Oracle® Containers for J2EE

Enterprise JavaBeans 開発者ガイド

10*g*（10.1.3.1.0）

部品番号 : B31852-03

2008 年 8 月

ORACLE®

Oracle Containers for J2EE Enterprise JavaBeans 開発者ガイド , 10*g*（10.1.3.1.0）

部品番号 : B31852-03

原本名 : Oracle Containers for J2EE Enterprise JavaBeans Developer's Guide, 10*g* (10.1.3.1.0)

原本部品番号 : B28221-03

原本著者 : Peter Purich

原本協力者 : Debu Panda, Raghu Kodali

制限付権利の説明

このプログラム（ソフトウェアおよびドキュメントを含む）には、オラクル社およびその関連会社に所有権のある情報が含まれています。このプログラムの使用または開示は、オラクル社およびその関連会社との契約に記された制約条件に従うものとします。著作権、特許権およびその他の知的財産権と工業所有権に関する法律により保護されています。独立して作成された他のソフトウェアとの互換性を得るために必要な場合、もしくは法律によって規定される場合を除き、このプログラムのリバース・エンジニアリング、逆アセンブル、逆コンパイル等は禁止されています。

このドキュメントの情報は、予告なしに変更される場合があります。オラクル社およびその関連会社は、このドキュメントに誤りが無いことの保証は致し兼ねます。これらのプログラムのライセンス契約で許諾されている場合を除き、プログラムを形式、手段（電子的または機械的）、目的に関係なく、複製または転用することはできません。

このプログラムが米国政府機関、もしくは米国政府機関に代わってこのプログラムをライセンスまたは使用する者に提供される場合は、次の注意が適用されます。

U.S. GOVERNMENT RIGHTS

Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations.As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software--Restricted Rights (June 1987).Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

このプログラムは、核、航空産業、大量輸送、医療あるいはその他の危険が伴うアプリケーションへの用途を目的としておりません。このプログラムをかかる目的で使用する際、上述のアプリケーションを安全に使用するために、適切な安全装置、バックアップ、冗長性（redundancy）、その他の対策を講じることは使用者の責任となります。万一かかるプログラムの使用に起因して損害が発生いたしましても、オラクル社およびその関連会社は一切責任を負いかねます。

Oracle、JD Edwards、PeopleSoft、Siebel は米国 Oracle Corporation およびその子会社、関連会社の登録商標です。その他の名称は、他社の商標の可能性があります。

このプログラムは、第三者の Web サイトへリンクし、第三者のコンテンツ、製品、サービスへアクセスすることがあります。オラクル社およびその関連会社は第三者の Web サイトで提供されるコンテンツについては、一切の責任を負いかねます。当該コンテンツの利用は、お客様の責任になります。第三者の製品またはサービスを購入する場合は、第三者と直接の取引となります。オラクル社およびその関連会社は、第三者の製品およびサービスの品質、契約の履行（製品またはサービスの提供、保証義務を含む）に関しては責任を負いかねます。また、第三者との取引により損失や損害が発生いたしましても、オラクル社およびその関連会社は一切の責任を負いかねます。

目次

2 EJB アプリケーション開発について

3 OC4J での EJB サポートについて

第 II 部　EJB 3.0 セッション Bean

4　EJB 3.0 セッション Bean の実装

5　EJB 3.0 セッション Bean の使用方法

第 III 部　JPA エンティティ

6　JPA エンティティの実装

7　Java 永続性 API の使用方法

8 JPA 問合せの実装

第 IV 部 EJB 3.0 メッセージドリブン Bean

9 EJB 3.0 メッセージドリブン Bean の実装

10 EJB 3.0 メッセージドリブン Bean の使用方法

第 V 部　EJB 2.1 セッション Bean

11　EJB 2.1 セッション Bean の実装

12　EJB 2.1 セッション Bean の使用方法

第 VI 部　EJB 2.1 エンティティ Bean

13　EJB 2.1 エンティティ Bean の実装

14 コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の使用方法

15 Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の使用方法

16 EJB 2.1 問合せの実装

第 VII 部 EJB 2.1 メッセージドリブン Bean

17 EJB 2.1 メッセージドリブン Bean の実装

18 EJB 2.1 メッセージドリブン Bean の使用方法

第 VIII 部　OC4J EJB サービスの構成

19　JNDI サービスの構成

20 データ ソースの構成

21 トランザクション・サービスの構成

22 セキュリティ・サービスの構成

23 メッセージ・サービスの構成

24 OC4J EJB アプリケーション・クラスタリング・サービスの構成

25 タイマー・サービスの構成

第 IX 部 EJB アプリケーションのパッケージ化およびデプロイ

26 デプロイメント・ディスクリプタ・ファイルの構成

27 EJB アプリケーションのパッケージ化

28 OC4J への EJB アプリケーションのデプロイ

第 X 部 アプリケーションでの EJB の使用方法

29 クライアントからの Enterprise Bean へのアクセス

30 EJB および Web サービスの使用方法

31 EJB アプリケーションの管理

32 EJB パフォーマンスの最適化

A orion-ejb-jar.xml 要素の XML 参照

はじめに

このマニュアルでは、次の構成要素を使用して Oracle Containers for J2EE（OC4J）用の Enterprise JavaBeans を作成する手順について説明します。

- Java Enterprise Edition（EE）5 Enterprise JavaBeans（EJB）3.0 および TopLink Java 永続性 API（JPA）永続性プロバイダ
- J2EE 1.4 EJB 2.1 および TopLink EJB 2.1 永続性マネージャ

アプリケーション開発を支援するサンプル・コードが含まれています。

Orion 永続性マネージャは推奨されません。新規開発には OC4J および TopLink JPA 永続性プロバイダを使用することをお薦めします。移行ツール（3-15 ページの「TopLink EJB 2.1 永続性マネージャへの移行」を参照）を使用すると、Orion 永続性マネージャで EJB 2.0 エンティティ Bean を使用する既存の OC4J アプリケーションを簡単に移行して、TopLink 永続性マネージャで EJB 2.0 エンティティ Bean を使用できます。

OC4J について疑問がある場合は、`http://forums.oracle.com/forums/category.jspa?categoryID=13` にある OC4J ユーザーのフォーラムを参照してください。

このドキュメントに関する質問またはフィードバックがある場合は、`http://forums.oracle.com/forums/forum.jspa?forumID=165` にあるドキュメント・フィードバック・フォーラムを参照してください。

対象読者

このマニュアルは、OC4J 用の Enterprise JavaBeans を開発するあらゆる人に役立ちます。このマニュアルは、特にプログラマを対象としていますが、アーキテクチャ設計者、システム・アナリスト、プロジェクト・マネージャおよびその他 OC4J にデプロイされる EJB アプリケーションに関心のある人なら誰にとっても役に立ちます。

このマニュアルは、Java EE、EJB 3.0 仕様および EJB 2.1 仕様に関する実務上の知識があることを前提としています。

ドキュメントのアクセシビリティについて

オラクル社は、障害のあるお客様にもオラクル社の製品、サービスおよびサポート・ドキュメントを簡単にご利用いただけることを目標としています。オラクル社のドキュメントには、ユーザーが障害支援技術を使用して情報を利用できる機能が組み込まれています。HTML 形式のドキュメントで用意されており、障害のあるお客様が簡単にアクセスできるようにマークアップされています。標準規格は改善されつつあります。オラクル社はドキュメントをすべてのお客様がご利用できるように、市場をリードする他の技術ベンダーと積極的に連携して技術的な問題に対応しています。オラクル社のアクセシビリティについての詳細情報は、Oracle Accessibility Program の Web サイト `http://www.oracle.com/accessibility/` を参照してください。

ドキュメント内のサンプル・コードのアクセシビリティについて

スクリーン・リーダーは、ドキュメント内のサンプル・コードを正確に読めない場合があります。コード表記規則では閉じ括弧だけを行に記述する必要があります。しかし JAWS は括弧だけの行を読まない場合があります。

外部 Web サイトのドキュメントのアクセシビリティについて

このドキュメントにはオラクル社およびその関連会社が所有または管理しない Web サイトへのリンクが含まれている場合があります。オラクル社およびその関連会社は、それらの Web サイトのアクセシビリティに関しての評価や言及は行っておりません。

Oracle サポート・サービスへの TTY アクセス

アメリカ国内では、Oracle サポート・サービスへ 24 時間年中無休でテキスト電話（TTY）アクセスが提供されています。TTY サポートについては、(800)446-2398 にお電話ください。アメリカ国外からの場合は、+1-407-458-2478 にお電話ください。

関連ドキュメント

詳細は、OC4J ドキュメント・セットの次のドキュメントを参照してください。

- Oracle Application Server のリリース・ノート
- 『Oracle Containers for J2EE 構成および管理ガイド』
- 『Oracle Containers for J2EE リソース・アダプタ管理者ガイド』
- 『Oracle Containers for J2EE 開発者ガイド』
- 『Oracle Containers for J2EE サービス・ガイド』
- 『Oracle Containers for J2EE セキュリティ・ガイド』
- 『Oracle Containers for J2EE デプロイメント・ガイド』
- 『Oracle Containers for J2EE ジョブ・スケジューラ開発者ガイド』
- 『Oracle Containers for J2EE サーブレット開発者ガイド』
- 『Oracle Application Server Annotations API Reference』
- 『Oracle TopLink 開発者ガイド』
- 『Oracle TopLink API Reference』
- EJB 仕様 : `http://java.sun.com/products/ejb/docs.html`
- EJB API ドキュメント : `http://www.javasoft.com`
- EJB チュートリアル : `http://java.sun.com/developer/onlineTraining/`
- EJB 設計パターン : `http://java.sun.com/blueprints/patterns/`

表記規則

このマニュアルでは次の表記規則を使用します。

規則	意味
太字	太字は、操作に関連する Graphical User Interface 要素、または本文中で定義されている用語および用語集に記載されている用語を示します。
イタリック体	イタリックは、ユーザーが特定の値を指定するプレースホルダ変数を示します。
固定幅フォント	固定幅フォントは、段落内のコマンド、URL、サンプル内のコード、画面に表示されるテキスト、または入力するテキストを示します。

サポートおよびサービス

次の各項に、各サービスに接続するための URL を記載します。

Oracle サポート・サービス

オラクル製品サポートの購入方法、および Oracle サポート・サービスへの連絡方法の詳細は、次の URL を参照してください。

```
http://www.oracle.co.jp/support/
```

製品マニュアル

製品のマニュアルは、次の URL にあります。

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

研修およびトレーニング

研修に関する情報とスケジュールは、次の URL で入手できます。

```
http://www.oracle.co.jp/education/
```

その他の情報

オラクル製品やサービスに関するその他の情報については、次の URL から参照してください。

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

注意： ドキュメント内に記載されている URL や参照ドキュメントには、Oracle Corporation が提供する英語の情報も含まれています。日本語版の情報については、前述の URL を参照してください。

第 I 部

EJB の概要

第 I 部では、EJB アーキテクチャ、EJB アプリケーション開発および OC4J EJB サポートを理解する上で役立つ概念的な情報を示します。

第 I 部は次の各章で構成されています。

1

Enterprise JavaBeans について

Java Enterprise Edition（Java EE）Enterprise JavaBeans（EJB）は、エンタープライズ規模のオブジェクト指向分散アプリケーションの開発およびデプロイに使用するコンポーネント・アーキテクチャです。EJB アーキテクチャに従って作成されたアプリケーションは、拡張性が高くトランザクション対応で安全です。作成されるコンポーネント・タイプは、一般に Enterprise JavaBeans と呼ばれます。

この章の内容は次のとおりです。

Enterprise JavaBeans とは

EJB アーキテクチャには、表 1-1 にリストするオブジェクトを実装するための十分な柔軟性があります。

表 1-1 EJB のタイプ

タイプ	説明	参照先
セッション	クライアントの操作を実行するために使用される単一のクライアント / サーバー・セッションの継続時間中、クライアントによって作成される EJB 3.0 または EJB 2.1 コンポーネント。	1-30 ページの「セッション Bean とは」
ステートレス	対話状態を維持しないセッション Bean。特定のクライアントに接続していない再利用可能なビジネス・サービスに使用されます。	1-31 ページの「ステートレス・セッション Bean とは」
ステートフル	対話状態を維持するセッション Bean。インスタンス変数の値やトランザクション状態などの状態を（存続期間中）維持する単一クライアントとの対話型セッションに使用されます。	1-32 ページの「ステートフル・セッション Bean とは」
エンティティ	永続性ユニットで指定された Java 永続性 API（JPA）永続性プロバイダを使用してリレーショナル・データベースに格納されている永続データを表す EJB 3.0 準拠の軽量エンティティ・オブジェクト（2-10 ページの「persistence.xml ファイルとは」を参照）。	1-38 ページの「JPA エンティティとは」
エンティティ Bean	リレーショナル・データベースに格納されている永続データを表す EJB 2.1 Enterprise Bean コンポーネント。	1-44 ページの「EJB 2.1 エンティティ Bean とは」
CMP	コンテナ管理の永続性（CMP）を備えたエンティティ Bean は、その Bean をホスティングしているコンテナで使用される永続性マネージャに永続性管理を委任するエンティティ Bean です。	1-45 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean とは」
BMP	Bean 管理の永続性（BMP）を備えたエンティティ Bean は、自身の永続性を管理するエンティティ Bean です。	1-49 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean とは」
MDB	メッセージドリブン Bean（MDB）は、Java Message Service（JMS）メッセージの非同期コンシューマとして機能する EJB 3.0 または EJB 2.1 コンポーネントです。	1-59 ページの「メッセージドリブン Bean とは」

詳細は、次を参照してください。

- EJB 3.0 Enterprise Bean の構造
- EJB 2.1 Enterprise Bean の構造
- Enterprise Bean のライフ・サイクル
- EJB コンテキストとは
- アノテーションおよびリソース・インジェクションの動作
- 使用する Enterprise Bean のタイプ

EJB 3.0 Enterprise Bean の構造

EJB 3.0 を使用している場合、EJB 実装のインタフェースは EJB のタイプによって制限されません。たとえば、JPA エンティティの実装では、Plain Old Java Object（POJO）および任意の Plain Old Java Interface（POJI）を使用して EJB を実装できます。`javax.ejb.EntityBean` のようなインタフェースを実装する必要はなく、`EJBHome`、`EJBLocalHome`、`EJBObject` または `EJBLocalObject` を拡張する別のインタフェースも不要です。クライアントは、EJB 3.0 POJO エンティティ・インスタンスを `new` で（または 1-42 ページの「JPA エンティティの問合せ方法」に説明されている `EntityManager` で）インスタンス化します。クライアントは、依存性注入または JNDI ルックアップを使用して EJB 3.0 セッション Bean をインスタンス化します。詳細は、3-2 ページの「EJB 3.0 サポート」を参照してください。

表 1-2 に、EJB 3.0 Enterprise Bean の開発時に作成する構成要素をリストします。

表 1-2 EJB 3.0 EJB の構成要素

構成要素	タイプ	説明
ホーム・インタフェース	POJI	`@Home` アノテーションが付けられたオプションの POJI であり、コンテナ自体が実装するオブジェクト、つまりホーム・オブジェクトを指定します。`@Home` は、EJB 3.0 Bean が必要に応じて EJB 2.1 クライアントと相互運用するのを支援するためにのみ用意されています。ほとんどの EJB 3.0 Bean インスタンスでは、ホーム・インタフェースを提供する必要はありません。
コンポーネント・インタフェース	POJI	`@Remote` または `@Local`（デフォルト）のアノテーションが付けられた必須の POJI であり、Bean に実装してクライアントから起動できるビジネス・メソッドを指定します。デフォルトのコンテナ動作をオーバーライドする必要がないかぎり、他のコンテナ・サービス・メソッドを実装する必要はありません。Bean クラスでは、このインタフェースを実装する必要はありません。
Bean の実装	POJO	コンポーネント・インタフェースをオプションで実装でき、オプションのホーム・インタフェースおよびコンポーネント・インタフェース（ビジネス・メソッド）で定義されたメソッドを実装する Java コードを含む必須の POJO です。必要に応じて、コンテナ・ライフ・サイクル・コールバック関数として機能するよう任意のメソッドにアノテーションを付けることができます。

デプロイメント・ディスクリプタ	`ejb-jar.xml`
	`orion-ejb-jar.xml`
	`toplink-ejb-jar.xml`
	`ejb3-toplink-sessions.xml`
	`persistence.xml`
	`orm.xml`
		デプロイのために Bean の属性を指定するオプションの手段。これらにより、環境、インタフェース名、トランザクションのサポート、EJB のタイプ、および永続性情報など、構成の詳細を決定します。このメタデータはアノテーション（またはデフォルト）によって完全に表現できるため、デプロイメント・ディスクリプタ XML ファイルは EJB 3.0 では重要性が高くありません。デプロイメント・ディスクリプタ XML ファイル内の構成により、対応するアノテーション構成がオーバーライドされます（存在する場合）。詳細は、2-6 ページの「EJB デプロイメント・ディスクリプタ・ファイルについて」を参照してください。

図 1-1 に示すように、EJB 3.0 EJB インスタンスを取得するために、Web クライアント（サーブレットなど）または Java クライアントは JNDI を使用しますが、EJB クライアントは JNDI またはリソース・インジェクションを使用できます。EJB クライアントの詳細は、29-2 ページの「使用しているクライアントのタイプ」を参照してください。

エンティティ Bean の場合、EJB 3.0 は JPA エンティティの作成、検索、マージおよび維持に使用する `EntityManager` を提供します（1-42 ページの「JPA エンティティの問合せ方法」を参照）。

図 1-1 コンポーネント・インタフェースによる EJB 3.0 ステートフル・セッション Bean を使用したクライアント

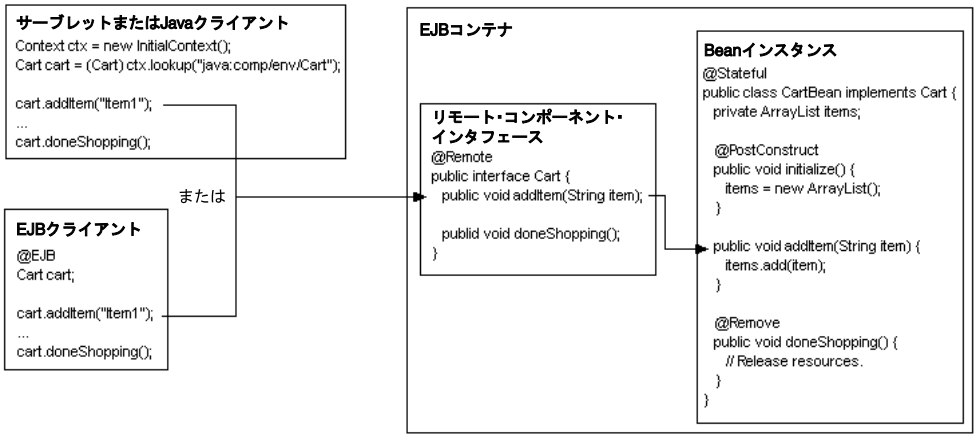


図 1-1 のクライアントは、EJB に次のようにアクセスします。

1. クライアントは、Bean のコンポーネント・インタフェースを取得します。

   サーブレットまたは Java クライアントは、JNDI を使用して `Cart` のインスタンスをルックアップします。

   EJB クライアントは、`Cart` インスタンス変数に `@EJB` アノテーションを付けることによりリソース・インジェクションを使用します。実行時、EJB コンテナにより変数が適宜初期化されます。

   どちらの場合も、EJB コンテナがインスタンス化を管理します。ホーム・インタフェースは不要です。

2. クライアントは、コンポーネント・インタフェース（リモートまたはローカル・インタフェース）で定義されているメソッドを起動し、これにより、メソッド・コールが Bean インスタンス内の対応するメソッドに（スタブを通じて）委任されます。

3. クライアントは、コンポーネント・インタフェース内で、`@Remove` アノテーションが付けられた Bean インスタンス内のメソッドを起動することにより、ステートフル・セッション Bean インスタンスを破棄できます。

   ステートレス・セッション Bean には、`remove` メソッドは不要です。コンテナが必要に応じて Bean を削除します。コンテナは、構成されたタイムアウトを超えるステートフル・セッション Bean を削除するか、最大の構成済プール・サイズを維持できます。エンティティには、`remove` メソッドは不要です。EJB 3.0 `EntityManager` を使用してエンティティを作成および破棄します。

EJB 2.1 Enterprise Bean の構造

EJB 2.1 を使用している場合、EJB 実装のインタフェースは EJB のタイプに基づきます。たとえば、EJB 2.1 エンティティ Bean の実装では、`javax.ejb.EntityBean` インタフェースを実装し、`EJBHome` または `EJBLocalHome`、および `EJBObject` または `EJBLocalObject` を拡張する別のインタフェースを指定する必要があります。クライアントは、EJB ホーム・インタフェースが提供する `create` メソッドでのみ EJB 2.1 Enterprise Bean インスタンスをインスタンス化できます。詳細は、3-13 ページの「EJB 2.1 サポート」を参照してください。

表 1-3 に、EJB 2.1 Enterprise Bean の開発時に作成する構成要素をリストします。

表 1-3 EJB 2.1 EJB の構成要素

構成要素	タイプ	説明
ホーム・インタフェース	`javax.ejb.EJBHome`（リモート） `javax.ejb.EJBLocalHome`	コンテナ自体が実装するオブジェクト、つまりホーム・オブジェクトのインタフェースを指定します。ホーム・インタフェースには、Bean の作成方法を指定する `create` メソッドなどのライフ・サイクル・メソッドが含まれています。
コンポーネント・インタフェース	`javax.ejb.EJBObject`（リモート） `javax.ejb.EJBLocalObject`	Bean で実装するビジネス・メソッドを指定します。また、Bean に、その他のコンテナ・サービス・メソッドも実装する必要があります。EJB コンテナは、Bean のライフ・サイクル中に様々なタイミングで、これらのメソッドを起動します。

Bean の実装	`javax.ejb.SessionBean`
	`javax.ejb.EntityBean`
	`javax.ejb.MessageDrivenBean`
		ホーム・インタフェースで定義されるメソッド（ライフ・サイクル・メソッド）、コンポーネント・インタフェースで定義されるメソッド（ビジネス・メソッド）、および必須のコンテナ・メソッド（コンテナ・コールバック関数）を実装する Java コードが含まれています。

デプロイメント・ディスクリプタ	`ejb-jar.xml`
	`toplink-ejb-jar.xml`
	`orion-ejb-jar.xml`
		デプロイする際の Bean の属性を指定します。これらにより、環境、インタフェース名、トランザクションのサポート、EJB のタイプ、および永続性情報など、構成の詳細を決定します。

図 1-2 に示すように、クライアントは、ホーム・インタフェースを使用して EJB 2.1 Enterprise Bean インスタンスを取得し、コンポーネント・インタフェースを使用してビジネス・メソッドを起動します。EJB クライアントの詳細は、29-2 ページの「使用しているクライアントのタイプ」を参照してください。

図 1-2 ホーム・インタフェースおよびコンポーネント・インタフェースによる EJB 2.1 ステートレス・セッション Bean を使用したクライアント

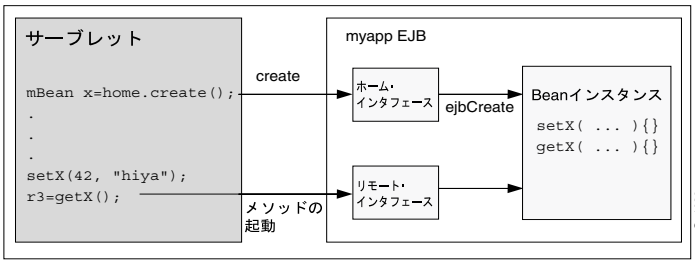


図 1-2 のクライアントは、EJB に次のようにアクセスします。

1. クライアントは、Bean のホーム・インタフェースを通常は JNDI を通じて取得します。
2. クライアントは、ホーム・インタフェースの参照（ホーム・オブジェクト）で `create` メソッドを起動します。これにより、Bean インスタンスが作成され、Bean のコンポーネント・インタフェース（リモートまたはローカル・インタフェース）への参照が返されます。
3. クライアントは、コンポーネント・インタフェース（リモートまたはローカル・インタフェース）で定義されているメソッドを起動し、これにより、メソッド・コールが Bean インスタンス内の対応するメソッドに（スタブを通じて）委任されます。
4. クライアントは、コンポーネント・インタフェース（リモートまたはローカル・インタフェース）で定義されている `remove` メソッドを起動することにより、Bean のインスタンスを破棄できます。

   ステートレス・セッション Bean などの一部の Bean では、`remove` メソッドをコールしても何も行われません。この場合は、コンテナが Bean インスタンスの削除を行います。

Enterprise Bean のライフ・サイクル

Enterprise Bean のライフ・サイクルには、作成、非アクティブ化、アクティブ化、削除などの重要イベントが関係します。

このような各イベントは、コールバック・メソッドに関連付けられています。ライフ・サイクル・コールバック・メソッドは、次のクラスで定義できます。

- 任意の Bean タイプの Enterprise Bean クラス自体（1-6 ページの「Bean クラスのライフ・サイクル・コールバック・メソッド」を参照）
- EJB 3.0 セッション Bean およびメッセージドリブン Bean 用の Enterprise Bean のインターセプタ・クラス（1-6 ページの「EJB 3.0 インターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッド」を参照）
- JPA エンティティのエンティティ・リスナー・クラス（1-6 ページの「JPA エンティティ・リスナー・クラスのライフ・サイクル・コールバック・リスナー・メソッド」を参照）

これらのオプションを組み合せて使用できます。たとえば、一部のライフ・サイクル・コールバックをセッション Bean クラスのメソッドとして定義し、一部をそのセッション Bean に関連付けられたインターセプタ・クラスに定義できます。

コンテナは、（イベント・タイプに応じて）ライフ・サイクル・イベントの前または直後にコールバックを起動します。

Enterprise Bean に関連付けられているライフ・サイクル・イベント、およびコンテナと Bean プロバイダのどちらがコールバックの実装を行うかは、（適切な EJB インタフェースで指定された）開発している Enterprise Bean のタイプによって決まります。

EJB 3.0 Enterprise Bean では、コンテナがライフ・サイクル・コールバックを行う場合は、追加ロジックを実行しないかぎり Bean に実装を提供する必要はありません。

EJB 2.1 Enterprise Bean では、コンテナがライフ・サイクル・コールバックを行う場合、また追加ロジックを実行しない場合でも、少なくともライフ・サイクルの空の実装を用意して、該当する EJB インタフェースの要件を満たす必要があります。

詳細は、次を参照してください。

- 1-31 ページの「ステートレス・セッション Bean のライフ・サイクル」
- 1-33 ページの「ステートフル・セッション Bean のライフ・サイクル」
- 1-40 ページの「JPA エンティティのライフ・サイクル」
- 1-46 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル」
- 1-49 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル」
- 1-60 ページの「メッセージドリブン Bean のライフ・サイクル」

Bean クラスのライフ・サイクル・コールバック・メソッド

EJB 3.0 Enterprise Bean タイプの場合は、オプションで任意の EJB クラス・メソッドにライフ・サイクル・メソッドとしてアノテーションを付けることができます。

EJB 2.1 Enterprise Bean の場合は、少なくともライフ・サイクル・メソッドの空の実装を用意して、該当する EJB インタフェースの要件を満たす必要があります。

EJB 3.0 インターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッド

EJB 3.0 セッション Bean またはメッセージドリブン Bean の場合は、オプションで Bean クラスをインターセプタ・クラスに関連付け、任意のインターセプタ・クラス・メソッドにライフ・サイクル・メソッドとしてアノテーションを付けることができます。

詳細は、次を参照してください。

- 2-12 ページの「EJB 3.0 インターセプタについて」
- 5-6 ページの「EJB 3.0 セッション Bean のインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドの構成」
- 10-13 ページの「EJB 3.0 MDB のインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドの構成」

JPA エンティティ・リスナー・クラスのライフ・サイクル・コールバック・リスナー・メソッド

JPA エンティティの場合は、Bean クラスをエンティティ・リスナー・クラスに関連付け、任意のエンティティ・リスナー・クラス・メソッドにライフ・サイクル・メソッドとしてアノテーションを付けることができます。

詳細は、7-19 ページの「JPA エンティティのエンティティ・リスナー・クラスのライフ・サイクル・コールバック・リスナー・メソッドの構成」を参照してください。

EJB コンテキストとは

EJBContext インタフェースは、EJB 2.1 Enterprise Bean インスタンスのコンテナ提供ランタイム・コンテキストへのアクセス権をインスタンスに提供します。このインタフェースは、エンタープライズ・インタフェース Bean タイプに固有の追加メソッドを提供するために SessionContext、EntityContext および MessageDrivenContext インタフェースにより拡張されます。

javax.ejb.EJBContext インタフェースの定義は次のとおりです。

```
public interface EJBContext {
    public EJBHome          getEJBHome();
    public Properties       getEnvironment();
    public Principal        getCallerPrincipal();
    public boolean          isCallerInRole(String roleName);
    public UserTransaction getUserTransaction();
    public boolean          getRollbackOnly();
    public void             setRollbackOnly();
}
```

Bean は、表 1-4 に示された操作を実行する際、EJB コンテキストを必要とします。

表 1-4 EJB 2.1 EJBContext の動作

メソッド	説明
getEnvironment	Bean のプロパティの値を取得します。
getUserTransaction	トランザクション・コンテキストを取得します。これにより、Bean 管理のトランザクション（BMT）の使用時にプログラムによるトランザクション境界が有効になります。これは、トランザクション対応の Bean でのみ有効です。
setRollbackOnly	現在のトランザクションをコミットできないよう設定します。コンテナ管理トランザクションにのみ適用されます。
getRollbackOnly	現在のトランザクションがロールバック専用に指定されているかどうかを調べます。コンテナ管理トランザクションにのみ適用されます。
getEJBHome	Bean の対応する EJBHome（ホーム・インタフェース）のオブジェクト参照を取得します。
lookup	JNDI を使用して、環境変数名で Bean を取得します。このメソッドを使用している場合は、Bean 参照に接頭辞 "java:comp/env" を付けないでください。

EJBContext を IntialContext と混同しないようにしてください（19-20 ページの「初期コンテキスト・ファクトリの構成」を参照）。

詳細は、次を参照してください。

- 1-37 ページの「セッション・コンテキストとは」
- 1-51 ページの「エンティティ・コンテキストとは」
- 1-61 ページの「メッセージ・ドリブン・コンテキストとは」
- 29-30 ページの「EJB 2.1 EJBContext へのアクセス」

アノテーションおよびリソース・インジェクションの動作

アノテーションにより、アプリケーションの動作とデプロイを制御できます。メタデータ・アノテーションを使用して、コンテナ動作の適切な要件の指定、サービスやリソースの注入のリクエスト、およびオブジェクト・リレーショナル・マッピングの指定を行うことができます。

アノテーションを使用すると、EJB 3.0 Enterprise Bean は依存性注入メカニズムを使用して、環境内のリソースまたはその他のオブジェクトへの参照を取得できます。たとえば、次のアノテーションを使用できます。

- `@Resource`: データベース接続などの EJB 以外のリソースを注入します。
- `@EJB`: セッション Bean などの Enterprise Bean を注入します。
- `@PersistenceContext`: EJB 3.0 エンティティを作成、参照、更新および削除する `EntityManager` インスタンスを注入します。

EJB 3.0 Enterprise Bean が依存性注入を利用している場合、OC4J は Bean インスタンスの作成後、ビジネス・メソッドが起動される前にこれらの参照を注入します。

EJB コンテキストに対する依存性が宣言されている場合は、EJB コンテキストも注入されます（1-7 ページの「EJB コンテキストとは」を参照）。

依存性注入に失敗した場合、OC4J は Bean インスタンスを破棄します。

OC4J では、アノテーションの継承がサポートされます（1-9 ページの「アノテーションおよび継承」を参照）。

このリリースでは、Web 層でアノテーションおよびリソース・インジェクションを使用できます（1-9 ページの「Web 層でのアノテーション」を参照）。

アノテーションは、XML を使用せずに環境参照を指定する別の方法です。フィールドまたはプロパティにアノテーションを付ける場合、コンテナは JNDI からルックアップすることでユーザーにかわって値を Bean に注入します。参照がアノテーションを使用して指定されている場合でも、JNDI を使用してルックアップできます。例 1-1 に、アノテーションと JNDI の関連を示します。この例のアノテーションは、例 1-2 にある同等の `ejb-jar.xml` ファイルに対応しています。かわりにこの XML と JNDI が使用された場合も、コードの動作はまったく同じです。

アノテーション構成は、デプロイ XML を使用してオーバーライドできます（1-22 ページの「デプロイメント・ディスクリプタ・エントリによるアノテーションのオーバーライド」を参照）。

例 1-1 アノテーションおよびリソース・インジェクションの使用方法

```
@Stateless
@EJB(name="bean1", businessInterface=Bean1.class)
public class MyBean {
    @EJB Bean2 bean2;

    public void doSomething() {
        // Bean2 is already injected and available
        bean2.foo();
        // or it can be looked up from JNDI
        ((Bean2)(new InitialContext().lookup("java:comp/env/bean2"))).foo();
        // Bean1 has not been injected and is only available through JNDI
        ((Bean1)(new InitialContext().lookup("java:comp/env/bean1"))).foo();
    }
}
```

例 1-2 同等の ejb-jar.xml ファイル構成

```
<ejb-local-ref>
    <ejb-ref-name>bean1</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local>Bean1.class</local>
</ejb-local-ref>
```

```
<ejb-local-ref>
    <ejb-ref-name>bean2</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local>Bean2.class</local>
    <injection-target>
        <injection-target-name>bean2</injection-target-name>
    </injection-target>
</ejb-local-ref>
```

Web 層でのアノテーション

このリリースでは、OC4J により Web 層でのアノテーションおよびリソース・インジェクションがサポートされます。Web 層でアノテーションおよびリソース・インジェクションを使用するには、クライアントで Java SE 1.5 および Servlet 2.5 以上を使用している必要があります。

Web 層では、次のアノテーションを使用できます。

- `@EJB`
- `@Resource` および `@Resources`
- `@PersistenceUnit` および `@PersistenceUnits`
- `@PersistenceContext` および `@PersistenceContexts`
- `@WebServiceRef`
- `@PostConstruct`
- `@PreDestroy`
- `@DeclaresRoles`
- `@RunAs`

詳細は、次を参照してください。

- 『Oracle Containers for J2EE サーブレット開発者ガイド』
- 29-11 ページの「Web クライアントでのエンティティ・マネージャの取得」
- 29-19 ページの「EJB 3.0 を使用した JMS 宛先へのメッセージの送信」

アノテーションおよび継承

アノテーションは、継承に含まれます。アノテーションをホスト・クラスに対してローカルに機能させるには、次の点を考慮してください。

- クラス・レベルのアノテーションは、アノテーション先のクラスとそのメンバー（メソッドおよびフィールド）にのみ影響します。アノテーションは、スーパークラスによって宣言されたメンバーには影響しません。これは、そのメンバーがサブジェクト・サブクラスにより隠蔽されない場合や、オーバーライドされない場合でも同様です。
- 明示的なメンバー・レベルのアノテーションは、そのアノテーションが潜在的に付加的である場合（インターセプタ・アノテーションなど）を除き、クラス・レベルのアノテーションにより暗黙的に示されるメンバー・レベルのアノテーションに優先します。メンバーが特定のメンバー・レベルのアノテーションを継承する場合、クラス・レベルのアノテーションにより示される同じタイプのアノテーションは無視されます。
- クラスにより実装されるインタフェースは、アノテーションをクラス自体またはその任意のメンバーに渡しません。
- スーパークラスから継承されたメンバー（隠蔽またはオーバーライドされないメンバー）は、そのメンバーの宣言元のクラスに含まれていたアノテーションを維持します。このアノテーションには、クラス・レベルのアノテーションにより暗黙的に示されるメンバー・レベルのアノテーションも含まれます。

- 隠蔽されるメンバーまたはオーバーライドされるメンバーのメンバー・レベルのアノテーションは、常に無視されます。

クラス・メンバーに影響しているアノテーションを検出するには、クラス・メンバーの最後の公開宣言または非オーバーライド宣言を追跡調査する必要があります。アノテーションを検出できない場合、エンクロージング・クラス宣言を調査する必要があります。これに失敗しても、他のソース・ファイルは調査しないでください。

表 1-5 に、アノテーションをリストし、各アノテーションが Bean クラスの継承に関してどのように動作するかを示します。

表 1-5 アノテーションおよび継承

アノテーション	継承の影響	コメント	OC4J サポート

`@Stateless`
`@Stateful`
`@MessageDriven`
	スーパークラスのアノテーションは無視されます。
	例 :
	`@Stateful`
	`class Base {}`
	
	`@Stateless`
	`class A extends Base {}`
	
	`class B extends Base {}`
	
	この場合 :
	- Bean `Base` はステートフル・セッション Bean です。
	- Bean `A` はステートレス・セッション Bean です。親 Bean `Base` の `@Stateful` アノテーションは適用されません（無視されます）。
	- Bean `B` は POJO クラスです。親 Bean `Base` の `@Stateful` アノテーションは適用されません（無視されます）。
		クラス・アノテーションまたはデプロイメント・ディスクリプタ XML ファイルを通じて Bean クラスを明示的に定義する必要があります。これは、その Bean が別の Bean クラスのサブクラスである場合でも同様です。	サポートされます。
			OC4J では、スーパークラス・レベルの Bean タイプのアノテーションは無視されます。

`@Local`
`@Remote`
`@LocalHome`
`@Home`
	スーパークラスのアノテーションは無視されます。
	実行時の問題を回避するため、アノテーションを適切に定義する必要があります。
	例 :
	`@Local`
	`interface Base {}`
	
	`@Remote`
	`interface A extends Base {}`
	
	`interface B extends Base {}`
	
	この場合 :
	- `Base` はローカル・ビジネス・インタフェースです。
	- `A` はリモート・インタフェースです。親 Bean `Base` の `@Local` アノテーションは適用されません（無視されます）。
	- `B` は POJO インタフェースです。親 Bean `Base` の `@Local` アノテーションは適用されません（無視されます）。
		Bean に対するアノテーションの動作も同様です。
		例 :
		`@Stateful`
		`@Local(I1.class)`
		`class A {}`
		
		`@Stateful`
		`class B extends A {}`
		
		注意 : `A` とは異なり、Bean `B` は `I1` ビジネス・インタフェースを持ちません。
			サポートされます。
			OC4J では、スーパークラス・レベルの Bean タイプのアノテーションは無視されます。

表 1-5 アノテーションおよび継承（続き）

アノテーション	継承の影響	コメント	OC4J サポート

`@TransactionManagement(TransactionManagementType.CONTAINER)`
`@TransactionManagement(TransactionManagementType.APPLICATION)`
	スーパークラスのアノテーションは無視されます。
	例：
	`@ TransactionManagement (type=TransactionManagementType.CONTAINER) class Base {}`
	`@ TransactionManagement (type=TransactionManagementType.APPLICATION) class A extends Base {}`
	`class B extends Base {}`
	この場合：
	- A は、Bean 管理のトランザクションを使用する Bean です。
	- B は、**デフォルト**のコンテナ管理のトランザクションを使用する Bean です。
		クラス・レベルのトランザクション管理を継承しない場合、Bean 管理のトランザクションおよびコンテナ管理のトランザクションを使用する Bean がアプリケーション内で混在します。これにより、実行時に問題が発生する可能性があります。
		明示的なアノテーションが存在しない場合、Bean ではデフォルトでコンテナ管理のトランザクションが使用されます。
			サポートされます。
			OC4J では、スーパークラス・レベルの Bean タイプのアノテーションは無視されます。

表 1-5 アノテーションおよび継承（続き）

アノテーション	継承の影響	コメント	OC4J サポート

`@TransactionAttribute(TransactionAttributeType.REQUIRED) {MANDATORY, REQUIRED, REQUIRES_NEW, SUPPORTS, NOT_SUPPORTED, NEVER}`	メソッド・レベルの継承と、仮想メソッド・アノテーションの継承が許可されます。
	例：
	`@Transaction(REQUIRED)`
	`class Base {`
	`  @Transaction(NONE)`
	`  public void foo() {...}`
	`  public void bar() {...}`
	`}`
	
	`class A extends Base {`
	`  public void foo() {...}`
	`}`
	
	`public class B extends Base {`
	`  @Transaction(NEW)`
	`  public void foo() {...}`
	`}`
	
	`@Transaction(NEW)`
	`public class C extends Base {`
	`  public void foo() {...}`
	`  public void bar() {...}`
	`}`
	
	`@Transaction(NEW)`
	`public class D extends Base {`
	`  public void bar() {...}`
	`}`
	
	`@Transaction(NEW)`
	`public class E extends Base {`
	
	この場合：
	- Bean A において、foo メソッドにはアノテーションがありません。Bean A は、親 Bean Base の foo メソッドをアノテーションなしでオーバーライドします。したがって、Bean A の foo メソッドは、@Transaction(NONE) アノテーションを継承しません。
	- Bean B において、@Transaction(NEW) アノテーションは、foo メソッドに適用されます。Bean B は、親 Bean Base の foo メソッドを @Transaction(NEW) アノテーション付きでオーバーライドします。その結果、Bean Base の foo メソッドの @Transaction(NONE) アノテーションは、子 Bean B でオーバーライドされるメソッドに適用されません。
	- Bean C において、@Transaction(NEW) アノテーションは、foo メソッドに適用されます。Bean C は、親 Bean Base の foo メソッドをアノテーションなしでオーバーライドします。したがって、Bean C の foo メソッドは、@Transaction(NONE) アノテーションを継承しません。ただし、Bean C は、foo メソッドに適用されるクラス・レベルのアノテーションの @Transaction(NEW) を持ちます。
	- Bean D において、@Transaction(NEW) アノテーションは、bar メソッドに適用されます。Bean D は、親 Bean Base の bar メソッドをアノテーションなしでオーバーライドします。したがって、Bean D の bar メソッドは、@Transaction(NONE) アノテーションを継承しません。ただし、Bean D は、bar メソッドに適用されるクラス・レベルのアノテーションの @Transaction(NEW) を持ちます。
		スーパーコール・クラス・レベルでアノテーションが付けられ、クラス内のすべてのメソッドに適用される仮想メソッド・アノテーションがサポートされます。
		詳細は、http://jcp.org/en/jsr/detail?id=250 の JSR 250 を参照してください。
			サポートされます。

表 1-5 アノテーションおよび継承（続き）

アノテーション	継承の影響	コメント	OC4J サポート
@TransactionAttribute(TransactionAttributeType.REQUIRED) {MANDATORY, REQUIRED, REQUIRES_NEW, SUPPORTS, NOT_SUPPORTED, NEVER} （前行の続き）	（前行の続き） - Bean E において、@Transaction(REQUIRED) アノテーションは、bar メソッドに適用されます。Bean E は、クラス・レベルのアノテーションの @Transaction(NEW) を持ちますが、親 Bean Base の bar メソッドはオーバーライドしません。したがって、Bean E の bar メソッドは、親 Bean Base からクラス・レベルの @Transaction(REQUIRED) アノテーションを継承します。	（前行の続き）	（前行の続き）

@EJB
@EJBs
	このアノテーションのすべての使用を検出するため、すべてのスーパークラスが調査されます。これには、プライベート・メソッドおよびフィールドと、オーバーライドされたプライベート・メソッド（親と子の両方のプライベート）が含まれます。
	例：
	`@EJB(beanName = "Bean1"…)`
	`public class Base {`
	`  @EJB(beanName =" Bean2"..)`
	`  private Bean2 b2;`
	`  @EJB(beanName =" Bean3"..)`
	`  protected void setB3(Bean3 b3){}`
	`}`
	`@EJB(beanName = "Bean4"…)`
	`public class A extends Base {`
	`  @EJB(beanName =" Bean5"..)`
	`  private Bean5 b5;`
	`}`
	`public class B extends Base {}`
	Bean A の解析時には、Bean1、Bean2、Bean3、Bean4 および Bean5 を含むスーパークラスで定義されているすべての @EJB 参照が解析および追加されます。アノテーション付きのフィールドおよびメソッドも注入されます。
	Bean B の解析時には、Bean1、Bean2、Bean3 および Bean4 を含むスーパークラスで定義されているすべての @EJB 参照が解析および追加されます。アノテーション付きのフィールドおよびメソッドも注入されます。
			サポートされます。

@PersistenceUnit @PersistenceUnits @PersistenceContext @PersistenceContexts	@EJB および @EJBs と同様です（前述の行を参照）。		サポートされます。
@Resources @Resource	@EJB および @EJBs と同様です（前述の行を参照）。		サポートされます。

表 1-5 アノテーションおよび継承（続き）

アノテーション	継承の影響	コメント	OC4J サポート

@Interceptors
@ExcludeDefaultInterceptor
@ExcludeClassInterceptor
	継承が許可されます。
	デフォルト・インターセプタと Bean クラス（およびそのスーパークラス）に定義されたインターセプタ以外に、メソッド・レベルのビジネス・メソッド・インターセプタが起動されます。
	例：
	デフォルト・インターセプタ：D1.class
	`@Interceptors({C1.class})`
	`class Base {`
	`  @Interceptors({M1.class})`
	`  public void foo() {...}`
	`  public void bar() {...}`
	`}`
	`@Interceptors({C2.class})`
	`class A extends Base {`
	`  public void foo() {...}`
	`}`
	`@Interceptors({C2.class})`
	`class B extends Base {`
	`  @Interceptors({M2.class})`
	`  public void foo() {...}`
	`}`
	`@Interceptors({C2.class})`
	`class C extends Base {`
	`  public void bar() {...}`
	`}`
	`@Interceptors({C3.class, C4.class})`
	`class E extends Base {}`
	`@Interceptors({C3.class, C4.class})`
	`class F extends Base {`
	`  @ExcludedDefaultInterceptor`
	`  @ExcludedClassInterceptor`
	`  @Interceptors({M2.class})`
	`  public void bar() {...}`
	`}`
	この場合：
	- Bean Base の foo メソッドのインターセプタは、D1、C1、M1 です。D1 はデフォルト・インターセプタであり、C1 は Bean クラス・レベルのインターセプタとして、M1 はメソッド・レベルの foo のインターセプタとして定義されています。
	- Bean A の foo メソッドのインターセプタは、D1、C2 です。D1 はデフォルト・インターセプタであり、C1 は Bean クラス・レベルのインターセプタとして定義されています。Bean A は、foo メソッドをオーバーライドし、そのメソッドのメソッド・レベルのインターセプタを定義しません。
			サポートされます。

表 1-5 アノテーションおよび継承（続き）

アノテーション	継承の影響	コメント	OC4J サポート
@Interceptors @ExcludeDefaultInterceptor @ExcludeClassInterceptor （前行の続き）	（前行の続き） - Bean B の foo メソッドのインターセプタは、D1、C2、M2 です。D1 はデフォルト・インターセプタであり、C2 は Bean クラス・レベルのインターセプタとして定義されています。Bean B は、foo メソッドをオーバーライドし、メソッド・レベルのインターセプタとして M2 を定義します。 - Bean C の bar メソッドのインターセプタは、D1、C2 です。D1 はデフォルト・インターセプタであり、C2 は Bean クラス・レベルのインターセプタとして定義されています。 - Bean E の bar メソッドのインターセプタは、D1、C1 です。D1 はデフォルト・インターセプタです。Bean E は、クラス・レベルのアノテーションの @Interceptors({C3.class, C4.class}) を持ちますが、親 Bean Base の bar メソッドはオーバーライドしません。したがって、Bean E の bar メソッドは、親 Bean Base からクラス・レベルの @Interceptors({C1.class}) アノテーションを継承します。 - Bean F の bar メソッドのインターセプタは、M2 です。bar に @ExcludeDefaultInterceptor というアノテーションが付けられているため、デフォルト・インターセプタの D1 はこのメソッドに適用されません。bar には @ExcludeClassInterceptor というアノテーションも付けられているため、クラス・レベルで定義されたインターセプタも適用されません。Bean F は、bar メソッドをオーバーライドし、それに @Interceptors({M2.class}) アノテーションを付けます（このアノテーションのみが適用されます）。	（前行の続き）	（前行の続き）

@AroundInvoke	Bean クラスにスーパークラスがある場合、@AroundInvoke アノテーション付きでそれらのスーパークラスに定義されたメソッドが起動されます（最も一般的なスーパークラスのメソッドが最初に起動されます）。
	例：
	`class Base {`
	`  @AroundInvoke`
	`  public Object foo(InvocationContext cts) {...}`
	`}`
	`class A extends Base {`
	`  @AroundInvoke`
	`  public Object bar(InvocationContext cts) {...}`
	`}`
	`class B extends Base {`
	`  public Object foo(InvocationContext cts) {...}`
	`}`
	この場合：
	- Bean Base において、インターセプタ・メソッドは foo() です。
	- Bean A には、foo および bar の 2 つのインターセプタ・メソッドがあります。bar メソッドは Bean A で定義されており、foo メソッドは親 Bean Base から Bean A に継承されています。最初に foo が起動され、次に bar が起動されます。
	- Bean B には、インターセプタ・メソッドはありません。Bean B は、@AroundInvoke アノテーションなしで foo メソッドをオーバーライドするため、このメソッドはインターセプタ・メソッドではなくなります。
		インターセプタ・クラスにスーパークラスがある場合、インターセプタ・クラスのスーパークラスで定義されたインターセプタ・メソッドは、インターセプタ・クラスで定義されたインターセプタ・メソッドの前に起動されます（最も一般的なスーパークラスのメソッドが最初に起動されます）。	サポートされます。

表 1-5 アノテーションおよび継承（続き）

アノテーション	継承の影響	コメント	OC4J サポート

@PostConstruct
@PreDestroy
@PostActivate
@PrePessivate
	Bean クラスにスーパークラスがある場合、それらのスーパークラスに定義されたライフ・サイクル・コールバック（インターセプタ）メソッドが起動されます（最も一般的なスーパークラスのメソッドが最初に起動されます）。
	注意：オーバーライドされたライフ・サイクル・メソッドは起動されません。
	例：
	`class Base {`
	`  @PostConstruct`
	`  @PostActivate`
	`  void foo() {...}`
	`}`
	
	`class A extends Base {`
	`  @PostConstruct`
	`  void bar() {...}`
	`  @PostActivate`
	`  void ping() {...}`
	`}`
	
	`class B extends Base {`
	`  @PreDestroy`
	`  void foo() {...}`
	`}`
	
	`class C extends Base {`
	`  ejbCreate() {...}`
	`}`
	
	`class D extends Base {`
	`  @PostConstruct`
	`  ping() {...}`
	`  ejbCreate() {...}`
	`}`
	
	この場合：
	- Bean `Base` には、`post-construct` メソッドの `foo` および `post-activate` ライフ・サイクル・メソッドの `foo` の 2 つのライフ・サイクル・メソッドがあります。
	- Bean `A` には、`foo` および `bar` の 2 つの `post-construct` メソッドがあります。最初に `foo` メソッドが起動され、次に `bar` が起動されます。また、Bean `A` には、`foo` および `ping` の 2 つの `post-activate` ライフ・サイクル・メソッドがあります。最初に `foo` メソッドが起動され、次に `ping` が起動されます。
	- Bean `B` において、`foo` メソッドは `@PreDestroy` アノテーション付きでオーバーライドされています。したがって、`post-construct` メソッドは Bean `B` で定義されず、`post-activate` ライフ・サイクル・メソッドも定義されません。Bean `B` で定義されるのは、`pre-destroy` ライフ・サイクル・メソッドの `foo` のみです。
	- Bean `C` には、親 Bean `Base` から継承されて最初に起動される `foo` と、Bean `C` で定義されて 2 番目に起動される `ejbCreate` の 2 つの `post-construct` メソッドがあります。
	- Bean `D` ではエラーが発生します。EJB 2.1 スタイルのライフ・サイクル・コールバック（`ejbCreate()` メソッドなど）は、対応する EJB 3.0 スタイル（`@PostConstruct` アノテーションなど）のコールバックと 1 つの Bean クラス内で共存できません。
		インターセプタ・クラスにスーパークラスがある場合、インターセプタ・クラスのスーパークラスで定義されたライフ・サイクル・コールバック・インターセプタ・メソッドは、インターセプタ・クラスで定義されたライフ・サイクル・コールバック・インターセプタ・メソッドの前に起動されます（最も一般的なスーパークラスのメソッドが最初に起動されます）。	サポートされます。

表 1-5 アノテーションおよび継承（続き）

アノテーション	継承の影響	コメント	OC4J サポート

@Timeout	継承階層で最大 1 つのタイムアウト・メソッドが許可されます。
	例 :
	`class Base {`
	`  @Timeout`
	`  public void foo(Timer) {...}`
	`}`
	
	`class A extends Base {`
	`  @Timeout`
	`  public void bar(Timer) {...}`
	`}`
	
	`class B extends Base {`
	`  public void foo(Timer) {...}`
	`}`
	
	`class C extends Base implements TimedObject {`
	`  public void ejbTimeout(Timer) {...}`
	`}`
	
	この場合 :
	- foo は、Bean Base のタイムアウト・メソッドです。
	- Bean A ではエラーが発生します。bar は、Bean A で定義されているタイムアウト・メソッドです。Bean A は、親 Bean Base からも foo タイムアウト・メソッドを継承しています。この場合、Bean A に 2 つのタイムアウト・メソッドが存在することになりますが、これは許可されません。
	- Bean B には、タイムアウト・メソッドが存在しません。Bean B は、アノテーションなしで foo メソッドをオーバーライドするため、タイムアウト・メソッドはなくなります。
	- Bean C ではエラーが発生します。ejbTimeout は、Bean C で定義されているタイムアウト・メソッドです。また、Bean C は、親 Bean Base から foo タイムアウト・メソッドを継承しています。この場合、Bean C に 2 つのタイムアウト・メソッドが存在することになりますが、Bean C に 2 つのタイムアウト・メソッドが存在することは許可されません。
		ベースおよびスーパークラスの両方でメソッドにアノテーションが付けられている場合（異なるメソッド名）、EJB 3.0 仕様では Bean ごとに 1 つのタイムアウト・メソッドのみ許可されるため、コンテナから例外がスローされます。	サポートされます。

表 1-5 アノテーションおよび継承（続き）

アノテーション	継承の影響	コメント	OC4J サポート

@Remove	複数の削除が許可されます。
	例 :
	class Base {
	@Remove
	void foo() {...}
	}
	
	class A extends Base {
	@Remove
	void bar() {...}
	}
	
	class B extends Base {
	void foo() {...}
	}
	
	class C extends Base {
	@Remove
	void foo(int) {...}
	}
	
	この場合 :
	- foo は、Bean Base の削除メソッドです。
	- foo および bar は、Bean A の削除メソッドです。bar メソッドは、Bean A で削除メソッドとして明示的に定義されています。Bean A は、foo メソッドをオーバーライドしないため、親 Bean Base から削除メソッドとして foo を継承します。
	- Bean B には、削除メソッドはありません。Bean B は、アノテーションなしで foo メソッドをオーバーライドします（メソッドがオーバーライドされなければ、Bean Base の foo メソッドの @Remove アノテーションが継承されます）。
	- foo() および foo(int) は、Bean C の削除メソッドです。foo(int) メソッドは、Bean C で削除メソッドとして明示的に定義されています。Bean C は、foo() メソッドをオーバーライドしないため、親 Bean Base から削除メソッドとして foo() を継承します。
		@Remove アノテーションは、本質的に付加的です。1 つの Bean 内で複数の削除メソッドが許可されます。	サポートされます。

表 1-5 アノテーションおよび継承（続き）

アノテーション	継承の影響	コメント	OC4J サポート

@RolesAllowed
@DenyAll
@PermitAll
	メソッド・レベルの継承のみ許可されます。
	例：
	@PermitAll
	class Base {
	@DenyAll
	public void foo() {...}
	void bar() {...}
	}
	
	class A extends Base {
	public void foo() {...}
	}
	
	public class B extends Base {
	@RolesAllowed({admin})
	public void foo( ) {...}
	}
	
	@RolesAllowed({guest}, {admin})
	public class C extends Base {
	public void foo() {...}
	void bar(){...}
	}
	
	@DenyAll
	public class D extends Base {
	public void bar() {...}
	}
	
	@RolesAllowed({guest}, {admin})
	public class E extends Base {}
	
	@RolesAllowed({guest}, {admin})
	class F extends Base {
	@RolesAllowed ({admin})
	public void bar() {...}
	}
	
	この場合：
	- Bean A において、セキュリティ権限は foo メソッドに付与されません（どのロールも許可されません）。Bean A は、親クラス Base からクラス・レベルのアノテーションを継承できません。さらに、Bean A は、その親の foo メソッドをオーバーライドし、そのオーバーライド後のメソッドにアノテーションを付けないため、Bean A の foo メソッドは、アノテーションなしのメソッドとして動作します。
	- Bean B において、foo メソッドでは admin ロールのみが許可されます。Bean B は、独自のクラス・レベルのアノテーションを持たず、親クラス Base からもクラス・レベルのアノテーションを継承しません。ただし、Bean B は、その親の foo メソッドをオーバーライドし、そのオーバーライド後のメソッドに @RolesAllowed({admin}) アノテーションを付けるため、Bean B の foo メソッドでは、admin ロールが設定されます。
		これは、トランザクション属性の使用例と同様です。
		注意： EJB 3.0 仕様には、メソッド・レベルのセキュリティ・アノテーションがクラス・レベルのアノテーションをオーバーライドすると記載されています。ただし、ejb-jar.xml の method-permission 要素では、この設定は付加的（またはクラス・レベルとメソッド・レベル両方のロールの組合せ）となります。この例は、Bean F の場合です。
			サポートされます。

表 1-5 アノテーションおよび継承（続き）

アノテーション	継承の影響	コメント	OC4J サポート
@RolesAllowed @DenyAll @PermitAll （前行の続き）	（前行の続き） - Bean C において、foo メソッドでは guest および admin ロールが許可されます。Bean C は、@RolesAllowed({guest},{admin}) という独自のクラス・レベルのアノテーションを持ちますが、親クラス Base からクラス・レベルのアノテーションを継承しません。Bean C は、その親の foo メソッドをオーバーライドし、そのオーバーライド後のメソッドにアノテーションを付けないため、Bean C の foo メソッドでは、guest および admin ロールが設定されます。 - Bean D において、bar メソッドではどのロールも許可されません（すべて拒否）。Bean D は、@DenyAll という独自のクラス・レベルのアノテーションを持ちますが、親クラス Base からクラス・レベルのアノテーションを継承しません。Bean D は、その親の bar メソッドをオーバーライドし、そのオーバーライド後のメソッドにアノテーションを付けないため、Bean D の bar メソッドでは、すべてのセキュリティ権限が拒否されます。 - Bean E において、bar メソッドではすべてのロールが許可されます（すべて許可）。Bean E は、@RolesAllowed({guest},{admin}) という独自のクラス・レベルのアノテーションを持ちますが、親クラス Base からクラス・レベルのアノテーションを継承しません。ただし、Bean E は、その親の bar メソッドをオーバーライドしないため、Bean Base から @PermitAll アノテーション付きでこのメソッドを継承します。 - Bean F の説明は、「コメント」列を参照してください。	（前行の続き）	（前行の続き）

@RunAs	クラス・レベルの継承は許可されません。
	例 :
	`@RunAs ("bob")`
	`class Base {}`
	
	`@RunAs ("joe")`
	`class A extends Base {}`
	
	`class B extends Base {}`
	
	この場合 :
	- 実行者は、Bean A では joe として定義されます。Bean A は、親クラスからアノテーションを継承できないため、親クラス Base から実行者 bob を継承しません。
	- 実行者は、Bean B では定義されません。Bean B は、親クラスからアノテーションを継承できないため、親クラス Base からロール bob を継承しません。
			サポートされます。
			スーパークラス・レベルの Bean タイプのアノテーションは無視されます。

表 1-5 アノテーションおよび継承（続き）

アノテーション	継承の影響	コメント	OC4J サポート

@DeclareRoles	クラス・レベルの継承は許可されません。
	例：
	`@DeclareRoles ({"bob"})`
	`class Base {}`
	
	`@DeclareRoles ({"joe"})`
	`class A extends Base {}`
	
	`class B extends Base {}`
	
	この場合：
	- Bean A は、ロール joe を宣言しています。Bean A は、親クラスからアノテーションを継承できないため、親クラス Base からロール bob を継承しません。
	- Bean B は、ロールを宣言していません。Bean B は、親クラスからアノテーションを継承できないため、親クラス Base からロール bob を継承しません。
			サポートされます。

@WebService	クラス・レベルの継承は許可されません。
	例：
	`@WebServices`
	`class Base {}`
	
	`@Stateless`
	`class A extends Base {}`
	
	この場合：
	- Bean A は、親クラス Base から @WebService アノテーションを継承しないため、Web サービス・エンドポイントではありません。
			サポートされます。
			スーパークラス・レベルの Bean タイプのアノテーションは無視されます。

@StatefulDeployemt
@StatelessDeployemt
@MessageDrivenDeployemt
	クラス・レベルの継承は許可されません。
	例：
	`@StatefulDeployment (timeout=60)`
	`class Base {}`
	
	`@StatefulDeployment (timeout=30)`
	`class A extends Base {}`
	
	`class B extends Base {}`
	
	この場合：
	- Bean A には、30 のステートフル・デプロイ・タイムアウトが設定されます。Bean A は、親クラスからアノテーションを継承できないため、親クラス Base から 60 のステートフル・デプロイ・タイムアウトを継承しません。
	- Bean B には、タイムアウトが設定されません。Bean B は、親クラスからアノテーションを継承できないため、親クラス Base から 60 のステートフル・デプロイ・タイムアウトを継承しません。
		これらは OC4J 固有のアノテーションであり、EJB 3.0 仕様には定義されていません。	サポートされます。
			スーパークラス・レベルの Bean タイプのアノテーションは無視されます。

デプロイメント・ディスクリプタ・エントリによるアノテーションのオーバーライド

アプリケーション設計では、アノテーションとデプロイメント・ディスクリプタの使用を組み合せることができます。この場合、デプロイメント・ディスクリプタは、アノテーションのオーバーライド・メカニズムとして機能します。XML ディスクリプタを使用してアノテーションをオーバーライドするときに適用されるルールのリストは、EJB 3.0 仕様を参照してください。

OC4J では、EJB 3.0 仕様に定義されたアノテーション・オーバーライド・ルールがサポートされます。現在のリリースの OC4J では、デプロイメント・ディスクリプタによるオーバーライドがこれらのルールに違反すると、OC4J により警告が記録されてそのオーバーライドは無視され、アノテーション構成が使用されます。たとえば、あるクラスに `@Stateful` アノテーションを付け、次に `ejb-jar.xml` ファイルの `<entity>` エントリでこの設定をオーバーライドすると、Bean タイプはオーバーライドできないというオーバーライド・ルールに違反します。この場合、OC4J は、警告を記録してオーバーライドを無視し、そのクラスを引き続きステートフル・セッション Bean として処理します。

注意： 将来のリリースの OC4J では、デプロイを中断する例外で警告が置き換えられる予定です。

表 1-6 に、EJB 3.0 仕様に定義されている XML によるアノテーションのオーバーライド・ルールと、それらのルールに関する OC4J（リリース 10.1.3.1、EJB レイヤー）の動作をリストします。

表 1-6 XML によるアノテーションのオーバーライド

有効範囲	アノテーション	XML	EJB 3.0 仕様のオーバーライド・ルール	OC4J 固有の動作（リリース 10.1.3.1）
セッション Bean タイプ	`@Stateless` `@Stateful`	`<session-type>`	EJB 3.0 仕様の 19.2 項には、Bean タイプを `@Stateless`、`@Stateful` または `@MessageDriven` アノテーションで指定している場合、そのタイプはデプロイメント・ディスクリプタでオーバーライドできないと記載されています。Bean タイプ（およびそのセッション・タイプ）を指定する場合、アノテーションでの指定と同じにする必要があります。	このルールに違反すると、OC4J により警告が記録されます。 **注意 :** OC4J の 11*g* リリースでは、コンテナにより検証例外がスローされる予定です。
トランザクション・タイプ : BMT または CMT	`@TransactionManagement(TransactionManagementType.CONTAINER, TransactionManagementType.APPLICATION)`	`<transaction-type>`	EJB 3.0 仕様の 13.3.6 項には、トランザクション・タイプのオーバーライドは許可されないと記載されています。	このルールに違反すると、OC4J により警告が記録されます。 **注意 :** OC4J の 11*g* リリースでは、コンテナにより検証例外がスローされる予定です。
トランザクション属性	`@TransactionAttribute(TransactionAttributeType.REQUIRED) {MANDATORY, REQUIRED, REQUIRES_NEW, SUPPORTS, NOT_SUPPORTED, NEVER}`	`<container-transaction>` `<trans-attribute>`	EJB 3.0 仕様の 13.3.7 項には、トランザクション属性を指定するメタデータ・アノテーションの代替として（またはトランザクション属性のメタデータ・アノテーションを補足またはオーバーライドする手段として）XML を使用できると記載されています。デプロイメント・ディスクリプタに指定されたトランザクション属性は、アノテーションに指定されたトランザクション属性をオーバーライドまたは補足するとみなされます。	OC4J は、このオーバーライド・ルールに準拠しています。

インターセプタ	`@Interceptors`
	`@ExcludeDefaultInterceptor`
	`@ExcludeClassInterceptor`
		`<interceptor-binding>`
		`<exclude-default-interceptors>`
		`<exclude-class-interceptors>`
		`<interceptor-order>`
			EJB 3.0 仕様の 12.8.2 項には、クラスへのインターセプタのバインドは付加的であると記載されています。XML を使用して、アノテーションにより定義されたインターセプタおよびインターセプタ・メソッドを拡張できます。仕様には、インターセプタの起動順序を指定する場合や、メタデータ・アノテーションで指定された順序をオーバーライドする場合にも代替として XML を使用できると記載されています。	OC4J では、複数の `interceptor-order` 定義は許可されません。`interceptor-order` を使用する場合、`exclude-class-interceptors` および `exclude-default-interceptors` フラグは無効にできません。また、`interceptor-order` の外部では `interceptor-class` を定義できません。これらは EJB 3.0 仕様には定義されていません。

表 1-6 XML によるアノテーションのオーバーライド（続き）

有効範囲	アノテーション	XML	EJB 3.0 仕様のオーバーライド・ルール	OC4J 固有の動作（リリース 10.1.3.1）
インターセプタ・コールバック	@PostConstruct @PreDestroy @PostActivate @PrePessivate @AroundInvoke	<post-construct-method> <pre-destroy-method> <post-activate-method> <pre-passivate-method> <around-invoke-method>	EJB 3.0 仕様の 12.8.1 項には、デプロイメント・ディスクリプタを使用してインターセプタを定義しているかどうか、またはアノテーションとデプロイメント・ディスクリプタ要素の組合せを使用しているかどうかにかかわらず、around-invoke メソッド、post-construct メソッド、pre-destroy メソッド、pre-passivate メソッドまたは post-activate メソッドとして特定のインターセプタ・クラスで指定できるのは、最大で 1 つのメソッドであると記載されています。	OC4J では、シングルトン制限を検証することなく、ライフ・サイクル・コールバック・メソッドがディスクリプタ・リストに追加されます。
セキュリティ識別情報	@DeclareRoles @RunAs	<security-role> <role-name>	EJB 3.0 仕様の 17.3.4 項には、XML の <security-identity> 要素を使用して、メタデータに指定されたセキュリティ識別情報をオーバーライドできると記載されています。<security-identity> 要素の値は、use-caller-identity または run-as です。	OC4J は、このオーバーライド・ルールに準拠しています。
メソッド許可	@RolesAllowed @DenyAll @PermitAll	<method-permission>	EJB 3.0 仕様の 17.3.2.2 項には、メソッド許可を指定するメタデータ・アノテーションの代替として（またはメソッド許可の値のメタデータ・アノテーションを補足またはオーバーライドする手段として）XML の <method-permission> 要素を指定できると記載されています。デプロイメント・ディスクリプタに明示的に指定された値は、アノテーションに指定された任意の値をオーバーライドします。オーバーライドの粒度は、メソッド・レベルです。メソッド許可の関係は、個々の <method-permission> 要素に定義されたすべてのメソッド許可の組合せとして定義されます。	OC4J は、このオーバーライド・ルールに準拠しています。

表 1-6 XML によるアノテーションのオーバーライド（続き）

有効範囲	アノテーション	XML	EJB 3.0 仕様のオーバーライド・ルール	OC4J 固有の動作（リリース 10.1.3.1）
EJB 参照	`@EJB` `@EJBs`	`<ejb-ref>` `<ejb-local-ref>`	EJB 3.0 仕様の 16.5.2.1 項には、XML エントリで `@EJB` または `@EJBs` アノテーションをオーバーライドする場合、次のルールが適用されると記載されています。 ■ 関連するデプロイメント・ディスクリプタ・エントリは、（デフォルトの、または明示的に指定された）アノテーションとともに使用される JNDI 名に基づいて配置します。 ■ `<remote>`、`<local>`、`<remote-home>` または `<local-home>` 要素を使用してデプロイメント・ディスクリプタに指定されるタイプと、`<ejb-link>` 要素で参照される Bean は、フィールドやプロパティのタイプか、または `@EJB` アノテーションの `beanInterface` 要素で指定されるタイプに割当て可能である必要があります。 ■ description を指定すると、アノテーションの description 要素がオーバーライドされます。 ■ 注入ターゲットを指定する場合、アノテーション付きのフィールドまたはプロパティ・メソッドを正確に指定する必要があります。	OC4J は、このオーバーライド・ルールに準拠しています。

表 1-6 XML によるアノテーションのオーバーライド（続き）

有効範囲	アノテーション	XML	EJB 3.0 仕様のオーバーライド・ルール	OC4J 固有の動作（リリース 10.1.3.1）
リソース参照	@Resource @Resources	<env-entry> <resource-ref> <resource-env-ref>	EJB 3.0 仕様の 16.2.3 項には、XML エントリで @Resource または @Resources アノテーションをオーバーライドする場合、次のルールが適用されると記載されています。 ■ 関連するデプロイメント・ディスクリプタ・エントリは、（デフォルトの、または明示的に指定された）アノテーションとともに使用される JNDI 名に基づいて配置します。 ■ デプロイメント・ディスクリプタに指定されるタイプは、フィールドやプロパティのタイプか、または @Resource アノテーションで指定されるタイプに割当て可能である必要があります。 ■ description を指定すると、アノテーションの description 要素がオーバーライドされます。 ■ 注入ターゲットを指定する場合、アノテーション付きのフィールドまたはプロパティ・メソッドを正確に指定する必要があります。 ■ <res-sharing-scope> 要素を指定すると、アノテーションの shareable 要素がオーバーライドされます。 ■ <res-auth> 要素を指定すると、アノテーションの authenticationType 要素がオーバーライドされます。	OC4J は、このオーバーライド・ルールに準拠しています。

表 1-6 XML によるアノテーションのオーバーライド（続き）

有効範囲	アノテーション	XML	EJB 3.0 仕様のオーバーライド・ルール	OC4J 固有の動作（リリース 10.1.3.1）
永続性ユニット	`@PersistenceUnits` `@PersistenceUnit`	`<persistence-units>` `<persistence-unit>`	EJB 3.0 仕様の 16.10.2.1 項には、XML エントリで `@PersistenceUnit` または `@PersistenceUnits` アノテーションをオーバーライドする場合、次のルールが適用されると記載されています。 ■ 関連するデプロイメント・ディスクリプタ・エントリは、（デフォルトの、または明示的に指定された）アノテーションとともに使用される JNDI 名に基づいて配置します。 ■ デプロイメント・ディスクリプタの `<persistence-unit-name>` 要素は、アノテーションの `unitName` 要素をオーバーライドします。 ■ 注入ターゲットを指定する場合、アノテーション付きのフィールドまたはプロパティ・メソッドを正確に指定する必要があります。	OC4J は、このオーバーライド・ルールに準拠しています。

表 1-6 XML によるアノテーションのオーバーライド（続き）

有効範囲	アノテーション	XML	EJB 3.0 仕様のオーバーライド・ルール	OC4J 固有の動作（リリース 10.1.3.1）
永続性コンテキスト	`@PersistenceContext` `@PersistenceContexts`	`<persistence-context>` `<persistence-contexts>`	EJB 3.0 仕様の 16.11.2 項には、XML エントリで `@PersistenceContext` または `@PersistenceContexts` アノテーションをオーバーライドする場合、次のルールが適用されると記載されています。 ■ 関連するデプロイメント・ディスクリプタ・エントリは、（デフォルトの、または明示的に指定された）アノテーションとともに使用される JNDI 名に基づいて配置します。 ■ デプロイメント・ディスクリプタの `<persistence-context-type>` 要素は、アノテーションの `type` 要素をオーバーライドします。 ■ 任意の `<persistence-property>` 要素は、`@PersistenceContext` または `@PersistenceContexts` アノテーションで指定された設定に追加されます。指定されたプロパティの名前が `@PersistenceContext` アノテーションで指定された名前と同じである場合、アノテーションで指定された値はオーバーライドされます。 ■ 注入ターゲットを指定する場合、アノテーション付きのフィールドまたはプロパティ・メソッドを正確に指定する必要があります。	OC4J は、このオーバーライド・ルールに準拠しています。
タイムアウト	`@Timeout`	`<timeout-method>`	EJB 3.0 仕様の 18.2.2 項には、`@Timeout` アノテーションが使用される場合、または Bean に `TimedObject` インタフェースが実装される場合、指定した `<timeout-method>` XML は、同じメソッドを参照する目的にのみ使用できると記載されています。	OC4J は、このオーバーライド・ルールに準拠しています。

表 1-6 XML によるアノテーションのオーバーライド（続き）

有効範囲	アノテーション	XML	EJB 3.0 仕様のオーバーライド・ルール	OC4J 固有の動作（リリース 10.1.3.1）
削除	@Remove (retainIfException =true|false)	<remove-method> <retain-if-exception>	EJB 3.0 仕様の 4.3.11 項には、XML の <remove-method> 要素の <retain-if-exception> サブ要素を明示的に指定して、@Remove アノテーションにより指定またはデフォルト設定された retainIfException の値をオーバーライドできると記載されています。	OC4J では、ステートフル・セッション Bean で削除メソッドが適切に処理されます。
アクティブ化構成	@MessageDriven activationConfig	<activation-config>	EJB 3.0 仕様の 5.4.13 項には、デプロイメント・ディスクリプタで指定されたアクティブ化構成プロパティは、@MessageDriven アノテーションで指定された設定に追加されると記載されています。同じ名前のプロパティが両方で指定されると、アノテーションで指定された値はデプロイメント・ディスクリプタの値でオーバーライドされます。	OC4J は、このオーバーライド・ルールに準拠しています。 **注意：**現在のリリースの OC4J には不具合があります。コンテナではオーバーライドが実行されない場合があります。かわりに、新規アクティブ化構成オブジェクトがリストに作成されます。
デプロイ	@StatefulDeployemt @StatelessDeployemt @MessageDrivenDeployemt	<session-deployment> <message-driven-deployment>	これらのアノテーションは EJB 3.0 仕様には定義されていません。これらは OC4J 固有のアノテーションです。	OC4J では、適用時に、XML のデプロイ設定でアノテーションがオーバーライドされます。

OC4J によるアノテーション属性 mappedName のサポート

OC4J では、@EJB および @Resource の mappedName 属性がサポートされます。これらのアノテーションの mappedName は、orion-ejb-jar.xml に含まれる session-deployment、entity-deployment および message-driven-deployment 要素の location 属性と同じものです。

OC4J では、@Stateless、@Stateful または @MessageDriven アノテーションの mappedName 属性はサポートされません。

セッション Bean とは

セッション Bean は、単一のクライアント / サーバー・セッションの継続時間中、クライアントによって作成される EJB 3.0 または EJB 2.1 Enterprise Bean コンポーネントです。セッション Bean は、クライアントの操作を実行します。セッション Bean はトランザクション対応ですが、システム障害の発生時にリカバリできません。セッション Bean オブジェクトは、ステートレス（1-31 ページの「ステートレス・セッション Bean とは」を参照）またはステートフル（メソッド・コールおよびトランザクション間で対話状態を維持、1-32 ページの「ステートフル・セッション Bean とは」を参照）です。セッション Bean が状態を維持する場合、OC4J は、オブジェクトをメモリーから削除する必要がある場合にこの状態を管理します（1-35 ページの「ステートフル・セッション Bean の非アクティブ化が発生する状況」を参照）。ただし、セッション Bean オブジェクトは自身の永続データを管理する必要があります。

クライアントの観点からは、セッション Bean は非永続オブジェクトであり、アプリケーション・サーバーで稼働するいくつかのビジネス・ロジックを実装します。たとえば、オンライン・ストア・アプリケーションでは、セッション Bean を使用して、`Cart` インタフェースを提供する `ShoppingCartBean` を実装します。クライアントは、このインタフェースを使用して、`purchaseItem` や `checkout` などのメソッドを起動します。

各クライアントには、固有のセッション・オブジェクトが割り当てられます。クライアントはセッション Bean のクラスのインスタンスに直接にはアクセスしません。クライアントは、Bean のホーム（11-7 ページの「ホーム・インタフェースの実装」を参照）およびコンポーネント（11-10 ページの「コンポーネント・インタフェースの実装」を参照）インタフェースを通じてセッション・オブジェクトにアクセスします。セッション Bean のクライアントは、Bean により提供されるインタフェースおよびクライアントにより使用されるインタフェースに応じて、ローカル・クライアント、リモート・クライアントまたは Web サービス・クライアント（ステートレス・セッション Bean のみ）となります。

OC4J は、各セッション Bean インスタンスのセッション・コンテキスト（1-37 ページの「セッション・コンテキストとは」を参照）を維持します。このセッション・コンテキストを使用して、コンテナにコールバック・リクエストを行います。

この項の内容は次のとおりです。

- ステートレス・セッション Bean とは
- ステートフル・セッション Bean とは
- セッション・コンテキストとは

詳細は、次を参照してください。

- 第 4 章「EJB 3.0 セッション Bean の実装」
- 第 11 章「EJB 2.1 セッション Bean の実装」

ステートレス・セッション Bean とは

ステートレス・セッション Bean は、対話状態のないセッション Bean です。特定のステートレス・セッション Bean クラスのインスタンスはすべて同一です。

ステートレス・セッション Bean およびそのクライアントは、メソッド間で状態または識別情報を共有しません。ステートレス・セッション Bean は、1 回のみ起動可能な Bean です。特定のクライアントに固有ではない、再利用可能なビジネス・サービスで使用されます。たとえば、一般的な為替換算、ローン金利の計算などに使用されます。ステートレス・セッション Bean には、クライアントから独立した、コール間に渡る読取り専用の状態が格納される場合があります。その後のコールは、プール内の他のステートレス・セッション Bean によって処理されます。情報は、1 回の起動中にのみ使用されます。

OC4J は、複数のクライアントを処理するために、これらのステートレス Bean のプールを維持しています。クライアントがリクエストを送信すると、プールからインスタンスが取得されます。Bean の情報を初期化する必要はありません。

ステートレス・セッション Bean のクライアントは、Web サービス・クライアントである場合があります。Web サービス・クライアント・ビューを提供できるのは、ステートレス・セッション Bean のみです。

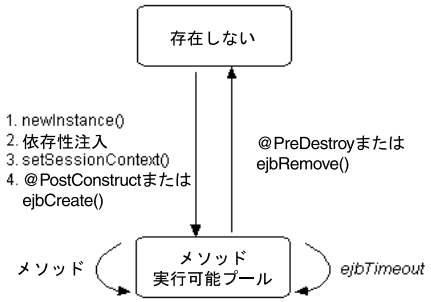
詳細は、次を参照してください。

- 4-2 ページの「EJB 3.0 ステートレス・セッション Bean の実装」
- 11-2 ページの「EJB 2.1 ステートレス・セッション Bean の実装」
- 30-2 ページの「Web サービスとしてのステートレス・セッション Bean の公開」

ステートレス・セッション Bean のライフ・サイクル

図 1-3 に、ステートレス・セッション Bean のライフ・サイクルを示します。`@PostConstruct` などのアノテーションは、EJB 3.0 のステートレス・セッション Bean にのみ適用されます。

図 1-3 ステートレス・セッション Bean のライフ・サイクル



EJB 3.0 と EJB 2.1 のステートレス・セッション Bean のライフ・サイクルは同一です。違いは、ライフ・サイクル・コールバック・メソッドの登録方法です（表 1-7 および表 1-8 を参照）。

表 1-7 に、アノテーションを使用して定義できる EJB 3.0 ステートレス・セッション Bean のオプションのライフ・サイクル・コールバック・メソッドをリストします。EJB 3.0 ステートレス・セッション Bean では、これらのメソッドを実装する必要はありません。

表 1-7 EJB 3.0 ステートレス・セッション Bean のライフ・サイクル・メソッド

アノテーション	説明
`@PostConstruct`	このオプションのメソッドは、Bean に対して最初のビジネス・メソッドを起動する前に、ステートフル・セッション Bean に対して起動されます。これは、任意の依存性注入がコンテナにより実行された後の時点です。
`@PreDestroy`	このオプションのメソッドは、インスタンスがコンテナにより削除されているときにステートフル・セッション Bean に対して起動されます。通常、インスタンスは保持していたリソースを解放します。

表 1-8 に、javax.ejb.SessionBean インタフェースでの指定に従って、ステートフル・セッション Bean が実装する必要のある EJB 2.1 ライフ・サイクル・メソッドをリストします。EJB 2.1 ステートフル・セッション Bean では、最低でも、すべてのコールバック・メソッド用に空の実装を用意する必要があります。

表 1-8 EJB 2.1 ステートレス・セッション Bean のライフ・サイクル・メソッド

EJB メソッド	説明
ejbCreate	コンテナは、Bean の作成直前にこのメソッドを起動します。このメソッドを使用して、データソースの取得など、クライアントに固有でない情報を初期化します。
ejbActivate	このメソッドは、ステートレス・セッション Bean に対しては決してコールされません。空の実装のみ用意します。
ejbPassivate	このメソッドは、ステートレス・セッション Bean に対しては決してコールされません。空の実装のみ用意します。
ejbRemove	コンテナは、ステートレス・セッション Bean を破棄する前にこのメソッドを起動します。このメソッドを使用して、データソースなどの外部リソースのクローズなど、必要なクリーンアップを実行します。
setSessionContext	コンテナは、Bean を最初にインスタンス化した後にこのメソッドを起動します。このメソッドを使用して、Bean のコンテキストの参照を取得します。詳細は、11-11 ページの「setSessionContext メソッドの実装」を参照してください。

詳細は、次を参照してください。

- 1-5 ページの「Enterprise Bean のライフ・サイクル」
- 5-5 ページの「EJB 3.0 セッション Bean のライフ・サイクル・コールバック・インターセプタ・メソッドの構成」
- 5-6 ページの「EJB 3.0 セッション Bean のインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドの構成」
- 12-4 ページの「EJB 2.1 セッション Bean のライフ・サイクル・コールバック・メソッドの構成」

ステートフル・セッション Bean とは

ステートフル・セッション Bean は、対話状態を維持するセッション Bean です。

ステートフル・セッション Bean は、対話型セッションで使用します。対話型セッションでは、インスタンス変数値やトランザクションの状態などをメソッド間で維持する必要があります。これらのセッション Bean は、単一クライアントの存続期間中、そのクライアントにマッピングされます。

ステートフル・セッション Bean は、メソッド・コール間で状態を維持します。したがって、各クライアントに対し、ステートフル・セッション Bean のインスタンスが 1 つずつ作成されます。それぞれのステートフル・セッション Bean には、個別のクライアントの識別情報と、1 対 1 のマッピングが含まれています。

コンテナが、（リソースを解放するために）メモリーからステートフル・セッション Bean を削除する必要があると判断した場合、コンテナは非アクティブ化（ディスクへの Bean のシリアライズ）によって Bean の状態を維持します。そのため、非アクティブ化する状態はシリアライズ可能である必要があります。ただし、システム障害が発生した場合、この情報は維持されません。Bean のインスタンスがクライアントによって再びリクエストされると、コンテナは以前に非アクティブ化した Bean インスタンスをアクティブ化します。

保存される状態のタイプには、リソースは含まれません。コンテナにより Bean 内の ejbPassivate メソッドが起動され、Bean がリソースのクリーンアップを行います。これらのリソースには、保持されたソケット、データベース接続、および静的情報が含まれているハッシュテーブルなどがあります。これらのリソースは、すべて ejbActivate メソッド中に再割当ておよび再作成可能です。

注意： ステートフル・セッション Bean の非アクティブ化はオフにできます（12-2 ページの「非アクティブ化の構成」を参照）。

Bean のインスタンスでエラーが発生すると、Bean 内で継続的に状態を保存するアクションを実行していないかぎり、状態が失われます。ただし、フェイルオーバーに備えて常に状態を保存する必要がある場合、実装にエンティティ Bean を使用することをお薦めします。または、`SessionSynchronization` インタフェースを使用して、状態をトランザクションによって維持することも可能です。

たとえば、ステートフル・セッション Bean は、ショッピング・カート・オンライン・アプリケーションのサーバー・サイドを実装可能です。このアプリケーションには、購入可能な商品のリストを返し、アイテムを顧客のショッピング・カートに入れ、発注を行い、顧客のプロファイルを変更するなどの作業を行うためのメソッドが含まれます。

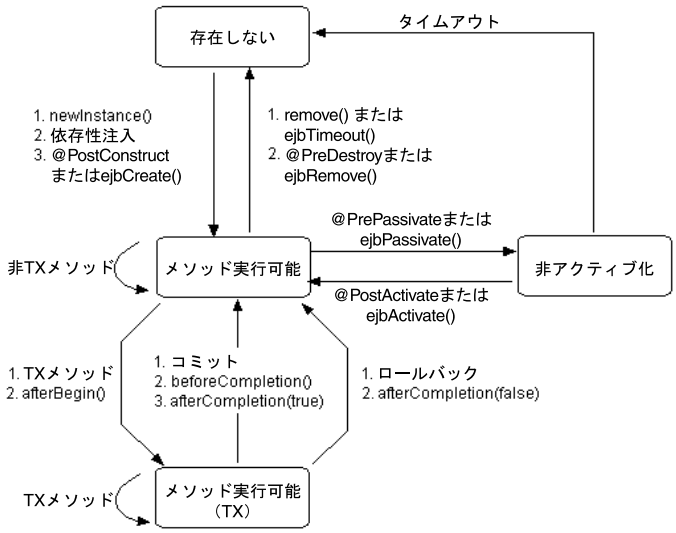
詳細は、次を参照してください。

- 4-3 ページの「EJB 3.0 ステートフル・セッション Bean の実装」
- 11-4 ページの「EJB 2.1 ステートフル・セッション Bean の実装」

ステートフル・セッション Bean のライフ・サイクル

図 1-4 に、ステートフル・セッション Bean のライフ・サイクルを示します。`@PostConstruct` などのアノテーションは、EJB 3.0 のステートフル・セッション Bean にのみ適用されます。

図 1-4 ステートフル・セッション Bean のライフ・サイクル



EJB 3.0 と EJB 2.1 のステートフル・セッション Bean のライフ・サイクルは同一です。違いは、ライフ・サイクル・コールバック・メソッドの登録方法です（表 1-9 および表 1-10 を参照）。

表 1-9 に、アノテーションを使用して定義できる EJB 3.0 ステートフル・セッション Bean のオプションのライフ・サイクル・コールバック・メソッドをリストします。EJB 3.0 ステートフル・セッション Bean では、これらのメソッドを実装する必要はありません。

表 1-9 EJB 3.0 ステートフル・セッション Bean のライフ・サイクル・メソッド

アノテーション	説明
`@PostConstruct`	このオプションのメソッドは、Bean に対して最初のビジネス・メソッドを起動する前に、ステートフル・セッション Bean に対して起動されます。これは、任意の依存性注入がコンテナにより実行された後の時点です。
`@PreDestroy`	このオプションのメソッドは、インスタンスがコンテナにより削除されているときにステートフル・セッション Bean に対して起動されます。通常、インスタンスは保持していたリソースを解放します。
`@PrePassivate`	コンテナは、ステートフル・セッション Bean を非アクティブ化する直前にこのメソッドを起動します。詳細は、次を参照してください。 ■ 1-35 ページの「ステートフル・セッション Bean の非アクティブ化が発生する状況」 ■ 1-35 ページの「非アクティブ化できるオブジェクト・タイプ」 ■ 1-36 ページの「非アクティブ化されたステートフル・セッション Bean の格納場所」
`@PostActivate`	コンテナは、以前に非アクティブ化したステートフル・セッション Bean を再びアクティブ化した直後にこのメソッドを起動します。

表 1-10 に、`javax.ejb.SessionBean` インタフェースでの指定に従って、ステートフル・セッション Bean が実装する必要のある EJB 2.1 ライフ・サイクル・メソッドをリストします。EJB 2.1 ステートフル・セッション Bean では、最低でも、すべてのコールバック・メソッド用に空の実装を用意する必要があります。

表 1-10 EJB 2.1 ステートフル・セッション Bean のライフ・サイクル・メソッド

EJB メソッド	説明
`ejbCreate`	コンテナは、Bean の作成直前にこのメソッドを起動します。ステートレス・セッション Bean は、このメソッドでは何も行いません。ステートフル・セッション Bean は、このメソッドで状態を初期化可能です。
`ejbActivate`	コンテナは、Bean を再びアクティブ化した直後にこのメソッドを起動します。
`ejbPassivate`	コンテナは、Bean を非アクティブ化する直前にこのメソッドを起動します。詳細は、次を参照してください。 ■ 1-35 ページの「ステートフル・セッション Bean の非アクティブ化が発生する状況」 ■ 1-35 ページの「非アクティブ化できるオブジェクト・タイプ」 ■ 1-36 ページの「非アクティブ化されたステートフル・セッション Bean の格納場所」
`ejbRemove`	コンテナは、セッション・オブジェクトを破棄する前にこのメソッドを起動します。このメソッドにより、ファイル・ハンドルなどの外部リソースのクローズなど、必要なクリーンアップが実行されます。
`setSessionContext`	コンテナは、Bean を最初にインスタンス化した後にこのメソッドを起動します。このメソッドを使用して、Bean のコンテキストの参照を取得します。詳細は、11-11 ページの「setSessionContext メソッドの実装」を参照してください。

詳細は、次を参照してください。

- 1-5 ページの「Enterprise Bean のライフ・サイクル」
- 5-5 ページの「EJB 3.0 セッション Bean のライフ・サイクル・コールバック・インターセプタ・メソッドの構成」
- 5-6 ページの「EJB 3.0 セッション Bean のインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドの構成」
- 12-4 ページの「EJB 2.1 セッション Bean のライフ・サイクル・コールバック・メソッドの構成」

ステートフル・セッション Bean の非アクティブ化が発生する状況 非アクティブ化を使用すると、コンテナは、Bean とその状態を 2 次記憶装置にシリアライズしてメモリーから削除することによって、非アクティブなアイドル状態の Bean インスタンスの対話状態を保持できます。非アクティブ化の前に、コンテナは `PrePassivate` または `ejbPassivate` メソッドを起動し、データベース接続、TCP/IP ソケットまたはオブジェクトのシリアライズによって透過的に非アクティブ化されないリソースなど、保持されたリソースを Bean 開発者がクリーンアップできるようにします。特定のオブジェクト・タイプのみシリアライズおよび非アクティブ化できます（1-35 ページの「非アクティブ化できるオブジェクト・タイプ」を参照）。

非アクティブ化はデフォルトで有効化されています。非アクティブ化の有効化および無効化の詳細は、12-2 ページの「非アクティブ化の構成」を参照してください。

OC4J は、次の基準の任意の組合せが満たされる場合にステートフル・セッション Bean を非アクティブ化します。

- アイドル・タイムアウトの超過
- 最大インスタンス数のしきい値超過または絶対的な最大インスタンス数の超過
- 最大 JVM メモリー消費のしきい値の超過
- OC4J インスタンスのシャットダウン

Bean の非アクティブ化は、最低使用頻度アルゴリズムを使用して実行されます。つまり、非アクティブ化に対して適格な Bean のうち、OC4J は使用頻度が最低の Bean を最初に非アクティブ化します。

また、OC4J がこの基準をチェックする頻度および基準が満たされたときに非アクティブ化するインスタンス数を指定できます。

この基準の構成の詳細は、12-2 ページの「非アクティブ化基準の構成」を参照してください。

非アクティブ化時のシリアライズに失敗した場合、コンテナは Bean をメモリーにリカバリして、処理前の状態にしようとします。非アクティブ化に失敗した Bean については、その後非アクティブ化は試行されません。また、アクティブ化に失敗した場合、Bean とその参照はコンテナから完全に削除されます。

非アクティブ化された Bean インスタンスのメソッドの 1 つをクライアントが起動すると、Bean を 2 次記憶装置からデシリアライズしてメモリーに戻すことによって、保持されていた対話状態のデータがアクティブ化されます。アクティブ化の前に、コンテナは `ejbActivate` メソッドを起動し、`ejbPassivate` 時に解放したリソースを開発者がリストアできるようにします。非アクティブ化の詳細は、EJB の仕様を参照してください。

ステートフル・セッション Bean では、1-35 ページの「非アクティブ化できるオブジェクト・タイプ」に示されている特定のタイプのオブジェクトのみが非アクティブ化されます。ユーザーがすべてのリソースを解放し、使用可能なタイプのオブジェクト内でのみ状態が存在するようにして、ステートフル・セッション Bean を非アクティブ化する準備をしていない場合、非アクティブ化は常に失敗します。

クラスタ内の非アクティブ化された Bean に対して新規 Bean データが伝播されると、その Bean インスタンスのデータは、伝播されたデータによって上書きされます。

非アクティブ化できるオブジェクト・タイプ ステートフル・セッション Bean が非アクティブ化されると、その Bean は 2 次記憶装置にシリアライズされます。シリアライズが成功するために、Bean の対話状態は、プリミティブ値と次のデータ型のみで構成されている必要があります。

- シリアライズ可能オブジェクト（フィールドがシリアライズ可能なフィールド・タイプのサブクラスで初期化されているかぎり、フィールド・タイプをシリアライズ可能として宣言する必要はないことに注意）
- NULL
- EJB ビジネス・インタフェースの参照
- サブクラスがシリアライズ可能でない場合でも、EJB リモート・インタフェースの参照

- サブクラスがシリアライズ可能でない場合でも、EJB リモート・ホーム・インタフェースの参照
- シリアライズ可能でない場合でも、EJB ローカル・インタフェースの参照
- シリアライズ可能でない場合でも、EJB ローカル・ホーム・インタフェースの参照
- シリアライズ可能でない場合でも、`SessionContext` オブジェクトの参照
- 環境命名コンテキスト（つまり、`java:comp/env` JNDI コンテキスト）またはその任意のサブコンテキストの参照
- `UserTransaction` インタフェースの参照
- リソース・マネージャのコネクション・ファクトリの参照
- シリアライズ可能でない場合でも、`EntityManager` オブジェクトの参照
- シリアライズ可能でない場合でも、`EntityManagerFactory` オブジェクトの参照
- `javax.ejb.Timer` オブジェクトの参照
- 直接にはシリアライズされず、オブジェクトのシリアライズ中にシリアライズ可能なオブジェクトによって EJB ビジネス・インタフェース、EJB ホーム・インタフェースおよび EJB コンポーネント・インタフェースへの参照、`SessionContext` オブジェクトへの参照、`java:comp/env` JNDI コンテキストとそのサブコンテキストへの参照、`UserTransaction` インタフェースへの参照、および `EntityManager` または `EntityManagerFactory`（あるいはその両方）への参照を置換することでシリアライズ可能になるオブジェクト

`PrePassivate` メソッド（5-5 ページの「EJB 3.0 セッション Bean のライフ・サイクル・コールバック・インターセプタ・メソッドの構成」を参照）または `ejbPassivate` メソッド（12-4 ページの「EJB 2.1 セッション Bean のライフ・サイクル・コールバック・メソッドの構成」を参照）の完了後に、すべての非一時的フィールドがこれらの型であることを確認してください。このメソッド内では、すべての一時フィールドまたは非シリアライズ可能フィールドを NULL に設定する必要があります。

非アクティブ化されたステートフル・セッション Bean の格納場所 デフォルトでは、OC4J は、ステートフル・セッション Bean を非アクティブ化したときに、シリアライズされたインスタンスを *<OC4J_HOME>*`¥j2ee¥home¥persistence` に書き込みます。

非アクティブ化によってこのディレクトリ内の領域が使用され、非アクティブ化された Bean が格納されます。非アクティブ化によって大量のディスク領域が割り当てられる場合は、使用可能な領域があるシステム上の別の場所にディレクトリを変更してください（12-4 ページの「非アクティブ化の場所の構成」を参照）。

セッション・コンテキストとは

OC4J は、各セッション Bean インスタンスの `javax.ejb.SessionContext` を維持し、このセッション・コンテキストを Bean に対して使用可能にします。Bean は、セッション・コンテキスト内のメソッドを使用して、コンテナへのコールバック・リクエストを送信できます。また、`EJBContext` から継承されたメソッドを使用できます（1-7 ページの「EJB コンテキストとは」を参照）。

詳細は、次を参照してください。

- 29-22 ページの「EJB 3.0 EJBContext へのアクセス」
- 29-30 ページの「EJB 2.1 EJBContext へのアクセス」

OC4J は、最初に Bean をインスタンス化した後で、セッション・コンテキストを初期化します。Bean がセッション・コンテキストを取得できるようにするのは、Bean プロバイダの役割です。コンテナは、トランザクション・コンテキストからはこのメソッドをコールしません。この時点で Bean がセッション・コンテキストを保存しなかった場合、Bean は二度とセッション・コンテキストにアクセスできなくなります。

コンテナはこのメソッドをコールする際、`SessionContext` オブジェクトの参照を Bean に渡します。Bean は、この参照を後の使用のために格納できます。

オブジェクトが参照する対話状態をセッション Bean インスタンスが（`setSessionContext` メソッドまたはリソース・インジェクションを使用して）`SessionContext` に格納する場合、OC4J はインスタンスの非アクティブ化の後も参照を保存およびリストアできます。OC4J は、アクティブ化の際に、別の機能的に同等の `SessionContext` オブジェクトで元の `SessionContext` オブジェクトを置換できます。

注意： OC4J では、`SessionContext` のメソッド `getInvokedBusinessInterface` がサポートされません。このメソッドをコールすると、OC4J により `UnsupportedOperationException` がスローされます。

JPA エンティティとは

Java Enterprise Edition 5（Java EE 5）EJB 3.0 仕様の一部である Java 永続性 API（JPA）により、Java の永続性は大幅に簡略化されています。この API を使用すると、オブジェクト・リレーショナル・マッピング・アプローチを通じて、Java EE 5 アプリケーション・サーバーの内部と Java Standard Edition 5（Java SE 5）アプリケーションの EJB コンテナの外部で動作する標準的で移植可能な方法に基づいて、Java オブジェクトをリレーショナル・データベース表にマッピングする方法を宣言的に定義できます。

JPA を使用すると、JPA エンティティとして任意の POJO クラスを指定できます。この Java オブジェクトには、（Java EE EJB コンテナの内部または Java SE アプリケーションの EJB コンテナの外部にある）JPA 永続性プロバイダから取得されるエンティティ・マネージャのサービスを通じてリレーショナル・データベースに永続化される非一時的フィールドが含まれます。

エンティティには、次の特徴があります。

- EJB 3.0 に準拠
- 軽量
- JPA エンティティ・マネージャと連携して永続データを管理
- 複雑なビジネス・ロジックを実行
- 複数の依存性のある Java オブジェクトを使用可能
- 主キーにより一意に識別可能

エンティティは、コンテナ管理の永続性を使用してリレーショナル・データベースに自動的に格納される永続データを表します。各データはデータベースなどのデータ記憶域システムの形式で永続的に格納されるため、これらのエンティティは永続的です。サーバー障害、フェイルオーバー、ネットワーク障害が発生しても存続し続けます。エンティティが再びインスタンス化されると、以前のインスタンスの状態が自動的にリストアされます。

エンティティは、ビジネス・エンティティをモデル化するか、または単一のビジネス・プロセス内の複数のアクションをモデル化します。エンティティは、データを使用するビジネス・サービスの提供、およびそのデータの計算によく使用されます。たとえば、開発者が、発注されたアイテムを取得し計算するエンティティを実装する場合があります。エンティティで、タスクの実行中に複数の依存性のある永続オブジェクトを管理できます。

エンティティは、リモートにアクセス可能なコンポーネントではないため、ファイングレインな永続オブジェクトを表すことができます。

エンティティはオブジェクトを集約し、JPA 永続性プロバイダのトランザクション、セキュリティおよび同時実行性サービスを使用してデータと関連オブジェクトを効率的に維持できます。

この項の内容は次のとおりです。

- JPA エンティティのコンテナ管理の永続性フィールドとは
- JPA エンティティのコンテナ管理の関連性フィールドとは
- データベース・リソースの競合の回避
- JPA エンティティのライフ・サイクル
- JPA エンティティの主キー
- JPA エンティティの問合せ方法

詳細は、第 6 章「JPA エンティティの実装」を参照してください。

JPA エンティティのコンテナ管理の永続性フィールドとは

コンテナ管理の永続性フィールドは、データベースに保存する必要のあるデータを表す状態フィールドです。

`@Transient` アノテーションを付けないかぎり、JPA エンティティのすべてのデータ・メンバーは永続性フィールドとみなされます。

エンティティの永続性ユニットで指定される JPA 永続性プロバイダ（2-10 ページの「persistence.xml ファイルとは」を参照）により、永続性フィールドがデータベースに永続化されることが保証されます。

デフォルトでは、JPA 永続性プロバイダにより、ほとんどの Java プリミティブ型、プリミティブ型のラッパーおよび列挙の基本マッピングが自動的に構成されます。このマッピングは、`@Basic`、`@Enumerated`、`@Temporal` および `@Lob` アノテーションを使用してカスタマイズできます。

JPA エンティティのコンテナ管理の関連性フィールドとは

コンテナ管理の関連性（CMR）フィールドは、1 つ以上の他の EJB 3.0 エンティティまたは EJB 2.1 コンテナ管理エンティティ Bean との永続関係を表す関連付けフィールドです。たとえば、受注管理アプリケーションでは、`OrderEJB` は `LineItemEJB` Bean のコレクションおよび単一の `CustomerEJB` Bean に関連している場合があります。

`@Transient` アノテーションを付けないかぎり、JPA エンティティのすべてのデータ・メンバーは永続性フィールドとみなされます。

エンティティの永続性ユニットで指定される JPA 永続性プロバイダ（2-10 ページの「persistence.xml ファイルとは」を参照）により、永続性フィールドがデータベースに永続化されることが保証されます。

アノテーションまたは `persistence.xml` を使用してエンティティを構成し、他のエンティティに対するマッピングを指定する必要があります。この構成により、エンティティ間の相互関係と、JPA 永続性プロバイダでリレーショナル・データベースに参照をマッピングする方法を指定します。

たとえば、関連マッピング・アノテーションの `@OneToOne`、`@ManyToOne`、`@OneToMany` および `@ManyToMany` を使用して、任意の永続関係の関連マッピングを構成できます。

エンティティ関連（E-R）には、次の特徴があります。

- 多重度 : 1 対 1、多対 1、1 対多、多対多という 4 つのタイプの多重度があり、Oracle Application Server では、そのすべてをサポートしています。
- 方向性 : 関連の方向は、双方向または単方向のどちらかになります。双方向の関連の場合、各エンティティ Bean には他の Bean を参照する関連フィールドがあります。エンティティ Bean のコードは、この関連フィールドを介して関連するオブジェクトにアクセスできます。エンティティ Bean に関連フィールドがある場合は、関連するオブジェクトが認識されています。たとえば、`ProjectEJB` Bean は複数の `TaskEJB` Bean が属していることを認識し、各 TaskEJB Bean は `ProjectEJB` Bean に属していることを認識している場合、これらの Bean には双方向の関連があります。単方向の関連の場合、1 つのエンティティ Bean にのみ他の Bean を参照する関連フィールドがあります。Oracle Application Server は、Enterprise Bean 間の双方向と単方向の両方の関連をサポートしています。
- Java 永続性問合せ言語のサポート : JP QL は、Enterprise JavaBeans 問合せ言語（EJB QL）の拡張版であり、バルク更新と削除、`JOIN`、`GROUP BY`、`HAVING`、投影、副問合せおよび名前付きパラメータの機能が追加されています。この言語では、静的問合せと動的問合せの両方がサポートされます。

  JP QL 問合せは、多くの場合、複数の関連をナビゲートします。関連の方向によって、問合せで Bean 間をナビゲートできるかどうかが決まります。

詳細は、次を参照してください。

- 7-10 ページの「JPA エンティティのコンテナ管理の関連性フィールドの構成」
- 第 8 章「JPA 問合せの実装」

JPA エンティティのライフ・サイクル

図 1-5 に、JPA エンティティのライフ・サイクルを示します。

図 1-5 JPA エンティティのライフ・サイクル

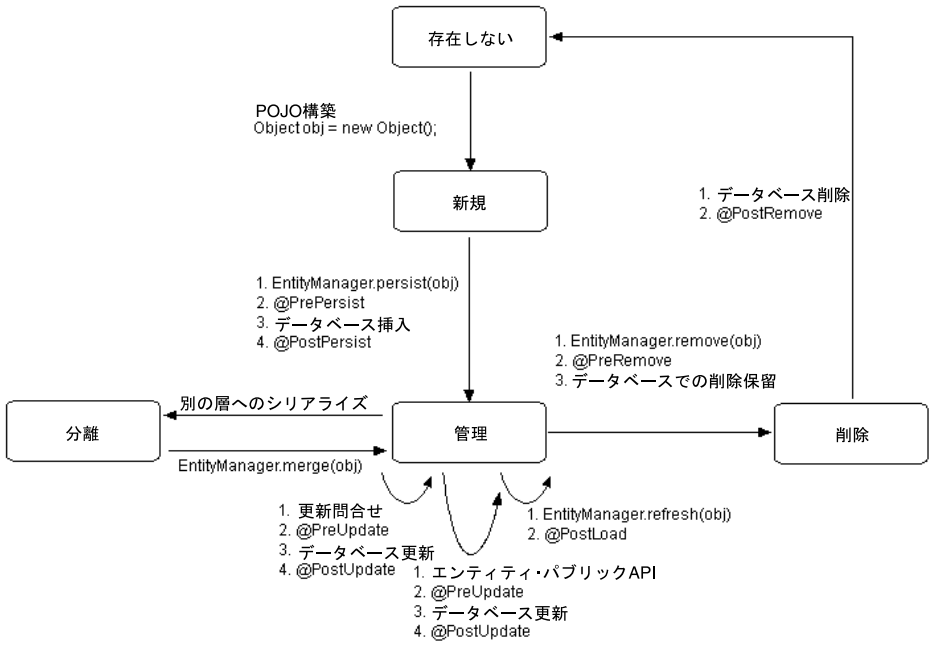


表 1-11 に、アノテーションを使用して定義できる JPA エンティティのオプションのライフ・サイクル・コールバック・メソッドをリストします。EJB 3.0 エンティティでは、これらのメソッドを実装する必要はありません。

表 1-11 JPA エンティティのライフ・サイクル・メソッド

アノテーション	説明
`@PrePersist`	このオプションのメソッドは、対応する `EntityManager` 永続化操作が実行される前にエンティティに対して起動されます。このコールバックは、これらの操作がカスケードされるすべてのエンティティに対して起動されます。このコールバックが `Exception` をスローした場合は、現在のトランザクションがロールバックされます。
`@PostPersist`	このオプションのメソッドは、対応する `EntityManager` 永続化操作が実行された後にエンティティに対して起動されます。このコールバックは、これらの操作がカスケードされるすべてのエンティティに対して起動されます。このメソッドは、データベース挿入操作の後に起動されます。これは、永続化操作の直後、フラッシュ操作の直後またはトランザクションの終了時の場合があります。このコールバックが `Exception` をスローした場合は、現在のトランザクションがロールバックされます。
`@PreRemove`	このオプションのメソッドは、対応する `EntityManager` 削除操作が実行される前にエンティティに対して起動されます。このコールバックは、これらの操作がカスケードされるすべてのエンティティに対して起動されます。このコールバックが `Exception` をスローした場合は、現在のトランザクションがロールバックされます。
`@PostRemove`	このオプションのメソッドは、対応する `EntityManager` 削除操作が実行された後にエンティティに対して起動されます。このコールバックは、これらの操作がカスケードされるすべてのエンティティに対して起動されます。このメソッドは、データベース削除操作の後に起動されます。これは、削除操作の直後、フラッシュ操作の直後またはトランザクションの終了時の場合があります。このコールバックが `Exception` をスローした場合は、現在のトランザクションがロールバックされます。
`@PreUpdate`	このオプションのメソッドは、エンティティ・データに対するデータベース更新操作の前に起動されます。これは、エンティティ状態更新時、フラッシュ操作時またはトランザクションの終了時の場合があります。 OC4J は、実際の更新が必要な場合のみ（データベースへの SQL の送信が準備されている場合のみ）このメソッドをコールします。この点を、実際の変更が必要かどうかにかかわらずコールされる post-update コールバックと比較してください。

表 1-11 JPA エンティティのライフ・サイクル・メソッド（続き）

アノテーション	説明
`@PostUpdate`	このオプションのメソッドは、エンティティ・データに対するデータベース更新操作の後に起動されます。これは、エンティティ状態更新時、フラッシュ操作時またはトランザクションの終了時の場合があります。 OC4J は、実際の更新が不要な場合（データベースに SQL を送信する必要がないと判断された場合）でも、このメソッドをコールします。オブジェクトが実際に変更された場合にのみ通知するには、pre-update コールバックを使用します。
`@PostLoad`	このオプションのメソッドは、エンティティがデータベースから現在の永続性コンテキストにロードされた後またはリフレッシュ操作が適用された後、かつ問合せ結果が返されるか、アクセスされるか、または関連付けが横断される前に起動されます。

詳細は、次を参照してください。

- 1-5 ページの「Enterprise Bean のライフ・サイクル」
- 7-18 ページの「JPA エンティティのライフ・サイクル・コールバック・メソッドの構成」
- 7-19 ページの「JPA エンティティのエンティティ・リスナー・クラスのライフ・サイクル・コールバック・リスナー・メソッドの構成」

JPA エンティティの主キー

各 JPA エンティティには、他のインスタンスから一意に識別するための主キーが存在する必要があります。主キー（または主キーとなる複合キー内のフィールド）は、永続フィールドである必要があります。

主キー内のすべてのフィールドは次の型に制限されます。

- プリミティブ・オブジェクト型
- シリアライズ可能型
- SQL 型にマッピングできる型

このリリースでは、単一の一般的なシリアライズ可能 Java プリミティブまたはオブジェクト型から構成される主キーを定義できます。Bean クラス内で宣言される主キー変数は、`public` として宣言する必要があります（7-2 ページの「JPA エンティティの単純な主キー・フィールドの構成」を参照）。

主キー値を自分で割り当てるか、より一般的には自動生成された主キーを作成できます（「JPA エンティティの自動主キー生成の構成」を参照）。

注意： エンティティ Bean の主キーが設定されると、EJB 3.0 仕様では、その変更が禁止されています。したがって、エンティティ・コンポーネント・インタフェースに主キー設定メソッドを公開しないでください。

詳細は、7-2 ページの「JPA エンティティの主キーの構成」を参照してください。

JPA エンティティの問合せ方法

EJB 3.0 では、`javax.persistence.EntityManager` を使用して EJB 3.0 エンティティを作成、検索、マージおよび維持します。エンティティを検索するには、`EntityManager` 問合せ API を使用します（1-42 ページの「JPA EntityManager 問合せ API について」を参照）。

適切な問合せ構文を使用して、選択基準を表現できます（1-42 ページの「JPA エンティティの問合せ構文について」を参照）。

問合せヒントを使用すると、EJB 3.0 JPA 永続性プロバイダでこの API のベンダー拡張を使用できます（8-4 ページの「JPA 問合せでの TopLink 問合せヒントの構成」を参照）。

JPA EntityManager 問合せ API について

EJB 3.0 では、`javax.persistence.EntityManager` および `javax.persistence.Query` API を使用して名前付き問合せまたは動的問合せを作成および実行できます。

`Query` API を使用して、パラメータのバインド、ヒントの構成および返される結果数の制御を行うことができます。

詳細は、次を参照してください。

- 1-42 ページの「JPA 動的（非定型）問合せ」
- 1-42 ページの「JPA の名前付き（事前定義）問合せ」
- 29-14 ページの「EntityManager を使用した JPA エンティティの問合せ」

JPA の名前付き（事前定義）問合せ 名前付き問合せは、EJB 2.1 finder メソッドを EJB 3.0 で改良したものです。EJB 3.0 では、メタデータを使用して名前付き問合せを実装でき（8-2 ページの「JPA 名前付き問合せの実装」を参照）、実行時に名前を指定して問合せを作成および実行できます（29-14 ページの「EntityManager での名前付き問合せの作成」を参照）。

OC4J では、Java 永続性問合せ言語とネイティブ SQL の名前付き問合せの両方がサポートされます。

JPA 動的（非定型）問合せ 動的問合せは、実行時に作成、構成および実行できる問合せです。名前付き問合せに加えて動的問合せを使用できます。

OC4J では、Java 永続性問合せ言語とネイティブ SQL の名前付き問合せの両方がサポートされます。

TopLink 問合せおよび式フレームワークを使用して動的問合せを作成することもできます（29-15 ページの「EntityManager を使用した動的 TopLink 式問合せの作成」を参照）。

JPA エンティティの問合せ構文について

表 1-12 に、JPA エンティティの問合せの定義に使用できる問合せ構文のタイプをまとめます。

表 1-12 OC4J による JPA エンティティの問合せ構文のサポート

問合せ構文	参照先
Java 永続性問合せ言語	1-42 ページの「Java 永続性問合せ言語の問合せ構文について」
ネイティブ SQL	1-55 ページの「EJB 2.1 のネイティブ SQL 問合せ構文について」

移植と最適化が可能なため、Java 永続性問合せ言語の使用をお薦めします。

Java 永続性問合せ言語の問合せ構文について Java 永続性問合せ言語は、移植と最適化が可能な形式で問合せセマンティクスを定義するために使用できる指定言語です。

SQL と似ていますが、Java 永続性問合せ言語はネイティブ SQL よりもはるかに優れています。SQL では列名を使用して表に対して問合せを行いますが、Java 永続性問合せ言語では、Bean の抽象スキーマ名およびフィールドを問合せ内で使用し、EJB 3.0 エンティティに対して問合せを行います。Java 永続性問合せ言語の文では、オブジェクト用語を使用します。JPA 永続性プロバイダは、アプリケーションのデプロイ時に、Java 永続性問合せ言語の文を適切なデータ

ベース SQL 文に変換します。したがって、JPA 永続性プロバイダは、エンティティ名とその永続フィールド名を、適切なデータベース表名と列名に変換します。Java 永続性問合せ言語は、OC4J でサポートされているすべてのデータベースに移植可能です。

EJB 3.0 では、Java 永続性問合せ言語構文には EJB 2.1 EJB QL（1-53 ページの「EJB 2.1 問合せ構文について」を参照）のすべての機能に加えて、バルク更新と削除、`JOIN` 操作、`GROUP BY`、`HAVING`、投影、副問合せ、EJB 3.0 の `EntityManager` API を使用した動的問合せ（1-42 ページの「JPA 動的（非定型）問合せ」を参照）での Java 永続性問合せ言語の使用などの追加機能があります。

詳細は、JSR-220 Enterprise JavaBeans バージョン 3.0 の Java 永続性 API 仕様の第 4 章を参照してください。

OC4J では、Java 永続性問合せ言語を次の重要な機能とともにサポートしています。

- 自動コード生成 : Java 永続性問合せ言語の問合せは、エンティティ Bean のデプロイメント・ディスクリプタで定義されます。この問合せは、Oracle Application Server への Enterprise Bean のデプロイ時に、コンテナによってターゲット・データ・ストアの SQL 言語に自動的に変換されます。この変換によって、コンテナ管理の永続性を使用するエンティティ Bean は移植可能となり、そのコードは特定タイプのデータ・ストアに固定されなくなります。
- 最適化された SQL コード生成 : SQL コードの生成時に、Oracle Application Server は、データベース・アクセスを効率的にするために、バルク SQL の使用や、バッチ処理された文のディスパッチなど、いくつかの最適化を行います。
- Oracle データベースおよび Oracle 以外のデータベースのサポート : Oracle Application Server では、あらゆるデータベース（Oracle、MS SQL-Server、IBM DB/2、Informix、Sybase など）に対して Java 永続性問合せ言語を実行できます。
- 関連 : Oracle Application Server は、単一のエンティティ Bean および関連を持つエンティティ Bean の両方について Java 永続性問合せ言語をサポートし、あらゆる多重度と方向性をサポートします。

EJB 3.0 を使用した場合、OC4J では、SQRT や日付、時刻、タイムスタンプの各オプションなど、EJB 3.0 永続性仕様で定義されているすべての拡張 Java 永続性問合せ言語機能がサポートされます。

EJB 3.0 のネイティブ SQL 問合せ構文について このリリースでは、TopLink JPA 永続性プロバイダは、指定された問合せ構文を受け取り（1-42 ページの「JPA エンティティの問合せ構文について」を参照）、基礎となるリレーショナル・データベースに固有の Structured Query Language（SQL）を生成します。

Java 永続性問合せ言語は、移植と最適化が可能なため、推奨される構文です。

ネイティブ SQL は、Java 永続性問合せ言語でサポートされない基礎となるリレーショナル・データベースの高度な問合せ機能を利用する場合に適しています。

OC4J では、名前付き問合せと動的問合せの両方でネイティブ SQL がサポートされます。

EJB 2.1 エンティティ Bean とは

エンティティ Bean は EJB 2.1 Enterprise Bean コンポーネントであり、永続データの管理および複雑なビジネス・ロジックの実行を行います。複数の依存性のある Java オブジェクトを使用可能で、主キーによって一意に識別可能です。

エンティティ Bean は、次のいずれかの方法により、ビジネス・データを永続的にします。

- コンテナ管理の永続性を備えたエンティティ Bean を使用して、コンテナによって自動的に行います（1-45 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean とは」を参照）。
- Bean 管理の永続性を備えたエンティティ Bean 内で実装されるメソッドを使用して、プログラムによって行います（1-49 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean とは」を参照）。これらのメソッドでは、永続性を管理するために、JDBC、SQLJ または永続性フレームワーク（TopLink など）を使用します。

コンテナ管理の永続性アーキテクチャと Bean 管理の永続性アーキテクチャを選択する方法の詳細は、1-62 ページの「Bean 管理の永続性を使用する場合とコンテナ管理の永続性を使用する場合」を参照してください。

エンティティ Bean のデータはデータベースなどのデータ記憶域の形式で永続的に格納されるため、エンティティ Bean は永続的です。サーバー障害、フェイルオーバー、ネットワーク障害が発生しても存続し続けます。エンティティ Bean が再びインスタンス化されると、以前のインスタンスの状態が自動的にリストアされます。OC4J は、エンティティ Bean をメモリーから削除する必要がある場合にこの状態を管理します（1-51 ページの「エンティティ Bean の非アクティブ化が発生する状況」を参照）。

エンティティ Bean は、ビジネス・エンティティをモデル化するか、または単一のビジネス・プロセス内の複数のアクションをモデル化します。エンティティ Bean は、データを使用するビジネス・サービスの提供、およびそのデータの計算によく使用されます。たとえば、開発者が、発注されたアイテムを取得し計算するエンティティ Bean を実装する場合があります。エンティティ Bean で、タスクの実行中に複数の依存性のある永続オブジェクトを管理できます。

一般的な設計パターンでは、エンティティ Bean をクライアント・インタフェースとして機能するセッション Bean と対にします。エンティティ Bean は、機能をカプセル化し、永続データおよび依存オブジェクト（通常はファイングレイン）との関連を表すコースグレインなオブジェクトとして機能します。したがって、クライアントをデータから分離できるため、データが変更されてもクライアントは影響を受けません。効率を上げるため、セッション Bean をエンティティ Bean と同一 JVM 上に置き、ローカル・インタフェースを通じて複数のエンティティ Bean 間を調整できます。これは、セッション・ファサード・デザインと呼ばれます。セッション・ファサード・デザインの詳細は、Web サイト `http://java.sun.com` を参照してください。

エンティティ Bean はオブジェクトを集約し、コンテナのトランザクション、セキュリティおよび同時実行性サービスを使用してデータと関連オブジェクトを効率的に維持できます。

この項の内容は次のとおりです。

- コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean とは
- Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean とは
- エンティティ・コンテキストとは
- データベース・リソースの競合の回避
- EJB 2.1 エンティティ Bean の問合せ方法
- エンティティ Bean の非アクティブ化が発生する状況
- エンティティ Bean のコミット・オプション

詳細は、第 13 章「EJB 2.1 エンティティ Bean の実装」を参照してください。

コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean とは

エンティティ Bean の永続データをコンテナで管理することを選択した場合は、コンテナ管理の永続性を備えたエンティティ Bean を定義します。コンテナ管理の永続性を備えたエンティティ Bean のクラスは、抽象クラス（コンテナが、実行時に使用される実装クラスを提供する）であり、その永続データは、単純なデータに対してはコンテナ管理の永続性フィールド（1-45 ページの「コンテナ管理の永続性フィールドとは」を参照）として指定され、コンテナ管理の永続性を備えた他のエンティティ Bean との関連に対してはコンテナ管理の関連性フィールド（1-45 ページの「コンテナ管理の関連性フィールドとは」を参照）として指定されます。この場合、コンテナにより、永続データのデータベースへの格納およびリロードが行われるため、Bean のデータの永続性を管理するための一部のコールバック・メソッドを実装する必要がありません（1-46 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル」を参照）。コンテナ管理の永続性を使用する場合、コンテナが永続的マネージャ・クラスを起動し、これによって永続的管理ビジネス・ロジックが提供されます。OC4J では、TopLink 永続性マネージャがデフォルトで使用されます。さらに、主キー（1-48 ページの「コンテナ管理の永続性を備えたエンティティ Bean の主キー」を参照）用の管理を提供する必要がありません。コンテナによって Bean のキーが提供されます。

詳細は、次を参照してください。

- 13-2 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の実装」
- 1-51 ページの「エンティティ・コンテキストとは」
- 1-63 ページの「データベース・リソースの競合の回避」
- 1-53 ページの「EJB 2.1 エンティティ Bean の問合せ方法」
- 1-51 ページの「エンティティ Bean の非アクティブ化が発生する状況」
- 1-51 ページの「エンティティ Bean のコミット・オプション」

コンテナ管理の永続性フィールドとは

コンテナ管理の永続性フィールドは、データベースに保存する必要のあるデータを表す状態フィールドです。

コンテナ管理の永続性フィールドを指定することで、OC4J に対して、フィールドの値を必ずデータベースに保存するよう指示します。コンテナ管理の永続性を備えたエンティティ Bean の他のすべてのフィールドは、非永続（一時的）とみなされます。

EJB 2.1 を使用した場合は、コンテナ管理の永続性フィールドを明示的に指定する必要があります（14-8 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean におけるコンテナ管理の永続性フィールドの構成」を参照）。

コンテナ管理の関連性フィールドとは

コンテナ管理の関連性フィールドは、コンテナ管理の永続性を備えた他の 1 つ以上のエンティティ Bean との永続関係を表す関連付けフィールドです。たとえば、受注管理アプリケーションでは、`OrderEJB` は `LineItemEJB` Bean のコレクションおよび単一の `CustomerEJB` Bean に関連している場合があります。

コンテナ管理の関連性フィールドを指定することで、OC4J に対して、コンテナ管理の永続性を備えた 1 つ以上の関連エンティティ Bean への参照を必ずデータベースに保存するよう指示します。このため、コンテナ管理の永続性を備えたエンティティ Bean 間の関連性は、コンテナ管理の関連性（またはコンテナ管理の永続性を備えたあるエンティティ Bean から別のエンティティ Bean へのマッピング）と呼ばれることがあります。

コンテナ管理の関連性には次の特性があります。

- 多重度 : 4 つのタイプの多重度があり、Oracle Application Server では、そのすべてをサポートしています。
- 方向性 : 関連の方向は、双方向または単方向のどちらかになります。双方向の関連の場合、各エンティティ Bean には他の Bean を参照する関連フィールドがあります。エンティティ Bean のコードは、この関連フィールドを介して関連するオブジェクトにアクセスできます。エンティティ Bean に関連フィールドがある場合は、関連するオブジェクトが認識されています。たとえば、`ProjectEJB` Bean は複数の `TaskEJB` Bean が属していることを認識し、各 TaskEJB Bean は `ProjectEJB` Bean に属していることを認識している場合、これらの Bean には双方向の関連があります。単方向の関連の場合、1 つのエンティティ Bean にのみ他の Bean を参照する関連フィールドがあります。Oracle Application Server は、Enterprise Bean 間の双方向と単方向の両方の関連をサポートしています。
- EJB QL 問合せサポート : EJB QL 問合せは、多くの場合、複数の関連をナビゲートします。関連の方向によって、問合せで Bean 間をナビゲートできるかどうかが決まります。OC4J では、EJB QL 問合せは、あらゆるタイプの多重度および双方向または単方向の関連について、コンテナ管理の関連性を横断できます。

詳細は、次を参照してください。

- 14-9 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean におけるコンテナ管理の関連性フィールドの構成」
- 第 16 章「EJB 2.1 問合せの実装」

コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル

図 1-6 に、コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクルを示します。

図 1-6 コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル

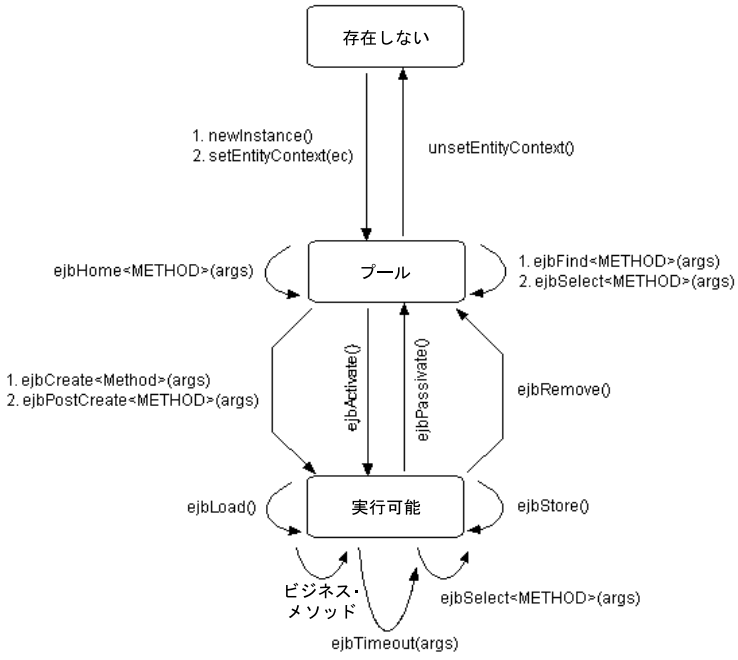


表 1-13 に、javax.ejb.EntityBean インタフェースでの指定に従って、コンテナ管理の永続性を備えたエンティティ Bean が実装する必要のある EJB 2.1 Enterprise Bean のライフ・サイクル・メソッドをリストします。コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean では、最低でも、すべてのコールバック・メソッド用に空の実装を用意する必要があります。

表 1-13 コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル・メソッド

EJB メソッド	説明
ejbCreate	ホーム・インタフェースで宣言された各 create メソッドに対し、対応する ejbCreate メソッドを実装する必要があります。クライアントが create メソッドを起動すると、コンテナはまずコンストラクタを起動してオブジェクトをインスタンス化し、次に対応する ejbCreate メソッドを起動します。 コンテナ管理の永続性を備えたエンティティ Bean の場合は、このメソッドを使用してコンテナ管理の永続性フィールドを初期化します。 すべての ejbCreate メソッドの戻り型は、Bean の主キーの型です。 オプションで、一意の主キーで Bean を初期化して返すこともできます。コンテナに依存して主キーを作成および初期化する場合は、null を返します。
ejbPostCreate	コンテナは、環境の設定後、このメソッドを起動します。各 ejbCreate メソッドに対し、同じ引数を持つ ejbPostCreate メソッドが存在する必要があります。 コンテナ管理の永続性を備えたエンティティ Bean の場合は、この実装を空のままにするか、実装を使用してエンティティ・コンテキストからパラメータを初期化できます。
ejbRemove	コンテナは、エンティティ Bean を破棄する前にこのメソッドを起動します。 コンテナ管理の永続性を備えたエンティティ Bean の場合は、この実装を空のままにするか、実装を使用して必要なクリーンアップ（たとえば、ファイル・ハンドルなどの外部リソースのクローズなど）を実行できます。
ejbStore	コンテナは、トランザクションのコミットの直前にこのメソッドを起動します。これにより、永続データを、データベースなどの外部リソースに格納します。 コンテナ管理の永続性を備えたエンティティ Bean では、この実装を空のままにできます。
ejbLoad	コンテナは、データをデータベースから再初期化する必要がある場合にこのメソッドを起動します。通常、これはエンティティ Bean のアクティブ化の後に行われます。 コンテナ管理の永続性を備えたエンティティ Bean では、この実装を空のままにできます。
ejbActivate	コンテナは、以前に非アクティブ化されたオブジェクトをアクティブ化する前にこのメソッドを直接コールします。リソースの再取得が必要な場合、このメソッドで実行します。
ejbPassivate	コンテナは、オブジェクトを非アクティブ化する前にこのメソッドをコールします。ejbActivate で容易に再作成できるリソースを解放することにより、記憶領域を節約します。通常は、ソケットまたはデータベース接続などのように、非アクティブ化できないリソースを解放します。これらのリソースは、ejbActivate メソッドで取得します。

