスクを完遂するために必要なクエリ数を減らすことができる
- 一括りにされたデータベース処理をトランザクションの最後に正しく順序付けし、データベース一貫性の問題が起こらないようにすることができる
- 関連する Bean をカスケード削除機能で自動的に削除することができる

CMR について

この節では、WebLogic Server の CMR の特長と制限について説明します。CMR のコンフィグレーション手順については、5-46 ページの「コンテナ管理による関係の定義」を参照してください。

要件と制限

同じ .jar にパッケージ化され、データが同じデータベースに格納される 2 つの WebLogic Server エンティティ Bean 間で関係を定義できます。同じ関係に参加するエンティティは、同じデータソースにマップされる必要があります。

WebLogic Server は、異なるデータソースにマップされたエンティティ Bean 間の関係をサポートしていません。コンテナ管理による関係に参加する各 Bean の抽象スキーマは、同じ ejb-jar.xml ファイルで定義する必要があります。

注意： EJB 2.1 仕様では、エンティティ Bean にローカル インタフェースがない場合、そのエンティティ Bean が参加できる唯一の CMR は一方向 (当該エンティティ Bean から別のエンティティ Bean) の CMR であるとされています。

ただし、WebLogic Server ではリモート インタフェースのみのエンティティ Bean に以下のことが許可されます。

- 双方向の CMR に参加する、または
- 別のエンティティとの一方向 CMR の対象となる

この機能は EJB 2.1 では規定されていないので、リモート インタフェースのみを持ち、双方向の関係に参加しているか一方向の関係の対象であるエンティティ Bean は、他のアプリケーション サーバに移植できない場合もあります。

関係のカーディナリティ

エンティティ Bean は、別のエンティティ Bean と 1 対 1、1 対多、または多対多の関係を持つことができます。

関係の方向

CMR は、1 対 1 、1 対多、または多対多のどれであるかに関係なく、一方向と双方向のどちらにもなります。CMR の方向は、関係の片側の Bean がもう一方の側の Bean からアクセス可能かどうかを決定します。

一方向の CMR は一方通行です。「従属」側の Bean は、関係のもう一方の Bean を意識しません。カスケード削除などの CMR 関連の機能は、従属 Bean にのみ適用できます。たとえば、EJB1 から EJB2 の一方向の CMR でカスケード削除がコンフィグレーションされている場合、EJB1 を削除すると EJB2 も削除されますが、EJB2 を削除しても EJB1 は削除されません。

注意： カスケード削除機能にとって、関係のカーディナリティは重要な要素です。たとえ関係が双方向であっても、カスケード削除は関係の「多」側からはサポートされません。

双方向の関係は両側通行です。関係の各 Bean が他方の Bean を意識します。CMR 関連の機能は、両方向でサポートされます。たとえば、EJB1 と EJB2 の双方向の CMR でカスケード削除がコンフィグレーションされている場合、CMR のいずれかの Bean を削除すると他方の Bean も削除されます。

関係の削除

関係に参加している Bean インスタンスが削除されると、コンテナは自動的にその関係を削除します。たとえば、従業員と部署の関係がある場合には、従業員を削除すると、コンテナは従業員と部署の関係も削除します。

コンテナ管理による関係の定義

CMR の定義では、関係とそのカーディナリティおよび方向を ejb-jar.xml で指定します。weblogic-cmp-jar.xml では、関係のデータベース マッピングの細目を定義し、リレーションシップ キャッシングを有効にします。それらの手順は、以下の節で説明します。

- 5-46 ページの「ejb-jar.xml での関係の指定」
- 5-49 ページの「weblogic-cmp-jar.xml での関係の指定」

ejb-jar.xml での関係の指定

コンテナ管理による関係は、ejb-jar.xml の ejb-relation スタンザで定義します。図 5-10 は、2 つのエンティティ EJB (TeacherEJB と StudentEJB) の関係の ejb-relation スタンザを示しています。

ejb-relation スタンザでは、関係のそれぞれの側の ejb-relationship-role を指定します。ロール スタンザでは、関係に対する各 Bean の認識を指定します。

図 5-10 ejb-jar.xml における 1 対多で双方向の CMR

```
<ejb-relation>
   <ejb-relation-name>TeacherEJB-StudentEJB</ejb-relation-name>
      <ejb-relationship-role>
         <ejb-relationship-role-name>teacher-has-student
         </ejb-relationship-role-name>
         <multiplicity>One</multiplicity>
```

```
            <relationship-role-source>
               <ejb-name>TeacherEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>teacher</cmr-field-name>
            </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
         <ejb-relationship-role-name>student-has-teacher
         </ejb-relationship-role-name>
         <multiplicity>Many</multiplicity>
         <relationship-role-source>
            <ejb-name>StudentEJB</ejb-name>
         </relationship-role-source>
         <cmr-field>
            <cmr-field-name>student</cmr-field-name>
            <cmr-field-type>java.util.Collection
         <cmr-field>
      </ejb-relationship-role>
```

関係のカーディナリティの指定

関係のそれぞれの側のカーディナリティは、`ejb-relationship-role` スタンザの `<multiplicity>` 要素で指定します。

図 5-10 では、関係 `TeacherEJB-StudentEJB` のカーディナリティは 1 対多です。そのカーディナリティは、`TeacherEJB` 側で `multiplicity` を `one` に設定し、`StudentEJB` 側では `Many` に設定することで指定されています。

図 5-11 の CMR のカーディナリティは 1 対 1 です。この関係では、両方のロールスタンザで `multiplicity` が `one` に設定されています。

図 5-11 ejb-jar.xml における 1 対 1 で一方向の CMR

```
<ejb-relation>
   <ejb-relation-name>MentorEJB-StudentEJB</ejb-relation-name>
      <ejb-relationship-role>
         <ejb-relationship-role-name>mentor-has-student
         </ejb-relationship-role-name>
         <multiplicity>One</multiplicity>
         <relationship-role-source>
            <ejb-name>MentorEJB</ejb-name>
         </relationship-role-source>
         <cmr-field>
             <cmr-field-name>mentorID</cmr-field-name>
```

```
      </cmr-field>
   </ejb-relationship-role>
   <ejb-relationship-role>
      <ejb-relationship-role-name>student-has-mentor
      </ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
         <ejb-name>StudentEJB</ejb-name>
      </relationship-role-source>
   </ejb-relationship-role>
```

関係の片側で <multiplicity> が Many に設定されていて、その <cmr-field> がコレクションである場合は、<cmr-field-type> を java.util.Collection として指定する必要があります (図 5-10 の関係の StudentEJB 側を参照)。cmr-field が単一値のオブジェクトである場合は、cmr-field-type を指定する必要はありません。

Table 5-2 は、関係の各 Bean の cmr-field の内容を、その関係のカーディナリティに基づいて示しています。

表 5-2 カーディナリティと cmr-field-type

EJB1 と EJB2 の関係のカーディナリティ	EJB1 の cmr-field の内容	EJB2 の cmr-field の内容
1 対 1	単一値のオブジェクト	単一値のオブジェクト
1 対多	単一値のオブジェクト	コレクション
多対多	コレクション	コレクション

関係の方向の指定

CMR の方向は、関係のそれぞれの側の ejb-relationship-role スタンザで cmr-field を設定する、またはしないことでコンフィグレーションします。

双方向の CMR では、関係の両側の ejb-relationship-role スタンザに cmr-field 要素があります (図 5-10 を参照)。

一方向の関係の場合は、関係の片方のロール スタンザにのみ cmr-field があります。起点側の EJB の ejb-relationship-role には cmr-field がありますが、対象側 Bean のロール スタンザにはありません。図 5-11 では、MentorEJB から StudentEJB の一方向の関係が指定されています。StudentEJB の ejb-relationship-role スタンザには、cmr-field 要素がありません。

weblogic-cmp-jar.xml での関係の指定

ejb-jar.xml で定義された各 CMR は、weblogic-cmp-jar.xml の weblogic-rdbms-relation スタンザでも定義する必要があります。weblogic-rdbms-relation は関係を識別し、関係の片側または両側について relationship-role-map スタンザを格納します。このスタンザは、関係に参加している Bean 間のデータベースレベルの関係をマップします。

weblogic-rdbms-relation の relation-name は、ejb-jar.xml の CMR の ejb-relation-name と同じでなければなりません。

1 対 1 の関係と 1 対多の関係

1 対 1 と 1 対多の関係の場合、relationship-role-map は関係の片側のみで定義します。

1 対 1 の関係では、一方の Bean の外部キーがもう一方の Bean の主キーにマップされます。

図 5-12 は、MentorEJB と StudentEJB の 1 対 1 の関係の weblogic-rdbms-relation スタンザです。その <ejb-relation> は、図 5-11 で示されています。

図 5-12 1 対 1 の CMR の weblogic-cmp-jar.xml

```
<weblogic-rdbms-relation>
   <relation-name>MentorEJB-StudentEJB</relation-name>
   <weblogic-relationship-role>
       <relationship-role-name>
       mentor-has-student
       </relationship-role-name>
        <relationship-role-map>
            <column-map>
               <foreign-key-column>student</foreign-key-column>
               <key-column>StudentID/key-column>
```

```
            </column-map>
          <relationship-role-map>
     </weblogic-relationship-role>
```

1 対多の関係では常に、ある Bean の外部キーが別の Bean の主キーにマップされます。1 対多の関係では、外部キーが常に、関係の「多」サイドの Bean に関連付けられます。

図 5-13 は、`TeacherEJB` と `StudentEJB` の 1 対多の関係の `weblogic-rdbms-relation` スタンザです。その `<ejb-relation>` は、図 5-10 で示されています。

図 5-13 1 対多の CMR の weblogic-rdbms-relation

```
<weblogic-rdbms-relation>
   <relation-name>TeacherEJB-StudentEJB</relation-name>
   <weblogic-relationship-role>
       <relationship-role-name>
       teacher-has-student
       </relationship-role-name>
        <relationship-role-map>
            <column-map>
               <foreign-key-column>student</foreign-key-column>
               <key-column>StudentID/key-column>
            </column-map>
        <relationship-role-map>
    </weblogic-relationship-role>
```

多対多の関係

多対多の関係では、関係のそれぞれの側で `weblogic-relationship-role` スタンザを指定します。そのマッピングには、結合テーブルが関わります。結合テーブルの各行には、関係に関与するエンティティの主キーに対応する 2 つの外部キーが格納されます。関係の方向は、関係のデータベース マッピングの指定方法に影響しません。

図 5-14 は、2 人の従業員の関係 `friends` の weblogic-rdbms-relation スタンザを示しています。

`FRIENDS` 結合テーブルには、`first-friend-id` および `second-friend-id` という 2 つのカラムがあります。各カラムには、他の従業員の友人である特定の従業員を示す外部キーが格納されています。従業員テーブルの主キー カラムは `id` です。この例では、従業員 Bean が 1 つのテーブルにマップされています。従業員

Bean が複数のテーブルにマップされている場合、主キー カラムを格納するテーブルを relation-role-map で指定する必要があります。例については、5-51 ページの「複数のテーブルにマップされる EJB の CMR の指定」を参照してください。

図 5-14　多対多の CMR の weblogic-rdbms-relation

```
<weblogic-rdbms-relation>
  <relation-name>friends</relation-name>
  <table-name>FRIENDS</table-name>
  <weblogic-relationship-role>
    <relationship-role-name>first-friend
    </relationship-role-name>
    <relationship-role-map>
      <column-map>
        <foreign-key-column>first-friend-id</foreign-key-column>
        <key-column>id</key-column>
     </column-map
    </relationship-role-map>
  <weblogic-relationship-role>
    <weblogic-relationship-role>
      <relationship-role-name>second-friend</relationship-role-
      name>
    <relationship-role-map>
      <column-map>
        <foreign-key-column>second-friend-id</foreign-key-column>
        <key-column>id</key-column>
      </column-map>
    </relationship-role-map>
   </weblogic-relationship-role>
</weblogic-rdbms-relation>
```

複数のテーブルにマップされる EJB の CMR の指定

関係に関与する CMP Bean は、複数の DBMS テーブルにマップできます。

- 1 対 1 または 1 対多の関係の外部キー側の Bean が複数のテーブルにマップされる場合は、外部キー カラムを格納するテーブルの名前を foreign-key-table 要素で指定する必要があります。
- 逆に、1 対 1 または 1 対多の関係の主キー側の Bean または多対多の関係に参加している Bean が複数のテーブルにマップされる場合は、主キーを格納するテーブルの名前を primary-key-table 要素で指定する必要があります。

関係のどの Bean も複数のテーブルにマップされない場合は、使用されるテーブルが暗黙的に了解されるので、foreign-key-table 要素と primary-key-table 要素は省略することができます。

図 5-15 は、1 対 1 の関係の外部キー側の Bean (Fk_Bean) が 2 つのテーブル (Fk_BeanTable_1 と Fk_BeanTable_2) にマップされる CMR の relationship-role-map です。

関係の外部キー カラム (Fk_column_1 と Fk_column_2) は、Fk_BeanTable_2 に配置されます。主キー側の Bean (Pk_Bean) は、主キー カラム Pk_table_pkColumn_1 と Pk_table_pkColumn_2 のある 1 つのテーブルにマップされます。

外部キー カラムを持つテーブルは、<foreign-key-table> 要素で指定します。

図 5-15 1 対 1 の CMR (1 つの Bean が複数のテーブルにマップされる)

```
<relationship-role-map
   <foreign-key-table>Fk_BeanTable_2</foreign-key-table>
   <column-map>
      <foreign-key-column>Fk_column_1</foreign-key-column>
      <key-column>Pk_table_pkColumn_1</key-column>
   </column-map>
   <column-map>
      <foreign-key-column>Fk_column_2</foreign-key-column>
      <key-column>Pk_table_pkColumn_2</key-column>
   </column-map>
</relationship-role-map>
```

CMR でのリレーションシップ キャッシングの使用

リレーションシップ キャッシングでは、関係する複数 Bean をキャッシュにロードし、それらに結合クエリを発行してクエリの発行回数を削減することによって、エンティティ Bean のパフォーマンスが高められます。

たとえば、次のような関係のエンティティ Bean があるとします。

customerBean	1 対多関係を持つ	accountBean
accountBean	1 対 1 関係を持つ	addressBean
customerBean	1 対 1 関係を持つ	phoneBean

accountBean および addressBean の EJB コード (1 対 1 の関係を持つ) を検討します。

```
Account acct = acctHome.findByPrimaryKey("103243");
Address addr = acct.getAddress();
```

リレーションシップ キャッシングが行われなければ、コードの 1 行目で accountBean をロードする SQL クエリが発行され、2 行目で addressBean をロードする SQL クエリが発行されます。これにより、データベースには 2 つのクエリが生じます。

リレーションシップ キャッシングが行われると、accountBean と addressBean の双方をロードする 1 つのクエリがコードの 1 行目で発行されるため、パフォーマンスは向上するはずです。したがって、特定のファインダ メソッドを実行した後に関連する Bean にアクセスすることがわかっている場合は、リレーションシップ キャッシング機能を通じてファインダ メソッドに通知しておくことをお勧めします。

リレーションシップ キャッシングは、weblogic-cmp-jar.xml で以下のスタンザを使用して指定します。

- relationship-caching ー 関連 Bean の caching-name と caching-element を定義します (図 5-16 を参照)。
- weblogic-query の caching-name 要素 ー 指定すると、ファインダ クエリが実行されたときに、WebLogic Server によって関連 Bean がキャッシュにロードされます。例については、図 5-17 を参照してください。
- weblogic-rdbms-jar (weblogic-cmp-jar.xml のルート) の database-type ー リレーションシップ キャッシングは、外部結合のクエリを使用します (構文はデータベースによって異なる)。

注意： weblogic-ejb-jar.xml ファイル上で 、ある関係が指定されている Bean (たとえば、上記の XML コード例では customerBean) の finders-load-bean 要素が False に設定されていないことを確かめます。このようにしなければ、リレーションシップ キャッシングが有効になりません。finder-load-bean 要素のデフォルトは True です。

図 5-16 weblogic-cmp-jar.xml の relationship-caching

```
<relationship-caching>
  <caching-name>cacheMoreBeans</caching-name>
  <caching-element>
    <cmr-field>accounts</cmr-field>
  </caching-element>
</relationship-caching>
```

図 5-17 weblogic-cmp-jar.xml の weblogic-query

```
<weblogic-query>
  <query-method>
    <method-name>findBigAccounts</method-name>
      <method-params>
        <method-param>java.lang.String</method-param>
        <method-param>java.lang.Integer</method-param>
     </method-params>
  </query-method>
  <caching-name>cacheMoreBeans</caching-name>
</weblogic-query>
```

caching-element のネスト

ネストされた caching-element を使用すると、複数レベルの関連 Bean をロードすることができます。現在は、指定できる caching-element の数に制限はありません。しかし、caching-element のレベルをあまり多く設定しすぎると、そのトランザクションのパフォーマンスに影響すると考えられます。

リレーションシップ キャッシングでは結合クエリを使用し、結合クエリではおそらく結果を ResultSet 内のテーブルに複写するため、指定した caching-element 要素の数が ResultSet に複写される結果の数に直に影響してきます。1 対多関係では、relationship-caching 要素であまり多くの caching-element デプロイメント記述子を指定しないでください。caching-element デプロイメント記述子の数に対して複写される結果の数が乗算される可能性があります。

リレーションシップ キャッシングの制限

リレーションシップ キャッシングの機能には次のような制限があります。

1. リレーションシップ キャッシングは 1 対 1 または 1 対多関係に対してのみ働きます。
2. weblogic-ql を使用した場合、この機能は、EJBObject Bean または EJBLocalObject Bean への参照を返すファインダ メソッドを使ったときのみ機能します。

3. ファインダ メソッドまたは選択メソッドでリレーションシップ キャッシングを有効にする場合、クエリの結果は、たとえ distinct キーワードを指定していなくても常に異なる集合になります。ResultSet に複写されているものが元のデータの結果なのか、外部結合の結果なのか見分けることができないためです。

カスケード削除

カスケード削除メカニズムは、エンティティ Bean オブジェクトを削除する場合に使用します。カスケード削除を特定の関係に対して指定した場合、エンティティ オブジェクトの有効期間は他方のエンティティ オブジェクトに依存します。1 対 1 関係と 1 対多関係に対してはカスケード削除を指定できますが、多対多関係に対しては指定できません。cascade delete() メソッドは WebLogic Server の削除機能を使用し、database cascade delete() メソッドでは、基盤データベースに組み込まれているカスケード削除のサポートを使用するよう WebLogic Server に指示します。

この機能を有効にするには、Bean コードを再コンパイルしてデプロイメント記述子の変更を有効にする必要があります。

カスケード削除を有効にするには、以下の 2 つの方法のいずれかに従います。

カスケード削除メソッド

cascade delete() メソッドでは、WebLogic Server を使用してオブジェクトを削除します。削除したエンティティに関連するエンティティ Bean に対して cascade delete 要素が指定されている場合、カスケード削除が行われ、関連するエンティティ Bean もすべて削除されます。

カスケード削除を指定するには、ejb-jar.xml デプロイメント記述子要素の cascade-delete 要素を使用します。これはデフォルト メソッドです。データベースの設定には変更を加えません。WebLogic Server は、カスケード削除が行われる場合に削除対象のエンティティ オブジェクトをキャッシュします。

カスケード削除を指定するには、ejb-jar.xml ファイルの cascade-delete 要素を使用します。

図 5-18 カスケード削除の指定

```
<ejb-relation>
       <ejb-relation-name>Customer-Account</ejb-relation-name>
       <ejb-relationship-role>
              <ejb-relationship-role-name>Account-Has-Customer
              </ejb-relationship-role-name>
              <multiplicity>one</multiplicity>
              <cascade-delete/>
       </ejb-relationship-role>
</ejb-relation>
```

注意： この cascade delete() メソッドは、ejb-relation 要素に含まれる一方の ejb-relationship-role 要素に対してのみ指定できます。この場合、同じ ejb-relation 要素の他方の ejb-relationship-role 要素に値が one の multiplicity 属性が指定されている必要があります。

データベース カスケード削除メソッド

database cascade delete() メソッドを使用すると、アプリケーションはデータベースに組み込まれているカスケード削除のサポートを利用できるので、パフォーマンスの向上を見込めます。db-cascade-delete 要素を weblogic-cmp-rdbms-jar.xml ファイルにまだ指定していない場合、データベースのカスケード削除機能を有効にしないでください。有効にすると、データベースで不正な結果が生成されます。

weblogic-cmp-rdbms-jar.xml ファイルの db-cascade-delete 要素では、基盤となる DBMS の組み込みカスケード削除機能をカスケード削除処理で使用するよう指定します。この機能はデフォルトでは無効になっているので、EJB コンテナは Bean ごとに SQL DELETE 文を発行してカスケード削除に関連する Bean を削除します。

db-cascade-delete 要素を weblogic-cmp-rdbms-jar.xml に指定する場合、cascade-delete 要素を ejb-jar.xml に指定する必要があります。

db-cascade-delete を有効にすると、データベース テーブルの追加設定が必要になります。たとえば、dept がデータベースで削除された場合、Oracle データベース テーブルの次の設定によって、すべての従業員がカスケード削除されます。

図 5-19 カスケード削除用の Oracle テーブルの設定

```
  CREATE TABLE dept
```

```
  (deptno   NUMBER(2) CONSTRAINT pk_dept PRIMARY KEY,
   dname    VARCHAR2(9) );
CREATE TABLE emp
  (empno    NUMBER(4) PRIMARY KEY,
   ename    VARCHAR2(10),
   deptno   NUMBER(2)   CONSTRAINT fk_deptno
            REFERENCES dept(deptno)
            ON DELETE CASCADE );
```

CMR とローカル インタフェース

WebLogic Server は、セッション Bean およびエンティティ Bean 用のローカル インタフェースをサポートしています。ローカル インタフェースを使用すると、エンタープライズ JavaBean は、同じ EJB コンテナ内で別のセマンティクスと実行コンテキストを使用して動作できます。通常、EJB は同じ EJB 内にあり、同じ Java 仮想マシン (JVM) 内で動作します。このようにして、EJB は通信にネットワークを使用せず、Java Remote Method Invocation-Internet Inter-ORB Protocol (RMI-IIOP) 接続によるオーバーヘッドの発生を防ぎます。

EJB とコンテナ管理による永続性の関係は、EJB のローカル インタフェースに基づいています。関係に関わる EJB には、ローカル インタフェースが必要です。ローカル インタフェース オブジェクトは、軽量の永続的オブジェクトです。これらのオブジェクトを使用すると、リモート オブジェクトを使用するよりも完成度の高いコーディングが可能になります。ローカル インタフェースも参照渡しです。ゲッターはローカル インタフェースに含まれています。

以前のバージョンの WebLogic Server では、リモート インタフェースに基づいて関係を指定できます。しかし、新しいコードでは、リモート インタフェースを使用する CMP の関係を使用しないことをお勧めします。

EJB コンテナを使用すると、ローカル クライアントが JNDI 経由でローカル ホーム インタフェースにアクセスできるようになります。ローカル インタフェースを参照するには、ローカルの JNDI 名が必要です。エンティティ Bean のローカル ホスト インタフェースを実装するオブジェクトは、`EJBLocalHome` オブジェクトと呼ばれます。weblogic-ejb-jar.xml ファイルの `jndi-name`、また

は local-jndi-name を指定できます。デプロイメント記述子を指定する手順の詳細については、6-6 ページの「EJB デプロイメント記述子の指定と編集」を参照してください。

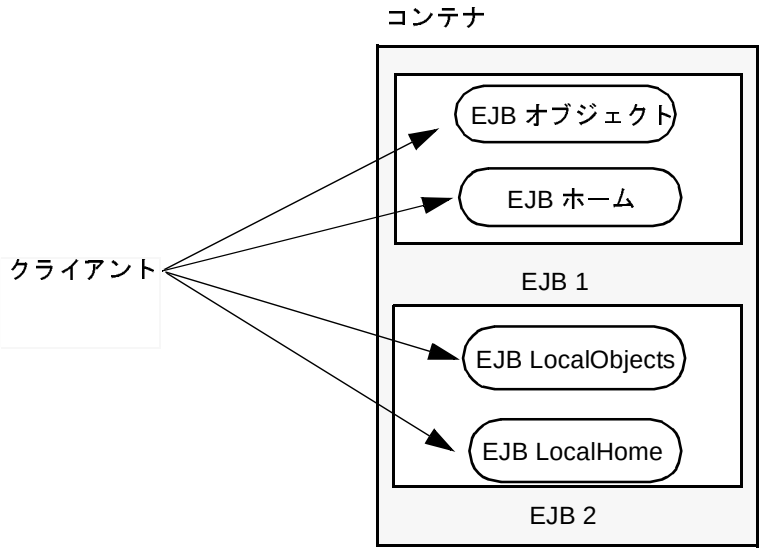
WebLogic Server の以前のバージョンでは、リモート インタフェースを返すために ejbSelect メソッドが使用されていました。現在では、クエリの結果をローカルまたはリモートのどちらのオブジェクトにマップするかを示す ejb-jar.xml ファイルの result-type-mapping 要素を指定します。

ローカル クライアントの使用

セッション Bean またはエンティティ Bean のローカル クライアントは、セッション Bean、エンティティ Bean、メッセージ駆動型 Bean など別の EJB です。ローカル クライアントは、同じ EAR ファイルに含まれており、かつその EAR ファイルがリモートでない限り、サーブレットでもかまいません。ローカル Bean のクライアントは、EAR またはスタンドアロン JAR の一部でなければなりません。

ローカル クライアントは、Bean のローカル インタフェースとローカル ホーム インタフェースを介してセッション Bean またはエンティティ Bean にアクセスします。コンテナは、Bean のローカル インタフェースとローカル ホーム インタフェースを実装するクラスを提供します。これらのインタフェースを実装するオブジェクトは、ローカル Java オブジェクトです。次の図は、ローカル クライアントとローカル インタフェースを持つコンテナを示しています。

図 5-20　ローカル クライアントとローカル インタフェース



WebLogic Server は、EJB 間でローカルの関係と一方向のリモート関係の両方をサポートしています。EJB が同じサーバ上にあり、同じ JAR ファイルを構成している場合、EJB はローカルの関係を持ちます。EJB が同じサーバ上にない場合、関係はリモートでなければなりません。ローカル Bean の関係では、その関係を実装するキーが複合キーである場合は複数カラムのマッピングを指定します。リモート Bean の場合は、リモート Bean の主キーが不明であるため、単一の column-map だけが指定されます。ロールが group-name だけを指定している場合、column-map は指定されません。関係がリモートの場合は group-name は指定されません。

ローカル インタフェースに関するコンテナの変更

ローカル インタフェースを格納するために、コンテナの構造が変更され、以下のものが追加されています。

- EJB ローカル ホーム
- 例外を処理して正しい例外をクライアントに伝播する新しいモデル

グループ

コンテナ管理による永続性では、グループを使用して、エンティティ Bean の特定の永続的な属性を指定します。field-group は、Bean の cmp-field と CMR-field のサブセットを表します。Bean 内の関連フィールドを、障害のあったグループにまとめて 1 つのユニットとしてメモリ内に入れることができます。グループをクエリまたは関係に関連付けることができます。それによって、クエリを実行するか、または関係に従った結果として Bean がロードされたときに、グループ内の指定フィールドのみがロードされます。

指定したグループを持たないクエリと関係に対して、「default」という特殊なグループを使用します。デフォルトでは、default グループには、Bean のすべての CMP-field と、Bean の永続的な状態に外部キーを追加するすべての CMR-field が格納されます。

フィールドは複数のグループに関連付けられている場合があります。この場合、フィールドに対して `getXXX()` メソッドを実行すると、そのフィールドを含む最初のグループで障害が発生します。

フィールド グループの指定

フィールド グループは、`weblogic-rdbms-cmp-jar.xml` ファイルで次のように指定します。

```
<weblogic-rdbms-bean>
       <ejb-name>XXXBean</ejb-name>
       <field-group>
              <group-name>medical-data</group-name>
              <cmp-field>insurance</cmp-field>
              <cmr-field>doctors</cmr-fields>
       </field-group>
</weblogic-rdbms-bean>
```

フィールド グループは、フィールドのサブセットにアクセスする必要があるときに使用します。

EJB リンクの使用

WebLogic Server では、EJB 2.0 仕様で定義されている EJB リンクを完全にサポートしています。アプリケーション コンポーネント内で EJB 参照を宣言し、これを同じ J2EE アプリケーション内で宣言されたエンタープライズ Bean にリンクさせることができます。

`ejb-link` を作成するには、次の手順に従います。

1. 参照元のアプリケーション コンポーネントにある `ejb-ref` 要素の任意指定のデプロイメント記述子要素、`ejb-link` を使って EJB へのリンクを指定します。

   `ejb-link` の値は必ず、参照先となる EJB の `ejb-name` にします。参照先の EJB は、参照元アプリケーション コンポーネントと同じ J2EE アプリケーションの EJB JAR ファイルのどれかに含まれていると考えられます。

   ejb-name は EJB JAR ファイル間で必ずしもユニークでなくてもよいため、そのリンクの絶対パスを与える必要があるかもしれません。

2. 次の構文を使って、同じ J2EE アプリケーション内の EJB へのパス名を与えます。

   ```
   <ejb-link>../products/product.jar#ProductEJB</ejb-link>
   ```

   この参照では参照先の EJB が含まれる EJB JAR ファイルのパス名を与え、「#」でパス名と区別して参照先 Bean の `ejb-name` を追加します。このパス名は、参照元のアプリケーション コンポーネント JAR ファイルからの相対パスです。

デプロイメント記述子を指定する手順については、6-6 ページの「EJB デプロイメント記述子の指定と編集」を参照してください。

CMP フィールドの Java データ型

次の表は、WebLogic Server で使用される CMP フィールドの Java データ型と、それに対応する標準 SQL データ型の Oracle 拡張を示しています。

表 5-3 CMP フィールドの Java データ型

CMP フィールドの Java の型	Oracle のデータ型
boolean	**SMALLINT**
byte	**SMALLINT**
char	**SMALLINT**
double	**NUMBER**
float	**NUMBER**
int	**INTEGER**
long	**NUMBER**
short	**SMALLINT**
java.lang.String	**VARCHAR/VARCHAR2**
java.lang.Boolean	**SMALLINT**
java.lang.Byte	**SMALLINT**
java.lang.Character	**SMALLINT**
java.lang.Double	**NUMBER**
java.lang.Float	**NUMBER**
java.lang.Integer	**INTEGER**
java.lang.Long	**NUMBER**
java.lang.Short	**SMALLINT**

CMP フィールドの Java の型	Oracle のデータ型
java.sql.Date	**DATE**
java.sql.Time	**DATE**
java.sql.Timestamp	**DATE**
java.math.BigDecimal	**NUMBER**
byte[]	**RAW、LONG RAW**
serializable	**RAW、LONG RAW**

SQL CHAR データ型は、CMP フィールドにマップされるデータベース カラムには使用しないでください。このことは、主キーの一部であるフィールドで特に重要です。なぜなら、JDBC ドライバによって返されるパディングの空白は、等しいかどうかの比較を不適切に失敗させるからです。SQL CHAR の代わりに、SQL VARCHAR データ型を使用してください。

`byte[]` 型の CMP フィールドは、`equals()` メソッドと `hashCode()` メソッドを備えるユーザ定義の主キー クラスでラップされていない限り主キーとしては使用できません。なぜなら、`byte[]` クラスには実質的な equals および hashCode が備わっていないからです。

6 WebLogic Server コンテナ用の EJB のパッケージ化

以下の節では、WebLogic Server コンテナにデプロイするために EJB をパッケージ化する方法について説明します。ソース ファイル、デプロイメント記述子、およびデプロイメント モードを始めとしてデプロイメント パッケージの内容も説明します。

- EJB のパッケージ化に必要な手順
- EJB コンポーネント ソース ファイルの見直し
- WebLogic Server の EJB デプロイメント ファイル
- EJB デプロイメント記述子の指定と編集
- デプロイメント ファイルの作成
- 他の EJB およびリソースへの参照
- デプロイメント ディレクトリへの EJB のパッケージ化
- EJB クラスのコンパイルと EJB コンテナ クラスの生成
- WebLogic Server への EJB クラスのロード
- ejb-client.jar の指定
- マニフェスト クラスパス

EJB のパッケージ化に必要な手順

WebLogic Server にデプロイするために EJB を EJB コンテナにパッケージ化するには、次の手順を実行します。

1. EJB ソース ファイル コンポーネントを見直します。
2. EJB デプロイメント ファイルを作成します。
3. EJB デプロイメント記述子を編集します。
4. デプロイメント モードを設定します。
5. EJB コンテナ クラスを生成します。
6. EJB を JAR または EAR ファイルにパッケージ化します。
7. WebLogic Server へ EJB クラスをロードします。

EJB コンポーネント ソース ファイルの見直し

エンティティ Bean とセッション Bean を実装するには、以下のコンポーネントを使用します。

コンポーネント	説明
Bean クラス	Bean クラスは、Bean のビジネス メソッドとライフ サイクル メソッドを実装する。
リモート インタフェース	リモート インタフェースは、Bean の EJB コンテナに入っていないアプリケーションからアクセス可能な Bean のビジネス メソッドを定義する。
リモート ホーム インタフェース	リモート ホーム インタフェースは、Bean の EJB コンテナに入っていないアプリケーションからアクセス可能な Bean のライフ サイクル メソッドを定義する。
ローカル インタフェース	ローカル インタフェースは、同じ EJB コンテナに入っている他の Bean が使用可能な Bean のビジネス ロジックを定義する。

コンポーネント	説明
ローカル ホーム インタフェース	ローカル ホーム インタフェースは、同じ EJB コンテナに入っている他の Bean が使用可能な Bean のライフ サイクル メソッドを定義する。
主キー	主キーは、データベースのポインタを提供する。エンティティ Bean だけが主キーを必要とする。

