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 6.1
マニュアルの日付：2002 年 6 月 24 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	2002 年 6 月 24 日	BEA WebLogic Server 6.1

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 の使い方

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

9. WebLogic Server EJB のユーティリティ

10. weblogic-ejb-jar.xml 文書型定義

11. 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 の使い方」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 8 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 9 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 10 章「weblogic-ejb-jar.xml 文書型定義」では、WebLogic Server 6.1 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 11 章「weblogic-cmp-rdbms-jar.xml 文書型定義」では、WebLogic Server 6.1 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を一度に 1 ファイルずつ印刷できます。

このマニュアルの PDF 版は、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、第 10 章「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 11 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
  - 『BEA WebLogic RMI プログラマーズ ガイド』
  - 『BEA WebLogic RMI over IIOP プログラマーズ ガイド』

サポート情報

BEA 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@bea.com** までお送りください。寄せられた意見については、WebLogic Server の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 6.1 には、Sun Microsystems の EJB 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 6.1 は Sun の J2EE 仕様および EJB 1.1 仕様に準拠しています。また、準備段階の EJB 2.0 仕様の実装を含んでいます。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。このバージョンの WebLogic Server では既存の EJB 1.1 Bean をデプロイすることもできますが、新しい Bean を開発する場合は、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 6.1 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

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、コンテナでも Bean でも管理できますが、使用するのはコンテナ管理のトランザクション境界定義だけです。
- **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。

準備段階の仕様の実装

以降の節では、最終確定されていない Java 仕様に沿って WebLogic Server を使用する方法を説明します。

準備段階の J2EE 仕様

WebLogic Server 6.1 は、次の特徴を備えた 2 種類のバージョンのいずれかを使用できます。

- J2EE 1.2 機能に加えて高度な J2EE 1.3 機能に対応
- J2EE 1.2 仕様に完全準拠した実装である J2EE 1.2 機能のみに対応

どちらのバージョンも、J2EE に適用される規則に準拠します。いずれのバージョンも同じコンテナを提供しますが、利用可能な API だけ異なります。

準備段階の EJB 2.0 仕様

WebLogic Server では、エンタープライズ JavaBean 2.0 の実装がフルサポートされ、それをプロダクション段階で使用することもできます。ただし、Sun Microsystems EJB 2.0 仕様はまだ最終的なものではありません。EJB 2.0 アーキテクチャの WebLogic Server 実装は、公開されている最新の草稿に基づいています。したがって、最終の仕様が公開されたら、WebLogic Server の将来のバージョンでエンタープライズ JavaBean 2.0 の実装に変更があるかもしれません。その場合、WebLogic Server 6.1 向けに開発されたアプリケーション コードは、将来のリリースでサポートされる EJB 2.0 の実装と互換性を持たなくなる可能性があります。

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、9-4 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 の使い方」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 10 章「weblogic-ejb-jar.xml 文書型定義」と第 11 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 8 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 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。

read-only マルチキャスト無効化のサポート

トランザクション非対応のエンティティ Bean を使用すると、キャッシュ データを無効化または更新する場合に便利です。ユーザは、ホーム インタフェースの invalidate メソッドを利用して、read-only エンティティ Bean を無効にできます。トランザクション非対応のエンティティ Bean のキャッシングについては、4-17 ページの「read-only マルチキャストの無効化」を参照してください。

主キーの自動生成のサポート

WebLogic Server EJB コンテナは、自動的に生成される主キーを提供できます。この機能では、Oracle または SQLServer に用意されているネイティブの自動キー生成機能を使用できます。これらのデータベースを使用していない場合には、ユーザ指定のキー テーブルを通じてキーを生成できます。主キーの自動生成の詳細については、5-22 ページの「EJB 2.0 CMP に対する自動主キー生成」を参照してください。

自動テーブル作成

テーブルが作成されていない場合には、デプロイメント ファイルおよび Bean クラスのデプロイメント記述子に基づいて、テーブルを自動的に作成することができます。この機能は開発段階で使用するためのもので、プロダクション レベルのサポートを提供しません。ただし、プロダクション環境にデプロイする前の設計テストでは非常に便利な機能です。自動テーブル作成の詳細については、5-26 ページの「自動テーブル作成」を参照してください。

Oracle SELECT HINT

WebLogic QL クエリの 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ができます。これらのクエリにより、Oracle データベース エンジンにヒントが提供されます。この機能は、検索先のデータベースがこれらのヒントから恩恵を受けることがわかっている場合に非常に有効です。Oracle SELECT HINT の詳細については、5-13 ページの「Oracle の SELECT HINT の使用」を参照してください。

EJB デプロイメント記述子エディタ

EJB デプロイメント記述子エディタは、グラフィカルな環境で EJB 用のデプロイメント記述子を編集するための WebLogic Server Administration Console の拡張機能です。このエディタの詳細については、6-6 ページの「EJB デプロイメント記述子の指定と編集」と Administration Console のオンライン ヘルプを参照してください。

ejb-client.jar サポート

クライアントを 1 つの JAR ファイルにコンパイルするために必要なすべてのクラスをパッケージ化するには、`ejb-client.jar` ファイル を使用します。`ejb-client.jar` は、EJB を呼び出すために必要な EJB インタフェースを格納します。WebLogic EJB コンパイラ（`weblogic.ejbc`）が自動的に `ejb-client.jar` ファイルを作成するよう、デプロイメント記述子ファイルに指定します。`ejb-client.jar` ファイルの詳細については、6-15 ページの「ejb-client.jar の指定」を参照してください。

BLOB および CLOB のサポート

Oracle で大きなオブジェクトをバイト配列または文字列に変換するには、BLOB または CLOB を使用します。BLOB および CLOB の詳細については、5-14 ページの「Oracle DBMS の BLOB および CLOB DBMS カラムのサポート」を参照してください。

カスケード削除のサポート

カスケード削除機能を使用すると、エンティティ オブジェクトを削除できます。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。カスケード削除の詳細については、5-16 ページの「カスケード削除」を参照してください。

ローカル インタフェースのサポート

WebLogic Server の EJB コンテナは、ローカル インタフェースのサポートを提供します。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。このリリースでもコンテナ管理による永続性（CMP）関係でリモート インタフェースを使用できますが、新しく開発する場合には使用しないことをお勧めします。ローカル インタフェースのサポートの詳細については、5-31 ページの「ローカル クライアントの使用」を参照してください。

CMP キャッシュのフラッシュ機能のサポート

変更が結果に反映されるようにクエリの前にコンテナ管理による永続性（CMP）キャッシュをフラッシュするよう指定できます。この機能の詳細については、5-19 ページの「CMP キャッシュのフラッシュ」を参照してください。

CMP 1.1 のチューニングのサポート

このリリースでは、EJB コンテナはコンテナ管理による永続性（CMP）1.1 エンティティ Bean の更新の調整をサポートしています。EJB コンテナは、トランザクションで変更されたコンテナ管理によるフィールドだけを自動的に判別し、データベースに書き込むことができます。フィールドが変更されていない場合には、データベースは更新されません。この機能はデフォルトで有効になっており、無効にすることもできますが、パフォーマンスを考慮して有効にしておくことをお勧めします。CMP のチューニング サポートの詳細については、5-18 ページの「WebLogic Server での EJB 1.1 CMP の調整更新」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、「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。

EJB デプロイメント記述子エディタ

WebLogic Server の 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、「Web アプリケーション デプロイメント記述子エディタ ヘルプ」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、BEA dev2dev 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 JavaBeans（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
- 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、モデルかに 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 の使い方」を参照してください。

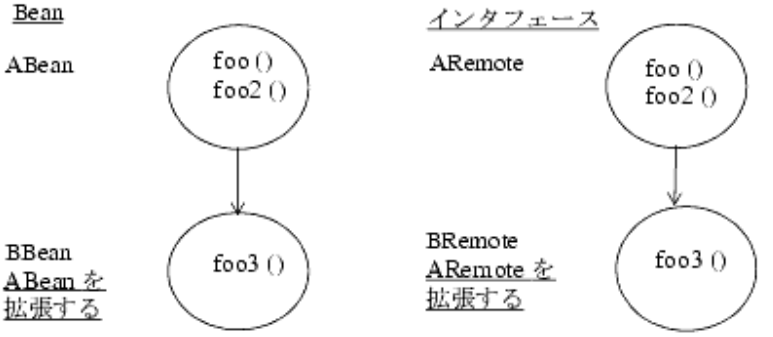
EJB での継承の使用

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。

コード リスト 2-1 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);
   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、4-39 ページの「排他的ロック サービス」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。[サーバ] ノードをクリックします。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 ではその IP アドレスが EJB の HomeHandle オブジェクトに格納されます。このことは、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. 右ペインでコンフィグレーション情報を確認します。
5. [チューニング] タブを選択します。
6. [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、4-29 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。

コンテナ管理の EJB には常にトランザクション データソースを使用する

コンテナ管理の EJB で使用するために JDBC データソース ファクトリをコンフィグレーションする場合は必ず、非トランザクション データソース（`DataSource`）ではなくトランザクション データソース（`TXDataSource`）をコンフィグレーションします。非トランザクション データソースを使用すると、JDBC 接続は自動コミット モードで動作し、データベースに対する毎回の挿入および更新操作は、コンテナ管理トランザクションの一部として扱われず直ちにコミットされます。

3　メッセージ駆動型 Bean の使い方

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 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 Java Messaging Service（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。

注意： メッセージ駆動型 Bean は EJB 2.0 の機能です。

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。メッセージ駆動型 Bean はデプロイメント時にトピックやキューなどの JMS 送り先と関連付けられ、受信メッセージを処理するために必要になった時点で、WebLogic Server によっ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、4-29 ページの「トランザクション管理」と 3-14 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（3-14 ページの「WebLogic Server でのメッセージ駆動型 Bean のデプロイ」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ます。
- 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 インタフェースがありません。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ます。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： 複数のメッセージを一度に受信するために、`max-beans-in-free-pool` デプロイメント記述子を使って MDB の最大数をコンフィグレーションした場合、MDB の数は実行スレッドの最大数を超えることはできません。たとえば、`max-beans-in-free-pool` を 50 に設定しても、許容される実行スレッドの最大数が 25 の場合は、実際にメッセージを受信する MDB は 25 だけです。

`max-beans-in-free-pool` の詳細については、10-47 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 Bean 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` インタフェースを両方とも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。
- 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 3-12 ページの「メッセージ駆動型 Bean の呼び出し」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. 8-9 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。

コード リスト 3-1　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 Bean の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。

JMS 送り先に対するプリンシパルの指定とパーミッションの設定

メッセージ駆動型 Bean は、run-as プリンシパルを使用して JMS 送り先に接続します。run-as プリンシパルは、ejb-jar.xml ファイルで設定する run-as 要素に対応します。この設定では、メッセージ駆動型 Bean のメソッドの実行に使われる run-as ID を指定します。WebLogic Server コンテナにメッセージ駆動型 Bean をデプロイすると、Bean は JMS の送り先と関連付けられます。JMS の送り先はキューまたはトピックのどちらかです。JMS の送り先は、メッセージ駆動型 Bean の ejb-jar.xml ファイルで、jms-destination-type 要素に値 queue または topic を設定することによって指定します。

メッセージ駆動型 Bean を JMS の送り先に接続するときは、後で説明するように、Bean の `run-as` プリンシパルに対するパーミッションを `receive` に設定します。これによりメッセージ駆動型 Bean は、同じドメイン内のリモート キューに接続できるようになり、ほかのドメイン内で同じプリンシパルが定義されていれば別のドメイン内のリモート キューにも接続できます。`run-as` プリンシパルを指定しない場合、WebLogic Server はデフォルトの `guest` ユーザを使用します。ただし、`run-as` プリンシパルと `guest` のどちらを使用する場合でも、セキュリティ プリンシパルに `receive` パーミッションを割り当てる必要があります。

`receive` パーミッションを設定するには、まず新しいアクセス制御リスト（ACL）を作成するか、または既存のリストを修正する必要があります。ACL はリソースにアクセスするためのパーミッションを持つユーザおよびグループのリストです。パーミッションはリソースにアクセスするために必要な権限で、ファイルの読み取り、書き込み、送信、および受信、サーブレットのロード、ライブラリへのリンクなどを行うためのパーミッションがあります。

注意： JMS の送り先に接続するメッセージ駆動型 Bean に対しては、`system` ユーザを使用しないでください。`system` を使用すると、別のドメイン内にある送り先にメッセージ駆動型 Bean が接続できません。

セキュリティ プリンシパル ユーザの詳細については、「ユーザの定義」を参照してください。

ACL を作成し、プリンシパルを指定し、パーミッションを設定するには、次の手順に従います。

1. WebLogic Server Administration Console を起動します。
2. Administration Console の左ペインで、[セキュリティ] → [ACL] ノードに移動します。
3. Administration Console の右ペインで、[新しい ACL の作成] リンクをクリックします。

   [ACL コンフィグレーション] ウィンドウが表示されます。
4. [新しい ACL 名] フィールドに、ACL を使用して保護する WebLogic Server リソースの名前を指定します。

   たとえば、`topic` という名前の JMS 送り先用の ACL を作成します。
5. [作成] をクリックします。

6. [新しい Permission を追加] リンクをクリックします。
7. JMS 送り先リソース `topic` に対して `receive` パーミッションを指定します。
8. リソースに対して指定されたパーミッションを持つユーザとして `run-as-principal` ユーザを指定します。
9. [適用] をクリックします。

恒久サブスクライバとしてのメッセージ駆動型 Bean の指定

メッセージ駆動型 Bean をトピックに関連付ける場合、トピックを恒久的なものとして指定できます。トピックの恒久サブスクリプションは、サーバが動作していない場合でもメッセージが失われないことを保証します。サーバが切断された場合でも、恒久的なトピックが引き続きメッセージを受信して格納し、サーバは再起動されるとメッセージの受信を再開します。メッセージ駆動型 Bean をトピックに関連付ける一方でトピックを恒久的なものとして指定しない場合、デフォルトでトピックは非恒久的となります。

