HITACHI
Inspire the Next

Cosminexus V9 アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)

解説書

3020-3-Y07-60

■ 対象製品

マニュアル「アプリケーションサーバ & BPM/ESB 基盤 概説」の前書きの対象製品の説明を参照してください。

■ 輸出時の注意

本製品を輸出される場合には、外国為替及び外国貿易法の規制並びに米国輸出管理規則など外国の輸出関連法規をご確認の上、必要な手続きをお取りください。
なお、不明な場合は、弊社担当営業にお問い合わせください。

■ 商標類

■ 発行

2015 年 4 月 3020-3-Y07-60

■ 著作権

変更内容

変更内容(3020-3-Y07-60) uCosminexus Application Server 09-70，uCosminexus Application Server(64) 09-70，uCosminexus Client 09-70，uCosminexus Developer 09-70，uCosminexus Service Architect 09-70，uCosminexus Service Platform 09-70，uCosminexus Service Platform(64) 09-70

追加・変更内容	変更個所
記載内容は変更なし（リンク情報だけを変更した）。	－

uCosminexus Application Server 09-60，uCosminexus Application Server(64) 09-60，uCosminexus Client 09-60，uCosminexus Developer 09-60，uCosminexus Service Architect 09-60，uCosminexus Service Platform 09-60，uCosminexus Service Platform(64) 09-60

追加・変更内容	変更個所
記載内容は変更なし（リンク情報だけを変更した）。	－

単なる誤字・脱字などはお断りなく訂正しました。

はじめに

このマニュアルをお読みになる際の前提情報については，マニュアル「アプリケーションサーバ & BPM/ESB 基盤 概説」のはじめにの説明を参照してください。

目次

1 アプリケーションサーバの機能

この章では，アプリケーションサーバの機能の分類と目的，および機能とマニュアルの対応について説明します。また, このバージョンで変更した機能についても説明しています。

1.1　機能の分類

アプリケーションサーバは，Java EE 6 に準拠した J2EE サーバを中心としたアプリケーションの実行環境を構築したり，実行環境上で動作するアプリケーションを開発したりするための製品です。Java EE の標準仕様に準拠した機能や，アプリケーションサーバで独自に拡張された機能など，多様な機能を使用できます。目的や用途に応じた機能を選択して使用することで，信頼性の高いシステムや，処理性能に優れたシステムを構築・運用できます。

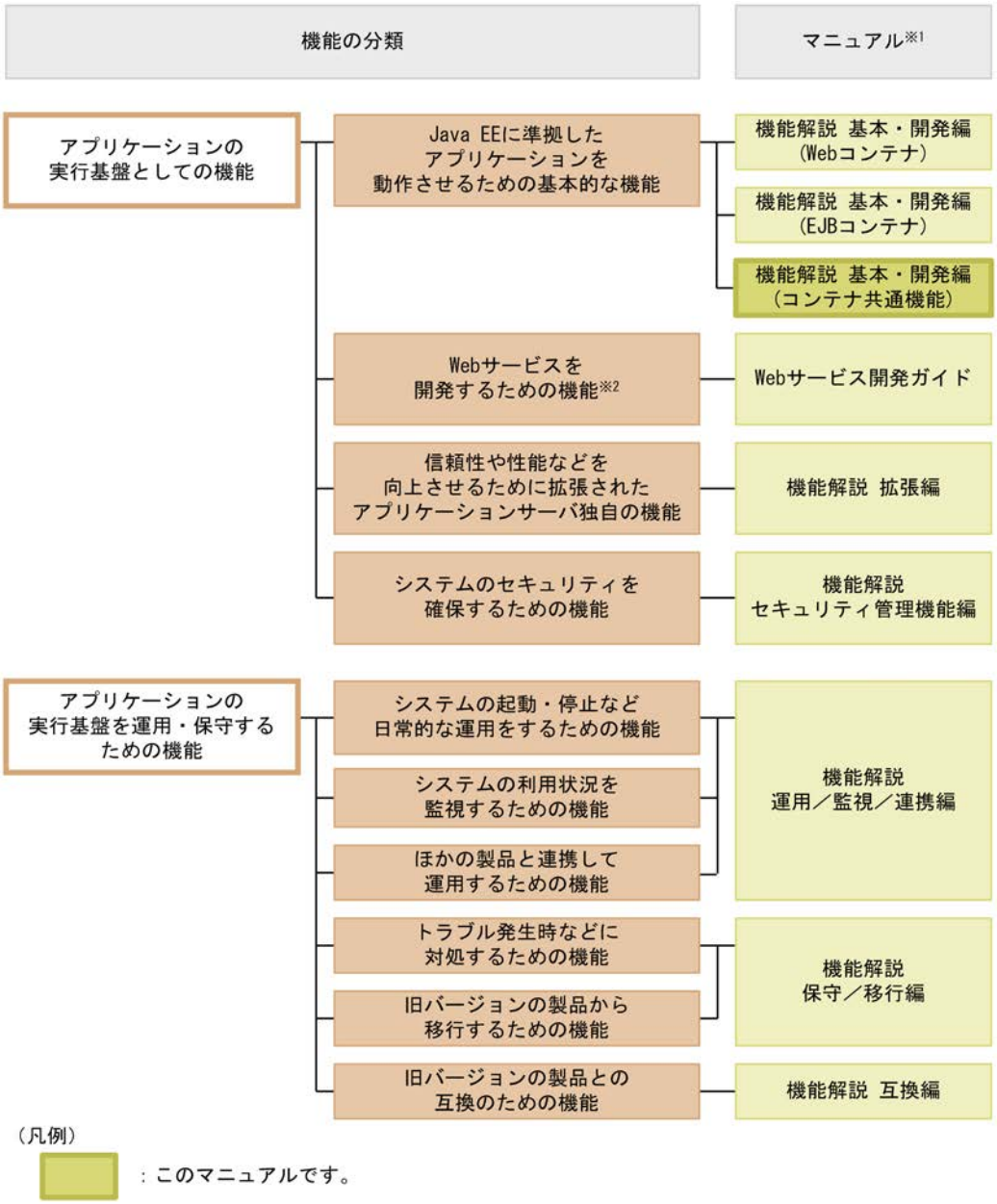
アプリケーションサーバの機能は，大きく分けて，次の二つに分類できます。

- アプリケーションの実行基盤としての機能
- アプリケーションの実行基盤を運用・保守するための機能

二つの分類は，機能の位置づけや用途によって，さらに詳細に分類できます。アプリケーションサーバのマニュアルは，機能の分類に合わせて提供しています。

アプリケーションサーバの機能の分類と対応するマニュアルについて，次の図に示します。

図 1-1　アプリケーションサーバの機能の分類と対応するマニュアル



注※1

マニュアル名称の「アプリケーションサーバ」を省略しています。

注※2

アプリケーションサーバでは，SOAP Web サービスと RESTful Web サービスを実行できます。目的によっては，マニュアル「アプリケーションサーバ Web サービス開発ガイド」以外の次のマニュアルも参照してください。

SOAP アプリケーションを開発・実行する場合

- アプリケーションサーバ SOAP アプリケーション開発の手引

SOAP Web サービスや SOAP アプリケーションのセキュリティを確保する場合

- XML Security - Core ユーザーズガイド

- アプリケーションサーバ Web サービスセキュリティ構築ガイド

XML の処理について詳細を知りたい場合

- XML Processor ユーザーズガイド

ここでは，機能の分類について，マニュアルとの対応と併せて説明します。

1.1.1　アプリケーションの実行基盤としての機能

アプリケーションとして実装されたオンライン業務やバッチ業務を実行する基盤となる機能です。システムの用途や求められる要件に応じて，使用する機能を選択します。

アプリケーションの実行基盤としての機能を使用するかどうかは，システム構築やアプリケーション開発よりも前に検討する必要があります。

アプリケーションの実行基盤としての機能について，分類ごとに説明します。

(1)　アプリケーションを動作させるための基本的な機能（基本・開発機能）

アプリケーション（J2EE アプリケーション）を動作させるための基本的な機能が該当します。主に J2EE サーバの機能が該当します。

アプリケーションサーバでは，Java EE 6 に準拠した J2EE サーバを提供しています。J2EE サーバでは，標準仕様に準拠した機能のほか，アプリケーションサーバ独自の機能も提供しています。

基本・開発機能は，機能を使用する J2EE アプリケーションの形態に応じて，さらに三つに分類できます。アプリケーションサーバの機能解説のマニュアルは，この分類に応じて分冊されています。

それぞれの分類の概要を説明します。

- **Web アプリケーションを実行するための機能（Web コンテナ）**
  Web アプリケーションの実行基盤である Web コンテナの機能，および Web コンテナと Web サーバが連携して実現する機能が該当します。
- **Enterprise Bean を実行するための機能（EJB コンテナ）**
  Enterprise Bean の実行基盤である EJB コンテナの機能が該当します。また，Enterprise Bean を呼び出す EJB クライアントの機能も該当します。
- **Web アプリケーションと Enterprise Bean の両方で使用する機能（コンテナ共通機能）**
  Web コンテナ上で動作する Web アプリケーションおよび EJB コンテナ上で動作する Enterprise Bean の両方で使用できる機能が該当します。

(2)　Web サービスを開発するための機能

Web サービスの実行環境および開発環境としての機能が該当します。

アプリケーションサーバでは，次のエンジンを提供しています。

- JAX-WS 仕様に従った SOAP メッセージのバインディングを実現する JAX-WS エンジン
- JAX-RS 仕様に従った RESTful HTTP メッセージのバインディングを実現する JAX-RS エンジン

(3)　信頼性や性能などを向上させるために拡張されたアプリケーションサーバ独自の機能（拡張機能）

アプリケーションサーバで独自に拡張された機能が該当します。バッチサーバ，CTM，データベースなど，J2EE サーバ以外のプロセスを使用して実現する機能も含まれます。

アプリケーションサーバでは，システムの信頼性を高め，安定稼働を実現するための多様な機能が拡張されています。また，J2EE アプリケーション以外のアプリケーション（バッチアプリケーション）を Java の環境で動作させる機能も拡張しています。

(4)　システムのセキュリティを確保するための機能（セキュリティ管理機能）

アプリケーションサーバを中心としたシステムのセキュリティを確保するための機能が該当します。不正なユーザからのアクセスを防止するための認証機能や，通信路での情報漏えいを防止するための暗号化機能などがあります。

1.1.2　アプリケーションの実行基盤を運用・保守するための機能

アプリケーションの実行基盤を効率良く運用したり，保守したりするための機能です。システムの運用開始後に，必要に応じて使用します。ただし，機能によっては，あらかじめ設定やアプリケーションの実装が必要なものがあります。

アプリケーションの実行基盤を運用・保守するための機能について，分類ごとに説明します。

(1)　システムの起動・停止など日常的な運用をするための機能（運用機能）

システムの起動や停止，アプリケーションの開始や停止，入れ替えなどの，日常運用で使用する機能が該当します。

(2)　システムの利用状況を監視するための機能（監視機能）

システムの稼働状態や，リソースの枯渇状態などを監視するための機能が該当します。また，システムの操作履歴など，監査で使用する情報を出力する機能も該当します。

(3)　ほかの製品と連携して運用するための機能（連携機能）

JP1 やクラスタソフトウェアなど，ほかの製品と連携して実現する機能が該当します。

(4)　トラブル発生時などに対処するための機能（保守機能）

トラブルシューティングのための機能が該当します。トラブルシューティング時に参照する情報を出力するための機能も含みます。

(5)　旧バージョンの製品から移行するための機能（移行機能）

旧バージョンのアプリケーションサーバから新しいバージョンのアプリケーションサーバに移行するための機能が該当します。

(6)　旧バージョンの製品との互換のための機能（互換機能）

旧バージョンのアプリケーションサーバとの互換用の機能が該当します。なお，互換機能については，対応する推奨機能に移行することをお勧めします。

1.1.3　機能とマニュアルの対応

アプリケーションサーバの機能解説のマニュアルは，機能の分類に合わせて分冊されています。

機能の分類と，それぞれの機能について説明しているマニュアルとの対応を次の表に示します。

表 1-1　機能の分類と機能解説のマニュアルの対応

分類	機能	マニュアル※1
基本・開発機能	Web コンテナ	基本・開発編(Web コンテナ)
	JSF および JSTL の利用	
	Web サーバ連携	
	インプロセス HTTP サーバ	
	サーブレットおよび JSP の実装	
	EJB コンテナ	基本・開発編(EJB コンテナ)
	EJB クライアント	
	Enterprise Bean 実装時の注意事項	
	ネーミング管理	基本・開発編(コンテナ共通機能)※2
	リソース接続とトランザクション管理	
	OpenTP1 からのアプリケーションサーバの呼び出し（TP1 インバウンド連携機能）	
	アプリケーションサーバでの JPA の利用	
	CJPA プロバイダ	
	CJMS プロバイダ	
	JavaMail の利用	
	アプリケーションサーバでの CDI の利用	
	アプリケーションサーバでの Bean Validation の利用	
	アプリケーションの属性管理	
	アノテーションの使用	
	J2EE アプリケーションの形式とデプロイ	
	コンテナ拡張ライブラリ	
拡張機能	バッチサーバによるアプリケーションの実行	拡張編
	CTM によるリクエストのスケジューリングと負荷分散	
	バッチアプリケーションのスケジューリング	
	J2EE サーバ間のセッション情報の引き継ぎ（セッションフェイルオーバ機能）	
	データベースセッションフェイルオーバ機能	

分類	機能	マニュアル※1
拡張機能	EADs セッションフェイルオーバ機能	拡張編
	明示管理ヒープ機能を使用した FullGC の抑止	
	アプリケーションのユーザログ出力	
	スレッドの非同期並行処理	
セキュリティ管理機能	統合ユーザ管理による認証	セキュリティ管理機能編
	アプリケーションの設定による認証	
	SSL/TLS 通信での TLSv1.2 の使用	
	API による直接接続を使用する負荷分散機の運用管理機能からの制御	
運用機能	システムの起動と停止	運用／監視／連携編
	J2EE アプリケーションの運用	
監視機能	稼働情報の監視（稼働情報収集機能）	
	リソースの枯渇監視	
	監査ログ出力機能	
	データベース監査証跡連携機能	
	運用管理コマンドによる稼働情報の出力	
	Management イベントの通知と Management アクションによる処理の自動実行	
	CTM の稼働統計情報の収集	
	コンソールログの出力	
連携機能	JP1 と連携したシステムの運用	
	システムの集中監視（JP1/IM との連携）	
	ジョブによるシステムの自動運転（JP1/AJS との連携）	
	監査ログの収集および一元管理（JP1/Audit Management - Manager との連携）	
	クラスタソフトウェアとの連携	
	1:1 系切り替えシステム（クラスタソフトウェアとの連携）	
	相互系切り替えシステム（クラスタソフトウェアとの連携）	
	N:1 リカバリシステム（クラスタソフトウェアとの連携）	
	ホスト単位管理モデルを対象にした系切り替えシステム（クラスタソフトウェアとの連携）	
保守機能	トラブルシューティング関連機能	保守／移行編
	性能解析トレースを使用した性能解析	

分類	機能	マニュアル※1
保守機能	日立固有の製品の JavaVM（以降，JavaVM と略す場合があります）の機能	保守／移行編
移行機能	旧バージョンのアプリケーションサーバからの移行	
	推奨機能への移行	
互換機能	ベーシックモード	互換編
	サーブレットエンジンモード	
	基本・開発機能の互換機能	
	拡張機能の互換機能	

注※1　マニュアル名称の「アプリケーションサーバ 機能解説」を省略しています。

注※2　このマニュアルです。

1.2　システムの目的と機能の対応

アプリケーションサーバでは，構築・運用するシステムの目的に合わせて，適用する機能を選択する必要があります。

この節では，Web コンテナおよび EJB コンテナが共通で使用できる機能について，どのようなシステムの場合に使用するとよいかを示します。機能ごとに，次の項目への対応を示しています。

- **信頼性**
  高い信頼が求められるシステムの場合に使用するとよい機能です。
  アベイラビリティ（安定稼働性）およびフォールトトレランス（耐障害性）を高める機能や，ユーザ認証などのセキュリティを高めるための機能が該当します。
- **性能**
  性能を重視したシステムの場合に使用するとよい機能です。
  システムのパフォーマンスチューニングで使用する機能などが該当します。
- **運用・保守**
  効率の良い運用・保守をしたい場合に使用するとよい機能です。
- **拡張性**
  システム規模の拡大・縮小および構成の変更への柔軟な対応が必要な場合に使用するとよい機能です。
- **そのほか**
  そのほかの個別の目的に対応するための機能です。

また，Web コンテナおよび EJB コンテナが共通で使用できる機能には，Java EE 標準機能とアプリケーションサーバが独自に拡張した機能があります。機能を選択するときには，必要に応じて，Java EE 標準への準拠についても確認してください。

1.2.1　ネーミング管理の機能

ネーミング管理の機能を次の表に示します。システムの目的に合った機能を選択してください。機能の詳細については，参照先を確認してください。

表 1-2　ネーミング管理の機能とシステムの目的の対応

機能	システムの目的					Java EE 標準への準拠		参照先
	信頼性	性能	運用・保守	拡張性	そのほか	標準	拡張	
JNDI 名前空間へのオブジェクトのバインドとルックアップ	−	−	−	○	−	○	○	2.3
Portable Global JNDI 名でのルックアップ	−	−	−	○	−	○	○	2.4
HITACHI_EJB から始まる名称でのルックアップ	−	−	−	○	−	○	○	2.5

機能	システムの目的					Java EE 標準への準拠		参照先
	信頼性	性能	運用・保守	拡張性	そのほか	標準	拡張	
Enterprise Bean または J2EE リソースへの別名付与（ユーザ指定名前空間機能）	－	－	－	○	－	－	○	2.6
ラウンドロビンポリシーによる CORBA ネーミングサービスの検索	－	－	－	○	－	－	○	2.7
ネーミング管理機能でのキャッシング	－	○	－	－	－	－	○	2.8
ネーミングサービスの障害検知	○	－	－	－	－	－	○	2.9
CORBA ネーミングサービスの切り替え	－	－	－	○	－	－	○	2.10
EJB ホームオブジェクトリファレンスの再利用（EJB ホームオブジェクトへの再接続機能）	－	－	－	○	－	○	○	2.11

（凡例）　○：対応する　　－：対応しない

注　「Java EE 標準への準拠」の「標準」と「拡張」の両方に○が付いている機能は，Java EE 標準の機能にアプリケーションサーバ独自の機能が拡張されていることを示します。「拡張」だけに○が付いている機能はアプリケーションサーバ独自の機能であることを示します。

1.2.2　リソース接続とトランザクション管理の機能

リソース接続とトランザクション管理の機能を次の表に示します。システムの目的に合った機能を選択してください。機能の詳細については，参照先を確認してください。

表 1-3　リソース接続とトランザクション管理の機能とシステムの目的の対応

機能名	システムの目的					Java EE 標準への準拠		参照先
	信頼性	性能	運用・保守	拡張性	そのほか	標準	拡張	
コネクションプーリング	－	○	－	－	－	○	○	3.14.1
コネクションプーリングで使用できる機能	－	○	－	－	－	－	○	3.14.2
コネクションシェアリング・	－	○	－	－	－	○	－	3.14.3

機能名	システムの目的					Java EE 標準への準拠		参照先
	信頼性	性能	運用・保守	拡張性	そのほか	標準	拡張	
アソシエーション	−	○	−	−	−	○	−	3.14.3
ステートメントプーリング	−	○	−	−	−	−	○	3.14.4
ライトトランザクション	−	○	−	−	−	−	○	3.14.5
インプロセストランザクションサービス	−	○	−	−	−	○	−	3.14.6
DataSource オブジェクトのキャッシング	−	○	−	−	−	−	○	3.14.7
DB Connector のコンテナ管理でのサインオンの最適化	−	○	−	−	−	−	○	3.14.8
コネクションの障害検知	○	−	−	−	−	○※1	○※1	3.15.1
コネクション枯渇時のコネクション取得待ち	○	−	−	−	−	−	○	3.15.2
コネクションの取得リトライ	○	−	−	−	−	−	○	3.15.3
コネクションプールの情報表示	○	−	−	−	−	−	○	3.15.4
コネクションプールのクリア	○	−	−	−	−	−	○	3.15.5
コネクションの自動クローズ	○	−	−	−	−	○	−	3.15.6
コネクションスイーパ	○	−	−	−	−	−	○	3.15.7
トランザクションタイムアウトとステートメントキャンセル	○	−	−	−	−	○	−	3.15.8
トランザクションリカバリ	○	−	−	−	−	○	−	3.15.9

機能名	システムの目的					Java EE 標準への準拠		参照先
	信頼性	性能	運用・保守	拡張性	そのほか	標準	拡張	
障害調査用SQL の出力	−	−	○	−	−	−	○	3.15.10
オブジェクトの自動クローズ	○	−	−	−	−	○	−	3.15.11
リソースアダプタのライフサイクル管理※2	−	−	−	−	−	○	−	3.16.1
リソースアダプタのワーク管理※2	−	−	−	−	−	○	−	3.16.2
メッセージインフロー※2	−	−	−	−	−	○	−	3.16.3
トランザクションインフロー※3	−	−	−	−	−	○	−	3.16.4
管理対象オブジェクトのルックアップ※2	−	−	−	−	−	○	−	3.16.5
コネクション定義の複数指定※2	−	−	−	−	−	○	−	3.16.6
クラスタコネクションプール機能(コネクションプールの一時停止・再開・状態表示)	○	−	−	−	−	−	○	3.17
リソースへの接続テスト	−	−	○	−	−	−	○	3.18
ファイアウォール環境での運用のための機能	−	−	○	−	−	−	○	3.19

（凡例）　○：対応する　　−：対応しない

注　「Java EE 標準への準拠」の「標準」と「拡張」の両方に○が付いている機能は，Java EE 標準の機能にアプリケーションサーバ独自の機能が拡張されていることを示します。「拡張」だけに○が付いている機能はアプリケーションサーバ独自の機能であることを示します。

注※1　Connector 1.0 の場合は「拡張」です。Connector 1.5 の場合は「Java EE 標準」です。

注※2　Connector 1.5 仕様に準拠したリソースアダプタを使用する場合だけ使用できる機能です。

注※3　Connector 1.5 仕様に準拠したリソースアダプタのうち，TP1 インバウンドアダプタを使用する場合だけ使用できる機能です。

1.2.3　OpenTP1 からのアプリケーションサーバの呼び出し（TP1 インバウンド連携機能）の機能

OpenTP1 からアプリケーションサーバを呼び出す際の TP1 インバウンド連携機能を次の表に示します。システムの目的に合った機能を選択してください。機能の詳細については，参照先を確認してください。

表 1-4　OpenTP1 からのアプリケーションサーバの呼び出し（TP1 インバウンド連携機能）とシステムの目的の対応

機能名	システムの目的					Java EE 標準への準拠		参照先
	信頼性	性能	運用・保守	拡張性	そのほか	標準	拡張	
コネクション管理機能	－	－	○	－	○	－	○	4.6
RPC 通信機能	－	－	○	－	○	－	○	4.7
スケジュール機能	－	－	○	－	○	－	○	4.8
トランザクション連携機能	－	－	○	－	○	－	○	4.9

（凡例）　○：対応する　　－：対応しない

注　「拡張」だけに○が付いている機能はアプリケーションサーバ独自の機能であることを示します。

1.2.4　アプリケーションサーバで使用する JPA の機能

アプリケーションサーバで使用する JPA の機能を次の表に示します。システムの目的に合った機能を選択してください。機能の詳細については，参照先を確認してください。

表 1-5　アプリケーションサーバで使用する JPA の機能とシステムの目的の対応

機能名	システムの目的					Java EE 標準への準拠		参照先
	信頼性	性能	運用・保守	拡張性	そのほか	標準	拡張	
JPA	○	－	○	－	－	○	○	5 章

（凡例）　○：対応する　　－：対応しない

注　「Java EE 標準への準拠」の「標準」と「拡張」の両方に○が付いている機能は，Java EE 標準の機能にアプリケーションサーバ独自の機能が拡張されていることを示します。

1.2.5　CJPA プロバイダの機能

CJPA プロバイダの機能を次の表に示します。システムの目的に合った機能を選択してください。機能の詳細については，参照先を確認してください。

表 1-6　CJPA プロバイダの機能とシステムの目的の対応

機能名	システムの目的					Java EE 標準への準拠		参照先
	信頼性	性能	運用・保守	拡張性	そのほか	標準	拡張	
エンティティを使用したデータベースの更新	○	○	−	−	−	○	○	6.3
EntityManager によるエンティティの操作	−	−	−	−	○	○	○	6.4
データベースと Java オブジェクトとのマッピング情報の定義	−	−	−	−	○	○	○	6.5
エンティティのリレーションシップ	−	○	−	−	−	○	○	6.6
エンティティオブジェクトのキャッシュ機能	−	○	−	−	−	−	○	6.7
プライマリキー値の自動採番	−	−	−	−	○	○	○	6.8
クエリ言語によるデータベース操作	−	−	○	−	−	○	○	6.9
楽観的ロック	○	−	−	−	−	○	○	6.10
JPQL での悲観的ロック	○	−	−	−	−	−	○	6.11

（凡例）　○：対応する　　−：対応しない

注　「Java EE 標準への準拠」の「標準」と「拡張」の両方に○が付いている機能は，Java EE 標準の機能にアプリケーションサーバ独自の機能が拡張されていることを示します。「拡張」だけに○が付いている機能はアプリケーションサーバ独自の機能であることを示します。

1.2.6　CJMS プロバイダの機能

CJMS プロバイダの機能を次の表に示します。システムの目的に合った機能を選択してください。機能の詳細については，参照先を確認してください。

表 1-7　CJMS プロバイダの機能とシステムの目的の対応

機能名	システムの目的					Java EE 標準への準拠		参照先
	信頼性	性能	運用・保守	拡張性	そのほか	標準	拡張	
JMS 仕様に準拠したメッセージの送受信	−	−	−	−	○	○	−	7.4 7.5 7.6

機能名	システムの目的					Java EE 標準への準拠		参照先
	信頼性	性能	運用・保守	拡張性	そのほか	標準	拡張	
メッセージ配信の高信頼性確保の仕組み	○	−	−	−	−	○	−	7.7
CJMSP ブローカーの機能	−	−	○	−	−	○	−	7.8
CJMSP リソースアダプタの機能	−	−	−	−	○	○	−	7.9
Message-driven Bean の呼び出し	−	−	−	−	○	○	−	7.10

（凡例）　○：対応する　　−：対応しない

1.2.7　JavaMail の機能

JavaMail の機能を次の表に示します。システムの目的に合った機能を選択してください。機能の詳細については，参照先を確認してください。

表 1–8　JavaMail の機能とシステムの目的の対応

機能名	システムの目的					Java EE 標準への準拠		参照先
	信頼性	性能	運用・保守	拡張性	そのほか	標準	拡張	
JavaMail	−	−	−	−	○	○	−	8 章

（凡例）　○：対応する　　−：対応しない

1.2.8　アプリケーションサーバで使用する CDI の機能

アプリケーションサーバで使用する CDI の機能を次の表に示します。システムの目的に合った機能を選択してください。機能の詳細については，参照先を確認してください。

表 1–9　CDI の機能とシステムの目的の対応

機能名	システムの目的					Java EE 標準への準拠		参照先
	信頼性	性能	運用・保守	拡張性	そのほか	標準	拡張	
CDI	−	−	−	−	○	○	−	9 章

（凡例）　○：対応する　　−：対応しない

1.2.9　アプリケーションサーバで使用する Bean Validation の機能

アプリケーションサーバで使用する Bean Validation の機能を次の表に示します。システムの目的に合った機能を選択してください。機能の詳細については，参照先を確認してください。

表 1-10　Bean Validation の機能とシステムの目的の対応

機能名	システムの目的					Java EE 標準への準拠		参照先
	信頼性	性能	運用・保守	拡張性	そのほか	標準	拡張	
Bean Validation	－	－	－	－	○	○	－	10 章

（凡例）　○：対応する　　－：対応しない

1.2.10　アプリケーションの属性管理

アプリケーションの属性管理の機能を次の表に示します。システムの目的に合った機能を選択してください。機能の詳細については，参照先を確認してください。

表 1-11　アプリケーションの属性管理の機能とシステムの目的の対応

機能名	システムの目的					Java EE 標準への準拠		参照先
	信頼性	性能	運用・保守	拡張性	そのほか	標準	拡張	
cosminexus.xml を含むアプリケーション	－	－	－	○	－	－	○	11.3
DD の省略	－	－	－	○	－	○	○	11.4

（凡例）　○：対応する　　－：対応しない

注　「Java EE 標準への準拠」の「標準」と「拡張」の両方に○が付いている機能は，Java EE 標準の機能にアプリケーションサーバ独自の機能が拡張されていることを示します。「拡張」だけに○が付いている機能はアプリケーションサーバ独自の機能であることを示します。

1.2.11　アノテーションの機能

アノテーションの機能を次の表に示します。システムの目的に合った機能を選択してください。機能の詳細については，参照先を確認してください。

表 1-12　アノテーションの機能とシステムの目的の対応

機能名	システムの目的					Java EE 標準への準拠		参照先
	信頼性	性能	運用・保守	拡張性	そのほか	標準	拡張	
アノテーション	－	－	－	○	－	○	○	12 章

（凡例）　○：対応する　　－：対応しない

注　「Java EE 標準への準拠」の「標準」と「拡張」の両方に○が付いている機能は，Java EE 標準の機能にアプリケーションサーバ独自の機能が拡張されていることを示します。

1.2.12　J2EE アプリケーションの形式とデプロイの機能

J2EE アプリケーションの形式とデプロイの機能を次の表に示します。システムの目的に合った機能を選択してください。機能の詳細については，参照先を確認してください。

表 1-13　J2EE アプリケーションの形式とデプロイの機能とシステムの目的の対応

機能名	システムの目的					Java EE 標準への準拠		参照先
	信頼性	性能	運用・保守	拡張性	そのほか	標準	拡張	
J2EE アプリケーションのデプロイとアンデプロイ	－	－	－	－	－	○	○	13.5
J2EE アプリケーションの入れ替え	－	－	－	－	－	○	－	13.6
J2EE アプリケーションのリデプロイ	－	－	○	－	－	－	○	13.7
J2EE アプリケーションの更新検知とリロード	－	－	○	－	－	－	○	13.8

（凡例）　○：対応する　　－：対応しない

注　「Java EE 標準への準拠」の「標準」と「拡張」の両方に○が付いている機能は，Java EE 標準の機能にアプリケーションサーバ独自の機能が拡張されていることを示します。「拡張」だけに○が付いている機能はアプリケーションサーバ独自の機能であることを示します。

1.2.13　コンテナ拡張ライブラリの機能

コンテナ拡張ライブラリの機能を次の表に示します。システムの目的に合った機能を選択してください。機能の詳細については，参照先を確認してください。

表 1-14　コンテナ拡張ライブラリの機能とシステムの目的の対応

機能名	システムの目的					Java EE 標準への準拠		参照先
	信頼性	性能	運用・保守	拡張性	そのほか	標準	拡張	
コンテナ拡張ライブラリ機能	－	－	－	○	－	－	○	14.3
サーバ起動・停止フック機能	－	－	－	○	－	－	○	14.4
スマートエージェント経由での CORBA オブジェクトの呼び出し	－	－	－	○	－	－	○	14.5

（凡例）　○：対応する　　－：対応しない

注　「拡張」だけに○が付いている機能はアプリケーションサーバ独自の機能であることを示します。

1.3　このマニュアルに記載している機能の説明

ここでは，このマニュアルで機能を説明するときの分類の意味と，分類を示す表の例について説明します。

1.3.1　分類の意味

このマニュアルでは，各機能について，次の五つに分類して説明しています。マニュアルを参照する目的によって，必要な個所を選択して読むことができます。

- 解説
  機能の解説です。機能の目的，特長，仕組みなどについて説明しています。機能の概要について知りたい場合にお読みください。
- 実装
  コーディングの方法や DD の記載方法などについて説明しています。アプリケーションを開発する場合にお読みください。
- 設定
  システム構築時に必要となるプロパティなどの設定方法について説明しています。システムを構築する場合にお読みください。
- 運用
  運用方法の説明です。運用時の手順や使用するコマンドの実行例などについて説明しています。システムを運用する場合にお読みください。
- 注意事項
  機能を使用するときの全般的な注意事項について説明しています。注意事項の説明は必ずお読みください。

1.3.2　分類を示す表の例

機能説明の分類については，表で説明しています。表のタイトルは，「この章の構成」または「この節の構成」となっています。

次に，機能説明の分類を示す表の例を示します。

機能説明の分類を示す表の例

表 X-1　この章の構成（○○機能）

分類	タイトル	参照先
解説	○○機能とは	X.1
実装	アプリケーションの実装	X.2
	DD および cosminexus.xml※での定義	X.3
設定	実行環境での設定	X.4
運用	○○機能を使用した運用	X.5
注意事項	○○機能使用時の注意事項	X.6

注※　cosminexus.xml については，「11.　アプリケーションの属性管理」を参照してください。

ポイント

cosminexus.xml を含まないアプリケーションのプロパティ設定

cosminexus.xml を含まないアプリケーションでは，実行環境へのインポート後にプロパティを設定，または変更します。設定済みのプロパティも実行環境で変更できます。

実行環境でのアプリケーションの設定は，サーバ管理コマンドおよび属性ファイルで実施します。サーバ管理コマンドおよび属性ファイルでのアプリケーションの設定については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「3.5.2　J2EE アプリケーションのプロパティの設定手順」を参照してください。

属性ファイルで指定するタグは，DD または cosminexus.xml と対応しています。DD または cosminexus.xml と属性ファイルのタグの対応については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「2.2　アプリケーション属性ファイル（cosminexus.xml）で指定する各属性の詳細」を参照してください。

なお，各属性ファイルで設定するプロパティは，アプリケーション統合属性ファイルでも設定できます。

1.4　アプリケーションサーバ 09-70 での主な機能変更

この節では，アプリケーションサーバ 09-70 での主な機能の変更について，変更目的ごとに説明します。

説明内容は次のとおりです。

- アプリケーションサーバ 09-70 で変更になった主な機能と，その概要を説明しています。機能の詳細については参照先の記述を確認してください。「参照先マニュアル」および「参照個所」には，その機能についての主な記載個所を記載しています。
- 「参照先マニュアル」に示したマニュアル名の「アプリケーションサーバ 」は省略しています。

(1)　標準機能・既存機能への対応

標準機能・既存機能への対応を目的として変更した項目を次の表に示します。

表 1-15　標準機能・既存機能への対応を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
運用管理ポータルでの JSP コンパイルバージョンの追加	J2EE サーバでの JSP から生成されたサーブレットのコンパイル方法に「JDK1.7 の仕様に従ったコンパイル」と「JDK7 の仕様に従ったコンパイル」を追加する。	運用管理ポータル操作ガイド	10.9.4
		リファレンス 定義編(サーバ定義)	4.14.1
JDK8 でのメタスペース対応	JavaVM の起動で使用している Permanent 領域用のオプションを Metaspace 領域用のオプションに変更する。	システム構築・運用ガイド	付録 A.2
		運用管理ポータル操作ガイド	10.8.3, 10.9.8
		リファレンス 定義編(サーバ定義)	5.2, 5.3, 10.4
統合ユーザ管理でのユーザ認証の SHA-2 対応	統合ユーザ管理でのユーザ認証のハッシュアルゴリズムとして SHA-224, SHA-256, SHA-384, SHA-512 を追加する。	機能解説 セキュリティ管理機能編	5.3.1, 5.3.9, 5.10.6, 11.4.3, 12.4.3, 12.5.3, 13.2, 14.3

(2)　運用性の維持・向上

運用性の維持・向上を目的として変更した項目を次の表に示します。

表 1-16　運用性の維持・向上を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
V9.7 へのバージョンアップ対応	バージョンアップ時の JavaVM の起動で使用している Permanent 領域用のオプションを Metaspace 領域用のオプションに変更する手順を追加する。	機能解説 保守／移行編	10.4.2, 10.4.3, 10.4.4, 10.4.5

(3)　そのほかの目的

そのほかの目的で変更した項目を次の表に示します。

表 1-17　そのほかの目的による変更

項目	変更の概要	参照先マニュアル	参照個所
snapshot ログの収集対象	snapshot ログの収集対象として JavaVM イベントログと Management Server のスレッドダンプを追加する。	機能解説 保守／移行編	付録 A.2

2 ネーミング管理

この章では，ネーミング管理の機能について説明します。ネーミング管理は，J2EE サービスで提供されている機能の一つです。ネーミング管理機能を使用して，Enterprise Bean 参照またはリソース参照のための名前解決を実現します。

2.1　この章の構成

ネーミング管理は，J2EE サービスで提供されている機能の一つです。J2EE サービスは，J2EE コンテナの部品機能として利用される機能です。

ネーミング管理の機能と参照先を次の表に示します。

表 2-1　ネーミング管理の機能と参照先

機能	参照先
ネーミング管理の概要	2.2
JNDI 名前空間へのオブジェクトのバインドとルックアップ※	2.3
Portable Global JNDI 名でのルックアップ	2.4
HITACHI_EJB から始まる名称でのルックアップ	2.5
Enterprise Bean または J2EE リソースへの別名付与（ユーザ指定名前空間機能）	2.6
ラウンドロビンポリシーによる CORBA ネーミングサービスの検索	2.7
ネーミング管理機能でのキャッシング	2.8
ネーミングサービスの障害検知	2.9
CORBA ネーミングサービスの切り替え	2.10
EJB ホームオブジェクトリファレンスの再利用（EJB ホームオブジェクトへの再接続機能）	2.11

注※　ルックアップには何種類かの方法があります。ルックアップ名称および方法については，「2.3.1　ルックアップで使用する名称の種類」を参照してください。

2.2　ネーミング管理の概要

この節では，ネーミング管理の機能，およびネーミング管理で使用するネーミングサービスについて説明します。

2.2.1　ネーミング管理の機能

ネーミング管理では，オブジェクト（Enterprise Bean に対応する EJB ホームオブジェクト，ビジネスインタフェースのリファレンスおよび J2EE リソース）の名前と格納場所を管理しています。ネーミング管理の機能を使用することで，EJB クライアントは，呼び出す Enterprise Bean またはリソースの格納場所を知らなくても，名前から必要なオブジェクトを利用できるようになります。

また，Connector 1.5 仕様に準拠したリソースアダプタを使用する場合，管理対象オブジェクトもネーミング管理で管理されています。管理対象オブジェクトは，J2EE アプリケーションの中からメッセージの送信および同期受信をするときに使用するオブジェクトです。管理対象オブジェクトについては，「3.16.5 管理対象オブジェクトのルックアップ」を参照してください。

ネーミング管理機能の JNDI では，CORBA オブジェクトリファレンス以外のオブジェクト（RMI-IIOP のリモートオブジェクトや JDBC データソースなどのオブジェクト）を次のように扱います。

- CORBA オブジェクトリファレンス以外の登録は，対象のオブジェクトを CORBA オブジェクトに変換し，CORBA オブジェクトリファレンスを CORBA ネーミングサービスへ登録することで実現しています。
- CORBA オブジェクト以外のオブジェクトの検索は，CORBA オブジェクトリファレンスを検索し，CORBA オブジェクトから逆変換して元のオブジェクトを取得することで実現しています。

なお，アプリケーションサーバで提供するネーミング管理の機能には，J2EE で規定された機能にアプリケーションサーバ独自の機能を拡張したものと，アプリケーションサーバ独自の機能として提供しているものがあります。アプリケーションサーバ独自の機能かどうかについては，「1.　アプリケーションサーバの機能」を参照してください。

アプリケーションサーバが提供するネーミング管理の機能と対象になるオブジェクトの関係を次の表に示します。

表 2-2　アプリケーションサーバが提供するネーミング管理の機能と対象になるオブジェクトの関係

機能	Enterprise Bean		J2EE リソース
	EJB ホームオブジェクト	ビジネスインタフェース	
JNDI 名前空間へのオブジェクトのバインドとルックアップ	○	○	○※1
Portable Global JNDI 名でのルックアップ	○	○	○
HITACHI_EJB から始まる名称でのルックアップ	○	○	×
Enterprise Bean または J2EE リソースへの別名付与（ユーザ指定名前空間機能）	○	○	○
ラウンドロビンポリシーによる CORBA ネーミングサービスの検索	○	○	×

機能	Enterprise Bean		J2EE リソース
	EJB ホームオブジェクト	ビジネスインタフェース	
ネーミング管理機能でのオブジェクトのキャッシング	○	×※2	×
CORBA ネーミングサービスの切り替え	○	×	×

（凡例）

○：使用できる　×：使用できない

注※1

HITACHI_EJB から始まる名称でのルックアップはできません。

注※2

ビジネスインタフェース使用時に，プロパティ「ejbserver.jndi.cache」に「on」を設定している状態で J2EE サーバを再起動した場合，ビジネスメソッド実行時に javax.ejb.EJBException が発生する場合があります。

! 注意事項

Web アプリケーションでカスタムエラーページを設定している場合，設定されたエラーページから JNDI を利用できません。

2.2.2　ネーミングサービス

ネーミング管理機能を使用する場合，オブジェクトの名前と格納場所は，**ネーミングサービス**によって管理されます。

アプリケーションサーバでは，ネーミング管理の機能として，**CORBA ネーミングサービス**を利用した JNDI をサポートしています。オブジェクトの登録，削除，または検索などの JNDI のインタフェースが呼び出されたときに，対応する CORBA ネーミングサービスのインタフェースを呼び出します。したがって，ネーミング管理機能を利用するためには，CORBA ネーミングサービスへの接続情報としてプロトコル，ホスト名，およびポート番号を設定する必要があります。

参考

CORBA ネーミングサービスは J2EE サーバのインプロセスで起動できます。インプロセスで起動することで，個別の起動・終了が不要になり，運用性が向上します。

2.3　JNDI 名前空間へのオブジェクトのバインドとルックアップ

ルックアップの対象になるオブジェクトは，JNDI 名前空間の名前と関連づけて管理されます。

J2EE サーバ内で実行するアプリケーションでは，利用したいリソースマネジャや呼び出したい Enterprise Bean などのオブジェクトを参照するために，JNDI の名前空間で検索（ルックアップ）を実行します。

この節では，ルックアップに使用する名称の種類，バインドされる名称の規則，およびルックアップの仕組みについて説明します。また，JNDI 名前空間の確認方法についても説明します。

この節の構成を次の表に示します。

表 2-3　この節の構成（JNDI 名前空間へのオブジェクトのバインドとルックアップ）

分類	タイトル	参照先
解説	ルックアップで使用する名称の種類	2.3.1
	JNDI 名前空間のマッピングとルックアップ	2.3.2
	JNDI 名前空間の確認方法	2.3.3
実装	cosminexus.xml での定義	2.3.4
設定	実行環境での設定	2.3.5

注　「運用」について，この機能固有の説明はありません。

2.3.1　ルックアップで使用する名称の種類

アプリケーションサーバでは，次に示す 4 種類の名称を使用したルックアップができます。

- Portable Global JNDI 名でのルックアップ

  Java EE で規定された java:global, java:app, または java:module から始まる名称（Portable Global JNDI 名）を指定してルックアップする方法です。Enterprise Bean または J2EE リソースをルックアップするときに使用できます。
- java:comp/env を使用した名称でのルックアップ

  Java EE で規定された java:comp/env を使用した名称を指定してルックアップする方法です。Enterprise Bean または J2EE リソースをルックアップするときに使用できます。
- HITACHI_EJB から始まる名称でのルックアップ

  アプリケーションサーバ独自の命名規則で自動的にバインドされる名称を指定してルックアップする方法です。Enterprise Bean をルックアップするときに使用できます。
- ユーザ指定名前空間機能を利用して付与した別名でのルックアップ

  アプリケーションサーバ独自の機能（ユーザ指定名前空間機能）を利用して登録した任意の名称を指定してルックアップする方法です。Enterprise Bean または J2EE リソースをルックアップするときに使用できます。

EJB クライアントの形態ごとに，利用できるルックアップ方法，および推奨するルックアップ方法を次の表に示します。

表 2-4　ルックアップ方法の種類

EJB クライアントの形態	ルックアップの方法			
	Portable Global JNDI 名でのルックアップ	java:comp/env を使用した名称でのルックアップ	HITACHI_EJB から始まる名称でのルックアップ	ユーザ指定名前空間機能を利用して付与した別名でのルックアップ
EJB クライアントアプリケーション	◎	×	○	○
JSP，サーブレット	○	◎	○	○
EJB	○	◎	○	○

（凡例）　◎：利用を推奨する　○：利用できる　×：利用できない

それぞれのルックアップ名称とその名称を使用したルックアップ方法について説明します。

(1)　Portable Global JNDI 名でのルックアップ

Portable Global JNDI 名でのルックアップについて，名称の定義方法，ルックアップできる範囲や特徴について説明します。

- 定義方法

  Portable Global JNDI 名を使用したルックアップには，自動的にバインドされる名称でのルックアップとリソース参照の定義に指定した名称でのルックアップがあります。

  - 自動的にバインドされる名称のルックアップの場合

    アプリケーションサーバでの J2EE アプリケーションのデプロイ時に，オブジェクトリファレンスに自動的に付与されます。この名称は，Java EE で規定されている命名規則で定義され，java:global，java:app，または java:module から始まります。この名称を **Portable Global JNDI 名**といいます。

  - リソース参照の定義に指定した名称のルックアップの場合

    Portable Global JNDI 名を DD のリソース参照の定義に指定します。

- ルックアップできる範囲

  同じ CORBA ネーミングサービスを利用するすべての J2EE サーバまたは EJB クライアントアプリケーション内で，Enterprise Bean または J2EE リソースをルックアップできます。

- 特徴

  J2EE アプリケーションのデプロイ時に，Enterprise Bean に対応する EJB ホームオブジェクトリファレンス，およびビジネスインタフェースのリファレンスが，自動的に java:global，java:app，または java:module から始まる名称にバインドされます。バインドされる名称には，標準アプリケーション名や標準モジュール名が付与されるので，名前空間の衝突を回避できます。また，Portable Global JNDI 名は，他社製のアプリケーションサーバ間に共通の JNDI 名のため，アプリケーションサーバ間の J2EE アプリケーションの移行の際に，ソースコードや DD の修正が不要です。

Portable Global JNDI 名でのルックアップについては，「2.4　Portable Global JNDI 名でのルックアップ」を参照してください。また，標準アプリケーション名および標準モジュール名については，「2.4.2　自動的にバインドされるオブジェクト」を参照してください。

(2)　java:comp/env を使用した名称でのルックアップ

java:comp/env を使用した名称のルックアップについて，名称の定義方法，ルックアップできる範囲や特徴について説明します。

- 定義方法

  Java EE で定義されている java:comp/env を使用した参照名と，実際の名称間のリンクの対応を DD のリソース参照に定義します。これによって，デプロイ時に参照名と実際の名称間のリンクを解決します。

- ルックアップできる範囲

  一つのコンポーネント内でルックアップできます。ただし，Web アプリケーションの場合は，一つの Web アプリケーション内でルックアップできます。

- 特徴

  java:comp/env は，Java EE で定義されている，名前空間のコンテキストルートです。
  java:comp/env を使用して，J2EE アプリケーション内の異なる EJB-JAR および Web アプリケーション間でのルックアップや，J2EE リソースのルックアップができます。

java:comp/env を使用した名称のルックアップについては，Java EE の規定どおりの仕様です。

(3)　HITACHI_EJB から始まる名称でのルックアップ

HITACHI_EJB から始まる名称のルックアップについて，名称の定義方法，ルックアップできる範囲や特徴について説明します。

- 定義方法

  アプリケーションサーバでの J2EE アプリケーションのデプロイ時に，オブジェクトリファレンスに自動的に付与されます。この名称は，アプリケーションサーバ独自の命名規則で定義されます。

- ルックアップできる範囲

  同じ CORBA ネーミングサービスを利用するすべての J2EE サーバまたは EJB アプリケーション内で Enterprise Bean をルックアップできます。

- 特徴

  J2EE アプリケーションのデプロイ時に，Enterprise Bean に対応する EJB ホームオブジェクトリファレンス，およびビジネスインタフェースのリファレンスが，自動的に HITACHI_EJB から始まる名称にバインドされます。バインドされる名称には，J2EE アプリケーションのサーバ名やアプリケーション名が付与されるので，名前空間の衝突を回避できます。

HITACHI_EJB から始まる名称でのルックアップについては，「2.5　HITACHI_EJB から始まる名称でのルックアップ」を参照してください。

(4)　ユーザ指定名前空間機能を利用して付与した別名でのルックアップ

ユーザ指定名前空間機能を利用して付与した別名でのルックアップについて，名称の定義方法，ルックアップできる範囲や特徴について説明します。

- 定義方法

  アプリケーションサーバの**ユーザ指定名前空間機能**を利用して，Enterprise Bean または J2EE リソースに別名を定義します。

- ルックアップできる範囲

同じ CORBA ネーミングサービスを利用するすべての J2EE サーバ内で Enterprise Bean または J2EE リソースをルックアップできます。

- 特徴

  ユーザ指定名前空間機能を利用すると，Enterprise Bean の場合は JNDI の名前へのバインドと合わせて，EJB ホームオブジェクトリファレンスまたはビジネスインタフェースのリファレンスの別名を登録できます。また，J2EE リソース（リソースアダプタ，メールコンフィグレーション，JavaBeans リソース）に対しても，別名を登録できます。

  別名を登録することで，ユーザが指定した任意の名称で Enterprise Bean および J2EE リソースをルックアップできるようになります。

Enterprise Bean または J2EE リソースへの別名付与の詳細については，「2.6　Enterprise Bean または J2EE リソースへの別名付与（ユーザ指定名前空間機能）」を参照してください。

参考

Java EE の仕様では，java:comp/env を使用した名称でのルックアップが推奨されています。

2.3.2　JNDI 名前空間のマッピングとルックアップ

ここでは，JNDI 名前空間のマッピングの仕組みと，ルックアップでの指定する名称と JNDI 名前空間および DD の関係について説明します。

(1)　Enterprise Bean を参照する仕組みと使い方（ejb-ref）

J2EE サーバ内で実行するアプリケーションでは，Enterprise Bean を呼び出すために，Enterprise Bean 参照の名前解決が必要です。正確には，Enterprise Bean に対応する EJB ホームインタフェース，EJB ローカルホームインタフェース，ビジネスインタフェースなどの名前解決をします。

EJB ホームインタフェース，またはビジネスインタフェースの名前解決をするためには，次の 2 種類の作業が必要です。

- **参照側アプリケーションの作成時の作業**

  アプリケーションの作成時に参照用の名称を決め，Enterprise Bean やサーブレットなどのプログラム内での lookup の引数に，その参照用の名称を指定します。また，java:comp/env を使用する場合は，DD の<ejb-ref>に，決めた参照用の名称を記載します。

- **参照側アプリケーションのデプロイ時の作業**

  作成したアプリケーションを J2EE サーバ上にデプロイするときに，Enterprise Bean をカスタマイズして，参照用の名称と実際の名称を linked-to で結び付けます。

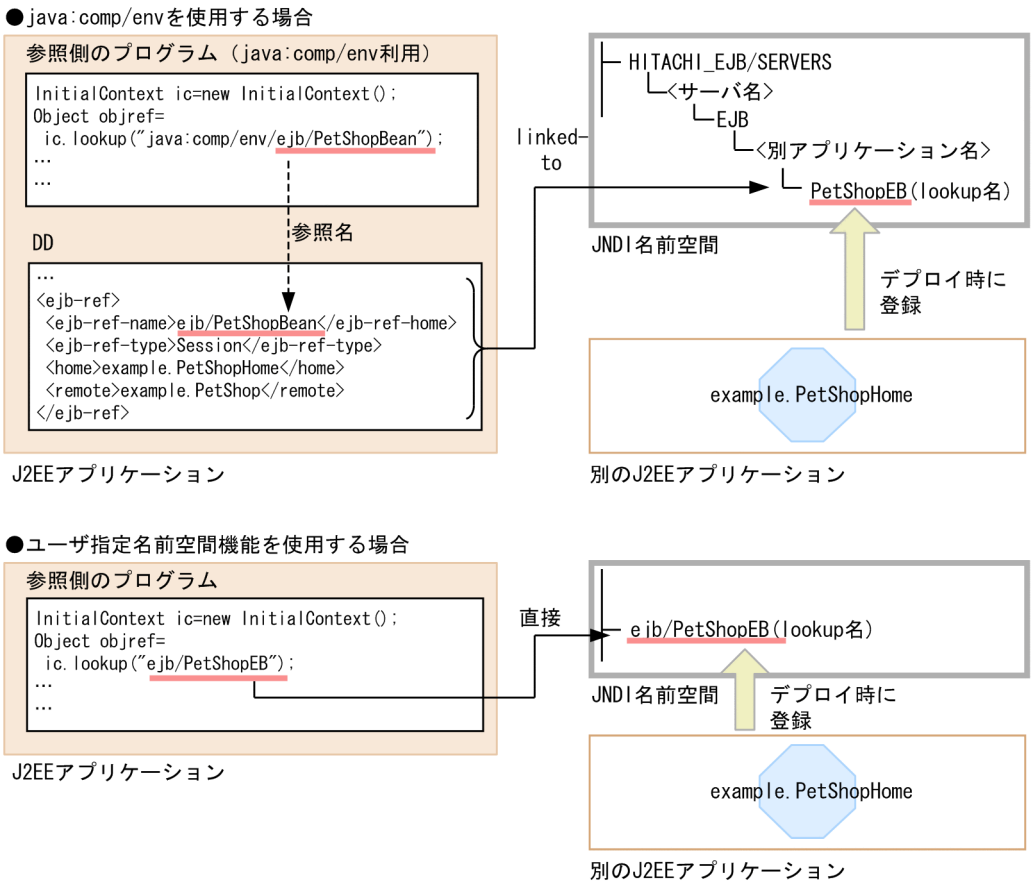
  この操作は，サーバ管理コマンドを使用して実行できます。

マッピングの仕組みは，参照側のアプリケーションと参照先のアプリケーションが同じ J2EE アプリケーションか，別の J2EE アプリケーションかによって異なります。次に，EJB ホームインタフェースの名前解決の例として，別の J2EE アプリケーションの Enterprise Bean を参照する場合と，同じ J2EE アプリケーション内の Enterprise Bean を参照する場合の例を示します。

(a)　別の J2EE アプリケーションの Enterprise Bean を参照する場合

J2EE アプリケーションから別の J2EE アプリケーションに含まれる Enterprise Bean を参照する例を，次の図に示します。

図 2-1　J2EE アプリケーションから別の J2EE アプリケーションに含まれる Enterprise Bean を参照する例



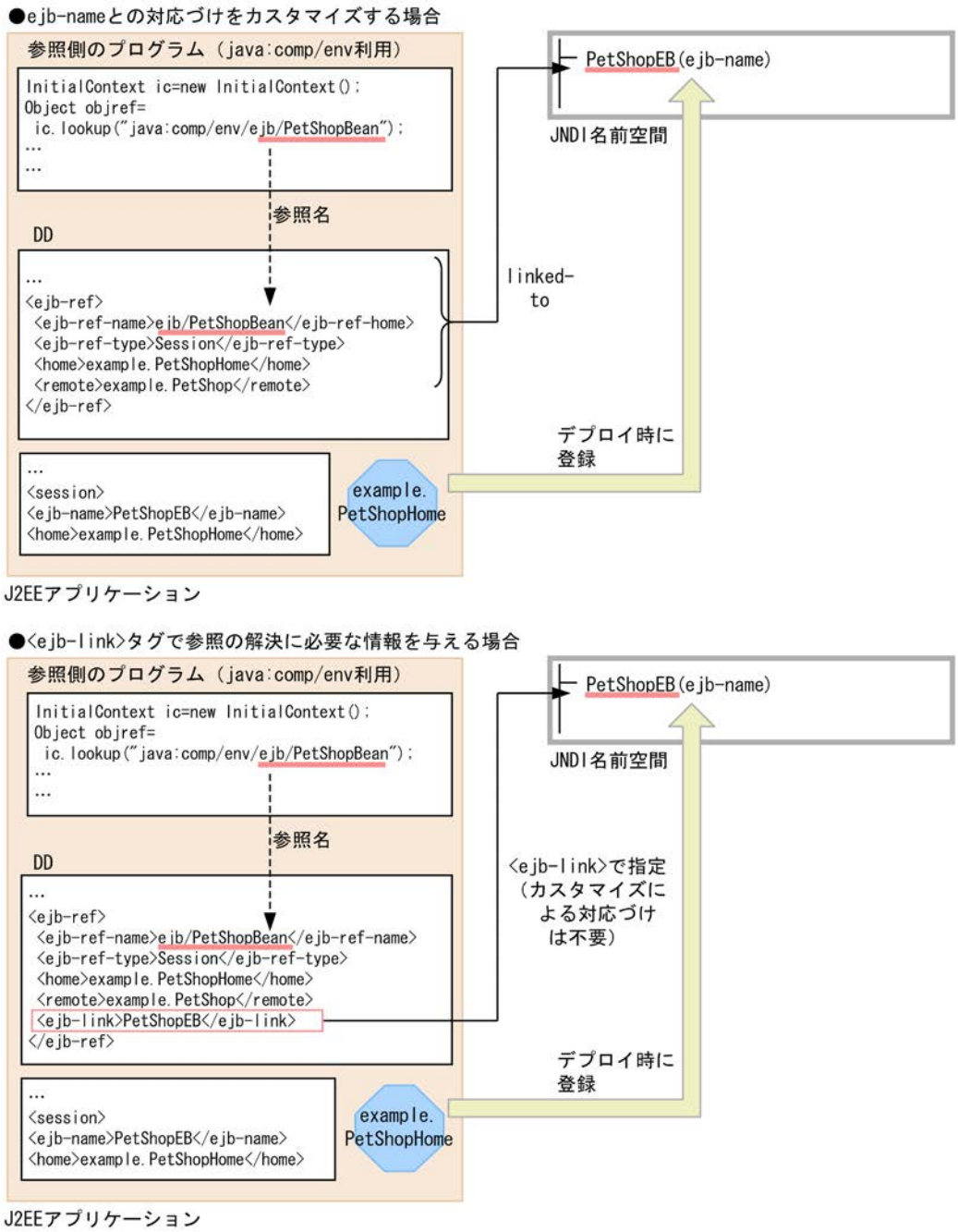
java:comp/env を使用する場合，参照側のプログラムの lookup に指定する java:comp/env 下の参照名と実際に JNDI 名前空間に登録されている lookup 名は，デフォルトで対応づけられます（linked-to）。なお，lookup 名を JNDI 名前空間に登録されている名称と異なる名称にする場合は，サーバ管理コマンドでカスタマイズする必要があります。図の場合は，PetShopEB を ejb/PetShopBean として呼び出すようにカスタマイズします。

ユーザ指定名前空間機能を使用する場合，参照側のプログラムの lookup に別名を指定できます。<ejb-ref>の定義は不要です。

(b)　同じ J2EE アプリケーション内の Enterprise Bean を参照する場合

J2EE アプリケーションから同じ J2EE アプリケーションに含まれる Enterprise Bean を参照する例を，次の図に示します。

図 2-2　同じ J2EE アプリケーションに含まれる Enterprise Bean を参照する例



同じアプリケーション内の Enterprise Bean を呼び出す場合は，lookup 名による参照の解決ではなく，<ejb-ref-name>と ejb-name を結び付けて解決します（linked-to）。なお，同一の J2EE アプリケーションの場合，参照に必要な情報はアプリケーション開発時に判明しているので，DD の<ejb-link>タグに直接 ejb-name を書き込んでおくこともできます。

(2)　リソースを参照する仕組みと使い方（resource-ref）

J2EE サーバ内で実行するアプリケーションでリソースアダプタを利用する場合，lookup によるリソース参照の名前解決が必要です。正確には，リソースマネジャへのコネクションを作成するファクトリの名前解決をします。

リソースマネジャへのコネクションを作成するファクトリの名前解決をするためには，次の 2 種類の作業が必要です。

- **参照側アプリケーションの作成時の作業**

  アプリケーションの作成時に参照用の名称を決め，Enterprise Bean やサーブレットなどのプログラム内での lookup の引数に，その参照用の名称を指定します。また，DD の<resource-ref>に，決めた参照用の名称を記載します。

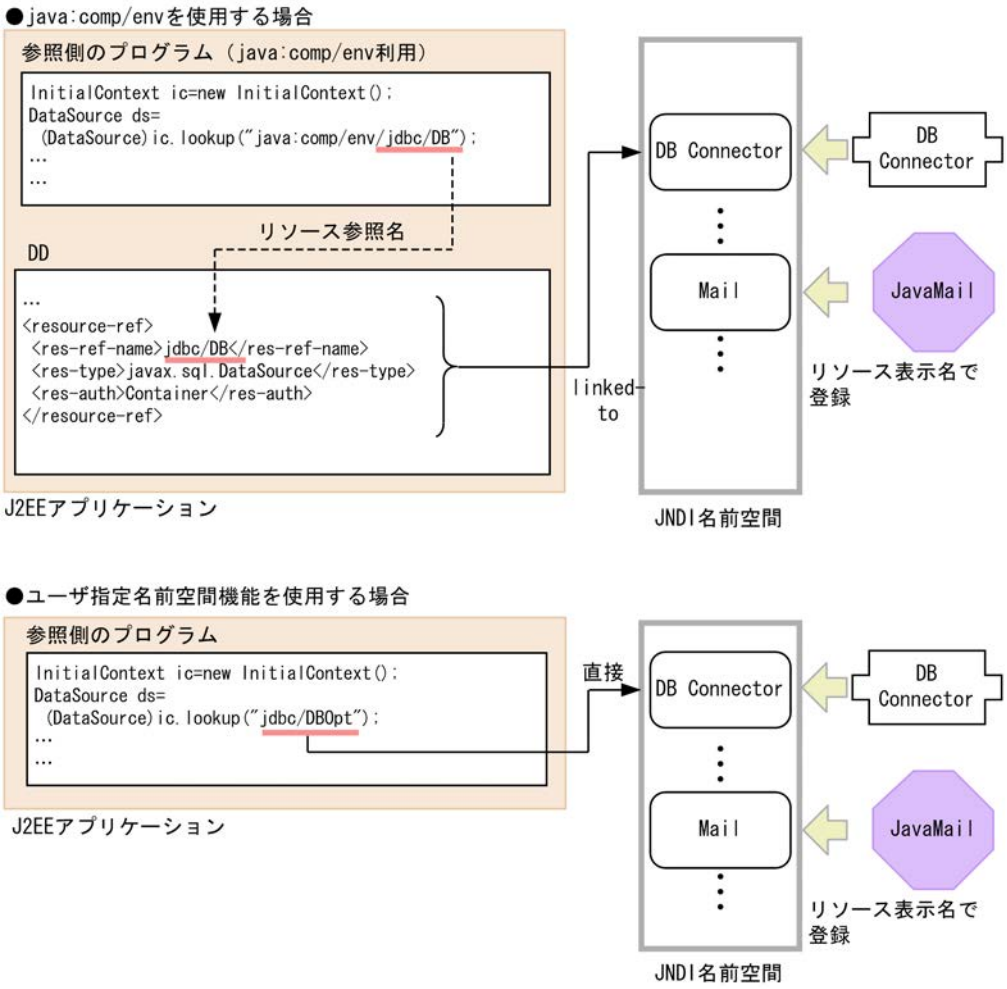
- **参照側アプリケーションのデプロイ時の作業**

  作成したアプリケーションを J2EE サーバ上にデプロイするときに，リソースをカスタマイズして，参照用の名称と J2EE サーバ上に登録済みのリソースを linked-to で結び付けます。

  この操作は，サーバ管理コマンドを使用して実行できます。

次に DB Connector 経由でデータベースにアクセスする場合の例を示します。

図 2-3　DB Connector 経由でデータベースにアクセスする場合の例



この例では，リソースの参照名称とリソースの表示名称を結び付けています。

2.3.3　JNDI 名前空間の確認方法

コマンドを使用して，CORBA ネーミングサービスで認識されている名称を表示して確認できます。なお，J2EE リソースの別名や EJB ローカルホームオブジェクトリファレンスの登録情報は確認できません。使

用するコマンドは，nsutil コマンドです。nsutil コマンドの使用方法，使用条件などについては，マニュアル「Borland(R) Enterprise Server VisiBroker(R) デベロッパーズガイド」を参照してください。

JNDI 名前空間の「HITACHI_EJB/SERVERS/MyServer/EJB」の下位の一覧を表示する場合の表示例を次に示します。

```
# nsutil -VBJprop ORBInitRef=NameService=corbaname::localhost:900 list HITACHI_EJB/SERVERS/MyServer/EJB
Bindings in HITACHI_EJB/SERVERS/MyServer/EJB
Context: App1
Context: App2
Context: App3
```

2.3.4　cosminexus.xml での定義

ネーミング管理の機能の定義は，cosminexus.xml の<ejb-jar>タグ内に指定します。cosminexus.xml でのネーミング管理のための定義について次の表に示します。

表 2-5　cosminexus.xml でのネーミング管理の機能の定義

指定するタグ	設定内容
<session>－<lookup-name>タグ <entity>－<lookup-name>タグ	ルックアップ名称を指定します。

cosminexus.xml については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「2.　アプリケーション属性ファイル（cosminexus.xml)」を参照してください。

2.3.5　実行環境での設定

ネーミング管理機能を使用する場合，J2EE サーバの設定，サーバ管理コマンドのカスタマイズ，J2EE アプリケーションの設定が必要です。

(1)　J2EE サーバの設定

J2EE サーバの設定は，簡易構築定義ファイルで実施します。ネーミング管理機能の定義は，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に指定します。簡易構築定義ファイルでのネーミング管理機能の定義について次の表に示します。

表 2-6　簡易構築定義ファイルでのネーミング管理の機能の定義

項目	指定するパラメタ	設定内容
基本設定	ejbserver.naming.host※	CORBA ネーミングサービスのホスト名を指定します。
	ejbserver.naming.port※	CORBA ネーミングサービスのポート番号を指定します。
	ejbserver.naming.startupMode	CORBA ネーミングサービスの起動モードを指定します。
ネーミングサービスの通信タイムアウト	ejbserver.jndi.request.timeout	ネーミングサービスとの通信タイムアウト時間を指定します。

注※　デフォルトの設定では，J2EE サーバはホスト名「localhost」，ポート番号「900」の CORBA ネーミングサービスをインプロセスで自動起動して使用します。設定を変更したい場合に変更してください。

簡易構築定義ファイルおよびパラメタについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.6　簡易構築定義ファイル」を参照してください。

(2)　サーバ管理コマンドのカスタマイズ

サーバ管理コマンドの動作設定をカスタマイズできます。ここでは，サーバ管理コマンドが使用する CORBA ネーミングサービスの設定について説明します。

ネーミング管理の機能を使用するためのサーバ管理コマンドのカスタマイズは，usrconf.properties（サーバ管理コマンド用システムプロパティファイル）で設定します。設定内容を次に示します。なお，キーの詳細，およびここで説明していないキーについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「5.　サーバ管理コマンドで使用するファイル」を参照してください。

表 2-7　ネーミング管理機能を使用するためのサーバ管理コマンドのカスタマイズ

項目	指定するキー	設定内容
CORBA ネーミングサービスが動作するホスト名	ejbserver.naming.host	サーバ管理コマンドが使用する CORBA ネーミングサービスが動作するホストを指定します。
CORBA ネーミングサービスのポート番号	ejbserver.naming.port	サーバ管理コマンドが使用する CORBA ネーミングサービスのポート番号を指定します。
CORBA ネーミングサービスへのアクセスプロトコル	ejbserver.naming.protocol	サーバ管理コマンドが使用する CORBA ネーミングサービスへのアクセスプロトコルを指定します。

(3)　J2EE アプリケーションの設定

実行環境での J2EE アプリケーションの設定は，サーバ管理コマンドおよび属性ファイルで実施します。ネーミング管理の機能の定義には，Session Bean 属性ファイルまたは Entity Bean 属性ファイルを使用します。

これらの属性ファイルで指定するタグは，cosminexus.xml と対応しています。cosminexus.xml での定義については，「2.3.4　cosminexus.xml での定義」を参照してください。

2.4　Portable Global JNDI 名でのルックアップ

J2EE アプリケーションをデプロイすると，EJB ホームオブジェクトリファレンスおよびビジネスインタフェースのリファレンスの JNDI の名前に，Portable Global JNDI 名が自動的にバインドされます。ルックアップ時には，バインドされた名前を使用します。

ここでは，Java EE で定義されている名前空間，Portable Global JNDI 名でバインドされるオブジェクト，命名規則，DD での定義，および実行環境での設定について説明します。

この節の構成を次の表に示します。

表 2-8　この節の構成（Portable Global JNDI 名でのルックアップ）

分類	タイトル	参照先
解説	JNDI 名前空間の種類	2.4.1
	自動的にバインドされるオブジェクト	2.4.2
	Portable Global JNDI 名の命名規則	2.4.3
	Portable Global JNDI 名の登録抑止	2.4.4
	CTM を使用する場合の Portable Global JNDI 名のルックアップ	2.4.5
	リソース参照の名称	2.4.6
	アノテーションでの Portable Global JNDI 名の指定	2.4.7
実装	DD での定義	2.4.8
設定	実行環境での設定	2.4.9

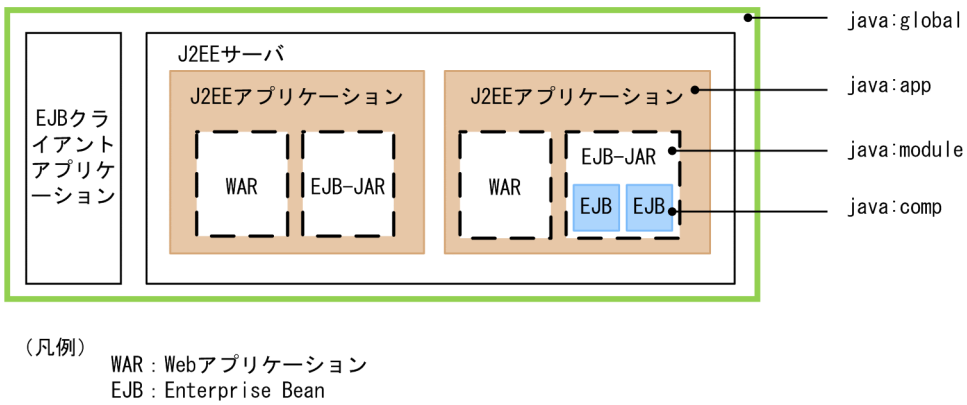
注　「運用」および「注意事項」について，この機能固有の説明はありません。

2.4.1　JNDI 名前空間の種類

Java EE で定義されている JNDI 名前空間の種類とルックアップできる範囲について説明します。

JNDI 名前空間の範囲を次の図に示します。

図 2-4　JNDI 名前空間の範囲



（凡例）
WAR：Webアプリケーション
EJB：Enterprise Bean

- java:global

すべての J2EE アプリケーションで共有される名前空間です。アプリケーションサーバでは，同じ CORBA ネーミングサービスを利用するすべての J2EE サーバまたは EJB クライアントアプリケーションで共有されます。

- java:app
  J2EE アプリケーション内で共有される名前空間です。一つの J2EE アプリケーション内のすべてのコンポーネント（Enterprise Bean，サーブレット，JSP，フィルタ，およびリソースアダプタ）で共有されます。
- java:module
  各種モジュール（EJB-JAR，Web アプリケーションまたはリソースアダプタ）内で共有される名前空間です。一つのモジュール内のすべてのコンポーネント（Enterprise Bean，サーブレット，JSP，フィルタ，およびリソースアダプタ）で共有されます。
- java:comp
  各種コンポーネント（Enterprise Bean，サーブレット，JSP，フィルタ，またはリソースアダプタ）内で共有される名前空間です。一つのコンポーネント内でだけ共有されます。ただし，Web アプリケーションの場合は，Web アプリケーション内のすべてのサーブレットまたは JSP で共有されます。

ルックアップ対象のコンポーネントがルックアップ元と同じプロセス，アプリケーション，モジュールまたはコンポーネントに含まれているかによって，使用できる名前空間が異なります。ルックアップするコンポーネントとの関係と，名前空間ごとのルックアップ可否を次の表に示します。

表 2-9　名前空間ごとのルックアップ可否

項番	ルックアップするコンポーネントとの関係				ルックアップする名前空間			
	サーバ（プロセス）	アプリケーション	モジュール	コンポーネント	java:global	java:app	java:module	java:comp
1	同じ	同じ	同じ	同じ	○	○	○	○
2	同じ	同じ	同じ	異なる	○	○	○	△※1
3	同じ	同じ	異なる	異なる	○	○	×	×
4	同じ	異なる	異なる	異なる	○	×	×	×
5	異なる※2	異なる	異なる	異なる	○※3	×	×	×

（凡例）

○：ルックアップできる。

△：コンポーネントの種類によってルックアップできる。

×：ルックアップできない。

注※1

Web アプリケーション内のコンポーネント（サーブレット，JSP またはフィルタ）の場合，ルックアップできます。

注※2

サーバ（プロセス）は異なるが，同一の CORBA ネーミングサービスを利用している場合を指します。EJB クライアントアプリケーションから J2EE サーバ内の J2EE アプリケーションをルックアップする場合を含みます。

注※3

自動的にバインドされる EJB のリファレンスだけルックアップできます（アプリケーションサーバ独自の仕様）。

2.4.2　自動的にバインドされるオブジェクト

アプリケーションサーバでは，標準アプリケーション名，標準モジュール名およびEJBのリファレンスが，Java EEで定義されている名称に自動的にバインドされます。なお，アプリケーションサーバでは標準アプリケーション名，および標準モジュール名を次のように定義しています。

- 標準アプリケーション名

  Java EE 6で導入されたJ2EEアプリケーションを一意に識別する名称です。アプリケーションサーバでは，application.xmlの<display-name>に指定する「J2EEアプリケーション名」または「アプリケーション表示名」とは区別して，「標準アプリケーション名」と呼びます。

- 標準モジュール名

  Java EE 6で導入されたJ2EEアプリケーション内の各種モジュール（EJB-JAR，Webアプリケーションまたはリソースアダプタ）を一意に識別する名称です。アプリケーションサーバでは，EJB-JARファイルやWARファイルのファイル名を指す「モジュール名」と区別して，「標準モジュール名」と呼びます。

自動的にバインドされるオブジェクトと名称を次の表に示します。

表 2-10　自動的にバインドされるオブジェクトと名称

バインドされるオブジェクト		バインドされる名称
標準アプリケーション名（java.lang.String型）		java:app/AppName※1
標準モジュール名（java.lang.String型）		java:module/ModuleName※1
EJBのリファレンス※2	Session Beanのホームオブジェクト	java:global[/<標準アプリケーション名>]/<標準モジュール名>/<Enterprise Bean名>[!<完全修飾クラス名>]
		java:app/<標準モジュール名>/<Enterprise Bean名>[!<完全修飾クラス名>]
		java:module/<Enterprise Bean名>[!<完全修飾クラス名>]
	Session Beanのビジネスインタフェース	java:global[/<標準アプリケーション名>]/<標準モジュール名>/<Enterprise Bean名>[!<完全修飾クラス名>]
		java:app/<標準モジュール名>/<Enterprise Bean名>[!<完全修飾クラス名>]
		java:module/<Enterprise Bean名>[!<完全修飾クラス名>]

注※1

EJB-JARまたはWebアプリケーションからの場合だけルックアップできます。

注※2

EJB 3.1仕様では，一つのEnterprise Beanに対してEJBのリファレンス（ホームオブジェクトやビジネスインタフェース）が一つの場合に，完全修飾クラス名を省略したクラス名省略書式でルックアップできます。

さらに，アプリケーションサーバでは，リファレンスが複数ある場合でも，次の優先順位で最初に登録されたものは，クラス名省略書式でルックアップできます。

1. リモートホームオブジェクト
2. ローカルホームオブジェクト
3. ローカルビジネスインタフェース（ビジネスインタフェース省略時を含む）
4. リモートビジネスインタフェース

ただし，リファレンスが複数ある場合のクラス名省略書式に関する仕様はJava EEで規定されていないため，クラス名省略書式でのルックアップは推奨しません。

HITACHI_EJB から始まる名称でルックアップできるオブジェクトがアプリケーション内にある場合は，Portable Global JNDI 名でもルックアップできます。

なお，アプリケーションの開始時，Portable Global JNDI 名は，バインドされた EJB リファレンスごとにメッセージ KDJE47701-I に出力されます。Enterprise Bean のクラス名を省略した書式やアプリケーション名を省略した書式が有効な場合は，ルックアップできるすべての書式がコンマと空白区切り（, ）で列挙されます。

2.4.3　Portable Global JNDI 名の命名規則

Portable Global JNDI 名には標準アプリケーション名，標準モジュール名，および Enterprise Bean 名が含まれています。これらの名前は Java EE で命名規則や使用できる文字が定義されています。

ここでは，Portable Global JNDI 名に使用される標準アプリケーション名，標準モジュール名，Enterprise Bean 名の命名規則，および使用できる文字について説明します。

(1)　標準アプリケーション名の命名規則

標準アプリケーション名は次の規則で設定されます。

表 2-11　標準アプリケーション名の命名規則

<table>
<tr><th colspan="4">アプリケーションの前提条件</th><th>標準アプリケーション名</th></tr>
<tr><td rowspan="4">J2EE アプリケーション（EAR ファイル）</td><td rowspan="3">application.xml がある</td><td rowspan="2"><application-name>タグがある</td><td>値がある</td><td><application-name>タグの値※1</td></tr>
<tr><td>値が空文字※2</td><td rowspan="3">デフォルトの標準アプリケーション名※3</td></tr>
<tr><td><application-name>タグがない</td><td>－</td></tr>
<tr><td>application.xml がない</td><td>－</td><td>－</td></tr>
<tr><td rowspan="2">WAR アプリケーション（WAR ファイル）</td><td colspan="3">cjimportwar コマンドの-name オプションに指定がある</td><td>cjimportwar コマンドの-name オプションの値※1</td></tr>
<tr><td colspan="3">cjimportwar コマンドの-name オプションに指定がない</td><td>デフォルトの標準アプリケーション名※3</td></tr>
</table>

（凡例）

－：該当なし。

注※1

値の前後の空白文字（半角スペース，タブ，改行）は除かれます。単語の間にある空白文字は除かれません。

注※2

長さが 0 の文字列を指します。空白文字だけで構成された値も空文字と判断されます。

注※3

デフォルトの標準アプリケーション名は，J2EE アプリケーションの形式によって異なります。デフォルトの標準アプリケーション名の命名規則を次の表に示します。

表 2-12　デフォルトの標準アプリケーション名の命名規則

アプリケーションの前提条件		デフォルトの標準アプリケーション名
J2EE アプリケーション（EAR ファイル）	アーカイブ形式	EAR ファイルのファイル名から拡張子を除いた文字列。 ただし，ピリオド「.」がファイル名の先頭にしかない場合は拡張子を除かない。
	展開ディレクトリ形式	• application.xml がある場合 J2EE アプリケーション名（<display-name>タグの値）。 • application.xml がない場合 アプリケーションディレクトリのディレクトリ名。
WAR アプリケーション（WAR ファイル）	アーカイブ形式	WAR ファイルのファイル名から拡張子を除いた文字列。 ただし，ピリオド「.」がファイル名の先頭にしかない場合は拡張子を除かない。
	展開ディレクトリ形式	WAR ディレクトリのディレクトリ名。

標準アプリケーション名に指定できる値を次に示します。

<application-name>タグで指定する標準アプリケーション名

標準アプリケーション名は，半角英数字（0～9，A～Z，a～z），および次の特殊文字の場合に登録されます。

スペース（ ），エクスクラメーションマーク（!），ダブルクォーテーション（"），シャープ（#），ドル記号（$），パーセント（%），アンパサンド（&），シングルクォーテーション（'），バーレン（(）（)），アスタリスク（*），プラス（+），コンマ（,），ハイフン（-），ピリオド（.），コロン（:），セミコロン（;），レスザン（<），イコール（=），グレーターザン（>），クエスチョン（?），単価記号（@），ブラケット（[）（]），円マーク（¥），キャレット（^），アンダースコア（_），バッククォート（`），ブレイス（{）（}），ストローク（|），およびチルダ（~）

ただし，次の名称は登録されません。

- 先頭または末尾がピリオド（.）の名称
- ピリオド（.）だけの名称
- 文字列長が 256 文字以上の名称
- 「env」と一致する名称

<display-name>タグで指定する標準アプリケーション名

標準アプリケーション名（<display-name>タグの値）は，半角英数字（0～9，A～Z，a～z），プラス（+），ハイフン（-），ピリオド（.），アンダースコア（_），およびキャレット（^）の場合に登録されます。

ただし，次の名称は登録されません。

- 先頭または末尾がピリオド（.）の名称
- ピリオド（.）だけの名称
- 文字列長が 1 文字未満の名称
- 文字列長が 256 文字以上の名称
- 「env」と一致する名称

登録されない名称の例

foo/bar，.foo，.foobar.，env

標準アプリケーション名が登録できない名称の場合，アプリケーション開始時に KDJE47710-W が出力され，アプリケーションの開始処理は続行されます。ただし，そのアプリケーションに含まれるすべてのオブジェクトに対して Portable Global JNDI 名の登録はされないため，Portable Global JNDI 名でのルックアップはできません。Portable Global JNDI 名でのルックアップをする場合は，必要に応じて，DD，EAR ファイルのファイル名，またはアプリケーションディレクトリのディレクトリ名を命名規則に従って変更してください。

(2)　標準モジュール名の命名規則

標準モジュール名は次の規則で設定されます。

表 2–13　標準モジュール名の命名規則

モジュールの前提条件				標準モジュール名
EJB-JAR モジュール（EJB-JAR ファイル）	ejb-jar.xml がある	<module-name>タグがある	値がある	<module-name>タグの値※1
			値が空文字※2	デフォルトの標準モジュール名※3
		<module-name>タグがない	－	
	ejb-jar.xml がない	－	－	
Web モジュール（WAR ファイル）	web.xml がある	<module-name>タグがある	値がある	<module-name>タグの値※1
			値が空文字※2	デフォルトの標準モジュール名※3
		<module-name>タグがない	－	
	web.xml がない	－	－	
リソースアダプタモジュール（RAR ファイル）	－			デフォルトの標準モジュール名※3

（凡例）

－：該当なし。

注※1

値の前後の空白文字（半角スペース，タブ，改行）は除かれます。単語の間にある空白文字は除かれません。

注※2

長さが 0 の文字列を指します。空白文字だけで構成された値も空文字と判断されます。

注※3

デフォルトの標準モジュール名は，モジュールおよび J2EE アプリケーションの形式によって異なります。デフォルトの標準モジュール名の命名規則を次の表に示します。

表 2-14　デフォルトの標準モジュール名の命名規則

モジュール	J2EE アプリケーションの形式	デフォルトの標準モジュール名
EJB-JAR モジュール（EJB-JAR ファイル）	アーカイブ形式	アプリケーションパッケージのルートディレクトリから EJB-JAR ファイルまでの相対パスから拡張子を除いた文字列。 ただし，ピリオド「.」が EJB-JAR ファイル名の先頭にしかない場合は拡張子を除かない。 パス区切り文字が「¥」の場合は「/」に変換する。
	展開ディレクトリ形式	• application.xml がある場合 application.xml の<module>/<ejb>タグで指定したパス。 • application.xml がない場合 アプリケーションディレクトリから EJB-JAR ディレクトリまでの相対パスから，末尾の「_jar」を除いた文字列。 パス区切り文字が「¥」の場合は「/」に変換する。
Web モジュール（WAR ファイル）	アーカイブ形式	アプリケーションパッケージのルートディレクトリから WAR ファイルまでの相対パスから，拡張子を除いた文字列。 ただし，ピリオド「.」が WAR ファイル名の先頭にしかない場合は拡張子を除かない。 パス区切り文字が「¥」の場合は「/」に変換する。
	展開ディレクトリ形式	• application.xml がある場合 application.xml の<module>/<web>/<web-uri>タグで指定したパス。 • application.xml がない場合 アプリケーションディレクトリから WAR ディレクトリまでの相対パスから，末尾の「_war」を除いた文字列。 パス区切り文字が（¥）の場合は（/）に変換する。
リソースアダプタモジュール	アーカイブ形式	アプリケーションパッケージのルートディレクトリから RAR ファイルまでの相対パスから，拡張子を除いた文字列。 ただし，ピリオド（.）が RAR ファイル名の先頭にしかない場合は拡張子を除かない。 パス区切り文字が（¥）の場合は（/）に変換する。

なお，標準モジュール名（<module-name>タグの値）は，半角英数字（0～9，A～Z，a～z），および次の特殊文字の場合に登録されます。

スペース（ ），エクスクラメーションマーク（!），ダブルクォーテーション（"），シャープ（#），ドル記号（$），パーセント（%），アンパサンド（&），シングルクォーテーション（'），バーレン（(）（)），アスタリスク（*），プラス（+），コンマ（,），ハイフン（-），ピリオド（.），区切り文字としてのスラッシュ（/），コロン（:），セミコロン（;），レスザン（<），イコール（=），グレーターザン（>），クエスチョン（?），単価記号（@），ブラケット（[）（]），円マーク（¥），キャレット（^），アンダースコア（_），バッククォート（`），ブレイス（{）（}），ストローク（|），チルダ（~）

ただし，次の名称は登録されません。

- 先頭または末尾がスラッシュ（/）の名称
- スラッシュ（/）だけの名称
- スラッシュ（/）が連続する名称

- 先頭または末尾がピリオド（.）の名称
- ピリオド（.）だけの名称
- スラッシュ（/）とピリオド（.）が連続する名称
- 文字列長が 256 文字以上の名称
- 「env」と一致する名称
- 「env/」から始まる名称
- 「AppName」と一致する名称
- 「AppName/」から始まる名称

登録されない名称の例

/foo，/foobar/，/，foo//bar，bar.，.foobar.，foo/.bar，AppName，AppName/foo，env/foo/bar

標準モジュール名が登録できない名称の場合，アプリケーション開始時にモジュールごとに KDJE47711-W が出力され，アプリケーションの開始処理は続行されます。ただし，そのモジュールに含まれるすべてのオブジェクトに対して Portable Global JNDI 名での登録はされないため，Portable Global JNDI 名でのルックアップはできません。Portable Global JNDI 名でのルックアップをする場合は，必要に応じて，DD，アプリケーションのディレクトリ名などを命名規則に従って変更してください。

(3)　Enterprise Bean 名の命名規則

Enterprise Bean 名には，半角英数字（0～9，A～Z，a～z），および次の特殊文字の場合に登録されます。

スペース（ ），エクスクラメーションマーク（!），ダブルクォーテーション（"），シャープ（#），ドル記号（$），パーセント（%），アンパサンド（&），シングルクォーテーション（'），バーレン（() ()），アスタリスク（*），プラス（+），コンマ（,），ハイフン（-），ピリオド（.），コロン（:），セミコロン（;），レスザン（<），イコール（=），グレーターザン（>），クエスチョン（?），単価記号（@），ブラケット（[） (]），円マーク（¥），キャレット（^），アンダースコア（_），バッククォート（`），ブレイス（{） (}），ストローク（|），チルダ（~）

ただし，次の名称は登録されません。

- 先頭または末尾がピリオド（.）の名称
- ピリオド（.）だけの名称
- 文字列長が 256 文字以上の名称
- 「ModuleName」と一致する名称
- 「env」と一致する名称

登録されない名称の例

.foo，.foobar.，ModuleName，env

Enterprise Bean 名が登録できない名称の場合，アプリケーション開始時に Enterprise Bean ごとに KDJE47712-W が出力され，アプリケーションの開始処理は続行されます。ただし，その Enterprise Bean に含まれるすべてのオブジェクトに対して Portable Global JNDI 名での登録はされません。なお，Enterprise Bean 名が登録できない名称の場合，Portable Global JNDI 名でのルックアップはできませんが，それ以外のアプリケーションの機能は従来どおり動作します。

また，指定できる文字列長については，Enterprise Bean のインタフェース（ホームインタフェースまたはビジネスインタフェース）ごとにもチェックされます。例えば，次の文字列長が 256 文字以上の場合，インタフェースごとに KDJE47713-W が出力され，そのインタフェース名を指定した形式の Portable Global JNDI 名での登録はされません。

Enterprise Bean名の長さ+インタフェースのクラス名※の長さ+1

注※　パッケージ名を含む完全修飾クラス名

なお，EJB 3.1 では，インタフェースが省略されている場合は，インタフェースのクラス名の代わりに Enterprise Bean のクラス名が適用されます。

(4)　標準アプリケーション名または標準モジュール名が重複した場合の動作

アプリケーションサーバでは，アプリケーション開始時に，すでに同じ標準アプリケーション名が名前空間に登録されていた場合，KDJE47720-W が出力され，名前空間に登録されません。これと同様に，同一アプリケーション内ですでに同じ標準モジュール名が名前空間に登録されていた場合，次のメッセージが出力され，名前空間に登録されません。ただし，アプリケーションの開始処理は続行されます。

- EJB-JAR 場合：KDJE47721-W
- Web アプリケーションの場合：KDJE47722-W
- リソースアダプタの場合：KDJE47723-W

なお，アプリケーションサーバでは，次の順番でモジュールが登録されます。

1. リソースアダプタ
2. EJB-JAR
3. Web アプリケーション

登録済みの標準アプリケーション名や標準モジュール名と完全に一致していなくても，スラッシュ(/)で区切られた階層の単位で，登録済みのオブジェクトと名称が一致した場合は，重複していると判断されることがあります。次に例を示します。

例 1

次の順番でアプリケーションを開始した場合，標準アプリケーション名が重複していると判断されます。

1. 標準アプリケーション名が「foo/bar」のアプリケーションを開始
2. 標準アプリケーション名が「foo」のアプリケーションを開始

例 2

次の順番でアプリケーションを開始した場合，標準アプリケーション名が重複していると判断されます。

1. 標準アプリケーション名が「foo」で，標準モジュール名が「bar」のアプリケーションを開始
2. 標準アプリケーション名が「foo/bar」のアプリケーションを開始

2.4.4　Portable Global JNDI 名の登録抑止

09-00 以降のアプリケーションサーバで，Portable Global JNDI 名でのルックアップをしない場合，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に次の指定をします。こ

の指定をした場合，アプリケーションの開始時に Portable Global JNDI 名は登録されません。また，Portable Global JNDI 名の登録に関する情報や警告も表示されません。

- ejbserver.jndi.global.enabled パラメタに false を指定する。

2.4.5　CTM を使用する場合の Portable Global JNDI 名でのルックアップ

CTM を使用したシステム構成の場合，EJB クライアントアプリケーションから CTM デーモンに接続されている CORBA ネーミングサービスに対してルックアップすることで，EJB リモートホームオブジェクトを取得できます。EJB をルックアップする際には，HITACHI_EJB から始まる名称，Portable Global JNDI 名およびユーザ指定名前空間機能を使用して付与した別名が使用できます。

CTM を使用する場合の EJB のルックアップでは，EJB に別名が指定されているときは別名が使用されます。別名が指定されていないときは，デフォルトのルックアップ名が使用されます。デフォルトのルックアップ名は，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内の ejbserver.ctm.useGlobalJNDI パラメタで切り替えられます。

CTM を使用する場合の EJB のルックアップに使用される名称を次の表に示します。

表 2-15　CTM を使用する場合の EJB のルックアップに使用される名称

前提条件				ルックアップ名称
EJB の別名指定	ejbserver.jndi.global.enabled パラメタの値※1	ejbserver.ctm.useGlobalJNDI パラメタの値※2	Portable Global JNDI 名の登録可否※3	
あり	－	－	－	EJB の別名
なし	true（デフォルト）	true	登録できる	Portable Global JNDI 名
			登録できない	HITACHI_EJB から始まる名称
		false（デフォルト）	－	
	false	－	－	

（凡例）

－：該当なし

注※1

簡易構築定義ファイルの ejbserver.jndi.global.enabled パラメタの指定値を示します。true の場合，アプリケーションの開始時に Portable Global JNDI 名を登録します。false の場合，登録しません。

注※2

簡易構築定義ファイルの ejbserver.ctm.useGlobalJNDI パラメタの指定値を示します。true の場合，CTM を使用するシステム構成での EJB のルックアップに Portable Global JNDI 名を使用します。false の場合，HITACHI_EJB から始まる名称を使用します。

注※3

EJB を Portable Global JNDI 名で登録できるかどうかを示します。標準アプリケーション名や標準モジュール名が重複している場合や，登録できない文字列の場合は，Portable Global JNDI 名を登録できません。

2.4.6　リソース参照の名称

Java EE 6 以降，リソース参照の名称には，java:comp/env を使用した名称のほかに JNDI 完全修飾名を指定できます。リソース参照の名称に JNDI 完全修飾名を指定することで，一つのアプリケーション内のすべてのコンポーネント（Enterprise Bean，サーブレット，JSP，フィルタ，およびリソースアダプタ）で共有されるリソース参照や，EJB-JAR または Web アプリケーション内のすべてのコンポーネントで共有されるリソース参照を定義できます。

また，DD（web.xml）の<lookup-name>タグ，または@Resource アノテーションや@EJB アノテーションの lookup 属性を指定することで，ほかのリソース参照を JNDI 完全修飾名でルックアップできます。

ここでは，リソース参照の名称への JNDI 完全修飾名の指定，およびほかのリソース参照のルックアップについて説明します。

！ 注意事項

ejbserver.jndi.global.enabled パラメタに false を指定して Portable Global JNDI 名の登録抑止をしている場合は，リソース参照の名称に「java:」で始まる名称を指定しないでください。

(1)　リソース参照の名称への JNDI 完全修飾名の指定

アプリケーションサーバでは，Portable Global JNDI 名の登録が有効な場合に，リソース参照の名称に「java:」から始まる名称を指定すると，JNDI 完全修飾名が指定されたものと判断され，指定した名称がそのまま JNDI 名として扱われます。

Portable Global JNDI 名の登録が無効な場合，または「java:」以外の文字列から始まる名称を指定した場合は，java:comp/env から始まる名称またはリソースアダプタの別名として扱われます。

DD（web.xml）で，リソース参照の名称に JNDI 完全修飾名を指定できるリソース参照の種別を次の表に示します。

表 2-16　JNDI 完全修飾名を指定できるリソース参照の種別（DD）

項番	リソース参照の種別	リソース参照の名称を指定するタグ
1	環境エントリ	<env-entry> - <env-entry-name>
2	リソース参照	<resource-ref> - <res-ref-name>
3	リソース環境変数参照	<resource-env-ref> - <resource-env-ref-name>
4	リモート EJB 参照	<ejb-ref> - <ejb-ref-name>
5	ローカル EJB 参照	<ejb-local-ref> - <ejb-local-ref-name>
6	永続化コンテキストの参照	<persistence-context-ref> - <persistence-context-ref-name>
7	永続化ユニットの参照	<persistence-unit-ref> - <persistence-unit-ref-name>

また，name 属性に JNDI 完全修飾名を指定できるリソース参照のアノテーションを次の表に示します。

表 2-17　JNDI 完全修飾名を指定できるリソース参照の種別（アノテーション）

項番	リソース参照の種別	リソース参照の名称を指定するアノテーション
1	環境エントリ	@Resource の name 属性
2	リソース参照	
3	リソース環境変数参照	
4	リモート EJB 参照	@EJB の name 属性
5	ローカル EJB 参照	
6	永続化コンテキストの参照	@PersistenceContext の name 属性
7	永続化ユニットの参照	@PersistenceUnit の name 属性

Portable Global JNDI 名の登録が有効な場合に，リソース参照の名称に JNDI 完全修飾名を指定すると，指定する名前空間と，標準アプリケーション名や標準モジュール名が命名規則に反していないかの組み合わせによっては，アプリケーション開始時に KDJE47730-E が出力され，アプリケーションの開始に失敗することがあります。

指定する JNDI 完全修飾名の名前空間とアプリケーションの開始可否を次の表に示します。なお，「java:」から始まるが，指定できる名前空間ではない名称が指定された場合も，アプリケーション開始時に KDJE47730-E が出力され，アプリケーションの開始に失敗します。

表 2-18　指定する JNDI 完全修飾名の名前空間とアプリケーションの開始可否

前提条件		指定する名前空間			
リソース参照を定義したアプリケーションの標準アプリケーション名	リソース参照を定義した EJB-JAR または Web アプリケーションの標準モジュール名	java:comp	java:module	java:app	java:global
不正値または重複	－	○	×	×	○
正常	不正値または重複	○	×	○	○
	正常	○	○	○	○

（凡例）

○：アプリケーションを開始できる。

×：アプリケーションを開始できない。

－：該当なし。

アプリケーションサーバでは，リソース参照の名称を JNDI 完全修飾名で指定する場合，サブコンテキストの名称に制限はありません。ただし，すでに同名でコンテキストまたはオブジェクトが登録済みの場合は，アプリケーション開始時に KDJE47721-W が出力され，アプリケーションの開始に失敗します。

なお，リソース参照の名称に Portable Global JNDI 名を定義した場合でも，そのリソース参照を別プロセスの J2EE サーバや EJB クライアントアプリケーションからルックアップできません。Portable Global JNDI 名が定義された別プロセス上のリソース参照をルックアップした場合，NameNotFoundException 例外が発生します。

(2)　ほかのリソース参照のルックアップ

アプリケーションサーバでは，java:comp/env から始まる名称で登録されたリソース参照に加え，web.xml の<loolup-name>タグ，または@Resource アノテーションや@EJB アノテーションの lookup 属性を指定することで，EJB-JAR または Web アプリケーション内で共有されるリソースやアプリケーション内で共有されるリソースについて，DI やリソース参照を介してルックアップできます。

例えば，Web アプリケーション内の web.xml で「java:app」名前空間に定義した環境エントリは，別のモジュールである EJB-JAR 内の EJB からも DI できます。

リソース参照からルックアップできる名称には，リソースアダプタの名称，EJB の別名が使用できます。

リソース参照からルックアップできる名称を次の表に示します。

表 2-19　リソース参照からルックアップできる名称

ルックアップ元のリソース参照		ルックアップできる名称
アノテーション	DD（web.xml）	
@Resource の lookup 属性※1	<env-entry> - <lookup-name>※2	環境エントリの名称（JNDI 完全修飾名）
	<resource-ref> - <lookup-name>※1	リソース参照の名称（JNDI 完全修飾名）
		リソースアダプタの別名
	<resource-env-ref> - <lookup-name>※1	リソース環境変数参照の名称(JNDI 完全修飾名)
@EJB の lookup 属性	<ejb-ref> - <lookup-name>	EJB リモート参照の名称(JNDI 完全修飾名)
		EJB リモートオブジェクトの Portable Global JNDI 名
		EJB リモートオブジェクトの別名
	<ejb-local-ref> - <lookup-name>	ローカル EJB 参照の名称（JNDI 完全修飾名）
		ローカル EJB オブジェクトの Portable Global JNDI 名
		ローカル EJB オブジェクトの別名

注※1

Java EE の仕様に従い，注入先のデータ型または web.xml の<resource-ref-type>タグや<resource-env-ref-type>タグに設定されたデータ型が次のどれかの場合，@Resource アノテーションの lookup 属性または<lookup-name>タグの設定は無視されます（未設定時と同じ動作になります)。

org.omg.CORBA.ORB
org.omg.CORBA_2_3.ORB
javax.ejb.EJBContext
javax.ejb.SessionContext
javax.ejb.TimerService
javax.transaction.UserTransaction
javax.validation.Validator
javax.validation.ValidatorFactory
javax.enterprise.inject.spi.BeanManager

注※2

Java EE の仕様に従い，<env-entry-value>タグに値が設定されている場合，@Resource アノテーションの lookup 属性または<lookup-name>の設定は無視されます（未設定時と同じ動作となります）。

なお，属性ファイルの<linked-to>タグ，<linked-queue>タグ，<linked-adminobject>タグや，@Resource アノテーションの mappedName 属性が指定されている場合は，@Resource アノテーションの lookup 属性および web.xml の<lookup-name>タグの設定は無視されます。

! 注意事項

ルックアップ先のリソース参照をルックアップ元自身にしたり，循環参照にしたりしないでください。循環参照となるようなリソース参照が定義された場合，アプリケーションからルックアップしたとき，または DI が実行されたときに，無限再帰呼び出しとなり java.lang.StackOverflowError 例外が発生し，アプリケーションからの応答が返らなくなります。

循環参照となる例を次に示します。

```
<env-entry>
<env-entry-name>java:app/env/sample1</env-entry>
<lookup-name>java:app/env/sample2</env-entry>
</env-entry>
<env-entry>
<env-entry-name>java:app/env/sample2</env-entry>
<lookup-name>java:app/env/sample1</env-entry>
</env-entry>
```

この定義に対して，アプリケーションから「java:app/env/sample1」をルックアップすると，無限再帰呼び出しとなり，java.lang.StackOverflowError 例外が発生します。

2.4.7　アノテーションでの Portable Global JNDI 名の指定

アプリケーションサーバでは，Java EE の仕様に従い@EJB アノテーションの lookup 属性および mapped 属性に，Portable Global JNDI 名を指定できます。これによって，直接 EJB のビジネスインタフェースのリファレンス，またはホームオブジェクトのリファレンスを注入することができます。

@EJB アノテーションに次の条件に一致する指定をした場合，注入する Session Bean の種別に関係なく，ビジネスメソッドの呼び出し，またはタイムアウトコールバックメソッドの呼び出しのたびに DI が実施されます。

- beanName 属性に<module-name>/<bean-name>の形式で指定した場合
- lookup 属性に Portable Global JNDI 名を指定した場合
- mappedName 属性に Portable Global JNDI 名を指定した場合

2.4.8　DD での定義

Portable Global JNDI 名でのルックアップをする場合に必要な DD の定義について説明します。

Portable Global JNDI 名でのルックアップの定義は，application.xml，ejb-jar.xml，および web.xml に指定します。

Portable Global JNDI 名でのルックアップに関する DD での定義について次の表に示します。

表 2–20　DD での Portable Global JNDI 名でのルックアップの定義

DD の種類	タグ名	設定内容
application.xml	<application> - <application-name>	標準アプリケーション名を指定します。
ejb-jar.xml	<ejb-jar> - <module-name>	標準モジュール名を指定します。
web.xml	<web-app> - <module-name>	標準モジュール名を指定します。複数回指定された場合は，最初に指定されたタグの指定値が適用され，2 回目以降の値は無視されます。
	<web-app> - <env-entry> - <lookup-name>	ほかの env-entry を参照する Portable Global JNDI 名を指定します。
	<web-app> - <resource-ref> - <lookup-name>	ほかの resource-ref を参照する Portable Global JNDI 名を指定します。
	<web-app> - <resource-env-ref> - <lookup-name>	ほかの resource-env-ref を参照する Portable Global JNDI 名を指定します。
	<web-app> - <ejb-ref> - <lookup-name>	ほかの ejb-ref を参照する Portable Global JNDI 名，またはリモートビジネスインタフェースがバインドされた Portable Global JNDI 名を指定します。
	<web-app> - <ejb-local-ref> - <lookup-name>	ほかの ejb-local-ref を参照する Portable Global JNDI 名，またはローカルビジネスインタフェースがバインドされた Portable Global JNDI 名を指定します。

! 注意事項

cosminexus.xml やアプリケーション統合属性ファイルにある<module-name>タグおよび<lookup-name>タグと，上記の ejb-jar.xml または web.xml で指定する<module-name>タグおよび<lookup-name>タグは指定する内容が異なります。

2.4.9　実行環境での設定

Portable Global JNDI 名でルックアップをする場合，J2EE サーバの設定が必要です。

J2EE サーバの設定は，簡易構築定義ファイルで実施します。Portable Global JNDI 名でのルックアップの定義は，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に指定します。簡易構築定義ファイルでの設定を次の表に示します。

表 2-21　簡易構築定義ファイルでの Portable Global JNDI 名でルックアップをするための定義

項目	指定するパラメタ	設定内容
Portable Global JNDI 名の登録	ejbserver.jndi.global.enabled	アプリケーションの開始時に，ネーミングサービスに対して Portable Global JNDI 名を登録するかどうかを指定します。
CTM を使用する場合のデフォルトのルックアップ名	ejbserver.ctm.useGlobalJNDI	CTM を使用する場合に, EJB に別名を指定していないときに使用する，デフォルトのルックアップ名を指定します。

参考

Portable Global JNDI 名を登録する設定にした場合，アプリケーションの開始時に登録された名称の情報や標準アプリケーション名の重複などの警告がコンソールに表示されます。

簡易構築定義ファイルおよびパラメタについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.6　簡易構築定義ファイル」を参照してください。

2.5　HITACHI_EJB から始まる名称でのルックアップ

ここでは，HITACHI_EJB から始まる名称について説明します。J2EE アプリケーションをデプロイすると，EJB ホームオブジェクトリファレンスおよびビジネスインタフェースのリファレンスの JNDI の名前に，HITACHI_EJB から始まる名称が自動的にバインドされます。ルックアップ時には，バインドされた名前を使用します。

なお，JNDI 名前空間のマッピングの仕組みと使い方については，「2.3.2　JNDI 名前空間のマッピングとルックアップ」を参照してください。

! 注意事項

HITACHI_EJB から始まる名称は，ローカルインタフェース使用時にはバインドされません。ほかの方法でルックアップしてください。

(1)　EJB ホームオブジェクトリファレンスが自動的にバインドされる名称

J2EE アプリケーションを開始（デプロイ）したとき，Enterprise Bean の EJB ホームオブジェクトリファレンスは，次に示す名称で，JNDI の名前にバインドされます。

```
HITACHI_EJB/SERVERS/<サーバ名>/EJB/<J2EEアプリケーション名>/<Enterprise Bean名>
```

<サーバ名>
　J2EE サーバのサーバ名称

<J2EE アプリケーション名>
　J2EE アプリケーションのルックアップ名称

<Enterprise Bean 名>
　Enterprise Bean のルックアップ名称

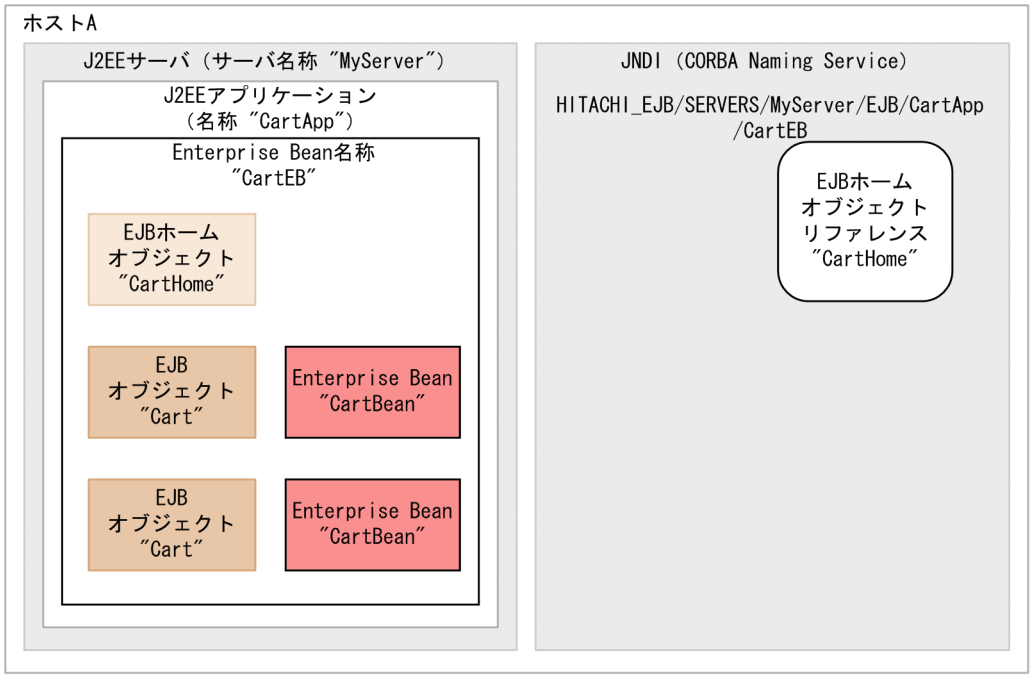
J2EE アプリケーション間の Enterprise Bean の呼び出しや，EJB クライアントアプリケーションからの Enterprise Bean の呼び出しのとき，クライアントはバインドされた JNDI の名前で EJB ホームオブジェクトリファレンスをルックアップします。

次に示す図では，下記の条件で J2EE アプリケーションを開始したとき，"CartHome"インタフェースを実装した EJB ホームオブジェクトが生成され，そのリファレンスが JNDI の名前"HITACHI_EJB/SERVERS/MyServer/EJB/CartApp/CartEB"にバインドされることを示しています。

条件

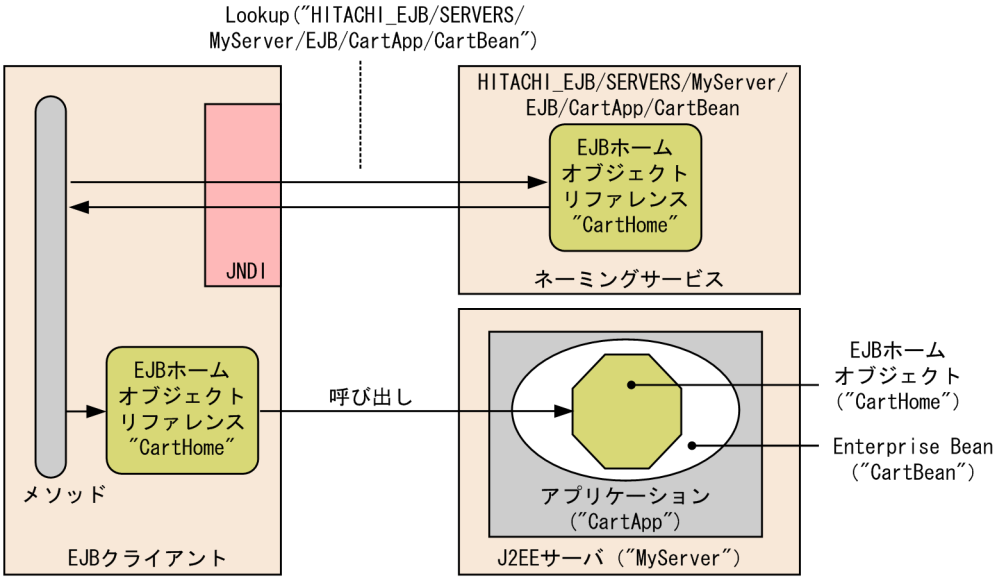
- Enterprise Bean："CartBean"
- リモートインタフェース名："Cart"
- ホームインタフェース："CartHome"
- サーバ名："MyServer"
- J2EE アプリケーションのルックアップ名："CartApp"
- Enterprise Bean のルックアップ名："CartEB"

図 2-5　EJB ホームオブジェクトリファレンスの JNDI 名前空間へのバインド



次に，HITACHI_EJB から始まる名称を利用して EJB ホームオブジェクトのリファレンスをルックアップする場合のルックアップとオブジェクトの取得の流れを示します。

図 2-6　HITACHI_EJB から始まる名称を利用したルックアップとオブジェクトの取得の流れ



(2)　ビジネスインタフェースのリファレンスが自動的にバインドされる名称

J2EE アプリケーションを開始（デプロイ）したとき，ビジネスインタフェースのリファレンスは，次に示す名称で JNDI の名前にバインドされます。

HITACHI_EJB/SERVERS/<サーバ名>/EJBBI/<J2EEアプリケーション名>/<Enterprise Bean名>

<サーバ名>
J2EE サーバのサーバ名称

<J2EE アプリケーション名>
J2EE アプリケーションのルックアップ名称

<Enterprise Bean 名>
Enterprise Bean のルックアップ名称

J2EE アプリケーション間の Enterprise Bean の呼び出しや，EJB クライアントアプリケーションからの Enterprise Bean の呼び出しのとき，クライアントはバインドされた JNDI の名前でビジネスインタフェースのリファレンスをルックアップします。

2.6　Enterprise Bean または J2EE リソースへの別名付与（ユーザ指定名前空間機能）

Enterprise Bean または J2EE リソースに対して，ユーザが別の名称を付けて JNDI 名前空間に登録する機能を，**ユーザ指定名前空間機能**といいます。この機能によって，Enterprise Bean または J2EE リソースを，設定した任意の名称でルックアップできるようになります。

なお，別名を付与するためには，サーバ管理コマンドの動作設定のカスタマイズが必要です。設定方法については，「2.6.7　実行環境での設定」を参照してください。

! 注意事項

Java EE の仕様では，java:comp/env を利用した名称でのルックアップが推奨されています。

この節の構成を次の表に示します。

表 2-22　この節の構成（ユーザ指定名前空間機能）

分類	タイトル	参照先
解説	別名を付けられる対象	2.6.1
	別名の付与規則	2.6.2
	別名が登録または削除されるタイミング	2.6.3
	クライアントからの検索	2.6.4
実装	Enterprise Bean の別名の設定	2.6.5
	J2EE リソースの別名の設定	2.6.6
設定	実行環境での設定	2.6.7
注意事項	ユーザ指定名前空間機能を使用する場合の注意事項	2.6.8

注　「運用」について，この機能固有の説明はありません。

2.6.1　別名を付けられる対象

ここでは，別名を付けられる対象について説明します。

別名は，Enterprise Bean または J2EE リソースに付けられます。

(1)　Enterprise Bean に対する別名

別名を付けられるのは，次のインタフェースを持つ Enterprise Bean です。

- リモートホームインタフェース
- ローカルホームインタフェース
- リモートビジネスインタフェース
- ローカルビジネスインタフェース

なお，以降，リモートホームインタフェースとリモートビジネスインタフェースをあわせて，**リモートインタフェース**といいます。また，ローカルホームインタフェースとローカルビジネスインタフェースをあわせて**ローカルインタフェース**といいます。

参考

リモートインタフェースとローカルインタフェースの別名は，異なる属性として設定します。詳細は，「2.6.5 Enterprise Bean の別名の設定」を参照してください。

(2)　J2EE リソースに対する別名

別名を付与できるのは，次の表に示す J2EE リソースです。

表 2-23　別名を付与できる J2EE リソース

J2EE リソースの種別		別名付与の可否
リソースアダプタ	DB Connector	○※1
	DB Connector for Reliable Messaging	○
	Reliable Messaging	○※2
	TP1 Connector	○
	TP1/Message Queue - Access	○※2
	そのほかのリソースアダプタ	○※3
メールコンフィグレーション		○
JavaBeans リソース		○

（凡例）　○：別名を付けられる

注※1　クラスタコネクションプール機能を使用する場合，メンバリソースアダプタには別名を付けられません。

注※2　属性ファイルの<resource-env-ref>に指定する javax.jms.ConnectionFactory オブジェクトに別名を付けられます。<resource-env-ref>に指定する javax.jms.Destination オブジェクトには別名を付けられません。

注※3　Connector 1.5 仕様に準拠するリソースアダプタの場合，管理対象オブジェクトに別名は付与できません。

2.6.2　別名の付与規則

ここでは，Enterprise Bean および J2EE リソースに別名を設定する場合の付与規則について説明します。

(1)　指定できる名称

別名に指定できる文字，および指定する場合の制約について説明します。

(a)　別名に指定できる文字

別名には，次の文字を使用した名称を指定できます。

- 英数字（A～Z，a～z，0～9）
- アンダースコア（_）
- スラッシュ（/）
- ピリオド（.）
- ハイフン（-）

ただし，スラッシュは，パスの区切り文字として使用する場合だけ，名前に使用できます。

(b)　別名を指定するときの制約

次に示す名前は，別名として指定できません。指定した場合，J2EE アプリケーションまたは J2EE リソースを開始できません。

- 名前の先頭または名前の末尾に，スラッシュ（/）またはピリオド（.）を指定した名前。
- スラッシュ（/）だけ，またはピリオド（.）だけの名前。
- スラッシュ（/）が連続している名前。
- スラッシュ（/）とピリオド（.）が連続している名前。
- 「HITACHI_EJB」で始まる名前（大文字・小文字を区別する）。
- 文字列長が 255 バイトよりも長い名前。

このほか，J2EE リソースの場合，同じ名称が指定されたときには，あとから指定されている定義が有効になります。

(2)　別名の重複

Enterprise Bean および J2EE リソースに別名を付ける場合の別名の重複可否を次の表に示します。

表 2-24　Enterprise Bean および J2EE リソースの別名の重複可否

別名を付ける対象	リモートインタフェースを持つ Enterprise Bean	ローカルインタフェースを持つ Enterprise Bean	J2EE アプリケーションに含まれるリソースアダプタ	J2EE リソース※1
リモートインタフェースを持つ Enterprise Bean	×	○※2	×	×
ローカルインタフェースを持つ Enterprise Bean	○※2	△	△	×
J2EE アプリケーションに含まれるリソースアダプタ	×	△	△	×
J2EE リソース※1	×	×	×	×

（凡例）

○：重複できる

△：J2EE アプリケーションが異なる場合は重複できる

×：重複できない

注※1　J2EE アプリケーションに含まれるリソースアダプタを除きます。

注※2　ローカル呼び出し最適化機能を使用するかどうかに関係なく重複できます。

なお，重複できない組み合わせの場合，名称が完全に一致していなくても指定できないことがあります。

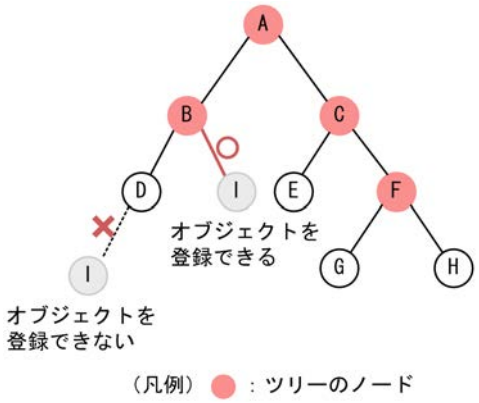
例を使用して説明します。

例

ネーミング管理機能では，CORBA ネーミングサービスに登録されるオブジェクトを名前で管理しています。オブジェクトに付与された名前は「/」を階層構造の区切りとして解析し，ツリー構造で管理されます。ツリーのなかで，子ノードを持つノードでは，そのノードの下に新しくオブジェクトを登録できます。

次の図の，ノード A，B，C，F は子ノードを持つノードです。このため，これらのノードの下には新しくオブジェクトを登録できます。

図 2-7　名称が完全に一致していなくても指定できない例



この状態で，別名「A/B/D」がすでに使用されている場合は，別名「A/B/I」を指定できます。

しかし，D，E，G，H のようなノードには，新しくオブジェクトを登録することはできません。このため，別名「A/B/D/I」を指定することはできません。

重複できない別名を指定した場合，別名を指定した対象ごとに，次のタイミングでエラーが発生します。

- Enterprise Bean の場合は，J2EE アプリケーションの開始に失敗します。
- メールコンフィグレーション以外の J2EE リソースの場合は，リソースの開始に失敗します。
- メールコンフィグレーションの場合は，属性設定時にエラーが発生します。

2.6.3　別名が登録または削除されるタイミング

ここでは，Enterprise Bean または J2EE リソースに対する別名の設定方法，別名が登録されるタイミング，および別名が削除されるタイミングについて説明します。

(1)　Enterprise Bean の別名が登録または削除されるタイミング

Enterprise Bean のオブジェクトに指定した別名が名前空間に登録されるのは，J2EE アプリケーションを開始した時，または J2EE サーバを起動した時です。

Enterprise Bean のオブジェクトに指定した別名が名前空間から削除されるのは，J2EE アプリケーションを停止した時，または J2EE サーバを停止した時です。

参考

ログレベルを「Warning」にしている場合，別名が登録または削除されたことをメッセージログで確認できます。

- 別名登録時：KDJE47605-I が出力されます。
- 別名削除時：KDJE47606-I が出力されます。

ただし，デフォルトのログレベルの設定では，これらのメッセージは出力されません。ログレベルの設定については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「3.3.6　J2EE サーバのログ取得の設定」を参照してください。

(2)　J2EE リソースの別名が登録または削除されるタイミング

J2EE リソースの別名が名前空間に登録されるのは，J2EE リソースを開始した時です。

J2EE リソースの別名が名前空間から削除されるのは，J2EE リソースを停止した時です。

参考

別名が登録または削除されたことは，メッセージログで確認できます。

- 別名登録時：KDJE47602-I が出力されます。
- 別名削除時：KDJE47603-I が出力されます。

2.6.4　クライアントからの検索

ここでは，別名を付けた Enterprise Bean または J2EE リソースをクライアントから検索する方法について説明します。

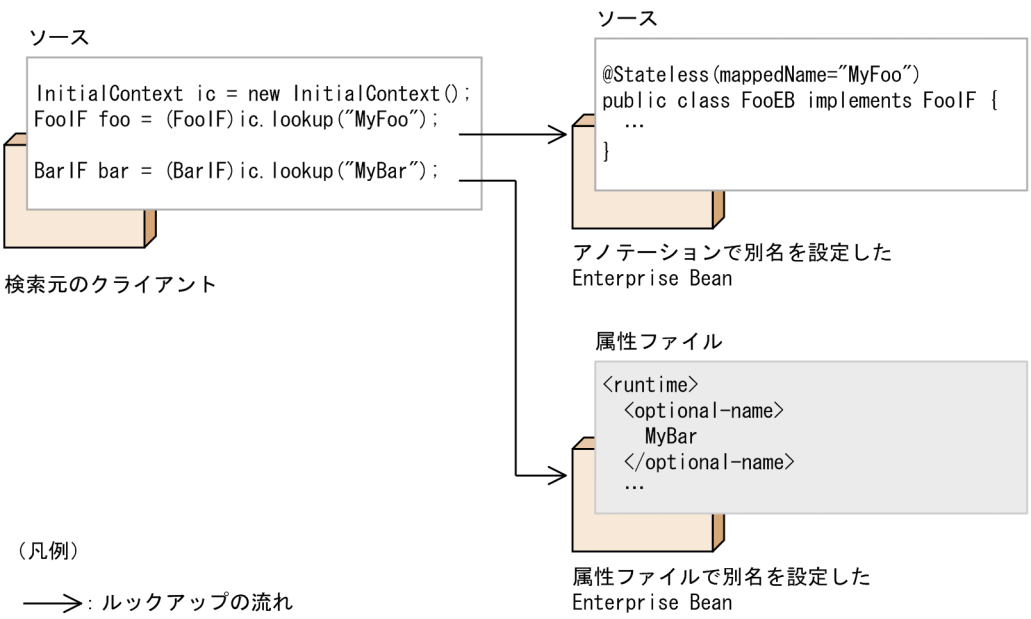
(1)　クライアントのソースと検索先オブジェクトの設定の関係

別名を付けた Enterprise Bean または J2EE リソースを検索する場合のクライアントのソースでの指定方法と，検索先オブジェクトでの設定方法について説明します。

クライアントのソースには，ルックアップする名称として，検索先のオブジェクトの別名を指定します。検索先のオブジェクトには，アノテーションまたは属性ファイルを使用して，対応する別名を設定します。

検索元のソースの記述と，検索先オブジェクトの設定の関係を次の図に示します。

図 2-8　クライアントのソースと検索先オブジェクトの設定の関係



(2)　Enterprise Bean の検索

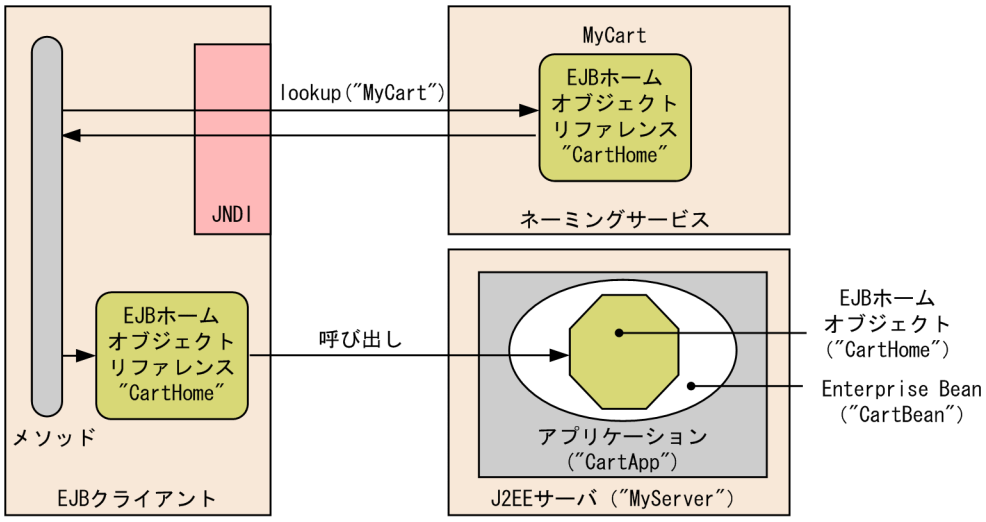
別名を使用して Enterprise Bean を検索する場合の，クライアントのコーディング例を次に示します。この例では，「MyCart」という別名を設定した EJB ホームオブジェクトを検索します。

```
   :
javax.naming.Context ctx = new javax.naming.InitialContext();
Object obj = ctx.lookup("MyCart");
SampleHome home =
(SampleHome)javax.rmi.PortableRemoteObject.narrow(obj, SampleHome.class);
Sample mybean = home.create();    //リモートオブジェクトの生成
String name = mybean.ping();      //ビジネスメソッドの実行
   :
```

EJB ホームオブジェクト取得後の操作は，別名を使用しないで取得した EJB ホームオブジェクトに対する操作と同じです。

EJB ホームオブジェクトのリファレンスをルックアップする場合の，ルックアップおよびオブジェクト取得の流れを次の図に示します。

図 2-9　EJB ホームオブジェクトのリファレンスのルックアップとオブジェクト取得の流れ



(3)　J2EE リソースの検索

別名を使用して J2EE リソースを検索する場合の，クライアントのコーディング例を次に示します。この例では，「jdbc/DBOpt」という別名を設定した J2EE リソースを検索します。

```
Context initCtx = new InitialContext();
DataSource ds = (DataSource) initCtx.lookup("jdbc/DBOpt");
```

なお，J2EE リソースのオブジェクト取得後の操作は，別名を使用しないで取得したオブジェクトに対する操作と同じです。

! 注意事項

EJB クライアントアプリケーションからは，J2EE リソースを検索できません。検索した場合は，javax.naming.NameNotFoundException 例外が発生します。

2.6.5　Enterprise Bean の別名の設定

ここでは，Enterprise Bean に別名を設定する方法について説明します。

Enterprise Bean の別名は，次の 2 種類の方法で設定できます。

- cosminexus.xml で設定する方法
- アノテーションで指定する方法

それぞれの方法について説明します。

なお，この機能を使用する場合は，サーバ管理コマンドの usrconf.properties で別名を使用するかどうかを指定しておく必要があります。設定方法については，「2.6.7　実行環境での設定」を参照してください。

(1)　cosminexus.xml で設定する方法

Enterprise Bean の別名を設定するには，cosminexus.xml の<ejb-jar>タグ内に指定します。cosminexus.xml での Enterprise Bean の別名の設定について次の表に示します。

表 2-25　cosminexus.xml での Enterprise Bean の別名を設定

項目	指定するタグ	設定内容
リモートインタフェース	<session>－<optional-name>タグ	Session Bean のリモートインタフェースの別名を指定します。
	<entity>－<optional-name>タグ	Entity Bean のリモートインタフェースの別名を指定します。
ローカルインタフェース	<session>－<local-optional-name>タグ	Session Bean のローカルインタフェースの別名を指定します。
	<entity>－<local-optional-name>タグ	Entity Bean のローカルインタフェースの別名を指定します。

指定するタグの詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「2.2　アプリケーション属性ファイル（cosminexus.xml）で指定する各属性の詳細」を参照してください。

参考

Enterprise Bean の別名は，アノテーションで設定することもできます。アノテーションによって別名が設定されている場合に，サーバ管理コマンドで異なる別名を設定すると，サーバ管理コマンドで設定した値が有効になります。

詳細は，「(3) サーバ管理コマンドとアノテーションの両方で別名を設定した場合」を参照してください。

(2)　アノテーションで指定する方法

@Stateless，@Stateful，または@Singleton の mappedName 属性で指定します。

アノテーションで別名を指定する場合のコーディング例を次に示します。この例は，@Stateless の mappedName 属性に Stateless Session Bean の別名「MyFoo」を設定する場合の例です。

```
@Stateless(mappedName="MyFoo")
public class FooEB implements FooIF {
    ...
}
```

```
public interface FooIF {
...
}
```

@Stateless，@Stateful，または@Singleton の mappedName 属性に指定した別名は，Session Bean 属性ファイルの<hitachi-session-bean-property><mapped-name>タグに設定されます。

(3)　サーバ管理コマンドとアノテーションの両方で別名を設定した場合

@Stateless，@Stateful，または@Singleton の mappedName 属性を指定して別名を指定している場合に，cosminexus.xml で<optional-name>タグまたは<local-optional-name>タグに別名を設定したときには，<optional-name>タグおよび<local-optional-name>タグの指定が有効になります。

サーバ管理コマンドとアノテーションの両方で別名を指定した場合に有効になるタグについて，次の表に示します。

表 2-26　サーバ管理コマンドとアノテーションの両方で別名を指定した場合に有効になるタグ

別名の設定対象	@Stateless，@Stateful，@Singleton の mappedName 属性だけを指定	<optional-name>タグ，<local-optional-name>タグだけを指定	@Stateless，@Stateful，@Singleton の mappedName 属性および<optional-name>タグ，<local-optional-name>タグの両方を指定した場合
リモートインタフェース	<mapped-name>タグ※	<optional-name>タグ	<optional-name>タグ
ローカルインタフェース	<mapped-name>タグ※	<local-optional-name>タグ	<local-optional-name>タグ

注※　@Stateless，@Stateful，@Singleton の mappedName 属性に指定した値が設定されるタグです。

2.6.6　J2EE リソースの別名の設定

ここでは，J2EE リソースに別名を設定する方法について説明します。

J2EE リソースの別名は，cosminexus.xml で設定できます。

なお，この機能を使用する場合は，サーバ管理コマンドの usrconf.properties で別名を使用するかどうかを指定しておく必要があります。設定方法については，「2.6.7　実行環境での設定」を参照してください。

J2EE リソースの別名を設定するには，cosminexus.xml の<rar>タグ内に指定します。cosminexus.xml での J2EE リソースの別名の設定について次の表に示します。

表 2-27　cosminexus.xml での J2EE リソースの別名を設定

項目	指定するタグ	設定内容
リソースアダプタ	<resource-external-property>－<optional-name>タグ	リソースアダプタの別名を指定します。

注　メールコンフィグレーションおよび JavaBeans リソースの別名は，属性ファイルを使用して設定します。設定方法については，「2.6.7　実行環境での設定」を参照してください。

指定するタグの詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「2.2　アプリケーション属性ファイル（cosminexus.xml）で指定する各属性の詳細」を参照してください。

2.6.7　実行環境での設定

ユーザ指定名前空間機能を使用するためには，サーバ管理コマンドのカスタマイズ，J2EE アプリケーションの設定が必要です。

(1)　サーバ管理コマンドのカスタマイズ

サーバ管理コマンドの動作設定をカスタマイズできます。ここでは，サーバ管理コマンドが使用するユーザ指定名前空間機能の設定について説明します。

サーバ管理コマンドのカスタマイズは，usrconf.properties（サーバ管理コマンド用システムプロパティファイル）で設定します。設定内容を次に示します。

表 2-28　ユーザ指定名前空間機能を使用するためのサーバ管理コマンドのカスタマイズ

項目	指定するキー	設定内容
EJB ホームオブジェクトリファレンスの別名付与（ユーザ指定名前空間機能）	ejbserver.cui.optionalname.enabled	EJB ホームオブジェクトの別名を指定するかどうかを指定します。 また，EJB ホームオブジェクトに別名を付与するためには，このキーの指定に加えて，EJB ホームオブジェクトの別名※を指定する必要があります。
J2EE リソースの別名付与（ユーザ指定名前空間機能）	ejbserver.cui.optionalname.enabled	J2EE リソースの別名を指定するかどうかを指定します。 J2EE リソースに別名を付与するためには，このキーの指定に加えて，J2EE リソースの別名※を指定する必要があります。

注※　サーバ管理コマンドを使用して J2EE アプリケーションのプロパティを定義するときに別名を指定します。JNDI 名前空間に登録される名称の参照と変更については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「9.13　JNDI 名前空間に登録される名称の参照と変更」を参照してください。

(2)　J2EE アプリケーションの設定

実行環境での J2EE アプリケーションの設定は，サーバ管理コマンドおよび属性ファイルで実施します。Enterprise Bean の別名の設定と J2EE リソースの設定に分けて説明します。なお，ここで説明するタグは，cosminexus.xml と対応しています。cosminexus.xml での定義については，「2.6.5　Enterprise Bean の別名の設定」または「2.6.6　J2EE リソースの別名の設定」を参照してください。

(a)　Enterprise Bean の別名の設定

Enterprise Bean に別名を付与するための設定には，Session Bean 属性ファイルまたは Entity Bean 属性ファイルを使用します。インタフェースの種別によってタグ名が異なります。インタフェースの種別ごとに使用する属性ファイルとタグを次に示します。

- リモートインタフェースの別名

  Session Bean の場合は Session Bean 属性ファイル，Entity Bean の場合は Entity Bean 属性ファイルの<optional-name>タグに指定します。

- ローカルインタフェースの別名

  Session Bean の場合は Session Bean 属性ファイル，Entity Bean の場合は Entity Bean 属性ファイルの<local-optional-name>タグに指定します。

属性ファイルでの別名の指定例を次に示します。この例は，SessionBean 属性ファイルを使用して，Stateful Session Bean に別名を設定する場合の例です。

```
<hitachi-session-bean-property>
  <display-name>MyAdder</display-name>
  <session-type>Stateful</session-type>
  <transaction-type>Container</transaction-type>
  <runtime>
    <lookup-name>MyAdder</lookup-name>
    <optional-name>user/Adder</optional-name>
    <local-optional-name>user/localAdder</local-optional-name>
    <maximum-sessions>0</maximum-sessions>
    <stateful>
      <maximum-active-sessions>0</maximum-active-sessions>
      <inactivity-timeout>0</inactivity-timeout>
      <removal-timeout>0</removal-timeout>
```

この例の場合は，リモートインタフェースの別名として「user/Adder」，ローカルインタフェースの別名として「user/localAdder」が設定されます。

(b)　J2EE リソースの別名の設定

J2EE リソースに別名を付与するための設定は，DB Connector，メールコンフィグレーション，または JavaBeans リソースの属性として設定します。設定に使用するコマンドおよび属性ファイルは，リソースの種別ごとに異なります。リソースの種別と別名を設定するコマンドおよび属性ファイルを次の表に示します。

表 2-29　リソースの種別と別名を設定するコマンドおよび属性ファイル

リソースの種別	コマンド	属性ファイル	指定するタグ
リソースアダプタ	cjsetrarprop コマンド	Connector 属性ファイル	<resource-external-property>－<optional-name>タグ
メールコンフィグレーション	cjsetresprop コマンド	メール属性ファイル	
JavaBeans リソース	cjsetjbprop コマンド	JavaBeans リソース属性ファイル	<resource-env-external-property>－<optional-name>タグ

注　メールコンフィグレーションおよび JavaBeans リソースについては，cosminexus.xml では設定できません。属性ファイルで設定してください。

属性ファイルでの別名の指定例を次に示します。この例は，Connector 属性ファイルを使用して，リソースアダプタに別名を設定する場合の例です。

```
<hitachi-connector-property>
  <description></description>
  <display-name>DB_Connector_for_Oracle</display-name>
  <icon />
  <vendor-name>Hitachi, Ltd.</vendor-name>
  :
  <connector-runtime>
    <resource-external-property>
    <description></description>
    <optional-name>jdbc/TestDB1</optional-name>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
    </resource-external-property>
  </connector-runtime>
</hitachi-connector-property>
```

この例の場合は，DB Connector の別名として「jdbc/TestDB1」が設定されます。

2.6.8　ユーザ指定名前空間機能を使用する場合の注意事項

ここでは，ユーザ指定名前空間機能を使用する場合の注意事項について説明します。

(1)　別名を使用した検索を実行する場合の注意

- ローカルインタフェースを持つ Enterprise Bean は，J2EE アプリケーション外から検索できません。
- ローカルインタフェースを持つ Enterprise Bean の別名に対して，ネーミングコンテキスト単位の検索はできません。
- リモートインタフェースを持つ Enterprise Bean とローカルインタフェースを持つ Enterprise Bean には，同じ別名を設定できます。ただし，この場合に重複している別名をルックアップすると，必ずローカルインタフェースを持つ Enterprise Bean がルックアップされます。

(2)　ネーミングサービスについての注意

- 複数の J2EE サーバで，一つの CORBA ネーミングサービスを共用している場合，ユーザ指定名前空間機能は使用できません。
- J2EE サーバと CORBA ネーミングサービスは同じタイミングで起動・停止するようにしてください。

  J2EE サーバまたは CORBA ネーミングサービスのどちらか一方がダウンした場合は，J2EE サーバと CORBA ネーミングサービスを共に再起動してください。
- CORBA ネーミングサービスを共有する場合，ユーザ指定名前空間機能で指定する別名として「Cosminexus」は使用できません。

(3)　J2EE リソースの別名を指定する場合の注意

- 別名を登録した J2EE リソースを停止，削除，または属性変更（JavaMail セッションの場合）する場合は，J2EE サーバ上で開始されているすべての J2EE アプリケーションを先に停止してください。
- J2EE リソースのユーザ指定名前空間機能を使用する場合，J2EE アプリケーションで生成する InitialContext で指定しているプロバイダ URL（java.naming.provider.url）のホスト名と，J2EE サーバ側のサーバ定義に指定する ejbserver.naming.host キーの値は同じ文字列にしてください。なお，次の条件に当てはまる場合は，J2EE アプリケーションで生成している InitialContext に対して，プロバイダ URL を指定する必要はありません。
  - J2EE サーバ用ユーザプロパティファイルの ejbserver.naming.host キーに「localhost」（デフォルト値）を指定する。
  - 同一 J2EE サーバ上のネーミングサービスへ接続する。

  ejbserver.naming.host キーに「localhost」を指定していて，J2EE アプリケーションで生成している InitialContext のプロバイダ URL を指定する場合は，プロバイダ URL のホスト名に次の API で取得できる値を指定してください。java.net.InetAddress.getLocalHost().getHostName();
- J2EE リソースのユーザ指定名前空間機能では，別名の中に"/"文字を含めた場合も，ネーミングコンテキスト単位で検索できません。ルックアップ名称として使用できる文字列は，指定した別名だけです。

  別名として"jdbc/TestDB"を付与してデプロイおよび開始している J2EE リソース（DB Connector）の場合に，使用できる JNDI のルックアップ名称と使用できないルックアップ名の例を次に示します。

  使用できるルックアップ名の例

  ```
  DataSource ds = (DataSource) initCtx.lookup("jdbc/TestDB");
  ```

  使用できないルックアップ名の例

  ```
  Context ctx = (Context) initCtx.lookup("jdbc");
  ```

2.7　ラウンドロビンポリシーによる CORBA ネーミングサービスの検索

複数の CORBA ネーミングサービスに登録されている同一名称（別名）の EJB ホームオブジェクトまたはビジネスインタフェースのリファレンスを，ラウンドロビンポリシーに従ってルックアップできます。これを，**ラウンドロビン検索**といいます。

JNDI の名前空間から該当する名前で EJB ホームオブジェクトまたはビジネスインタフェースのリファレンスをラウンドロビン検索した場合，クライアントアプリケーションは複数の CORBA ネーミングサービス上に存在する EJB ホームオブジェクトまたはビジネスインタフェースのリファレンスから，ラウンドロビンのポリシーで選択された EJB ホームオブジェクトまたはビジネスインタフェースのリファレンスを取得できます。これによって，J2EE サーバをクラスタ構成で開始して，負荷を分散できます。また，EJB クライアントからは，クラスタ構成を意識しないで，J2EE サーバの Enterprise Bean を呼び出すことができます。

この節の構成を次の表に示します。

表 2-30　この節の構成（ラウンドロビンポリシーによる CORBA ネーミングサービスの検索）

分類	タイトル	参照先
解説	ラウンドロビン検索の範囲	2.7.1
	ラウンドロビン検索の動作	2.7.2
設定	ラウンドロビン検索をするために必要な設定	2.7.3
	ラウンドロビン検索機能を使用する場合の推奨する設定	2.7.4
注意事項	ラウンドロビン検索をする場合の注意事項	2.7.5

注　「実装」および「運用」について，この機能固有の説明はありません。

ポイント

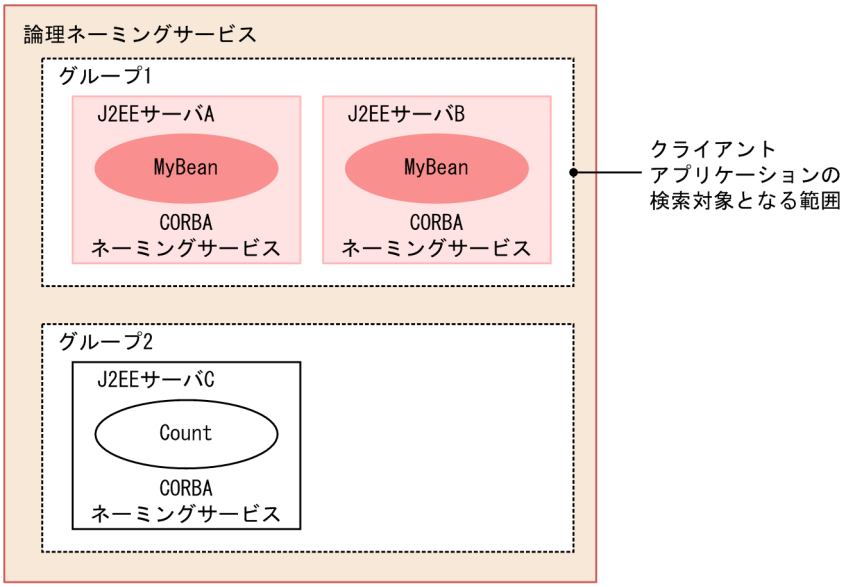
ラウンドロビン検索をする場合，検索対象となる EJB ホームオブジェクトまたはビジネスインタフェースのリファレンスは同じ名称で CORBA ネーミングサービスに登録されている必要があります。このため，ユーザ指定名前空間機能を使用して，それぞれの EJB ホームオブジェクトまたはビジネスインタフェースのリファレンスに同じ名称を指定してください。

2.7.1　ラウンドロビン検索の範囲

JNDI によるラウンドロビン検索の対象範囲は，**論理ネーミングサービス**内のグループとなります。

論理ネーミングサービスは一つ以上のグループで構成されます。1 グループにつき一つ以上の CORBA ネーミングサービスを含みます。論理ネーミングサービスの構成を次の図に示します。

図 2-10　論理ネーミングサービスの構成



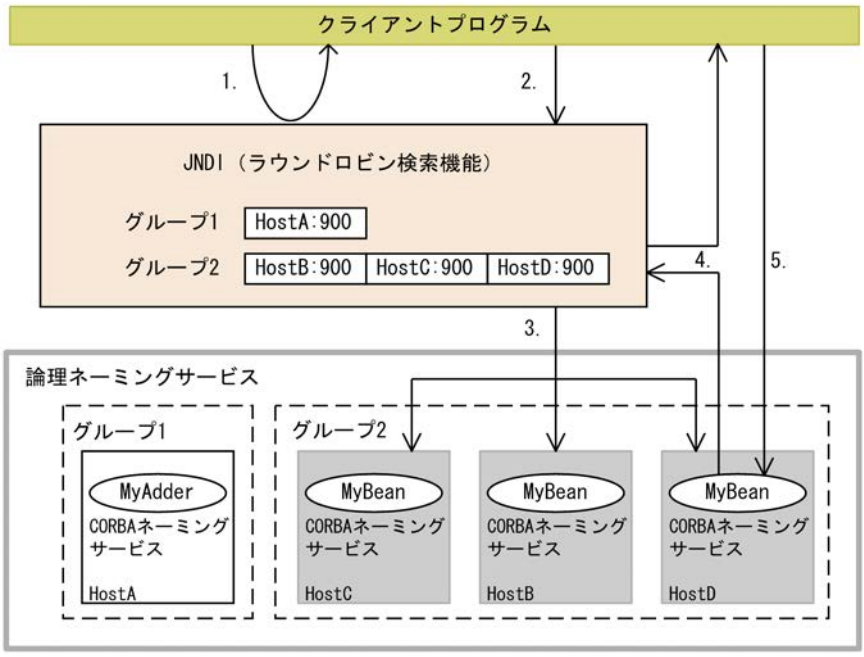
複数のグループで，グループのメンバに同一の CORBA ネーミングサービスを指定できます。

2.7.2　ラウンドロビン検索の動作

クライアントプログラムからのラウンドロビン検索時の動作について次に示す図で説明します。

なお，図中の各 CORBA ネーミングサービスには，ラウンドロビン検索の対象となる EJB ホームオブジェクトが「MyBean」という名前で登録されていると仮定します。検索対象となる複数の EJB ホームオブジェクトには，ユーザ指定名前空間機能によって，同一の名称が設定されています。

図 2-11　ラウンドロビン検索時の動作



上記の図について説明します。

1. クライアントアプリケーションは，検索対象の論理ネーミングサービスのグループ名を引数にして，ラウンドロビン用の InitialContext インスタンスを生成します。
2. クライアントアプリケーションは，生成したラウンドロビン用の InitialContext インスタンスに，「MyBean」という名前を指定して検索を要求します。
3. ラウンドロビン検索機能は，ラウンドロビンポリシーで検索対象のグループに属する CORBA ネーミングサービスの検索先を決定します。
4. 検索に成功すると，クライアントアプリケーションに結果を返します。
   なお，検索に失敗した場合は，グループに属する別の CORBA ネーミングサービスを検索します。
5. クライアントアプリケーションは，検索された EJB ホームオブジェクトにアクセスします。

! 注意事項

InitialContext の初回発行時は，すべてのグループのプロバイダ URL に接続を試みます。このため，接続できないプロバイダ URL の記載がある場合，遅延などが発生することがあります。

2.7.3　ラウンドロビン検索をするために必要な設定

ラウンドロビン検索をするために必要な設定とグループ名の命名規則などについて説明します。

ラウンドロビン検索をする場合，次の設定が必要です。

- ラウンドロビン検索の対象になる論理ネーミングサービスのグループ
- 各グループに属する CORBA ネーミングサービスのルート位置
- InitialContextFactory の実装を委譲しているクラス

ラウンドロビン検索機能を使用するためには，システムプロパティに設定します。なお，InitialContextFactory の実装を委譲しているクラスについては，システムプロパティの設定に加えて，各アプリケーションの InitialContext 生成時に引数で指定することもできます。また，InitialContext 生成時の引数では，システムプロパティに指定した論理ネーミングサービスのグループのうち，特定のネーミングサービスを選択して指定することもできます。

なお，システムプロパティだけで設定している場合は，特定のグループを指定したラウンドロビン検索はできません。論理ネーミングサービス上にある全グループのネーミングサービスが検索の対象となります。

ここでは，それぞれの設定方法の概要について説明します。

(1)　システムプロパティの設定によるグループとグループに属する CORBA ネーミングサービスのルート位置の特定

ラウンドロビン検索を実行する場合，システムプロパティに，ラウンドロビン検索の対象になる論理ネーミングサービスのグループと，グループに属するネーミングサービスのルート位置を指定します。また，InitialContextFactory の実装を委譲しているクラスとして，java.naming.factory.initial=com.hitachi.software.ejb.jndi.GroupContextFactory を指定する必要があります。

システムプロパティは，ラウンドロビン検索機能を利用するアプリケーションの種類ごとに，次の個所に設定方法が異なります。

(2)　J2EE サーバで動作する J2EE アプリケーション（Enterprise Bean またはサーブレット）の場合

J2EE サーバのプロパティをカスタマイズして設定します。設定は，簡易構築定義ファイルの，J2EE サーバ用のユーザプロパティに設定します。

ラウンドロビン検索をするための定義は，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に指定します。簡易構築定義ファイルでのラウンドロビン検索をするための定義について次の表に示します。

表 2-31　簡易構築定義ファイルでのラウンドロビン検索をするための定義

指定するパラメタ	設定内容
ejbserver.jndi.namingservice.group.list	CORBA ネーミングサービスのグループを指定します。
ejbserver.jndi.namingservice.group.<Specify group name>.providerurls	各グループに属する，CORBA ネーミングサービスのルート位置を指定します。
java.naming.factory.initial	InitialContextFactory の実装をデレゲートしているクラスを指定します。

なお，ラウンドロビン検索は，ユーザ指定名前空間機能を使用していることが前提になります。ユーザ指定名前空間機能を使用する場合，サーバ管理コマンドの動作設定のカスタマイズと，J2EE アプリケーションのプロパティの定義が必要です。設定方法については，「2.6.7　実行環境での設定」および「2.6.5 Enterprise Bean の別名の設定」を参照してください。

(3)　J2EE サーバ以外で動作する EJB クライアントアプリケーションの場合

次のどちらかの方法で設定します。

- EJB クライアントアプリケーション起動時にプロパティとして設定する。
- System.setProperty メソッドを使用してアプリケーション内で設定する。

なお，EJB クライアントアプリケーションのプロパティの設定方法は，EJB クライアントアプリケーションの開始に使用するコマンドによって，EJB クライアントアプリケーションのプロパティの設定方法が異なります。EJB クライアントアプリケーションのプロパティの設定方法および指定例を説明します。

(a)　EJB クライアントアプリケーションのプロパティの設定方法

プロパティの設定方法は，cjclstartap コマンドを使用する場合と vbj コマンドを使用する場合で異なります。

- cjclstartap コマンドの場合

  cjclstartap コマンドを使用する場合は，EJB クライアントアプリケーションのプロパティファイル（usrconf.properties）でプロパティを設定します。なお，指定できるプロパティについては，「(2) J2EE サーバで動作する J2EE アプリケーション（Enterprise Bean またはサーブレット）の場合」を参照してください。

- vbj コマンドの場合

  vbj コマンドを使用する場合は，バッチファイル/シェルスクリプトファイル，またはコマンドの引数で，プロパティを設定します。

(b)　プロパティの指定例

指定例を次に示します。この例は，プロパティを usrconf.properties で指定する場合の指定例です。なお，それぞれのキーの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「14.3　usrconf.properties（Java アプリケーション用ユーザプロパティファイル)」を参照してください。

```
# 論理ネーミングサービスの構成を定義
ejbserver.jndi.namingservice.group.list=g1;g2;g3
ejbserver.jndi.namingservice.group.g1.providerurls=corbaname::hostA:900;corbaname::hostB:900
ejbserver.jndi.namingservice.group.g2.providerurls=corbaname::hostD:700;corbaname::hostE:700
ejbserver.jndi.namingservice.group.g3.providerurls=corbaname::hostF:800;corbaname::hostG:800
# InitialContextFactoryの実装を委譲するクラスを指定
java.naming.factory.initial=com.hitachi.software.ejb.jndi.GroupContextFactory
     :
```

指定例の中の ejbserver.jndi.namingservice.group.list キー，ejbserver.jndi.namingservice.group.<Specify group name>.providerurls キー，および java.naming.factory.initial キーには，それぞれ次の内容を指定します。

ejbserver.jndi.namingservice.group.list キー

ラウンドロビン検索をする場合に，検索対象になる論理ネーミングサービスのグループを定義します。指定する各グループ名は，論理ネーミング内で一意に識別できる名称にします。

ejbserver.jndi.namingservice.group.<Specify group name>.providerurls キー

各グループに属するネーミングサービスのルート位置をプロバイダ URL で指定します。<Specify group name>には，ejbserver.jndi.namingservice.group.list 内に指定したグループ名を指定します。

java.naming.factory.initial キー

InitialContextFactory の実装を委譲しているクラスを指定します。

java.naming.factory.initial キーに，"com.hitachi.software.ejb.jndi.GroupContextFactory"を指定した場合，ラウンドロビン検索が実施されます。指定しなかった場合，J2EE サーバがネーミングサービスとして利用する CORBA ネーミングサービスが検索の対象になります。

なお，「(4)　InitialContext 生成時の引数指定による検索対象グループの選択」に示す方法で InitialContext 生成時の引数に java.naming.factory.initial キーを設定する場合，システムプロパティでこのキーに値を指定する必要はありません。

(4)　InitialContext 生成時の引数指定による検索対象グループの選択

ラウンドロビン検索を実行する設定になっている場合に，クライアントアプリケーション内で InitialContext 生成時の引数に特定のグループを指定することによって，ラウンドロビン対象でのルックアップ対象になるグループを選択できます。なお，InitialContext 生成時の引数の指定は任意です。

指定例を次に示します。

```
          :
Hashtable env = new Hashtable();
env.put("ejbserver.jndi.namingservice.groupname", "g1");
env.put("java.naming.factory.initial",
                "com.hitachi.software.ejb.jndi.GroupContextFactory");
InitialContext ic = new InitialContext(env);
          :
```

指定例の中の ejbserver.jndi.namingservice.groupname キーおよび java.naming.factory.initial キーには，それぞれ次の内容を指定します。

ejbserver.jndi.namingservice.groupname キー

検索対象となるグループ名を「*gl*」部分に指定します。グループ名は，システムプロパティ（usrconf.properties の ejbserver.jndi.namingservice.group.list キー）で，すでに定義されているものを指定してください。なお，ejbserver.jndi.namingservice.groupname キーにデフォルト値はありません。指定しなかった場合，システムプロパティで設定した，すべてのグループを検索の対象とします。

java.naming.factory.initial キー

InitialContextFactory の実装を委譲しているクラスを指定します。java.naming.factory.initial キーに，"com.hitachi.software.ejb.jndi.GroupContextFactory"を指定した場合，ラウンドロビン検索が実施されます。システムプロパティの java.naming.factory.initial キーの指定を省略した場合に，引数でこのキーの指定を省略すると，ejbserver.jndi.namingservice.groupname キーで指定したグループの検索は実施されないで，J2EE サーバがネーミングサービスとして利用する CORBA ネーミングサービスが検索の対象になります。

(5)　グループ名の命名規則

グループ名に使用できる文字を次に示します。

- 英数字（A～Z，a～z，0～9）
- アンダースコア（_）

なお，グループ名は，論理ネーミングサービス内で一意となる名称にしてください。

(6)　プロパティでの設定内容

EJB クライアントアプリケーションのプロパティは，Java アプリケーションから Enterprise Bean を呼び出す場合に使用されます。J2EE アプリケーションの内容に応じて，必要なプロパティを設定してください。設定できるプロパティについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」を参照してください。

プロパティで設定できる内容の一例

- **EJB クライアントアプリケーションのログの設定**

  ejbserver.client.log で始まるキーや ejbserver.logger で始まるキーなどで，システムが出力するシステムログと，EJB クライアントアプリケーションが出力するユーザログの出力先やログレベルなどを変更できます。システムログについては，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「3.8　EJB クライアントアプリケーションのシステムログ出力」，ユーザログについては，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「9.　アプリケーションのユーザログ出力」を参照してください。

- **EJB クライアントアプリケーションのトランザクションの設定**

  ejbserver.client.transaction で始まるキーで，EJB クライアントアプリケーションでトランザクションを使用するかどうか，トランザクションサービスが使用するクライアント名などを指定できます。なお，Client を使用して EJB クライアント環境を構築する場合は，EJB クライアントアプリケーションのトランザクションは使用できません。詳細については，「3.20　EJB クライアントアプリケーションでトランザクションを開始する場合の注意事項」を参照してください。

- **EJB のリモートインタフェースでの通信障害発生時の EJB クライアントの動作**

  ejbserver.container.rebindpolicy キーに，EJB クライアント側でのコネクションの再接続動作とリクエストの再送動作を指定できます。

- **EJB クライアントアプリケーションからラウンドロビン検索を実行する設定**

ejbserver.jndi.namingservice.group.list キー，ejbserver.jndi.namingservice.group.<Specify group name>.providerurls キーおよび java.naming.factory.initial キーに，CORBA ネーミングサービスのグループ，各グループに属する CORBA ネーミングサービスのルート位置，および InitialContextFactory の実装をデレゲートしているクラスを指定できます。なお，ラウンドロビン検索は，J2EE サーバのサーバ管理コマンドのカスタマイズでユーザ指定名前空間機能を使用する設定をしている場合に有効になります。

- EJB クライアントアプリケーションから CTM へのリクエストの優先度の設定

  ejbserver.client.ctm.RequestPriority キーに，EJB クライアントアプリケーションから CTM に送信するリクエストの優先度を設定できます。

(7)　コマンドによって指定の要否が異なるプロパティ（EJB クライアントアプリケーションの場合）

ここでは，EJB クライアントアプリケーションのコマンド（vbj コマンド）によって指定の要否が異なるプロパティについて説明します。EJB クライアントアプリケーションのコマンドによって指定の要否が異なるプロパティのキーを次の表に示します。

表 2-32　EJB クライアントアプリケーションのコマンドによって指定の要否が異なるプロパティのキー

プロパティのキー	種別	コマンド	
		cjclstartap	vbj
org.omg.CORBA.ORBClass	固定	－	－
org.omg.CORBA.ORBSingletonClass	固定	－	－
javax.rmi.CORBA.UtilClass	固定	－	○
javax.rmi.CORBA.StubClass	固定	－	－
javax.rmi.CORBA.PortableRemoteObjectClass	固定	－	○
java.endorsed.dirs	可変	－	－

（凡例）

固定：該当するキーに対する値は固定で，指定する必要がある

可変：システムの実行環境に従って値を指定する必要がある

○：コマンドにキーを指定する必要がある

－：コマンドにキーを指定する必要がない

2.7.4　ラウンドロビン検索機能を使用する場合の推奨する設定

ラウンドロビン検索機能を使用するときには，ネーミングサービスの障害検知機能をあわせて使用することを推奨します。

ネーミングサービスの障害検知機能と組み合わせた設定例については，「2.9.4　実行環境の設定（障害検知機能を使用する場合）」を参照してください。

2.7.5　ラウンドロビン検索をする場合の注意事項

ここでは，ラウンドロビン検索をする場合の注意事項について説明します。

- ラウンドロビン検索用に取得したコンテキストでは，lookup メソッドだけをサポートしています。javax.naming.Context で定義されているほかの API は使用できません。
- 別名を使用したラウンドロビン検索を実行すると，次の順序で Enterprise Bean が検索されます。
  1. 検索を実行した J2EE サーバの名前空間から，ローカルインタフェースを持つ Enterprise Bean が検索されます。
  2. 1.で見つからなかった場合は，ラウンドロビン検索によってリモートインタフェースを持つ Enterprise Bean が検索されます。

InitialContextFactory の処理を委譲しているクラスごとの検索可否を次の表に示します。

表 2-33　InitialContextFactory の処理を委譲しているクラスごとの検索可否

検索対象	GroupContextFactory	InsContextFactory
リモートインタフェースを持つ Enterprise Bean	○※	○
リモートインタフェースを持つ Enterprise Bean（別名による検索）	○	○
ローカルインタフェースを持つ Enterprise Bean	○※	○
ローカルインタフェースを持つ Enterprise Bean（別名による検索）	○	○
J2EE リソース	○※	○
J2EE リソース（別名による検索）	×	○

（凡例）　○：検索できる　×：検索できない

注※　java:comp/env でのルックアップの場合，ラウンドロビン検索はしません。ルックアップを実行した自身の J2EE サーバだけから検索されます。

InitialContextFactory の実装を委譲しているクラスは，次の方法で指定します。両方指定した場合は引数で指定した方が有効になります。

- usrconf.properties の java.naming.factory.initial キーに指定する。
- InitialContext を生成するときの引数(Hashtable)で，java.naming.factory.initial キーに指定する。

2.8　ネーミング管理機能でのキャッシング

J2EE サービスのネーミング管理機能には，キャッシング機能があります。キャッシング機能とは，JNDI を介して EJB ホームオブジェクトリファレンスを検索した場合に，該当オブジェクトをキャッシュに一時的に保存し，以降，同一のオブジェクトを検索するときには，キャッシュに保存されたオブジェクトを返す機能です。

この節では，キャッシングの流れ，およびキャッシュ領域のクリアについて説明します。

なお，ネーミング管理機能でのキャッシングをするための設定は，J2EE サーバまたは EJB クライアントアプリケーションのプロパティとして設定します。

この節の構成を次の表に示します。

表 2-34　この節の構成（ネーミング管理機能でのキャッシング）

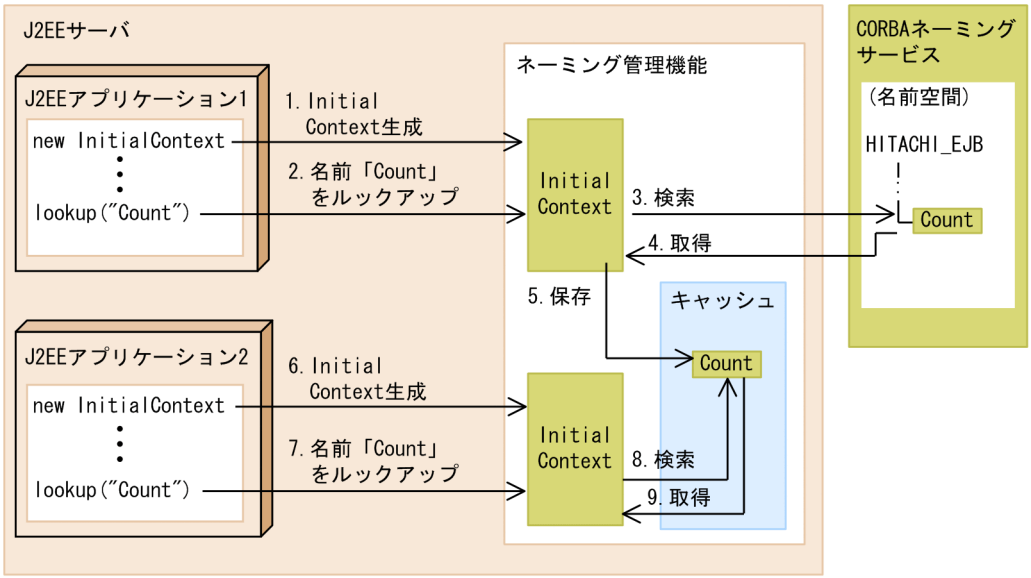
分類	タイトル	参照先
解説	キャッシングの流れ	2.8.1
	ネーミングで使用したキャッシュのクリア	2.8.2
設定	キャッシング機能を使用するための設定	2.8.3
注意事項	ネーミングでのキャッシングの注意事項	2.8.4

注　「実装」および「運用」について，この機能固有の説明はありません。

2.8.1　キャッシングの流れ

次の図に示すような流れで，ネーミングはキャッシングされます。

図 2-12　ネーミングのキャッシングの流れ



キャッシングの流れを説明します。この流れでは，同じ J2EE サーバ上の二つの J2EE アプリケーションから，同じ名前「Count」で EJB ホームオブジェクトのリファレンスをルックアップします。1.～5.が J2EE アプリケーション 1 から実行される処理，6.～9.が J2EE アプリケーション 2 から実行される処理です。

1. J2EE アプリケーション 1 が，JNDI の javax.naming.InitialContext クラスのインスタンスを生成します。
2. J2EE アプリケーション 1 が，javax.naming.InitialContext クラスのインスタンスに対して，EJB ホームオブジェクトリファレンスの検索（ルックアップ）を要求します。このとき，名前に「Count」を指定します。
3. 要求を受けたネーミング管理機能が，CORBA ネーミングサービスの名前空間から EJB ホームオブジェクトリファレンスを検索します。
4. ネーミング管理機能は，検索結果として，EJB ホームオブジェクトのリファレンスを取得します。
5. ネーミング管理機能が，取得した EJB ホームオブジェクトのリファレンスをキャッシュに保存します。
6. 同じプロセス上にある J2EE アプリケーション 2 が，JNDI の javax.naming.InitialContext クラスのインスタンスを生成します。
7. J2EE アプリケーション 2 が，javax.naming.InitialContext クラスのインスタンスに対して，EJB ホームオブジェクトリファレンスの検索（ルックアップ）を要求します。このとき，名前に，2.で指定した名前と同じ名前「Count」を指定します。
8. 要求を受けたネーミング管理機能が，キャッシュから，EJB ホームオブジェクトリファレンスを検索します。
9. ネーミング管理機能が，検索結果として，キャッシュから EJB ホームオブジェクトのリファレンスを取得します。

2.8.2　ネーミングで使用したキャッシュのクリア

ネーミングで使用しているキャッシュは，クリアできます。ただし，クリアするキャッシュサイズは設定できません。ここでは，キャッシュクリアが行われるタイミングと，クリアされる範囲について説明します。

(1)　キャッシュクリアのタイミング

キャッシュの内容は次のどちらかのタイミングでクリアされます。

- JNDI および RMI-IIOP の API で例外が発生した場合，キャッシュのクリアを強制的に実施します。
- システムプロパティに指定した値の間隔でキャッシュをクリアします（デフォルト値は 0 秒で，キャッシュはクリアされません）。

(2)　キャッシュクリアの範囲

ネーミングサービスでのキャッシュクリアの範囲について説明します。

キャッシュクリアの範囲は 2 とおりあります。

1. キャッシュ領域をすべてクリアします。
2. 無効なキャッシュ領域だけをクリアします。

1.の場合，キャッシュの全領域をクリアします。一方，2.の場合は，定期的に，キャッシュに保存されたオブジェクトが有効かどうかを確認し，無効なオブジェクトだけをキャッシュからクリアします。また，2.の場合，キャッシュクリアと同じタイミングで，一度検索した CORBA ネーミングサービスの状態を監視します。これによって，一度検索された CORBA ネーミングサービスの場合，CORBA ネーミングサービスがダウンしたあとは，該当する CORBA ネーミングサービスの検索を実施しません。また，CORBA ネーミングサービスが再起動すると，該当する CORBA ネーミングサービスの検索を自動的に開始します。

キャッシュクリアの設定方法は，「2.8.3　キャッシング機能を使用するための設定」を参照してください。

また，障害検知機能を使用する場合は，「2.9.4　実行環境の設定（障害検知機能を使用する場合）」を参照してください。

参考

EJB ホームオブジェクトの再接続機能を使用している場合，J2EE サーバを再起動したあとでも，キャッシュは無効になりません。

このため，無効なキャッシュ領域だけをクリアする指定をしているときでも，CORBA ネーミングサービスの EJB ホームオブジェクトのオブジェクトリファレンスは，キャッシュ領域から削除されません。

EJB ホームオブジェクトのリファレンスの検索（ルックアップ）で，削除されなかったキャッシュ上のオブジェクトリファレンスがそのまま使用できます。

2.8.3　キャッシング機能を使用するための設定

キャッシング機能を使用するための設定について説明します。J2EE アプリケーションの場合と EJB クライアントアプリケーションの場合で設定個所が異なります。

(1)　J2EE アプリケーションでキャッシング機能を使用する場合

J2EE アプリケーションでキャッシング機能を使用するための定義は，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に指定します。簡易構築定義ファイルでのキャッシング機能を使用するための定義について次の表に示します。

表 2-35　キャッシング機能を使用するための定義

指定するパラメタ	設定内容
ejbserver.jndi.cache	ネーミングでのキャッシングを有効にするかどうかを指定します。
ejbserver.jndi.cache.interval	ネーミングでのキャッシングをする場合，キャッシュをクリアする間隔（単位：秒）を指定します。
ejbserver.jndi.cache.interval.clear.option	インターバル経過後のネーミングでのキャッシュ領域に対する動作（キャッシュクリアの範囲）を決定します。

キャッシュを定期的にクリアするときの設定例（物理ティアの定義の場合）を次に示します。

（例）

```
<configuration>
  <logical-server-type>j2ee-server</logical-server-type>
  <param>
    <param-name>ejbserver.jndi.cache</param-name>
    <param-value>on</param-value>
  </param>
  <param>
    <param-name>ejbserver.jndi.cache.interval</param-name>
    <param-value>60</param-value>
  </param>
  <param>
    <param-name>ejbserver.jndi.cache.interval.clear.option</param-name>
    <param-value>check</param-value>
  </param>
   :
</configuration>
```