WebLogic Server の EJB デプロイメント ファイル

EJB のデプロイメント記述子要素を指定するには、以下の WebLogic Server デプロイメント ファイルを使用します。

- `ejb-jar.xml`
- `weblogic-ejb-jar.xml`
- `weblogic-cmp-rdbms-jar.xml` (省略可能、CMP 専用)

Bean をコンパイルすると、デプロイメント ファイルは EJB デプロイメントの一部となります。XML デプロイメント記述子ファイルには、EJB に対するデプロイメント記述子の最低限の設定を含める必要があります。いったんファイルを作成すると、6-6 ページの「EJB デプロイメント記述子の指定と編集」の手順に従って後で編集できます。デプロイメント記述子ファイルは、使用する各ファイルの文書型定義 (DTD) のバージョンに準拠する必要があります。ファイルの文書型定義 (DTD) には、EJB XML デプロイメント記述子ファイルのすべての要素および下位要素 (属性) の名前を記述します。各ファイルの説明については、以下の節を参照してください。

ejb-jar.xml

ejb-jar.xml ファイルには、Sun Microsystems 固有の EJB DTD が格納されます。このファイルのデプロイメント記述子は、エンタープライズ Bean の構造を記述し、内部依存関係とアプリケーション アセンブリ情報を宣言します。アプリケーション アセンブリ情報とは、ejb-jar ファイルのエンタープライズ Bean をアプリケーション デプロイメント ユニットとしてアセンブルする方法を記述するものです。このファイルの要素の説明については、JavaSoft 仕様を参照してください。

weblogic-ejb-jar.xml

weblogic-ejb-jar.xml ファイルには、EJB の同時実行、キャッシング、クラスタ化、および動作を定義する WebLogic Server 固有の EJB DTD が格納されます。また、使用可能な WebLogic Server リソースを EJB にマップする記述子も格納されます。WebLogic Server リソースには、セキュリティ ロール名、データソース (JDBC プールや JMS 接続ファクトリなど)、およびデプロイ済みの他の EJB があります。このファイルの要素の説明については、第 9 章「weblogic-ejb-jar.xml 文書型定義」を参照してください。

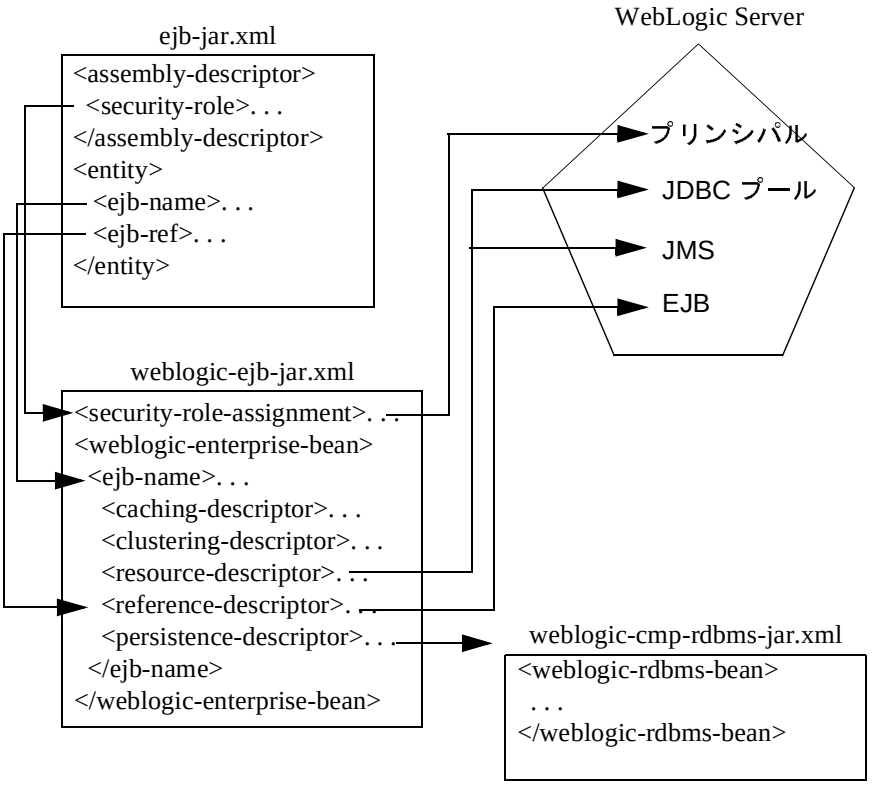
weblogic-cmp-rdbms.xml

weblogic-cmp-rdbms.xml ファイルには、コンテナ管理による永続性サービスを定義する WebLogic Server 固有の EJB DTD が格納されます。このファイルを使用して、コンテナがエンティティ Bean のインスタンス フィールドとデータベースのデータとの同期を処理する方法を指定します。このファイルの要素の説明については、第 10 章「weblogic-cmp-rdbms-jar.xml 文書型定義」を参照してください。

デプロイメント ファイル間の関係

`weblogic-ejb-jar.xml` 内の記述子は、`ejb-jar.xml` 内の EJB 名、動作中の WebLogic Server のリソース名、および `weblogic-cmp-rdbms-jar.xml` (コンテナ管理による永続性を使用するエンティティ EJB の場合) 内に定義されている永続性タイプ データにリンクされています。次の図は、デプロイメント ファイルと WebLogic Server 間の関係を示しています。

図 6-1 デプロイメント ファイルのコンポーネント間の関係



EJB デプロイメント記述子の指定と編集

以下のいずれかの方法で、EJB デプロイメント記述子を指定または編集します。

- テキスト エディタを使用して、Bean のデプロイメント ファイルを手動で編集する。デプロイメント ファイルを手動で編集する方法については、6-7 ページの「EJB デプロイメント記述子の手動編集」を参照してください。
- WebLogic Server Administration Console の EJB デプロイメント記述子エディタを使用して、Bean のデプロイメント ファイルを編集する。デプロイメン

ト記述子エディタの使用方法については、6-8 ページの「EJB デプロイメント記述子エディタの使用」を参照してください。

- WebLogic Server コマンド ライン ユーティリティ ツール、`DDConverter` を使用して、EJB 1.1 デプロイメント記述子を EJB 2.0 XML に変換する。`DDConverter` ツールの使用方法については、8-26 ページの「DDConverter」を参照してください。

デプロイメント ファイルの作成

各ファイルの文書型定義 (DTD) のバージョンに準拠した基本の XML デプロイメント ファイルを EJB 用に作成します。既存の EJB デプロイメント ファイルをテンプレートとして使用することも、WebLogic Server 配布キットの EJB サンプルからコピーすることもできます。

```
SAMPLES_HOME\server\config\examples\applications
```

EJB デプロイメント記述子の手動編集

XML デプロイメント記述子要素を手動で編集するには、次の手順に従います。

1. XML の形式の変更や、ファイルを無効にする可能性のある文字の挿入を行わない ASCII テキスト エディタを使用します。
2. 編集する XML デプロイメント記述子ファイルを開きます。
3. 変更を入力します。使用しているオペレーティング システムで大文字小文字が区別されない場合であっても、ファイル名やディレクトリ名の大文字小文字は正確に指定します。
4. 省略可能な要素に対してデフォルト値を使用する場合は、要素の定義全体を省略するか、または次のように空白値を指定します。

```
<max-beans-in-cache></max-beans-in-cache>
```

EJB デプロイメント記述子エディタの使用

WebLogic Server Administration Console で EJB デプロイメント記述子を編集するには、次の手順に従います。

1. WebLogic Server を起動します。
2. Administration Console を起動して、右ペインの [デプロイメント] ノードを選択し、[EJB] を選択します。
3. 展開されたデプロイ済み EJB のリストから、編集したい Bean を右クリックし、[EJB 記述子の編集] を選択します。
4. EJB デプロイメント記述子エディタが表示されたら、[永続化] ボタンまたは [検証] ボタンをクリックします。
   - その EJB のデプロイメント記述子ファイルでの変更を保存したい場合には、[永続化] を選択します。
   - WebLogic Server で EJB のデプロイメント記述子ファイルの構造を検査し、XML ファイルが正しく解析できるかを確認する場合には、[検証] を選択します。
5. 左ペインで [EJB] をクリックし、ノードを展開します。

   EJB デプロイメント記述子ファイルを表す以下のノードが表示されます。
   - **[ejb-jar] :** この EJB の `ejb-jar.xml` ファイル デプロイメント記述子を表します。
   - **[webLogic-ejb-jar] :** この EJB の `weblogic-ejb-jar.xml` ファイル デプロイメント記述子を表します。
   - **[CMP] :** この EJB の `weblogic-cmp-rdbms-jar.xml` ファイル デプロイメント記述子を表します。
6. 編集するデプロイメント記述子のノードを展開します。

   選択した EJB の現在のデプロイメント記述子が左ペインに、コンフィグレーションされている設定が右ペインに表示されます。リストの項目を右クリックすると、その項目のダイアログ ウィンドウが右ペインに表示されます。
7. 丸をクリックすると、さまざまな設定が右ペインのダイアログ ウィンドウに表示されます。

ダイアログ ウィンドウの設定を変更すると、デプロイメント記述子を編集できます。

8. フォルダをクリックすると、設定を表示するテーブルが右ペインに表示されます。

   通常、ここで新しい記述子をコンフィグレーションしたり、既存の設定を参照したりします。下線が付いている表の項目をクリックすると、設定を変更するためのダイアログが表示されます。

9. 右ペインでデプロイメント記述子の項目を右クリックすると、記述子を削除することもできます。

注意： EJB デプロイメント記述子の詳細については、Administration Console のオンライン ヘルプまたは第 9 章「weblogic-ejb-jar.xml 文書型定義」および第 10 章「weblogic-cmp-rdbms-jar.xml 文書型定義」を参照してください。

他の EJB およびリソースへの参照

デプロイメント記述子の EJB 参照を指定することにより、EJB が WebLogic Server にデプロイされている他の EJB をルックアップし使用することができます。EJB 参照を作成する際の要件は、参照される EJB が呼び出し側の EJB にとって外部的か、もしくは、同じアプリケーション EAR ファイルの一部としてデプロイされているかによって異なります。

外部 EJB の参照

外部 EJB を参照するためには、呼び出し側の EJB の `weblogic-ejb-jar.xml` ファイルに `<reference-descriptor>` スタンザを追加します。次の XML コードに、外部 EJB を参照するスタンザの例を示します。

図 6-2 外部 EJB を参照する XML コード例

```
<reference-descriptor>
   <ejb-reference-description>
      <ejb-ref-name>AdminBean</ejb-ref-name>
      <jndi-name>payroll.AdminBean</jndi-name>
```

```
   </ejb-reference-description>
76</reference-descriptor>
```

このスタンザ内の ejb-ref-name 要素は、呼び出し側 EJB がその外部 EJB をルックアップする際に使用する名前を指定しています。jndi-name 要素は、指定された ejb-ref-name をルックアップするときに使用する汎用的な JNDI 名を指定しています。

アプリケーション スコープの EJB の参照

同じ EAR ファイルの一部として複数の EJB をデプロイする場合、アプリケーションのローカル JNDI ツリーに一連の EJB 名を追加します。そのアプリケーション内の EJB およびそれ以外のコンポーネントは、java:comp/env から相対的に、JNDI ツリー内にあるアプリケーション スコープの他のコンポーネントを直接、ルックアップできます。

ある EJB が同じ EAR ファイルの一部としてデプロイされている他の EJB を参照する場合には、weblogic-ejb-jar.xml ファイルで汎用的な JNDI 名を指定する必要はありません。実際には、WebLogic 固有の他の機能のデプロイメント記述子が必要なければ、weblogic-ejb-jar.xml ファイルまるごとを省略できます。

同じ EAR ファイルの一部としてデプロイされた EJB を参照するためには、呼び出し側の EJB の ejb-jar.xml デプロイメント記述子ファイルに <ejb-local-ref> スタンザを追加します。次に例を示します。

図 6-3 同じ EAR ファイルの EJB を参照する XML コード例

```
<ejb-local-ref>
   <description>Reference to application EJB</description>
   <ejb-ref-name>ejb1</ejb-ref-name>
   <ejb-ref-type>Session</ejb-ref-type>
   <local-home>mypackage.ejb1.MyHome</home>
   <local>mypackage.ejb1.MyRemote</local>
   <ejb-link>ejb1.jar#myejb</ejb-link>
</ejb-local-ref>
```

上記の例で、ejb-ref-name 要素は、呼び出し側の EJB がアプリケーション スコープの EJB をルックアップするのに使用する名前を示します。ejb-link 要素は、指定された <ejb-ref-name> を、その EAR ファイルでデプロイされる別の EJB にマップします。この例で <ejb-link> の名前が 2 番目の EJB を保存するファイル名で修飾されている点に注意してください。この EAR ファイル内の 2

つ以上の EJB が同じ名前を持つ場合には、このように EJB 名を修飾することが必要になります。これにより、ファイル名の修飾子による参照の一意性が保証されます。

EJB リンクの詳細については、5-61 ページの「EJB リンクの使用」を参照してください。

アプリケーション スコープの JDBC データソースの参照

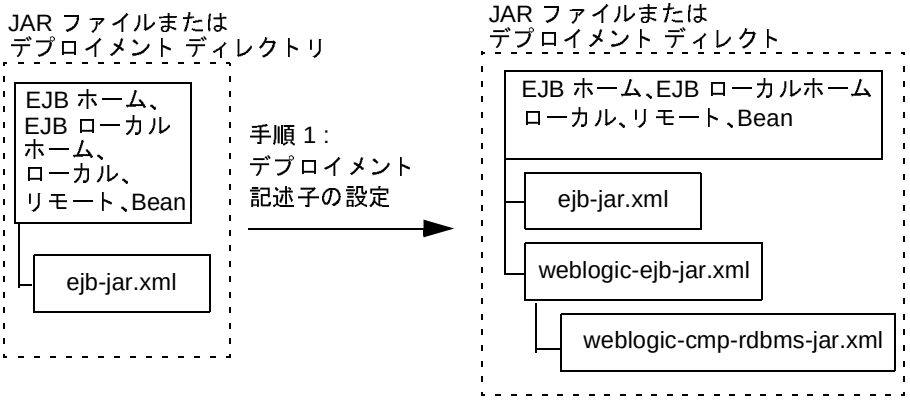
EJB が、同じ EAR ファイルの一部としてデプロイされている JDBC データソースにアクセスすることもできます。`weblogic-application.xml` デプロイメント記述子で示されるデータソースに、`java:comp/env` からローカルにアクセスできます。詳細については、『Web アプリケーションのアセンブルとコンフィグレーション』の「アプリケーション スコープのリソースのコンフィグレーション」を参照してください。

デプロイメント ディレクトリへの EJB のパッケージ化

デプロイメント プロセスは、EJB プロバイダによって作成されたコンパイル済み EJB インタフェースと実装クラスを格納する JAR ファイルまたはデプロイメント ディレクトリで開始されます。JAR ファイルとデプロイメント ディレクトリは、どちらがコンパイル済みクラスを格納している場合でも、Java パッケージ構造と一致するサブディレクトリに入っている必要があります。

また、EJB プロバイダが、付属の EJB を記述する EJB 準拠の `ejb-jar.xml` ファイルを提供する必要があります。`ejb-jar.xml` ファイルとその他に必要な XML デプロイメント ファイルの場所は、JAR またはデプロイメント ディレクトリの `META-INF` サブディレクトリの最上位でなければなりません。次の図は、EJB とデプロイメント記述子ファイルをデプロイメント ディレクトリまたは JAR ファイルにパッケージ化する作業の第 1 段階を示しています。

図 6-4 デプロイメント ディレクトリへの EJB クラスとデプロイメント記述子のパッケージ化



基本の JAR またはデプロイメント ディレクトリは、そのまま WebLogic Server にデプロイすることができません。まず、`weblogic-ejb-jar.xml` ファイルの WebLogic 固有のデプロイメント記述子要素を作成してコンフィグレーションし、そのファイルをデプロイメント ディレクトリまたは `ejb.jar` ファイルに追加します。デプロイメント記述子ファイルの作成手順については、6-3 ページの「WebLogic Server の EJB デプロイメント ファイル」を参照してください。

コンテナ管理の永続性を使用するエンティティ EJB をデプロイする場合は、Bean の永続性タイプに対応する WebLogic 固有のデプロイメント記述子要素も追加する必要があります。通常、WebLogic Server のコンテナ管理による永続性 (CMP) サービスの場合、ファイルの名前は `weblogic-cmp-rdbms-jar.xml` です。CMP を使用する Bean ごとに別々のファイルが必要です。サードパーティの永続性ベンダを使用する場合は、`weblogic-cmp-rdbms-jar.xml` とは内容だけでなくファイル タイプも異なることがあるので、詳細については、永続性ベンダのマニュアルを参照してください。

EJB に必要なデプロイメント記述子ファイルがない場合は、手動で作成しなければなりません。既存のファイルをコピーした上で、必要に応じて EJB の設定を編集する方法が最も簡単です。ファイルを作成するには、6-6 ページの「EJB デプロイメント記述子の指定と編集」の手順に従います。

ejb.jar ファイル

ejb.jar ファイルを作成するには、Java Jar ユーティリティ (javac) を使用します。このユーティリティは、EJB クラスとデプロイメント記述子を、ディレクトリ構造を保持する 1 つの Java アーカイブ (JAR) ファイルにまとめます。ejb-jar ファイルが、WebLogic Server にデプロイするユニットとなります。

EJB クラスのコンパイルと EJB コンテナ クラスの生成

デプロイメント ユニットの作成手順の一部として、EJB クラスをコンパイルし、デプロイメント記述子をデプロイメント ユニットに追加し、デプロイメント ユニットにアクセスするためのコンテナ クラスを作成する必要があります。

1. コマンド ラインから javac コンパイラを使用して、EJB クラスをコンパイルします。
2. 6-3 ページの「WebLogic Server の EJB デプロイメント ファイル」のガイドラインに従って、適切な XML デプロイメント記述子ファイルをコンパイル済みユニットに追加します。
3. ejbc を使用して、Bean にアクセスするためのコンテナ クラスを生成します。

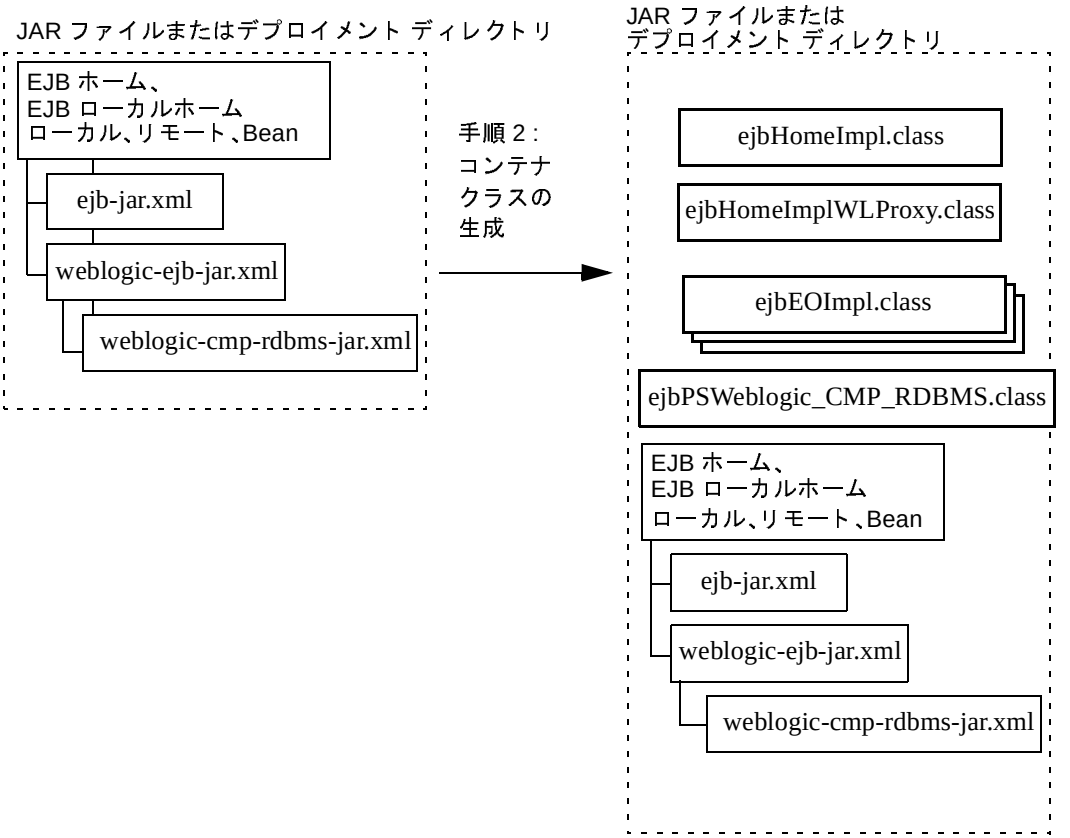
   コンテナ クラスには、WebLogic Server が使用する EJB の内部表現に加えて、クライアントが使用する外部インタフェース (ホーム、ローカル、またはリモート) の実装も格納されます。

ejbc コンパイラは、WebLogic 固有の XML デプロイメント記述子ファイルで指定した XML デプロイメント記述子ファイルに従ってコンテナ クラスを生成します。たとえば EJB をクラスタで使用するよう指定した場合、ejbc は、そのデプロイメント用の特別なクラスタ対応クラスを作成します。

また、コマンド ラインから ejbc を直接使用して、必要なオプションと引数を指定することもできます。詳細については、8-21 ページの「ejbc」を参照してください。

次の図は、JAR ファイルの作成時にデプロイメント ユニットに追加されるコンテナ クラスを示しています。

図 6-5 EJB コンテナ クラスの生成



デプロイメント ユニットの作成後、JAR、EAR、または WAR アーカイブのいずれかとしてファイル拡張子を指定できます。

生成クラス名の衝突の可能性

まれなことではありますが、ejbc でクラス名を生成したとき、生成したクラス名の衝突に遭遇し、これが `ClassCastException` や他の例外を招くことがあります。これは、生成されるクラス名が、Bean クラス名、Bean クラス パッケージ、その Bean の ejb-name の 3 つのキーをもとにしているためです。この問題が起こるのは、複数の JAR ファイルを含む EAR ファイルを使用し、その 2 つ以上の JAR ファイルのそれぞれに Bean クラス、パッケージ、または、クラス名を同じくする EJB が含まれ、双方の EJB がそれぞれの JAR ファイル内で同じ ejb-name を持っている場合です。この問題が発生した場合は、いずれかの Bean の `ejb-name` をユニークなものに変更してください。

`ejb-name` はファイル名の基となるキーの 1 つであり、`ejb-name` は JAR ファイル内でユニークでなければならないので、同じ JAR ファイルにある 2 つの EJB 間ではこの問題は起きません。また、EAR ファイルではファイルごとに個別のクラスローダになるため、別々の EAR ファイルに置かれた 2 つの EJB 間でこの問題は起きません。

WebLogic Server への EJB クラスのロード

Weblogic Server のクラスローダは階層的です。WebLogic Server の起動時に、Java システム クラスローダはアクティブになり、その後に WebLogic Server が作成するすべてのクラスローダの親になります。WebLogic Server では、アプリケーションをデプロイするときに、EJB 用と Web アプリケーション用の 2 つの新しいクラスローダを作成します。EJB クラスローダは Java システム クラスローダの子、Web アプリケーション クラスローダは EJB クラスローダの子です。

クラスローダの詳細については、『WebLogic Server アプリケーションの開発』の「クラスローダの概要」と「アプリケーションのクラスローダ」を参照してください。

ejb-client.jar の指定

WebLogic Server では、ejb-client.jar ファイルを使用できます。

ejb-client.jar には、ホーム インタフェースとリモート インタフェース、主キー クラス (適当な場合)、およびそれらが表すファイルが格納されます。WebLogic Server は、クラスパスで参照されるファイルを ejb-client.jar に追加しません。これにより、WebLogic Server では、java.lang.String などの汎用クラスを追加することなく、必要なカスタム クラスを ejb-client.jar に追加することができます。

たとえば、ShoppingCart リモート インタフェースが Item クラスを返すメソッドを持っているとします。このクラスはこのリモート インタフェースが参照し、ejb-jar ファイルに入っているので、クライアント Jar に含まれます。

ejb-client.jar ファイルの作成は、Bean の ejb-jar.xml デプロイメント記述子ファイルでコンフィグレーションします。ejbc で Bean をコンパイルすると、WebLogic Server が ejb-client.jar を作成します。

ejb-client.jar を指定するには、次の手順に従います。

1. コマンド ラインから javac コンパイラを使用して、Bean の Java クラスをディレクトリにコンパイルします。
2. 6-3 ページの「WebLogic Server の EJB デプロイメント ファイル」のガイドラインに従って、EJB XML デプロイメント記述子ファイルをコンパイル済みユニットに追加します。
3. Bean の ejb-jar.xml ファイルの ejb-client-jar デプロイメント記述子を次のように編集して、ejb-client.jar のサポートを指定します。

```
<ejb-client-jar>ShoppingCartClient.jar</ejb-client-jar>
```

4. weblogic.ejbc を使用して Bean にアクセスするためのコンテナ クラスを作成し、次のコマンドを使用して ejb-client.jar を作成します。

```
$ java weblogic.ejbc <ShoppingCart.jar>
```

コンテナ クラスには、WebLogic Server が使用する EJB の内部表現に加えて、クライアントが使用する外部インタフェース (ホーム、ローカル、またはリモート) の実装も格納されます。

外部クライアントは、`ejb-client.jar` をそれぞれのクラスパスに含めます。Web アプリケーションは、`ejb-client.jar` を `\lib` ディレクトリに含めます。

注意 : WebLogic Server のクラスローディング動作は、クライアントがスタンドアロンかどうかによって異なります。`ejb-client.jar` にアクセスできるスタンドアロンのクライアントは、ネットワーク経由で必要なクラスをロードできます。ただしセキュリティ上の理由から、サーバ インスタンスで動作しているプログラムに基づくクライアントはネットワーク経由でクラスをロードできません。

マニフェスト クラスパス

JAR ファイルが別の JAR ファイルを参照できるかどうかを指定するには、マニフェスト ファイルを使用します。スタンドアロン EJB ではマニフェスト クラスパスを使用できません。マニフェスト クラスパスは、EAR ファイル内にデプロイされているコンポーネントに対してのみサポートされています。クライアントは、マニフェスト ファイルのクラスパス エントリにある `client.jar` を参照します。

マニフェスト ファイルを使用して別の JAR ファイルを参照するには、次の手順に従います。

1. 参照先 JAR ファイルの名前を、参照元 JAR ファイルのマニフェスト ファイルの Class-Path ヘッダに指定します。

   参照先 JAR ファイルの名前には、参照元 JAR ファイルの URL を基準にした URL を使用します。

2. マニフェスト ファイル、META-INF/MANIFEST.MF を JAR ファイルに指定します。
3. マニフェスト ファイルの Class-Path エントリは次のようになります。

   ```
   Class-Path:AAyy.jar BByy.jar CCyy.jar.
   ```

注意： このエントリは、スペース区切りの JAR ファイル リストです。

EJB のホーム / リモート インタフェースを呼び出し側コンポーネントのクラスパスに配置するには、次の手順に従います。

1. `ejbc` を使用して、EJB を JAR ファイルにコンパイルします。
2. `client.jar` ファイルを作成します。`client.jar` の使用方法については、6-16 ページの「ejb-client.jar の指定」を参照してください。
3. `client.jar` を Bean のすべてのクライアントと一緒に EAR に配置します。
4. EAR をマニフェスト ファイルで参照します。

7 WebLogic Server への EJB のデプロイ

以下の各節では、WebLogic Server の起動時、または 動作中の WebLogic Server に EJB をデプロイする手順について説明します。

- 役割と分担
- WebLogic Server 起動時の EJB のデプロイメント
- 動作中の WebLogic Server への EJB のデプロイ
- 動作中の環境への新しい EJB のデプロイメント
- デプロイ済み EJB のアンデプロイ
- EJB の再デプロイ
- コンパイル済み EJB ファイルのデプロイ
- 未コンパイルの EJB ファイルのデプロイ
- コンテナ管理による関係に関するデプロイメントの制限

役割と分担

以降の節は主に次の読者を対象としています。

- WebLogic Server コンテナで動作するように EJB をコンフィグレーションするデプロイヤ
- 複数の EJB と EJB リソースをリンクしてより大規模な Web アプリケーション システムを作成するアプリケーション アセンブラ
- 新規の EJB JAR ファイルを作成およびコンフィグレーションする EJB 開発者

WebLogic Server の 1 つまたは複数のインスタンスに EJB を作成、変更、およびデプロイできます。EJB デプロイメントを設定し、EJB 参照を実際のリソース ファクトリ、ロール、およびサーバ上で使用可能な他の EJB に割り当てるには、XML デプロイメント記述子ファイルを編集します。

WebLogic Server 起動時の EJB のデプロイメント

WebLogic Server の起動時に EJB を自動的にデプロイするには、次の手順に従います。

1. 6-6 ページの「EJB デプロイメント記述子の指定と編集」の手順に従って、必要な WebLogic Server XML デプロイメント ファイルがデプロイ可能な EJB JAR ファイルまたはデプロイメント ディレクトリに入っていることを確認します。
2. テキスト エディタまたは Administration Console の EJB デプロイメント記述子エディタを使用して、XML デプロイメント記述子要素を必要に応じて編集します。
3. 6-13 ページの「EJB クラスのコンパイルと EJB コンテナ クラスの生成」の手順に従って、WebLogic Server に必要な実装クラスをコンパイルします。

   コンテナ クラスをコンパイルすると、デプロイメント ディレクトリに JAR ファイルが格納されます。EJB を WebLogic Server の起動時に自動的にデプロイする場合は、デプロイする EJB を次のディレクトリに格納します。

   `mydomain\applications` ディレクトリ

   EJB JAR ファイルが別のディレクトリにある場合、このファイルを起動時にデプロイするには、このディレクトリにコピーしておく必要があります。
4. WebLogic Server を起動します。

   起動すると WebLogic Server は、指定した EJB JAR ファイルまたはデプロイメント ディレクトリを自動的にデプロイしようとします。
5. Administration Console を起動します。

6. 左ペインで [デプロイメント] をクリックし、[EJB] ノードをクリックします。

   サーバの EJB デプロイメントのリストがノードの下に表示されます。

異なるアプリケーションへの EJB のデプロイメント

リモート呼び出しによって複数の異なったアプリケーションに EJB をデプロイするとき、EJB を呼び出すために `call-by-reference` は使用できません。代わりに、`pass-by-value` を使用します。一般に、相互に対話するコンポーネントは `call-by-reference` を使用できるように同じアプリケーション内に配置する必要があります。デフォルトでは、同じサーバから呼び出された EJB メソッドは引数を参照で渡します。パラメータはコピーされないので、これによってメソッド呼び出しのパフォーマンスが向上します。EJB がリモートで (同じサーバ以外から) 呼び出される場合は、`pass-by-value` にする必要があります。

動作中の WebLogic Server への EJB のデプロイ

EJB JAR ファイルまたはデプロイメント ディレクトリを `wlserver/config/mydomain/applications` ディレクトリに配置すると EJB を直ちにデプロイできますが、デプロイ済みの EJB を変更した場合は、その変更を有効にするために EJB を再デプロイする必要があります。

WebLogic Server を再起動できない場合に備えて、自動デプロイメントという方法が用意されています。自動デプロイメントでは、更新された EJB のみを管理サーバにデプロイし、ドメインの管理対象サーバには EJB をデプロイしません。自動デプロイメント機能を使用すると、次の作業を行うことができます。

- 新しく開発した EJB を動作中のプロダクション システムにデプロイする

- デプロイ済みの EJB を削除して、データへのアクセスを制限する
- デプロイ済みの EJB 実装クラスを更新して、バグを修正したり、新機能をテストしたりする

コマンド ラインまたは Administration Console から EJB をデプロイする場合でも、更新する場合でも、自動デプロイメント機能を利用することになります。以降の節では、自動デプロイメントの概念と手順について説明します。

EJB デプロイメント名

EJB JAR ファイルまたはデプロイメント ディレクトリをデプロイする場合は、デプロイメント ユニットの名前を指定します。この名前を使用すると、後で EJB をアンデプロイしたり更新したりする場合に、EJB デプロイメントを簡単に参照できます。

EJB をデプロイする場合は、WebLogic Server が、JAR ファイルまたはデプロイメント ディレクトリのパスおよびファイル名と一致するデプロイメント名を明示的に割り当てます。この名前を使用すると、サーバが起動した後に Bean をアンデプロイまたは更新できます。

注意： EJB デプロイメント名は、サーバが再起動されるまで、WebLogic Server 内でアクティブなままです。EJB をアンデプロイしても、関連付けられたデプロイメント名は削除されません。Bean をデプロイするために後でその名前を使う場合があるからです。