詳細は、次を参照してください。

- 1-5 ページの「Enterprise Bean のライフ・サイクル」
- 14-17 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル・コールバック・メソッドの構成」

コンテナ管理の永続性を備えたエンティティ Bean の主キー

各エンティティ Bean には、他のインスタンスから一意に識別するための主キーが存在します。主キー（または主キーとなる複合キー内のフィールド）を、デプロイメント・ディスクリプタのコンテナ管理による永続的フィールドとして宣言する必要があります。

主キー内のすべてのフィールドは次の型に制限されます。

- プリミティブ・オブジェクト型
- シリアライズ可能型
- SQL 型にマッピングできる型

主キーは、次のいずれかの方法で定義できます。

- 単一の一般的なシリアライズ可能 Java プリミティブ型またはオブジェクト型から構成される単純な主キーを定義します。Bean クラス内で宣言される主キー変数は、`public` として宣言する必要があります（14-2 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の主キー・フィールドの構成」を参照）。
- シリアライズ可能な `<name>PK` クラス内の 1 つ以上の一般的なシリアライズ可能 Java プリミティブ型およびオブジェクト型から構成されるコンポジット主キー・クラスを定義します（「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のコンポジット主キー・クラスの構成」を参照）。

通常は、OC4J によって自動的に主キー値が割り当てられます。OC4J による主キー値の割当て方法を構成するには、TopLink 永続性 API を使用します。詳細は、次を参照してください。

- 3-15 ページの「TopLink EJB 2.1 永続性マネージャのカスタマイズ」
- 『Oracle TopLink 開発者ガイド』の「リレーショナル・プロジェクトにおける順序付けの概要」

> **注意：** エンティティ Bean の主キーが設定されると、EJB 2.1 仕様では、その変更が禁止されています。したがって、エンティティ Bean コンポーネント・インタフェースに主キー設定メソッドを公開しないでください。

詳細は、14-2 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の主キーの構成」を参照してください。

Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean とは

エンティティ Bean の永続データを自分で管理することを選択した場合は、Bean 管理の永続性を備えたエンティティ Bean を定義します。Bean 管理の永続性を備えたエンティティ Bean のクラスは、具象クラス（実行時に使用される実装をユーザーが提供）であり、その永続データは、単純なデータに対しては Bean 管理の永続性フィールド（1-49 ページの「Bean 管理の永続性フィールドとは」を参照）として指定され、Bean 管理の永続性を備えた他のエンティティ Bean との関連に対しては Bean 管理の関連性フィールド（1-49 ページの「Bean 管理の関連性フィールドとは」を参照）として指定されます。この場合、すべてのコールバック・メソッドを実装して、永続データのデータベースへの格納およびリロードなど、Bean のデータの永続性を管理する必要があります（1-49 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル」を参照）。Bean 管理の永続性を使用する場合は、永続性管理ビジネス・ロジックを実現するコードを提供する必要があります。また、主キーの管理を提供する必要があります（1-50 ページの「Bean 管理の永続性を備えたエンティティ Bean の主キー」を参照）。

Bean 管理の永続性を備えたエンティティ Bean を読取り専用として指定し（15-4 ページの「Bean 管理の永続性を備えた読取り専用エンティティ Bean の構成」を参照）、選択するコミット・オプションに応じて OC4J が Bean 管理の永続性を備えた読取り専用エンティティ Bean を提供する最適化を利用できます（1-51 ページの「エンティティ Bean のコミット・オプション」を参照）。

詳細は、次を参照してください。

- 13-7 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の実装」
- 1-51 ページの「エンティティ・コンテキストとは」
- 1-63 ページの「データベース・リソースの競合の回避」
- 1-53 ページの「EJB 2.1 エンティティ Bean の問合せ方法」
- 1-51 ページの「エンティティ Bean の非アクティブ化が発生する状況」
- 1-51 ページの「エンティティ Bean のコミット・オプション」

Bean 管理の永続性フィールドとは

Bean 管理の永続性を使用する場合は、記述するコードによって、Bean 管理の永続性を備えたエンティティ Bean で維持されるフィールドが決まります。

Bean 管理の関連性フィールドとは

Bean 管理の永続性を使用する場合は、記述するコードによって、Bean 管理の永続性を備えたエンティティ Bean 間の関連が実装されます。

Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル

表 1-14 に、`javax.ejb.EntityBean` インタフェースでの指定に従って、Bean 管理の永続性を備えたエンティティ Bean が実装する必要のあるライフ・サイクル・メソッドをリストします。

Bean 管理の永続性を備えたエンティティ Bean の場合は、すべてのライフ・サイクル・メソッドの完全な実装を提供する必要があります。

表 1-14 Bean 管理の永続性を備えたエンティティ Bean の EJB ライフ・サイクル・メソッド

EJB メソッド	説明
ejbCreate	ホーム・インタフェースで宣言された各 create メソッドに対し、対応する ejbCreate メソッドを実装する必要があります。クライアントが create メソッドを起動すると、コンテナはまずコンストラクタを起動してオブジェクトをインスタンス化し、次に対応する ejbCreate メソッドを起動します。ejbCreate メソッドにより、次の処理が実行されます。 ■ データ用に、データベース行などの永続記憶域を作成します。 ■ 一意の主キーを初期化して返します。
ejbPostCreate	コンテナは、環境の設定後、このメソッドを起動します。各 ejbCreate メソッドに対し、同じ引数を持つ ejbPostCreate メソッドが存在する必要があります。このメソッドを使用して、パラメータの初期化をエンティティ・コンテキスト内で、またはそこから実行することが可能です。
ejbRemove	コンテナは、セッション・オブジェクトを破棄する前にこのメソッドを起動します。このメソッドにより、ファイル・ハンドルなどの外部リソースのクローズなど、必要なクリーンアップが実行されます。
ejbStore	コンテナは、トランザクションのコミットの直前にこのメソッドを起動します。これにより、永続データを、データベースなどの外部リソースに格納します。
ejbLoad	コンテナは、データをデータベースから再初期化する必要がある場合にこのメソッドを起動します。通常、これはエンティティ Bean のアクティブ化の後に行われます。
ejbActivate	コンテナは、以前に非アクティブ化されたオブジェクトをアクティブ化する前にこのメソッドを直接コールします。リソースの再取得が必要な場合、このメソッドで実行します。
ejbPassivate	コンテナは、オブジェクトを非アクティブ化する前にこのメソッドをコールします。ejbActivate で容易に再作成できるリソースを解放することにより、記憶領域を節約します。通常は、ソケットまたはデータベース接続などのように、非アクティブ化できないリソースを解放します。これらのリソースは、ejbActivate メソッドで取得します。

詳細は、次を参照してください。

- 1-5 ページの「Enterprise Bean のライフ・サイクル」
- 15-8 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル・コールバック・メソッドの構成」

Bean 管理の永続性を備えたエンティティ Bean の主キー

エンティティ Bean の主キーは、特定のタイプのエンティティ Bean クラスのインスタンスを別のインスタンスと区別する、一意に識別可能な値です。各エンティティ Bean には、永続的な識別情報が関連付けられています。つまり、主キーを保有している場合に取得可能な一意の識別情報が含まれています。主キーがあれば、クライアントはエンティティ Bean を取得可能です。Bean が使用不可の場合、コンテナは Bean をインスタンス化し、永続データを再移入します。

一意のキーのタイプは、Bean プロバイダによって定義されています。

主キー内のすべてのフィールドは次の型に制限されます。

- プリミティブ・オブジェクト型
- シリアライズ可能型
- SQL 型にマッピングできる型
- RMI-IIOP で有効な値タイプとなる型
- hashCode() および equals(Object) メソッドの適切な実装を提供する型

主キーは、次のいずれかの方法で定義できます（どちらの場合も、Bean 管理の永続性を備えたエンティティ Bean の場合は、`ejbCreate` メソッドで主キーを作成します）。

- 主キーに、一般的な Java 型を定義します。Bean クラス内で宣言される主キー変数は、`public` として宣言する必要があります（15-2 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の主キー・フィールドの構成」を参照）。
- 主キーの型を、シリアライズ可能な *<name>*`PK` クラス内のシリアライズ可能なオブジェクトとして定義します（15-3 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の主キー・クラスの構成」を参照）。

エンティティ・コンテキストとは

OC4J は、コンテナ管理の永続性を備えた各 EJB 2.1 エンティティ Bean または Bean 管理の永続性を備えたエンティティ Bean インスタンスの `javax.ejb.EntityContext` を維持し、このエンティティ・コンテキストを Bean に対して使用可能にします。Bean は、エンティティ・コンテキスト内のメソッドを使用して、コンテナへのコールバック・リクエストを送信できます。また、`EJBContext` から継承されたメソッドを使用できます（1-7 ページの「EJB コンテキストとは」を参照）。

詳細は、次を参照してください。

- 13-21 ページの「setEntityContext および unsetEntityContext メソッドの実装」
- 29-30 ページの「EJB 2.1 EJBContext へのアクセス」

エンティティ Bean の非アクティブ化が発生する状況

エンティティ Bean の非アクティブ化は、コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean にのみ適用されます。

OC4J は、コンテナがエンティティ・オブジェクト識別情報からインスタンスの関連付けを解除すること、また使用可能なインスタンスのプールにインスタンスを戻すことを決定したときに、インスタンスを非アクティブ化します。OC4J は、インスタンスの `ejbPassivate` メソッドをコールして、インスタンスがプールにある間は保持できないリソース（通常は `ejbActivate` メソッドに割り当てられている）を解放する機会をインスタンスに与えます。このメソッドは、未指定のトランザクション・コンテキストで実行されます。エンティティ Bean は、このメソッド中にアクセッサ・メソッドを使用して永続状態または関連へのアクセスを試行することはできません。

エンティティ Bean のコミット・オプション

コミット・オプションにより、トランザクション・コミット時にエンティティ Bean インスタンスの状態が決定され、OC4J が特定のアプリケーション条件を最適化できる柔軟性が提供されます。

表 1-15 に、EJB 2.1 仕様によって定義されているコミット・オプションをリストし、OC4J でサポートされるオプションを示します。

表 1-15 OC4J によるエンティティ Bean のコミット・オプションのサポート

コミット・オプション	OC4J サポート	説明	インスタンス状態がデータベースに書き込まれるか	インスタンスは準備完了	インスタンス状態は有効なまま	利点	短所
A	✔[1]	キャッシュされた Bean: トランザクションの最後に、インスタンスは準備完了状態（キャッシュ済）になり、インスタンス状態は有効です（アクティブ化の後に `ejbLoad` が 1 回コールされる）。	✔	✔	✔	データベース・アクセスが最も少なくなります。	排他的アクセスが必要です。 複数スレッドが同じ Bean インスタンスを共有します（パフォーマンスが悪くなる）。
B		失効 Bean: トランザクションの最後に、インスタンスは準備完了状態（キャッシュ済）になりますが、インスタンス状態は無効です。トランザクションごとに `ejbLoad` および `ejbStore` がコールされます。	✔	✔		データベース・アクセスが中程度です。 同時リクエストが可能です。	同じデータを表す複数の Bean インスタンスのオーバーヘッドが生じます。 各トランザクションが `ejbLoad` をコールします。
C	✔[2]	プールされた Bean: トランザクションの最後に、インスタンスもその状態も無効になります（インスタンスは非アクティブ化され、プールに返される）。すべてのクライアント・コールで、`ejbActivate`、`ejbLoad`、ビジネス・メソッド、`ejbStore` および `ejbPassivate` がこの順序で実行されます。	✔			スケーラビリティが最大です。 同時リクエストが可能です。 接続を保持する必要がありません。	データベース・アクセスが最も多くなります（ビジネス・メソッド・コールのたび）。 キャッシングが行われません。

[1] Bean 管理の永続性を備えたエンティティ Bean のみ（1-53 ページの「コミット・オプションおよび BMP アプリケーション」を参照）。

[2] コンテナ管理の永続性を備えたエンティティ Bean のみ（1-52 ページの「コミット・オプションおよび CMP アプリケーション」を参照）。

コミット・オプションおよび CMP アプリケーション

TopLink 永続性マネージャを使用して OC4J にデプロイされた EJB 2.1 CMP アプリケーションでは、デフォルトで、OC4J はコミット・オプション C に近い TopLink 構成を使用します。このオプションにより、最も広範なアプリケーションに対してパフォーマンスとスケーラビリティが最大になります。

OC4J EJB 2.1 CMP は、ライフ・サイクル・メソッド・コールに関してオプション C に準拠します。ただし、TopLink 永続マネージャには次の新機能が導入されています。

- TopLink キャッシュを使用したキャッシングを提供します。
- インスタンスがすでに TopLink キャッシュ内にある場合は、トランザクションが開始するたびにインスタンスをデータソースと同期させることはありません。

TopLink ペシミスティックまたはオプティミスティック・ロック・ポリシーとともにロックまたは同期を使用して、同じ Bean への同時サービスを処理できます。これにより、インスタンスが失効データで更新されないことを保証したまま、同じインスタンスの同時アクセスのパフォーマンスを最大にすることができます。

ファイングレインな TopLink 構成変更の詳細は、次を参照してください。

- 3-15 ページの「TopLink EJB 2.1 永続性マネージャのカスタマイズ」
- 『Oracle TopLink 開発者ガイド』のロック・ポリシーの構成に関する項

コミット・オプションおよび BMP アプリケーション

OC4J にデプロイされた EJB 2.1 BMP アプリケーションの場合は、コミット・オプション A または C を構成できます（15-6 ページの「Bean 管理の永続性を備えたエンティティ Bean のコミット・オプションの構成」を参照）。

Bean 管理の永続性を備えたエンティティ Bean を読取り専用として構成した場合、OC4J はコミット・オプション A の特殊ケースを使用してパフォーマンスを向上させます。この場合、OC4J はインスタンスをキャッシュし、トランザクションのコミット時にインスタンスの更新や `ejbStore` のコールは行いません。詳細は、15-4 ページの「Bean 管理の永続性を備えた読取り専用エンティティ Bean の構成」を参照してください。

BMP コミット・オプション A および Bean 管理の永続性を備えた読取り専用エンティティ Bean は個別に使用できます（つまり、読取り専用を使用せずにコミット・オプション A で Bean 管理の永続性を備えたエンティティ Bean を構成でき、またコミット・オプション A で Bean 管理の永続性を備えたエンティティ Bean を構成せずに読取り専用を使用できる）。

EJB 2.1 エンティティ Bean の問合せ方法

EJB 2.1 エンティティ Bean インスタンスを問い合せるには、finder または select メソッドを使用します（1-56 ページの「finder メソッドについて」および 1-58 ページの「select メソッドについて」を参照）。

どちらの場合も、適切な問合せ構文を使用して、選択基準を表現します（1-53 ページの「EJB 2.1 問合せ構文について」を参照）。

詳細は、第 16 章「EJB 2.1 問合せの実装」を参照してください。

EJB 2.1 問合せ構文について

表 1-16 に、EJB 問合せの定義に使用できる問合せ構文のタイプをまとめます。

表 1-16 OC4J EJB 2.1 問合せ構文のサポート

問合せ構文	参照先
EJB QL	1-53 ページの「EJB 2.1 問合せ構文について」
TopLink	1-54 ページの「TopLink 問合せ構文について」
事前定義の finder	1-56 ページの「事前定義の TopLink finder」
デフォルトの finder	1-57 ページの「デフォルトの TopLink finder」
カスタム finder	1-57 ページの「カスタム TopLink finder」
カスタム select	1-59 ページの「カスタム TopLink select メソッド」
ネイティブ SQL	1-55 ページの「EJB 2.1 のネイティブ SQL 問合せ構文について」

移植と最適化が可能なため、EJB QL の使用をお薦めします。

EJB QL 問合せ構文について EJB QL は、移植と最適化が可能な形式で finder および select メソッド（1-56 ページの「finder メソッドについて」および 1-58 ページの「select メソッドについて」を参照）のセマンティクスを定義するために使用される指定言語です。EJB QL 文は、各 finder および select メソッドに関連付けられています。

SQL と似ていますが、EJB QL はネイティブ SQL よりもはるかに優れています。SQL では列名を使用して表に対して問合せを行いますが、EJB QL では、Bean の抽象スキーマ名およびコンテナ管理の永続性フィールドと関連性フィールドを問合せ内で使用し、コンテナ管理の永続性を備えたエンティティ Bean に対して問合せを行います。EJB QL 文では、オブジェクト用語を使用します。コンテナは、アプリケーションのデプロイ時に、EJB QL 文を適切なデータベース SQL 文に変換します。したがって、コンテナは、エンティティ Bean 名、コンテナ管理の永続性フィールド名およびコンテナ管理の関連性フィールド名を、適切なデータベース表名と列名に変換します。EJB QL は、OC4J でサポートされているすべてのデータベースに移植可能です。

EJB 2.1 では、EJB QL は SQL92 のサブセットであり、エンティティ Bean の抽象スキーマに定義されている関連へのナビゲーションを可能にする拡張機能があります。抽象スキーマは、エンティティ Bean のデプロイメント・ディスクリプタの一部で、Bean の永続フィールドと関連を定義します。「抽象」という用語によって、このスキーマは基礎となるデータ・ストアの物理的なスキーマと区別されます。EJB QL 問合せの有効範囲には、同じ EJB JAR ファイルにパッケージされている関連のエンティティ Bean の抽象スキーマが含まれるため、抽象スキーマ名は EJB QL 問合せで参照されます。

コンテナ管理の永続性を使用するエンティティ Bean の場合、EJB QL 問合せはすべての finder メソッド（`findByPrimaryKey` を除く）について定義する必要があります。OC4J を TopLink 永続性マネージャとともに使用すると、事前定義およびデフォルトの finder および select メソッドを利用できます（1-56 ページの「TopLink finder」および 1-59 ページの「カスタム TopLink select メソッド」を参照）。EJB QL 問合せによって、finder または select メソッドの起動時に EJB コンテナで実行される問合せが決まります。

Oracle Application Server では、EJB QL を次の重要な機能とともにサポートしています。

- 自動コード生成 : EJB QL 問合せは、エンティティ Bean のデプロイメント・ディスクリプタで定義されます。この問合せは、Oracle Application Server への Enterprise Bean のデプロイ時に、コンテナによってターゲット・データ・ストアの SQL 言語に自動的に変換されます。この変換によって、コンテナ管理の永続性を使用するエンティティ Bean は移植可能となり、そのコードは特定タイプのデータ・ストアに固定されなくなります。
- 最適化された SQL コード生成 : SQL コードの生成時に、Oracle Application Server は、データベース・アクセスを効率的にするために、バルク SQL の使用や、バッチ処理された文のディスパッチなど、いくつかの最適化を行います。
- Oracle データベースおよび Oracle 以外のデータベースのサポート : Oracle Application Server では、あらゆるデータベース（Oracle、MS SQL-Server、IBM DB/2、Informix、Sybase など）に対して EJB QL を実行できます。
- 関連を持つ CMP: Oracle Application Server は、単一のエンティティ Bean および関連を持つエンティティ Bean の両方について EJB QL をサポートし、あらゆる多重度と方向性をサポートします。

EJB 2.1 を使用した場合、OC4J は、EJB 2.1 で使用できない SQRT と日付、時刻、タイムスタンプの各オプションをサポートするために独自の EJB QL 拡張を提供します（16-9 ページの「OC4J EJB 2.1 EJB QL 拡張」を参照）。

TopLink 問合せ構文について このリリースでは、TopLink はデフォルトの永続性マネージャ（3-14 ページの「TopLink EJB 2.1 永続性マネージャ」を参照）であるため、TopLink 問合せおよび式フレームワークを使用して EJB 2.1 の finder または select メソッドの選択基準を表現できます。この EJB QL のかわりとなる手段には多数の利点があります（1-55 ページの「TopLink 問合せおよび式の利点」を参照）。

TopLink Workbench を使用して、`ejb-jar.xml` ファイルをカスタマイズし、TopLink 問合せおよび式フレームワークを使用して高度な finder および select メソッドを作成できます。

TopLink 永続性マネージャが提供する事前定義およびデフォルトの finder および select メソッドを利用することもできます（1-56 ページの「TopLink finder」および 1-59 ページの「カスタム TopLink select メソッド」を参照）。

詳細は、次を参照してください。

- 『Oracle TopLink 開発者ガイド』の TopLink の問合せの理解に関する項
- 『Oracle TopLink 開発者ガイド』の TopLink の式の理解に関する項
- 『Oracle TopLink 開発者ガイド』のディスクリプタ・レベルでの名前付き問合せの構成に関する項
- 『Oracle TopLink 開発者ガイド』の EJB ファインダの使用に関する項
- 『Oracle TopLink 開発者ガイド』の ejb-jar.xml ファイルの使用に関する項

TopLink 問合せおよび式の利点

TopLink 式フレームワークを使用すると、ドメイン・オブジェクト・モデルに基づいて問合せ検索基準を指定できます。

式には、SQL と比べてデータベースへのアクセス時に次の利点があります。

- EJB QL と同様にデータベースは抽象化されているため、式の方がメンテナンスが簡単です。
- 記述子またはデータベース表に対する変更は、アプリケーションの問合せ構造に影響しません。
- 従来の Java コール表記に似せて `Query` インタフェースを標準化することにより、式は判読性を向上させます。たとえば、`Employee` クラスの `Address` オブジェクトから番地を取得するために必要な Java コードは次のようになります。

  ```
  emp.getAddress().getStreet().equals("Meadowlands");
  ```

  同じ情報を取得するための式も同様です。

  ```
  emp.get("address").get("street").equal("Meadowlands");
  ```

- 式では、関連を共有する 2 つのクラス間で読取り問合せを透過的に問い合せることができます。これらのクラスがデータベース内の複数の表に格納されている場合、TopLink では、両方の表から情報を返すために適切な結合文が自動的に生成されます。
- 式では、複合操作が単純化されます。たとえば、次の Java コードは、給与が 10,000 を超える、「Meadowlands」に住むすべての従業員を取得します。

  ```
  ExpressionBuilder emp = new ExpressionBuilder();
  Expression exp = emp.get("address").get("street").equal("Meadowlands");
  Vector employees = session.readAllObjects(Employee.class,
    exp.and(emp.get("salary").greaterThan(10000)));
  ```

  TopLink では、そのコードから適切な SQL が自動的に生成されます。

  ```
  SELECT t0.VERSION, t0.ADDR_ID, t0.F_NAME, t0.EMP_ID, t0.L_NAME, t0.MANAGER_ID,
  t0.END_DATE, t0.START_DATE, t0.GENDER, t0.START_TIME, t0.END_TIME,t0.SALARY FROM
  EMPLOYEE t0, ADDRESS t1 WHERE (((t1.STREET = 'Meadowlands')AND (t0.SALARY > 10000))
  AND (t1.ADDRESS_ID = t0.ADDR_ID))
  ```

EJB 2.1 のネイティブ SQL 問合せ構文について このリリースでは、TopLink 永続性マネージャは、指定された問合せ構文を受け取り（1-53 ページの「EJB QL 問合せ構文について」または 1-54 ページの「TopLink 問合せ構文について」を参照）、基礎となるリレーショナル・データベースに固有の Structured Query Language（SQL）を生成します。

EJB QL は、移植と最適化が可能なため、推奨される構文です。

ネイティブ SQL は、EJB QL でサポートされない基礎となるリレーショナル・データベースの高度な問合せ機能を利用する場合に適しています。

EJB 2.1 および TopLink 問合せ構文を使用した場合、次のものを使用できます。

- ネイティブ SQL 文字列を受け取るデフォルトの finder（1-57 ページの「デフォルトの TopLink finder」を参照）
- ネイティブ SQL コールを使用するカスタム finder または select メソッド（1-56 ページの「TopLink finder」および 1-59 ページの「カスタム TopLink select メソッド」を参照）

ネイティブ SQL を使用するには、直接 JDBC コールを使用する必要があります。

finder メソッドについて

finder メソッドは、名前が find で始まる EJB メソッドであり、EJB の Home インタフェースで定義し（13-19 ページの「EJB 2.1 ホーム・インタフェースの実装」を参照）、問合せに関連付けてその EJB タイプの 1 つ以上のインスタンスを返します。デプロイ時に、OC4J は、関連付けられている問合せを実行するこのメソッドの実装を提供します。

finder メソッドは、クライアントがコンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean を取得する手段です。EJB 2.1 を使用して、次の処理を実行できます。

- OC4J および TopLink 永続性マネージャがコンテナ管理の永続性を備えたすべてのエンティティ Bean に提供する事前定義およびデフォルトの finder を公開します（1-56 ページの「事前定義の TopLink finder」および 1-57 ページの「デフォルトの TopLink finder」を参照）。
- カスタム EJB QL finder（16-2 ページの「EJB 2.1 EJB QL finder メソッドの実装」を参照）およびカスタム TopLink finder（1-57 ページの「カスタム TopLink finder」を参照）を定義します。

単一の EJB インスタンスを返す finder の戻り型は、その EJB インスタンスの型です。

複数の EJB インスタンスを返す finder の戻り型は、Collection です。一致する項目が見つからない場合は、空の Collection が返されます。重複した項目を返さないようにするには、関連付けられている EJB 問合せで DISTINCT キーワードを指定します。

すべての finder は、FinderException をスローします。

最低でも、findByPrimaryKey finder メソッドを公開して、主キーを使用して各エンティティの Bean の参照を取得する必要があります。

TopLink finder TopLink 永続性マネージャは、OC4J エンティティ Bean に対して様々な事前定義（1-56 ページの「事前定義の TopLink finder」を参照）およびデフォルト（1-57 ページの「デフォルトの TopLink finder」を参照）の finder を提供します。これらの finder は、他の finder と同様にクライアントに公開できます。対応する問合せを指定する必要はありません。カスタム TopLink finder も作成できます（1-57 ページの「カスタム TopLink finder」を参照）。

事前定義の TopLink finder

表 1-17 に、コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean に対して公開できる事前定義の finder をリストします。TopLink 永続性マネージャは、表 1-17 にリストされているメソッド名を予約しています。

表 1-17 事前定義の TopLink CMP finder

メソッド	引数	戻り型
findAll	()	Collection
findManyByEJBQL	(String ejbql) (String ejbql, Vector args)	Collection
findManyByQuery	(DatabaseQuery query) (DatabaseQuery query, Vector args)	Collection
findManyBySQL	(String sql) (String sql, Vector args)	Collection
findByPrimaryKey	(Object primaryKeyObject)	EJBObject または EJBLocalObject[1]
findOneByEJBQL	(String ejbql)	コンポーネント・インタフェース
findOneByEJBQL	(String ejbql, Vector args)	EJBObject または EJBLocalObject[1]
findOneByQuery	(DatabaseQuery query)	コンポーネント・インタフェース
findOneByQuery	(DatabaseQuery query, Vector args)	EJBObject または EJBLocalObject[1]
findOneBySQL	(String sql)	コンポーネント・インタフェース
findOneBySQL	(String sql, Vector args)	EJBObject または EJBLocalObject[1]

[1] finder がホーム・インタフェースとコンポーネント・インタフェースのどちらで定義されているかによって決まります。

例 1-3 に、2 つの事前定義の finder（findByPrimaryKey および findManyBySQL）を定義する EJBHome を示します。TopLink は、これらの finder の問合せ実装を提供します。

例 1-3 事前定義の TopLink finder の指定

```
public interface EmpBeanHome extends EJBHome {
    public EmpBean create(Integer empNo, String empName) throws CreateException;

    /**
    * Finder methods. These are implemented by the container. You can
    * customize the functionality of these methods in the deployment
    * descriptor through EJB-QL.
    **/

    // Predefined Finders: <query> element in ejb-jar.xml not required

    public Topic findByPrimaryKey(Integer key) throws FinderException;
    public Collection findManyBySQL(String sql, Vector args) throws FinderException

}
```

デフォルトの TopLink finder

名前が findBy<*CMP-FIELD-NAME*>（<*CMP-FIELD-NAME*> は Bean の永続フィールドの名前）と一致するエンティティ Bean のホーム・インタフェースで定義されている各 finder について、TopLink は、TopLink 式フレームワークを使用する TopLink 問合せ実装などの finder 実装を生成します。戻り型が単一の Bean 型である場合、TopLink は oracle.toplink.queryframework.ReadObjectQuery を作成します。戻り型が Collection の場合、TopLink は oracle.toplink.queryframework.ReadAllQuery を作成します。これらの finder は、他の finder と同様にクライアントに公開できます。対応する問合せを指定する必要はありません。

例 1-4 に、デフォルトの finder（findByEmpNo）を定義する EJBHome を示します。TopLink は、この finder の問合せ実装を提供します。

例 1-4 デフォルトの TopLink finder の指定

```
public interface EmpBeanHome extends EJBHome {
    public EmpBean create(Integer empNo, String empName) throws CreateException;

    /**
    * Finder methods. These are implemented by the container. You can
    * customize the functionality of these methods in the deployment
    * descriptor through EJB-QL.
    **/

    // Default Finder: <query> element in ejb-jar.xml not required

    public Topic findByEmpNo(Integer empNo);
}
```

カスタム TopLink finder

TopLink 問合せおよび式フレームワークを利用して、Call、DatabaseQuery、主キー、Expression、EJB QL、ネイティブ SQL、リダイレクト finder（任意のヘルパー・クラスの静的メソッドとして定義する実装に実行を委任する）など、高度な finder を定義できます。

EJB 2.1 を使用してカスタム TopLink finder を作成するには、TopLink Workbench で既存の toplink-ejb-jar.xml ファイルを使用します（16-5 ページの「TopLink Workbench の使用方法」を参照）。

select メソッドについて

エンティティ Bean の select メソッドは、コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean インスタンス内で内部的に使用する問合せメソッドです。select メソッドを抽象エンティティ Bean クラス自体の抽象メソッドとして定義し、EJB QL 問合せを関連付けます。ホームまたはコンポーネント・インタフェースではクライアントに select メソッドは公開しません。1 つ以上の select メソッドを定義すること、または select メソッドを定義しないことができます。関連付ける EJB QL 問合せに基づいて select メソッドの実装を提供するのはコンテナです。

通常は、ビジネス・メソッド内で select メソッドをコールして、コンテナ管理の永続性フィールドの値またはコンテナ管理の関連性フィールドのエンティティ Bean 参照を取得します。select メソッドは、起動側ビジネス・メソッドのトランザクション属性で判断されるトランザクション・コンテキストで実行します。

select メソッドには次のシグネチャがあります。

```
public abstract <ReturnType> ejbSelect<METHOD>(...) throws FinderException
```

- これは、`public` および `abstract` として宣言する必要があります。
- 戻り型は、select メソッドの戻り型ルールに準拠する必要があります（1-58 ページの「select メソッドが返す型」を参照）。
- メソッド名は、`ejbSelect` で始まる必要があります。
- メソッドは `javax.ejb.FinderException` をスローする必要があり、他のアプリケーション固有の例外もスローできます。

select メソッドは、起動されたエンティティ Bean インスタンスの識別情報に基づいていませんが、エンティティ Bean の主キーを引数として使用できます。これにより、特定のエンティティ Bean インスタンスに論理的に範囲設定された問合せが作成されます。

EJB 2.1 を使用している場合は、カスタム EJB QL select メソッド（16-6 ページの「EJB 2.1 EJB QL select メソッドの実装」を参照）を定義でき、カスタム TopLink select メソッド（1-59 ページの「カスタム TopLink select メソッド」を参照）を定義できます。

select メソッドが返す型 select メソッドの戻り型は、select が起動されるエンティティ Bean のタイプに制限されません。コンテナ管理の永続性またはコンテナ管理の関連性フィールドに対応する任意の型を返すことができます。

select メソッドは、次の戻り型ルールに準拠している必要があります。

- すべての値は `Object` として返される必要があり、プリミティブ型は対応する `Object` 型でラップされます（たとえば、`int` プリミティブ型は `Integer` オブジェクトでラップされる）。
- 単一オブジェクトの場合 : select メソッドが単一の項目のみを返す場合、コンテナは、select メソッド・シグネチャで指定されたものと同じ型を返します。

  複数のオブジェクトが返された場合は、`FinderException` が発生します。

  オブジェクトが見つからない場合は、`FinderException` が発生します。
- 複数オブジェクトの場合 : select メソッドが複数の項目を返す場合は、戻り型を `Collection` として定義する必要があります。

  ニーズに合せて `Collection` 型を選択します。たとえば、`Collection` には重複が含まれる場合があり、`Set` は重複を除外し、`SortedSet` は順序付き `Collection` を返します。

  オブジェクトが見つからない場合は、空の `Collection` が返されます。

  - コンテナ管理の永続性値の場合 : 複数のコンテナ管理の永続性値を返す場合、コンテナはオブジェクトの `Collection` を返し、EJB QL の select 文からその型を判断します。
  - コンテナ管理の関連性値の場合 : 複数のコンテナ管理の関連性値を返す場合、デフォルトでは、コンテナはオブジェクトの `Collection` を返します。その型はローカル Bean インタフェース型です。

これをアノテーション付きリモート Bean インタフェースまたはデプロイ XML 構成に変更できます。詳細は、16-6 ページの「EJB 2.1 EJB QL select メソッドの実装」を参照してください。

カスタム TopLink select メソッド EJB 2.1 を使用して、カスタム TopLink select メソッドを作成できます。

EJB 2.1 を使用している場合は、TopLink 問合せおよび式フレームワークを利用して、`Call`、`DatabaseQuery`、`Expression`、EJB QL、ネイティブ SQL など、任意の TopLink 問合せおよび式フレームワーク機能を利用できる高度な select メソッドを定義できます。
詳細は、16-9 ページの「TopLink Workbench の使用方法」を参照してください。

メッセージドリブン Bean とは

メッセージドリブン Bean（MDB）は、非同期メッセージ・コンシューマとして機能する EJB 3.0 または EJB 2.1 Enterprise Bean コンポーネントです。MDB にはクライアント固有の状態はありませんが、開いたデータベース接続や別の EJB へのオブジェクト参照などのメッセージ処理状態を含むことができます。クライアントは、MDB を使用して、Bean がメッセージ・リスナーとなっている送信先にメッセージを送信します。

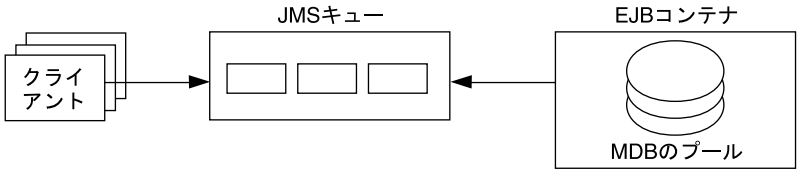
OC4J を使用すると、MDB を様々なメッセージ・プロバイダとともに使用できます（2-26 ページの「MDB で使用できるメッセージ・サービス・プロバイダ」を参照）。次のように、MDB を既存のメッセージ・プロバイダに関連付け、必要な設定の多くをコンテナが処理します。

- EJB コンテナによって、リスナーに対してタイプ `QueueReceiver` または `TopicSubscriber` のコンシューマが作成されます。
- デプロイ時に、EJB コンテナによって、MDB はコンシューマ（`QueueReceiver` または `TopicSubscriber`）およびそのファクトリに登録されます。
- EJB コンテナによって、メッセージ通知モードが指定されます。
- EJB コンテナは、メッセージをデキューし、メッセージ・リスナー・メソッドを使用してそれらのメッセージを MDB に渡します。
- EJB コンテナは、受信確認を送信します（送信するように構成されている場合）。

MDB の目的は、プール内に存在し、メッセージ・プロバイダからの受信メッセージを受け取り、処理することです。コンテナは、キューから Bean を起動して、キューからの受信メッセージを処理します。MDB を直接起動するオブジェクトはありません。MDB の起動は、すべてコンテナから指示されます。いったんコンテナが MDB を起動すると、他の Enterprise Bean または Java オブジェクトを起動して、リクエストを続行することが可能です。

MDB は、対話状態を保存せず、複数の受信リクエストの処理に使用される点において、ステートレス・セッション Bean に似ています。MDB は、クライアントから直接受信したリクエストを処理するのではなく、キューに入れられたリクエストを処理します。図 1-7 に、このように、クライアントがリクエストをキューに入れる様子を示します。コンテナは、キューからリクエストを取り出し、そのリクエストをプール内の MDB に渡します。

図 1-7 メッセージドリブン Bean



この項の内容は次のとおりです。

- メッセージドリブン Bean のライフ・サイクル
- メッセージ・ドリブン・コンテキストとは

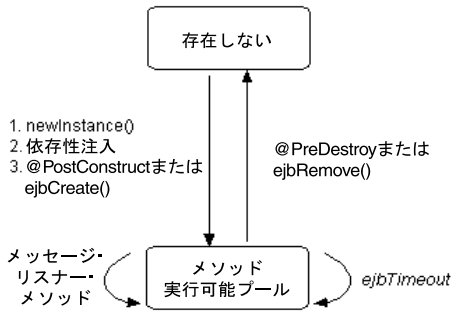
詳細は、次を参照してください。

- 第 9 章「EJB 3.0 メッセージドリブン Bean の実装」
- 第 17 章「EJB 2.1 メッセージドリブン Bean の実装」

メッセージドリブン Bean のライフ・サイクル

図 1-8 に、メッセージドリブン Bean のライフ・サイクルを示します。`@PostConstruct` などのアノテーションは、EJB 3.0 のメッセージドリブン Bean にのみ適用されます。

図 1-8 EJB 2.1 MDB のライフ・サイクル



EJB 3.0（表 1-18 を参照）と EJB 2.1（表 1-19 を参照）のメッセージドリブン Bean のライフ・サイクルは同一です。違いは、ライフ・サイクル・コールバック・メソッドの登録方法です。

表 1-18 に、アノテーションを使用して定義できる EJB 3.0 メッセージドリブン Bean のオプションのライフ・サイクル・コールバック・メソッドをリストします。EJB 3.0 メッセージドリブン Bean では、これらのメソッドを実装する必要はありません。

表 1-18 EJB 3.0 メッセージドリブン Bean のライフ・サイクル・メソッド

アノテーション	説明
`@PostConstruct`	このオプションのメソッドは、Bean に対して最初のビジネス・メソッドを起動する前に、メッセージドリブン Bean に対して起動されます。これは、任意の依存性注入がコンテナにより実行された後の時点です。
`@PreDestroy`	このオプションのメソッドは、インスタンスがコンテナにより削除されているときにメッセージドリブン Bean に対して起動されます。通常、インスタンスは保持していたリソースを解放します。

表 1-19 に、`javax.ejb.MessageDrivenBean` インタフェースでの指定に従って、メッセージドリブン Bean が実装する必要のある EJB 2.1 ライフ・サイクル・メソッドをリストします。EJB 2.1 メッセージドリブン Bean では、最低でも、すべてのコールバック・メソッド用に空の実装を用意する必要があります。

表 1-19 EJB 2.1 メッセージドリブン Bean のライフ・サイクル・メソッド

EJB メソッド	説明
`ejbCreate`	コンテナは、Bean の作成直前にこのメソッドを起動します。メッセージドリブン Bean は、このメソッドでは何も行いません。
`ejbRemove`	コンテナは、MDB を破棄する前にこのメソッドを起動します。このメソッドを使用して、ファイル・ハンドルなどの外部リソースのクローズなど、必要なクリーンアップを実行します。

詳細は、次を参照してください。

- 1-5 ページの「Enterprise Bean のライフ・サイクル」
- 10-12 ページの「EJB 3.0 MDB のライフ・サイクル・コールバック・インターセプタ・メソッドの構成」
- 10-13 ページの「EJB 3.0 MDB のインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドの構成」
- 18-11 ページの「EJB 2.1 MDB のライフ・サイクル・コールバック・メソッドの構成」

メッセージ・ドリブン・コンテキストとは

OC4J は、メッセージドリブン Bean インスタンスの `javax.ejb.MessageDrivenContext` を維持し、このメッセージドリブン・コンテキストを Bean に対して使用可能にします。Bean は、メッセージドリブン・コンテキスト内のメソッドを使用して、コンテナへのコールバック・リクエストを送信できます。

また、`EJBContext` から継承されたメソッドを使用できます（1-7 ページの「EJB コンテキストとは」を参照）。

詳細は、次を参照してください。

- 29-22 ページの「EJB 3.0 EJBContext へのアクセス」
- 29-30 ページの「EJB 2.1 EJBContext へのアクセス」

使用する Enterprise Bean のタイプ

この項の内容は次のとおりです。

- 使用するセッション Bean のタイプ
- Bean 管理の永続性を使用する場合とコンテナ管理の永続性を使用する場合

使用するセッション Bean のタイプ

ステートレス・セッション Bean は、主に、頻繁な短いリクエストを処理するための Bean のプールを持つ中間層アプリケーション・サーバーに使用されます。

Bean 管理の永続性を使用する場合とコンテナ管理の永続性を使用する場合

表 1-20 で、具体的に、BMP と CMP 両方の定義、およびそれらのプログラム面での違いと宣言の違いを示します。

表 1-20 Bean 管理の永続性とコンテナ管理の永続性の比較

管理項目	Bean 管理の永続性	コンテナ管理の永続性
永続性の管理	永続性管理を、`ejbStore`、`ejbLoad`、`ejbCreate` および `ejbRemove` `EntityBean` メソッド内に実装する必要があります。これらのメソッドには、永続データの格納およびリストアのためのロジックが含まれている必要があります。 たとえば、`ejbStore` メソッドの場合、エンティティ Bean のデータを適切なデータベースに格納するためのロジックが含まれている必要があります。そうでない場合、データが失われる可能性があります。	永続データの管理をユーザーが行う必要がありません。つまり、コンテナが、Bean のかわりに永続マネージャを起動します。 コミット前のデータの準備、またはデータベースからリフレッシュされた後のデータ操作には、`ejbStore` および `ejbLoad` を使用します。コンテナは、必ず、コミットの直前に `ejbStore` メソッドを起動します。さらに、CMP データをデータベースから再インスタンス化した直後に `ejbLoad` メソッドを起動します。
使用可能な finder メソッド	`findByPrimaryKey` メソッドおよびその他の finder メソッドが使用可能です。	`findByPrimaryKey` メソッドおよびその他の finder メソッド句が使用可能です。
コンテナ管理の永続性フィールドの定義	N/A	EJB デプロイメント・ディスクリプタ内で必須。主キーは、コンテナ管理の永続性フィールドとしても宣言する必要があります。
リソース格納先へのコンテナ管理の永続性フィールドのマッピング	N/A	必須。永続マネージャによって異なります。
永続マネージャの定義	N/A	Oracle 固有のデプロイメント・ディスクリプタ内で必須。OC4J では TopLink 永続性マネージャがデフォルトで使用されます。

CMP では、独自の低レベル JDBC ベースの永続性システムを作成しなくても、Java EE でサポートされるアプリケーション・サーバーおよびデータベースに EJB の状態を保存できる EJB 2.0 仕様にコンポーネントを構築できます。

BMP では、追加のコーディングやサポート作業を負担することでアプリケーションの永続性レイヤーを調整できます。

詳細は、次を参照してください。

- 1-45 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean とは」
- 1-49 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean とは」

データベース・リソースの競合の回避

OC4J、TopLink EJB 3.0 JPA 永続性プロバイダおよび EJB 2.1 永続性マネージャは、トランザクション分離（1-63 ページの「トランザクション分離」を参照）と同時実行性モード（1-64 ページの「同時実行性（ロック）モード」を参照）の組合せを使用して、データベース・リソースの競合を回避し、データベース表への同時アクセスを許可します。

トランザクション分離

同じデータに対する同時（パラレル）トランザクションが対話を許可される程度は、構成されているトランザクション分離のレベルによって決まります。ANSI/SQL は、表 1-21 に示すように、データベース・トランザクション分離の 4 つのレベルを定義します。各レベルには、パフォーマンスと次のような望ましくないアクションの防止との間にトレードオフがあります。

- 内容を保証しない読取り：トランザクションは、同時トランザクションによって書き込まれた未コミット・データを読み取ります。
- 非リピータブル・リード：トランザクションはデータを再び読み取り、初期読取り操作後にコミットされた他のトランザクションによってそのデータが変更されたことを検出します。
- 仮読取り：トランザクションは問合せを再実行し、返されるデータは、初期読取り操作後にコミットされた他のトランザクションによって変更されています。

表 1-21 トランザクション分離レベル

トランザクション分離レベル	内容を保証しない読取り	非リピータブル・リード	仮読取り
未コミット・データの読取り	○	○	○
コミット・データの読取り	×	○	○
リピータブル・リード	×	×	○
シリアライズ可能	×	×	×

デフォルトでは、OC4J、TopLink EJB 3.0 JPA 永続性プロバイダおよび EJB 2.1 永続性マネージャは、コミット読取りトランザクション分離を提供します。

トランザクション分離モードを構成するには、TopLink EJB 3.0 JPA 永続性プロバイダまたは EJB 2.1 永続性マネージャをカスタマイズする必要があります。

詳細は、次を参照してください。

- 3-4 ページの「JPA 永続性プロバイダのカスタマイズ」
- 3-15 ページの「TopLink EJB 2.1 永続性マネージャのカスタマイズ」
- 『Oracle TopLink 開発者ガイド』の作業ユニットとトランザクション分離に関する項
- 『Oracle TopLink 開発者ガイド』のデータベースのトランザクション分離レベルに関する項

同時実行性（ロック）モード

OC4J では、EJB 3.0 エンティティおよびコンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean 内でのリソースの競合およびパラレル実行を処理するため、同時実行性モードも提供されます。

Bean 管理の永続性を備えたエンティティ Bean は、Bean 実装自体内でリソースのロックを管理します。

同時実行性モードには、次のものが含まれます。

- オプティミスティック・ロック : 複数のユーザーがデータに対する読取りアクセス権を持ちます。ユーザーが変更を行おうとすると、アプリケーションによりバージョン・フィールド（書込みロック・フィールド）がチェックされ、ユーザーがデータを読み取った後にデータが変更されていないことが確認されます。

  オプティミスティック・ロックが有効化されると、TopLink は、データソースからオブジェクトを読み取る際にこのバージョン・フィールドの値をキャッシュします。クライアントがオブジェクトの書込みを試行すると、TopLink により、キャッシュされたバージョン値がデータソース内の現在のバージョン値と次の方法で比較されます。

  - 値が同じ場合は、TopLink により、オブジェクト内のバージョン・フィールドが更新され、データソースに対する変更がコミットされます。
  - 値が異なる場合は、このクライアントが最初にオブジェクトを読み取った後で別のクライアントがそのオブジェクトを更新したため、書込み操作が禁止されます。
- ペシミスティック・ロック : 更新する目的でデータにアクセスする最初のユーザーが、更新の完了までデータをロックします。これによりリソースの競合が管理され、パラレル実行はできません。エンティティ Bean を実行できるのは、一度に 1 ユーザーのみです。
- 読取り専用 : 複数のユーザーがパラレルでエンティティ Bean を実行できます。コンテナでは、Bean の状態を更新できません。

これらの同時実行性モードは Bean ごとに定義され、トランザクション境界に適用されます。

EJB 3.0 のデフォルトでは、JPA 永続性マネージャにより、データ整合性の確保はアプリケーションの役割であるとみなされます。`@Version` アノテーションを使用してバージョン・フィールドを指定し、JPA 管理のオプティミスティック・ロックを有効化することをお薦めします。

EJB 2.1 のデフォルトでは、TopLink 永続性マネージャは、オブジェクト変更がコミットされるたびに TopLink により更新されるコード生成の数値バージョン・フィールドを使用してオプティミスティック・ロックを実行します。

別の方法で同時実行性モードを構成するには、TopLink EJB 3.0 JPA 永続性プロバイダまたは EJB 2.1 永続性マネージャをカスタマイズする必要があります。

詳細は、次を参照してください。

- 3-4 ページの「JPA 永続性プロバイダのカスタマイズ」
- 3-15 ページの「TopLink EJB 2.1 永続性マネージャのカスタマイズ」
- 『Oracle TopLink 開発者ガイド』のロックに関する項
- 『Oracle TopLink 開発者ガイド』のロック・ポリシーの構成に関する項
- 『Oracle TopLink 開発者ガイド』の読取り専用ディスクリプタの構成に関する項

2

EJB アプリケーション開発について

この章の内容は次のとおりです。

EJB 開発ツールの使用方法

この項では、次のツールを使用した EJB アプリケーションの開発について説明します。

- JDeveloper の使用方法
- Eclipse の使用方法
- TopLink Workbench の使用方法

JDeveloper の使用方法

Oracle JDeveloper は、広範な自動化、迅速なデプロイとテストのための組込み OC4J およびその他多くの生産性向上機能を提供することで、Java EE アプリケーションの開発、パッケージングおよびデプロイを大幅に単純化します。次に例を示します。

- セッション Bean の開発 :
  `http://www.oracle.com/technology/products/jdev/101/viewlets/101/ejb30sessionbeanviewlet_viewlet_swf.htm`
- エンティティ Bean の開発 :
  `http://www.oracle.com/technology/products/jdev/101/viewlets/101/ejb30entitybeanviewlet_viewlet_swf.htm`

JDeveloper の詳細は、`http://www.oracle.com/technology/products/jdev/index.html` を参照してください。

Eclipse の使用方法

Eclipse は、Java EE アプリケーションの開発、パッケージングおよびデプロイを単純化する目的で広く採用されている統合開発環境です。

オラクル社では、EJB 3.0 エンティティのオブジェクト・リレーショナル（O/R）マッピングの定義および編集用に拡張可能なフレームワークおよび典型的なツールを Eclipse プラットフォームで開発しています。EJB 3.0 O/R マッピング・サポートは、作成ウィザードと自動化された初期マッピング・ウィザード、および動的な問題識別などのプログラミング支援を提供することで、マッピングの複雑さを最小化することに重点を置きます。

Eclipse での EJB 3.0 サポートの詳細は、`http://www.eclipse.org/dali/` を参照してください。

TopLink Workbench の使用方法

TopLink Workbench を使用して、次のファイルを作成および構成できます。

- EJB 3.0 `toplink-ejb-jar.xml` および `ejb3-toplink-sessions.xml` ファイル
- EJB 2.1 `toplink-ejb-jar.xml` ファイル
- `ejb-jar.xml` ファイル

詳細は、次を参照してください。

- 『Oracle TopLink 開発者ガイド』の TopLink Workbench の理解に関する項
- 2-6 ページの「EJB デプロイメント・ディスクリプタ・ファイルについて」

EJB に使用可能な OC4J サービス

表 2-1 に、OC4J に用意されている重要なサービスの一部をリストし、それらのサービスで使用できる EJB タイプを示します。

表 2-1 OC4J サービスと EJB サポート

OC4J サービス	ステートフル・セッション Bean	ステートレス・セッション Bean	EJB 3.0 エンティティ	CMP エンティティ Bean	BMP エンティティ Bean	メッセージドリブン Bean
2-15 ページの「EJB 永続性サービスについて」			✔	✔	✔	
2-17 ページの「EJB JNDI サービスについて」	✔	✔	✔	✔	✔	✔
2-17 ページの「EJB データソース・サービスについて」	✔	✔	✔	✔	✔	✔
2-20 ページの「EJB トランザクション・サービスについて」	✔	✔	✔	✔	✔	✔
2-24 ページの「EJB セキュリティ・サービスについて」	✔	✔	✔	✔	✔	✔
2-25 ページの「メッセージ・サービスについて」						✔
2-35 ページの「OC4J EJB アプリケーション・クラスタリング・サービスについて」	✔					
2-37 ページの「EJB タイマー・サービスについて」		✔		✔	✔	✔

OC4J サービスの詳細は、表 2-2 に示されている適切な OC4J マニュアルを参照してください。

表 2-2 Java EE のトピックに関する参照マニュアル

Java EE のトピック	トピックが説明されている OC4J マニュアル
最適化	『Oracle Application Server パフォーマンス・ガイド』
Web サービス	『Oracle Application Server Web Services 開発者ガイド』
セキュリティ	『Oracle Containers for J2EE セキュリティ・ガイド』
JNDI	『Oracle Containers for J2EE サービス・ガイド』
データソース	『Oracle Containers for J2EE サービス・ガイド』
RMI および RMI/IIOP	『Oracle Containers for J2EE サービス・ガイド』
CSiV2	『Oracle Containers for J2EE サービス・ガイド』
JMS	『Oracle Containers for J2EE サービス・ガイド』
クラスタリング	『Oracle Containers for J2EE サービス・ガイド』
タイマー	『Oracle Containers for J2EE サービス・ガイド』
J2EE Connector Architecture（J2CA）	『Oracle Containers for J2EE サービス・ガイド』
Java Object Cache	『Oracle Containers for J2EE サービス・ガイド』
HTTPS	『Oracle Containers for J2EE サービス・ガイド』
トランザクション（JTA）	『Oracle Containers for J2EE サービス・ガイド』
デフォルトの永続性	『Oracle TopLink 開発者ガイド』

EJB アプリケーションのパッケージ化およびデプロイの方法

この項の内容は次のとおりです。

パッケージ化について

Java EE アーキテクチャには、アプリケーションとその各種 Java EE コンポーネントをパッケージ化（またはアセンブル）する様々な方法が用意されています。

Java EE アプリケーションをパッケージ化する最も効率的な方法は、JDeveloper や Eclipse などの Java EE ツールを使用することです。

詳細は、次を参照してください。

- 2-2 ページの「EJB 開発ツールの使用方法」
- 第 27 章「EJB アプリケーションのパッケージ化」
- 『Oracle Application Server エンタープライズ・デプロイメント・ガイド』

デプロイについて

Java EE アプリケーションをパッケージ化した後で、そのアプリケーションを実行し、エンド・ユーザーが使用できるようにするために、OC4J にデプロイします。

Java EE アプリケーションを OC4J にデプロイする最も効率的な方法は、Oracle Enterprise Manager 10*g* Application Server Control を使用することです。

詳細は、次を参照してください。

- 2-5 ページの「OC4J が EJB モジュールをデプロイする順序」
- 2-6 ページの「EJB デプロイメント・ディスクリプタ・ファイルについて」
- 31-2 ページの「Oracle Enterprise Manager 10g Application Server Control の使用方法」
- 第 28 章「OC4J への EJB アプリケーションのデプロイ」
- 『Oracle Application Server エンタープライズ・デプロイメント・ガイド』

OC4J が EJB モジュールをデプロイする順序

OC4J は、application.xml デプロイメント・ディスクリプタに出現する順序で EJB モジュールをデプロイします。一般に、ロード順序はコンポーネントに固有であり、各コンポーネント・タイプの自然な順序に基づきます。

たとえば、例 2-1 に示す application.xml ファイルについて考えます。

例 2-1 application.xml

```
<application>
  <display-name>master-application</display-name>
  <module>
    <ejb>ejb1.jar</ejb>
  </module>
  <module>
    <ejb>ejb2.jar</ejb>
  </module>
  <module>
    <java>appclient.jar</java>
  </module>
  <module>
    <web>
        <web-uri>clientweb.war</web-uri>
        <context-root>webapp</context-root>
    </web>
  </module>
  <module>
    <ejb>ejb3.jar</ejb>
  </module>
```

この application.xml ファイルに基づいて、OC4J はコンポーネントを次の順序でロードします。

1. ejb1
2. ejb2
3. ejb3
4. clientweb.war
5. appclient.jar

EJB デプロイメント・ディスクリプタ・ファイルについて

このセクションでは、OC4J にデプロイされる EJB アプリケーションで使用する様々な EJB デプロイメント・ディスクリプタ・ファイルについて説明します。

表 2-3 に、OC4J にデプロイされる EJB アプリケーションで使用する様々な EJB デプロイメント・ディスクリプタ・ファイルをリストします。各デプロイメント・ディスクリプタ・ファイルについて、デプロイメント・ディスクリプタが適用される EJB タイプと、使用している EJB 仕様でデプロイメント・ディスクリプタがオプション、必須、適用なしのいずれであるかを示します。

表 2-3 OC4J EJB デプロイメント・ディスクリプタ・ファイル

デプロイメント・ディスクリプタ・ファイル	セッション Bean	JPA エンティティ	EJB 2.1 エンティティ Bean	メッセージドリブン Bean	EJB 3.0	EJB 2.1
2-6 ページの「ejb-jar.xml ファイルとは」	✔		✔	✔	オプション	必須
2-7 ページの「orion-ejb-jar.xml ファイルとは」	✔	✔[1]	✔	✔	オプション	オプション
2-8 ページの「toplink-ejb-jar.xml ファイルとは」		✔	✔		オプション	必須
2-9 ページの「ejb3-toplink-sessions.xml ファイルとは」		✔			オプション	適用なし
2-10 ページの「persistence.xml ファイルとは」		✔			オプション	適用なし
2-11 ページの「orm.xml ファイルとは」		✔			オプション	適用なし

[1] `<entity-deployment>` 要素の `disable-default-persistent-unit` 属性のみ。

ejb-jar.xml ファイルとは

`ejb-jar.xml` ファイルは、EJB デプロイメント・ディスクリプタ・ファイルであり、使用される場合は次の内容を記述します。

- 含まれるすべての Enterprise Bean に関する必須構造情報
- コンテナ管理の関連性のディスクリプタ（存在する場合）
- ejb-jar の ejb-client-jar ファイルのオプションの名前
- オプションのアプリケーション・アセンブリ・ディスクリプタ

必要な場合、`ejb-jar.xml` ファイルは Java EE アプリケーション・サーバーに適用される EJB 情報を記述します。この情報は、アプリケーション・サーバー固有の EJB デプロイメント・ディスクリプタ・ファイルによって補強される場合があります（2-7 ページの「orion-ejb-jar.xml ファイルとは」および 2-8 ページの「toplink-ejb-jar.xml ファイルとは」を参照）。

詳細は、26-2 ページの「ejb-jar.xml ファイルの構成」を参照してください。

EJB 3.0

EJB 3.0 を使用している場合、このデプロイメント・ディスクリプタ・ファイルはオプションです。かわりにアノテーションを使用できます。このリリースでは、OC4J で、セッション Bean およびメッセージドリブン Bean のすべてのオプションに対して EJB 3.0 アノテーションと `ejb-jar.xml` の両方の使用がサポートされます。`ejb-jar.xml` ファイルは、EJB 3.0 エンティティでは使用されません。`ejb-jar.xml` ファイル内の構成によって、アノテーションがオーバーライドされます（1-22 ページの「デプロイメント・ディスクリプタ・エントリによるアノテーションのオーバーライド」を参照）。

EJB 3.0 エンティティの場合、アノテーションを使用するか、TopLink JPA 永続性プロバイダのデプロイ XML ファイル（`toplink-ejb-jar.xml` および `ejb3-toplink-sessions.xml`）を使用する必要があります。

詳細は、次を参照してください。

- 「toplink-ejb-jar.xml ファイルとは」
- 「ejb3-toplink-sessions.xml ファイルとは」

EJB 2.1

EJB 2.1 を使用している場合、このデプロイメント・ディスクリプタ・ファイルは必須です。

XML 参照

このデプロイメント・ディスクリプタ・ファイルの XML 参照は、使用している EJB バージョンに依存します。

EJB 3.0 の場合、このデプロイメント・ディスクリプタ・ファイルは `http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd` にある XML Schema 文書に準拠します。

EJB 2.1 の場合、このデプロイメント・ディスクリプタ・ファイルは `http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd` にある XML Schema 文書に準拠します。

orion-ejb-jar.xml ファイルとは

`orion-ejb-jar.xml` ファイルは、OC4J 固有オプションをすべて含む EJB デプロイメント・ディスクリプタ・ファイルです。このファイルでは、`ejb-jar.xml` ファイルで指定する構成を拡張します（2-6 ページの「ejb-jar.xml ファイルとは」を参照）。

詳細は、次を参照してください。

- 26-3 ページの「orion-ejb-jar.xml ファイルの構成」
- 付録 A「orion-ejb-jar.xml 要素の XML 参照」

EJB 3.0

EJB 3.0 を使用している場合、このファイルはオプションです。`orion-ejb-jar.xml` ファイルを使用せずにデプロイし、OC4J 固有のアノテーション（`@StatelessDeployment`、`@StatefulDeployment`、`@MessageDrivenDeployment` など）または Application Server Control を使用して OC4J 固有オプションを設定できます。`orion-ejb-jar.xml` ファイルのベンダー拡張設定は、OC4J 固有のアノテーションを使用した拡張設定に優先します。`orion-ejb-jar.xml` ファイル内の構成によって、アノテーションがオーバーライドされます（1-22 ページの「デプロイメント・ディスクリプタ・エントリによるアノテーションのオーバーライド」を参照）。

詳細は、次を参照してください。

- 5-12 ページの「EJB 3.0 セッション Bean の OC4J 固有のデプロイ・オプションの構成」
- 10-20 ページの「EJB 3.0 MDB の OC4J 固有のデプロイ・オプションの構成」

EJB 2.1

EJB 2.1 を使用している場合、`orion-ejb-jar.xml` ファイルはすべての OC4J 固有オプションに対して必須です。

詳細は、3-15 ページの「TopLink EJB 2.1 永続性マネージャのカスタマイズ」を参照してください。

XML 参照

このデプロイメント・ディスクリプタ・ファイルは、`http://www.oracle.com/technology/oracleas/schema/index.html` にある XML Schema 文書に準拠します。

toplink-ejb-jar.xml ファイルとは

`toplink-ejb-jar.xml` ファイル（TopLink `project.xml` ファイルとも呼ぶ）は、TopLink JPA プレビュー永続性構成ディスクリプタ・ファイルであり、使用されている場合は、TopLink ディスクリプタやマッピングなどの TopLink プロジェクトレベルのオプションを記述します（『Oracle TopLink 開発者ガイド』のリレーショナル・プロジェクトの構成に関する項を参照）。

注意： OC4J では、TopLink Essentials JPA 永続性プロバイダがデフォルトで使用されます。この場合、TopLink JPA 拡張を使用して、TopLink ディスクリプタレベルのオプション（マッピングを含む）を構成できます（3-5 ページの「TopLink Essentials JPA 永続性を使用した TopLink API への実行時アクセス」を参照）。

詳細は、26-2 ページの「toplink-ejb-jar.xml ファイルの構成」を参照してください。

EJB 3.0

デフォルトの TopLink Essentials JPA 永続性プロバイダとともに EJB 3.0 を使用している場合、このファイルは使用されません。

EJB 3.0 を使用している場合、`toplink-ejb-jar.xml` ファイルは TopLink JPA プレビュー永続性プロバイダ構成のカスタマイズにのみ使用されます（3-4 ページの「JPA 永続性プロバイダのカスタマイズ」を参照）。このファイルを使用して TopLink 永続性プロバイダをカスタマイズする場合は、`ejb3-toplink-sessions.xml` ファイルも使用する必要があります（2-9 ページの「ejb3-toplink-sessions.xml ファイルとは」を参照）。

EJB 2.1

EJB 2.1 を使用している場合、`toplink-ejb-jar.xml` ファイルはオプションです。アプリケーションでこのファイルを省略する場合は、OC4J で自動的に作成されるように構成できます（14-6 ページの「デフォルトの関連性生成の構成」を参照）。または、このファイルを使用して TopLink 永続性オプションを自分で構成できます（3-15 ページの「TopLink EJB 2.1 永続性マネージャのカスタマイズ」を参照）。

XML 参照

`toplink-ejb-jar.xml` ファイルは、`<OC4J_HOME>¥toplink¥config¥xsds` にある XML Schema 文書に準拠します。このファイルは手動で構成しないことをお薦めします。このファイルを作成および構成するには、TopLink Workbench を使用します（『Oracle TopLink 開発者ガイド』の TopLink Workbench の理解に関する項を参照）。

ejb3-toplink-sessions.xml ファイルとは

ejb3-toplink-sessions.xml ファイルは、TopLink JPA プレビュー永続性構成ディスクリプタ・ファイルであり、TopLink JPA プレビュー永続性プロバイダとともに使用されている場合は、データソース、ログイン情報、キャッシュ・オプション、ロギングなどの TopLink セッションレベル・オプション（『Oracle TopLink 開発者ガイド』のサーバー・セッションの構成に関する項を参照）を記述します。TopLink ユーザーが使い慣れている sessions.xml ファイルと同等です。

注意： OC4J では、TopLink Essentials JPA 永続性プロバイダがデフォルトで使用されます。この場合、TopLink JPA 拡張を使用して、TopLink セッションレベルのオプションを構成できます（3-5 ページの「TopLink Essentials JPA 永続性を使用した TopLink API への実行時アクセス」を参照）。

このファイルは、使用されている場合はプライマリ・プロジェクトへの参照を提供します（2-8 ページの「toplink-ejb-jar.xml ファイルとは」を参照）。

詳細は、26-4 ページの「ejb3-toplink-sessions.xml ファイルの構成」を参照してください。

EJB 3.0

デフォルトの TopLink Essentials JPA 永続性プロバイダとともに EJB 3.0 を使用している場合、このファイルは使用されません。

EJB 3.0 を使用している場合、ejb3-toplink-sessions.xml ファイルは TopLink JPA プレビュー永続性プロバイダ構成のカスタマイズにのみ使用されます（3-4 ページの「JPA 永続性プロバイダのカスタマイズ」を参照）。このファイルを使用して TopLink JPA プレビュー永続性プロバイダをカスタマイズする場合は、toplink-ejb-jar.xml ファイルも使用できます（2-8 ページの「toplink-ejb-jar.xml ファイルとは」を参照）。

EJB 2.1

EJB 2.1 を使用している場合、ejb3-toplink-sessions.xml ファイルは使用されません。

XML 参照

ejb3-toplink-sessions.xml ファイルは、<*OC4J_HOME*>¥toplink¥config¥xsds にある XML Schema 文書に準拠します。このファイルは手動で構成しないことをお薦めします。このファイルを作成および構成するには、TopLink Workbench を使用します（『Oracle TopLink 開発者ガイド』の TopLink Workbench の理解に関する項を参照）。

persistence.xml ファイルとは

persistence.xml ファイルは、エンティティを使用する EJB 3.0 アプリケーションで 1 つ以上の永続性ユニットを定義するために使用する永続性ディスクリプタ・ファイルです。

このリリースでは、EJB JAR、WAR または EAR で persistence.xml を定義できます。

永続性ユニットでは、エンティティ・マネージャの構成を定義します。エンティティ・マネージャを取得するときに、永続性ユニットを名前で指定します（29-9 ページの「EntityManager の取得」を参照）。または、OC4J のデフォルト永続性ユニットを利用できます（2-10 ページの「OC4J の永続性ユニットのデフォルトについて」を参照）。

永続性ユニットは、次のものを含む論理的なグループです。

- エンティティ・マネージャ : エンティティ・マネージャ・プロバイダ、そこから取得されたエンティティ・マネージャおよびエンティティ・マネージャ構成を含みます。
- データソース（26-6 ページの「永続性ユニットでのデータソースの指定」を参照）。
- ベンダー拡張（26-6 ページの「永続性ユニットでのベンダー拡張の構成」を参照）。
- 永続管理クラス : エンティティ・マネージャを使用して管理するクラス、つまり、エンティティ・クラス、埋込み可能クラスおよびマッピングされたスーパークラス（26-5 ページの「この永続性ユニットに含まれる永続管理クラス」を参照）。

  特定の永続性ユニットのすべての永続管理クラスは、単一のデータベースに対するマッピングにまとめて配置する必要があります。
- マッピング・メタデータ : データベース表に永続管理クラスをマッピングする方法を説明する情報。マッピング・メタデータは、永続管理クラスに対するアノテーションおよび orm.xml ファイル（2-11 ページの「orm.xml ファイルとは」を参照）を使用して指定できます。

詳細は、次を参照してください。

- 26-4 ページの「persistence.xml ファイルの構成」
- 27-2 ページの「JPA エンティティ・アプリケーションのパッケージ化」
- EJB 3.0 仕様

OC4J の永続性ユニットのデフォルトについて

永続性ユニットの構成を単純化するために、次の OC4J 機能を使用できます。

- スマート・デフォルト設定
- デフォルトの永続性ユニット名によるエンティティ・マネージャの取得

詳細は、次を参照してください。

- 26-6 ページの「OC4J のデフォルト永続性ユニットの persistence.xml ファイルの構成」
- 29-9 ページの「EntityManager の取得」

スマート・デフォルト設定

EJB モジュール専用として、次の場合に OC4J にデフォルトの persistence.xml ファイルの構築を任せ、適切なデフォルト値でそのファイルを構成し、デフォルト名でデフォルトの永続性ユニットを定義できます。

- persistence.xml なしでアプリケーションをデプロイし、アプリケーションに @Entity アノテーション付きのクラスが少なくとも 1 つ含まれる場合
- 空の persistence.xml とともにアプリケーションをデプロイする場合

デフォルトの永続性ユニット名によるエンティティ・マネージャの取得

アプリケーションで（明示的に、またはスマート・デフォルト設定を通じて）永続性ユニットを 1 つのみ指定する場合、エンティティ・マネージャの取得時に永続性ユニット名を指定する必要はありません。この場合、OC4J によりデフォルトの永続性ユニット名が指定されます。

この機能を無効にするには、orion-ejb-jar.xml ファイルの属性 disable-default-persistent-unit を true に設定します。

この機能を無効にしても、persistence.xml ファイルで空の永続性ユニットを指定する場合は OC4J のデフォルト永続性ユニットを使用でき、その永続性ユニットの有効範囲内のエンティティ・マネージャを取得するときに、永続性ユニットを名前で指定する必要はありません。この場合、OC4J は、固有のデフォルト永続性ユニットを使用し、永続性ユニット・ルート内のすべての JPA エンティティ・クラスがその永続性ユニットに属していることを前提とします。このような空の永続性ユニットは、アプリケーションに 1 つのみ指定できます。

EJB 3.0

EJB 3.0 エンティティを使用している場合、（OC4J のデフォルト永続性ユニットを使用していないかぎり）persistence.xml ファイルは必須です。

EJB 2.1

EJB 2.1 を使用している場合、persistence.xml ファイルは使用されません。

XML 参照

EJB 3.0 の場合、このデプロイメント・ディスクリプタ・ファイルは http://java.sun.com/products/ejb/docs.html にある EJB 3.0 仕様で定義されている XML Schema 文書に準拠します。

orm.xml ファイルとは

orm.xml ファイルは、オブジェクト・リレーショナル・マッピング構成の指定に使用する XML デプロイメント・ディスクリプタです。orm.xml ファイルは、アノテーションのかわりに、またアノテーションに優先して使用できます。

複数の orm.xml ファイルを指定でき、これらのファイルはクラスパスの任意の場所に存在できます。

詳細は、次を参照してください。

- 2-10 ページの「persistence.xml ファイルとは」
- 27-2 ページの「JPA エンティティ・アプリケーションのパッケージ化」

EJB 3.0

EJB 3.0 エンティティを使用している場合、orm.xml ファイルはオプションです。

EJB 2.1

EJB 2.1 を使用している場合、orm.xml ファイルは使用されません。

XML 参照

EJB 3.0 の場合、このデプロイメント・ディスクリプタ・ファイルは http://java.sun.com/products/ejb/docs.html にある EJB 3.0 仕様で定義されている XML Schema 文書に準拠します。

アプリケーションでの Enterprise Bean の使用方法

一般に、Enterprise Bean はクライアントから使用します（2-12 ページの「クライアント・アクセスについて」を参照）。

Enterprise Bean を使用して、メソッド起動フローのファイングレインな制御を実装することもできます（2-12 ページの「EJB 3.0 インターセプタについて」を参照）。

Web サービス・クライアントまたは Web サービス・エンドポイントとして Web サービスとともに Enterprise Bean を使用することもできます（2-14 ページの「EJB および Web サービスについて」を参照）。

デプロイされた EJB アプリケーションでは、Java EE アプリケーションのコンポーネントの性質を利用して、EJB のパフォーマンスおよびリソース使用率を監視および制御できます（2-14 ページの「EJB 管理について」を参照）。

クライアント・アクセスについて

一般には、Enterprise Bean をクライアントから使用して（29-2 ページの「使用しているクライアントのタイプ」を参照）、セッション、永続性またはメッセージ処理の管理などのアプリケーション・タスクを実行します。詳細は、第 29 章「クライアントからの Enterprise Bean へのアクセス」を参照してください。

EJB 3.0 インターセプタについて

インターセプタは、EJB 3.0 セッション Bean のビジネス・メソッドまたはメッセージドリブン Bean のメッセージ・リスナー・メソッドに関連付けるメソッドです。クライアントがこのようなメソッドを起動すると、OC4J は、クライアント起動の続行を許可する前にクライアントの起動をインターセプトし、インターセプタ・メソッドを起動します。

インターセプタ・メソッドおよびインターセプタ・ライフ・サイクル・コールバック・メソッドは、Bean クラスまたは Bean に関連付けられた個別のインターセプタ・クラスで定義できます。

ライフ・サイクル・コールバック以外のインターセプタは、各 Bean に 1 つのみ定義できます。ビジネス・メソッドが起動されるたびに、OC4J では最初に `AroundInvoke` メソッドが起動されます。ライフ・サイクル・コールバック・インターセプタ・メソッドは、対応するライフ・サイクル・イベントが発生した場合にのみ起動されます。

個別のインターセプタ・クラスに定義されるインターセプタ・メソッドは、起動コンテキストを引数として使用します。コンテキストを使用することで、インターセプタ・メソッド実装では、元のセッション Bean のビジネス・メソッドまたはメッセージドリブン Bean のメッセージ・リスナー・メソッド起動の詳細にアクセスできます。

この項の内容は次のとおりです。

- インターセプタの制限
- シングルトン・インターセプタ

詳細は、次を参照してください。

- 5-5 ページの「EJB 3.0 セッション Bean のライフ・サイクル・コールバック・インターセプタ・メソッドの構成」
- 5-6 ページの「EJB 3.0 セッション Bean のインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドの構成」
- 5-8 ページの「EJB 3.0 セッション Bean の AroundInvoke インターセプタ・メソッドの構成」
- 5-10 ページの「EJB 3.0 セッション Bean のインターセプタ・クラスの構成」
- 10-12 ページの「EJB 3.0 MDB のライフ・サイクル・コールバック・インターセプタ・メソッドの構成」
- 10-13 ページの「EJB 3.0 MDB のインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドの構成」
- 10-15 ページの「EJB 3.0 MDB の AroundInvoke インターセプタ・メソッドの構成」
- 10-17 ページの「EJB 3.0 MDB のインターセプタ・クラスの構成」
- EJB 3.0 仕様

インターセプタの制限

インターセプタは、セッション Bean（ステートレスおよびステートフル）とメッセージドリブン Bean で使用できます。

OC4J は、インターセプタを Bean のすべてのビジネス・メソッドに適用します。

複数のインターセプタ（1 つのインターセプタ・メソッドと、1 つの独自のインターセプタ・メソッドをそれぞれ含む 1 つ以上のインターセプタ・クラス）が存在する場合、クライアントがビジネス・メソッドを起動するたびに、OC4J は、クライアントによる起動の続行を許可する前にインターセプタ・クラスを定義順に起動し、次にインターセプタ・メソッドを起動します。

インターセプタ・メソッドは、ビジネス・メソッドになることができません。

インターセプタ・メソッドは、次のシグネチャを持つ必要があります。

```
Object <METHOD>(InvocationContext) throws Exception
```

インターセプタ・メソッドには、public、private、protected または package レベルのアクセスを割り当てることができますが、`final` または `static` として宣言することはできません。

インターセプタ内では、`InvocationContext` を使用してクライアント起動メタデータにアクセスできます。

インターセプタ・メソッドの起動は、起動されるビジネス・メソッドと同じトランザクションおよびセキュリティ・コンテキスト内で行われます。

インターセプタ・メソッドは、ランタイム例外をスローするか、次のように `EJBContext` オブジェクトを使用して `setRollbackOnly` をコールすることにより、トランザクションにロールバックのマークを付けることができます。

```
InvocationContext.getEJBContext().setRollbackOnly();
```

インターセプタは、`InvocationContext.proceed()` のコールの前後にこのロールバックを引き起こすことがあります。

詳細は、21-13 ページの「ロールバック計画の使用方法」を参照してください。

コンテナ管理のトランザクション（2-21 ページの「コンテナ管理のトランザクションとは」を参照）を使用する場合、インターセプタでは、コンテナのトランザクション境界の設定に干渉するリソース・マネージャ固有のトランザクション管理メソッドを使用できません。たとえば、インターセプタは、`java.sql.Connection` インタフェースの `commit`、`setAutoCommit` および `rollback` メソッド、または `javax.jms.Session` インタフェースの `commit` および `rollback` メソッドを使用できません。インターセプタは、`javax.transaction.UserTransaction` インタフェースの取得または使用を試行しません。

シングルトン・インターセプタ

EJB 3.0 仕様に指定されているとおり、OC4J では、デフォルトで Bean インターセプタが作成されます。Bean インターセプタ・インスタンスのライフ・サイクルは、関連付けられている Bean インスタンスのライフ・サイクルと同じです。Bean インスタンスが作成されると、Bean に定義されたインターセプタ・クラスごとにインターセプタ・インスタンスが作成されます。これらのインターセプタ・インスタンスは、Bean インスタンスの削除時に破棄されます。このようにして、インターセプタに状態を格納できます。

インターセプタがステートレスの場合、OC4J による EJB 3.0 仕様の最適化拡張を使用して、シングルトン・インターセプタを指定できます。シングルトン・インターセプタを使用するようセッション Bean またはメッセージドリブン Bean を構成し、Bean をインターセプタ・クラスに関連付けると、OC4J により、すべての Bean インスタンスで共有できるインターセプタ・クラスの単一のインスタンスが作成されます。これにより、メモリー要件が低下し、ライフ・サイクルのオーバーヘッドが減少します。

詳細は、次を参照してください。

- 5-12 ページの「セッション Bean でのシングルトン・インターセプタの指定」
- 10-19 ページの「MDB でのシングルトン・インターセプタの指定」

EJB および Web サービスについて

ステートレス・セッション Bean は Web サービス・エンドポイントとして公開できます。任意の EJB タイプを Web サービスのクライアントにできます。

詳細は、第 30 章「EJB および Web サービスの使用方法」を参照してください。

EJB 管理について

Java EE アプリケーションのデプロイ後に、Java EE 管理機能を使用して実行時にアプリケーションを監視および最適化できます。

詳細は、次を参照してください。

- 第 31 章「EJB アプリケーションの管理」
- 第 32 章「EJB パフォーマンスの最適化」

EJB 永続性サービスについて

OC4J では、次の永続性 API がサポートされます。

- TopLink EJB 3.0 JPA 永続性プロバイダ（3-3 ページの「EJB 3.0 アプリケーションで OC4J が永続性を管理する方法」を参照）
- TopLink EJB 2.1 永続性マネージャ（3-14 ページの「EJB 2.1 アプリケーションで OC4J が永続性を管理する方法」を参照）
- Orion EJB 2.0 永続性マネージャ（非推奨：『Oracle Containers for J2EE Orion CMP 開発者ガイド』を参照）

OC4J は、定義するオブジェクト・リレーショナル・マッピングのタイプおよび特定のデプロイ XML ファイルが存在するかどうかに基づいて、使用する永続性のタイプを選択します。OC4J の選択は、デプロイしている EJB アプリケーションのタイプに応じて変化します。

- EJB 3.0 アプリケーション
- EJB 2.n アプリケーション

EJB 3.0 アプリケーション

`ejb-jar.xml` ファイルを使用せずに EJB 3.0 エンティティを `ejb.jar` にデプロイする場合、または OC4J で 1 つ以上の EJB 3.0 アノテーションが検出された場合、OC4J は TopLink EJB 3.0 JPA 永続性プロバイダを使用します。

詳細は、次を参照してください。

- 3-2 ページの「EJB 3.0 アプリケーションの定義方法」
- 3-4 ページの「JPA 永続性プロバイダのカスタマイズ」

EJB 2.*n* アプリケーション

EJB 2.1 および EJB 2.0 アプリケーションでは、OC4J は、アクションごとに表 2-4 にまとめられているアルゴリズムを使用します。たとえば、`toplink-ejb-jar.xml` ファイルを使用せずに CMP アプリケーションをデプロイする場合、OC4J は TopLink 永続性マネージャを使用し、デフォルトの TopLink オブジェクト・リレーショナル・マッピングを作成します。

表 2-4 OC4J EJB 2.n 永続性マネージャの選択

アクション	toplink-ejb-jar.xml	orion-ejb-jar.xml	永続性マネージャ	マッピング・タイプ
1. デプロイ。	なし	オプション。存在する場合、マッピングおよび `persistence-manager` 要素は含まれません。	Toplink	デフォルトの TopLink
1. デプロイ。	存在	オプション。存在する場合、マッピングおよび `persistence-manager` 要素は含まれません。	Toplink	`toplink-ejb-jar.xml` で定義されている TopLink（デフォルトの永続性マネージャ・プロパティ）
1. `orion-ejb-jar.xml` ファイルを編集して `persistence-manager` 要素の `name` 属性を `toplink` に設定[1]。 2. 追加の `persistence-manager` サブエントリを編集[1]。 3. デプロイ。	存在	オプション。存在する場合はマッピングが含まれません。	Toplink	`toplink-ejb-jar.xml` で定義されている TopLink（カスタム永続性マネージャ・プロパティ）
1. デプロイ。	なし	存在し、Orion マッピングを含みます。`persistence-manager` 要素はオプションです。	Orion	`orion-ejb-jar.xml` で定義されている Orion
1. `orion-ejb-jar.xml` ファイルを編集して `persistence-manager` 要素の `name` 属性を `orion` に設定[1]。 2. デプロイ。	なし	オプション。存在する場合はマッピングが含まれません。	Orion	デフォルトの Orion

[1] A-4 ページの「<persistence-manager>」を参照してください。

詳細は、次を参照してください。

- 3-13 ページの「EJB 2.1 モジュールの定義方法」
- 3-15 ページの「TopLink EJB 2.1 永続性マネージャのカスタマイズ」

EJB JNDI サービスについて

Java Naming and Directory Interface（JNDI）では、複数のネーミングおよびディレクトリ・サービスへの統一インタフェースが Java EE アプリケーションに提供されます。JNDI を使用して、分散 Java EE 環境でコンポーネントを編成および特定できます。Java EE コンポーネントおよび関連付けられている JNDI プロパティの環境参照を定義できます。

JNDI を使用すると、次のものを使用してこれらのコンポーネントをルックアップおよび取得できます。

- JNDI 初期コンテキスト
- EJB コンテキスト
- EJB 3.0 アノテーションおよびリソース・インジェクション

詳細は、次を参照してください。

- 第 19 章「JNDI サービスの構成」
- 1-8 ページの「アノテーションおよびリソース・インジェクションの動作」

EJB データ ソース・サービスについて

データソースは、OC4J がエンティティを維持する物理的なエンタープライズ情報システムを表す Java オブジェクトです。アプリケーションでは、データソース・オブジェクトを使用して、データソースが表すエンタープライズ情報システムへの接続を取得します。

この項の内容は次のとおりです。

- OC4J でサポートされるデータソースのタイプ
- OC4J での接続 URL の定義方法
- データソースでサポートされるトランザクションのタイプ
- OC4J でデータソース情報を構成する場所
- デフォルトのデータソース
- OC4J で複数のデータソースを処理する方法

詳細は、次を参照してください。

- 第 20 章「データソースの構成」
- 『Oracle Containers for J2EE サービス・ガイド』の「データソース」

OC4J でサポートされるデータソースのタイプ

OC4J では、次のタイプのデータソースがサポートされます。

- マネージド・データソース
- ネイティブ・データソース

表 2-5 に、これらの OC4J データソースの特性をリストします。

表 2-5 OC4J データソース・タイプの特性

特性	マネージド	ネイティブ
OC4J 接続プールを使用するか	○	×
接続がグローバル・トランザクションに参加できるか	○	×
接続が OC4J `Connection` プロキシにラップされるか	○	×

マネージド・データソース

マネージド・データソース（例 2-2 を参照）は、JDBC ドライバまたはデータソースのラッパーとして機能する java.sql.DataSource インタフェースの OC4J 提供の実装です。マネージド・データソースを個別の接続プールに関連付けることができます。複数のマネージド・データソースで同じ接続プールを共有できます。

例 2-2 マネージド・データソース

```
<connection-pool name="ScottConnectionPool">
    <connection-factory
        factory-class="oracle.jdbc.pool.OracleDataSource"
        user="scott"
        password="tiger"
        url="jdbc:oracle:thin:@//localhost:1521/ORCL" >
    </connection-factory>
</connection-pool>

<managed-data-source
    name="OracleManagedDS"
    jndi-name="jdbc/OracleDS"
    connection-pool-name="ScottConnectionPool"
/>
```

詳細は、20-2 ページの「Oracle データベースのデータソースの構成」を参照してください。

ネイティブ・データソース

ネイティブ・データソース（例 2-3 を参照）は、java.sql.DataSource インタフェースの JDBC ベンダー提供の実装です。選択するデータソース・インスタンスで提供される接続プールを使用します。各ネイティブ・データソースでは独自の接続プールを使用する必要があります。

例 2-3 ネイティブ・データソース

```
<native-data-source
    name="nativeDataSource"
    jndi-name="jdbc/nativeDS"
    description="Native DataSource"
    data-source-class="com.ddtek.jdbcx.sqlserver.SQLServerDataSource"
    user="frank"
    password="frankpw"
    url="jdbc:datadirect:sqlserver://server_name:1433;User=usr;Password=pwd">
</native-data-source>
```

詳細は、20-3 ページの「サード・パーティ・データベースのデータソースの構成」を参照してください。

OC4J での接続 URL の定義方法

接続 URL を指定して、OC4J に基礎となる物理データソースの検索場所を指示します。

マネージド・データソース（2-18 ページの「マネージド・データソース」を参照）を定義する場合、接続 URL は関連付ける接続プールの属性です（例 2-2 を参照）。

ネイティブ・データソース（2-18 ページの「ネイティブ・データソース」を参照）を定義する場合、接続 URL はネイティブ・データソースの属性です（例 2-3 を参照）。

Oracle データベースへの接続 URL を指定する場合は、サービスベースの URL を使用する必要があります。つまり、例 2-4 に示すように（`host:port:SID` ではなく）`host:port/SID` の形式の URL です。

例 2-4 OC4J サービスベースの接続 URL

```
url="jdbc:oracle:thin:@//localhost:1521/ORCL"
```

Oracle 以外のデータベースへの接続 URL を指定する場合は、そのシステムに適した URL を使用します。例 2-5 に、SQLServer データベースの典型的な接続 URL を示します。

例 2-5 Oracle 以外の接続 URL

```
url="jdbc:datadirect:sqlserver://server_name:1433;User=usr;Password=pwd"
```

データソースでサポートされるトランザクションのタイプ

マネージド・データソースでは、ローカル・トランザクションとグローバル（2 フェーズ・コミット）トランザクションの両方がサポートされます。デフォルトでは、これらはグローバル・トランザクションをサポートするように構成されます。詳細は、20-2 ページの「Oracle データベースのデータソースの構成」を参照してください。

ネイティブ・データソースではローカル・トランザクションのみサポートされます。

OC4J でデータ ソース情報を構成する場所

OC4J では、`data-sources.xml` ファイルにデータソース情報を構成します。

EAR には `data-sources.xml` ファイルを含めることができますが、OC4J では複数の `data-sources.xml` ファイルはサポートされません。

EJB 3.0 アプリケーションでは、データソースを永続性ユニットに関連付けます（26-6 ページの「永続性ユニットでのデータソースの指定」を参照）。

詳細は、次を参照してください。

- 2-20 ページの「OC4J で複数のデータソースを処理する方法」
- 2-19 ページの「デフォルトのデータソース」

デフォルトのデータ ソース

アプリケーション構成を単純化するために、デフォルトのデータソースを定義できます。

デフォルトのデータソースの定義方法は、デフォルトのデータソースにアクセスするアプリケーションのタイプによって決まります。

異なるタイプのアプリケーションのデータソースを構成する方法の詳細は、次を参照してください。

- 20-4 ページの「EJB 3.0 アプリケーションのデフォルトのデータソースの構成」
- 20-4 ページの「EJB 2.1 アプリケーションのデフォルトのデータソースの構成」

OC4J で複数のデータソースを処理する方法

OC4J では、orion-ejb-jar.xml ファイルの異なるエンティティ内にある複数のデータソースはサポートされません。

アプリケーションが複数の EAR から構成され、各 EAR に data-sources.xml が含まれている場合、アプリケーションのデプロイ時に、OC4J は最後のエンティティ Bean の data-source.xml ファイルをすべてのエンティティ Bean に対して使用します。

このシナリオに対処するには、データソースを orion-application.xml ファイルで指定するか、デフォルトのデータソースを指定します。

詳細は、次を参照してください。

- 2-5 ページの「OC4J が EJB モジュールをデプロイする順序」
- 2-19 ページの「デフォルトのデータソース」

EJB トランザクション・サービスについて

OC4J で、Java Transaction Service（JTS）でサポートされる Java Transaction API（JTA）を使用してトランザクションを管理できます。アノテーションまたはデプロイメント・ディスクリプタを使用して、設計またはデプロイ時に Enterprise Bean のトランザクション・プロパティを定義し、OC4J でトランザクション管理を引き継ぎます。

注意： フラットなトランザクションのみサポートされます。ネストされたトランザクションはサポートされません。

この項の内容は次のとおりです。

- トランザクションの管理担当
- クライアントがビジネス・メソッドを起動する際のトランザクションの処理方法
- グローバル・トランザクションまたは 2 フェーズ・コミット（2PC）トランザクションへの参加方法

詳細は、次を参照してください。

- 第 21 章「トランザクション・サービスの構成」
- 21-12 ページの「トランザクションのベスト・プラクティス」
- 『Oracle Containers for J2EE サービス・ガイド』の「OC4J トランザクション・サポート」

トランザクションの管理担当

トランザクションは、コンテナ（2-21 ページの「コンテナ管理のトランザクションとは」を参照）または Bean（2-22 ページの「Bean 管理のトランザクションとは」を参照）で管理できます。

コンテナ管理のトランザクション管理がデフォルトです。

Enterprise Bean でトランザクション管理を構成する場合、次の制限を考慮してください。

- EJB 3.0 エンティティは、トランザクション管理タイプで構成できません。EJB 3.0 エンティティは、コール元のトランザクション・コンテキスト内で実行されます。
- EJB 2.1 エンティティ Bean は、常にコンテナ管理のトランザクション境界を使用する必要があります。EJB 2.1 エンティティ Bean は、Bean 管理のトランザクション境界で指定できません。

他のすべての EJB タイプでは、コンテナ管理または Bean 管理のトランザクション管理を選択できます。

詳細は、次を参照してください。

- 21-2 ページの「EJB 3.0 トランザクション管理の構成」
- 21-5 ページの「EJB 2.1 トランザクション管理の構成」

コンテナ管理のトランザクションとは

コンテナ管理のトランザクション（CMT）を使用する場合、EJB は、トランザクションが開始したことを確認して適宜コミットする責任をコンテナに委任します。

すべてのセッション Bean とメッセージドリブン Bean で CMT を使用できます。

EJB 2.1 エンティティ Bean では CMT を使用する必要があります。

EJB 3.0 エンティティは、トランザクション管理タイプで構成できません。EJB 3.0 エンティティは、コール元のトランザクション・コンテキスト内で実行されます。

CMT を使用する Enterprise Bean の開発時には、次の点を考慮します。

- `java.sqlConnection` のメソッド `commit`、`setAutoCommit`、`rollback`、または `javax.jms.Session` のメソッド `commit`、`rollback` など、リソース・マネージャ固有のトランザクション管理メソッドは使用しません。
- `javax.transaction.UserTransaction` インタフェースは取得または使用しません。
- CMT を使用しているステートフル・セッション Bean では、`javax.ejb.SessionSynchronization` インタフェースを実装できます。
- CMT を使用する Enterprise Bean では、`javax.ejb.EJBContext` のメソッド `setRollbackOnly` および `getRollbackOnly` を使用できます。

CMT を使用する EJB では、各ビジネス・メソッドについて、クライアントがメソッドを起動したときにコンテナがトランザクションを管理する方法を決定するトランザクション属性も指定できます（2-23 ページの「クライアントがビジネス・メソッドを起動する際のトランザクションの処理方法」を参照）。

Bean 管理のトランザクションとは

Bean 管理のトランザクション（BMT）を使用する場合は、トランザクションが開始したことを確認して適宜コミットする責任を Bean プロバイダが持ちます。

セッション Bean とメッセージドリブン Bean のみが BMT を使用できます。

BMT を使用する EJB の開発時には、次の点を考慮します。

- `javax.transaction.UserTransaction` のメソッド `begin` および `commit` を使用してトランザクションの境界を設定します。
- ステートフル・セッション Bean インスタンスは、ビジネス・メソッドが返る前に開始したトランザクションをコミットできますが、このコミットは必須ではありません。

  トランザクションがビジネス・メソッドの終わりまでに完了しなかった場合、最終的にインスタンスがトランザクションを完了するまで、コンテナは複数のクライアント・コールにわたってトランザクションとインスタンス間の関連付けを保持します。
- ステートレス・セッション Bean インスタンスは、ビジネス・メソッドまたはタイムアウト・コールバック・メソッドが返る前に、開始されたすべてのトランザクションをコミットする必要があります。
- メッセージドリブン Bean インスタンスは、メッセージ・リスナー・メソッドまたはタイムアウト・コールバック・メソッドが返る前にトランザクションをコミットする必要があります。
- トランザクションの開始後は、`java.sqlConnection` のメソッド `commit`、`setAutoCommit`、`rollback` や、`javax.jms.Session` のメソッド `commit`、`rollback` など、リソース・マネージャ固有のトランザクション管理メソッドを使用しないでください。
- BMT を使用する Bean では、`EJBContext` のメソッド `getRollbackOnly` および `setRollbackOnly` を使用しないでください。`UserTransaction` のメソッド `getStatus` および `rollback` をかわりに使用する必要があります。

クライアントがビジネス・メソッドを起動する際のトランザクションの処理方法

CMT（2-21 ページの「コンテナ管理のトランザクションとは」を参照）を使用する Enterprise Bean では、クライアントが Bean メソッドを起動したときにコンテナがトランザクションを管理する方法を決定するトランザクション属性を指定できます。

次のタイプの Bean メソッドごとにトランザクション属性を指定できます。

- Bean のビジネス・インタフェースのメソッド
- メッセージドリブン Bean のメッセージ・リスナー・メソッド
- タイムアウト・コールバック・メソッド
- ステートレス・セッション Bean の Web サービス・エンドポイント・メソッド
- EJB 2.1 およびそれ以前では、セッションまたはエンティティ Bean のホームまたはコンポーネント・インタフェースのメソッド

表 2-6 に、トランザクション属性の構成方法およびクライアント管理のトランザクションがメソッドの起動時に存在するかどうかに応じて EJB メソッド起動で使用されるトランザクション（存在する場合）を示します。

OC4J はコンテナ管理のトランザクションを暗黙的に開始し、クライアント管理のトランザクションがない状態で Bean メソッドが起動されたときのトランザクション属性構成を満足させます。

表 2-6 トランザクション属性ごとの EJB トランザクション・サポート

トランザクション属性	クライアント管理のトランザクションが存在	クライアント管理のトランザクションが存在しない
Not Supported	コンテナはクライアント・トランザクションを一時停止します。	トランザクションを使用しません。
Supports	クライアント管理のトランザクションを使用します。	トランザクションを使用しません。
Required[1]	クライアント管理のトランザクションを使用します。	コンテナは新規トランザクションを開始します。
Requires New	コンテナはクライアント・トランザクションを一時停止して新規トランザクションを開始します。	コンテナは新規トランザクションを開始します。
Mandatory	クライアント管理のトランザクションを使用します。	例外が発生します。
Never	例外が発生します。	トランザクションを使用しません。

[1] （メモリー内またはファイルベースの）OEMS JMS メッセージ・サービス・プロバイダを使用するメッセージドリブン Bean では、Oracle JMS コネクタを使用してこのメッセージ・サービス・プロバイダにアクセスする場合にのみ Required がサポートされます。詳細は、2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」を参照してください。

「トランザクションを使用しません」と示されている条件では、エンティティ Bean を変更しないことをお薦めします。また、`Supports` トランザクション属性を使用すると、クライアントがトランザクションを明示的に提供しない場合に非トランザクション状態になるため、このトランザクション属性の使用は避けることもお薦めします。

TopLink CMP を使用する場合、トランザクションは EJB 2.X CMP エンティティ Bean を変更するために存在する必要があります。トランザクションが存在しない場合、TopLink 永続性マネージャによって Bean の読取り専用コピーが返されます。

詳細は、次を参照してください。

- 21-3 ページの「EJB 3.0 トランザクション属性の構成」
- 21-6 ページの「EJB 2.1 トランザクション属性の構成」
- 3-14 ページの「EJB 2.1 アプリケーションで OC4J が永続性を管理する方法」

グローバル・トランザクションまたは 2 フェーズ・コミット（2PC）トランザクションへの参加方法

トランザクションに関与しているすべてのリソースが XA に準拠している場合、OC4J はグローバルまたは 2 フェーズ・コミット・トランザクションを自動的に調整します。

このリリースの OC4J では、推奨されないデータベース内調整に置き換わるトランザクション調整機能が OC4J にあります。また、中間層コーディネータは異種対応であるため、Oracle のみでなくすべての XA 互換リソースがサポートされます。

中間層コーディネータでは、次の機能がサポートされます。

- 任意の XA 準拠リソース
- 相互位置付けおよびトランザクション・インフロー
- 最後のリソース・コミットの最適化
- リカバリ・ロギング

詳細は、次を参照してください。

- 2-34 ページの「2 フェーズ・コミット（2PC）トランザクション用のメッセージ・サービスの構成」
- 『Oracle Containers for J2EE サービス・ガイド』の「中間層 2 フェーズ・コミット（2PC）コーディネータ」

EJB セキュリティ・サービスについて

OC4J が提供する次のような Java EE セキュリティ・サービスを使用するように EJB を構成できます。

- Java 2 セキュリティ・モデル
- Java Authentication and Authorization Service（JAAS）

詳細は、次を参照してください。

- 第 22 章「セキュリティ・サービスの構成」
- 『Oracle Containers for J2EE セキュリティ・ガイド』の「標準セキュリティの概要」

メッセージ・サービスについて

メッセージ・サービス・プロバイダは、クライアントがメッセージを送信できる送信先およびメッセージドリブン Bean が処理するメッセージを受信できる受信先を提供します。

OC4J では、XA 対応の 2 フェーズ・コミット（2PC）トランザクションと XA 非対応のトランザクションの両方で様々なメッセージ・サービス・プロバイダがサポートされます。

メッセージ・サービス・プロバイダには、直接アクセスするか、Oracle JMS コネクタなどの J2EE Connector Architecture（J2CA）リソース・アダプタを通じてアクセスします。Oracle JMS コネクタの詳細は、2-26 ページの「Oracle JMS コネクタ : J2EE Connector Architecture（J2CA）ベース・プロバイダ」を参照してください。

注意 : メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」を参照してください。

詳細は、次を参照してください。

- 1-59 ページの「メッセージドリブン Bean とは」
- 2-26 ページの「MDB で使用できるメッセージ・サービス・プロバイダ」
- 2-31 ページの「メッセージ・サービス構成オプション : アノテーションまたは XML の選択と属性またはアクティブ化構成プロパティの選択」
- 2-34 ページの「2 フェーズ・コミット（2PC）トランザクション用のメッセージ・サービスの構成」
- 第 23 章「メッセージ・サービスの構成」
- 10-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」
- 10-4 ページの「直接メッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」
- 18-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」
- 18-4 ページの「直接メッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」

MDB で使用できるメッセージ・サービス・プロバイダ

OC4J では、MDB を次の Oracle Enterprise Messaging Service（OEMS）プロバイダと組み合せて使用できます。

- Oracle JMS コネクタ : J2EE Connector Architecture（J2CA）ベース・プロバイダ
- OEMS JMS: メモリー内またはファイルベース・プロバイダ
- OEMS JMS データベース : アドバンスト・キューイング（AQ）ベース・プロバイダ

> **注意 :** メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」を参照してください。

Oracle JMS コネクタ : J2EE Connector Architecture（J2CA）ベース・プロバイダ

Oracle JMS コネクタは、J2CA 1.5 準拠のリソース・アダプタです。この機能により、OC4J 管理アプリケーションに対して、JMS 1.1 または 1.02b を実装する任意の JMS プロバイダにアクセスする統一メカニズムが提供されます。Oracle JMS コネクタを使用すると、図 2-1 に示すように、OC4J を Oracle および Oracle 以外の様々な JMS プロバイダと統合できます。

OC4J の観点からは、J2CA は、メッセージドリブン Bean で使用するためにメッセージ・サービス・プロバイダにアクセスする手段としてのみ使用されます。

Oracle JMS コネクタでは、2 フェーズ・コミット（2PC）トランザクション用の XA ファクトリと、2PC を必要としないトランザクション用の非 XA ファクトリの両方がサポートされます。

図 2-1 MDB と J2CA JMS 宛先との対話例

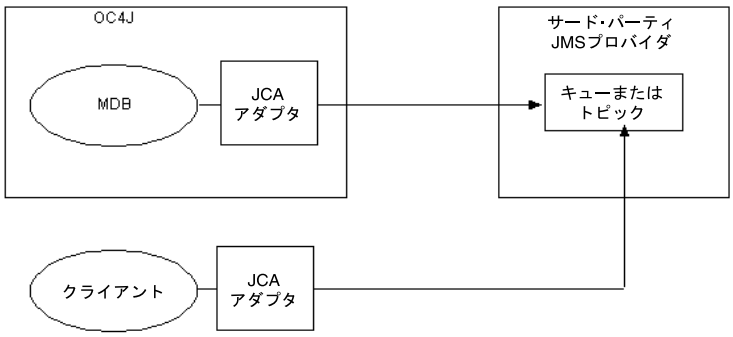


表 2-7 に、Oracle JMS コネクタでサポートされる JMS メッセージ・サービス・プロバイダをまとめます。

表 2-7 Oracle JMS コネクタでの JMS メッセージ・サービス・プロバイダのサポート

JMS プロバイダ	バージョン
OEMS JMS: メモリー内またはファイルベース・プロバイダ	すべて
OEMS JMS データベース : アドバンスト・キューイング（AQ）ベース・プロバイダ	すべて
IBM WebSphere MQ ベースの JMS	サーバー・バージョン 5.3 および 6.0
TIBCO Enterprise for JMS	3.1.0
SonicMQ	6.0

注意： メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」を参照してください。

詳細は、次を参照してください。

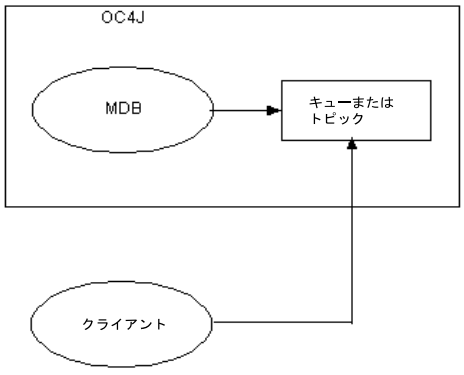
- 23-2 ページの「メッセージ・サービス・プロバイダで使用するための J2CA リソース・アダプタの構成」
- 23-3 ページの「OC4J J2CA リソース・アダプタのデプロイ XML ファイルの構成」
- 2-24 ページの「グローバル・トランザクションまたは 2 フェーズ・コミット（2PC）トランザクションへの参加方法」
- 10-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」
- 18-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」
- 『Oracle Containers for J2EE リソース・アダプタ管理者ガイド』の Oracle JMS サポートおよび汎用 JMS リソース・アダプタの概要に関する項
- 『Oracle Containers for J2EE リソース・アダプタ管理者ガイド』のリソース・アダプタの管理の概要に関する項

注意： J2CA メッセージ・サービス・プロバイダ・リソース・アダプタおよび MDB アプリケーションの構成のコード例は、`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-gjra-with-oracleasjms/doc/how-to-gjra-with-oracleasjms.html` を参照してください。

OEMS JMS: メモリー内またはファイルベース・プロバイダ

OEMS JMS は、メモリー内またはファイルベースの永続性を提供するネイティブ Java JMS プロバイダ実装であり、OC4J と緊密に統合されています。これは、OC4J に含まれているデフォルトの JMS プロバイダです。図 2-2 に、OC4J 内に内部的に格納されている OEMS JMS のキューまたはトピックに、クライアントが非同期リクエストを直接送信する方法を示します。MDB は OEMS JMS からメッセージを直接受信します。

図 2-2 MDB と OEMS JMS 宛先との対話例



OEMS JMS には、直接アクセスするか、Oracle JMS コネクタ（2-26 ページの「Oracle JMS コネクタ : J2EE Connector Architecture（J2CA）ベース・プロバイダ」を参照）を通じてアクセスできます。

注意： メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」を参照してください。

OEMS JMS では、2 フェーズ・コミット（2PC）トランザクション用の XA ファクトリと、2PC を必要としないトランザクション用の非 XA ファクトリの両方がサポートされます。2PC サポートの詳細は、2-24 ページの「グローバル・トランザクションまたは 2 フェーズ・コミット（2PC）トランザクションへの参加方法」を参照してください。XA ファクトリをサポートするよう OEMS JMS を構成する方法の詳細は、23-5 ページの「jms.xml の構成」を参照してください。

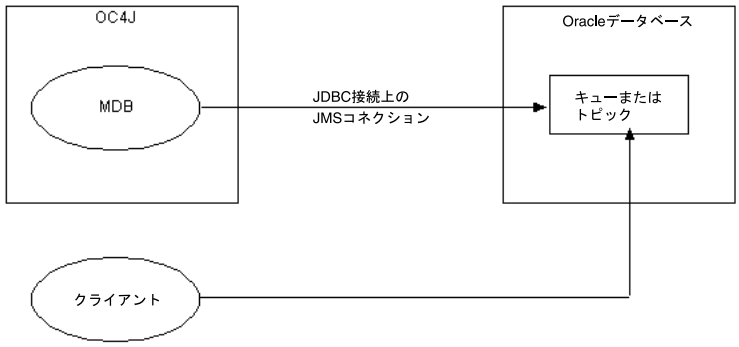
詳細は、次を参照してください。

- 23-4 ページの「OEMS JMS メッセージ・サービス・プロバイダの構成」
- 10-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」
- 10-4 ページの「直接メッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」
- 18-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」
- 18-4 ページの「直接メッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」
- 『Oracle Containers for J2EE サービス・ガイド』の「Java Message Service（JMS）」

OEMS JMS データベース：アドバンスト・キューイング（AQ）ベース・プロバイダ

OEMS JMS データベースは、Oracle Database Streams アドバンスト・キューイング（AQ）機能への JMS インタフェースです。Oracle AQ は、Oracle データベースの統合メッセージ・キューイング機能であり、Oracle データベース内でインストールおよび構成する Oracle Streams 情報統合インフラストラクチャに構築されています（図 2-3 を参照）。

図 2-3 MDB と OEMS JMS データベース宛先との対話例



MDB は OEMS JMS データベースを次のように使用します。

1. MDB は、ユーザー名とパスワードを持つデータソースを使用して、JMS コネクションをデータベースに対してオープンします。このデータソースは Oracle JMS プロバイダを表し、JDBC ドライバを使用して JMS コネクションを提供します。
2. MDB は、JMS コネクション上で JMS セッションをオープンします。
3. MDB のメッセージは、MDB の `onMessage` メソッドにルーティングされます。

クライアントはいつでも、MDB がリスニングしている Oracle JMS のトピックまたはキューにメッセージを送信できます。Oracle JMS のトピックまたはキューは、データベースにあります。

Oracle JMS を使用する前に、データベースに適切なキューまたは表を作成しておく必要があります。

注意： MDB は特定のバージョンの Oracle データベースでのみ動作します。詳細は、『Oracle Containers for J2EE サービス・ガイド』の JMS に関する項に記載されている動作保証のマトリックスを参照してください。

OEMS JMS データベースには、直接アクセスするか、Oracle JMS コネクタ（2-26 ページの「Oracle JMS コネクタ : J2EE Connector Architecture（J2CA）ベース・プロバイダ」を参照）を通じてアクセスできます。

注意： メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」を参照してください。

OEMS JMS データベースでは、2 フェーズ・コミット（2PC）トランザクション用の XA ファクトリと、2PC を必要としないトランザクション用の非 XA ファクトリの両方がサポートされます。2PC サポートの詳細は、2-24 ページの「グローバル・トランザクションまたは 2 フェーズ・コミット（2PC）トランザクションへの参加方法」を参照してください。XA ファクトリをサポートするよう OEMS JMS を構成する方法の詳細は、23-7 ページの「OEMS JMS データベース・プロバイダのインストールと構成」に記載されている手順 2 を参照してください。

詳細は、次を参照してください。

- 23-6 ページの「OEMS JMS データベース・メッセージ・サービス・プロバイダの構成」
- 10-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」
- 10-4 ページの「直接メッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」
- 18-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」
- 18-4 ページの「直接メッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」
- 『Oracle Streams アドバンスト・キューイング・ユーザーズ・ガイドおよびリファレンス』
- 『Oracle Containers for J2EE サービス・ガイド』の「Java Message Service（JMS）」

J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限

メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします（2-26 ページの「Oracle JMS コネクタ : J2EE Connector Architecture（J2CA）ベース・プロバイダ」を参照）。

J2CA リソース・アダプタを使用しない場合、次の制限を考慮してください。

- MDB でコンテナ管理のトランザクションを使用し、トランザクション属性を `Required` または `RequiresNew` に設定する場合、Oracle JMS コネクタを使用する必要があります。
- グローバルな 2 フェーズ・コミット（2PC）トランザクションに登録するには、JMS プロデューサとコンシューマの両方を J2CA ベースの XA コネクション・ファクトリから取得する必要があります。

  JMS プロデューサまたはコンシューマが J2CA ベースの XA コネクション・ファクトリから導出されない場合、グローバル・トランザクションが存在してもそのトランザクションに登録されません。この場合、JMS は、グローバル・トランザクションの有効範囲外であるかのように動作します。これは、次のようにトランザクション・パラメータを `true` に設定して（J2CA 以外の）JMS セッションを作成する場合でも同様です。

  ```
  QueueConnection.createQueueSession(true,1)
  ```

  この場合、グローバル・トランザクションとは無関係に `Session` のメソッド `commit` または `rollback` を起動する必要があります。

  > **注意：** リリース 10.1.3.1 では、MDB で J2CA および XA ファクトリを使用している場合にのみ、OC4J によりデフォルトで MDB 接続が自動登録されます。詳細は、2-34 ページの「2 フェーズ・コミット（2PC）XA トランザクションへの MDB の自動登録」を参照してください。

- グローバル・トランザクションへの参加中に、J2CA ベースの XA コネクション・ファクトリから取得されていないプロデューサまたはコンシューマの JMS メッセージをプロデュースまたはコンシュームしようとすると、JMS 実行時例外が発生します。
- コンテナ管理のトランザクション（CMT）および Bean 管理のトランザクション（BMT）と OEMS JMS プロバイダ（2-27 ページの「OEMS JMS: メモリー内またはファイルベース・プロバイダ」を参照）の組合せは、J2CA リソース・アダプタの使用を通じてのみサポートされます。これらの機能は、`oc4j.jms.pseudoTransactionEnlistment` 下位互換性フラグを使用する場合にはサポートされません（2-34 ページの「2 フェーズ・コミット（2PC）XA トランザクションへの MDB の自動登録」を参照）。

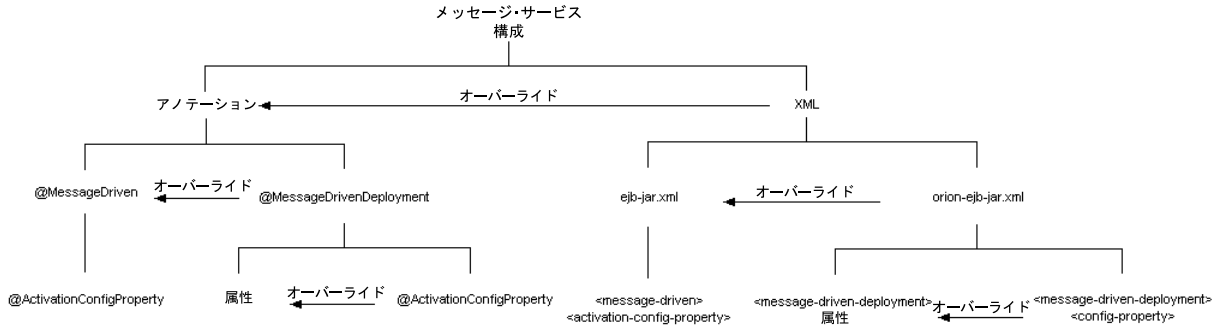
メッセージ・サービス構成オプション : アノテーションまたは XML の選択と属性またはアクティブ化構成プロパティの選択

図 2-4 に示すように、メッセージ・サービス・プロバイダ・オプションを構成する場合、次の 2 つの方法があります。

- アノテーションを使用したメッセージ・サービス構成
- XML を使用したメッセージ・サービス構成

原則として、アノテーションまたは XML のどちらを選択するかにかかわらず、属性ではなくアクティブ化構成プロパティを使用することをお薦めします。

図 2-4 メッセージ・サービス構成オプション



アノテーションを使用したメッセージ・サービス構成

アノテーションを使用する場合、`@MessageDrivenDeployment` または `@MessageDriven` アノテーションを使用してメッセージ・サービス・オプションを構成できます。`@MessageDrivenDeployment` 構成は、`@MessageDriven` 構成をオーバーライドします。

`@MessageDrivenDeployment` を使用する場合、ネストされた `@ActivationConfigProperty` アノテーションを使用するか、`@MessageDrivenDeployment` の属性を使用してメッセージ・サービス・オプションを構成できます。`@ActivationConfigProperty` 構成は、`@MessageDrivenDeployment` の属性をオーバーライドします。

`@MessageDriven` を使用する場合、ネストされた `@ActivationConfigProperty` アノテーションのみを使用してメッセージ・サービス・オプションを構成できます。

`@MessageDrivenDeployment` の属性を使用した構成の場合、アプリケーションでは、J2CA リソース・アダプタなしでメッセージ・サービス・プロバイダにアクセスすることのみ可能です。後から J2CA リソース・アダプタを使用してメッセージ・サービス・プロバイダにアクセスすると、アプリケーションはデプロイに失敗します。ネストされた `@ActivationConfigProperty` アノテーションを使用した構成の場合、アプリケーションでは、J2CA リソース・アダプタの有無にかかわらずメッセージ・サービス・プロバイダにアクセスできます。アノテーションを使用して構成する場合、`@ActivationConfigProperty` による方法を使用することをお薦めします。

例 2-6 に、@ActivationConfigProperty アノテーションを使用した @MessageDrivenDeployment および @MessageDriven アノテーションによるメッセージ・サービス構成を示します。@MessageDrivenDeployment アノテーションの DestinationName アクティブ化構成プロパティは、@MessageDriven アノテーションの同じプロパティをオーバーライドします。

例 2-6 J2CA のメッセージ・サービス・プロバイダの @MessageDriven および @MessageDrivenDeployment アノテーション

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.ejb.ActivationConfigProperty;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="DestinationName", propertyValue="OracleASjms/MyQueue"
        )
    }
)

@MessageDrivenDeployment(
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="DestinationName", propertyValue="OracleASjms/DeployedQueue"
        ),
        @ActivationConfigProperty(
            propertyName="ResourceAdapter", propertyValue="OracleASjms"
        )
    }
)

public class JCAQueueMDB implements MessageListener
{
    public void onMessage(Message msg) {
        ...
    }
}
```

XML を使用したメッセージ・サービス構成

XML を使用する場合、orion-ejb-jar.xml ファイルの <message-driven-deployment> 要素または ejb-jar.xml ファイルの <message-driven> 要素を使用してメッセージ・サービス・オプションを構成できます。orion-ejb-jar.xml 構成は、ejb-jar.xml 構成をオーバーライドします。

orion-ejb-jar.xml ファイルの <message-driven-deployment> 要素を使用する場合、ネストされた <config-property> 要素を使用するか、<message-driven-deployment> の属性を使用してメッセージ・サービス・オプションを構成できます。<config-property> 構成は、<message-driven-deployment> の属性をオーバーライドします。

ejb-jar.xml ファイルの <message-driven> 要素を使用する場合、ネストされた <activation-config-property> 要素のみを使用してメッセージ・サービス・オプションを構成できます。

orion-ejb-jar.xml ファイルの <message-driven-deployment> 要素の属性を使用した構成の場合、アプリケーションでは、J2CA リソース・アダプタなしでメッセージ・サービス・プロバイダにアクセスすることのみ可能です。後から J2CA リソース・アダプタを使用してメッセージ・サービス・プロバイダにアクセスすると、アプリケーションはデプロイに失敗します。ネストされた <config-property> 要素を使用した構成の場合、アプリケーションでは、J2CA リソース・アダプタの有無にかかわらずメッセージ・サービス・プロバイダにアクセ

スできます。XML を使用して構成する場合、<config-property> による方法を使用することをお薦めします。

例 2-8 に、orion-ejb-jar.xml ファイルの <message-driven-deployment> 要素で <config-property> 要素を使用する方法を示します。また、例 2-7 に、ejb-jar.xml ファイルの <message-driven> 要素で <activation-config-property> 要素を使用する方法を示します。<message-driven-deployment> 要素の DestinationName アクティブ化構成プロパティは、<message-driven> 要素の同じプロパティをオーバーライドします。また、ejb-jar.xml ファイルの <message-driven> 要素では、<activation-config-property> 要素が <activation-config> 要素に含まれます。

例 2-7 ejb-jar.xml の <activation-config-property>

```
<message-driven>
    <ejb-name>JCA_QueueMDB</ejb-name>
    <ejb-class>test.JCA_MDB</ejb-class>
    <messaging-type>javax.jms.MessageListener</messaging-type>
    <transaction-type>Container</transaction-type>
    ...
    <activation-config>
        <activation-config-property>
            <activation-config-property-name>
                DestinationName
            </activation-config-property-name>
            <activation-config-property-value>
                OracleASJMSSubcontext
            </activation-config-property-value>
        </activation-config-property>
    </activation-config>
    ...
</message-driven>
```

例 2-8 orion-ejb-jar.xml の <config-property>