メッセージ駆動型 Bean を恒久サブスクライバとして設定するには、以下のデプロイメント記述子要素を指定します。

1. Bean の `ejb-jar.xml` ファイルの `message-driven-destination` 要素を設定して、Bean の用途をトピックまたはキューのどちらにするかを指定します。
2. 関連付けられたトピックを恒久的にするかどうかを指定するときは、Bean の `ejb-jar.xml` ファイルの `subscription-durability` サブ要素を設定します。
3. Bean の `weblogic-ejb-jar.xml` ファイルの `jms-client-id` 要素を設定します。

   XML ファイルの要素を指定する方法については、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。

注意 : 標準の JMS リスナの代わりにメッセージ駆動型 Bean を使用してメッセージを処理する場合、個々のメッセージ駆動型 Bean は 1 つのトピックだけに関連付けることをお勧めします。アプリケーションにおいて、単一の JMS コンシューマによって複数のトピックまたはキューからのメッ

セージを処理しなければならない場合、標準の JMS コンシューマを使用するか、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。

JMS サーバまたは外部サービス プロバイダへの再接続

メッセージ駆動型 Bean は、クラスタ化されていない WebLogic Server インスタンスにデプロイされた JMS サーバ上、または外部のサービス プロバイダ上にある関連付けられた JMS 送り先をリスンします。サーバが停止したために送り先への接続が失われると、メッセージ駆動型 Bean はその送り先への再接続を定期的に試みます。送り先への再接続を試みる間隔の秒数は、Bean の `weblogic-ejb-jar.xml` ファイルの `jms-polling-interval-seconds` 要素を設定することで指定できます。

XML ファイルを指定する方法については、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。

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。`max-beans-in-free-pool` 属性が `maxBeans` (最大設定) に達したためフリー プールに使用可能な Bean インスタンスがない場合、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、10-47 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。Bean では、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。

   3-9 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、8-2 ページの「WebLogic Server 起動時の EJB のデプロイメント」または 8-3 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。

クライアントは呼び出しに対するトランザクション コンテキストをメッセージ駆動型 Bean に提供しないので、ejb-jar.xml ファイルの container-transaction 要素に対して指定されている trans-attribute の値 (Required または NotSupported) に従って、コンテナ管理のトランザクションを使用する Bean を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。

コード リスト 3-2 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ります。

- 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ります。

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。

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
- WebLogic Server における EJB のライフサイクル
- ステートレス セッション Bean と BMP EJB のパフォーマンス比較
- max-beans-in-free-pool の使用
- エンティティ EJB に対する ejbLoad() と ejbStore() の動作
- エンティティ EJB の read-only への設定
- WebLogic Server クラスタにおける EJB
- トランザクション管理
- リソース ファクトリ
- エンティティ EJB のロック サービス

EJB コンテナ

EJB コンテナは、WebLogic Server の起動時に自動的に作成されるデプロイ済み EJB を収容する実行時コンテナです。エンティティ オブジェクトのライフサイクル全体（作成から削除まで）を通じて、オブジェクトは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、コンテナ内の全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エンティティ 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 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 JNDI を通じてエンティティ 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。

WebLogic Server における EJB のライフサイクル

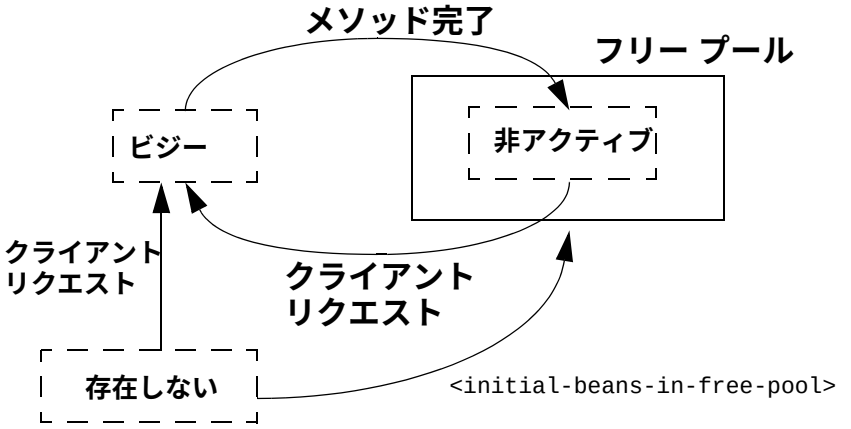
以降の節では、コンテナによるキャッシュ サービスのサポートに関する情報を示します。また、WebLogic Server での EJB インスタンスのライフサイクルを、WebLogic Server の視点から説明します。これらの節では EJB インスタンスという用語を使用しますが、これは EJB クラスの実際のインスタンスという意味です。EJB インスタンスは、クライアント側から見た EJB の論理インスタンスを意味するものではありません。

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 とメッセージ駆動型 Bean のパフォーマンスおよび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 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 4-1 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 インスタンスの初期化

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 クラス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。

`weblogic-ejb-jar.xml` の `initial-beans-in-free-pool` 要素をオプションで設定すると、起動時に非アクティブな EJB インスタンスをフリー プールに自動的に作成できます。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、使用可能メモリ、または `max-beans-in-free-pool` デプロイメント要素の値のいずれかによって制限されます。

ステートレス セッション EJB のアクティブ化とプーリング

ステートレス 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。

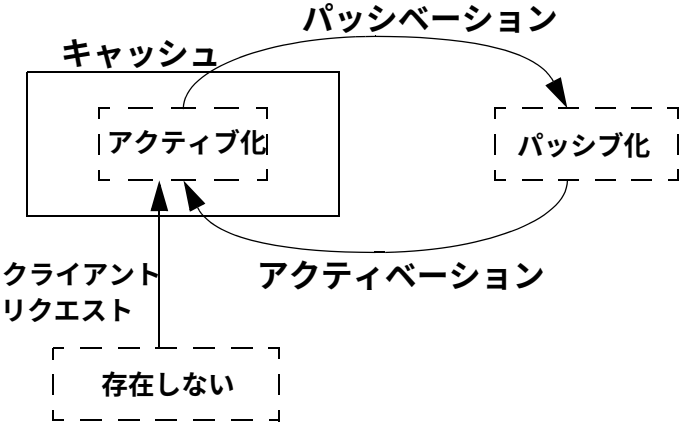
EJB クラスのすべてのインスタンスがアクティブで、`max-beans-in-free-pool` に達した場合、EJB クラスを要求する新しいクライアントは、アクティブ EJB がメソッド呼び出しを完了するまでブロックされます。トランザクションがタイムアウトになった場合（トランザクション非対応の呼び出しでは、5 分経過した場合）、WebLogic Server は `RemoteException` を送出します。

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。アクティブ EJB は、以下の節で示すように、クライアントが現在使用しているインスタンスと最近使用したインスタンスから構成されます。キャッシュ内のステートフル セッション Bean が特定のクライアントにバインドされるのに対し、フリープール内のステートレス セッション Bean はクライアントに関連付けられないという点で、キャッシュはフリープールとは異なります。

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。点線は、WebLogic Server 側から見た EJB の状態を表しています。

図 4-2 ステートフル セッション EJB のライフサイクルを示す WebLogic Server キャッシュ



ステートフル セッション EJB インスタンスのアクティブ化と使用

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。クライアントが個々の Bean への参照をルックアップして取得すると、WebLogic Server は EJB クラスの新しいインスタンスを初期化してキャッシュに格納します。

ステートフル セッション EJB のパッシブ化

ハイ パフォーマンスを実現するために、WebLogic Server はクライアントが現在使用している EJB と最近使用した EJB をキャッシュに保存します。EJB がこれらの基準を満たさなくなると、それらはパッシベーションの対象となります。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 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 は最小限の WebLogic Server リソースしか使用せず、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 リクエストのために即座に使用することはできません。

注意： ステートフル セッション EJB は、一定のルールに従わなければなりません。これにより、Bean フィールドを永続ストレージにシリアライズできます。詳細については、4-8 ページの「ステートフル セッション EJB の要件」を参照してください。

`weblogic-ejb-jar.xml` ファイルの `max-beans-in-cache` デプロイメント要素を使用すると、いつ EJB のパッシベーションを行うかを指定できます。

`max-beans-in-cache` 制限に達し、使用されていない EJB がキャッシュに存在する場合、WebLogic Server はそれらの一部のパッシベーションを行います。パッシベーションは、使用されていない Bean がその `idle-timeout-seconds` 制限に達していない場合でも行われます。`max-beans-in-cache` に達し、キャッシュ内のすべての EJB がクライアントによって使用されている場合、WebLogic Server は `CacheFullException` を送出します。

注意： EJB がパッシベーションの対象になっても、WebLogic Server がその Bean のパッシベーションを即座に行うとは限りません。実際には、その Bean のパッシベーションがまったく行われない場合もあります。パッシベーションが行われるのは、EJB がパッシベーションの対象となり、かつサーバ リソースが不足している場合か、または WebLogic Server が通常のキャッシュ メンテナンスを実行した場合だけです。

`weblogic-ejb-jar.xml` ファイルで `cache-type` 要素を設定することにより、`idle-timeout-seconds` に達したステートフル EJB の明示的パッシベーションを指定することができます。この設定には、最長時間未使用（least recently used : `LRU`）と最近未使用（not recently used :`NRU`）の 2 つの値があります。EJB をより活発にアクティブにする必要があるときは `LRU` を指定します。キャッシュ内のメモリの制約があり、クライアントによる使用頻度が最も高い Bean をメモリに保持するときは `NRU` を指定します。

ステートフル セッション EJB インスタンスの削除

`max-beans-in-cache` および `idle-timeout-seconds` デプロイメント要素を使用すると、ステートフル セッション EJB をいつキャッシュまたはディスクから削除するかを指定することもできます。

- キャッシュされた EJB インスタンス : WebLogic Server が max-beans-in-cache 制限に達しようとしている EJB クラスを検出したとき。それらの Bean の中から、WebLogic Server は idle-timeout-seconds の間使用されていない EJB を選択して、ディスク上でパッシベーションを行わずにキャッシュから削除します。パッシベーションではなく削除を行うと、「非アクティブ」な EJB が WebLogic Server のキャッシュまたはディスク リソースを消費することはありません。

  ステートフル セッション Bean が idle-timeout-seconds より長くアイドル状態になっている場合、WebLogic Server は、EJB クラスの max-beans-in-cache 制限に達していない場合でも、通常のキャッシュ メンテナンスによってそのインスタンスをメモリから削除します。

注意 : idle-timeout-seconds を 0 に設定すると、WebLogic Server は通常のキャッシュ メンテナンスで EJB を削除しません。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。

- パッシブ化されている EJB インスタンス : ステートフル セッション EJB インスタンスのパッシベーションが行われたら、クライアントは idle-timeout-seconds に達するまでにその EJB インスタンスを使用しなければなりません。使用しなかった場合、WebLogic Server はそのインスタンスをディスクから削除します。

ステートフル セッション EJB の要件

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 によってそのデータがシリアライズされてその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。

ステートレス セッション Bean と BMP EJB のパフォーマンス比較

パフォーマンスを向上させるために、データの取得には BMP（Bean 管理の永続性）エンティティ Bean ではなくステートレス セッション Bean または CMP（コンテナ管理の永続性）エンティティ Bean を使用することをお勧めします。BMP エンティティ Bean がファインダ クエリの間にデータをキャッシュできないことにより、ステートレス セッション Bean のパフォーマンスは BMP エンティティ Bean よりも 90% 程度高速な場合がある、ということが各種のテストで実証されています。たとえば、ファインダ クエリからの 100 個の Bean を返す BMP エンティティ Bean は、ファインダ クエリの実行時に 1 回の JDBC 呼び出しを行って Bean 参照を作成し、その後、クライアントが各 Bean にアクセスするたびに JDBC 呼び出しを行って Bean をロードします。つまり、BMP エンティティ Bean のファインダ クエリは合計でデータベースに対して 101 回の呼び出しを行います。比較すると、ステートレス セッション Bean は JDBC 呼び出しを 1 回しか行わないため、そのパフォーマンスはずっと高速です。

BMP エンティティ Bean がスケーラビリティの面で不利であるという事実は、パフォーマンス上の既知の制限事項です。

加えて、CMP エンティティ Bean はファインダ クエリの間にデータをキャッシュできます。そのため、ステートレス セッション Bean の場合と同様にクエリの実行は 1 回で済みます。

max-beans-in-free-pool の使用

通常は、`max-beans-in-free-pool` 要素を指定しないでください。`max-beans-in-free-pool` を設定するのは、基底のリソースへのアクセスを制限する場合です。たとえば、ステートレス セッション EJB を使用して従来の接続プールを実装する場合、従来のシステムがサポートする接続数を超える数の Bean インスタンスを割り当てる必要はありません。フリー プールの Bean インスタンスを要求する場合、次の 3 つのシナリオが考えられます。

- **オプション 1 :** 使用可能なインスタンスがプールにあります。WebLogic Server はそのインスタンスを使用可能にし、ユーザは処理を続行します。
- **オプション 2 :** 使用可能なインスタンスがプールにありませんが、使用中のインスタンス数は `max-beans-in-free-pool` 未満です。WebLogic Server は新しい Bean インスタンスを割り当て、ユーザに提供します。
- **オプション 3 :** 使用可能なインスタンスがプールにありませんが、使用中のインスタンス数は `max-beans-in-free-pool` に達しています。トランザクションがタイムアウトするか、プール内の既存の Bean インスタンスが使用可能になるまで待機します。