(2)　EJB クライアントアプリケーションでキャッシング機能を使用する場合

EJB クライアントアプリケーション起動時にプロパティとして設定します。

EJB クライアントアプリケーションのプロパティの設定方法は，EJB クライアントアプリケーションの開始に使用するコマンドによって，EJB クライアントアプリケーションのプロパティの設定方法が異なります。

- cjclstartap コマンドの場合

  cjclstartap コマンドを使用する場合は，EJB クライアントアプリケーションのプロパティファイル（usrconf.properties）で，プロパティを設定します。

- vbj コマンドの場合

  vbj コマンドを使用する場合は，バッチファイル/シェルスクリプトファイル，またはコマンドの引数で，プロパティを設定します。

キャッシュを定期的にクリアするときの設定例を次に示します。

EJB クライアントのシステムプロパティ設定例

```
   :
# キャッシュの設定
ejbserver.jndi.cache=on
ejbserver.jndi.cache.interval=60
ejbserver.jndi.cache.interval.clear.option=check
```

2.8.4　ネーミングでのキャッシングの注意事項

ネーミングのキャッシングに関する注意事項について説明します。

- アプリケーションで EJB ホームオブジェクトリファレンスや JDBC データソースをキャッシングしている場合，ネーミングでのキャッシングを無効にすることを推奨します。
- キャッシュを定期的にクリアするには，簡易構築定義ファイルで設定します。論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に，次のパラメタで設定します。

  ejbserver.jndi.cache.interval.clear.option

  　キャッシュをクリアする範囲を指定します。

  ejbserver.jndi.cache

  　キャッシュを実行するかどうかを指定します。ここでは「ON」を設定してください。

  ejbserver.jndi.cache.interval

  　キャッシュのクリア間隔を指定します。

- 論理 J2EE サーバ（j2ee-server）の<configuration>タグ内で，プロパティ「ejbserver.jndi.cache.interval.clear.option」に「check」を設定している場合，CORBA ネーミングサービスの監視は，プロパティ「ejbserver.jndi.cache.interval」に指定したキャッシュクリアのタイミングだけで行われます。CORBA ネーミングサービスの再起動後，CORBA ネーミングサービスの回復を検知するためには，最大で，プロパティ「ejbserver.jndi.cache.interval」に設定した時間が必要です。
- キャッシング機能を使用している場合，キャッシュに EJB ホームオブジェクトのオブジェクトリファレンスが保存された状態で，EJB ホームオブジェクトが保存されている J2EE サーバがダウンするか，または J2EE アプリケーションの再デプロイが実行されると，キャッシュに保存されている EJB ホームオブジェクトのオブジェクトリファレンスは無効な情報となります。この状態で EJB ホームオブジェクトの検索要求（lookup）を受けると，キャッシュ上の無効なオブジェクトリファレンスが検索要求元に返却されます。このオブジェクトリファレンスに対して，javax.rmi.PortableRemoteObject.narrow() メソッド，または create メソッドなどのメソッドを実行すると，CORBA 例外（org.omg.CORBA.OBJECT_NOT_EXIST など）が発生する場合があります。なお，CORBA 例外が

発生した場合，キャッシュ情報はすべて削除されます。次回の検索要求（lookup）では CORBA ネーミングサービスへ接続して有効な情報が取得されます。

- CTM を使用している場合に，指定した値の間隔で無効なキャッシュ領域だけをクリアするとき，J2EE サーバや J2EE アプリケーションが停止していても，グローバル CORBA ネーミングサービスの EJB ホームオブジェクトのオブジェクトリファレンスはキャッシュ領域からクリアされません。EJB ホームオブジェクトの検索要求（lookup）を受けると，クリアされなかったキャッシュ上のオブジェクトリファレンスが検索要求元に返却されます。J2EE アプリケーションが再度開始されていた場合は，キャッシュしたオブジェクトリファレンスはそのまま使用できます。J2EE アプリケーションが再度開始されていない場合は，返却されたオブジェクトリファレンスに対して，create メソッドなどのメソッドを実行すると，CORBA 例外（org.omg.CORBA.NO_IMPLEMENT）などが発生します。

  なお，CORBA 例外が発生した場合，キャッシュ情報はすべて削除されます。J2EE アプリケーションが再開始している場合は，次回の検索要求（lookup）でグローバル CORBA ネーミングサービスに接続して有効な情報が取得されます。

- ビジネスインタフェース使用時に，プロパティ「ejbserver.jndi.cache」に「on」を設定している状態で J2EE サーバを再起動した場合，ビジネスメソッド実行時に javax.ejb.EJBException が発生する場合があります。

  なお，javax.ejb.EJBException が発生した場合，次回の検索要求（lookup）では CORBA ネーミングサービスへ接続して有効な情報が取得されます。

2.9　ネーミングサービスの障害検知

ネーミングサービスの障害検知は，キャッシング機能のオプションとして使用します。

ネーミングサービスの障害検知機能を使用すると，ネーミングサービスの障害が発生した場合に，EJB クライアントが，より早くエラーを検知できます。

この節では，ネーミングサービスの障害検知機能の概要，およびネーミングサービスの障害検知機能と併用して使用することを推奨する機能について説明します。

この節の構成を次に示します。

表 2-36　この節の構成（ネーミング管理の障害検知）

分類	タイトル	参照先
解説	ネーミング管理の障害検知機能とは	2.9.1
	ラウンドロビン検索機能との併用	2.9.2
	ネーミング管理の障害検知機能の挙動	2.9.3
設定	実行環境の設定（障害検知機能を使用する場合）	2.9.4
注意事項	ネーミング管理の障害検知機能の注意事項	2.9.5

注　「実装」および「運用」について，この機能固有の説明はありません。

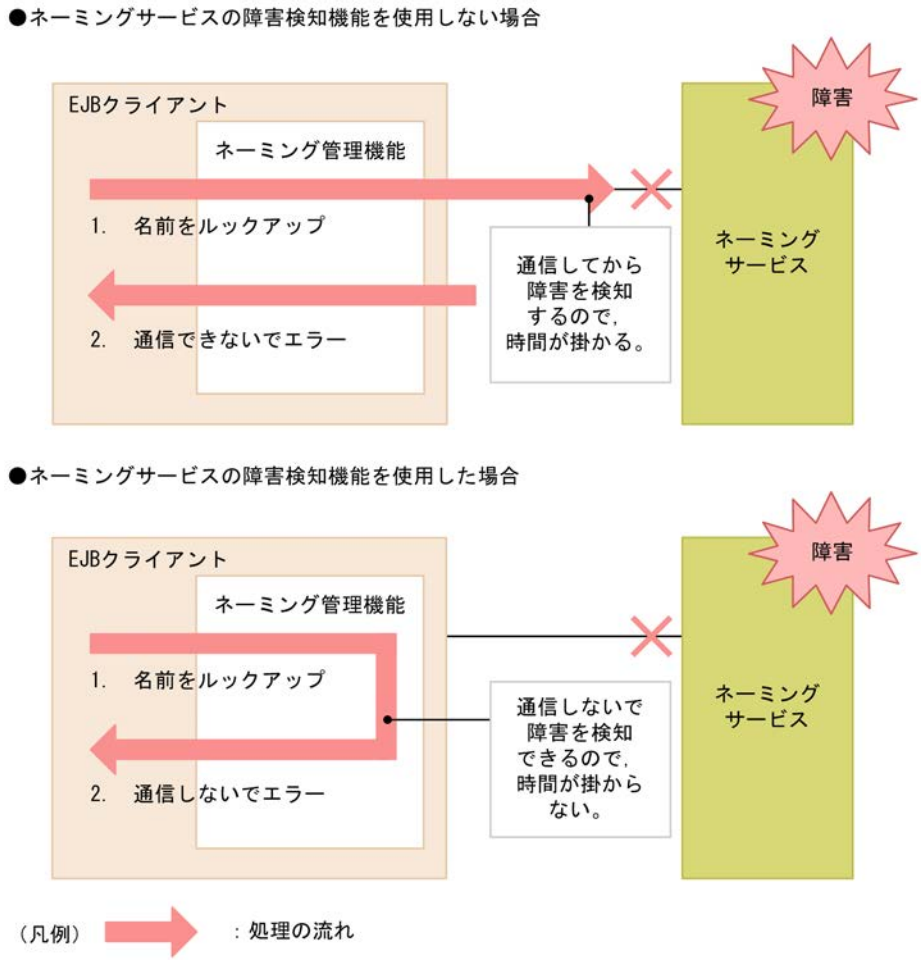
2.9.1　ネーミングサービスの障害検知機能とは

ネーミングサービスの障害検知機能を使用すると，ネーミングサービスが停止したり，アプリケーションサーバなどでマシンの障害やネットワーク障害が発生したりした場合，J2EE サーバが障害を検知します。

ネーミングサービスの障害検知機能では，J2EE サーバがネーミングサービスの機能の状態を監視し，通信ができないと判断したネーミングサービスの機能を使用することを抑止できます。そのため，EJB クライアントではむだな通信をしないで済みます。

ネーミングサービスの障害検知機能を使用した場合の処理の流れを次の図に示します。

図 2-13　ネーミングサービスの障害検知機能を使用した場合の処理の流れ



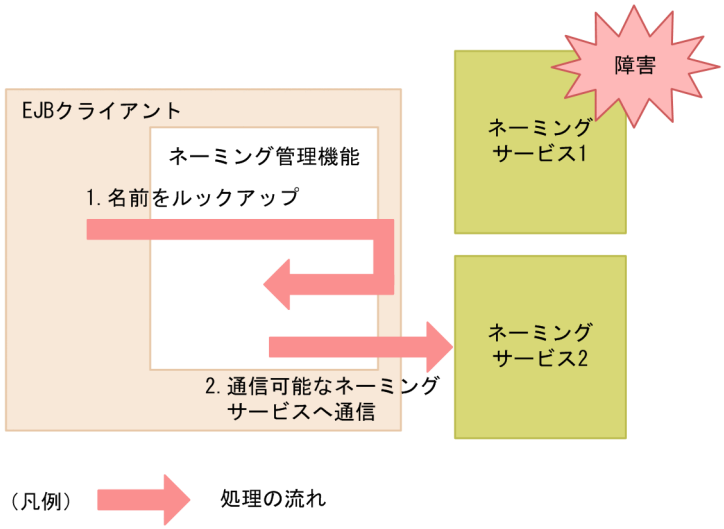
ネーミングサービスの障害検知機能を使用しない場合，名前をルックアップするときには必ずネーミングサービスに通信します。このため，通信できないでエラーになるまでに時間が掛かることがあります。一方，ネーミングサービスの障害検知機能を使用している場合，障害が発生しているネーミングサービスに対する通信はしないで，エラーを検知できます。このため，時間が掛かりません。

2.9.2　ラウンドロビン検索機能との併用

ラウンドロビン検索機能とネーミングサービスの障害検知機能を併用すると，一つの系で障害が発生したときに，障害が発生した系を容易に切り離しできます。

次に，ラウンドロビン検索機能とネーミングサービスの障害検知機能を併用した場合の通信について図で説明します。

図 2-14　ラウンドロビン検索機能とネーミングサービスの障害検知機能を併用した場合の通信



名前をルックアップした場合に特定のネーミングサービスに障害が発生していることを検知した場合，ラウンドロビン検索機能によって，通信可能なネーミングサービスへの通信に切り替えられます。

2.9.3　ネーミングサービスの障害検知機能の挙動

ネーミングサービスの障害検知機能の挙動について説明します。

(1)　閉塞のタイミング

ネーミングサービスの障害検知では，次のタイミングでネーミングサービスの状態を確認し，応答がなかった場合に閉塞します。

1. sweep インターバル時間経過のタイミングに応答がなかった場合
2. RMI/IIOP 通信エラーなどを契機にキャッシュの領域がすべてクリアされたあとや，キャッシュがない状態で，次の操作を最初に実行した時にネーミングサービスの応答がなかった場合
   - InitilalContext を生成した時
   - lookup をした時

キャッシュをすべてクリアしたあとのネーミングサービスに対する操作の挙動について表で説明します。

表 2-37　キャッシュをすべてクリアしたあとのネーミングサービスに対する操作の挙動

ネーミング管理の機能での操作	ラウンドロビン検索を使用している場合	ラウンドロビン検索を使用していない場合
InitialContext を生成したとき	EJB クライアントプロセス起動後の初回 InitialContext 生成時に，ネーミングサービスの稼働確認で応答がない場合は，閉塞される。 初回以外の場合は，閉塞されない。	生死確認をして，ネーミングサービスからの応答が検知できない場合，閉塞される。
lookup したとき	lookup の発行時に通信しようとしているネーミングサービスに対してだけ，状態確認をする。	生死確認をして，ネーミングサービスからの応答が検知できない場合，閉塞される。

ネーミング管理の機能での操作	ラウンドロビン検索を使用している場合	ラウンドロビン検索を使用していない場合
lookup したとき	応答が検知できない場合は，閉塞される。例えば三つのネーミングサービスでラウンドロビン検索をしている場合には，ダウンしたネーミングサービスが閉塞されるのは最大で 3 回目の lookup 時になる。	生死確認をして，ネーミングサービスからの応答が検知できない場合，閉塞される。

ネーミングサービスの障害検知機能では，閉塞されると同時に KDJE47111-I メッセージをログに出力します。メッセージが出力されたあとに，稼働中のネーミングサービスの機能に対する通信はすべて抑止されて，javax.naming.NamingException をスローされます。

(2)　閉塞した場合の挙動

ネーミングサービスの障害検知機能によって閉塞されたネーミングサービスに対して，EJB クライアントから InitialContext の生成，または lookup した場合のネーミングサービスの挙動について説明します。

閉塞したネーミングサービスに対する操作について，ラウンドロビン検索機能を使用しているか，使用していないかに分けて表で示します。

表 2-38　閉塞したネーミングサービスに対する操作

EJB クライアントからの操作	ラウンドロビン検索を使用している場合	ラウンドロビン検索を使用していない場合
InitialContext を生成したとき	ラウンドロビン用の InitialContext を返す。	常に通信を抑止する。 EJB クライアントに javax.naming.NamingException を返す。
lookup したとき	ラウンドロビングループに登録されている，別のネーミングサービスからオブジェクトを検索して返す。	常に通信を抑止する。 EJB クライアントに javax.naming.NamingException を返す。

(3)　ネーミングサービスの閉塞解除のタイミング

閉塞を解除するタイミングについて説明します。次に示すタイミングでネーミングサービスの状態確認をして，ネーミングサービスからの応答が検知できたときに閉塞を解除します。

- sweep インターバル時間経過のタイミング

閉塞を解除すると同時に，KDJE47110-I メッセージをログに出力します。メッセージが出力されたあとは，稼働中のネーミングサービスに対する通信は抑止されません。

2.9.4　実行環境の設定（障害検知機能を使用する場合）

ネーミングサービスの障害検知機能は，キャッシング機能のオプションです。このため，キャッシング機能の設定が前提となります。ネーミングサービスのキャッシング機能の設定については，「2.8.3　キャッシング機能を使用するための設定」を参照してください。

ネーミングサービスの障害検知機能を使用する場合は，次の表に示す値を設定してください。

表 2-39　ネーミングサービスの障害検知機能を使用する場合の設定

指定するパラメタ	値
ejbserver.jndi.cache	on
ejbserver.jndi.cache.interval	1～2,147,483,647
ejbserver.jndi.cache.interval.clear.option	check

EJB クライアントのシステムプロパティの設定例を次に示します。この例は，usrconf.properties に設定した場合の例です。また，この例では，ラウンドロビン検索機能もあわせて設定しています。

EJB クライアントのシステムプロパティ設定例

```
  :
# キャッシュの設定
ejbserver.jndi.cache=on
ejbserver.jndi.cache.interval=60
ejbserver.jndi.cache.interval.clear.option=check

# 論理ネーミングサービスの構成を定義
ejbserver.jndi.namingservice.group.list=g1;g2;g3
ejbserver.jndi.namingservice.group.g1.providerurls= corbaname::hostA:900;corbaname::hostB:900;corbaname::hostC:
900
ejbserver.jndi.namingservice.group.g2.providerurls=
                          corbaname::hostD:700;corbaname::hostE:700
ejbserver.jndi.namingservice.group.g3.providerurls=
  corbaname::hostF:800;corbaname::hostG:800;corbaname::hostH:800
  :
```

2.9.5　ネーミングサービスの障害検知機能の注意事項

ネーミングサービスの障害検知機能の注意事項について説明します。

(1)　閉塞解除のタイミングについて

ネーミングサービスの障害検知機能を使用しない場合，障害が発生したネーミングサービスへのクライアントアプリケーションからの検索は，ネーミングサービスおよび J2EE サーバが再起動した直後から成功します。

しかし，ネーミングサービスの障害検知機能を使用している場合，sweep インターバルのタイミングでしか閉塞の解除ができません。ネーミング管理機能では，ネーミングサービスが停止したあと，sweep インターバルが経過しないと実際のネーミングサービスに接続しません。つまり，ネーミングサービスの機能が回復したあとから検索に成功するまでに最大 sweep インターバル時間を必要とします。ネーミングサービスの障害検知機能を使用する場合，sweep インターバルの設定時間（ejbserver.jndi.cache.interval プロパティの値）に，短い時間（推奨値は 60）を設定することを推奨します。

(2)　07-60 までとの挙動の違い

アプリケーションサーバのバージョンが 07-60 までと 08-00 以降では，閉塞するタイミングが異なります。08-00 以降では，sweep インターバル時間経過タイミングに加えて，クライアントアプリケーションから検索した場合に稼働状態の確認を実施します。

(3)　ネーミング管理の機能の生死確認

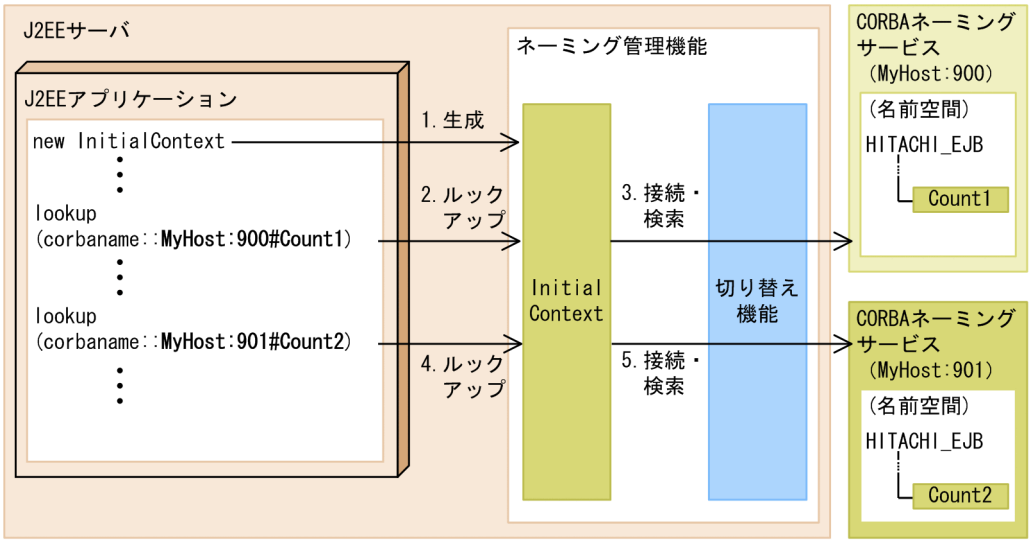
ネーミングサービスの障害検知機能を使って稼働状態の確認を実施した場合に，一時的なネットワーク障害，サーバ側高負荷や FullGC 中などで応答がないときに，ネーミングサービスが停止したと判断されることがあります。

2.10　CORBA ネーミングサービスの切り替え

ネーミング管理機能では，JNDI を介して分散オブジェクトを検索するときに，JNDI が接続する CORBA ネーミングサービスの切り替えができます。切り替えは，InitialContext クラスのインスタンスに対して，プリフィックス「corbaname:」を付与した名前を lookup メソッドの引数に渡すことで実現できます。

CORBA ネーミングサービスの切り替えの流れを次の図に示します。

図 2-15　CORBA ネーミングサービスの切り替えの流れ



ネーミングサービスの切り替えの流れを説明します。この流れでは，J2EE アプリケーションから，プリフィックス「corbaname:」を付与したルックアップを 2 回実行します。プリフィックスの後ろに指定したホスト名に応じて，接続する CORBA ネーミングサービスが切り替えられます。

1. J2EE アプリケーションが，JNDI の javax.naming.InitialContext クラスのインスタンスを生成します。
2. J2EE アプリケーションが，javax.naming.InitialContext クラスのインスタンスに対して，オブジェクトの検索（ルックアップ）を要求します。このとき，プリフィックス「corbaname:」を付与した名前「corbaname::MyHost:900#Count1」を指定します。
3. 要求を受けたネーミング管理機能から「MyHost:900」の CORBA ネーミングサービスへの接続が実行されます。接続後，ルックアップで指定した名前の EJB ホームのオブジェクトリファレンス「Count1」が検索されます。
4. J2EE アプリケーションから，InitialContext クラスのインスタンスに対して，オブジェクトの検索（ルックアップ）を要求します。このとき，プリフィックス「corbaname:」を付与した名前「corbaname::MyHost:901#Count2」を指定します。
5. 要求を受けたネーミング管理機能から「MyHost:901」の CORBA ネーミングサービスへの接続が実行されます。接続後，ルックアップで指定した名前の EJB ホームのオブジェクトリファレンス「Count2」が検索されます。

CORBA ネーミングサービスの切り替えは，サーバ管理コマンドを使用した，Enterprise Bean リファレンスの解決で実行します。操作については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「9.　J2EE アプリケーションのプロパティ設定」を参照してください。

なお，CORBA ネーミングサービスがローカルホスト上で動作する場合，ネーミングサービスのホスト名に関する設定では，"localhost"の文字列ではなく，マシン名または IP アドレスを指定してください。

ネーミングサービスのホスト名の設定は，J2EE サーバのプロパティをカスタマイズして設定します。設定方法については，「2.3.5　実行環境での設定」を参照してください。

2.11　EJB ホームオブジェクトリファレンスの再利用（EJB ホームオブジェクトへの再接続機能）

EJB ホームオブジェクトへの再接続機能とは，J2EE サーバの障害などのあとで，J2EE サーバを再起動した場合や J2EE アプリケーションを再開始した場合に，EJB クライアントアプリケーションが取得した EJB ホームオブジェクトを再利用できる機能です。J2EE サーバの再起動後や J2EE アプリケーションの再開始後に，EJB クライアントアプリケーションが取得した EJB ホームオブジェクトを，再ルックアップしないでそのまま利用できます。

Session Bean（Stateless Session Bean または Stateful Session Bean）の EJB ホームオブジェクトのリファレンスに利用できます。

この節の構成を次の表に示します。

表 2-40　この節の構成（EJB ホームオブジェクトへの再接続機能）

分類	タイトル	参照先
設定	実行環境での設定（J2EE サーバの設定）	2.11.1
注意事項	EJB ホームオブジェクトリファレンスを再利用する場合の注意事項	2.11.2

注　「解説」,「実装」，および「運用」について，この機能固有の説明はありません。

2.11.1　実行環境での設定（J2EE サーバの設定）

EJB ホームオブジェクトへの再接続機能を使用する場合, J2EE サーバの設定が必要です。J2EE サーバの設定は，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に指定します。

簡易構築定義ファイルでの EJB ホームオブジェクトへの再接続機能の定義について次の表に示します。

表 2-41　簡易構築定義ファイルでの EJB ホームオブジェクトへの再接続機能の定義

指定するパラメタ	設定内容
ejbserver.container.ejbhome.sessionbean.reconnect.enabled	EJB ホームオブジェクトへの再接続機能を有効にするかどうかを指定します。

簡易構築定義ファイルおよび指定するパラメタの詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.6　簡易構築定義ファイル」を参照してください。

2.11.2　EJB ホームオブジェクトリファレンスを再利用する場合の注意事項

EJB ホームオブジェクトへの再接続機能を使用する場合の注意事項を次に示します。

- J2EE サーバの再起動や J2EE アプリケーションの再開始の際に，J2EE アプリケーションを変更しないでください。J2EE アプリケーションを変更すると，EJB ホームオブジェクトのリファレンスは再利用できません。
- J2EE サーバの通信ポートを固定してください。また，J2EE サーバを再起動するときに，固定した値を変更しないでください。

定義方法については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「2.14.1　通信ポートの固定」，およびマニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「2.14.2　IP アドレスの固定」を参照してください。

- Java アプリケーション用ユーザプロパティファイルのリモートインタフェースでの通信障害発生時の EJB クライアントの動作（ejbserver.container.rebindpolicy キー）の値に，「VB_TRANSPARENT」を指定してください。

  定義方法については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「2.13　EJB のリモートインタフェースの呼び出し」を参照してください。

- EJB 呼び出しで通信障害が発生した場合，通信障害発生時の EJB クライアントの動作（ejbserver.container.rebindpolicy キー）の値に「VB_TRANSPARENT」を指定していると，コネクションが再接続され，リクエストが再送されます。

  そのため，EJB ホームオブジェクトへの再接続機能は参照系のシステムでの使用を推奨します。

3 リソース接続とトランザクション管理

この章では，アプリケーションサーバが接続できるリソースと，リソースへの接続について説明します。また，リソース接続でのトランザクション管理についても説明します。

3.1　この章の構成

リソース接続とトランザクション管理の機能と参照先を次の表に示します。

表 3-1　リソース接続とトランザクション管理の機能と参照先

機能	参照先
リソース接続とトランザクション管理の概要	3.2
リソース接続	3.3
トランザクション管理	3.4
リソースへのサインオン方式	3.5
データベースへの接続	3.6
データベース上のキューとの接続	3.7
OpenTP1 との Outbound での接続（SPP または TP1/Message Queue）	3.8
OpenTP1 との Inbound での接続	3.9
CJMS プロバイダとの接続	3.10
SMTP サーバとの接続	3.11
JavaBeans リソースの利用	3.12
そのほかのリソースとの接続	3.13
パフォーマンスチューニングのための機能	3.14
フォールトトレランスのための機能	3.15
そのほかのリソースアダプタの機能（Connector 1.5 仕様に準拠するリソースアダプタの場合）	3.16
クラスタコネクションプール機能	3.17
リソースへの接続テスト	3.18
ファイアウォール環境での運用のための機能	3.19
EJB クライアントアプリケーションでトランザクションを開始する場合の注意事項	3.20

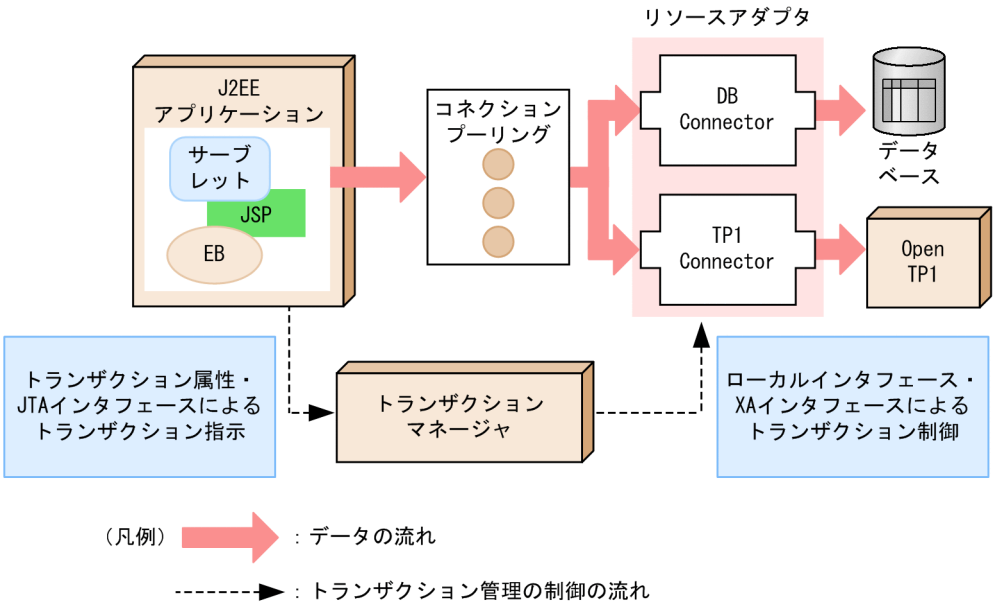
3.2　リソース接続とトランザクション管理の概要

J2EE サーバ上の J2EE アプリケーションに含まれる EJB，サーブレット，JSP などの J2EE コンポーネントからは，データベースや OpenTP1 などのリソースに接続できます。データベースや OpenTP1 などのリソースに接続する場合は，リソースアダプタを使用します。アプリケーションサーバでは，Connector 1.0 仕様または Connector 1.5 仕様に準拠したリソースアダプタを使用できます。SMTP サーバや JavaBeans リソースなどのリソースに，リソースアダプタを使用しないで接続する機能も提供しています。

また，アプリケーションサーバでは，リソースに効率的かつ信頼性の高い方法でアクセスするために，コネクションプーリングやトランザクション管理の機能を提供しています。コネクションプーリングを使用すると，リソースに対するコネクションをプーリングして，効率的にコネクションを使用できます。また，障害が発生したコネクションを適切にコネクションプールから取り除きます。また，トランザクション管理の機能を使用すると，トランザクションマネージャが，EJB のメソッドごとに指定するトランザクション属性や JTA インタフェース（UserTransaction）による指示に基づいて，リソースアクセスのトランザクションを適切に制御します。複数のリソースに対してトランザクション管理をする場合には，グローバルトランザクションを使用します。グローバルトランザクションを使用すると，2 フェーズコミットプロトコルによってトランザクションが管理されるため，リソース間で更新の整合性が確保されます。

コネクションプーリング，およびトランザクション管理の機能を使用したリソースへの接続の例を次の図に示します。

図 3-1　コネクションプーリングおよびトランザクション管理の機能を使用したリソースへの接続の例



3.3　リソース接続

この節では，リソースの種類ごとの接続方法と，接続に使用するリソースアダプタの種類について説明します。

また，リソースアダプタを使用する場合の説明として，リソースアダプタの使用方法，機能および注意事項についても説明します。

この節の構成を次の表に示します。

表 3-2　この節の構成（リソース接続）

分類	タイトル	参照先
解説	リソースへの接続方法	3.3.1
	リソースアダプタの種類	3.3.2
	リソースアダプタの使用方法	3.3.3
	リソースアダプタの機能	3.3.4
	リソースアダプタ以外の機能	3.3.5
実装	リソースに接続するための実装	3.3.6
設定	リソースアダプタの設定方法	3.3.7
	リソースアダプタの設定の流れ（J2EE リソースアダプタとしてデプロイして使用する場合）	3.3.8
	リソースアダプタの設定の流れ（J2EE アプリケーションに含めて使用する場合）	3.3.9
	リソースアダプタの設定の流れ（Inbound で使用する場合）	3.3.10
	リソースアダプタの設定の流れ（クラスタコネクションプールを使用する場合）	3.3.11
	リソースアダプタ以外を使用する接続の設定	3.3.12
注意事項	リソースアダプタについての注意事項	3.3.13

注　「運用」について，この機能固有の説明はありません。

3.3.1　リソースへの接続方法

アプリケーションサーバでは，リソースの種類によって，接続にリソースアダプタを使用するものと，リソースアダプタを使用しないものがあります。それぞれのリソースへの接続方法の概要を，リソースの種類ごとに説明します。

(1)　接続にリソースアダプタを使用するリソース

接続にリソースアダプタを使用するリソースを次に示します。

データベース

データベースと接続できます。データベースと接続するためには，リソースアダプタとして DB Connector を使用します。

DB Connector で接続できるのは，次のデータベースです。

- HiRDB
- Oracle
- SQL Server※
- XDM/RD E2

注※　SQL Server と接続できるのは Windows の場合だけです。

データベースとの接続については，「3.6.1　DB Connector による接続の概要」を参照してください。

データベース上のキュー

Reliable Messaging で使用するデータベース上のキューと接続できます。データベース上のキューと接続するためには，リソースアダプタとして DB Connector for Reliable Messaging および Reliable Messaging を使用します。

DB Connector for Reliable Messaging および Reliable Messaging で接続できるのは，次のデータベースです。

- HiRDB
- Oracle

データベース上のキューとの接続については，「3.7.1　DB Connector for Reliable Messaging と Reliable Messaging による接続の概要」を参照してください。

OpenTP1

OpenTP1 の SPP，TP1/Message Queue と Outbound で接続できます。OpenTP1 の SPP と Outbound で接続するには，リソースアダプタとして TP1 Connector を使用します。TP1/Message Queue と接続するには，リソースアダプタとして TP1/Message Queue - Access を使用します。

また，OpenTP1 の SUP からアプリケーションサーバに Inbound で接続することもできます。OpenTP1 の SUP から Inbound で接続するには，リソースアダプタとして TP1 インバウンドアダプタを使用します。

OpenTP1 との接続の詳細については，接続に使用するリソースアダプタごとに説明しています。参照先を次の表に示します。

表 3-3　OpenTP1 との接続の詳細説明の参照先

接続方法	参照先
TP1 Connector を使用した接続	「3.8.1　TP1 Connector による接続」
TP1/Message Queue - Access を使用した接続	「3.8.2　TP1/Message Queue - Access による接続」
TP1 インバウンドアダプタを使用した接続	「4.　OpenTP1 からのアプリケーションサーバの呼び出し（TP1 インバウンド連携機能）」

CJMSP ブローカー

CJMS プロバイダの機能を使用する場合，CJMSP ブローカーと接続できます。CJMSP ブローカーと接続するためには，リソースアダプタとして CJMSP リソースアダプタを使用します。

CJMSP リソースアダプタを使用した CJMSP ブローカーとの接続については，「7.　CJMS プロバイダ」を参照してください。

そのほかのリソース

リソースの種類に関係なく，Connector 1.0 仕様または Connector 1.5 仕様に準拠したリソースアダプタで接続できるリソースに接続できます。

使用できるリソースアダプタについては，「3.13　そのほかのリソースとの接続」を参照してください。また，Connector 1.5 仕様に準拠したリソースアダプタを使用する場合に，アプリケーションサーバで使用できる機能については，「3.16　そのほかのリソースアダプタの機能（Connector 1.5 仕様に準拠するリソースアダプタの場合）」を参照してください。

(2)　接続にリソースアダプタを使用しないリソース

接続にリソースアダプタを使用しないリソースについて次に示します。

SMTP サーバ

SMTP サーバと接続できます。SMTP サーバとの接続については，「3.11　SMTP サーバとの接続」を参照してください。

JavaBeans

リソースとして JavaBeans リソースを利用できます。JavaBeans リソースの利用については，「3.12　JavaBeans リソースの利用」を参照してください。

3.3.2　リソースアダプタの種類

アプリケーションサーバでは，Connector 1.0 仕様または Connector 1.5 仕様に準拠したリソースアダプタを使用できます。

ここでは，それぞれの仕様に準拠したリソースアダプタについて説明します。また，リソースアダプタの DD のスキーマの違いについても説明します。

(1)　Connector 1.0 仕様に準拠したリソースアダプタ

アプリケーションサーバでは，次のリソースアダプタを使用できます。

- DB Connector
- DB Connector for Reliable Messaging および Reliable Messaging
- TP1 Connector
- TP1/Message Queue - Access

これらのリソースアダプタでは，Connector 1.0 仕様に対応する機能が使用できます。

なお，DB Connector および DB Connector for Reliable Messaging では，Connector 1.0 仕様に対応する機能に加えて，アプリケーションサーバで追加された機能を使用できます。使用できる機能については，「3.3.4　リソースアダプタの機能」を参照してください。

ポイント

アプリケーションサーバでは，これらのリソースアダプタ以外にも，標準仕様の Connector 1.0 仕様に準拠したリソースアダプタを使用できます。ただし，標準仕様の Connector 1.0 仕様に準拠したリソースアダプタを使用する場合，DD（ra.xml）の次のタグの設定内容は無視されます。

- <security-permission>
  セキュリティの設定には server.policy ファイルを使用してください。
- <authentication-mechanism>
  設定内容に関係なく，BasicPassword が適用されます。

(2)　Connector 1.5 仕様に準拠したリソースアダプタ

アプリケーションサーバでは，Connector 1.5 仕様に準拠したリソースアダプタを使用できます。アプリケーションサーバで使用できるのは，Connector 1.5 仕様の規約のうち，次の規約に対応した機能です。

- Lifecycle Management
- Work Management
- メッセージインフロー
- トランザクションインフロー

アプリケーションサーバでは，次のリソースアダプタを提供しています。

- TP1 インバウンドアダプタ
- CJMSP リソースアダプタ

使用できる機能については，「3.3.4　リソースアダプタの機能」を参照してください。

また，Connector 1.5 仕様に準拠したリソースアダプタについては，「3.13.1　そのほかのリソースとの接続に使用するリソースアダプタ」を参照してください。

(3)　Connector 1.0 仕様と Connector 1.5 仕様のリソースアダプタのスキーマの違い

Connector 1.0 仕様に準拠したリソースアダプタと，Connector 1.5 仕様に準拠したリソースアダプタの，DD のスキーマの違いについて説明します。リソースアダプタの DD は，ra.xml です。

Connector 1.0 仕様のスキーマから Connector 1.5 仕様のスキーマに対する主な変更点を次の表に示します。ここで示す以外の変更点については，Connector 1.5 仕様を参照してください。

表 3-4　Connector 1.0 仕様のスキーマから Connector 1.5 仕様のスキーマに対する主な変更点

Connector 1.5 仕様で変更になった内容	Connector 1.5 仕様の DD の内容
javax.resource.spi.ResourceAdapter インタフェースの実装クラスの指定	<connector>-<resourceadapter>-<resourceadapter-class>タグ <connector>-<resourceadapter>-<config-property>タグ
Outbound リソースアダプタの指定	<connector>-<resourceadapter>-<outbound-resourceadapter>タグ
Inbound リソースアダプタの指定	<connector>-<resourceadapter>-<inbound-resourceadapter>タグ
adminobject の指定	<connector>-<resourceadapter>-<adminobject>タグ

Connector 1.5 仕様で変更になった内容の概要を説明します。

javax.resource.spi.ResourceAdapter インタフェースの実装クラスの指定

javax.resource.spi.ResourceAdapter インタフェースの実装クラスと，そのコンフィグレーションプロパティを指定する要素が追加されました。javax.resource.spi.ResourceAdapter インタフェースの追加に伴って実現できる機能については，「3.16.1　リソースアダプタのライフサイクル管理」および「3.16.2　リソースアダプタのワーク管理」を参照してください。

Outbound リソースアダプタの指定

Outbound のリソースアダプタを明示的に定義するための要素が追加されました。

なお，Outbound リソースアダプタでは，一つの DD 内にコネクション定義を複数指定できます。コネクション定義の複数指定については，「3.16.6　コネクション定義の複数指定」を参照してください。

Inbound リソースアダプタの指定

Inbound のリソースアダプタを明示的に定義するための要素が追加されました。

adminobject の指定

管理対象オブジェクトについての情報を指定する要素が追加されました。

(4)　リソースアダプタごとの RAR ファイルの種類

プロパティを定義するリソースアダプタは，接続するリソースや使用するトランザクションの種類などによって異なります。ここでは，次の場合に使用するリソースアダプタについて説明します。

- DB Connector を使用する場合
- DB Connector for Reliable Messaging と Reliable Messaging を使用する場合
- ほかのリソースアダプタを使用する場合

(a)　DB Connector を使用する場合

DB Connector は，接続するデータベースの種類や使用するトランザクションの種類によって，ファイルが異なります。DB Connector の種類を次の表に示します。

表 3-5　DB Connector の種類

RAR ファイル名	説明
DBConnector_HiRDB_Type4_CP.rar	HiRDB Type4 JDBC Driver を使用して，ローカルトランザクションまたはトランザクション管理なしで，HiRDB，XDM/RD E2 に接続する場合に選択します。
DBConnector_HiRDB_Type4_XA.rar	HiRDB Type4 JDBC Driver を使用して，グローバルトランザクションで，HiRDB に接続する場合に選択します。
DBConnector_Oracle_CP.rar	Oracle JDBC Thin Driver を使用して，ローカルトランザクションまたはトランザクション管理なしで，Oracle に接続する場合に選択します。
DBConnector_Oracle_XA.rar	Oracle JDBC Thin Driver を使用して，グローバルトランザクションで，Oracle に接続する場合に選択します。
DBConnector_SQLServer_CP.rar	SQL Server JDBC Driver を使用して，ローカルトランザクションまたはトランザクション管理なしで，SQL Server （Windows の場合だけ）に接続する場合に選択します。
DBConnector_CP_ClusterPool_Root.rar	クラスタコネクションプール機能のルートリソースアダプタです。ルートリソースアダプタに属するメンバリソースアダプタが，ローカルトランザクションまたはトランザクション管理なしで，Oracle に接続する場合に選択します。
DBConnector_Oracle_CP_ClusterPool_Member.rar	クラスタコネクションプール機能のメンバリソースアダプタです。 Oracle JDBC Thin Driver を使用して，ローカルトランザクションまたはトランザクション管理なしで，Oracle に接続する場合に選択します。 なお，J2EE アプリケーションのリソースリファレンスに設定して使用できません。

注　新規に，DB Connector の RAR ファイルを使用する場合，アプリケーションサーバで提供する Connector 属性ファイルのテンプレートファイルを使用して，プロパティを定義できます。Connector 属性ファイルのテンプレートファイルは，すべての DB Connector の RAR ファイルに対して提供しています。提供しているテンプレートファイルについては，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1.14　Connector 属性ファイルのテンプレートファイル」を参照してください。

(b)　DB Connector for Reliable Messaging と Reliable Messaging を使用する場合

Reliable Messaging と連携してデータベースに接続する場合には，Reliable Messaging 連携用のリソースアダプタ（DB Connector for Reliable Messaging）と，Reliable Messaging で提供するリソースアダプタの両方をインポートする必要があります。Reliable Messaging で提供するリソースアダプタについては，マニュアル「Reliable Messaging」を参照してください。

DB Connector for Reliable Messaging は，使用するトランザクションの種類や接続するデータベースの種類によって，ファイルが異なります。DB Connector for Reliable Messaging の種類を次の表に示します。

表 3-6　DB Connector for Reliable Messaging の種類

RAR ファイル名	説明
DBConnector_HiRDB_Type4_CP_Cosminexus_RM.rar	HiRDB Type4 JDBC Driver を使用して，ローカルトランザクションまたはトランザクション管理なしで，HiRDB に接続する場合に選択します。
DBConnector_HiRDB_Type4_XA_Cosminexus_RM.rar	HiRDB Type4 JDBC Driver を使用して，グローバルトランザクションで，HiRDB に接続する場合に選択します。
DBConnector_Oracle_CP_Cosminexus_RM.rar	Oracle JDBC Thin Driver を使用して，ローカルトランザクションまたはトランザクション管理なしで，Oracle に接続する場合に選択します。
DBConnector_Oracle_XA_Cosminexus_RM.rar	Oracle JDBC Thin Driver を使用して，グローバルトランザクションで，Oracle に接続する場合に選択します。

注　新規に，DB Connector for Reliable Messaging の RAR ファイルを使用する場合，アプリケーションサーバで提供する Connector 属性ファイルのテンプレートファイルを使用して，プロパティを定義できます。Connector 属性ファイルのテンプレートファイルは，すべての DB Connector の RAR ファイルに対して提供しています。提供しているテンプレートファイルについては，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1.14　Connector 属性ファイルのテンプレートファイル」を参照してください。

ポイント

DB Connector for Reliable Messaging と Reliable Messaging を使用して同一トランザクション内で JDBC と JMS のアクセスを行う場合に，同じ物理コネクションを共有することで，ローカルトランザクションの適用，グローバルトランザクションの 1 相コミット決着ができます。1 相コミット決着のための条件を次に示します。

- DB Connector for Reliable Messaging と Reliable Messaging で使用するデータベースシステムが同一であり，かつサインオン方式とセキュリティ情報（ユーザ名，パスワード）が同一である。
- リソースアダプタを使用する J2EE アプリケーションの属性ファイル（Session Bean 属性ファイル，Entity Bean 属性ファイルなど）で<res-sharing-scope>タグに「Shareable」を指定している。

(c)　ほかのリソースアダプタを使用する場合

OpenTP1 の SPP と Outbound で接続する場合には，TP1 Connector および TP1/Client/J の提供するリソースアダプタを使用します。詳細については，TP1 Connector のドキュメント，およびマニュアル「OpenTP1 クライアント使用の手引 TP1/Client/J 編」を参照してください。

OpenTP1 の SUP から Inbound で接続する場合には，TP1 インバウンドアダプタを使用します。詳細については，「4.　OpenTP1 からのアプリケーションサーバの呼び出し（TP1 インバウンド連携機能)」を参照してください。

また，CJMS プロバイダを使用する場合，CJMSP ブローカーと接続するために，CJMSP リソースアダプタを使用します。詳細については，「7.　CJMS プロバイダ」を参照してください。

なお，新規に TP1 インバウンドアダプタまたは CJMSP リソースアダプタの RAR ファイルを使用する場合，アプリケーションサーバで提供するテンプレートファイルを使用して，プロパティを定義できます。詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1.14　Connector 属性ファイルのテンプレートファイル」を参照してください。

TP1/Message Queue と接続する場合には，TP1/Message Queue - Access の提供するリソースアダプタを使用します。詳細については，マニュアル「OpenTP1 Version 7 メッセージキューイングアクセス機能 TP1/Message Queue - Access 使用の手引」を参照してください。

また，アプリケーションサーバでは，Connector 1.0 仕様または Connector 1.5 仕様に準拠したリソースアダプタを使用して，任意のリソースに接続できます。これらのリソースアダプタを使用する場合は，リソースアダプタのドキュメントを参照してください。

3.3.3　リソースアダプタの使用方法

ここでは，リソースアダプタの使用方法について説明します。リソースアダプタを使用してリソースと接続する方法には，次の 2 種類があります。

- J2EE リソースアダプタとしてデプロイして使用する方法
- J2EE アプリケーションに含めて使用する方法

ここでは，それぞれの方法について説明します。

(1)　J2EE リソースアダプタとしてデプロイして使用する

J2EE サーバにインポートしたリソースアダプタを，共有スタンドアロンモジュールとしてデプロイします。その J2EE サーバ上で動作するすべての J2EE アプリケーションで使用できるようになります。J2EE サーバ上に配置されたリソースアダプタを **J2EE リソースアダプタ**といいます。

(2)　J2EE アプリケーションに含めて使用する

リソースアダプタを J2EE アプリケーションに含めて使用します。同じ J2EE アプリケーションに含まれる EJB や WAR から，リソースアダプタを使用できます。リソースアダプタを J2EE アプリケーションに含めて使用する手順については，「3.3.9　リソースアダプタの設定の流れ（J2EE アプリケーションに含めて使用する場合)」を参照してください。

なお，J2EE アプリケーションに含めることができるリソースアダプタには，制限があります。詳細については，「(3)　使用方法ごとに使用できるリソースアダプタ」を参照してください。

(3)　使用方法ごとに使用できるリソースアダプタ

リソースアダプタは，Application Server のバージョンに適したリソースアダプタに統一する必要があります。また，アプリケーションサーバでは，Connector 1.0 仕様，および Connector 1.5 仕様に準拠したリソースアダプタをデプロイして使用できます。ただし，J2EE アプリケーションに含めて使用する場合，次のリソースアダプタは使用できません。

- XATransaction に設定したリソースアダプタ
- ネイティブライブラリを含むリソースアダプタ
- 起動順序の制御が必要なリソースアダプタ
- アプリケーションサーバ独自の機能を使用しているリソースアダプタ

アプリケーションサーバが提供するリソースアダプタの使用方法を次の表に示します。

表 3-7　アプリケーションサーバが提供するリソースアダプタの使用方法

リソースアダプタ			使用方法	
			リソースアダプタを J2EE リソースアダプタとしてデプロイして使用する	リソースアダプタを J2EE アプリケーションに含めて使用する
DB Connector	NoTransaction または LocalTransaction に設定		○	○
	XATransaction に設定		○	−
	クラスタ構成	ルートリソースアダプタ	○	−
		メンバリソースアダプタ	○	−
TP1 Connector	NoTransaction または LocalTransaction に設定		○	○
	XATransaction に設定		○	−
TP1/Message Queue - Access			○	−
Reliable Messaging	Reliable Messaging（データベースなし）		○	−
	Reliable Messaging（データベースあり）		○	−
	DB Connector for Reliable Messaging		○	−
TP1 インバウンドアダプタ			○	−
CJMSP リソースアダプタ			○	−

（凡例）　○：使用できる　−：該当しない

使用できないリソースアダプタに対して次の操作をすると，エラーメッセージが出力されて，操作に失敗します。

- J2EE アプリケーションに含めてインポートする
- J2EE アプリケーションに含めて接続テストまたは開始する

3.3.4　リソースアダプタの機能

ここでは，次のリソースアダプタの機能について説明します。

- DB Connector
- クラスタコネクションプールで使用する DB Connector（ルートリソースアダプタおよびメンバリソースアダプタ）
- DB Connector for Reliable Messaging
- TP1 インバウンドアダプタ
- CJMSP リソースアダプタ

- そのほかの Connector 1.5 仕様に準拠したリソースアダプタ

なお，これ以外のリソースアダプタの機能については，使用するリソースアダプタのドキュメントを参照してください。

使用できる機能をリソースアダプタの種類ごとに次の表に示します。それぞれの機能の詳細については，参照先の説明を参照してください。

表 3-8　リソースアダプタの種類ごとの使用できる機能

機能	リソースアダプタの種類							参照先
	DB Connector	ルートリソースアダプタ	メンバリソースアダプタ	DB Connector for Reliable Messaging	TP1 インバウンドアダプタ	CJMSP リソースアダプタ	そのほかの Connector 1.5 仕様に準拠したリソースアダプタ※1	
コネクションプーリング	○	×	◎	○	−	○	○	3.14.1
コネクションプールのウォーミングアップ	○	×	○	○	−	○	○	3.14.2
コネクション数調節機能	○	×	○	○	−	○	○	
コネクションシェアリング・アソシエーション	○	×	○	○	−	×	○	3.14.3
ステートメントプーリング	○	×	○	○	−	×	○	3.14.4
DataSource オブジェクトのキャッシング	○	○	×	○	−	×	×	3.14.7
DB Connector のコンテナ管理でのサインオンの最適化	○	×	○	○	−	×	×	3.14.8
コネクションの障害検知	○	×	◎	○	−	×	○	3.15.1
コネクション枯渇時のコネクション取得待ち	○	×	◎	○	−	○	○	3.15.2
コネクションの取得リトライ	○	×	×	○	−	○	○	3.15.3
コネクションプールの情報表示	○	×	○	○	−	○	×	3.15.4
コネクションプールのクリア	○	×	○	○	−	○	○	3.15.5

機能	リソースアダプタの種類							参照先
	DB Connector	ルートリソースアダプタ	メンバリソースアダプタ	DB Connector for Reliable Messaging	TP1 インバウンドアダプタ	CJMSP リソースアダプタ	そのほかの Connector 1.5 仕様に準拠したリソースアダプタ※1	
コネクションの自動クローズ	○	×	○	○	−	×	○	3.15.6
コネクションスイーパ	○	×	○	○	−	○	○	3.15.7
ステートメントキャンセル	○	×	○	○	−	×	○	3.15.8
障害調査用 SQL の出力	◎	×	◎	◎	−	×	×	3.15.10
オブジェクトの自動クローズ	○	×	○	○	−	×	×	3.15.11
リソースアダプタのライフサイクル管理	×	×	×	×	○	○	○	3.16.1
リソースアダプタのワーク管理	×	×	×	×	○	○	○	3.16.2
メッセージインフロー	×	×	×	×	○	○	○	3.16.3
トランザクションインフロー	×	×	×	×	○	×	×	3.16.4
管理対象オブジェクトのルックアップ	×	×	×	×	−	○	○	3.16.5
コネクション定義の複数指定	×	×	×	×	−	○	○	3.16.6
コネクションプールの一時停止	×	×	○	×	−	×	×	3.17.4
コネクションプールの再開	×	×	○	×	−	×	×	
コネクションプールの状態表示	○	×	○	○	−	×	○	
リソースへの接続テスト	○	○	○	○	−	○	○	3.18
J2EE リソースへの別名付与	○	○	×	○	−	○	○	2.6
コネクション ID の PRF トレース	○	○	○	○	−	−	−	※2

（凡例）　◎：必ず有効になる　○：使用できる　×：使用できない　－：該当しない

注※1　この表で示しているのはアプリケーションサーバが提供する機能が使用できるかどうかです。リソースアダプタが提供する機能については，使用するリソースアダプタのドキュメントを参照してください。

注※2　マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「4.6　性能解析トレース」を参照してください。

3.3.5　リソースアダプタ以外の機能

ここでは，リソースアダプタ以外で実現される機能について説明します。ここで説明する機能は，リソースアダプタの種類に関係なく使用できます。

リソースアダプタ以外で実現される機能を，次の表に示します。それぞれの機能の詳細については，参照先の説明を参照してください。

表 3-9　リソースアダプタ以外の機能

機能	参照先
ライトトランザクション	3.14.5
インプロセストランザクションサービス	3.14.6
トランザクションタイムアウトとステートメントキャンセル	3.15.8
トランザクションリカバリ	3.15.9

3.3.6　リソースに接続するための実装

アプリケーションからリソースに接続するには，Enterprise Bean やサーブレットでリソースへの参照を取得する必要があります。リソースへの参照を取得する方法として，ルックアップを使用する方法と DI（Dependency Injection）を使用する方法があります。

なお，EJB 3.0 以降を使用する場合，DI を使用してリソースへの参照を取得してください。

(1)　ルックアップを使用してリソースへの参照を取得する方法

ルックアップを使用する場合，次の手順でアプリケーションからリソースに接続します。

1. **リソースに接続するコネクションを取得するためのファクトリクラスを，JNDI を使用してルックアップします。**
   ルックアップする名前は，Enterprise Bean やサーブレットの DD で指定します。該当するタグは<resource-ref>タグ中の<res-ref-name>タグです。
2. **コネクションのファクトリクラスを使用して，コネクションを取得します。**
3. **取得したコネクションを使用して，リソースに接続します。**
4. **使用済みのコネクションをクローズします。**

コネクションプーリングを使用している場合は，手順 2.でプーリングされているコネクションが取得され，手順 4.でコネクションがプールに返却されます。ユーザプログラムでコネクションプーリングを意識したコーディングは必要ありません。

(2)　DI を使用してリソースへの参照を取得する方法

DI を使用してリソースへの参照を取得する場合，DD の定義は不要となります。DI の概要および DI 使用時の注意事項については，「12.4　DI の使用」を参照してください。

(3)　リソースへの接続を実装する場合の注意事項

ユーザプログラムでリソースへのコネクションを取得した場合，使用後には必ずクローズしてください。具体的には，例外などが発生した場合にも必ずクローズするように，finally 節でコネクションをクローズしてください。

なお，finalize メソッドが呼ばれるタイミングは JavaVM の GC のタイミングに依存するので，finalize メソッドでコネクションをクローズする設計にはしないでください。ユーザプログラムでコネクションが正しくクローズされない場合，取得できるコネクションの最大数に達してしまい，それ以上のコネクションが取得できなくなるおそれがあります。

3.3.7　リソースアダプタの設定方法

リソースアダプタの設定方法には，次の二つの方法があります。

- **J2EE リソースアダプタとしてデプロイして設定する方法**
  リソースアダプタを，直接 J2EE サーバにデプロイして設定する方法です。
- **J2EE アプリケーションに含めて設定する方法**
  J2EE アプリケーションに含めて使用するリソースアダプタに対して設定する方法です。

参考

リソースアダプタが DB Connector，TP1 インバウンドアダプタ，または CJMSP リソースアダプタの場合，アプリケーションサーバが提供する Connector 属性ファイルのテンプレートファイルを使用できます。テンプレートファイルを使用すると，リソースアダプタをインポートする前に，Connector 属性ファイルを編集しておくことができます。このため，編集対象の Connector 属性ファイルをサーバ管理コマンド（cjgetrarprop コマンドまたは cjgetresprop コマンド）で取得する操作が不要になります。

Connector 属性ファイルのテンプレートファイルの格納先，およびテンプレートファイル使用時の注意事項については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1.14　Connector 属性ファイルのテンプレートファイル」を参照してください。

! 注意事項

旧バージョンのアプリケーションサーバで使用していたリソースアダプタを使用する場合，リソースアダプタの移行処理が必要です。リソースの移行方法については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「10.9　リソースアダプタの移行」を参照してください。

3.3.8　リソースアダプタの設定の流れ（J2EE リソースアダプタとしてデプロイして使用する場合）

ここでは，リソースアダプタを J2EE リソースアダプタとしてデプロイして使用する場合の，次の流れについて説明します。

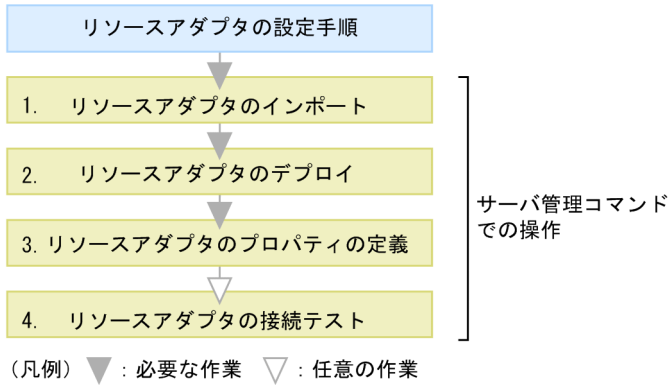
- リソースアダプタの新規設定の流れ
- リソースアダプタの設定変更の流れ
- リソースアダプタの入れ替えの流れ

リソースアダプタの設定には，サーバ管理コマンドを使用します。

(1)　リソースアダプタの新規設定の流れ

データベースやほかのリソースに接続する場合のリソースアダプタの新規設定の流れを次の図に示します。

図 3-2　リソースアダプタの新規設定の流れ



図中の 1.～4.について説明します。

1. サーバ管理コマンドを使用してリソースアダプタをインポートします。

cjimportres コマンドを使用して，リソースアダプタをインポートします。

DB Connector を使用してデータベースに接続する場合と，ほかのリソースアダプタを使用して OpenTP1 などの各種リソースに接続する場合では，インポートする RAR ファイルが異なります。インポートするリソースアダプタについては，「3.3.2　リソースアダプタの種類」を参照してください。

2. サーバ管理コマンドを使用してリソースアダプタをデプロイします。

cjdeployrar コマンドを使用して，リソースアダプタをデプロイします。

リソースアダプタは，デプロイすると J2EE リソースアダプタとして使用できます。**J2EE リソースアダプタ**とは，J2EE サーバに共有スタンドアロンモジュールとして配備したリソースアダプタのことです。サーバ管理コマンドでインポートしたリソースアダプタをデプロイすると，その J2EE サーバ上で動作するすべての J2EE アプリケーションから使用できるようになります。

3. サーバ管理コマンドを使用してリソースアダプタのプロパティを定義します。

cjgetrarprop コマンドで Connector 属性ファイルを取得し，ファイル編集後に，cjsetrarprop コマンドで編集内容を反映させます。

使用する機能ごとに設定するリソースアダプタのプロパティについては，それぞれ次の個所を参照してください。

- リソース接続とトランザクション管理機能を使用するための設定
  「3.4.12　実行環境での設定」
- パフォーマンスチューニングのための機能
  「3.14.10　実行環境での設定」
- フォールトトレランスのための機能
  「3.15.13　実行環境での設定」
- J2EE リソースの別名の設定
  「2.6.6　J2EE リソースの別名の設定」

4. サーバ管理コマンドを使用してリソースアダプタの接続テストを実施します。

cjtestres コマンドを使用して，リソースアダプタの接続テストを実施します。リソースごとの接続テストでの検証内容については，「3.18　リソースへの接続テスト」を参照してください。

! 注意事項

DB Connector for Reliable Messaging と Reliable Messaging を使用してデータベースに接続する場合の接続テストには次のような順序があります。

1. DB Connector for Reliable Messaging を開始します。
2. Reliable Messaging の接続テストを実施します。
3. Reliable Messaging を開始します。
4. DB Connector for Reliable Messaging の接続テストを実施します。

DB Connector for Reliable Messaging の場合の，J2EE リソースアダプタの接続テストについては，マニュアル「Reliable Messaging」の「2.7　DB Connector for Reliable Messaging の機能」を参照してください。

サーバ管理コマンドでの操作については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「3.　サーバ管理コマンドの基本操作」を参照してください。また，コマンドについては，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「2.4　J2EE サーバで使用するリソース操作コマンド」を参照してください。属性ファイルについては，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.　リソースの設定で使用する属性ファイル」を参照してください。

! 注意事項

リソースアダプタを使用する場合，J2EE アプリケーションからリソースアダプタへのリファレンスを解決しておく必要があります。リソースアダプタを使用している J2EE アプリケーションのプロパティを定義するときに，J2EE アプリケーションからリソースアダプタへのリファレンスを解決しておいてください。

参考

次のような場合，リソースアダプタをエクスポート・インポートすることで，効率良くリソースアダプタを設定できます。

- 開発環境で設定したリソースアダプタをエクスポートして，運用環境にインポートして使用する場合
- 運用環境ですでに動いているリソースアダプタをエクスポートして，増設した J2EE サーバにインポートして使用する場合

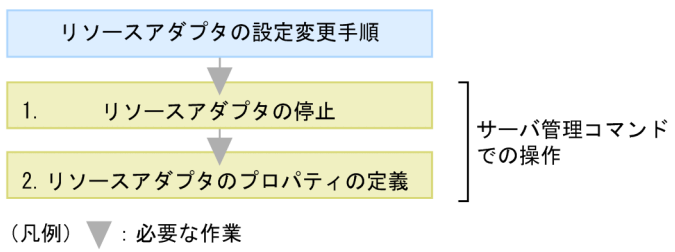
エクスポートとインポートは cjexportrar と cjimportres で実行します。

なお，アプリケーションサーバのバージョンやプラットフォームが異なるホスト間では，リソースアダプタをエクスポート・インポートして使用することはできません。リソースアダプタをエクスポートするホストと，アプリケーションサーバのバージョンやプラットフォームが異なるホストでリソースアダプタを設定する場合は，リソースアダプタを新規に設定してください。

(2)　リソースアダプタの設定変更の流れ

デプロイ済みのリソースアダプタの設定を変更する場合の流れについて説明します。設定変更の流れを次の図に示します。

図 3-3　リソースアダプタの設定変更の流れ



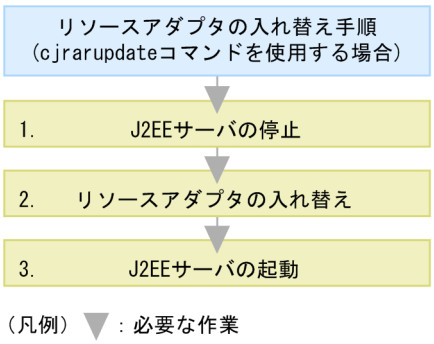
図中の 1.～2.について説明します。

1. サーバ管理コマンドを使用して，リソースアダプタを停止します。
   cjstoprar コマンドを使用してリソースアダプタを停止します。なお，リソースアダプタを停止する前に，そのリソースアダプタを使用している J2EE アプリケーションをすべて停止してください。
2. サーバ管理コマンドを使用してリソースアダプタのプロパティを定義します。
   リソースアダプタはデプロイ済みであるため，cjgetrarprop コマンドを使用して属性ファイルを取得し，ファイル編集後に，cjsetrarprop コマンドで編集内容を反映させます。

(3)　リソースアダプタの入れ替えの流れ

リソースアダプタを入れ替える場合の流れについて説明します。リソースアダプタの入れ替えの流れを次の図に示します。

図 3-4　リソースアダプタの入れ替えの流れ

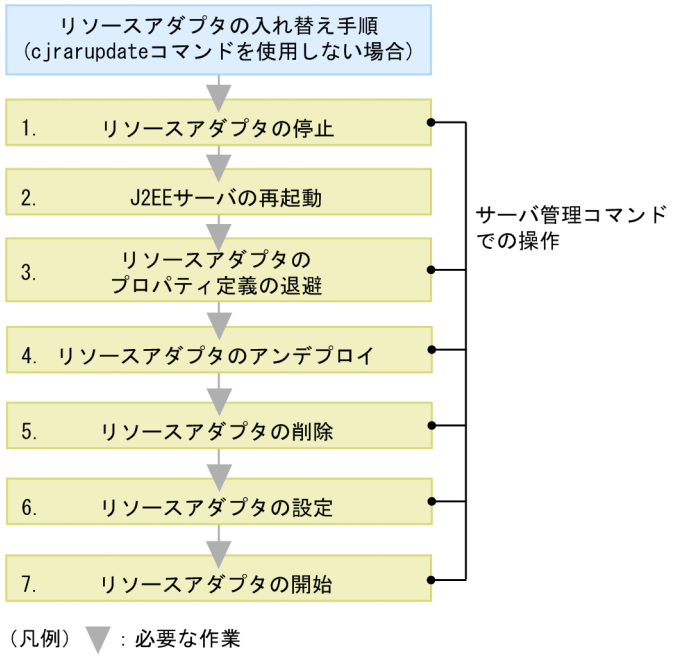


図中の 1.～3.について説明します。

1. J2EE サーバを停止します。
   cjstopsv コマンドを使用して J2EE サーバを停止します。
2. リソースアダプタを入れ替えます。
   cjrarupdate コマンドを使用して，リソースアダプタを入れ替えます。
3. J2EE サーバを起動します。
   cjstartsv コマンドを使用して J2EE サーバを起動します。

cjrarupdate コマンドを使用しないでリソースアダプタを入れ替えることもできます。cjrarupdate コマンドを使用しないでリソースアダプタを入れ替えるときの流れを次の図に示します。

図 3-5　リソースアダプタの入れ替えの流れ（cjrarupdate コマンドを使用しない場合）



図中の 1.～7.について説明します。

1. **サーバ管理コマンドを使用して，リソースアダプタを停止します。**

   cjstoprar コマンドを使用して入れ替えるリソースアダプタを停止します。なお，リソースアダプタを停止する前に，そのリソースアダプタを使用している J2EE アプリケーションをすべて停止してください。

2. **J2EE サーバを再起動します。**

   cjstopsv コマンドを使用して J2EE サーバを停止し，cjstartsv コマンドを使用して J2EE サーバを起動します。

3. **サーバ管理コマンドを使用して，リソースアダプタのプロパティ定義を退避します。**

   リソースアダプタのプロパティ定義を引き継ぐ場合は，cjgetrarprop コマンド，または cjgetresprop コマンドを使用して，リソースアダプタの Connector 属性ファイルを取得します。

4. **サーバ管理コマンドを使用してリソースアダプタをアンデプロイします。**

   cjundeployrar コマンドを使用して，入れ替えるリソースアダプタをアンデプロイします。

5. **サーバ管理コマンドを使用してリソースアダプタを削除します。**

   cjdeleteres コマンドを使用して，入れ替えるリソースアダプタを削除します。

6. **サーバ管理コマンドを使用してリソースアダプタを設定します。**

   「(1)　リソースアダプタの新規設定の流れ」に従って，リソースアダプタを設定します。リソースアダプタのプロパティ定義を引き継ぐ場合は，3.で取得した Connector 属性ファイルを使用します。

7. **サーバ管理コマンドを使用して，リソースアダプタを開始します。**

   cjstartrar コマンドを使用してリソースアダプタを開始します。なお，リソースアダプタを開始したあとに，リソースアダプタを使用する J2EE アプリケーションを開始してください。

3.3.9　リソースアダプタの設定の流れ（J2EE アプリケーションに含めて使用する場合）

リソースアダプタの設定には，サーバ管理コマンドを使用します。ここでは，リソースアダプタを J2EE アプリケーションに含めて使用する場合のリソースアダプタの設定の流れについて説明します。

なお，リソースアダプタをアプリケーションに含めて使用する場合のリソースアダプタの設定方法には，次の二つの方法があります。

- リソースアダプタを含めた J2EE アプリケーションを J2EE サーバにインポートする方法
- J2EE サーバにインポート済みの J2EE アプリケーションにリソースアダプタを追加する方法

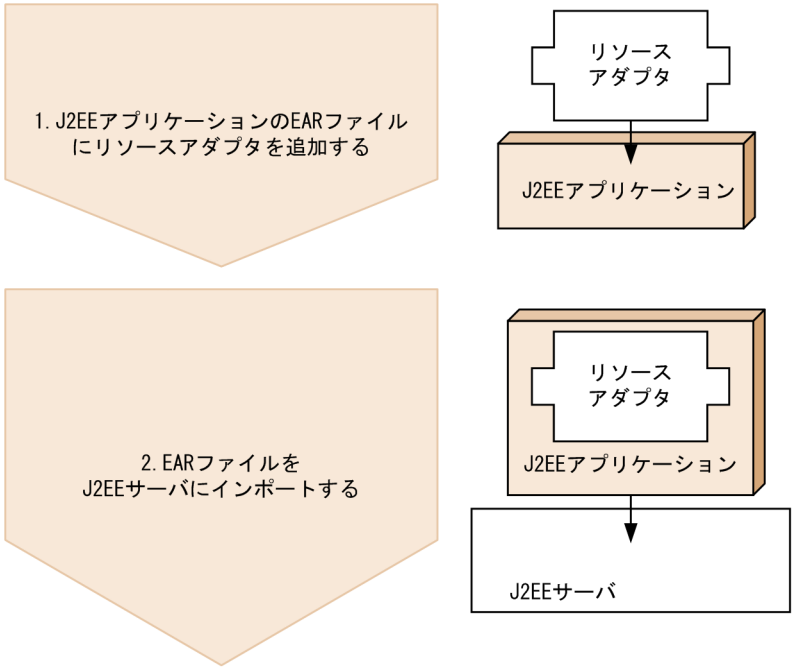
ここでは，それぞれの方法について説明します。

なお，J2EE アプリケーションに含められるリソースアダプタには，制限があります。J2EE アプリケーションに含められるリソースアダプタについては，「3.3.3　リソースアダプタの使用方法」を参照してください。

(1)　リソースアダプタを含めた J2EE アプリケーションを J2EE サーバにインポートする方法

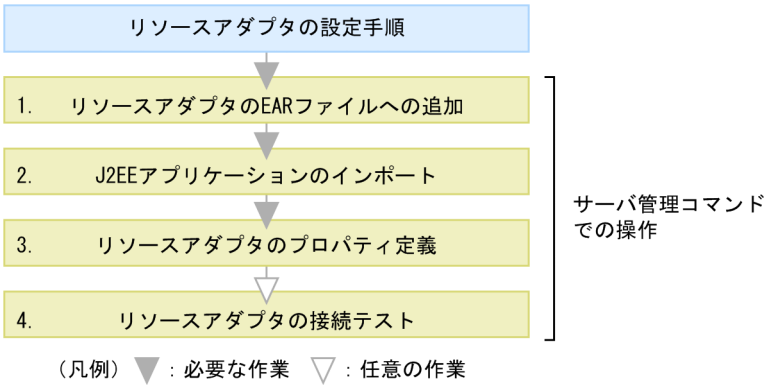
リソースアダプタの RAR ファイルを含めて J2EE アプリケーションを作成して，その EAR ファイルを J2EE サーバにインポートする方法です。この方法の概要を次の図に示します。

図 3-6　リソースアダプタを含めた J2EE アプリケーションを J2EE サーバにインポートする方法の概要



リソースアダプタを含めた J2EE アプリケーションを J2EE サーバにインポートする流れを次の図に示します。

図 3-7　リソースアダプタを含めた J2EE アプリケーションを J2EE サーバにインポートする流れ



図中の 1.～4.について説明します。

1. **サーバ管理コマンドを使用してリソースアダプタの RAR ファイルを J2EE アプリケーションの EAR ファイルに追加します。**

   cjaddapp コマンドを使用して，リソースアダプタの RAR ファイルを J2EE アプリケーションの EAR ファイルに追加します。

   EAR ファイルに含められるリソースアダプタには，制限があります。EAR ファイルに含められるリソースアダプタについては，「3.3.3　リソースアダプタの使用方法」を参照してください。

2. **サーバ管理コマンドを使用して J2EE アプリケーションをインポートします。**

   cjimportapp コマンドを使用して，J2EE アプリケーションをインポートします。

3. **サーバ管理コマンドを使用してリソースアダプタのプロパティを定義します。**

   cjgetappprop コマンドで Connector 属性ファイルを取得し，ファイル編集後に，cjsetappprop コマンドで編集内容を反映させます。

   使用する機能ごとに設定するリソースアダプタのプロパティについては，それぞれ次の個所を参照してください。

   - リソース接続とトランザクション管理機能を使用するための設定
     「3.4.12　実行環境での設定」
   - パフォーマンスチューニングのための機能
     「3.14.10　実行環境での設定」
   - フォールトトレランスのための機能
     「3.15.13　実行環境での設定」
   - J2EE リソースの別名の設定
     「2.6.6　J2EE リソースの別名の設定」

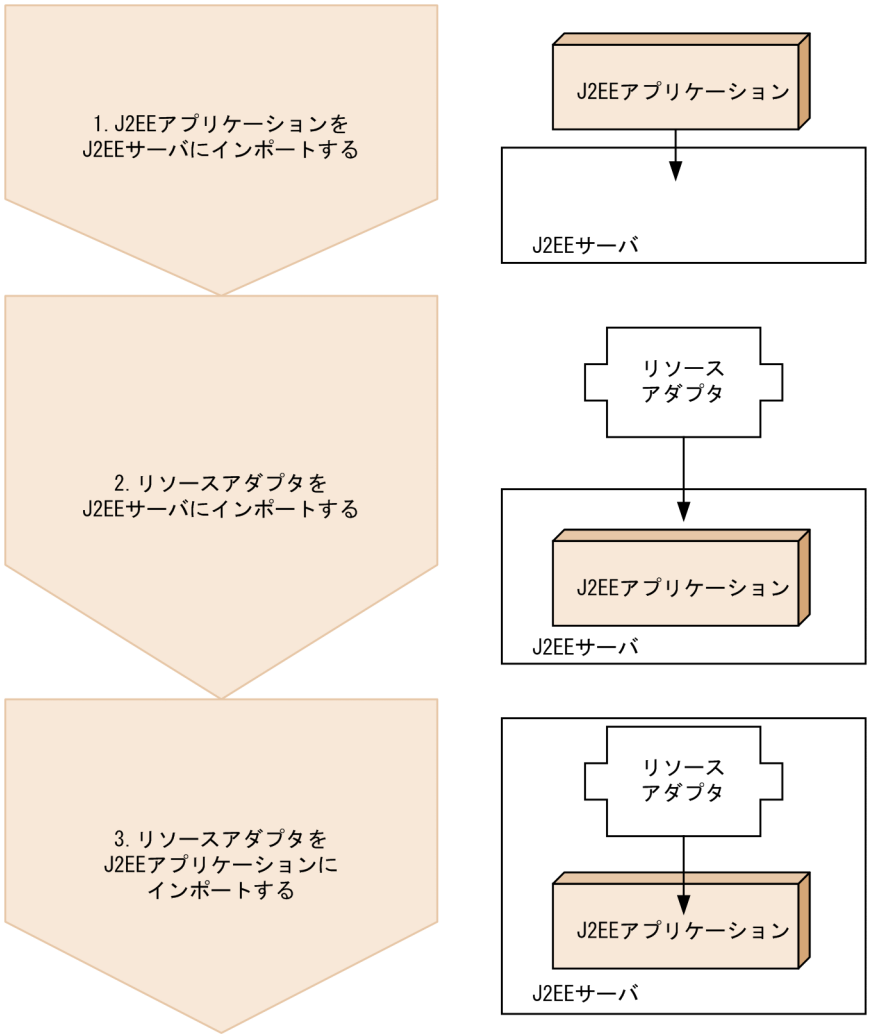
4. **サーバ管理コマンドを使用してリソースアダプタの接続テストを実施します。**

   cjtestres コマンドを使用して，リソースアダプタの接続テストを実施します。リソースごとの接続テストでの検証内容については，「3.18　リソースへの接続テスト」を参照してください。

(2)　インポート済みの J2EE アプリケーションにリソースアダプタを追加する方法

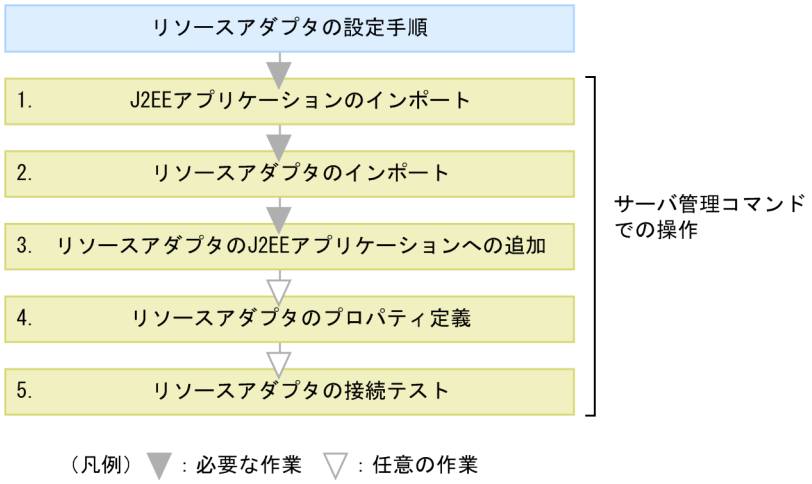
リソースアダプタ（RAR ファイル）を J2EE サーバにインポートしてから，同じ J2EE サーバにインポート済みの J2EE アプリケーションにリソースアダプタを追加する方法です。この方法の概要を次の図に示します。

図 3-8　インポート済みの J2EE アプリケーションにリソースアダプタを追加する方法の概要



インポート済みの J2EE アプリケーションにリソースアダプタを追加する流れを次の図に示します。

図 3-9　インポート済みの J2EE アプリケーションにリソースアダプタを追加する流れ



図中の 1.～5.について説明します。

1. **サーバ管理コマンドを使用して J2EE アプリケーションをインポートします。**
   cjimportapp コマンドを使用して，J2EE アプリケーションをインポートします。
2. **サーバ管理コマンドを使用してリソースアダプタをインポートします。**
   cjimportres コマンドを使用して，リソースアダプタをインポートします。
3. **サーバ管理コマンドを使用してリソースアダプタを J2EE アプリケーションに追加します。**
   cjaddapp コマンドを使用して，リソースアダプタを J2EE アプリケーションに追加します。
4. **サーバ管理コマンドを使用してリソースアダプタのプロパティを定義します。**
   cjgetappprop コマンドで Connector 属性ファイルを取得し，ファイル編集後に，cjsetappprop コマンドで編集内容を反映させます。
   使用する機能ごとに設定するリソースアダプタのプロパティについては，それぞれ次の個所を参照してください。
   - リソース接続とトランザクション管理機能を使用するための設定
     「3.4.12　実行環境での設定」
   - パフォーマンスチューニングのための機能
     「3.14.10　実行環境での設定」
   - フォールトトレランスのための機能
     「3.15.13　実行環境での設定」
   - J2EE リソースの別名の設定
     「2.6.6　J2EE リソースの別名の設定」
5. **サーバ管理コマンドを使用してリソースアダプタの接続テストを実施します。**
   cjtestres コマンドを使用して，リソースアダプタの接続テストを実施します。リソースごとの接続テストでの検証内容については，「3.18　リソースへの接続テスト」を参照してください。

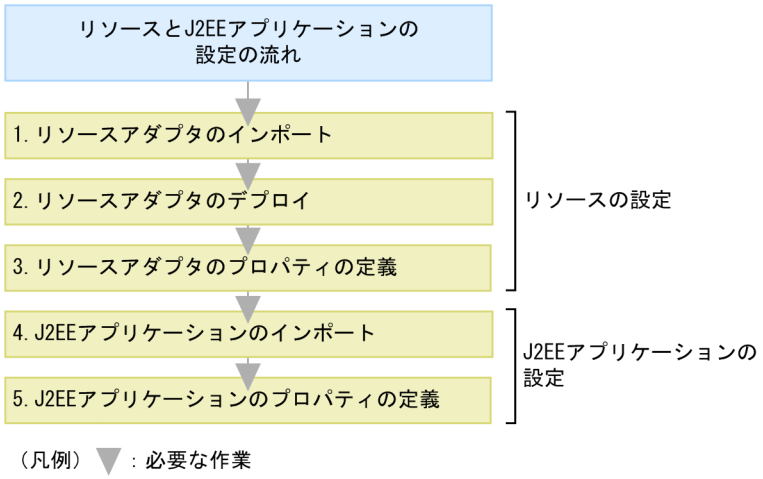
3.3.10　リソースアダプタの設定の流れ（Inbound で使用する場合）

ここでは，メッセージインフローを実行する場合のリソースアダプタと J2EE アプリケーションの設定の流れについて説明します。設定の流れは，リソースアダプタを J2EE サーバに直接デプロイして使用するか，J2EE アプリケーションに含めて使用するかによって異なります。

(1)　リソースアダプタを J2EE サーバに直接デプロイして使用する場合

リソースアダプタを J2EE サーバに直接デプロイして使用する場合の設定の流れを次の図に示します。

図 3-10　リソースアダプタを J2EE サーバに直接デプロイして使用する場合の設定の流れ



図中の 1.～5.について説明します。ここでは，サーバ管理コマンドによる操作を示します。

1. Connector 1.5 仕様に準拠したリソースアダプタをインポートします。

   cjimportres コマンドに-type rar を指定して実行します。

2. リソースアダプタをデプロイします。

   cjdeployrar コマンドを実行します。

3. リソースアダプタのプロパティを定義します。

   cjgetrarprop コマンドで Connector 属性ファイルを取得します。ファイル編集後に，cjsetrarprop コマンドで編集内容を反映させます。

   ここでは，管理対象オブジェクトを設定します。設定内容については，「3.16.8(2) 管理対象オブジェクトの設定」を参照してください。

4. Message-driven Bean を含む J2EE アプリケーションをインポートします。

   cjimportapp コマンドを使用します。

5. J2EE アプリケーションのプロパティを定義します。

   cjgetappprop コマンドに-type all を指定して実行し，アプリケーション統合属性ファイルを取得します。ファイル編集後に，cjsetappprop コマンドに-type all を指定して実行し，編集内容を反映させます。

   ここでは，次の項目を設定します。

   - Message-driven Bean とリソースアダプタの対応づけ

     設定内容については「3.16.8(3) Message-driven Bean とリソースアダプタの対応づけの設定」を参照してください。

   - Message-driven Bean が使用するインタフェース

     設定内容については「3.16.8(4) Message-driven Bean が使用するインタフェースの設定」を参照してください。

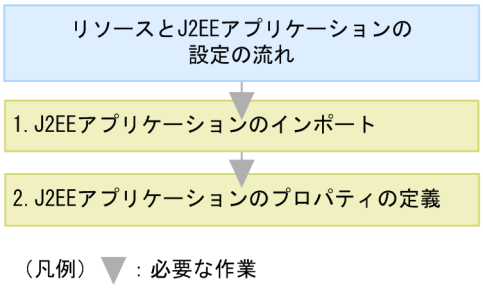
   - ActivationSpec の設定

     設定内容については「3.16.8(5) ActivationSpec の設定」を参照してください。

(2)　リソースアダプタを J2EE アプリケーションに含めて使用する場合

リソースアダプタを J2EE アプリケーションに含めて使用する場合の設定の流れを次の図に示します。

図 3-11　リソースアダプタを J2EE アプリケーションに含めて使用する場合の設定の流れ



図中の 1.～2.について説明します。なお，ここでは，サーバ管理コマンドによる操作を示します。

1. Message-driven Bean を含む J2EE アプリケーションをインポートします。

   cjimportapp コマンドを使用します。

2. J2EE アプリケーションのプロパティを定義します。

   cjgetappprop コマンドに-type all を指定して実行し，アプリケーション統合属性ファイルを取得します。ファイル編集後に，cjsetappprop コマンドに-type all を指定して実行し，編集内容を反映させます。

   ここでは，次の項目を設定します。

   - 管理対象オブジェクトの情報

     設定内容については「3.16.8(2) 管理対象オブジェクトの設定」を参照してください。

   - Message-driven Bean とリソースアダプタの対応づけ

     設定内容については「3.16.8(3) Message-driven Bean とリソースアダプタの対応づけの設定」を参照してください。

   - Message-driven Bean が使用するインタフェース

     設定内容については「3.16.8(4) Message-driven Bean が使用するインタフェースの設定」を参照してください。

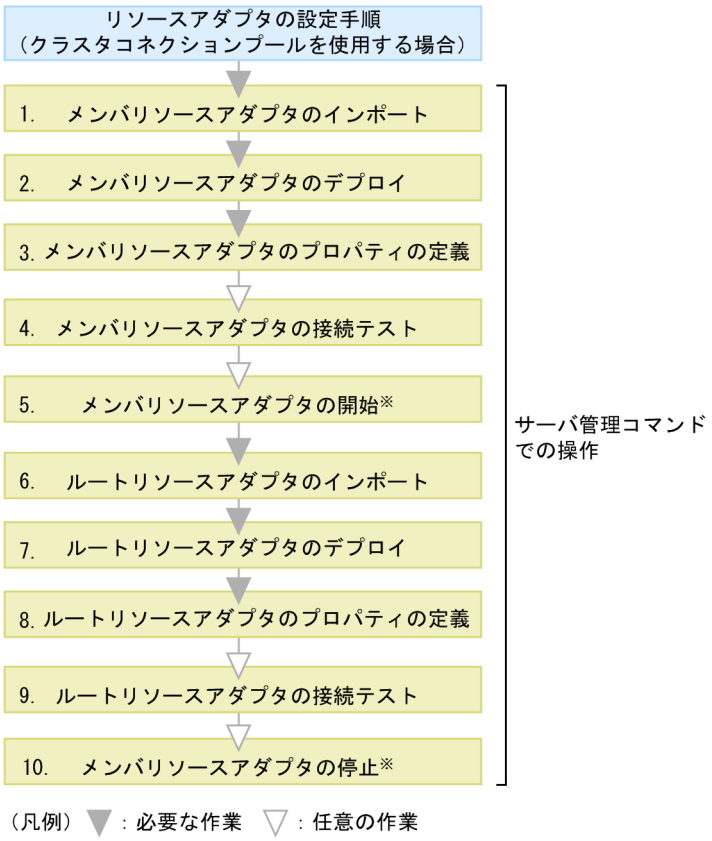
   - ActivationSpec の設定

     設定内容については「3.16.8(5) ActivationSpec の設定」を参照してください。

3.3.11　リソースアダプタの設定の流れ（クラスタコネクションプールを使用する場合）

コネクションプールをクラスタ化している場合の，データベースに接続するときのリソースアダプタの設定の流れを次の図に示します。

図 3-12　クラスタコネクションプールでのリソースアダプタの設定の流れ



注※　ルートリソースアダプタの接続テストを実施する場合に必要な作業です。

図中の 1.～10.について説明します。

1. サーバ管理コマンドを使用してメンバリソースアダプタをインポートします。

cjimportres コマンドを使用して，メンバリソースアダプタをインポートします。

インポートするリソースアダプタについては，「3.3.2　リソースアダプタの種類」を参照してください。

2. サーバ管理コマンドを使用してメンバリソースアダプタをデプロイします。

cjdeployrar コマンドを使用して，メンバリソースアダプタをデプロイします。

3. サーバ管理コマンドを使用してメンバリソースアダプタのプロパティを定義します。

cjgetrarprop コマンドで Connector 属性ファイルを取得し，ファイル編集後に，cjsetrarprop コマンドで編集内容を反映させます。

メンバリソースアダプタとルートリソースアダプタのプロパティ定義では設定できる項目が異なります。メンバリソースアダプタおよびルートリソースアダプタで設定できるプロパティ定義の項目については，「3.17　クラスタコネクションプール機能」を参照してください。

使用する機能ごとに設定するリソースアダプタのプロパティについては，それぞれ次の個所を参照してください。

- リソース接続とトランザクション管理機能を使用するための設定
  「3.4.12　実行環境での設定」
- パフォーマンスチューニングのための機能
  「3.14.10　実行環境での設定」
- フォールトトレランスのための機能

「3.15.13　実行環境での設定」

- J2EE リソースの別名の設定
  「2.6.6　J2EE リソースの別名の設定」

4. **サーバ管理コマンドを使用してメンバリソースアダプタの接続テストを実施します。**
   cjtestres コマンドを使用して，メンバリソースアダプタの接続テストを実施します。
   メンバリソースアダプタの接続テストでの検証内容については，「3.18　リソースへの接続テスト」を参照してください。
   また，1.～4.までの流れを，メンバリソースアダプタの数だけ繰り返します。
5. **サーバ管理コマンドを使用してメンバリソースアダプタを開始します。**
   ルートリソースアダプタの接続テストを実施する場合には，あらかじめメンバリソースアダプタを開始しておいてください。cjstartrar コマンドを使用して，メンバリソースアダプタを開始します。
6. **サーバ管理コマンドを使用してルートリソースアダプタをインポートします。**
   cjimportres コマンドを使用して，ルートリソースアダプタをインポートします。
   インポートするリソースアダプタについては，「3.3.2　リソースアダプタの種類」を参照してください。
7. **サーバ管理コマンドを使用してルートリソースアダプタをデプロイします。**
   cjdeployrar コマンドを使用してルートリソースアダプタをデプロイします。
8. **サーバ管理コマンドを使用してルートリソースアダプタのプロパティを定義します。**
   cjgetrarprop コマンドで Connector 属性ファイルを取得し，ファイル編集後に，cjsetrarprop コマンドで編集内容を反映させます。
9. **サーバ管理コマンドを使用してルートリソースアダプタの接続テストを実施します。**
   cjtestres コマンドを使用して，ルートリソースアダプタの接続テストを実施します。
   ルートリソースアダプタの接続テストでの検証内容については，「3.18　リソースへの接続テスト」を参照してください。
10. **サーバ管理コマンドを使用してメンバリソースアダプタを停止します。**
    ルートリソースアダプタの接続テストを実施した場合には，メンバリソースアダプタを停止してください。cjstoprar コマンドを使用して，メンバリソースアダプタを停止します。

サーバ管理コマンドでの操作については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「3.　サーバ管理コマンドの基本操作」を参照してください。また，コマンドについては，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「2.4　J2EE サーバで使用するリソース操作コマンド」を参照してください。属性ファイルについては，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.　リソースの設定で使用する属性ファイル」を参照してください。

! 注意事項

コネクションプールをクラスタ化する場合，J2EE アプリケーションからルートリソースアダプタへのリファレンスを解決しておく必要があります。ルートリソースアダプタを使用している J2EE アプリケーションのプロパティを定義するときに，J2EE アプリケーションからルートリソースアダプタへのリファレンスを解決しておいてください。

3.3.12　リソースアダプタ以外を使用する接続の設定

リソースアダプタ以外の，JavaBeans リソースや JavaMail を使用する場合の設定については，それぞれ次の個所を参照してください。

- JavaBeans リソースを使用する場合
  「3.12.4　JavaBeans リソースの設定」を参照してください。
- JavaMail を使用する場合
  メールコンフィグレーションは SMTP サーバとの接続だけで使用します。
  メールの受信には POP3 サーバを利用します。IMAP サーバは利用できません。
  詳細は「3.11　SMTP サーバとの接続」を参照してください。

3.3.13　リソースアダプタについての注意事項

ここでは，リソースアダプタについての注意事項を説明します。

デプロイ方法によって使用できない機能

リソースアダプタを J2EE アプリケーションに含めて使用する場合，J2EE アプリケーションに含めたリソースが更新されても，次の機能は有効になりません。

- J2EE アプリケーションの更新検知
- J2EE アプリケーションのリロード

詳細は，「13.8.13　リロードの注意事項および制限事項」を参照してください。

リソースアダプタの表示名についての注意事項

J2EE アプリケーション内の EJB や WAR は，J2EE リソースアダプタとしてデプロイするリソースアダプタ，および J2EE アプリケーションに含めて使用するリソースアダプタを同時に使用できます。ただし，一つの J2EE サーバで同じ表示名のリソースアダプタを二つ以上使用することはできません。一つの J2EE サーバに同じ表示名のリソースアダプタを二つ以上使用しようとすると，エラーメッセージが出力されて，リソースアダプタの開始に失敗します。リソースアダプタの開始に失敗する手順の例を次に示します。

例 1.

1.「Rar1」という表示名のリソースアダプタを J2EE サーバにデプロイする。
2.「Rar1」という表示名のリソースアダプタを含めた J2EE アプリケーションをインポートする。

例 2.

1.「Rar2」という表示名のリソースアダプタを J2EE サーバにデプロイする。
2.「Rar2」という表示名のリソースアダプタを，インポート済みの J2EE アプリケーションに追加する。

例 3.

1.「Rar3」という表示名のリソースアダプタを含めた J2EE アプリケーションをインポートする。
2.「Rar3」という表示名のリソースアダプタを J2EE サーバにデプロイする。

リソースアダプタのオプショナル名についての注意事項

同じオプショナル名で複数のリソースアダプタをデプロイしている場合，エラーメッセージが出力されて，リソースアダプタの開始に失敗します。

リソースアダプタの開始処理についての注意事項

J2EE サーバの起動途中でリソースアダプタが開始処理に失敗した場合，開始状態から停止状態に遷移します。この場合，J2EE サーバの次回起動時にも開始されません。また，該当のリソースアダプタを使用しているアプリケーションの開始に失敗します。

このような場合は，エラー情報を参照してエラーの原因を取り除いたあとに，次の対処をしてください。

リソースアダプタのトランザクションサポートレベルが LocalTransaction または NoTransaction の場合

1. 停止状態に遷移したリソースアダプタを開始してください。
2. リソースアダプタを使用しているアプリケーションを開始してください。

リソースアダプタのトランザクションサポートレベルが XATransaction の場合

1. 停止状態に遷移したリソースアダプタを開始してください。
2. 未決着トランザクションをリカバリするため，J2EE サーバを再起動してください。
3. リソースアダプタを使用しているアプリケーションを開始してください。

そのほかの注意事項

URL コネクションの Resource Factory の参照機能は未サポートです。

3.4　トランザクション管理

この節では，トランザクション管理の概要について説明します。

リソース接続時のトランザクションを管理する方法には，アプリケーションサーバで管理する方法と，ユーザが直接管理する方法があります。

アプリケーションサーバでトランザクションを管理する場合，アプリケーションサーバのトランザクションマネージャを使用してトランザクションを管理できます。

この節の構成を次の表に示します。

表 3-10　この節の構成（トランザクション管理）

分類	タイトル	参照先
解説	リソース接続でのトランザクション管理の方法	3.4.1
	ローカルトランザクションとグローバルトランザクション	3.4.2
	リソースごとに使用できるトランザクションの種類	3.4.3
	トランザクションサービスで提供する機能	3.4.4
	システム例外発生時のトランザクションの動作	3.4.5
	トランザクションマネージャの取得	3.4.6
実装※	コンテナ管理のトランザクション（CMT）を使用する場合の処理概要と留意点	3.4.7
	UserTransaction インタフェースを使用する場合の処理概要と留意点	3.4.8
	リソース固有のトランザクション管理インタフェースを使用する場合の処理概要と留意点	3.4.9
	トランザクションを使用しない場合の処理概要と留意点	3.4.10
	JTA によるトランザクション実装時の注意事項	3.4.11
設定	実行環境での設定	3.4.12

注　「運用」について，この機能固有の説明はありません。

注※　EJB クライアントでのトランザクションの実装方法については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「3.5　EJB クライアントアプリケーションでのトランザクションの実装」を参照してください。

ポイント

リソースのうち，SMTP サーバおよび JavaBeans リソースについては，トランザクション管理の対象外です。

参考

EJB クライアントアプリケーションでもトランザクションを開始できます。EJB クライアントアプリケーションでトランザクションを開始する場合の注意事項については，「3.20　EJB クライアントアプリケーションでトランザクションを開始する場合の注意事項」を参照してください。

3.4.1　リソース接続でのトランザクション管理の方法

リソース接続でのトランザクションの管理方法には，アプリケーションサーバが管理する方法と，アプリケーションサーバが管理しない方法（ユーザが直接管理する方法）があります。ここでは，それぞれのトランザクションの管理方法について説明します。

(1)　アプリケーションサーバが管理するトランザクション

アプリケーションサーバのトランザクションマネージャ経由でトランザクションを管理する方法です。ユーザは，javax.transaction.UserTransaction インタフェースの API を操作するか，EJB メソッドの CMT 属性を設定することで，トランザクションを管理します。

- UserTransaction インタフェースによる管理

  サーブレット，JSP，または EJB（BMT）から，javax.transaction.UserTransaction インタフェースの API を発行することで，トランザクションを管理できます。BMT の詳細については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「2.7.2　BMT」を参照してください。

- EJB の CMT 属性による管理

  EJB（CMT）のメソッド単位で指定するトランザクション属性によって，トランザクションを管理できます。CMT の詳細については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「2.7.3　CMT」を参照してください。

アプリケーションサーバがトランザクションを管理する場合，トランザクションの種類としてローカルトランザクションまたはグローバルトランザクションを選択できます。アプリケーションサーバが管理するトランザクションの種類については，「3.4.2　ローカルトランザクションとグローバルトランザクション」を参照してください。

(2)　ユーザが直接管理するトランザクション（アプリケーションサーバが管理しないトランザクション）

リソース固有の API によって，ユーザが直接トランザクションを管理する方法です。例えば，データベースに JDBC インタフェースで接続する場合，java.sql.Connection インタフェースの setAutoCommit()，commit()，rollback()などの API をユーザが直接操作します。

3.4.2　ローカルトランザクションとグローバルトランザクション

アプリケーションサーバが管理するトランザクションを使用する場合，アプリケーションサーバのトランザクションマネージャと，リソースを管理するリソースマネジャ（DBMS など）が連携して，トランザクションを管理します。この場合，トランザクションの種類として，ローカルトランザクションとグローバルトランザクションのどちらかを選択します。

ローカルトランザクションと，グローバルトランザクションについて説明します。

(1)　ローカルトランザクション

トランザクション管理を行うリソースが一つだけの場合に，ローカルトランザクションを使用します。ローカルトランザクションを使用する場合，トランザクションの決着はリソースマネジャが行います。

(2)　グローバルトランザクション

トランザクション管理を行うリソースが複数ある場合に，グローバルトランザクションを使用します。グローバルトランザクションを使用する場合，トランザクションマネージャが複数のリソースのトランザク

ションを調整し，整合性が崩れないように決着します。トランザクションの決着には，2 フェーズコミットプロトコルが使用されます。なお，グローバルトランザクション使用時には，インプロセストランザクションサービスを使用します。インプロセストランザクションサービスについては，「3.14.6　インプロセストランザクションサービス」を参照してください。

グローバルトランザクションには比較的高い処理コストが掛かりますので，トランザクション管理を行うリソースが一つだけの場合には，ローカルトランザクションを使用することをお勧めします。

なお，デフォルトではライトトランザクション機能※が有効になっているため，グローバルトランザクションを使用することはできません。グローバルトランザクションを使用するには，ライトトランザクション機能を無効にする必要があります。

注※

ライトトランザクションの詳細については，「3.14.5　ライトトランザクション」を参照してください。

(3)　ライトトランザクション機能とトランザクションの管理方法の関係

ライトトランザクション機能とは，ローカルトランザクションに，最適化された環境を提供する機能です。

トランザクションの管理方法とライトトランザクションの対応を次の表に示します。

表 3-11　トランザクションの管理方法とライトトランザクションの対応

トランザクションの管理方法		ライトトランザクション有効	ライトトランザクション無効
アプリケーションサーバが管理するトランザクション	ローカルトランザクション	○	○
	グローバルトランザクション	×	○
アプリケーションサーバが管理しないトランザクション		○	○

（凡例）　○：使用できる　　×：使用できない

ライトトランザクション機能については，「3.14.5　ライトトランザクション」を参照してください。

3.4.3　リソースごとに使用できるトランザクションの種類

ここでは，次に示すリソースごとに使用できるトランザクションの種類について説明します。

- データベース（接続方法：DB Connector）
- データベース上のキュー（接続方法：DB Connector for Reliable Messaging と Reliable Messaging）
- OpenTP1（Outbound での接続）（接続方法：TP1 Connector または TP1/Message Queue - Access）
- OpenTP1（Inbound での接続）（接続方法：TP1 インバウンドアダプタ）
- CJMSP ブローカー（接続方法：CJMSP リソースアダプタ）
- そのほかのリソース（接続方法：Connector 1.5 仕様に準拠したリソースアダプタ）

それぞれのリソースで使用できるトランザクションの種類は，次の項目の設定内容によって決まります。

リソースアダプタ単位で設定するトランザクションサポートレベル

次に示す 3 種類のトランザクションサポートレベルごとに，使用できるトランザクションの種類が異なります。

- NoTransaction
  リソースをトランザクション管理しません。
- LocalTransaction
  リソースをローカルトランザクションでトランザクション管理します。
- XATransaction
  リソースをグローバルトランザクションでトランザクション管理します。

なお，トランザクションサポートレベルの設定は，リソースアダプタのプロパティとして設定します。リソースアダプタの設定については，「3.4.12　実行環境での設定」を参照してください。

ライトトランザクション機能の有効／無効

ライトトランザクション機能を有効にしているか無効にしているかによって，使用できるトランザクションの種類が異なります。

(1)　データベース接続の場合

接続方法，トランザクションサポートレベルの対応で決定される，トランザクションの種類を次の表に示します。

表 3-12　使用できるトランザクションの種類（データベース接続の場合）

接続方法	トランザクションサポートレベル	ライトトランザクション機能	
		有効	無効
DB Connector (DBConnector_HiRDB_Type4_CP.rar) (DBConnector_Oracle_CP.rar) (DBConnector_SQLServer_CP.rar) (DBConnector_CP_ClusterPool_Root.rar) (DBConnector_Oracle_CP_ClusterPool_Member.rar)	NoTransaction	なし	なし
	LocalTransaction	ローカル	ローカル
DB Connector (DBConnector_HiRDB_Type4_XA.rar) (DBConnector_Oracle_XA.rar)	XATransaction	−	グローバル

（凡例）

グローバル：グローバルトランザクション

ローカル：ローカルトランザクション

なし：トランザクション管理なし

−：指定できない

(2)　データベース上のキューとの接続の場合

接続方法，トランザクションサポートレベルの対応で決定される，トランザクションの種類を次の表に示します。

表 3-13　使用できるトランザクションの種類（データベース上のキューとの接続の場合）

接続方法	トランザクションサポートレベル	ライトトランザクション機能	
		有効	無効
DB Connector for Reliable Messaging と Reliable Messaging （DBConnector_HiRDB_Type4_CP_Cosminexus_RM.rar） （DBConnector_Oracle_CP_Cosminexus_RM.rar）	NoTransaction	なし	なし
	LocalTransaction	ローカル	ローカル
DB Connector for Reliable Messaging と Reliable Messaging （DBConnector_HiRDB_Type4_XA_Cosminexus_RM.rar） （DBConnector_Oracle_XA_Cosminexus_RM.rar）	XATransaction	－	グローバル

（凡例）

グローバル：グローバルトランザクション

ローカル：ローカルトランザクション

なし：トランザクション管理なし

－：指定できない

(3)　OpenTP1 接続の場合（Outbound での接続）

接続方法，トランザクションサポートレベルの対応で決定される，トランザクションの種類を次の表に示します。

表 3-14　使用できるトランザクションの種類（OpenTP1 接続の場合（Outbound での接続））

接続方法	トランザクションサポートレベル	ライトトランザクション機能	
		有効	無効
TP1 Connector	NoTransaction	なし	なし
	LocalTransaction	ローカル	ローカル
	XATransaction	－	グローバル
TP1/Message Queue - Access	NoTransaction	なし	なし
	LocalTransaction	ローカル	ローカル
	XATransaction	－	グローバル

（凡例）

グローバル：グローバルトランザクション

ローカル：ローカルトランザクション

なし：トランザクション管理なし

－：指定できない

(4)　OpenTP1 接続の場合（Inbound での接続）

TP1 インバウンドアダプタで使用できるトランザクションの種類については，「3.13.2　そのほかのリソースとの接続で使用できる機能」を参照してください。

(5)　CJMSP ブローカー接続の場合

接続方法，トランザクションサポートレベルの対応で決定される，トランザクションの種類を次の表に示します。

表 3-15　使用できるトランザクションの種類（CJMSP ブローカー接続の場合）

接続方法	トランザクションサポートレベル	ライトトランザクション機能	
		有効	無効
CJMSP リソースアダプタ	NoTransaction	なし	なし
	LocalTransaction	ローカル	ローカル
	XATransaction	－	－

（凡例）

ローカル：ローカルトランザクション

なし：トランザクション管理なし

－：指定できない

(6)　そのほかのリソースの場合

Connector 1.5 仕様に準拠したリソースアダプタで使用できるトランザクションの種類については，「3.13.2　そのほかのリソースとの接続で使用できる機能」を参照してください。

3.4.4　トランザクションサービスで提供する機能

トランザクションサービスでは，次に示す機能を提供します。なお，アプリケーションサーバでは，トランザクションサービスは J2EE サーバのインプロセスで起動します。

- **グローバルトランザクションの開始，コミット決着，ロールバック決着の制御**

  2 フェーズコミットプロトコルによってトランザクション処理を制御します。2 フェーズコミットプロトコルとは，同期点での処理をプリペア処理（資源のアップデート準備）とコミット処理（資源のアップデート処理）という 2 段階に分ける方法です。2 フェーズコミットプロトコルでは，DBMS などの複数のリソースオブジェクトに対して同期を取り，コミットまたはロールバックができます。また，2 フェーズコミットプロトコルではトランザクション処理中に障害が発生した場合でも，すべてのリソースオブジェクトを矛盾なく自動的にロールバックできます。

- **トランザクションコンテキストの伝播（RMI-IIOP による Enterprise Bean 呼び出し）**

  グローバルトランザクションの状態を表すトランザクションコンテキストを管理します。例えば，サーブレット/JSP などのクライアントがリモートの Enterprise Bean のメソッドを呼び出したとき，クライアント側のトランザクションコンテキストをサーバ側の Enterprise Bean に伝達します。

- **ステータスファイルを使用したトランザクション情報の管理と障害発生後の J2EE サーバの再起動によるシステム回復**

  システム障害で J2EE サーバが停止したときに実行中だったアプリケーションプログラムのトランザクション処理を回復して，ロールバックまたはコミットします。トランザクションをロールバックするか，コミットするかは，トランザクション処理がどこまで進んでいたかで決まります。トランザクショ

ン処理が，2 フェーズコミットのうち，1 フェーズ目の完了前まで進んでいる場合には，グローバルトランザクションをロールバックします。2 フェーズコミットのうちの 1 フェーズ目が完了している場合には，ルートトランザクションブランチでの決定に従って，グローバルトランザクションをロールバック，またはコミットします。

- **ステータスファイルの二重化**

  ステータスファイルの二重化の機能を使用した場合，一方のステータスファイルが配置されているディスクで障害が発生すると，他方のステータスファイルでトランザクション回復処理が実行されます※。ただし，この機能を利用すると，二重にディスクアクセスするため，オンライン処理の応答時間は遅くなります。

  注※

  オンライン処理は継続できません。

- **リソースマネジャでのヒューリスティック決着時のエラー通知**

  データベースなどのリソースマネジャでのヒューリスティック決着を検知したとき，メッセージによってエラーを通知します。

3.4.5　システム例外発生時のトランザクションの動作

EJB 呼び出し時に，呼び出し先でシステム例外が発生したときの呼び出し元のトランザクションの挙動は，システム定義によって次のように変わります。

(1)　呼び出し元のトランザクションを呼び出し先が引き継ぐ場合（呼び出し先のトランザクション属性が CMT の Required，Supports，Mandatory の場合）

呼び出し先でシステム例外が戻ってきた場合，トランザクションはコンテナによってロールバックされます。この動作は，EJB 仕様で規定されています。

(2)　呼び出し元のトランザクションを呼び出し先が引き継がない場合（呼び出し先のトランザクション属性が BMT，または CMT の NotSupported，RequiresNew，Never の場合）

呼び出し元，呼び出し先のトランザクションは，それぞれ次のように動作します。

呼び出し元のトランザクション

- EJB のリモートインタフェースでのリモート呼び出し時

  ライトトランザクションが無効の場合：

  トランザクションは OTS によってロールバックにマークされます。

  ライトトランザクションが有効の場合：

  トランザクションはロールバックにマークされません。

- EJB のリモートインタフェースでのローカル呼び出しの最適化時

  usrconf.properties の ejbserver.distributedtx.rollbackClientTxOnSystemException キーの値によって動作が異なります。

  true の場合：

  トランザクションはロールバックにマークされます。

  false の場合：

  トランザクションはロールバックにマークされません。

- EJB のローカルインタフェース呼び出し時

トランザクションはロールバックにマークされます。

呼び出し先のトランザクション

トランザクションはコンテナによってロールバックされます。この動作は，EJB 仕様で規定されています。

なお，ローカル呼び出しの最適化については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「2.13.1　EJB のリモートインタフェースでのローカル呼び出しの最適化」を参照してください。

3.4.6　トランザクションマネージャの取得

トランザクションマネージャ（javax.transaction.TransactionManager または javax.transaction.Transaction）は，トランザクションを管理するための API を提供します。トランザクションマネージャの API を使用するフレームワークを使用する場合は, JNDI を使用してトランザクションマネージャを取得できます。トランザクションマネージャを取得するには，「java:comp/TransactionManager」の名前でルックアップします。

アプリケーションサーバがサポートするトランザクションマネージャの API，および Synchronization を使用する場合の注意について説明します。

(1)　サポートする API

アプリケーションサーバがサポートするトランザクションマネージャの API を次の表に示します。

表 3-16　アプリケーションサーバがサポートするトランザクションマネージャの API

インタフェース	メソッド	使用可否
javax.transaction.TransactionManager	begin	○
	commit	○
	getStatus	○
	getTransaction	○
	resume	○
	rollback	○
	setRollbackOnly	○
	setTransactionTimeout	○
	suspend	○
javax.transaction.Transaction	commit	○
	delistResource	−
	enlistResource	−
	getStatus	○
	registerSynchronization	○

インタフェース	メソッド	使用可否
javax.transaction.Transaction	rollback	○
	setRollbackOnly	○

（凡例）　○：使用できる　－：使用できない

注　使用できないメソッドを使用しようとすると，javax.transaction.SystemException がスローされます。

(2)　トランザクションマネージャを使用する場合の注意事項

- ルックアップで取得したトランザクションマネージャを使用する場合，トランザクション管理に含める処理（データベースのコネクションの取得，使用，解放など）は，トランザクションの開始後から決着前，または開始後から中断前および再開後から決着前の範囲内で実装してください。トランザクションの開始前や決着後，中断中に実装された処理はトランザクション管理に含まれません。
- トランザクションマネージャを使用してトランザクションを制御するコンポーネントでは，UserTransaction を使用しないでください。
- トランザクションマネージャ（javax.transaction.TransactionManager）の begin メソッドで開始したトランザクションは，begin メソッドの呼び出し前に setTransactionTimeout メソッドでタイムアウト時間を指定した場合，指定された時間でタイムアウトします。setTransactionTimeout メソッドを呼び出していない場合は，簡易構築定義ファイルの論理 J2EE サーバのプロパティ ejbserver.jta.TransactionManager.defaultTimeOut の指定値（デフォルト 180 秒）でタイムアウトします。

(3)　Synchronization を使用する場合の注意事項

トランザクション（javax.transaction.Transaction）の registerSynchronization メソッドで登録する Synchronization（javax.transaction.Synchronization）の beforeCompletion メソッド，および afterCompletion メソッドでは，J2EE サーバが提供するサービスを使用できません。使用できないサービスの例を次に示します。

- リソースアクセス
- UserTransaction または CMT によるトランザクション操作
- EJB アクセス
- JNDI アクセス

これらのサービスのうちリソースアクセスを行った場合，リソースアクセスでトランザクションマネージャが管理するトランザクションの一部として管理されないため，不整合が発生することがあります。リソースアクセスをする場合は，フレームワークの責任で，フレームワークがリソースに対して直接トランザクションを制御するようにしてください。

上記のような注意事項があるため，ユーザプログラムで Synchronization を利用することは推奨しません。ユーザプログラムからトランザクション決着のタイミングを利用したい場合には，EJB の javax.ejb.SessionSynchronization を使用してください。

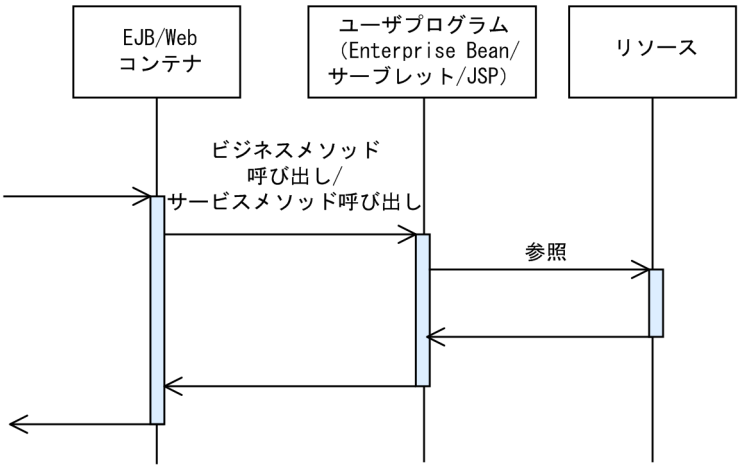
3.4.7　コンテナ管理のトランザクション（CMT）を使用する場合の処理概要と留意点

コンテナ管理のトランザクションを使用すれば，Enterprise Bean のビジネスメソッドが呼ばれる直前に自動的にトランザクションを開始し，ビジネスメソッドの処理が終了した直後に自動的にトランザクションを

コミットできます。ユーザプログラムのコーディングとしてトランザクション管理処理をまったく記述する必要がなく，容易にリソースへのアクセスのトランザクションを管理できます。

コンテナ管理のトランザクションを使用する場合のシーケンスを示します。

図 3-13　コンテナ管理のトランザクション使用時のシーケンス



コンテナ管理のトランザクションを使用する場合，次の点に留意して実装してください。

- コンテナ管理のトランザクションを使用する場合，Enterprise Bean のメソッドごとにトランザクション属性を指定できます。指定できるのは，Required，RequiresNew，Mandatory，Supports，NotSupported，Never のどれかです。トランザクションを使用する場合，DD の<transaction-type>タグに Container を指定し，メソッドごとのトランザクション属性は<trans-attribute>タグに指定します。また，DD を使用しないで，アノテーションで定義することもできます。トランザクション属性の詳細については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「2.7.3　CMT」を参照してください。アノテーションについては，マニュアル「アプリケーションサーバ リファレンス API 編」の「2.　アプリケーションサーバが対応しているアノテーションおよび Dependency Injection」を参照してください。
- トランザクションを開始したあとのビジネスメソッドの中でリソースにアクセスした場合には，そのリソースアクセスは自動的にトランザクション管理がされます。
- トランザクション開始後に複数のリソースへアクセスする場合は，グローバルトランザクションに対応したリソースアダプタを使用し，リソースアダプタのトランザクションサポートレベルを XATransaction に設定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使用する場合，ユーザプログラムのコーディングとしてトランザクション管理のための処理を記述する必要はありません。
- コンテナ管理のトランザクションは Enterprise Bean で使用できます。サーブレットおよび JSP では使用できません。

3.4.8　UserTransaction インタフェースを使用する場合の処理概要と留意点

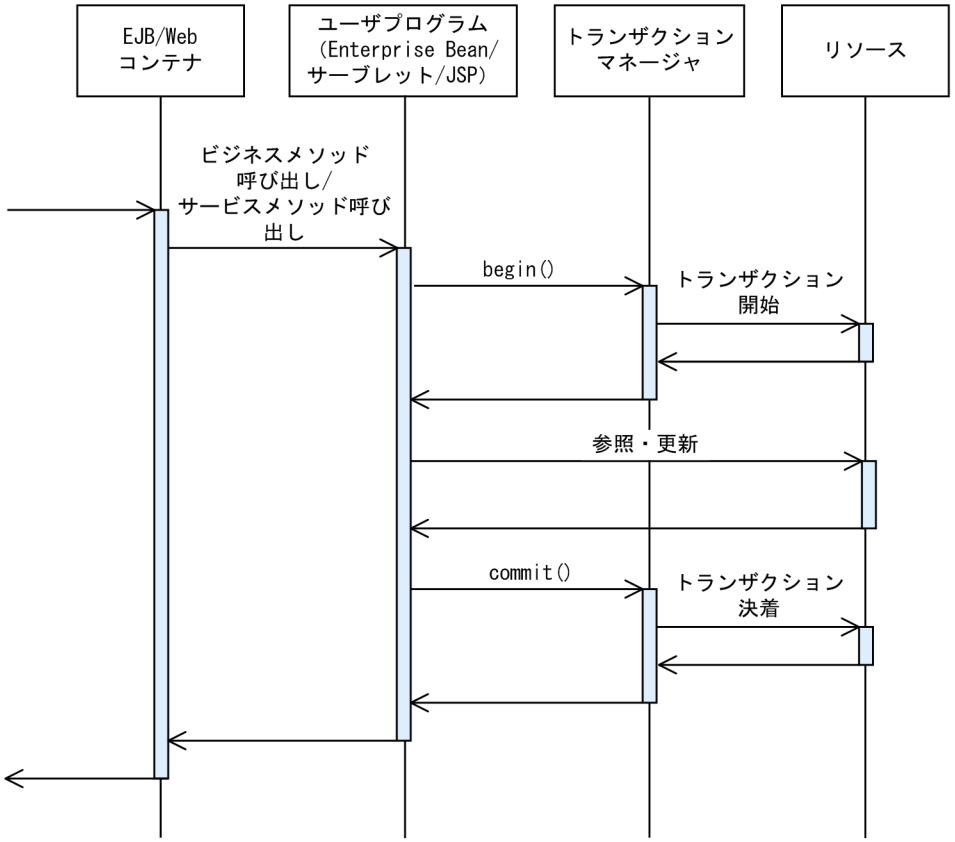
UserTransaction インタフェースを使用すれば，ユーザプログラムからトランザクションマネージャに対して，トランザクションの開始，決着の指示を出すことができます。ユーザプログラムで，トランザクションの細かい制御をしたい場合には，この方法を使用します。

ユーザプログラムからトランザクションマネージャに対して，トランザクションの開始，決着を指示する手順を示します。

1. UserTransaction オブジェクトを取得します。
   UserTransaction オブジェクトを取得するには，次の方法があります。
   - ネーミングサービスから JNDI を使用して"java:comp/UserTransaction"をルックアップする方法
   - EJBContext インタフェースの getUserTransaction メソッドを呼び出して取得する方法
   - DI を使用して取得する方法
2. UserTransaction オブジェクトの begin メソッドを呼び出して，トランザクションを開始します。
3. リソースにアクセスします。
4. UserTransaction オブジェクトの commit メソッドまたは rollback メソッドを呼び出して，トランザクションを決着します。

UserTransaction インタフェースを使用する場合のシーケンスを示します。

図 3-14　UserTransaction インタフェース使用時のシーケンス



UserTransaction インタフェースを使用する場合，次の点に留意して実装してください。

- UserTransaction インタフェースを使用する場合，DD の<transaction-type>タグに Bean を指定します。また，DD を使用しないで，アノテーションで定義することもできます。アノテーションについては，マニュアル「アプリケーションサーバ リファレンス API 編」の「2. アプリケーションサーバが対応しているアノテーションおよび Dependency Injection」を参照してください。

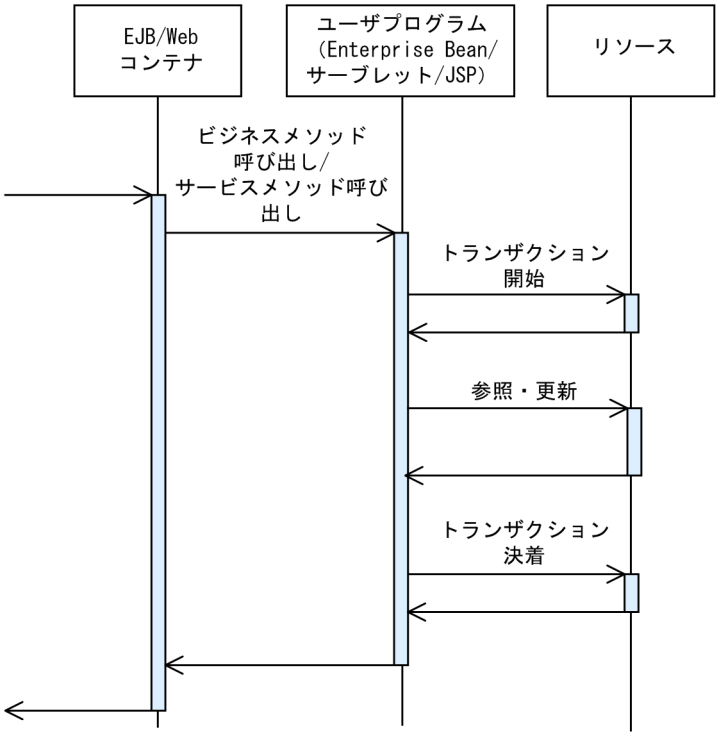
- トランザクションを開始後にリソースにアクセスした場合には，そのリソースアクセスは自動的にトランザクション管理されます。
- トランザクション開始後に複数のリソースへアクセスする場合は，グローバルトランザクションに対応したリソースアダプタを使用し，リソースアダプタのトランザクションサポートレベルを XATransaction に設定する必要があります。
- Enterprise Bean，サーブレット，および JSP で使用できます。
- UserTransaction インタフェースを使用して，ユーザプログラムで開始したトランザクションは，例外などが発生した場合にも，ユーザプログラムで commit または rollback を発行して決着させる必要があります。決着させなかった場合，リソースのロックが解放されない，または次のトランザクションが開始できないなどの問題が発生するおそれがあります。

3.4.9　リソース固有のトランザクション管理インタフェースを使用する場合の処理概要と留意点

リソース固有のインタフェースを使用して，ユーザプログラムが直接リソースのトランザクションを制御することもできます。例えば，DB Connector であれば，Connection インタフェースの setAutoCommit メソッド，commit メソッド，および rollback メソッドを使用して，ユーザプログラムが直接リソースのトランザクションを制御できます。

リソース固有のトランザクション管理インタフェースを使用する場合のシーケンスを示します。

図 3-15　リソース固有のトランザクション管理インタフェース使用時のシーケンス



リソース固有のトランザクション管理インタフェースを使用する場合，次の点に留意して実装してください。

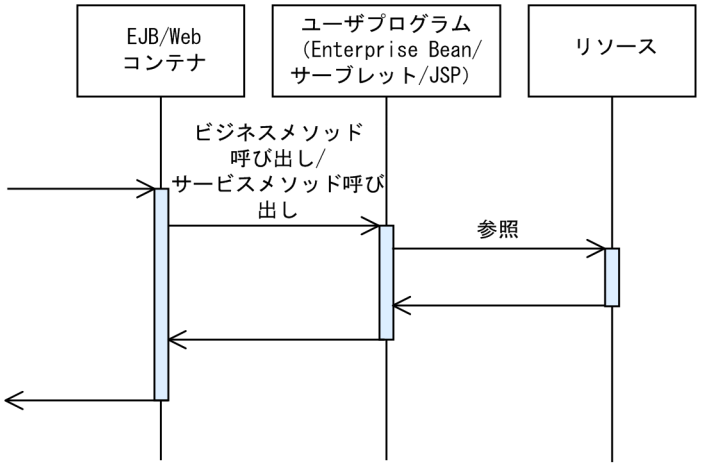
- トランザクションマネージャが提供するトランザクションタイムアウトなどの機能は使用できません。
- 複数のリソースへのアクセスをトランザクション管理することはできません。

3.4.10　トランザクションを使用しない場合の処理概要と留意点

リソースへのアクセスをトランザクション管理しないこともできます。リソースに対して参照しかしない場合などには，トランザクション管理のコストを削減するために，この方法を使用できます。

トランザクションを使用しない場合のシーケンスを示します。

図 3-16　トランザクション未使用時のシーケンス



トランザクションを使用しない場合，次の点に留意して実装してください。

- Enterprise Bean の DD の<transaction-type>タグに Container を指定して，<trans-attribute>タグに NotSupported，Never のどちらかを指定すれば，トランザクションは使用されません。また，DD を使用しないで，アノテーションで定義することもできます。トランザクション属性の詳細については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「2.7.3 CMT」を参照してください。アノテーションについては，マニュアル「アプリケーションサーバ リファレンス API 編」の「2.　アプリケーションサーバが対応しているアノテーションおよび Dependency Injection」を参照してください。
- Enterprise Bean の DD の<transaction-type>タグに Bean を指定して，UserTransaction オブジェクトの begin メソッドを呼び出さなければ，トランザクションは使用されません。また，DD を使用しないで，アノテーションで定義することもできます。アノテーションについては，マニュアル「アプリケーションサーバ リファレンス API 編」の「2.　アプリケーションサーバが対応しているアノテーションおよび Dependency Injection」を参照してください。
- サーブレットおよび JSP で UserTransaction オブジェクトの begin メソッドを呼び出さなければ，トランザクションは使用されません。
- 特定のリソースアダプタへのアクセスだけをトランザクションで管理しないこともできます。これを実現するには，トランザクションで管理しないリソースアダプタのトランザクションサポートレベルを NoTransaction に設定します。トランザクションサポートレベルを NoTransaction に設定したリソースアダプタでは，トランザクション開始後にリソースにアクセスしてもトランザクション管理の対象とはなりません。

3.4.11　JTA によるトランザクション実装時の注意事項

JTA を使用してトランザクションを実装したプログラムの処理の内容，および動作を次の表に示します。

表 3-17　JTA の動作

処理の内容	動作
• EJB 呼び出しで，java.rmi.RemoteException または java.rmi.RemoteException を継承した例外が発生した場合 • CORBA のシステム例外が発生した場合	クライアント側のトランザクションをロールバックにマークするかしないかは，プロパティ（ejbserver.distributedtx.rollbackClientTxOnSystemException）の設定で変わります。プロパティの設定値とその動作については，「3.4.5　システム例外発生時のトランザクションの動作」を参照してください。
トランザクションタイムアウト発生後に，javax.transaction.UserTransaction.commit メソッドを呼び出した場合	javax.transaction.RollbackException 例外が発生します。ただし，トランザクションタイムアウトが発生したあと，2 回以上 UserTransaction.commit メソッドを呼び出すと，2 回目以降は java.lang.IllegalStateException 例外が発生します。
トランザクションタイムアウト発生後に javax.transaction.UserTransaction.rollback メソッドを呼び出した場合	例外は発生しません。
トランザクションタイムアウト後の javax.transaction.UserTransaction.getStatus メソッドの戻り値	javax.transaction.Status.STATUS_ROLLEDBACK が返されます。ただし，UserTransaction.commit メソッド，または UserTransaction.rollback メソッドを呼び出したあとは，javax.transaction.Status.STATUS_NO_TRANSACTION が返されます。
javax.ejb.SessionSynchronization を実装した EJB の beforeCompletion メソッド，または afterCompletion メソッドからの EJB 呼び出し	例外が発生することなく，呼び出すことができます。

3.4.12　実行環境での設定

リソース接続とトランザクション管理機能を使用する場合，J2EE サーバ，およびリソースアダプタの設定が必要です。

(1)　J2EE サーバの設定

J2EE サーバの設定は，簡易構築定義ファイルで実施します。リソース接続とトランザクション管理の機能の定義は，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に指定します。簡易構築定義ファイルでの設定を次の表に示します。なお，トランザクション管理では，トランザクションのタイムアウトも指定できます。

簡易構築定義ファイルおよびパラメタについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.6　簡易構築定義ファイル」を参照してください。

トランザクションのタイムアウトについては，「3.15.8　トランザクションタイムアウトとステートメントキャンセル」を参照してください。

表 3-18　簡易構築定義ファイルでのリソース接続とトランザクション管理の機能を使用するための定義

項目	指定するパラメタ	設定内容
トランザクションの種類	ejbserver.distributedtx.XATransaction.enabled	ライトトランザクションを使用するか，グローバルトランザク

項目	指定するパラメタ	設定内容
トランザクションの種類	ejbserver.distributedtx.XATransaction.enabled	ションを使用するかどうかを指定します。デフォルトの設定では，ライトトランザクションが有効になっています。
システム例外発生時のクライアントトランザクションの動作	ejbserver.distributedtx.rollbackClientTxOnSystemException	システム例外が発生したときにクライアントトランザクションをロールバックにマークするかどうかを指定します。
ステータスファイルの格納ディレクトリ	• ejbserver.distributedtx.ots.status.directory1 • ejbserver.distributedtx.ots.status.directory2	インプロセストランザクションサービスのステータスファイルとステータスファイルのバックアップを格納するディレクトリとして指定します。
障害検知のタイムアウト	ejbserver.connectionpool.validation.timeout	コネクション障害検知機能のタイムアウト時間およびコネクション数調節機能によるコネクション削除処理のタイムアウト時間を指定します。

! 注意事項

インプロセストランザクションサービスのステータスファイルとステータスファイルのバックアップを格納するディレクトリの指定について

インプロセストランザクションサービスでは，トランザクションの整合性を保証するため，ホスト名または IP アドレスを J2EE サーバの識別情報としてステータスファイル内に取り込みます。このため，J2EE サーバのコンフィグレーション定義で vbroker.se.iiop_tp.host パラメタを設定していて値を変更する場合，または vbroker.se.iiop_tp.host パラメタを設定しないで J2EE サーバを起動するマシンの IP アドレスを変更する場合は，次の手順に従ってください。

1. J2EE サーバ上でトランザクションが存在しない状態で，J2EE サーバを停止してください。
2. IP アドレス，または vbroker.se.iiop_tp.host パラメタの設定を変更してください。
3. ejbserver.distributedtx.ots.status.directory1 パラメタで指定したディレクトリを削除してください。
4. J2EE サーバを起動してください。

(2)　リソースアダプタの設定

実行環境でのリソースアダプタの設定は，サーバ管理コマンドおよび属性ファイルを使用します。トランザクション管理のための機能の定義には，Connector 属性ファイルの<resourceadapter>タグに指定します。設定内容について，次の表に示します。

Connector 属性ファイルについては，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1　Connector 属性ファイル」を参照してください。

表 3-19　Connector 属性ファイルでのトランザクション管理のための機能の定義

項目	指定するタグ	設定内容
トランザクションサポートレベル	<outbound-resourceadapter>－<transaction-support>タグ	トランザクションサポートレベルを設定します。

項目	指定するタグ	設定内容
トランザクションサポートレベル	<outbound-resourceadapter>－<transaction-support>タグ	トランザクション管理なし(NoTransaction), ローカルトランザクション(LocalTransaction), またはグローバルトランザクション（XATransaction）を指定します。

3.5　リソースへのサインオン方式

リソースへのサインオンの方式として，次のどちらかの方式を選択できます。

- **コンテナ管理サインオン**

  アプリケーションサーバで自動的にサインオンする方式です。この方式を使用する場合，リソースアダプタごとにユーザ名とパスワードを設定しておけば，コネクション取得時に，アプリケーションサーバによって自動的にユーザ名とパスワードがリソースに伝達されます。

  コンテナ管理サインオンを使用するには，Enterprise Bean やサーブレットの DD の，<resource-ref>タグ中の<res-auth>タグに，Container を指定してください。

- **コンポーネント管理サインオン**

  ユーザプログラムでリソースにサインオンする方式です。この方式を使用する場合，コネクションを取得するときのユーザ名とパスワードをユーザプログラムで指定します。

  （例）DB Connector の場合

  DataSource クラスの getConnection を呼び出すときに，引数でユーザ名とパスワードを指定します。

  コンポーネント管理サインオンを使用するには，Enterprise Bean やサーブレットの DD の<resource-ref>タグ中の<res-auth>タグに，Application を指定してください。

コネクションプーリングを使用する場合は，コネクションを効率良く再利用できるコンテナ管理サインオンの使用をお勧めします。

なお，セットアップウィザードを使用して構築したシステムの場合，ejbserver.connectionpool.applicationAuthentication.disabled に true が指定されているため，アプリケーション認証が使えません。

3.6　データベースへの接続

この節では，DB Connector を使用してデータベースと接続する機能について説明します。

この節の構成を次の表に示します。

表 3-20　この節の構成（データベースへの接続）

分類	タイトル	参照先
解説	DB Connector による接続の概要	3.6.1
	使用できる J2EE コンポーネントおよび機能	3.6.2
	接続できるデータベース	3.6.3
	DB Connector（RAR ファイル）の種類	3.6.4
	HiRDB と接続する場合の前提条件と注意事項	3.6.5
	Oracle と接続する場合の前提条件と注意事項	3.6.6
	SQL Server と接続する場合の前提条件と注意事項	3.6.7
	XDM/RD E2 と接続する場合の前提条件と注意事項	3.6.8
設定	実行環境での設定（リソースアダプタでの設定）	3.6.9

注　「実装」および「運用」について，この機能固有の説明はありません。

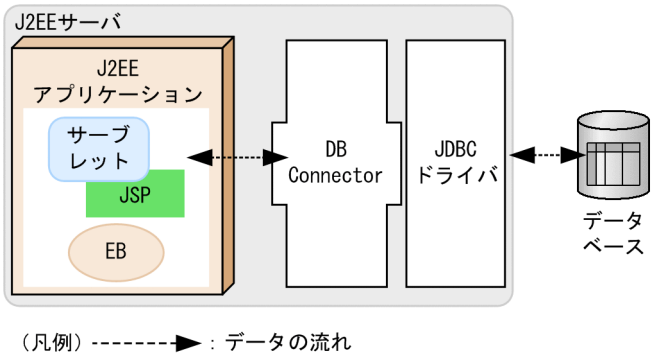
DB Connector を使用して接続できるデータベースには，HiRDB，Oracle，SQL Server，XDM/RD E2 があります。データベースと接続する場合，接続先のデータベースの種類によって，データベースの接続方法，使用できる JDBC ドライバなどが異なります。ここでは，データベース接続の前提条件と，使用できる機能について説明します。

3.6.1　DB Connector による接続の概要

データベースと接続する場合，リソースアダプタとして DB Connector を使用できます。DB Connector は，JDBC を利用したデータベースアクセスをするためのリソースアダプタです。DB Connector は，J2EE アプリケーションからコネクションファクトリ（javax.sql.DataSource インタフェース）をルックアップして使用します。

DB Connector による接続の概要を次の図に示します。

図 3-17　DB Connector による接続の概要



DB Connector でのデータベース接続では，JDBC ドライバに，HiRDB Type4 JDBC Driver，Oracle JDBC Thin Driver，または SQL Server の JDBC ドライバを使用します。

DB Connector の設定の詳細については，「3.6.9　実行環境での設定（リソースアダプタでの設定)」，およびマニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「4.2　データベースと接続するための設定」を参照してください。

また，データベース上のキューと接続する場合の接続方法については，「3.7　データベース上のキューとの接続」を参照してください。

! 注意事項

リソースアダプタを使用する場合，J2EE アプリケーションからリソースアダプタへのリファレンスを解決しておく必要があります。リソースアダプタを使用している J2EE アプリケーションをカスタマイズするときに，J2EE アプリケーションからリソースアダプタへのリファレンスを解決しておいてください。

3.6.2　使用できる J2EE コンポーネントおよび機能

ここでは，データベース接続で使用できる機能について説明します。

データベース接続で使用できる J2EE コンポーネントおよび機能を，データベースごとに次の表に示します。

表 3-21　使用できる J2EE コンポーネントおよび機能

項目		HiRDB	Oracle	SQL Server	XDM/RD E2
J2EE コンポーネント	Servlet/JSP	○	○	○	○
	Stateless Session Bean	○	○	○	○
	Stateful Session Bean	○	○	○	○
	Singleton Session Bean	○	○	○	○
	Entity Bean（BMP）	○	○	○	×
	Entity Bean（CMP1.1）	○	×	×	×
	Entity Bean（CMP2.0）	○※1	×	×	×
	Message-driven Bean（onMessage メソッドからのデータベースアクセス）	○	○	○	○
使用できる機能	コネクションプーリング	○	○	○	○
	コネクションプールのウォーミングアップ	○	○	○	○
	コネクションシェアリング・アソシエーション	○	○	○	○
	DataSource オブジェクトのキャッシング	○	△※5	○	○

項目		HiRDB	Oracle	SQL Server	XDM/RD E2
使用できる機能	DB Connector のコンテナ管理でのサインオンの最適化	○	○	○	○
	コネクション数調節機能	○	○	○	○
	コネクションスイーパ機能	○	○	○	○
	コネクションプールの情報表示（cjlistpool コマンド）	○	○	○	○
	コネクションプールのクリア（cjclearpool コマンド）	○	○	○	○
	リソースへの接続テスト	○	○	○	○
	コネクションの障害検知	○	○	○	○
	コネクション障害検知のタイムアウト	○	○	○	○
	コネクション枯渇時のコネクション取得待ち	○	○	○	○
	コネクション取得リトライ	○	○	○	○
	コネクション自動クローズ	○	○	○	○
	コネクションプールの一時停止	○	△※6	×	○
	コネクションプールの再開	○	△※6	×	○
	ステートメントプーリング	○※2	○	○	○※2
	ステートメントキャンセル	○	○※3	○	○
	ステートメント setQueryTimeout メソッド	○	○※3	○	×
	J2EE リソースのオプショナル名機能	○	○	○	○
	クライアント API で発生した例外の通知機能	○	○	○	○
	コネクション ID の PRF トレース出力	○	○	×	○
	障害調査用 SQL の出力	○	○	○	○

項目		HiRDB	Oracle	SQL Server	XDM/RD E2
使用できる機能	クラスタコネクションプール	×	○※4	×	×

（凡例）○：使用できる　　×：使用できない　　△：一部制限あり

注※1　CMP2.0 の機能である，CMR 機能は利用できません。

注※2　HiRDB 自動再接続機能とステートメントプーリング機能の併用はできません。

注※3　Oracle に接続する場合の注意事項があります。注意事項については，「3.6.6　Oracle と接続する場合の前提条件と注意事項」を参照してください。

注※4　RAC 機能を使用している場合で，Oracle JDBC Thin Driver を使用して接続するときに使用できます。

注※5　メンバリソースアダプタを使用して Oracle に接続した場合は使用できません。

注※6　メンバリソースアダプタを使用して Oracle に接続した場合に使用できます。

3.6.3　接続できるデータベース

ここでは，DB Connector を使用して接続できるデータベースについて説明します。

(1)　接続できるデータベースの種類

DB Connector を利用して接続できるデータベースには，HiRDB，Oracle，SQL Server，XDM/RD E2 があります。なお，SQL Server および XDM/RD E2 ではグローバルトランザクションは使用できません。

(2)　データベースと JDBC ドライバの対応

DB Connector を使用してデータベースに接続するためには，データベースに対応する JDBC ドライバが必要です。接続先のデータベースと使用できる JDBC ドライバの対応について次の表に示します。

表 3-22　接続先のデータベースと使用できる JDBC ドライバ（データベース接続の場合）

データベース	JDBC ドライバ		
	HiRDB Type4 JDBC Driver	Oracle JDBC Thin Driver	SQL Server の JDBC ドライバ
HiRDB	◎	×	×
Oracle	×	◎	×
SQL Server	×	×	○
XDM/RD E2	○	×	×

（凡例）

◎：使用できる。かつ使用を推奨する。

○：使用できる。

×：使用できない。

(3)　DB Connector がサポートする JDBC 仕様

接続に使用する JDBC ドライバと DB Connector がサポートする JDBC 仕様を次の表に示します。ただし，接続に使用する JDBC ドライバが，JDBC 仕様で規定された機能をサポートしていない場合は，DB Connector でその機能を使用できません。

表 3-23　接続に使用する JDBC ドライバと DB Connector がサポートする JDBC 仕様

接続に使用する JDBC ドライバ	DB Connector がサポートする JDBC 仕様
HiRDB Type4 JDBC Driver	JDBC 4.0
Oracle JDBC Thin Driver	JDBC 4.0
SQL Server JDBC Driver 3.0	JDBC 3.0
JDBC Driver 4.0 for SQL Server	JDBC 4.0

ポイント

Oracle JDBC Thin Driver は，JDBC 4.0 の仕様で規定された範囲の機能をサポートしています。Oracle パッケージのクラスおよびインタフェースを使用した Oracle 拡張機能は使用できません。

また，アプリケーションサーバで Oracle JDBC Thin Driver を使用してデータベースと接続する場合，Oracle JDBC Thin Driver の拡張機能である，文キャッシュおよび接続キャッシュ機能を利用することはできません。アプリケーションサーバで提供するステートメントプーリング，またはコネクションプーリングを使用してください。ステートメントプーリングまたはコネクションプーリングについては，「3.14　パフォーマンスチューニングのための機能」を参照してください。

なお，java.sql.Wrapper インタフェースは未サポートです。Wrapper インタフェースのメソッドを使用した場合，UnsupportedOperationException 例外がスローされます。

! 注意事項

JDBC 仕様の java.sql.Statement インタフェースのメソッドのうち，次に示すものは未サポートです。

- setCursorName(String name)
- setFetchDirection(int direction)

DB Connector を使用してデータベースに接続する場合，JDBC 4.1 および JDBC 4.2 仕様で追加された機能およびインタフェース・メソッドはサポートしません。

try-with-resources 文を使用した JDBC リソースの自動解放もサポートしません。従来どおり finally ブロックで明示的に JDBC リソースを close してください。

3.6.4　DB Connector（RAR ファイル）の種類

DB Connector を使用してデータベースに接続する場合，使用する JDBC ドライバに対応した RAR ファイルを使用します。RAR ファイルは，サーバ管理コマンドを使用して操作します。サーバ管理コマンドを使用して RAR ファイルを操作する方法については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」を参照してください。

JDBC ドライバの種類と対応する RAR ファイルについて次の表に示します。

表 3-24　JDBC ドライバと RAR ファイルの対応

JDBC ドライバ	RAR ファイル	説明
HiRDB Type4 JDBC Driver	DBConnector_HiRDB_Type4_CP.rar	トランザクション管理をしない場合，またはローカルトランザクションを使用する場合に使用します。
	DBConnector_HiRDB_Type4_XA.rar※	グローバルトランザクションを使用する場合に使用します。
Oracle JDBC Thin Driver	DBConnector_Oracle_CP.rar	トランザクション管理をしない場合，またはローカルトランザクションを使用する場合に使用します。

JDBC ドライバ	RAR ファイル	説明
Oracle JDBC Thin Driver	DBConnector_Oracle_XA.rar※	グローバルトランザクションを使用する場合に使用します。
	DBConnector_CP_ClusterPool_Root.rar	クラスタコネクションプール機能でルートリソースアダプタを使用する場合に使用します。
	DBConnector_Oracle_CP_ClusterPool_Member.rar	クラスタコネクションプール機能でメンバリソースアダプタを使用する場合に使用します。
SQL Server の JDBC ドライバ	DBConnector_SQLServer_CP.rar	SQL Server への接続に使用する RAR ファイルです。トランザクション管理をしない場合，またはローカルトランザクションを使用する場合に使用します。

注※　ライトトランザクション機能は有効にできません。

3.6.5　HiRDB と接続する場合の前提条件と注意事項

ここでは，HiRDB と接続する場合の前提条件と注意事項について説明します。HiRDB のバージョンによって，使用できる JDBC ドライバが異なります。

(1)　前提条件

- **使用できる JDBC ドライバ**
  使用できる JDBC ドライバは HiRDB Type4 JDBC Driver となります。
- **接続方法**
  次のどちらかの RAR ファイルを使用します。
  - DBConnector_HiRDB_Type4_CP.rar
  - DBConnector_HiRDB_Type4_XA.rar

  RAR ファイルの種類に応じて，指定できるトランザクションサポートレベルが異なります。また，ライトトランザクションを使用できるかどうかも異なります。
  RAR ファイルごとに指定できるトランザクションサポートレベルを次の表に示します。

表 3-25　RAR ファイルごとに使用できるトランザクションサポートレベル

使用する DB Connector (RAR ファイル)	トランザクションサポートレベル	ライトトランザクション	
		有効	無効
DBConnector_HiRDB_Type4_CP.rar	NoTransaction LocalTransaction	○	○
DBConnector_HiRDB_Type4_XA.rar	XATransaction	－	○

（凡例）　○：使用できる　　－：該当しない

(2)　HiRDB と接続するときの注意事項

HiRDB と接続する場合の注意を次に示します。

- HiRDB の自動再接続機能を使用すると，データベースネットワーク障害などで HiRDB サーバとの接続が切断された場合に，コネクションが自動的に再接続されます。ただし，トランザクション中に接続が切断されると，SQLException 例外が発生します。J2EE アプリケーションで SQLException 例外を受

け取ったときには，処理を続行しないでください。続けて DBMS にアクセスするとデータの不整合などの問題が発生するおそれがあります。

- コネクションプール機能とステートメントプーリング機能を有効にして，Statement.cancel()を使用すると，SQLException 例外が発生することがあります。この場合，コネクションプール機能，またはステートメントプーリング機能のどちらかを無効にすることをお勧めします。
- J2EE アプリケーションが次の条件をすべて満たす場合は，コネクションが二つ使用され，HiRDB への接続が同時に利用しているユーザ数の 2 倍になります。HiRDB のシステム共通定義の pd_max_users オペランドに，同時に利用しているユーザ数の 2 倍の値を指定してください。pd_max_users オペランドについては，マニュアル「HiRDB システム定義」を参照してください。
  1. トランザクションサポートレベルが XATrasaction の DB Connector を使用する。
  2. アプリケーションサーバが管理するトランザクション内でコネクション※を使ってデータベースにアクセスする。
  3. 2.のトランザクションが決着する前に，トランザクション外でコネクション※を使ってデータベースにアクセスする。

  注※　このコネクションは 1.の DB Connector から取得したコネクションで，かつ同一コネクションです。
- J2EE アプリケーションが次の条件をすべて満たす場合は，エラーが発生します。
  1. トランザクションサポートレベルが XATrasaction の DB Connector を使用する。
  2. アプリケーションサーバが管理するトランザクション内でコネクション※を使ってデータベースにアクセスする。
  3. 2.のトランザクションが決着する前に，トランザクション外でコネクション※を使ってデータベースにアクセスする。

  注※　このコネクションは 1.の DB Connector から取得したコネクションで，かつ同一コネクションです。
- 次の条件をすべて満たす場合，同一コネクションを複数の異なるグローバルトランザクションに同時に参加させることはできません。トランザクションごとに使用するコネクションを分けてください。
  1. トランザクションサポートレベルが XATrasaction の DB Connector を使用する。
  2. アプリケーションサーバが管理するトランザクション内でコネクション※を使ってデータベースにアクセスする。
  3. 2.のトランザクションが決着する前に，トランザクション外でコネクション※を使ってデータベースにアクセスする。

  注※　このコネクションは 1.の DB Connector から取得したコネクションで，かつ同一コネクションです。
- HiRDB のコネクション自動再接続機能およびステートメントプール機能を有効にすると，自動再接続機能によってコネクションが再接続されたあとの SQL 実行でメッセージ「KFPA11901-E」を含んだ SQLException 例外が発生することがあります。トランザクションサポートレベルが LocalTransaction，または NoTransaction の DB Connector でステートメントプール機能を有効にして HiRDB に接続する場合は，HiRDB のコネクション自動再接続機能は使用しないでください。障害が発生した場合は，コネクションの障害検知機能，または cjclearpool コマンドを実行してください。

3.6.6　Oracle と接続する場合の前提条件と注意事項

ここでは，Oracle と接続する場合の前提条件と注意事項について説明します。

(1)　前提条件

- **使用できる JDBC ドライバ**
  使用できる JDBC ドライバは Oracle JDBC Thin Driver となります。
- **接続方法**
  次のどれかの RAR ファイルを使用します。
  - DBConnector_Oracle_CP.rar
  - DBConnector_Oracle_XA.rar
  - DBConnector_CP_ClusterPool_Root.rar
  - DBConnector_Oracle_CP_ClusterPool_Member.rar

  RAR ファイルの種類に応じて，指定できるトランザクションサポートレベルが異なります。また，ライトトランザクションを使用できるかどうかも異なります。
  RAR ファイルごとに指定できるトランザクションサポートレベルを次の表に示します。

表 3-26　RAR ファイルごとに使用できるトランザクションサポートレベル

使用する DB Connector (RAR ファイル)	トランザクションサポートレベル	ライトトランザクション	
		有効	無効
DBConnector_Oracle_CP.rar	NoTransaction LocalTransaction	○	○
DBConnector_Oracle_XA.rar	XATransaction	－	○
DBConnector_CP_ClusterPool_Root.rar DBConnector_Oracle_CP_ClusterPool_Member.rar	NoTransaction LocalTransaction	○	○

（凡例）　○：使用できる　　－：該当しない

(2)　Oracle と接続する場合の注意事項

ここでは，Oracle に接続する場合の使用時の注意について説明します。

(a)　使用できる J2EE コンポーネントの違い

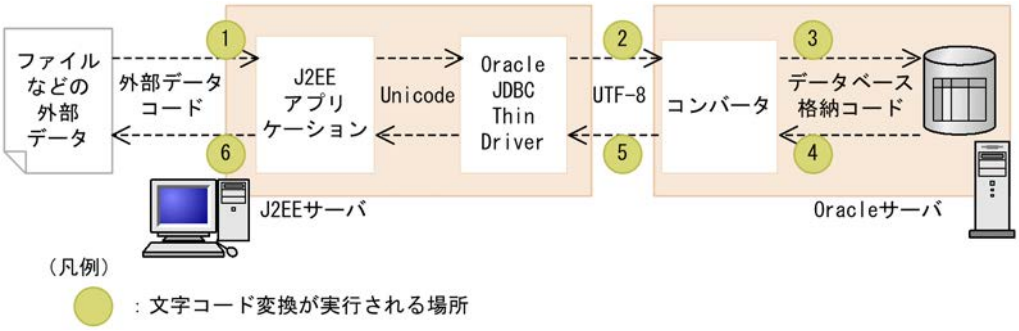
Oracle JDBC Thin Driver で Oracle に接続する場合に使用できる J2EE コンポーネントを次に示します。

- サーブレット/JSP
- Stateless Session Bean
- Stateful Session Bean
- Entity Bean（BMP）
- Message-driven Bean

(b)　日本語文字コード変換の相違点と Oracle JDBC Thin Driver での文字化けを回避するための設定

JDBC ドライバでは，データをデータベースに格納するとき，およびデータベースからデータを取り出すときに，Unicode とデータベース格納コードとの間で適宜，文字コードを変換します。ここでは，Oracle JDBC Thin Driver を使用する場合に文字コード変換が実施される場所を次の図に示します。

図 3-18　Oracle JDBC Thin Driver を使用する場合に文字コード変換が実施される場所



図中の場所ごとに実行される文字コード変換について，次の表に示します。

表 3-27　Oracle JDBC Thin Driver を使用する場合に文字コード変換が実施される場所

場所	実施される内容
1	ネットワーク，ファイルなどの外部データを J2EE サーバで読み込む時に，外部データコードから Unicode に変換されます。
2	J2EE サーバで読み込んだデータを Oracle サーバに格納する時に，Unicode から UTF-8 に変換されます。J2EE サーバ上の JDBC ドライバ部分で実施されるコード変換です。
3	Oracle サーバのコンバータによって，UTF-8 からデータベース格納コードへの変換が実施されます。Oracle サーバ上で実施されるコード変換です。
4	Oracle サーバのコンバータによって，データベース格納コードから UTF-8 への変換が実施されます。
5	Oracle サーバから J2EE サーバが格納したデータを取得する時に，UTF-8 から Unicode に変換されます。
6	J2EE サーバで取得したデータをネットワークやファイルなどに書き出す時に，Unicode から外部データコードに変換されます。

Oracle JDBC Thin Driver を使用する場合，JavaVM と Oracle サーバのコンバータがサポートしているマッピング規則の相違から，文字化けが発生することがあります。文字化けは，外部データの文字コードとデータベース格納コードの組み合わせや，文字コード変換を行うコンバータの組み合わせによって発生します。

文字化けを回避するためには，外部データの文字コードと，データベース格納コードを，次に示す組み合わせで使用してください。

- 外部データの文字コードがシフト JIS（CP932）の場合

  データベース格納コードに「AL32UTF8」または「JA16SJISTILDE」を指定してください。なお，データベース格納コードに「JA16SJIS」を指定した場合，「～」の文字で文字化けが発生しますのでご注意ください。

- 外部データの文字コードが SJIS の場合

  データベース格納コードに「AL32UTF8」を指定してください。なお，データベース格納コードに「JA16SJIS」を指定した場合，「¢」「£」「¬」「‖」「−」で文字化けが発生しますのでご注意ください。

- 外部データの文字コードが EUC の場合

  データベース格納コードに「AL32UTF8」を指定してください。なお，データベース格納コードに「JA16EUC」を指定した場合，「¢」「£」「¬」で文字化けが発生しますのでご注意ください。

(c)　専用サーバ接続

専用サーバ接続の場合，次の機能は使用できません。次の機能を使用する場合は共有サーバ接続にしてください。詳細については，オラクルのサポートサービスにお問い合わせください。

- ステートメントキャンセル
- クエリータイムアウト

(d)　JTA トランザクション決着処理中の select 文の戻り値

XA トランザクションで Oracle を利用した場合に，JTA トランザクションの決着処理中に，トランザクション内で更新処理を実施した複数のデータベースに対し，Oracle の select 文を発行すると，各 select 文の戻り値が異なることがあります。これは，Oracle の分散読取り一貫性の制限に起因しています。

(3)　Oracle RAC を使用した Oracle への接続方法

Oracle RAC を使用した Oracle への接続方法は，負荷分散に使用する機能によって異なります。負荷分散に使用する機能および使用する RAR ファイルの対応を次の表に示します。

表 3-28　使用する機能および使用する RAR ファイルの対応

負荷分散に使用する機能	DB Connector の RAR ファイル名
アプリケーションサーバの機能（クラスタコネクションプール機能）	DBConnector_CP_ClusterPool_Root.rar DBConnector_Oracle_CP_ClusterPool_Member.rar
Oracle の機能	DBConnector_Oracle_CP.rar DBConnector_Oracle_CP_Cosminexus_RM.rar

3.6.7　SQL Server と接続する場合の前提条件と注意事項

ここでは，SQL Server と接続する場合の前提条件と注意事項について説明します。

(1)　SQL Server の場合の前提条件

SQL Server の場合の前提条件について示します。

- **使用できる JDBC ドライバ**
  使用できる JDBC ドライバは SQL Server の JDBC ドライバとなります。
- **接続方法**
  DBConnector_SQLServer_CP.rar を使用します。使用する RAR ファイルは，接続する SQL Server によって異なります。
  RAR ファイルごとに指定できるトランザクションサポートレベルを次の表に示します。

表 3-29　RAR ファイルごとに使用できるトランザクションサポートレベル（SQL Server）

使用する DB Connector（RAR ファイル）	トランザクションサポートレベル	ライトトランザクション	
		有効	無効
DBConnector_SQLServer_CP.rar	NoTransaction LocalTransaction	○	○

（凡例）　○：使用できる

なお，SQL Server と接続できるのは Windows の場合だけです。

(2)　SQL Server と接続する場合の注意事項

ここでは，SQL Server と接続する場合のシステムでの文字コード変換に関する注意事項と DB Connector 設定時の注意事項について説明します。

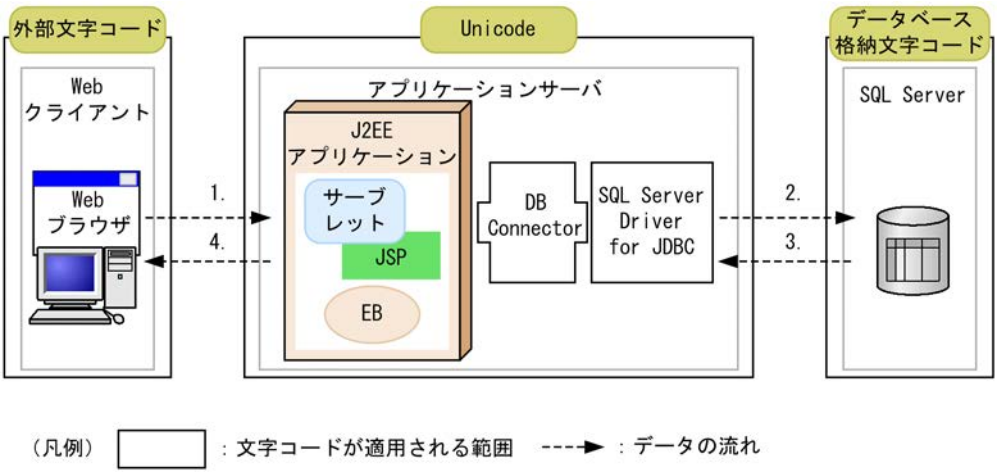
(a)　システムでの文字コード変換時の注意

SQL Server 接続時，データベースに日本語文字コードを含むデータを格納する場合，システムでの文字コード変換について考慮する必要があります。ここでは，システムでの文字コード変換の概要と，文字化けを回避するために設定時に注意することについて説明します。

Java では日本語文字コードは Unicode で表現されます。SQL Server を使用する場合，Web クライアントとアプリケーションサーバ間，およびアプリケーションサーバと SQL Server 間の処理で文字コード変換が実施されます。

SQL Server を使用する場合の文字コード変換の概要を次の図に示します。

図 3-19　SQL Server を使用する場合の文字コード変換の概要



図中の 1.～4.について説明します。

1. アプリケーションサーバが Web クライアントからデータを受信する時に，外部文字コードから Unicode に変換されます。
2. アプリケーションサーバがデータを SQL Server に格納する時に，Unicode からデータベース格納文字コードに変換されます。
3. SQL Server に格納されたデータをアプリケーションサーバが取得する時に，データベース格納文字コードから Unicode に変換されます。
4. アプリケーションサーバがデータを Web クライアントに送信する時に，Unicode から外部文字コードに変換されます。

SQL Server を使用する場合，外部文字コードとデータベース格納文字コードの組み合わせや，文字コード変換を実施する場合に使用されるコンバータの種類によっては，文字化けなどの問題が発生することがあります。このような文字化けを回避するためには，文字コードの設定に注意する必要があります。

SQL Server では次の表に示す文字データ型をサポートしています。SQL Server を使用する場合，Unicode データ型を使用することによって，文字コード変換時に発生する文字化けを防ぐことができます。

表 3-30　SQL Server でサポートする文字データ型

カテゴリ	文字データ型
Unicode データ型	nchar，nvarchar，ntext
非 Unicode 文字データ型	char，varchar，text

次に，SQL Server でのデータベース格納文字コードとして，Unicode データ型を使用する場合と，非 Unicode データ型を使用する場合の文字コード変換について説明します。

Unicode データ型を使用する場合

DB Connector のプロパティの sendStringParametersAsUnicode キーに true（デフォルト値）を設定している場合，文字化けは発生しません。false を設定した場合，外部文字コードに Shift_JIS，EUC-JP，ISO-2022-JP，または UTF-8 を使用すると文字化けが発生することがあります。

非 Unicode 文字データ型を使用する場合

外部文字コードの設定によって，文字化けが発生することがあります。

- 外部文字コードに Windows-31J を使用する場合
  文字化けは発生しません。
- 外部文字コードに Shift_JIS，EUC-JP，または ISO-2022-JP を使用する場合
  次に示す文字などで文字化けが発生します。

  「¢」「£」「¬」「‖」「−」「～」

- 外部文字コードに UTF-8 を使用する場合
  次に示す文字などで文字化けが発生します。

  「¢」「£」「¬」「¥」「¦」「¯」「– 」「～」

(b)　DB Connector の selectMethod プロパティ設定時の注意

DB Connector の selectMethod プロパティ（<config-property-name>の項目名）の値に「direct」を設定した場合の注意事項を次の表に示します。

表 3-31　DB Connector の selectMethod プロパティ設定時の注意事項

条件	注意事項
コネクションの障害検知機能が有効	コネクションに障害が発生していても正常であると誤診することがあります。その結果，ユーザアプリケーションプログラムに障害が発生したコネクションを返すことがあるので，コネクションの障害検知機能を使用しないでください。障害が発生したら，cjclearpool コマンドを実行してください。
Statement，PreparedStatement，CallableStatement を複数同時に生成	SQL Server の JDBC ドライバによって，同時に生成したステートメントごとに SQL Server への接続が生成されます。 また，ステートメントプーリング機能を使用する場合には，プールされているステートメントごとに接続が生成され，メモリを多く消費するので注意してください。

(3)　SQL Server 2005 に接続時の注意事項

SQL Server 2005 に接続する場合，コネクションプーリング機能を必ず使用してください。

また，SQL Server JDBC Driver 3.0 の挙動によって，次の現象が発生することがあります。

(a)　データベースとのセッションが切断されないで，使用されないセッションが残ってしまう。

GC（ガーベージコレクション）が発生するまで，データベースとのセッションが切断されないで，使用されないセッションが残ってしまうことがあります。この現象を回避するため，javagc コマンドを実行してください。

なお，この現象は次の条件で発生します。

次の操作を行ったとき。

- DB Connector の接続テスト
- DB Connector の停止
- cjclearpool コマンドの実行

次の機能が働いたとき。

- メソッドキャンセル機能
  メッセージ KDJE31016-W が出力されます。
- トランザクションタイムアウト機能
  メッセージ KDJE31002-W が出力されます。
- コネクション数調節機能
  メッセージ KDJE49532-I が出力されます。
- コネクションスイーパ機能
  DB Connector が出力するログ・トレースのレベルを WARNING または INFORMATION に設定することによって，コネクションスイーパ機能が働いたときを含め，マネージドコネクションが破棄されたときメッセージ KDJE50010-I が出力されます。

(b)　コネクションの取得に失敗する。

上記（a）のデータベースとのセッションが切断されないことに起因して，データベースとのセッションの数がデータベースの同時ユーザ接続の最大数に達した場合には，新規セッションが生成できなくなります。ユーザプログラムから javax.sql.DataSource インタフェースの getConnection メソッドを実行したときに，java.sql.SQLException が発生することがあります。この現象を回避するため，データベースの同時ユーザ接続の最大数を 0（無制限）に設定してください。

なお，この現象はデータベースの同時ユーザ接続の最大数を 1 以上の値に設定した場合に発生します。

(4)　SQL Server 2008 または SQL Server 2012 に接続時の注意事項

SQL Server 2008 または SQL Server 2012 に接続する場合の注意事項を次に示します。なお，SQL Server 2005 使用時の注意事項はどれも該当しません。

- SQL Server 2008 または SQL Server 2012 の認証モードは，混合モード（Windows 認証と SQL Server 認証）に設定してください。認証モードの詳細については，SQL Server 2008 または SQL Server 2012 のマニュアルを参照してください。
- SQL Server JDBC Driver 3.0 の JDBC ドライバの JAR ファイルと，JDBC Driver 4.0 for SQL Server の JDBC ドライバの JAR ファイルの両方を J2EE サーバのユーザクラスパスに指定することはできません。
- 次の設定は，DBConnector_SQLServer_CP.rar では設定できません。

SQL Server JDBC Driver 2.0 で新規に追加された設定
responseBuffering
encrypt
hostNameCertificate
trustServerCertificate
trustStore
trustStorePassword

SQL Server JDBC Driver 3.0 で新規に追加された設定
sendTimeAsDatetime

JDBC Driver 4.0 for SQL Server で新規に追加された設定
authenticationScheme

このため，responseBuffering のデフォルト値が adaptive として動作するため，アダプティブバッファリング機能は常に有効として動作します。詳細については，SQL Server 2008 または SQL Server 2012 のマニュアルを参照してください。

3.6.8　XDM/RD E2 と接続する場合の前提条件と注意事項

ここでは，XDM/RD E2 と接続する場合の前提条件と注意事項について説明します。

- **使用できる JDBC ドライバ**
  使用できる JDBC ドライバは HiRDB Type4 JDBC Driver となります。
- **接続方法**
  DBConnector_HiRDB_Type4_CP.rar を使用します。
  RAR ファイルごとに指定できるトランザクションサポートレベルを次の表に示します。

表 3-32　RAR ファイルごとに使用できるトランザクションサポートレベル（XDM/RD E2）

使用する DB Connector（RAR ファイル）	トランザクションサポートレベル	ライトトランザクション	
		有効	無効
DBConnector_HiRDB_Type4_CP.rar	NoTransaction LocalTransaction	○	○

（凡例）　○：使用できる

3.6.9　実行環境での設定（リソースアダプタでの設定）

データベースに接続するための機能を使用する場合，属性ファイルの設定が必要です。

実行環境でのリソースアダプタの設定は，サーバ管理コマンドおよび属性ファイルを使用します。データベースに接続するための機能の定義には，Connector 属性ファイルを使用します。

ここでは，使用する機能に関係なく，データベースに接続するために共通の DB Connector の設定について説明します。

- **データベースコネクション確立までの待ち時間**
  <config-property>タグの loginTimeout で，データベースコネクション確立までの J2EE アプリケーションの待ち時間を指定します。

なお，データベースに接続するための設定については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「4.2　データベースと接続するための設定」を参照してください。

3.7　データベース上のキューとの接続

この節では，DB Connector for Reliable Messaging と Reliable Messaging を使用してデータベース上のキューと接続する機能について説明します。

この節の構成を次の表に示します。

表 3-33　この節の構成（データベース上のキューとの接続）

分類	タイトル	参照先
解説	DB Connector for Reliable Messaging と Reliable Messaging による接続の概要	3.7.1
	DB Connector for Reliable Messaging と Reliable Messaging による接続の特徴	3.7.2
	使用できる機能	3.7.3
	接続できるデータベース	3.7.4
	DB Connector for Reliable Messaging（RAR ファイル）の種類	3.7.5
	HiRDB のキューに接続する場合の前提条件	3.7.6
	Oracle のキューに接続する場合の前提条件	3.7.7
設定	データベース上のキューに接続するための設定	3.7.8

注　「実装」および「運用」について，この機能固有の説明はありません。

データベース上のキューと接続するためには，リソースアダプタとして DB Connector for Reliable Messaging と Reliable Messaging を使用します。

Reliable Messaging の詳細については，マニュアル「Reliable Messaging」を参照してください。また，接続できるデータベースについては，「3.6　データベースへの接続」を参照してください。

3.7.1　DB Connector for Reliable Messaging と Reliable Messaging による接続の概要

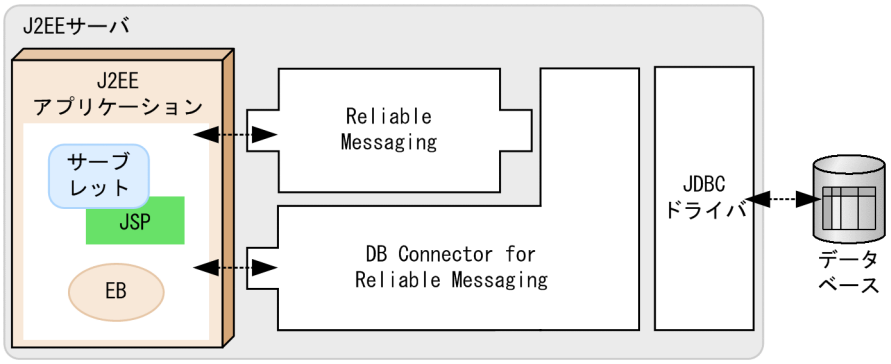
DB Connector for Reliable Messaging は，Reliable Messaging と連携して，JMS インタフェースを使用してデータベースと接続するためのリソースアダプタです。

Reliable Messaging と連携することによって，サーブレット，JSP，Enterprise Bean（Session Bean，Entity Bean，Message-driven Bean）から，JMS インタフェースを使用してデータベース上のキューにアクセスできます。また，JDBC インタフェースを使用して，データベース上のテーブルにもアクセスできます。

JMS インタフェースと JDBC インタフェースを使用して同じデータベースにアクセスするとき，グローバルトランザクションを 1 フェーズコミットで処理できます。これによって，処理性能が向上します。また，JMS インタフェースと JDBC インタフェースによるデータベース接続に使用する物理コネクションを共通できるため，リソースを有効活用できるようになります。