```
<message-driven-deployment
    name="JCA_QueueMDB"
    resource-adapter="OracleASjms">
    ...
    <config-property>
        <config-property-name>
            DestinationName
        </config-property-name>
        <config-property-value>
            OracleASJMSRASubcontext
        </config-property-value>
    </config-property>
    ...
</message-driven-deployment>
```

2 フェーズ・コミット（2PC）トランザクション用のメッセージ・サービスの構成

OC4J では、XA 対応リソースで 2PC トランザクションがサポートされます（2-24 ページの「グローバル・トランザクションまたは 2 フェーズ・コミット（2PC）トランザクションへの参加方法」を参照）。

> **注意：** リリース 10.1.3.1 では、MDB で J2CA および XA ファクトリを使用している場合にのみ、OC4J によりデフォルトで MDB 接続が自動登録されます。詳細は、2-34 ページの「2 フェーズ・コミット（2PC）XA トランザクションへの MDB の自動登録」を参照してください。

XA 準拠の JMS メッセージ・サービス・プロバイダを構成する方法の詳細は、次を参照してください。

- Oracle JMS コネクタ : 23-3 ページの「OC4J J2CA リソース・アダプタのデプロイ XML ファイルの構成」
- OEMS JMS: 23-5 ページの「jms.xml の構成」
- OEMS JMS データベース : 23-7 ページの「OEMS JMS データベース・プロバイダのインストールと構成」に記載されている手順 2

2 フェーズ・コミット（2PC）XA トランザクションへの MDB の自動登録

リリース 10.1.2 および 10.1.3.0 の OC4J では、通常の非 XA JMS コネクションと XA JMS コネクションの両方が、ネイティブ OEMS JMS プロバイダにより自動的に OC4J グローバル・トランザクションに登録されていました。リリース 10.1.3.1 では、XA JMS コネクションも通常の JMS コネクションも OC4J グローバル・トランザクションに登録されません。付属の JMS API を使用する場合、OEMS JMS の `javax.jms.XA*` 実装を使用して XA 接続を明示的に OC4J グローバル・トランザクションに登録する必要があります。また、非 XA JMS コネクションから作成された特定の JMS セッションのローカル・トランザクションも、明示的にコミットまたはロールバックする必要があります。

下位互換性を確保する目的から、リリース 10.1.3.1 でも自動登録機能を使用することは可能です（ただし非推奨）。自動登録はデフォルトで無効化されていますが、グローバル OC4J システム・プロパティ `oc4j.jms.pseudoTransactionEnlistment` を `true` に設定することで有効化できます。

XA コネクション・ファクトリを使用する（したがって XA セッションを作成する）ように構成された J2CA MDB は、通常の動作として自動登録されます。非 XA ファクトリで作成されたセッションは、登録されません。`oc4j.jms.pseudoTransactionEnlistment` プロパティは、非 XA セッションを強制的に登録する場合にのみ必要です。この設定は、主に J2CA を使用しないレガシー・アプリケーションに関連します。

詳細は、『Oracle Containers for J2EE サービス・ガイド』の「Java Message Service（JMS）」を参照してください。

OC4J EJB アプリケーション・クラスタリング・サービスについて

Oracle Application Server には、クラスタリングなどの高可用性およびフェイルオーバー・オプションの広範なスイートが用意されています。クラスタリングでは、適切なホスト間通信手段が構成された複数のホストにアプリケーション・サーバーおよびエンドユーザー・アプリケーション・コンポーネントが分散されます。

OC4J アプリケーション・クラスタリングは、HTTP セッションおよびステートフル・セッション Bean で使用可能な状態管理サービスです。このコンテキストでは、クラスタは、同じアプリケーション・セットをホスティングしている 2 つ以上の OC4J サーバー・ノードとして定義されます。このリリースでは、構成が単純化され、HTTP セッションとステートフル・セッション Bean の両方に対して同一になっています。

トランザクションはフェイルオーバーできません。中断されたトランザクションを別の Bean に再インスタンス化する機能はありません。かわりに、トランザクションはロールバックされるため、最初からやりなおす必要があります。詳細は、2-20 ページの「EJB トランザクション・サービスについて」を参照してください。

ステートフル・セッション Bean のクラスタリングのパフォーマンスは、選択するレプリケーションのタイプ（2-36 ページの「状態レプリケーション」を参照）およびロード・バランシング（2-36 ページの「ロード・バランシング」を参照）のオプションによって変わります。

レプリケーション頻度と堅牢性のバランスを適切に選択する必要があります。レプリケートの頻度が高いほど、状態が失われる可能性のある期間が短くなりますが、アプリケーション・サーバーとネットワークに対する負荷が高くなります。

注意： ステートフル・セッション Bean を起動するサーブレット（または他の Web コンポーネント）がある場合は、HTTP セッションとステートフル・セッション Bean の両方のクラスタリングを構成する必要があります。

この項では、次のようなステートフル・セッション Bean の OC4J アプリケーション・クラスタリングについて説明します。

- 状態レプリケーション
- ロード・バランシング

詳細は、次を参照してください。

- 第 24 章「OC4J EJB アプリケーション・クラスタリング・サービスの構成」
- 『Oracle Containers for J2EE 構成および管理ガイド』の「クラスタリングの概要」
- 『Oracle Containers for J2EE 構成および管理ガイド』の「OC4J でのアプリケーションのクラスタリング」
- 『Oracle Application Server 高可用性ガイド』のアクティブ / アクティブ・トポロジでの Oracle Application Server クラスタ（OC4J）に関する項
- 『Oracle Application Server 高可用性ガイド』の Oracle Application Server クラスタ（OC4J）でのステートフル・セッション EJB の状態レプリケーションに関する項

状態レプリケーション

クラスタリングされた OC4J EJB アプリケーションのレプリケーション・ポリシーを構成する場合、OC4J はステートフル・セッション Bean インスタンスに含まれているオブジェクトと値のレプリケーションを処理します。クラスタリングできるのはステートフル・セッション Bean のみです。ステートレス・セッション Bean にはレプリケートする状態がないため、クラスタリングする必要はありません。

フェイルオーバーを利用するにはレプリケーション・ポリシーを構成する必要があります。元の Bean が予期せず終了した場合にリクエストがクラスタ内の別の OC4J プロセスに透過的に転送されるように、Bean の状態がレプリケートされる必要があります。ロード・バランシングのみ利用する場合は、レプリケーションは不要です（2-36 ページの「ロード・バランシング」を参照）。

レプリケーション・ポリシーによって、状態レプリケーション・トリガー（Bean 状態がクラスタ内の他のすべての OC4J プロセスにブロードキャストされる条件）が決定されます。ステートフル・セッション Bean の場合、レプリケーションがトリガーされると、ステートフル・セッション Bean のすべての属性が（変更されたかどうかに関係なく）レプリケートされます。

レプリケーションは、アプリケーション・サーバーおよびネットワーク・パフォーマンスに影響することがあります。状態を送信する回数が少ないほど、パフォーマンスは向上します。ただし、パフォーマンスと、Bean インスタンスの障害の全範囲を対象とするように Bean の状態がレプリケートされる確実性との間にはトレードオフがあります。

詳細は、次を参照してください。

- 24-2 ページの「EJB 3.0 および EJB 2.1 ステートフル・セッション Bean レプリケーション・ポリシーの構成」
- 2-35 ページの「OC4J EJB アプリケーション・クラスタリング・サービスについて」

ロード・バランシング

ロード・バランシングは、着信クライアント・リクエストがクラスタ内のすべての OC4J インスタンスに配布される方法を表します。次のロード・バランシング計画から選択できます。

- レプリケーションベース : クラスタリングされた OC4J EJB アプリケーション（2-36 ページの「状態レプリケーション」を参照）のレプリケーション・ポリシーを構成する場合、OC4J は最初のクライアント・リクエストが処理されたときにクラスタ内の OC4J インスタンスのプールから OC4J インスタンスを自動的にランダムに選択します。
- 静的検出 : EJB のレプリケーションを使用せずに、静的に指定した複数の OC4J プロセス間のクライアント・リクエストをロード・バランシングする場合は、これらのプロセスすべての URL を JNDI の URL プロパティに指定することによって、静的な検出を使用できます。詳細は、24-3 ページの「静的検出ロード・バランシングの構成」を参照してください。
- DNS: EJB のレプリケーションを使用せずに、複数の DNS 管理 OC4J プロセス間のクライアント・リクエストをロード・バランシングする場合は、目的の OC4J ホスト IP アドレスに関連付けられている単一のホスト名で DNS サーバーを構成し、このホスト名を JNDI の URL プロパティに指定することによって、DNS 検出を使用できます。詳細は、24-4 ページの「DNS ロード・バランシングの構成」を参照してください。

すべてのロード・バランシング計画において、クライアントのリクエストをクラスタ内の OC4J インスタンス間でロード・バランシングする方法を構成できます（24-5 ページの「ロード・バランシングの動作の構成」を参照）。

EJB タイマー・サービスについて

指定時刻、指定した時間の経過後または指定間隔でタイムアウト・コールバック・メソッドを起動するタイマーを設定できます。

注意： タイマーは、ステートフル・セッション Bean および EJB 3.0 エンティティを除くすべての EJB タイプに適用されます。

EJB タイマーは、単一の JVM で稼働する OC4J インスタンスでのみサポートされます（`opmn.xml` 構成ファイルの `<process-set>` 要素で、numprocs=1）。

EJB 3.0 アプリケーションでは、`@Timeout` アノテーションを使用して、EJB メソッドをタイムアウト・コールバック・メソッドとして指定できます。

EJB 2.1 アプリケーションでは、EJB で `TimedObject` インタフェースを実装し、`ejbTimeout` というタイムアウト・コールバック・メソッドを提供する必要があります。

タイマーは、アプリケーションレベルのプロセスのモデリング用です。リアルタイムのイベントには使用しません。

OC4J には、標準の Java EE タイマー以外に、UNIX の cron ユーティリティと同様の構成を可能にする Java EE タイマーの便利な拡張機能が付属します。

詳細は、次を参照してください。

- 2-37 ページの「Java EE タイマー・サービスについて」
- 2-37 ページの「OC4J cron タイマー・サービスについて」

タイマーとタイムアウト・コールバック・メソッドは、トランザクション内でコールされる必要があります。OC4J では、タイムアウト・コールバック用に `REQUIRES_NEW` トランザクション属性がサポートされます。トランザクション属性の詳細は、2-23 ページの「クライアントがビジネス・メソッドを起動する際のトランザクションの処理方法」を参照してください。

Enterprise Bean は、依存性注入、`EJBContext` インタフェースまたは JNDI ネームスペースのルックアップによって EJB タイマー・サービスにアクセスします。

詳細は、第 25 章「タイマー・サービスの構成」を参照してください。

Java EE タイマー・サービスについて

EJB タイマー・サービスは、時間ベースのイベントを対象にスケジュールされる EJB のコールバック・メソッドを定義するためのコンテナ管理のサービスです。EJB タイマー・サービスには、時間指定されたイベントに対する信頼性のあるトランザクション通知サービスが用意されています。タイマー通知は、指定時刻、指定した時間の経過後または特定の反復間隔で行うようにスケジュールできます。EJB のコールバック・メソッドを定義して、これらの時間ベースのイベントを受信できます。Java EE タイマー・サービスは、OC4J により実装されます。

詳細は、25-2 ページの「Java EE タイマーを使用する Enterprise Bean の構成」を参照してください。

OC4J cron タイマー・サービスについて

UNIX では、指定された間隔で定期的に実行されるように cron タイマーをスケジュールできます。オラクル社は、EJB で cron タイマーをサポートするように OC4J を拡張しました。OC4J にデプロイされる EJB とともにタイマー・イベントをスケジュールするために cron 式を使用できます。OC4J の cron タイマーを使用すると、タイムアウト・コールバック・メソッドまたは任意の Java クラスの `main` メソッドを起動するタイマーを作成できます。

詳細は、25-4 ページの「OC4J cron タイマーを使用する Enterprise Bean の構成」を参照してください。

3

OC4J での EJB サポートについて

この章の内容は次のとおりです。

詳細は、Oracle Application Server のリリース・ノートを参照してください。

EJB 3.0 サポート

このリリースでは、OC4J は EJB 3.0 最終仕様（http://jcp.org/aboutJava/communityprocess/pr/jsr220/index.html）で指定されたごく一部のサブセットを除くすべての機能をサポートします。

OC4J が完全な EJB 3.0 準拠に更新された後で、EJB 3.0 OC4J アプリケーションに若干のコード変更を加えることが必要な場合があります。詳細は、3-7 ページの「リリース 10.1.3.0 の TopLink JPA プレビュー・アプリケーションからリリース 10.1.3.1 の TopLink Essentials JPA への移行」を参照してください。

このリリースの OC4J では、オブジェクト・リレーショナル・エンティティ・マッピング・タイプ（基本、バイナリ・ラージ・オブジェクト（LOB）、シリアライズ、1 対 1、多対 1、1 対多、多対多および集約マッピング）を除くすべての EJB 3.0 機能に対して、アノテーションまたは標準のデプロイ XML（ejb-jar.xml または orion-ejb-jar.xml）の一方または両方の機能を使用できます。この場合、アノテーションまたは TopLink JPA 永続性プロバイダのカスタマイズを使用する必要があります。

OC4J では、『Oracle Application Server Annotations API Reference』に記載されている固有の EJB 3.0 アノテーションがサポートされます。

詳細は、次を参照してください。

- 第 6 章「JPA エンティティの実装」
- 3-4 ページの「JPA 永続性プロバイダのカスタマイズ」

このリリースの OC4J では、Web 層でのリソース・インジェクションがサポートされます。詳細は、1-9 ページの「Web 層でのアノテーション」を参照してください。

この項の内容は次のとおりです。

- 必要な JDK
- EJB 3.0 アプリケーションの定義方法
- EJB 3.0 アプリケーションで OC4J が永続性を管理する方法

必要な JDK

デフォルトでは、EJB 3.0 を使用している場合は、JDK 1.5 を使用する必要があります。OC4J のデフォルトでは、EJB 3.0 と JDK 1.4 の使用はサポートされません。

OC4J で EJB 3.0（アノテーションおよびインターセプタを除く）と JDK 1.4 の使用がサポートされるのは、TopLink JPA プレビュー永続性プロバイダを使用する場合のみです。詳細は、3-3 ページの「JPA 永続性 JAR ファイル」に記載されているシステム・プロパティ default.persistence.provider の説明を参照してください。

EJB 3.0 アプリケーションの定義方法

エンティティの場合、OC4J は、EJB JAR が ejb-jar.xml ファイルなしでデプロイされる場合にアプリケーションが EJB 3.0 アプリケーションであると想定します。詳細は、2-15 ページの「EJB 永続性サービスについて」を参照してください。

セッション Bean およびメッセージドリブン Bean の場合、OC4J は、ejb-jar.xml ファイルの <ejb-jar> 要素の version 属性が 3.0 に設定されている場合にアプリケーションが EJB 3.0 アプリケーションであると想定します。

EJB 3.0 アプリケーションで OC4J が永続性を管理する方法

EJB 3.0 アプリケーションでは、OC4J は JPA 永続性プロバイダに永続性操作を委任します。このリリースの OC4J は、EJB 3.0 リファレンス実装の JPA 永続性プロバイダである TopLink Essentials を使用します（3-3 ページの「TopLink Essentials JPA 永続性プロバイダ」を参照）。

TopLink Essentials JPA 永続性プロバイダ

Oracle TopLink は、高度なオブジェクト永続性およびオブジェクト変換フレームワークであり、開発作業と保守作業を削減し、エンタープライズ・アプリケーションの機能を高める開発ツールおよびランタイム機能を提供します。

このリリースでは、OC4J は EJB 3.0 リファレンス実装の JPA 永続性プロバイダである TopLink Essentials を使用して EJB 3.0 エンティティを管理します。詳細は、『Oracle TopLink 開発者ガイド』の TopLink の概要に関する項を参照してください。

OC4J には、EJB 3.0 永続性仕様に規定されているクラスと、TopLink Essentials JPA 永続性プロバイダ実装を構成するクラスの両方に対応する JAR ファイルが付属します。永続性 JAR ファイルの詳細は、3-3 ページの「JPA 永続性 JAR ファイル」を参照してください。

EJB 3.0 プロジェクトの場合は、アノテーションまたは `persistence.xml` ファイルを通じて永続性プロパティを構成します。OC4J は、このメタデータを TopLink 構成に変換します。TopLink Essentials EJB 3.0 JPA 永続性プロバイダのカスタマイズの詳細は、3-4 ページの「JPA 永続性プロバイダのカスタマイズ」を参照してください。

JPA 永続性 JAR ファイル

OC4J では、TopLink Essentials JPA 永続性プロバイダ実装を提供するために、表 3-1 にリストされた JAR ファイルを使用します。これらの JAR ファイルは、`<ORACLE_HOME>/toplink/jlib` ディレクトリにあります。

OC4J で使用される JPA 永続性プロバイダ実装は、システム・プロパティ `default.persistence.provider` により決定されます。有効な値は次のとおりです。

- `essentials`（デフォルト）: OC4J では、`<ORACLE_HOME>/j2ee/home/lib/persistence.jar` を使用して EJB 3.0 永続性仕様に規定された EJB 3.0 JPA クラスを提供し、永続性プロバイダ実装として `<ORACLE_HOME>/toplink/jlib/toplink-essentials.jar` および `toplink-essentials-agent.jar` を使用して EJB 3.0 永続性の最終仕様の完全サポートを提供します。
- `toplink`: OC4J では、`<ORACLE_HOME>/j2ee/home/lib/preview-persistence.jar` を使用して EJB 3.0 パブリック・レビュー永続性仕様に規定された EJB 3.0 JPA クラスを提供し、永続性プロバイダ実装として `<ORACLE_HOME>/toplink/jlib/toplink.jar` を使用して EJB 3.0 パブリック・レビュー・ドラフトに指定された一部の機能に基づく JPA プレビューを提供します。このオプションを使用すると、プレビュー API に記述されたアプリケーションを実行できます。このオプションは使用しないことをお薦めします。

表 3-1 TopLink JAR ファイル

JAR ファイル	内容
antlr.jar	この JAR ファイルには、Antlr（ANother Tool for Language Recognition）ツールが含まれます。
toplink.jar	この JAR ファイルには、Oracle JDBC 依存を伴うクラスなど、TopLink API を構成するすべてのクラスが含まれます。 OC4J とともにインストールされる Oracle JDBC ドライバのデフォルト・バージョンとは異なるバージョンを使用する場合、20-5 ページの「TopLink と Oracle JDBC ドライバとの関連付け」を参照してください。
toplink-essentials.jar	この JAR ファイルには、EJB 3.0 リファレンス実装の JPA 永続性プロバイダであるオープン・ソース JPA エディションの TopLink が含まれます。
toplink-essentials-agent.jar	この JAR ファイルには、JPA エンティティに対するバイトコード・ウィービングを実行して ValueHolder インダイレクションなどの機能を自動的に有効化する際に TopLink が使用するクラスが含まれます。toplink-essentials-agent.jar を起動するには、クライアントの JVM コマンドラインに -javaagent:toplink-essentials-agent.jar を追加するか、persistence.xml ファイルに定義できる toplink.weaving TopLink JPA 拡張を使用します。このオプションは、サーバーの JVM コマンドラインには不要です。 この JAR ファイルでは、EJB 3.0 リファレンス実装の JPA 永続性プロバイダである TopLink Essentials の一部が提供されます。このファイルは、toplink-essentials.jar と組み合せて使用します。 この JRA はオプションです。クラスパスには配置せず、-javaagent の一部としてのみ使用してください。
toplink-oc4j.jar	この JAR には、Oracle Containers for J2EE との統合のために TopLink が使用するクラスが含まれます。 この JAR ファイルは、OC4J でのみ使用します。コンテナは、toplink-oc4j.jar を使用するよう事前構成されています。OC4J 以外のアプリケーションでは、toplink.jar を使用してください。

JPA 永続性プロバイダのカスタマイズ

通常は、オブジェクト・リレーショナル・アノテーション（7-10 ページの「JPA エンティティのコンテナ管理の関連性フィールドの構成」を参照）を使用して、OC4J がデータベースに永続性フィールドを格納する方法を指定します。また、このような各アノテーションについてデフォルトの TopLink EJB 3.0 JPA 永続性プロバイダ構成に依存します。ただし、このデフォルト動作は、アプリケーション要件に合うようオーバーライドできます。同様に、TopLink EJB 3.0 JPA 永続性プロバイダは JPA 準拠ですが、JPA 仕様に定義された機能を超える追加の拡張機能を使用できます。

OC4J JPA 永続性プロバイダは、次の方法でカスタマイズできます。

- 名前付き問合せまたは動的問合せにベンダー固有の問合せヒントを設定します（8-4 ページの「JPA 問合せでの TopLink 問合せヒントの構成」を参照）。
- persistence.xml ファイルの <properties> 要素に、または javax.persistence.Persistence のメソッド createEntityManagerFactory に渡されるプロパティの Map にベンダー固有のプロパティを設定します（26-6 ページの「永続性ユニットでのベンダー拡張の構成」を参照）。
- TopLink Essentials 永続性プロバイダを使用している場合（デフォルト）、TopLink JPA 拡張を使用して実行時に JPA エンティティ・アプリケーションで TopLink API にアクセスできます（3-5 ページの「TopLink Essentials JPA 永続性を使用した TopLink API への実行時アクセス」を参照）。
- TopLink JPA プレビュー永続性プロバイダを使用している場合、ejb3-toplink-sessions.xml および toplink-ejb-jar.xml ファイルを作成し、EJB 3.0 エンティティを含む EJB-JAR の META-INF ディレクトリにこれらをパッケージ化することにより、実行時に JPA エンティティ・アプリケーションで TopLink API にアクセスできます（3-5 ページの「TopLink JPA プレビュー永続性を使用した TopLink API への実行時アクセス」を参照）。

- TopLink EJB 3.0 JPA 永続性プロバイダを構成して、OC4J とともにインストールされる Oracle JDBC ドライバのデフォルト・バージョンとは異なるバージョンを使用します（20-5 ページの「TopLink と Oracle JDBC ドライバとの関連付け」を参照）。

TopLink Essentials JPA 永続性を使用した TopLink API への実行時アクセス

TopLink Essentials JPA 永続性プロバイダを使用している場合（デフォルト）、TopLink JPA 拡張の `toplink.session.customizer` および `toplink.descriptor.customizer.<ENTITY>` を使用して実行時に JPA エンティティ・アプリケーションで TopLink API にアクセスできます（26-18 ページの表 26-5 を参照）。

TopLink JPA プレビュー永続性を使用した TopLink API への実行時アクセス

TopLink JPA プレビュー永続性プロバイダを使用している場合、`ejb3-toplink-sessions.xml` ファイル（2-9 ページの「ejb3-toplink-sessions.xml ファイルとは」を参照）および `toplink-ejb-jar.xml` ファイル（2-8 ページの「toplink-ejb-jar.xml ファイルとは」を参照）を作成して実行時に JPA エンティティ・アプリケーションで TopLink API にアクセスできます。

> **注意：** OC4J では、TopLink Essentials JPA 永続性プロバイダがデフォルトで使用されます。この場合、TopLink JPA 拡張を使用して実行時に JPA エンティティ・アプリケーションで TopLink API にアクセスします（3-5 ページの「TopLink Essentials JPA 永続性を使用した TopLink API への実行時アクセス」を参照）。

EJB 3.0 エンティティを含む EJB-JAR の META-INF ディレクトリにこれらのファイルをパッケージ化します。

- TopLink セッションレベル・オプションのカスタマイズに必要なのは、`ejb3-toplink-sessions.xml` ファイルのみです。
- TopLink の永続性固有のオプションをカスタマイズするには、`ejb3-toplink-sessions.xml` ファイルと `toplink-ejb-jar.xml` ファイルの両方が必要です。

TopLink API を使用して、アノテーションをオーバーライドするかアノテーションを完全に置換することにより、永続性をカスタマイズできます。たとえば、オブジェクト・リレーショナル・マッピングの大部分および `ejb3-toplink-sessions.xml` ファイルと `toplink-ejb-jar.xml` ファイルでアノテーションを使用して、アノテーションに適していない複雑な関連のサブセットに対してオブジェクト・リレーショナル・マッピングを指定できます。

エンティティ・クラスで使用している JDK 1.5 言語拡張がアノテーションのみの場合は、TopLink Workbench を使用してこれらのファイルを作成および構成できます。TopLink Workbench を使用して、これらのファイルを作成および構成することをお薦めします。

TopLink JPA プレビュー永続性プロバイダをカスタマイズするには、次のようにします。

1. リレーショナル TopLink Workbench プロジェクトを作成します（『Oracle TopLink 開発者ガイド』のプロジェクトの作成に関する項を参照）。
2. JDK 1.5 準拠のエンティティ・クラスを含むように TopLink Workbench プロジェクト・クラスパスを構成します（『Oracle TopLink 開発者ガイド』のプロジェクトのクラスパスの構成に関する項を参照）。
3. プロジェクトのデプロイ XML ファイル名を（`toplink-ejb-jar.xml` として）構成し、場所を保存します（『Oracle TopLink 開発者ガイド』のプロジェクトのデプロイ XML オプションの構成に関する項を参照）。
4. オプションで、他の TopLink プロジェクトレベル・オプションを構成します（『Oracle TopLink 開発者ガイド』のリレーショナル・プロジェクトの構成に関する項を参照）。

5. カスタマイズするエンティティ・クラスの TopLink リレーショナル・ディスクリプタを構成します（『Oracle TopLink 開発者ガイド』のリレーショナル・ディスクリプタの作成に関する項および『Oracle TopLink 開発者ガイド』のリレーショナル・ディスクリプタの構成に関する項を参照）。
6. カスタマイズする永続性フィールドの TopLink リレーショナル・マッピングを構成します（『Oracle TopLink 開発者ガイド』のマッピングの作成に関する項および『Oracle TopLink 開発者ガイド』のリレーショナル・マッピングの構成に関する項を参照）。
7. TopLink Workbench プロジェクトを `toplink-ejb-jar.xml` XML ファイルにエクスポートします（『Oracle TopLink 開発者ガイド』のデプロイ XML 情報のエクスポートに関する項を参照）。
8. `ejb3-toplink-sessions.xml` という名前の TopLink セッション構成ファイルを作成します（『Oracle TopLink 開発者ガイド』のサーバー・セッションの作成に関する項を参照）。
9. `ejb3-toplink-sessions.xml` ファイルのプライマリ・プロジェクトを `toplink-ejb-jar.xml` ファイルに設定します（『Oracle TopLink 開発者ガイド』のプライマリ・マッピング・プロジェクトの構成に関する項を参照）。
10. オプションで、他の任意の TopLink セッションレベル・オプションを構成します（『Oracle TopLink 開発者ガイド』のサーバー・セッションの構成に関する項を参照）。
11. TopLink Workbench セッション構成ファイルを保存します。
12. EJB 3.0 エンティティを含む EJB-JAR の META-INF ディレクトリに `ejb3-toplink-sessions.xml` および `toplink-ejb-jar.xml` ファイルをパッケージ化します。

注意： または、JDeveloper を使用して、`ejb3-toplink-sessions.xml` および `toplink-ejb-jar.xml` ファイルを作成できます（2-2 ページの「EJB 開発ツールの使用方法」を参照）。

リリース 10.1.3.0 の TopLink JPA プレビュー・アプリケーションからリリース 10.1.3.1 の TopLink Essentials JPA への移行

リリース 10.1.3.0 の場合、OC4J では、EJB 3.0 パブリック・レビュー・ドラフトで指定された機能のサブセットに基づく TopLink JPA プレビュー永続性プロバイダを使用します。

リリース 10.1.3.1 の場合、OC4J では、EJB 3.0 リファレンス実装の JPA 永続性プロバイダである TopLink Essentials JPA 永続性プロバイダを使用して、EJB 3.0 最終仕様に従って JPA を完全にサポートします。

OC4J リリース 10.1.3.1 で TopLink Essentials および EJB 3.0 最終版 API と組み合せて使用する前に、JPA プレビューに基づくアプリケーションのコードを変更する必要があります。

一般的に、次の手順を実行する必要があります。

1. プレビューベースのアプリケーションをアンデプロイします。
2. OC4J をリリース 10.1.3.0 から 10.1.3.1 にアップグレードします。
3. アップグレード後に必要な構成ファイルの変更を行います（3-7 ページの「OC4J 構成ファイルの変更」を参照）。
4. 新規 EJB 3.0 API を使用するようコードを移行します。

   次の項で、変更箇所の識別に役立つ TopLink JPA プレビューと TopLink 完全版 JPA の重要な相違点について説明します。

   - 3-8 ページの「javax.persistence の変更」
   - 3-12 ページの「oracle.toplink.essentials.platform.database の変更」
   - 3-12 ページの「インターセプタ・サポートの変更」
   - 3-12 ページの「エンティティ・マネージャの取得」
   - 3-13 ページの「新規 JAR ファイル」
5. 更新したアプリケーションを再デプロイします。

OC4J 構成ファイルの変更

OC4J リリース 10.1.3.0.0 にリリース 10.1.3.1.0 のパッチ・セットを適用したら、次のように OC4J 構成ファイルを手動で編集する必要があります。

1. `<ORACLE_HOME>/j2ee/home/config/server.xml` ファイルを編集して次の要素を追加します。

```
<shared-library name="oracle.persistence" version="1.0" library-compatible="true">
    <code-source path="../../../toplink/jlib/toplink-essentials.jar"/>
</shared-library>
```

2. `<ORACLE_HOME>/j2ee/home/config/system-application.xml` ファイルを編集して `<imported-shared-libraries>` 要素に次の属性を追加します。

```
<import-shared-library name="oracle.persistence"/>
```

javax.persistence の変更

表 3-2 に、リリース 10.1.3.0 と 10.1.3.1 の javax.persistence パッケージで追加、削除および変更された内容をリストします。アプリケーションでこれらのクラスを使用する場合、最新の EJB 3.0 仕様と JPA Javadoc でその詳細を確認してください。

表 3-2 javax.persistence の変更

10.1.3.0	10.1.3.1	説明
AccessMode	削除	リリース 10.1.3.1 では、EJB 3.0 エンティティにローカルまたはリモート・インタフェースは必要ありません。すべてのエンティティ・アクセスは EntityManager を通じて行われるため、クライアントではアクセスがローカルかリモートかを考慮する必要がありません。
AccessType	削除	リリース 10.1.3.1 では、EJB 3.0 永続性を指定する場合にエンティティ階層の単一のアクセス・タイプを使用する必要があります。マッピング・アノテーションの置換により、有効なアクセス・タイプが決定されます。
N/A	AssociationOverride	継承用のアノテーションに対する変更の一環としてリリース 10.1.3.1 で追加されました。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#AssociationOverride を参照してください。
N/A	AssociationOverrides	継承用のアノテーションに対する変更の一環としてリリース 10.1.3.1 で追加されました。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#AssociationOverrides を参照してください。
Basic	Basic	リリース 10.1.3.1 では、属性 temporalType が省略されています。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#Basic を参照してください。
N/A	DiscriminatorValue	継承用のアノテーションに対する変更の一環としてリリース 10.1.3.1 で追加されました。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#InheritanceAnnotations を参照してください。
EmbeddableSuperclass	削除	継承用のアノテーションに対する変更の一環としてリリース 10.1.3.1 で追加されました。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#InheritanceAnnotations を参照してください。
Entity	Entity	リリース 10.1.3.1 では、属性 access が省略されています。
N/A	EntityExistsException	リリース 10.1.3.1 で追加されました。
EntityListener	削除	リリース 10.1.3.1 では、かわりに EntityListners を使用します。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#EntityListeners を参照してください。
N/A	EntityListeners	リリース 10.1.3.1 で追加されました。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#EntityListeners を参照してください。

表 3-2 javax.persistence の変更（続き）

10.1.3.0	10.1.3.1	説明
EntityManager	EntityManager	リリース 10.1.3.1 では、EntityManager のメソッド getUserTransaction の名前は、getTransaction になりました。 リリース 10.1.3.1 で追加された新規メソッドは、次のとおりです。 ■ setFlushMode ■ getFlushMode ■ lock ■ clear ■ joinTransaction ■ getDelegate リリース 10.1.3.1 では、このクラスのメソッドにより、EntityExistsException や IllegalStateException などの追加の例外がスローされます。また、IllegalArgumentException のかわりに IllegalStateException がスローされることもあります。
EntityNotFoundException	EntityNotFoundException	リリース 10.1.3.1 では、この例外は RuntimeException ではなく PersistenceException を拡張しています。
EntityTransaction	EntityTransaction	リリース 10.1.3.1 で追加された新規メソッドは、次のとおりです。 ■ setRollbackOnly ■ getRollbackOnly
EntityType	削除	リリース 10.1.3.1 では、EJB 3.0 仕様により BMP の概念が削除されています。 Java EE 5 BMP アプリケーションを開発およびデプロイするには、EJB 2.1 API を使用する必要があります。
N/A	Enumerated	列挙タイプのダイレクト・マッピングをサポートするため、リリース 10.1.3.1 で追加されました。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#Enumerated を参照してください。
N/A	EnumType	列挙タイプのダイレクト・マッピングをサポートするため、リリース 10.1.3.1 で追加されました。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#Enumerated を参照してください。
N/A	ExcludeDefaultListeners	ライフ・サイクル・コールバックのデフォルト・リスナーを管理するため、リリース 10.1.3.1 で追加されました。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#ExcludeDefaultListeners を参照してください。
N/A	ExcludeSuperclassListeners	ライフ・サイクル・コールバックのスーパークラス・リスナーを管理するため、リリース 10.1.3.1 で追加されました。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#ExcludeSuperclassListeners を参照してください。
FlushMode	削除	リリース 10.1.3.1 では、フラッシュ・モードを設定するために、EntityManager のメソッド setFlushMode を使用して FlushModeType を設定します。

表 3-2 javax.persistence の変更（続き）

10.1.3.0	10.1.3.1	説明
N/A	GeneratedValue	主キー（ID）の自動生成をサポートするため、リリース 10.1.3.1 で追加されました。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#GeneratedValue を参照してください。
N/A	GenerationType	主キー（ID）の自動生成をサポートするため、リリース 10.1.3.1 で追加されました。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#GeneratedValue を参照してください。
GeneratorType	削除	リリース 10.1.3.1 では、GeneratedValue の属性 strategy を使用して GenerationType を設定します。
Inheritance	Inheritance	リリース 10.1.3.1 では、次の属性が省略されています。 ■ discriminatorType ■ discriminatorValue 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#InheritanceAnnotations を参照してください。
JoinColumn	JoinColumn	リリース 10.1.3.1 では、属性 secondaryTable が table に変更されています。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#JoinColumn を参照してください。
JoinTable	JoinTable	リリース 10.1.3.1 では、属性 table（Table 型）が name（String 型）に変更されています。 次の属性が追加されました。 ■ catalog ■ schema ■ uniqueConstraints 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#JoinTable を参照してください。
LobType	削除	リリース 10.1.3.1 では、ダイレクト・マッピング・アノテーション @Lob を使用して LOB 型を指定します。LOB には、バイナリ型または文字型があります。永続性プロバイダは、永続性フィールドまたはプロパティの型から LOB 型を推測します。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#Lob を参照してください。
N/A	LockModeType	リリース 10.1.3.1 で追加されました。
N/A	MappedSuperclass	継承用のアノテーションに対する変更の一環としてリリース 10.1.3.1 で追加されました。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#MappedSuperclass を参照してください。

表 3-2 javax.persistence の変更（続き）

10.1.3.0	10.1.3.1	説明
NamedNativeQuery	NamedNativeQuery	リリース 10.1.3.1 では、属性 queryString が query に変更され、属性 hints が追加されました。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#NamedNativeQuery を参照してください。
NamedQuery	NamedQuery	リリース 10.1.3.1 では、属性 queryString が query に変更され、属性 hints が追加されました。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#NamedQuery を参照してください。
NoResultException	NoResultException	リリース 10.1.3.1 では、この例外は RuntimeException ではなく PersistenceException を拡張しています。
N/A	OptimisticLockException	リリース 10.1.3.1 で追加されました。
PersistenceContext	PersistenceContext	リリース 10.1.3.1 では、属性 properties が追加されました。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#PersistenceContext を参照してください。
N/A	PersistenceProperty	詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#PersistenceProperty を参照してください。
N/A	QueryHint	詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#QueryHint を参照してください。
N/A	RollbackException	リリース 10.1.3.1 で追加されました。
SecondaryTable	SecondaryTable	リリース 10.1.3.1 では、属性 pkJoin が pkJoinColumns に変更されています。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#SecondaryTable を参照してください。
SequenceGenerator	SequenceGenerator	リリース 10.1.3.1 では、属性 initialValue のデフォルトが 0 から 1 に変更されています。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#SequenceGenerator を参照してください。
N/A	SqlResultSetMappings	詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#SqlResultSetMappings を参照してください。
Table	Table	リリース 10.1.3.1 では、属性 specified が省略されています。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#Table を参照してください。

表 3-2 javax.persistence の変更（続き）

10.1.3.0	10.1.3.1	説明
TableGenerator	TableGenerator	リリース 10.1.3.1 では、次の属性が追加されました。 ■ catalog ■ schema ■ uniqueConstraints 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#TableGenerator を参照してください。
N/A	Temporal	詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#Temporal を参照してください。
TemporalType	TemporalType	リリース 10.1.3.1 では、列挙値 NONE が省略されています。 詳細は、http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#Temporal を参照してください。
TransactionRequiredException	TransactionRequiredException	リリース 10.1.3.1 では、この例外は RuntimeException ではなく PersistenceException を拡張しています。

oracle.toplink.essentials.platform.database の変更

リリース 10.1.3.1 では、次の新規クラスが追加されました。

- DerbyPlatform
- JavaDBPlatform
- PostgreSQLPlatform

インターセプタ・サポートの変更

OC4J で TopLink JPA を使用する場合、インターセプタは、TopLink JPA プレビューと EJB 3.0 最終仕様の間で大幅に変更されていることに注意してください。

EJB 3.0 最終仕様では、インターセプタとライフ・サイクル・イベント・リスナーがマージされ、その両方で javax.interceptors.Interceptors が使用されます。このことは、特にセッション Bean とメッセージドリブン Bean でインターセプタまたはライフ・サイクル・イベント・リスナーを使用しているすべてのコードに影響します。

詳細は、次を参照してください。

- 1-5 ページの「Enterprise Bean のライフ・サイクル」
- 2-12 ページの「EJB 3.0 インターセプタについて」

エンティティ・マネージャの取得

リリース 10.1.3.0 では、java.persistence.setup.config プロパティを使用して、エンティティ・マネージャが管理するエンティティのリストでクラスを識別します。

リリース 10.1.3.1 では、このプロパティは廃止されました。かわりに、EJB 3.0 仕様で指定されているとおり、永続性ユニットに管理対象エンティティ・クラスを定義する必要があります。

リリース 10.1.3.0 では、@Resource アノテーションを使用してエンティティ・マネージャを注入します。

リリース 10.1.3.1 では、次のように @PersistenceContext アノテーションを使用してエンティティ・マネージャを注入します。

```
@PersistenceContext protected EntityManager entityManager;
```

リリース 10.1.3.1 では、下位互換性を確保する目的で、OC4J により @Resource を使用したエンティティ・マネージャの注入がサポートされます。ただし、EJB 3.0 仕様に準拠するため、かわりに @PersistenceContext アノテーションを使用することをお薦めします。

詳細は、次を参照してください。

- 2-10 ページの「persistence.xml ファイルとは」
- 29-9 ページの「EntityManager の取得」
- http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#EntityManagerAnnotations
- http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-extensions.html#persistence-xml

新規 JAR ファイル

リリース 10.1.3.0 の場合、OC4J では、persistence-preview.jar および toplink.jar ファイルを使用して JPA プレビュー実装を提供します。

リリース 10.1.3.1 の場合、OC4J では、persistence.jar、toplink-essentials.jar および toplink-essentials-agent.jar ファイルを使用して完全な JPA 実装を提供します。

使用中の IDE で、プロジェクトに関連付けられたライブラリ定義にリリース 10.1.3.1 の JPA ライブラリのみが含まれ、旧リリースの 10.1.3.0 のライブラリは除外されていることを確認してください。

TopLink JAR ファイルの詳細は、3-3 ページの「JPA 永続性 JAR ファイル」を参照してください。

EJB 2.1 サポート

このリリースでは、OC4J は EJB 2.1 最終リリース仕様（http://java.sun.com/products/ejb/docs.html）で指定されている機能をサポートしています。

この項の内容は次のとおりです。

- 必要な JDK
- EJB 2.1 モジュールの定義方法
- EJB 2.1 アプリケーションで OC4J が永続性を管理する方法

必要な JDK

EJB 2.1 を使用している場合は、JDK 1.4 以上を使用する必要があります。

EJB 2.1 モジュールの定義方法

デフォルトでは、モジュール・バージョン（ejb-jar.xml ファイルの <ejb-jar> 要素の version 属性）は、2.x に設定されます。

通常、この値は、明示的に 3.0 に設定するか、ejb-jar.xml ファイルを省略した場合にのみ変化します。

CMP バージョン（ejb-jar.xml ファイルの <cmp-version> 要素）は、EJB モジュール・バージョンとは無関係です。EJB 2.x CMP エンティティ Bean の場合、<cmp-version> を 2.x に設定します。

この設定は、EJB 2.x CMP エンティティ Bean と EJB 3.0 エンティティを両方とも使用する EJB 3.0 モジュールが存在する場合に有効です。

詳細は、2-15 ページの「EJB 永続性サービスについて」を参照してください。

EJB 2.1 アプリケーションで OC4J が永続性を管理する方法

OC4J は、永続性マネージャに永続性操作を委任します。このリリースでは、OC4J はデフォルトで TopLink 永続性マネージャを使用します（3-14 ページの「TopLink EJB 2.1 永続性マネージャ」を参照）。

Orion 永続性マネージャは推奨されません。新規開発には OC4J および TopLink 永続性マネージャを使用することをお薦めします。移行ツール（3-15 ページの「TopLink EJB 2.1 永続性マネージャへの移行」を参照）を使用すると、Orion 永続性マネージャで EJB 2.0 エンティティ Bean を使用する既存の OC4J アプリケーションを簡単に移行して、TopLink 永続性マネージャで EJB 2.0 エンティティ Bean を使用できます。Orion 永続性マネージャの詳細は、『Oracle Containers for J2EE Orion CMP 開発者ガイド』を参照してください。

TopLink EJB 2.1 永続性マネージャ

Oracle TopLink は、高度なオブジェクト永続性およびオブジェクト変換フレームワークであり、開発作業と保守作業を削減し、エンタープライズ・アプリケーションの機能を高める開発ツールおよびランタイム機能を提供します。

このリリースでは、OC4J は、コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の永続性マネージャとして TopLink を使用します。TopLink 永続性マネージャの詳細は、『Oracle TopLink 開発者ガイド』の TopLink の概要に関する項を参照してください。

OC4J には、TopLink EJB 2.1 永続性マネージャ実装を構成するクラスの JAR ファイルが付属します。永続性 JAR ファイルの詳細は、3-15 ページの「EJB 2.1 永続性 JAR ファイル」を参照してください。

EJB 2.1 プロジェクトの場合は、TopLink Workbench（『Oracle TopLink 開発者ガイド』の TopLink Workbench の理解に関する項を参照）を使用して、`toplink-ejb-jar.xml` ファイル（2-8 ページの「toplink-ejb-jar.xml ファイルとは」を参照）で永続性プロパティを構成します。Orion CMP アプリケーションを TopLink 永続性に移行する場合（3-15 ページの「TopLink EJB 2.1 永続性マネージャへの移行」を参照）、TopLink 移行ツールは TopLink Workbench プロジェクトを自動的に作成します。

TopLink カスタマイズ・クラスを使用して実行時にこの構成をカスタマイズできます（3-15 ページの「TopLink EJB 2.1 永続性マネージャのカスタマイズ」を参照）。

EJB 2.1 永続性 JAR ファイル

OC4J では、TopLink EJB 2.1 永続性マネージャ実装を提供するために、表 3-3 にリストされた TopLink JAR ファイルを使用します。これらの JAR ファイルは、<ORACLE_HOME>/toplink/jlib ディレクトリにあります。

表 3-3 TopLink JAR ファイル

JAR ファイル	内容
antlr.jar	この JAR には、Antlr（ANother Tool for Language Recognition）ツールが含まれます。
toplink.jar	この JAR には、Oracle JDBC 依存を伴うクラスなど、TopLink API を構成するすべてのクラスが含まれます。 OC4J とともにインストールされる Oracle JDBC ドライバのデフォルト・バージョンとは異なるバージョンを使用する場合、20-5 ページの「TopLink と Oracle JDBC ドライバとの関連付け」を参照してください。
toplink-agent.jar	この JAR には、EJB 2.1 エンティティ Bean クラスに対するバイトコード・ウィービングを実行して ValueHolder を使用せずに透過的な 1 対 1 および多対 1 のインダイレクションを有効化する際に TopLink が使用するクラスが含まれます。toplink-agent.jar を起動するには、アプリケーションの JVM コマンドラインに -javaagent:toplink-agent.jar を追加します。この JAR は、TopLink アプリケーションのクラスパスには含めないでください。 この JRA はオプションです。クラスパスには配置せず、-javaagent の一部としてのみ使用してください。
toplink-oc4j.jar	この JAR には、Oracle Containers for J2EE との統合のために TopLink が使用するクラスが含まれます。 この JAR ファイルは、OC4J でのみ使用します。コンテナは、toplink-oc4j.jar を使用するよう事前構成されています。OC4J 以外のアプリケーションでは、toplink.jar を使用してください。

TopLink EJB 2.1 永続性マネージャのカスタマイズ

実行時に、TopLink 永続性マネージャ API を使用して高度な TopLink 機能を利用できます。

EJB 2.1 CMP アプリケーションで TopLink 永続性マネージャ API にアクセスするために、デプロイ JAR に TopLink カスタマイズ・クラスを含めることができます。

このオプションの Java クラスは、oracle.toplink.ejb.cmp.DeploymentCustomization を実装して、TopLink マッピングおよびランタイム構成のデプロイのカスタマイズを可能にします。デプロイ時に、TopLink ランタイムはこのクラスの新規インスタンスを作成し、そのメソッド beforeLoginCustomization（TopLink ランタイムがセッションにログインする前）および afterLoginCustomization（TopLink ランタイムがセッションにログインした後）を起動して、TopLink セッションをパラメータとして渡します。

beforeLoginCustomization メソッドの実装を使用して、キャッシュ調整、パラメータ使用の SQL、ネイティブ SQL、バッチ書込み / バッチ・サイズ、バイト配列 / 文字列バインディング、ログイン、イベント・リスナー、表の修飾子、順序付けなどの TopLink セッション属性を構成します。

EJB 2.1 の場合は、TopLink カスタマイズ・クラスを使用して、TopLink Workbench GUI からはアクセスできない TopLink 永続性マネージャ API にアクセスできます。

詳細は、次を参照してください。

- 『Oracle TopLink 開発者ガイド』の pm-properties の構成に関する項
- 『Oracle TopLink API Reference』

TopLink EJB 2.1 永続性マネージャへの移行

TopLink 移行ツールを使用すると、Orion 永続性マネージャで EJB 2.0 エンティティ Bean を使用する既存の OC4J アプリケーションを簡単に移行して、TopLink 永続性マネージャで EJB 2.0 エンティティ Bean を使用できます。

TopLink 移行ツールの使用方法の詳細は、『Oracle TopLink 開発者ガイド』の OC4J Orion 永続性から OC4J TopLink 永続性への移行に関する項を参照してください。

第 II 部

EJB 3.0 セッション Bean

第 II 部では、EJB 3.0 セッション Bean の実装および構成の手順に関する情報を示します。概念的な情報は、第 I 部「EJB の概要」を参照してください。

第 II 部は次の各章で構成されています。

- 第 4 章「EJB 3.0 セッション Bean の実装」
- 第 5 章「EJB 3.0 セッション Bean の使用方法」

4

EJB 3.0 セッション Bean の実装

この章では、次のような EJB 3.0 セッション Bean の実装方法を説明します。

- EJB 3.0 ステートレス・セッション Bean の実装
- EJB 3.0 ステートフル・セッション Bean の実装

詳細は、次を参照してください。

- 1-30 ページの「セッション Bean とは」
- 第 5 章「EJB 3.0 セッション Bean の使用方法」

EJB 3.0 ステートレス・セッション Bean の実装

EJB 3.0 では、ステートレス・セッション Bean の開発が大幅に単純化され、多くの複雑な開発タスクが排除されています。次に例を示します。

- Bean クラスは、Plain Old Java Object（POJO）にすることができます。`javax.ejb.SessionBean` を実装する必要はありません。
- ビジネス・インタフェースはオプションです。

  ホーム（`javax.ejb.EJBHome` および `javax.ejb.EJBLocalHome`）およびコンポーネント（`javax.ejb.EJBObject` および `javax.ejb.EJBLocalObject`）ビジネス・インタフェースは不要です。

  EJB 3.0 API に記述されたセッション Bean の EJB 3.0 ローカルまたはリモート・クライアントは、ビジネス・インタフェースを通じてセッション Bean にアクセスします。EJB 3.0 セッション Bean のビジネス・インタフェースは、Bean に対してローカルまたはリモート・アクセスが提供されるかどうかにかかわらず、通常の Java インタフェースです。
- 多くの機能にアノテーションが使用されます。
- `SessionContext` は不要です。単純に `this` を使用してセッション Bean を自身に関連付けできます。

詳細は、次を参照してください。

- 1-31 ページの「ステートレス・セッション Bean とは」
- 4-4 ページの「EJB 2.1 クライアントへの EJB 3.0 ステートレス・セッション Bean の適用」

注意： EJB 3.0 ステートレス・セッション Bean のコード例は、`http://www.oracle.com/technology/tech/java/oc4j/10131/how_to/how-to-ejb30-stateless-ejb/doc/how-to-ejb30-stateless-ejb.html` からダウンロードできます。

EJB 3.0 ステートレス・セッション Bean を実装するには、次のようにします。

1. ステートレス・セッション Bean クラスを作成します。

   Plain Old Java Object（POJO）を作成し、`@Stateless` アノテーションを使用してそれをステートレス・セッション Bean として定義できます。

   注意： OC4J では、`@Stateless` の属性 `mappedName` は無視されます。詳細は、1-29 ページの「OC4J によるアノテーション属性 mappedName のサポート」を参照してください。

   ---

2. ビジネス・メソッドを実装します。

   注意： ステートレス・セッション Bean には、remove メソッドは不要です。

   ---

3. オプションで、適切なアノテーションを使用してライフ・サイクル・コールバック・メソッドを定義します。

   ライフ・サイクル・メソッドを定義する必要はありません。このようなメソッドの実装はすべて OC4J に用意されています。ステートレス・セッション Bean のライフ・サイクルの特定の時点で独自のアクションを実行する場合にのみ、ステートレス・セッション Bean クラスのメソッドをライフ・サイクル・コールバック・メソッドとして定義します。

   詳細は、5-5 ページの「EJB 3.0 セッション Bean のライフ・サイクル・コールバック・インターセプタ・メソッドの構成」を参照してください。

4. オプションで、OC4J 固有のデプロイ・オプションを定義します。

   EJB 3.0 アプリケーションでこれを行うには、ステートレス・セッション Bean クラスに OC4J 固有の `oracle.j2ee.ejb.@StatelessDeployment` アノテーションを付けます（5-12 ページの「EJB 3.0 セッション Bean の OC4J 固有のデプロイ・オプションの構成」を参照）。

5. セッション Bean の構成を完了します（第 5 章「EJB 3.0 セッション Bean の使用方法」を参照）。

EJB 3.0 ステートフル・セッション Bean の実装

EJB 3.0 では、ステートフル・セッション Bean の開発が大幅に単純化され、多くの複雑な開発タスクが排除されています。次に例を示します。

- Bean クラスは、POJO にすることができます。`javax.ejb.SessionBean` を実装する必要はありません。
- ビジネス・インタフェースはオプションです。

  ホーム（`javax.ejb.EJBHome` および `javax.ejb.EJBLocalHome`）およびコンポーネント（`javax.ejb.EJBObject` および `javax.ejb.EJBLocalObject`）ビジネス・インタフェースは不要です。

  EJB 3.0 API に記述されたセッション Bean の EJB 3.0 ローカルまたはリモート・クライアントは、ビジネス・インタフェースを通じてセッション Bean にアクセスします。EJB 3.0 セッション Bean のビジネス・インタフェースは、Bean に対してローカルまたはリモート・アクセスが提供されるかどうかにかかわらず、通常の Java インタフェースです。
- 多くの機能にアノテーションが使用されます。
- `SessionContext` は不要です。単純に `this` を使用してセッション Bean を自身に関連付けできます。

詳細は、次を参照してください。

- 1-31 ページの「ステートレス・セッション Bean とは」
- 4-5 ページの「EJB 2.1 クライアントへの EJB 3.0 ステートフル・セッション Bean の適用」

注意： EJB 3.0 ステートフル・セッション Bean のコード例は、`http://www.oracle.com/technology/tech/java/oc4j/10131/how_to/how-to-ejb30-stateful-ejb/doc/how-to-ejb30-stateful-ejb.html` からダウンロードできます。

EJB 3.0 ステートフル・セッション Bean を実装するには、次のようにします。

1. ステートフル・セッション Bean クラスを作成します。

   POJO を作成し、`@Stateful` アノテーションを使用してそれをステートフル・セッション Bean として定義できます。

   注意： OC4J では、`@Stateful` の属性 `mappedName` は無視されます。

   ---

2. ビジネス・メソッドを実装します。

   ステートフル・セッション Bean クラスを `remove` メソッドとして定義するには、`@Remove` アノテーションを使用します。

3. オプションで、適切なアノテーションを使用してライフ・サイクル・コールバック・メソッドを定義します。

   ライフ・サイクル・メソッドを定義する必要はありません。このようなメソッドの実装はすべて OC4J に用意されています。ステートフル・セッション Bean のライフ・サイクルの特定の時点で独自のアクションを実行する場合にのみ、ステートフル・セッション Bean クラスのメソッドをライフ・サイクル・コールバック・メソッドとして定義します。

   詳細は、5-5 ページの「EJB 3.0 セッション Bean のライフ・サイクル・コールバック・インターセプタ・メソッドの構成」を参照してください。

4. オプションで、OC4J 固有のデプロイ・オプションを定義します。

   EJB 3.0 アプリケーションでこれを行うには、ステートフル・セッション Bean クラスに OC4J 固有の `oracle.j2ee.ejb.@StatefulDeployment` アノテーションを付けます（5-12 ページの「EJB 3.0 セッション Bean の OC4J 固有のデプロイ・オプションの構成」を参照）。

5. セッション Bean の構成を完了します（第 5 章「EJB 3.0 セッション Bean の使用方法」を参照）。

EJB 2.1 クライアントへの EJB 3.0 ステートレス・セッション Bean の適用

EJB 3.0 ステートレス・セッション Bean を EJB 2.1 ホームおよびコンポーネント・インタフェースに関連付けることで（4-4 ページの「アノテーションの使用方法」を参照）、EJB 2.1 クライアントからアクセスできるように EJB 3.0 ステートレス・セッション Bean を調整できます。

この技術を使用することで、EJB 2.1 アプリケーションから EJB 3.0 への増分移行を管理することや、既存の EJB 2.1 クライアントから EJB 3.0 を使用して実装した新規開発機能にアクセスすることが可能になります。

EJB 2.1 ホームおよびコンポーネント・インタフェースの詳細は、次を参照してください。

- 11-7 ページの「ホーム・インタフェースの実装」
- 11-10 ページの「コンポーネント・インタフェースの実装」

アノテーションの使用方法

EJB 2.1 クライアントに EJB 3.0 ステートレス・セッション Bean を適用するには、次のようにします。

1. EJB 2.1 ホーム・インタフェースを EJB 3.0 ステートレス・セッション Bean に関連付けます。

   次のように、リモート・ホーム・インタフェースには `@RemoteHome` アノテーションを使用し、ローカル・ホーム・インタフェースには `@LocalHome` アノテーションを使用します。

```
@Stateless
@RemoteHome (value=Ejb21RemoteHome1.class)
@LocalHome (value=Ejb21LocalHome.class)
public class MyStatelessSB {
...
}
```

注意： ステートレス・セッション Bean は、最大で 1 つのリモートおよびローカル・ホーム・インタフェースに関連付けることができます。

2. ホーム・インタフェースの `create` メソッドをサポートする要件を検討します。

   EJB 3.0 ステートレス・セッション Bean には、ホーム・インタフェースがある場合でも `ejbCreate` メソッドは必要ありません。かわりに、post-construct ライフ・サイクル・コールバック・メソッドを定義できます（5-5 ページの「EJB 3.0 セッション Bean のライフ・サイクル・コールバック・インターセプタ・メソッドの構成」を参照）。

3. EJB 2.1 コンポーネント・インタフェースを EJB 3.0 ステートレス・セッション Bean に関連付けます。

   次のように、リモート・コンポーネント・インタフェースには `@Remote` アノテーションを使用し、ローカル・コンポーネント・インタフェースには `@Local` アノテーションを使用します。

```
@Stateless
@Remote (value={Ejb21Remote1.class, Ejb21Remote2.class})
@Local (value={Ejb21Local.class})
public class MyStatelessSB {
...
}
```

注意： ステートレス・セッション Bean は、1 つ以上のリモートおよびローカル・コンポーネント・インタフェースに関連付けることができます。

EJB 2.1 クライアントへの EJB 3.0 ステートフル・セッション Bean の適用

EJB 3.0 ステートフル・セッション Bean を EJB 2.1 ホームおよびコンポーネント・インタフェースに関連付けることで（4-6 ページの「アノテーションの使用方法」を参照）、EJB 2.1 クライアントからアクセスできるように EJB 3.0 ステートフル・セッション Bean を調整できます。

この技術を使用することで、EJB 2.1 アプリケーションから EJB 3.0 への増分移行を管理することや、既存の EJB 2.1 クライアントから EJB 3.0 を使用して実装した新規開発機能にアクセスすることが可能になります。

EJB 2.1 ホームおよびコンポーネント・インタフェースの詳細は、次を参照してください。

- 11-7 ページの「ホーム・インタフェースの実装」
- 11-10 ページの「コンポーネント・インタフェースの実装」

アノテーションの使用方法

EJB 2.1 クライアントに EJB 3.0 ステートフル・セッション Bean を適用するには、次のようにします。

1. EJB 2.1 ホーム・インタフェースを EJB 3.0 ステートフル・セッション Bean に関連付けます。

   次のように、リモート・ホーム・インタフェースには `@RemoteHome` アノテーションを使用し、ローカル・ホーム・インタフェースには `@LocalHome` アノテーションを使用します。

```
@Stateful
@RemoteHome (value=Ejb21RemoteHome1.class)
@LocalHome (value=Ejb21LocalHome.class)
public class MyStatefulSB {
...
}
```

   注意： ステートフル・セッション Bean は、最大で 1 つのリモートおよびローカル・ホーム・インタフェースに関連付けることができます。

2. ホーム・インタフェースの `create` メソッドをサポートする要件を検討します。

   次のように、ホーム・インタフェースの `create<METHOD>` ごとに、EJB 3.0 ステートフル・セッション Bean に同じシグネチャ（引数の数、順序および型）を持つ初期化メソッドを実装し、そのメソッドに `@Init` アノテーションを付けます。

```
@Stateful
@RemoteHome (value=Ejb21RemoteHome1.class)
@LocalHome (value=Ejb21LocalHome.class)
public class MyStatefulSB {
    private String message;
    private String name;
...
    // Corresponds to home interface method create()

    @Init
    public void initDefault() throws CreateException {
        this.message = "Default Message";
        this.name = "Default Name";
    }

    // Corresponds to home interface method createWithMessage(String)

    @Init
    public void initWithMsg(String message) throws CreateException {
        this.message = message;
    }

    // Corresponds to home interface method createWithName(String)
    // Use @Init attribute value to disambiguate createWithName(String)
    // from createWithMessage(String).

    @Init(value="createWithName")
    public void initWithName(String message) throws CreateException {
        this.name = name;
    }

...
}
```

初期化メソッドには、任意のメソッド名を指定できます。OC4J は、シグネチャに基づいてホーム・インタフェースの create<METHOD> とステートフル・セッション Bean の初期化メソッドを照合します。別の方法として、@Init の属性 value を使用して明示的にホーム・インタフェースの create<METHOD> の名前を指定することも可能です。この方法は、ホーム・インタフェースの 2 つ以上の create<METHOD> メソッドに同じシグネチャが割り当てられている場合に役立ちます。

初期化メソッドは、post-construct ライフ・サイクル・メソッドが存在する場合、そのメソッドの起動後に起動されます（5-5 ページの「EJB 3.0 セッション Bean のライフ・サイクル・コールバック・インターセプタ・メソッドの構成」を参照）。

3. EJB 2.1 コンポーネント・インタフェースを EJB 3.0 ステートフル・セッション Bean に関連付けます。

   次のように、リモート・コンポーネント・インタフェースには @Remote アノテーションを使用し、ローカル・コンポーネント・インタフェースには @Local アノテーションを使用します。

```
@Stateful
@Remote (value={Ejb21Remote1.class, EJB21Remote2.class})
@Local (value={Ejb21Local.class})
public class MyStatefulSB {
...
}
```

注意： ステートフル・セッション Bean は、1 つ以上のリモートおよびローカル・コンポーネント・インタフェースに関連付けることができます。

5

EJB 3.0 セッション Bean の使用方法

この章では、EJB 3.0 セッション Bean を使用するために構成する必要のある様々なオプションについて説明します。

表 5-1 に、これらのオプションをリストし、基本オプション（ほとんどのアプリケーションに適用可能）であるか拡張オプション（より特殊なアプリケーションに適用可能）であるかを示します。また、ステートレス・セッション Bean に適用可能なオプションとステートフル・セッション Bean に適用可能なオプションも示します。

詳細は、次を参照してください。

- 1-30 ページの「セッション Bean とは」
- 第 4 章「EJB 3.0 セッション Bean の実装」

表 5-1 EJB 3.0 セッション Bean の構成オプション

オプション	ステートレス	ステートフル	タイプ
5-2 ページの「非アクティブ化の構成」		✔	拡張
5-2 ページの「非アクティブ化基準の構成」		✔	拡張
5-4 ページの「非アクティブ化の場所の構成」		✔	拡張
31-5 ページの「Bean インスタンスのプール・サイズの構成」	✔	✔	基本
31-7 ページの「セッション Bean の Bean インスタンス・プール・タイムアウトの構成」	✔	✔	拡張
21-8 ページの「セッション Bean のトランザクション・タイムアウトの構成」	✔	✔	拡張
5-5 ページの「EJB 3.0 セッション Bean のライフ・サイクル・コールバック・インターセプタ・メソッドの構成」	✔	✔	基本
5-6 ページの「EJB 3.0 セッション Bean のインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドの構成」	✔	✔	基本
5-8 ページの「EJB 3.0 セッション Bean の AroundInvoke インターセプタ・メソッドの構成」	✔	✔	拡張
5-9 ページの「EJB 3.0 セッション Bean のインターセプタ・クラスの AroundInvoke インターセプタ・メソッドの構成」	✔	✔	拡張
5-10 ページの「EJB 3.0 セッション Bean のインターセプタ・クラスの構成」	✔	✔	拡張
5-12 ページの「EJB 3.0 セッション Bean の OC4J 固有のデプロイ・オプションの構成」	✔	✔	拡張

非アクティブ化の構成

ステートフル・セッション Bean の非アクティブ化は、server.xml ファイルを使用して有効および無効にできます（5-2 ページの「デプロイ XML の使用方法」を参照）。

次のいずれかの理由で非アクティブ化を無効にすることを選択できます。

- 互換性のないオブジェクト・タイプ : 非アクティブ化でサポートされているオブジェクト・タイプ（1-35 ページの「非アクティブ化できるオブジェクト・タイプ」を参照）でステートフル・セッション Bean の非一時属性を表すことができない場合は、非アクティブ化を無効にすることにより、メモリー消費の増加と引換えに他のオブジェクト・タイプを使用できます。
- パフォーマンス : 非アクティブ化によりアプリケーションにパフォーマンスの問題が発生していると判断した場合は、非アクティブ化を無効にすることにより、メモリー消費の増加と引換えにパフォーマンスを改善できます。
- 2 次ストレージの制限 : 十分な 2 次ストレージを提供できない場合は（5-4 ページの「非アクティブ化の場所の構成」を参照）、非アクティブ化を無効にすることにより、メモリー消費の増加と引換えに 2 次ストレージ所要量を削減できます。

詳細は、次を参照してください。

- 1-35 ページの「ステートフル・セッション Bean の非アクティブ化が発生する状況」
- 5-2 ページの「非アクティブ化基準の構成」
- 5-4 ページの「非アクティブ化の場所の構成」

デプロイ XML の使用方法

EJB 3.0 ステートフル・セッション Bean では、EJB 2.1 ステートフル・セッション Bean と同様に、server.xml ファイルで非アクティブ化を構成します（12-2 ページの「デプロイ XML の使用方法」を参照）。

非アクティブ化基準の構成

OC4J が EJB 3.0 ステートフル・セッション Bean を非アクティブ化する条件は、OC4J 固有のアノテーション（5-3 ページの「アノテーションの使用方法」を参照）または orion-ejb-jar.xml ファイル（5-3 ページの「デプロイ XML の使用方法」を参照）を使用して指定できます。

orion-ejb-jar.xml ファイルの構成は、OC4J 固有のアノテーションを使用して設定された対応する構成をオーバーライドします。

詳細は、次を参照してください。

- 1-35 ページの「ステートフル・セッション Bean の非アクティブ化が発生する状況」
- 5-2 ページの「非アクティブ化の構成」
- 5-4 ページの「非アクティブ化の場所の構成」

アノテーションの使用方法

EJB 3.0 ステートフル・セッション Bean に対応する OC4J 固有のデプロイ・オプションは、OC4J 固有の `@StatefulDeployment` アノテーションを使用して指定できます。例 5-1 に、`@StatefulDeployment` アノテーションの次の属性を使用して EJB 3.0 ステートレス・セッション Bean の非アクティブ化基準を構成する方法を示します。

- `idletime`
- `memoryThreshold`
- `maxInstances`
- `maxInstancesThreshold`
- `passivateCount`
- `resourceCheckInterval`

これらの `@StatefulDeployment` の属性の詳細は、表 A-1 を参照してください。`@StatefulDeployment` アノテーションの詳細は、5-12 ページの「EJB 3.0 セッション Bean の OC4J 固有のデプロイ・オプションの構成」を参照してください。

例 5-1 @StatefulDeployment を使用した非アクティブ化基準の構成

```
import javax.ejb.Stateless;
import oracle.j2ee.ejb.StatelessDeployment;

@Stateless
@StatefulDeployment(
    idletime=100,
    memoryThroshold=90,
    maxInstances=10,
    maxInstancesThreshold=80,
    passivateCount=3,
    resourceCheckInterval=90
)
public class HelloWorldBean implements HelloWorld {
    public void sayHello(String name) {
        System.out.println("Hello "+name +" from first EJB3.0");
    }
}
```

デプロイ XML の使用方法

EJB 3.0 ステートフル・セッション Bean では、EJB 2.1 ステートフル・セッション Bean と同様に、`orion-ejb-jar.xml` ファイルで非アクティブ化基準を構成します（12-3 ページの「デプロイ XML の使用方法」を参照）。

非アクティブ化の場所の構成

非アクティブ化時に OC4J が EJB 3.0 ステートフル・セッション Bean をシリアライズするディレクトリおよびファイル名は、OC4J 固有のアノテーション（5-4 ページの「アノテーションの使用方法」を参照）または `orion-ejb-jar.xml` ファイル（5-4 ページの「デプロイ XML の使用方法」を参照）を使用して指定できます。

詳細は、次を参照してください。

- 1-36 ページの「非アクティブ化されたステートフル・セッション Bean の格納場所」
- 5-2 ページの「非アクティブ化の構成」
- 5-2 ページの「非アクティブ化基準の構成」

アノテーションの使用方法

EJB 3.0 ステートフル・セッション Bean に対応する OC4J 固有のデプロイ・オプションは、OC4J 固有の `@StatefulDeployment` アノテーションを使用して指定できます。例 5-1 に、`@StatefulDeployment` アノテーションの `persistenceFileName` 属性を使用して EJB 3.0 ステートレス・セッション Bean の非アクティブ化の場所を構成する方法を示します。

この `@StatefulDeployment` の属性の詳細は、表 A-1 を参照してください。`@StatefulDeployment` アノテーションの詳細は、5-12 ページの「EJB 3.0 セッション Bean の OC4J 固有のデプロイ・オプションの構成」を参照してください。

例 5-2 @StatefulDeployment を使用した非アクティブ化の場所の構成

```
import javax.ejb.Stateless;
import oracle.j2ee.ejb.StatelessDeployment;

@Stateless
@StatefulDeployment(
    persistenceFileNazme="C:¥sfsb¥sfsb.persistence",
)
public class HelloWorldBean implements HelloWorld {
    public void sayHello(String name) {
        System.out.println("Hello "+name +" from first EJB3.0");
    }
}
```

デプロイ XML の使用方法

EJB 3.0 ステートフル・セッション Bean では、EJB 2.1 ステートフル・セッション Bean と同様に、`orion-ejb-jar.xml` ファイルで非アクティブ化の場所を構成します（12-4 ページの「デプロイ XML の使用方法」を参照）。

EJB 3.0 セッション Bean のライフ・サイクル・コールバック・インターセプタ・メソッドの構成

EJB 3.0 セッション Bean クラス・メソッドを次のライフ・サイクル・イベントのコールバック・インターセプタ・メソッドとして指定できます（5-5 ページの「アノテーションの使用方法」を参照）。

- post-construct
- pre-destroy
- pre-passivate（ステートフル・セッション Bean のみ）
- post-activate（ステートフル・セッション Bean のみ）

注意： ステートレス・セッション Bean に対して pre-passivate または post-activate ライフ・サイクル・コールバック・メソッドを指定しないでください。

セッション Bean クラスのライフ・サイクル・コールバック・メソッドは、次のシグネチャを持つ必要があります。

```
void <METHOD>()
```

EJB 3.0 セッション Bean に関連付けるインターセプタ・クラスで 1 つ以上のライフ・サイクル・コールバック・メソッドを指定することもできます（5-6 ページの「EJB 3.0 セッション Bean のインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドの構成」を参照）。

詳細は、次を参照してください。

- 1-31 ページの「ステートレス・セッション Bean のライフ・サイクル」
- 1-33 ページの「ステートフル・セッション Bean のライフ・サイクル」
- 1-6 ページの「Bean クラスのライフ・サイクル・コールバック・メソッド」

アノテーションの使用方法

次のいずれかのアノテーションを使用して、EJB 3.0 セッション Bean クラス・メソッドをライフ・サイクル・コールバック・メソッドとして指定できます。

- `@PostConstruct`
- `@PreDestroy`
- `@PrePassivate`（ステートフル・セッション Bean のみ）
- `@PostActivate`（ステートフル・セッション Bean のみ）

例 5-3 に、`@PostConstruct` アノテーションを使用して EJB 3.0 ステートフル・セッション Bean のクラス・メソッド `initialize` をライフ・サイクル・コールバック・メソッドとして指定する方法を示します。

例 5-3 @PostConstruct

```
@Stateful
public class CartBean implements Cart {
    private ArrayList items;

    @PostConstruct
    public void initialize() {
        items = new ArrayList();
    }
...
}
```

EJB 3.0 セッション Bean のインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドの構成

EJB 3.0 セッション Bean のインターセプタ・クラスのインターセプタ・メソッドをライフ・サイクル・コールバック・インターセプタ・メソッドとして指定できます。

インターセプタ・クラスでライフ・サイクル・コールバック・インターセプタ・メソッドを構成するには、次のようにします。

1. インターセプタ・クラスを作成します。

   これは、任意の POJO クラスにすることができます。

2. ライフ・サイクル・コールバック・インターセプタ・メソッドを実装します。

   Bean のインターセプタ・クラスに定義するコールバック・メソッドには、次のシグネチャを割り当てます。

   ```
   Object <METHOD>(InvocationContext)
   ```

3. ライフ・サイクル・イベントをコールバック・インターセプタ・メソッドに関連付けます（5-7 ページの「アノテーションの使用方法」を参照）。

   1 つのライフ・サイクル・イベントは、1 つのコールバック・インターセプタ・メソッドにのみ関連付けることができますが、1 つのライフ・サイクル・コールバック・インターセプタ・メソッドは、複数のコールバック・イベントに割り込むために使用できます。たとえば、`@PostConstruct` と `@PreDestroy` は、インターセプタ・クラス内で 1 回のみ出現可能ですが、`@PostConstruct` と `@PreDestroy` の両方を同じコールバック・インターセプタ・メソッドに関連付けることができます。

4. インターセプタ・クラスを EJB 3.0 セッション Bean に関連付けます（5-10 ページの「EJB 3.0 セッション Bean のインターセプタ・クラスの構成」を参照）。

詳細は、次を参照してください。

- 1-31 ページの「ステートレス・セッション Bean のライフ・サイクル」
- 1-33 ページの「ステートフル・セッション Bean のライフ・サイクル」
- 1-6 ページの「EJB 3.0 インターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッド」

アノテーションの使用方法

次のいずれかのアノテーションを使用して、インターセプタ・クラス・メソッドを EJB 3.0 セッション Bean のライフ・サイクル・コールバック・メソッドとして指定できます。

- `@PostConstruct`
- `@PreDestroy`
- `@PrePassivate`（ステートフル・セッション Bean のみ）
- `@PostActivate`（ステートフル・セッション Bean のみ）

例 5-4 に、ステートフル・セッション Bean のインターセプタ・クラスを示します。この例では、`@PrePassivate` アノテーションを使用して、メソッド `myPrePassivateInterceptorMethod` を pre-passivate ライフ・サイクル・イベントのライフ・サイクル・コールバック・インターセプタ・メソッドとして指定しています。また、`@PostConstruct` および `@PostActivate` アノテーションを使用して、メソッド `myPostConstructInterceptorMethod` を post-construct と post-activate 両方のライフ・サイクル・イベントのライフ・サイクル・コールバック・インターセプタ・メソッドとして指定しています。OC4J は、適切なライフ・サイクル・イベントが発生した場合にのみ、対応するライフ・サイクル・メソッドを起動します。OC4J は、セッション Bean のビジネス・メソッドが起動されるたびに、ライフ・サイクル・インターセプタ・メソッド以外のすべてのインターセプタ・メソッド（`myInterceptorMethod` など）を起動します（5-10 ページの「EJB 3.0 セッション Bean のインターセプタ・クラスの構成」を参照）。

例 5-4 インターセプタ・クラス

```
public class MyStatefulSessionBeanInterceptor {
    ...
    protected void myInterceptorMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }

    @PostConstruct
    @PostActivate
    protected void myPostConstructInterceptorMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }
    @PrePassivate
    protected void myPrePassivateInterceptorMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }
}
```

EJB 3.0 セッション Bean の AroundInvoke インターセプタ・メソッドの構成

1 つの非ビジネス・メソッドをステートレスまたはステートフル・セッション Bean のインターセプタ・メソッドとして指定できます。クライアントがセッション Bean のビジネス・メソッドを起動するたびに、OC4J は起動をインターセプトし、インターセプタ・メソッドを起動します。クライアント起動は、インターセプタ・メソッドが `InvocationContext.proceed()` を返す場合にのみ続行されます。

インターセプタ・メソッドには次のシグネチャがあります。

```
Object <METHOD>(InvocationContext) throws Exception
```

インターセプタ・メソッドには、public、private、protected または package レベルのアクセスを割り当てることができますが、final または static として宣言することはできません。

このメソッドは、EJB 3.0 セッション Bean クラスに指定するか（5-8 ページの「アノテーションの使用方法」を参照）、EJB 3.0 セッション Bean に関連付けるインターセプタ・クラスに指定できます（5-9 ページの「EJB 3.0 セッション Bean のインターセプタ・クラスの AroundInvoke インターセプタ・メソッドの構成」を参照）。

詳細は、2-12 ページの「EJB 3.0 インターセプタについて」を参照してください。

アノテーションの使用方法

例 5-5 に、`@AroundInvoke` アノテーションを使用してセッション Bean クラスのメソッドをインターセプタ・メソッドとして指定する方法を示します。クライアントがこのステートレス・セッション Bean のビジネス・メソッドを起動するたびに、OC4J は起動をインターセプトし、インターセプタ・メソッド `myInterceptor` を起動します。クライアント起動は、インターセプタ・メソッドが `InvocationContext.proceed()` を返す場合にのみ続行されます。

例 5-5 EJB 3.0 セッション Bean の @AroundInvoke

```
@Stateless
public class HelloWorldBean implements HelloWorld  {
    public void sayHello()  {
        System.out.println("Hello!");
    }

    @AroundInvoke
    protected Object myInterceptor(InvocationContext ctx) throws Exception {
        Principal p = ctx.getEJBContext().getCallerPrincipal;
        if (!userIsValid(p)) {
            throw new SecurityException(
                "Caller: '" + p.getName() +
                "' does not have permissions for method " + ctx.getMethod()
            );
        }
        return ctx.proceed();
    }
}
```

EJB 3.0 セッション Bean のインターセプタ・クラスの AroundInvoke インターセプタ・メソッドの構成

1 つの非ビジネス・メソッドをステートレスまたはステートフル・セッション Bean のインターセプタ・メソッドとして指定できます。クライアントがセッション Bean のビジネス・メソッドを起動するたびに、OC4J は起動をインターセプトし、インターセプタ・メソッドを起動します。クライアント起動は、インターセプタ・メソッドが `InvocationContext.proceed()` を返す場合にのみ続行されます。

このメソッドは、EJB 3.0 セッション Bean に関連付けるインターセプタ・クラスに指定するか、EJB 3.0 セッション Bean クラスそれ自体に指定できます（5-8 ページの「EJB 3.0 セッション Bean の AroundInvoke インターセプタ・メソッドの構成」を参照）。

インターセプタ・クラスでインターセプタ・メソッドを構成するには、次のようにします。

1. インターセプタ・クラスを作成します。

   これは、任意の POJO クラスにすることができます。

2. インターセプタ・メソッドを実装します。

   インターセプタ・メソッドには次のシグネチャがあります。

   ```
   Object <METHOD>(InvocationContext) throws Exception
   ```

   インターセプタ・メソッドには、public、private、protected または package レベルのアクセスを割り当てることができますが、final または static として宣言することはできません。

3. メソッドをインターセプタ・メソッドとして指定します（5-9 ページの「アノテーションの使用方法」を参照）。

4. インターセプタ・クラスを EJB 3.0 セッション Bean に関連付けます（5-10 ページの「EJB 3.0 セッション Bean のインターセプタ・クラスの構成」を参照）。

詳細は、2-12 ページの「EJB 3.0 インターセプタについて」を参照してください。

アノテーションの使用方法

例 5-6 に、`@AroundInvoke` アノテーションを使用してインターセプタ・クラス・メソッドの `myInterceptor` を EJB 3.0 セッション Bean のインターセプタ・メソッドとして指定する方法を示します。このインターセプタ・クラスをセッション Bean に関連付けると（5-10 ページの「EJB 3.0 セッション Bean のインターセプタ・クラスの構成」を参照）、セッション Bean のビジネス・メソッドが起動するたびに、OC4J は起動をインターセプトし、`myInterceptor` メソッドを起動します。クライアント起動は、このメソッドが `InvocationContext.proceed()` を返す場合にのみ続行されます。

例 5-6 インターセプタ・クラス

```
public class MyInterceptor {
    ...
    @AroundInvoke
    protected Object myInterceptor(InvocationContext ctx) throws Exception {
        Principal p = ctx.getEJBContext().getCallerPrincipal;
        if (!userIsValid(p)) {
            throw new SecurityException(
                "Caller: '" + p.getName() +
                "' does not have permissions for method " + ctx.getMethod()
            );
        }
        return ctx.proceed();
    }

    @PreDestroy
    public void myPreDestroyMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
```

```
        ...
    }
}
```

EJB 3.0 セッション Bean のインターセプタ・クラスの構成

インターセプタ・クラスは、Bean クラスそれ自体とは異なる 1 つのクラスであり、そのメソッドは Bean のビジネス・メソッドの起動およびライフ・サイクル・イベントの発生に応じて起動されます。Bean クラスは、任意の数のインターセプタ・クラスに関連付けることができます。

インターセプタ・クラスは、EJB 3.0 ステートレスまたはステートフル・セッション Bean に関連付けることができます。

インターセプタ・クラスを使用して EJB 3.0 セッション Bean を構成するには、次のようにします。

1. インターセプタ・クラスを作成します（5-11 ページの「インターセプタ・クラスの作成」を参照）。

   これは、任意の POJO クラスにすることができます。

2. インターセプタ・クラスにインターセプタ・メソッドを実装します。

   インターセプタ・メソッドには次のシグネチャがあります。

   ```
   Object <METHOD>(InvocationContext) throws Exception
   ```

   インターセプタ・メソッドには、public、private、protected または package レベルのアクセスを割り当てることができますが、final または static として宣言することはできません。

   インターセプタ・メソッドには、ライフ・サイクル・コールバックとして（5-6 ページの「EJB 3.0 セッション Bean のインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドの構成」を参照）、または `AroundInvoke` メソッドとして（5-9 ページの「EJB 3.0 セッション Bean のインターセプタ・クラスの AroundInvoke インターセプタ・メソッドの構成」を参照）アノテーションを付けることができます。

3. インターセプタ・クラスを EJB 3.0 セッション Bean に関連付けます（5-11 ページの「インターセプタ・クラスとセッション Bean との関連付け」を参照）。

4. オプションで、シングルトン・インターセプタを使用するようセッション Bean を構成します（5-12 ページの「セッション Bean でのシングルトン・インターセプタの指定」を参照）。

詳細は、2-12 ページの「EJB 3.0 インターセプタについて」を参照してください。

アノテーションの使用方法

この項の内容は次のとおりです。

- インターセプタ・クラスの作成
- インターセプタ・クラスとセッション Bean との関連付け
- セッション Bean でのシングルトン・インターセプタの指定

インターセプタ・クラスの作成

例 5-7 に、EJB 3.0 セッション Bean のインターセプタ・クラスに `AroundInvoke` インターセプタ・メソッドおよびライフ・サイクル・コールバック・インターセプタ・メソッドを指定する方法を示します。このインターセプタ・クラスをセッション Bean に関連付けると（例 5-8 を参照）、セッション Bean のビジネス・メソッドが起動するたびに、OC4J は `AroundInvoke` メソッドの `myInterceptor` を起動します。適切なライフ・サイクル・イベントが発生すると、OC4J は `myPreDestroyMethod` などの対応するライフ・サイクル・コールバック・インターセプタ・メソッドを起動します。

例 5-7 インターセプタ・クラス

```
public class MyInterceptor {
    ...
    @AroundInvoke
    protected Object myInterceptor(InvocationContext ctx) throws Exception {
        Principal p = ctx.getEJBContext().getCallerPrincipal;
        if (!userIsValid(p)) {
            throw new SecurityException(
                "Caller: '" + p.getName() +
                "' does not have permissions for method " + ctx.getMethod()
            );
        }
        return ctx.proceed();
    }

    @PreDestroy
    public void myPreDestroyMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }
}
```

インターセプタ・クラスとセッション Bean との関連付け

インターセプタ・クラスは、`@Interceptors` アノテーションを使用して EJB 3.0 セッション Bean に関連付けることができます。例 5-8 に、例 5-7 のインターセプタ・クラスを EJB 3.0 セッション Bean クラスに関連付ける方法を示します。

`@PostConstruct` のライフ・サイクル・メソッドは、EJB 3.0 セッション Bean クラスそれ自体のメソッドですが（5-5 ページの「EJB 3.0 セッション Bean のライフ・サイクル・コールバック・インターセプタ・メソッドの構成」を参照）、`@PreDestroy` のライフ・サイクル・メソッドは、このセッション Bean に関連付けられたインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドです（5-6 ページの「EJB 3.0 セッション Bean のインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドの構成」を参照）。

例 5-8 インターセプタ・クラスと EJB 3.0 セッション Bean の関連付け

```
@Stateful
@Interceptors(MyInterceptor.class)
public class CartBean implements Cart {
    private ArrayList items;
```

```
    @PostConstruct
    public void initialize() {
        items = new ArrayList();
    }
    ...
}
```

セッション Bean でのシングルトン・インターセプタの指定

例 5-9 に示すように、@StatelessDeployment または @StatefulDeployment の属性 interceptorType を singleton に設定することで、シングルトン・インターセプタ・クラスを使用するよう OC4J を構成できます。このセッション Bean のすべてのインスタンスは、MyInterceptor の同じインスタンスを共有します。MyInterceptor クラスは、ステートレスである必要があります。

この属性の詳細は、表 A-1 を参照してください。シングルトン・インターセプタの詳細は、2-14 ページの「シングルトン・インターセプタ」を参照してください。

例 5-9 EJB 3.0 ステートフル・セッション Bean でのシングルトン・インターセプタ・クラスの指定

```
@Stateful
@StatefulDeployment(interceptorType="singleton")
@Interceptors(MyInterceptor.class)
public class CartBean implements Cart {
    private ArrayList items;

    @PostConstruct
    public void initialize() {
        items = new ArrayList();
    }
    ...
}
```

EJB 3.0 セッション Bean の OC4J 固有のデプロイ・オプションの構成

EJB 3.0 セッション Bean の OC4J 固有のデプロイ・オプションは、OC4J 固有のアノテーション（5-13 ページの「アノテーションの使用方法」を参照）または orion-ejb-jar.xml ファイル（5-14 ページの「デプロイ XML の使用方法」を参照）を使用して構成できます。

orion-ejb-jar.xml ファイルの構成は、OC4J 固有のアノテーションを使用して設定された対応する構成をオーバーライドします。

アノテーションの使用方法

EJB 3.0 セッション Bean の OC4J 固有のデプロイ・オプションは、次の OC4J 固有のアノテーションを使用して指定できます。

- `@StatelessDeployment`: ステートレス・セッション Bean 用。
- `@StatefulDeployment`: ステートフル・セッション Bean 用。

例 5-10 に、`@StatelessDeployment` アノテーションを使用して EJB 3.0 ステートレス・セッション Bean の OC4J 固有のデプロイ・オプションを構成する方法を示します。

`@StatelessDeployment` の属性の詳細は、表 A-1 を参照してください。

例 5-10 @StatelessDeployment

```
import javax.ejb.Stateless;
import oracle.j2ee.ejb.StatelessDeployment;

@Stateless
@StatelessDeployment(
    minInstances=5,
    poolCacheTimeout=90
)
public class HelloWorldBean implements HelloWorld {
    public void sayHello(String name) {
        System.out.println("Hello "+name +" from first EJB3.0");
    }
}
```

例 5-11 に、`@StatefulDeployment` アノテーションを使用して EJB 3.0 ステートフル・セッション Bean の OC4J 固有のデプロイ・オプションを構成する方法を示します。

`@StatefulDeployment` の属性の詳細は、表 A-1 を参照してください。

例 5-11 @StatefulDeployment

```
import javax.ejb.Stateful
import oracle.j2ee.ejb.StatefulDeployment;

@Stateful
@StatefulDeployment(
    idletime=100
    passivateCount=3
)
public class CartBean implements Cart {
    private ArrayList items;
    ...
}
```

デプロイ XML の使用方法

例 5-12 に示すように、OC4J 固有のデプロイ・オプションは、orion-ejb-jar.xml ファイルの <session-deployment> 要素を使用して指定できます。

<session-deployment> 要素の詳細は、A-5 ページの「<session-deployment>」を参照してください。

例 5-12 orion-ejb-jar.xml ファイルの <session-deployment> 要素

```
<?xml version="1.0" encoding="utf-8"?>
<orion-ejb-jar
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion-ejb-jar-10_
0.xsd"
    deployment-version="10.1.3.1.0"
    deployment-time="10b1fb5cdd0"
    schema-major-version="10"
    schema-minor-version="0"
>
    <enterprise-beans>
        <session-deployment
            name="MBeanServerEjb"
            call-timeout="0"
            location="MBeanServerEjb"
            local-location="admin_ejb_MBeanServerEjbLocal"
            timeout="0"
            ...
        >
        </session-deployment>
    ...
    </enterprise-beans>
    ...
</orion-ejb-jar>
```

第 III 部

JPA エンティティ

第 III 部では、JPA エンティティおよび JPA エンティティ問合せの実装および構成の手順に関する情報を示します。概念的な情報は、第 I 部「EJB の概要」を参照してください。

第 III 部は次の各章で構成されています。

- 第 6 章「JPA エンティティの実装」
- 第 7 章「Java 永続性 API の使用方法」
- 第 8 章「JPA 問合せの実装」

6

JPA エンティティの実装

この章では、JPA エンティティの実装方法を説明します。

詳細は、次を参照してください。

- 1-38 ページの「JPA エンティティとは」
- 第 7 章「Java 永続性 API の使用方法」

JPA エンティティの実装

EJB 3.0 では、Enterprise Bean の開発が大幅に単純化され、多くの複雑な開発タスクが排除されています。次に例を示します。

- Bean クラスは、POJO にすることができます。`javax.ejb.EntityBean` を実装する必要はありません。
- ビジネス・インタフェースはオプションです。このインタフェースは、Plain Old Java Interface（POJI）にすることができます。

  ホーム（`javax.ejb.EJBHome` および `javax.ejb.EJBLocalHome`）およびコンポーネント（`javax.ejb.EJBObject` および `javax.ejb.EJBLocalObject`）ビジネス・インタフェースは不要です。
- アノテーションは、コンテナ管理の関連性（オブジェクト・リレーショナル・マッピング）などの多くの機能に使用されます。
- `EntityContext` は不要です。単純に `this` を使用してエンティティを自身に関連付けできます。

詳細は、1-38 ページの「JPA エンティティとは」を参照してください。

注意： JPA エンティティのコード例は、`http://www.oracle.com/technology/tech/java/oc4j/10131/how_to/how-to-ejb30-entity-ejb/doc/how-to-ejb30-entity-ejb.html` からダウンロードできます。

JPA エンティティを実装するには、次のようにします。

1. エンティティ Bean クラスを作成します。

   POJO を作成し、`@Entity` アノテーションを使用してそれをコンテナ管理の永続性を備えたエンティティ Bean として定義できます。

   `@Transient` アノテーションを付けないかぎり、すべてのデータ・メンバーはデフォルトでコンテナ管理の永続性フィールドとみなされます。

2. `@Table` および `@Column` アノテーションを使用して、OC4J がエンティティ Bean クラスをデータベースに維持する方法を定義します。

   既存のデータベース・スキーマがない場合は、これらのアノテーションを省略して、表および列の定義を OC4J に委任できます。デプロイ時に、OC4J はクラス名およびデータ・メンバー名に基づいてデフォルトの表名および列名を作成します。

   詳細は、7-7 ページの「表および列情報の構成」を参照してください。

3. `@Id` アノテーションを使用して、1 つのデータ・メンバーを主キー・フィールドとして定義します。

   データ・メンバー自体またはその getter メソッドにアノテーションを付けることができます。詳細は、7-2 ページの「JPA エンティティの主キーの構成」を参照してください。

4. `@OneToOne` などの適切なオブジェクト・リレーショナル・マッピング・アノテーションを使用して、コンテナ管理の関連性を定義します。

   詳細は、7-10 ページの「JPA エンティティのコンテナ管理の関連性フィールドの構成」を参照してください。

5. オプションで、`@NamedQuery` アノテーションを使用して finder および問合せを定義します。

   実行時に、TopLink 永続性マネージャが提供する事前定義の finder（1-56 ページの「事前定義の TopLink finder」を参照）およびデフォルトの finder（1-57 ページの「デフォルトの TopLink finder」を参照）を使用できます。

   詳細は、第 8 章「JPA 問合せの実装」を参照してください。

6. オプションで、適切なアノテーションを使用してライフ・サイクル・コールバック・メソッドを定義します。

   ライフ・サイクル・メソッドを定義する必要はありません。このようなメソッドの実装はすべて OC4J に用意されています。エンティティ Bean のライフ・サイクルの特定の時点で独自のアクションを実行する場合にのみ、エンティティ Bean クラスのメソッドをライフ・サイクル・コールバック・メソッドとして定義します。

   詳細は、7-18 ページの「JPA エンティティのライフ・サイクル・コールバック・メソッドの構成」を参照してください。

7. エンティティ Bean の構成を完了します（第 7 章「Java 永続性 API の使用方法」を参照）。

7

Java 永続性 API の使用方法

この章では、JPA エンティティを使用するために構成できる様々なオプションについて説明します。

表 7-1 に、これらのオプションをリストし、基本オプション（ほとんどのアプリケーションに適用可能）であるか拡張オプション（より特殊なアプリケーションに適用可能）であるかを示します。

詳細は、次を参照してください。

- 1-38 ページの「JPA エンティティとは」
- 6-2 ページの「JPA エンティティの実装」

表 7-1 JPA エンティティの構成オプション

オプション	タイプ
7-2 ページの「JPA エンティティの主キーの構成」	基本
7-7 ページの「表および列情報の構成」	基本
7-10 ページの「JPA エンティティのコンテナ管理の関連性フィールドの構成」	基本
7-11 ページの「基本マッピングの構成」	基本
7-11 ページの「ラージ・オブジェクト・マッピングの構成」	拡張
7-12 ページの「シリアライズ・オブジェクト・マッピングの構成」	拡張
7-12 ページの「1 対 1 マッピングの構成」	基本
7-13 ページの「多対 1 マッピングの構成」	基本
7-14 ページの「1 対多マッピングの構成」	基本
7-14 ページの「多対多マッピングの構成」	基本
7-15 ページの「集約マッピングの構成」	拡張
7-17 ページの「オプティミスティック・ロック・バージョン・フィールドの構成」	拡張
第 8 章「JPA 問合せの実装」	基本
7-20 ページの「JPA エンティティの継承の構成」	拡張
7-17 ページの「遅延ロードの構成」	基本
7-18 ページの「JPA エンティティのライフ・サイクル・コールバック・メソッドの構成」	拡張
7-19 ページの「JPA エンティティのエンティティ・リスナー・クラスのライフ・サイクル・コールバック・リスナー・メソッドの構成」	拡張

JPA エンティティの主キーの構成

すべての JPA エンティティには主キーが必要です。

主キーは単一のプリミティブ型または JDK オブジェクト型のエンティティ・フィールドとして指定できます（7-2 ページの「JPA エンティティの単純な主キー・フィールドの構成」を参照）。

個別のコンポジット主キー・クラスを使用して、1 つ以上のプリミティブ型または JDK オブジェクト型から構成されるコンポジット主キーを指定できます（7-3 ページの「JPA エンティティのコンポジット主キー・クラスの構成」を参照）。

主キー値を自分で割り当てるか、主キー値ジェネレータに主キー・フィールドを関連付けることができます（7-5 ページの「JPA エンティティの自動主キー生成の構成」を参照）。

JPA エンティティの単純な主キー・フィールドの構成

最も単純な主キーは、単一のプリミティブ型または JDK オブジェクト型のエンティティ・フィールドとして指定する主キーです（7-2 ページの「アノテーションの使用方法」を参照）。

注意： JPA エンティティの主キー・フィールドのコード例は、`http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#id` を参照してください。

アノテーションの使用方法

例 7-1 に、`@Id` アノテーションを使用してエンティティ・フィールドを主キーとして指定する方法を示します。この例では、主キー値は表ジェネレータを使用して生成されています（7-5 ページの「JPA エンティティの自動主キー生成の構成」を参照）。

例 7-1 @Id を使用した主キー

```
@Id(generate=TABLE, generator="ADDRESS_TABLE_GENERATOR")
@TableGenerator(
    name="ADDRESS_TABLE_GENERATOR",
    tableName="EMPLOYEE_GENERATOR_TABLE",
    pkColumnValue="ADDRESS_SEQ"
)
@Column(name="ADDRESS_ID")
public Integer getId() {
    return id;
}
```

JPA エンティティのコンポジット主キー・クラスの構成

コンポジット主キーは、通常、2 つ以上のプリミティブ型または JDK オブジェクト型から構成されます。一般的に、コンポジット主キーが割り当てられるのは、データベース・キーが複数の列から構成されるレガシー・データベースからマッピングを行う場合です。このようなコンポジット主キーは、個別のコンポジット主キー・クラスを使用して指定できます（7-3 ページの「アノテーションの使用方法」を参照）。

コンポジット主キー・クラスには、次の特性があります。

- POJO クラスです。
- public クラスであり、引数のない public コンストラクタを保持する必要があります。
- プロパティベースのアクセスを使用する場合、主キー・クラスのプロパティは、public または protected である必要があります。
- シリアライズ可能である必要があります。
- `equals` および `hashCode` メソッドを定義する必要があります。

  これらのメソッドにおける値の等価性のセマンティクスは、キーがマップされるデータベース・タイプのデータベースの等価性と一致している必要があります。

コンポジット主キー・クラスは、エンティティ・クラスに所有される埋込みクラスとするか、エンティティ・クラスの複数のフィールドまたはプロパティにマップされるフィールドを持つ非埋込みクラスとすることが可能です。非埋込みクラスの場合、コンポジット主キー・クラスの主キー・フィールドまたはプロパティの名前とエンティティ・クラスの主キー・フィールドまたはプロパティの名前は一致する必要があり、それらの型も同じである必要があります。

アノテーションの使用方法

例 7-2 に、典型的な埋込み可能コンポジット主キー・クラスを示します。また、例 7-3 に、`@EmbeddedId` アノテーションを使用してこの埋込みコンポジット主キー・クラスで JPA エンティティを構成する方法を示します。

例 7-2 埋込み可能コンポジット主キー・クラス

```
@Embeddable
public class EmployeePK implements Serializable {
    private String name;
    private long id;

    public EmployeePK() {
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public int hashCode() {
        return (int) name.hashCode() + id;
    }

    public boolean equals(Object obj) {
```

```
        if (obj == this) return true;
        if (!(obj instanceof EmployeePK)) return false;
        if (obj == null) return false;
        EmployeePK pk = (EmployeePK) obj;
        return pk.id == id && pk.name.equals(name);
    }
}
```

例 7-3 埋込みコンポジット主キー・クラスを使用した JPA エンティティ

```
@Entity
public class Employee implements Serializable {
    EmployeePK primaryKey;

    public Employee() {
    }

    @EmbeddedId
    public EmployeePK getPrimaryKey() {
        return primaryKey;
    }

    public void setPrimaryKey(EmployeePK pk) {
        primaryKey = pk;
    }

    ...
}
```

例 7-4 に、非埋込みコンポジット主キー・クラスを示します。このクラスの empName および birthDay フィールドは、その名前と型がエンティティ・クラスのプロパティに一致している必要があります。例 7-5 に、@IdClass アノテーションを使用してこの非埋込みコンポジット主キー・クラスで JPA エンティティを構成する方法を示します。エンティティ・クラスの empName および birthDay フィールドは主キーで使用されるため、それらに @Id アノテーションを付ける必要もあります。

例 7-4 非埋込みコンポジット主キー・クラス

```
public class EmployeePK implements Serializable {
    private String empName;
    private Date birthDay;

    public EmployeePK() {
    }

    public String getName() {
        return empName;
    }

    public void setName(String name) {
        empName = name;
    }

    public long getDateOfBirth() {
        return birthDay;
    }

    public void setDateOfBirth(Date date) {
        birthDay = date;
    }

    public int hashCode() {
        return (int) empName.hashCode();
    }
```

```
    public boolean equals(Object obj) {
        if (obj == this) return true;
        if (!(obj instanceof EmployeePK)) return false;
        if (obj == null) return false;
        EmployeePK pk = (EmployeePK) obj;
        return pk.birthDay == birthDay && pk.empName.equals(empName);
    }
}
```

例 7-5 マップ済のコンポジット主キー・クラスを使用した JPA エンティティ

```
@IdClass(EmployeePK.class)
@Entity
public class Employee {
    @Id String empName;
    @Id Date birthDay;
...
}
```

JPA エンティティの自動主キー生成の構成

通常は、主キー・フィールド（「JPA エンティティの単純な主キー・フィールドの構成」を参照）を主キー値ジェネレータに関連付けて、エンティティ・インスタンスが作成されたときに新しい一意の主キー値が自動的に割り当てられるようにします。

表 7-2 に、定義できる主キー値ジェネレータのタイプをリストします。

表 7-2 JPA エンティティの主キー値ジェネレータ

タイプ	説明	参照先
生成された ID 表	エンティティに対して生成された主キー値をコンテナが格納するために使用するデータベース表。通常は、表ベースの主キー生成を使用する複数のエンティティ・タイプにより共有されます。各エンティティ・タイプは、通常は表内の独自の行を使用して、そのエンティティ・クラスの主キー値を生成します。主キー値は正の整数です。	『Oracle TopLink 開発者ガイド』の表の順序付けに関する項
表ジェネレータ	名前で参照できる主キー・ジェネレータであり、パッケージ、クラス、メソッドまたはフィールド・レベルで定義されます。定義するレベルは、目的の可視性およびジェネレータの共有に依存します。有効範囲決定規則または可視性規則は実際には強制されません。使用されるレベルでジェネレータを定義することをお薦めします。 このジェネレータは、データベース表に基づいています。	『Oracle TopLink 開発者ガイド』の表の順序付けに関する項
シーケンス・ジェネレータ	名前で参照できる主キー・ジェネレータであり、パッケージ、クラス、メソッドまたはフィールド・レベルで定義されます。定義するレベルは、目的の可視性およびジェネレータの共有に依存します。有効範囲決定規則または可視性規則は実際には強制されません。使用されるレベルでジェネレータを定義することをお薦めします。 このジェネレータは、データベース・サーバーが提供する順序オブジェクトに基づいています。	『Oracle TopLink 開発者ガイド』の Oracle データベース・プラットフォームでのネイティブ順序付けに関する項 『Oracle TopLink 開発者ガイド』の Oracle データベース・プラットフォーム以外でのネイティブ順序付けに関する項

注意： EJB 3.0 自動主キー生成のコード例は、`http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#sequencing` を参照してください。

アノテーションの使用方法

例 7-6 に、@TableGenerator アノテーションを使用してデータベース表に基づいて主キー値ジェネレータを指定する方法を示します。TopLink JPA 永続性プロバイダは、デプロイ時にこの表の作成を試行します。JPA 永続性プロバイダで作成できない場合は、データベース・ドキュメントに従って、デプロイ前にこの表が存在していることを確認する必要があります。Address の新規インスタンスが作成されると、エンティティ・フィールド id の新しい値が ADDRESS_GENERATOR_TABLE から取得されます。この場合は、@GeneratedValue アノテーションの属性 strategy を TABLE に設定し、generator を ADDRESS_TABLE_GENERATOR に設定する必要があります。

例 7-6 GeneratedValue の strategy の表 : @TableGenerator

```
@Entity
@Table(name="EJB_ADDRESS")
public class Address implements Serializable {
...
    @TableGenerator(
        name="ADDRESS_TABLE_GENERATOR",
        tableName="ADDRESS_GENERATOR_TABLE",
        pkColumnValue="ADDRESS_SEQ"
    )
    @Id @GeneratedValue(strategy="TABLE", generator="ADDRESS_TABLE_GENERATOR")
    @Column(name="ADDRESS_ID")
    public Integer getId() {
        return id;
    }
...
}
```

例 7-7 に、@SequenceGenerator アノテーションを使用して、データベースにより提供される順序オブジェクトに基づいて主キー値ジェネレータを指定する方法を示します。TopLink JPA 永続性プロバイダは、デプロイ時にこのオブジェクトの作成を試行します。JPA 永続性プロバイダで作成できない場合は、データベース・ドキュメントに従って、デプロイ前にこの順序オブジェクトが存在していることを確認する必要があります。Address の新規インスタンスが作成されると、エンティティ・フィールド id の新しい値がデータベース順序オブジェクト ADDRESS_SEQ から取得されます。この場合は、@GeneratedValue アノテーションの属性 strategy を SEQUENCE に設定し、generator を ADDRESS_SEQUENCE_GENERATOR に設定する必要があります。

例 7-7 GeneratedValue の strategy の順序 : @SequenceGenerator

```
@Entity
@Table(name="EJB_ADDRESS")
public class Address implements Serializable {
...
    @SequenceGenerator(
        name="ADDRESS_SEQUENCE_GENERATOR",
        sequenceName="ADDRESS_SEQ"
    )
    @Id @GeneratedValue(strategy="SEQUENCE", generator="ADDRESS_SEQUENCE_GENERATOR")
    @Column(name="ADDRESS_ID")
    public Integer getId() {
        return id;
    }
...
}
```

例 7-8 に、@GeneratedValue アノテーションを使用して主キー ID 列（自動番号列）に基づいて主キー値ジェネレータを指定する方法を示します。Address の新規インスタンスが保存されている場合は、データベースによって値が ID 列に割り当てられます。この場合、TopLink JPA 永続性プロバイダによって、挿入済の行が再び読み取られ、id がこの値に設定されるようメモリー内の Address エンティティが更新されます。

例 7-8 @GeneratedValue の strategy の ID

```
@Entity
@Table(name="EJB_ADDRESS")
public class Address implements Serializable {
...
    @Id @GeneratedValue(strategy="IDENTITY")
    public Integer getId() {
        return id;
    }
...
}
```

表および列情報の構成

次のように、TopLink エンティティ・マネージャがエンティティを維持するデータベース表の特性を定義できます。

- プライマリ表の構成
- セカンダリ表の構成
- 列の構成
- 結合列の構成

これは、既存のデータベース・スキーマがある場合に特に重要です。

既存のデータベース・スキーマがない場合は、この構成を省略して、表および列の定義を OC4J に委任できます。デプロイ時に、OC4J はクラス名およびデータ・メンバー名に基づいてデフォルトの表名および列名を作成します。

注意： JPA エンティティ表および列のコード例は、http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html からダウンロードできます。

プライマリ表の構成

プライマリ表は、TopLink エンティティ・マネージャがエンティティを維持する表です。特に、エンティティの主キーを格納する表です（7-2 ページの「JPA エンティティの主キーの構成」を参照）。オプションで、エンティティの永続データが複数の表に格納されている場合は、1 つ以上のセカンダリ表（7-8 ページの「セカンダリ表の構成」を参照）も指定できます。

プライマリ表はエンティティ・クラス・レベルで定義します。

アノテーションの使用方法

例 7-9 に、`@Table` アノテーションを使用して `Employee` クラスのプライマリ表を定義する方法を示します。TopLink エンティティ・マネージャは、このエンティティのインスタンスを `EJB_EMPLOYEE` という名前の表に維持します。

例 7-9 @Table

```
@Entity
@Table(name="EJB_EMPLOYEE")
public class Employee implements Serializable {
...
}
```

セカンダリ表の構成

1 つ以上のセカンダリ表を指定して、エンティティの永続データが複数の表に格納されていることを示します。セカンダリ表を指定する前に、まずプライマリ表（7-8 ページの「プライマリ表の構成」を参照）を指定する必要があります。

セカンダリ表はエンティティ・クラス・レベルで定義します。

1 つ以上のセカンダリ表を指定する場合は、その表に格納される永続フィールドの列定義（7-8 ページの「列の構成」を参照）にセカンダリ表名を指定できます。これにより、エンティティ永続フィールドを複数の表に分散できます。

アノテーションの使用方法

例 7-10 に、`@SecondaryTable` アノテーションを使用して、エンティティの永続データの一部が `EJB_SALARY` という名前の表に格納されることを指定する方法を示します。

例 7-10 @SecondaryTable

```
@Entity
@Table(name="EJB_EMPLOYEE")
@SecondaryTable(name="EJB_SALARY")
public class Employee implements Serializable {
...
}
```

列の構成

デフォルトでは、列は TopLink エンティティ・マネージャがフィールドの値を格納するプライマリ表（7-8 ページの「プライマリ表の構成」を参照）の列の名前です。

プロパティの 1 つ（getter または setter メソッド）またはエンティティのフィールド・レベルで列を定義します。

1 つ以上のセカンダリ表を指定した場合（7-8 ページの「セカンダリ表の構成」を参照）は、列定義にセカンダリ表名を指定できます。これにより、エンティティ永続フィールドを複数の表に分散できます。

アノテーションの使用方法

例 7-11 に、@Column アノテーションを使用してフィールド firstName のプライマリ表の列 F_NAME を指定する方法を示します。

例 7-11 プライマリ表の @Column

```
@Column(name="F_NAME")
public String getFirstName() {
    return firstName;
}
```

例 7-12 に、@Column アノテーションを使用してフィールド salary のセカンダリ表 EMP_SALARY の列 SALARY を指定する方法を示します。

例 7-12 セカンダリ表の @Column

```
@Column(name="SALARY", secondaryTable="EMP_SALARY")
public String getSalary() {
    return salary;
}
```

結合列の構成

結合列では、エンティティ・アソシエーションまたはセカンダリ表を結合するためのマッピングされた外部キー列を指定します。

結合列は、次のもので定義できます。

- セカンダリ表（例 7-13 を参照）
- 1 対 1 マッピング（例 7-14 を参照）
- 多対 1 マッピング（例 7-15 を参照）
- 1 対多マッピング（例 7-16 を参照）

アノテーションの使用方法

例 7-13 に、@JoinColumn アノテーションを使用してセカンダリ表で結合列を指定する方法を示します。詳細は、7-8 ページの「セカンダリ表の構成」を参照してください。

例 7-13 セカンダリ表での @JoinColumn

```
@Entity
@Table(name="EJB_EMPLOYEE")
@SecondaryTable(name="EJB_SALARY")
@JoinColumn(name="EMP_ID", referencedColumnName="EMP_ID")
public class Employee implements Serializable {
...
}
```

例 7-14 に、@JoinColumn アノテーションを使用して 1 対 1 マッピングで結合列を指定する方法を示します。詳細は、7-12 ページの「1 対 1 マッピングの構成」を参照してください。

例 7-14 1 対 1 マッピングでの @JoinColumn

```
@OneToOne(cascade=ALL, fetch=LAZY)
@JoinColumn(name="ADDR_ID")
public Address getAddress() {
    return address;
}
```

例 7-15 に、@JoinColumn アノテーションを使用して多対 1 マッピングで結合列を指定する方法を示します。詳細は、7-13 ページの「多対 1 マッピングの構成」を参照してください。

例 7-15 多対 1 マッピングでの @JoinColumn

```
@ManyToOne(cascade=PERSIST, fetch=LAZY)
@JoinColumn(name="MANAGER_ID", referencedColumnName="EMP_ID")
public Employee getManager() {
    return manager;
}
```

例 7-16 に、@JoinColumn アノテーションを使用して 1 対多マッピングで結合列を指定する方法を示します。詳細は、7-14 ページの「1 対多マッピングの構成」を参照してください。

例 7-16 1 対多マッピングでの @JoinColumn

```
@OneToMany(cascade=PERSIST)
@JoinColumn(name="MANAGER_ID", referencedColumnName="EMP_ID")
public Collection getManagedEmployees() {
    return managedEmployees;
}
```

JPA エンティティのコンテナ管理の関連性フィールドの構成

JPA エンティティでは、次のようにコンテナ管理の関連性（CMR）フィールド（1-39 ページの「JPA エンティティのコンテナ管理の関連性フィールドとは」を参照）を定義します。

- 7-11 ページの「基本マッピングの構成」
- 7-11 ページの「ラージ・オブジェクト・マッピングの構成」
- 7-12 ページの「シリアライズ・オブジェクト・マッピングの構成」
- 7-12 ページの「1 対 1 マッピングの構成」
- 7-13 ページの「多対 1 マッピングの構成」
- 7-14 ページの「1 対多マッピングの構成」
- 7-14 ページの「多対多マッピングの構成」

注意： JPA エンティティのコンテナ管理の関連性フィールドのコード例は、http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html からダウンロードできます。

基本マッピングの構成

基本マッピングを使用して、プリミティブまたは JDK オブジェクト値を含むフィールドをマッピングします。たとえば、基本マッピングを使用して、`String` 属性を `VARCHAR` 列に格納します。

プロパティの 1 つ（getter または setter メソッド）またはエンティティのフィールド・レベルで基本マッピングを定義します。

オプションで、データベースからデータをフェッチする方針を定義できます（7-17 ページの「遅延ロードの構成」を参照）。

詳細は、『Oracle TopLink 開発者ガイド』のフィールドへの直接マッピングの理解に関する項を参照してください。

注意： EJB 3.0 基本マッピングのコード例は、`http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#basic` を参照してください。

アノテーションの使用方法

例 7-17 に、`@Basic` アノテーションを使用してフィールド `firstName` の基本マッピングを指定する方法を示します。

例 7-17 @Basic

```
@Basic()
@Column(name="F_NAME")
public String getFirstName() {
    return firstName;
}
```

ラージ・オブジェクト・マッピングの構成

ラージ・オブジェクト（LOB）マッピングを使用して、永続プロパティまたはフィールドをデータベースでサポートされている LOB タイプに LOB として維持することを指定します。LOB には、バイナリ（BLOB）型または文字（CLOB）型があります。

プロパティの 1 つ（getter または setter メソッド）またはエンティティのフィールド・レベルでラージ・オブジェクト・マッピングを定義します。

詳細は、次を参照してください。

- 『Oracle TopLink 開発者ガイド』のフィールドへの直接マッピングの理解に関する項
- 『Oracle TopLink 開発者ガイド』のコンバータ・マッピングの使用に関する項
- 『Oracle TopLink 開発者ガイド』のタイプ変換コンバータに関する項

アノテーションの使用方法

例 7-18 に、@Lob アノテーションを使用してフィールド image のラージ・オブジェクト・マッピングを指定する方法を示します。

例 7-18 @Lob

```
@Lob(fetch=EAGER, type=BLOB)
@Column(name="IMAGE")
public Byte[] getImage() {
    return image;
}
```

シリアライズ・オブジェクト・マッピングの構成

シリアライズ・オブジェクト・マッピングを使用して、永続プロパティをバイトのシリアライズ・ストリームに維持することを指定します。

プロパティの 1 つ（getter または setter メソッド）またはエンティティのフィールド・レベルでシリアライズ・オブジェクトを定義します。

詳細は、次を参照してください。

- 『Oracle TopLink 開発者ガイド』のフィールドへの直接マッピングの理解に関する項
- 『Oracle TopLink 開発者ガイド』のコンバータ・マッピングの使用に関する項
- 『Oracle TopLink 開発者ガイド』のシリアライズ・オブジェクト・コンバータに関する項

アノテーションの使用方法

例 7-19 に、@Serialized アノテーションを使用してフィールド picture のシリアライズ・オブジェクト・マッピングを指定する方法を示します。

例 7-19 @Serialized

```
@Serialized(fetch=EAGER)
@Column(name="PICTURE")
public Byte[] getPicture() {
    return picture;
}
```

1 対 1 マッピングの構成

1 対 1 マッピングを使用して、2 つの Java オブジェクト間の単純なポインタ参照を表します。Java では、属性に格納されている単一のポインタが、ソース・オブジェクトとターゲット・オブジェクト間のマッピングを表します。リレーショナル・データベース表は、外部キーを使用してこれらのマッピングを実装します。

プロパティの 1 つ（getter または setter メソッド）またはエンティティのフィールド・レベルで 1 対 1 マッピングを定義します。

詳細は、『Oracle TopLink 開発者ガイド』の 1 対 1 マッピングの理解に関する項を参照してください。

注意： EJB 3.0 基本マッピングのコード例は、http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#onetoone を参照してください。

アノテーションの使用方法

例 7-20 に、@OneToOne アノテーションを使用してフィールド address の 1 対 1 マッピングを指定する方法を示します。

例 7-20 @OneToOne

```
@OneToOne(cascade=ALL, fetch=LAZY)
@JoinColumn(name="ADDR_ID")
public Address getAddress() {
    return address;
}
```

多対 1 マッピングの構成

多対 1 マッピングを使用して、2 つの Java オブジェクト間の単純なポインタ参照を表します。Java では、属性に格納されている単一のポインタが、ソース・オブジェクトとターゲット・オブジェクト間のマッピングを表します。リレーショナル・データベース表は、外部キーを使用してこれらのマッピングを実装します。

プロパティの 1 つ（getter または setter メソッド）またはエンティティのフィールド・レベルで多対 1 マッピングを定義します。

詳細は、『Oracle TopLink 開発者ガイド』の多対 1 マッピングの理解に関する項を参照してください。

注意： EJB 3.0 基本マッピングのコード例は、http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#manytoone を参照してください。

アノテーションの使用方法

例 7-21 に、@ManyToOne アノテーションを使用してフィールド manager の多対 1 マッピングを指定する方法を示します。

例 7-21 @ManyToOne

```
@ManyToOne(cascade=PERSIST, fetch=LAZY)
@JoinColumn(name="MANAGER_ID", referencedColumnName="EMP_ID")
public Employee getManager() {
    return manager;
}
```

1 対多マッピングの構成

1 対多マッピングを使用して、単一のソース・オブジェクトとターゲット・オブジェクトのコレクションの間の関連を表します。この関連は、Java ではターゲット・オブジェクトの Vector（またはその他のコレクション・タイプ）を使用して簡単に実装でき、リレーショナル・データベースを使用すると実装するのが難しいものの例です。

プロパティの 1 つ（getter または setter メソッド）またはエンティティのフィールド・レベルで 1 対多マッピングを定義します。

詳細は、『Oracle TopLink 開発者ガイド』の 1 対多マッピングの理解に関する項を参照してください。

注意： EJB 3.0 基本マッピングのコード例は、http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#onetomany を参照してください。

アノテーションの使用方法

例 7-22 に、@OneToMany アノテーションを使用してフィールド managedEmployees の 1 対多マッピングを指定する方法を示します。

例 7-22 @OneToMany

```
@OneToMany(cascade=PERSIST)
@JoinColumn(name="MANAGER_ID", referencedColumnName="EMP_ID")
public Collection getManagedEmployees() {
    return managedEmployees;
}
```

多対多マッピングの構成

多対多マッピングを使用して、ソース・オブジェクトのコレクションとターゲット・オブジェクトのコレクションの間の関連を表します。このマッピングには、ソース・レコードとターゲット・レコードの間の関連付けを管理するための中間表（関連表）の作成が必要です。

プロパティの 1 つ（getter または setter メソッド）またはエンティティのフィールド・レベルで多対多マッピングを定義します。

詳細は、『Oracle TopLink 開発者ガイド』の多対多マッピングの理解に関する項を参照してください。

注意： EJB 3.0 基本マッピングのコード例は、http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#manytomany を参照してください。

アノテーションの使用方法

例 7-23 に、@ManyToMany アノテーションを使用してフィールド projects の多対多マッピングを指定する方法、および @JoinTable アノテーションを使用して関連表を指定する方法を示します。

例 7-23 @ManyToMany

```
@ManyToMany(cascade=PERSIST)
@JoinTable(
    name="EJB_PROJ_EMP",
    joinColumns=@JoinColumn(name="EMP_ID", referencedColumnName="EMP_ID"),
    inverseJoinColumns=@JoinColumn(name="PROJ_ID", referencedColumnName="PROJ_ID")
)
public Collection getProjects() {
    return projects;
}
```

集約マッピングの構成

所有（親またはソース）エンティティおよび被所有（子またはターゲット）エンティティという 2 つのエンティティの間に厳密な 1 対 1 関連があり、被所有エンティティのすべての属性を所有エンティティと同じ表から取得できる場合、これらのエンティティは集約により関連しています。したがって、所有エンティティが存在する場合、被所有エンティティも存在する必要があり、所有エンティティが破棄された場合は、被所有エンティティも破棄されます。

集約マッピングでは、被所有エンティティのデータ・メンバーを所有エンティティの基礎となるデータベース表のフィールドに関連付けることができます。

所有エンティティでは、被所有フィールドまたは setter を埋込みとして指定します。

被所有エンティティでは、クラスを埋込み可能として指定し、それを所有エンティティの表名に関連付けます。

所有エンティティでは、被所有エンティティで行われた列指定（7-8 ページの「列の構成」を参照）をオーバーライドできます。

詳細は、『Oracle TopLink 開発者ガイド』の集約マッピングの理解に関する項を参照してください。

注意： EJB 3.0 基本マッピングのコード例は、http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#embedded を参照してください。

アノテーションの使用方法

例 7-24 に、@Embedded アノテーションを使用してフィールド period の集約マッピングを指定する方法を示します。このフィールドには、EmploymentPeriod のインスタンスが含まれています。例 7-25 に、@Embeddable アノテーションを使用して EmploymentPeriod エンティティ・クラスを集約マッピングで使用可能として指定する方法、および @Table アノテーション（7-8 ページの「プライマリ表の構成」を参照）を使用してこのクラスを所有エンティティの表に関連付ける方法を示します。

例 7-24 @Embedded

```
@Entity
@Table(name="EJB_EMPLOYEE")
public class Employee implements Serializable {
...
    @Embedded
    public EmploymentPeriod getPeriod() {
        return period;
    }
...
}
```

例 7-25 @Embeddable

```
@Embeddable
@Table(name="EJB_EMPLOYEE")
public class EmploymentPeriod implements Serializable {
    private Date startDate;
    private Date endDate;
...
}
```

所有エンティティで @AttributeOverride を使用して（例 7-26 を参照）、被所有エンティティで行われた列定義をオーバーライドできます（例 7-27 を参照）。

例 7-26 @Embedded および @AttributeOverride

```
@Entity
@Table(name="EJB_EMPLOYEE")
public class Employee implements Serializable {
...
    @Embedded({
            @AttributeOverride(name="startDate", column=@Column("EMP_START")),
            @AttributeOverride(name="endDate", column=@Column("EMP_END"))}
    )
    public EmploymentPeriod getPeriod() {
        return period;
    }
...
}
```

例 7-27 @Embeddable および @Column

```
@Embeddable
@Table(name="EJB_EMPLOYEE")
public class EmploymentPeriod implements Serializable {
    @Column("START_DATE")
    private Date startDate;

    @Column("END_DATE")
    private Date endDate;
...
}
```

オプティミスティック・ロック・バージョン・フィールドの構成

TopLink オプティミスティック・バージョン・ロック・ポリシーで使用するためにエンティティ・フィールドをバージョン・フィールドとして機能させることを指定できます。OC4J は、再アタッチ時に整合性を保証するため（29-18 ページの「エンティティ Bean インスタンスの連結解除およびマージ」を参照）、また全体的なオプティミスティック同時性制御のためにこのバージョン・フィールドを使用します。

オプティミスティック・ロック・バージョン・フィールドは、プロパティの 1 つ（getter または setter メソッド）またはエンティティのフィールド・レベルで定義します。

詳細は、『Oracle TopLink 開発者ガイド』のオプティミスティック・バージョン・ロック・ポリシーに関する項を参照してください。

アノテーションの使用方法

例 7-28 に、`@Version` アノテーションを使用して、列 `VERSION` を使用したオプティミスティック・バージョン・ロック・ポリシーを定義する方法を示します。

例 7-28 @Version

```
@Version
@Column(name="VERSION")
public int getVersion() {
    return version;
}
```

遅延ロードの構成

EJB 3.0 のすべてのマッピング・タイプ（基本および関連マッピング）で、次のいずれかを使用してデータベースからデータをフェッチする方針を定義できます。

- `FetchType.LAZY`: エンティティが取得されるときに、永続フィールドの値は取得されません。値が取得されるのは、フィールドへのアクセスが発生した時点です。
- `FetchTyep.EAGER`: エンティティが取得されるときに、永続フィールドの値も取得されます。

デフォルトでは、すべての永続フィールドが即座にフェッチされます。

EJB 3.0 アプリケーションで finder を使用する場合、finder レベルで遅延ロードを構成できます。これは、EJB 2.1 の `orion-ejb-jar.xml` ファイルを使用して構成する Oracle 固有のオプションです。詳細は、14-16 ページの「finder メソッドにおける遅延ロードの構成」を参照してください。

アノテーションの使用方法

例 7-29 に、`@Basic` アノテーションを使用して `LAZY` のフェッチ方針を定義する方法を示します。

例 7-29 @Basic の fetch 属性

```
@Basic(fetch=FetchType.LAZY)
@Column(name="F_NAME")
public String getFirstName() {
    return firstName;
}
```

JPA エンティティのライフ・サイクル・コールバック・メソッドの構成

JPA エンティティ・クラス・メソッドを次のライフ・サイクル・イベントのコールバック・メソッドとして指定できます。

- pre-persist
- post-persist
- pre-remove
- post-remove
- pre-update
- post-update
- post-load

エンティティ・クラス・メソッドは、次のシグネチャを持つ必要があります。

```
int <METHOD>()
```

エンティティ・クラス・メソッドには、`ejb` で始まらないかぎり自由にメソッド名を割り当てることができます。

詳細は、次を参照してください。

- 1-40 ページの「JPA エンティティのライフ・サイクル」
- 1-6 ページの「Bean クラスのライフ・サイクル・コールバック・メソッド」
- 『Oracle TopLink 開発者ガイド』のディスクリプタ・イベント・マネージャに関する項
- 『Oracle TopLink 開発者ガイド』の、イベント・ハンドラとしてのドメイン・オブジェクト・メソッドの構成に関する項

注意： EJB 3.0 ライフ・サイクル・コールバック・メソッドのコード例は、`http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30mappingannotations/doc/how-to-ejb30-mapping-annotations.html#callbacks` を参照してください。

アノテーションの使用方法

次のいずれかのアノテーションを使用して、JPA エンティティ・クラス・メソッドをライフ・サイクル・コールバック・メソッドとして指定できます。

- `@PrePersist`
- `@PostPersist`
- `@PreRemove`
- `@PostRemove`
- `@PreUpdate`
- `@PostUpdate`
- `@PostLoad`

例 7-30 に、`@PrePersist` アノテーションを使用して、JPA エンティティ・クラス・メソッド `initialize` をライフ・サイクル・コールバック・メソッドとして指定する方法を示します。

例 7-30 @PrePersist

```
@Entity
```

```
@Table(name="EJB_PROJECT")
public class Project implements Serializable {
    ...
    @Id()
    @Column(name="PROJECT_ID", primaryKey=true)
    public Integer getId() {
         return id;
    }

    ...

    @PrePersist
    public int initialize() {
        ...
    }
}
```

JPA エンティティのエンティティ・リスナー・クラスのライフ・サイクル・コールバック・リスナー・メソッドの構成

JPA エンティティのエンティティ・リスナー・クラスのエンティティ・リスナー・メソッドをライフ・サイクル・コールバック・メソッドとして指定できます。

エンティティ・リスナー・クラスでライフ・サイクル・コールバック・リスナー・メソッドを構成するには、次のようにします。

1. エンティティ・リスナー・クラスを作成します。

   これは、任意の POJO クラスにすることができます。

2. エンティティ・リスナー・クラスにライフ・サイクル・コールバック・リスナー・メソッドを実装します。

   JPA エンティティ・リスナー・クラスに定義するコールバック・メソッドには、次のシグネチャを割り当てます。

   ```
   void <METHOD>(Object)
   ```

   `Object` の引数型、またはエンティティ・リスナー・クラスを関連付ける JPA エンティティ・クラスの型を指定できます。

3. ライフ・サイクル・イベントをコールバック・リスナー・メソッドに関連付けます。

   ライフ・サイクル・イベントは 1 つのコールバック・リスナー・メソッドにのみ関連付けることができますが、特定のコールバック・リスナー・メソッドは 1 つ以上のライフ・サイクル・イベントに関連付けることができます。

   詳細は、次を参照してください。

   - 7-19 ページの「アノテーションの使用方法」

4. インターセプタ・クラスを JPA エンティティに関連付けます。

   詳細は、次を参照してください。

   - 7-19 ページの「アノテーションの使用方法」

詳細は、次を参照してください。

- 1-40 ページの「JPA エンティティのライフ・サイクル」
- 1-6 ページの「JPA エンティティ・リスナー・クラスのライフ・サイクル・コールバック・リスナー・メソッド」

アノテーションの使用方法

次のいずれかのアノテーションを使用して、JPA エンティティ・リスナー・メソッドをライフ・サイクル・コールバック・メソッドとして指定できます。

- @PrePersist
- @PostPersist
- @PreRemove
- @PostRemove
- @PreUpdate
- @PostUpdate
- @PostLoad

例 7-31 に、@PostConstruct および @PreDestroy アノテーションを使用して、JPA エンティティ・リスナー・メソッド myPostConstruct および myPreDestroy をそれぞれライフ・サイクル・コールバック・メソッドとして指定する方法を示します。

例 7-31 @PrePersist ライフ・サイクル・リスナー・コールバック・メソッド

```
public class MyProjectEntityListener {
    ...
    @PostConstruct
    public void myPostConstruct (Project obj) { // or just Object
        ...
    }

    @PreDestroy
    public void myPreDestroy (Project obj)  { // or just Object
        ...
    }
}
```

エンティティ・リスナー・クラスは、@EntityListeners アノテーションを使用して JPA エンティティに関連付けることができます。例 7-32 に、例 7-31 のエンティティ・リスナー・クラスを JPA エンティティ・クラスに関連付ける方法を示します。

@PrePersist のライフ・サイクル・メソッドは、JPA エンティティ・クラスそれ自体のメソッドです（7-18 ページの「JPA エンティティのライフ・サイクル・コールバック・メソッドの構成」を参照）。

例 7-32 エンティティ・リスナー・クラスと JPA エンティティの関連付け

```
@Entity
@EntityListeners(MyProjectEntityListener.class)
@Table(name="EJB_PROJECT")
public class Project implements Serializable {
    ...
    @Id
    @Column(name="PROJECT_ID", primaryKey=true)
    public Integer getId() {
        return id;
    }

    ...