デフォルトでは、`max-beans-in-free-pool` は最大を表す `Int.max` に設定されています。しかし、20 億のインスタンスを使用できるということではありません。新しい Bean インスタンスを 1 つ割り当てるだけなので、オプション 3 の状況は基本的に発生しません。ただし、実行スレッドの数による制限があります。ほとんどの場合、各スレッドに必要な Bean インスタンスは最大でも 1 つです。

max-beans-in-free-pool の特殊な使い方

以下のオプションでは、`max-beans-in-free-pool` を `0` に設定した場合の特殊なケースについて説明します。

- **エンティティ Bean :** WebLogic Server はエンティティ Bean のインスタンスをプールしません。代わりに、WebLogic Server は新しいインスタンスを作成します。
- **ステートレス セッション Bean :** WebLogic Server は常にステートレス セッション Bean の新しいインスタンスを作成します。
- **ステートフル セッション Bean :** スレートフル セッション Bean には該当しません。これらの Bean はプールされません。
- **メッセージ駆動型 Bean :** デプロイ時にメッセージ駆動型 Bean の初期インスタンスが作成され、JMS リスナに登録されます。WebLogic Server は実行時に新しいインスタンスを作成しません。したがって、`initial-beans-in-free-pool` は `> 0` に設定する必要があります。

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。

db-is-shared を使用した ejbLoad() 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。

単一の WebLogic Server インスタンスのみが特定の EJB に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 インスタンスが特定の 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 db-is-shared 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、db-is-shared は各 EJB に対して Bean の weblogic-ejb-jar.xml ファイルで true 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 インスタンスだけが EJB の基盤データにアクセスするようなケースでは、同時実行性オプションが [Exclusive] に設定されている場合に、その Bean の weblogic-ejb-jar.xml ファイルの db-is-shared を「false」に設定できます。db-is-shared を false 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、その Bean の ejbLoad() を呼び出します。

- EJB がキャッシュに存在しない場合
- EJB のトランザクションがロールバックされた場合

db-is-shared に関する制限と警告

db-is-shared を「false」に設定すると、EJB の基盤データが 1 つの WebLogic Server インスタンスによって更新されるか、複数のクライアントによって更新されるかにかかわらず、WebLogic Server のデフォルト ejbLoad() 動作（コンテナ管理による永続性）がオーバーライドされます。間違って db-is-shared を「false」に設定し、複数のクライアント（データベース クライアント、別の WebLogic Server インスタンスなど）が Bean データを更新する場合、データの完全性が失われる恐れがあります。

エンティティ Bean の concurrency strategy を「Database」オプションに設定する場合は、db-is-shared を「false」に設定しないでください。このように設定しても、WebLogic Server は db-is-shared の設定を無視します。

データベース ロックが指定されていると、EJB コンテナは、エンティティ Bean クラスのインスタンスのキャッシュを引き続き行います。ただし、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はキャッシュしません。代わりに、WebLogic Server が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、最新の EJB データを取得します。つまり、db-is-shared を「false」に設定して WebLogic Server が各トランザクションの開始時に ejbload() を呼び出さないようにするのは無意味です。

また、キャッシュの制限のために、WebLogic Server クラスタでは db-is-shared を「false」に設定できません。

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

このメソッドは現在のバージョンでは必須ではありません。

注意： 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。ただし、この要素は EJB 2.0 では必須ではなくなりました。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ます。WebLogic Server の is-modified-method-name デプロイメント要素は、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、10-37 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。これを行うには、`delay-updates-until-end-of-tx` を「false」に設定します。

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る。

エンティティ EJB の read-only への設定

エンティティ EJB は、`read-only` 同時方式を使用して `ejbLoad()` と `ejbStore()` の基本動作を変更することもできます。以降の各項では、EJB コンテナによる同時方式サービスのサポートについて説明します。

read-only キャッシュ方式を設定するには、weblogic-ejb-jar.xml デプロイメント ファイルの concurrency-strategy 要素を編集します。デプロイメント記述子の編集方法の詳細については、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。

read-only 同時方式

read-only 同時方式は、EJB クライアントによって変更されないが、外部リソースによって定期的に更新されるエンティティ EJB に対して使用できます。たとえば、read-only エンティティ EJB を使用すると、WebLogic Server システムの外部で更新される、特定の企業の株価を表すことができます。

WebLogic Server では、read-only エンティティ EJB に対して ejbStore() は呼び出されません。ejbLoad() は、EJB が最初に作成されたときに呼び出されます。それ以後 WebLogic Server は、read-timeout-seconds デプロイメント パラメータによって定義されている間隔で ejbLoad() を呼び出します。

read-only 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 read-only 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ません。
- トランザクション属性は NotSupported に設定する必要があります（Bean はトランザクションに依存できません）。
- 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りません。詳細については、4-23 ページの「クラスタ内のセッション 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 によって制御されます。

read-only マルチキャストの無効化

read-only 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。

read-only 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。

Figure 4-3 CachingHome および CachingLocalHome インタフェースを示すサンプル コード

```
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

}
```

次のサンプル コードは、ホームを CachingHome にキャストし、続いてメソッドを呼び出す方法を示しています。

Figure 4-4 ホームをキャストしてメソッドを呼び出す方法を示すサンプル コード

```
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、read-only 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 read-only 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。

標準の read-only エンティティ Bean

WebLogic Server は、デプロイメント記述子で設定する read-timeout 要素によって、標準の read-only エンティティ Bean を引き続きサポートします。ReadOnly オプションが concurrency strategy 要素で選択され、read-timeout-seconds 要素が weblogic-ejb-jar.xml ファイルで設定されている場合、read-only エンティティ Bean が呼び出されると、WebLogic Server はキャッシュされているデータが read-timeout 設定よりも古いかどうかを確認します。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。したがって、以前のバージョンの read-only エンティティ Bean はこのバージョンの WebLogic Server で機能します。

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。

```
wlserver6.1\samples\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、read-only エンティティ Bean と read-write 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 read-only エンティティ Bean を使用し、データを更新するには read-write エンティティ Bean を使用します。read-mostly パターンの詳細については、4-18 ページの「read-mostly パターン」を参照してください。

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` を同じ値に設定します。
- 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します。
- 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します。
- 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します。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。

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、4-27 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。

read-write キャッシュ方式

`read-write` 方式では、WebLogic Server のエンティティ EJB のデフォルト キャッシング動作を定義します。

`read-write` EJB の場合、WebLogic Server は各トランザクションの開始時、または 4-12 ページの「db-is-shared を使用した ejbLoad() の呼び出しの制限」で説明したとおり、EJB データをキャッシュにロードします。WebLogic Server は、トランザクションが正常にコミットされたとき、または 4-13 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）」で説明したとおり、`ejbStore()` を呼び出します。

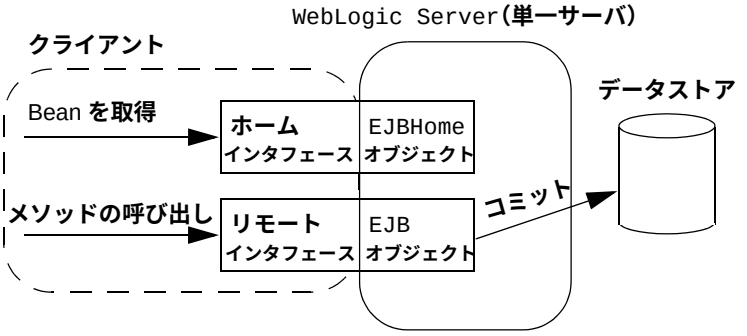
WebLogic Server クラスタにおける EJB

この節では、EJB コンテナによるクラスタ化サービスのサポートについての情報を示します。また、WebLogic Server クラスタでの EJB とそれに関連付けられるトランザクションについて説明し、クラスタでの EJB の動作に影響を与えるデプロイメント記述子について解説します。

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 4-5 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。

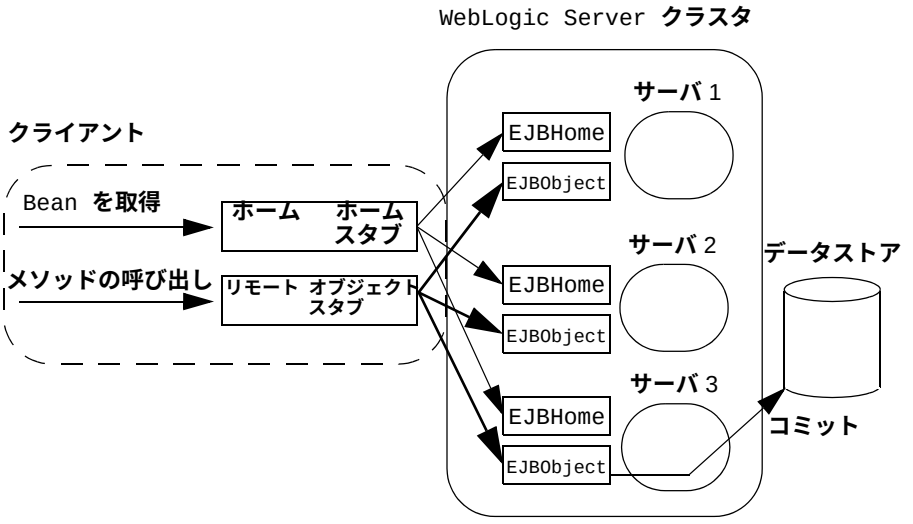
クラスタ化された EJBHome オブジェクト

WebLogic Server クラスタでは、ホーム オブジェクトのサーバサイド表現は、クラスタ対応の「スタブ」によって置き換えることができます。クラスタ対応のホーム スタブは、クラスタ内のすべての WebLogic Server に存在する EJB ホーム オブジェクトの知識を備えています。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。

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。この要素を「true」（デフォルト）に設定すると、`ejbc` は適切なオプションを使用して `rmic` コンパイラを呼び出し、EJB に対してクラスタ対応のホーム スタブを作成します。

次の図は、WebLogic Server クラスタ環境での EJB の動作を示しています。クラスタ化されたサーバ環境での EJB の動作についての図解は、図 4-6 を参照してください。

図 4-6 クラスタ化されたサーバ環境



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 サービスを提供します。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キャッシュ方式によって決まります。

クラスタ内のセッション EJB

この節では、ステートフル セッション EJB とステートレス セッション EJB のそれぞれについて、クラスタの機能と制限を説明します。

ステートレス セッション EJB

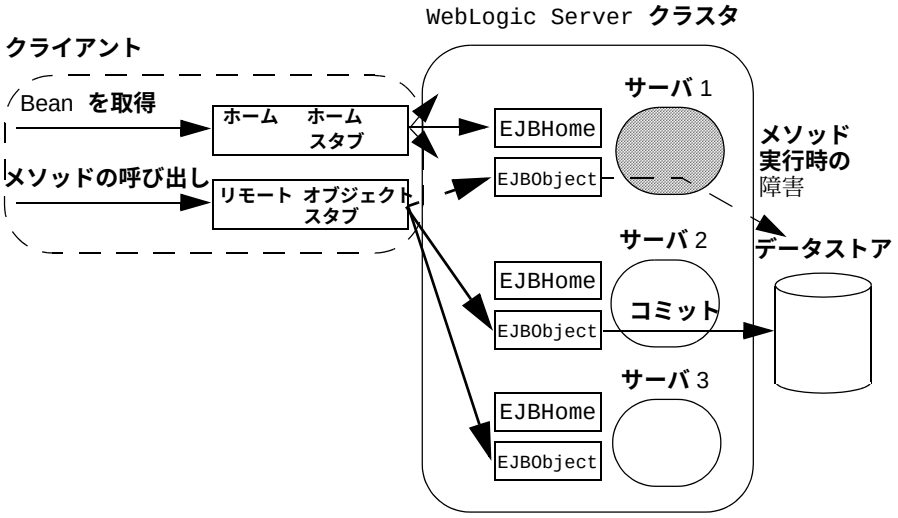
ステートレス セッション EJB は、クラスタ対応のホーム スタブとレプリカ対応の EJBObject スタブの両方を持つことができます。デフォルトによって、WebLogic Server は EJB メソッド呼び出しに対するフェイルオーバ サービスを提供しますが、これは障害がメソッド呼び出しの合間に発生した場合に限ります。たとえば、メソッドの完了後に障害が発生した場合や、メソッドがサーバに接続されなかった場合などには、フェイルオーバが自動的にサポートされます。しかし、EJB メソッドの実行中に障害が発生した場合、WebLogic Server は別のサーバにそのメソッドを自動的にフェイルオーバしません。

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 `stateless-bean-methods-are-idempotent` デプロイメント プロパティが用意されています。`weblogic-ejb-jar.xml` でこのプロパティを「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。

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 4-7 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



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、4-25 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。

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

以降の節では、EJB コンテナによるレプリケーション サービスのサポートについて説明します。WebLogic Server EJB コンテナは、ステートフル セッション EJB に対する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したが、EJB コンテナでは、EJB の状態もクラスタ化された WebLogic Server インスタンス間でレプリケートできます。

ステートフル セッション EJB に対するレプリケーション 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 インスタンスのリストと、その Bean の状態をレプリケートするために使われるセカンダリ WebLogic Server の名前を保持します。

EJB のクライアントが 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 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には、WebLogic Server はクラスタから新しいセカンダリ サーバ インスタンスを確保します。

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 4-26 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。

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 インスタンス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が基になっています。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 4-8 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。ただし、まれに発生する次のような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場合があります。

- 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 インスタンスを作成し直し、トランザクションを再度実行する必要があります。

- 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 インスタンスを作成し直し、トランザクションを再度実行する必要があります。

クラスタ内のエンティティ EJB

すべての EJB タイプの場合と同じように、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 cache-strategy デプロイメント要素によって決まります。