動作中の環境への新しい EJB のデプロイメント

デプロイされていない EJB JAR ファイルまたはデプロイメント ディレクトリを WebLogic Server をデプロイするには、次の手順に従います。

1. WebLogic Server Administration Console を起動します。
2. 作業対象のドメインを選択します。
3. Console の左ペインで、[デプロイメント] をクリックします。

4. Console の左ペインで、[EJB] をクリックします。Console の右ペインのテーブルに、デプロイされたすべての EJB が表示されます。
5. [新しい EJB のコンフィグレーション] を選択します。
6. コンフィグレーションする EAR、WAR、または JAR ファイルを指定します。展開されたアプリケーションまたはコンポーネント ディレクトリをコンフィグレーションすることもできます。WebLogic Sever は、指定したディレクトリおよびその下位ディレクトリで見つかった全コンポーネントをデプロイします。
7. ディレクトリまたはファイルの左側の [select] をクリックし、目的のファイルを選択して次の手順に進みます。
8. [使用可能なサーバ] から対象サーバを選択します。
9. 表示されたフィールドに EJB またはアプリケーションの名前を入力します。
10. [コンフィグレーションとデプロイ] をクリックし、EJB またはアプリケーションをインストールします。Administration Console の [デプロイ] パネルに、EJB のデプロイメント ステータスとデプロイメント アクティビティが一覧表示されます。
11. 使用可能なタブで、以下の情報を入力します。
    - [コンフィグレーション] - ステージング モードを編集し、デプロイメント順を入力します。
    - [対象] - [選択可] リストから [選択済み] リストにサーバを移動し、このコンフィグレーション済み EJB またはアプリケーションの対象サーバを指定します。
    - [デプロイ] - EJB、アプリケーションを、すべての対象サーバ、または選択した対象サーバにデプロイします。あるいは、すべての対象サーバ、または選択した対象サーバから EJB、アプリケーションをアンデプロイします。
    - [モニタ] - EJB またはアプリケーションのセッションをモニタします。
    - [メモ] - EJB またはアプリケーションに対する注釈を入力します。

固定 EJB のデプロイメント - 特別な手順が必要

.jar ファイルに未コンパイルのクラスおよびインタフェースが含まれる場合に、クラスタ内の 1 つのサーバ インスタンスに EJB をデプロイまたは再デプロイする (固定デプロイメントと呼ばれる) と、問題が生じることが知られています。

未コンパイルの EJB は、デプロイ中にクラスタ内の各サーバ インスタンスにコピーされますが、この EJB はデプロイ先のサーバ インスタンス上でしかコンパイルされません。その結果、EJB の対象とされなかったクラスタのサーバ インスタンスには、コンパイルの過程で生成される、EJB の呼び出しに必要なクラスが存在しません。別のサーバ インスタンスのクライアントが固定 EJB を呼び出そうとしても失敗し、RMI レイヤでアサーション エラーが送出されます。

クラスタ内の 1 つのサーバ インスタンスに EJB をデプロイまたは再デプロイする場合は、生成されたクラスが、クラスタ内のすべてのノードに利用可能なすべてのサーバ インスタンスに必ずコピーされるように、デプロイ前にその EJB を ejbc でコンパイルしてください。

固定デプロイメントの詳細については、「WebLogic クラスタの設定」を参照してください。

デプロイ済み EJB の表示

デプロイ済み EJB を表示するには、次の手順に従います。

1. Administration Console を起動します。
2. 左ペインの [デプロイメント] ノードをクリックし、EJB サブノードを選択します。ドメインにデプロイされた EJB のリストが、[EJB] の下と右ペインに表示されます。

デプロイ済み EJB のアンデプロイ

EJB のアンデプロイメントは、すべてのクライアントにその EJB を使用できなくする効果的な方法です。EJB をアンデプロイすると、直ちに、指定した EJB の実装クラスがサーバ内で使用不可になったことが示されます。WebLogic Server は実装クラスを自動的に削除して、その Bean を使用していたすべてのクライアントに `UndeploymentException` を伝播します。

アンデプロイメントによって、指定した EJB のパブリック インタフェース クラスがすべて自動的に削除されるわけではありません。これらのクラスへの参照がすべて解放されるまで、パブリック インタフェースで参照されるホーム インタフェース、リモート インタフェース、およびすべてのサポート クラスの実装は、サーバ内に残ります。パブリック クラスは、参照が解放された時点で、通常の Java ガベージ コレクション ルーチンによって削除できます。

同様に、EJB をアンデプロイしても、`ejb.jar` ファイルまたはデプロイメント ディレクトリに関連付けられたデプロイメント名は削除されません。デプロイメント名は、後で EJB を更新することができるようにサーバ内に残ります。

EJB のアンデプロイメント

デプロイ済み EJB をアンデプロイするには次の手順に従います。

WebLogic Server Administration Console を使用する場合

1. 左ペインでコンポーネントを選択します。
2. コンポーネントの [デプロイメント] テーブルで、アンデプロイするコンポーネントを選択します。
3. [適用] をクリックします。

EJB をアンデプロイしても、WebLogic Server から EJB デプロイメント名は削除されません。EJB は、アンデプロイされた後に変更された場合を除いて、サーバ セッションが持続している間、アンデプロイされたままです。サーバを再起動す

るまで、deploy 引数でデプロイメント名を再利用することはできません。次の項で説明するように、デプロイメントの更新にそのデプロイメント名を再使用できます。

EJB の再デプロイ

デプロイ済みの EJB のクラスを変更した場合、以下のいずれかを行うまで、変更は WebLogic Server で反映されません。

- サーバを再起動する (JAR またはディレクトリを自動的にデプロイする場合)。

  または

- EJB デプロイメントを再デプロイする。

EJB デプロイメントを再デプロイすると、EJB プロバイダがデプロイ済みの EJB のクラスを変更し、再コンパイルしてから、動作中のサーバのクラスを「更新」できるようになります。

再デプロイ プロセス

EJB デプロイメントを再デプロイすると、直ちに、ロード済みの EJB のクラスがサーバ内で使用不可になったことが示され、EJB のクラスローダと関連クラスが削除されます。同時に、改版された EJB クラスをロードして維持する新規の EJB クラスローダが作成されます。

クライアントが次に EJB への参照を必要とする場合、クライアントの EJB メソッドの呼び出しでは、変更済みの EJB クラスが使用されます。

再デプロイ手順

スタンドアロンまたはアプリケーションの一部を構成する EJB を再デプロイするには、`weblogic.Deployer` ツールまたは Administration Console を使用します。

`weblogic.Deployer` を使用して再デプロイするには、次の手順に従います。

1. `-deploy` フラグを使用します。

   ```
   java weblogic.Deployer -deploy ejb_name
   ```

WebLogic Server Administration Console を使用して再デプロイするには、次の手順に従います。

1. コンソールの左ペインの [デプロイメント] ノードから [EJB] を選択するか、または EJB がアプリケーションの一部となっている場合は、[アプリケーション] を選択してから、アプリケーション名を選択します。
2. 再デプロイする EJB の名前をクリックします。
3. 右ペインの [デプロイ] タブをクリックします。
4. [再デプロイ] をクリックします。

コンパイル済み EJB ファイルのデプロイ

コンパイル済みの EJB 2.0 JAR または EAR ファイルを作成するには、次の手順に従います。

1. `javac` を使用して、EJB クラスとインタフェースをコンパイルします。
2. EJB クラスとインタフェースを有効な JAR または EAR ファイルにパッケージ化します。
3. JAR ファイルに対して `weblogic.ejbc` を使用して、WebLogic Server コンテナ クラスを生成します。ejbc の使用方法については、8-21 ページの「ejbc」を参照してください。

以前のバージョンの WebLogic Server からコンパイル済みの EJB を作成するには、次の手順に従います。

1. EJB JAR ファイルに対して `weblogic.ejbc` を実行して、EJB 2.0 のコンテナクラスを生成します。
2. コンパイルした ejb JAR ファイルを次の場所にコピーします。

   `mydomain\applications\DefaultWebApp` ディレクトリ

注意： 以前のバージョンの EJB は、EJB コンテナにデプロイする前に、すべて手動で再コンパイルする必要があります。これをしない場合、WebLogic Server が自動的にそれらの EJB を再コンパイルし、エラーがあればコンパイラから出力が別のログ ファイルに送られます。

(再パッケージ化、再コンパイル、または既存の `ejb.jar` ファイルにコピーして) `applications` 内のコンパイル済み `ejb.jar` ファイルの内容を変更した場合、WebLogic Server は、自動デプロイメント機能を利用して、`ejb.jar` を自動的に再デプロイしようとします。

未コンパイルの EJB ファイルのデプロイ

WebLogic Server コンテナを使用すると、未コンパイルの EJB クラスおよびインタフェースが入った JAR ファイルも自動的にデプロイすることができます。未コンパイル EJB ファイルはコンパイル済みファイルと同じ構造を持っていますが、次の点で異なります。

- 個々のクラス ファイルとインタフェースをコンパイルする必要がありません。
- パッケージ化された JAR ファイルに対して `weblogic.ejbc` を使用して、WebLogic Server コンテナ クラスを生成する必要がありません。

JAR ファイル内の `.java` または `.class` ファイルは、Java パッケージ階層と同じサブディレクトリにパッケージ化する必要があります。また、すべての `ejb.jar` ファイルと同じように、適切な XML デプロイメント ファイルを `META-INF` ディレクトリの最上位に置く必要があります。

未コンパイルのクラスをパッケージ化したら、JAR を `wlserver\config\mydomain\applications` ディレクトリにコピーするだけです。WebLogic Server は、必要に応じて、`javac` を自動的に実行して (またはユーザがコンパイラを指定して)、`.java` ファイルをコンパイルし、`weblogic.ejbc` を実行してコンテナ クラスを生成します。コンパイルされたクラスは、`mydomain\applications\DefaultWebApp` の新しい JAR ファイルにコピーされ、EJB コンテナにデプロイされます。

(再パッケージ化するか、JAR ファイルにコピーするかして) `applications` ディレクトリ内の未コンパイル ejb.jar を変更した場合、WebLogic Server は、変更された JAR を同じ手順で自動的に再コンパイルして再デプロイします。

コンテナ管理による関係に関するデプロイメントの制限

コンテナ管理による関係を持つ EJB は、同じ JAR ファイルでデプロイする必要があります。コンテナ管理による関係の詳細については、5-44 ページの「コンテナ管理による関係」を参照してください。

8 WebLogic Server EJB のユーティリティ

以下の節は、EJB で使用する WebLogic Server に付属のユーティリティおよびサポート ファイルの詳細なリファレンスです。

- EJBGen
- ejbc (`weblogic.ejbc`)
- DDConverter (`weblogic.ejb.utils.DDConverter`)
- weblogic.Deployer (`weblogic.Deployer`)
- weblogic.deploy (`weblogic.deploy`)

EJBGen

`EJBGen` は、エンタープライズ JavaBean 2.0 のコード ジェネレータです。Bean クラスファイルに javadoc タグで注釈を付けた後、`EJBGen` を使用して EJB アプリケーションのリモート クラス、ホーム クラス、および、デプロイメント記述子ファイルを生成することができるので、編集および管理すべき EJB ファイルの数を 1 つに減らせます。

BEA WebLogic 7.0 のサンプルをインストール済みであれば、*`SAMPLES_HOME`*`\server\src\examples\ejb20\ejbgen` に、EJBGen を使用した ¢Bands£ という名前のサンプル アプリケーションがあります。

EJBGen 構文

```
javadoc -docletpath weblogic.jar -doclet
weblogic.tools.ejbgen.EJBGen (YourBean).java
```

クラスパス内に weblogic.jar がない場合は、次のように weblogic.jar にパスを追加します。

```
javadoc -docletpath <path_to_weblogic.jar> weblogic.jar -doclet
weblogic.tools.ejbgen.EJBGen (YourBean).java
```

他の EJB との関係がある EJB に対して EJBGen を呼び出す場合は、次のように EJB の後に、関係する EJB を指定してその EJB を呼び出します。

```
javadoc -docletpath weblogic.jar -doclet
weblogic.tools.ejbgen.EJBGen (YourBean).java (RelatedBean).java
```

EJBGen では次のオプションを使用します。

オプション	定義
-d [directory]	このディレクトリの下にすべてのファイルが作成される。
-ignorePackage	このフラグを設定すると、EJBGen は生成した Java ファイルのパッケージ名を無視し、-d フラグで指定された出力ディレクトリにファイルを作成する (-d フラグが指定されていない場合は、カレント ディレクトリ)。
-pfd1	このフラグを設定すると、EJBGen は EJB 2.0 の最終草案 バージョン 1 と互換したデプロイメント記述子を生成する。Weblogic 6.1 より前のバージョンを使用する場合には、このフラグを必ず使用する。
-ejbPrefix [string] (default: "")	EJB クラス生成時に使用するプレフィックス。
-ejbSuffix [string] (default: "Bean" or "EJB")	EJB クラス生成時に使用するサフィックス。
-localHomePrefix [string] (default: "")	ローカル EJB クラス生成時に使用するプレフィックス。

オプション	定義
-localHomeSuffix [string] (default: "LocalHome")	ローカル EJB クラス生成時に使用するサフィックス。
-remoteHomePrefix [string] (default: "")	リモート EJB ホーム クラス生成時に使用するプレフィックス。
-remoteHomeSuffix [string] (default: "Home")	リモート EJB ホーム クラス生成時に使用するサフィックス。
-remotePrefix [string] (default: "")	リモート EJB クラス生成時に使用するプレフィックス。
-remoteSuffix [string] (default: "")	リモート EJB クラス生成時に使用するサフィックス。
-localPrefix [string] (default: "")	ローカル EJB クラス生成時に使用するプレフィックス。
-localSuffix [string] (default: "Local")	ローカル EJB クラス生成時に使用するサフィックス。
-valueObjectPrefix [string] (default: "")	Value オブジェクト クラス生成時に使用するプレフィックス。
-valueObjectSuffix [string] (default: "Value")	Value オブジェクト クラス生成時に使用するサフィックス。
-jndiPrefix [string] (default: "")	@remote-jndi-name と @local-jndi-name で使用するプレフィックス。
-jndiSuffix [string] (default: "")	@remote-jndi-name と @local-jndi-name で使用するサフィックス。
-checkTags	このオプションで起動した場合、EJBGen はどのクラスも生成しないが、コマンド ラインで指定したクラスに対して有効でない EJBGen タグを見つける。

オプション	定義
-docTags	EJBGen によって把握された、すべてのタグを出力する。注意点として、このオプションでどのソースファイルも必要ないにもかかわらず、コマンドラインで既存の Java クラスを指定しなくてはならない。これをしないと、このフラグを認識したにもかかわらず、Javadoc はエラーメッセージを返す。
-docTag tagName	把握されているすべての属性など、このタグの詳細な説明を出力する。注意点として、このオプションでどのソースファイルも必要ないにもかかわらず、コマンドラインで既存の Java クラスを指定しなくてはならない。これをしないと、このフラグを認識したにもかかわらず、Javadoc はエラーメッセージを返す。
-docTagsHtml	-docTags と同じであるが、HTML 文書を生成する。
-propertyFile [fileName]	EJBGen が置換変数を定義するのに読み込む、プロパティ ファイルの名前。置換変数のドキュメントを参照のこと。
-valueBaseClass [className]	削除された。value.baseClass 変数を使用する。
-noValueClasses	これを指定すると、Value クラスは生成されなくなる。

EJBGen の例

注釈を付けた Bean ファイルを次に示しますが、これを基に EJBGen はリモート インタフェース、ホーム インタフェース、デプロイメント記述子ファイルを生成します。AccountBean.java がメインの Bean クラスです。これは CMP EJB 2.0 エンティティ Bean です。

```
/**
 * @ejbgen:entity
 *   ejb-name = AccountEJB-OneToMany
 *   data-source-name = examples-dataSource-demoPool
 *   table-name = Accounts
 *   prim-key-class = java.lang.String
 *
 * @ejbgen:jndi-name
 *   local = one2many.AccountHome


 * @ejbgen:finder
 *   signature = "Account findAccount(double balanceEqual)"
 *   ejb-ql = "WHERE balance = ?1"
 *
 * @ejbgen:finder
 *   signature = "Collection findBigAccounts(double
balanceGreaterThan)"
 *   ejb-ql = "WHERE balance > ?1"
 *
 * @ejbgen:relation
 *   name = Customer-Account
 *   target-ejb = CustomerEJB-OneToMany
 *   multiplicity = many
 *   cmr-field = customer
 *
```

```
 */
abstract public class AccountBean implements EntityBean {


  /**
   * @ejbgen:cmp-field column = acct_id
   * @ejbgen:primkey-field
   * @ejbgen:remote-method transaction-attribute = Required
   */
  abstract public String getAccountId();
  abstract public void setAccountId(String val);
  // ....
}
```

この例でも見られるように、タグには、それが使用できる場所に応じてクラス タグとメソッド タグの 2 種類があります。

ファイルの編集を終えたら、次の javadoc コマンドを通して EJBGen を起動します。

```
javadoc -docletpath weblogic.jar -doclet
weblogic.tools.ejbgen.EJBGen AccountBean.java
```

javadoc が存在すれば、次のファイルが生成されます。

- Account.java
- AccountHome.java
- ejb-jar.xml
- weblogic-ejb-jar.xml
- weblog-cmp-rdbms-jar.xml

EJBGen タグ

次のタグを使って Bean ファイルに注釈を付けます。

@ejbgen:automatic-key-generation

使用場所 : クラス

適用対象 : エンティティ Bean

属性	説明	必須
cache-size	主要キャッシュのサイズ。	必須
name	ジェネレータの名前。	必須
type	ジェネレータのタイプ。	必須

@ejbgen:cmp-field

使用場所 : メソッド

適用対象 : エンティティ Bean

属性	説明	必須
column	CMP フィールドがマップされるカラム。	必須
column-type	このカラムのタイプ (OracleClob | OracleBlob)。	省略可能
ordering-number (0..n)	このフィールドがシグネチャおよびコンストラクタ内に置かれるべき順番。この順序付けを有効にするためには、すべての CMR および CMP フィールドがこの属性で異なる数値を持つ必要がある。	省略可能

@ejbgen:cmr-field

使用場所 : メソッド

適用対象 : エンティティ

属性	説明	必須
ordering-number (0..n)	このフィールドがシグネチャおよびコンストラクタ内に置かれるべき順番。この順序付けを有効にするためには、すべての CMR および CMP フィールドがこの属性で異なる数値を持つ必要がある。	省略可能

@ejbgen:create-default-rdbms-tables

使用場所 : クラス

適用対象 : エンティティ Bean

@ejbgen:ejb-client-jar

使用場所 : クラス

適用対象 : すべて Bean のタイプ

属性	説明	必須
file-name	生成するクライアント jar の名前。複数の EJB にこのタグがあった場合、指定された jar ファイルのうち 1 つだけがデプロイメント記述子に含められる。	必須

@ejbgen:ejb-local-ref

使用場所 : クラス

適用対象 : すべて Bean のタイプ

属性	説明	必須
home	Bean のローカル クラス	省略可能
jndi-name	参照の JNDI 名。	省略可能
link	Bean のリンク。	省略可能
local	Bean のホーム クラス。	省略可能
name	参照の名前。	省略可能
type	(Entity | Session)	省略可能

@ejbgen:ejb-ref

使用場所 : クラス

適用対象 : すべて Bean のタイプ

属性	説明	必須
home	Bean のリモート クラス。	省略可能
jndi-name	参照の JNDI 名。	省略可能
link	Bean のリンク。	省略可能
name	参照の名前。	省略可能
remote	Bean のホーム クラス。	省略可能
type	(Entity | Session)	省略可能

@ejbgen:entity

使用場所 : クラス

適用対象 : エンティティ Bean

属性	説明	必須
ejb-name	エンティティ Bean の名前。	必須
prim-key-class	NULL	必須
abstract-schema-name	この EJB の抽象スキーマ名。指定されない場合、ejb-name の値が使用される。	省略可能
concurrency-strategy	この Bean の同時方式を定義する (Optimistic | ReadOnly | Exclusive | Database)。	省略可能
data-source-name	データソース名 (config.xml で宣言したのと同じ)。	省略可能
db-is-shared	(True | False)	省略可能
default-transaction	詳細なトランザクション属性を設定しない、すべてのメソッドに適用されるトランザクション属性。	省略可能
delay-database-insert-until	(ejbCreate | ejbPostCreate)	省略可能
delay-updates-until-end-of-tx	トランザクション コミット後に更新が送信されるかどうか (True | False)。	省略可能
idle-timeout-seconds	ある EJB がキャッシュ内に置かれる最長の時間。	省略可能
invalidation-target	コンテナ管理による永続性 エンティティ EJB が変更された場合に無効とすべき、読み込み専用エンティティ Bean の ejb-name。	省略可能
max-beans-in-cache	キャッシュ内の Bean の最大数。	省略可能

属性	説明	必須
persistence-type	このエンティティ Bean のタイプ (cmp | bmp)。デフォルトは cmp。	省略可能
prim-key-class-nogen	(True | False)。このキーワードを指定した場合、EJBGen は (各自が提供するものと仮定して) 主キークラスを生成しなくなる。	省略可能
read-timeout-seconds	読み込み専用エンティティ Bean に ejbLoad() を呼び出す間隔の秒数。	省略可能
reentrant	(True | False)	省略可能
run-as	この EJB のロール名を指定する。	省略可能
run-as-identity-principal	ロールを複数プリンシパルにマップする場合のプリンシパル名を指定する。	省略可能
table-name	主キーの Java クラス。複合主キーの場合、このクラスは EJBGen によって生成される。	省略可能
trans-timeout-seconds	トランザクションのタイムアウト (秒数)。	省略可能
use-caller-identity	この EJB が呼び出し側の ID を使用するかどうか (True | False)。	省略可能

@ejbgen:env-entry

使用場所 : クラス

適用対象 : すべて Bean のタイプ

属性	説明	必須
name	この環境エントリの名前。	必須
type	この環境エントリの Java タイプ (java.lang であっても完全修飾しなければならない)。	必須

属性	説明	必須
value	この環境エントリの値。	必須

@ejbgen:finder

使用場所 : **クラス**
適用対象 : エンティティ Bean

属性	説明	必須
ejb-ql	デプロイメント記述子に含まれるものと同じ、EJB QL リクエスト。	必須
signature	ホーム クラスに生成したいシグネチャと正確に一致させなければならない。EJBGen は適合例外を追加するが、各パラメータについて、java.lang であっても完全修飾したタイプを必ず指定する。	必須
isolation-level	このメソッドのトランザクション アイソレーションのタイプ。	省略可能
transaction-attribute	このローカル メソッドのトランザクション属性。指定されない場合、デフォルトのトランザクション属性が使用される。このタグを付けたメソッドが、ローカル クラス上に生成される。	省略可能
weblogic-ejb-ql	デプロイメント記述子のものと同じ、Weblogic EJB QL リクエスト。注意 : このリスエストが必要になる場合、EJB QL と WebLogic EJBQL を二重引用符で囲む必要がある。	省略可能

@ejbgen:jndi-name

使用場所 : クラス

適用対象 : すべて Bean のタイプ

属性	説明	必須
local	この EJB のローカル JNDI 名。指定されない場合、いずれのローカル インタフェースも生成されない。	省略可能
remote	この EJB のリモート JNDI 名。指定されない場合、いずれのリモート インタフェースも生成されない。	省略可能

@ejbgen:local-home-method

使用場所 : メソッド

適用対象 : エンティティ Bean、セッション Bean

属性	説明	必須
transaction-attribute	このローカル メソッドのトランザクション属性。指定されない場合、デフォルトのトランザクション属性が使用される。このタグを付けたメソッドが、ローカル クラス上に生成される。	省略可能

@ejbgen:local-method

使用場所 : メソッド

適用対象 : エンティティ Bean、セッション Bean

属性	説明	必須
isolation-level	このメソッドのトランザクション アイソレーションのタイプ。	省略可能

属性	説明	必須
transaction-attribute	このローカル メソッドのトランザクション属性。指定されない場合、デフォルトのトランザクション属性が使用される。このタグを付けたメソッドが、ローカル クラス上に生成される。	省略可能

@ejbgen:message-driven

使用場所 : クラス

適用対象 : メッセージ駆動型 Bean

属性	説明	必須
destination-jndi-name	送り先の JNDI 名。	必須
ejb-name	このメッセージ駆動型 Bean の名前。	必須
acknowledge-mode	確認応答モード (auto-acknowledge | dups-ok-acknowledge)。	省略可能
default-transaction	詳細なトランザクション属性を設定しない、すべてのメソッドに適用されるトランザクション属性。	省略可能
destination-type	(javax.jms.Queue | javax.jms.Topic)	省略可能
durable	destination-type が Topic の場合にこの属性を設定すると、恒久サブスクリプションとなる (True | False)。	省略可能
initial-beans-in-free-pool	フリー プール内の Bean の初期数。	省略可能
max-beans-in-free-pool	フリー プール内の Bean の最大数。	省略可能
message-selector	JMS メッセージ セレクタ。	省略可能

属性	説明	必須
run-as	この EJB のロール名を指定する。	省略可能
run-as-identity-principal	ロールを複数プリンシパルにマップする場合のプリンシパル名を指定する。	省略可能
trans-timeout-seconds	トランザクションのタイムアウト (秒数)。	省略可能
use-caller-identity	この EJB が呼び出し側の ID を使用するかどうか (True | False)。	省略可能

@ejbgen:primkey-field

使用場所 : メソッド

適用対象 : エンティティ Bean

@ejbgen:relation

使用場所 : クラス

適用対象 : エンティティ Bean

属性	説明	必須
multiplicity	(one | many)	必須
name	関係の名前。ロールの関係の両端で同じ名前を使用するようにする (一方向でもこの制約が同様に適用される)。	必須
target-ejb	この関係のターゲット EJB 名。	必須
cascade-delete	(True | False)	省略可能

属性	説明	必須
cmr-field	この関係を格納する CMR フィールド。このフィールドは省略可能である。これを指定しない場合、関係は一方向となる。これを指定する場合には、fk-column 属性も合わせて指定する。	省略可能
fk-column	少なくとも 1 つの一方向関係のある場合のみ必要になる。一方向以外の EJB では、相手の主キーを保持するカラムを宣言しなければならない。	省略可能
joint-table	多対多関係のみで必要になる。この関係を含む結合テーブルを格納するのに使用する既存テーブルの名前にすること。複合主キーを使用する場合、これに関わる一連の外部キーをカンマで区切って指定する必要がある。	省略可能
role-name	このロールの名前 (たとえば、「ParentHasChildren」)。指定されない場合、EJBGen が代わりにこれを生成する。関係を継続的に使用しようとする場合には、ロール名を指定する必要がある。	省略可能

@ejbgen:remote-home-method

使用場所 : メソッド

適用対象 : エンティティ Bean、セッション Bean

属性	説明	必須
transaction-attribute	このリモート メソッドのトランザクション属性。指定されない場合、デフォルトのトランザクション属性が使用される。このタグを付けたメソッドが、リモート クラス上に生成される。	省略可能

@ejbgen:remote-method

使用場所 : メソッド

適用対象 : エンティティ Bean、セッション Bean

属性	説明	必須
isolation-level	このメソッドのトランザクション アイソレーションのタイプ。	省略可能
transaction-attribute	このリモート メソッドのトランザクション属性。指定されない場合、デフォルトのトランザクション属性が使用される。このタグを付けたメソッドが、リモート クラス上に生成される。	省略可能

@ejbgen:resource-env-ref

使用場所 : クラス

適用対象 : すべてのタイプの Bean

属性	説明	必須
name	リソース環境参照の名前。	必須
type	リソース環境参照のタイプ (javax.jms.Queue など)	必須
jndi-name	リソースの JNDI 名。	省略可能

@ejbgen:resource-ref

使用場所 : クラス

適用対象 : すべて Bean のタイプ

属性	説明	必須
auth	(Application | Container)	必須
jndi-name	リソースの JNDI 名。	必須
name	リソース名。	必須
type	リソースのタイプ (javax.sql.DataSource など)。	必須
sharing-scope	(Shareable | Unshareable)	省略可能

@ejbgen:role-mapping

使用場所 : クラス

適用対象 : すべて Bean のタイプ

属性	説明	必須
principals	このロールのプリンシパル名 (カンマで区切る)。	必須
role-name	ロール名。	必須

@ejbgen:select

使用場所 : メソッド

適用対象 : エンティティ Bean

属性	説明	必須
ejb-ql	この選択メソッドを定義する EJB-QL。注意 : メソッド名を「ejbSelect」から始めること。	必須
result-type-mapping	返されたオブジェクトを EJBLocalObject または EJBOject にマップするかどうか (Remote | Local)。	省略可能
weblogic-ejb-ql	デプロイメント記述子のものと同じ、Weblogic EJB QL リクエスト。注意 : このリスエストが必要になる場合、EJB QL と WebLogic EJBQL を二重引用符で囲む必要がある。	省略可能

@ejbgen:session

使用場所 : クラス

適用対象 : セッション Bean

属性	説明	必須
ejb-name	このセッション Bean の名前。	必須
call-router-class-name	ホーム メソッド呼び出しのルーティングに使用するクラス名。	省略可能
default-transaction	詳細なトランザクション属性を設定しない、すべてのメソッドに適用されるトランザクション属性。	省略可能
idle-timeout-seconds	ある EJB がキャッシュ内に置かれる最長の時間。	省略可能

属性	説明	必須
initial-beans-in-free-pool	フリー プール内の Bean の初期数。	省略可能
is-clusterable	この Bean がクラスタ対応かどうか (True | False)。	省略可能
load-algorithm	ホームのレプリカ間のバランシングを行うためのアルゴリズムの名前 (RoundRobin | Random | WeightBased)。	省略可能
max-beans-in-cache	キャッシュ内の Bean の最大数。	省略可能
max-beans-in-free-pool	フリー プール内の Bean の最大数。	省略可能
methods-are-idempotent	ステートレス セッション Bean のメソッドが多重呼び出し不変かどうか (True | False)。	省略可能
run-as	この EJB のロール名を指定する。	省略可能
run-as-identity-principal	ロールを複数プリンシパルにマップする場合のプリンシパル名を指定する。	省略可能
trans-timeout-seconds	トランザクションのタイムアウト (秒数)。	省略可能
type	セッション Bean のタイプ (Stateless | Stateful)。この属性が指定されない場合、EJBGen は各自のクラスの ejbCreate() メソッドを調べることによってこのタイプを正確に推測する。	省略可能
use-caller-identity	この EJB が呼び出し側の ID を使用するかどうか (True | False)。	省略可能

@ejbgen:value-object

使用場所 : クラス

適用対象 : すべて Bean のタイプ

属性	説明	必須
reference	他の EJB のクラスにアクセスする時に、Value オブジェクト クラスがどのオブジェクトを参照すべきかを指定する (Local | Value)。	必須

ejbc

EJB 2.0 および 1.1 のコンテナ クラスを生成およびコンパイルするには、`weblogic.ejbc` ユーティリティを使用します。EJB コンテナにデプロイするために JAR ファイルをコンパイルする場合は、`weblogic.ejbc` を使用して、コンテナ クラスを生成する必要があります。

`weblogic.ejbc` では、次の処理を実行します。

- 指定した JAR ファイルに EJB クラス、インタフェース、および XML デプロイメント記述子を配置する。
- すべての EJB クラスおよびインタフェースが EJB 仕様に準拠しているかどうかをチェックする。
- EJB 用の WebLogic Server コンテナ クラスを生成する。
- RMI コンパイラを使用して各 EJB コンテナ クラスを実行して、クライアントサイドの動的プロキシとサーバサイド バイト コードを作成する。

注意 : `ejbc` は、JAR ファイルおよび展開ディレクトリの両方を入力として受け付けます。

出力 JAR ファイルを指定すると、`ejbc` は、生成するファイルをすべて JAR ファイルに入れます。

ejbc は、デフォルトで javac をコンパイラとして使用します。パフォーマンスを向上させるには、-compiler フラグを使用して別のコンパイラ (Symantec の sj など) を指定します。

WebLogic 固有の XML デプロイメント記述子ファイルの複数バージョンが Web サイト上で公開され利用可能ですが、weblogic.ejbc で使用される内部バージョンは製品に同梱されています。

weblogic-ejb-jar.xml のパブリック バージョンの場所については、9-2 ページの「DOCTYPE ヘッダ情報」を参照してください。
weblogic-cmp-rdbms-jar.xml のパブリック バージョンの場所については、10-2 ページの「DOCTYPE ヘッダ情報」を参照してください。

ejbc の利点

ejbc ユーティリティの利点は以下のとおりです。

- ejbc のエラーを簡単に特定して修正できる。

  コマンド ラインで ejbc を実行中にエラーが発生すると、エラー メッセージが表示され、ejbc が終了します。

  それに対して、コンパイルをデプロイメント時に行うことにした場合、コンパイル エラーが発生すると、サーバはデプロイメントが失敗しても作業を続けます。デプロイメントの失敗の原因を特定するには、サーバ出力を調べ、問題を修正し、再デプロイする必要があります。

- デプロイメントの前に ejbc を実行すると、Bean のコンパイル時間を短縮できる。

  たとえば、.jar ファイルを 3 つのサーバのクラスタにデプロイする場合、.jar ファイルが各サーバにコピーされてデプロイされます。.jar ファイルがコンパイルされていない場合、各サーバは、デプロイメント時にファイルをコンパイルする必要があります。

ejbc の構文

```
prompt> java weblogic.ejbc [options] <source directory or jar file>
```

<target directory or jar file>

注意： 出力先が JAR ファイルの場合、出力 JAR には入力 JAR と異なる名前を付けなければなりません。

ejbc の引数

引数	説明
<source directory or jar file>	コンパイル済み EJB クラス、インタフェース、および XML デプロイメント ファイルを格納する展開ソース ディレクトリまたは JAR ファイルを指定する。
<target directory or jar file>	`ejbc` が出力 JAR を格納する送り先 JAR ファイルまたはデプロイメント ディレクトリを指定する。出力 JAR ファイルを指定した場合、`ejbc` は元の EJB クラス、インタフェース、および XML デプロイメント ファイルだけでなく、`ejbc` が生成する新規コンテナ クラスも JAR に格納する。

ejbc のオプション

オプション	説明
`-help`	コンパイラで使用可能なすべてのオプションのリストを出力する。
`-version`	`ejbc` のバージョン情報を出力する。
`-basicClientJar`	EJB 用に生成されたクライアント JAR のデプロイメント記述子を含まない。