    @PrePersist
    public int initialize() {
        ...
    }
}
```

JPA エンティティの継承の構成

OC4J では、クラスまたはクラス階層をリレーショナル・データベース・スキーマにマッピングするために次の継承計画がサポートされます。

- 結合されたサブクラス
- 各クラス階層の単一表

アノテーションを使用して、これらのアプローチを構成できます（7-22 ページの「アノテーションの使用方法」を参照）。

注意： EJB 3.0 継承のコード例は、
`http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30inheritance/doc/how-to-ejb30-inheritance.html` を参照してください。

結合されたサブクラス

この計画では、サブクラスに固有のフィールドが、親クラスに共通のフィールドとは異なる表にマッピングされ、サブクラスをインスタンス化するために結合が実行されます。

クラス階層のルートは、単一表で表されます。各サブクラスは、（スーパークラスから継承されていない）サブクラスに固有の列およびサブクラスの主キーを表す列を含む別の表で表されます。スーパークラスに対する追加の状態がサブクラスにない場合は、別の表は不要です。

サブクラス表に主キー列がある場合、その主キー列はスーパークラス表の主キーに対する外部キーとして機能します。サブクラス表の主キー列の名前がスーパークラス表の主キー列の名前と同じ場合、OC4J はこの関連を推測します。サブクラス表の主キー列名がスーパークラス表の主キー列の名前と同じでない場合（または、サブクラス表に主キー列がない場合）は、エンティティ・サブクラスのプライマリ表をスーパークラスのプライマリ表に結合するために使用するサブクラス表の列を指定する必要があります。

スーパークラスのプライマリ表には、識別子列として機能する列もあります。つまり、行で表されるインスタンスが属する特定のサブクラスを識別する値を持つ列です。

詳細は、7-22 ページの「アノテーションによる結合サブクラスの継承の構成」を参照してください。

各クラス階層の単一表

この計画では、階層内のすべてのクラスが単一表にマッピングされます。表には、識別子列として機能する列があります。追加の状態を追加する各サブクラスは、この単一表でのみこの新しい状態にマッピングされます。このような列は、そのサブクラスでのみ使用されます。

詳細は、7-23 ページの「アノテーションによる単一表の継承の構成」を参照してください。

アノテーションの使用方法

この項の内容は次のとおりです。

アノテーションによる結合サブクラスの継承の構成

次の例では、結合サブクラスのアプローチ（7-21 ページの「結合されたサブクラス」を参照）を使用して継承を構成する方法を示します。例 7-33 では、@Inheritance アノテーションをベース・クラス Project で使用する方法を示します。例 7-34 および例 7-35 では、@Inheritance アノテーションをそれぞれ派生クラス LargeProject および SmallProject で使用する方法を示します。

プライマリ表は、Project と SmallProject の両方がマッピングされる EJB_PROJECT です。EJB_PROJECT には、値がそれぞれ P、L および S である Project、LargeProject および SmallProject を表す PROJ_TYPE という識別子列があります。LargeProject は、状態を Project に追加するため、固有の表 EJB_LPROJECT にマッピングされます。この表には、LargeProject に固有の BUDGET などのフィールドが含まれています。EJB_LPROJECT には、主キー列がありません。かわりに、EJB_PROJECT の主キーと同じ名前を持つ外部キー（PROJ_ID）があります。

例 7-34 では、LargeProject クラスの主キー列名（LARGE_PROJECT_ID）はスーパークラス表（ID）の主キー列の名前と同じでないため、@InheritanceJoinColumn アノテーションを使用して、LargeProject プライマリ表をそのスーパークラスのプライマリ表に結合するために使用する列を指定する必要があります。

例 7-33 @Inheritance: 結合サブクラス継承のベース・クラス Project

```
@Entity
@Table(name="EJB_PROJECT")
@Inheritance(strategy=JOINED, discriminatorValue="P")
@DiscriminatorColumn(name="PROJ_TYPE")
public class Project implements Serializable {
...
    @Id()
    @Column(name="PROJECT_ID", primaryKey=true)
    public Integer getId() {
        return id;
    }
...
}
```

例 7-34 @Inheritance: 結合サブクラス継承の派生クラス LargeProject

```
@Entity
@Table(name="EJB_LPROJECT")
@Inheritance(discriminatorValue="L")
@InheritanceJoinColumn(name="LARGE_PROJECT_ID")
public class LargeProject extends Project {
...
    @Id()
    @Column(name="LARGE_PROJECT_ID", primaryKey=true)
    public Integer getProjectId() {
        return projectId;
    }
...
}
```

例 7-35 @Inheritance: 結合サブクラス継承の派生クラス SmallProject

```
@Entity
@Table(name="EJB_PROJECT")
@Inheritance(discriminatorValue="S")
public class SmallProject extends Project {
...
}
```

アノテーションによる単一表の継承の構成

次の例では、各クラス階層の単一表のアプローチ（7-21 ページの「各クラス階層の単一表」を参照）を使用して継承を構成する方法を示します。例 7-33 では、@Inheritance アノテーションをベース・クラス Project で使用する方法を示します。例 7-34 および例 7-35 では、@Inheritance アノテーションがそれぞれ派生クラス LargeProject および SmallProject でどのように不要であるかを示します。

プライマリ表は、Project と SmallProject の両方がマッピングされる EJB_PROJECT です。EJB_PROJECT 表には、Project のすべての列および LargeProject でのみ使用される追加列（BUDGET）が含まれます。

例 7-36 @Inheritance: 単一表継承のベース・クラス Project

```
@Entity
@Table(name="EJB_PROJECT")
@Inheritance(strategy=SINGLE_TABLE, discriminatorValue="P")
@DiscriminatorColumn(name="PROJ_TYPE")
public class Project implements Serializable {
...
}
```

例 7-37 @Inheritance: 単一表継承の派生クラス LargeProject

```
@Entity
@Inheritance(discriminatorValue="L")
public class LargeProject extends Project {
...
}
```

例 7-38 @Inheritance: 単一表継承の派生クラス SmallProject

```
@Entity
@Inheritance(discriminatorValue="S")
public class SmallProject extends Project {
...
}
```

8

JPA 問合せの実装

この章では、次のような、実行時にアクセスできる事前定義済の静的問合せの作成方法について説明します。

- JPA 名前付き問合せの実装
- JPA 動的問合せの実装
- JPA 問合せでの TopLink 問合せヒントの構成

詳細は、1-42 ページの「JPA エンティティの問合せ方法」を参照してください。

JPA 名前付き問合せの実装

名前付き問合せは、作成してコンテナ管理のエンティティに関連付ける事前定義済の問合せです（8-2 ページの「アノテーションの使用方法」を参照）。デプロイ時に、OC4J は EntityManager に名前付き問合せを格納します。実行時に、EntityManager を使用して、名前付き問合せを取得、構成および実行します。

詳細は、次を参照してください。

- 29-9 ページの「EntityManager の取得」
- 29-14 ページの「EntityManager での名前付き問合せの作成」
- 29-16 ページの「問合せの実行」

アノテーションの使用方法

例 8-1 に、@NamedQuery アノテーションを使用して Java 永続性問合せ言語の問合せを定義する方法を示します。この問合せは、実行時に EntityManager を使用して名前 findAllEmployeesByFirstName で取得できます。

例 8-1 @NamedQuery を使用した問合せの実装

```
@Entity
@NamedQuery(
    name="findAllEmployeesByFirstName",
    queryString="SELECT OBJECT(emp) FROM Employee emp WHERE emp.firstName = 'John'"
)
public class Employee implements Serializable {
...
}
```

例 8-2 では、@NamedQuery アノテーションを使用して、firstname という名前のパラメータを受け取る Java 永続性問合せ言語の問合せを定義する方法を示します。例 8-3 では、EntityManager を使用してこの問合せを取得し、Query のメソッド setParameter を使用して firstname パラメータを設定する方法を示します。名前付き問合せでの EntityManager の使用方法の詳細は、29-14 ページの「EntityManager を使用した JPA エンティティの問合せ」を参照してください。

オプションで、問合せヒントで名前付き問合せを構成し、JPA 永続性プロバイダのベンダー拡張を使用できます（8-4 ページの「JPA 問合せでの TopLink 問合せヒントの構成」を参照）。

例 8-2 @NamedQuery を使用したパラメータ付き問合せの実装

```
@Entity
@NamedQuery(
    name="findAllEmployeesByFirstName",
    queryString="SELECT OBJECT(emp) FROM Employee emp WHERE emp.firstName = :firstname"
)
public class Employee implements Serializable {
...
}
```

例 8-3 名前付き問合せのパラメータの設定

```
Query queryEmployeesByFirstName = em.createNamedQuery("findAllEmployeesByFirstName");
queryEmployeeByFirstName.setParameter("firstName", "John");
Collection employees = queryEmployessByFirstName.getResultList();
```

JPA 動的問合せの実装

EntityManager のメソッド createQuery または createNativeQuery を使用して、実行時に Query オブジェクトを動的に作成できます（8-3 ページの「Java の使用方法」を参照）。Query のメソッド getResultList、getSingleResult または executeUpdate を使用して、問合せを実行できます（29-16 ページの「問合せの実行」を参照）。

オプションで、問合せヒントで名前付き問合せを構成し、JPA 永続性プロバイダのベンダー拡張を使用できます（8-4 ページの「JPA 問合せでの TopLink 問合せヒントの構成」を参照）。

詳細は、次を参照してください。

- 29-9 ページの「EntityManager の取得」
- 29-15 ページの「EntityManager での動的 Java 永続性問合せ言語の問合せの作成」
- 29-15 ページの「EntityManager を使用した動的 TopLink 式問合せの作成」
- 29-16 ページの「EntityManager を使用した動的ネイティブ SQL 問合せの作成」
- 29-16 ページの「問合せの実行」

Java の使用方法

例 8-4 に、パラメータを指定して動的 EJB QL 問合せを作成する方法および問合せを実行する方法を示します。この例では、問合せは複数の結果を返すため、Query のメソッド getResultList を使用します。

例 8-4 動的問合せの実装および実行

```
Query queryEmployeeByFirstName = entityManager.createQuery(
    "SELECT OBJECT(emp) FROM Employee emp WHERE emp.firstName = :firstname"
);

queryEmployeeByFirstName.setParameter("firstName", "Joan");

Collection employees = queryEmployeeByFirstName.getResultList();
```

JPA 問合せでの TopLink 問合せヒントの構成

表 8-1 に、JPA 問合せを構築する場合（例 8-5 を参照）、または @QueryHint アノテーションを使用して JPA 問合せを指定する場合（例 8-6 を参照）に設定できる TopLink EJB 3.0 JPA 永続性プロバイダの問合せヒントをリストします。ヒントを設定する場合、次のような oracle.toplink.essentials.config の適切な構成クラス内の対応する public static final フィールドを使用して値を設定できます。

- PessimisticLock
- TopLinkQueryHints
- HintValues

> **注意：** これらのクラスにアクセスするには、適切な OC4J 永続性 JAR をクラスパスに配置します（3-3 ページの「JPA 永続性 JAR ファイル」を参照）。

例 8-5 TopLink JPA 問合せヒントの指定

```
import oracle.toplink.essentials.config.HintValues;
import oracle.toplink.essentials.config.TopLinkQueryHints;

Customer customer = (Customer)entityMgr.createNamedQuery("findCustomerBySSN").
setParameter("SSN", "123-12-1234").setHint(TopLinkQueryHints.BIND_PARAMETERS,
HintValues.PERSISTENCE_UNIT_DEFAULT).getSingleResult();
```

例 8-6 @QueryHint を使用した *TopLink* JPA 問合せヒントの指定

```
import oracle.toplink.essentials.config.HintValues;
import oracle.toplink.essentials.config.TopLinkQueryHints;

@Entity
@NamedQuery(
    name="findAllEmployees",
    query="SELECT * FROM EMPLOYEE WHERE MGR=1"
    hints={
        @QueryHint={name=TopLinkQueryHints.BIND_PARAMETERS, value=HintValues.PERSISTENCE_
UNIT_DEFAULT}
    }
)
public class Employee implements Serializable {
    ...
}
```

表 8-1 TopLink JPA 問合せヒント

ヒント	使用方法	デフォルト

toplink.jdbc.bind-parameters	問合せでパラメータ・バインディングを使用するかどうかを制御します。詳細は、『Oracle TopLink 開発者ガイド』の「パラメータ使用の SQL（バインド）とプリコンパイルされた SQL 文のキャッシュ」を参照してください。
	有効な値 : oracle.toplink.essentials.config.HintValues
	■ true: すべてのパラメータをバインドします。
	■ false: すべてのパラメータをバインドしません。
	■ PersistenceUnitDefault: TopLink セッションのデータベース・ログインで指定されたパラメータ・バインディング設定を使用します。
	詳細は、『Oracle TopLink 開発者ガイド』の「JDBC オプションの構成」を参照してください。
	例 : JPA 問合せ API
	`import oracle.toplink.essentials.config.HintValues;`
	`import oracle.toplink.essentials.config.TopLinkQueryHints;`
	`query.setHint(TopLinkQueryHints.BIND_PARAMETERS, HintValues.TRUE);`
	例 : @QueryHint
	`import oracle.toplink.essentials.config.HintValues;`
	`import oracle.toplink.essentials.config.TargetDatabase;`
	`@QueryHint(name=TopLinkQueryHints.BIND_PARAMETERS,`
	`value=HintValues.PERSISTENCE_UNIT_DEFAULT);`
		PersistenceUnitDefault

toplink.pessimistic-lock	ペシミスティック・ロックを使用するかどうかを制御します。
	有効な値 : oracle.toplink.essentials.config.PessimisticLock
	■ NoLock: ペシミスティック・ロックを使用しません。
	■ Lock: TopLink は SELECT.... FOR UPDATE を発行します。
	■ LockNoWait: TopLink は SELECT.... FOR UPDATE NO WAIT を発行します。
	例 : JPA 問合せ API
	`import oracle.toplink.essentials.config.PessimisticLock;`
	`import oracle.toplink.essentials.config.TopLinkQueryHints;`
	`query.setHint(TopLinkQueryHints.PESSIMISTIC_LOCK,`
	`PessimisticLock.LockNoWait);`
	例 : @QueryHint
	`import oracle.toplink.essentials.config.PessimisticLock;`
	`import oracle.toplink.essentials.config.TopLinkQueryHints;`
	`@QueryHint(name=TopLinkQueryHints.PESSIMISTIC_LOCK,`
	`value=PessimisticLock.LockNoWait);`
		NoLock

toplink.refresh	問合せから返されるオブジェクトで TopLink セッション・キャッシュを更新するかどうかを制御します。
	有効な値 : oracle.toplink.essentials.config.HintValues
	■ true: キャッシュをリフレッシュします。
	■ false: キャッシュをリフレッシュしません。
	例 : JPA 問合せ API
	`import oracle.toplink.essentials.config.HintValues;`
	`import oracle.toplink.essentials.config.TopLinkQueryHints;`
	`query.setHint(TopLinkQueryHints.REFRESH, HintValues.TRUE);`
	例 : @QueryHint
	`import oracle.toplink.essentials.config.HintValues;`
	`import oracle.toplink.essentials.config.TopLinkQueryHints;`
	`@QueryHint(name=TopLinkQueryHints.REFRESH, value=HintValues.TRUE);`
		false

第 IV 部

EJB 3.0 メッセージドリブン Bean

第 IV 部では、EJB 3.0 メッセージドリブン Bean の実装および構成の手順に関する情報を示します。概念的な情報は、第 I 部「EJB の概要」を参照してください。

第 IV 部は次の各章で構成されています。

- 第 9 章「EJB 3.0 メッセージドリブン Bean の実装」
- 第 10 章「EJB 3.0 メッセージドリブン Bean の使用方法」

9

EJB 3.0 メッセージドリブン Bean の実装

この章では、EJB 3.0 メッセージドリブン Bean（MDB）の実装方法を説明します。

詳細は、次を参照してください。

- 1-59 ページの「メッセージドリブン Bean とは」
- 第 10 章「EJB 3.0 メッセージドリブン Bean の使用方法」

EJB 3.0 MDB の実装

EJB 3.0 では、Enterprise Bean の開発が大幅に単純化され、多くの複雑な開発タスクが排除されています。次に例を示します。

- Bean クラスは、POJO にすることができます。`javax.ejb.MessageDrivenBean` を実装する必要はありません。
- アノテーションは、メッセージの宛先やトピック（または問合せ）ファクトリなどの多くの機能に使用されます。
- インジェクションを使用して、`MessageDrivenEntityContext` を取得できます。

詳細は、1-59 ページの「メッセージドリブン Bean とは」を参照してください。

注意： EJB 3.0 メッセージドリブン Bean のコード例は、`http://www.oracle.com/technology/tech/java/oc4j/10131/how_to/how-to-ejb30-mdb/doc/how-to-ejb30-mdb.html` からダウンロードできます。

EJB 3.0 メッセージドリブン Bean を実装するには、次のようにします。

1. メッセージ・サービス・プロバイダを構成します。

   詳細は、次を参照してください。

   - 2-26 ページの「MDB で使用できるメッセージ・サービス・プロバイダ」
   - 第 23 章「メッセージ・サービスの構成」

2. メッセージドリブン Bean クラスを作成します。

   POJO を作成し、`@MessageDriven` アノテーションを使用してそれをメッセージドリブン Bean として定義できます。

   注意： OC4J では、`@MessageDriven` の属性 `mappedName` は無視されます。

   ---

3. メッセージ・サービス・プロバイダ情報を構成します。

   この情報は、`@ActivationConfigProperty` アノテーションで定義できます。

   詳細は、次を参照してください。

   - 10-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」
   - 10-4 ページの「直接メッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」

4. `MessageDrivenContext` のデータ・メンバーを追加します。

   getter および setter メソッドを使用しなくても、リソース・インジェクションを使用してこのデータ・メンバーを簡単に初期化できます。

5. 適切なメッセージ・リスナー・インタフェースを実装します。

   JMS メッセージドリブン Bean では、`javax.jms.MessageListener` インタフェースを実装して `onMessages` メソッドに次のシグネチャを提供します。

   ```
   public void onMessage(javax.jms.Message message)
   ```

   このメソッドは、着信メッセージを処理します。ほとんどの MDB は、メッセージをキューまたはトピックから受信し、メッセージ内のリクエストを処理するために、エンティティ Bean を起動します。

   このメソッドでは、`TimedObject` インタフェースを実装した場合（手順 6 を参照）は、`MessageDrivenContext` を使用して `javax.ejb.TimerService` を取得および構成できます。

6. オプションで、`javax.ejb.TimedObject` インタフェースを実装します。

   次のシグネチャを持つ `ejbTimeout` メソッドを実装します。

   ```
   public void ejbTimeout(javax.ejb.Timer timer)
   ```

7. オプションで、適切なアノテーションを使用してライフ・サイクル・コールバック・メソッドを定義します。

   ライフ・サイクル・メソッドを定義する必要はありません。このようなメソッドの実装はすべて OC4J に用意されています。メッセージドリブン Bean のライフ・サイクルの特定の時点で独自のアクションを実行する場合にのみ、メッセージドリブン Bean クラスのメソッドをライフ・サイクル・コールバック・メソッドとして定義します。

   詳細は、10-12 ページの「EJB 3.0 MDB のライフ・サイクル・コールバック・インターセプタ・メソッドの構成」を参照してください。

8. オプションで、OC4J 固有のデプロイ・オプションを定義します。

   EJB 3.0 アプリケーションでこれを行うには、メッセージドリブン Bean クラスに OC4J 固有の `oracle.j2ee.ejb.@MessageDrivenDeployment` アノテーションを付けます（10-20 ページの「EJB 3.0 MDB の OC4J 固有のデプロイ・オプションの構成」を参照）。

9. メッセージドリブン Bean の構成を完了します（第 10 章「EJB 3.0 メッセージドリブン Bean の使用方法」を参照）。

10

EJB 3.0 メッセージドリブン Bean の使用方法

この章では、EJB 3.0 メッセージドリブン Bean を使用するために構成する必要のある様々なオプションについて説明します。

表 10-1 に、これらのオプションをリストし、基本オプション（ほとんどのアプリケーションに適用可能）であるか拡張オプション（より特殊なアプリケーションに適用可能）であるかを示します。

詳細は、次を参照してください。

- 1-59 ページの「メッセージドリブン Bean とは」
- 第 9 章「EJB 3.0 メッセージドリブン Bean の実装」

表 10-1 EJB 3.0 メッセージドリブン Bean の構成オプション

オプション	タイプ
10-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」	基本
10-4 ページの「直接メッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」	基本
18-6 ページの「Windows オペレーティング・システムでの高速アンデプロイのための MDB の構成」	拡張
18-6 ページの「Oracle RAC フェイルオーバー用の MDB の構成」	拡張
31-5 ページの「Bean インスタンスのプール・サイズの構成」	基本
21-9 ページの「メッセージドリブン Bean のトランザクション・タイムアウトの構成」	拡張
10-6 ページの「パラレル・メッセージ処理の構成」	拡張
10-8 ページの「最大配信数の構成」	拡張
10-10 ページの EJB 3.0 MDB の接続障害リカバリの構成	拡張
10-12 ページの「EJB 3.0 MDB のライフ・サイクル・コールバック・インターセプタ・メソッドの構成」	基本
10-13 ページの「EJB 3.0 MDB のインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドの構成」	拡張
10-15 ページの「EJB 3.0 MDB の AroundInvoke インターセプタ・メソッドの構成」	拡張
10-16 ページの「EJB 3.0 MDB のインターセプタ・クラスの AroundInvoke インターセプタ・メソッドの構成」	拡張
10-17 ページの「EJB 3.0 MDB のインターセプタ・クラスの構成」	拡張
10-20 ページの「EJB 3.0 MDB の OC4J 固有のデプロイ・オプションの構成」	拡張

J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成

Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してメッセージ・サービス・プロバイダにアクセスするよう EJB 3.0 MDB を構成できます。

これを行うには、アノテーション（10-2 ページの「アノテーションの使用方法」を参照）またはデプロイ XML（10-3 ページの「デプロイ XML の使用方法」を参照）を使用します。

注意： メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」を参照してください。

OC4J では、2 フェーズ・コミット（2PC）トランザクション用の XA ファクトリと、2PC を必要としないトランザクション用の非 XA ファクトリの両方がサポートされます。

詳細は、次を参照してください。

- 2-26 ページの「Oracle JMS コネクタ : J2EE Connector Architecture（J2CA）ベース・プロバイダ」
- 2-31 ページの「メッセージ・サービス構成オプション : アノテーションまたは XML の選択と属性またはアクティブ化構成プロパティの選択」
- 2-24 ページの「グローバル・トランザクションまたは 2 フェーズ・コミット（2PC）トランザクションへの参加方法」

アノテーションの使用方法

J2CA リソース・アダプタを使用して JMS メッセージ・サービス・プロバイダにアクセスするよう EJB 3.0 MDB を構成するには、次のようにします。

1. リソース・アダプタの名前を指定します。

   OC4J 固有の `@MessageDrivenDeployment` アノテーションの resourceAdapter 属性（例 10-1 を参照）、またはそれと同等の `orion-ejb-jar.xml` ファイルの `<message-driven-deployment>` 要素の `resource-adapter` 属性（10-3 ページの「デプロイ XML の使用方法」を参照）を使用できます。

2. 必要なアクティブ化構成プロパティを指定します。

   アクティブ化構成プロパティは、`@MessageDrivenDeployment` および `@MessageDriven` アノテーション（例 10-1 を参照）とデプロイ XML（10-3 ページの「デプロイ XML の使用方法」を参照）の任意の組合せを使用して指定できます。

   詳細は、次を参照してください。

   - 付録 B「J2CA アクティブ化構成プロパティ」
   - 2-31 ページの「メッセージ・サービス構成オプション : アノテーションまたは XML の選択と属性またはアクティブ化構成プロパティの選択」

例 10-1 に、`OracleASjms` という名前の Oracle JMS リソース・アダプタを使用するようにメッセージドリブン Bean を構成する方法を示します。メッセージ・サービス・プロバイダの構成時に、コネクション・ファクトリ `OracleASjms/MyQCF` を `oc4j-ra.xml` ファイルに定義し、接続先名 `OracleASjms/MyQueue` を `oc4j-connectors.xml` ファイルに定義してあることを前提とします。2 フェーズ・コミット（2PC）をサポートする XA 対応ファクトリを定義するか、または 2PC サポートが必要でない場合は非 XA ファクトリを定義します。J2CA メッセージ・サービス・プロバイダの構成の詳細は、23-2 ページの「メッセージ・サービス・プロバイダで使用するための J2CA リソース・アダプタの構成」を参照してください。

例 10-1 J2CA のメッセージ・サービス・プロバイダの @MessageDriven および @MessageDrivenDeployment アノテーション

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.ejb.ActivationConfigProperty;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="ConnectionFactoryJndiName", propertyValue="OracleASjms/MyQCF"),
        @ActivationConfigProperty(
            propertyName="DestinationName", propertyValue="OracleASjms/MyQueue"),
        @ActivationConfigProperty(
            propertyName="DestinationType", propertyValue="javax.jms.Queue"),
        @ActivationConfigProperty(
            propertyName="messageSelector", propertyValue="RECIPIENT = 'simple_jca_test'")
    })

// associate MDB with the resource adapter
@MessageDrivenDeployment(resourceAdapter = "OracleASjms")

public class JCAQueueMDB implements MessageListener {
    public void onMessage(Message msg) {
        ...
    }
}
```

使用する実際の名前は、メッセージ・サービス・プロバイダのインストール環境によって決まります。詳細は、23-2 ページの「J2CA メッセージ・サービス・プロバイダのコネクション・ファクトリ名」を参照してください。

デプロイ XML の使用方法

J2CA リソース・アダプタを使用して JMS メッセージ・サービス・プロバイダにアクセスするよう EJB 3.0 MDB を構成するには、EJB 2.1 MDB の場合と同様に（18-2 ページの「デプロイ XML の使用方法」を参照）、`ejb-jar.xml` と `orion-ejb.jar.xml` という 2 つのデプロイ XML ファイルを使用する必要があります。

アノテーション構成が存在する場合（10-2 ページの「アノテーションの使用方法」を参照）、このデプロイ XML 構成でオーバーライドできます。

直接メッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成

（J2CA リソース・アダプタを使用せずに）直接メッセージ・サービス・プロバイダにアクセスするよう EJB 3.0 MDB を構成できます。

これを行うには、アノテーション（10-4 ページの「アノテーションの使用方法」を参照）またはデプロイ XML（10-5 ページの「デプロイ XML の使用方法」を参照）を使用します。

注意： メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、次を参照してください。

- 2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」
- 10-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」

OC4J では、2 フェーズ・コミット（2PC）トランザクション用の XA ファクトリと、2PC を必要としないトランザクション用の非 XA ファクトリの両方がサポートされます。

詳細は、次を参照してください。

- 2-27 ページの「OEMS JMS: メモリー内またはファイルベース・プロバイダ」
- 2-28 ページの「OEMS JMS データベース : アドバンスト・キューイング（AQ）ベース・プロバイダ」
- 2-31 ページの「メッセージ・サービス構成オプション : アノテーションまたは XML の選択と属性またはアクティブ化構成プロパティの選択」
- 2-24 ページの「グローバル・トランザクションまたは 2 フェーズ・コミット（2PC）トランザクションへの参加方法」

アノテーションの使用方法

J2CA リソース・アダプタを使用せずに JMS メッセージ・サービス・プロバイダにアクセスするよう EJB 3.0 MDB を構成するには、次のようにします。

1. 必要なアクティブ化構成プロパティを指定します。

   アクティブ化構成プロパティは、`@MessageDrivenDeployment` アノテーション、`@MessageDriven` アノテーションおよびデプロイ XML の任意の組合せを使用して指定できます。

   詳細は、次を参照してください。

   - 付録 B「J2CA アクティブ化構成プロパティ」
   - 2-31 ページの「メッセージ・サービス構成オプション : アノテーションまたは XML の選択と属性またはアクティブ化構成プロパティの選択」

例 10-2 に、（J2CA リソース・アダプタを使用せずに）直接 JMS メッセージ・サービス・プロバイダにアクセスするようメッセージドリブン Bean を構成する方法を示します。メッセージ・サービス・プロバイダの構成時に、コネクション・ファクトリ `jms/MyQCF` およびキュー `jms/MyQueue` を定義してあることを前提とします。2 フェーズ・コミット（2PC）をサポートする XA 対応ファクトリを定義するか、または 2PC サポートが必要でない場合は非 XA ファクトリを定義します。メッセージ・サービス・プロバイダの構成の詳細は、第 23 章「メッセージ・サービスの構成」を参照してください。

例 10-2 J2CA 以外のメッセージ・サービス・プロバイダの @MessageDriven アノテーション

```
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.TextMessage;
import javax.jms.MessageListener;

@MessageDriven(
    messageListenerInterface=MessageListener.class,
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="connectionFactoryJndiName", propertyValue="jms/MyQCF"),
        @ActivationConfigProperty(
            propertyName="destinationName", propertyValue="jms/MyQueue"),
        @ActivationConfigProperty(
            propertyName="destinationType", propertyValue="javax.jms.Queue"),
        @ActivationConfigProperty(
            propertyName="messageSelector", propertyValue="RECIPIENT = 'simple_test'")
    })

public class QueueMDB implements MessageListener {
    public void onMessage(Message msg) {
        ...
    }
}
```

使用する実際の名前は、メッセージ・サービス・プロバイダのインストール環境によって決まります。詳細は、次を参照してください。

- 23-4 ページの「OEMS JMS 宛先名およびコネクション・ファクトリ名」
- 23-7 ページの「OEMS JMS データベース宛先名およびコネクション・ファクトリ名」

デプロイ XML の使用方法

（J2CA リソース・アダプタを使用せずに）直接 JMS メッセージ・サービス・プロバイダにアクセスするよう EJB 3.0 MDB を構成するには、EJB 2.1 MDB の場合と同様に（18-4 ページの「デプロイ XML の使用方法」を参照）、`ejb-jar.xml` と `orion-ejb.jar.xml` という 2 つのデプロイ XML ファイルを使用する必要があります。

アノテーション構成が存在する場合（10-4 ページの「アノテーションの使用方法」を参照）、このデプロイ XML 構成でオーバーライドできます。

パラレル・メッセージ処理の構成

デフォルトでは、OC4J はメッセージ・ロケーションのメッセージをポーリングするために 1 つの受信スレッドを使用します。

2 つ以上の受信スレッドを使用すると、メッセージをパラレルに受信できるため、パフォーマンスが向上する可能性があります。

メッセージ・ロケーションがトピックの場合、受信スレッドの数は 1 に固定されます。

メッセージ・ロケーションがキューの場合、OC4J 固有のアノテーション（10-6 ページの「アノテーションの使用方法」を参照）または `orion-ejb-jar.xml` ファイル（10-7 ページの「デプロイ XML の使用方法」を参照）を使用して受信スレッドの数を構成できます。

MDB プールの Bean インスタンスの最小数は、受信スレッドがメッセージ処理のためにプールから Bean インスタンスを取得できるように、少なくとも受信スレッドの数と同じに設定する必要があります。

詳細は、次を参照してください。

- 2-31 ページの「メッセージ・サービス構成オプション : アノテーションまたは XML の選択と属性またはアクティブ化構成プロパティの選択」
- 31-5 ページの「Bean インスタンスのプール・サイズの構成」

アノテーションの使用方法

このオプションの構成方法は、使用するメッセージ・サービス・プロバイダへのアクセス方法によって決まります。

- J2CA リソース・アダプタを使用したメッセージ・サービス・プロバイダへのアクセス
- J2CA リソース・アダプタを使用しないメッセージ・サービス・プロバイダへのアクセス

J2CA リソース・アダプタを使用したメッセージ・サービス・プロバイダへのアクセス

J2CA リソース・アダプタを使用してメッセージ・サービス・プロバイダにアクセスする場合、例 10-3 に示すようにアクティブ化構成プロパティ `ReceiverThreads` を設定します。

`ReceiverThreads` の詳細は、表 B-2 を参照してください。

例 10-3 J2CA アダプタを使用したメッセージ・サービス・プロバイダのためのパラレル・メッセージ処理の構成

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.ejb.ActivationConfigProperty;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(propertyName="ReceiverThreads", propertyValue="3"),
        ...
    }
)

@MessageDrivenDeployment(
    resourceAdapter = "OracleASjms",
    ...
)

public class JCAQueueMDB implements MessageListener {
    public void onMessage(Message msg) {
        ...
    }
}
```

J2CA リソース・アダプタを使用しないメッセージ・サービス・プロバイダへのアクセス

(J2CA リソース・アダプタを使用せずに) 直接メッセージ・サービス・プロバイダにアクセスする場合、例 10-4 に示すように OC4J 固有のアノテーション `@MessageDrivenDeployment` の属性 listenerThreads を設定します。

この `@MessageDrivenDeployment` の属性の詳細は、表 A-3 を参照してください。`@MessageDrivenDeployment` アノテーションの詳細は、10-20 ページの「EJB 3.0 MDB の OC4J 固有のデプロイ・オプションの構成」を参照してください。

注意 : メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、次を参照してください。

- 2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」
- 10-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」

例 10-4 J2CA アダプタを使用しないメッセージ・サービス・プロバイダのためのパラレル・メッセージ処理の構成

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(
    ...
)

@MessageDrivenDeployment(
    listenerThreads=3
)

public class QueueMDB implements MessageListener {
    public void onMessage(Message msg) {
        ...
    }
}
```

デプロイ XML の使用方法

EJB 3.0 メッセージドリブン Bean では、EJB 2.1 メッセージドリブン Bean と同様に、`orion-ejb-jar.xml` ファイルでパラレル・メッセージ処理を構成します (18-8 ページの「デプロイ XML の使用方法」を参照)。

最大配信数の構成

メッセージドリブン Bean のメッセージ・リスナー・メソッド（JMS メッセージ・リスナーの onMessage メソッドなど）がエラーを返した場合（確認応答操作の起動に失敗した場合、または例外をスローした場合、あるいはその両方の場合）に、OC4J がそのメソッドにメッセージの即時再配信を試行する最大回数を構成できます。

この回数の再配信が行われた後で、メッセージは配信不能とみなされ、メッセージ・サービス・プロバイダのポリシーに従って処理されます。たとえば、OEMS JMS はその例外キュー（jms/Oc4jJmsExceptionQueue）にメッセージを挿入します。

最大配信数は、OC4J 固有のアノテーション（10-8 ページの「アノテーションの使用方法」を参照）または orion-ejb-jar.xml ファイル（10-9 ページの「デプロイ XML の使用方法」を参照）を使用して構成できます。

詳細は、2-31 ページの「メッセージ・サービス構成オプション : アノテーションまたは XML の選択と属性またはアクティブ化構成プロパティの選択」を参照してください。

アノテーションの使用方法

このオプションの構成方法は、使用するメッセージ・サービス・プロバイダのタイプによって決まります。

- J2CA リソース・アダプタを使用したメッセージ・サービス・プロバイダへのアクセス
- J2CA リソース・アダプタを使用しないメッセージ・サービス・プロバイダへのアクセス

J2CA リソース・アダプタを使用したメッセージ・サービス・プロバイダへのアクセス

J2CA リソース・アダプタを使用してメッセージ・サービス・プロバイダにアクセスする場合、例 10-5 に示すようにアクティブ化構成プロパティ MaxDeliveryCnt を設定します。

MaxDeliveryCnt の詳細は、表 B-2 を参照してください。

例 10-5 J2CA アダプタを使用したメッセージ・サービス・プロバイダのための最大配信数の構成

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.ejb.ActivationConfigProperty;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(propertyName="MaxDeliveryCnt", propertyValue="3"),
        ...
    }
)

@MessageDrivenDeployment(
    resourceAdapter = "OracleASjms",
    ...
)

public class JCAQueueMDB implements MessageListener {
    public void onMessage(Message msg) {
        ...
    }
}
```

J2CA リソース・アダプタを使用しないメッセージ・サービス・プロバイダへのアクセス

(J2CA リソース・アダプタを使用せずに) 直接メッセージ・サービス・プロバイダにアクセスする場合、例 10-6 に示すように OC4J 固有のアノテーション `@MessageDrivenDeployment` の属性 maxDeliveryCount を設定します。

この `@MessageDrivenDeployment` の属性の詳細は、表 A-3 を参照してください。`@MessageDrivenDeployment` アノテーションの詳細は、10-20 ページの「EJB 3.0 MDB の OC4J 固有のデプロイ・オプションの構成」を参照してください。

注意： メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、次を参照してください。

- 2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」
- 10-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」

例 10-6 J2CA アダプタを使用しないメッセージ・サービス・プロバイダのための最大配信数の構成

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(
    ...
)

@MessageDrivenDeployment(
    maxDeliveryCount=3
)

public class QueueMDB implements MessageListener {
    public void onMessage(Message msg) {
        ...
    }
}
```

デプロイ XML の使用方法

EJB 3.0 メッセージドリブン Bean では、EJB 2.1 メッセージドリブン Bean と同様に、`orion-ejb-jar.xml` ファイルで最大配信数を構成します (18-9 ページの「デプロイ XML の使用方法」を参照)。

EJB 3.0 MDB の接続障害リカバリの構成

ネットワークや JMS サーバーの停止などのイベントを原因とする接続障害に対し、メッセージドリブン Bean のリスナー・スレッドでどのように応答するかを構成できます。

これらのオプションは、メッセージドリブン Bean のコンテナ管理のトランザクションにのみ適用されます。

接続障害リカバリ・オプションは、OC4J 固有のアノテーション（10-10 ページの「アノテーションの使用方法」を参照）または `orion-ejb-jar.xml` ファイル（10-11 ページの「デプロイ XML の使用方法」を参照）を使用して構成できます。

詳細は、次を参照してください。

- 2-35 ページの「OC4J EJB アプリケーション・クラスタリング・サービスについて」
- 2-31 ページの「メッセージ・サービス構成オプション : アノテーションまたは XML の選択と属性またはアクティブ化構成プロパティの選択」

アノテーションの使用方法

このオプションの構成方法は、使用するメッセージ・サービス・プロバイダのタイプによって決まります。

- J2CA リソース・アダプタを使用したメッセージ・サービス・プロバイダへのアクセス
- J2CA リソース・アダプタを使用しないメッセージ・サービス・プロバイダへのアクセス

J2CA リソース・アダプタを使用したメッセージ・サービス・プロバイダへのアクセス

J2CA リソース・アダプタを使用してメッセージ・サービス・プロバイダにアクセスする場合、Oracle JMS コネクタは、JMS リソースのポーリングを無制限に再試行します。この再試行間隔は、例 10-7 に示すようにアクティブ化構成プロパティ `EndpointFailureRetryInterval` で構成できます。

再試行後のメッセージのリカバリでは、メッセージの順序は保証されません。また、JMS トピックに対する MDB サブスクリプションが非永続的な場合、メッセージは失われるか、重複する可能性があります。

詳細は、表 B-2 の `EndpointFailureRetryInterval` を参照してください。

例 10-7 J2CA アダプタを使用したメッセージ・サービス・プロバイダのための接続障害リカバリの構成

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.ejb.ActivationConfigProperty;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="EndpointFailureRetryInterval",
            propertyValue="20000"
        ),
        ...
    }
)

@MessageDrivenDeployment(
    resourceAdapter = "OracleASjms",
    ...
)

public class JCAQueueMDB implements MessageListener {
    public void onMessage(Message msg) {
```

```
        ...
    }
}
```

J2CA リソース・アダプタを使用しないメッセージ・サービス・プロバイダへのアクセス

（J2CA リソース・アダプタを使用せずに）直接メッセージ・サービス・プロバイダにアクセスする場合、例 10-8 に示すように OC4J 固有のアノテーション `@MessageDrivenDeployment` の属性 dequeueRetryCount および dequeueRetryInterval を設定します。

この `@MessageDrivenDeployment` の属性の詳細は、表 A-3 を参照してください。`@MessageDrivenDeployment` アノテーションの詳細は、10-20 ページの「EJB 3.0 MDB の OC4J 固有のデプロイ・オプションの構成」を参照してください。

注意： メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、次を参照してください。

- 2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」
- 10-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」

例 10-8 J2CA アダプタを使用しないメッセージ・サービス・プロバイダのための接続障害リカバリの構成

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(
        ...
)

@MessageDrivenDeployment(
    dequeueRetryCount=3,
    dequeueRetryInterval=90
)

public class QueueMDB implements MessageListener {
    public void onMessage(Message msg) {
        ...
    }
}
```

デプロイ XML の使用方法

EJB 3.0 メッセージドリブン Bean では、EJB 2.1 メッセージドリブン Bean と同様に、`orion-ejb-jar.xml` ファイルでデキュー再試行を構成します（18-9 ページの「デプロイ XML の使用方法」を参照）。

EJB 3.0 MDB のライフ・サイクル・コールバック・インターセプタ・メソッドの構成

EJB 3.0 メッセージドリブン Bean クラス・メソッドを次のライフ・サイクル・イベントのコールバック・メソッドとして指定できます（10-12 ページの「アノテーションの使用方法」を参照）。

- post-construct
- pre-destroy

メッセージドリブン Bean クラスのライフ・サイクル・コールバック・メソッドは、次のシグネチャを持つ必要があります。

```
void <METHOD>()
```

EJB 3.0 メッセージドリブン Bean に関連付けるインターセプタ・クラスで 1 つ以上のライフ・サイクル・コールバック・メソッドを指定することもできます（10-13 ページの「EJB 3.0 MDB のインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドの構成」を参照）。

詳細は、次を参照してください。

- 1-60 ページの「メッセージドリブン Bean のライフ・サイクル」
- 1-6 ページの「Bean クラスのライフ・サイクル・コールバック・メソッド」

アノテーションの使用方法

次のいずれかのアノテーションを使用して、EJB 3.0 メッセージドリブン Bean クラス・メソッドをライフ・サイクル・コールバック・メソッドとして指定できます。

- `@PostConstruct`
- `@PreDestroy`

例 10-9 に、`@PostConstruct` アノテーションを使用して EJB 3.0 メッセージドリブン Bean のクラス・メソッド `initialize` をライフ・サイクル・コールバック・メソッドとして指定する方法を示します。

例 10-9 EJB 3.0 メッセージドリブン Bean の @PostConstruct

```
@MessageDriven
public class MessageLogger implements MessageListener {
    @Resource javax.ejb.MessageDrivenContext mc;

    public void onMessage(Message message) {
    ....
    }

    @PostConstruct
    public void initialize() {
        // Initialization logic
    }
...
}
```

EJB 3.0 MDB のインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドの構成

EJB 3.0 メッセージドリブン Bean のインターセプタ・クラスのインターセプタ・メソッドをライフ・サイクル・コールバック・インターセプタ・メソッドとして指定できます。

インターセプタ・クラスでライフ・サイクル・コールバック・インターセプタ・メソッドを構成するには、次のようにします。

1. インターセプタ・クラスを作成します。

   これは、任意の POJO クラスにすることができます。

   インターセプタ・クラスは、引数のない public コンストラクタを保持する必要があります。

2. ライフ・サイクル・コールバック・インターセプタ・メソッドを実装します。

   Bean のインターセプタ・クラスに定義するコールバック・メソッドには、次のシグネチャを割り当てます。

   ```
   Object <METHOD>(InvocationContext)
   ```

3. ライフ・サイクル・イベントをコールバック・インターセプタ・メソッドに関連付けます。

   1 つのライフ・サイクル・イベントは、1 つのコールバック・インターセプタ・メソッドにのみ関連付けることができますが、1 つのライフ・サイクル・コールバック・インターセプタ・メソッドは、複数のコールバック・イベントに割り込むために使用できます。たとえば、`@PostConstruct` と `@PreDestroy` は、インターセプタ・クラス内で 1 回のみ出現可能ですが、`@PostConstruct` と `@PreDestroy` の両方を同じコールバック・インターセプタ・メソッドに関連付けることができます。

   詳細は、次を参照してください。

   - 10-14 ページの「アノテーションの使用方法」

4. インターセプタ・クラスを EJB 3.0 メッセージドリブン Bean に関連付けます（10-17 ページの「EJB 3.0 MDB のインターセプタ・クラスの構成」を参照）。

詳細は、次を参照してください。

- 1-60 ページの「メッセージドリブン Bean のライフ・サイクル」
- 1-6 ページの「EJB 3.0 インターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッド」

アノテーションの使用方法

次のいずれかのアノテーションを使用して、インターセプタ・クラス・メソッドを EJB 3.0 メッセージドリブン Bean のライフ・サイクル・コールバック・メソッドとして指定できます。

- `@PostConstruct`
- `@PreDestroy`

例 10-10 に、`@PostConstruct` および `@PreDestroy` アノテーションを使用して、`myPostConstructMethod` および `myPreDestroyMethod` をライフ・サイクル・コールバック・インターセプタ・メソッドとして指定する方法を示します。OC4J は、適切なライフ・サイクル・イベントが発生した場合にのみ、対応するライフ・サイクル・メソッドを起動します。OC4J は、メッセージドリブン Bean のビジネス・メソッドが起動されるたびに、ライフ・サイクル・インターセプタ・メソッド以外のすべてのインターセプタ・メソッド（`myInterceptorMethod` など）を起動します（10-17 ページの「EJB 3.0 MDB のインターセプタ・クラスの構成」を参照）。

例 10-10 インターセプタ・クラス

```
public class MyInterceptor {
    ...
    public void myInterceptorMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }

    @PostConstruct
    public void myPostContructMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }

    @PreDestroy
    public void myPreDestroyMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }
}
```

EJB 3.0 MDB の AroundInvoke インターセプタ・メソッドの構成

1 つの非ビジネス・メソッドを EJB 3.0 メッセージドリブン Bean のインターセプタ・メソッドとして指定できます。onMessage メソッドが起動するたびに、OC4J は起動をインターセプトし、インターセプタ・メソッドを起動します。onMessage メソッド起動は、インターセプタ・メソッドが InvocationContext.proceed() を返す場合にのみ続行されます。

インターセプタ・メソッドには次のシグネチャがあります。

```
Object <METHOD>(InvocationContext) throws Exception
```

インターセプタ・メソッドには、public、private、protected または package レベルのアクセスを割り当てることができますが、final または static として宣言することはできません。

このメソッドは、EJB 3.0 メッセージドリブン Bean クラスに指定するか（10-15 ページの「アノテーションの使用方法」を参照）、EJB 3.0 メッセージドリブン Bean に関連付けるインターセプタ・クラスに指定できます（10-16 ページの「EJB 3.0 MDB のインターセプタ・クラスの AroundInvoke インターセプタ・メソッドの構成」を参照）。

詳細は、2-12 ページの「EJB 3.0 インターセプタについて」を参照してください。

アノテーションの使用方法

例 10-11 に、@AroundInvoke アノテーションを使用してメッセージドリブン Bean クラスのメソッドをインターセプタ・メソッドとして指定する方法を示します。onMessage メソッドが起動するたびに、OC4J は起動をインターセプトし、インターセプタ・メソッド myInterceptor を起動します。onMessage メソッド起動は、インターセプタ・メソッドが InvocationContext.proceed() を返す場合にのみ続行されます。

例 10-11 EJB 3.0 メッセージドリブン Bean の @AroundInvoke

```
@MessageDriven
public class MessageLogger implements MessageListene {
    @Resource javax.ejb.MessageDrivenContext mc;

    public void onMessage(Message message) {
    ....
    }

    @AroundInvoke
    public Object myInterceptor(InvocationContext ctx) throws Exception  {
        if (!userIsValid(ctx. getEJBContext().getCallerPrincipal())) {
            throw new SecurityException(
                "Caller: '" + ctx.getEJBContext().getCallerPrincipal().getName() +
                "' does not have permissions for method " + ctx.getMethod()
            );
        }
        return ctx.proceed();
    }
}
```

EJB 3.0 MDB のインターセプタ・クラスの AroundInvoke インターセプタ・メソッドの構成

1 つの非ビジネス・メソッドを EJB 3.0 メッセージドリブン Bean のインターセプタ・メソッドとして指定できます。onMessage メソッドが起動するたびに、OC4J は起動をインターセプトし、インターセプタ・メソッドを起動します。onMessage の起動は、インターセプタ・メソッドが InvocationContext.proceed() を返す場合にのみ続行されます。

このメソッドは、EJB 3.0 MDB に関連付けるインターセプタ・クラスに指定するか、EJB 3.0 MDB クラスそれ自体に指定できます（10-15 ページの「EJB 3.0 MDB の AroundInvoke インターセプタ・メソッドの構成」を参照）。

インターセプタ・クラスでインターセプタ・メソッドを構成するには、次のようにします。

1. インターセプタ・クラスを作成します。

   これは、任意の POJO クラスにすることができます。

2. インターセプタ・メソッドを実装します。

   インターセプタ・メソッドには次のシグネチャがあります。

   ```
   Object <METHOD>(InvocationContext) throws Exception
   ```

   インターセプタ・メソッドには、public、private、protected または package レベルのアクセスを割り当てることができますが、final または static として宣言することはできません。

3. メソッドをインターセプタ・メソッドとして指定します（10-16 ページの「アノテーションの使用方法」を参照）。

4. インターセプタ・クラスを EJB 3.0 MDB に関連付けます（10-17 ページの「EJB 3.0 MDB のインターセプタ・クラスの構成」を参照）。

詳細は、2-12 ページの「EJB 3.0 インターセプタについて」を参照してください。

アノテーションの使用方法

例 10-12 に、@AroundInvoke アノテーションを使用してインターセプタ・クラス・メソッドの myInterceptor を EJB 3.0 MDB のインターセプタ・メソッドとして指定する方法を示します。このインターセプタ・クラスを MDB に関連付けると（10-17 ページの「EJB 3.0 MDB のインターセプタ・クラスの構成」を参照）、onMessage メソッドが起動するたびに、OC4J は起動をインターセプトし、インターセプタ・メソッド myInterceptor を起動します。onMessage メソッド起動は、インターセプタ・メソッドが InvocationContext.proceed() を返す場合にのみ続行されます。

例 10-12 インターセプタ・クラス

```
public class MyInterceptor {
    ...
    @AroundInvoke
    protected Object myInterceptor(InvocationContext ctx) throws Exception {
        Principal p = ctx.getEJBContext().getCallerPrincipal;
        if (!userIsValid(p)) {
            throw new SecurityException(
                "Caller: '" + p.getName() +
                "' does not have permissions for method " + ctx.getMethod()
            );
        }
        return ctx.proceed();
    }

    @PreDestroy
    public void myPreDestroyMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }
}
```

EJB 3.0 MDB のインターセプタ・クラスの構成

インターセプタ・クラスは、Bean クラスそれ自体とは異なる 1 つのクラスであり、そのメソッドは Bean のビジネス・メソッドの起動およびライフ・サイクル・イベントの発生に応じて起動されます。Bean クラスは、任意の数のインターセプタ・クラスに関連付けることができます。

インターセプタ・クラスは、EJB 3.0 メッセージドリブン Bean に関連付けることができます。

インターセプタ・クラスを使用して EJB 3.0 メッセージドリブン Bean を構成するには、次のようにします。

1. インターセプタ・クラスを作成します（10-18 ページの「インターセプタ・クラスの作成」を参照）。

   これは、任意の POJO クラスにすることができます。

2. インターセプタ・クラスにインターセプタ・メソッドを実装します。

   インターセプタ・メソッドには次のシグネチャがあります。

   ```
   Object <METHOD>(InvocationContext) throws Exception
   ```

   インターセプタ・メソッドには、public、private、protected または package レベルのアクセスを割り当てることができますが、final または static として宣言することはできません。

   インターセプタ・メソッドには、ライフ・サイクル・コールバックとして（10-13 ページの「EJB 3.0 MDB のインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドの構成」を参照）、または `AroundInvoke` メソッドとして（10-16 ページの「EJB 3.0 MDB のインターセプタ・クラスの AroundInvoke インターセプタ・メソッドの構成」を参照）アノテーションを付けることができます。

3. インターセプタ・クラスを EJB 3.0 メッセージドリブン Bean に関連付けます（10-18 ページの「インターセプタ・クラスと MDB との関連付け」を参照）。

4. オプションで、シングルトン・インターセプタを使用するようメッセージドリブン Bean を構成します（10-19 ページの「MDB でのシングルトン・インターセプタの指定」を参照）。

アノテーションの使用方法

この項の内容は次のとおりです。

- インターセプタ・クラスの作成
- インターセプタ・クラスと MDB との関連付け
- MDB でのシングルトン・インターセプタの指定

インターセプタ・クラスの作成

例 10-13 に、EJB 3.0 メッセージドリブン Bean のインターセプタ・クラスに AroundInvoke インターセプタ・メソッドおよびライフ・サイクル・コールバック・インターセプタ・メソッドを指定する方法を示します。このインターセプタ・クラスをメッセージドリブン Bean に関連付けると（例 10-14 を参照）、onMessage メソッドが起動するたびに、OC4J は起動をインターセプトし、AroundInvoke メソッドの myInterceptor を起動します。適切なライフ・サイクル・イベントが発生すると、OC4J は myPreDestroyMethod などの対応するライフ・サイクル・コールバック・インターセプタ・メソッドを起動します。

例 10-13 インターセプタ・クラス

```
public class MyInterceptor {
    ...
    @AroundInvoke
    protected Object myInterceptor(InvocationContext ctx) throws Exception {
        Principal p = ctx.getEJBContext().getCallerPrincipal;
        if (!userIsValid(p)) {
            throw new SecurityException(
                "Caller: '" + p.getName() +
                "' does not have permissions for method " + ctx.getMethod()
            );
        }
        return ctx.proceed();
    }

    @PreDestroy
    public void myPreDestroyMethod (InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }
}
```

インターセプタ・クラスと MDB との関連付け

インターセプタ・クラスは、@Interceptors アノテーションを使用して EJB 3.0 メッセージドリブン Bean に関連付けることができます。例 10-14 に、例 10-13 のインターセプタ・クラスを EJB 3.0 メッセージドリブン Bean クラスに関連付ける方法を示します。

@PostConstruct のライフ・サイクル・メソッドは、EJB 3.0 メッセージドリブン Bean クラスそれ自体のメソッドですが（10-12 ページの「EJB 3.0 MDB のライフ・サイクル・コールバック・インターセプタ・メソッドの構成」を参照）、@PreDestroy のライフ・サイクル・メソッドは、このメッセージドリブン Bean に関連付けられたインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドです（10-13 ページの「EJB 3.0 MDB のインターセプタ・クラスのライフ・サイクル・コールバック・インターセプタ・メソッドの構成」を参照）。

例 10-14 インターセプタ・クラスと EJB 3.0 MDB の関連付け

```
@MessageDriven
@Interceptors(MyInterceptor.class)
public class MessageLogger implements MessageListener {
    @Resource javax.ejb.MessageDrivenContext mc;

    public void onMessage(Message message) {
    ....
    }

    @PostConstruct
    public void initialize() {
        items = new ArrayList();
    }
    ...
}
```

MDB でのシングルトン・インターセプタの指定

例 10-15 に示すように、`@MessageDrivenDeployment` の属性 interceptorType を `singleton` に設定することで、シングルトン・インターセプタ・クラスを使用するよう OC4J を構成できます。このメッセージドリブン Bean のすべてのインスタンスは、`MyInterceptor` の同じインスタンスを共有します。`MyInterceptor` クラスは、ステートレスである必要があります。

この属性の詳細は、表 A-3 を参照してください。シングルトン・インターセプタの詳細は、2-14 ページの「シングルトン・インターセプタ」を参照してください。

例 10-15 EJB 3.0 MDB でのシングルトン・インターセプタ・クラスの指定

```
@MessageDriven
@MessageDrivenDeployment(interceptorType="singleton")
@Interceptors(MyInterceptor.class)
public class MessageLogger implements MessageListener {
    @Resource javax.ejb.MessageDrivenContext mc;

    public void onMessage(Message message) {
    ....
    }

    @PostConstruct
    public void initialize() {
        items = new ArrayList();
    }
    ...
}
```

EJB 3.0 MDB の OC4J 固有のデプロイ・オプションの構成

EJB 3.0 メッセージドリブン Bean の OC4J 固有のデプロイ・オプションは、OC4J 固有のアノテーション（10-20 ページの「アノテーションの使用方法」を参照）または `orion-ejb-jar.xml` ファイル（10-21 ページの「デプロイ XML の使用方法」を参照）を使用して構成できます。

`orion-ejb-jar.xml` ファイルの構成は、OC4J 固有のアノテーションを使用して設定された対応する構成をオーバーライドします。

詳細は、2-31 ページの「メッセージ・サービス構成オプション : アノテーションまたは XML の選択と属性またはアクティブ化構成プロパティの選択」を参照してください。

アノテーションの使用方法

EJB 3.0 メッセージドリブン Bean の OC4J 固有のデプロイ・オプションは、OC4J 固有の `@MessageDrivenDeployment` アノテーションを使用して指定できます。

例 10-16 に、`@MessageDrivenDeployment` アノテーションを使用して EJB 3.0 メッセージドリブン Bean の OC4J 固有のデプロイ・オプションを構成する方法を示します。`@MessageDrivenDeployment` の属性の詳細は、表 A-3 を参照してください。

`@MessageDriven` アノテーションの `activationConfig` 属性設定（10-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」を参照）は、`@MessageDrivenDeployment` の属性を使用してアクティブ化構成プロパティを設定することでオーバーライドできます。アノテーション構成は、デプロイ XML を使用してオーバーライドできます（10-21 ページの「デプロイ XML の使用方法」を参照）。

例 10-16 @MessageDrivenDeployment

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.ejb.ActivationConfigProperty;
import javax.annotation.Resource;

@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="messageListenerInterface",
            propertyValue="javax.jms.MessageListener"),
        @ActivationConfigProperty(
            propertyName="connectionFactoryJndiName",
            propertyValue="jms/TopicConnectionFactory"),
         @ActivationConfigProperty(
            propertyName="destinationName",
            propertyValue="jms/demoTopic"),
         @ActivationConfigProperty(
            propertyName="destinationType",
            propertyValue="javax.jms.Topic"),
         @ActivationConfigProperty(
            propertyName="messageSelector",
            propertyValue="RECIPIENT = 'MDB'")
    }
)
@MessageDrivenDeployment(
    maxInstances=10,
    poolCacheTimeout=30
)
public class MessageLogger implements MessageListener, TimedObject {
    @Resource javax.ejb.MessageDrivenContext mc;

    public void onMessage(Message message) {
    ...
    }
```

```
    public void ejbTimeout(Timer timer) {
    ...
    }
}
```

デプロイ XML の使用方法

例 10-17 に示すように、メッセージドリブン Bean の OC4J 固有のデプロイ・オプションは、orion-ejb-jar.xml ファイルの <message-driven-deployment> 要素を使用して指定できます。<message-driven-deployment> 要素の詳細は、A-19 ページの「<message-driven-deployment>」を参照してください。

例 10-17 orion-ejb-jar.xml ファイルの <message-driven-deployment> 要素

```
<?xml version="1.0" encoding="utf-8"?>
<orion-ejb-jar
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion-ejb-jar-10_
0.xsd"
    deployment-version="10.1.3.1.0"
    deployment-time="10b1fb5cdd0"
    schema-major-version="10"
    schema-minor-version="0"
>
    <enterprise-beans>
        <message-driven-deployment
            name="MessageLogger"
            max-instances="10"
            cache-timeout="30"
            ...
        >
        </message-driven-deployment>
    ...
    </enterprise-beans>
    ...
</orion-ejb-jar>
```

第 V 部

EJB 2.1 セッション Bean

第 V 部では、EJB 2.1 セッション Bean の実装および構成の手順に関する情報を示します。概念的な情報は、第 I 部「EJB の概要」を参照してください。

第 V 部は次の各章で構成されています。

- 第 11 章「EJB 2.1 セッション Bean の実装」
- 第 12 章「EJB 2.1 セッション Bean の使用方法」

11

EJB 2.1 セッション Bean の実装

この章では、次のような EJB 2.1 セッション Bean の実装方法を説明します。

- 11-2 ページの「EJB 2.1 ステートレス・セッション Bean の実装」
- 11-4 ページの「EJB 2.1 ステートフル・セッション Bean の実装」

> **注意：** EJB コード例は、`http://www.oracle.com/technology/tech/java/oc4j/demos` からダウンロードできます。

詳細は、次を参照してください。

- 1-30 ページの「セッション Bean とは」
- 第 12 章「EJB 2.1 セッション Bean の使用方法」

EJB 2.1 ステートレス・セッション Bean の実装

表 11-1 に、EJB 2.1 ステートレス・セッション Bean の重要な構成要素をまとめ、次の手順でこれらの構成要素の実装方法を説明します。一般的な実装は、11-3 ページの「Java の使用方法」を参照してください。詳細は、1-31 ページの「ステートレス・セッション Bean とは」を参照してください。

表 11-1 EJB 2.1 ステートレス・セッション Bean の構成要素

構成要素	説明
ホーム・インタフェース（リモートまたはローカル）	`javax.ejb.EJBHome` および `javax.ejb.EJBLocalHome` を拡張し、引数を取らない 1 つの `create()` ファクトリ・メソッド、および 1 つの `remove()` メソッドを必要とします。
コンポーネント・インタフェース（リモートまたはローカル）	リモート・インタフェースの場合は `javax.ejb.EJBObject` を拡張し、ローカル・インタフェースの場合は `javax.ejb.EJBLocalObject` を拡張します。また、Bean 実装で実装されるビジネス・ロジック・メソッドを定義します。
`TimedObject` インタフェース	オプションで、`javax.ejb.TimedObject` インタフェースを実装します。詳細は、2-37 ページの「EJB タイマー・サービスについて」を参照してください。
Bean の実装	`SessionBean` を実装します。このクラスは、パブリックとして宣言する必要があり、パブリックで空のデフォルト・コンストラクタを含み、`finalize()` メソッドは使用しません。また、コンポーネント・インタフェースで定義されたメソッドを実装します。ホーム・インタフェースの `create()` メソッドに一致するような、引数を持たない 1 つの `ejbCreate` メソッドが含まれている必要があります。`ejbRemove` などのコンテナ・サービス・メソッド用の空の実装を含めます。

1. Bean のホーム・インタフェースを作成します（11-7 ページの「ホーム・インタフェースの実装」を参照）。

   リモート・ホーム・インタフェースは、クライアントが Bean をインスタンス化するためにリモートで起動できる `create` メソッドを定義します。ローカル・ホーム・インタフェースは、Bean をインスタンス化するために同一 JVM 上の Bean がローカルで起動できる `create` メソッドを定義します。

   a. リモート・ホーム・インタフェースを作成するには、`javax.ejb.EJBHome` を拡張します（11-8 ページの「リモート・ホーム・インタフェースの実装」を参照）。

   b. ローカル・ホーム・インタフェースを作成するには、`javax.ejb.EJBLocalHome` を拡張します（11-9 ページの「ローカル・ホーム・インタフェースの実装」を参照）。

2. Bean のコンポーネント・インタフェースを作成します（11-10 ページの「コンポーネント・インタフェースの実装」を参照）。

   リモート・コンポーネント・インタフェースは、クライアントがリモートで起動できるビジネス・メソッドを宣言します。ローカル・インタフェースは、同一 JVM 上の Bean がローカルで起動できるビジネス・メソッドを宣言します。

   a. リモート・コンポーネント・インタフェースを作成するには、`javax.ejb.EJBObject` を拡張します（11-10 ページの「リモート・コンポーネント・インタフェースの実装」を参照）。

   b. ローカル・コンポーネント・インタフェースを作成するには、`javax.ejb.EJBLocalObject` を拡張します（11-11 ページの「ローカル・コンポーネント・インタフェースの実装」を参照）。

3. 次のようにステートレス・セッション Bean を実装します。

   a. ホーム・インタフェースの `create` メソッドに一致する、パラメータを持たない 1 つの `ejbCreate` メソッドを実装します。

   b. ホームおよびコンポーネント・インタフェースで宣言されたビジネス・メソッドを実装します。

   c. `javax.ejb.SessionBean` インタフェースを実装して、定義するコンテナ・コールバック・メソッドを実装します（12-4 ページの「EJB 2.1 セッション Bean のライフ・サイクル・コールバック・メソッドの構成」を参照）。

d. SessionContext のインスタンスを取得する setSessionContext メソッドを実装します（11-11 ページの「setSessionContext メソッドの実装」を参照）。

ステートレス・セッション Bean では、このメソッドは通常は何も行いません（実際には SessionContext をセッション Bean の状態に追加しません）。

4. Bean 実装と一致するように ejb-jar.xml ファイルを構成します（11-4 ページの「デプロイ XML の使用方法」を参照）。

Java の使用方法

例 11-1 に、ステートレス・セッション Bean の一般的な実装を示します。

例 11-1 EJB 2.1 ステートレス・セッション Bean の実装

```
package hello;
import javax.ejb.*;

public class HelloBean implements SessionBean {

    /* ---------------------------------------
    * Begin business methods. The following methods
    * are called by the client code.
    * ------------------------------------- */

    public String sayHello(String myName) throws EJBException {
        return ("Hello " + myName);
    }

    /* ---------------------------------------
    * Begin private methods. The following methods
    * are used internally
    * ------------------------------------- */

...

    /* -----------------------------------------------------
    * Begin EJB-required methods. The following methods are called
    * by the container, and never called by client code
    * ----------------------------------------------------- */

    public void ejbCreate() throws CreateException {
        // when bean is created
    }

    public void setSessionContext(SessionContext ctx) {
    }

// Life Cycle Methods

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void ejbCreate() {
    }

    public void ejbRemove() {
    }
}
```

デプロイ XML の使用方法

例 11-2 に、例 11-1 に示したステートレス・セッション Bean に対応する ejb-jar.xml セッション要素を示します。

例 11-2 ステートレス・セッション Bean の ejb-jar.xml

```
...
    <enterprise-beans>
        <session>
            <ejb-name>Hello</ejb-name>
            <home>hello.HelloHome</home>
            <remote>hello.Hello</remote>
            <ejb-class>hello.HelloBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
...
```

デプロイ・ファイルの詳細は、第 26 章「デプロイメント・ディスクリプタ・ファイルの構成」を参照してください。

EJB 2.1 ステートフル・セッション Bean の実装

表 11-2 に、EJB 2.1 ステートフル・セッション Bean の重要な構成要素をまとめ、次の手順でこれらの構成要素の実装方法を説明します。一般的な実装は、11-6 ページの「Java の使用方法」を参照してください。詳細は、1-32 ページの「ステートフル・セッション Bean とは」を参照してください。

表 11-2 EJB 2.1 ステートフル・セッション Bean の構成要素

構成要素	説明
ホーム・インタフェース（リモートまたはローカル）	javax.ejb.EJBHome および javax.ejb.EJBLocalHome を拡張し、1 つ以上の create() ファクトリ・メソッドおよび 1 つの remove() メソッドを必要とします。
コンポーネント・インタフェース（リモートまたはローカル）	リモート・インタフェースの場合は javax.ejb.EJBObject を拡張し、ローカル・インタフェースの場合は javax.ejb.EJBLocalObject を拡張します。また、Bean 実装で実装されるビジネス・ロジック・メソッドを定義します。
Bean の実装	SessionBean を実装します。このクラスは、パブリックとして宣言する必要があり、パブリックで空のデフォルト・コンストラクタを含み、finalize メソッドは使用しません。また、リモート・インタフェースで定義されたメソッドを実装します。ホーム・インタフェースで定義された create メソッドに対応する ejbCreate メソッドが含まれている必要があります。つまり、各 ejbCreate メソッドは、パラメータ・シグネチャによって、ホーム・インタフェースで定義されている create メソッドに一致している必要があります。ejbRemove などのコンテナ・サービス・メソッドを実装します。また、オプションでコンテナ管理のトランザクション用の SessionSynchronization インタフェースを実装します。これには、afterBegin、beforeCompletion および afterCompletion が含まれます。

1. Bean のホーム・インタフェースを作成します（11-7 ページの「ホーム・インタフェースの実装」を参照）。

   リモート・ホーム・インタフェースは、クライアントが Bean をインスタンス化するためにリモートで起動できる `create` メソッドを定義します。ローカル・ホーム・インタフェースは、Bean をインスタンス化するために同一 JVM 上の Bean がローカルで起動できる `create` メソッドを定義します。

   a. リモート・ホーム・インタフェースを作成するには、`javax.ejb.EJBHome` を拡張します（11-8 ページの「リモート・ホーム・インタフェースの実装」を参照）。

   b. ローカル・ホーム・インタフェースを作成するには、`javax.ejb.EJBLocalHome` を拡張します（11-9 ページの「ローカル・ホーム・インタフェースの実装」を参照）。

2. Bean のコンポーネント・インタフェースを作成します（11-10 ページの「コンポーネント・インタフェースの実装」を参照）。

   リモート・コンポーネント・インタフェースは、クライアントがリモートで起動できるビジネス・メソッドを宣言します。ローカル・インタフェースは、同一 JVM 上の Bean がローカルで起動できるビジネス・メソッドを宣言します。

   a. リモート・コンポーネント・インタフェースを作成するには、`javax.ejb.EJBObject` を拡張します（11-10 ページの「リモート・コンポーネント・インタフェースの実装」を参照）。

   b. ローカル・コンポーネント・インタフェースを作成するには、`javax.ejb.EJBLocalObject` を拡張します（11-11 ページの「ローカル・コンポーネント・インタフェースの実装」を参照）。

3. 次のようにステートフル・セッション Bean を実装します。

   a. ホーム・インタフェースの `create` メソッドに一致する `ejb<METHOD>` メソッドを実装します。

      ステートフル・セッション Bean の場合は、ホーム・インタフェースの各 `create` メソッドに対応する引数リストを `ejbCreate` メソッドに提供します。

   b. ホームおよびコンポーネント・インタフェースで宣言されたビジネス・メソッドを実装します。

   c. `javax.ejb.SessionBean` インタフェースを実装して、定義するコンテナ・コールバック・メソッドを実装します（12-4 ページの「EJB 2.1 セッション Bean のライフ・サイクル・コールバック・メソッドの構成」を参照）。

   d. `SessionContext` のインスタンスを取得する `setSessionContext` メソッドを実装します（11-11 ページの「setSessionContext メソッドの実装」を参照）。

      ステートフル・セッション Bean では、このメソッドは通常は `SessionContext` をセッション Bean の状態に追加します。

4. Bean 実装と一致するように `ejb-jar.xml` ファイルを構成します（11-7 ページの「デプロイ XML の使用方法」を参照）。

Java の使用方法

例 11-3 に、ステートフル・セッション Bean の一般的な実装を示します。

例 11-3 EJB 2.1 ステートフル・セッション Bean の実装

```
package hello;
import javax.ejb.*;

public class HelloBean implements SessionBean {
    /* ---------------------------------------
    * State
    * ------------------------------------- */

    private SessionContext ctx;
    private Collection messages;
    private String defaultMessage = "Hello, World!";

    /* ---------------------------------------
    * Begin business methods. The following methods
    * are called by the client code.
    * ------------------------------------- */

    public String sayHello(String myName) throws EJBException {
        return ("Hello " + myName);
    }

    public String sayHello() throws EJBException {
        return defaultMessage;
    }

    /* ---------------------------------------
    * Begin private methods. The following methods
    * are used internally.
    * ------------------------------------- */

...

    /* -----------------------------------------------------
    * Begin EJB-required methods. The following methods are called
    * by the container, and never called by client code.
    * ----------------------------------------------------- */

    public void ejbCreate() throws CreateException {
        // when bean is created
    }

    public void ejbCreate(String message) throws CreateException {
        this.defaultMessage = message;
    }

    public void ejbCreate(Collection messages) throws CreateException {
        this.messages = messages;
    }

    public void setSessionContext(SessionContext ctx) {
        this.ctx = ctx;
    }

// Life Cycle Methods

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }
```

```
    public void ejbCreate() {
    }

    public void ejbRemove() {
    }
}
```

デプロイ XML の使用方法

例 11-4 に、例 11-3 に示したステートフル・セッション Bean に対応する ejb-jar.xml セッション要素を示します。

例 11-4 ステートフル・セッション Bean の ejb-jar.xml

```
...
    <enterprise-beans>
        <session>
            <ejb-name>Hello</ejb-name>
            <home>hello.HelloHome</home>
            <remote>hello.Hello</remote>
            <ejb-class>hello.HelloBean</ejb-class>
            <session-type>Stateful</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
...
```

デプロイ・ファイルの詳細は、第 26 章「デプロイメント・ディスクリプタ・ファイルの構成」を参照してください。

ホーム・インタフェースの実装

ホーム・インタフェース（リモートおよびローカル）は、セッション Bean インスタンスの作成に使用され、Bean の `create` メソッドを定義します。表 11-3 に示すように、定義する `create` メソッドのタイプは、作成しているセッション bean のタイプによって決まります。

表 11-3 ホーム・インタフェースの create メソッド

セッション Bean タイプ	create メソッド
ステートレス・セッション Bean	パラメータなしの 1 つの `create` メソッドのみ。
ステートフル・セッション Bean	Bean の状態を定義するパラメータを持つ 1 つ以上の create メソッド。

各 `create` メソッドにつき、対応する `ejbCreate` メソッドを Bean 実装で定義します。

リモート・ホーム・インタフェースの実装

リモート・クライアントは、リモート・インタフェースを介して EJB を起動します。クライアントは、リモート・ホーム・インタフェースで宣言された create メソッドを起動します。コンテナは、Bean 実装内の、適切なパラメータ・シグネチャを持つ ejbCreate メソッドにクライアント・コールを渡します。リモート・ホーム・インタフェースを開発するための要件は次のとおりです。

- リモート・ホーム・インタフェースでは、javax.ejb.EJBHome インタフェースを拡張する必要があります。
- すべての create メソッドで、次の例外をスローできます。
  - javax.ejb.CreateException
  - javax.ejb.RemoteException
  - オプションのアプリケーション例外
- どの create メソッドでも、次の例外をスローできません。
  - javax.ejb.EJBException
  - java.lang.RunTimeException

例 11-5 に、ステートレス・セッション Bean の HelloHome というリモート・ホーム・インタフェースを示します。

例 11-5 ステートレス・セッション Bean のリモート・ホーム・インタフェース

```
package hello;

import javax.ejb.*;
import java.rmi.*;

public interface HelloHome extends EJBHome {
  public Hello create() throws CreateException, RemoteException;
}
```

例 11-6 に、ステートフル・セッション Bean の HelloHome というリモート・ホーム・インタフェースを示します。様々な create メソッドに渡される引数を使用して、セッション Bean の状態を初期化します。

例 11-6 ステートフル・セッション Bean のリモート・ホーム・インタフェース

```
package hello;

import javax.ejb.*;
import java.rmi.*;

public interface HelloHome extends EJBHome {
  public Hello create() throws CreateException, RemoteException;
  public Hello create(String message) throws CreateException, RemoteException;
  public Hello create(Collection messages) throws CreateException, RemoteException;
}
```

ローカル・ホーム・インタフェースの実装

EJB は、同じコンテナに存在するクライアントからローカルでコールできます。したがって、同一 JVM 上の Bean、JSP またはサーブレットは、ローカル・ホーム・インタフェースで宣言された `create` メソッドを起動します。コンテナは、Bean 実装内の、適切なパラメータ・シグネチャを持つ `ejbCreate` メソッドにクライアント・コールを渡します。ローカル・ホーム・インタフェースを開発するための要件は、次のとおりです。

- ローカル・ホーム・インタフェースでは、`javax.ejb.EJBLocalHome` インタフェースを拡張する必要があります。
- すべての `create` メソッドで、次の例外をスローできます。
  - `javax.ejb.CreateException`
  - `javax.ejb.RemoteException`
  - オプションのアプリケーション例外
- どの `create` メソッドでも、次の例外をスローできません。
  - `javax.ejb.EJBException`
  - `java.lang.RunTimeException`

例 11-7 に、ステートレス・セッション Bean の `HelloLocalHome` というローカル・ホーム・インタフェースを示します。

例 11-7 ステートレス・セッション Bean のローカル・ホーム・インタフェース

```
package hello;

import javax.ejb.*;

public interface HelloLocalHome extends EJBLocalHome {
 public HelloLocal create() throws CreateException;
}
```

例 11-8 に、ステートフル・セッション Bean の `HelloLocalHome` というローカル・ホーム・インタフェースを示します。様々な `create` メソッドに渡される引数を使用して、セッション Bean の状態を初期化します。

例 11-8 ステートフル・セッション Bean のローカル・ホーム・インタフェース

```
package hello;

import javax.ejb.*;

public interface HelloLocalHome extends EJBLocalHome {
  public HelloLocal create() throws CreateException;
  public HelloLocal create(String message) throws CreateException;
  public HelloLocal create(Collection messages) throws CreateException;
}
```

コンポーネント・インタフェースの実装

コンポーネント・インタフェースでは、クライアントから起動可能な Bean のビジネス・メソッドを定義します。

リモート・コンポーネント・インタフェースの実装

リモート・インタフェースでは、リモート・クライアントによって起動可能なビジネス・メソッドを定義します。リモート・コンポーネント・インタフェースを開発するための要件は、次のとおりです。

- Bean のリモート・コンポーネント・インタフェースは、`javax.ejb.EJBObject` インタフェースを拡張する必要があり、そのメソッドは `java.rmi.RemoteException` 例外をスローする必要があります。
- リモート・インタフェースとそのメソッドは、リモート・クライアントに対する `public` として宣言する必要があります。
- リモート・コンポーネント・インタフェース、すべてのメソッド・パラメータおよび戻り型はシリアライズ可能である必要があります。一般的に、RMI は両側のオブジェクトをマーシャリングおよびアンマーシャリングするため、クライアントと EJB の間で受渡しされるオブジェクトは、すべてシリアライズ可能である必要があります。
- シリアライズ可能であれば、どのような例外でもクライアントにスロー可能です。`EJBException` および `RemoteException` を含めた実行時例外は、リモート実行時例外としてクライアントに転送されます。

例 11-9 に、`Hello` というリモート・コンポーネント・インタフェースとその定義済のメソッドを示します。各メソッドは、対応するセッション Bean で実装されます。

例 11-9 EJB 2.1 セッション Bean のリモート・コンポーネント・インタフェース

```
package hello;

import javax.ejb.*;
import java.rmi.*;

public interface Hello extends EJBObject {
    public String sayHello(String myName) throws RemoteException;
    public String sayHello() throws RemoteException;
}
```

ローカル・コンポーネント・インタフェースの実装

ローカル・コンポーネント・インタフェースでは、ローカル（同一 JVM 上の）クライアントから起動可能な Bean のビジネス・メソッドを定義します。ローカル・コンポーネント・インタフェースを開発するための要件は、次のとおりです。

- Bean のローカル・コンポーネント・インタフェースでは、`javax.ejb.EJBLocalObject` インタフェースを拡張する必要があります。
- ローカル・コンポーネント・インタフェースとそのメソッドは、`public` として宣言します。

例 11-10 に、`HelloLocal` というローカル・コンポーネント・インタフェースとその定義済のメソッドを示します。各メソッドは、対応するセッション Bean で実装されます。

例 11-10 EJB 2.1 セッション Bean のローカル・コンポーネント・インタフェース

```
package hello;

import javax.ejb.*;

public interface HelloLocal extends EJBLocalObject {
    public String sayHello(String myName);
    public String sayHello();
}
```

setSessionContext メソッドの実装

このメソッドを使用して、Bean のコンテキストの参照を取得します。セッション Bean には、コンテナによって維持され、Bean から使用可能なセッション・コンテキストが存在します。Bean は、セッション・コンテキスト内のメソッドを使用して、コンテナへのコールバック・リクエストを送信できます。

コンテナは、Bean をインスタンス化した後、`setSessionContext` メソッドを起動して、Bean からセッション・コンテキストを取得できるようにします。コンテナは、トランザクション・コンテキストからはこのメソッドをコールしません。この時点で Bean がセッション・コンテキストを保存しなかった場合、Bean は二度とセッション・コンテキストにアクセスできなくなります。

コンテナはこのメソッドをコールする際、`SessionContext` オブジェクトの参照を Bean に渡します。Bean は、この参照を後の使用のために格納できます。

例 11-11 に、セッション・コンテキストを `sessctx` 変数に格納するセッション Bean を示します。

例 11-11 setSessionContext メソッドの実装

```
import javax.ejb.*;

public class myBean implements SessionBean {
   SessionContext sessctx;

   public void setSessionContext(SessionContext ctx) {
      sessctx = ctx;   // session context is stored in instance variable
   }
   // other methods in the bean
}
```

12

EJB 2.1 セッション Bean の使用方法

この章では、EJB 2.1 セッション Bean を使用するために構成する必要のある様々なオプションについて説明します。

表 12-1 に、これらのオプションをリストし、基本オプション（ほとんどのアプリケーションに適用可能）であるか拡張オプション（より特殊なアプリケーションに適用可能）であるかを示します。

詳細は、次を参照してください。

- 1-30 ページの「セッション Bean とは」
- 第 11 章「EJB 2.1 セッション Bean の実装」

表 12-1 EJB 2.1 セッション Bean の構成オプション

オプション	タイプ
12-2 ページの「非アクティブ化の構成」	拡張
12-2 ページの「非アクティブ化基準の構成」	拡張
12-4 ページの「非アクティブ化の場所の構成」	拡張
31-5 ページの「Bean インスタンスのプール・サイズの構成」	基本
31-7 ページの「セッション Bean の Bean インスタンス・プール・タイムアウトの構成」	拡張
21-8 ページの「セッション Bean のトランザクション・タイムアウトの構成」	拡張
12-4 ページの「EJB 2.1 セッション Bean のライフ・サイクル・コールバック・メソッドの構成」	基本

非アクティブ化の構成

ステートフル・セッション Bean の非アクティブ化は有効および無効にできます（12-2 ページの「デプロイ XML の使用方法」を参照）。

次のいずれかの理由で非アクティブ化を無効にすることを選択できます。

- 互換性のないオブジェクト・タイプ : 非アクティブ化でサポートされているオブジェクト・タイプ（1-35 ページの「非アクティブ化できるオブジェクト・タイプ」を参照）でステートフル・セッション Bean の非一時属性を表すことができない場合は、非アクティブ化を無効にすることにより、メモリー消費の増加と引換えに他のオブジェクト・タイプを使用できます。
- パフォーマンス : 非アクティブ化によりアプリケーションにパフォーマンスの問題が発生していると判断した場合は、非アクティブ化を無効にすることにより、メモリー消費の増加と引換えにパフォーマンスを改善できます。
- 2 次ストレージの制限 : 十分な 2 次ストレージを提供できない場合は（12-4 ページの「非アクティブ化の場所の構成」を参照）、非アクティブ化を無効にすることにより、メモリー消費の増加と引換えに 2 次ストレージ所要量を削減できます。

詳細は、次を参照してください。

- 1-35 ページの「ステートフル・セッション Bean の非アクティブ化が発生する状況」
- 12-2 ページの「非アクティブ化基準の構成」
- 12-4 ページの「非アクティブ化の場所の構成」

デプロイ XML の使用方法

表 12-2 に、`server.xml` ファイルの要素 `sfsb-config` で非アクティブ化を構成するための属性、値およびデフォルトをリストします。

表 12-2 server.xml 要素 sfsb-config の非アクティブ化構成

属性	値	デフォルト
enable-passivation	`true`、`false`	`true`

非アクティブ化基準の構成

OC4J がステートフル・セッション Bean を非アクティブ化する条件を指定できます（12-3 ページの「デプロイ XML の使用方法」を参照）。

詳細は、次を参照してください。

- 1-35 ページの「ステートフル・セッション Bean の非アクティブ化が発生する状況」
- 12-2 ページの「非アクティブ化の構成」
- 12-4 ページの「非アクティブ化の場所の構成」

デプロイ XML の使用方法

表 12-3 に、orion-ejb-jar.xml ファイルの要素 session-deployment で非アクティブ化基準を構成するための属性、値およびデフォルトをリストします。

表 12-3 orion-ejb-jar.xml の要素 session-deployment の非アクティブ化基準

属性	値	デフォルト
idletime	非アクティブ化が行われるまでの秒数（正の整数）。 この基準を無効にするには、値 never を指定します。	300
memory-threshold	非アクティブ化が行われるまでに消費できる JVM メモリーの割合。 この基準を無効にするには、値 never を指定します。	80
max-instances	メモリー内に存在できるインスタンス化またはプールされた Bean インスタンスの最大数（正の整数）。 この値に達すると、OC4J は最低使用頻度（LRU）アルゴリズムを使用して Bean を非アクティブ化しようとします。Bean インスタンスの数を無限に許可する場合は、max-instances 属性を 0（ゼロ）に設定できます。デフォルトは 0（ゼロ）で、無限を意味します。この属性は、ステートレス・セッション Bean およびステートフル・セッション Bean の両方に適用されます。 インスタンス・プーリングを無効にするには、max-instances を負の数に設定します。これにより、EJB コールの開始時に新規インスタンスを作成し、コールの終了時に解放します。 詳細は、31-5 ページの「Bean インスタンスのプール・サイズの構成」を参照してください。	0（無制限）
max-instances-threshold	非アクティブ化が行われるまでにメモリー内に存在できる Bean の max-instances 数に対する割合。 パーセンテージとして解釈される整数を指定します。max-instances を 100、max-instances-threshold を 90% に定義した場合は、アクティブ Bean インスタンス数が 90 以上になると、Bean の非アクティブ化が発生します。デフォルトは 90% です。 この属性を無効にするには、never を指定します。	90
passivate-count	いずれかのリソースしきい値（memory-threshold または max-instances-threshold）に達した場合に非アクティブ化する Bean の数（正の整数）。 Bean の非アクティブ化は、最低使用頻度アルゴリズムを使用して実行されます。 このオプションを無効にするには、値 0 を指定します。	max-instances の 1/3
resource-check-interval	OC4J がリソースしきい値（memory-threshold または max-instances-threshold）をチェックする頻度（正の整数の秒数）。 このオプションを無効にするには、値 never を指定します。	180

非アクティブ化の場所の構成

OC4J が非アクティブ化時にステートフル・セッション Bean をシリアライズするディレクトリおよびファイル名を指定できます（12-4 ページの「デプロイ XML の使用方法」を参照）。

詳細は、次を参照してください。

- 1-36 ページの「非アクティブ化されたステートフル・セッション Bean の格納場所」
- 12-2 ページの「非アクティブ化の構成」
- 12-2 ページの「非アクティブ化基準の構成」

デプロイ XML の使用方法

表 12-4 に、`orion-ejb-jar.xml` ファイルの要素 `session-deployment` で非アクティブ化の場所を構成するための属性、値およびデフォルトをリストします。

表 12-4 orion-ejb-jar.xml の要素 session-deployment の非アクティブ化の場所の構成

属性	値	デフォルト
persistence-filename	OC4J が非アクティブ化中に Bean インスタンスをシリアライズするファイルの完全修飾パスおよびファイル名	*<OC4J_HOME>*¥j2ee¥home¥application-deployments¥persistence

EJB 2.1 セッション Bean のライフ・サイクル・コールバック・メソッドの構成

次に、`javax.ejb.SessionBean` インタフェースでの指定に従って、セッション Bean が実装する必要のある EJB 2.1 ライフ・サイクル・メソッドを示します（12-4 ページの「Java の使用方法」を参照）。

- `ejbCreate`
- `ejbActivate`（ステートフル・セッション Bean のみ）
- `ejbPassivate`（ステートフル・セッション Bean のみ）
- `ejbRemove`
- `setSessionContext`

注意： EJB 2.1 を使用する場合は、すべてのセッション Bean コールバック・メソッドを実装する必要があります。何もアクションを行う必要がない場合、またはコールバック・メソッドをセッション Bean に適用しない場合は、空のメソッドを実装します。

詳細は、次を参照してください。

- 1-31 ページの「ステートレス・セッション Bean のライフ・サイクル」
- 1-33 ページの「ステートフル・セッション Bean のライフ・サイクル」

Java の使用方法

例 12-1 では、EJB 2.1 セッション Bean のコールバック・メソッドの実装方法を説明します。

例 12-1 EJB 2.1 セッション Bean のコールバック・メソッドの実装

```
public void ejbActivate() {
    // when bean is activated
}
```

第 VI 部

EJB 2.1 エンティティ Bean

第 VI 部では、EJB 2.1 エンティティ Bean およびエンティティ Bean 問合せの実装および構成の手順に関する情報を示します。概念的な情報は、第 I 部「EJB の概要」を参照してください。

第 VI 部は次の各章で構成されています。

- 第 13 章「EJB 2.1 エンティティ Bean の実装」
- 第 14 章「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の使用方法」
- 第 15 章「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の使用方法」
- 第 16 章「EJB 2.1 問合せの実装」

13

EJB 2.1 エンティティ Bean の実装

この章では、次のような EJB 2.1 エンティティ Bean の実装方法を説明します。

- 13-2 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の実装」
- 13-7 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の実装」

> **注意：** EJB コード例は、`http://www.oracle.com/technology/tech/java/oc4j/demos` からダウンロードできます。

詳細は、次を参照してください。

- 1-44 ページの「EJB 2.1 エンティティ Bean とは」
- 第 14 章「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の使用方法」
- 第 15 章「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の使用方法」

コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の実装

表 13-1 に、コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の重要な構成要素をまとめ、次の手順でこれらの構成要素の実装方法を説明します。一般的な実装は、13-4 ページの「Java の使用方法」を参照してください。詳細は、1-45 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean とは」を参照してください。

表 13-1 コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の構成要素

構成要素	説明
ホーム・インタフェース（リモートまたはローカル）	リモート・ホーム・インタフェースの `javax.ejb.EJBHome`、ローカル・ホーム・インタフェースの `javax.ejb.EJBLocalHome` を拡張し、引数を取らない 1 つの `create` ファクトリ・メソッド、および 1 つの `remove` メソッドを必要とします。
コンポーネント・インタフェース（リモートまたはローカル）	リモート・インタフェースの場合は `javax.ejb.EJBObject` を拡張し、ローカル・インタフェースの場合は `javax.ejb.EJBLocalObject` を拡張します。また、Bean 実装で実装されるビジネス・ロジック・メソッドを定義します。
Bean の実装	`EntityBean` を実装します。このクラスは、パブリックとして宣言する必要があり、パブリックで空のデフォルト・コンストラクタを含み、`finalize` メソッドは使用しません。また、コンポーネント・インタフェースで定義されたメソッドを実装します。ホーム・インタフェースの `create` メソッドに一致する 1 つ以上の `ejbCreate` メソッドが含まれている必要があります。`ejbRemove` などのコンテナ・サービス・メソッド用の空の実装を含めます。

1. Bean のホーム・インタフェースを作成します（13-19 ページの「EJB 2.1 ホーム・インタフェースの実装」を参照）。

   リモート・ホーム・インタフェースは、クライアントが Bean をインスタンス化するためにリモートで起動できる `create` および `finder` メソッドを定義します。ローカル・ホーム・インタフェースは、Bean をインスタンス化するために同一 JVM 上の Bean がローカルで起動できる `create` および `finder` メソッドを定義します。

   finder の詳細は、1-56 ページの「finder メソッドについて」を参照してください。

   a. リモート・ホーム・インタフェースを作成するには、`javax.ejb.EJBHome` を拡張します（13-19 ページの「リモート・ホーム・インタフェースの実装」を参照）。

   b. ローカル・ホーム・インタフェースを作成するには、`javax.ejb.EJBLocalHome` を拡張します（13-20 ページの「ローカル・ホーム・インタフェースの実装」を参照）。

2. Bean のコンポーネント・インタフェースを作成します（13-20 ページの「EJB 2.1 コンポーネント・インタフェースの実装」を参照）。

   リモート・コンポーネント・インタフェースは、クライアントがリモートで起動できるビジネス・メソッドを宣言します。ローカル・インタフェースは、同一 JVM 上の Bean がローカルで起動できるビジネス・メソッドを宣言します。

   a. リモート・コンポーネント・インタフェースを作成するには、`javax.ejb.EJBObject` を拡張します（13-20 ページの「リモート・コンポーネント・インタフェースの実装」を参照）。

   b. ローカル・コンポーネント・インタフェースを作成するには、`javax.ejb.EJBLocalObject` を拡張します（13-21 ページの「ローカル・コンポーネント・インタフェースの実装」を参照）。

3. Bean の主キーを定義します（14-2 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の主キーの構成」を参照）。

   主キーはシリアライズ可能なクラスで、各エンティティ Bean インスタンスを識別します。単純なデータ型クラス（`java.lang.String` など）を使用したり、複合クラス（主キーのコンポーネントとして複数のオブジェクトを持つクラスなど）を定義できます。

4. 次のようにコンテナ管理の永続性を備えたエンティティ Bean を実装します。

   a. ホーム・インタフェースで宣言された getter および setter メソッドに対応する抽象 getter および setter メソッドを実装します。

      コンテナ管理の永続性を備えたエンティティ Bean の場合、コンテナが実装を行うため、getter および setter メソッドは `public abstract` です。

   b. ホームおよびコンポーネント・インタフェースで宣言したビジネス・メソッドを実装します（存在する場合）。これらの各メソッドのシグネチャは、Bean が `RemoteException` をスローしない場合を除き、リモートまたはローカル・インタフェースのシグネチャに一致している必要があります。ローカル・インタフェースおよびリモート・インタフェースは Bean 実装を使用するため、Bean 実装では `RemoteException` をスローできません。

      エンティティ Bean の場合、これらのメソッドはセッション Bean に委任されることがあります（1-30 ページの「セッション Bean とは」を参照）。

   c. ビジネス・ロジックに使用される Bean またはパッケージに対してプライベートであるメソッドを実装します。これには、パブリック・メソッドがリクエストされた作業の完了に使用するプライベート・メソッドも含まれます。

   d. ホーム・インタフェースで宣言された各 `create` メソッドに対応する `ejbCreate` メソッドを実装します。クライアントが `create` メソッドを起動すると、コンテナによって対応する `ejbCreate` メソッドが起動されます。

      すべての `ebjCreate` メソッドの戻り型は、Bean の主キーの型です。

      コンテナ管理の永続性を備えたエンティティ Bean の場合は、コンテナがデータベースに維持する値をクライアントが渡せるようにする create メソッドを提供します。

   e. 各 `javax.ejb.EntityBean` インタフェース・コンテナのコールバック・メソッドの空の実装を提供します。

      詳細は、14-17 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル・コールバック・メソッドの構成」を参照してください。

   f. （`EntityContext` のインスタンスを受け取る）`setEntityContext` メソッドおよび `unsetEntityContext` メソッドを実装します（13-21 ページの「setEntityContext および unsetEntityContext メソッドの実装」を参照）。

   g. オプションで、エンティティ Bean のビジネス・メソッドで使用する 0 個以上の `public abstract` の select メソッド（1-58 ページの「select メソッドについて」を参照）を定義します。

5. エンティティ Bean の適切なデータベース・スキーマ（表と列）を作成します。

   コンテナ管理の永続性を備えたエンティティ Bean の場合は、永続性属性をデータベースに格納する方法を指定するか、表の作成を管理するようにコンテナを構成できます。

   詳細は、次を参照してください。

   - 14-5 ページの「表および列情報の構成」
   - 14-5 ページの「自動的なデータベース表作成の構成」

6. Bean 実装と一致するように、また `data-sources.xml` ファイルで定義されているデータソースを参照するように `ejb-jar.xml` ファイルを構成します（13-6 ページの「デプロイ XML の使用方法」を参照）。

7. エンティティ Bean の構成を完了します（第 14 章「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の使用方法」を参照）。

Java の使用方法

例 13-1 に、コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の一般的な実装を示します。例 13-2 では対応するリモート・ホーム・インタフェースを示し、例 13-3 では対応するリモート・コンポーネント・インタフェースを示します。

例 13-1 コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の実装

```
package cmpapp;

import javax.ejb.*;
import java.rmi.*;

public abstract class EmployeeBean implements EntityBean {

    private EntityContext ctx;

    // container-managed persistent fields accessors
    public abstract Integer getEmpNo();
    public abstract void setEmpNo(Integer empNo);

    public abstract String getEmpName();
    public abstract void setEmpName(String empName);

    public abstract Float getSalary();
    public abstract void setSalary(Float salary);

    public void EmployeeBean() {
        // Empty constructor, don't initialize here but in the create().
        // passivate() may destroy these attributes in the case of pooling
    }

    public EmployeePK ejbCreate(Integer empNo, String empName, Float salary)
        throws CreateException {
        setEmpNo(empNo);
        setEmpName(empName);
        setSalary(salary);
        return new EmployeePK(empNo);
    }

    public void ejbPostCreate(Integer empNo, String empName, Float salary)
        throws CreateException {
        // when just after bean created
    }

    public void ejbStore() {
        // when bean persisted
    }

    public void ejbLoad() {
        // when bean loaded
    }

    public void ejbRemove() {
        // when bean removed
    }

    public void ejbActivate() {
        // when bean activated
    }

    public void ejbPassivate() {
        // when bean deactivated
    }

    public void setEntityContext(EntityContext ctx) {
```

```
        this.ctx = ctx;
    }

    public void unsetEntityContext() {
        this.ctx = null;
    }

}
```

例 13-2 EJB 2.1 CMP リモート・ホーム・インタフェース

```
package cmpapp;

import java.rmi.*;
import java.util.*;
import javax.ejb.*;

public interface EmployeeHome extends EJBHome {
     public Employee create(Integer empNo, String empName, Float salary)
         throws CreateException, RemoteException;

     public Employee findByPrimaryKey(EmployeePK pk)
         throws FinderException, RemoteException;

     public Collection findByName(String empName)
         throws FinderException, RemoteException;

     public Collection findAll()
         throws FinderException, RemoteException;
}
```

例 13-3 EJB 2.1 CMP リモート・コンポーネント・インタフェース

```
package cmpapp;

import javax.ejb.*;
import java.rmi.*;

public interface Employee extends EJBObject {
    // container-managed persistent fields accessors
    public Integer getEmpNo() throws RemoteException;
    public void setEmpNo(Integer empNo) throws RemoteException;

    public String getEmpName() throws RemoteException;
    public void setEmpName(String empName) throws RemoteException;

    public Float getSalary() throws RemoteException;
    public void setSalary(Float salary) throws RemoteException;
}
```

デプロイ XML の使用方法

例 13-4 に、例 13-1 に示したコンテナ管理の永続性を備えたエンティティ Bean に対応する ejb-jar.xml ファイルの entity 要素を示します。

例 13-4 コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の ejb-jar.xml

```
...
    <enterprise-beans>
        <entity>
            <description>no description</description>
            <display-name>EmployeeBean</display-name>
            <ejb-name>EmployeeBean</ejb-name>
            <home>cmpapp.EmployeeHome</home>
            <remote>cmpapp.Employee</remote>
            <ejb-class>cmpapp.EmployeeBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <cmp-version>2.x</cmp-version>
            <abstract-schema-name>EmployeeBean</abstract-schema-name>
            <prim-key-class>cmpapp.EmployeePK</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-field><field-name>empNo</field-name></cmp-field>
            <cmp-field><field-name>empName</field-name></cmp-field>
            <cmp-field><field-name>salary</field-name></cmp-field>
            <query>
                <description></description>
                <query-method>
                <method-name>findAll</method-name>
                <method-params/>
                </query-method>
                <ejb-ql>Select OBJECT(e) From EmployeeBean e</ejb-ql>
            </query>
            <query>
                <description></description>
                <query-method>
                <method-name>findByName</method-name>
                <method-params>
                <method-param>java.lang.String</method-param>
                </method-params>
                </query-method>
                <ejb-ql>Select OBJECT(e) From EmployeeBean e where e.empName = ?1</ejb-ql>
            </query>
        </entity>
    </enterprise-beans>
...
```

Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の実装

表 13-2 に、Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の重要な構成要素をまとめます。次の手順では、これらの構成要素の実装方法を説明します。一般的な実装は、13-9 ページの「Java の使用方法」を参照してください。詳細は、1-49 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean とは」を参照してください。

表 13-2 Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の構成要素

構成要素	説明
ホーム・インタフェース（リモートまたはローカル）	リモート・ホーム・インタフェースの `javax.ejb.EJBHome`、ローカル・ホーム・インタフェースの `javax.ejb.EJBLocalHome` を拡張し、引数を取らない 1 つの `create` ファクトリ・メソッド、および 1 つの `remove` メソッドを必要とします。
コンポーネント・インタフェース（リモートまたはローカル）	リモート・インタフェースの場合は `javax.ejb.EJBObject` を拡張し、ローカル・インタフェースの場合は `javax.ejb.EJBLocalObject` を拡張します。また、Bean 実装で実装されるビジネス・ロジック・メソッドを定義します。
Bean の実装	`EntityBean` を実装します。このクラスは、パブリックとして宣言する必要があり、パブリックで空のデフォルト・コンストラクタを含み、`finalize` メソッドは使用しません。また、コンポーネント・インタフェースで定義されたメソッドを実装します。ホーム・インタフェースの `create` メソッドに一致する 1 つ以上の `ejbCreate` メソッドが含まれている必要があります。`ejbStore`、`ejbLoad`、`ejbRemove` などのコンテナ・サービス・メソッド用の完全な実装を含めます。

1. Bean のホーム・インタフェースを作成します（13-19 ページの「EJB 2.1 ホーム・インタフェースの実装」を参照）。

   リモート・ホーム・インタフェースは、クライアントが Bean をインスタンス化するためにリモートで起動できる `create` メソッドを定義します。ローカル・ホーム・インタフェースは、Bean をインスタンス化するために同一 JVM 上の Bean がローカルで起動できる `create` メソッドを定義します。

   a. リモート・ホーム・インタフェースを作成するには、`javax.ejb.EJBHome` を拡張します（13-19 ページの「リモート・ホーム・インタフェースの実装」を参照）。

   b. ローカル・ホーム・インタフェースを作成するには、`javax.ejb.EJBLocalHome` を拡張します（13-20 ページの「ローカル・ホーム・インタフェースの実装」を参照）。

2. Bean のコンポーネント・インタフェースを作成します（13-20 ページの「EJB 2.1 コンポーネント・インタフェースの実装」を参照）。

   リモート・コンポーネント・インタフェースは、クライアントがリモートで起動できるビジネス・メソッドを宣言します。ローカル・インタフェースは、同一 JVM 上の Bean がローカルで起動できるビジネス・メソッドを宣言します。

   a. リモート・コンポーネント・インタフェースを作成するには、`javax.ejb.EJBObject` を拡張します（13-20 ページの「リモート・コンポーネント・インタフェースの実装」を参照）。

   b. ローカル・コンポーネント・インタフェースを作成するには、`javax.ejb.EJBLocalObject` を拡張します（13-21 ページの「ローカル・コンポーネント・インタフェースの実装」を参照）。

3. Bean の主キーを定義します（15-2 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の主キーの構成」を参照）。

   主キーはシリアライズ可能なクラスで、各エンティティ Bean インスタンスを識別します。単純なデータ型クラス（`java.lang.String` など）を使用したり、複合クラス（主キーのコンポーネントとして複数のオブジェクトを持つクラスなど）を定義できます。

4. 次のように Bean 管理の永続性を備えたエンティティ Bean を実装します。

   a. ホーム・インタフェースで宣言された get および set メソッドに対応する get および set メソッドの完全な実装を提供します。

      Bean 管理の永続性を備えたエンティティ Bean の場合、ユーザーが実装を行うため、getter および setter メソッドは public です。

   b. ホームおよびコンポーネント・インタフェースで宣言したビジネス・メソッドを実装します（存在する場合）。これらの各メソッドのシグネチャは、Bean が `RemoteException` をスローしない場合を除き、リモートまたはローカル・インタフェースのシグネチャに一致している必要があります。ローカル・インタフェースおよびリモート・インタフェースは Bean 実装を使用するため、Bean 実装では `RemoteException` をスローできません。

      エンティティ Bean の場合、これらのメソッドはセッション Bean に委任されることがあります（1-30 ページの「セッション Bean とは」を参照）。

   c. ビジネス・ロジックに使用される Bean またはパッケージに対してプライベートであるメソッドを実装します。これには、パブリック・メソッドがリクエストされた作業の完了に使用するプライベート・メソッドも含まれます。

   d. ホーム・インタフェースで宣言された各 `create` メソッドに対応する `ejbCreate` メソッドを実装します。クライアントが `create` メソッドを起動すると、コンテナによって対応する `ejbCreate` メソッドが起動されます。

      すべての `ebjCreate` メソッドの戻り型は、Bean の主キーの型です。

      Bean 管理の永続性を備えたエンティティ Bean の場合は、コンテナがデータベースに維持する値をクライアントが渡せるようにする create メソッドを提供します。データベースのインスタンスを作成するためにデータベースと（通常は直接 JDBC コールを通じて）対話する実装をユーザーが提供します。

      詳細は、13-16 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の ejbCreate メソッドの実装」を参照してください。

   e. 各 `javax.ejb.EntityBean` インタフェース・コンテナのコールバック・メソッドの完全な実装を提供します（15-8 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル・コールバック・メソッドの構成」を参照）。

      Bean 管理の永続性を備えたエンティティ Bean の場合は、データベースの永続性を管理するためにデータベースと（通常は直接 JDBC コールを通じて）対話するこれらの各メソッドの実装をユーザーが提供します。

   f. `EntityContext` のインスタンスを受け取る `setEntityContext` メソッドおよび `unsetEntityContext` メソッドを実装します（13-21 ページの「setEntityContext および unsetEntityContext メソッドの実装」を参照）。

   g. 必須の `findByPrimaryKey` finder メソッドを実装し、オプションで他の finder を実装します（15-6 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の問合せの構成」を参照）。

5. エンティティ Bean の適切なデータベース・スキーマ（表と列）を作成します。

   Bean 管理の永続性を備えたエンティティ Bean の場合は、アプリケーションが Bean 管理の永続性を備えたエンティティ Bean のインスタンスの作成を試行する前に、ユーザーが（`data-sources.xml` ファイルで定義されている）データベースにこのスキーマを作成します。

6. Bean 実装と一致するように、また `data-sources.xml` ファイルで定義されているデータソースを参照するように `ejb-jar.xml` ファイルを構成します（13-15 ページの「デプロイ XML の使用方法」を参照）。

7. エンティティ Bean の構成を完了します（第 15 章「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の使用方法」を参照）。

Java の使用方法

例 13-5 に、Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の一般的な実装を示します。例 13-7 では対応するホーム・インタフェースを示し、例 13-6 では対応するリモート・インタフェースを示します。

例 13-5 Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の実装

```
package bmpapp;

import java.util.*;
import java.rmi.*;
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
import javax.ejb.*;

public class EmployeeBean implements EntityBean {

    public Integer empNo;

    public EntityContext ctx;
    private Connection conn = null;
    private PreparedStatement ps = null;
    private EmployeePK pk;
    private static final String dsName = "jdbc/OracleDS";

    private static final String insertStatement =
        "INSERT INTO EMP (EMPNO, ENAME, SAL) VALUES (?, ?, ?)";
    private static final String updateStatement =
        "UPDATE EMP SET ENAME=?, SAL=? WHERE EMPNO=?";
    private static final String deleteStatement =
        "DELETE FROM EMP WHERE EMPNO=?";
    private static final String findAllStatement =
        "SELECT EMPNO, ENAME, SAL FROM EMP";
    private static final String findByPKStatement =
        "SELECT EMPNO, ENAME, SAL FROM EMP WHERE EMPNO = ?";
    private static final String findByNameStatement =
        "SELECT EMPNO, ENAME, SAL FROM EMP WHERE ENAME = ?";
    // or you can define a variable specific to orion to implement finder-method:
    // or use <finder-method/> in orion-ejb-jar.xml
    public static final String findByNameQuery="full: " +
        "SELECT EMPNO, ENAME, SAL FROM EMP WHERE ENAME = $1";

    public EmployeeBean() {
        // Empty constructor, don't initialize here but in the create().
        // passivate() may destroy these attributes in the case of pooling
    }

    public EmployeePK ejbCreate(Integer empNo, String empName, Float salary)
        throws CreateException {
        try {
            pk = new EmployeePK(empNo, empName, salary);
            conn = getConnection(dsName);
            ps = conn.prepareStatement(insertStatement);
            ps.setInt(1, empNo.intValue());
            ps.setString(2, empName);
            ps.setFloat(3, salary.floatValue());
            ps.executeUpdate();
            return pk;
        }
        catch (SQLException e) {
            System.out.println("Caught an exception 1 " + e.getMessage() );
            throw new CreateException(e.getMessage());
        }
        catch (NamingException e) {
```

```
            System.out.println("Caught an exception 1 " + e.getMessage() );
            throw new EJBException(e.getMessage());
        }
        finally {
            try {
                ps.close();
                conn.close();
            } catch (SQLException e) {
                throw new EJBException(e.getMessage());
            }
        }
    }

    public void ejbPostCreate(Integer empNo, String empName, Float salary)
        throws CreateException {
    }

    public EmployeePK ejbFindByPrimaryKey(EmployeePK pk)
        throws FinderException {
        if (pk == null || pk.empNo == null) {
            throw new FinderException("Primary key cannot be null");
        }
        try {
            conn = getConnection(dsName);
            ps = conn.prepareStatement(findByPKStatement);
            ps.setInt(1, pk.empNo.intValue());
            ps.executeQuery();
            ResultSet rs = ps.getResultSet();
            if (rs.next()) {
                pk.empNo = new Integer(rs.getInt(1));
                pk.empName = new String(rs.getString(2));
                pk.salary = new Float(rs.getFloat(3));
            }
            else {
                throw new FinderException("Failed to select this PK");
            }
        }
        catch (SQLException e) {
            throw new FinderException(e.getMessage());
        }
        catch (NamingException e) {
            System.out.println("Caught an exception 1 " + e.getMessage() );
            throw new EJBException(e.getMessage());
        }
        finally {
            try {
                ps.close();
                conn.close();
            }
        catch (SQLException e) {
                throw new EJBException(e.getMessage());
            }
        }
        return pk;
    }

    public Collection ejbFindAll() throws FinderException {
        //System.out.println("EmployeeBean.ejbFindAll(): begin");
        Vector recs = new Vector();
        try {
            conn = getConnection(dsName);
            ps = conn.prepareStatement(findAllStatement);
            ps.executeQuery();
            ResultSet rs = ps.getResultSet();
            int i = 0;
            while (rs.next()) {
```

```
                pk = new EmployeePK();
                pk.empNo = new Integer(rs.getInt(1));
                pk.empName = new String(rs.getString(2));
                pk.salary = new Float(rs.getFloat(3));
                recs.add(pk);
            }
        }
        catch (SQLException e) {
            throw new FinderException(e.getMessage());
        }
        catch (NamingException e) {
            System.out.println("Caught an exception 1 " + e.getMessage() );
            throw new EJBException(e.getMessage());
        }
        finally {
            try {
                ps.close();
                conn.close();
            }
            catch (SQLException e) {
                throw new EJBException(e.getMessage());
            }
        }
        return recs;
    }

    public Collection ejbFindByName(String empName)
        throws FinderException {
        //System.out.println("EmployeeBean.ejbFindByName(): begin");
        if (empName == null) {
            throw new FinderException("Name cannot be null");
        }
        Vector recs = new Vector();
        try {
            conn = getConnection(dsName);
            ps = conn.prepareStatement(findByNameStatement);
            ps.setString(1, empName);
            ps.executeQuery();
            ResultSet rs = ps.getResultSet();
            int i = 0;
            while (rs.next()) {
                pk = new EmployeePK();
                pk.empNo = new Integer(rs.getInt(1));
                pk.empName = new String(rs.getString(2));
                pk.salary = new Float(rs.getFloat(3));
                recs.add(pk);
            }
        }
        catch (SQLException e) {
            throw new FinderException(e.getMessage());
        }
        catch (NamingException e) {
            System.out.println("Caught an exception 1 " + e.getMessage() );
            throw new EJBException(e.getMessage());
        }
        finally {
            try {
                ps.close();
                conn.close();
            }
            catch (SQLException e) {
                throw new EJBException(e.getMessage());
            }
        }
        return recs;
    }
```

```
public void ejbLoad() throws EJBException {
    //Container invokes this method to instruct the instance to
    //synchronize its state by loading it from the underlying database
    //System.out.println("EmployeeBean.ejbLoad(): begin");
    try {
        pk = (EmployeePK) ctx.getPrimaryKey();
        ejbFindByPrimaryKey(pk);
    }
    catch (FinderException e) {
        throw new EJBException (e.getMessage());
    }
}

public void ejbStore() throws EJBException {
    //Container invokes this method to instruct the instance to
    //synchronize its state by storing it to the underlying database
    //System.out.println("EmployeeBean.ejbStore(): begin");
    try {
        pk = (EmployeePK) ctx.getPrimaryKey();
        conn = getConnection(dsName);
        ps = conn.prepareStatement(updateStatement);
        ps.setString(1, pk.empName);
        ps.setFloat(2, pk.salary.floatValue());
        ps.setInt(3, pk.empNo.intValue());
        if (ps.executeUpdate() != 1) {
            throw new EJBException("Failed to update record");
        }
    }
    catch (SQLException e) {
        throw new EJBException(e.getMessage());
    }
    catch (NamingException e) {
        System.out.println("Caught an exception 1 " + e.getMessage() );
        throw new EJBException(e.getMessage());
    }
    finally {
        try {
            ps.close();
            conn.close();
        }
        catch (SQLException e) {
            throw new EJBException(e.getMessage());
        }
    }
}

public void ejbRemove() throws RemoveException {
    //Container invokes this method befor it removes the EJB object
    //that is currently associated with the instance
    //System.out.println("EmployeeBean.ejbRemove(): begin");
    try {
        pk = (EmployeePK) ctx.getPrimaryKey();
        conn = getConnection(dsName);
        ps = conn.prepareStatement(deleteStatement);
        ps.setInt(1, pk.empNo.intValue());
        if (ps.executeUpdate() != 1) {
            throw new RemoveException("Failed to delete record");
        }
    }
    catch (SQLException e) {
        throw new RemoveException(e.getMessage());
    }
    catch (NamingException e) {
        System.out.println("Caught an exception 1 " + e.getMessage() );
        throw new EJBException(e.getMessage());
```

```
        }
        finally {
            try {
                ps.close();
                conn.close();
            }
            catch (SQLException e) {
                throw new EJBException(e.getMessage());
            }
        }
    }

    public void ejbActivate() {
        // Container invokes this method when the instance is taken out
        // of the pool of available instances to become associated with
        // a specific EJB object
        //System.out.println("EmployeeBean.ejbActivate(): begin");
    }

    public void ejbPassivate() {
        // Container invokes this method on an instance before the instance
        // becomes disassociated with a specific EJB object
        //System.out.println("EmployeeBean.ejbPassivate(): begin");
    }

    public void setEntityContext(EntityContext ctx) {
        //Set the associated entity context
        //System.out.println("EmployeeBean.setEntityContext(): begin");
        this.ctx = ctx;
    }

    public void unsetEntityContext() {
        //Unset the associated entity context
        //System.out.println("EmployeeBean.unsetEntityContext(): begin");
        this.ctx = null;
    }

    /**
     * methods inherited from EJBObject
     */
    public Integer getEmpNo() {
        pk = (EmployeePK) ctx.getPrimaryKey();
        return pk.empNo;
    }

    public String getEmpName() {
        pk = (EmployeePK) ctx.getPrimaryKey();
        return pk.empName;
    }

    public Float getSalary() {
        pk = (EmployeePK) ctx.getPrimaryKey();
        return pk.salary;
    }

    public void setEmpNo(Integer empNo) {
        pk = (EmployeePK) ctx.getPrimaryKey();
        pk.empNo = empNo;
    }

    public void setEmpName(String empName) {
        pk = (EmployeePK) ctx.getPrimaryKey();
        pk.empName = empName;
    }

    public void setSalary(Float salary) {
```

```
        pk = (EmployeePK) ctx.getPrimaryKey();
        pk.salary = salary;
    }

    public EJBHome getEJBHome() {
        return ctx.getEJBHome();
    }

    public Handle getHandle() throws RemoteException {
        return ctx.getEJBObject().getHandle();
    }

    public Object getPrimaryKey() throws RemoteException {
        return ctx.getEJBObject().getPrimaryKey();
    }

    public boolean isIdentical(EJBObject remote) throws RemoteException {
        return ctx.getEJBObject().isIdentical(remote);
    }

    public void remove() throws RemoveException, RemoteException{
        ctx.getEJBObject().remove();
    }

    /**
     * Private methods
     */
    private Connection getConnection(String dsName)
             throws SQLException, NamingException {
        DataSource ds = getDataSource(dsName);
        return ds.getConnection();
    }

    private DataSource getDataSource(String dsName) throws NamingException {
        DataSource ds = null;
        Context ic = new InitialContext();
        ds = (DataSource) ic.lookup(dsName);
        return ds;
    }
}
```

例 13-6 EJB 2.1 BMP リモート・ホーム・インタフェース

```
package bmpapp;

import java.rmi.*;
import java.util.*;
import javax.ejb.*;

public interface EmployeeHome extends EJBHome {

    public Employee create(Integer empNo, String empName, Float salary)
        throws CreateException, RemoteException;

    public Employee findByPrimaryKey(EmployeePK pk)
        throws FinderException, RemoteException;

    public Collection findByName(String empName)
        throws FinderException, RemoteException;

    public Collection findAll()
        throws FinderException, RemoteException;
}
```

例 13-7 EJB 2.1 BMP リモート・コンポーネント・インタフェース

```
package bmpapp;

import java.rmi.*;
import javax.ejb.*;

public interface Employee extends EJBObject {

    // getter remote methods
    public Integer getEmpNo() throws RemoteException;
    public String getEmpName() throws RemoteException;
    public Float getSalary() throws RemoteException;

    // setter remote methods
    public void setEmpNo(Integer empNo) throws RemoteException;
    public void setEmpName(String empName) throws RemoteException;
    public void setSalary(Float salary) throws RemoteException;
}
```

デプロイ XML の使用方法

例 13-8 に、例 13-5 に示した Bean 管理の永続性を備えたエンティティ Bean に対応する ejb-jar.xml エンティティ要素を示します。

例 13-8 Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の ejb-jar.xml

```
...
    <enterprise-beans>
        <entity>
            <description>no description</description>
            <display-name>EmployeeBean</display-name>
            <ejb-name>EmployeeBean</ejb-name>
            <home>bmpapp.EmployeeHome</home>
            <remote>bmpapp.Employee</remote>
            <ejb-class>bmpapp.EmployeeBean</ejb-class>
            <persistence-type>Bean</persistence-type>
            <prim-key-class>bmpapp.EmployeePK</prim-key-class>
            <reentrant>False</reentrant>
            <resource-ref>
                <res-ref-name>jdbc/OracleDS</res-ref-name>
                <res-type>javax.sql.DataSource</res-type>
                <res-auth>Application</res-auth>
            </resource-ref>
        </entity>
    </enterprise-beans>
...
```

例 13-9 に、例 13-8 で示した ejb-jar.xml ファイルで使用した res-ref-name（jdbc/OracleDS）を指定する data-sources.xml ファイルの data-source 要素の ejb-location 属性を示します。

例 13-9 Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean データ ソースの data-sources.xml

```
    <connection-pool name="Example Connection Pool">
        <!-- This is an example of a connection factory that emulates XA behavior. -->
        <connection-factory factory-class="oracle.jdbc.pool.OracleDataSource"
            user="scott"
            password="tiger"
            url="jdbc:oracle:thin:@//localhost:1521/oracle.regress.rdbms.dev.us.oracle.com">
        </connection-factory>
    </connection-pool>

    <managed-data-source name="OracleDS"
        connection-pool-name="Example Connection Pool"
        jndi-name="jdbc/OracleDS"/>
```

Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の ejbCreate メソッドの実装

ejbCreate メソッドは、主に主キーの作成を実行します。次のものが含まれます。

1. 主キーの作成
2. キーに対する永続データ表現の作成
3. 一意の値へのキーの初期化および重複がないことの確認
4. コンテナへのこのキーの返却

コンテナにより、キーがエンティティ Bean の参照にマッピングされます。

次の例では、employee の例の ejbCreate メソッドを示します。このメソッドは、主キーである empNo を初期化します。これは、本来なら、自動的に、次に使用可能な従業員番号である主キーを生成します。ただし、ここでは例を簡単にするために、この ejbCreate メソッドはユーザーに一意の従業員番号を入力するよう求めます。

注意： 単純化のために、この例ではサンプル内の try ブロックが省略されています。

さらに、従業員の全データがこのメソッドで提供されるため、データはこのインスタンスのコンテキスト変数内に格納されます。初期化後、このキーがコンテナに返されます。

```
// The create methods takes care of generating a new empNo and returns
// its primary key to the container
public Integer ejbCreate (Integer empNo, String empName, Float salary)
   throws CreateException {
  // in this implementation, the client gives the employee number,
  // so only need to assign it, not create it
  this.empNo = empNo;
  this.empName = empName;
  this.salary = salary;

  // insert employee into database
  conn = getConnection(dsName);
  ps = conn.prepareStatement("INSERT INTO EMPLOYEEBEAN (EmpNo, EmpName, SAL)
       VALUES ( "+this.empNo.intValue()+", "+this.empName+","
       + this.salary.floatValue()+")");
  ps.executeUpdate();
  ps.close();

  // return the new primary key
  return (empNo);
}
```

このデプロイメント・ディスクリプタでは、<prim-key-class> 要素内で、主キー・クラスのみ定義します。Bean がデータを保存するため、デプロイメント・ディスクリプタには、永続データの定義は存在しません。ただし、デプロイメント・ディスクリプタの <resource-ref> 要素で、Bean が使用するデータベースを定義します。データベース構成の詳細は、13-15 ページの「デプロイ XML の使用方法」を参照してください。

```
<enterprise-beans>
   <entity>
      <display-name>EmployeeBean</display-name>
      <ejb-name>EmployeeBean</ejb-name>
      <local-home>employee.EmployeeLocalHome</local-home>
      <local>employee.EmployeeLocal</local>
      <ejb-class>employee.EmployeeBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>False</reentrant>
      <resource-ref>
            <res-ref-name>jdbc/OracleDS</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Application</res-auth>
      </resource-ref>
   </entity>
</enterprise-beans>
```

または、複数のデータ型に基づいた複合主キーを作成可能です。複合主キーは、次のように、そのクラス内で定義します。

```
package employee;

import java.io.*;
java.io.Serializable;

...

public class EmployeePK implements java.io.Serializable {
  public Integer empNo;
  public String empName;
  public Float salary;

  public EmployeePK(Integer empNo) {
    this.empNo = empNo;
    this.empName = null;
    this.salary = null;
  }

  public EmployeePK(Integer empNo, String empName, Float salary) {
    this.empNo = empNo;
    this.empName = empName;
    this.salary = salary;
  }

}
```

主キー・クラスの場合、クラスを <prim-key-class> 要素で定義します。これは、単純な主キー定義と同じです。

```
<enterprise-beans>
   <entity>
      <display-name>EmployeeBean</display-name>
      <ejb-name>EmployeeBean</ejb-name>
      <local-home>employee.EmployeeLocalHome</local-home>
      <local>employee.EmployeeLocal</local>
      <ejb-class>employee.EmployeeBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>employee.EmployeePK</prim-key-class>
      <reentrant>False</reentrant>
      <resource-ref>
            <res-ref-name>jdbc/OracleDS</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Application</res-auth>
      </resource-ref>
   </entity>
</enterprise-beans>
```

employee の例では、ユーザーが従業員番号を Bean に対して指定する必要があります。別の手段としては、次に使用可能な従業員番号を計算し、この番号を従業員の名前および勤務地と組み合せて従業員番号を生成する方法があります。

複合主キー・クラスの定義後、次のように、ejbCreate メソッド内で主キーを作成します。

```
public EmployeePK ejbCreate(Integer empNo, String empName, Float salary)
    throws CreateException {
  pk = new EmployeePK(empNo, empName, salary);
  ...
}
```

ejbCreate（または ejbPostCreate）が処理する作業には、他に、Bean の存続期間中に必要なリソースの割当てがあります。この例では、すでに従業員情報が存在するため、ejbCreate は次の処理を実行します。

1. データベースへの接続を取得します。この接続は、Bean の存続期間中オープンされています。データベース内の従業員情報の更新に使用されます。ejbPassivate および ejbRemove で解放し、ejbActivate で再割当てを行います。
2. データベースの従業員情報を更新します。

これは、次のように実行されます。

```
public EmployeePK ejbCreate(Integer empNo, String empName, Float salary)
    throws CreateException {
  pk = new EmployeePK(empNo, empName, salary);
  conn = getConnection(dsName);
  ps = conn.prepareStatement("INSERT INTO EMPLOYEEBEAN (EmpNo, EmpName, SAL)
      VALUES ( "+this.empNo.intValue()+", "+this.empName+","
      + this.salary.floatValue()+")");
  ps.executeUpdate();
  ps.close();
  return pk;
}
```

EJB 2.1 ホーム・インタフェースの実装

ホーム・インタフェースは、クライアントがエンティティ Bean インスタンスの作成または取得に使用するメソッドを指定するために使用します。

ホーム・インタフェースには、クライアントが Bean のインスタンスを作成するために起動する `create` メソッドが含まれている必要があります。エンティティ Bean には、それぞれ定義されたパラメータを持つ 0（ゼロ）以上の `create` メソッドを使用できます。各 `create` メソッドにつき、対応する `ejbCreate` メソッドを Bean 実装で定義します。

すべてのエンティティ Bean で、1 つ以上の finder メソッドをホーム・インタフェースに定義する必要があります。そのうちの 1 つ以上は `findByPrimaryKey` メソッドである必要があります。オプションで、事前定義およびデフォルトの finder を含む他の finder メソッドを定義できます。これらには、`find<name>` のように名前を付けます。詳細は、1-56 ページの「finder メソッドについて」を参照してください。

メソッドの作成および取得に加えて、ホーム・インタフェース内でホーム・インタフェースのビジネス・メソッドを提供できます。このメソッドの機能では、特定のエンティティ・オブジェクトのデータにアクセスできません。このメソッドは、単一のエンティティ Bean インスタンスに関連がない情報を取得するために使用します。クライアントがホーム・インタフェースの任意のビジネス・メソッドを起動すると、エンティティ Bean はプールから移動され、リクエストを処理します。したがって、このメソッドを使用すると、Bean に関連する一般的な情報に関する操作を実行できます。

たとえば、employee アプリケーションでは、ローカル・ホーム・インタフェースに `create`、`findByPrimaryKey`、`findAll` および `calcSalary` メソッドを提供できます。`calcSalary` メソッドは、全従業員の給与合計を計算する、ホーム・インタフェースのビジネス・メソッドです。このメソッドは特定の従業員の情報にはアクセスしませんが、全従業員のデータベースに対して SQL 問合せを実行します。

ホーム・インタフェースには次の 2 種類があります。

- リモート・ホーム・インタフェースは、`javax.ejb.EJBHome` を拡張します（13-19 ページの「リモート・ホーム・インタフェースの実装」を参照）。
- ローカル・ホーム・インタフェースは、`javax.ejb.EJBLocalHome` を拡張します（13-20 ページの「ローカル・ホーム・インタフェースの実装」を参照）。

リモート・ホーム・インタフェースの実装

リモート・クライアントは、リモート・インタフェースを介して EJB を起動します。クライアントは、リモート・ホーム・インタフェースで宣言された `create` メソッドを起動します。コンテナは、Bean 実装内の、適切なパラメータ・シグネチャを持つ `ejbCreate` メソッドにクライアント・コールを渡します。リモート・ホーム・インタフェースを開発するための要件は次のとおりです。

- リモート・ホーム・インタフェースでは、`javax.ejb.EJBHome` インタフェースを拡張する必要があります。
- すべての `create` メソッドで、次の例外をスローできます。
  - `javax.ejb.CreateException`
  - `javax.ejb.EJBException` または別の `RuntimeException`

例 13-2 に、例 13-1 のコンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean に対応するリモート・ホーム・インタフェースを示します。例 13-6 に、例 13-5 の Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean に対応するリモート・ホーム・インタフェースを示します。

ローカル・ホーム・インタフェースの実装

EJB は、同じコンテナに存在するクライアントからローカルでコールできます。したがって、同一 JVM 上の Bean、JSP またはサーブレットは、ローカル・ホーム・インタフェースで宣言された `create` メソッドを起動します。コンテナは、Bean 実装内の、適切なパラメータ・シグネチャを持つ `ejbCreate` メソッドにクライアント・コールを渡します。ローカル・ホーム・インタフェースを開発するための要件は、次のとおりです。

- ローカル・ホーム・インタフェースでは、`javax.ejb.EJBLocalHome` インタフェースを拡張する必要があります。
- すべての `create` メソッドで、次の例外をスローできます。
  - `javax.ejb.CreateException`
  - `javax.ejb.EJBException` または別の `RuntimeException`

EJB 2.1 コンポーネント・インタフェースの実装

コンポーネント・インタフェースでは、クライアントから起動可能な Bean のビジネス・メソッドを定義します。

エンティティ Bean コンポーネント・インタフェースは、クライアントがそのメソッドを起動できるインタフェースです。コンポーネント・インタフェースは、エンティティ Bean インスタンスのビジネス・ロジック・メソッドを定義します。

コンポーネント・インタフェースには次の 2 種類があります。

- リモート・コンポーネント・インタフェースは、`javax.ejb.EJBObject` を拡張します（13-20 ページの「リモート・コンポーネント・インタフェースの実装」を参照）。
- ローカル・コンポーネント・インタフェースは、`javax.ejb.EJBLocalObject` を拡張します（13-21 ページの「ローカル・コンポーネント・インタフェースの実装」を参照）。

リモート・コンポーネント・インタフェースの実装

リモート・インタフェースでは、リモート・クライアントによって起動可能なビジネス・メソッドを定義します。リモート・コンポーネント・インタフェースを開発するための要件は、次のとおりです。

- Bean のリモート・コンポーネント・インタフェースは、`javax.ejb.EJBObject` インタフェースを拡張する必要があり、そのメソッドは `java.rmi.RemoteException` 例外をスローする必要があります。
- リモート・インタフェースとそのメソッドは、リモート・クライアントに対する `public` として宣言する必要があります。
- リモート・コンポーネント・インタフェース、すべてのメソッド・パラメータおよび戻り型はシリアライズ可能である必要があります。一般的に、RMI は両側のオブジェクトをマーシャリングおよびアンマーシャリングするため、クライアントと Enterprise Bean の間で受渡しされるオブジェクトは、すべてシリアライズ可能である必要があります。
- クライアントには任意の例外をスローできます。`EJBException` および `RemoteException` を含めた実行時例外は、リモート実行時例外としてクライアントに転送されます。
- リモート・コンポーネント・インタフェースは、指定されたアプリケーション例外をスローできます。

例 13-3 に、例 13-1 のコンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean に対応するリモート・コンポーネント・インタフェースを示します。例 13-7 に、例 13-5 の Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean に対応するリモート・コンポーネント・インタフェースを示します。

ローカル・コンポーネント・インタフェースの実装

ローカル・コンポーネント・インタフェースでは、ローカル（同一 JVM 上の）クライアントから起動可能な Bean のビジネス・メソッドを定義します。ローカル・コンポーネント・インタフェースを開発するための要件は、次のとおりです。

- Bean のローカル・コンポーネント・インタフェースでは、`javax.ejb.EJBLocalObject` インタフェースを拡張する必要があります。
- ローカル・コンポーネント・インタフェースとそのメソッドは、`public` として宣言します。

setEntityContext および unsetEntityContext メソッドの実装

エンティティ Bean のインスタンスは、このメソッドを使用して、コンテキストへの参照を維持します。エンティティ Bean には、コンテナによって維持され、Bean から使用可能なコンテキストが存在します。エンティティ・コンテキスト内のメソッドを使用して、セキュリティおよびトランザクションのロールなどの Bean に関する情報の取得が、Bean によって行われる場合があります。Bean に関してコンテキストから取得可能なすべての情報は、Sun 社の EJB 仕様を参照してください。

コンテナは、Bean をインスタンス化すると、`setEntityContext` メソッドを起動して、Bean からコンテキストを取得できるようにします。コンテナは、トランザクション・コンテキストからはこのメソッドをコールしません。この時点で Bean がコンテキストを保存しなかった場合、Bean は二度とコンテキストにアクセスできなくなります。

注意： インスタンスの存続期間中存在するリソースの割当ておよび破棄には、`setEntityContext` および `unsetEntityContext` メソッドも使用可能です。

コンテナはこのメソッドをコールする際、`EntityContext` オブジェクトの参照を Bean に渡します。Bean は、この参照を後の使用のために格納できます。次の例では、Bean がコンテキストを **`this.ctx`** 変数に格納するところを示します。

このメソッドを使用して、Bean のコンテキストの参照を取得します。エンティティ Bean には、コンテナによって維持され、Bean から使用可能なエンティティ・コンテキストが存在します。Bean は、エンティティ・コンテキスト内のメソッドを使用して、コンテナへのコールバック・リクエストを送信できます。

例 13-10 に、セッション・コンテキストを `entityctx` 変数に格納するエンティティ Bean を示します。

例 13-10 setEntityContext および unsetEntityContext メソッドの実装

```
import javax.ejb.*;

public class MyBean implements EnityBean {
   EntityContext entityctx;

   public void setEntityContext(EntityContext ctx) {
      entityctx = ctx;   // entity context is stored in instance variable
   }

    public void unsetEntityContext() {
        entityctx = null;
    }

   // other methods in the bean
}
```

14

コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の使用方法

この章では、コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean を使用するために構成する必要のある様々なオプションについて説明します。

表 14-1 に、これらのオプションをリストし、基本オプション（ほとんどのアプリケーションに適用可能）であるか拡張オプション（より特殊なアプリケーションに適用可能）であるかを示します。

詳細は、次を参照してください。

- 1-45 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean とは」
- 13-2 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の実装」

表 14-1 EJB 2.1 CMP エンティティ Bean の構成オプション

オプション	タイプ
14-2 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の主キーの構成」	基本
14-5 ページの「表および列情報の構成」	拡張
14-5 ページの「自動的なデータベース表作成の構成」	拡張
14-6 ページの「デフォルトの関連性生成の構成」	拡張
14-8 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean におけるコンテナ管理の永続性フィールドの構成」	基本
14-9 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean におけるコンテナ管理の関連性フィールドの構成」	基本
14-11 ページの「1 対 1 関連の構成」	基本
14-14 ページの「多対 1 関連の構成」	基本
14-12 ページの「1 対多関連の構成」	基本
14-15 ページの「多対多関連の構成」	基本
14-16 ページの「finder メソッドにおける遅延ロードの構成」	拡張
31-5 ページの「Bean インスタンスのプール・サイズの構成」	基本
31-8 ページの「エンティティ Bean の Bean インスタンス・プール・タイムアウトの構成」	拡張
14-17 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル・コールバック・メソッドの構成」	基本

コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の主キーの構成

コンテナ管理の永続性を備えたすべての EJB 2.1 エンティティ Bean には主キー・フィールドが必要です。

主キーは、一般的な Java 型（14-2 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の主キー・フィールドの構成」を参照）またはユーザーが作成する特殊な型（14-3 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のコンポジット主キー・クラスの構成」を参照）として構成できます。

詳細は、1-48 ページの「コンテナ管理の永続性を備えたエンティティ Bean の主キー」を参照してください。

通常は、OC4J によって自動的に主キー値が割り当てられます。OC4J による主キー値の割当て方法を構成するには、TopLink 永続性 API を使用します。詳細は、次を参照してください。

- 3-15 ページの「TopLink EJB 2.1 永続性マネージャのカスタマイズ」
- 『Oracle TopLink 開発者ガイド』の「リレーショナル・プロジェクトにおける順序付けの概要」

コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の主キー・フィールドの構成

コンテナ管理の永続性を備えた単純な EJB 2.1 エンティティ Bean の場合は、次のように主キーを一般的な Java 型として定義できます。

- 主キー・クラスの型を返すように Bean の `ejbCreate` メソッドをコーディングします（13-2 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の実装」を参照）。
- それを使用するようにデプロイ XML を構成します（14-3 ページの「デプロイ XML の使用方法」を参照）。

定義した後、コンテナはエンティティ Bean 表に主キー用の 1 つまたは複数の列を作成でき、デプロイメント・ディスクリプタで定義した主キーをこの列にマッピングできます。コンテナはこの型の主キーのインスタンス化を管理し、エンティティ Bean の主キー・フィールドをそれに応じて初期化します。

デプロイ XML の使用方法

例 14-1 に、主キーを一般的な Java 型 `Integer` として指定するように構成されている `ejb-jar.xml` ファイルの `entity` 要素の属性 `prim-key-class` および `primkey-field` を示します。

例 14-1 コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の Integer 型主キー・フィールド用の ejb-jar.xml

```
<enterprise-beans>
  <entity>
    <display-name>Employee</display-name>
    <ejb-name>EmployeeBean</ejb-name>
    <local-home>employee.EmployeeLocalHome</local-home>
    <local>employee.EmployeeLocal</local>
    <ejb-class>employee.EmployeeBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Integer</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>Employee</abstract-schema-name>
    <cmp-field><field-name>empNo</field-name></cmp-field>
    <cmp-field><field-name>empName</field-name></cmp-field>
    <cmp-field><field-name>salary</field-name></cmp-field>
    <primkey-field>empNo</primkey-field>
    </entity>
...
</enterprise-beans>
```

コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のコンポジット主キー・クラスの構成

主キーが一般的な Java データ型よりも複雑な場合は、独自の主キー・クラスを定義できます。

主キー・クラスには、次の特性が必要です。

- *<name>*`PK` という名前であること
- `public` および `serializable` であること
- 主キー・インスタンスを作成するためのコンストラクタを提供すること

クラスには、主キーの構成に使用するいくつかのインスタンス変数が含まれていることがあります。インスタンス変数には、次の特性が必要です。

- `public` であること
- プリミティブまたはシリアライズ可能なデータ型、または SQL 型にマッピングできる型を使用すること

主キー・クラスを定義したら（14-4 ページの「Java の使用方法」を参照）、それをエンティティ Bean 内で使用するために、次の処理を行う必要があります。

- 主キー・クラスの型を返すように Bean の `ejbCreate` メソッドをコーディングします（13-2 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の実装」を参照）。
- それを使用するようにデプロイ XML を構成します（14-4 ページの「デプロイ XML の使用方法」を参照）。

定義した後、コンテナはエンティティ Bean 表に主キー用の 1 つまたは複数の列を作成でき、デプロイメント・ディスクリプタで定義した主キーをこの列にマッピングできます。コンテナはこの型の主キーのインスタンス化を管理し、エンティティ Bean の主キー・フィールドをそれに応じて初期化します。

Java の使用方法

例 14-2 に、主キー・クラスの例を示します。

例 14-2 コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の主キー・クラスの実装

```
package employee;

import java.io.*;
import java.io.Serializable;
...