DB Connector for Reliable Messaging と Reliable Messaging による接続の概要を次の図に示します。

図 3-20　DB Connector for Reliable Messaging と Reliable Messaging による接続の概要



（凡例）--------▶：データの流れ

DB Connector for Reliable Messaging と Reliable Messaging によるデータベース接続では，JDBC ドライバに，HiRDB Type4 JDBC Driver または Oracle JDBC Thin Driver を使用します。

DB Connector for Reliable Messaging の設定の詳細については，マニュアル「Reliable Messaging」の「2.7　DB Connector for Reliable Messaging の機能」を参照してください。

3.7.2　DB Connector for Reliable Messaging と Reliable Messaging による接続の特徴

ここでは，DB Connector for Reliable Messaging と Reliable Messaging による接続の特徴について説明します。

(1)　DB Connector for Reliable Messaging および Reliable Messaging による処理の概要

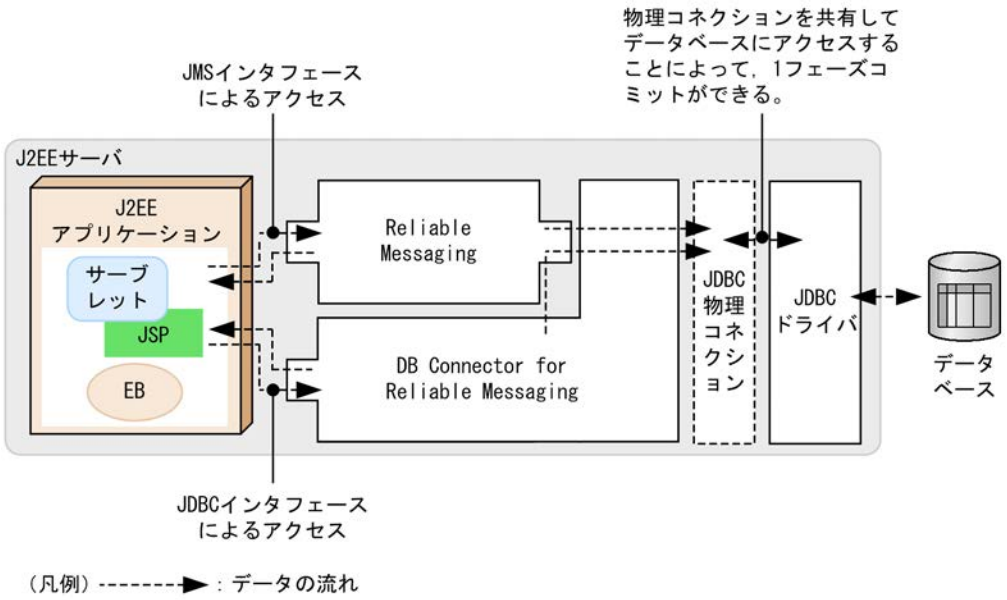
DB Connector for Reliable Messaging を使用して Reliable Messaging と連携することによって，データベース上のキューに対する JMS インタフェースによるメッセージの送受信や，メッセージによって起動される Message-driven Bean の呼び出しなどができます。

DB Connector for Reliable Messaging と Reliable Messaging によってデータベースに接続する場合，JMS インタフェースと JDBC インタフェースによるアクセスで使用する物理コネクションを共有することによって次のことが実現できるため，処理性能が向上します。

- ローカルトランザクションの適用
- グローバルトランザクションの 1 フェーズコミット

サーブレット，JSP，または，Enterprise Bean から，JMS インタフェースと JDBC インタフェースを使用して，データベースにアクセスするときの DB Connector for Reliable Messaging および Reliable Messaging の処理の概要を次の図に示します。

図 3-21　DB Connector for Reliable Messaging および Reliable Messaging の処理の概要



JMS インタフェースによるアクセスは Reliable Messaging を経由して，JDBC インタフェースによるアクセスは DB Connector for Reliable Messaging を経由して，データベースにアクセスします。なお，Reliable Messaging では，キューをデータベース上に構築するため，内部的に JDBC インタフェースによるデータベースへのアクセスを使用します。そのため，JDBC ドライバを経由してデータベースにアクセスします。

また，DB Connector for Reliable Messaging では，Reliable Messaging を経由して，Reliable Messaging と物理コネクションを共有してデータベースにアクセスします。これらのリソースアダプタでは，Reliable Messaging 用のコネクションプールを利用してコネクションを取得します。これによって，物理コネクションの共有を実現しています。

(2)　DB Connector for Reliable Messaging と Reliable Messaging によるデータベース接続の構成

DB Connector for Reliable Messaging と Reliable Messaging を使用して接続する場合のリソースの構成パターンについて説明します。ここでは，次に示す場合のリソースの構成について説明します。

- JMS インタフェースだけを使用する場合
- JMS インタフェースと JDBC インタフェースで同一データベースにアクセスする場合
- JMS インタフェースと JDBC インタフェースで異なるデータベースにアクセスする場合
- Message-driven Bean を使用する場合

なお，JMS インタフェースと JDBC インタフェースで同一データベースにアクセスする場合と，Message-driven Bean を使用する場合は，コネクションの共有ができます。ただし，コネクションを共有するためには，次に示す前提条件を満たしている必要があります。

コネクション共有の前提条件

- ユーザプログラムで，同一トランザクション内で JMS インタフェースによるデータベース上のキューへのアクセスと，JDBC インタフェースによるテーブルへのアクセスを実施する。

- Reliable Messaging によってキューを構築するデータベースと，テーブルにアクセスするデータベースが同じである。
- JMS インタフェースよるデータベース上のキューへのアクセスと，JDBC インタフェースによるデータベース上のテーブルへのアクセスで使用するセキュリティ情報（ユーザ名，パスワード）と，サインオンの方式が同じである。
- DB Connector for Reliable Messaging と Reliable Messaging へのリファレンスを設定している J2EE アプリケーションで，DD の<res-sharing-scope>タグの値に「Shareable」が設定されている。

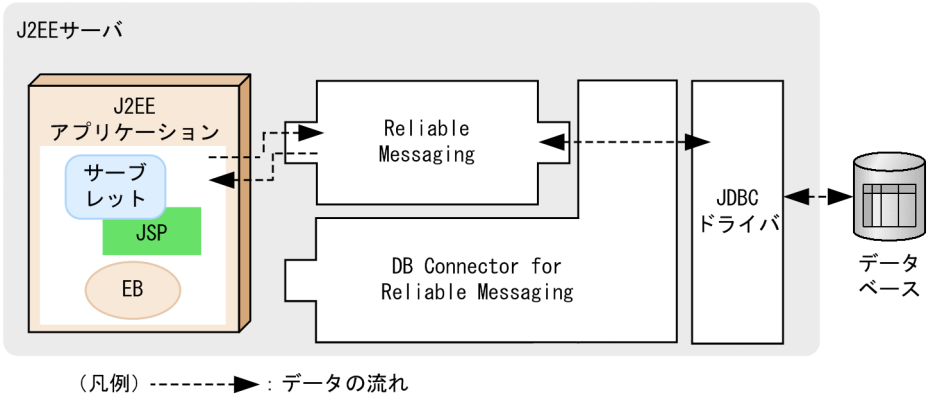
注意事項

- JDBC インタフェースだけを使用してデータベースにアクセスする場合，リソースアダプタとして DB Connector を使用してください。
- コネクション共有の前提条件を満たさない場合，DB Connector for Reliable Messaging と Reliable Messaging によって JMS インタフェースと JDBC インタフェースの両方を使用したデータベースへのアクセスはしないでください。この場合，「(c)　JMS インタフェースと JDBC インタフェースで異なるデータベースにアクセスする場合」に示す構成にして，テーブルへのアクセスには DB Connector を使用してください。

(a)　JMS インタフェースだけを使用する場合

ユーザプログラムでデータベース上のキューに対してだけアクセスする場合，次の図に示す構成で使用します。

図 3-22　JMS インタフェースだけを使用する場合の構成

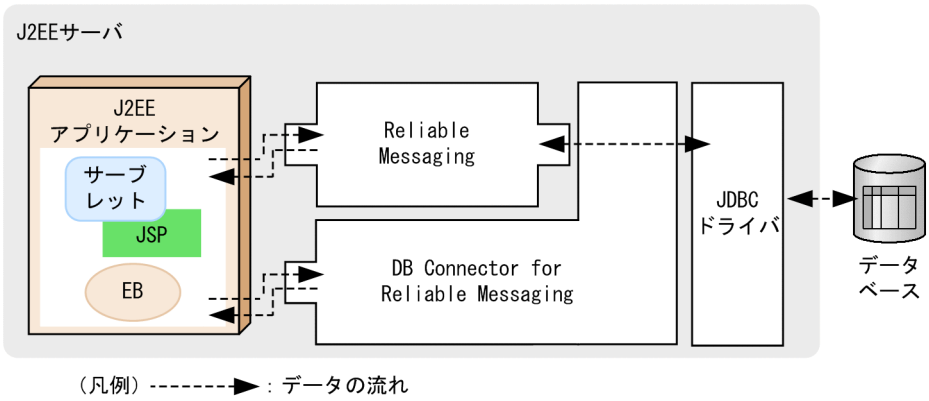


(b)　JMS インタフェースと JDBC インタフェースで同一データベースにアクセスする場合

ユーザプログラムで JMS インタフェースと JDBC インタフェースを使用してデータベースにアクセスする場合，コネクション共有の前提条件を満たしているときは，2 種類のインタフェースで同一のデータベースにアクセスできます。このとき，次の図に示す構成で使用します。

この構成にすることによって，コネクションの共有によるトランザクション処理性能の向上とリソースの有効利用を実現できます。

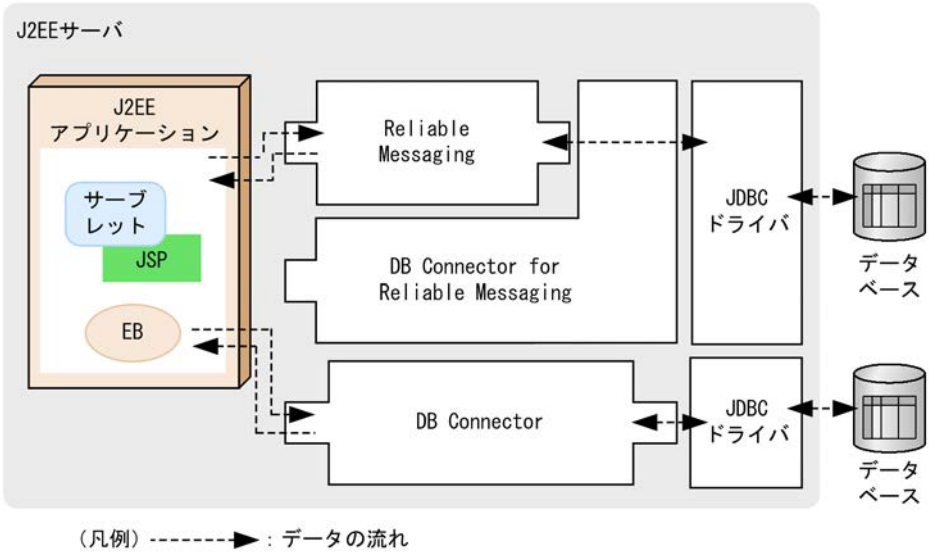
図 3-23　JMS インタフェースと JDBC インタフェースでアクセスするデータベースが同じ場合の構成



(c)　JMS インタフェースと JDBC インタフェースで異なるデータベースにアクセスする場合

ユーザプログラムで JMS インタフェースと JDBC インタフェースを使用してデータベースにアクセスする場合，コネクション共有の前提条件を満たしていないときは，インタフェースごとに，異なるデータベースにアクセスする構成にする必要があります。このとき，次の図に示す構成で使用します。

図 3-24　JMS インタフェースと JDBC インタフェースでアクセスするデータベースが異なる場合の構成



データベース上のキューへのアクセスには，DB Connector for Reliable Messaging と Reliable Messaging を使用します。また，データベース上のテーブルへのアクセスには，DB Connector を別途用意して使用する必要があります。

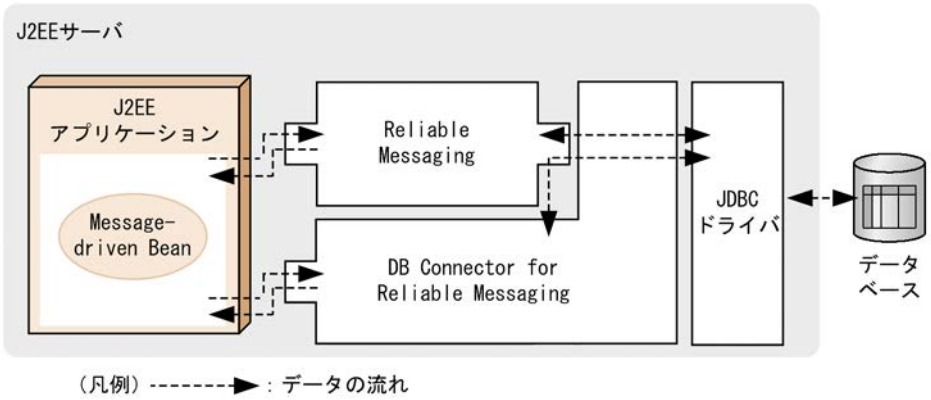
(d)　Message-driven Bean を使用する場合

Message-driven Bean を使用する場合，次の図に示す構成で使用します。コネクションの共有条件を満たす場合には，DB Connector for Reliable Messaging の JDBC インタフェースを使用して，Message-driven Bean と JDBC インタフェースでコネクションを共有することができます。

なお，Message-driven Bean を使用する場合，関連づけるリソースアダプタが Reliable Messaging 01-01 以降のときは，グローバルトランザクションおよびローカルトランザクションを使用できます。Reliable Messaging 01-00 のリソースアダプタを使用する場合は，グローバルトランザクションを使用する必要が

あります。このとき，ローカルトランザクションを使用することはできませんが，コネクション共有の前提条件を満たしていれば，グローバルトランザクションは 1 フェーズで決着されます。

図 3-25　Message-driven Bean を使用する場合の構成



3.7.3　使用できる機能

DB Connector for Reliable Messaging と Reliable Messaging による接続で使用できる機能の詳細については，マニュアル「Reliable Messaging」の「2.7　DB Connector for Reliable Messaging の機能」を参照してください。

なお，JDBC インタフェースを使用してデータベースに接続する場合，DB Connector を使用して接続する場合と同じ機能を利用できます。利用できる機能の詳細については，「3.3.4　リソースアダプタの機能」を参照してください。

3.7.4　接続できるデータベース

ここでは，DB Connector for Reliable Messaging と Reliable Messaging を使用して接続できるデータベースについて説明します。

(1)　接続できるデータベースの種類

DB Connector for Reliable Messaging と Reliable Messaging を使用して接続できるデータベースには，HiRDB，および Oracle があります。なお，SQL Server，および XDM/RD E2 には接続できません。

(2)　データベースと JDBC ドライバの対応

DB Connector for Reliable Messaging と Reliable Messaging を使用してデータベースに接続するためには，データベースに対応する JDBC ドライバが必要です。

接続先のデータベースと使用できる JDBC ドライバについて次の表に示します。

表 3-34　接続先のデータベースと使用できる JDBC ドライバ（データベース上のキューとの接続の場合）

データベース	JDBC ドライバ	
	HiRDB Type4 JDBC Driver	Oracle JDBC Thin Driver
HiRDB	○	×
Oracle	×	○

（凡例）　○：使用できる　　×：使用できない

(3)　注意事項

DB Connector for Reliable Messaging と Reliable Messaging を使用してデータベースに接続する場合の注意事項を次に示します。

- DB Connector for Reliable Messaging が提供する JDBC コネクション（java.sql.Connection. Connection）では，JDBC 固有のトランザクション制御はできません。JDBC 固有のトランザクション制御を行った場合，setAutoCommit(boolean)メソッドの引数 false での呼び出しのときと，releaseSavepoint(SavePoint)メソッド，rollback(Savepoint)メソッド，setSavepoint()メソッド，および setSavepoint(String)メソッドの呼び出しのときに例外が発生します。JDBC 固有のトランザクション制御を行う場合は，DB Connector を使用してください。
- DB Connector for Reliable Messaging の場合，性能解析トレースの取得ポイントおよびイベント ID が DB Connector の場合と一部異なります。

  JDBC コネクション（java.sql.Connection. Connection）に関しては，1 回のアクセスで DB Connector for Reliable Messaging の JDBC コネクションと，DB Connector での JDBC コネクションの二つのトレース取得ポイントで性能解析トレースが出力されます。

  また，JDBC コネクションの生成物（java.sql.Statement など）は，DB Connector と同様のトレース取得ポイントで性能解析トレースが出力されます。java.sql.DataSource は，DB Connector for Reliable Messaging 用のトレース取得ポイントでだけ性能解析トレースが出力されます。
- DB Connector for Reliable Messaging では，稼働情報監視をした場合，DB Connector for Reliable Messaging のリソースアダプタの次の項目について，正しい値が出力されません。DB Connector for Reliable Messaging はコネクションを Reliable Messaging 経由で取得するため，連携するリソースアダプタの監視稼働情報は Reliable Messaging 側に累積され出力されます。
  - トランザクションサポートレベル
  - プール現在値（総数）
  - 使用中のコネクション数
  - 未使用のコネクション数
  - ManagedConnectionFactory の createManagedConnection()メソッドの実行回数
  - ManagedConnection の getConnection()メソッドの実行回数
  - ManagedConnection の cleanup()メソッドの実行回数
  - ManagedConnection の destroy()メソッドの実行回数
  - ConnectionManager の allocateConnection()メソッドの実行時間
  - ManagedConnectionFactory の createManagedConnection()メソッドの実行時間
  - ConnectionManager の allocateConnection()メソッドの失敗回数
  - ManagedConnection で FATAL エラーが発生した回数
- DB Connector for Reliable Messaging は，連携する Reliable Messaging 側のコネクションプールを共有します。そのため，DB Connector for Reliable Messaging には，コネクションプールの各機能の設定をする必要はありません。
- リソースアダプタを使用する場合，J2EE アプリケーションからリソースアダプタへのリファレンスを解決しておく必要があります。リソースアダプタを使用している J2EE アプリケーションをカスタマイズするときに，J2EE アプリケーションからリソースアダプタへのリファレンスを解決しておいてください。

3.7.5　DB Connector for Reliable Messaging（RAR ファイル）の種類

DB Connector for Reliable Messaging を使用してデータベースに接続する場合，使用する JDBC ドライバに応じた RAR ファイルを使用します。RAR ファイルは，サーバ管理コマンドを使用して操作します。サーバ管理コマンドを使用して RAR ファイルを操作する方法については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」を参照してください。

JDBC ドライバの種類と対応する RAR ファイルについて次の表に示します。

表 3-35　JDBC ドライバと RAR ファイルの対応

JDBC ドライバ	RAR ファイル	説明
HiRDB Type4 JDBC Driver	DBConnector_HiRDB_Type4_CP_Cosminexus_RM.rar	トランザクション管理をしない場合，またはローカルトランザクションを使用する場合に使用します。
	DBConnector_HiRDB_Type4_XA_Cosminexus_RM.rar※	グローバルトランザクションを使用する場合に使用します。
Oracle JDBC Thin Driver	DBConnector_Oracle_CP_Cosminexus_RM.rar	トランザクション管理をしない場合，またはローカルトランザクションを使用する場合に使用します。
	DBConnector_Oracle_XA_Cosminexus_RM.rar※	グローバルトランザクションを使用する場合に使用します。

注※　ライトトランザクション機能は有効にできません。

3.7.6　HiRDB のキューに接続する場合の前提条件

ここでは，HiRDB のキューに接続する場合の前提条件について説明します。

(1)　前提条件

- **使用できる JDBC ドライバ**
  使用できる JDBC ドライバは HiRDB Type4 JDBC Driver となります。
- **接続方法**
  次のどちらかの RAR ファイルを使用します。
  - DBConnector_HiRDB_Type4_CP_Cosminexus_RM.rar
  - DBConnector_HiRDB_Type4_XA_Cosminexus_RM.rar

  RAR ファイルの種類に応じて，指定できるトランザクションサポートレベルが異なります。また，ライトトランザクションを使用できるかどうかも異なります。
  RAR ファイルごとに指定できるトランザクションサポートレベルを次の表に示します。

表 3-36　RAR ファイルごとに使用できるトランザクションサポートレベル

使用する DB Connector for Reliable Messaging (RAR ファイル)	トランザクションサポートレベル	ライトトランザクション	
		有効	無効
DBConnector_HiRDB_Type4_CP_Cosminexus_RM.rar	NoTransaction LocalTransaction	○	○
DBConnector_HiRDB_Type4_XA_Cosminexus_RM.rar	XATransaction	－	○

（凡例）　○：使用できる　　－：該当しない

3.7.7　Oracle のキューに接続する場合の前提条件

ここでは，Oracle のキューに接続する場合の前提条件について説明します。

- 使用できる JDBC ドライバ

  使用できる JDBC ドライバは Oracle JDBC Thin Driver となります。

- 接続方法

  DBConnector_Oracle_CP_Cosminexus_RM.rar または
  DBConnector_Oracle_XA_Cosminexus_RM.rar を使用します。

  RAR ファイルの種類に応じて，指定できるトランザクションサポートレベルが異なります。また，ライトトランザクションを使用できるかどうかも異なります。

  RAR ファイルごとに指定できるトランザクションサポートレベルを次の表に示します。

  表 3-37　RAR ファイルごとに使用できるトランザクションサポートレベル

使用する DB Connector for Reliable Messaging (RAR ファイル)	トランザクションサポートレベル	ライトトランザクション	
		有効	無効
DBConnector_Oracle_CP_Cosminexus_RM.rar	NoTransaction LocalTransaction	○	○
DBConnector_Oracle_XA_Cosminexus_RM.rar	XATransaction	－	○

（凡例）　○：使用できる　　－：該当しない

3.7.8　データベース上のキューに接続するための設定

データベース上のキューに接続するための設定については，マニュアル「Reliable Messaging」の「2.7 DB Connector for Reliable Messaging の機能」を参照してください。

3.8　OpenTP1 との Outbound での接続（SPP または TP1/Message Queue）

この節では，OpenTP1（SPP または TP1/Message Queue）と Outbound で接続する機能について説明します。

この節の構成を次の表に示します。

表 3-38　この節の構成（OpenTP1 との Outbound での接続（SPP または TP1/Message Queue））

分類	タイトル	参照先
解説	TP1 Connector による接続	3.8.1
	TP1/Message Queue - Access による接続	3.8.2
設定	OpenTP1 と Outbound で接続するための設定	3.8.3

注　「実装」および「運用」について，この機能固有の説明はありません。

アプリケーションサーバでは，OpenTP1 SPP または TP1/Message Queue と接続できます。OpenTP1 との接続には，次のリソースアダプタを使用します。

- OpenTP1 SPP との接続の場合，TP1 Connector を使用します。
- TP1/Message Queue との接続の場合，TP1/Message Queue - Access を使用します。

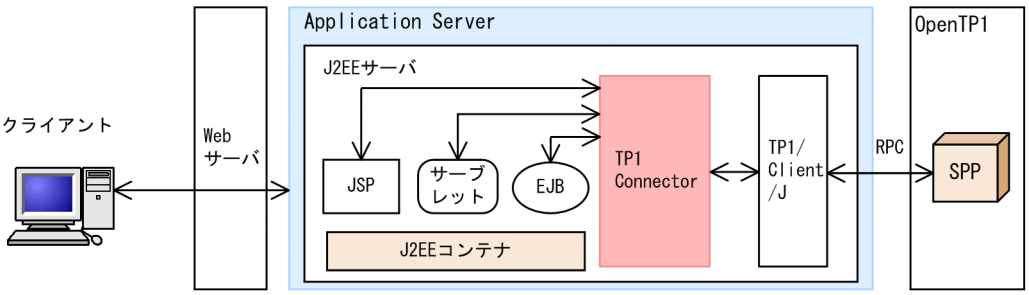
ここでは，OpenTP1 との接続について説明します。

3.8.1　TP1 Connector による接続

J2EE サーバと OpenTP1 SPP の接続には，リソースアダプタとして TP1 Connector と TP1/Client/J を組み合わせて利用します。

OpenTP1 との連携について次の図に示します。

図 3-26　TP1 Connector を利用した OpenTP1 との連携



TP1 Connector は，TP1/Client/J の RPC を使用して OpenTP1 と接続し，OpenTP1 の SPP にアクセスします。これによって，J2EE サーバでは，OpenTP1 の SPP と連携することができます。

! 注意事項

リソースアダプタを使用する場合，J2EE アプリケーションからリソースアダプタへのリファレンスを解決しておく必要があります。リソースアダプタを使用している J2EE アプリケーションをカスタマイズするときに，J2EE アプリケーションからリソースアダプタへのリファレンスを解決しておいてください。

TP1 Connector の設定項目の詳細については，TP1 Connector のドキュメントを参照してください。TP1/Client/J の設定項目の詳細については，マニュアル「OpenTP1 クライアント使用の手引 TP1/Client/J 編」を参照してください。

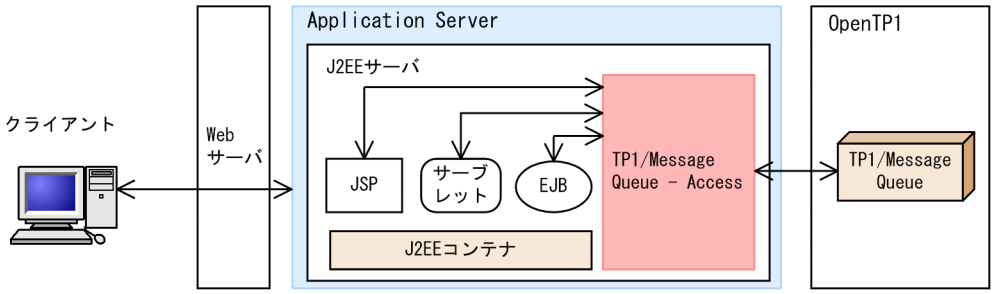
また，サーバ管理コマンドを使用したリソースアダプタの設定方法については，TP1 Connector のドキュメント，マニュアル「OpenTP1 クライアント使用の手引 TP1/Client/J 編」，およびマニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「4.4　そのほかのリソースと接続するための設定」を参照してください。

3.8.2　TP1/Message Queue - Access による接続

アプリケーションサーバと TP1/Message Queue の接続には，リソースアダプタとして TP1/Message Queue - Access を利用します。

TP1/Message Queue - Access を利用した TP1/Message Queue との連携について次の図に示します。

図 3-27　TP1/Message Queue - Access を利用した TP1/Message Queue との連携



TP1/Message Queue - Access では，JMS インタフェースを提供しています。この JMS インタフェースを利用して，J2EE サーバは TP1/Message Queue と接続します。

! 注意事項

リソースアダプタを使用する場合，J2EE アプリケーションからリソースアダプタへのリファレンスを解決しておく必要があります。リソースアダプタを使用している J2EE アプリケーションをカスタマイズするときに，J2EE アプリケーションからリソースアダプタへのリファレンスを解決しておいてください。

TP1/Message Queue - Access の設定項目の詳細については，マニュアル「OpenTP1 Version 7 メッセージキューイングアクセス機能 TP1/Message Queue - Access 使用の手引」を参照してください。

また，サーバ管理コマンドを使用したリソースアダプタの設定方法については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「4.4　そのほかのリソースと接続するための設定」を参照してください。

3.8.3　OpenTP1 と Outbound で接続するための設定

TP1 Connector によって接続する場合の設定については，TP1 Connector のドキュメント，マニュアル「OpenTP1 クライアント使用の手引 TP1/Client/J 編」，およびマニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「4.4　そのほかのリソースと接続するための設定」を参照してください。

TP1/Message Queue - Access によって接続する場合の設定については，マニュアル「OpenTP1 Version 7 メッセージキューイングアクセス機能 TP1/Message Queue - Access 使用の手引」，およびマ

ニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「4.4　そのほかのリソースと接続するための設定」を参照してください。

3.9　OpenTP1 との Inbound での接続

OpenTP1 を使用したレガシーシステム上の SUP から，アプリケーションサーバの J2EE サーバ上で動作している業務プログラムを呼び出す場合，OpenTP1 と J2EE サーバの接続には，リソースアダプタとして TP1 インバウンドアダプタを使用します。

TP1 インバウンドアダプタは，Connector 1.5 仕様に準拠したリソースアダプタです。また，次の機能を備えています。

- OpenTP1 のスケジュールサービスの代替機能
- OpenTP1 と RPC 通信を実行するための機能

これらの機能を使用することで，OpenTP1 の SUP からは，OpenTP1 の SPP を呼び出す場合と同様の手順で J2EE サーバ上の業務プログラムを呼び出せます。

TP1 インバウンドアダプタを使用した OpenTP1 との接続の詳細については，「4.　OpenTP1 からのアプリケーションサーバの呼び出し（TP1 インバウンド連携機能）」を参照してください。

! 注意事項

TP1 インバウンド連携機能は SUP，SPP および MHP からの呼び出しが可能ですが，このマニュアルでは，SUP からの呼び出しを例に説明します。

3.10　CJMS プロバイダとの接続

アプリケーションサーバが提供する JMS プロバイダの機能を使用する場合，メッセージの送信先の管理に CJMSP ブローカーというプロセスを使用します。J2EE サーバと CJMSP ブローカーとの接続には，CJMSP リソースアダプタを使用します。

CJMSP リソースアダプタを使用することで，PTP メッセージングモデルまたは Pub/Sub メッセージングモデルでのメッセージの送受信を実現できます。

CJMSP リソースアダプタを使用した OpenTP1 との接続の詳細については，「7.　CJMS プロバイダ」を参照してください。

3.11　SMTP サーバとの接続

J2EE アプリケーションは，JavaMail を使用して，SMTP サーバにメールを送信できます。

SMTP サーバとの接続の設定については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「6.3　メールコンフィグレーションの設定」を参照してください。

3.12　JavaBeans リソースの利用

この節では，JavaBeans リソースの利用について説明します。

この節の構成を次の表に示します。

表 3-39　この節の構成（JavaBeans リソースの利用）

分類	タイトル	参照先
解説	JavaBeans リソースの機能	3.12.1
	JavaBeans リソースの開始処理の流れ	3.12.2
実装	JavaBeans リソースの実装	3.12.3
設定	JavaBeans リソースの設定	3.12.4
	JavaBeans リソースの入れ替え	3.12.5

注　「運用」について，この機能固有の説明はありません。

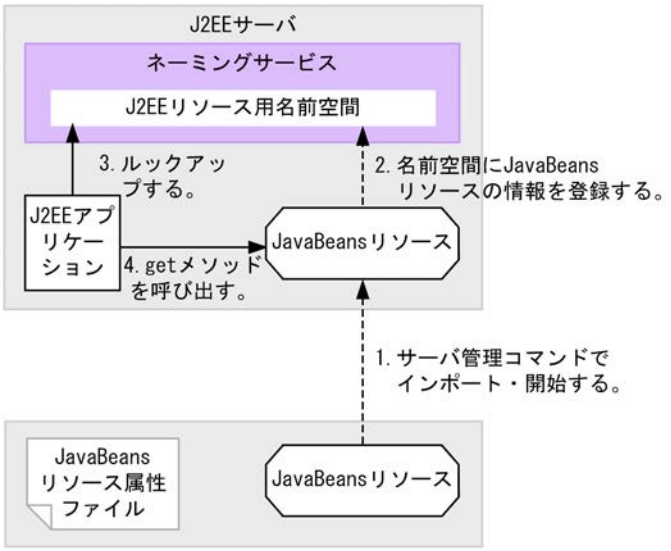
3.12.1　JavaBeans リソースの機能

JavaBeans リソースでは，J2EE アプリケーションを稼働させるための設定値などを一括して保持，管理できます。J2EE アプリケーションから JNDI を使用して JavaBeans リソースをルックアップすることで，設定値を取得できます。

3.12.2　JavaBeans リソースの開始処理の流れ

J2EE サーバで，複数の JavaBeans リソースを開始状態にできます。サーバ管理コマンドを使用した場合の，JavaBeans リソースの開始処理の流れを次の図に示します。

図 3-28　JavaBeans リソースの開始処理の流れ



1. JavaBeans リソースをインポートして，開始する。

サーバ管理コマンドを使用して，J2EE サーバに JavaBeans リソースをインポートして開始します。

2. ネーミングサービスに JavaBeans リソースのリファレンスを登録する。

JavaBeans リソースの開始時に，ネーミングサービスの J2EE リソース用の名前空間に JavaBeans リソースのリファレンスを登録します。

3. J2EE アプリケーションからネーミングサービスにルックアップする。

JavaBeans リソースを利用する J2EE アプリケーションから，ネーミングサービスにルックアップを行います。ルックアップ処理の延長で JavaBeans リソースの set メソッドが呼ばれて，JavaBeans リソース属性ファイルに指定したプロパティ値が設定されます。

4. 取得したリファレンスを使用して，JavaBeans リソースの get メソッドを呼び出す。

ルックアップして取得したリファレンスを使用して，JavaBeans リソースの get メソッドの呼び出しを行い，set メソッドで設定した値を get メソッドで取得します。

JavaBeans リソースの実装クラスは，単独のクラスで構成される場合はクラスファイルの形式で，JavaBeans リソースのクラスと関連クラスのように複数のクラスで構成される場合は JAR ファイルの形式で利用できます。

3.12.3　JavaBeans リソースの実装

JavaBeans リソースの実装手順を，例を用いて説明します。

(1)　インポート時の設定

サーバ管理コマンド（cjimportjb コマンド）で JavaBeans リソースをインポートするときに必要な設定と注意事項を示します。

(a)　JavaBeans リソース属性ファイルの設定

JavaBeans リソース属性ファイルは，次に示す点に留意して作成してください。

- <res-type>タグおよび<class-name>タグには，JavaBeans リソースのクラス名，実装クラス名を指定します。<res-type>タグおよび<class-name>タグで同じ値を設定する場合は<res-type>タグは省略できます。
- <property-name>タグには，set メソッドおよび get メソッドのメソッド名称を指定します。
- <property-type>タグには，set メソッドの引数の型を指定します。
  <property-type>タグの値と実際の該当 set メソッドの引数型と合わない場合は lookup 時にエラーになります。
- <property-value>タグには，set メソッドの引数に渡す値を指定します。

JavaBeans リソース属性ファイルの設定例を示します。

```
<!DOCTYPE hitachi-javabeans-resource-property PUBLIC '-//Hitachi, Ltd.//DTD JavaBeans Resource Property
7.0//EN'
'http://localhost/hitachi-javabeans-resource-property_7_0.dtd'>

<hitachi-javabeans-resource-property>
  <description></description>
  <display-name>JavaBean_resource</display-name>
    <class-name>com.mycompany.mypackage.MyJavaBean</class-name>
    <runtime>
      <property>
        <property-name>UserName</property-name>
        <property-type>java.lang.String</property-type>
        <property-value>Hitachi</property-value>
      </property>
      <property>
        <property-name>UserID</property-name>
        <property-type>java.lang.String</property-type>
```

```
        <property-value>01234567</property-value>
      </property>
    </runtime>
</hitachi-javabeans-resource-property>
```

JavaBeans リソース属性ファイルのテンプレートファイル（jb_template.xml）は，次のディレクトリに格納されています。

<Application Serverのインストールディレクトリ>¥CC¥admin¥templates

(b)　-d オプションの使い方

インポート時に-d オプションを使用することで，アーカイブを作成しないで，ディレクトリ構成のままインポートできます。-d で指定するディレクトリは，インポートするディレクトリの最上位を指定します。

インポートするときの-d オプションの指定例を次に示します。この指定例では，パッケージ名が com.mycompany.mypackage の MyJavaBean クラスをインポートします。

```
-dで指定するディレクトリ¥
  └ com¥
    └ mycompany
      └ mypackage
        └ MyJavaBean.class
```

-d オプションは指定されたディレクトリ下に存在するものをすべてインポートするので，不要なファイルはディレクトリ内に含めないでください。

複数の JavaBeans リソースをインポートする場合の注意

インポート済みの JavaBeans リソースと同じ実装クラス名の JavaBeans リソースはインポートできません。先にインポートした JavaBeans リソースを削除してからインポートするか，実装クラス名を変更し，作成し直してからインポートしてください。

(2)　JavaBeans リソースの実装クラスの作成

JavaBeans で管理するデータ（プロパティ）を操作するメソッドを宣言します。データを登録する場合，set メソッド（set ＋プロパティ名）を設定します。データを参照する場合，get メソッド（get ＋プロパティ名）を設定します。

次に，JavaBeans リソースを登録，参照するクラスの実装例を示します。

```
package com.mycompany.mypackage;
public class MyJavaBean {
    private String username;
    private String userid;

    public void setUserName(String user_name) {
        this.username = user_name;
    }
    public void setUserID(String user_id) {
        this.userid = user_id;
    }
    public String getUserName() {
        return this.username;
    }
    public int getUserID() {
        return this.userid;
    }
}
```

(3)　アプリケーションの設定

JavaBeans リソースを利用するときに，アプリケーション側で必要な実装および定義について説明します。

(a)　lookup の実装（JavaBeans リソース）

JavaBeans リソースを，ルックアップまたは DI を使用して取得します。ここでは，"java:comp/env"形式でルックアップする方法を示します。

```
Context initCtx = new InitialContext();
MyJavaBean jb = (MyJavaBean) initCtx.lookup("java:comp/env/bean/myJB");
```

JavaBeans リソースをルックアップできる範囲は，ほかのリソース同様に，同じ J2EE サーバプロセス内のアプリケーションです。

(b)　DD の定義内容（JavaBeans リソース）

ルックアップを使用して JavaBeans リソースを取得する場合，ルックアップする名前や実装クラス名の情報を DD（ejb-jar.xml または web.xml）に定義します。設定するタグを次に示します。

- <resource-env-ref-name>タグに，ルックアップで"java:comp/env"形式に指定する値を指定します。
- <resource-env-ref-type>タグに，JavaBeans リソースの実装クラス名を指定します。

また，作成したアプリケーションを J2EE サーバ上にデプロイするときに，ルックアップでの参照用の名称と実際の名称を linked-to で結び付けます。この操作をするには，サーバ管理コマンド（cjsetappprop コマンド）を使用します。

- cjgetappprop で該当アプリケーションの属性ファイルを取得します。
- <resource-env-ref>タグに<linked-to>タグを追加して，利用する JavaBeans リソースの表示名を指定します。
- cjsetappprop で該当アプリケーションの属性ファイルを設定します。

次に，cjsetappprop で渡す属性ファイルの<resource-env-ref>タグの例を示します。

```
<resource-env-ref>
  <resource-env-ref-name>bean/myJB</resource-env-ref-name>
  <resource-env-ref-type>com.mycompany.mypackage.MyJavaBean</resource-env-ref-type>
  <linked-to>JavaBean_resource</linked-to>
</resource-env-ref>
```

(4)　アプリケーションの開始と終了

JavaBeans リソースを利用するアプリケーションの開始，終了は，サーバ管理コマンドまたは Management Server で操作します。アプリケーションの開始方法については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「10.2.1　J2EE アプリケーションの開始」を参照してください。アプリケーションの終了方法については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「10.2.2　J2EE アプリケーションの停止」を参照してください。

アプリケーションサーバでは，JavaBeans リソースのサンプルプログラムを提供します。サンプルプログラムの概要および実行方法については，マニュアル「アプリケーションサーバ システム構築・運用ガイド」の「付録 M.6　JavaBeans リソースのサンプルプログラム」を参照してください。

3.12.4　JavaBeans リソースの設定

ここでは，JavaBeans リソースの設定について説明します。

サーバ管理コマンドを使用して JavaBeans リソースのプロパティを設定し，JavaBeans リソースをインポートします。

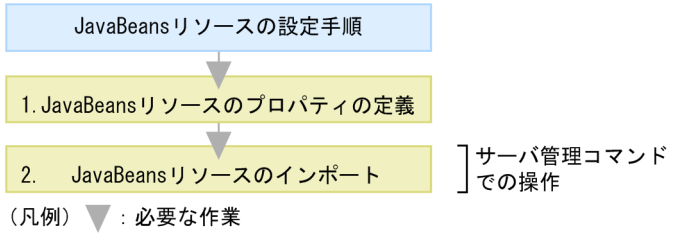
JavaBeans リソースの設定には，サーバ管理コマンドを使用します。

ここでは，JavaBeans リソースの新規設定の流れ，設定変更の流れ，および入れ替えの流れについて説明します。

(1)　JavaBeans リソースの新規設定の流れ

JavaBeans リソースの新規設定の流れを次の図に示します。

図 3-29　JavaBeans リソースの新規設定の流れ



図中の 1.～2.について説明します。

1. JavaBeans リソース属性ファイルを作成し，JavaBeans リソースのプロパティを定義します。

   JavaBeans リソース属性ファイルのテンプレートを使用して JavaBeans リソース属性ファイルを作成して，JavaBeans リソースのプロパティを定義してください。JavaBeans リソース属性ファイルのテンプレートは次のディレクトリに格納されています。

   - Windows の場合

     <Application Server のインストールディレクトリ>¥CC¥admin¥templates¥jb_template.xml

   - UNIX の場合

     /opt/Cosminexus/CC/admin/templates/jb_template.xml

   JavaBeans リソースのプロパティ定義で設定できる内容については，「(4)　JavaBeans リソースのプロパティ定義で設定できること」を参照してください。

2. サーバ管理コマンドを使用して，JavaBeans リソースをインポートします。

   1.で設定した JavaBeans リソース属性ファイルと，JavaBeans リソースを含む JAR ファイルのパスを引数に指定し，cjimportjb コマンドを使用して，JavaBeans リソースをインポートします。

サーバ管理コマンドでの操作については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「3.　サーバ管理コマンドの基本操作」を参照してください。また，コマンドについては，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「2.4　J2EE サーバで使用するリソース操作コマンド」を参照してください。属性ファイルについては，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.2　JavaBeans リソース属性ファイル」を参照してください。

参考

- JavaBeans リソースを複数インポートする場合，インポート済みの JavaBeans リソースと同じ実装クラス名の JavaBeans リソースはインポートできません。先にインポートした JavaBeans リソースを削除してからインポートするか，または実装クラス名を変更して再作成したあとでインポートしてください。

  また，JavaBeans リソースの実装クラス以外にほかのクラスファイルを使用している場合，ほかのクラスファイルに関してはチェックされません。

- インポート時に cjimportjb コマンドに-d オプションを使用すると，アーカイブを生成しないで，ディレクトリ構成のままインポートできます。ディレクトリは，インポートしたいディレクトリの最上位を指定します。

なお，指定したディレクトリがディレクトリ構成でない場合は，指定したディレクトリ直下のファイルをすべてインポートします。

指定したディレクトリ下にあるすべてのファイルをインポートするため，不要なファイルをディレクトリに含めないでください。

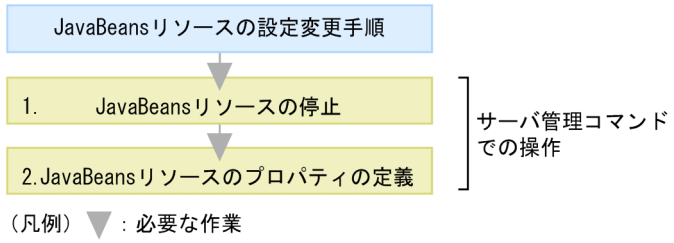
注意事項

JavaBeans リソースを使用する場合，J2EE アプリケーションから JavaBeans リソースへのリファレンスを解決しておく必要があります。JavaBeans リソースを使用している J2EE アプリケーションのプロパティを定義するときに，J2EE アプリケーションから JavaBeans リソースへのリファレンスを解決しておいてください。

(2)　JavaBeans リソースの設定変更の流れ

JavaBeans リソースの設定変更の流れを次の図に示します。

図 3-30　JavaBeans リソースの設定変更の流れ



図中の 1.～2.について説明します。

1. サーバ管理コマンドを使用して，JavaBeans リソースを停止します。

   cjstopjb コマンドを使用して JavaBeans リソースを停止します。なお，JavaBeans リソースを停止する前に，その JavaBeans リソースを使用している J2EE アプリケーションをすべて停止してください。

2. サーバ管理コマンドを使用して，JavaBeans リソースのプロパティを定義します。

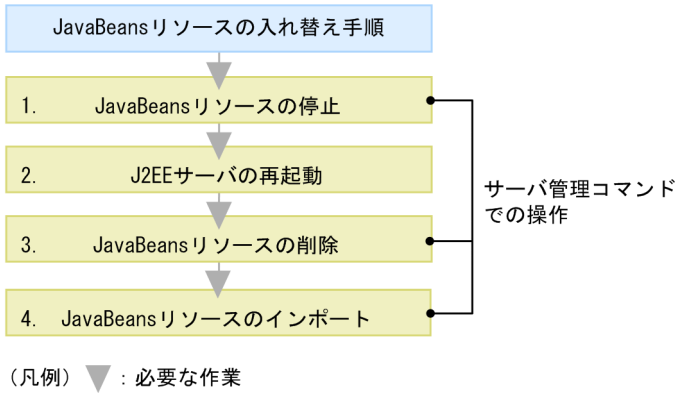
   cjgetjbprop コマンドを使用して JavaBeans リソース属性ファイルを取得し，ファイル編集後に，cjsetjbprop コマンドで編集内容を反映させます。

   JavaBeans リソースのプロパティ定義で設定できる内容については，「(4)　JavaBeans リソースのプロパティ定義で設定できること」を参照してください。

(3)　JavaBeans リソースの入れ替えの流れ

JavaBeans リソースの入れ替えの流れを次の図に示します。

図 3-31　JavaBeans リソースの入れ替えの流れ



図中の 1.～4.について説明します。

1. **サーバ管理コマンドを使用して，JavaBeans リソースを停止します。**
   cjstopjb コマンドを使用して入れ替える JavaBeans リソースを停止します。なお，JavaBeans リソースを停止する前に，その JavaBeans リソースを使用している J2EE アプリケーションをすべて停止してください。
2. **J2EE サーバを再起動します。**
   cjstopsv コマンドを使用して J2EE サーバを停止し，cjstartsv コマンドを使用して J2EE サーバを再起動します。
3. **サーバ管理コマンドを使用して，JavaBeans リソースを削除します。**
   cjdeletejb コマンドを使用して，入れ替える JavaBeans リソースを削除します。
4. **サーバ管理コマンドを使用して JavaBeans リソースをインポートします。**
   cjimportjb コマンドを使用して，新しい JavaBeans リソースをインポートします。

(4)　JavaBeans リソースのプロパティ定義で設定できること

JavaBeans リソースのプロパティを新規に設定する場合は，JavaBeans リソース属性ファイルのテンプレートを使用して JavaBeans リソース属性ファイルを作成して，プロパティを定義します。また，設定変更の場合は，サーバ管理コマンドで JavaBeans リソース属性ファイルを取得して，プロパティを定義します。cjgetjbprop コマンドで JavaBeans リソース属性ファイルを取得して，ファイル編集後に，cjsetjbprop コマンドで編集内容を反映させてください。

JavaBeans リソースのプロパティ定義で設定できる主な内容を次の表に示します。

表 3-40　JavaBeans リソースのプロパティ定義で設定できる主な内容

分類	項目	設定内容
実行時プロパティ	リソースのタイプ	<res-type>タグに JavaBeans リソースのクラス名を指定します。
	プロパティ情報	<property>タグ下の次のタグに設定します。 • <property-name>タグに JavaBeans リソースの set メソッド，get メソッドのメソッド名を指定します。 • <property-type>タグに JavaBeans リソースの set メソッドの引数の型を指定します。

分類	項目	設定内容
実行時プロパティ	プロパティ情報	• <property-value>タグに JavaBeans リソースの set メソッドの引数に渡す値を指定します。
別名情報	別名	JavaBeans リソースに別名を付与する場合，<resource-env-external-property>－<optional-name>タグに JavaBeans リソースの別名を指定します。 別名の設定については，「2.6.6　J2EE リソースの別名の設定」を参照してください。

参考

DD（ra.xml）の<resource-ref>タグで指定していた<res-auth>属性と，<res-sharing-scope>属性は，JavaBeans リソース属性ファイルの<resource-env-external-property>タグには指定できません。

3.12.5　JavaBeans リソースの入れ替え

ここでは，JavaBeans リソースを入れ替える手順について説明します。JavaBeans リソースは，J2EE アプリケーションおよび JavaBeans リソースを停止して，J2EE サーバを再起動してから入れ替えます。JavaBeans リソースの入れ替えには，サーバ管理コマンドを使用します。

手順を次に示します。

1. 入れ替える JavaBeans リソースを使用している J2EE アプリケーションを停止します。

   J2EE アプリケーションを停止するには，cjstopapp コマンドを実行します。cjstopapp コマンドの実行形式と実行例を次に示します。

   実行形式

```
cjstopapp 〈J2EEサーバ名〉 -name 〈J2EEアプリケーション名〉
```

   実行例

```
cjstopapp MyServer -name App1
```

2. 入れ替える JavaBeans リソースを停止します。

   JavaBeans リソースを停止するには，cjstopjb コマンドを実行します。cjstopjb コマンドの実行形式と実行例を次に示します。

   実行形式

```
cjstopjb 〈サーバ名称〉 -resname 〈JavaBeansリソース表示名〉
```

   実行例

```
cjstopjb MyServer -resname javabeansname
```

3. J2EE サーバを再起動します。

   いったん開始した JavaBeans リソースを削除するには，J2EE サーバを再起動する必要があります。J2EE サーバを停止するには，cjstopsv コマンドを実行します。cjstopsv コマンドの実行形式と実行例を次に示します。

   実行形式

```
cjstopsv 〈J2EEサーバ名〉
```

実行例

```
cjstopsv MyServer
```

J2EE サーバを停止したあと，もう一度開始します。J2EE サーバを開始するには，cjstartsv コマンドを実行します。cjstartsv コマンドの実行形式と実行例を次に示します。

実行形式

```
cjstartsv 〈J2EEサーバ名〉
```

実行例

```
cjstartsv MyServer
```

4. 入れ替える JavaBeans リソースを削除します。

JavaBeans リソースを削除するには，cjdeletejb コマンドを実行します。cjdeletejb コマンドの実行形式と実行例を次に示します。

実行形式

```
cjdeletejb 〈サーバ名称〉 -resname 〈JavaBeansリソース表示名〉
```

実行例

```
cjdeletejb MyServer -resname MyJavaBeans
```

5. 入れ替え後の JavaBeans リソースをインポートします。

JavaBeans リソースをインポートするには，cjimportjb コマンドを実行します。cjimportjb コマンドの実行形式と実行例を次に示します。

実行形式

```
cjimportjb 〈サーバ名称〉 -f 〈JARファイルパス〉 -c 〈JavaBeansリソース属性ファイルパス〉
```

実行例

```
cjimportjb MyServer -f Myjavabeans.jar -c Myjavabeansprop.xml
```

6. JavaBeans リソースを開始します。

JavaBeans リソースを開始するには，cjstartjb コマンドを実行します。cjstartjb コマンドの実行形式と実行例を次に示します。

実行形式

```
cjstartjb 〈サーバ名称〉 -resname 〈JavaBeansリソース表示名〉
```

実行例

```
cjstartjb MyServer -resname javabeansname
```

7. J2EE アプリケーションを開始します。

J2EE アプリケーションを開始するには，cjstartapp コマンドを実行します。cjstartapp コマンドの実行形式と実行例を次に示します。

実行形式

```
cjstartapp 〈J2EEサーバ名〉 -name 〈J2EEアプリケーション名〉
```

実行例

```
cjstartapp MyServer -name App1
```

3.13　そのほかのリソースとの接続

ここでは，これまでの節で説明したリソース以外のリソースとの接続について説明します。

3.13.1　そのほかのリソースとの接続に使用するリソースアダプタ

アプリケーションサーバでは，Connector 1.0 仕様または Connector 1.5 仕様に準拠したリソースアダプタを使用して，任意のリソースに接続できます。

Connector 1.5 仕様では，J2EE サーバとリソースアダプタ間の通信モデルとして，次の 2 種類のモデルが決められています。アプリケーションサーバでは，これらの通信モデルに対応したリソースアダプタを使用できます。

- Outbound
  J2EE サーバからリソースアダプタにアクセスする通信モデルです。
- Inbound
  リソースアダプタから J2EE サーバにアクセスする通信モデルです。

アプリケーションサーバでは，Connector 1.5 仕様の次の規約に対応しています。

- Lifecycle Management
  リソースアダプタの開始処理または終了処理を管理するための規約です。
- Work Management
  リソースアダプタがスレッドを扱うための規約です。
- メッセージインフロー
  EIS からのメッセージを受信して，リソースアダプタから Message-driven Bean を使用するための規約です。
- トランザクションインフロー
  メッセージインフローで，トランザクションを扱うための規約です。

これらの規約に基づいたアプリケーションサーバの機能については，「3.16.1　リソースアダプタのライフサイクル管理」，「3.16.2　リソースアダプタのワーク管理」，「3.16.3　メッセージインフロー」および「3.16.4　トランザクションインフロー」を参照してください。

また，既存の次の規約に対して Connector1.5 仕様で追加された機能についても，アプリケーションサーバで対応しています。

- Connection Management
  不正コネクションを検知できます。
  不正コネクションを検知する機能については，「3.15.1　コネクションの障害検知」を参照してください。
- Common Client Interface
  メッセージインフローに関連した API を使用できます。この API に対してアプリケーションサーバで独自に規定した仕様については，「3.16.7　アプリケーションサーバ独自の Connector 1.5 API 仕様」を参照してください。

3.13.2　そのほかのリソースとの接続で使用できる機能

Connector 1.5 で規定された機能のうち，アプリケーションサーバで使用できる機能について説明します。使用できる機能は，Outbound の場合と Inbound の場合で異なります。なお，Connector 1.5 仕様に準拠したリソースアダプタで使用できるアプリケーションサーバの機能については，「3.3.4　リソースアダプタの機能」を参照してください。

(1)　Outbound で使用できる機能

Outbound では，次の機能が使用できます。

- ローカルトランザクションおよびグローバルトランザクションによるトランザクション管理
- メッセージキューの使用

(2)　Inbound で使用できる機能

Inbound では，Non-Transacted Delivery（EIS がトランザクションに参加しないメッセージ配信）および Transacted Delivery（EIS がアプリケーションサーバのグローバルトランザクションに参加するメッセージ配信）を使用できます。

参考

トランザクションインフロー（Message-driven Bean の呼び出しが EIS のトランザクションに参加するメッセージ配信）は，TP1 インバウンド連携機能でだけ対応しています。

(3)　リソースアダプタの利用方法とトランザクションレベルの対応

リソースアダプタを Outbound で使用する場合に指定できるトランザクションレベルを次の表に示します。

表 3-41　Outbound で使用する場合に指定できるトランザクションレベル

利用する通信モデル	トランザクションの管理方法	トランザクション属性	トランザクションサポートレベル		
			NoTransaction	LocalTransaction	XATransaction※
Outbound	CMT	Required RequiresNew Supports NotSupported Mandatory Never	○	○	○
	BMT	－	○	○	○

（凡例）
○：使用できる　－：該当しない

注※　リソースアダプタを J2EE アプリケーションに含める場合は，XATransaction は使用できません。

ポイント

リソースアダプタを Inbound で使用する場合に指定できるトランザクション属性を次の表に示します。

表 3-42　Inbound で使用する場合に指定できるトランザクション属性

利用する通信モデル	トランザクションの管理方法	トランザクション属性
Inbound	CMT	Required※
		NotSupported
	BMT	－

（凡例）

－：該当しない

注※　CJMSP リソースアダプタまたは FTP インバウンドアダプタの場合は指定できません。

3.14　パフォーマンスチューニングのための機能

この節では，パフォーマンスチューニングのための機能について説明します。

この節の構成を次の表に示します。

表 3-43　この節の構成（パフォーマンスチューニングのための機能）

分類	タイトル	参照先
解説	コネクションプーリング	3.14.1
	コネクションプーリングで使用できる機能	3.14.2
	コネクションシェアリング・アソシエーション	3.14.3
	ステートメントプーリング	3.14.4
	ライトトランザクション	3.14.5
	インプロセストランザクションサービス	3.14.6
	DataSource オブジェクトのキャッシング	3.14.7
	DB Connector のコンテナ管理でのサインオンの最適化	3.14.8
実装	cosminexus.xml での定義	3.14.9
設定	実行環境での設定	3.14.10

注　「運用」について，この機能固有の説明はありません。

3.14.1　コネクションプーリング

サーブレット，JSP，EJB などの J2EE コンポーネントからのリソースアクセス量に応じて，リソースコネクション（JDBC コネクション，およびリソースアダプタのコネクション）をプーリングする機能です。コネクションをプーリングすることによって，ユーザアプリケーションからのリソース接続要求を高速に処理します。

(1)　前提条件

コネクションプーリング機能は，リソースの種類，接続方法の組み合わせによって，使用できる場合とできない場合があります。コネクションプーリング機能の使用について次の表に示します。

表 3-44　コネクションプーリング機能の利用可否

リソースの種類	接続方法	利用可否
データベース	DB Connector	○
データベース上のキュー	DB Connector for Reliable Messaging と Reliable Messaging	○
OpenTP1	TP1 Connector	○
	TP1/Message Queue - Access	○
SMTP サーバ	メールコンフィグレーション	×

リソースの種類	接続方法	利用可否
JavaBeans リソース	－	×
そのほかのリソース	Connector 1.0 仕様または Connector 1.5 仕様に準拠したリソースアダプタ	○

（凡例）○：使用できる　×：使用できない　－：該当しない

(2)　コネクションプールの生成および初期化

コネクションプールが生成・初期化されるタイミングは，リソースのスタート処理時です。コネクションプールのウォーミングアップ機能を有効にした場合，この時点でコネクションを生成します。コネクションプールのウォーミングアップ機能を無効にした場合，リソースのスタート時にコネクションは生成されず，最初のコネクション取得要求の発生時にコネクションを生成します。

コネクションプールの生成単位は次のとおりです。

- Connector 1.0 仕様に対応したリソースアダプタを使用する場合，リソース単位に一つのコネクションプールが生成されます。
- Connector 1.5 仕様に対応したリソースアダプタを使用する場合，コネクション定義ごとに一つのコネクションプールが生成されます。

(3)　コネクションプールの終了処理

リソースのアンデプロイ時や J2EE サーバの終了時は，コネクションプール内のすべてのコネクションを削除し，コネクションプール自体も削除します。なお，コネクションプールの終了処理では，コネクションに関するトランザクションなどがすべて決着済みのものとして処理します。

(4)　例外が発生したコネクションの破棄

データベース障害などが発生すると，コネクションプールに格納しているコネクションは使えなくなるため，コネクションプールから速やかに破棄する必要があります。

アプリケーションサーバは，コネクション，または Statement のようなコネクションからの生成物に対する処理で例外が発生すると，コネクションクローズ時に該当コネクションをコネクションプールから破棄します。ただし，ローカルトランザクションの決着処理が正常終了した場合には，コネクションが正常であると判断するため破棄しません。

コネクションが正常に維持している状態で，コネクションやコネクションからの生成物で例外が発生すると，グローバルトランザクションを使用している場合には，トランザクションの決着処理が正常終了してもアプリケーションサーバはコネクションをコネクションプールに戻さないで破棄します。そのため余分なコネクション生成が発生して性能に影響を与えることがあります。

なお，トランザクションタイムアウト発生時には，トランザクション種別に関係なくコネクションをコネクションプールから破棄します。

(5)　コネクションプール利用上の注意事項

コネクションプールを利用する場合の注意事項について説明します。

- データベースサーバ側から強制的に切断する機能（タイムアウトなど）を利用しないでください。
- コネクションプールで管理するコネクションは，取得時に使用した認証情報（ユーザ名，パスワードなど）を保持するため，サインオンの形態によっては注意が必要です。

コンテナ管理でのサインオンの場合

一つのコネクションプールに対して，コネクション取得要求時に使用する認証情報は常に一つとなるため，特に注意は必要ありません。

コンポーネント管理でのサインオンの場合

複数のユーザ名とパスワードの組み合わせを利用する場合に注意が必要です。コネクションプールはリソースごとに一つであるため，一つのリソースに対して複数のユーザが利用する場合，複数のユーザで一つのコネクションプールを共有することになります。この場合，一人のユーザが，コネクションプールの最大値に設定した数までコネクションを利用できないことがあります。

また，コネクションプール内の未使用コネクションの中に，認証情報が一致するコネクションがない場合，プール内のコネクション総数によって，次のように動作します。

コネクション総数が，指定したコネクションプールの最大値に達している場合，未使用状態のコネクションを破棄して，新規にコネクションを生成します。

コネクション総数が，指定したコネクションプールの最大値に達していない場合，新規にコネクションを生成します。

(6)　コネクションプーリングの動作

リソースアダプタのコネクションプーリングの動作を次の表に示します。

表 3-45　コネクションプールの状態と動作

ユーザアプリケーションプログラム処理	コネクションプールの状態	コネクションプールの動作
コネクションの取得要求	プール内に未使用状態のコネクションがある	未使用状態の時間がいちばん長いコネクションが選択され，ユーザアプリケーションプログラムに渡されます。選択されたコネクションは，プール内で使用中状態になります。
	プール内に未使用状態のコネクションがなく，プール内のコネクションの総数が「MaxPoolSize」の値未満	新規にコネクションを確立します。確立したコネクションはユーザアプリケーションに渡され，コネクションはプール内で使用中状態となります。
	プール内に未使用状態のコネクションがなく，プール内のコネクションの総数が「MaxPoolSize」の値に達している	ユーザアプリケーションプログラムに例外が通知され，コネクションの取得は失敗します。再取得するためには，「Retry Count」と「Retry Interval」を設定します。
	プール内に未使用状態のコネクションがあるが，認証情報が一致するコネクションがない	プール内のコネクションの総数によって次の処理が実施されます。 • **プール内のコネクションの数が「MaxPoolSize」の値未満の場合** 新規にコネクションを確立し，ユーザアプリケーションに引き渡されます。 • **プール内のコネクションの数が「MaxPoolSize」の値に達している場合** 使用されていないコネクションのうち，最初にプーリングされたコネクションが破棄されたあと，新しいコネクションが作成されて，ユーザアプリケーションに引き渡されます。

ユーザアプリケーションプログラム処理	コネクションプールの状態	コネクションプールの動作
コネクションを解放	解放したコネクションに異常がなく，再利用できる	このコネクションはプール内で未使用状態に戻ります。
	解放したコネクションが再利用できない	このコネクションは破棄されます。
－	コネクションプールのウォーミングアップ機能を使用	リソースアダプタの開始時またはリソースアダプタを開始済みの状態でのサーバ起動時に，「MinPoolSize」の値までコネクションが生成されてプールされます。 コネクションプールのウォーミングアップ機能を使用するためには，「Warmup」を設定します。
	コネクション枯渇時のコネクション取得待ち	コネクションプールにコネクションが最大数プールされていて利用できるコネクションがプールにない状態（コネクション枯渇）のときに，コネクションの取得要求を待ち状態にできます。 待ち状態になっているコネクション取得要求は，コネクションが解放されるとすぐにコネクションを取得できます。 コネクション枯渇時のコネクション取得待ちは，「RequestQueueEnable」「RequestQueueTimeout」で設定します。

（凡例）　－：該当なし

なお，コネクションプールのウォーミングアップ機能を使用する場合は，次の表に示す注意事項があります。

表 3-46　コネクションプールのウォーミングアップ機能を使用する場合の注意事項

条件	注意事項
コンテナ認証用の認証情報(ユーザ名，パスワードなど）の設定※	コンポーネント管理でのサインオンで，複数のユーザ名とパスワードの組み合わせで利用する場合に注意が必要です。コネクションプールはリソースごとに一つであるため，一つのリソースに対して複数のユーザが利用する場合，複数のユーザで一つのコネクションプールを共有することになります。この場合，一人のユーザが，コネクションプールの最大値に設定した数までコネクションを利用できないことがあります。 コンテナ管理でのサインオンの場合は，一つのコネクションプールに対して，コネクション取得要求時に使用する認証情報は常に一つとなるため，特に注意は必要ありません。
「MinPoolSize」の設定	次のような場合，MinPoolSize に指定した値よりもプールされているコネクションが少なくなることがあります。 • コネクションスイーパによってコネクション解放処理が実行されたとき • cjclearpool（コネクションプール内のコネクション削除）コマンドを実行したとき • コネクションに障害が発生したとき リソースの開始時に MinPoolSize 分のコネクションが生成されるため，コネクションプールのウォーミングアップ機能を使用しない場合に比べてサーバの起動またはリソースアダプタの開始に時間が掛かります。

条件	注意事項
「MinPoolSize」の設定	また，このときに「bufSize に指定した値 × 生成したコネクション数分のメモリ」を Java ヒープ領域に確保します。 MinPoolSize に必要以上に大きな値を設定してウォーミングアップ機能を使用すると，メモリを確保したときに Java ヒープを使い切り，OutOfMemoryError が発生するおそれがあります。 このため，MinPoolSize の値は，使用するリソースマネジャの最大同時接続数以下に設定してください。

注※　コネクションプールで管理するコネクションは，ウォーミングアップ機能の動作時に使用したコンテナ認証用の認証情報（ユーザ名，パスワードなど）を保持します。

3.14.2　コネクションプーリングで使用できる機能

コネクションプーリング機能を使用すると，さらに次に示す機能も使用できます。

(1)　コネクションプールのウォーミングアップ

コネクションプールのウォーミングアップとは，サーバ起動時またはリソースアダプタのスタート時に実行されるリソースのスタート処理で，あらかじめ，コネクションを，コネクションプールの設定で定義した最小値までプーリングする機能です。これによって，コネクションプールの使用を開始した直後の，コネクション要求のレスポンスを向上できます。

コネクションプールのウォーミングアップは，リソースアダプタの属性（プロパティ）として設定します。リソースアダプタのプロパティ定義で設定できることについては，「3.14.10　実行環境での設定」を参照してください。

(2)　コネクション数調節機能

コネクション数調節機能とは，コネクションプール内の不要なコネクションを指定したコネクションプールの最小値から最大値の範囲で段階的に減少させる機能です。この機能を有効にすると，実際の稼働実績に適した数までプール内のコネクション数を徐々に減らせるため，指定したコネクションプールの最小値を超えた場合のコネクション生成コストの削減やリソース資源の節約ができます。

また，コネクション数の調節時のコネクション削除処理に対して，応答時間にタイムアウトを設定できます。サーバ障害やネットワーク障害が発生してリソースからの応答が返らない場合，コネクション削除処理に対しても応答が返らなくなることがあります。このように，リソースから応答が返らない場合でも，タイムアウトを設定することでコネクション削除処理を終了し，処理を継続できます。

コネクション数調節機能の動作

コネクション数調節機能を使用すると，プール内のコネクション数は，前回コネクション数を調節したときから今回コネクション数を調節するまでの間に同時に使用されたコネクションの最大数で調節されます。コネクション数の調節時に，プール内コネクション数が同時に使用されたコネクションの最大数より多い場合は，コネクションの削除処理が動作します。例えば，現在，プール内にあるコネクション数が 8 の場合で，前回のコネクション数調整から今回のコネクション数調節までの間に同時に使用されたコネクションの最大数が 5 のときは，プール内のコネクション数が 3 多いため，コネクションプールから 3 コネクションが削除され，調節後のコネクション数は 5 となります。

なお，コネクション数調節機能は一定間隔で動作します。

コネクション数調節機能の設定については，「3.14.10　実行環境での設定」を参照してください。

コネクション削除処理のタイムアウト

コネクション数調節機能のコネクション削除処理に対して，その応答時間にタイムアウトを設定できます。なお，コネクション削除処理のタイムアウト時間は簡易構築定義ファイルの J2EE サーバで指定するキーに，任意の時間を指定できます（デフォルト値は 5 秒）。

ただし，コネクションプールの最大数が無制限の場合，コネクション削除処理のタイムアウトは無効になります。

また，コネクション削除処理のタイムアウトにはコネクション管理スレッドを使用するため，コネクション削除処理のタイムアウトを設定すると，設定しない場合に比べて多くのメモリを消費します。設定する場合は，必要なメモリ使用量を適切に見積もってください。

コネクション管理スレッドについては，「3.15.1　コネクションの障害検知」を参照してください。

(3)　コネクションスイーパの動作

一定間隔でコネクションプール内の未使用コネクションを破棄するための，コネクションスイーパ機能は，次のように動作します。

- 前回のコネクションスイーパの動作が終了してから「SweeperInterval」の値の経過後，コネクションスイーパが動作します。
- コネクションスイーパは，プール内の未使用状態のコネクションを監視します。

  最後に利用した時点からの経過時間が「ConnectionTimeout」の値以上の未使用状態コネクションを破棄します。未使用状態コネクションで，最後に利用した時点からの経過時間が「ConnectionTimeout」の値未満のコネクションについては，何もしません。

3.14.3　コネクションシェアリング・アソシエーション

コネクションシェアリングとコネクションアソシエーションは，コネクション共有機能です。コネクション共有機能を利用することで，リソースを効率的に使用し，パフォーマンスを向上できます。必要に応じて，コネクションアソシエーション機能を有効にしてください。

ローカルトランザクションでトランザクション管理を行っている場合，一つのリソースコネクションだけでデータベースなどのリソースにアクセスする必要があります。コネクション共有機能を使用するとユーザアプリケーションで意識することなく，一つのリソースコネクションだけでリソースアクセスができます。

グローバルトランザクションの場合も同様に，一つのリソースコネクションしかトランザクションに参加しない場合は，1 フェーズコミットに最適化されるため，トランザクションの決着コストを下げることができます。

(1)　物理コネクションと論理コネクション

物理コネクションとは，接続先リソースのコネクションを指します。通常，サーブレットや Enterprise Bean などの J2EE コンポーネントが直接操作することはなく，コンテナが操作します。例えば，リソースアダプタの場合は ManagedConnection（javax.resource.spi.ManagedConnection）です。

論理コネクションとは，サーブレットや Enterprise Bean などの J2EE コンポーネントが直接操作するコネクションを指します。例えば，リソースアダプタの場合は javax.resource.cci.Connection やリソースアダプタが独自に提供するコネクションです。

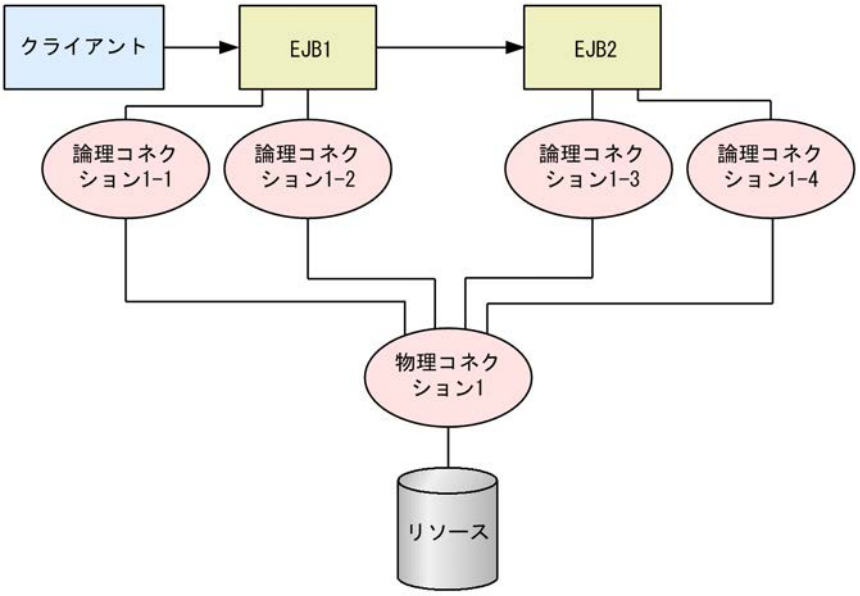
物理コネクションと論理コネクションの関係は，一般的に，物理コネクションから論理コネクションを生成するという関係になります。物理コネクションはコネクションプールで管理され，コネクションプールが物理コネクションの取得およびクローズを行います。

サーブレットや Enterprise Bean などの J2EE コンポーネントからのコネクション取得要求に対しては，コネクションプールが物理コネクションから論理コネクションを生成して返却します。コネクション解放要求に対しては，論理コネクションだけをクローズし，物理コネクションはクローズしないでプール管理します。

(2)　コネクションシェアリング

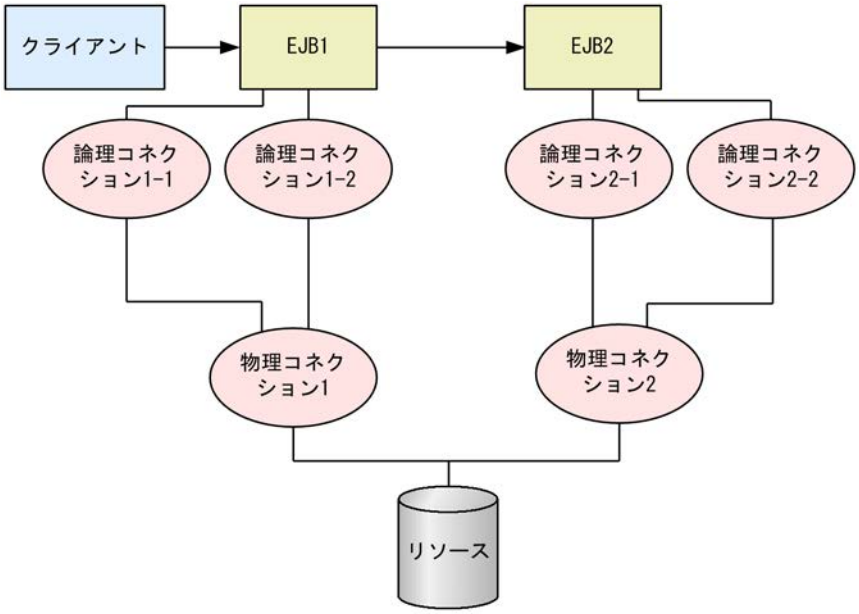
コネクションアソシエーション機能が有効でない場合，アプリケーションサーバが管理するトランザクション内でコネクションシェアリングが行われます。トランザクション内コネクションシェアリングは，リソースコネクションを最も効率的に使用します。トランザクション内のコネクションシェアリングについて次の図に示します。

図 3-32　論理コネクションと物理コネクションの関係（トランザクション内コネクションシェアリング）



コネクションアソシエーション機能が有効な場合，およびアプリケーションサーバが管理するトランザクションの外でのコネクションシェアリングは，サーブレットや Enterprise Bean などの J2EE コンポーネントインスタンス内でコネクションシェアリングが行われます。コンポーネント内のコネクションシェアリングについて，次の図に示します。

図 3-33　論理コネクションと物理コネクションの関係（コンポーネント内コネクションシェアリング）



(a)　コネクションシェアリングの条件

コネクションシェアリングをするためには，次の条件をすべて満たす必要があります。

- 同一の J2EE サーバ内であること
- 同一のリソースであること
- サインオンの方法とセキュリティ情報（ユーザ名とパスワード）が同じであること
- 標準 DD の<res-sharing-scope>に Shareable が指定されていること
- 同一のアプリケーションサーバが管理するトランザクション内であること※

注※

アプリケーションサーバが管理するトランザクションの外でもコネクションシェアリングを行うことができます。

アプリケーションサーバが管理するトランザクションの外でコネクションシェアリングを有効にする場合は，J2EE サーバのプロパティをカスタマイズして設定します。J2EE サーバの動作設定のカスタマイズについては，「3.14.10　実行環境での設定」を参照してください。

なお，リソースアダプタのトランザクションサポートレベルに NoTransaction を指定した場合は，コネクションシェアリングは行われません。

なお，コネクションシェアリングの定義については，「3.14.9　cosminexus.xml での定義」を参照してください。

(b)　コネクションシェアリングの定義

コネクションを共有するかどうかは，サーブレットや Enterprise Bean の標準 DD または cosminexus.xml の<res-sharing-scope>タグで指定します。リソース参照ごとに指定できます。コネクションの共有はデフォルトで有効になっています。無効にする場合は，<res-sharing-scope>に Unshareable を指定します。

J2EE アプリケーションの設定については，「3.14.9　cosminexus.xml での定義」を参照してください。

(c)　注意事項

java.sql.Connection を，複数のトランザクション間で再利用することはできません。

java.sql.Connection をトランザクション内で利用する場合には，トランザクションごとに，javax.sql.DataSource から getConnection()メソッドを使用してコネクションを取得してください。

なお，java.sql.Connection は，トランザクション内とトランザクション外との間でも再利用することはできません。

(3)　コネクションアソシエーション

トランザクション範囲を超えた永続性を持つコネクションを使用する場合，トランザクション内コネクションシェアリングを使用できません。その場合は，コネクションアソシエーション機能を有効にしてください。

コネクションアソシエーションは，論理コネクションと物理コネクションの関係を切り替えて，一つのリソースコネクションによるリソースアクセスを実現します。

図 3-34　論理コネクションと物理コネクションの関係（コネクションアソシエーション）

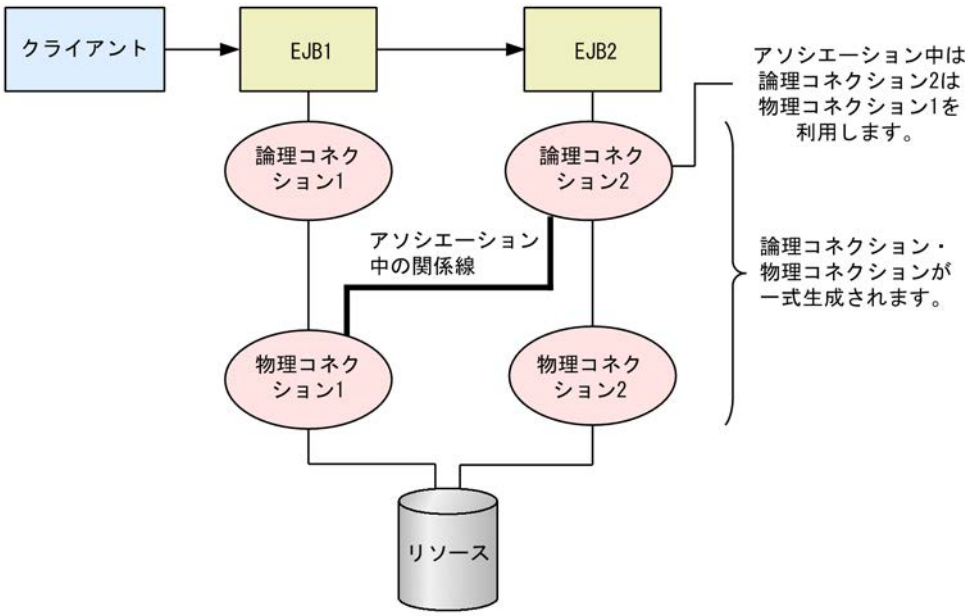
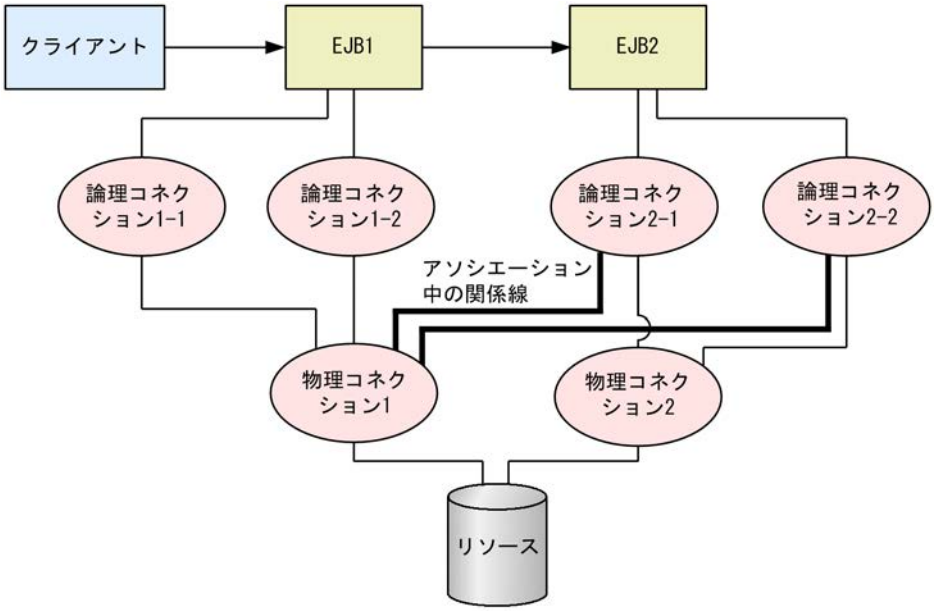


図 3-35　論理コネクションと物理コネクションの関係（コネクションアソシエーションとコネクションシェアリングの併用時）



(a)　コネクションアソシエーションの条件

コネクションアソシエーションをするためには，次の条件のすべてを満たす必要があります。

- コネクションアソシエーション機能が有効であること
- 同一の J2EE サーバ内であること
- 同一のリソースであること
- サインオンの方法とセキュリティ情報（ユーザ名とパスワード）が同じであること
- 標準 DD の<res-sharing-scope>に Shareable が指定されていること※
- 同一のアプリケーションサーバが管理するトランザクション内であること

注※

usrconf.properties の ejbserver.connectionpool.association.enabledDespiteUnshareableSetting キーに「true」を設定すると標準 DD の<res-sharing-scope>に Unshareable が指定されていてもコネクションアソシエーションを行います。

このプロパティは下位互換性のためだけに提供されています。

(b)　コネクションアソシエーションの定義

コネクションアソシエーションはデフォルトで無効になっています。

コネクションアソシエーションを有効にするための設定は，J2EE サーバのプロパティをカスタマイズして設定します。J2EE サーバの動作設定のカスタマイズについては，「3.14.10　実行環境での設定」を参照してください。

(c)　注意事項

java.sql.Connection からの生成物（例：java.sql.Statement）をトランザクションの範囲を超えて使用することはできません。

3.14.4　ステートメントプーリング

DB Connector を使用した場合，JDBC の API である java.sql.PreparedStatement と java.sql.CallableStatement の再利用を行うプーリング機能を使用できます。ステートメントプーリング機能によって，PreparedStatement と CallableStatement を使用した場合のパフォーマンス向上が図れます。なお，PreparedStatement と CallableStatement のプールサイズは，DB Connector の設定で指定します。ステートメントプーリングを使用する際のプールサイズ設定の指針については，マニュアル「アプリケーションサーバ システム設計ガイド」の「8.5.2　ステートメントプーリングを使用する」を参照してください。また，DB Connector のプロパティ定義については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「4.2.2　DB Connector のプロパティ定義」を参照してください。

ステートメントプーリング機能では，ステートメントを再利用する際にステートメントを初期化します。初期化する内容は J2EE サーバのプロパティをカスタマイズして設定します。ステートメントプーリング機能の設定については，「3.14.10　実行環境での設定」を参照してください。

ステートメントプーリング機能を利用するには，コネクションプーリング機能を使用する必要があります。また，トランザクションサポートレベルにグローバルトランザクションを指定した場合，HiRDB のバージョンによっては利用できないことがあります。

! 注意事項

HiRDB Type4 JDBC Driver を使用する場合，PreparedStatement，CallableStatement のプールサイズに制限があります。指定できるプールサイズについては，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1.10　DB Connector に設定する<config-property>タグに指定できるプロパティ」を参照してください。

なお，ステートメントプーリング機能を使用しない場合は，PreparedStatement，および CallableStatement のプールサイズを両方とも 0 にしてください。

(1)　前提条件

トランザクションサポートレベル，コネクションプーリング使用の有無，使用するデータベースの種類との対応による，ステートメントプーリング利用について次の表に示します。

表 3-47　ステートメントプーリングの利用

トランザクションサポートレベル	コネクションプーリングを使用する				コネクションプーリングを使用しない
	HiRDB※1	Oracle	SQL Server	XDM/RD E2	HiRDB／Oracle／SQL Server
トランザクションなし（NoTransaction）	○	○	○	○	×
ローカルトランザクション（LocalTransaction）	○	○	○	○	×
グローバルトランザクション（XATransaction）	○※2	○	×	×	×

（凡例）　○：利用できる　　×：利用できない

注※1　ステートメントプーリング機能を使用した状態で，定義 SQL を実行しないでください。定義 SQL を実行する場合，ステートメントプーリング機能は使用できません。また，定義 SQL を実行する場合は，HiRDB のクライアント環境変数として「PDDDLDEAPRP=YES」を設定する必要があります。

注※2　JDBC の DatabaseMetaData#supportsStatementPooling()の戻り値が true の場合に，利用できます。

なお，XDM/RD E2 を使用する場合は XDM/RD E2 11-03 以降のバージョンと HiRDB Type4 JDBC Driver 08-02 以降を使用するときにだけステートメントプーリング機能を使用できます。ステートメントプーリング機能の設定については，「3.14.10　実行環境での設定」を参照してください。

(2)　ステートメントプーリングの動作

リソースアダプタのコンフィグレーションプロパティで設定するステートメントプーリングの動作について説明します。

ステートメントプーリング機能の動作を，次の表に示します。

表 3-48　ステートメントプールの状態と動作

ユーザアプリケーションプログラム処理	ステートメントプールの状態	ステートメントプールの動作
PreparedStatement，CallableStatement の生成を要求	プール内に未使用状態の PreparedStatement，CallableStatement がある	プール内で未使用状態の PreparedStatement，CallableStatement の一つが選択され，ユーザアプリケーションプログラムに渡されます。選択された PreparedStatement，CallableStatement は，プール内で使用中状態になります。
	プール内に未使用状態の PreparedStatement，CallableStatement がなく，プール内の PreparedStatement，CallableStatement の総数が「PreparedStatementPoolSize」，「CallableStatementPoolSize」の値に満たない	新規に PreparedStatement，CallableStatement を生成します。生成した PreparedStatement，CallableStatement はユーザアプリケーションに渡され，PreparedStatement，CallableStatement はプール内で使用中状態となります。
	プール内に未使用状態の PreparedStatement，CallableStatement がなく，プール内の PreparedStatement，CallableStatement の総数が「PreparedStatementPoolSize」，「CallableStatementPoolSize」の値以上	タイムスタンプが最も古い※ PreparedStatement，CallableStatement をプールから削除したあと，新規に PreparedStatement，CallableStatement を生成します。生成した PreparedStatement，CallableStatement はユーザアプリケーションに渡され，PreparedStatement，CallableStatement はプール内で使用中状態となります。
PreparedStatement，CallableStatement を解放	－	PreparedStatement，CallableStatement はプール内で未使用状態に戻ります。

（凡例）　－：該当なし

注※　プール内の PreparedStatement，CallableStatement のタイムスタンプが更新されるタイミングは，次のとおりです。

- 新規に生成した PreparedStatement，CallableStatement がプールに追加されるとき

- プール内の PreparedStatement，CallableStatement が使用中状態になるとき

(3)　ステートメントプーリング機能を使用する場合の注意事項

ステートメントプーリング機能を使用する場合の注意事項を次の表に示します。

表 3-49　ステートメントプーリング機能を使用する場合の注意事項

条件	注意事項
－	• ステートメントプーリング機能を使用しない場合と比べて，プールされている PreparedStatement，CallableStatement の数だけメモリが消費されます。各ステートメントのメモリ使用量については，使用する JDBC ドライバのマニュアルを参照してください。 • ステートメントプールは，コネクションプール内のコネクションごとにあるため，プールされる PreparedStatement，CallableStatement の最大数は，「MaxPoolSize」×（「PreparedStatementPoolSize」＋「CallableStatementPoolSize」）となります。
－	• ステートメントプーリング機能をコネクション数調節機能またはコネクション障害検知機能と併用した場合，コネクションプールから取り除かれた未使用コネクションは，コネクションプール内のコネクション数としてカウントされません。このため，プールされている PreparedStatement，CallableStatement の数がコネクションプールの最大値×PreparedStatementPoolSize，およびコネクションプールの最大値×CallableStatementPoolSize を一時的に超える場合があります。 • ステートメントプーリング機能を使用してステートメントを再利用する場合，setMaxFieldSize の設定値が初期化されないことがあります。 初期化されないのは，次の条件がすべて重なるときです。 ・Oracle JDBC Thin Driver に対応した DB Connector を使用した ・setMaxFieldSize メソッドを使用して，java.sql.PreparedStatement，または java.sql.CallableStatement の値を変更した
Oracle または SQL Server に接続	• **ステートメントプーリング機能と，コネクションの障害検知機能を併用する場合** コネクションの障害検知に使用される PreparedStatement がコネクションごとに一つプーリングされます。そのため，PreparedStatementPoolSize，CallableStatementPoolSize を決定する際には，コネクションの障害検知用の PreparedStatement もリソース数の見積もりに加えて，CallableStatementPoolSize を最大ステートメント数未満に設定してください。※ • **ステートメントプーリング機能と，コネクションプールのウォーミングアップ機能を併用する場合** コネクションが生成されてプーリングされるときに，コネクションの障害検知に使用される SQL で生成した PreparedStatement がコネクションごとに一つプーリングされます。 • **ステートメントプーリング機能と，リザルトセットの保持機能を併用する場合** Oracle 接続でリザルトセットの保持機能を使用する場合，Connection.prepareStatement()メソッドまたは prepareCall()メソッドの引数にリザルトセットの保持機能を指定してください。Connection.setHoldability()メソッドでは，リザルトセットの保持機能は指定できません。

（凡例）　－：該当なし

注※

コネクションの障害検知機能を使用する場合には，CallableStatementPoolSize を最大ステートメント数未満に設定してください。

コネクションの障害検知機能を使用すると，プーリングされている CallableStatement の数が最大ステートメント数に達したとき，コネクションの障害検知が実行されます。このとき，CallableStatementPoolSize=＜最大ステートメント数＞と設定していると，JDBC ドライバの一つのコネクションで利用できるリソース数の最大値を超えるため，例外が発生します。例外が発生すると障害が発生したと判断されるので，そのコネクションはコネクションプールから削除され，同時にステートメントプールも破棄されます。つまり，ステートメントプーリング機能を使用する意味がなくなってしまいます。

3.14.5　ライトトランザクション

ライトトランザクションとは，ローカルトランザクションに最適化された環境を提供する機能です。これによって，良好なローカルトランザクション処理性能を得られます。ライトトランザクションは，ローカルトランザクションだけを使用する場合に適用できます。ライトトランザクション機能を有効にした場合，呼び出し先が BMT のときだけ，トランザクション中の EJB リモート呼び出しができます。

ライトトランザクション機能は，デフォルトでは有効となっています。ライトトランザクション機能を有効にした状態で，グローバルトランザクションを使用するとエラーになります。このため，グローバルトランザクションを使用する場合には，ライトトランザクション機能を無効にする必要があります。

ライトトランザクション機能を有効にするための設定は，J2EE サーバのプロパティをカスタマイズして設定します。J2EE サーバの動作設定のカスタマイズについては，「3.4.12　実行環境での設定」を参照してください。

3.14.6　インプロセストランザクションサービス

アプリケーションサーバでは，トランザクションサービスを J2EE サーバのインプロセスで起動できます。インプロセスでトランザクションサービスを起動すると，トランザクション処理を J2EE サーバのプロセス内で実行するように最適化されるので，パフォーマンスの高いシステムを実現できます。

インプロセストランザクションサービスは，CORBA で規定された OTS1.3 の仕様をサポートしています。

3.14.7　DataSource オブジェクトのキャッシング

J2EE アプリケーションからデータベースにアクセスする場合，JNDI インタフェースを使用して javax.sql.DataSource 型のオブジェクト（以降，DataSource オブジェクト）の検索を要求します。J2EE サーバのデフォルトの動作では，該当する DataSource が登録されている場合に，DataSource オブジェクトのインスタンスを生成してアプリケーションに返します。

このとき，DataSource オブジェクトのキャッシングを使用すると，J2EE サーバは，登録されている DataSource オブジェクトのインスタンスをキャッシングして，検索の要求に対して同じインスタンスを返します。DataSource オブジェクトのキャッシングを使用すると，DataSource オブジェクトの検索時間が短くなります。

DataSource オブジェクトのキャッシングをするための設定は，J2EE サーバのプロパティをカスタマイズして設定します。J2EE サーバの動作設定のカスタマイズについては，「3.14.10　実行環境での設定」を参照してください。

(1)　前提条件

DataSource オブジェクトのキャッシング機能は，Enterprise Bean のビジネスメソッド，およびサーブレット/JSP のサービスメソッド内で DataSource をルックアップし，該当するインスタンスをメンバ変数などに保持しない場合に有効です。

ビジネスメソッドやサービスメソッド内でルックアップした DataSource オブジェクトをメンバ変数に保持し，ほかのメソッドでも利用する場合は有効ではありません。また，Enterprise Bean の ejbCreate メソッドやサーブレット/JSP の init メソッドなどの初期化メソッド内でルックアップした DataSource オブジェクトをメンバ変数に保持し，ビジネスメソッドやサービスメソッドで使用する場合も有効ではありません。

(2)　注意事項

DataSource オブジェクトのキャッシング機能を使用する場合に，次に示すようなリソースアダプタを使用すると，動作時に属性ファイルの<resource-ref>の定義が有効にならないことがあります。

- 同一 J2EE コンポーネントから同一リソースに対して複数の<resource-ref>タグを定義し，それぞれの<resource-ref>タグにある<res-sharing-scope>タグや<res-auth>タグに異なる値を指定した，リソースアダプタを使用する。

同一 J2EE コンポーネントで複数の<resource-ref>タグを定義する場合は，別のリソース（デプロイ単位が別になるリソースアダプタ）を使用するようにしてください。

3.14.8　DB Connector のコンテナ管理でのサインオンの最適化

DB Connector では，コンテナ管理のサインオンとコンポーネント管理のサインオンをサポートしています。

それぞれの方法の特徴を次に示します。

コンテナ管理でのサインオンの場合

コンテナ管理のサインオンを使用する場合，DB Connector で設定するユーザ名とパスワードを使用して，データベースにアクセスします。

コンポーネント管理でのサインオンの場合

コンポーネント管理でのサインオンを使用する場合，コネクションファクトリの getConnection メソッドに渡されたユーザ名とパスワードを使用してデータベースにアクセスします。

なお，DBConnector_DABJ_XA.rar を使用する場合（グローバルトランザクションを使用する場合），XAOpen 文字列に指定されたユーザ名とパスワードだけが有効となります。このため，getConnection メソッドでユーザ名とパスワードを指定して利用する，コンポーネント管理でのサインオンはできません。

コンテナ管理のサインオンをするときに，サインオンの最適化機能を使用すると，コンテナ管理でのサインオン動作が最適化され，データベースとのコネクション取得のパフォーマンスが向上します。

DB Connector のコンテナ管理でのサインオンの最適化をするための設定は，J2EE サーバのプロパティをカスタマイズして設定します。J2EE サーバの動作設定のカスタマイズについては，「3.14.10　実行環境での設定」を参照してください。

注意事項

- コンテナ管理でのサインオンの最適化は，コンポーネント管理のサインオンをしない場合に使用してください。コンテナ管理でのサインオンの最適化をすると，コンポーネント管理でのサインオンは利用できなくなります。
- クラスタコネクションプール機能を利用する場合，一つのクラスタコネクションプール内で，コンテナ管理でのサインオンとコンポーネント管理でのサインオンを混在して使用できません。

3.14.9　cosminexus.xml での定義

パフォーマンスチューニングのための機能のうち，コネクションシェアリングの定義は，cosminexus.xml の<rar>タグ内に指定します。cosminexus.xml でのパフォーマンスチューニングのための機能の定義について次の表に示します。

表 3-50　cosminexus.xml でのパフォーマンスチューニングのための機能の定義

指定するタグ	設定内容
<resource-external-property>－<res-sharing-scope>タグ	取得したコネクションを共有するかどうかを設定します。

cosminexus.xml については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「2.　アプリケーション属性ファイル（cosminexus.xml)」を参照してください。

3.14.10　実行環境での設定

パフォーマンスチューニングのための機能を使用する場合，J2EE サーバの設定，リソースアダプタの設定，および J2EE アプリケーションの設定が必要です。

(1)　J2EE サーバの設定

J2EE サーバの設定は，簡易構築定義ファイルで実施します。パフォーマンスチューニングの定義は，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に指定します。

簡易構築定義ファイルでのパフォーマンスチューニングのための機能の定義について，次の表に示します。

表 3-51　簡易構築定義ファイルでのパフォーマンスチューニングのための機能の定義

項目	指定するパラメタ	設定内容
アプリケーションサーバが管理するトランザクションの外でコネクションシェアリングの有効化	ejbserver.connectionpool.sharingOutsideTransactionScope.enabled	アプリケーションサーバが管理するトランザクションの外で複数回コネクションの取得を行ったときの，コネクションシェアリングの動作を指定します。
コネクションアソシエーション	ejbserver.connectionpool.association.enabled	コネクションアソシエーション機能を使用するかどうかを指定します。
	ejbserver.connectionpool.association.enabledDespiteUnshareableSetting	システムプロパティには，リソースの非共有

項目	指定するパラメタ	設定内容
コネクションアソシエーション	ejbserver.connectionpool.association.enabledDespiteUnshareableSetting	を設定している場合※も，コネクションアソシエーション機能を使用するかどうかを指定します。
ステートメントプーリング機能のステートメントの初期化	ejbserver.connector.statementpool.clear.backcompat	ステートメントプーリング機能でステートメントを再利用する際に初期化するステートメントの内容を指定します。
DataSource オブジェクトのキャッシング	ejbserver.jndi.cache.reference	DataSource オブジェクトのキャッシングを有効にするかどうかを指定します。
DB Connector のコンテナ管理でのサインオンの最適化	ejbserver.connectionpool.applicationAuthentication.disabled	コンテナ管理のサインオンの最適化機能を有効にするかどうかを指定します。
障害検知のタイムアウト	ejbserver.connectionpool.validation.timeout	コネクション障害検知機能のタイムアウト時間およびコネクション数調節機能によるコネクション削除処理のタイムアウト時間を指定します。

注※　WAR 属性ファイル，Entity Bean 属性ファイル，Session Bean 属性ファイル，または Message-driven Bean 属性ファイルの<res-sharing-scope>タグで，Unshareable（リソースを共有しない）を指定している場合のことです。

ライトトランザクションの設定については，「3.4.12　実行環境での設定」を参照してください。

なお，インプロセストランザクションサービスは，デフォルトの設定で使用される機能です。設定は不要です。

簡易構築定義ファイルおよびパラメタについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.6　簡易構築定義ファイル」を参照してください。

(2)　リソースアダプタの設定

実行環境でのリソースアダプタの設定は，サーバ管理コマンドおよび属性ファイルを使用します。パフォーマンスチューニングのための機能の定義には，Connector 属性ファイルを使用します。

Connector 属性ファイルでのパフォーマンスチューニングのための機能の定義について次の表に示します。

表 3-52　Connector 属性ファイルでのパフォーマンスチューニングのための機能の定義

分類	項目	指定するタグ	設定内容
コネクションプーリング	コネクションの最小値と最大値	<property>タグの MinPoolSize と MaxPoolSize	コネクションプールにプールするコネクションの最小値と最大値を指定します。
	コネクションプールのウォーミングアップ	<property>タグの Warmup	コネクションプールのウォーミングアップ機能を使用する指定をします。
	コネクション管理スレッド	<property>タグの NetworkFailureTimeout	コネクション管理スレッドを使用する指定をします。 コネクション管理スレッドを使用する場合，コネクションの障害検知機能，およびコネクション数調節機能のタイムアウトを使用する設定になります。07-60 以前のバージョンでは 5 秒で固定です。 08-00 以降のバージョンでは任意の時間を指定できます（デフォルト値は 5 秒）。
	コネクション数調節機能	<property>タグの ConnectionPoolAdjustmentInterval	コネクション数調節機能が動作する間隔を指定します。 なお，コネクション数調節機能にタイムアウトを設定する場合には，NetworkFailureTimeout でコネクション管理スレッドの使用を有効にします。※1
ステートメントプーリング	PreparedStatement のプールサイズ※2	<config-property>タグの PreparedStatementPoolSize	PreparedStatement のプールサイズを指定します。
	CallableStatement のプールサイズ※2	<config-property>タグの CallableStatementPoolSize	CallableStatement のプールサイズを指定します。

注※1　コネクションの障害検知機能のタイムアウトと同じキーです。このため，コネクションの障害検知機能でタイムアウトを使用する場合は，コネクション数調節機能でもタイムアウトを使用する設定となります。

注※2　XDM/RD E2 11-01 以前のバージョンの場合，ステートメントプーリング機能を利用できないため，これらのプロパティには 0 を指定してください。

Connector 属性ファイルについては，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1　Connector 属性ファイル」を参照してください。

(3)　J2EE アプリケーションの設定

実行環境での J2EE アプリケーションの設定は，サーバ管理コマンドおよび属性ファイルで実施します。パフォーマンスチューニングのための機能の定義には，WAR 属性ファイル，Session Bean 属性ファイル，Entity Bean 属性ファイルまたは Message-driven Bean 属性ファイルを使用します。

これらの属性ファイルで指定するタグは，cosminexus.xml または DD と対応しています。なお，cosminexus.xml での定義については，「3.14.9　cosminexus.xml での定義」を参照してください。

3.15　フォールトトレランスのための機能

この節では，フォールトトレランスのための機能について説明します。

この節の構成を次の表に示します。

表 3-53　この節の構成（フォールトトレランスのための機能）

分類	タイトル	参照先
解説	コネクションの障害検知	3.15.1
	コネクション枯渇時のコネクション取得待ち	3.15.2
	コネクションの取得リトライ	3.15.3
	コネクションプールの情報表示	3.15.4
	コネクションプールのクリア	3.15.5
	コネクションの自動クローズ	3.15.6
	コネクションスイーパ	3.15.7
	トランザクションタイムアウトとステートメントキャンセル	3.15.8
	トランザクションリカバリ	3.15.9
	障害調査用 SQL の出力	3.15.10
	オブジェクトの自動クローズ	3.15.11
実装	cosminexus.xml での定義	3.15.12
設定	実行環境での設定	3.15.13

注　「運用」について，この機能固有の説明はありません。

3.15.1　コネクションの障害検知

コネクションプーリングを使用している場合，リソースダウン・ネットワーク障害などが発生すると，ユーザプログラムのコネクション取得要求に対し，障害が発生したコネクションを返すおそれがあります。コネクションの障害検知機能を使用すると，プーリングされているコネクションに障害が発生していないかをチェックし，障害が発生したコネクションをなるべく返さないようにできます。

コネクション障害検知機能は，コネクション取得リトライ機能と併用できます。この場合，ユーザプログラムのコネクション取得要求時にコネクション障害を検知すると，新しいコネクション取得のリトライを実施し，障害が復旧した時点でコネクションをユーザプログラムに返すことができます。

なお，新しいコネクションを作成して返すことは，コネクションの障害検知のタイミングを「コネクション取得要求時」に設定している場合だけ有効になります。

コネクションの障害検知のタイミングについては，「(2)　障害検知を実施するタイミング」を参照してください。

(1)　前提条件

コネクションの障害検知機能の利用可否を次の表に示します。

表 3-54　コネクションの障害検知機能の利用可否

リソース	接続方法	利用可否
データベース	DB Connector	○
データベース上のキュー	DB Connector for Reliable Messaging と Reliable Messaging	○
OpenTP1	TP1 Connector	×※1
	TP1/Message Queue - Access	×※1
SMTP サーバ	メールコンフィグレーション	×
JavaBeans リソース	－	×
そのほかのリソース	Connector 1.0 仕様または Connector 1.5 仕様に準拠したリソースアダプタ	△※2

（凡例）○：使用できる　×：使用できない　△：条件によって異なる　－：該当なし

注※1　アプリケーションサーバが提供するコネクションの障害検知機能は使用できませんが，TP1 Connector または TP1/Message Queue - Access では，コネクションの障害検知機能と同様の機能が提供されています。

注※2　Connector 1.5 仕様に準拠したリソースアダプタを使用している場合に，リソースアダプタが ValidatingManagedConnectionFactory インタフェースの getInvalidConnections メソッドを実装しているときに使用できます。

(2)　障害検知を実施するタイミング

コネクションの障害を検知するタイミングは，表に示したタイミングのどちらかを選択できます。

表 3-55　コネクションの障害検知のタイミング

障害検知のタイミング	内容
コネクション取得要求時	コネクション取得時に，毎回，障害検知を実施します。この場合，コネクション取得要求のレスポンス性能が劣化しますが，障害が発生したコネクションを取得する確率は低くなります。コネクション取得要求の頻度が少ない場合や，障害が発生したコネクションの取得がほとんど許容されない場合に有用です。
一定間隔	一定時間の間隔で，障害検知を実施します。障害検知の間隔をある程度長くすることで，障害検知処理による性能劣化を少なくできます。ただし，障害が発生したコネクションを取得する確率は高くなります。コネクション取得要求の頻度が多い場合や，障害が発生したコネクションの取得がある程度許容される場合に有用です。

なお，デフォルトでは，コネクション取得要求時に障害検知を実施する設定となっています。コネクションの障害検知を実施しない設定もできます。

コネクション障害検知の有効・無効の切り替えや，検知するタイミングの設定は，リソースアダプタのプロパティとして設定します。リソースアダプタの設定については，「3.15.13　実行環境での設定」を参照してください。

(3)　障害検知のタイムアウト

コネクション障害検知に対する応答時間にタイムアウトを設定できます。

サーバ障害やネットワーク障害が発生してリソースからの応答が返らない場合，コネクション障害検知の実行に対しても応答が返らなくなることがあります。タイムアウトを設定することで，リソースからの応答が

返らない場合も，コネクションのチェックを終了して，処理を継続できます。なお，障害検知のタイムアウト時間は簡易構築定義ファイルの J2EE サーバで指定するキーに，任意の時間を指定できます（デフォルト値は 5 秒）。

コネクション障害検知のタイムアウトは，コネクション管理スレッドを使用して動作します。コネクション管理スレッドは，コネクションプールのコネクション数の最大値に応じてリソースアダプタ開始時にシステム内で作成されます。コネクション管理スレッドの数と，コネクションプールのコネクション数の最大値の関係を次に示します。

コネクション管理スレッド数 ＝ コネクションプールのコネクション数の最大値×2

このため，コネクション障害検知にタイムアウトを設定すると，設定しない場合に比べて多くのメモリを消費します。必要なメモリ使用量を適切に見積もってください。

また，コネクションプールのコネクション数の最大値の設定が無制限の場合，メッセージが出力されて，コネクション障害検知のタイムアウトが無効になります。

なお，コネクション障害検知のタイムアウトを有効にする場合の注意事項の詳細については，「(5)　注意事項」を参照してください。

コネクション障害検知のタイムアウトの有効・無効は，リソースアダプタのプロパティをカスタマイズして設定します。リソースアダプタのプロパティの設定については，「3.15.13　実行環境での設定」を参照してください。

(4)　チェック時の動作

コネクション障害検知をコネクション取得要求時に実施する場合と，一定間隔で実施する場合の動作について説明します。

- **コネクション障害検知をコネクション取得要求時に実施する場合**

  コネクション障害検知をコネクション取得要求時に実施する場合の動作を次に示します。

  1. コネクションプールから未使用コネクションが取得されます。
  2. コネクションチェックメソッドによって，コネクションに障害が発生していないかどうかチェックされます。

     コネクション障害検知のタイムアウトが無効な場合，このチェックはコネクション取得要求の延長で実施されます。

     コネクション障害検知のタイムアウトが有効な場合，このチェックはコネクション管理スレッドで実施されます。サーバ障害やネットワーク障害などでチェックメソッドの応答がタイムアウト時間以内に返らないときは，コネクションに障害が発生していると判断されます。このとき，メッセージが出力されます。

     なお，リソースアダプタにコネクションチェックメソッドが実装されていない場合，コネクションの状態はチェックされません。
  3. コネクションに障害が発生している場合，チェックに使用したコネクションが破棄されて，新たに作成されたコネクションがユーザプログラムに返されます。

     コネクションに障害が発生していない場合，チェックに使用したコネクションがユーザプログラムに返されます。

- **コネクション障害検知を一定間隔で実施する場合**

  コネクション障害検知を一定間隔で実施する場合の動作を次に示します。

  1. コネクションプールから未使用コネクションが一つ取得されて，そのコネクションが有効であるかどうかがチェックされます。

ただし，未使用コネクションがプーリングされていない場合は，チェックされません。

2. コネクション障害検知のタイムアウトが無効な場合，コネクションチェックメソッドによって，コネクションに障害が発生していないかどうかのチェックが一定間隔で実施されます。

   コネクション障害検知のタイムアウトが有効な場合，コネクション管理スレッドで，チェックが一定間隔で実施されます。サーバ障害やネットワーク障害などでチェックメソッドの応答がタイムアウト時間以内に返らないときは，コネクションに障害が発生していると判断されます。このとき，メッセージが出力されます。

   なお，リソースアダプタにコネクションチェックメソッドが実装されていない場合，コネクションの状態はチェックされません。

3. コネクションに障害が発生している場合に，コネクション障害検知のタイムアウトが無効のときは，コネクションチェックに使用されたコネクションが破棄されて，コネクションプール内の使用中コネクションが再利用不可にマークされます。また，未使用コネクションが破棄されます。

   コネクション障害検知のタイムアウトが有効のときは，コネクションチェックに使用したコネクションが破棄されて，コネクション管理スレッドによってコネクションプール内の使用中コネクションが再利用不可にマークされます。また，未使用コネクションが，コネクションプールから取り除かれて，破棄されます。

   コネクションに障害が発生していない場合は，コネクションチェックに使用されたコネクションがコネクションプールに返却されます。

(5)　注意事項

障害検知機能についての注意事項を説明します。

(a)　パラレルサーバ構成のリソースを使用する場合の注意事項

コネクション障害検知を一定間隔で実施する場合，コネクションプール内のコネクションをサンプリングチェックするため，一部のサーバに障害が発生しても検知できないことがあります。

(b)　コネクション障害検知のタイムアウトを有効にした場合の注意事項

コネクション障害検知のタイムアウトを有効にした場合の注意事項を次に示します。

- コネクション障害検知にタイムアウトを設定した場合，システム内で，コネクションプールのコネクション数に応じた障害検知用のスレッドが生成されます。このため，コネクション障害検知にタイムアウトを設定すると，設定しない場合に比べて多くのメモリを消費するので注意が必要です。

  障害検知用のスレッドは，コネクションプールの最大コネクション数の 2 倍の数だけ作成されます。必要なメモリ使用量を適切に見積もってください。

- サーバ障害やネットワーク障害などが繰り返し発生する場合にシステムの運用を続けると，コネクションチェックメソッドを実行中のコネクション管理スレッドが増え続け，システム側が用意したすべてのコネクション管理スレッドを使い切るおそれがあります。使用できるコネクション管理スレッドがない場合，メッセージが出力されて，コネクション取得要求がエラーとなります。

- コネクション障害検知を実施する場合，コネクションプールから取り除いた未使用コネクションは，コネクションプール内のコネクション数としてカウントされません。そのため，コネクションプール内のコネクションとコネクションプールから取り除いた未使用コネクションの総数が，コネクションプールのコネクション数の最大値を一時的に超える場合があります。

(c)　障害検知機能を利用できないリソースを使用する場合の注意事項

障害検知機能を利用できないリソースに対しては，Connector 属性ファイルで次のパラメタを指定して，コネクション管理スレッドを使用しない設定にしてください。

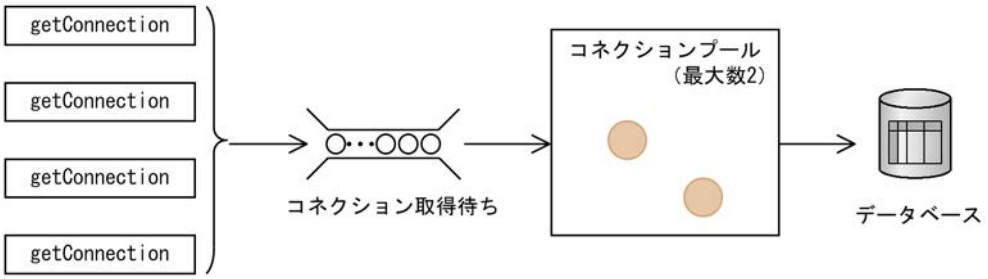
- コネクションの障害検知のタイムアウト
  <property>タグの ValidationType=0

3.15.2　コネクション枯渇時のコネクション取得待ち

コネクションがコネクションプールに指定した最大コネクション数までプールされていて，利用できるコネクションがプール中にない状態を，**コネクション枯渇**といいます。

コネクション枯渇状態のときには，コネクション取得要求を待ち状態にできます。待ち状態になっているコネクション取得要求は，コネクションが解放されるとすぐにコネクションを取得できます。これによって，コネクション枯渇時に，効率良くコネクションを取得できます。コネクション枯渇時の取得待ちについて，次の図で説明します。

図 3-36　コネクション枯渇時のコネクション取得待ち



この図では，コネクションプールの最大数は 2 となっています。このため，四つの getConnection からコネクション取得要求があっても，二つまでしか処理できません。最初の二つのコネクションの取得要求については，コネクションプールにコネクションがあるため，コネクションの取得ができます。残りの二つについては，コネクション枯渇の状態になるため，コネクション取得待ちのキューに入り，コネクションが解放されるのを待ちます。

なお，コネクション取得待ちは，コネクションプールを利用している場合に設定できます。

コネクション取得待ちをする場合には，リソースアダプタのプロパティとして，次の二つの内容を設定する必要があります。

- コネクション枯渇時にコネクションの取得待ちをするかどうか
- コネクション取得待ちの最大時間

リソースアダプタの設定については，「3.15.13　実行環境での設定」を参照してください。

(1)　コネクション枯渇時の動作

コネクション取得待ちを設定している場合，コネクション枯渇時のコネクション取得要求は待ち状態になります。コネクション取得要求の待ち時間が最大待ち時間を越えた場合，ユーザプログラムに例外をスローします。

また，コネクション取得要求を再開するタイミングは，次のどちらかになります。

- コネクションが解放されて，未使用コネクションができたとき
- コネクションが破棄されて，コネクション数が最大数未満になったとき

なお，コネクション取得要求の再開後にエラーが発生した場合，コネクション取得のリトライ機能が有効になっているときは，リトライ処理が実施されます。

3.15.3　コネクションの取得リトライ

コネクション取得リトライは，使用できるコネクションがコネクションプールにない場合や，物理コネクションの確立に失敗した場合に，自動的にコネクションの取得をリトライする機能です。コネクション取得リトライ機能を使用することで，コネクション取得に失敗した場合に，ユーザプログラムでリトライをする必要がなくなります。

次の条件のどちらかに当てはまる場合に，コネクション取得をリトライできます。

- 次のすべての条件を満たし，使用できるコネクション（未使用状態のコネクション）がコネクションプールにない場合
  - コネクション枯渇時のコネクション取得待ち行列が無効。
  - コネクションプールのコネクション総数がプーリングするコネクションの最大数に達している。
- 次の条件のどれかで物理コネクションの確立に失敗した場合
  - コネクションプールのコネクション総数がプーリングするコネクションの最大数に達していて，認証情報が一致する未使用状態のコネクションがない。
  - コネクションプールのコネクション総数がプーリングするコネクションの最大数に達していない。
  - コネクションプーリングが無効。

なお，リトライしてもコネクションが取得できない場合は，アプリケーションプログラムに例外が通知され，コネクションの取得は失敗します。

コネクションプールが枯渇したときの動作は，「3.15.2　コネクション枯渇時のコネクション取得待ち」に従います。

コネクション取得リトライを実施する場合には，リソースアダプタのプロパティとして，次の二つの内容を設定する必要があります。

- リトライの回数
  リトライする回数を設定します。
- リトライまでの間隔
  次のリトライまでの間隔を秒単位で設定します。

なお，リトライの回数，およびリトライまでの間隔を大きくすると，コネクション取得処理が重なった場合に，待ちが発生するおそれがあります。

リソースアダプタの設定については，「3.15.13　実行環境での設定」を参照してください。

(1)　前提条件

コネクション取得のリトライ機能を使用するための前提条件を次の表に示します。

表 3-56　コネクション取得のリトライ機能の利用

リソース	接続方法	利用可否
データベース	DB Connector	○
データベース上のキュー	DB Connector for Reliable Messaging と Reliable Messaging	○
OpenTP1	TP1 Connector	○

リソース	接続方法	利用可否
OpenTP1	TP1/Message Queue - Access	○
SMTP サーバ	メールコンフィグレーション	×
JavaBeans リソース	−	×
そのほかのリソース	Connector 1.0 仕様または Connector 1.5 仕様に準拠したリソースアダプタ	○

（凡例）○：使用できる　×：使用できない　−：該当なし

3.15.4　コネクションプールの情報表示

コネクションプール内のコネクションの情報は，cjlistpool コマンドを使用して表示できます。cjlistpool コマンドは，クラスタコネクションプール機能を使用した場合のメンバリソースアダプタに対しても実行できます。

コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjlistpool（コネクションプールの一覧表示）」を参照してください。また，クラスタコネクションプール機能については，「3.17　クラスタコネクションプール機能」を参照してください。

メンバリソースアダプタのコネクション情報の表示例を次に示します。なお，コネクションプールが無効な場合は，コネクション情報は表示されません。

図 3-37　メンバリソースアダプタのコネクション情報の表示例

```
Resource Name = {0}  Connection Definition = {1}                    }コネクションプール情報
Minimum Size = {2},  Maximum Size = {3},  Current Size = {4}        }メンバコネクションプー
Root Resource Adapter Name = {5},  Status = {6}                     }ル情報※1
-------------------------------------------------------------------

Active Connection = {7}
------------------------------
Hashcode = XXXXXXXXXX  rootAP = 999.999.999.999/999/XXXXXXXXXX      }
  Create Time    = yyyy/MM/dd HH:mm:ss.SSS,  Lapsed Time(s) = 99999 } 使用中コネクション
  Get Time       = yyyy/MM/dd HH:mm:ss.SSS,  Lapsed Time(s) = 99999 }
  Connection ID = XXXXXXXXX:9999:9999:9999                          }

  Hashcode = {8}     rootAP = {9}
  Create Time    = {10},  Lapsed Time(s) = {11}
  Get Time       = {12},  Lapsed Time(s) = {13}
  Connection ID = {14}

      :
      :

Free Connection = {15}
------------------------------
Hashcode = XXXXXXXXXX                                               }
  Create Time    = yyyy/MM/dd HH:mm:ss.SSS,  Lapsed Time(s) = 99999 } 未使用コネクション
  Last Use Time = yyyy/MM/dd HH:mm:ss.SSS,  Lapsed Time(s) = 99999  }
  Connection ID = XXXXXXXXX:9999:9999:9999                          }

Hashcode = {8}
  Create Time    = {10},  Lapsed Time(s) = {11}
  Last Use Time = {16},  Lapsed Time(s) = {17}
  Connection ID = {14}

      :
      :
```

（凡例）

{0}：リソース名
{1}：コネクション定義識別子
{2}：コネクション数の最小値
{3}：コネクション数の最大値
{4}：コネクション数の現在値
{5}：ルートリソースアダプタ名※2
{6}：コネクションプールの状態※3
{7}：使用中のコネクション数
{8}：コネクションハッシュコード
{9}：ルートAP情報
{10}：コネクション生成時刻
{11}：コネクション生成経過時間（単位：秒）
{12}：コネクション取得時刻
{13}：コネクション使用経過時間（単位：秒）
{14}：コネクションID※4
{15}：未使用のコネクション数
{16}：コネクション返却時刻※4
{17}：コネクション未使用時間（単位：秒）

注※1
メンバコネクションプール情報は，メンバリソースアダプタの場合だけ出力されます。
注※2
ルートリソースアダプタが存在しない場合や開始していない場合は，「N/A」が出力されます。
注※3
cjlistrarコマンドと同じ状態が出力されます。なお，コネクションプールの状態が取得できない場合や，状態が不正な場合は，「invalid」が出力されます。
注※4
コネクションID，コネクション返却時刻，またはコネクション定義識別子がない場合は，「N/A」が出力されます。

3.15.5　コネクションプールのクリア

データベースサーバのダウンなどによって，コネクションが切断された場合，プールされているコネクションはコネクションプール内に残ってしまいます。これらのコネクションは，cjclearpool コマンドによって削除できます。

なお，コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjclearpool（コネクションプール内のコネクション削除）」を参照してください。

3.15.6　コネクションの自動クローズ

ユーザプログラムがオープンしたリソースのコネクションは，ユーザプログラムでクローズする必要があります。例外発生などの理由で，ユーザプログラムがコネクションをクローズすることができない場合，Web コンテナや EJB コンテナが自動的にコネクションをクローズする機能があります。

コネクション自動クローズ機能は，機能によって有効と無効を切り替えることができます。コネクションの自動クローズの方法と切り替えについて次の表に示します。

表 3-57　コネクション自動クローズの方法と機能の切り替え

自動クローズの方法	有効と無効の切り替え	備考
Web コンテナによるコネクション自動クローズ	○※1，※2	－
EJB コンテナによるコネクション自動クローズ	×※1	常に有効になります。

（凡例）○：切り替えできる　×：切り替えできない　－：該当しない

注※1　対象リソースは，リソースアダプタのコネクションとなります。

注※2　Web コンテナによるコネクションの自動クローズ機能は，デフォルトでは有効となっています。無効にする場合は，J2EE サーバのプロパティをカスタマイズして設定します。J2EE サーバの動作設定のカスタマイズについては，「3.15.13　実行環境での設定」を参照してください。

(1)　Web コンテナによるコネクション自動クローズ

サーブレットや JSP のサービスメソッド内で，取得・オープンされた JDBC コネクションのうち，メソッドの実行完了時にクローズが実行されていないコネクションに対して，J2EE コンテナ側で自動的にコネクションをクローズします。これによって，オープンされたまま蓄積しているコネクションを自動でクローズできます。

なお，コネクションの自動クローズ機能は，サーブレット内部の別のメソッド，または外部のクラスでコネクションを取得・利用した場合にも有効です。

Web コンテナによるコネクション自動クローズは，サーブレットや JSP の service メソッドが完了したときに実行されます。

(2)　EJB コンテナによるコネクション自動クローズ

例外発生などの理由で，EJB 内で取得したリソースのコネクションをユーザがクローズできない場合に，EJB コンテナが自動的にコネクションをクローズして解放する機能です。

EJB コンテナによるコネクションの自動クローズは，次のタイミングで実施されます。

- EJB が破棄されるとき
- Stateless Session Bean で，ビジネスメソッドがリターンするとき
- Stateful Session Bean がパッシベイトされるとき
- Singleton Session Bean で，ビジネスメソッドがリターンするとき

なお，@Asynchronous アノテーションを指定した Singleton Session Bean または Stateless Session Bean では，次のタイミングでコネクションの自動クローズが実行されます。

- Singleton Session Bean または Stateless Session Bean のインスタンスが破棄された時
- Singleton Session Bean または Stateless Session Bean 内の@Asynchronous アノテーションを指定したメソッドの実行が完了した時

(3)　コネクション自動クローズが実行されたことを確認する方法

コネクション自動クローズが実行されたことは，メッセージログまたは PRF トレースで確認できます。

テスト実行時や障害調査時など，性能を考慮しなくてよい場合は，メッセージログによって確認することをお勧めします。実運用時に確認したい場合は，PRF トレースによって確認することをお勧めします。

(a)　メッセージログで確認する方法

ログレベルを「Warning」に設定してください。

コネクション自動クローズが実行された場合は，J2EE サーバのメッセージログに KDJE31010-W が出力されます。

ログレベルの設定方法については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「3.3.6　J2EE サーバのログ取得の設定」を参照してください。

(b)　PRF トレースでの確認方法

JCA コンテナの機能レイヤの PRF トレース取得レベルを「詳細」に設定してください。

コネクション自動クローズが実行された場合は，PRF トレースファイルに次のイベント ID が出力されます。

- 0x8B84
- 0x8B85

PRF トレースの取得レベルの設定方法については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「3.3.6　J2EE サーバのログ取得の設定」を参照してください。また，機能レイヤごとの PRF トレース取得レベルの設定方法については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cprflevel（PRF トレース取得レベルの表示と変更)」またはマニュアル「アプリケーションサーバ リファレンス コマンド編」の「cprfstart（PRF デーモンの開始)」を参照してください。

(4)　注意事項

- Servlet/JSP でユーザが生成したスレッド（ユーザスレッド）で取得したコネクションは，自動クローズされません。
- 自動クローズの際に，未決着の JDBC トランザクションがあった場合，JDBC トランザクションはロールバックされます。また，ユーザプログラムでコネクションをクローズした際に，未決着の JDBC トランザクションがあった場合，暗黙的にコミットされます。

3.15.7　コネクションスイーパ

コネクションスイーパは，一定間隔でコネクションプールを監視し，一定時間使用していない未使用コネクションを破棄する機能です。該当コネクションを最後に利用した時点からの経過時間が，指定された時間以上の場合に，コネクションを破棄します。この機能は，アプリケーションサーバとデータベースサーバ間に

ファイアウォールなどのゲートウェイが存在し，コネクションが一定時間で切断されるケースなどに使用できます。

コネクションスイーパ機能は，デフォルトでは無効になっています。なお，コネクションスイーパ機能の設定は，リソースアダプタのプロパティとして設定します。リソースアダプタの設定については，「3.15.13 実行環境での設定」を参照してください。

なお，コネクションスイーパを使用する場合，Connector 属性ファイルの<property>タグで ConnectionTimeout と SweeperInterval の合計値がコネクション切断時間未満になるように設定してください。また，ConnectionTimeout を長めに設定することで，コネクションを保持する期間を長くできます。

3.15.8　トランザクションタイムアウトとステートメントキャンセル

トランザクションタイムアウト機能とは，トランザクションの開始時点からの経過時間を監視し，指定された時間が経過した時点※でトランザクションをロールバックする機能です。監視期間は，トランザクションの開始完了時点から，トランザクションの決着開始時点までです。トランザクションタイムアウトが発生したあと，該当するトランザクションでの JTA，および JDBC インタフェースへのアクセスは例外が通知されます。

トランザクションタイムアウト機能は，アプリケーションサーバが管理するトランザクションの場合に有効です。

なお，BMT，サーブレット，または JSP の場合，トランザクションタイムアウトが発生したあと，該当するトランザクションに対して UserTransaction.commit/rollback を発行すると，JTA，および JDBC インタフェースにアクセスできるようになります。

注※　タイムアウトしたかどうかを 1 秒ごとに確認するため，トランザクションタイムアウトの設定時刻に対して最大 1 秒の誤差が発生します。

(1)　トランザクションタイムアウトの設定

トランザクションタイムアウトの設定は，J2EE サーバ，UserTransaction インタフェース，または EJB の CMT 属性で設定できます。

(a)　J2EE サーバでの設定

J2EE サーバでの設定は，J2EE サーバのプロパティをカスタマイズして設定します。J2EE サーバの動作設定のカスタマイズについては，「3.15.13　実行環境での設定」を参照してください。

(b)　UserTransaction インタフェースでの設定

UserTransaction インタフェースの setTransactionTimeout メソッドを使用して，デフォルト値から変更できます。

(c)　EJB の CMT 属性での設定

EJB の CMT 属性で設定するには，サーバ管理コマンドを使用して設定します。設定単位は，Bean 単位，インタフェース単位，およびメソッド単位の三つから選択できます。設定が有効になるメソッドは，次のどちらかです。

- トランザクション属性が「Required」に設定され，EJB コンテナがトランザクションを開始するメソッド

- トランザクション属性が「RequiresNew」に設定されたメソッド

クライアントから伝播するトランザクションで動作する Supports 属性および Mandatory 属性の場合，トランザクションを使用しない NotSupported 属性および Never 属性の場合は，タイムアウトの設定は無効になります。なお，トランザクション属性については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「2.7.3　CMT」を参照してください。

なお，設定の優先順位は，次のとおりです。

1. メソッド単位での設定
2. インタフェース単位での設定
3. Bean 単位での設定
4. J2EE サーバでの設定

Bean 単位のトランザクションタイムアウトの設定は，J2EE アプリケーションに含まれる Session Bean，Entity Bean または Message-driven Bean の属性（プロパティ）として設定します。J2EE アプリケーションの設定については，「3.15.13　実行環境での設定」を参照してください。

(2)　トランザクションタイムアウト発生時のステートメントキャンセル

実行中の SQL 処理が返ってこない状態でトランザクションタイムアウトが発生すると，ステートメントがキャンセルされます。

ステートメントのキャンセルを有効にする場合，DB Connector の CancelStatement プロパティに true を設定する必要があります。リソースアダプタの設定については，「3.15.13　実行環境での設定」を参照してください。

3.15.9　トランザクションリカバリ

J2EE サーバやリソースマネジャの障害で，プリペア状態またはヒューリスティック完了状態となった 2 フェーズトランザクションを決着させる機能です。グローバルトランザクションを使用する場合に有効になります。

J2EE サーバを開始すると，インポートされているリソースに対して，無条件にトランザクションの全面回復処理を実行します。また，トランザクション実行中にリソースマネジャがダウンした場合にも，部分回復処理をします。

なお，トランザクションの状態は，インプロセストランザクションサービスを使用している場合，サーバ管理コマンドの cjlisttrn コマンドで表示できます。また，停止中の J2EE サーバのステータスファイルに残っている，未決着トランザクション情報は cjlisttrnfile コマンドで表示できます。

ライトトランザクションが有効なときには，トランザクションリカバリ機能は使用できません。

! 注意事項

J2EE サーバ再起動によるトランザクションリカバリをしない場合，トランザクションの回復は各リソースの回復手順に従い，ユーザ責任で手動回復してください。

(1)　J2EE サーバ終了時の未決着トランザクションの確認とタイムアウトの設定

J2EE サーバを正常停止する時, J2EE サーバは未決着のトランザクションがないことを確認してから停止します。未決着のトランザクションがあるときは，それらが完了するまで無限に待ちます。また，そのトランザクションが決着するまでリソースを削除できません。

これに対して，システム開発時など，トランザクションを早急に解決する必要がない場合は，未決着トランザクションの確認時間にタイムアウトを設定できます。タイムアウトが発生した場合は，未決着のトランザクションの確認処理が完了していなくても，J2EE サーバの停止処理がされます。ただし，タイムアウトは，J2EE アプリケーション開発時などに設定してください。J2EE アプリケーション運用時には，トランザクションの信頼性を保証するために，タイムアウトを設定しないことをお勧めします。

タイムアウトの設定は，J2EE サーバのプロパティをカスタマイズして設定します。J2EE サーバの動作設定のカスタマイズについては，「3.15.13　実行環境での設定」を参照してください。

(2)　注意事項

トランザクションリカバリについての注意事項を説明します。

(a)　J2EE サーバ起動時にリソースアダプタの開始に失敗した場合の注意

J2EE サーバ起動時に XATransaction を利用するリソースアダプタの開始に失敗した場合，J2EE サーバはトランザクションリカバリを実行しないで，メッセージ KDJE48605-E を出力して強制停止します。この場合は，リソースアダプタの開始処理が失敗する原因を取り除いてから，J2EE サーバを再起動してください。これによって，プリペア状態またはヒューリスティック状態のトランザクションは決着されます。

(b)　J2EE サーバを再起動する時の注意

- J2EE サーバが強制終了または異常終了したあと，リソースを削除するとリカバリできなくなります。このため，再起動時にリソースの構成を変更しないでください。
- 強制終了または異常終了前の受信ポートと同じポートで再起動する必要があります。このため，簡易構築定義ファイル内の<configuration>タグ内のパラメタ「ejbserver.distributedtx.recovery.port」の値を変更しないでください。なお，Management Server を利用しないでシステムを構築する場合は，usrconf.properties の ejbserver.distributedtx.recovery.port キーを変更しないでください。

(c)　トランザクションリカバリを実行するための権限

- リカバリはリソース（XADataSource など）に設定したデフォルトユーザで実行します。リソースマネジャによっては，未決着トランザクションの走査に特別な権限や設定が必要になります。また，複数のユーザでサインオンする場合，デフォルトユーザにそのほかのユーザのトランザクションを決着できるリソースマネジャでの適正な権限を付与する必要があります。なお，詳細については，各リソースのマニュアルを参照してください。
- Oracle でリカバリをする場合で，JDBC ドライバに Oracle JDBC Thin Driver を使用するときは，ユーザに次の権限が必要です。
  - SYS.DBA_PENDING_TRANSACTIONS への SELECT 権限
  - FORCE ANY TRANSACTION 権限
  - SYS.DBMS_SYSTEM を EXECUTE する権限

  Oracle を使用する場合の設定については，マニュアル「アプリケーションサーバ システム構築・運用ガイド」の「4.1.7　データベース接続環境を設定する（Oracle の設定）」を参照してください。

(d)　使用するコネクション数

トランザクションリカバリをする場合に使用するコネクション数に注意してください。

J2EE サーバでは，トランザクションサポートレベルが XATransaction の一つのリソースアダプタに対して，次のコネクションを確立します。

- コネクションプールにプーリングされるコネクション

- リカバリ用のコネクション（一つ）

同一のリソースマネジャで必要となる最大コネクション数は，次の式で示す値になります。リソースマネジャにコネクション数の上限がある場合は，注意してください。

同一のリソースマネジャで必要となる最大コネクション数＝ IR(1) ＋ … ＋ IR(N) ＋ N

IR(i)

i 番目のリソースアダプタのプール設定値の最大コネクション数。

1≦i≦N です。

N

- Connector 1.0 仕様に準拠したリソースアダプタの場合は，同一のリソースマネジャに接続するリソースアダプタ数。
- Connector 1.5 仕様に準拠したリソースアダプタの場合は，同一のリソースマネジャに接続するリソースアダプタ内のコネクション定義数の総数。

対象になるのは，開始状態で，かつトランザクションサポートレベルが XATransaction のリソースアダプタです。

(3)　トランザクション情報の確認手順

ここでは，稼働中および停止中の J2EE サーバのトランザクションの情報を確認する方法について説明します。稼働中の J2EE サーバでのトランザクションの状態や，停止中の J2EE サーバでの未決着のトランザクションの有無などの情報を確認できます。

(a)　稼働中のトランザクションの確認手順

J2EE サーバで稼働中のトランザクションの情報を確認できます。トランザクションの状態，グローバルトランザクション ID，経過時間，ブランチの種類などの情報を確認できます。

稼働中のトランザクションの確認には，cjlisttrn コマンドを使用します。実行形式と実行例を次に示します。

実行形式

```
cjlisttrn [<サーバ名称>] -bqual
```

実行例

```
cjlisttrn MyServer -bqual
```

また，未決着のトランザクションの状態も確認できます。未決着のトランザクションについての情報を確認するときは，引数に「-pending」を指定します。未決着のトランザクションについての情報を確認する場合の実行形式と実行例を次に示します。

実行形式

```
cjlisttrn [<サーバ名称>] -pending -bqual
```

実行例

```
cjlisttrn MyServer -pending -bqual
```

cjlisttrn コマンド，および取得できる情報の詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjlisttrn（稼働中の J2EE サーバのトランザクション情報の表示）」を参照してください。

(b)　停止中のトランザクションの確認手順

停止中の J2EE サーバのトランザクションの情報を確認できます。トランザクションの状態，グローバルトランザクション ID，経過時間，ブランチの種類等の情報を確認できます。また，未決着のトランザクションが残っているかどうかも確認できます。

停止中のトランザクションの確認には，cjlisttrnfile コマンドを使用します。実行形式と実行例を次に示します。

実行形式

```
cjlisttrnfile［<サーバ名称>］-bqual
```

実行例

```
cjlisttrnfile MyServer -bqual
```

J2EE サーバが停止中の状態で未決着のトランザクションが存在する場合は，必要に応じて次の処理を実行して，トランザクションを決着させてください。

- J2EE サーバを再起動する（cjstartsv コマンド）
- J2EE サーバをリカバリモードで起動する（cjstartrecover コマンド）

cjlisttrnfile コマンド，および取得できる情報の詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjlisttrnfile（停止中の J2EE サーバのトランザクション情報の表示）」を参照してください。

3.15.10　障害調査用 SQL の出力

デッドロックやスローダウンなどの障害が発生した場合，発行した SQL が障害の要因となった可能性があります。そこで，発行した SQL をログに出力することによって，障害要因の解析が容易になります。ログに出力される SQL の情報を**障害調査用 SQL** と呼びます。

(1)　出力されるタイミング

障害調査用 SQL は，次のタイミングで出力されます。

- トランザクションタイムアウト発生時
- J2EE アプリケーションの強制停止実行時
- メソッドキャンセルコマンド実行時
- メソッドタイムアウト発生後のメソッドキャンセル実行時

(2)　出力先

障害調査用 SQL は，リソースアダプタの稼働ログ，および性能解析トレースに出力されます。

リソースアダプタの稼働ログでは，KDJE50080-W のメッセージに出力されます。詳細については，マニュアル「アプリケーションサーバ メッセージ(構築／運用／開発用)」の「KDJE50080-W」を参照してください。

性能解析トレースでは，0x8C41 のイベント ID に出力されます。詳細については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「8.　性能解析トレースのトレース取得ポイントと PRF トレース取得レベル」を参照してください。

(3)　出力内容

出力対象のコネクションで SQL を発行している場合，物理コネクションが SQL を保持しています。この物理コネクションが保持している SQL が，障害調査用 SQL として出力されます。

SQL を保持する物理コネクション

障害調査用 SQL が出力されるタイミングごとに，SQL を保持している物理コネクションを示します。

- **トランザクションタイムアウト発生時**
  トランザクションに参加しているコネクションに対応する物理コネクション。
- **J2EE アプリケーションを強制停止した時，メソッドキャンセルコマンド実行時，またはメソッドタイムアウト発生後のメソッドキャンセル実行時**
  トランザクション処理中の場合，トランザクションに参加しているコネクションに対応する物理コネクション。
  トランザクションを使用していない場合，アプリケーション強制停止またはメソッドキャンセルを実行するインスタンスで使用中のコネクションに対応する物理コネクション。なお，クローズしたコネクションは，障害調査用 SQL 出力の対象外になります。

SQL を保持する API

ユーザアプリケーションで次の API が呼び出されたとき，引数で渡された SQL を物理コネクションに保持します。保持する SQL は物理コネクションごとに一つです。API が呼び出されるたびに，最新の SQL が上書きされます。SQL を保持する API を次の表に示します。

表 3-58　SQL を保持する API の一覧

インタフェース	メソッド
java.sql.Connection	prepareCall(String sql)
	prepareCall(String sql, int resultSetType, int resultSetConcurrency)
	prepareCall(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)
	prepareStatement(String sql)
	prepareStatement(String sql, int autoGeneratedKeys)
	prepareStatement(String sql, int[] columnIndexes)
	prepareStatement(String sql, int resultSetType, int resultSetConcurrency)
	prepareStatement(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)
	prepareStatement(String sql, String[] columnNames)
java.sql.Statement	addBatch(String sql)
	execute(String sql)
	execute(String sql, int autoGeneratedKeys)
	execute(String sql, int[] columnIndexes)
	execute(String sql, String[] columnNames)

インタフェース	メソッド
java.sql.Statement	executeBatch()※1
	executeQuery(String sql)
	executeUpdate(String sql)
	executeUpdate(String sql, int autoGeneratedKeys)
	executeUpdate(String sql, int[] columnIndexes)
	executeUpdate(String sql, String[] columnNames)
java.sql.PreparedStatement	java.sql.Statement から継承した表示対象メソッド
	addBatch()※2
	execute()※2
	executeQuery()※2
	executeUpdate()※2
java.sql.CallableStatement	java.sql.PreparedStatement から継承した表示対象メソッド

注※1　executeBatch()メソッドを実行した場合は，最後に addBatch(String sql)メソッド，addBatch()メソッドで加えられた SQL を物理コネクションに保持します。

注※2　addBatch()メソッド，execute()メソッド，executeQuery()メソッド，executeUpdate()メソッドを実行した場合，java.sql.Connection の prepareStatement メソッド，prepareCall メソッドの引数で与えられた SQL を物理コネクションに保持します。ただし，SQL の IN パラメタプレースホルダー（"？"）は，置換しないで"？"のまま出力します。

(4)　注意事項

障害調査用 SQL の出力機能についての注意事項を説明します。

- DB Connector を使用している場合，この機能は常に有効になります。DB Connector 以外を使用している場合は無効になります。
- リソースアダプタの稼働ログの障害調査用 SQL ログに出力されるメッセージが 4 キロバイトを超えた場合，4 キロバイトまでのメッセージだけが出力されます。
- コネクションシェアリング，コネクションアソシエーションによって共有されたコネクションは，物理コネクションが一つのため，障害調査用 SQL ログは一つだけ出力されます。

3.15.11　オブジェクトの自動クローズ

ユーザプログラムでオープンした Statement オブジェクトなどは，ユーザプログラムでクローズする必要があります。ただし，クローズできなかった場合，DB Connector は，ユーザハンドル（java.sql.Connection）から生成された，次のオブジェクトを自動的にクローズできます。

- Statement オブジェクト
- PreparedStatement オブジェクト
- CallableStatement オブジェクト
- 各種ステートメントや DatabaseMetaData から生成された ResultSet オブジェクト

3.15.12　cosminexus.xml での定義

フォールトトレランスのための機能のうち，CMT のトランザクションタイムアウトの定義は，cosminexus.xml の<ejb-jar>タグ内に指定します。cosminexus.xml でのフォールトトレランスのための機能の定義について次の表に示します。

表 3-59　cosminexus.xml でのフォールトトレランスのための機能の定義

指定するタグ	設定内容
<message>－<ejb-transaction-timeout>タグ <entity>－<ejb-transaction-timeout>タグ <session>－<ejb-transaction-timeout>タグ	CMT の場合のトランザクションをタイムアウトする時間を指定します。

cosminexus.xml については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「2.　アプリケーション属性ファイル（cosminexus.xml)」を参照してください。

3.15.13　実行環境での設定

ここでは，フォールトトレランスのための機能を使用する場合の設定について説明します。

なお，次に示す機能については事前に設定する必要はありません。

- コネクションプールの情報表示
- コネクションプールのクリア
- 障害調査用 SQL の出力
- オブジェクトの自動クローズ

(1)　J2EE サーバの設定

J2EE サーバの設定は，簡易構築定義ファイルで実施します。フォールトトレランスの定義は，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に指定します。

簡易構築定義ファイルでのフォールトトレランスのための機能の定義について，次の表に示します。

表 3-60　簡易構築定義ファイルでのフォールトトレランスのための機能の定義

分類	プロパティ	設定内容
コネクションの自動クローズ	ejbserver.webj2ee.connectionAutoClose.enabled	Web アプリケーションでコネクションを自動クローズするかどうかを指定します。
トランザクションタイムアウト（J2EE サーバ単位）※1	ejbserver.jta.TransactionManager.defaultTimeOut	J2EE サーバ上で開始されるトランザクションのタイムアウトのデフォルト値を指定します。
トランザクションのリカバリ	ejbserver.distributedtx.recovery.port	グローバルトランザクションを使用する場合に，トランザクションのリカバリで使用する固定ポート番号を指定します。
未決着トランザクションの確認時間のタイムアウト※2	ejbserver.distributedtx.recovery.completionCheckOnStopping.timeout	J2EE サーバ停止時に行われるトランザクション仕掛かり完了確認のタイムアウト時間を指定します。

分類	プロパティ	設定内容
障害検知のタイムアウト	ejbserver.connectionpool.validation.timeout	コネクション障害検知機能のタイムアウト時間およびコネクション数調節機能によるコネクション削除処理のタイムアウト時間を指定します。

注※1　CMT の場合，Enterprise Bean，インタフェース，メソッド単位の設定もできます。Enterprise Bean，インタフェース，メソッド単位に設定する場合には，J2EE アプリケーションの設定時にサーバ管理コマンドを使用して属性ファイルに設定します。J2EE アプリケーションの設定については，「3.15.12　cosminexus.xml での定義」を参照してください。

注※2　アプリケーション開発時にタイムアウトを設定してください。J2EE アプリケーション運用時には，トランザクションの信頼性を保証するために，タイムアウトを設定しないことをお勧めします。

簡易構築定義ファイルおよびパラメタについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.6　簡易構築定義ファイル」を参照してください。

(2)　リソースアダプタの設定

実行環境でのリソースアダプタの設定は，サーバ管理コマンドおよび属性ファイルを使用します。フォールトトレランスのための機能の定義には，Connector 属性ファイルを使用します。

Connector 属性ファイルでのフォールトトレランスのための機能の定義について次の表に示します。

表 3-61　Connector 属性ファイルでのフォールトトレランスのための機能の定義

項目	指定するパラメタ	設定内容
コネクションの障害検知	<property>タグの ValidationType および ValidationInterval	コネクションの障害を検知するタイミングおよび障害を検知する間隔を指定します。 なお，コネクションの障害検知にタイムアウトを設定する場合には，NetworkFailureTimeout でコネクション管理スレッドの使用を有効にします。※1
コネクション管理スレッド	<property>タグの NetworkFailureTimeout	コネクション管理スレッドを使用するかどうかを指定します。 コネクション管理スレッドを使用する場合，コネクションの障害検知機能，およびコネクション数調節機能のタイムアウトを使用する設定になります。07-60 より前のバージョンでは 5 秒で固定です。08-00 以降のバージョンでは任意の時間を指定できます（デフォルト値は 5 秒）。
コネクション枯渇時のコネクション取得待ち	<property>タグの RequestQueueEnable	コネクション枯渇時のコネクション取得待ちを有効にするかどうかを指定します。
	<property>タグの RequestQueueTimeout	コネクション取得の待ち時間を指定します。
コネクションの取得リトライ	<property>タグの RetryCount	コネクション取得に失敗した場合のリトライ回数を指定します。

項目	指定するパラメタ	設定内容
コネクションの取得リトライ	<property>タグの RetryInterval	コネクション取得に失敗した場合のリトライ間隔を指定します。
コネクションスイーパ	<property>タグの SweeperInterval※2	コネクションの自動破棄(コネクションスイーパ）が動作する間隔を指定します。
	<property>タグの ConnectionTimeout※2	コネクションの最終利用時刻からコネクションを自動破棄するかどうかを判定するまでの時間を指定します。
ステートメントキャンセル	<config-property>タグの CancelStatement	トランザクションタイムアウト発生時のステートメントキャンセルを有効にするかどうかを指定します。

注※1　コネクションの障害検知機能とコネクション数調節機能のタイムアウトを同じキーで設定します。このため，コネクションの障害検知機能でタイムアウトを使用する場合は，コネクション数調節機能でもタイムアウトを使用する設定となります。

注※2　設定値が 3600 秒未満の場合，KDJE48604-W が出力されます。

Connector 属性ファイルについては，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1　Connector 属性ファイル」を参照してください。

(3)　J2EE アプリケーションの設定

実行環境での J2EE アプリケーションの設定は，サーバ管理コマンドおよび属性ファイルで実施します。パフォーマンスチューニングのための機能の定義には，Session Bean 属性ファイル，Entity Bean 属性ファイルまたは Message-driven Bean 属性ファイルを使用します。

これらの属性ファイルで指定するタグは，cosminexus.xml と対応しています。cosminexus.xml での定義については，「3.15.12　cosminexus.xml での定義」を参照してください。

3.16　そのほかのリソースアダプタの機能（Connector 1.5 仕様に準拠するリソースアダプタの場合）

この節では，「3.14　パフォーマンスチューニングのための機能」，「3.15　フォールトトレランスのための機能」，および「3.17　クラスタコネクションプール機能」で説明する機能以外のリソースアダプタの機能について説明します。

この節の構成を次の表に示します。

表 3-62　この節の構成（そのほかのリソースアダプタの機能（Connector 1.5 仕様に準拠するリソースアダプタの場合））

分類	タイトル	参照先
解説	リソースアダプタのライフサイクル管理	3.16.1
	リソースアダプタのワーク管理	3.16.2
	メッセージインフロー	3.16.3
	トランザクションインフロー	3.16.4
	管理対象オブジェクトのルックアップ	3.16.5
	コネクション定義の複数指定	3.16.6
実装	アプリケーションサーバ独自の Connector 1.5 API 仕様	3.16.7
設定	Connector 1.5 仕様に準拠したリソースアダプタを使用する場合の設定	3.16.8
	属性ファイルの指定例	3.16.9
注意事項	Connector 1.5 仕様に準拠したリソースアダプタを使用する場合の注意事項	3.16.10

注　「運用」について，この機能固有の説明はありません。

これらの機能は，Connector 1.5 仕様に準拠したリソースアダプタで使用できます。アプリケーションサーバで使用できる Connector 1.5 仕様に準拠したリソースアダプタについては，「3.3.2(2)　Connector 1.5 仕様に準拠したリソースアダプタ」を参照してください。

参考

Connector 1.5 仕様に準拠したリソースアダプタなど，「3.3.2(1)　Connector 1.0 仕様に準拠したリソースアダプタ」で示したリソースアダプタ以外のリソースアダプタを使用する場合には，リソースアダプタの処理の入り口と出口で，トレースを出力することをお勧めします。出力したトレースは，障害が発生したときの要因を切り分けるために使用します。

3.16.1　リソースアダプタのライフサイクル管理

Connector 1.5 仕様に準拠したリソースアダプタを利用する場合，J2EE サーバによってリソースアダプタのライフサイクルを管理できます。

リソースアダプタのライフサイクル管理とは，リソースアダプタの開始処理と停止処理を，J2EE サーバで管理する機能です。

(1)　前提条件

リソースアダプタのライフサイクル管理は，リソースアダプタが次の条件を満たす場合に有効になります。

- リソースアダプタに，「(2)　ライフサイクル管理に使用するクラス」で示すクラスが実装されている。
- DD（ra.xml）の<connector>-<resourceadapter>-<resourceadapter-class>に，javax.resource.spi.ResourceAdapter インタフェースを実装した JavaBeans のクラス名を指定している。

なお，DD の<connector>-<resourceadapter>-<resourceadapter-class>の指定が省略されている場合，リソースアダプタのライフサイクル管理は実行されません。

(2)　ライフサイクル管理に使用するクラス

リソースアダプタのライフサイクル管理に使用するクラスについて説明します。使用するクラスには，リソースアダプタでの実装が必要なクラスと，J2EE サーバが提供するクラスがあります。

● リソースアダプタで実装が必要なクラス

リソースアダプタでは，次のクラスの実装が必要です。なお，これらのクラスは JavaBean として実装することが，Connector 1.5 仕様で規定されています。

- javax.resource.spi.ResourceAdapter インタフェースの実装クラス
- javax.resource.spi.ManagedConnectionFactory インタフェースの実装クラス
- AdminObject（管理対象オブジェクト）の実装クラス

詳細な実装については，Connector 1.5 仕様を参照してください。

● J2EE サーバが提供するクラス

J2EE サーバは，次のクラスを提供します。

- javax.resource.spi.work.WorkManager インタフェースの実装クラス
  このクラスには，doWork(Work)メソッド，scheduleWork(Work)メソッド，startWork(Work)メソッドなどが実装されています。
- javax.resource.spi.BootstrapContext インタフェースの実装クラス
  このクラスには，createTimer()メソッド，getWorkManager()メソッド，および getXATerminator()メソッドが実装されています。
  なお，アプリケーションサーバでは，BootstrapContext インタフェースの getXATerminator メソッドを呼び出した場合，null が返却されるように実装されています。

(3)　ライフサイクル管理の制御

ここでは，ライフサイクル管理の対象になる，リソースアダプタの開始処理と停止処理の制御について説明します。

リソースアダプタの開始処理および停止処理は，次のタイミングで実行されます。

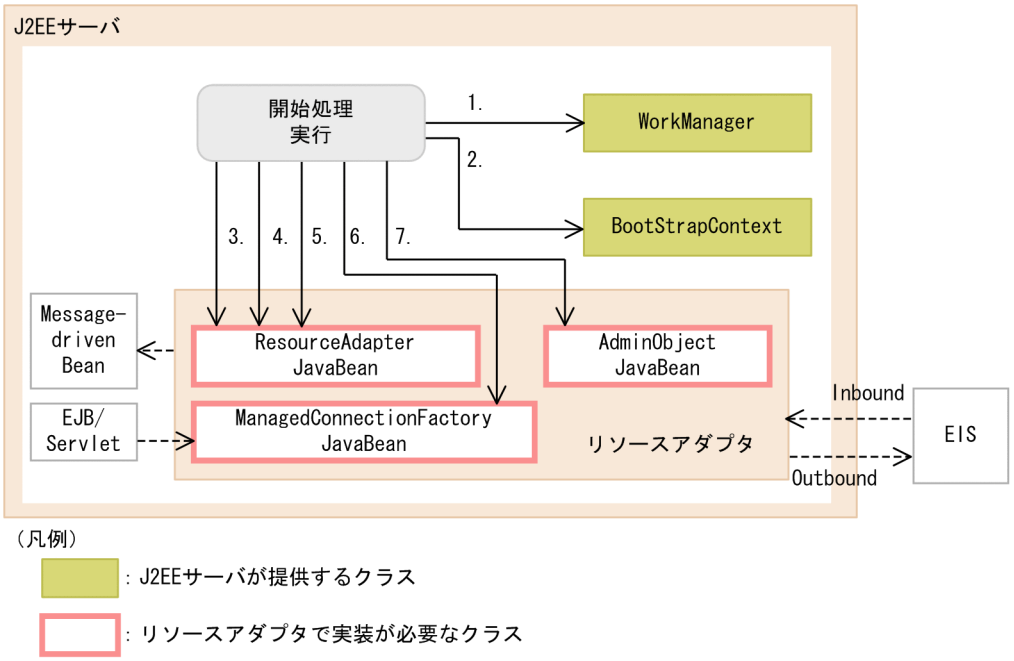
- J2EE リソースアダプタを開始または停止したとき（cjstartrar コマンドまたは cjstoprar コマンドを実行したとき）
- J2EE サーバを起動または停止したとき（cjstartsv コマンドまたは cjstopsv コマンドを実行したとき）

- リソースアダプタを含むアプリケーションを開始または停止したとき（cjstartapp コマンドまたは cjstopapp コマンドを実行したとき）

● リソースアダプタの開始処理を実行した場合の制御

リソースアダプタの開始処理を実行した場合の制御の流れを次の図に示します。

図 3-38　リソースアダプタの開始処理を実行した場合の制御の流れ



リソースアダプタの開始処理によって実行される制御について説明します。なお，項番は図中の数字と対応しています。

1. WorkManager（javax.resource.spi.work.WorkManager インタフェースの実装クラス）が生成されます。
2. BootstrapContext（javax.resource.spi.BootstrapContext インタフェースの実装クラス）が生成されます。
3. ResourceAdapterJavaBean（javax.resource.spi.ResourceAdapter インタフェースの実装クラス）が生成されます。

   ResourceAdapterJavaBean として生成される実装クラスは，リソースアダプタの DD（ra.xml）の<connector>-<resourceadapter>-<resourceadapter-class>に指定したクラスです。

   このタグで指定したクラスのインスタンス化に失敗した場合，リソースアダプタの開始が失敗します※。このとき，KDJE48589-E のメッセージが出力されます。
4. ResourceAdapterJavaBean にプロパティが設定されます。

   DD（ra.xml）の<connector>-<resourceadapter>-<config-property>に指定した値は，3.で生成された ResourceAdapterJavaBean に設定されます。設定は，JavaBean の仕様に従い，setter メソッドで実行されます。setter メソッドの呼び出しで例外が発生した場合は，KDJE48594-W のメッセージが出力されます。ただし，処理は継続されます。
5. javax.resource.spi.ResourceAdapter インタフェースの start メソッドが呼び出され，リソースアダプタが開始されます。

このメソッドの呼び出しによって例外がスローされた場合は，リソースアダプタの開始が失敗します※。このとき，KDJE48590-E のメッセージが出力されます。

6. ResourceAdapterJavaBean と ManagedConnectionFactoryJavaBean が関連づけられます（Outbound の場合）。

ライフサイクル管理機能を使用する場合，javax.resource.spi.ManagedConnectionFactory インタフェースの実装クラスは javax.resource.spi.ResourceAdapterAssociation インタフェースを実装しています。ResourceAdapterJavaBean と ManagedConnectionFactory の関連づけは，javax.resource.spi.ResourceAdapterAssociation インタフェースの setResourceAdapter(ResourceAdapter)メソッドで実行されます。

また，次の場合には，リソースアダプタの開始が失敗します※。このとき，KDJE48591-E が出力されます。

- ManagedConnectionFactoryJavaBean が javax.resource.spi.ResourceAdapterAssociation インタフェースを実装していなかった場合
- setResourceAdapter メソッドの呼び出しに対して例外がスローされた場合

7. AdminObjectJavaBean（管理対象オブジェクト）が生成され，プロパティが設定されます。

AdminObjectJavaBean として生成される実装クラスは，リソースアダプタの DD（ra.xml）の <connector>-<resourceadapter>-<adminobject>-<adminobject-class>に指定したクラスです。このタグで指定したクラスのインスタンス化に失敗した場合，リソースアダプタの開始が失敗します※。このとき，KDJE48597-E のメッセージが出力されます。

また，DD（ra.xml）の<connector>-<resourceadapter>-<adminobject>-<config-property>で指定した値が，AdminObjectJavaBean に設定されます。設定は，JavaBean の仕様に従い，setter メソッドで実行されます。AdminObjectJavaBean の setter メソッドの呼び出しで例外が発生した場合は，KDJE48598-W のメッセージが出力されます。ただし，処理は継続されます。

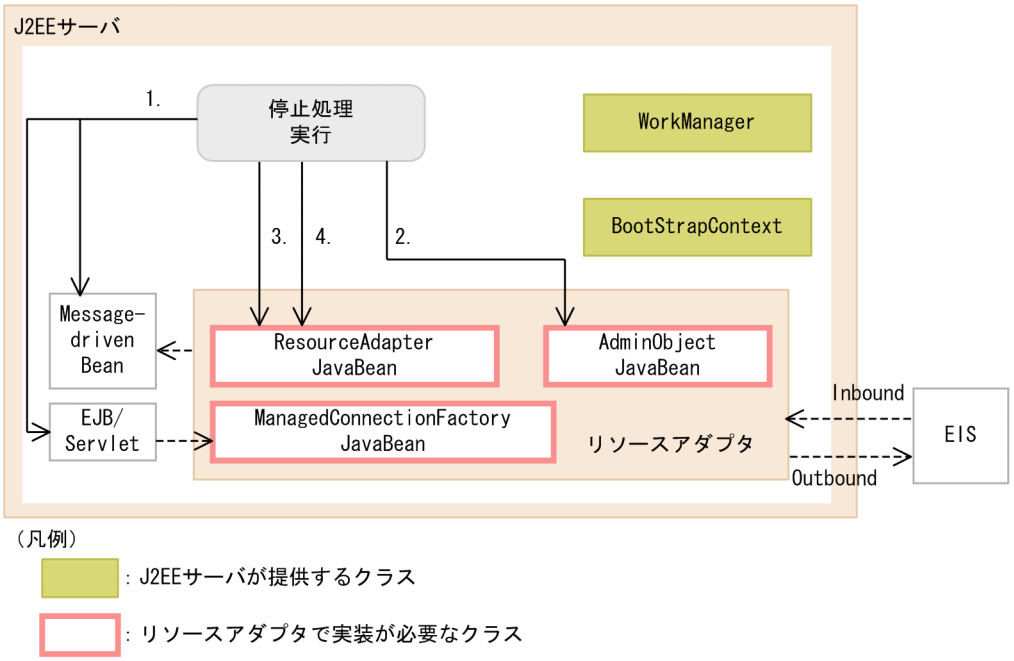
注※

アプリケーションに含まれるリソースアダプタの場合，リソースアダプタの開始が失敗したときには，そのリソースアダプタを含むアプリケーションの開始処理も失敗します。

● リソースアダプタの停止処理を実行した場合の制御

リソースアダプタの停止処理を実行した場合の制御を次の図に示します。

図 3-39　リソースアダプタの停止処理を実行した場合の制御



リソースアダプタの停止処理によって実行される制御について説明します。なお，項番は図中の数字と対応しています。

1. 停止するリソースアダプタを参照している EJB，サーブレット，Message-driven Bean がすべて停止していることが確認されます。

   次の要素が使用されていないことが確認されます。

   - Inbound リソースアダプタ
   - コネクションファクトリ
   - 管理対象オブジェクト

   これらが使用されていた場合は，リソースアダプタの停止処理が中止されます。

2. AdminObjectJavaBean（管理対象オブジェクト）が破棄されます。
3. javax.resource.spi.ResourceAdapter インタフェースの stop メソッドが呼び出され，リソースアダプタが停止します。

   このメソッドの呼び出しによって例外がスローされた場合は，KDJE48590-E のメッセージが出力されて，リソースアダプタの停止処理が失敗します。

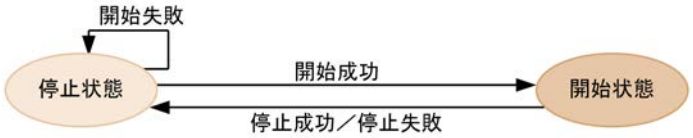
4. ResourceAdapterJavaBean が破棄されます。

● リソースアダプタの状態遷移

リソースアダプタは，リソースアダプタの開始処理または停止処理が実行されたタイミングで，「開始状態」または「停止状態」に遷移します。

リソースアダプタの状態遷移を次の図に示します。

図 3-40　リソースアダプタの状態遷移



(4)　ライフサイクル管理機能を使用するときの注意

ライフサイクル管理機能を使用する場合は，次の点に注意してください。

- J2EE サーバにデプロイされているリソースアダプタが複数ある場合，開始および停止処理が実行される順序は不定です。また，アプリケーションに含まれるリソースアダプタが複数ある場合，開始および停止処理が実行される順序は不定です。実行順序に依存する処理がある場合，動作は保証されません。
- アプリケーションに含まれるリソースアダプタの場合，アプリケーション内に含まれる EJB や Servlet の開始処理よりも前に，リソースアダプタの開始処理が実行されます。また，アプリケーション内に含まれる EJB や Servlet の停止処理よりもあとで，リソースアダプタの停止処理が実行されます。
- ResourceAdapter.stop の処理では，次に示す処理を適切に実行してください。
  - BootstrapContext.createTimer で生成した Timer の破棄(Timer#cancel)
  - WorkManager に登録した Work オブジェクトへの終了指示(Work#release)

3.16.2　リソースアダプタのワーク管理

Connector 1.5 仕様に準拠したリソースアダプタを使用する場合，リソースアダプタが使用するスレッドを J2EE サーバによって管理できます。

リソースアダプタのワーク管理とは，リソースアダプタがマルチスレッドで動作する場合に，スレッドを適切に使用するための機能です。J2EE サーバは，スレッドをプールに管理しておき，必要なリソースアダプタに割り当てます。

! 注意事項

ワーク管理の対象になる Work を実行するスレッドは，J2EE アプリケーションの実行時間監視の対象にはなりません。

J2EE アプリケーションの実行時間監視については，マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「5.3　J2EE アプリケーションの実行時間の監視とキャンセル」を参照してください。

(1)　前提条件

リソースアダプタのワーク管理機能は，リソースアダプタのライフサイクルが管理されている場合に使用できます。リソースアダプタのライフサイクル管理については，「3.16.1　リソースアダプタのライフサイクル管理」を参照してください。

(2)　ワーク管理に使用するクラス

リソースアダプタのワークを管理する場合に使用するクラスについて説明します。使用するクラスには，リソースアダプタでの実装が必要なクラスと，J2EE サーバが提供するクラスがあります。なお，J2EE サーバが提供するクラスは，ライフサイクル管理で使用するクラスと同じです。「3.16.1(2)　ライフサイクル管理に使用するクラス」を参照してください。

● リソースアダプタで実装が必要なクラス

リソースアダプタでは，次のクラスの実装が必要です。

- javax.resource.spi.work.Work インタフェースの実装クラス
  このクラスの run メソッドには，実行する処理を実装しておく必要があります。

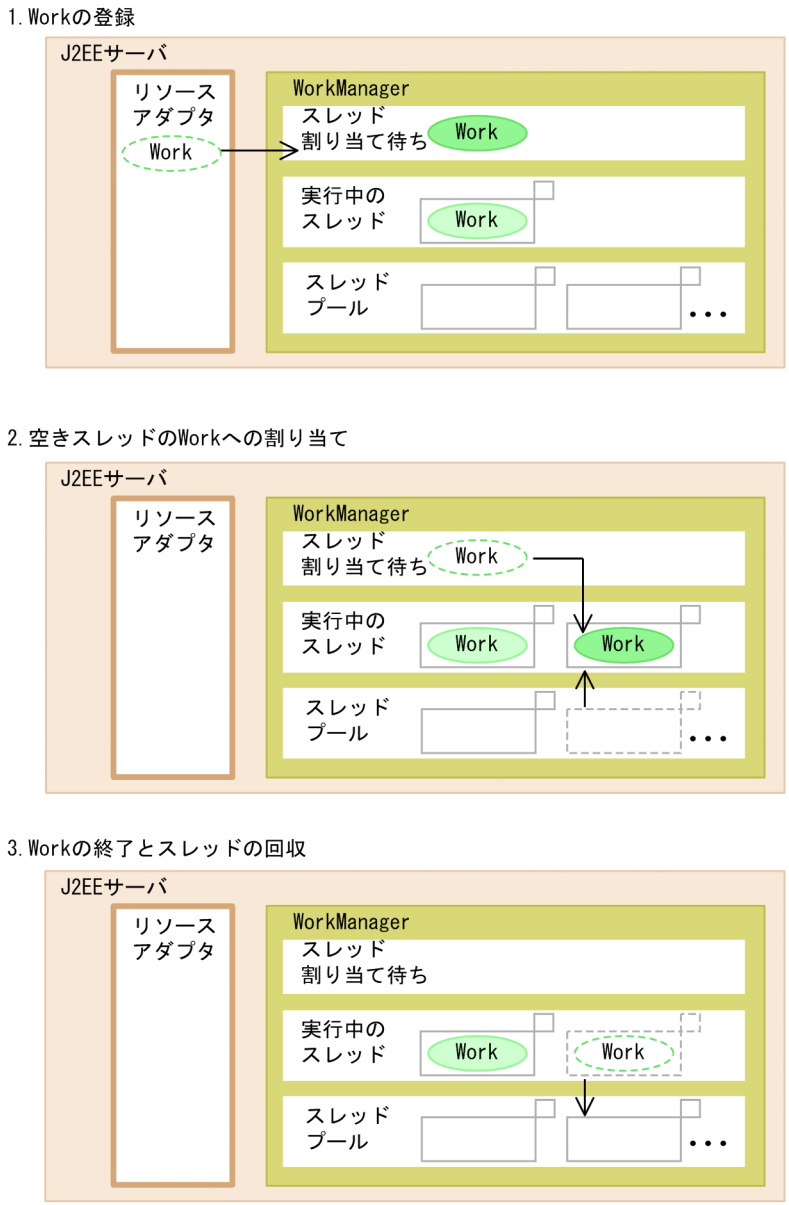
なお，ワークの登録，ワークのスレッドへの割り当て，およびワークの終了のタイミングをリソースアダプタで管理したい場合は，javax.resource.spi.work.WorkListener インタフェースの実装クラスも実装してください。

(3)　ワーク管理の流れ

ここでは，ワーク管理の処理の流れについて説明します。

ワーク管理の概要を次の図に示します。

図 3-41　ワーク管理の概要



図で示した流れについて説明します。

1. Work の登録

リソースアダプタは，Work（javax.resource.spi.work.Work インタフェースの実装クラス）を生成して，WorkManager に登録します。

このとき，WorkManager のインスタンスは，BootstrapContext 経由でリソースアダプタに渡されます。WorkManager については，「3.16.1　リソースアダプタのライフサイクル管理」を参照してください。

また，Work の登録と同時に，WorkManager に WorkListener（javax.resource.spi.work.WorkListener）も登録すると，以降，Work の登録が完了したとき（1.が実行されたとき），Work にスレッドが割り当てられたとき（2.が実行されたとき），および Work の処理が完了したとき（3.が実行されたとき）に，それぞれイベント(javax.resource.spi.work.WorkEvent インタフェースの実装クラス)を取得できます。

2. 空きスレッドへのワークの割り当て

J2EE サーバは，WorkManager に登録された Work に対して，空きスレッドを割り当てて，Work で実装されている run メソッドを実行します。