-dispatchPolicy <queueName>	WebLogic Server で実行スレッドを取得するために EJB が使用するコンフィグレーション済み実行キューを指定する。詳細については、「実行キューによるスレッド使用の制御」を参照。
-forceGeneration	EJB クラスを強制的に生成する。このフラグを使用しない場合、クラスが再生成されないことがある (必要ないと判断された場合)。
-idl	リモート インタフェース用に CORBA インタフェース定義言語 (IDL) を生成する。
-idlNoValueTypes	値タイプ、およびそれを含むメソッドと属性を生成しない。
-idlFactories	値タイプ用にファクトリ メソッドを生成する。
-idlVisibroker	Visibroker 4.5 C++ と多少の互換性を持つ IDL を生成する。
-idlOrbix	Orbix 2000 2.0 C++ と多少の互換性を持つ IDL を生成する。
-idlOverwrite	既存の IDL ファイルを上書きする。
-idlVerbose	IDL の生成中に verbose 情報を表示する。
-idlDirectory <*dir*>	ejbc が IDL ファイルを生成するディレクトリを指定する。デフォルトでは、ejbc はカレント ディレクトリを使用する。
-idlMethodSignatures <>	IDL コードを生成するトリガとして使用されるメソッド シグネチャを指定する。
-iiop	EJB 用に CORBA のスタブを生成する。
-iiopDirectory <dir>	IIOP のスタブ ファイルを記述するディレクトリを指定する (デフォルトでは、対象ディレクトリ または JAR)。
-J	weblogic.ejbc のヒープ サイズを指定する。次のように指定する。 `java weblogic.ejbc -J-mx256m input.jar output.jar`

`-keepgenerated`	コンパイル中に生成される中間 Java ファイルを保存する。
`-compiler <compiler name>`	使用する `ejbc` のコンパイラを設定する。
`-normi`	RMI スタブの生成を中止する場合に Symantec の Java コンパイラ `sj` に渡される。それ以外の場合、`sj` は EJB には不必要な独自の RMI スタブを作成する。
`-classpath <path>`	コンパイル時に使用する CLASSPATH を設定する。これにより、システムまたはシェル CLASSPATH が付加される。
`-convertDD`	最新のデプロイメント記述子への更新を試みる。

ejbc の例

次の例では、`c:\%SAMPLES_HOME%\server\src\examples\ejb\basic\containerManaged\build` 内の入力 JAR ファイルに対して `javac` コンパイラを使用します。出力 JAR ファイルは、`c:\%SAMPLES_HOME%\server\config\examples\applications` 内に置かれます。

```
prompt> java weblogic.ejbc -compiler javac
c:\%SAMPLES_HOME%\server\samples\src\examples\ejb\basic\container
Managed\build\std_ejb_basic_containerManaged.jar
c:%SAMPLES_HOME%\server\config\examples\ejb_basic_containerManage
d.jar
```

次の例では、JAR ファイルが EJB 1.1 仕様に準拠しているかどうかをチェックして、WebLogic Server コンテナ クラスを生成しますが、RMI スタブは生成しません。

```
prompt> java weblogic.ejbc -normi
c:%SAMMPLES_HOME%\server\src\examples\ejb\basic\containerManaged\
build\std_ejb_basic_containerManaged.jar
```

DDConverter

DDConverter は、以前のバージョンの EJB デプロイメント記述子を本バージョンの WebLogic Server に準拠した EJB デプロイメント記述子に変換するコマンドライン ユーティリティです。WebLogic Server EJB コンテナは、EJB 1.1 および EJB 2.0 文書型定義 (DTD) を含む EJB 1.1 および EJB 2.0 仕様をサポートしています。各 WebLogic Server EJB デプロイメントには、以下のファイルの標準デプロイメント記述子が含まれています。

- ejb-jar.xml

  J2EE 固有の EJB デプロイメント記述子を含む XML ファイル。

- weblogic-ejb-jar-.xml

  WebLogic 固有の EJB デプロイメント記述子を含む XML ファイル。

- weblogic-cmp-rdbms-jar.xml

  WebLogic 固有のコンテナ管理による永続性 (CMP) デプロイメント記述子を含む XML ファイル。

DDConverter で利用できる変換オプション

DDConverter コマンドライン ユーティリティには、次の変換オプションがあります。

- 以前のバージョンの WebLogic Server の Bean を変換します (WLS)。
- 以前のバージョンの EJB 仕様の CMP および CMP 以外の Bean を変換します。

次の表は、DDconverter の各種変換オプションのリストです。

表 8-1

DDConverter ユーティリティの変換オプション

WLS		CMP 以外の EJB		EJB CMP	
変換前	**変換後**	**変換前**	**変換後**	**変換前**	**変換後**
WLS 4.5 - WLS 7.0		**注意 1** 参照		EJB CMP 1.0 - EJB CMP 1.1 **注意 2** 参照	
WLS 4.5 - WLS 7.0		EJB 1.1 - EJB 2.0		EJB CMP 1.0 - EJB CMP 2.0	
WLS 5.x - WLS 7.0		EJB 1.1 - EJB 2.0		**注意 3** 参照	
WLS 6.x - WLS 7.0		EJB 1.1 - EJB 2.0		EJB CMP 1.1 - EJB CMP 2.0	

注意： EJB 1.0 の非 CMP を EJB 1.1 の非 CMP に変換する必要はありません。非 CMP EJB 1.1 のデプロイメント記述子は、非 CMP EJB 2.0 のデプロイメント記述子と同じだからです。

注意： EJB CMP 1.0 を EJB CMP 1.1 に変換する際には、DDConverter のコマンドライン オプション `-EJBVer` を使用します。このオプションの内容については、8-29 ページの「DDConverter のオプション」を参照してください。

注意： WLS 5.x CMP 1.1 Bean と WLS 7.0 CMP 1.1 Bean は異なるものですが、WLS 5.1 CMP 1.1 Bean はソースコードを変更することなく WebLogic Server 7.0 上で動きます。

DDConverter を使用した後は必ず Bean を再コンパイルします。`weblogic.ejbc` を使用し、改めて生成した JAR ファイルをデプロイすることをお勧めします。Bean を再コンパイルすることにより、そのコードが確実に EJB 仕様に準拠するようになり、また、サーバ起動時の再コンパイル処理を省略できるので時間を短縮できます。

- WLS 4.5 EJB 1.0 Bean を WLS 7.0 EJB 1.1 Bean に変換する場合、`DDConverter` への入力は WebLogic 4.5 デプロイメント記述子のテキストです。WebLogic 7.0 デプロイメント記述子だけを含む JAR ファイルが出力さ

れます。weblogic-ejbc を起動してソースコードに何らかの変更を加える必要があるかどうかを確かめ、8-28 ページの「DDConverter による EJB の変換」の手順に従います。DDConverter ユーティリティの変換オプションの表の最初の行を参照してください。

- WLS 4.5 EJB 1.1 Bean を WLS 7.0 EJB 2.0 Bean に変換する場合、DDConverter への入力は WebLogic 4.5 デプロイメント記述子のテキストです。WebLogic 7.0 デプロイメント記述子だけを含む JAR ファイルが出力されます。weblogic-ejbc を起動してソースコードに何らかの変更を加える必要があるかどうかを確かめ、8-28 ページの「DDConverter による EJB の変換」の手順に従います。DDConverter ユーティリティの変換オプションの表の 2 行目を参照してください。
- WLS 7.0 の下位互換性により、ソースコードを変更することなく WLS 5.x EJB 1.1 Bean を WLS 7.0 にデプロイできます。WLS 7.0 は、以前のバージョンの WLS の Bean を検出し、再コンパイルし、デプロイします。しかし、DDConverter を使って WLS 5.x EJB 1.1 Bean を WLS 7.0 EJB 2.0 Bean にアップグレードすることをお勧めします。

  WLS 5.x EJB 1.1 Bean を WLS 7.0 EJB 2.0 Bean に変換する場合、DDConverter への入力は WebLogic 5.1 JAR ファイルです。このファイルには、デプロイメント記述子ファイルとクラス ファイルが含まれます。ここでは、WebLogic 7.0 デプロイメント記述子ファイルと必要なクラス ファイルすべてが含まれた 1 つの JAR ファイルに出力が格納されます。DDConverter ユーティリティの変換オプションの表の 3 行目を参照してください。

  ソースコードをほとんど、または、まったく変更せずに、CMP 以外の Bean を EJB 2.0 Bean に変換することができます。これを行うためには、output.jar ファイルに対し weblogic.ejbc を実行し、生成された JAR ファイルをデプロイします。CMP Bean の場合、8-28 ページの「DDConverter による EJB の変換」の手順に従い、ソースコードに変更を加える必要があります。

DDConverter による EJB の変換

WebLogic Server で使用するために以前のバージョンの EJB を変換するには、次の手順に従います。

1. 8-29 ページの「DDConverter の構文」に示したコマンド ライン形式に従って、デプロイメント記述子を `DDConverter` に入力します。

   JAR ファイルが出力されます。

2. その JAR ファイルから XML デプロイメント記述子を抽出します。
3. JavaSoft EJB 仕様に従って、ソースコードを変更します。
4. `weblogic.ejbc` を使用して JAR ファイルを作成し、抽出した XML デプロイメント記述子とともに変更済みの Java ファイルをコンパイルします。
5. JAR ファイルをデプロイします。

DDConverter の構文

```
prompt> java weblogic.ejb20.utils.DDConverter [options] file1
[file2...]
```

DDConverter の引数

`DDConverter` は、*`file1 [file2...]`* といった引数をとります。このファイルは次のどちらかになります。

- EJB 1.0 準拠のデプロイメント記述子を含むテキスト ファイル
- EJB 1.1 準拠のデプロイメント記述子を含む JAR ファイル

`DDConverter` は、テキスト デプロイメント記述子の EJB の `beanHomeName` プロパティを使用して、出力される `ejb-jar.xml` ファイルで新規の `ejb-name` 要素を定義します。

DDConverter のオプション

次の表は、DDConverter のコマンドライン オプションのリストです。

オプション	説明

`-d` destDir	JAR ファイルが出力される送り先ディレクトリを指定する。 このオプションは必須。
`-c` jar name	ソース ファイルのすべての Bean を組み合わせる JAR ファイルを指定する。
`-EJBVer` *`output EJB version`*	2.0 または 1.1 などの出力 EJB バージョン番号を指定する。デフォルトは 2.0。
`-log` log file	`ddconverter.log` の代わりにログ情報の格納先となるファイルを指定する。
`-verboseLog`	変換に関して `ddconverter.log` に格納する補足情報を指定する。
`-help`	`DDConverter` ユーティリティで使用可能なすべてのオプションのリストを出力する。

DDConverter の例

次の例では、WLS 5.x EJB 1.1 Bean を WLS 7.0 EJB 2.0 Bean に変換します。

サブディレクトリ *`destDir`* に JAR ファイルが作成されます。

```
prompt> java weblogic.ejb20.utils.DDConverter -d destDir
Employee.jar
```

ここでは、`Employee` Bean が WLS 5.x EJB 1.1 JAR ファイルです。

weblogic.Deployer

`weblogic.Deployer` コマンドライン ユーティリティは、非推奨となった以前の weblogic.deploy ユーティリティに代わり、WebLogic Server 7.0 で新たに追加されたものです。`weblogic.Deployer` ユーティリティは、WebLogic Server デプロイメント API にコマンドライン インタフェースを提供する、Java ベースの開発

ツールです。このユーティリティは、コマンドライン、シェル スクリプト、または、Java 以外の自動化されたあらゆる環境からデプロイメントを実行する管理者と開発者向けに開発されたものです。

weblogic.Deployer の使い方とコマンド リストについては、「WebLogic Server デプロイメント」を参照してください。

weblogic.deploy

注意： このツールはこのリリースでは使用されません。コマンドライン ツール、weblogic.Deployer を使用することを強くお勧めします。より洗練されたデプロイメント オプションが提供されています。

`weblogic.deploy` コマンドライン ユーティリティ ツールは、EJB をデプロイする際に使用します。EJB 準拠の JAR ファイルを指定すると、JAR の EJB は動作中の WebLogic Server にデプロイされます。

deploy の構文

```
prompt> java weblogic.deploy [options]
[list|deploy|undeploy|update] password {name} {source}
```

deploy の引数

引数	説明
`list`	指定した WebLogic Server のすべての EJB デプロイメント ユニットをリストする。
`deploy`	EJB JAR を指定したサーバにデプロイする。

undeploy	既存の EJB デプロイメント ユニットを指定したサーバから削除する。
update	EJB デプロイメント ユニットを指定したサーバに再デプロイする。
password	WebLogic Server のシステム パスワードを指定する。
{*name*}	EJB デプロイメント ユニットの名前を識別する。この名前は、deploy または console ユーティリティを使用して、デプロイメント時に指定する。
{*source*}	EJB JAR ファイルの絶対パス、または EJB デプロイメント ディレクトリの最上位までのパスを指定する。

deploy のオプション

オプション	説明
-help	deploy ユーティリティで使用可能なすべてのオプションのリストを出力する。
-version	ユーティリティのバージョンを出力する。
-port <port>	JAR ファイルをデプロイするために使用する WebLogic Server のポート番号を指定する。このオプションを指定しなかった場合、deploy ユーティリティはポート番号 7001 を使用して接続を試みる。
-host <host>	JAR ファイルをデプロイするために使用する WebLogic Server のホスト名を指定する。このオプションを指定しなかった場合、deploy ユーティリティはホスト名 localhost を使用して接続を試みる。

`-user`	JAR ファイルをデプロイするために使用する WebLogic Server のシステム ユーザ名を指定する。このオプションを指定しなかった場合、`deploy` は、システム ユーザ名 `system` を使用して接続を試みる。システム ユーザ名を定義するには、`weblogic.system.user` プロパティを使用する。
`-debug`	デプロイメント プロセス中に詳細なデバッグ情報を出力する。

9 weblogic-ejb-jar.xml 文書型定義

以下の節では、weblogic 固有の XML 文書型定義 (DTD) ファイル、`weblogic-ejb-jar.xml` ファイルの EJB 1.1 および EJB 2.0 デプロイメント記述子要素について説明します。これらの定義を使用して、EJB デプロイメントを構成する WebLogic 固有の `weblogic-ejb-jar.xml` ファイルを作成します。

- EJB デプロイメント記述子
- DOCTYPE ヘッダ情報
- 2.0 の weblogic-ejb-jar.xml デプロイメント記述子ファイルの構造
- 2.0 の weblogic-ejb-jar.xml デプロイメント記述子要素
- 5.1 の weblogic-ejb-jar.xml デプロイメント記述子ファイルの構造
- 5.1 の weblogic-ejb-jar.xml デプロイメント記述子ファイルの構造

EJB デプロイメント記述子

EJB デプロイメント記述子は、エンタープライズ Bean の構造およびアセンブリ情報を格納します。この情報を指定するには、3 つの EJB XML DTD ファイルのデプロイメント記述子に値を指定します。ファイルは次のとおりです。

- `ejb-jar.xml`
- `weblogic-ejb-jar.xml`
- `weblogic-cmp-rdbms-jar.xml`

この 3 つの XML ファイルを EJB および他のクラスと一緒にデプロイ可能なコンポーネント、通常は `ejb.jar` という JAR ファイルにパッケージ化します。

ejb-jar.xml ファイルは、Sun Microsystems の ejb.jar.xml ファイルのデプロイメント記述子に基づいています。その他の 2 つの XML ファイルは weblogic 固有のファイルで、weblogic-ejb-jar.xml と weblogic-cmp-rdbms-jar.xml のデプロイメント記述子に基づいています。

DOCTYPE ヘッダ情報

XML デプロイメント ファイルの編集、作成時に、デプロイメント ファイルに合わせて正しい DOCTYPE ヘッダを指定することが重要です。特に、DOCTYPE ヘッダ内部に不正な PUBLIC 要素を使用すると、原因究明が困難なパーサ エラーになることがあります。

WebLogic では、WebLogic 固有の DTD ファイル、weblogic-ejb-jar.xml の正確なテキストにアクセスできる場所を公開しています。一方で、この DTD ファイルと同一のバージョンが WebLogic Server に組み込まれ内部で使用されています。weblogic.ejbc では、XML パーサがデプロイメント記述子ファイルの並びをチェックする際にこれを使用します。

WebLogic Server 固有の weblogic-ejb-jar.xml ファイルの PUBLIC 要素には、次のようにテキストを指定する必要があります。

XML ファイル	PUBLIC 要素の文字列
weblogic-ejb-jar.xml	'-//BEA Systems, Inc.//DTD WebLogic 7.0.0 EJB//EN' 'http://www.bea.com/servers/wls700/dtd/weblogic-ejb-jar.dtd'
weblogic-ejb-jar.xml	'-//BEA Systems, Inc.//DTD WebLogic 6.0.0 EJB//EN' 'http://www.bea.com/servers/wls600/dtd/weblogic-ejb-jar.dtd'
weblogic-ejb-jar.xml	'-//BEA Systems, Inc.//DTD WebLogic 5.1.0 EJB//EN' 'http://www.bea.com/servers/wls510/dtd/weblogic-ejb-jar.dtd'

Sun Microsystems 固有の ejb-jar.xml ファイルの PUBLIC 要素には、次のようにテキストを指定する必要があります。

XML ファイル	PUBLIC 要素の文字列
ejb-jar.xml	'-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'
ejb-jar.xml	'-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN' 'http://www.java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'

たとえば、weblogic-ejb-jar.xml ファイルの DOCTYPE ヘッダは次のようになります。

```
<!DOCTYPE weblogic-ejb-jar PUBLIC
'-//BEA Systems, Inc.//DTD WebLogic 7.0.0 EJB//EN'
'http://www.bea.com/servers/wls700/dtd/weblogic-ejb-jar.dtd'>
```

XML の解析ユーティリティ (ejbc など) でヘッダ情報が不正な XML ファイルを解析すると、次のようなエラー メッセージが表示されることがあります。

```
SAXException: This document may not have the identifier 'identifier_name'
```

identifier_name には通常、PUBLIC 要素内の不正な文字列が表示されます。

検証用 DTD (Document Type Definitions : 文書型定義)

XML ファイルの内容および要素の配置は、使用する各ファイルの文書型定義 (DTD) に従っている必要があります。WebLogic Server では、XML デプロイメント ファイルの DOCTYPE ヘッダ内部に埋めこまれた DTD は無視され、代わりにサーバと一緒にインストールされた DTD の場所が使用されます。ただし、DOCTYPE ヘッダ情報には、パーサ エラーを避けるために、有効な URL 構文を指定する必要があります。

注意： ほとんどのブラウザでは、.dtd ファイルの内容は表示されません。DTD ファイルの内容をブラウザで見るには、リンクをテキスト ファイルとして保存し、テキスト エディタで開いて表示します。

weblogic-ejb-jar.xml

以下のリンクでは、WebLogic Server で使用される weblogic-ejb-jar.xml デプロイメント ファイル用のパブリック DTD の場所が示されています。

- weblogic-ejb-jar.xml 7.0 DTD の場合：

  http://www.bea.com/servers/wls700/dtd/weblogic-ejb-jar.dtd には、weblogic-ejb-jar.xml の作成に使用する DTD が含まれています。この DTD では、WebLogic Server へのデプロイメントに使用する EJB プロパティを定義します。

- weblogic-ejb-jar.xml 6.0 DTD の場合：

  http://www.bea.com/servers/wls600/dtd/weblogic-ejb-jar.dtd には、weblogic-ejb-jar.xml の作成に使用する DTD が含まれています。この DTD では、WebLogic Server へのデプロイメントに使用する EJB プロパティを定義します。

- weblogic-ejb-jar.xml 5.1 DTD の場合：

  weblogic-ejb-jar.dtd には、weblogic-ejb-jar.xml を作成する場合に使用される DTD が含まれています。この DTD では、WebLogic Server へのデプロイメントに使用する EJB プロパティを定義します。この DTD ファイルは、http://www.bea.com/servers/wls510/dtd/weblogic-ejb-jar.dtd にあります。

ejb-jar.xml

以下のリンクでは、WebLogic Server で使用される ejb-jar.xml デプロイメント ファイル用のパブリック DTD の場所が示されています。

- ejb-jar.xml 2.0 DTD の場合：

  http://www.java.sun.com/dtd/ejb-jar_2_0.dtd には、すべての EJB で必要な標準 ejb-jar.xml デプロイメント ファイル用の DTD が含まれています。この DTD は、JavaSoft EJB 2.0 仕様の一部として維持管理されていま

す。ejb-jar.dtd で使用される要素の詳細については JavaSoft 仕様を参照してください。

- ejb-jar.xml 1.1 DTD の場合：

  ejb-jar.dtd には、すべての EJB で必須の標準 ejb-jar.xml デプロイメント ファイル用の DTD が含まれています。この DTD は、JavaSoft EJB 1.1 仕様の一部として維持管理されています。ejb-jar.dtd で使用されている要素については、JavaSoft 仕様を参照してください。

注意： ejb-jar.xml デプロイメント記述子の説明については、該当する JavaSoft EJB 仕様を参照してください。

2.0 の weblogic-ejb-jar.xml デプロイメント記述子ファイルの構造

WebLogic Server 7.0 の weblogic-ejb-jar.xml デプロイメント記述子ファイルには、WebLogic Server 固有の要素を記述します。WebLogic Server 7.0 と WebLogic Server 6.0 の両バージョンのデプロイメント記述子を EJB コンテナで使用できますが、weblogic-ejb-jar.xml には、WebLogic Server バージョン 7.0 と WebLogic Server バージョン 6.0 の間で違いがあります。

WebLogic Server 7.0 の weblogic-ejb-jar.xml の最上位要素は次のとおりです。

- description
- weblogic-version
- weblogic-enterprise-bean
  - ejb-name
  - entity-descriptor | stateless-session-descriptor | stateful-session-descriptor | message-driven-descriptor
  - transaction-descriptor
  - reference-descriptor
  - enable-call-by-reference
  - clients-on-same-server

    - jndi-name
- security-role-assignment
- transaction-isolation

2.0 の weblogic-ejb-jar.xml デプロイメント記述子要素

- concurrency-strategy
- cache-between-transactions
- delay-updates-until-end-of-tx
- destination-jndi-name
- finders-load-bean
- home-call-router-class-name
- home-is-clusterable
- home-load-algorithm
- idle-timeout-seconds
- initial-beans-in-free-pool
- is-modified-method-name
- isolation-level
- jndi-name
- max-beans-in-cache
- max-beans-in-free-pool
- message-driven-descriptor
- method
- method-intf
- method-name
- method-param
- method-params
- persistence
- persistence-type
- persistence-use
- persistent-store-dir
- pool

- principal-name
- read-timeout-seconds
- reference-descriptor
- replication-type
- res-ref-name
- resource-description
- role-name
- security-role-assignment
- stateful-session-cache
- stateful-session-clustering
- stateful-session-descriptor
- stateless-bean-call-router-class-name
- stateless-bean-is-clusterable
- stateless-bean-load-algorithm
- stateless-bean-methods-are-idempotent
- stateless-clustering
- stateless-session-descriptor
- transaction-descriptor
- transaction-isolation
- trans-timeout-seconds
- type-identifier
- type-storage
- type-version
- 9-24 ページの「ejb-local-reference-description」
- 9-26 ページの「enable-dynamic-queries」
- 9-28 ページの「entity-cache-name」

- 9-29 ページの「entity-cache-ref」
- 9-33 ページの「externally-defined」
- 9-32 ページの「estimated-bean-size」
- 9-38 ページの「idempotent-methods」
- 9-39 ページの「identity-assertion」
- 9-41 ページの「iiop-security-descriptor」
- 9-43 ページの「initial-context-factory」
- 9-44 ページの「integrity」
- 9-44 ページの「invalidation-target」
- 9-47 ページの「jms-polling-interval-seconds」
- 9-48 ページの「jms-client-id」
- 9-50 ページの「local-jndi-name」
- 9-62 ページの「principal-name」
- 9-62 ページの「provider-url」
- 9-64 ページの「relationship-description」
- 9-66 ページの「res-env-ref-name」
- 9-67 ページの「resource-description」
-
- 9-69 ページの「security-permission」
- 9-70 ページの「security-permission-spec」
- 9-71 ページの「security-role-assignment」
- 9-72 ページの「session-timeout-seconds」
- 9-82 ページの「transport-requirements」
- 9-87 ページの「weblogic-ejb-jar」
- 9-87 ページの「weblogic-enterprise-bean」

allow-concurrent-calls

指定できる値：	`true	false`
デフォルト値：	`false`
要件：	ステートフル セッション Bean インスタンスによるメソッド呼び出しの処理中に、同時に他のメソッド呼び出しがサーバに送信されたときに、サーバが `RemoteException` を送出すること。
親要素：	`weblogic-enterprise-bean` `stateful-session-descriptor`

機能

`allow-concurrent-calls` 要素は、ステートフル セッション Bean インスタンスがメソッドの同時呼び出しを許可するかどうかを指定します。デフォルトでは、`allows-concurrent-calls` は `false` です。ただし `true` に設定すると、EJB コンテナはメソッドの同時呼び出しをブロックするため、前の呼び出しが完了してから次のメソッドが呼び出されるようになります。

例

9-75 ページの「stateful-session-descriptor」を参照してください。

allow-remove-during-transaction

指定できる値：	True | False
デフォルト値：	False
要件：	
親要素：	weblogic-enterprise-bean stateful-session-descriptor

機能

allow-remove-during-transaction 要素は、ステートフル セッション Bean の削除メソッドがトランザクション コンテキストで呼び出せることを指定します。

Synchronization インタフェースを実装するステートフル セッション Bean ではこのタグを使用しないで、トランザクションが終了する前に削除メソッドを呼び出す必要があります。このタグを使用すると、EJB コンテナは同期コールバックを呼び出しません。

例

9-75 ページの「stateful-session-descriptor」を参照してください。

cache-between-transactions

指定できる値：	`true	false`
デフォルト値：	`false`
要件：	省略可能。エンティティ EJB でのみ有効。
親要素：	`weblogic-enterprise-bean, entity-descriptor, persistence`

機能

`cache-between-transactions` 要素は、以前の `db-is-shared` 要素に相当し、EJB コンテナがあるエンティティの永続データをトランザクション間でキャッシュするかどうかを指定します。

`cache-between-transactions` 要素は、エンティティ Bean に対してのみ有効です。EJB コンテナによるデータの長期キャッシングの実行を有効にする場合、`True` を指定します。EJB コンテナによるデータの短期キャッシングの実行を有効にする場合、`False` を指定します。これはデフォルト設定です。

読み込み専用 Bean の場合、WebLogic Server はこれの長期キャッシングを常に実行するので、`cache-between-transactions` 要素の値は無視されます。

詳細については、4-27 ページの「トランザクション間のキャッシング」を参照してください。

例

9-58 ページの「persistence」を参照してください。

cache-type

指定できる値：	NRU	LRU
デフォルト値：	NRU
要件：	
親要素：	weblogic-enterprise-bean stateful-session-cache

機能

cache-type 要素は、EJB をキャッシュから削除する順序を指定します。値は次のとおりです。

- 最長時間未使用 (LRU : Least Recently Used)
- 最近未使用 (NRU : Not Recently Used)

  NRU の最小キャッシュ サイズは 8 です。max-beans-in-cache が 3 より小さい場合、WebLogic Server では cache-type の値に 8 が使用されます。

例

次の例では、cache-type 要素の構造を示します。

```
<stateful-session-cache>
<cache-type>NRU</cache-type>
</stateful-session-cache>
```

client-authentication

指定できる値：	none | supported | required
デフォルト値：	
要件：	なし
親要素：	weblogic-enterprise-bean iiop-security-descriptor

機能

client-authentication 要素は、EJB がクライアント認証をサポートするか、または、必要とするかを指定します。

例

9-41 ページの「iiop-security-descriptor」を参照してください。

client-cert-authentication

指定できる値：	none | supported | required
デフォルト値：	
要件：	なし
親要素：	weblogic-enterprise-bean iiop-security-descriptor transport-requirements

機能

client-cert-authentication 要素は、EJB が転送レベルでのクライアント証明書認証をサポートするか、または、必要とするかを指定します。

例

9-82 ページの「transport-requirements」を参照してください。

clients-on-same-server

指定できる値：	`true	false`
デフォルト値：	`false`
要件：	なし
親要素：	`weblogic-enterprise-bean`

機能

`clients-on-same-server` 要素によって、EJB がデプロイされた際、WebLogic Server がこの EJB に JNDI の通知を送るかどうかが指定されます。この属性が「false」(デフォルト) の場合、WebLogic Server クラスタでは、ある特定のサーバ上にあるこの EJB の場所を示すために、自動的に JNDI ツリーを更新します。これにより、そのクライアントが同じサーバ上で連結されてなくても、すべてのクライアントがこの EJB にアクセスできることが保証されます。

EJB にアクセスするクライアントのすべてが、その Bean がデプロイされているのと同じサーバ上からアクセスすることが分かっている場合には、`clients-on-same-server` を true に設定できます。この場合、EJB がデプロイされる際、WebLogic Server クラスタは JNDI の通知をこの EJB に送信しません。クラスタ内の JNDI の更新ではマルチキャスト トラフィックが利用されるため、`clients-on-same-server` を true に設定することにより大容量のクラスタでの起動時間を短縮できます。

連結された EJB の詳細については、『WebLogic Server クラスタ ユーザーズ ガイド』の「連結されたオブジェクトの最適化」を参照してください。

例

次の例では、EJB メソッドが値で渡すことができるようになります。

```
<weblogic-enterprise-bean>
        <ejb-name>AccountBean</ejb-name>
        ...
        <clients-on-same-server>true</clients-on-same-server>
</weblogic-enterprise-bean>
```

concurrency-strategy

指定できる値：	`Exclusive | Database | ReadOnly | Optimistic`
デフォルト値：	`Database`
要件：	省略可能。エンティティ EJB でのみ有効。

親要素：	`weblogic-enterprise-bean,`
	`entity-descriptor,`
	`entity-cache`

機能

`concurrency-strategy` 要素は、コンテナがエンティティ Bean への同時アクセスを管理する方法を指定します。以下の 4 つの値のいずれかに設定します。

- `Exclusive` を指定すると、Bean がトランザクションに関連付けられている場合、WebLogic Server はキャッシュされたエンティティ EJB インスタンスを排他的にロックする。EJB インスタンスに対するリクエストは、トランザクションが完了するまでブロックされます。このオプションは、WebLogic Server バージョン 3.1 ～ 5.1 までのデフォルトのロック動作です。
- `Database` を設定すると、WebLogic Server はエンティティ EJB に対するリクエストのロックを基底のデータストアに委ねる。`Database` 同時方式では、WebLogic Server は、独立したエンティティ Bean インスタンスを割り当て、ロックとキャッシングをデータベースが処理できるようにします。これは現在のデフォルト オプションです。
- `ReadOnly` は、読み込み専用エンティティ Bean で使用される。トランザクションごとに新たなインスタンスをアクティブ化するようにし、これによりリクエストは並列処理されます。WebLogic Server は、`read-timeout-seconds` パラメータを基に、`ReadOnly` の Bean に対して `ejbLoad()` を呼び出します。
- `Optimistic` では、トランザクションの間に EJB コンテナまたはデータベースでロックを行わない。EJB コンテナは、トランザクションをコミットする前に、あるトランザクションが更新したデータが変化していないことを確かめます。更新データのどれかが変わっていた場合、EJB コンテナはトランザクションをロールバックします。

Exclusive および Database のロック動作の詳細については、4-16 ページの「EJB の同時方式」を参照してください。read-only エンティティ EJB の詳細については、4-23 ページの「読み込み専用マルチキャストの無効化」を参照してください。

例

次の例では、AccountBean クラスを読み込み専用エンティティ EJB として指定します。

```
<weblogic-enterprise-bean>
       <ejb-name>AccountBean</ejb-name>
       <entity-descriptor>
              <entity-cache>

<concurrency-strategy>ReadOnly</concurrency-strategy>
              </entity-cache>
       </entity-descriptor>
</weblogic-enterprise-bean>
```

confidentiality

指定できる値:	none | supported | required
デフォルト値:	なし
要件:	なし
親要素:	weblogic-enterprise-bean iiop-security-descriptor transport-requirements n

機能

confidentiality 要素は、その EJB における転送の機密性の要件を指定します。confidentiality 要素を使用すると、他のエンティティに内容を見られることなく、クライアントとサーバ間でデータが送信されることを保証します。

例

9-82 ページの「transport-requirements」を参照してください。

connection-factory-jndi-name

指定できる値：	有効な名前
デフォルト値：	config.xml の weblogic.jms.MessageDrivenBeanConnectionFactory
要件：	ステートフル セッション Bean インスタンスによるメソッド呼び出しの処理中に、同時に他のメソッド呼び出しがサーバに送信されたときに、サーバが RemoteException を送出すること。
親要素：	weblogic-enterprise-bean message-driven-descriptor

機能

connection-factory-jndi-name 要素は、キューおよびトピックを作成するためにメッセージ駆動型 Bean がルックアップする JMS 接続ファクトリの JNDI 名を指定します。この要素を指定しなかった場合、config.xml の weblogic.jms.MessageDrivenBeanConnectionFactory がデフォルトになります。

例

次の例では、connection-factory-jndi-name 要素の構造を示します。

```
<message-driven-descriptor>
<connection-factory-jndi-name>weblogic.jms.MessageDrivenBean
ConnectionFactory</connection-factory-jndi-name>
</message-driven-descriptor>
```

delay-updates-until-end-of-tx

指定できる値：	`true	false`
デフォルト値：	`true`
要件：	省略可能。エンティティ EJB でのみ有効。
親要素：	`weblogic-enterprise-bean, entity-descriptor, persistence`

機能

トランザクションの完了時にトランザクションですべての Bean の永続ストレージを更新するには、`delay-updates-until-end-of-tx` 要素を `true` (デフォルト) に設定します。通常、この設定によって不要な更新を防ぐことができるため、パフォーマンスが向上します。ただし、データベース トランザクション内のデータベース更新の順序は保持されません。