public class EmployeePK implements java.io.Serializable {
  public Integer empNo;

  public EmployeePK() {
    this.empNo = null;
  }

  public EmployeePK(Integer empNo) {
    this.empNo = empNo;
  }
}
```

デプロイ XML の使用方法

例 14-3 に示すように、ejb-jar.xml ファイルの <prim-key-class> 要素内で主キー・クラスを定義します。各主キー・クラス・インスタンス変数は、主キー・クラスで使用されるのと同じ変数名を使用して <cmp-field><field-name> 要素で定義します。

例 14-3 コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の主キー・クラスおよびそのインスタンス変数の ejb-jar.xml

```
<enterprise-beans>
  <entity>
    <description>no description</description>
    <display-name>EmployeeBean</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <local-home>employee.LocalEmployeeHome</home>
         <local>employee.LocalEmployee</remote>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
        <prim-key-class>employee.EmployeePK</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-version>2.x</cmp-version>
         <abstract-schema-name>Employee</abstract-schema-name>
        <cmp-field><field-name>empNo</field-name></cmp-field>
         <cmp-field><field-name>empName</field-name></cmp-field>
         <cmp-field><field-name>salary</field-name></cmp-field>
      </entity>
</enterprise-beans>
```

定義した後、コンテナはエンティティ Bean 表に主キー用の 1 つまたは複数の列を作成でき、デプロイメント・ディスクリプタで定義した主キー・クラスをこの列にマッピングできます。

表および列情報の構成

EJB 2.1 仕様には、エンティティ Bean の抽象永続性スキーマを永続ストアのリレーショナル・スキーマ（またはその他のスキーマ）にマッピングする方法や、そのようなマッピングの記述方法は規定されていません。

ただし、OC4J および TopLink 永続性 API を使用して、次のことを実行できます。

- コンテナ管理の永続性を備えたエンティティ Bean に関連付けられたデータベース表の表名と列名を指定できます。
- コンテナ管理の永続性フィールドとコンテナ管理の関連性フィールドをリレーショナル・スキーマにマッピングする方法を指定できます。
- オブジェクトを維持するデータベース表を自動的に作成（オプションで削除）できます。

詳細は、次を参照してください。

- 3-15 ページの「TopLink EJB 2.1 永続性マネージャのカスタマイズ」
- 『Oracle TopLink 開発者ガイド』の「リレーショナル・マッピングの概要」
- 『Oracle TopLink 開発者ガイド』の「関連表の構成」
- 14-5 ページの「自動的なデータベース表作成の構成」

注意： このリリースでは、`orion-ejb-jar.xml` ファイルの `<entity-deployment>` のサブ要素 `<cmp-field-mapping>` は使用しません。詳細は、A-11 ページの「<entity-deployment>」を参照してください。

自動的なデータベース表作成の構成

オブジェクトを維持するデータベース表を自動的に作成（オプションで削除）するように OC4J を構成できます（14-5 ページの「デプロイ XML の使用方法」を参照）。

デフォルト・マッピングとともにこの機能を使用できます（14-6 ページの「デフォルトの関連性生成の構成」を参照）。

デプロイ XML の使用方法

自動的なデータベース表作成は、表 14-2 に示す 3 つのレベルのいずれかで構成できます。システム・レベル構成はアプリケーション・レベルでオーバーライドでき、システムおよびアプリケーション構成は EJB モジュール・レベルでオーバーライドできます。

表 14-2 自動的な表生成の構成

レベル	構成ファイル	設定	値
システム（グローバル）	*<OC4J_HOME>*/config/application.xml	autocreate-tables autodelete-tables	True[1] または False True または False[1]
アプリケーション（EAR）	orion-application.xml	autocreate-tables autodelete-tables	True[1] または False True または False[1]
EJB モジュール（JAR）	orion-ejb-jar.xml	pm-properties のサブ要素 default-mapping の属性 db-table-gen[2]	Create、DropAndCreate または UseExisting[3]

1 デフォルト。

2 詳細は、3-15 ページの「TopLink EJB 2.1 永続性マネージャのカスタマイズ」を参照してください。

3 表 14-3 を参照してください。

EJB モジュール・レベルで自動的な表生成を構成する場合、表 14-3 に示すように、db-table-gen 属性に割り当てる値は autocreate-tables および autodelete-tables 設定に対応します。

表 14-3 db-table-gen の同等の設定

db-table-gen 設定	autocreate-tables 設定	autodelete-tables 設定
Create	True	False
DropAndCreate	True	True
UseExisting	False	NA

デフォルトの関連性生成の構成

デプロイ時に必要なすべての関連性を自動的に生成するように OC4J を構成できます（14-7 ページの「デプロイ XML の使用方法」を参照）。この機能を使用するには、次の処理を行う必要があります。

- コンテナ管理の関連性構成をすべて省略します（14-9 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean におけるコンテナ管理の関連性フィールドの構成」を参照）。
- toplink-ejb-jar.xml が EJB モジュールに存在しないことを確認します（2-8 ページの「toplink-ejb-jar.xml ファイルとは」を参照）。

自動的なデータベース表作成とともにこの機能を使用できます（14-5 ページの「自動的なデータベース表作成の構成」を参照）。

デプロイ XML の使用方法

デフォルトの関連性生成を構成するには、表 14-4 に示すように、orion-ejb-jar.xml ファイルの要素 pm-properties のサブ要素 default-mapping を構成します。

表 14-4 デフォルト・マッピングの orion-ejb-jar.xml ファイルの pm-properties サブエントリ

エントリ	説明
db-table-gen	マッピングされるデータベース表を準備するために TopLink が行うことを決定するオプションの要素。有効な値は次のとおりです。 ■ Create（デフォルト）: この値は、デプロイ時にマッピングされる表を作成するよう TopLink に指示します。表がすでに存在する場合、TopLink は適切な警告メッセージ（「表はすでに存在しています ...」など）をログに記録し、デプロイを続行します。 ■ DropAndCreate: この値は、デプロイ時に表を作成する前に表を削除するよう TopLink に指示します。最初に表が存在しない場合に削除操作を実行すると、ドライバを通じて SQLException がスローされます。ただし、TopLink は例外を処理（ログに記録して無視）し、表作成操作の処理に進みます。デプロイは、削除操作と作成操作の両方が失敗した場合にのみ失敗します。 ■ UseExisting: この値は、表操作を実行しないよう TopLink に指示します。表が存在しない場合も、デプロイはエラーなしで実行されます。 orion-ejb-jar.xml ファイルが EAR ファイルで定義されていない場合は、OC4J コンテナがデプロイ時にこれを生成します。この場合、db-table-gen の値を指定するには、TopLink のシステム・プロパティ toplink.defaultmapping.dbTableGenSetting を使用します。たとえば、-Dtoplink.defaultmapping.dbTableGenSetting="DropAndCreate" などです。 orion-ejb-jar.xml プロパティは、システム・プロパティをオーバーライドします。orion-ejb-jar.xml プロパティとシステム・プロパティの両方が存在する場合、TopLink は orion-ejb-jar.xml ファイルから設定を取得します。 この設定により、autocreate-tables および autodelete-tables 構成がアプリケーション（EAR）レベルまたはシステム・レベルでオーバーライドされます。詳細は、14-5 ページの「自動的なデータベース表作成の構成」を参照してください。
extended-table-names	生成された表名が短すぎて一意にならない場合に使用される要素。値は true または false（デフォルト）に制限されます。true に設定されている場合、TopLink ランタイムは生成された表名が一意であることを確認します。 デフォルト・マッピングでは、各エンティティが 1 つの表にマッピングされます。唯一の例外は、ソース・エンティティとターゲット・エンティティに追加の関連表が 1 つ含まれる多対多マッピングです。 extended-table-names が false（デフォルト）に設定されている場合、単純な表ネーミング・アルゴリズムが使用されます。つまり、表名は TL_<bean_name> として定義されます。たとえば、Bean 名が Employee の場合、関連付けられている表名は TL_EMPLOYEE になります。 ただし、アプリケーションの複数の JAR ファイルまたは複数のアプリケーションに同じエンティティが定義されている場合、表のネーミングの競合は避けられません。 この問題に対処するには、extended-table-names を true に設定します。true に設定されている場合、TopLink は別の表ネーミング・アルゴリズムを使用します。つまり、表名は *<bean_name>_<jar_name>_<app_name>* として定義されます。このアルゴリズムでは、Bean、JAR および EAR 名の組合せを使用して、アプリケーション全体で一意の表名を形成します。たとえば、Demo.ear（アプリケーション名は Demo）にある Test.jar 内に Employee という名前の Bean がある場合、対応する表名は EMPLOYEE_TEST_DEMO になります。 orion-ejb-jar.xml ファイルが EAR ファイルで定義されていない場合は、OC4J コンテナがデプロイ時にこれを生成します。この場合、extended-table-names の値を指定するには、TopLink のシステム・プロパティ toplink.defaultmapping.useExtendedTableNames を使用します。たとえば、-Dtoplink.defaultmapping.useExtendedTableNames="true" などです。 orion-ejb-jar.xml プロパティは、システム・プロパティをオーバーライドします。orion-ejb-jar.xml プロパティとシステム・プロパティの両方が存在する場合、TopLink は orion-ejb-jar.xml ファイルから設定を取得します。

コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean におけるコンテナ管理の永続性フィールドの構成

エンティティ Bean クラスにはコンテナ管理の永続性フィールドを定義しません。コンテナ管理の永続性フィールドは仮想のみです。OC4J では、コンテナ管理の永続性フィールドの実装が提供されます。

EJB 表記規則（14-8 ページの「Java の使用方法」を参照）を使用して、public abstract の getter および setter メソッドをコンテナ管理の永続性フィールドに対して定義する必要があります。OC4J では、これらのメソッドの実装が提供されます。これらの getter および setter メソッドは、エンティティ Bean のリモート・インタフェースに公開しないでください。

コンテナ管理の永続性フィールドには、Java プリミティブ型および Java シリアライズ可能型のみ割り当てることができます。エンティティ Bean のローカル・インタフェース型（またはそのコレクション）をコンテナ管理の永続性フィールドに割り当てることはできません。

コンテナ管理の永続性フィールドは、cmp-field 要素を使用して ejb-jar.xml デプロイメント・ディスクリプタで指定する必要があります（14-9 ページの「デプロイ XML の使用方法」を参照）。これらのフィールドの名前は、有効な Java 識別子である必要があり、java.lang.Character.isLowerCase で判断されるように小文字で始まる必要があります。

注意： このリリースでは、orion-ejb-jar.xml ファイルの <entity-deployment> のサブ要素 <cmp-field-mapping> は使用しません。詳細は、A-11 ページの「<entity-deployment>」を参照してください。

アクセッサ・メソッドは、デプロイメント・ディスクリプタで指定される cmp-field の名前を持つ必要があり、cmp-field の名前の最初の文字は大文字で、get または set という接頭辞が付いている必要があります。

詳細は、1-45 ページの「コンテナ管理の永続性フィールドとは」を参照してください。

Java の使用方法

例 14-4 に、ejb-jar.xml ファイル（14-9 ページの「デプロイ XML の使用方法」を参照）で指定されたコンテナ管理の永続性フィールドの抽象 getter および setter メソッドを示します。

例 14-4 EJB 2.1 コンテナ管理の永続性フィールド

```
package cmpapp;

import javax.ejb.*;
import java.rmi.*;

public abstract class EmployeeBean implements EntityBean {

    private EntityContext ctx;

    // container-managed persistent fields accessors
    public abstract Integer getEmpNo();
    public abstract void setEmpNo(Integer empNo);

    public abstract String getEmpName();
    public abstract void setEmpName(String empName);

    public abstract Float getSalary();
    public abstract void setSalary(Float salary);
...
}
```

デプロイ XML の使用方法

例 14-5 に、Bean クラスで指定された getter および setter メソッド（14-8 ページの「Java の使用方法」を参照）の cmp-field 要素を示します。

例 14-5 EJB 2.1 コンテナ管理の永続性フィールドの ejb-jar.xml

```
<enterprise-beans>
    <entity>
        <ejb-name>Topic</ejb-name>
        <local-home>faqapp.TopicLocalHome</local-home>
        <local>faqapp.TopicLocal</local>
        <ejb-class>faqapp.TopicBean</ejb-class>
        <persistence-type>Container</persistence-type>
        <prim-key-class>java.lang.Integer</prim-key-class>
        <primkey-field>topicID</primkey-field>
        <reentrant>False</reentrant>
        <cmp-version>2.x</cmp-version>
        <abstract-schema-name>TopicBean</abstract-schema-name>
        <cmp-field>
            <field-name>topicID</field-name>
        </cmp-field>
        <cmp-field>
            <field-name>topicDesc</field-name>
        </cmp-field>
        ...
    </entity>
</enterprise-beans>
```

コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean におけるコンテナ管理の関連性フィールドの構成

エンティティ Bean クラスにはコンテナ管理の関連性フィールドを定義しません。コンテナ管理の関連性フィールドは仮想のみです。OC4J では、コンテナ管理の関連性フィールドの実装が提供されます。

EJB 表記規則（14-10 ページの「Java の使用方法」を参照）を使用して、関連するエンティティ Bean のローカル・インタフェースで public abstract の getter および setter メソッドをコンテナ管理の関連性フィールドに対して定義する必要があります。OC4J では、これらのメソッドの実装が提供されます。これらの getter および setter メソッドは、エンティティ Bean のリモート・インタフェースに公開しないでください。

コンテナ管理の関連性フィールドには、Java プリミティブ型および Java シリアライズ可能型のみ割り当てることができます。コンテナ管理の関連性フィールドには、エンティティ Bean のローカル・インタフェース型（またはそのコレクション）を割り当てることができます。

コンテナ管理の関連性フィールドは、cmr-field 要素を使用して ejb-jar.xml デプロイメント・ディスクリプタで指定する必要があります（14-11 ページの「デプロイ XML の使用方法」を参照）。これらのフィールドの名前は、有効な Java 識別子である必要があり、java.lang.Character.isLowerCase で判断されるように小文字で始まる必要があります。

アクセッサ・メソッドは、デプロイメント・ディスクリプタで指定されるコンテナ管理の関連性フィールド（cmr-field）の名前を持つ必要があり、cmr-field の名前の最初の文字は大文字で、get または set という接頭辞が付いている必要があります。

1 対多または多対多関連のコンテナ管理の関連性フィールドのアクセッサ・メソッドでは、java.util.Collection または java.util.Set のいずれかのコレクション・インタフェースを利用する必要があります。関連で使用されるコレクション・インタフェースは、デプロイメント・ディスクリプタで指定されます。コンテナ管理の関連性フィールドに使用されるコレクション・クラスの実装は、コンテナにより提供されます。コンテナ管理の関連性に使用されるコレクション・クラスは、エンティティ Bean のリモート・インタフェースを通じて公開しないでください。

詳細は、次を参照してください。

- 1-45 ページの「コンテナ管理の関連性フィールドとは」
- 14-6 ページの「デフォルトの関連性生成の構成」
- 14-11 ページの「1 対 1 関連の構成」
- 14-12 ページの「1 対多関連の構成」
- 14-14 ページの「多対 1 関連の構成」
- 14-15 ページの「多対多関連の構成」

OC4J および TopLink 永続性 API を使用して、コンテナ管理の関連性フィールドをリレーショナル・スキーマにマッピングする方法を構成できます。詳細は、次を参照してください。

- 3-15 ページの「TopLink EJB 2.1 永続性マネージャのカスタマイズ」
- 『Oracle TopLink 開発者ガイド』の「リレーショナル・マッピングの概要」

Java の使用方法

例 14-6 に、`ejb-jar.xml` ファイル（14-11 ページの「デプロイ XML の使用方法」を参照）で指定されたコンテナ管理の関連性フィールドの抽象 getter および setter メソッドを示します。

例 14-6 EJB 2.1 コンテナ管理の関連性フィールド

```
package cmpapp;

import javax.ejb.*;
import java.rmi.*;

public abstract class EmployeeBean implements EntityBean {

    private EntityContext ctx;

    // container-managed persistent fields accessors
    public abstract Integer getEmpNo();
    public abstract void setEmpNo(Integer empNo);

    public abstract String getEmpName();
    public abstract void setEmpName(String empName);

    public abstract Float getSalary();
    public abstract void setSalary(Float salary);

    public abstract void setProjects(Collection projects);
    public abstract Collection getProjects();
...
}
```

デプロイ XML の使用方法

例 14-7 に、Bean クラスで指定された getter および setter メソッド（14-10 ページの「Java の使用方法」を参照）の `cmr-field` 要素を示します。

例 14-7 EJB 2.1 コンテナ管理の関連性フィールドの ejb-jar.xml

```
...
    <relationships>
        <ejb-relation>
            <ejb-relation-name>Topic-Faqs</ejb-relation-name>
            <ejb-relationship-role>
                <ejb-relationship-role-name>Topic-has-Faqs</ejb-relationship-role-name>
                <multiplicity>Many</multiplicity>
                <relationship-role-source>
                    <ejb-name>TopicBean</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>faqs</cmr-field-name>
                    <cmr-field-type>java.util.Collection</cmr-field-type>
                </cmr-field>
            </ejb-relationship-role>
        <ejb-relation>
...
    <relationships>
```

1 対 1 関連の構成

1 対 1 関連では、エンティティ Bean の 1 つのインスタンスが別のエンティティ Bean の 1 つのインスタンスに関連付けられます。

コンテナ管理の 1 対 1 関連は、`ejb-jar.xml` デプロイメント・ディスクリプタで指定します（14-12 ページの「デプロイ XML の使用方法」を参照）。

詳細は、14-9 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean におけるコンテナ管理の関連性フィールドの構成」を参照してください。

デプロイ XML の使用方法

例 14-8 に、Order と ShippingAddress 間の単方向の 1 対 1 関連を定義する <ejb-relationship-role> 要素のペアを示します。双方向の関連の場合、適切な cmr-field を ShippingAddress の <ejb-relationship-role> に追加します。

例 14-8 EJB 2.1 の単方向 1 対 1 関連の ejb-jar.xml

```
...
<relationships>
    <ejb-relation>
        <ejb-relation-name>Order-ShippingAddress</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>order-has-address</ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>OrderEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>shippingAddress</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>address-for-order</ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>AddressEJB</ejb-name>
            </relationship-role-source>
        </ejb-relationship-role>
    </ejb-relation>
...
<relationships>
```

1 対多関連の構成

1 対多関連では、エンティティ Bean の 1 つのインスタンスが別のエンティティ Bean の複数のインスタンスに関連付けられます。

コンテナ管理の 1 対多関連は、ejb-jar.xml デプロイメント・ディスクリプタで指定します（14-13 ページの「デプロイ XML の使用方法」を参照）。

詳細は、14-9 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean におけるコンテナ管理の関連性フィールドの構成」を参照してください。

デプロイ XML の使用方法

例 14-9 に、Order と LineItem 間の双方向の 1 対多関連を定義する <ejb-relationship-role> 要素のペアを示します。単方向の関連の場合、適切な <ejb-relationship-role> 要素から cmr-field を削除します。

例 14-9 EJB 2.1 の双方向 1 対多関連の ejb-jar.xml

```
...
<relationships>
    <ejb-relation>
        <ejb-relation-name>Order-LineItem</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>order-has-lineitems</ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>OrderEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>lineItems</cmr-field-name>
                <cmr-field-type>java.util.Collection</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>lineitem-belongsto-order</ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <cascade-delete/>
            <relationship-role-source>
                <ejb-name>LineItemEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>order</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
    </ejb-relation>
...
<relationships>
```

多対 1 関連の構成

多対 1 関連では、エンティティ Bean の複数のインスタンスが別のエンティティ Bean の 1 つのインスタンスに関連付けられます。この多重度は、1 対多関連の場合と正反対です。

コンテナ管理の多対 1 関連は、ejb-jar.xml デプロイメント・ディスクリプタで指定します(14-14 ページの「デプロイ XML の使用方法」を参照)。

詳細は、14-9 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean におけるコンテナ管理の関連性フィールドの構成」を参照してください。

デプロイ XML の使用方法

例 14-10 に、Employees と Department 間の双方向の多対 1 関連を定義する <ejb-relationship-role> 要素のペアを示します。単方向の関連の場合、適切な <ejb-relationship-role> 要素から cmr-field を削除します。

例 14-10 EJB 2.1 の双方向多対 1 関連の ejb-jar.xml

```
...
<relationships>
    <ejb-relation>
        <ejb-relation-name>Employee-Department</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>employees-belongto-dept</ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>DepartmentEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>dept</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>dept-has-employees</ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <cascade-delete/>
            <relationship-role-source>
                <ejb-name>LineItemEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>employees</cmr-field-name>
                <cmr-field-type>java.util.Collection</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
    </ejb-relation>
...
<relationships>
```

多対多関連の構成

多対多関連では、エンティティ Bean の複数のインスタンスが別のエンティティ Bean の複数のインスタンスに関連付けられます。

コンテナ管理の多対多関連は、ejb-jar.xml デプロイメント・ディスクリプタで指定します（14-14 ページの「デプロイ XML の使用方法」を参照）。

詳細は、14-9 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean におけるコンテナ管理の関連性フィールドの構成」を参照してください。

デプロイ XML の使用方法

例 14-11 に、Teams と Players 間の多対多関連を定義する <ejb-relationship-role> 要素のペアを示します。

例 14-11 EJB 2.1 の多対多関連の ejb-jar.xml

```
...
<relationships>
    <ejb-relation>
        <ejb-relationship-role>
            <ejb-relationship-role-name>team-has-players</ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>TeamEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>players</cmr-field-name>
                <cmr-field-type>java.util.Collection</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>player-has-teams</ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>PlayerEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>teams</cmr-field-name>
                <cmr-field-type>java.util.Collection</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
    </ejb-relation>
...
<relationships>
```

finder メソッドにおける遅延ロードの構成

各 finder メソッドでは、1 つ以上のオブジェクトが取得されます。デフォルト（遅延ロードの設定は「NO」）を使用する場合は、finder メソッドによって、単一の SQL select 文がデータベースに対して実行されます。コンテナ管理の永続性を備えたエンティティ Bean の場合、1 つ以上のオブジェクトがそのすべてのコンテナ管理の永続性フィールドとともに取得されます。このため、たとえば、`findAllEmployees` メソッドを実行した場合は、この finder によって、すべての従業員オブジェクトが各従業員オブジェクトのすべてのコンテナ管理の永続性フィールドとともに取得されます。

遅延ロードをオンにすると、finder 内で取得されたオブジェクトの主キーのみが戻されます。その後、実装内でオブジェクトにアクセスしたときのみ、OC4J によって、実際のオブジェクトが主キーに基づいてアップロードされます。`findAllEmployees` finder メソッドの例では、すべての従業員の主キーが `Collection` に戻されます。`Collection` 内のいずれかの従業員に初めてアクセスすると、OC4J では、主キーを使用してデータベースから単一の従業員オブジェクトが取得されます。取得するオブジェクト数が大量で、ローカル・キャッシュにすべてロードするとパフォーマンスが低下する恐れがある場合は、遅延ロード機能をオンにすることができます。

遅延ロードを使用する際にパフォーマンスを考慮する必要があります。複数のオブジェクトを取得しても使用するのはその中の一部である場合は、遅延ロードをオンにすることをお薦めします。また、`getPrimaryKey` メソッドを通じてのみオブジェクトを使用する場合も、遅延ロードをオンにすることをお薦めします。

デプロイ XML の使用方法

`findByPrimaryKey` メソッドで遅延ロードをオンにするには、次のように `findByPrimaryKey-lazy-loading` 属性を `true` に設定します。

```
<entity-deployment ... findByPrimaryKey-lazy-loading="true" ... >
```

カスタムの finder メソッドで遅延ロードをオンにするには、次のように、そのカスタムの finder に対する `<finder-method>` 要素の `lazy-loading` 属性を `true` に設定します。

```
<finder-method ... lazy-loading="true" ...>
  ...
</finder-method>
```

コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル・コールバック・メソッドの構成

次に、javax.ejb.EntityBean インタフェースでの指定に従って、コンテナ管理の永続性を備えたエンティティ Bean が実装する必要のある EJB 2.1 ライフ・サイクル・メソッドを示します（14-17 ページの「Java の使用方法」を参照）。

- ejbCreate
- ejbPostCreate
- ejbRemove
- ejbStore
- ejbLoad
- ejbActivate
- ejbPassivate

注意： EJB 2.1 を使用する場合は、すべてのエンティティ Bean コールバック・メソッドを実装する必要があります。何もアクションを行う必要がない場合は、空のメソッドを実装します。

詳細は、1-46 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル」を参照してください。

Java の使用方法

例 14-12 では、EJB 2.1 エンティティ Bean のライフ・サイクル・コールバック・メソッドの実装方法を説明します。

例 14-12 EJB 2.1 エンティティ Bean のライフ・サイクル・コールバック・メソッドの実装

```
public void ejbActivate() {
    // when bean is activated
}
```

15

Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の使用方法

この章では、Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean を使用するために構成する必要のある様々なオプションについて説明します。

表 15-1 に、これらのオプションをリストし、基本オプション（ほとんどのアプリケーションに適用可能）であるか拡張オプション（より特殊なアプリケーションに適用可能）であるかを示します。

詳細は、次を参照してください。

- 1-49 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean とは」
- 13-7 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の実装」

表 15-1 Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の構成オプション

オプション	タイプ
15-2 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の主キーの構成」	基本
15-4 ページの「Bean 管理の永続性を備えた読取り専用エンティティ Bean の構成」	拡張
15-6 ページの「Bean 管理の永続性を備えたエンティティ Bean のコミット・オプションの構成」	拡張
15-6 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の問合せの構成」	基本
15-8 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル・コールバック・メソッドの構成」	基本

Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の主キーの構成

Bean 管理の永続性を備えたすべての EJB 2.1 エンティティ Bean には主キー・フィールドが必要です。

主キーは、一般的な Java 型（15-2 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の主キー・フィールドの構成」を参照）またはユーザーが作成する特殊な型（15-3 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の主キー・クラスの構成」を参照）として構成できます。

詳細は、「Bean 管理の永続性を備えたエンティティ Bean の主キー」を参照してください。

Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の場合、通常は `ejbCreate` メソッドでユーザーが主キー値を割り当てます（15-8 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル・コールバック・メソッドの構成」を参照）。

Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の主キー・フィールドの構成

Bean 管理の永続性を備えた単純な EJB 2.1 エンティティ Bean の場合は、次のように主キーを一般的な Java 型として定義できます。

- 主キー・クラスの型を返すように Bean の `ejbCreate` メソッドをコーディングします（13-7 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の実装」を参照）。
- それを使用するようにデプロイ XML を構成します（15-2 ページの「デプロイ XML の使用方法」を参照）。

デプロイ XML の使用方法

例 15-1 に、主キーを一般的な Java 型 `Integer` として指定するように構成されている `ejb-jar.xml` ファイルの `<entity>` 要素の `<prim-key-class>` および `<primkey-field>` サブ要素を示します。

例 15-1 Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の Integer 型主キー・フィールド用の ejb-jar.xml

```
<enterprise-beans>
  <entity>
    <display-name>Employee</display-name>
    <ejb-name>EmployeeBean</ejb-name>
    <local-home>employee.EmployeeLocalHome</local-home>
    <local>employee.EmployeeLocal</local>
    <ejb-class>employee.EmployeeBean</ejb-class>
    <persistence-type>Bean</persistence-type>
    <prim-key-class>java.lang.Integer</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>Employee</abstract-schema-name>
    <cmp-field><field-name>empNo</field-name></cmp-field>
    <cmp-field><field-name>empName</field-name></cmp-field>
    <cmp-field><field-name>salary</field-name></cmp-field>
    <primkey-field>empNo</primkey-field>
    </entity>
...
</enterprise-beans>
```

Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の主キー・クラスの構成

主キーが一般的な Java データ型よりも複雑な場合は、独自の主キー・クラスを定義できます。

主キー・クラスには、次の特性が必要です。

- *<name>*PK という名前であること
- public および serializable であること
- 主キー・インスタンスを作成するためのコンストラクタを提供すること

クラスには、主キーの構成に使用するいくつかのインスタンス変数が含まれていることがあります。インスタンス変数には、次の特性が必要です。

- プリミティブ・オブジェクト型
- シリアライズ可能型
- SQL 型にマッピングできる型
- RMI-IIOP で有効な値タイプとなる型
- hashCode() および equals(Object) メソッドの適切な実装を提供する型

主キー・クラスを定義したら（15-3 ページの「Java の使用方法」を参照）、それを EJB 内で使用するために、次の処理を行う必要があります。

- 主キー・クラスの型を返すように Bean の ejbCreate メソッドをコーディングします（13-7 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の実装」を参照）。
- それを使用するようにデプロイ XML を構成します（15-4 ページの「デプロイ XML の使用方法」を参照）。

Java の使用方法

例 15-2 に、主キー・クラスの例を示します。

例 15-2 Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の主キー・クラスの実装

```
package employee;

import java.io.*;
import java.io.Serializable;
...

public class EmployeePK implements java.io.Serializable {
  public Integer empNo;

  public EmployeePK() {
    this.empNo = null;
  }

  public EmployeePK(Integer empNo) {
    this.empNo = empNo;
  }
}
```

デプロイ XML の使用方法

例 15-3 に示すように、ejb-jar.xml ファイルの <prim-key-class> 要素内で主キー・クラスを定義します。各主キー・クラス・インスタンス変数は、主キー・クラスで使用されるのと同じ変数名を使用して <cmp-field><field-name> 要素で定義します。

例 15-3 Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の主キー・クラスおよびそのインスタンス変数の ejb-jar.xml

```
<enterprise-beans>
  <entity>
    <description>no description</description>
    <display-name>EmployeeBean</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <local-home>employee.LocalEmployeeHome</home>
         <local>employee.LocalEmployee</remote>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Bean</persistence-type>
        <prim-key-class>employee.EmployeePK</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-version>2.x</cmp-version>
         <abstract-schema-name>Employee</abstract-schema-name>
        <cmp-field><field-name>empNo</field-name></cmp-field>
         <cmp-field><field-name>empName</field-name></cmp-field>
         <cmp-field><field-name>salary</field-name></cmp-field>
      </entity>
</enterprise-beans>
```

Bean 管理の永続性を備えた読取り専用エンティティ Bean の構成

コンテナ管理の永続性を備えたエンティティ Bean は読取り専用として構成できます。これにより、OC4J と取決めを行い、コンテナ管理の永続性を備えたエンティティ Bean の状態がアクティブ化の後も変化しないことを保証できます。コンテナ管理の永続性を備えた読取り専用エンティティ Bean とは異なり、Bean 管理の永続性を備えた読取り専用 Bean を更新しても例外はスローされません。

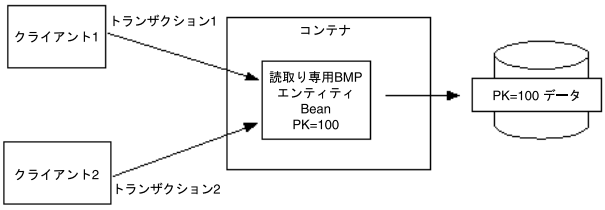
Bean 管理の永続性を備えたエンティティ Bean を読取り専用として構成した場合、OC4J はコミット・オプション A（15-6 ページの「Bean 管理の永続性を備えたエンティティ Bean のコミット・オプションの構成」を参照）の特殊ケースを使用して次のようにパフォーマンスを向上させます。

- インスタンスのキャッシング。
- アクティブ化の後に ejbLoad をコールしない。
- トランザクションのコミット時にインスタンスの更新または ejbStore のコールを行わない。

図 15-1 に示すように、主キーによって Bean 管理の永続性を備えた同じ読取り専用エンティティ Bean にアクセスする複数のクライアントには、1 つのインスタンスが割り当てられます。クライアント 1 とクライアント 2 の両方が、Bean 管理の永続性を備えた読取り専用エンティティ Bean の同じキャッシュ済インスタンスによって満たされます。Bean 管理の永続性を備えたエンティティ Bean は読取り専用であるため、両方のトランザクションをパラレルで処理できます。

この最適化を行わない場合、各クライアントには別々のインスタンスが割り当てられ、各インスタンスはすべてのライフ・サイクル・メソッドの実行を必要とします。

図 15-1 Bean 管理の永続性を備えた読取り専用エンティティ Bean およびコミット・オプション A



デプロイ XML の使用方法

例 15-4 に、Bean 管理の永続性を備えたエンティティ Bean を読取り専用として指定するように構成された `orion-ejb-jar.xml` ファイルの `entity-deployment` 要素の `locking-mode` 属性の `mode` を示します。

例 15-4 読取り専用の orion-ejb-jar.xml

```
<entity-deployment
    name=EmployeeBean"
    location="bmpapp/EmployeeBean"
    locking-mode="read-only"
>
...
</entity-deployment>
```

Bean 管理の永続性を備えたエンティティ Bean のコミット・オプションの構成

Bean 管理の永続性を備えたエンティティ Bean では、コミット・オプション A と C のどちらかを選択できます。

コミット・オプション A では、ejbLoad へのコールを延期することでパフォーマンスが向上します。

Bean 管理の永続性を備えた読取り専用エンティティ Bean でコミット・オプション A を使用するよう構成する場合（15-4 ページの「Bean 管理の永続性を備えた読取り専用エンティティ Bean の構成」を参照）、Bean 管理の永続性を備えた読取り専用エンティティ Bean のキャッシングを利用することでパフォーマンスをさらに向上させることができます（1-53 ページの「コミット・オプションおよび BMP アプリケーション」を参照）。

コミット・オプション C がデフォルトです。

詳細は、1-51 ページの「エンティティ Bean のコミット・オプション」を参照してください。

デプロイ XML の使用方法

例 15-5 に、orion-ejb-jar.xml ファイルの entity-deployment 要素の commit-option サブ要素の属性 mode を示します。有効な設定は、A および C です。number-of-buckets 属性は、許可されるキャッシュ・インスタンスの最大数であり、コミット・オプション A にのみ適用できます。

例 15-5 コミット・オプションの orion-ejb-jar.xml

```
<entity-deployment name=EmployeeBean" location="bmpapp/EmployeeBean" >
   <resource-ref-mapping name="jdbc/OracleDS" />
   <commit-option mode="A" number-of-buckets="10" />
</entity-deployment>
```

Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の問合せの構成

Bean 管理の永続性を備えたエンティティ Bean の ejbFindByPrimaryKey メソッドを実装する必要があります（15-7 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の ejbFindByPrimaryKey メソッドの実装」を参照）。オプションで、他の finder を構成できます（15-8 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の他の finder メソッドの実装」を参照）。

詳細は、第 16 章「EJB 2.1 問合せの実装」を参照してください。

Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の ejbFindByPrimaryKey メソッドの実装

ejbFindByPrimaryKey 実装は、Bean 管理の永続性を備えたすべてのエンティティ Bean に必要です。主な用途は、主キーが有効な Bean に対応しているかを確認することです。妥当性が検証されると、主キーをコンテナに返し、コンテナはそのキーを使用して Bean 参照をユーザーに返します。

このサンプルでは、従業員番号が有効であることを検証し、主キー（従業員番号）をコンテナに返します。主キーがクラスの場合、より複雑な検証が必要になります。

```
public EmployeePK ejbFindByPrimaryKey(EmployeePK pk)
    throws FinderException {
    if (pk == null || pk.empNo == null) {
        throw new FinderException("Primary key cannot be null");
    }
    try {
        conn = getConnection(dsName);
        ps = conn.prepareStatement(findByPKStatement);
        ps.setInt(1, pk.empNo.intValue());
        ps.executeQuery();
        ResultSet rs = ps.getResultSet();
        if (rs.next()) {
            pk.empNo = new Integer(rs.getInt(1));
            pk.empName = new String(rs.getString(2));
            pk.salary = new Float(rs.getFloat(3));
        }
        else {
            throw new FinderException("Failed to select this PK");
        }
    }
    catch (SQLException e) {
        throw new FinderException(e.getMessage());
    }
    catch (NamingException e) {
        System.out.println("Caught an exception 1 " + e.getMessage() );
        throw new EJBException(e.getMessage());
    }
    finally {
        try {
            ps.close();
            conn.close();
        }
        catch (SQLException e) {
            throw new EJBException(e.getMessage());
        }
    }
    return pk;
}
```

Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean の他の finder メソッドの実装

オプションで、1 つの ejbFindByPrimaryKey メソッド以外にも、他の finder メソッドを作成できます。

他の finder メソッドを作成するには、次のようにします。

1. finder メソッドをホーム・インタフェースに追加します。
2. Bean 管理の永続性を備えたエンティティ Bean の Bean 実装で、finder メソッドを実装します。

finder メソッドでは、WHERE 句に従って、1 つ以上の Bean を取得できます。複数の Bean を返す場合、Bean の finder メソッドは、主キーの Collection を返す必要があります。これらの finder メソッドは、ユーザーに返す必要のあるすべてのエンティティ Bean の主キーのみ収集する必要があります。コンテナは、Collection 内に存在する各エンティティ Bean への参照（複数の参照が返された場合）、または単一のクラス型への参照へ、主キーをマッピングします。

次の例では、すべての従業員のレコードを返す finder メソッドの実装を示します。

```
public Collection ejbFindAll() throws FinderException {
  ArrayList recs = new ArrayList();

  ps = conn.prepareStatement("SELECT EMPNO FROM EMPLOYEEBEAN");
  ps.executeQuery();
  ResultSet rs = ps.getResultSet();

  int i = 0;

  while (rs.next()) {
   retEmpNo = new Integer(rs.getInt(1));
   recs.add(retEmpNo);
  }

  ps.close();
  return recs;
}
```

Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル・コールバック・メソッドの構成

次に、javax.ejb.EntityBean インタフェースでの指定に従って、Bean 管理の永続性を備えたエンティティ Bean が実装する必要のある EJB 2.1 ライフ・サイクル・メソッドを示します（15-9 ページの「Java の使用方法」を参照）。

- ejbCreate
- ejbPostCreate
- ejbRemove
- ejbStore
- ejbLoad
- ejbActivate
- ejbPassivate

Bean 管理の永続性を備えたエンティティ Bean の場合は、すべてのライフ・サイクル・メソッドの完全な実装をユーザーが提供します。

詳細は、1-49 ページの「Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean のライフ・サイクル」を参照してください。

Java の使用方法

例 15-6 では、EJB 2.1 エンティティ Bean のライフ・サイクル・コールバック・メソッドの実装方法を説明します。

例 15-6 EJB 2.1 エンティティ Bean のライフ・サイクル・コールバック・メソッドの実装

```
public void ejbActivate() {
    // when bean is activated
}
```

16

EJB 2.1 問合せの実装

この章の内容は次のとおりです。

- EJB 2.1 EJB QL finder メソッドの実装
- EJB 2.1 EJB QL select メソッドの実装
- OC4J EJB 2.1 EJB QL 拡張

詳細は、次を参照してください。

- 1-53 ページの「EJB 2.1 エンティティ Bean の問合せ方法」
- 第 13 章「EJB 2.1 エンティティ Bean の実装」

注意： OC4J EJB QL アプリケーションの例は、`http://www.oracle.com/technology/sample_code/tech/java/ejb_corba/ejbql/Readme.html` を参照してください。

EJB 2.1 EJB QL finder メソッドの実装

次の手順では、EJB 2.1 EJB QL finder メソッドの実装方法を説明します。

finder メソッドを実装する前に、OC4J に用意されている事前定義およびデフォルトの finder について検討します（1-56 ページの「事前定義の TopLink finder」および 1-57 ページの「デフォルトの TopLink finder」を参照）。

詳細は、1-56 ページの「finder メソッドについて」を参照してください。

1. finder メソッドをホーム・インタフェースに定義します（16-3 ページの「Java の使用方法」を参照）。

   事前定義またはデフォルトの finder（1-56 ページの「事前定義の TopLink finder」および 1-57 ページの「デフォルトの TopLink finder」を参照）のみ公開している場合は、すでに定義しています。

   カスタム finder を公開している場合は、手順 2 に進みます。

2. `ejb-jar.xml` ファイルを構成します（16-4 ページの「デプロイ XML の使用方法」を参照）。

   注意： この処理は、ここで説明するように手動で行うか、TopLink Workbench を使用して（16-5 ページの「TopLink Workbench の使用方法」を参照）この手順を自動化し、高度な TopLink finder 構成を利用できます。

   a. EJB QL 問合せで参照することを計画している各エンティティ Bean について、`<entity>` 要素の `<abstract-schema-name>` サブ要素を構成します。

      `<abstract-schema-name>` サブ要素では、EJB QL 文のエンティティ Bean を識別する名前を定義します。たとえば、`EmpBean` という名前のエンティティ Bean クラスがある場合、その `<abstract-schema-name>` を `Employee` として定義すると、EJB QL 文で名前 `Employee` を使用した場合にコンテナはその名前を `EmpBean` エンティティ Bean にマッピングします（例 16-2 を参照）。

   b. EJB ホーム・インタフェースで公開した各 finder メソッドの `<query>` 要素を定義します。

      注意： `findByPrimaryKey` など、事前定義またはデフォルトの finder の `<query>` 要素は定義しないでください。

      `<query>` 要素には次のサブ要素があります。

      - `<description>`: オプションの説明テキスト。
      - `<query-method>`: finder メソッドを記述し、次のサブ要素を含みます。

        `<method-name>`: finder メソッドを識別します。この要素は、ホーム・インタフェースで定義したメソッド名と同じ名前で構成します。

        `<method-params>`: finder が引数を受け取る場合は、この要素を定義し、各引数について、引数型を与える `<method-param>` サブ要素を定義します。引数の型と順序は、この finder のシグネチャで指定されている型および順序と一致している必要があります。
      - `<ejb-ql>`: このメソッドの EJB QL 文が含まれます。

        完全な問合せ、または条件文（`WHERE` 句）のみを定義できます。

        finder メソッドが `Collection` を返す場合、重複した項目を返さないようにするには、EJB QL 文で `DISTINCT` キーワードを指定します。

        EJB QL でパラメータ（`<method-params>` で指定）を使用するには、`<integer>?` 表記を使用します。`<integer>` は 1 から開始します。たとえば、`?1` は最初の `<method-param>` 要素に対応し、`?2` は 2 番目の `<method-param>`

要素に対応します。以下も同様です（例 16-2 の findAllByEmpName finder を参照）。

この EJB を別の EJB に関連付ける EJB QL 文を定義するには、最初に適切なコンテナ管理の関連性を定義する必要があります。例 16-2 の findByDeptNo finder には、<ejb-relation-name> Employee-Departments との関連が必要です。詳細は、14-9 ページの「コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean におけるコンテナ管理の関連性フィールドの構成」を参照してください。

Java の使用方法

例 16-1 に、EmpBeanHome というリモート・ホーム・インタフェースを示します。

例 16-1 コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のリモート・ホーム・インタフェースの finder メソッド

```
package cmpapp;

import javax.ejb.*;
import java.rmi.*;

public interface EmpBeanHome extends EJBHome {
    public EmpBean create(Integer empNo, String empName) throws CreateException;

    /**
     * Finder methods. These are implemented by the container. You can
     * customize the functionality of these methods in the deployment
     * descriptor through EJB-QL.
     **/

// Predefined Finders: <query> element in ejb-jar.xml not required

    public Topic findByPrimaryKey(Integer key) throws FinderException;
    public Collection findManyBySQL(String sql, Vector args) throws FinderException

// Default Finder: <query> element in ejb-jar.xml not required

    public Topic findByEmpNo(Integer empNo) throws FinderException;

// Custom Finders: <query> element is required in ejb-jar.xml

    public Collection findAllRegionalEmployees(Integer empNo) throws FinderException;
    public Collection findAllByEmpName(String empName) throws FinderException;
    public Topic findByDeptNo(Integer deptNo) thorws FinderException
    public Collection findAllBetweenSalaries(Integer lowSalary, Integer highSalary);

}
```

デプロイ XML の使用方法

例 16-2 に、例 16-1 に示したホーム・インタフェースで宣言される finder の ejb-jar.xml を示します。

例 16-2 EJB 2.1 EJB QL finder の ejb-jar.xml

```
<enterprise-beans>
    <entity>
        <display-name>EmpBean</display-name>
        <ejb-name>EmpBean</ejb-name>
        ...
        <abstract-schema-name>Employee</abstract-schema-name>
        <cmp-field><field-name>empNo</field-name></cmp-field>
        <cmp-field><field-name>empName</field-name></cmp-field>
        <cmp-field><field-name>salary</field-name></cmp-field>
        <primkey-field>empNo</primkey-field>
        <prim-key-class>java.lang.Integer</prim-key-class>
        ...
        <query>
            <description>Regional employees have empNo greater than 10000</description>
            <query-method>
                    <method-name>findAllRegionalEmployees</method-name>
                    <method-params></method-params>
            </query-method>
            <ejb-ql>SELECT OBJECT(e) FROM Employee e WHERE e.empNo > 10000</ejb-ql>
        </query>
        <query>
            <description>Find all employees with the given name</description>
            <query-method>
                <method-name>findAllByEmpName</method-name>
                <method-params>
                    <method-param>java.lang.String</method-param>
                </method-params>
            </query-method>
            <ejb-ql>SELECT OBJECT(e) FROM Employee e WHERE e.empName = ?1</ejb-ql>
        </query>
        <query>
            <description>Relationship finder</description>
            <query-method>
                <method-name>findByDeptNo</method-name>
                <method-params>
                    <method-param>java.lang.Integer</method-param>
                </method-params>
            </query-method>
            <ejb-ql>
              SELECT DISTINCT OBJECT(e) From Employee e, IN (e.dept) AS d WHERE d.deptNo = ?1
           </ejb-ql>
        </query>
        <query>
            <description>Find all employees with salaries in the given range</description>
            <query-method>
                <method-name>findAllBetweenSalaries</method-name>
                <method-params>
                    <method-param>java.lang.Integer</method-param>
                    <method-param>java.lang.Integer</method-param>
                </method-params>
            </query-method>
            <ejb-ql>
                SELECT  OBJECT (e) FROM Employee e WHERE e.salary BETWEEN ?1 and ?2
            </ejb-ql>
        </query>
    ...
    </entity>
...
</enterprise-beans>
```

```
<relationships>
    <ejb-relation>
        <ejb-relation-name>Employee-Departments</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>Employee-has-Departments</ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>Department</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>dept</cmr-field-name>
                <cmr-field-type>java.lang.Integer</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
    <ejb-relation>
...
<relationships>
```

TopLink Workbench の使用方法

TopLink Workbench を使用して、`toplink-ejb-jar.xml` ファイルをカスタム TopLink finder で構成し、`ejb-jar.xml` ファイルを更新できます。

詳細は、次を参照してください。

- 『Oracle TopLink 開発者ガイド』のファインダの作成に関する項
- 『Oracle TopLink 開発者ガイド』のディスクリプタ・レベルでの名前付き問合せの構成に関する項

EJB 2.1 EJB QL select メソッドの実装

次の手順では、EJB 2.1 EJB QL select メソッドの実装方法を説明します。

詳細は、1-58 ページの「select メソッドについて」を参照してください。

1. select メソッドを抽象エンティティ Bean クラスの `public abstract` メソッドとして定義します（16-7 ページの「Java の使用方法」を参照）。
2. `ejb-jar.xml` ファイルで、次の操作を行います（16-8 ページの「デプロイ XML の使用方法」を参照）。
   a. EJB QL 問合せで参照することを計画している各エンティティ Bean について、`<entity>` 要素の `<abstract-schema-name>` サブ要素を構成します。

      `<abstract-schema-name>` サブ要素では、EJB QL 文のエンティティ Bean を識別する名前を定義します。たとえば、`EmpBean` という名前のエンティティ Bean クラスがある場合、その `<abstract-schema-name>` を `Employee` として定義すると、EJB QL 文で名前 `Employee` を使用した場合にコンテナはその名前を `EmpBean` エンティティ Bean にマッピングします。

   b. EJB ホーム・インタフェースで公開する各 select メソッドの `<query>` 要素を定義します。

      完全な問合せ、または条件文（`WHERE` 句）のみを定義できます。

      select メソッドが `Collection` を返す場合、重複した項目を返さないようにするには、EJB QL 文で `DISTINCT` キーワードを指定します。

      `<query>` 要素には、次の 2 つの主要な要素があります。

      - `<method-name>` 要素は select メソッドを識別します。この要素は、Bean クラスで定義した名前と同じ名前で構成します。
      - `<ejb-ql>` 要素には、このメソッドの EJB QL 文が含まれます。

   c. 問合せが CMR 値の `Collection` を返す場合は、返すインタフェースの型を決定します。

      `ejb-jar.xml` ファイルの `<result-type-mapping>` 要素では、select メソッドの戻り型を決定します。`EJBObjects` を返すにはフラグを `Remote` に設定し、`EJBLocalObjects` を返すには `Local` に設定します。

Java の使用方法

例 16-3 に、select メソッドのあるコンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の UserAccountBean という抽象エンティティ Bean クラスを示します。

例 16-3 select メソッドのあるコンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean の実装

```
package oracle.otnsamples.ejbql;

import javax.ejb.*;
import java.util.*;

public abstract class UserAccountBean implements EntityBean {

    // Non-Persistent State

    protected EntityContext ctx;

    /**
     * Begin abstract get/set methods. Container-managed
     * persistent fields are specified in the ejb-jar.xml
     * deployment descriptor.
     */

    public abstract Long getAccountnumber();
    public abstract void setAccountnumber(Long newAccountnumber);

    public abstract Long getCreditlimit();
    public abstract void setCreditlimit(Long newCreditlimit);

    /**
     * Select methods. These are implemented by the container. You can
     * customize the functionality of these methods in the deployment
     * descriptor through EJB-QL.
     *
     * These methods are NOT exposed in the bean's home interface.
     */

    public abstract Long ejbSelectCreditLimit(Long accountnumber) throws FinderException;
    public abstract Collection ejbSelectByTopAccounts() throws FinderException;

    /**
     * Begin buisness logic methods that use select methods.
     *
     * These methods are exposed in the bean's home interfaces.
     */

    /**
     * Method to perform post-processing operations on all the
     * UserAccounts retrieved by calling ejbSelectByTopAccounts. This
     * method further process the retrieved UserAccounts and checks
     * for the Accounts with TopCredits (credit limits) and returns the
     * collection of input number of UserAccounts.
     * Post-processing information within the EJB container itself
     * has the following two advantages:
     *     1) It improves performance as the application can now leverage
     *            the advantage of the vast resources available to the server.
     *     2) The data-processing code should go into the business logic
     *            and not the Web-tier. This helps in maintaining the code.
     * Consider these advantages when deciding between ejbFind and
     * ejbSelect methods.
     *
     * @return Collection of <input number of> Top (credited) UserAccounts
     */
    public Collection ejbHomeTopAccounts(String accountNumbers) throws FinderException {
        // Invoke the ejbSelect method and get all the Account Information.
```

```
        Collection collection = this.ejbSelectByTopAccounts();
        ...
        return topAccounts;
    }

    /**
     * Method to call ejbSelectCreditLimit and return the credit limit value
     * for the input accountnumber without post-processing.
     * Please note that this method returns a Long instead of a collection
     * that is returned normally by the EJB container. This is a major
     * advantage of ejbSelect methods. Using these methods, You can return
     * an object from 'within' the CMP instead of 'the' CMP. This way, the
     * application uses the server and the EJB container resources more
     * effeciently.
     *
     * @return Credit Limit of the input UserAccount
     */
    public Long ejbHomeCreditLimit(Long accountnumber) throws FinderException {
        // Return the Credit Limit of the specified Account
        return this.ejbSelectCreditLimit(accountnumber);
    }
    ...
}
```

デプロイ XML の使用方法

例 16-4 に、例 16-3 に示した抽象エンティティ Bean クラスで定義されている select メソッドの ejb-jar.xml ファイルを示します。

例 16-4 EJB 2.1 EJB QL select メソッドの ejb-jar.xml

```
<enterprise-beans>
  <entity>
    <description>Entity Bean ( CMP )</description>
    <display-name>UserAccount</display-name>
    <ejb-name>UserAccount</ejb-name>
    <local-home>oracle.otnsamples.ejbql.UserAccountLocalHome</local-home>
    <local>oracle.otnsamples.ejbql.UserAccount</local>
    <ejb-class>oracle.otnsamples.ejbql.UserAccountBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Long</prim-key-class>
    <abstract-schema-name>UserAccount</abstract-schema-name>
    <cmp-field>
      <field-name>accountnumber</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>creditlimit</field-name>
    </cmp-field>
    <primkey-field>accountnumber</primkey-field>
    <query>
      <description>Selects all accounts and post-process to find top accounts</description>
      <query-method>
        <method-name>ejbSelectByTopAccounts</method-name>
      </query-method>
      <ejb-ql>select distinct object(ua) from UserAccount ua</ejb-ql>
    </query>
    <query>
      <description>Retrieves the Credit Limit for an Account</description>
      <query-method>
        <method-name>ejbSelectCreditLimit</method-name>
        <method-params>
          <method-param>java.lang.Long</method-param>
        </method-params>
      </query-method>
      <ejb-ql>
          select ua.creditlimit from UserAccount ua where ua.accountnumber = ?1
```

```
      </ejb-ql>
    </query>
  </entity>
</enterprise-beans>
```

TopLink Workbench の使用方法

TopLink Workbench を使用して、`toplink-ejb-jar.xml` ファイルをカスタム TopLink `ejbSelect` メソッドで構成し、`ejb-jar.xml` ファイルを更新できます。

詳細は、『Oracle TopLink 開発者ガイド』のファインダの作成に関する項を参照してください。

OC4J EJB 2.1 EJB QL 拡張

EJB 2.1 では平方根、日付、時刻およびタイムスタンプ型はサポートされませんが、OC4J では、EJB 2.1 でこれらの型をサポートするために次のような独自の EJB QL 拡張を提供します。

- `SQRT(v)`: double プリミティブ型と `java.lang.Double` 型の両方が、引数に対してサポートされます（例 16-5 を参照）。
- 次の日付、時刻およびタイムスタンプ型は、等式などの EJB QL バイナリ式で使用できます。
  - `java.util.Date`（例 16-6 を参照）
  - `java.sql.Date`（例 16-7 を参照）
  - `java.sql.Time`（例 16-8 を参照）
  - `java.sql.Timestamp`（例 16-9 を参照）

> **注意：** これらの型は、EJB 3.0 EJB QL では完全にサポートされています。

例 16-5 SQRT の EJB 2.1 EJB QL 拡張の使用方法

```
<query>
  <query-method>
    <method-name>ejbSelectDoubleTypeSqrt</method-name>
    <method-params>
      <method-param>double</method-param>
    </method-params>
  </query-method>
  <result-type-mapping>Remote</result-type-mapping>
  <ejb-ql>
    SELECT OBJECT(a) FROM Dept a WHERE a.deptDoubleType = SQRT(?1)
  </ejb-ql>
</query>
```

例 16-6 java.util.Date の EJB 2.1 EJB QL 拡張の使用方法

```
<query>
  <query-method>
   <method-name>ejbSelectDate</method-name>
   <method-params>
    <method-param>java.util.Date</method-param>
   </method-params>
  </query-method>
  <result-type-mapping>Remote</result-type-mapping>
  <ejb-ql>
    SELECT OBJECT(a) FROM Dept a WHERE a.deptDate = ?1
   </ejb-ql>
</query>
```

例 16-7 java.sql.Date の EJB 2.1 EJB QL 拡張の使用方法

```
<query>
  <query-method>
   <method-name>ejbSelectSqlDate</method-name>
   <method-params>
    <method-param>java.sql.Date</method-param>
   </method-params>
  </query-method>
  <result-type-mapping>Remote</result-type-mapping>
  <ejb-ql>
    SELECT OBJECT(a) FROM Dept a WHERE a.deptSqlDate = ?1
  </ejb-ql>
</query>
```

例 16-8 java.sql.Time の EJB 2.1 EJB QL 拡張の使用方法

```
<query>
  <query-method>
   <method-name>findByTimestamp</method-name>
   <method-params>
    <method-param>java.sql.Time</method-param>
   </method-params>
  </query-method>
  <result-type-mapping>Remote</result-type-mapping>
  <ejb-ql>
    SELECT OBJECT(a) FROM Dept a WHERE a.deptTime = ?1
  </ejb-ql>
</query>
```

例 16-9 java.sql.Timestamp の EJB 2.1 EJB QL 拡張の使用方法

```
<query>
  <query-method>
   <method-name>findByTimestamp</method-name>
   <method-params>
    <method-param>java.sql.Timestamp</method-param>
   </method-params>
  </query-method>
  <result-type-mapping>Remote</result-type-mapping>
  <ejb-ql>
    SELECT OBJECT(a) FROM Dept a WHERE a.deptTimestamp = ?1
  </ejb-ql>
</query>
```

第 VII 部

EJB 2.1 メッセージドリブン Bean

第 VII 部では、EJB 2.1 メッセージドリブン Bean の実装および構成の手順に関する情報を示します。概念的な情報は、第 I 部「EJB の概要」を参照してください。

第 VII 部は次の各章で構成されています。

- 第 17 章「EJB 2.1 メッセージドリブン Bean の実装」
- 第 18 章「EJB 2.1 メッセージドリブン Bean の使用方法」

17

EJB 2.1 メッセージドリブン Bean の実装

この章では、EJB 2.1 メッセージドリブン Bean（MDB）の実装方法を説明します。

詳細は、次を参照してください。

- 1-59 ページの「メッセージドリブン Bean とは」
- 第 18 章「EJB 2.1 メッセージドリブン Bean の使用方法」

EJB 2.1 MDB の実装

表 17-1 に、EJB 2.1 メッセージドリブン Bean の重要な構成要素をまとめ、次の手順でこれらの構成要素の実装方法を説明します。一般的な実装は、17-3 ページの「Java の使用方法」を参照してください。

表 17-1 EJB 2.1 MDB エンティティ Bean の構成要素

構成要素	説明
Bean の実装	このクラスは public として宣言される必要があり、空のデフォルトの public コンストラクタ、引数のない 1 つの public void ejbCreate メソッドが含まれ、finalize() メソッドは含まれません。 ライフ・サイクル・メソッド ejbRemove の空の実装および setMessageDrivenContext メソッドの実装を提供するように javax.ejb.MessageDrivenBean を実装します。 onMessage メソッドの実装を提供するように javax.jms.MessageListener を実装します。

詳細は、1-59 ページの「メッセージドリブン Bean とは」を参照してください。

注意： EJB コード例は、http://www.oracle.com/technology/tech/java/oc4j/demos からダウンロードできます。

EJB 2.1 メッセージドリブン Bean を実装するには、次のようにします。

1. MDB エンティティ Bean を実装します。
   a. 引数のない public コンストラクタを実装します。
   b. ビジネス・ロジックに使用される Bean またはパッケージに対してプライベートであるメソッドを実装します。これには、パブリック・メソッドがリクエストされた作業の完了に使用するプライベート・メソッドも含まれます。
   c. ejbCreate メソッドを実装します。コンテナは、MDB をインスタンス化するときに、このメソッドを起動します。

      ejbCreate メソッドの戻り型は void です。
   d. 各 javax.ejb.MessageDrivenBean インタフェース・コンテナのコールバック・メソッドの空の実装を提供します。

      詳細は、18-11 ページの「EJB 2.1 MDB のライフ・サイクル・コールバック・メソッドの構成」を参照してください。
   e. MessageDrivenContext のインスタンスを取得する setMessageDrivenContext メソッドを実装します（17-7 ページの「setMessageDrivenContext メソッドの実装」を参照）。
   f. 適切なメッセージ・リスナー・インタフェースを実装します。

      JMS メッセージドリブン Bean では、javax.jms.MessageListener インタフェースを実装して onMessages メソッドにシグネチャを提供します。

      ```
      public void onMessage(javax.jms.Message message)
      ```

      JMS 以外のメッセージ・サービス・プロバイダでは、指定する 1 つまたは複数のメッセージ・リスナー・インタフェースを実装します。

      このメソッドは、着信メッセージを処理します。ほとんどの MDB は、メッセージをキューまたはトピックから受信し、メッセージ内のリクエストを処理するために、エンティティ Bean を起動します。

2. メッセージ・サービス・プロバイダ情報を構成します（17-5 ページの「デプロイ XML の使用方法」を参照）。

   a. 使用するメッセージ・コネクション・ファクトリおよび Destination を EJB デプロイメント・ディスクリプタ（ejb-jar.xml）で定義します。永続的なサブスクリプションまたはメッセージ・セレクタを使用するかどうかを定義します。

      詳細は、次を参照してください。

      – 18-4 ページの「直接メッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」

      – 18-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」

   b. リソース参照を使用する場合は、これらを ejb-jar.xml ファイルで定義し、OC4J 固有のデプロイメント・ディスクリプタ（orion-ejb-jar.xml）で実際の JNDI 名にマッピングします。

   c. MDB でコンテナ管理のトランザクション境界が使用される場合は、ejb-jar.xml ファイルの <container-transaction> 要素に onMessage メソッドを指定します。

      MDB に関するすべての手順は、onMessage メソッドに記述されている必要があります。MDB はステートレスであるため、onMessage メソッドがすべての作業を実行する必要があります。

      一般に、ejbCreate メソッドに、メッセージ・サービス接続およびセッションを作成しないでください。

      注意： ただし、OEMS JMS を使用している場合（2-27 ページの「OEMS JMS: メモリー内またはファイルベース・プロバイダ」を参照）は、JMS コネクションとセッションを ejbCreate メソッドで作成し、ejbRemove メソッドで破棄することで、MDB を最適化できます。

Java の使用方法

例 17-1 に、EJB 2.1 MDB の一般的な実装を示します。

例 17-1 EJB 2.1 MDB の実装

```
import java.util.*;
import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;

public class rpTestMdb implements MessageDrivenBean, MessageListener {

    private QueueConnection         m_qc    = null;
    private QueueSession            m_qs    = null;
    private QueueSender             m_snd   = null;
    private MessageDrivenContext    m_ctx   = null;

    // Constructor, which is public and takes no arguments
    public rpTestMdb() {
    }

    /**
     * Begin private methods. The following methods
     * are used internally.
     */

...

    /**
```

```
 * Begin EJB-required methods. The following methods are called
 * by the container, and never called by client code.
 */

/**
 * ejbCreate method, declared as public (but not final or
 * static), with a return type of void, and with no arguments.
 */
public void ejbCreate() {
}

public void setMessageDrivenContext(MessageDrivenContext ctx) {
  // As with all enterprise beans, you must set the context in order to be
  // able to use it at another time within the MDB methods
  m_ctx = ctx;
}

// life cycle Methods

public void ejbRemove() {
}


/**
 * JMS MessageListener-required methods. The following
 * methods are called by the container, and never called by
 * client code.
 */

// Receives the incoming Message and displays the text.
public void onMessage(Message msg) {
   // MDB does not carry state for an individual client
   try {
        Context ctx = new InitialContext();
        // 1. Retrieve the QueueConnectionFactory using a
        // resource reference defined in the ejb-jar.xml file.
        QueueConnectionFactory qcf = (QueueConnectionFactory)
            ctx.lookup("java:comp/env/jms/myQueueConnectionFactory");
        ctx.close();

        // 2. Create the queue connection
        m_qc = qcf.createQueueConnection();
        // 3. Create the session over the queue connection.
        m_qs = m_qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
        // 4. Create the sender to send messages over the session.
        m_snd = m_qs.createSender(null);

        // When the onMessage method is called, a message has been sent.
        // You can retrieve attributes of the message using the Message object.
        String txt = ("mdb rcv: " + msg.getJMSMessageID());
        System.out.println(txt + " redel="
            + msg.getJMSRedelivered() + " cnt="
            + msg.getIntProperty("JMSXDeliveryCount"));

        // Create a new message using the createMessage method.
        // To send it back to the originator of the other message,
        // set the String property of "RECIPIENT" to "CLIENT."
        // The client only looks for messages with string property CLIENT.
        // Copy the original message ID into new msg's Correlation ID for
        // tracking purposes using the setJMSCorrelationID method. Finally,
        // set the destination for the message using the getJMSReplyTo method
        // on the previously received message. Send the message using the
        // send method on the queue sender.

        // 5. Create a message using the createMessage method
        Message rmsg = m_qs.createMessage();
```

```
            // 6. Set properties of the message.
            rmsg.setStringProperty("RECIPIENT", "CLIENT");
            rmsg.setIntProperty("count", msg.getIntProperty("JMSXDeliveryCount"));
            rmsg.setJMSCorrelationID(msg.getJMSMessageID());
            // 7. Retrieve the reply destination.
            Destination d = msg.getJMSReplyTo();
            // 8. Send the message using the send method of the sender.
            m_snd.send((Queue) d, rmsg);
            System.out.println(txt + " snd: " + rmsg.getJMSMessageID());
            // close the connection
            m_qc.close();
        }
        catch (Throwable ex) {
            ex.printStackTrace();
        }
    }
}
```

デプロイ XML の使用方法

ejb-jar.xml ファイルを使用している場合は、message-driven 要素内で、MDB の名前、クラス、JNDI 参照および JMS の Destination タイプ（キューまたはトピック）を定義します。トピックが指定されている場合は、それが永続的であるかどうかを定義します。リソース参照を使用している場合は、コネクション・ファクトリと Destination オブジェクトの両方に対してリソース参照を定義します。

例 17-2 に、例 17-1 に示した MDB に対応する ejb-jar.xml ファイルの message-driven 要素を示します。

次の点に注意してください。

- MDB 名は <ejb-name> 要素で指定されます。
- MDB クラスは <ejb-class> 要素で定義されます。この要素によって、<message-driven> 要素が特定の MDB 実装に結び付けられます。
- JMS の Destination タイプは、<message-driven-destination><destination-type> 要素で指定されている Queue です。
- メッセージ・セレクタによって、この MDB は、RECIPIENT が MDB のメッセージのみを受信することが指定されます。

> **注意：** このタイプ定義でトピックも指定できます。タイプに Topic を指定した場合は、トピックの永続性も定義できます。これを指定するには、<message-driven-destination> の <subscription-durability> 要素を Durable または nonDurable に設定します。

- 使用するトランザクションのタイプは、<transaction-type> 要素で定義されます。値は、Container または Bean のいずれかです。Container を指定した場合は、<container-transaction> 要素内で CMT サポート・タイプを指定して onMessage メソッドを定義します。
- コネクション・ファクトリのリソース参照は <resource-ref> 要素で定義されます。Destination オブジェクトのリソース参照は <resource-env-ref> 要素で定義されます。

例 17-2 EJB 2.1 MDB の ejb-jar.xml

```
...
    <enterprise-beans>
        <message-driven>
            <display-name>testMdb</display-name>
            <ejb-name>testMdb</ejb-name>
            <ejb-class>rpTestMdb</ejb-class>
            <transaction-type>Container</transaction-type>
            <message-selector>RECIPIENT='MDB'</message-selector>
            <message-driven-destination>
                <destination-type>javax.jms.Queue</destination-type>
            </message-driven-destination>
            <resource-ref>
                <description>description</description>
                <res-ref-name>jms/myQueueConnectionFactory</res-ref-name>
                <res-type>javax.jms.QueueConnectionFactory</res-type>
                <res-auth>Application</res-auth>
                <res-sharing-scope>Shareable</res-sharing-scope>
            </resource-ref>
            <resource-env-ref>
               <resource-env-ref-name>jms/persistentQueue
             </resource-env-ref-name>
             <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
            </resource-env-ref>
        </message-driven>
    </enterprise-beans>

    <assembly-descriptor>
        <container-transaction>
            <method>
                <ejb-name>testMdb</ejb-name>
                <method-name>onMessage</method-name>
                <method-params>
                    <method-param>javax.jms.Message</method-param>
                </method-params>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
...
```

キューのかわりに永続的な Topic を構成しようとしている場合は、<message-driven-destination> 要素を例 17-3 のように構成します。

例 17-3 永続的なトピックの EJB 2.1 MDB の ejb-jar.xml

```
<message-driven-destination>
  <destination-type>javax.jms.Topic</destination-type>
  <subscription-durability>Durable</subscription-durability>
</message-driven-destination>
```

詳細は、18-4 ページの「直接メッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」を参照してください。

setMessageDrivenContext メソッドの実装

MDB のインスタンスは、このメソッドを使用して、コンテキストへの参照を維持します。メッセージドリブン Bean には、コンテナによって維持され、Bean から使用可能なコンテキストが存在します。メッセージドリブン・コンテキスト内のメソッドを使用して、セキュリティおよびトランザクションのロールなどの Bean に関する情報の取得が、Bean によって行われる場合があります。Bean に関してコンテキストから取得可能なすべての情報は、Sun 社の EJB 仕様を参照してください。

コンテナは、最初に Bean をインスタンス化した後、`setMessageDrivenContext` メソッドを起動して、Bean からコンテキストを取得できるようにします。コンテナは、トランザクション・コンテキストからはこのメソッドをコールしません。この時点で Bean がコンテキストを保存しなかった場合、Bean は二度とコンテキストにアクセスできなくなります。

例 17-4 に、メッセージドリブン・コンテキストを `ctx` 変数に格納する MDB を示します。

例 17-4 setMessageDrivenContext メソッドの実装

```
import javax.ejb.*;

public class myBean implements MessageDrivenBean, MessageListener {

   MessageDrivenContext m_ctx;

    // setMessageDrivenContext method
    public void setMessageDrivenContext(MessageDrivenContext ctx) {
      // As with all enterprise beans, you must set the context in order to be
      // able to use it at another time within the MDB methods
      m_ctx = ctx;
    }

   // other methods in the bean
}
```

18

EJB 2.1 メッセージドリブン Bean の使用方法

この章では、EJB 2.1 メッセージドリブン Bean を使用するために構成する必要のある様々なオプションについて説明します。

表 18-1 に、これらのオプションをリストし、基本オプション（ほとんどのアプリケーションに適用可能）であるか拡張オプション（より特殊なアプリケーションに適用可能）であるかを示します。

詳細は、次を参照してください。

- 1-59 ページの「メッセージドリブン Bean とは」
- 第 17 章「EJB 2.1 メッセージドリブン Bean の実装」

表 18-1 EJB 2.1 メッセージドリブン Bean の構成オプション

オプション	タイプ
18-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」	基本
18-4 ページの「直接メッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」	基本
18-6 ページの「Windows オペレーティング・システムでの高速アンデプロイのための MDB の構成」	拡張
18-6 ページの「Oracle RAC フェイルオーバー用の MDB の構成」	拡張
31-5 ページの「Bean インスタンスのプール・サイズの構成」	基本
21-9 ページの「メッセージドリブン Bean のトランザクション・タイムアウトの構成」	拡張
18-8 ページの「パラレル・メッセージ処理の構成」	拡張
18-10 ページの「EJB 2.1 MDB の接続障害リカバリの構成」	拡張
18-11 ページの「EJB 2.1 MDB のライフ・サイクル・コールバック・メソッドの構成」	基本

J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成

Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してメッセージ・サービス・プロバイダにアクセスするよう EJB 2.1 MDB を構成できます。

これを行うには、デプロイ XML を使用します（18-2 ページの「デプロイ XML の使用方法」を参照）。

> **注意：** メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」を参照してください。

OC4J では、2 フェーズ・コミット（2PC）トランザクション用の XA ファクトリと、2PC を必要としないトランザクション用の非 XA ファクトリの両方がサポートされます。

詳細は、次を参照してください。

- 2-26 ページの「Oracle JMS コネクタ : J2EE Connector Architecture（J2CA）ベース・プロバイダ」
- 2-31 ページの「メッセージ・サービス構成オプション : アノテーションまたは XML の選択と属性またはアクティブ化構成プロパティの選択」
- 2-24 ページの「グローバル・トランザクションまたは 2 フェーズ・コミット（2PC）トランザクションへの参加方法」

デプロイ XML の使用方法

J2CA リソース・アダプタを使用して JMS メッセージ・サービス・プロバイダにアクセスするよう EJB 2.1 MDB を構成するには、`ejb-jar.xml` と `orion-ejb.jar.xml` という 2 つのデプロイ XML ファイルを使用する必要があります。`orion-ejb-jar.xml` ファイル構成を使用して、`ejb-jar.xml` の設定をオーバーライド、またリソース・アダプタの OC4J 固有の設定を追加します。たとえば、`ejb-jar.xml` で定義するコネクション・ファクトリおよび接続先名は、ローカル JNDI 環境には存在しない論理名である場合があります。デプロイ担当者は、`orion-ejb-jar.xml` ファイル内のこれらの設定をオーバーライドし、実際の名前にマッピングできます。論理名のマッピングの詳細は、19-15 ページの「JMS 宛先またはコネクション・リソース・マネージャのコネクション・ファクトリへの環境参照の構成（JMS 1.0）」を参照してください。

J2CA メッセージ・サービス・プロバイダを使用するように EJB 2.1 MDB を構成するには、次のようにします。

1. リソース・アダプタの名前を指定します。

   これを行うには、例 18-1 に示すように、`orion-ejb-jar.xml` ファイルの `<message-driven-deployment>` 要素の `resource-adapter` 属性を使用します。

2. 必要なアクティブ化構成プロパティを指定します。

   アクティブ化構成プロパティを指定するには、`orion-ejb-jar.xml` ファイルに含まれる `<message-driven-deployment>` 要素の `<config-property>` 要素と（例 18-1 を参照）、`ejb-jar.xml` ファイルに含まれる `<message-driven>` 要素の `<activation-config-property>` 要素（例 18-2 を参照）の任意の組合せを使用します。`orion-ejb-jar.xml` ファイルの構成は、`ejb-jar.xml` ファイルの構成をオーバーライドします。

   詳細は、次を参照してください。

   - 付録 B「J2CA アクティブ化構成プロパティ」
   - 2-31 ページの「メッセージ・サービス構成オプション : アノテーションまたは XML の選択と属性またはアクティブ化構成プロパティの選択」

例 18-1 に、orion-ejb-jar.xml ファイルを構成して、OracleASjms という名前の Oracle JMS リソース・アダプタを使用するようにこのメッセージドリブン Bean を構成する方法を示します。resource-adapter 属性を設定する必要があります。オプションで、1 つ以上の config-property 要素を使用してアクティブ化構成プロパティをオーバーライドまたは追加構成できます。

例 18-1 J2CA メッセージ・サービス・プロバイダの orion-ejb-jar.xml

```
<message-driven-deployment
    name="JCA_QueueMDB"
    resource-adapter="OracleASjms">
    ...
    <config-property>
        <config-property-name>DestinationName</config-property-name>
        <config-property-value>OracleASJMSRASubcontext/MyQ</config-property-value>
    </config-property>
    ...
</message-driven-deployment>
```

例 18-2 に、ejb-jar.xml を構成して、OracleASjms という名前の Oracle JMS リソース・アダプタを使用するようにメッセージドリブン Bean を構成する方法を示します。メッセージ・サービス・プロバイダの構成時に、コネクション・ファクトリ OracleASjms/MyQCF を oc4j-ra.xml ファイルに定義し、接続先名 OracleASjms/MyQueue を oc4j-connectors.xml に定義してあることを前提とします。2 フェーズ・コミット（2PC）をサポートする XA 対応ファクトリを定義するか、または 2PC サポートが必要でない場合は非 XA ファクトリを定義します。詳細は、第 23 章「メッセージ・サービスの構成」を参照してください。

例 18-2 J2CA メッセージ・サービス・プロバイダの ejb-jar.xml

```
<message-driven>
    <ejb-name>JCA_QueueMDB</ejb-name>
    <ejb-class>test.JCA_MDB</ejb-class>
    <messaging-type>javax.jms.MessageListener</messaging-type>
    <transaction-type>Container</transaction-type>

    <activation-config>
        <activation-config-property>
            <activation-config-property-name>
                DestinationType
            </activation-config-property-name>
            <activation-config-property-value>
                javax.jms.Queue
            </activation-config-property-value>
        </activation-config-property>
        <activation-config-property>
            <activation-config-property-name>
                DestinationName
            </activation-config-property-name>
            <activation-config-property-value>
                OracleASjms/MyQueue
            </activation-config-property-value>
        </activation-config-property>
        <activation-config-property>
            <activation-config-property-name>
                ConnectionFactoryJndiName
            </activation-config-property-name>
            <activation-config-property-value>
                OracleASjms/MyQCF
            </activation-config-property-value>
        </activation-config-property>
    </activation-config>
</message-driven>
```

表 A-3 にリストされているオプションの属性も設定できます。

使用する実際の名前は、メッセージ・サービス・プロバイダのインストール環境によって決まります。詳細は、23-2 ページの「J2CA メッセージ・サービス・プロバイダのコネクション・ファクトリ名」を参照してください。

直接メッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成

(J2CA リソース・アダプタを使用せずに) 直接メッセージ・サービス・プロバイダにアクセスするよう EJB 2.1 MDB を構成できます。

注意： メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、次を参照してください。

- 2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」
- 18-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」

これを行うには、デプロイ XML を使用します（18-4 ページの「デプロイ XML の使用方法」を参照）。

OC4J では、2 フェーズ・コミット（2PC）トランザクション用の XA ファクトリと、2PC を必要としないトランザクション用の非 XA ファクトリの両方がサポートされます。2PC サポートの詳細は、2-24 ページの「グローバル・トランザクションまたは 2 フェーズ・コミット（2PC）トランザクションへの参加方法」を参照してください。

デプロイ XML の使用方法

(J2CA リソース・アダプタを使用せずに) 直接 JMS メッセージ・サービス・プロバイダにアクセスするよう EJB 2.1 MDB を構成するには、`ejb-jar.xml` または `orion-ejb.jar.xml` のいずれかのデプロイ XML ファイルを使用します。`orion-ejb-jar.xml` ファイルの構成を使用して、`ejb-jar.xml` の設定をオーバーライドまたは OC4J 固有の設定を追加します。たとえば、`ejb-jar.xml` で定義するコネクション・ファクトリおよび接続先名は、ローカル JNDI 環境には存在しない論理名である場合があります。デプロイ担当者は、`orion-ejb-jar.xml` ファイル内のこれらの設定をオーバーライドし、実際の名前にマッピングできます。論理名のマッピングの詳細は、19-15 ページの「JMS 宛先またはコネクション・リソース・マネージャのコネクション・ファクトリへの環境参照の構成（JMS 1.0）」を参照してください。

構成を行うには、次のようにします。

1. 必要なアクティブ化構成プロパティを指定します。

   アクティブ化構成プロパティを指定するには、`orion-ejb-jar.xml` ファイルに含まれる `<message-driven-deployment>` 要素の `<config-property>` 要素と、`ejb-jar.xml` ファイルに含まれる `<message-driven>` 要素の `<activation-config-property>` 要素（例 18-3 を参照）の任意の組合せを使用します。`orion-ejb-jar.xml` ファイルの構成は、`ejb-jar.xml` ファイルの構成をオーバーライドします。

   詳細は、次を参照してください。

   - 付録 B「J2CA アクティブ化構成プロパティ」
   - 2-31 ページの「メッセージ・サービス構成オプション : アノテーションまたは XML の選択と属性またはアクティブ化構成プロパティの選択」

例 18-3 に、`ejb-jar.xml` を構成して、J2CA 以外の JMS メッセージ・サービス・プロバイダを使用するようにメッセージドリブン Bean を構成する方法を示します。メッセージ・サービス・プロバイダの構成時に、コネクション・ファクトリ `jms/MyQCF` およびキュー `jms/MyQueue` を定義してあることを前提とします。2 フェーズ・コミット（2PC）をサポート

する XA 対応ファクトリを定義するか、または 2PC サポートが必要でない場合は非 XA ファクトリを定義します。詳細は、第 23 章「メッセージ・サービスの構成」を参照してください。

例 18-3 J2CA 以外のメッセージ・サービス・プロバイダの ejb-jar.xml

```
<message-driven>
    <ejb-name>QueueMDB</ejb-name>
    <ejb-class>test.QueueMDB</ejb-class>
    <message-destination-type>javax.jms.Queue</message-destination-type>
    <transaction-type>Container</transaction-type>

    <activation-config>
        <activation-config-property>
            <activation-config-property-name>
                ConnectionFactoryJndiName
            </activation-config-property-name>
            <activation-config-property-value>
                jms/MyQCF
            </activation-config-property-value>
        </activation-config-property>
        <activation-config-property>
            <activation-config-property-name>
                DestinationName
            </activation-config-property-name>
            <activation-config-property-value>
                jms/MyQueue
            </activation-config-property-value>
        </activation-config-property>
    </activation-config>
</message-driven>
```

使用する実際の名前は、メッセージ・サービス・プロバイダのインストール環境によって決まります。詳細は、次を参照してください。

- 23-4 ページの「OEMS JMS 宛先名およびコネクション・ファクトリ名」
- 23-7 ページの「OEMS JMS データベース宛先名およびコネクション・ファクトリ名」

Windows オペレーティング・システムでの高速アンデプロイのための MDB の構成

MDB を使用する場合、MDB は着信メッセージを待機する受信状態にブロックされています。Windows 以外の環境では、MDB が待機状態のときに OC4J をシャットダウンした場合、OC4J は適時にシャットダウンします。

OEMS JMS データベース・プロバイダ（2-28 ページの「OEMS JMS データベース : アドバンスト・キューイング（AQ）ベース・プロバイダ」を参照）でメッセージドリブン Bean を使用し、OC4J が Windows 環境で稼働している場合、またはバックエンド・データベースが Windows 環境で稼働し、MDB が待機状態のときに OC4J をシャットダウンした場合は、OC4J インスタンスを停止できず、MDB を適時にアンデプロイできません。この場合、OC4J プロセスが 2.5 時間以上停止します。

`oracle.mdb.fastUndeploy` システム・プロパティ（18-6 ページの「システム・プロパティの使用方法」を参照）を使用している場合は、Windows 環境で MDB の動作を変更して、メッセージドリブン Bean をアンデプロイ可能にし、必要に応じて OC4J を適時にシャットダウンすることを可能にできます。

システム・プロパティの使用方法

`oracle.mdb.fastUndeploy` システム・プロパティには、MDB が着信メッセージの処理中でなく待機状態のときに、OC4J がデータベースをポーリングして（データベースへのラウンドトリップが必要）、セッションがシャットダウンされているかどうかを確認する頻度（正の整数の秒数）が設定されます。

パフォーマンスを最適化するために、妥当な値は 120 秒以上です。

このプロパティを 120（秒）に設定した場合、OC4J では 120 秒ごとにデータベースをポーリングします。

Oracle RAC フェイルオーバー用の MDB の構成

MDB アプリケーションが OEMS JMS データベースを Oracle RAC データベースとともに使用する場合は、次のようにデータベース・フェイルオーバー・シナリオを処理するようにアプリケーションを構成する必要があります。

- メッセージ・デキューに失敗した場合に再試行するようにメッセージドリブン Bean を構成します（18-7 ページの「デプロイ XML の使用方法」を参照）。
- 接続の取得に失敗した場合に再試行するように MDB クライアントを構成します（18-7 ページの「Java の使用方法」を参照）。

> **注意：** データソースの Oracle RAC 対応属性の詳細は、『Oracle Containers for J2EE サービス・ガイド』の「データソース」の章を参照してください。

デプロイ XML の使用方法

Oracle RAC フェイルオーバーをサポートするには、例 18-4 に示すように、orion-ejb-jar.xml ファイルの要素 message-driven-deployment の属性 dequeue-retry-count および dequeue-retry-interval を構成する必要があります。

dequeue-retry-count 属性は、障害が発生した場合にデータベース接続を再試行する回数をコンテナに指示します。デフォルトは 0 回です。

dequeue-retry-interval 属性は、次の再試行を行う前に Oracle RAC データベースのフェイルオーバーの完了を待つ時間の長さをコンテナに指示します。デフォルトは 60 秒です。

例 18-4 MDB での Oracle RAC フェイルオーバー用の orion-ejb-jar.xml

```
<message-driven-deployment name="MessageBeanTpc"
   connection-factory-location="java:comp/resource/cartojms1/TopicConnectionFactories/aqTcf"
   destination-location="java:comp/resource/cartojms1/Topics/topic1"
   subscription-name="MDBSUB"
   dequeue-retry-count=3
   dequeue-retry-interval=90/>
...
```

Java の使用方法

Oracle RAC フェイルオーバーをサポートするには、接続の取得に失敗した場合に再試行するように、Oracle RAC データベースに対して実行しているスタンドアロン OEMS JMS データベース・クライアントを構成する必要があります。

接続オブジェクトが無効かどうかを判断するには、com.evermind.sql.DbUtil のメソッド oracleFatalError を使用することをお薦めします（例 18-5 を参照）。その場合、必要に応じてデータベース接続を再度確立します。

例 18-5 接続取得に失敗した後に再試行するクライアント

```
import com.evermind.sql.DbUtil;
...
getMessage(QueueSesssion session) {
    try {
        QueueReceiver rcvr = session.createReceiver(rcvrQueue);
        Message msgRec = rcvr.receive();
    }
    catch(Exception e ) {
        if (exc instanceof JMSException) {
            JMSException  jmsexc = (JMSException) exc;
            sql_ex = (SQLException)(jmsexc.getLinkedException());
            db_conn = oracle.jms.AQjmsSession)session.getDBConnection();
            if ((DbUtil.oracleFatalError(sql_ex, db_conn)) {
               // failover logic
            }
        }
    }
}
```

パラレル・メッセージ処理の構成

デフォルトでは、OC4J はメッセージ・ロケーションのメッセージをポーリングするために 1 つの受信スレッドを使用します。

2 つ以上の受信スレッドを使用すると、メッセージをパラレルに受信できるため、パフォーマンスが向上する可能性があります。

メッセージ・ロケーションがトピックの場合、受信スレッドの数は 1 に固定されます。

メッセージ・ロケーションがキューの場合、受信スレッドの数を構成できます（18-8 ページの「デプロイ XML の使用方法」を参照）。

MDB プールの Bean インスタンスの最小数は、受信スレッドがメッセージ処理のためにプールから Bean インスタンスを取得できるように、少なくとも受信スレッドの数と同じに設定する必要があります。

詳細は、次を参照してください。

- 2-31 ページの「メッセージ・サービス構成オプション : アノテーションまたは XML の選択と属性またはアクティブ化構成プロパティの選択」
- 31-5 ページの「Bean インスタンスのプール・サイズの構成」

デプロイ XML の使用方法

パラレル・メッセージ処理は、`orion-ejb-jar.xml` ファイルで構成します。このオプションの構成方法は、使用するメッセージ・サービス・プロバイダのタイプによって決まります。

- J2CA アダプタ・メッセージ・サービス・プロバイダ
- J2CA 以外のアダプタ・メッセージ・サービス・プロバイダ

どちらの場合も、OC4J を再起動して変更を適用する必要があります。

J2CA アダプタ・メッセージ・サービス・プロバイダ

J2CA アダプタ・メッセージ・サービス・プロバイダを使用している場合は、`<config-property>` 要素を使用して、`ReceiverThreads` 構成プロパティを設定します。

たとえば、J2CA アダプタ・メッセージ・サービス・プロバイダを使用しており、3 つのメッセージドリブン Bean インスタンスでメッセージ・ロケーションからパラレルに受信する場合、次のように `ReceiverThreads` 構成プロパティを `3` に設定します。

```
<message-driven-deployment ... >
...
    <config-property>
        <config-property-name>RecieverThreads</config-property-name>
        <config-property-value>3</config-property-value>
    </config-property>
...
</message-driven-deployment>
```

`ReceiverThreads` の詳細は、表 B-2 を参照してください。

J2CA 以外のアダプタ・メッセージ・サービス・プロバイダ

OEMS JMS や OEMS JMS データベースなどの J2CA 以外のアダプタ・メッセージ・サービス・プロバイダを使用している場合は、`<message-driven-deployment>` 要素の `listener-threads` 属性を使用します。

たとえば、OEMS JMS または OEMS JMS データベースを使用しており、3 つのメッセージドリブン Bean インスタンスでメッセージ・ロケーションからパラレルに受信する場合、次のように `listener-threads` 属性を `3` に設定します。

```
<message-driven-deployment ...  listener-threads="3"
  ...
</message-driven-deployment>
```

listener-threads の詳細は、表 A-3 を参照してください。

注意： メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、次を参照してください。

- 2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」
- 18-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」

最大配信数の構成

メッセージドリブン Bean の onMessage メソッドがエラーを返した場合、つまり、確認応答操作の起動に失敗した場合、例外をスローした場合、またはその両方の場合（18-9 ページの「デプロイ XML の使用方法」を参照）に、OC4J がこのメソッドにメッセージの即時再配信を試行する最大回数を構成できます。

この回数の再配信が行われた後で、メッセージは配信不能とみなされ、メッセージ・サービス・プロバイダのポリシーに従って処理されます。たとえば、OEMS JMS はその例外キュー（jms/Oc4jJmsExceptionQueue）にメッセージを挿入します。

デプロイ XML の使用方法

orion-ejb-jar.xml ファイルで、最大配信数を設定します。この値の構成方法は、使用しているメッセージドリブン・プロバイダのタイプによって決まります。

- J2CA アダプタ・メッセージ・サービス・プロバイダ
- J2CA 以外のアダプタ・メッセージ・サービス・プロバイダ

J2CA アダプタ・メッセージ・サービス・プロバイダ

J2CA アダプタ・メッセージ・サービス・プロバイダを使用している場合は、<config-property> 要素を使用して、MaxDeliveryCnt 構成プロパティを設定します。

たとえば、J2CA アダプタ・メッセージ・サービス・プロバイダを使用していて、最大配信数を 3 に設定する場合は、次のようにします。

```
<message-driven-deployment ... >
...
    <config-property>
        <config-property-name>MaxDeliveryCnt</config-property-name>
        <config-property-value>3</config-property-value>
    </config-property>
...
</message-driven-deployment>
```

MaxDeliveryCnt の詳細は、表 B-2 を参照してください。

J2CA 以外のアダプタ・メッセージ・サービス・プロバイダ

OEMS JMS や OEMS JMS データベースなどの J2CA 以外のアダプタ・メッセージ・サービス・プロバイダを使用している場合は、<message-driven-deployment> 要素の max-delivery-count 属性を使用します。

たとえば、OEMS JMS または OEMS JMS データベースを使用していて、最大配信数を 3 に設定する場合は、次のようにします。

```
<message-driven-deployment ...  max-delivery-count="3"
  ...
</message-driven-deployment>
```

max-delivery-count の詳細は、表 A-3 を参照してください。

注意： メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、次を参照してください。

- 2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」
- 18-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」

EJB 2.1 MDB の接続障害リカバリの構成

ネットワークや JMS サーバーの停止などのイベントを原因とする接続障害に対し、メッセージドリブン Bean のリスナー・スレッドでどのように応答するかを構成できます。

これらのオプションは、メッセージドリブン Bean のコンテナ管理のトランザクションにのみ適用されます。

接続障害リカバリ・オプションは、`orion-ejb-jar.xml` ファイル（18-10 ページの「デプロイ XML の使用方法」を参照）を使用して構成できます。

フェイルオーバーの詳細は、2-35 ページの「OC4J EJB アプリケーション・クラスタリング・サービスについて」を参照してください。

デプロイ XML の使用方法

`orion-ejb-jar.xml` ファイルで、デキュー再試行カウントおよび間隔を設定します。この値の構成方法は、使用しているメッセージドリブン・プロバイダのタイプによって決まります。

- J2CA アダプタ・メッセージ・サービス・プロバイダ
- J2CA 以外のアダプタ・メッセージ・サービス・プロバイダ

どちらの場合も、OC4J を再起動して変更を適用する必要があります。

J2CA アダプタ・メッセージ・サービス・プロバイダ

J2CA リソース・アダプタを使用してメッセージ・サービス・プロバイダにアクセスする場合、Oracle JMS コネクタは、JMS リソースのポーリングを無制限に再試行します。この再試行間隔は、例 18-6 に示すようにアクティブ化構成プロパティ `EndpointFailureRetryInterval` で構成できます。

再試行後のメッセージのリカバリでは、メッセージの順序は保証されません。また、JMS トピックに対する MDB サブスクリプションが非永続的な場合、メッセージは失われるか、重複する可能性があります。

詳細は、表 B-2 の `EndpointFailureRetryInterval` を参照してください。

例 18-6 orion-ejb-jar.xml での EndpointFailureRetryInterval の構成

```
<message-driven-deployment ... >
...
    <config-property>
        <config-property-name>EndpointFailureRetryInterval</config-property-name>
        <config-property-value>20000</config-property-value>
    </config-property>
...
</message-driven-deployment>
```

J2CA 以外のアダプタ・メッセージ・サービス・プロバイダ

OEMS JMS や OEMS JMS データベースなどの J2CA 以外のアダプタ・メッセージ・サービス・プロバイダを使用している場合は、<message-driven-deployment> 要素の dequeue-retry-count および dequeue-retry-interval 属性を使用します。デフォルトのデキュー再試行カウントは 0 で、デフォルトのデキュー再試行間隔は 60 秒です。

たとえば、OEMS JMS または OEMS JMS データベースを使用していて、デキュー再試行カウントを 3 に設定し、デキュー再試行間隔を 90 秒に設定する場合は、次のようにします。

```
<message-driven-deployment ... dequeue-retry-count="3" dequeue-retry-interval="90"
  ...
</message-driven-deployment>
```

dequeue-retry-count および dequeue-retry-interval の詳細は、表 A-3 を参照してください。

注意： メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、次を参照してください。

- 2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」
- 18-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」

EJB 2.1 MDB のライフ・サイクル・コールバック・メソッドの構成

次に、javax.ejb.MessageDrivenBean インタフェースでの指定に従って、メッセージドリブン Bean が実装する必要のある EJB 2.1 ライフ・サイクル・メソッドを示します（18-11 ページの「Java の使用方法」を参照）。

- ejbCreate
- ejbRemove

注意： EJB 2.1 を使用する場合は、すべてのメッセージドリブン Bean コールバック・メソッドを実装する必要があります。何もアクションを行う必要がない場合は、空のメソッドを実装します。

詳細は、1-60 ページの「メッセージドリブン Bean のライフ・サイクル」を参照してください。

Java の使用方法

例 18-7 では、EJB 2.1 メッセージドリブン Bean のライフ・サイクル・コールバック・メソッドの実装方法を説明します。

例 18-7 EJB 2.1 MDB のライフ・サイクル・コールバック・メソッドの実装

```
public void ejbRemove() {
    // when bean is removed
}
```

第 VIII 部

OC4J EJB サービスの構成

第 VIII 部では、EJB 3.0 および EJB 2.1 Enterprise JavaBean の OC4J EJB サービスの構成の手順に関する情報を示します。概念的な情報は、第 I 部「EJB の概要」を参照してください。

第 VIII 部は次の各章で構成されています。

- 第 19 章「JNDI サービスの構成」
- 第 20 章「データソースの構成」
- 第 21 章「トランザクション・サービスの構成」
- 第 22 章「セキュリティ・サービスの構成」
- 第 23 章「メッセージ・サービスの構成」
- 第 24 章「OC4J EJB アプリケーション・クラスタリング・サービスの構成」
- 第 25 章「タイマー・サービスの構成」

19

JNDI サービスの構成

この章の内容は次のとおりです。

詳細は、次を参照してください。

- 2-17 ページの「EJB JNDI サービスについて」
- 第 29 章「クライアントからの Enterprise Bean へのアクセス」
- 『Oracle Containers for J2EE サービス・ガイド』の「Oracle JNDI」

環境参照の構成

実行時に JNDI を使用して EJB から重要なリソースにアクセスする前に、それらのリソースへの環境参照を定義する必要があります。環境参照は静的で、Bean によって変更できません。

この項では次の参照の構成について説明します。

- EJB 環境参照
- リソース・マネージャのコネクション・ファクトリ環境参照
- 環境変数の環境参照
- Web サービス環境参照
- 永続性コンテキスト参照

EJB 3.0 では、環境参照を定義するかわりにアノテーション、リソース・インジェクションおよびデフォルトの JNDI 名（クラス名およびインタフェース名に基づく）を使用できます。別の方法として、OC4J 固有のデプロイメント・ディスクリプタまたは OC4J 固有のアノテーションを使用して環境参照を定義できます。

EJB 2.1 では、`<ejb-ref>` または `<ejb-local-ref>` 要素を適切なデプロイメント・ディスクリプタに定義する必要があります。

どちらの場合も、環境参照を定義する場合は、実際の JNDI 名を使用するか、関連付けられている論理名を使用してデプロイの柔軟性を高めることができます。

詳細は、次を参照してください。

- 19-4 ページの「EJB 環境参照を構成する場所」
- 19-4 ページの「論理名を使用する必要があるかどうか」

EJB 環境参照

クライアントのロールで動作している 1 つの Enterprise Bean（ソース Enterprise Bean と呼ぶ）が別の Enterprise Bean（ターゲット Enterprise Bean と呼ぶ）にアクセスするには、その前にソース Enterprise Bean のデプロイメント・ディスクリプタでターゲット Enterprise Bean への EJB 参照を定義する必要があります。

注意： EJB 3.0 では、ターゲット Enterprise Bean への環境参照は不要です。リソース・インジェクションを使用して、ターゲット Enterprise Bean に直接アクセスできます（29-5 ページの「EJB 3.0 Enterprise Bean へのアクセス」を参照）。

詳細は、次を参照してください。

- リモート EJB への環境参照の構成 : クラスタ化または結合された Web 層および EJB 層
- リモート EJB への環境参照の構成 : クラスタ化されていない個別の Web 層および EJB 層
- ローカル EJB への環境参照の構成

リソース・マネージャのコネクション・ファクトリ環境参照

JDBC データソース、JMS トピックまたはキュー、Java mail、HTTP URL などのサービスへの接続を提供するリソース・マネージャのコネクション・ファクトリへの環境参照を定義できます。これらの参照は、提供する実際のリソース・マネージャ・コネクション・ファクトリへのデプロイ時に、その OC4J がバインドする論理名です。

注意： EJB 3.0 では、リソース・マネージャのコネクション・ファクトリへの環境参照は不要です。リソース・インジェクションを使用して、リソース・マネージャのコネクション・ファクトリに直接アクセスできます（19-24 ページの「EJB 3.0 リソース・マネージャのコネクション・ファクトリのルックアップ」を参照）。

リソース・マネージャのコネクション・ファクトリにアクセスする各クライアントについて、クライアント・ソース・コードにこれを注入するか、クライアントのデプロイメント・ディスクリプタに環境参照を定義する必要があります。

詳細は、次を参照してください。

- 19-12 ページの「JDBC データソース・リソース・マネージャのコネクション・ファクトリへの環境参照の構成」
- 「JMS 宛先リソース・マネージャのコネクション・ファクトリへの環境参照の構成（JMS 1.1）」
- 19-15 ページの「JMS 宛先またはコネクション・リソース・マネージャのコネクション・ファクトリへの環境参照の構成（JMS 1.0）」

環境変数の環境参照

環境変数を環境参照とともに定義して、JNDI を使用して環境変数値をアクセス可能にできます。

詳細は、19-17 ページの「環境変数への環境参照の構成」を参照してください。

Web サービス環境参照

Web サービスを環境参照とともに定義して、JNDI を使用して Web サービスをアクセス可能にできます。

詳細は、19-18 ページの「Web サービスへの環境参照の構成」を参照してください。

永続性コンテキスト参照

エンティティ・マネージャへの推奨されるアクセス方法は、アノテーションおよび依存性注入を使用することです（29-10 ページの「OC4J のデフォルト・エンティティ・マネージャの取得」および 29-11 ページの「JNDI を使用したエンティティ・マネージャの取得」を参照）。

アノテーションおよびインジェクションをサポートしないクラス（ヘルパー・クラスおよび Web クライアント）でエンティティ・マネージャを取得するには、最初に永続性コンテキスト参照を定義してから、JNDI を使用してエンティティ・マネージャをルックアップする必要があります。

詳細は、次を参照してください。

- 19-19 ページの「永続性コンテキストへの環境参照の構成」
- 29-12 ページの「ヘルパー・クラスでのエンティティ・マネージャの取得」

EJB 環境参照を構成する場所

環境参照を使用することを選択した場合、EJB 参照を構成する場所は、表 19-1 に示すようにクライアントのタイプによって決まります。

表 19-1 クライアント・タイプ別のデプロイメント・ディスクリプタ

クライアント・タイプ	説明	デプロイメント・ディスクリプタ	OC4J 固有のデプロイメント・ディスクリプタ
EJB	コンテナ内から Enterprise Bean を起動する別の Enterprise Bean	`ejb-jar.xml`	`orion-ejb-jar.xml`
スタンドアロン・クライアント	コンテナの外部から Enterprise Bean を起動する Pure Java クライアント	`application-client.xml`	`orion-application-client.xml`
サーブレットまたは JSP	コンテナの外部から Enterprise Bean を起動するサーブレットまたは JSP	`web.xml`	`orion-web.xml`

EJB 3.0 で EJB 環境参照を定義する場合、OC4J 固有のデプロイメント・ディスクリプタのかわりに OC4J 固有のアノテーションを使用できます。

論理名を使用する必要があるかどうか

環境参照を定義する場合は、リソースを論理名またはその JNDI 名で識別できます。アプリケーション・アセンブリおよびデプロイの柔軟性を最大にするために、通常はアプリケーション環境で定義する論理名でリソースを参照することにより EJB アプリケーションを開発します。このように間接的に設定することにより、Bean 開発者は、アプリケーションのアセンブルおよびデプロイの方法によって変化する可能性のある実際の名前を指定せずに、Enterprise Bean、他のリソース（JDBC `DataSource` など）および環境変数を参照できます。この章の手順では、論理名または JNDI 名の構成方法を説明します。

リモート EJB への環境参照の構成 : クラスタ化または結合された Web 層および EJB 層

同じ OC4J インスタンス上に Web 層と EJB 層の両方を含むクラスタ化された OC4J アーキテクチャまたは単一インスタンスの OC4J アーキテクチャでは、次のいずれかの方法を使用してターゲット Enterprise Bean のリモート・インスタンスへの EJB 参照を定義できます（次の方法は、後の方がアセンブリおよびデプロイの柔軟性が高くなるように配列されています）。

- ターゲット Bean の実際の名前を指定する適切なクライアント EJB デプロイメント・ディスクリプタで `<ejb-ref>` 要素を構成します（19-5 ページの「クライアントの ejb-ref の構成 : インダイレクションなし」を参照）。
- 論理名とその論理名を実際の Bean と関連付ける `<ejb-link>` 要素を指定する適切なクライアント EJB デプロイメント・ディスクリプタで `<ejb-ref>` 要素を構成します（19-5 ページの「クライアントの ejb-ref の構成 : ejb-link を使用したインダイレクションの解決」を参照）。
- 論理名を指定する適切なクライアント EJB デプロイメント・ディスクリプタで `<ejb-ref>` 要素を構成し、その論理名を実際の Bean と関連付ける OC4J 固有の適切なデプロイメント・ディスクリプタで `<ejb-ref-mapping>` を構成します（19-6 ページの「クライアントの ejb-ref の構成 : orion-ejb-jar.xml の ejb-ref-mapping を使用したインダイレクションの解決」を参照）。

注意 : EJB 3.0 では、ターゲット Enterprise Bean への環境参照は不要です。リソース・インジェクションを使用して、ターゲット Enterprise Bean に直接アクセスできます（29-5 ページの「EJB 3.0 Enterprise Bean へのアクセス」を参照）。

Web 層と EJB 層が異なるホスト上の個別の OC4J インスタンスにデプロイされる非クラスタ化アーキテクチャの場合は、19-7 ページの「リモート EJB への環境参照の構成 : クラスタ化されていない個別の Web 層および EJB 層」を参照してください。

ターゲット Enterprise Bean のルックアップの詳細は、第 29 章「クライアントからの Enterprise Bean へのアクセス」を参照してください。

クライアントの ejb-ref の構成 : インダイレクションなし

Bean インタフェースが 1 つのみの場合（ただ 1 つのセッション Bean のみがインタフェース `Cart.class` を使用する場合など）、またはアセンブリおよびデプロイの柔軟性を提供するインダイレクションを使用しない場合、このオプションを選択します。

1. 適切なクライアント・デプロイメント・ディスクリプタの `<ejb-ref>` 要素を定義し（19-4 ページの「EJB 環境参照を構成する場所」を参照）、例 19-1 に示すように次のサブ要素を構成します。
   - `<ejb-ref-name>`: ターゲット Enterprise Bean の実際の名前
   - `<ejb-ref-type>`: ターゲット Enterprise Bean のタイプ（`Session` または `Entity` のいずれか）
   - `<home>`: ターゲット Enterprise Bean のリモート・ホーム・インタフェースのパッケージ名とクラス名
   - `<remote>`: ターゲット Enterprise Bean のリモート・コンポーネント・インタフェースのパッケージ名とクラス名

例 19-1 ejb-ref-name の構成

```
<ejb-ref>
 <ejb-ref-name>myBeans/BeanA</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>myBeans.BeanAHome</home>
 <remote>myBeans.BeanA</remote>
</ejb-ref>
```

クライアントの ejb-ref の構成 : ejb-link を使用したインダイレクションの解決

Bean インタフェースが 1 つのみではない場合、またはアセンブリおよびデプロイの柔軟性を提供するインダイレクションを使用する場合、このオプションを選択します。

1. 適切なクライアント・デプロイメント・ディスクリプタの `<ejb-ref>` 要素を定義し（19-4 ページの「EJB 環境参照を構成する場所」を参照）、例 19-2 に示すように次のサブ要素を構成します。
   - `<ejb-ref-name>`: ターゲット Enterprise Bean の論理名
   - `<ejb-ref-type>`: ターゲット Enterprise Bean のタイプ（`Session` または `Entity` のいずれか）
   - `<home>`: ターゲット Enterprise Bean のリモート・ホーム・インタフェースのパッケージ名とクラス名
   - `<remote>`: ターゲット Enterprise Bean のリモート・コンポーネント・インタフェースのパッケージ名とクラス名
   - `<ejb-link>`: ターゲット Bean の実際の名前

例 19-2 ejb-link により解決される論理名での ejb-ref-name の構成

```
<ejb-ref>
 <ejb-ref-name>ejb/nextVal</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>myBeans.BeanAHome</home>
 <remote>myBeans.BeanA</remote>
 <ejb-link>myBeans/BeanA</ejb-link>
</ejb-ref>
```

クライアントの ejb-ref の構成 : orion-ejb-jar.xml の ejb-ref-mapping を使用したインダイレクションの解決

次の条件が満たされる場合にこのオプションを使用します。

- Bean インタフェースが 1 つのみではない場合
- アセンブリおよびデプロイに最大の柔軟性を提供するインダイレクションを使用する場合

1. 適切なクライアント・デプロイメント・ディスクリプタの `<ejb-ref>` 要素を定義し（19-4 ページの「EJB 環境参照を構成する場所」を参照）、例 19-3 に示すように次のサブ要素を構成します。
   - `<ejb-ref-name>`: ターゲット Enterprise Bean の論理名
   - `<ejb-ref-type>`: ターゲット Enterprise Bean のタイプ（`Session` または `Entity` のいずれか）
   - `<home>`: ターゲット Enterprise Bean のリモート・ホーム・インタフェースのパッケージ名とクラス名
   - `<remote>`: ターゲット Enterprise Bean のリモート・コンポーネント・インタフェースのパッケージ名とクラス名

例 19-3 ejb-ref-mapping により解決される論理名での ejb-ref-name の構成

```
<ejb-ref>
 <ejb-ref-name>ejb/nextVal</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>myBeans.BeanAHome</home>
 <remote>myBeans.BeanA</remote>
</ejb-ref>
```

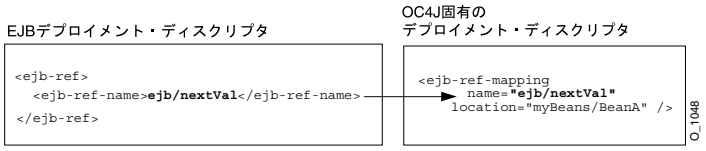
2. `orion-ejb-jar.xml` デプロイメント・ディスクリプタ内で、例 19-4 に示すように論理名をターゲット Bean の実際の名前にマッピングする `<ejb-ref-mapping>` 要素を定義します。

例 19-4 ejb-ref-mapping での実際の名前への論理名のマッピング

```
<ejb-ref-mapping name="ejb/nextVal" location="myBeans/BeanA"/>
```

図 19-1 に示すように、<ejb-ref-mapping> 要素では、name 属性を <ejb-ref-name> と一致するように構成し、location 属性をターゲット Bean の実際の名前で構成します。例 19-4 では、論理名 ejb/nextVal は、ターゲット Bean myBeans/BeanA の実際の名前にマッピングされます。

図 19-1 ejb-ref-name と ejb-ref-mapping の関連付け



OC4J は、論理名を、クライアント・サイドの実際の JNDI 名にマッピングします。サーバー・サイドで JNDI 名を受信し、これを JNDI ツリー内で解決します。

リモート EJB への環境参照の構成 : クラスタ化されていない個別の Web 層および EJB 層

一般的な Java EE アプリケーション・アーキテクチャでは、非クラスタ化環境で Web 層を 1 つの OC4J インスタンスにデプロイし、EJB 層を別のホスト上にあるもう 1 つの OC4J インスタンスにデプロイします。

このアーキテクチャでリモート Enterprise Bean にアクセスするには、コンテキストの作成時に Web 層コードの必須 JNDI プロパティを設定する必要があります（19-24 ページの「初期コンテキストでの JNDI プロパティの設定」を参照）。これらのハードコードされたプロパティにより、テスト環境から本番環境への移行などの際に移植性の問題が発生する可能性があります。

OC4J 固有のデプロイ XML（19-8 ページの「デプロイ XML の使用方法」を参照）を使用すると、リモート Enterprise Bean への参照を、必須の JNDI コンテキスト変数を含む JNDI プロパティ・ファイルと関連付けることができます。これにより、アセンブリとデプロイが容易になります。

図 19-2 は JSP またはサーブレット・クライアントのアーキテクチャを示しており、図 19-3 は EJB クライアントのアーキテクチャを示しています。

図 19-2 Web 層および EJB 層のリモート EJB アクセス : JSP またはサーブレット・クライアント

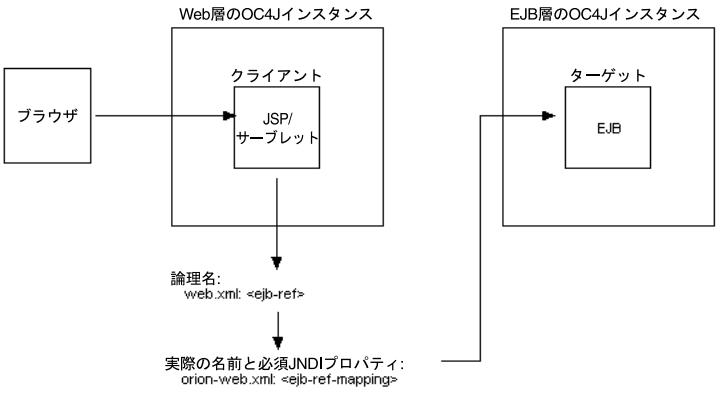
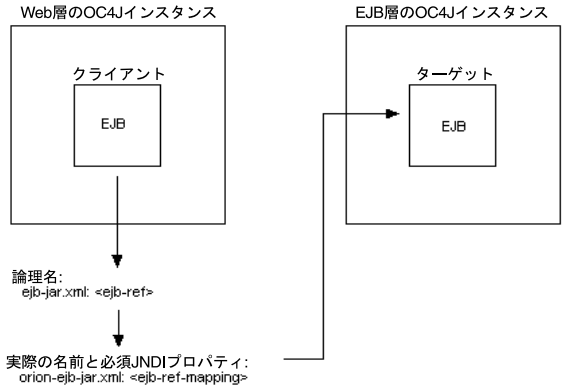


図 19-3 Web 層および EJB 層のリモート EJB アクセス : EJB クライアント



JNDI プロパティ・ファイルの詳細は、19-23 ページの「JNDI プロパティ・ファイルでの JNDI プロパティの設定」を参照してください。

デプロイ XML の使用方法

OC4J 固有の要素 `<ejb-ref-mapping>` を使用して、リモート Enterprise Bean への参照を必須の JNDI コンテキスト変数を含む JNDI プロパティ・ファイルと関連付けるには、Web 層の OC4J インスタンスを次のように構成します。

1. 適切なクライアント・デプロイメント・ディスクリプタの `<ejb-ref>` 要素を定義し（19-4 ページの「EJB 環境参照を構成する場所」を参照）、例 19-5 に示すように次のサブ要素を構成します。
   - `<ejb-ref-name>`: ターゲット Enterprise Bean の論理名
   - `<ejb-ref-type>`: ターゲット Enterprise Bean のタイプ（`Session` または `Entity` のいずれか）
   - `<home>`: ターゲット Enterprise Bean のリモート・ホーム・インタフェースのパッケージ名とクラス名
   - `<remote>`: ターゲット Enterprise Bean のリモート・コンポーネント・インタフェースのパッケージ名とクラス名

例 19-5 ejb-ref-mapping により解決される論理名での ejb-ref-name の構成

```
<ejb-ref>
 <ejb-ref-name>ejb/emp</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>myBeans.EmployeeBeanHome</home>
 <remote>myBeans.EmployeeBean</remote>
</ejb-ref>
```

このアーキテクチャにおいて、クライアント・デプロイメント・ディスクリプタは Web 層の OC4J インスタンス上にあります。リモート Enterprise Bean のクライアントは、次のいずれかです。

- Web 層にデプロイされた JSP またはサーブレット : `web.xml` ファイルを使用します。
- Web 層にデプロイされた Enterprise Bean: `ejb-jar.xml` ファイルを使用します。

2. クライアントのタイプに応じ、orion-web.xml または orion-ejb-jar.xml デプロイメント・ディスクリプタ内で、例 19-6 に示すように次の操作を実行する <ejb-ref-mapping> 要素を定義します。

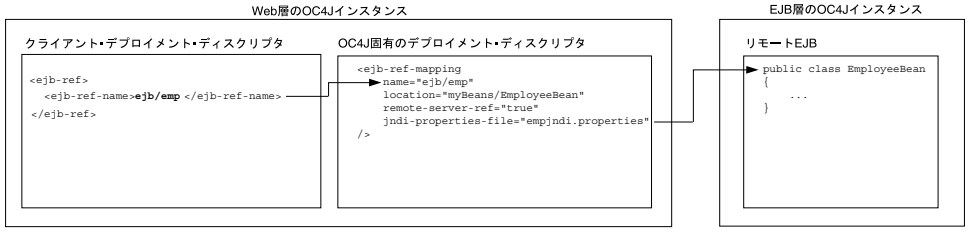
   - ターゲット Bean の実際の名前（myBeans/EmployeeBean）に論理名（ejb/emp）をマッピングします。
   - ターゲット EJB インスタンスがリモート・ホストに存在することを指定します（remote-server-ref="true"）。
   - 参照を JNDI プロパティ・ファイルと関連付けます（jndi-properties-file="empjndi.properties"）。このプロパティ・ファイルには、ターゲット EJB インスタンスがデプロイされているリモート・ホストへのアクセス時にクライアントが必要とする JNDI コンテキスト変数が含まれます。

例 19-6 リモート・ターゲット EJB に対応する ejb-ref-mapping での実際の名前への論理名のマッピング

```
<ejb-ref-mapping
    name="ejb/emp"
    location="myBeans/EmployeeBean"
    remote-server-ref="true"
    jndi-properties-file="empjndi.properties"
/>
```

図 19-1 に示すように、<ejb-ref-mapping> 要素では、name 属性を <ejb-ref-name> と一致するように構成し、location 属性をターゲット Bean の実際の名前で構成します。例 19-4 では、論理名 ejb/emp は、ターゲット Bean myBeans/EmployeeBean の実際の名前にマッピングされます。

図 19-4 リモート・ターゲット EJB に対応する ejb-ref-name と ejb-ref-mapping の関連付け



Web 層のクライアント（Web 層にデプロイされた JSP、サーブレットまたは Enterprise Bean）が（インジェクションまたは JNDI ルックアップを使用して）リモート・ターゲット Enterprise Bean にアクセスする場合、Web 層の OC4J インスタンスは、（Web 層の OC4J インスタンスの web.xml または ejb-jar.xml ファイルで指定された）論理名を（Web 層の OC4J インスタンスの orion-web.xml または orion-ejb-jar.xml ファイルで指定された）実際の名前にマッピングします。Web 層の OC4J インスタンスは、EJB 層の OC4J インスタンスにアクセスするために <ejb-ref-mapping> 要素で指定された JNDI プロパティ・ファイルを使用し、実際の名前を EJB 層の OC4J インスタンス上のターゲット Enterprise Bean に解決します。

ローカル EJB への環境参照の構成

次のいずれかの方法を使用してターゲット Enterprise Bean のローカル・インタフェースへの EJB 参照を定義できます（次の方法は、後の方がアセンブリおよびデプロイの柔軟性が高くなるように配列されています）。

- ターゲット Bean の実際の名前を指定する適切なクライアント EJB デプロイメント・ディスクリプタで <ejb-local-ref> 要素を構成します（19-10 ページの「クライアントの ejb-local-ref の構成 : インダイレクションなし」を参照）。
- 論理名とその論理名を実際の Bean と関連付ける <ejb-link> 要素を指定する適切なクライアント EJB デプロイメント・ディスクリプタで <ejb-local-ref> 要素を構成します（19-11 ページの「クライアントの ejb-local-ref の構成 : ejb-link を使用したインダイレクションの解決」を参照）。
- 論理名を指定する適切なクライアント EJB デプロイメント・ディスクリプタで <ejb-local-ref> 要素を構成し、その論理名を実際の Bean と関連付ける OC4J 固有の適切なデプロイメント・ディスクリプタで <ejb-ref-mapping> を構成します（19-11 ページの「クライアントの ejb-local-ref の構成 : orion-ejb-jar.xml の ejb-ref-mapping を使用したインダイレクションの解決」を参照）。

> **注意 :** EJB 3.0 では、ターゲット Enterprise Bean への環境参照は不要です。リソース・インジェクションを使用して、ターゲット Enterprise Bean に直接アクセスできます（29-5 ページの「EJB 3.0 Enterprise Bean へのアクセス」を参照）。

ターゲット Enterprise Bean のルックアップの詳細は、第 29 章「クライアントからの Enterprise Bean へのアクセス」を参照してください。

クライアントの ejb-local-ref の構成 : インダイレクションなし

Bean インタフェースが 1 つのみの場合（ただ 1 つのセッション Bean のみがインタフェース Cart.class を使用する場合など）、またはアセンブリおよびデプロイの柔軟性を提供するインダイレクションを使用しない場合、このオプションを選択します。

1. 適切なクライアント・デプロイメント・ディスクリプタの <ejb-local-ref> 要素を定義し（19-4 ページの「EJB 環境参照を構成する場所」を参照）、例 19-1 に示すように次のサブ要素を構成します。
   - <ejb-ref-name>: ターゲット Enterprise Bean の実際の名前
   - <ejb-ref-type>: ターゲット Bean のタイプ（Session または Entity）
   - <local-home>: ターゲット Bean のローカル・ホーム・インタフェースのパッケージ名とクラス名
   - <local>: ターゲット Bean のローカル・コンポーネント・インタフェースのパッケージ名とクラス名

例 19-7 ejb-local-ref-name の構成

```
<ejb-local-ref>
 <ejb-ref-name>myBeans/BeanA</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <local-home>myBeans.BeanAHome</home>
 <local>myBeans.BeanA</remote>
</ejb-ref>
```

クライアントの ejb-local-ref の構成 : ejb-link を使用したインダイレクションの解決

Bean インタフェースが 1 つのみではない場合、またはアセンブリおよびデプロイの柔軟性を提供するインダイレクションを使用する場合、このオプションを選択します。

1. 適切なクライアント・デプロイメント・ディスクリプタの <ejb-local-ref> 要素を定義し（19-4 ページの「EJB 環境参照を構成する場所」を参照）、例 19-8 に示すように次のサブ要素を構成します。
   - <ejb-ref-name>: ターゲット Enterprise Bean の論理名
   - <ejb-ref-type>: ターゲット Bean のタイプ（Session または Entity）
   - <local-home>: ターゲット Bean のローカル・ホーム・インタフェースのパッケージ名とクラス名
   - <local>: ターゲット Bean のローカル・コンポーネント・インタフェースのパッケージ名とクラス名
   - <ejb-link>: ターゲット Bean の実際の名前

例 19-8 ejb-link により解決される論理名での ejb-ref-name の構成

```
<ejb-local-ref>
 <ejb-ref-name>ejb/nextVal</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <local-home>myBeans.BeanAHome</home>
 <local>myBeans.BeanA</remote>
 <ejb-link>myBeans/BeanA</ejb-link>
</ejb-ref>
```

クライアントの ejb-local-ref の構成 : orion-ejb-jar.xml の ejb-ref-mapping を使用したインダイレクションの解決

次の条件が満たされる場合にこのオプションを使用します。

- Bean インタフェースが 1 つのみではない場合
- アセンブリおよびデプロイに最大の柔軟性を提供するインダイレクションを使用する場合

1. 適切なクライアント・デプロイメント・ディスクリプタの <ejb-ref> 要素を定義し（19-4 ページの「EJB 環境参照を構成する場所」を参照）、例 19-9 に示すように次のサブ要素を構成します。
   - <ejb-ref-name>: ターゲット Enterprise Bean の論理名
   - <ejb-ref-type>: ターゲット Bean のタイプ（Session または Entity）
   - <local-home>: ターゲット Bean のローカル・ホーム・インタフェースのパッケージ名とクラス名
   - <local>: ターゲット Bean のローカル・コンポーネント・インタフェースのパッケージ名とクラス名

例 19-9 ejb-ref-mapping により解決される論理名での ejb-ref-name の構成

```
<ejb-local-ref>
 <ejb-ref-name>ejb/nextVal</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <local-home>myBeans.BeanAHome</home>
 <local>myBeans.BeanA</remote>
</ejb-ref>
```

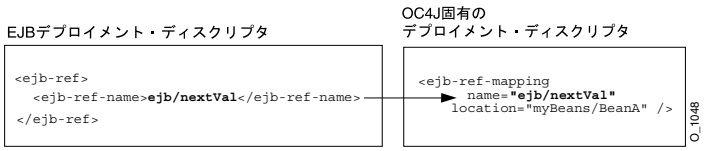
2. orion-ejb-jar.xml デプロイメント・ディスクリプタ内で、例 19-10 に示すように論理名をターゲット Bean の実際の名前にマッピングする <ejb-ref-mapping> 要素を定義します。

例 19-10 ejb-ref-mapping での実際の名前への論理名のマッピング

```
<ejb-ref-mapping name="ejb/nextVal" location="myBeans/BeanA"/>
```

図 19-5 に示すように、<ejb-ref-mapping> 要素では、name 属性を <ejb-ref-name> と一致するように構成し、location 属性をターゲット Bean の実際の名前で構成します。例 19-10 では、論理名 ejb/nextVal は、ターゲット Bean myBeans/BeanA の実際の名前にマッピングされます。

図 19-5 ejb-ref-name と ejb-ref-mapping の関連付け



OC4J は、論理名を、クライアント・サイドの実際の JNDI 名にマッピングします。サーバー・サイドで JNDI 名を受信し、これを JNDI ツリー内で解決します。

JDBC データソース・リソース・マネージャのコネクション・ファクトリへの環境参照の構成

デプロイ XML を使用して JDBC の DataSource 用の環境要素を作成することにより、JDBC を通じてデータベースにアクセスできます（19-13 ページの「デプロイ XML の使用方法」を参照）。

注意： EJB 3.0 では、リソース・マネージャのコネクション・ファクトリへの環境参照は不要です。リソース・インジェクションを使用して、リソース・マネージャのコネクション・ファクトリに直接アクセスできます（19-24 ページの「EJB 3.0 リソース・マネージャのコネクション・ファクトリのルックアップ」を参照）。

リソース・マネージャのコネクション・ファクトリのルックアップの詳細は、次を参照してください。

- 19-24 ページの「EJB 3.0 リソース・マネージャのコネクション・ファクトリのルックアップ」
- 19-26 ページの「EJB 2.1 リソース・マネージャのコネクション・ファクトリのルックアップ」

デプロイ XML の使用方法

デプロイ XML を使用して JDBC DataSource への参照を定義するには、次のようにします。

1. data-sources.xml ファイルで、目的の DataSource を定義し、その実際の JNDI 名を指定します（第 20 章「データソースの構成」を参照）。

   この例では、DataSource が、/test/OrderDataSource という JNDI 名で data-sources.xml ファイルで指定されていることを前提としています。

2. 適切なクライアント・デプロイメント・ディスクリプタの <resource-ref> 要素を定義し（19-4 ページの「EJB 環境参照を構成する場所」を参照）、例 19-11 に示すように次のサブ要素を構成します。

   - <res-ref-name>: JDBC データソースの論理名。

     参照名には jdbc という接頭辞を付けることをお薦めしますが、これは必須ではありません。Bean のソース・コードでこの参照をルックアップするために初期コンテキストを使用する場合は（19-26 ページの例 19-30 を参照）、必ず論理名に java:comp/env/ という接頭辞を付けます（java:comp/env/jdbc/OrderDB など）。

   - <res-type>: リソースの Java タイプ。JDBC DataSource オブジェクトの場合、これは javax.sq.DataSource です。

   - <res-auth>: 認証情報のソース（Application または Container）。

例 19-11 ejb-jar.xml での <resource-ref> の構成

```
<enterprise-beans>
...
    <resource-ref>
        <res-ref-name>jdbc/OrderDB</res-ref-name>
        <res-type>javax.sq.DataSource</res-type>
        <res-auth>Application</res-auth>
    </resource-ref>

</enterprise-beans>
```

3. orion-ejb-jar.xml デプロイメント・ディスクリプタで、例 19-12 に示すように <resource-ref-mapping> を定義して次の属性を構成します。

   - name: （ejb-jar.xml に定義された）データソースの論理名

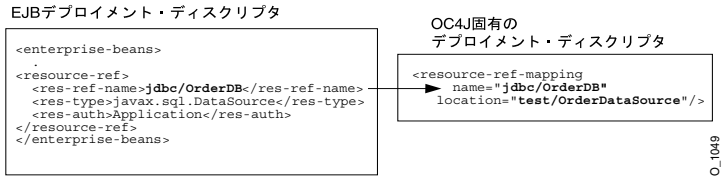
   - location: （data-sources.xml に定義された）データソースの実際の名前

例 19-12 <resource-ref-mapping> を使用した論理名から実際の JDBC データソース・リソース・マネージャのコネクション・ファクトリへのマッピング

```
<resource-ref-mapping
    name="jdbc/OrderDB"
    location="test/OrderDataSource"
/>
```

図 19-6 に、name 属性が jdbc/OrderDB（ejb-jar.xml で定義された論理名）に設定され、location 属性が test/OrderDataSource（data-sources.xml で定義された JNDI 名）に設定された <resource-ref-mapping> 要素を示します。

図 19-6 論理名から実際の JDBC データソース・リソース・マネージャのコネクション・ファクトリへのマッピング



Bean の実装では、論理名 java:comp/env/jdbc/OrderDB を使用して、このデータソースの JDBC データソース・リソース・マネージャのコネクション・ファクトリをルックアップできます（19-26 ページの例 19-30 を参照）。

JMS 宛先リソース・マネージャのコネクション・ファクトリへの環境参照の構成（JMS 1.1）

JMS 1.0 の場合と同じように（19-15 ページの「JMS 宛先またはコネクション・リソース・マネージャのコネクション・ファクトリへの環境参照の構成（JMS 1.0）」を参照）、JMS 1.1 を使用して、JMS コネクション・リソース・マネージャのコネクション・ファクトリへの環境参照を定義します。ただし、クライアント・デプロイメント・ディスクリプタの <message-destination-ref> 要素および対応する OC4J 固有のデプロイメント・ディスクリプタの <message-destination-ref-mapping> 要素を使用して JMS 宛先への環境参照を定義できます（19-4 ページの「EJB 環境参照を構成する場所」を参照）。

<message-destination-ref-mapping> を使用して、クライアント <message-destination-ref-name> を OC4J 環境で使用できる別の場所にマッピングします。これにより、メッセージ・コンシューマおよびプロデューサを 1 つ以上の共通の論理的な宛先にリンクする手段が提供されます。

<message-destination-ref> はすべての EJB タイプで使用できるため、<message-destination-ref-mapping> はメッセージドリブン・デプロイに限定されません。

詳細は、『Oracle Containers for J2EE サービス・ガイド』の「Oracle Enterprise Messaging Service（OEMS）」を参照してください。

注意： EJB 3.0 では、リソース・マネージャのコネクション・ファクトリへの環境参照は不要です。リソース・インジェクションを使用して、リソース・マネージャのコネクション・ファクトリに直接アクセスできます（19-24 ページの「EJB 3.0 リソース・マネージャのコネクション・ファクトリのルックアップ」を参照）。

リソース・マネージャのコネクション・ファクトリのルックアップの詳細は、次を参照してください。

- 19-24 ページの「EJB 3.0 リソース・マネージャのコネクション・ファクトリのルックアップ」
- 19-26 ページの「EJB 2.1 リソース・マネージャのコネクション・ファクトリのルックアップ」

JMS 宛先またはコネクション・リソース・マネージャのコネクション・ファクトリへの環境参照の構成（JMS 1.0）

JMS 宛先（キューまたはトピック）および JMS コネクション・リソース・マネージャのコネクション・ファクトリには、デプロイ XML を使用してそれらのコネクション・ファクトリへの環境参照を作成することでアクセスできます（19-15 ページの「デプロイ XML の使用方法」を参照）。

注意： EJB 3.0 では、リソース・マネージャのコネクション・ファクトリへの環境参照は不要です。アノテーションおよびリソース・インジェクションを使用して、リソース・マネージャのコネクション・ファクトリに直接アクセスできます（19-24 ページの「EJB 3.0 リソース・マネージャのコネクション・ファクトリのルックアップ」を参照）。

リソース・マネージャのコネクション・ファクトリのルックアップの詳細は、次を参照してください。

- 19-24 ページの「EJB 3.0 リソース・マネージャのコネクション・ファクトリのルックアップ」
- 19-26 ページの「EJB 2.1 リソース・マネージャのコネクション・ファクトリのルックアップ」

デプロイ XML の使用方法

JMS 宛先および JMS コネクション・リソース・マネージャのコネクション・ファクトリを定義するには、次のようにします。

1. JMS サービス・プロバイダを構成します。

   詳細は、次を参照してください。

   - 23-2 ページの「メッセージ・サービス・プロバイダで使用するための J2CA リソース・アダプタの構成」
   - 23-4 ページの「OEMS JMS メッセージ・サービス・プロバイダの構成」
   - 23-6 ページの「OEMS JMS データベース・メッセージ・サービス・プロバイダの構成」

2. JMS 宛先およびコネクション・ファクトリの JNDI 名を定義します。

   詳細は、次を参照してください。

   - 23-2 ページの「J2CA メッセージ・サービス・プロバイダのコネクション・ファクトリ名」
   - 23-4 ページの「OEMS JMS 宛先名およびコネクション・ファクトリ名」
   - 23-7 ページの「OEMS JMS データベース宛先名およびコネクション・ファクトリ名」

3. JMS 宛先およびコネクション・ファクトリの論理名を定義します。

   使用する JMS プロバイダのタイプにかかわらず、論理名の定義方法は同じです。

   a. 適切なクライアント・デプロイメント・ディスクリプタに `<resource-env-ref>` 要素を定義し（19-4 ページの「EJB 環境参照を構成する場所」を参照）、次のサブ要素を構成します。

      - `<resource-env-ref-name>`: JMS 宛先リソース・マネージャのコネクション・ファクトリの論理名。
      - `<resource-env-ref-type>`: 宛先クラス・タイプ。`javax.jms.Queue` または `javax.jms.Topic` です。

例 19-13 に、JMS トピック・リソース・マネージャのコネクション・ファクトリの <resource-env-ref> 要素を示します。

例 19-13 JMS トピック宛先の <resource-env-ref>