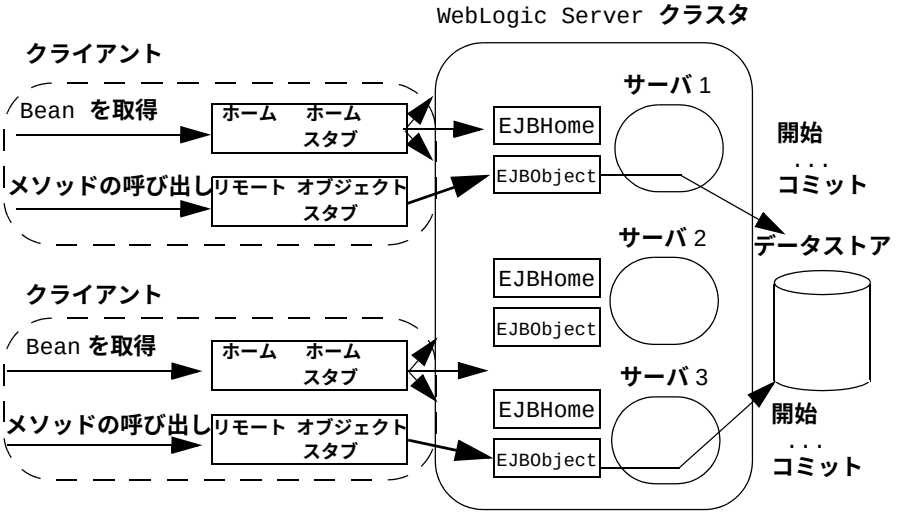
read-write クラスタ内のエンティティ EJB

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 を使用できます。
- ejbLoad() は、常に各トランザクションの最初に呼び出されます。db-is-shared デプロイメント パラメータはクラスタ内では「false」に設定できないためです。
- ejbStore() の動作は、4-11 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 4-9 は、WebLogic Server クラスタ環境での read-write 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 4-9 クラスタ化されたサーバ環境の read-write 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。

read-write 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とき、クラスタ内の管理対象サーバを識別するクラスタ アドレスを指定します。クラスタ アドレスはエンティティ Bean とステートレス Bean で、URL のホスト名部分を構成するために使用されます。ク

ラスタ アドレスが設定されていない場合、EJB による処理が正しく機能しない場合があります。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。トランザクションと 2 フェーズ コミット プロトコルの使い方については、「トランザクションについて」を参照してください。

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。

- **Bean 管理のトランザクション**では、トランザクションの境界定義を EJB が管理します。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である場合、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。トランザクション境界の定義の詳細については、4-30 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 の EJB コンテナがトランザクションの境界定義を管理します。コンテナ管理トランザクションを使用する EJB （またはコンテナ管理トランザクションと Bean 管理トランザクションを併用する 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 はデフォルトによって supports 属性を使用します。

トランザクション イベントの順序は、コンテナ管理トランザクションの場合と Bean 管理トランザクションの場合で異なります。

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。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、アプリケーションで Bean 管理またはコンテナ管理のどちらによるトランザクション活性化を使用するかによって異なります。以下の節では、これらの両方のシナリオ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 4-10 を参照してください。

図 4-10 ユーザ トランザクションのアイソレーション レベルを設定するサンプルの Java コード

```
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 レベルの設定と、基盤の永続ストレージの同時実行性制御の両方によって決まります。コンテナ管理トランザクションのアイソレーション レベルの設定について、詳しくは『WebLogic JTA プログラマーズ ガイド』を参照してください。

TRANSACTION_SERIALIZABLE の制限

多くのデータストアでは、単一のユーザ接続が対象の場合でも、シリアライゼーションに関する問題の検出は限定的にしかサポートされていません。したがって、`transaction-isolation` を `TRANSACTION_SERIALIZABLE` に設定する場合でも、データストアの制限が原因でシリアライゼーションの問題に遭遇する場合があります。

アイソレーション レベルのサポートの詳細については、使用している RDBMS のマニュアル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

Oracle ではオプティミスティックな同時実行性が採用されています。その結果として、`TRANSACTION_SERIALIZABLE` に設定したとしても、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
```

EJB に対して TRANSACTION_SERIALIZABLE 設定を使用する場合でも、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、メソッドに対してトランザクションのアイソレーション レベルを TRANSACTION_READ_COMMITTED_FOR_UPDATE に設定できます。設定されると、その時点から各 SELECT クエリには、選択したデータのロックを必要とする FOR_UPDATE が追加されます。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。

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 コンテキストからの複数の EJB の呼び出し

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。

コード リスト 4-1　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のカプセル化

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 クラスタにおける EJB 間のトランザクションの分散

WebLogic Server は、WebLogic Server クラスタ内の EJB を使用するトランザクションのパフォーマンスを向上します。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。

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 Server による新しい Bean の挿入を遅らせるには、`weblogic-cmp-rdbms-jar.xml` ファイルの `delay-database-insert-until` 要素で、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、データベースの挿入を `ejbPostCreate` の後に遅らせる必要があります。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。

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

- ejbCreate — 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 — 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）。

コード リスト 4-2 delay-database-insert until 要素を指定する XML の例

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

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。

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. 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. [データ ソース] を選択し、[新しい JDBC Data Source のコンフィグレーション] オプションを選択します。
6. [名前]、[JNDI 名]、[プール名] を入力します。. 各 resultSet のクライアントと WebLogic Server との間の行のプリフェッチを行う場合は、行プリフェッチが有効になっていることを確認した上で、[Row Prefetch サイズ] および [ストリーム チャンク サイズ] を指定します。
   a. トランザクション非対応の JDBC データソースの場合は、データソースにバインドするための完全な WebLogic Server JNDI 名と、WebLogic Server 接続プールの名前を入力します。
   b. トランザクション対応の JDBC データソースの場合は、トランザクション対応データソースにバインドするための完全な WebLogic Server JNDI 名と、WebLogic Server 接続プールの名前を入力します。

   トランザクション対応およびトランザクション非対応のデータソースをコンフィグレーションする方法の詳細については、「JDBC 接続の管理」を参照してください。
7. [作成] をクリックして新しい JDBC データ ソースを保存します。

8. 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。

   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` 要素を直接編集して、既存の EJB リソース ファクトリ参照を JNDI 名にマップします。デプロイメント記述子の編集についての説明は、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。

エンティティ EJB のロック サービス

以降の節では、EJB コンテナによるロック サービスのサポートについて説明します。WebLogic Server コンテナでは、データベース ロックと排他的ロック メカニズムの両方がサポートされています。EJB 1.1 および EJB 2.0 の Bean に対しては、デフォルト メカニズムであるデータベース ロックを使用することをお勧めします。

排他的ロック サービス

WebLogic Server 5.1 および 4.5.1 では、排他的ロックがデフォルトでした。ペシミスティック 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。

データベース ロック サービス

WebLogic Server 6.x では、データベース ロックがデフォルトとなっています。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。

データベース ロックの設定

`weblogic-ejb-jar.xml` の `concurrency-strategy` デプロイメント パラメータを設定することによって、EJB 用に使用するロック メカニズムを指定します。`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 `weblogic-ejb-jar.xml` からの引用に、データベース ロックを使用する EJB を示します。

コード リスト 4-3 データベース ロックの例

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

concurrency-strategy を指定しない場合、4-39 ページの「データベース ロック サービス」で説明したとおり、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。

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 コンテナで新しく導入されたコンテナ管理による永続性サービス（CMP）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
- EJB 2.0 用 EJB QL の使い方
- 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
- カスケード削除
- 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
- EJB 2.0 CMP に対する自動主キー生成
- 自動テーブル作成
- コンテナ管理による永続性関係
- グループ
- 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

WebLogic Server のコンテナは、EJB とサーバ間の統一的なインタフェースを提供する役割を持っています。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 6.1 より前の 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『WebLogic Server 6.1 リリース ノート』の『WebLogic Server 6.0 アプリケーションの WebLogic Server 6.1 への移行』を、変換ツールの使い方については 9-4 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。

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-type および persistence-use 要素に格納されます。`persistence-type` には、EJB が使用できる 1 つまたは複数の自動サービスが定義されます。`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-type および 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

WebLogic Server RDBMS ベースの永続性サービスを EJB に対して使用するには、専用の XML デプロイメント ファイルを作成して、コンテナ管理の永続性を使用する各 EJB に対して永続性要素を定義します。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` ファイルは、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、5-10 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。EJB プロバイダは、`EJBHome` インタフェースにファインダのメソッド シグネチャを記述し、`ejb-jar.xml` デプロイメント ファイルにその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

EJB プロバイダは、find*MethodName*() というフォームを使用して、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： EJB プロバイダは、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) メソッドも定義できます。

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 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、5-6 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とことを想定して、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、5-10 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ーライドされます。

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。

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* 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)

演算子	説明	サンプル構文
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*')
desc	結果を降順に並び替える。orderBy と組み合わせたときだけ使用できる	(orderBy '*column_name* desc')

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。
- $*n* によって識別されるファインダ パラメータまたは Java 式。*n* はパラメータまたは式の数です。デフォルトによって、$*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

次のサンプル コードは、基本的な WLQL 式を使用する weblogic-cmp-rdbms-jar.xml ファイルからの引用です。

- 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> スタンザのサンプルは次のとおりです。

```
<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、サンプル 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> スタンザのサンプルでは、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>
```

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。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

WLQL のサンプル構文	対応する EJB QL の構文
(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ーライドします。

SELECT DISTINCT

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

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。

コード リスト 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。

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、一連のアクセサ メソッドを使用します。これらのメソッドの名前の先頭には、*set* と *get* が付いています。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これらのメソッドを使用します。コンテナによって生成されるこれらのクラスは、「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する必要があります。また、これらのメソッドは、`public`、`protected`、および `abstract` として宣言します。

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 は、CLOB をデータ型 `java.lang.string` にしかマップしません。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。

コード リスト 5-2　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。

コード リスト 5-3 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>
```

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。

コード リスト 5-4　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 がデータベースに削除された場合、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。

コード リスト 5-5 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 );
```

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 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。

コード リスト 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

主キーとは、エンティティ Bean をそのホーム内で一意に識別するオブジェクトです。コンテナはエンティティ Bean の主キーを操作できる必要があります。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。エンティティ Bean クラスで 1 つまたは複数のフィールドに主キーをマップすることにより、永続性がコンテナによって管理されるエンティティ Bean に対して主キー クラスを指定できます。

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である 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、パラメータを取らない 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である 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 ファイル内の主キー フィールドと同じ名前を持たなければなりません。

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のフィールドにしない。

  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でください。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。これは、Java 言語と各種データベースの精度の違いによるものです。たとえば、このメソッドでは 7.1 と 7.10 は等しいと判断されません。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。

  主キーとして BigDecimal を使用する必要がある場合、次の手順に従ってください。

  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 は読み取り専用となります。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、5-24 ページの「Oracle 用主キー サポートの指定」と 5-25 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、5-25 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。

注意： キー フィールドは、Bean の抽象「get」および「set」メソッドで `java.lang.Integer` 型として宣言しなければなりません。

キー フィールドの有効値

Bean の抽象「get」および「set」メソッドでは、キー フィールドを以下のいずれかの型として宣言できます。

- 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。

コード リスト 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。

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。

コード リスト 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。

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。

コード リスト 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。

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。

WebLogic Server は、できる限り新しいテーブルを作成しようとします。しかし、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 タイプにマップできない場合、TABLE CREATION 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。
2. 次の構文で指定します。

   ```
   <create-default-dbms-tables>True</create-default-dbms-tables>
   ```

自動テーブル作成機能ではすべての Java フィールド タイプを対象データベースに正しくマップできない場合もあるので、マッピングの内容を把握できるように次のリストが用意されています。

表 5-1 Java フィールド タイプ

Java の型	DBMS カラム タイプ
boolean	INTEGER
byte	INTEGER
char	CHAR
double	DOUBLE PRECISION
float	FLOAT
int	INTEGER
long	INTEGER
short	INTEGER
java.lang.String	VARCHAR (150)

Java の型	DBMS カラム タイプ
java.lang.BigDecimal	DECIMAL (38, 19)
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)

コンテナ管理による永続性関係

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。

エンティティ Bean は、他の Bean と関係を持つことができます。それらの関係は、双方向の場合と一方向の場合があります。

関係は、`ejb-jar.xml` ファイルと `weblogic-cmp-rdbms-jar.xml` ファイルで指定します。コンテナ管理フィールドのマッピングは、`weblogic-cmp-rdbms-jar.xml` ファイルで指定します。

WebLogic Server では、WebLogic のコンテナ管理による永続性（CMP）によって管理される以下の 3 種類の関係をマップできます。

- 1 対 1
- 1 対多
- 多対多

1 対 1 の関係

WebLogic Server の 1 対 1 の関係では、一方の Bean の外部キーが他方の Bean の主キーに物理的にマップされます。主キーの詳細については、5-20 ページの「主キー」を参照してください。

1 対多の関係

WebLogic Server の 1 対多の関係では、一方の Bean の外部キーが他方の Bean の主キーに物理的にマップされます。ただし 1 対多の関係では、外部キーが常に、関係の「多」サイドのロールに格納されます。

多対多の関係

WebLogic Server の多対多の関係には、結合テーブルの物理的なマッピングが伴います。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マップする 2 つの外部キーが格納されます。

一方向の関係

一方向の関係では、一方向でのみナビゲートできます。これらの関係はリモート Bean との間に使用され、一方向の関係だけがリモートの関係になります。リモート Bean とは、同じ `EJB-jar` ファイルで関係を持つ Bean として定義されていない抽象的永続性スキーマを持つ Bean のことです。たとえば、エンティティ A とエンティティ B が 1 対 1 の関係にあり、その方向がエンティティ A からエンティティ B への一方向である場合、エンティティ A はエンティティ B の存在を認識していますが、エンティティ B はエンティティ A の存在を認識していません。このタイプの関係は、ナビゲーションが行われるエンティティ Bean に CMR-field があり、対象のエンティティ Bean に関連する CMR-field がない状態で実装されます。

双方向の関係

双方向の関係では、双方向でナビゲートできます。このタイプのコンテナ管理の関係は、抽象永続性スキーマが同じ `EJB-jar` ファイルで定義されており、したがって同じコンテナ マネージャによって管理される Bean の間だけで成立します。たとえば、エンティティ A とエンティティ B が 1 対 1 で双方向の関係にある場合、両者は互いに認識し合います。

関係内の Bean の削除

別の Bean と関係を持つ Bean が削除されると、コンテナはその関係を自動的に削除します。

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。したがって、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。