データベースがアイソレーション レベルとして `TRANSACTION_READ_UNCOMMITTED` を使用している場合は、進行中のトランザクションに関する中間結果を他のデータベースのユーザに表示することもできます。この場合、`delay-updates-until-end-of-tx` を `false` に設定して、各メソッド呼び出しの完了時に Bean の永続ストレージを更新するように指定します。詳細については、4-13 ページの「エンティティ EJB に対する ejbLoad() と ejbStore() の動作」を参照してください。

注意： `delay-updates-until-end-of-tx` を false に設定しても、各メソッド呼び出しの後にデータベース更新が「コミットされた」状態になるわけではありません。更新はデータベースに送信されるだけです。更新は、トランザクションの完了時にのみデータベースにコミットまたはロールバックされます。

例

次の例では、`delay-updates-until-end-of-tx` スタンザを示します。

```
<entity-descriptor>
       <persistence>
```

```
<delay-updates-until-end-of-tx>false</delay-updates-until-end-of-
tx>
       </persistence>
</entity-descriptor>
```

description

指定できる値：	なし
デフォルト値：	なし
要件：	なし
親要素：	weblogic-enterprise-bean, transaction-isolation method

機能

description 要素は、親要素を示すテキストの指定に使用します。

例

次の例では、description 要素を指定します。

```
<description>Contains a description of parent element</description>
```

destination-jndi-name

指定できる値：	有効な JNDI 名
デフォルト値：	なし
要件：	`message-driven-descriptor` で必須。
親要素：	`weblogic-enterprise-bean` `message-driven-descriptor`

機能

`destination-jndi-name` 要素は、WebLogic Server JNDI ツリーにデプロイされている実際の JMS キューまたはトピックにメッセージ駆動型 Bean を関連付けるために使用する JNDI 名を指定します。

例

9-53 ページの「message-driven-descriptor」を参照してください。

ejb-name

指定できる値：	`ejb-jar.xml` で定義した EJB 名
デフォルト値：	なし
要件：	method スタンザで必須。この名前は、NMTOKEN の命名規則に準拠しなければならない。
親要素：	`weblogic-enterprise-bean` `method`

機能

`ejb-name` は、WebLogic Server がアイソレーション レベルのプロパティに適用する EJB の名前を指定します。この名前は、`ejb-jar` ファイルのデプロイメント記述子で割り当てます。名前は、同じ `ejb.jar` ファイル内のエンタープライズ Bean の名前の中でユニークでなければなりません。エンタープライズ Bean

のコードは名前に左右されないので、アプリケーション アセンブリ処理中に名前を変更してもエンタープライズ Bean の機能には影響しません。デプロイメント記述子の ejb-name と、デプロイヤがエンタープライズ Bean のホームに割り当てる JNDI 名との間には、組み込みの関係はありません。

例

9-54 ページの「method」を参照してください。

ejb-reference-description

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	省略可能。
親要素 :	weblogic-enterprise-bean reference-descriptor

機能

ejb-reference-description 要素は、Bean によって参照される、WebLogic Server における EJB の JNDI 名を ejb-reference 要素にマップします。

- ejb-ref-name ではリソース参照名を指定する。これは、EJB プロバイダによって ejb-jar.xml デプロイメント記述子ファイル内に記載される参照です。
- jndi-name では、WebLogic Server で使用可能な実際のリソース ファクトリの JNDI 名を指定する。

例

ejb-reference-description スタンザを次に示します。

```
<ejb-reference-description>
       <ejb-ref-name>AdminBean</ejb-ref-name>
              <jndi-name>payroll.AdminBean</jndi-name>
       </ejb-reference-description>
```

ejb-ref-name

指定できる値:	なし
デフォルト値:	なし
要件:	省略可能。
親要素:	weblogic-enterprise-bean reference-description ejb-reference-description

機能

ejb-ref-name 要素はリソース参照名を指定します。この要素は、EJB プロバイダが ejb-jar.xml デプロイメント ファイル内に配置する参照です。

例

ejb-ref-name スタンザを次に示します。

```
<reference-descriptor>
       <ejb-reference-description>
              <ejb-ref-name>AdminBean</ejb-ref-name>
              <jndi-name>payroll.AdminBean</jndi-name>
       </ejb-reference-description>
</reference-descriptor>
```

ejb-local-reference-description

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	省略可能。
親要素 :	`weblogic-enterprise-bean` `reference-descriptor`

機能

`ejb-local-reference-description` 要素は、Bean が `ejb-local ref` で参照する WebLogic Server の EJB の JNDI 名をマップします。

例

次の例では、`ejb-local-reference-description` 要素を示します。

```
<ejb-local-reference-description>
       <ejb-ref-name>AdminBean</ejb-ref-name>
       <jndi-name>payroll.AdminBean</jndi-name>
</ejb-local-reference-description>
```

enable-call-by-reference

指定できる値：	`true	false`
デフォルト値：	`true`
要件：	省略可能。

親要素：	`weblogic-enterprise-bean`
	`reference-descriptor`
	`ejb-reference-description`

機能

デフォルトでは、同じアプリケーション (EAR) から呼び出された EJB メソッドは引数を参照で渡します。パラメータはコピーされないので、これによってメソッド呼び出しのパフォーマンスが向上します。

`enable-call-by-reference` を `false` に設定すると、EJBE 1.1 の仕様に従って EJB メソッドへのパラメータがコピーされ (値で渡され) ます。EJB がリモートで (同じアプリケーション以外から) 呼び出される場合は、常に値で渡す必要があります。

例

次の例では、EJB メソッドが値で渡すことができるようになります。

```
<weblogic-enterprise-bean>
       <ejb-name>AccountBean</ejb-name>
       ...
       <enable-call-by-reference>false</enable-call-by-reference>
</weblogic-enterprise-bean>
```

enable-dynamic-queries

指定できる値：	`true | false`
デフォルト値：	`true`
要件：	省略可能。
親要素：	`weblogic-enterprise-bean` `entity-descriptor`

機能

動的クエリを有効にするには、任意指定の enable-dynamic-queries 要素を true に設定する必要があります。動的クエリは、EJB 2.0 CMP Bean を使用しているときのみ使えます。

例

次の例では、動的クエリを有効にしています。

```
<enable-dynamic-queries>True</enable-dynamic-queries>
```

entity-cache

指定できる値：	なし (XML スタンザ)
デフォルト値：	なし (XML スタンザ)
要件：	`entity-cache` スタンザは省略可能。エンティティ EJB でのみ有効。
親要素：	`weblogic-enterprise-bean`, `entity-descriptor`

機能

`entity-cache` 要素は、WebLogic Server 内のエンティティ EJB インスタンスのキャッシュに使用する以下のオプションを定義します。

- `max-beans-in-cache`

- idle-timeout-seconds
- read-timeout-seconds
- concurrency-strategy

WebLogic Server で使用可能なキャッシュ サービスについては、4-2 ページの「EJB のライフサイクル」を参照してください。

例

entity-cache スタンザを次に示します。

```
<entity-descriptor>
       <entity-cache>
              <max-beans-in-cache>...</max-beans-in-cache>
              <idle-timeout-seconds>...</idle-timeout-seconds>
              <read-timeout-seconds>...<read-timeout-seconds>
              <concurrency-strategy>...</concurrency-strategy>
       </entity-cache>
       <persistence>...</persistence>
       <entity-clustering>...</entity-clustering>
</entity-descriptor>
```

entity-cache-name

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	entity-cache-name で指定する値は、weblogic-application.xml ファイルでアプリケーション レベルのエンティティ キャッシュに当てた名前と一致させること。
親要素 :	weblogic-enterprise-bean, entity-descriptor entity-cache-ref

機能

entity-cache-name 要素は、エンティティ Bean が使用する、アプリケーション レベルのエンティティ キャッシュを参照します。アプリケーション レベルのキャッシュは、同じアプリケーション内の複数エンティティ Bean によって共有されるキャッシュです。

weblogic-application.xml ファイルの詳細については、「アプリケーション デプロイメント記述子の要素」を参照してください。

例

9-29 ページの「entity-cache-ref」を参照してください。

entity-cache-ref

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	`entity-cache-ref` スタンザ内の `entity-cache-name` 要素が、アプリケーション レベルのキャッシュの名前を含むこと。
親要素 :	`weblogic-enterprise-bean`, `entity-descriptor`

機能

`entity-cache-ref` 要素は、同じアプリケーションの一部である複数エンティティ Bean のインスタンスをキャッシュすることのできる、アプリケーション レベルのエンティティ キャッシュを参照します。アプリケーション レベルのエンティティ キャッシュは、`weblogic-application.xml` ファイルで宣言されます。

9-16 ページの「concurrency-strategy」を使用し、その Bean に使用させたい同時方式のタイプを定義します。同時方式はアプリケーション レベル キャッシュのキャッシュ戦略と適合していなくてはなりません。たとえば、排他的キャッシュは、`Exclusive` 同時方式である Bean だけをサポートします。一方、マルチバージョン キャッシュは、Database、ReadOnly、および Optimistic 同時方式をサポートします。

例

`entity-cache-ref` スタンザを次に示します。

```
<entity-cache-ref>
       <entity-cache-name>AllEntityCache</entity-cache-name>
       <concurrency-strategy>ReadOnly</concurrency-strategy>
       <estimated-bean-size>20</estimated-bean-size>
</entity-cache-ref>
```

entity-clustering

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	省略可能。クラスタ内のエンティティ EJB でのみ有効。
親要素 :	`weblogic-enterprise-bean`, `entity-descriptor`

機能

`entity-clustering` 要素は、以下のオプションを使用して、エンティティ Bean を WebLogic クラスタ内でレプリケートする方法をしています。

- `home-is-clusterable`
- `home-load-algorithm`
- `home call-router-class-name`

例

次の例では、`stateful-session-clustering` スタンザの構造を示します。

```
<entity-clustering>
       <home-is-clusterable>true</home-is-clusterable>
       <home-load-algorithm>random</home-load-algorithm>

<home-call-router-class-name>beanRouter</home-call-router-class-name>
</entity-clustering>
```

entity-descriptor

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	*.jar* 内のエンティティ EJB ごとに 1 つの `entity-descriptor` スタンザが必須。
親要素 :	`weblogic-enterprise-bean`

機能

`entity-descriptor` 要素は、エンティティ Bean に適用する以下のデプロイメント パラメータを指定します。

- `pool`
- `entity-cache`
- `persistence`
- `entity-clustering`

例

次の例では、`entity-descriptor` スタンザの構造を示します。

```
<entity-descriptor>
       <entity-cache>...</entity-cache>
       <persistence>...</persistence>
       <entity-clustering>...</entity-clustering>
</entity-descriptor>
```

estimated-bean-size

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	なし
親要素 :	`weblogic-enterprise-bean` `entity-descriptor`

機能

`estimated-bean-size` 要素は、エンティティ Bean のインスタンスについて評価された平均サイズをバイト数で指定します。各インスタンスが消費するメモリのバイト数の平均値です。

`estimated-bean-size` 要素は、Bean をキャッシュするのに使用する、アプリケーション レベルのキャシュがともにバイト数やメガバイト単位で指定されている場合に合わせて使用します。

エンティティ Bean インスタンスが消費する正確なバイト数が分からなくても、サイズを指定することにより、ある一時点でキャッシュを共有する Bean に対し相対的な重みを与えることができます。

たとえば、Bean A と Bean B が 1000 バイトのキャッシュを共有し、A のサイズが 10 バイト、B のサイズが 20 バイトであるとします。その場合、キャッシュに格納できるのは、最大で、A のインスタンスで 100、B のインスタンスで 50 です。A のインスタンスが 100 個キャッシュされた場合、B のインスタンスは 0 個、キャッシュされたことになります。

例

9-29 ページの「entity-cache-ref」を参照してください。

externally-defined

指定できる値：	`True | False`
デフォルト値：	
要件：	なし
親要素：	`weblogic-enterprise-bean` `security-role-assignment`

機能

`externally-defined` 要素は、特定のセキュリティ ロールがデプロイメント記述子ではなくセキュリティ レルムで定義されていることを示します。セキュリティロールと関連するプリンシパル名のマッピングは別の場所で定義されているので、プリンシパル名はデプロイメント記述子には指定されていません。このタグは、`principal-name` 要素の代わりに、プレースホルダとして使用されます。

finders-load-bean

指定できる値：	`true | false`
デフォルト値：	`true`
要件：	省略可能。CMP エンティティ EJB でのみ有効。

親要素：	`weblogic-enterprise-bean,`
	`entity-descriptor,`
	`persistence`

機能

`finders-load-bean` は、ファインダ メソッドの呼び出しによって Bean への参照が返されてから WebLogic Server が EJB をキャッシュにロードするかどうかを指定します。この要素を `true` に設定した場合、Bean への参照がファインダによって返されると、WebLogic Server はすぐにその Bean をキャッシュにロード

します。この要素を false に設定した場合、WebLogic Server は、最初のメソッドが呼び出されるまで Bean を自動的にロードしません。この動作は、EJB 1.1 の仕様と一致しています。

例

次のエントリでは、ファインダ メソッドによって Bean への参照が返されたら、EJB が自動的に WebLogic Server キャッシュにロードされるように指定します。

```
<entity-descriptor>
       <persistence>
              <finders-load-bean>true</finders-load-bean>
       </persistence>
</entity-descriptor>
```

global-role

指定できる値：	True | False
デフォルト値：	True
要件：	
親要素：	weblogic-enterprise-bean security-role-assignment

機能

global-role 要素は、特定のセキュリティ ロールがセキュリティ レルムでグローバルに定義されていることを示します。セキュリティロールと関連するプリンシパル名のマッピングは別の場所で定義されているので、プリンシパル名はデプロイメント記述子には指定されていません。このタグは、principal-name 要素の代わりに、プレースホルダとして使用されます。

home-call-router-class-name

指定できる値：	有効なルータ クラス名
デフォルト値：	null
要件：	省略可能。クラスタ内のエンティティ EJB、ステートフル セッション EJB、およびステートレス セッション EJB でのみ有効。

親要素：	`weblogic-enterprise-bean,`
	`  entity-descriptor,`
	`  entity-clustering`
	および
	`weblogic-enterprise-bean`
	`  stateful-session-descriptor`
	`  stateful-session-clustering`

機能

`home-call-router-class-name` は、Bean メソッド呼び出しのルーティングに使用するカスタム クラスの名前を指定します。このクラスは `weblogic.rmi.cluster.CallRouter()` を実装する必要があります。指定すると、このクラスのインスタンスは各メソッド呼び出しの前に呼び出されます。ルータ クラスでは、メソッドのパラメータを基に、ルーティングするサーバを選択できます。このクラスは、サーバ名を返すか、または現在のロード アルゴリズムがサーバを選択する必要があることを示す null を返します。

例

9-30 ページの「entity-clustering」と 9-74 ページの「stateful-session-clustering」を参照してください。

home-is-clusterable

指定できる値：	`true	false`
デフォルト値：	`true`
要件：	省略可能。クラスタ内のエンティティ EJB、ステートレス セッション EJB、およびステートフル セッション EJB でのみ有効。

親要素：	`weblogic-enterprise-bean, entity-descriptor, entity-clustering`
	および
	`weblogic-enterprise-bean stateful-session-descriptor stateful-session-clustering`
	および
	`weblogic-enterprise-bean stateless-session-descriptor stateless-session-clustering`

機能

`home-is-clusterable` は、エンティティ Bean、ステートレス セッション Bean、またはステートフル セッション Bean のホーム インタフェースがクラスタ化されているかどうかを示す場合に使用します。

クラスタにデプロイされている EJB に関して `home-is-clusterable` が `true` の場合、各サーバ インスタンスは Bean のホーム インタフェースをクラスタの JNDI ツリーに同じ名前でバインドします。クライアントがクラスタから Bean のホーム インタフェースをリクエストすると、ルックアップを実行するサーバ インスタンスは、各サーバのホームへの参照を持つ `EJBHome` スタブを返します。

クライアントが `create()` または `find()` 呼び出しを発行すると、スタブは、ロード バランシング アルゴリズムに従ってレプリカ リストからサーバを選択し、そのサーバのホーム インタフェースに呼び出しをルーティングします。選択されたホーム インタフェースは呼び出しを受け取り、そのサーバ インスタンス上に Bean インスタンスを作成し、呼び出しを実行します。

例

9-30 ページの「entity-clustering」を参照してください。

home-load-algorithm

指定できる値:	`round-robin | random | weight-based`
デフォルト値:	`weblogic.cluster.defaultLoadAlgorithm` の値
要件:	省略可能。クラスタ内のエンティティ EJB およびステートフル セッション EJB でのみ有効。

親要素:	`weblogic-enterprise-bean`
	`  entity-descriptor`
	`  entity-clustering`
	および
	`weblogic-enterprise-bean`
	`  stateful-session-descriptor`
	`  stateful-session-clustering`

機能

`home-load-algorithm` では、EJB ホームのレプリカ間でのロード バランシングに使用するアルゴリズムを指定します。この要素を定義しない場合、WebLogic Server はサーバ要素、`weblogic.cluster.defaultLoadAlgorithm` で指定されたアルゴリズムを使用します。

`home-load-algorithm` は以下のいずれかの値に定義できます。

- `round-robin` :Bean のホスト サーバ間で順番にロード バランシングを実行します。
- `random` :Bean のホスト サーバ間で EJB ホームのレプリカがランダムにデプロイされます。
- `weight-based` : ホスト サーバの現在の負荷に従って、EJB ホームのレプリカをデプロイするサーバが決まります。

例

9-30 ページの「entity-clustering」と 9-74 ページの「stateful-session-clustering」を参照してください。

idempotent-methods

指定できる値：	なし
デフォルト値：	なし
要件：	EJB でクラスタ化が有効であること。
親要素：	`weblogic-enterprise-bean`

機能

`idempotent-method` 要素は、同じメソッドを同じ引数で繰り返し呼び出したとしても、1 回呼び出したときとまったく同じ結果になるように記述される、メソッドのリストを定義します。これにより、フェールオーバ ハンドラは、失敗したサーバ上で実際に呼び出しがコンパイルされるかどうかを知らなくても、失敗した呼び出しを再試行できるようになります。あるメソッドに対し多重呼び出し不変メソッドを有効にした場合、EJB スタブは、EJB のホスト サーバに接続できるかぎり、あらゆる失敗から自動的に回復することができます。

クラスタ化を有効にする手順については、9-30 ページの「entity-clustering」、9-74 ページの「stateful-session-clustering」、および 9-79 ページの「stateless-clustering」を参照してください。

ステートレス セッション Bean ホームおよび読み込み専用 エンティティ Bean のメソッドは、自動的に多重呼び出し不変に設定されます。これらについては多重呼び出し不変を明示的に設定する必要はありません。

例

`method` スタンザには、以下の要素を指定できます。

```
<idempotent-method>
       <method>
              <description>...</description>
              <ejb-name>...</ejb-name>
              <method-intf>...</method-intf>
```

```
                <method-name>...</method-name>
                <method-params>...</method-params>
            </method>
    </idempotent-method>
```

identity-assertion

指定できる値：	`none	supported	required`
デフォルト値：	
要件：	なし
親要素：	`weblogic-enterprise-bean` `iiop-security-descriptor`

機能

`identity-assertion` 要素は、EJB が ID アサーションをサポートするか、あるいは、必要とするかどうかを指定します。

例

9-41 ページの「iiop-security-descriptor」を参照してください。

idle-timeout-seconds

指定できる値：	1 から *maxSeconds* で指定した値。*maxSeconds* は整数の最大値。
デフォルト値：	600
要件：	省略可能
親要素：	weblogic-enterprise-bean, entity-descriptor, entity-cache および weblogic-enterprise-bean, stateful-session-descriptor, stateful-session-cache

機能

idle-timeout-seconds では、ステートフル EJB がキャッシュに保持される最長時間を定義します。この時間を過ぎると、キャッシュ内の Bean の数が max-beans-in-cache で指定した値を超えている場合、WebLogic Server では Bean インスタンスが削除されます。削除された Bean インスタンスに対してはパッシベーションが行われます。詳細については、4-2 ページの「EJB のライフサイクル」を参照してください。

注意： idle-timeout-seconds は entity-cache スタンザで使用しますが、WebLogic Server 7. 0 SP01、SP02、SP03、および SP04 ではエンティティ EJB のライフサイクルの管理でその値を使用しません。それらのサービス パックでは、エンティティ Bean がいつキャッシュから削除されるかについて idle-timeout-seconds の設定は影響しません。

例

次のエントリでは、max-beans-in-cache の値に達し、Bean がキャッシュに 20 分保持されている場合、ステートフル セッション EJB AccountBean が削除の対象になります。

```
<weblogic-enterprise-bean>
       <ejb-name>AccountBean</ejb-name>
       <stateful-session-descriptor>
```

```
                <stateful_session-cache>
                        <max-beans-in-cache>200</max-beans-in-cache>
<idle-timeout-seconds>1200</idle-timeout-seconds>
                </stateful-session-cache>
        </stateful-session-descriptor>
</weblogic-enterprise-bean>
```

iiop-security-descriptor

指定できる値:	なし
デフォルト値:	なし
要件:	なし
親要素:	weblogic-enterprise-bean

機能

iiop-security-descriptor 要素は、Bean レベルのセキュリティ コンフィグレーション パラメータを指定します。これらのパラメータにより、IOR に含まれる IIOP セキュリティ情報が決定します。

例

iiop-security-descriptor スタンザには、以下の要素を指定できます。

```
<iiop-security-descriptor>
        <transport-requirements>...</transport-requirements>
        <client-authentication>supported<client-authentication>
        <identity-assertion>supported</identity-assertion>
</iiop-security-descriptor>
```

initial-beans-in-free-pool

指定できる値：	0 から *maxBeans* で指定した値
デフォルト値：	0
要件：	省略可能。ステートレス セッション、エンティティ、およびメッセージ駆動型 EJB で有効。
親要素：	weblogic-enterprise-bean, stateless-session-descriptor, message-bean-descriptor, entity-descriptor pool

機能

initial-bean-in-free-pool の値を指定すると、WebLogic Server では、起動時に、すべての Bean クラスの指定した数の Bean インスタンスがフリー プールに生成されます。この方法で Bean インスタンスをフリー プールに格納しておくと、要求が来てから新しいインスタンスを生成せずに Bean に対する初期要求が可能になるため、EJB の初期応答時間が短縮されます。

例

9-61 ページの「pool」を参照してください。

initial-context-factory

指定できる値：	`true | false`
デフォルト値：	`weblogic.jndi.WLInitialContextFactory`
要件：	ステートフル セッション Bean インスタンスによるメソッド呼び出しの処理中に、同時に他のメソッド呼び出しがサーバに送信されたときに、サーバが `RemoteException` を送出すること。
親要素：	`weblogic-enterprise-bean` `message-driven-descriptor`

機能

`initial-context-factory` 要素は、コンテナが接続ファクトリを作成するために使用する初期 contextFactory を指定します。initial-context-factory を指定しなかった場合、デフォルトは `weblogic.jndi.WLInitialContextFactory` になります。

例

次の例では、`initial-context-factory` 要素の構造を示します。

```
<message-driven-descriptor>
<initial-context-factory>weblogic.jndi.WLInitialContextFactory
</initial-context-factory>
</message-driven-descriptor>
```

integrity

指定できる値：	`none	supported	required`
デフォルト値：	
要件：	なし

親要素：	`weblogic-enterprise-bean`
	`iiop-security-descriptor`
	`transport-requirements`

機能

`integrity` 要素は、その EJB の転送の整合性の要件を指定します。integrity 要素を使用すると、クライアントとサーバ間で、データが途中で変化することなく転送されることが保証されます。

例

9-82 ページの「transport-requirements」を参照してください。

invalidation-target

指定できる値：	
デフォルト値：	
要件：	対象の `ejb-name` は読み込み専用エンティティ EJB でなければならないので、この要素は EJB 2.0 のコンテナ管理による永続性エンティティ EJB に対してのみ指定できる。
親要素：	`weblogic-enterprise-bean` `entity-descriptor`

機能

invalidation-target 要素は、コンテナ管理による永続性エンティティ EJB が変更された場合に、無効化する読み込み専用エンティティ EJB を指定します。

例

次の例では、EJB が変更されると StockReaderEJB という EJB が無効化されるように指定します。

```
<invalidation-target>
       <ejb-name>StockReaderEJB</ejb-name>
</invalidation-target>
```

is-modified-method-name

指定できる値：	有効なエンティティ EJB メソッド名
デフォルト値：	なし
要件：	省略可能。エンティティ EJB でのみ有効。
親要素：	weblogic-enterprise-bean, entity-descriptor, persistence

機能

is-modified-method-name では、EJB の保存時に WebLogic Server によって呼び出されるメソッドを指定します。指定したメソッドはブール値を返す必要があります。メソッドを指定しない場合、WebLogic Server では、EJB は常に変更されていると見なされて保存されます。

メソッドを指定して適切な値を設定すると、EJB 1.1 準拠の Bean、および Bean 管理の永続性を使用する Bean のパフォーマンスが向上します。ただし、メソッドの戻り値にエラーがあると、データに矛盾が発生する場合があります。詳細については、4-14 ページの「is-modified-method-name を使用した ejbStore() の呼び出しの制限 (EJB 1.1 のみ)」を参照してください。

注意： isModified() は、EJB 2.0 仕様に基づく 2.0 CMP エンティティ EJB では不要ですが、BMP および 1.1 CMP の EJB では有効です。コンテナ管理の永続性を使用して EJB 2.0 エンティティ Bean をデプロイする場合、WebLogic Server によって、変更されている EJB フィールドが自動的に検出され、そのフィールドのみが基底のデータストアに書き込まれます。

例

次の例では、EJB が変更されると semidivine という EJB メソッドが WebLogic Server に通知するように指定します。

```
<entity-descriptor>
       <persistence>

<is-modified-method-name>semidivine</is-modified-method-name>
       </persistence>
</entity-descriptor>
```

isolation-level

指定できる値:	TransactionSerializable | TransactionReadCommitted | TransactionReadUncommitted | TransactionRepeatableRead | TransactionReadCommittedForUpdate | TransactionReadCommittedForUpdateNoWait
デフォルト値:	なし
要件:	省略可能。
親要素:	weblogic-enterprise-bean, transaction-isolation

機能

isolation-level は、すべての EJB データベース処理に対するアイソレーション レベルを指定します。isolation-level には以下の値を指定できます。

- TransactionReadCommittedUncommitted: トランザクションはコミットしていない他のトランザクションの更新を参照できます。
- TransactionReadCommitted: トランザクションはコミットされた他のトランザクションの更新だけを参照できます。
- TransactionRepeatableRead: トランザクションでデータの一部を読み取ると、そのデータが他のトランザクションで変更されても、最初の読み取り時と同じ値が返されます。

- `TransactionSerializable`：このトランザクションを同時に複数回実行すると、トランザクションを順番に複数回実行することと同じことになります。

Oracle データベースの場合のみ：

- `TransactionReadCommittedForUpdate`
- `TransactionReadCommittedForUpdateNoWait`

  Oracle 固有の `isolation-level` 値の背景情報については、4-45 ページの「Oracle データベースに関する特別な注意」を参照してください。

異なるアイソレーション レベルの関係と各アイソレーション レベルのサポートの詳細については、各データベースのマニュアルを参照してください。

例

9-81 ページの「transaction-isolation」を参照してください。

jms-polling-interval-seconds

指定できる値：	なし
デフォルト値：	10 秒
要件：	なし
親要素：	`weblogic-enterprise-bean`

機能

`jms-polling-interval-seconds` 要素は、JMS 送り先に再接続しにいく間隔の秒数を指定します。各メッセージ駆動型 Bean は、関連付けられている JMS 送り先にリスンします。JMS 送り先が別の WebLogic Server インスタンスまたは外部 JMS プロバイダに配置されている場合には、JMS 送り先にアクセスできなくなることもあります。この場合、EJB コンテナは自動的に JMS サーバに再接続しにいきます。JMS サーバが再び起動した時点で、メッセージ駆動型 Bean は再びメッセージを受信できるようになります。

例

次のエントリでは、メッセージ駆動型 Bean の JMS ポーリング間隔を指定しています。

```
<jms-polling-interval-seconds>5</jms-polling-interval seconds>
```

jms-client-id

指定できる値：	なし
デフォルト値：	デフォルト クライアント ID はこの EJB の `ejb-name`。
要件：	`jms-client-id` が JMS トピックに対する恒久サブスクリプションで必要となる。
親要素：	`weblogic-enterprise-bean`

機能

`jms-client-id` は、JMS コンシューマに関連付ける ID を指定します。恒久サブスクリプションを持つメッセージ駆動型 Bean では、関連付けられたクライアント ID が必要になります。個別の接続ファクトリを使用する場合には、その接続ファクトリ上にクライアント ID を設定できます。この場合、メッセージ駆動型 Bean はこのクライアント ID を使用します。

関連する接続ファクトリにクライアント ID がない場合、または、デフォルトの接続ファクトリを使用する場合には、メッセージ駆動型 Bean は jms-cliend-id の値をそれのクライアント ID として使用します。

例

次のエントリでは、JMS コンシューマに関連付ける ID を指定しています。

```
<jms-client-id>MyClientID</jms-client-id>
```

jndi-name

指定できる値：	有効な JNDI 名
デフォルト値：	なし
要件：	`resource-description` と `ejb-reference-description` で必須。

親要素：	`weblogic-enterprise-bean`
	および
	`weblogic-enterprise-bean`
	`  reference-descriptor`
	`  resource-description`
	および
	`weblogic-enterprise-bean`
	`  reference-descriptor`
	`  ejb-reference-description`

機能

`jndi-name` は、WebLogic Server で使用可能な実際の EJB、リソース、または参照の JNDI 名を指定します。

例

9-67 ページの「resource-description」と 9-22 ページの「ejb-reference-description」を参照してください。

local-jndi-name

指定できる値：	有効な JNDI 名
デフォルト値：	なし
要件：	Bean がローカル ホームを持つ場合には必須。
親要素：	`weblogic-enterprise-bean`

機能

`local-jndi-name` 要素は、Bean のローカル ホームの jndi-name を指定します。Bean がリモート ホームとローカル ホームを持つ場合は、それぞれのホームに 1 つずつの JNDI 名を指定する必要があります。

例

次の例では、`local-jndi-name` 要素の構造を示します。

```
<local-jndi-name>weblogic.jndi.WLInitialContext
</local-jndi-name>
```

max-beans-in-cache

指定できる値：	1 から *maxBeans* で指定した値
デフォルト値：	1000
要件：	省略可能
親要素：	weblogic-enterprise-bean, entity-descriptor, entity-cache および weblogic-enterprise-bean stateful-session-descriptor stateful-session-cache

機能

max-beans-in-cache 要素は、メモリに保持可能なこのクラスのオブジェクトの最大数を指定します。max-bean-in-cache の値に達すると、WebLogic Server では、最近クライアントに使用されていない EJB の一部に対してパッシベーションが行われます。また、4-16 ページの「EJB の同時方式」で説明されているように、max-beans-in-cache の値は、EJB を WebLogic Server のキャッシュからいつ削除するかにも影響を与えます。

例

次の例では、WebLogic Server が AccountBean クラスのインスタンスを最大で 200 個キャッシュできるようにします。

```
<weblogic-enterprise-bean>
       <ejb-name>AccountBean</ejb-name>
       <entity-descriptor>
              <entity-cache>
                     <max-beans-in-cache>200</max-beans-in-cache>
              </entity-cache>
       </entity-descriptor>
</weblogic-enterprise-bean>
```

max-beans-in-free-pool

指定できる値：	0 から *maxBeans* で指定した値
デフォルト値：	1000
要件：	省略可能。
親要素：	weblogic-enterprise-bean, stateless-session-descriptor, pool weblogic-enterprise-bean, message-bean-descriptor, pool weblogic-enterprise-bean, entity-descriptor, pool

機能

WebLogic Server は、すべてのエンティティ Bean、ステートレス セッション Bean、およびメッセージ駆動型 Bean クラスに対して EJB のフリー プールを保持します。max-beans-in-free-pool 要素は、このフリー プールのサイズを定義します。詳細については、4-6 ページの「ステートレス セッション EJB のライフサイクル」と 3-3 ページの「メッセージ駆動型 Bean とステートレス セッション EJB との違い」を参照してください。

例

9-61 ページの「pool」を参照してください。

message-driven-descriptor

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	
親要素 :	`weblogic-enterprise-bean`

機能

`message-driven-descriptor` 要素は、メッセージ駆動型 Bean を WebLogic Server の JMS 送り先に関連付けます。この要素は、以下のデプロイメント パラメータを指定します。

- `pool`
- `destination-jndi-name`
- `initial-context-factory`
- `provider-url`
- `connection-factory-jndi-name`

例

次の例では、`message-driven-descriptor` スタンザの構造を示します。

```
<message-driven-descriptor>
       <destination-jndi-name>...</destination-jndi-name>
</message-driven-descriptor>
```

method

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	省略可能。複数の method スタンザを指定して複数の EJB メソッドをコンフィグレーションできる。
親要素 :	weblogic-enterprise-bean transaction-isolation

機能

method 要素は、エンタープライズ Bean のホームまたはリモート インタフェースのメソッドあるいはメソッドのセットを定義します。

例

method スタンザには、以下の要素を指定できます。

```
<method>
       <description>...</description>
       <ejb-name>...</ejb-name>
       <method-intf>...</method-intf>
       <method-name>...</method-name>
       <method-params>...</method-params>
</method>
```

method-intf

指定できる値:	`Home | Remote | Local | Localhome`
デフォルト値:	なし
要件:	省略可能。

親要素:	`weblogic-enterprise-bean`
	`transaction-isolation`
	`method`

機能

`method-intf` では、WebLogic Server がアイソレーション レベル プロパティを適用する EJB インタフェースを指定します (メソッドが複数のインタフェースで同じシグネチャを持つ場合)。