3. Work の終了とスレッドの回収

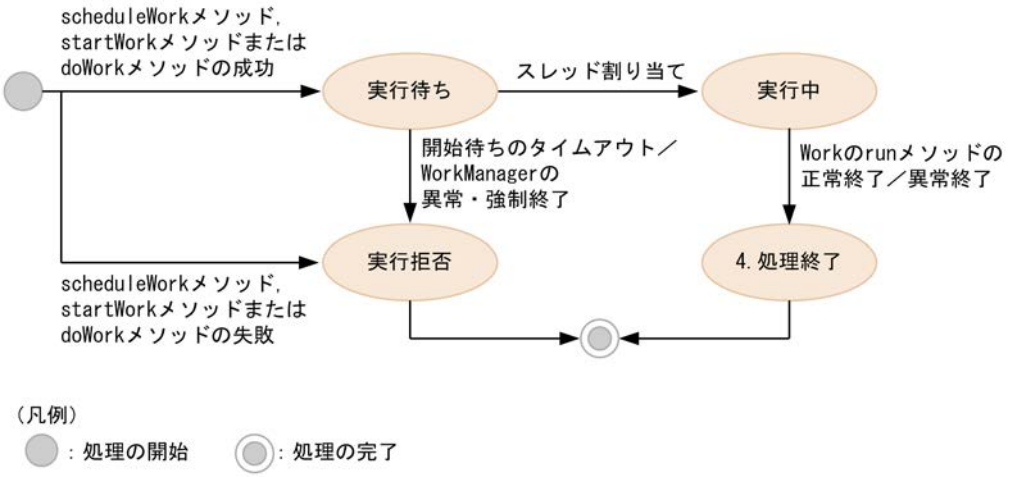
J2EE サーバは，処理が完了した Work に割り当てたスレッドを，スレッドプールに回収します。

回収されたスレッドは，スレッドプーリングでの設定に従って，プールに戻って待機状態になるか，解放されます。

なお，この流れの中で，javax.resource.spi.work.Work インタフェースまたは javax.resource.spi.work.WorkListener インタフェースのメソッドで例外が発生した場合は，それぞれ KDJE48592-E または KDJE48593-E のメッセージが出力されます。

WorkManager に登録された Work の状態遷移を，次の図に示します。

図 3-42　Work の状態遷移



なお，WorkManager に Work を登録する時に使用するメソッドによって，メソッドがリターンするタイミングが異なります。それぞれのメソッドがリターンするタイミングを次の表および図に示します。なお，リターンするタイミング以外，これらのメソッドの処理に違いはありません。

表 3-63　メソッドがリターンするタイミング

メソッド	リターンするタイミング
scheduleWork	Work を登録して，すぐにリターンします。
startWork	Work にスレッドが割り当てられたタイミングでリターンします。
doWork	Work の処理が完了したタイミングでリターンします。

図 3-43　scheduleWork メソッドがリターンするタイミング

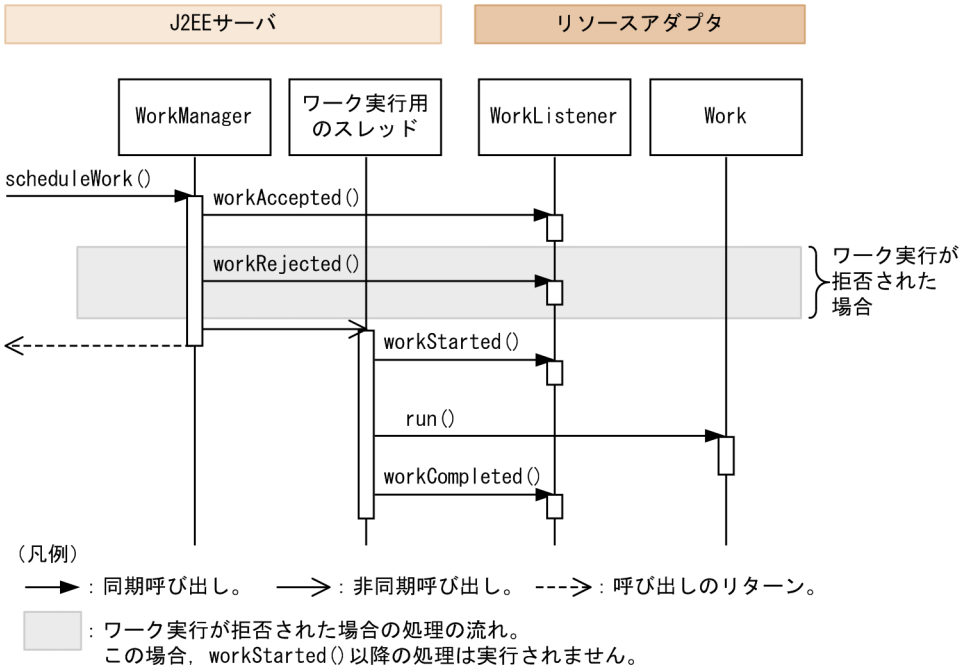


図 3-44　startWork メソッドがリターンするタイミング

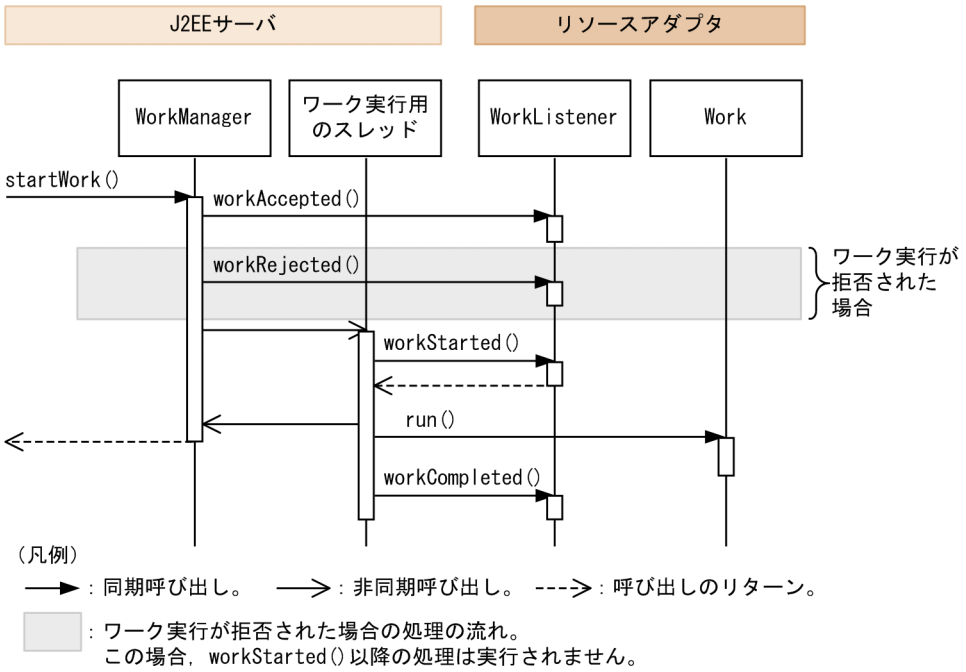
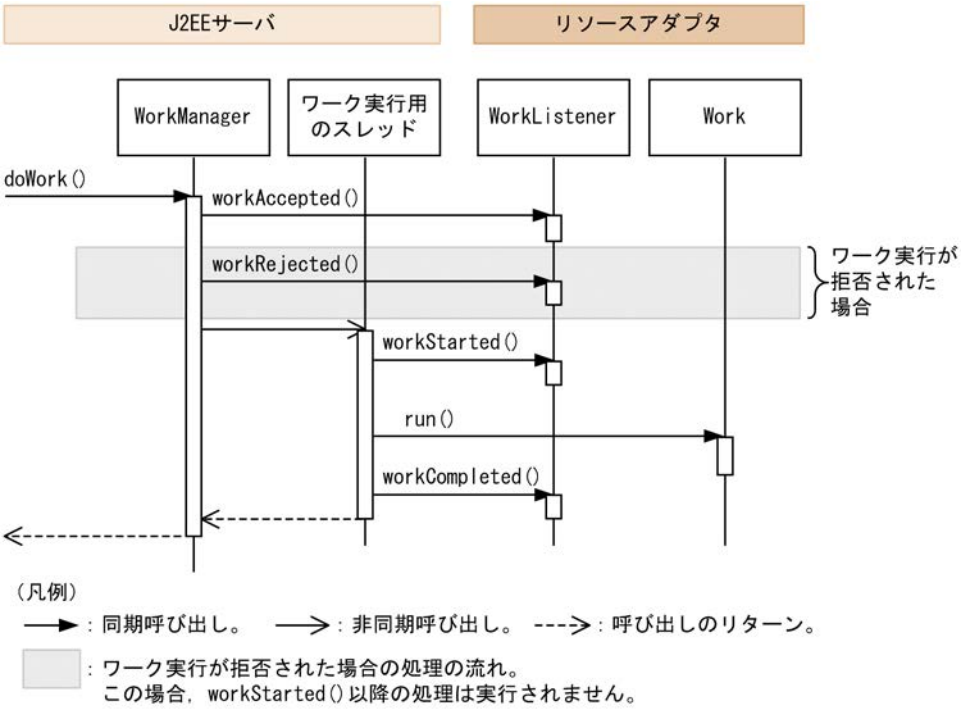


図 3-45　doWork メソッドがリターンするタイミング



(4)　スレッドプーリング

スレッドプーリングは，ワークに割り当てるスレッドを，スレッドプールで管理する機能です。

● スレッドプーリングの設定

スレッドプーリングの設定は，リソースアダプタを J2EE サーバにインポートしたあとで，Connector 属性ファイルの<hitachi-connector-property>-<resourceadapter-runtime>-<property>タグに指定して，リソースアダプタごとのプロパティとして設定します。設定手順については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「5.4　リソースアダプタのプロパティ定義」を参照してください。

スレッドプールに設定できる値を次の表に示します。

表 3-64　スレッドプールに設定できる値

設定できる値（プロパティ名）	説明
同時に実行される最大のスレッド数（MaxTPoolSize）	WorkManager で同時に実行される最大のスレッド数を設定します。 Work が登録されたときに空きスレッドがない場合，WorkManager で実行中のスレッドの数がこの値未満であれば，新規にスレッドが生成されて，Work に割り当てられます。 実行中のスレッドの数がこの値以上のときには，Work の受け付けは拒否されて，javax.resource.spi.work.WorkRejectedException がスローされます。
スレッドプールの最小スレッド数（MinTPoolSize）	スレッドプールにプールする最小スレッド数を設定します。 WorkManager に Work がまったく登録されていない場合でも，この値分のスレッドは常にプールされます。

設定できる値（プロパティ名）	説明
スレッドプールの最小スレッド数（MinTPoolSize）	また，この値に 0 を指定した場合は，Work が登録されるまで，スレッドは生成されません。
Work が割り当たっていないスレッドが解放されるまでの最大生存期間［単位：秒］（TPoolKeepalive）	Work が割り当てられていないスレッドが，スレッドプールから解放されるまでの期間を秒単位で設定します。 空きスレッドになってから TPoolKeepalive 秒経過しても Work が割り当てられなかった場合，そのスレッドは解放されます。 ただし，スレッドプールに存在するスレッド数が MinTPoolSize の値分しかない場合は，解放されません。

指定方法および指定できる値の詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1　Connector 属性ファイル」を参照してください。

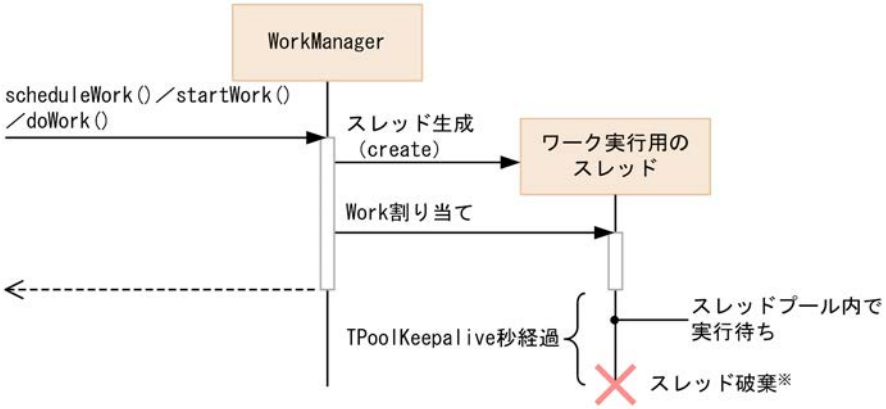
なお，ライフサイクル管理機能が有効でない場合（リソースアダプタの DD（ra.xml）に<resourceadapter-class>が指定されていない場合），スレッドプーリングを設定するプロパティの値は無視されます。

● ワーク管理で使用するスレッドのライフサイクル

ワーク管理機能で使用するスレッドのライフサイクルを次に図に示します。

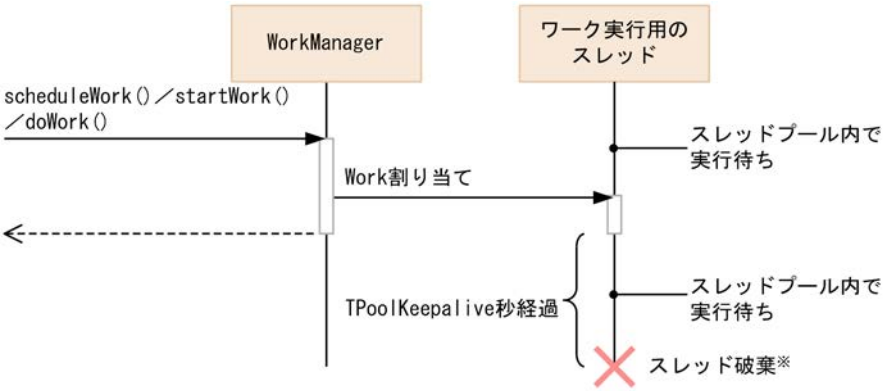
スレッドのライフサイクルは，スレッドプールと実行中スレッドの状態によって異なります。

図 3-46　スレッドのライフサイクル（スレッドプールに空きスレッドがなく，実行中のスレッド数が MaxTPoolSize 未満の場合）



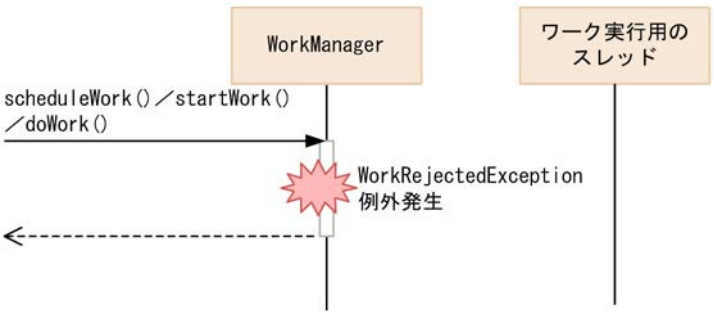
注※　スレッドプールのスレッド数がMinTPoolSizeの場合は，破棄されません。

図 3-47　スレッドのライフサイクル（スレッドプールに空きスレッドがある場合）



注※　スレッドプールのスレッド数がMinTPoolSizeの場合は，破棄されません。

図 3-48　スレッドのライフサイクル（スレッドプールに空きスレッドがなく，実行中のスレッド数が MaxTPoolSize に達している場合）



スレッドが生成されるのは，リソースアダプタによって scheduleWork メソッド，startWork メソッドまたは doWork メソッドが呼び出された場合に，次の状態の両方に当てはまるときです。

- スレッドプールに空きスレッドがない。
- WorkManager で実行中のスレッド数が MaxTPoolSize 未満である。

スレッドは，Work の処理が完了するとスレッドプールで実行待ち状態になります。次の Work が登録されると，再度実行状態になります。

スレッドプール内で，TPoolKeepalive で指定した秒数が経過しても Work が割り当てられなかった場合，スレッドは解放されます。ただし，そのスレッドを解放することでスレッドプール内のスレッド数が MinTPoolSize 未満になる場合は，解放されません。

● Message-driven Bean のインスタンスプールとスレッドプーリングとの関係

リソースアダプタが Work を使用して Message-driven Bean を呼び出す場合の動作は，Message-driven Bean のインスタンスプールの最大数と，リソースアダプタの MaxTPoolSize の関係によって異なります。

Message-driven Bean のインスタンスプールの最大数とリソースアダプタの MaxTPoolSize の関係ごとの動作を次の表に示します。これは，一つのリソースアダプタから一つの Message-driven Bean を呼び出す場合の動作です。

表 3-65　リソースアダプタが Work によって Message-driven Bean を呼び出すときの動作

Message-driven Bean のインスタンスプールの最大数とリソースアダプタの MaxTPoolSize の関係	動作
MaxTPoolSize>Message-driven Bean のインスタンスプールの最大数	Message-driven Bean のインスタンスプールの最大数に達するまで，Work から Message-driven Bean を実行できます。 Message-driven Bean のインスタンスプールの最大数を超える数の Work は，Message-driven Bean のインスタンスの取得待ちの状態になります。
MaxTPoolSize=Message-driven Bean のインスタンスプールの最大数	Message-driven Bean のインスタンスプールの最大数に達するまで，Work から Message-driven Bean を実行できます。 Message-driven Bean のインスタンスプールの最大数を超える数の Work に対しては，javax.resource.spi.work.WorkRejectedException 例外がスローされます。インスタンスの取得待ちにはなりません。
MaxTPoolSize<Message-driven Bean のインスタンスプールの最大数	同時に使用できる Message-driven Bean のインスタンス数は，MaxTPoolSize の数になります。実行中の Work の数が MaxTPoolSize に達するまで，Work から Message-driven Bean を実行できます。 MaxTPoolSize を超える数の Work に対しては，javax.resource.spi.work.WorkRejectedException 例外がスローされます。実行待ちにはなりません。

ポイント

ワーク管理機能のスレッドプーリングはリソースアダプタ単位に設定します。一方，Message-driven Bean のインスタンスプールは Message-driven Bean 単位に設定します。

リソースアダプタが複数の Message-driven Bean を呼び出す場合，リソースアダプタの MaxTPoolSize には，それぞれの Message-driven Bean で必要なスレッド数を考慮して，合計した値を設定してください。

(5)　ワーク管理の開始処理と終了処理

ワーク管理の開始処理および終了処理は，ライフサイクル管理機能の開始処理および終了処理のタイミングで実行されます。

それぞれの処理で実行されることを次に示します。

● 開始処理

ワーク管理の開始処理では，スレッドプーリングの設定に応じて，スレッドの生成が実行されます。MinTPoolSize が 1 以上に設定されている場合は，MinTPoolSize 分のスレッドが生成されて，空きスレッドとしてスレッドプールに格納されます。

● 終了処理

終了処理で実行される内容は，正常終了の場合と，異常終了または強制終了の場合とで一部異なります。なお，異常終了とは，ResourceAdapter インタフェースの stop メソッド実行時に例外がスローされる場合のことです。また，強制終了とは，cjstopapp -force コマンドまたは cjstopapp -cancel コマンド実行の延長で実行される終了処理のことです。

参考

cjstopsv -f コマンドを実行した場合は，リソースアダプタの終了処理が実行されないで J2EE サーバが停止されます。このため，ここで説明する処理は実行されません。

1. J2EE サーバから ResourceAdapter インタフェースの stop メソッドが呼び出され，リソースアダプタに対して，停止処理が指示されます。

   また，この処理の延長で Work の release メソッドが呼び出されて，Work が解放されます。

2. リソースアダプタによって，新規の Work の受け付けが停止されます。

   1.の処理を実行したあとなので，通常，Work の登録要求はありません。Work の登録要求があった場合，例外として，javax.resource.spi.work.WorkRejectedException がスローされます。

3. Work の処理完了が待機されます。

   実行中の Work の run メソッドの実行がすべて完了するまで待機されます。

ポイント

- Work から呼び出している Message-driven Bean に対してメソッドキャンセルが実行されると，Message-driven Bean は強制終了します。ただし，その場合，実行中の Work の処理は停止されません。
- Work から Message-driven Bean を呼び出している場合，リソースアダプタよりも先に Message-driven Bean が終了します。Message-driven Bean 停止後に Work から Message-driven Bean を呼び出したり，javax.resource.spi.endpoint.MessageEndpointFactory の createEndpoint メソッドを呼び出したりすると，例外がスローされます。

  詳細は，「3.16.3　メッセージインフロー」を参照してください。

(6)　ワーク管理機能を使用するときの注意

- Work および WorkListener のメソッド内では，J2EE サーバの機能として，メッセージインフロー規約に基づく Message-driven Bean の呼び出しだけが使用できます。
- Work および WorkListener のメソッドは，スレッドセーフにしてください。
- WorkListener のメソッドには，実行スレッドに依存した処理を実装しないでください。実行スレッドに依存した処理がある場合，動作は保証されません。

3.16.3　メッセージインフロー

メッセージインフローは，リソースアダプタと Message-driven Bean 間の規約です。Connector 1.5 仕様のリソースアダプタは，EIS などのメッセージプロバイダからのメッセージを受け付けて，アプリケーションサーバ上のメッセージエンドポイント（Message-driven Bean）を動作させることができます。メッセージエンドポイントでは，メッセージプロバイダから送信されたメッセージを，非同期に処理します。

アプリケーションサーバでは，メッセージの配信方法として，次の 2 種類の方法を使用できます。

- Non-Transacted Delivery
- Transacted Delivery

これらは，呼び出し元の EIS がトランザクションに参加するかどうかが異なります。

(1)　前提条件

メッセージインフローは，リソースアダプタおよび Message-driven Bean が次の条件を満たす場合に有効になります。

- メッセージインフローで呼び出される Message-driven Bean は，EJB 2.1 以降に準拠している必要があります。

  EJB 2.1 以降では，Message-driven Bean は javax.jms.MessageListener だけではなく，任意のメッセージリスナを実装できます。メッセージリスナは，リソースアダプタと Message-driven Bean 間の

メッセージ配送に使用するリスナです。リソースアダプタがサポートしているメッセージリスナを Message-driven Bean に実装することで，汎用的なメッセージ受信を実現できます。

EJB 2.0 の Message-driven Bean でメッセージインフローを実行しようとした場合，アプリケーション開始時にエラーになり，開始に失敗します。この場合は，KDJE42088-E のメッセージが出力されます。

- メッセージインフローを実行する場合，リソースアダプタおよび Message-driven Bean の属性として，次の設定が必要です。
  - 管理対象オブジェクトに設定する情報（リソースアダプタ）
  - Message-driven Bean とリソースアダプタの対応づけ（Message-driven Bean およびリソースアダプタ）
  - ActivationSpec に設定する情報（Message-driven Bean）
  - Message-driven Bean が使用するインタフェース（Message-driven Bean およびリソースアダプタ）

  メッセージインフローの処理を実行する場合のリソースアダプタと J2EE アプリケーションの設定については，「3.16.8　Connector 1.5 仕様に準拠したリソースアダプタを使用する場合の設定」を参照してください。

また，Transacted Delivery は，次の表に示す条件を満たす場合に有効になります。

表 3-66　Transacted Delivery が有効になる条件

J2EE サーバの設定		Message-driven Bean の設定		
		トランザクション管理種別<transaction-type>が Container		トランザクション管理種別<transaction-type>が Bean
		トランザクション属性<trans-attribute>が Required	トランザクション属性<trans-attribute>が NotSupported	
ejbserver.distributedtx.XATransaction.enabled	true	○	×	×
	false	△	×	×

（凡例）

○：メッセージを配信する EIS をグローバルトランザクションに含めてトランザクションを開始する。

△：メッセージを配信する EIS をトランザクションに含めないでローカルトランザクションを開始する。

×：トランザクションを開始しない。

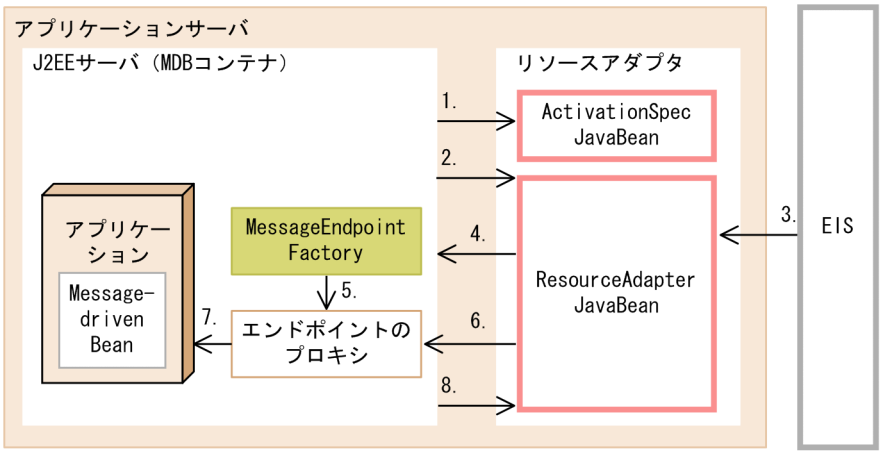
また，Transacted Delivery を使用する場合は，Connector 1.5 仕様に準拠し，Transacted Delivery に対応したリソースアダプタを使用する必要があります。設定の詳細については，ご使用のリソースアダプタのドキュメントを参照してください。

(2)　メッセージインフローの制御の流れ（Non-Transacted Delivery の場合）

Non-Transacted Delivery とは，メッセージを配信する EIS がトランザクションに参加しないメッセージ配信です。

Non-Transacted Delivery でメッセージインフローを使用する場合の制御の流れを次の図に示します。

図 3-49　Non-Transacted Delivery でメッセージインフローを使用する場合の制御の流れ



注　MDBコンテナは，Message-driven Beanを実行する機能を持った専用のEJBコンテナです。

Non-Transacted Delivery でメッセージインフローを使用した場合に実行される制御について説明します。なお，項番は図中の数字と対応しています。

1. アプリケーションの開始処理によって，リソースアダプタの ActivationSpecJavaBean にプロパティが設定されます。

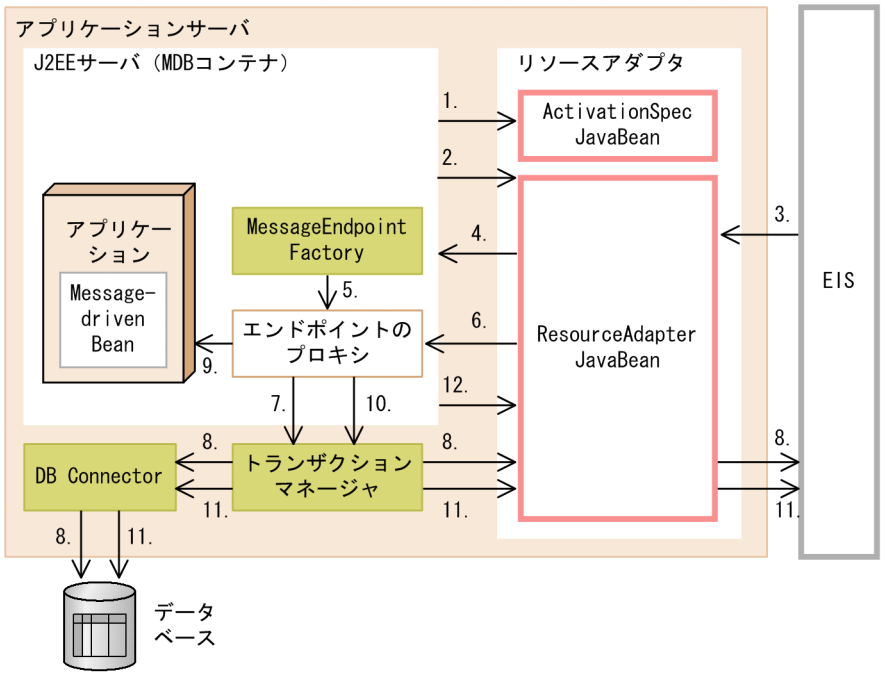
   Message-driven Bean の属性の<activation-config>に指定した内容が設定されます。
2. J2EE サーバから endpointActivation メソッドが呼び出されます。これによって，リソースアダプタにメッセージエンドポイントの開始が通知されます。
3. EIS からメッセージが送信されます。
4. リソースアダプタから MessageEndpointFactory に対して，createEndpoint メッセージが呼び出されます。
5. MessageEndpointFactory がエンドポイントのプロキシを生成します。
6. リソースアダプタからエンドポイントのプロキシに対して，onMessage メソッドなどのメッセージリスナのメソッドが呼び出されます。
7. エンドポイントのプロキシから Message-driven Bean に対して，onMessage メソッドなどのメッセージリスナのメソッドが呼び出されます。
8. 処理が完了したら，J2EE サーバから endpointDeactivation メソッドが呼び出されます。これによって，リソースアダプタにメッセージエンドポイントの停止が通知されます。

(3)　メッセージインフローの制御の流れ（Transacted Delivery の場合）

Transacted Delivery とは，メッセージを配信する EIS がトランザクションに参加するメッセージ配信です。

Transacted Delivery でメッセージインフローを使用する場合の制御の流れを次の図に示します。

図 3-50　Transacted Delivery でメッセージインフローを使用する場合の制御の流れ



（凡例）

：リソースアダプタで実装が必要なクラス

注　MDBコンテナは，Message-driven Beanを実行する機能を持った専用のEJBコンテナです。

Transacted Delivery でメッセージインフローを使用した場合に実行される制御について説明します。なお，項番は図中の数字と対応しています。

1. アプリケーションの開始処理によって，リソースアダプタの ActivationSpecJavaBean にプロパティが設定されます。

   Message-driven Bean の属性の<activation-config>に指定した内容が設定されます。
2. J2EE サーバから endpointActivation メソッドが呼び出されます。これによって，リソースアダプタにメッセージエンドポイントの開始が通知されます。
3. EIS からメッセージが送信されます。
4. リソースアダプタから MessageEndpointFactory に対して，createEndpoint メッセージが呼び出されます。
5. MessageEndpointFactory がエンドポイントのプロキシを生成します。
6. リソースアダプタからエンドポイントのプロキシに対して，onMessage メソッドなどのメッセージリスナのメソッドが呼び出されます。
7. 呼び出されたエンドポイントのプロキシからトランザクションマネージャに対して，トランザクションの開始指示が呼び出されます。
8. トランザクションマネージャによって start メソッドが呼び出され，トランザクションが開始されます。
9. エンドポイントのプロキシから Message-driven Bean に対して，onMessage メソッドなどのメッセージリスナのメソッドが呼び出されます。
10. エンドポイントのプロキシからトランザクションマネージャに対して，トランザクションの決着が指示されます。

11. トランザクションマネージャによって prepare メソッド，commit メソッドなどのトランザクション決着メソッドが呼び出され，トランザクションが決着します。
12. 処理が完了したら，J2EE サーバから endpointDeactivation メソッドが呼び出されます。これによって，リソースアダプタにメッセージエンドポイントの停止が通知されます。

(4)　メッセージエンドポイントのデプロイとアンデプロイ

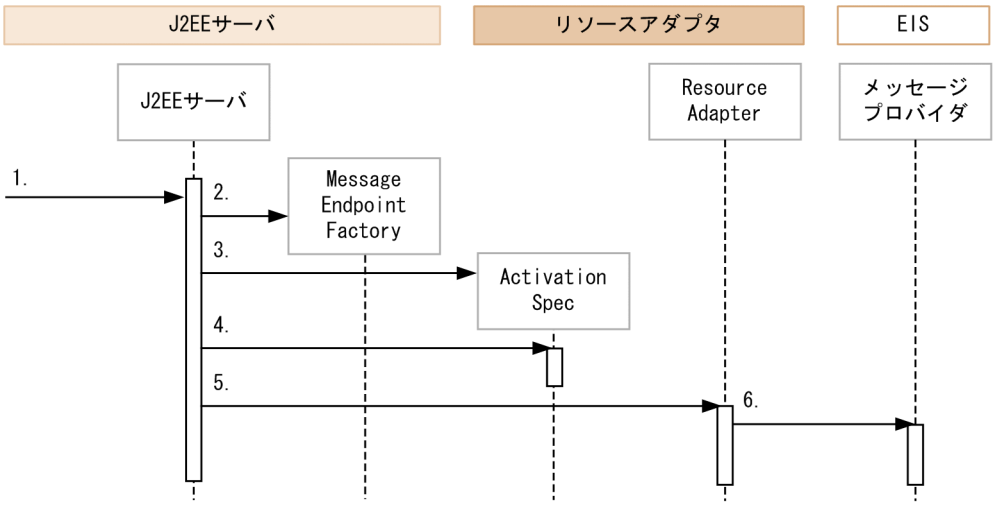
ここでは，メッセージエンドポイントのデプロイとアンデプロイで実行される処理について説明します。

● メッセージエンドポイントのデプロイ

メッセージエンドポイントのデプロイ時に実行される処理について説明します。メッセージエンドポイントのデプロイは，リソースアダプタが開始済みの状態で，Message-driven Bean を含むアプリケーションを開始したときに実行されます。

メッセージエンドポイントのデプロイ時の処理を次の図に示します。

図 3-51　メッセージエンドポイントのデプロイ時の処理



メッセージエンドポイントのデプロイで実行される処理について説明します。項番は図中の番号に対応しています。

1. リソースアダプタが開始済みの状態で，Message-driven Bean を含む J2EE アプリケーションを開始します。
2. J2EE サーバが，MessageEndpointFactory を生成します。
   MessageEndpointFactory は，J2EE サーバが提供する javax.resource.spi.endpoint.MessageEndpointFactory のインスタンスです。
   javax.resource.spi.endpoint.MessageEndpointFactory はリソースアダプタにエンドポイントのインスタンスを提供するファクトリクラスです。
3. J2EE サーバが，ActivationSpec を生成します。
   ActivationSpec は，Message-driven Bean（エンドポイント）の開始に必要な情報を設定する JavaBean です。
4. J2EE サーバが，ActivationSpec のプロパティを設定します。
   ActivationSpec のプロパティに設定する情報は，Message-driven Bean を含むアプリケーションの属性として設定した情報です。

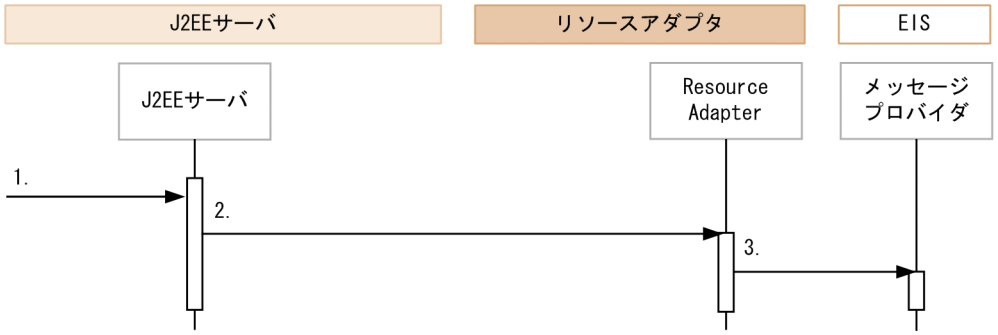
5. J2EE サーバが，
javax.resource.spi.ResourceAdapter#endpointActivation(MessageEndpointFactory, ActivationSpec)メソッドを呼び出します。
このとき，引数として，生成・設定した MessageEndpointFactory と ActivationSpec のインスタンスが指定されます。なお，endpointActivation メソッドの呼び出しで例外が発生した場合は，KDJE43174-E のメッセージが出力され，アプリケーションの開始が中止されます。
6. リソースアダプタは，5.のメソッドが呼び出されると，メッセージプロバイダからのメッセージ受信に対する準備をします。

● メッセージエンドポイントのアンデプロイ

メッセージエンドポイントのアンデプロイ時に実行される処理について説明します。メッセージエンドポイントのアンデプロイは，Message-driven Bean を含むアプリケーションを停止したときに実行されます。

メッセージエンドポイントのアンデプロイ時の処理を次の図に示します。

図 3-52　メッセージエンドポイントのアンデプロイ時の処理



メッセージエンドポイントのアンデプロイで実行される処理について説明します。項番は図中の番号に対応しています。

1. Message-driven Bean を含む J2EE アプリケーションを停止します。
2. J2EE サーバが，
javax.resource.spi.ResourceAdapter#endpointDeactivation(MessageEndpointFactory, ActivationSpec)を呼び出します。
このとき，引数として，デプロイ時に指定した MessageEndpointFactory と ActivationSpec と同じインスタンスが指定されます。なお，このメソッドの呼び出しで例外が発生した場合，メッセージ KDJE43175-W が出力されます。ただし，例外が発生した場合も，アプリケーションの停止処理は続行されます。
3. リソースアダプタは，2.のメソッドが呼び出されると，メッセージプロバイダからのメッセージ受信終了の処理をします。

● メッセージ配送をするときのリソースアダプタの処理

ここでは，メッセージ配送をするときのリソースアダプタの処理について説明します。

リソースアダプタからのメッセージエンドポイント（Message-driven Bean）の呼び出しは，メッセージエンドポイントのプロキシを使用して実行されます。このプロキシは，リソースアダプタから javax.resource.spi.endpoint.MessageEndpointFactory の createEndpoint メソッドを呼び出すことで得られます。

! 注意事項

- アプリケーションの停止後に MessageEndpointFactory のメソッドを呼び出すと，javax.resource.spi.UnavailableException 例外がスローされます。このとき，メッセージ KDJE43177-E が出力されます。また，メッセージエンドポイントのメソッドを呼び出した場合は，java.lang.IllegalStateException 例外がスローされます。このとき，メッセージ KDJE43177-E が出力されます。
- Transacted Delivery の場合，メッセージ配信中に J2EE アプリケーションを停止すると，トランザクションがロールバックされることがあります。その際に KDJE31011-E または KDJE31012-E エラーが発生することがあります。J2EE サーバの停止時に動作する J2EE アプリケーションの停止でも，これらのエラーが発生することがあります。

3.16.4　トランザクションインフロー

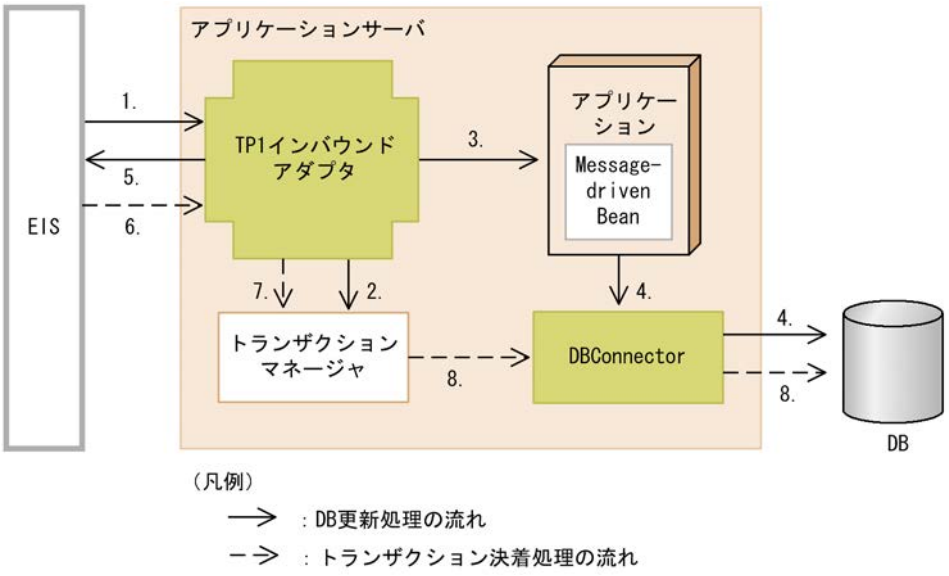
トランザクションインフローとは，アプリケーションサーバ上のメッセージエンドポイント（Message-driven Bean）をメッセージプロバイダのトランザクションに参加させる際のリソースアダプタとアプリケーションサーバ間の規約です。

メッセージエンドポイント（Message-driven Bean）は，メッセージプロバイダのトランザクション識別子に関連づけられトランザクションに参加します。

なお，トランザクションインフローは，TP1 インバウンド連携機能でだけ使用できます。

トランザクションインフローの制御の流れを次に示します。

図 3-53　トランザクションインフローの制御の流れ



図中の処理の流れについて説明します。なお，項番は図中の数字と対応しています。

1. EIS から TP1 インバウンドアダプタへメッセージが送信されます。
2. TP1 インバウンドアダプタは，トランザクションマネージャへ EIS のトランザクション識別子を伝播します。
3. TP1 インバウンドアダプタが Message-driven Bean を実行します。
4. Message-driven Bean から，SQL を実行して DB を更新します。

5. Message-driven Bean 実行後，TP1 インバウンドアダプタから EIS へ実行結果が送信されます。
6. 実行結果を受けて，EIS が TP1 インバウンドアダプタへトランザクションの決着指示を送信します。
7. TP1 インバウンドアダプタから，トランザクションマネージャへトランザクション決着指示が送信されます。
8. トランザクションが決着します。

3.16.5　管理対象オブジェクトのルックアップ

管理対象オブジェクト（AdminObject）をルックアップによって取得できます。管理対象オブジェクトは，J2EE アプリケーションの中からメッセージを送信したり，同期受信したりする場合に，メッセージの送信先の情報を得るために必要です。管理対象オブジェクトをルックアップするためには，リソースアダプタと J2EE アプリケーションの設定が必要です。ここでは，設定の概要について説明します。

参考

管理対象オブジェクトの仕様は，リソースアダプタの仕様に依存します。詳細は，使用するリソースアダプタの仕様に従ってください。

(1)　ルックアップの対象にする管理対象オブジェクトの設定

ルックアップの対象にする管理対象オブジェクトの情報は，リソースアダプタのプロパティとして設定します。リソースアダプタのプロパティは，Connector 属性ファイルで設定します。

管理対象オブジェクトをルックアップの対象にする場合は，<adminobject-name>タグに管理対象オブジェクト名を必ず指定してください。管理対象オブジェクト名は，管理対象オブジェクトがリソースアダプタごとに複数定義された場合に，管理対象オブジェクトを一意に識別するために使用されます。

設定項目については，「3.16.8　Connector 1.5 仕様に準拠したリソースアダプタを使用する場合の設定」を参照してください。

(2)　J2EE アプリケーションの設定

管理対象オブジェクトをルックアップする J2EE アプリケーションの設定は，属性ファイルまたはアノテーションで設定できます。

(a)　属性ファイルを使用する場合

<resource-env-ref>タグ下の次の要素を設定します。

<resource-env-ref-name>タグ下にルックアップで使用する名称を指定します。<resource-env-ref-type>タグ下に参照する管理対象オブジェクトの型を指定します。<linked-adminobject>タグ下にリソースアダプタの管理対象オブジェクトの名前，リソースアダプタの表示名を指定します。

! 注意事項

EJB 2.1 以降で管理対象オブジェクトを参照するための要素として規定されている<message-destination-ref>タグは，アプリケーションサーバでは使用できません。

設定については，「3.16.8　Connector 1.5 仕様に準拠したリソースアダプタを使用する場合の設定」を参照してください。

(b)　アノテーションを使用する場合

@Resource の mappedName 属性に，ルックアップする管理対象オブジェクト名を指定します。次の形式で指定します。リソースアダプタの表示名と管理対象オブジェクト名を区切る区切り文字には「!#」を使用します。

```
@Resource(mappedName="リソースアダプタの表示名!#管理対象オブジェクト名")
```

3.16.6　コネクション定義の複数指定

Connector 1.5 仕様に準拠したリソースアダプタを使用する場合，一つのリソースアダプタに対して複数のコネクション定義を指定できます。コネクションプールやログ出力などについての設定は，コネクション定義ごとに設定できます。

リソースアダプタ内のコネクション定義の識別には，**コネクション定義識別子**を使用します。コネクション識別子は，DD 内の<connectionfactory-interface>の指定値です。<connectionfactory-interface>の値は，リソースアダプタ内のコネクション定義ごとに一意の値を持っています。

なお，リソースアダプタに含まれるコネクション定義ごとのコネクション定義識別子については，次のサーバ管理コマンドに-outbound オプションを指定して実行すると確認できます。

- cjlistapp コマンド（アプリケーションに含まれる RAR ファイルの場合）
  詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjlistapp（アプリケーションの一覧表示)」を参照してください。
- cjlistres コマンド
  詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjlistres（リソースの一覧表示)」を参照してください。
- cjlistrar コマンド
  詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjlistrar（リソースアダプタの一覧表示)」を参照してください。

ここでは，コネクション定義に指定できる要素と指定個所について説明します。また，コネクション定義を複数指定する場合に留意することについて説明します。

(1)　コネクション定義に指定できる要素と指定個所

ここでは，コネクション定義に指定できる要素と指定個所について説明します。

● DD での指定個所とコネクション定義に指定できる要素

コネクション定義は，Connector 1.5 仕様に対応した DD の，<connection-definition>下に指定します。<connection-definition>の下に指定する要素を次に示します。

- <managedconnectionfactory-class>
- <config-property>
- <connectionfactory-interface>
- <connectionfactory-impl-class>
- <connection-interface>
- <connection-impl-class>

● Connector 属性ファイルの階層構造とコネクション定義の対応

DD（ra.xml）の階層構造の変更に合わせて，属性ファイルにもコネクション定義に対応する階層が追加されます。

J2EE サーバにインポート済みのリソースアダプタの場合，リソースアダプタのプロパティの変更には，Connector 属性ファイルを使用します。Connector 属性ファイルに定義した値は，サーバ管理コマンドを使用して J2EE サーバ上のリソースアダプタに反映します。属性ファイルによるプロパティの設定手順については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「3.5　属性ファイルによるプロパティの設定」を参照してください。

なお，リソースアダプタの動作に関連するプロパティには，コネクション定義ごとに定義する項目と，リソースアダプタ全体に定義する項目があります。コネクション定義ごとに指定する項目については，Connector 属性ファイルの<outbound-resourceadapter>-<connection-definition>下に指定します。リソースアダプタ全体に対して指定する項目については，<resourceadapter-runtime>-<property>下に指定します。

指定方法の詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1.1　Connector 属性ファイルの指定内容」を参照してください。

● J2EE アプリケーションでのリンク解決の定義方法（<resource-ref>-<linked-to>，<cmp-map>-<datasource-name>または mappedName での指定方法）

Connector 1.5 仕様に準拠したリソースアダプタを参照する J2EE アプリケーションでは，リンク解決のために，J2EE アプリケーションがどのコネクション定義を参照するかを定義する必要があります。

定義は，次のどれかの方法で指定します。

- J2EE アプリケーションの属性ファイル（WAR 属性ファイル，Session Bean 属性ファイル，Entity Bean 属性ファイルまたは MessageDrivenBean 属性ファイル）の<resource-ref>-<linked-to>
- J2EE アプリケーションの属性ファイル（Entity Bean 属性ファイル）の<cmp-map>-<datasource-name>
- アノテーション@Resource の mappedName 属性

! 注意事項

参照先のリソースが Connector 1.5 仕様に準拠したリソースアダプタの場合は，必ずコネクション定義識別子を指定してください。指定を省略した場合，リンク解決に失敗します。また，参照先のリソースが Connector 1.0 仕様に準拠したリソースアダプタの場合は，コネクション定義識別子を指定しないでください。指定した場合，リンク解決に失敗します。

コネクション定義識別子を含む参照先のリソースは，次の形式で指定します。

```
<リソースアダプタの表示名>!<コネクション定義識別子>
```

指定内容について説明します。

<リソースアダプタの表示名>

参照先のリソースアダプタの DD（ra.xml）の<connector>-<display-name>要素の値です。

<コネクション定義識別子>

参照先のリソースアダプタの DD（ra.xml）の<connector>-<resourceadapter>-<outbound-resourceadapter>-<connection-definition>-<connectionfactory-interface>要素の値です。

なお，リソースアダプタの表示名およびコネクション定義識別子は，次のサーバ管理コマンドでも確認できます。

- cjlistapp コマンド（アプリケーションに含まれる RAR ファイルの場合）
- cjlistres コマンド
- cjlistrar コマンド

(2)　コネクションプールを使用する場合の留意事項

複数のコネクション定義が指定されているリソースアダプタでは，コネクションプールをコネクション定義単位で管理できます。コネクションプールについての定義は，Connector 属性ファイルの<outbound-resourceadapter>-<connection-definition>-<connector-runtime>-<property>下の要素で指定して設定してください。

次に，コネクションプールが複数になる場合の留意事項について説明します。

● コネクションプールを対象にしたコマンドの実行

コネクションプールを対象にしたサーバ管理コマンドを実行するときには，どのコネクションプールを対象にするかを指定する必要があります。

次のコマンドを実行する場合には，コネクション定義識別子の指定が必要です。コネクション定義識別子は，これらのコマンドのオプションで指定します。

- cjclearpool コマンド
  詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjclearpool（コネクションプール内のコネクション削除）」を参照してください。
- cjlistpool コマンド
  詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjlistpool（コネクションプールの一覧表示）」を参照してください。

参考

クラスタコネクションプールで使用するコマンドについては，対象になりません。クラスタコネクションプールで使用するコマンドは，アプリケーションサーバが提供しているリソースアダプタだけで実行できます。

● コネクションプールのウォーミングアップ処理

コネクションプールのウォーミングアップ処理は，コネクション定義ごとに実行されます。

コネクションプールのウォーミングアップ処理が実行される場合に，特定のコネクション定義に対応するコネクションプールのウォーミングアップに失敗しても，それ以外のコネクションプールに対するウォーミングアップ処理は継続されます。

(3)　トランザクションリカバリをする場合の留意事項

トランザクションリカバリは，コネクション定義ごとに実行されます。このため，コネクション定義を複数指定した場合，OTS にはコネクション定義ごとにリソースが登録されます。

(4)　リソースへの接続テストをする場合の留意事項

リソースアダプタの接続テストを実行する場合，特定のコネクション定義に対する接続テストでエラーが発生した場合も，処理は中止されません。すべてのコネクション定義に対して接続テストが実行されます。た

だし，どれかのコネクション定義でエラーが発生した場合，コマンドの戻り値は異常終了を示す戻り値になります。

(5)　稼働情報およびリソース枯渇監視情報を出力する場合の留意事項

コネクション定義が複数指定されている場合，稼働情報およびリソース枯渇監視情報は次のように出力されます。

稼働情報（稼働情報ファイルに出力される情報）

一つ目のコネクション定義についての情報だけが出力されます。一つ目のコネクション定義とは，DD（ra.xml）で最初に定義されているコネクション定義です。

稼働情報（運用管理コマンドを使用して出力する情報）

出力できません。Connector 1.5 仕様に準拠したリソースアダプタについての稼働情報を出力しようとした場合，出力内容は保証されません。

リソース枯渇監視情報

リソースアダプタに指定されているすべてのコネクション定義の情報が出力されます。

(6)　リソースアダプタの稼働ログを出力する場合の留意事項

Connector 1.5 仕様に準拠したリソースアダプタの稼働ログを出力する場合，出力ファイルのファイル名は，次の形式になります。

<リソースアダプタの表示名>_<コネクション定義の並び順>_<ログファイルの通番>.log

<コネクション定義の並び順>は，DD（ra.xml）内で，対応する<connection-definition>が出現した順番（1，2，...）に対応します。

3.16.7　アプリケーションサーバ独自の Connector 1.5 API 仕様

ここでは，Connector 1.5 で仕様のインタフェースに対して，アプリケーションサーバ独自で規定した仕様について説明します。

(1)　javax.resource.spi.endpoint.MessageEndpointFactory インタフェース

アプリケーションサーバでは，二つのメソッドの仕様を規定しています。

(a)　createEndpoint メソッド

形式

```
public MessageEndpoint createEndpoint(XAResource xaResource) throws UnavailableException
```

アプリケーションサーバ独自の仕様

- EJB を含む J2EE アプリケーションの停止処理を開始したあとでこのメソッドが呼び出された場合は，javax.resource.spi.UnavailableException がスローされます。
- このメソッドの中で Message-driven Bean のインスタンス取得処理が実行されます。Message-driven Bean のインスタンスプールの上限に達していた場合は，Message-driven Bean のインスタンスを取得できるか，J2EE アプリケーションが停止するまで，メソッドがリターンされません。

例外

アプリケーションサーバで規定した例外の動作について示します。

例外	例外が発生するタイミング
javax.resource.spi.UnavailableException	• EJB を含む J2EE アプリケーションの停止処理を開始したあとでこのメソッドが呼び出した場合 • Message-driven Bean のインスタンスの取得に失敗した場合

(b)　isDeliveryTransacted メソッド

形式

```
public boolean isDeliveryTransacted(Method method) throws NoSuchMethodException
```

アプリケーションサーバ独自の仕様

J2EE サーバ用ユーザプロパティファイルの ejbserver.distributedtx.XATransaction.enabled プロパティに true を指定すると，引数で渡されたメソッドのトランザクション属性に CMT の Required が指定されている場合に，true を返却されます。

(2)　javax.resource.spi.endpoint.MessageEndpoint インタフェース

アプリケーションサーバでは，三つのメソッドの仕様を規定しています。

(a)　beforeDelivery メソッド

形式

```
public void beforeDelivery(Method method) throws NoSuchMethodException, ResourceException
```

アプリケーションサーバ独自の仕様

例外について規定しています。

例外

アプリケーションサーバで規定した例外の動作について示します。

例外	例外が発生するタイミング
java.lang.IllegalStateException	• EJB を含む J2EE アプリケーションの停止処理を開始したあとでメッセージリスナのメソッドを呼び出した場合 • release()メソッドがすでに呼び出されている場合
javax.resource.spi.IllegalStateException	• 不正な順序で呼び出した場合（beforeDelivery メソッド呼び出し後に再度 beforeDelivery メソッドを呼び出した場合。またはメッセージリスナのメソッドの呼び出し後に beforeDelivery メソッドを呼び出した場合）
javax.resource.spi.UnavailableException	• 予期しない例外が発生した場合

(b)　afterDelivery メソッド

形式

```
public void afterDelivery() throws ResourceException
```

アプリケーションサーバ独自の仕様

例外について規定しています。

例外

アプリケーションサーバで規定した例外の動作について示します。

例外	例外が発生するタイミング
java.lang.IllegalStateException	• EJB を含む J2EE アプリケーションの停止処理を開始したあとでメッセージリスナのメソッドを呼び出した場合 • release()メソッドがすでに呼び出されている場合
javax.resource.spi.IllegalStateException	• 不正な順序で呼び出した場合（Option B※のメッセージリスナのメソッド呼び出し後以外のタイミングで afterDelivery()を呼び出した場合）
javax.resource.spi.UnavailableException	• 予期しない例外が発生した場合

注※　Connector 1.5 仕様に記述されているメッセージ配送オプションです。

(c)　release メソッド

形式

```
public void release()
```

アプリケーションサーバ独自の仕様

このメソッドを呼び出すと，エンドポイントに対応づけられた Message-driven Bean のインスタンスが解放され，インスタンスプールに戻ります。

エンドポイントを使用し終わったら，必ずこのメソッドを呼び出して，Message-driven Bean をインスタンスプールに戻すように実装してください。

例外

アプリケーションサーバで規定した例外の動作について示します。

例外	例外が発生するタイミング
java.lang.IllegalStateException	• EJB を含む J2EE アプリケーションの停止処理を開始したあとでメッセージリスナのメソッドを呼び出した場合 • release()メソッドがすでに呼び出されている場合

(3)　メッセージリスナのメソッド

インタフェースはメッセージリスナのインタフェースで定義されます。ここでは，アプリケーションサーバで規定した例外の動作について示します。

例外	例外が発生するタイミング
メッセージリスナのインタフェースに指定された例外	• Message-driven Bean の実行時（システム例外が発生した時以外）
javax.ejb.EJBException	• Message-driven Bean の実行時（システム例外が発生した時）
java.lang.IllegalStateException	• EJB を含む J2EE アプリケーションの停止処理を開始したあとでメッセージリスナのメソッドを呼び出した場合 • release()メソッドがすでに呼び出されている場合

例外	例外が発生するタイミング
java.lang.IllegalStateException	• 不正な順序で呼び出した場合（Option B※のメッセージ配送で，メッセージリスナのメソッドを呼び出したあと，続けてメッセージリスナのメソッドを呼び出した場合）

注※　Connector 1.5 仕様に記述されているメッセージ配送オプションです。

3.16.8　Connector 1.5 仕様に準拠したリソースアダプタを使用する場合の設定

ここでは，Connector 1.5 仕様に準拠したリソースアダプタを使用する場合の J2EE アプリケーションとリソースアダプタの設定について説明します。

設定は，それぞれアプリケーション統合属性ファイル※と Connector 属性ファイルに指定します。

注※

この項の説明は，すべてアプリケーション統合属性ファイルを使用する場合の説明です。cjgetappprop コマンドで Message-driven Bean 属性ファイルを取得した場合は，Message-driven Bean 属性ファイルに指定できます。Message-driven Bean 属性ファイルを使用する場合は，「アプリケーション統合属性ファイルの<hitachi-connector-property>タグ下」を「Message-driven Bean 属性ファイル」に読み替えてください。

(1)　設定できる項目

Connector 1.5 仕様に準拠したリソースアダプタを使用する場合に設定できる項目について説明します。

(a)　J2EE アプリケーションの設定

J2EE アプリケーションに設定できる内容を次の表に示します。

表 3-67　J2EE アプリケーションの設定（Connector 1.5 仕様に準拠するリソースアダプタを使用する場合）

機能	項目	対象	設定内容
管理対象オブジェクトのルックアップ	管理対象オブジェクトのルックアップで使用する情報	WAR，Session Bean，Entity Bean，Message-driven Bean	<resource-env-ref>タグ下の次のタグに設定します。 • <resource-env-ref-name>タグに，ルックアップで使用する名前を指定します。 • <resource-env-ref-type>タグに，管理対象オブジェクトの型を指定します。 • <linked-adminobject>－<resourceadapter-name>タグに，管理対象オブジェクトが含まれるリソースアダプタの表示名を指定します。 • <linked-adminobject>－<adminobject-name>タグに，管理対象オブジェクトのオブジェクト名を指定します。

機能	項目	対象	設定内容
メッセージインフロー※	リソースアダプタとの対応づけ	Message-driven Bean	<message-ref>－<connection-destination>タグ下の<resource-adapter>タグに，対応づけるリソースアダプタの表示名を指定します。
	Message-driven Bean が使用するインタフェース	Message-driven Bean	<messaging-type>タグに，Message-driven Bean が実装するインタフェースを指定します。
	ActivationSpec の設定	Message-driven Bean	<activation-config>－<activation-config-property>タグ下の<activation-config-property-name>タグおよび<activation-config-property-value>タグに，ActivationSpec に設定するプロパティを指定します。

注※　設定値の一部は，使用するリソースアダプタの設定値と合わせる必要があります。

(b)　リソースアダプタの設定

リソースアダプタのプロパティとして設定する項目のうち，Connector 1.5 仕様に準拠したリソースアダプタで設定できる主な項目と，その項目の設定方法について説明します。なお，ここでは，Connector 1.5 仕様に準拠したリソース固有の内容を説明します。

Connector 1.5 仕様に準拠したリソースアダプタのプロパティとして設定できる内容を次の表に示します。

表 3-68　Connector 1.5 仕様に準拠したリソースアダプタのプロパティ定義で設定できる内容

分類	項目	設定内容
管理対象オブジェクト	管理対象オブジェクト名※1	<adminobject>－<adminobject-name>タグ※2 にルックアップで使用するオブジェクト名を指定します。管理対象オブジェクトをルックアップの対象にする場合は必ず指定してください。オブジェクト名には，連続したアンダースコア"__"を含めてはいけません。 なお，管理対象オブジェクトをルックアップする場合には，J2EE アプリケーションの設定も必要です。「(a)　J2EE アプリケーションの設定」を参照してください。
	インタフェース	<adminobject>－<adminobject-interface>タグに使用するインタフェースを指定します。javax.jms.Queue，javax.jms.Topic などを指定します。
	実装クラスとプロパティ	<adminobject-class>タグに実装クラスを指定します。 <config-property>タグ下に管理対象オブジェクトのプロパティを設定します。

注※1　メッセージインフローで使用する管理対象オブジェクトの場合，指定する必要はありません。

注※2　DD（ra.xml）には存在しないタグです。

(2)　管理対象オブジェクトの設定

メッセージインフローで使用する管理対象オブジェクトの設定は，Connector 属性ファイルに記述します。

管理対象オブジェクトに設定する内容は，リソースアダプタに依存します。使用するリソースアダプタのドキュメントを参照してください。

管理対象オブジェクトは，Message-driven Bean と対応づけて使用します。対応づけについては，「(5) ActivationSpec の設定」を参照してください。

(3)　Message-driven Bean とリソースアダプタの対応づけの設定

Message-driven Bean とリソースアダプタの対応づけは，アプリケーション統合属性ファイルの<resource-adapter>タグに指定します。リソースアダプタの表示名を指定してください。

<resource-adapter>の階層を次に示します。

```
<hitachi-message-bean-property>
  <message-ref>
    <connection-destination>
      <resource-adapter>
```

これらのタグは，DD（ejb-jar.xml）には存在しない要素です。ただし，<hitachi-message-bean-property>は DD の<message-driven>に対応します。

リソースアダプタとの対応づけは，Message-driven Bean を含む J2EE アプリケーションを開始したときに実行されます。なお，次の場合には，Message-driven Bean を含む J2EE アプリケーションの開始が失敗します。この場合は，KDJE42359-E のメッセージが出力されます。

- <resource-adapter>タグにリソースアダプタの表示名が指定されていない場合
- <resource-adapter>タグに指定した表示名のリソースアダプタが開始されていない場合
- <resource-adapter>タグに指定した表示名のリソースアダプタが見つからない場合

Message-driven Bean は，管理対象オブジェクトとも対応づける必要があります。対応づけについては，「(5)　ActivationSpec の設定」を参照してください。

参考

アプリケーションサーバでは，<message-driven>－<message-destination-link>タグの情報は指定できません。

(4)　Message-driven Bean が使用するインタフェースの設定

Message-driven Bean が使用するインタフェースは，アプリケーション統合属性ファイルの<messaging-type>タグに記述します。Message-driven Bean が実装するインタフェース名を指定してください。

<messaging-type>の階層を次に示します。

```
<hitachi-message-bean-property>
  <messaging-type>
```

<hitachi-message-bean-property>は DD（ejb-jar.xml）の<message-driven>に対応します。

Message-driven Bean が使用するインタフェースには，リソースアダプタがサポートしているインタフェースを指定する必要があります。リソースアダプタがサポートするインタフェースは，リソースアダプタの DD（ra.xml）の<messagelistener-type>タグに指定されています。このタグに指定されている以外

のインタフェースを指定した場合は，Message-driven Bean を含む J2EE アプリケーションの起動が失敗します。この場合は，KDJE43167-E のメッセージが出力されます。

リソースアダプタがサポートしているインタフェースは，サーバ管理コマンドの次のコマンドでも確認できます。これらのコマンドの引数に，-resname および-inbound を指定して実行してください。

- cjlistres コマンド
  詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjlistres（リソースの一覧表示)」を参照してください。
- cjlistrar コマンド
  詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjlistrar（リソースアダプタの一覧表示)」を参照してください。
- cjlistapp コマンド
  詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjlistapp（アプリケーションの一覧表示)」を参照してください。

なお，<messaging-type>タグの指定を省略した場合は，デフォルト値として javax.jms.MessageListener インタフェースが使用されます。

参考

javax.jms.MessageListener インタフェースは，EJB2.0 まで固有のメッセージリスナインタフェースとして使用されていたインタフェースです。

(5)　ActivationSpec の設定

ActivationSpec とは，Message-driven Bean を活性化するために必要な設定を持つ JavaBeans です。ActivationSpec には，リソースアダプタの設定に合わせて設定が必要なプロパティがあります。

(a)　ActivationSpec の設定の概要

ActivationSpec に設定する値は，アプリケーション統合属性ファイルの<activation-config>タグ下に記述します。プロパティ名とプロパティ値を指定してください。複数指定できます。

<activation-config>の階層を次に示します。

```
<hitachi-message-bean-property>
  <activation-config>
    <activation-config-property>
      <activation-config-property-name>
      <activation-config-property-value>
```

ActivationSpec についてのデフォルトの設定値は，リソースアダプタの DD（ra.xml）に設定できます。<resourceadapter>－<config-property>タグに設定した値が，デフォルト値として使用されます。デフォルト値を記載している場合にアプリケーション統合属性ファイルで ActivationSpec の値を設定したときには，<hitachi-message-bean-property>タグ下で指定した値で上書きされます。

ActivationSpec のプロパティとして必ず設定しなくてはいけないプロパティについては，「(b)　設定が必要なプロパティ」で説明します。

(b)　設定が必要なプロパティ

ActivationSpec の設定では，リソースアダプタの DD（ra.xml）の<activationspec>－<required-config-property>タグに指定されているプロパティの値を設定する必要があります。設定が必要なプロパティは，

サーバ管理コマンドの次のコマンドでも確認できます。これらのコマンドの引数に，-resname および-listenertype を指定して実行してください。

- cjlistres コマンド
- cjlistrar コマンド
- cjlistapp コマンド

このほか，ActivationSpec に設定できるプロパティはリソースアダプタに依存します。使用するリソースアダプタのドキュメントを参照してください。

(c)　JMS をサポートするメッセージリスナを使用する場合に設定が必要なプロパティ

JMS をサポートするメッセージリスナ（javax.jms.MessageListener インタフェースを使用するメッセージリスナ）を使用する場合，ActivationSpec に次のプロパティが必要です。<activation-config>タグ下に指定してください。

- destination
- destinationType

なお，アプリケーションサーバでは，これらのプロパティが指定されているかどうかのチェックは実施されません。プロパティ設定が適切かどうかは，リソースアダプタが提供する ActivationSpec#validate メソッドで確認してください。

参考

EJB 2.0 で<message-driven>タグ下の要素として指定していた次の項目は，EJB 2.1 以降では削除されています。

- <message-selector>
- <acknowledge-mode>
- <message-driven-destination>－<subscription-durability>

Connector 1.5 仕様では，これらの要素を<activation-config>下のプロパティとして指定できることが推奨されています。ただし，指定できるかどうかは使用するリソースアダプタに依存します。

JMS をサポートするメッセージリスナを使用する場合の ActivationSpec 仕様の詳細については，Connector 1.5 仕様のドキュメントを参照してください。

(d)　ActivationSpec の生成・設定で発生するエラー

ActivationSpec を生成，設定するときに，次の表に示す状態になった場合，エラーが発生します。エラーが発生すると，J2EE アプリケーションの開始は失敗します。

エラーが発生する場合と出力されるメッセージを次の表に示します。

表 3-69　エラーが発生する場合と出力されるメッセージの対応

エラーが発生する場合	出力されるメッセージ
ActivationSpec の実装クラスのインスタンス生成に失敗した場合	KDJE43168-E
ActivationSpec に指定したクラスが javax.resource.spi.ActivationSpec インタフェースを実装していなかった場合	KDJE43172-E

エラーが発生する場合	出力されるメッセージ
<activation-config-property-name>タグに対応するプロパティの setter メソッドが，ActivationSpec にない場合	KDJE43169-E
ActivationSpec に各プロパティを設定したあとで，設定が適切であるかをチェックするメソッド(ActivationSpec#validate メソッド)の呼び出しで例外がスローされた場合	KDJE43170-E
リソースアダプタの DD (ra.xml) で指定したプロパティ<required-config-property>タグに対応する設定項目が，アプリケーション統合属性ファイルの<activation-config>タグとして記述されていない場合	KDJE43171-E
ActivationSpec の setResourceAdapter メソッドで例外がスローされた場合※	KDJE43173-E

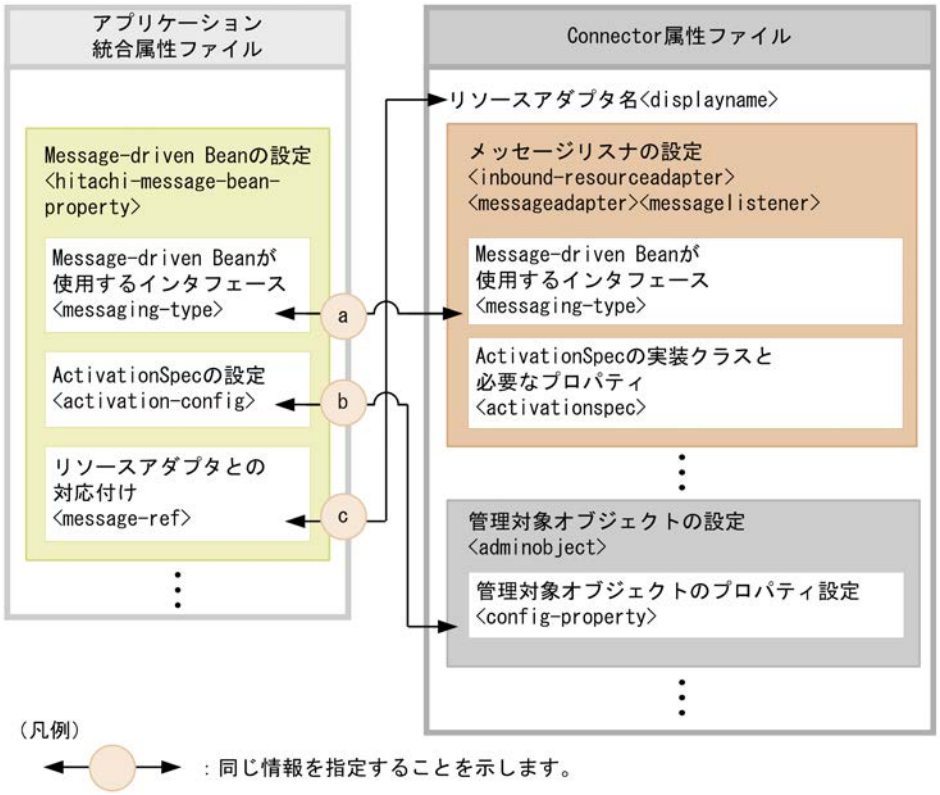
注※　setResourceAdapter メソッドは javax.resource.spi.ResourceAdapterAssociation インタフェースで定義するメソッドです。javax.resource.spi.ResourceAdapterAssociation インタフェースは，javax.resource.spi.ActivationSpec インタフェースのスーパーインタフェースです。

3.16.9　属性ファイルの指定例

ここでは，メッセージインフローを実行する場合の属性ファイルの指定方法について，例を使用して説明します。

Connector 属性ファイルとアプリケーション統合属性ファイルでは，次の図に示す個所の設定を合わせる必要があります。

図 3-54　Connector 属性ファイルとアプリケーション統合属性ファイルで同じ値を指定する個所



それぞれの指定内容は次のとおりです。

- a には，Message-driven Bean が使用するインタフェースを指定します。
- b には，ActivationSpec と管理対象オブジェクトの対応づけを指定します。
- c には，Message-driven Bean とリソースアダプタの対応づけを指定します。

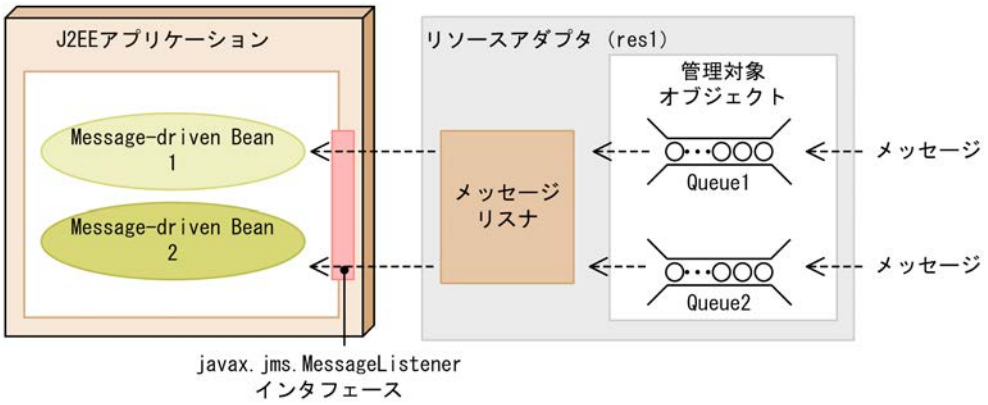
以降，a～c に指定する内容を中心に説明します。

(1)　javax.jms.MessageListener インタフェースを使用した Message-driven Bean およびリソースアダプタの場合

メッセージリスナのインタフェースとして javax.jms.MessageListener インタフェースを使用する場合の属性ファイルの指定例を示します。

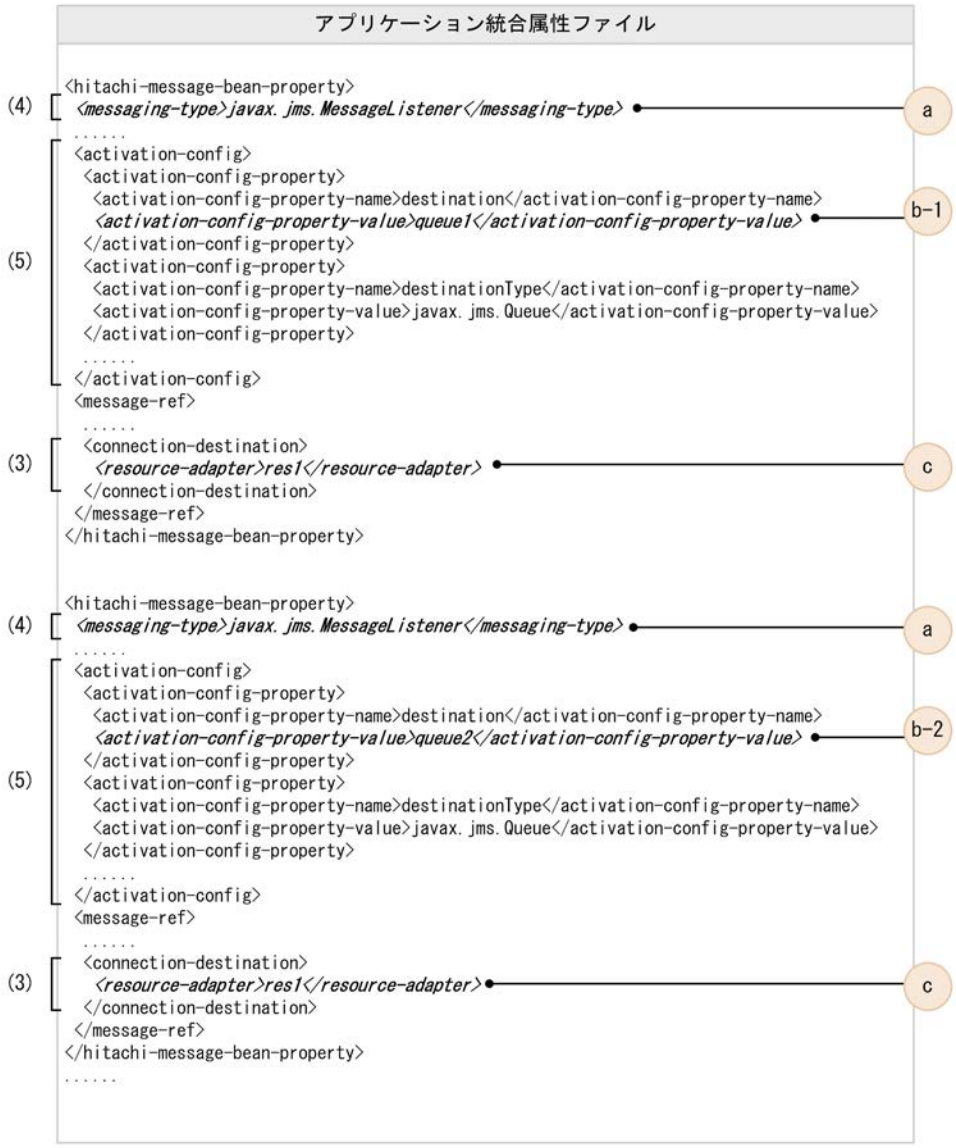
ここでは，次の図に示す構成を例にして説明します。JMS に対応した二つの Message-driven Bean が，それぞれ別の管理オブジェクト（javax.jms.Queue）からメッセージを受信する例です。

図 3-55　javax.jms.MessageListener インタフェースを使用した Message-driven Bean およびリソースアダプタの構成例



アプリケーション統合属性ファイルの指定例を次に示します。

図 3-56　アプリケーション統合属性ファイルの指定例（javax.jms.MessageListener インタフェースを使用する場合）



図の説明：

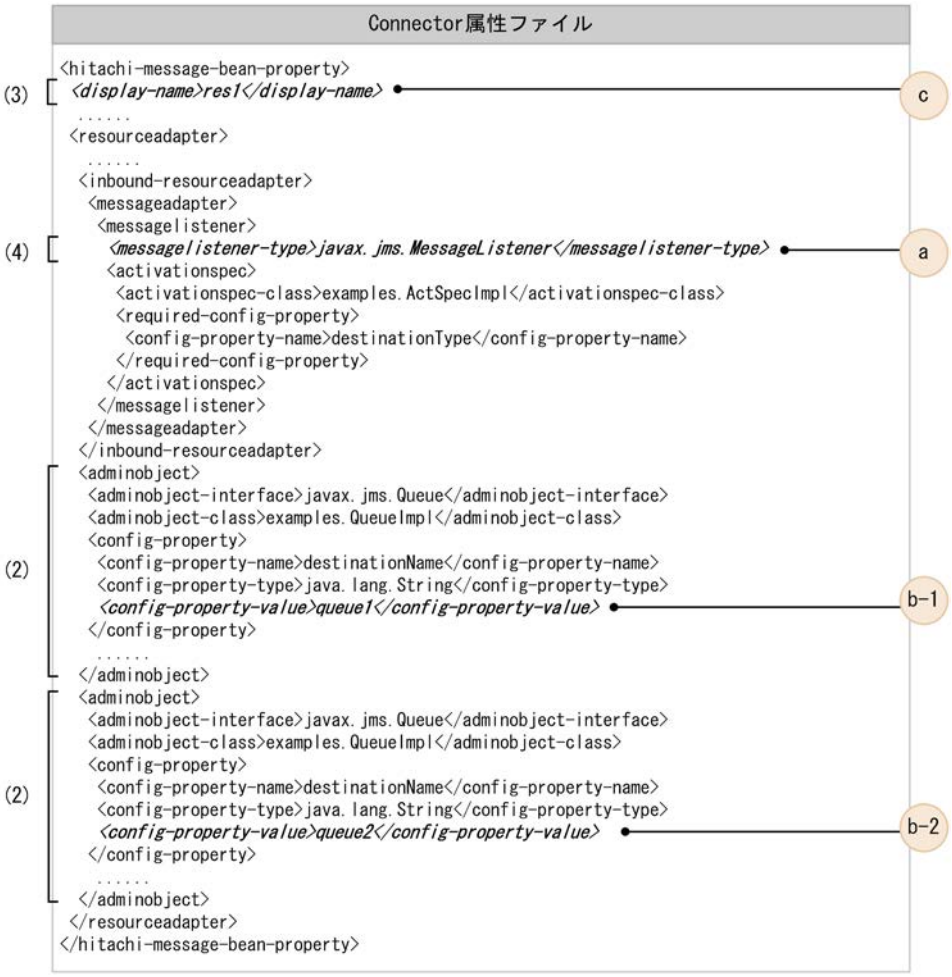
図中の(3)～(5)は，「3.16.8　Connector 1.5 仕様に準拠したリソースアダプタを使用する場合の設定」での説明の項番と対応しています。また，a，b-1，b-2，c は，それぞれ次の設定を示します。

- a：Message-driven Bean が使用するインタフェース
- b-1，b-2：ActivationSpec と管理対象オブジェクトの対応づけ
- c：Message-driven Bean とリソースアダプタの対応づけ

なお，a，b-1，b-2，c の設定値は，図 3-56 と対応しています。

Connector 属性ファイルの指定例を次の図に示します。

図 3-57　Connector 属性ファイルの指定例（javax.jms.MessageListener インタフェースを使用する場合）



図の説明：

図中の(2)～(4)は，「3.16.8　Connector 1.5 仕様に準拠したリソースアダプタを使用する場合の設定」での説明の項番と対応しています。また，a，b-1，b-2，c は，それぞれ次の設定を示します。

- a：Message-driven Bean が使用するインタフェース
- b-1，b-2：ActivationSpec と管理対象オブジェクトの対応づけ
- c：Message-driven Bean とリソースアダプタの対応づけ

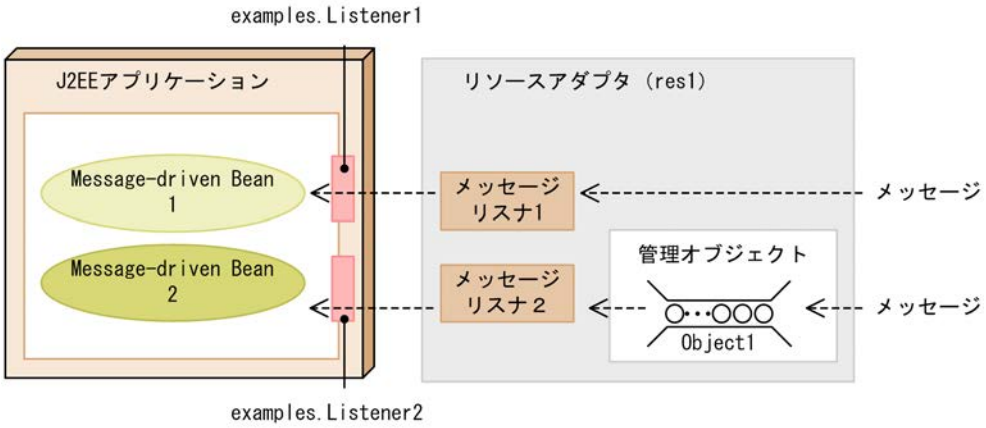
なお，a，b-1，b-2，c の設定値は，図 3-57 と対応しています。

(2)　任意のメッセージリスナインタフェースを使用した Message-driven Bean およびリソースアダプタの場合

メッセージリスナのインタフェースとして任意のインタフェースを使用する場合の属性ファイルの指定例を示します。

ここでは，次の図に示す構成を例にして説明します。

図 3-58　任意のメッセージリスナインタフェースを使用した Message-driven Bean およびリソースアダプタの構成例



この例では，リソースアダプタが次に示す二つの独自のインタフェースに対応しています。

- examples.Listener1 は管理対象オブジェクトと独立して使用するメッセージリスナインタフェースです。
- examples.Listener2 は管理対象オブジェクトと関連づけて使用するメッセージリスナインタフェースです。

アプリケーション統合属性ファイルの指定例を次の図に示します。

図 3-59　アプリケーション統合属性ファイルの指定例（任意のメッセージリスナインタフェースを使用する場合）

図の説明：

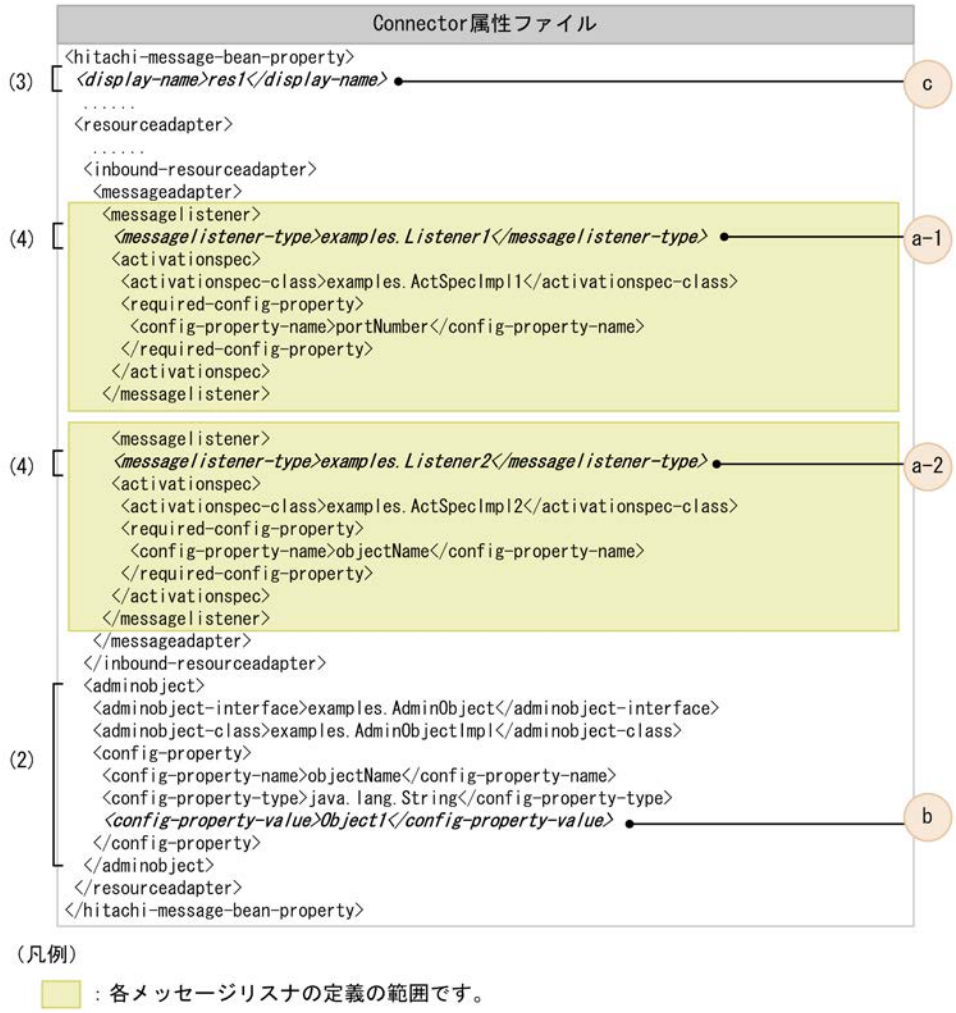
図中の(3)～(5)は，「3.16.8　Connector 1.5 仕様に準拠したリソースアダプタを使用する場合の設定」での説明の項番と対応しています。また，a-1，a-2，b，c は，それぞれ次の設定を示します。

- a-1，a-2：Message-driven Bean が使用するインタフェース
- b：ActivationSpec と管理対象オブジェクトの対応づけ
- c：Message-driven Bean とリソースアダプタの対応づけ

なお，a-1，a-2，b，c の設定値は，図 3-59 と対応しています。

Connector 属性ファイルの指定例を次の図に示します。

図 3-60　Connector 属性ファイルの指定例（任意のメッセージリスナインタフェースを使用する場合）



図の説明：

図中の(2)～(4)は，「3.16.8　Connector 1.5 仕様に準拠したリソースアダプタを使用する場合の設定」での説明の項番と対応しています。また，a-1，a-2，b，c は，それぞれ次の設定を示します。

- a-1，a-2：Message-driven Bean が使用するインタフェース
- b：ActivationSpec と管理対象オブジェクトの対応づけ
- c：Message-driven Bean とリソースアダプタの対応づけ

なお，a-1，a-2，b，c の設定値は，図 3-60 と対応しています。

3.16.10　Connector 1.5 仕様に準拠したリソースアダプタを使用する場合の注意事項

Connector 1.5 仕様に準拠したリソースアダプタを使用する場合の注意事項について説明します。注意事項はリソースアダプタの使用方法によって異なります。

(1)　共通の注意事項

- Connector 1.5 仕様に準拠するリソースアダプタに関する稼働情報は，運用管理ポータルまたは mngsvrutil コマンドを使用しても取得できません。

- 運用管理ポータルまたは mngsvrutil コマンドを使用して，Connector 1.5 仕様に準拠するリソースアダプタのステータス情報を取得した場合，リソースアダプタが稼働していても，停止を示す情報が返されます。
- Connector 1.5 仕様に準拠するリソースアダプタに対して cmx_stop_resource コマンドを実行すると，コマンドは常に成功しても，リソースアダプタは停止されません。

(2)　J2EE リソースアダプタとしてデプロイして使用する場合（J2EE アプリケーションに含めないで使用する場合）

管理対象オブジェクトをアノテーションで指定する場合は, コンテナ拡張ライブラリに管理対象オブジェクトのインタフェースを追加する必要があります。

コンテナ拡張ライブラリについては,「14.　コンテナ拡張ライブラリ」を参照してください。

(3)　J2EE アプリケーションに含めて使用する場合

- リロード機能を使用する場合は，次のクラス（インタフェース）をコンテナ拡張ライブラリに追加する必要があります。
  - javax.resource.spi.ActivationSpec インタフェースの実装クラス
  - リソースアダプタがサポートするメッセージリスナのインタフェース
- 管理対象オブジェクトをアノテーションで指定する場合は，コンテナ拡張ライブラリに管理対象オブジェクトのインタフェースを追加する必要があります。
- Inbound のリソースアダプタを使用した場合，グローバルトランザクションは使用できません。

コンテナ拡張ライブラリについては,「14.　コンテナ拡張ライブラリ」を参照してください。

3.17　クラスタコネクションプール機能

この節では，クラスタコネクションプール機能について説明します。

この節の構成を次の表に示します。

表 3-70　この節の構成（クラスタコネクションプール機能）

分類	タイトル	参照先
解説	Oracle RAC を使用した Oracle への接続	3.17.1
	クラスタコネクションプールの概要	3.17.2
	使用するリソースアダプタ	3.17.3
	クラスタコネクションプールの動作	3.17.4
	手動によるコネクションプールの停止・開始の流れ	3.17.5
設定	コネクションプールをクラスタ化するために必要な設定	3.17.6

注　「実装」および「運用」について，この機能固有の説明はありません。

クラスタコネクションプール機能は，データベースをクラスタ構成にしているシステムの場合に，最適な動作をするための機能です。Oracle RAC を使用した Oracle への接続の場合に使用できます。クラスタコネクションプール機能を使用することで，障害発生時やメンテナンス時の可用性の低下を防ぐことができます。クラスタコネクションプール機能の使用時に，データベースノードに障害が発生した場合，およびデータベースノードのメンテナンスを行う場合の動作について説明します。

- **データベースノードに障害が発生した場合**

  OS，ハード，ソフト障害など，コネクションが取得できない状況になった場合，障害が発生しているデータベースノードに接続しているコネクションプールを自動で一時停止できます（自動一時停止機能）。J2EE アプリケーションからリソースアダプタにコネクションの取得要求が出された場合でも，一時停止されたコネクションプールにはコネクションの取得要求が出されないため，TCP/IP のタイムアウトまで処理が中断することはありません。これによって，J2EE アプリケーションは，ほかの正常なデータベースノードに接続しているコネクションプールからコネクションを取得し，業務を継続できます。

  また，データベースノードの障害が回復した時に，自動でコネクションプールを再開できます（自動再開機能）。コネクションプールが再開されると，自動的に回復したデータベースノードから再びアクセスされるため，データベースノード回復時に cjclearpool コマンドを実行して，コネクションプールを削除する必要はありません。

- **データベースノードのメンテナンスを行う場合**

  データベースノードのメンテナンスを行う場合，コマンドを使用して任意のタイミングでメンバコネクションプールを一時停止できます（手動一時停止機能）。これによって，そのデータベースノードを切り離して，メンテナンスを行えます。

  また，メンテナンスが終了して再開する時に，コマンドを使用して任意のタイミングでコネクションプールを再開できます（手動再開機能）。

なお，クラスタコネクションプール機能では，コネクション取得時に障害を検知した場合，正常なほかのメンバリソースアダプタからコネクションを取得します。その際，アプリケーションでエラーは発生しません。

この節では，Oracle RAC 機能を使用してクラスタ化した Oracle との接続方法，およびコネクションプールをクラスタ化する場合のコネクションプール（クラスタコネクションプール）の特徴，および機能について説明します。

3.17.1　Oracle RAC を使用した Oracle への接続

Oracle RAC を使用した Oracle への接続方法は，Oracle のバージョン，または負荷分散に使用する機能によって異なります。なお，接続できるトランザクションの種類はローカルトランザクションです。

Oracle のバージョン，負荷分散に使用する機能および使用する RAR ファイルの対応については，「3.6.6 Oracle と接続する場合の前提条件と注意事項」を参照してください。

負荷分散に使用する機能ごとに，Oracle への接続方法について説明します。

(1)　アプリケーションサーバの機能を使用した接続

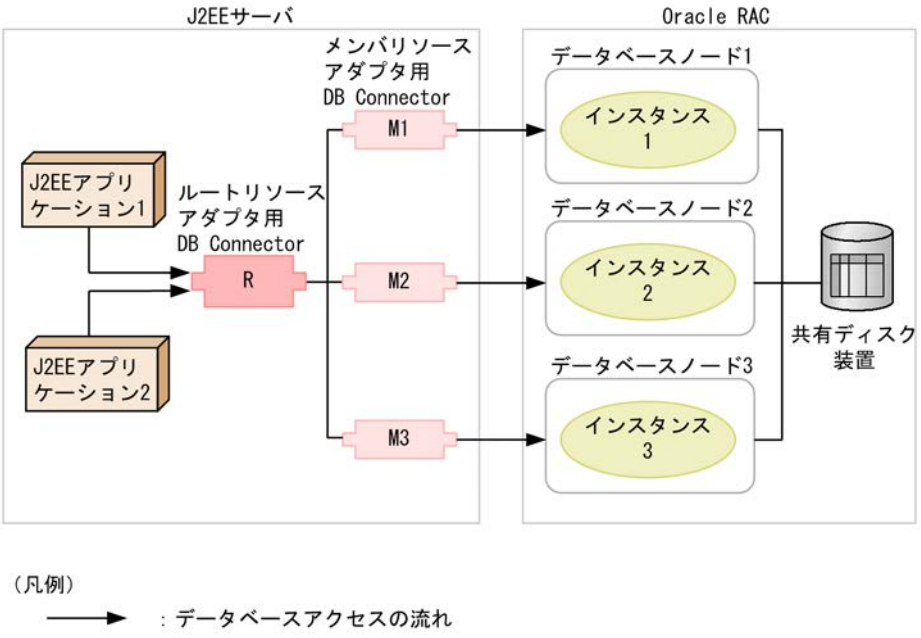
アプリケーションサーバのクラスタコネクションプール機能を使用して，Oracle RAC に接続します。アプリケーションサーバがデータベースアクセスの負荷を分散します。

クラスタコネクションプール機能の詳細については，3.17.2 以降の説明を参照してください。

(a)　負荷分散処理の流れと設定

クラスタコネクションプール機能を使用した場合の負荷分散処理の流れと設定を次の図に示します。

図 3-61　クラスタコネクションプール機能を使用した接続



この図では，3 台構成の Oracle RAC システムで，データベースノード 1 にはインスタンス 1，データベースノード 2 にはインスタンス 2，データベースノード 3 にはインスタンス 3 があります。データベースに接続するための設定を次に示します。

1. インスタンス 1，2，3 に対応したメンバリソースアダプタ用 DB Connector M1，M2，M3 を生成します。また，メンバリソースアダプタに振り分ける機能を持つルートリソースアダプタ用 DB Connector R も生成します。

2. J2EE アプリケーションはルートリソースアダプタ用 DB Connector R に関連づけます。

この設定によって，J2EE アプリケーション 1，2 からのデータベースアクセスはデータベースノード 1，2，3 に分散されます。

(b)　データベース障害発生時と回復時の動作

データベース障害が発生した場合，アプリケーションサーバが障害を検知します。障害が発生したデータベースに対応するメンバリソースアダプタが閉塞され，残っているインスタンスで処理が続行されます。

データベース障害が回復するとアプリケーションサーバは自動で閉塞を解除します。また，手動でも閉塞を解除できます。

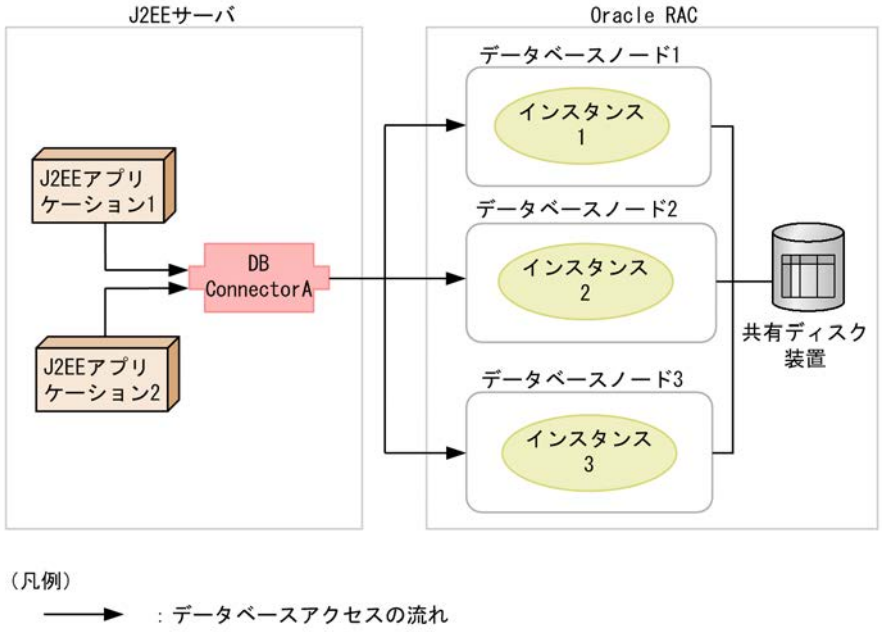
(2)　Oracle の機能を使用した接続

DB Connector から Oracle RAC に接続し，Oracle RAC の機能でデータベースアクセスの負荷を分散します。

(a)　負荷分散処理の流れと設定

Oracle RAC の機能を使用した場合の負荷分散処理の流れと設定を次の図に示します。

図 3-62　Oracle RAC の機能を使用した接続



この図では，3 台構成の Oracle RAC システムで，データベースノード 1 にはインスタンス 1，データベースノード 2 にはインスタンス 2，データベースノード 3 にはインスタンス 3 があります。データベースに接続するための設定について説明します。

1. それぞれのインスタンスに分散して接続する DB Connector A を生成します。
2. DB Connector A は，グローバルデータベース名またはサービス名を使用するように設定します。
3. J2EE アプリケーション 1，および 2 を DB Connector A に関連づけます。

この設定によって，J2EE アプリケーション 1，および 2 からのデータベースアクセスはデータベースノード 1，2，3 に分散されます。

(b)　データベース障害発生時と回復時の動作

データベース障害が発生した場合，Oracle RAC 機能によって障害が発生したインスタンスが切り離され，残っているインスタンスで処理が続行されます。

アプリケーションサーバのコネクションプールを使用している場合，データベース回復時には次のどちらかの操作を実行してください。コネクションプールがクリアされ，これ以降のアクセスが正常に分散されます。

- cjclearpool コマンドを実行する。
- J2EE サーバを再起動する。

3.17.2　クラスタコネクションプールの概要

クラスタ化されたコネクションプールのことをクラスタコネクションプールといいます。ここでは，クラスタコネクションプールの構成，およびクラスタコネクションプールで使用できる機能について説明します。

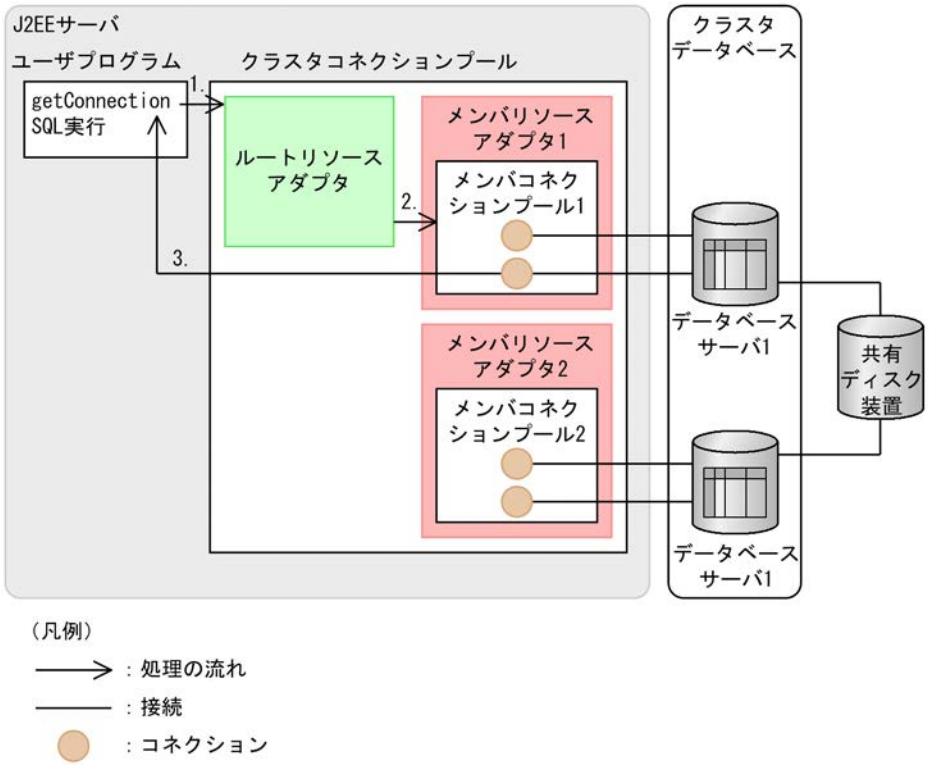
(1)　クラスタコネクションプールの構成

クラスタコネクションプールは，各データベースノードと接続するメンバリソースアダプタと，それら複数のメンバリソースアダプタを束ねるルートリソースアダプタで構成されています。ルートリソースアダプタとメンバリソースアダプタについて説明します。

- ルートリソースアダプタ
  クラスタ化したコネクションプール（クラスタコネクションプール）を使用するときに，J2EE アプリケーションからアクセスされるリソースアダプタです。ルートリソースアダプタは，メンバリソースアダプタを束ねる役割を持ちます。ルートリソースアダプタでは，ルートリソースアダプタに対する処理要求をメンバリソースアダプタに振り分けます。なお，ルートリソースアダプタは，コネクションプールを持ちません。
- メンバリソースアダプタ
  クラスタ化されている個々のデータベースノードに接続するリソースアダプタです。メンバリソースアダプタは，必ずルートリソースアダプタを経由してアクセスされます。メンバリソースアダプタのコネクションプールのことを，メンバコネクションプールといいます。

コネクションプールをクラスタ化しているときのコネクション取得時の処理の流れを次の図に示します。

図 3-63　コネクション取得時の処理の流れ



1. J2EE アプリケーションは，ルートリソースアダプタに対しコネクション取得要求をする。
2. ルートリソースアダプタは，メンバコネクションプールを一つ選択し，コネクション取得要求を出す。
3. メンバコネクションプールからコネクションが選択され，J2EE アプリケーションに返す。

(2)　前提条件

クラスタコネクションプール機能を利用できるデータベースは，Oracle RAC 機能を使用している場合だけです。使用できる JDBC ドライバは，Oracle JDBC Thin Driver です。

(3)　データベース接続で使用できる J2EE コンポーネントおよび機能

クラスタコネクションプール機能を使用している場合に，データベース接続で使用できる J2EE コンポーネントおよび機能を次の表に示します。

表 3-71　データベース接続で使用できる J2EE コンポーネントおよび機能（クラスタコネクションプール機能）

項目		Oracle（クラスタコネクションプール機能を使用した場合）
J2EE コンポーネント	Servlet/JSP	○
	Stateless Session Bean	○
	Stateful Session Bean	○
	Singleton Session Bean	○
	Entity Bean（BMP）	○

項目		Oracle（クラスタコネクションプール機能を使用した場合）
J2EE コンポーネント	Entity Bean（CMP1.1）	×
	Entity Bean（CMP2.0）	×
	Message-driven Bean	×
使用できる機能	コネクションプーリング	○
	コネクションプールのウォーミングアップ	○
	コネクションプールの情報表示（cjlistpool コマンド）	○
	コネクションプールのクリア	○
	リソースへの接続テスト	○
	コネクションの障害検知	○
	ステートメントプーリング	○
	ステートメントキャンセル	○※
	ステートメント setQueryTimeout メソッド	○※
	コネクション ID の PRF トレース出力	○
	障害調査用 SQL の出力	○

（凡例）○：使用できる　×：使用できない

注※　Oracle に接続する場合の注意事項があります。注意事項については,「3.6.6　Oracle と接続する場合の前提条件と注意事項」を参照してください。

! 注意事項

リソースアダプタを使用する場合，J2EE アプリケーションからリソースアダプタへのリファレンスを解決しておく必要があります。リソースアダプタを使用している J2EE アプリケーションをカスタマイズするときに，J2EE アプリケーションからリソースアダプタへのリファレンスを解決しておいてください。

(4)　使用できるリソース接続とトランザクション管理の機能

ルートリソースアダプタ，およびメンバリソースアダプタで使用できる機能については,「3.3.4　リソースアダプタの機能」を参照してください。

3.17.3　使用するリソースアダプタ

ルートリソースアダプタおよびメンバリソースアダプタで実行できるコマンド，設定の概要，および注意事項について説明します。

(1)　実行できるコマンド

ルートリソースアダプタおよびメンバリソースアダプタごとの，コマンドの実行可否を次の表に示します。なお，これ以外のコマンドについてはどちらのリソースアダプタでも実行できます。

表 3-72　ルートリソースアダプタおよびメンバリソースアダプタのコマンドの実行可否

コマンド	リソースアダプタの種類	
	ルートリソースアダプタ	メンバリソースアダプタ
cjstartrar	○	○
cjstoprar	○	○
cjtestres	○	○
cjlistrar	○	○
cjclearpool	×	○
cjlistpool	×	○
cjsuspendpool	×	○
cjresumepool	×	○

（凡例）○：実行できる　　×：実行できない

(2)　DB Connector の設定

ルートリソースアダプタおよびメンバリソースアダプタの DB Connector で設定する項目を次の表に示します。

表 3-73　ルートリソースアダプタおよびメンバリソースアダプタの DB Connector で設定する項目

設定項目	リソースアダプタの種類	
	ルートリソースアダプタ	メンバリソースアダプタ
トランザクションサポートレベル	×	○※1
ログ取得の可否	○	○
データベース接続情報	－	○
DB Connector 固有の設定（ステートメントプールなど）	－	○
セキュリティ情報（ユーザ名，パスワード）	×	○※2
コネクションプールサイズ	×	○
クラスタコネクションプール固有の設定	○	－

（凡例）○：設定が必要　×：設定は不要　－：設定項目がない

注※1　一つのクラスタコネクションプールを構成するメンバリソースアダプタのトランザクションサポートレベルは，すべて同じにする必要があります。

注※2　一つのクラスタコネクションプールを構成するメンバリソースアダプタのユーザ名は，すべて同じにする必要があります。

DB Connector の設定の詳細については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「4.2　データベースと接続するための設定」，またはマニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1　Connector 属性ファイル」を参照してください。

(3)　ルートリソースアダプタについての注意事項

- cjclearpool コマンドは実行できません。実行した場合は，メッセージが出力され，コマンドが異常終了します。クラスタコネクションプール内のコネクションを cjclearpool コマンドでクリアする場合は，メンバリソースアダプタに対してコマンドを実行してください。
- cjlistpool コマンドは実行できません。実行した場合は，メッセージが出力され，コマンドが異常終了します。クラスタコネクションプール内のコネクションプールの情報を表示する場合は，cjlistpool コマンドの-resname にメンバリソースアダプタの表示名を指定するか，-resall を指定して実行してください。
- 一つのクラスタコネクションプール内で，コンテナ管理でのサインオンとコンポーネント管理でのサインオンを混在して使用できません。コンポーネント管理でのサインオンを使用する場合は，ルートリソースアダプタに属するすべてのメンバリソースアダプタのユーザ名を空白にしてください。
- ルートリソースアダプタは，ルートリソースアダプタに属するすべてのメンバリソースアダプタが開始状態の場合に開始できます。これ以外の場合に，ルートリソースアダプタを開始すると，コンソールまたは画面にエラーメッセージが出力され，リソースアダプタの開始に失敗します。

(4)　メンバリソースアダプタについての注意事項

- メンバリソースアダプタでは，次の機能が前提となります。これらの機能は，デフォルトで有効となり，無効にすることはできません。
  - コネクションプーリング

    コネクションプールの最大値に 0 より大きい値を設定してください。0 以下の値を設定した場合，コネクションプールの最大値と最小値にデフォルト値が指定されたものとして動作します。
  - コネクション取得時のコネクションの障害検知

    コネクション取得時のコネクションの障害検知およびコネクション障害検知のタイムアウトは，設定値に関係なく有効となります。
  - コネクション枯渇時のコネクション取得待ち

    コネクション枯渇時のコネクション取得待ちは，設定値に関係なく有効となります。
  - loginTimeout

    loginTimeout のプロパティに 0 よりも大きい値を設定してください。0 以下の値を設定した場合はデフォルト値が指定されたものとして動作します。
- メンバリソースアダプタでは，次の機能を使用できません。これらの機能は，デフォルトで無効となり，有効にすることはできません。
  - コネクションの取得リトライ
  - J2EE リソースのユーザ指定名前空間
- メンバコネクションプール内のコネクションは，すべて同じデータベースノードに接続している必要があるため，データベースで接続先を変更する機能は使用しないでください。次にその機能の例を挙げます。なお，各機能を無効にする設定については，オラクルのマニュアルを参照してください。
  - クライアント・ロード・バランシング機能
  - 接続時フェイルオーバー機能
  - データベースサービス
  - リスナによる負荷分散機能
- メンバリソースアダプタにアクセスするときは，必ずルートリソースアダプタを経由します。そのため，次の個所には設定できません。

- J2EE アプリケーションのリソースリファレンス
- CMP Entity Bean のマッピング定義

- メンバリソースアダプタは，所属するルートリソースアダプタが停止状態の場合に停止できます。開始状態のときに，メンバリソースアダプタを停止すると，コンソールまたは画面にエラーメッセージが出力され，リソースアダプタの停止に失敗します。
- メンバコネクションプールの閉塞および一時停止時，コネクションを破棄するために，コネクションプールから未使用コネクションを取り除きます。この際，コネクションの破棄が完了していない時点でコネクションプールが再開されると，コネクションプール内のコネクションとコネクションプールから取り除いた未使用コネクションの総数が，コネクションプールのコネクション数の最大値を一時的に超える場合があります。

3.17.4　クラスタコネクションプールの動作

コネクションプールをクラスタ化した場合に使用できる機能と動作について説明します。クラスタコネクションプールでは次の機能を実行できます。

- コネクションプールの一時停止
- コネクションプールの再開

また，コネクションプールの状態，およびコネクションの取得要求を受けたときの，コネクションプールの選択方式についても説明します。

(1)　コネクションプールの一時停止

メンバコネクションプールを閉塞および一時停止できます。一時停止を実行すると，メンバコネクションプールが閉塞し一時停止します。

J2EE アプリケーションがルートリソースアダプタにコネクション取得要求をした場合，閉塞または一時停止したメンバコネクションプールにはコネクション取得要求は出されません。

次の場合にメンバコネクションプールを一時停止してください。

- データベースノードに障害が発生した場合
- データベースノードのメンテナンスをする場合

なお，一時停止の方法には次の 2 とおりがあります。

- 自動一時停止
- 手動一時停止

参考

一時停止処理では，J2EE アプリケーションで使用し終わったコネクションを破棄し，コネクションプール内のすべてのコネクションを破棄してから，コネクションプールを一時停止します。ネットワーク障害やデータベースノード障害が発生した場合，ネットワークのタイムアウトまで待ってからコネクションを破棄するため，閉塞してから一時停止するまでに時間が掛かることがあります。

(a)　自動一時停止

データベースノードの障害時にメンバコネクションプールを自動で一時停止できます。障害を検知すると，メンバコネクションプールは自動的に閉塞状態になり，そのあと一時停止します。

コネクションの取得時に次の現象が発生した場合，データベースノードに障害が発生したと判断し，メンバコネクションプールが自動的に一時停止します。

- コネクション取得時のコネクションの障害検知がタイムアウトした場合
- 物理コネクションの取得に失敗した場合
- 物理コネクションの取得がタイムアウトした場合
- コネクション管理スレッドが枯渇した場合

この機能は，デフォルトで有効となっています。有効/無効の設定は，ルートリソースアダプタのプロパティとして設定します。リソースアダプタの設定については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「5.4　リソースアダプタのプロパティ定義」を参照してください。

(b)　手動一時停止

データベースノードのメンテナンスをする場合などに，cjsuspendpool コマンドを実行することで，コネクションプールを手動で一時停止できます。手動一時停止は，メンバコネクションプールが開始状態，開始予約状態，自動一時停止状態，および自動一時停止予約状態の場合に実行できます。メンバコネクションプールが自動一時停止状態の場合に cjsuspendpool コマンドを実行すると，手動一時停止状態になります。これによって，自動一時停止後に自動再開させない運用ができます。手動での一時停止手順については，「3.17.5(2)　コネクションプールの一時停止」を参照してください。コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjsuspendpool（メンバコネクションプールの一時停止)」を参照してください。

! 注意事項

cjsuspendpool コマンドで手動一時停止したコネクションプールは，自動で再開されません。手動で再開してください。

(2)　コネクションプールの再開

一時停止したメンバコネクションプールを再開できます。J2EE アプリケーションからルートリソースアダプタにコネクション取得要求をした時に，再開したメンバコネクションプールには再びコネクションの取得要求が出されるようになります。

なお，コネクションプールを再開するには，未使用のコネクション管理スレッドの個数が，コネクションプールのコネクション数の最大数以上必要です。

次の場合にコネクションプールを再開してください。

- データベースノードの障害が回復した場合
- データベースノードのメンテナンスが終了した場合

再開の方法には次の 2 とおりがあります。

- 自動再開
- 手動再開

参考

- コネクションプールが停止されている場合に，未使用のコネクション管理スレッドの個数がコネクションプールのコネクション数の最大数になると，メッセージが出力されます。cjresumepool コマンドの実行に失敗した場合は，出力されたメッセージを確認してから再度コマンドを実行してください。
- コネクションプールのウォーミングアップ機能が有効な場合には，コネクションプールの設定で定義した最小値までコネクションをプールしてから，メンバコネクションプールを開始します。コネクションを

プーリングしているときにコネクションの生成に失敗した場合は，メンバコネクションプールは一時停止状態に戻ります。また，コネクションプールのウォーミングアップ機能が無効な場合は，すぐに開始状態になります。

(a)　自動再開

自動一時停止したメンバコネクションプールを自動で再開できます。

自動一時停止したメンバコネクションプールでは，データベースノードの状態をチェックするために，一定間隔で物理コネクションの取得要求を出します。このとき，コネクションの取得に成功すると，データベースノードが回復したと判断し，自動的に再開処理が行われます。また，自動一時停止したメンバコネクションプールを自動再開させない運用にするには，自動一時停止後に手動一時停止してください。再開時には，手動再開してください。

なお，ルートリソースアダプタが停止状態の場合，自動再開処理は行われません。ただし，メンバコネクションプールが自動再開中状態の場合にルートリソースアダプタを停止したときは，実行中の自動再開処理が継続されます。

この機能は，デフォルトで有効となっています。有効/無効の切り替えや，データベースノードの状態をチェックする間隔の設定は，ルートリソースアダプタのプロパティとして設定します。リソースアダプタの設定については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「5.4　リソースアダプタのプロパティ定義」を参照してください。

(b)　手動再開

自動一時停止または手動一時停止したコネクションプールを手動で再開できます。手動再開するには，cjresumepool コマンドを使用します。手動再開は，コネクションプールが次の状態の場合に実行できます。

- 自動一時停止状態
- 手動一時停止状態
- 自動一時停止予約状態
- 手動一時停止予約状態

手動での手動再開手順については，「3.17.5(3)　コネクションプールの再開」を参照してください。コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjresumepool（メンバコネクションプールの再開始）」を参照してください。

(3)　コネクションプールの状態

コネクションプールの状態は，メンバコネクションプールの場合だけ存在するものです。なお，コネクションプールの状態は，J2EE サーバやリソースアダプタの再起動後も維持されます。

メンバコネクションプールの状態は次の方法で確認できます。手動一時停止または手動再開をする場合は，コマンドを実行する前にコネクションプールの状態を確認してください。

- cjlistrar コマンド

  cjlistrar コマンドは，デプロイされているすべてのリソースアダプタについて，リソースアダプタ名とリソースアダプタの状態を標準出力に出力します。-clusterpool を指定した場合には，メンバコネクションプールの状態も表示できます。

  コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjlistrar（リソースアダプタの一覧表示）」を参照してください。

- cjlistpool コマンド

  cjlistpool コマンドは，コネクションプールの情報を表示するコマンドです。メンバリソースアダプタの場合は，メンバコネクションプールの状態も表示できます。メンバリソースアダプタのコネクション情報の表示例については，「3.15.4　コネクションプールの情報表示」を参照してください。

  コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjlistpool（コネクションプールの一覧表示）」を参照してください。

ここでは，メンバコネクションプールの状態について説明します。

メンバコネクションプールの状態は，J2EE サーバやメンバリソースアダプタを再起動しても維持されます。コネクションプールの状態を確認する方法については，「3.17.5(1)　コネクションプールの状態の確認」を参照してください。

なお，メンバリソースアダプタ以外のコネクションプールには状態はありません。

(a)　コネクションプールの状態遷移

メンバコネクションプールの状態遷移を次の図に示します。

図 3-64　メンバコネクションプールの状態遷移（手動で一時停止を実行する場合）

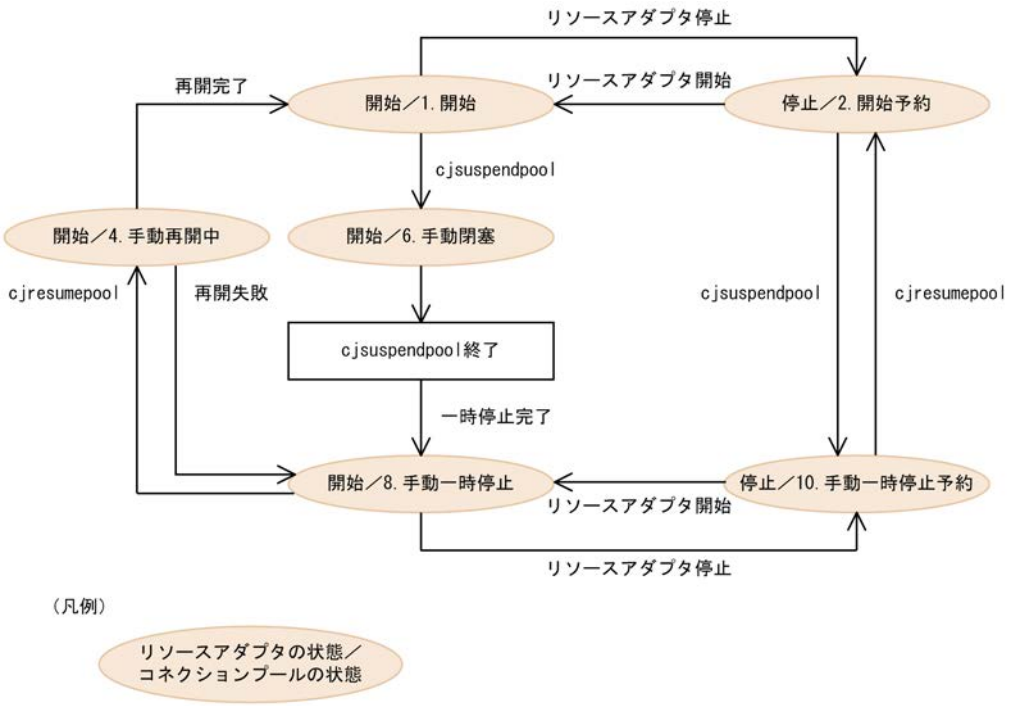
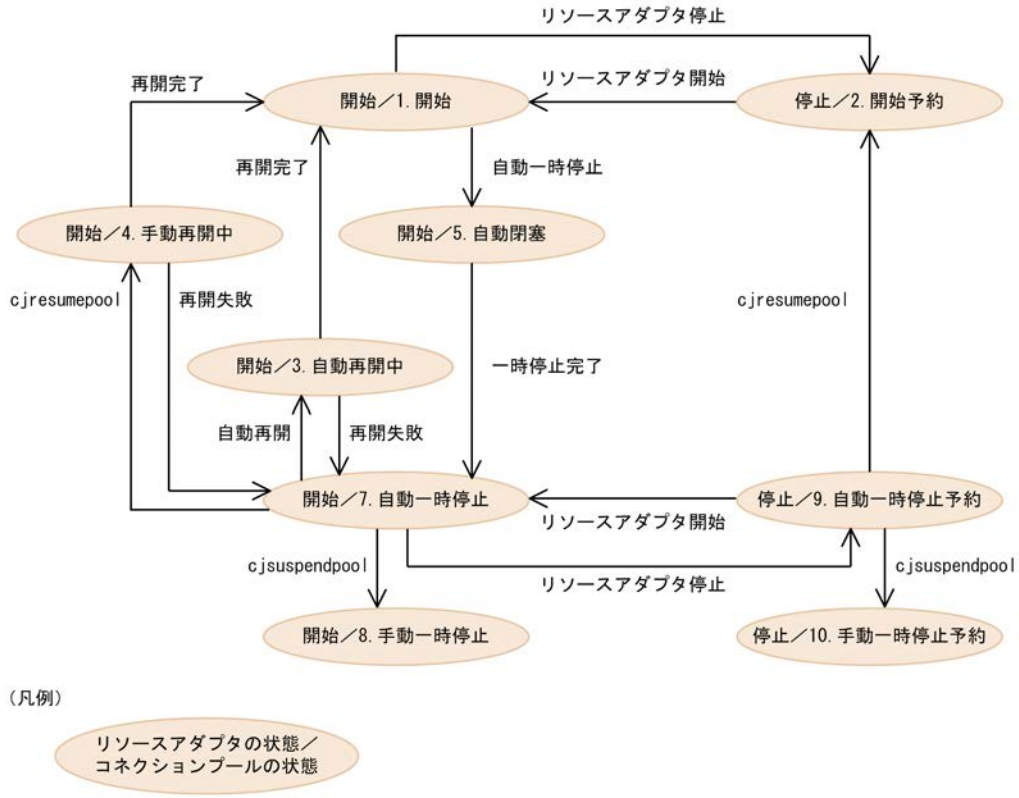


図 3-65　メンバコネクションプールの状態遷移（自動で一時停止が実行される場合）



なお，J2EE サーバの開始時にコネクションプールが再開中状態，または閉塞状態の場合，コネクションプールは一時停止状態に遷移します。

それぞれの状態の詳細について次の表に示します。

表 3-74　メンバコネクションプールの状態

番号	状態	メッセージに表示される状態	説明
1	開始状態	running	コネクションプールが処理を受け付けている状態です。ルートリソースアダプタへのコネクション取得要求時には，開始状態のコネクションプールだけに処理が行われます。
2	開始予約状態	runningReserved	コネクションプールが開始状態のときにリソースアダプタが停止された状態です。開始予約状態のコネクションプールは，リソースアダプタの開始時に開始状態になります。リソースアダプタをデプロイした直後は，この状態になります。

番号	状態	メッセージに表示される状態	説明
3	自動再開中状態	resumingAutomatically	自動再開機能によって，コネクションプールが再開処理を行っている状態です。再開処理が成功すると開始状態になります。再開処理が失敗すると，自動一時停止状態に戻ります。自動再開中状態のコネクションプールには，ルートリソースアダプタへのコネクション取得要求時に処理が出されません。
4	手動再開中状態	• resumingManuallyFromSuspendedAutomatically • resumingManuallyFromSuspendedManually • resumingManually	自動一時停止状態，または手動一時停止状態のときにcjresumepool コマンドを実行して，コネクションプールが再開処理を行っている状態です。再開処理が成功すると，開始状態になります。再開処理が失敗すると，コマンド実行前の状態に戻ります。手動再開中状態のコネクションプールには，ルートリソースアダプタへのコネクション取得要求時に処理が出されません。
5	自動閉塞状態	blockedAutomatically	自動一時停止機能によって，コネクションプールが閉塞されて，一時停止処理を行っている状態です。一時停止処理が完了すると，自動一時停止状態になります。自動閉塞状態のコネクションプールには，ルートリソースアダプタへのコネクション取得要求時に処理が出されません。
6	手動閉塞状態	blockedManually	cjsuspendpool コマンドによって，コネクションプールが閉塞され，一時停止処理を行っている状態です。一時停止処理が完了すると，手動一時停止状態になります。手動閉塞状態のコネクションプールには，ルートリソースアダプタへのコネクション取得要求時に処理が出されません。
7	自動一時停止状態	suspendedAutomatically	自動一時停止機能によって，コネクションプールが一時停止された状態です。コネクションプール内にコネクショ

番号	状態	メッセージに表示される状態	説明
7	自動一時停止状態	suspendedAutomatically	ンは存在しません。自動一時停止状態のコネクションプールには，ルートリソースアダプタへのコネクション取得要求時に処理が出されません。
8	手動一時停止状態	suspendedManually	cjsuspendpool コマンドによって，コネクションプールが一時停止された状態です。コネクションプール内にコネクションは存在しません。手動一時停止状態のコネクションプールには，ルートリソースアダプタへのコネクション取得要求時に処理が出されません。
9	自動一時停止予約状態	suspendedAutomaticallyReserved	コネクションプールが自動一時停止状態のときにリソースアダプタを停止した状態です。自動一時停止予約状態のコネクションプールは，リソースアダプタの開始時に自動一時停止状態になります。なお，このとき，コネクションプールのウォーミングアップ機能は無効になります。
10	手動一時停止予約状態	suspendedManuallyReserved	コネクションプールが手動一時停止状態のときにリソースアダプタを停止した状態です。手動一時停止予約状態のコネクションプールは，リソースアダプタの開始時に手動一時停止状態になります。なお，このとき，コネクションプールのウォーミングアップ機能は無効になります。

注　番号は，図 3-64 および図 3-65 中の番号を示します。

(b)　コネクションプールの状態によるコマンド実行の可否

コネクションプールの状態によって，実行できるコマンドと実行できないコマンドがあります。コネクションプールの状態ごとに，各コマンドの実行の可否を次に示します。

表 3-75　コネクションプールの状態によるコマンド実行の可否

コマンド	コネクションプールの状態							
	開始	開始予約	自動再開中/手動再開中	自動閉塞/手動閉塞	自動一時停止	手動一時停止	自動一時停止予約	手動一時停止予約
cjstartrar	×	○	×	×	×	×	○	○
cjstoprar	○	×	△※1	△※2	○	○	×	×
cjclearpool	○	×	×	×	×	×	×	×
cjlistrar	○	○	○	○	○	○	○	○
cjsuspendpool	○	○	×	×	○	×	○	×
cjresumepool	×	×	×	×	○	○	○	○
cjstopsv(通常停止時)	○	○	△※1	△※2	○	○	○	○
cjstopsv(強制停止時)	○	○	○	○	○	○	○	○

（凡例）　○：実行できる　△：制限あり　×：実行できない

注※1　コマンドは受け付けられますが，開始または一時停止状態になってから処理が実行されます。

注※2　コマンドは受け付けられますが，一時停止状態になってから処理が実行されます。

(4)　コネクションプールの選択方式

J2EE アプリケーションがルートリソースアダプタにコネクションの取得要求を出した時に，メンバコネクションプールが一つ選択されます。このときメンバコネクションプールが選択される方式は，ラウンドロビン方式です。

選択対象のコネクションプールは，開始状態のメンバコネクションプールです。コネクションに空きがあるメンバコネクションプールが優先的に選択されます。

ただし，コネクション枯渇状態のメンバコネクションプールしかない場合は，コネクション取得要求が待ち状態になります。さらに，コネクション取得待ちのタイムアウトが発生すると，コネクションの取得に失敗します。また，コネクション取得要求が待ち状態の場合にコネクションプールが閉塞されたときは，コネクション取得要求が再開され，次の優先度のメンバコネクションプールからコネクションの取得を試みます。すべてのメンバコネクションプールからコネクションが取得できない場合には，コネクションの取得に失敗します。

なお，開始状態のメンバコネクションプールがない場合にコネクション取得要求があったときには，コネクションの取得に失敗します。

また，各メンバコネクションプールの最大サイズを設定する場合，次の指針に従って設定します。

メンバコネクションプールの最大サイズ（数）＝システムで許容される最大同時接続数÷データベースノードの数

3.17.5　手動によるコネクションプールの停止・開始の流れ

ここでは，クラスタコネクションプールの流れについて説明します。データベースに発生した障害に対応する場合や，データベースをメンテナンスする場合，一部のコネクションプールを手動で停止，再開始するこ

とで，システム全体を止めることなくデータベースをメンテナンスできます。一部のコネクションプールを手動で停止，再開始して，データベースをメンテナンスする作業は次の流れで行います。

1. コネクションプールの状態を確認する（3.17.5(1)参照）
   サーバ管理コマンドを使用して実行します。
2. コネクションプールを一時停止する（3.17.5(2)参照）
   サーバ管理コマンドを使用して実行します。
3. コネクションプールを再開する（3.17.5(3)参照）
   サーバ管理コマンドを使用して実行します。
4. コネクションプールの状態を確認する（3.17.5(1)参照）
   サーバ管理コマンドを使用して実行します。

(1)　コネクションプールの状態の確認

クラスタ構成のデータベースをメンテナンスする前，およびメンテナンスしたあとに，コネクションプールの状態を確認します。

コネクションプールの状態の詳細については「3.17.4(3)　コネクションプールの状態」を参照してください。

コネクションプールの状態の確認は，cjlistrar コマンドで実行できます。

cjlistrar コマンドの実行形式と実行例を次に示します。

実行形式

```
cjlistrar <サーバ名> -clusterpool
```

実行例

```
cjlistrar MyServer -clusterpool
```

(2)　コネクションプールの一時停止

ここでは，コネクションプールを一時停止する手順について説明します。

コネクションプールの一時停止は，cjsuspendpool コマンドで実行できます。cjsuspendpool コマンドを実行すると，コネクションプールが閉塞されて一時停止状態になり，コネクション取得要求を受け付けなくなります。

cjsuspendpool コマンドは，コネクションプールが次の状態の場合に実行できます。

- 開始
- 開始予約
- 自動一時停止
- 自動一時停止予約

コネクションプールの状態を確認する方法については，「(1)　コネクションプールの状態の確認」を参照してください。

cjsuspendpool コマンドの実行形式と実行例を次に示します。

実行形式

```
cjsuspendpool ＜サーバ名＞ -resname ＜一時停止対象となるメンバリソースアダプタの表示名＞
```

実行例

```
cjsuspendpool MyServer -resname DB_Connector_for_Oracle_ClusterPool_Member
```

(3)　コネクションプールの再開

ここでは，コネクションプールを再開する手順について説明します。

コネクションプールの再開は，cjresumepool コマンドで実行できます。cjresumepool コマンドを実行すると，コネクションプールは手動再開中状態になり，再開処理が実行されます。再開処理が完了すると，コネクションプールは開始状態になり，コネクション取得要求を受け付けるようになります。

cjresumepool コマンドは，コネクションプールが次の場合に実行できます。

- 使用されていないコネクション管理スレッドが，コネクションプールのコネクションの最大値以上ある
- コネクションプールが，手動一時停止，手動一時停止予約，自動一時停止，または自動一時停止予約の状態である

コネクションプールの状態を確認する方法については，「(1)　コネクションプールの状態の確認」を参照してください。

cjresumepool コマンドの実行形式と実行例を次に示します。

実行形式

```
cjresumepool ＜サーバ名＞ -resname ＜再開対象となるメンバリソースアダプタの表示名＞
```

実行例

```
cjresumepool MyServer -resname DB_Connector_for_Oracle_ClusterPool_Member
```

なお，cjresumepool コマンドを実行したあと，コネクションプールの状態を確認して，再開処理が正しく実行されたかどうかを確認してください。コネクションプールの状態を確認する方法については，「(1)　コネクションプールの状態の確認」を参照してください。

3.17.6　コネクションプールをクラスタ化するために必要な設定

コネクションプールをクラスタ化するためには，DB Connector のプロパティ設定が必要になります。DB Connector のプロパティ設定については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「4.3　データベースと接続するための設定（クラスタコネクションプールの場合）」を参照してください。

3.18　リソースへの接続テスト

リソースアダプタのプロパティ設定後，設定した内容が正しいかどうか，検証することができます。これを**接続テスト機能**といいます。接続テストで検証される内容は，接続するリソースごとに異なります。リソースごとの接続テストでの検証内容を，次の表に示します。

なお，接続テストの実行手順や，リソースアダプタの設定については，「3.3.8　リソースアダプタの設定の流れ（J2EE リソースアダプタとしてデプロイして使用する場合）」，および「3.3.9　リソースアダプタの設定の流れ（J2EE アプリケーションに含めて使用する場合）」を参照してください。

表 3-76　接続テストでの検証内容

リソースの種類	接続方法	接続テストでの検証内容
データベース	DB Connector	• 1.4 モードであること • 指定されたトランザクションサポートレベルが使用できること※1，※2 • DB Connector の設定が正しいこと • 使用する JDBC ドライバの設定が正しいこと • データベースとの接続を確立できること • SQL 発行が成功すること
	DB Connector（クラスタコネクションプール機能を使用した場合のルートリソースアダプタ）	• 1.4 モードであること • 指定されたトランザクションサポートレベルが使用できること※1 • DB Connector の設定が正しいこと • ルートリソースアダプタの開始に必要な設定がされていること※3
	DB Connector（クラスタコネクションプール機能を使用した場合のメンバリソースアダプタ）	• 1.4 モードであること • 指定されたトランザクションサポートレベルが使用できること※1 • DB Connector の設定が正しいこと • Oracle JDBC Thin Driver の設定が正しいこと • データベースとの接続を確立できること • SQL 発行が成功すること • メンバリソースアダプタの前提となる機能を使用していること※4 • メンバリソースアダプタで使用できない機能を使用していないこと※4
データベース上のキュー	Reliable Messaging	• 1.4 モードであること • 指定されたトランザクションサポートレベルが使用できること※1

リソースの種類	接続方法	接続テストでの検証内容
データベース上のキュー	Reliable Messaging	• Reliable Messaging の設定が正しいこと • 連携する DB Connector for Reliable Messaging が開始していること • 連携する DB Connector for Reliable Messaging の設定が正しいこと • 使用するデータベースとの接続を確立できること
	DB Connector for Reliable Messaging	• DB Connector for Reliable Messaging の設定が正しいこと • DB Connector for Reliable Messaging が開始していること • 連携する Reliable Messaging の設定が正しいこと • 連携する Reliable Messaging が開始していること • 使用する JDBC ドライバの設定が正しいこと • 使用するデータベースとの接続を確立できること • SQL 発行が成功すること
OpenTP1	TP1/Message Queue - Access	• 1.4 モードであること • 指定されたトランザクションサポートレベルが使用できること※1 • TP1/Message Queue - Access の設定が正しいこと • TP1/Message Queue - Access が，TP1/Message Queue との接続を確立できること
	TP1 Connector	• 1.4 モードであること • 指定されたトランザクションサポートレベルが使用できること※1 • TP1 Connector の設定が正しいこと
	TP1 インバウンドアダプタ	• Inbound での通信のため，接続テストはできません。接続テストをしようとすると，KDJE48606-E のメッセージが出力されます。
CJMSP ブローカー	CJMSP リソースアダプタ	• 1.4 モードであること • CJMSP ブローカーとの接続を確立できること
SMTP サーバ	メールコンフィグレーション	• SMTP サーバへの接続を確立できること

リソースの種類	接続方法	接続テストでの検証内容
そのほか	この製品が提供しているリソースアダプタ以外のリソースアダプタ	• 1.4 モードであること • リソースへの接続を確立できること※5（javax.resource.spi.ManagedConnectionFactory の実装クラスの createManagedConnection メソッド呼び出しが成功することによって，リソースとの接続が確立できたものとします）。 • Connector1.5 に準拠したリソースアダプタの場合は，リソースアダプタを開始していないときに，開始と停止ができること（javax.resource.spi.ResourceAdapter の実装クラスの start メソッドおよび stop メソッド呼出しが成功することによって，リソースアダプタを開始・停止できたものとします）。

注※1　リソースアダプタのトランザクションサポートレベルにグローバルトランザクション（XATransaction）が指定されている場合は，usrconf.properties の ejbserver.distributedtx.XATransaction.enabled キーが true になっていれば使用できます。なお，トランザクションサポートレベルについては，「3.4.3　リソースごとに使用できるトランザクションの種類」を参照してください。

注※2　接続先のデータベースが XDM/RD E2 のときは，グローバルトランザクション（XATransaction）は使用できません。トランザクションリカバリ用の物理コネクションの取得に失敗するため，DB Connector の開始ができません。

注※3　ルートリソースアダプタの開始に必要な設定については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「4.3.3　ルートリソースアダプタ用 DB Connector の設定」を参照してください。

注※4　メンバリソースアダプタの前提となる機能，および使用できない機能については，「3.17.3　使用するリソースアダプタ」を参照してください。

注※5　Outbound の通信をする場合にリソースとの接続に必要なコネクション定義がないときは，メッセージ KDJE48606-E が出力されてテストに失敗します。

3.19　ファイアウォール環境での運用のための機能

ファイアウォールをシステムに組み込んで運用する場合，通信に使用するポートを固定し，固定したポートを使用した通信だけを許可する必要があります。

この節の構成を次の表に示します。

表 3-77　この節の構成（ファイアウォール環境での運用のための機能）

分類	タイトル	参照先
解説	トランザクションリカバリ用通信ポート	3.19.1
	スマートエージェントが使用する通信ポート	3.19.2
設定	ファイアウォール環境での運用のための設定	3.19.3

注　「実装」および「運用」について，この機能固有の説明はありません。

参考

コネクションスイーパによる通信性能低下の回避

ファイアウォールには，一定時間以上無通信のセッションを強制切断する機能を持つものがあります。ファイアウォールがアプリケーションサーバとデータベースサーバの間に配置されていて，コネクションプールが有効の場合には，しばらくの間使用されなかったコネクションがファイアウォールによって切断されることがあります。その後の切断されたコネクションの使用は，長時間の待ちが発生します。この現象は，ファイアウォールの無通信切断時間よりも早い時間で未使用コネクションを破棄するようにコネクションスイーパ機能を設定することで回避できます。

3.19.1　トランザクションリカバリ用通信ポート

グローバルトランザクションを使用する場合，トランザクションリカバリ処理用の通信ポートを使用します。ファイアウォールをシステムに組み込んで運用する場合，ファイアウォールの構成によっては，このポートの通信を許可する必要があります。デフォルトではポート番号として 20302 が使用されますが，変更することもできます。

トランザクションリカバリ用通信ポートの設定は，J2EE サーバのプロパティをカスタマイズして設定します。J2EE サーバの動作設定のカスタマイズについては,「3.19.3　ファイアウォール環境での運用のための設定」を参照してください。

3.19.2　スマートエージェントが使用する通信ポート

TPBroker のスマートエージェント（OSAgent）は通信ポートを使用します。ファイアウォールをシステムに組み込んで運用する場合，ファイアウォールの構成によっては，このポートを使用する通信を許可する必要があります。

スマートエージェントが使用する通信ポートの設定は，J2EE サーバのプロパティをカスタマイズして設定します。J2EE サーバの動作設定のカスタマイズについては,「3.19.3　ファイアウォール環境での運用のための設定」を参照してください。

なお，J2EE サーバのプロパティに設定した通信ポートは，環境変数 OSAGENT_PORT にも指定する必要があります。J2EE サーバのプロパティと，環境変数 OSAGENT_PORT には，必ず，同じ値を設定してください。

3.19.3　ファイアウォール環境での運用のための設定

ファイアウォール環境で運用する場合，J2EE サーバの設定が必要です。簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に，次の表に示す項目を設定します。

表 3-78　簡易構築定義ファイルでのファイアウォール環境での運用のための設定

項目	設定するパラメタ	設定内容
トランザクションのリカバリ	ejbserver.distributedtx.recovery.port	トランザクションのリカバリで使用する固定ポート番号を指定します。
スマートエージェントが使用する通信ポート	vbroker.agent.port	スマートエージェントが使用する通信ポートを指定します。

3.20　EJB クライアントアプリケーションでトランザクションを開始する場合の注意事項

この節では, EJB クライアントアプリケーションでトランザクションを開始する場合の注意事項について説明します。

ポイント

EJB クライアントアプリケーションでトランザクションを開始できるのは，EJB クライアントマシンにアプリケーションサーバをインストールしている場合です。Client をインストールした EJB クライアントマシンでは，EJB クライアントアプリケーションでトランザクションを開始することはできません。

EJB クライアントアプリケーションでトランザクションを開始した場合, グローバルトランザクションを使用してアプリケーションサーバ側の EJB を呼び出せます。このとき，トランザクションマネージャとトランザクションサービスは，EJB クライアントとアプリケーションサーバ間でトランザクションを伝播し，最終的に 2 フェーズコミットを実行します。

EJB クライアントアプリケーションで開始したトランザクション内では, アプリケーションサーバ上の複数の EJB を呼び出せます。アプリケーションサーバ側では，複数のリソースにアクセスできます。なお，EJB クライアントアプリケーションからリソースに直接アクセスすることはできません。

ここでは, EJB クライアントアプリケーションでトランザクションを開始する場合に注意することについて説明します。

3.20.1　アプリケーション開発時の注意事項

EJB クライアントアプリケーションでトランザクションを開始する場合に, アプリケーション開発時に注意することを次に示します。EJB クライアントアプリケーションでのトランザクションの実装については, マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「3.5　EJB クライアントアプリケーションでのトランザクションの実装」を参照してください。

- EJB クライアントアプリケーションでグローバルトランザクションを使用するには，EJB クライアントアプリケーション起動直後に，EJB クライアントアプリケーションで使用するサービスを初期化するための処理をユーザプログラムから呼び出す必要があります。この初期化処理を呼び出すことで，トランザクションマネージャとトランザクションサービスが初期化されます。初期化処理としては，EJBClientInitializer クラスの initialize()メソッドを呼び出してください。
- トランザクション処理中に EJB クライアントアプリケーションが異常終了した場合も，サービスを初期化するための処理をユーザプログラムで呼び出すことによって，トランザクションのリカバリ処理が開始できるようになります。なお, リカバリ処理はバックグラウンドで実行されるため, initialize()メソッドはリカバリ処理の完了を待たないでリターンします。EJB クライアントアプリケーションの再起動によってグローバルトランザクションのリカバリ処理が開始するように，EJB クライアントアプリケーションを設計してください。
- EJB クライアントアプリケーションでトランザクションを開始している場合，必ずすべてのトランザクションを決着させてから，EJB クライアントアプリケーションが停止するように設計してください。トランザクションの決着処理を待たないで EJB クライアントアプリケーションが停止すると，未決着トランザクションがトランザクション内に残留するおそれがあります。この状態になると，アプリケーションサーバの正常停止やリソースアダプタの停止ができなくなったり，リソースのロックが解除されなくなったりする場合があります。

3.20.2　システム構築時の注意事項

ここでは，システム構築時に注意することと設定について説明します。

(1)　システム構築時の注意

EJB クライアントアプリケーションでトランザクションを開始する場合に，システム構築時に注意することを次に示します。EJB クライアントアプリケーションのトランザクションの設定については，「(2)　EJB クライアントアプリケーションでトランザクションを使用するための設定」を参照してください。

- EJB クライアントアプリケーションでトランザクションを開始する場合，グローバルトランザクションを使用するために，アプリケーションサーバ側でライトトランザクション機能を無効に設定する必要があります（デフォルトでは有効になっています）。
- 呼び出される EJB を，EJB コンテナで管理するトランザクション（CMT）で Mandatory 属性，Required 属性，Supports 属性などに指定すると，EJB クライアントアプリケーション側で開始したトランザクションの範囲内で実行されます。

(2)　EJB クライアントアプリケーションでトランザクションを使用するための設定

ここでは，EJB クライアントアプリケーションでトランザクションを使用するための設定について説明します。

！ 注意事項

Client を使用して EJB クライアント環境を構築する場合は，EJB クライアントアプリケーションのトランザクションは使用できません。

EJB クライアントアプリケーションでトランザクションを使用するためには，次の設定が必要です。

- JAR ファイルの設定
- プロパティの設定
- UserTransaction の取得の設定

UserTransaction の取得の設定は，J2EE アプリケーション開発時の設定内容です。EJB クライアントアプリケーションでトランザクションを使用するために，EJB クライアントアプリケーションから UserTransaction（javax.transaction.UserTransaction）を取得するための設定が必要です。設定方法については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「3.5　EJB クライアントアプリケーションでのトランザクションの実装」を参照してください。

ここでは，EJB クライアントアプリケーションでトランザクションを使用するための JAR ファイルとプロパティの設定について説明します。

(3)　JAR ファイルの設定

EJB クライアントアプリケーションでトランザクションを使用するために，次の JAR ファイルをクラスパスに設定します。

Windows の場合

- <Application Server のインストールディレクトリ>¥TPB¥lib¥tpotsinproc.jar
- <Application Server のインストールディレクトリ>¥CC¥lib¥ejbserver.jar※

UNIX の場合

- /opt/Cosminexus/TPB/lib/tpotsinproc.jar
- /opt/Cosminexus/CC/lib/ejbserver.jar※

注※

クラスパスの設定で，ejbserver.jar は HiEJBClientStatic.jar よりも後ろに設定してください。

クラスパスへの JAR ファイルの設定については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「3.7.4　EJB クライアントアプリケーションのクラスパスへの JAR ファイルの設定」を参照してください。

(4)　プロパティの設定

EJB クライアントアプリケーションでトランザクションを使用するために設定が必要なキーについて説明します。プロパティの設定については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「3.3.5　EJB クライアントアプリケーションのプロパティの設定」を参照してください。各キーの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「14.3　usrconf.properties（Java アプリケーション用ユーザプロパティファイル)」を参照してください。

EJB クライアントアプリケーションでトランザクションを使用するためには，次のキーを設定します。

(a)　必須のプロパティ

- ejbserver.client.transaction.enabled
  「true」を指定して，EJB クライアントアプリケーションでのトランザクションの使用を有効にします。
- ejbserver.distributedtx.recovery.port
  グローバルトランザクションを使用するとき，トランザクションリカバリで使用する固定ポート番号を指定します。EJB クライアントのプロセスごとに別のポート番号を指定してください。また，同一マシン上で動作するアプリケーションサーバのリカバリ用ポート番号とは別のポート番号を指定してください。
- ejbserver.client.transaction.clientName
  トランザクションサービスが使用するクライアント名を指定します。ここでは，EJB クライアントのプロセスごとに別の名前を指定してください。また，同一マシン上で動作する J2EE サーバ名とは別の名前を指定してください。
- ejbserver.distributedtx.ots.status.directory1
  トランザクションサービスが使用するステータスファイルや，そのバックアップを配置するディレクトリを指定します。EJB クライアントのプロセスごとに別のディレクトリを指定してください。また，J2EE サーバに指定したステータスファイルのディレクトリとは別のディレクトリを指定してください。

(b)　任意のプロパティ

- ejbserver.jta.TransactionManager.defaultTimeOut
  トランザクションタイムアウトが発生するまでの時間を指定します。
- ejbserver.distributedtx.ots.status.directory2
  トランザクションサービスが使用するステータスファイルや，そのバックアップを配置する二つ目のディレクトリを指定します。EJB クライアントのプロセスごとに別のディレクトリを指定してください。また，J2EE サーバに指定したステータスファイルのディレクトリとは別のディレクトリを指定してください。

(5)　注意事項

EJB クライアントアプリケーションのトランザクションの設定での注意事項を次に示します。

グローバルトランザクションを使用するためには，アプリケーションサーバ側でライトトランザクション機能を無効にする必要があります。デフォルトの設定では，ライトトランザクション機能が有効になっています。J2EE サーバの usrconf.properties の ejbserver.distributedtx.XATransaction.enabled キーで「true」を指定すると，ライトトランザクション機能が無効となり，グローバルトランザクションが利用できるようになります。

3.20.3　システム運用時の注意事項

EJB クライアントアプリケーションでトランザクションを開始した場合に，運用時に注意することを次に示します。

- EJB クライアントアプリケーションの停止後に，アプリケーションサーバやリソースアダプタを停止できない場合は，未決着トランザクションがトランザクション内に残留しているおそれがあります。この場合，EJB クライアントアプリケーションを再起動して，グローバルトランザクションのリカバリ処理を実施してください。

  なお，リカバリ処理は，EJB クライアントアプリケーション内で，再起動後にサービスの初期化処理を呼び出されることで実行されます。
- トランザクション処理中に障害などの理由で EJB クライアントマシンがダウンした場合は，EJB クライアントアプリケーションを再起動して，グローバルトランザクションのリカバリ処理を実施する必要があります。
- EJB クライアントアプリケーションで開始したトランザクションは，メソッドキャンセル機能を使用して強制決着させることはできません。
- サービスの初期化処理で例外が発生した場合，EJB クライアントアプリケーションの実行時のシステムプロパティなどが正しく設定されていないおそれがあります。例外メッセージに従って対処してください。
- EJB クライアントアプリケーションでトランザクションを開始した場合，JTA や OTS が出力する性能解析トレースにルートアプリケーション情報やクライアントアプリケーション情報が含まれません。リクエストをトレースする場合には，スレッドのハッシュコードや XID の情報を使用してください。

  また，トランザクションタイムアウトが発生した時に出力されるメッセージには，ルートアプリケーション情報の代わりに，トランザクションを開始したスレッドのハッシュコードが含まれています。この情報を使用してトレースすることもできます。

4 OpenTP1 からのアプリケーションサーバの呼び出し（TP1 インバウンド連携機能）

この章では，OpenTP1 システムで動作する SUP からアプリケーションサーバ上の J2EE アプリケーションを Inbound で呼び出す機能である，TP1 インバウンド連携機能について説明します。

なお，この機能を使用できる製品については，「4.3　TP1 インバウンド連携機能の前提条件」を参照してください。

4.1　この章の構成

この章では，TP1 インバウンド連携機能について説明します。

この章の構成を次の表に示します。

表 4-1　この章の構成（TP1 インバウンド連携機能）

分類	タイトル	参照先
解説	TP1 インバウンド連携機能の概要	4.2
	TP1 インバウンド連携機能の前提条件	4.3
	TP1 インバウンド連携機能で使用するメモリの見積もり	4.4
	OpenTP1 の機能との対応	4.5
	コネクション管理機能	4.6
	RPC 通信機能	4.7
	スケジュール機能	4.8
	トランザクション連携機能	4.9
実装	Message-driven Bean（サービス）の実装	4.10
	OpenTP1 のアプリケーションプログラムの作成	4.11
設定	アプリケーションサーバの実行環境での設定	4.12
	OpenTP1 での設定	4.13
	OpenTP1 とアプリケーションサーバ間のタイムアウトの設定	4.14
運用	TP1 インバウンドアダプタと Message-driven Bean（サービス）の開始と終了	4.15
	性能解析トレース情報の引き継ぎ	4.16
注意事項	TP1 インバウンドアダプタで発生する RPC エラー応答	4.17

4.2　TP1 インバウンド連携機能の概要

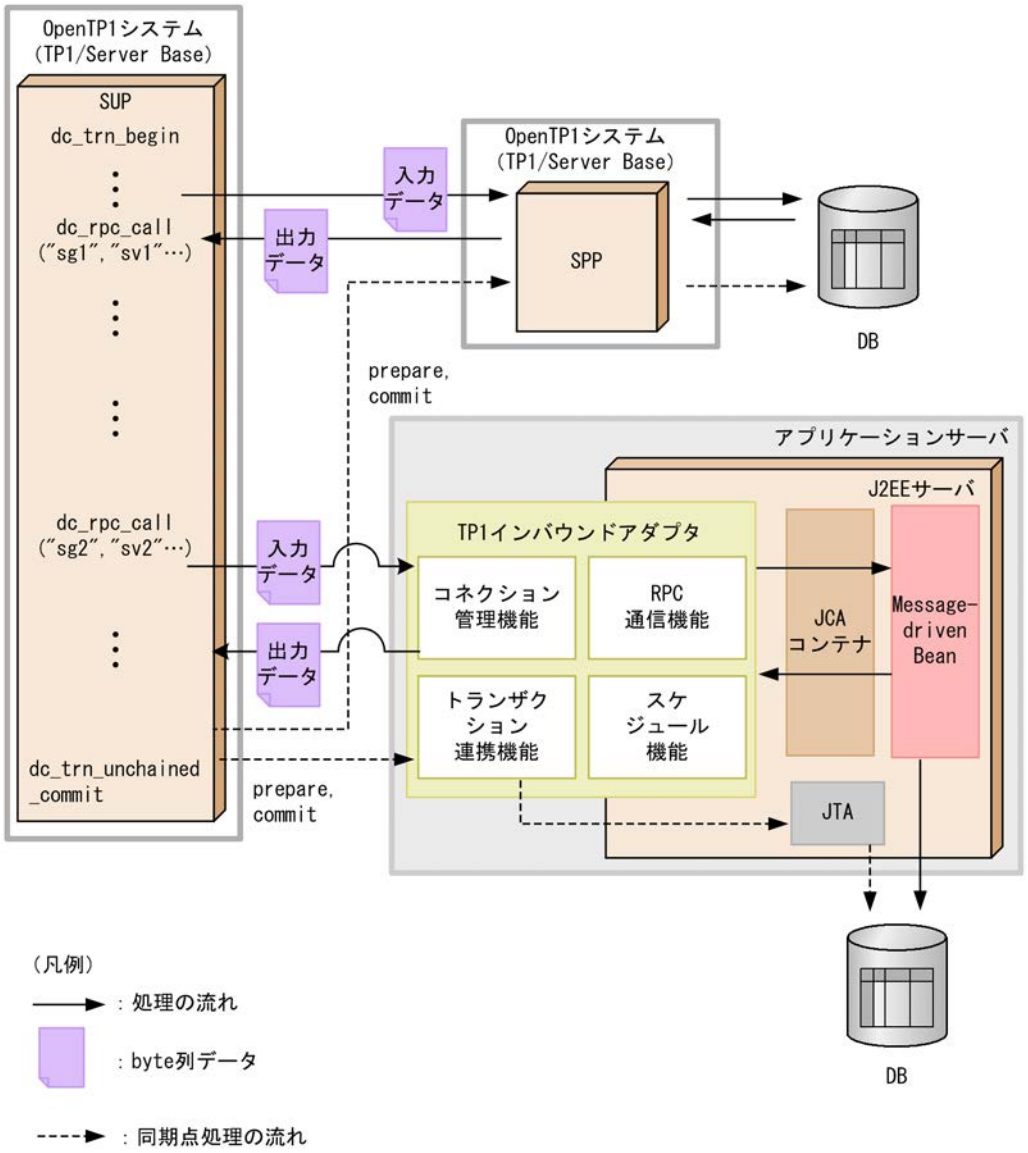
TP1 インバウンド連携機能は，OpenTP1 システム上で動作するユーザアプリケーションプログラムから，アプリケーションサーバ上で動作する J2EE アプリケーションを業務として呼び出すための機能です。この機能によって，OpenTP1 を使用したレガシーシステムを運用している場合に，新たに追加する業務を J2EE アプリケーションとして実装して，既存のシステムに組み込んで使用できます。また，OpenTP1 システム上で動作するユーザアプリケーションプログラムで開始されたトランザクションに，アプリケーションサーバの Message-driven Bean（CMT）で実行する処理を参加させられます。

TP1 インバウンド連携機能では，OpenTP1 のユーザアプリケーションプログラムからの呼び出しをアプリケーションサーバ上で動作する TP1 インバウンドアダプタで受け付け，J2EE サーバ上の Message-driven Bean で業務処理を実行します。**TP1 インバウンドアダプタ**は，Connector 1.5 仕様に準拠したリソースアダプタです。

トランザクション連携機能を使用する場合の TP1 インバウンド連携機能の概要について示します。

なお，TP1 インバウンド連携機能は，ユーザアプリケーションプログラムのうち SUP，SPP，および MHP からの呼び出しが可能ですが，このマニュアルでは，SUP からの呼び出しを例に説明します。

図 4-1　TP1 インバウンド連携機能の概要



TP1 インバウンド連携機能を使用する場合，クライアントとなる SUP からは，OpenTP1 システムの SPP を呼び出す場合と同様の手順でアプリケーションサーバ上の業務を呼び出します。また，SPP の呼び出し時と同様にデータをやり取りできます。

さらに，トランザクション連携機能を使用することで，SUP からのリソース更新と，Message-driven Bean からのリソース更新を同期できます。

TP1 インバウンドアダプタは，SPP と同様の手順での業務の呼び出しを実現するために，次の 4 種類の機能を備えています。

- **コネクション管理機能**

  OpenTP1 とアプリケーションサーバ間のコネクションの接続，切断，およびプーリングを制御する機能です。これによって，コネクションを接続したままで OpenTP1 と通信できます。また，コネクションの接続と切断に伴うボトルネックを削減でき，通信効率を向上できます。なお，OpenTP1 とアプリケーションサーバとの間のコネクション数がしきい値に達すると，コネクションを切断（一時クローズ処理）します。

- **RPC 通信機能**

  OpenTP1 で扱う RPC 電文を解析し，応答電文を生成して，RPC 通信を実行する機能です。JCA コンテナを介して Message-driven Bean を呼び出します。

- **スケジュール機能**

  RPC 要求のスケジューリングを実行する機能です。OpenTP1 のスケジュールサービスと同様の処理を実現するために使用します。

- **トランザクション連携機能**

  OpenTP1 の SUP で開始されたトランザクションに Message-driven Bean（CMT）で実行した処理を参加させる機能です。

ただし，TP1 インバウンドアダプタでは，OpenTP1 のネームサービスの代替機能は提供しません。このため，TP1 インバウンドアダプタを使用するためには，クライアントとなる OpenTP1 側でユーザサービスネットワーク定義によるスケジューラダイレクト機能を利用する必要があります。

4.3　TP1 インバウンド連携機能の前提条件

TP1 インバウンド連携機能は，次のどれかの製品で利用できます。

- TP1/Server Base Version 5 以降
- uCosminexus TP1/Server Base Version 7 以降
- uCosminexus TP1/LiNK Version 7 以降

4.4　TP1 インバウンド連携機能で使用するメモリの見積もり

TP1 インバウンド連携機能を使用する際に追加で必要となるメモリ所要量（単位：メガバイト）は，次の計算式に従って見積もってください。

TP1 インバウンド連携機能を使用する際に追加で必要となるメモリ所要量

=2+1.8×MDB 最大インスタンス数の合計

＋ RPC 要求電文の最大サイズ×（最大同時接続数+スケジュールキュー長の合計+MDB の最大インスタンス数の合計）

＋ RPC 応答電文の最大サイズ×MDB 最大インスタンス数の合計

4.5　OpenTP1 の機能との対応

この節では，TP1 インバウンド連携機能と OpenTP1 の機能との対応について説明します。

4.5.1　クライアントでサポートする OpenTP1 の機能

アプリケーションサーバを呼び出す OpenTP1 のクライアントでの OpenTP1 の機能の使用可否を次の表に示します。なお，OpenTP1 のそれぞれの機能の詳細については，マニュアル「OpenTP1 解説」およびマニュアル「OpenTP1 プログラム作成の手引」を参照してください。

表 4-2　クライアントでの OpenTP1 の機能の使用可否

機能の分類	OpenTP1 の機能	使用可否
ユーザアプリケーションプログラムの種類	SUP	○
	SPP	○
	MHP	○
RPC の種類	同期応答型 RPC	○
	非同期応答型 RPC	×
	非応答型 RPC	×
	連鎖 RPC	×
	トランザクショナル RPC	○
	認証 RPC	×
呼び出し先の RPC 受信形態	キュー受信型サーバ	○
	ソケット受信型サーバ	×
リモート API 機能	リモート API 機能	×
RPC のサービス要求方式	ユーザサービスネットワーク定義によるスケジューラダイレクト機能	○
	dc_rpc_call_to 関数によるスケジューラダイレクト機能	○
	ネームサービス機能	×
	ドメインネームシステム機能	×
	グローバル検索機能	×
	マルチスケジューラ機能	×
電文操作	ユーザデータ電文の圧縮機能	×
ユーザ電文	ユーザ電文長 8 メガバイト拡張	○
そのほか	コネクションの保持	○
	テストモード	×
	要求用 TCP/IP コネクションの接続リトライ	○

機能の分類	OpenTP1 の機能	使用可否
そのほか	詳細エラーコード取得機能	×

（凡例）
○：使用できる　×：使用できない

各機能に対する TP1 インバウンドアダプタでの対応について，機能の分類ごとに説明します。

(1)　ユーザアプリケーションプログラムの種類

TP1 インバウンドアダプタでは，SUP（サービス利用プログラム），SPP（サービス提供プログラム），および MHP（メッセージ処理プログラム）からの処理を受け付けます。

(2)　RPC の種類

TP1 インバウンドアダプタでは，OpenTP1 で使用できる RPC 形態のうち，同期応答型 RPC およびトランザクショナル RPC を使用できます。

非同期応答型 RPC または非応答型 RPC を使用して TP1 インバウンドアダプタを呼び出した場合，クライアントにはエラーが返されます。

連鎖 RPC とは，RPC を複数回実行する場合に，各 RPC 呼び出しを必ず同じプロセスで実行することを保証する OpenTP1 の機能です。TP1 インバウンドアダプタでは，連鎖 RPC を保証しません。クライアントで連鎖 RPC を有効にして RPC 通信を実行した場合は，エラーが返されます。

なお，TP1 インバウンドアダプタでは，認証 RPC はサポートしません。

(3)　呼び出し先の RPC 受信形態

TP1 インバウンドアダプタでは，キュー受信型サーバだけを使用できます。

TP1 インバウンドアダプタでは，必ずスケジュールキューを経由してサービス要求を受信します。このため，スケジュールキューを経由しないで直接サービス要求を受信する形態であるソケット受信型サーバ機能は使用できません。

(4)　リモート API 機能

TP1 インバウンドアダプタでは，リモート API 機能は使用できません。

リモート API 機能とは，OpenTP1 の SPP 側で RPC の代理実行をする機能です。OpenTP1 では，リモート API を使用することで，ファイアウォールの内側にある UAP に対してサービスを要求できます。TP1 インバウンドアダプタを使用する場合，ファイアウォールの内側の UAP に対するサービス要求はできません。

(5)　RPC のサービス要求方式

TP1 インバウンドアダプタでは，ユーザサービスネットワーク定義によるスケジューラダイレクト機能を利用して，SUP から SPP を呼び出します。**ユーザサービスネットワーク定義によるスケジューラダイレクト機能**とは，あらかじめ，ユーザサービスネットワーク定義にサービス情報を静的に定義しておくことで，ネームサービスを参照しないで，直接スケジュールサービスにアクセスするための機能です。TP1 インバウンドアダプタを呼び出す場合，必ずこの機能を使用してください。

ユーザサービスネットワーク定義に複数のホストを記載することで，OpenTP1 の機能によってランダムに決定されたノードにサービス要求を転送することができます。一度決定した転送先ノードは，基本的には固

定されます。また，オプションの定義によって，ノード決定を定期間隔で再決定できるようにチューニングできます。

TP1 インバウンドアダプタでは，OpenTP1 のネームサービスを代替する機能は使用できません。このため，ネームサービスを使用したサービス情報の参照はできません。ユーザサービスネットワーク定義によるスケジューラダイレクト機能を使用する設定がない場合，SUP でサービス情報を取得できないため，TP1 インバウンドアダプタを呼び出せません。

なお，TP1 インバウンドアダプタでは，ネームサービスの代替機能を使用できないため，ネームサービスの付加機能である次の機能も使用できません。

- ドメインネームシステム機能
- グローバル検索機能

また，TP1 インバウンドアダプタでは，OpenTP1 のマルチスケジューラ機能を使用できません。マルチスケジューラ機能とは，複数のスケジューラを起動して複数の RPC を同時に受け付ける機能です。ただし，TP1 インバウンドアダプタのスケジュール機能では，受け付けた要求をマルチスレッドで処理するため，すでに電文を処理している場合も，新たに受け付けた別の RPC を同時に処理できます。

参考

OpenTP1 のネームサービス機能

OpenTP1 では，ネームサービスを使用して，各ノードにあるサービスに関する情報を検索できます。

OpenTP1 で SUP から SPP を呼び出す通常の手順では，SUP で dc_rpc_call 関数を実行する際に，ローカル環境の OpenTP1 システム中のネームサービスに対してサービス情報の検索を実行します。ローカル環境のネームサービスは，ほかのノードのネームサービスと情報連携して，各ノードに存在するサービス情報を保持しています。検索実行後，ネームサービスから目的のサービスグループに対応するサービスの情報（スケジュールサービスのあるホストおよびポート番号の情報）が返されます。SUP では，返却された情報を基に，スケジュールサービスに RPC 通信を実行します。

(6)　電文操作

TP1 インバウンドアダプタでは，ユーザデータ電文圧縮機能を使用できません。

ユーザデータ電文圧縮機能とは，RPC によってネットワーク上に送信されたユーザ電文を圧縮することによって，パケット数を削減し，ネットワークの混雑を緩和する OpenTP1 の機能です。

TP1 インバウンドアダプタでは，ユーザデータが圧縮されていた場合，エラーを返却します。

(7)　ユーザ電文

RPC 通信で，8 メガバイトまでのユーザ電文を送受信できます。

(8)　そのほか

TP1 インバウンドアダプタでは，表 4-2 で「そのほか」に分類した機能のうち，コネクションの保持，および要求用 TCP/IP コネクションの接続リトライを実行できます。コネクションの保持では，ノード間で確立したコネクションを保持できます。また，要求用 TCP/IP コネクションの接続リトライは，RPC 要求時にコネクションの確立に失敗した場合に，コネクションの確立をリトライする OpenTP1 の機能です。

次の機能は使用できません。

- テストモードは有効にできません。テストモードが有効になっている場合は，エラーを返却します。

- 詳細エラーコード取得機能は使用できません。なお，詳細エラーコード出力機能とは，エラーが発生した場合に，OpenTP1 の呼び出し元に返却されるエラーコードを詳細化する機能です。

4.5.2　OpenTP1 のサーバの機能との対応

TP1 インバウンドアダプタは，「4.5.1　クライアントでサポートする OpenTP1 の機能」で示した機能のうち，TP1 インバウンドアダプタで使用できる機能を使用して送信された RPC を受け付ける機能を備えています。ただし，OpenTP1 の SPP およびスケジュールサービスで提供されているすべての機能を同じように使用できるわけではありません。

TP1 インバウンドアダプタを使用して実現できることの詳細については，「4.7　RPC 通信機能」および「4.8　スケジュール機能」で説明します。

ここでは，TP1 インバウンドアダプタと OpenTP1 のサーバで動作する SPP の機能との対応を次の表に示します。また，機能差または代替機能がある場合にその概要についても説明します。なお，OpenTP1 の機能の詳細については，マニュアル「OpenTP1 解説」およびマニュアル「OpenTP1 プログラム作成の手引」を参照してください。

表 4-3　OpenTP1 のサーバの機能との対応

機能の分類	OpenTP1 の機能	使用可否	機能差または代替機能の概要
受け付け可能な RPC 形態	同期応答型 RPC	○	機能差はありません。
	非同期応答型 RPC	×	使用できません。
	非応答型 RPC	×	
	連鎖 RPC	×	
	トランザクショナル RPC	○	機能差はありません。
RPC 機能	ネストを使用したコール	×	TP1 インバウンドアダプタでは，別の OpenTP1 の SPP へのネストコールは実行できません。 ただし，TP1/Client などの OpenTP1 に Outbound での接続をするための機能によって，ネストコールを実現できます。 また，リモート EJB 呼び出しなど，アプリケーションサーバ間でネストの呼び出しは実現できます。
	リカーシブコール	×	使用できません。
	コールバック※	×	
リモート API 機能	リモート API の受け付け	×	
サービス	サービスリトライ機能	×	
	サービス実行のタイマ監視	△	アプリケーションサーバ独自の方式でのタイマ監視を実行できます。詳細は，「4.8.4　サービス実行のタイムアウト」を参照してください。
	サービスの並列処理	○	スレッドを使用してサービスを並列処理できます。
	サービスグループ閉塞	×	使用できません。