6.1 より前のバージョンの WebLogic Server 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。

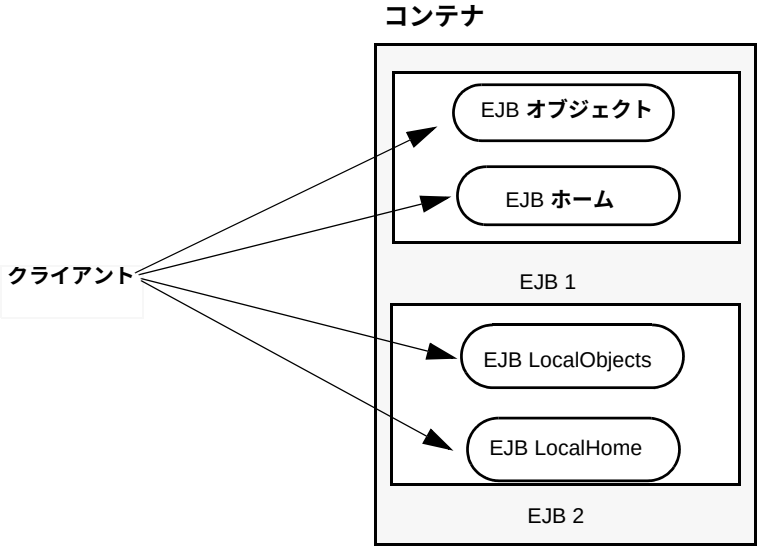
ローカル クライアントの使用

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 5-1 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。

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 5-2 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
java.sql.Date	DATE

CMP フィールドの Java の型	Oracle のデータ型
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

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 ロジックを定義する。
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。

コンポーネント	説明
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。
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 のキャッシング、クラスタ化、およびパフォーマンスの各動作を定義する 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 10 章「weblogic-ejb-jar.xml 文書型定義」を参照してください。

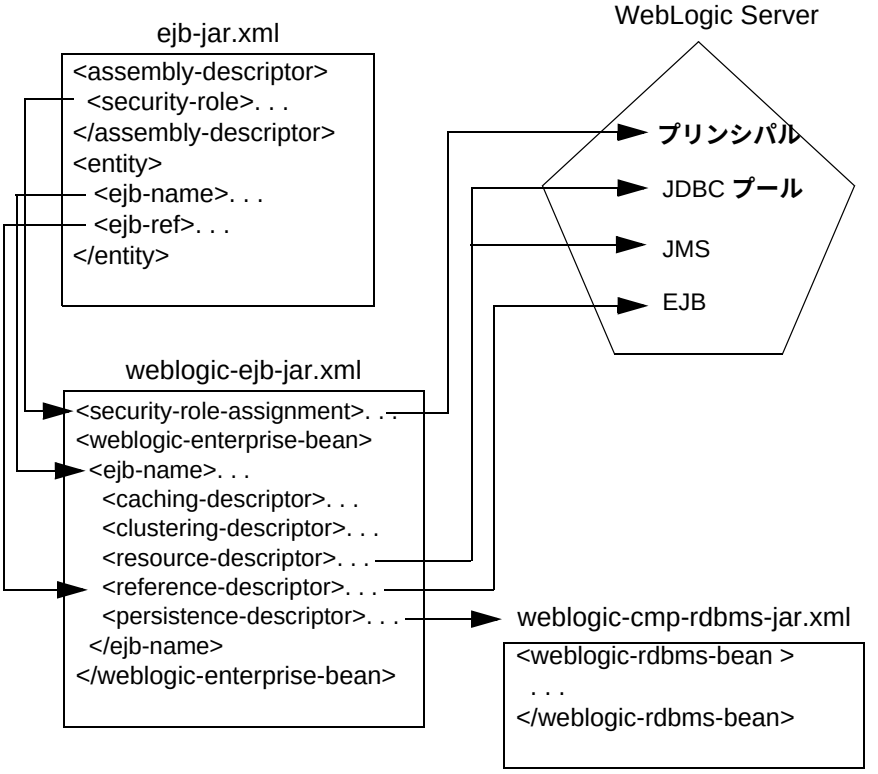
weblogic-cmp-rdbms.xml

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 11 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します。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 ファイルを編集します。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 に変換します。`DDConverter` ツールの使用方法については、9-4 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。

```
wlserver\samples\examples\ejb20
```

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 [EJB] を選択します。
3. 左ペインで、使用しているサーバ ドメインの [デプロイメント] ノードを選択します。
4. [デプロイメント] ノードを展開し、[EJB] を選択します。
5. 展開されたデプロイ済み EJB のリストから、編集する Bean を右クリックします。
6. [EJB 記述子の編集 ...] をクリックします。
7. EJB デプロイメント記述子エディタが表示されたら、選択した EJB をクリックして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項目が表示されます。

   - **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。
8. 編集するデプロイメント記述子のノードを展開します。

   選択したデプロイメント記述子ファイルの現在の設定が左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。
9. 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。

10. 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。

11. 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 10 章「weblogic-ejb-jar.xml 文書型定義」と第 11 章「weblogic-cmp-rdbms-jar.xml 文書型定義」を参照してください。

WebLogic Server デプロイメント モードの設定

次のいずれかの方法で、エンタープライズ アーカイブ ファイル（EAR）または EJB を WebLogic Server にデプロイします。

- 自動モード。EJB または EAR はサーバのアプリケーション ディレクトリに自動的にデプロイされます。
- プロダクション モード。EJB または EAR は、`config.xml` ファイルで指定したとおりにデプロイされます。

自動モードによるデプロイメント

自動モード デプロイメント オプションはデフォルトです。この機能は、開始時とサーバの実行中にアクティブなサーバのアプリケーション ディレクトリを自動的にポーリングして、EJB デプロイメントが変更されていないどうかを調べます。デプロイメントが変更されていた場合、サーバをポーリングしたときに自動的にデプロイされます。開発モードでは、デプロイする EJB または EAR の `applications` ディレクトリを使用します。デプロイ後、これらの EJB/EAR は `config.xml` ファイルに合わせて自動的に保持されます。

また、WebLogic Server は、EJB デプロイメントが変更されていないか調べるために、10 秒ごとに applications の内容をチェックします。デプロイメントが変更されていた場合、動的デプロイメント機能を使用して自動的に再デプロイされます。

EJB サンプルの自動デプロイ

WebLogic Server に付属の EJB サンプルは、wlserver\config\applications ディレクトリに自動的にデプロイされます。

- wlserver\samples\examples\ejb ディレクトリのサンプルはビルドされた状態で出荷されるので、examples サーバを起動すると自動的にデプロイされます。
- wlserver\samples\examples\ejb20 ディレクトリのサンプルはビルドされていない状態で出荷されるので、サーバにデプロイする前にビルドする必要があります。

プロダクション モードによるデプロイメント

プロダクション モード デプロイメント オプションは、自動デプロイメントを無効にします。プロダクション モード デプロイメントを有効にすると、config.xml ファイルで指定されたアプリケーションがサーバの起動時にデプロイされます。

このモードを有効にするには、コマンド ラインで次のコマンドを true に設定します。

```
-d production mode enabled true
```

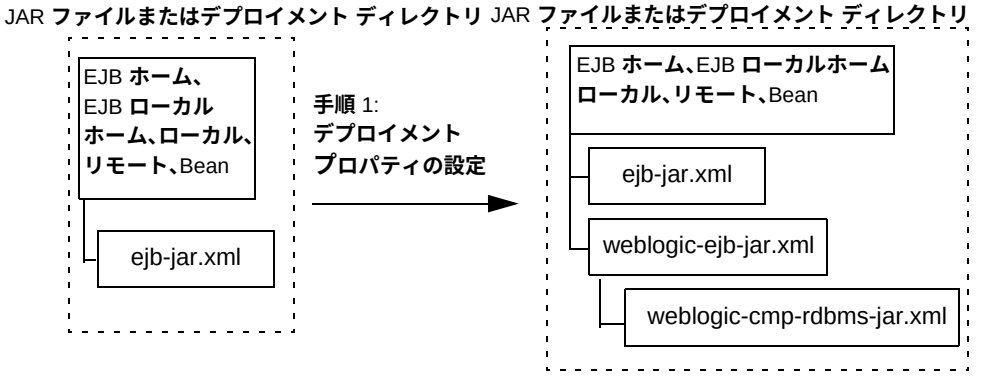
プロダクション モードの詳細については、『管理者ガイド』の startstop.html にある「コマンドラインからの WebLogic 管理サーバの起動」を参照してください。

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 6-2 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。

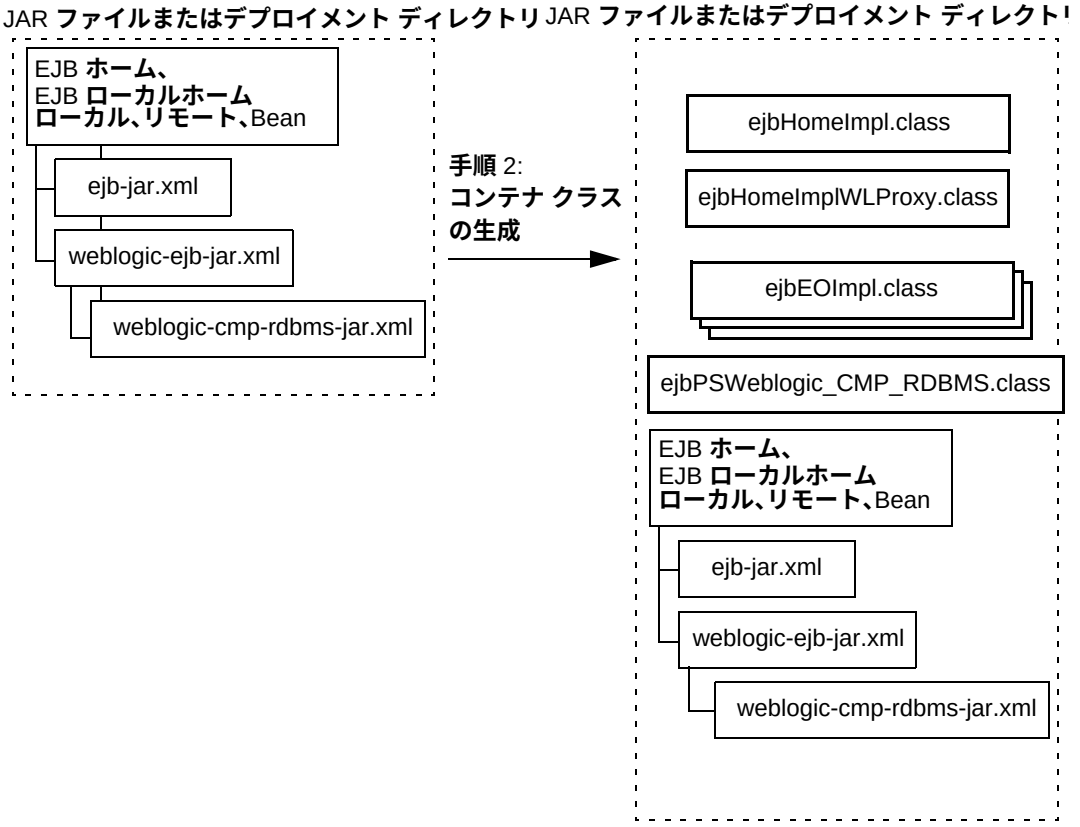
   コンテナ クラスには、WebLogic Server が使用する EJB の内部表現に加えて、クライアントが使用する外部インタフェース（ホーム、ローカル、またはリモート）の実装も格納されます。

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、9-1 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 6-3 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。

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` ファイルを作成するには、この機能を Bean の `ejb-jar.xml` デプロイメント記述子ファイルを指定してから、`weblogic.ejbc` を使用して `ejb-client.jar` ファイルを生成します。`ejb-client.jar` には、`ejb-jar` ファイルの EJB を呼び出すためにクライアント プログラムに必要なクラス ファイルが格納されます。これらのファイルは、クライアントをコンパイルするために必要なクラスです。この機能を指定した場合、WebLogic Server は `ejb-client.jar` を自動的に作成します。

`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>
<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。

ejb-client.jar には、エンティティ Bean のホーム インタフェース、リモート インタフェース、および主キー クラスが必ず格納されます。また、これらのインタフェースが参照する ejb-jar ファイルのすべてのクラスのコピーも格納されます。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。このリモート インタフェースはこのクラスを参照し、ejb-jar ファイルに入っているので、EJB client.jar に含まれます。

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。

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、6-15 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 EJB でのセキュリティのコンフィグレーション

EJB へのアクセスを制限することで、EJB にセキュリティ対策を施すことができます。EJB を指定してアクセスを制限するには、EJB にセキュリティ制約を適用します。

セキュリティ制約のコンフィグレーション

セキュリティ制限をコンフィグレーションするには、以下の手順を実行します。

1. `weblogic\examples\ejb\basic\containerManaged\index.html` の説明に従って、環境を設定します。
2. Bean に対するセッション スタンザの最後、`<transaction-type>` の後に、次の記述を追加します。

```
<security-role-ref>
    <role-name>admin</role-name>
    <role-link<admin</role-link>
</security-role-ref>
```

3. `ejb-jar.xml` の `<assembly-descriptor>` スタンザの先頭に以下の記述を追加し、EJB メソッドにアクセスできるロールを指定します。

```
<security-role>
<description></description>
    <role-name>admin</role-name>
</security-role>
<method-permission>
    <role-name>admin</role-name>
    <method>
           <ejb-name>containerManaged</ejb-name>
       <method-name>*</method-name>
    </method>
</method-permission>
```

4. weblogic-ejb-jar.xml の weblogic-ejb-jar スタンザの最後に以下の記述を追加し、セキュリティ レルム内の特定のユーザおよびグループに、ロール名をマップします。

```
<security-role-assignment>
  <role-name>admin</role-name>
        <principal-name>Accounting Managers</principal-name>
      <principal-name>HR Managers</principal-name>
      <principal-name>system</principal-name>
</security-role-assignment>
```