例

9-54 ページの「method」を参照してください。

method-name

指定できる値:	`ejb-jar.xml` で定義した EJB の名前 | *
デフォルト値:	なし
要件:	`method` スタンザで必須。

親要素:	`weblogic-enterprise-bean`
	`transaction-isolation`
	`method`

機能

`method-name` は、WebLogic Server がアイソレーション レベルのプロパティを適用する個々の EJB メソッドの名前を指定します。アスタリスク (*) を使用して EJB のホームおよびリモート インタフェースの全メソッドを指定します。

method-name を指定すると、そのメソッドが指定した ejb-name で使用可能になります。

例

9-54 ページの「method」を参照してください。

method-param

指定できる値：	Java タイプのメソッド パラメータの完全修飾名
デフォルト値：	なし
要件：	method-params で必須。
親要素：	weblogic-enterprise-bean transaction-isolation method method-params

機能

method-param 要素には、Java タイプのメソッド パラメータの完全修飾名を指定します。

例

9-57 ページの「method-params」を参照してください。

method-params

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	省略可能
親要素 :	weblogic-enterprise-bean transaction-isolation method

機能

method-params スタンザには、各メソッド パラメータの Java タイプ名を定義する 1 つまたは複数の要素があります。

例

method-params スタンザには、次のように 1 つまたは複数の method-param 要素が含まれます。

```
<method-params>
       <method-param>java.lang.String</method-param>
       ...
</method-params>
```

persistence

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	コンテナ管理の永続性サービスを使用するエンティティ EJB でのみ必須。
親要素 :	`weblogic-enterprise-bean, entity-descriptor`

機能

`persistence` 要素は、WebLogic Server のエンティティ EJB に対する永続性タイプ、トランザクション コミット動作、`ejbLoad()` 動作、および `ejbStore()` 動作を決定するプロパティを定義します。

- `is-modified-method-name`
- `delay-updates-until-end-of-tx`
- `finders-load-bean`
- `db-is-shared`
- `persistence-use`

例

次の例では、`persistence` 要素の構造を指定します。

```
<entity-descriptor>
       <persistence>
              <is-modified-method-name>...</is-modified-
              method-name>
              <delay-updates-until-end-of-tx>
              </delay-updates-until-end-of-tx>
              <finders-load-beand>...</finders-load-bean>
              <db-is-shared>false</db-is-shared>
              <persistence-use>...</persistence-use>

       </persistence>
</entity-descriptor>
```

persistence-use

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	コンテナ管理の永続性サービスを使用するエンティティ EJB でのみ必須。
親要素 :	`weblogic-enterprise-bean`, `entity-descriptor`, `persistence`

機能

`persistence-use` 要素は、この Bean に使用する永続性タイプの識別子を格納します。

例

次の例では、`persistence-use` スタンザの例を示します。

```
<persistence-use>
       <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
       <type-version>5.1.0</type-version>
       <type-storage>META-INF/weblogic-cmp-jar.xml</type-storage>
</persistence-use>
```

persistent-store-dir

指定できる値：	pstore
要件：	省略可能。
親要素：	weblogic-enterprise-bean stateful-session-descriptor

機能

WebLogic Server がパッシベーションされたステートフル セッション Bean インスタンスの状態を格納するファイル システム ディレクトリを指定します。詳細については、4-12 ページの「パッシベーションされた Bean の永続ストア ディレクトリの指定」を参照してください。

例

```
<stateful-session-descriptor>
   <stateful-session-cache>...</stateful-session-cache>
   <allow-concurrent-calls>...</allow-concurrent-calls>
   <persistent-store-dir>MyPersistenceDirr</persistent-store-dir>
   <stateful-session-clustering>...</stateful-session-clustering>
   <allow-remove-during-transaction>
</stateful-session-descriptor>
```

pool

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	省略可能。
親要素 :	`weblogic-enterprise-bean` `stateless-session-descriptor, message-bean-descriptor, entity-descriptor`

機能

`pool` 要素は、EJB 用の WebLogic Server フリー プールの動作をコンフィグレーションします。

例

`pool` スタンザには、以下の要素を指定できます。

```
<stateless-session-descriptor>
   <pool>
     <max-beans-in-free-pool>500</max-beans-in-free-pool>
     <initial-beans-in-free-pool>250</initial-beans-in-free-pool>
   </pool>
</stateless-session-descriptor>
```

principal-name

指定できる値：	有効な WebLogic Server プリンシパル名
デフォルト値：	なし
要件：	security-role-assignment スタンザには、最低 1 つの principal-name が必須。各 role-name に対しては、複数の principal-name を定義できる。
親要素：	weblogic-enterprise-bean security-role-assignment

機能

principal-name は、指定した role-name に適用される実際の WebLogic Server プリンシパルの名前を指定します。

例

9-71 ページの「security-role-assignment」を参照してください。

provider-url

指定できる値：	有効な名前
デフォルト値：	なし
要件：	initial-context-factory および connection-factory-jndi-name と組み合わせて使用する。
親要素：	weblogic-enterprise-bean message-driven-descriptor

機能

provider-url 要素は、InitialContext が使用する URL プロバイダを指定します。通常、指定するのはホスト：ポートで、initial-context-factory および connection-factory-jndi-name と組み合わせて使用します。

例

次の例では、provider-url 要素の構造を指定します。

```
<message-driven-descriptor>
<provider-url>WeblogicURL:Port</provider-url>
</message-driven-descriptor>
```

read-timeout-seconds

指定できる値：	0 から *maxSeconds* で指定した値。*maxSeconds* は整数の最大値。
デフォルト値：	600
要件：	省略可能。エンティティ EJB でのみ有効。
親要素：	weblogic-enterprise-bean, entity-descriptor, entity-cache

機能

read-timeout-seconds 要素では、Read-Only エンティティ Bean での各 ejbLoad() 呼び出しの間隔を秒数で指定します。0 に設定すると、WebLogic Server では、その Bean がキャッシュされた場合にのみ、ejbLoad() が呼び出されます。

例

次の例では、インスタンスが最初にキャッシュされた場合にのみ WebLogic Server が AccountBean クラスのインスタンスに対して ejbLoad() を呼び出します。

```
<weblogic-enterprise-bean>
    <ejb-name>AccountBean</ejb-name>
        <entity-descriptor>
            <entity-cache>
                <read-timeout-seconds>0</read-timeout-seconds>
            </entity-cache>
        </entity-descriptor>
</weblogic-enterprise-bean>
```

reference-descriptor

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	省略可能。
親要素 :	`weblogic-enterprise-bean`

機能

`reference-descriptor` 要素では、`ejb-jar.xml` ファイル内の参照が、WebLogic Server で実際に使用可能なリソース ファクトリと EJB の JNDI 名にマップされます。

例

`reference-descriptor` スタンザには、リソース ファクトリ参照および EJB 参照を定義するために 1 つまたは複数のスタンザが追加されます。次の例に、これらの要素の構造を示します。

```
<reference-descriptor>
       <resource-description>
              ...
       </resource-description>
       <ejb-reference-description>
              ...
       </ejb-reference-description>
</reference-descriptor>
```

relationship-description

現在、この要素は WebLogic Server でサポートされていません。

replication-type

指定できる値：	`InMemory	None`
デフォルト値：	なし
要件：	省略可能。クラスタ内のステートフル セッション EJB でのみ有効。

親要素：	`weblogic-enterprise-bean`
	`stateful-session-descriptor`
	`stateful-session-clustering`

機能

`replication-type` 要素は、クラスタ内の WebLogic Server インスタンスのステートフル セッション EJB の状態を WebLogic Server がレプリケートするかどうかを指定します。`InMemory` を指定した場合、EJB の状態はレプリケートされます。`None` を指定した場合、状態はレプリケートされません。

詳細については、4-37 ページの「ステートフル セッション EJB のインメモリ レプリケーション」を参照してください。

例

9-74 ページの「stateful-session-clustering」を参照してください。

res-env-ref-name

指定できる値：	ejb-jar.xml ファイルにある有効なリソース環境参照名
デフォルト値：	なし
要件：	なし
親要素：	weblogic-enterprise-bean reference-descriptor resource-env-description

機能

res-env-ref-name はリソース環境参照名を指定します。

例

9-67 ページの「resource-description」を参照してください。

res-ref-name

指定できる値：	ejb-jar.xml ファイルにある有効なリソース参照名
デフォルト値：	なし
要件：	EJB が ejb-jar.xml のリソース参照を指定する場合にのみ必須。
親要素：	weblogic-enterprise-bean reference-descriptor resource-description

機能

res-ref-name は resourcefactory 参照名を指定します。これは、EJB プロバイダによって ejb-jar.xml デプロイメント記述子ファイル内に記載される参照です。

例

9-67 ページの「resource-description」を参照してください。

resource-description

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	省略可能。
親要素 :	`weblogic-enterprise-bean` `reference-descriptor`

機能

`resource-description` 要素は、`ejb-jar.xml` で定義されたリソース参照を、WebLogic Server で使用可能な実際のリソースの JNDI 名にマップします。

例

`resource-description` スタンザには、以下のように要素を追加できます。

```
<reference-descriptor>
       <resource-description>
              <res-ref-name>.. .</res-ref-name>
              <jndi-name>...</jndi-name>
       </resource-description>
       <ejb-reference-description>
              <ejb-ref-name>.. .</ejb-ref-name>
              <jndi-name>.. .</jndi-name>
       </ejb-reference-description>
</reference-descriptor>
```

resource-env-description

指定できる値：	なし (XML スタンザ)
デフォルト値：	なし (XML スタンザ)
要件：	省略可能。
親要素：	`weblogic-enterprise-bean` `reference-descriptor`

機能

`resource-env-description` 要素は、`ejb-jar.xml` で定義されたリソース環境参照を、WebLogic Server で使用可能な実際のリソースの JNDI 名にマップします。

例

`resource-env-description` スタンザには、以下のように要素を追加できます。

```
<reference-descriptor>
       <resource-env-description>
              <res-env-ref-name>.. .</res-env-ref-name>
              <jndi-name>...</jndi-name>
       <reference-env-description>
</reference-descriptor>
```

role-name

指定できる値:	ejb-jar.xml で定義した EJB ロール名
デフォルト値:	なし
要件:	security-role-assignment で必須。
親要素:	weblogic-enterprise-bean security-role-assignment

機能

role-name 要素は、EJB プロバイダが ejb-jar.xml デプロイメント ファイルに指定したアプリケーションのロール名を示します。スタンザの次の principal-name 要素で、WebLogic Server プリンシパルを、指定した role-name にマップします。

例

9-71 ページの「security-role-assignment」を参照してください。

security-permission

指定できる値:	なし (XML スタンザ)
デフォルト値:	なし (XML スタンザ)
要件:	
親要素:	なし

機能

security-permission 要素は、J2EE Sandbox と関連するセキュリティ パーミッションを指定します。

詳細については、Sun によるセキュリティ パーミッション仕様の実装を参照してください。

http://java.sun.com/j2se/1.3/docs/guide/security/PolicyFiles.html#FileSyntax

例

`security-permission` スタンザには、以下の要素を 1 つまたは複数、指定できます。

```
<security-permission> </security-permission>
```

security-permission-spec

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	なし
親要素 :	`security-permission`

機能

`security-permission-spec` 要素は、J2EE sandbox と関連するセキュリティ パーミッションを指定します。

詳細については、Sun によるセキュリティ パーミッション仕様の実装を参照してください。

http://java.sun.com/j2se/1.3/docs/guide/security/PolicyFiles.html#FileSyntax

例

「java.vm.version」に「read」パーミッションを付与して、上書きされないようにするには、次の手順に従います。

1. `security-permission-spec` を次のように設定します。

```
<security-permission>
   <description>Optional explanation goes here</description>
   <security-permission-spec> grant { permission
   java.util.PropertyPermission "java.vm.version", "read"; };
   </security-permission-spec>
</security-permission>
```

2. 次のオプションを使用してサーバを起動するように、startWeblogic スクリプトを変更します。

   ```
   JAVA_OPTIONS=-Djava.security.manager
   ```

3. ドメイン ディレクトリ内に lib という名前のディレクトリを作成します。

4. %WL_HOME%\server\lib\weblogic.policy ファイルに次の行を追加します。

   ```
   add grant codeBase "file:/<Your user_projects
   dir>/YourDomain/lib/-" { permission
   java.security.AllPermission; };
   ```

   EJB スタブのクラスパスが lib であるため、この行が必要です。

security-role-assignment

指定できる値:	なし (XML スタンザ)
デフォルト値:	なし (XML スタンザ)
要件:	ejb-jar.xml がアプリケーション ロールを定義する場合に必須。
親要素:	なし

機能

security-role-assignment 要素は、ejb-jar.xml ファイル内のアプリケーション ロールを、WebLogic Server で使用可能なセキュリティ プリンシパル名にマップします。

例

security-role-assignment スタンザには、以下の要素を 1 つまたは複数、指定できます。

```
<security-role-assignment>
       <role-name>PayrollAdmin</role-name>
       <principal-name>Tanya</principal-name>
       <principal-name>system</principal-name>
       ...
</security-role-assignment>
```

session-timeout-seconds

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	session-timeout-seconds スタンザは、ステートフル セッション EJB でのみ有効。
親要素 :	weblogic-enterprise-bean, stateful-session-descriptor stateful-session-cache

機能

session-timeout-seconds 要素は、パッシベーションされたステートフル セッション Bean がディスクから削除されるまで EJB コンテナが待機する時間を指定します。

idle-timeout-seconds 要素は、ステートフル セッション Bean でパッシベーションが行われるまで、つまりキャッシュから削除されてディスクに書き込まれるまで EJB コンテナが待機する時間を指定します。

以前のリリースでは、EJB コンテナは、パッシベーションされた EJB をディスクから削除するまでの待機時間を指定するためにも idle-timeout-seconds を使用していました。session-timeout-seconds の追加により、ステートフル セッション Bean がキャッシュ内に留まる時間と、それらがディスクに留まる時間を別々の要素で指定できます。

例

次の例では、session-timeout-seconds 要素の指定方法を示します。

```
<session-timeout-seconds>3600</session-timeout-seconds>
```

stateful-session-cache

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	stateful-session-cache スタンザは省略可能。この要素は、ステートフル セッション EJB でのみ有効。
親要素 :	weblogic-enterprise-bean, stateful-session-descriptor

機能

stateful-session-cache 要素は、WebLogic Server 内のステートフル セッション EJB インスタンスのキャッシュに使用する以下のオプションを定義します。

- max-beans-in-cache
- idle-timeout-seconds
- cache-type

WebLogic Server で使用可能なキャッシュ サービスについては、4-2 ページの「EJB のライフサイクル」を参照してください。

例

次の例では、stateful-session-cache 要素の指定方法を示します。

```
<stateful-session-cache>
       <max-beans-in-cache>...</max-beans-in-cache>
       <idle-timeout-seconds>...</idle-timeout-seconds>
       <cache-type>...</cache-type>
</stateful-session-cache>
```

stateful-session-clustering

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	省略可能。クラスタ内のステートフル セッション EJB でのみ有効。
親要素 :	`weblogic-enterprise-bean`, `stateful-session-descriptor`

機能

`stateful-session-clustering` スタンザ要素では、クラスタ内のステートフル セッション EJB インスタンスのクラスタ化動作を指定します。

例

次の例では、`stateful-session-clustering` スタンザの構造を示します。

```
<stateful-session-clustering>
       <home-is-clusterable>true</home-is-clusterable>
       <home-load-algorithm>random</home-load-algorithm>

<home-call-router-class-name>beanRouter</home-call-router-class-name>
       <replication-type>InMemory</replication-type>
</stateful-session-clustering>
```

stateful-session-descriptor

指定できる値 :	なし (XML スタンザ)
デフォルト値 :	なし (XML スタンザ)
要件 :	.jar 内のステートフル セッション EJB ごとに 1 つの stateful-session-descriptor スタンザが必須。
親要素 :	weblogic-enterprise-bean

機能

stateful-session-descriptor 要素では、ステートフル セッション EJB のデプロイメント動作を指定します。

例

次の例では、stateful-session-descriptor スタンザの構造を示します。

```
<stateful-session-descriptor>
   <stateful-session-cache>...</stateful-session-cache>
   <allow-concurrent-calls>...</allow-concurrent-calls>
   <persistent-store-dir>...</persistent-store-dir>
   <stateful-session-clustering>...</stateful-session-clustering>
   <allow-remove-during-transaction>
</stateful-session-descriptor>
```

stateless-bean-call-router-class-name

指定できる値：	有効なルータ クラス名
デフォルト値：	なし
要件：	省略可能。クラスタのステートレス セッション EJB でのみ有効。

親要素：	`weblogic-enterprise-bean`,
	`stateless-session-descriptor`
	`stateless-clustering`

機能

`stateless-bean-call-router-class-name` 要素は、Bean メソッド呼び出しのルーティングに使用するカスタム クラスの名前を指定します。このクラスは `weblogic.rmi.cluster.CallRouter()` を実装する必要があります。指定すると、このクラスのインスタンスは各メソッド呼び出しの前に呼び出されます。ルータ クラスでは、メソッドのパラメータを基に、ルーティングするサーバを選択できます。このクラスは、サーバ名を返すか、または現在のロード アルゴリズムがサーバを選択する必要があることを示す null を返します。

例

9-79 ページの「stateless-clustering」を参照してください。

stateless-bean-is-clusterable

指定できる値：	`true	false`
デフォルト値：	`true`
要件：	省略可能。クラスタのステートレス セッション EJB でのみ有効。

親要素：	`weblogic-enterprise-bean,`
	`stateless-session-descriptor`
	`stateless-clustering`

機能

`stateless-bean-is-clusterable` は、ステートレス セッション Bean の `EJBObject` インタフェースがクラスタ化されているかどうかを示す場合に使用します。クラスタ化されている `EJBObject` は、ロード バランシングとフェイルオーバをサポートしています。

`stateless-bean-is-clusterable` が `true` の場合、クラスタ化されているステートレス セッション Bean のホーム インタフェースが Bean インスタンスを作成すると、クラスタ内のすべてのサーバを示す `EJBObject` スタブがクライアントに返されます。ステートレスという Bean の特性のため、どのインスタンスもすべてのクライアントにサービスできます。

例

9-79 ページの「stateless-clustering」を参照してください。

stateless-bean-load-algorithm

指定できる値：	`round-robin	random	weight-based`
デフォルト値：	`weblogic.cluster.defaultLoadAlgorithm` の値
要件：	省略可能。クラスタのステートレス セッション EJB でのみ有効。

親要素：	`weblogic-enterprise-bean`,
	`stateless-session-descriptor`
	`stateless-clustering`

機能

`stateless-bean-load-algorithm` は、EJB ホームのレプリカ間でのロード バランシングに使用するアルゴリズムを指定します。このプロパティを定義しない場合、WebLogic Server はサーバ プロパティ `weblogic.cluster.defaultLoadAlgorithm` で指定されたアルゴリズムを使用します。

`stateless-bean-load-algorithm` を以下のいずれかの値で定義できます。

- `round-robin` :Bean のホスト サーバ間で順番にロード バランシングを実行します。
- `random` :Bean のホスト サーバ間で EJB ホームのレプリカがランダムにデプロイされます。
- `weight-based` : ホスト サーバの現在の負荷に従って、EJB ホームのレプリカをデプロイするサーバが決まります。

例

9-79 ページの「stateless-clustering」を参照してください。

stateless-bean-methods-are-idempotent

指定できる値:	`true | false`
デフォルト値:	`false`
要件:	省略可能。クラスタのステートレス セッション EJB でのみ有効。

親要素:	`weblogic-enterprise-bean,`
	`stateless-session-descriptor`
	`stateless-clustering`

機能

非推奨 : stateless-bean-methods-are-idempoten 要素は現在、非推奨となっており、WebLogic の将来のバージョンでは削除される予定です。

代わりに、`idempotent-methods` 要素を使用してください。

例

9-79 ページの「stateless-clustering」を参照してください。

stateless-clustering

指定できる値:	なし (XML スタンザ)
デフォルト値:	なし (XML スタンザ)
要件:	省略可能。クラスタのステートレス セッション EJB でのみ有効。
親要素:	`weblogic-enterprise-bean,` `stateless-session-descriptor`

機能

`stateless-clustering` 要素は、WebLogic Server がクラスタ内のステートレス セッション EJB インスタンスをレプリケートする方法を決めるオプションを指定します。

例

次の例では、stateless-clustering スタンザの構造を示します。

```
<stateless-clustering>
   <home-is-clusterable>.../home-is-clusterable>
   <home-load-algorithm>...</home-load-algorithm>
   <home-call-router-class-name>...</home-call-router-class-name>
   <use-serverside-stubs>...</use-serverside-stubs>
   <stateless-bean-is-clusterable>...</stateless-bean-is-
   clusterable>
   <stateless-bean-load-algorithm>...</stateless-bean-load-
   algorithm>
   <stateless-bean-call-router-class-name>...
   </stateless-bean-call-router-class-name>
   <stateless-bean-methods-are-idempotent>...
   </stateless-bean-methods-are-idempotent>
</stateless-clustering>
```

stateless-session-descriptor

要件：	JAR ファイルのステートレス セッション EJB ごとに 1 つの stateless-session-descriptor 要素が必須。
親要素：	weblogic-enterprise-bean

機能

stateless-session-descriptor 要素は、WebLogic Server のステートレス セッション EJB に対するキャッシュ、クラスタ化、および永続性などのデプロイメント動作を定義します。

例

次の例では、stateless-session-descriptor スタンザの構造を示します。

```
<stateless-session-descriptor>
       <pool>...</pool>
       <stateless-clustering>...</stateless-clustering>
</stateless-session-descriptor>
```

transaction-descriptor

指定できる値:	なし (XML スタンザ)
デフォルト値:	なし (XML スタンザ)
要件:	省略可能。
親要素:	`weblogic-enterprise-bean`

機能

`transaction-descriptor` 要素は、WebLogic Server のトランザクション動作を定義するオプションを指定します。現在、このスタンザには `trans-timeout-seconds` という要素のみがあります。

例

次の例では、`transaction-descriptor` スタンザの構造を示します。

```
<transaction-descriptor>
       <trans-timeout-seconds>20</trans-timeout-seconds>
</transaction-descriptor>
```

transaction-isolation

指定できる値:	なし (XML スタンザ)
デフォルト値:	なし (XML スタンザ)
要件:	省略可能。
親要素:	`weblogic-enterprise-jar`

機能

`transaction-isolation` 要素は、EJB に対してメソッド レベル トランザクションのアイソレーション設定を定義します。

例

transaction-isolation スタンザには、以下の要素を指定できます。

```
<transaction-isolation>
       <isolation-level>TransactionSerializable</isolation-level>
       <method>
              <description>...</description>
              <ejb-name>...</ejb-name>
              <method-intf>...</method-intf>
              <method-name>...</method-name>
              <method-params>...</method-params>
       </method>
</transaction-isolation>
```

transport-requirements

指定できる値：	なし
デフォルト値：	なし
要件：	なし
親要素：	weblogic-enterprise-bean, iiop-security-descriptor

機能

transport-requirements 要素は、EJB の転送の要件を指定します。

例

transport-requirements スタンザには、次の要素を指定できます。

```
<iiop-security-descriptor>
       <transport-requirements>
              <confidentiality>supported</confidentiality>
              <integrity>supported</integrity>
```

```
                    <client-cert-authorization>suppoted
                          </client-cert-authentication>
                </transport-requirements>
        </iiop-security-descriptor>
```

trans-timeout-seconds

指定できる値：	0 ～ *max*
デフォルト値：	30
要件：	省略可能。任意のタイプの EJB で有効。
親要素：	weblogic-enterprise-bean, transaction-descriptor

機能

trans-timeout-seconds 要素は、EJB のコンテナで初期化されたトランザクションの最長継続時間を指定します。トランザクションの継続時間が trans-timeout-seconds の値を超えると、WebLogic Server によってトランザクションがロールバックされます。

例

9-81 ページの「transaction-descriptor」を参照してください。

type-identifier

指定できる値：	有効な文字列 WebLogic_CMP_RDBMS は、WebLogic Server RDBMS ベースの永続性を指定します。
デフォルト値：	なし
要件：	コンテナ管理の永続性サービスを使用するエンティティ EJB でのみ必須。
親要素：	weblogic-enterprise-bean, entity-descriptor, persistence persistence-use

機能

type-identifier 要素には、エンティティ EJB の永続性タイプを示すテキストを指定します。WebLogic Server RDBMS ベースの永続性では、WebLogic_CMP_RDBMS という識別子を使用します。別の永続性ベンダを使用している場合の正しい type-identifier についてはベンダのマニュアルを参照してください。

例

WebLogic Server RDBMS ベースの永続性に関する詳細な persistence-use の定義の例については、9-59 ページの「persistence-use」を参照してください。

type-storage

指定できる値：	EJB 1.1 の WebLogic 永続性では `5.1.0` EJB 2.0 の WebLogic 永続性では `6.0`
デフォルト値：	なし
要件：	コンテナ管理の永続性サービスを使用するエンティティ EJB でのみ必須。

親要素：	`weblogic-enterprise-bean,`
	`entity-descriptor,`
	`persistence`
	`persistence-use`

機能

`type-storage` 要素では、この永続性タイプのデータを格納するファイルの絶対パスを定義します。パスは、EJB のデプロイメント JAR ファイルまたはデプロイメント ディレクトリの最上位を基準にしたファイルの場所を指定する必要があります。

WebLogic Server RDBMS ベースの永続性では通常、Bean の永続性データを格納するのに `weblogic-cmp-rdbms-jar.xml` という XML ファイルを使用します。このファイルは、JAR ファイルの `META-INF` サブディレクトリにあります。

例

WebLogic Server RDBMS ベースの永続性に関する詳細な `persistence-use` の定義の例については、9-59 ページの「persistence-use」を参照してください。

type-version

指定できる値：	有効な文字列
デフォルト値：	なし
要件：	コンテナ管理の永続性サービスを使用するエンティティ EJB でのみ必須。
親要素：	`weblogic-enterprise-bean, entity-descriptor, persistence, persistence-use`

機能

たとえば、WebLogic 2.0 CMP の永続性の場合は次の値を使用します。

```
6.0
```

WebLogic 1.1 CMP の永続性の場合は、次の値を使用します。

```
5.1.0
```

この要素は、同じ永続性タイプの複数のバージョンがインストールされている場合に必須です。

注意： WebLogic Server の RDBMS ベースの永続性を使用する場合、指定したバージョンは、WebLogic Server の RDBMS 永続性バージョンと完全に一致させる必要があります。バージョンが一致していないと、次のエラーが発生します。

```
weblogic.ejb.persistence.PersistenceSetupException: Error
initializing the CMP Persistence Type for your bean: No installed
Persistence Type matches the signature of (identifier
'Weblogic_CMP_RDBMS', version 'version_number').
```

例

WebLogic Server RDBMS ベースの永続性に関する詳細な `persistence-use` の定義の例については、persistence-use を参照してください。

weblogic-ejb-jar

指定できる値:	なし
デフォルト値:	なし
要件:	なし
親要素:	なし

機能

`weblogic-ejb-jar` は、EJB デプロイメント記述子の weblogic コンポーネントのルート要素です。

weblogic-enterprise-bean

指定できる値:	
デフォルト値:	
要件:	
親要素:	`weblogic-ejb-jar`

機能

`weblogic-enterprise-bean` 要素には、WebLogic Server 内で利用可能な Bean に関するデプロイメント情報が含まれます。

5.1 の weblogic-ejb-jar.xml デプロイメント記述子ファイルの構造

WebLogic Server 5.1 の `weblogic-ejb-jar.xml` ファイルは、EJB 1.1 Bean で使用する EJB 文書型定義 (DTD) を定義します。これらのデプロイメント記述子要素は WebLogic 固有です。WebLogic Server 5.1 の `weblogic-ejb-jar.xml` の最上位要素は次のとおりです。

- `description`
- `weblogic-version`
- `weblogic-enterprise-bean`
  - `ejb-name`
  - `caching-descriptor`
  - `persistence-descriptor`
  - `clustering-descriptor`
  - `transaction-descriptor`
  - `reference-descriptor`
  - `jndi-name`
  - `transaction-isolation`
- `security-role-assignment`

5.1 の weblogic-ejb-jar.xml デプロイメント記述子要素

以下の節では、WebLogic Server 5.1 の `weblogic-ejb-jar.xml` デプロイメント記述子の要素について説明します。

caching-descriptor

caching-descriptor スタンザは、WebLogic Server キャッシュ内の EJB の数、および EJB に対してパッシベーションが行われるまたは EJB がプールされるまでの時間の長さに影響します。このスタンザは、各要素だけでなく、スタンザ全体が省略可能です。要素が定義されていない場合、WebLogic Server ではデフォルト値が使用されます。

以下は、caching-descriptor スタンザの例であり、この節で説明するキャッシング要素を示しています。

```
<caching-descriptor>
   <max-beans-in-free-pool>500</max-beans-in-free-pool>
   <initial-beans-in-free-pool>50</initial-beans-in-free-pool>
   <max-beans-in-cache>1000</max-beans-in-cache>
   <idle-timeout-seconds>20</idle-timeout-seconds>
   <cache-strategy>Read-Write</cache-strategy>
   <read-timeout-seconds>0</read-timeout-seconds>
</caching-descriptor>
```

max-beans-in-free-pool

注意： この要素は、ステートレス セッション EJB に対してのみ有効です。

WebLogic Server では、すべての Bean クラスに対して EJB のフリー プールが維持管理されています。この省略可能な要素では、フリー プールのサイズを定義します。デフォルトでは、max-beans-in-free-pool は無制限です。フリー プール内の Bean の最大数はメモリによってのみ制限されます。詳細については、4-2 ページの「EJB のライフサイクル」を参照してください。

initial-beans-in-free-pool

注意： この要素は、ステートレス セッション EJB に対してのみ有効です。

initial-bean-in-free-pool の値を指定すると、WebLogic Server では、起動時に、指定した数の Bean インスタンスがフリー プールに生成されます。この方法で Bean インスタンスをフリー プールに格納しておくと、要求が来てから新しいインスタンスを生成せずに Bean に対する初期要求が可能になるため、EJB の初期応答時間が短縮されます。

initial-bean-in-free-pool が定義されていない場合のデフォルト値は 0 です。

max-beans-in-cache

注意： この要素は、ステートフル セッション EJB に対してのみ有効です。

この要素では、メモリの許容範囲内で、このクラスのオブジェクトの最大数を指定します。max-bean-in-cache の値に達すると、WebLogic Server では、最近クライアントに使用されていない EJB の一部に対してパッシベーションが行われます。また、4-2 ページの「EJB のライフサイクル」で説明されているように、max-beans-in-cache の値は、EJB を WebLogic Server のキャッシュからいつ削除するかにも影響を与えます。

max-beans-in-cache のデフォルト値は 100 です。

idle-timeout-seconds

idle-timeout-seconds では、ステートフル EJB がキャッシュに保持される最長時間を定義します。この時間を過ぎると、キャッシュ内の Bean の数が max-beans-in-cache で指定した値を超えている場合、WebLogic Server では Bean インスタンスが削除されます。詳細については、4-2 ページの「EJB のライフサイクル」を参照してください。

idle-timeout-seconds のデフォルト値は 600 です。

cache-strategy

cache-strategy 要素には、以下のいずれかを指定できます。

- Read-Write

- Read-Only

デフォルト値は Read-Write です。

read-timeout-seconds

read-timeout-seconds 要素では、Read-Only エンティティ Bean での各 ejbLoad() 呼び出しの間隔を秒数で指定します。デフォルトでは、read-timeout-seconds は 600 秒に設定されています。この値を 0 に設定すると、WebLogic Server では、その Bean がキャッシュされた場合にのみ、ejbLoad が呼び出されます。

persistence-descriptor

persistence-descriptor スタンザでは、エンティティ EJB に対する永続性オプションを指定します。以下に、persistence-descriptor スタンザに含まれるすべての要素を示します。

```
<persistence-descriptor>
   <is-modified-method-name>. . .</is-modified-method-name>
   <delay-updates-until-end-of-tx>. .
.</delay-updates-until-end-of-tx>
   <persistence-use>
      <type-identifier>. . .</type-identifier>
      <type-version>. . .</type-version>
      <type-storage>. . .</type-storage>
   </persistence-use>
   <db-is-shared>. . .</db-is-shared>
   <stateful-session-persistent-store-dir>
      . . .
   </stateful-session-persistent-store-dir>
   <persistence-use>. . .</persistence-use>
</persistence-descriptor>
```

is-modified-method-name

is-modified-method-name では、EJB の保存時に WebLogic Server によって呼び出されるメソッドを指定します。指定したメソッドはブール値を返す必要があります。メソッドを指定しない場合、WebLogic Server では、EJB は常に変更されていると見なされて保存されます。

メソッドを指定して適切な値を設定すると、パフォーマンスが向上します。ただし、メソッドの戻り値にエラーがあると、データに矛盾が発生する場合があります。詳細については、4-14 ページの「is-modified-method-name を使用した ejbStore() の呼び出しの制限 (EJB 1.1 のみ)」を参照してください。

delay-updates-until-end-of-tx

トランザクションの完了時にトランザクションですべての Bean の永続ストレージを更新するには、このプロパティを true (デフォルト) に設定します。通常、これによって不要な更新を防ぐことができるため、パフォーマンスが向上します。ただし、データベース トランザクション内のデータベース更新の順序は保持されません。