```
<resource-env-ref>
  <resource-env-ref-name>rpTestTopic</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Topic</resource-env-ref-type>
</resource-env-ref>
```

b. 同じクライアント・デプロイメント・ディスクリプタに <resource-ref> 要素を定義し、次のサブ要素を構成します。

- <res-ref-name>: JMS コネクション・リソース・マネージャのコネクション・ファクトリの論理名。
- <res-type>: コネクション・ファクトリ・クラス・タイプ。javax.jms.QueueConnectionFactory または javax.jms.TopicConnectionFactory です。
- <res-auth>: 認証を行う機能。Container または Bean です。
- <res-sharing-scope>: 共有スコープ。Shareable または Unshareable です。

例 19-14 に、JMS トピック・コネクション・リソース・マネージャのコネクション・ファクトリの <resource-ref> 要素を示します。

例 19-14 JMS トピックのコネクション・ファクトリの <resource-ref>

```
<resource-ref>
  <res-ref-name>myTCF</res-ref-name>
  <res-type>javax.jms.TopicConnectionFactory</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

4. 論理名を実際の JNDI 名にマッピングします。

a. 対応する OC4J 固有のデプロイメント・ディスクリプタ（19-4 ページの「EJB 環境参照を構成する場所」を参照）で <resource-env-ref-mapping> 要素を定義し、その name 属性を JMS 宛先論理名（<resource-env-ref> で定義）に、またその location 属性を JMS プロバイダの構成時に定義した JNDI 名に構成します（手順 2 を参照）。

例 19-15 に、OEMS JMS の <resource-env-ref-mapping> 要素を示します。

例 19-15 OEMS JMS の <resource-env-ref-mapping>

```
<resource-env-ref-mapping
    name="rpTestTopic"
    location="jms/Topic/rpTestTopic">
</resource-env-ref-mapping>
```

b. 同じ OC4J 固有のデプロイメント・ディスクリプタ（19-4 ページの「EJB 環境参照を構成する場所」を参照）の <resource-ref-mapping> を定義し、その name 属性を JMS コネクション・ファクトリの論理名（<resource-ref> で定義）に、またその location 属性を JMS プロバイダの構成時に定義した JNDI 名（手順 2 を参照）に構成します。

例 19-16 に、OEMS JMS の <resource-ref-mapping> 要素を示します。

例 19-16 OEMS JMS の <resource-ref-mapping>

```
<resource-ref-mapping
    name="myTCF"
    location="jms/Topic/myTCF">
</resource-ref-mapping>
```

環境変数への環境参照の構成

InitialContext の JNDI ルックアップを通じて Bean がアクセスする環境変数を作成できます。これらの変数は ejb-jar.xml ファイルの <env-entry> 要素で定義します。型は、String、Integer、Boolean、Double、Byte、Short、Long、Float のいずれかです。環境変数の名前は <env-entry-name> サブ要素で定義し、型は <env-entry-type> サブ要素で定義します。初期値は <env-entry-value> サブ要素で定義します。<env-entry-name> は java:comp/env コンテキストに対して相対的です。

例 19-17 に、ejb-jar.xml ファイルでの java:comp/env/minBalance および java:comp/env/maxCreditBalance の環境変数の定義方法を示します。

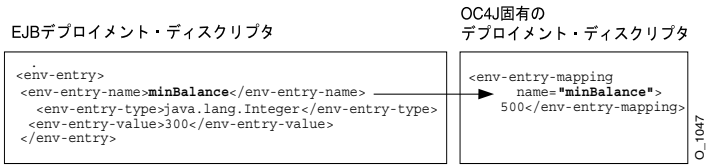
例 19-17 環境変数の ejb-jar.xml

```
<env-entry>
     <env-entry-name>minBalance</env-entry-name>
     <env-entry-type>java.lang.Integer</env-entry-type>
     <env-entry-value>500</env-entry-value>
</env-entry>
<env-entry>
     <env-entry-name>maxCreditBalance</env-entry-name>
     <env-entry-type>java.lang.Integer</env-entry-type>
     <env-entry-value>10000</env-entry-value>
</env-entry>
```

ejb-jar.xml ファイルで定義した環境変数の値は、name 属性が ejb-jar.xml ファイルで定義した env-entry-name と一致する env-entry-mapping 要素を orion-ejb-jar.xml ファイルに定義することでオーバーライドできます。ejb-jar.xml ファイルで指定された型は変わりません。

図 19-7 に、minBalance 環境変数の値を orion-ejb-jar.xml ファイルでオーバーライドして 500 に設定する方法を示します。

図 19-7 orion-ejb-jar.xml での ejb-jar.xml の環境変数のオーバーライド



環境変数のルックアップの詳細は、次を参照してください。

- 19-25 ページの「EJB 3.0 環境変数のルックアップ」
- 19-27 ページの「EJB 2.1 環境変数のルックアップ」

Web サービスへの環境参照の構成

Web サービスには、その Web サービスを参照するリソース・マネージャのコネクション・ファクトリを作成することでステートレス・セッション Bean からアクセスできます。

注意： EJB 3.0 では、Web サービスへの環境参照は不要です。アノテーションおよびリソース・インジェクションを使用して、Web サービスに直接アクセスできます。

リソース・マネージャのコネクション・ファクトリにアクセスする各クライアントについて、クライアント・ソース・コードにこれを注入するか、クライアントのデプロイメント・ディスクリプタに環境参照を定義する必要があります。

Web サービスへの環境参照を作成するには、次のようにします。

1. Web サービスの論理名を定義します。

   適切なクライアント・デプロイメント・ディスクリプタに `<service-ref>` 要素を定義し（19-4 ページの「EJB 環境参照を構成する場所」を参照）、次のサブ要素を構成します。

   - `<service-ref-name>`: Web サービスの論理名
   - `<service-interface>`: Web サービス・インタフェース

   例 19-18 に、Web サービスの `<service-ref>` 要素を示します。

   参照名には `service` という参照名を付けることをお薦めしますが、これは必須ではありません。Bean コードでは、この参照（30-4 ページの例 30-5 を参照）のルックアップは、常に先頭に `java:comp/env` が付きます（たとえば、`java:comp/env/service/myService` など）。

例 19-18 Web サービス論理名の ejb-jar.xml

```
<service-ref>
    <service-ref-name>service/StockQuoteService</service-ref-name>
    <service-interface>com.example.StockQuoteService</service-interface>
</service-ref>
```

2. 論理名を実際の JNDI 名にマッピングします。

   対応する OC4J 固有のデプロイメント・ディスクリプタ（19-4 ページの「EJB 環境参照を構成する場所」を参照）の `<service-ref-mapping>` を定義し、その `name` 属性を Web サービス論理名（`<service-ref>` で定義）に構成し、`<service-qname>` サブ要素を構成します。

   例 19-19 に、Web サービスの `<service-ref-mapping>` 要素を示します。

例 19-19 Web サービスの論理名から JNDI 名へのマッピング用の orion-ejb-jar.xml

```
<service-ref-mapping name="service/WebServiceBroker">
    <service-qname namespaceURI="urn:WebServiceBroker" localpart="WebServiceBroker"/>
</service-ref-mapping>
```

Web サービスのルックアップおよび使用方法の詳細は、第 30 章「EJB および Web サービスの使用方法」を参照してください。

永続性コンテキストへの環境参照の構成

エンティティ・マネージャを取得する最も簡単な方法は、@PersistenceContext アノテーションを使用することです（29-9 ページの「EntityManager の取得」を参照）。

ただし、アノテーションおよびインジェクションをサポートしないクラス、つまりヘルパー・クラスでエンティティ・マネージャを取得するには、最初に適切なデプロイメント・ディスクリプタ・ファイルに persistence-context-ref を定義する必要があります。

永続性コンテキストへの環境参照を作成するには、次のようにします。

1. 永続性コンテキストの論理名を定義します。

   適切なクライアント・デプロイメント・ディスクリプタに <persistence-context-ref> 要素を定義し（19-4 ページの「EJB 環境参照を構成する場所」を参照）、次のサブ要素を構成します。

   - <persistence-context-ref-name>: 永続性コンテキストの論理名。
   - <persistence-unit-name>: この永続性コンテキストに関連付けられている永続性ユニットの名前。

     この名前の永続性ユニットは persistence.xml ファイルで定義する必要があります。

     詳細は、次を参照してください。

     - 2-10 ページの「persistence.xml ファイルとは」
     - 26-4 ページの「persistence.xml ファイルの構成」

   例 19-20 に、web.xml ファイル内の永続性コンテキストの <persistence-context-ref> 要素を示します。

   参照名は persistence から開始することをお薦めしますが、これは必須ではありません。Bean コードでは、この参照（29-12 ページの「ヘルパー・クラスでのエンティティ・マネージャの取得」を参照）のルックアップは、常に先頭に java:comp/env が付きます（たとえば、java:comp/env/persistence/InventoryAppMgr など）。

例 19-20 永続性コンテキストの web.xml

```
...
    <servlet>
        <servlet-name>webTierEntryPoint</servlet-name>
        <servlet-class>com.sun.j2ee.blueprints.waf.controller.web.MainServlet</servlet-class>
        <init-param>
            <param-name>default_locale</param-name>
            <param-value>en_US</param-value>
        </init-param>
        <persistence-context-ref>
            <description>
                Persistence context for the inventory management application.
            </description>
            <persistence-context-ref-name>
                persistence/InventoryAppMgr
            </persistence-context-ref-name>
            <persistence-unit-name>
                InventoryManagement <!-- Defined in persistenc.xml -->
            </persistence-unit-name>
        </persistence-context-ref>
    </servlet>
...
```

エンティティ・マネージャのルックアップおよび使用方法の詳細は、29-12 ページの「ヘルパー・クラスでのエンティティ・マネージャの取得」を参照してください。

初期コンテキスト・ファクトリの構成

初期コンテキスト・ファクトリを使用して、初期コンテキスト（JNDI ネームスペースへの参照）を取得します。初期コンテキストを使用すると、JNDI API を使用して Enterprise Bean、リソース・マネージャのコネクション・ファクトリ、環境変数、または JNDI でアクセス可能なその他のオブジェクトをルックアップできます。

使用する初期コンテキスト・ファクトリのタイプは、表 19-2 に示すように、使用しているクライアントのタイプによって決まります。

表 19-2 クライアントの初期コンテキストの要件

クライアント・タイプ	ターゲット EJB との関連	初期コンテキスト・ファクトリ
任意のクライアント	クライアントとターゲット Enterprise Bean が同一 JVM 上に置かれます。	デフォルト（19-20 ページの「デフォルトの初期コンテキスト・ファクトリの構成」を参照）
任意のクライアント	クライアントとターゲット Enterprise Bean は同じアプリケーションにデプロイされます。	デフォルト（19-20 ページの「デフォルトの初期コンテキスト・ファクトリの構成」を参照）
任意のクライアント	クライアントの親として指定されるアプリケーションにデプロイされるターゲット Enterprise Bean。[1]	デフォルト（19-20 ページの「デフォルトの初期コンテキスト・ファクトリの構成」を参照）
EJB クライアント サーブレットまたは JSP クライアント	クライアントとターゲット Enterprise Bean は同一 JVM 上に置かれず、同じアプリケーションにデプロイされません。ターゲット EJB アプリケーションはクライアントの親ではありません。[1]	`oracle.j2ee.rmi.RMIInitialContextFactory`（19-21 ページの「Oracle 初期コンテキスト・ファクトリの構成」を参照）
スタンドアロン Java クライアント	クライアントとターゲット Enterprise Bean は同一 JVM 上に置かれず、同じアプリケーションにデプロイされません。ターゲット EJB アプリケーションはクライアントの親ではありません。[1]	`oracle.j2ee.naming.ApplicationClientInitialContextFactory`（19-21 ページの「Oracle 初期コンテキスト・ファクトリの構成」を参照）

[1] アプリケーションの親の設定方法は、『Oracle Containers for J2EE 開発者ガイド』を参照してください。

注意： このリリースでは、RMI およびアプリケーション・クライアントの初期コンテキスト・ファクトリの新しいパッケージ名に注意してください。

詳細は、次を参照してください。

- 『Oracle Containers for J2EE セキュリティ・ガイド』
- 『Oracle Containers for J2EE サービス・ガイド』

デフォルトの初期コンテキスト・ファクトリの構成

ターゲット Bean と同一 JVM 上に置かれているクライアント（表 19-2 を参照）は、そのノードの JNDI プロパティに自動的にアクセスします。したがって、JNDI プロパティは必要ないため、Enterprise Bean へのアクセスは簡単です。

例 19-21 デフォルトの初期コンテキストの構成

```
//Get the Initial Context for the JNDI lookup for a local EJB
InitialContext ic = new InitialContext();
//Retrieve the Home interface using JNDI lookup
Object helloObject = ic.lookup("java:comp/env/ejb/HelloBean");
```

Oracle 初期コンテキスト・ファクトリの構成

クライアントに Oracle 初期コンテキスト・ファクトリが必要な場合は（表 19-2 を参照）、次の JNDI プロパティを設定する必要があります。

JNDI プロパティの設定の詳細は、19-23 ページの「Enterprise Bean での JNDI プロパティの設定」を参照してください。

1. `java.naming.factory.initial` プロパティをクライアントに適した Oracle 初期コンテキスト・ファクトリ（表 19-2 を参照）で定義します。
2. `java.naming.provider.url` プロパティを OC4J インストール環境に適したネーミング・プロバイダ URL で定義します。
   - 19-22 ページの「OC4J および Oracle Application Server のネーミング・プロバイダの構成」
   - 19-22 ページの「OC4J スタンドアロンのネーミング・プロバイダ URL の構成」
3. 例 19-22 に示すように、`HashTable` を作成し、`javax.naming.Context` フィールドをキーとし、`String` オブジェクトを値として使用して必要なプロパティを移入します。

例 19-22 初期コンテキスト・ファクトリのプロパティの指定

```
Hashtable env = new Hashtable();
env.put("java.naming.factory.initial",
        "oracle.j2ee.server.ApplicationClientInitialContextFactory");
env.put("java.naming.provider.url",
        "opmn:ormi://opmnhost:6004:oc4j_inst1/ejbsamples");
```

4. 初期コンテキストをインスタンス化する場合は、例 19-23 に示すように `HashTable` を初期コンテキスト・コンストラクタに渡します。

例 19-23 JNDI でアクセス可能なリソースの初期コンテキスト・ルックアップのインスタンス化

```
Context ic = new InitialContext (env);
```

5. 初期コンテキストを使用して、JNDI でアクセス可能なリソースをルックアップします。
   - 19-24 ページの「EJB 3.0 リソース・マネージャのコネクション・ファクトリのルックアップ」
   - 19-25 ページの「EJB 3.0 環境変数のルックアップ」
   - 19-26 ページの「EJB 2.1 リソース・マネージャのコネクション・ファクトリのルックアップ」
   - 19-27 ページの「EJB 2.1 環境変数のルックアップ」
   - 第 29 章「クライアントからの Enterprise Bean へのアクセス」

OC4J および Oracle Application Server のネーミング・プロバイダの構成

Oracle Application Server のインストールで、OPMN は 1 つ以上の OC4J インスタンスを管理します。この場合、java.naming.provider.url の値は次の形式である必要があります。

```
opmn:ormi://<hostname>:<opmn-request-port>:<oc4j-instance-name>/<application-name>
```

このプロバイダ URL のフィールドは、次のように定義されます。

- <hostname>: Oracle Application Server が実行されているホストの名前。
- <opmn-request-port>: この構成では、ORMI ポートを使用するかわりに OPMN リクエスト・ポートを使用する必要があります。OPMN リクエスト・ポートは、次のように opmn.xml にあります。

  ```
  <notification-server>
      <port local="6100" remote="6200" request="6003"/>
      ...
  </notification-server>
  ```

  デフォルトの OPMN リクエスト・ポートは 6003 です。
- <oc4j-instance-name>: この構成では、OPMN がロード・バランシング / フェイルオーバーに使用する複数の OC4J プロセスが存在する場合があります。アプリケーションにデプロイされるインスタンスの名前を使用します。

  デフォルトのインスタンス名は home です。

たとえば、ホスト名が dpanda-us、リクエスト・ポートが 6003、インスタンス名が home1 の場合、プロバイダ URL は次のようになります。

```
opmn:ormi://dpanda-us:6003:home1/ejbsamples
```

詳細は、次を参照してください。

- 『Oracle Containers for J2EE サービス・ガイド』の「RMI 用の JNDI プロパティの設定」
- 24-3 ページの「静的検出ロード・バランシングの構成」
- 24-4 ページの「DNS ロード・バランシングの構成」

OC4J スタンドアロンのネーミング・プロバイダ URL の構成

スタンドアロンの OC4J インストールでは、java.naming.provider.url の値は次の形式である必要があります。

```
ormi://<hostname>:<ormi-port>/<application-name>
```

このプロバイダ URL のフィールドは、次のように定義されます。

- <hostname>: OC4J が実行されているホストの名前
- <ormi-port>: 次のように rmi.xml ファイルで構成されている ORMI ポート

  ```
  <rmi-server
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/rmi-server-10_0.xsd"
      port="23791"
      schema-major-version="10"
      schema-minor-version="0"
  >
  ...
  </rmi-server>
  ```

  デフォルト・ポートは 23791 です。
- <application-name>: server.xml ファイルで構成されているアプリケーション名

たとえば、ホスト名が dpanda-us、ORMI ポートが 23793、アプリケーション名が ejb30slsb の場合、プロバイダ URL は次のようになります。

```
ormi://dpanda-us:23793/ejb30slsb
```

詳細は、次を参照してください。

- 『Oracle Containers for J2EE サービス・ガイド』の「RMI 用の JNDI プロパティの設定」
- 24-3 ページの「静的検出ロード・バランシングの構成」
- 24-4 ページの「DNS ロード・バランシングの構成」

Enterprise Bean での JNDI プロパティの設定

クライアントがターゲットと同一 JVM 上に置かれていて、ターゲットと同じアプリケーション内に存在している場合、またはターゲットが親の中に存在する場合、JNDI プロパティを初期化する必要はありません。それ以外の場合は、次のいずれかの方法で JNDI プロパティを初期化する必要があります。

この項の内容は次のとおりです。

- JNDI プロパティ・ファイルでの JNDI プロパティの設定
- システム・プロパティでの JNDI プロパティの設定
- 初期コンテキストでの JNDI プロパティの設定

詳細は、次を参照してください。

- 22-11 ページの「EJB クライアントの資格証明の指定」
- 『Oracle Containers for J2EE サービス・ガイド』

JNDI プロパティ・ファイルでの JNDI プロパティの設定

java.util.Properties のメソッド load で指定されている要件に準拠した、jndi.properties という名前の JNDI プロパティを設定できます。

JNDI プロパティは次のように設定します。

```
<PropertyName>=<PropertyValue>
```

次に例を示します。

```
java.naming.factory.initial= oracle.j2ee.server.ApplicationClientInitialContextFactory
```

プロパティ名については、javax.naming.Context のフィールド定義を参照してください。

例については、22-12 ページの「JNDI プロパティの資格証明の指定」を参照してください。

jndi.properties ファイル内で JNDI プロパティを設定する場合は、必ずこのファイルをクライアントの CLASSPATH からアクセス可能にしてください。または、このファイルを OC4J 固有の適切なデプロイ XML ファイルの ejb-ref-mapping の属性 jndi-properties-file に指定してください（19-7 ページの「リモート EJB への環境参照の構成 : クラスタ化されていない個別の Web 層および EJB 層」を参照）。

システム・プロパティでの JNDI プロパティの設定

JNDI プロパティは、コマンドラインで -D 引数として、または環境参照（19-17 ページの「環境変数への環境参照の構成」を参照）として指定されたシステム・プロパティとして設定できます。

初期コンテキストでの JNDI プロパティの設定

JNDI プロパティは、HashTable を作成し、javax.naming.Context フィールドをキーとして、また String オブジェクトを値として使用して必要なプロパティを移入することにより設定できます。初期コンテキストをインスタンス化する場合、HashTable を初期コンテキスト・コンストラクタに渡します。

例については、22-12 ページの「初期コンテキストでの資格証明の指定」を参照してください。

EJB 3.0 リソース・マネージャのコネクション・ファクトリのルックアップ

EJB 3.0 を使用している場合は、リソース・インジェクション（19-24 ページの「アノテーションの使用方法」を参照）または InitialContext（19-24 ページの「初期コンテキストの使用方法」を参照）を使用してリソース管理接続をルックアップできます。

アノテーションの使用方法

例 19-24 に、アノテーションおよび依存性注入を使用して EJB 3.0 リソース・マネージャのコネクション・ファクトリにアクセスする方法を示します。

例 19-24 EJB 3.0 リソース・マネージャのコネクション・ファクトリの注入

```
@Stateless public class EmployeeServiceBean implements EmployeeService {
    ...
    public void sendEmail(String emailAddress) {
        @Resource Session testMailSession;
        ...
    }
}
```

初期コンテキストの使用方法

例 19-25 に、初期コンテキストを使用して EJB 3.0 リソース・マネージャのコネクション・ファクトリをルックアップする方法を示します。

例 19-25 EJB 3.0 リソース・マネージャのコネクション・ファクトリのルックアップ

```
@Stateless public class EmployeeServiceBean implements EmployeeService {
    ...
    public void sendEmail(String emailAddress) {
        InitialContext ic = new InitialContext();
        Session session = (Session) ic.lookup("java:comp/env/mail/testMailSession");
        ...
    }
}
```

詳細は、19-20 ページの「初期コンテキスト・ファクトリの構成」を参照してください。

EJB 3.0 環境変数のルックアップ

EJB 3.0 を使用している場合は、リソース・インジェクション（19-25 ページの「リソース・インジェクションの使用方法」を参照）または `InitialContext`（19-26 ページの「初期コンテキストの使用方法」を参照）を使用して環境変数をルックアップできます。

リソース・インジェクションの使用方法

リソース・インジェクションを使用している場合は、次のいずれかを使用して、コンテナに依存してフィールドまたは setter メソッド（プロパティ）を初期化できます。

- デフォルトの JNDI 名（`java:comp/env/<FieldOrPropertyName>` の形式）
- 指定する明示的な JNDI 名（名前に接頭辞 `java:comp/env` を付けないでください。）

同じ JNDI 名を使用してフィールドと setter の両方を注入することはできません。

次の例では、デフォルトの JNDI 名 `java:comp/env/maxExemptions` で環境変数に対して指定された値で `maxExemptions` フィールドを初期化する方法を示します。

例 19-27 に示すように、フィールド・レベル（例 19-26 を参照）または setter メソッド（プロパティ）レベルでリソース・インジェクションを使用できます。

例 19-26 デフォルトの環境変数名を使用したフィールド・レベルでのリソース・インジェクション

```
@Stateless public class EmployeeServiceBean implements EmployeeService {
    ...
    // The maximum number of tax exemptions, configured by Deployer
    // Assumes JNDI name java:comp/env/maxExemptions.
    @Resource int maxExemptions;
    ...
    public void setMaxExemptions(int maxEx) {
        maxExemptions = maxEx;
    }
    ...
}
```

例 19-27 デフォルトの環境変数名を使用したプロパティ・レベルでのリソース・インジェクション

```
@Stateless public class EmployeeServiceBean implements EmployeeService {
    ...
    int maxExemptions;
    ...
    // Assumes JNDI name java:comp/env/maxExemptions.
    @Resource
    public void setMaxExemptions(int maxEx) {
        maxExemptions = maxEx;
    }
    ...
}
```

例 19-28 に示すように、明示的な JNDI 名を指定できます。

例 19-28 特定の環境変数名でのリソース・インジェクション

```
@Stateless public class EmployeeServiceBean implements EmployeeService {
    ...
    int maxExemptions;
    ...
    @Resource(name="ApplicationDefaults/maxExemptions")
    public void setMaxExemptions(int maxEx) {
        maxExemptions = maxEx;
    }
    ...
}
```

初期コンテキストの使用方法

例 19-29 に、InitialContext を使用して Bean のコード内でこれらの環境変数をルックアップする方法を示します。

例 19-29 環境変数のルックアップ

```
InitialContext ic = new InitialContext();
Integer min = (Integer) ic.lookup("java:comp/env/minBalance");
Integer max = (Integer) ic.lookup("java:comp/env/maxCreditBalance"));
```

環境変数の値を取得するには、各環境変数の先頭に java:comp/env/ を付加する必要があります。これは、コンテナが環境変数を格納する場所です。

詳細は、19-20 ページの「初期コンテキスト・ファクトリの構成」を参照してください。

EJB 2.1 リソース・マネージャのコネクション・ファクトリのルックアップ

EJB 2.1 を使用している場合は、InitialContext（19-26 ページの「初期コンテキストの使用方法」を参照）を使用してリソース・マネージャのコネクション・ファクトリをルックアップできます。

リソースの構成の詳細は、19-3 ページの「リソース・マネージャのコネクション・ファクトリ環境参照」を参照してください。

初期コンテキストの使用方法

例 19-30 に、java:comp/env/jdbc 接頭辞の付いた EJB デプロイメント・ディスクリプタ（19-12 ページの「JDBC データソース・リソース・マネージャのコネクション・ファクトリへの環境参照の構成」を参照）で定義されている論理名を持つ InitialContext を使用して Bean のコード内の JDBC データソース・リソース・マネージャのコネクション・ファクトリをルックアップする方法を示します。

例 19-30 JDBC データソース・リソース・マネージャのコネクション・ファクトリのルックアップ

```
javax.sql.DataSource db;
java.sql.Connection conn;
...
InitialContext ic = new InitialContext();
db = (javax.sql.DataSource) initCtx.lookup("java:comp/env/jdbc/OrderDB");
conn = db.getConnection();
```

詳細は、19-20 ページの「初期コンテキスト・ファクトリの構成」を参照してください。

EJB 2.1 環境変数のルックアップ

EJB 2.1 を使用している場合は、`InitialContext`（19-27 ページの「初期コンテキストの使用方法」を参照）を使用して環境変数をルックアップできます。

環境変数の構成の詳細は、19-17 ページの「環境変数への環境参照の構成」を参照してください。

初期コンテキストの使用方法

例 19-29 に、`InitialContext` を使用して Bean のコード内でこれらの環境変数をルックアップする方法を示します。

例 19-31 環境変数のルックアップ

```
InitialContext ic = new InitialContext();
Integer min = (Integer) ic.lookup("java:comp/env/minBalance");
Integer max = (Integer) ic.lookup("java:comp/env/maxCreditBalance"));
```

環境変数の値を取得するには、各環境変数の先頭に `java:comp/env/` を付加する必要があります。これは、コンテナが環境変数を格納する場所です。

詳細は、19-20 ページの「初期コンテキスト・ファクトリの構成」を参照してください。

20

データソースの構成

この章の内容は次のとおりです。

- Oracle データベースのデータソースの構成
- サード・パーティ・データベースのデータソースの構成
- EJB 3.0 アプリケーションのデフォルトのデータソースの構成
- EJB 2.1 アプリケーションのデフォルトのデータソースの構成
- TopLink と Oracle JDBC ドライバとの関連付け

詳細は、次を参照してください。

- 2-17 ページの「EJB データソース・サービスについて」
- 26-6 ページの「永続性ユニットでのデータソースの指定」
- 『Oracle Containers for J2EE サービス・ガイド』の「データソース」

注意： データソース・コード例は、`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html` からダウンロードできます。

Oracle データベースのデータソースの構成

Oracle データベースのデータソースを作成するには、マネージド・データソースを作成します。マネージド・データソースは、Application Server Control コンソール（20-2 ページの「Application Server Control コンソールの使用方法」を参照）またはデプロイ XML（20-2 ページの「デプロイ XML の使用方法」を参照）を使用して作成できます。

詳細は、次を参照してください。

- 2-17 ページの「OC4J でサポートされるデータソースのタイプ」
- 『Oracle Containers for J2EE サービス・ガイド』の「データソース」

Application Server Control コンソールの使用方法

Application Server Control コンソールを使用して、OC4J を再起動せずにマネージド・データソースを動的に作成できます。

詳細は `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html` を参照してください。

デプロイ XML の使用方法

例 20-1 に示すように、Oracle データベースのマネージド・データソースは、`data-sources.xml` ファイルで `connection-pool` 要素および `managed-data-source` 要素を構成することにより構成できます。

例 20-1 Oracle JDBC データソースの data-sources.xml

```
<connection-pool name="ScottConnectionPool">
    <connection-factory
        factory-class="oracle.jdbc.pool.OracleDataSource"
        user="scott"
        password="tiger"
        url="jdbc:oracle:thin:@//localhost:1521/ORCL" >
    </connection-factory>
</connection-pool>

<managed-data-source
    name="OracleManagedDS"
    jndi-name="jdbc/OracleDS"
    connection-pool-name="ScottConnectionPool"
    tx-level="global"
/>
```

`connection-factory` 要素には、サービスベースの接続 URL を必ず指定してください（2-19 ページの「OC4J での接続 URL の定義方法」を参照）。

マネージド・データソースでは、デフォルトでグローバル（2 フェーズ・コミット）トランザクションがサポートされます。ローカル・トランザクションのみをサポートするようにマネージド・データソースを構成するには、`managed-data-source` の属性 `tx-level` を `local` に設定します。詳細は、2-19 ページの「データソースでサポートされるトランザクションのタイプ」を参照してください。

詳細は、次を参照してください。

- `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`
- `http://www.oracle.com/technology/tech/java/newsletter/articles/oc4j_datasource_config.html`

この方法を使用してマネージド・データソースを構成する場合は、OC4J を再起動して変更を適用する必要があります。または、Application Server Control コンソールを使用して、OC4J を再起動せずにデータソースを動的に作成できます（20-2 ページの「Application Server Control コンソールの使用方法」を参照）。

サード・パーティ・データベースのデータソースの構成

サード・パーティ（Oracle 以外）のデータベースのデータソースを作成するには、ネイティブ・データソースを作成します。ネイティブ・データソースは、Application Server Control コンソール（20-2 ページの「Application Server Control コンソールの使用方法」を参照）またはデプロイ XML（20-2 ページの「デプロイ XML の使用方法」を参照）を使用して作成できます。

詳細は、次を参照してください。

- 2-17 ページの「OC4J でサポートされるデータソースのタイプ」
- 『Oracle Containers for J2EE サービス・ガイド』の「データソース」

Application Server Control コンソールの使用方法

Application Server Control コンソールを使用して、OC4J を再起動せずにネイティブ・データソースを動的に作成できます。

詳細は `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html` を参照してください。

デプロイ XML の使用方法

例 20-2 に、サード・パーティ・データベース（この例では SQLServer）のネイティブ・データソース要素を定義する方法を示します。

例 20-2 サード・パーティ・データベースの data-sources.xml

```
<native-data-source
  name="nativeDataSource"
  jndi-name="jdbc/nativeDS"
  description="Native DataSource"
  data-source-class="com.ddtek.jdbcx.sqlserver.SQLServerDataSource"
  user="frank"
  password="frankpw"
  url="jdbc:datadirect:sqlserver://server_name:1433;User=usr;Password=pwd">
</native-data-source>
```

ネイティブ・データソースでは、デフォルトでローカル・トランザクションのみがサポートされます。グローバル（2 フェーズ・コミット）トランザクションの場合は、マネージド・データソースを構成します。詳細は、2-19 ページの「データソースでサポートされるトランザクションのタイプ」を参照してください。

詳細は、次を参照してください。

- `http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html`
- `http://www.oracle.com/technology/tech/java/newsletter/articles/oc4j_datasource_config.html`

この方法を使用してネイティブ・データソースを構成する場合は、OC4J を再起動して変更を適用する必要があります。または、Application Server Control コンソールを使用して、OC4J を再起動せずにネイティブ・データソースを動的に作成できます（20-3 ページの「Application Server Control コンソールの使用方法」を参照）。

EJB 3.0 アプリケーションのデフォルトのデータソースの構成

デプロイ XML を使用して EJB 3.0 アプリケーションのデフォルトのデータソースを構成できます（20-4 ページの「デプロイ XML の使用方法」を参照）。

詳細は、次を参照してください。

- 2-19 ページの「デフォルトのデータソース」
- 『Oracle Containers for J2EE サービス・ガイド』の「データソース」

デプロイ XML の使用方法

EJB 3.0 アプリケーションのデフォルトのデータソースを構成するには、次のようにします。

1. `orion-application.xml` ファイルの `default-data-source` 属性で、デフォルトのデータソースの名前を設定します。
2. EJB 3.0 アプリケーションをカスタマイズして、`ejb3-toplink-session.xml` ファイルにこの名前のデータソースを定義します。

   詳細は、次を参照してください。

   - 2-9 ページの「ejb3-toplink-sessions.xml ファイルとは」
   - 3-4 ページの「JPA 永続性プロバイダのカスタマイズ」

EJB 2.1 アプリケーションのデフォルトのデータソースの構成

デプロイ XML を使用して EJB 2.1 アプリケーションのデフォルトのデータソースを構成できます（20-4 ページの「デプロイ XML の使用方法」を参照）。

詳細は、次を参照してください。

- 2-19 ページの「デフォルトのデータソース」
- 『Oracle Containers for J2EE サービス・ガイド』の「データソース」

デプロイ XML の使用方法

EJB 2.1 アプリケーションのデフォルトのデータソースを構成するには、次のようにします。

1. `orion-application.xml` ファイルの `orion-application` 要素の `default-data-source` 属性で、デフォルトのデータソースの名前を設定します。
2. `orion-ejb-jar.xml` ファイルの `entity-deployment` 要素の `data-source` 属性で、デフォルトのデータソースの名前を設定します。
3. `<OC4J_HOME>/j2ee/home/config/data-sources.xml` ファイルで、デフォルトのデータソースを定義します。

TopLink と Oracle JDBC ドライバとの関連付け

このリリースでは、デフォルトで TopLink が Oracle JDBC ドライバのバージョン 10.2（ojdbc14_102.jar）と関連付けられます。

このバージョンの Oracle JDBC ドライバが、使用する Oracle データベースのリリースに適切ではない場合、TopLink を別のバージョンの Oracle JDBC ドライバに関連付けることができます。

TopLink を別のバージョンの Oracle JDBC ドライバに関連付ける方法は、次のように作成するアプリケーションのタイプに応じて異なります。

- EJB 3.0 アプリケーションおよび EJB 2.1 CMP 以外のアプリケーション
- EJB 2.1 CMP アプリケーション
- EIS AQ コネクタ・アプリケーション

詳細は、『Oracle Containers for J2EE 開発者ガイド』の OC4J クラス・ロード・フレームワークの仕様に関する項を参照してください。

EJB 3.0 アプリケーションおよび EJB 2.1 CMP 以外のアプリケーション

EJB 3.0 アプリケーションおよび EJB 2.1 CMP 以外のアプリケーションの場合、次の制限に注意してください。

- サーバーでは Oracle JDBC ドライバの複数のバージョンを同時に使用できますが、各アプリケーションで使用できるバージョンは 1 つのみです。
- oracle.jdbc 共有ライブラリのバージョンごとに、対応するバージョンの oracle.toplink 共有ライブラリを server.xml に定義する必要があります。
- Oc4jPlatform のみ使用できます。Oc4jPlatform_10_1_3 は使用できません。
- インポートする共有ライブラリのバージョンを指定しない場合、最高バージョンの共有ライブラリがインポートされます。たとえば、oracle.jdbc のバージョン 10.2 が server.xml に定義されており、system-application.xml でバージョンを指定せずに oracle.jdbc をインポートすると、oracle.toplink では oracle.jdbc のバージョン 10.2 が使用されます。

EJB 3.0 または EJB 2.1 の CMP 以外のアプリケーションをデフォルトとは異なる特定のバージョンの Oracle JDBC ドライバに関連付ける場合、次のようにします。

1. 新規 Oracle JDBC ドライバの共有ライブラリ用として <ORACLE_HOME>/j2ee/home/shared-lib/oracle.jdbc にフォルダを作成します。

   この例では、<ORACLE_HOME>/j2ee/home/shared-lib/oracle.jdbc/10.3 というフォルダを作成します。

   実際の Oracle JDBC ドライバの JAR ファイルを参照する場合は、このディレクトリを基準にします。Oracle JDBC ドライバの JAR ファイルは、このディレクトリに配置して単純に名前で参照するか、他のディレクトリに配置してこのディレクトリに対する相対的な部分パスを使用して参照します。

2. 例 20-3 に示すように、新規 Oracle JDBC ドライバの共有ライブラリを server.xml に定義します。

例 20-3 server.xml での Oracle JDBC ドライバ・バージョン 10.3 の共有ライブラリの定義

```
...
<shared-library name="oracle.jdbc" version="10.3">
    <code-source path="ojdbc14_103.jar"/>
</shared-library>
...
```

   oracle.jdbc 共有ライブラリの名前を、使用する Oracle JDBC ドライバのバージョンに対応する様々なバージョン番号（この例では 10.3）と組み合せて使用します。

この例では、code-source の属性 path は、ojdbc14_103.jar という単純な名前です（この場合、<ORACLE_HOME>/j2ee/home/shared-lib/oracle.jdbc/10.3 に JAR ファイルが配置されていると仮定します）。または、<ORACLE_HOME>/j2ee/home/shared-lib/oracle.jdbc/10.3 ディレクトリに対する相対的な部分パスとして path を設定することも可能です。

3. 例 20-4 に示すように、対応する TopLink 共有ライブラリを server.xml に定義します。

例 20-4 server.xml での Oracle JDBC ドライバ・バージョン 10.3 に対応する oracle.toplink 共有ライブラリの定義

```
...
<shared-library name="oracle.jdbc" version="10.3">
    <code-source path="ojdbc14_103.jar"/>
</shared-library>
<shared-library name="oracle.toplink" version="10.3" library-compatible="true">
    <code-source path="../../../../../toplink/jlib/toplink.jar"/>
    <code-source path="../../../../../toplink/jlib/antlr.jar"/>
    <code-source path="../../../../../toplink/jlib/cciblackbox-tx.jar"/>
    <import-shared-library name="oc4j.internal"/>
    <import-shared-library name="oracle.xml"/>
    <import-shared-library name="oracle.jdbc" max-version="10.3"/>
    <import-shared-library name="oracle.dms"/>
</shared-library>
...
```

oracle.toplink 共有ライブラリの名前を、使用する Oracle JDBC ドライバのバージョンに対応する様々なバージョン番号（この例では 10.3）と組み合せて使用します。この oracle.toplink 共有ライブラリでは、必ず oracle.jdbc 共有ライブラリの適切なバージョン（この例では max-version="10.3"）をインポートしてください。

注意： 新規 oracle.toplink ライブラリで元の JAR ファイルと同じ JAR ファイルを使用し、単に同じものを作成して oracle.jdbc の別のバージョンを指定する場合、このライブラリに対応するフォルダを <ORACLE_HOME>/j2ee/home/shared-lib に作成することを避けるには、例 20-4 に示すように shared-library の属性 library-compatible を true に設定します。

---

4. 次のように、アプリケーションの新規共有ライブラリをインポートします。

   a. 新規 oracle.jdbc および oracle.toplink 共有ライブラリを OC4J インスタンスのすべてのアプリケーションのデフォルトに設定するには、例 20-5 に示すように system-applications.xml を更新します。

例 20-5 system-applications.xml におけるすべてのアプリケーション用の共有ライブラリのインポート

```
...
    <imported-shared-libraries>
...
        <import-shared-library name="oracle.jdbc" min-version="10.3" max-version="10.3"/>
        <import-shared-library name="oracle.toplink" min-version="10.3" max-version="10.3"/>
...
    </imported-shared-libraries>
...
```

   b. 新規 oracle.jdbc および oracle.toplink 共有ライブラリを特定のアプリケーションにのみ適用するには、例 20-6 に示すようにそのアプリケーションの orion-applications.xml を更新します。

   この場合、例 20-6 に示すように、orion-applications.xml ファイルと同じフォルダに存在し、orion-applications.xml ファイルで参照される data-sources.xml ファイルにデータソースを定義する必要があります。

例 20-6 orion-applications.xml における特定のアプリケーション用の共有ライブラリのインポート

```
...
<orion-application>
    <ejb-module remote="true" path="simpleobject_ejb.jar" />
    <client-module path="simpleobject_ejb.jar" auto-start="false" />
    <persistence path="persistence" />
    <imported-shared-libraries>
        <import-shared-library name="oracle.jdbc" max-version="10.3"/>
        <import-shared-library name="oracle.toplink" max-version="10.3"/>
    </imported-shared-libraries>
    <log>
        <file path="application.log" />
    </log>
    <data-sources path="data-sources.xml" />
    <namespace-access>
        ..........
    </namespace-access>
</orion-application>
```

EJB 2.1 CMP アプリケーション

EJB 2.1 CMP アプリケーションの場合、次の制限に注意してください。

- TopLink で Oracle JDBC ドライバの複数のバージョンを使用することはできません。
- インポートする共有ライブラリのバージョンを指定しない場合、元のバージョンが使用されます。たとえば、`oracle.jdbc` のバージョン 10.2 が `server.xml` に定義されており、`system-application.xml` でバージョンを指定せずに `oracle.jdbc` をインポートすると、`oracle.toplink` では元の `oracle.jdbc` のバージョン 10.1 が使用されます。
- この場合、TopLink ランタイムは、任意の `oracle.toplink.platform.server.oc4j` プラットフォーム・インスタンスを使用できます。詳細は、3-15 ページの「TopLink EJB 2.1 永続性マネージャのカスタマイズ」を参照してください。

EIS AQ コネクタ・アプリケーション

EIS AQ コネクタを使用するアプリケーションをデフォルトとは異なる特定のバージョンの Oracle JDBC ドライバに関連付ける場合、20-5 ページの「EJB 3.0 アプリケーションおよび EJB 2.1 CMP 以外のアプリケーション」の手順に従います。

この場合、例 20-7 に示すように、新規 `oracle.jdbc` 共有ライブラリで `aqapi.jar` ファイルをリロードする必要もあります。

例 20-7 server.xml での Oracle JDBC ドライバ・バージョン 10.3 の共有ライブラリの定義

```
...
<shared-library name="oracle.jdbc" version="10.3">
    <code-source path="ojdbc14_103.jar"/>
    <code-source path="../../../../../rdbms/jlib/aqapi.jar"/>
</shared-library>
...
```

21

トランザクション・サービスの構成

この章の内容は次のとおりです。

- EJB 3.0 トランザクション管理の構成
- EJB 3.0 トランザクション属性の構成
- EJB 2.1 トランザクション管理の構成
- EJB 2.1 トランザクション属性の構成
- トランザクション・タイムアウトの構成
- トランザクションのベスト・プラクティス

詳細は、次を参照してください。

- 2-20 ページの「EJB トランザクション・サービスについて」
- 『Oracle Containers for J2EE サービス・ガイド』の Java Transaction API（JTA）に関する項

EJB 3.0 トランザクション管理の構成

EJB 3.0 EJB トランザクション管理を構成するには、アノテーション（21-2 ページの「アノテーションの使用方法」を参照）またはデプロイ XML（21-3 ページの「デプロイ XML の使用方法」を参照）を使用します。

注意： EJB 3.0 エンティティは、トランザクション管理タイプで構成できません。EJB 3.0 エンティティは、コール元のトランザクション・コンテキスト内で実行されます。

詳細は、次を参照してください。

- 2-21 ページの「トランザクションの管理担当」
- 2-21 ページの「コンテナ管理のトランザクションとは」
- 2-22 ページの「Bean 管理のトランザクションとは」

アノテーションの使用方法

例 21-1 に示すように、トランザクション管理は `@TransactionManagement` アノテーションの属性 `value` を使用して構成できます。次のいずれかの値を指定できます。

- `TransactionManagementType.CONTAINER`: コンテナ管理のトランザクション（デフォルト）
- `TransactionManagementType.BEAN`: Bean 管理のトランザクション

`@TransactionManagement` アノテーションは、クラス・レベルで適用します。

例 21-1 EJB 3.0 セッション Bean のトランザクション管理の構成

```
import javax.ejb.Stateful
import javax.annotation.PostConstruct;
import javax.ejb.Remove;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;

@Stateful
@TransactionManagement(value=TransactionManagementType.CONTAINER)
public class CartBean implements Cart {
    private ArrayList items;

    @PostConstruct
    public void initialize() {
        items = new ArrayList();
    }

    @Remove
    public void finishedShipping() {
        // Release any resources.
    }

    public void addItem(String item) {
        items.add(item);
    }

    public void removeItem(String item) {
        items.remove(item);
    }
}
```

デプロイ XML の使用方法

EJB 3.0 EJB では、EJB 2.1 Enterprise Bean と同様に、`ejb-jar.xml` ファイルでトランザクション管理を構成します（21-5 ページの「デプロイ XML の使用方法」を参照）。

EJB 3.0 トランザクション属性の構成

クライアントがコンテナ管理のトランザクション用に構成された EJB 3.0 Enterprise Bean のメソッドを起動したときにコンテナがトランザクションを管理する方法を構成するには、アノテーション（21-3 ページの「アノテーションの使用方法」を参照）またはデプロイ XML（21-4 ページの「デプロイ XML の使用方法」を参照）を使用します。

詳細は、次を参照してください。

- 2-21 ページの「トランザクションの管理担当」
- 2-21 ページの「コンテナ管理のトランザクションとは」
- 2-23 ページの「クライアントがビジネス・メソッドを起動する際のトランザクションの処理方法」

アノテーションの使用方法

例 21-2 に示すように、トランザクション管理は `@TransactionAttribute` アノテーションの属性 `value` を使用して構成できます。表 21-1 に、ユーザーが指定できる `TransactionAttributeType` の値と、メソッドの起動時にクライアント管理のトランザクションが存在するかどうかに応じて変化するコンテナの対応動作を示します。

表 21-1 @TransactionAttribute の TransactionAttributeType の値

トランザクション属性	クライアント管理のトランザクションが存在	クライアント管理のトランザクションが存在しない
`NOT_SUPPORTED`	コンテナはクライアント・トランザクションを一時停止します。	トランザクションを使用しません。
`SUPPORTS`	クライアント管理のトランザクションを使用します。	トランザクションを使用しません。
`REQUIRED`[1]	クライアント管理のトランザクションを使用します。	コンテナは新規トランザクションを開始します。
`REQUIRES_NEW`	クライアント管理のトランザクションを使用します。	コンテナは新規トランザクションを開始します。
`MANDATORY`	クライアント管理のトランザクションを使用します。	例外が発生します。
`NEVER`	例外が発生します。	トランザクションを使用しません。

[1] デフォルト。

`@TransactionAttribute` アノテーションをクラス・レベルで適用すると、Enterprise Bean のすべてのビジネス・メソッドに対応するデフォルトのトランザクション属性を指定できます。このアノテーションをメソッド・レベルで適用すると、そのメソッドのトランザクション属性を指定できます。メソッド・レベルで適用されたアノテーションは、そのメソッドのクラス・レベルのアノテーション（存在する場合）をオーバーライドします。

例 21-2 EJB 3.0 セッション Bean のトランザクション属性の構成

```
import javax.ejb.Stateful;
import javax.annotation.PostConstruct;
import javax.ejb.Remove;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;
import javax.ejb.TransactionAttribute;
import static javax.ejb.TransactionAttributeType.REQUIRED;
import static javax.ejb.TransactionAttributeType.REQUIRES_NEW;
import com.acme.Cart;

@Stateful
@TransactionManagement(value=TransactionManagementType.CONTAINER)
@TransactionAttribute(value=REQUIRED)
public class CartBean implements Cart {
    private ArrayList items;

    @PostConstruct
    public void initialize() {
        items = new ArrayList();
    }

    @Remove
    @TransactionAttribute(value=REQUIRES_NEW)
    public void finishedShipping() {
        // Release any resources.
    }

    public void addItem(String item) {
        items.add(item);
    }

    public void removeItem(String item) {
        items.remove(item);
    }
}
```

デプロイ XML の使用方法

EJB 3.0 Enterprise Bean では、EJB 2.1 Enterprise Bean と同様に、`orion-ejb-jar.xml` ファイルでトランザクション属性を構成します（21-6 ページの「デプロイ XML の使用方法」を参照）。

EJB 2.1 トランザクション管理の構成

特定の EJB 2.1 Enterprise Bean に関連するトランザクションの管理方法を構成できます（21-5 ページの「デプロイ XML の使用方法」を参照）。

注意： EJB 2.1 エンティティ Bean は、常にコンテナ管理のトランザクション境界を使用する必要があります。EJB 2.1 エンティティ Bean は、Bean 管理のトランザクション境界で指定できません。

詳細は、次を参照してください。

- 2-21 ページの「トランザクションの管理担当」
- 2-21 ページの「コンテナ管理のトランザクションとは」
- 2-22 ページの「Bean 管理のトランザクションとは」

デプロイ XML の使用方法

トランザクション管理を構成するには、例 21-3 に示すように、`ejb-jar.xml` ファイルの `<transaction-type>` サブ要素を使用します。

有効な値は、`Container` または `Bean` です。デフォルトは `Container` です。

例 21-3 EJB 2.1 セッション Bean のトランザクション管理の構成

```
<enterprise-beans>
    <session>
        <display-name>A Credit-Service Bean</display-name>
        <ejb-name>CreditService</ejb-name>
        <home>creditService.ejb.CreditServiceHome</home>
        <remote>creditService.ejb.CreditServiceRemote</remote>
        <ejb-class>creditService.ejb.CreditServiceBean</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
        ...
    </session>
...
</enterprise-beans>
```

すべてのセッション Bean、エンティティ Bean およびメッセージドリブン Bean を対象に `<transaction-type>` を構成できます。ただし、EJB 2.1 エンティティ Bean に対しては、`<transaction-type>` を `Container` としてのみ構成できます。

EJB 2.1 トランザクション属性の構成

コンテナ管理のトランザクションを使用する Enterprise Bean では、クライアントが Bean メソッドを起動したときにコンテナがトランザクションを管理する方法を構成できます（21-6 ページの「デプロイ XML の使用方法」を参照）。

詳細は、次を参照してください。

- 2-21 ページの「トランザクションの管理担当」
- 2-21 ページの「コンテナ管理のトランザクションとは」
- 2-23 ページの「クライアントがビジネス・メソッドを起動する際のトランザクションの処理方法」

デプロイ XML の使用方法

クライアントが Bean メソッドを起動したときにコンテナがトランザクションを管理する方法を構成するには、例 21-4 に示すように、`ejb-jar.xml` ファイルの `<assembly-descriptor>` のサブ要素 `<container-transaction>` を使用します。

例 21-4 EJB 2.1 セッション Bean のトランザクション属性の構成

```
<assembly-descriptor>
    <container-transaction>
        <method>
            <ejb-name>CreditService</ejb-name>
            <method-name>setLimit</method-name>
            <method-params>
                <method-param>int</method-param>
            </method-params>
        </method>
        <trans-attribute>Required</trans-attribute>
    </container-transaction>
...
</assembly-descriptor>
```

`<container-transaction>` 要素には、1 つ以上の `<method>` 要素と 1 つの `<trans-attribute>` 要素が含まれます。`<trans-attribute>` 要素は、すべての `<method>` 要素に適用されます。メソッドは、名前で、または名前とパラメータ（シグネチャ）で指定できます。または、ワイルドカード `<method-name>*</method-name>` を使用して、特定の Enterprise Bean のすべてのメソッドを指定できます。

表 21-2 に、ユーザーが指定できる <trans-attribute> 要素の値と、メソッドの起動時にクライアント管理のトランザクションが存在するかどうかに応じて変化するコンテナの対応動作を示します。

表 21-2 <trans-attribute> 要素の有効な値

トランザクション属性	クライアント管理のトランザクションが存在	クライアント管理のトランザクションが存在しない
NotSupported[1]	コンテナはクライアント・トランザクションを一時停止します。	トランザクションを使用しません。
Supports[2]	クライアント管理のトランザクションを使用します。	トランザクションを使用しません。
Required[3]	クライアント管理のトランザクションを使用します。	コンテナは新規トランザクションを開始します。
RequiresNew	コンテナはクライアント・トランザクションを一時停止して新規トランザクションを作成します。	コンテナは新規トランザクションを開始します。
Mandatory	クライアント管理のトランザクションを使用します。	例外が発生します。
Never	例外が発生します。	トランザクションを使用しません。

[1] EJB 2.1 メッセージドリブン Bean のデフォルト。
[2] EJB 2.1 セッション Bean および BMP エンティティ Bean のデフォルト。
[3] EJB 2.1 CMP エンティティ Bean のデフォルト。

トランザクション・タイムアウトの構成

アプリケーション・パフォーマンスを向上させるために、OC4J がトランザクションのコミットまたはロールバックを待機する時間の長さを決定するトランザクション・タイムアウトを構成できます。

この項の内容は次のとおりです。

- グローバル・トランザクション・タイムアウトの構成
- セッション Bean のトランザクション・タイムアウトの構成
- メッセージドリブン Bean のトランザクション・タイムアウトの構成

グローバル・トランザクション・タイムアウトの構成

OC4J がセッションおよびエンティティ Bean について管理するすべてのトランザクションにグローバルに適用するトランザクション・タイムアウトを設定できます。

次の方法でグローバル・トランザクション・タイムアウトを構成できます。

- Application Server Control コンソールの使用方法
- デプロイ XML の使用方法

Application Server Control コンソールの使用方法

Application Server Control コンソール（31-2 ページの「Oracle Enterprise Manager 10g Application Server Control の使用方法」を参照）を使用した場合は、JTAResource MBean の属性 transactionTimeout を設定できます。

詳細は、『Oracle Containers for J2EE サービス・ガイド』の「OC4J トランザクション・マネージャの構成」を参照してください。

デプロイ XML の使用方法

<OC4J_HOME>¥j2ee¥home¥config¥transaction-manager.xml ファイルで、<transaction-manager> 要素の transaction-timeout 属性を使用してグローバル・トランザクション・タイムアウトを設定します。

たとえば、グローバル・トランザクション・タイムアウトを 180 秒に設定する場合は、次のようにします。

```
<transaction-manager ...  transaction-timeout="180"
  ...
</transaction-manager>
```

この方法を使用してこのプロパティを変更する場合は、OC4J を再起動して変更を適用する必要があります。または、Application Server Control コンソールを使用して、OC4J を再起動せずにこのパラメータを動的に変更できます（21-7 ページの「Application Server Control コンソールの使用方法」を参照）。

セッション Bean のトランザクション・タイムアウトの構成

各セッション Bean のトランザクション・タイムアウトは、OC4J 固有のアノテーション（21-8 ページの「アノテーションの使用方法」を参照）または orion-ejb-jar.xml ファイル（21-9 ページの「デプロイ XML の使用方法」を参照）を使用して指定できます。

セッション Bean のトランザクション・タイムアウトは、グローバル・トランザクション・タイムアウトをオーバーライドします（21-7 ページの「グローバル・トランザクション・タイムアウトの構成」を参照）。

デプロイ XML の構成は、アノテーションを使用して設定された対応する構成をオーバーライドします。

アノテーションの使用方法

EJB 3.0 セッション Bean のトランザクション・タイムアウトは、次の OC4J 固有のアノテーションとその属性を使用して指定できます。

- @StatelessDeployment の属性 transactionTimeout
- @StatefulDeployment の属性 transactionTimeout

これらの属性の詳細は、表 A-1 を参照してください。

例 21-5 に、@StatelessDeployment アノテーションを使用して EJB 3.0 ステートレス・セッション Bean でこれらの属性を構成する方法を示します。

例 21-5 @StatelessDeployment の transactionTimeout 属性

```
import javax.ejb.Stateless;
import oracle.j2ee.ejb.StatelessDeployment;

@Stateless
@StatelessDeployment(transactionTimeout=10)
public class HelloWorldBean implements HelloWorld {
    public void sayHello(String name) {
        System.out.println("Hello " + name + " from first EJB3.0");
    }
}
```

デプロイ XML の使用方法

orion-ejb-jar.xml ファイルで、<session-deployment> 要素の transaction-timeout 属性を使用してセッション Bean のトランザクション・タイムアウトを設定します。

たとえば、グローバル・トランザクション・タイムアウトを 180 秒に設定する場合は、次のようにします。

```
<session-deployment ...  transaction-timeout="180"
  ...
</session-deployment>
```

この方法を使用してこのプロパティを変更する場合は、OC4J を再起動して変更を適用する必要があります。

メッセージドリブン Bean のトランザクション・タイムアウトの構成

メッセージドリブン Bean のトランザクション・タイムアウトは、OC4J 固有のアノテーション（21-10 ページの「アノテーションの使用方法」を参照）または orion-ejb-jar.xml ファイル（21-11 ページの「デプロイ XML の使用方法」を参照）を使用して構成できます。

グローバル・トランザクション・タイムアウト（21-7 ページの「グローバル・トランザクション・タイムアウトの構成」を参照）はメッセージドリブン Bean に適用されないため、メッセージドリブン Bean のデフォルトのトランザクション・タイムアウトを変更する場合は、各メッセージドリブン Bean のトランザクション・タイムアウトを構成する必要があります。

使用するメッセージ・サービス・プロバイダのタイプ（2-26 ページの「MDB で使用できるメッセージ・サービス・プロバイダ」を参照）は、次のようにトランザクション・タイムアウト・オプションに影響します。

- J2EE Connector Architecture（J2CA）アダプタ・メッセージ・プロバイダ : トランザクション・タイムアウトを変更できます（21-11 ページの「J2CA アダプタ・メッセージ・サービス・プロバイダ」を参照）。
- OEMS JMS: トランザクション・タイムアウトは、デフォルトの 86,400 秒（1 日）から変更できません。
- OEMS JMS データベース : トランザクション・タイムアウトを変更できます（21-11 ページの「J2CA 以外のアダプタ・メッセージ・サービス・プロバイダ」を参照）。

デプロイ XML の構成は、アノテーションを使用して設定された対応する構成をオーバーライドします。

アノテーションの使用方法

EJB 3.0 セッション Bean のトランザクション・タイムアウトは、OC4J 固有のアノテーション `@MessageDrivenDeployment` の属性 transactionTimeout を使用して指定できます。

この属性の詳細は、表 A-3 を参照してください。

例 21-6 に、`@MessageDrivenDeployment` アノテーションを使用して EJB 3.0 メッセージドリブン Bean でこの属性を構成する方法を示します。

例 21-6 @MessageDrivenDeployment

```
import javax.ejb.MessageDriven;
import oracle.j2ee.ejb.MessageDrivenDeployment;
import javax.ejb.ActivationConfigProperty;
import javax.annotation.Resource;

@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="messageListenerInterface",
            propertyValue="javax.jms.MessageListener"),
        @ActivationConfigProperty(
            propertyName="connectionFactoryJndiName",
            propertyValue="jms/TopicConnectionFactory"),
         @ActivationConfigProperty(
            propertyName="destinationName",
            propertyValue="jms/demoTopic"),
         @ActivationConfigProperty(
            propertyName="destinationType",
            propertyValue="javax.jms.Topic"),
         @ActivationConfigProperty(
            propertyName="messageSelector",
            propertyValue="RECIPIENT = 'MDB'")
    }
)
@MessageDrivenDeployment(transactionTimeout=10)
public class MessageLogger implements MessageListener, TimedObject {
    @Resource javax.ejb.MessageDrivenContext mc;

    public void onMessage(Message message) {
    ...
    }

    public void ejbTimeout(Timer timer) {
    ...
    }
}
```

デプロイ XML の使用方法

orion-ejb-jar.xml ファイルで、トランザクション・タイムアウトを設定します。この値の構成方法は、使用しているメッセージドリブン・プロバイダのタイプによって決まります。

- J2CA 以外のアダプタ・メッセージ・サービス・プロバイダ
- J2CA アダプタ・メッセージ・サービス・プロバイダ

J2CA 以外のアダプタ・メッセージ・サービス・プロバイダ

OEMS JMS や OEMS JMS データベースなどの J2CA 以外のアダプタ・メッセージ・サービス・プロバイダを使用している場合は、<message-driven-deployment> 要素の transaction-timeout 属性を使用します。

たとえば、OEMS JMS または OEMS JMS データベースを使用していて、トランザクション・タイムアウトを 180 秒に設定する場合は、次のようにします。

```
<message-driven-deployment ...  transaction-timeout="180"
  ...
</message-driven-deployment>
```

J2CA アダプタ・メッセージ・サービス・プロバイダ

J2CA アダプタ・メッセージ・サービス・プロバイダを使用している場合は、<config-property> 要素を使用して、transactionTimeout 構成プロパティを設定します。

たとえば、J2CA アダプタ・メッセージ・サービス・プロバイダを使用していて、トランザクション・タイムアウトを 180 秒に設定する場合は、次のようにします。

```
<message-driven-deployment ... >
...
    <config-property>
        <config-property-name>transactionTimeout</config-property-name>
        <config-property-value>180</config-property-value>
    </config-property>
...
</message-driven-deployment>
```

どちらの場合も、このメソッドを使用してこのプロパティを変更する場合は、OC4J を再起動して変更を適用する必要があります。

トランザクションのベスト・プラクティス

この項では、EJB アプリケーションのトランザクションを使用する次のような推奨アプローチを説明します。

- データソース接続でのコンテナ管理のトランザクションの使用方法
- ロールバック計画の使用方法

データソース接続でのコンテナ管理のトランザクションの使用方法

コンテナ管理のトランザクションを使用していて、データソース接続を使用する場合は、トランザクションのコミットまで接続が解放されないことに注意してください。このことは、ループ内でデータソース接続を使用している場合に特に重要です。この場合は、ループの外部で接続を取得および解放して、接続プールが不注意で使い果されるのを回避する必要があります。

コンテナ管理のトランザクションを構成するセッション Bean について考えます。このセッション Bean には、例 21-7 に示すようにメソッド `runQueryConnectionEveryTime` があります。このメソッドがコールされると、コンテナ管理のトランザクションが開かれます。`for` ループを反復するたびに、接続が取得されて閉じられます。ただし、閉じられた接続は、メソッドが返され、コンテナ管理のトランザクションがコミットするまで解放されません。反復の回数に応じて、この設計では接続プールが使い果される場合があります。

この問題を回避するには、例 21-8 に示すように、ループの外部で接続を取得し、閉じる必要があります。これにより、コンテナ管理のトランザクションがコミットするまで 1 つの接続しか保持されないようにできます。

例 21-7 不正な例 : コミットまで複数の接続が保持される

```
public static long runQueryConnectionEveryTime (int count) {
    InitialContext ic = new InitialContext();
    DataSource ds = (DataSource) ic.lookup("jdbc/OracleDS");

    for (int i = 0; i < count; i++) {
        Connection con = ds.getConnection();  //connection created inside loop

        PreparedStatement ps = con.prepareStatement(
            "select AAA_ID, AAA_A FROM AAA_TABLE where AAA_ID = ? ");

        OracleStatement os = (OracleStatement)ps;
        os.defineColumnType(1, Types.BIGINT);
        ps.setLong(1, i);
        ResultSet rs = ps.executeQuery();
        rs.close();
        ps.close();

        con.close(); //connection closed inside loop
    }
}
```

例 21-8 正しい例 : コミットまで保持される接続数は 1 つのみ

```
public static long runQueryConnectionEveryTime (int count) {
    InitialContext ic = new InitialContext();
    DataSource ds = (DataSource) ic.lookup("jdbc/OracleDS");

    Connection con = ds.getConnection();  //connection created outside loop

    for (int i = 0; i < count; i++) {
        PreparedStatement ps = con.prepareStatement(
            "select AAA_ID, AAA_A FROM AAA_TABLE where AAA_ID = ? ");

        OracleStatement os = (OracleStatement)ps;
        os.defineColumnType(1, Types.BIGINT);
```

```
            ps.setLong(1, i);
            ResultSet rs = ps.executeQuery();
            rs.close();
            ps.close();
        }

        con.close(); //connection closed outside loop
    }
```

ロールバック計画の使用方法

コンテナ管理のトランザクション境界のある Enterprise Bean では、その `javax.ejb.EJBContext` オブジェクトの `setRollbackOnly` メソッドを使用して、コミットできないようにトランザクションをマークできます。

通常は、この処理を行って、アプリケーション例外が原因でコンテナがトランザクションを自動的にロールバックしない場合にアプリケーション例外をスローする前に、データ整合性を保護します。

たとえば、1 つのアカウントを借方計上し、別のアカウントに貸方計上する `AccountTransfer` Bean は、借方計上を正常に実行し、貸方操作で障害が発生した場合に、トランザクションにロールバックのマークを付けることができます。

詳細は、次を参照してください。

- 1-7 ページの「EJB コンテキストとは」
- 29-22 ページの「EJB 3.0 EJBContext へのアクセス」
- 29-30 ページの「EJB 2.1 EJBContext へのアクセス」

22

セキュリティ・サービスの構成

この章では、Java EE アプリケーションに特に適用される次のようなセキュリティ・サービス構成について説明します。

- ブラウザにおける権限の付与
- EJB アプリケーションでのユーザー、グループおよびロールの定義
- EJB クライアントの資格証明の指定
- EJB 3.0 セキュリティ・アノテーションの使用方法
- JAAS API を使用した Enterprise Bean からの資格証明の取得
- EJB アプリケーションのカスタム JAAS ログイン・モジュールの定義

詳細は、次を参照してください。

- 2-24 ページの「EJB セキュリティ・サービスについて」
- 『Oracle Containers for J2EE セキュリティ・ガイド』

ブラウザにおける権限の付与

Security Manager がアクティブなクライアントで EJB アプリケーションをダウンロードする場合は、ダウンロードの前に次の権限を付与する必要があります。

```
permission java.net.SocketPermission "*:*", "connect,resolve";
permission java.lang.RuntimePermission "createClassLoader";
permission java.lang.RuntimePermission "getClassLoader";
permission java.util.PropertyPermission "*", "read";
permission java.util.PropertyPermission "LoadBalanceOnLookup", "read,write";
```

EJB アプリケーションでのユーザー、グループおよびロールの定義

EJB の認証と認可については、EJB デプロイメント・ディスクリプタを構成して、各メソッドを実行する基礎となるプリンシパルを定義します。コンテナでは、メソッドを実行するユーザーが、デプロイメント・ディスクリプタで定義されたユーザーと同じであることが必要です。

EJB デプロイメント・ディスクリプタを使用すると、各メソッドを実行できるセキュリティ・ロールを定義できます。このメソッドは、OC4J 固有のデプロイメント・ディスクリプタで、ユーザーまたはグループにマッピングされます。ユーザーとグループは、指定したセキュリティ・ユーザー・マネージャ（Oracle Application Server Java Authentication and Authorization Service（JAAS）Provider（OracleAS JAAS Provider）または XML ユーザー・マネージャのいずれかを使用）内で定義されます。セキュリティ・ユーザー・マネージャの詳細は、『Oracle Containers for J2EE サービス・ガイド』を参照してください。

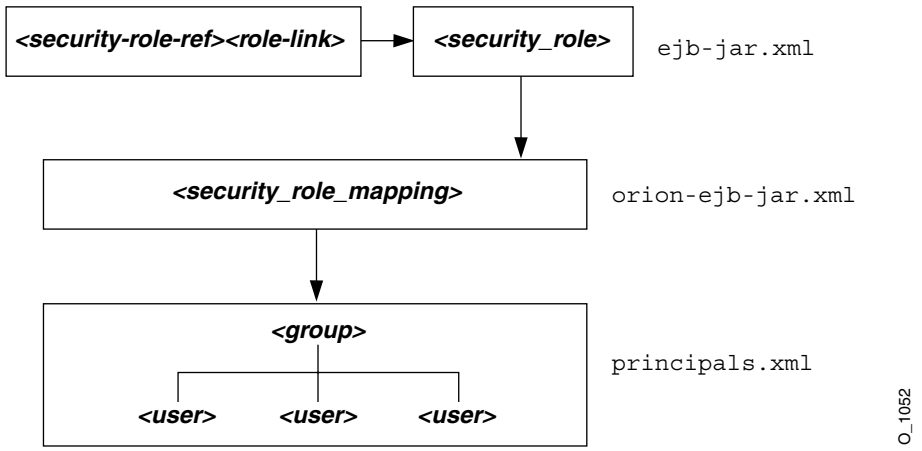
この項では、認証と認可に関して、EJB デプロイメント・ディスクリプタ内の XML 構成を説明します。EJB の認可は、EJB および OC4J 固有のデプロイメント・ディスクリプタ内で指定されます。セキュリティの許可の部分は、次のように、デプロイメント・ディスクリプタ内で管理できます。

- EJB デプロイメント・ディスクリプタは、論理ロールを使用して、アクセス・ルールを記述します。
- OC4J 固有のデプロイメント・ディスクリプタは、論理ロールを具体的なユーザーおよびグループにマッピングします。これは、OracleAS JAAS Provider または XML ユーザー・マネージャのいずれかで定義されています。

ユーザーおよびグループは、コンテナによって認識される認識情報です。ロールは、アプリケーションが、各オブジェクトへのアクセス権を示すために使用する論理識別情報です。ユーザー名およびパスワードには、デジタル証明が使用可能で、SSL の場合、秘密鍵も使用可能です。

ロールの定義およびマッピングを、図 22-1 に示します。

図 22-1 ロールのマッピング



ユーザー、グループおよびロールの定義について、次の各項で説明します。

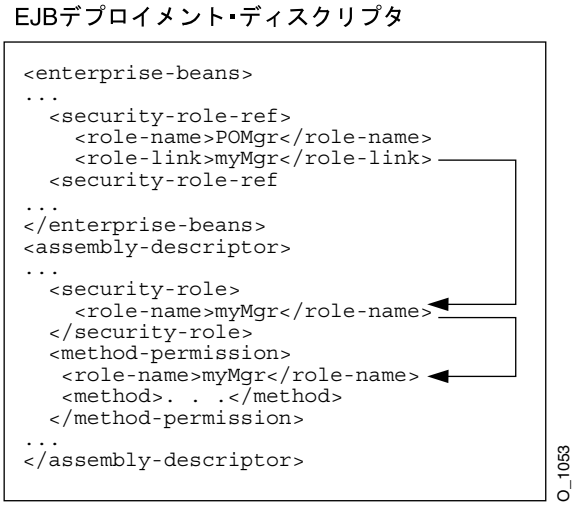
ユーザーおよびグループの指定

OC4J では、ユーザーおよびグループの定義をサポートしています。これには、すべてのデプロイ済アプリケーションで共有されているものと、特定のアプリケーション固有のものの両方が含まれます。共有またはアプリケーション固有のユーザーとグループは、OracleAS JAAS Provider または XML ユーザー・マネージャのいずれかで定義します。詳細は、『Oracle Containers for J2EE サービス・ガイド』を参照してください。

EJB デプロイメント・ディスクリプタでの論理ロールの指定

図 22-2 に示すように、Bean 実装内でロールの論理名を使用して、この論理名を適切なデータベース・ロールまたはユーザーにマッピングできます。論理名のデータベース・ロールへのマッピングは、OC4J 固有のデプロイメント・ディスクリプタで指定されます。詳細は、22-9 ページの「ユーザーおよびグループへの論理ロールのマッピング」を参照してください。

図 22-2 セキュリティのマッピング



isCallerInRole などのメソッドの Bean 実装内でデータベース・ロールの論理名を使用する場合は、次の手順を実行して、実際のデータベース・ロールに論理名をマッピングできます。

1. <enterprise-beans> セクションの <security-role-ref> 要素内で論理名を宣言します。たとえば、発注の例で使用するロールを定義する場合は、Bean 実装内で、コール元が発注にサインする認可を受けているかどうかをチェックしておくことができます。したがって、コール元は、適切なロールでサインオンする必要があります。Bean によるデータベース・ロールの認識を不要にするため、POMgr などの論理名で isCallerInRole をチェックできます。これは、注文を承認できるのは発注マネージャのみであるためです。したがって、論理セキュリティ・ロールの POMgr を、次のように、<enterprise-beans> セクションの <security-role-ref><role-name> 要素で定義します。

```
<enterprise-beans>
...
  <security-role-ref>
   <role-name>POMgr</role-name>
   <role-link>myMgr</role-link>
  </security-role-ref>
</enterprise-beans>
```

<security-role-ref> 要素内の <role-link> 要素は、実際のデータベース・ロールの場合があり、<assembly-descriptor> セクション内で詳細に定義されます。また、この要素は別の論理名の場合があり、<assembly-descriptor> セクションで詳細に定義され、Oracle 固有のデプロイメント・ディスクリプタ内で実際のデータベース・ロールにマッピングされます。

注意： <security-role-ref> 要素は必要ありません。この要素を指定するのは、Bean 内でセキュリティ・コンテキスト・メソッドを使用する場合のみです。

---

2. ロールおよびロールを適用するメソッドを定義します。発注の例にある PurchaseOrder Bean で実行されるメソッドは、myMgr として認可されている必要があります。PurchaseOrder は、<entity | session><ejb-name> 要素で宣言された名前です。

   次の例では、ロールを myMgr、Enterprise Bean を PurchaseOrder、およびすべてのメソッドを * 記号で示して定義しています。

注意： <security-role> 要素内の myMgr ロールは、<enterprise-beans> セクション内の <role-link> 要素と同じです。これによって、POMgr の論理名が myMgr 定義に関連付けられます。

```
<assembly-descriptor>
 <security-role>
  <description>Role needed purchase order authorization</description>
  <role-name>myMgr</role-name>
 </security-role>
 <method-permission>
  <role-name>myMgr</role-name>
  <method>
   <ejb-name>PurchaseOrder</ejb-name>
   <method-name>*</method-name>
  </method>
 </method-permission>
...
</assembly-descriptor>
```

前述の 2 つの手順を実行した後は、Bean 実装内で POMgr を参照でき、コンテナは POMgr を myMgr に変換します。

注意： 同じ Bean のメソッドに対して <method-permission> 要素内で別のロールを定義すると、この Bean のメソッドに対して定義されたすべてのメソッド許可の組合せが付与されます。

EJB メソッドに対するロールの指定

Enterprise Bean メソッドの起動を許可されるセキュリティ・ロールを指定できます。

EJB 3.0 アプリケーションでは、アノテーションを使用できます（22-5 ページの「アノテーションの使用方法」を参照）。

EJB 3.0 または EJB 2.1 アプリケーションでは、ejb-jar.xml デプロイメント・ディスクリプタを使用できます（22-5 ページの「デプロイ XML の使用方法」を参照）。

アノテーションの使用方法

EJB 3.0 アプリケーションでは、例 22-1 に示すように @RolesAllowed アノテーションを使用して、アプリケーションでのメソッドへのアクセスを許可されるセキュリティ・ロールを指定できます。

例 22-1 @RolesAllowed

```
@RolesAllowed("Users")
public class Calculator {

    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
    ...
    }
}
```

このアノテーションは、クラス、メソッドまたはその両方に適用できます。

メソッドに適用される場合、指定によってクラス指定がオーバーライドされます（存在する場合）。

セキュリティ・アノテーションの詳細は、22-13 ページの「EJB 3.0 セキュリティ・アノテーションの使用方法」を参照してください。

デプロイ XML の使用方法

<method-permission><method> 要素を使用して、インタフェースまたは実装内の 1 つ以上のメソッドについてセキュリティ・ロールを指定します。この定義は、EJB 仕様に従って、次のいずれかの形式になります。

- Bean 名を指定し、Bean 内のすべてのメソッドを示す * 文字を使用して、Bean 内のすべてのメソッドを定義します。次に例を示します。

```
<method-permission>
  <role-name>myMgr</role-name>
  <method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
```

- Bean 内で一意に識別できる特定のメソッドを定義します。適切なインタフェース名とメソッド名を使用します。次に例を示します。

```
<method-permission>
  <role-name>myMgr</role-name>
  <method>
    <ejb-name>myBean</ejb-name>
    <method-name>myMethodInMyBean</method-name>
  </method>
</method-permission>
```

> **注意：** オーバーロードされた同じ名前のメソッドが複数ある場合、このスタイルの要素は、オーバーロードされた名前を持つすべてのメソッドを参照します。

- オーバーロードされた多数のメソッドの中から、特定のシグネチャを持つメソッドを定義します。次に例を示します。

```
<method-permission>
  <role-name>myMgr</role-name>
  <method>
    <ejb-name>myBean</ejb-name>
    <method-name>myMethod</method-name>
    <method-params>
      <method-param>javax.lang.String</method-param>
      <method-param>javax.lang.String</method-param>
    </method-params>
  </method>
</method-permission>
```

パラメータは、完全に修飾された Java タイプのメソッドの入力パラメータです。メソッドに入力引数がない場合、<method-params> 要素内に要素は含まれません。配列を指定するには、配列要素のタイプの後に 1 つ以上の角カッコ（int[ ][ ] など）を指定します。

EJB メソッドに対するセキュリティ・チェックなしの指定

特定のメソッドでセキュリティ・ロールをチェックしないようにするには、そのメソッドをチェックなしとして定義します。

EJB 3.0 アプリケーションでは、アノテーションを使用できます（22-7 ページの「アノテーションの使用方法」を参照）。

EJB 3.0 または EJB 2.1 アプリケーションでは、ejb-jar.xml デプロイメント・ディスクリプタを使用できます（22-7 ページの「デプロイ XML の使用方法」を参照）。

アノテーションの使用方法

EJB 3.0 アプリケーションでは、例 22-2 に示すように @PermitAll アノテーションを使用して、すべてのセキュリティ・ロールがアプリケーションでのメソッドへのアクセスを許可されるように指定できます。

例 22-2 @PermitAll

```
@RolesAllowed("Users")
public class Calculator {

    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
    ...
    }
    @PermitAll
    public long convertCurrency(long amount) {
    ...
    }
}
```

このアノテーションは、クラスまたはメソッドに適用できます。

クラスに適用される場合、指定はすべてのメソッドに適用されます。

メソッドに適用される場合、指定はそのメソッドにのみ適用されます。

このアノテーションを使用する場合は、22-13 ページの「EJB 3.0 セキュリティ・アノテーションの使用方法」で説明する制限に注意してください。

デプロイ XML の使用方法

次のように <method-permission><unchecked> 要素を使用して、すべてのセキュリティ・ロールがメソッドへのアクセスを許可されるように指定します。

```
<method-permission>
  <unchecked/>
  <method>
     <ejb-name>EJBNAME</ejb-name>
     <method-name>*</method-name>
  </method>
</method-permission>
```

<role-name> 要素を定義するかわりに、<unchecked/> 要素を定義します。これによって、EJBNAME Bean で任意のメソッドを実行すると、コンテナはセキュリティをチェックしません。チェックなしのメソッドは、常に、他のロール定義をオーバーライドします。

runAs セキュリティ識別情報の指定

Enterprise Bean のすべてのメソッドが特定の識別情報を使用して実行されるように指定できます。つまり、コンテナは、特定のメソッドを実行する許可について別のロールをチェックせず、かわりに、指定されたセキュリティ識別情報を使用してすべての Enterprise Bean メソッドを実行します。セキュリティ識別情報として、特定のロールまたはコール元の識別情報を指定できます。

EJB 3.0 アプリケーションでは、アノテーションを使用できます（22-8 ページの「アノテーションの使用方法」を参照）。

EJB 3.0 または EJB 2.1 アプリケーションでは、`ejb-jar.xml` デプロイメント・ディスクリプタを使用できます（22-8 ページの「デプロイ XML の使用方法」を参照）。

アノテーションの使用方法

EJB 3.0 アプリケーションでは、例 22-3 に示すように `@RunAs` アノテーションを使用して、Java EE コンテナでの実行中にアプリケーションのロールを指定できます。

例 22-3 @RunAs

```
@RunAs("Admin")
public class Calculator {
    ...
}
```

このアノテーションは、クラスに適用できます。

セキュリティ・アノテーションの詳細は、22-13 ページの「EJB 3.0 セキュリティ・アノテーションの使用方法」を参照してください。

デプロイ XML の使用方法

`runAs` セキュリティ識別情報は、`<enterprise-beans>` セクションの `<security-identity>` 要素で指定します。次の XML は、すべてのエンティティ Bean メソッドが `POMgr` というロールを使用して実行されることを示します。

```
<enterprise-beans>
  <entity>
  ...
    <security-identity>
      <run-as>
        <role-name>POMgr</role-name>
      </run-as>
    </security-identity>
...
  </entity>
</enterprise-beans>
```

また、次の XML の例は、コール元の識別情報を使用して Bean のすべてのメソッドを実行するように指定する方法を示します。

```
<enterprise-beans>
  <entity>
  ...
    <security-identity>
      <use-caller-identity/>
    </security-identity>
...
  </entity>
</enterprise-beans>
```

ユーザーおよびグループへの論理ロールのマッピング

論理ロールまたは実際のユーザーとグループは、EJB デプロイメント・ディスクリプタで使用できます。ただし、論理ロールを使用する場合は、OracleAS JAAS Provider または XML ユーザー・マネージャのいずれかで定義した実際のユーザーとグループに、論理ロールをマッピングする必要があります。

アプリケーションのデプロイメント・ディスクリプタで定義した論理ロールを OracleAS JAAS Provider または XML ユーザー・マネージャのユーザーまたはグループにマッピングするには、OC4J 固有のデプロイメント・ディスクリプタで、次のように `<security-role-mapping>` 要素を使用します。

- この要素の `name` 属性では、マッピングされる論理ロールを定義します。
- `group` または `user` 要素では、論理ロールをグループまたはユーザー名にマッピングします。このグループまたはユーザーは、OracleAS JAAS Provider または XML ユーザー・マネージャ構成で定義する必要があります。OracleAS JAAS Provider および XML ユーザー・マネージャの説明は、『Oracle Containers for J2EE サービス・ガイド』を参照してください。

例 22-4 論理ロールの実際のロールへのマッピング

この例では、論理ロール `POMGR` を、`orion-ejb-jar.xml` ファイル内の `managers` グループにマッピングします。このグループの一部としてログイン可能なユーザーは、すべて `POMGR` ロールを所有しているとみなされます。したがって、`PurchaseOrderBean` のメソッドを実行可能です。

```
<security-role-mapping name="POMGR">
 <group name="managers" />
</security-role-mapping>
```

注意： 論理ロールは、1 つのグループにマッピングすることも、複数のグループにマッピングすることも可能です。

このロールを特定のユーザーにマッピングするには、次のようにします。

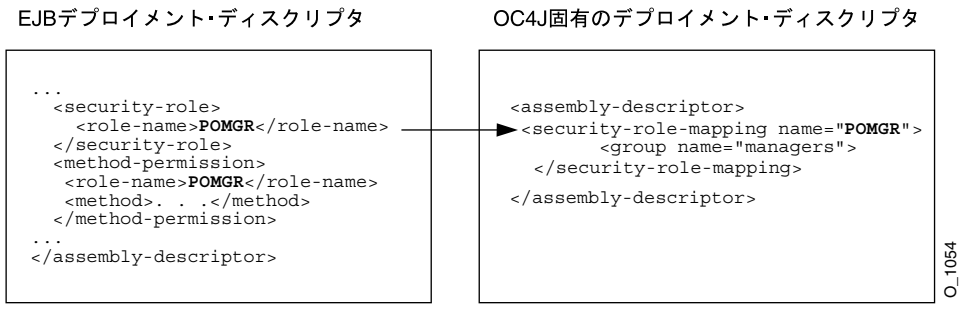
```
<security-role-mapping name="POMGR">
 <user name="guest" />
</security-role-mapping>
```

最後に、次のように、ロールを特定のグループ内の特定のユーザーにマッピングすることも可能です。

```
<security-role-mapping name="POMGR">
 <group name="managers" />
 <user name="guest" />
</security-role-mapping>
```

図 22-3 に示すように、EJB デプロイメント・ディスクリプタで定義されている `POMGR` の論理ロール名は、OC4J 固有のデプロイメント・ディスクリプタ内の `<security-role-mapping>` 要素で `managers` にマッピングされています。

図 22-3 セキュリティのマッピング



EJB デプロイメント・ディスクリプタ内の <role-name> は、OC4J 固有のデプロイメント・ディスクリプタ内の <security-role-mapping> 要素内の name 属性と同じです。これによりマッピングが識別されます。

未定義メソッドに対するデフォルト・ロール・マッピングの指定

メソッドがロール・マッピングに関連付けられていない場合、そのメソッドは、orion-ejb-jar.xml ファイルの <default-method-access> 要素を介してデフォルト・セキュリティ・ロールにマッピングされます。次に、保護されていないメソッドの自動マッピングを示します。

```
<default-method-access>
    <security-role-mapping
        name="&lt;default-ejb-caller-role&gt;"
        impliesAll="true"
    >
    </security-role-mapping>
</default-method-access>
```

デフォルト・ロールは <default-ejb-caller-role> で、name 属性で定義されます。この文字列は、デフォルト・ロールの名前に置換できます。

impliesAll 属性は、メソッドに対するセキュリティ・ロールのチェックが実行されるかどうかを示します。orion-ejb-jar.xml ファイルで、impliesAll 属性には次のデフォルトが割り当てられます。

- <security-role-mapping> が orion-ejb-jar.xml ファイルで指定されており、impliesAll が設定されていない場合、この属性のデフォルトは false に設定されます。コンテナにより、各メソッドでこのデフォルト・ロールがチェックされます。
- <security-role-mapping> が orion-ejb-jar.xml ファイルで指定されていない場合、OC4J EJB レイヤーにより、この属性のデフォルトは true に設定されます。各メソッドでは、セキュリティ・ロールのチェックは実行されません。

impliesAll 属性が false の場合は、<user> 要素および <group> 要素を使用して、name 属性で定義したデフォルト・ロールを OracleAS JAAS Provider または XML のユーザーまたはグループにマッピングする必要があります。次の例では、メソッド許可に関連付けられていないすべてのメソッドを others グループにマッピングする方法を示します。

```
<default-method-access>
    <security-role-mapping name="default-role" impliesAll="false" >
        <group name="others" />
    </security-role-mapping>
</default-method-access>
```

クライアントによるユーザーとグループの指定

クライアントは、ユーザーとグループにより保護されたメソッドにアクセスするために、OracleAS JAAS Provider または XML ユーザー・マネージャが認識できる正確なユーザー名またはグループ名とパスワードを提供する必要があります。また、ユーザーまたはグループは、対象となるメソッドのセキュリティ・ロールで指定されている内容と同じであることが必要です。詳細は、22-11 ページの「EJB クライアントの資格証明の指定」を参照してください。

注意： CSiV2 などの基本的な OC4J セキュリティ構成の詳細は、『Oracle Containers for J2EE セキュリティ・ガイド』を参照してください。

EJB クライアントの資格証明の指定

クライアントのタイプに応じて、クライアントが Enterprise Bean、または JNDI でアクセス可能なその他のリソースにアクセスする前に、セキュリティ資格証明を指定することが必要な場合があります。

表 22-1 に、起動するターゲットの Enterprise Bean を基準としたデプロイの場所により EJB クライアントを分類します。そのターゲット Enterprise Bean を基準としたクライアントのデプロイ場所により、セキュリティ資格証明を指定する必要があるかどうかが決定されます。

表 22-1 クライアントのセキュリティ資格証明の要件

クライアント・タイプ	ターゲット EJB との関連	資格証明の設定
任意のクライアント	クライアントとターゲット Enterprise Bean が同一 JVM 上に置かれます。	×
任意のクライアント	クライアントとターゲット Enterprise Bean は同じアプリケーションにデプロイされます。	×
任意のクライアント	クライアントの親として指定されるアプリケーションにデプロイされるターゲット Enterprise Bean。[1]	×
任意のクライアント	クライアントとターゲット Enterprise Bean は同一 JVM 上に置かれず、同じアプリケーションにデプロイされません。ターゲット EJB アプリケーションはクライアントの親ではありません。[1]	○

[1] アプリケーションの親の設定方法は、『Oracle Containers for J2EE 開発者ガイド』を参照してください。

リモート・コンテナの Enterprise Bean にアクセスする場合（つまり、クライアントおよびターゲット Enterprise Bean が同一 JVM 上に置かれておらず、同じアプリケーションにデプロイされず、ターゲット Enterprise Bean アプリケーションがクライアントの親でない場合）は、リモート・コンテナに有効な資格証明を渡す必要があります。クライアントが資格証明を渡す方法は、クライアントのタイプによって決まります。

- EJB クライアント: リモート Enterprise Bean をルックアップするために作成される資格証明を `InitialContext` 内で渡します（22-12 ページの「初期コンテキストでの資格証明の指定」を参照）。
- スタンドアロン Java クライアント: EAR ファイルとともにデプロイされた `jndi.properties` ファイルで資格証明を定義します（22-12 ページの「JNDI プロパティの資格証明の指定」を参照）。
- サーブレットまたは JSP クライアント: リモート Enterprise Bean をルックアップするために作成される資格証明を `InitialContext` 内で渡します（22-12 ページの「初期コンテキストでの資格証明の指定」を参照）。

また、すべてのクライアントは `ejb_sec.properties` ファイルでセキュリティ・プロパティを指定できます（22-13 ページの「ejb_sec.properties ファイルでの EJB クライアント・セキュリティ・プロパティの指定」を参照）。

詳細は、次を参照してください。

- 29-2 ページの「使用しているクライアントのタイプ」
- 『Oracle Containers for J2EE セキュリティ・ガイド』

JNDI プロパティの資格証明の指定

jndi.properties ファイルで資格証明を指定するには、次のようにします。

1. jndi.properties ファイルを作成または変更します。
2. 例 22-5 に示すように、jndi.properties ファイルに適切な資格証明を構成します。

   プロパティ名については、javax.naming.Context のフィールド定義を参照してください。

例 22-5 JNDI プロパティの資格証明の指定

```
java.naming.security.principal=POMGR
java.naming.security.credentials=welcome
java.naming.factory.initial=
    oracle.j2ee.server.ApplicationClientInitialContextFactory
java.naming.provider.url=opmn:ormi://opmnhost:6004:oc4j_inst1/ejbsamples
```

3. jndi.properties ファイルがクライアントのクラスパスにあることを確認します。
4. 例 22-6 に示すように、クライアントで JNDI API を使用して、JNDI でアクセス可能なリソースをルックアップします。

例 22-6 JNDI でアクセス可能なリソースのルックアップ

```
Context ic = new InitialContext();
CustomerHome = (CustomerHome)ic.lookup("java:comp/env/purchaseOrderBean");
```

実行時に、JNDI は ClassLoader のメソッド getResources を使用して、クラスパス内の jndi.properties という名前のすべてのアプリケーション・リソース・ファイルを特定します。このときに、例 22-6 で設定した JNDI プロパティを使用して、purchaseOrderBean にアクセスします。

詳細は、19-23 ページの「JNDI プロパティ・ファイルでの JNDI プロパティの設定」を参照してください。

初期コンテキストでの資格証明の指定

JNDI でアクセス可能なリソースをルックアップするために使用する資格証明を初期コンテキストで指定するには、次のようにします。

1. 例 22-7 に示すように、HashTable を作成し、javax.naming.Context フィールドをキーとし、String オブジェクトを値として使用して必要なプロパティを移入します。

例 22-7 初期コンテキストでの資格証明の指定

```
Hashtable env = new Hashtable();
env.put(Context.SECURITY_PRINCIPAL, "POMGR");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
env.put("java.naming.factory.initial",
        "oracle.j2ee.server.ApplicationClientInitialContextFactory");
env.put("java.naming.provider.url",
        "opmn:ormi://opmnhost:6004:oc4j_inst1/ejbsamples");
```

2. 初期コンテキストをインスタンス化する場合は、例 22-8 に示すように HashTable を初期コンテキスト・コンストラクタに渡します。

例 22-8 JNDI でアクセス可能なリソースのルックアップ

```
Context ic = new InitialContext (env);
CustomerHome = (CustomerHome) ic.lookup ("java:comp/env/purchaseOrderBean");
```

詳細は、次を参照してください。

- 19-20 ページの「初期コンテキスト・ファクトリの構成」
- 19-24 ページの「初期コンテキストでの JNDI プロパティの設定」

ejb_sec.properties ファイルでの EJB クライアント・セキュリティ・プロパティの指定

サーバーの内部で実行しているかどうかにかかわらず、すべてのクライアントには、`ejb_sec.properties` ファイルで制御される EJB セキュリティ・プロパティがあります。このファイルを使用して、一般的なセキュリティ・オプションおよび Common Secure Interoperability Version 2 プロトコル（CSIv2）に固有のオプションを指定します。

詳細は、『Oracle Containers for J2EE セキュリティ・ガイド』の Common Secure Interoperability プロトコルに関する項を参照してください。

EJB 3.0 セキュリティ・アノテーションの使用方法

EJB 3.0 アプリケーションでは、JSR250 で定義されている `javax.annotation.security` アノテーションを使用して、EJB 3.0 セッション Bean のセキュリティ・オプションを構成できます。

表 22-2 に、OC4J でサポートされるセキュリティ・アノテーションをまとめます。これらのアノテーションの使用例は、22-14 ページの「アノテーションの使用方法」を参照してください。

表 22-2 セキュリティ・アノテーション

アノテーション	説明	適用対象
`@RunAs`	Java EE コンテナでの実行中にアプリケーションのロールを定義します。ロールは、コンテナのセキュリティ・レルムのユーザー / グループ情報にマッピングする必要があります。詳細は、22-8 ページの「runAs セキュリティ識別情報の指定」を参照してください。	クラス
`@RolesAllowed`	アプリケーションでのメソッドへのアクセスを許可されるセキュリティ・ロールを指定します。詳細は、22-5 ページの「EJB メソッドに対するロールの指定」を参照してください。	クラス、メソッドまたはその両方 メソッド指定によりクラス指定がオーバーライドされます（存在する場合）。
`@PermitAll`	すべてのセキュリティ・ロールが指定のメソッドの起動を許可されるように指定します。詳細は、22-6 ページの「EJB メソッドに対するセキュリティ・チェックなしの指定」を参照してください。	クラスまたはメソッド クラス指定はすべてのメソッドに適用されます。 メソッド指定はそのメソッドにのみ適用されます。
`@DenyAll`	指定のメソッドの起動を許可されるセキュリティ・ロールがないことを指定します。	クラスまたはメソッド クラス指定はすべてのメソッドに適用されます。 メソッド指定はそのメソッドにのみ適用されます。
`@DeclareRoles`	アプリケーションで使用されるセキュリティ・ロールを指定します。	クラス

@PermitAll、@DenyAll および @RolesAllowed を使用している場合は、次の制限に注意してください。

- @PermitAll、@DenyAll および @RolesAllowed アノテーションは、同じメソッドまたはクラスに適用できません。
- 次の場合は、メソッド・レベルのアノテーションがクラス・レベルのアノテーションに優先します。
  - @PermitAll がクラス・レベルで指定され、@RolesAllowed または @DenyAll が同じクラスのメソッドに指定されている。
  - @DenyAll がクラス・レベルで指定され、@PermitAll または @RolesAllowed が同じクラスのメソッドに指定されている。
  - @RolesAllowed がクラス・レベルで指定され、@PermitAll または @DenyAll が同じクラスのメソッドに指定されている。

注意： EJB 3.0 セキュリティ・アノテーションのコード例は、http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30security/doc/how-to-ejb30-security-ejb.html からダウンロードできます。

アノテーションの使用方法

例 22-9 に、@RolesAllowed アノテーションの使用方法を示します。詳細および例は、JSR250 の仕様を参照してください。

例 22-9 @RolesAllowed

```
@RolesAllowed("Users")
public class Calculator {

    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
    ...
    }
}
```

JAAS API を使用した Enterprise Bean からの資格証明の取得

OC4J では、標準の JAAS API を使用した、セッション Bean（ステートレスおよびステートフル）およびエンティティ Bean のビジネス・メソッドおよびライフ・サイクル・メソッドからの Subject、Principal および資格証明の取得をサポートしています。

例 22-10 に、JAAS API を使用して、OC4J にデプロイされる Enterprise Bean のビジネス・メソッドで資格証明を取得する方法を示します。

例 22-10 JAAS API を使用した資格証明の取得

```
public class Calculator {
    // Buisness method
    public void setNewRate(int rate) {
    ...
        AccessControlContext actx = AccessController.getContext();
        Subject subject = Subject.getSubject(actx);
        Set principals = subject.getPrincipals();
    ...
    }
}
```

EJB アプリケーションのカスタム JAAS ログイン・モジュールの定義

JAAS Pluggable Authentication フレームワーク内では、アプリケーション・サーバーおよび基礎となる認証サービスが相互に独立しています。アプリケーション・サービスは、アプリケーション・サーバーまたはアプリケーション・コードの変更を必要とせずに、JAAS ログイン・モジュールを通じてプラグインできます。ログイン・モジュールは、提供される資格証明（パスワードなど）の認証を行い、適切なプリンシパル（ロールなど）をサブジェクトに追加します。JAAS ログイン・モジュールの可能なタイプには、プリンシパル・マッピング JAASm モジュール、資格証明マッピング JAAS モジュール、Kerberos JAAS モジュールまたはカスタム・ログイン・モジュールがあります。

カスタム JAAS ログイン・モジュールを Enterprise Bean で使用するには、次の要素を構成する必要があります。

- `system-jazn-data.xml` の `<jazn-loginconfig>`
- `orion-application.xml` の `<jazn>`
- `orion-application.xml` の `<namespace-access>`

詳細は、『Oracle Containers for J2EE セキュリティ・ガイド』の「ログイン・モジュール」を参照してください。

23

メッセージ・サービスの構成

この章では、次のような Java Message Service（JMS）および JMS 以外のメッセージ・サービス・プロバイダを構成する方法を説明します。

- メッセージ・サービス・プロバイダで使用するための J2CA リソース・アダプタの構成
- OEMS JMS メッセージ・サービス・プロバイダの構成
- OEMS JMS データベース・メッセージ・サービス・プロバイダの構成

詳細は、次を参照してください。

- 2-26 ページの「MDB で使用できるメッセージ・サービス・プロバイダ」
- 第 9 章「EJB 3.0 メッセージドリブン Bean の実装」
- 第 17 章「EJB 2.1 メッセージドリブン Bean の実装」
- 『Oracle Containers for J2EE サービス・ガイド』の「Java Message Service（JMS）」

メッセージ・サービス・プロバイダで使用するための J2CA リソース・アダプタの構成

メッセージ・サービス・プロバイダで使用するために Oracle JMS コネクタ（2-26 ページの「Oracle JMS コネクタ : J2EE Connector Architecture（J2CA）ベース・プロバイダ」を参照）などの J2CA リソース・アダプタを構成するには、次の処理を行う必要があります。

1. J2CA アダプタをインストールおよび構成します（23-3 ページの「J2CA アダプタのインストールと構成」を参照）。
2. コネクション・ファクトリに適した JNDI 名を選択します（23-2 ページの「J2CA メッセージ・サービス・プロバイダのコネクション・ファクトリ名」を参照）。
3. 適切なデプロイ XML ファイルを構成します（23-3 ページの「OC4J J2CA リソース・アダプタのデプロイ XML ファイルの構成」を参照）。

   これらのデプロイ XML ファイルを使用して、XA 非準拠のファクトリ（2 フェーズ・コミット（2PC）トランザクションが不要な場合）または XA 準拠のファクトリ（2PC トランザクションが必要な場合）を指定できます。2PC の詳細は、2-24 ページの「グローバル・トランザクションまたは 2 フェーズ・コミット（2PC）トランザクションへの参加方法」を参照してください。
4. J2CA リソース・アダプタを使用してメッセージ・サービス・プロバイダにアクセスするようメッセージドリブン Bean を構成します。

   詳細は、次を参照してください。

   - 10-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」
   - 18-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」

注意 : メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」を参照してください。

注意 : J2CA メッセージ・サービス・プロバイダ・リソース・アダプタおよび MDB アプリケーションの構成の詳細なコード例は、`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/how-to-gjra-with-oracleasjms/doc/how-to-gjra-with-oracleasjms.html` を参照してください。

J2CA メッセージ・サービス・プロバイダのコネクション・ファクトリ名

宛先およびコネクション・ファクトリの実際の JNDI 名は、`oc4j-connectors.xml` ファイルで定義されている J2CA インストール環境によって決まります（23-3 ページの「OC4J J2CA リソース・アダプタのデプロイ XML ファイルの構成」を参照）。

通常、この名前は `java:<Prefix>/<FactoryName>` から構成されています。`<Prefix>` は `comp/env/eis` などのオプションの JNDI の場所で、`<FactoryName>` はアダプタの `javax.cci.ConnectionFactory` の名前です。

J2CA アダプタのインストールと構成

OC4J には、Oracle JMS コネクタが含まれています。このコネクタは、OC4J を OEMS JMS および OEMS JMS データベース・メッセージ・サービス・プロバイダ、また WebSphereMQ、Tibco、SonicMQ などの Oracle 以外の JMS プロバイダと統合する汎用 JMS J2CA リソース・アダプタです。

詳細は、次を参照してください。

- 『Oracle Containers for J2EE リソース・アダプタ管理者ガイド』のリソース・アダプタの管理の概要に関する項
- 2-26 ページの「Oracle JMS コネクタ : J2EE Connector Architecture（J2CA）ベース・プロバイダ」

OC4J J2CA リソース・アダプタのデプロイ XML ファイルの構成

J2CA メッセージ・サービス・プロバイダを構成するには、次のデプロイ XML ファイルを構成する必要があります。

- `ra.xml`
- `oc4j-ra.xml`
- `oc4j-connectors.xml`

これらのデプロイ XML ファイルを使用して、XA 非準拠のファクトリ（2 フェーズ・コミット（2PC）トランザクションが不要な場合）または XA 準拠のファクトリ（2PC トランザクションが必要な場合）を指定できます。2PC の詳細は、2-24 ページの「グローバル・トランザクションまたは 2 フェーズ・コミット（2PC）トランザクションへの参加方法」を参照してください。

詳細は、次を参照してください。

- 『Oracle Containers for J2EE リソース・アダプタ管理者ガイド』のコネクション・ファクトリのバインドと構成の基本設定に関する項
- 『Oracle Containers for J2EE リソース・アダプタ管理者ガイド』の OC4J リソース・アダプタ構成ファイルに関する項

OEMS JMS メッセージ・サービス・プロバイダの構成

OEMS JMS メッセージ・サービス・プロバイダ（2-27 ページの「OEMS JMS: メモリー内またはファイルベース・プロバイダ」を参照）を構成するには、次の処理を行う必要があります。

1. 宛先およびコネクション・ファクトリに適した JNDI 名を選択します（「OEMS JMS 宛先名およびコネクション・ファクトリ名」を参照）。
2. *<OC4J_HOME>*/j2ee/home/config/jms.xml ファイル（「jms.xml の構成」を参照）を構成して、宛先およびコネクション・ファクトリのタイプを指定します。

   XA 非準拠のファクトリ（2 フェーズ・コミット（2PC）トランザクションが不要な場合）または XA 準拠のファクトリ（2PC トランザクションが必要な場合）を指定できます。2PC の詳細は、2-24 ページの「グローバル・トランザクションまたは 2 フェーズ・コミット（2PC）トランザクションへの参加方法」を参照してください。
3. オプションで、実際の JNDI 名を論理名にマッピングします（19-15 ページの「JMS 宛先またはコネクション・リソース・マネージャのコネクション・ファクトリへの環境参照の構成（JMS 1.0）」を参照）。
4. OEMS JMS メッセージ・サービス・プロバイダにアクセスするようメッセージドリブン Bean を構成します。

   詳細は、次を参照してください。

   - 10-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」
   - 10-4 ページの「直接メッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」
   - 18-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」
   - 18-4 ページの「直接メッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」

注意： メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」を参照してください。

OEMS JMS 宛先名およびコネクション・ファクトリ名

JMS 宛先およびコネクション・ファクトリの実際の JNDI 名は、jms.xml ファイル内でユーザーが指定します（23-5 ページの「jms.xml の構成」を参照）。

表 23-1 に、これらの名前の形式をリストします。

表 23-1 OEMS JMS 宛先名およびコネクション・ファクトリ名

タイプ	形式
キュー	jms/Queue/<QName>
キュー・コネクション・ファクトリ	jms/Queue/<QCFName>
トピック	jms/Topic/<TName>
トピック・コネクション・ファクトリ	jms/Topic/<TCFName>

jms.xml の構成

<OC4J_HOME>/j2ee/home/config/jms.xml ファイルで OEMS JMS オプションを構成します。このリリースでは、jms.xml は http://www.oracle.com/technology/oracleas/schema/jms-server-10_1.xsd にある XML Schema 文書（XSD）で定義されています。

jms.xml ファイルで構成できるオプションの一部を次に示します。

- MDB で使用される JMS の Destination オブジェクト。
- クライアントが MDB 宛のすべてのメッセージを送信する jms.xml ファイル内のトピックまたはキュー。
- いずれの Destination タイプの場合も、名前、位置およびコネクション・ファクトリを指定する必要があります。

  XA 非準拠のファクトリ（2 フェーズ・コミット（2PC）トランザクションが不要な場合）または XA 準拠のファクトリ（2PC トランザクションが必要な場合）を指定できます。2PC の詳細は、2-24 ページの「グローバル・トランザクションまたは 2 フェーズ・コミット（2PC）トランザクションへの参加方法」を参照してください。
- 照会などのために MDB でデータベースにアクセスする場合は、使用されるデータソースを構成できます。データソースの構成の詳細は、『Oracle Containers for J2EE サービス・ガイド』の「データソース」の章を参照してください。
- OEMS JMS イベントおよびエラーが記述されるファイルへのパス。

例 23-1 に、メッセージドリブン Bean rpTestMdb（例 17-1 を参照）で使用されるキュー（jms/Queue/rpTestQueue という名前）を指定する EJB 2.1 MDB の jms.xml ファイル構成を示します。キュー・コネクション・ファクトリは、jms/Queue/myQCF で定義されます。また、トピックの名前は jms/Topic/rpTestTopic、コネクション・ファクトリは jms/Topic/myTCF で定義されます。

例 23-1 OEMS JMS ファクトリを使用した EJB 2.1 MDB の jms.xml

```
<jms>
    <jms-server port="9128">
        <queue location="jms/Queue/rpTestQueue"></queue>
        <queue-connection-factory location="jms/Queue/myQCF"></queue-connection-factory>
        <topic location="jms/Topic/rpTestTopic"></topic>
        <topic-connection-factory location="jms/Topic/myTCF"></topic-connection-factory>
        <log>
             <!-- path to the log-file where JMS-events and errors are written -->
             <file path="../log/jms.log" />
        </log>
    </jms-server>
</jms>
```

例 23-2 に、2 フェーズ・コミット（2PC）対応の XA ファクトリを使用した同じ MDB の jms.xml ファイル構成を示します。

例 23-2 OEMS JMS XA ファクトリを使用した EJB 2.1 MDB の jms.xml

```
<jms>
    <jms-server port="9128">
        <queue location="jms/Queue/rpTestQueue"></queue>
        <xa-queue-connection-factory location="jms/Queue/myXAQCF"></queue-connection-factory>
        <topic location="jms/Topic/rpTestTopic"></topic>
        <xa-topic-connection-factory location="jms/Topic/myXATCF"></topic-connection-factory>
        <log>
             <!-- path to the log-file where JMS-events and errors are written -->
             <file path="../log/jms.log" />
        </log>
    </jms-server>
</jms>
```

OEMS JMS データベース・メッセージ・サービス・プロバイダの構成

OEMS JMS データベース・メッセージ・サービス・プロバイダ（2-28 ページの「OEMS JMS データベース : アドバンスト・キューイング（AQ）ベース・プロバイダ」を参照）を構成するには、次の処理を行う必要があります。

1. OEMS JMS データベース・プロバイダをインストールおよび構成します（23-7 ページの「OEMS JMS データベース・プロバイダのインストールと構成」を参照）。

   XA 準拠リソースを無効にする権限（2 フェーズ・コミット（2PC）トランザクションが不要な場合）または XA 準拠リソースを有効にする権限（2PC トランザクションが必要な場合）を付与できます。2PC の詳細は、2-24 ページの「グローバル・トランザクションまたは 2 フェーズ・コミット（2PC）トランザクションへの参加方法」を参照してください。

2. 宛先およびコネクション・ファクトリに適した JNDI 名を選択します（23-7 ページの「OEMS JMS データベース宛先名およびコネクション・ファクトリ名」を参照）。

3. `data-sources.xml` ファイルを構成してデータベースを識別します（23-9 ページの「data-sources.xml の構成」を参照）。

4. オプションで、実際の JNDI 名を論理名にマッピングします（19-15 ページの「JMS 宛先またはコネクション・リソース・マネージャのコネクション・ファクトリへの環境参照の構成（JMS 1.0）」を参照）。

5. `application.xml`（または `orion-application.xml`）ファイルを構成して、`<resource-provider>` 要素内で OEMS JMS データベース・プロバイダとして使用するデータソースの JNDI 名を識別します（23-9 ページの「application.xml または orion-application.xml の構成」を参照）。

6. OEMS JMS データベース・メッセージ・サービス・プロバイダにアクセスするようメッセージドリブン Bean を構成します。

   詳細は、次を参照してください。

   - 10-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」
   - 10-4 ページの「直接メッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」
   - 18-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」
   - 18-4 ページの「直接メッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」

注意 : メッセージ・サービス・プロバイダには、Oracle JMS コネクタなどの J2CA リソース・アダプタを使用してアクセスすることをお薦めします。詳細は、2-30 ページの「J2CA リソース・アダプタを使用せずにメッセージ・サービス・プロバイダにアクセスする場合の制限」を参照してください。

OEMS JMS データベース宛先名およびコネクション・ファクトリ名

表 23-2 に示すように、JMS 宛先およびコネクション・ファクトリの実際の JNDI 名は、OEMS JMS データベースのインストール環境によって決まります。

表 23-2 OEMS JMS データベース宛先名およびコネクション・ファクトリ名

タイプ	形式
キュー	`java:comp/resource/<ProviderName>/Queues/<QName>`
キュー・コネクション・ファクトリ	`java:comp/resource/<ProviderName>/QueueConnectionFactories/<QCFName>`
トピック	`java:comp/resource/<ProviderName>/Topics/<TName>`
トピック・コネクション・ファクトリ	`java:comp/resource/<ProivderName>/TopicConnectionFactories/<TCFName>`

表 23-2 の変数の値は、次のように定義されます。

- `<ProviderName>`: OEMS JMS データベース・サービスを提供しているデータソースの JNDI 名（23-9 ページの「application.xml または orion-application.xml の構成」を参照）。
- `<QName>`: データベースで作成したキューの名前（23-7 ページの「OEMS JMS データベース・プロバイダのインストールと構成」の手順 3b を参照）。
- `<QCFName>`: キュー・コネクション・ファクトリの名前。任意の名前を指定できます。
- `<TName>`: データベースで作成したトピックの名前（23-7 ページの「OEMS JMS データベース・プロバイダのインストールと構成」の手順 3b を参照）。
- `<TCFName>`: トピック・コネクション・ファクトリの名前。任意の名前を指定できます。

OEMS JMS データベース・プロバイダのインストールと構成

注意： 次の各項では、キュー、トピック、それらの表の作成および権限の割当てに SQL を使用します。この SQL は、OC4J のサンプル・コードのページ `http://www.oracle.com/technology/tech/java/oc4j/demos` にある MDB デモ内で提供されています。

---

1. ユーザーまたは DBA は、『Oracle Streams アドバンスト・キューイング・ユーザーズ・ガイドおよびリファレンス』および汎用データベース・マニュアルに従って Oracle AQ をインストールする必要があります。
2. ユーザーまたは DBA は、MDB でデータベースに接続するための RDBMS ユーザーを作成し、OEMS JMS データベース操作を実行するための適切なアクセス権限をこのユーザーに付与する必要があります。

   必要な権限は、リクエストする機能によって決まります。各タイプの機能に必要な権限の詳細は、『Oracle Streams アドバンスト・キューイング・ユーザーズ・ガイドおよびリファレンス』を参照してください。

   次の例では、Oracle AQ 操作に必要な権限のある `jmsuser` を作成します。このユーザーは、そのスキーマ内で作成する必要があります。これらの文を実行するには `SYS DBA` であることが必要です。

```
DROP USER jmsuser CASCADE ;

GRANT connect, resource,AQ_ADMINISTRATOR_ROLE TO jmsuser IDENTIFIED BY jmsuser ;
GRANT execute ON sys.dbms_aqadm  TO  jmsuser;
GRANT execute ON sys.dbms_aq     TO  jmsuser;
GRANT execute ON sys.dbms_aqin   TO  jmsuser;
GRANT execute ON sys.dbms_aqjms  TO  jmsuser;
```

```
connect jmsuser/jmsuser;
```

ユーザーの必要に応じて、XA 準拠の 2 フェーズ・コミット（2PC）権限やシステム管理権限など、他の権限の付与が必要な場合があります。

2PC の詳細は、次を参照してください。

- 2-24 ページの「グローバル・トランザクションまたは 2 フェーズ・コミット（2PC）トランザクションへの参加方法」
- 『Oracle Containers for J2EE サービス・ガイド』の JTA に関する章

3. ユーザーまたは DBA は、JMS の `Destination` オブジェクトをサポートする表およびキューを作成する必要があります。

`DBMS_AQADM` パッケージおよび Oracle AQ メッセージ・タイプの詳細は、『Oracle Streams アドバンスト・キューイング・ユーザーズ・ガイドおよびリファレンス』を参照してください。

a. JMS の `Destination`（キューまたはトピック）を処理する表を作成します。

OEMS JMS データベースでは、トピックとキューの両方でキュー表が使用されます。`rpTestMdb` JMS の例では、キュー用に単一の表 `rpTestQTab` が作成されます。

キュー表を作成するには、次の SQL を実行します。

```
DBMS_AQADM.CREATE_QUEUE_TABLE(
        Queue_table             => 'rpTestQTab',
        Queue_payload_type      => 'SYS.AQ$_JMS_MESSAGE',
        sort_list => 'PRIORITY,ENQ_TIME',
        multiple_consumers  => false,
        compatible              => '8.1.5');
```

`multiple_consumers` パラメータは、複数のコンシューマが存在するかどうかを示します。したがって、このパラメータは常に、キューについては `false`、トピックについては `true` に設定します。

b. JMS の `Destination` を作成します。トピックを作成する場合は、トピックの各サブスクライバを追加する必要があります。`rpTestMdb` JMS の例では、単一のキュー `rpTestQueue` が必要です。

次の例では、キュー表 `rpTestQTab` 内に `rpTestQueue` というキューを作成します。作成後にキューを開始します。

```
DBMS_AQADM.CREATE_QUEUE(
      Queue_name          => 'rpTestQueue',
      Queue_table         => 'rpTestQTab');

DBMS_AQADM.START_QUEUE(
      queue_name         => 'rpTestQueue');
```

トピックを追加する場合のために、次の例で、トピック表 `rpTestTTab` 内に `rpTestTopic` というトピックを作成する方法を示します。作成後に、2 つの永続サブスクライバがトピックに追加されます。最後に、トピックが開始され、ユーザーにそのトピックに関する権限が付与されます。

注意： Oracle AQ では、`DBMS_AQADM.CREATE_QUEUE` メソッドを使用してキューとトピックの両方が作成されます。

```
DBMS_AQADM.CREATE_QUEUE_TABLE(
        Queue_table             => 'rpTestTTab',
        Queue_payload_type      => 'SYS.AQ$_JMS_MESSAGE',
        multiple_consumers  => true,
        compatible              => '8.1.5');
DBMS_AQADM.CREATE_QUEUE( 'rpTestTopic', 'rpTestTTab');
```

```
DBMS_AQADM.ADD_SUBSCRIBER('rpTestTopic',
                          sys.aq$_agent('MDSUB', null, null));
DBMS_AQADM.ADD_SUBSCRIBER('rpTestTopic',
                          sys.aq$_agent('MDSUB2', null, null));
DBMS_AQADM.START_QUEUE('rpTestTopic');
```

注意： ここで定義する名前は、orion-ejb-jar.xml ファイルでキューまたはトピックを定義するために使用した名前と同じ名前にする必要があります。

data-sources.xml の構成

OEMS JMS データベース・プロバイダがインストールされているデータベースに対してデータソースを構成します。JMS のトピックおよびキューは、データベース表とキューを使用してメッセージ機能を提供します。使用するデータソースのタイプは、必要な機能によって決まります。

例 23-3 に、デフォルトでグローバル（2 フェーズ・コミット）トランザクションをサポートする一般的なマネージド・データソースを示します。

例 23-3 シン JDBC ドライバを使用するエミュレートされたデータソース

```
<connection-pool name="ScottConnectionPool">
    <connection-factory
        factory-class="oracle.jdbc.pool.OracleDataSource"
        user="scott"
        password="tiger"
        url="jdbc:oracle:thin:@//localhost:1521/ORCL" >
    </connection-factory>
</connection-pool>

<managed-data-source
    name="OracleDS"
    jndi-name="jdbc/OracleDS"
    connection-pool-name="ScottConnectionPool"
/>
```

詳細は、2-17 ページの「EJB データソース・サービスについて」を参照してください。

application.xml または orion-application.xml の構成

<resource-provider> 要素内で、OEMS JMS データベース・プロバイダとして使用されるデータソースの JNDI 名を識別します。

- すべてのアプリケーション用（グローバル）の JMS プロバイダの場合は、グローバルな application.xml ファイルを構成します。
- 1 つのアプリケーション用（ローカル）の JMS プロバイダの場合は、アプリケーションの orion-application.xml ファイルを構成します。

次のコード例は、OEMS JMS データベースの XML 構文を使用して JMS プロバイダを構成する方法を示します。

- class 属性：OEMS JMS データベース・プロバイダは、class 属性で構成される oracle.jms.OjmsContext クラスによって実装されます。
- property 属性：property 要素内で、この JMS プロバイダとして使用されるデータソースを識別します。トピックまたはキューは、このデータソースに接続して、メッセージ機能を提供する表とキューにアクセスします。

次の例では、jdbc/OracleDS によって識別されるデータソースが、OEMS JMS データベース・プロバイダとして使用されるデータソースです。この JNDI 名は、例 23-3 の managed-data-source 要素の jndi-name 属性で指定されます。この例でエミュレートされていないデータソースを使用した場合、名前は location 要素内の名前と同じになります。

```
<resource-provider
    class="oracle.jms.OjmsContext"
    name="myProvider">
    <description>OJMS/AQ</description>
    <property name="datasource" value="jdbc/OracleDS"></property>
</resource-provider>
```

24

OC4J EJB アプリケーション・クラスタリング・サービスの構成

この章では、EJB アプリケーションについて構成できる次のような OC4J アプリケーション・クラスタリング・オプションについて説明します。

- EJB 3.0 および EJB 2.1 ステートフル・セッション Bean レプリケーション・ポリシーの構成
- 静的検出ロード・バランシングの構成
- DNS ロード・バランシングの構成
- ロード・バランシングの動作の構成

詳細は、2-35 ページの「OC4J EJB アプリケーション・クラスタリング・サービスについて」を参照してください。

EJB 3.0 および EJB 2.1 ステートフル・セッション Bean レプリケーション・ポリシーの構成

EJB 3.0 または EJB 2.1 ステートフル・セッション Bean の EJB アプリケーション・クラスタリングを構成する一般的な手順は、次のとおりです。

1. OC4J アプリケーション・クラスタを構成します（『Oracle Containers for J2EE 構成および管理ガイド』の「OC4J でのアプリケーションのクラスタリング」の章を参照）。
2. 各ノードのステートフル・セッション Bean のレプリケーション・ポリシーを構成します（24-2 ページの「デプロイ XML の使用方法」を参照）。
3. ロード・バランシングの動作を構成します（24-5 ページの「ロード・バランシングの動作の構成」を参照）。
4. Enterprise Bean をクラスタ内のいずれかのノードにデプロイします。

詳細は、次を参照してください。

- 2-36 ページの「状態レプリケーション」
- 『Oracle Application Server 高可用性ガイド』の Oracle Application Server クラスタ（OC4J）でのステートフル・セッション EJB の状態レプリケーションに関する項

デプロイ XML の使用方法

レプリケーション・ポリシーを構成するには、表 24-1 にリストする 1 つ以上の適切なデプロイメント・ディスクリプタ・ファイルに `<replication-policy>` 要素を追加します。OC4J がグローバルに適用する 1 つのレプリケーション・ポリシーを指定するか、Web コンポーネントと EJB コンポーネントの両方または EJB コンポーネントのみに対してファイングレインなレプリケーション・ポリシーをアプリケーション・レベルで指定できます。

`trigger` 属性を次のいずれかに構成します。

- `inherited`: ステートフル・セッション Bean は、アプリケーション・レベルで構成する状態レプリケーション・トリガー設定を使用します。これはデフォルト値です。
- `onRequestEnd`: ステートフル・セッション Bean の状態は、各 EJB メソッドのコールの終了時に、クラスタ内の（マルチキャスト・アドレス、ポートが同一の）すべてのホストにレプリケートされます。ノードの電源が切断された場合でも、状態はすでにレプリケートされています。この方法は、状態の送信回数が多くなるため、JVM 終了レプリケーション・モードよりパフォーマンスが低下します。ただし、信頼性の保証は高くなります。
- `onShutdown`: ステートフル・セッション Bean は、JVM が終了すると、クラスタ内の（マルチキャスト・アドレス、ポートが同一の）別のホストの 1 つにのみレプリケートされます。このオプションは、状態のレプリケートが 1 回のみであるため、最高のパフォーマンスを得ることができます。ただし、次の理由により、信頼性は高くありません。
  - 予期しないときにホストが終了した場合、状態はレプリケートされません。
  - Bean の状態は常に 1 つのホストにのみ存在するため、状態がレプリケートされずに失われる危険性が高くなります。
- `none`: このステートフル・セッション Bean のクラスタリングが無効化されます。

注意： Application Server Control を使用して `trigger` 属性を `inherited` または `none` に構成することはできません。これらの値を設定するには、デプロイ XML を手動で編集します。詳細は、31-2 ページの「Oracle Enterprise Manager 10g Application Server Control の使用方法」を参照してください。

ステートフル・セッション Bean の場合、`scope` 属性は常に `allAttributes` に設定されます。

詳細は、2-36 ページの「状態レプリケーション」を参照してください。

表 24-1 レプリケーション・ポリシー構成のデプロイ XML ファイル

有効範囲	コンポーネント	デプロイ XML ファイル	参照先
グローバル	Web および EJB	`application.xml`	『Oracle Application Server 高可用性ガイド』の Oracle Application Server クラスタ（OC4J）でのステートフル・セッション EJB の状態レプリケーションに関する項
アプリケーションレベル	Web および EJB	`orion-application.xml`	『Oracle Application Server 高可用性ガイド』の Oracle Application Server クラスタ（OC4J）でのステートフル・セッション EJB の状態レプリケーションに関する項
アプリケーションレベル	EJB	`orion-ejb-jar.xml`	24-3 ページの「EJB コンポーネントの orion-ejb-jar.xml ファイルのアプリケーションレベル・レプリケーション・ポリシーのオーバーライド」

EJB コンポーネントの orion-ejb-jar.xml ファイルのアプリケーションレベル・レプリケーション・ポリシーのオーバーライド

`orion-ejb-jar.xml` ファイルをステートフル・セッション Bean の状態レプリケーション・ポリシー（例 24-1 を参照）で構成する場合、各 Bean では、Web コンポーネント・レプリケーション・タイプに関係なく、異なるタイプのレプリケーションを使用できます。

例 24-1 EJB のアプリケーションレベル・レプリケーション・ポリシーの orion-ejb-jar.xml

```
<orion-ejb-jar>
...
    <session-deployment
        name="AirlinePOEndpointBean"
        max-tx-retries="0"
        location="AirlinePOEndpointBean"
        persistence-filename="AirlinePOEndpointBean">
...
        <replication-policy
            trigger="onRequestEnd"
            scope="allAttributes"
        />
...
    </session-deployment>
...
</orion-ejb-jar>
```

静的検出ロード・バランシングの構成

ロード・バランシングのために OC4J インスタンスの静的検出を使用するには、次のようにします。

1. 各クライアント内で、JNDI プロパティを次のように構成します（24-4 ページの「JNDI プロパティの使用方法」を参照）。
   - `java.naming.factory.initial` では、初期コンテキスト・ファクトリを使用します。
   - `java.naming.provider.url` では、`<prefix>://<hostname>:<port>/<application-name>` の形式で OC4J ノードのカンマ区切りリストを指定します。ここで、`<prefix>` は、`opmn:ormi`（Oracle Application Server の OC4J の場合）または `ormi`（スタンドアロン OC4J の場合）になります。
2. ロード・バランシングの動作を構成します（24-5 ページの「ロード・バランシングの動作の構成」を参照）。

詳細は、次を参照してください。

- 2-36 ページの「ロード・バランシング」
- 19-22 ページの「OC4J および Oracle Application Server のネーミング・プロバイダの構成」
- 19-22 ページの「OC4J スタンドアロンのネーミング・プロバイダ URL の構成」

JNDI プロパティの使用方法

例 24-2 に、ロード・バランシングを使用するために、3 つの OC4J ノード（それぞれホスト名が `s1`、`s2` および `s3` で、ポートが 23791、23792 および 23793）があるクライアント・コンテナを提供する URL 定義を示します。

例 24-2 静的検出ロード・バランシングの JNDI プロパティ

```
java.naming.factory.initial= oracle.j2ee.rmi.RMIInitialContextFactory
java.naming.provider.url=ormi://s1:23791/ejbs, ormi://s2:23792/ejbs, ormi://s3:23793/ejbs;
java.naming.security.principal=admin
java.naming.security.credentials=welcome
```

DNS ロード・バランシングの構成

DNS ロード・バランシングを使用するには、次のようにします。

1. DNS 内で、1 つのホスト名を複数の IP アドレスにマッピングします。各ポート番号は、それぞれの IP アドレスに対して同じであることが必要です。DNS サーバーは、ラウンドロビン法またはランダムでアドレスを返すように設定します。

   IP アドレスで OC4J の実行を識別します。ポート番号は RMI ポート番号です。

2. クライアントでの DNS のキャッシュをオフにします。UNIX マシンの場合は、次の手順で DNS のキャッシュをオフにする必要があります。

   a. クライアントでの NSCD デーモン・プロセスを停止します。

   b. `-Dsun.net.inetaddr.ttl=0` オプションを使用して、OC4J クライアントを起動します。

3. 各クライアント内で、JNDI プロパティを次のように構成します（24-5 ページの「JNDI プロパティの使用方法」を参照）。

   - `java.naming.factory.initial` では、初期コンテキスト・ファクトリを使用します。
   - `java.naming.provider.url` では、`<prefix>://<hostname>:<port>/<application-name>` の形式で OC4J の IP アドレスがマッピングされる単一のホスト名と共通 RMI ポートを指定します。ここで、`<prefix>` は、`opmn:ormi`（Oracle Application Server の OC4J の場合）または `ormi`（スタンドアロン OC4J の場合）になります。