注意 : プリンシパルとしては、セキュリティ レルム内のユーザまたはグループを指定できます。

5. ビルド スクリプトを使用して、Bean をビルドし直します。

注意 : サービス パックで EJB に関する修正が行われた場合は、サービス パックの jar ファイルをビルド スクリプトのクラスパスの最初に追加して、修正を有効にする必要があります。

6. InitialContext をプログラミングするときは、ユーザと資格を使用するように Client.java を変更します。

7. 次のコマンドを使用して、クライアントを実行します。

```
java examples.ejb.basic.containerManaged.Client
"t3://WebLogicURL:Port" user password
```

パラメータは省略可能ですが、指定した場合は、次の順序で解釈されます。

a. url - サーバ ユーザの URL。「t3://localhost:7001」など。

b. user - ユーザ名。デフォルトは Null。

c. password - ユーザのパスワード。デフォルトは Null。

d. accountID - テストするアカウント ID の文字列。デフォルトは「10020」。

8 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

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

1. 6-3 ページの「WebLogic Server の EJB デプロイメント ファイル」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. 6-12 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。

   コンテナをコンパイルすると、JAR ファイルはデプロイメント記述子で指定したデプロイメント ディレクトリに配置されます。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配置します。

   ```
   wlserver\config\mydomain\applications
   ```

   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. 右ペインで、[EJB] 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右ペインに表示されます。

異なるアプリケーションでの EJB のデプロイメント

複数の異なったアプリケーションにリモート呼び出しによって EJB をデプロイするとき、EJB を呼び出すために参照による呼び出しは使用できません。代わりに、値による呼び出しを使用します。一般に、相互に対話するコンポーネントは参照で呼び出せるように同じアプリケーション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、常に値で渡す必要があります。

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、次の作業を行えます。

- 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 Administartion Console から EJB をデプロイする場合でも、更新する場合でも、自動動的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 にデプロイするには、次の手順に従います。

次のコマンドを使用します。

```
% java weblogic.deploy -port port_number -host host_name
      deploy password name source
```

各値の説明は次のとおりです。

- *name* はこの EJB デプロイメント ユニットに割り当てる文字列です。
- *source* はデプロイする EJB JAR ファイルの絶対パスとファイル名、または EJB デプロイメント ディレクトリの絶対パスです。

次に例を示します。

```
% java weblogic.deploy -port 7001 -host localhost deploy
       weblogicpwd CMP_example
       c:\weblogic\myserver\unjarred\containerManaged\
```

固定された EJB のデプロイで必要になる特別な手順

.jar ファイルにコンパイルされていないクラスやインタフェースが含まれる場合、クラスタ内の単一のサーバ インスタンスに EJB をデプロイまたは再デプロイしようとすると (固定されたデプロイメント)、問題が発生することが確認されています。

デプロイメントの際に、コンパイルされていない EJB は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、コンパイルが行われるのは、EJB がデプロイされたサーバ インスタンスだけです。その結果、EJB のデプロイ対象になっていないサーバ インスタンスには、EJB を呼び出すために必要なクラスでコンパイルの際に生成されるものが存在していません。別のサーバ インスタンス上のクライアントが固定された EJB を呼び出そうとすると失敗し、RMI レイヤで Assertion エラーが送出されます。

クラスタ内の単一のサーバ 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、デプロイする前に appc または ejbc を使って EJB をコンパイルし、生成されるクラスがすべてのサーバ インスタンスにコピーされて、クラスタ内の全ノードで利用できるようにする必要があります。

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。

- コマンドラインを使用する場合

1. ローカルの WebLogic Server にデプロイする EJB をリストするには、次のように入力します。

   ```
   % java weblogic.deploy list password
   ```

   password は WebLogic Server のシステム アカウントのパスワードです。

2. リモート サーバにデプロイされている EJB をリストするには、port および host オプションを次のように指定します。

```
% java weblogic.deploy -port port_number -host host_name
       list password
```

- WebLogic Server の Administration Console を使用する場合

1. Console の左ペインで [デプロイメント] の [EJB] を選択します。
2. サーバにデプロイされている EJB のリストを表示します。

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。

コマンドラインを使用する場合

次のように、割り当て済みのデプロイメント ユニット名を参照します。

```
% java weblogic.deploy -port 7001 -host localhost undeploy
       weblogicpwd CMP_example
```

WebLogic Server の Administration Console を使用する場合

1. Console の左ペインの [デプロイメント] から [EJB] を選択します。
2. リストからアンデプロイする EJB をクリックします。
3. 右ペインのダイアログから [コンフィグレーション] タブを選択し、アンデプロイ ボックスのチェックをはずします。

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。

デプロイ済み EJB の更新

WebLogic Server にデプロイ済みの `ejb.jar` ファイルまたはデプロイメント ディレクトリの内容を更新すると、更新は次のいずれかの操作を行うまで、WebLogic Server に反映されません。

- サーバを再起動します（JAR またはディレクトリを自動的にデプロイする場合）。
- WebLogic Server Administration Console を使用して、EJB デプロイメントを更新します。

EJB デプロイメントを更新すると、EJB プロバイダがデプロイ済みの EJB 実装クラスを変更し、再コンパイルしてから、動作中のサーバの実装クラスを「更新」できるようになります。

weblogic.deploy の更新と対象

WebLogic Server 6.1 では、アプリケーションの対象になっているサーバ インスタンスの 1 つでそのアプリケーションを更新すると、対象になっているすべてのサーバでアプリケーションが更新されます。たとえば、アプリケーションの対象がクラスタの場合、クラスタ化されたサーバ インスタンスの 1 つでアプリケーションを更新すると、アプリケーションはクラスタの全メンバで更新されます。同様に、クラスタとスタンドアロン サーバ インスタンスがアプリケーションの対象になっている場合は、スタンドアロン サーバのインスタンスでアプリケーションを更新すると、クラスタでもアプリケーションが更新されます。また、逆の場合も同様の処理が行われます。

アプリケーションまたはコンポーネントを対象のサーバ インスタンス群のサブセットで選択的に更新する必要がある場合は、アプリケーションのユニークなインスタンスを異なる対象にデプロイします。

更新処理

ロード済みの実装を更新すると、直ちに、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、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 の実装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。

EJB 全体が再ロードされます。EJB JAR の一部を再デプロイすることはできません。

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、更新済みの EJB 実装クラスが使用されます。

注意： 更新できるのは、6-15 ページの「WebLogic Server への EJB クラスのロード」で説明されているように、EJB 実装クラスだけです。EJB のパブリック インタフェース、またはそのパブリック インタフェースが使用するサポート クラスを更新することはできません。EJB のパブリック クラスを変更してから EJB を更新しようとした場合、クライアントがその EJB インタフェースを次に使用するときに、WebLogic Server には、クラスの変更が互換性を持たないことを示すエラーが表示されます。

EJB の更新

EJB 実装クラスを更新するには次の手順に従います。

コマンドラインを使用する場合

update 引数を使用して、アクティブな EJB デプロイメント名を指定します。

```
% java weblogic.deploy -port 7001 -host localhost update
       weblogicpwd CMP_example
```

WebLogic Server の Administration Console を使用する場合

1. Console の左ペインの [デプロイメント] から [EJB] を選択します。
2. リストの中で更新する EJB をクリックします。
3. 右ペインのダイアログから [コンフィグレーション] タブを選択し、デプロイ ボックスをチェックして EJB を更新します。

更新できるのは EJB 実装クラスだけで、パブリック インタフェースまたはパブリック サポート クラスは更新できません。

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、9-1 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. コンパイル済み ejb.jar ファイルを wlserver\config\mydomain\applications にコピーします。

（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。

注意： 自動再デプロイメント機能は動的デプロイメントを利用するので、WebLogic Server が再デプロイできるのは EJB の実装クラスだけです。EJB のパブリック インタフェースを再デプロイすることはできません。

未コンパイルの EJB ファイルのデプロイメント

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。コンパイル済みクラスは、`wlserver\config\mydomain\applications` の新規 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` 内の未コンパイル 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。

注意： 自動再デプロイメント機能は動的デプロイメントを利用するので、WebLogic Server が再デプロイできるのは EJB の実装クラスだけです。EJB のパブリック インタフェースを再デプロイすることはできません。

9 WebLogic Server EJB のユーティリティ

以下の節は、WebLogic Server EJB 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。

- ejbc (`weblogic.ejbc`)
- DDConverter (`weblogic.ejb.utils.DDConverter`)
- deploy (`weblogic.deploy`)

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` コマンドライン 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 Server のコマンドライン ユーティリティである `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します。
- 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します。
- EJB 用の WebLogic Server コンテナ クラスを生成します。
- 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 コードを作成します。

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。

注意： weblogic.ejbc のバージョン不一致の問題があると、EJB のデプロイメントに関して問題が発生することがあります。WebLogic Server を起動すると、コンテナ クラスのコンパイルにどのバージョンの weblogic.ejbc が使用されたかのチェックが行われます。クラスのコンパイルに使用された weblogic.ejbc のバージョンが現在実行中のバージョンと異なる場合、EJB はデプロイされません。この問題を避けるためには、不必要なクラスをクラスパスに含めないようにします。

ejbc の構文

```
$ java weblogic.ejbc [options] <source 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

次の表は、weblogic.ejbc の引数の一覧です。

引数	説明
<source 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 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

次の表は、weblogic.ejbc のコマンドライン オプションの一覧です。

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。
-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。
-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。
-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
-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。
-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 はオーバーライドされる。

ejbc の例

次の例では、`c:\wlserver\samples\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:\wlserver\config\examples\applications` 内に置かれます。

```
$ java weblogic.ejbc -compiler javac
c:\wlserver\samples\examples\ejb\basic\containerManaged\build\std
_ejb_basic_containerManaged.jar
c:\wlserver\config\examples\ejb_basic_containerManaged.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。

```
$ java weblogic.ejbc -normi
c:\wlserver\samples\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 WebLogic Server 6.x バージョンに準拠した 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`

  この XML ファイルには、Sun Microsoft 固有の EJB デプロイメント記述子が含まれます。

- `weblogic-ejb-jar-.xml`

  この XML ファイルには、WebLogic 固有の EJB デプロイメント記述子が含まれます。

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

  この XML ファイルには、WebLogic 固有のコンテナ管理永続性（CMP）デプロイメント記述子が含まれます。

DDConverter の変換オプション

`DDConverter` コマンドライン ユーティリティでは、オプションの指定によって次の変換を行うことができます。

- WebLogic Server（WLS）の以前のバージョンからの Bean の変換
- EJB 仕様の以前のバージョンからの CMP Bean および非 CMP Bean の変換

次の表は、DDConverter で使用できる各種の変換オプションの一覧です。

DDConverter ユーティリティの変換オプション

WLS		EJB 非 CMP		EJB CMP	
変換前	**変換後**	**変換前**	**変換後**	**変換前**	**変換後**
WLS 4.5 - WLS 6.x		**注 1** を参照		EJB CMP 1.0 - EJB CMP 1.1	**注 2** を参照
WLS 4.5 - WLS 6.x		EJB 1.1 - EJB 2.0		EJB CMP 1.0 - EJB CMP 2.0	
WLS 5.x - WLS 6.x		EJB 1.1 - EJB 2.0		**注 3** を参照	

注意 1: EJB 1.1 の非 CMP デプロイメント記述子は EJB 2.0 の非 CMP デプロイメント記述子と同じであるため、非 CMP EJB 1.0 Bean から非 CMP EJB 1.1 Bean への変換は不要です。

注意 2: EJB CMP 1.0 から EJB CMP 1.1 への変換を行うには、DDConverter のコマンドライン オプションである `-EJBVer` を使用します。このオプションについての説明は、9-8 ページの「DDConverter のオプション」を参照してください。

注意 3: WLS 5.x の CMP 1.1 Bean と WLS 6.x の CMP 1.1 Bean は異なっていますが、WLS 5.1 の CMP 1.1 Bean は、ソース コードを変更することなく WebLogic Server 6.x で実行できます。

DDConverter を使用した後は、常に Bean を再コンパイルするようにします。`weblogic.ejbc` を使用してから、新しく生成された 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、コードが EJB 仕様に準拠していることが保証され、また、サーバ起動時の再コンパイル プロセスを省略できることにより時間の節約にもなります。

- WLS 4.5 の EJB 1.0 Bean を WLS 6.x の EJB 1.1 Bean に変換するとき、`DDConverter` への入力は WebLogic 4.5 のデプロイメント記述子テキストです。出力は、WebLogic 6.x のデプロイメント記述子だけを含む JAR ファイルです。9-7 ページの「DDConverter による EJB の変換」の手順に従ってソース コードをさらに変更しなければならないかどうかを確認するには、`weblogic-ejbc` を実行します。「DDConverter ユーティリティの変換オプション」表の 1 行目を参照してください。

- WLS 4.5 の EJB 1.1 Bean を WLS 6.x の EJB 2.0 Bean に変換するとき、`DDConverter` への入力は WebLogic Server 4.5 のデプロイメント記述子テキストです。出力は、WebLogic 6.x のデプロイメント記述子だけを含む JAR ファイルです。9-7 ページの「DDConverter による EJB の変換」の手順に従ってソース コードをさらに変更しなければならないかどうかを確認するには、`weblogic-ejbc` を実行します。「DDConverter ユーティリティの変換オプション」表の 2 行目を参照してください。

- WLS 6.x は下位互換性を備えているため、WLS 5.x の EJB 1.1 Bean は、ソース コードを変更することなく WLS 6.x にデプロイできます。WLS 6.x は、以前のバージョンの WLS で作成された Bean を検出し、再コンパイルしてからデプロイします。ただし、WLS 5.x の EJB 1.1 Bean は、DDConverter を使用して WLS 6.x の 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 の EJB 1.1 Bean を WLS 6.x の EJB 2.0 Bean に変換するとき、`DDConverter` への入力は WebLogic 5.1 の 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 ファイルが含まれています。出

力される JAR ファイルの内容は、WebLogic 6.0 のデプロイメント記述子ファイルと、必要なすべてのクラス ファイルです。「DDConverter ユーティリティの変換オプション」表の 3 行目を参照してください。

非 CMP Bean は、ソース コードをほとんどまたは一切変更することなく EJB 2.0 Bean に変換できます。この変換を行うには、`output.jar` ファイルに対して 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 については、9-7 ページの「DDConverter による EJB の変換」の手順に従ってソース コードに変更を行う必要があります。

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. 9-7 ページの「DDConverter の構文」で示されているコマンドライン形式を使用して、`DDConverter` への入力となる EJB デプロイメント記述子ファイルを指定します。

   出力は JAR ファイルです。

2. 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 コードを修正します。
4. 抽出した XML デプロイメント記述子を使用して、修正済みの java ファイルをコンパイルし、`weblogic.ejbc` を使用して JAR ファイルを作成します。
5. JAR ファイルをデプロイします。

DDConverter の構文