データベースがアイソレーション レベルとして TRANSACTION_READ_UNCOMMITTED を使用している場合は、進行中のトランザクションに関する中間結果を他のデータベースのユーザに表示することもできます。この場合、delay-updates-until-end-of-tx を false に設定して、各メソッド呼び出しの完了時に Bean の永続ストレージを更新するように指定します。詳細については、4-13 ページの「エンティティ EJB に対する ejbLoad() と ejbStore() の動作」を参照してください。

注意： delay-updates-until-end-of-tx を false に設定しても、各メソッド呼び出しの後にデータベース更新が「コミットされた」状態になるわけではありません。更新はデータベースに送信されるだけです。更新は、トランザクションの完了時にのみデータベースにコミットまたはロールバックされます。

persistence-use

persistence-use では、EJB で使用可能な永続性サービスを定義します。複数の永続性サービスを持つ EJB をテストするために、weblogic-ejb-jar.xml で複数の persistence-use エントリを定義できます。

persistence-use には、サービスのプロパティを定義する要素が含まれます。

- type-identifier では、指定した永続性タイプを示すテキストを指定する。たとえば、WebLogic Server RDBMS ベースの永続性では WebLogic_CMP_RDBMS という識別子が使用されます。
- type-version では、指定した永続性タイプのバージョンを指定する。

注意： 指定したバージョンは、WebLogic Server の RDBMS の永続性のバージョンと正確に一致している必要があります。バージョンが一致していないと、次のエラーが発生します。

```
weblogic.ejb.persistence.PersistenceSetupException: Error
initializing the CMP Persistence Type for your bean: No installed
Persistence Type matches the signature of (identifier
'Weblogic_CMP_RDBMS', version 'version_number').
```

- type-storage では、この永続性タイプのデータを格納するファイルの絶対パスを定義する。パスは、EJB のデプロイメント JAR ファイルまたはデプロイメント ディレクトリの最上位を基準にしたファイルの場所を指定する必要があります。

  WebLogic Server RDBMS ベースの永続性では通常、Bean の永続性データを格納するのに weblogic-cmp-rdbms-jar.xml という XML ファイルを使用します。このファイルは、JAR ファイルの META-INF サブディレクトリにあります。

以下は、WebLogic Server RDBMS の永続性について適切な値が指定されている persistence-use スタンザの例です。

```
<persistence-use>
    <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
    <type-version>5.1.0</type-version>
    <type-storage>META-INF\weblogic-cmp-rdbms-jar.xml</type-stora
ge>
</persistence-use>
```

db-is-shared

db-is-shared 要素はエンティティ Bean に対してのみ有効です。true (デフォルト値) に設定すると、WebLogic Server では、EJB データがトランザクション間で変更されると見なされ、各トランザクションの開始時にデータが再ロードされます。false に設定すると、WebLogic Server では、永続ストレージの EJB データに排他的にアクセスすると見なされます。詳細については、4-31 ページの「トランザクション間のキャッシュを使用した ejbStore() の呼び出しの制限」を参照してください。

stateful-session-persistent-store-dir

stateful-session-persistent-store-dir 要素では、WebLogic Server で、パッシベーションが行われたステートフル セッション Bean インスタンスの状態が格納されるファイル システム ディレクトリを指定します。

clustering-descriptor

clustering-descriptor スタンザでは、WebLogic Server クラスタにデプロイされた EJB のレプリケーション プロパティと動作を定義します。clustering-descriptor スタンザとその各要素は省略可能であり、単一サーバ システムには適用できません。

以下に、clustering-descriptor スタンザに含まれるすべての要素を示します。

```
<clustering-descriptor>
   <home-is-clusterable>. . .</home-is-clusterable>
   <home-load-algorithm>. . .</home-load-algorithm>
   <home-call-router-class-name>...</home-call-router-class-name>
   <stateless-bean-is-clusterable>...</stateless-bean-is-
   clusterable>
   <stateless-bean-load-algorithm>
   </stateless-bean-load-algorithm>
   <stateless-bean-call-router-class-name>
   </stateless-bean-call-router-class-name>
   <stateless-bean-methods-are-idempotent>
   </stateless-bean-methods-are-idempotent>
</clustering-descriptor>
```

home-is-clusterable

この要素は、true または false に設定できます。home-is-clusterable が「true」の場合、クラスタ内の複数の WebLogic Server から EJB をデプロイできます。ホーム スタブの呼び出しは、Bean がデプロイされるサーバ間で負荷が分散されます。Bean のホスト サーバにアクセスできなかった場合、呼び出しは、Bean を提供する他のサーバに自動的にフェイルオーバします。

home-load-algorithm

home-load-algorithm では、EJB ホームのレプリカ間でのロード バランシングに使用するアルゴリズムを指定します。このプロパティを定義しない場合、WebLogic Server はサーバ プロパティ weblogic.cluster.defaultLoadAlgorithm で指定されたアルゴリズムを使用します。

home-load-algorithm は以下のいずれかの値に定義できます。

- round-robin : Bean のホスト サーバ間で順番にロード バランシングを実行します。
- random : Bean のホスト サーバ間で EJB ホームのレプリカがランダムにデプロイされます。
- weight-based : ホスト サーバの現在の負荷に従って、EJB ホームのレプリカをデプロイするサーバが決まります。

home-call-router-class-name

home-call-router-class-name では、Bean メソッド呼び出しのルーティングに使用するカスタム クラスを指定します。このクラスは weblogic.rmi.cluster.CallRouter() を実装する必要があります。指定すると、このクラスのインスタンスは各メソッド呼び出しの前に呼び出されます。ルータ クラスでは、メソッドのパラメータを基に、ルーティングするサーバを選択できます。このクラスは、サーバ名を返すか、または現在のロード アルゴリズムがサーバを選択する必要があることを示す null を返します。

stateless-bean-is-clusterable

stateless-bean-is-clusterable は、ステートレス セッション Bean の EJBObject インタフェースがクラスタ化されているかどうかを示す場合に使用します。クラスタ化されている EJBObject は、ロード バランシングとフェイルオーバをサポートしています。

stateless-bean-is-clusterable が true の場合、クラスタ化されているステートレス セッション Bean のホーム インタフェースが Bean インスタンスを作成すると、クラスタ内のすべてのサーバを示す EJBObject スタブがクライアントに返されます。ステートレスという Bean の特性のため、どのインスタンスもすべてのクライアントにサービスできます。

stateless-bean-load-algorithm

このプロパティは home-load-algorithm に似ていますが、ステートレス セッション EJB にのみ適用できます。

stateless-bean-call-router-class-name

このプロパティは home-call-router-class-name に似ていますが、ステートレス セッション EJB にのみ適用できます。

stateless-bean-methods-are-idempotent

この要素は、true または false に設定できます。同一引数での同一メソッドの多重呼び出しが1回だけの呼び出しとまったく同じになるように設計されている Bean に対してのみ、stateless-bean-methods-are-idempotent を true に設定します。これによって、フェイルオーバ ハンドラは、失敗した呼び出しが失敗したサーバで実際に完了していたかどうかがわからなくても失敗した呼び出しを再試行できます。このプロパティを true に設定すると、Bean を提供する他のサーバが使用できるようになっている限り、Bean スタブは障害から自動的に回復できます。

注意： このプロパティは、ステートレス セッション EJB にのみ適用できます。

transaction-descriptor

transaction-descriptor スタンザには、WebLogic Server のトランザクション動作を定義する要素があります。現在、このスタンザには 1 つの要素のみがあります。

```
<transaction-descriptor>
   <trans-timeout-seconds>20</trans-timeout-seconds>
<transaction-descriptor>
```

trans-timeout-seconds

trans-timeout-seconds 要素は、EJB のコンテナで初期化されたトランザクションの最長継続時間を指定します。トランザクションの継続時間が trans-timeout-seconds の値を超えると、WebLogic Server によってトランザクションがロールバックされます。

trans-timeout-seconds に値を指定しなかった場合、コンテナで初期化されたトランザクションはデフォルトで 30 秒後にタイムアウトになります。

reference-descriptor

reference-descriptor スタンザでは、ejb-jar.xml ファイル内の参照が、WebLogic Server で実際に使用可能なリソース ファクトリと EJB の JNDI 名にマップされます。

reference-descriptor スタンザには、リソース ファクトリ参照および EJB 参照を定義するために 1 つまたは複数のスタンザが追加されます。次の例に、これらの要素の構造を示します。

```
<reference-descriptor>
   <resource-description>
      <res-ref-name>.. .</res-ref-name>
      <jndi-name>.. .</jndi-name>
   </resource-description>
```

```
    <ejb-reference-description>
       <ejb-ref-name>.. .</ejb-ref-name>
       <jndi-name>.. .</jndi-name>
    </ejb-reference-description>
</reference-descriptor>
```

resource-description

以下の要素で、各 resource-description を定義します。

- res-ref-name ではリソース参照名を指定する。これは、EJB プロバイダによって ejb-jar.xml デプロイメント記述子ファイル内に記載される参照です。
- jndi-name では、WebLogic Server で使用可能な実際のリソース ファクトリの JNDI 名を指定する。

ejb-reference-description

以下の要素で、各 ejb-reference-description を定義します。

- ejb-ref-name は EJB 参照名を指定する。これは、EJB プロバイダによって ejb-jar.xml デプロイメント記述子ファイル内に記載される参照です。
- jndi-name では、WebLogic Server で使用可能な実際の EJB の JNDI 名を指定する。

enable-call-by-reference

デフォルトでは、同じ EAR から呼び出された EJB メソッドは引数を参照で渡します。パラメータはコピーされないので、これによってメソッド呼び出しのパフォーマンスが向上します。

enable-call-by-reference を false に設定すると、EJB 1.1 の仕様に従って EJB メソッドへのパラメータがコピーされ (値で渡され) ます。EJB がリモートで (同じアプリケーション以外から) 呼び出される場合は、常に値で渡す必要があります。

jndi-name

jndi-name 要素は、Bean、リソース、または参照の JNDI 名を指定します。

transaction-isolation

transaction-isolation スタンザでは、EJB メソッドに対するトランザクションのアイソレーション レベルを指定します。このスタンザは、EJB メソッドの範囲に適用される 1 つまたは複数の isolation-level 要素で構成されます。次に例を示します。

```
<transaction-isolation>
        <isolation-level>TransactionSerializable</isolation-level>
        <method>
                <description>...</description>
                <ejb-name>...</ejb-name>
                <method-intf>...</method-intf>
                <method-name>...</method-name>
                <method-params>...</method-params>
        </method>
</transaction-isolation>
```

以降の節では、transaction-isolation 内の各要素について説明します。

isolation-level

isolation-level では、特定の EJB メソッドに適用される有効なトランザクションのアイソレーション レベルを定義します。isolation-level には以下の値を指定できます。

- TransactionReadUncommitted：トランザクションはコミットしていない他のトランザクションの更新を参照できます。
- TransactionReadCommitted：トランザクションはコミットされた他のトランザクションの更新だけを参照できます。

- TransactionRepeatableRead：トランザクションでデータの一部を読み取ると、そのデータが他のトランザクションで変更されても、最初の読み取り時と同じ値が返されます。
- TransactionSerializable：このトランザクションを同時に複数回実行すると、トランザクションを順番に複数回実行することと同じことになります。

Oracle データベースの場合のみ：

- TransactionReadCommittedForUpdate
- TransactionReadCommittedForUpdateNoWait

  Oracle 固有の isolation-level 値の背景情報については、4-45 ページの「Oracle データベースに関する特別な注意」を参照してください。

異なるアイソレーション レベルの関係と各アイソレーション レベルのサポートの詳細については、各データベースのマニュアルを参照してください。

method

method スタンザは、アイソレーション レベルを適用する EJB メソッドを定義します。method では、以下の要素を使用してメソッドの範囲を定義します。

- description は、メソッドを説明する省略可能な要素。
- ejb-name では、WebLogic Server によってアイソレーション レベル プロパティが適用される EJB を指定します。
- method-intf は、指定したメソッドが EJB のホームまたはリモートのいずれのインタフェースに存在するかを示す、省略可能な要素。この要素の値は、「Home」または「Remote」でなければなりません。method-intf を指定しない場合は、どちらのインタフェースにあるメソッドにもアイソレーションを適用できます。
- method-name では、EJB メソッドの名前、またはすべての EJB メソッドを示すアスタリスク (*) を指定する。
- method-params は、各メソッドのパラメータの Java クラスのタイプを示す、省略可能なスタンザ。各パラメータのタイプは、method-params スタンザ内の method-param 要素を使用して、順に示す必要があります。

たとえば、次の method スタンザは、「AccountBean」EJB 内のすべてのメソッドを示しています。

```
<method>
        <ejb-name>AccountBean</ejb-name>
        <method-name>*</method-name>
</method>
```

次の method スタンザは、「AccountBean」のリモート インタフェース内のすべてのメソッドを示しています。

```
<method>
        <ejb-name>AccountBean</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>*</method-name>
</method>
```

security-role-assignment

security-role-assignment スタンザでは、ejb-jar.xml ファイル内のアプリケーション ロールが、WebLogic Server で使用可能なセキュリティ プリンシパル名にマップされます。

security-role-assignment には、以下の要素を 1 つまたは複数、指定できます。

- role-name は、EJB プロバイダが ejb-jar.xml デプロイメント ファイルに指定したアプリケーションのロール名。
- principal-name では、実際の WebLogic Server プリンシパルの名前を指定する。

10 weblogic-cmp-rdbms-jar.xml 文書型定義

この章では、weblogic 固有の XML 文書型定義 (DTD) ファイル、`weblogic-cmp-rdbms-jar.xml` ファイルの EJB 1.1 および EJB 2.0 デプロイメント記述子要素について説明します。これらの定義を使用して、EJB デプロイメントを構成する WebLogic 固有の `weblogic-cmp-rdbms-jar.xml` ファイルを作成します。

以下の節では、DOCTYPE ヘッダ情報を含めて、2 つのバージョンの WebLogic 固有の XML をリファレンス形式で詳細にまとめてあります。これらのデプロイメント記述子要素を使用して、コンテナ管理による永続性 (CMP) を指定します。

- EJB デプロイメント記述子
- DOCTYPE ヘッダ情報
- 2.0 の weblogic-cmp-rdbms-jar.xml デプロイメント記述子ファイルの構造
- 2.0 の weblogic-cmp-rdbms-jar.xml デプロイメント記述子要素
- 1.1 の weblogic-cmp-rdbms-jar.xml デプロイメント記述子ファイルの構造
- 1.1 の weblogic-cmp-rdbms-jar.xml デプロイメント記述子要素

EJB デプロイメント記述子

EJB デプロイメント記述子は、エンタープライズ Bean の構造およびアセンブリ情報を提供します。この情報を指定するには、3 つの EJB XML DTD ファイルのデプロイメント記述子に値を指定します。ファイルは次のとおりです。

- `ejb-jar.xml`
- `weblogic-ejb-jar.xml`

- weblogic-cmp-rdbms-jar.xml

この 3 つの XML ファイルを EJB および他のクラスと一緒にデプロイ可能なコンポーネント、通常は ejb.jar という JAR ファイルにパッケージ化します。

ejb-jar.xml ファイルは、Sun Microsystems の ejb.jar.xml ファイルのデプロイメント記述子に基づいています。その他の 2 つの XML ファイルは weblogic 固有のファイルで、weblogic-ejb-jar.xml と weblogic-cmp-rdbms-jar.xml のデプロイメント記述子に基づいています。

DOCTYPE ヘッダ情報

XML デプロイメント ファイルの編集、作成時に、各デプロイメント ファイルに対して正しい DOCTYPE ヘッダを指定することが重要です。特に、DOCTYPE ヘッダ内部に不正な PUBLIC 要素を使用すると、原因究明が困難なパーサ エラーになることがあります。各 XML デプロイメント ファイルで適切な PUBLIC 要素は、次のとおりです。

WebLogic Server 固有の weblogic-cmp-rdbms-jar.xml ファイルの PUBLIC 要素には、次のようにテキストを指定する必要があります。

XML ファイル	PUBLIC 要素の文字列
weblogic-cmp-rdbms -jar.xml	'-// BEA Systems, Inc.//DTD WebLogic 7.0.0 EJB RDBMS Persistence//EN' 'http://www.bea.com/servers/wls700/dtd/weblogic-rdbms20-persistence-700.dtd'
weblogic-cmp-rdbms -jar.xml	'-// BEA Systems, Inc.//DTD WebLogic 6.0.0 EJB RDBMS Persistence//EN' 'http://www.bea.com/servers/wls600/dtd/weblogic-rdbms20-persistence-600.dtd'

Sun Microsystems 固有の ejb-jar ファイルの PUBLIC 要素には、次のようにテキストを指定する必要があります。

XML ファイル	PUBLIC 要素の文字列
ejb-jar.xml	'-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'
ejb-jar.xml	'-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN' 'http://www.java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'

たとえば、weblogic-cmp-rdbms-jar.xml ファイルの DOCTYPE ヘッダは次のようになります。

```
<!DOCTYPE weblogic-cmp-rdbms-jar PUBLIC
'-//BEA Systems, Inc.//DTD WebLogic 7.0.0 EJB RDBMS
Persistence//EN'
'http://www.bea.com/servers/wls700/dtd/weblogic-rdbms20-persisten
ce-700.dtd '>
```

XML の解析ユーティリティ (ejbc など) でヘッダ情報が不正な XML ファイルを解析すると、次のようなエラー メッセージが表示されることがあります。

```
SAXException: This document may not have the identifier 'identifier_name'
```

identifier_name には通常、PUBLIC 要素内の不正な文字列が表示されます。

検証用 DTD (Document Type Definitions : 文書型定義)

XML ファイルの内容および要素の配置は、使用する各ファイルの文書型定義 (DTD) に準拠している必要があります。WebLogic Server ユーティリティでは、XML デプロイメント ファイルの DOCTYPE ヘッダ内に埋め込まれた DTD は無視され、代わりにサーバと共にインストールされた DTD の場所が使用されます。ただし、DOCTYPE ヘッダ情報には、パーサ エラーを避けるために有効な URL 構文を指定する必要があります。

注意： ほとんどのブラウザでは、.dtd ファイルの内容は表示されません。DTD ファイルの内容をブラウザで見るには、リンクをテキスト ファイルとして保存し、テキスト エディタで開いて表示します。

weblogic-cmp-rdbms-jar.xml

以下のリンクでは、WebLogic Server で使用される weblogic-cmp-rdbms-jar.xml デプロイメント ファイル用のパブリック DTD の場所が示されています。

- weblogic-cmp-rdbms-jar.xml 2.0 DTD の場合：

  http://www.bea.com/servers/wls700/dtd/weblogic-rdbms20-persistence-700.dtd には、エンティティ EJB のコンテナ管理による永続性プロパティを定義する DTD が含まれています。この DTD は WebLogic Server バージョン 6.0 から変更されていますが、WebLogic Server RDBMS ベースの永続性を使用するエンティティ EJB に対して weblogic-cmp-rdbms-jar.xml ファイルを指定する必要があります。

  既存の DTD ファイルは次の場所にあります。

  http://www.bea.com/servers/wls700/dtd/weblogic-rdbms-persistence-700.dtd

  http://www.bea.com/servers/wls600/dtd/weblogic-rdbms20-persistence-600.dtd には、エンティティ EJB のコンテナ管理による永続性プロパティを定義する DTD が含まれています。この DTD は WebLogic Server バージョン 5.1 から変更されていますが、WebLogic Server RDBMS ベースの永続性を使用するエンティティ EJB に対して weblogic-cmp-rdbms-jar.xml ファイルを指定する必要があります。

  既存の DTD ファイルは次の場所にあります。

  http://www.bea.com/servers/wls600/dtd/weblogic-rdbms-persistence-600.dtd

ejb-jar.xml

以下のリンクでは、WebLogic Server で使用される ejb-jar.xml デプロイメント ファイル用のパブリック DTD の場所が示されています。

- ejb-jar.xml 2.0 DTD の場合：

`http://www.java.sun.com/dtd/ejb-jar_2_0.dtd` には、すべての EJB で必要な標準 `ejb-jar.xml` デプロイメント ファイル用の DTD が含まれています。この DTD は、JavaSoft EJB 2.0 仕様の一部として維持管理されています。`ejb-jar.dtd` で使用される要素については JavaSoft 仕様を参照してください。

- `ejb-jar.xml` 1.1 DTD の場合：

  `ejb-jar.dtd` には、すべての EJB で必須の標準 `ejb-jar.xml` デプロイメント ファイル用の DTD が含まれています。この DTD は、JavaSoft EJB 1.1 仕様の一部として維持管理されています。`ejb-jar.dtd` で使用される要素については、JavaSoft 仕様を参照してください。

注意： `ejb-jar.xml` デプロイメント記述子の説明については、該当する JavaSoft EJB 仕様を参照してください。

2.0 の weblogic-cmp-rdbms-jar.xml デプロイメント記述子ファイルの構造

`weblogic-cmp-rdbms-jar.xml` ファイルは、WebLogic Server の RDBMS ベースの永続性サービスを使用するエンティティ EJB のデプロイメント記述子を定義します。EJB コンテナでは、WebLogic Server バージョン 6.x で提供された XML とは異なるバージョンの `weblogic-cmp-rdbms-jar.xml` を使用します。

WebLogic Server バージョン 7.0 にデプロイする旧バージョンの EJB 1.1 用 `weblogic-cmp-rdbms-jar.xml` の DTD も使用できます。ただし、CMP 2.0 の新機能を使用する場合は、下記の新しい DTD を使用する必要があります。

WebLogic Server 7.0 の `weblogic-cmp-rdbms-jar.xml` の最上位要素は、`weblogic-rdbms-jar` スタンザで構成されます。

```
description
weblogic-version
weblogic-rdbms-jar
        weblogic-rdbms-bean
                ejb-name
```

```
        data-source-name
        table-map
        field-group
        relationship-caching
        weblogic-query
        delay-database-insert-until
        automatic-key-generation
        check-exists-on-method

weblogic-rdbms-relation
        relation-name
        table-name
        weblogic-relationship-role
create-default-dbms-tables
validate-db-schema-with
database-type
```

2.0 の weblogic-cmp-rdbms-jar.xml デプロイメント記述子要素

automatic-key-generation

指定できる値：	なし
デフォルト値：	なし
要件：	省略可能
親要素：	weblogic-rdbms-bean
デプロイメント ファイル：	weblogic-cmp-rdbms-jar.xml

機能

automatic-key-generation 要素は、シーケンス / キー生成機能の使い方を指定します。

例

XML スタンザには、以下の要素を指定できます。

```
<automatic-key-generation>
       <generator-type>ORACLE</generator-type>
       <generator-name>test_sequence</generator-name>
       <key-cache-size>10</key-cache-size>
</automatic-key-generation>
```

```
<automatic-key-generation>
       <generator-type>SQL-SERVER</generator-type>
</automatic-key-generation>
```

```
<automatic-key-generation>
       <generator-type>NAMED_SEQUENCE_TABLE</generator-type>
       <generator-name>MY_SEQUENCE_TABLE_NAME</generator-name>
       <key-cache-size>100</key-cache-size>
</automatic-key-generation>
```

caching-element

指定できる値：	なし
デフォルト値：	なし
要件：	なし
親要素：	weblogic-rdbms-jar weblogic-rdbms-bean relationship-caching
デプロイメント ファイル：	weblogic-cmp-rdbms-jar.xml

機能

caching-element は、関連する Bean に対してコンテナ管理による関係 (cmr-field) と、関連する Bean 内の group-name を指定します。group-name を指定しなかった場合、デフォルトの group-name (全フィールドをロード) が使用されます。group-name の詳細については、10-32 ページの「group-name」を参照してください。

caching-element 記述子は、関連する Bean に対してコンテナ管理による関係 (cmr-field) と、関連する Bean 内の group-name を指定します。group-name を指定しなかった場合、デフォルトの group-name (全フィールドをロード) が使用されます。group-name の詳細については、10-32 ページの「group-name」を参照してください。

WebLogic Server バージョン 7.0 サービス パック 3 から、EJB コンテナでは複数の caching-element 下位要素を指定できるようになりました。関連する DTD エントリは次のようになります。

```
<!ELEMENT caching-element (
cmr-field,
group-name?,
caching-element*
```

```
)>
```

以前 の DTD エントリは次のようになっていました。

```
<!ELEMENT caching-element (
cmr-field,
group-name?,
caching-element?
)>
```

例

10-44 ページの「relationship-caching」を参照してください。

caching-name

指定できる値：	有効な名前
デフォルト値：	なし
要件：	なし

親要素：	`weblogic-rdbms-jar`
	`weblogic-rdbms-bean`
	`relationship-caching`

デプロイメント ファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`caching-name` 要素は、リレーションシップ キャッシュの名前を指定します。

例

10-44 ページの「relationship-caching」を参照してください。

check-exists-on-method

指定できる値：	`True|False`
デフォルト値：	`False`
要件：	
親要素：	`weblogic-rdbms-bean`
デプロイメント ファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

デフォルトでは、EJB コンテナはトランザクションの完了を待機してから、コンテナ管理による永続性 (CMP) エンティティ Bean の有無をチェックします。これにより、大半のアプリケーションで高いパフォーマンスを実現しつつ、十分なレベルのチェックが提供されます。

EJB コンテナが Bean 上で呼び出されたビジネス メソッドの完了前に Bean の有無をチェックするように指定するには、`check-exists-on-method` を `True` に設定します。これは、削除されたコンテナ管理のエンティティ Bean 上で何らかのビジネス メソッドが呼び出されると、直ちにコンテナがアプリケーションに対して通知を行うことを意味します。

例

次の例では、削除された CMP エンティティ上でビジネス メソッドが呼び出されたことを WebLogic Server がアプリケーションに通知するように指定しています。

```
<check-exists-on-method>True</check-exists-on-method>
```

cmp-field

指定できる値：	有効な名前
デフォルト値：	なし
要件：	このフィールドは大文字 / 小文字を区別するので、Bean のフィールド名と正しく一致していること。また、`ejb-jar.xml` に `cmp-entry` が指定されていること。

親要素：	`weblogic-rdbms-bean`
	`  field-map`
	`weblogic-rdbms-relation`
	`  field-group`

デプロイメント ファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

この名前は、Bean インスタンスのマップされたフィールドを指定します。Bean インスタンスのフィールドには、データベースから取得した情報が指定されている必要があります。

例

10-27 ページの「field-map」を参照してください。

cmr-field

指定できる値：	有効な名前
デフォルト値：	なし
要件：	`cmr-field` で参照されるフィールドが、対応する `cmr-field` エントリを `ejb-jar.xml` に持っていること。
親要素：	`weblogic-rdbms-relation` `field-group`
デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`cmr-field` 要素は、コンテナ管理による関係フィールド (cmr-field) 名を指定します。

例

`XML` スタンザには、以下の要素を指定できます。

```
<weblogic-rdbms-jar>
       <weblogic-rdbms-relation>
              <field-group>employee</field-group>
```

```
                                <cmp-field>employee stock
purchases</cmp-field>
                                <cmr-field>stock options</cmr-field>
        </weblogic-rdbms-relation>
</weblogic-rdbms-jar>
```

column-map

指定できる値：	なし
デフォルト値：	なし
要件：	`foreign-key-column` がリモート Bean を参照する場合は、`key-column` 要素を指定しないこと。
親要素：	`weblogic-rdbms-bean` `weblogic-relationship-role`
デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

この要素は、データベース内のテーブルの外部キー カラムと対応する主キーのマッピングを表します。2 つのカラムは同じテーブルにある場合も別のテーブルにある場合もあります。カラムが属しているテーブルは、`column-map` 要素がデプロイメント記述子に表示されるコンテキストに対しては暗黙的です。

例

`XML` スタンザには、以下の要素を指定できます。

```
<weblogic-rdbms-jar>
        <weblogic-rdbms-bean>
```

```
                <column-map

<foreign-key-column>account-id</foreign-key-column>
                                <key-column>id</key-column>
                </column-map>


  </weblogic-rdbms-bean>
</weblogic-rdbms-jar>
```

create-default-dbms-tables

指定できる値：	True | False
デフォルト値：	False
要件：	この要素は、開発および試作段階で役立つ場合にのみ使用する。使用する DBMS CREATE 文のテーブル スキーマがコンテナの最適な定義になる。一般に、プロダクション環境にはより正確なスキーマ定義が必要になる。
親要素：	`weblogic-rdbms-jar`
デプロイメント ファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`create-default-dbms-table` 要素は、デプロイメント ファイルおよび Bean クラスの記述に基づいてデフォルト テーブルを自動作成する機能を有効化 / 無効化します。False に設定すると、この機能は無効化されるので、テーブルは自動的に作成されません。`True` に設定すると、この機能は有効化されるので、テーブルは自動的に作成されます。`TABLE CREATION` が失敗した場合、`Table Not Found` エラーが送出されるので、テーブルを手動で作成しなければなりません。

例

次の例では、create-default-dbms-tables 要素を指定します。

```
<create-default-dbms-tables>True</create-default-dbms-tables>
```

database-type

指定できる値：	DB2 | INFORMIX | ORACLE | SQL_SERVER | SYBASE | POINTBASE
デフォルト値：	なし
要件：	なし
親要素：	weblogic-rdbms-jar
デプロイメントファイル：	weblogic-cmp-rdbms-jar.xml

機能

database-type 要素は、基盤 DBMS として使用するデータベースを指定します。

例

次の例では、基盤データベースを指定しています。

```
<database-type>POINTBASE</database-type>
```

data-source-name

指定できる値：	この Bean に対するすべてのデータベース接続で使用するデータ ソースの有効な名前
デフォルト値：	なし
要件：	データベース接続の標準 WebLogic Server JDBC データ ソースとして定義すること。データ ソースの詳細については、『WebLogic JDBC プログラマーズ ガイド』を参照。
親要素：	`weblogic-rdbms-bean`
デプロイメント ファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

この Bean のすべてのデータベース接続に使用する JDBC データ ソース名を指定する `data-source-name` です。

例

10-51 ページの「table-name」を参照してください。

db-cascade-delete

指定できる値：	
デフォルト値：	なし
要件：	Oracle データベースに対してのみサポート。1 対 1 または 1 対多関係についてのみ指定可能。
親要素：	`weblogic-rdbms-bean` `weblogic-relationship-role`
デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`db-cascade-delete` 要素は、データベース カスケード機能を有効にするかどうかを指定します。この要素を指定しなかった場合、WebLogic Server では、データベース カスケード削除が指定されていないものと見なされます。

例

5-55 ページの「カスケード削除メソッド」を参照してください。

dbms-column

指定できる値：	有効な名前
デフォルト値：	なし
要件：	大文字 / 小文字を区別しないデータベースの場合でも、`dbms-column` では大文字 / 小文字を区別する。
親要素：	`weblogic-rdbms-bean` `field-map`
デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

フィールドがマップされるデータベース カラムの名前です。

例

10-27 ページの「field-map」を参照してください。

dbms-column-type

指定できる値：	有効な名前
デフォルト値：	なし
要件：	Oracle データベースでのみ使用可能。
親要素：	weblogic-rdbms-bean field-map
デプロイメントファイル：	weblogic-cmp-rdbms-jar.xml

機能

dbms-column-type 要素は、現在のフィールドを Oracle データベース内の Blob または Clob、または Sybase データベース内の LongString または SybaseBinary にマップします。この要素には、以下のいずれかを指定できます。

- OracleBlob
- OracleCLob
- LongString
- SybaseBinary

例

```
<field-map>
       <cmp-field>photo</cmp-field>
       <dbms-column>PICTURE</dbms-column>
       <dbms_column-type>OracleBlob</dbms-column-type>
</field-map>
```

description

指定できる値：	なし
デフォルト値：	なし
要件：	なし
親要素：	weblogic-rdbms-jar weblogic-rdbms-bean weblogic-query
デプロイメントファイル：	weblogic-cmp-rdbms-jar.xml

機能

description 要素は、親要素を示すテキストの指定に使用します。

例

次の例では、description 要素を指定します。

```
<description>Contains a description of parent element</description>
```

delay-database-insert-until

指定できる値：	`ejbCreate | ejbPostCreate | commit`
デフォルト値：	`ejbPostCreate`
要件：	`cmr-field` が null 値を許可しない `foreign-key column` にマップされている場合、データベースの挿入は `ejbPostCreate` の後に遅延される。この場合、`cmr-field` を `ejbPostCreate` で null でない値に設定してから Bean をデータベースに挿入しなければならない。 `cmr-fields` は、Bean の主キーが不明な段階で `ejbCreate` の中で設定することができない。
親要素：	`weblogic-rdbms-bean`
デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`delay-database-insert-until` 要素は、RDBMS CMP を使用する新しい Bean をデータベースに挿入する正確な時間を指定します。

`ejbPostCreate` メソッドが Bean の永続フィールドを変更するまで、データベースの挿入を遅らせることをお勧めします。これにより、不要な保存操作を行わずに済むので、パフォーマンスが向上します。

最大限の柔軟性を実現するには、関連 Bean を `ejbPostCreate` メソッドで作成することは避ける必要があります。データベースの制約によって関連 Bean が未作成の Bean を参照できない場合、データベースの挿入を遅らせることができなくなる可能性があります。

例

次の例では、`delay-database-insert-until` 要素を指定します。

```
<delay-database-insert-until>ejbPostCreate</delay-database-insert
-until>
```

ejb-name

指定できる値：	EJB の有効な名前
デフォルト値：	なし
要件：	`ejb-jar.xml` で定義した CMP エンティティ Bean の `ejb-name` に一致していること。
親要素：	`weblogic-rdbms-bean`
デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

ejb-cmp-rdbms.xml で定義した EJB を指定する名前です。この名前は、`ejb-jar.xml` で定義した CMP エンティティ Bean の `ejb-name` に一致している必要があります。

例

10-51 ページの「table-name」を参照してください。

enable-tuned-updates

指定できる値：	True/False
デフォルト値：	True
要件：	
親要素：	weblogic-rdbms-bean
デプロイメントファイル：	weblogic-cmp-rdbms-jar.xml