機能の分類	OpenTP1 の機能	使用可否	機能差または代替機能の概要
サービス	サービス閉塞	×	使用できません。
	サービスの同時実行数制御	○	アプリケーションサーバ独自の方式で同時実行数を制御できます。詳細は，「4.8.3　Message-driven Bean（サービス）の同時実行数の制御」を参照してください。
電文操作	電文の圧縮	×	常に非圧縮の状態で応答します。TP1 インバウンドアダプタが圧縮された電文を受け取った場合は，エラーを返却します。
スケジュールサービス	サービス単位のスケジュールキュー	×	使用できません。
	サービスグループ単位のスケジュールキュー	○	機能差はありません。
	スケジュール閉塞	×	使用できません。
	スケジュールプライオリティ制御	×	
	サービスのスケジュール方式	△	サービスのスケジュールは常に FIFO 方式で制御されます。
	スケジュールキューの滞留の監視	△	アプリケーションサーバ独自の方式でスケジュールキューの滞留を監視できます。
負荷バランス	マルチサーバ負荷バランス	△	アプリケーションサーバでは，プロセスではなくスレッドを使用します。 また，アプリケーションサーバ独自の機能を使用して，負荷に応じてスレッド数を調整できます。詳細は，「4.8.3　Message-driven Bean（サービス）の同時実行数の制御」を参照してください。
	マルチスケジューラ	×	使用できません。 ただし，アプリケーションサーバではマルチスレッドでリクエストを処理するため，マルチスケジューラ機能を使用しなくても一つのスケジュールサービスで複数のリクエストを処理できます。
	ノード間負荷バランス	×	使用できません。 アプリケーションサーバでは，負荷情報は管理しません。 ノード間の振り分けには，ユーザサービスネットワーク定義による，ランダムなサービス要求だけが使用できます。
コネクション	コネクション確立/通信制御データのタイマ監視	△	アプリケーションサーバ独自の方式でのタイマ監視を実行できます。詳細は，「4.8.4　サービス実行のタイムアウト」を参照してください。

機能の分類	OpenTP1 の機能	使用可否	機能差または代替機能の概要
コネクション	コネクション確立要求のキュー	△	アプリケーションサーバ独自の方式で管理するキューを使用できます。詳細は，「4.7　RPC 通信機能」を参照してください。
	一時クローズ処理機能	○	機能差はありません。
	応答用 TCP/IP コネクションの接続リトライ	○	使用できます。 詳細は，「4.7　RPC 通信機能」を参照してください。
性能/障害調査	同期型 dc_rpc_call 関数のタイムアウト時のサーバ負荷軽減機能	×	アプリケーションサーバでは，クライアント側のタイムアウトに関係なく，サービス応答を実行します。
	性能トレース取得機能	○	使用できます。 ただし，出力ファイルおよびファイルの内容は，アプリケーションサーバの性能解析トレースとしてのものになります。
	OpenTP1 デバッグ情報取得機能	×	OpenTP1 独自のログは使用できません。 アプリケーションサーバでは，性能解析トレースまたはメッセージログを使用してデバッグを実行する必要があります。
	RPC トレース	×	OpenTP1 独自のログは使用できません。 アプリケーションサーバでは，通信トレースに相当するログは取得できません。ただし，クライアントの OpenTP1 側の RPC トレースや，性能解析トレースを使用してリクエストの調査は実現できます。
	統計情報取得	×	OpenTP1 独自のログは使用できません。 ただし，アプリケーションサーバの稼働統計情報は取得できます。
	監査ログ	×	OpenTP1 独自の監査ログは使用できません。 ただし，アプリケーションサーバの監査ログは使用できます。
そのほか	MCF サービス	×	使用できません。
	XATMI インタフェース(X/Open 準拠)を使った通信	×	
	TX インタフェース(X/Open 準拠)を使った通信	×	
	TxRPC インタフェース(X/Open 準拠) を使った通信	×	
	OSI TP プロトコルを使った通信	×	

（凡例）

○：使用できる

×：使用できない

△：使用できるが機能差がある

注※

dc_rpc_call 関数で呼び出された SPP 内で，呼び出し元のサービスに対して ee_rpc_call 関数を発行する機能です。

4.6　コネクション管理機能

この節では，コネクション管理機能について説明します。

4.6.1　コネクション管理機能の概要

コネクション管理機能は，OpenTP1 と TP1 インバウンドアダプタ間のコネクションの接続，切断，およびプーリングを制御する機能です。コネクションを接続したままで，OpenTP1 と TP1 インバウンドアダプタの間で通信できます。そのため，コネクションの接続・切断に伴うボトルネックを削減し，通信効率が向上します。

コネクション管理機能が管理するコネクションは 2 種類あります。コネクション管理機能が管理するコネクションの種類，および役割について次の表に示します。

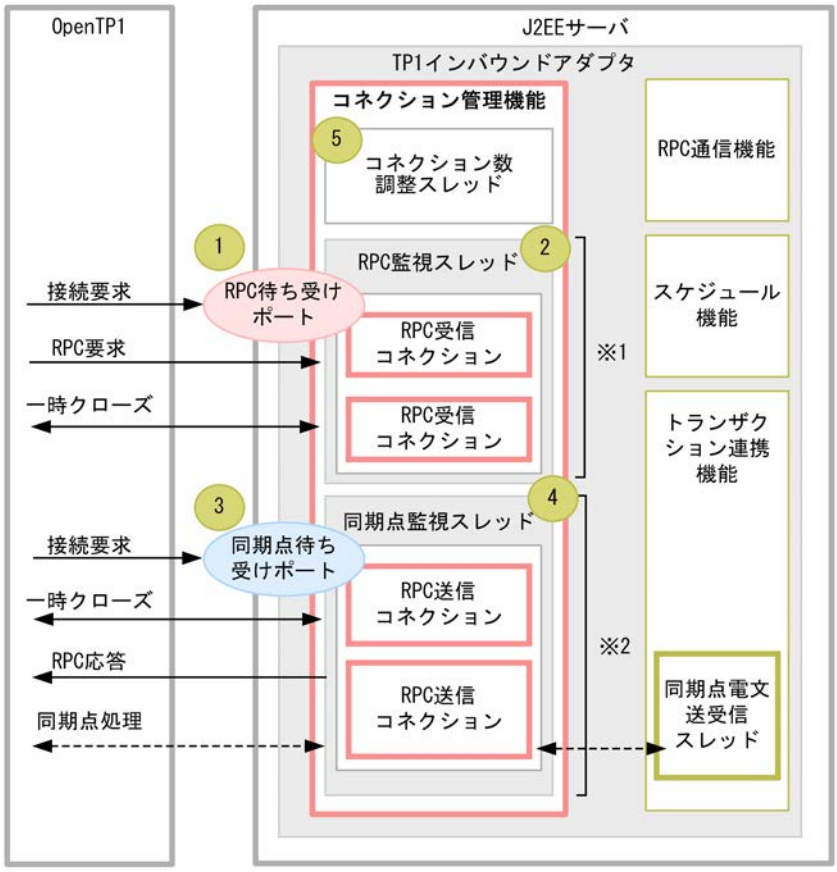
表 4–4　コネクション管理機能が管理するコネクションの種類および役割

コネクションの種類	コネクションの役割
RPC 受信コネクション	OpenTP1 からの RPC 要求を受信します。
RPC 送信コネクション	OpenTP1 への RPC 応答を送信します。 また，トランザクション連携機能を使用している場合，OpenTP1 との同期点処理の送受信をします。

なお，一時クローズ処理の送受信では，どちらのコネクションも使用します。一時クローズ処理については，「4.6.6　一時クローズ処理によるコネクション数の調整」を参照してください。

TP1 インバウンドアダプタでのコネクション管理機能の概要と位置づけについて次の図に示します。

図 4-2　コネクション管理機能の位置づけ



（凡例）

：処理の流れ

：同期点処理の流れ

n　：コネクション管理機能の要素の説明に対応しています。

注※1　RPC要求用のコネクションを管理します。

注※2　RPC応答と同期点処理用のコネクションを管理します。

OpenTP1 からの接続要求を RPC 待ち受けポートで受け付けます。受け付けた要求は，RPC 受信監視スレッド内にある RPC 受信コネクションが受信します。RPC 受信監視スレッドが RPC 受信コネクションの受信を検知すると，RPC 通信機能のリクエスト受け付けスレッドへリクエストを送信します。

コネクション管理機能は，次の要素で構成されています。

1. RPC 待ち受けポート

   OpenTP1 からの RPC 要求を受信する際にコネクションの接続要求を待ち受けるポートです。TP1 インバウンドアダプタの開始時に開き，終了時に閉じます。

2. RPC 監視スレッド

   RPC 受信コネクションに電文が送信されてきているかどうかを監視するスレッドです。RPC 受信コネクションで電文の受信を検知すると，RPC 通信機能のリクエスト受け付けスレッドに電文受信処理を渡します。

   TP1 インバウンドアダプタの開始時に作成し，終了時に消滅します。

3. **同期点待ち受けポート**

   RPC 応答を送信した RPC 送信コネクションが切断済みの場合に，同期点待ち受けポートを利用して OpenTP1 から RPC 送信コネクションを接続します。

   すでに RPC 応答を送信したコネクションがある場合はそのコネクションを使って同期点処理を実行します。

   TP1 インバウンドアダプタの開始時に開き，終了時に閉じます。

4. **同期点監視スレッド**

   RPC 送信コネクションに電文が送信されてきているかどうかを監視するスレッドです。RPC 送信コネクションでの電文の受信を検知すると，トランザクション連携機能の同期点電文送受信スレッドに電文受信処理を渡します。

   TP1 インバウンドアダプタの開始時に作成し，終了時に消滅します。

5. **コネクション数調整スレッド**

   OpenTP1 とのコネクションの接続数を監視し，一定のしきい値に達した場合に一時クローズ処理によって未使用のコネクションを切断するスレッドです。

   コネクション数調整スレッドは，RPC 受信コネクションと RPC 送信コネクションをまとめて監視するため，TP1 インバウンドアダプタごとに 1 スレッドが動作します。

   コネクション数調整スレッドは，TP1 インバウンドアダプタの開始時に作成し，終了時に消滅します。

   一時クローズ処理については，「4.6.6　一時クローズ処理によるコネクション数の調整」を参照してください。

コネクション管理機能は，OpenTP1 とアプリケーションサーバ間の通信で必要となる，コネクションに関する次の処理を制御します。

1. コネクション接続要求の受信
2. 電文の受信
3. コネクション接続要求の送信
4. 電文の送信
5. 一時クローズ処理によるコネクション数の調整
6. コネクション接続要求または電文送信失敗時のリトライ

4.6.2　コネクション接続要求の受信

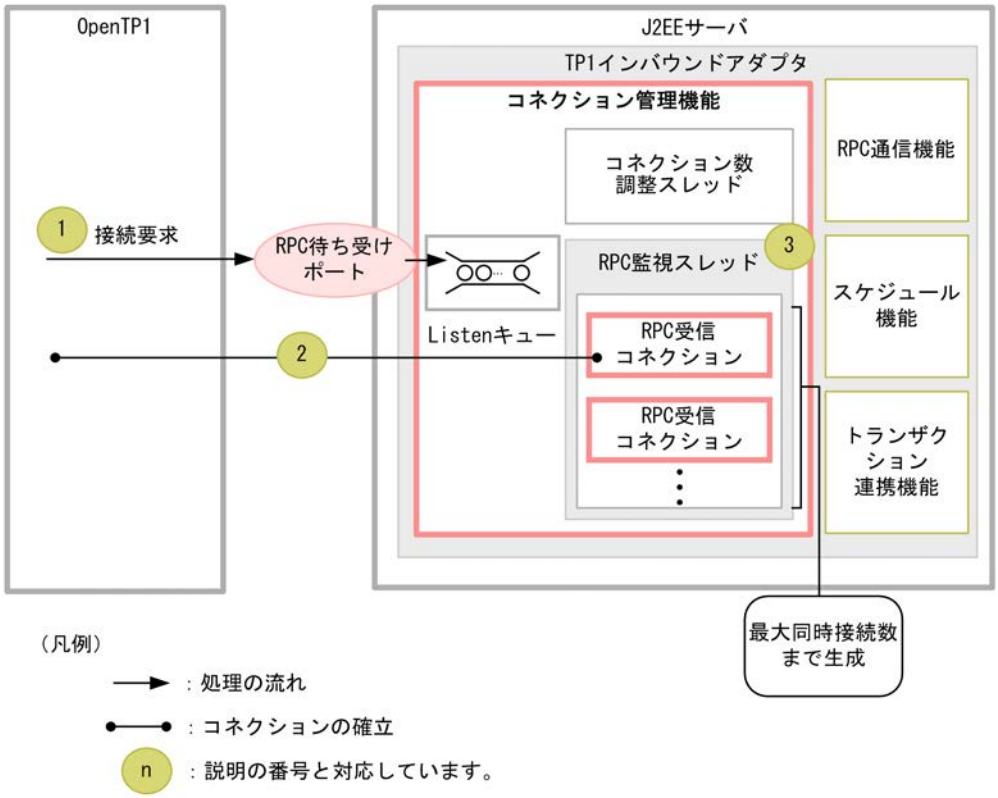
OpenTP1 と TP1 インバウンドアダプタ間で通信する際，接続済みのコネクションがあれば利用し，なければ OpenTP1 から TP1 インバウンドアダプタへコネクション接続要求を送信してコネクションを接続します。

OpenTP1 からのコネクションの接続要求は，RPC 待ち受けポート，または同期点待ち受けポートで受け取ります。それぞれのポートが，OpenTP1 からのコネクションの接続要求を受け取ったときの動作，およびコネクション接続要求に関する設定について説明します。

(1)　RPC 待ち受けポートがコネクション接続要求を受信した際の動作

TP1 インバウンドアダプタの RPC 待ち受けポートが，コネクション接続要求を受信した際の動作を次に示します。

図 4-3　RPC 待ち受けポートがコネクション接続要求を受信した際の動作

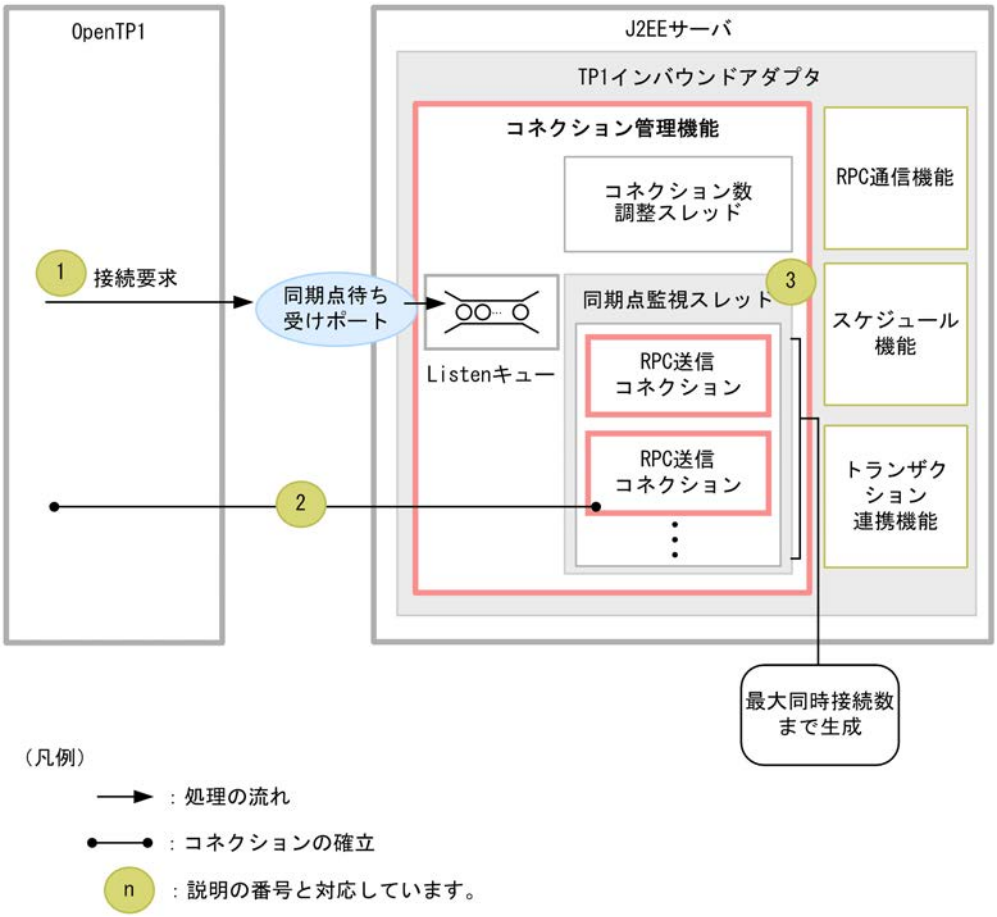


1. OpenTP1 が RPC 待ち受けポートに対してコネクション接続要求を送信します。コネクション接続要求は，TP1 インバウンドアダプタが動作する OS の Listen キュー（backlog）に格納されます。
2. TP1 インバウンドアダプタのコネクション管理機能では，RPC 監視スレッドが Listen キューのコネクション接続要求を受信し，OpenTP1 とのコネクションを確立します。
3. RPC 監視スレッドは，確立したコネクションが OpenTP1 から電文を受信しているかどうか監視を始めます。

(2)　同期点待ち受けポートがコネクション接続要求を受信した際の動作

TP1 インバウンドアダプタの同期点待ち受けポートが，コネクション接続要求を受信した際の動作を次に示します。

図 4-4　同期点待ち受けポートがコネクション接続要求を受信した際の動作



OpenTP1 の同期点処理は，RPC 応答を受信したコネクションを使用します。このコネクションが切断されていた際に，TP1 インバウンドアダプタの同期点待ち受けポートに OpenTP1 がコネクション接続要求をします。

1. OpenTP1 が，同期点待ち受けポートに対してコネクション接続要求を送信します。コネクション接続要求は，TP1 インバウンドアダプタが動作する OS の Listen キュー（backlog）に格納されます。
2. TP1 インバウンドアダプタのコネクション管理機能では，同期点監視スレッドが Listen キューのコネクション接続要求を受信し，OpenTP1 とのコネクションを確立します。
3. 同期点監視スレッドは，確立したコネクションが OpenTP1 から電文を受信しているかどうか監視を始めます。

(3)　コネクション接続要求に関する設定

コネクション接続要求に関する設定のうち，Connector 属性ファイルに指定する項目は次のとおりです。

- Listen キューの長さ
  - backlog_count プロパティ
- コネクション接続要求を受信するポート番号
  - scd_port プロパティ
  - trn_port プロパティ

- コネクションの最大同時接続数
  - max_connections プロパティ
  - trn_max_connections プロパティ

指定方法の詳細については，「4.12.2　リソースアダプタの設定」を参照してください。

コネクション接続要求に関する設定のうち，IP アドレス（バインド先アドレス）については，usrconf.properties（J2EE サーバ用ユーザプロパティファイル）の ejbserver.jca.adapter.tp1.bind_host プロパティに設定します。

バインド先アドレスを指定すると，TP1 インバウンドアダプタが動作するホストが複数の物理ネットワークインタフェースを持つ場合に，特定の IP アドレスだけを使用するように設定できます。この設定をしない場合は，システムによって自動的に選択されたローカルアドレスが使用されます。指定方法の詳細については，「4.12.1　J2EE サーバの設定」を参照してください。

(4)　コネクション接続要求に失敗する原因および OpenTP1 での動作

TP1 インバウンドアダプタが次に示す条件に当てはまる場合，OpenTP1 からのコネクション接続要求に失敗します。コネクション接続要求に失敗する原因，および OpenTP1 での動作について説明します。

表 4-5　コネクション接続要求に失敗する原因および OpenTP1 での動作

コネクション接続要求に失敗する原因	OpenTP1 での動作
• TP1 インバウンドアダプタが起動していない • TP1 インバウンドアダプタで通信障害が発生している	OpenTP1 の dc_rpc_call 関数，または同期点処理がエラーを返します。※
OS の Listen キューにコネクション接続要求が上限まで滞留している状態で，OpenTP1 が新たなコネクション接続要求を送信した（Listen キューあふれ）	新たなコネクション接続要求は，OS の Listen キューによって拒否されます。OpenTP1 の dc_rpc_call 関数，または同期点処理がエラーを返します。※
コネクションの接続数がコネクション管理機能の最大同時接続数の設定値に達している	コネクション接続要求は，コネクションの接続数が最大同時接続数未満になるまで OS の Listen キューに滞留します。OpenTP1 の dc_rpc_call 関数，または同期点処理は最大応答待ち時間に達してからエラーを返します。※

注※

エラーコードについては，「4.17　TP1 インバウンドアダプタで発生する RPC エラー応答」を参照してください。

4.6.3　電文の受信

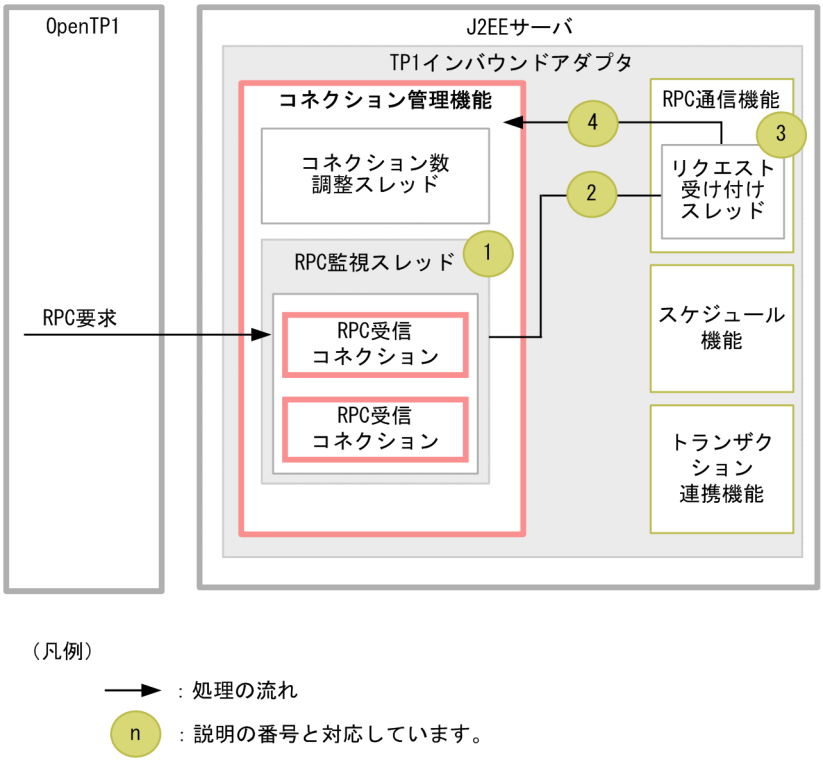
OpenTP1 からの接続要求を受け，TP1 インバウンドアダプタで電文を受信します。

TP1 インバウンドアダプタが OpenTP1 から受信する電文は，RPC 要求，および同期点処理です。それぞれの電文を受信したときの動作について説明します。

(1)　RPC 要求受信時の動作

TP1 インバウンドアダプタが RPC 要求を受信したときの動作を次に示します。

図 4-5　RPC 要求受信時の動作

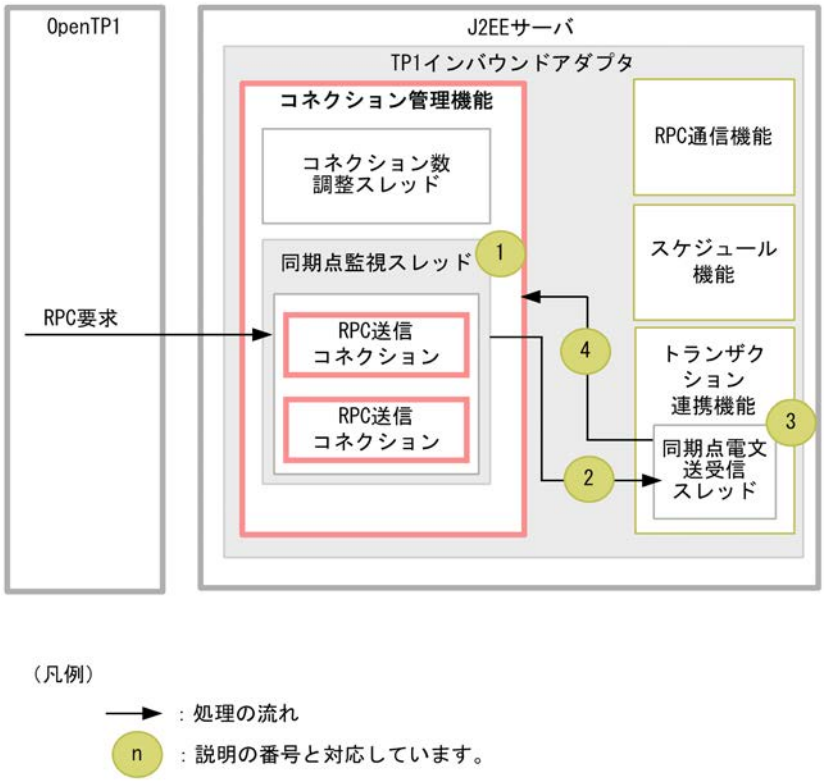


1. コネクション管理機能の RPC 監視スレッドは，電文受信中でない RPC 受信コネクションが OpenTP1 からの電文を受信していないかを監視します。
2. RPC 監視スレッドが OpenTP1 からの電文の受信を検知します。RPC 監視スレッドは該当するコネクションで電文の受信が開始したことを RPC 通信機能に通知します。RPC 監視スレッドは，ほかの RPC 受信コネクションが OpenTP1 からの電文を受信していないかどうか監視を続けます。RPC 通信機能の詳細は，「4.7　RPC 通信機能」を参照してください。
3. RPC 通信機能のリクエスト受け付けスレッドは，電文の受信開始が通知されたコネクションから，順次 OpenTP1 からの電文を受信します。
4. リクエスト受け付けスレッドは，OpenTP1 からの電文の受信を完了すると，該当するコネクションでの受信が完了したことをコネクション管理機能に通知します。これによって，RPC 受信監視スレッドは該当するコネクションで次の電文を受信していないかどうかの監視を始めます。

(2)　同期点処理の受信時の動作

TP1 インバウンドアダプタが同期点処理を受信したときの動作を次に示します。

図 4-6　同期点処理の受信時の動作



1. コネクション管理機能の同期点監視スレッドは，電文受信中でない RPC 送信コネクションが OpenTP1 からの電文を受信していないかどうか監視します。
2. 同期点監視スレッドが OpenTP1 からの電文の受信を検知すると，同期点監視スレッドは該当するコネクションで電文の受信が開始したことをトランザクション管理機能に通知します。同期点監視スレッドは，ほかの RPC 送信コネクションが OpenTP1 からの電文を受信していないかどうかの監視を続けます。詳細は，「4.9　トランザクション連携機能」を参照してください。
3. トランザクション連携機能の同期点電文送受信スレッドは，電文の受信開始が通知されたコネクションから，順次 OpenTP1 からの電文を読み込みます。
4. 同期点電文送受信スレッドは，OpenTP1 からの電文の受信を完了すると，使用しているコネクションでの受信が完了したことをコネクション管理機能に通知します。これによって，同期点監視スレッドは該当するコネクションで次の電文を受信していないかどうかの監視を始めます。

(3)　電文の受信に関する設定

電文の受信に関する設定は，Connector 属性ファイルに指定します。指定する項目は次のとおりです。

- 電文受信時の TCP/IP のバッファサイズ
  - receive_buffer_size プロパティ
- 受信タイムアウト時間
  - tcp_receive_timeout プロパティ

指定方法の詳細については，「4.12.2　リソースアダプタの設定」を参照してください。

4.6.4　コネクション接続要求の送信

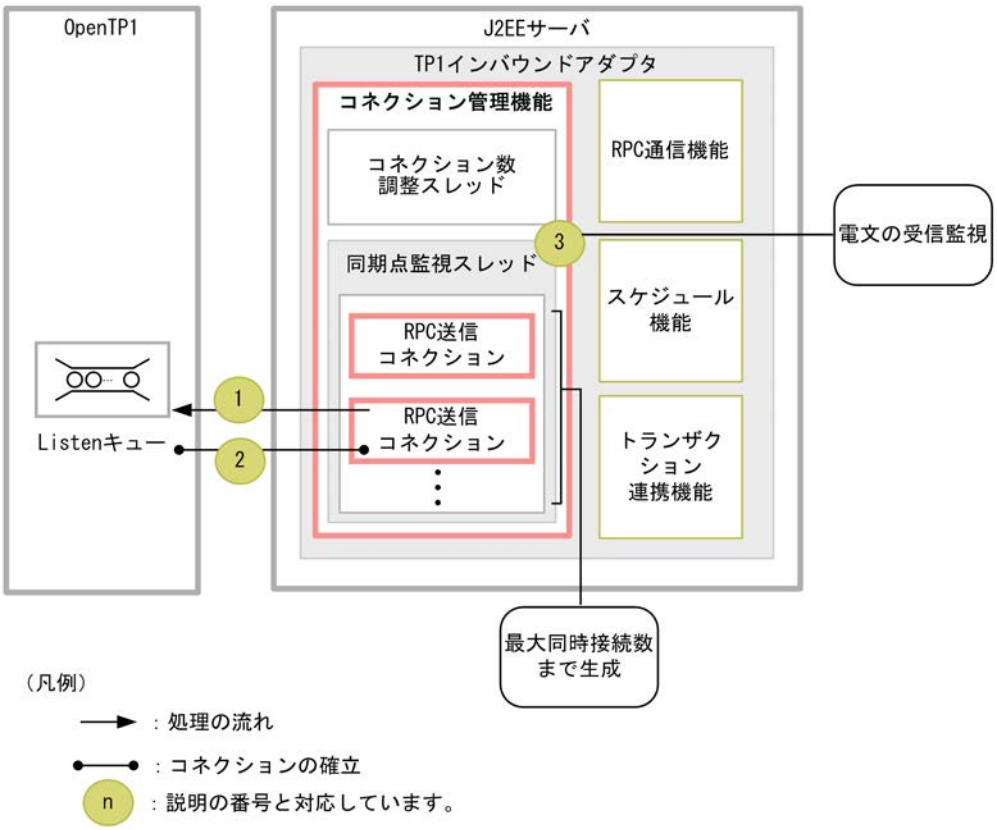
TP1 インバウンドアダプタが OpenTP1 と通信する際に，送信先の OpenTP1 と接続済みのコネクションがある場合は，接続済みのコネクションを利用します。接続済みのコネクションがない場合は，コネクション接続要求を送信してコネクションを接続します。

TP1 インバウンドアダプタから OpenTP1 へのコネクションの接続要求には，RPC 応答，および同期点処理の 2 種類があります。それぞれのコネクション接続要求を送信したときの動作について説明します。

(1)　RPC 応答および同期点処理でのコネクション接続要求送信時の動作

RPC 応答および同期点処理でのコネクション接続要求送信時の動作について，次の図に示します。

図 4–7　RPC 応答および同期点処理でのコネクション接続要求送信時の動作



それぞれの動作の流れについて，(a)および(b)で説明します。

(a)　RPC 応答でのコネクション接続要求送信時の動作

RPC 応答でのコネクション接続要求は，RPC 通信機能のリクエスト受け付けスレッド，スケジュール機能の Message-driven Bean（サービス）呼び出しスレッド，および Message-driven Bean（サービス）の停止処理を実行するスレッドが送信します。コネクション接続要求を OpenTP1 に送信する際の流れを次に示します。

1. TP1 インバウンドアダプタから OpenTP1 へコネクション接続要求を送信します。コネクション接続要求は，OpenTP1 が動作する OS の Listen キュー（backlog）に格納されます。
2. OpenTP1 が Listen キューのコネクション接続要求を受信し，TP1 インバウンドアダプタとのコネクションを確立します。

3. コネクションが確立したら，同期点監視スレッドは，OpenTP1 への電文の送信と並行して OpenTP1 からの電文を受信できるように RPC 送信コネクションが OpenTP1 から電文を受信しているかどうかの監視を始めます。

(b)　同期点処理でのコネクション接続要求送信時の動作

同期点処理でのコネクション接続要求は，トランザクション連携機能の同期点電文送受信スレッドが送信します。コネクション接続要求を OpenTP1 に送信する際の流れを次に示します。

1. TP1 インバウンドアダプタから OpenTP1 へコネクション接続要求を送信します。コネクション接続要求は，OpenTP1 が動作する OS の Listen キュー（backlog）に格納されます。
2. OpenTP1 が，Listen キューのコネクション接続要求を受信し，TP1 インバウンドアダプタとのコネクションを確立します。
3. コネクションが確立したら，同期点監視スレッドは，OpenTP1 への電文の送信と並行して OpenTP1 からの電文を受信できるように，RPC 送信コネクションが OpenTP1 から電文を受信しているかどうかの監視を始めます。

(2)　コネクション接続要求の送信に関する設定

コネクション接続要求の送信に関する設定は，Connector 属性ファイルに指定します。指定する項目は次のとおりです。

- コネクションの最大同時接続数
  - max_connections プロパティ
  - trn_max_connections プロパティ
- コネクション接続失敗時のリトライ
  - send_retry_count プロパティ
  - send_retry_interval プロパティ
- コネクション接続時のタイムアウト時間
  - connection_timeout プロパティ

指定方法の詳細については，「4.12.2　リソースアダプタの設定」を参照してください。

(3)　コネクション接続要求に失敗する原因および TP1 インバウンドアダプタでの動作

OpenTP1 が，TP1 インバウンドアダプタからのコネクション接続要求の受信に失敗する原因，および TP1 インバウンドアダプタでの動作を次に示します。

表 4-6　コネクション接続要求に失敗する原因および TP1 インバウンドアダプタでの動作

コネクション接続要求に失敗する原因	TP1 インバウンドアダプタでの動作
• OpenTP1 に障害が発生している • 通信障害が発生している	TP1 インバウンドアダプタは，コネクション接続要求をリトライします。※1 リトライしてもコネクションを接続できなかった場合は，エラーメッセージ（KDJE58400-E）を出力して未送信の電文を破棄します。
OS の Listen キューにコネクション接続要求が上限まで滞留している状態で，TP1 インバウンドアダプタが新たなコネクション接続要求を送信した（Listen キューあふれ）	
TCP 送信コネクション接続タイムアウトが発生した	TP1 インバウンドアダプタは，エラーメッセージ（KDJE58400-E）を出力して未送信の電文を破棄します。

コネクション接続要求に失敗する原因	TP1 インバウンドアダプタでの動作
コネクションの接続数が最大同時接続数に達している	TP1 インバウンドアダプタは，コネクションの接続数が最大同時接続数未満になるまでコネクション接続要求の送信を待機します。※2

注※1

コネクション接続要求のリトライについては，「4.12.2　リソースアダプタの設定」を参照してください。

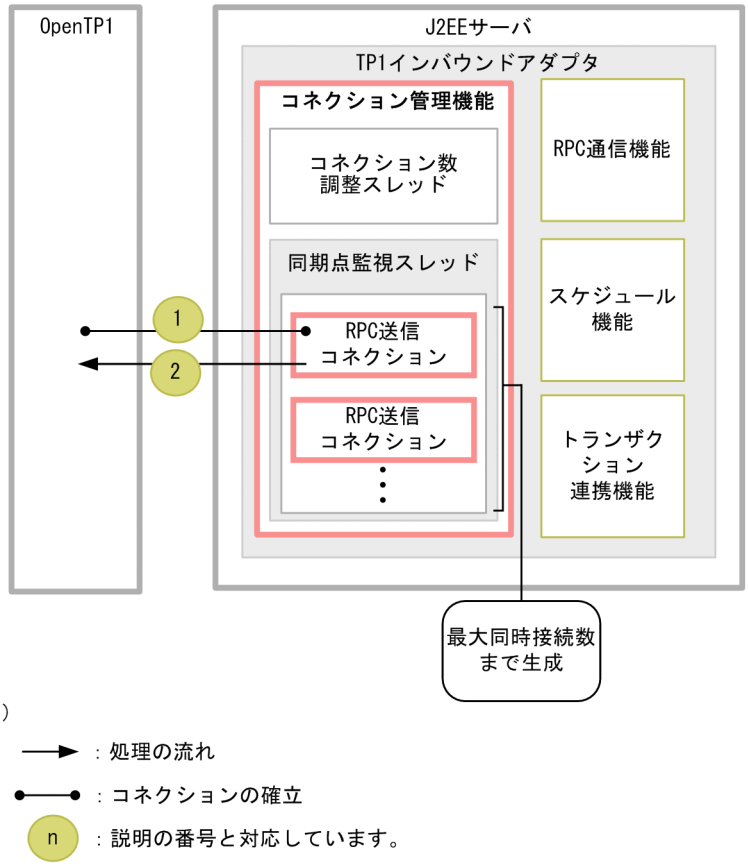
注※2

どのコネクションからも一時クローズ処理の応答が返ってこない場合に，コネクションを切断できなかったとき，指定した一時クローズ応答の待ち時間が経過すると，TP1 インバウンドアダプタは一時クローズ要求の応答を待つコネクションを強制的に切断します。このため，コネクション接続要求の送信を待ち続けることはありません。

4.6.5　電文の送信

TP1 インバウンドアダプタが OpenTP1 へ電文を送信する際の動作を次に示します。

図 4-8　電文の送信する流れ



(1)　RPC 応答と同期点処理送信時の動作

RPC 応答は，RPC 通信機能のリクエスト受け付けスレッド，受信タイマ監視スレッド，スケジュール機能の Message-driven Bean（サービス）呼び出しスレッド，および Message-driven Bean（サービス）の

停止処理を実行するスレッドが送信します。同期点処理は，トランザクション連携機能の同期点電文送受信スレッドが送信します。

次に RPC 応答と同期点処理を OpenTP1 に送信するときの流れを示します。

1. OpenTP1 へ電文を送信する際，コネクション管理機能から送信先の OpenTP1 とのコネクションを取得します。
2. コネクションを取得したスレッドは，取得したコネクションを使用して OpenTP1 へ電文を送信します。

(2)　電文の送信時の設定

電文の送信時の設定は，Connector 属性ファイルに指定します。指定する項目は次のとおりです。

- 電文送信時の TCP/IP のバッファサイズ
  - send_buffer_size プロパティ
- 送信リトライ
  - send_retry_count プロパティ
  - send_retry_interval プロパティ
- 送信タイムアウト時間
  - tcp_send_timeout プロパティ
- TCP_NODELAY オプション
  - ipc_tcpnodelay プロパティ

指定方法の詳細については，「4.12.2　リソースアダプタの設定」を参照してください。

4.6.6　一時クローズ処理によるコネクション数の調整

OpenTP1 と TP1 インバウンドアダプタでは，コネクション接続時のオーバーヘッドを削減するために，一度接続したコネクションを切断しないで次回の通信で再利用します。そのため，コネクション管理機能の最大同時接続数の設定よりも多くの接続を受け付けるシステムでは，コネクションの接続数がすぐに最大同時接続数に達してしまいます。

これを防ぐために，コネクションの接続数が一定のしきい値に達したところで未使用のコネクションを切断する機能（一時クローズ処理）を使用します。

一時クローズ処理によってコネクション数を調整することで，コネクションを枯渇させることなく，効率良くコネクションを管理できます。

一時クローズ処理は，OpenTP1 側から実行する場合と，TP1 インバウンドアダプタ側から実行する場合の 2 種類があります。どちらの場合も，接続先に一時クローズ処理要求を送信して，接続先から一時クローズ処理応答を受信したタイミングでコネクションを切断します。一時クローズ処理要求の送信と一時クローズ処理の応答の受信は，非同期に処理されます。

なお，OpenTP1 と TP1 インバウンドアダプタ間のコネクションが一時クローズ処理中であっても，新しい RPC 要求を受け付けることができます。また，新たなコネクションを接続することで RPC 応答を送信することができるため，一時クローズ処理によって OpenTP1 や TP1 インバウンドアダプタでの処理がエラーになることはありません。

(1)　一時クローズ処理対象とするコネクション数の指定

処理開始のしきい値には，TP1 インバウンドアダプタが一時クローズ処理を開始するコネクション数の割合をパーセンテージで指定します。指定したしきい値に応じて，コネクションの種類ごとに次の条件で一時クローズ処理を開始します。

RPC 受信コネクションの一時クローズ処理開始の条件

RPC 受信コネクション数 ≧
RPC 受信コネクションの最大同時接続数 ×
RPC 受信コネクションの一時クローズ処理開始のしきい値 ／ 100

RPC 送信コネクションの一時クローズ処理開始の条件

RPC 送信コネクション数 ≧
RPC 送信コネクションの最大同時接続数 ×
RPC 送信コネクションの一時クローズ処理開始のしきい値 ／ 100

(2)　一時クローズ処理の対象外とするコネクション数の指定

一時クローズ処理の対象外とするコネクション数の割合をパーセンテージで指定します。

RPC 受信コネクションの一時クローズ処理非対象コネクション数の算出

RPC 受信コネクションの一時クローズ処理非対象コネクション数＝
RPC 受信コネクションの最大同時接続数 ×
RPC 受信コネクションの一時クローズ処理非対象とするコネクション数の割合／100

RPC 送信コネクションの一時クローズ処理非対象コネクション数の算出

RPC 送信コネクションの一時クローズ処理非対象コネクション数＝
RPC 送信コネクションの最大同時接続数 ×
RPC 送信コネクションの一時クローズ処理非対象とするコネクション数の割合／100

(3)　コネクション数が最大に達した際の一時クローズ処理の応答待ち時間の指定

一時クローズ処理では，コネクションを接続した OpenTP1 と TP1 インバウンドアダプタ間で，どちらかから一時クローズ処理の要求を送信したあと，その応答を受信してからコネクションを切断します。一時クローズ処理の応答を受信するまでコネクションを切断することはできません。

一時クローズ処理要求後にどのコネクションからも応答が返ってこなければ，コネクション数に空きができないため，新たなコネクションを接続することができなくなります。これを防ぐため，TP1 インバウンド連携機能では，一時クローズ応答の待ち時間を指定することで，コネクション数が最大に達してからコネクション数に空きができるまでの時間を監視します。指定した一時クローズ応答の待ち時間を経過してもコネクション数が最大値のままの場合，エラーメッセージを出力してコネクションを切断します。

RPC 受信コネクション，および送信コネクションでのコネクション数の監視について説明します。

(a)　RPC 受信コネクションでのコネクション数の監視

RPC 受信コネクションが最大 RPC 受信コネクション数に達してから，一時クローズ処理によってコネクション数に空きができるまでの時間を監視します。

指定した一時クローズ応答の待ち時間を経過しても，コネクション数が最大 RPC 受信コネクション数のままの場合は，エラーメッセージ（KDJE58501-E）を出力し，一時クローズ処理の応答を待っている RPC 受信コネクションをすべて切断します。

(b)　RPC 送信コネクションでのコネクション数の監視

RPC 受信コネクション同様に，RPC 送信コネクションが最大 RPC 送信コネクション数に達してから，一時クローズ処理によってコネクション数に空きができるまでの時間を監視します。

指定した一時クローズ応答の待ち時間を経過しても，コネクション数が最大 RPC 送信コネクション数のままの場合は，エラーメッセージ（KDJE58501-E）を出力し，一時クローズ処理の応答を待っている RPC 送信コネクションをすべて切断します。

(4)　一時クローズ処理に関する設定

一時クローズ処理に関する設定は，Connector 属性ファイルに指定します。指定する項目は次のとおりです。

- RPC 受信コネクションの一時クローズ処理
  - rpc_sockctl_highwater プロパティ
  - rpc_sockctl_lowwater プロパティ
- RPC 送信コネクションの一時クローズ処理
  - trn_sockctl_highwater プロパティ
  - trn_sockctl_lowwater プロパティ
- 一時クローズ応答の待ち時間
  - ipc_sockctl_watchtime プロパティ

指定方法の詳細については，「4.12.2　リソースアダプタの設定」を参照してください。

4.6.7　コネクション接続要求または電文送信失敗時のリトライ

コネクション管理機能では，OpenTP1 へのコネクション接続要求に失敗した場合，または電文送信に失敗した場合に，リトライします。コネクション接続要求に失敗した場合は，再度コネクション接続要求を試みます。電文送信に失敗した場合は，コネクションを切断して，新たなコネクションを接続してから再度送信を試みます。

ただし，コネクション接続要求時または電文送信時にタイムアウトが発生した場合は，リトライしません。コネクションのタイムアウトについては「4.14　OpenTP1 とアプリケーションサーバ間のタイムアウトの設定」を参照してください。

(1)　リトライ時に出力されるメッセージ

コネクション管理機能では，初回のリトライ時にだけ警告メッセージを出力します。リトライ時に出力する警告メッセージについて次に示します。

表 4-7　リトライ時に出力する警告メッセージ

リトライの種別	メッセージ
コネクション接続要求のリトライ	KDJE58402-W
電文送信のリトライ	KDJE58403-W

コネクション接続要求が設定したリトライ回数分失敗した場合，未送信の電文を破棄します。

また，電文送信のリトライが設定したリトライ回数分失敗した場合，コネクションを切断して未送信の電文を破棄します。これらの場合，最後のリトライの種別に応じて次のエラーメッセージを出力します。

表 4-8　リトライ回数までリトライしても失敗した場合のエラーメッセージ

リトライの種別	メッセージ
コネクション接続要求のリトライ	KDJE58400-E
電文送信のリトライ	KDJE58401-E

(2)　コネクション接続要求または電文送信失敗時のリトライに関する設定

コネクション接続要求または電文送信失敗時のリトライの設定は，Connector 属性ファイルに指定します。指定する項目は次のとおりです。

- コネクション接続要求または電文送信失敗時のリトライ
  - send_retry_count プロパティ
  - send_retry_interval プロパティ

指定方法の詳細については，「4.12.2　リソースアダプタの設定」を参照してください。

4.7　RPC 通信機能

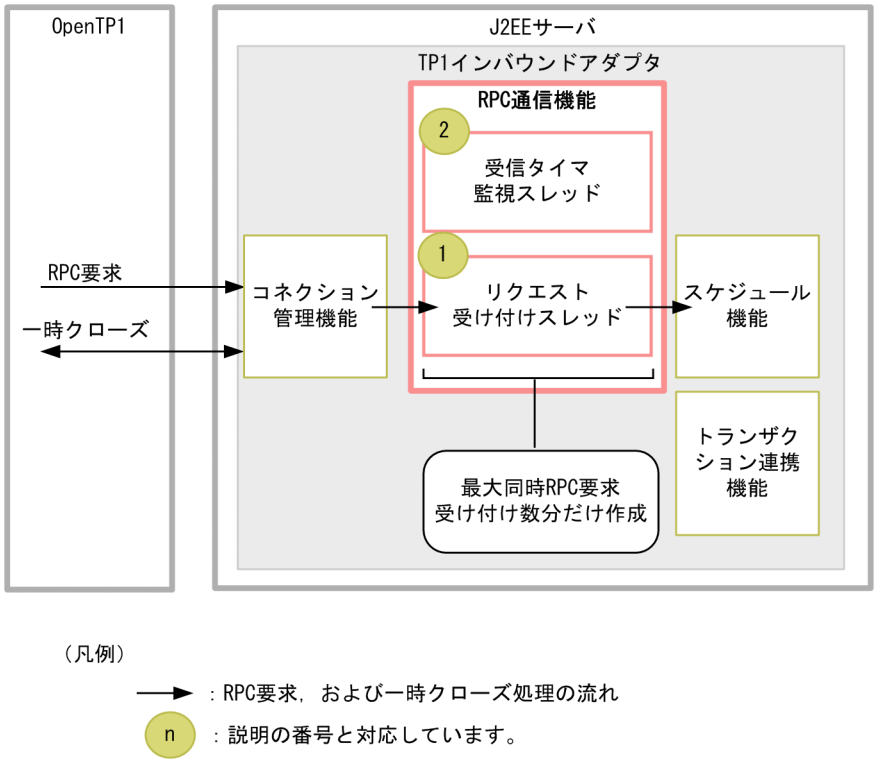
この節では，RPC 通信機能について説明します。

4.7.1　RPC 通信機能の概要

RPC 通信機能は，OpenTP1 から受信した RPC 要求を解析・組み立てし，OpenTP1 に送信する RPC 応答を生成・分割する機能です。

TP1 インバウンドアダプタでの RPC 通信機能の位置づけを次の図に示します。

図 4-9　RPC 通信機能の位置づけ



（凡例）

：RPC要求，および一時クローズ処理の流れ

n　：説明の番号と対応しています。

RPC 通信機能は，次の要素で構成されます。

1. **リクエスト受け付けスレッド**

   OpenTP1 からの RPC 要求を受け付けるスレッドです。

   TP1 インバウンドアダプタの開始時に最大同時接続数分だけ作成し，TP1 インバウンドアダプタ終了時に消滅します。最大同時接続数の設定については，「4.7.5　最大同時 RPC 要求受け付け数および受信タイムアウト」で説明します。

2. **受信タイマ監視スレッド**

   RPC 要求の受信タイムアウトを監視するスレッドです。TP1 インバウンドアダプタごとに 1 スレッドが動作します。

   受信タイマ監視スレッドは，TP1 インバウンドアダプタの開始時に作成し，終了時に消滅します。

RPC 通信機能では，OpenTP1 とアプリケーションサーバ間で RPC 通信によって電文を送受信するための次の処理を制御します。

- RPC 要求の受信と組み立て
- RPC 要求のチェック
- RPC 応答の生成から送信まで
- 最大同時 RPC 要求受け付け数および受信タイムアウト

OpenTP1 では，電文を分割して送受信します。このため，TP1 インバウンドアダプタでは，RPC 通信の際に，受信した電文の組み立ておよび送信する電文の分割処理を実行します。

なお，RPC 要求の受信と RPC 応答の送信には，別のコネクションを使用します。

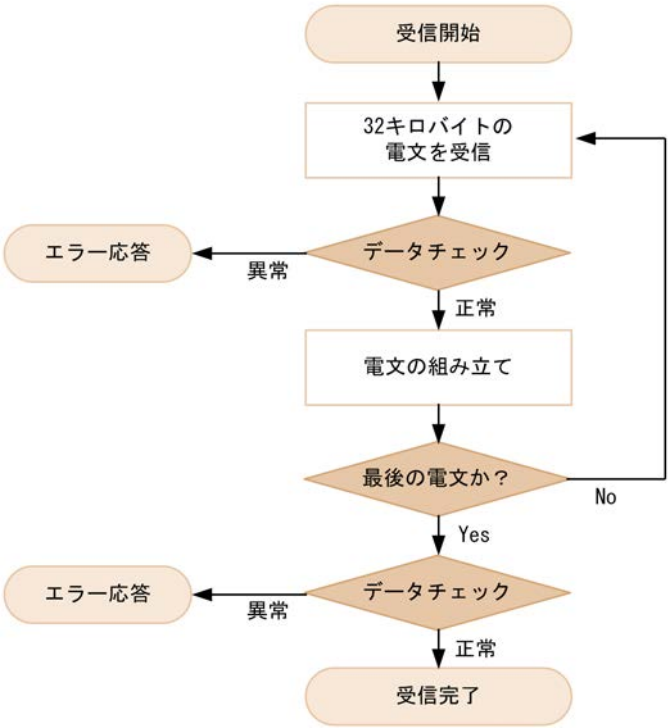
4.7.2　RPC 要求の受信と組み立て

コネクションの確立後，OpenTP1 からの RPC 要求が，32 キロバイトずつに分割された電文として送信されます。

RPC 通信機能のリクエスト受け付けスレッドは，コネクション管理機能によって割り当てられたコネクションを使用して，分割された電文を順次受信し，一つの要求電文として組み立てます。

電文の受信と組み立ての流れを次の図に示します。

図 4–10　電文の受信と組み立ての流れ



TP1 インバウンドアダプタでは，分割された電文を受信すると，データチェックを実行します。データチェックで正常と判定した電文について，組み立てを実行します。組み立て後，次の電文を受信します。最後の電文の場合は，最後のデータチェックを実行し，電文受信を終了します。なお，データチェックで電文の異常を検知した場合は，エラー応答を送信します。

要求電文の受信中に，ネットワーク障害や OpenTP1 の障害などによってコネクションが切断された場合，TP1 インバウンドアダプタはエラーメッセージ（KDJE58354-E）を出力して電文の受信処理を中断します。なお，OpenTP1 では，コネクションの切断を検知すると，再度コネクションを確立して，送信の途中

だった電文の送信をリトライします。TP1 インバウンドアダプタでは，OpenTP1 からのリトライに備え，受信済みの電文を**電文組み立てリスト**に保持します。電文組み立てリストは，最大同時接続数分保持できます。このリストは，残りの電文をすべて受信し終えるか，または RPC 要求の受信タイムアウトが発生するまで保持されます。

コネクション障害が頻発することによって電文組み立てリストが蓄積して電文の組み立てができなくなると，新たな RPC 要求の受信ができなくなります。この場合，エラーメッセージ（KDJE58359-E）が出力されます。また，呼び出し元の dc_rpc_call 関数には RPC エラー（トランザクショナル RPC でない場合は DCRPCER_NO_BUFS(-304)，トランザクショナル RPC の場合は DCRPCER_NET_DOWN(-323)）が返されます。

電文組み立てリストが不足した状態は，OpenTP1 から残りの電文がすべて再送されるか，または RPC 要求の受信タイムアウトが発生すると，解消します。

RPC 要求の受信時のエラー，および電文組み立てリストが不足した場合のエラーについては，「4.17　TP1 インバウンドアダプタで発生する RPC エラー応答」を参照してください。

また，最大同時接続数，および要求電文の受信処理のタイムアウトの設定は，Connector 属性ファイルに指定します。指定する項目は次のとおりです。

- 最大同時接続数
  - max_connections プロパティ
- 要求電文の受信処理のタイムアウト
  - rpc_receive_timeout プロパティ

指定方法の詳細については，「4.12.2　リソースアダプタの設定」を参照してください。

4.7.3　RPC 要求のチェック

RPC 通信機能では，受信した電文の形式と内容をチェックします。チェック内容および不正を検知した場合の動作を次に示します。

- **電文形式のチェック**

  受信した RPC 要求が OpenTP1 から送信された電文かどうかをチェックします。

  不正を検知した場合は，エラーメッセージ（KDJE58355-E）を出力して，コネクションを切断します。応答送信はしません。エラーの詳細については，「4.17　TP1 インバウンドアダプタで発生する RPC エラー応答」を参照してください。

- **TP1 インバウンドアダプタで使用できない機能を使用していないかのチェック**

  TP1 インバウンドアダプタで使用できない OpenTP1 の機能を使用していないかどうかをチェックします。TP1 インバウンドアダプタで使用できる機能については，「4.17　TP1 インバウンドアダプタで発生する RPC エラー応答」を参照してください。

  不正を検知した場合は，エラーメッセージ（KDJE58356-E）を出力して，コネクションを切断します。また，呼び出し元の dc_rpc_call 関数に RPC エラーを応答します。エラーの詳細については，「4.17　TP1 インバウンドアダプタで発生する RPC エラー応答」を参照してください。

- **dc_rpc_call 関数の引数チェック**

  呼び出し元の dc_rpc_call 関数の引数が適切に指定されているかどうかをチェックします。

  不正を検知した場合は，エラーメッセージを出力して，コネクションを切断します。また，呼び出し元の dc_rpc_call 関数に RPC エラーを応答します。

引数チェックでチェックされる内容，不正検知時に出力されるエラーメッセージ，および RPC エラー応答について次の表に示します。

表 4-9　dc_rpc_call 関数の引数チェック

引数	チェック内容	不正検知時のエラーメッセージ	不正検知時の RPC エラー応答
group	引数に指定したサービスグループ名が，TP1 インバウンドアダプタのサービスグループ名（Connector 属性ファイルに指定した service_group プロパティ）と一致しているか。	KDJE58357-E	DCRPCER_SERVICE_NOT_UP(-314)
service	引数に指定したサービス名が，TP1 インバウンドアダプタが呼び出すサービス名（Message-driven Bean（サービス）の ActivationSpec の service プロパティ）の中に存在するか。	KDJE58362-E	DCRPCER_NO_SUCH_SERVICE(-311)
flags	引数に指定した RPC 形態が，TP1 インバウンドアダプタで利用できる次のどちらかの形態か。 • DCNOFLAGS（同期応答型 RPC） • DCNOFLAGS|DCRPC_TPNOTRAN（同期応答型，かつトランザクションを引き継がない RPC）	KDJE58356-E	DCRPCER_TIMED_OUT(-307)

エラーの詳細については，「4.17　TP1 インバウンドアダプタで発生する RPC エラー応答」を参照してください。

4.7.4　RPC 応答の生成から送信まで

ここでは，RPC 応答の生成から送信までの処理について説明します。

(1)　応答電文生成

RPC 通信機能が呼び出し元に送信する電文を生成します。

(2)　応答電文の分割と送信

RPC 通信機能では，一つの RPC 応答電文を 32 キロバイトごとに分割して順次送信します。

送信にはコネクション管理機能のコネクションを使用します。応答電文の送信処理には，リトライやタイムアウトを設定できます。設定方法については，「4.14　OpenTP1 とアプリケーションサーバ間のタイムアウトの設定」を参照してください。

4.7.5　最大同時 RPC 要求受け付け数および受信タイムアウト

最大同時 RPC 要求受け付け数および受信タイムアウトを指定する項目は次のとおりです。

- OpenTP1 から受信した RPC 要求の最大同時 RPC 要求受け付け数
  - rpc_max_thread_count プロパティ

Connector 属性ファイルの指定方法の詳細については，「4.12.2　リソースアダプタの設定」を参照してください。

- RPC 要求受信のタイムアウト時間

  詳細については，「4.14　OpenTP1 とアプリケーションサーバ間のタイムアウトの設定」を参照してください。

4.8　スケジュール機能

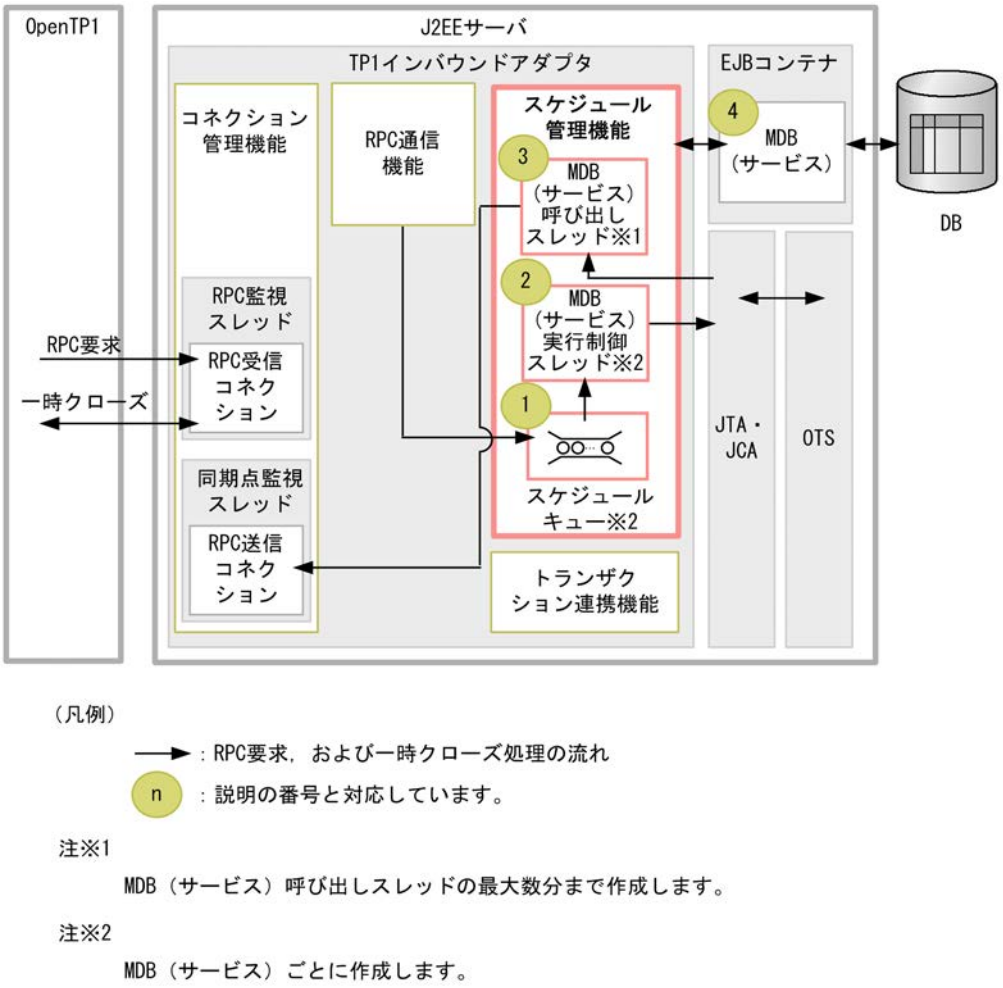
この節では，スケジュール機能について説明します。

4.8.1　スケジュール機能の概要

スケジュール機能は，RPC 通信機能によって受信した RPC 要求電文を J2EE サーバで効率良く処理するための機能です。TP1/Server Base のスケジュールサービスの一部の機能に相当します。

TP1 インバウンドアダプタでのスケジュール機能の位置づけを次の図に示します。

図 4-11　スケジュール通信機能の位置づけ



スケジュール機能は，次の要素で構成されます。

1. スケジュールキュー

   RPC 通信機能によって受信した RPC 要求を保持するキューです。RPC 要求は，FIFO 方式で保持します。

   スケジュールキューは，Message-driven Bean（サービス）開始時に Message-driven Bean（サービス）ごとに一つ作成され，Message-driven Bean（サービス）の終了時に削除されます。

2. Message-driven Bean（サービス）実行制御スレッド

スケジュールキューから RPC 要求を取り出して，空いている Message-driven Bean（サービス）呼び出しスレッドに処理を委譲します。

Message-driven Bean（サービス）実行制御スレッドは，Message-driven Bean（サービス）開始時に，Message-driven Bean（サービス）ごとに一つ作成され，Message-driven Bean（サービス）の終了時に削除されます。

3. Message-driven Bean（サービス）呼び出しスレッド

   Message-driven Bean（サービス）を実行するスレッドです。Connector 1.5 仕様に準拠した WorkManager によって管理されます。

   Message-driven Bean（サービス）実行制御スレッドが RPC 要求をスケジュールキューから取り出して受け付けたときに WorkManager によって割り当てられます。

   RPC 応答後は，Message-driven Bean（サービス）呼び出しスレッドは WorkManager に返されます。WorkManager が管理するスレッドは，TP1 インバウンドアダプタを開始したときに生成され，TP1 インバウンドアダプタ終了時に削除されます。

   スレッドの詳細については，「4.8.3　Message-driven Bean（サービス）の同時実行数の制御」を参照してください。

4. Message-driven Bean（サービス）

   業務処理を実行するサービスとなる Message-driven Bean です。OpenTP1 の SPP に相当します。

スケジュール機能の処理の流れを次に示します。

1. Message-driven Bean（サービス）を開始した時に，Message-driven Bean 単位にスケジュールキューおよび Message-driven Bean（サービス）実行制御スレッドが作成されます。
2. RPC 通信機能での RPC 要求受信後，受信した RPC 要求はスケジュールキューに格納されます。
3. スケジュールキューに格納された RPC 要求が Message-driven Bean（サービス）実行制御スレッドによって取り出されます。このタイミングで，Connector 1.5 仕様に準拠した WorkManager によって，Message-driven Bean（サービス）呼び出しスレッドが割り当てられます。
4. Message-driven Bean（サービス）実行制御スレッドが Message-driven Bean（サービス）呼び出しスレッドに RPC 要求を委譲します。
5. Message-driven Bean（サービス）呼び出しスレッドによって，Message-driven Bean が実行されます。

なお，TP1 インバウンドアダプタでは，OpenTP1 の機能であるスケジュールの優先順位（ユーザサービス定義の schedule_priority）を指定した RPC 要求を受信した場合も，優先順位は無視します。優先順位を指定した RPC 要求を受信した場合も，優先順序が設定されていない RPC 要求と同じように扱い，受信した順番で処理を実行します。

スケジュール機能では，これらの処理をする際に，次の項目を制御できます。

- スケジュールキューの制御
- Message-driven Bean（サービス）の同時実行数の制御
- サービス実行のタイムアウト

4.8.2　スケジュールキューの制御

スケジュールキューの制御では，スケジュールキューの長さを設定できます。また，スケジュールキューに滞留している RPC 要求の数がキュー長に近づいていることを検知するために，スケジュールキューを監視できます。

(1)　スケジュールキューの長さの設定

スケジュールキューの長さは，Message-driven Bean（サービス）ごとに設定できます。

設定は，Message-driven Bean の ActivationSpec の queue_max_length プロパティに設定します。設定を省略した場合は，スケジュールキューの長さは 100 になります。ActivationSpec の設定については，「4.10.5　ejb-jar.xml の定義」を参照してください。

スケジュールキューに滞留している RPC 要求の数がキュー長に達している状態で新たに RPC 要求が送信されると，スケジュール機能では，エラーメッセージ（KDJE58360-E）を出力します。呼び出し元の dc_rpc_call 関数には RPC エラー(DCRPCER_NO_BUFS(-304))が返されます。エラーの詳細については，「4.17　TP1 インバウンドアダプタで発生する RPC エラー応答」を参照してください。

スケジュールキューのキュー長を RPC 要求数に対して小さい値を設定していると，キューあふれによって RPC エラーが発生するおそれがあります。そのため，RPC 要求に正常に応答するためには，OpenTP1 から同時に送信される RPC 要求数と同程度のキュー長を設定してください。ただし，スケジュールキューに保持する RPC 要求はメモリ上に保持するため，多くの RPC 要求がスケジュールキューに滞留し，メモリ使用量が増加するおそれがあります。TP1 インバウンドアダプタのメモリ使用量については，「4.4　TP1 インバウンド連携機能で使用するメモリの見積もり」を参照してください。

(2)　スケジュールキューの監視

スケジュール機能では，Message-driven Bean（サービス）ごとにしきい値を設定してスケジュールキューを監視することで，滞留している RPC 要求の数がキュー長に近づいていることを事前に検知できます。しきい値を設定しておくと，スケジュールキューに滞留している RPC 要求の数がしきい値を超えたときに，警告メッセージ（KDJE58361-W）が出力されます。なお，スケジュールキューの監視は，RPC 通信機能のリクエスト受け付けスレッドによって RPC 要求がスケジュールキューに登録されたタイミングで実行されます。

また，一度しきい値を超えたことを示すメッセージが出力されてから，再度メッセージを出力するまでの間隔（警告メッセージ抑止時間）も設定できます。

これらの設定は，Message-driven Bean の ActivationSpec の次のプロパティに指定します。

- stay_watch_queue_rate プロパティ
  滞留数のしきい値を指定します。監視しない場合は 0 を指定します。
- stay_watch_check_interval プロパティ
  警告メッセージの出力抑止時間を指定します。

ActivationSpec の設定については，「4.10.5　ejb-jar.xml の定義」を参照してください。

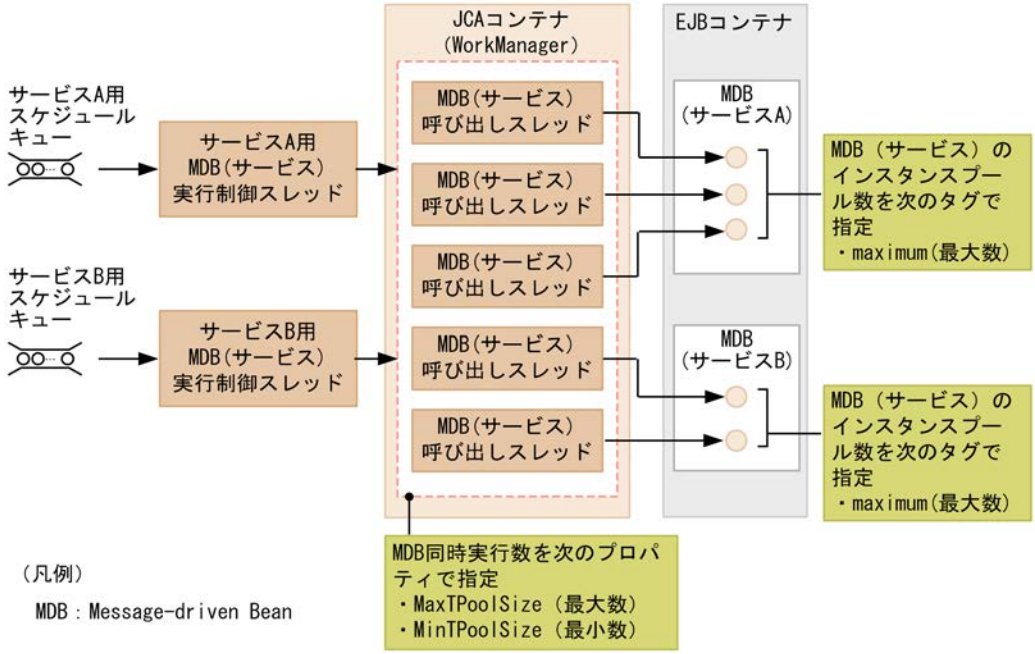
4.8.3　Message-driven Bean（サービス）の同時実行数の制御

スケジュールキューに滞留している RPC 要求は，Message-driven Bean（サービス）実行制御スレッドによって順次取り出され，Connector 1.5 仕様に準拠した WorkManager によって管理されている Message-driven Bean（サービス）呼び出しスレッドに委譲されます。RPC 要求を委譲された Message-driven Bean（サービス）呼び出しスレッドによって，Message-driven Bean（サービス）が実行されます。

スケジュール機能では，WorkManager の Message-driven Bean（サービス）呼び出しスレッド数と，Message-driven Bean（サービス）のインスタンスプール数によって，同時に実行する RPC 要求の数を制御します。

Message-driven Bean 同時実行数の制御の概要を次の図に示します。

図 4–12　Message-driven Bean 同時実行数の制御の概要



Message-driven Bean（サービス）の同時実行数を制御するプロパティについて，次の表に示します。これらの値を適切に設定しないと，正しく同時実行数を制御できません。

表 4–10　Message-driven Bean（サービス）の同時実行数を制御するプロパティ

設定項目	設定対象 (設定個所)	指定するタグ名	指定する内容
Message-driven Bean (サービス)の同時実行数	TP1 インバウンドアダプタ (Connector 属性ファイル)	<hitachi-connector-property>下の<resourceadapter-runtime><property>の<MaxTPoolSize>	Message-driven Bean（サービス）呼び出しスレッドの最大数
		<hitachi-connector-property>下の<resourceadapter-runtime><property>の<MinTPoolSize>	Message-driven Bean（サービス）呼び出しスレッドの最小数
Message-driven Bean (サービス)のインスタンスプール数	Message-driven Bean (cosminexus.xml または MessageDrivenBean 属性ファイル)	<pooled-instance><maximum>	Message-driven Bean（サービス）のインスタンスプール数の最大数
		<pooled-instance><minimum>	Message-driven Bean（サービス）のインスタンスプール数の最小数 (ただし，指定しても常に 1 とみなされます)

TP1 インバウンドアダプタ単位の最大同時実行数は，WorkManager のスレッドプール（MaxTPoolSize）に指定します。また，個々の Message-driven Bean（サービス）単位の最大同時実行数は，Message-driven Bean のインスタンスプール（maximum）で指定します。

これらの最大値は，次の関係になるように指定してください。

Message-driven Bean（サービス）呼び出しスレッドの最大数（MaxTPoolSize）≧ それぞれの Message-driven Bean（サービス）のインスタンスプール（maximum）の総和

注

指定した値がこの式の関係になっていない場合，Message-driven Bean（サービス）開始時に警告メッセージ（KDJE58452-W）が出力されます。

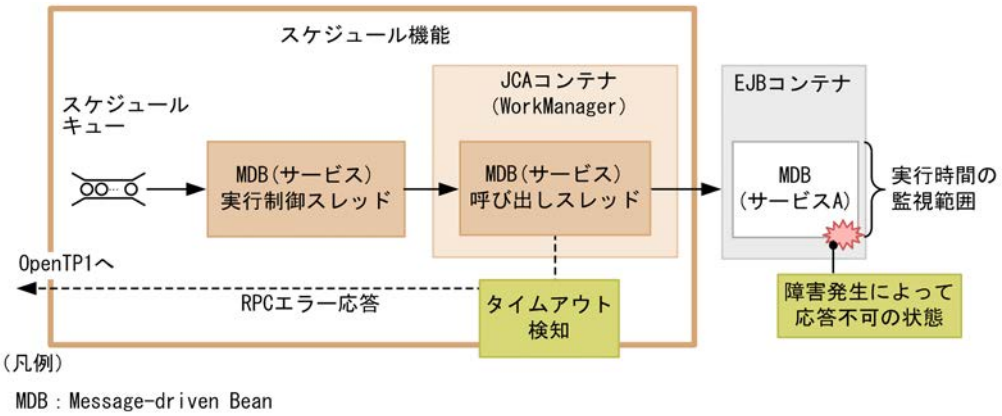
Message-driven Bean（サービス）実行制御スレッドでは，Message-driven Bean（サービス）呼び出しスレッド数が Message-driven Bean（サービス）のインスタンスプールの最大値（maximum）の総和を超えることがないよう，Message-driven Bean（サービス）呼び出しスレッドの実行数を制御します。MaxTPoolSize と maximum の関係が上記の式の関係になっていない場合，RPC 要求時に Message-driven Bean（サービス）呼び出しスレッドまたは Message-driven Bean（サービス）のインスタンスが不足することがあります。この場合，RPC 要求はスケジュールキューに滞留した状態になります。

4.8.4　サービス実行のタイムアウト

Message-driven Bean（サービス）の実行にタイムアウトを設定することで，業務処理で無限ループやデッドロックが発生した場合などに，タイムアウトによって業務を強制的に停止できます。タイムアウトによる業務処理の強制停止には，アプリケーションサーバのメソッドキャンセル機能を使用します。

TP1 インバウンドアダプタでのサービス実行のタイムアウトの概要を次の図に示します。

図 4-13　TP1 インバウンドアダプタでのサービス実行のタイムアウト



Message-driven Bean（サービス）の実行時間を監視することによって，障害が発生して応答がない場合，業務を強制停止して，Message-driven Bean（サービス）呼び出しスレッドでタイムアウトを検知できます。タイムアウトを検知した Message-driven Bean（サービス）呼び出しスレッドは，OpenTP1 に RPC エラー応答を返します。

なお，Message-driven Bean での処理の実行状態によっては，メソッドキャンセル機能で強制停止を実行できないことがあります。この場合，業務処理の終了後，TP1 インバウンドアダプタは呼び出し元の OpenTP1 に RPC 応答送信を試みます。呼び出し元が RPC 最大応答待ち時間の経過によってすでに応答を待っていなかった場合には，応答送信のコネクション確立に失敗するため，エラーメッセージ（KDJE58400-E）が出力されます。

タイムアウトの詳細については，「4.14　OpenTP1 とアプリケーションサーバ間のタイムアウトの設定」を参照してください。

4.9　トランザクション連携機能

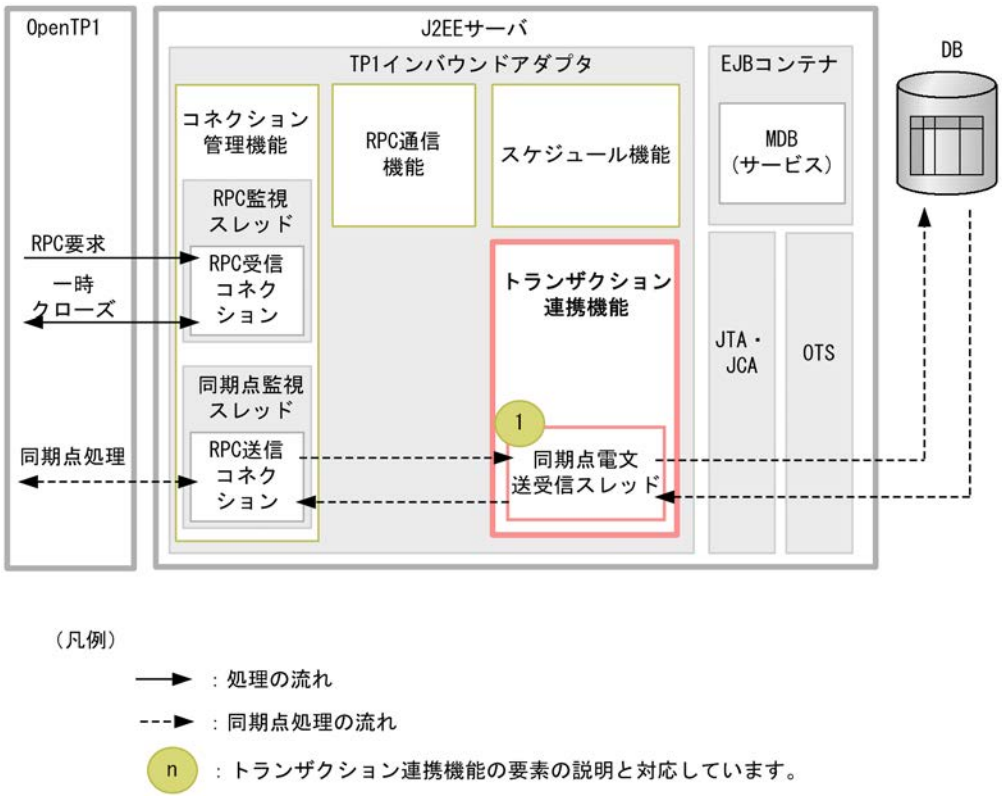
この節では，トランザクション連携機能について説明します。

4.9.1　トランザクション連携機能の概要

トランザクション連携機能は，OpenTP1 の SUP が開始したトランザクションにアプリケーションサーバの Message-driven Bean（サービス）が開始したトランザクションを参加させる機能です。これによって，OpenTP1 の SUP でのリソース更新と，アプリケーションサーバの Message-driven Bean（サービス）でのリソース更新の同期を取り，データの整合性を保てます。

TP1 インバウンドアダプタでのトランザクション連携機能の位置づけを次の図に示します。

図 4-14　トランザクション連携機能の位置づけ



同期点監視スレッドが，OpenTP1 からの同期点処理を受け付けます。RPC 送信コネクションは，トランザクション連携機能の同期点電文送受信スレッドに対して，リクエストを送信し，処理を同期します。

トランザクション通信機能は，次の要素で構成されます。

1. **同期点電文送受信スレッド**

   OpenTP1 からの同期点処理（プリペア・コミット・ロールバック）を受信して，トランザクションマネージャ（JTA・OTS）に対してトランザクションの同期点処理を委譲するスレッドです。このスレッドは，TP1 インバウンドアダプタの開始時に最大同時同期点処理数分作成し，終了時に消滅します。最大同時同期点処理数の指定方法については「4.12.2　リソースアダプタの設定」を参照してください。

トランザクション連携機能では，次の処理を制御します。

- グローバルトランザクションの範囲
- グローバルトランザクションへの Message-driven Bean（サービス）の参加条件
- トランザクショナル RPC の受信と応答
- 同期点処理
- 同期点処理の最適化
- トランザクション連携機能の設定
- トランザクション障害発生時の対処

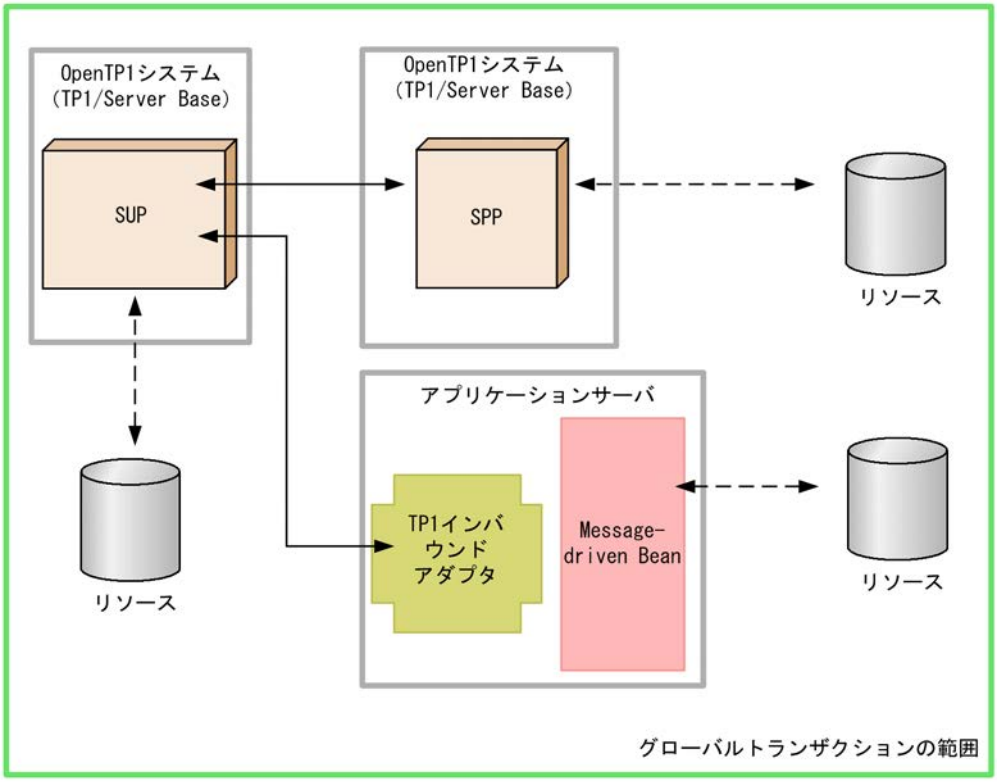
4.9.2　グローバルトランザクションの範囲

トランザクション連携機能で実現できるグローバルトランザクションの範囲について説明します。

グローバルトランザクションとは，OpenTP1 のアプリケーションプログラムで開始したトランザクションと，そのトランザクションに参加する RPC 通信先の OpenTP1 またはアプリケーションサーバのトランザクションの集合であり，データの整合性を確保する範囲となります。

トランザクション連携機能でのグローバルトランザクションの範囲を次の図に示します。

図 4-15　トランザクション連携機能でのグローバルトランザクションの範囲



（凡例）
：RPC通信の流れ　----► ：リソース更新の流れ

この図の中で示しているすべてのリソースに対して，データの整合性が確保されます。

4.9.3　グローバルトランザクションへの Message-driven Bean（サービス）の参加条件

OpenTP1 のアプリケーションプログラムが開始したトランザクションに，アプリケーションサーバの Message-driven Bean（サービス）が参加できる条件は二つあります。それぞれについて説明します。

アプリケーションサーバのライトトランザクション機能が無効な場合

ライトトランザクション機能が無効な場合，Message-driven Bean（サービス）はグローバルトランザクションに参加できます。一方，ライトトランザクション機能が有効になっている場合，アプリケーションサーバのトランザクションマネージャを使用できません。このため，ライトトランザクション機能が有効な場合に TP1 インバウンドアダプタが OpenTP1 からトランザクショナル RPC を受信したときは，エラーメッセージ（KDJE58363-E）を出力し，OpenTP1 が RPC エラー（DCRPCER_TRNCHK_EXTEND（-372））を返します。

Message-driven Bean（サービス）のトランザクション属性が Required の場合

OpenTP1 のトランザクションに参加できる Message-driven Bean（サービス）について次の表に示します。

表 4-11　OpenTP1 のトランザクションに参加できる Message-driven Bean（サービス）

Message-driven Bean（サービス）のトランザクションの設定		OpenTP1 のトランザクションへの参加
タイプ	属性	
CMT	Required	○
	NotSupported	×
BMT	－	×

（凡例）

○：OpenTP1 のトランザクションに参加できます。

×：OpenTP1 のトランザクションに参加できません（Message-driven Bean（サービス）の呼び出しはできます）。この場合，OpenTP1 ではノーアクセス最適化が適用されます。

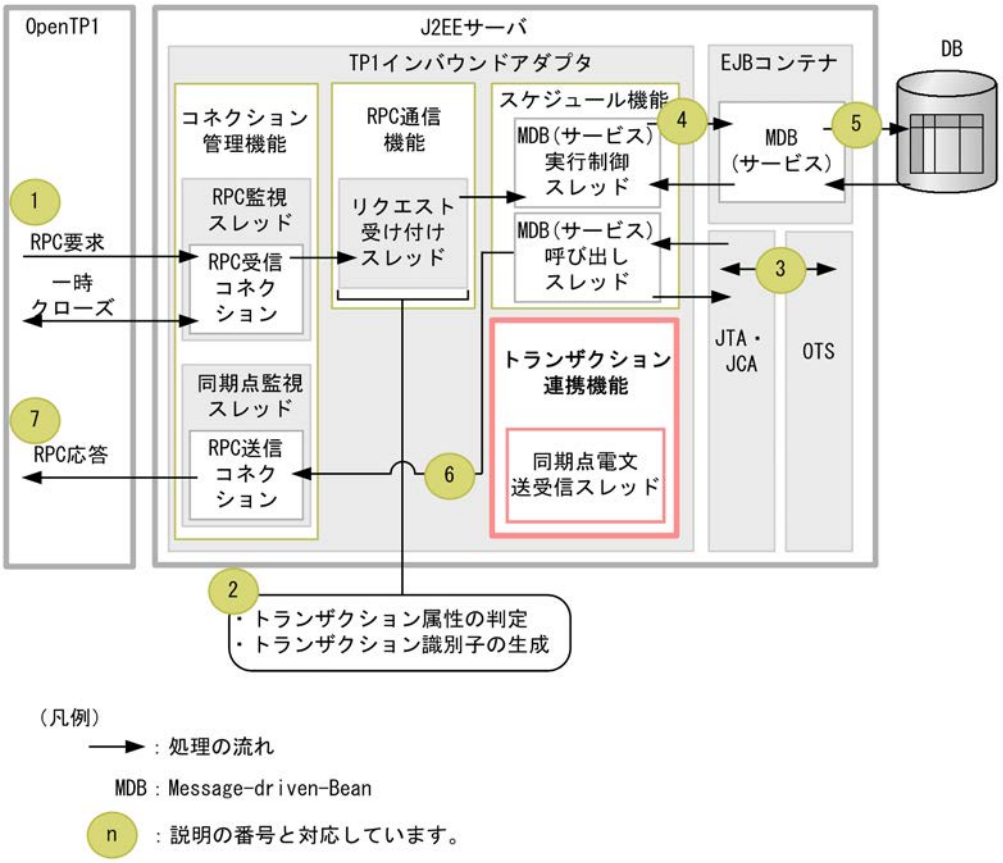
－：該当しません。

4.9.4　トランザクショナル RPC の受信と応答

OpenTP1 の SUP がトランザクション処理中に RPC 通信をすると，RPC 通信先はトランザクションブランチとして OpenTP1 のグローバルトランザクションに参加します。この場合の RPC 通信を**トランザクショナル RPC** と呼びます。

TP1 インバウンドアダプタがトランザクショナル RPC を受信して，OpenTP1 へ応答を送信するまでの流れを次の図に示します。

図 4-16　トランザクショナル RPC の受信と応答



1. OpenTP1 のアプリケーションプログラムが TP1 インバウンドアダプタに対してトランザクショナル RPC を送信します。

   TP1 インバウンドアダプタでは，コネクション管理機能が OpenTP1 からの電文を検知して，RPC 通信機能のリクエスト受け付けスレッドが電文を受信します。詳細は，「4.6.2　コネクション接続要求の受信」を参照してください。

2. RPC 通信機能のリクエスト受け付けスレッドは，受信した電文がトランザクショナル RPC の場合，OpenTP1 とトランザクション連携が可能な設定になっているかどうか（アプリケーションサーバのライトトランザクション機能が無効になっているか）を判定します。なお，受信した電文がトランザクショナル RPC でなければ判定しません。

   OpenTP1 とトランザクション連携が可能な設定については，「4.9.3　グローバルトランザクションへの Message-driven Bean（サービス）の参加条件」を参照してください。

   OpenTP1 とトランザクション連携が可能な設定と判定した場合，リクエスト受け付けスレッドは，トランザクション識別子を生成し，RPC 要求に関連づけて，その RPC 要求をスケジュール機能のスケジュールキューに登録します。スケジュール機能の詳細は，「4.8　スケジュール機能」を参照してください。

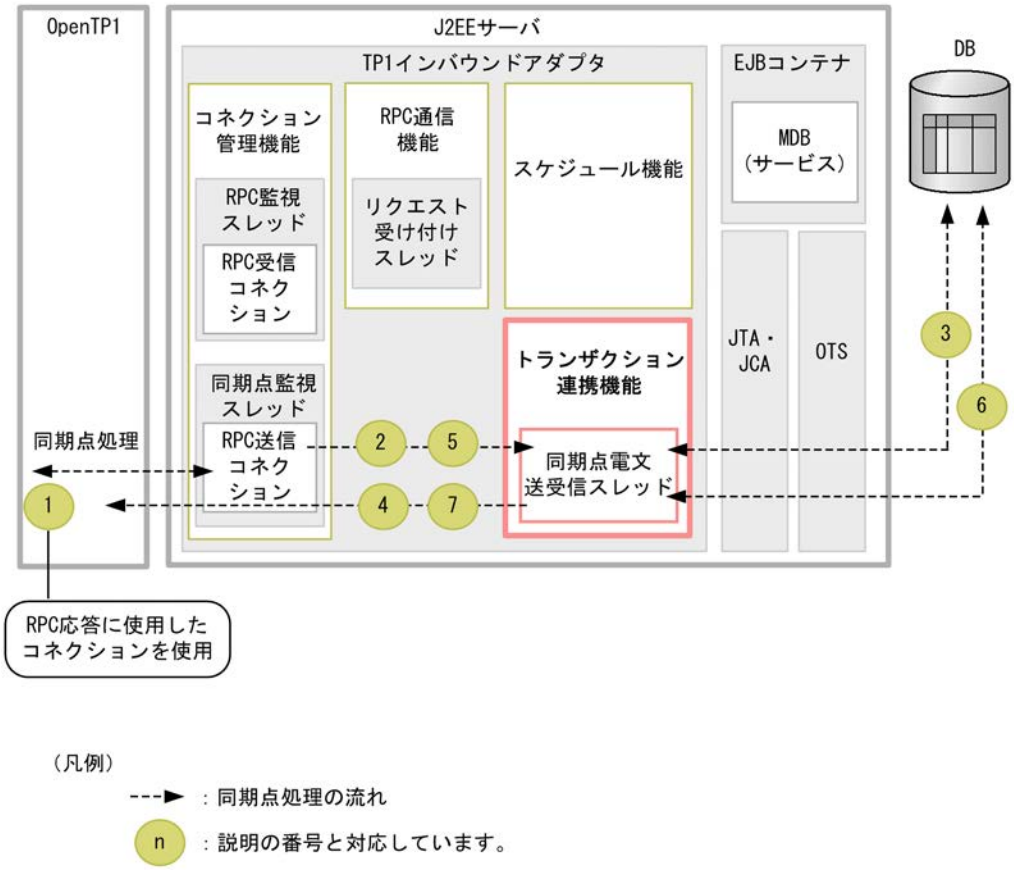
3. スケジュール機能の Message-driven Bean（サービス）実行制御スレッドは，スケジュールキューから RPC 要求を取り出し，その RPC 要求に関連づいたトランザクション識別子をトランザクションマネージャに登録します。トランザクションマネージャは，登録されたトランザクション識別子をアプリケーションサーバのトランザクション識別子とマッピングします。これによって，アプリケーションサーバのトランザクションが開始します。

4. Message-driven Bean（サービス）実行制御スレッドは，Message-driven Bean（サービス）呼び出しスレッドに RPC 要求の処理を委譲します。Message-driven Bean（サービス）呼び出しスレッドは Message-driven Bean（サービス）を実行します。
5. Message-driven Bean（サービス）が，データベースなどのリソースを更新します。
6. Message-driven Bean（サービス）呼び出しスレッドは，2.で生成したトランザクション識別子を RPC 応答に含めて OpenTP1 に送信します。この際，送信に使用するコネクションは RPC 送信コネクションになります。
7. TP1 インバウンドアダプタから RPC 応答を受信した OpenTP1 は，RPC 応答に含まれているトランザクション識別子をトランザクションブランチの識別子として管理し，その後の同期点処理のトランザクション識別子として使用します。

4.9.5　同期点処理

同期点処理では，OpenTP1 からのトランザクション処理（コミット・プリペア・ロールバック）を実行します。同期点処理は，トランザクション連携機能の同期点電文送受信スレッドが送受信します。トランザクション連携機能を使用した同期点処理の流れを次の図に示します。

図 4–17　同期点処理の流れ



1. OpenTP1 の SUP が，グローバルトランザクションをコミット（dc_trn_unchained_commit 関数）します。

   OpenTP1 は，TP1 インバウンドアダプタからの RPC 応答を受信したコネクションを使用して，TP1 インバウンドアダプタへプリペア（更新準備処理）を送信します。RPC 応答を受信したコネクションが切断している場合，OpenTP1 では，新たに TP1 インバウンドアダプタとコネクションを接続してか

らプリペアを送信します。コネクションの接続は，「4.6.2(2)　同期点待ち受けポートがコネクション接続要求を受信した際の動作」を参照してください。

2. コネクション管理機能の同期点監視スレッドでは，OpenTP1 からの電文の受信を検知すると，該当するコネクションにトランザクション連携機能の同期点電文送受信スレッドを割り当てます。

   コネクションを割り当てられた同期点電文送受信スレッドは，OpenTP1 からのプリペアを受信します。

3. 同期点電文送受信スレッドは，トランザクションマネージャに対してプリペアを実行します。トランザクションマネージャのプリペアが完了すると，同期点電文送受信スレッドは，プリペアを受信したコネクションを使用して，OpenTP1 へプリペアの応答を送信します。

4. OpenTP1 は，TP1 インバウンドアダプタからプリペアの応答を受信すると，同じコネクションを使用して TP1 インバウンドアダプタへコミットまたはロールバック（更新処理）を送信します。プリペアの応答を受信したコネクションが切断している場合，OpenTP1 では，新たに TP1 インバウンドアダプタとコネクションを接続してからコミットまたはロールバックを送信します。コネクションの接続は，「4.6.2(2)　同期点待ち受けポートがコネクション接続要求を受信した際の動作」を参照してください。

5. コネクション管理機能の同期点監視スレッドで OpenTP1 からの電文の受信を検知すると，同期点監視スレッドは該当するコネクションにトランザクション連携機能の同期点電文送受信スレッドを割り当てます。

   コネクションを割り当てられた同期点電文送受信スレッドは，OpenTP1 からのコミットまたはロールバックを受信します。

6. 同期点電文送受信スレッドは，トランザクションマネージャに対してコミットまたはロールバックを実行します。

7. トランザクションマネージャのコミット・ロールバックが完了すると，同期点電文送受信スレッドは，コミット・ロールバックを受信したコネクションを使用して OpenTP1 へコミット・ロールバックの応答を送信します。

ポイント

2 相コミットによるトランザクションの決着

OpenTP1 では，グローバルトランザクションに参加している RPC 通信先の OpenTP1，またはアプリケーションサーバのトランザクションの決着には 2 相コミットを使用しています。

2 相コミットとは，アプリケーションサーバのトランザクションの決着処理をプリペアと，コミット・ロールバックの 2 段階に分ける方式です。これにより，複数のトランザクションを矛盾なく更新できます。

2 相コミットの詳細については，マニュアル「OpenTP1 解説」の 2 相コミットに関する説明を参照してください。

4.9.6　同期点処理の最適化

OpenTP1 では，トランザクションを効率良く決着するために，同期点処理を最適化しています。
OpenTP1 の最適化の種類と，TP1 インバウンドアダプタのトランザクション連携機能での使用可否の対応を次の表に示します。

なお，それぞれの最適化処理の詳細については，マニュアル「OpenTP1 プログラム作成の手引」のトランザクションの最適化に関する説明を参照してください。

表 4-12　トランザクション連携機能を使用した際に適用できる OpenTP1 の最適化の種類

OpenTP1 の最適化の種類	概要	トランザクション連携機能での使用可否
1 相コミット最適化	アプリケーションサーバが更新するリソースが一つの場合，アプリケーションサーバの同期点処理だけで済ませます。 これによって，2 回分の同期点処理の通信が不要になります。	○
リードオンリー最適化	アプリケーションサーバがリソースを更新しない場合，またはリソースにアクセスしていない場合，OpenTP1 で同期点処理の 2 相目（コミット／ロールバック）を実行しません。 これによって，2 回分の同期点処理の通信が不要になります。 OpenTP1 との機能差 OpenTP1 では，リソースにアクセスしていない場合にはノーアクセス最適化が適用されますが，TP1 インバウンド連携機能では，リードオンリー最適化が適用されます。	△
ノーアクセス最適化	トランザクション属性が NotSupported，またはトランザクション管理種別が BMT の Message-driven Bean（サービス）をトランザクショナル RPC で呼び出した場合，OpenTP1 で同期点処理を実行しません。 これによって，4 回分の同期点処理の通信が不要になります。 OpenTP1 との機能差 OpenTP1 では，リソースにアクセスしていない場合にノーアクセス最適化が適用されます。	
コミット最適化	アプリケーションサーバで実行する同期点処理の 2 相目（コミット／ロールバック）を OpenTP1 で実行します。	×
ロールバック最適化	アプリケーションサーバでロールバック処理を実行した場合，ほかのトランザクションブランチと同期を取らないでロールバックします。	
プリペア最適化	アプリケーションサーバで実行する同期点処理の 1 相目（プリペア）を，OpenTP1 で実行します。	
非同期プリペア最適化	アプリケーションサーバの Message-driven Bean（サービス）の実行が終了した時点でプリペアを実行します。	

（凡例）

○：使用できます。

×：使用できません。

△：使用できますが機能差があります。

4.9.7　トランザクション連携機能の設定

トランザクション連携機能で指定する項目は次のとおりです。

- OpenTP1 と送受信する同期点処理の最大同時同期点処理数
  - trn_max_thread_count プロパティ

    Connector 属性ファイルの指定方法の詳細については，「4.12.2　リソースアダプタの設定」を参照してください。

- Message-driven Bean（サービス）のトランザクションタイムアウト
  詳細については「4.14　OpenTP1 とアプリケーションサーバ間のタイムアウトの設定」を参照してください。

4.9.8　トランザクション障害発生時の対処

OpenTP1 とアプリケーションサーバ間のトランザクション連携中にサーバに障害が発生した場合や，サーバを強制停止した場合，通常はサーバを再起動すれば OpenTP1 とアプリケーションサーバの自動決着機能によってトランザクションは自動的に決着します。しかし，タイミングによっては，再起動してもトランザクションが自動的に決着しないで，仕掛かり中のまま残ることがあります。

また，OpenTP1 とアプリケーションサーバ間で一時的な通信障害が発生した場合でも，通常は OpenTP1 とアプリケーションサーバの通信リトライによってトランザクションは自動的に決着します。しかし，障害が発生したタイミングや復旧までに掛かった時間によっては，トランザクションが自動的に決着しないで，仕掛かり中のまま残ることがあります。このような場合，コマンドを使用して，仕掛かり中のトランザクションを決着する必要があります。

ここでは，次に示す障害発生時での，仕掛かり中のトランザクションへの対処方法を示します。

- OpenTP1 に障害が発生，または強制停止した場合
- アプリケーションサーバに障害が発生，または強制停止した場合
- OpenTP1 とアプリケーションサーバに障害が発生した場合
- 通信障害が発生した場合
- OpenTP1 に障害が発生した場合（再開始できない場合）
- アプリケーションサーバに障害が発生した場合（再起動できない場合）
- OpenTP1 とアプリケーションサーバに障害が発生した場合（再開始または再起動できない場合）
- 仕掛かり中トランザクションの手動決着の順序

(1)　OpenTP1 に障害が発生，または強制停止した場合

トランザクション連携中に OpenTP1 に障害が発生した場合や，強制停止した場合，次に示す手順で仕掛かり中のトランザクションの有無を確認し，トランザクションを決着してください。

1. OpenTP1 の再開始
   障害発生，または強制停止した原因を取り除き，OpenTP1 を再開始します。OpenTP1 を再開始できない場合の対処は，「(5)　OpenTP1 に障害が発生した場合（再開始できない場合）」を参照してください。
2. OpenTP1 とアプリケーションサーバの仕掛かり中トランザクションの決着
   OpenTP1 の再開始後に，「(8)　仕掛かり中トランザクションの手動決着の順序」に従って OpenTP1 とアプリケーションサーバの仕掛かり中のトランザクションを決着してください。
3. アプリケーションサーバだけに残っている仕掛かり中トランザクションの決着
   手順 2.の完了後，30～45 秒経過後にアプリケーションサーバのメッセージログに KFCB40136-W メッセージが出力される場合は，アプリケーションサーバ側だけに仕掛かり中のトランザクションが残っています。この場合，KFCB40136-W メッセージの対処に従ってトランザクションを決着してください。KFCB40136-W については，マニュアル「アプリケーションサーバ メッセージ(構築／運用／開発用)」の「17.4　KFCB40000 から KFCB49999 までのメッセージ」を参照してください。

(2)　アプリケーションサーバに障害が発生，または強制停止した場合

トランザクション連携中にアプリケーションサーバに障害が発生した場合や，強制停止した場合，次に示す手順で仕掛かり中のトランザクションの有無を確認し，トランザクションを決着してください。

1. アプリケーションサーバの再起動

   障害発生，または強制停止した原因を取り除き，アプリケーションサーバを再起動します。

   アプリケーションサーバを再起動できない場合の対処は，「(6)　アプリケーションサーバに障害が発生した場合（再起動できない場合)」を参照してください。

2. OpenTP1 とアプリケーションサーバの仕掛かり中トランザクションの決着

   アプリケーションサーバの障害発生，または強制停止したあとに，OpenTP1 の標準出力に KFCA00991-W および KFCA00960-I メッセージが出力された場合は，OpenTP1 またはアプリケーションサーバに仕掛かり中のトランザクションが残っているおそれがあります。この場合は，「(8)　仕掛かり中トランザクションの手動決着の順序」に従って OpenTP1 とアプリケーションサーバの仕掛かり中のトランザクションを決着してください。

(3)　OpenTP1 とアプリケーションサーバに障害が発生した場合

トランザクション連携中に OpenTP1 とアプリケーションサーバの両方に障害が発生した場合，次に示す手順で，仕掛かり中のトランザクションの有無を確認し，トランザクションを決着してください。

1. アプリケーションサーバの再起動と OpenTP1 の再開始

   障害発生の原因を取り除き，アプリケーションサーバを再起動して，OpenTP1 を再開始してください。

   アプリケーションサーバを再起動できない場合の対処は，「(6)　アプリケーションサーバに障害が発生した場合（再起動できない場合)」を参照してください。OpenTP1 を再開始できない場合の対処は，「(5)　OpenTP1 に障害が発生した場合（再開始できない場合)」を参照してください。

   アプリケーションサーバの再起動と OpenTP1 の再開始の順序に決まりはありません。

2. OpenTP1 とアプリケーションサーバの仕掛かり中トランザクションの決着

   OpenTP1 の再開始後に，「(8)　仕掛かり中トランザクションの手動決着の順序」に従って OpenTP1 とアプリケーションサーバの仕掛かり中のトランザクションを決着してください。

3. アプリケーションサーバだけに残っている仕掛かり中トランザクションの決着

   手順 2.完了後，30～45 秒経過後にアプリケーションサーバのメッセージログに KFCB40136-W メッセージが出力される場合は，アプリケーションサーバ側だけに仕掛かり中のトランザクションが残っています。

   この場合，KFCB40136-W メッセージの対処に従ってトランザクションを決着してください。

   KFCB40136-W については，マニュアル「アプリケーションサーバ メッセージ(構築／運用／開発用)」の「17.4　KFCB40000 から KFCB49999 までのメッセージ」を参照してください。

(4)　通信障害が発生した場合

トランザクション連携中に OpenTP1 とアプリケーションサーバ間，または，アプリケーションサーバとリソース間で通信障害が発生した場合，次に示す手順で，仕掛かり中のトランザクションの有無を確認し，トランザクションを決着させてください。

1. 通信路の復旧

   通信障害の原因を取り除き，通信路を復旧します。

2. OpenTP1 とアプリケーションサーバの仕掛かり中トランザクションの決着

OpenTP1 の標準出力に KFCA00991-W および KFCA00960-I メッセージが出力されている場合は，OpenTP1 またはアプリケーションサーバに仕掛かり中のトランザクションが残っているおそれがあります。この場合は，「(8)　仕掛かり中トランザクションの手動決着の順序」に従って OpenTP1 とアプリケーションサーバの仕掛かり中のトランザクションを決着します。

3. アプリケーションサーバだけに残っている仕掛かり中トランザクションの決着

手順 2.完了後，30～45 秒経過後にアプリケーションサーバのメッセージログに KFCB40136-W メッセージが出力される場合は，アプリケーションサーバ側だけに仕掛かり中のトランザクションが残っています。

この場合，KFCB40136-W メッセージの対処に従ってトランザクションを決着します。KFCB40136-W については，マニュアル「アプリケーションサーバ メッセージ(構築／運用／開発用)」の「17.4 KFCB40000 から KFCB49999 までのメッセージ」を参照してください。

(5)　OpenTP1 に障害が発生した場合（再開始できない場合）

トランザクション連携中に OpenTP1 に障害が発生し，すぐに再開始できない場合は，次に示す手順で，仕掛かり中のトランザクションの有無を確認し，トランザクションを決着させてください。

1. トランザクションの決着状態の確認

トランザクション処理を開始して 30～45 秒経過後，アプリケーションサーバのメッセージログに KFCB40136-W メッセージが出力される場合は，アプリケーションサーバ側に仕掛かり中のトランザクションが残っています。

この場合，KFCB40136-W メッセージの内容を確認して，OpenTP1 に関連づいたリソースがコミット決着したかロールバック決着したかを確認してください。

2. トランザクションの決着

手順 1.の確認結果に応じてアプリケーションサーバ側のトランザクションを cjcommittrn コマンドまたは cjrollbacktrn コマンドで決着してください。

(6)　アプリケーションサーバに障害が発生した場合（再起動できない場合）

トランザクション連携中にアプリケーションサーバに障害が発生し，すぐに再起動できない場合は，cjlisttrnfile コマンドとアプリケーションサーバの PRF トレースを使用して，次に示す手順で仕掛かり中のトランザクションの有無を確認し，トランザクションを決着させてください。

1. アプリケーションサーバに残っている仕掛かり中トランザクションの有無の確認

アプリケーションサーバに障害が発生している場合は，cjlisttrn コマンドを使用できないため，cjlisttrnfile コマンドで仕掛かり中トランザクションの有無を確認します。

確認結果の出力例を次に示します。

```
> cjlisttrnfile
[Global transaction information(status file)]
status file1: C:¥Program Files¥HITACHI¥Cosminexus¥CC¥server¥public¥ejb¥MyServer¥
otsstatus¥transaction_1.status
host       (recorded in status file): "apsv1"
J2EE server(recorded in status file): "Myserver"

Status    GlobalTransactionId                                          BranchType
--------- ------------------------------------------------------------ ----------
Prepared  d13800010000000000000000000000fefb58e6480000000000000001 Sub
Prepared  d13800010000000000000000000000fefb58e6480000000000000002 Sub

total count: 2
```

2. アプリケーションサーバ側のトランザクションの決着状態の確認

手順 1.で出力されたトランザクションについて，アプリケーションサーバに関連するリソースがコミット決着またはロールバック決着したかを確認します。

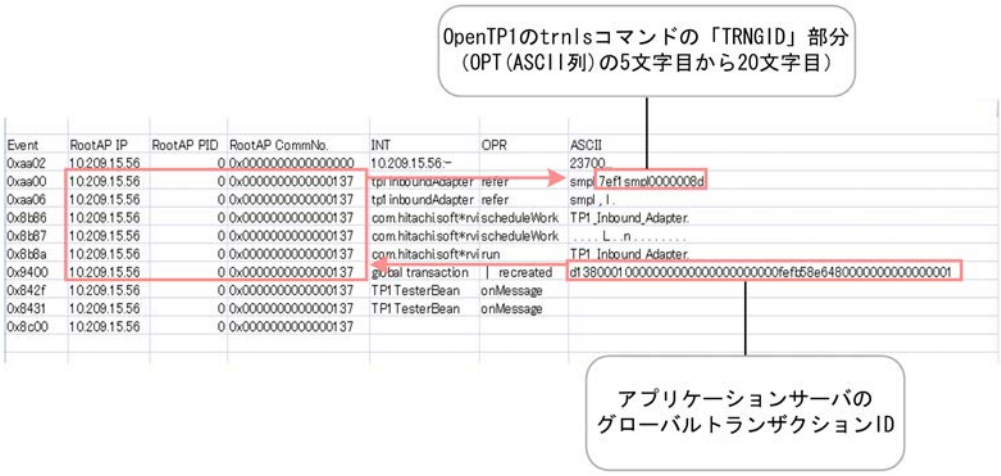
なお，トランザクションの一貫性が保たれていない場合は修復してください。

アプリケーションサーバに関連するリソースのトランザクションの決着状態の確認方法，およびトランザクションの修復方法については，リソースのマニュアルを参照してください。

3. OpenTP1 のトランザクショングローバル識別子の特定

アプリケーションサーバの PRF トレースの内容を基に，OpenTP1 のトランザクショングローバル識別子を特定します。トランザクショングローバル識別子の特定方法の例を次の図に示します。

図 4-18　OpenTP1 のトランザクショングローバル識別子の特定



0x9400 の OPT（ASCII 列）が手順 1.の出力例の「GlobalTransactionId」部分と一致する PRF トレースを特定します。0x9400 と同じルート AP 情報の 0xAA00 の OPT（ASCII 列）の 5 文字目から 20 文字目までの文字列が，OpenTP1 のトランザクショングローバル識別子となります。

4. OpenTP1 側のトランザクションの決着状態の確認

手順 3.で特定した OpenTP1 のトランザクショングローバル識別子について，OpenTP1 に関連するリソースがコミット決着またはロールバック決着したかを確認します。トランザクションの一貫性が保たれていない場合は，修復してください。

(7)　OpenTP1 とアプリケーションサーバに障害が発生した場合（再開始または再起動できない場合）

トランザクション連携中に OpenTP1 とアプリケーションサーバに障害が発生し，すぐに再開始または再起動できない場合は，「(6)　アプリケーションサーバに障害が発生した場合（再起動できない場合）」の手順に従って仕掛かり中のトランザクションの有無を確認し，仕掛かり中のトランザクションがあれば決着してください。

(8)　仕掛かり中トランザクションの手動決着の順序

OpenTP1 とアプリケーションサーバ間のトランザクション連携中に障害が発生すると，OpenTP1 またはアプリケーションサーバに仕掛かり中のトランザクションが残っているおそれがあります。この場合，次に示す手順で仕掛かり中のトランザクションの有無を確認し，トランザクションを決着させてください。仕掛かり中のトランザクションが複数ある場合は，次に示す手順を繰り返してください。

1. OpenTP1 の仕掛かり中トランザクションの状態確認

OpenTP1 の trnls コマンドを実行し，OpenTP1 のトランザクションの状態を確認します。このとき，-t オプションを指定します。

```
> trnls -t
TRNGID                TRNBID             状態                          PID    サーバ    サービス
cab9TP1S0000074f      TP1STP1S0000074f   HEURISTIC_FORGETTING(p,n)  0   TP1S
```

2. OpenTP1 の仕掛かり中トランザクションの強制終了

手順 1.の実行結果の「状態」部分が"HEURISTIC_FORGETTING"以外の場合は，手順 3.へ進みます。「状態」部分が"HEURISTIC_FORGETTING"の場合は，OpenTP1 の trnfgt コマンドを実行し，OpenTP1 のトランザクションを強制終了します。このとき，-T オプションに手順 1.の実行結果の「TRNGID」部分を指定します。

```
> trnfgt -T cab9TP1S0000074f
```

トランザクションの実行状態によっては，trnfgt コマンドは正常終了しても OpenTP1 のトランザクションを強制終了できていなかったり，trnfgt コマンドが KFCA00970-E を出力して異常終了していたりすることがあります。この場合は手順 3.に進み，アプリケーションサーバの仕掛かり中トランザクションをすべて決着してから，手順 5.を実施してください。

3. アプリケーションサーバの仕掛かり中トランザクションの特定

アプリケーションサーバの cjlisttrn コマンドを実行し，手順 1.の実行結果の「TRNGID」部分と，cjlisttrn コマンドの実行結果の「TP1TRNGID」部分が一致するトランザクションを特定します。このとき，-tp1 オプションと-pending オプションを指定します。

なお，この例では，決着するアプリケーションサーバの仕掛かり中トランザクションの「Status」部分は"Prepared"となります。

```
> cjlisttrn MyServer -tp1 -pending
[Global transaction information]
Status            GlobalTransactionId                                    ElapsedTime BranchType
TP1TRNGID        TP1TRNBID
----------------- ------------------------------------------------------ ----------- ----------
---------------- ----------------
Prepared          d13800010000000000000000000000fefb58e6480000000000000001          30 Sub
cab9TP1S0000074f @@@@TP1S0000074f

total count: 1
```

4. アプリケーションサーバの仕掛かり中トランザクションの決着

- 手順 1.の実行結果の「状態」部分が"HEURISTIC_FORGETTING"，または"COMMIT"だった場合

  アプリケーションサーバの cjcommittrn コマンドを実行し，アプリケーションサーバの仕掛かり中トランザクションをコミットで決着します。このとき，-gid オプションに手順 3.で特定したアプリケーションサーバの仕掛かり中トランザクションの「GlobalTransactionId」部分を指定します。

```
> cjcommittrn MyServer -gid d13800010000000000000000000000fefb58e6480000000000000001
```

- 「状態」部分が"ROLLBACK"だった場合

  アプリケーションサーバの cjrollbacktrn コマンドを実行し，アプリケーションサーバの仕掛かり中トランザクションをロールバックで決着します。このとき，-gid オプションに手順 3.で特定したアプリケーションサーバの仕掛かり中トランザクションの「GlobalTransactionId」部分を指定します。

```
> cjrollbacktrn MyServer -gid d13800010000000000000000000000fefb58e6480000000000000001
```

5. コマンドで決着できない OpenTP1 の仕掛かり中トランザクションの強制終了

手順 1.の実行結果の「状態」部分が"COMMIT"，または，手順 2.の trnfgt コマンドを実行しても強制終了できなかった"HEURISTIC_FORGETTING"のトランザクションは，コマンドを使った手動決着ができません。

そのため，すべての仕掛かり中のトランザクションに対して手順 1.～4.を実施したあと，再度 OpenTP1 の trnls コマンドを実行します。コマンド実行後，「状態」部分に"COMMIT"または

"HEURISTIC_FORGETTING"のトランザクションが残っている場合は，OpenTP1 を強制停止してから強制正常開始してください。

4.10　Message-driven Bean（サービス）の実装

TP1 インバウンドアダプタを利用した Message-driven Bean（サービス）の実装手順を次に示します。

1. Message-driven Bean（サービス）の実装の準備
   TP1 インバウンドアダプタを開始します。
2. Message-driven Bean（サービス）の作成
   TP1 インバウンドアダプタから呼び出す Message-driven Bean（サービス）を作成します。
   次の作業を実施してください。
   - クラスパスの設定
   - リスナインタフェースの実装
   - onMessage メソッドでの業務ロジックの実装
   - ejb-jar.xml の定義
   - cosminexus.xml の定義
3. Message-driven Bean（サービス）の開始
   Message-driven Bean（サービス）を含む J2EE アプリケーションを開始します。

参考

アプリケーションサーバでは，TP1 インバウンドアダプタから呼び出す Message-driven Bean（サービス）のサンプルプログラムを提供しています。サンプルプログラムは，次に示すディレクトリに格納されています。

- Windows の場合
  <Application Serverのインストールディレクトリ>¥CC¥examples¥tp1inbound
- UNIX の場合
  /opt/Cosminexus/CC/examples/tp1inbound

4.10.1　Message-driven Bean（サービス）の実装の準備

Message-driven Bean（サービス）を実装する前に J2EE サーバの設定，および TP1 インバウンドアダプタの開始を実施します。

(1)　トランザクション連携機能を使用する場合の J2EE サーバの設定

トランザクション連携機能を使用する場合に J2EE サーバの設定が必要です。

トランザクション連携機能では，グローバルトランザクションを利用します。そのため，トランザクション連携機能を使用する場合は，J2EE サーバのライトトランザクション機能を無効にする必要があります。

J2EE サーバのライトトランザクション機能を無効にするプロパティについて次に示します。なお，トランザクション連携機能を使用しない場合は，次のプロパティの設定は，有効または無効のどちらでもかまいません。

ejbserver.distributedtx.XATransaction.enabled
: ライトトランザクションを使用するか，グローバルトランザクションを使用するかを指定します。true を指定した場合，グローバルトランザクションを利用できます。

(2)　TP1 インバウンドアダプタの開始

TP1 インバウンドアダプタを開始する手順を示します。

1. TP1 インバウンドアダプタをインポートします。

   cjimportres コマンドを実行して，TP1 インバウンドアダプタをインポートします。アプリケーションサーバでは，TP1 インバウンドアダプタ（TP1InboundAdapter.rar）を Connector1.5 仕様に準拠したインバウンドリソースアダプタとして提供しています。

   実行形式を次に示します。

```
cjimportres <サーバ名> -type rar -f <ファイルパス>
```

   <ファイルパス>には，TP1 インバウンドアダプタのパスとして，次のパスを指定してください。

   - Windows の場合

     ”<Application Serverのインストールディレクトリ>¥CC¥adapters¥OpenTP1¥TP1InboundAdapter.rar”

   - UNIX の場合

     /opt/Cosminexus/CC/adapters/OpenTP1/TP1InboundAdapter.rar

2. TP1 インバウンドアダプタをデプロイします。

   cjdeployrar コマンドを実行して，インポートした TP1 インバウンドアダプタをデプロイします。

   実行形式を次に示します。

```
cjdeployrar <サーバ名> -resname TP1_Inbound_Adapter
```

3. TP1 インバウンドアダプタを定義します。

   TP1 インバウンドアダプタの Connector 属性ファイルを編集して，TP1 インバウンドアダプタの Connector 属性を定義します。

   TP1 インバウンドアダプタの Connector 属性の定義の詳細については,「4.12.2　リソースアダプタの設定」を参照してください。

4. TP1 インバウンドアダプタを開始します。

   cjstartrar コマンドを実行して，TP1 インバウンドアダプタを開始します。

   実行形式を次に示します。

```
cjstartrar <サーバ名> -resname TP1_Inbound_Adapter
```

4.10.2　クラスパスの設定

TP1 インバウンドアダプタのリスナインタフェースクラスは ejbserver.jar に含まれています。

Message-driven Bean（サービス）を作成する環境で ejbserver.jar のクラスパスを設定してください。

- Windows の場合

```
set CLASSPATH=%CLASSPATH%;%COSMINEXUS_HOME%¥CC¥lib¥ejbserver.jar
```

- UNIX（sh）の場合

```
CLASSPATH=${CLASSPATH}:/opt/Cosminexus/CC/lib/ejbserver.jar
export CLASSPATH
```

- UNIX（csh）の場合

```
setenv CLASSPATH ${CLASSPATH}:/opt/Cosminexus/CC/lib/ejbserver.jar
```

4.10.3　リスナインタフェースの実装

Message-driven Bean（サービス）に，TP1 インバウンドアダプタのリスナインタフェースを実装してください。

TP1 インバウンドアダプタのリスナインタフェースを次に示します。

- TP1MessageListener

Message-driven Bean（サービス）にリスナインタフェース（TP1MessageListener）を実装する例を次に示します。

```
public class SampleMDB
implements TP1MessageListener,MessageDrivenBean
{
  . . .
}
```

4.10.4　onMessage メソッドでの業務ロジックの実装

onMessage メソッドに，次に示す業務ロジックを実装します。

```
public TP1OutMessage onMessage(TP1InMessage in)
{
  //入力パラメタ取得
  byte[] inputdata = in.getInputData();
   . . .
  //出力データオブジェクトの生成
  TP1OutMessage out = in.createOutMessage();
   . . .
   . . .
  //出力データ格納領域の長さを取得
  int outLen = out.getMaxOutputLength();
   . . .
  try {
     . . .
    //出力データ格納領域の取得
    byte[] outputdata = out.getOutputData(outLen);
     . . .
    //業務処理
    //outputdata中に出力データを格納
     . . .
  } catch(Exception e) {
    //errorLenにエラー処理用の出力データ長を設定
    //エラー処理用の出力データ格納領域の取得
    byte[] outputdata = out.getOutputData(errorLen);
     . . .
    //outputdata中にエラー情報を格納
     . . .
  }
  return out;
}
```

- TP1InMessage からユーザデータを取得して業務処理を実行します。
- 実行結果の出力データを TP1OutMessage の getOutputData メソッドで取得した byte 配列中に格納して返します。
- getOutputData メソッドの引数 outLen には，応答の長さを指定します。応答の長さは，0～（TP1OutMessage インタフェースの getMaxOutputLength メソッドの戻り値）以下の範囲で指定してください。
- onMessage メソッドの戻り値が null の場合，または getOutputData メソッドを呼び出していない場合は，出力データが未設定（null）のため，TP1 インバウンドアダプタはエラーメッセージを出力して，RPC エラー(DCRPCER_SYSERR_AT_SERVER(-316))を返します。この場合，呼び出し元の dc_rpc_call 関数は RPC エラー(トランザクショナル RPC でない場合は

DCRPCER_SYSERR_AT_SERVER(-316)，トランザクショナル RPC の場合は DCRPCER_SYSERR_AT_SERVER_RB(-325))を返します。

- 業務処理では例外をすべてキャッチして，エラー時もエラー情報を TP1OutMessage インタフェースの getOutputData メソッドで取得した byte 配列にエラー情報を格納して返してください。業務処理で発生した例外をキャッチしなかった場合，または null を返した場合は，エラーメッセージ（KDJE58459-E）が出力されて，RPC エラー(DCRPCER_SYSERR_AT_SERVER(-316))になります。この場合，呼び出し元の dc_rpc_call 関数は RPC エラー(トランザクショナル RPC でない場合は DCRPCER_SYSERR_AT_SERVER(-316)，トランザクショナル RPC の場合は DCRPCER_SYSERR_AT_SERVER_RB(-325))を返します。RPC エラー応答の詳細については，「4.17　TP1 インバウンドアダプタで発生する RPC エラー応答」を参照してください。

Message-driven Bean（サービス）のコーディング例は，トランザクション連携機能を使用しない場合と，トランザクション連携機能を使用する場合の 2 種類があります。Message-driven Bean（サービス）のコーディング例を次に示します。

● トランザクション連携機能を使用しない場合のコーディング例

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import javax.ejb.EJBException;
import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.naming.InitialContext;
import javax.sql.DataSource;
import javax.transaction.UserTransaction;

import com.hitachi.software.ejb.adapter.tp1.TP1InMessage;
import com.hitachi.software.ejb.adapter.tp1.TP1MessageListener;
import com.hitachi.software.ejb.adapter.tp1.TP1OutMessage;

public class SampleMDB
implements TP1MessageListener, MessageDrivenBean {

    private MessageDrivenContext context;
    public TP1OutMessage onMessage(TP1InMessage in) {
        //入力パラメタ取得
        byte[] inputdata = in.getInputData();

        //出力データオブジェクトの生成
        TP1OutMessage out = in.createOutMessage();

        //出力データ格納領域の長さを取得
        int outLen = out.getMaxOutputLength();

        UserTransaction ut = null;
        Connection con = null;
        PreparedStatement prepStmt = null;
        try {
            //DB更新する業務処理を実装
            InitialContext ic = new InitialContext();
            DataSource ds = (DataSource)ic.lookup(
                                ”java:comp/env/jdbc/SampleDB”);
            ut = context.getUserTransaction();
            ut.begin();
            con = ds.getConnection();
            prepStmt = con.prepareStatement(
                            ”update sample set name = ? where id = 1”);
            prepStmt.setString(1, new String(inputdata));
            prepStmt.executeUpdate();
            ut.commit();
            byte[] result =”update complete.”.getBytes();
            //出力データ格納領域の取得
            byte[] outputdata = out.getOutputData(outLen);

            //業務処理の結果を出力データにコピー
            System.arraycopy(result,0,outputdata,0,result.length);
        } catch(Exception e) {
            //エラー処理
            try {
                if(ut != null) {
                    ut.rollback();
                }
```

```
            } catch(Exception ex) {
            }
            byte[] wk = "An error occurred.".getBytes();

            //エラー処理用の出力データ格納領域の取得
            byte[] outputdata = out.getOutputData(wk.length);
            //エラー処理の内容を出力データにコピー
            System.arraycopy(wk,0,outputdata,0,wk.length);
        }
        finally {
            if(prepStmt != null) {
                try {
                    prepStmt.close();
                } catch(Exception exp) {
                }
            }
            if(con != null) {
                try {
                    con.close();
                } catch(Exception exp) {
                }
            }
        }
        return out;
    }

    public void ejbCreate() {
    }
    public void ejbRemove() throws EJBException {
    }
    public void setMessageDrivenContext(MessageDrivenContext ctx)
            throws EJBException {
        this.context = ctx;
    }
}
```

● トランザクション連携機能を使用する場合のコーディング例

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import javax.ejb.EJBException;
import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.naming.InitialContext;
import javax.sql.DataSource;

import com.hitachi.software.ejb.adapter.tp1.TP1InMessage;
import com.hitachi.software.ejb.adapter.tp1.TP1MessageListener;
import com.hitachi.software.ejb.adapter.tp1.TP1OutMessage;
public class SampleMDB
implements TP1MessageListener, MessageDrivenBean {
    public TP1OutMessage onMessage(TP1InMessage in) {
        //入力パラメタ取得
        byte[] inputdata = in.getInputData();

        //出力データオブジェクトの生成
        TP1OutMessage out = in.createOutMessage();

        //出力データ格納領域の長さを取得
        int outLen = out.getMaxOutputLength();
        Connection con = null;
        PreparedStatement prepStmt = null;
        try {
            //DB更新する業務処理を実装
            InitialContext ic = new InitialContext();
            DataSource ds = (DataSource)ic.lookup(
                                "java:comp/env/jdbc/SampleDB");
            con = ds.getConnection();
            prepStmt = con.prepareStatement(
                            "update sample set name = ? where id = 1");
            prepStmt.setString(1, new String(inputdata));
            prepStmt.executeUpdate();
            byte[] result ="update complete.".getBytes();

            //出力データ格納領域の取得
            byte[] outputdata = out.getOutputData(outLen);
            //業務処理の結果を出力データにコピー
            System.arraycopy(result,0,outputdata,0,result.length);
        } catch(Exception e) {
            //エラー処理
```

```
            byte[] wk = "An error occurred.".getBytes();

            //エラー処理用の出力データ格納領域の取得
            byte[] outputdata = out.getOutputData(wk.length);

            //エラー処理の内容を出力データにコピー
            System.arraycopy(wk,0,outputdata,0,wk.length);
        }
        finally {
            if(prepStmt != null) {
                try {
                    prepStmt.close();
                } catch(Exception exp) {
                }
            }
            if(con != null) {
                try {
                    con.close();
                } catch(Exception exp) {
                }
            }
        }
        return out;
    }

    public void ejbCreate() {
    }
    public void ejbRemove() throws EJBException {
    }
    public void setMessageDrivenContext(MessageDrivenContext ctx)
            throws EJBException {
    }
}
```

4.10.5　ejb-jar.xml の定義

Message-driven Bean の ejb-jar.xml を定義します。

定義項目を次に示します。

- Message-driven Bean（サービス）が実装するリスナインタフェース

  <messaging-type>タグに次の値を設定します。

  `com.hitachi.software.ejb.adapter.tp1.TP1MessageListener`

- トランザクションの管理方法

  <transaction-type>タグに次のどちらかを設定します。

  - `Bean`
  - `Container`

- トランザクションの属性

  <trans-attribute>タグに「Container」を指定した場合，Message-driven Bean のメソッドの属性に次のどちらかを設定します。

  - `Required`
  - `NotSupported`

- ActivationSpec の設定

  ActivationSpec に設定する値を<activation-config>タグの下の<activation-config-property>タグ内で定義します。プロパティ名とプロパティ値を指定してください。

  <activation-config-property>タグ内で指定する値を次に示します。

activation-config-property-name※	データ型	activation-config-property-value
service	java.lang.String	Message-driven Bean（サービス）のサービス名を 1～31 文字で指定します。 1 文字目は英字（A～Z　a～z），2 文字目以降は英数字（A～Z　a～z　0～9）またはアンダースコア（_）で指定してください。 一つのサービスグループ内で一意のサービス名を指定してください。
queue_max_length	java.lang.Integer	サービスごとのスケジュールキューの長さを 1～65535 の整数で指定します。デフォルト値は 100 です。
stay_watch_queue_rate		スケジュールキューの滞留監視でのしきい値として，メッセージを出力するキューの滞留数の割合を，0～100 の整数で指定します。デフォルト値は 70 です。 指定値が 0 の場合は滞留監視をしません。
stay_watch_check_interval		スケジュールキューの滞留監視での警告メッセージの抑止時間を 1～2147483647「秒」の整数で指定します。デフォルト値は 60 秒です。

注※　ActivationSpec の設定では，リソースアダプタの DD（ra.xml）の<activationspec>-<required-config-property>タグに指定されているプロパティを設定する必要があります。

ActivationSpec の設定の詳細については，「3.16.8　Connector 1.5 仕様に準拠したリソースアダプタを使用する場合の設定」を参照してください。

4.10.6　cosminexus.xml の定義

cosminexus.xml に，Message-driven Bean（サービス）に関連する項目を定義します。

次に定義項目を示します。

- 使用する TP1 インバウンドアダプタの表示名
  <resource-adapter>タグに Connector 属性ファイルの<display-name>タグの値と同じ値を指定します。
- プール内のインスタンスの設定
  <pooled-instance>タグの下で，プール内のインスタンスを定義します。
  - minimum（Message-driven Bean（サービス）のインスタンスプールの最小数）
    この値は変更できません。常に「1」です。
  - maximum（Message-driven Bean（サービス）のインスタンスプールの最大数）

定義項目の詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「2.2　アプリケーション属性ファイル（cosminexus.xml）で指定する各属性の詳細」を参照してください。

4.10.7　Message-driven Bean（サービス）の開始

次に，Message-driven Bean（サービス）を開始する手順を示します。

1. Message-driven Bean（サービス）をインポートします。

cjimportapp のコマンドを実行して，Message-driven Bean（サービス）を含む J2EE アプリケーションをインポートします。

実行形式を次に示します。

```
cjimportapp 〈サーバ名〉 -f ”〈J2EEアプリケーションのファイルパス〉”
```

2. J2EE アプリケーションを開始します。

cjstartapp コマンドを実行して，インポートした J2EE アプリケーションを開始します。

実行形式を次に示します。

```
cjstartapp 〈サーバ名〉 -name 〈J2EEアプリケーション名〉
```

4.11　OpenTP1 のアプリケーションプログラムの作成

ここでは，C 言語でプログラムを作成する場合を例にして，TP1 インバウンドアダプタを呼び出すアプリケーションプログラムの作成について説明します。

OpenTP1 のアプリケーションプログラムからの TP1 インバウンドアダプタの呼び出しは，dc_rpc_call 関数を使用します。dc_rpc_call 関数の各引数に指定する値を次の表に示します。

表 4-13　dc_rpc_call 関数の引数に指定する値

引数	指定する値
group	TP1 インバウンドアダプタの Connector 属性ファイルの <service_group>タグで定義したサービスグループ名を指定します。
service	Message-driven Bean（サービス）の ActivationSpec の<service>タグで定義したサービス名を指定します。
in	サービスの入力パラメタを指定します。詳細は，マニュアル「OpenTP1 プログラム作成の手引」を参照してください。
in_len	サービスの入力パラメタ長を指定します。詳細は，マニュアル「OpenTP1 プログラム作成の手引」を参照してください。
out	応答を格納する領域を指定します。詳細は，マニュアル「OpenTP1 プログラム作成の手引」を参照してください。
out_len	応答の長さを指定します。詳細は，マニュアル「OpenTP1 プログラム作成の手引」を参照してください。
flags	RPC の形態を指定します。指定可能な値については「4.13.3　トランザクションの設定」を参照してください。

コーディング例は，トランザクション連携機能を使用しない場合と，トランザクション連携機能を使用する場合の 2 種類があります。

TP1 インバウンドアダプタを呼び出すアプリケーションの SUP のコーディング例を次に示します。なお，この例は SUP 固有のもので，SPP，MHP には該当しません。

● トランザクション連携機能を使用しない場合のコーディング例

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <dcrpc.h>
#include <dctrn.h>
#include <dcadm.h>

main ()
{

  static char    in_buf[1024];
  static DCLONG  in_buf_len;
  static char    out_buf[1024];
  static DCLONG  out_buf_len;
  int rc;

  rc = dc_rpc_open(DCNOFLAGS);
  if (rc != DC_OK) {
    printf("SUP01:dc_rpc_openに失敗しました。CODE = %d¥n", rc);
    goto PROG_END;
  }

  rc = dc_adm_complete(DCNOFLAGS);
```

```
  if (rc != DC_OK) {
    printf("SUP01:dc_adm_completeに失敗しました。CODE = %d¥n", rc);
    goto PROG_END;
  }

  strcpy(in_buf, "SUP01:DATA OpenTP1!!");
  in_buf_len = strlen(in_buf) + 1;
  out_buf_len = 1024;
  rc = dc_rpc_call("spp01grp", "svr01", in_buf, &in_buf_len, out_buf,
      &out_buf_len, DCNOFLAGS | DCRPC_TPNOTRAN);
  if (rc != DC_OK) {
    printf("SUP01:サービス要求に失敗しました。CODE = %d¥n", rc);
  }
  printf("SUP01:SERVICE FUNCTION RETURN = %s¥n", out_buf);

  PROG_END:
  dc_rpc_close(DCNOFLAGS);
  printf("SUP01:処理を終了しました。¥n");
  exit(0);
}
```

● トランザクション連携機能を使用する場合のコーディング例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dcadm.h>
#include <dcrpc.h>
#include <dctrn.h>

main()
{

  static char     in_buf[1024];
  static DCULONG  in_buf_len;
  static char     out_buf[1024];
  static DCULONG  out_buf_len;
  int rc;

  rc = dc_rpc_open(DCNOFLAGS);
  if (rc != DC_OK) {
    printf("SUP01:dc_rpc_openに失敗しました。CODE = %d¥n", rc);
    goto PROG_END;
  }

  rc = dc_adm_complete(DCNOFLAGS);
  if (rc != DC_OK) {
    printf("SUP01:dc_adm_completeに失敗しました。CODE = %d¥n", rc);
    goto PROG_END;
  }

  rc = dc_trn_begin();
  if (rc != DC_OK) {
    printf("SUP01:dc_trn_beginに失敗しました。CODE = %d¥n", rc);
    goto TRAN_END;
  }

  strcpy(in_buf, "SUP01:DATA OpenTP1!!");
  in_buf_len = strlen(in_buf) + 1;
  out_buf_len = 1024;
  rc = dc_rpc_call("spp01grp", "svr01", in_buf, &in_buf_len, out_buf,
                   &out_buf_len, DCNOFLAGS);
  if (rc != DC_OK) {
    printf("SUP01:サービス要求に失敗しました。CODE = %d¥n", rc);
    goto TRAN_END;
  }
  printf("SUP01:SERVICE FUNCTION RETURN = %s¥n", out_buf);

  TRAN_END:
  rc = dc_trn_unchained_commit();
  if (rc != DC_OK) {
    printf("SUP01:dc_trn_unchained_commitに失敗しました。CODE = %d¥n", rc);
  }

  PROG_END:
  dc_rpc_close(DCNOFLAGS);
  printf("SUP01:処理を終了しました。¥n");
  exit(0);
}
```

4.12　アプリケーションサーバの実行環境での設定

TP1 インバウンド連携機能を使用する場合，J2EE サーバの設定とリソースアダプタの設定が必要です。

4.12.1　J2EE サーバの設定

J2EE サーバの設定は，簡易構築定義ファイルで実施します。TP1 インバウンド連携機能の定義は，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に指定します。簡易構築定義ファイルでの定義を次の表に示します。

表 4-14　TP1 インバウンド連携機能を使用するための簡易構築定義ファイルの定義

項目	指定するパラメタ	設定内容
通信に使用する IP アドレス	ejbserver.jca.adapter.tp1.bind_host	TP1 インバウンド連携機能の通信（受信および送信）に使用する IP アドレスを指定します。ホスト名も指定できます。 指定がない場合，またはホスト名のアドレスが解決できない場合，システムによって自動的に選択された有効なローカルアドレスが使用されます。

簡易構築定義ファイルおよびパラメタについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.6　簡易構築定義ファイル」を参照してください。

4.12.2　リソースアダプタの設定

TP1 インバウンド連携機能の定義には，TP1 インバウンドアダプタの Connector 属性ファイルを使用します。

次に TP1 インバウンドアダプタの Connector 属性ファイルの設定について説明します。

(1)　TP1 インバウンドアダプタの RAR ファイル

TP1 インバウンドアダプタの RAR ファイルは次のディレクトリに格納されています。

- Windows の場合

```
<Application Serverのインストールディレクトリ>¥CC¥adapters¥OpenTP1¥TP1InboundAdapter.rar
```

- UNIX の場合

```
/opt/Cosminexus/CC/adapters/OpenTP1/TP1InboundAdapter.rar
```

(2)　Connector 属性ファイルの取得と設定

Connector 属性ファイルを取得して，値を編集してから TP1 インバウンドアダプタに設定します。ここでは，取得と設定の手順について説明します。

参考

TP1 インバウンドアダプタでは，アプリケーションサーバが提供する Connector 属性ファイルのテンプレートファイルを使用できます。テンプレートファイルを使用すると，TP1 インバウンドアダプタをインポートする前に，Connector 属性ファイルを編集しておくことができます。テンプレートファイルを使用する場合は，編集対象の Connector 属性ファイルをサーバ管理コマンド（cjgetrarprop コマンドまたは cjgetresprop コマンド）で取得する操作が不要です。

Connector 属性ファイルのテンプレートファイルの格納先を次に示します。

- Windows の場合

```
<Application Serverのインストールディレクトリ>¥CC¥admin¥templates¥TP1InboundAdapter_cfg.xml
```

- UNIX の場合

```
/opt/Cosminexus/CC/admin/templates/TP1InboundAdapter_cfg.xml
```

テンプレートファイル使用時の注意事項については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1.14　Connector 属性ファイルのテンプレートファイル」を参照してください。

● Connector 属性ファイルの取得

cjgetrarprop コマンドを実行して，TP1 インバウンドアダプタの Connector 属性を取得します。

実行形式を次に示します。

```
cjgetrarprop <サーバ名> -resname TP1_Inbound_Adapter -c <rar属性ファイルのファイルパス>
```

● Connector 属性ファイルの設定

cjsetrarprop コマンドを実行して，編集後の TP1 インバウンドアダプタの Connector 属性ファイルを設定します。TP1 インバウンドアダプタの Connector 属性の編集内容については，「(3)　Connector 属性の定義」を参照してください。

実行形式を次に示します。

```
cjsetrarprop <サーバ名> -resname TP1_Inbound_Adapter -c <rar属性ファイルのファイルパス>
```

(3)　Connector 属性の定義

TP1 インバウンドアダプタの Connector 属性ファイルを定義します。

● TP1 インバウンドアダプタのプロパティの設定

次の表に，TP1 インバウンドアダプタのプロパティの設定内容を示します。

表 4-15　Connector 属性ファイルでの TP1 インバウンド連携機能の定義

項目	指定するタグ	設定内容
Deploy ツール上に表示されるリソースアダプタの名称	<display-name>	「TP1_Inbound_Adapter」を指定します。
リソースアダプタのコンフィグレーションプロパティの定義	<resourceadapter>-<config-property>	リソースアダプタのコンフィグレーションプロパティのプロパティ名とプロパティ値を指定します※1。
リソースアダプタの実行時プロパティの定義	<resourceadapter-runtime>-<property>	リソースアダプタの実行時プロパティのプロパティ名とプロパティ値を指定します※2。

注※1　リソースアダプタのコンフィグレーションプロパティのプロパティ名とプロパティ値を表 4-16 に示します。

注※2　リソースアダプタの実行時プロパティのプロパティ名とプロパティ値を表 4-17 に示します。

表 4-16　リソースアダプタのコンフィグレーションプロパティ

config-property-name	config-property-value
backlog_count	RPC 要求を待ち受けるポートの Listen キューの長さを指定します。
connection_timeout	RPC 要求の応答を送信する時のコネクション確立処理のタイムアウト時間を指定します。
ipc_sockctl_watchtime※	一時クローズ処理要求の応答待ち時間を指定します。
ipc_tcpnodelay※	OpenTP1 とのコネクションの接続で使用するソケットに TCP_NODELAY オプションを使用するかどうかを指定します。
max_connections※	RPC 要求を待ち受けるポートへの最大同時接続数を指定します。
node_id※	ノード識別子を指定します。
receive_buffer_size※	受信時の受信バッファサイズを指定します。
rpc_close_after_send※	コネクションを切断します。
rpc_max_thread_count※	最大同時 RPC 要求受け付け数を指定します。
rpc_receive_timeout	RPC 要求の受信のタイムアウト時間を指定します。
rpc_receive_timeout_interval	RPC 要求の受信のタイムアウトの監視間隔を指定します。
rpc_sockctl_highwater※	RPC 受信コネクションの一時クローズ処理開始のしきい値を指定します。
rpc_sockctl_lowwater※	RPC 受信コネクションの一時クローズ処理非対象とするコネクション数の割合を指定します。
scd_port	RPC 要求を待ち受けるポートのポート番号を指定します。
send_buffer_size※	応答時の送信バッファサイズを指定します。
send_retry_count	サーバの応答送信時の TCP/IP コネクションの接続でエラーが発生したときのリトライ回数を指定します。
send_retry_interval	サーバの応答送信時の TCP/IP コネクションの接続でエラーが発生したときのリトライ間隔を指定します。
service_group※	TP1 インバウンドアダプタのサービスグループ名を指定します。
tcp_receive_timeout	コネクションを接続してから，もしくはデータを受信してから次のデータを受信するまでのタイムアウト時間を指定します。
tcp_send_timeout	RPC 要求の応答を送信する際の，1 回のデータ送信ごとのタイムアウト時間を指定します。
tcp_send_timeout_interval	RPC 要求の応答を送信する際の，一回のデータ送信のタイムアウトの監視間隔を指定します。 なお，08-53 以降ではこのプロパティの設定は無視されます。
trn_max_connections※	RPC 応答または同期点電文の送受信で使用するコネクションの最大同時接続数を指定します。
trn_max_thread_count※	最大同時同期点処理数を指定します。
trn_port	同期点電文を待ち受けるポートのポート番号を指定します。
trn_sockctl_highwater※	RPC 送信コネクションの一時クローズ処理開始のしきい値を指定します。
trn_sockctl_lowwater※	RPC 送信コネクションの一時クローズ処理非対象とするコネクション数の割合を指定します。

注※

プロパティに設定する値の考え方については，「(4)　config-property-name の説明」を参照してください。

表 4-17　リソースアダプタの実行時プロパティ

property-name	property-value
MaxTPoolSize	Message-driven Bean（サービス）呼び出しスレッドの最大数を指定します。
MinTPoolSize	Message-driven Bean（サービス）呼び出しスレッドの最小数を指定します。
TPoolKeepalive	空いている Message-driven Bean（サービス）呼び出しスレッドの最大生存時間を指定します。

(4)　config-property-name の説明

リソースアダプタのコンフィグレーションプロパティに設定するプロパティ値について説明します。

なお，各プロパティの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1.11　TP1 インバウンドアダプタに設定する<config-property>に指定できるプロパティ」を参照してください。

(a)　サービスグループ名（service_group）

OpenTP1 でのサービスグループ名とは，サービスをまとめた単位であり，ネットワークで接続されるすべての OpenTP1 システムの中で一意な識別子です。OpenTP1 のユーザサービス定義の service_group 句に設定します。

TP1 インバウンドアダプタでのサービスグループ名とは，TP1 インバウンドアダプタが呼び出す Message-driven Bean（サービス）のグループです。OpenTP1 と同様にネットワークで接続されるすべての OpenTP1 システムの中で一意な識別子にしてください。サービスグループ名は Connector 属性ファイルの service_group プロパティに設定します。

(b)　ノード識別子（node_id）

OpenTP1 でのノード識別子とは，OpenTP1 のセットアップ単位に一意に割り当てる 4 文字の識別子のことです。OpenTP1 のシステム共通定義の node_id 句に設定します。

TP1 インバウンドアダプタのノード識別子とは，OpenTP1 と同様にネットワークで接続されるすべての OpenTP1 システムの中で一意に割り当てる 4 文字の識別子です。Connector 属性ファイルの node_id プロパティに設定します。

なお，ノード識別子は，トランザクションの障害が発生し，コマンドを使ってトランザクションを強制決着する際にノードを特定するための情報です。そのため，トランザクション連携機能を使用する場合，OpenTP1 システムの中で重複しない一意な文字列を設定しておくことを推奨します。トランザクション連携機能を使用しない場合，TP1 インバウンドアダプタのノード識別子はデフォルトの設定値のままで問題ありません。

(c)　RPC 要求の同時実行数（max_connections，rpc_max_thread_count）

RPC 要求の同時実行数は，次の表に示す Connector 属性ファイルのプロパティに設定します。

表 4-18　RPC 要求の同時実行数の設定で使用するプロパティ

設定項目	Connector 属性ファイルのプロパティ
RPC 受信コネクションの最大同時接続数	max_connections
最大同時 RPC 要求受け付け数	rpc_max_thread_count

max_connections プロパティには，TP1 インバウンドアダプタに同時に RPC 要求を送信する OpenTP1 のプロセス数を設定します。

rpc_max_thread_count プロパティには，TP1 インバウンドアダプタで処理する RPC 要求の処理時間とスループットから算出したスレッド数を設定します。

例えば，RPC 要求の処理時間が 50ms，RPC 要求のスループットが 200TPS を実現する場合に必要なスレッド数は，CPU 使用率を 100%としたときに，0.05（秒）×200（TPS）＝ 10（スレッド）となります。RPC 要求の処理時間は，各システムでの実測値とします。また，CPU100%以下を想定する場合は，それに応じた安全係数を考慮してください。

(d)　RPC 応答と同期点処理の同時実行数（trn_max_connections，trn_max_thread_count）

RPC 応答と同期点処理の同時実行数は，次の表に示す Connector 属性ファイルのプロパティに設定します。

表 4-19　RPC 応答と同期点処理の同時実行数の設定

設定項目	Connector 属性ファイルのプロパティ
RPC 送信コネクションの最大同時接続数	trn_max_connections
最大同時同期点処理数	trn_max_thread_count

trn_max_connections プロパティには，TP1 インバウンドアダプタから OpenTP1 へ RPC 応答を送信するコネクションの最大同時接続数を設定します。設定値は，RPC 受信コネクションの最大同時接続数（max_connections プロパティ）と同じ値を設定してください。

trn_max_thread_count プロパティには，TP1 インバウンドアダプタのトランザクション連携機能を使用する場合に指定する同期点処理の同時処理数を設定します。設定値は，最大同時 RPC 要求受け付け数（rpc_max_thread_count プロパティ）と同じ値を設定してください。トランザクション連携機能を使用しない場合は，値はデフォルトのままで問題ありません。

(e)　一時クローズ処理（rpc_sockctl_highwater，rpc_sockctl_lowwater，trn_sockctl_highwater，trn_sockctl_lowwater，ipc_sockctl_watchtime）

一時クローズ処理は，次の表に示す Connector 属性ファイルのプロパティに設定します。TP1 インバウンドアダプタと OpenTP1 の一時クローズ処理の設定項目の対応を次の表に示します。

表 4-20　一時クローズ処理の設定項目の対応

設定項目	TP1 インバウンドアダプタの Connector 属性ファイルのプロパティ	OpenTP1 のシステム共通定義の定義句
RPC 受信コネクションの一時クローズ処理開始のしきい値（%）	rpc_sockctl_highwater	ipc_sockctl_highwater※

設定項目	TP1 インバウンドアダプタの Connector 属性ファイルのプロパティ	OpenTP1 のシステム共通定義の定義句
RPC 受信コネクションの一時クローズ処理非対象とするコネクション数の割合（%）	rpc_sockctl_lowwater	ipc_sockctl_highwater※
RPC 送信コネクションの一時クローズ処理開始のしきい値（%）	trn_sockctl_highwater	ipc_sockctl_highwater※
RPC 送信コネクションの一時クローズ処理非対象とするコネクション数の割合（%）	trn_sockctl_lowwater	ipc_sockctl_highwater※
一時クローズ応答の待ち時間（秒）	ipc_sockctl_watchtime	ipc_sockctl_watchtime

注※

一時クローズ処理のしきい値と一時クローズ処理非対象とするコネクション数の割合をコンマ区切りで指定します。

rpc_sockctl_lowwater プロパティには，rpc_sockctl_highwater プロパティの設定値以下の値を設定してください。rpc_sockctl_highwater プロパティよりも大きい値を設定した場合は，rpc_sockctl_highwater プロパティと同じ値が設定されます。

trn_sockctl_lowwater プロパティには，trn_sockctl_highwater プロパティの設定値以下の値を設定してください。trn_sockctl_highwater プロパティよりも大きい値を設定した場合は，trn_sockctl_highwater プロパティと同じ値が設定されます。

また，コネクションの最大同時接続数および一時クローズ処理のしきい値には，小さい値を指定しないでください。コネクションの最大同時接続数（max_connections プロパティと trn_max_connections プロパティ）の設定値，および一時クローズ処理のしきい値（rpc_sockctl_highwater プロパティと trn_sockctl_highwater プロパティ）に小さい値を指定すると，一時クローズ処理が多発し，性能に影響を与えたり，通信障害が発生したりするおそれがあります。

(f)　TCP/IP の通信バッファサイズ（receive_buffer_size，send_buffer_size）

TCP/IP の受信バッファサイズは，Connector 属性ファイルの receive_buffer_size プロパティに設定します。また，TCP/IP の送信バッファサイズは，Connector 属性ファイルの send_buffer_size プロパティに設定します。

バッファサイズを調整する場合，receive_buffer_size プロパティの設定値は，OpenTP1 のユーザサービス定義の ipc_sendbuf_size オペランドの設定値に合わせることを推奨します。また，send_buffer_size プロパティの設定値は，OpenTP1 のユーザサービス定義の ipc_recvbuf_size オペランドの設定値に合わせることを推奨します。

なお，設定したバッファサイズは，Java SE の仕様上，設定したサイズどおりにならない場合があります。詳細は Java SE の API ドキュメントを参照してください。

(g)　コネクション切断オプション（rpc_close_after_send）

OpenTP1 と TP1 インバウンドアダプタ間のコネクションを接続すると，デフォルトの設定では RPC 通信および同期点処理後も接続したままとなります。

RPC 通信および同期点処理後にコネクションを切断するには，TP1 インバウンドアダプタと OpenTP1 のコネクション切断オプションを設定します。

コネクション切断オプションは次のように設定します。

- アプリケーションサーバの Connector 属性ファイルの rpc_close_after_send プロパティに true を指定
- OpenTP1 のシステム共通定義の rpc_close_after_send 句に Y を指定

なお，OpenTP1 の rpc_close_after_send 句は，ユーザサービス定義にも指定できます。OpenTP1 システム全体にコネクション切断オプションを有効にする場合はシステム共通定義に設定してください。サービス単位に有効にする場合はユーザサービス定義に設定してください。

OpenTP1 のシステム共通定義とユーザサービス定義の詳細は，マニュアル「OpenTP1 システム定義」を参照してください。

(h)　TCP_NODELAY オプション

OpenTP1 とのコネクションの接続で使用するソケットに TCP_NODELAY オプションを使用するかどうかを指定します。TCP_NODELAY オプションを使用すると，Nagle アルゴリズムが無効になるので，送信済みデータの応答待ちの状態でも遅延させることなくデータを送信できます。ただし，
TCP_NODELAY オプションを使用することで，INET ドメイン通信時の送信効率が低下し，ネットワークの負荷が大きくなる場合があります。

TCP_NODELAY オプションの設定は，Connector 属性ファイルの ipc_tcpnodelay プロパティに指定します。

なお，このプロパティを指定する場合は，receive_buffer_size プロパティ，send_buffer_size プロパティ，ネットワークの帯域などを考慮し，この機能の必要性を十分に検討してください。

4.13　OpenTP1 での設定

ここでは，次の OpenTP1 での設定について説明します。

- ユーザサービスネットワーク定義によるスケジューラダイレクトの設定
- 性能解析トレースの設定
- トランザクションの設定

OpenTP1 に関する詳細な設定や作業については，OpenTP1 のマニュアルを参照してください。

4.13.1　ユーザサービスネットワーク定義によるスケジューラダイレクトの設定

TP1 インバウンドアダプタをユーザサービスネットワーク定義によるスケジューラダイレクト機能を使って呼び出すために，次の定義を設定します。

- ユーザサービスネットワーク定義
- OpenTP1 のユーザサービス定義

(1)　ユーザサービスネットワーク定義

ユーザサービスネットワーク定義の設定項目を次の表に示します。

表 4-21　ユーザサービスネットワーク定義の設定項目

設定項目	設定する値
サービスグループ名	TP1 インバウンドアダプタの Connector 属性ファイルの service_group プロパティで定義したサービスグループ名を指定します。
ホスト名	アプリケーションサーバのホスト名を指定します※。
ポート番号	TP1 インバウンドアダプタの Connector 属性ファイルの scd_port プロパティで定義したポート番号を指定します。

注※　アプリケーションサーバのホスト名については，次の注意事項を参照してください。

! 注意事項

OpenTP1 では，RPC 送信元と送信先の IP アドレスを比較し，同一マシンであれば UNIX ドメイン通信，異なるマシンであれば INET ドメイン通信を実施します。TP1 インバウンドアダプタでは，INET ドメイン通信だけをサポートするため，OpenTP1 と TP1 インバウンドアダプタを同一マシンで利用する場合は，ユーザサービスネットワーク定義のホスト名には IP alias 機能によって割り振った送信元とは異なる IP アドレスを指定してください。

ユーザサービスネットワーク定義の例を次に示します。

```
dcsvgdef -g ＜サービスグループ名＞ -h ホスト名[:ポート番号]
```

(2)　OpenTP1 のユーザサービス定義

ユーザサービス定義の rpc_destination_mode に definition を設定します。definition を設定することで，dc_rpc_call 関数を呼び出すときに，ユーザサービスネットワーク定義の指定値から宛先を検索するようになります。

ユーザサービス定義の例を次に示します。

```
set rpc_destination_mode = definition
```

4.13.2　性能解析トレースの設定

TP1 インバウンドアダプタで OpenTP1 の性能解析トレース（PRF）の情報を引き継ぐため，システム共通定義に性能解析トレース（PRF）の定義を次の表に示す内容で設定してください。

表 4-22　性能解析トレース（PRF）の設定

設定内容	説明
set prf_trace=Y	性能検証用トレース情報を取得します。
set rpc_prf_trace_level=00010001	OpenTP1 の PRF 情報をアプリケーションサーバに伝播するために設定します。

4.13.3　トランザクションの設定

トランザクション連携機能を使用する場合，OpenTP1 には TP1 インバウンドアダプタとの連携でトランザクションを発生させるように設定する必要があります。

一方，トランザクション連携機能を使用しない場合は，OpenTP1 に TP1 インバウンドアダプタとの連携でトランザクションを発生させないように設定する必要があります。それぞれのトランザクションの設定について，SUP または SSP の場合，および MHP の場合に分けて表に示します。

表 4-23　トランザクションの設定（SUP または SPP の場合）

トランザクション連携の有無	設定内容		説明
	ユーザサービス定義	dc_rpc_call 関数	
トランザクション連携機能を使用する	set atomic_update=Y	引数 flags に DCNOFLAGS を指定します。	呼び出し元の OpenTP1 のサービスグループのプロセスでトランザクションを発生させ，TP1 インバウンドアダプタをトランザクションに含めます。
トランザクション連携機能を使用しない	set atomic_update=Y	引数 flags に DCNOFLAGS | DCRPC_TPNOTRAN を指定します。	呼び出し元の OpenTP1 のサービスグループのプロセスでトランザクションを発生させますが，TP1 インバウンドアダプタをトランザクションに含めません。
	set atomic_update=N	引数 flags に DCNOFLAGS または，DCNOFLAGS | DCRPC_TPNOTRAN を指定します。	呼び出し元の OpenTP1 のサービスグループのプロセスでトランザクションを発生させません。

注　OpenTP1 のユーザサービス定義や dc_rpc_call 関数の詳細は，マニュアル「OpenTP1 システム定義」またはマニュアル「OpenTP1 プログラム作成の手引」を参照してください。

表 4-24　トランザクションの設定（MHP の場合）

トランザクション連携の有無	設定内容			説明
	アプリケーション属性定義	ユーザサービス定義	dc_rpc_call 関数	
トランザクション連携機能を使用する	mcfaalcap コマンドの-n オプションの trnmode オペランドに"trn"を指定します。※	set atomic_update=Y	引数 flags に DCNOFLAGS を指定します。	呼び出し元の OpenTP1 のサービスグループのプロセスでトランザクションを発生させ，TP1 インバウンドアダプタをトランザクションに含めます。
トランザクション連携機能を使用しない	mcfaalcap コマンドの-n オプションの trnmode オペランドに"trn"を指定します。※	set atomic_update=Y	引数 flags に DCNOFLAGS | DCRPC_TPNOTRAN を指定します。	呼び出し元の OpenTP1 のサービスグループのプロセスでトランザクションを発生させますが，TP1 インバウンドアダプタをトランザクションに含めません。
	mcfaalcap コマンドの-n オプションの trnmode オペランドに"nontrn"を指定します。	set atomic_update=Y または set atomic_update=N	引数 flags に DCNOFLAGS または DCNOFLAGS | DCRPC_TPNOTRAN を指定します。	呼び出し元の OpenTP1 のサービスグループのプロセスでトランザクションを発生させません。

注　OpenTP1 のアプリケーション属性定義，ユーザサービス定義や dc_rpc_call 関数の詳細は，マニュアル「OpenTP1 システム定義」またはマニュアル「OpenTP1 プログラム作成の手引」を参照してください。

注※　mcfaalcap コマンドの-n オプションの trnmode オペランドに"trn"を設定し，set atomic_update=N を設定した場合，MHP は設定誤りのため異常終了します。

4.14　OpenTP1 とアプリケーションサーバ間のタイムアウトの設定

TP1 インバウンドアダプタでは，次のタイムアウトを設定できます。

- RPC 要求受信および RPC 応答送信時のタイムアウト
- トランザクションに関するタイムアウト

それぞれについて説明します。

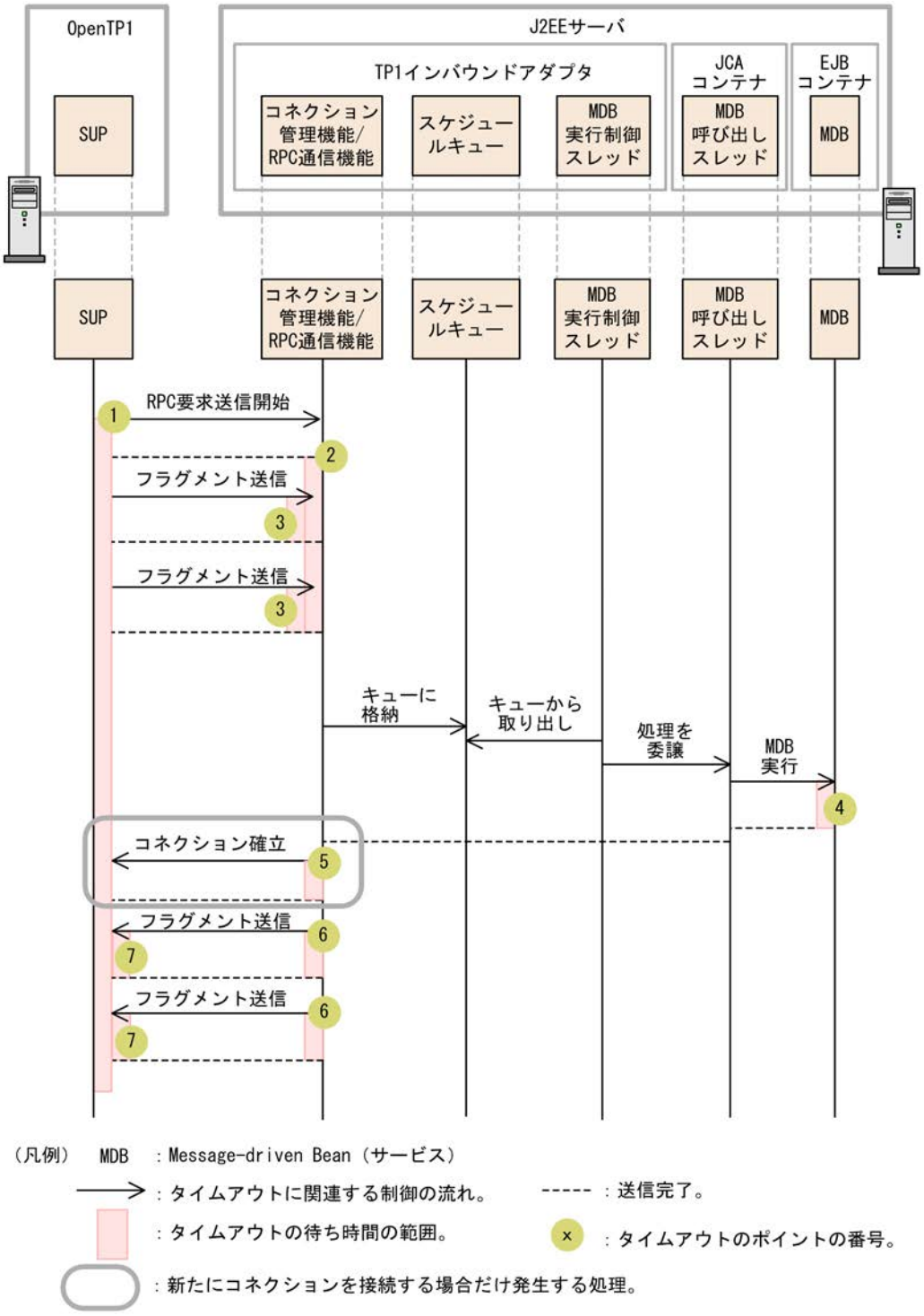
4.14.1　RPC 要求受信および RPC 応答送信時のタイムアウト

TP1 インバウンドアダプタでは，RPC 要求受信および RPC 応答送信時のタイムアウトを設定できます。これにより，ネットワークの障害や OpenTP1 の障害などが発生し，RPC 通信が待ち状態となった場合に，通信タイムアウトが発生することで障害を検知できます。

また，TP1 インバウンドアダプタでは Message-driven Bean（サービス）の実行時間タイムアウトも設定できます。これにより，業務処理で無限ループやデッドロックが発生した場合などに，実行時間タイムアウトによって強制的に業務処理を終了できます。

タイムアウトが設定できるポイントとタイムアウトの有効範囲を次の図に示します。

図 4-19　タイムアウトが設定できるポイントとタイムアウトの有効範囲



次に，この図のタイムアウトが設定できるポイントでのタイムアウトの設定方法について説明します。

(1)　OpenTP1 での設定

図 4-19 の 1 のポイントと 7 のポイントのタイムアウトの設定は，OpenTP1 で実施します。タイムアウトの設定方法を次の表に示します。

表 4-25　OpenTP1 でのタイムアウトの設定方法

図中番号	設定項目	設定ファイル	説明
1	RPC 最大応答待ち時間	ユーザサービス定義※ （watch_time）	RPC 要求を送信してからサービスの応答が返るまでの待ち時間を秒単位で指定します。
7	TCP 受信タイムアウト	ユーザサービス定義※ （ipc_header_recv_time）	データを受信してから通信制御用のデータを受信するまでのタイムアウト時間を秒単位で指定します。

注　タイムアウトが発生した場合，dc_rpc_call 関数にはエラーが返されます。タイムアウトが発生した場合の TP1 インバウンドアダプタの RPC エラー応答の詳細については，「4.17　TP1 インバウンドアダプタで発生する RPC エラー応答」を参照してください。

注※　詳細については，マニュアル「OpenTP1 システム定義」のユーザサービス定義の説明を参照してください。

(2)　アプリケーションサーバの実行環境での設定

図 4-19 の 2 のポイントから 6 のポイントまでのタイムアウトの設定は，アプリケーションサーバの実行環境で実施します。タイムアウトの設定方法を次の表に示します。

表 4-26　アプリケーションサーバの実行環境でのタイムアウトの設定方法

図中番号	設定項目	設定ファイル※1	説明
2	RPC 要求の受信タイムアウト	Connector 属性ファイル （rpc_receive_timeout プロパティ）	OpenTP1 から最初の要求電文を受信し始めてから，すべての要求電文の受信を完了するまでのタイムアウト時間を秒単位で指定します。なお，すべての要求電文を受信する前にコネクションが切断された RPC 要求もタイムアウトの対象に含まれます※2。
3	TCP 受信タイムアウト	Connector 属性ファイル （tcp_receive_timeout プロパティ）	要求電文の受信（read）を開始してから完了するまでのタイムアウト時間を秒単位で指定します。
4	サービス実行タイムアウト※3	cosminexus.xml （<ejb-method-observation-timeout>タグ）	Message-driven Bean（サービス）の実行時間を秒単位で指定します。
		cosminexus.xml （<method-observation-recovery-mode>タグ）	「thread」を指定します。
5	TCP 送信コネクション確立タイムアウト	Connector 属性ファイル （connection_timeout プロパティ）	電文を送信する際，新たにコネクションを接続する場合のコネクション確立処理のタイムアウト時間※4 を秒単位で指定します。
6	TCP 送信タイムアウト	Connector 属性ファイル （tcp_send_timeout プロパティ）	応答電文の送信（write）を開始してから完了するまでのタイムアウト時間を秒単位で指定します。

注　タイムアウトが発生した場合，エラーメッセージが出力されます。また，dc_rpc_call 関数にはエラーが返されます。タイムアウトが発生した場合の TP1 インバウンドアダプタの RPC エラー応答の詳細については，「4.17　TP1 インバウンドアダプタで発生する RPC エラー応答」を参照してください。

注※1　属性ファイルの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1　Connector 属性ファイル」を参照してください。

注※2　分割された要求電文の受信中は TCP 受信タイムアウトのチェックが優先されるため，RPC 要求の受信タイムアウトは発生しません。また，Connector 属性ファイルの rpc_receive_timeout_interval プロパティに指定した時間の分，タイムアウトの発生が遅れる場合があります。

注※3　サービス実行のタイムアウトには，アプリケーションサーバのメソッドキャンセル機能を使用します。タイムアウトが発生した場合，Message-driven Bean（サービス）は，EJB 仕様で定められたシステム例外が発生した場合と同等の動作をしたあとで，OpenTP1 に RPC エラーを応答します。