4. ロード・バランシングの動作を構成します（24-5 ページの「ロード・バランシングの動作の構成」を参照）。

DNS サーバーでルックアップが発生するたびに、DNS サーバーは、マップされている IP アドレスの 1 つを返します。

詳細は、次を参照してください。

- 2-36 ページの「ロード・バランシング」
- 19-22 ページの「OC4J および Oracle Application Server のネーミング・プロバイダの構成」
- 19-22 ページの「OC4J スタンドアロンのネーミング・プロバイダ URL の構成」

JNDI プロパティの使用方法

例 24-3 では、初期コンテキスト・ファクトリは RMIInitialContextFactory（ただし、DNS ロード・バランシングの任意の初期コンテキスト・ファクトリを使用できる）、myserver はサーバーのリスト用に DNS サーバーに設定されているホスト名、RMI ポートはデフォルトのポートです。

例 24-3 DNS ロード・バランシングの JNDI プロパティ

```
java.naming.factory.initial= oracle.j2ee.rmi.RMIInitialContextFactory
java.naming.provider.url=ormi://myserver/applname
java.naming.security.principal=admin
java.naming.security.credentials=welcome
```

ロード・バランシングの動作の構成

EJB 3.0 と EJB 2.1 の両方、およびすべてのロード・バランシング計画（レプリケーションベース、静的検出または DNS）において、クライアントのリクエストをクラスタ内の OC4J インスタンス間でロード・バランシングする方法を構成できます（24-5 ページの「システム・プロパティの使用方法」を参照）。

詳細は、2-36 ページの「ロード・バランシング」を参照してください。

システム・プロパティの使用方法

このリリースでは、oracle.j2ee.rmi.loadBalance システム・プロパティを構成して、アプリケーション・クラスタ内でのロード・バランシングを指定します。

このシステム・プロパティの値は次のいずれかになります。

- client: クライアントは、変換全体の最初のルックアップで最初に選択された OC4J プロセスと対話します（デフォルト）。
- context: クライアントは、別のコンテキストが使用されている場合に新しいサーバーに移動します（非推奨の dedicated.rmicontext と同様）。
- lookup: クライアントは、リクエストのたびに新しい（ランダムに選択された）サーバーに移動します。

このシステム・プロパティは、OC4J のコマンドラインで -D 引数として、または環境参照（19-17 ページの「環境変数への環境参照の構成」を参照）として構成できます。このシステム・プロパティは、すべてのクライアントに適用されます。

25

タイマー・サービスの構成

この章の内容は次のとおりです。

注意： EJB タイマーは、単一の JVM で稼働する OC4J インスタンスでのみサポートされます（`opmn.xml` 構成ファイルの `<process-set>` 要素で、numprocs=1）。

EJB タイマーのコード例は、`http://www.oracle.com/technology/tech/java/oc4j/demos` および `http://www.oracle.com/technology/tech/java/oc4j/1003/how_to/how-to-ejb-timer.html` からダウンロードできます。

詳細は、2-37 ページの「EJB タイマー・サービスについて」を参照してください。

Java EE タイマーを使用する Enterprise Bean の構成

Java EE タイマーを使用する次のタイプの Enterprise Bean を構成できます。

- EJB 3.0 のステートレス・セッション Bean およびメッセージドリブン Bean
- EJB 2.1 のステートレス・セッション Bean、コンテナ管理の永続性を備えたエンティティ Bean、Bean 管理の永続性を備えたエンティティ Bean およびメッセージドリブン Bean

Java EE タイマーを使用する Enterprise Bean を構成するには、次のようにします。

1. 次のいずれかの方法で `javax.ejb.TimerService` を取得します。
   - EJB 3.0 Enterprise Bean では、例 25-1 に示すように `@Resource` アノテーションを使用します。
   - EJB 3.0 または EJB 2.1 Enterprise Bean では、例 25-1 に示すように `EJBContext` または `InitalContext` のメソッド `getTimerService` を使用します。
2. 例 25-1 および例 25-1 に示すように、`TimerService` のメソッド `createTimer` を使用して適切なタイプのタイマーを作成します（`javax.ejb.TimerService` API を参照）。

   EJB 2.1 エンティティ Bean で `Timer` を作成すると、その主キーで識別される特定のエンティティ Bean インスタンスのタイムアウト・コールバック・メソッドがコンテナにより起動されます。特定のエンティティ Bean に対して作成されたタイマーは、そのエンティティ Bean の削除時に削除されます。

   他の任意のタイプの Enterprise Bean で `Timer` を作成すると、プール状態にあるそのタイプのインスタンスのタイムアウト・コールバック・メソッドがコンテナにより起動されます。
3. タイムアウト・コールバック・メソッドを実装します。

   このメソッドは、`static` または `final` にすることはできません。また、このメソッドは次のシグネチャを持つ必要があります。

   ```
   void <METHOD>(Timer timer)
   ```

   タイムアウト・コールバック・メソッドは、次のいずれかの方法で実装できます。
   - EJB 3.0 Enterprise Bean では、例 25-1 に示すように任意の Bean メソッドに `@Timeout` アノテーションを付けます。
   - EJB 3.0 または EJB 2.1 Enterprise Bean では、例 25-1 に示すように `javax.ejb.TimedObject` インタフェースを実装します。

例 25-1 EJB 3.0 ステートレス・セッション Bean での Java EE タイマーの構成

```
import javax.ejb.Stateless;
import javax.ejb.TimerService;
import javax.ejb.Timeout;

import javax.ejb.Timer;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;

@Stateless;
public class TimerServiceBean implement MyTimerService {
    // injection of TimerService
    @Resource TimerService timerService;

    // implement bean business interface MyTimerService
    @TransactionAttribute(value=TransactionAttributeType.REQUIRES_NEW)
    // default TransactionAttributeType.REQUIRED
    public void createTimer(Serializable timerInfo) {
        timerService.createTimer(timeout, info);
    }

    ...
```

```
    // user annotated timeout method
    @Timeout
    @TransactionAttribute(value=TransactionAttributeType.REQUIRES_NEW)
    public void timeoutCallback(Timer timer) {
        ...
    }
}
```

例 25-2 EJB 2.1 ステートレス・セッション Bean での Java EE タイマーの構成

```
import java.io.Serializable;
import java.rmi.RemoteException;

import javax.ejb.EJBContext;
import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.TimedObject;
import javax.ejb.Timer;
import javax.ejb.TimerService;

class ServiceBean_2_1 implements SessionBean, MyTimerService, TimedObject {
    EJBContext ctx;

    // implement bean business interface MyTimerService
    public void createTimer(long duration, Serializable info) {
        TimerService timerService = ctx.getTimerService();
        timerService.createTimer(duration, info);
    }

    // implement TimedObject
    public void ejbTimeout(Timer timer) {
        System.out.println("Timeout: " + timer.getInfo());
    }

    ...
    // implement SessionBean
    public void setSessionContext(SessionContext ctx) throws EJBException,
                    RemoteException {
        this.ctx = ctx;
    }
}
```

OC4J cron タイマーを使用する Enterprise Bean の構成

OC4J cron タイマーは、次の Bean とともに使用できます。

- EJB 3.0 ステートレス・セッション Bean およびメッセージドリブン Bean
- 任意のタイプの EJB 2.1 Enterprise Bean

指定された間隔で定期的に実行されるようにタイマーをスケジュールすることができます。UNIX では、cron タイマーと呼ばれています。

例 25-3 に、cron タイマーの様々なスケジュールの設定方法を示します。アスタリスクがある場合、すべての値が有効です。

例 25-3 様々な cron タイマーの構成方法

```
20  * * * * --> 20 minutes after every hour, such as 00:20, 01:20, and so on
 5 22 * * * --> Every day at 10:05 P.M.
 0  8 1 * * --> First day of every month at 8:00 A.M.
 0  8 4 7 * --> The fourth of July at 8:00 A.M.
15 12 * * 5 --> Every Friday at 12:15 P.M.
```

cron の time 変数のフォーマットでは、次の 5 つの時間フィールドを使用します。

- 分 : 0 ～ 59
- 時間 : 0 ～ 23
- 日 : 1 ～ 31
- 月 : 1 ～ 12。または、Jan、Feb、Mar、Apr、May、Jun、Jul、Aug、Sep、Oct、Nov、Dec の文字列で指定します。
- 曜日 : 0 ～ 7。または、Sun、Mon、Tue、Wed、Thu、Fri、Sat の文字列で指定します。0 と 7 は両方とも日曜日を示します。

1 つのフィールド内で複数の値をカンマまたはダッシュで区切って、例 25-4 に示すように複雑なタイマーを定義できます。

例 25-4 複雑な cron タイマー

```
0  8   *   * 1,3,5 --> Every Monday, Wednesday, and Friday at 8:00 A.M.
0  8  1,15 *  *   --> The first and 15th of every month at 8:00 A.M.
0 8-17 *   * 1-5  --> Every hour from 8 A.M. through 5 P.M., Monday through Friday
```

OC4J cron タイマーを使用する Enterprise Bean を構成するには、次のようにします。

1. 次のいずれかの方法で `oracle.j2ee.ejb.timer.EJBTimerService` を取得します。
   - EJB 3.0 Enterprise Bean では、例 25-5 に示すように `@Resource` アノテーションを使用します。
   - EJB 3.0 または EJB 2.1 Enterprise Bean では、例 25-6 に示すように `EJBContext` または `InitalContext` のメソッド `getTimerService` を使用します。
2. 例 25-5 および例 25-6 に示すように、`EJBTimerService` のメソッド `createTimer` を使用して適切なタイプのタイマーを作成します。

   次に示す `EJBTimerService` の任意のメソッドを使用できます。これらのメソッドは、すべて `javax.ejb.Timer` 型のオブジェクトを返し、`IllegalArgumentException` および `IllegalStateException` をスローします。

- `createTimer(String cronline, Serializable info)`

  このメソッドは、次の例に示すように、`String` cron 行を渡すことで Bean のタイムアウト・コールバック・メソッドを起動する OC4J cron タイマーを作成する場合に使用します。`info` 引数を使用して、アプリケーション情報を OC4J に渡します。この引数には `null` を設定できます。

```
...
import oracle.j2ee.ejb.timer.EJBTimerService;
import javax.ejb.timer.Timer;
...
@Resource EJBTimerService ets;

String cron = "1 * * * *";
String info = "";
Timer et = ets.createTimer(cron, info);
...
```

- `createTimer(int minute, int hour, int dayOfMonth, int month, int dayOfWeek, Serializable info)` または

  `createTimer(int minute, int hour, int dayOfMonth, int month, int dayOfWeek, int year, Serializable info)`

  このメソッドは、次の例に示すように、個別の引数として各 cron フィールドを渡すことで Bean のタイムアウト・コールバック・メソッドを起動する OC4J cron タイマーを作成する場合に使用します。`info` 引数を使用して、アプリケーション情報を OC4J に渡します。この引数には `null` を設定できます。

```
...
import oracle.j2ee.ejb.timer.EJBTimerService;
import javax.ejb.timer.Timer;
...
@Resource EJBTimerService ets;

int min=15;  // minutes
int hr=13;   // hour (1 PM)
int dom=28;  // day of month
int mo=1;    // month (January)
int dow=3;   // day of week (Wednesday)
String info = "";
Timer et = ets.createTimer(min, hr, dom, mo, dow, info);
...
```

- `createTimer(String cronline, String className, Serializable info)`

  このメソッドは、次の例に示すように、`String` cron 行を渡すことで特定の Java クラスの `main` メソッドを起動する OC4J cron タイマーを作成する場合に使用します。`info` 引数には、`null` を設定するか、クラスの `main` メソッドに渡すパラメータの `String[]` を設定できます。

```
...
import mypackage.MyClass;
import oracle.j2ee.ejb.timer.EJBTimerService;
import javax.ejb.timer.Timer;
...
@Resource EJBTimerService ets;

String cron = "1 * * * *";
String info = "";
Timer et = ets.createTimer(cron, MyClass.class.getName(), info);
...
```

- `createTimer(int minute, int hour, int dayOfMonth, int month, int dayOfWeek, String className, Serializable info)` または

  `createTimer(int minute, int hour, int dayOfMonth, int month, int dayOfWeek, int year, String className, Serializable info)`

  このメソッドは、次の例に示すように、個別の引数として各 cron フィールドを渡すことで特定の Java クラスの `main` メソッドを起動する OC4J cron タイマーを作成する場合に使用します。`info` 引数には、`null` を設定するか、クラスの `main` メソッドに渡すパラメータの `String[]` を設定できます。

```
...
import mypackage.MyClass;
import oracle.j2ee.ejb.timer.EJBTimerService;
import javax.ejb.timer.Timer;
...
@Resource EJBTimerService ets;

int min=15;  // minutes
int hr=13;   // hour (1 PM)
int dom=28;  // day of month
int mo=1;    // month (January)
int dow=3;   // day of week (Wednesday)
String info = "";
Timer et = ets.createTimer(min, hr, dom, mo, dow,
                           MyClass.class.getName(), info);
...
```

EJB 2.1 エンティティ Bean で `Timer` を作成すると、その主キーで識別される特定のエンティティ Bean インスタンスのタイムアウト・コールバック・メソッドがコンテナにより起動されます。特定のエンティティ Bean に対して作成されたタイマーは、そのエンティティ Bean の削除時に削除されます。

他の任意のタイプの EJB で `Timer` を作成すると、プール状態にあるそのタイプのインスタンスのタイムアウト・コールバック・メソッドがコンテナにより起動されます。

3. タイマーの起動時に OC4J で実行するアクションに応じて、次のように構成を完了します。

   a. `Class` を取得しない `createTimer` メソッドを使用するタイマーを作成した場合、OC4J により、タイムアウト・コールバック・メソッドがタイマーの起動時に実行されます。

      タイムアウト・コールバック・メソッドは、`static` または `final` にすることはできません。また、このメソッドは次のシグネチャを持つ必要があります。

```
void <METHOD>(Timer timer)
```

      タイムアウト・コールバック・メソッドは、次のいずれかの方法で実装できます。

      - EJB 3.0 Enterprise Bean では、例 25-5 に示すように任意の Bean メソッドに `@Timeout` アノテーションを付けます。
      - EJB 3.0 または EJB 2.1 Enterprise Bean では、例 25-6 に示すように `javax.ejb.TimedObject` インタフェースを実装します。

   b. `Class` を取得する `createTimer` メソッドを使用するタイマーを作成した場合、OC4J により、指定された `Class` の `main` メソッドがタイマーの起動時に実行されます。

      `main` メソッドは、次のシグネチャを持つ必要があります。

```
public static void main(String args[])
```

例 25-5 EJB 3.0 ステートレス・セッション Bean での OC4J cron タイマーの構成

```
import javax.ejb.Stateless;
import javax.annotation.PostConstruct;
import oracle.j2ee.ejb.timer.EJBTimerService;
import javax.ejb.Timer;
import javax.ejb.Timeout;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;

@Stateless
public class MySession {
    @PostConstruct
    @TransactionAttribute(value=REQUIRES_NEW)
    public void initialize() {
        @Resource EJBTimerService ets;

        String cron = "1 * * * *";
        String info = "";
        Timer et = ets.createTimer(cron, info);
    }

    ...

    @Timeout
    @TransactionAttribute(value=REQUIRES_NEW)
    public void timeoutCallback(Timer timer) {
    ...
    }
}
```

例 25-6 EJB 2.1 ステートレス・セッション Bean での OC4J cron タイマーの構成

```
import javax.ejb.SessionBean;
import oracle.j2ee.ejb.timer.EJBTimerService;
import javax.ejb.TimedObject;

public class MySession implements SessionBean, TimedObject {
    public void initialize() {
        String cron = "1 * * * *";
        String info = "";
        InitialContext ctx = new InitialContext();
        EJBTimerService ets = (EJBTimerService) ctx.getTimerService();
        Timer et = ets.createTimer(cron, info);
    }
    ...
    public void ejbTimeout(Timer timer) {
    ...
    }
}
```

タイマーのトラブルシューティング

この項の内容は次のとおりです。

タイマーに関する情報の取得

Timer オブジェクトを使用して、タイマーの情報を取得したり、タイマーをキャンセルしたりすることができます。使用できるメソッドは、cancel、getTimeRemaining、getNextTimeout、getHandle および getInfo です。オブジェクトの等価性を比較するには、Timer.equals(Object obj) メソッドを使用します。

永続的なタイマーの取得

タイマーは、Bean（ejbLoad、ejbStore など）のライフ・サイクル中持続するように設定する必要があります。永続 Timer オブジェクトは、そのハンドルを使用して取得できます。TimerHandle は、Timer.getHandle メソッドを使用して取得します。次に、TimerHandle.getTimer メソッドを使用して、永続 Timer オブジェクトを取得できます。

注意： タイマーとそのハンドルは、ローカル・オブジェクトです。このため、Bean のリモート・インタフェースを介して渡すことは避けてください。

トランザクションの有効範囲内でのタイマーの使用

タイマーは、通常、トランザクションの有効範囲内で作成またはキャンセルします。このため、Bean は、RequiresNew を使用してトランザクション内に存在するよう構成するのが普通です。トランザクションがロールバックされると、コンテナはタイムアウトを再試行します。

トランザクションの詳細は、『Oracle Containers for J2EE サービス・ガイド』を参照してください。

タイマーについて NoSuchObjectLocalException が発生する場合

正常に起動された、またはキャンセルされたタイマー・オブジェクトでメソッドを起動しようとすると、NoSuchObjectLocalException が返されます。

第 IX 部

EJB アプリケーションのパッケージ化およびデプロイ

第 IX 部では、EJB 3.0 および EJB 2.1 Enterprise JavaBeans を使用して J2EE アプリケーションをパッケージ化およびデプロイする手順に関する情報を示します。概念的な情報は、第 I 部「EJB の概要」を参照してください。

第 IX 部は次の各章で構成されています。

- 第 26 章「デプロイメント・ディスクリプタ・ファイルの構成」
- 第 27 章「EJB アプリケーションのパッケージ化」
- 第 28 章「OC4J への EJB アプリケーションのデプロイ」

26

デプロイメント・ディスクリプタ・ファイルの構成

この章では、OC4J アプリケーションで使用できる次のような様々なデプロイメント・ディスクリプタ・ファイルの構成方法を説明します。

- ejb-jar.xml ファイルの構成
- toplink-ejb-jar.xml ファイルの構成
- orion-ejb-jar.xml ファイルの構成
- ejb3-toplink-sessions.xml ファイルの構成
- persistence.xml ファイルの構成

詳細は、2-6 ページの「EJB デプロイメント・ディスクリプタ・ファイルについて」を参照してください。

ejb-jar.xml ファイルの構成

この項の内容は次のとおりです。

- 移行時の ejb-jar.xml の作成
- デプロイ時の ejb-jar.xml ファイルの作成
- JDeveloper での ejb-jar.xml の作成

詳細は、2-6 ページの「ejb-jar.xml ファイルとは」を参照してください。

移行時の ejb-jar.xml の作成

EJB 2.1 の場合のみ、移行時に `ejb-jar.xml` ファイルを自動的に生成できます（3-15 ページの「TopLink EJB 2.1 永続性マネージャへの移行」を参照）。生成後に、TopLink Workbench を使用して、このファイルをカスタマイズおよび再エクスポートできます（2-2 ページの「TopLink Workbench の使用方法」を参照）。

デプロイ時の ejb-jar.xml ファイルの作成

1 つ以上のアノテーションのある EJB 3.0 アプリケーションのデプロイ時に、OC4J はそのメモリー内の `ejb-jar.xml` ファイルをデプロイ・ディレクトリ内の `orion-ejb-jar.xml` ファイルと同じ場所（`<ORACLE_HOME>/j2ee/home/application-deployments/my_application/META-INF`）に書き込みます。

この `ejb-jar.xml` ファイルは、アノテーションとデプロイ済 `ejb-jar.xml` ファイル（存在する場合）の両方から取得された構成を表します。

JDeveloper での ejb-jar.xml の作成

JDeveloper を使用して、`ejb-jar.xml` ファイルを生成および更新できます。

詳細は、2-2 ページの「JDeveloper の使用方法」を参照してください。

toplink-ejb-jar.xml ファイルの構成

`toplink-ejb-jar.xml` ファイルは、TopLink JPA プレビュー永続性プロバイダを使用している場合にのみ適用されます。

> **注意：** OC4J では、TopLink Essentials JPA 永続性プロバイダがデフォルトで使用されます。この場合、TopLink JPA 拡張を使用して、TopLink ディスクリプタレベルのオプション（マッピングを含む）を構成できます（3-5 ページの「TopLink Essentials JPA 永続性を使用した TopLink API への実行時アクセス」を参照）。

この項の内容は次のとおりです。

- 移行時の toplink-ejb-jar.xml の作成
- TopLink Workbench での toplink-ejb-jar.xml の作成

詳細は、次を参照してください。

- 2-8 ページの「toplink-ejb-jar.xml ファイルとは」
- 『Oracle TopLink 開発者ガイド』の OC4J および toplink-ejb-jar.xml ファイルに関する項

移行時の toplink-ejb-jar.xml の作成

EJB 2.1 プロジェクトの場合のみ、Orion CMP アプリケーションを TopLink 永続性に移行する場合（3-15 ページの「TopLink EJB 2.1 永続性マネージャへの移行」を参照）、TopLink 移行ツールは `toplink-ejb-jar.xml` ファイルを自動的に作成します。

生成後に、TopLink Mapping Workbench を使用して、カスタマイズおよび再エクスポートできます（『Oracle TopLink 開発者ガイド』の「TopLink Workbench」を参照）。

TopLink Workbench での toplink-ejb-jar.xml の作成

EJB 3.0 プロジェクトでは、エンティティ・クラスで使用している JDK 1.5 言語拡張がアノテーションのみの場合は、TopLink Workbench を使用して `toplink-ejb-jar.xml` ファイルを作成および構成できます。TopLink Workbench を使用して、このファイルを作成および構成することをお薦めします。

EJB 2.1 プロジェクトの場合は、TopLink Workbench を使用して、`toplink-ejb-jar.xml` ファイルで永続性プロパティを構成します。Orion CMP アプリケーションを TopLink 永続性に移行する場合（3-15 ページの「TopLink EJB 2.1 永続性マネージャへの移行」を参照）、TopLink 移行ツールは TopLink Workbench プロジェクトを自動的に作成します。TopLink Workbench プロジェクトを使用して `toplink-ejb-jar.xml` ファイルを作成できます。

詳細は、次を参照してください。

- 『Oracle TopLink 開発者ガイド』の TopLink Workbench の理解に関する項
- 『Oracle TopLink 開発者ガイド』の TopLink Workbench を使用した project.xml の作成に関する項

orion-ejb-jar.xml ファイルの構成

OC4J 固有のオプションを指定するため、`orion-ejb-jar.xml` ファイルを作成し、次の適切な要素を構成できます。

- A-5 ページの「<session-deployment>」
- A-11 ページの「<entity-deployment>」
- A-19 ページの「<message-driven-deployment>」

詳細は、2-7 ページの「orion-ejb-jar.xml ファイルとは」を参照してください。

注意： 別の方法として、EJB 3.0 アプリケーションでは、セッション Bean およびメッセージドリブン Bean に対して OC4J 固有のアノテーションを使用できます。`orion-ejb-jar.xml` ファイルのベンダー拡張設定は、OC4J 固有のアノテーションを使用した拡張設定に優先します。

詳細は、次を参照してください。

- 5-12 ページの「EJB 3.0 セッション Bean の OC4J 固有のデプロイ・オプションの構成」
- 10-20 ページの「EJB 3.0 MDB の OC4J 固有のデプロイ・オプションの構成」

---

ejb3-toplink-sessions.xml ファイルの構成

ejb3-toplink-sessions.xml ファイルは、TopLink JPA プレビュー永続性プロバイダを使用している場合にのみ適用されます。

> **注意：** OC4J では、TopLink Essentials JPA 永続性プロバイダがデフォルトで使用されます。この場合、TopLink JPA 拡張を使用して、TopLink セッションレベルのオプションを構成できます（3-5 ページの「TopLink Essentials JPA 永続性を使用した TopLink API への実行時アクセス」を参照）。

この項の内容は次のとおりです。

- TopLink Workbench での ejb3-toplink-sessions.xml の作成

詳細は、2-9 ページの「ejb3-toplink-sessions.xml ファイルとは」を参照してください。

TopLink Workbench での ejb3-toplink-sessions.xml の作成

EJB 3.0 アプリケーションでは、エンティティ・クラスで使用している JDK 1.5 言語拡張がアノテーションのみの場合は、TopLink Workbench を使用して ejb3-toplink-sessions.xml ファイルを作成および構成できます。TopLink Workbench を使用して、このファイルを作成および構成することをお薦めします。

詳細は、次を参照してください。

- 『Oracle TopLink 開発者ガイド』の TopLink Workbench の理解に関する項
- 『Oracle TopLink 開発者ガイド』の TopLink Workbench を使用した project.xml の作成に関する項

persistence.xml ファイルの構成

この項の内容は次のとおりです。

- 名前付き永続性ユニットを含む persistence.xml ファイルの構成
- OC4J のデフォルト永続性ユニットの persistence.xml ファイルの構成
- 永続性ユニットでのデータソースの指定
- 永続性ユニットでのベンダー拡張の構成

詳細は、2-10 ページの「persistence.xml ファイルとは」を参照してください。

名前付き永続性ユニットを含む persistence.xml ファイルの構成

例 26-1 に、1 つの永続性ユニットを含む persistence.xml ファイルの例を示します。

例 26-1 名前付き永続性ユニット

```
<persistence-unit name="OrderManagement5">
    <provider>com.acme.persistence</provider>
    <transaction-type>RESOURCE_LOCAL</transaction-type>
    <mapping-file>order1.xml</mapping-file>
    <jar-file>order.jar</jar-file>
    <class>com.acme.Order</class>
    <properties>
        <property name="com.acme.persistence.sql-logging" value="on"/>
    </properties>
</persistence-unit>
```

この永続性ユニットは OrderManagement5 という名前で、EntityManager プロバイダ com.acme.persistence を使用します。その <transaction-type> では、この永続性ユニットで JTA 以外のデータソースのみが必要とされるよう指定します。<mapping-file>、<jar-file>、<class> の要素をすべて使用して永続管理クラスの設定を定義します（26-5 ページの「この永続性ユニットに含まれる永続管理クラス」を参照）。<property> 要素を使用して、プロパティ com.acme.persistence.sql.logging を値 on に設定します。

<persistence-unit> 要素の属性およびサブ要素の詳細は、EJB 3.0 の仕様を参照してください。

この永続性ユニットに含まれる永続管理クラス

次の 1 つ以上を使用して、永続性ユニットに関連付けられている永続管理クラスを指定できます。

- <mapping-file> 要素 : 1 つ以上のオブジェクト・リレーショナル・マッピング XML ファイル（orm.xml ファイル）を指定します。
- <jar-file> 要素 : クラスを検索する 1 つ以上の JAR ファイルを指定します。
- <class> 要素 : クラスの明示的なリストを指定します。
- 永続性ユニットのルートに含まれるアノテーション付き管理永続性クラス。

  永続性ユニットのルートは、META-INF ディレクトリに persistence.xml ファイルを含む JAR ファイルまたはディレクトリです。管理永続性クラスを除外するには、永続性ユニットに <exclude-unlisted-classes> 要素を追加します。

OC4J のデフォルト永続性ユニットの persistence.xml ファイルの構成

OC4J のデフォルトの永続性ユニットを使用すると、永続性ユニットを名前で指定しなくてもエンティティ・マネージャを取得できます（2-10 ページの「OC4J の永続性ユニットのデフォルトについて」を参照）。

デフォルトでは、OC4J のデフォルト永続性ユニットを使用するために、`persistence.xml` ファイルをデプロイする必要は一切ありません。

`orion-ejb-jar.xml` ファイルの属性 `disable-default-persistent-unit` を `true` に設定した場合、OC4J は `persistence.xml` ファイルを必要とします。この場合も、空の永続性ユニットを指定する場合に OC4J のデフォルトの永続性ユニットを使用できます。`persistence.xml` ファイルは、次のいずれかを使用して空の永続性ユニットで構成します。

- 空の `<persistence>` 要素

  ```
  <persistence>
  </persistence>
  ```

- 自己終結 `<persistence/>` 要素
- 完全に空の（長さが 0 の）`persistence.xml` ファイル

有効範囲またはモジュールごとに 1 つの永続性ユニットを指定できます（EJB JAR ごとに 1 つなど）。

永続性ユニットでのデータソースの指定

Java EE アプリケーションでは、例 26-2 に示すように、`<jta-data-source>` 要素でデータソースを指定します。詳細は、第 20 章「データソースの構成」を参照してください。

Java SE アプリケーションでは、例 26-3 に示すように、JDBC ベンダー拡張を使用してデータソースを指定します。詳細は、26-8 ページの「JDBC 用の TopLink JPA 拡張（Java SE）」を参照してください。

または、OC4J を通じて、デフォルトのデータソースを使用できます（2-19 ページの「デフォルトのデータソース」を参照）。

永続性ユニットでのベンダー拡張の構成

この項では、永続性ユニットに定義できる次のような TopLink JPA ベンダー拡張について説明します。

- JDBC 用の TopLink JPA 拡張（Java SE）
- キャッシング用の TopLink JPA 拡張
- ロギング用の TopLink JPA 拡張
- データベース、セッションおよびアプリケーション・サーバー用の TopLink JPA 拡張
- カスタマイズ用の TopLink JPA 拡張
- スキーマ生成用の TopLink JPA 拡張

これらのベンダー拡張を指定するには、persistence.xml ファイルの <properties> 要素を使用します。例 26-2 に、Java EE アプリケーションで persistence.xml ファイルに TopLink JPA 永続性ユニットの拡張を設定する方法を示し、例 26-3 に、Java SE アプリケーションで同じ拡張を設定する方法を示します。

例 26-2 persistence.xml ファイルでのベンダー拡張の構成（Java EE）

```
<persistence-unit name="default" transaction-type="JTA">
    <provider>
        oracle.toplink.essentials.PersistenceProvider
    </provider>
    <jta-data-source>
        jdbc/MyDataSource
    </jta-data-source>
    <properties>
        <property name="toplink.logging.level" value="INFO"/>
    </properties>
</persistence-unit>
```

例 26-3 persistence.xml ファイルでのベンダー拡張の構成（Java SE）

```
<persistence-unit name="default" transaction-type="RESOURCE_LOCAL">
    <provider>
        oracle.toplink.essentials.PersistenceProvider
    </provider>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
        <property name="toplink.logging.level" value="INFO"/>
        <property name="toplink.jdbc.driver" value="oracle.jdbc.OracleDriver"/>
        <property name="toplink.jdbc.url" value="jdbc:oracle:thin:@myhost:1521:MYSID"/>
        <property name="toplink.jdbc.password" value="tiger"/>
        <property name="toplink.jdbc.user" value="scott"/>
    </properties>
</persistence-unit>
```

または、例 26-4 に示すように、TopLink JPA 永続性ユニットの拡張をプロパティの Map に設定し、javax.persistence.Persistence のメソッド createEntityManagerFactory へのコールに渡すことができます。この方法で、persistence.xml ファイルに設定された拡張をオーバーライドできます。プロパティの Map に拡張を設定する場合、次のような oracle.toplink.essentials.config の適切な構成クラスの public static final フィールドを使用してその値を設定できます。

- CacheType
- TargetDatabase
- TargetServer
- TopLinkProperties

> **注意：** これらのクラスにアクセスするには、適切な OC4J 永続性 JAR をクラスパスに配置します。詳細は、3-3 ページの「TopLink Essentials JPA 永続性プロバイダ」を参照してください。

例 26-4 に、CacheType 構成クラスを使用して拡張 toplink.cache.type.default の値を設定する方法を示します。

例 26-4 EntityManagerFactory 作成時のベンダー拡張の構成

```
import oracle.toplink.essentials.config.CacheType;

Map properties = new HashMap();
properties.put(TopLinkProperties.CACHE_TYPE_DEFAULT, CacheType.Full);
EntityManagerFactory emf = Persistence.createEntityManagerFactory("default", properties);
```

JDBC 用の TopLink JPA 拡張（Java SE）

表 26-1 に、JDBC ドライバ・パラメータを構成するために persistence.xml ファイルに定義できる TopLink JPA 拡張をリストします。これらの拡張は、EJB コンテナの外部で使用する場合にのみ適用されます。

表 26-1 JDBC 用の TopLink JPA 拡張（Java SE）

プロパティ	使用方法	デフォルト
toplink.jdbc.bind-parameters	問合せでパラメータ・バインディングを使用するかどうかを制御します。 詳細は、『Oracle TopLink 開発者ガイド』の「一致する問合せおよびディスクリプタの使用」を参照してください。 **有効な値:** ■ true: すべてのパラメータをバインドします。 ■ false: パラメータをバインドしません。 **例:** persistence.xml ファイル <property name="toplink.jdbc.bind-parameters" value="true"/> **例:** プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_BIND_PARAMETERS, "true");	true
toplink.jdbc.driver	パッケージ名で完全修飾された、使用する JDBC ドライバのクラス名。このクラスは、アプリケーション・クラスパスに含める必要があります。 **例:** persistence.xml ファイル <property name="toplink.jdbc.driver" value="oracle.jdbc.driver.OracleDriver"/> **例:** プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_DRIVER, "oracle.jdbc.driver.OracleDriver");	
toplink.jdbc.password	JDBC ユーザーのパスワード。 **例:** persistence.xml ファイル <property name="toplink.jdbc.password" value="tiger"/> **例:** プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_PASSWORD, "tiger");	

表 26-1 JDBC 用の TopLink JPA 拡張（Java SE）（続き）

プロパティ	使用方法	デフォルト
toplink.jdbc.read-connections.max	JDBC 読取り接続プールで使用可能な接続の最大数。 **有効な値：**（JDBC ドライバに応じて）String の 0 ～ Integer.MAX_VALUE **例：**persistence.xml ファイル <property name="toplink.jdbc.read-connections.max" value="3"/> **例：**プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_READ_CONNECTIONS_MAX, "3");	2
toplink.jdbc.read-connections.min	JDBC 読取り接続プールで使用可能な接続の最小数。 **有効な値：**（JDBC ドライバに応じて）String の 0 ～ Integer.MAX_VALUE **例：**persistence.xml ファイル <property name="toplink.jdbc.read-connections.min" value="1"/> **例：**プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_READ_CONNECTIONS_MIN, "1");	2
toplink.jdbc.read-connections.shared	共有読取り接続の同時使用を許可するかどうかを指定します。 **有効な値：** ■ true: 共有読取り接続の同時使用を許可します。 ■ false: 共有読取り接続の同時使用を許可しません。同時読取りが発生した場合、それぞれに独自の読取り接続が割り当てられます。 **例：**persistence.xml ファイル <property name="toplink.jdbc.read-connections.shared" value="true"/> **例：**プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_READ_CONNECTIONS_SHARED, "true");	false
toplink.jdbc.url	JDBC ドライバで必要とされる JDBC 接続 URL。 **例：**persistence.xml ファイル <property name="toplink.jdbc.url" value="jdbc:oracle:thin:@MYHOST:1521:MYSID"/> **例：**プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_URL, "jdbc:oracle:thin:@MYHOST:1521:MYSID");	
toplink.jdbc.user	JDBC ユーザーのユーザー名。 **例：**persistence.xml ファイル <property name="toplink.jdbc.user" value="scott"/> **例：**プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_USER, "scott");	

表 26-1 JDBC 用の TopLink JPA 拡張（Java SE）（続き）

プロパティ	使用方法	デフォルト
toplink.jdbc.write-connections.max	JDBC 書込み接続プールで使用可能な接続の最大数。 **有効な値：**（JDBC ドライバに応じて）String の 0 ～ Integer.MAX_VALUE **例：**persistence.xml ファイル <property name="toplink.jdbc.write-connections.max" value="5"/> **例：**プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_WRITE_CONNECTIONS_MAX, "5");	10
toplink.jdbc.write-connections.min	JDBC 書込み接続プールで使用可能な接続の最小数。 **有効な値：**（JDBC ドライバに応じて）String の 0 ～ Integer.MAX_VALUE **例：**persistence.xml ファイル <property name="toplink.jdbc.write-connections.min" value="2"/> **例：**プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.JDBC_WRITE_CONNECTIONS_MIN, "2");	5

キャッシング用の TopLink JPA 拡張

表 26-2 に、TopLink キャッシュを構成するために persistence.xml ファイルに定義できる TopLink JPA 拡張をリストします。

詳細は、『Oracle TopLink 開発者ガイド』のキャッシュの理解に関する項を参照してください。

表 26-2 キャッシング用の TopLink JPA 拡張

プロパティ	使用方法	デフォルト
toplink.cache.type.default	セッション・キャッシュのデフォルト・タイプ。 セッション・キャッシュは、特定のセッションに関連付けられたクライアントにサービスを提供する共有キャッシュです。クライアント・セッションを使用するデータソースを対象にオブジェクトの読取りまたは書込みが発生すると、TopLink によりそのオブジェクトのコピーが親のサーバー・セッションのキャッシュに保存され、子のクライアント・セッションからアクセス可能になります。 **有効な値:** oracle.toplink.essentials.config.CacheType ■ Full: このオプションでは、完全なキャッシングが実行され、識別情報が保証されます。オブジェクトは、削除されないかぎりメモリーからフラッシュされません。 詳細は、『Oracle TopLink 開発者ガイド』の「完全アイデンティティ・マップ」を参照してください。 ■ HardWeak: このオプションは、Weak に似ていますが、ハード参照を使用する最多使用サブキャッシュを維持するところが異なります。 詳細は、『Oracle TopLink 開発者ガイド』の「ソフト / ハード・キャッシュ弱アイデンティティ・マップ」を参照してください。 ■ NONE: このオプションでは、オブジェクト識別情報は維持されず、オブジェクトはキャッシュされません。このオプションは使用しないことをお薦めします。 詳細は、『Oracle TopLink 開発者ガイド』の「アイデンティティ・マップなし」を参照してください。 ■ SoftWeak: このオプションは、Weak に似ていますが、ソフト参照を使用する最多使用サブキャッシュを維持するところが異なります。通常の環境でキャッシュに使用されるメモリーを制御する手段としては、このアイデンティティ・マップを使用することをお薦めします。 詳細は、『Oracle TopLink 開発者ガイド』の「ソフト / ハード・キャッシュ弱アイデンティティ・マップ」を参照してください。 ■ Weak: このオプションは、Full に似ていますが、弱い参照を使用してオブジェクトを参照するところが異なります。このオプションでは、使用されるメモリーが Full より減少しますが、クライアント / サーバー・トランザクション間でキャッシュ計画は維持されません。開始したトランザクションがサーバー・サイドで継続される場合は、このアイデンティティ・マップを使用することをお薦めします。 詳細は、『Oracle TopLink 開発者ガイド』の「弱アイデンティティ・マップ」を参照してください。 **例:** persistence.xml ファイル <property name="toplink.cache.type.default" value="Full"/> **例:** プロパティ Map import oracle.toplink.essentials.config.CacheType; import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.CACHE_TYPE_DEFAULT, CacheType.Full);	SoftWeak

表 26-2 キャッシング用の TopLink JPA 拡張（続き）

プロパティ	使用方法	デフォルト
toplink.cache.size.default	TopLink キャッシュに許可するデフォルトの最大オブジェクト数。 **有効な値 :** String の 0 ～ Integer.MAX_VALUE **例 :** <property name="toplink.cache.size.default" value="5000"/> **例 :** プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.CACHE_SIZE_DEFAULT, 1000);	1000
toplink.cache.shared.default	TopLink セッション・キャッシュを複数のクライアント・セッションで共有するかどうかのデフォルト。 **有効な値 :** ■ true: セッション・キャッシュは、そのセッションに関連付けられたすべてのクライアントにサービスを提供します。クライアント・セッションを使用するデータソースを対象にオブジェクトの読取りまたは書込みが発生すると、TopLink によりそのオブジェクトのコピーが親のサーバー・セッションのキャッシュに保存され、セッション内の他のすべてのプロセスからアクセス可能になります。 ■ false: セッション・キャッシュは、単一の孤立クライアントにのみ排他的にサービスを提供します。孤立クライアントは、共有セッション・キャッシュのオブジェクトを参照できますが、他のクライアントは孤立クライアントの排他キャッシュのオブジェクトを参照できません。 **例 :** <property name="toplink.cache.shared.default" value="true"/> **例 :** プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.CACHE_SHARED_DEFAULT, "true");	true
toplink.cache.type.<ENTITY>	<ENTITY> という JPA エンティティのセッション・キャッシュのタイプ。 エンティティ名の詳細は、@Entity を参照してください。 **有効な値 :** oracle.toplink.essentials.config.CacheType ■ Full: toplink.cache.type.default を参照してください。 ■ HardWeak: toplink.cache.type.default を参照してください。 ■ NONE: toplink.cache.type.default を参照してください。 ■ SoftWeak: toplink.cache.type.default を参照してください。 ■ Weak: toplink.cache.type.default を参照してください。 **例 :** persistence.xml ファイル <property name="toplink.cache.type.Order" value="Full"/> **例 :** プロパティ Map import oracle.toplink.essentials.config.CacheType import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(TopLinkProperties.CACHE_TYPE+".Order", CacheType.Full);	SoftWeak
toplink.cache.size.<ENTITY>	TopLink キャッシュに許可する、JPA エンティティ名 <ENTITY> で示されるタイプの JPA エンティティの最大数。 エンティティ名の詳細は、@Entity を参照してください。 **有効な値 :** String の 0 ～ Integer.MAX_VALUE **例 :** <property name="toplink.cache.size.Order" value="5000"/> **例 :** プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.CACHE_SIZE+".Order", 1000);	1000

表 26-2 キャッシング用の TopLink JPA 拡張（続き）

プロパティ	使用方法	デフォルト
toplink.cache.shared.<ENTITY>	TopLink セッション・キャッシュを、JPA エンティティ名 <ENTITY> で示されるタイプの JPA エンティティの複数のクライアント・セッションで共有するかどうか。 エンティティ名の詳細は、@Entity を参照してください。 **有効な値:** ■ true: セッション・キャッシュは、そのセッションに関連付けられたすべてのクライアントにサービスを提供します。クライアント・セッションを使用するデータソースを対象にオブジェクトの読取りまたは書込みが発生すると、TopLink によりそのオブジェクトのコピーが親のサーバー・セッションのキャッシュに保存され、セッション内の他のすべてのプロセスからアクセス可能になります。 ■ false: セッション・キャッシュは、単一の孤立クライアントにのみ排他的にサービスを提供します。孤立クライアントは、共有セッション・キャッシュのオブジェクトを参照できますが、他のクライアントは孤立クライアントの排他キャッシュのオブジェクトを参照できません。 **例:** <property name="toplink.cache.shared.Order" value="true"/> **例:** プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(ToplinkProperties.CACHE_SHARED+".Order", "true");	true

ロギング用の TopLink JPA 拡張

表 26-3 に、TopLink ロギングを構成するために persistence.xml ファイルに定義できる TopLink JPA 拡張をリストします。

詳細は、『Oracle TopLink 開発者ガイド』の「ロギングの構成」を参照してください。

表 26-3 ロギング用の TopLink JPA 拡張

プロパティ	使用方法	デフォルト
toplink.logging.level	ログ・レベルを構成してログ出力の量と詳細度を制御します（次のログ・レベルは、情報量の少ない順に記載されています）。 **有効な値:** java.util.logging.Level ■ OFF: ロギングは無効です。 ■ Level.SEVERE: TopLink が続行不可能であることを示す例外と、ログイン中に生成されるすべての例外を記録します。これには、スタック・トレースが含まれます。 ■ WARNING: SEVERE レベルで記録されないすべての例外を含め、TopLink を停止させることのない例外を記録します。これには、スタック・トレースは含まれません。 ■ INFO: ユーザー名を含め、各サーバー・セッションのログインおよびログアウトを記録します。セッションの取得後に、詳細な情報が記録されます。 ■ CONFIG: ログイン、JDBC 接続およびデータベース情報のみを記録します。 ■ FINE: SQL を記録します。 ■ FINER: WARNING と同様です。スタック・トレースが含まれます。 ■ FINEST: 低レベルな追加情報が含まれます。 **例:** persistence.xml ファイル <property name="toplink.logging.level" value="WARNING"/> **例:** プロパティ Map import java.util.logging.Level; import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(TopLinkProperties.LOGGING_LEVEL, Level.INFO);	CONFIG
toplink.logging.timestamp	各ログ・エントリにタイムスタンプを記録するかどうかを制御します。 **有効な値:** ■ true: タイムスタンプを記録します。 ■ false: タイムスタンプを記録しません。 **例:** persistence.xml ファイル <property name="toplink.logging.timestamp" value="true"/> **例:** プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(TopLinkProperties.LOGGING_TIMESTAMP, "true");	true
toplink.logging.thread	各ログ・エントリにスレッド識別子を記録するかどうかを制御します。 **有効な値:** ■ true: スレッド識別子を記録します。 ■ false: スレッド識別子を記録しません。 **例:** persistence.xml ファイル <property name="toplink.logging.thread" value="true"/> **例:** プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(TopLinkProperties.LOGGING_THREAD, "true");	true

表 26-3 ロギング用の TopLink JPA 拡張（続き）

プロパティ	使用方法	デフォルト
toplink.logging.session	各ログ・エントリに TopLink セッション識別子を記録するかどうかを制御します。 **有効な値:** ■ true: TopLink セッション識別子を記録します。 ■ false: TopLink セッション識別子を記録しません。 **例:** persistence.xml ファイル <property name="toplink.logging.session" value="true"/> **例:** プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(TopLinkProperties.LOGGING_SESSION, "true");	true
toplink.logging.exceptions	TopLink コード内からスローされる例外をコール側アプリケーションに返す前に記録するかどうかを制御します。すべての例外が記録され、アプリケーション・コードでマスクされないことが保証されます。 **有効な値:** ■ true: すべての例外を記録します。 ■ false: 例外を記録しません。 **例:** persistence.xml ファイル <property name="toplink.logging.exceptions" value="true"/> **例:** プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(TopLinkProperties.LOGGING_EXCEPTIONS, "true");	false

データベース、セッションおよびアプリケーション・サーバー用の TopLink JPA 拡張

表 26-4 に、データベース、セッションおよびアプリケーション・サーバー用の TopLink 拡張を構成するために persistence.xml ファイルに定義できる TopLink JPA 拡張をリストします。

表 26-4 データベース、セッションおよびアプリケーション・サーバー用の TopLink JPA 拡張

プロパティ	使用方法	デフォルト

toplink.target-database	JPA アプリケーションで使用するデータベースのタイプを指定します。TargetDatabase 列挙には、サポートされる多くの一般的なデータベース・タイプのエントリが含まれます。
	有効な値 : oracle.toplink.essentials.config.TargetDatabase
	■ Attunity: Attunity データベースを使用するよう永続性プロバイダを構成します。
	■ Auto: TopLink は、データベースにアクセスし、JDBC が提供するメタデータを使用してターゲット・データベースを決定します。この設定は、このメタデータをサポートする JDBC ドライバで使用可能です。
	■ Cloudscape: Cloudscape データベースを使用するよう永続性プロバイダを構成します。
	■ Database: ターゲット・データベースがここにリストされておらず、Auto オプションに必要なメタデータの使用が JDBC ドライバでサポートされない場合、一般的な選択オプションを使用するよう永続性プロバイダを構成します。
	■ DB2: DB2 データベースを使用するよう永続性プロバイダを構成します。
	■ DB2Mainframe: DB2Mainframe データベースを使用するよう永続性プロバイダを構成します。
	■ DBase: DBase データベースを使用するよう永続性プロバイダを構成します。
	■ Derby: Derby データベースを使用するよう永続性プロバイダを構成します。
	■ HSQL: HSQL データベースを使用するよう永続性プロバイダを構成します。
	■ Informix: Informix データベースを使用するよう永続性プロバイダを構成します。
	■ JavaDB: JavaDB データベースを使用するよう永続性プロバイダを構成します。
	■ MySQL4: MySQL4 データベースを使用するよう永続性プロバイダを構成します。
	■ Oracle: Oracle データベースを使用するよう永続性プロバイダを構成します。
	■ PointBase: PointBase データベースを使用するよう永続性プロバイダを構成します。
	■ PostgreSQL: PostgreSQL データベースを使用するよう永続性プロバイダを構成します。
	■ SQLAnyWhere: SQLAnyWhere データベースを使用するよう永続性プロバイダを構成します。
	■ SQLServer: SQLServer データベースを使用するよう永続性プロバイダを構成します。
	■ Sybase: Sybase データベースを使用するよう永続性プロバイダを構成します。
	■ TimesTen: TimesTen データベースを使用するよう永続性プロバイダを構成します。
	例 : persistence.xml ファイル
	`<property name="toplink.target-database" value="Oracle"/>`
	例 : プロパティ Map
	`import oracle.toplink.essentials.config.TargetDatabase;`
	`import oracle.toplink.essentials.config.TopLinkProperties;`
	`propertiesMap.put(TopLinkProperties.TARGET_DATABASE,`
	`TargetDatabase.Oracle);`
		Auto

表 26-4 データベース、セッションおよびアプリケーション・サーバー用の TopLink JPA 拡張（続き）

プロパティ	使用方法	デフォルト

toplink.session-name	静的セッション・マネージャに TopLink セッションを格納する場合の名前を指定します。Java 永続性 API のコンテキストの外部で TopLink 共有セッションにアクセスする必要がある場合、このオプションを使用します。
	有効な値 : サーバー・デプロイ内で一意となる有効な TopLink セッション名
	例 : persistence.xml ファイル
	`<property name="toplink.session-name" value="MySession"/>`
	例 : プロパティ Map
	`import oracle.toplink.essentials.config.TopLinkProperties;`
	`propertiesMap.put(TopLinkProperties.SESSION_NAME, "MySession");`
		TopLink が生成する一意の名前

toplink.target-server	JPA アプリケーションで使用するアプリケーション・サーバーのタイプを指定します。
	有効な値 : oracle.toplink.essentials.config.TargetServer
	■ None: アプリケーション・サーバーを使用しないよう永続性プロバイダを構成します。
	■ OC4J_10_1_3: OC4J リリース 10.1.3.0 を使用するよう永続性プロバイダを構成します。
	■ SunAS9: Sun Application Server バージョン 9 を使用するよう永続性プロバイダを構成します。
	例 : persistence.xml ファイル
	`<property name="toplink.target-server" value="OC4J_10_1_3"/>`
	例 : プロパティ Map
	`import oracle.toplink.essentials.config.TargetServer;`
	`import oracle.toplink.essentials.config.TopLinkProperties;`
	`propertiesMap.put(TopLinkProperties.TARGET_SERVER,`
	`TargetServer.OC4J_10_1_3);`
		None

カスタマイズ用の TopLink JPA 拡張

表 26-5 に、TopLink のカスタマイズおよび検証を構成するために persistence.xml ファイルに定義できる TopLink JPA 拡張をリストします。

表 26-5 カスタマイズおよび検証用の TopLink JPA 拡張

プロパティ	使用方法	デフォルト
toplink.weaving	エンティティ・クラスのウィービングを実行するかどうかを制御します。ウィービングは、@OneToOne および @ManyToOne の関連の遅延フェッチを使用するために必要です。 **有効な値:** ■ true: エンティティ・クラスをウィービングします。 ■ false: エンティティ・クラスをウィービングしません。 ■ static: エンティティ・クラスを静的にウィービングします。 JVM コマンドラインで -javagent:toplink-essentials-agent.jar を使用できない環境において Java EE 5 コンテナの外部でアプリケーションを実行する場合、このオプションを使用します。 **例:** persistence.xml ファイル <property name="toplink.weaving" value="true"/> **例:** プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(TopLinkProperties.WEAVING, "true");	true
toplink.session.customizer	TopLink セッション・カスタマイザ・クラスを指定します。このクラスは、oracle.toplink.essentials.tools.sessionconfiguration.SessionCustomizer インタフェースを実装し、デフォルトの（引数なしの）コンストラクタを提供する Java クラスです。このクラスの customize メソッドを使用して oracle.toplink.essentials.sessions.Session を取得し、拡張 TopLink セッション API にプログラム的にアクセスします。 詳細は、セッションのカスタマイズに関する項を参照してください。 **有効な値:** パッケージ名で完全修飾された SessionCustomizer クラスの名前 **例:** persistence.xml ファイル <property name="toplink.session.customizer" value="acme.sessions.MySessionCustomizer"/> **例:** プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(TopLinkProperties.SESSION_CUSTOMIZER, "acme.sessions.MySessionCustomizer");	

表 26-5 カスタマイズおよび検証用の TopLink JPA 拡張（続き）

プロパティ	使用方法	デフォルト
toplink.descriptor.customizer.<ENTITY>	TopLink ディスクリプタ・カスタマイザ・クラスを指定します。このクラスは、oracle.toplink.essentials.tools.sessionconfiguration.DescriptorCustomizer インタフェースを実装し、デフォルトの（引数なしの）コンストラクタを提供する Java クラスです。このクラスの customize メソッドを使用して oracle.toplink.essentials.descriptors.ClassDescriptor を取得し、<ENTITY> という JPA エンティティに関連付けられたディスクリプタの拡張 TopLink ディスクリプタおよびマッピング API にプログラム的にアクセスします。 エンティティ名の詳細は、@Entity を参照してください。 TopLink ディスクリプタの詳細は、次を参照してください。 ■ ディスクリプタの理解に関する項 ■ ディスクリプタのカスタマイズに関する項 **有効な値 :** パッケージ名で完全修飾された DescriptorCustomizer クラスの名前 **例 :** persistence.xml ファイル <property name="toplink.descriptor.customizer.Order" value="acme.sessions.MyDescriptorCustomizer"/> **例 :** プロパティ Map import oracle.toplink.essentials.config.TopLinkProperties; propertiesMap.put(TopLinkProperties.DESCRIPTOR_CUSTOMIZER+".Order", "acme.sessions.MyDescriptorCustomizer");	

スキーマ生成用の TopLink JPA 拡張

表 26-6 に、スキーマ生成を構成するために persistence.xml ファイルに定義できる TopLink JPA 拡張をリストします。

表 26-6 スキーマ生成用の TopLink JPA 拡張

プロパティ	使用方法	デフォルト
toplink.ddl-generation	JPA エンティティに対するデータ定義言語（DDL）の生成アクションを指定します。DDL 生成ターゲットを指定する方法は、toplink.ddl-generation.output-mode を参照してください。 **有効な値:** oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider ■ none: DDL を生成しません。スキーマは生成されません。 ■ create-tables: 存在しない表に対して DDL を作成します。既存の表は変更されません（toplink.create-ddl-jdbc-file-name も参照）。 ■ drop-and-create-tables: すべての表に対して DDL を作成します。既存の表はすべて削除されます（toplink.create-ddl-jdbc-file-name および toplink.drop-ddl-jdbc-file-name も参照）。 **例:** persistence.xml ファイル <property name="toplink.ddl-generation" value="create-tables"/> **例:** プロパティ Map import oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider; propertiesMap.put(EntityManagerFactoryProvider.DDL_GENERATION, EntityManagerFactoryProvider.CREATE_ONLY);	none
toplink.application-location	生成された DDL ファイルを TopLink が書き込む場所を指定します（toplink.create-ddl-jdbc-file-name および toplink.drop-ddl-jdbc-file-name を参照）。ファイルが書き込まれるのは、toplink.ddl-generation が none 以外に設定されている場合です。 **有効な値:** 書込みアクセス権のあるディレクトリに対するファイル指定。このファイル指定は、現在の作業ディレクトリに対する相対パスか、または絶対パスとすることが可能です。ファイル指定がファイル・セパレータで終了していない場合、使用中のオペレーティング・システムに有効なセパレータが TopLink により追加されます。 **例:** persistence.xml ファイル <property name="toplink.application-location" value="C:¥ddl¥"/> **例:** プロパティ Map import oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider; propertiesMap.put(EntityManagerFactoryProvider.APP_LOCATION, "C:¥ddl¥");	"."+File.separator

表 26-6 スキーマ生成用の TopLink JPA 拡張（続き）

プロパティ	使用方法	デフォルト

toplink.create-ddl-jdbc-file-name	TopLink により生成される、JPA エンティティ用の表を作成するための SQL 文を含む SQL ファイルの名前を指定します。このファイルは、toplink.ddl-generation が create-tables または drop-and-create-tables に設定されている場合に、toplink.application-location によって指定された場所に書き込まれます。
	有効な値 : 使用中のオペレーティング・システムに有効なファイル名。toplink.application-location + toplink.create-ddl-jdbc-file-name の連結が使用中のオペレーティング・システムに有効なファイル指定である場合は、オプションでファイル名の先頭にファイル・パスを追加できます。
	例 : persistence.xml ファイル
	`<property name="toplink.create-ddl-jdbc-file-name" value="create.sql"/>`
	例 : プロパティ Map
	`import oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider; propertiesMap.put(EntityManagerFactoryProvider.CREATE_JDBC_DDL_FILE, "create.sql");`	

toplink.drop-ddl-jdbc-file-name	TopLink により生成される、JPA エンティティ用の表を削除するための SQL 文を含む SQL ファイルの名前を指定します。このファイルは、toplink.ddl-generation が drop-and-create-tables に設定されている場合に、toplink.application-location によって指定された場所に書き込まれます。
	有効な値 : 使用中のオペレーティング・システムに有効なファイル名。toplink.application-location + toplink.drop-ddl-jdbc-file-name の連結が使用中のオペレーティング・システムに有効なファイル指定である場合は、オプションでファイル名の先頭にファイル・パスを追加できます。
	例 : persistence.xml ファイル
	`<property name="toplink.drop-ddl-jdbc-file-name" value="drop.sql"/>`
	例 : プロパティ Map
	`import oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider; propertiesMap.put(EntityManagerFactoryProvider.DROP_JDBC_DDL_FILE, "drop.sql");`	

表 26-6 スキーマ生成用の TopLink JPA 拡張（続き）

プロパティ	使用方法	デフォルト
toplink.ddl-generation.output-mode	このプロパティを使用して、DDL 生成ターゲットを指定します。 **有効な値：** oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider ■ both: SQL ファイルを生成し、データベースで実行します。 toplink.ddl-generation が create-tables に設定されている場合、toplink.create-ddl-jdbc-file-name が toplink.application-location に書き込まれ、データベースで実行されます。 toplink.ddl-generation が drop-and-create-tables に設定されている場合、toplink.create-ddl-jdbc-file-name と toplink.drop-ddl-jdbc-file-name の両方が toplink.application-location に書き込まれ、両方の SQL ファイルがデータベースで実行されます。 ■ database: データベースで SQL を実行するのみです（SQL ファイルは生成しません）。 toplink.ddl-generation が create-tables に設定されている場合、toplink.create-ddl-jdbc-file-name がデータベースで実行されます。toplink.application-location には書き込まれません。 toplink.ddl-generation が drop-and-create-tables に設定されている場合、toplink.create-ddl-jdbc-file-name と toplink.drop-ddl-jdbc-file-name の両方がデータベースで実行されます。どちらも toplink.application-location には書き込まれません。 ■ sql-script: SQL ファイルを生成するのみです（データベースでは実行しません）。 toplink.ddl-generation が create-tables に設定されている場合、toplink.create-ddl-jdbc-file-name が toplink.application-location に書き込まれます。データベースでは実行されません。 toplink.ddl-generation が drop-and-create-tables に設定されている場合、toplink.create-ddl-jdbc-file-name と toplink.drop-ddl-jdbc-file-name の両方が toplink.application-location に書き込まれます。どちらもデータベースでは実行されません。 **例：**persistence.xml ファイル <property name="toplink.ddl-generation.output-mode" value="database"/> **例：**プロパティ Map import oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider; propertiesMap.put(EntityManagerFactoryProvider.DDL_GENERATION_MODE, EntityManagerFactoryProvider.DDL_DATABASE_GENERATION);	Java EE モード（createContainerEntityManagerFactory がコールされる場合）：both Java SE モード（createEntityManagerFactory がコールされる場合）：sql-script

27

EJB アプリケーションのパッケージ化

この項の内容は次のとおりです。

- JPA エンティティ・アプリケーションのパッケージ化
- EJB 3.0 と EJB 2.1 の両方の Enterprise Bean があるアプリケーションのパッケージ化
- EJB アプリケーション間でのクラスの共有

詳細は、次を参照してください。

- 『Oracle Application Server エンタープライズ・デプロイメント・ガイド』
- 2-4 ページの「パッケージ化について」

JPA エンティティ・アプリケーションのパッケージ化

EJB 3.0 エンティティを使用するアプリケーションをパッケージ化する場合、次の点を考慮します。

- 永続性ユニットのパッケージ化
- マッピング・メタデータのパッケージ化

永続性ユニットのパッケージ化

EJB 3.0 JPA 永続性ユニットは、`persistence.xml` ファイル、1 つ以上のオプションの `orm.xml` ファイル、および永続性ユニットに属する管理エンティティ・クラスで構成されます。

永続性ユニットを独自の永続性アーカイブにパッケージ化し、永続性ユニットへのアクセスを必要とする任意の Java EE モジュールにそのアーカイブを含めることができます（27-2 ページの「永続性アーカイブの作成」を参照）。または、永続性ユニット・ファイルを様々な Java EE モジュールに直接パッケージ化することも可能です（27-3 ページの「Java EE モジュールへの永続性ユニット・ファイルの直接パッケージ化」を参照）。

`META-INF` ディレクトリに `persistence.xml` ファイルを含む JAR ファイルまたはディレクトリは、永続性ユニットのルートと呼ばれます。エンティティを使用する EJB 3.0 アプリケーションでは、少なくとも 1 つの永続性ユニット・ルートを定義する必要があります。

永続性ユニットの有効範囲は、永続性ユニット・ルートを定義する場所によって決まります。

詳細は、次を参照してください。

- 2-10 ページの「persistence.xml ファイルとは」
- 26-4 ページの「persistence.xml ファイルの構成」

永続性アーカイブの作成

永続性アーカイブは、例 27-1「永続性アーカイブ」に示すように、`persistence.xml` ファイル、1 つ以上のオプションの `orm.xml` ファイル、および永続性ユニットに属する管理エンティティ・クラスを含む単純な JAR ファイルです。

例 27-1 永続性アーカイブ

```
employee-persistence.jar
    META-INF/persistence.xml
    META-INF/orm.xml
    com/acme/model/Employee.class
    com/acme/model/Address.class
    ...
```

永続性アーカイブは、次のいずれかの方法でパッケージ化します。

- WAR: `WEB-INF/lib` ディレクトリ。永続性ユニットは、この WAR 内のクラスからのみアクセスできます。
- EAR: ルートまたはアプリケーション・ライブラリ・ディレクトリ。永続性ユニットは、すべてのアプリケーション・コンポーネントからアクセスできます。

永続性アーカイブを使用すると、複数の Java EE モジュールで永続性ユニットを簡単に共有できます。

Java EE モジュールへの永続性ユニット・ファイルの直接パッケージ化

永続性ユニット・ファイルは、次のいずれかの Java EE モジュールにパッケージ化できます。

- EJB-JAR ファイル
- WAR ファイル
  - `WEB-INF/classes` ディレクトリ
  - `WEB-INF/lib`（この場合、`persistence.xml` ファイルは JAR にあることが必要）
- EAR
  - EAR のルートにある JAR 内の `persistence.xml` ファイル
  - EAR ライブラリ・ディレクトリにある JAR 内の `persistence.xml` ファイル
- アプリケーション・クライアント JAR

表 27-1 OC4J による META-INF および WEB-INF の persistence.xml の処理方法

WEB-INF/classes/META-INF	WEB-INF	OC4J が persistence.xml を使用する場所
`persistence.xml`	`persistence.xml`	`META-INF`（`WEB-INF` 内のファイルを無視）
	`persistence.xml`	`META-INF`
`persistence.xml`		`WEB-INF`

永続性ユニット・ファイルと Java EE モジュールを分離し、複数の Java EE モジュールで永続性ユニットを簡単に共有するには、永続性ユニットを永続性アーカイブにパッケージ化することを検討してください（27-2 ページの「永続性アーカイブの作成」を参照）。

マッピング・メタデータのパッケージ化

EJB 3.0 JPA マッピング・メタデータは、アノテーションまたは 1 つ以上のオプションの `orm.xml` ファイル（あるいはその両方）を使用して指定できます。`orm.xml` ファイルは、次のいずれかにパッケージ化できます。

- 永続性ユニット・ルート（`META-INF` ディレクトリに `persistence.xml` ファイルを含む JAR ファイルまたはディレクトリ）の `META-INF` ディレクトリ
- `persistence.xml` ファイルで参照される任意の JAR ファイルの `META-INF` ディレクトリ
- `persistence.xml` ファイルの `<persistence-unit>` 要素の `<mapping-file>` サブ要素
- 永続性アーカイブ

詳細は、次を参照してください。

- 2-11 ページの「orm.xml ファイルとは」
- 27-2 ページの「永続性ユニットのパッケージ化」
- 27-2 ページの「永続性アーカイブの作成」

EJB 3.0 と EJB 2.1 の両方の Enterprise Bean があるアプリケーションのパッケージ化

EJB 3.0 と EJB 2.1 の両方の Bean をアプリケーションで組み合せることができます。たとえば、`ejb-jar.xml` ファイルのない 3 つのアノテーション付き EJB 3.0 エンティティ、`ejb-jar.xml` ファイルのある 2 つの EJB 2.1 エンティティ Bean、および `ejb-jar.xml` ファイル、アノテーションまたはその両方のある 3 つの EJB 3.0 セッション Bean（この場合、`ejb-jar.xml` はアノテーションをオーバーライドする）を含むアプリケーションが可能です。

EJB アプリケーション間でのクラスの共有

Enterprise Bean 間でクラスを共有する場合は、次のいずれかを実行します。

- 2 つの Enterprise Bean が同じクラスを使用する場合は、すべてのクラスと Enterprise Bean を同じ JAR ファイルに含めます。デプロイ後、両方の Enterprise Bean が同じ共通クラスを使用できるようになります。
- 共有クラスを、アプリケーションのそれぞれの JAR ファイルに配置します。次のように、EJB JAR `manifest.mf` ファイルの `class-path` にある共有 JAR ファイルを参照します。

  ```
  Class-Path:shared_classes.jar
  ```

  `shared_classes.jar` の場所は、これを参照する JAR ファイルが EAR ファイル内のどこに存在するかによって異なります。この例では、`shared_classes.jar` ファイルは、EJB JAR ファイルと同じレベルに存在します。
- すべてのアプリケーションがこれらのクラスを参照するようにする場合は、共有クラスを JAR ファイルにアーカイブし、この JAR ファイルをデフォルト・アプリケーションの共有ライブラリ・ディレクトリに配置します。デフォルトの共有ライブラリは `home/lib` です。ただし、共有ライブラリ・ディレクトリは、Enterprise Manager を使用してデフォルト・アプリケーションの「一般プロパティ」ページで設定できます。
- 特定のアプリケーションのみがこれらのクラスを参照するようにする場合は、共有クラスをそれぞれの共有クラス自身のアプリケーションにアーカイブし、アプリケーションの EAR ファイルをデプロイし、共有クラスを参照するアプリケーションで、その共有クラス・アプリケーションを親として宣言します。OracleAS のデフォルトの親は、デフォルト・アプリケーションです。

  子アプリケーションは、親アプリケーションの名前空間を認識します。これは、Enterprise Bean などのサービスを複数のアプリケーションで共有するために使用されます。親アプリケーションの指定方法は、『Oracle Containers for J2EE 開発者ガイド』を参照してください。

EJB アプリケーションと Web アプリケーション間でクラスを共有する場合は、参照されるクラスを共有 JAR ファイルに配置する必要があります。

EJB アプリケーション間でクラスを共有する場合、次の問題に注意してください。

- 実行時のメモリー不足例外の処理
- 実行時のクラス・キャスト例外の処理

実行時のメモリー不足例外の処理

実行時に OC4J メモリーが一貫して増大し続ける場合は、`application.xml` ファイルに無効なシンボリック・リンクが指定されている可能性があります。OC4J は、この `application.xml` ファイルのリンクを使用して、すべてのリソースをロードします。これらのリンクが無効な場合は、C ヒープの増大が続き、OC4J がメモリー不足となります。すべてのシンボリック・リンクが有効であることを確認し、OC4J を再起動します。

また、シンボリック・リンクが指し示すディレクトリ内の JAR ファイル数は最小限にしてください。使用していない JAR ファイルすべてをこれらのディレクトリから削除します。OC4J は、クラスとリソースの JAR をすべて検索します。したがって、アドレス空間へのマッピングおよびファイル・キャッシュによる時間やメモリーの消費量が増加します。

実行時のクラス・キャスト例外の処理

実行時に ClassCastException が発生した場合は、次の状況であることが考えられます。

- 開発を容易にするために、サーブレットが存在している WAR ファイルに EJB インタフェースをコピーしていて、WAR ファイルを作成する前にインタフェースの削除を忘れていた場合。**さらに**
- orion-web.xml ファイルで、<web-app-class-loader> 要素の search_local_classes_first 属性をオンにしていた場合。

この問題を解決するには、コピーしたクラスを WAR ファイルから削除するか、または search_local_classes_first 属性をオフにします。この属性をオンにすると、クラス・ローダーは、EJB JAR ファイル内のクラスなどの他のクラスをロードする前に、WAR ファイル内のクラスをロードします。この属性の詳細は、『Oracle Containers for J2EE サーブレット開発者ガイド』の「サーブレットの開発」の、OC4J でのシステム・クラスより前の WAR ファイル・クラスのロードに関する項を参照してください。

他の共有 EJB クラスを参照する EJB または Web アプリケーションを使用する場合は、参照されるクラスを共有 JAR ファイルに配置する必要があります。状況によっては、WAR ファイルまたは共有 EJB クラスを参照する別のアプリケーションに共有 EJB クラスをコピーすると、クラス・ローダーの問題のために ClassCastException が発生する場合があります。正常に終了するためには、参照される EJB クラスを、そのアプリケーションの WAR ファイルまたは別のアプリケーションにコピーしないでください。

28

OC4J への EJB アプリケーションのデプロイ

この項の内容は次のとおりです。

詳細は、次を参照してください。

> **注意：** Application Server Control を使用している場合、セッション Bean とともにデプロイされる EJB 3.0 エンティティは、EJB JAR モジュールの Application Server Control ビューには表示されません。詳細は、31-2 ページの「Oracle Enterprise Manager 10g Application Server Control の使用方法」を参照してください。

- 『Oracle Application Server エンタープライズ・デプロイメント・ガイド』
- 2-4 ページの「デプロイについて」

大規模な EJB アプリケーションのデプロイ

この項の内容は次のとおりです。

詳細は、『Oracle Containers for J2EE デプロイメント・ガイド』の大規模アプリケーションのデプロイに関する項を参照してください。

デプロイ時のメモリー不足エラーを回避するための VM の調整

非常に大きなアプリケーション（EAR）が OC4J にデプロイされる場合、デプロイ時に `OutOfMemory` 例外がスローされることがあります。

VM ヒープおよび永続領域構成により、このような例外が発生することがあります。デフォルトでは、ヒープおよび永続領域は 64MB に設定されます。

ヒープ領域の問題がある場合は、ヒープ領域を `java -Xmx750m -Xms512m` と指定する必要があります。

永続領域の問題がある場合は、永続領域を `java -Xmx750m -Xms512m -XX:PermSize=128m -XX:MaxPermSize=256m` と指定する必要があります。

デプロイ時のメモリー不足エラーを回避するための一時ディレクトリの構成

デプロイ・プロセスがなんらかの理由で中断された場合、`temp` ディレクトリのクリーンアップが必要となる場合があります。このディレクトリは、デフォルトでシステム上の `/var/tmp` です。デプロイ・ウィザードによって、デプロイ・プロセス時に情報を格納するために、一時ディレクトリのスワップ領域で 20MB が使用されます。プロセス完了時に、`temp` ディレクトリから追加のファイルがクリーンアップされます。ただし、ウィザードが中断されると、`temp` ディレクトリをクリーンアップする時間や機会がない場合があります。したがって、このディレクトリから追加のデプロイメント・ファイルを手動でクリーンアップする必要があります。クリーンアップを実行しないと、ディレクトリがいっぱいになる可能性があり、その後のデプロイができなくなります。`OutOfMemory` 例外を受信した場合は、`temp` ディレクトリの使用可能領域をチェックしてください。

`temp` ディレクトリを変更するには、OC4J プロセスのコマンドライン・オプションを `java.io.tmpdir=<new_tmp_dir>` に設定します。このコマンドライン・オプションは「サーバー・プロパティ」ページで設定できます。最初に OC4J のホームページにドリルダウンします。次に「管理」セクションにスクロールダウンします。「サーバー・プロパティ」を選択します。このページで、「コマンドライン・オプション」セクションにスクロールダウンし、「OC4J オプション」行に `java.io.tmpdir` の変数定義を追加します。新規 OC4J プロセスはすべて、このプロパティを使用して起動されます。

デプロイ時のメモリー不足エラーを回避するためのバッチ・コンパイルの無効化

アプリケーション（EAR）に複数の JAR ファイルが含まれている場合は、OutOfMemory 例外を修正するようにバッチ・デプロイを無効にできます。ただし、EAR ファイルに JAR ファイルが 1 つのみ含まれている場合、このアプローチではこのような例外が修正されません。この場合は、VM を調整する必要があります（28-2 ページの「デプロイ時のメモリー不足エラーを回避するための VM の調整」を参照）。

OC4J がデプロイ時に OutOfMemory 不足例外をスローし、VM の調整（28-2 ページの「デプロイ時のメモリー不足エラーを回避するための VM の調整」を参照）と一時ディレクトリの調整（28-2 ページの「デプロイ時のメモリー不足エラーを回避するための一時ディレクトリの構成」を参照）をすでに試した場合は、非バッチ・モードでコンパイルを試すこともできます。非バッチ・モードでは必要なメモリー量が少なくなりますが、このモードではデプロイ時間が長くなります。

バッチ・コンパイルを有効または無効にするには、<application> または <orion-application> 要素の属性 batch-compile を使用します。

batch-compile のデフォルト値は true です。

バッチ・コンパイルを無効にするには、この属性を false に設定します。

例 28-1 に、orion-applicatin.xml デプロイメント・ディスクリプタでこの属性を構成する方法を示します。

例 28-1 orion-application.xml ファイルでのバッチ・コンパイルの無効化

```
<orion-application batch-compile ="false">
...
</orion-application>
```

メモリー不足エラーが解決しない場合は、バッチ・コンパイルの無効化を試します。

増分デプロイ

OC4J では、デプロイされたアプリケーションの一部である EJB モジュールの増分または部分的な再デプロイがサポートされます。この機能により、モジュール全体を再デプロイする必要なく、EJB JAR 内の変更された Bean のみ再デプロイできます。以前にデプロイされ、変更されていない Bean は、引き続き使用されます。

この機能は、OC4J の以前のリリースよりも大幅に拡張されています。以前のリリースでは、EJB モジュールが 1 つの単位として扱われ、最初にモジュールをアンデプロイしてから更新とともに再デプロイする必要がありました。

OC4J の再起動は、再デプロイ・プロセス中に EJB 構成データに変更が行われた場合にのみ必要です。変更が行われていない場合は、OC4J を再起動せずにホット・デプロイを実行できます。

増分再デプロイ操作は、Enterprise Bean を含む更新対象のアプリケーションを自動的に停止してから、終了時にアプリケーションを自動的に再起動します。

注意： 再デプロイ時に、更新される Enterprise Bean へのすべてのアイドル・クライアント接続は失われます。すべての既存のリクエストは完了できますが、新しいリクエストはアプリケーションが再起動するまで許可されません。Enterprise Bean を再デプロイする前に、アプリケーションを停止することを強くお薦めします。

CMP または BMP エンティティ Bean の場合、OC4J はコード生成機能を使用して EJB インタフェースのサーバー実装（ラッパー）を生成します。この場合、一般的には、アプリケーション全体を再デプロイするよりも変更された Bean のみを増分再デプロイする方が効率的です。

セッション Bean、メッセージドリブン Bean および EJB 3.0 JPA エンティティの場合、OC4J はバイト・コード生成機能を使用してラッパーを生成します。この方法ではデプロイ時間が大幅

に短縮されるため、アプリケーション全体を再デプロイしても変更された Bean のみを再デプロイしても効率的には同じになる可能性があります。この場合、増分再デプロイの実行は任意です。

増分デプロイを使用するための一般的な手順は、次のとおりです。

1. 多数の Enterprise Bean があるアプリケーションをデプロイします。
2. EJB モジュールで Bean 関連のクラス・ファイルを変更し、EJB JAR ファイル（`myBeans-ejb.jar` など）を再構築します。
3. 次のいずれかを使用して、更新された EJB JAR を OC4J に発行します。
   - JDeveloper
   - Enterprise Manager
   - `<OC4J_HOME>¥j2ee¥home¥admin.jar` または `admin_client.jar`（`updateEBJModule` コマンドを使用）

   例 28-2 に、`admin.jar` の使用方法を示します。

例 28-2 admin.jar を使用した増分デプロイ

```
java -jar admin.jar ormi://localhost:23791 admin welcome -application -updateEJBModule -jar
myBeans-ejb.jar
```

4. 手順 2 と 3 を繰り返します。

詳細は、『Oracle Containers for J2EE デプロイメント・ガイド』の更新された EJB モジュールの増分再デプロイに関する項を参照してください。

展開デプロイ

通常、アプリケーションは、OC4J にデプロイする前に EAR ファイルにパッケージ化します。ただし、アプリケーションは、展開ディレクトリ構造に配置したままの状態でデプロイできます。この方法は、パッケージ化の手順を省略できるため、デプロイ時とテスト時に役立ちます。たとえば、JDeveloper を使用して、展開ディレクトリ構造のアプリケーションを操作できます。展開デプロイを使用すると、各デプロイの前に再アーカイブすることなく、何度でも展開ディレクトリ構造をデプロイできます。

展開デプロイ用に OC4J を構成するには、*<OC4J_HOME>*`¥j2ee¥home¥config¥server.xml` ファイルを編集します。使用するアプリケーションの `application` 要素を変更し、`path` にそのアプリケーションの展開ディレクトリのルートを指定します。例 28-3 に、`path` 属性を展開ディレクトリのルートに設定したアプリケーション `myapp` の `application` 要素を示します。

例 28-3 展開デプロイ用の server.xml

```
<application-server  ...>
    ...
    <!-- Regular EAR deployment -->
    <application name="app" path="../../home/applications/app.ear" start="true" />

    <!-- Expanded deployment -->
    <application name="myapp" path="C:/projects/myapp" start="true" />
    ...
</application-server>
```

アプリケーション・デプロイのトラブルシューティング

1 つ以上のアノテーションのある EJB 3.0 アプリケーションのデプロイ時に、OC4J はそのメモリー内の ejb-jar.xml ファイルをデプロイ・ディレクトリ内の orion-ejb-jar.xml ファイルと同じ場所（<*ORACLE_HOME*>/j2ee/home/application-deployments/my_application/META-INF）に自動的に書き込みます。

この ejb-jar.xml ファイルは、アノテーションとデプロイ済 ejb-jar.xml ファイル（存在する場合）の両方から取得された構成を表します。

EJB 2.1 アプリケーションをデプロイする場合、生成されたラッパー・コードを保持するには、システム・プロパティ KeepWrapperCode を設定する必要があります（31-10 ページの「生成されたラッパー・コードのデバッグ」を参照）。

詳細は、31-9 ページの「EJB アプリケーションのトラブルシューティング」を参照してください。

第 X 部

アプリケーションでの EJB の使用方法

第 X 部では、J2EE アプリケーションで EJB 3.0 または EJB 2.1 Enterprise JavaBeans を使用する手順に関する情報を示します。概念的な情報は、第 I 部「EJB の概要」を参照してください。

第 X 部は次の各章で構成されています。

- 第 29 章「クライアントからの Enterprise Bean へのアクセス」
- 第 30 章「EJB および Web サービスの使用方法」
- 第 31 章「EJB アプリケーションの管理」
- 第 32 章「EJB パフォーマンスの最適化」

29

クライアントからの Enterprise Bean へのアクセス

この章では、次のようなクライアントからの EJB へのアクセス方法を説明します。

- 使用しているクライアントのタイプ
- クライアントの構成
- EJB 3.0 Enterprise Bean へのアクセス
- 別のアプリケーションの EJB 3.0 Enterprise Bean へのアクセス
- EntityManager を使用した JPA エンティティへのアクセス
- EJB 3.0 を使用した JMS 宛先へのメッセージの送信
- EJB 3.0 EJBContext へのアクセス
- EJB 2.1 Enterprise Bean へのアクセス
- 別のアプリケーションの EJB 2.1 Enterprise Bean へのアクセス
- EJB 2.1 を使用した JMS 宛先へのメッセージの送信
- EJB 2.1 EJBContext へのアクセス
- パラメータの処理
- 例外の処理

詳細は、次を参照してください。

- 2-12 ページの「アプリケーションでの Enterprise Bean の使用方法」
- 19-24 ページの「EJB 3.0 リソース・マネージャのコネクション・ファクトリのルックアップ」
- 19-25 ページの「EJB 3.0 環境変数のルックアップ」
- 19-26 ページの「EJB 2.1 リソース・マネージャのコネクション・ファクトリのルックアップ」
- 19-27 ページの「EJB 2.1 環境変数のルックアップ」

注意： EJB コード例は、`http://www.oracle.com/technology/tech/java/oc4j/demos` からダウンロードできます。

使用しているクライアントのタイプ

Enterprise Bean には、次のような様々なクライアントからアクセスできます。

- EJB クライアント
- スタンドアロン Java クライアント
- サーブレットまたは JSP クライアント

Enterprise Bean、リソースまたは環境変数へのアクセス方法は、クライアントのタイプおよびアプリケーションのアセンブルとデプロイの方法によって決まります。

詳細は、29-3 ページの「クライアントの構成」を参照してください。

EJB クライアント

1 つの Enterprise Bean（ソース Enterprise Bean と呼ぶ）が別の Enterprise Bean（ターゲット Enterprise Bean と呼ぶ）にアクセスする場合、ソース Enterprise Bean はターゲット Enterprise Bean のクライアントです。

EJB 3.0 を使用している場合は、アノテーションおよび依存性注入を通じて、OC4J はターゲット参照に対応するインスタンス変数を初期化します。

EJB 2.1 を使用している場合、このシナリオでは JNDI ルックアップを使用する必要があります。

スタンドアロン Java クライアント

スタンドアロン Java クライアントは、OC4J の外部で実行し、OC4J にデプロイされる EJB リソースにアクセスするクライアントです。

通常、スタンドアロン Java クライアントは、Java RMI コールを利用して EJB リソースにアクセスします。スタンドアロン Java クライアントは、OC4J で実施されるセキュリティおよび認証の要件を満たすようにコーディングする必要があります。

デフォルトでは、OC4J は設定範囲内で RMI ポートを動的に割り当てるように構成されます。このリリースでは、厳密な RMI ポートを指定せずに、スタンドアロン Java クライアントから OC4J でデプロイされる Enterprise Bean をルックアップできます。厳密なポート番号を使用するように OC4J を構成する必要はありません。

EJB 3.0 を使用している場合、スタンドアロン Java クライアントではアノテーションおよび依存性注入がサポートされないことに注意してください。

EJB 2.1 を使用している場合は、このシナリオに適応するように初期コンテキストを構成する必要があります（29-24 ページの「RMI を使用した、スタンドアロン Java クライアントからの EJB 2.1 Enterprise Bean へのアクセス」を参照）。

サーブレットまたは JSP クライアント

サーブレットまたは JSP は Enterprise Bean にアクセスできます。

このリリースでは、OC4J により Web 層でのアノテーションおよびリソース・インジェクションがサポートされます（1-9 ページの「Web 層でのアノテーション」を参照）。

依存性注入は、EJB 3.0 アプリケーションでサーブレットまたは JSP クライアントから使用できます。

JNDI ルックアップは、EJB 3.0 アプリケーションと EJB 2.1 アプリケーションの両方でサーブレットまたは JSP クライアントから使用できます。

クライアントの構成

クライアントから Enterprise Bean にアクセスする前に、次の点を考慮する必要があります。

- OC4J のクライアント・クラスパスの構成
- 初期コンテキスト・ファクトリ・クラスの選択
- セキュリティ資格証明の指定
- EJB 参照の選択

OC4J のクライアント・クラスパスの構成

表 29-1 に、クライアントのルックアップ対象に応じてクライアントにインストールする必要のある OC4J 固有の JAR ファイルをリストします。「ソース」列は、*<OC4J_HOME>* から必要な JAR のコピーを取得する場所を示します。

クライアント・クラスパスに含める必要があるのは、`oc4jclient.jar` のみです。クライアントにより必要とされる他のすべての JAR ファイルは、`oc4jclient.jar` のマニフェスト・クラスパスで参照されます。

表 29-1 OC4J クライアントのクラスパス要件

OC4J JAR	ソース（<OC4J_HOME> に対する相対パス）	EJB ルックアップ	JMS コネクタ・ルックアップ	OEMS JMS ルックアップ	OEMS JMS データベース・ルックアップ
adminclient.jar	/j2ee/<*instance*>/lib		✔		✔
bcel.jar	/j2ee/<*instance*>/lib				✔
connector.jar	/j2ee/<*instance*>/lib		✔		
dms.jar	/j2ee/<*instance*>/lib				✔
ejb.jar	/j2ee/<*instance*>/lib	✔			✔
javax77.jar	/j2ee/<*instance*>/lib		✔	✔	✔
jazncore.jar	/j2ee/<*instance*>		✔		
jdbc.jar	/j2ee/<*instance*>/../../lib				
jms.jar	/j2ee/<*instance*>/lib		✔	✔	✔
jmxri.jar	/j2ee/<*instance*>/lib		✔		
jndi.jar	/j2ee/<*instance*>/lib		✔	✔	✔
jta.jar	/j2ee/<*instance*>/lib		✔	✔	✔
oc4j.jar	/j2ee/<*instance*>		✔		
oc4jclient.jar	/j2ee/<*instance*>	✔	✔	✔	✔
oc4j-internal.jar	/j2ee/<*instance*>/lib		✔		
ojdbc14dms.jar	/j2ee/<*instance*>/../../oracle/jdbc/lib				✔
optic.jar[1]	/opmn/lib			✔	✔

[1] `Context.PROVIDER_URL` での JNDI ルックアップで `opmn:ormi` 接頭辞を使用することを計画している場合にのみ必要です（19-21 ページの「Oracle 初期コンテキスト・ファクトリの構成」を参照）。

これらの JAR ファイルのいずれかをブラウザにダウンロードする場合は、特定の権限を付与する必要があります（22-2 ページの「ブラウザにおける権限の付与」を参照）。

初期コンテキスト・ファクトリ・クラスの選択

初期コンテキスト・ファクトリを使用して、初期コンテキスト（JNDI ネームスペースへの参照）を取得します。初期コンテキストを使用すると、JNDI API を使用して Enterprise Bean、リソース・マネージャのコネクション・ファクトリ、環境変数、または JNDI でアクセス可能なその他のオブジェクトをルックアップできます。使用する初期コンテキスト・ファクトリのタイプは、使用しているクライアント・タイプおよび OC4J の使用方法（スタンドアロンまたは Oracle Application Server の一部として）によって決まります（19-20 ページの「初期コンテキスト・ファクトリの構成」を参照）。

セキュリティ資格証明の指定

クライアントとターゲット Enterprise Bean が同一 JVM 上になく、同じアプリケーションにデプロイされず、ターゲット EJB アプリケーションがクライアントの親でない場合、クライアントでは、ターゲット Enterprise Bean にアクセスする前に資格証明を指定する必要があります（22-11 ページの「EJB クライアントの資格証明の指定」を参照）。

EJB 参照の選択

EJB 3.0 では、EJB クライアントで EJB 3.0 Enterprise Bean またはリソースにアクセスするために、事前定義の環境参照で JNDI ルックアップを行うかわりに、アノテーション、リソース・インジェクションおよびデフォルトの JNDI 名（クラス名およびインタフェース名に基づく）を使用できます。

EJB 2.1 または EJB 3.0（スタンドアロン Java クライアントの場合）では、Enterprise Bean またはリソースにアクセスするには、事前定義の環境参照で JNDI ルックアップを行う必要があります（19-2 ページの「環境参照の構成」を参照）。EJB 2.1 Enterprise Bean またはリソースにアクセスするには、適切な事前定義環境参照（実際または論理、ローカルまたはリモート）を選択し、JNDI 初期コンテキスト（29-4 ページの「初期コンテキスト・ファクトリ・クラスの選択」を参照）を使用してそれをルックアップします。

クライアント実装からの参照により Enterprise Bean にアクセスする場合は、EJB デプロイメント・ディスクリプタで定義されている `<ejb-ref-name>` を使用して JNDI ルックアップを実行します。ターゲット Enterprise Bean への EJB 参照の定義の詳細は、19-2 ページの「EJB 環境参照」を参照してください。

表 29-2 に、参照に `java:comp/env/ejb/` という接頭辞を付ける場合を示します。これは、コンテナがデプロイメント・ディスクリプタで定義された EJB 参照を入れる場所です。

表 29-2 java:comp/env/ejb/ 接頭辞を使用する場合

クライアント	初期コンテキスト・ファクトリ	接頭辞の使用方法
EJB クライアント	デフォルト `RMIInitialContext`	オプション 未使用
スタンドアロン Java クライアント	デフォルト `ApplicationClientInitialContext`	オプション 必須
サーブレットまたは JSP クライアント	デフォルト `RMIInitialContext`	オプション 未使用

例 29-1 に、java:comp/env/ejb/ 接頭辞を使用して論理名 ejb/HelloWorld で Enterprise Bean をルックアップする方法を示し、例 29-2 に、接頭辞を使用せずにこの Enterprise Bean をルックアップする方法を示します。

例 29-1 接頭辞を使用した Enterprise Bean のルックアップ

```
InitialContext ic = new InitialContext();
HelloHome hh = (HelloHome)ic.lookup("java:comp/env/ejb/HelloWorld");
```

例 29-2 接頭辞を使用しない Enterprise Bean のルックアップ

```
InitialContext ic = new InitialContext();
HelloHome hh = (HelloHome)ic.lookup("ejb/HelloWorld");
```

EJB 3.0 Enterprise Bean へのアクセス

JNDI から Bean インスタンスを直接ルックアップ（または EJB 3.0 EJB クライアントでリソース・インジェクションを使用）し、ホーム・インタフェースを使用せずに Bean インスタンスを取得できます。<home> または <local-home> 要素が EJB 参照から削除される場合、Bean インスタンスはホームではなく JNDI から返されます。

Bean インスタンスは、ホーム・インタフェースで引数なしの create メソッドを実行することで作成されます。ステートフル・セッション Bean およびエンティティ Bean でもこのショートカットを使用できますが、これらの Bean には引数なしの create メソッドが必要であり、ない場合はルックアップ時に例外がスローされます。

どちらの場合も、EJB ビジネス・インタフェースへの参照の取得で使用される構文は、ビジネス・インタフェースがローカルとリモートのどちらであるかに依存しません。リモート・アクセスの場合、参照される Enterprise Bean および EJB コンテナの実際の場所は、一般に Bean のリモート・ビジネス・インタフェースを使用しているクライアントに対して透過的です。

EJB 3.0 を使用している場合は、リソース・インジェクション（29-5 ページの「アノテーションの使用方法」を参照）または InitialContext（29-6 ページの「初期コンテキストの使用方法」を参照）を使用して Enterprise Bean をルックアップできます。

別の方法として、OC4J 固有のアノテーションまたはデプロイ XML を使用して EJB 3.0 への環境参照を定義できます（19-2 ページの「EJB 環境参照」を参照）。

アノテーションの使用方法

例 29-3 に、アノテーションおよび依存性注入を使用して EJB クライアントから EJB 3.0 Enterprise Bean にアクセスする方法を示します。

例 29-3 EJB 3.0 EJB クライアントでの EJB 3.0 Enterprise Bean の注入

```
@EJB AdminService bean;

    public void privilegedTask() {
        bean.adminTask();
    }
```

初期コンテキストの使用方法

この項の内容は次のとおりです。

- ejb-ref を使用した EJB 3.0 Enterprise Bean のリモート・インタフェースのルックアップ
- location を使用した EJB 3.0 Enterprise Bean のリモート・インタフェースのルックアップ
- local-ref を使用した EJB 3.0 Enterprise Bean のローカル・インタフェースのルックアップ
- local-location を使用した EJB 3.0 Enterprise Bean のローカル・インタフェースのルックアップ

詳細は、19-20 ページの「初期コンテキスト・ファクトリの構成」を参照してください。

ejb-ref を使用した EJB 3.0 Enterprise Bean のリモート・インタフェースのルックアップ

`ejb-ref` を使用して Enterprise Bean のリモート・インタフェースをルックアップするには、次のようにします。

1. `ejb-jar.xml` ファイルで Enterprise Bean の `ejb-ref` 要素を定義します。

例 29-4 ejb-ref 要素の ejb-jar.xml

```
<ejb-ref>
    <ejb-ref-name>ejb/Test</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local>Test</local>
</ejb-ref>
```

詳細は、19-4 ページの「リモート EJB への環境参照の構成 : クラスタ化または結合された Web 層および EJB 層」を参照してください。

2. 接頭辞が必要かどうかを判断します（29-4 ページの「EJB 参照の選択」を参照）。
3. `ejb-ref-name` 要素および適切な接頭辞（必要な場合）を使用して Enterprise Bean をルックアップします。

例 29-5 初期コンテキストを使用した EJB 3.0 EJB クライアントでの ejb-ref を使用したルックアップ

```
InitialContext ic = new InitialContext();
Cart cart = (Cart)ic.lookup("java:comp/env/ejb/Test");
```

詳細は、19-20 ページの「初期コンテキスト・ファクトリの構成」を参照してください。

location を使用した EJB 3.0 Enterprise Bean のリモート・インタフェースのルックアップ

`location` を使用して EJB のリモート・インタフェースをルックアップするには、次のようにします。

1. `orion-ejb-jar.xml` ファイル内の `entity-deployment` 要素の `location` 属性を定義します。

例 29-6 location 属性の orion-ejb-jar.xml

```
<entity-deployment
    name="Test"
    location="app/Test"
    ...
>
...
</entity-deployment>
```

`location` 属性のデフォルト値は、`entity-deployment` の属性 `name` の値です。

2. 接頭辞が必要かどうかを判断します（29-4 ページの「EJB 参照の選択」を参照）。
3. location を使用して Enterprise Bean をルックアップします。

例 29-7 初期コンテキストを使用した EJB 3.0 EJB クライアントでの location を使用したルックアップ

```
InitialContext ic = new InitialContext();
Cart cart = (Cart)ic.lookup("java:comp/env/app/Test");
```

詳細は、19-20 ページの「初期コンテキスト・ファクトリの構成」を参照してください。

local-ref を使用した EJB 3.0 Enterprise Bean のローカル・インタフェースのルックアップ

ejb-local-ref を使用して EJB のリモート・インタフェースをルックアップするには、次のようにします。

1. ejb-jar.xml ファイルで Enterprise Bean の ejb-local-ref 要素を定義します。

例 29-8 ejb-local-ref 要素の ejb-jar.xml

```
<ejb-local-ref>
    <ejb-ref-name>ejb/Test</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local>Test</local>
</ejb-local-ref>
```

詳細は、19-10 ページの「ローカル EJB への環境参照の構成」を参照してください。

2. 接頭辞が必要かどうかを判断します（29-4 ページの「EJB 参照の選択」を参照）。
3. ejb-ref-name および適切な接頭辞（必要な場合）を使用して Enterprise Bean をルックアップします。

例 29-9 初期コンテキストを使用した EJB 3.0 EJB クライアントでの local-ref を使用したルックアップ

```
InitialContext ic = new InitialContext();
Cart cart = (Cart)ctx.lookup("java:comp/env/ejb/Test");
```

詳細は、19-20 ページの「初期コンテキスト・ファクトリの構成」を参照してください。

local-location を使用した EJB 3.0 Enterprise Bean のローカル・インタフェースのルックアップ

local-location を使用して EJB のローカル・インタフェースをルックアップするには、次のようにします。

1. orion-ejb-jar.xml ファイル内の entity-deployment 要素の local-location 属性を定義します。

例 29-10 local-location 属性の orion-ejb-jar.xml

```
<entity-deployment
    name="Test"
    local-location="app/Test"
    ...
>
...
</entity-deployment>
```

local-location のデフォルト値は、entity-deployment の属性 name の値です。

2. 接頭辞が必要かどうかを判断します（29-4 ページの「EJB 参照の選択」を参照）。

3. `local-location` を使用して Enterprise Bean をルックアップします。

例 29-11 初期コンテキストを使用した EJB 3.0 EJB クライアントでの local-location を使用したルックアップ

```
InitialContext ic = new InitialContext();
Cart cart = (Cart)ctx.lookup("java:comp/env/app/Test");
```

詳細は、19-20 ページの「初期コンテキスト・ファクトリの構成」を参照してください。

別のアプリケーションの EJB 3.0 Enterprise Bean へのアクセス

通常、Enterprise Bean は、複数の EAR ファイル間、つまり異なる EAR ファイルにデプロイされたアプリケーション間で通信を行うことはできません。ある Enterprise Bean が、異なる EAR ファイルにデプロイされている Enterprise Bean にアクセスする唯一の方法は、Enterprise Bean をクライアントの親として宣言することです。子のみが親の中でメソッドを起動できます。

たとえば、`Sales` および `Inventory` という 2 つの Enterprise Bean があり、それぞれ別の EAR ファイル内にデプロイされているとします。`Sales` Enterprise Bean は、`Inventory` Enterprise Bean を起動して十分なウィジェットが使用可能かどうかをチェックする必要があります。この 2 つの Enterprise Bean は異なる EAR ファイルにデプロイされているため、`Sales` Enterprise Bean で `Inventory` Enterprise Bean を親として定義しないかぎり、`Sales` Enterprise Bean は `Inventory` Enterprise Bean 内でメソッドを起動できません。したがって、`Inventory` Enterprise Bean を `Sales` Enterprise Bean の親として定義すると、`Sales` Enterprise Bean は親の中でメソッドを起動できるようになります。

親を定義できるのは、デプロイ・ウィザードを使用してデプロイを行うときのみです。Bean の親アプリケーションの定義方法については、『Oracle Containers for J2EE 構成および管理ガイド』の「admin.jar ユーティリティの使用方法」の章の「アプリケーションのデプロイ / アンデプロイ」を参照してください。

EntityManager を使用した JPA エンティティへのアクセス

EJB 3.0 アプリケーションでは、javax.persistence.EntityManager はエンティティの永続化およびデータベースからのエンティティのロードを行うためのランタイム・アクセス・ポイントです。

この項の内容は次のとおりです。

- EntityManager の取得
- 新規エンティティ・インスタンスの作成
- EntityManager を使用した JPA エンティティの問合せ
- エンティティ・インスタンスの変更
- エンティティ Bean インスタンスの連結解除およびマージ

詳細は、1-42 ページの「JPA エンティティの問合せ方法」を参照してください。

注意： JPA エンティティ・マネージャのコード例は、http://www.oracle.com/technology/tech/java/oc4j/ejb3/howtos-ejb3/howtoejb30entitymanager/doc/how-to-ejb30-entitymanager.html からダウンロードできます。

EntityManager の取得

EntityManager を使用する前に、EntityManager インスタンスを取得する必要があります。エンティティ・マネージャの取得方法は、クライアント・タイプによって決まります（29-2 ページの「使用しているクライアントのタイプ」を参照）。

エンティティ・マネージャを取得するときに、永続性ユニットを指定します。永続性ユニットでは、使用するファクトリ、エンティティ・マネージャが管理できる永続管理クラス、使用するオブジェクト・リレーショナル・マッピング・メタデータなどの詳細を含む、エンティティ・マネージャの構成を定義します。クライアントが永続性ユニットの有効範囲内にある場合は、特定の永続性ユニットのエンティティ・マネージャのみ取得できます。詳細は、2-10 ページの「persistence.xml ファイルとは」を参照してください。

次の方法でエンティティ・マネージャを取得できます。

- OC4J のデフォルト・エンティティ・マネージャの取得
- 名前付きエンティティ・マネージャの取得
- JNDI を使用したエンティティ・マネージャの取得
- Web クライアントでのエンティティ・マネージャの取得
- ヘルパー・クラスでのエンティティ・マネージャの取得

OC4J のデフォルト・エンティティ・マネージャの取得

@PersistenceContext アノテーションを使用して、EJB 3.0 セッション Bean クライアント（ステートフルまたはステートレス・セッション Bean、メッセージドリブン Bean、サーブレットなど）で EntityManager を注入できます。例 29-12 に示すように、unitName 属性を指定せずに @PersistenceContext を使用して、OC4J のデフォルト永続性ユニットを使用できます。

例 29-12 OC4J のデフォルト永続性ユニットでの @PersistenceContext の使用方法

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {

    @PersistenceContext protected EntityManager entityManager;

    public void createEmployee(String fName, String lName) {
        Employee employee  = new Employee();
        employee.setFirstName(fName);
        employee.setLastName(lName);
        entityManager.persist(employee);
    }
...
}
```

詳細は、2-10 ページの「OC4J の永続性ユニットのデフォルトについて」を参照してください。

名前付きエンティティ・マネージャの取得

@PersistenceContext アノテーションを使用して、EJB 3.0 セッション Bean クライアント（ステートフルまたはステートレス・セッション Bean、メッセージドリブン Bean、サーブレットなど）で EntityManager を注入できます。例 29-13 に示すように、@PersistenceContext の属性 unitName を使用して、永続性ユニットを名前で指定できます。この場合は、persistence.xml ファイルで永続性ユニットを構成する必要があります。

例 29-13 名前付き永続性ユニットでの @PersistenceContext の使用方法

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {

    @PersistenceContext(unitName="myPersistenceUnit") protected EntityManager entityManager;

    public void createEmployee(String fName, String lName) {
        Employee employee  = new Employee();
        employee.setFirstName(fName);
        employee.setLastName(lName);
        entityManager.persist(employee);
    }
...
}
```

詳細は、次を参照してください。

- 2-10 ページの「persistence.xml ファイルとは」
- 26-4 ページの「persistence.xml ファイルの構成」

JNDI を使用したエンティティ・マネージャの取得

別の方法として、例 29-14 に示すように、アノテーションを使用して永続性コンテキストを注入し、JNDI を使用してエンティティ・マネージャをルックアップできます。この場合は、persistence.xml ファイルで永続性ユニットを定義する必要があります。

例 29-14 初期コンテキストを使用した、ステートレス・セッション Bean での EntityManager のルックアップ

```
@PersistenceContext(
    name="persistence/InventoryAppMgr",
    unitName=InventoryManagement  // defined in a persistence.xml file
)
@Stateless
public class InventoryManagerBean implements InventoryManager {

    EJBContext ejbContext;
    public void updateInventory(...) {
        ...
        // obtain the initial JNDI context
        Context initCtx = new InitialContext();
        // perform JNDI lookup to obtain container-managed entity manager
        javax.persistence.EntityManager entityManager = (javax.persistence.EntityManager)
            initCtx.lookup("java:comp/env/persistence/InventoryAppMgr");
        ...
    }
}
```

詳細は、次を参照してください。

- 19-20 ページの「初期コンテキスト・ファクトリの構成」
- 2-10 ページの「persistence.xml ファイルとは」
- 26-4 ページの「persistence.xml ファイルの構成」

Web クライアントでのエンティティ・マネージャの取得

このリリースでは、例 29-15 に示すように、@PersistenceContext アノテーションを使用してサーブレットなどの Web クライアントに EntityManager を注入できます。この例では、デフォルトの EntityManager を注入していますが、例 29-13 のように名前付きエンティティ・マネージャを注入することもできます。詳細は、1-9 ページの「Web 層でのアノテーション」を参照してください。

例 29-15 @PersistenceContext を使用したサーブレットへの EntityManager の注入

```
    @Resource
    UserTransaction ut;
    @PersistenceContext
    EntityManager entityManager;
    ...
    try {
        ut.begin();

        Employee employee = new Employee();
        employee.setEmpNo(empId);
        employee.setEname(name);
        employee.setSal(sal);

        entityManager.persist(employee);
        ut.commit();

        this.getServletContext().getRequestDispatcher(
            "/jsp/success.jsp").forward(request, response);
    }
    catch(Exception e) {
    ...
```

```
}
```

ヘルパー・クラスでのエンティティ・マネージャの取得

アノテーションと注入をサポートしないクラス、つまりヘルパー・クラスでエンティティ・マネージャを取得するには、次の処理を行う必要があります。

1. `persistence.xml` ファイルで永続性ユニットを定義します。

   詳細は、次を参照してください。

   - 2-10 ページの「persistence.xml ファイルとは」
   - 26-4 ページの「persistence.xml ファイルの構成」

2. ヘルパー・クラスを利用する各 Java EE コンポーネントで、この永続性ユニットに対する参照をクラス・レベルで宣言します。永続性ユニットは、Java EE コンポーネントの環境（`java:comp/env`）に出現します。

   これを行うには、次のいずれかの方法を使用します。

   a. 次のように、ヘルパー・クラスを利用する Java EE コンポーネントで `@PersistenceContext` アノテーションを使用します。

   ```
   @PersistenceContext(name="helperPC", unitName="HelperPU")
   @Stateless
   public class EmployeeDemoSessionEJB implements EmployeeDemoSession {
       import com.acme.Helper;
       ...
       void doSomething() {
           Helper.createNewEmployee();
       }
   }
   ```

   `@PersistenceContext` アノテーションで、次のものを指定します。

   - `name`: 永続性コンテキストのルックアップに使用する名前
   - `unitName`: 返されたエンティティ・マネージャの特性を定義する、手順 1 で作成した永続性ユニットの名前

   b. ヘルパー・クラスを利用する Java EE コンポーネント用の適切なデプロイメント・ディスクリプタ・ファイルで `persistence-context-ref` を使用します（19-19 ページの「永続性コンテキストへの環境参照の構成」を参照）。

   `persistence-context-ref` で、次のものを指定します。

   - `persistence-context-ref`: 永続性コンテキストのルックアップに使用する名前
   - `persistence-unit-name`: 返されたエンティティ・マネージャの特性を定義する、手順 1 で作成した永続性ユニットの名前

3. ヘルパー・クラスで、定義した永続性ユニット名を使用して、JNDI を通じてエンティティ・マネージャをルックアップします。

   ```
   public class Helper {
       ...
       int createNewEmployee()
       {
           UserTransaction ut = null;
           ...
           try {
               Context initCtx = new InitialContext();

               ut = (UserTransaction)initCtx.lookup("java:comp/UserTransaction");
               ut.begin();
   ```

```
            Employee employee = new Employee();
            employee.setEmpNo(empId);

            // obtain the initial JNDI context
            Context initCtx = new InitialContext();
            javax.persistence.EntityManager entityManager =
                (javax.persistence.EntityManager)initCtx.lookup(
                    "java:comp/env/helperPC"
                );

            entityManager.persist(employee);

            ut.commit();
        }
        catch(Exception e) {
            ...
        }
    }
}
```

注意： ヘルパー・クラスで EntityManager を使用する場合、トランザクション内で EntityManager を使用する必要があるため、UserTransaction API を使用してトランザクションの境界を手動で設定する必要があります。

詳細は、19-20 ページの「初期コンテキスト・ファクトリの構成」を参照してください。

新規エンティティ・インスタンスの作成

EntityManager の取得後に（29-9 ページの「EntityManager の取得」を参照）、新規エンティティ・インスタンスを作成するには、例 29-16 に示すように、エンティティ Object を渡す EntityManager のメソッド persist を使用します。このメソッドをコールすると、メソッドは新規インスタンスにデータベースへの挿入をマークします。このメソッドは、渡したインスタンスと同じインスタンスを返します。

このメソッドは、トランザクション・コンテキスト内でコールする必要があります。

注意： 新規エンティティでは、EntityManager のメソッド persist のみ使用します。既存のエンティティに変更を加えた場合、それらの変更は現在のトランザクションのコミット時にデータベースに書き込まれます（29-18 ページの「フラッシュの使用方法」も参照）。

例 29-16 EntityManager でのエンティティの作成

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {

    @PersistenceContext protected EntityManager entityManager;
...
    public void createEmployee(String fName, String lName) {
        Employee employee  = new Employee();
        employee.setFirstName(fName);
        employee.setLastName(lName);
        entityManager.persist(employee);
    }
...
}
```

EntityManager を使用した JPA エンティティの問合せ

この項では、次のように EntityManager を使用して EJB 3.0 エンティティを問い合せる方法について説明します。

- エンティティ・マネージャを使用した主キーによるエンティティの検索
- EntityManager での名前付き問合せの作成
- EntityManager での動的 Java 永続性問合せ言語の問合せの作成
- EntityManager を使用した動的 TopLink 式問合せの作成
- EntityManager を使用した動的ネイティブ SQL 問合せの作成
- 問合せの実行

詳細は、次を参照してください。

- 1-42 ページの「JPA エンティティの問合せ方法」
- 第 8 章「JPA 問合せの実装」
- 8-4 ページの「JPA 問合せでの TopLink 問合せヒントの構成」

エンティティ・マネージャを使用した主キーによるエンティティの検索

例 29-17 に示すように、主キーがわかっている場合は、EntityManager のメソッド find を使用して、問合せを作成しなくても、対応するエンティティをデータベースから取得できます。

例 29-17 EntityManager を使用した主キーによるエンティティの検索

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {
...
    public void removeEmployee(Integer employeeId) {
        Employee employee = (Employee)entityManager.find("Employee", employeeId);
        ...
        entityManager.remove(employee);
    }
...
}
```

EntityManager での名前付き問合せの作成

例 29-18「EntityManager での名前付き問合せの作成」に示すように、名前付き問合せ（8-2 ページの「JPA 名前付き問合せの実装」を参照）を実装した後で、EntityManager のメソッド createNamedQuery を使用して実行時にその問合せを取得できます。名前付き問合せがパラメータを受け取る場合は、Query のメソッド setParameter を使用してこれらのパラメータを設定します。

例 29-18 EntityManager での名前付き問合せの作成

```
Query queryEmployeesByFirstName = entityManager.createNamedQuery(
    "findAllEmployeesByFirstName"
);
queryEmployeeByFirstName.setParameter("firstName", "John");
Collection employees = queryEmployessByFirstName.getResultList();
```

オプションで、問合せヒントで問合せを構成し、JPA 永続性プロバイダのベンダー拡張を使用できます（8-4 ページの「JPA 問合せでの TopLink 問合せヒントの構成」を参照）。

EntityManager での動的 Java 永続性問合せ言語の問合せの作成

例 29-19 に、EntityManager のメソッド createQuery を使用して実行時に非定型 EJB QL 問合せを作成する方法を示します。

例 29-19 EntityManager を使用した動的問合せの作成

```
Query queryEmployees = entityManager.createQuery(
    "SELECT OBJECT(employee) FROM Employee employee"
);
```

例 29-20 に、EntityManager のメソッド createQuery を使用して firstname という名前のパラメータを受け取る非定型問合せを作成する方法を示します。パラメータは、Query のメソッド setParameter を使用して設定します。

例 29-20 EntityManager を使用したパラメータ付き動的 Java 永続性問合せ言語の問合せの作成

```
Query queryEmployees = entityManager.createQuery(
    "SELECT OBJECT(emp) FROM Employee emp WHERE emp.firstName = :firstname"
);
queryEmployeeByFirstName.setParameter("firstName", "John");
```

オプションで、問合せヒントで問合せを構成し、JPA 永続性プロバイダのベンダー拡張を使用できます（8-4 ページの「JPA 問合せでの TopLink 問合せヒントの構成」を参照）。

EntityManager を使用した動的 TopLink 式問合せの作成

例 29-21 に示すように、oracle.toplink.ejb.cmp3.EntityManager のメソッド createQuery(Expression expression, Class resultType) を使用して、TopLink Expression に基づいて問合せを作成できます。

オプションで、問合せヒントで問合せを構成し、JPA 永続性プロバイダのベンダー拡張を使用できます（8-4 ページの「JPA 問合せでの TopLink 問合せヒントの構成」を参照）。

詳細は、『Oracle TopLink 開発者ガイド』の TopLink の式の理解に関する項を参照してください。

例 29-21 エンティティ・マネージャを使用した動的 TopLink 式問合せの作成

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {
...
    public Collection findManyProjectsByQuery(Vector params) {
        ExpressionBuilder builder = new ExpressionBuilder();
        Query query = ((oracle.toplink.ejb.cmp3.EntityManager)em).createQuery(
            builder.get("name").equals(builder.getParameter("projectName")),
            Project.class);
        query.setParameter("projectName", params.firstElement());
        Collection projects = query.getResultList();
        return projects;
    }
...
}
```

EntityManager を使用した動的ネイティブ SQL 問合せの作成

例 29-22 に示すように EntityManager のメソッド createNativeQuery(String sqlString) または createNativeQuery(String sqlString, Class resultType) を使用して、指定するネイティブ SQL 文字列に基づいて問合せを作成できます。

例 29-22 EntityManager を使用した動的ネイティブ SQL 問合せの作成

```
Query queryEmployees = entityManager.createNativeQuery(
    "Select * from EMP_TABLE where Salary < 50000", Employee.class
);
```

例 29-23 に、EntityManager のメソッド createNativeQuer(String sqlString, Class resultClass) を使用して salary という名前のパラメータを受け取る非定型ネイティブ SQL 問合せを作成する方法を示します。パラメータは、Query のメソッド setParameter を使用して設定します。

例 29-23 EntityManager を使用したパラメータ付き動的ネイティブ SQL 問合せの作成

```
Query queryEmployees = entityManager.createNativeQuery(
    "Select * from EMP_TABLE where Salary < #salary", Employee.class
);
queryEmployeeByFirstName.setParameter("salary", 50000);
```

オプションで、問合せヒントで問合せを構成し、JPA 永続性プロバイダのベンダー拡張を使用できます（8-4 ページの「JPA 問合せでの TopLink 問合せヒントの構成」を参照）。

問合せの実行

例 29-24 に示すように、複数の結果を返す問合せを実行するには、Query のメソッド getResultList を使用します。このメソッドは、java.util.List を返します。

例 29-24 複数の結果を返す問合せの実行

```
Collection employees = queryEmployees.getResultList();
```

例 29-25 に示すように、1 つの結果を返す問合せを実行するには、Query のメソッド getSingleResult を使用します。このメソッドは、java.lang.Object を返します。

例 29-25 1 つの結果を返す問合せの実行

```
Object obj = query.getSingleResult();
```

例 29-26 に示すように、エンティティを更新（変更または削除）する問合せを実行するには、Query のメソッド executeUpdate を使用します。このメソッドは、影響を受ける（更新または削除される）行数を int として返します。

例 29-26 更新問合せの実行

```
Query queryRenameCity = entityManager.createQuery(
    "UPDATE Address add SET add.city = 'Ottawa' WHERE add.city = 'Nepean'");
int rowCount = queryRenameCity.executeUpdate();
```

エンティティ・インスタンスの変更

エンティティ・インスタンスは、次のいずれかの方法で変更できます。

- 更新問合せの使用方法
- エンティティのパブリック API の使用方法
- データベースからのリフレッシュ
- エンティティの削除

これらの操作は、トランザクション・コンテキスト内で実行する必要があります。現在のトランザクションのコミット時に、更新はデータベースにコミットされます。

コミット前に、トランザクション内でデータベースに更新を送信することもできます（29-18 ページの「フラッシュの使用方法」を参照）。

更新問合せの使用方法

更新問合せ（29-14 ページの「EntityManager での名前付き問合せの作成」または 29-15 ページの「EntityManager での動的 Java 永続性問合せ言語の問合せの作成」を参照）を作成し、`EntityManager` を使用して問合せを実行します（29-16 ページの「問合せの実行」を参照）。

エンティティのパブリック API の使用方法

EntityManager を使用して、エンティティの検索または問合せを行います（29-14 ページの「EntityManager を使用した JPA エンティティの問合せ」を参照）。

エンティティのパブリック API を使用して、永続状態を変更します。

データベースからのリフレッシュ

例 29-27 に示すように、`EntityManager` のメソッド refresh を使用して、エンティティ・インスタンスの現在の状態を、データベースからの現在コミットされている状態で上書きできます。

例 29-27 データベースからのエンティティのリフレッシュ

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {
...
    public void undoUpdateEmployee(Integer employeeId) {
        Employee employee = (Employee)entityManager.find("Employee", employeeId);
        em.refresh(employee);
    }
...
}
```

エンティティの削除

例 29-28 に示すように、`EntityManager` のメソッド `remove` を使用して、データベースからエンティティを削除できます。

例 29-28 エンティティの削除

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {
...
    public void removeEmployee(Integer employeeId) {
        Employee employee = (Employee)entityManager.find("Employee", employeeId);
        ...
        entityManager.remove(employee);
    }
...
}
```

フラッシュの使用方法

例 29-29 に示すように、EntityManager のメソッド flush を使用して、トランザクションがコミットされる前にトランザクション内で更新をデータベースに送信できます。同じトランザクション内の後続の問合せでは、更新されたデータが返されます。この機能は、特定のトランザクションが複数の操作にまたがる場合に便利です。

例 29-29 トランザクション内でのデータベースへの更新の送信

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {
...
    public void terminateEmployee(Integer employeeId, Date endDate) {
        Employee employee = (Employee) entityManager.find("Employee", employeeId);
        employee.getPeriod().setEndDate(endDate);
        entityManager.flush();
    }
...
}
```

エンティティ Bean インスタンスの連結解除およびマージ

EntityManager は、永続性コンテキストを持つと言われます。EntityManager インスタンスを使用してエンティティを作成（29-13 ページの「新規エンティティ・インスタンスの作成」を参照）または検索（29-14 ページの「EntityManager を使用した JPA エンティティの問合せ」を参照）する場合、エンティティはその EntityManager の永続性コンテキストの一部であると言われます。

エンティティは EntityManager の永続性コンテキストの一部ですが、これは永続性エンティティであると言われます。

エンティティがこの永続性コンテキストの一部でなくなった場合は、連結解除されたエンティティであると言われます。

エンティティは、永続性コンテキストの終了時または（たとえば、別のアプリケーション層への）エンティティのシリアライズ時に、永続性コンテキストから連結解除されます。

例 29-30 に示すように、EntityManager のメソッド merge を使用して、連結解除されたエンティティの状態を EntityManager の現在の永続性コンテキストにマージできます。

例 29-30 EntityManager の永続性コンテキストへのエンティティのマージ

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {
...
    public void updateAddress(Address addressExample) {
        entityManager.merge(addressExample);
    }
...
}
```

永続性コンテキストの詳細は、次を参照してください。

- 2-10 ページの「persistence.xml ファイルとは」
- 26-4 ページの「persistence.xml ファイルの構成」

EJB 3.0 を使用した JMS 宛先へのメッセージの送信

クライアントは、MDB に直接にはアクセスしません。かわりに、クライアントは MDB に関連付けられている JMS 宛先（キューまたはトピック）を通じてメッセージを送信することにより MDB にアクセスします。

EJB 3.0 を使用して JMS 宛先にメッセージを送信するには、次のようにします。

1. JMS 宛先（キューまたはトピック）とそのコネクション・ファクトリの両方を注入します。

   これらのリソースは、事前定義の論理名または JMS プロバイダの構成時に定義した明示的な JNDI 名を使用して注入できます。この手順および例で示すように、論理名を使用することをお薦めします。

   詳細は、次を参照してください。

   - 19-14 ページの「JMS 宛先リソース・マネージャのコネクション・ファクトリへの環境参照の構成（JMS 1.1）」
   - 19-15 ページの「JMS 宛先またはコネクション・リソース・マネージャのコネクション・ファクトリへの環境参照の構成（JMS 1.0）」

2. コネクション・ファクトリを使用して接続を作成します。

   キューに対するメッセージを受信している場合は、接続を開始します。

3. 接続を使用してセッションを作成します。
4. 取得した JMS 宛先を使用して、キューのセンダーまたはトピックのパブリッシャを作成します。
5. メッセージを作成します。
6. キューのセンダーまたはトピックのパブリッシャのいずれかを使用して、メッセージを送信します。
7. キュー・セッションを閉じます。
8. 接続を閉じます。

例 29-31 に、サーブレット・クライアントがキューにメッセージを送信する方法を示します。

例 29-31 クライアントがメッセージをキューに送信するサーブレット

```
public final class testResourceProvider extends HttpServlet {

  private String resProvider = "myResProvider";
  private HashMap msgMap = new HashMap();

  // 1a. Rely on Servlet container to inject queue connection factory
  @Resource(name=resProvider+"QueueConnectionFactories/myQCF")
  private QueueConnectionFactory qcf;

  // 1b. Rely on Servlet container to inject queue
  @Resource(name=resProvider+"/Queues/rpTestQueue")
  private Queue queue;

  public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    doPost(req, res);
 }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
   // Retrieve the name of the JMS provider from the request,
   // which is to be used in creating the JNDI string for retrieval
    String rp = req.getParameter ("provider");
    if (rp != null) resProvider = rp;

    try {
```

```
      // 2a. Create queue connection using the connection factory
      QueueConnection qconn = qcf.createQueueConnection();
      // 2b. You are receiving messages, so start the connection
      qconn.start();

      // 3. Create a session over the queue connection
      QueueSession sess = qconn.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

       // 4. Since this is for a queue, create a sender on top of the session
       // This is used to send out the message over the queue
      QueueSender snd = sess.createSender (q);

    drainQueue (sess, q);
    TextMessage msg = null;

    // Send messages to queue
    for (int i = 0; i < 3; i++) {
      // 5. Create message
      msg = sess.createTextMessage();
      msg.setText ("TestMessage:" + i);

      // Set property of the recipient to be the MDB
      // and set the reply destination.
      msg.setStringProperty ("RECIPIENT", "MDB");
      msg.setJMSReplyTo(q);

      // 6. Send the message using the sender
      snd.send (msg);

      // You can store the messages IDs and sent-time  in a map (msgMap),
      // so that when messages are received, you can verify if you
      // *only* received those messages that you were
      // expecting. See receiveFromMDB() method where msgMap gets used
      msgMap.put( msg.getJMSMessageID(), new Long (msg.getJMSTimestamp()));
    }

    // receive a reply from the MDB
    receiveFromMDB (sess, q);

     // 7. Close sender, session, and connection for queue
     snd.close();
     sess.close();
     qconn.close();
    }
    catch (Exception e) {
      System.err.println ("** TEST FAILED **"+ e.toString());
      e.printStackTrace();
    }
    finally {
    }
}

 // Receive any messages sent to you through the MDB
 private void receiveFromMDB (QueueSession sess, Queue q)
    throws Exception {
    // The MDB sends out a message (as a reply) to this client. The MDB sets
    // the receipient as CLIENT. Thus, You will only receive messages that have
    // RECIPIENT set to 'CLIENT'
    QueueReceiver rcv = sess.createReceiver (q, "RECIPIENT = 'CLIENT'");

    int nrcvd = 0;
    long trtimes = 0L;
    long tctimes = 0L;
    // First message needs to come from MDB.
    // May take  a little while receiving messages
    for (Message msg = rcv.receive (30000); msg != null; msg = rcv.receive (30000)) {
```

```
        nrcvd++;
        String rcp = msg.getStringProperty ("RECIPIENT");

        // Verify if message is in message Map
        // Check the msgMap to see if this is the message that you are expecting
        String corrid = msg.getJMSCorrelationID();
        if (msgMap.containsKey(corrid)) {
          msgMap.remove(corrid);
        }
        else {
          System.err.println ("** received unexpected message [" + corrid + "] **");
        }
      }
      rcv.close();
    }

    // Drain messages from queue
    private int drainQueue (QueueSession sess, Queue q)
      throws Exception {
      QueueReceiver rcv = sess.createReceiver (q);
      int nrcvd = 0;

      // First drain any old messages from queue
      for (Message msg = rcv.receive(1000); msg != null; msg = rcv.receive(1000))
        nrcvd++;

      rcv.close();

      return nrcvd;
    }
  }
```

EJB 3.0 EJBContext へのアクセス

EJB 3.0 セッションおよびメッセージドリブン Bean の場合は、OC4J で提供される EJBContext にアクセスできます（29-22 ページの「リソース・インジェクションの使用方法」を参照）。

詳細は、次を参照してください。

- 1-7 ページの「EJB コンテキストとは」
- 1-37 ページの「セッション・コンテキストとは」
- 1-61 ページの「メッセージ・ドリブン・コンテキストとは」

リソース・インジェクションの使用方法

EJB 3.0 EJB クライアントでは、例 29-32 に示すように、@Resource 注入を使用して EJBContext にアクセスできます。

例 29-32 @Resource を使用した EJBContext へのアクセス

```
@Resource SessionContext ctx;
```

EJB 2.1 Enterprise Bean へのアクセス

この項の内容は次のとおりです。

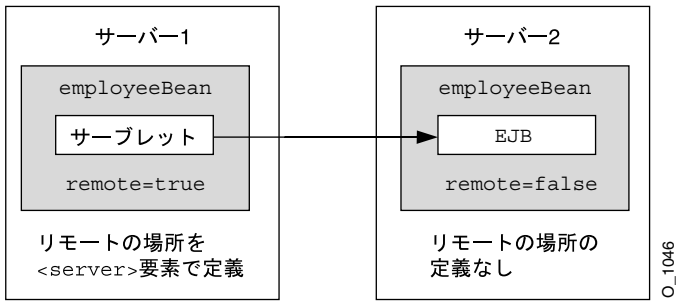
- EJB 2.1 Enterprise Bean へのリモート・アクセス
- EJB 2.1 Enterprise Bean へのローカル・アクセス
- RMI を使用した、スタンドアロン Java クライアントからの EJB 2.1 Enterprise Bean へのアクセス
- EJB 3.0 クライアントからの EJB 2.1 Enterprise Bean へのアクセス

EJB 2.1 Enterprise Bean へのリモート・アクセス

あるサーバーで実行されているサーブレットが別のサーバーの Enterprise Bean に接続して通信する場合は、リモート複数層の状況になります。サーブレットと Enterprise Bean は、両方とも同じアプリケーションに含まれています。アプリケーションを 2 つの異なるサーバーにデプロイすると、通常、サーブレットはローカル Enterprise Bean を最初に検索します。

図 29-1 では、HelloBean アプリケーションがサーバー 1 と 2 の両方にデプロイされています。サーバー 1 のサーブレットからサーバー 2 の Enterprise Bean へのコールのみを行うには、アプリケーションを両方のサーバーにデプロイする前に、アプリケーションで remote 属性を適切に設定する必要があります。

図 29-1 複数層の例



EJB モジュールの orion-application.xml における <ejb-module> 要素の remote 属性は、このアプリケーションの Enterprise Bean がデプロイされているかどうかを示します。

1. サーバー 1 では、orion-application.xml ファイルの <ejb-module> 要素で remote=true を設定してから、アプリケーションをデプロイする必要があります。アプリケーション内の EJB モジュールはデプロイされません。したがって、サーブレットはローカルで Enterprise Bean を検索しませんが、EJB リクエストに対してリモート・サーバーにアクセスします。
2. サーバー 2 では、orion-application.xml ファイルの <ejb-module> 要素で remote=false を設定してから、アプリケーションをデプロイする必要があります。EJB モジュールも含めて、アプリケーションは通常どおりデプロイされます。remote 属性のデフォルトは false です。したがって、remote 属性が true でないことを確認し、アプリケーションを再度デプロイします。
3. RMI オプションの構成
   - スタンドアロン OC4J では、RMI 構成ファイル rmi.xml で RMI サーバー・データを指定します。このファイルの場所は、OC4J 構成ファイル server.xml で指定します。デフォルトでは、この両方のファイルは *<ORACLE_HOME>*/j2ee/home/config にインストールされます。

     詳細は、『Oracle Containers for J2EE サービス・ガイド』の「スタンドアロン OC4J インストール環境での RMI の構成」を参照してください。
   - Oracle Application Server 環境では、opmn.xml ファイルを編集して、このローカル RMI サーバーが RMI リクエストをリスニングするポート範囲を指定する必要があります。Oracle Application Server 環境の構成ファイルへの手動変更は、各 OC4J インスタンスで手動で更新する必要があります。

     詳細は、『Oracle Containers for J2EE サービス・ガイド』の「Oracle Application Server 環境での RMI の構成」を参照してください。
4. JNDI のプロパティ java.naming.provider.url および java.naming.factory.initial を設定します。

   詳細は、次を参照してください。
   - 19-20 ページの「初期コンテキスト・ファクトリの構成」

- 『Oracle Containers for J2EE サービス・ガイド』の「RMI 用の JNDI プロパティの設定」

5. リモート Enterprise Bean をルックアップします。

   複数のリモート・サーバーが構成されている場合、OC4J はすべてのリモート・サーバーで目的の EJB アプリケーションを検索します。

詳細は、『Oracle Containers for J2EE サービス・ガイド』の「OC4J での Remote Method Invocation の使用方法」を参照してください。

EJB 2.1 Enterprise Bean へのローカル・アクセス

ローカル複数層の状況は、サーブレットと Enterprise Bean の両方が同じアプリケーションに含まれ、同じサーバーにデプロイされる場合に存在します。

EJB モジュールの `orion-application.xml` における `<ejb-module>` 要素の `remote` 属性は、このアプリケーションの Enterprise Bean がデプロイされているかどうかを示します。

1. アプリケーションをデプロイするサーバーでは、`orion-application.xml` ファイルの `<ejb-module>` 要素で `remote=false` を設定してから、アプリケーションをデプロイする必要があります。EJB モジュールも含めて、アプリケーションは通常どおりデプロイされます。`remote` 属性のデフォルトは `false` です。
2. JNDI のプロパティ `java.naming.provider.url` および `java.naming.factory.initial` を設定します。

   詳細は、次を参照してください。

   - 19-20 ページの「初期コンテキスト・ファクトリの構成」
   - 『Oracle Containers for J2EE サービス・ガイド』の「RMI 用の JNDI プロパティの設定」
3. ローカル EJB をルックアップします。

RMI を使用した、スタンドアロン Java クライアントからの EJB 2.1 Enterprise Bean へのアクセス

例 29-33 に、RMI ポートを指定しなくても、OC4J でデプロイされる Enterprise Bean をルックアップするためにこのリリースでスタンドアロン Java クライアント（29-2 ページの「スタンドアロン Java クライアント」を参照）から使用できるルックアップのタイプを示します。
例 29-33 では、ホスト `myServer` で実行されている `oc4j_inst1` という名前の OC4J インスタンスにデプロイされる Java EE アプリケーション `ejbsamples` で、`MyCart` という名前の Enterprise Bean をルックアップする方法を示します。

例 29-33 RMI を使用した、スタンドアロン Java クライアントからの EJB 2.1 Enterprise Bean へのアクセス

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,"oracle.j2ee.rmi.RMIInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "oc4jadmin");
env.put(Context.SECURITY_CREDENTIALS, "password");
env.put(Context.PROVIDER_URL,"opmn:ormi://myServer:oc4j_inst1/ejbsamples");

Context context = new InitialContext(env);

Object homeObject = context.lookup("MyCart");
CartHome home = (CartHome)PortableRemoteObject.narrow(homeObject,CartHome.class);
```