機能

enable-tuned-updates 要素は、ejbStore が呼び出された場合に、EJB コンテナがコンテナ管理フィールドの変更の有無を自動的に判定し、変更されたフィールドだけをデータベースに書き込むように指定します。

例

次の例では、enable-tuned-updates 要素の指定方法を示します。

```
<enable-tuned-updates>True</enable-tuned-updates>
```

field-group

指定できる値：	有効な名前
デフォルト値：	指定したグループを持たないファインダと関係に対して、`default` という特殊なグループを使用する。
要件：	デフォルト グループには、Bean の cmp-field がすべて含まれるが、cmr-field は含まれない。
親要素：	`weblogic-rdbms-relation`
デプロイメント ファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`field-group` 要素は、Bean の cmp-field と cmr-field のサブセットを表します。Bean 内の関連フィールドを、障害のあったグループに 1 つのユニットとしてまとめることができます。グループをファインダまたは関係に関連付けることができます。それによって、ファインダを実行するか、または関係に従った結果として Bean がロードされたときに、グループ内の指定フィールドのみがロードされます。

フィールドは複数のグループに関連付けられている場合があります。この場合、フィールドに対して `getXXX` メソッドを実行すると、そのフィールドを含む最初のグループで障害が発生します。

例

`XML` スタンザには、以下の要素を指定できます。

```
<weblogic-rdbms-bean>
       <ejb-name>XXXBean</ejb-name>
       <field-group>
              <group-name>medical-data</group-name>
```

```
                <cmp-field>insurance</cmp-field>
                <cmr-field>doctors</cmr-fields>
        </field-group>
</weblogic-rdbms-bean>
```

field-map

指定できる値：	有効な名前
デフォルト値：	なし
要件：	データベースのカラムにマップされたフィールドが、Bean の CMP フィールドに対応していること。
親要素：	`weblogic-rdbms-bean`
デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

データベースの特定のカラム用にマップされたフィールド名です。Bean インスタンスの CMP フィールドに対応しています。

例

XML スタンザには、以下の要素を指定できます。

```
<weblogic-rdbms-jar>
        <weblogic-rdbms-bean>

        <field-map>
                <cmp-field>accountId</cmp-field>
                        <dbms-column>id</dbms-column>
        </field-map>

        <field-map>
                <cmp-field>balance</cmp-field>
```

```
                        <dbms-column>bal</dbms-column>
      </field-map>

    <field-map>
              <cmp-field>accountType</cmp-field>
                        <dbms-column>type</dbms-column>
      </field-map>


  </weblogic-rdbms-bean>
</weblogic-rdbms-jar>
```

foreign-key-column

指定できる値：	有効な名前
デフォルト値：	なし
要件：	外部キーのカラムに対応していること。
親要素：	`weblogic-rdbms-bean` `column-map`
デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`foreign-key-column` 要素は、データベース内の外部キーのカラムを表します。

例

10-15 ページの「column-map」を参照してください。

foreign-key-table

指定できる値：	有効な名前
デフォルト値：	なし
要件：	なし

親要素：	`weblogic-rdbms-jar`
	`weblogic-rdbms-relation`
	`weblogic-relationship-role`
	`relationship-role-map`

デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`foreign-key-table` 要素は、外部キーを格納する DBMS テーブル名を指定します。

例

10-46 ページの「relationship-role-map」を参照してください。

generator-name

指定できる値：	なし
デフォルト値：	なし
要件：	省略可能
親要素：	`weblogic-rdbms-bean` `automatic-key-generation`
デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`generator-name` 要素は、ジェネレータの名前を指定する場合に使用します。

次に例を示します。

- `generator-type` 要素が `ORACLE` の場合、`generator-name` 要素は使用される `ORACLE_SEQUENCE` の名前。
- `generator-type` 要素が `NAMED_SEQUENCE_TABLE` の場合、`generator-name` 要素は使用される `SEQUENCE_TABLE` の名前。

例

10-9 ページの「automatic-key-generation」を参照してください。

generator-type

指定できる値：	なし
デフォルト値：	なし
要件：	省略可能
親要素：	`weblogic-rdbms-bean` `automatic-key-generation`
デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

generator-type 要素は、使用するキー生成方法を指定します。オプションは以下のとおりです。

- `ORACLE`
- `SQL_SERVER`
- `NAMED_SEQUENCE_TABLE`

例

10-9 ページの「automatic-key-generation」を参照してください。

group-name

指定できる値：	有効な名前
デフォルト値：	なし
要件：	なし
親要素：	weblogic-rdbms-relation field-group weblogic-rdbms-bean finder finder-query
デプロイメント ファイル：	weblogic-cmp-rdbms-jar.xml

機能

group-name 要素はフィールド グループ名を指定します。

例

XML スタンザには、以下の要素を指定できます。

```
<weblogic-rdbms-jar>
       <weblogic-rdbms-relation>
                     <field-group>employee</field-group>
                            <cmp-field>employee stock
purchases</cmp-field>
                            <cmr-field>stock options</cmr-field>
                                   <group-name>financial
data</group-name>
       </weblogic-rdbms-relation>
```

```
</weblogic-rdbms-jar>
```

include-updates

指定できる値：	`True | False`
デフォルト値：	`False`
要件：	デフォルトは `False` で、この設定は最大限のパフォーマンスを実現する。
親要素：	`weblogic-rdbms-bean` `weblogic-query`
デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

include-updates 要素は、現在のトランザクション中の更新を必ずクエリの結果に反映するように指定します。この要素を `True` に設定した場合、コンテナは現在のトランザクションによる変更をすべてディスクにフラッシュしてからクエリを実行します。

例

`XML` スタンザには、以下の要素を指定できます。

```
<include-updates>False</include_updates>
```

key-cache-size

指定できる値：	なし
デフォルト値：	1
要件：	省略可能
親要素：	`weblogic-rdbms-bean` `automatic-key-generation`
デプロイメント ファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

key-cache-size 要素は、自動主キー生成機能で利用可能な主キー キャッシュのサイズをオプションとして指定します。

例

10-9 ページの「automatic-key-generation」を参照してください。

key-column

指定できる値：	有効な名前
デフォルト値：	なし
要件：	主キーのカラムに対応していること。
親要素：	`weblogic-rdbms-bean` `column-map`
デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`key-column` 要素は、データベース内の主キーのカラムを表します。

例

10-15 ページの「column-map」を参照してください。

max-elements

指定できる値：	なし
デフォルト値：	なし
要件：	なし
親要素：	weblogic-rdbms-bean weblogic-query
デプロイメントファイル：	weblogic-cmp-rdbms-jar.xml

機能

max-elements は多値クエリによって返される要素の最大数を指定します。この要素は、JDBC の maxRows 機能とほぼ同じです。

例

XML スタンザには、以下の要素を指定できます。

```
<max-elements>100</max-elements>
      <!ELEMENT max-element (PCDATA)>
```

method-name

指定できる値：	なし
デフォルト値：	なし
要件：	「*」文字はワイルドカードとして使用できない。
親要素：	`weblogic-rdbms-bean` `query-method`
デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`method-name` 要素は、ファインダ メソッドまたは `ejbSelect` メソッドの名前を指定します。

例

10-56 ページの「weblogic-query」を参照してください。

method-param

指定できる値：	有効な名前
デフォルト値：	なし
要件：	なし
親要素：	weblogic-rdbms-bean method-params
デプロイメント ファイル：	weblogic-cmp-rdbms-jar.xml

機能

method-param 要素には、Java タイプのメソッド パラメータの完全修飾名が含まれます。

例

XML スタンザには、以下の要素を指定できます。

```
<method-param>java.lang.String</method-param>
```

method-params

指定できる値：	有効な名前のリスト
デフォルト値：	なし
要件：	なし
親要素：	`weblogic-rdbms-bean` `query-method`
デプロイメント ファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`method-params` 要素には、Java タイプのメソッド パラメータの完全修飾名の順序付きリストが含まれます。

例

10-56 ページの「weblogic-query」を参照してください。

optimistic-column

指定できる値：	なし
デフォルト値：	なし
要件：	すべてのデータベースが大文字と小文字を区別するわけではないものの、この要素では、大文字と小文字を区別する。
親要素：	`weblogic-rdbms-bean` `table-map`
デプロイメント ファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`optimistic-column` 要素は、オプティミスティックな同時実行性を実装するためのバージョン値またはタイムスタンプ値を格納するデータベース カラムを示します。オプティミスティックな同時実行性の詳細については、4-19 ページの「Optimistic 同時方式」を参照してください。

例

次の XML 例では、`optimistic-column` 要素の使い方を示します。

```
<optimistic-column>ROW_VERSION</optimistic-column>
```

primary-key-table

指定できる値：	なし
デフォルト値：	なし
要件：	すべてのデータベースが大文字と小文字を区別するわけではないものの、この要素では、大文字と小文字を区別する。

親要素：	`weblogic-rdbms-jar`
	`weblogic-rdbms-relation`
	`weblogic-relationship-role`
	`relationship-role-map`

デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`primary-key-table` 要素は、主キーを格納する DBMS テーブル名を指定します。主キーの詳細については、5-33 ページの「主キーの使用」を参照してください。

次の XML スタンザでは、1 対 1 の関係の `primary-key` 側の Bean (`Pk_bean`) は複数のテーブルにマップされていますが、関係の `foreign-key` 側の Bean (`Fk_Bean`) は 1 つのテーブル (`Fk_BeanTable`) にマップされています。foreign-key カラム名は `Fk_column_1` および `Fk_column_2` です。

詳細については、5-44 ページの「コンテナ管理による関係」を参照してください。

例

次の XML 例では、`primary-key-table` 要素の使い方を示します。

```
<relationship-role-map
       <primary-key-table->Pk_BeanTable_1</primary-key-table>
       <column-map>
              <foreign-key-column>Fk_column_1</foreign-key-column>
              <key-column>Pk_table1_pkColumn_1</key-column>
       </column-map>
       <column-map>
              <foreign-key-column>Fk_column_2</foreign-key-column>
              <key-column>Pk_table1_pkColumn_2</key-column>
       </column-map>
</relationship-role-map>
```

query-method

指定できる値：	なし
デフォルト値：	なし
要件：	なし
親要素：	`weblogic-rdbms-bean`
デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`query-method` 要素は、`weblogic-query` に関連付けるメソッドを指定します。`ejb-jar.xml` 記述子と同じ形式を使用します。

例

10-56 ページの「weblogic-query」を参照してください。

relation-name

指定できる値：	有効な名前
デフォルト値：	なし
要件：	関連する ejb-jar.xml デプロイメント記述子ファイルで定義した ejb-relation の ejb-relation-name に一致していること。ejb-relation-name は省略可能な要素だが、関連する ejb-jar.xml デプロイメント記述子ファイルで定義した ejb-relation ごとに必要になる。
親要素：	weblogic-rdbms-relation
デプロイメントファイル：	weblogic-cmp-rdbms-jar.xml

機能

relation-name 要素は関係の名前を指定します。

例

XML スタンザには、以下の要素を指定できます。

```
<weblogic-rdbms-jar>
       <weblogic-rdbms-relation>
              <relation-name>stocks-holders</relation-name>
                     <table-name>stocks</table-name>
       </weblogic-rdbms-relation>
</weblogic-rdbms-jar>
```

relationship-caching

指定できる値：	なし
デフォルト値：	なし
要件：	なし
親要素：	weblogic-rdbms-jar weblogic-rdbms-bean
デプロイメントファイル：	weblogic-cmp-rdbms-jar.xml

機能

relation-caching 要素は、リレーションシップ キャッシングを指定します。リレーションシップ キャッシングの詳細については、5-52 ページの「CMR でのリレーションシップ キャッシングの使用」を参照してください。

例

```
<relationship-caching>
       <caching-name>cacheMoreBeans</caching-name>
       <caching-element>
              <cmr-field>accounts<</cmr-field>
              <group-name>acct_group</group-name>
              <caching-element>
                     <cmr-field>address</cmr-field>
                     <group-name>addr_group</group-name>
              </caching-element>
       </caching-element>

       <caching-element>
              <cmr-field>phone</cmr-field>
              <group-name>phone_group</group-name>
       </caching-element>
</relationship-caching>
```

accounts フィールドと phone フィールドは、customerBean テーブル上のコンテナ管理による関係 (CMR) フィールドです。また、address フィールドは accountBean テーブル上の CMR フィールドです。addr_group および phone_group は、addressBean および phoneBean 内のグループです。

caching-element をネストすることにより、関係する Bean を複数レベルでロードできるようになります。この例では、addressBean が accountBean 内でネストされているので、第 2 レベルの関連 Bean にあたります。現在、指定できる caching-element の数に制限はありません。しかし、caching-element のレベルをあまり多く設定しすぎると、そのトランザクションのパフォーマンスに影響すると考えられます。

リレーションシップ キャッシングでは結合クエリを使用し、結合クエリではおそらく結果を ResultSet 内のテーブルに複写するため、relationship-caching 要素で指定した caching-element デプロイメント記述子の数が ResultSet に複写される結果の数に直に影響してきます。1 対多関係では、relationship-caching 要素であまり多くの caching-element デプロイメント記述子を指定しないでください。caching-element デプロイメント記述子の数に対して複写される結果の数が乗算される可能性があります。

relationship-role-map

デフォルト値：	なし
要件：	関連する ejb-jar.xml 記述子ファイルで定義した ejb-relationship-role の ejb-relationship-role-name に一致していること。
親要素：	weblogic-rdbms-relation weblogic-relationship-role
デプロイメントファイル：	weblogic-cmp-rdbms-jar.xml

機能

relationship-role-map 要素は、ある関係に加わっている Bean のキーカラムマッピングに対する 外部キー カラムを指定します。

詳細については、5-44 ページの「コンテナ管理による関係」を参照してください。

例

XML スタンザには、以下の要素を指定できます。

```
<relationship-role-map
       <foreign-key-table>Fk_BeanTable_2</foreign-key-table>
       <column-map>
              <foreign-key-column>Fk_column_1</foreign-key-column>
              <key-column>Pk_table_pkColumn_1</key-column>
       </column-map>
       <column-map>
              <foreign-key-column>Fk_column_2</foreign-key-column>
              <key-column>Pk_table_pkColumn_2</key-column>
       </column-map>
</relationship-role-map>
```

relationship-role-name

指定できる値：	有効な名前
要件：	関連する ejb-jar.xml 記述子ファイルで定義した ejb-relationship-role の ejb-relationship-role-name に一致していること。
親要素：	weblogic-rdbms-relation weblogic-relationship-role
デプロイメントファイル：	weblogic-cmp-rdbms-jar.xml

機能

relationship-role-name 要素は関係のロール名を指定します。

例

XML スタンザには、以下の要素を指定できます。

```
<relationship-role-name></relationship-role-name>
```

sql-select-distinct

指定できる値：	`True	False`
デフォルト値：	`False`
要件：	Oracle データベースでは、SELECT DISTINCT を FOR UPDATE 句と一緒に使用できない。したがって、`transaction-isolation` 要素に `isolation-level` 下位要素を設定し、その下位要素の値を `TransactionReadCommittedForUpdate` に設定したメソッドを、呼び出しチェーンの Bean が備えている場合、`sql-select-distinct` 要素を使用することはできない。`transaction-isolation` 要素は `weblogic-ejb-jar.xml` ファイルで定義する。
親要素：	`weblogic-query`
デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`sql-select-distinct` 要素は、生成される SQL SELECT 文に DISTINCT 修飾子が含まれるかどうかを指定します。DISTINCT 修飾子を使用すると、データベースからユニークな行が返されます。

例

この要素を含む `XML` の例を示します。

```
<sql-select-distinct>True</sql-select-distinct>
```

table-map

指定できる値：	なし
デフォルト値：	なし
要件：	各 table-map 要素に、その Bean の主キー フィールドのマッピングが含まれること。
親要素：	weblogic-rdbms-bean
デプロイメント ファイル：	weblogic-cmp-rdbms-jar.xml

機能

table-map 要素は、ある 1 つの Bean の cmp-field と 1 つのテーブルのカラムの間のマッピングを指定するものであり、そのテーブルにマッピングするすべての cmp-field を扱います。1 つの CMP Bean を n 個の DBMS テーブルにマップする場合、その Bean の n 個の table-map 要素を、n 個の DBMS テーブルごとに 1 つずつ指定しなくてはなりません。

1 つの CMP Bean を複数のテーブルにマップする場合、各テーブルには、1 つの特定の Bean インスタンスにマップする 1 行が含まれます。そのため、すべてのテーブルにはどの時点でも同数の行が含まれます。また、各テーブルには均一の主キーの値の並びが含まれます。したがって、各テーブルには均一の主キー カラムが含まれ、別々のテーブルの対応している主キー カラムどうしは名前は違うとしても同じタイプでなければなりません。

table-map 要素は、特定テーブルの主キー カラムから、Bean の主キー カラム フィールドへのマッピングを指定しなければなりません。主キー以外のフィールドは 1 つのテーブルにマップできるのみです。

例

XML スタンザには、以下の要素を指定できます。

```
<table-map>
       <table-nme>DeptTable</table-name>

       <field-map>
              <cmp-field>deptId1</cmp-field>
              <dbms-column>t1_deptId1_column</dbms-column>
       </field-map>
       <field-map>
              <cmp-field>deptId2</cmp-field>
              <dbms-column>t1_deptId2_column</dbms-column>
       </field-map>
       <field-map>
              <cmp-field>location</cmp-field>
              <dbms-column>location_column</dbms-column>
       </field-map>
              <cmp-field>budget</cmp-field>
              <dbms-column>budget</dbms-column>
       </field-map>
       <fieldmap
</table-map>
```

table-name

指定できる値：	データベース内にあるソース テーブルの有効な完全修飾 SQL 名
デフォルト値：	なし
要件：	`table-name` を必ず設定すること。
親要素：	`weblogic-rdbms-bean` `weblogic-rdbms-relation`
デプロイメント ファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

テーブルの完全修飾 SQL 名です。この Bean の `data-source` 用に定義したユーザには、指定したテーブルの読み取りおよび書き込み特権が必要ですが、スキーマ変更特権は必要ありません。

例

`XML` スタンザには、以下の要素を指定できます。

```
<weblogic-rdbms.jar>
  <weblogic-rdbms-bean>
    <ejb-name>containerManaged</ejb-name>

<data-source-name>examples-dataSource-demoPool</data-source-name>
    <table-name>ejbAccounts</table-name>


</weblogic-rdbms-bean>
</weblogic-rdbms-jar>
```

use-select-for-update

指定できる値：	True | False
デフォルト値：	False
要件：	なし
親要素：	weblogic-rdbms-bean
デプロイメント ファイル：	weblogic-cmp-rdbms-jar.xml

機能

Bean 単位でペシミスティックな同時方式を強制します。このフラグで True を指定すると、Bean がデータベースからロードされるときは常に SELECT ... FOR UPDATE が使用されます。これは、トランザクション レベルではなく Bean レベルで設定されるという点で、トランザクションのアイソレーション レベル TRANSACTION_READ_COMMITTED_FOR_UPDATE とは異なります。

例

XML スタンザには、以下の要素を指定できます。

```
<weblogic-rdbms.jar>
  <weblogic-rdbms-bean>
    <ejb-name>containerManaged</ejb-name>
    <use-select-for-update>true</use-select-for-update>
</weblogic-rdbms-bean>
</weblogic-rdbms-jar>
```

validate-db-schema-with

指定できる値：	MetaData | TableQuery
デフォルト値：	TableQuery
要件：	なし
親要素：	weblogic-rdbms-jar
デプロイメント ファイル：	weblogic-cmp-rdbms-jar.xml

機能

validate-db-schema-with 要素を指定すると、デプロイメント時に Bean が有効なデータベース スキーマにマップされているかどうかをコンテナ管理による永続性でチェックします。

MetaData を指定すると、WebLogic Server は、JDBC メタデータを使用してスキーマを検証します。

TableQuery (デフォルト設定) を指定すると、WebLogic Server は、テーブルを直接クエリし、CMP 実行時によって予期されているスキーマがテーブルにあるかどうかを確認します。

例

XML スタンザには、以下の要素を指定できます。

```
<validate-db-schema-with>TableQuery</validate-db-schema-with>
```

verify-columns

指定できる値：	Read | Modified | Version | Timestamp
デフォルト値：	なし
要件：	table-name を必ず設定すること。
親要素：	weblogic-rdbms-bean table-map
デプロイメントファイル：	weblogic-cmp-rdbms-jar.xml

機能

Optimistic 同時方式を使用する場合は、verify-columns 要素によって有効性をチェックするテーブル カラムが指定されます。WebLogic Server は、トランザクションの終了時、データベースにコミットする前にカラムをチェックし、他のどのトランザクションもそのデータを変更していないことを確かめます。

- Read ー トランザクションの間に読み込まれたテーブルのすべてのカラムがチェックされます。
- Modified ー 現在のトランザクションで更新されたカラムだけがチェックされます。
- Version および Timestamp ー テーブルにバージョンまたはタイムスタンプ (疑似) カラムが含まれ、このカラムがオプティミスティックな同時方式を実施するのに使われるように指定します。EJB コンテナは、テーブルの行が更新されるとバージョンまたはタイムスタンプ カラムを自動的にインクリメントします。バージョン (またはタイムスタンプ) カラムの名前は、optimistic-column 要素で指定します。それを希望する場合を除いて、このカラムを cmp フィールドにマップする必要はありません。

Bean が複数のテーブルにマップされる場合、トランザクション中に更新したテーブルに対してのみチェックが実行されます。各テーブルの verify-columns 要素は、値が同じである必要はありません。

詳細については、4-19 ページの「Optimistic 同時方式」を参照してください。

例

XML スタンザには、以下の要素を指定できます。

```
<verify-columns>Modified</verify-columns>
```

weblogic-ql

指定できる値：	なし
デフォルト値：	なし
要件：	なし
親要素：	weblogic-rdbms-bean weblogic-query
デプロイメントファイル：	weblogic-cmp-rdbms-jar.xml

機能

weblogic-ql 要素は、EJB-QL に対する WebLogic 固有の拡張機能を含むクエリを指定します。ejb-jar.xml デプロイメント記述子では、EJB-QL 言語の標準機能だけを使用するクエリを指定しなければなりません。

例

10-56 ページの「weblogic-query」を参照してください。

weblogic-query

指定できる値：	なし
デフォルト値：	なし
要件：	なし
親要素：	`weblogic-rdbms-bean`
デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`weblogic-query` 要素を使用すると、必要に応じて、WebLogic 固有の属性をクエリに関連付けることができます。たとえば、WebLogic 固有の EJB-QL に対する拡張機能を含むクエリを指定するために使用できます。EJB-QL に対する WebLogic の拡張機能を使用しないクエリは、`ejb-jar.xml` デプロイメント記述子で指定する必要があります。

また、クエリによってあらかじめキャッシュにロードしておく必要があるエンティティ Bean をクエリで取得する場合は、`weblogic-query` 要素を使用して、`field-group` をクエリに関連付けます。

例

`XML` スタンザには、以下の要素を指定できます。

```
<weblogic-query>
        <query-method>
          <method-name>findBigAccounts</method-name>
          <method-params>
```

```
                                        <method-param>double</method-param>
            </method-params>
          <query-method>
                                    <weblogic-ql>WHERE BALANCE>10000
ORDERBY NAME</weblogic-ql>
               </weblogic-query>
```

weblogic-rdbms-bean

指定できる値：	なし
デフォルト値：	なし
要件：	なし
親要素：	weblogic-rdbms-jar
デプロイメント ファイル：	weblogic-cmp-rdbms-jar.xml

機能

weblogic-rdbms-bean は、WebLogic RDBMS CMP 永続性タイプによって管理された 1 つのエンティティ Bean を表します。

例

weblogic-rdbms-bean の XML 構造を示します。

```
weblogic-rdbms-bean
       ejb-name
       data-source-name
              table-map
       field-group
       relationship-caching
```

```
weblogic-query
delay-database-insert-until
automatic-key-generation
check-exists-on-method
```

weblogic-rdbms-jar

指定できる値：	なし
デフォルト値：	なし
要件：	なし
親要素：	なし
デプロイメントファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`weblogic-rdbms-jar` 要素は、WebLogic RDMBS CMP デプロイメント記述子の最上位の要素です。この要素には、1 つまたは複数のエンティティ Bean と、一連の関係 (これについては省略可能) についてのデプロイメント情報が含まれます。

例

weblogic-rdbms-jar の `XML` 構造を示します。

```
weblogic-rdbms-jar
       weblogic-rdbms-bean
       weblogic-rdbms-relation
       create-default-dbms-tables
       validate-db-schema-with
       database-type
```

weblogic-rdbms-relation

指定できる値：	なし
デフォルト値：	なし
要件：	なし
親要素：	`weblogic-rdbms-jar`
デプロイメント ファイル：	`weblogic-cmp-rdbms-jar.xml`

機能

`weblogic-rdbms-relation` 要素は、1 つのコンテナ管理による関係 (CMR) を表します。

CMR の詳細については、5-44 ページの「コンテナ管理による関係」を参照してください。

例

1 対 1 の関係を示す `weblogic-rdbms-relation` の `XML` 構造は、次のとおりです。

```
<weblogic-rdbms-relation>
   <relation-name>...</relation-name>
   <weblogic-relationship-role>...</weblogic-relationship-role>
</weblogic-rdbms-relation>
```

weblogic-relationship-role

指定できる値：	有効な名前
デフォルト値：	なし
要件：	ロールのテーブルへのマッピングを、関連する weblogic-rdbms-bean および ejb-relation 要素で指定すること。
親要素：	weblogic-rdbms-jar weblogic-rdbms-relation
デプロイメントファイル：	weblogic-cmp-rdbms-jar.xml

機能

weblogic-relationship-role 要素は、外部キーから主キーへのマッピングを表すのに使用します。1 対 1 または 1 対多の関係では、1 つのマッピングのみを指定します。多対多の関係では、2 つのマッピングを指定する必要があります。

キーが複数の場合、複数カラムのマッピングは単独のロールに対して指定します。ロールが group-name を指定しているだけの場合、column-map は指定されません。

詳細については、5-44 ページの「コンテナ管理による関係」を参照してください。

例

XML スタンザには、以下の要素を指定できます。

```
<weblogic-relationship-role>
   <relationship-role-name>...</relationhsip-role-name>
   <relationship-role-map>
                    <<column-map>
                           <foreign-key-column>manager-id
```

```
                                    </foreign-key-column>
                                <key-column>id</key-column>
                            </column-map>
                        <relationship-role-name>
                    </weblogic-relationship-role>
```

1.1 の weblogic-cmp-rdbms-jar.xml デプロイメント記述子ファイルの構造

weblogic-cmp-rdbms-jar.xml では、WebLogic Server RDBMS ベースの永続性サービスを使用するエンティティ EJB のデプロイメント要素を定義します。

WebLogic Server 1.1 の weblogic-cmp-rdbms-jar.xml の最上位要素は、weblogic-enterprise-bean スタンザで構成されます。

```
description
weblogic-version
<weblogic-enterprise-bean>
   <pool-name>finance_pool</pool-name>
   <schema-name>FINANCE_APP</schema-name>
   <table-name>ACCOUNT</table-name>
   <attribute-map>
      <object-link>
         <bean-field>accountID</bean-field>
         <dbms-column>ACCOUNT_NUMBER</dbms-column>
      </object-link>
      <object-link>
         <bean-field>balance</bean-field>
         <dbms-column>BALANCE</dbms-column>
      </object-link>
   </attribute-map>
   <finder-list>
```

```
        <finder>
           <method-name>findBigAccounts</method-name>
           <method-params>
              <<method-param>double</method-param>
           </method-params>
        <finder-query><![CDATA[(> balance $0)]]></finder-query>
        <finder-expression>. . .</finder-expression>
        </finder>
    </finder-list>
</weblogic-enterprise-bean>
```

1.1 の weblogic-cmp-rdbms-jar.xml デプロイメント記述子要素

RDBMS 定義要素

この節では RDBMS 定義要素について説明します。

pool-name

pool-name では、EJB のデータベース接続に使用する WebLogic Server 接続プールの名前を指定します。詳細については、『WebLogic JDBC プログラマーズ ガイド』を参照してください。

schema-name

schema-name では、データベースに置かれるソース テーブルのスキーマを指定します。この要素は、EJB の接続プールで定義されたユーザに対してデフォルトスキーマではないスキーマを使用する場合にのみ必須です。

注意： 多くの SQL 実装では大文字小文字が無視されますが、このフィールドは大文字小文字が区別されます。

table-name

`table-name` では、データベース内のソース テーブルを指定します。この要素はすべての場合に必須です。

注意： EJB の接続プールで定義されたユーザには、指定されたテーブルに対する読み書き特権が必要です。ただし、スキーマ変更権限は必ずしも必要ありません。多くの SQL 実装では大文字小文字が無視されますが、このフィールドは大文字小文字が区別されます。

EJB フィールド マッピング要素

この節では EJB フォールド マッピング要素について説明します。

attribute-map

`attribute-map` スタンザでは、EJB インスタンスの単一フィールドがデータベース テーブル内の特定のカラムにリンクされます。`attribute-map` には、WebLogic Server RDBMS ベースの永続性を使用する EJB のフィールドごとに 1 つのエントリが必要です。

object-link

各 `attribute-map` エントリは、データベース内のカラムと EJB インスタンス内のフィールドとのリンクを表す `object-link` スタンザから構成されます。

bean-field

`bean-field` では、データベースからの移行が必要な EJB インスタンスのフィールドを指定します。この要素では大文字小文字が区別されます。この要素は、Bean インスタンスのフィールド名に正確に一致している必要があります。

また、このタグで参照されるフィールドは、Bean の ejb-jar.xml ファイル内に定義されている cmp-field 要素も持っている必要があります。

dbms-column

dbms-column では、EJB フィールドがマップされるデータベース カラムを指定します。多くのデータベースでは大文字小文字が無視されますが、このフィールドでは大文字小文字が区別されます。

注意： WebLogic Server では、引用符で囲まれた RDBMS キーワードは dbms-column のエントリとしてはサポートされていません。たとえば、基になるデータストアで「create」や「select」が予約語になっている場合、それらをカラム名にして属性マップを作成することができません。

ファインダ要素

この節ではファインダ要素について説明します。

finder-list

finder-list スタンザでは、Bean の集合を見つけるために生成されるすべてのファインダの集合を定義します。詳細については、5-5 ページの「EJB 1.1 CMP の RDBMS 永続性用の記述」を参照してください。

findByPrimarykey の場合を除いて、finder-list には、ホーム インタフェース内に定義されているファインダ メソッドごとに 1 つのエントリが必要です。findByPrimaryKey の場合は、エントリを指定しなくても、コンパイル時に finder-list が生成されます。

注意： findByPrimaryKey に対してエントリを指定すると、WebLogic Server では、正当性が検証されずにそのエントリが使用されます。ほとんどの場合では、findByPrimaryKey に対するエントリを定義せずに、デフォルトで生成されるメソッドを受け入れることをお勧めします。

finder

finder スタンザでは、ホーム インタフェース内に定義されるファインダ メソッドを記述します。finder スタンザに含まれる要素によって、WebLogic Server では、ホーム インタフェース内に記述されているメソッドが識別され、必要なデータベース操作を実行できるようになります。

method-name

method-name では、ホーム インタフェース内のファインダ メソッドの名前を定義します。このタグには、メソッドの正確な名前を指定する必要があります。

method-params

method-params スタンザでは、method-name で指定されるファインダ メソッドに対するパラメータのリストを定義します。

注意： WebLogic Server では、このリストは EJB のホーム インタフェース内のファインダ メソッドのパラメータ タイプと比較されます。パラメータ リストの順序とパラメータ タイプは、ホーム インタフェースで定義されている順序とパラメータ タイプに正確に一致している必要があります。

method-param

method-param では、パラメータ タイプの完全修飾名を定義します。このタイプ名は java.lang.Class オブジェクトで評価され、その結果のオブジェクトは EJB のファインダ メソッド内の各パラメータに正確に一致している必要があります。

「double」や「int」のようなプリミティブ名を使用してプリミティブ パラメータを指定できます。method-param 要素内に非プリミティブなタイプを使用する場合には、完全修飾名を指定する必要があります。たとえば、Timestamp ではなく java.sql.Timestamp を使用します。完全修飾名を使用しないと、デプロイメント ユニットのコンパイル時に ejbc でエラー メッセージが生成されます。

finder-query

`finder-query` では、このファインダ用にデータベースから値を取り出す場合に使用する WebLogic クエリ言語 (WLQL) 文字列を指定します。詳細については、5-7 ページの「EJB 1.1 CMP 用の WebLogic クエリ言語 (WLQL) の使用」を参照してください。

注意： `finder-query` 値のテキストは、常に、XML `CDATA` 属性を使用して定義してください。`CDATA` を使用すると、WLQL 文字列中に特殊文字が入っていても、ファインダをコンパイルしたときにエラーが発生しないようになります。

finder-expression

`finder-expression` では、このファインダ用のデータベース クエリ中で変数として使用される Java 言語の式を指定します。

注意： WebLogic Server EJB コンテナの将来のバージョンでは、EJB QL クエリ言語が使用される予定です (このクエリ言語は EJB 2.0 仕様では必須です)。EJB QL では、埋め込まれた Java 式はサポートされていません。そのため、将来の EJB コンテナに簡単にアップグレードできるよう、WLQL でも Java 式を埋め込まずにエンティティ EJB ファインダを作成してください。

索引

F

G

I

J

N

O

R

S

T

U

W

あ

い

え

お

か

き

く

け

こ

さ

し

と

は

ひ

ふ

へ

ほ

ま

め

も

ゆ

よ

り

れ

ろ