```
$ java weblogic.ejb20.utils.DDConverter [options] file1 [file2...]
```

DDConverter の引数

DDConverter は、*file1 [file2...]* のような形式の引数を取ります。この引数に指定するファイルは次のどちらかです。

- 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、新規の ejb-name 要素を出力される ejb-jar.xml ファイルで定義します。

DDConverter のオプション

次の表は、DDConverter のコマンドライン オプションの一覧です。

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。
-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 の EJB 1.1 Bean を、WLS 6.x の EJB 2.0 Bean に変換します。

JAR ファイルは *destDir* サブディレクトリに作成されます。

```
$ java weblogic.ejb20.utils.DDConverter -d destDir Employee.jar
```

`Employee.jar` は、WLS 5.x の EJB 1.1 Bean の JAR ファイル です。

deploy

`weblogic.deploy` コマンドライン ユーティリティを使用して、EJB 準拠の JAR ファイル（JAR の EJB）を、実行中の WebLogic Server のインスタンスにデプロイします。

deploy の構文

```
$ java weblogic.deploy [options] [list|deploy|undeploy|update]
password {name} {source}
```

deploy の引数

次の表は、weblogic.deploy のコマンドライン引数の一覧です。

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。
`delete`	EJB デプロイメント ユニットを削除する。

`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 ユニットを再デプロイする。 **注意：** アプリケーションまたはコンポーネントの対象になっているサーバ インスタンスの 1 つでそのアプリケーションまたはコンポーネントを更新すると、対象になっているすべてのサーバ上で同じ更新が行われる。たとえば、アプリケーションの対象がクラスタの場合、クラスタ化されたサーバ インスタンスの 1 つでアプリケーションを更新すると、アプリケーションはクラスタの全メンバで更新される。同様に、クラスタとスタンドアロン サーバ インスタンスがアプリケーションの対象になっている場合は、スタンドアロン サーバのインスタンスでアプリケーションを更新すると、クラスタでもアプリケーションが更新される。また、逆の場合も同様の処理が行われる。
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

次の表は、weblogic.deploy のコマンドライン オプションの一覧です。

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。

10 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 5.1 および EJB 6.0 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。

注意： 6.x バージョンの WebLogic Server では、6.0 デプロイメント記述子を使用してください。

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 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 6.0.0 EJB//EN' 'http://www.bea.com/servers/wls600/dtd/weblogic-ejb-jar.dtd'`

XML ファイル	PUBLIC 要素の文字列
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' '
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 6.0.0 EJB//EN'
'http://www.bea.com/servers/wls600/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 ファイル用の新しいパブリック DTD の場所が示されています。

- weblogic-ejb-jar.xml 6.0 DTD

  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

  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

  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 仕様の一部として保持されています。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

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。

6.0 の weblogic-ejb-jar.xml デプロイメント記述子ファイルの構造

WebLogic Server 6.0 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。どちらのバージョンのデプロイメント記述子も EJB コンテナで使用できますが、weblogic-ejb-jar.xml には、WebLogic Server 6.0 バージョンと WebLogic Server バージョン 5.1 の間で違いがあります。

WebLogic Serve 6.0 の weblogic-ejb-jar.xml には、ステートフル セッション EJB レプリケーションの有効化、エンティティ EJB ロック動作のコンフィグレーション、メッセージ駆動型 Bean への JMS キューとトピック名の割り当てを行うための要素が含まれています。

WebLogic Server 6.0 の weblogic-ejb-jar.xml の最上位要素は次のとおりです。

- 説明
- 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
  - jndi-name

- security-role-assignment
- transaction-isolation

6.0 の weblogic-ejb-jar.xml デプロイメント記述子要素

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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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

10-78 ページの「stateful-session-descriptor」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
デプロイメントファイル：	weblogic-ejb-jar.xml

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）

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>
```

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
デプロイメント ファイル:	weblogic-ejb-jar.xml

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>
```

concurrency-strategy

指定できる値:	`Exclusive | Database | ReadOnly`
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`

デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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 3 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します。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ます。`Database` 同時方式では、WebLogic Server は、トランザクションに関するエンティティ EJB の中間結果をキャッシュしません。これは現在のデフォルト オプションです。
- `ReadOnly` は、変更されることのないエンティティ EJB を示します。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。

Exclusive および Database のロック動作の詳細については、4-39 ページの「エンティティ EJB のロック サービス」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、4-17 ページの「read-only 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 クラスを読み取り専用のエンティティ 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>
```

db-is-shared

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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

機能

`db-is-shared` 要素はエンティティ Bean に対してのみ有効です。`true` に設定すると、WebLogic Server では、EJB データがトランザクションで変更された可能性があると見なし、各トランザクションの開始時にデータを再ロードします。`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、4-12 ページの「db-is-shared を使用した ejbLoad() の呼び出しの制限」を参照してください。

例

10-55 ページの「persistence」を参照してください。

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`

デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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、4-11 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
デプロイメントファイル:	weblogic-ejb-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。

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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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

10-48 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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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 の名前の中で一意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

10-49 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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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します。このリソース参照は、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。
- `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 名を指定します。

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
デプロイメント ファイル:	weblogic-ejb-jar.xml

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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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`

デプロイメントファイル:	`weblogic-ejb-jar.xml`

機能

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（同じサーバ以外から）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>
```

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
デプロイメント ファイル:	weblogic-ejb-jar.xml

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、4-3 ページの「WebLogic Server における EJB のライフサイクル」を参照してください。

例

```
<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>
        <lifecycle>...</lifecycle>
        <persistence>...</persistence>
        <entity-clustering>...</entity-clustering>
</entity-descriptor>
```

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`
デプロイメントファイル:	`weblogic-ejb-jar.xml`

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、`entity-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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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`
- `lifecycle`
- `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>
        <lifecycle>...</lifecycle>
        <persistence>...</persistence>
        <entity-clustering>...</entity-clustering>
</entity-descriptor>
```

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`

デプロイメント ファイル:	`weblogic-ejb-jar.xml`

機能

`finders-call-ejbload` 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決定します。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>
```

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`

デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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.extensions.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

10-24 ページの「entity-clustering」と 10-76 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 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`

デプロイメント ファイル:	`weblogic-ejb-jar.xml`

機能

`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。

例

10-24 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
デプロイメント ファイル:	weblogic-ejb-jar.xml

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。

例

10-24 ページの「entity-clustering」と 10-76 ページの「stateful-session-clustering」を参照してください。

idle-timeout-seconds

指定できる値:	1 ～ *maxSeconds*。*maxSeconds* は int 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
デプロイメント ファイル:	weblogic-ejb-jar.xml

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、4-3 ページの「WebLogic Server における EJB のライフサイクル」を参照してください。

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>
```

initial-beans-in-free-pool

指定できる値:	0 ～ *maxBeans*
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
デプロイメント ファイル:	weblogic-ejb-jar.xml

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、リクエストが来てから新しいインスタンスを生成せずに Bean に対する初期のリクエストが可能になるため、EJB の初期の応答時間が短縮されます。

例

10-61 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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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>
```

invalidation-target

指定できる値:	
デフォルト値:	
要件:	対象の ejb-name は読み出し専用エンティティ 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
デプロイメントファイル:	weblogic-ejb-jar.xml

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、無効化する読み出し専用エンティティ 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`

デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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、4-13 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:	Serializable | ReadCommitted | ReadUncommitted | RepeatableRead
デフォルト値:	なし
要件:	省略可能。
親要素:	weblogic-enterprise-bean, transaction-isolation
デプロイメント ファイル:	weblogic-ejb-jar.xml

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。

- TRANSACTION_READ_UNCOMMITTED : 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。

- TRANSACTION_READ_COMMITTED：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。
- TRANSACTION_REPEATABLE_READ：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。
- TRANSACTION_SERIALIZABLE：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。

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

10-89 ページの「transaction-isolation」を参照してください。

jms-client-id

指定できる値:	なし
デフォルト値:	EJB に対する EJB 名。
要件:	JMS トピックに対する恒久サブスクリプションに必須。
親要素:	`weblogic-enterprise-bean`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

機能

`jms-client-id` 要素は、メッセージ駆動型 Bean に関連付けられるクライアント ID を指定します。JMS トピックに対する恒久サブスクリプションでは、この ID が必要です。

JMS の仕様では、JMS コンシューマは関連付けられる ID を指定できます。恒久サブスクリプションを使用するメッセージ駆動型 Bean では、関連付けられるクライアント ID が必要です。独立した接続ファクトリを使用する場合は、接続ファクトリにクライアント 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。

関連付けられるクライアント ID 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している場合は、メッセージ駆動型 Bean はクライアント ID として `jms-client-id` を使用します。

例

次の例は、`jms-client-id` 要素の使い方を示したものです。

```
<jms-client-id>MyClientID</jms-client-id>
```

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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

機能

`jms-polling-interval-seconds` 要素は、WebLogic Server が JMS 送り先への再接続を試みる間隔の秒数を決定します。

各メッセージ駆動型 Bean は、関連付けられている JMS 送り先をリスンします。JMS 送り先が別の WebLogic Server インスタンスまたは外部の JMS プロバイダに存在している場合、その JMS 送り先にアクセスできなくなることがあります。このような場合、EJB コンテナは、自動的に、JMS サーバへの再接続を試みます。JMS サーバが再び稼働したなら、メッセージ駆動型 Bean は再び JMS メッセージを受信できます。

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

次の例は、`jms-pollig-interval-seconds` 要素の使い方を示したものです。

```
<jms-polling-interval-seconds>5</jms-polling-interval-seconds>
```

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`

デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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

10-70 ページの「resource-description」と 10-18 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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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>
```

lifecycle

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:	`lifecycle` スタンザは省略可能。

親要素:	`weblogic-enterprise-bean`,
	`entity-descriptor`
	および
	`weblogic-enterprise-bean`
	`stateful-session-descriptor`

デプロイメントファイル:	`weblogic-ejb-jar.xml`

機能

`lifecycle` 要素は、WebLogic Server 内のステートフルおよびエンティティ EJB インスタンスのライフサイクルに関するオプションを定義します。現在、`lifecycle` 要素には、`passivation-strategy` という 1 つの要素だけがあります。

例

次の例では、`lifecycle` 要素の構造を示します。

```
<entity-descriptor>
       <lifecycle>
              <passivation-strategy>...</passivation-strategy>
       </lifecycle>
</entity-descriptor>
```

max-beans-in-cache

指定できる値:	1 ～ *maxBeans*
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
デプロイメント ファイル:	weblogic-ejb-jar.xml

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、4-39 ページの「エンティティ EJB のロック サービス」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 ～ *maxBeans*
デフォルト値:	*max Int*
要件:	省略可能。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, pool
デプロイメント ファイル:	weblogic-ejb-jar.xml

機能

WebLogic Server は、すべてのステートレス 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。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、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

10-61 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
デプロイメント ファイル:	weblogic-ejb-jar.xml

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
デプロイメント ファイル:	weblogic-ejb-jar.xml

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`
デフォルト値:	なし
要件:	省略可能。

親要素:	`weblogic-enterprise-bean`
	`transaction-isolation`
	`method`

デプロイメント ファイル:	`weblogic-ejb-jar.xml`

機能

`method-intf` 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。この要素は、EJB のホームとリモート インタフェースで同じシグネチャを持つメソッドを区別する必要がある場合にのみ使用します。

例

10-49 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
デプロイメント ファイル:	weblogic-ejb-jar.xml

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

10-49 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`

デプロイメント ファイル:	`weblogic-ejb-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を指定します。

例

10-53 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
デプロイメント ファイル:	weblogic-ejb-jar.xml

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>
```

passivation-strategy

指定できる値:	`default	transaction`
デフォルト値:	`default`
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, lifecycle`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

機能

`passivation-strategy` 要素は、WebLogic Server がキャッシュにエンティティ EJB の中間的な状態を保持するかどうかを決定します。詳細については、4-39 ページの「エンティティ EJB のロック サービス」を参照してください。

例

次の例では、WebLogic Server のロック動作およびキャッシュ動作に戻します。

```
<entity-descriptor>
       <lifecycle>
              <passivation-strategy>default</passivation-strategy>
       </lifecycle>
</entity-descriptor>
```

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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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`
- `persistence-type`
- `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>
               <persistence-type>...</persistence-type>
               <db-is-shared>false</db-is-shared>
               <persistence-use>...</persistence-use>
       </persistence>
</entity-descriptor>
```

persistence-type

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`

デプロイメントファイル:	`weblogic-ejb-jar.xml`

機能

`persistence-type` 要素は、エンティティ 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-type` スタンザを定義できます。デプロイメントでは、`persistence-use` で定義した永続性タイプのみが実際に使用されます。