詳細は、次を参照してください。

- 19-21 ページの「Oracle 初期コンテキスト・ファクトリの構成」
- 19-22 ページの「OC4J および Oracle Application Server のネーミング・プロバイダの構成」
- 19-22 ページの「OC4J スタンドアロンのネーミング・プロバイダ URL の構成」

EJB 3.0 クライアントからの EJB 2.1 Enterprise Bean へのアクセス

EJB 3.0 クライアントから EJB 2.1 Enterprise Bean にアクセスするには、次のようにします。

1. 例 29-34 に示すように、EJB 2.1 Enterprise Bean のホームおよびリモート・インタフェースへの環境参照を作成します。

   この例では、EJB 2.1 `Scheduler` Bean のホームおよびリモート・インタフェースへの環境参照を構成します。ジョブ・スケジューラの詳細は、『Oracle Containers for J2EE ジョブ・スケジューラ開発者ガイド』を参照してください。

例 29-34 EJB 2.1 Enterprise Bean のホームおよびリモート・インタフェースへの環境参照の作成

```
<ejb-ref>
    <ejb-ref-name>ejb/scheduler</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>oracle.ias.scheduler.SchedulerHome</home>
    <remote>oracle.ias.scheduler.SchedulerRemote</remote>
</ejb-ref>
```

   詳細は、第 19 章「JNDI サービスの構成」を参照してください。

2. EJB 3.0 クライアントから EJB 2.1 Enterprise Bean にアクセスします。

   EJB 3.0 クライアントは、次のように様々な方法で EJB 2.1 Enterprise Bean にアクセスできます（これらの方法に限定されるわけではありません）。

   a. 例 29-35 に示すように、`@EJB` アノテーションを使用して EJB 2.1 ホーム・インタフェースを注入します。

      この例では、`@EJB` アノテーションの name 属性を EJB 2.1 Enterprise Bean の `<ejb-ref-name>` に設定します。

例 29-35 @EJB を使用した EJB 2.1 ホーム・インタフェースの注入

```
...
public class MyEJB30Client {

    @EJB(name="ejb/scheduler")
    SchedulerHome home;

    public void bar() {
        home.create();
        ...
    }
}
```

   b. デプロイ XML の `<injection-target>` 要素を使用して EJB 2.1 ホーム・インタフェースを注入します。

      例 29-36 に、`<injection-target>` 要素をデプロイ XML に追加して、EJB 2.1 ホーム・インタフェースを `home` というインスタンス変数に関連付ける方法を示します。例 29-37 に示すように、EJB 3.0 クライアントのインスタンス変数 `home` は、デプロイ時に OC4J によって適切に初期化されます。

例 29-36 デプロイ XML への <injection-target> の追加

```
<ejb-ref>
    <ejb-ref-name>ejb/scheduler</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>oracle.ias.scheduler.SchedulerHome</home>
    <remote>oracle.ias.scheduler.SchedulerRemote</remote>
    <injection-target>
        <injection-target-name>home</injection-target-name>
    </injection-target>
</ejb-ref>
```

例 29-37 インスタンス変数への EJB 2.1 ホーム・インタフェースの注入

```
...
public class MyEJB30Client {

    SchedulerHome home;

    public void bar() {
        home.create();
        ...
    }
}
```

c. 例 29-38 に示すように、JNDI を使用して EJB 2.1 ホーム・インタフェースをルックアップします。

この例では、EJB 2.1 Enterprise Bean の `java:comp/env/` という接頭辞付きの `<ejb-ref-name>` をルックアップします。

例 29-38 ホーム・インタフェースの JNDI ルックアップの実行

```
...
public class MyEJB30Client {

    SchedulerHome home;

    public void bar() {
        InitalContext ic = new InitialContext();
        home = ic.lookup("java:comp/env/ejb/scheduler");
        home.create();
        ...
    }
}
```

別のアプリケーションの EJB 2.1 Enterprise Bean へのアクセス

通常、Enterprise Bean は、複数の EAR ファイル間、つまり異なる EAR ファイルにデプロイされたアプリケーション間で通信を行うことはできません。ある Enterprise Bean が、異なる EAR ファイルにデプロイされている Enterprise Bean にアクセスする唯一の方法は、Enterprise Bean をクライアントの親として宣言することです。子のみが親の中でメソッドを起動できます。

たとえば、`Sales` および `Inventory` という 2 つの Enterprise Bean があり、それぞれ別の EAR ファイル内にデプロイされているとします。`Sales` Enterprise Bean は、`Inventory` Enterprise Bean を起動して十分なウィジェットが使用可能かどうかをチェックする必要があります。この 2 つの Enterprise Bean は異なる EAR ファイルにデプロイされているため、`Sales` Enterprise Bean で `Inventory` Enterprise Bean を親として定義しないかぎり、`Sales` Enterprise Bean は `Inventory` Enterprise Bean 内でメソッドを起動できません。したがって、`Inventory` Enterprise Bean を `Sales` Enterprise Bean の親として定義すると、`Sales` Enterprise Bean は親の中でメソッドを起動できるようになります。

親を定義できるのは、デプロイ・ウィザードを使用してデプロイを行うときのみです。Bean の親アプリケーションの定義方法については、『Oracle Containers for J2EE 構成および管理ガイド』の「admin.jar ユーティリティの使用方法」の章の「アプリケーションのデプロイ / アンデプロイ」を参照してください。

EJB 2.1 を使用した JMS 宛先へのメッセージの送信

クライアントは、MDB に直接にはアクセスしません。かわりに、クライアントは MDB に関連付けられている JMS 宛先（キューまたはトピック）を通じてメッセージを送信することにより MDB にアクセスします。

EJB 2.1 を使用して JMS 宛先にメッセージを送信するには、次のようにします。

1. JMS 宛先（キューまたはトピック）とそのコネクション・ファクトリの両方をルックアップします。

   これらのリソースは、事前定義の論理名または JMS プロバイダの構成時に定義した明示的な JNDI 名を使用してルックアップできます。この手順および例で示すように、論理名を使用することをお薦めします。

   詳細は、次を参照してください。

   - 19-14 ページの「JMS 宛先リソース・マネージャのコネクション・ファクトリへの環境参照の構成（JMS 1.1）」
   - 19-15 ページの「JMS 宛先またはコネクション・リソース・マネージャのコネクション・ファクトリへの環境参照の構成（JMS 1.0）」

2. コネクション・ファクトリを使用して接続を作成します。

   キューに対するメッセージを受信している場合は、接続を開始します。

3. 接続を使用してセッションを作成します。
4. 取得した JMS 宛先を使用して、キューのセンダーまたはトピックのパブリッシャを作成します。
5. メッセージを作成します。
6. キューのセンダーまたはトピックのパブリッシャのいずれかを使用して、メッセージを送信します。
7. キュー・セッションを閉じます。
8. 接続を閉じます。

例 29-39 に、サーブレット・クライアントがキューにメッセージを送信する方法を示します。

例 29-39 クライアントがメッセージをキューに送信するサーブレット

```
public final class testResourceProvider extends HttpServlet {

  private String resProvider = "myResProvider";
  private HashMap msgMap = new HashMap();
  Context ctx = new InitialContext();

  public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    doPost(req, res);
  }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
   // Retrieve the name of the JMS provider from the request,
   // which is to be used in creating the JNDI string for retrieval
    String rp = req.getParameter ("provider");
    if (rp != null) resProvider = rp;

    try {
      // 1a. Look up the Queue Connection Factory
      QueueConnectionFactory qcf = (QueueConnectionFactory)
          ctx.lookup("java:comp/resource/" + resProvider +
                     "/QueueConnectionFactories/myQCF");
      // 1b. Lookup the Queue
      Queue queue = (Queue)ctx.lookup("java:comp/resource/" +
```

```
                        resProvider + "/Queues/rpTestQueue");

    // 2a. Create queue connection using the connection factory
    QueueConnection qconn = qcf.createQueueConnection();
    // 2a. You are receiving messages, so start the connection
    qconn.start();

    // 3. Create a session over the queue connection
    QueueSession sess = qconn.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

     // 4. Since this is for a queue, create a sender on top of the session
     //This is used to send out the message over the queue
    QueueSender snd = sess.createSender (q);

  drainQueue (sess, q);
  TextMessage msg = null;

  // Send msgs to queue
  for (int i = 0; i < 3; i++) {
    // 5. Create message
    msg = sess.createTextMessage();
    msg.setText ("TestMessage:" + i);

    // Set property of the recipient to be the MDB
    // and set the reply destination
    msg.setStringProperty ("RECIPIENT", "MDB");
    msg.setJMSReplyTo(q);

    //6. Send the message using the sender
    snd.send (msg);

    // You can store the messages IDs and sent-time  in a map (msgMap),
    // so that when messages are received, you can verify if you
    // *only* received those messages that you were
    // expecting. See receiveFromMDB() method where msgMap gets used
    msgMap.put( msg.getJMSMessageID(), new Long (msg.getJMSTimestamp()));
  }

  // receive a reply from the MDB
  receiveFromMDB (sess, q);

   // 7. Close sender, session, and connection for queue
   snd.close();
   sess.close();
   qconn.close();
  }
  catch (Exception e) {
    System.err.println ("** TEST FAILED **"+ e.toString());
    e.printStackTrace();
  }
  finally {
  }
}

 // Receive any messages sent through the MDB
 private void receiveFromMDB (QueueSession sess, Queue q)
  throws Exception {
  // The MDB sends out a message (as a reply) to this client. The MDB sets
  // the receipient as CLIENT. Thus, you will only receive messages that have
  // RECIPIENT set to 'CLIENT'
  QueueReceiver rcv = sess.createReceiver (q, "RECIPIENT = 'CLIENT'");

  int nrcvd = 0;
  long trtimes = 0L;
  long tctimes = 0L;
  // First message needs to come from MDB. May take
```

```
    // a while receiving messages
    for (Message msg = rcv.receive (30000); msg != null; msg = rcv.receive (30000)) {
      nrcvd++;
      String rcp = msg.getStringProperty ("RECIPIENT");

      // Verify if messages in message Map
      // Check the msgMap to see if this is the message that you are expecting
      String corrid = msg.getJMSCorrelationID();
      if (msgMap.containsKey(corrid)) {
        msgMap.remove(corrid);
      }
      else {
        System.err.println ("** received unexpected message [" + corrid + "] **");
      }
    }
    rcv.close();
  }

  // Drain messages from queue
  private int drainQueue (QueueSession sess, Queue q)
    throws Exception {
    QueueReceiver rcv = sess.createReceiver (q);
    int nrcvd = 0;

    // First drain any old messages from queue
    for (Message msg = rcv.receive(1000); msg != null; msg = rcv.receive(1000))
      nrcvd++;

    rcv.close();

    return nrcvd;
  }
}
```

EJB 2.1 EJBContext へのアクセス

EJB 2.1 セッション、エンティティおよびメッセージドリブン Bean の場合は、Bean の実装時に適切な getter および setter メソッドを提供することにより、OC4J で提供される `EJBContext` にアクセスできます。

詳細は、次を参照してください。

- 1-7 ページの「EJB コンテキストとは」
- 11-11 ページの「setSessionContext メソッドの実装」
- 13-21 ページの「setEntityContext および unsetEntityContext メソッドの実装」
- 17-7 ページの「setMessageDrivenContext メソッドの実装」

パラメータの処理

この項の内容は次のとおりです。

- Enterprise Bean へのパラメータ情報の受渡し
- Enterprise Bean から返されるパラメータの処理

Enterprise Bean へのパラメータ情報の受渡し

Enterprise Bean を実装する場合、または EJB メソッドをコールするクライアント・コードを作成する場合、Enterprise Bean で使用されるパラメータの受渡し規則に注意する必要があります。

Bean メソッドに渡すパラメータ（または Bean メソッドからの戻り値）には、シリアライズ可能なすべての Java タイプを使用可能です。`int`、`double` など、Java のプリミティブ型は、シリアライズ可能です。`java.io.Serializable` インタフェースを実装する非リモート・オブジェクトは、すべて受渡し可能です。パラメータとして Bean に渡されるか Bean から返される非リモート・オブジェクトは、参照渡しではなく、値渡しされます。たとえば、次のように Bean メソッドをコールしたとします。

```
public class theNumber {
   int x;
}
...
bean.method1(theNumber);
```

この場合、Bean 内の `method1()` は、`theNumber` のコピーを受信します。Bean によってサーバーの `theNumber` オブジェクトの値が変更されても、値渡しのセマンティクスを使用しているため、クライアントにはこの変更は反映されません。

非リモート・オブジェクトが複合的である場合（複数のフィールドが含まれているクラスなど）、非静的で非一時的なフィールドのみコピーされます。

リモート・オブジェクトをパラメータとして渡す場合、リモート・オブジェクトのスタブが渡されます。パラメータとして渡されるリモート・オブジェクトは、リモート・インタフェースを拡張する必要があります。

次の項では、Bean へのパラメータの受渡しと、戻り値としてのリモート・オブジェクトについて説明します。

Enterprise Bean から返されるパラメータの処理

EmployeeBean のメソッド getEmployee は EmpRecord オブジェクトを返すため、このオブジェクトをアプリケーション内で定義しておく必要があります。この例では、EmpRecord クラスは、EJB インタフェースと同じパッケージに含まれています。

クラスは public として宣言されており、シリアライズされたリモート・オブジェクトとしてクライアントに値を返せるよう、java.io.Serializable インタフェースを実装する必要があります。次のように宣言します。

```
package employee;

public class EmpRecord implements java.io.Serializable {
  public String ename;
  public int empno;
  public double sal;
}
```

> **注意：** java.io.Serializable インタフェースではメソッドを指定しません。クラスがシリアライズ可能であることのみ示します。そのため、EmpRecord クラスで他のメソッドを実装する必要はありません。

例外の処理

この項の内容は次のとおりです。

- リモート Enterprise Bean へのアクセス中に発生する NamingException からのリカバリ
- リモート Enterprise Bean へのアクセス中に発生する NullPointerException からのリカバリ
- デッドロック状態からのリカバリ

リモート Enterprise Bean へのアクセス中に発生する NamingException からのリカバリ

Enterprise Bean にリモートでアクセスしようとし、javax.naming.NamingException エラーが発生する場合、JNDI プロパティが正しく初期化されていない可能性があります。リモート・オブジェクトまたはリモート・サーブレットから Enterprise Bean にアクセスするときの JNDI プロパティの設定に関する説明は、2-36 ページの「ロード・バランシング」を参照してください。

リモート Enterprise Bean へのアクセス中に発生する NullPointerException からのリカバリ

Web アプリケーションからリモート Enterprise Bean にアクセスするときに、「java.lang.NullPointerException: domain was null」というエラーが表示されます。この場合、dedicated.rmicontext が true に設定されている Enterprise Bean にアクセスするときは、環境プロパティをクライアントに設定する必要があります。

次の例は、この追加環境プロパティを使用する方法を示しています。

```
Hashtable env = new Hashtable( );
env.put (Context.INITIAL_CONTEXT_FACTORY,
  "oracle.j2ee.rmi.RMIInitialContextFactory");
env.put (Context.SECURITY_PRINCIPAL, "oc4jadmin");
env.put (Context.SECURITY_CREDENTIALS, "oc4jadmin");
env.put (Context.PROVIDER_URL, "ormi://myhost-us/ejbsamples");
env.put ("dedicated.rmicontext", "true"); // for 9.0.2.1 and later
Context context = new InitialContext (env);
```

dedicated.rmicontext の詳細は、2-36 ページの「ロード・バランシング」を参照してください。

デッドロック状態からのリカバリ

デッドロックの原因が複数の Bean のコール・シーケンスにある場合、OC4J はデッドロック状態を検出し、違反している Bean の 1 つにあるデッドロック状態の詳細を示す `Remote` 例外をスローします。

30

EJB および Web サービスの使用方法

この項の内容は次のとおりです。

- Web サービスとしてのステートレス・セッション Bean の公開
- Enterprise Bean からの Web サービスへのアクセス

詳細は、『Oracle Application Server Web Services 開発者ガイド』を参照してください。

Web サービスとしてのステートレス・セッション Bean の公開

ステートレス・セッション Bean のクライアントは、Web サービス・クライアントである場合があります。Web サービス・クライアント・ビューを提供できるのは、ステートレス・セッション Bean のみです。WSDL ドキュメントに記述されているように、Web サービス・クライアントは、Enterprise Bean の Web サービス・クライアント・ビューを利用します。Bean のクライアント・ビュー Web サービス・エンドポイント・インタフェースは、JAX-RPC インタフェースです。

EJB 3.0 を使用している場合は、アノテーションを使用して、ステートレス・セッション Bean を Web サービスとして簡単に公開できます（30-3 ページの「アノテーションの使用方法」を参照）。

EJB 2.1 を使用している場合も、ステートレス・セッション Bean を Web サービスとして公開できます（『Oracle Application Server Web Services 開発者ガイド』の「EJB を使用した Web サービスのアセンブル」の章を参照）。

アノテーションの使用方法

`@WebService` および `@WebMethod` アノテーションを使用して、例 30-1 に示すように Web サービス・エンドポイント・インタフェースを定義でき、例 30-2 に示すように Web サービスをステートレス・セッション Bean として実装できます。

例 30-1 アノテーション付き Web サービス・エンドポイント・インタフェース

```
package oracle.ejb30.ws;

import javax.ejb.Remote;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
/**
* This is an Enterprise Java Bean Service Endpoint Interface
*/
public interface HelloServiceInf extends java.rmi.Remote {

    /**
    * @param phrase java.lang.String
    * @return java.lang.String
    * @throws String The exception description.
    */
    @WebMethod
    java.lang.String sayHello(java.lang.String name) throws java.rmi.RemoteException;
}
```

例 30-2 ステートレス・セッション Bean としての Web サービスの実装

```
package oracle.ejb30.ws;

import java.rmi.RemoteException;
import java.util.Properties;
import javax.ejb.Stateless;

/**
* This is a session bean class
*/
@Stateless(name="HelloServiceEJB")
public class HelloServiceBean implements HelloServiceInf {

    public String sayHello(String name) {
        return("Hello "+name +" from first EJB3.0 Web Service");
    }
}
```

OC4J では、J2SE 5.0 Web サービス・アノテーション（Java Platform JSR-181 仕様では Web サービス・メタデータと呼ばれる）がサポートされます。仕様では、Web サービスをプログラミングするためのアノテーション付き Java 構文が定義されています。

Oracle 拡張を含む Web サービス・アノテーションの使用方法の詳細は、『Oracle Application Server Web Services 開発者ガイド』の「注釈を使用した Web サービスのアセンブル」の章を参照してください。

その他の EJB Web サービスの例は、`http://www.oracle.com/technology/tech/java/ejb30.html` にあるステートレス・セッション EJB Web サービスの使用方法または Adventure Builder の使用方法を参照してください。

Enterprise Bean からの Web サービスへのアクセス

Enterprise Bean から、Web サービスを取得してそのメソッドを起動できます。

EJB 3.0 を使用している場合は、Web サービスの環境参照を作成しなくてもアノテーションおよびリソース・インジェクション（30-3 ページの「アノテーションの使用方法」）を使用できます。

EJB 2.1 を使用している場合は、初期コンテキストを使用する必要があります（30-4 ページの「初期コンテキストの使用方法」を参照）。また、ルックアップする前に Web サービスの環境参照を作成する必要があります（19-18 ページの「Web サービスへの環境参照の構成」を参照）。

詳細は、『Oracle Application Server Web Services 開発者ガイド』の「J2EE Web サービス・クライアントのアセンブル」の章を参照してください。

アノテーションの使用方法

例 30-3 に示す Web サービスでは、例 30-4 に示すようにリソース・インジェクションを使用して EJB 3.0 ステートレス・セッション Bean から Web サービスにアクセスできます。

例 30-3 Web サービスのアノテーション付け

```
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class StockQuoteProvider {

    @WebMethod
    public Float getLastTradePrice() {
        ...
    }
}
```

例 30-4 リソース・インジェクションにより取得された Web サービスのコール

```
@Stateless
public class InvestmentBean implements Investment {

    public void checkPortfolio(...) {
        ...
        @Resource StockQuoteProvider sqp;

        // Get a quote
        Float quotePrice = sqp.getLastTradePrice(...);
        ...
    }
}
```

初期コンテキストの使用方法

Web サービスへの環境参照を定義した後で（19-18 ページの「Web サービスへの環境参照の構成」を参照）、例 30-5 に示すように初期コンテキストを使用して Web サービスをルックアップし、ステートレス・セッション Bean からそのメソッドを起動できます。

例 30-5 初期コンテキストから取得された Web サービスのコール

```
@Stateless
public class InvestmentBean implements Investment {

    public void checkPortfolio(...) {
        ...
        // Obtain the default initial JNDI context
        Context initCtx = new InitialContext();
        // Look up the stock quote service in the environment
        com.example.StockQuoteService sqs = (com.example.StockQuoteService)initCtx.lookup(
            "java:comp/env/service/StockQuoteService");
        // Get the stub for the service endpoint
        com.example.StockQuoteProvider sqp = sqs.getStockQuoteProviderPort();
        // Get a quote
        float quotePrice = sqp.getLastTradePrice(...);
        ...
    }
}
```

31

EJB アプリケーションの管理

この章の内容は次のとおりです。

詳細は、2-14 ページの「EJB 管理について」を参照してください。

OC4J EJB JMX サポート

OC4J は、MBeans をデプロイして、すべてのタイプの EJB の JSR77 統計および Oracle Dynamic Monitoring System（DMS）センサー・データを収集します。

これらの統計およびセンサーには、Application Server Control（31-2 ページの「Oracle Enterprise Manager 10g Application Server Control の使用方法」を参照）などの任意の JMX 準拠の管理ツールを使用してアクセスできます。

Oracle Enterprise Manager 10*g* Application Server Control の使用方法

Application Server Control は、OC4J 内でのアプリケーションのデプロイ、構成および監視、またアプリケーションで使用される OC4J サーバー・インスタンスと Web サービスの管理を行うための JMX 準拠の Web ベース・ユーザー・インタフェースです。

Application Server Control JMX 管理タスクを使用すると、Oracle Application Server の再起動やアプリケーションの再デプロイを行わなくても、次のように OC4J にデプロイされるすべての EJB タイプのプロパティを変更できます。

1. Application Server Control を起動します。
2. **「管理」**リンクをクリックします。
3. **「システム MBean ブラウザ」**をクリックします。
4. 特定の MBean インスタンスには、コンソールの左にあるナビゲーション・ペインからアクセスします。ナビゲーション・ペインのノードを開き、アクセスする MBean にドリルダウンします。

   たとえば、スタンドアロン OC4J の場合は、**「J2EEServer」**→**「スタンドアロン」**→**「J2EEApplication」**→「*application-name*」→**「EJBModule」**→「*module-name*」を選択します。
5. **「StatelessSessionBean」**、**「MessageDrivenBean」**、**「WebServicePort」**などの Enterprise Bean のタイプを選択します。
6. MBean インスタンスを選択します。
7. 右側のペインで適切なタブをクリックします。
   - **「属性」**タブをクリックして MBean の属性にアクセスします。属性値を変更する場合は、**「変更を適用」**をクリックして OC4J ランタイムに変更を適用します。
   - **「操作」**タブをクリックして MBean の操作にアクセスします。特定の操作を選択した後で、**「起動」**をクリックしてその操作をコールします。
   - **「通知」**タブをクリックして MBean の通知をサブスクライブします。特定の通知を選択した後で、**「適用」**をクリックしてその通知をサブスクライブします。
   - **「統計」**タブをクリックして MBean の統計を表示します。

ほとんどの管理タスクに Application Server Control を使用できます。

詳細は、次を参照してください。

- 『Oracle Containers for J2EE 構成および管理ガイド』の「Oracle Enterprise Manager 10*g* Application Server Control コンソール」
- Application Server Control に用意されているオンライン・ヘルプ

EJB ロギングの構成

OC4J は、標準 JDK である java.util.logging パッケージを使用し、デフォルトでログ・メッセージを <OC4J_HOME>/j2ee/home/log/<group>/oc4j/log.xml ファイルに書き込みます。

この項の内容は次のとおりです。

ロギング名前空間

次の java.util.logging 名前空間に対してロガーを構成できます。

- oracle.j2ee.ejb.annotation
- oracle.j2ee.ejb.compilation
- oracle.j2ee.ejb.database
- oracle.j2ee.ejb.deployment
- oracle.j2ee.ejb.lifecycle
- oracle.j2ee.ejb.pooling
- oracle.j2ee.ejb.runtime
- oracle.j2ee.ejb.transaction

ロギング・レベル

FINER、FINE、CONFIG、INFO、WARNING および SEVERE のログ・レベルを構成できます。

Application Server Control ロギング MBean でのロギングの構成

OC4J ロギングを構成する最も単純な方法は、Application Server Control を使用することです（31-2 ページの「Oracle Enterprise Manager 10g Application Server Control の使用方法」を参照）。

Application Server Control ではすべての EJB 関連ロガー名が表示され、Application Server Control インタフェースを使用してログ・レベルなどの属性を指定できます。

j2ee-logging.xml ファイルを使用したロギングの構成

例 31-1 に示すように、<OC4J_HOME>/j2ee/home/config/j2ee-logging.xml ファイルを使用して OC4J ロギングを構成できます。

例 31-1 j2ee-logging.xml ファイル

```
<logger
    name='oracle.j2ee.ejb'
    level='NOTIFICATION:1'
    useParentHandlers='false'>
    <handler name='oc4j-handler'/>
    <handler name='console-handler'/>
</logger>
```

詳細は、次を参照してください。

- 31-3 ページの「ロギング名前空間」
- 31-3 ページの「ロギング・レベル」

システム・プロパティを使用したロギングの構成

oracle.j2ee.logging システム・プロパティを使用して、OC4J ロギングを構成できます。このシステム・プロパティの形式は次のとおりです。

```
oracle.j2ee.logging.<log-level>=<log-namespace>
```

この場合 :

- <log-level> は、fine、finer、finest のいずれかです。
- <log-namespace> は、oracle.j2ee.ejb 名前空間です（31-3 ページの「ロギング名前空間」を参照）。

例 31-2 に、oracle.j2ee.ejb.deployment 名前空間のロガーを finest に構成する方法を示します。

例 31-2 システム・プロパティでのロガーの構成

```
oracle.j2ee.logging.finest=oracle.j2ee.ejb.deployment
```

TopLink ロギングの構成

EJB 3.0 JPA アプリケーションでは、ベンダー拡張を使用して TopLink JPA 永続性プロバイダによるロギング方法をカスタマイズできます。

詳細は、26-14 ページの「ロギング用の TopLink JPA 拡張」を参照してください。

Oracle JMS コネクタ・ロギングの構成

Oracle JMS コネクタを使用して JMS メッセージ・サービスにアクセスするアプリケーションでは、アクティブ化構成プロパティ LogLevel を使用して Oracle JMS コネクタによるロギング方法をカスタマイズできます。

詳細は、10-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」を参照してください。

Bean インスタンス・プールの管理

OC4J には、Bean インスタンスの作成頻度を削減してパフォーマンスを向上させるために構成できる EJB プーリング属性が用意されています。

この項の内容は次のとおりです。

- Bean インスタンスのプール・サイズの構成
- セッション Bean の Bean インスタンス・プール・タイムアウトの構成
- エンティティ Bean の Bean インスタンス・プール・タイムアウトの構成

Bean インスタンスのプール・サイズの構成

セッション Bean、エンティティおよびメッセージドリブン Bean の Bean インスタンス・プールの最小数および最大数を設定できます。

次の方法で Bean プール・サイズを構成できます。

- Oracle Enterprise Manager 10g Application Server Control の使用方法
- アノテーションの使用方法
- デプロイ XML の使用方法

デプロイ XML の構成は、アノテーションを使用して設定された対応する構成をオーバーライドします。

アノテーションの使用方法

EJB 3.0 セッション Bean およびメッセージドリブン Bean の Bean インスタンス・プール・サイズは、次の OC4J 固有のアノテーションとその属性を使用して指定できます。

- `@StatelessDeployment` の属性:
  - `maxInstances`
  - `minInstances`

  これらの属性の詳細は、表 A-1 を参照してください。
- `@StatefulDeployment` の属性:
  - `maxInstances`
  - `maxInstancesThreshold`

  これらの属性の詳細は、表 A-1 を参照してください。
- `@MessageDrivenDeployment` の属性:
  - maxInstances
  - minInstances

  これらの属性の詳細は、表 A-3 を参照してください。

例 31-3 に、`@StatelessDeployment` アノテーションを使用して EJB 3.0 ステートレス・セッション Bean でこれらの属性を構成する方法を示します。

例 31-3 @StatelessDeployment の maxInstances および minInstances 属性

```
import javax.ejb.Stateless;
import oracle.j2ee.ejb.StatelessDeployment;

@Stateless
@StatelessDeployment(
    maxInstances=10,
    minInstances=3
)
```

```
public class HelloWorldBean implements HelloWorld {

    public void sayHello(String name) {
        System.out.println("Hello "+name +" from first EJB3.0");
    }
}
```

デプロイ XML の使用方法

EJB 3.0 セッション Bean およびメッセージドリブン Bean の Bean インスタンス・プール・サイズは、次の orion-ejb-jar.xml ファイルの要素とその属性を使用して指定できます。

- ステートレス・セッション Bean の <session-deployment> の属性 :
  - max-instances
  - min-instances

  これらの属性の詳細は、表 A-1 を参照してください。
- ステートフル・セッション Bean の <session-deployment> の属性 :
  - max-instances
  - max-instances-threshold

  これらの属性の詳細は、表 A-1 を参照してください。
- メッセージドリブン Bean の <message-driven-deployment> の属性 :
  - max-instances
  - min-instances

  これらの属性の詳細は、表 A-3 を参照してください。

例 31-4 に、orion-ejb-jar.xml ファイルを使用して EJB 3.0 ステートレス・セッション Bean でこれらの属性を構成する方法を示します。

例 31-4 ステートレス・セッション Bean の Bean インスタンス・プール・サイズの orion-ejb-jar.xml

```
<?xml version="1.0" encoding="utf-8"?>
<orion-ejb-jar
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion-ejb-jar-10_
0.xsd"
    deployment-version="10.1.3.1.0"
    deployment-time="10b1fb5cdd0"
    schema-major-version="10"
    schema-minor-version="0"
>
    <enterprise-beans>
        <session-deployment
            max-instances="10"
            min-instances="3"
            ...
        >
        </session-deployment>
    ...
    </enterprise-beans>
    ...
</orion-ejb-jar>
```

この方法を使用してこのプロパティを変更する場合は、OC4J を再起動して変更を適用する必要があります。または、Application Server Control コンソールを使用して、OC4J を再起動せずにこのパラメータを動的に変更できます（31-2 ページの「Oracle Enterprise Manager 10g Application Server Control の使用方法」を参照）。

セッション Bean の Bean インスタンス・プール・タイムアウトの構成

セッション Bean が Bean インスタンス・プールにキャッシングされる最大時間を設定できます。

次の方法でセッション Bean のプール・タイムアウトを構成できます。

- Oracle Enterprise Manager 10g Application Server Control の使用方法
- アノテーションの使用方法
- デプロイ XML の使用方法

デプロイ XML の構成は、アノテーションを使用して設定された対応する構成をオーバーライドします。

アノテーションの使用方法

例 31-5 に、`@StatelessDeployment` アノテーションの `poolCacheTimeout` 属性を使用して EJB 3.0 ステートレス・セッション Bean の Bean インスタンス・プール・タイムアウトを構成する方法を示します。

この `@StatelessDeployment` の属性の詳細は、表 A-1 を参照してください。
`@StatelessDeployment` アノテーションの詳細は、5-12 ページの「EJB 3.0 セッション Bean の OC4J 固有のデプロイ・オプションの構成」を参照してください。

例 31-5 @StatelessDeployment の poolCacheTimeout 属性

```
import javax.ejb.Stateless;
import oracle.j2ee.ejb.StatelessDeployment;

@Stateless
@StatelessDeployment(
    poolCacheTimeout=90
)
public class HelloWorldBean implements HelloWorld {

    public void sayHello(String name) {
        System.out.println("Hello "+name +" from first EJB3.0");
    }
}
```

例 31-6 に、`@StatefulDeployment` アノテーションの `timeout` 属性を使用して EJB 3.0 ステートフル・セッション Bean の Bean インスタンス・プール・タイムアウトを構成する方法を示します。

この `@StatelessDeployment` の属性の詳細は、表 A-1 を参照してください。
`@StatelessDeployment` アノテーションの詳細は、5-12 ページの「EJB 3.0 セッション Bean の OC4J 固有のデプロイ・オプションの構成」を参照してください。

例 31-6 @StatefulDeployment の timeout 属性

```
import javax.ejb.Stateful
import oracle.j2ee.ejb.StatefulDeployment;

@Stateful
@StatefulDeployment(
    timeout=100
)
public class CartBean implements Cart {

    private ArrayList items;
    ...
}
```

デプロイ XML の使用方法

orion-ejb-jar.xml ファイルで、セッション Bean の <session-deployment> 要素の次の属性を使用して Bean プール・タイムアウトを設定します。

- pool-cache-timeout 属性はステートレス・セッション Bean に適用でき、キャッシュされたステートレス・セッションをプール内に保持する時間を設定します。デフォルトは 0 秒で、タイムアウトしないことを意味します。

  たとえば、pool-cache-timeout を 90 秒に設定する場合は、次のようにします。

  ```
  <session-deployment ...  pool-cache-timeout="90"
    ...
  </session-deployment>
  ```

- timeout 属性は、ステートフル・セッション Bean に適用されます。ステートフル・セッション Bean が非アクティブなままこの時間が経過すると、その Bean は Bean インスタンス・プールから削除されます。デフォルトは 1800 秒です。

  たとえば、ステートフル・セッション Bean の非アクティブ・タイムアウトを 900 秒に設定する場合は、次のようにします。

  ```
  <session-deployment ...  timeout="900"
    ...
  </session-deployment>
  ```

この方法を使用してこのプロパティを変更する場合は、OC4J を再起動して変更を適用する必要があります。または、Application Server Control コンソールを使用して、OC4J を再起動せずにこのパラメータを動的に変更できます（31-2 ページの「Oracle Enterprise Manager 10g Application Server Control の使用方法」を参照）。

エンティティ Bean の Bean インスタンス・プール・タイムアウトの構成

エンティティが Bean インスタンス・プールにキャッシングされる最大時間を設定できます。

次の方法でエンティティのプール・タイムアウトを構成できます。

- Oracle Enterprise Manager 10g Application Server Control の使用方法
- デプロイ XML の使用方法

デプロイ XML の使用方法

orion-ejb-jar.xml ファイルで、エンティティの <entity-deployment> 要素の次の属性を使用して Bean プール・タイムアウトを設定します。

- pool-cache-timeout 属性では、エンティティ Bean 実装インスタンスが「プーリング」された（未割当て）状態に保持される時間の長さを設定します。デフォルトは 60 秒です。この属性を never に設定すると、タイムアウトは発生しません。

  たとえば、エンティティの pool-cache-timeout を 90 秒に設定する場合は、次のようにします。

  ```
  <entity-deployment ...  pool-cache-timeout="90"
    ...
  </entity-deployment>
  ```

この方法を使用してこのプロパティを変更する場合は、OC4J を再起動して変更を適用する必要があります。または、Application Server Control コンソールを使用して、OC4J を再起動せずにこのパラメータを動的に変更できます（31-2 ページの「Oracle Enterprise Manager 10g Application Server Control の使用方法」を参照）。

EJB アプリケーションの起動および停止

Application Server Control を使用して、EJB アプリケーションを起動および停止できます。

アプリケーションの停止中、クライアントはそのアプリケーションにアクセスできません。

詳細は、31-2 ページの「Oracle Enterprise Manager 10g Application Server Control の使用方法」を参照してください。

EJB アプリケーションのトラブルシューティング

この項の内容は次のとおりです。

XML ファイルの検証

XML ファイルを検証するように OC4J を構成するには、OC4J 起動スクリプト（`<OC4J_HOME>/BIN/oc4j.cmd` または `oc4j`）で使用されるコマンドラインに `-validateXML` オプションを追加します。

例 31-7 に、`oc4j.cmd` ファイルでのこのオプションの設定方法を示します。

例 31-7 oc4j.cmd での -validateXML の設定

```
...
"%JAVA_HOME%¥bin¥java" %JVMARGS% -jar %OC4J_JAR% %CMDARGS% -validateXML
...
```

このオプションを設定すると、OC4J が XML ファイルを読み取る際に、OC4J は XML ファイルを指定のスキーマに照らして厳密に検証します。OC4J はエラーをログに記録します（31-3 ページの「EJB ロギングの構成」を参照）。

ejb-jar.xml ファイルのデバッグ

1 つ以上のアノテーションのある EJB 3.0 アプリケーションのデプロイ時に、OC4J はそのメモリー内の `ejb-jar.xml` ファイルをデプロイ・ディレクトリ内の `orion-ejb-jar.xml` ファイルと同じ場所（`<`*`ORACLE_HOME`*`>/j2ee/home/application-deployments/my_application/META-INF`）に自動的に書き込みます。

この `ejb-jar.xml` ファイルは、アノテーションとデプロイ済 `ejb-jar.xml` ファイル（存在する場合）の両方から取得された構成を表します。

EJB 2.1 アプリケーションをデプロイする場合、生成されたラッパー・コードを保持するには、システム・プロパティ `KeepWrapperCode` を設定する必要があります（31-10 ページの「生成されたラッパー・コードのデバッグ」を参照）。

31-9 ページの「XML ファイルの検証」も参照してください。

生成されたラッパー・コードのデバッグ

デフォルトでは、OC4J は、EJB 2.1 CMP アプリケーションをデプロイする際に、<OC4J_HOME>/j2ee/home/application-deployments/<ear-name>/<ejb-name>/generated にラッパー・コードを生成し、そのコードをコンパイルし、コンパイルされたクラスを含む JAR ファイルを作成してから、生成したラッパー・コードを削除します。

生成したラッパー・コードを保持するように OC4J を構成できます。ラッパー・コードの検証は、アプリケーションの問題のデバッグに役立ちます。

この項の内容は次のとおりです。

注意： 生成されたラッパー・コードのデバッグは、このリリースでは推奨されていません。

これらのオプションは、コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean にのみ適用されます。セッション Bean、メッセージドリブン Bean または EJB 3.0 エンティティには適用されません。OC4J は、コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean ごとに 1 つのファイルを生成します。EJB 3.0 エンティティのみを使用している場合、OC4J は何も生成しません。

生成されたラッパー・コードのデフォルト・ディレクトリへの保持

生成コードを保持するよう OC4J を構成するには、OC4J の起動コマンドラインでシステム・プロパティ KeepWrapperCode を true に設定します。例 31-8 に、<OC4J_HOME>/bin/oc4j.cmd ファイルの場合を示します。

例 31-8 oc4j.cmd での KeepWrapperCode の設定

```
...
"%JAVA_HOME%¥bin¥java" %JVMARGS% -DKeepWrapperCode=true -jar "%OC4J_JAR%" %CMDARGS%
...
```

KeepWrapperCode が true の場合、OC4J は、生成したラッパー・コードをデフォルト・ディレクトリ <OC4J_HOME>/j2ee/home/application-deployments/<ear-name>/<ejb-name>/generated に保持します。または、OC4J がラッパー・コードの保持に使用するディレクトリを指定することもできます（31-10 ページの「生成されたラッパー・コードの指定ディレクトリへの保持」を参照）。

アプリケーションをアンデプロイすると、このディレクトリ内のラッパー・コードが OC4J によって削除されます。

生成されたラッパー・コードの指定ディレクトリへの保持

システム・プロパティ KeepWrapperCode を true に設定し、システム・プロパティ WrapperCodeDir をディレクトリ（<specified-wrapper-dir> と呼ぶ）に設定した場合、OC4J では、このディレクトリにラッパー・コードが生成され、アプリケーションをアンデプロイしてもラッパー・コードが保持されます。例 31-9 に、<OC4J_HOME>/bin/oc4j.cmd ファイルの場合を示します。

例 31-9 oc4j.cmd での KeepWrapperCode および WrapperCodeDir の設定

```
...
"%JAVA_HOME%¥bin¥java" %JVMARGS% -DKeepWrapperCode=true -DWrapperCodeDir=C:¥wrappers -jar
"%OC4J_JAR%" %CMDARGS%
...
```

<specified-wrapper-dir> は、絶対パス（C:¥wrappers など）または相対パス（./wrappers など）に設定できます。相対パスは、<OC4J_HOME>/j2ee/home からの相対です。

（権限の問題や領域の不足などにより）指定したディレクトリに OC4J が生成できない場合は、OC4J によってデフォルト・ディレクトリ <OC4J_HOME>/j2ee/home/application-deployments/<ear-name>/<ejb-name>/generated にラッパー・コードが生成され、アプリケーションをアンデプロイしてもこのラッパー・コードが保持されます。

生成されたラッパー・コードの変更

システム・プロパティ KeepWrapperCode を true に設定し、システム・プロパティ DoNotReGenerateWrapperCode を true に設定した場合、OC4J では、ラッパー・コードが生成され、アプリケーションをアンデプロイしてもラッパー・コードが保持されます。例 31-10 に、<OC4J_HOME>/bin/oc4j.cmd ファイルの場合を示します。この場合、再デプロイ時には、OC4J はラッパー・コードを再生成せずに、かわりにデフォルト・ディレクトリ（31-10 ページの「生成されたラッパー・コードのデフォルト・ディレクトリへの保持」を参照）または指定したディレクトリ（31-10 ページの「生成されたラッパー・コードの指定ディレクトリへの保持」を参照）に保持されたバージョンのラッパー・コードを使用します。

例 31-10 oc4j.cmd での KeepWrapperCode および DoNotReGenerateWrapperCode の設定

```
...
"%JAVA_HOME%¥bin¥java" %JVMARGS% -DKeepWrapperCode=true -DDoNotReGenerateWrapperCode=true
-jar "%OC4J_JAR%" %CMDARGS%
...
```

これらのシステム・プロパティを使用すると、デバッグ文を追加するなど、ラッパー・コードを変更できます。再デプロイ時に、OC4J は、ユーザーに変更された保持バージョンのラッパー・コードを再コンパイルして使用します。

生成されたラッパー・コードの保持の無効化

生成されたラッパー・コードの保持を無効化するには、システム・プロパティ KeepWrapperCode を false に設定し、システム・プロパティ DoNotReGenerateWrapperCode を false に設定します。または、これらのシステム・プロパティを設定しないままとします。

32

EJB パフォーマンスの最適化

この章では、次のような EJB のパフォーマンスを向上するために使用できるいくつかの重要なオプションについて簡単に説明します。

- セッション Bean のパフォーマンス
- JPA エンティティのパフォーマンス
- コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のパフォーマンス
- Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean のパフォーマンス
- メッセージドリブン Bean のパフォーマンス

パフォーマンス情報の詳細は、『Oracle Application Server パフォーマンス・ガイド』を参照してください。

セッション Bean のパフォーマンス

セッション Bean のパフォーマンスを向上するには、次の点を考慮します。

- Bean インスタンスのプーリング
- シングルトン・インターセプタ

Bean インスタンスのプーリング

セッション Bean では、Bean インスタンスのプーリングを使用して Bean 作成によるオーバーヘッドを削減することで、パフォーマンスが向上する可能性があります。詳細は、31-5 ページの「Bean インスタンス・プールの管理」を参照してください。

シングルトン・インターセプタ

EJB 3.0 セッション Bean では、使用するインターセプタがステートレスの場合、シングルトン・インターセプタを指定できます。OC4J EJB 3.0 拡張により、インターセプタ・クラスごとにシングルトンが作成されます。このシングルトンは、そのインターセプタ・クラスを使用するすべてのセッション Bean インスタンスで共有されます。これにより、メモリー要件とライフ・サイクル管理が削減されます。詳細は、2-14 ページの「シングルトン・インターセプタ」を参照してください。

JPA エンティティのパフォーマンス

JPA エンティティのパフォーマンスを向上するには、次の点を考慮します。

- Bean インスタンスのプーリング
- フェッチ・タイプ

Bean インスタンスのプーリング

EJB 3.0 エンティティでは、Bean インスタンスのプーリングを使用して Bean 作成によるオーバーヘッドを削減することで、パフォーマンスが向上する可能性があります。詳細は、31-5 ページの「Bean インスタンス・プールの管理」を参照してください。

フェッチ・タイプ

すべての EJB 3.0 マッピング・タイプで、遅延ロードまたは即時ロードのいずれかを使用してデータベースからデータをフェッチする方針を定義できます。これにより、エンティティの特定部分に対するアクセスが少ないとわかっている場合は、パフォーマンスを向上できる可能性があります。この方法は、実行される SQL の量の削減、問合せ実行時間の短縮、およびオブジェクトのロード時間の短縮が可能な関連マッピングに対して特に有効です。詳細は、7-17 ページの「遅延ロードの構成」を参照してください。

コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のパフォーマンス

コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean のパフォーマンスを向上するには、次の点を考慮します。

- Bean インスタンスのプーリング
- コンテナ管理の永続性を備えた読取り専用エンティティ Bean

詳細は、『Oracle Application Server パフォーマンス・ガイド』の EJB CMP 2.1 パフォーマンスの向上に関する項を参照してください。

Bean インスタンスのプーリング

コンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean では、Bean インスタンスのプーリングを使用して Bean 作成によるオーバーヘッドを削減することで、パフォーマンスが向上する可能性があります。詳細は、31-5 ページの「Bean インスタンス・プールの管理」を参照してください。

コンテナ管理の永続性を備えた読取り専用エンティティ Bean

アクティブ化後に変更されないコンテナ管理の永続性を備えた EJB 2.1 エンティティ Bean では、ロック・モードを読取り専用に指定できます。詳細は、1-64 ページの「同時実行性（ロック）モード」を参照してください。

Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean のパフォーマンス

Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean のパフォーマンスを向上するには、次の点を考慮します。

- Bean 管理の永続性を備えた読取り専用エンティティ Bean
- コミット・オプション A

Bean 管理の永続性を備えた読取り専用エンティティ Bean

アクティブ化後に変更されない Bean 管理の永続性を備えた EJB 2.1 エンティティ Bean では、エンティティ Bean を読取り専用に指定できます。Bean 管理の永続性を備えたエンティティ Bean を読取り専用として構成した場合、OC4J はコミット・オプション A の特殊ケースを使用して次の操作を実行し、パフォーマンスを向上させます。

- インスタンスのキャッシング。
- アクティブ化の後に `ejbLoad` をコールしない。
- トランザクションのコミット時にインスタンスの更新または `ejbStore` のコールを行わない。

詳細は、15-4 ページの「Bean 管理の永続性を備えた読取り専用エンティティ Bean の構成」を参照してください。

コミット・オプション A

EJB 2.1 BMP アプリケーションでは、BMP コミット・オプションを A または C に構成できます。コミット・オプション A では、`ejbLoad` メソッドへのコールを延期することでパフォーマンスが向上します。詳細は、1-53 ページの「コミット・オプションおよび BMP アプリケーション」を参照してください。

Bean 管理の永続性を備えた読取り専用エンティティ Bean でコミット・オプション A を使用するよう構成する場合、Bean 管理の永続性を備えた読取り専用エンティティ Bean のキャッシングを利用することでパフォーマンスをさらに向上させることができます。詳細は、32-3 ページの「Bean 管理の永続性を備えた読取り専用エンティティ Bean」を参照してください。

メッセージドリブン Bean のパフォーマンス

メッセージドリブン Bean のパフォーマンスを向上するには、次の点を考慮します。

- Bean インスタンスのプーリング
- シングルトン・インターセプタ

詳細は、『Oracle Application Server パフォーマンス・ガイド』の MDB パフォーマンスの向上に関する項を参照してください。

Bean インスタンスのプーリング

メッセージドリブン Bean では、Bean インスタンスのプーリングを使用して Bean 作成によるオーバーヘッドを削減することで、パフォーマンスが向上する可能性があります。詳細は、31-5 ページの「Bean インスタンス・プールの管理」を参照してください。

シングルトン・インターセプタ

EJB 3.0 メッセージドリブン Bean では、使用するインターセプタがステートレスの場合、シングルトン・インターセプタを指定できます。OC4J EJB 3.0 拡張により、インターセプタ・クラスごとにシングルトンが作成されます。このシングルトンは、そのインターセプタ・クラスを使用するすべてのメッセージドリブン Bean インスタンスで共有されます。これにより、メモリー要件とライフ・サイクル管理が削減されます。詳細は、2-14 ページの「シングルトン・インターセプタ」を参照してください。

A

orion-ejb-jar.xml 要素の XML 参照

この付録では、OC4J 固有の EJB デプロイメント・ディスクリプタである `orion-ejb-jar.xml` に含まれている次のような要素について説明します。

詳細は、次を参照してください。

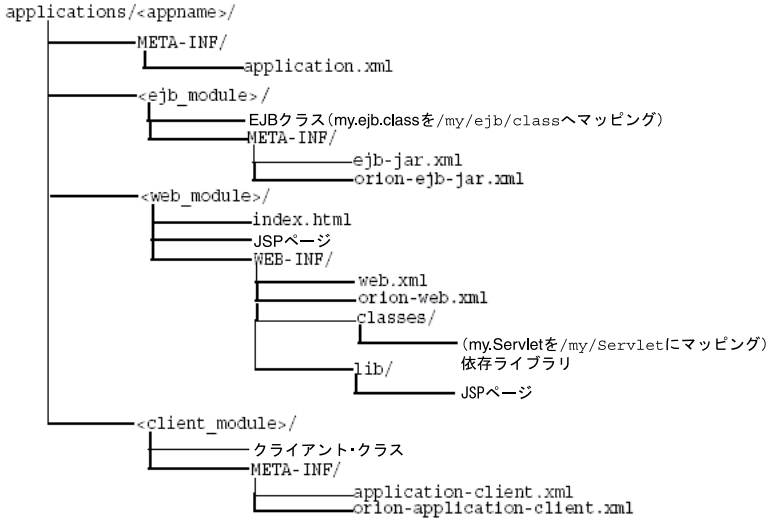
- 2-6 ページの「EJB デプロイメント・ディスクリプタ・ファイルについて」
- `http://www.oracle.com/technology/oracleas/schema/index.html`

OC4J および orion-ejb-jar.xml ファイル

アプリケーションをデプロイするたびに、OC4J により、デフォルトの要素を持つ OC4J 固有の XML ファイルが自動的に生成されます。これらのデフォルトを変更する場合、orion-ejb-jar.xml ファイルを、元の ejb-jar.xml ファイルが存在している場所にコピーし、ここで変更する必要があります。XML ファイルをデプロイした場所で変更すると、アプリケーションが再びデプロイされた場合に、OC4J によってこれらの変更が上書きされます。開発ディレクトリで変更が行われた場合のみ、変更が維持されます。

オラクル社では、OC4J 固有の XML ファイルを、Figure A-1 に示す推奨する開発用ディレクトリ構造に追加することをお薦めします。

図 A-1 開発アプリケーションのディレクトリ構造



TopLink 永続性サポート

表 A-2 に、orion-ejb-jar.xml ファイルの <entity-deployment> 要素のすべての属性を説明し、orion-ejb-jar.xml ファイルで構成するオプションおよび TopLink 永続性 API を使用して構成するオプションを示します。

次に例を示します。

- <entity-deployment> の属性 call-timeout を構成するには、対応する TopLink 永続性 API を使用する必要があります。orion-ejb-jar.xml ファイル内で call-timeout 属性を設定した場合、OC4J ではこの属性が無視されます。
- <entity-deployment> の属性 clustering-schema を構成するには、orion-ejb-jar.xml ファイルを使用する必要があります。対応する TopLink 永続性 API はありません。

EJB 3.0 アプリケーションの場合は、TopLink 固有のデプロイメント・ディスクリプタ・ファイル ejb3-toplink-sessions.xml および toplink-ejb-jar.xml で orion-ejb-jar.xml 構成を補強することにより、TopLink 永続性 API にアクセスします。詳細は、3-4 ページの「JPA 永続性プロバイダのカスタマイズ」を参照してください。

EJB 2.1 アプリケーションでは、orion-ejb-jar.xml の要素 pm-properties を使用して TopLink 永続性 API にアクセスします。詳細は、3-15 ページの「TopLink EJB 2.1 永続性マネージャのカスタマイズ」を参照してください。

注意： TopLink デプロイメント・ディスクリプタ・ファイルを変更するには、TopLink Workbench を使用します。

詳細は、次を参照してください。

- 『Oracle TopLink 開発者ガイド』の OC4J Orion 永続性から OC4J TopLink 永続性への移行に関する項
- 『Oracle TopLink 開発者ガイド』の TopLink Workbench の理解に関する項
- *<OC4J_HOME>*¥toplink¥config¥xsds にある TopLink 固有のデプロイメント・ディスクリプタ XML Schema 文書

<orion-ejb-jar>

OC4J 固有のデプロイメント・ディスクリプタには、セッション Bean、エンティティ Bean、メッセージドリブン Bean、およびこれらの Enterprise Bean のセキュリティに関する広範なデプロイ情報が含まれています。このデプロイメント・ディスクリプタ内の主な要素構造は、次のようになっています。

```
<orion-ejb-jar deployment-time=... deployment-version=...>
 <enterprise-beans>
   <persistence-manager ...></persistence-manager>
   <session-deployment ...></session-deployment>
   <entity-deployment ...></entity-deployment>
   <message-driven-deployment ...></message-driven-deployment>
 </enterprise-beans>
 <assembly-descriptor>
   <security-role-mapping ...></security-role-mapping>
   <default-method-access></default-method-access>
 </assembly-descriptor>
</orion-ejb-jar>
```

<orion-ejb-jar> メイン・タグの下の各セクションは、それぞれ用途が決まっています。これらのセクションについては、次の各項で説明します。

- <enterprise-beans>
- <assembly-descriptor>

<enterprise-beans>

<enterprise-beans> セクションでは、すべての Enterprise Bean（セッション Bean、エンティティ Bean およびメッセージドリブン Bean）の追加のデプロイ情報を定義します。各 EJB のタイプごとにセクションが分かれています。

次の各項で、<enterprise-beans> 要素内の要素を説明します。

- <persistence-manager>
- <session-deployment>
- <entity-deployment>
- <message-driven-deployment>
- <cmp-field-mapping>

<persistence-manager>

<persistence-manager> セクションでは、EJB 2.1 アプリケーションに対してのみ、TopLink 永続性マネージャの追加のデプロイ情報が提供されます。EJB 3.0 アプリケーションでは、OC4J は常に TopLink エンティティ・マネージャを使用します。

<persistence-manager> セクションには、次の構造が含まれています。

```
<persistence-manager name=... class=... descriptor=... >
  <pm-properties>
    <session-name>...</session-name>
    <project-class>...</project-class>
    <db-platform-class>...</db-platform-class>
    <default-mapping db-table-gen=... >...</default-mapping>
    <remote-relationships>...</remote-relationships>
    <cache-synchronization mode=... >...</cache-synchronization>
    <customization-class>...</customization-class>
  </pm-properties>
</persistence-manager>
```

<persistence-manager> 要素の複数の定義は無効です。OC4J が解析時に <persistence-manager> 要素の複数の定義を検出した場合、OC4J は警告メッセージをログに記録します。この場合、OC4J は最初のエントリのみ使用し、後続のエントリを無視します。

永続性マネージャを明示的に指定する場合は、<persistence-manager> 要素の name 属性を使用します。有効な値は次のとおりです。

- toplink: TopLink 永続性マネージャを選択します（デフォルト）。
- orion: 推奨されない Orion 永続性マネージャを選択します。

TopLink 永続性マネージャを使用しており、TopLink デプロイメント・ディスクリプタに toplink-ejb-jar.xml（2-8 ページの「toplink-ejb-jar.xml ファイルとは」を参照）以外の名前を付ける場合は、<persistence-manager> 要素の descriptor 属性を使用して名前を指定します。

<pm-properites> 要素は、TopLink 永続性マネージャにのみ適用されます。

詳細は、次を参照してください。

- 2-15 ページの「EJB 永続性サービスについて」
- 3-4 ページの「JPA 永続性プロバイダのカスタマイズ」
- 3-15 ページの「TopLink EJB 2.1 永続性マネージャのカスタマイズ」
- 『Oracle TopLink 開発者ガイド』の pm-properties の構成に関する項

<session-deployment>

<session-deployment> セクションでは、この JAR ファイル内でデプロイされたセッション Bean の追加のデプロイ情報を提供します。

<session-deployment> セクションには、次の構造が含まれています。

```
<session-deployment pool-cache-timeout=... call-timeout=... copy-by-value=...
      location=... max-instances=... min-instances=... max-tx-retries=...
      tx-retry-wait=... name=... persistence-filename=... replication=...
      timeout=... idletime=... memory-threshold=... max-instances-threshold=...
      resource-check-interval=... passivate-count=... wrapper=...
      local-wrapper=... interceptor-type= ...
  <ior-security-config>
    <transport-config>
      <integrity></integrity>
      <confidentiality></confidentiality>
      <establish-trust-in-target></establish-trust-in-target>
      <establish-trust-in-client></establish-trust-in-client>
    </transport-config>
    <as-context>
      <auth-method></auth-method>
      <realm></realm>
      <required></required>
    </as-context>
    <sas-context>
      <caller-propagation></caller-propagation>
    </sas-context>
  </ior-security-config>
  <env-entry-mapping name=...> </env-entry-mapping
  <ejb-ref-mapping location=... name=... remote=... jndi-properties=... />
  <resource-ref-mapping location=... name=... >
    <lookup-context location=...>
       <context-attribute name=... value=... />
    </lookup-context>
  </resource-ref-mapping>
  <resource-env-ref-mapping location=... name=... />
  <message-destination-ref-mapping location=... name=... />
</session-deployment>
```

注意： 別の方法として、EJB 3.0 アプリケーションでは、OC4J 固有のアノテーション @StatelessDeployment および @StatefulDeployment を使用できます。orion-ejb-jar.xml ファイルの <session-deployment> 構成を使用すると、@StatelessDeployment および @StatefulDeployment 構成をオーバーライドできます。詳細は、5-12 ページの「EJB 3.0 セッション Bean の OC4J 固有のデプロイ・オプションの構成」を参照してください。

これらの要素とサブ要素の詳細は、次を参照してください。

- <session-deployment> の属性
- <ior-security-config>
- <env-entry-mapping>
- <ejb-ref-mapping>
- <resource-ref-mapping>
- <resource-env-ref-mapping>
- <message-destination-ref-mapping>

例

<session-deployment>、@StatefulDeployment または @StatelessDeployment 構成（関連する場合）を含むセッション Bean の例は、次を参照してください。

- 第 4 章「EJB 3.0 セッション Bean の実装」
- 第 11 章「EJB 2.1 セッション Bean の実装」
- 5-12 ページの「EJB 3.0 セッション Bean の OC4J 固有のデプロイ・オプションの構成」

<session-deployment> の属性

表 A-1 に、<session-deployment> 要素の属性とそれに対応する @StatelessDeployment および @StatefulDeployment アノテーションの属性（該当する場合）をリストし、各属性がステートレス・セッション Bean のみ、ステートフル・セッション Bean のみ、またはその両方のいずれに適用されるかを示します。

表 A-1 <session-deployment> 要素の属性

属性	@StatelessDeployment の同等属性	@StatefulDeployment の同等属性	ステートレス	ステートフル	説明
call-timeout	callTimeout	callTimeout	✔	✔	このパラメータは、ビジネス・メソッドまたはライフ・サイクル・メソッドを起動するリソースを待機する最大時間を指定します。これは、ビジネス・メソッドが起動するまでのタイムアウトではありません。 タイムアウトに達すると、TimedOutException がスローされます。これによって、データベース接続が除外されます。 デフォルト値は 90000 ミリ秒です。0（ゼロ）に設定すると、タイムアウトはありません。詳細は、『Oracle Application Server パフォーマンス・ガイド』の EJB の項を参照してください。
copy-by-value	copyByValue	copyByValue	✔	✔	EJB コールのすべての受信および送信パラメータをコピーする（クローンを作成する）かどうか。速度を上げるには、使用するアプリケーションが copy-by-value セマンティクスを前提としないことが確実な場合、この値を false に設定します。デフォルト値は true です。
idletime		idletime		✔	各 Bean に対してアイドル・タイムアウトを設定できます。このタイムアウトが経過すると、非アクティブ化が発生します。この属性を適切な秒数に設定します。デフォルトは 300 秒（5 分）です。この属性を無効にするには、負の数を指定します。

表 A-1 <session-deployment> 要素の属性（続き）

属性	@StatelessDeployment の同等属性	@StatefulDeployment の同等属性	ステートレス	ステートフル	説明
interceptor-type	interceptorType	interceptorType	✔	✔	属性は、OC4J によるインターセプタ・クラスのライフ・サイクルの処理方法を示します。インターセプタのタイプは、bean（ステートフル・クラス）または singleton（ステートレス・クラス）に設定できます。 bean に設定すると、OC4J では、インターセプタ・クラスに関連付けられたセッション Bean インスタンスごとに個別のインターセプタ・クラス・インスタンスが作成されます。これは、EJB 3.0 仕様に準拠した処理です。この場合、インターセプタ・クラスはステートフルである必要があります。 singleton に設定すると、OC4J では、すべてのセッション Bean インスタンスで共有される単一のインターセプタ・クラス・インスタンスが作成されます。この場合、インターセプタ・クラスはステートレスである必要があります。 詳細は、2-14 ページの「シングルトン・インターセプタ」を参照してください。
local-location	localLocation	localLocation	✔	✔	この Enterprise Bean がバインドされるローカル JNDI 名。
local-wrapper			✔	✔	この Bean の OC4J ローカル・ホーム・ラッパー・クラスの名前。内部サーバー値のため、編集しないでください。
location	location	location	✔	✔	この Bean がバインドされる JNDI 名。
max-instances	maxInstances	maxInstances	✔	✔	メモリー内に存在できるインスタンス化またはプールされた Bean インスタンスの数。この値に達すると、OC4J は最低使用頻度（LRU）アルゴリズムを使用して Bean を非アクティブ化しようとします。Bean インスタンスの数を無限に許可する場合は、max-instances 属性を 0（ゼロ）に設定できます。デフォルトは 0（ゼロ）で、無限を意味します。この属性は、ステートレス・セッション Bean およびステートフル・セッション Bean の両方に適用されます。 インスタンス・プーリングを無効にするには、max-instances を負の数に設定します。これにより、EJB コールの開始時に新規インスタンスを作成し、コールの終了時に解放します。 詳細は、次を参照してください。 ■ 12-2 ページの「非アクティブ化基準の構成」 ■ 31-5 ページの「Bean インスタンスのプール・サイズの構成」

表 A-1 <session-deployment> 要素の属性（続き）

属性	@StatelessDeployment の同等属性	@StatefulDeployment の同等属性	ステートレス	ステートフル	説明
max-instances-threshold		maxInstancesThreshold		✔	非アクティブ化が行われるまでにメモリー内に存在できる Bean の max-instances 数に対する割合。 パーセンテージとして解釈される整数を指定します。max-instances を 100、max-instances-threshold を 90% に定義した場合は、アクティブ Bean インスタンス数が 90 以上になると、Bean の非アクティブ化が発生します。デフォルトは 90% です。 この属性を無効にするには、負の数を指定します。
max-tx-retries	maxTransactionRetries	maxTransactionRetries	✔	✔	このパラメータは、システム・レベルの障害によってロールバックされたトランザクションの再試行回数を指定します。デフォルトは 0（ゼロ）です。 ステートフル・セッション Bean では、RuntimeException、Error または RemoteException がスローされた場合に、OC4J は再試行しません。 通常は、再試行によって解決できるエラーがある場合のみ再試行の回数を追加することをお薦めします。たとえば、分離レベル serializable を使用していて、競合が発生したとき自動的にトランザクションが再試行されるようにする場合には、再試行を使用できます。ただし、競合の発生時に Bean に通知する場合は、max-tx-retries=0 のままにしておく必要があります。 詳細は、『Oracle Application Server パフォーマンス・ガイド』の EJB の項を参照してください。
memory-threshold		memoryThreshold		✔	非アクティブ化が発生するまでに使用可能な JVM メモリーの量に対するしきい値を定義します。 パーセンテージとして解釈される整数を指定します。 この値に達すると、アイドル・タイムアウトが経過していない場合でも Bean は非アクティブ化されます。デフォルトは 80% です。 この属性を無効にするには、負の数を指定します。
min-instances	minInstances		✔		インスタンス化またはプールされた状態で維持される Bean 実装インスタンスの最小数。デフォルトは 0（ゼロ）です。この設定が有効なのは、ステートレス・セッション Bean のみです。
name			✔	✔	Bean の名前。これは、EJB デプロイメント・ディスクリプタのアセンブリ・セクション（ejb-jar.xml）内の Bean の名前に一致します。

表 A-1 <session-deployment> 要素の属性（続き）

属性	@StatelessDeployment の同等属性	@StatefulDeployment の同等属性	ステートレス	ステートフル	説明
passivate-count		passivateCount		✔	いずれかのリソースしきい値に達した場合に非アクティブ化される Bean の数を定義する整数です。Bean の非アクティブ化は、最低使用頻度アルゴリズムを使用して実行されます。デフォルトは、max-instances 属性の 1/3 です。この属性を無効にするには、カウントを 0（ゼロ）または負の数に設定します。
persistence-filename		persistenceFileName		✔	再起動のたびにセッションが保存されるファイルへのパス。
pool-cache-timeout	poolCacheTimeout		✔		pool-cache-timeout は、ステートレス・セッション Enterprise Bean に適用されます。このパラメータは、プールにキャッシュされたステートレス・セッションを維持する期間を指定します。 ステートレス・セッション Bean の場合、pool-cache-timeout を指定すると、pool-cache-timeout ごとに、プール内の対応する Bean タイプの Bean がすべて削除されます。値が 0（ゼロ）または負の場合は、pool-cache-timeout が無効になり、Bean はプールから削除されません。 デフォルト値は 60（秒）です。
replication		replicationType		✔	ステートフル・セッション Bean に対する状態レプリケーションの構成。値は、inherited（デフォルト）、onShutdown、onRequestEnd または none のいずれかです。詳細は、2-36 ページの「状態レプリケーション」を参照してください。
resource-check-interval		resourceCheckInterval		✔	コンテナは、すべてのリソースをこの時間間隔でチェックします。この時点でいずれかのしきい値に達している場合は、非アクティブ化が発生します。デフォルトは 180 秒（3 分）です。 この属性を無効にするには、負の数を指定します。

表 A-1 <session-deployment> 要素の属性（続き）

属性	@StatelessDeployment の同等属性	@StatefulDeployment の同等属性	ステートレス	ステートフル	説明
timeout		timeout		✔	ステートフル・セッション Bean がプールのクリーンアップの対象となるまでに非アクティブ状態を継続できる最大秒数。値が 0（ゼロ）または負の数の場合、すべてのタイムアウトが使用禁止になります。 30 秒ごとに、プール・クリーンアップ・ロジックが起動します。プール・クリーンアップ・ロジックの実行中に、タイムアウト値を渡すことによって削除されるのは、タイムアウトしたセッションのみです。 アプリケーションでのステートフル・セッション Bean の使用状況に応じて、タイムアウトを調整します。たとえば、ステートフル・セッション Bean を明示的に削除しないアプリケーションの場合は、多数のステートフル・セッション Bean が作成されるため、タイムアウト値を短い時間に設定できます。 アプリケーションでステートフル・セッション Bean を 1800 秒（30 分）以上使用可能にする必要がある場合は、タイムアウト値をそれにあわせて調整します。 デフォルト値は 1800 秒です。
transaction-timeout	transactionTimeout	transactionTimeout	✔	✔	このステートレスまたはステートフル・セッション Bean によって開始された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のを OC4J が待機する最大秒数。値が 0（ゼロ）または負の数の場合、タイムアウトは無効になります。
tx-retry-wait	transactionRetryWait	transactionRetryWait	✔	✔	このパラメータは、トランザクションの再試行の時間間隔を秒数で指定します。デフォルト値は 60 秒です。
wrapper			✔	✔	この Bean の OC4J ラッパー・クラスの名前。内部サーバー値のため、編集しないでください。

<ior-security-config>

<ior-security-config> 要素は相互運用性の要素です。詳細は、『Oracle Containers for J2EE サービス・ガイド』の相互運用性に関する章で説明されています。

<env-entry-mapping>

<env-entry-mapping> 要素は、環境変数を JNDI 名にマッピングします。
詳細は、19-17 ページの「環境変数への環境参照の構成」で説明されています。

<ejb-ref-mapping>

<ejb-ref-mapping> 要素は、EJB 参照を JNDI 名にマッピングします。
詳細は、19-2 ページの「EJB 環境参照」で説明されています。

<resource-ref-mapping>

<resource-ref-mapping> 要素は、EJB 参照を JNDI 名にマッピングします。
詳細は、19-3 ページの「リソース・マネージャのコネクション・ファクトリ環境参照」で説明されています。

<resource-env-ref-mapping>

<resource-env-ref-mapping> 要素は、リソースの管理オブジェクトをマッピングするために使用されます。たとえば、JMS を使用するために、Bean は JMS ファクトリ・オブジェクトと接続先オブジェクトの両方を取得する必要があります。これらのオブジェクトは、JNDI から同時に取得されます。<resource-ref> 要素で JMS ファクトリを宣言し、<resource-env-ref> 要素を使用して接続先を宣言します。したがって、<resource-env-ref-mapping> 要素は接続先オブジェクトをマッピングします。
詳細は、19-3 ページの「リソース・マネージャのコネクション・ファクトリ環境参照」を参照してください。

<message-destination-ref-mapping>

<message-destination-ref-mapping> 要素は、JMS 1.1 を使用している場合にのみ使用します。この要素を使用してクライアント・デプロイメント・ディスクリプタの message-destination-ref-name を OC4J 環境で使用できる別の場所にマッピングします。これにより、メッセージ・コンシューマおよびプロデューサを 1 つ以上の共通の論理的な宛先にリンクする手段が提供されます。
詳細は、19-14 ページの「JMS 宛先リソース・マネージャのコネクション・ファクトリへの環境参照の構成（JMS 1.1）」を参照してください。

<entity-deployment>

<entity-deployment> セクションでは、この JAR ファイル内でデプロイされた EJB 2.x または EJB 1.1 エンティティ Bean の追加のデプロイ情報を提供します。

> **注意：** <entity-deployment> の属性とサブ要素は、EJB 2.x または EJB 1.1 エンティティ Bean のみに適用されます。これらは JPA エンティティには適用されません。

<entity-deployment> セクションには、次の構造が含まれています。

```
<entity-deployment call-timeout=... clustering-schema=...
      copy-by-value=... data-source=... exclusive-write-access=...
      disable-default-persistent-unit=...
      do-select-before-insert=... isolation=...
      location=... local-location=... locking-mode=...
      max-instances=... min-instances=...
```

```
      max-tx-retries=... tx-retry-wait=... update-changed-fields-only=...
      name=... pool-cache-timeout=...
      table=... validity-timeout=... force-update=...
      wrapper=... local-wrapper=... delay-updates-until-commit=...
      findByPrimaryKey-lazy-loading=... >
  <ior-security-config>
    <transport-config>
      <integrity></integrity>
      <confidentiality></confidentiality>
      <establish-trust-in-target></establish-trust-in-target>
      <establish-trust-in-client></establish-trust-in-client>
    </transport-config>
    <as-context>
      <auth-method></auth-method>
      <realm></realm>
      <required></required>
    </as-context>
    <sas-context>
      <caller-propagation></caller-propagation>
    </sas-context>
  </ior-security-config>
  <primkey-mapping>
   <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
      persistence-type=...></cmp-field-mapping>
  </primkey-mapping>
  <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
      persistence-type=...> </cmp-field-mapping>
  <finder-method partial=... query=... lazy-loading=... prefetch-size=... >
   <method></method>
  </finder-method>
  <env-entry-mapping name=...></env-entry-mapping>
  <ejb-ref-mapping location=... name=... remote=... jndi-properties=... />
  <resource-ref-mapping location=... name=... >
    <lookup-context location=...>
       <context-attribute name=... value=... />
    </lookup-context>
  </resource-ref-mapping>
  <resource-env-ref-mapping location=... name=... />
</entity-deployment>
```

これらの要素とサブ要素の詳細は、次を参照してください。

- <entity-deployment> の属性
- <ior-security-config>
- <primkey-mapping>
- <cmp-field-mapping>
- <finder-method>
- <env-entry-mapping>
- <ejb-ref-mapping>
- <service-ref-mapping>
- <resource-ref-mapping>
- <resource-env-ref-mapping>
- <message-destination-ref-mapping>
- <commit-option>

例

<entity-deployment> 構成（関連する場合）を含むエンティティ Bean の例は、第 13 章「EJB 2.1 エンティティ Bean の実装」を参照してください。

<entity-deployment> の属性

表 A-2 に、<entity-deployment> 要素の属性をリストします。

OC4J による TopLink 永続性サポートの詳細は、A-2 ページの「TopLink 永続性サポート」を参照してください。

表 A-2 <entity-deployment> 要素の属性

属性	orion-ejb-jar.xml ファイルで構成可能	TopLink 永続性 API を使用して構成可能	説明
call-timeout		✔	TopLink 永続性 API を使用して、問合せが結果を返すのを OC4J が待機する最大時間を指定できます。問合せタイムアウトにより、結果を適時に返さないハングした問合せや長時間かかる問合せによってアプリケーションが永久にブロックされることがなくなります。 問合せタイムアウトは、識別子および問合せレベルで指定できます。 識別子レベルの問合せタイムアウトは、識別子の参照クラスのすべての問合せに適用されます。特定のオブジェクト・タイプのすべての問合せに同じタイムアウトを適用するには、識別子レベルの問合せタイムアウトを指定します。 問合せレベルの問合せタイムアウトは、その問合せにのみ適用されます。 詳細は、次を参照してください。 ■ 『Oracle TopLink 開発者ガイド』のディスクリプタ・レベルでの問合せタイムアウトの構成に関する項 『Oracle TopLink 開発者ガイド』の名前付き問合せの詳細オプションの構成に関する項 ■ 『Oracle TopLink 開発者ガイド』の問合せレベルでの問合せタイムアウトの構成に関する項
clustering-schema	✔		使用しません。このリリースでは必要ありません。
copy-by-value	✔		EJB コールのすべての受信および送信パラメータをコピーする（クローンを作成する）かどうか。速度を上げるには、使用するアプリケーションが copy-by-value セマンティクスを前提としないことが確実な場合、この値を false に設定します。 デフォルト値は true です。
data-source	✔		コンテナ管理の永続性を使用している場合、データソースの名前。
delay-updates-until-commit		✔	TopLink 永続性 API を使用して、遅延または非遅延変更用に OC4J を構成できます。デフォルトでは、TopLink はコミット時まですべての変更を遅延します。これは、データソースの対話回数を最小にする最も効率的なアプローチです。または、非遅延変更用にエンティティ Bean の識別子を構成できます。これは、エンティティ Bean の永続フィールドを変更すると、OC4J によってリレーショナル・スキーマが即時に変更されることを意味します。 詳細は、『Oracle TopLink 開発者ガイド』の非遅延変更に関する項を参照してください。

表 A-2 <entity-deployment> 要素の属性（続き）

属性	orion-ejb-jar.xml ファイルで構成可能	TopLink 永続性 API を使用して構成可能	説明
`disable-default-persistent-unit`	✔		デフォルトでは、アプリケーションが OC4J のデフォルトの永続性ユニットのみ使用している場合に、OC4J では `persistence.xml` ファイルがなくても EJB 3.0 エンティティをデプロイできます。この機能を無効にするには、この属性を `true` に設定します。 デフォルトは `false` です。 詳細は、2-10 ページの「OC4J の永続性ユニットのデフォルトについて」を参照してください。
`do-select-before-insert`			TopLink では、変更を書き出す前に選択を実行しません。同時上書きの可能性に対処するために、オプティミスティック・ロックを使用することをお薦めします。 詳細は、1-64 ページの「同時実行性（ロック）モード」を参照してください。
`exclusive-write-access`		✔	TopLink 永続性 API を使用している場合、TopLink は作業ユニット・トランザクション領域を使用して変更セットを計算し、変更を書き出すため、OC4J ではエンティティ・インスタンスへの排他的書込みアクセスが前提とされます。作業ユニット・トランザクション領域は、共有セッション・キャッシュから分離されています。 詳細は、『Oracle TopLink 開発者ガイド』の作業ユニット・アーキテクチャに関する項を参照してください。
`findByPrimaryKey-lazy-loading`		✔	TopLink 永続性 API を使用している場合は、フェッチ・グループを構成できます。これにより、Bean の属性のサブセットを取得できます。これは、遅延ロードと同等です。 詳細は、『Oracle TopLink 開発者ガイド』のフェッチ・グループによる問合せの使用に関する項を参照してください。
`force-update`		✔	TopLink 永続性 API を使用している場合は、いずれかの永続データが変更されたことを OC4J が認識していない場合でも OC4J で永続性関連のライフ・サイクル・メソッドを実行するかどうかを構成できます。 `true` に設定されている場合、このオプションは、OC4J が `ejbStore` メソッドを起動することで EJB ライフ・サイクルを実行することを意味します。これによって、一時フィールドのデータが管理され、`ejbStore` メソッドの実行時に適切な永続フィールドが設定されます。たとえば、イメージを、メモリー内に保持している形式とは異なる形式で、データベース内に格納することができます。 デフォルト値は `false` です。 詳細は、『Oracle TopLink 開発者ガイド』の EJB 情報を使用したディスクリプタの構成に関する項を参照してください。

表 A-2 <entity-deployment> 要素の属性（続き）

属性	orion-ejb-jar.xml ファイルで構成可能	TopLink 永続性 API を使用して構成可能	説明
isolation		✔	TopLink 永続性 API を使用している場合、これはオブジェクト・キャッシュと作業ユニット・トランザクション領域を提供するため、データベース・トランザクション分離レベルは TopLink に関連しません。かわりに、TopLink 作業ユニットおよびキャッシュ分離レベルの構成を検討してください。 データベース分離レベルを通じたロックの処理は、めったに行われません。通常、ロックはオプティミスティックまたはペシミスティック・ロックを通じて行われます。 TopLink データベース・ログインに対してトランザクション分離レベルを構成できます。この設定は、データベース・ログインを使用するすべての Bean およびトランザクションに適用されます。 デフォルトでは、TopLink はデータベースに対して設定されている分離レベルを使用します。 詳細は、次を参照してください。 ■ 1-63 ページの「データベース・リソースの競合の回避」 ■ 1-64 ページの「同時実行性（ロック）モード」 ■ 『Oracle TopLink 開発者ガイド』のデータベースのトランザクション分離レベルに関する項 ■ 『Oracle Application Server パフォーマンス・ガイド』
local-location	✔		この Bean がバインドされるローカル JNDI 名を定義します。
local-wrapper	✔		この Bean の OC4J ローカル・ホーム・ラッパー・クラスの名前。内部サーバー値のため、編集しないでください。
location	✔		この Bean がバインドされる JNDI 名。
locking-mode		✔	TopLink 永続性 API を使用して、次のロック・モードを構成できます。 ■ オプティミスティック・ロック : 複数のユーザーがデータに対する読取りアクセス権を持ちます。ユーザーが変更を行おうとすると、アプリケーション・チェックが行われ、ユーザーがデータを読み取った後にデータが変更されていないことが確認されます。TopLink では、バージョン（推奨）、タイムスタンプおよびフィールドレベルのロックがサポートされます。 ■ ペシミスティック・ロック : 更新する目的でデータにアクセスする最初のユーザーが、更新の完了までデータをロックします。これによりリソースの競合が管理され、パラレル実行はできません。エンティティ Bean を実行できるのは、一度に 1 ユーザーのみです。 ■ 読取り専用 : 複数のユーザーがパラレルでエンティティ Bean を実行できます。コンテナでは、Bean の状態を更新できません。 詳細は、1-64 ページの「同時実行性（ロック）モード」を参照してください。
max-instances	✔		インスタンス化またはプールされた状態で維持される Bean 実装インスタンスの最大数。デフォルトは 0（ゼロ）で、無限を意味します。 インスタンス・プーリングを無効にするには、max-instances を負の数に設定します。これにより、EJB コールの開始時に新規インスタンスを作成し、コールの終了時に解放します。 詳細は、31-5 ページの「Bean インスタンスのプール・サイズの構成」を参照してください。

表 A-2 <entity-deployment> 要素の属性（続き）

属性	orion-ejb-jar.xml ファイルで構成可能	TopLink 永続性 API を使用して構成可能	説明
max-tx-retries	✔		このパラメータは、システム・レベルの障害によってロールバックされたトランザクションの再試行回数を指定します。 デフォルト値は 0（ゼロ）です。 通常は、再試行によって解決できるエラーがある場合のみ再試行の回数を追加することをお薦めします。たとえば、分離レベル serializable を使用していて、競合が発生したとき自動的にトランザクションが再試行されるようにする場合には、再試行を使用できます。ただし、競合の発生時に Bean に通知する場合は、max-tx-retries=0 のままにしておく必要があります。 詳細は、『Oracle Application Server パフォーマンス・ガイド』の EJB の項を参照してください。
min-instances	✔		インスタンス化またはプールされた状態で維持される Bean 実装インスタンスの最小数。 デフォルト値は 0（ゼロ）です。 詳細は、31-5 ページの「Bean インスタンスのプール・サイズの構成」を参照してください。
name	✔		Bean の名前。これは、EJB デプロイメント・ディスクリプタのアセンブリ・セクション（ejb-jar.xml）内の Bean の名前に一致します。
pool-cache-timeout	✔		Bean 実装インスタンスを、プールされた（割り当てられていない）状態で維持する秒単位の時間。負の数を指定すると、ガベージ・コレクションが行われるまでインスタンスを維持します。デフォルトは 60 です。詳細は、31-8 ページの「エンティティ Bean の Bean インスタンス・プール・タイムアウトの構成」を参照してください。
table		✔	TopLink 永続性 API を使用して、この Bean に関連付けられているデータベース表の名前を指定できます。 詳細は、『Oracle TopLink 開発者ガイド』の関連表の構成に関する項を参照してください。
tx-retry-wait	✔		このパラメータは、トランザクションの再試行の時間間隔を秒数で指定します。デフォルトは 60 秒です。[1]
update-changed-fields-only		✔	TopLink 永続性 API を使用している場合、TopLink 作業ユニットは変更セットを常に計算し、変更されたフィールドのみについて更新文を生成します。
validity-timeout		✔	TopLink 永続性 API を使用して、無効化ポリシーを構成できます。 詳細は、『Oracle TopLink 開発者ガイド』のキャッシュの無効化に関する項を参照してください。
wrapper	✔		この Bean の OC4J リモート・ホーム・ラッパー・クラスの名前。内部サーバー値のため、編集しないでください。

[1] この属性を指定した場合、XML 検証は有効化できません（31-9 ページの「XML ファイルの検証」を参照）。

<ior-security-config>

<ior-security-config> 要素は、相互運用性の CSIv2 セキュリティ・ポリシーを構成します。詳細は、『Oracle Containers for J2EE サービス・ガイド』の相互運用性に関する章で説明されています。

<primkey-mapping>

<primkey-mapping> 要素は、主キーを対応するコンテナ管理の永続性フィールドにマッピングします。このリリースでは、この機能は orion-ejb-jar.xml ファイルで構成されません。OC4J により自動的に構成されます。この機能を手動で構成するには、TopLink 永続性 API を使用します。

詳細は、次を参照してください。

- 3-4 ページの「JPA 永続性プロバイダのカスタマイズ」
- 3-15 ページの「TopLink EJB 2.1 永続性マネージャのカスタマイズ」
- 『Oracle TopLink 開発者ガイド』の「リレーショナル・プロジェクトにおける順序付けの概要」

<cmp-field-mapping>

コンテナ管理の永続性を備えた EJB 1.1 エンティティ Bean を使用する場合は、<cmp-field-mapping> 要素を使用して、コンテナ管理の永続性フィールドをデータベースにマッピングします。

次に、コンテナ管理の永続性を備えた EJB 1.1 エンティティ Bean 用の orion-ejb-jar.xml ファイル内で、コンテナ管理の永続データ・フィールドのマッピングに使用される XML 要素を示します。

```
<cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
      persistence-type=...>
   <fields>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
          persistence-type=...></cmp-field-mapping>
   </fields>
   <properties>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
           persistence-type=...></cmp-field-mapping>
   </properties>
   <entity-ref home=...>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
          persistence-type=...></cmp-field-mapping>
   </entity-ref>
   <collection-mapping table=...>
      <primkey-mapping>
         <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
             persistence-type=...></cmp-field-mapping>
      </primkey-mapping>
      <value-mapping immutable="true|false" type=...>
         <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
             persistence-type=...></cmp-field-mapping>
      </value-mapping>
   </collection-mapping>
   <set-mapping table=...>
      <primkey-mapping>
         <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
             persistence-type=...></cmp-field-mapping>
      </primkey-mapping>
      <value-mapping immutable="true|false" type=...>
         <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
             persistence-type=...></cmp-field-mapping>
```

```
      </value-mapping>
    </set-mapping>
</cmp-field-mapping>
```

EJB 3.0 エンティティおよび EJB 2.1 エンティティ Bean では、この機能は orion-ejb-jar.xml ファイルで構成されません。OC4J により自動的に構成されます。この機能を手動で構成するには、TopLink 永続性 API を使用します。

詳細は、次を参照してください。

- 1-45 ページの「コンテナ管理の永続性フィールドとは」
- 3-4 ページの「JPA 永続性プロバイダのカスタマイズ」
- 3-15 ページの「TopLink EJB 2.1 永続性マネージャのカスタマイズ」
- 『Oracle TopLink 開発者ガイド』の「リレーショナル・マッピングの概要」

<finder-method>

<finder-method> 要素は、EJB 1.1 エンティティ Bean の finder メソッドを作成するために使用されます。

詳細は、次を参照してください。

- 1-42 ページの「JPA エンティティの問合せ方法」
- 1-53 ページの「EJB 2.1 エンティティ Bean の問合せ方法」

<env-entry-mapping>

<env-entry-mapping> 要素は、環境変数を JNDI 名にマッピングします。詳細は、19-17 ページの「環境変数への環境参照の構成」で説明されています。

<ejb-ref-mapping>

<ejb-ref-mapping> 要素は、EJB 参照を JNDI 名にマッピングします。詳細は、19-2 ページの「EJB 環境参照」で説明されています。

<service-ref-mapping>

<service-ref-mapping> 要素は、EJB 参照を Web サービスにマッピングします。詳細は、19-18 ページの「Web サービスへの環境参照の構成」で説明されています。

<resource-ref-mapping>

<resource-ref-mapping> 要素は、EJB 参照を JNDI 名にマッピングします。詳細は、19-3 ページの「リソース・マネージャのコネクション・ファクトリ環境参照」で説明されています。

<resource-env-ref-mapping>

<resource-env-ref-mapping> 要素は、リソースの管理オブジェクトをマッピングするために使用されます。たとえば、JMS を使用するために、Bean は JMS ファクトリ・オブジェクトと接続先オブジェクトの両方を取得する必要があります。これらのオブジェクトは、JNDI から同時に取得されます。<resource-ref> 要素で JMS ファクトリを宣言し、<resource-env-ref> 要素を使用して接続先を宣言します。したがって、<resource-env-ref-mapping> 要素は接続先オブジェクトをマッピングします。詳細は、19-3 ページの「リソース・マネージャのコネクション・ファクトリ環境参照」を参照してください。

<message-destination-ref-mapping>

<message-destination-ref-mapping> 要素は、JMS 1.1 を使用している場合にのみ使用します。この要素を使用してクライアント・デプロイメント・ディスクリプタの message-destination-ref-name を OC4J 環境で使用できる別の場所にマッピングします。これにより、メッセージ・コンシューマおよびプロデューサを 1 つ以上の共通の論理的な宛先にリンクする手段が提供されます。詳細は、19-14 ページの「JMS 宛先リソース・マネージャのコネクション・ファクトリへの環境参照の構成（JMS 1.1）」を参照してください。

<commit-option>

<commit-option> 要素により、トランザクション・コミット時にエンティティ Bean インスタンスの状態が決定され、OC4J が特定のアプリケーション条件を最適化できる柔軟性が提供されます。詳細は、1-51 ページの「エンティティ Bean のコミット・オプション」で説明されています。

<message-driven-deployment>

<message-driven-deployment> セクションでは、この JAR ファイル内でデプロイされたメッセージドリブン Bean の追加のデプロイ情報を提供します。

<message-driven-deployment> セクションには、次の構造が含まれています。

```
<message-driven-deployment cache-timeout=... connection-factory-location=...
      destination-location=... name=... subscription-name=...
      listener-threads=... transaction-timeout=...
      dequeue-retry-count=... dequeue-retry-interval=... interceptor-type=... >
  <env-entry-mapping name=...></env-entry-mapping>
  <ejb-ref-mapping location=... name=... remote=... jndi-properties=... />
  <resource-ref-mapping location=... name=... >
    <lookup-context location=...>
       <context-attribute name=... value=... />
    </lookup-context>
  </resource-ref-mapping>
  <resource-env-ref-mapping location=... name=... />
  <message-destination-ref-mapping location=... name=... />
  <config-property>
    <config-property-name> ... </config-property-name>
    <config-property-value> ... </config-property-value>
  </config-property>
</message-driven-deployment>
```

> **注意：** 別の方法として、EJB 3.0 アプリケーションでは、OC4J 固有のアノテーション @MessageDrivenDeployment を使用できます。orion-ejb-jar.xml ファイルの <message-driven-deployment> 構成を使用すると、@MessageDrivenDeployment 構成をオーバーライドできます。詳細は、10-20 ページの「EJB 3.0 MDB の OC4J 固有のデプロイ・オプションの構成」を参照してください。

これらの要素とサブ要素の詳細は、次を参照してください。

- <message-driven-deployment> の属性
- <env-entry-mapping>
- <ejb-ref-mapping>
- <resource-ref-mapping>
- <resource-env-ref-mapping>
- <message-destination-ref-mapping>

- <config-property>

例

<message-driven-deployment> 要素を含むメッセージドリブン Bean の例は、次を参照してください。

- 第 9 章「EJB 3.0 メッセージドリブン Bean の実装」
- 第 17 章「EJB 2.1 メッセージドリブン Bean の実装」
- 5-12 ページの「EJB 3.0 セッション Bean の OC4J 固有のデプロイ・オプションの構成」

<message-driven-deployment> の属性

表 A-3 に、メッセージ・サービス・オプションを構成するのに使用できる <message-driven-deployment> 要素の属性をリストします。この表には、対応する次の代替構成も含まれます。

- @MessageDrivenDeployment アノテーションの属性
- 次のアノテーションおよび要素で使用できるアクティブ化構成プロパティ名
  - <message-driven-deployment> 要素に所有される <config-property> 要素
  - @MessageDrivenDeployment または @MessageDriven アノテーションに所有される @ActivationConfigProperty アノテーション

注意： 属性を使用した構成の場合、アプリケーションでは、J2CA リソース・アダプタなしでメッセージ・サービス・プロバイダにアクセスすることのみ可能です。後から J2CA リソース・アダプタを使用してメッセージ・サービス・プロバイダにアクセスすると、アプリケーションはデプロイに失敗します。アクティブ化構成プロパティを使用した構成の場合、アプリケーションでは、J2CA リソース・アダプタの有無にかかわらずメッセージ・サービス・プロバイダにアクセスできます。<config-property> または @ActivationConfigProperty オプションを使用することをお薦めします。

詳細は、次を参照してください。

- 2-31 ページの「メッセージ・サービス構成オプション : アノテーションまたは XML の選択と属性またはアクティブ化構成プロパティの選択」
- 2-26 ページの「MDB で使用できるメッセージ・サービス・プロバイダ」

表 A-3 <message-driven-deployment> 要素の属性

<message-driven-deployment> の属性	@MessageDrivenDeployment の属性	説明	アクティブ化構成プロパティ名[1]
cache-timeout	poolCacheTimeout	このパラメータは、プールにキャッシュされたメッセージドリブン Bean を維持する期間を指定します。 プールの cache-timeout を指定すると、キャッシュ・タイムアウト間隔ごとに、プール内の対応する Bean タイプの Bean がすべて削除されます。値が 0（ゼロ）または負の場合は、キャッシュ・タイムアウトが無効になり、Bean はプールから削除されません。 デフォルト値は 60（秒）です。	N/A
connection-factory-location	N/A	使用するコネクション・ファクトリの JNDI の場所。JMS の Destination Connection Factory は、この属性で指定されます。構文は、java:comp/resource + リソース・プロバイダ名 + TopicConnectionFactories または QueueConnectionFactories + ユーザー定義名です。nnnConnectionFactories は、定義するファクトリのタイプを指定します。	表 B-1 の「ConnectionFactoryJndiName」を参照
dequeue-retry-count	dequeueRetryCount	データベース・フェイルオーバーの発生後、リスナー・スレッドによって試行される JMS セッションの再取得頻度を指定します。これは、MDB のコンテナ管理のトランザクションにのみ適用されます。 デフォルト値は 0（ゼロ）です。 詳細は、次を参照してください。 ■ 18-10 ページの「EJB 2.1 MDB の接続障害リカバリの構成」 ■ 2-35 ページの「OC4J EJB アプリケーション・クラスタリング・サービスについて」	N/A 表 B-2 の「EndpointFailureRetryInterval」を参照
dequeue-retry-interval	dequeueRetryInterval	再試行の間隔を指定します。 デフォルト値は 60 秒です。 詳細は、次を参照してください。 ■ 18-10 ページの「EJB 2.1 MDB の接続障害リカバリの構成」 ■ 2-35 ページの「OC4J EJB アプリケーション・クラスタリング・サービスについて」	N/A 表 B-2 の「EndpointFailureRetryInterval」を参照
destination-location	destinationLocation	使用する接続先（キュー、トピック）の JNDI の場所。JMS の Destination は、destination-location 属性で指定されます。構文は、java:comp/resource + リソース・プロバイダ名 + Topics または Queues + Destination 名です。Topic または Queue は、定義される Destination タイプを指定します。Destination 名は、データベースで定義された実際のキュー名またはトピック名です。	表 B-1 の「DestinationName」を参照

表 A-3 <message-driven-deployment> 要素の属性（続き）

<message-driven-deployment> の属性	@MessageDrivenDeployment の属性	説明	アクティブ化構成プロパティ名[1]
interceptor-type	interceptorType	属性は、OC4J によるインターセプタ・クラスのライフ・サイクルの処理方法を示します。インターセプタのタイプは、bean（ステートフル・クラス）または singleton（ステートレス・クラス）に設定できます。 bean に設定すると、OC4J では、インターセプタ・クラスに関連付けられたセッション Bean インスタンスごとに個別のインターセプタ・クラス・インスタンスが作成されます。これは、EJB 3.0 仕様に準拠した処理です。この場合、インターセプタ・クラスはステートフルである必要があります。 singleton に設定すると、OC4J では、すべてのセッション Bean インスタンスで共有される単一のインターセプタ・クラス・インスタンスが作成されます。この場合、インターセプタ・クラスはステートレスである必要があります。 詳細は、2-14 ページの「シングルトン・インターセプタ」を参照してください。	N/A
listener-threads	listenerThreads	リスナー・スレッドは、JMS メッセージを並行してコンシュームするために使用されます。デフォルトのスレッドは 1 つです。トピックに指定できるスレッドは 1 つのみです。キューには複数のスレッドを指定できます。 詳細は、18-8 ページの「パラレル・メッセージ処理の構成」を参照してください。	表 B-2 の「ReceiverThreads」を参照
max-delivery-count	maxDeliveryCount	メッセージドリブン Bean の onMessage メソッドがエラーを返した場合（確認応答操作の起動に失敗した場合、例外をスローした場合、またはその両方の場合）に OC4J がこのメソッドにメッセージの即時再配信を試行する最大回数です。この回数の再配信が行われた後で、メッセージは配信不能とみなされ、メッセージ・サービス・プロバイダのポリシーに従って処理されます。たとえば、OEMS JMS はその例外キュー（jms/Oc4jJmsExceptionQueue）にメッセージを挿入します。 詳細は、18-9 ページの「最大配信数の構成」を参照してください。	表 B-2 の「MaxDeliveryCnt」を参照
max-instances	maxInstances	インスタンス化またはプールされた状態で維持される Bean 実装インスタンスの最大数。デフォルトは 0（ゼロ）で、無限を意味します。 インスタンス・プーリングを無効にするには、max-instances を負の数に設定します。これにより、EJB コールの開始時に新規インスタンスを作成し、コールの終了時に解放します。 メッセージドリブン Bean の場合、通常はデフォルトのプーリング設定が適切です。この値は、MDB ライフ・サイクル・メソッドのコストが非常に高く、プール内でのインスタンスの作成および管理の頻度に対してファイングレインな制御を行う必要がある場合にのみ変更します。 詳細は、31-5 ページの「Bean インスタンスのプール・サイズの構成」を参照してください。	N/A
min-instances	minInstances	インスタンス化またはプールされた状態で維持される Bean 実装インスタンスの最小数。 デフォルト値は 0（ゼロ）です。 詳細は、31-5 ページの「Bean インスタンスのプール・サイズの構成」を参照してください。	N/A

表 A-3 <message-driven-deployment> 要素の属性（続き）

<message-driven-deployment> の属性	@MessageDrivenDeployment の属性	説明	アクティブ化構成プロパティ名[1]
name	name	Bean の名前。これは、EJB デプロイメント・ディスクリプタのアセンブリ・セクション（ejb-jar.xml）内の Bean の名前に一致します。	N/A
resource-adapter	resourceAdapter	この MDB が使用するリソース・アダプタ・インスタンスの名前。この MDB が J2CA メッセージ・サービス・プロバイダを使用している場合にのみ適用されます。リソース・アダプタにより受信されたメッセージで MDB をアクティブにするには、MDB とリソース・アダプタを接続する必要があります。 詳細は、23-2 ページの「メッセージ・サービス・プロバイダで使用するための J2CA リソース・アダプタの構成」を参照してください。	N/A
subscription-name	subscriptionName	このメッセージドリブン Bean がサブスクライブするトピックの名前。	表 B-2 の「SubscriptionName」を参照
transaction-timeout	transactionTimeout	この属性によって、コンテナ管理のトランザクション MDB のトランザクション・タイムアウト時間（秒）を制御します。デフォルトは 1 日（86,400 秒）です。この時間枠内でトランザクションが完了しない場合、トランザクションはロールバックされます。このことは、通常の JMS と J2CA リソース・アダプタベースのメッセージ・プロバイダの両方に適用されます。 詳細は、21-9 ページの「メッセージドリブン Bean のトランザクション・タイムアウトの構成」を参照してください。	表 B-2 の「TransactionTimeout」を参照

[1] <message-driven-deployment> 要素の <config-property> サブ要素、または @MessageDrivenDeployment あるいは @MessageDriven アノテーションに所有される @ActivationConfigProperty アノテーションで使用される場合。

<env-entry-mapping>

<env-entry-mapping> 要素は、環境変数を JNDI 名にマッピングします。詳細は、19-17 ページの「環境変数への環境参照の構成」で説明されています。

<ejb-ref-mapping>

<ejb-ref-mapping> 要素は、EJB 参照を JNDI 名にマッピングします。詳細は、19-2 ページの「EJB 環境参照」で説明されています。

<resource-ref-mapping>

<resource-ref-mapping> 要素は、リソース・マネージャ参照を JNDI 名にマッピングします。詳細は、19-3 ページの「リソース・マネージャのコネクション・ファクトリ環境参照」で説明されています。

<resource-env-ref-mapping>

<resource-env-ref-mapping> 要素は、リソースの管理オブジェクトをマッピングするために使用されます。たとえば、JMS を使用するために、Bean は JMS ファクトリ・オブジェクトと接続先オブジェクトの両方を取得する必要があります。これらのオブジェクトは、JNDI から同時に取得されます。<resource-ref> 要素で JMS ファクトリを宣言し、<resource-env-ref> 要素を使用して接続先を宣言します。したがって、<resource-env-ref-mapping> 要素は接続先オブジェクトをマッピングします。詳細は、19-15 ページの「JMS 宛先またはコネクション・リソース・マネージャのコネクション・ファクトリへの環境参照の構成（JMS 1.0）」を参照してください。

<message-destination-ref-mapping>

<message-destination-ref-mapping> 要素は、JMS 1.1 を使用している場合にのみ使用します。この要素を使用してクライアント・デプロイメント・ディスクリプタの message-destination-ref-name を OC4J 環境で使用できる別の場所にマッピングします。これにより、メッセージ・コンシューマおよびプロデューサを 1 つ以上の共通の論理的な宛先にリンクする手段が提供されます。詳細は、19-14 ページの「JMS 宛先リソース・マネージャのコネクション・ファクトリへの環境参照の構成（JMS 1.1）」を参照してください。

<config-property>

<config-property> 要素は、J2CA メッセージ・サービス・プロバイダを使用している場合にのみ使用されます。この要素を使用して J2CA リソース・アダプタ構成プロパティを設定します。J2CA メッセージ・サービス・プロバイダを使用するように構成されている MDB を OC4J がデプロイする場合、OC4J はリソース・アダプタに MDB のアクティブ化仕様を提供します。この仕様には、<config-property> 要素で設定したプロパティが含まれます。

別の方法として、EJB 3.0 メッセージドリブン Bean では、@MessageDriven の属性 configProperty および @ActivationConfig アノテーションを使用して J2CA リソース・アダプタ構成プロパティを設定できます。

orion-ejb-jar.xml ファイルの <config-property> 構成を使用すると、@MessageDriven 構成をオーバーライドできます。

詳細は、次を参照してください。

- 10-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」
- 18-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」

<assembly-descriptor>

個別の Bean に対するデプロイ情報の指定以外に、<assembly-descriptor> セクションで、セキュリティ用の追加デプロイ・マッピング情報を指定できます。<assembly-descriptor> セクションには、次の構造が含まれています。

```
<assembly-descriptor>
  <security-role-mapping impliesAll=... name=...>
   <group name=... />
   <user name=... />
  </security-role-mapping>
  <message-destination-mapping location=... name=...>
  </message-destination-mapping>
  <default-method-access>
   <security-role-mapping impliesAll=... name=...>
    <group name=... />
    <user name=... />
   </security-role-mapping>
  </default-method-access>
</assembly-descriptor>
```

これらの要素とサブ要素の詳細は、次を参照してください。

- <security-role-mapping>
- <message-destination-mapping>
- <default-method-access>
- <method>

例

<assembly-descriptor> 要素の構成例は、次を参照してください。

- 22-3 ページの「EJB デプロイメント・ディスクリプタでの論理ロールの指定」
- 17-2 ページの「EJB 2.1 MDB の実装」

<security-role-mapping>

<security-role-mapping> 要素については、22-9 ページの「ユーザーおよびグループへの論理ロールのマッピング」で説明されています。

<message-destination-mapping>

<default-method-access>

<default-method-access> 要素については、22-10 ページの「未定義メソッドに対するデフォルト・ロール・マッピングの指定」で説明されています。

<method>

<method> 要素は、次のように Enterprise Bean のメソッド（および場合によってはそのパラメータ）の指定に使用します。

```
<method>
    <description></description>
    <ejb-name></ejb-name>
    <method-intf></method-intf>
    <method-name></method-name>
    <method-params>
        <method-param></method-param>
    </method-params>
</method>
```

<method> 要素は、次のいずれかのスタイルを使用して構成できます。

- 指定した Enterprise Bean のホーム・インタフェースおよびリモート・インタフェースのすべてのメソッドを指す場合、次のようにしてメソッドを指定します。

  ```
  <method>
      <ejb-name>EJBNAME</ejb-name>
      <method-name>*</method-name>
  </method>
  ```

- 同じオーバーロードされた名前を持つ複数のメソッドを指す場合、次のようにしてメソッドを指定します。

  ```
  <method>
      <ejb-name>EJBNAME</ejb-name>
      <method-name>METHOD</method-name>
  </method>
  ```

- オーバーロードされた名前を持つ一連のメソッドのうちの 1 つのメソッドを指す場合、次のようにしてメソッド内の各パラメータを指定します。

  ```
  <method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>METHOD</method-name>
    <method-params>
      <method-param>PARAM-1</method-param>
      <method-param>PARAM-2</method-param>
  ```

```
      ...
      <method-param>PARAM-n</method-param>
    </method-params>
  </method>
```

B

J2CA アクティブ化構成プロパティ

この付録では、Oracle JMS コネクタなどの J2CA コネクタを使用して JMS メッセージ・プロバイダにアクセスする際のメッセージ・サービス・オプションを指定できる J2EE Connector Architecture（J2CA）アクティブ化構成プロパティについて説明します。

表 B-1 に、設定する必要のある必須の J2CA アクティブ化構成プロパティをリストします。

表 B-2 に、設定可能なオプションの J2CA アクティブ化構成プロパティをリストします。

すべてのアクティブ化構成プロパティの完全なリストは、`http://www.oracle.com/technology/tech/java/oc4j/1013/how_to/index.html` から `how-to-gjra-with-<RESOURCE-PROVIDER-NAME>.zip` ファイルの 1 つをダウンロードして解凍します。`<RESOURCE-PROVIDER-NAME>` は関連リソース・プロバイダの名前です。`orion-ejb-jar.xml` デモ・ファイルには、すべてのアクティブ化構成プロパティを説明するコメントが含まれています。

詳細は、次を参照してください。

- 2-31 ページの「メッセージ・サービス構成オプション : アノテーションまたは XML の選択と属性またはアクティブ化構成プロパティの選択」
- 10-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 3.0 MDB の構成」
- 18-2 ページの「J2CA を使用してメッセージ・サービス・プロバイダにアクセスするための EJB 2.1 MDB の構成」
- 『Oracle Containers for J2EE サービス・ガイド』の JMS リソース・アダプタに関する項

表 B-1 必須の J2CA @ActivationConfigProperty 属性

プロパティ名	値
`ConnectionFactoryJndiName`	メッセージ・サービス・プロバイダ・コネクション・ファクトリの JNDI 名。メッセージ・サービス・プロバイダの構成時にこの名前を定義します。
`DestinationName`	メッセージ・サービス・プロバイダ接続先名の JNDI 名。メッセージ・サービス・プロバイダの構成時にこの名前を定義します。
`DestinationType`	メッセージ・サービス・プロバイダの接続先タイプの完全修飾 `String` クラス名。JMS MDB では、`javax.jms.Queue` または `javax.jms.Topic` です。

表 B-2 オプションの J2CA @ActivationConfigProperty 属性

プロパティ名	値
AcknowledgeMode	メッセージをコンシュームし、メッセージドリブン Bean のメッセージ・リスナー・メソッド（JMS メッセージ・リスナーの onMessage メソッドなど）をコールするリスナー・スレッドがメッセージの配信を確認する方法。 有効な値は次のとおりです。 ■ Auto-acknowledge（デフォルト）: リスナー・スレッドは、メッセージが MDB に受信されると同時に確認応答を送信します。 ■ Dups-ok-acknowledge: リスナー・スレッドは、確認応答を遅延送信します。これにより、パフォーマンスが向上する可能性があります。MDB では、確認応答が実際に送信されるまでメッセージを重複して受信できます。このオプションを選択する場合、MDB では重複メッセージを処理できる必要があります。
ClientId	リスナー・スレッドがメッセージドリブン Bean のかわりに取得した接続に設定する String 名。 デフォルトは名前なしです。
EndpointFailureRetryInterval	リスナー・スレッドの数が 0（ゼロ）に低下するなんらかの障害の発生後に、OC4J が新規リスナー・スレッドを開始して JMS プロバイダへの再接続を試行するまでに待機するミリ秒数。 リスナー・スレッドは、JMS プロバイダの障害時に自動的に終了します。たとえば、MessageConsumer のメソッド receive をコールするリスナー・スレッドに例外が返された場合などです。持続的なネットワーク障害や JMS サーバーの停止などを原因としてエンドポイントの実行中にすべてのリスナー・スレッドが終了すると、MDB ではメッセージを受信できません。この場合、Oracle JMS コネクタは、EndpointFailureRetryInterval ミリ秒ごとに新規リスナー・スレッドを開始して JMS プロバイダへの再接続を試行します。この処理は、成功するか、エンドポイントが停止するまで続きます。リスナー・スレッドの作成に成功すると、通常のリスナー・スレッド管理が再開します（「ReceiverThreads」、「ListenerThreadMaxIdleDuration」および「ListenerThreadMinBusyDuration」を参照）。 接続障害リカバリの結果として、次の点を考慮してください。 ■ メッセージの順序 : 接続障害からのリカバリでは、新規 JMS セッションを作成する必要がありますが、JMS メッセージの順序の保証は単一のセッション内に受信されたメッセージにのみ適用されるため、再接続後に受信されたメッセージの順序は、再接続前に受信されたメッセージに対して不適切となる可能性があります。 ■ メッセージの紛失 : エンドポイントが非永続サブスクライバの場合、メッセージは失われるか、重複する可能性があります。この問題は、キューを使用している場合、または永続サブスクライバを備えたトピックを使用している場合は発生しません。 これらの問題が発生するかどうかは、使用している JMS プロバイダの特定の動作によって決まります。 デフォルトは 60000 ミリ秒です。
ExceptionQueueName	例外キューとして使用する javax.jms.Queue オブジェクトの JNDI 名。 このプロパティは、UseExceptionQueue が true の場合は必須で、false の場合は無視されます。
IncludeBodiesInExceptionQueue	例外キューに送信されるメッセージにメッセージ・ボディを含めるかどうかを指定します。 有効な値は次のとおりです。 ■ true（デフォルト）: OC4J は、例外キューに送信されるメッセージにメッセージ・ボディを含めます。 ■ false: OC4J は、例外キューに送信されるメッセージにメッセージ・ボディを含めません。通常の操作中に多くのメッセージが例外キューに送信される状況で、そのメッセージ・ボディが例外キューに不要の場合、このプロパティを false に設定するとパフォーマンスが向上する可能性があります。 次の場合にはこのプロパティは適用されません。 ■ UseExceptionQueue が false の場合。 ■ 元のメッセージにメッセージ・ボディが含まれない場合、例外キューに送信されるメッセージにもメッセージ・ボディは含まれません。 ■ なんらかの理由で元のメッセージのコピーを作成できない場合、かわりに元のメッセージが例外キューに送信されることがあります。この場合、メッセージ・ボディが例外キューに送信される可能性があります。

表 B-2 オプションの J2CA @ActivationConfigProperty 属性（続き）

プロパティ名	値
`ListenerThreadMaxIdleDuration`	メッセージを受信していないリスナー・スレッドを OC4J で維持するミリ秒数。エンドポイントがアクティブである場合、少なくとも 1 つのリスナー・スレッドは維持されます。 デフォルトは 300000 ミリ秒です。
`ListenerThreadMaxPollInterval`	Oracle JMS コネクタの適用可能なポーリング間隔アルゴリズムのポーリング間隔に対する上限値（ミリ秒単位）。 Oracle JMS コネクタでは、適用可能なアルゴリズムを使用して実際のポーリング間隔を決定します。アクティブな期間中は、短いポーリング間隔（高頻度のポーリング）を使用し、非アクティブな期間中は、このプロパティを超えない範囲で長いポーリング間隔（低頻度のポーリング）を使用します。 リスナー・スレッドは、処理を待機中のメッセージが存在するかどうかを確認するためにポーリングを行います。このポーリングの実行頻度がより短くなると、所定のリスナー・スレッドは新規メッセージに（平均して）より高速に応答できます。ただし、高頻度のポーリングには、オーバーヘッドが伴います。リソース・プロバイダでは、ポーリングのたびに受信リクエストを処理する必要があります。 デフォルトは 5000 ミリ秒です。
`ListenerThreadMinBusyDuration`	過去の `ListenerThreadMinBusyDuration` ミリ秒間の任意の時点でリスナー・スレッドがメッセージを受信してアイドル状態（新規メッセージの到着を待機する状態）ではなくなり、このエンドポイントに対応するリスナー・スレッドの現在の数が `ReceiverThreads` 未満の場合、OC4J は可能であれば追加のリスナー・スレッドを作成します。 デフォルトは 10000 ミリ秒です。
`LogLevel`	Oracle JMS コネクタのログ・メッセージの詳細レベルを決定します。主に Oracle JMS コネクタ自体のデバッグを目的としていますが、これらのメッセージは、デバッグ問題が Oracle JMS コネクタの使用方法に関連している場合にも役立つ可能性があります。このプロパティは、デバッグ目的で一時的に設定することをお薦めします。本番用のコードには設定しないでください。（特定のログ・メッセージとログ・レベルは、Oracle JMS コネクタの将来のバージョンで追加、削除または変更される可能性があります。） 有効な値は次のとおりです。 ■ `ConnectionPool`: 接続プール関連のメッセージのみを記録します。 ■ `ConnectionOps`: 接続関連のメッセージのみを記録します。 ■ `TransactionalOps`: トランザクション関連のメッセージのみを記録します。 ■ `ListenerThreads`: リスナー・スレッド関連のメッセージのみを記録します。 ■ `INFO`: ユーザー名を含め、各サーバー・セッションのログインおよびログアウトを記録します。セッションの取得後に、詳細な情報が記録されます。 ■ `CONFIG`: ログイン、JDBC 接続およびデータベース情報のみを記録します。 ■ `FINE`: SQL を記録します。 ■ `FINER`: WARNING と同様です。スタック・トレースが含まれます。 ■ `FINEST`: 低レベルな追加情報が含まれます。 ■ `SEVERE`: TopLink が続行不可能であることを示す例外と、ログイン中に生成されるすべての例外を記録します。これには、スタック・トレースが含まれます。 ■ `WARNING`: SEVERE レベルで記録されないすべての例外を含め、TopLink を停止させることのない例外を記録します。これには、スタック・トレースは含まれません。 ■ `OFF`: ロギングは無効です。

表 B-2 オプションの J2CA @ActivationConfigProperty 属性（続き）

プロパティ名	値
MaxDeliveryCnt	リスナー・スレッドがメッセージドリブン Bean へのメッセージ配信を試行する最大回数。0（ゼロ）の値を指定すると、メッセージは破棄されません。 MaxDeliveryCnt を超える値が設定された JMSXDeliveryCount プロパティがメッセージに含まれる場合、そのメッセージは破棄されます。つまり、リスナー・スレッドは、メッセージドリブン Bean のメッセージ・リスナー・メソッド（JMS メッセージ・リスナーの onMessage メソッドなど）をコールしません。 例外キューが有効の場合（「UseExceptionQueue」を参照）、メッセージのコピーが例外キューに送信されます。 0（ゼロ）の値を使用する場合は注意してください。メッセージを破棄しないように MaxDeliveryCnt を 0 に設定した状態で、メッセージドリブン Bean が特定のメッセージに常に例外をスローして応答する場合、同じメッセージが何度も再配信されるためメッセージドリブン Bean はその処理を無制限に失敗し続ける可能性があります。 デフォルト値は 5 です。
MessageSelector	MDB が受信する必要のあるメッセージのタイプに一致するメッセージ・プロパティの String セレクタ式。式に一致しないメッセージはフィルタされます（MDB に配信されません）。 これは、リスナー・スレッドで作成された JMS セッション用の messageSelector として使用されます。 デフォルトはフィルタなしです。
ReceiverThreads	このエンドポイントに作成するリスナー・スレッドの最大数。 キューの場合、2 つ以上のスレッドを使用すると、メッセージをコンシュームする速度が向上する可能性があります。 トピックの場合、この値は常に 1 に設定する必要があります。 各リスナー・スレッドは、独自のセッションとトピック・サブスクライバを取得します。永続サブスクライバの場合、同じサブスクリプション名で複数のサブスクライバを保持するとエラーが発生します。非永続サブスクライバの場合、複数のスレッドを保持するサブスクライバがあっても効果はありません。スレッドが増加するとサブスクライバが増加し、各メッセージのコピーも増加するためです。 「ListenerThreadMinBusyDuration」も参照してください。 デフォルトは 1 です。
ResPassword	OC4J がリソース・プロバイダに渡す String パスワード。ResPassword プロパティでは、標準のパスワード・インダイレクション・オプションがサポートされます（たとえば、->joeuser を使用して joeuser のパスワードを表すことができます）。 設定すると、その値は OC4J によりパスワード引数として create*Connection メソッドに渡されます。 ResUser または ResPassword のいずれか 1 つのみが設定されている場合、OC4J では未設定のプロパティに対して NULL が渡されます。 ResUser も ResPassword も設定されていない場合、この MDB のインバウンド・メッセージ処理と例外キュー処理（「UseExceptionQueue」を参照）に使用される接続が、引数なしの create*Connection メソッドを使用して作成されます。 デフォルトは null です。
ResUser	OC4J がリソース・プロバイダに渡す String ユーザー名。 設定すると、その値は OC4J により user 引数として create*Connection メソッドに渡されます。 ResUser または ResPassword のいずれか 1 つのみが設定されている場合、OC4J では未設定のプロパティに対して NULL が渡されます。 ResUser も ResPassword も設定されていない場合、この MDB のインバウンド・メッセージ処理と例外キュー処理（「UseExceptionQueue」を参照）に使用される接続が、引数なしの create*Connection メソッドを使用して作成されます。 デフォルトは null です。

表 B-2 オプションの J2CA @ActivationConfigProperty 属性（続き）

プロパティ名	値
SubscriptionDurability	リスナー・スレッドに使用されるトピック・コンシューマの永続性を決定します。これは、トピックにのみ適用されます（キューにはこのプロパティを設定しません）。 有効な値は次のとおりです。 ■ Durable: メッセージは、OC4J が稼働していない場合でも失われません。信頼性の高いアプリケーションでは、通常、非永続的なトピック・サブスクリプションではなく永続的なトピック・サブスクリプションを使用します。このプロパティを Durable に設定する場合（および DestinationType が javax.jms.Topic または javax.jms.Destination の場合）、SubscriptionName プロパティが必要です。 ■ NonDurable（デフォルト）: OC4J でメッセージドリブン Bean がメッセージに対応することが保証されるのは、OC4J が稼働中である場合にかぎります。OC4J が一定の期間停止すると、メッセージは失われる可能性があります。
SubscriptionName	永続サブスクライバの作成時にリスナー・スレッドが使用する String 名。特定の JMS サーバーでは、（1 つ以下のリスナー・スレッドを保持する必要のある）最大 1 つの MDB に特定のサブスクリプション名を割り当てる必要があります。 SubscriptionDurability が Durable の場合（および DestinationType が javax.jms.Topic または javax.jms.Destination の場合）、このプロパティは必須です。他のすべての場合、このプロパティは無視されます。
TransactionTimeout	現在のトランザクションを終了する前に、Oracle JMS コネクタがメッセージの到着を待機する上限値（ミリ秒単位）。 OC4J トランザクション・マネージャは、トランザクションの継続時間を制限します（transaction-manager.xml の transaction-timeout を参照）。特に問題がないかぎり、メッセージドリブン Bean のメッセージ・リスナー・メソッド（JMS メッセージ・リスナーの onMessage メソッドなど）へのコール中は、トランザクション・マネージャがトランザクションのタイムアウトを行わないようこのプロパティを設定します。たとえば、トランザクション・マネージャのタイムアウトが 30 秒に設定され、通常の状態で onMessage ルーチンが 10 秒を超えない場合、TransactionTimeout を 20 秒（20000 ミリ秒）に設定できます。 デフォルトは 300000 ミリ秒です。

表 B-2 オプションの J2CA @ActivationConfigProperty 属性（続き）

プロパティ名	値
UseExceptionQueue	メッセージドリブン Bean のメッセージ・リスナー・メソッド（JMS メッセージ・リスナーの onMessage メソッドなど）が例外をスローしたため、またはリスナー・スレッドが MaxDeliveryCnt しきい値を超えたために配信できなくなったメッセージを OC4J で処理する方法を決定します。 有効な値は次のとおりです。 ■ true: OC4J は、後で説明するとおり、配信不能メッセージを例外キューに送信します。 この場合、ExceptionQueueName プロパティが必要です。 ■ false（デフォルト）: OC4J は、配信不能メッセージを破棄します。 デフォルトは false です。 UseExceptionQueue が true に設定されている場合、OC4J は配信不能メッセージを次の手順で例外キューに送信します。 1. 同じタイプの新規メッセージを作成します。 2. 元のメッセージから新規メッセージにプロパティとボディをコピーします。 3. getJMS{Header} を通じて取得された各ヘッダーを GJRA_CopyOfJMS{Header} プロパティに割り当てることで、元のヘッダーをコピーのプロパティに変換します。javax.jms.Destination は無効なプロパティ・タイプのため、宛先ヘッダーは説明的なメッセージに変換されます。 このサービスは、JMSX* プロパティ（特に JMSXDeliveryCount プロパティ）には提供されません。 ヘッダーがコピーされると、例外キューへのメッセージ送信によりそのほとんどが失われる（リソース・プロバイダに上書きされる）ため、OC4J はこの方法でヘッダーを変換します。 4. メッセージが配信されなかった理由を示す GJRA_DeliveryFailureReason という文字列プロパティを追加します。 5. 配信障害の直前にメッセージドリブン Bean のメッセージ・リスナー・メソッドが例外を生成した場合、例外情報を含む GJRA_onMessageExceptions という文字列プロパティを追加します。 6. 新規メッセージを次のように検証します。 コピーおよび拡張プロセスに成功した場合、値 true を含む GJRA_CopySuccessful という boolean プロパティが追加されます。 コピーまたは拡張プロセスの一部に失敗した場合でも、OC4J は停止しません。残りの手順を完了するよう試みます。Bytes、Map および Stream メッセージ・タイプでは、ボディの一部がコピーされ、残りはコピーされない可能性があります。この場合、値 false を含む GJRA_CopySuccessful という boolean プロパティが追加されます。 7. ConnectionFactoryJndiName プロパティで指定されたコネクション・ファクトリを使用して、生成されたメッセージを例外キューに送信します。 例外キューへのメッセージの送信は、1 回のみ試行されます。この試行に失敗すると、メッセージは例外キューに配置されることなく破棄されます。 OC4J では、ConnectionFactoryJndiName プロパティで指定されたコネクション・ファクトリを（第 1 宛先以外にも）使用して例外キューにメッセージを送信するため、（DestinationName プロパティで指定された）第 1 宛先がトピックの場合、コネクション・ファクトリはキューとトピックの両方に対応している必要があります（つまり、コネクション・ファクトリは javax.jms.ConnectionFactory または javax.jms.XAConnectionFactory である必要があります）。 これらの手順に適用できるバリエーションは、「IncludeBodiesInExceptionQueue」を参照してください。

用語集

この用語集では、このマニュアルで頻繁に使用される用語を定義する。追加の Java EE 用語は、`http://java.sun.com/javaee/reference/glossary/index.jsp` を参照。

EntityManager

永続性コンテキストへのアクセスに使用するインタフェース。エンティティ・マネージャを使用して、**エンティティ**・インスタンスを作成、参照、更新および削除する。

J2CA

J2EE Connector Architecture: リソース・アダプタの JMS プロバイダ・クライアント・ライブラリをラップすることで **JMS** プロバイダを J2EE アプリケーション・サーバーに統合するための標準的な方法。

JMS

Java Message Service。

JPA

EJB 3.0 Java 永続性 API。

OEMS

Oracle Enterprise Messaging Service: OC4J にサポートされる一連の **JMS** プロバイダ。OEMS JMS コネクタ（**J2CA** ベース・プロバイダ）、OEMS JMS（メモリー内またはファイルベース・プロバイダ）および OEMS JMS データベース（**Oracle AQ** ベース・プロバイダ）を含む。

Oracle AQ

AQ は、Oracle Streams 情報統合インフラストラクチャに構築された固有のデータベース統合メッセージ・キューイング機能。この機能により、多様なアプリケーションがメッセージを通じて非同期に通信できる。データベースとの統合により、セキュリティ、スケジューリングおよびメッセージ・メタデータ分析のための監査、追跡、メッセージ永続化など、独自のメッセージ管理機能が提供される。

AQ には、PL/SQL、Java（`oracle.AQ` パッケージを使用）、Java Message Service（JMS）を使用するか、または HTTP、HTTPS、SMTP などの転送プロトコルを使用してインターネット経由でアクセスできる。インターネット・アクセスの場合、クライアント（ユーザーまたはインターネット・アプリケーション）と Oracle サーバーが構造化 XML メッセージを交換する。

AQ により、エンタープライズ・アプリケーション統合に役立つ変換が提供され、メッセージング・ゲートウェイにより OracleAQ キューとの間でメッセージが自動的に伝播される。

詳細は、`http://otn.oracle.com/products/aq/index.html` を参照。

POJI

Plain Old Java Interface: ユーザーが定義するインタフェース（Java EE の指定によるインタフェースの拡張は不要）。EJB 3.0 では、ビジネス・インタフェースを POJI で指定できる。

POJO

Plain Old Java Object: ユーザーが定義する Java クラス（Java EE の指定によるクラスの拡張やインタフェースの実装は不要）。EJB 3.0 では、**エンティティ**を POJO で指定できる。

RA

リソース・アダプタ : 特に、**J2CA** に準拠するリソース・アダプタ。

アノテーション（Annotation）

対応する Java クラス・ファイルにコンパイルされるメタデータで Java ソース・コードを修飾するための単純で明示的な方法。これらのメタデータは、JPA の動作を管理するために **JPA 永続性プロバイダ**により実行時に解釈される。アノテーションにより、Java EE 5 アプリケーション・サーバーの内部と Java Standard Edition（Java SE）5 アプリケーションの EJB コンテナの外部で動作する標準的で移植可能な方法に基づいて、Java オブジェクトをリレーショナル・データベース表にマッピングする方法を宣言的に定義できる。JPA アノテーションは、`javax.persistence` パッケージで指定される。

永続性コンテキスト（Persistence Context）

任意の永続性エンティティの識別情報に対して一意のエンティティ・インスタンスが存在する**エンティティ**・インスタンスのセット。永続性コンテキストは、**エンティティ・マネージャ**・インスタンスに関連付けられる。エンティティ・マネージャは、この永続性コンテキスト内でエンティティ・インスタンスとそのライフ・サイクルを管理する。

永続性プロバイダ（Persistence Provider）

EJB 3.0 Java 永続性 API 実装。

永続性ユニット（Persistence Unit）

エンティティ・マネージャ・インスタンスが管理するエンティティのセット。永続性ユニットを使用して、アプリケーションにより関連付けるすべてのクラスのセットと、単一のデータベースにマッピングする対象を定義する。

エンティティ（Entity）

（Java EE EJB コンテナの内部または Java SE アプリケーションの EJB コンテナの外部にある）JPA **永続性プロバイダ**から取得される **JPA エンティティ・マネージャ**のサービスを通じてリレーショナル・データベースに永続化される非一時的フィールドを含む Java オブジェクト。JPA を使用すると、`@Entity` **アノテーション**を適用することで任意の **POJO** をエンティティとして指定できる。

索引

記号

数字

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

あ

い

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

て

と

な

ね

は

ひ

ふ

へ

ほ

ま

め

ゆ

よ

ら

り

る

れ

ろ