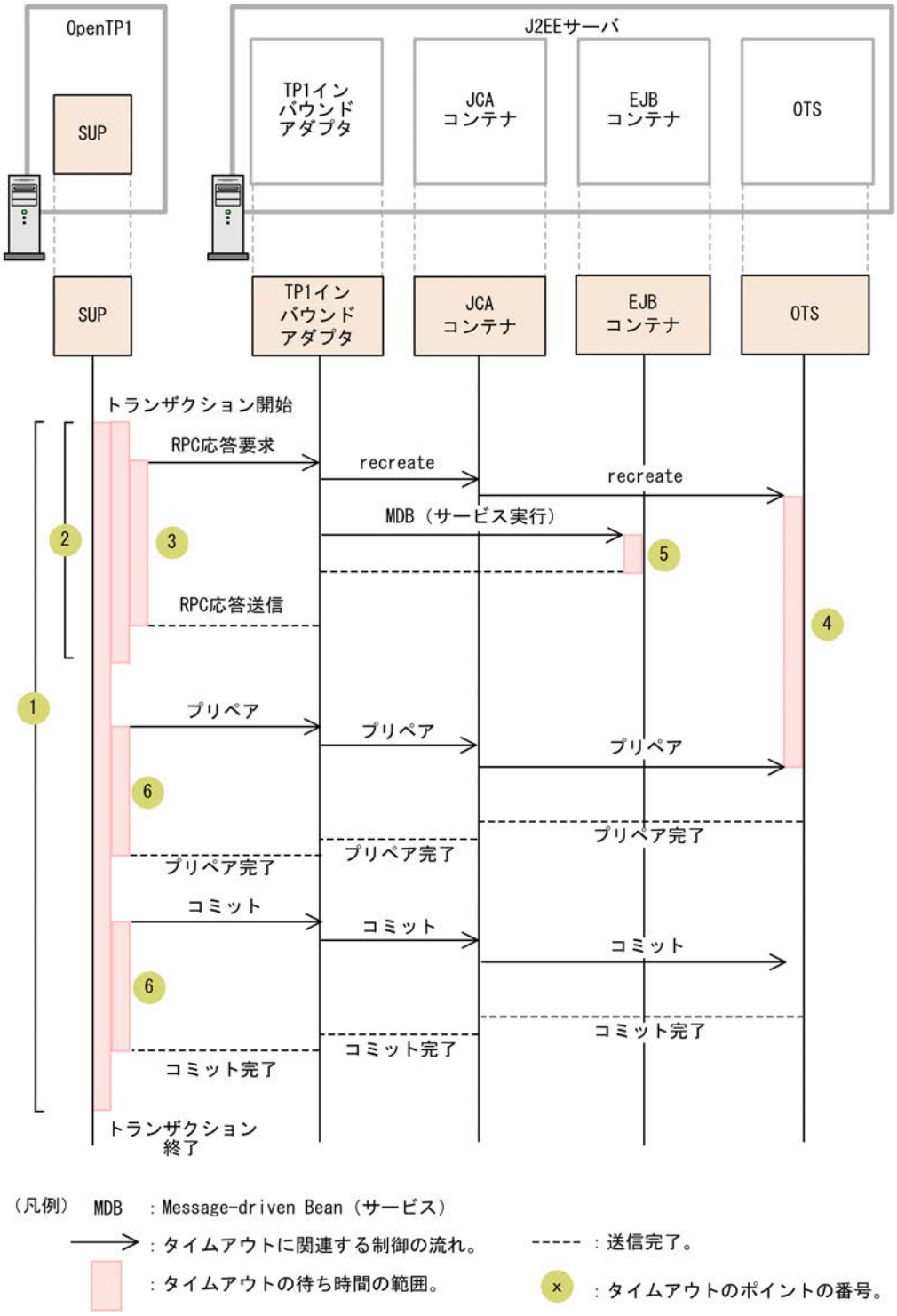
注※4　RPC 応答送信時のコネクション確立処理のタイムアウト時間は，Connector 属性ファイルの connection_timeout プロパティに設定します。タイムアウト発生時，コネクション確立処理はリトライしないで RPC 応答送信を終了します。ただし，OS に設定されている TCP 通信のタイムアウト値によっては，Connector 属性ファイルの connection_timeout プロパティの設定時間よりも早くコネクション確立処理がタイムアウトすることがあります。この場合, Connector 属性ファイルの send_retry_count プロパティと send_retry_interval プロパティの設定値に従って，コネクション確立処理をリトライします。TCP 通信のタイムアウトについては，OS のドキュメントを参照してください。

4.14.2　トランザクションに関するタイムアウト

TP1 インバウンドアダプタでは，トランザクションを開始してから決着処理を開始するまでのタイムアウトを設定できます。タイムアウトを設定しておくことで，ネットワークの障害や OpenTP1 の障害などが発生して，トランザクションが決着できない場合に，自動でロールバックできます。

OpenTP1 での設定，およびトランザクションに関するタイムアウトの有効範囲と設定方法を次に示します。

図 4-20　トランザクションに関するタイムアウトの有効範囲



(1)　OpenTP1 での設定

図 4-20 の 1 のポイントから 3 のポイント，および 6 のポイントのタイムアウトの設定は，OpenTP1 で実施します。タイムアウトの設定方法を次の表に示します。

表 4-27　OpenTP1 でのタイムアウトの設定方法

図中番号	設定項目	設定ファイル※	説明
1	トランザクション完了限界時間	ユーザサービス定義 （trn_completion_limit_time）	トランザクションの開始から終了までの時間を指定します。
2	トランザクションブランチ限界経過時間	ユーザサービス定義 （trn_expiration_time）	トランザクションの開始から同期点処理を開始するまでの時間を指定します。
3	RPC 最大応答待ち時間	ユーザサービス定義 （watch_time）	RPC 要求を送信してからサービスの応答が返るまでの待ち時間を秒単位で指定します。
6	トランザクション同期点処理時の最大通信待ち時間	ユーザサービス定義 （trn_watch_time）	トランザクションの同期点処理で，トランザクションブランチ間の通信の受信待ち時間を指定します。

注　タイムアウトが発生した場合，dc_rpc_call 関数にエラーが返されます。タイムアウトが発生した場合の TP1 インバウンドアダプタの RPC エラー応答の詳細については，「4.17　TP1 インバウンドアダプタで発生する RPC エラー応答」を参照してください。

注※　ユーザサービス定義の詳細については，マニュアル「OpenTP1 システム定義」のユーザサービス定義の説明を参照してください。

(2)　アプリケーションサーバの実行環境での設定

図 4-20 の 4 のポイントおよび 5 のポイントのタイムアウトの設定は，アプリケーションサーバの実行環境で実施します。タイムアウトの設定方法を次の表に示します。

表 4-28　アプリケーションサーバの実行環境でのタイムアウトの設定方法

図中番号	設定項目	設定ファイル※1	説明
4	トランザクションタイムアウト時間	cosminexus.xml （<ejb-transaction-timeout>タグ）	アプリケーションサーバ内でのトランザクション開始からトランザクション決着処理開始までの時間を指定します。
		J2EE サーバ用ユーザプロパティファイル （ejbserver.jta.TransactionManager.defaultTimeOut）	
5	サービス実行タイムアウト※2	cosminexus.xml （<ejb-method-observation-timeout>タグ）	Message-driven Bean（サービス）の実行時間を秒単位で指定します。
		cosminexus.xml （<method-observation-recovery-mode>タグ）	「thread」を指定します。

注　タイムアウトが発生した場合，エラーメッセージが出力されます。また，dc_rpc_call 関数にエラーが返されます。タイムアウトが発生した場合の TP1 インバウンドアダプタの RPC エラー応答の詳細については，「4.17　TP1 インバウンドアダプタで発生する RPC エラー応答」を参照してください。

注※1　cosminexus.xml の詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「2.　アプリケーション属性ファイル（cosminexus.xml）」を参照してください。なお，J2EE サーバ用ユーザプロパティファイルについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「2.4　usrconf.properties（J2EE サーバ用ユーザプロパティファイル）」を参照してください。

注※2　サービス実行タイムアウトは，アプリケーションサーバのメソッドキャンセル機能を使用します。メソッドキャンセル機能の詳細は，マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「5.3.2　J2EE アプリケーション実行時間の監視とは」を参照してください。

4.15　TP1 インバウンドアダプタと Message-driven Bean（サービス）の開始と終了

TP1 インバウンドアダプタと Message-driven Bean（サービス）の開始と終了は，次の手順で実施してください。

1. TP1 インバウンドアダプタを開始します。
2. Message-driven Bean（サービス）を開始します。
3. Message-driven Bean（サービス）を終了します。
4. TP1 インバウンドアダプタを終了します。

ここでは，TP1 インバウンドアダプタと Message-driven Bean（サービス）の開始と終了について説明します。

4.15.1　TP1 インバウンドアダプタの開始と終了

TP1 インバウンドアダプタの開始と終了について説明します。

(1)　TP1 インバウンドアダプタの開始

TP1 インバウンドアダプタは，cjstartrar コマンドで開始します。

TP1 インバウンドアダプタを開始すると，Connector 属性ファイルの定義内容が読み込まれます。

service_group プロパティが不正な場合，エラーメッセージを出力して TP1 インバウンドアダプタの開始処理を終了します。出力するメッセージを次に示します。

service_group プロパティが指定されていない場合
　エラーメッセージ（KDJE58302-E）

設定範囲外の値を指定した場合
　エラーメッセージ（KDJE58300-E）

すでに同じ名前のサービスグループ名の TP1 インバウンドアダプタが開始済みの場合
　エラーメッセージ（KDJE58303-E）

そのほかの Connector 属性ファイルの定義が不正な場合
　警告メッセージ（KDJE58301-W）を出力し，デフォルト値を適用して TP1 インバウンドアダプタの開始処理を続行します。

TP1 インバウンドアダプタの開始によって，リソースアダプタの Connector 属性ファイルに指定した番号のポートが開かれ，RPC 通信の待ち受けが開始されます。

指定した番号のポートが開けない場合，エラーメッセージが出力されて，TP1 インバウンドアダプタの開始に失敗します。Connector 属性ファイルに指定するプロパティについては，「4.12.2　リソースアダプタの設定」を参照してください。

なお，TP1 インバウンドアダプタを開始した時点では，RPC 通信を待ち受けるポートは開いていますが，Message-driven Bean（サービス）は開始していません。Message-driven Bean（サービス）が開始していない状態で，OpenTP1 から RPC 要求を受信した場合は，エラーメッセージが出力されて RPC エラーが返されます。

(2)　TP1 インバウンドアダプタの終了

TP1 インバウンドアダプタは，cjstoprar コマンドで終了します。

TP1 インバウンドアダプタを終了する場合，TP1 インバウンドアダプタを使用しているすべての Message-driven Bean（サービス）をあらかじめ停止しておいてください。

TP1 インバウンドアダプタは，次に示す手順に沿って終了してください。

1. TP1 インバウンドアダプタを使用しているすべての Message-driven Bean（サービス）を停止します。
2. Message-driven Bean（サービス）が使用していたすべての Outbound のリソースアダプタ（DBConnector）を停止します。
3. TP1 インバウンドアダプタを終了します。

TP1 インバウンドアダプタの終了時に Message-driven Bean（サービス）が起動されている場合は，エラーメッセージ（KDJE48547-E）が出力され，TP1 インバウンドアダプタの終了に失敗します。なお，正常に終了した場合は，ログファイルに KDJE58351-I が出力されます。

トランザクション連携機能を使用している場合，手順 3.を手順 2.の先に実施すると，OpenTP1 とアプリケーションサーバに未決着のトランザクションが残ってしまうおそれがあります。この場合，TP1 インバウンドアダプタを再開始し，未決着のトランザクションを決着させてください。詳細は「4.9.8(4)　通信障害が発生した場合」を参照してください。

4.15.2　Message-driven Bean（サービス）の開始と終了

Message-driven Bean（サービス）の開始と終了について説明します。

! 注意事項

TP1 インバウンド連携機能使用時に Message-driven Bean をリロードする場合は，OpenTP1 からの RPC の発行を止めてからリロードしてください。リロード中に RPC を発行した場合は，Message-driven Bean は終了していると見なされ，RPC の発行元にはエラー応答が返ります。

(1)　Message-driven Bean（サービス）の開始

TP1 インバウンドアダプタを使用する J2EE アプリケーションを cjstartapp コマンドで起動すると，Message-driven Bean（サービス）は開始します。

ActivationSpec の設定値が不正な場合，エラーメッセージが出力され，Message-driven Bean（サービス）の起動は失敗します。ActivationSpec の設定については，「4.10.5　ejb-jar.xml の定義」を参照してください。

(2)　終了

Message-driven Bean（サービス）は，TP1 インバウンドアダプタを使用する J2EE アプリケーションが次のどちらかで停止した場合に終了します。

- cjstopapp コマンドで通常停止した場合
- cjstopapp コマンドの-force オプションまたは-cancel オプションを指定して強制停止した場合

Message-driven Bean（サービス）の終了処理中は，新たな RPC 電文を受け付けないようにスケジュールキューは停止されます。また，未処理の RPC 要求と実行中の Message-driven Bean（サービス）は次のように処理されます。

新たに受信した未処理の RPC 要求

リクエスト受け付けスレッドが受け付けた RPC 要求です。RPC 要求を破棄して，OpenTP1 に RPC エラーを応答します。

RPC エラーコードとして DCRPCER_NO_SUCH_SERVICE（-311）が返され，エラーメッセージが出力されます。

受信済みの未処理の RPC 要求

スケジュールキューに登録済みの RPC 要求です。RPC 要求を破棄して，OpenTP1 に RPC エラーを応答します。

RPC エラーコードとして DCRPCER_SERVICE_TERMINATING（-313）が返され，エラーメッセージが出力されます。

実行中の Message-driven Bean（サービス）

停止種別によって処理が次のように異なります。

- 通常停止の場合

  終了するまで待機します。

- 強制停止の場合

  強制終了（メソッドキャンセル）されます。

  RPC エラーコードに DCRPCER_TIMED_OUT（-307）が返され，エラーメッセージが出力されます。

  強制終了（メソッドキャンセル）は，Message-driven Bean（サービス）実行中にだけ実行できます。なお，保護区では強制終了（メソッドキャンセル）できません。

参考

正常に開始／終了した場合は，ログファイルにメッセージが出力されます。

4.16　性能解析トレース情報の引き継ぎ

TP1 インバウンドアダプタを使用して OpenTP1 の SUP からアプリケーションサーバ上の業務を呼び出す場合に，OpenTP1 の SPP を呼び出す場合と同様に性能解析情報を引き継ぎ，OpenTP1 とアプリケーションサーバの処理を突き合わせることができます。

この節では，性能解析情報を引き継ぐための準備，OpenTP1 とアプリケーションサーバの性能解析情報の対応，および性能解析情報を突き合わせる方法について説明します。

4.16.1　性能解析情報を引き継ぐための準備

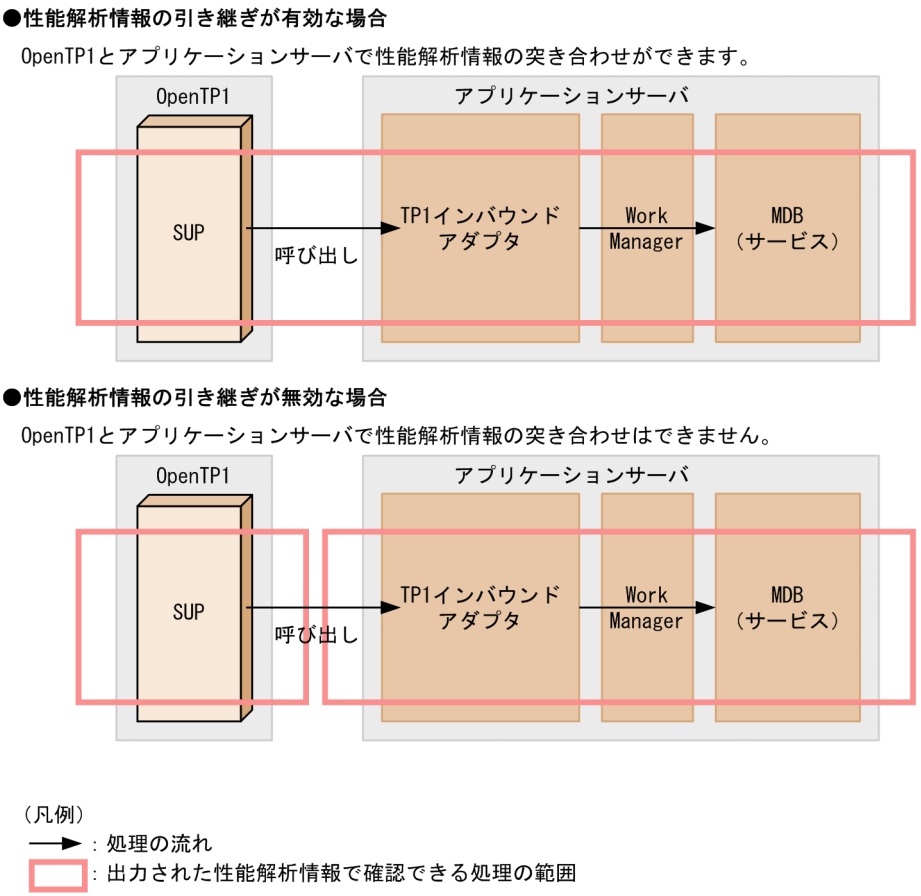
性能解析情報を引き継ぐ場合，OpenTP1 のシステム共通定義に次の設定が必要です。

- set prf_trace=Y
- set rpc_prf_trace_level=00010001

これらの定義がある場合，OpenTP1，アプリケーションサーバ間で性能解析情報が引き継がれます。

性能解析情報の引き継ぎ有無と性能解析情報で確認できる処理の範囲を次の図に示します。

図 4–21　性能解析情報の引き継ぎ有無と性能解析情報で確認できる処理の範囲



4.16.2　性能解析情報の対応

TP1 インバウンドアダプタの RPC 通信機能では，性能解析情報の引き継ぎが有効な場合，RPC 要求電文を解析するときに OpenTP1 の RPC 要求電文に設定されている性能解析情報を取得して，アプリケーションサーバの性能解析情報として使用します。具体的には，TP1 インバウンドアダプタの処理の延長で出力される RootAP IP と RootAP CommNo.に，OpenTP1 の呼び出し元の IP とルート RPC 通信番号がそれぞれ引き継がれます。

引き継がれる情報の対応を次の表に示します。

表 4-29　引き継がれる情報の対応

アプリケーションサーバの性能解析情報	引き継ぎ元の OpenTP1 の性能解析情報
RootAP IP	呼び出し元の OpenTP1 の IP
RootAP PID	0000（固定）
RootAP CommNo.	呼び出し元の OpenTP1 のルート RPC 通信番号

また，TP1 インバウンドアダプタの入り口で出力される PRF トレース（0xAA00）では，RootAP IP と RootAP CommNo.に加えて，オプション文字列（OPT）に OpenTP1 のルート RPC ノード識別子などが出力されます。

PRF トレース（0xAA00）で出力される情報を次の表に示します。

表 4-30　PRF トレース（0xAA00）で出力される情報

アプリケーションサーバの性能解析情報	引き継ぎ元の OpenTP1 の性能解析情報
RootAP IP	呼び出し元の OpenTP1 の IP
RootAP PID	0000（固定）
RootAP CommNo.	呼び出し元の OpenTP1 のルート RPC 通信番号
INT	サービスグループ名
OPR	サービス名
OPT	呼び出し元の OpenTP1 のルート RPC ノード識別子（4 文字）。なお，トランザクショナル RPC の場合は，ノード識別子のあとにトランザクショングローバル識別子。

これらの情報を使用して性能解析情報を突き合わせることで，OpenTP1 とアプリケーションサーバ間での処理を追跡できます。

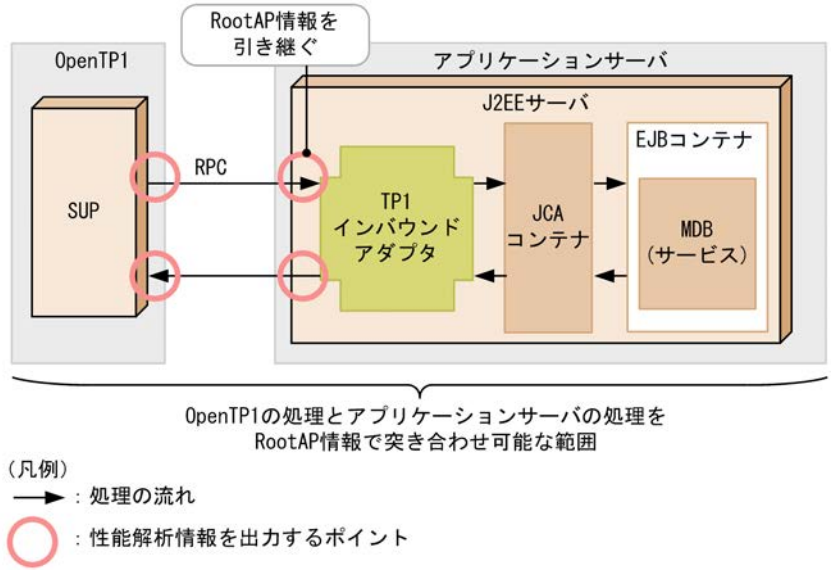
また，性能解析トレース情報の引き継ぎはトランザクション連携機能を使用する場合と使用しない場合とで異なります。それぞれの場合の性能解析トレース情報の引き継ぎについて説明します。

4.16.3　トランザクション連携機能を使用しない場合の性能解析トレース情報の引き継ぎ

トランザクション連携機能を使用しない場合の性能解析トレース情報の引き継ぎについて説明します。ここでは，アプリケーションサーバの性能解析情報と引き継ぎ元の OpenTP1 の性能解析情報を突き合わせることによって，OpenTP1 とアプリケーションサーバ間での RPC 通信を追跡する例を示します。

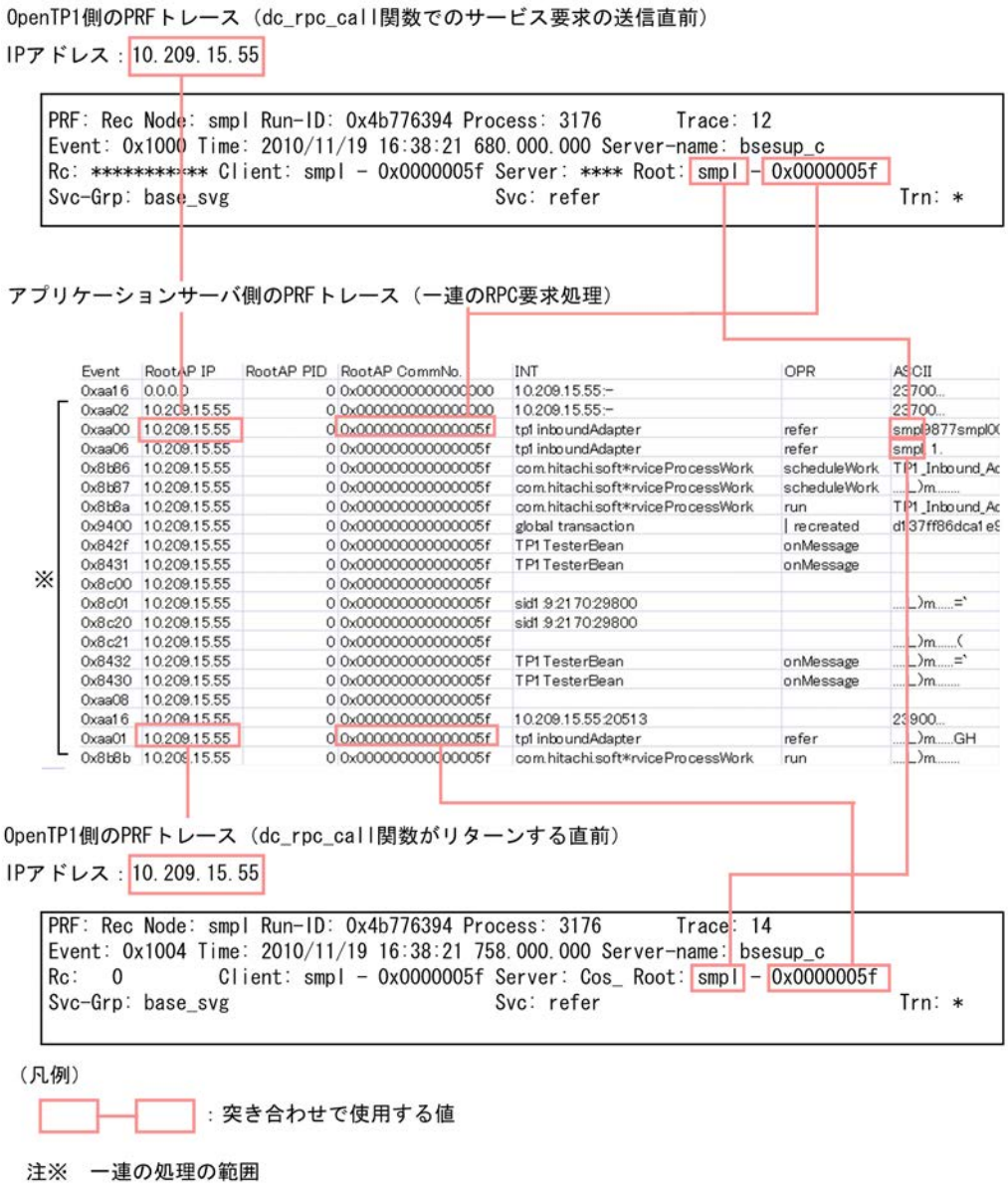
RPC の実行によって突き合わせ可能な範囲を次の図に示します。

図 4–22　RPC の実行によって突き合わせ可能な範囲



OpenTP1 の dc_rpc_call 関数でのサービス要求の送信直前，および dc_rpc_call 関数のリターン直前の PRF トレースと，アプリケーションサーバの PRF トレースの突き合わせの例を次の図に示します。

図 4-23　突き合わせの例



Event	RootAP IP	RootAP PID	RootAP CommNo.	INT	OPR	ASCII
0xaa16	0.0.0.0	0	0x0000000000000000	10.209.15.55:-		23700...
0xaa02	10.209.15.55	0	0x0000000000000000	10.209.15.55:-		23700...
0xaa00	10.209.15.55	0	0x000000000000005f	tp1 inboundAdapter	refer	smpl9877smpl0(
0xaa06	10.209.15.55	0	0x000000000000005f	tp1 inboundAdapter	refer	smpl 1.
0x8b86	10.209.15.55	0	0x000000000000005f	com.hitachi.soft*rviceProcessWork	scheduleWork	TP1_Inbound_Ac
0x8b87	10.209.15.55	0	0x000000000000005f	com.hitachi.soft*rviceProcessWork	scheduleWork	....)m.....
0x8b8a	10.209.15.55	0	0x000000000000005f	com.hitachi.soft*rviceProcessWork	run	TP1_Inbound_Ac
0x9400	10.209.15.55	0	0x000000000000005f	global transaction	recreated	d137ff86dca1e9
0x842f	10.209.15.55	0	0x000000000000005f	TP1TesterBean	onMessage	
0x8431	10.209.15.55	0	0x000000000000005f	TP1TesterBean	onMessage	
0x8c00	10.209.15.55	0	0x000000000000005f			
0x8c01	10.209.15.55	0	0x000000000000005f	sid1:9:2170:29800		....)m....=`
0x8c20	10.209.15.55	0	0x000000000000005f	sid1:9:2170:29800		
0x8c21	10.209.15.55	0	0x000000000000005f			....)m.....(
0x8432	10.209.15.55	0	0x000000000000005f	TP1TesterBean	onMessage	....)m....=`
0x8430	10.209.15.55	0	0x000000000000005f	TP1TesterBean	onMessage	....)m.....
0xaa08	10.209.15.55	0	0x000000000000005f			
0xaa16	10.209.15.55	0	0x000000000000005f	10.209.15.55:20513		23900...
0xaa01	10.209.15.55	0	0x000000000000005f	tp1 inboundAdapter	refer	....)m....GH
0x8b8b	10.209.15.55	0	0x000000000000005f	com.hitachi.soft*rviceProcessWork	run	....)m.....

OpenTP1 のルート OpenTP1 識別子は，アプリケーションサーバの PRF トレースの 0xAA00，0xAA06 の OPT（ASCII 列）に出力されます。

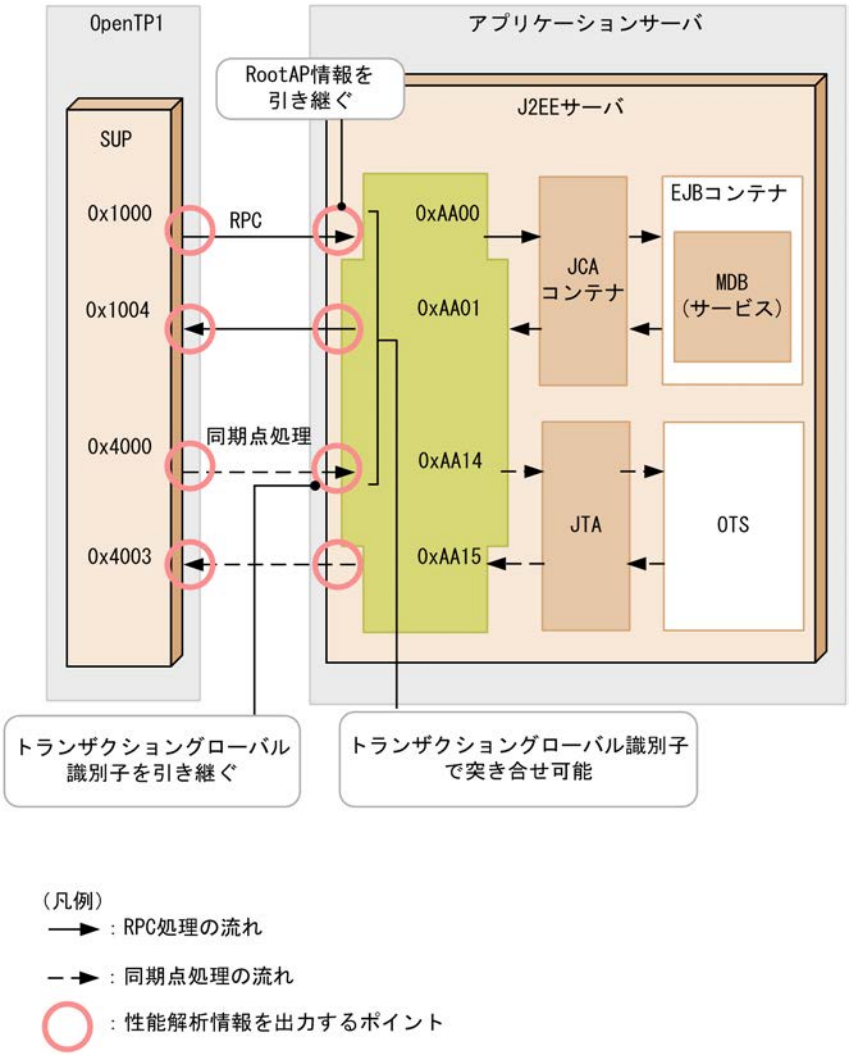
なお，OpenTP1 のルート通信番号とアプリケーションサーバの RootAP 情報は，通信番号のけたが異なります。このため，数値は同じですが，文字列としては一致しません。

4.16.4　トランザクション連携機能を使用する場合の性能解析トレース情報の引き継ぎ

トランザクション連携機能を使用する場合の性能解析トレース情報の引き継ぎについて説明します。ここでは，アプリケーションサーバの性能解析情報と引き継ぎ元の OpenTP1 の性能解析情報を突き合わせることによって，OpenTP1 とアプリケーションサーバ間での RPC 通信を追跡する例を示します。

RPC および同期点処理の実行によって突き合わせ可能な範囲を次の図に示します。

図 4-24　RPC および同期点処理の実行によって突き合わせ可能な範囲



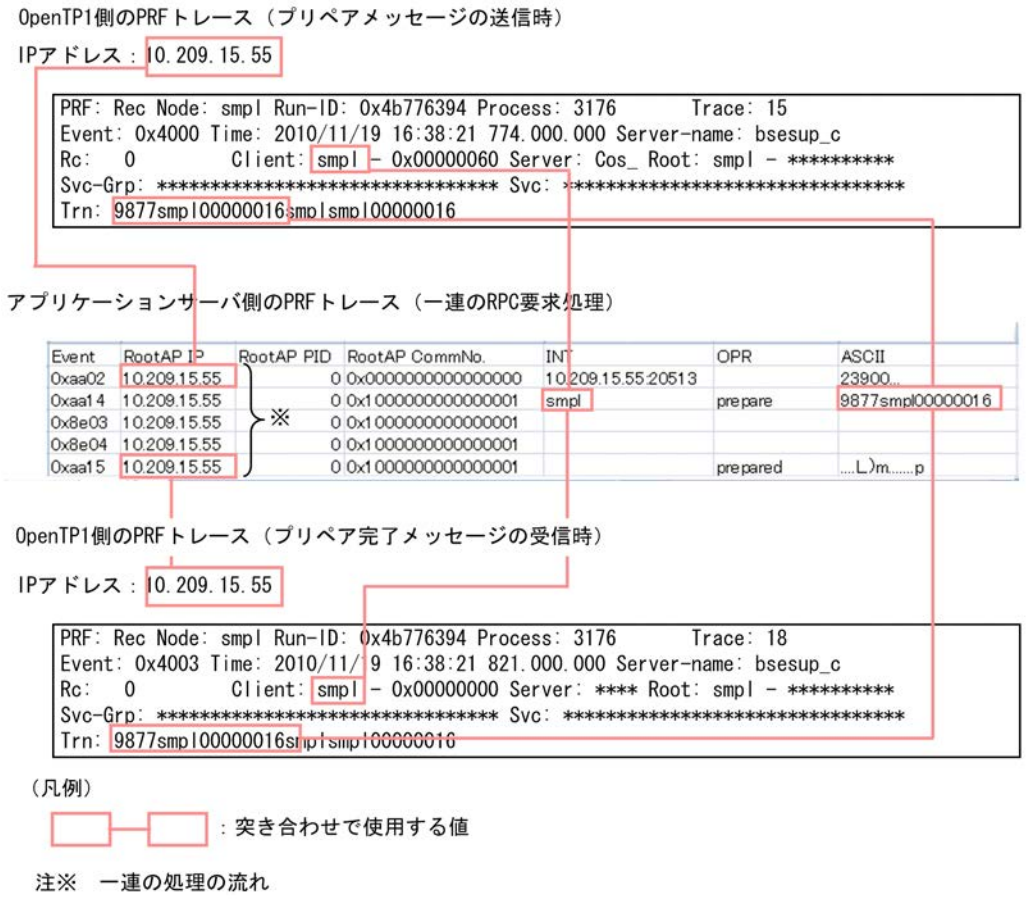
次に示す性能解析トレース情報を突き合わせて，トレース情報を追跡します。

- RPC 要求を受信したときの OpenTP1 と TP1 インバウンドアダプタ間の性能解析トレース情報
  アプリケーションサーバの性能解析情報と引き継ぎ元の OpenTP1 の性能解析情報を突き合わせます。詳細は「4.16.3　トランザクション連携機能を使用しない場合の性能解析トレース情報の引き継ぎ」を参照してください。
- 同期点処理時の OpenTP1 と TP1 インバウンドアダプタ間の性能解析トレース情報
  トランザクショングローバル識別子を突き合わせます。詳細は「(1)　同期点処理での OpenTP1 と TP1 インバウンドアダプタ間の性能解析トレース情報の突き合わせ」を参照してください。
- アプリケーションサーバ内での RPC 要求受信時と同期点処理時の間の性能解析トレース情報
  PRF トレース（0xAA00）で出力したトランザクショングローバル識別子，および同期点処理時に出力するトランザクショングローバル識別子を突き合わせます。詳細は「(2)　RPC 要求と同期点処理間の性能解析トレース情報の突き合わせ」を参照してください。

(1)　同期点処理での OpenTP1 と TP1 インバウンドアダプタ間の性能解析トレース情報の突き合わせ

同期点処理では，OpenTP1 から引き継がれるトランザクショングローバル識別子を突き合わせます。アプリケーションサーバの性能解析情報と引き継ぎ元の OpenTP1 の性能解析情報を突き合わせることによって，OpenTP1 とアプリケーションサーバ間での同期点処理を追跡する例を示します。

図 4-25　同期点処理での PRF トレースの突き合わせの例



OpenTP1 の PRF（プリペアメッセージの送信時の場合は 0x4000）の Trn の先頭 16 文字と，0xAA14 の OPT（ASCII 列）を突き合わせます。

また，0xAA14 の RootAP 情報の IP アドレス，および INT に出力する送信元のノード識別子も OpenTP1 の PRF と突き合わせます。

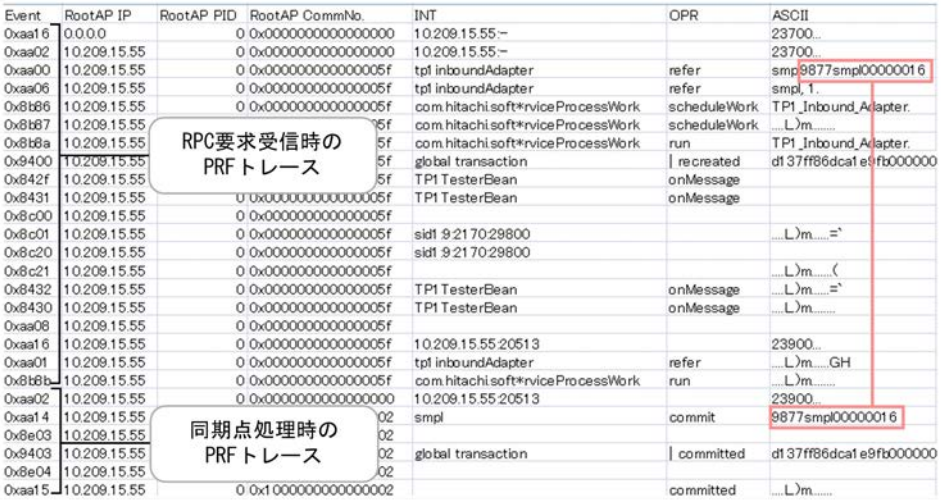
応答も同様に 0xAA14 の OPT（ASCII 列）と OpenTP1 の PRF（プリペア完了メッセージの受信時であれば 0x4003）の Trn の先頭 16 文字と突き合わせます。

(2)　RPC 要求と同期点処理間の性能解析トレース情報の突き合わせ

RPC 要求と同期点処理間の PRF トレースは，0xAA00 で出力した OPT（ASCII 列）の 4 文字目から 19 文字目のトランザクショングローバル識別子と，同期点処理時に出力される 0xAA14 の OPT（ASCII 列）のトランザクショングローバル識別子を突き合わせます。

RPC 要求と同期点処理間の PRF トレースの突き合わせの例を次に示します。

図 4–26　RPC 要求と同期点処理間の PRF トレースの突き合わせの例



Event	RootAP IP	RootAP PID	RootAP CommNo.	INT	OPR	ASCII
0xaa16	0.0.0.0	0	0x0000000000000000	10.209.15.55:−		23700...
0xaa02	10.209.15.55	0	0x0000000000000000	10.209.15.55:−		23700...
0xaa00	10.209.15.55	0	0x000000000000005f	tp1 inboundAdapter	refer	smp9877smpl00000016
0xaa06	10.209.15.55	0	0x000000000000005f	tp1 inboundAdapter	refer	smpl, 1,
0x8b86	10.209.15.55	0	0x000000000000005f	com.hitachi.soft*rviceProcessWork	scheduleWork	TP1_Inbound_Adapter.
0x8b87	10.209.15.55		5f	com.hitachi.soft*rviceProcessWork	scheduleWork	....L)m.....
0x8b8a	10.209.15.55		5f	com.hitachi.soft*rviceProcessWork	run	TP1_Inbound_Adapter.
0x9400	10.209.15.55		5f	global transaction	| recreated	d137ff86dca1e9fb000000
0x842f	10.209.15.55		5f	TP1 TesterBean	onMessage	
0x8431	10.209.15.55	0	0x000000000000005f	TP1 TesterBean	onMessage	
0x8c00	10.209.15.55	0	0x000000000000005f			
0x8c01	10.209.15.55	0	0x000000000000005f	sid1:9:21:70:29800		....L)m.....=`
0x8c20	10.209.15.55	0	0x000000000000005f	sid1:9:21:70:29800		
0x8c21	10.209.15.55	0	0x000000000000005f			....L)m.....(
0x8432	10.209.15.55	0	0x000000000000005f	TP1 TesterBean	onMessage	....L)m.....=`
0x8430	10.209.15.55	0	0x000000000000005f	TP1 TesterBean	onMessage	....L)m.....
0xaa08	10.209.15.55	0	0x000000000000005f			
0xaa16	10.209.15.55	0	0x000000000000005f	10.209.15.55:20513		23900...
0xaa01	10.209.15.55	0	0x000000000000005f	tp1 inboundAdapter	refer	....L)m.....GH
0x8b8b	10.209.15.55	0	0x000000000000005f	com.hitachi.soft*rviceProcessWork	run	....L)m.....
0xaa02	10.209.15.55	0	0x0000000000000000	10.209.15.55:20513		23900...
0xaa14	10.209.15.55		02	smpl	commit	9877smpl00000016
0x8e03	10.209.15.55		02			
0x9403	10.209.15.55		02	global transaction	| committed	d137ff86dca1e9fb000000
0x8e04	10.209.15.55		02			
0xaa15	10.209.15.55	0	0x1000000000000002		committed	....L)m.....

4.17　TP1 インバウンドアダプタで発生する RPC エラー応答

この節では，それぞれの RPC エラー応答について説明します。

なお，ここで説明しているエラー応答の一覧には，TP1 インバウンドアダプタに対して RPC 要求が送信された場合に発生するものだけを記載しています。OpenTP1 側で発生するエラー応答については，マニュアル「OpenTP1 プログラム作成リファレンス C 言語編」の dc_rpc_call のリターン値に関する説明を参照してください。

TP1 インバウンドアダプタの RPC エラー応答一覧を次の表に示します。

表 4-31　RPC エラー応答一覧

dc_rpc_call 関数の戻り値	発生原因	TP1 インバウンドアダプタの出力メッセージ
DCRPCER_NO_BUFS(-304)	TP1 インバウンドアダプタの電文組み立てリストが不足している状態で，TP1 インバウンドアダプタへトランザクショナル RPC ではない RPC 要求を送信した場合	KDJE58359-E
	TP1 インバウンドアダプタのスケジュールキューの電文滞留数が最大キュー長（Message-driven Bean（サービス）の DD の queue_max_length プロパティ）に達している状態で，TP1 インバウンドアダプタへ RPC 要求を送信した場合	KDJE58360-E
DCRPCER_NET_DOWN(-306)	TP1 インバウンドアダプタへ RPC 要求を送信する際にコネクションを確立できなかった場合	－
	TP1 インバウンドアダプタの backlog_count プロパティの設定数だけ Listen キューにコネクション確立要求が滞留している状態で，TP1 インバウンドアダプタへ RPC 要求を送信した場合	－
	TP1 インバウンドアダプタへ RPC 要求を送信中にコネクションが切断された場合	KDJE58354-E
	OpenTP1 のスケジューラダイレクト機能を使用しないで TP1 インバウンドアダプタへ RPC 要求を送信した場合	KDJE58355-E 詳細情報：Scheduler direct function is not used.
	TP1 インバウンドアダプタの電文組み立てリストが不足している状態で，TP1 インバウンドアダプタへ RPC 要求を送信した場合	KDJE58359-E
	TP1 インバウンドアダプタへ RPC 要求を送信中に，TP1 インバウンドアダプタの電文読み込み処理がタイムアウトした場合	KDJE58354-E
	TP1 インバウンドアダプタへ RPC 要求を送信中に，TP1 インバウンドアダプタの RPC 要求を受信し始めてから受信し終わるまでの時間がタイムアウトした場合	KDJE58358-E

dc_rpc_call 関数の戻り値	発生原因	TP1 インバウンドアダプタの出力メッセージ
DCRPCER_NET_DOWN(-306)	TP1 インバウンドアダプタが OpenTP1 からの RPC 要求ではない電文を受信した場合（プロトコルエラー）	KDJE58355-E 詳細情報：It is not the first request data.
DCRPCER_TIMED_OUT(-307)	TP1 インバウンドアダプタへ RPC 要求を送信中に，TP1 インバウンドアダプタの電文読み込み処理がタイムアウトした場合	KDJE58354-E
	TP1 インバウンドアダプタへ RPC 要求を送信中に，TP1 インバウンドアダプタの RPC 要求を受信し始めてから受信し終わるまでの時間がタイムアウトした場合	KDJE58358-E
	Message-driven Bean（サービス）の実行中に Message-driven Bean（サービス）の実行時間タイムアウトが発生した場合	KDJE58457-E
	TP1 インバウンドアダプタが OpenTP1 へ RPC 応答を送信する際のコネクション確立要求でタイムアウトが発生した場合	KDJE58400-E
	TP1 インバウンドアダプタが OpenTP1 へ RPC 応答を送信する際の電文書き込み処理でタイムアウトが発生した場合	KDJE58401-E
	dc_rpc_call 関数の引数 flags に DCNOFLAGS か DCRPC_TPNOTRAN 以外を指定して TP1 インバウンドアダプタへ RPC 要求を送信した場合	KDJE58356-E 詳細情報：The value of the flags argument of the dc_rpc_call function is illegal.
	OpenTP1 のスケジューラダイレクト機能を使用しないで TP1 インバウンドアダプタへ RPC 要求を送信した場合	KDJE58355-E 詳細情報：Scheduler direct function is not used.
	Message-driven Bean（サービス）の実行中に Message-driven Bean（サービス）を強制終了（cjstopapp コマンドの-force，-cancel オプション）した場合	KDJE58455-E
	TP1 インバウンドアダプタへ RPC 要求を送信中にコネクションが切断された場合	KDJE58354-E
	TP1 インバウンドアダプタが OpenTP1 へ RPC 応答を送信する際にコネクションを確立できなかった場合	KDJE58400-E
	TP1 インバウンドアダプタが OpenTP1 へ RPC 応答を送信する際の電文書き込み処理でエラーが発生した場合	KDJE58401-E
	TP1 インバウンドアダプタが OpenTP1 からの RPC 要求ではない電文を受信した場合（プロトコルエラー）	KDJE58355-E 詳細情報：It is not the first request data.

dc_rpc_call 関数の戻り値	発生原因	TP1 インバウンドアダプタの出力メッセージ
DCRPCER_NO_SUCH_SERVICE(-311)	dc_rpc_call 関数の引数 service に不正なサービス名を指定して TP1 インバウンドアダプタへ RPC 要求を送信した場合	KDJE58362-E
	Message-driven Bean（サービス）が開始していない，または終了中に TP1 インバウンドアダプタへ RPC 要求を送信した場合	KDJE58362-E
DCRPCER_SERVICE_TERMINATING(-313)	Message-driven Bean（サービス）が開始していない，または終了中に TP1 インバウンドアダプタのスケジュールキューに登録されていた RPC 要求が処理された場合	KDJE58456-E
DCRPCER_SERVICE_NOT_UP(-314)	dc_rpc_call 関数の引数 group に不正なサービスグループ名を指定して TP1 インバウンドアダプタへ RPC 要求を送信した場合	KDJE58357-E※
DCRPCER_SYSERR_AT_SERVER(-316)	未サポートバージョンの OpenTP1 から TP1 インバウンドアダプタへトランザクショナル RPC ではない RPC 要求を送信した場合	KDJE58356-E 詳細情報：Version of OpenTP1 is not supported.
	OpenTP1 でユーザデータ圧縮機能（システム共通定義の rpc_datacomp オペランドに Y）を使用して TP1 インバウンドアダプタへトランザクショナル RPC ではない RPC 要求を送信した場合	KDJE58356-E 詳細情報：User data compression function is not supported.
	TP1 インバウンドアダプタへトランザクショナル RPC ではない RPC 要求をした際に，Message-driven Bean（サービス）の実行中に例外が発生した場合，または Message-driven Bean（サービス）の実行結果に出力データを設定していない，もしくは J2EE サーバでシステム例外が発生した場合	KDJE58459-E
DCRPCER_NO_BUFS_RB(-323)	TP1 インバウンドアダプタの電文組み立てリストが不足している状態で，TP1 インバウンドアダプタへトランザクショナル RPC 要求を送信した場合	KDJE58359-E
DCRPCER_SYSERR_AT_SERVER_RB(-325)	未サポートバージョンの OpenTP1 から，TP1 インバウンドアダプタへトランザクショナル RPC 要求を送信した場合	KDJE58356-E 詳細情報：Version of OpenTP1 is not supported.
	OpenTP1 でユーザデータ圧縮機能（システム共通定義の rpc_datacomp オペランドに Y）を使用して TP1 インバウンドアダプタへトランザクショナル RPC 要求を送信した場合	KDJE58356-E 詳細情報：User data compression function is not supported.
	TP1 インバウンドアダプタへトランザクショナル RPC 要求をした際に，Message-driven Bean（サービス）の実行中に例外が発生した場合，または，Message-driven Bean（サービス）の実行結果に出力データを設定していない，または，J2EE サーバでシステム例外が発生した場合	KDJE58459-E
DCRPCER_TESTMODE(-366)	OpenTP1 でオンラインテスタを使用して TP1 インバウンドアダプタへ RPC 要求を送信した場合	KDJE58356-E 詳細情報：

dc_rpc_call 関数の戻り値	発生原因	TP1 インバウンドアダプタの出力メッセージ
DCRPCER_TESTMODE(-366)	OpenTP1 でオンラインテスタを使用して TP1 インバウンドアダプタへ RPC 要求を送信した場合	TP1/Online Tester is not supported.
DCRPCER_TRNCHK_EXTEND(-372)	OpenTP1 とのトランザクション連携でライトトランザクション（usrconf.properties の ejbserver.distributedtx.XATransaction.enabled プロパティが false）の場合	－

（凡例）　－：該当なし

注※

サービスグループ名が不正の場合，OpenTP1 は RPC 要求を再送します。そのため，このエラーメッセージは，OpenTP1 から再送された回数分だけ出力されます。

5 アプリケーションサーバでの JPA の利用

この章では，JPA の概要とアプリケーションサーバでの JPA の利用について説明します。

5.1　この章の構成

JPA とは，データベース関連処理の設計およびコーディングの簡略化を目的として，Java のオブジェクトとリレーショナルデータベースとのマッピング（O/R マッピング）に関して定められた仕様です。JPA を使用することで，ユーザはデータベースの持つ情報を Java のオブジェクト（エンティティ）として操作することができるため，効率良くシステムを構築できます。

この章では，JPA の概要と，アプリケーションサーバでの JPA の利用について説明します。この章の構成を次の表に示します。

表 5-1　この章の構成（アプリケーションサーバでの JPA の利用）

分類	タイトル	参照先
解説	JPA の特長	5.2
	アプリケーションサーバで使用できる JPA の機能	5.3
	EntityManager とは	5.4
	永続化コンテキストとは	5.5
実装	コンテナ管理の EntityManager を取得する方法	5.6
	アプリケーション管理の EntityManager を取得する方法	5.7
	persistence.xml での定義	5.8
	persistence.xml の配置	5.9
	JPA のインタフェース	5.10
注意事項	アプリケーション設定時の注意事項	5.11

注　「設定」および「運用」について，この機能固有の説明はありません。

5.2　JPA の特長

ここでは，JPA を使用したアプリケーションの特長について説明します。

5.2.1　JPA を使用したアプリケーションの利点

JPA を利用したアプリケーションを使用すると，次に示すことが実現できます。

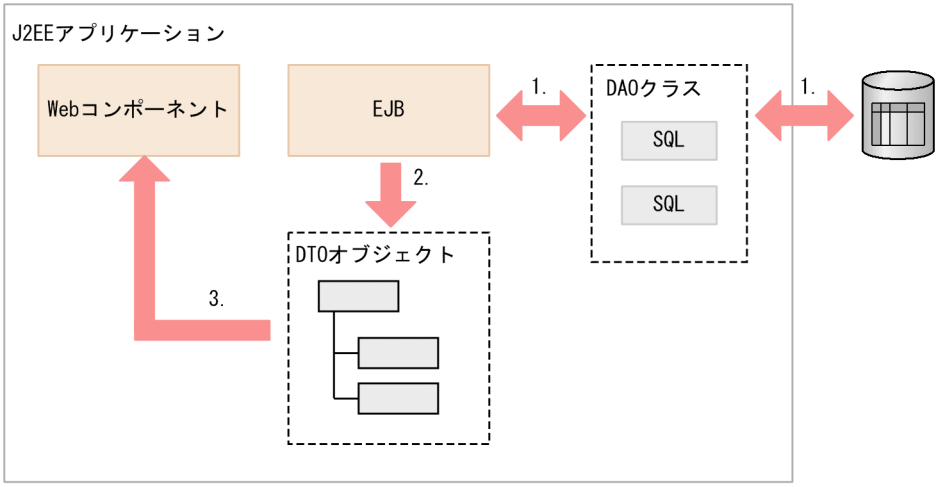
- O/R マッピングやデータベースアクセス処理の隠ぺいによって，ユーザプログラミングが容易になります。
- アノテーションを利用することで，定義ファイル作成コストを削減し，POJO（Plain Old Java Object）によるコーディングが実現します。
- デフォルト値の設定によって，ユーザのコーディング量が削減できます。
- CJPA プロバイダは JPA 仕様書に準じているため，ほかの JPA プロバイダで使用しているアプリケーションを移行できます。

ここでは，JPA を使用しない場合と JPA を使用する場合のデータアクセスモデルを比較して，JPA を使用したアプリケーションの利点を説明します。

(1)　JPA を使用しない場合のデータアクセスモデル

JPA を使用しない場合，J2EE アプリケーションからデータベースにアクセスするには，一般的に，次の図に示すようなデータアクセスモデルを使用してアプリケーションを作成します。

図 5-1　JPA を使用しない場合のデータベースアクセスモデル



上記の図について説明します。

図の場合のデータアクセスモデルでは，SQL をビジネスロジックから隠ぺいするために，テーブルに対応する DAO と呼ばれるクラスを作成します。DAO クラスの中に，JDBC インタフェースで SQL を発行する処理を作成します。図の流れについて説明します。

1. ビジネスロジックを記述してある EJB では，DAO を使用してデータベースからデータを読み出します。
2. 取得したデータを DTO と呼ばれるオブジェクトに格納します。

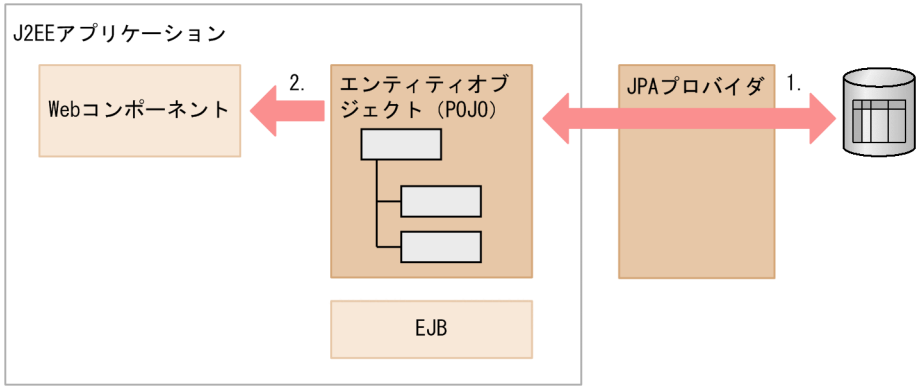
3. DTO オブジェクトを Web コンポーネントに返します。Web コンポーネントではデータベースから取得したデータを Web ページに出力します。

このようなデータアクセスモデルでは，データベースのデータモデルが大きく複雑になると，作成しなければならない DAO，SQL，DTO クラスの量も多くなります。DAO，SQL，および DTO の作成は単調な手作業のため，アプリケーション開発の生産性を低下させる原因となります。

(2)　JPA を使用した場合のデータアクセスモデル

JPA を使用した場合のデータアクセスモデルは次に示す図のようになります。

図 5-2　JPA を使用した場合のデータベースアクセスモデル



JPA を使用した場合，データベースのテーブルの行に対応したクラスを作成します。これを，エンティティクラスといいます。ビジネスロジックを記述してある EJB では，このエンティティクラスのオブジェクトを直接データベースに格納するかのように，処理を記述できます。

図について説明します。

1. JPA プロバイダと呼ばれる JPA のエンジンがデータベースに対して SQL を発行します。また，JPA プロバイダがエンティティオブジェクトとデータベースのテーブルの状態を自動的に同期を取ります。
2. エンティティオブジェクトは，取得したデータをそのまま Web コンポーネントに渡すことができます。

このように，JPA を使用すると DTO を作成する必要はありません。また，エンティティクラスは，Eclipse などの開発ツールを使用してデータベースのテーブルスキーマから自動的に生成することもできます。従来のデータアクセスモデルで生産性を低下させる原因となっていた DAO，SQL，DTO を作成する必要がないため，従来よりもアプリケーションの生産性を向上させることができます。

なお，エンティティクラスおよび JPA プロバイダについては，「5.2.2　エンティティクラスとは」および「5.2.3　JPA プロバイダとは」を参照してください。

5.2.2　エンティティクラスとは

JPA を使用する場合，アプリケーションではデータの入れ物となるクラスを作成します。これをエンティティクラスといいます。通常，エンティティクラスを作成する際には，エンティティクラスの一つのオブジェクトがデータベースのテーブルの一行に対応するように作成します。エンティティクラスは，普通の Java クラス（POJO）で作成します。特別なインタフェースをインプリメントする必要はありません。

エンティティクラスのフィールドの値をデータベーステーブルのどのカラムに格納するのかといったマッピングは，エンティティクラスのフィールドなどにアノテーションを使用して指定します。ただし，JPA には開発容易性を向上させるために CoC の考え方が取り入れられており，マッピングを明示的に指定しなく

てもデフォルトでマッピングが行われるルールがあります。例えば，フィールドのマッピングを明示的に指定していない場合，フィールド名から対応するカラムが推定されてマッピングされます。

5.2.3　JPA プロバイダとは

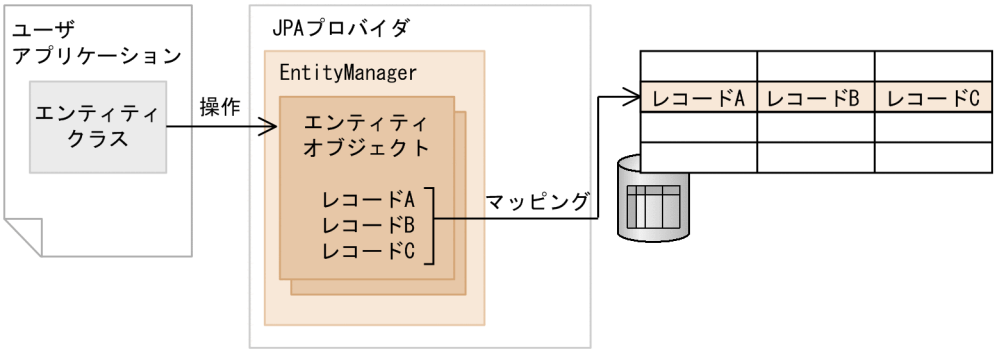
JPA プロバイダとは，次に示すマッピング機能，API，およびクエリ言語を提供する JPA 実装です。JPA プロバイダでは次の機能を提供しています。

- Java オブジェクトとデータベースのマッピング機能
- データベースへの処理をカプセル化した API
- JPA 仕様で共通に使用できるクエリ言語

JPA プロバイダを使用すると，データベースとのやり取りに関する処理を意識しないでアプリケーションを設計できるという利点があります。また，JPA プロバイダで利用できる JPQL というクエリ言語を使用することで，データベースに詳しくなくてもクエリを送信できるという利点もあります。

JPA プロバイダが提供するマッピング機能について次の図に示します。

図 5-3　JPA プロバイダが提供するマッピング機能の概要



図について説明します。

アプリケーション内にエンティティクラスを用意し，エンティティクラスからエンティティオブジェクトを生成します。ユーザは，エンティティオブジェクトの内容を変更することで，データベースの内容を変更します。これによって，データベースへの処理を意識しないで，データベースの内容を更新できます。

JPA プロバイダは，エンティティオブジェクトをデータベースのレコードとマッピングします。ユーザが実施した検索，挿入，削除，または更新の処理をデータベースに対して実行します。

生成されたエンティティオブジェクトは EntityManager の管理下に置かれます。エンティティオブジェクトのフィールドの値が変更されると，EntityManager は変更を自動的に検知して，変更をデータベースのテーブルに反映します。

EntityManager は次の処理をするときに呼び出されます。

- エンティティクラスのオブジェクトのデータをデータベースのテーブルに追加する。
- データベースにすでに格納されているデータを検索してエンティティクラスのオブジェクトとして取り出す。

EntityManager の詳細については，「5.4　EntityManager とは」を参照してください。

5.3　アプリケーションサーバで使用できる JPA の機能

ここでは，JPA を利用する場合に，アプリケーションサーバで使用できる JPA プロバイダ，コンポーネント，リソースアダプタなどについて説明します。

5.3.1　使用できる JPA プロバイダ

JPA プロバイダとは EntityManager の機能を提供するエンジンのことです。アプリケーションサーバで利用できる JPA プロバイダには，CJPA プロバイダとほかのベンダが提供する JPA プロバイダの 2 種類があります。それぞれの JPA プロバイダの利用について説明します。

(1)　CJPA プロバイダを利用する場合

アプリケーションサーバで提供している JPA プロバイダです。CJPA プロバイダでは, JPA 1.0 仕様に基づく機能のほかに，CJPA プロバイダ独自の機能も提供しています。CJPA プロバイダの概要，アプリケーション実装時の注意事項，および CJPA プロバイダの使用方法については，「6.　CJPA プロバイダ」を参照してください。

(2)　ほかのベンダの JPA プロバイダを利用する場合

ほかのベンダで提供している JPA プロバイダです。JPA 仕様で JPA プロバイダとアプリケーションサーバの間のインタフェースが明確にされているため, アプリケーションサーバではほかのベンダが提供している JPA 1.0 仕様に準拠した JPA プロバイダを利用することもできます。

アプリケーションサーバからほかのベンダの JPA プロバイダを利用する場合は，次の設定が必要になります。

- JAR ファイルの指定

  次のどちらかの方法で JPA プロバイダの実装が含まれる JAR ファイルを指定します。

  - 簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ下に JAR ファイルを指定します。JAR ファイルを指定するには，<param-name>タグに add.class.path, <param-value>タグに JAR ファイルを指定します。なお，簡易構築定義ファイルおよび指定するパラメタの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.6　簡易構築定義ファイル」を参照してください。
  - ライブラリとして J2EE アプリケーションに含めます。
- persistence.xml での定義

  persistence.xml の<provider>タグに，利用する JPA プロバイダが提供する javax.persistence.PersistenceProvider の実装クラス名を指定します。詳細については，「5.8.2(2) <provider>タグ」を参照してください。

また，アプリケーションサーバで提供する J2EE アプリケーション実行時間の監視機能を使用する場合は，JPA プロバイダのクラスおよびエンティティクラスを保護区リストに追加する必要があります。保護区リストへの追加方法については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「2.6　criticalList.cfg（保護区リストファイル)」を参照してください。

参考

JPA を使用したアプリケーションの実行では，アプリケーションサーバで提供する性能解析トレース機能を使用できます。

- JPA プロバイダとして CJPA プロバイダを使用する場合

  性能解析トレースはアプリケーションサーバと CJPA プロバイダの両方で出力できます。

- ほかのベンダの JPA プロバイダを使用する場合

  アプリケーションサーバで出力する性能解析トレースだけ使用できます。

なお，アプリケーションサーバでは，javax.persistence パッケージの EntityManagerFactory，EntityManager，EntityTransaction，および Query の API で性能解析トレースを出力します。また，JPA プロバイダでは，エンティティライフサイクルコールバックに関する性能解析トレースを出力します。

なお，性能解析トレースの概要については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「7.2.1　アプリケーションサーバの性能解析トレースの概要」を参照してください。性能解析トレースの出力ポイントについては，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「8.　性能解析トレースのトレース取得ポイントと PRF トレース取得レベル」を参照してください。

5.3.2　使用できるコンポーネント

アプリケーションサーバでは，EJB，Web アプリケーションで JPA を使用できます。また，Web アプリケーションからユーザスレッドを使用した場合も，JPA を使用できます。なお，次に示す環境またはライブラリでは JPA を使用できません。

- EJB クライアントアプリケーション環境
- J2EE アプリケーションクライアント環境
- コンテナ拡張ライブラリ

JPA を使用できるコンポーネントを次の表に示します。

表 5-2　JPA を使用できるコンポーネント

コンポーネント		JPA の使用
EJB	Stateless Session Bean（EJB3.0 以降）※1	○
	Stateful Session Bean（EJB3.0 以降）※1	○
	Stateless Session Bean（EJB3.0 より前）	×
	Stateful Session Bean（EJB3.0 より前）	×
	インターセプタ	○
	Message-driven Bean	×
	Entity Bean	×
Web アプリケーション	サーブレット，フィルタ，イベントリスナ（Servlet2.5 以降）	○
	JSP，JSP のタグハンドラ，JSP のイベントリスナ，JSP のタグライブラリイベントリスナ※2（Servlet2.5 以降）	○
	サーブレット，フィルタ，イベントリスナ（Servlet2.5 より前）	×
	JSP，JSP のタグハンドラ，JSP のイベントリスナ，JSP のタグライブラリイベントリスナ（Servlet2.5 より前）	×

（凡例）○：使用できる　×：使用できない

注※1　アプリケーションサーバでは EJB3.0 の ejb-jar.xml を使用した JPA の定義はできません。このため，@PersistenceUnit や@PersistenceContext などのアノテーションを使用して，永続化ユニットや永続化コンテキストの参照を定義してください。

注※2　アプリケーションサーバでは，JSP タグライブラリイベントリスナでのアノテーションの使用はできません。JSP タグライブラリイベントリスナで JPA の機能を使用する場合は，web.xml の<persistence-unit-ref>タグや<persistence-context-ref>タグを使用して永続化ユニットや永続化コンテキストの参照を定義してください。

! 注意事項

Servlet 2.5 以降の web.xml の metadata-complete 属性に true が設定されている場合，Web コンポーネントのアノテーションが読み込まれません。このため，アノテーションを使用して，永続化コンテキストまたは永続化ユニットの参照を定義することはできません。ただし，web.xml に参照を定義することはできます。

5.3.3　サポートするアプリケーションの形式

JPA を使用した J2EE アプリケーションは，次のどちらかの形式でアプリケーションサーバにデプロイします。

- アーカイブ形式の J2EE アプリケーション
- 展開ディレクトリ形式の J2EE アプリケーション

また，デプロイ済みの JPA を使用した J2EE アプリケーションを入れ替えることもできます。アーカイブ形式の場合はリデプロイ機能を，展開ディレクトリ形式の場合は，リロード機能を使用してください。

なお，リロード機能を使用する場合，O/R マッピングファイルは更新検知の対象になりません。ただし，リロードされるタイミングで，O/R マッピングファイルは再読み込みされます。次の表に，更新検知の対象およびリロード時の再読み込みについて示します。

表 5-3　リロード実行時の更新検知および再読み込みの対象

対象となるクラスおよびファイル	更新検知	再読み込み
エンティティクラス	○	○
マップドスーパークラス	○	○
埋め込みクラス	○	○
persistence.xml	○	○
O/R マッピングファイル （META-INF の下に orm.xml を配置する場合）	×	○
O/R マッピングファイル （persistence.xml の<mapping-file>タグに指定した場所に orm.xml を配置する場合）	○	○

（凡例）○：対象になる　×：対象にならない

アーカイブ形式の J2EE アプリケーション，および展開ディレクトリ形式の J2EE アプリケーションについては，「13.　J2EE アプリケーションの形式とデプロイ」を参照してください。

! 注意事項

展開ディレクトリ形式で JPA を使用したアプリケーションを利用する場合，アプリケーション動作中はクラスやライブラリ JAR を削除しないでください。削除した場合，アプリケーションサーバや JPA プロバイダが予期しない動作をするおそれがあります。

5.3.4　サポートするクラスローダ構成

JPA を使用した J2EE アプリケーションでは，次の表に示すクラスローダをサポートしています。

表 5-4　JPA を使用した J2EE アプリケーションでサポートするクラスローダ

クラスローダ	サポート
デフォルトのクラスローダ構成	○
ローカル呼び出し最適化をしているとき※のクラスローダ構成	○
下位互換用のクラスローダ構成	×

（凡例）○：サポートする　×：サポートしない

注

下位互換用のクラスローダ構成はベーシックモードだけの使用となるため，サポートしていません。

注※

簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に次のように指定しているときを示します。

<param-name>ejbserver.rmi.localinvocation.scope</param-name>

<param-value>all</param-value>

5.3.5　使用できるリソースアダプタ

JPA を使用した J2EE アプリケーションを実行する場合，アプリケーションサーバでは，コネクションファクトリインタフェースが javax.sql.DataSource であるリソースアダプタを使用できます。アプリケーションサーバで提供するリソースアダプタでは，DB Connector を使用できます。

また，リソースアダプタを使用するには，J2EE リソースアダプタとしてデプロイする必要があります。JPA を使用した J2EE アプリケーションの場合，リソースアダプタを J2EE アプリケーションに含めてデプロイすることはできません。

アプリケーションサーバで使用できるリソースアダプタを次の表に示します。

表 5-5　アプリケーションサーバで使用できるリソースアダプタ

コネクションファクトリインタフェース	リソースアダプタのデプロイ形式	JPA からの使用
javax.sql.DataSource	J2EE リソースアダプタとしてデプロイする	○
	J2EE アプリケーションに含めてデプロイする	×
javax.sql.DataSource 以外	－	×

（凡例）○：使用できる　×：使用できない　－：該当しない

なお，CJPA プロバイダを使用する場合に使用できるリソースアダプタについては，「6.2.3(3)　使用できる DB Connector」を参照してください。

5.4　EntityManager とは

EntityManager はデータベースに対してエンティティを登録したり，削除したりするためのインタフェースを持つオブジェクトです。ここでは，EntityManager の概要について説明します。

5.4.1　EntityManager で提供するメソッド

EntityManager ではメソッドを提供しています。代表的なメソッドを次に示します。

- persist メソッド（SQL の INSERT に相当）
  アプリケーションで new を実行したエンティティオブジェクトをデータベースに追加するためのメソッドです。
- find メソッド（SQL の SELECT に相当）
  エンティティオブジェクトをデータベースから検索するためのメソッドです。
- remove メソッド（SQL の DELETE に相当）
  エンティティオブジェクトをデータベースから削除するためのメソッドです。

EntityManager から find メソッドで検索したエンティティオブジェクトや，persist メソッドで EntityManager に渡したエンティティオブジェクトは，EntityManager の管理下に置かれます。EntityManager の管理下にあるエンティティオブジェクトについてフィールドの値が変更されると，EntityManager は変更を自動的に検知して，変更をデータベースのテーブルに反映します。

JPA では，EntityManager の管理下にあるエンティティオブジェクトのことを managed 状態のエンティティといいます。なお，デフォルトではトランザクションが決着したときに，エンティティは EntityManager の管理下から外れます。EntityManager の管理下から外れたエンティティのことを detached 状態のエンティティと呼びます。

5.4.2　EntityManager の種類

EntityManager の種類には，コンテナ管理の EntityManager およびアプリケーション管理の EntityManager の 2 種類があります。それぞれ説明します。

(1)　コンテナ管理の EntityManager

EntityManager の作成や破棄をコンテナに任せる方法です。コンテナ管理の EntityManager を使用すると，EntityManager の生成や破棄を意識しないでアプリケーションをコーディングできます。コンテナ管理の EntityManager を取得する方法と破棄する方法を説明します。

- 取得する方法
  コンテナ管理の EntityManager を取得するには，アプリケーションで DI または JNDI ルックアップを使用します。この方法で取得する EntityManager は，コンテナによって作成された EntityManager です。アプリケーションのコードでは，コンテナから取得した EntityManager をそのまま使うことができます。
  なお，アプリケーションからコンテナ管理の EntityManager を取得する方法については，「5.6　コンテナ管理の EntityManager を取得する方法」を参照してください。
- 破棄する方法
  アプリケーション内で，EntityManager の作成や破棄をコーディングする必要はありません。

(2)　アプリケーション管理の EntityManager

EntityManager の作成や破棄をアプリケーションが明示的に行う方法です。アプリケーションのコードによって明示的にライフサイクルが管理されます。アプリケーション管理の EntityManager を取得する方法と破棄する方法を説明します。

- **取得する方法**
  EntityManagerFactory を使用して，アプリケーションで EntityManager を作成します。EntityManagerFactory を取得するには，アプリケーションで DI または JNDI ルックアップを使用します。
  なお，アプリケーション管理の EntityManager を取得する方法については，「5.7　アプリケーション管理の EntityManager を取得する方法」を参照してください。
- **破棄する方法**
  EntityManager の close メソッドを呼んで，EntityManager を破棄します。

5.4.3　トランザクションの制御と EntityManager

トランザクションの制御方法によって，EntityManager には次の 2 種類があります。

- **JTA エンティティマネージャ**
  トランザクションが JTA によって制御される EntityManager です。
- **リソースローカルエンティティマネージャ**
  トランザクションが EntityTransaction API によって制御される EntityManager です。

EntityManager の種類とトランザクションの制御方法の関係について次の表に示します。

表 5-6　EntityManager の種類とトランザクションの制御方法の関係

EntityManager の種類	トランザクションの制御方法	
	JTA	リソースローカル
コンテナ管理の EntityManager	○※1	×
アプリケーション管理の EntityManager※2	○	○

（凡例）
○：制御できる
×：制御できない
JTA：JTA エンティティマネージャ
リソースローカル：リソースローカルエンティティマネージャ

注※1　トランザクションは必ず JTA によって制御されます。

注※2　トランザクションを JTA によって制御するか，またはアプリケーションが明示的に EntityTransaction という API を使用して制御するかを選択できます。EntityTransaction API を使用する場合は，そのトランザクションはリソースローカルトランザクションとなります。JTA トランザクションが存在した場合でも，JTA トランザクションとは無関係に制御されます。

JTA エンティティマネージャを使用するかリソースローカルエンティティマネージャを使用するかは，永続化ユニットの定義で指定します。永続化ユニットでの定義方法については，「5.8.1(2)　transaction-type 属性」を参照してください。

5.4.4　永続化ユニットとは

アプリケーションから JPA を使用する場合，次のような情報を定義する必要があります。

- アプリケーション内のエンティティクラスの情報
- エンティティクラスとデータベーステーブルとのマッピング情報
- JPA プロバイダがデータベースコネクションを取得するためのデータソースの情報

これらの情報を定義したものを永続化ユニットといいます。

永続化ユニットは, persistence.xml で定義します。Java EE 環境で JPA を使用する場合, persistence.xml はユーザがアプリケーションをパッケージングするときに，EJB-JAR，WAR，または EAR の中の決められた場所に配置します。

persistence.xml ファイルの中には，永続化ユニットの定義を複数含めることができます。また，一つのアプリケーションの中に複数の persistence.xml を含めることもできます。これによって, 一つのアプリケーションの中に, 複数の永続化ユニットを定義できます。アプリケーション内に複数の永続化ユニットが定義されている場合，アプリケーションがどの永続化ユニットを使用するかは，@PersistenceContext の unitName 属性で指定します。なお，アプリケーション内に一つだけ永続化ユニットが定義されている場合など，使用する永続化ユニットを一意に特定できる場合には，unitName 属性は省略できます。

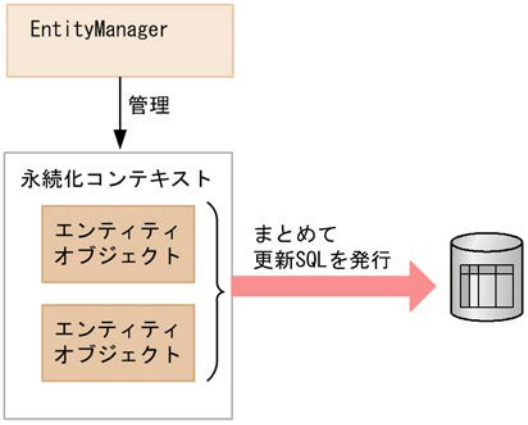
5.5　永続化コンテキストとは

EntityManager は更新するエンティティオブジェクトや検索されたエンティティオブジェクトをキャッシングします。永続化コンテキストは，EntityManager でキャッシングされたエンティティオブジェクトのキャッシュです。

5.5.1　EntityManager と永続化コンテキスト

EntityManager と永続化コンテキストの関係を次の図に示します。

図 5–4　EntityManager と永続化コンテキストの関係



EntityManager は，永続化コンテキストの中に managed 状態のエンティティオブジェクトを入れて管理しています。アプリケーションがエンティティオブジェクトを persist メソッドに渡したときや，managed 状態のエンティティオブジェクトについてフィールドの値を更新すると，永続化コンテキスト内のエンティティオブジェクトの状態が変更されます。EntityManager は，トランザクションがコミットされる直前に，永続化コンテキスト内のエンティティオブジェクトとデータベースのテーブルの状態を同期させます。永続化コンテキスト内のエンティティオブジェクトの状態をデータベースのテーブルに反映するために，このときにまとめて更新 SQL を発行します。これによって，データベースのロックの期間を短くできるので，同時実行性を向上したり，効率良くデータの更新をしたりできます。

(1)　EntityManager の種類

永続化コンテキストとトランザクションの関係によって，コンテナ管理の EntityManager とするか，アプリケーション管理の EntityManager にするかを決めます。

- **永続化コンテキストが JTA トランザクションとともに自動的に伝播する必要がある場合**

  コンテナ管理の EntityManager を使用します。コンテナ管理の EntityManager を使用すると，永続化コンテキストが JTA トランザクションとともに自動的に伝播されます。このため，一つの JTA トランザクションの中で複数のコンポーネントが呼ばれた場合に，同じ JTA トランザクションの中で使用される EntityManager は同じ永続化コンテキストに関連づけることができます。

  これによって，アプリケーションは EntityManager のリファレンスを，コンポーネントから別のコンポーネントに呼び出すときの引数に渡す必要がありません。

- **アプリケーションが JTA トランザクションとは独立した永続化コンテキストを使用する必要がある場合**

アプリケーション管理の EntityManager を使用します。アプリケーション管理の EntityManager を使用すると，同じ JTA トランザクションの中で別の EntityManager を使用したときでも，これらの EntityManager は永続化コンテキストを共有しないで，独立した永続化コンテキストを持ちます。

(2)　永続化コンテキストの種類

永続化コンテキストは生存区間によって次の 2 種類があります。

- **トランザクションスコープの永続化コンテキスト**
- **拡張された永続化コンテキスト**

コンテナ管理の EntityManager の場合，永続化コンテキストの種類を選択できます。永続化コンテキストの種類は，@PersistenceContext の type 属性で指定します。デフォルトはトランザクションスコープの永続化コンテキストとなります。

なお，アプリケーション管理の EntityManager の場合，常に拡張された永続化コンテキストとなります。永続化コンテキストの種類は選択できません。

5.5.2　コンテナ管理の EntityManager を使用する場合の永続化コンテキスト

コンテナ管理の EntityManager を使用する場合，永続化コンテキストのライフサイクルはコンテナによって管理され，JTA トランザクションとともに自動的に伝播されます。永続化コンテキストのライフサイクルの種類として，トランザクションスコープの永続化コンテキスト，または拡張された永続化コンテキストを選択できます。

それぞれの永続化コンテキストについて説明します。

(1)　トランザクションスコープの永続化コンテキスト

JPA 仕様では，トランザクションと同じライフサイクルの永続化コンテキストのことをトランザクションスコープの永続化コンテキストといいます。

EntityManager のライフサイクルは，デフォルトでトランザクションのライフサイクルと同じです。このため，EntityManager が持つ永続化コンテキストにキャッシュされた更新は，トランザクションがコミットするときにデータベースに反映されます。

(a)　永続化コンテキストのライフサイクル

トランザクションスコープの永続化コンテキストのライフサイクルを次に説明します。

- **永続化コンテキストの作成**

  トランザクションスコープの永続化コンテキストは，コンテナ管理の EntityManager が，JTA トランザクションの中で初めて呼び出されたときに作成されます。

  作成された永続化コンテキストは JTA トランザクションに関連づけられます。

  その後，同じ JTA トランザクションの中でコンテナ管理の EntityManager を使用した場合には，この永続化コンテキストが使用されます。

- **永続化コンテキストの破棄**

  JTA トランザクションがコミットまたはロールバックするときに，トランザクションスコープの永続化コンテキストは破棄されます。

コンテナ管理の EntityManager がトランザクションの外で呼び出された場合には，EntityManager のメソッド呼び出しが終了した時点で，データベースからロードされたすべてのエンティティは即座に detached 状態となります。

(2)　拡張された永続化コンテキスト

Java EE 環境では，Stateful Session Bean から EntityManager を使用する場合には，永続化コンテキストの生存期間を Stateful Session Bean の生存期間と同じにすることができます。この場合, 更新はトランザクションがコミットするたびにデータベースに反映されますが, 永続化コンテキスト内で管理されているエンティティオブジェクトは，複数のトランザクションにわたって managed 状態のまま保持されます。Stateful Session Bean と同じライフサイクルの永続化コンテキストのことを，JPA では拡張永続化コンテキストと呼びます。

拡張された永続化コンテキストは，Stateful Session Bean が作成されるのと同時に作成され，その Stateful Session Bean に関連づけされます。その後，Stateful Session Bean が破棄されるのと同時に破棄されます。

Stateful Session Bean が別の Stateful Session Bean を作成する場合で，作成する側と作成される側が共に拡張永続化コンテキストを使用するように定義されている場合には，作成する側の永続化コンテキストが，作成される側に引き継がれます。この引き継ぎは，Stateful Session Bean を作成する時点でトランザクションがアクティブであるかどうかにかかわらず行われます。Stateful Session Bean の作成時に永続化コンテキストの引き継ぎが行われた場合には，その永続化コンテキストを共有するすべての Stateful Session Bean が破棄されるときに，永続化コンテキストも破棄されます。

(3)　拡張された永続化コンテキストとトランザクション

拡張された永続化コンテキストは，EntityManager のインスタンスが作成されたときからクローズされるまで存在します。複数のトランザクションと EntityManager のトランザクション外の呼び出しに対応しています。

トランザクションとの関連を次に示します。

- EntityManager がトランザクションの範囲内で呼び出されるか, 永続化コンテキストがバインドしている stateful session bean がトランザクションのスコープで呼び出されると，EntityManager で管理されたエンティティがトランザクションに参加します。
- トランザクションが実行中かどうかにかかわらず，persist，remove，merge，refresh 操作が行われることがあります。この場合，EntityManager がトランザクションに参加して，トランザクションがコミットするときにデータベースに反映されます。
- トランザクションがコミットされたあとでも，エンティティオブジェクトへの参照は保持されます。エンティティオブジェクトは EntityManager に管理され，トランザクション間では，管理されているオブジェクト（managed 状態のエンティティ）として更新されます。

(4)　永続化コンテキストの伝播

コンテナ管理の EntityManager を使用する場合, 永続化コンテキストは JTA トランザクションによって伝播され，複数の EntityManager と関連づくことがあります。ただし，永続化コンテキストが伝播するのは, 同じアプリケーションサーバ内だけです。リモートのアプリケーションサーバに永続化コンテキストが伝播されることはありません。

次に，永続化コンテキストの伝播についてコンポーネントが呼び出されたときの状態ごとに説明します。

コンポーネントが呼び出されたときに，JTA トランザクションが存在しないか，または JTA トランザクションに永続化コンテキストが関連づいていない場合

永続化コンテキストは伝播されません。このコンポーネントから EntityManager が呼び出した時の動作は，次のようになります。

- トランザクションスコープの永続化コンテキストを使用する EntityManager が呼び出された場合には，新しい永続化コンテキストが作成されます。
- 拡張永続化コンテキストを使用する EntityManager が呼び出された場合には，呼び出された Stateful Session Bean に関連づいている拡張永続化コンテキストが使用されます。
- EntityManager が呼び出されたときに，JTA トランザクションが存在する場合には，JTA トランザクションに永続化コンテキストを関連づけます。

コンポーネントが呼び出されたときに，JTA トランザクションが伝播され，JTA トランザクションに永続化コンテキストが関連づいている場合

このコンポーネントから EntityManager を呼び出した時の動作は次のようになります。

- コンポーネントがすでに拡張永続化コンテキストを持っている Stateful Session Bean にもかかわらず，JTA トランザクションに別の永続化コンテキストが関連づいている場合は，コンテナによって EJBException がスローされます。
- トランザクションスコープの永続化コンテキストを使用する EntityManager が呼び出された場合，伝播してきた JTA トランザクションに関連づいている永続化コンテキストが使用されます。

5.5.3　アプリケーション管理の EntityManager を使用する場合の永続化コンテキスト

アプリケーション管理の EntityManager を使用する場合，アプリケーションが JPA プロバイダの EntityManagerFactory を直接呼び出して，EntityManager のライフサイクルおよび永続化コンテキストの作成や破棄を管理します。アプリケーション管理の永続化コンテキストのライフサイクルは，複数のトランザクションにわたったライフサイクルを管理できます。

(1)　EntityManager のライフサイクルの管理

アプリケーションでは EntityManager の close メソッドや isOpen メソッドを使用して，アプリケーション管理の EntityManager のライフサイクルを管理します。EntityManager の close メソッドが呼ばれると，EntityManager，EntityManager に関連づいた永続化コンテキストやそのほかのリソースが解放されます。close メソッドを呼び出したあとは，アプリケーションでは EntityManager の getTransaction メソッドと isOpen メソッド以外のメソッドを呼び出さないでください。呼び出した場合は，IllegalStateException がスローされます。トランザクションがアクティブのときに close メソッドを呼び出した場合は，トランザクションが決着するまで永続化コンテキストは保持されたままとなります。EntityManager の isOpen メソッドは，EntityManager がクローズされるまでは true を返し，クローズされたあとは false を返します。

(2)　永続化コンテキストのライフサイクル

アプリケーション管理の永続化コンテキストのライフサイクルを次に示します。

- **永続化コンテキストの作成**
  EntityManagerFactory の createEntityManager メソッドが呼ばれたときに作成されます。
- **永続化コンテキストの破棄**
  EntityManager の close メソッドが呼ばれたときに破棄されます。

アプリケーション管理の永続化コンテキストは，トランザクションとは独立した永続化コンテキストです。このため，JTA トランザクションとともに伝播されることはありません。

(3)　JTA エンティティマネージャを使用する場合の注意

アプリケーション管理の EntityManager で JTA エンティティマネージャを使用する場合, アプリケーションが EntityManager を JTA トランザクションのスコープの外で作成するときは，EntityManager の joinTransaction を呼ぶのはアプリケーションの責任になります。アプリケーションは EntityManager にトランザクションが開始されたことを通知する必要があるので，トランザクション開始後に EntityManager の joinTransaction メソッドを呼んでください。

5.6　コンテナ管理の EntityManager を取得する方法

コンテナ管理の EntityManager を J2EE アプリケーションから取得するには，次の 2 種類の方法があります。

- DI を使用してアプリケーションのフィールドや setter メソッドに EntityManager をインジェクトする方法
- アプリケーションから JNDI を使用して EntityManager をルックアップする方法

それぞれの方法について説明します。

5.6.1　アプリケーションに EntityManager をインジェクトする方法

アプリケーションのフィールドや setter メソッドに EntityManager をインジェクトする場合，さらに次の 2 種類の方法があります。

1. @PersistenceContext をインジェクト先のフィールドまたはメソッドに付加する方法
2. DD（web.xml）の<persistence-context-ref>タグで定義する方法

ただし，アプリケーションサーバでは EJB3.0 の ejb-jar.xml を使用した JPA の定義はできません。このため，EJB で JPA を使用する場合は，1.の方法で定義してください。

それぞれの方法について説明します。

(1)　@PersistenceContext を使用する方法

@PersistenceContext を使用して EntityManager をインジェクトする場合，インジェクト先のフィールドや setter メソッドに@PersistenceContext を付加します。@PersistenceContext に指定できる属性について，次に示します。

(a)　unitName 属性

unitName 属性には，persistence.xml で定義された永続化ユニットの名前を指定します。ただし，EJB-JAR や WAR，EAR 内に一つだけ永続化ユニットが定義されている場合など，使用する永続化ユニットが一意に特定できる場合には，unitName 属性を省略することができます。unitName 属性を省略した場合に，どの永続化ユニットが使用されるかについては，「5.11.2　永続化ユニット名の参照スコープ」を参照してください。

(b)　type 属性

type 属性には，永続化コンテキストのライフサイクルの種類を指定します。指定できる種類は，PersistenceContextType.TRANSACTION または PersistenceContextType.EXTENDED です。

- PersistenceContextType.TRANSACTION を指定した場合

  トランザクションスコープの永続化コンテキストが使用され，トランザクションの生存期間と永続化コンテキストの生存期間が同じになります。
- PersistenceContextType.EXTENDED を指定した場合

  拡張永続化コンテキストが使用され，Stateful Session Bean の生存期間と永続化コンテキストの生存期間が同じとなります。

  なお，type 属性に PersistenceContextType.EXTENDED を指定した@PersistenceContext は，Stateful Session Bean のフィールドまたはメソッドにだけ付けることができます。

type 属性を省略した場合のデフォルトは，PersistenceContextType.TRANSACTION です。

(c)　properties 属性

properties 属性には，永続化ユニットを設定するための JPA プロバイダ用のプロパティを指定できます。ここに指定したプロパティは，JPA プロバイダから EntityManager を取得するときに，JPA プロバイダに渡されます。

(d)　name 属性

インジェクションを使用する場合，通常 name 属性を指定する必要はありませんが，指定した場合には，name 属性に指定した名前で EntityManager が JNDI 名前空間（java:comp/env）に登録されます。@PersistenceContext を使用して EntityManager をインジェクトする例を次に示します。

```
@Stateless
public class InventoryManagerBean implements InventoryManager {
  @PersistenceContext(unitName=”myUnit”)
  private EntityManager em;
   ・・・
}
```

(2)　DD の<persistence-context-ref>を使用する方法

DD を使用して EntityManager をインジェクトする場合，DD の<persistence-context-ref>タグに次に示すタグを定義します。

(a)　<description>タグ

<description>タグには，定義する EntityManager リファレンスの説明をユーザが自由に記述できます。このタグを指定した場合でも，指定した内容がアプリケーションの動作に影響を与えることはありません。また，このタグは省略できます。

(b)　<persistence-context-ref-name>タグ

<persistence-context-ref-name>タグには，EntityManager が JNDI 名前空間に登録されるときの名前を指定します。指定する名前は, java:comp/env からの相対パスです。EntityManager の JNDI 登録名は必須ではありませんが，JPA 仕様では，java:comp/env/persistence 以下にすることが推奨されています。

(c)　<persistence-unit-name>タグ

<persistence-unit-name>タグには，persistence.xml に定義された永続化ユニットの名前を指定します。EJB-JAR や WAR，EAR 内に一つだけ永続化ユニットが定義されている場合など，使用する永続化ユニットを一意に特定できる場合には，<persistence-unit-name>タグは省略できます。<persistence-unit-name>タグを省略した場合に，どの永続化ユニットが使用されるかについては，「5.11.2　永続化ユニット名の参照スコープ」を参照してください。

(d)　<persistence-context-type>タグ

<persistence-context-type>タグには，永続化コンテキストのライフサイクルの種類を指定します。「Transaction」または「Extended」を指定します。

- 「Transaction」を指定した場合

  トランザクションスコープの永続化コンテキストが使用され，トランザクションの生存期間と永続化コンテキストの生存期間が同じになります。

- 「Extended」を指定した場合

拡張永続化コンテキストが使用され，Stateful Session Bean の生存期間と永続化コンテキストの生存期間が同じとなります。

なお，<persistence-context-type>タグに「Extended」を指定した<persistence-context-ref>タグは，Stateful Session Bean に対してだけ定義できます。

<persistence-context-type>タグを省略した場合のデフォルトは，「Transaction」になります。

(e)　<persistence-property>タグ

<persistence-property>タグには，永続化ユニットを設定するための JPA プロバイダ用のプロパティを指定できます。ここに指定したプロパティは，JPA プロバイダから EntityManager のファクトリを取得するときに，JPA プロバイダに渡されます。このタグは省略できます。

(f)　<injection-target>タグ

<injection-target>タグの<injection-target-class>タグにはインジェクト先のクラスを指定します。<injection-target>タグの<injection-target-name>タグには，インジェクト先のフィールド名または setter メソッド名を指定します。web.xml に<persistence-context-ref>タグを定義して EntityManager をインジェクトする例を次に示します。

```
...
<web-app>
  ...
  <servlet>
    <display-name>InventoryManagerServlet</display-name>
    <servlet-name>InventoryManagerServlet</servlet-name>
    <servlet-class>com.hitachi.InventoryManagerServlet</servlet-class>
  </servlet>
  ...
  <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</persistence-unit-name>
    <persistence-context-type>Transaction</persistence-context-type>
    <injection-target>
      <injection-target-class>
        com.hitachi.InventoryManagerServlet
      </injection-target-class>
      <injection-target-name>em</injection-target-name>
    </injection-target>
  </persistence-context-ref>
  ...
</web-app>
...
```

5.6.2　アプリケーションから EntityManager をルックアップする方法

JNDI を使用してアプリケーションから EntityManager をルックアップする場合，さらに次の 2 種類の方法があります。

1. EntityManager をルックアップするクラスに@PersistenceContext を付加して，EntityManager のリファレンスを定義する方法
2. DD（web.xml）で<persistence-context-ref>タグを定義して，EntityManager のリファレンスを定義する方法

ただし，アプリケーションサーバでは EJB3.0 の ejb-jar.xml を使用した JPA の定義はできません。このため，EJB で JPA を使用する場合は，1.の方法で定義してください。

それぞれの方法について説明します。

(1)　@PersistenceContext を使用する方法

@PersistenceContext を使用して EntityManager のリファレンスを定義する場合，ルックアップを実行するクラスに@PersistenceContext を付加します。

@PersistenceContext に指定できる属性について，次に説明します。

(a)　name 属性

name 属性には，アプリケーションのコードが EntityManager をルックアップするときのルックアップ名を指定します。指定するルックアップ名は，java:comp/env からの相対パスです。EntityManager のルックアップ名は，必須ではありませんが，JPA 仕様では，java:comp/env/persistence 以下にすることが推奨されています。

@PersistenceContext のそのほかの属性については，「5.6.1　アプリケーションに EntityManager をインジェクトする方法」と同じです。なお，拡張スコープの EntityManager のルックアップは，Stateful Session Bean だけできます。また，一つのクラスに複数の@PersistenceContext を付加する場合には，クラスに@PersistenceContexts を付加し，その value 属性として@PersistenceContext の配列を指定してください。次に，@PersistenceContext を使用して SessionContext から EntityManager をルックアップする例を示します。

```
@Stateless
@PersistenceContext(name="persistence/OrderEM")
public class MySessionBean implements MyInterface {
  @Resource SessionContext ctx;
  public void doSomething() {
     ・・・
    EntityManager em = (EntityManager)ctx.lookup("persistence/OrderEM");
     ・・・
  }
}
```

次に，@PersistenceContext を使用して InitialContext から EntityManager をルックアップする例を示します。

```
@Stateless
@PersistenceContext(name="persistence/InventoryAppMgr")
public class InventoryManagerBean implements InventoryManager {
  public void updateInventory(...) {
     ・・・
    Context initCtx = new InitialContext();
    EntityManager em = (EntityManager)
      initCtx.lookup("java:comp/env/persistence/InventoryAppMgr");
     ・・・
  }
}
```

(2)　DD の<persistence-context-ref>を使用する方法

DD を使用して EntityManager のリファレンスを定義する場合，DD の<persistence-context-ref>タグに次に示すタグを定義します。

(a)　<persistence-context-ref-name>タグ

<persistence-context-ref-name>タグには，アプリケーションのコードが EntityManager をルックアップするときのルックアップ名を指定します。指定するルックアップ名は，java:comp/env からの相対パスです。EntityManager のルックアップ名は，必須ではありませんが，JPA 仕様では java:comp/env/persistence 以下にすることが推奨されています。

<persistence-context-ref>タグのそのほかのタグについては，「5.6.1　アプリケーションに EntityManager をインジェクトする方法」と同じです。ただし，JNDI ルックアップで EntityManager を

取得する場合，<injection-target>は指定しません。なお，拡張スコープの EntityManager のルックアップは，Stateful Session Bean からできます。

次に，web.xml に<persistence-context-ref>を定義する例を示します。

```
...
<web-app>
  ...
  <servlet>
    <display-name>InventoryManagerServlet</display-name>
    <servlet-name>InventoryManagerServlet</servlet-name>
    <servlet-class>com.hitachi.InventoryManagerServlet</servlet-class>
  </servlet>
  ...
  <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</persistence-unit-name>
    <persistence-context-type>Transaction</persistence-context-type>
  </persistence-context-ref>
  ...
</web-app>
...
```

5.6.3　DD による@PersistenceContext 定義のオーバーライド

@PersistenceContext をアプリケーションに記載している場合に，DD で<persistence-context-ref>タグを定義していると，アノテーションで定義した内容は DD で定義した内容で上書きされます。この場合，アノテーションと DD との対応は，@PersistenceContext の name 属性と，DD の<persistence-context-ref>タグ下にある<persistence-context-ref-name>タグの対応で判断されます。なお，@PersistenceContext で name 属性が明示的に指定されていない場合でも，name 属性にはデフォルト値が存在するので注意が必要です。

次に@PersistenceContext に指定した属性が DD のタグでオーバーライドされるときの注意について説明します。

(1)　<persistence-unit-name>タグと unitName 属性

DD の<persistence-unit-name>タグは@PersistenceContext の unitName 属性をオーバーライドします。通常，永続化ユニット名を変更すると，アプリケーションは動作しなくなるので，DD およびアノテーションを定義する際には注意してください。

(2)　<persistence-context-type>タグと type 属性

DD の<persistence-context-type>タグは@PersistenceContext の type 属性をオーバーライドします。通常，永続化コンテキストのライフサイクルの種類を変更すると，アプリケーションは動作しなくなるので，DD およびアノテーションを定義する際には注意してください。

(3)　<persistence-property>タグと properties 属性

DD の<persistence-property>に指定されたプロパティは，@PersistenceContex の properties 属性に指定されたプロパティに追加されます。ただし，プロパティ名が同じ場合には，プロパティ値がオーバーライドされます。

(4)　<injection-target>タグ

インジェクションターゲットをオーバーライドすることはできません。なお, DD に<injection-target>タグを記述する場合には，@PersistenceContex が付加されたフィールドやメソッドを正確に指定してください。

5.7　アプリケーション管理の EntityManager を取得する方法

アプリケーション管理の EntityManager を使用する場合, アプリケーションは EntityManagerFactory を使用して EntityManager を作成します。アプリケーションが EntityManagerFactory を取得するには, 次の 2 種類の方法があります。

- DI を使用してアプリケーションのフィールドや setter メソッドに EntityManagerFactory をインジェクトする方法
- アプリケーションから JNDI を使用して EntityManagerFactory をルックアップする方法

それぞれの方法について説明します。

5.7.1　アプリケーションに EntityManagerFactory をインジェクトする方法

アプリケーションのフィールドや setter メソッドに EntityManagerFactory をインジェクトする場合，さらに次の 2 種類の方法があります。

1. @PersistenceUnit をインジェクト先のフィールドまたはメソッドに付加する方法
2. DD（web.xml）の<persistence-unit-ref>タグで定義する方法

ただし，アプリケーションサーバでは EJB3.0 の ejb-jar.xml を使用した JPA の定義はできません。このため，EJB で JPA を使用する場合は，1.の方法で定義してください。

それぞれの方法について説明します。

(1)　@PersistenceUnit を使用する方法

@PersistenceUnit を使用して EntityManagerFactory をインジェクトする場合，インジェクト先のフィールドや setter メソッドに@PersistenceUnit を付加します。@PersistenceUnit に指定できる属性について説明します。

(a)　unitName 属性

unitName 属性には，persistence.xml で定義された永続化ユニットの名前を指定します。ただし，EJB-JAR や WAR，EAR 内に一つだけ永続化ユニットが定義されている場合など，使用する永続化ユニットが一意に特定できる場合には，unitName 属性を省略できます。unitName 属性を省略した場合に，どの永続化ユニットが使用されるかについては，「5.11.2　永続化ユニット名の参照スコープ」を参照してください。

(b)　name 属性

インジェクションを使用する場合，通常 name 属性を指定する必要はありませんが，指定した場合には，name 属性に指定した名前で EntityManager が JNDI 名前空間（java:comp/env）に登録されます。次に，@PersistenceUnit を使用して EntityManagerFactory をインジェクトする例を示します。

```
@Stateless
public class InventoryManagerBean implements InventoryManager {
  @PersistenceUnit(unitName="myUnit")
  private EntityManagerFactory emf;
   ・・・
}
```

(2)　DD の<persistence-unit-ref>タグを使用する方法

DD を使用して EntityManagerFactory をインジェクトする場合，DD の<persistence-unit-ref>タグに次に示すタグを定義します。

(a)　<persistence-unit-ref-name>タグ

<persistence-unit-ref-name>タグには，EntityManagerFactory が JNDI 名前空間に登録されるときの名前を指定します。指定する名前は，java:comp/env からの相対パスです。

EntityManagerFactory の JNDI 登録名は必須ではありませんが，JPA 仕様では，java:comp/env/persistence 以下にすることが推奨されています。

(b)　<description>タグ

<description>タグには，定義する EntityManagerFactory リファレンスの説明をユーザが自由に記述することができます。このエレメントを指定した場合でも，指定した内容がアプリケーションの動作に影響を与えることはありません。また，このタグは省略できます。

(c)　<persistence-unit-name>タグ

<persistence-unit-name>タグには，persistence.xml に定義された永続化ユニットの名前を指定します。EJB-JAR や WAR，EAR 内に一つだけ永続化ユニットが定義されている場合など，使用する永続化ユニットを一意に特定できる場合には，<persistence-unit-name>タグは省略できます。<persistence-unit-name>タグを省略した場合に，どの永続化ユニットが使用されるかについては，「5.11.2　永続化ユニット名の参照スコープ」を参照してください。

(d)　<injection-target>タグ

<injection-target>タグの<injection-target-class>タグにはインジェクト先のクラスを指定します。<injection-target>タグの<injection-target-name>タグには，インジェクト先のフィールド名または setter メソッド名を指定します。次に，web.xml に<persistence-unit-ref>を定義して EntityManagerFactory をインジェクトする例を示します。

```
...
<web-app>
  ...
  <servlet>
    <display-name>InventoryManagerServlet</display-name>
    <servlet-name>InventoryManagerServlet</servlet-name>
    <servlet-class>com.hitachi.InventoryManagerServlet</servlet-class>
  </servlet>
  ...
  <persistence-unit-ref>
    <description>
      Persistence unit for the inventory management application.
    </description>
    <persistence-unit-ref-name>persistence/InventoryAppDB
    </persistence-unit-ref-name>
    <persistence-unit-name>InventoryManagement</persistence-unit-name>
    <injection-target>
      <injection-target-class>
        com.hitachi.InventoryManagerServlet
      </injection-target-class>
      <injection-target-name>emf</injection-target-name>
    </injection-target>
  </persistence-unit-ref>
  ...
</web-app>
...
```

5.7.2　アプリケーションから EntityManagerFactory をルックアップする方法

JNDI を使用してアプリケーションから EntityManagerFactory をルックアップする場合，さらに次の 2 種類の方法があります。

1. EntityManagerFactory をルックアップするクラスに@PersistenceUnit を付加して，EntityManagerFactory のリファレンスを定義する方法
2. DD（web.xml）で<persistence-unit-ref>タグを定義して，EntityManagerFactory のリファレンスを定義する方法

ただし，アプリケーションサーバでは EJB3.0 の ejb-jar.xml を使用した JPA の定義はできません。このため，EJB で JPA を使用する場合は，1.の方法で定義してください。

それぞれの方法について説明します。

(1)　@PersistenceUnit を使用する方法

@PersistenceUnit を使用して EntityManagerFactory のリファレンスを定義する場合，ルックアップを行うクラスに@PersistenceUnit を付加します。@PersistenceUnit に指定できる属性について説明します。

(a)　name 属性

name 属性には，アプリケーションのコードが EntityManagerFactory をルックアップするときのルックアップ名を指定します。指定するルックアップ名は，java:comp/env からの相対パスです。EntityManagerFactory のルックアップ名は，必須ではありませんが，JPA 仕様では java:comp/env/persistence 以下にすることが推奨されています。

@PersistenceUnit のそのほかの属性については，「5.7.1　アプリケーションに EntityManagerFactory をインジェクトする方法」と同じです。なお，一つのクラスに複数の@PersistenceUnit を付加する場合には，クラスに@PersistenceUnits を付加し，その value 属性として@PersistenceUnit の配列を指定してください。次に，@PersistenceContext を使用して SessionContext から EntityManagerFactory をルックアップする例を示します。

```
@Stateless
@PersistenceUnit(name="persistence/InventoryAppDB")
public class InventoryManagerBean implements InventoryManager {
  @Resource SessionContext ctx;
  public void updateInventory(...) {
    ・・・
    EntityManagerFactory emf = (EntityManagerFactory)
      ctx.lookup("persistence/InventoryAppDB");
    EntityManager em = emf.createEntityManager();
    ・・・
  }
}
```

次に，@PersistenceContext を使用して InitialContext から EntityManagerFactory をルックアップする例を示します。

```
@Stateless
@PersistenceUnit(name="persistence/InventoryAppDB")
public class InventoryManagerBean implements InventoryManager {
  public void updateInventory(...) {
    Context initCtx = new InitialContext();
    EntityManagerFactory emf = (EntityManagerFactory)
      initCtx.lookup("java:comp/env/persistence/InventoryAppDB");
    EntityManager em = emf.createEntityManager();
```

```
    . . .
  }
}
```

(2)　DD の<persistence-unit-ref>タグを使用する方法

DD を使用して EntityManagerFactory のリファレンスを定義する場合，DD の<persistence-unit-ref>タグを定義します。

(a)　<persistence-unit-ref-name>タグ

<persistence-unit-ref-name>タグには，アプリケーションのコードが EntityManagerFactory をルックアップするときのルックアップ名を指定します。指定するルックアップ名は，java:comp/env からの相対パスです。EntityManagerFactory のルックアップ名は，必須ではありませんが，JPA 仕様で java:comp/env/persistence 以下にすることが推奨されています。

<persistence-unit-ref>のそのほかのタグについては，「5.7.1　アプリケーションに EntityManagerFactory をインジェクトする方法」と同じです。ただし，JNDI ルックアップで EntityManagerFactory を取得する場合，<injection-target>タグは指定しません。次に，web.xml に<persistence-unit-ref>を定義する例を示します。

```
...
<web-app>
  ...
  <servlet>
    <display-name>InventoryManagerServlet</display-name>
    <servlet-name>InventoryManagerServlet</servlet-name>
    <servlet-class>com.hitachi.InventoryManagerServlet</servlet-class>
  </servlet>
  ...
  <persistence-unit-ref>
    <description>
     Persistence unit for the inventory management application.
    </description>
    <persistence-unit-ref-name>
      persistence/InventoryAppDB
    </persistence-unit-ref-name>
    <persistence-unit-name>InventoryManagement</persistence-unit-name>
  </persistence-unit-ref>
  ...
</web-app>
...
```

5.7.3　DD による@PersistenceUnit 定義のオーバーライド

@PersistenceUnit をアプリケーションに記載している場合に，DD で<persistence-unit-ref>タグを定義していると，アノテーションで定義した内容は DD で定義した内容で上書きされます。この場合，アノテーションと DD との対応は，@PersistenceUnit の name 属性と DD の<persistence-unit-ref>タグ下にある<persistence-unit-ref-name>タグの対応で判断されます。@PersistenceUnit で name 属性が明示的に指定されていない場合，name 属性にはデフォルト値が存在するので注意が必要です。

次に@PersistenceUnit に指定した属性が DD のタグでオーバーライドされるときの注意について説明します。

(1)　<persistence-unit-name>と unitName 属性

DD の<persistence-unit-name>は@PersistenceUnit の unitName 属性をオーバーライドします。通常，永続化ユニット名を変更すると，アプリケーションは動作しなくなるので，変更には注意が必要です。

(2)　<injection-target>タグ

DD でインジェクションターゲットをオーバーライドすることはできません。DD に<injection-target>を記述する場合には，@PersistenceContext が付加されたフィールドやメソッドを正確に指定しなければなりません。

5.8　persistence.xml での定義

永続化ユニットの情報は persistence.xml の<persistence-unit>タグを使用して定義します。ここでは，<persistence-unit>タグの属性および<persistence-unit>タグ下に指定するタグについて説明します。

なお，<persistence-unit>タグの属性および<persistence-unit>タグ下に指定するタグに指定した値の先頭および末尾に付加された空白文字や改行文字は無視されます。

persistence.xml のタグの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「6.2　persistence.xml」を参照してください。

5.8.1　<persistence-unit>タグに指定する属性

<persistence-unit>タグには name 属性と transaction-type 属性を指定します。

(1)　name 属性

定義する永続化ユニットの名前を指定します。ここで指定した名前は，アノテーションの場合，@PersistenceUnit または@PersistenceContext の unitName 属性から参照されます。また，DD の場合は，<persistence-context-ref>タグ下または<persistence-unit-ref>タグ下の<persistence-unit-name>タグから参照されます。

name 属性は省略できません。また，アプリケーションサーバで JPA を使用する場合，name 属性には空文字を指定できません。1 文字以上の文字列を指定してください。

(2)　transaction-type 属性

定義する永続化ユニットで，トランザクションを JTA によってコントロールするか，javax.persistence.EntityTransaction を使用してアプリケーションがコントロールするかを指定します。

- **トランザクションを JTA によってコントロールする場合**
  transaction-type 属性には「JTA」を指定します。「JTA」を指定した場合には，同時に<jta-data-source>タグも指定する必要があります。
- **EntityTransaction を使用してアプリケーションがトランザクションをコントロールする場合**
  transaction-type 属性には「RESOURCE_LOCAL」を指定します。「RESOURCE_LOCAL」を指定した場合には，<non-jta-data-source>タグも同時に指定する必要があります。

なお，transaction-type 属性を省略した場合，デフォルトは「JTA」となります。

5.8.2　<persistence-unit>タグ下に指定するタグ

<persistence-unit>タグ下には，次の表に示すタグを指定します。

表 5-7　<persistence-unit>タグ下に指定するタグ

指定するタグ	設定内容
<description>タグ	永続化ユニットの説明を記載します。
<provider>タグ	使用する JPA プロバイダを指定します。
<jta-data-source>タグ <non-jta-data-source>タグ	JPA プロバイダで使用する JTA データソースまたは非 JTA データソースを指定します。

指定するタグ	設定内容
<mapping-file>タグ	使用する O/R マッピングファイルを指定します。
<jar-file>タグ	entity クラス，embeddable クラス，mappedsuper クラスを含む JAR ファイル名を指定します。
<class>タグ	entity クラス，embeddable クラス，mappedsuper クラスを指定します。
<exclude-unlisted-classes>タグ	Perisitence クラスとして扱うかどうかを指定します。
<properties>タグ	JPA プロバイダ固有のプロパティを指定します。

それぞれのタグについて説明します。

(1)　<description>タグ

ユーザが永続化ユニットの説明を自由に記述できます。ここに指定した内容がアプリケーションの動作に影響を与えることはありません。なお，このタグは省略できます。

(2)　<provider>タグ

永続化ユニットで使用する JPA プロバイダを指定します。JPA プロバイダの javax.persistence.spi.PersistenceProvider インタフェースの実装クラス名を，パッケージ名を含めた完全修飾名で指定します。このタグは省略できます。

このタグを省略した場合，簡易構築定義ファイルで指定したデフォルトの JPA プロバイダが使用されます。また，このタグを省略した場合で，簡易構築定義ファイルでデフォルトの JPA プロバイダが指定されていないときには，JPA プロバイダとして CJPA プロバイダが使用されます。

なお，アプリケーションが特定の JPA プロバイダの機能や挙動に依存している場合は，<provider>タグを必ず指定してください。

ポイント

簡易構築定義ファイルでデフォルトの JPA プロバイダを指定するには，論理 J2EE サーバの<param-name>タグに ejbserver.jpa.defaultProviderClassName を指定して，<param-value>タグにデフォルトの JPA プロバイダクラス名を指定します。

なお，簡易構築定義ファイルで，論理 J2EE サーバの<param-name>タグに ejbserver.jpa.overrideProvider パラメタが指定されているときは，ejbserver.jpa.overrideProvider パラメタの<param-value>タグに指定されている JPA プロバイダクラス名が，<provider>タグや ejbserver.jpa.defaultProviderClassName パラメタに指定した値よりも優先して使用されます。

簡易構築定義ファイルに指定するパラメタについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.6.2　簡易構築定義ファイルの指定内容」を参照してください。

永続化ユニットで使用される JPA プロバイダを決定する優先順位を次の表に示します。

表 5-8　永続化ユニットで使用される JPA プロバイダを決定する優先順位

優先度	使用する JPA プロバイダ
1	簡易構築定義ファイルの ejbserver.jpa.overrideProvider プロパティに指定された値
2	persistence.xml の<provider>タグに指定された値
3	CJPA プロバイダ

優先度	使用する JPA プロバイダ
3	（簡易構築定義ファイルの ejbserver.jpa.defaultProviderClassName パラメタを使用して変更することもできる）

(3)　<jta-data-source>タグ，<non-jta-data-source>タグ

JPA プロバイダが使用する JTA データソースまたは非 JTA データソースを指定します。ここに指定する値は，JPA 仕様上は製品依存となっていますが，アプリケーションサーバでは次のようにデータソースの参照を定義します。

- Connector 1.0 に準拠したリソースアダプタを参照する場合
  「<リソースアダプタの表示名>」または「<リソースアダプタの別名>」を指定してください。
- Connector 1.5 に準拠したリソースアダプタを参照する場合
  「<リソースアダプタの表示名>!<コネクション定義識別子>」または「リソースアダプタの別名」を指定してください。

指定した値は「<リソースアダプタの表示名>」または「<リソースアダプタの表示名>!<コネクション定義識別子>」として解釈され，該当するリソースアダプタを検索します。該当するリソースアダプタが存在しない場合，指定した値は「<リソースアダプタの別名>」として解釈され，該当するリソースアダプタを検索します。

また，参照するリソースアダプタは，J2EE リソースアダプタとしてデプロイする必要があります（スタンドアロンモジュールとしてデプロイする方法）。リソースアダプタは，永続化ユニットを含むアプリケーションを開始するより前に，開始してください。

<jta-data-source>タグ，<non-jta-data-source>タグは省略できます。省略した場合には，簡易構築定義ファイルの ejbserver.jpa.defaultJtaDsName パラメタまたは ejbserver.jpa.defaultNonJtaDsName パラメタに指定した値が使用されます。ただし，これらのプロパティにはデフォルト値がありません。

ejbserver.jpa.overrideJtaDsName パラメタまたは ejbserver.jpa.overrideNonJtaDsName パラメタに値が指定されている場合は，<jta-data-source>タグ，<non-jta-data-source>タグに指定された値や，ejbserver.jpa.defaultJtaDsName パラメタ，ejbserver.jpa.defaultNonJtaDsName パラメタに指定された値よりも優先して使用されます。

なお，<jta-data-source>タグに指定するリソースアダプタのトランザクションサポートレベルには，LocalTransaction または XATransaction を指定する必要があります。また，<non-jta-data-source>に指定するリソースアダプタのトランザクションサポートレベルには，NoTransaction を指定する必要があります。

永続化ユニットで使用される JTA データソースおよび非 JTA データソースを決定するときの優先順位を次の表に示します。

表 5–9　永続化ユニットで使用される JTA データソースおよび非 JTA データソースを決定する優先順位

優先度	使用する JPA プロバイダ
1	簡易構築定義ファイルの ejbserver.jpa.overrideJtaDsName プロパティまたは ejbserver.jpa.overrideNonJtaDsName プロパティに指定された値
2	persistence.xml の<jta-data-source>または<non-jta-data-source>エレメントに指定された値
3	簡易構築定義ファイルの ejbserver.jpa.defaultJtaDsName プロパティまたは ejbserver.jpa.defaultNonJtaDsName プロパティに指定した値

! 注意事項

persistence.xml の<jta-data-source>タグまたは<non-jta-data-source>タグで，半角英数字およびアンダースコア(_)以外の文字が含まれているリソースアダプタの表示名を指定する場合，その文字をアンダースコア（_）に置き換えて指定してください。ただし，置き換えたあとの表示名の文字列がほかのリソースアダプタの表示名や別名と重複した場合，永続化ユニットは意図しないデータソースを使用して動作をしてしまう場合があるので注意してください。

(4)　<mapping-file>，<jar-file>，<class>，<exclude-unlisted-classes>タグ

永続化ユニットに含めるエンティティクラス，埋め込み可能クラス，マッピングされたスーパークラスを指定するには次の二つの方法があります。

1. O/R マッピングファイルや<class>タグを使用して明示的に指定する方法
2. 明示的には指定しないで JPA プロバイダによる自動検索を使用する方法

ここでは，1.の方法について説明します。

(a)　O/R マッピングファイルを使って指定する

永続化ユニットルートの META-INF の下，または永続化ユニットから<jar-file>タグで参照している別の JAR ファイル内の META-INF の下に，「orm.xml」という名前の XML ファイルが配置されている場合，<mapping-file>タグに指定しなくても自動的に O/R マッピングファイルとして扱われます。さらに，<mapping-file>タグにクラスパス上でロードできる XML ファイル名を指定した場合，その XML ファイルも O/R マッピングファイルとして扱われます。一つの永続化ユニットに複数の O/R マッピングファイルが含まれている場合，すべての O/R マッピングファイルからマッピング情報が読み込まれます。ただし，複数の O/R マッピングファイルの間で，重複してマッピングを定義している場合の動作は規定されていません。

(b)　パシステンスクラスを検索する JAR ファイルを指定する

<jar-file>タグには，パシステンスクラスや O/R マッピングファイルを含んだ JAR ファイルを指定できます。<jar-file>タグに指定された JAR ファイルからは，@Entity，@Embeddable，@MappedSuperclass が付加されたクラスが検索され，マッピング情報が自動的に取得されます。指定した JAR ファイル内に META-INF/orm.xml が存在する場合，orm.xml からもマッピング情報が取得されます。なお，<jar-file>タグと<mapping-file>タグを併用することもできます。

<jar-file>タグで指定できる JAR ファイルは，クラスパスに含まれている必要があります。指定できるのは次に示す JAR ファイルとなります。

- EAR のルートに置いた JAR ファイル
- EAR のライブラリディレクトリに置いた JAR ファイル
- EJB-JAR
- WAR の中の WEB-INF/lib に置いた JAR ファイル

ただし，EAR レベルまたは EJB-JAR レベルに定義された永続化ユニットの<jar-file>タグに，WAR の中の WEB-INF/lib に置いた JAR ファイルを指定することはできません。これは，WAR の中の WEB-INF/lib に置いた JAR ファイルは WEB アプリケーション用のクラスローダでロードされるので，WAR の中のコンポーネント以外からは参照できないためです。

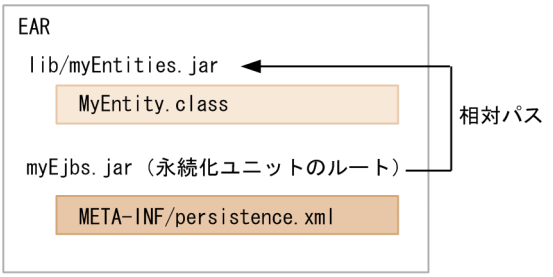
また，WAR レベルに定義された永続化ユニットの<jar-file>タグからは，同じ WAR の中に含まれた JAR ファイルだけが指定できます。これは，同じ WAR 以外の場所に配置された JAR ファイルに含まれるクラスは，WAR レベルに定義された永続化ユニットのデプロイメントが行われるよりも前にクラスローダに

よってロードされる可能性があるためです。このような場合，JPA プロバイダによるバイトコードの変換が正しく行われないおそれがあります。

<jar-file>タグには，永続化ユニットルートから JAR ファイルへの相対パスを指定します。指定例を次に示します。

例 1

次の図に示す場合の相対パスの指定について説明します。

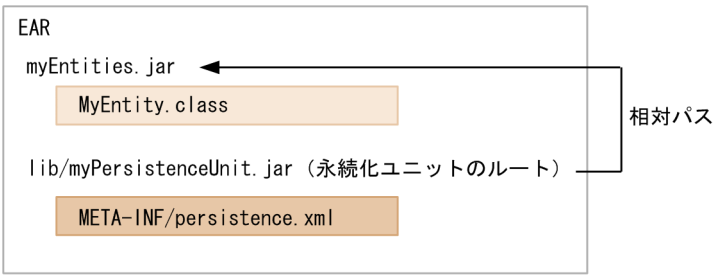


- EAR の lib にエンティティクラスを格納した myEntities.jar がある。
- EAR のルートに置いた EJB-JAR の META-INF/persistence.xml の<jar-file>タグで myEntities.jar を指定する。

この図の場合，永続化ユニットのルートは EJB-JAR そのものになるので，EJB-JAR から myEntities.jar への相対パスを指定します。相対パスは，「lib/myEntities.jar」です。このため，<jar-file>タグに「lib/myEntities.jar」と指定します。

例 2

次の図に示す場合の相対パスの指定について説明します。

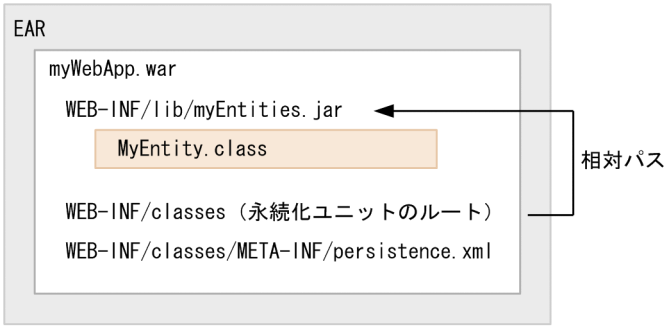


- EAR のルートにエンティティクラスを格納した myEntities.jar がある。
- EAR の lib/myPersistenceUnit.jar の META-INF/persistence.xml の<jar-file>タグで myEntities.jar を指定する。

この図の場合，永続化ユニットのルートは myPersistenceUnit.jar になるので，myPersistenceUnit.jar から myEntities.jar への相対パスを指定します。相対パスは，「../myEntities.jar」です。このため，<jar-file>タグには「../myEntities.jar」と指定します。

例 3

次の図に示す場合の相対パスの指定について説明します。



- WAR の WEB-INF/lib にエンティティクラスを格納した myEntities.jar がある。
- WAR の WEB-INF/classes/META-INF/persistence.xml の<jar-file>タグで myEntities.jar を指定する。

この図の場合，永続化ユニットルートは WAR の WEB-INF/classes になるので，WEB-INF/classes から myEntities.jar への相対パスを指定します。相対パスは「../lib/myEntities.jar」です。このため，<jar-file>タグには「../lib/myEntities.jar」と指定します。

(c)　パシステンスクラスのリストを明示的に指定する

<class>タグを使用すると，パシステンスクラスのリストを明示的に指定できます。マッピング情報は，指定されたクラスに付加されたアノテーションから取得されます。なお，<class>タグは，<mapping-file>タグや<jar-file>タグと併用することもできます。

(d)　アノテーションを付加したパシステンスクラスを永続化ユニットのルートに配置する

永続化ユニットのルートからは，@Entity，@Embeddable，@MappedSuperclass の付加されたパシステンスクラスが自動的に検索されます。マッピング情報はクラスに付加されたアノテーションから取得されます。永続化ユニットのルートに配置したアノテーションの付加されたクラスを永続化ユニットに加えたくない場合には，<exclude-unlisted-classes>タグを指定しておく必要があります。

(5)　<properties>タグ

JPA プロバイダのベンダ特有のプロパティを指定できます。JPA プロバイダが理解できないプロパティを指定した場合は，単に無視されます。なお，「javax.persistence」で始まるプロパティを<properties>に指定することはできません。

システムプロパティとして，プロパティ名がプレフィックス「ejbserver.jpa.emfprop.」で始まるプロパティを指定しておくと，プレフィックスを除去したプロパティが，永続化ユニットのプロパティに追加されます。

5.9　persistence.xml の配置

persistence.xml は EJB-JAR，WAR，EAR の中に配置します。persistence.xml は必ず META-INF の下に置く必要があります。META-INF の下に persistence.xml を持つパスを永続化ユニットのルートと呼びます。

persistence.xml を配置できる場所を次に示します。

- EJB-JAR の META-INF/persistence.xml
- WAR の WEB-INF/classes/META-INF/persistence.xml
- WAR の WEB-INF/lib の下に置いた jar ファイルの中の META-INF/persistence.xml
- EAR のルートディレクトリに置いた jar ファイルの中の META-INF/persistence.xml
- EAR のライブラリディレクトリに置いた jar ファイルの中の META-INF/persistence.xml

一つの persistence.xml には複数の永続化ユニットを定義できます。永続化ユニットには名前を付ける必要があります。ただし，一つの EJB-JAR，WAR，EAR 内には，重複した名前の永続化ユニットを複数定義することはできません。

EAR 内に定義した永続化ユニットで管理されるクラスは，アプリケーションのクラスローダでロードされ，アプリケーション内のすべてのコンポーネントから参照できます。

また，別々の EJB-JAR や WAR 内のコンポーネントから，同じエンティティクラスを参照する場合，仮に永続化ユニットが別々であったとしても参照されるクラスは同じ一意なクラスとなります。

5.10　JPA のインタフェース

JPA のインタフェースを紹介します。ここでは，javax.persistence.EntityManager インタフェースおよび javax.persistence.EntityManagerFactory インタフェースについて説明します。

5.10.1　javax.persistence.EntityManager インタフェース

ここでは，javax.persistence.EntityManager インタフェースについてインタフェース定義と注事事項を説明します。

(1)　インタフェース定義

```
package javax.persistence;

/**
* 永続化コンテキストを操作するためのインタフェース。
*
* EntityManagerインスタンスは永続化コンテキストと関連づけられています。
* 永続化コンテキストは，エンティティインスタンスのセットであり，
* 永続化されたエンティティごとに，エンティティのインスタンスは
* ユニークになっています。
* 永続化コンテキスト内では，エンティティのインスタンスと
* そのライフサイクルが管理されています。
* EntityManagerインタフェースは，永続化コンテキストを操作するための
* メソッドを定義しており，永続化されたエンティティインスタンスの作成・削除や，
* プライマリキーによるエンティティの検索，エンティティのクエリの実行などに
* 使用されます。
*
* EntityManagerによって管理できるエンティティのセットは，永続化ユニットで
* 定義します。
* 永続化ユニットは，アプリケーションによって使用されるエンティティクラスの
* グループを定義し，エンティティクラスとデータベースとのマッピングも
* 定義します。
*/
public interface EntityManager {

  /**
  * インスタンスをマネージド状態にして，永続化します。
  * @param entity 永続化するエンティティのインスタンス
  * @throws EntityExistsException エンティティがすでに存在した場合
  * （persistメソッドが呼ばれた時にEntityExistsExceptionがスローされるか，
  * flushまたはコミット時にEntityExistsExceptionまたは
  * そのほかのPersistenceExceptionがスローされます）
  * @throws IllegalArgumentException エンティティではない場合
  * @throws TransactionRequiredException PersistenceContextType.TRANSACTIONが指定された
  * コンテナ管理のエンティティマネージャが，トランザクションが存在しない時に
  * 呼ばれた場合
  */
  public void persist(Object entity);

  /**
  * エンティティの状態を現在の永続化コンテキストにマージします。
  * @param entity エンティティ
  * @return 状態が永続化コンテキストにマージされたインスタンス
  * @throws IllegalArgumentExceptionインスタンスがエンティティ
  * ではないか，removed状態のエンティティである場合
  * @throws TransactionRequiredException PersistenceContextType.TRANSACTION
  * を指定したコンテナ管理のエンティティマネージャが，トランザクションが
  * 存在しない時に呼ばれた場合
  */
  public <T> T merge(T entity);

  /**
  * エンティティのインスタンスを削除します。
  * @param entity エンティティ
  * @throws IllegalArgumentExceptionインスタンスがエンティティ
  * ではないか，detached状態のエンティティである場合
  * @throws TransactionRequiredException PersistenceContextType.TRANSACTION
```

```
* を指定したコンテナ管理のエンティティマネージャが，トランザクションが
* 存在しない時に呼ばれた場合
*/
public void remove(Object entity);

/**
* プライマリキーを検索します。
* @param entityClass エンティティクラス
* @param primaryKey プライマリキー
* @return 検索したエンティティのインスタンス
* エンティティが存在しない場合はnull
* @throws IllegalArgumentException entityClass引数がエンティティの型
* でない場合，またはprimaryKey引数がそのエンティティのプライマリキー
* として有効な型でない場合
*/
public <T> T find(Class<T> entityClass, Object primaryKey);

/**
* 状態が遅延フェッチされるインスタンスを取得します。
* リクエストしたエンティティがデータベースに存在しない場合，
* インスタンスの状態に最初にアクセスしたときに，
* EntityNotFoundExceptionがスローされます。
* （getReferenceが呼ばれた時に，JPAプロバイダが
* EntityNotFoundExceptionをスローすることも許されています）
* アプリケーションは，エンティティマネージャがオープンしている間に
* インスタンスにアクセスしない場合，デタッチ時にインスタンスの状態に
* アクセスできることを期待しないでください。
* @param entityClass エンティティクラス
* @param primaryKey プライマリキー
* @return 検索したエンティティのインスタンス。
* @throws IllegalArgumentException entityClass引数がエンティティの型
* でない場合，またはprimaryKey引数がそのエンティティのプライマリキー
* として有効な型でない場合
* @throws EntityNotFoundException エンティティの状態に
* アクセスできない場合
*/
public <T> T getReference(Class<T> entityClass, Object primaryKey);

/**
* 永続化コンテキストの状態をデータベースにフラッシュさせます。
* @throws TransactionRequiredException トランザクションが存在しない場合
* @throws PersistenceException フラッシュに失敗した場合
*/
public void flush();

/**
* 永続化コンテキストに含まれるすべてのオブジェクトに適用される
* フラッシュモードを設定します。
* @param flushMode フラッシュモード
*/
public void setFlushMode(FlushModeType flushMode);

/**
* 永続化コンテキストに含まれるすべてのオブジェクトに適用される
* フラッシュモードを取得します。
* @return flushMode フラッシュモード
*/
public FlushModeType getFlushMode();

/**
* 永続化コンテキストに含まれるエンティティオブジェクトのロックモードを
* セットします。
* @param entity エンティティ
* @param lockMode ロックモード
* @throws PersistenceException サポートされていないロック呼び出しが
* 行われた場合
* @throws IllegalArgumentException インスタンスがエンティティ
* でない場合，またはデタッチされたエンティティである場合
* @throws TransactionRequiredException トランザクションが存在しない場合
*/
public void lock(Object entity, LockModeType lockMode);

/**
* インスタンスの状態をデータベースの状態にリフレッシュします。
```

```
* インスタンスの状態が変更されている場合，データベースの状態で
* 上書きされます。
* @param entity エンティティ
* @throws IllegalArgumentException エンティティでないか，
* managed状態でない場合
* @throws TransactionRequiredException PersistenceContextType.TRANSACTION
* が指定された
* コンテナ管理のエンティティマネージャが，トランザクションが存在しない時に
* 呼ばれた場合
* @throws EntityNotFoundException エンティティがすでにデータベースに
* 存在しない場合
*/
public void refresh(Object entity);

/**
* 永続化コンテキストをクリアし，すべてのmanaged状態のエンティティを
* デタッチします。
* エンティティにデータベースにフラッシュされていない変更がある場合，
* 永続化されません。
*/
public void clear();

/**
* インスタンスが現在の永続化コンテキストに含まれているかをチェックします。
* @param entity エンティティ
* @return 含まれている場合true
* @throws IllegalArgumentException エンティティではない場合
*/
public boolean contains(Object entity);

/**
* JPQL(Java Persistence Query language)の文を実行するための
* Queryインスタンスを作成します。
* @param qlString Java Persistence Queryの文
* @return 新しいQueryのインスタンス
* @throws IllegalArgumentException クエリ文が有効でない場合
*/
public Query createQuery(String qlString);

/**
* 名前付きクエリ（JPQLまたはネイティブSQL）を実行するための
* Queryインスタンスを作成します。
* @param name メタデータで定義されたクエリの名前
* @return 新しいQueryのインスタンス
* @throws IllegalArgumentException 指定された名前のクエリが
* 定義されていない場合
*/
public Query createNamedQuery(String name);

/**
* ネイティブSQL文（update文，delete文）を実行するための
* Queryインスタンスを作成します。
* @param sqlString ネイティブSQL文
* @return 新しいQueryのインスタンス
*/
public Query createNativeQuery(String sqlString);

/**
* ネイティブSQLのクエリを実行するためのQueryインスタンスを作成します。
* @param sqlString ネイティブSQL文
* @param resultClass 返り値となるインスタンスのクラス
* @return 新しいQueryのインスタンス
*/
public Query createNativeQuery(String sqlString, Class result-
Class);

/**
* ネイティブSQLのクエリを実行するためのQueryインスタンスを作成します。
* @param sqlString ネイティブSQL文
* @param resultSetMapping 結果セットマッピングの名前
* @return 新しいQueryのインスタンス
*/
public Query createNativeQuery(String sqlString, String result-
SetMapping);
```

```
    /**
    * EntityManagerにJTAトランザクションがアクティブになったことを通知します。
    * このメソッドは，トランザクションのスコープの外で作成された
    * アプリケーション管理のJTAエンティティマネージャを
    * 現在のJTAトランザクションに関連づけるために，
    * アプリケーションによって呼び出されます。
    * @throws TransactionRequiredException トランザクションが存在しない場合
    */
    public void joinTransaction();

    /**
    * 下位にあるプロバイダのオブジェクトが存在する場合は返します。
    * このメソッドの返り値は実装依存ですが，
    * Component Containerでコンテナ管理のエンティティマネージャを
    * 使用する場合は，JPAプロバイダのEntityManagerオブジェクトを返します。
    * @return JPAプロバイダのEntityManagerオブジェクト
    * /
    public Object getDelegate();

    /**
    * アプリケーション管理のEntityManagerをクローズします。
    * closeメソッドが呼び出されたあとは，EntityManagerインスタンスと
    * EntityManagerから取得したQueryオブジェクトの，
    * getTransactionとisOpen（falseを返します）以外のすべてのメソッドは，
    * IllegalStateExceptionをスローします。
    * EntityManagerがアクティブなトランザクションに関連づいている時に
    * このメソッドが呼ばれた場合，トランザクションが決着するまで
    * 永続化コンテキストは存続します。
    * @throws IllegalStateException コンテナ管理のEntityManagerである場合
    */
    public void close();

    /**
    * EntityManagerがオープンされているかどうかを返す。
    * @return EntityManagerがクローズされるまではtrueを返す。
    */
    public boolean isOpen();

    /**
    * リソースレベルのトランザクションオブジェクトを返します。
    * EntityTransactionインスタンスは，複数のトランザクションを，
    * シリアルに開始，コミットするために使用されます。
    * @return EntityTransactionのインスタンス
    * @throws IllegalStateException JTAエンティティマネージャで
    * このメソッドが呼び出された場合
    */
    public EntityTransaction getTransaction();

}
```

(2)　注意事項

- トランザクションスコープの永続化コンテキストを使用している場合，persist，merge，remove，refresh メソッドは，トランザクションコンテキスト内で呼び出す必要があります。トランザクションコンテキストが存在しない場合，javax.persistence.TransactionRequiredException がスローされます。
- find，getReference メソッドは，トランザクションコンテキスト外で呼び出すこともできます。トランザクションスコープの永続化コンテキストを使用している場合，結果のエンティティはデタッチド状態になります。拡張永続化コンテキストを使用している場合，結果のエンティティは managed 状態になります。
- EntityManager から取得した Query と EntityTransaction オブジェクトは，EntityManager がオープンされている間，使用できます。
- createQuery メソッドの引数が有効な JPQL（Java Persistence Query Language）でない場合，IllegalArgumentException がスローされるか，クエリの実行が失敗します。ネイティブクエリが正しくない場合や，結果セットの定義がクエリの結果と互換性がない場合には，クエリが実行される時に

PersistenceException がスローされ，クエリの実行が失敗します。PersistenceException は，可能な場合はデータベースの例外をラップしたものになっています。

- EntityManager インタフェースのメソッドでランタイム例外がスローされる場合，トランザクションはロールバックします。
- close，isOpen，joinTransaction，getTransaction はアプリケーション管理の EntityManager を管理するためのメソッドです。
- EJB 仕様では，Stateless Session Bean で，EntityManager のメソッドを呼び出せるのは次のメソッドです。
  - ビジネスインタフェースまたはコンポーネントインタフェースのビジネスメソッド
  - ビジネスメソッド
  - インターセプタメソッド
  - タイムアウトコールバックメソッド

  コンストラクタや，DI のセッターメソッド（setSessionContext メソッドを含む），ライフサイクルコールバックメソッド（PostConstruct や PreDestroy）では，EntityManager のメソッドを呼び出せません。許可されていない場所で EntityManager のメソッドを呼び出した場合，KDJE56538-E メッセージが出力され，java.lang.IllegalStateException がスローされます。
- EJB 仕様では，Stateful Session Bean で EntityManager のメソッドを呼び出せるのは，次のメソッドです。
  - ライフサイクルコールバックメソッド（PostConstruct や PreDestroy）
  - ビジネスインタフェースまたはコンポーネントインタフェースのビジネスメソッド
  - ビジネスメソッド
  - インターセプタメソッド
  - SessionSynchronization の afterBegin および beforeCompletion メソッド

  コンストラクタや，DI のセッターメソッド，SessionSynchronization の afterCompletion メソッドでは，EntityManager のメソッドを呼び出せません。許可されていない場所で EntityManager のメソッドを呼び出した場合，KDJE56538-E メッセージが出力され，java.lang.IllegalStateException がスローされます。

なお，アプリケーションサーバで JPA を使用する場合は次の点にも注意してください。

- アプリケーションがインジェクト，JNDI でのルックアップ，および EntityManagerFactory を使用して取得した EntityManager は，JPA プロバイダが提供する EntityManager オブジェクトではなく，アプリケーションサーバが提供する EntityManager のプロキシクラスになっています。このプロキシクラスの getDelegate()を使用すると，JPA プロバイダが提供する EntityManager オブジェクトを取得できます。ただし，コンテナ管理の EntityManager を使用する場合，EntityManager のライフサイクルはコンテナが管理しているので，getDelegate()を使用して取得した EntityManager オブジェクトの close()をアプリケーションから呼ばないでください。
- トランザクションスコープの永続化コンテキストを使用する場合で，トランザクションが存在しないときに EntityManager の getDelegate()を呼び出すと，コンテナは返された EntityManager をクローズできません。この場合には，アプリケーションで EntityManager.close()を呼び出す必要があります。

5.10.2　javax.persistence.EntityManagerFactory インタフェース

ここでは，javax.persistence.EntityManagerFactory インタフェースについてインタフェース定義と注事事項を説明します。

(1)　インタフェース定義

```
package javax.persistence;

/**
* EntityManagerFactoryインタフェースは,
* アプリケーションがアプリケーション管理のEntityManagerを取得するために
* 使用します。
* アプリケーションがEntityManagerFactoryの使用を終了する時には,
* アプリケーションはEntityManagerFactoryをクローズする必要があります。
* EntityManagerFactoryがクローズされたあとは,
* EntityManagerFactoryから作成したすべてのEntityManagerはクローズされた
* ものとして扱われます。
*/
public interface EntityManagerFactory {
  /**
  * 新しいEntityManagerを作成します。
  * このメソッドは，呼び出されるたびに新しいEntityManagerのインスタンスを
  * 返します。
  * このメソッドが返すEntityManagerのisOpenメソッドはtrueを返します。
  * @return 新しいEntityManager
  */
  public EntityManager createEntityManager();

  /**
  * 指定したプロパティのマップを使用して新しいEntityManagerを作成します。
  * このメソッドは，呼び出されるたびに新しいEntityManagerのインスタンスを
  * 返します。
  * このメソッドが返すEntityManagerのisOpenメソッドはtrueを返します。
  * @param map EntityManagerのプロパティを格納したMap
  * @return 新しいEntityManager
  */
  public EntityManager createEntityManager(Map map);

  /**
  * ファクトリをクローズし，ファクトリが保持しているすべてのリソースを
  * 解放します。
  * ファクトリがクローズされたあとは，isOpen以外のメソッドを呼び出すと，
  * IllegalStateExceptionがスローされます。isOpenメソッドは，falseを
  * 返します。
  * EntityManagerFactoryがクローズされると，ファクトリから作成した
  * すべてのEntityManagerも，クローズされたものとして扱われます。
  */
  public void close();

  /**
  * ファクトリがオープンしているかどうかを返します。
  * @return ファクトリがクローズされるまではtrue
  */
  public boolean isOpen();

}
```

(2)　注意事項

- createEntityManager に渡すマップには,JPA プロバイダのベンダ特有プロパティを含めることができます。JPA プロバイダが認識できないプロパティは，単に無視されます。
- EJB 仕様では，Stateless Session Bean で EntityManagerFactory のメソッドを呼び出せるのは，次のメソッドです。
  - ライフサイクルコールバックメソッド（PostConstruct や PreDestroy）
  - ビジネスインタフェースまたはコンポーネントインタフェースのビジネスメソッド
  - ビジネスメソッド
  - インターセプタメソッド
  - タイムアウトコールバックメソッドです。

コンストラクタや，DI のセッターメソッド（setSessionContext メソッドを含む）では，EntityManagerFactory のメソッドを呼び出せません。

- EJB 仕様では，Stateful Session Bean で EntityManagerFactory のメソッドを呼び出せるのは，次のメソッドです。
  - ライフサイクルコールバックメソッド（PostConstruct や PreDestroy）
  - ビジネスインタフェースまたはコンポーネントインタフェースのビジネスメソッド
  - ビジネスメソッド
  - インターセプタメソッド
  - SessionSynchronization の afterBegin および beforeCompletion メソッドです。

  コンストラクタや，Dependency Injection のセッターメソッド，SessionSynchronization の afterCompletion メソッドでは，EntityManagerFactory のメソッドを呼び出せません。

5.11　アプリケーション設定時の注意事項

ここでは，アプリケーションサーバ上で動作させる JPA を利用したアプリケーションを設定するときの注意事項を説明します。

5.11.1　エンティティクラス配置時の注意

アプリケーションサーバでは，エンティティクラスは EAR，EJB-JAR，または WAR の中の JPA 仕様で定められた場所にパッケージングしてください。なお，エンティティクラスは，システムクラスパスに追加しないでください。

エンティティクラスは，アプリケーションのクラスローダまたは Web アプリケーションのクラスローダでローディングされるときに，Lazy フェッチなどを実現するために，JPA プロバイダによってクラスのバイトコード変換が行われることがあります。エンティティクラスがシステムクラスパスに含まれていると，エンティティクラスがシステムクラスローダでローディングされてしまうため，バイトコード変換が動作しません。このため，JPA プロバイダが正しく動作できなくなります。

5.11.2　永続化ユニット名の参照スコープ

アプリケーションに含まれる EJB やサーブレットなどのコンポーネントは，@PersistenceUnit や @PersistenceContext の unitName 属性，DD に定義した<persistence-context-ref>タグ下または <persistence-unit-ref>タグ下の<persistence-unit-name>タグで永続化ユニット名を指定して，使用する永続化ユニットを参照します。ただし，各コンポーネントから参照できる永続化ユニットのスコープは，次のとおりになります。

- EJB-JAR または WAR の中に定義した永続化ユニットは，EJB-JAR または WAR に含まれているコンポーネントから参照できます。
- EAR の中に定義した永続化ユニットは，EAR に含まれるすべてのコンポーネントから参照できます。

EAR で定義した永続化ユニットの名前と，EJB-JAR または WAR で定義した永続化ユニットの名前が重複している場合，EJB-JAR または WAR 内のコンポーネントからは，より狭い範囲で定義された永続化ユニットが優先されます。例えば，EAR と WAR 内に同じ名前の永続化ユニットが定義されている場合，WAR に含まれているコンポーネントからは，WAR で定義した永続化ユニットが優先して見えます。EAR の中にある永続化ユニットは見えません。

(1)　永続化ユニット名を省略した場合に使用される永続化ユニット

アプリケーションサーバで JPA を使用する場合，コンポーネントが参照する永続化ユニット名を省略すると，次に示すルールで使用する永続化ユニットが決定されます。

- コンポーネントが含まれる EJB-JAR または WAR の中に，永続化ユニットが一つだけ定義されている場合は，その永続化ユニットが使用されます。
- コンポーネントが含まれる EJB-JAR または WAR の中に，永続化ユニットが一つも定義されていなく，EAR 内に永続化ユニットが一つだけ定義されている場合は，EAR 内の永続化ユニットが使用されます。

なお，次の条件に該当する場合は，使用する永続化ユニットを一つに特定できないため，永続化ユニット名は省略できません。

- コンポーネントが含まれる EJB-JAR または WAR の中に，永続化ユニットが二つ以上定義されている場合

- コンポーネントが含まれる EJB-JAR または WAR の中に，永続化ユニットが一つも定義されていなく，EAR 内に永続化ユニットが二つ以上定義されている場合

(2)　「#」文法を使用した EAR レベル永続化ユニットの明示的な参照

EAR で定義した永続化ユニットの名前と，EJB-JAR または WAR で定義した永続化ユニットの名前が重複している場合，EJB-JAR または WAR 内のコンポーネントからは，より狭い範囲で定義された永続化ユニットが優先して見えます。ただし，永続化ユニットの参照名に「#」文法を使用することで，EAR に定義した永続化ユニットを明示的に参照することもできます。「#」文法を使用する場合，永続化ユニット名は次のとおり指定します。

「コンポーネントが含まれる EJB-JAR や WAR から永続化ユニットルートへの相対パス」#「永続化ユニット名」

例えば，EAR の ejbs/myEjbs.jar に含まれた EJB から，EAR の lib/persistenceUnitRoot.jar に含まれた永続化ユニット「myPersistenceUnit」を参照する場合，参照する永続化ユニット名は「../lib/persistenceUnitRoot.jar#myPersistenceUnit」になります。

5.11.3　アプリケーションのデプロイ時にチェックされる項目

JPA を利用したアプリケーションを J2EE サーバにデプロイするとき，次の項目がチェックされます。

- 永続化ユニット定義のチェック
- EntityManager や EntityManagerFactory のリファレンスのチェック

それぞれについて説明します。

(1)　永続化ユニット定義のチェック

永続化ユニット定義でチェックされる内容を説明します。

(a)　persistence.xml のバリデーション

アプリケーションに含まれる persistence.xml が persistence_1_0.xsd スキーマに従っているかを検証します。検証でエラーが発見された場合は，エラーメッセージ KDJE56526-E を出力してデプロイを中止します。

(b)　永続化ユニット名のチェック

永続化ユニット名が空文字でないことをチェックします。永続化ユニット名が空文字である場合，エラーメッセージ KDJE56505-E を出力してデプロイを中止します。

また，アプリケーションの EAR や，一つの EJB-JAR または WAR の中に，重複した名前の永続化ユニットが定義されていないかをチェックします。重複した名前の永続化ユニットが定義されている場合には，警告メッセージ KDJE56500-W を出力し，デプロイを継続します。なお，この場合は実際にデプロイされる永続化ユニットは一つだけです。

(c)　永続化ユニットが参照するデータソースの存在チェック

永続化ユニットのトランザクションタイプによってチェックされる内容が異なります。

- **永続化ユニットのトランザクションタイプが JTA の場合**
  チェックされる内容とチェック時にエラーが発生したときの動作を次の表に示します。

表 5-10　チェックされる内容とチェック時にエラーが発生したときの動作（トランザクションタイプが JTA の場合）

チェックされる内容	チェック時にエラーが発生したときの動作
永続化ユニットで使用する JTA データソース（persistence.xml の<jta-data-source>タグに指定されたもの，またはシステムプロパティで指定したデフォルト値）が指定されていること。	エラーメッセージ KDJE56527-E を出力して，デプロイを中止します。
JTA データソースを提供するリソースアダプタが存在すること。	エラーメッセージ KDJE56529-E を出力して，デプロイを中止します。
JTA データソースを提供するリソースアダプタが開始されていること。	
リソースアダプタのトランザクションサポートレベルが，LocalTransaction または XATrasnaction になっていること。	エラーメッセージ KDJE56531-E を出力して，デプロイを中止します。
リソースアダプタのコネクションファクトリインタフェースが「javax.sql.DataSource」であること。	エラーメッセージ KDJE56533-E を出力して，デプロイを中止します。

- 永続化ユニットのトランザクションタイプが RESOURCE_LOCAL の場合

  チェックされる内容とチェック時にエラーが発生したときの動作を次の表に示します。

  表 5-11　チェックされる内容とチェック時にエラーが発生したときの動作（トランザクションタイプが RESOURCE_LOCAL の場合）

チェックされる内容	チェック時にエラーが発生したときの動作
永続化ユニットで使用する非 JTA データソース（persistence.xml の<non-jta-data-source>タグに指定されたもの，またはシステムプロパティで指定したデフォルト値）が指定されていること。	エラーメッセージ KDJE56528-E を出力して，デプロイを中止します。
非 JTA データソースを提供するリソースアダプタが存在すること。	エラーメッセージ KDJE56530-E を出力して，デプロイを中止します。
非 JTA データソースを提供するリソースアダプタが開始されていること。	
リソースアダプタのトランザクションサポートレベルが，NoTransaction になっていること。	エラーメッセージ KDJE56532-E を出力して，デプロイを中止します。
リソースアダプタのコネクションファクトリインタフェースが「javax.sql.DataSource」であること。	エラーメッセージ KDJE56533-E を出力して，デプロイを中止します。

(d)　永続化ユニットに指定されたプロバイダクラスのチェック

永続化ユニットが使用する JPA プロバイダのクラス（persistence.xml の<provider>タグに指定されたクラス）を，アプリケーションからロードできることをチェックします。ロードに失敗した場合には，エラーメッセージ KDJE56503-E を出力して，デプロイを中止します。

(e)　永続化ユニットが参照する JAR ファイルの存在チェック

永続化ユニットが参照する JAR ファイル（persistence.xml の<jar-file>タグに指定された JAR ファイル）が存在するかをチェックします。JAR ファイルが存在しない場合には，エラーメッセージ KDJE56506-E を出力して，デプロイを中止します。

(f)　JPA プロバイダによる永続化ユニットの定義内容のチェック

JPA コンテナが解析した persistence.xml の内容から JPA プロバイダが永続化ユニットを生成します。永続化ユニットの定義に問題があり，JPA プロバイダがエラーを返した場合，エラーメッセージ（KDJE56539-E）を表示してデプロイメントを中止します。

(2)　EntityManager や EntityManagerFactory のリファレンスのチェック

EntityManager や EntityManagerFactory のリファレンスでチェックされる内容を説明します。

(a)　永続化ユニットの存在チェック

EJB や Web コンポーネントで定義した EntityManager や EntityManagerFactory のリファレンスで，指定した永続化ユニット名が実際に参照できる永続化ユニット名であるかをチェックします。また，永続化ユニット名が省略されている場合は，使用する永続化ユニットを特定できるかをチェックします。このチェックでエラーが発見された場合には，エラーメッセージ KDJE56501-E を出力して，デプロイを中止します。

(b)　コンテナ管理の EntityManager を使用する場合のトランザクションタイプのチェック

コンテナ管理の EntityManager を使用する場合は，永続化ユニットのトランザクションタイプが JTA になっていることをチェックします。このチェックでエラーが発生した場合は，エラーメッセージ KDJE56534-E を出力して，デプロイを中止します。

(c)　拡張永続化コンテキストが Stateful Session Bean 以外から使用されていないことをチェック

EntityManager のリファレンスでは，永続化コンテキストのタイプに EXTENDED が指定されている場合，そのリファレンスが定義されている場所が Stateful Session Bean であることをチェックします。Stateful Session Bean 以外の場所で定義されている場合には，エラーメッセージ KDJE56535-E を出力して，デプロイを中止します。

5.11.4　アプリケーションサーバで JPA を使用するときの注意事項

アプリケーションが取得できる EntityManager の型は，JPA プロバイダの EntityManager オブジェクトではなく，アプリケーションサーバが提供する EntityManager のプロキシクラスです。

EntityManager オブジェクトは，インジェクション，JNDI ルックアップ，または EntityManagerFactory で取得できます。

なお，インジェクションを使用して EntityManager を取得する場合には，EntityManager をインジェクトするフィールドまたはメソッド引数の型は，javax.persistence.EntityManager にしてください。

また，インジェクション，JNDI ルックアップ，または EntityManagerFactory で取得した EntityManager は，JPA プロバイダの EntityManager の実装クラスにキャストできません。

JPA プロバイダの EntityManager オブジェクトを取得する必要がある場合は，インジェクション，JNDI ルックアップ，または EntityManagerFactory で取得した EntityManager プロキシオブジェクトの getDelegate メソッドを使用してください。

5.11.5　JPA 機能を使用しないときの注意事項

JPA 機能では，アプリケーションに persistence.xml が含まれている場合，デフォルトの設定では，JPA 機能を使用するかどうかに関係なく persistence.xml の読み込みが実行されます。アプリケーションサーバで persistence.xml の解釈ができない場合，アプリケーションの開始に失敗します。

この状況を避けるため，アプリケーションに persistence.xml が含まれ，かつ JPA 機能を使用しない場合には，J2EE サーバの ejbserver.jpa.disable プロパティに「true」を設定してください。これによって，アプリケーションサーバでは persistence.xml の読み込みを実行しなくなります。

また，ejbserver.jpa.disable プロパティの設定値に関係なく，アプリケーションサーバで使用できる JPA のバージョンは，JPA1.0 となります。

6 CJPA プロバイダ

この章では，CJPA プロバイダの機能概要，アプリケーションの実装方法，および実行環境の設定について説明します。

6.1　この章の構成

CJPA プロバイダとは，JPA 仕様書に定められた API や内部処理を実装し，ライブラリ形式でユーザに提供するアプリケーションサーバが提供する JPA プロバイダです。

この章では，CJPA プロバイダの機能について説明します。この章の構成を次の表に示します。

表 6–1　この章の構成（CJPA プロバイダの機能）

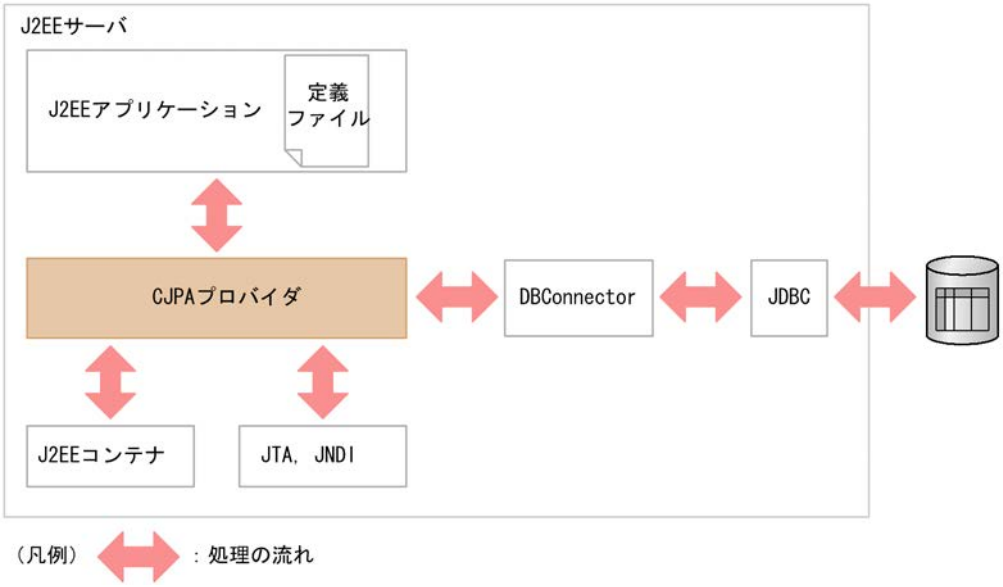
分類	タイトル	参照先
解説	CJPA プロバイダとは	6.2
	エンティティを使用したデータベースの更新	6.3
	EntityManager によるエンティティの操作	6.4
	データベースと Java オブジェクトとのマッピング情報の定義	6.5
	エンティティのリレーションシップ	6.6
	エンティティオブジェクトのキャッシュ機能	6.7
	プライマリキー値の自動採番	6.8
	クエリ言語によるデータベース操作	6.9
	楽観的ロック	6.10
	JPQL での悲観的ロック	6.11
実装	エンティティクラスの作成	6.12
	エンティティクラスの継承方法	6.13
	EntityManager および EntityManagerFactory の使用方法	6.14
	コールバックメソッドの指定方法	6.15
	クエリ言語を利用したデータベースの参照および更新方法	6.16
	JPQL の記述方法	6.17
	persistence.xml の定義	6.18
設定	実行環境での設定	6.19

注　「運用」について，この機能固有の説明はありません。

6.2　CJPA プロバイダとは

CJPA プロバイダとは，アプリケーションサーバが提供する JPA プロバイダです。CJPA プロバイダの位置づけを次の図に示します。

図 6-1　CJPA プロバイダの位置づけ



CJPA プロバイダは J2EE サーバの一機能です。JTA および JNDI を使用して，データベースを操作します。ここでは，CJPA プロバイダの機能概要について説明します。

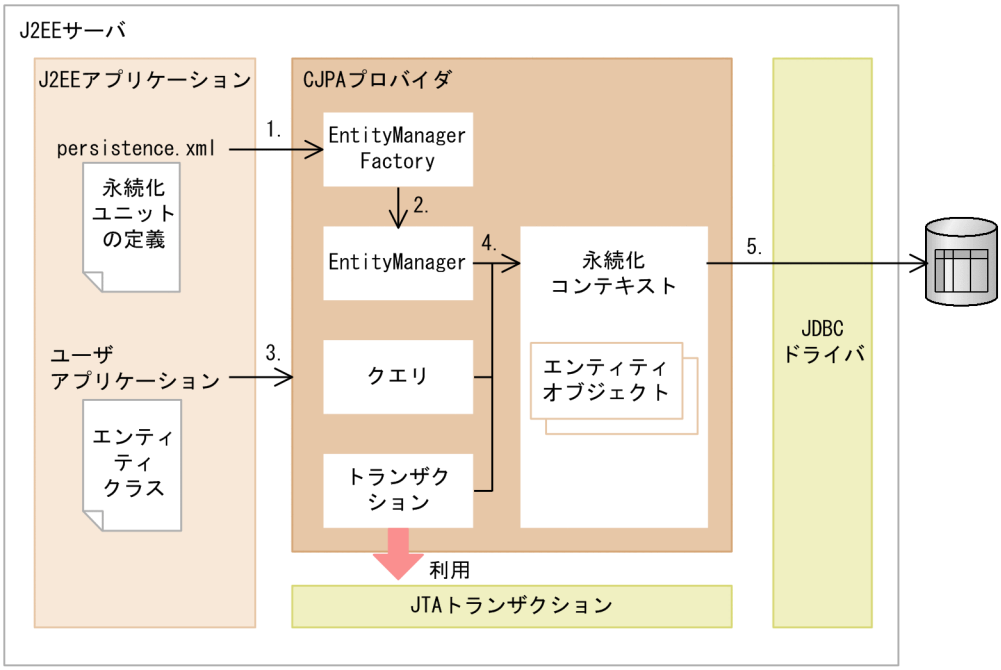
6.2.1　CJPA プロバイダでの処理

CJPA プロバイダでは，次の機能を提供しています。

- エンティティオブジェクトとデータベースのマッピング
- エンティティオブジェクトの管理
- JPQL によるデータの操作
- JDBC を通じてのデータベースへのアクセス

JPA を使用するには，ユーザが作成したアプリケーションで CJPA プロバイダが提供するインタフェースを操作します。CJPA プロバイダで実行する処理を次の図に示します。

図 6-2　CJPA プロバイダで実行される処理



図について説明します。

1. persistence.xml に定義されている永続化ユニットの情報から，CJPA プロバイダは EntityManagerFactory を生成します。
   永続化ユニットとは，マッピング対象となるクラス，O/R マッピングの情報，データソースなどを集めた論理的なグループです。永続化ユニットの詳細については，「5.4.4　永続化ユニットとは」を参照してください。
2. 生成された EntityManagerFactory から EntityManager を生成します。
3. ユーザアプリケーションから，EntityManager，クエリ，またはトランザクションを操作します。
4. EntityManager，クエリ，トランザクションから永続化コンテキストを通して，エンティティオブジェクトを操作します。
5. エンティティオブジェクトの変更を JDBC ドライバを介してデータベースに反映します。

なお，EntityManager には，次の 2 種類があります。

- コンテナ管理 EntityManager
  J2EE コンテナによって管理される EntityManager です。
- アプリケーション管理 EntityManager
  アプリケーションによって管理される EntityManager です。

EntityManager の種類の詳細については，「5.5.1　EntityManager と永続化コンテキスト」を参照してください。

また，トランザクションではトランザクションのコミットやロールバックなどを管理します。CJPA プロバイダでは，トランザクションとしてリソースローカルトランザクションを使用できます。JTA トランザクションと連携する場合，J2EE コンテナが提供する機能を使用します。

6.2.2　CJPA プロバイダが提供する機能

CJPA プロバイダでは次に示す機能を提供しています。

表 6-2　CJPA プロバイダが提供する機能

機能	機能の概要	標準/拡張※	参照先
エンティティを使用したデータベースの更新	ユーザのアプリケーションから生成したエンティティによって，データベースのデータを更新します。	標準	6.3 6.4
データベースと Java オブジェクトとのマッピング情報の定義	Java オブジェクトとデータベースとのマッピング情報は，アノテーションまたは O/R マッピングファイルで定義できます。	標準	6.5
エンティティのリレーションシップの設定	データベースのテーブル間の関連をエンティティで設定します。	標準	6.6
エンティティオブジェクトのキャッシュ	EntityManager のエンティティオブジェクトの内容をメモリに保持する機能です。同じ内容のエンティティオブジェクトが呼ばれた場合，メモリのエンティティオブジェクトを使用します。	拡張	6.7
プライマリキーの自動採番	エンティティオブジェクトを使用してデータベースにデータを挿入するときに，CJPA プロバイダはプライマリキーを自動的に格納します。ユーザがプライマリキーを指定しなくても，一意の値が格納されます。	標準	6.8
クエリ言語によるデータベース操作	クエリ言語を使用してデータベースを操作できます。CJPA プロバイダでは，クエリ言語として JPQL または SQL を使用できます。	標準	6.9
楽観的ロック	データベースの排他方法の一つです。データベースのデータがほかのトランザクションから更新されていないかを確認してからデータを更新します。楽観的ロックではデータをロックしないため，デッドロックは発生しません。	標準	6.10
JPQL での悲観的ロック	データベースの排他方法の一つです。ほかのトランザクションから更新されないように，レコードを占有ロックします。悲観的ロックは JPQL を使用した場合だけ実行されます。	拡張	6.11

注※　JPA1.0 仕様に基づいた機能かどうかを示します。

標準：JPA1.0 仕様に基づいた機能であることを示します。

拡張：CJPA プロバイダ独自の機能であることを示します。

これらの機能の詳細については，表中の「参照先」に示す節を参照してください。

また，CJPA プロバイダでは，PRF トレースを取得できます。PRF トレースの取得ポイントについては，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「8.　性能解析トレースのトレース取得ポイントと PRF トレース取得レベル」を参照してください。

6.2.3　CJPA プロバイダを使用するための前提条件

ここでは，CJPA プロバイダを使用するための前提条件について説明します。

(1)　使用できるコンポーネント

CJPA プロバイダを利用するコンポーネントは，EJB の場合は EJB 3.0 以降，Web コンポーネントの場合は Servlet 2.5 以降となります。使用できるコンポーネントの詳細については，「5.3.2　使用できるコンポーネント」を参照してください。

(2)　接続できるデータベース

CJPA プロバイダを使用する場合，次に示すデータベースに接続できます。

- Oracle
- HiRDB

(3)　使用できる DB Connector

CJPA プロバイダは，DB Connector を使用してデータベースを更新します。CJPA プロバイダでは次の表に示す DB Connector を使用できます。

表 6-3　CJPA プロバイダで使用できる DB Connector

使用するデータベース	トランザクションタイプ	使用できる DB Connector
Oracle	LocalTransaction NoTransaction	DBConnector_Oracle_CP.rar
	XATransaction	DBConnector_Oracle_XA.rar
Oracle RAC	LocalTransaction NoTransaction	DBConnector_Oracle_CP.rar DBConnector_CP_ClusterPool_Root.rar DBConnector_Oracle_CP_ClusterPool_Member.rar
HiRDB	LocalTransaction NoTransaction	DBConnector_HiRDB_Type4_CP.rar
	XATransaction	DBConnector_HiRDB_Type4_XA.rar

(4)　使用できない環境

次の環境では CJPA プロバイダを使用できません。

- バッチアプリケーションの実行環境
- EJB クライアントアプリケーションの実行環境

(5)　メソッドキャンセル機能の使用について

CJPA プロバイダでは，OneToOne および ManyToOne の LAZY フェッチの場合，アクセサメソッドに独自のバイナリコードを埋め込みます。これによって，アクセサメソッドがメソッドキャンセルの対象になります。このため，OneToOne リレーションシップ，または ManyToOne リレーションシップで LAZY フェッチが指定されている場合は，エンティティクラス，埋め込み可能クラス，およびマップドスーパークラスを保護区に登録する必要があります。

なお，保護区に登録しない場合は，埋め込んだバイナリコード上でメソッドキャンセルが起こるおそれがあります。保護区に登録しない場合の動作は保証しません。

保護区に登録されている場合でも，登録されていない場合でも，メソッドタイムアウトで下記のようなスタックトレースが出力されます。ただし，保護区に登録しているとメソッドキャンセルは発生しません。

次に示す例では，ユーザが呼び出した EntityClass1.getMappingClass2 メソッドの延長で，埋め込んだ EntityClass1._toplink_getmappingClass2 が呼ばれています。そこでメソッドタイムアウトが発生していますが，メソッドキャンセルは発生しません。

```
message-id       message(LANG=ja)
KDJE52703-W      A timeout occurred while the user program was executing. (threadID = 23794987, rootAPInfo =
10.209.11.124/5964/0x4828eb62000128e0, application = JPA_JavaEE_TP, bean = TestBean, method = doTest)
    jpa.test.annotation.onetoone.entity.EntityClass1._toplink_getmappingClass2(EntityClass1.java)
                        locals:
                          (jpa.test.annotation.onetoone.entity.EntityClass1) this = <0x11e35878>
(jpa.test.annotation.onetoone.entity.EntityClass1)
                         at jpa.test.annotation.onetoone.entity.EntityClass1.getMappingClass2(EntityClass1.java:
34)
                        locals:
                          (jpa.test.annotation.onetoone.entity.EntityClass1) this = <0x11e35878>
(jpa.test.annotation.onetoone.entity.EntityClass1)
```

保護区への登録については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「2.6　criticalList.cfg（保護区リストファイル)」を参照してください。

(6)　アノテーションの参照抑止機能の使用について

CJPA プロバイダを使用する場合，アノテーションの参照抑止機能は使用できません。アノテーションの参照抑止機能が有効になっていると，永続化コンテキストおよび永続化ユニットの参照を定義できません。

6.2.4　DB Connector のコネクション数の見積もり

CJPA プロバイダを使用する場合に必要となる DB Connector のリソースの見積もりについて説明します。

CJPA プロバイダでは，DB Connector のコネクションを取得します。CJPA プロバイダが使用するコネクション数の見積もり式を次に示します。

コネクション数の見積もり式

CJPA プロバイダのアプリケーションで利用されるコネクション数

＝ JPA 機能を利用したアプリケーションの同時実行数※

注※　JPA 機能を利用したアプリケーションで，複数の永続化ユニットの EntityManager を利用する場合や，ビジネスメソッドの呼び出しで EntityManager が利用するトランザクションが異なる場合，EntityManager ごとにそれぞれコネクションが必要となります。

! 注意事項

ユーザが JPA 機能とは別にコネクションを取得する場合，コネクションシェアリング機能を利用することでコネクション数を削減できます。コネクションシェアリングの詳細については，「3.14.3　コネクションシェアリング・アソシエーション」を参照してください。

6.3　エンティティを使用したデータベースの更新

CJPA プロバイダでは，エンティティを使用したデータベースの更新ができます。

JPA を使用するには，ユーザはデータベースのテーブルを Java のオブジェクトとして扱うためのエンティティクラスを作成する必要があります。エンティティクラスを使用すると，CJPA プロバイダを通してデータベースのデータを操作します。このとき，CJPA プロバイダが提供する EntityManager インタフェースを介して，データベースを操作します。

エンティティクラスを使用したデータベースの操作は次のとおりです。

1. エンティティクラスのインスタンス（エンティティオブジェクト）を生成します。
2. EntityManager インタフェースの引数に，エンティティオブジェクトを渡し，データベースを操作します。
   この操作をするためには，エンティティクラスを作成する必要があります。