`persistence-type` には、永続性タイプを示す要素がいくつか含まれます。

- type-identifier
- type-version
- type-storage

例

次の例では、persistence-type スタンザのサンプルを示します。

```
<persistence>
       <persistence-type>
<type-identifier>WebLogic_CMP_RDBMS</type-identifier>
              <type-version>5.1.0</type-version>
<type-storage>META-INF\weblogic-cmp-rdbms-jar.xml</type-storage>
       </persistence-type>
</persistence>
```

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`

デプロイメント ファイル:	`weblogic-ejb-jar.xml`

機能

`persistence-use` 要素は persistence-type と似ていますが、この要素はデプロイ中に実際に使用される永続性サービスを定義します。`persistence-use` では、`persistence-type` で定義した type-identifier 要素と type-version 要素を使用してサービスを指定します。

例

persistence-type で定義した WebLogic Server RDBMS ベースの永続性サービスを使用して EJB をデプロイするには、次の `persistence-use` スタンザを使用します。

```
<persistence-use>
       <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
       <type-version>5.1.0</type-version>
</persistence-use>
```

persistent-store-dir

指定できる値:	完全修飾ファイルシステム パス
デフォルト値:	なし
要件:	省略可能。
親要素:	`weblogic-enterprise-bean` `stateful-session-descriptor`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

機能

`persistent-store-dir` 要素は、WebLogic Server が、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。

例

10-78 ページの「stateful-session-descriptor」を参照してください。

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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

機能

`pool` 要素は、ステートレス セッション EJB およびメッセージ駆動型 Bean 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。オプションは以下のとおりです。

- `max-beans-in-free-pool`
- `initial-beans-in-free-pool`

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`
デプロイメントファイル:	`weblogic-ejb-jar.xml`

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

10-74 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
デプロイメントファイル:	weblogic-ejb-jar.xml

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 ～ *maxSeconds*。*maxSeconds* は int 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
デプロイメント ファイル:	weblogic-ejb-jar.xml

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。デフォルトでは、read-timeout-seconds は 600 に設定されており、WebLogic Server は、Bean がキャッシュされた場合にのみ ejbLoad() を呼び出します。

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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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:	`None`
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`

デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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決定します。`InMemory` を指定した場合、EJB の状態はレプリケートされます。`InMemory` を指定した場合、状態はレプリケートされます。

詳細については、4-25 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

10-76 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
デプロイメント ファイル:	weblogic-ejb-jar.xml

機能

res-env-ref-name はリソース環境参照名を指定します。

例

10-70 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
デプロイメント ファイル:	weblogic-ejb-jar.xml

機能

res-ref-name は resourcefactory 参照名を指定します。このリソース参照は、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

10-70 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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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
デプロイメント ファイル:	weblogic-ejb-jar.xml

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

10-74 ページの「security-role-assignment」を参照してください。

run-as-identity-principal

注意: この要素は廃止されています。以下の情報は、下位互換性のためだけに使用してください。

指定できる値:	ejb-jar.xml で定義した、ID として使用するプリンシパル

デフォルト値:	なし
要件:	security-role-assignment で必須。
親要素:	weblogic-enterprise-bean
デプロイメント ファイル:	weblogic-ejb-jar.xml

機能

run-as-identity-principal 要素は、ejb-jar.xml で設定した security-identity.run-as-specified-identity を持つ Bean の ID として使用するプリンシパルを指定します。

この要素で指定したプリンシパルは、run-as-specified--identity ロールにマップされるプリンシパルのいずれかである必要があります

例

run-as-identity-principal スタンザには、以下のように要素を追加できます。

```
<weblogic-ejb-jar>
    <weblogic-enterprise-bean>
        <run-as-identity-principal>Fred</run-as-identity-principal>
    </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

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
デプロイメント ファイル:	weblogic-ejb-jar.xml

機能

security-role-assignment スタンザは、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、以下の要素のうち 1 つまたは複数を指定できます。

```
<security-role-assignment>
       <role-name>PayrollAdmin</role-name>
       <principal-name>Tanya</principal-name>
       <principal-name>system</principal-name>
       ...
</security-role-assignment>
```

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
デプロイメント ファイル:	weblogic-ejb-jar.xml

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、4-3 ページの「WebLogic Server における 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-descriptor 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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

機能

`stateful-session-clustering` 要素には、WebLogic Server がクラスタ内のステートフル 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以下のオプションを指定します。

- `home-is-clusterable`
- `home-load-algorithm`
- `home-call-router-class-name`
- `replication-type`

例

次の例では、entity-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-n
ame>
        <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
デプロイメント ファイル:	weblogic-ejb-jar.xml

機能

stateful-session-descriptor 要素は、WebLogic Server のステートフル セッション EJB 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。

- stateful-session-cache
- lifecycle
- persistent-store-dir
- stateful-session-clustering
- allow-concurrent-calls

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>
       <lifecycle>...</lifecycle>
       <persistence>...</persistence>
       <allow-concurrent-calls>...</allow-concurrent-calls>

<persistent-store-dir>/weblogic/myserver</persistent-store-dir>

<stateful-session-clustering>...</stateful-session-clustering>
</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`

デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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.extensions.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

10-85 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`

デプロイメント ファイル:	`weblogic-ejb-jar.xml`

機能

`stateless-bean-is-clusterable` が `true` の場合、EJB をクラスタ内の複数の WebLogic Server からデプロイできます。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。

例

10-85 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
デプロイメント ファイル:	weblogic-ejb-jar.xml

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

10-85 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`

デプロイメント ファイル:	`weblogic-ejb-jar.xml`

機能

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。

例

10-85 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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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以下のオプションを指定します。

- `stateless-bean-is-clusterable`
- `stateless-bean-load-algorithm`
- `stateless-bean-call-router-class-name`
- `stateless-bean-methods-are-idempotent`

例

次の例では、stateless-clustering スタンザの構造を示します。

```
<stateless-clustering>

<stateless-bean-is-clusterable>true</stateless-bean-is-clusterable>

<stateless-bean-load-algorithm>random</stateless-bean-load-algorithm>

<stateless-bean-call-router-class-name>beanRouter</stateless-bean-call-router-class-name>

<stateless-bean-methods-are-idempotent>true</stateless-bean-methods-are-idempotent>

</stateless-clustering>
```

stateless-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* 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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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 パラメータを定義します。

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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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。`trans-timeout-seconds`.

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-bean`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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>Serializable</isolation-level>
       <method>
              <description>...</description>
              <ejb-name>...</ejb-name>
              <method-intf>...</method-intf>
              <method-name>...</method-name>
              <method-params>...</method-params>
       </method>
</transaction-isolation>
```

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
デプロイメント ファイル:	weblogic-ejb-jar.xml

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

10-88 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
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-type`
	および
	`weblogic-enterprise-bean,`
	`entity-descriptor,`
	`persistence`
	`persistence-use`

デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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、正しい `type-identifier` の詳細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完全な `persistence-type` の定義の例については、10-57 ページの「persistence-type」を参照してください。

type-storage

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-type`

デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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 `.jar` デプロイメント 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完全な `persistence-type` の定義の例については、10-57 ページの「persistence-type」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-type`
	および
	`weblogic-enterprise-bean,`
	`  entity-descriptor,`
	`  persistence`
	`  persistence-use`

デプロイメント ファイル:	`weblogic-ejb-jar.xml`

機能

`type-version` 要素では、指定した永続性タイプのバージョンを指定します。

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完全な `persistence-type` の定義の例については、「persistence-type」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
デプロイメントファイル:	`weblogic-ejb-jar.xml`

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`
デプロイメント ファイル:	`weblogic-ejb-jar.xml`

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、サンプルの caching-descriptor スタンザであり、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、4-1 ページの「WebLogic Server EJB コンテナとサポートされるサービス」の 4-6 ページの「ステートフル セッション EJB インスタンスのアクティブ化と使用」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、4-7 ページの「ステートフル セッション EJB インスタンスの削除」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、4-3 ページの「WebLogic Server における EJB のライフサイクル」を参照してください。

定義されていない場合、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。詳細については、4-15 ページの「エンティティ EJB の read-only への設定」を参照してください。

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-type>
      <type-identifier>. . .</type-identifier>
      <type-version>. . .</type-version>
      <type-storage>. . .</type-storage>
   </persistence-type>
   <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、4-13 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、4-11 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-type

persistence-type 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-type エントリを定義できます。デプロイメントでは、10-102 ページの「persistence-use」で定義した永続性タイプのみが実際に使用されます。

persistence-type 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します。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します。

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します。パスは、EJB の *.jar* デプロイメント 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-type スタンザの例です。

```
<persistence-type>
    <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-type>
```

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、4-12 ページの「db-is-shared を使用した ejbLoad() 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。

persistence-use

persistence-use プロパティは persistence-type とほぼ同じですが、このプロパティはデプロイ中に実際に使用される永続性サービスを定義します。persistence-use では、persistence-type で定義されている type-identifier 要素と type-version 要素を使用してサービスを指定します。

たとえば、persistence-type で定義されている WebLogic Server RDBMS ベースの永続性サービスを使用して実際に EJB をデプロイする場合、persistence-use スタンザは次のようになります。

```
<persistence-use>
   <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
   <type-version>5.1.0</type-version>
</persistence-use>
```

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.extensions.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

このプロパティは home-is-clusterable 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-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 trans-timeout-seconds という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します。このリソース参照は、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。
- 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 名を指定します。

ejb-reference-description

以下の要素で、各 ejb-reference-description を定義します。

- ejb-ref-name は EJB 参照名を指定します。このリソース参照は、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。
- jndi-name では、WebLogic Server で使用可能な実際の EJB の JNDI 名を指定します。

enable-call-by-reference

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する。

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（同じサーバ以外から）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>Serializable</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。

- TRANSACTION_READ_UNCOMMITTED：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。
- TRANSACTION_READ_COMMITTED：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。
- TRANSACTION_REPEATABLE_READ：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。
- TRANSACTION_SERIALIZABLE：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。

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です。
- 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、省略可能な要素です。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（*）を指定します。
- 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です。各パラメータのタイプは、method-params スタンザ内の method-params 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、以下の要素のうち 1 つまたは複数を指定できます。

- `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です。
- `principal-name` では、実際の WebLogic Server プリンシパルの名前を指定します。

11 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 5.1 および EJB 6.0 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

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 6.0.0 EJB RDBMS20 Persistence//EN' 'http://www.bea.com/servers/wls600/dtd/weblogic-rdbms 20-persistence-600.dtd'`

XML ファイル	PUBLIC 要素の文字列
weblogicmp-rdbms -jar.xml	'-//BEA Systems, Inc.//DTD WebLogic 5.1.0 EJB RDBMS Persistence//EN' http://www.bea.com/servers/wls510/dtd/weblogic-rdbms-persistence.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' '
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 6.0.0 EJB RDBMS20
Persistence//EN'
'http://www.bea.com/servers/wls600/dtd/weblogic-rdbms20-persisten
ce-600.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 6.0 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

- weblogic-cmp-rdbms-jar.xml 5.1 DTD

  weblogic-rdbms-persistence.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 永続性サービスの使用に必要な weblogic-rdbms-persistence.xml ファイルの作成に使用されます。サー

ドパーティの永続性ベンダは、この DTD に従った XML デプロイメント ファイルを作成することもできます。この DTD ファイルは、http://www.bea.com/servers/wls510/dtd/weblogic-rdbms-persistence.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

  `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 仕様の一部として保持されています。`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

  `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。

6.0 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` 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 2.0 コンテナでは、WebLogic Server バージョン 5.1 で提供されたものとは異なるバージョンの `weblogic-cmp-rdbms-jar.xml` を使用します。詳細については、エンティティ EJB のロック サービスを参照してください。

WebLogic Server バージョン 6.0 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 6.0 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-name
              field-name
              field-map
              field-group
              weblogic-query
              delay-database-insert-until
              automatic-key-generation

       weblogic-rdbms-relation
              relation-name
       table-name
       weblogic-relationship-role
```

6.0 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>
```

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

11-21 ページの「field-map」を参照してください。

cmp-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 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>
```

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。詳細については、『WebLogic JDBC プログラミング 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

11-37 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

5-16 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

11-21 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 データベースの Blob または Clob、あるいは Sybase データベースの LongString にマップします。この要素は、次のいずれかです。

- OracleBlob
- OracleCLob
- LongString

例

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

delay-database-insert-until

指定できる値:	
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

11-37 ページの「table-name」を参照してください。

enable-tuned-updates

注意： このデプロイメント記述子は、EJB 1.1 に対してのみ適用されます。

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します。
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

11-11 ページの「column-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 の名前になります。
- generator-type 要素が NAMED_SEQUENCE_TABLE の場合、generator-name 要素は使用される SEQUENCE_TABLE の名前になります。

例

11-8 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

11-8 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 element 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

11-8 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

11-11 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

11-39 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

11-39 ページの「weblogic-query」を参照してください。

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` と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

11-39 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 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。
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 要素は関連名を指定します。

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>
                <relation-name>stocks-holders</relation-name>
                        <table-name>stocks</table-name>
        </weblogic-rdbms-relation>
</weblogic-rdbms-jar>
```

relationship-role-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 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>
       <weblogic-relationship-role>stockholder</weblogic-
relationship-role>
            <relationship-role-name>stockholders</relationship-
role-name>
       </weblogic-rdbms-relation>
</weblogic-rdbms-jar>
```

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 TRANSACTION_READ_COMMITED_FOR_UPDATE 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、データベースから一意の行が返されます。

例

この要素を含む XML の例を示します。

```
<sql-select-distinct>True</sql-select-distinct>
```

table-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>
```

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

11-39 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-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-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 つのみのマッピングは、関係がローカルの場合、1 対 1 の関係で指定します。ただし、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。

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-relationship-role>stockholder</weblogic-
relationship-role>

       </weblogic-rdbms-relation>

</weblogic-rdbms-jar>
```

5.1 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。詳細については、4-39 ページの「エンティティ EJB のロック サービス」を参照してください。

WebLogic Server 5.1 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>
```

5.1 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、『JDBC プログラミング 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、5-4 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 メソッドにごとに 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、5-6 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。