EntityManager とはエンティティを操作して，状態を制御するためのオブジェクトです。エンティティの操作には，persist 操作，remove 操作，merge 操作，flush 操作，および refresh 操作があります。これらの操作の詳細については，「6.4.1　エンティティの状態遷移」を参照してください。

6.4　EntityManager によるエンティティの操作

エンティティの状態は，CJPA プロバイダで提供する EntityManager で制御されます。ここでは，EntityManager でのエンティティの操作について説明します。

6.4.1　エンティティの状態遷移

エンティティには状態があります。エンティティに対して操作するとエンティティの状態は遷移します。ここでは，エンティティの状態の種類，エンティティに対して実行する操作，およびエンティティの操作と状態遷移について説明します。

(1)　エンティティの状態の種類

エンティティの状態には，new，managed，detached，removed の 4 種類があります。EntityManager のメソッドを利用してエンティティを操作することで，エンティティの状態が変更されます。

エンティティのそれぞれの状態について次の表に示します。

表 6–4　エンティティの状態

エンティティインスタンスの状態	説明
new	新しく生成された状態のエンティティです。 新しく生成されたエンティティインスタンスは，永続化アイデンティティを持ちません。このため，永続化コンテキストに関連づいていません。
managed	永続化コンテキストに関連づけられた永続化アイデンティティを持ち，永続化コンテキストで管理されている状態です。
detached	永続化コンテキストに関連づけられていない永続化アイデンティティを持つ状態です。
removed	永続化アイデンティティを持ち，永続化コンテキストに関連づけられている状態です。また，エンティティインスタンスがデータベースから削除されることが予定されている状態でもあります。

(2)　エンティティに対する操作

ユーザがデータベースのレコードを検索すると，CJPA プロバイダは取得したデータをエンティティのフィールドに保持します。また，ユーザがデータベースの内容を更新する場合は，永続化コンテキストに登録されているエンティティの状態を変更して，トランザクションをコミットすることでデータベースにエンティティの状態を反映します。このように，エンティティに対して操作することで，データベースの情報を更新します。

エンティティに対する操作の種類を次の表に示します。

表 6–5　エンティティに対する操作の種類

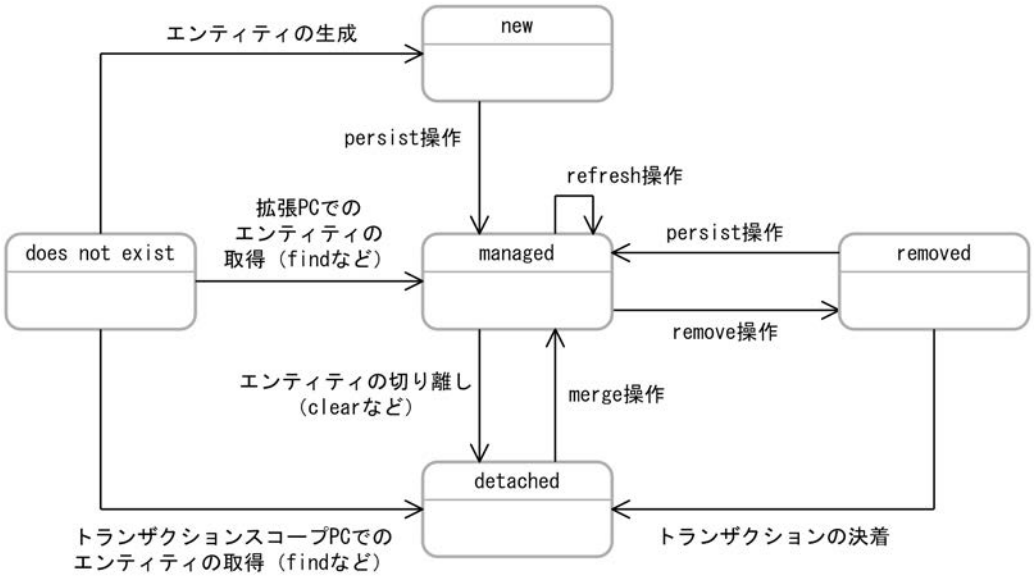
操作	説明
flush	エンティティオブジェクトの内容をデータベースに反映します。
merge	EntityManager の管理対象外となっているエンティティオブジェクトを EntityManager の管理対象とします。

操作	説明
persist	エンティティオブジェクトを管理して，永続化します。
refresh	エンティティオブジェクトにデータベースの内容を反映します。
remove	エンティティオブジェクトを削除予定状態にします。

(3)　エンティティに対する操作と状態遷移

エンティティインスタンスに対する操作と状態遷移について次の図で説明します。

図 6-3　エンティティインスタンスに対する操作と状態遷移



エンティティに対する操作と状態遷移について次の表にまとめます。

表 6-6　エンティティの状態遷移

操作	状態			
	new	managed	detached	removed
persist	managed※1	managed	managed※1	managed
remove	new	removed	例外※2，※3	Removed
merge	• コピー元 new • コピー先 managed	• コピー元 managed • コピー先 managed	• コピー元 detached • コピー先 managed	例外※3，※4
refresh	例外※2，※4	managed※5	例外※2，※4	例外※2，※4
commit	−	※6	−	detached
rollback	−	detached	−	detached
flush	−	managed	−	detached

操作	状態			
	new	managed	detached	removed
clear	−	detached	−	detached

（凡例）−：該当しない

注※1　persist 操作で例外が発生した場合，状態は遷移しないで元の状態になります。

注※2　発生する例外は IllegalArgumentException です。

注※3　データベース上に対応する行が存在しない場合，操作は無視されます。

注※4　例外時には状態は遷移しないで，元のままとなります。

注※5　データベース上に対応する行が存在しない場合，EntityNotFoundException が発生します。

注※6　トランザクションスコープの永続化コンテキストの場合は detached になります。拡張された永続化コンテキストの場合は managed になります。

また，persist，remove，merge，refresh をトランザクション外で実行した場合の動作は，永続化コンテキストの種類によって異なります。

- トランザクションスコープの永続化コンテキストの場合
  TransactionRequiredException となります。
- 拡張された永続化コンテキストの場合
  状態が遷移して，次のトランザクション決着のタイミングでデータベースに状態が反映されます。

(4)　エンティティに対する操作の伝播

エンティティがリレーションシップを持つ場合に，リレーションシップを表すアノテーションの cascade 属性を指定すると，エンティティに対する操作が関連するエンティティに対して伝播されます。cascade 属性には，次の表に示す値を指定できます。

表 6-7　cascade 属性の種類

cascade 属性の種類	説明
CascadeType.ALL	persist，remove，merge，refresh の操作が関連先に伝播されます。
CascadeType.PERSIST	persist 操作が関連先に伝播されます。
CascadeType.REMOVE	remove 操作が関連先に伝播されます。
CascadeType.MERGE	merge 操作が関連先に伝播されます。
CascadeType.REFRESH	refresh 操作が関連先に伝播されます。

6.4.2　エンティティに対する persist 操作

エンティティに対する persist 操作を実行するには，EntityManager の persist メソッドを呼び出します。EntityManager の persist メソッドを呼び出したり，persist 処理がカスケードされたりすると，エンティティはデータベースへの永続化や永続化コンテキストでの管理対象になります。

次の表に，persist 操作後のエンティティの状態をエンティティの状態ごとに示します。

表 6-8　Persist 操作でのエンティティの状態

エンティティの状態	persist 操作後のエンティティの状態
new※	managed に遷移します。managed に遷移したエンティティはトランザクションのコミット時またはコミット前にデータベースに追加されます。または，flush 操作の実行の結果としてデータベースに追加されます。
managed	persist 操作は無視されます。ただし，エンティティからほかのエンティティへのリレーションシップの cascade 属性に PERSIST または ALL が指定されていると，このエンティティが参照するエンティティに persist 操作が伝播されます。
detached※	データベースにエンティティと対応する行が存在しない場合は，managed に遷移します。対応する行が存在する場合，EntityExistsException が発生します。
removed	managed に遷移します。

注※　CJPA プロバイダの場合，エンティティの状態が new または detached の場合，データベース上にエンティティに対応するデータが存在するかどうかでエンティティの状態遷移の結果が異なります。CJPA プロバイダの場合，次のことに注意してください。

- データベースに永続化されていないエンティティを引数に指定した場合は，managed に遷移します。
- データベースに引数で渡されたエンティティと重複する行が存在し，そのエンティティが永続化コンテキストで管理されている場合，persist 処理時に EntityExistsException が発生します。
- データベースに引数で渡されたエンティティと重複する行が存在するが，永続化コンテキストでエンティティが管理されていない場合，flush やコミット時に EntityExistsException またはほかの PersistenceException が発生します。

! 注意事項

エンティティをデータベースに永続化したり，データベースからエンティティの情報を読み込んだりするタイミングで，エンティティは永続化コンテキストに登録されます。エンティティを managed に遷移したあとに，永続化処理をロールバックした場合は，エンティティは永続化コンテキストで管理されていないので注意してください。

6.4.3　エンティティに対する remove 操作

エンティティに対する remove 操作を実行するには，EntityManager の remove メソッドを呼び出します。EntityManager の remove メソッドを呼び出したり，remove 処理がカスケードされたりすると，エンティティは removed 状態になります。removed 状態のエンティティはトランザクションのコミット処理でデータベースから削除されます。

次の表に，remove 操作後のエンティティの状態をエンティティの状態ごとに示します。

表 6-9　remove 操作でのエンティティの状態

エンティティの状態	状態遷移の結果
new	remove 操作は無視されます。ただし，エンティティからほかのエンティティへのリレーションシップの cascade 属性に REMOVE または ALL が指定されていると，このエンティティが参照するエンティティに remove 操作が伝播されます。
managed	removed に遷移します。エンティティからほかのエンティティへのリレーションシップの cascade 属性に REMOVE または ALL が指定されていると，このエンティティが参照するエンティティに remove 操作が伝播されます。

エンティティの状態	状態遷移の結果
detached	次のように遷移します。 • 引数で渡された detached 状態のエンティティに対応する行がデータベースに存在する場合，remove 操作時に IllegalArgumentException がスローされます。 • CJPA プロバイダの場合，データベースに対応する行が存在しないときには，remove 操作は無視されます。ただし，ほかのエンティティへのリレーションシップの cascade 属性に REMOVE または ALL が指定されていれると，このエンティティが参照するエンティティに remove 操作が伝播されます。
removed	remove 操作は無視されます。ほかのエンティティへの伝播もしません。

注 1　removed 状態のエンティティはトランザクションのコミット時，トランザクションのコミット前，または flush 操作時の実行結果としてデータベースから削除されます。

注 2　エンティティが削除されたあと，エンティティの内容は remove 処理が呼び出された直後の内容になります。ただし，エンティティの生成直後の場合を除きます。

6.4.4　データベースからのエンティティの取得

EntityManager の find メソッドまたは getReference メソッドの呼び出しによって，引数で指定した主キーに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。

EntityManager の find メソッドまたは getReference メソッドで返されるエンティティは，トランザクションスコープの永続化コンテキストを利用します。このため，トランザクション内でメソッドを呼び出した場合は managed 状態になります。また，トランザクション外で find メソッドまたは getReference メソッドを呼び出した場合は detached 状態になります。

find 操作または getReference 操作で取得したエンティティの状態を次の表に示します。

表 6–10　find 操作または getReference 操作で取得したエンティティの状態

永続化コンテキスト	取得したエンティティの状態
トランザクションスコープの永続化コンテキスト（トランザクション外）	detached
トランザクションスコープの永続化コンテキスト（トランザクション内）	managed
拡張された永続化コンテキスト（トランザクション外）	managed
拡張された永続化コンテキスト（トランザクション内）	managed

なお，CJPA プロバイダの場合，find 操作または getReference 操作では次の点が JPA 仕様と異なります。なお，このほかに JPA 仕様との機能差はありません。

- JPA 仕様では，EntityManager の getReference メソッドでは実際のインスタンスではなく，引数で与えられた主キーに対応するエンティティに Proxy を返すことができます。ただし，CJPA プロバイダでは，@Basic に対する Lazy ローディングをサポートしていません。このため，CJPA プロバイダでは，getReference の戻り値として，Proxy ではなくエンティティインスタンスを返します。

  なお，リレーションシップに対する Lazy ローディングのサポート範囲については，「6.4.5(2)　データベースからのエンティティ情報の読み込み」を参照してください。
- CJPA プロバイダでは，find メソッドと getReference メソッドの間で存在しないエンティティを引数に指定した場合，find メソッドでは null 値が返ります。また，getReference メソッドでは EntityNotFoundException が発生します。

6.4.5　データベースとの同期

トランザクションのコミットまたは flush メソッドの実行時に，エンティティの状態がデータベースに書き込まれます。一方，明示的に refresh を呼び出さないかぎり，メモリにロードされたエンティティの状態のリフレッシュは実行されません。

ここでは，データベースへのエンティティ情報の書き込みと，データベースからのエンティティ情報の読み込みについて説明します。

(1)　データベースへのエンティティ情報の書き込み

flush 操作またはトランザクションのコミットでのエンティティの状態遷移について説明します。次の表ではエンティティ A の状態ごとに状態遷移の結果を示しています。

表 6-11　flush 操作またはトランザクションのコミットでのエンティティインスタンスの状態遷移

エンティティ A の状態	状態遷移の結果
new	flush 操作は無視されます。
managed	データベースと同期されます。
removed	エンティティ A はデータベースから削除されます。
detached	flush 操作は無視されます。

また，managed 状態のエンティティ A がエンティティ B に対してリレーションシップを持っている場合，flush 処理の延長で次の表に示す条件に従って，persist 操作がカスケードされます。

表 6-12　関連するエンティティ B への flush 処理およびコミットでの persist 操作のカスケード

エンティティ B へのリレーションシップの cascade 属性の指定	エンティティ B の状態	結果
PERSIST または ALL が指定されている	−	persist 操作がエンティティ B にカスケードされます。
PERSIST または ALL が指定されていない	new	• flush 操作の場合 IllegalStateException が発生してトランザクションはロールバックにマークされます。 • コミット処理の場合 IllegalStateException が発生してトランザクションのコミットに失敗します。
	managed	データベースと同期化されます。
	removed	• flush 操作の場合 IllegalStateException が発生してトランザクションはロールバックにマークされます。 • コミット処理の場合 IllegalStateException が発生してトランザクションのコミットに失敗します。
	detached	• エンティティ A がリレーションシップの所有者の場合 リレーションシップの変更はデータベースと同期されます。

エンティティ B へのリレーションシップの cascade 属性の指定	エンティティ B の状態	結果
PERSIST または ALL が指定されていない	detached	• エンティティ B がリレーションシップの所有者の場合 例外が発生します。CJPA プロバイダでは，この場合の動作はサポート対象外となります。

（凡例）－：該当しない

なお，トランザクションの外で flush メソッドを呼び出すと，TransactionRequiredException が発生します。

ポイント

リレーションシップとデータベースに対する永続化の関連

- 双方向のリレーションシップを持つ managed 状態のエンティティは，リレーションシップの所有者側で保持される参照を基に永続化されます。アプリケーション開発者は，変更が発生したときに所有者側と被所有者側でそれぞれ，メモリ上の状態に矛盾がないようにエンティティを保持するようアプリケーションを作成してください。
- 単方向の OneToOne, OneToMany の場合, エンティティで定義しているリレーションシップの関係とデータベースのテーブル間の関係が合っていることは，開発者の責任で保証してください。

(2)　データベースからのエンティティ情報の読み込み

EntityManager の refresh メソッドを呼び出すと，それまでに行われたエンティティの変更は破棄され，データベースの内容でエンティティの状態を上書きします。このとき, データベース上に対応する行が存在しない場合は EntityNotFoundException が発生します。

エンティティが managed 状態以外の場合に，EntityManager の refresh メソッドを呼び出すと IllegalArgumentException が発生します。

(a)　データベースからのエンティティ情報を読み込むタイミング

refresh メソッドもしくは find メソッドの実行時，またはクエリの発行時に，データベースからエンティティが読み込まれます。このときに，関連するエンティティもあわせて読み込むことができます。これをフェッチ戦略といいます。フェッチ戦略は各リレーションシップの fetch 属性で指定します。fetch 属性には次の 2 種類のどちらかを指定します。

- FetchType.EAGER
  データベースからエンティティの情報を読み込むときに，関連するフィールドやエンティティの情報を読み込みます。
- FetchType.LAZY
  フィールドまたは関連先に初めてアクセスしたときにデータベースからの読み込みが実行されます。これを，Lazy ローディングといいます。

FetchType.EAGER を指定するとデータベースからエンティティの情報を読み込むたびに，関連するフィールドやエンティティの情報を読み込みます。このため, 不要な関連先のエンティティの取得を回避したい場合は，FetchType.LAZY を指定してください。

CJPA プロバイダでの fetch 属性のサポート範囲を次の表に示します。

表 6-13　リレーションシップごとの fetch 属性のサポート範囲

リレーションシップのアノテーション	サポート範囲
@ManyToMany	デフォルトは FetchType.LAZY です。
@OneToMany	デフォルトは FetchType.LAZY です。
@OneToOne	デフォルトは FetchType.EAGER です。なお，LAZY を指定したときには，(b)を参照してください。
@ManyToOne	デフォルトでは FetchType.EAGER です。なお，LAZY を指定したときには，(b)を参照してください。
@Basic	fetch 属性は無視されます。デフォルトの FetchType.EAGER が常に適用されます。

(b)　@OneToOne および@ManyToOne での LAZY フェッチ

@OneToOne および@ManyToOne のリレーションシップに対してフェッチ戦略に LAZY を指定した場合，エンティティクラスのローディング時に，LAZY を指定したフィールドの getter メソッドにバイナリコードを埋め込みます。

getter メソッドが呼び出されると，埋め込んだバイナリコードの処理を通して，データベースから関連先のエンティティを取得します。getter メソッドがバイナリの埋め込み対象になるため，リレーションの対象となるフィールドにアクセスするための getter メソッドを設定しておく必要があります。また，その getter メソッドに@OneToOne または@ManyToOne を設定しておく必要があります。

! 注意事項

getter メソッドでは，リレーションシップの targetEntity の型から，関連先のエンティティの型を決定します。targetEntity に指定するクラスの型は，フィールド・プロパティの型にキャストできる必要があります。

6.4.6　永続化コンテキストからのエンティティの切り離しと merge 操作

永続化コンテキストから切り離されたエンティティを detached 状態のエンティティといいます。エンティティは次のタイミングで detached 状態になります。

- トランザクションスコープの永続化コンテキストでのトランザクションをコミットしたとき
- トランザクションのロールバックを実行したとき
- 永続化コンテキストをクリアしたとき（EntityManager.clear()を呼び出したとき）
- EntityManager をクローズしたとき
- エンティティのシリアライズおよびエンティティの値渡しをしたとき

切り離されたエンティティのインスタンスは，永続化や取得をした永続化コンテキストの外で存在し続けます。エンティティの状態とデータベースの状態は同期されません。

(1)　detached 状態のエンティティへのアクセス

アプリケーションは永続化コンテキストの終了後でも，detached 状態のエンティティにアクセスできます。この場合，エンティティのフィールドおよび関連先は，フェッチ済みである必要があります。detached 状態のエンティティでフェッチされていないフィールドおよび関連先のエンティティにはアクセスできま

せん。なお，フィールドに指定する@Basic には常に FetchType.EAGER が適用されるため，関連先のエンティティやフィールドの情報が取得済みとなります。

リレーションシップでは detached 状態のエンティティからアクセスするには，次に示すどれかの条件を満たす必要があります。

- find()で取得されたエンティティインスタンスであること
- クエリを使って取得されたエンティティまたは FETCH JOIN 節で明示的に要求されたエンティティであること
- プライマリキーでない永続状態のインスタンスをアプリケーションでアクセス済みであること
- fetch=EAGER と指定された関連をたどって別の有効なエンティティから取得済みのエンティティであること

利用できないインスタンスへのアクセス，および利用できるインスタンスの無効なステートへのアクセスを行った場合は，例外が発生します。ただし，CJPA プロバイダでは，FetchType.LAZY を指定した場合，EntityManager の終了後に detached 状態になっていても，フェッチされていないフィールドおよび関連先エンティティにアクセスすると，データベースから値を取得して，内容を参照することができる場合があります。

(2)　エンティティの merge 処理

EntityManager の merge メソッドの呼び出し，または merge 処理のカスケードによって，detached 状態のエンティティを EntityManager で管理される永続化コンテキストにマージできます。

次の表に merge 処理でのエンティティの状態遷移を，エンティティ A の状態ごとに示します。

表 6-14　merge 処理でのエンティティの状態遷移の結果

エンティティ A の状態	状態遷移の結果
new	managed 状態のエンティティ A'が新しく作成されて，エンティティ A の状態がエンティティ A'にコピーされます。merge の引数のエンティティは new のままであることに注意してください。
managed	merge 操作は無視されます。しかし，エンティティ A からほかのエンティティへのリレーションシップの cascade 属性に，MERGE または ALL が指定されている場合，エンティティ A が参照するエンティティに merge 操作が伝播されます。
detached	エンティティ A の状態が，同じ ID を持つすでに存在している管理されたエンティティ A'にコピーされます。または，エンティティ A のコピーである新しい managed 状態のエンティティが作成されます。merge の引数のエンティティは detached 状態のままであることに注意してください。
removed	merge 操作によって，IllegalArgumentException が発生します。または，トランザクションのコミットに失敗します。エンティティ A の状態は removed 状態のままであることに注意してください。

注 1　エンティティ A からエンティティ B へのリレーションが cascade=MERGE または cascade=ALL で指定されていると，すべてのエンティティ B は再帰的にエンティティ B'としてマージされます。エンティティ A'にはエンティティ B'がセットされます。なお，エンティティ A が managed 状態の場合，エンティティ A とエンティティ A'は同じであることに注意してください。

注 2　エンティティ A のリレーションに cascade=MERGE または cascade=ALL が指定されていない状態でエンティティ B を参照している場合，エンティティ A がエンティティ A'にマージされるときには，エンティティ A'から関連をたどるとエンティティ B と同じ永続化アイデンティティを持つ管理されたエンティティ B'の参照にたどり着きます。

JPA 仕様では，フェッチされていないことを意味する LAZY にマークされたフィールドはマージのときに無視されます。ただし，CJPA プロバイダでは，@Basic は EAGER として動作するため，リレーションシップでないすべてのフィールドはマージ処理の対象となります。

また，Version 列がエンティティで使われている場合は，マージ操作とそのあとに呼ばれるフラッシュおよびコミット処理時に，エンティティのバージョンチェックを実行します。Version 列が存在しない場合，merge 操作ではエンティティのバージョンチェックは実行されません。詳細については，「6.10.1　楽観的ロックの処理」を参照してください。

なお，CJPA プロバイダでは，ほかベンダとの間で，エンティティのマージ処理によって永続化コンテキストに戻すという処理はサポートしていません。

(3)　注意事項

- CJPA プロバイダの場合，トランザクションスコープの永続化コンテキストでは，managed 状態のエンティティはトランザクションのコミットによって detached 状態になります。一方，拡張された永続化コンテキストでは，managed 状態のエンティティは管理されたままとなります。
- トランザクションスコープでも拡張された永続化コンテキストでも，トランザクションをロールバックすると，すべての存在している managed 状態のインスタンスと removed 状態のインスタンスは，detached 状態になります。インスタンスの状態は，トランザクションがロールバックされた時点の状態です。
- トランザクションをロールバックすると，永続化コンテキストの状態はロールバックした時点の状態になるため，データベースの状態と矛盾します。なお，CJPA プロバイダの場合は，バージョン属性の状態と生成された状態は矛盾した状態になるため，merge 操作などを行うと例外になることがあります。

6.4.7　managed 状態のエンティティ

EntityManager の contains()メソッドはエンティティのインスタンスがカレントの永続化コンテキストで管理されているかを取得するために利用できます。

ここでは，contains()メソッドの戻り値の条件について説明します。

- contains()メソッドが true を返す条件
  - エンティティがデータベースから取得されていて，EntityManager から削除されていない場合，または切り離されていない場合
  - エンティティのインスタンスが生成され，persist メソッドがそのエンティティに対して実行されている場合，または persist 操作がそのエンティティに伝播されている場合
- contains()メソッドが false を返す条件
  - エンティティのインスタンスが切り離されている場合
  - remove メソッドがエンティティに対して実行されている場合，または remove 操作がエンティティに伝播されている場合
  - エンティティのインスタンスが生成され，persist メソッドがそのエンティティに対して実行されていない場合，または persist 操作がそのエンティティに伝播されていない場合

実際のデータベースでの insert および delete 処理がトランザクションの決着まで遅延されます。これに対して，persist や remove の伝播は contains メソッドではすぐに反映されることに注意してください。

なお，エンティティのインスタンスが単一の永続化コンテキストでだけ管理されていることは，アプリケーション側で保証してください。CJPA プロバイダでは，同じ Java のインスタンスを複数の永続化コンテキストで管理した場合の動作は保証しません。

6.5 データベースと Java オブジェクトとのマッピング情報の定義

CJPA プロバイダでは，データベースと Java オブジェクトをマッピングするための情報を定義できます。マッピング情報はアノテーションまたは O/R マッピングファイルで定義します。

- **アノテーションを使用した定義**
  アプリケーションのエンティティクラスに直接マッピング情報を定義します。
- **O/R マッピングファイルを使用した定義**
  O/R マッピングファイルとは，マッピング情報を記載するための XML 形式のファイルです。タグを使用してマッピング情報を定義します。

アノテーションと O/R マッピングファイルの両方でマッピング情報を定義している場合, O/R マッピングファイルの定義が優先されます。このため, アノテーションを使用して定義したマッピング情報を変更する場合に O/R マッピングファイルを使用すると，アプリケーションを変更することなくマッピング情報を変更できます。

アプリケーションの作成方針に合わせて，アノテーションを使用するか，O/R マッピングファイルを使用するか，またはアノテーションと O/R マッピングファイルを併用するかを決定してください。

6.6　エンティティのリレーションシップ

エンティティでは，データベースのテーブル間の関連をリレーションシップで表現できます。CJPA プロバイダでは，エンティティのリレーションシップを設定できます。

6.6.1　リレーションシップの種類

リレーションシップには関連と方向があります。関連には，OneToOne，ManyToOne，OneToMany，および ManyToMany があります。また，それぞれの関連の方向には，単方向と双方向があります。関連と方向を組み合わせると，リレーションシップの種類には次の 7 種類があります。

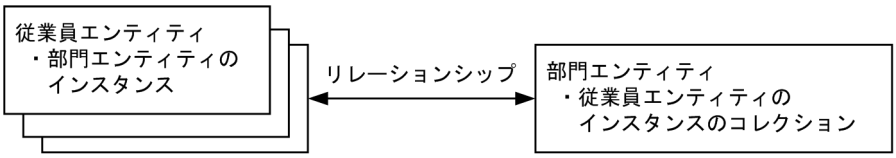
- 単方向の OneToOne リレーションシップ
- 単方向の ManyToOne リレーションシップ
- 単方向の OneToMany リレーションシップ
- 単方向の ManyToMany リレーションシップ
- 双方向の OneToOne リレーションシップ
- 双方向の ManyToOne/OneToMany リレーションシップ
- 双方向の ManyToMany リレーションシップ

ある会社の従業員と部門を例にリレーションシップについて説明します。従業員と部門は次のような関連があります。

- エンティティでは，従業員と部門をそれぞれ表現します。
- 従業員は必ずどこかの部門に所属するものとします。
- 部門では複数の従業員を保持します。
- 従業員から部門を参照したり，部門から従業員を参照したりします。

この例の場合，エンティティの双方向の ManyToOne/OneToMany リレーションシップで表現します。この場合，Many が従業員，One が部門となります。従業員のエンティティと部門のエンティティの関連を次の図に示します。

図 6-4　従業員のエンティティと部門のエンティティの関連



なお，エンティティには，関連先のエンティティに操作を伝播させる設定ができます。この設定をカスケードといいます。カスケードが設定されていると，エンティティに対して操作した場合，操作したエンティティとリレーションシップがあるエンティティにも自動的に同様の操作が実行されます。カスケードを利用すると，ユーザは関連先のエンティティへの操作の手間を省くことができます。

6.6.2　リレーションシップのアノテーション

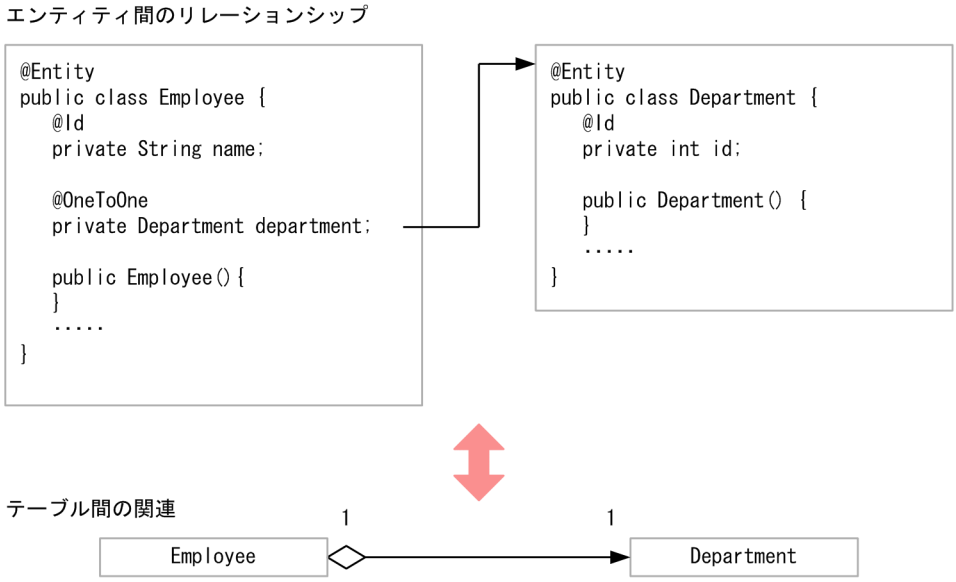
エンティティではテーブル間の関連をリレーションシップとして扱います。リレーションシップをエンティティで扱う場合，エンティティでフィールドとしてほかのエンティティへの参照を保持します。その上

で，次に示すリレーションシップのアノテーションを，エンティティを参照する永続化プロパティまたはインスタンス変数に設定してください。

- OneToOne（1 対 1）
- OneToMany（1 対多）
- ManyToOne（多対 1）
- ManyToMany（多対多）

エンティティのリレーションシップとテーブル間の関連を次の図に示します。

図 6-5　エンティティのリレーションシップとテーブル間の関連



エンティティへの参照に対して，リレーションシップのアノテーションが指定されていない場合は動作を保証しません。なお，コレクションを使用した参照で，generic 型を利用しない場合，リレーションシップの target として，対象のエンティティを指定する必要があります。

なお，アノテーションを使用する代わりに，O/R マッピングファイルでリレーションシップを定義することもできます。ただし，O/R マッピングファイルで定義している場合で，同じ定義をアノテーションに指定しているときには，アノテーションの定義は上書きされるので注意してください。

リレーションシップのアノテーションでのデフォルトマッピング

OneToOne，OneToMany，ManyToOne，ManyToMany のリレーションシップのアノテーションを適用した場合，デフォルトマッピングの規則が適用されます。O/R マッピングファイルでリレーションシップを指定した場合も同様に，デフォルトマッピングが適用されます。

リレーションシップのアノテーションのデフォルトマッピングを使用する場合，データベースのテーブルおよびリレーションが，「6.6.4　デフォルトマッピング（双方向のリレーションシップ）」または「6.6.5　デフォルトマッピング（単方向のリレーションシップ）」にあるデフォルトと異なると，実行時にデータベース問い合わせで SQL 文が正しく作成できません。例外が発生します。

6.6.3　リレーションシップの方向

リレーションシップには，双方向のリレーションシップと単方向のリレーションシップがあります。双方向のリレーションシップを扱う場合，リレーションシップには所有者と被所有者があります。単方向のリレー

ションシップには所有者だけがあります。リレーションシップの所有者は，データベースのリレーションシップの更新を決定できます。

双方向のリレーションシップでは，次のルールが適用されます。

- 双方向のリレーションシップの被所有者は，@OneToOne，@OneToMany，@ManyToMany の mappedBy 要素によって，所有者を指定します。mappedBy 要素では，所有者のエンティティで，被所有者側を参照しているプロパティまたはフィールドの名前を指定します。
- ManyToOne/OneToMany 双方向のリレーションシップでは，many 側を所有者にしてください。そのため，mappedBy 要素は@ManyToOne では指定できません。
- OneToOne の双方向のリレーションシップでは，外部キーを含む側のエンティティが所有者になります。
- ManyToMany の双方向のリレーションシップは，どちらが所有者でもかまいません。

リレーションシップのアノテーションには cascade 属性があります。cascade 属性を指定すると，エンティティに対する操作を参照先のエンティティに対しても伝播させることができます。ただし，リレーションシップのアノテーションの cascade 属性に REMOVE を指定できるのは，OneToOne または OneToMany のときだけです。ほかのリレーションに対して cascade=REMOVE を適用した場合の動作は保証しません。エンティティがリレーションシップを持つ場合の cascade 属性については，「6.4.1(4) エンティティに対する操作の伝播」を参照してください。

! 注意事項

CJPA プロバイダでは，実行時のリレーションシップの一貫性を保つためのチェックは実施しません。このため，アプリケーションの実行時にリレーションシップを更新する場合に，リレーションシップに矛盾が発生するような更新を行っても，警告や例外は発生しません。

なお，OneToMany，ManyToMany などのコレクションのリレーションシップでは，データベースから値をフェッチした場合に，関連するエンティティが存在しないときには，リレーションシップの値として空のコレクションを返します。

6.6.4　デフォルトマッピング（双方向のリレーションシップ）

ここでは，双方向のリレーションシップのデフォルトマッピングについて説明します。

(1)　双方向の OneToOne リレーションシップ

次に示す条件の場合に適用される双方向の OneToOne リレーションシップのデフォルトマッピングについて説明します。

条件

- エンティティ A はエンティティ B の単体のインスタンスを参照して，@OneToOne を設定します。
- エンティティ B はエンティティ A の単体のインスタンスを参照して，@OneToOne を設定します。@OneToOne の mappedBy 属性にエンティティ A でエンティティ B を参照する永続化プロパティ（またはフィールド）名を指定します。
- エンティティ A は，リレーションシップの所有者です。

適用されるデフォルトマッピング

- エンティティ A は，テーブル A にマップされます。
- エンティティ B は，テーブル B にマップされます。

- テーブル A は，テーブル B の外部キーを持つ必要があります。なお，外部キー列の名前は，次のようになります。

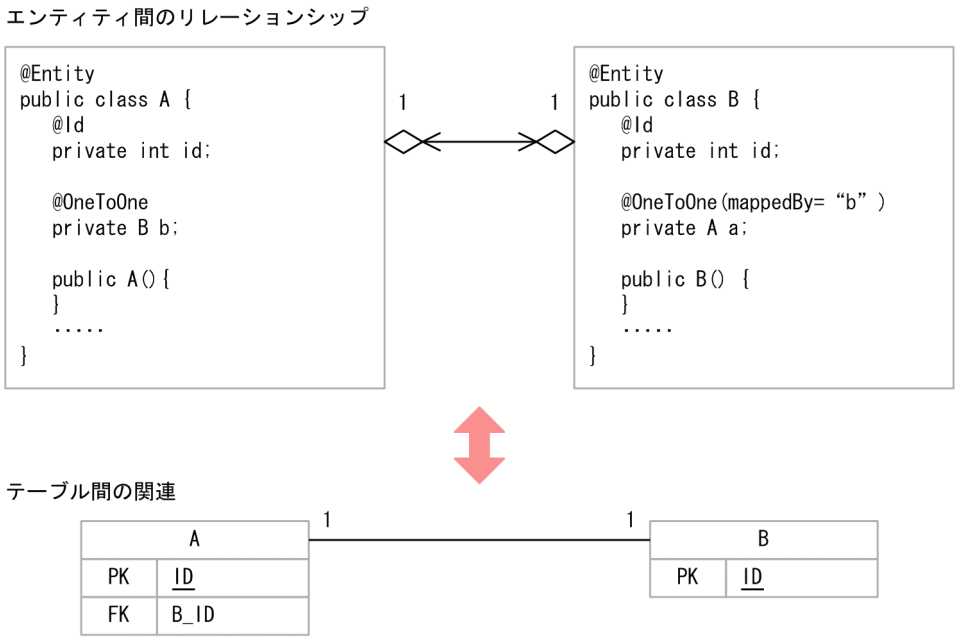
  外部キー列の名前

  エンティティ A のリレーションシッププロパティ（またはフィールド）の名前_テーブル B のプライマリキー列の名前

  注　斜体は可変値を表します。

また，外部キー列は，テーブル B のプライマリキーと同じ型を持ち，ユニークキー制約があります。

図 6-6　双方向の OneToOne でのデフォルトマッピング



(2)　双方向の ManyToOne/OneToMany リレーションシップ

次に示す条件の場合に適用される双方向の ManyToOne/OneToMany リレーションシップのデフォルトマッピングについて説明します。

条件

- エンティティ A はエンティティ B の単体のインスタンスを参照して，@ManyToOne（または，O/R マッピングファイルの該当する XML タグ）を設定します。
- エンティティ B は，エンティティ A のコレクションを参照して，@OneToMany（または，O/R マッピングファイルの該当する XML タグ）を設定します。@OneToMany には mappedBy 属性を指定します。mappedBy 属性は，エンティティ A で，エンティティ B を参照するために設定した永続プロパティ（またはフィールド）名を指定します。
- エンティティ A は，リレーションシップの所有者です。

適用されるデフォルトマッピング

- エンティティ A はテーブル A にマップされます。
- エンティティ B はテーブル B にマップされます。
- テーブル A は，テーブル B に対する外部キーを持つ必要があります。外部キー列の名前は，次のようになります。

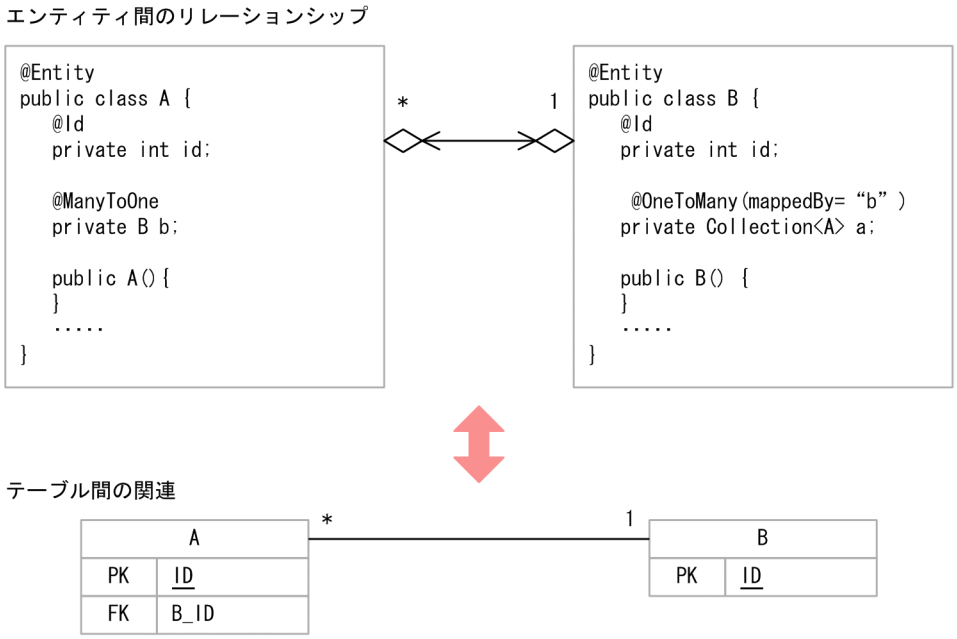
  外部キー列の名前

エンティティ A のリレーションシッププロパティ（またはフィールド）の名前_テーブル B のプライマリキー列の名前

注　斜体は可変値を表します。

外部キー列はテーブル B のプライマリキーと同じ型を持ちます。

図 6-7　双方向の ManyToOne/OneToMany でのデフォルトマッピング



(3)　双方向の ManyToMany リレーションシップ

次に示す条件の場合に適用される双方向の ManyToMany リレーションシップのデフォルトマッピングについて説明します。

条件

- エンティティ A は，エンティティ B のコレクションを参照します。コレクションには，@ManyToMany（または，O/R マッピングファイルで該当する XML 要素）を設定します。
- エンティティ B は，エンティティ A のコレクションを参照します。コレクションには，@ManyToMany(または，O/R マッピングファイルで該当する XML タグ)を設定して，mappedBy 属性を指定します。mappedBy 属性には，エンティティ A で，エンティティ B を参照するために設定した永続化プロパティ（またはフィールド）名を指定します。
- エンティティ A は，リレーションシップの所有者側です。

適用されるデフォルトマッピング

- エンティティ A は，テーブル A にマップされます。
- エンティティ B は，テーブル B にマップされます。
- テーブル A，B のほかに，所有者側のテーブルの名前が最初にくる A_B という名前の結合表が必要です。この結合表は，二つの外部キー列を持ちます。

  一つ目の外部キー列はテーブル A を参照し，テーブル A のプライマリキーと同じ型を持ちます。この外部キー列の名前は，次のようになります。

  外部キー列の名前

エンティティ B のリレーションシップのプロパティ（またはフィールド）の名前_テーブル A のプライマリキーの名前

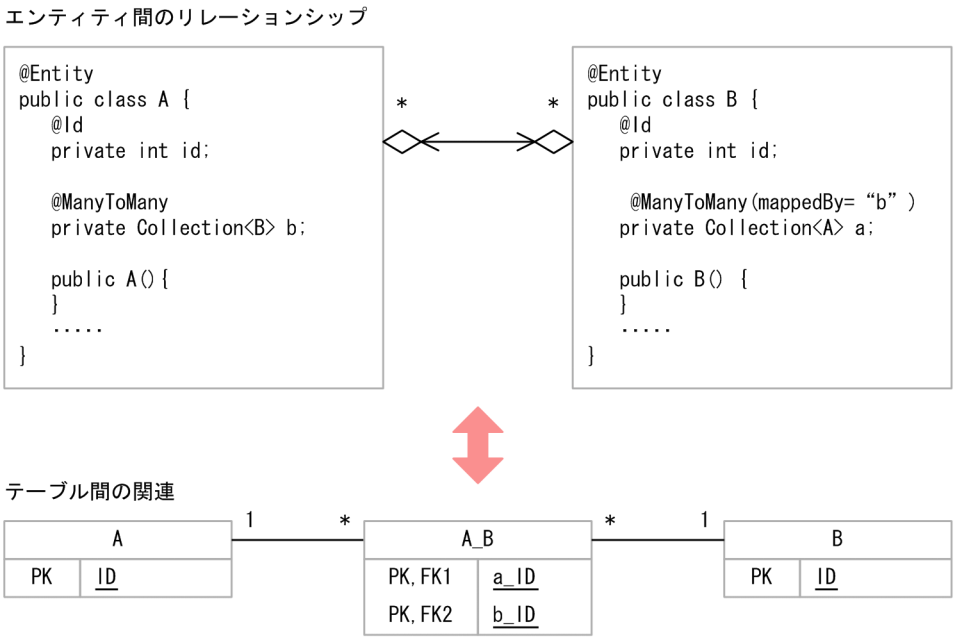
また，もう一つの外部キー列は，テーブル B を参照し，テーブル B のプライマリキーと同じ型を持ちます。この外部キー列の名前は，次のようになります。

外部キー列の名前

エンティティ A のリレーションシップのプロパティ（またはフィールド）の名前_テーブル B のプライマリキーの名前

注　斜体は可変値を表します。

図 6-8　双方向 ManyToMany でのデフォルトマッピング



6.6.5　デフォルトマッピング（単方向のリレーションシップ）

単方向のリレーションシップには, Single-Valued リレーションシップと Multi-Valued リレーションシップがあります。それぞれのリレーションシップについて説明します。

- **単方向の Single-Valued リレーションシップ**

  単方向の Single-Valued リレーションシップとは，単体のインスタンスを参照し，所有者だけが存在するリレーションシップのことです。

  単方向 Single-Valued リレーションシップには, 単方向の OneToOne リレーションシップと単方向の ManyToOne リレーションシップがあります。

- **単方向の Multi-Valued リレーションシップ**

  単方向の Multi-Valued リレーションシップとは，コレクションの形でエンティティを参照し，所有者だけが存在するリレーションシップのことです。

  単方向 Multi-Valued リレーションシップには，単方向の OneToMany リレーションシップと単方向の ManyToMany リレーションシップがあります。

ここでは，単方向のリレーションシップのデフォルトマッピングについて説明します。

(1)　単方向の Single-Valued リレーションシップ

単方向の OneToOne リレーションシップと単方向の ManyToOne リレーションシップのデフォルトマッピングについて説明します。

(a)　単方向の OneToOne リレーションシップ

次に示す条件の場合に適用される単方向の OneToOne リレーションシップのデフォルトマッピングについて説明します。

条件

- エンティティ A は，エンティティ B の単体のインスタンスを参照して，@OneToOne（または，O/R マッピングファイルで該当する XML タグ）を設定します。
- エンティティ B からは，エンティティ A を参照しません。
- エンティティ A がリレーションシップの所有者となります。

適用されるデフォルトマッピング

- エンティティ A は，テーブル A にマップされます。
- エンティティ B は，テーブル B にマップされます。
- テーブル A は，テーブル B に対する外部キーを持つ必要があります。外部キー列の名前は，次のようになります。

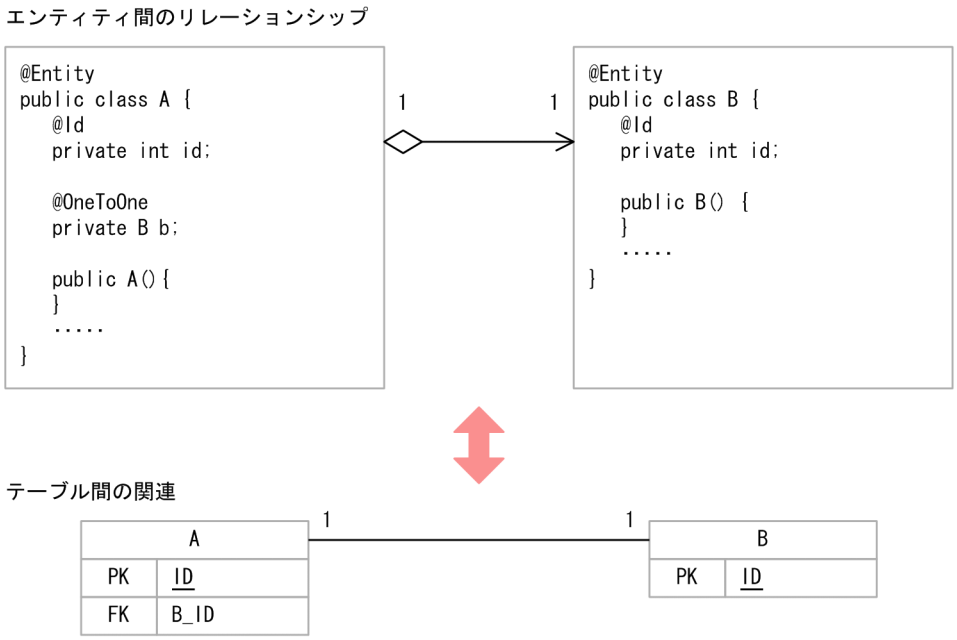
  外部キー列の名前

  エンティティ A のリレーションシップのプロパティ（または，フィールド）の名前_テーブル B のプライマリキー列の名前

  注　斜体は可変値を表します。

外部キー列は，テーブル B のプライマリキーと同じ型を持ち，ユニークキー制約です。

図 6-9　単方向の OneToOne でのデフォルトマッピング



(b)　単方向の ManyToOne リレーションシップ

次に示す条件の場合に適用される単方向の ManyToOne リレーションシップのデフォルトマッピングについて説明します。

条件

- エンティティ A は，エンティティ B の単体のインスタンスを参照し，@ManyToOne（または，O/R マッピングファイルで該当する XML タグ）を設定します。
- エンティティ B からは，エンティティ A を参照しません。

適用されるデフォルトマッピング

- エンティティ A は，テーブル A にマップされます。
- エンティティ B は，テーブル B にマップされます。
- テーブル A は，テーブル B に対する外部キーを持つ必要があります。外部キー列の名前は，次のようになります。

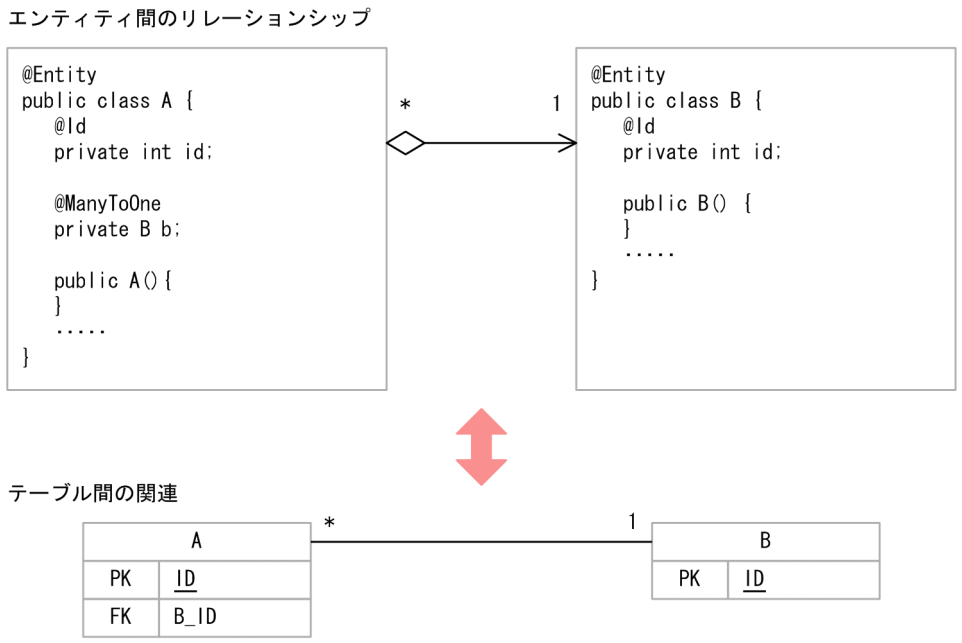
  外部キー列の名前

  エンティティ A のリレーションシップのプロパティ（またはフィールド）の名前_テーブル B のプライマリキー列の名前

  注　斜体は可変値を表します。

外部キー列は，テーブル B のプライマリキーと同じ型を持つ必要があります。

図 6-10　単方向の ManyToOne でのデフォルトマッピング



(2)　単方向の Multi-Valued リレーションシップ

単方向の OneToMany リレーションシップと単方向の ManyToMany リレーションシップのデフォルトマッピングについて説明します。

(a)　単方向の OneToMany リレーションシップ

次に示す条件の場合に適用される単方向の OneToMany リレーションシップのデフォルトマッピングについて説明します。

条件

- エンティティ A は，エンティティ B のコレクションを参照します。コレクションに@OneToMany（または，O/R マッピングファイルで該当する XML タグ）を設定します。
- エンティティ B からは，エンティティ A を参照しません。
- エンティティ A がリレーションシップの所有者です。

適用されるデフォルトマッピング

- エンティティ A は，テーブル A にマップされます。
- エンティティ B は，テーブル B にマップされます。
- テーブル A，B のほかに，所有者側のテーブルの名前が最初にくる A_B という名前の結合表が必要となります。この結合表は，二つの外部キー列を持ちます。

  一つ目の外部キー列は，テーブル A を参照し，テーブル A のプライマリキーと同じ型を持ちます。この外部キー列の名前は，次のようになります。

  外部キー列の名前

  エンティティ A の名前_テーブル A のプライマリキー列の名前

  また，もう一つの外部キー列は，テーブル B を参照し，テーブル B の外部キーと同じ型を持ち，ユニークキー制約があります。この外部キー列の名前は，次のようになります。

  外部キー列の名前

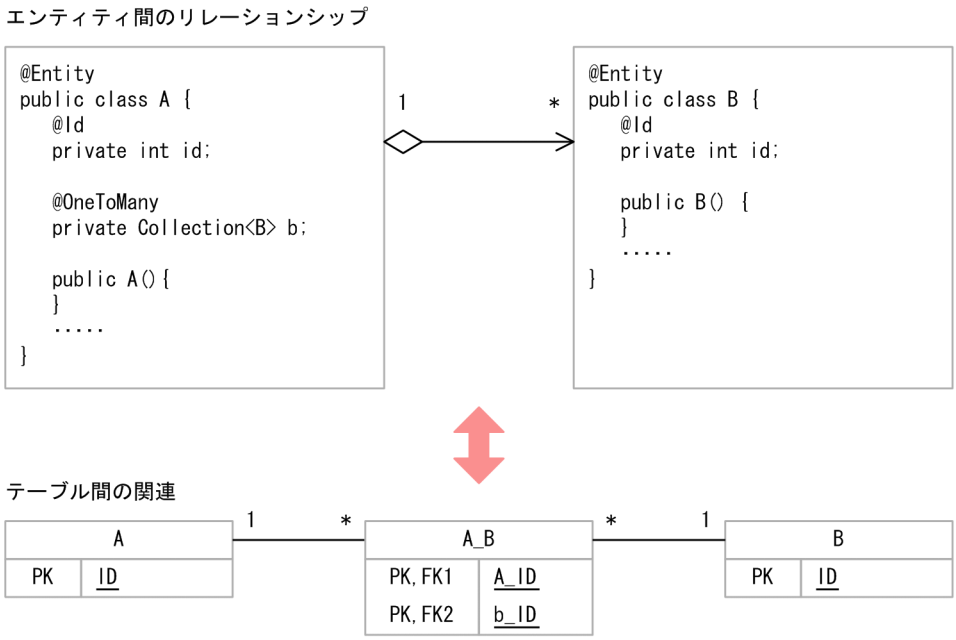
  エンティティ A のリレーションシッププロパティ（またはフィールド）の名前_テーブル B のプライマリキー列の名前

  注　斜体は可変値を表します。

  ! 注意事項

  この例のような結合表を使用した単方向の OneToMany リレーションシップは，CJPA プロバイダ以外の JPA プロバイダではサポートされていない可能性があります。単方向の OneToMany リレーションシップで作成したアプリケーションを CJPA プロバイダ以外の JPA プロバイダで動作させる場合は注意してください。

図 6-11　単方向 OneToMany でのデフォルトマッピング



(b)　単方向の ManyToMany リレーションシップ

次に示す条件の場合に適用される単方向の ManyToMany リレーションシップのデフォルトマッピングについて説明します。

条件

- エンティティ A は，エンティティ B のコレクションを参照します。コレクションに @ManyToMany（または，O/R マッピングファイルで該当する XML タグ）を設定します。
- エンティティ B は，エンティティ A を参照しません。
- 所有者はエンティティ A です。

適用されるデフォルトマッピング

- エンティティ A は，テーブル A にマップされます。
- エンティティ B は，テーブル B にマップされます。
- テーブル A，B のほかに，所有者側のテーブルの名前が最初にくる A_B という名前の結合表が必要となります。この結合表は，二つ外部キー列を持ちます。

  外部キー列の一つは，テーブル A を参照し，テーブル A の外部キーと同じ型を持ちます。この外部キー列の名前は，次のようになります。

  外部キー列の名前

  エンティティ A の名前_テーブル A のプライマリキーの名前

  また，もう一つの外部キー列は，テーブル B を参照し，テーブル B のプライマリキーと同じ型を持ちます。外部キー列の名前は，次のようになります。

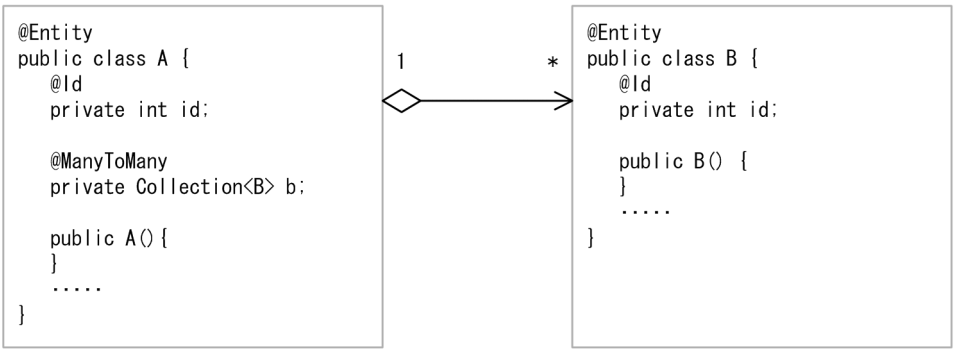
  外部キー列の名前

  エンティティ A のリレーションシップのプロパティ（またはフィールド）の名前_テーブル B のプライマリキーの名前

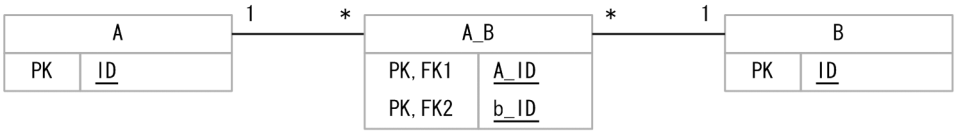
  注　斜体は可変値を表します。

図 6-12　単方向 ManyToMany でのデフォルトマッピング

エンティティ間のリレーションシップ



テーブル間の関連



6.7　エンティティオブジェクトのキャッシュ機能

エンティティオブジェクトのキャッシュ機能とは，アプリケーションで使用したエンティティオブジェクトをメモリ内で保持するための機能です。エンティティオブジェクトのキャッシュ機能を使用している場合に，同じエンティティオブジェクトが操作されると，CJPA プロバイダ内にキャッシュされているエンティティオブジェクトが使用されます。データベースから再度データを読み込むことがないので，データベースへのアクセスが最小限になり，処理性能の負荷を軽減できます。なお，この機能は，CJPA プロバイダ独自の機能です。

ここでは，エンティティオブジェクトのキャッシュ機能について説明します。

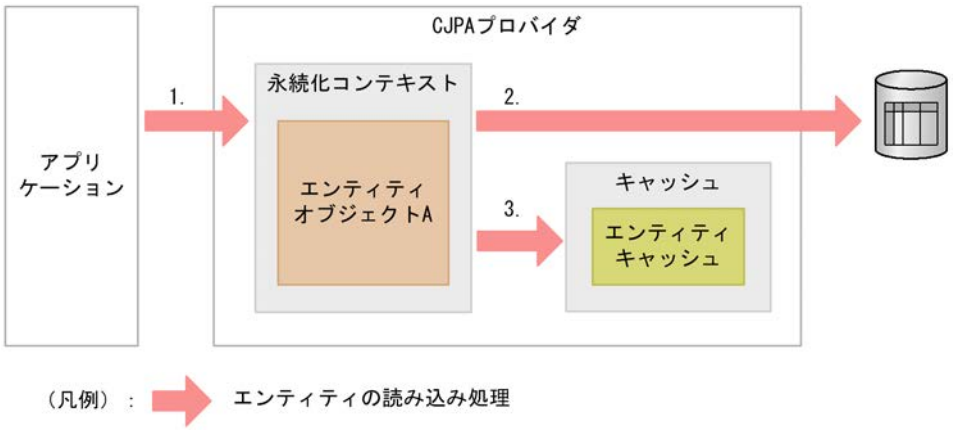
6.7.1　キャッシュ機能の処理

キャッシュ機能を使用している場合，同じエンティティに対して読み込みをすると，データベースではなく，キャッシュからデータが取得されるようになります。キャッシュ機能の処理の流れ，キャッシュの登録および更新のタイミング，およびキャッシュの更新処理の流れを説明します。

(1)　キャッシュ機能の処理の流れ

キャッシュ機能の処理の流れについて次の図に示します。

図 6-13　キャッシュ機能の処理の流れ



上記の図について説明します。

- **エンティティの読み込み処理**

  find などのメソッドで最初にエンティティを読み込むときは次の流れで処理されます。

  1. エンティティの読み込み処理をします。
  2. データベースからデータを取得します。
  3. 取得したデータのエンティティオブジェクトをキャッシュに登録します。

エンティティがリレーションシップを持つ場合でリレーションシップ先のエンティティを取得するとき，目的のエンティティがキャッシュに存在すると，データベースにアクセスしないでキャッシュのエンティティを参照します。

キャッシュは，永続化コンテキスト単位で存在します。

(2)　キャッシュの登録および更新のタイミング

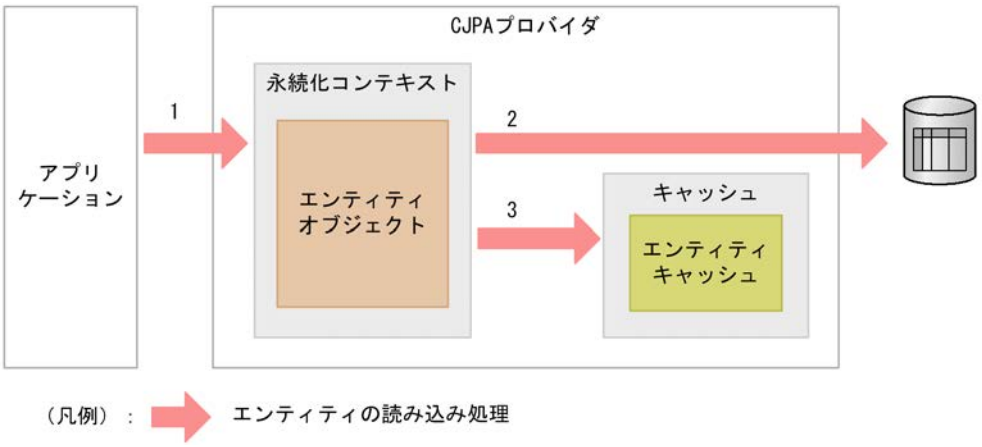
キャッシュへの登録および更新は次に示すタイミングで実施されます。

- キャッシュの登録のタイミング
  対象となるキャッシュが存在しない状態でのエンティティオブジェクトの読み込み時（find 操作）
- キャッシュの更新のタイミング
  - エンティティのリフレッシュ処理時（refresh 操作）
  - トランザクションのコミット時
  - 楽観的ロックの処理で例外が発生した場合
    なお，楽観的ロックの処理で OptimisticLockException 例外が発生した場合には，キャッシュに登録されているエンティティオブジェクトは削除されます。

(3)　キャッシュの更新処理の流れ

キャッシュの更新処理の流れについて次の図に示します。

図 6-14　キャッシュの更新処理



図について説明します。

1. エンティティオブジェクトに対して，次の操作を実施します。
   - refresh 操作
   - トランザクションのコミット
2. refresh 操作の場合は，データベースにアクセスします。
3. 永続化コンテキストにあるデータをキャッシュに登録して，キャッシュのデータを更新します。

なお，JPQL を実行した場合にもキャッシュは登録されます。登録されるタイミングは，対象となるキャッシュが存在しない状態で JPQL を実行した場合です。JPQL でもキャッシュのデータを使用しますが，キャッシュのデータの有無に関係なく，データベースのアクセスが発生します。このため，キャッシュによる処理性能の向上は期待できません。詳細については，(4)を参照してください。

キャッシュは，エンティティのオブジェクト単位で情報を保持しているため，クエリの実行時に返されるオブジェクトがエンティティ自身の場合には，キャッシュへの更新が実行されます。それ以外のフィールドを指定するような場合では，更新はされません。キャッシュの更新が有効になる場合と，ならない場合の JQPL の例を次に示します。

- キャッシュの更新が有効に行われる JPQL の例

```
SELECT emp FROM Employee AS emp
```

- キャッシュの更新がされない JPQL の例

```
SELECT emp.id, emp.name, emp.address FROM Employee AS emp
```

(4)　JPQL とキャッシュの関係

JPQL でエンティティオブジェクト全体を取得するような場合，キャッシュに登録されている情報が存在すればキャッシュの情報を取得します。CJPA プロバイダのキャッシュ機能では，対象となるエンティティを特定するための ID であるプライマリキーが必要になります。プライマリキーを取得するには，JPQL の結果を取得する必要があり，この際にデータベースへのアクセスが発生します。データベースへのアクセスの結果から，プライマリキーを抽出して，キャッシュからエンティティオブジェクトを取得します。また，キャッシュに対象となるデータが存在しない場合は，データベースの情報からエンティティを作成します。

JPQL では，キャッシュのデータの有無に関係なく必ずデータベースへのアクセスが発生します。そのため，JPQL の場合，キャッシュによる性能向上は望めません。

6.7.2　キャッシュの参照形態とキャッシュタイプ

キャッシュの参照形態には次の 3 種類があります。

- 強参照
  GC による回収の対象になりません。
- 弱参照（java.lang.ref.WeakReference）
  弱可到達の場合，GC による回収の対象になります。なお，弱可到達かどうかは，java.lang.ref.WeakReference の仕様に依存します。
  CJPA プロバイダで弱参照のキャッシュが回収対象にならない例を次に示します。
  - 永続化コンテキストに登録されているエンティティオブジェクトのキャッシュ
  - 弱可到達でないキャッシュにリレーションシップで参照されているキャッシュ
  - エンティティの継承戦略を使用している場合，継承関係にある別のエンティティオブジェクトのキャッシュが，弱可到達でないキャッシュ
- ソフト参照（java.lang.ref.SoftReference）
  メモリ残量の低下時にキャッシュアウトする参照形態です。
  リソースの消費率や生存期間に応じて，GC による回収の対象になります。回収されるタイミングや，回収対象となるオブジェクトの選択方法などの仕様は，JavaVM に依存します。

どの形態でキャッシュを参照するかによってキャッシュタイプの種類が異なります。キャッシュの参照形態とキャッシュタイプの対応を次の表に示します。

表 6-15　キャッシュの参照形態とキャッシュタイプの対応

キャッシュの参照形態	キャッシュタイプ
強参照	Full
強参照＋弱参照※1	HardWeak
弱参照＋ソフト参照※1	SoftWeak

キャッシュの参照形態	キャッシュタイプ
弱参照	Weak
なし※2	None

注※1　参照形態を組み合わせます。
注※2　エンティティオブジェクトをキャッシュしません。

CJPA プロバイダでは，キャッシュのタイプを選択できます。アプリケーションの設計や環境によってタイプを選択してください。キャッシュタイプは，persistence.xml で指定します。persistence.xml については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「6.2　persistence.xml」を参照してください。

次にキャッシュタイプについて説明します。

(1)　Full

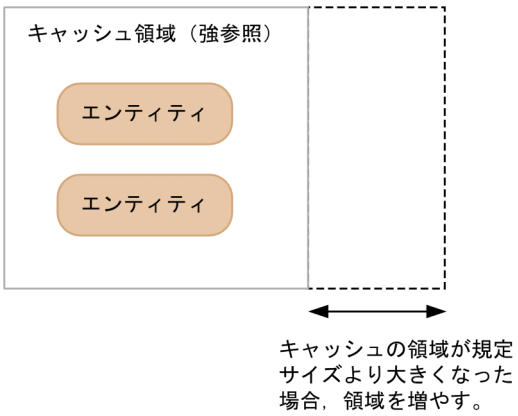
すべてのエンティティを強参照でキャッシュします。

キャッシュタイプに Full を指定した場合，データベースへのアクセスが少なくなるため，処理に対する負荷が少なくなります。ただし，メモリを占有し続けるため，メモリへの負荷が高くなります。

Full は，エンティティオブジェクトの存続期間が長く，頻繁なアクセスを必要とする少数のエンティティオブジェクトで作成される場合に指定します。また，多数のエンティティオブジェクトを読み取る場合は，メモリの負荷が高くなるので，複数レコードを一括更新する場合の使用はお勧めしません。

Full を指定した場合，指定されたキャッシュサイズで強参照領域を確保します。強参照領域が規定のサイズを超えた場合，Hashtable の仕様に基づいて領域を増やします。Full を指定した場合のキャッシュ内のイメージを次の図に示します。

図 6-15　Full の場合のキャッシュ内イメージ



(2)　HardWeak

強参照と弱参照でキャッシュします。

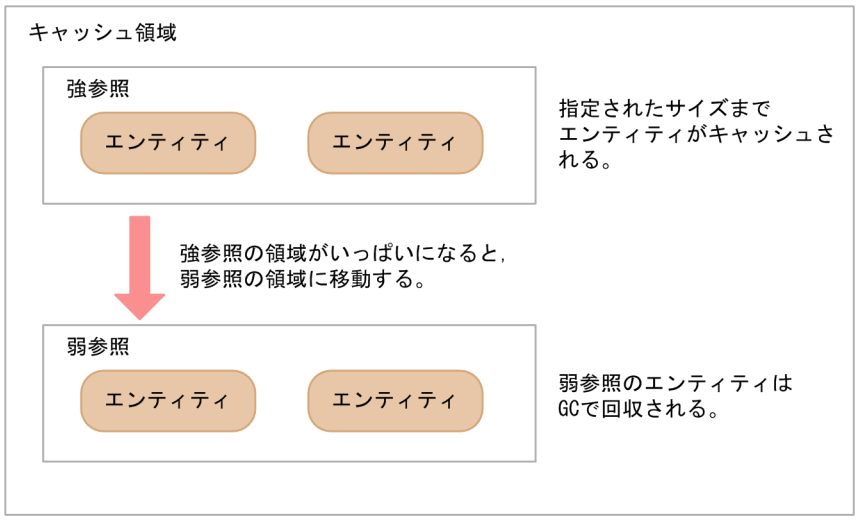
エンティティオブジェクトをリストで保持するときには，強参照を使用します。キャッシュサイズに指定した値だけ，強参照領域を固定長で作成します。キャッシュサイズが指定された値に達すると，古いエンティティオブジェクトを弱参照に移動します。このとき，キャッシュへの登録が最も長く使用されていないエン

ティティオブジェクトから順に弱参照に移動されます。弱参照に移動したエンティティオブジェクトが使用されると，再度，強参照の領域に格納されます。

HardWeak を指定すると，存続期間の長いエンティティオブジェクトを使用して，キャッシュで使用されるメモリを効率良く制御できます。

SoftWeak を使用したシステムでメモリ不足の状態が頻繁に発生する場合には，ソフト参照の利点を生かすことができないため，HardWeak を使用してください。HardWeak の場合のキャッシュ内イメージを次の図に示します。

図 6-16　HardWeak の場合のキャッシュ内イメージ



HardWeak の場合，強参照のキャッシュ領域では強参照でキャッシュを保持します。また，弱参照のキャッシュ領域では弱参照でキャッシュを保持します。

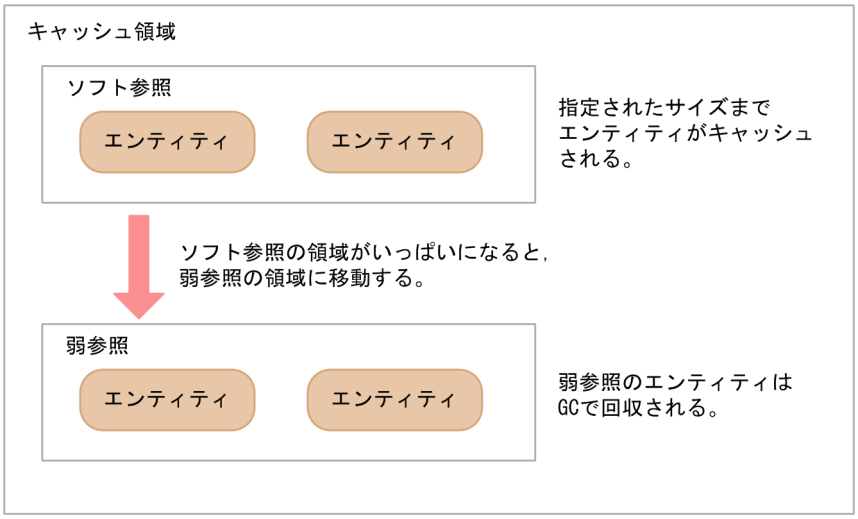
(3)　SoftWeak

ソフト参照と弱参照でキャッシュします。

ソフト参照を使用して，エンティティオブジェクトをリストで保持し，キャッシュサイズに指定した値だけ，ソフト参照領域を固定長で作成します。キャッシュサイズが指定された値に達すると，古いエンティティオブジェクトを弱参照の領域に移動します。このとき，キャッシュへの登録が最も長く使用されていないエンティティオブジェクトから順に弱参照に移動されます。弱参照に移動したエンティティオブジェクトが使用されると，再度，強参照の領域に格納されます。

SoftWeak を指定すると，存続期間の長いエンティティオブジェクトを使用して，キャッシュで使用されるメモリを効率良く制御できます。このため，キャッシュ機能を使用する場合は，SoftWeak を指定することをお勧めします。SoftWeak の場合のキャッシュ内イメージを次の図に示します。

図 6-17　SoftWeak の場合のキャッシュ内イメージ



SoftWeak の場合，ソフト参照のキャッシュ領域ではソフト参照でキャッシュを保持します。また，弱参照のキャッシュ領域では弱参照でキャッシュを保持します。

(4)　Weak

すべてのエンティティを弱参照でキャッシュします。

このため，すべてのエンティティオブジェクトが GC の対象になります。Weak を指定するとメモリへの負荷が少なくなりますが，GC によってキャッシュが削除されるおそれがあります。エンティティオブジェクトのキャッシュ機能を重視しないシステムで使用してください。Weak の場合のキャッシュ内イメージを次の図に示します。

図 6-18　Weak の場合のキャッシュ内イメージ

Weak の場合，弱参照でエンティティをキャッシュします。

(5)　NONE

エンティティオブジェクトはキャッシュされません。エンティティオブジェクトをデータベースから読み取ったあと，すぐに破棄する場合に使用します。

6.7.3　キャッシュ機能の有効範囲

キャッシュのデータは永続化コンテキスト単位で保持されます。このため，別の永続化コンテキストでエンティティオブジェクトが呼び出されていても，キャッシュからデータを取得できません。

なお，キャッシュのデータは，永続化コンテキストが生成されたときから破棄されるまでの間保持されます。

6.7.4　キャッシュ機能を使用するときの注意事項

ここでは，エンティティオブジェクトのキャッシュ機能を使用するときの注意事項を説明します。

(1)　クエリでデータを更新または削除した場合の注意

アプリケーション内で JPQL やネイティブクエリを使用して，データを更新したり，削除したりした場合，キャッシュの内容は更新されません。refresh 操作などを行って，データベースの内容を取得し直してください。

次に示す操作の場合，キャッシュのデータを読み込むため，データベースへの更新がされません。

1. データをエンティティオブジェクトに読み込む。
   キャッシュにもエンティティのデータが登録されます。
2. 1.で読み込んだデータを含む削除クエリを実行する。
   削除クエリの実行でデータは削除されますが，キャッシュは削除されません。
3. 1.と同じデータをエンティティオブジェクトに読み込む。
   1.と同じデータのため，キャッシュにあるデータを読み込みます。
4. 3.のデータを flush する。
   データベースには該当する行がないため，追加または更新処理ができません。

(2)　キャッシュを使用する永続化コンテキストが複数ある場合の注意

キャッシュを使用することで，データベースへのアクセス頻度を下げることができます。ただし，一方でキャッシュによるデータのタイムラグが発生して，楽観的ロック例外の発生頻度が高くなるおそれもあります。

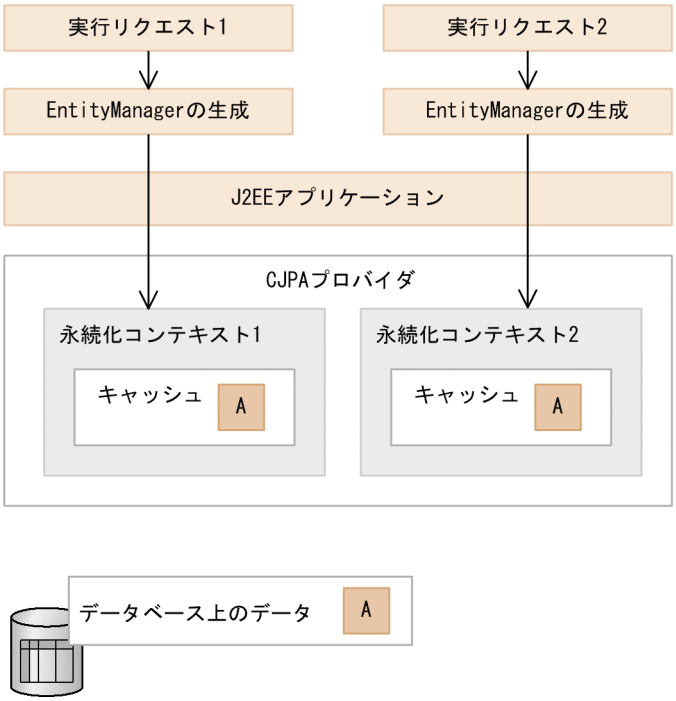
キャッシュは永続化コンテキスト単位に存在します。そのため，永続化コンテキストと対で存在する EntityManager が複数かつ同時期に生成され，同一プライマリキーのエンティティを同時に操作すると，一方でデータを更新しても他方でタイミング良く，更新後のデータを参照できないことがあります。これによって，楽観的ロック例外が発生しやすくなります。

次に，楽観的ロック例外が発生する仕組みと対処方法について説明します。

(a)　楽観的ロック例外が発生する例

ここでは，次の図に示す環境で，永続化コンテキスト単位にキャッシュが存在する場合を例に説明します。

図 6-19　この例で説明する環境

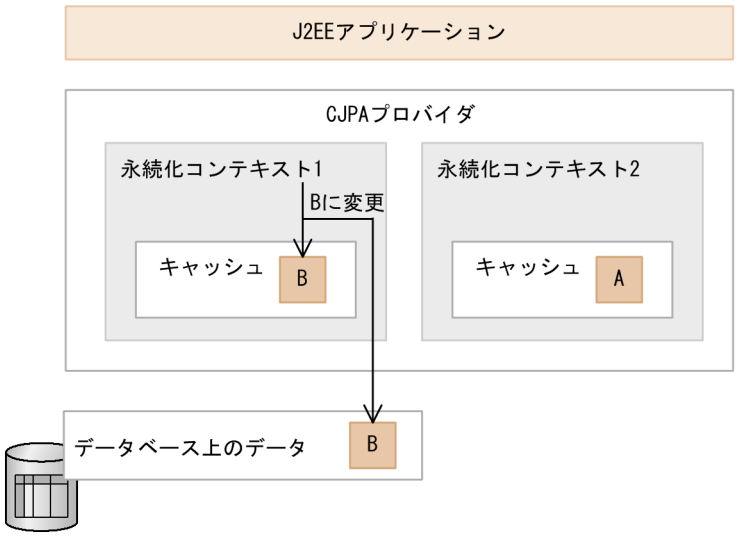


図の状態では，キャッシュとデータベースには不整合はなく，すでにキャッシュに A というデータが格納されているものとします。

1. 永続化コンテキスト 1 でデータを A から B に変更します。

   このとき，キャッシュとデータベースの内容が等しいため例外は発生しません。

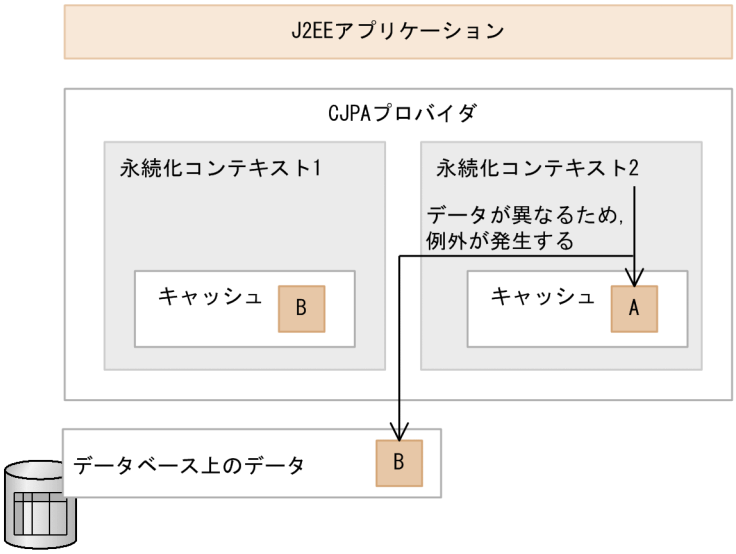
   図 6-20　永続化コンテキスト 1 でのデータの変更



2. 1.の処理が終了後，永続化コンテキスト 2 で A のデータを変更します。

   キャッシュのデータは変更されていないため，データベースのデータとキャッシュに不整合があります。このため，楽観的ロックによる例外が発生します。

図 6-21　永続化コンテキスト 2 でのデータの変更



このような環境の場合，キャッシュを使用すると，更新されないキャッシュが残ってしまい，楽観的ロックによる例外が発生することがあります。

(b)　対処方法

楽観的ロック例外が発生した場合は，キャッシュ内の該当オブジェクトは削除されます。このため，find メソッドを実行するか，または refresh メソッドを実行して関連するすべてのデータを再度データベースから取得してください。これによって，キャッシュのデータとデータベースを同期できます。

(3)　キャッシュの登録および更新タイミングに関する注意事項

- JPQL を使用して悲観的ロックを取得する場合，クエリの実行時に返されるエンティティオブジェクトはキャッシュに登録されません。
- ネイティブクエリを利用して，@SqlResultSetMapping でエンティティを戻り値とするクエリを実行した場合，戻り値のエンティティオブジェクトはキャッシュに格納されます。
- エンティティの refresh 操作で，エンティティオブジェクトの読み込み後にほかのスレッドで OptimisticLockException が発生してキャッシュが削除されると，リフレッシュ処理を実行してもキャッシュにエンティティオブジェクトは登録されません。

6.8　プライマリキー値の自動採番

プライマリキー採番機能とは，エンティティオブジェクトを使用してレコードを挿入する際に，プライマリキー値を自動で生成する機能です。この機能によって，ユーザがプライマリキー値を指定しなくても一意の値が格納されるようになります。CJPA プロバイダでは，プライマリキー採番機能を提供しています。

プライマリキー値の生成方法

プライマリキー値の生成方法には次の 4 種類があります。

- TABLE

  プライマリキー値を保存しておくためのテーブルを使用して，プライマリキー値を生成する方法です。

- SEQUENCE

  データベースのシーケンスオブジェクトを使用して，プライマリキー値を生成する方法です。ただし，データベースに HiRDB を使用している場合，CJPA プロバイダでは TABLE と同じ処理を実施します。

- IDENTITY

  データベースの identity 列を利用してプライマリキー値を生成する方法です。ただし，CJPA プロバイダでは，使用しているデータベースの種類によって動作が異なります。

  HiRDB の場合，TABLE と同じ処理を実施します。

  Oracle の場合，SEQUENCE と同じ処理を実施します。

- AUTO

  使用しているデータベースに適した生成方法を選択します。CJPA プロバイダでは，HiRDB の場合，Oracle の場合ともに TABLE を選択します。

プライマリキー値が採番されるタイミング

CJPA プロバイダの場合，flush 操作またはトランザクションのコミットのタイミングでプライマリキー値が採番されます。

プライマリキー値の生成方法が SEQUENCE の場合の例

プライマリキー値の生成方法が SEQUENCE の場合の例を示します。この例では，事前に EMP_SEQ という名称のシーケンスオブジェクトが作成されているものとします。

```
@Entity
public class Employee {
 . . .
    @SequenceGenerator(
        name="EMPLOYEE_GENERATOR",
        sequenceName="EMP_SEQ"

    )
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE,
 generator="EMPLOYEE_GENERATOR")
    @Column(name="EMPLOYEE_ID")
    public Integer getId() {
        return id;
    }
 . . .
}
```

6.9　クエリ言語によるデータベース操作

JPA を使用してデータベースのデータを操作するには, javax.persistence.Query インタフェースを通して実施します。javax.persistence.Query インタフェースを使用すると，複数のレコードに対してまとめて，検索，更新，削除などの操作ができます。javax.persistence.Query インタフェースを使用するには，ユーザはクエリ言語を使用してデータベースを操作します。

CJPA プロバイダでは，クエリ言語として JPQL および SQL を使用できます。JPQL および SQL についてそれぞれ説明します。

- JPQL
  JPA 仕様で定義されているクエリ言語です。データベースに依存しない言語で，エンティティクラスを対象に操作します。
- SQL
  データベースに依存したクエリ言語です。ネイティブクエリともいいます。データベースのデータを対象に操作します。

JPQL や SQL によるデータベース操作については,「6.16　クエリ言語を利用したデータベースの参照および更新方法」を参照してください。また，JPQL の文法については,「6.17　JPQL の記述方法」を参照してください。

6.10　楽観的ロック

CJPA プロバイダでは，EntityManager と永続化コンテキストを利用して永続化対象のエンティティを管理します。変更されたエンティティの情報は，flush メソッドまたはトランザクションのコミット処理が実行されたときにデータベースに反映されます。複数のトランザクションで同時にデータベースの同じ行を更新するおそれがある場合，データの整合性を保証する必要があります。データの整合性を保証するために，CJPA プロバイダでは楽観的ロックを提供しています。

楽観的ロックとは，データの更新処理を始めてから更新が完了するまでの間に，ほかから更新されていないことを確認するためのロック機能です。楽観的ロックはデータベースをロックしないので，デッドロックなどが発生しないという利点があります。

ここでは，楽観的ロックについて説明します。

6.10.1　楽観的ロックの処理

楽観的ロックを使用すると，CJPA プロバイダはデータベースのデータがほかのアプリケーションから更新されていないかをユーザに代わってチェックします。データベースのデータが更新されていると，CJPA プロバイダは例外を発生させて，ユーザにデータが更新されていることを通知します。また，トランザクションをロールバックにマークします。

(1)　データ更新の有無のチェック方法

データが更新されているかどうかは，データベースのテーブル上に用意したバージョン列の更新の有無によってチェックします。データベース上のデータが更新されると，バージョン列のバージョン番号が更新されます。これによって，ほかのアプリケーションなどからデータベースが更新されたことがわかります。データベースを更新するときのバージョン列の状態と動作について次の表に示します。

表 6-16　データベースを更新するときのバージョン列の状態と動作

テーブルのバージョン列の状態	動作
バージョン列の値が更新されていない場合	CJPA プロバイダは，エンティティの情報をデータベースに反映します。このとき，データベースのバージョン列の値を更新します。
バージョン列の値が更新されている場合	ほかのアプリケーションなどからデータベースのデータが更新されていることを表します。このため，CJPA プロバイダは OptimisticLockException を発生させて，トランザクションをロールバックにマークします。

このように，バージョン列の状態によって，エンティティを読み込んだ状態からデータベースを更新するまでに，ほかのトランザクションがデータを更新していないことを保証できます。

(2)　永続化フィールドおよびリレーションシップのバージョンチェック

楽観的ロックを使用するには，エンティティの永続化フィールドおよびリレーションシップの両方をバージョンチェックの対象にします。バージョンチェックの対象とするには，エンティティにバージョン列に対応する Version フィールド（プロパティ）を設定してください。Version フィールドは@Version または O/R マッピングファイルの<version>タグを使用して設定します。

エンティティのバージョンチェックは次のどちらかのタイミングで実行されます。

- エンティティの状態が変更され，その変更をデータベースに書き込むとき
- merge 処理によって，エンティティが managed 状態に変更されたとき※

注※

バージョンチェックは merge 実行時だけでなく，flush またはトランザクションのコミット時にもチェックが実行されます。

バージョンチェックによってエンティティのバージョンが古いことが判明した場合，OptimisticLockException が発生します。また，トランザクションはロールバックにマークされます。

(3) flush 操作またはトランザクションの決着時のバージョンチェック

エンティティを flush 操作またはトランザクションの決着時にバージョンチェックの対象にできます。バージョンチェックの対象とするには，EntityManager の lock メソッドにエンティティを指定します。EntityManager の lock メソッドを使用することで，トランザクションでのバージョンチェック対象にエンティティを追加したり，バージョン列の更新方針を変更したりできます。

CJPA プロバイダでは，バージョン列の更新タイミング（lock メソッドの LockModeType）として LockModeType.READ および LockModeType.WRITE の二つをサポートしています。バージョン列の更新タイミングの指定内容にかかわらず，CJPA プロバイダではトランザクションの決着時に次に示す二つの事象が起きないことを保証します。

- ダーティリード（Dirty Read）

  トランザクション「T1」が行を変更します。次に，「T1」がコミットやロールバックを行う前に別のトランザクション「T2」が同じ行を読み込み，変更した値を取得します。最終的に「T2」がコミットに成功します。

  「T1」がコミットかロールバックをするかどうかが重要ではなく，「T2」のコミットの前またはあとのどちらで「T1」のコミットかロールバックが行われるかが重要になります。

- 繰り返し不可能な読み込み（un-Repeatable Read）

  トランザクション「T1」が行を読み込みます。次に，「T1」がコミットする前に，もう一方のトランザクション「T2」がその行の変更や削除を実施します。最終的に両方のトランザクションはコミットに成功します。

LockModeType として LockModeType.WRITE を指定すると，エンティティの状態変更がない場合でもバージョン列は強制的に更新されます。バージョン列の更新は flush またはトランザクションのコミットが呼び出されたタイミングで実行されます。なお，バージョン列の更新前にエンティティが削除された場合，バージョン列の更新は省略されることもあります。

6.10.2 楽観的ロックに失敗した場合の例外処理

明示的に楽観的ロックの実行を確認したい場合は，flush()メソッドを呼び出します。flush()メソッドの呼び出しで楽観的ロック例外が発生した場合，例外処理をキャッチしてリカバリ処理を行うことができます。楽観的ロックに問題がない場合，flush()メソッドによってエンティティのバージョンのチェックおよび Version 列が更新されます。

例外処理の記述例を次に示します。

```
try{
  em.flush();
} catch (OptimisticLockException e){
  // 例外処理
}
```

この例に示すように，例外処理の中で OptimisticLockException をラップすることで楽観的ロックの例外をアプリケーション例外として見せることができます。ただし，この場合のトランザクションはロールバックにマークされているので，コミットできないことに注意してください。

なお，CJPA プロバイダでは，例外の原因となったエンティティを OptimisticLockException に格納しません。OptimisticLockException の getEntity()メソッドは常に null 値を返します。

6.10.3　楽観的ロックを使用する際の注意事項

ここでは，楽観的ロックを使用する際の注意事項について説明します。

(1)　Version フィールドの設定時の注意事項

Version フィールドの設定時の注意事項を次に示します。

- Version フィールドが設定されていない場合，そのエンティティに対するバージョンチェックは実施しません。その場合，エンティティとデータベースの間の整合性を保つようなアプリケーションをユーザ責任で作成してください。
- トランザクションに Version フィールドが設定されているエンティティと設定されていないエンティティが含まれている場合，Version フィールドが設定されているエンティティに対してだけバージョンチェックが実行されます。なお，エンティティにバージョンが含まれていないことはトランザクションの決着処理には影響ありません。
- ユーザは Version フィールドの値を参照できますが，更新はしないでください。ただし，バルク更新処理では Version フィールドの値を更新できます。

(2)　lock メソッド使用時の注意

lock メソッド使用時の注意事項を次に示します。

- Version フィールド（プロパティ）を持たないエンティティに対する EntityManager の lock メソッドはサポートしていません。Version フィールド（プロパティ）を持たないエンティティを指定して，lock(entity, LockModeType.READ)または lock(entity, LockModeType.WRITE)が呼ばれた場合は，PersistenceException が発生します。
- Version フィールド（プロパティ）を持つエンティティの状態が更新される場合は，lock メソッドの呼び出し有無にかかわらず，ダーティリード，および繰り返し不可能な読み込みの現象は発生しません。
- CJPA プロバイダでは LockModeType.READ が指定された場合，エンティティに該当するデータベースの値に変更がないかを確認するために，UPDATE 文を発行します。このため，データベースに対して UPDATE 文による排他が掛かります。UPDATE 文は，flush 処理の実行時およびトランザクションのコミット時に発行されます。エンティティオブジェクトの状態に変更がない場合でも発行されます。ただし，エンティティが削除された場合は発行されません。

(3)　HiRDB でのクライアント単位の排他制御について

CJPA プロバイダの楽観的ロックは，データベースの Isolation レベルが Read Committed でアクセスされることを想定したロック方式です。データベースが HiRDB の場合，Isolation レベルがデフォルトの設定では Repeatable Read であるため，Read Committed に変更する必要があります。

クライアント単位の Isolation レベルは，クライアント環境変数のデータ保証レベルの PDISLLVL パラメタで設定します。デフォルトの値は Repeatable Read (2)です。このため，Read Committed (1) に変更してください。変更例を次に示します。

変更例：PDISLLVL=1

クライアント環境変数は Connector 属性ファイルの<config-property>タグで environmentVariables プロパティの値に指定するか，または HiRDB のクライアント環境変数グループの設定ファイルに追加してください。

クライアント環境変数のデータ保証レベルをデフォルト設定の Repeatable Read で動作させた場合は，共有モードで排他が掛かります。このため，find メソッドなどの参照系 SQL の発行と flush メソッドなどの更新系 SQL の発行を組み合わせることによって，デッドロックが発生しやすくなるので注意してください。

6.11　JPQL での悲観的ロック

悲観的ロックとは，複数のトランザクションによってデータベース上の同じレコードを更新するときに，対象となるレコードを占有ロックする方法です。あるトランザクションがあるレコードに対して悲観的ロックを掛けると，ほかのトランザクションはそのレコードを参照または更新できません（ただし，Oracle の場合，参照はできます）。悲観的ロックは JPQL を使用しているときだけ使用できます。

悲観的ロックを使用すると，ロックを取得したトランザクションが終了するまでロックの解放待ちが発生します。このため，楽観的ロックよりも同時実行性はありませんが，楽観的ロックで発生するトランザクションのコミット時のエラーを回避できます。

悲観的ロックの指定方法

悲観的ロックは，CJPA プロバイダがサポートするクエリヒントを利用することで実現します。悲観的ロックは，Query メソッドの setHint()メソッドまたは@NamedQuery の属性に@QueryHint を指定して実行します。

悲観的ロック機能の実装例

悲観的ロック機能の実装例を次に示します。

実装例 1

Query メソッドの setHint()メソッドで悲観的ロックを指定する場合の例を次に示します。

```
Query query = manager.createQuery("SELECT emp FROM  Employee  AS emp");
query.setHint("cosminexus.jpa.pessimistic-lock","Lock");
```

実装例 2

@NamedQuery の属性の@QueryHint で悲観的ロックを指定する場合の例を次に示します。

```
@NamedQuery(
name="employee_list",
query="SELECT emp FROM  Employee AS emp",
hints={ @QueryHint(name="cosminexus.jpa.pessimistic-lock", value="Lock") }
)
@Entity
public class Employee{
 ・・・
}
```

! 注意事項

悲観的ロックが使用できるのは JPQL の時だけです。createNativeQuery メソッドや @NamedNativeQuery の hints 属性に@QueryHint を指定しても有効にはなりません。また，O/R マッピングファイルの<named-native-query>タグの hint 属性に指定した場合も有効にはなりません。なお，CJPA プロバイダでの悲観的ロックの排他仕様は，使用するデータベースの仕様に準じます。

6.12　エンティティクラスの作成

JPA を利用したアプリケーションを作成するには，アプリケーションでエンティティクラスを定義します。エンティティクラスは，データベースのテーブルのレコードを Java のオブジェクトとして扱うためのクラスです。ユーザがアプリケーションで new オペレーションを行うと，エンティティクラスのインスタンスが生成されます。

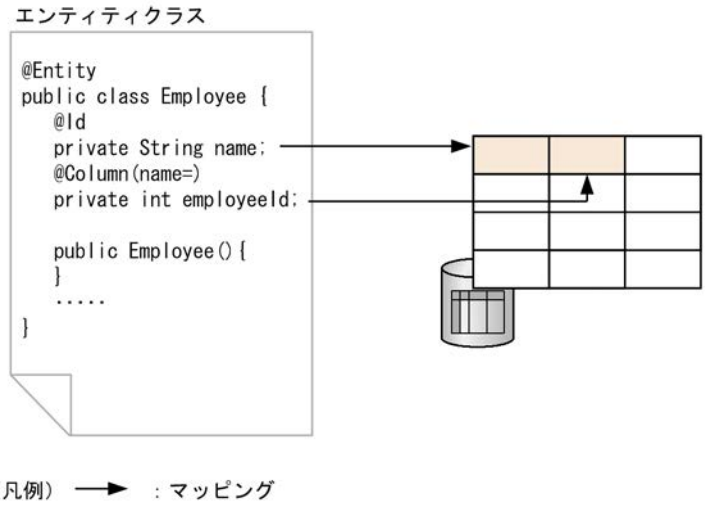
ここでは，JPA アプリケーションの作成について説明します。

6.12.1　エンティティクラスとデータベースの対応の定義

エンティティクラスは，データベースのテーブルの行に対応します。また，エンティティクラスが保持するフィールドは，テーブルのカラムの値に対応します。ユーザがエンティティクラスのインスタンスに対してフィールドの値を更新すると，CJPA プロバイダによって対応するデータベースのテーブルのカラムも更新されます。そのため，ユーザはデータベースに対して SQL を発行することなく，データベースの状態を変更できます。

エンティティクラスとデータベースのテーブルとのマッピングについて次の図に示します。

図 6-22　エンティティクラスとデータベースのテーブルとのマッピング



エンティティのフィールドとデータベースのカラムの対応関係は，アノテーションまたは O/R マッピングファイルで定義します。アノテーションと O/R マッピングファイルのどちらでも定義はできます。ただし，両方で定義している場合は，O/R マッピングファイルの設定がアノテーションよりも優先されます。アノテーションと O/R マッピングファイルとで同じ設定項目に対して異なる値を設定している場合は，アノテーションの値を O/R マッピングファイルの値で上書きします。

6.12.2　エンティティクラスの作成要件

JPA を使用したアプリケーションを作成する場合には，JPA 仕様で決められたエンティティクラスの作成要件やデータベースのマッピング要件を守る必要があります。作成要件を次に示します。

- エンティティクラスは@Entity または O/R マッピングファイルの<entity>タグに指定する必要があります。
- エンティティクラスは引数なしのコンストラクタを持ちます。

- enum やインタフェースはエンティティクラスとしてはいけません。
- エンティティクラスのクラス階層のルートとなるエンティティまたはマップドスーパークラスでは，プライマリキーを持つ必要があります。プライマリキーは，エンティティ階層で必ず一つ定義してください。
- エンティティクラスのインスタンスを値渡しでメソッドの引数として渡す場合，Serializable インタフェースを実装する必要があります。
- データベースのカラムの状態はエンティティのインスタンス変数で表現し，インスタンス変数は JavaBean のプロパティに対応します。なお，インスタンス変数は，クライアントから直接アクセスして値を変更してはいけません。アクセサメソッド（getter／setter メソッド）や，ビジネスメソッド経由で値を変更してください。
- エンティティの永続インスタンス変数は，private，protected，または package から参照できるアクセスレベルにしてください。
- 引数なしのコンストラクタは，public か protected で宣言してください。
- エンティティクラスは final にしないでください。また，エンティティクラスの永続化インスタンス変数とすべてのメソッドも final にしてはいけません。

CJPA プロバイダでは，これらの条件に合わない場合，例外が発生するおそれがあります。なお，例外が発生しない場合でもこれらの条件に合わないエンティティクラスを作成している場合の動作は保証しません。

また，CJPA プロバイダの場合，エンティティクラスでは，データベースの一つのカラムを複数のフィールドに対応づけないでください。この条件に合わない場合，アプリケーション実行時に例外が発生することがあります。例外が発生しない場合でも，動作は保証しません。

6.12.3　エンティティクラスのフィールドに対するアクセス方法の指定

CJPA プロバイダは，エンティティの状態をデータベースに書き込んだり，データベースの状態をエンティティとして読み込んだりするときに，エンティティクラスのフィールドにアクセスします。このときのアクセス方法をアクセスタイプと呼びます。アクセスタイプには，プロパティとフィールドの 2 種類があります。また，アクセスタイプはアノテーションまたは O/R マッピングファイルで指定します。アクセスタイプと指定方法について次の表に示します。

表 6-17　アクセスタイプと指定方法

アクセスタイプ	説明	指定方法	
		アノテーション	O/R マッピングファイル
プロパティ	getter メソッドを経由して取得する方法です。	フィールドの getter メソッドにアノテーションを指定します。	<access>タグに PROPERTY を指定します。
フィールド	インスタンス変数を直接参照する方法です。	フィールドにアノテーションを指定します。	<access>タグに FIELD を指定します。

アクセスタイプがプロパティの場合，エンティティが保持するプロパティを**永続化プロパティ**といいます。また，アクセスタイプがフィールドの場合，エンティティのフィールドを**永続化フィールド**といいます。

アクセスタイプに関連する注意事項

アクセスタイプを指定するときには，次の点を考慮してください。

- アクセスタイプがフィールドの場合，CJPA プロバイダは永続化フィールドに直接アクセスします。@Transient を設定されていないインスタンス変数は，永続化の対象となります。

- アクセスタイプがプロパティの場合, CJPA プロバイダはアクセサメソッドを利用して永続化プロパティの値を取得します。@Transient がアクセサメソッドに設定されていないプロパティは, 永続化の対象となります。
- アクセスタイプがプロパティの場合，setter メソッドにマッピングアノテーションは設定できません。CJPA プロバイダの場合，setter メソッドに設定したマッピングアノテーションは無視されます。
- @Transient または O/R マッピングファイルで<transient>タグが付与されたフィールドとプロパティには，マッピングアノテーションは設定できません。設定した場合には，アプリケーションの開始時に例外が発生します。
- プロパティのアクセサメソッドは public または protected にしてください。これは, JPA 仕様で禁止されていますが，アプリケーションサーバの JPA 機能ではチェックしません。また，private であっても例外も発生しません。
- 永続化フィールドと永続化プロパティのアクセサメソッドの両方にマッピングアノテーションが適用された場合，永続化プロパティのアクセサメソッドに設定されたアノテーションはすべて無視されます。

6.12.4 アクセサメソッドの作成

ここでは，アクセサメソッドのシグネチャ規則，およびアクセサメソッドへのビジネスロジックの追加について説明します。

(1) アクセサメソッドのメソッドシグネチャ規則

CJPA プロバイダが永続化プロパティに対してアクセスする場合，プロパティのアクセサメソッドは，次に示す JavaBeans と同じメソッドシグネチャ規則に従っている必要があります。

- T getProperty()
- void setProperty(T t)

boolean の戻り値を返すプロパティの場合，getter メソッドは isProperty という名前にすることもできます。なお, CJPA プロバイダを使用する場合, getter メソッドと setter メソッドのどちらかしかないときには，アプリケーション開始時に例外が発生します。

また，永続化フィールド，永続化プロパティでコレクション値を扱う場合は，次に示すコレクション値をインタフェースで定義してください。

- java.util.Collection
- java.util.Set
- java.util.List
- java.util.Map

永続化プロパティがコレクション値となる場合, アクセサメソッドのメソッドシグネチャはこれらのインタフェースのどれかにしてください。または, これらのコレクションのジェネリックな型を使用することもできます（例：Set<T>）。

(2)　アクセサメソッドへのビジネスロジックの追加

アクセサメソッドは，プロパティの setter/getter 処理に加えて，値を検証するなどのビジネスロジックを含むことができます。アクセスタイプがプロパティの場合，CJPA プロバイダがアクセサメソッドを呼び出すタイミングでビジネスロジックが動作します。

ただし，この場合には，次の点に注意してください。

- CJPA プロバイダの実行時に，永続状態をロードして格納するアクセサメソッドが呼び出される順番は定義されていません。このため，getter に含まれるロジックの実行順序は未定となります。
- アクセスタイプがプロパティで，永続化プロパティで lazy フェッチが指定されている場合で，移植性を求めるには，エンティティの内容が CJPA プロバイダにフェッチされるまで，エンティティの内容にアクセスしないことをお勧めします。
- アクセスタイプがプロパティの場合で，ビジネスロジックとして値を変更するロジックを追加したときには，CJPA プロバイダではデータの一貫性は保証しません。

プロパティアクセサメソッドで Runtime 例外が発生すると，現在のトランザクションはロールバックにマークされます。CJPA プロバイダがエンティティの永続状態の内容を読み込んだり，格納する際のアクセサメソッドで例外が発生したりした場合，トランザクションはロールバックされます。また，アプリケーション例外をラップする PersistenceException 例外が発生します。

6.12.5　エンティティの永続化フィールドおよび永続プロパティの型

エンティティの永続化フィールド／永続化プロパティは，次に示す型にしてください。

- Java のプリミティブ型
- java.lang.String
- ほかのシリアライズ型
- プリミティブ型のラッパークラス
- java.math.BigInteger
- java.math.BigDecimal
- java.util.Date
- java.util.Calendar
- java.sql.Date
- java.sql.Time
- java.sql.Timestamp
- ユーザが定義するシリアライズな型
- byte[]
- Byte[]
- char[]
- Character[]
- enum
- エンティティタイプとエンティティタイプで複製できるコレクション
- 埋め込みできるクラス

CJPA プロバイダを使用する場合は，上記以外の型が指定されたときの動作は保証しません。アプリケーションの実行時に例外が発生するおそれもあります。

ただし，CJPA プロバイダを使用している場合，エンティティのインスタンス変数の型は，データベースのカラムの型と対応づけしている必要があります。Java の型とデータベースの型の対応づけは，JDBC ドライバが行います。CJPA プロバイダでは，Oracle 接続時には Oracle が提供する Oracle JDBC Thin Driver を JDBC ドライバとして利用します。HiRDB 接続時には HiRDB Type4 JDBC Driver を JDBC ドライバとして利用します。ユーザは JDBC ドライバがサポートする型の変換に合わせてエンティティのインスタンス変数の型を決定してください。

6.12.6 エンティティでのプライマリキーの指定

エンティティでは, プライマリキーをクラス階層の中で必ず指定してください。プライマリキーを指定する場合には，次に示すルールに従ってください。

- 単体の（複合型でない）プライマリキーは，永続化フィールドまたは永続化プロパティに@Id を指定するか，O/R マッピングファイルで指定します。これによって，エンティティのフィールドとの対応づけをしてください。
- 複合型のプライマリキーの場合, @EmbeddedId として単一のフィールドでプライマリキークラスを指定するか，または@IdClass と@Id によって，フィールドのセットとして複合プライマリキーを指定します。
- 複合型のプライマリキーの場合，プライマリキークラスと呼ばれるプライマリキーを含むクラスを作成します。

これらの条件に従っていない場合は，アプリケーション開始時に例外が発生します。

また, アプリケーションでエンティティのプライマリキーの値を変更しないでください。アプリケーションでプライマリキーの値を変更した場合，実行時に例外が発生します。

(1) プライマリキーの型

単体または複合型のプライマリキーは，次に示す型のどれかにしてください。

- Java のプリミティブ型
- プリミティブをラッパした型
- java.lang.String
- java.util.Date
- java.sql.Date

なお，近似値型（例えば浮動小数点数型）をプライマリキーとして指定すると，CJPA プロバイダの場合, 丸め誤差が発生したり, equals メソッドの結果に信頼性がないという問題が発生したりします。このため, CJPA プロバイダではプライマリキーに近似値型を使用した場合の動作は保証しません。java.util.Date がプライマリキーとしてフィールド／プロパティで使用される場合，temporal type 属性は DATE 型として指定する必要があります。

(2) 複合型のプライマリキー

エンティティでは複合型のプライマリキーを扱うことができます。エンティティで複合型のプライマリキーを指定するには，埋め込み型クラスを利用する方法と@IdClass を利用する方法の 2 とおりがあります。それぞれの方法を説明します。

(a)　埋め込み型クラスを利用する方法

埋め込み型クラスを利用するには，@Embeddable を付与したクラスを作成し，そのクラスのフィールドとして複合型のプライマリキーを定義します。エンティティクラスでは，@Embeddable を付与したクラスの型のフィールドを定義して，@EmbeddedId をアノテートします。エンティティクラスの例と埋め込み型クラスの例を次に示します。

- エンティティクラスの例

```
@Entity
public class Employee {
    private EmployeePK employeePK;

    public Employee(){
    }

    @EmbeddedId
    public EmployeePK getEmployeePK(){
      retrun this.employeePK;
    }

    public void setEmployeePK(EmployeePK employeePK){
      this.employeePK = employeePK;
    }
     . . .
}
```

- 埋め込み型クラスの例

```
@Embeddable
public class EmployeePK {
    private String name;
    private int employeeId;

    public EmployeePK(){
    }

    public boolean equals(Object obj){
       . . .
    }

    public int hashCode(){
       . . .
    }
     . . .
}
```

埋め込み型クラスについては「(3)　埋め込み型クラス」を参照してください。なお，アノテーションの代わりに O/R マッピングファイルを利用することもできます。

(b)　@IdClass を利用する方法

@IdClass を利用するには，エンティティクラスでプライマリキーに対応する複数のインスタンス変数を定義して@Id を付与します。また，@IdClass を使用してプライマリキークラスを指定します。プライマリキークラスでは，エンティティで定義したプライマリキーと同じ名前と型を持つフィールドまたはプロパティを定義します。エンティティクラスの例とプライマリキークラスの例を次に示します。

- エンティティクラスの例

```
@Entity
@IdClass(EmployeePK.class)
public class Employee {
    private String name;
    private int employeeId;

    public Employee(){
    }

    @Id
    public String getName(){
      retrun this.name;
```

```
    }

    public void setName(String name){
      this.name = name;
    }

    @Id
    public int getEmployeeId(){
      retrun this.employeeId;
    }

    public void setName(int employeeId){
      this.employeeId = employeeId;
    }
     ・・・
}
```

- プライマリキークラスの例

```
public class EmployeePK implements Serializable {
    private String name;
    private int employeeId;

    public EmployeePK(){
    }

    public boolean equals(Object obj){
      ・・・
    }

    public int hashCode(){
      ・・・
    }
     ・・・
}
```

埋め込み型クラスの場合と同様に，アノテーションの代わりに O/R マッピングファイルを利用することもできます。なお，プライマリキークラスのアクセスタイプは，プライマリキーに対応するエンティティクラスのアクセスタイプで決定します。

複合型のプライマリキーを扱うためには，埋め込み型クラスまたは@IdClass のどちらかを使用します。ただし，次に示すルールに従ってください。

- プライマリキークラスは，public で引数のないコンストラクタを持たなければならなりません。
- 永続化プロパティを使用する場合，プライマリキークラスのプロパティは public または protected にしてください。
- プライマリキークラスは，直列化可能にしてください。
- プライマリキークラスでは equals と hashCode メソッドを定義します。マップされているデータベース上で主キーが等しい場合には equals で true を返し，hashCode の値を等しくする必要があります。
- 複合プライマリキーは，埋め込みクラスとしてマップされているか，エンティティクラスの複数のフィールド／プロパティをマップされている必要があります。
- プライマリキークラスがエンティティクラスの複合フィールド／プロパティにマップされる場合，プライマリキークラスのプライマリキーのフィールド／プロパティの名前と，エンティティクラスの名前を一致させてください。また，型も同じにしてください。

CJPA プロバイダでは，これらの条件を満たさない場合動作は保証しません。また，条件を満たさない場合はアプリケーションの開始時に例外が発生することもあります。

(3)　埋め込み型クラス

永続化対象の幾つかのフィールドをまとめたクラスを用意すると，エンティティのフィールドとして保持できます。このようなクラスを埋め込み型クラスといいます。

埋め込み型クラスは，エンティティに埋め込まれてエンティティと同じデータベースのテーブルにマップされます。このため，エンティティとは異なり，プライマリキーを持ちません。

ユーザが埋め込み型クラスを利用する場合，埋め込むクラスには@Embeddable を設定します。また，埋め込まれる側のエンティティクラスでは，埋め込み先のフィールド，プロパティに@Embedded を指定します。なお，アノテーションの代わりに O/R マッピングファイルで同様に定義することもできます。

埋め込み型クラスは複合型のプライマリキーを定義するために利用することもできます。この場合，埋め込まれるエンティティクラスでは，@Embedded の代わりに@EmbeddedId を設定します。

埋め込み型クラスでは，次に示す作成要件を必ず守ってください。

1. 埋め込み型クラスは，必ず@Embeddable で定義するか，O/R マッピングファイルの<embeddable>タグで定義してください。
2. enum やインタフェースを埋め込み型クラスとしないでください。
3. 埋め込み型クラスを含むエンティティクラスを detached オブジェクトとして値渡ししている場合，Serializable インタフェースを実装してください。
4. 埋め込み型クラスおよび埋め込み型クラスの永続化インスタンス変数と，すべてのメソッドでは final にしないでください。
5. 引数なしのコンストラクタを持つ必要があります。
6. 引数なしコンストラクタは，public か protected で宣言してください。
7. 埋め込み型クラスのインスタンス変数は，private，protected，または package から参照できなければなりません。
8. 埋め込み型クラスの永続化インスタンス変数は，クライアントから直接アクセスされないようにしてください。エンティティのアクセサメソッド（getter／setter メソッド）や，ほかのビジネスメソッドでアクセスしないでください。

CJPA プロバイダでは，1.および 2.の条件を満たさない場合には例外が発生して，アプリケーションの開始に失敗します。また，3.から 8.についても例外が発生するおそれがありますが，例外が発生しない場合でも動作は保証しません。

埋め込み型クラスのアクセスタイプは，埋め込まれた側のエンティティクラスのアクセスタイプで決定します。

埋め込み型クラスを利用する場合，埋め込みの階層は一つにしてください。また，複数のエンティティから埋め込み型クラスのオブジェクトを共有することはできません。これらの条件を満たさない場合，CJPA プロバイダでは動作を保証しません。

6.12.7　永続化フィールドおよび永続化プロパティのデフォルトマッピング規則

リレーションシップ以外の永続化フィールドまたは永続化プロパティに対して，O/R マッピング情報が指定されていない場合は，次に示すデフォルトのマッピング規則が適用されます。

- @Embeddable がアノテートされたクラスの場合，フィールド／プロパティは@Embedded 側のエンティティでの指定に従ってデータベースにマップされます。
- 永続化フィールド／永続化プロパティの型が次のどれかである場合，@Basic が定義されているのと同じ方法でマップされます。
  - Java のプリミティブ型
  - プリミティブ型のラッパ
  - java.lang.String
  - java.math.BigInteger
  - java.math.BigDecimal
  - java.util.Date
  - java.util.Calendar
  - java.sql.Date
  - java.sql.Time
  - java.sql.Timestamp
  - byte[]
  - Byte[]
  - char[]
  - Character[]
  - enum
  - Serializable を実装した任意の型

なお，上記以外の型が指定されている場合の動作は保証しません。

6.13　エンティティクラスの継承方法

エンティティクラスの継承には次に示す特長があります。

- エンティティクラスは抽象クラス，具象クラスのどちらでも使用できます。抽象クラスおよび具象クラスともに，@Entity が定義できます。また，エンティティとしてマップでき，抽象クラスおよび具象クラスの両方に対してクエリを発行できます。
- エンティティクラスはほかのエンティティクラスから継承できます。
- エンティティクラスは非エンティティクラスを継承できます。また，非エンティティクラスはエンティティクラスを継承できます。

ここでは，エンティティクラスの継承クラスの種類と継承マッピング戦略について説明します。

6.13.1　継承クラスの種類

エンティティクラスの継承クラスの種類には，抽象エンティティクラス，マップドスーパークラス，および非エンティティクラスがあります。それぞれ説明します。

(1)　抽象エンティティクラス

抽象クラスはエンティティクラスとして定義できます。抽象エンティティクラスは，直接インスタンスを作成できない点が具象エンティティクラスとは異なります。抽象エンティティクラスは，エンティティとしてマップすることも，サブクラスのエンティティを操作・取得するためにクエリの対象に指定することもできます。

サブクラスでは，プロパティのアクセサメソッドをオーバーライドできます。ただし，永続化フィールドおよび永続化プロパティの O/R マッピング情報をサブクラスでオーバーライドした場合，動作は保証しません。サブクラスで O/R マッピング情報をオーバーライドする場合は，@AssociationOverride や @AttributeOverride などを使用してください。

抽象エンティティクラスは，@Entity または O/R マッピングファイルで指定します。

(2)　マップドスーパークラス

マップドスーパークラスとは，エンティティクラスのスーパークラスになることができるクラスです。永続化フィールドやマッピング情報を定義でき，さらにこれらを継承するような構成にできます。

マップドスーパークラスは，特定のテーブルを指定できないため，@Table の指定はできません。そのため，マップドスーパークラスをエンティティにすることはできません。マップドスーパークラスはエンティティクラスとは異なり，問い合わせができないため，EntityManager やクエリの操作の引数に渡すこともできません。また，リレーションシップの対象にすることもできません。

マップドスーパークラスは，抽象クラスとしても，具象クラスとしても定義できます。クラスをマップドスーパークラスとして定義する場合は，@MappedSuperclass，または O/R マッピングファイルに <mapped-superclass>タグを定義します。マッピング情報は，継承されたエンティティクラスに対して提供されます。

マップドスーパークラスとして設計したクラスは，マッピングがサブクラスにだけ提供できるマッピングであることを除いて，エンティティクラスと同様の方法でマップできます。

エンティティがサブクラスとして適用されると，マップドスーパークラスで指定された情報が，サブクラスに継承されます。@AssociationOverride や O/R マッピングファイルで対応する XML 要素を使用して，マッピング情報をサブクラス上でオーバーライドすることができます。

(3) エンティティ継承階層構造の非エンティティのクラス

エンティティクラスは非エンティティクラスのスーパークラスを持つことができます。スーパークラスは具象クラスおよび抽象クラスとして定義できます。

非エンティティクラスのスーパークラスには次に示す特長があります。

- 振る舞いだけを継承します。
- 状態は永続ではありません。
- 継承したすべての状態は，継承したエンティティクラスでも永続ではありません。
- 永続でない状態は，EntityManager の制御の対象ではありません。
- スーパークラスのアノテーションは無視されます。

非エンティティクラスを EntityManager や Query インタフェースのメソッドの引数にしたり，マッピング情報にしたりしないでください。CJPA プロバイダの場合，アプリケーションの実行時に例外が発生します。

6.13.2 継承マッピング戦略

エンティティを継承した場合，クラス階層をテーブルにマッピングする方法を継承マッピング戦略として指定できます。継承マッピング戦略は，@Inheritance または O/R マッピングファイルの<entity>タグの<inheritance>タグを使用して指定します。

継承マッピング戦略には次の 3 種類があります。

- SINGLE TABLE 戦略
  SINGLE TABLE 戦略は，エンティティクラスの継承階層にあるすべてのクラスが一つのテーブルにマップする戦略方法です。
- JOINED 戦略
  JOINED 戦略は，クラス階層の最上位は単一のテーブルにマップする戦略方法です。
- TABLE PER CLASS 戦略
  エンティティクラスのクラス階層中の各クラスを別々のテーブルにマップする戦略方法です。

ただし，CJPA プロバイダでは，TABLE PER CLASS 戦略は使用できません。CJPA プロバイダを使用している場合で，TABLE PER CLASS 戦略を指定したときには，アプリケーションの起動時に例外が発生します。

! 注意事項

CJPA プロバイダを使用する場合，次のことに注意してください。

- クラス階層中で継承戦略を複数組み合わせて指定することはできません。また，複数の継承戦略を組み合わせているかどうかのチェックも実施しません。指定した場合の動作は保証しません。
- @DiscriminatorColumn で指定したカラムをエンティティのフィールドに定義した場合，persist 操作をしてコミットしてもエンティティのフィールドに設定した値はデータベースに反映されません。
  @DiscriminatorValue で指定した値またはそのデフォルト値が反映されます。また，コミット後には，コミット前にフィールドに設定されていた値が格納されたままとなるため注意が必要です。

これらは javax.persistence.DiscriminatorType 列挙型の値で指定されます。それぞれの戦略について説明します。

(1)　SINGLE TABLE 戦略

SINGLE TABLE 戦略とは，エンティティクラスの継承階層にあるすべてのクラスが一つのテーブルにマップする戦略方法です。そのため，テーブルではクラスを識別するためのカラムとして，識別カラムを持つ必要があります。

識別カラムは@DiscriminatorColumn，または O/R マッピングファイルで指定します。なお，デフォルトの識別カラム名は「DTYPE」です。データベースに識別カラムがない場合，アプリケーションの実行時に例外が発生します。

識別カラムに格納される値を指定したい場合には，@DiscriminatorValue または O/R マッピングファイルの<entity>タグ下の<discriminator-value>タグを使用して指定します。

! 注意事項

SINGLE TABLE 戦略を利用する場合，サブクラスのフィールドに対応するテーブルのカラムでは，null 値を指定できるようにする必要があります。

---

(2)　JOINED 戦略

JOINED 戦略とは，クラス階層の最上位は単一のテーブルにマップする戦略方法です。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特有のフィールドと，スーパークラス表の主キーの外部キーとして機能する主キー列を持つ別々のテーブルで示されます。

JOINED 戦略の場合も，SINGLE TABLE の場合と同様にスーパークラスがマップされるテーブルに識別カラムを持つ必要があります。

! 注意事項

JOINED 戦略を利用する場合，サブクラスのインスタンスの生成で複数回の結合の実行が必要になります。このため，階層構造が深くなると，性能が劣化するおそれがあります。また，クラス階層にわたる範囲でクエリを発行する場合，JOIN が必要です。

6.14 EntityManager および EntityManagerFactory の使用方法

ここでは，アプリケーションから利用する EntityManager および EntityManagerFactory の使用方法について説明します。

6.14.1 EntityManager でのエンティティのライフサイクル管理

EntityManager はデータベースに対して次の操作をするためのインタフェースを持つオブジェクトです。

- エンティティを登録したり，削除したりする。
- プライマリキーによってエンティティを検索する。
- エンティティをわたったクエリを発行する。

EntityManager に対してエンティティを登録すると，トランザクションのコミットなどの適切なタイミングでエンティティの状態がデータベースで永続化されます。

また，EntityManager はエンティティの集合を表す永続化コンテキストと関連を持ちます。
EntityManager にエンティティが登録されると，エンティティは特定の永続化コンテキストに属します。
また，EntityManager は，エンティティのライフサイクルを管理します。

EntityManager で管理するエンティティの集合は永続化ユニットという単位で定義します。永続化ユニットはアプリケーションの設定ファイルである persistence.xml で定義します。

永続化コンテキストと永続化ユニットの注意事項について説明します。

- 永続化コンテキスト内ではエンティティは一意となるようにします。このため，同一の永続化コンテキスト内では，データベースの同じ行を表すエンティティは一つとしてください。なお，永続化コンテキストが異なる場合は，データベースの同じ行を表すエンティティを複数持つことができます。この場合のデータベース上での排他方法については，「6.10　楽観的ロック」または「6.11　JPQL での悲観的ロック」を参照してください。
- 永続化ユニットはそれぞれ単一のデータベースにマッピングされます。定義の詳細については，「5.8 persistence.xml での定義」を参照してください。

6.14.2 EntityManager および EntityManagerFactory の設定方法

アプリケーションで利用する EntityManager および EntityManagerFactory は，アノテーションまたは DD で設定します。

- EntityManagerFactory の設定
  - アノテーションの場合：@PersistenceUnit で設定します。
  - DD の場合：<persistence-unit-ref>タグで設定します。
- EntityManager の設定
  - アノテーションの場合：@PersistenceContext で設定します。
  - DD の場合：<persistence-context-ref>タグで設定します。

なお，アノテーションの詳細については，マニュアル「アプリケーションサーバ リファレンス API 編」の「2.　アプリケーションサーバが対応しているアノテーションおよび Dependency Injection」を参照して

ください。また，DD に設定するタグの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」を参照してください。

6.14.3 EntityManager の API に関する注意事項

EntityManager が提供する API についての注意事項を次に示します。なお，EntityManager の API については，マニュアル「アプリケーションサーバ リファレンス API 編」の「2.7 javax.persistence パッケージ」を参照してください。

- トランザクションスコープ永続化コンテキストの EntityManager が使用されるとき，persist，merge，remove，refresh メソッドは，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。トランザクションコンテキストがない場合は，javax.persistence.TransactionRequiredException がスローされます。
- find メソッドと getReference メソッドは，トランザクションコンテキストでの実行を要求しません。このため，トランザクションスコープ永続化コンテキストの EntityManager が使用された場合，結果のエンティティは detached 状態になります。また, 拡張された永続化コンテキストの EntityManager が使用された場合，結果のエンティティは managed 状態になります。
- createQuery メソッドの引数が有効な JPQL の文字列でなければ，IllegalArgument 例外を送出し，クエリの実行が失敗します。
- 実行するネイティブクエリが，接続先のデータベースの仕様に合わない場合，または定義された結果のセットがクエリの結果と互換性がない場合，クエリの実行は失敗し，クエリの実行時に PersistenceExceptoin 例外がスローされます。
- EntityManager インタフェースのメソッドからランタイム例外が送出されると, カレントのトランザクションはロールバックにマークされます。
- EntityManager から取得した Query オブジェクト，および EntityTransaction オブジェクトは，EntityManager がオープンしている間は有効です。

6.15　コールバックメソッドの指定方法

エンティティのライフサイクルを受け取るために，メソッドをライフサイクルのコールバックメソッドに指定できます。コールバックメソッドとして定義されたメソッドは永続化に関するライフサイクルイベントに対応して呼び出されます。

ここでは，コールバックメソッドの指定個所，実装方法，および呼び出し順序について説明します。

6.15.1　コールバックメソッドの指定個所

コールバックメソッドは，次の場所に指定できます。

- エンティティクラスまたはマップドスーパークラス内
- エンティティクラスまたはマップドスーパークラスに関連づけられたエンティティリスナクラス

エンティティリスナクラスとは，コールバックメソッドを実装するための専用のクラスです。エンティティリスナクラスを使用すると，コールバックメソッドの実装部分を分離できます。

コールバックメソッドは，アノテーションまたは O/R マッピングファイルで指定します。ただし，デフォルトのコールバックメソッドは，O/R マッピングファイルで指定します。アノテーションでは指定できません。なお，デフォルトのコールバックメソッドとは，永続化ユニット内のすべてのエンティティに適用されるエンティティリスナを指します。

コールバックリスナの指定方法について説明します。

(1)　アノテーションでのコールバックメソッドの指定

コールバックメソッドの指定にアノテーションを使用する場合は，次の表にあるアノテーションをメソッドに設定してください。ライフサイクルイベントに応じてメソッドを呼び出すことができます。

表 6-18　アノテーションを使用したコールバックメソッドの指定

アノテーション	実行される内容
@PostLoad	エンティティが永続化コンテキストにロードされたあとか，または refresh 操作が適用されたあとに，コールバックメソッドが実行されます。キャッシュからエンティティを読み込んだあと，またはデータベースに SELECT 文を発行したタイミングで実行されます。
@PrePersist @PreRemove	EntityManager がエンティティの persist または remove 操作を行う前にコールバックメソッドが呼び出されます。マージ操作が適用され，新しく managed 状態のインスタンスが作成される場合，managed 状態のインスタンスへの PrePersist コールバックメソッドは，エンティティの状態がコピーされたあとに呼び出されます。これらの PrePersist または PreRemove のコールバックは操作がカスケードされるすべてのインスタンスに対しても呼び出されます。 PrePersist または PreRemove メソッドは常に persist，merge，または remove 操作の一部として同期を取って呼び出されます。
@PostPersist @PostRemove	PostPersist および PostRemove コールバックメソッドは，エンティティが persist または remove 操作で，永続化されたあとまたは削除されたあとで呼び出されます。 これらのコールバックは操作がカスケードされたすべてのエンティティに対しても呼び出されます。PostPersist または PostRemove メソッドは，それぞれデータベースの insert または delete 操作が行われたあとに呼び出されます。これらのデータベースへの操作は persist，merge，もしくは remove 操作の直後，または flush メソッドが呼び出されたあとになります。ただし，トランザクションの終了時になることもあります。生成されたプライマリキーは PostPersist メソッドで利用できます。

アノテーション	実行される内容
@PreUpdate @PostUpdate	PreUpdate および PostUpdate のコールバックはそれぞれエンティティデータのデータベースへの update 操作の前後に呼び出されます。 これらのデータベースへの操作はエンティティの状態が更新された場合，または状態がデータベースにフラッシュされた場合に実行されます。ただし，データベースへの操作は，トランザクションの終了時になることもあります。一つのトランザクション内で，エンティティを永続化してから更新したり，エンティティを更新してから削除したりした場合には，PreUpdate や PostUpdate のコールバックが発生しないことがあります。

エンティティリスナクラスを利用する場合，エンティティに対して@EntityListener を指定してエンティティリスナクラスを指定する必要があります。指定方法の例を次に示します。

```
@Entity
@EntityListeners(CallbackListener.class)
public class Employee implements Serializable{
 ・・・
}
```

(2)　O/R マッピングファイルでのコールバックリスナの指定

O/R マッピングファイルを使用してコールバックメソッドを指定する場合は次のように指定します。

- エンティティリスナのクラスおよびそのクラスのコールバックメソッドを指定する場合，O/R マッピングファイルの<entity-listener>タグを使用します。ライフサイクルリスナメソッドは，<entity-listener>タグ下の，<pre-persist>タグ，<post-persist,>タグ，<pre-remove>タグ，<post-remove>タグ，<pre-update>タグ，<post-update>タグ，および<post-load>タグを使用して指定します。
- エンティティリスナクラスのコールバックメソッドを指定する場合は，コールバックイベントごとに最大一つのメソッドを<entity-listener>タグ以下のタグを使用して指定できます。
- <persistence-unit-defaults>タグの<entity-listeners>タグの下位タグに対して，O/R マッピングファイルの<entity-listener>タグを指定するとデフォルトコールバックメソッドが指定できます。
- <entity>タグまたは<mapped-subclass>タグにある<entity-listeners>タグの下位タグに，<entity-listener>タグを指定すると，エンティティまたはマップドスーパークラスとそのサブクラスに対するコールバックリスナの指定になります。
- <entity-listeners>タグに指定したリスナの順番でコールバックリスナは呼び出されます。リスナの呼び出し順序については，「6.15.3　コールバックメソッドの呼び出し順序」を参照してください。

6.15.2　コールバックメソッドの実装

ユーザは必要に応じてコールバックメソッドを実装します。エンティティクラスやマップドスーパークラス内に実装するコールバックメソッドとエンティティリスナクラスのコールバックメソッドでコールバックメソッドのシグネチャは異なります。

エンティティクラスやマップドスーパークラスで定義されるコールバックメソッドは次に示すシグネチャになります。

```
void <METHOD>
```

また，エンティティリスナクラスで定義されるコールバックメソッドは次に示すシグネチャになります。

```
void <METHOD>(Object)
```

引数の Object には，コールバックメソッドが実行されるエンティティのインスタンスを指定します。

(1) コールバックメソッド使用時の注意事項

コールバックメソッドについては，次に示す注意事項があります。これらの条件を満たさない場合，アプリケーションの開始時に例外が発生して，アプリケーションの開始に失敗します。

- public で引数がないコンストラクタを持たなければなりません。
- コールバックメソッドでは public，private，protected，およびパッケージレベルのアクセスができます。ただし，static や final は使用できません。
- 一つのクラスが同じライフサイクルイベントに対して複数のライフサイクルコールバックメソッドを持つことはできません。ただし，同じメソッドが複数のコールバックイベントで使用されることがあります。

(2) コールバックメソッドに適用されるルール

コールバックメソッドについては次に示すルールが適用されます。

- コールバックメソッドでは，未チェックまたは実行時例外の送出が許可されています。トランザクション中に実行したコールバックメソッドでスローされた実行時例外は，トランザクションをロールバックさせます。コールバックメソッドが複数指定されている場合，実行時例外がスローされたあとには残りのコールバックメソッドは実行されません。
- コールバックメソッドでは，JNDI，JDBC，JMS，Enterprise Bean を実行できます。
- コールバックメソッドで次の操作をしないでください。
  - EntityManager を呼び出す。
  - クエリ操作を実行する。
  - ほかのエンティティインスタンスにアクセスする。
  - リレーションシップを更新する。

  このような方法で使用した場合，動作を保証しません。
- Java EE 環境でコールバックメソッドが呼び出された場合，エンティティのコールバックリスナは呼び出すコンポーネントのネーミングコンテキストを共有します。また，エンティティのコールバックメソッドは，コールバックメソッドが呼び出された時点の呼び出し元コンポーネントのトランザクションとセキュリティコンテキストで呼び出されます。

6.15.3 コールバックメソッドの呼び出し順序

エンティティに対して複数のコールバックメソッドを定義した場合，呼び出し順序は次の規則に従います。

1. O/R マッピングファイルに定義した順番でデフォルトリスナが呼び出されます。
   明示的に@ExcludeDefaultListeners または O/R マッピングファイルの<exclude-default-listeners>タグを指定しないかぎり，デフォルトリスナは永続化ユニット内のすべてのエンティティに適用されます。
2. @EntityListeners で指定された順番にコールバックメソッドが呼び出されます。
   なお，O/R マッピングファイルを使用すると，次の操作ができます。
   - エンティティに対するコールバックメソッドの呼び出し順序を指定する。
   - アノテーションでの指定順序をオーバーライドする。

3. エンティティ（またはマップドスーパークラス）に指定されたコールバックメソッドが呼び出されます。

(1) 継承階層内での呼び出し順序

エンティティクラスやマップドスーパークラスの継承階層の中で複数回のエンティティリスナを定義した場合，呼び出し順序は次のようになります。

1. デフォルトコールバックリスナがあれば，最初に呼び出されます。
2. エンティティリスナクラスのコールバックメソッドがスーパークラスに指定されているリスナから順番に呼び出されます。このとき，@EntityListener などの指定があれば，その順序に従います。
3. すべてのエンティティリスナのコールバックメソッドが呼び出されたあとで，エンティティ（またはマップドスーパークラス）に定義されたコールバックメソッドがスーパークラスに指定されているリスナから順番に呼び出されます。

コールバックメソッドをサブクラスでオーバーライドした場合には，オーバーライドされたメソッドは呼び出されません。オーバーライドしたコールバックメソッドが異なるライフサイクルのイベントを指定している場合，またはライフサイクルコールバックメソッドではない場合，オーバーライドされた側のメソッドは呼び出されます。また，メソッドのコールバックメソッドの設定はオーバーライドされます。

(2) コールバックメソッドの除外について

- @ExcludeDefaultListeners または O/R マッピングファイルの<exclude-default-listeners>タグを指定すると，デフォルトエンティティリスナはエンティティクラス（または mapped superclass）とそのサブクラスでは呼び出されません。
- @ExcludeSuperclassListeners または O/R マッピングファイルの<exclude-superclass-listeners>タグがエンティティクラスや mapped superclass に適用されると，そのクラスとそのサブクラスではリスナのコールバックメソッドは呼び出されません。@ExcludeSuperclassListeners または O/R マッピングファイルの<exclude-superclass-listeners>タグは，デフォルトのエンティティリスナの呼び出しを除外することにはなりません。
- O/R マッピングファイルを使用して，entity や mapped superclass への除外されたデフォルト/スーパークラスリスナを明示的に指定すると，エンティティやそのサブクラスに適用されることになります。

6.16 クエリ言語を利用したデータベースの参照および更新方法

クエリとは，データベースに対する処理要求（問い合わせ）を文字列として表したものです。データベース上のデータの検索，更新，削除などの命令をシステムに発行するときに使用します。クエリを実行するには，javax.persistence.Query インタフェースを使用します。クエリには，JPQL とネイティブクエリの 2 種類があります。ここでは，クエリを利用したデータベースの参照および更新方法について説明します。

6.16.1 JPQL でのデータベースの参照および更新方法

JPQL は，データベースを検索・更新したり，データベースが持っている集合関数などの機能を利用したりするためのクエリ言語です。SQL がテーブルを対象としたクエリ言語であるのに対して，JPQL はエンティティクラスを対象とした JPA 仕様で定義されているクエリ言語です。

クエリはアノテーションまたは O/R マッピングファイルで定義できます。クエリと同じ永続化ユニットでエンティティが定義されている場合，エンティティの集合を表す抽象スキーマ型をクエリで使用できます。また，パス式を使用すると，永続化ユニット内で定義されたリレーションシップをわたってクエリを使用できます。パス式の詳細については，「6.17.4(2) パス式」を参照してください。

アプリケーションに JPQL を記述して実行すると，次の順序で接続先のデータベースに対して SQL が発行されます。

1. JPQL が実行されると，CJPA プロバイダによって JPQL の内容が解釈される。
2. 対象となるエンティティクラス内に記述されたアノテーションや O/R マッピングファイルの情報を基に，接続先のデータベース製品固有の SQL 文に組み立てて発行する。

ここでは，JPQL の使用方法について説明します。

(1) Query オブジェクトの取得方法

JPQL を使用して Query オブジェクトを取得するためには，CJPA プロバイダが提供する次の EntityManager インタフェースのメソッドを使用します。EntityManager インタフェースのメソッドについて説明します。

(a) Query createQuery(String JPQL 文)

createQuery の記述例を次に示します。引数には実行する JPQL 文を指定します。

```
Query q = em.createQuery(
                  "SELECT c  " +
                  "FROM Customer c " +
                  " WHERE c.name LIKE "Smith");
```

(b) Query createNamedQuery(String クエリ名)

あらかじめ名前を付けて定義しておくことのできるクエリを名前付きクエリといいます。名前付きクエリは，@NamedQuery を任意のエンティティクラスに付与して定義します。@NamedQuery の name 属性にクエリ名を指定して，query 属性には JPQL 文を指定します。

CJPA プロバイダの場合，同じ名称の名前付きクエリを複数指定することはできません。同じ名称の名前付きクエリを複数指定した場合には，警告メッセージ KDJE55535-W を出力します。CJPA プロバイダで指定した場合，どのクエリが動作するかは保証しません。

次に，@NamedQuery の定義例を示します。この例では，@NamedQuery を使用してあらかじめ findAllCustomersWithName という名前でクエリを登録しています。アプリケーションの createNamedQuery メソッドに登録した名前付きクエリ名を渡すことで，事前に登録されているクエリを取得して利用します。

- @NamedQuery でのクエリ名の登録

```
@NamedQuery(
      name="findAllCustomersWithName",
      query="SELECT c FROM Customer c WHERE c.name LIKE :custName"
)
@Entity
public class Customer {
 ・・・
}
```

- createNamedQuery メソッドでの名前付きクエリの記述例

```
@Stateless
public class MySessionBean {
 ・・・
  @PersistenceContext
  public EntityManager em;
   ・・・
  public void doSomething() {
     ・・・
    Query q = em.createNamedQuery("findAllCustomersWithName")
                                      .setParameter("custName", "Smith");
  }
}
```

なお，同一の永続化ユニット内では，ほかのエンティティで定義した名前付きクエリを使用することもできます。

(2)　パラメタの指定方法

JPQL では，クエリを生成する際に，WHERE 節に記述する条件式にパラメタを使用して，動的に値を設定することができます。パラメタの値は Query インタフェースの setParameter メソッドで設定します。パラメタには，位置パラメタと名前付きパラメタがあります。それぞれについて説明します。

位置パラメタ

WHERE 節の中のパラメタを組み込みたい位置に，「?」と数値の組み合わせを記述します。パラメタの値は，Query インタフェースの setParameter メソッドで設定します。位置パラメタの記述形式と記述例を次に示します。

- 記述形式

```
Query setParameter(int 位置, Object 値)
```

- 記述例

```
Query q = em.createQuery(
      "SELECT c FROM Customer c WHERE c.balance < ?1")
.setParameter(1, 20000);
```

名前付きパラメタ

WHERE 節の中のパラメタを組み込みたい位置に，「:」と任意の文字列（ただし，0～9 の文字を除く）の組み合わせを記述します。パラメタの値は，Query インタフェースの setParameter メソッドで設定します。名前付きパラメタの記述形式と記述例を次に示します。

- 記述形式

```
Query setParameter(String パラメタ名, Object 値)
```

パラメタ名の先頭には「:」を指定しないでください。また，名前付きパラメタは大文字・小文字を区別します。

- 記述例

```
Query q = em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE :custName")
        .setParameter("custName", "John");
```

なお，パラメタを使用するときには次のことに注意してください。

- 一つのクエリで位置パラメタと名前付きパラメタを混在して使用しないでください。CJPA プロバイダの場合，混在して使用したときの動作は保証しません。
- setParameter メソッドには，java.util.Date 型または java.util.Calendar 型のオブジェクトをパラメタ値として設定するときに，次のように指定できます。なお，パラメタ値の時間タイプは TemporalType 列挙型で指定する必要があります。詳細については Java のドキュメントを参照してください。
  - Query setParameter(int 位置, Date 日付, TemporalType 時間タイプ)
  - Query setParameter(String パラメタ名, Date 日付, TemporalType 時間タイプ)
  - Query setParameter(int 位置, Calendar カレンダ, TemporalType 時間タイプ)
  - Query setParameter(String パラメタ名, Calendar カレンダ, TemporalType 時間タイプ)

(3) クエリ結果の取得および実行

生成したクエリを実行して，そのクエリ結果を返したり，更新クエリを実行したりするためには，Query インタフェースの次のメソッドを使用します。それぞれのメソッドについて説明します。

(a) Object getSingleResult()

このメソッドは，クエリ結果を単一のオブジェクトとして返す場合に使用します。

メソッドを実行するとデータを検索します。検索の結果，ヒットした単一の行をエンティティオブジェクトに格納し，Object 型で返します。Object 型の戻り値は対象のエンティティクラスにキャストする必要があります。

複数の行がヒットした場合は，NonUniqueResultException 例外が発生します。ヒットする行がなかった場合は，NoResultException 例外が発生します。

(b) List getResultList()

このメソッドは，クエリ結果をリストとして返す場合に使用します。

メソッドを実行するとデータを検索します。検索の結果，ヒットした複数の行をエンティティオブジェクトに格納し，リストで返します。実行結果として複数の行が返されること想定しているため，ヒットする行が一つもなかった場合は，空のリストが返ります。

(c) int executeUpdate()

このメソッドは，更新クエリを実行する場合に使用します。

メソッドを実行するとテーブルから複数の行を一斉に削除または更新するクエリを実行します。実行結果にはヒットした行の件数が返ります。

6.16.2　ネイティブクエリでのデータベースの参照および更新方法

CJPA プロバイダでは，JPQL 以外のクエリ言語として，データベース固有のネイティブクエリを直接記述して，データベースの参照・更新などを実行できます。

ここでは，ネイティブクエリでの使用方法について説明します。

(1)　Query オブジェクトの取得方法

ネイティブクエリを使用して Query オブジェクトを取得するためには，次に示す CJPA プロバイダが提供する EntityManager インタフェースのメソッドを使用します。

(a)　Query createNativeQuery(String SQL 文)

createNativeQuery の記述例を次に示します。引数には実行するネイティブクエリを指定します。

```
Query q = em.createNativeQuery(
                  ”SELECT o.id, o.quantity, o.item ” +
                  ”FROM Order o, Item i ” +
                  ”WHERE (o.item = i.id) AND (i.name = ’widget’)”);
```

(b)　Query createNativeQuery(String SQL 文，Class 結果格納クラス)

createNativeQuery の記述例を次に示します。第 1 引数には実行するネイティブクエリ，第 2 引数には実行結果を格納するクラスオブジェクトを指定します。

```
Query q = em.createNativeQuery(
                  ”SELECT o.id, o.quantity, o.item ” +
                  ”FROM Order o, Item i ” +
                  ”WHERE (o.item = i.id) AND (i.name = ’book’)”,
                  com.hitachi.Order.class);
```

この例の場合，クエリが実行されると「book」という名前のアイテムに対するすべての Order エンティティのコレクションを返します。

なお，SELECT 節で指定したクエリの結果と引数に指定したクラスオブジェクトの整合性がない場合は例外が発生します。

(c)　Query createNativeQuery(String SQL 文，String 結果セットマッピング名)

第 1 引数には実行するネイティブクエリ，第 2 引数には実行結果を格納する結果セットマッピング名を指定します。結果セットマッピングは，@ SqlResultSetMapping で指定します。なお，結果セットマッピングについては，「(2)　結果セットマッピング」を参照してください。

@SqlResultSetMapping の定義例と createNativeQuery の記述例を次に示します。

- @SqlResultSetMapping の定義例

```
@SqlResultSetMapping(name=”BookOrderResults”,
      entities=@EntityResult(entityClass=com.hitachi.Order.class))
```

- createNativeQuery の記述例

```
Query q = em.createNativeQuery(
                  ”SELECT o.id, o.quantity, o.item ” +
                  ”FROM Order o, Item i ” +
                  ”WHERE (o.item = i.id) AND (i.name = ’book’)”,
                  ”BookOrderResults”);
```

この例の場合，クエリが実行されると「book」という名前のアイテムに対するすべての Order エンティティのコレクションを返します。@SqlResultSetMapping を使用することで，「6.16.1(1) Query オブジェクトの取得方法」の@NamedQuery を使用した場合の記述例と同じ結果を得ることができます。

なお，SELECT 節で指定したクエリ結果と引数に指定した@SqlResultSetMapping 設定の整合性がない場合は例外が発生します。

(d) Query createNamedQuery(String クエリ名)

ネイティブクエリの場合も，JPQL と同じように createNamedQuery メソッドを使用できます。ネイティブクエリの場合は，引数に名前付きネイティブクエリ名を指定してください。

名前付きネイティブクエリは，@NamedNativeQuery を任意のエンティティクラスに付与して定義します。引数のクエリ名には，@NamedNativeQuery の name 属性で指定した名前を使用します。

CJPA プロバイダの場合，同じ名称の名前付きクエリを複数指定することはできません。同じ名称の名前付きクエリを複数指定した場合には，警告メッセージ KDJE55522-W を出力します。CJPA プロバイダで指定した場合，どのクエリが動作するかは保証しません。

次に，createNamedQuery メソッドの使用例を示します。この例では，@NamedNativeQuery を使用してあらかじめ findBookOrder という名前でクエリを登録しています。アプリケーションの createNamedQuery メソッドに登録した名前付きクエリ名を渡すことで，事前に登録されているクエリを取得して利用します。

- @NamedNativeQuery でのクエリ名の登録

```
@NamedNativeQuery( name="findBookOrder",
       query="SELECT o.id, o.quantity, o.item " +"
             "FROM Order o, Item i " +
             "WHERE (o.item = i.id) AND (i.name = 'book')")
)
@Entity
public class Order {
 . . .
}
```

- createNamedQuery メソッドでの名前付きネイティブクエリの記述例

```
@Stateless
public class MySessionBean {
   . . .
  @PersistenceContext
  public EntityManager em;
   . . .
  public void doSomething() {
     . . .
    Query q = em.createNamedQuery("findBookOrder ")
  }
}
```

なお，同一の永続化ユニット内では，ほかのエンティティで定義した名前付きネイティブクエリを使用することもできます。

(2) 結果セットマッピング

結果セットマッピングとは，ネイティブクエリの実行結果を任意のエンティティクラスにマッピングして受け取ったり，スカラ値で受け取ったりするための機能です。

結果セットマッピングでは，ネイティブクエリの実行結果として取得した各カラム値のマッピング情報を@SqlResultSetMapping を指定して任意のエンティティクラスに対して付与します。

(a) @SqlResultSetMapping の記述形式

@SqlResultSetMapping の記述形式を次に示します。

```
@SqlResultSetMapping(
  name= 結果セットマッピングの名前,
  entities= 結果をマッピングするためのエンティティクラス指定(@EntityResultの配列),
  columns= 結果をマッピングするためのカラム指定(@ColumnResultの配列) )
```

name 属性

結果セットマッピング名を指定します。

entities 属性

@EntityResult の配列を指定します。@EntityResult の記述形式を次に示します。

```
@EntityResult(
    entityClass= 結果をマッピングするためのクラスを指定,
    fields= 結果をマッピングするためのフィールド指定(@FieldResultの配列) )
```

@EntityResult の entityClass 属性には，カラム値を格納するエンティティクラスを指定します。また，field 属性には，@FieldResult の配列を指定します。

@EntityResult の記述形式を次に示します。

```
@FiledResult(
    name= クラスの永続プロパティ(またはフィールド)の名前,
    column= SELECT節のカラムの名前(または別名) )
```

@FieldResult の name 属性には，@EntityResult の entityClass 属性に指定したエンティティクラスの永続化フィールド名を指定します。また，column 属性にはカラム名を指定します。

columns 属性

columns 属性は，エンティティクラスに格納しないでスカラ値として受け取るために，@ColumnResult の配列を指定します。スカラ値を取り出す必要がない場合は指定する必要はありません。@ColumnResult の name 属性には，値を取り出すカラム名を指定してください。

@ColumnResult の記述形式を次に示します。

```
@ColumnResult(
    name= SELECT節のカラムの名前(または別名) )
```

なお，カラム名は AS で指定したエリアス名でも指定できます。SELECT 節に同じ名前の複数の列を含む場合は，列の別名を使用してください。

(b) 使用例

従業員テーブル（Employee）と部門テーブル（Department）から従業員番号 12003 のクエリ結果を任意のエンティティクラス（EmployeeSetmap）にマッピングして，スカラ値（EMP_MONTHLY_SALARY カラム）を受け取る例を次に示します。

結果セットマッピング名（NativeQuerySetMap）を createNativeQuery の第 2 引数に指定して，結果セットマッピングを実行します。

- 結果セットマッピングを使用したネイティブクエリの記述例

```
query = em.createNativeQuery(
                ”SELECT e.EMPLOYEE_ID AS EMP_EMPLOYEE_ID, ” +
                ”e.EMPLOYEE_NAME AS EMP_EMPLOYEE_NAME, ” +
                ”d.DEPARTMENT_NAME AS DEP_DEPARTMENT_NAME, ” +
                ”e.MONTHLY_SALARY AS EMP_MONTHLY_SALARY  ” +
                ”FROM EMPLOYEE e, DEPARTMENT d ” +
                ”WHERE e.DEPARTMENT_ID = d.DEPARTMENT_ID ” +
                ”AND e.EMPLOYEE_ID = 12003”,
                ”NativeQuerySetMap”);
```

- ネイティブクエリの実行結果を格納する任意のエンティティクラス

```
@Entity
public class EmployeeSetmap implements Serializable {
:
@Id
public int getEmployeeId() { return employeeId; }
public String getEmployeeName() { return employeeName; }
public String getDepartmentName() { return departmentName; }
:
}
```

- @ SqlResultSetMapping の記述例

```
@SqlResultSetMapping(
       name="NativeQuerySetmap",
       entities={ @EntityResult(
                        entityClass=EmployeeSetmap.class,
                        fields={ @FiledResult(
                                    name="employeeId",
                                    column="EMP_EMPLOYEE_ID"),
                                 @FiledResult(
                                    name="employeeName",
                                    column="EMP_EMPLOYEE_NAME"),
                                 @FiledResult(
                                    name="departmentName",
                                    column="DEP_DEPARTMENT_NAME") } ) },
       columns={ @ColumnResult(
                        name="EMP_MONTHLY_SALARY") }
```

なお，@ SqlResultSetMapping を実行してネイティブクエリを実行した結果の Object 型配列は，次のようになります。

Object[0]

EmployeeSetmap クラスのオブジェクト

（EMP_EMPLOYEE_ID カラム，EMP_EMPLOYEE_NAME カラム，DEP_DEPARTMENT_NAME カラムの値が各フィールドに格納される）

Object[1]

EMP_MONTHLY_SALARY カラムの値

(3)　パラメタの指定方法

ネイティブクエリは，JPQL と同様に，パラメタによって動的に値を設定することができます。WHERE 節の中のパラメタを組み込みたい位置に，「?」と数値の組み合わせを記述します。パラメタの値は，Query インタフェースの setParameter メソッドで設定します。ただし，ネイティブクエリでは，JPQL の名前付きパラメタは使用できません。

パラメタの記述形式を次に示します。

```
Query setParameter(int 位置, Object 値)
```

(4)　ネイティブクエリ結果の取得および実行

ネイティブクエリ結果の取得および実行は，JPQL と同様に，Query インタフェースの次のメソッドを使用します。

- Object getSingleResult()
- List getResultList()
- int executeUpdate()

これらのメソッドの詳細については，「6.16.1　JPQL でのデータベースの参照および更新方法」を参照してください。

6.16.3　クエリ結果件数の範囲指定

複数件あるクエリ結果の中から任意に指定した件数だけを取得したり，開始位置で指定したクエリ結果だけを取得したりすることができます。これらの情報を取得するには，Query インタフェースのメソッドを使用します。メソッドについて次に説明します。

(1)　setMaxResults メソッド

複数件あるクエリ結果の中から任意に指定した件数だけを取得するには Query インタフェースの setMaxResults メソッドを使用します。setMaxResults メソッドの記述形式を次に示します。

```
Query setMaxResults(int検索結果の最大数)
```

メソッドの引数には，検索結果の最大数を指定します。

(2)　setFirstResult メソッド

開始位置で指定したクエリ結果だけを取得するには setFirstResult メソッドを使用します。setFirstResult メソッドの記述形式を次に示します。

```
Query setFirstResult(int検索結果の開始位置)
```

メソッドの引数に検索結果の開始位置を指定します。なお，開始位置は 0 から始まる数値を指定してください。

(3)　Query インタフェースのメソッドの使用例

setMaxResults メソッドおよび setFirstResult メソッドの使用例を示します。この例では，従業員データ（Employee）から，月給（e.monthlySalary）の多い順に 10 番目から 5 人分の Employee オブジェクトを取得します。

```
Query query = em.createQuery( "SELECT e FROM Employee AS e " +
                              "ORDER BY e.monthlySalary DESC" )
                              .setFirstResult(9)
                              .setMaxResults(5);
List resultList = query.getResultList();
```

(4)　注意事項

setFirstResult メソッドを使用して開始位置を指定した場合，引数に指定した値によって，getResultList メソッドおよび getSingleResult メソッドを呼び出してから結果の値を取得するまでの時間が変わります。通常，引数に指定した開始位置の値に比例して，結果の値が戻ってくるまでの時間が掛かります。

6.16.4　フラッシュモードの指定

エンティティオブジェクトに対して実行された未コミット操作をクエリがどのように扱うかを指定できます。この指定をフラッシュモードの指定といいます。

フラッシュモードは，javax.persistence.Query インタフェースの setFlushMode メソッドで設定します。CJPA プロバイダでは，設定できる値は AUTO だけです。COMMIT は設定できません。

FlushModeType.AUTO の場合

トランザクション内でクエリが実行されたとき，クエリ結果に影響を及ぼす永続化コンテキストにあるすべてのエンティティの変更内容がクエリ結果に反映されます。

この設定は，EntityManager インタフェースの setFlushMode メソッドのフラッシュモードに関係なくクエリに適用されます。

6.16.5　クエリヒントの指定

クエリ実行時に，ベンダに依存するヒントとしてクエリヒントを指定できます。CJPA プロバイダでは，悲観的ロックを使用するときに指定します。

クエリヒントは次に示す場所に指定します。

- Query オブジェクトの setHint()メソッドの引数
- @NamedQuery の引数の@Hint
- O/R マッピングファイルの<named-query>タグの下位要素である<hint>タグ

クエリヒントに指定できる範囲以外の値が設定された場合は例外が発生します。なお，指定する値は大文字と小文字を区別しません。

例外が発生するタイミングは，クエリヒントが指定された個所によって異なります。次に例外発生のタイミングを示します。

- setHint()メソッドの場合，アプリケーションのクエリ実行時
- アノテーションの場合，デプロイ時
- O/R マッピングファイルの場合，アプリケーション開始時

CJPA プロバイダがサポートするクエリヒントについては，アノテーションを使用する場合は，マニュアル「アプリケーションサーバ リファレンス API 編」の「2.1　対応するアノテーションのサポート範囲」，O/R マッピングファイルを使用する場合は，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「6.3　O/R マッピングファイル」を参照してください。

6.16.6　クエリの実行時の注意事項

ここでは，クエリ実行時の注意事項について説明します。

- setMaxResults メソッドまたは setFirstResult メソッドで，コレクション同士が FETCH JOIN を含むクエリを実行した場合，結果は保証されません。
- executeUpdate メソッド以外の Query メソッドは，トランザクション内で実行する必要はありません。特に getResultList と getSingleResult メソッドはトランザクション内で実行する必要はありません。
- トランザクションスコープ永続化コンテキストの EntityManager で，クエリが実行された場合の結果のエンティティは，detached 状態になります。拡張永続化コンテキストの EntityManager でクエリが実行された場合，すべて managed 状態になります。
- Query インタフェースのメソッドからスローされる NoResultException と NonUniqueResultException 以外の実行時例外はカレントのトランザクションをロールバックします。

6.17　JPQL の記述方法

ここでは，JPQL の記述方法について説明します。

6.17.1　JPQL の構文

JPQL 文には，SELECT 文と UPDATE 文，DELETE 文があります。

JPQL 文は，動的に指定したり，アノテーションや O/R マッピングファイルのタグで静的に定義したりすることができます。また，JPQL のすべての文でパラメタの指定ができます。

JPQL は型付き言語で，すべての式は型を持ちます。式の型は，式の構成や識別変数で定義された抽象スキーマ型，永続化フィールドとリレーションシップを評価する型，およびリテラル型で構成されています。なお，構文の文法については，「付録 D　JPQL の BNF」を参照してください。

! 注意事項

CJPA プロバイダの場合，BNF 構文に準拠しない JPQL を使用すると，例外が発生するおそれがあります。例外が発生しない場合でも，動作は保証しません。また，BNF 構文に準拠した JPQL を使用した場合でも，使用するデータベースで該当する機能をサポートしていない場合の動作は保証しません。

(1)　抽象スキーマ型

JPQL では，エンティティを対象にクエリを発行します。そのため，クエリでは対象となるエンティティの抽象スキーマを定義する必要があります。エンティティの抽象スキーマ型は，アノテーションまたは O/R マッピングファイルによって提供されるエンティティクラスと O/R マッピングの情報によって定義されます。

抽象スキーマ型とは，エンティティクラスのことを指します。エンティティクラスを構成するものとして，フィールドや，アノテーションなどの O/R マッピング情報があります。

エンティティの抽象スキーマ型は次に示すフィールドを持ちます。

- **ステートフィールド**
  ステートフィールドは，エンティティクラスの永続化フィールド，または永続化プロパティで，リレーションシップによる関連が存在しないフィールド，またはプロパティです。
- **関連フィールド**
  関連フィールドは，エンティティクラスでリレーションシップによって関連づいた永続化フィールド，または永続化プロパティです。リレーションシップが OneToMany または ManyToMany の場合，フィールドはコレクションになります。

(2)　抽象スキーマ名

JPQL では，エンティティを示すために，その抽象スキーマ型を指定する必要があります。抽象スキーマ型を示すための名前を抽象スキーマ名といいます。抽象スキーマ名は@Entity の name 属性（または O/R マッピングファイルの<entity>タグの name 属性）で定義します。name 属性が指定されなかった場合は，エンティティクラスの（パッケージ名抜きの）クラス名となります。

抽象スキーマ名は，永続化ユニット単位で固有となります。

(3) クエリのドメイン

クエリのドメインは，永続化ユニット内で定義されたすべてのエンティティの抽象スキーマ型を参照できます。抽象スキーマ型の中で定義されている関連フィールドによって，ほかの関連するエンティティの抽象スキーマ型を参照できます。

6.17.2 SELECT 文

SELECT 文には次の節があります。

- SELECT 節
  検索するオブジェクトの型か集合関数による値を指定します。SELECT 節は必ず指定します。
- FROM 節
  検索が適用される範囲を指定します。FROM 節は必ず指定します。
- WHERE 節
  検索結果を絞るために使用します。WHERE 節は省略できます。
- GROUP BY 節
  検索結果をグループ化するために使用します。GROUP BY 節は省略できます。
- HAVING 節
  グループ化されたものをフィルタリングするために使用します。HAVING 節は省略できます。
- ORDER BY 節
  検索結果の順序化をするために使用します。ORDER BY 節は省略できます。

SELECT 文に，SELECT 節および FROM 節の指定がない場合は例外が発生します。

6.17.3 SELECT 節

SELECT 節ではクエリの結果を表します。一つ以上の値がクエリの SELECT 節から返ります。SELECT 節には，次に示す要素をコンマで区切って指定します。なお，コンマで区切られた一つのまとまりを select 式と呼びます。

- 抽象スキーマの識別子または識別子を付与した永続化フィールド
- パス式
- 集合関数
- コンストラクタ式

なお，DISTINCT キーワードは，重複する値をクエリ結果から除くときに指定してください。

SELECT 節の記述例を次に示します。

```
SELECT e.employeeName, e.monthlySalary
FROM Employee AS e
WHERE e.monthlySalary  < 150000
```

(1) コンストラクタ式

コンストラクタ式は，一つ以上の Java のインスタンスを返す SELECT 節の select 式で使用されます。生成名は完全修飾名を指定します。

コンストラクタ式は，エンティティのカラムの一部または全部や，関連する別のエンティティのカラムと組み合わせた形で取得できます。なお，この結果を格納するクラスはエンティティである必要はありません。

コンストラクタ式の構文は，select 式に NEW 演算子を付けて指定します。格納するクラスがエンティティの場合，エンティティクラスのインスタンスの状態は new になります。コンストラクタ式の記述例を次に示します。

```
SELECT NEW com.hitachi.jpa.test.entity.EmployeeTmp
(e.employeeId, e.employeeName, d.departmentName)
FROM Department AS d, d.employees AS e
WHERE e.employeeId = 12003
```

(2)　集合関数

SELECT 節では集合関数を使用できます。使用できる集合関数を次の表に示します。

表 6-19　JPQL の SELECT 節で使用できる集合関数

集合関数	引数	結果の型	適用される値がない場合の結果
AVG	数値型のステートフィールド※	Double 型	null
MAX	順序を指定できるフィールド型(数値型，文字列型，文字型，または日付型）※	適用するフィールドの型	null
MIN	順序を指定できるフィールド型(数値型，文字列型，文字型，または日付型）※	適用するフィールドの型	null
SUM	数値型のステートフィールド※	• 整数型の場合：Long 型 • 浮動小数点型の場合：Double 型 • BigInteger 型の場合：BigInteger 型 • BigDecimal 型の場合：BigDecimal 型	null
COUNT	識別変数(引数にパス式を指定する場合，ステートフィールドか関連フィールドを指定する)	Long 型	0

注　DISTINCT が指定されているかどうかにかかわらず，集合関数が適用される前に null 値は除去されます。

注※　引数にパス式を指定する場合，関連フィールドを指定することはできません。

なお，集合関数式の構文の詳細については，「付録 D　JPQL の BNF」を参照してください。

(3)　SELECT 節の実行結果

クエリの SELECT 節で定義されるクエリ結果の型は，次のどれか一つになります。なお，複数ある場合は順番に並べられます。

- エンティティ抽象スキーマ型
- フィールド型
- 集合関数の結果
- コンストラクタ式の結果

SELECT 節の結果の型は，SELECT 節に含まれる select 式の結果の型によって定義されます。複数の select 式が SELECT 節で使われている場合，クエリの結果は Object[]型となります。この結果の要素は SELECT 節で指定された順番に一致し，各 select 式の結果の型と一致します。

6.17.4　FROM 節

ここでは FROM 節について説明します。

(1)　範囲変数宣言と識別変数

範囲変数宣言とは，FROM 節内でエンティティクラスの論理的な名前を記述し，そのあとに AS と識別子を指定する宣言です（AS は省略できます）。この範囲変数宣言の識別子を識別変数といいます。範囲変数宣言と識別変数の例を次に示します。

```
SELECT ・・・（省略）・・・
FROM  Department AS dep
WHERE ・・・（省略）・・・
```

「Department AS dep」の部分が範囲変数宣言です。また，「dep」が識別変数となります。次に，範囲変数宣言の構文を示します。

```
range_variable_declaration ::=
abstract_schema_name [AS] identification_variable
```

範囲変数宣言での識別変数の構文は，SQL の構文と同じです。構文について説明します。

- キーワード AS の使用は任意です。
- 識別変数を省略することはできません。ただし，抽象スキーマと識別変数の間に指定する AS は省略できます。識別変数は FROM 節で指定します。
- 予約済みの識別子は使用できません。使用した場合は例外が発生します。
- 同じ永続化ユニットのほかのエンティティと同じ名前は使用できません。CJPA プロバイダでは，同じ名前のエンティティを使用した場合の動作は保証しません。
- 識別変数は大文字，小文字の区別をしません。
- 抽象スキーマ名と同一の名前は指定できません。CJPA プロバイダでは，同じ名前の抽象スキーマを指定した場合の動作は保証しません。
- Java 識別子文字で始まり，ほかのすべての文字は Java 識別子の部分文字となる必要があります。それ以外の文字が指定された場合は，例外が発生します。最初の文字は，Character.isJavaIdentifierStart メソッドの戻り値が true になる文字にします（アンダースコア（_）とドルマーク（$）文字を含む）。最初以外の文字は，Character.isJavaIdentifierPart メソッドの戻り値が true になる文字にします（ただし，クエスチョンマーク（?）は，JPQL の予約語のため使用できません）。
- JPQL の予約語は次のとおりです。

  SELECT，FROM，WHERE，UPDATE，DELETE，JOIN，OUTER，INNER，LEFT，GROUP，BY，HAVING，FETCH，DISTINCT，OBJECT，NULL，TRUE，FALSE，NOT，AND，OR，BETWEEN，LIKE，IN，AS，UNKNOWN，EMPTY，MEMBER，OF，IS，AVG，MAX，MIN，SUM，COUNT，ORDER，BY，ASC，DESC，MOD，UPPER，LOWER，TRIM，POSITION，CHARACTER_LENGTH，CHAR_LENGTH，BIT_LENGTH，CURRENT_TIME，CURRENT_DATE，CURRENT_TIMESTAMP，NEW，EXISTS，ALL，ANY，SOME

なお，「UNKNOWN」は JPA1.0 では利用していませんが，CJPA プロバイダでは予約語となっているので注意してください。

(2)　パス式

パス式は，識別変数のあとにピリオド（.）を付加し，ステートフィールドまたは関連フィールドを続けるための式です。このため，パス式の型は，ステートフィールドまたは関連フィールドの型になります。

パス式をたどって得た関連フィールドから，さらにパス式を組み立てることができます。ただし，基になるパス式の型がコレクション関連フィールドである場合，パス式を組み立てることはできません。コレクション型からパス式を作ることは，構成的に誤りとなります。

なお，パス式の途中の関連フィールドが null 値の場合，パスは値がないとみなされるので，クエリの結果には影響はありません。

パス式は，inner join を使用する構文で使用できます。パス式の構文の詳細については，「付録 D　JPQL の BNF」を参照してください。

参考

関連フィールドの種類を次に示します。

- コレクション関連フィールド（collection_valued_association_field）とは，関連フィールドがコレクションで指定されているものです。OneToMany または ManyToMany の関係で示されます。
- 非コレクション関連フィールド（single_valued_association_field）とは，関連フィールドが single-valued で指定されているものです。OneToOne または ManyToOne の関係で示されます。
- エンベッデドクラスフィールドは，embedded クラスに対応するエンティティのフィールド名です。

(3)　Joins 式

Joins 式は FROM 節で使用できます。使用できる Joins 式を次の表に示します。

表 6-20　FROM 節で使用できる Joins 式

Joins 式	内容	BNF 構文の構文名※1
Inner Joins	関係するフィールドで，二つのエンティティクラスを結合し，関連を持っているエンティティオブジェクトだけを抽出します。	join, join_spec
Left Outer Joins	関係するフィールドで，二つのエンティティクラスを結合し，関連を持っているエンティティオブジェクトおよび，関連を持っていないエンティティオブジェクトも抽出します。	join, join_spec
Fetch Joins	関係のあるフィールドで，二つエンティティクラスを結合します。なお，エンティティ間にはリレーションシップによる関連があるため，Select 節中には一つのエンティティクラスだけを指定します。※2	fetch_join

注※1　BNF 構文の構文名の詳細については，「付録 D　JPQL の BNF」を参照してください。

注※2　Fetch Joins でエンティティを取得した場合，右側で指定した関連先のエンティティの情報をクエリ実行と同時に取得します。これによって，フェッチ戦略に依存しないで関連先の情報を取得できます。Fetch Joins の記述例を次に示します。

```
SELECT emp FROM Employee AS emp JOIN FETCH emp.company
```

Join 式を使用する場合の注意事項

Join 式を使用する場合の注意事項について説明します。

Inner Joins の注意事項

INNER キーワードは，任意で使われます。

Left Outer Joins の注意事項

OUTER キーワードは，任意で使われます。

Fetch Joins の注意事項

- Fetch Joins では，一つのエンティティで二つのエンティティ情報を指定します。指定するエンティティと，そのエンティティと関連する別のエンティティ情報を結合します。

  なお，CJPA プロバイダの場合，一つのエンティティ情報で結合するため，リレーションシップを持つエンティティを指定してください。リレーションシップを持たないエンティティを指定した場合は例外が発生します。
- JOIN FETCH の右側で参照される関係は，クエリの結果として返されるエンティティに属する関係となる必要があります。CJPA プロバイダの場合，エンティティに属さないときは例外が発生します。
- JOIN FETCH の右側で参照されるエンティティには，識別子を指定できません。このため，クエリ内で参照することはできません。

(4) コレクションメンバの宣言

コレクションメンバ宣言の識別変数は，予約された識別子 IN を使って宣言します。コレクションメンバ式で定義された識別変数は，パス式を使ってコレクションの値を取得できます。コレクションメンバ式の記述例を次に示します。

```
SELECT emp.employeeId, emp.employeeName, dep.departmentName
FROM Department AS dep, IN (dep.employees) AS emp
WHERE dep.departmentId = 3
```

なお，コレクションメンバ式の構文の詳細については，「付録 D　JPQL の BNF」を参照してください。

(5) 注意事項

ここでは，FROM 節での注意事項について説明します。

- 識別変数の影響

  FROM 節で宣言された識別変数が WHERE 節の中で使用されなくても，宣言された識別変数がクエリの結果に反映されます。
- 多態性の注意事項

  JPQL はポリフォリズムです。FROM 節で明確に参照している特定のエンティティクラスのインスタンスだけでなく，そのサブクラスも対象になります。このため，クエリによって取得できるインスタンスは，クエリ条件を満たすサブクラスのインスタンスを含みます。

6.17.5 WHERE 節

WHERE 節は式を満たすオブジェクトまたは変数を検索するために使われる条件式で構成されています。WHERE 節では，select 文の結果や update や delete 操作の範囲を特定します。

(1)　WHERE 節で使用できる条件式

WHERE 節で使用できる条件式を次の表に示します。

表 6–21　WHERE 節で使用できる条件式

式	内容	BNF 構文の構文名※
BETWEEN	フィールドで指定した範囲に含まれていることを評価します。	between_expression
IN	フィールドで指定したどれかの値と一致することを評価します。	in_expression
LIKE	フィールドでワイルドカード記号を展開したあとの文字列と一致することを評価します。	like_expression
IS [NOT] NULL	null 値かどうかをテストします。	null_comparison_expression
IS [NOT] EMPTY	指定したコレクション値が空かどうかをテストします。	empty_collection_comparison_expression
[NOT] MEMBER [OF]	指定したコレクション値がコレクションのメンバであるかどうかをテストします。空のコレクションを表す場合，MEMBER OF 式の値は FALSE，NOT MEMBER OF 式の値は TRUE となります。	collection_member_expression
EXISTS	サブクエリの結果を判定します。一つ以上の値がある場合は true，それ以外は false を返します。	exists_expression
ALL	サブクエリで返されたすべての値と比較します。	all_or_any_expression
ANY (SOME)	サブクエリで返されたどれかの値と比較します。	all_or_any_expression
サブクエリ	select による結果を記述できます。	subquery
関数式	「(2)　関数式」を参照してください。	該当する関数構文を参照

注※　BNF 構文の構文名の詳細については，「付録 D　JPQL の BNF」を参照してください。

WHERE 節を使用する場合の注意事項について説明します。

- IN 式
  - 評価対象のフィールドが null 値か unknown の場合，式の値は unknown になります。
  - IN 式に対する値の集合を定義したコンマで区切られたリストでは，少なくとも一つ以上の要素が必要です。CJPA プロバイダの場合，一つ以上の要素がないと例外が発生します。
  - 評価対象のフィールドは，文字列，数値，または enum 値を持つ必要があります。
  - リテラルや入力パラメタ値，およびサブクエリの結果は，評価対象のフィールドと同じ抽象スキーマ型にしてください。CJPA プロバイダの場合，同じ抽象スキーマ型でないと例外が発生します。
- LIKE 式
  - BNF 構文で示される string_expression または pattern_value の値が，null または unknown の場合，LIKE 式の値は unknown になります。
  - BNF 構文で示される escape_character が指定されている場合で，値が null のとき，LIKE 式の値は unknown になります。
  - 指定するパターン値は，アンダースコア（_）が 1 文字に相当し，パーセント（%）文字が連続文字（空の連続文字も含む）に相当します。なお，ほかの文字は検索文字列を示します。

- オプションのエスケープ文字は，文字列リテラルかまたは character 文字です。パターン文字列のアンダースコアとパーセント文字を標準の文字として解釈させる場合に使用します。

- IS [NOT] NULL 式

  指定したコレクション値が null または unknown の場合，IS [NOT] EMPTY 式の値は unknown になります。

- [NOT] MEMBER [OF]式

  指定した値が null または unknown の場合，戻り値は unknown になります。

- ALL 式または ANY (SOME)式
  - 条件演算が true でも false でもない場合は，unknown になります。
  - 一緒に使用できる比較演算子は，=，<，<=，>，>=，<>です。
  - サブクエリの結果の型は比較演算子の型と同じにする必要があります。CJPA プロバイダの場合，型が異なると例外が発生します。
  - SOME は，ANY の同義語です。※

  注※　この注意事項は ANY (SOME)式の場合だけ該当します。

- サブクエリ

  WHERE 節や HAVING 節で使用します。FROM 節では使用できません。

(2)　関数式

JPQL では次の表に示す関数を WHERE 節や HAVING 節で使用できます。関数式の引数の値が null か unknown の場合，関数式の値は unknown になります。

表 6-22　JPQL で使用できる関数

分類	関数	返り値	引数についての補足説明
文字列関数	CONCAT	引数の連結した文字列	−
	SUBSTRING	引数で指定した開始位置と長さの文字列	2 番目と 3 番目引数は，返されるサブストリングの開始位置および長さを整数で指定します。文字列の先頭の位置は 1 です。
	TRIM	特定の文字を取り除いた文字列	取り除く文字がない場合は，スペース（空白）と見なされます。任意の trim 文字は character の入力文字列です。トリムの方法が指定されない場合は BOTH となります。
	LOWER	小文字化した文字列	−
	UPPER	大文字化した文字列	−
	LENGTH	文字列長の整数値	−
	LOCATE	指定された位置から検索し，与えられた文字列で最初に文字列を見つけた位置の整数値（文字列が見つからない場合は 0）	1 番目引数は検索する文字列，2 番目引数は検索される文字列を示します。任意となる 3 番目の引数は，検索を開始する文字の位置を示します（デフォルトは先頭から検索）。文字列の先頭位置は 1 です。
算術関数	ABS	関数の引数と同じ型（Integer，Float，または Double）の絶対値	引数として数値を渡します。

分類	関数	返り値	引数についての補足説明
算術関数	SQRT	引数で渡された数値の平方根（実数）	引数として数値を渡します。
	MOD	引数で渡された二つの数値の剰余（整数）	二つの整数を渡します。
	SIZE	コレクションの要素数（整数）	コレクションを渡します。
日付時間関数	CURRENT_DATE	データベースの日付	－
	CURRENT_TIME	データベースの時刻	－
	CURRENT_TIMESTAMP	データベースのタイムスタンプ	－

（凡例）－：該当しない

(3)　注意事項

WHERE 節の注意事項について説明します。

(a)　演算子の優先順位

演算子の優先順位を次に示します。

1. ピリオド（.）
2. 算術演算子
   単項演算（+，-），乗算と除算（*，/），加算と減算（+，-）
3. 比較演算子
   =，>，>=，<，<=，<>（not equal），[NOT] BETWEEN，[NOT] LIKE，[NOT] IN，IS [NOT] NULL，IS [NOT] EMPTY，[NOT] MEMBER [OF]
4. 論理演算子
   NOT，AND，OR

(b)　条件式の注意事項

- 条件式は，条件式，比較式，論理演算子，評価結果が boolean 値になるパス式，boolean リテラル，boolean 型の入力パラメタで構成されます。
- 算術式は，比較式で使用できます。算術式は，ほかの算術式，四則演算，結果が数値になるパス式，数値リテラル，数値型の入力パラメタで構成されます。
- 算術操作は，数値昇格（numeric promotion）を使用します。
- 式の評価順序を示すために括弧で囲むことができます。
- 集合関数は，HAVING 節の条件式だけで使用できます。
- 条件式の中では，シリアライズ形式または lobs としてマップされるフィールドを使用しないでください。

(c) リテラルの注意事項

- 文字リテラルは，シングルクォテーションで囲みます（例：'literal'）。シングルクォテーションを含む文字列リテラルは，二つシングルクォテーションを使用します。
- JavaString リテラルのように，クエリの文字列リテラルは Unicode 文字のエンコードを使用します。
- Java のエスケープ表記の使用は，クエリの文字列リテラルではサポートしません。
- enum のリテラルは，Java の enum リテラル構文のリテラルが使用できます。enum リテラルには enum クラス名が必要です。
- boolean リテラルは，TRUE と FALSE である。このリテラルは大文字小文字を区別しません。

(d) 識別変数の注意事項

- 識別変数は, クエリの FROM 節で宣言された識別子であるため, ほかの節では宣言できません。FROM 節以外で宣言した場合は例外が発生します。SELECT 文，または DELETE 文の WHERE か HAVING 節で使用されるすべての識別変数は，FROM 節で定義されたものを使用します。
- 識別変数の範囲は，WHERE 節と HAVING 節で決定されます。識別変数は，エンティティの抽象スキーマタイプのコレクションメンバやインスタンスは指定できますが，コレクション自体は指定できません。コレクションを指定した場合は例外が発生します。

(e) Path 式の注意事項

WHERE か HAVING 節内でのコレクションのパス式で，条件式の一部として IS [NOT] EMPTY 式や [NOT] MEMBER [OF]式，または SIZE 操作への引数以外を使用しないでください。

(f) 入力パラメタの注意事項

- 入力パラメタは，クエリの WHERE 節か HAVING 節で使用できます。
- 一つのクエリで位置パラメタと名前付きパラメタを混在して使用しないでください。混在した場合の動作は保証されません。
- 入力パラメタの値が null の場合, 入力パラメタを含む比較操作か算術操作で返される値は unknown になります。

なお，位置パラメタと名前付きパラメタの使い方については，「6.16.1(2)　パラメタの指定方法」を参照してください。

6.17.6　GROUP BY 節および HAVING 節

GROUP BY 節では，クエリの結果をグループごとにまとめます。また，HAVING 節では，クエリ結果をさらに絞り込むための条件指定ができます。グループを指定した上で，HAVING 節の条件を指定してください。

クエリに WHERE 節と GROUP BY 節の両方が含まれる場合，WHERE 節が最初に実行され，その後，GROUP BY 節で形式を整えて HAVING 節に従ってフィルタリングします。

SELECT 節に現れる集合関数以外のアイテムは，GROUP BY 節にも指定する必要があります。グルーピングでは, null 値も含めて条件として扱います。GROUP BY 節および HAVING 節の注意事項について説明します。

- エンティティによるグルーピングはできますが，シリアライズしたフィールドや，lob フィールドを含むことはできません。CJPA プロバイダでは，指定した場合には例外が発生します。

- HAVING 節にはグループアイテムに対する検索条件を指定するため，グループアイテムに適用する集合関数を指定する必要があります。CJPA プロバイダでは，検索条件の指定がない場合は例外が発生します。
- GROUP BY 節がない状態で HAVING 節は使用しないでください。使用した場合は例外が発生します。

GROUP BY 節および HAVING 節の記述例を次に示します。

```
SELECT e.department.departmentId
FROM Employee AS e
GROUP BY e.department.departmentId
HAVING COUNT(e.department.departmentId) <= 2
```

なお，GROUP BY 節および HAVING 節の構文については，「付録 D　JPQL の BNF」を参照してください。

6.17.7　ORDER BY 節

ORDER BY 節ではオブジェクトや値が順序付けされてクエリの結果を返します。ORDER BY 節の記述例を次に示します。

```
SELECT e
FROM Employee AS e
ORDER BY e.monthlySalary DESC
```

ORDER BY 節について説明します。

- 一つ以上の order 項目が指定された場合，orderby 項目要素の左から右へ順に優先度を決め，いちばん左の orderby 項目がいちばん高い優先度を持ちます。
- ASC は昇順のときに，DESC は降順のときに指定します。なお，デフォルト値は ASC です。
- null 値がある場合の順番は SQL のルールを適用します。
- ORDER BY 節が使用された場合，クエリの結果の順番はクエリメソッドの結果に保存されます。

なお，ORDER BY 節の構文については，「付録 D　JPQL の BNF」を参照してください。

また，ORDER BY 節では次に示す条件を満たす必要があります。

- ORDER BY 節で指定する order 項目は順序付けができること。
- ORDER BY 節で指定する order 項目は SELECT 節の select 式からたどれること。

これらの条件に合わない場合，CJPA プロバイダでは例外が発生するおそれがあります。ただし，例外が発生しない場合でも動作は保証しません。

6.17.8　バルク UPDATE 文およびバルク DELETE 文

複数のレコードを一括して更新することをバルク更新といいます。バルク更新をするには，バルク UPDATE 文およびバルク DELETE 文を使用します。バルク UPDATE と DELETE の操作は，単体のエンティティクラスまたはサブクラスと合わせたエンティティクラスに適用されます。UPDATE 文では UPDATE 節に，DELETE 文では FROM 節に識別変数を定義して，1 種類のエンティティを指定します。

バルク UPDATE 文およびバルク DELETE 文を使用するときの注意時について説明します。

- delete 操作は，指定されたクラスとそのサブクラスのエンティティにだけ適用されます。関係するエンティティへのカスケードはされません。
- update 操作で指定される更新値（new_value）は，割り当てられたフィールドの型との互換性が必要です。互換性がない場合は例外が発生します。
- バルク UPDATE 文は，楽観的ロックを掛けないでデータベースの更新操作を実行するため，楽観的ロックに関する処理は実行しません。このため，バージョン列の値を手動で参照・更新する必要があります。

! 注意事項

バルク UPDATE 文やバルク DELETE 文を実行するときは，データベースと活性化している永続化コンテキストのエンティティとの間で不一致になるため注意が必要です。バルク UPDATE 文とバルク DELETE 文での操作は，トランザクションと切り離したところで実行するか，トランザクションの開始時に実行する必要があります。

バルク UPDATE 文およびバルク DELETE 文の記述例を次に示します。

```
UPDATE EMPLOYEE
SET MONTHLY_SALARY = MONTHLY_SALARY + 1000
WHERE DEPARTMENT_ID = 3

DELETE FROM EMPLOYEE e
WHERE e.EMPLOYEE_NAME IS NULL AND e.monthlySalary  IS EMPTY
```

なお，バルク UPDATE 文，バルク DELETE 文の構文については，「付録 D　JPQL の BNF」を参照してください。

6.17.9　JPQL 使用時の注意事項

ここでは，JPQL 使用時の注意事項について説明します。

(1)　null 値の注意事項

- **クエリ結果の null 値**
  - クエリの結果の値が，null 値を持つ関連フィールドまたはステートフィールドに対応する場合，その null 値がクエリメソッドの結果として返されます。
  - IS NOT NULL の構文は，クエリの結果の集合から null 値を除去するために使用できます。
  - Java の数値系プリミティブ型によって定義されるフィールドには，クエリの結果として null 値を生成できません。
- **比較，条件式での null 値**
  - 参照の対象がデータベースに存在しない場合，その値は null とみなされます。null 値を検索するには，比較条件 IS NULL および IS NOT NULL だけが使用できます。
  - null 値を IS NULL や IS NOT NULL 以外の条件で使用して，その結果が null 値に依存する場合，結果は unknown になります。
  - null 値の比較または算術操作の結果は，常に unknown 値になります。
  - 二つの null 値の比較結果は unknown 値になります。
  - unknown 値のある比較または算術操作の結果は，常に unknown 値になります。
  - boolean 演算子は 3 値論理を使用します。AND の場合，OR の場合，および NOT の場合の 3 値論理式をそれぞれ表に示します。

表 6-23　AND の 3 値論理式（A AND B の結果）

A	B		
	True	False	Unknown
True	True	False	Unknown
False	False	False	False
Unknown	Unknown	False	Unknown

表 6-24　OR の 3 値論理式（A OR B の結果）

A	B		
	True	False	Unknown
True	True	True	True
False	True	False	Unknown
Unknown	True	Unknown	Unknown

表 6-25　NOT の 3 値論理式（NOT A の結果）

A	NOT A の結果
True	False
False	True
Unknown	Unknown

(2)　HiRDB で JPQL を使用する場合の注意事項

- 次に示す JPQL の関数の引数に，位置パラメタや名前パラメタを指定することはできません。

  TRIM，SQRT，ABS，LENGTH，LOWER，MOD，LOCATE，UPPER，CONCAT，SUBSTRING，IS [NOT] NULL

  これらの関数の引数に位置パラメタや名前パラメタを指定した場合，動作は保証されません。なお，位置パラメタや名前パラメタを指定した場合は HiRDB の SQL 構文エラーとなり，SQLException 例外を含んだ PersistenceException 例外をスローすることがあります。

  これらの関数の機能を使用したい場合は，ネイティブクエリを使用してください。

- 四則演算子（＋，－，＊，／）の両側に？パラメタは指定できません。また，比較演算子（=，>，>=，<，<=，<>）の両側に？パラメタを指定したり，片側に？パラメタを指定したりしてもう一方にリテラルを指定することはできません。指定した場合の動作は保証されません。指定した場合は HiRDB の SQL 構文エラーとなり，SQLException 例外を含んだ PersistenceException 例外をスローすることがあります。

  JPQL の中で四則演算や比較演算しないで，JPQL を使用する前に四則演算や比較演算の操作をしてから JPQL を使用するようにしてください。

HiRDB で JPQL を使用する場合に使用できないクエリの記述例を次に示します。

使用できない例 1：関数の引数に位置パラメタを指定する

```
Query query1 = em.createQuery(
"SELECT  o FROM TestEntity o "+
```

```
                ”WHERE o.name=TRIM(LEADING FROM ?1)”)
              .setParameter(1, ”  SatoTaro”);
```

使用できない例 2：四則演算子に位置パラメタを指定する

```
int no_A=2;
int no_B=4;
Query query2 = em.createQuery(
”SELECT  o FROM TestEntity o WHERE o.id = ?1 + ?2”)
               .setParameter(1, no_A)
               .setParameter(2, no_B);
```

使用できない例 3：比較演算子に位置パラメタを指定する

```
int cmp_no=3;
Query query3 = em.createQuery(
”SELECT  o FROM TestEntity o WHERE o.id = ?1 AND ?1 < 9”)
               .setParameter(1, cmp_no);
```

6.17.10 クエリ使用時に発生する例外

CJPA プロバイダの場合，クエリ関連のアノテーションで文法的に間違いがあると，アプリケーションをデプロイするとき例外が発生します。また，クエリ関連のメソッドの引数が不正であったり，指定された文字列が有効な JPQL の文字列でなかったりすると，IllegalArgumentException 例外が発生するか，クエリの実行が失敗します。

ネイティブクエリを使用しているデータベースのクエリに対して有効でない場合，または定義された結果のセットがクエリの結果と互換性がない場合，クエリの実行は失敗します。CJPA プロバイダでは，クエリの実行時に PersistenceException 例外がスローされます。

(1) EntityManager 内のクエリ関連インタフェースの API で発生する例外

EntityManager 内のクエリ関連インタフェースの API で発生する例外を次に示します。

- createQuery メソッドのクエリ文字列が正しくない場合は，IllegalArgumentException 例外がスローされます。
- createNamedQuery メソッドに指定された名前でクエリが定義されていない場合は，IllegalArgumentException 例外がスローされます。

(2) Query インタフェースの API で発生する例外

Query インタフェースの API で発生する例外を次に示します。

- getResultList メソッドで，JPQL の UPDATE 文または DELETE 文が呼び出された場合は，IllegalStateException 例外がスローされます。
- getSingleResult メソッドで，クエリ結果がない場合は，NoResultException 例外がスローされます。なお，クエリ結果が二つ以上ある場合は，NonUniqueResultException 例外がスローされます。JPQL の UPDATE 文，または DELETE 文が呼び出された場合は，IllegalStateException 例外がスローされます。
- executeUpdate メソッドで，JPQL の SELECT 文が呼び出された場合は，IllegalStateException 例外がスローされます。このとき，トランザクションがない場合は，TransactionRequiredException 例外がスローされます。
- setHint メソッドの第 2 引数が実装に対して正しくない場合は，IllegalArgumentException 例外がスローされます。

- setParameter メソッドの引数の位置がクエリの位置パラメタと一致しない場合，または引数のパラメタ名がクエリ文字列のパラメタと一致しない場合は，IllegalArgumentException 例外がスローされます。
- setMaxResults または setFirstResult メソッドの引数が負数の場合は，IllegalArgumentException 例外がスローされます。

なお，API の詳細については，Java のドキュメントを参照してください。

6.18　persistence.xml の定義

この節では，persistence.xml の定義について，CJPA プロバイダ独自の機能であるエンティティオブジェクトのキャッシュ機能の定義と，データソースの指定の注意を説明します。

(1)　エンティティオブジェクトのキャッシュ機能の定義

CJPA プロバイダで提供するエンティティオブジェクトのキャッシュ機能の定義は，persistence.xml の<property>タグ内に指定します。persistence.xml でのエンティティオブジェクトのキャッシュ機能の定義について次の表に示します。

表 6-26　persistence.xml でのエンティティオブジェクトのキャッシュ機能の定義

指定するプロパティ	設定内容
cosminexus.jpa.cache.size.<ENTITY>	エンティティをキャッシュする場合のキャッシュサイズを指定します。
cosminexus.jpa.cache.size.default	エンティティをキャッシュする場合のキャッシュサイズのデフォルトを指定します。
cosminexus.jpa.cache.type.<ENTITY>	エンティティのキャッシュタイプを指定します。
cosminexus.jpa.cache.type.default	エンティティのキャッシュタイプのデフォルトを指定します。
cosminexus.jpa.target-database	接続するデータベースの名前を指定します。

タグの詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「6.2.2 <property>タグに指定できる CJPA プロバイダ独自のプロパティ」を参照してください。

参考

ここで説明しているプロパティは，CJPA プロバイダ独自のプロパティです。persistence.xml には，このほかに JPA 仕様で定義されたプロパティを指定することができます。ただし，CJPA プロバイダの場合，JPA 仕様で規定されている javax から始まるプロパティは使用できません。

(2)　データソースの指定の注意

persistence.xml でのデータソースの指定では，アプリケーションサーバの機能であるユーザ指定名前空間機能を使用して，リソースアダプタに別名を付けることができます。persistence.xml の設定でリソースアダプタに別名を設定した場合は，Connector 属性ファイルでもリソースアダプタの別名の定義が必要です。詳細については，「6.19　実行環境での設定」を参照してください。

6.19　実行環境での設定

CJPA プロバイダを使用する場合，J2EE サーバの設定および DB Connector の設定が必要です。

(1)　J2EE サーバの設定

J2EE サーバの設定は，簡易構築定義ファイルで実施します。CJPA プロバイダが出力するログファイルの設定は，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に指定します。

簡易構築定義ファイルでの CJPA プロバイダが出力するログファイルの設定について次の表に示します。

表 6-27　簡易構築定義ファイルでの CJPA プロバイダが出力するログファイルの設定

指定するパラメタ	設定内容
ejbserver.logger.channels.define.JPAOperationLogFile.filenum	稼働ログの面数を指定します。
ejbserver.logger.channels.define.JPAMaintenanceLogFile.filenum	保守ログの面数を指定します。
ejbserver.logger.channels.define.JPAOperationLogFile.filesize	稼働ログのサイズを指定します。
ejbserver.logger.channels.define.JPAMaintenanceLogFile.filesize	保守ログのサイズを指定します。
cosminexus.jpa.logging.level.operation.<category>	稼働ログのログレベルを指定します。

簡易構築定義ファイルおよび指定するパラメタの詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.6　簡易構築定義ファイル」を参照してください。

(2)　DB Connector の設定

DB Connector の設定は，サーバ管理コマンドおよび Connector 属性ファイルで実施します。

Connector 属性ファイルでの設定について次の表に示します。

表 6-28　Connector 属性ファイルでの設定

指定するタグ	設定内容
<resource-external-property>－<optional-name>タグ	persistence.xml のデータソースの指定でリソースアダプタの別名を使用している場合，リソースアダプタの別名を設定します。persistence.xml でリソースアダプタの別名を使用しない場合は設定不要です。
<resource-external-property>－<res-auth>タグ	persistence.xml のデータソースの指定でリソースアダプタの別名を使用している場合，リソースの認証方式として必ず「Container」を指定してください。
<resource-external-property>－<res-sharing-scope>タグ	persistence.xml のデータソースの指定でリソースアダプタの別名を使用している場合，リソース接続を共有するかどうかの指定として必ず「Shareable」を指定してください。

指定するタグ	設定内容
<property>－<property-name>タグ <property>－<property-value>タグ	データベースへの接続情報を指定します。 • ユーザ名の指定 <property-name>タグに User を，<property-value>タグにデータベース接続で使用するユーザ名を指定します。 • パスワードの指定 <property-name>タグに Password を，<property-value>タグにデータベース接続で使用するパスワードを指定します。
<connection-definition>－<transaction-support>タグ	トランザクションのサポートレベルを指定します。ここに指定する値は，persistence.xml の<persistence-unit>タグの transaction-type 属性に指定した値と合わせてください。

Connector 属性ファイルおよび指定するタグの詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1　Connector 属性ファイル」を参照してください。

6.20　アプリケーション開発時の注意事項

(1)　Cosminexus 09-60 以降の環境で@OneToOne および@ManyToOne での LAZY フェッチを使用する場合の注意事項

Cosminexus 09-60 以降の環境でアプリケーションを開発する場合，次の条件が重なるエンティティクラスは Java SE 5 または 6 のソースファイルとして作成してください。またアプリケーションコンパイル時に Java SE 5 または 6 のクラスファイルを生成するように設定し，コンパイルしてください。

- エンティティクラスに@OneToOne または@ManyToOne が含まれる。
- @OneToOne または@ManyToOne の fetch 属性に FetchType.LAZY を指定している。

注　javac コマンドを使用してコンパイルする場合，「-target」および「-source」を使用することで，Java SE 5 または 6 のクラスファイルを生成できます。

7 CJMS プロバイダ

この章では, CJMS プロバイダを使用してメッセージを送受信する機能について説明します。CJMS プロバイダは，JMS 仕様に準拠したメッセージの送受信を実現するための機能です。

7.1　この章の構成

この章では，CJMS プロバイダの機能について説明します。

この章の構成を次の表に示します。

表 7-1　この章の構成（CJMS プロバイダ機能）

分類	タイトル	参照先
解説	CJMS プロバイダの概要	7.2
	CJMSP リソースアダプタと CJMSP ブローカーの配置	7.3
	メッセージングモデルの種類	7.4
	メッセージの構成	7.5
	メッセージセレクターによる受信メッセージの選択	7.6
	メッセージ配信の高信頼性確保の仕組み	7.7
	CJMSP ブローカーの機能	7.8
	CJMSP リソースアダプタの機能	7.9
	Message-driven Bean の呼び出し	7.10
実装	アプリケーション実装時の制限事項	7.11
	DD での定義	7.12
設定	実行環境の構築の流れ	7.13
	CJMSP ブローカーの設定	7.14
	CJMSP リソースアダプタの設定	7.15
	J2EE アプリケーションの設定	7.16
運用	CJMS プロバイダを使用する場合のシステムの開始と停止	7.17
	障害対応用の情報の確認	7.18
注意事項	CJMS プロバイダ使用時の注意事項	7.19

7.2　CJMS プロバイダの概要

この節では，CJMS プロバイダの概要について説明します。

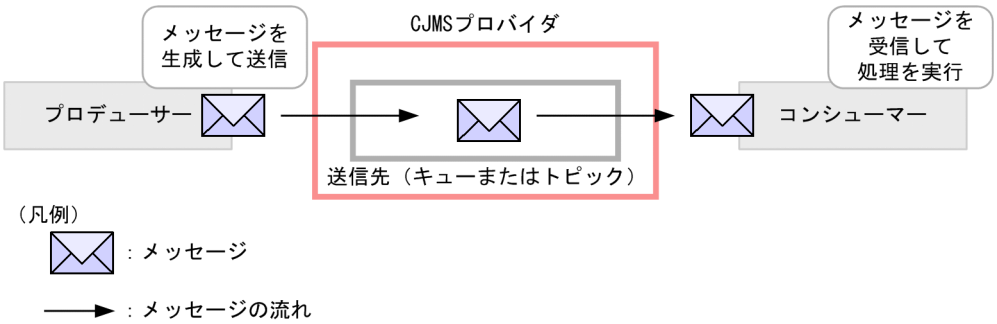
7.2.1　CJMS プロバイダとは

CJMS プロバイダは，メッセージの送受信を実現するためのアプリケーションサーバの機能です。JMS 1.1 の仕様に準拠しています。

CJMS プロバイダで送受信するメッセージは，**プロデューサー**によって作成され，送信されます。送信されたメッセージは，CJMS プロバイダが管理する**送信先**（キューまたはトピック）に登録されます。その後，**コンシューマー**に配信され，処理が実行されます。

CJMS プロバイダを使用したメッセージの送受信の概要を次の図に示します。

図 7-1　CJMS プロバイダを使用したメッセージの送受信の概要



送信先を介してメッセージをやり取りする仕組みによって，メッセージの送信側と受信側での非同期処理を実現します。

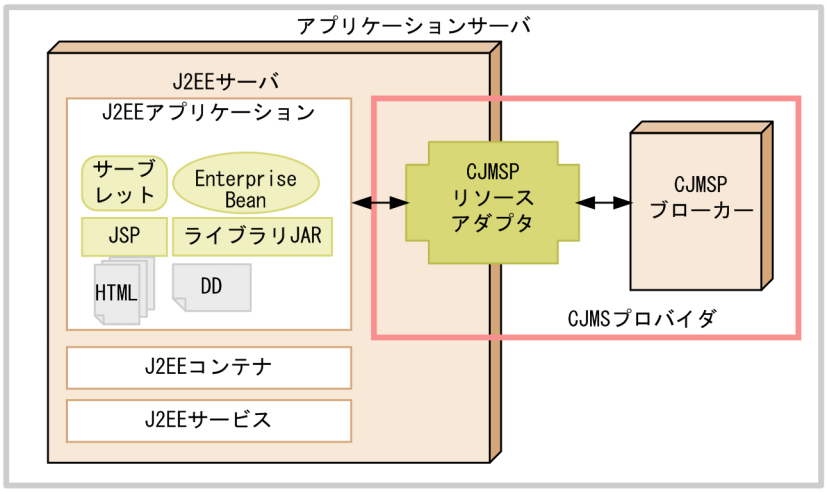
7.2.2　CJMS プロバイダのアプリケーションサーバ内での位置づけ

CJMS プロバイダは，CJMSP ブローカーと CJMSP リソースアダプタという二つのコンポーネントで構成されます。**CJMSP ブローカー**は，メッセージの送信先の管理をするためのコンポーネントです。J2EE サーバとは別のプロセスとして動作します。**CJMSP リソースアダプタ**は，J2EE サーバから CJMSP ブローカーに接続するためのリソースアダプタです。

これらのコンポーネントは，アプリケーションサーバだけで使用できます。

CJMS プロバイダのアプリケーションサーバ内での位置づけについて，次の図に示します。

図 7-2　CJMS プロバイダのアプリケーションサーバ内での位置づけ



CJMS プロバイダを使用する J2EE アプリケーションは，サーブレット，JSP，Enterprise Bean などで構成されます。J2EE アプリケーションは，CJMSP リソースアダプタを経由して，CJMSP ブローカーの機能を使用します。

CJMS プロバイダを使用してメッセージを送受信するプロデューサーおよびコンシューマーには，J2EE サーバ上の J2EE アプリケーションが該当します。メッセージの送信側であるプロデューサーでは，サーブレット，JSP，Enterprise Bean などが動作します。また，受信側であるコンシューマーでは，Message-driven Bean が動作します。送信側と受信側の J2EE サーバが別のマシンにある場合は，両方のマシンに CJMSP リソースアダプタが必要です。

なお，JMS の機能を使用する J2EE アプリケーションを **JMS アプリケーション**ともいいます。

7.2.3　CJMS プロバイダの機能の概要

CJMS プロバイダでは，JMS 仕様に準拠したメッセージ送受信を実現するための機能や，送受信を効率的に実現するための管理機能や監視機能などを提供しています。

CJMS プロバイダの機能の概要を次の表に示します。機能の詳細については，参照先の説明を参照してください。

表 7-2　CJMS プロバイダの機能

機能	概要	参照先
JMS 仕様に準拠したメッセージの送受信	PTP メッセージングモデルまたは Pub/Sub メッセージングモデルで JMS メッセージを送受信します。Pub/Sub メッセージングモデルでは，永続化サブスクライバーも使用できます。 受信側では，メッセージセレクターを使用して受信するメッセージを選択できます。	**メッセージングモデルの種類** 7.4 **メッセージの構成** 7.5 **メッセージセレクターによる受信メッセージの選択** 7.6

機能	概要	参照先
メッセージ配信の高信頼性確保の仕組み	JTA トランザクションの利用や，メッセージの種類ごとの流量制御の実施によって，メッセージ配信の高信頼性を確保します。	7.7
CJMSP ブローカーの機能	メッセージの送信先を管理します。また，永続化したメッセージも管理します。 そのほか，CJMSP ブローカーは，コネクションやルーティングの管理や，性能監視などをするための機能も備えています。	7.8
CJMSP リソースアダプタの機能	J2EE サーバ上の J2EE アプリケーションと CJMSP ブローカーを接続します。	7.9
Message-driven Bean の呼び出し	Message-driven Bean をメッセージのコンシューマーとして，業務を実行します。	7.10

このほか，CJMS プロバイダでは，実運用時に必要な運用管理のための機能や障害に対応するための機能も備えています。これらの機能によって，信頼性の高いメッセージングシステムを構築・運用できます。

7.3　CJMSP リソースアダプタと CJMSP ブローカーの配置

この節では，CJMSP リソースアダプタと CJMSP ブローカーの配置について説明します。CJMSP リソースアダプタと CJMSP ブローカーは，同じマシンまたは異なるマシンに配置できます。

一つの CJMSP ブローカーに対して，CJMSP リソースアダプタは複数配置できます。ただし，一つの J2EE サーバにデプロイできる CJMSP リソースアダプタは一つです。

CJMSP リソースアダプタおよび CJMSP ブローカー以外のコンポーネントの配置については，マニュアル「アプリケーションサーバ システム設計ガイド」の「3.8　サーバ間で非同期通信をする場合の構成を検討する」を参照してください。なお，CJMSP ブローカーが管理する永続化メッセージは，CJMSP ブローカーと同じマシンにファイルとして保存されます。

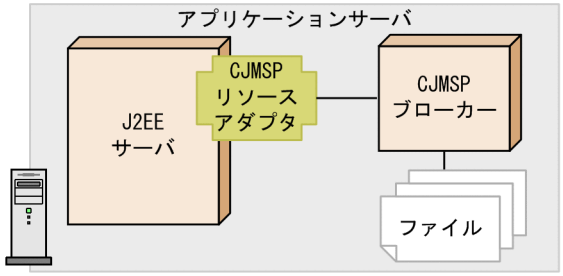
7.3.1　一つの CJMSP リソースアダプタに対して一つの CJMSP ブローカーを配置する構成

CJMSP リソースアダプタと CJMSP ブローカーを一つずつ配置する構成です。CJMSP ブローカーは，J2EE サーバと同じマシンまたは異なるマシンに配置します。CJMSP ブローカーが管理する永続化メッセージおよび管理情報はファイルで管理されます。

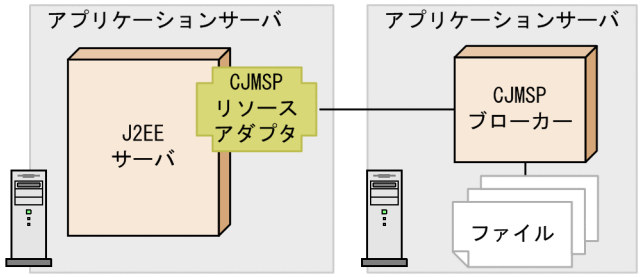
一つの CJMSP リソースアダプタに対して一つの CJMSP ブローカーを配置する構成を次の図に示します。

図 7-3　一つの CJMSP リソースアダプタに対して一つの CJMSP ブローカーを配置する構成

●CJMSPリソースアダプタとCJMSPブローカーを同じマシンに配置した場合



●CJMSPリソースアダプタとCJMSPブローカーを異なるマシンに配置した場合



7.3.2　複数の CJMSP リソースアダプタに対して一つの CJMSP ブローカーを配置する構成

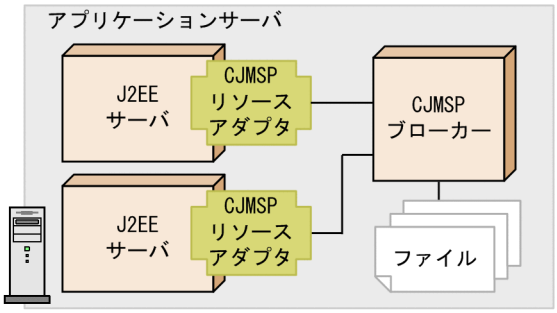
複数の CJMSP リソースアダプタに対して一つの CJMSP ブローカーを配置する構成です。CJMSP ブローカーは，J2EE サーバと同じマシンまたは異なるマシンに配置します。CJMSP リソースアダプタは，J2EE サーバごとに一つデプロイできます。

この構成では，複数の CJMSP リソースアダプタによって，一つの CJMSP ブローカーが管理する送信先やリソースが共有されます。CJMSP ブローカーが管理する永続化メッセージおよび管理情報はファイルで管理されます。

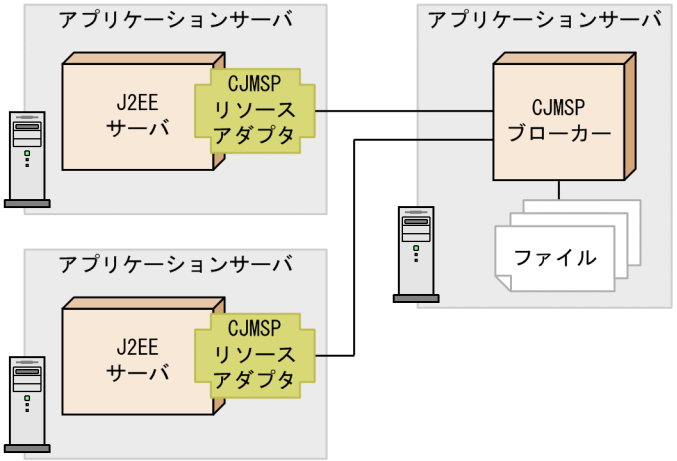
複数の CJMSP リソースアダプタに対して一つの CJMSP ブローカーを配置する構成を次の図に示します。

図 7-4　複数の CJMSP リソースアダプタに対して一つの CJMSP ブローカーを配置する構成

●CJMSPリソースアダプタとCJMSPブローカーを同じマシンに配置した場合



●CJMSPリソースアダプタとCJMSPブローカーを異なるマシンに配置した場合



7.4　メッセージングモデルの種類

メッセージの送受信の方法は，メッセージングモデルによって異なります。

CJMS プロバイダでは，次の 2 種類のメッセージングモデルに対応しています。

- PTP メッセージングモデル
- Pub/Sub メッセージングモデル

ここでは，それぞれのメッセージングモデルでのメッセージ送受信の概要について説明します。

なお，これらのメッセージングモデルは，JMS 仕様に準拠しています。

7.4.1　PTP メッセージングモデル

PTP メッセージングモデルは，ポイント・ツー・ポイント（Point-to-Point）方式でメッセージを送受信するためのモデルです。

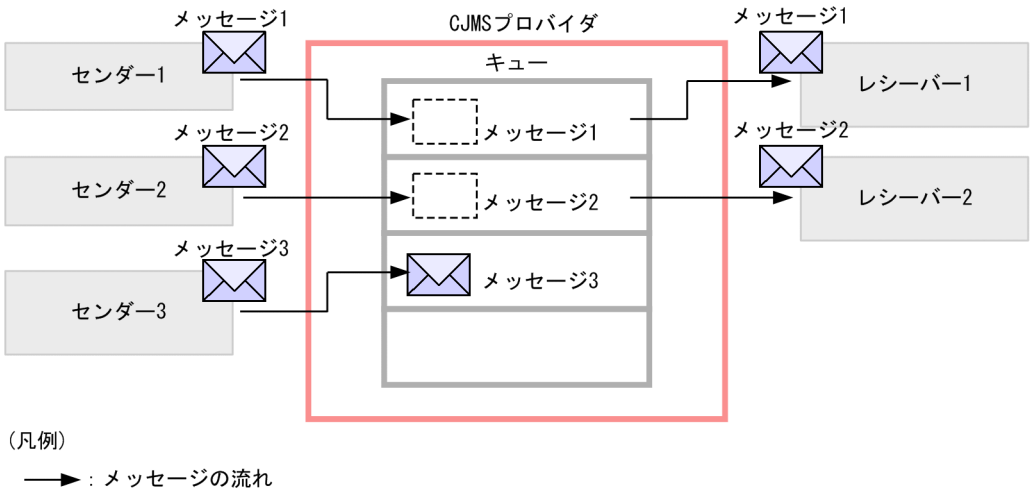
(1)　PTP メッセージングモデルによるメッセージの送受信

PTP メッセージングモデルでは，メッセージを作成して送信する送信側のクライアント（プロデューサー）を**センダー**といいます。また，メッセージを受信する側のクライアント（コンシューマー）を**レシーバー**といいます。

センダーから送信されたメッセージは，**キュー**という送信先に登録されます。キューに格納されたメッセージは，レシーバーが受け取ると，キューから削除されます。

PTP メッセージングモデルでのメッセージの流れを次の図に示します。

図 7-5　PTP メッセージングモデルでのメッセージの流れ



メッセージは，センダーから送信され，キューに登録されます。キューに登録されたメッセージは，その時に起動しているレシーバーのどれかに配信されます（メッセージ 1，メッセージ 2）。レシーバーに配信されたメッセージは，キューから削除されます。配信先のレシーバーがない場合，メッセージはキューに滞留します（メッセージ 3）。

なお，メッセージが登録されたキューの状態は，JMS 仕様で規定されているキューブラウザーを使用して確認できます。キューブラウザーについては，JMS 仕様を参照してください。

(2)　PTP メッセージングモデルの特徴

PTP メッセージングモデルの特徴を次に示します。

メッセージの送信～処理実行までの特徴

PTP メッセージングモデルでのメッセージの送信から処理実行までの特徴について説明します。

- 一つまたは複数のセンダーからキューにメッセージを送信できます。
- レシーバーは，メッセージを送信したセンダーがどれかに関係なく，メッセージを受け取り，処理します。
- 一つのメッセージは，一つのレシーバーだけで処理されます。
- センダーとレシーバーが処理を実行するタイミングに依存性はありません。センダーがメッセージを送信したときにレシーバーが起動していなくても，レシーバーは次に起動したときにメッセージを受け取ることができます。
- 複数のレシーバーが同じキューからメッセージを受け取る場合に，メッセージの順序性が重要でないときには，それぞれのレシーバーで処理が実行されます。
- メッセージは，センダーが送信した順序でキューに登録されます。
- メッセージの処理順序は，メッセージの有効期限，メッセージに設定された優先順位，およびレシーバーが使用しているメッセージセレクターによって決定され，この順序に従ってレシーバーが呼び出されます。
- レシーバーがまったく起動していない場合，送信されたメッセージはキューで保管されます。

システムの特徴

- センダーとレシーバーは，動的に追加・削除できます。これによって，使用状況に応じてシステムを拡張，縮小できます。
- CJMS プロバイダとの接続で使用するコネクションは，複数のセンダーで共有できます。また，CJMS プロバイダが管理するコネクションは，複数のレシーバーで共有できます。例えば，センダー 1 とセンダー 2 でコネクションを共有したり，レシーバー 1 とレシーバー 2 でコネクションを共有したりできます。コネクションについては，「7.8.1　コネクションサービス」を参照してください。

7.4.2　Pub/Sub メッセージングモデル

Pub/Sub メッセージングモデルは，パブリッシュ・サブスクライブ（Publish-Subscribe）方式でメッセージを送受信するためのモデルです。

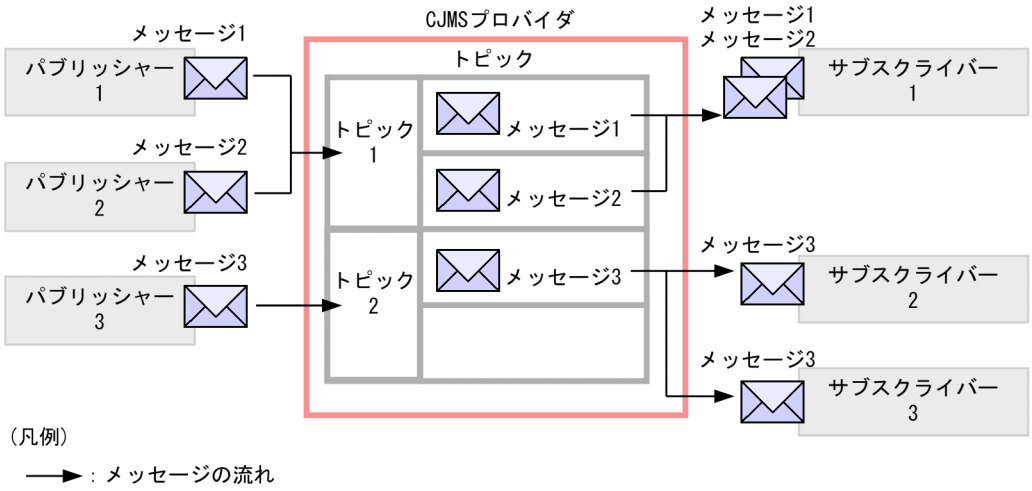
(1)　Pub/Sub メッセージングモデルによるメッセージの送受信

Pub/Sub メッセージングモデルでは，メッセージを作成して送信する送信側のクライアント（プロデューサー）を**パブリッシャー**といいます。また，メッセージを受信する側のクライアント（コンシューマー）を**サブスクライバー**といいます。

パブリッシャーから送信されたメッセージは，**トピック**という送信先に登録されます。トピックに登録されたメッセージは，そのトピックに対して配信を申し込んでいた一つまたは複数のサブスクライバーに配信されます。

Pub/Sub メッセージングモデルでのメッセージの流れを次の図に示します。

図 7-6　Pub/Sub メッセージングモデルでのメッセージの流れ



メッセージは，パブリッシャーから送信され，トピックに登録されます。図の場合，トピック 1 にはサブスクライバー 1，トピック 2 にはサブスクライバー 2 とサブスクライバー 3 が登録されています。このとき，トピック 1 に登録されたメッセージ 1 とメッセージ 2 はサブスクライバー 1 に配信されます。トピック 2 に登録されたメッセージ 3 はサブスクライバー 2 とサブスクライバー 3 に配信されます。

(2)　Pub/Sub メッセージングモデルの特徴

Pub/Sub メッセージングモデルの特徴を次に示します。

メッセージの送信～処理実行までの特徴

- 一つまたは複数のパブリッシャーから，メッセージをトピックに登録できます。
- 一つまたは複数のサブスクライバーが，トピックからメッセージを取り出して処理できます。
- サブスクライバーは，配信を申し込んだトピックに登録されたすべてのメッセージについて，受信して処理できます。ただし，メッセージセレクターで設定した基準に該当しないメッセージ，または受信する前にメッセージの有効期限が過ぎたメッセージについては受信できません。
- メッセージは，パブリッシャーが送信した順序でトピックに登録されます。ただし，サブスクライバーで処理される順序は，それぞれのメッセージの有効期限，優先順位，またはサブスクライバーで設定されたメッセージセレクターの内容によって決まります。
- パブリッシャーとサブスクライバーが処理を実行するタイミングには依存性があります。トピックに登録されたメッセージは，メッセージが登録される前に開始されていたサブスクライバーだけに配信されます。
- サブスクライバーの属性に「NoLocal」を指定した場合，サブスクライバーが使用しているコネクションと同じコネクションで送信されたメッセージの受信を抑止できます。この属性のデフォルトは「false」です。

システムの特徴

- パブリッシャーとサブスクライバーは動的に追加・削除できます。これによって，使用状況に応じてシステムを拡張，縮小できます。
- CJMS プロバイダとの接続で使用するコネクションは，複数のプロデューサーで共有できます。なお，コネクションを共有しているかどうかに関係なく，複数のプロデューサーから同じのトピックにメッセージを送信できます。

- CJMS プロバイダとの接続で使用するコネクションは，複数のサブスクライバーで共有できます。なお，コネクションを共有しているかどうかに関係なく，複数のサブスクライバーから同じトピックに接続できます。

(3)　永続化サブスクライバーの利用

トピックに登録されたメッセージは，メッセージ登録時に開始されていたサブスクライバーだけが受信できます。通常のサブスクライバーは，停止していた期間に登録されたメッセージは受信できません。

これに対して，サブスクライバーを永続化することで，停止していた期間に登録されたメッセージも受信できるようになります。永続化したサブスクライバーを**永続化サブスクライバー**といいます。

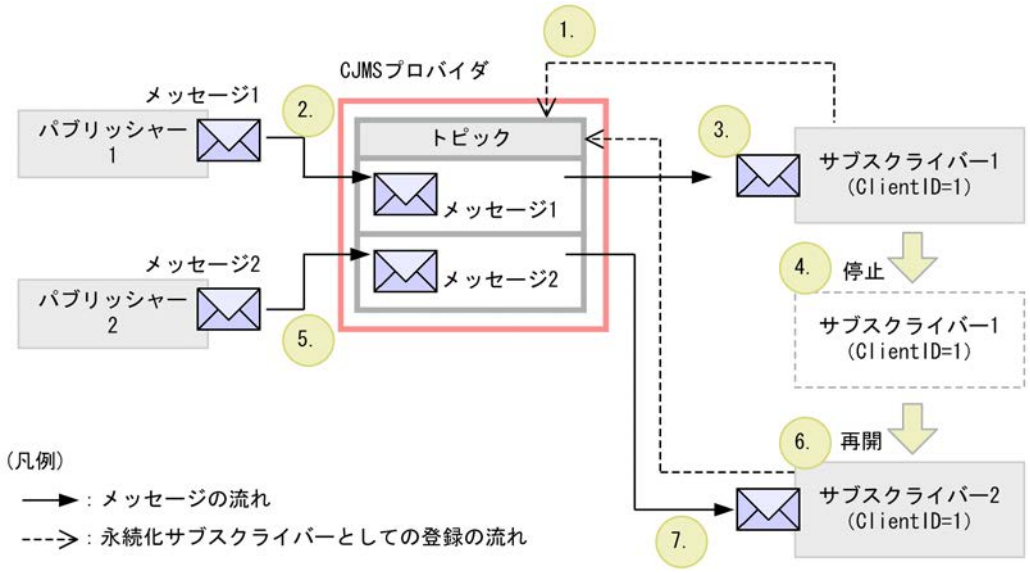
永続化サブスクライバーが登録されたトピックのメッセージは，次のどちらかの状態になるまで削除されません。

- 永続化サブスクライバーに配信された場合
- メッセージの有効期限が過ぎた場合

永続化サブスクライバーを利用する場合，そのサブスクライバーに固有の識別子と名称を登録します。登録した情報は，CJMSP ブローカーによって保持されます。永続化サブスクライバーが停止しているときに登録されたメッセージは，登録した識別子および名称のサブスクライバーが再開されたときに，そのサブスクライバーに配信されます。

永続化サブスクライバーを利用した場合のメッセージの送受信の流れを次の図に示します。

図 7-7　永続化サブスクライバーを利用した場合のメッセージの送受信の流れ



永続化サブスクライバーを利用した場合のメッセージの送受信の流れについて説明します。なお，説明の番号は図中の番号と対応しています。

1. サブスクライバー 1 を永続化サブスクライバーとしてトピックに登録します。
   識別子（ClientID）は，「1」とします。
2. パブリッシャー 1 が，メッセージ 1 をトピックに登録します。
3. サブスクライバー 1 は，メッセージ 1 をトピックから受信します。

4. サブスクライバー 1 を停止します。
   停止した状態になると，トピックに登録されたメッセージは受信できません。
5. パブリッシャー 2 が，メッセージ 2 をトピックに登録します。サブスクライバー 1 は停止中のため，メッセージ 2 を受信できません。ただし，サブスクライバー 1 は永続化サブスクライバーとして登録されているため，このメッセージは登録された永続化サブスクライバーがメッセージを受信するまでトピックで保管されます。
6. サブスクライバー 2 を永続化サブスクライバーとしてトピックに登録します。このとき，識別子（ClientID）を「1」として，サブスクライバーの名称はサブスクライバー 1 と同じ値にします。
   永続化サブスクライバーの情報を保持している CJMSP ブローカーは，識別子と名称からサブスクライバー 2 は登録済みの永続化サブスクライバーが再開したものと判断します。
7. トピックに保管されていたメッセージ 2 がサブスクライバー 2 に配信されます。

永続化サブスクライバーの指定を解除する場合は，次のどちらかの方法で解除してください。

- cjmsicmd destroy dur コマンドを実行する
- unsubscribe メソッドを使用する

cjmsicmd destroy dur コマンドについては，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjmsicmd destroy dur（永続化サブスクライバーの破棄)」を参照してください。

unsubscribe メソッドを使用すると，サブスクライバーのために保持されたトピックの状態を解除できます。ただし，このメソッドは，次のタイミングでは使用しないでください。

- 永続化サブスクライバーの指定を解除する対象のトピックに対して，開始されているサブスクライバーがある場合
- トピックに配信されたメッセージが，未決着のトランザクションである場合
- そのセッションでの配信が承認されていない場合

なお，確認応答モードが「CLIENT_ACKNOWLEDGE」のセッションによって永続化サブスクライバーが作成された場合に，メッセージ受信が確認されていない状態で unsubscribe メソッドが実行されたときには，メッセージは送信先に残ってしまいます。この状況を避けるため，unsubscribe メソッドを使用する場合，事前に cjmsicmd purge dur コマンドを実行して，永続化サブスクライバーに関連づけられているすべてのメッセージを削除してください。

! 注意事項

一時的な送信先として作成したトピックに対して，永続化サブスクライバーは登録できません。作成しようとすると，例外がスローされます。

7.5　メッセージの構成

CJMS プロバイダで扱うメッセージは，次に示す要素で構成されます。

- メッセージヘッダー
- メッセージプロパティ
- メッセージボディ

これらの要素については，JMS の仕様として規定されています。詳細は，JMS の仕様を参照してください。

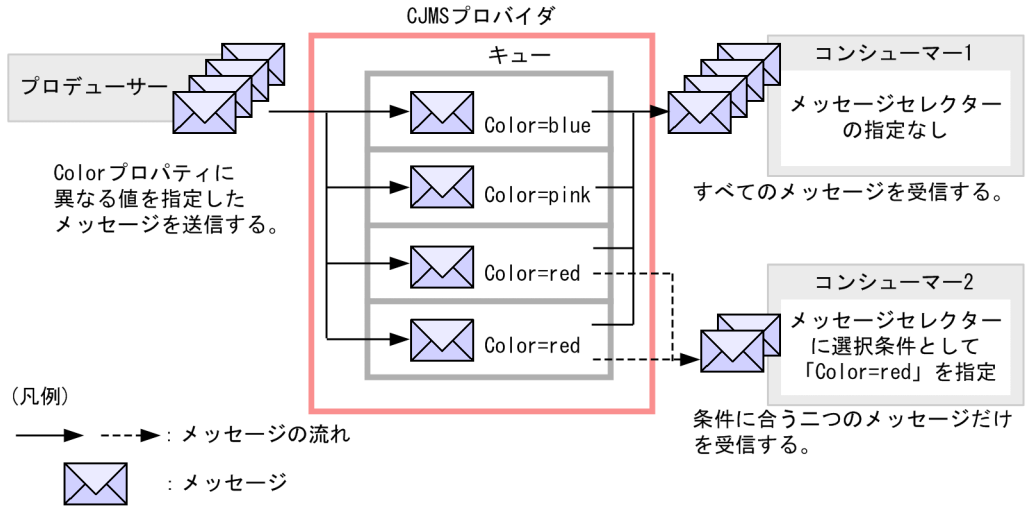
7.6　メッセージセレクターによる受信メッセージの選択

コンシューマーは，受信するメッセージを**メッセージセレクター**によって選択できます。メッセージセレクターを使用したコンシューマーでは，送信先に登録されたメッセージのうち，必要なものだけを受信できます。

メッセージセレクターでは，選択対象の指定にメッセージプロパティの値を使用します。コンシューマーは，メッセージセレクターに指定した選択条件とメッセージプロパティの値を比較し，条件に一致するメッセージだけを受信します。

メッセージセレクターを使用した受信メッセージの選択の概要を次の図に示します。

図 7-8　メッセージセレクターを使用した受信メッセージの選択の概要



この図では，コンシューマー 2 に「Color=red」という選択条件を指定しています。コンシューマー 1 にはメッセージセレクターは指定しません。

プロデューサーから Color プロパティが異なる四つのメッセージを送信した場合，メッセージセレクターを使用していないコンシューマー 1 はすべてのメッセージを受信します。コンシューマー 2 は選択条件に一致する二つのメッセージだけを受信します。

なお，メッセージセレクターの選択条件は，コンシューマーで動作する Message-driven Bean の属性，または JMS の API で指定します。属性の詳細については，「7.12　DD での定義」を参照してください。また，選択条件の構文および API については，JMS 仕様を参照してください。

参考

メッセージセレクターは，キューブラウザーでも使用できます。

7.7　メッセージ配信の高信頼性確保の仕組み

この節では，CJMS プロバイダを使用したメッセージ配信で高信頼性を確保するための仕組みについて説明します。

7.7.1　メッセージ配信時に発生するトラブルの種類と信頼性を確保する方法

メッセージは，次の 2 段階で配信されます。

1. プロデューサーから CJMSP ブローカーが管理する送信先への配信
2. CJMSP ブローカーが管理する送信先からコンシューマーへの配信

この間，次のタイミングでトラブルが発生した場合に，メッセージが失われるおそれがあります。

- メッセージ転送時
  - プロデューサーから CJMSP ブローカーが管理する送信先へのメッセージ転送時
  - CJMSP ブローカーが管理する送信先からコンシューマーへのメッセージ転送時
- CJMSP ブローカーの障害発生時

これらの場合に信頼性を確保する方法について，次に示します。

- **メッセージ転送時にトラブルが発生した場合**

  メッセージ転送時のメッセージの信頼性を確保する方法としては，JMS 仕様で規定されている，メッセージ配信の確認応答（Acknowledge Mode）の仕組みを使用できます。

  また，トランザクションの利用によって，メッセージの送信から受信までにトラブルが発生した場合に，システムの整合性を保つことができます。
- **CJMSP ブローカーに障害が発生した場合**

  CJMSP ブローカーに障害が発生した場合に送信先に登録されていたメッセージを保持するためには，メッセージを永続化しておくことが必要です。CJMS プロバイダでは，メッセージをファイルに保存して永続化できます。

この節では，これらの方法のうち，トランザクションの利用によって信頼性を確保する方法について，「7.7.2　トランザクションの利用」で説明します。

また，メッセージ配信時には，送受信されるメッセージの量によって，メモリリソースの問題が発生することもあります。メモリリソースを適切に運用するためにメッセージの流量を制御する方法については，「7.7.3　メッセージの流量制御」で説明します。

なお，メッセージ配信の確認応答の詳細については，JMS 仕様を参照してください。なお，CJMSP ブローカーによるファイルの永続化については，「7.8.4　管理情報およびメッセージの永続化サービス」で説明します。

7.7.2　トランザクションの利用

トランザクションを利用すると，メッセージの作成から処理の実行までを一連の処理としてまとめることができます。トランザクションに含まれるすべての操作は，クライアント側のアプリケーション（メッセージを送信したアプリケーション）がトランザクションをコミットしたときに完了します。

CJMS プロバイダでは，JTA トランザクションを利用できます。

JTA トランザクションは，アプリケーションサーバのトランザクション管理用 API（JTA）によって管理されます。トランザクション内の操作が失敗した場合，アプリケーションサーバによって例外が処理され，トランザクション内の処理がリトライまたはロールバックされます。ローカルトランザクションの管理には，BMT または CMT を使用できます。BMT または CMT によるトランザクションの管理については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「2.7　Enterprise Bean でのトランザクション管理」を参照してください。

! 注意事項

CJMS プロバイダでは，JTA トランザクションを利用する場合，コンテナが Bean の代わりに JMS セッションのトランザクションを管理します。このため，次のメソッドの引数「transacted」の値に「true」は指定しないでください。

- createSession(boolean transacted, int acknowledgeMode)メソッド
- createQueueSession(boolean transacted, int acknowledgeMode)メソッド
- createTopicSession(boolean transacted, int acknowledgeMode)メソッド

「true」を指定した場合は，無視され，警告メッセージが出力されます。

なお，EJB アプリケーションを使用する場合，トランザクション内で JMS の承認メソッドを使用しないでください。承認メソッドを指定していないトランザクションコンテキスト内では，コンテナによってメッセージの承認が管理されます。

また，一つのトランザクション内で，大量のメッセージを送信すると,メッセージの配信確認に関する情報を管理しているファイルの容量が増え，ディスク容量を圧迫するおそれがあります。大量のメッセージを送信する場合，性能テストを実施して，必要なファイルサイズを見積もってください。

7.7.3　メッセージの流量制御

ここでは，メッセージの流量制御について説明します。

CJMS プロバイダを使用したメッセージの送受信では，プロデューサーからコンシューマーに送信するメッセージのほか，CJMS プロバイダが使用する制御用のメッセージも送受信されます。これらのメッセージの送受信は，お互いに影響します。例えば，プロデューサーからコンシューマーに送信したメッセージに対する CJMSP ブローカーでの確認応答の承認が遅れた場合などは，制御用のメッセージの送受信も遅れてしまうことになり，システム全体のパフォーマンスが低下してしまいます。

コネクション上のこれらのメッセージの流量を制御することで，メッセージ送受信の信頼性を高め，システムのパフォーマンスを向上させることができます。

CJMS プロバイダでは，メッセージの流量を制御するために，コンシューマーごとに送信するメッセージの流量を制限できます。

例えば，PTP メッセージングモデルでメッセージを送受信している場合，それぞれのコンシューマーの処理速度が低いときには，コンシューマー単位に配信するメッセージを制限します。これによって，複数のコンシューマーを使用してラウンドロビン方式で処理を実行し，システム全体の処理性能を向上させることができます。コンシューマー単位のメッセージの流量には，複数のコンシューマーに分散してメッセージを配信するためのオーバーヘッドの増加との関係を考慮して，適切な値を決定してください。

コンシューマー単位の制限値（consumerFlowLimit）については，送信先ごとに cjmsicmd create dst コマンドで設定します。コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjmsicmd create dst（物理的送信先の作成）」を参照してください。なお，consumerFlowLimit の値は，コンシューマーで動作している Message-driven Bean 単位の同時実行数

（プールの最大インスタンス数）との関係も考慮して設定する必要があります。設定のパターンの例を次に示します。

「consumerFlowLimit の値＜ Message-driven Bean のプールの最大インスタンス数」と設定した場合

Message-driven Bean では，consumerFlowLimit で指定した数のスレッドで処理が実行されます。

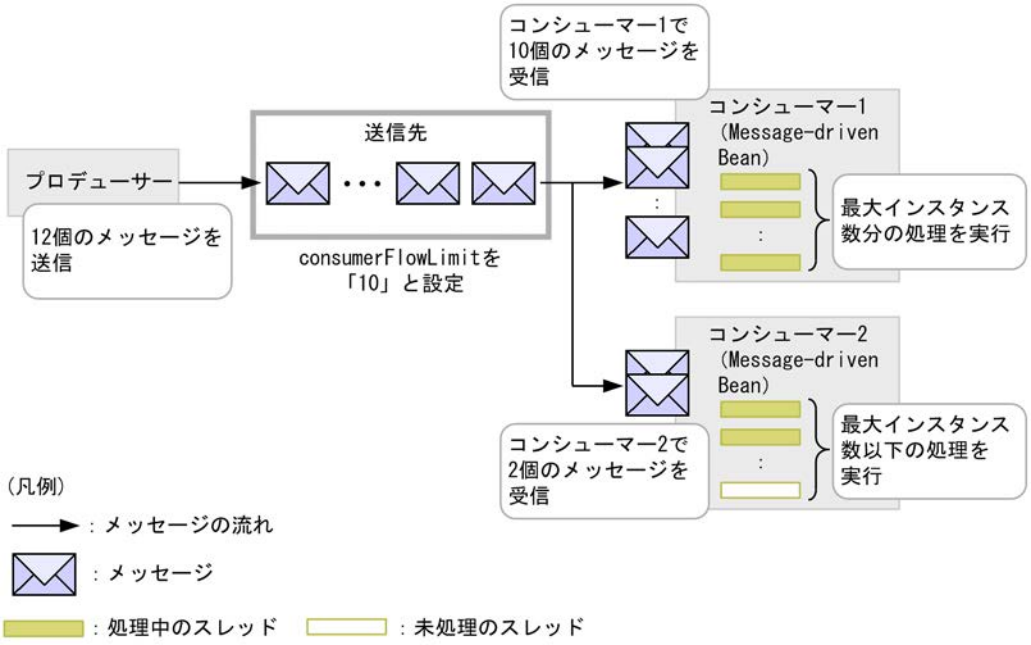
「consumerFlowLimit の値＝ Message-driven Bean のプールの最大インスタンス数」と設定した場合

プロデューサーから送信されたメッセージの数が consumerFlowLimit 以下の場合は，一つのコンシューマーですべての処理が実行されます。

プロデューサーから送信されたメッセージの数が consumerFlowLimit を超える場合は，複数のコンシューマーで処理が実行されます。

consumerFlowLimit の設定による流量制御の概要を次の図に示します。

図 7-9　consumerFlowLimit の設定による流量制御の概要



この例では，コンシューマー単位の制限値として，consumerFlowLimit を「10」と設定しています。プロデューサーから 12 個のメッセージが送信された場合，送信先からコンシューマー 1 には制限値である 10 個のメッセージが送信され，残りのメッセージはコンシューマー 2 に送信されます。各コンシューマーでは，Message-driven Bean の最大インスタンス数で設定した数以下のスレッドを使用して処理が実行されます。

7.8　CJMSP ブローカーの機能

この節では，CJMSP ブローカーの機能について説明します。

CJMSP ブローカーの主な機能を次に示します。

- コネクションサービス
- 送信先の管理とルーティングサービス
- CJMSP ブローカーの性能監視
- 管理情報とメッセージの永続化サービス

なお，このほか，CJMSP ブローカーはログ出力の機能も備えています。CJMSP ブローカーのログ出力については，「7.18　障害対応用の情報の確認」を参照してください。

7.8.1　コネクションサービス

コネクションサービスは，TCP 層上での CJMSP ブローカーと CJMSP リソースアダプタの接続（jms コネクション），および CJMSP ブローカーとシステムの管理ユーザの接続（admin コネクション）を管理するためのサービスです。

(1)　コネクションサービスの種類

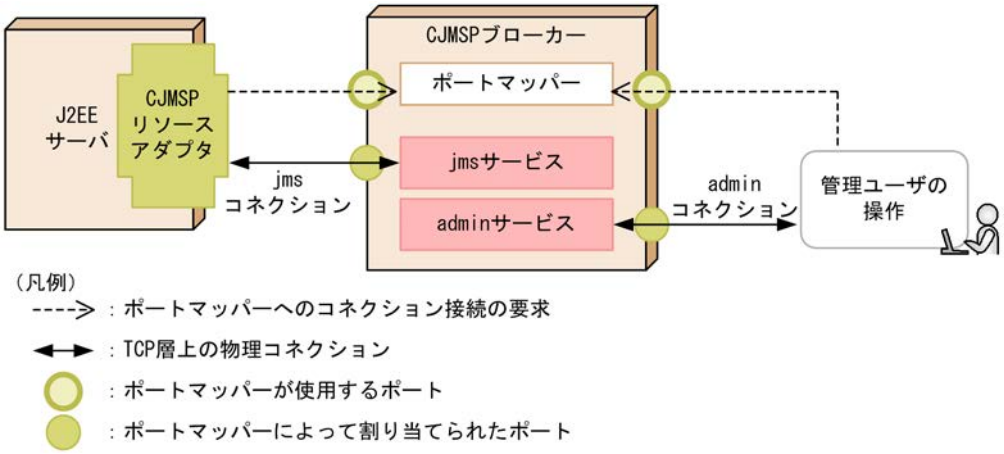
CJMS プロバイダが提供するコネクションサービスには，次の 2 種類があります。

- jms サービス
  CJMSP リソースアダプタから CJMSP ブローカーへの接続を管理するサービスです。メッセージを送受信する際, CJMSP リソースアダプタが CJMSP ブローカーに接続するときにこのサービスを使用します。
- admin サービス
  システムの管理ユーザから CJMSP ブローカーへの接続を管理するサービスです。管理ユーザがコマンドの実行によって次のような操作をする場合にこのサービスを使用します。
  - 物理的送信先の作成
  - 送信先または CJMSP ブローカーの問い合わせ

コネクションサービスでは，ポートマッパーによって静的または動的に割り当てられた，専用のポートを使用します。

ポートマッパーによるコネクションサービスへのポートの割り当ての概要を次の図に示します。

図 7-10　コネクションサービスへのポートの割り当ての概要



ポートマッパーは，デフォルトでは 7676 のポートを使用します。これ以外のポートを使用する場合は，CJMSP ブローカーの imq.portmapper.port プロパティで設定してください。

ポートの割り当ての流れを次に示します。

1. ポートマッパーに対して，CJMSP リソースアダプタから jms サービスでの接続をするためのリクエストを送信します。または管理ユーザから admin サービスでの接続をするためのリクエストを送信します。
2. ポートマッパーがコネクションサービスの要求に対して，ポートを動的に割り当てます。ただし，事前に CJMSP ブローカーのプロパティとして imq.jms.tcp.port プロパティまたは imq.admin.tcp.port プロパティを指定した場合は，コネクションサービスごとに指定したポートが割り当てられます。

なお，ポートマッパーがコネクションを接続するためのリクエストを複数同時に受け付けた場合，処理待ちのリクエストは OS のバックログに格納されて待機します。

コネクションサービスで使用する各プロパティの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「7.4　config.properties（CJMSP ブローカー個別プロパティファイル)」を参照してください。

(2)　コネクションサービスのスレッドプール管理

コネクションサービスはマルチスレッドで動作するため，複数のコネクションを同時に管理できます。コネクションサービスで使用するスレッドは，CJMSP ブローカーが管理するスレッドプールで管理されています。

なお，コネクションサービスのスレッドプールは，処理性能を向上させるために，独自の方式で管理されています。一つのコネクションでは，メッセージ受信用および送信用の二つのスレッドを使用します。

7.8.2　送信先の管理とルーティングサービス

ここでは，CJMSP ブローカーによる送信先の管理，およびルーティングサービスについて説明します。

CJMSP ブローカーは，物理的な送信先として，キューやトピックをシステムのメモリ内に作成・管理します。

ルーティングサービスは，リソースを効率的に使用しながら，プロデューサー，コンシューマーと送信先間での確実で円滑なメッセージの送受信を実現するためのサービスです。

(1) 送信先の種類

送信先は，作成方法および存在期間によって，次の 2 種類に分類できます。

- コマンドで作成する送信先
- API で作成される一時的な送信先

それぞれの送信先の特徴について説明します。

コマンドで作成する送信先

システムを管理するユーザがコマンドで明示的に作成した送信先です。

コマンドで作成した送信先は，明示的にコマンドで削除するまで削除されません。

API で作成される一時的な送信先

J2EE アプリケーション内の API によって作成される送信先です。プログラムの中でメッセージの送受信に必要な送信先が作成され，削除されます。一時的な送信先は，送信先を作成したコネクションの中だけで保持されます。また，メッセージの永続化はできず，CJMSP ブローカーを再起動しても再作成はされません。

! 注意事項

一時的送信先を使用するコンシューマーは，その一時的送信先を作成したコネクションだけで作成できます。

また，通常のメッセージ送受信に使用する送信先以外に，デッドメッセージキューという CJMSP ブローカーによって自動的に作成される送信先があります。

デッドメッセージキューは，CJMSP ブローカーを最初に起動したタイミングで作成されます。この送信先には，CJMSP ブローカーによって，次のメッセージ（**デッドメッセージ**）が登録されます。

- 処理できないメッセージ
- 有効期限を過ぎたメッセージ

デッドメッセージキューには，ほかの送信先で処理できずに廃棄されたメッセージが格納されるため，メッセージ送受信で発生した問題についての調査などに利用できます。

デッドメッセージキューの管理については，「(3)　デッドメッセージキューの管理」を参照してください。

(2) 送信先の設定と管理

CJMSP ブローカーでは，送信先を設定・管理するために，次の処理を実行できます。

- 送信先の作成・休止・再開・削除
- すべての送信先の一覧表示
- 個々のメッセージ，および CJMSP ブローカーで管理するメッセージ全体の制限値管理
- メッセージの最大数の管理
- 送信先の状態およびプロパティの表示
- 永続化メッセージを格納するディスクの圧縮
- デッドメッセージキューの管理

これらの処理は，コマンドで実行できます。詳細は，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「5.3　CJMSP ブローカー管理のコマンドの詳細」を参照してください。

(3)　デッドメッセージキューの管理

デッドメッセージキューは，すべての送信先で必ず使用する設定になっています。

メッセージがデッドメッセージキューに登録された場合の処理については，メッセージごとのメッセージプロパティに設定します。

メッセージプロパティに設定できる内容を次の表に示します。

表 7-3　デッドメッセージキューに登録する際の処理としてメッセージプロパティに設定できる内容

プロパティ	データ型	説明
CJMS_PRESERVE_UNDELIVERED※	Boolean	メッセージが配信できなかった場合の処理を設定します。 •「true」を設定した場合，メッセージはデッドメッセージキューに登録されます。 •「false」を設定した場合，メッセージはデッドメッセージキューには登録されません。
CJMS_LOG_DEAD_MESSAGES※	Boolean	メッセージが送信先から削除されてデッドメッセージキューに登録されたことをログに出力するかどうかを設定します。 •「true」を設定した場合，ログに出力します。 •「false」を設定した場合，ログに出力しません。
CJMS_TRUNCATE_MSG_BODY※	Boolean	デッドメッセージキューに登録するメッセージのメッセージボディを削除するかどうかを設定します。 •「true」を設定した場合，メッセージボディを削除してからデッドメッセージキューに登録します。 •「false」を設定した場合，メッセージボディを削除しないでデッドメッセージキューに登録します。

注※　CJMS プロバイダではサポートしていません。

また，デッドメッセージキューに登録されたデッドメッセージには，CJMSP ブローカーによってプロパティが設定されます。設定されたプロパティの情報によって，メッセージがデッドメッセージキューに登録された要因と，要因となった処理を確認できます。

CJMSP ブローカーによってデッドメッセージに設定されるプロパティについて，次の表に示します。

表 7-4　CJMSP ブローカーによってデッドメッセージに設定されるプロパティ

プロパティ	データ型	説明
JMSXDeliveryCount	Integer	特定のコンシューマーに対して配信されたメッセージの最大数が設定されます。 この値は，CJMS_DMQ_UNDELIVERED_REASON プロパティに ERROR または UNDELIVERABLE と設定されたプロパティの場合にだけ設定されます。

プロパティ	データ型	説明
CJMS_DMQ_UNDELIVERED_TIMESTAMP	Long	デッドメッセージキューに登録された時刻がミリ秒で設定されます。
CJMS_DMQ_UNDELIVERED_REASON	String	デッドメッセージキューに登録された要因を示す値が設定されます。設定されるのは次の値です。 • OLDEST 最も古いメッセージだったため処理されなかったことを示します。 • LOW_PRIORITY 優先順位が低かったため処理されなかったことを示します。 • EXPIRED メッセージの有効期限が過ぎたため処理されなかったことを示します。 • UNDELIVERABLE 配信されなかったことを示します。 • ERROR エラーが発生したことを示します。この要因が設定されたメッセージは,内部的な要因で処理ができなかったため,プロデューサーからメッセージを送信し直す必要があります。 なお，複数の要因が該当した場合は，どれか一つの要因だけが設定されます。
CJMS_DMQ_PRODUCING_BROKER	String	このメッセージを作成した CJMSP ブローカーの名前およびポート番号が設定されます。「null」が設定された場合は，ローカルで動作する CJMSP ブローカーであることを示します。
CJMS_DMQ_DEAD_BROKER	String	デッドメッセージキューを管理している CJMSP ブローカーの名前およびポート番号が設定されます。「null」が設定された場合は，ローカルで動作する CJMSP ブローカーであることを示します。
CJMS_DMQ_UNDELIVERED_EXCEPTION	String	例外の発生が原因で配信されなかったメッセージの場合に，アプリケーションまたは CJMSP ブローカーで発生した例外名が設定されます。
CJMS_DMQ_UNDELIVERED_COMMENT	String	メッセージが送信されなかった付加的な要因がある場合に設定されます。
CJMS_DMQ_BODY_TRUNCATED	Boolean	デッドメッセージキューへの登録時にメッセージボディが削除されたかどうかが設定されます。 「true」の場合，メッセージボディは削除されています。 「false」の場合，メッセージボディは削除されていません。

7.8.3 CJMSP ブローカーの性能監視

CJMSP ブローカーでは，CJMSP ブローカーの性能情報（メトリクス）を監視できます。

CJMSP ブローカーの性能情報には，ヒープサイズ，接続状況，受信メッセージの割合，送信メッセージの割合などの情報が含まれます。これらの情報は，コンソールに表示されて，ログファイルに出力されます。

CJMSP ブローカーの性能情報の出力方法は，imq.metrics.interval プロパティで指定します。このプロパティに指定した出力間隔で性能情報が出力されます。なお，デフォルトの設定は 0（出力しない）です。

CJMSP ブローカーの性能情報の出力例を次に示します。

```
KDAN24558-I Displaying broker metrics :
Connections: 10 JVM Heap: 13082624 bytes (1501448 free) Threads: 22 (14-1010)
In: 3001 msgs (631149 bytes) 4297 pkts (795219 bytes)
Out: 1253 msgs (263099 bytes) 5495 pkts (690645 bytes)
Rate In: 298 msgs/sec (62622 bytes/sec) 419 pkts/sec (77978 bytes/sec)
Rate Out: 125 msgs/sec (26271 bytes/sec) 540 pkts/sec (66573 bytes/sec)
```

出力項目について，次の表に示します。

表 7-5　CJMSP ブローカーの性能情報に出力される項目

出力項目	説明
Connections	接続状態のコネクション数です。
JVM Heap	使用している JavaVM のヒープサイズと使用可能なメモリ量です。
Threads	使用されているスレッド数です。 なお，出力例では 22 となっています。特定の時間だけ CJMSP ブローカーに接続している場合，使用されているスレッド数は自動で更新されます。 （　）内はスレッドプール値の最小値と最大値です。 最小値（14） 最小のスレッドプール値を示します（jms サービスの最小値は 10，admin サービスの最小値は 4 です。このため，最小値は 14 になります）。 最大値（1010） 最大のスレッドプール値を示します（jms サービスの最大値は 1000，admin サービスの最大値は 10 です。このため，最大値は 1010 となります）。
In	CJMSP ブローカーの受信メッセージ数です。
Out	CJMSP ブローカーから送信されたメッセージ数です。
Rate In	CJMSP ブローカーの受信メッセージの割合です。また，1 秒間のメッセージ数と 1 秒当たりのメッセージパケットが表示されます。
Rate Out	CJMSP ブローカーから送信されたメッセージの割合です。また，1 秒間のメッセージ数と 1 秒当たりのメッセージパケットが表示されます。

7.8.4　管理情報およびメッセージの永続化サービス

CJMSP ブローカーに障害が発生した場合，回復には，障害発生前のメッセージ送受信操作の状態を示す情報が必要です。CJMSP ブローカーでは，回復に必要な状態についての情報をファイルに保存して管理します。障害が発生した場合，保存された情報を基に CJMSP ブローカーの状態を回復して，操作を再開できるため，メッセージ配信の信頼性を確保できます。

CJMSP ブローカーは，次のような情報をファイルに保存して管理します。

- 送信先の情報
- 永続化サブスクライバーの情報
- メッセージの情報
- トランザクションの情報
- メッセージ配信の確認応答についての情報

障害発生後に CJMSP ブローカーを再起動すると，保存していた各情報を基に，送信先と永続化サブスクライバーの再作成，永続化メッセージの回復，トランザクションのロールバックが実施され，未配信のメッセージのためのルートが再作成されます。その後，メッセージ配信が再開されます。

(1)　管理情報およびメッセージの永続化の仕組み

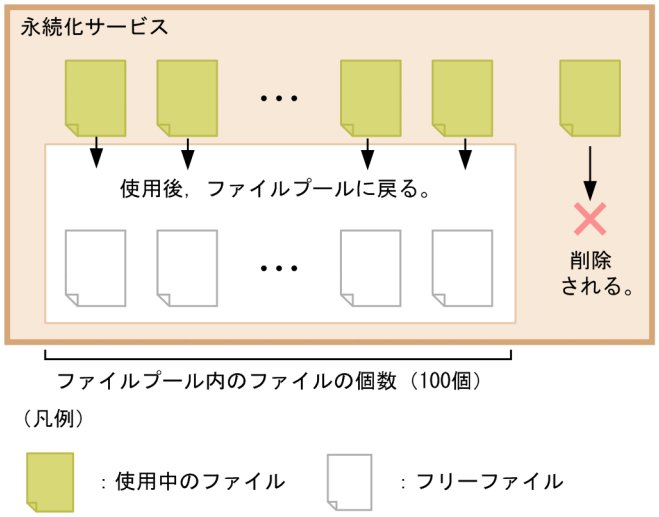
CJMSP ブローカーでは，永続化する情報の種類ごとにファイルを作成して管理します。保存用のファイルは，CJMSP ブローカーが作成されたときに自動的に作成されます。情報の種類ごとに保存先のファイルは異なります。

しきい値（1 メガバイト）以下のサイズのメッセージは，メッセージごとに一つの可変長のファイルに保存されます。また，しきい値を超えるサイズのメッセージは，ファイルプールの複数のファイルで管理されます。ファイルプールの各ファイルのサイズは 1 メガバイトです。

(2)　ファイルプールの利用

CJMSP ブローカーが管理するファイルプールの概要を次の図に示します。

図 7-11　CJMSP ブローカーが管理するファイルプールの概要



ファイルプールの特徴を次に示します。

- 不要になったファイルは，**フリーファイル**としてファイルプールに返却されます。ファイルプールは，送信先ごとに管理されています。
- ファイルプールで管理するファイル数の最大値は 100 です。ファイル数が最大値を超えた場合，不要になったファイルはプールに返却されないで削除されます。
- ファイルプール内のファイルは，CJMSP ブローカーを停止して再起動したあとも残ります。

(3)　ファイルへの書き込みのタイミング

ファイルへのデータの書き込みは，OS の機能で実行します。このとき，書き込みを同期・非同期のどちらで実行するかを imq.persist.file.sync.enabled プロパティで選択できます。なお，デフォルトでは書き込みは非同期で実施されます。

同期書き込み・非同期書き込みの利点と欠点を次の表に示します。

表 7-6　ファイルへの同期書き込み・非同期書き込みの利点と欠点

書き込みのタイミング	利点	欠点
非同期	書き込み処理による処理性能への影響がありません。	障害が発生したときに，一部のデータが失われることがあります。
同期	障害発生時点までのデータが保存されているため，障害発生直前の状態から CJMSP ブローカーの処理を再開できます。	書き込み処理による処理性能への影響があります。

(4)　ファイルの格納先

CJMSP ブローカーが管理する永続化データを保存したファイルは，デフォルトでは次のディレクトリに保存されています。なお，var ディレクトリの格納先は，cjmsbroker コマンドの-varhome オプションで変更できます。

Windows の場合

```
<Application Serverのインストールディレクトリ>¥CC¥cjmsp¥var¥instances¥<CJMSPブローカーの名称>¥fs370
```

UNIX の場合

```
/opt/Cosminexus/CC/cjmsp/var/instances/<CJMSPブローカーの名称>/fs370
```

fs370 ディレクトリ下の構成を次に示します。

表 7-7　fs370 ディレクトリ下の構成

ディレクトリ・ファイル名	説明
message ディレクトリ	このディレクトリ下に，送信先名ごとのサブディレクトリが作成されます。それぞれのサブディレクトリ内に，メッセージごとの個別のファイルが格納されています。
interest ファイル	コンシューマーの情報が格納されています。
destination ファイル	送信先の情報が格納されています。
property ファイル	内部情報が格納されています。
txn ファイル	トランザクション ID が格納されています。
txnack ファイル	すべてのトランザクション ID に対する配信確認が格納されています。
configrecord ファイル	内部情報が格納されています。

このディレクトリの内容は，不正にアクセスされないよう，OS のアクセス権などを設定して適切に管理してください。

! 注意事項

- fs370 ディレクトリ下のファイルを移動したり，ファイルの属性を読み取り専用に変更したりしないでください。CJMSP ブローカーが正しく動作しなくなるおそれがあります。
- ネットワークドライブや，ネットワーク上のファイルを使用することはできません。cjmsbroker コマンドの-varhome オプションで var ディレクトリの格納先を変更する場合は，CJMSP ブローカーと同じマシン上のディレクトリを選択するようにしてください。
- 一つのトランザクションの中で大量のメッセージを処理すると，txnack ファイルのファイルサイズが大幅に増加することがあります。大量のメッセージを送信する場合，性能テストを実施して，必要なファイルサイズを見積もってください。

---

(5) ファイルのバックアップ

永続化データのバックアップを取得する場合は，CJMSP ブローカー単位で取得します。var ディレクトリ下の instances ディレクトリをほかの場所にコピーして保存してください。

回復する場合は，まず，CJMSP ブローカーを停止します。バックアップしておいた instances ディレクトリ以下のファイルを var ディレクトリ下に上書きしてから，CJMSP ブローカーを再起動してください。

7.9　CJMSP リソースアダプタの機能

CJMSP リソースアダプタは，J2EE サーバと CJMSP ブローカーを接続するためのリソースアダプタです。J2EE サーバの Web コンテナまたは EJB コンテナ上で動作する J2EE アプリケーションと CJMSP ブローカー間のメッセージの送受信を実現します。

CJMSP リソースアダプタの動作は Connector 属性ファイルに設定します。なお，属性の設定個所に応じて，指定した内容の影響範囲が異なります。指定個所と影響範囲について次の表に示します。

表 7-8　Connector 属性ファイルの指定個所と指定内容

Connector 属性ファイルのタグ※	指定内容
<resourceadapter>	CJMSP リソースアダプタ全体の動作に影響する属性を指定します。
<connection-definition>	コネクションインタフェースごとの設定を指定します。

注※　JavaBean の実装と対応しています。

7.10　Message-driven Bean の呼び出し

Message-driven Bean は，コンシューマーとして動作し，メッセージの非同期処理を実行するための Enterprise Bean です。送信先にメッセージが到着したタイミングでアクティブになり，メッセージを受け取って処理します。

この節では，コンシューマーである Message-driven Bean の動作について説明します。

7.10.1　Message-driven Bean によるメッセージ処理の特徴

Message-driven Bean は，クライアントアプリケーションであるプロデューサーから直接呼び出すことはできません。プロデューサーは，送信先にメッセージを送信することで，Message-driven Bean を間接的に呼び出します。

Message-driven Bean は基本的に状態を保持せず，また特定のクライアントアプリケーションと関連づけられて処理を実行されることがありません。このため，送信されたメッセージは複数の Message-driven Bean によって同時に並行処理できます。

! 注意事項

- Message-driven Bean の複数のインスタンスの同時実行によって，複数のメッセージは並行処理されます。Message-driven Bean クラスのインスタンスによって実行される処理の順序性は保証されません。
- 大量のメッセージを Message-driven Bean に配信しようとしたときに，RejectedExecutionException 例外が発生することがあります。この例外が発生した場合，一部のメッセージは Message-driven Bean に配信されません。この状況を避けるためには，CJMSP リソースアダプタの実行時属性として，スレッドプールの最大数を Message-driven Bean のインスタンス数以上に設定してください。
- Message-driven Bean に設定した同時実行インスタンス数の制限によってメッセージの送信が遅れることがあります。この問題を防ぐためには，Message-driven Bean のインスタンス数以上の値を CJMSP リソースアダプタで実行するスレッド数として指定してください。

  また，Message-driven Bean のエンドポイント数は，メッセージを並列処理できる数になります。エンドポイント数の制限によって問題が発生するのを防ぐために，エンドポイント数には Message-driven Bean のインスタンス数以上の値を指定してください。式を示します。

```
Message-driven Beanのインスタンス数の総数<=CJMSPリソースアダプタのWorkスレッド数
Message-driven Beanのインスタンス数>=エンドポイントのインスタンス数
```

  なお，Message-driven Bean のインスタンス数は，cosminexus.xml または MessageDrivenBean 属性ファイルで定義します。Message-driven Bean のエンドポイント数は，DD で定義します。CJMSP リソースアダプタのスレッド数の定義は Connector 属性ファイルで定義します。

  cosminexus.xml の詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「2. アプリケーション属性ファイル（cosminexus.xml）」を参照してください。MessageDrivenBean 属性ファイルの詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「3.6　MessageDrivenBean 属性ファイル」を参照してください。Connector 属性ファイルの詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1　Connector 属性ファイル」を参照してください。

- CJMS プロバイダを使用して，Message-driven Bean にメッセージを配信する場合，メッセージ配信中に CJMSP ブローカーを停止すると，CJMSP ブローカーを再起動してもメッセージは配信されません。CJMSP ブローカーを停止した場合は，J2EE サーバを再起動してください。

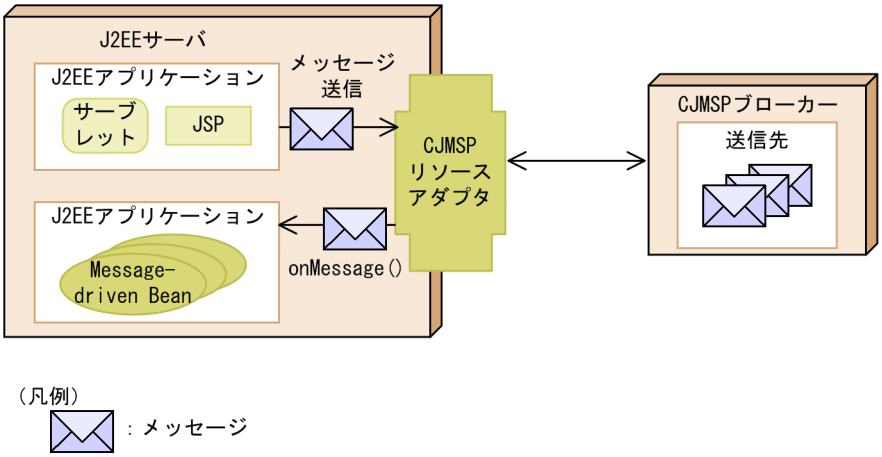
7.10.2　Message-driven Bean を呼び出す流れ

メッセージの送信元となるアプリケーションにとって，Message-driven Bean はメッセージのコンシューマーです。アプリケーションから Message-driven Bean にとってのメッセージリスナである送信先にメッセージを送信すると，Message-driven Bean がメッセージを受け取り，コンシューマーとして処理を実行します。

メッセージが到着すると，Message-driven Bean のメッセージリスナメソッドである javax.jms.MessageListener#onMessage メソッドが J2EE コンテナによって呼び出されます。メッセージリスナメソッドに含まれるビジネスロジックによって，処理が実行されます。

Message-driven Bean を動作させる流れを次の図に示します。

図 7-12　Message-driven Bean を動作させる流れ



なお，送信元の J2EE アプリケーションの JNDI 名前空間には，Message-driven Bean を動作させるための送信先が設定されている必要があります。

Message-driven Bean と送信先であるキューまたはトピックは，Message-driven Bean を含む J2EE アプリケーションを J2EE サーバにデプロイしたときに関連づけられます。

7.10.3　Message-driven Bean で必要な設定

Message-driven Bean の動作は，Message-driven Bean の属性として設定します。送信先の情報，メッセージの確認応答モード，およびメッセージセレクターなどが設定できます。属性は，DD または MessageDrivenBean 属性ファイルの<activation-config>タグで設定します。詳細は，「7.12　DD での定義」およびマニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「3.6　MessageDrivenBean 属性ファイル」を参照してください。

ここでは，主な設定項目について説明します。

(1)　送信先の設定

Message-driven Bean と関連づける送信先の種類として，キュー（javax.jms.Queue）またはトピック（javax.jms.Topic）のどちらかを設定します。トピックと設定した場合は，さらに永続化サブスクライバーを使用するかどうかも設定します。

送信先をキューとした場合，または永続化サブスクライバーを使用する設定にした場合は，Message-driven Bean が動作する J2EE アプリケーションが動作していないときなどには送信先がメッセージを保持します。

送信先をトピックとして永続化サブスクライバーを使用しない設定にした場合は，Message-driven Bean が動作する J2EE アプリケーションが動作していないときに送信されたメッセージは Message-driven Bean に配信されません。

送信先の種類や永続化サブスクライバーについては，「7.4　メッセージングモデルの種類」を参照してください。

(2)　メッセージの確認応答モードの設定

メッセージの確認応答モードとして，AUTO_ACKNOWLEDGE または DUPS_OK_ACKNOWLEDGE のどちらかを設定します。それぞれのモードの詳細については，JMS 仕様を参照してください。

メッセージの確認応答は，コンテナが実行します。トランザクションを CMT で管理している場合，確認応答はトランザクションをコミットする処理の一部として実行されます。BMT で管理している場合は，トランザクションとは別に，コンテナによって承認処理が実行されます。なお，確認応答を API で実行する実装はしないでください。

(3)　メッセージセレクターの設定

メッセージセレクターとして，Message-driven Bean が受信するメッセージの選択基準を設定します。これによって，メッセージプロパティに特定の値が設定されたメッセージだけを受信して，Message-driven Bean を動作させることができます。

メッセージセレクターについては，「7.6　メッセージセレクターによる受信メッセージの選択」を参照してください。

7.10.4　トランザクションコンテキストの設定

Message-driven Bean のメッセージリスナ，およびタイムアウトコールバックメソッドを呼び出すトランザクションのスコープを示す，トランザクションコンテキストを設定します。

トランザクション管理に CMT を使用する場合は，NOT_SUPPORTED 属性を指定してください。

なお，BMT を使用する Message-driven Bean が javax.transaction.UserTransaction インタフェースを使用してトランザクションを管理する場合，メッセージの受信はトランザクションの処理に含まれません。

7.11　アプリケーション実装時の制限事項

この節では，CJMS プロバイダを使用してアプリケーションを実装する際の制限事項について説明します。

CJMS プロバイダは JMS1.1 仕様に準拠していますが，一部のインタフェースおよびメソッドは使用できません。アプリケーションで使用できないインタフェースまたはメソッドを使用すると，JMSException 例外がスローされるおそれがあります。

使用できないインタフェースを次に示します。

- javax.jms.ServerSession
- javax.jms.ServerSessionPool
- javax.jms.ConnectionConsumer
- すべての javax.jms XA インタフェース

また，次の表に示すインタフェースの一部のメソッドは，アプリケーションクライアントコンテナで実行するアプリケーションで使用するメソッドです。CJMS プロバイダでは使用できません。

CJMS プロバイダで使用できないメソッドをインタフェースごとに次の表に示します。

表 7-9　CJMS プロバイダで使用できないメソッド

インタフェース	使用できないメソッド
javax.jms.Session	• setMessageListener(MessageListener listener) • getMessageListener() • run()
javax.jms.Connection	• setExceptionListener(ExceptionListener listener) • stop() • setClientID(String clientID) • トランザクション処理される Session オブジェクトを作成するメソッド （例えば，createSession メソッドの一つ目の引数「transacted」に「true」を指定することができません）。
javax.jms.QueueConnection	• createConnectionConsumer(Queue queue, String messageSelector, ServerSessionPool sessionPool, int maxMessages) • トランザクション処理される QueueSession オブジェクトを作成するメソッド （例えば，createQueueSession メソッドの一つ目の引数「transacted」に「true」を指定することができません）。
javax.jms.TopicConnection	• createConnectionConsumer(Topic topic, String messageSelector, ServerSessionPool sessionPool, int maxMessages) • createDurableConnectionConsumer(Topic topic, String subscriptionName, String messageSelector, ServerSessionPool sessionPool, int maxMessages) • トランザクション処理される TopicSession オブジェクトを作成するメソッド

インタフェース	使用できないメソッド
javax.jms.TopicConnection	（例えば，createTopicSession メソッドの一つ目の引数「transacted」に「true」を指定することができません）。

また，Web コンテナまたは EJB コンテナで動作するアプリケーションのコンポーネントでは，一つの接続で複数のアクティブなセッションオブジェクトを作成してはいけません。すでにアクティブなセッションオブジェクトがある場合に，createSession メソッドでオブジェクトを作成しようとした場合，JMSException 例外がスローされることがあります。

7.12　DD での定義

この節では，Message-driven Bean の DD で定義する項目について説明します。

Message-driven Bean の DD では，<activation-config>タグ下の属性で次の表に示す項目を設定します。

表 7-10　<activation-config>タグ下の属性で指定する項目

属性	データ型	指定可能値	省略値	説明
destination	String	−	−	Message-driven Bean にメッセージを送信する，送信先の名称を指定します。この属性には，CJMSP リソースアダプタの管理対象オブジェクトの Name に設定した名称を設定してください。
destinationType	String	• javax.jms.Queue • javax.jms.Topic	−	送信先の種類として，キュー（javax.jms.Queue）またはトピック（javax.jms.Topic）を指定します。
messageSelector	String	−	−	受信するメッセージを選択するためのメッセージセレクターを指定します。
subscriptionName	String	−	−	永続化サブスクライバーの名称を指定します。 subscriptionDurability 属性に「Durable」を指定した場合は，必ず指定してください。
subscriptionDurability	String	• Durable • NonDurable	NonDurable	永続化サブスクライバーにするかどうかを指定します。 永続化サブスクライバーにする場合は「Durable」，しない場合は「NonDurable」を指定します。 この属性は，destinationType 属性に「javax.jms.Topic」を指定した場合にだけ有効になります。 また，「Durable」と指定した場合，次の属性も必ず指定してください。 • subscriptionName • clientId
clientId	String	−	−	CJMSP ブローカーに接続するための永続化サブスクライバーの識別子を指定します。 subscriptionDurability 属性に「Durable」を指定した場合は，必ず指定してください。
acknowledgeMode	String	• Auto-acknowledge • Dups-ok-acknowledge	Auto-acknowledge	確認応答モードを指定します。指定するモードの詳細は JMS 仕様を参照してください。

属性	データ型	指定可能値	省略値	説明
endpointExceptionRedeliveryAttempts	Integer	1～2147483647	6	メッセージ配信中に例外が発生した場合に再配信する回数を指定します。
sendUndeliverableMsgsToDMQ	Boolean	• true • false	true	Message-driven Bean が実行時例外をスローした場合で，再配信した回数が endpointExceptionRedeliveryAttempts 属性に指定した回数を超えたときに，配信されなかったメッセージをデッドメッセージキューに登録するかどうかを指定します。 登録する場合は「true」，しない場合は「false」を指定します。 「false」を指定した場合，該当するメッセージは CJMSP ブローカーによって，有効なコンシューマーに再配信されます（再度同じ Message-driven Bean に配信される場合もあります）。
endpointPoolMaxSize	Integer	1～2147483647	1	エンドポイントのプールの最大値を指定します。
endpointPoolSteadySize	Integer	0～2147483647	1	エンドポイントのプールの最小値を指定します。
endpointExceptionRedeliveryInterval	Integer	1～2147483647	500	例外が発生した場合にエンドポイントにメッセージを再配信する間隔をミリ秒で指定します。

（凡例）

－：指定可能値の制限または省略値はなし

7.13　実行環境の構築の流れ

この節では，CJMS プロバイダを使用する場合の実行環境の構築の流れについて説明します。

メッセージの送受信を実行する J2EE アプリケーションの実行環境については，あらかじめ運用管理ポータル（Management Server）を使用して構築しておいてください。J2EE アプリケーションの実行環境の構築手順の詳細は，マニュアル「アプリケーションサーバ 運用管理ポータル操作ガイド」の「3.　J2EE アプリケーションを実行するシステムの構築と削除」を参照してください。

なお，CJMSP ブローカーは，Management Server の管理対象になりません。このため，CJMSP ブローカーの構築・運用は，コマンドおよび定義ファイルで実行します。

ポイント

CJMS プロバイダを使用する場合のユースケースについては，「付録 E　CJMS プロバイダのユースケース」を参照してください。

実行環境の構築の流れを次の図に示します。

図 7-13　実行環境の構築の流れ

操作の流れを説明します。説明の番号は図中の番号と対応しています。

1. **前提環境の準備として，Application Server のインストールと初期設定を実行します。**

   CJMSP ブローカーと J2EE サーバを異なるマシンに構築する場合は，それぞれのマシンに Application Server をインストールしてください。

   Application Server のインストールと初期設定については，マニュアル「アプリケーションサーバ システム構築・運用ガイド」の「2.2.1　Application Server のインストールについて」を参照してください。

2. **CJMSP ブローカーの共通プロパティと管理コマンドのプロパティを設定します。**

同じマシン内に作成するすべての CJMSP ブローカーに共通するプロパティを設定します。また，管理コマンドのログ出力に関するプロパティを設定します。設定方法については，「7.14.1　CJMSP ブローカーの共通プロパティと管理コマンドのプロパティの設定」を参照してください。

3. CJMSP ブローカーを作成します。

作成方法については，「7.14.2　CJMSP ブローカーの作成」を参照してください。

4. CJMSP ブローカーの個別プロパティを設定します。

それぞれの CJMSP ブローカーに対して，個別のプロパティを設定します。設定方法については，「7.14.3　CJMSP ブローカーの個別プロパティの設定」を参照してください。

5. CJMSP ブローカーを起動します。

起動方法については，「7.14.4　CJMSP ブローカーの起動」を参照してください。

6. 各 CJMSP ブローカーに送信先を作成します。

作成方法については，「7.14.5　送信先の作成」を参照してください。

7. CJMSP リソースアダプタを設定して，開始します。

CJMSP リソースアダプタへの属性の設定には，Connector 属性ファイルを使用します。

属性の設定後，CJMSP リソースアダプタを開始します。

設定および開始方法については，「7.15　CJMSP リソースアダプタの設定」を参照してください。

8. J2EE アプリケーションを設定して，開始します。

プロデューサーおよびコンシューマーとなる J2EE アプリケーションをそれぞれインポートして属性を設定します。なお，コンシューマーとなる J2EE アプリケーションの Message-driven Bean に属性を設定する場合は，DD，cosminexus.xml または MessageDrivenBean 属性ファイルを使用します。

設定および開始方法については，「7.16　J2EE アプリケーションの設定」を参照してください。

ポイント

CJMS プロバイダに関するファイルは，インストール時に次のディレクトリに格納されます。

- Windows の場合
  <Application Serverのインストールディレクトリ>¥CC¥cjmsp
- UNIX の場合
  /opt/Cosminexus/CC/cjmsp

! 注意事項

CJMSP リソースアダプタを開始する前に，CJMSP ブローカーを起動しておく必要があります。CJMSP リソースアダプタの開始中に CJMSP ブローカーが停止した場合，CJMSP リソースアダプタは，120 秒ごとに CJMSP ブローカーへの接続をリトライします。

7.14　CJMSP ブローカーの設定

この節では，CJMSP ブローカーの設定について説明します。

なお，CJMSP ブローカーへのプロパティの設定方法には，2 種類の方法があります。複数の設定方法で異なる値を指定した場合，次の表に示す番号の順で設定が優先されます。すべての CJMSP ブローカーに共通の設定をする場合は，commonconfig.properties に設定することをお勧めします。

表 7-11　CJMSP ブローカーへのプロパティの設定方法

優先順位	設定方法	設定個所	プロパティの有効範囲
1	CJMSP ブローカー個別プロパティファイルを使用する設定方法	config.properties※1	個別の CJMSP ブローカーだけに有効です。
2	CJMSP ブローカー共通プロパティファイルを使用する設定	commonconfig.properties※2	同じマシン内のすべての CJMSP ブローカーに有効です。

注※1　デフォルトでは次のディレクトリに格納されます。なお，var ディレクトリは CJMSP ブローカーを最初に起動したあとで使用できるディレクトリです。CJMSP ブローカー個別プロパティファイル（config.properties）については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「7.4　config.properties（CJMSP ブローカー個別プロパティファイル)」を参照してください。

Windows の場合

<Application Serverのインストールディレクトリ>¥CC¥cjmsp¥var¥instances¥<CJMSPブローカーの名称>¥props

UNIX の場合

/opt/Cosminexus/CC/cjmsp/var/instances/<CJMSPブローカーの名称>/props

注※2　次のディレクトリに格納されています。なお，CJMSP ブローカー共通プロパティファイル（commonconfig.properties）については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「7.3　commonconfig.properties（CJMSP ブローカー共通プロパティファイル)」を参照してください。

Windows の場合

<Application Serverのインストールディレクトリ>¥CC¥cjmsp¥lib¥props¥broker

UNIX の場合

/opt/Cosminexus/CC/cjmsp/lib/props/broker

また，管理コマンド（cjmsicmd）のログの出力方法についてのプロパティは，admin.properties に指定します。admin.properties は，次のディレクトリに格納されています。

admin.properties（管理コマンドプロパティファイル）については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「7.2　admin.properties（管理コマンドプロパティファイル)」を参照してください。

Windows の場合

<Application Serverのインストールディレクトリ>¥CC¥cjmsp¥var¥admin¥config

UNIX の場合

/opt/Cosminexus/CC/cjmsp/var/admin/config

! 注意事項

CJMSP ブローカーの設定をする場合，cjmsp ディレクトリ下のファイルのうち，次のファイル以外は編集しないでください。

- config.properties
- commonconfig.properties

- admin.properties

また，これらのファイルに対しても，OS の機能によるアクセス権の変更はしないでください。

なお，ファイアウォールを使用するシステムの場合，CJMS プロバイダによるポートの利用を許可する設定が必要です。

例えば，Windows の場合，次のコマンドの実行によって利用を許可できます。

```
netsh firewall add allowedprogram program= "<COSMINEXUS_HOME>/jdk/bin/java.exe" name="Java Virtual Machine" mode=ENABLE.
```

CJMS プロバイダが利用するポート番号は，次のとおりです。( ) 内はそのポート番号を設定するプロパティです。config.properties で設定できます。

- CJMSP ブローカーのポートマッパーが使用するポート番号（imq.portmapper.port）
- jms サービスが使用するポート番号（imq.jms.tcp.port）
- admin サービスが使用するポート番号（imq.admin.tcp.port）

7.14.1 CJMSP ブローカーの共通プロパティと管理コマンドのプロパティの設定

CJMSP ブローカーのプロパティには，デフォルト値が設定されています。必要に応じて，commonconfig.properties を編集してすべての CJMSP ブローカーに設定するプロパティの値を変更してください。

また，admin.properties を編集して，管理コマンドのログの出力に関するプロパティを設定してください。

7.14.2 CJMSP ブローカーの作成

CJMSP ブローカーは，cjmsbroker コマンドを実行して作成します。

手順を次に示します。

1. cd コマンドでカレントディレクトリを次のディレクトリに移動します。

   Windows の場合

   ```
   <Application Serverのインストールディレクトリ>¥CC¥cjmsp¥bin
   ```

   UNIX の場合

   ```
   /opt/Cosminexus/CC/cjmsp/bin
   ```

2. cjmsbroker コマンドを実行します。

   CJMSP ブローカーの名前を-name オプションに指定して実行します。

   実行例（名前「MyBroker」を指定して CJMSP ブローカーを作成する場合）

   ```
   cjmsbroker -name MyBroker
   ```

3. cjmsicmd shutdown bkr コマンドを実行します。

   作成した CJMSP ブローカーを停止します。以降で個別の CJMSP ブローカーのプロパティを設定するためです。

   実行例

   ```
   cjmsicmd shutdown bkr
   ```

手順 2.で cjmsbroker を実行したときに，var ディレクトリ下の instances 下に指定した名前（実行例の場合は「MyBroker」）のディレクトリが作成されます。名前を指定しなかった場合は，デフォルトの CJMSP ブローカーが起動します。

なお，var ディレクトリの格納場所は cjmsbroker コマンドの-varhome オプションで指定します。

! 注意事項

-varhome オプションでディレクトリを指定した場合は，2 回目以降の CJMSP ブローカー起動時にも，-varhome オプションで同じ値を指定してください。

2 回目以降の起動時に-varhome オプションを指定しなかったり，別の場所を指定したりすると，-name オプションに同じ名前を指定しても，指定した var ディレクトリまたはデフォルトディレクトリ下に新しいインスタンスが作成されます。

7.14.3　CJMSP ブローカーの個別プロパティの設定

ここでは，CJMSP ブローカーの個別プロパティの設定について説明します。

CJMSP ブローカーの個別のプロパティは，config.properties の指定で設定できます。ただし，config.properties を指定できるのは，一度 cjmsbroker コマンドで起動され，var ディレクトリ以下が作成されている CJMSP ブローカーだけです。

config.properties を使用したプロパティの指定例を次に示します。

config.properties を使用したプロパティの指定例

```
imq.portmapper.port=7777
broker.logger.MessageLogFile.filenum=4
```

指定内容の意味

- CJMSP ブローカーのポートマッパーが使用するポート番号は 7777 です。
- メッセージログファイルの面数は 4 です。

! 注意事項

複数の CJMSP ブローカーを起動するシステムの場合は，必ず imq.portmapper.port プロパティの値を変更してください。デフォルトポート（7676）を使用している CJMSP ブローカーがあると，新しい CJMSP ブローカーは作成できません。

7.14.4　CJMSP ブローカーの起動

ここでは，CJMSP ブローカーの起動について説明します。

手順を示します。カレントディレクトリは次のディレクトリに移動していることを前提とします。

Windows の場合

```
<Application Serverのインストールディレクトリ>¥CC¥cjmsp¥bin
```

UNIX の場合

```
/opt/Cosminexus/CC/cjmsp/bin
```

1. cjmsbroker コマンドを実行します。

   ここでは，「MyBroker」を起動する実行例を示します。

   実行例（「MyBroker」を起動する場合）

   ```
   cjmsbroker -name MyBroker
   ```

7.14.5　送信先の作成

ここでは，CJMSP ブローカーで管理する送信先の作成について説明します。

手順を示します。カレントディレクトリは次のディレクトリに移動していることを前提とします。

Windows の場合

```
<Application Serverのインストールディレクトリ>¥CC¥cjmsp¥bin
```

UNIX の場合

```
/opt/Cosminexus/CC/cjmsp/bin
```

1. cjmsicmd create dst コマンドを実行します。

   ここでは，キューを作成する実行例とトピックを作成する実行例を示します。

   実行例（キュー「Queue1」を作成する場合）

   ```
   cjmsicmd create dst -t q -n Queue1
   ```

   実行例（トピック「Topic1」を作成する場合）

   ```
   cjmsicmd create dst -t t -n Topic1
   ```

なお，送信先は，コマンドで作成するほか，アプリケーション内で API によって作成する方法があります。詳細は，「7.8.2　送信先の管理とルーティングサービス」を参照してください。

7.15　CJMSP リソースアダプタの設定

CJMSP リソースアダプタは，次のディレクトリ下に，cjmsra.rar として格納されています。デフォルトの名称は，「Cosminexus_JMS_Provider_RA」です。

Windows の場合

```
<Application Serverのインストールディレクトリ>¥CC¥cjmsp¥lib
```

UNIX の場合

```
/opt/Cosminexus/CC/cjmsp/lib
```

リソースアダプタのインポートから開始までの手順については，マニュアル「アプリケーションサーバ 運用管理ポータル操作ガイド」の「3.3　リソースの設定」を参照してください。

CJMSP リソースアダプタには，属性として，次の項目が設定できます。

- ConnectionFactory や送信先など，管理対象オブジェクトの属性
- トランザクションサポートレベル
- ログ出力に関する属性

これらの属性を変更する場合は，CJMSP リソースアダプタに設定する Connector 属性ファイルを編集してください。

CJMSP リソースアダプタの属性には，リソースアダプタ単位に指定する項目と，ManagedConnectionFactory クラス単位に指定する項目があります。CJMSP リソースアダプタで指定する主な属性を次の表に示します。

表 7-12　CJMSP リソースアダプタで指定する主な属性

対象	Connector 属性ファイルのタグ	指定する項目
リソースアダプタ単位の属性	<resourceadapter>下の<config-property>	CJMSP リソースアダプタ全体に影響する属性 • connectionURL • reconnectEnabled • reconnectAttempts • reconnectInterval ログに関する属性 • MsgLogLevel • MsgLogFileNum • MsgLogFileSize • ExpLogFileNum • ExpLogFileSize
ManagedConnectionFactory クラス単位の属性	<connection-definition>下の<config-property>	コネクションインタフェースごとの属性 • clientId • reconnectEnabled • reconnectAttempts • reconnectInterval

対象	Connector 属性ファイルのタグ	指定する項目
管理対象オブジェクトの属性	<adminobject>下の<adminobject-name>,<adminobject-interface>,<adminobject-class>	管理対象オブジェクトの名称，送信先の種類，およびクラスの情報
	<adminobject>下の<config-property>	送信先の名前（Message-driven Bean の destination 属性に指定する値） • Name

なお，CJMS プロバイダでは，Connector 属性ファイルのテンプレートファイルを提供しています。必要に応じて使用してください。テンプレートファイルの格納先については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1.14　Connector 属性ファイルのテンプレートファイル」を参照してください。

7.16　J2EE アプリケーションの設定

CJMS プロバイダを使用する J2EE アプリケーションをインポートします。

J2EE アプリケーションのインポートから開始までの手順については，マニュアル「アプリケーションサーバ 運用管理ポータル操作ガイド」の「3.4　J2EE アプリケーションの設定」を参照してください。

J2EE アプリケーションに設定する属性は，DD または cosminexus.xml に指定できます。CJMS プロバイダを使用する場合に DD に必要な設定については，「7.12　DD での定義」を参照してください。なお，cosminexus.xml を含まない J2EE アプリケーションに対して，J2EE サーバにインポートしたあとで属性を設定する場合は，属性ファイルを使用して，サーバ管理コマンドで設定ください。

7.17　CJMS プロバイダを使用する場合のシステムの開始と停止

この節では，CJMS プロバイダを使用する場合のシステムの開始手順と停止手順について説明します。

7.17.1　システムの開始手順

CJMS プロバイダを使用するシステムは，次の手順で開始します。なお，手順 1.と手順 2.は順不同です。

! 注意事項

CJMSP リソースアダプタを開始する前に，必ず CJMSP ブローカーを起動してください。CJMSP ブローカー起動前に CJMSP リソースアダプタを開始することはできません。また，CJMSP リソースアダプタが開始した状態で J2EE サーバを停止し，CJMSP ブローカーを停止してから J2EE サーバを再起動すると，再起動後の CJMSP リソースアダプタの再開始には失敗します。

---

1. J2EE サーバを含む J2EE アプリケーションの実行環境の各プロセスを起動します。

Smart Composer で構築したシステムの場合は，一括起動ができます。詳細は，マニュアル「アプリケーションサーバ 運用管理ポータル操作ガイド」の「4.1.1　システムの起動手順」を参照してください。

! 注意事項

この時点で CJMSP リソースアダプタおよび J2EE アプリケーションは開始しません。

---

2. CJMSP ブローカーを起動します。

ここでは，CJMSP ブローカー「MyBroker」を起動する場合の実行例を示します。

実行例

Windows の場合

```
<Application Serverのインストールディレクトリ>¥CC¥cjmsp¥bin¥cjmsbroker -name MyBroker
```

UNIX の場合

```
/opt/Cosminexus/CC/cjmsp/bin/cjmsbroker -name MyBroker
```

なお，CJMSP ブローカーを起動する際，必要に応じて CJMSP ブローカーが使用する Java ヒープサイズを変更してください※。

3. CJMSP リソースアダプタを開始します。

サーバ管理コマンドを使用した実行例を次に示します。この例では，J2EE サーバの名称は「cmx_MyWebSystem_unit1_J2EE_01」，CJMSP リソースアダプタの名称は「CJMS_Provider_RA」とします。

実行例

Windows の場合

```
<Application Serverのインストールディレクトリ>¥CC¥admin¥bin¥cjstartrar
"cmx_MyWebSystem_unit1_J2EE_01" -resname "CJMS_Provider_RA"
```

UNIX の場合

```
/opt/Cosminexus/CC/admin/bin/cjstartrar "cmx_MyWebSystem_unit1_J2EE_01" -resname
"CJMS_Provider_RA"
```

4. メッセージを送受信する J2EE アプリケーションを開始します。

サーバ管理コマンドを使用した実行例を次に示します。なお，J2EE サーバの名称は「cmx_MyWebSystem_unit1_J2EE_01」，J2EE アプリケーションの名称は「JMSSampleApp」とします。

実行例

Windows の場合

```
<Application Serverのインストールディレクトリ>¥CC¥admin¥bin¥cjstartapp
”cmx_MyWebSystem_unit1_J2EE_01” -name ”JMSSampleApp”
```

UNIX の場合

```
/opt/Cosminexus/CC/admin/bin/cjstartapp ”cmx_MyWebSystem_unit1_J2EE_01” -name
”JMSSampleApp”
```

注※　CJMSP ブローカーが使用する Java ヒープサイズの考え方

CJMSP ブローカーは，デフォルトの設定では，192 メガバイトの Java ヒープを使用する設定になっています。この値は，大量のメッセージを扱うには小さ過ぎるため，必要に応じて増加させてください。

使用する Java ヒープのメモリサイズが小さ過ぎると，空きメモリを確保するために J2EE アプリケーションとのコネクションをクローズする必要が発生し，処理できるメッセージの量が少なくなります。

ただし，CJMSP ブローカーで使用できる Java ヒープサイズの最大値を大きくし過ぎると，システムの物理メモリに影響が出て，システムに OutOfMemoryError を発生させる要因になります。OutOfMemoryError が発生すると，システムの処理性能を低下させるほか，予想外の状態で CJMSP ブローカーに障害が発生したり，ほかのアプリケーションやサービスの実行に影響を及ぼしたりします。

これらを考慮し，実行する環境に応じて，適切な値を設定してください。

CJMSP ブローカーが使用する Java ヒープのサイズは，CJMSP ブローカーを起動するときに cjmsbroker コマンドの-vmargs オプションで指定できます。

実行例を示します。

実行例

```
cjmsbroker -vmargs ”-Xms256m -Xmx1024m”
```

この例では，CJMSP ブローカーは，CJMSP ブローカーの起動時には 256 メガバイトのメモリを使用し，最大で 1 ギガバイトの Java ヒープを使用するように設定しています。

7.17.2　システムの停止手順

CJMS プロバイダを使用するシステムは，次の手順で停止します。なお，手順 3.と手順 4.は順不同です。

1. メッセージを送受信する J2EE アプリケーションを停止します。

サーバ管理コマンドを使用した実行例を次に示します。この例では，J2EE サーバの名称は「cmx_MyWebSystem_unit1_J2EE_01」，J2EE アプリケーションの名称は「JMSSampleApp」とします。

実行例

Windows の場合

```
<Application Serverのインストールディレクトリ>¥CC¥admin¥bin¥cjstopapp
”cmx_MyWebSystem_unit1_J2EE_01” -resname ”JMSSampleApp”
```

UNIX の場合

```
/opt/Cosminexus/CC/admin/bin/cjstopapp ”cmx_MyWebSystem_unit1_J2EE_01” -resname
”JMSSampleApp”
```

2. CJMSP リソースアダプタを停止します。

サーバ管理コマンドを使用した実行例を次に示します。なお，J2EE サーバの名称は「cmx_MyWebSystem_unit1_J2EE_01」，CJMSP リソースアダプタの名称は「CJMS_Provider_RA」とします。

実行例

Windows の場合

```
<Application Serverのインストールディレクトリ>¥CC¥admin¥bin¥cjstoprar
”cmx_MyWebSystem_unit1_J2EE_01” -resname ”CJMS_Provider_RA”
```

UNIX の場合

```
/opt/Cosminexus/CC/admin/bin/cjstoprar ”cmx_MyWebSystem_unit1_J2EE_01” -resname
”CJMS_Provider_RA”
```

3. CJMSP ブローカーを停止します。

CJMSP ブローカー「MyBroker」を停止する場合の実行例を示します。

実行例

Windows の場合

```
<Application Serverのインストールディレクトリ>¥CC¥cjmsp¥bin¥cjmsicmd shutdown bkr
```

UNIX の場合

```
/opt/Cosminexus/CC/cjmsp/bin/cjmsicmd shutdown bkr
```

ポイント

CJMSP ブローカーを停止する場合は，ホスト名とポート番号によって対象にする CJMSP ブローカーを特定します。ポート番号を指定しないで cjmsicmd shutdown bkr コマンドを実行すると，デフォルトの 7676 のポートを使用している CJMSP ブローカーが停止処理の対象となります。ほかのポートを使用している CJMSP ブローカーを停止する場合は，-b オプションでホスト名とポート番号を指定してください。実行例を次に示します。

実行例（7777 のポートを使用している CJMSP ブローカーを停止する場合）

```
cjmsicmd shutdown bkr -b localhost:7777
```

詳細は，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjmsicmd shutdown bkr（CJMSP ブローカーの停止）」を参照してください。

4. J2EE サーバを含む J2EE アプリケーションの実行環境の各プロセスを停止します。

Smart Composer で構築したシステムの場合は，一括停止ができます。詳細は，マニュアル「アプリケーションサーバ 運用管理ポータル操作ガイド」の「4.1.3　システムの停止手順」を参照してください。

7.17.3　CJMSP ブローカーの状態の確認

ここでは，CJMSP ブローカーの状態遷移および状態の確認方法について説明します。CJMSP ブローカーには，3 種類の状態があります。コマンドの実行などによって状態が遷移します。

CJMSP ブローカーの状態は，cjmsicmd list bkr コマンドを使用して確認できます。詳細は，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjmsicmd list bkr（CJMSP ブローカーの一覧表示）」を参照してください。

コマンド実行による CJMSP ブローカーの状態遷移を次に示します。

図 7-14　CJMSP ブローカーの状態遷移

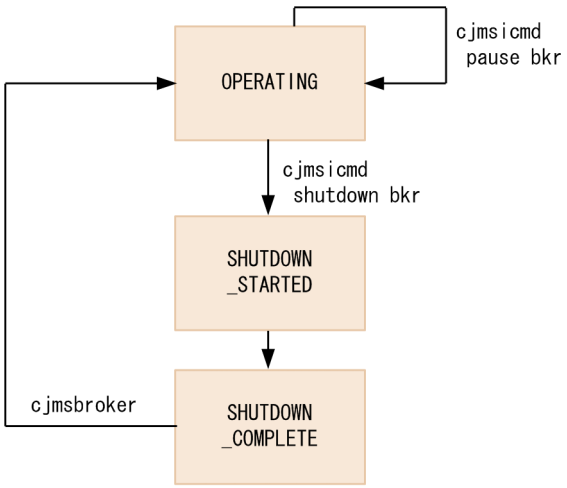


表 7-13　CJMSP ブローカーの状態

CJMSP ブローカーの状態	説明
OPERATING	CJMSP ブローカーがメッセージを処理できる状態です。 cjmsbroker コマンドによって CJMSP ブローカーを起動するとこの状態になります。なお，OPERATING の状態の CJMSP ブローカーに対して cjmsicmd pause bkr コマンドを実行した場合，状態は変わりません。
SHUTDOWN_STARTED	CJMSP ブローカーが停止処理（シャットダウン）を開始した状態です。直後または数分後に，CJMSP ブローカーはシャットダウンされます。 cjmsicmd shutdown bkr コマンドを実行するとこの状態になります。
SHUTDOWN_COMPLETE	CJMSP ブローカーの停止（シャットダウン）処理が完了した状態です。すべてのリソースは解放されています。CJMSP ブローカーがシャットダウンされるかどうかは，実行中のプロセスがあるかどうかに依存します。 cjmsicmd shutdown bkr コマンドを実行するとこの状態になります。

7.18　障害対応用の情報の確認

この節では，CJMS プロバイダ使用時に障害が発生した場合に使用する，障害対応用の情報について説明します。

CJMS プロバイダでは，次の 3 種類のコンポーネントがログを出力します。

- CJMSP リソースアダプタ
- CJMSP ブローカー
- 管理コマンド（cjmsicmd）

これらのコンポーネントは，メッセージログと例外ログを出力します。メッセージログと例外ログは，コンポーネントごとのログファイルに出力されます。

ログの形式は，トレース共通ライブラリ形式です。

メッセージログ

メッセージログには，J2EE アプリケーションから実行された操作が原因で発生した状態が出力されます。

出力するメッセージの種類は，ログレベルによって設定します。

CJMS プロバイダでのログレベルの意味を次の表に示します。

表 7-14　CJMS プロバイダでのログレベルの意味

ログレベル	意味
ERROR	システムの障害につながる深刻な問題が発生したことを示すメッセージのログレベルです。 エラーと例外の情報だけの，最小限の情報が出力されます。 このログレベルを指定した場合は，ERROR レベルに該当するメッセージだけがログ出力されます。
WARNING	システムの障害にはつながらないものの，注意が必要な状態であることを示すメッセージのログレベルです。 処理が正常終了し，状態に問題がある場合に出力されます。 このログレベルを指定した場合は，ERROR レベルと WARNING レベルに該当するメッセージがログ出力されます。
INFO	メトリクスやそのほかのメッセージを出力するログレベルです。 処理の流れや，通常処理でどのように操作が行われているかを示す情報が出力されます。 このログレベルを指定した場合は，ERROR レベル，WARNING レベルおよび INFO レベルに該当するメッセージがログ出力されます。

注

重要なメッセージについては，ログレベルの設定に関係なく，コンソールまたはログファイルに出力されることがあります。

例外ログ

例外ログには，J2EE アプリケーション実行時にエラーまたは例外が発生した場合の情報（例外メッセージ）が出力されます。

ログファイルに出力されるメッセージの詳細については，マニュアル「アプリケーションサーバ メッセージ(構築／運用／開発用)」の「4.　KDAN（CJMS プロバイダで出力されるメッセージ)」を参照してくだ

さい。また，トレース共通ライブラリ形式については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「5.2.1　トレース共通ライブラリ形式のログの出力形式と出力項目」を参照してください。

参考

- CJMS プロバイダでは，PRF トレースも取得できます。CJMS プロバイダ使用時の PRF トレースの取得ポイントについては，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「8.　性能解析トレースのトレース取得ポイントと PRF トレース取得レベル」を参照してください。
- CJMSP ブローカーが JavaVM の障害で停止した場合は，次のディレクトリにスレッドダンプまたは core ファイルが出力されます。

  Windows の場合

  <Application Serverのインストールディレクトリ>¥CC¥cjmsp¥var¥instances

  UNIX の場合

  /opt/Cosminexus/CC/cjmsp/var/instances

7.19 CJMS プロバイダ使用時の注意事項

この節では，CJMS プロバイダ使用時の注意事項について説明します。

(1) デフォルトの設定での CJMS プロバイダの動作

デフォルトの設定では，CJMS プロバイダは次のように動作します。

- CJMSP ブローカーが管理しているメッセージの数またはメッセージの容量が，すべての物理的送信先に対して設定した最大値に到達した場合，CJMSP ブローカーはメモリリソースを確保するために，最新のメッセージの受け付けを拒否します。また，そのメッセージが永続化メッセージの場合にだけ，メッセージを送信したクライアントに受け付けを拒否したことを通知します。
- デッドメッセージキューに送信された有効期限を過ぎたメッセージは，60 秒間保留されます。この期間を過ぎたメッセージをデッドメッセージキューから回復することはできません。
- 使用されなくなった一時的送信先は，120 秒後に破棄されます。
- 永続化データのファイルへの格納は，非同期で実行されます。このため，突然 CJMSP ブローカーが故障したり停止したりした場合，データが失われることがあります。

(2) メッセージ処理の優先順位についての注意事項

CJMS プロバイダのメッセージ送信では，優先順位の高いメッセージが最初に実行されることが保証されません。例えば，優先順位を 1，2，3 と設定した「メッセージ 1」「メッセージ 2」および「メッセージ 3」というの三つのメッセージをプロデューサーから送信した場合に，コンシューマーでは「メッセージ 1」「メッセージ 3」「メッセージ 2」のように異なる順番で処理が実行されることがあります。

優先順位と処理順の相違は，次のような要因で発生します。

- プロデューサーが先に優先順位の低いメッセージを送信したあとで，優先順位が高いメッセージを送信した場合，あとから送ったメッセージが届く前にコンシューマーではすでに優先順位の低いメッセージを受信し，処理を開始していることがあります。
- コンシューマーが優先順位の高いメッセージを受信する前に，そのメッセージの有効期限が過ぎた場合，優先順位の高いメッセージの処理は実行されません。
- プロデューサーが優先順位に沿ってメッセージを送信した場合も，内部処理の遅延などによって，コンシューマーに送信される順序が変わることがあります。
- 優先順位が高いメッセージが永続化メッセージでない場合，CJMSP ブローカーに障害が発生すると，そのメッセージは失われます。この場合，CJMSP ブローカーが再起動したあとでは，優先度が低い永続化メッセージの処理だけが実行されます。

(3) 有効期限が過ぎたメッセージの扱い

送信先があるマシンのシステム時間を変更した場合，クライアントは期限切れのメッセージを受信することがあります。

メッセージの有効期限は，JMSExpiration ヘッダフィールドにメッセージを送信したメソッドによって設定されます。もし，誤ってシステムの時間設定が変更されていた場合，有効期限が過ぎたメッセージがコンシューマーに送信されるおそれがあります。

(4) メッセージプロパティの順序性

メッセージプロパティの処理順序は保証されません。

メッセージプロパティは，順序性を維持した構造で管理されていません。このため，例えば，「プロパティ 1」「プロパティ 2」「プロパティ 3」というプロパティがある場合に，この順序で送信先に送信しても，同じ順序で処理されるとは限りません。ただし，プロパティとその値は，送信前と受信時で変わりません。

(5)　メッセージプロパティでのバイトメッセージの扱い

メッセージプロパティでは，バイトメッセージを使用しないでください。

バイトメッセージでは，すべてのデータ型のデータを読み出すことができます。例えば，long 型の値を含むメッセージを short 型のメソッドで呼び出して処理した場合，意味的には不正な値ですが，最初の 2 バイトの読み出しには成功してしまいます。これは，ほかのメッセージのデータ型では禁止されている処理です。

CJMS プロバイダでは，データのデータ型を保持し，正しい変換規則の適用に対応するよう，データを管理しています。

(6)　BytesMessage インタフェースでのメッセージの扱い

BytesMessage インタフェースの writeUTF()メソッドを使用した場合，65,535 バイトより大きいメッセージを書き込まないでください。writeUTF()メソッドに文字列長が 65,535 バイトより大きい文字列を渡した場合，例外が発生します。

(7)　デッドメッセージキューでのメッセージの扱い

デッドメッセージキューから受け取ったメッセージに対してロールバックを繰り返すと，そのメッセージは送信されない状態になり，そのままデッドメッセージキューに残ります。デッドメッセージキューに残ったメッセージを再度受信するためには，CJMSP ブローカーを再起動してください。

8 JavaMail の利用

この章では，SMTP プロバイダおよび SMTPS プロバイダ独自のセッションプロパティ，POP3 プロバイダ独自のセッションプロパティ，および JavaMail を利用する場合の注意事項について説明します。

8.1　この章の構成

この章では，SMTP プロバイダおよび SMTPS プロバイダ独自のセッションプロパティ，および POP3 プロバイダ独自のセッションプロパティについて説明します。

この章の構成を次の表に示します。

表 8–1　この章の構成（セッションプロパティ）

分類	タイトル	参照先
実装	セッションプロパティ	8.2
注意事項	JavaMail を利用する場合の注意事項	8.3

注　「解説」,「設定」，および「運用」について，この機能固有の説明はありません。

8.2　セッションプロパティ

JavaMail では，javax.mail.Session クラス作成時に，メールの送受信に関する設定をセッションプロパティとして設定します。

セッションプロパティには，次に示す種類があります。

- JavaMail API で提供するセッションプロパティ
- JavaMail 仕様で定められた SMTP プロバイダおよび SMTPS プロバイダが提供するセッションプロパティ
- JavaMail 仕様で定められた POP3 プロバイダが提供するセッションプロパティ
- SMTP プロバイダおよび SMTPS プロバイダ独自のセッションプロパティ
- POP3 プロバイダ独自のセッションプロパティ

セッションプロパティの指定値は，「"」で囲み文字列として指定する必要があります。

なお，ここでは SMTP プロバイダおよび SMTPS プロバイダ独自のセッションプロパティ，および POP3 プロバイダ独自のセッションプロパティについて説明します。

8.2.1　SMTP サーバに接続するためのセッションプロパティ

ここでは，SMTP プロバイダおよび SMTPS プロバイダ独自のセッションプロパティについて説明します。

SMTP プロバイダおよび SMTPS プロバイダ独自のセッションプロパティの一覧を次の表に示します。なお，使用するプロトコルとして"smtps"を指定した場合は，プロパティ名の"smtp"の部分は"smtps"となります。

表 8–2　SMTP プロバイダおよび SMTPS プロバイダ独自のセッションプロパティの一覧

項番	プロパティ	デフォルト値	有効な指定値	無効な値を指定した場合の動作
1	mail.smtp.allow8bitmime	false	true，false （大文字と小文字の区別はなし）	メールサーバへの送信処理でメッセージ KDJE59100-W を出力します。 値は，デフォルト値を使用します。
2	mail.smtp.auth	false	true，false （大文字と小文字の区別はなし）	メールサーバとの接続処理でメッセージ KDJE59100-W を出力します。 値は，デフォルト値を使用します。
3	mail.smtp.connectiontimeout	0 （無限待ち）	0～2147483647 （単位:ミリ秒）	メールサーバとの接続処理でメッセージ KDJE59100-W を出力します。 値は，デフォルト値を使用します。
4	mail.smtp.dsn.notify	なし	• SUCCESS，FAILURE，DELAY のうち一つ，または複数を「,」で区切って指定 • NEVER	指定した値をそのまま使用します。 メールサーバ側で不正と見なした場合は，RCPT コマンドが失敗し，メール送信処理が中断します。 メールサーバの処理は，メールサーバに依存します。

項番	プロパティ	デフォルト値	有効な指定値	無効な値を指定した場合の動作
5	mail.smtp.dsn.ret	なし	FULL，HDRS	指定した値をそのまま使用します。 メールサーバ側で不正と見なした場合は，MAIL コマンドが失敗し，メール送信処理が中断します。 メールサーバの処理は，メールサーバに依存します。
6	mail.smtp.ehlo	true	true，false （大文字と小文字の区別はなし）	メールサーバとの接続処理でメッセージ KDJE59100-W を出力します。 値は，デフォルト値を使用します。
7	mail.smtp.from	<mail.form>に指定した値	文字列	JavaMail，またはメールサーバから例外が返る可能性があります。 メールサーバがどのような場合に例外を返すかは，メールサーバに依存します。
8	mail.smtp.localhost	java.net.InetAddress.getLocalHost().getHostName()の戻り値	文字列	指定した値をそのまま使用します。
9	mail.smtp.noop.strict	true	true，false （大文字と小文字の区別はなし）	javax.mail.Transport オブジェクト取得時にメッセージ KDJE59100-W を出力します。 値は，デフォルト値を使用します。
10	mail.smtp.quitwait	false	true，false （大文字と小文字の区別はなし）	javax.mail.Transport オブジェクト取得時にメッセージ KDJE59100-W を出力します。 値は，デフォルト値を使用します。
11	mail.smtp.saslrealm	なし	文字列	指定した値をそのまま使用します。
12	mail.smtp.sendpartial	false	true，false （大文字と小文字の区別はなし）	メールサーバへの送信処理でメッセージ KDJE59100-W を出力します。 値は，デフォルト値を使用します。
13	mail.smtp.timeout	0 （無限待ち）	0～2147483647 （単位:ミリ秒）	メールサーバとの接続処理でメッセージ KDJE59100-W を出力します。 値は，デフォルト値を使用します。

それぞれのプロパティについて，以降で説明します。

(1)　mail.smtp.allow8bitmime

quoted-printable や base64 エンコーディングを使用するメッセージのテキスト部分を，8 ビットエンコーディングへ変換するかどうかを指定します。

true を指定した場合，次の条件を満たしていれば，quoted-printable や base64 エンコーディングを使用するメッセージのテキスト部分は 8 ビットエンコーディングへ変換されます。

- サーバが 8 ビット転送をサポートしている
- quoted-printable や base64 エンコーディングを使用するメッセージのテキスト部分が RFC2045 に準拠している

false を指定した場合は，8 ビットエンコーディングへは変換しません。

(2)　mail.smtp.auth

AUTH コマンドを使用したユーザ認証をするかどうかを指定します。

true を指定した場合，AUTH コマンドを使用してユーザ認証をします。false を指定した場合，AUTH コマンドを使用したユーザ認証はしません。

(3)　mail.smtp.connectiontimeout

コネクション確立までのタイムアウト時間（単位：ミリ秒）を指定します。

0 を指定すると，無限に待ち続けます。

このプロパティが指定されている場合, スレッドを使用してタイムアウト時間を監視します。ポリシーやシステムの制限でスレッドが作成できない場合は，タイムアウト時間を監視しません。

タイムアウト時間が，OS の TCP の再送タイマによるタイムアウト時間より大きいか，またはタイムアウト監視をしない設定の場合，TCP の再送タイマによるタイムアウト時間が有効となります。なお，再送タイマによるタイムアウトは，OS によって異なります。

(4)　mail.smtp.dsn.notify

RCPT コマンドの NOTIFY オプションを指定します。

このオプションは DSN（Delivery Status Notification：配送状態通知）を生成する条件を指定します。コンマを区切りにして，次の表に示す値を組み合わせて指定できます。

表 8–3　mail.smtp.dsn.notify に指定できる値

項番	値	指定される内容
1	NEVER	配送状態に関係なく，送信者に DSN を返さない。
2	SUCCESS	メッセージの配送が成功した場合，DSN を生成する。
3	FAILURE	メッセージの配送が失敗した場合，DSN を生成する。
4	DELAY	メッセージの配送が遅延した場合，DSN を生成する。

指定しない場合の DSN 生成については，RFC1891 では NOTIFY=FAILURE または NOTIFY=FAILURE,DELAY として指定されたと解釈してよいとされており，サーバの実装に依存します。

(5)　mail.smtp.dsn.ret

MAIL コマンドの RET 追加パラメタ値を指定します。

次の表に示す値を組み合わせて指定できます。

表 8–4　mail.smtp.dsn.ret 指定できる値

項番	値	指定される内容
1	FULL	配送失敗通知メッセージに送信メッセージ全体を含める。
2	HDRS	配送失敗通知メッセージに送信メッセージのヘッダだけを含める。

プロパティを指定しない場合，配送失敗通知メッセージに何が含まれるかは，サーバの実装に依存します。

(6)　mail.smtp.ehlo

EHLO コマンドを使用するかどうかを指定します。

true を指定した場合，EHLO コマンドを使用します。EHLO コマンドでエラーが発生した場合，代替に HELO コマンドを使用します。false を指定した場合，EHLO コマンドは使用しません。

(7)　mail.smtp.from

SMTP および SMTPS の MAIL コマンドの reverse-path に使用する電子メールアドレスを指定します。

(8)　mail.smtp.localhost

SMTP および SMTPS の HELO コマンドまたは EHLO コマンドで使用されるローカルホストのドメイン名を指定します。

JDK およびネームサービスが適切に定義されている場合，指定する必要はありません。

(9)　mail.smtp.noop.strict

メールサーバとのコネクション状態を確認する場合, 接続状態と判定するレスポンスを変更する時に使用します。

true を指定した場合は，NOOP コマンドに対するメールサーバからのレスポンスが 250 の時に接続状態と見なします。false を指定した場合は，NOOP コマンドに対するメールサーバからのレスポンスが，421 を除く正の整数値の時に接続状態と見なします。

(10)　mail.smtp.quitwait

QUIT コマンドに対する応答を待つかどうかを指定します。

true を指定した場合は，QUIT コマンドに対する応答を待ちます。false を指定した場合は，QUIT コマンド送信直後にコネクションをクローズします。

(11)　mail.smtp.saslrealm

DIGEST-MD5 認証で使用するレルムを指定します。

省略した場合, サーバからのチャレンジに含まれる最初のレルムを使用します。チャレンジにレルムが含まれていない場合，接続先ホスト名をレルムとして使用します。

(12)　mail.smtp.sendpartial

メッセージに指定されている電子メールアドレスに, 正しくない電子メールアドレスが含まれている場合の動作を指定します。

true を指定した場合は，メッセージを送信した上で，エラーになった電子メールアドレスを SendFailedException で報告します。false を指定した場合は，メッセージを送信しません。

(13)　mail.smtp.timeout

SMTP サーバとの通信（read）のタイムアウト時間（単位：ミリ秒）を指定します。

0 を指定すると，無限に待ち続けます。

タイムアウト時間が，OS の TCP の再送タイマによるタイムアウト時間より大きいか，またはタイムアウト監視をしない設定の場合，TCP の再送タイマによるタイムアウト時間が有効となります。なお，再送タイマによるタイムアウトは，OS によって異なります。

8.2.2　POP3 サーバに接続するためのセッションプロパティ

ここでは，POP3 プロバイダ独自のセッションプロパティについて説明します。

POP3 プロバイダ独自のセッションプロパティの一覧を次の表に示します。

表 8–5　POP3 プロバイダ独自のセッションプロパティの一覧

項番	プロパティ	デフォルト値	有効な指定値	無効な値を指定した場合の動作
1	mail.pop3.connectiontimeout	0 （無限待ち）	0～2147483647 （単位:ミリ秒）	メールサーバとの接続処理でメッセージ KDJE59100-W を出力します。 値は，デフォルト値を使用します。
2	mail.pop3.rsetbeforequit	false	true，false （大文字と小文字の区別はなし）	javax.mail.Store オブジェクト取得時にメッセージ KDJE59100-W を出力します。 値は，デフォルト値を使用します。
3	mail.pop3.timeout	0 （無限待ち）	0～2147483647 （単位:ミリ秒）	メールサーバとの接続処理でメッセージ KDJE59100-W を出力します。 値は，デフォルト値を使用します。

それぞれのプロパティについて，以降で説明します。

(1)　mail.pop3.connectiontimeout

コネクション確立までのタイムアウト時間（単位：ミリ秒）を指定します。

0 を指定すると，無限に待ち続けます。

タイムアウト時間が，OS の TCP の再送タイマによるタイムアウト時間より大きいか，またはタイムアウト監視をしない設定の場合，TCP の再送タイマによるタイムアウト時間が有効となります。なお，再送タイマによるタイムアウトは，OS によって異なります。

(2)　mail.pop3.rsetbeforequit

フォルダを閉じる場合に，QUIT コマンドの前に RSET コマンドを発行するかどうかを指定します。

true を指定した場合，QUIT コマンドの前に RSET コマンドを発行し，メッセージに付けた削除マークを取り除きます。自動的に削除マークが付けられるようなサーバの場合に，ユーザのリクエストなしにメッセージが削除されることを防げます。

false を指定した場合，QUIT コマンドの前に RSET コマンドを発行しません。

(3)　mail.pop3.timeout

POP3 サーバとの通信（read）のタイムアウト時間（単位：ミリ秒）を指定します。

0 を指定すると，無限に待ち続けます。

タイムアウト時間が，OS の TCP の再送タイマによるタイムアウト時間より大きいか，またはタイムアウト監視をしない設定の場合，TCP の再送タイマによるタイムアウト時間が有効となります。なお，再送タイマによるタイムアウトは，OS によって異なります。

8.3　JavaMail を利用する場合の注意事項

この節では，JavaMail を利用する場合の注意事項について説明します。

(1)　メソッドの例外発生について

次のクラスのメソッドでは，JavaMail 仕様どおりに例外が発生しません。

項番	クラス名	メソッド名
1	javax.mail.Message	setFlag，setFlags
2	javax.mail.Folder	setFlags

READ_ONLY を設定した javax.mail.Folder オブジェクトから取得した javax.mail.Message オブジェクトに対して，表中に示したメソッドでフラグの値を変更しても，READ_WRITE を設定したときと同様に正しくフラグが設定されます。

ただし，POP3 の場合，READ_ONLY でフォルダをオープンし，メッセージに DELETED フラグを設定しても，実際にはメールは削除されません。

(2)　POP3 での SEEN フラグの動作について

JavaMail 仕様では Message の getInputStream メソッドと getContent メソッドを使用した場合，SEEN フラグが設定される仕様となっていますが，実際には SEEN フラグは設定されません。

(3)　JavaMail のプロバイダについて

アプリケーションサーバで使用できる JavaMail のプロバイダは，アプリケーションサーバがデフォルトで提供する SMTP，SMTPS および POP3 のプロバイダだけです。javamail.providers ファイルを使用して別のプロバイダに差し替えた場合の動作は保証しません。

(4)　レスポンスコードについて

JavaMail が不正と見なすレスポンスコードは厳密な RFC 仕様とは異なり，RFC 仕様で不正と見なすレスポンスコードを不正と見なさない場合があります。そのため，RFC 仕様以外の不正なレスポンスを受け取ったことを示すメッセージ（KDJE59111-E または KDJE59112-E）ではなく，異常値のレスポンスコードを受け取ったことを示すメッセージ（KDJE59107-E，KDJE59108-E，KDJE59109-E または KDJE59110-E）を出力する場合があります。

(5)　アプリケーションでメールサーバとの接続およびメールの受信をしたときの注意

アプリケーションでメールサーバとの接続およびメールの受信をしたあとに，次のメソッドを呼ばないでアプリケーションの停止またはリロードを実行すると，KDJE59106-E メッセージを出力する場合があります。

- javax.mail.Store クラスの close メソッド
- javax.mail.Folder クラスの close メソッド

この場合，それぞれの close メソッドを呼んでいるかどうかを確認してください。

なお，KDJE59106-E メッセージは出力しますが，アプリケーションの停止およびリロードの処理には影響ありません。また，JavaMail の動作にも影響はありません。

(6)　JavaMail の仕様との相違点について

メールのヘッダで，Reply-To ヘッダフィールドに「Reply-To:」だけが指定され，アドレスの指定がない場合，アプリケーションサーバの JavaMail では，javax.mail.MimeMessage クラスの getReplyTo メソッドに，getFrom メソッドの戻り値を返さないで，長さ 0 の javax.mail.Address オブジェクトの配列を返します。

(7)　MimeUtility API の挙動差について

javax.mail.internet.MimeUtility クラスの encode メソッドでエンコード方式に"base64"を用いた場合，Component Container のバージョンおよび戻り値として得た OutputStream に対する close メソッドの実行有無により，末尾への改行（"¥r¥n"）付与の有無が次のように異なります。

Component Container のバージョン	OutputStream.close メソッドの実行の有無	末尾への改行付与
09-00 以前	有	付与しない
	無	付与しない
09-50 以降	有	付与する
	無	付与しない

MimeUtility.encode メソッドの戻り値として得た OutputStream の末尾に不要な改行が存在する場合は，改行文字を削除してください。

9 アプリケーションサーバでの CDI の利用

この章では，CDI の機能概要，実装方法，および注意事項について説明します。

9.1　この章の構成

この章では，CDI の機能について説明します。この章の構成を次の表に示します。

表 9-1　この章の構成（アプリケーションサーバでの CDI の利用）

分類	タイトル	参照先
解説	CDI の概要	9.2
	CDI を利用したアプリケーションの開発	9.3
実装	クラスパスの設定（開発環境の設定）	9.4
	CDI を利用するための設定（セキュリティの設定変更）	9.5
	デバッグ用ログ（開発調査ログ）の利用	9.6
設定	実行環境の設定	9.7
注意事項	CDI を利用する場合の注意事項	9.8

注　「運用」について，この機能固有の説明はありません。

9.2　CDI の概要

(1)　CDI とは

CDI は，Dependency Injection（DI）の仕様の一つです。アプリケーションサーバでは，CDI 1.0 に対応しています。

CDI を利用すると，アプリケーションのオブジェクトの DI や，DI 対象のオブジェクトの生成から破棄までのライフサイクルを管理できます。アプリケーションサーバでは，次に示す CDI の機能を提供します。

- クラスの型による Bean の解決
- 名前（文字列）による Bean の解決
- アノテーションによって指定されたコンテキストに基づくインスタンスの生成，および破棄
- EL（式言語）やアノテーションで指定されたほかの Bean インスタンスの自動的なインジェクション（注入）
- ライフサイクルコールバックインターセプタやインターセプタ

ポイント

CDI は，アプリケーションのリソース間でのインジェクション（注入）を実現することで，アプリケーション開発時の開発容易性を高める機能です。実現するためには，アプリケーションの任意のリソースを注入できるように J2EE サーバのセキュリティの設定を変更する必要があります。セキュリティの設定変更の詳細は，「9.5　CDI を利用するための設定（セキュリティの設定変更）」を参照してください。

(2)　Bean Validation との連携

アプリケーションサーバでは，CDI アプリケーションでの入力値の検証処理を簡略化するために，Bean Validation の機能を利用できます。

Bean Validation の利用に関しては，「10.　アプリケーションサーバでの Bean Validation の利用」を参照してください。

9.3　CDI を利用したアプリケーションの開発

ここでは，CDI を利用したアプリケーションを開発するために必要な beans.xml ファイルや，アプリケーションのバージョンについて説明します。

9.3.1　CDI の対象となるアプリケーション

CDI の対象となるアプリケーションは，beans.xml ファイルを含んだ J2EE モジュールを含むアプリケーションです。J2EE モジュールの種類ごとに，beans.xml ファイルの格納先を示します。

表 9-2　J2EE モジュールの種類ごとの beans.xml の格納先

J2EE モジュールの種類	beans.xml の格納先
EJB-JAR	META-INF/beans.xml
ライブラリ JAR	META-INF/beans.xml
WAR	WEB-INF/beans.xml

CDI の対象となる J2EE モジュールは，beans.xml ファイルを格納しないと CDI の管理対象となりません。表に示す格納先に beans.xml ファイルを必ず格納してください。

CDI の対象にできるコンポーネントのバージョン，および CDI を使用できるオブジェクトについて次の表に示します。

表 9-3　CDI の対象にできるコンポーネントのバージョン，および CDI を使用できるオブジェクト

コンポーネントの種類	サポートしているコンポーネントのバージョン	使用できるオブジェクト
EJB	EJB3.1 （ejb-jar.xml のバージョンが 3.1）	Session Bean
Servlet	Servlet3.0 （web.xml のバージョンが 3.0）	• Servlet • Listener • Filter
JSP	JSP2.1 （web.xml のバージョンが 3.0）	EL の構文に対応するオブジェクト
JSF	JSF2.1 （web.xml のバージョンが 3.0，およびアプリケーションサーバがサポートする JSF フレームワーク）	EL の構文に対応するオブジェクト

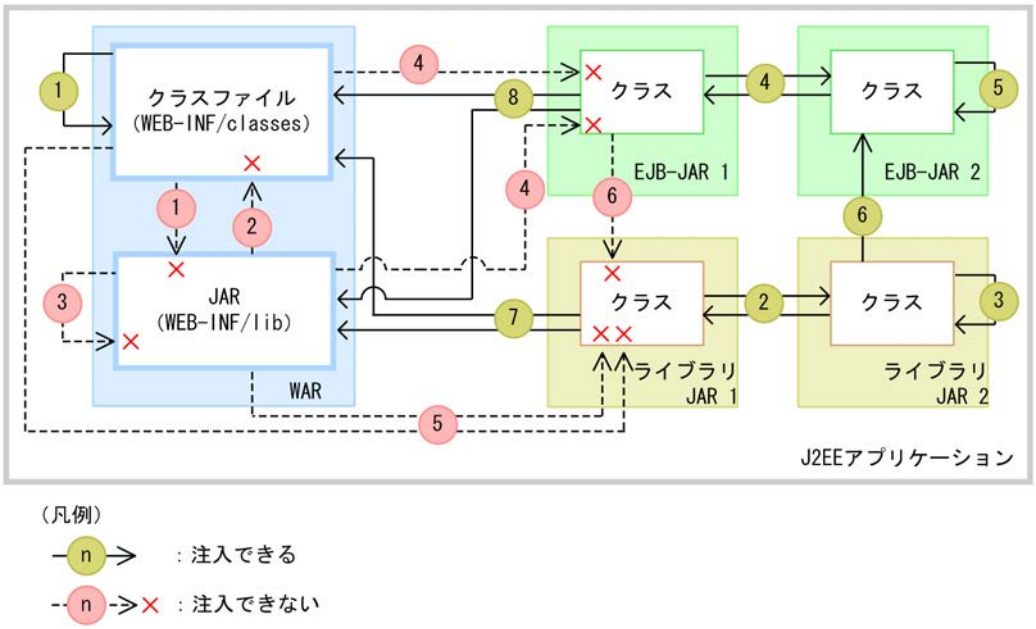
注　JAX-RS，JAX-WS，JPA など，上記以外のコンポーネントを使用する場合は，CDI の対象にできません。

9.3.2　CDI の対象となる J2EE モジュールの注入関係

CDI の対象となる J2EE モジュールの注入関係について説明します。

CDI の対象となる J2EE モジュールの注入関係について，次の図に示します。

図 9-1　CDI の対象となる J2EE モジュールの注入関係



CDI の対象として注入できる関係，および注入できない関係について次に示します。

CDI の対象として注入できる関係

1. WAR の WEB-INF/classes から，同じ WAR 内の WEB-INF/classes への注入
2. ライブラリ JAR に含まれるクラスから，ほかのライブラリ JAR に含まれるクラスへの注入
3. ライブラリ JAR に含まれるクラスから，同じライブラリ JAR に含まれるクラスへの注入
4. EJB-JAR に含まれるクラスから，ほかの EJB-JAR に含まれるクラスへの注入（ただし，EJB の注入はできません）
5. EJB-JAR に含まれるクラスから，同じ EJB-JAR に含まれるクラスへの注入（ただし，EJB の注入はできません）
6. ライブラリ JAR に含まれるクラスから，EJB-JAR に含まれるクラスへの注入
7. ライブラリ JAR に含まれるクラスから，WAR（WEB-INF/classes および WEB-INF/lib 以下の JAR）に含まれるクラスへの注入
8. EJB-JAR に含まれるクラスから WAR（WEB-INF/classes および WEB-INF/lib 以下の JAR）に含まれるクラスへの注入

CDI の対象として注入できない関係

1. WAR の WEB-INF/classes に含まれるクラスから，同じ WAR 内の WEB-INF/lib 以下の JAR に含まれるクラスへの注入
2. WAR の WEB-INF/lib 以下の JAR に含まれるクラスから，同じ WAR 内の WEB-INF/classes に含まれるクラスへの注入
3. WAR の WEB-INF/lib 以下の JAR に含まれるクラスから，同じ WAR 内の WEB-INF/lib 以下の JAR に含まれるクラスへの注入
4. WAR（WEB-INF/classes および WEB-INF/lib 以下の JAR）に含まれるクラスから，EJB-JAR に含まれるクラスへの注入

5. WAR（WEB-INF/classes および WEB-INF/lib 以下の JAR）に含まれるクラスから，ライブラリ JAR に含まれるクラスへの注入
6. EJB-JAR に含まれるクラスから，ライブラリ JAR に含まれるクラスへの注入

9.3.3　開発時の注意事項

CDI を利用したアプリケーションを開発する際の注意事項を次に示します。

● CDI を使用するアプリケーションでのコンストラクタの取り扱い

CDI のアノテーションを使用する EJB の Bean クラス，Servlet クラス，Listener クラス，および Filter クラスのコンストラクタに引数は指定できません。

9.4　クラスパスの設定（開発環境の設定）

この節では，CDI の利用のために必要なクラスパスの設定について説明します。

9.4.1　ファイルの格納先

アプリケーションサーバのインストール時にインストールされる CDI に関連するファイルとその格納先を次の表に示します。

表 9–4　CDI ライブラリの格納先

ファイル名	クラスパス
cjsf.jar	<アプリケーションサーバのインストールディレクトリ>¥CC¥lib¥cjsf.jar
cjstl.jar	<アプリケーションサーバのインストールディレクトリ>¥CC¥lib¥cjstl.jar

注　UNIX の場合は，「<アプリケーションサーバのインストールディレクトリ>」を「/opt/Cosminexus」に，「¥」を「/」に読み替えてください。

9.4.2　クラスパスの設定

CDI を使用したアプリケーションをアプリケーションサーバで動かすためには，J2EE サーバの usrconf.cfg（Java アプリケーション用オプション定義ファイル）の add.class.path キーにクラスパスを設定する必要があります。

例を示します。この例は，Windows の場合の例です。

```
add.class.path=<アプリケーションサーバのインストールディレクトリ>¥CC¥lib¥cjsf.jar
add.class.path=<アプリケーションサーバのインストールディレクトリ>¥CC¥lib¥cjstl.jar
```

! 注意事項

異なるバージョンのライブラリを同時にクラスパスに指定すると，アプリケーションが予期しない動作をするおそれがあるため，異なるバージョンのライブラリを同時にクラスパスに指定しないてください。アプリケーションのバージョンに合わせたライブラリをクラスパスに指定してください。

9.5　CDI を利用するための設定 (セキュリティの設定変更)

ここでは，CDI を利用するために必要なセキュリティの設定変更について説明します。

CDI を使用する場合，次のどちらかの方法でセキュリティの設定を変更する必要があります。

- セキュリティポリシーの設定変更
- SecurityManager の解除

それぞれの方法について説明します。

(1)　セキュリティポリシーの設定を変更する

server.policy に「permission java.security.AllPermission;」を追加します。編集例を次に示します。

```
// Grant minimal permissions to everything else:
//      EJBs
//  client implementation classes
grant  {
        permission java.util.PropertyPermission "*", "read";
        permission java.lang.RuntimePermission "queuePrintJob";
        permission java.net.SocketPermission "*", "connect";
        permission java.security.AllPermission;
};
```

server.policy は，J2EE サーバを構築したあとに設定してください。

(2)　SecurityManager を解除する

J2EE サーバを起動するときに，cjstartsv コマンドに-nosecurity オプションを指定し，SecurityManager を解除します。

```
# cjstartsv <サーバ名称> -nosecurity
```

9.6　デバッグ用ログ（開発調査ログ）の利用

CDI を利用したアプリケーションの開発では，デバッグ用のログとして，**開発調査ログ**を利用できます。

開発している CDI アプリケーションで，障害や不具合などがあった場合，原因の調査および対象機能の詳細な動作を把握するために，開発調査ログを利用できます。

CDI を利用したアプリケーションでは，開発に関するメッセージが開発調査ログに出力されます。CDI を利用したアプリケーションで確認が必要なメッセージは，メッセージに出力されたクラス名から判断できます。メッセージを出力するコンポーネントと出力されるクラス名について，次の表に示します。

表 9-5　CDI を利用したアプリケーションの開発調査ログに出力されるクラス名

コンポーネント名	出力されるクラス名	出力内容
CDI	com.hitachi.software.cdi.weld.～ com.cosminexus.cc.lib.jboss_interceptor.v2_0_0_CR1.～	• 障害に関するメッセージ • 設定ミスを知らせるメッセージ • 例外のスタックトレース • 機能の動作状況に関するメッセージ • 内部状態に関するメッセージ

なお，開発調査ログはデフォルトの設定では出力されません。必要に応じて設定を変更してください。

開発調査ログを出力するための設定，および開発調査ログの出力先，出力形式，ログレベルなどについては，マニュアル「アプリケーションサーバ メッセージ(構築／運用／開発用)」の「24.4　開発調査ログに出力されるメッセージ」を参照してください。

また，CDI アプリケーションで利用している Bean Validation の開発調査ログの詳細については，「10.6 デバッグ用ログ（開発調査ログ）の利用」を参照してください。

9.7　実行環境の設定

CDI を利用したアプリケーションの実行環境の設定は，開発環境の設定と同じです。

次の個所を参照して設定してください。

- クラスパスの設定
  「9.4　クラスパスの設定（開発環境の設定）」
- セキュリティの設定変更
  「9.5　CDI を利用するための設定（セキュリティの設定変更）」

9.8　CDI を利用する場合の注意事項

CDI を利用する場合の注意事項を次に示します。

● CDI の拡張インタフェース（Portable Extension API）

アプリケーションサーバでは CDI の拡張インタフェース（Portable Extension API）は使用できません。

● アプリケーションサーバで CDI を利用する場合の注意事項（標準仕様との違い）

- Servlet3.0 の機能と CDI を組み合わせて利用した場合の，アプリケーションサーバでのサポート範囲について説明します。

  CDI アノテーションは，web.xml で定義した Servlet，Listener，Filter,およびアノテーションで定義した Servlet，Listener，Filter で利用できます。

  サーブレットリスナの javax.servlet.ServletContextListener の contextInitialized メソッドで，javax.servlet.ServletContext の次に示すメソッドを利用して動的に定義した Servlet および Filter でだけ，CDI のアノテーションを利用できます。

  - addFilter(java.lang.String filterName, java.lang.Class<? extends Filter> filterClass)
  - addFilter(java.lang.String filterName, java.lang.String className)
  - addServlet(java.lang.String servletName, java.lang.Class<? extends Servlet> servletClass)
  - addServlet(java.lang.String servletName, java.lang.String className)

  これら以外の API を使って動的に定義した Servlet，Filter，および Listener での CDI のアノテーションの利用はサポートしていません。
- JavaVM のクラスパスで指定した META-INF ディレクトリ内に beans.xml を配置している場合，JavaVM のクラスパス内のクラスは CDI の管理対象にはなりません。
- リソースアダプタ（RAR）の META-INF ディレクトリに beans.xml を配置している場合，beans.xml は読み込まれず，リソースアダプタは CDI のモジュールとは認識されません。
- WAR の WEB-INF/lib 内に配置した JAR 内の META-INF ディレクトリに beans.xml を配置している場合，beans.xml は読み込まれず，JAR ファイルは CDI のモジュールとは認識されません。なお，WEB-INF/lib 以下に格納された JAR ファイル内にある，META-INF/resources に含まれる JSP に対しては，ManagedBean を注入できます。
- CDI の ManagedBean では，@EJB，@Resource，@PersistenceContext，および@PersistenceUnit の利用はサポートしません。
- CDI のアノテーションを用いて Session Bean（ローカルインタフェースの省略されたものも含む）を注入することはできません。
- EJB を注入するために CDI のアノテーションを使用した場合, KDJE59316-E のエラーメッセージが出力され，注入に失敗します。
- EL 式を使用して, @Named を付与した EJB を JSF や JSP へ注入することはできません。@Named を付与した EJB を参照するために，EL 式を使用していた場合，KDJE59316-E のエラーメッセージが出力され，注入に失敗します。
- @Dependent スコープのオブジェクトで，@Produces アノテーション，および@Disposes アノテーションが付与されている場合，同じクラスの中でプロデューサーメソッドを参照することはできません。もし，このようなアプリケーションを作成した場合，再帰的に注入が行われ，StackOverflowException が発生する場合があります。

- @Produces アノテーション，または@Disposes アノテーションを付与した Bean では，@PostConstruct アノテーションもしくは@PreDestroy アノテーションを利用できません。
- CDI アプリケーションでは，インターセプタを利用することができます。しかし，EJB において，EJB のインターセプタと CDI のインターセプタを同時に利用することはできません。もし利用した場合，CDI のインターセプタは実行されません。

● CDI を使用するアプリケーションでのリロード機能の利用

CDI アプリケーションのリロードは，アプリケーション単位になります。CDI アプリケーションは WAR および JSP 単位でのリロードはできません。また，JSF で Facelets を使用している場合，リロードはできません。

● アプリケーション開始性能についての注意事項

CDI アプリケーションに含まれる beans.xml の JAR ファイルの数が増えると，その JAR ファイルの増分と比較してデプロイ時間が極端に増加します。

● Managed Bean を利用した場合の注意事項

javax.enterprise.context SessionScoped アノテーションを指定された Managed Bean のインスタンスは，明示管理ヒープには格納されません。

10 アプリケーションサーバでの Bean Validation の利用

この章では，Bean Validation の機能概要，実装方法について説明します。

10.1　この章の構成

この章では，Bean Validation の機能について説明します。

この章の構成を次の表に示します。

表 10-1　この章の構成（アプリケーションサーバでの Bean Validation の利用）

分類	タイトル	参照先
解説	Bean Validation の概要	10.2
	Bean Validation の適用可能な範囲	10.3
実装	Bean Validation の機能および Bean Validation の動作	10.4
	Bean Validation の利用手順	10.5
	デバッグ用ログ（開発調査ログ）の利用	10.6
運用	validation.xml の内容が不正な場合の情報の出力	10.7
注意事項	Bean Validation 実装時の注意事項	10.8

注　「設定」について，この機能固有の説明はありません。

10.2　Bean Validation の概要

この節では，Bean Validation の概要について説明します。

Bean Validation は，JavaBeans のプロパティにアノテーションで指定したプロパティ値の有効範囲に対して，実際にアプリケーションに入力された値が有効範囲内であるかを検証する機能です。

Bean Validation の目的は，アプリケーションでの入力値の検証処理を簡略化することです。

Bean Validation は，ユーザアプリケーション上で利用できます。また，JSF，CDI を使用したアプリケーションでは，標準仕様で定義された連携機能を利用できます。

JSF と CDI についての詳細は，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(Web コンテナ)」の「3.2　JSF および JSTL の概要」，および「9.2　CDI の概要」を参照してください。

10.3　Bean Validation の適用可能な範囲

Bean Validation との連携で適用可能な範囲は，JSF，CDI およびユーザアプリケーションです。

参照ライブラリおよびコンテナ拡張ライブラリは使用できません。

10.4　Bean Validation の機能および Bean Validation の動作

この節では，Bean Validation の機能および Bean Validation の動作について説明します。

Bean Validation の機能を次の表に示します。

表 10–2　Bean Validation の機能

項番	機能		概要
1	入力値の検証		JavaBean に設定した値を指定したバリデーション定義で検証する機能です。
		グループ管理機能	検証をグループ化する機能です。
		メッセージ管理機能	検証結果がエラーの場合に返却するメッセージの管理を行う機能です。
		ペイロード管理機能	検証結果をカテゴリ分割する機能です。※
2	カスタムバリデータ作成機能		独自の検証処理を作成する機能です。
3	ブートストラップ機能		Bean Validation プロバイダを変更できる機能です。

注※

Bean Validation を JSF と連携して利用した場合，ペイロード管理機能を使用できません。

Bean Validation の機能は，アノテーションの指定によって利用できます。Bean Validation 機能が提供するアノテーションクラス，指定可能な変数の型，および指定できない変数の型にアノテーションを指定した場合の動作を次の表に示します。

表 10–3　Bean Validation 機能が提供するアノテーションクラスとその変数の型

項番	アノテーションクラス	指定可能な変数の型	指定できない変数の型にアノテーションを指定した場合の動作	備考
1	Null	すべての型に指定可能	−	−
2	NotNull	すべての型に指定可能	−	−
3	AssertTrue	• boolean/java.lang.Boolean	javax.validation.UnexpectedTypeException がスローされます。	−
4	AssertFalse	• boolean/java.lang.Boolean	javax.validation.UnexpectedTypeException がスローされます。	−
5	Min	• java.math.BigDecimal • java.math.BigInteger • byte/java.lang.Byte • short/java.lang.Short • int/java.lang.Integer • long/java.lang.Long	javax.validation.UnexpectedTypeException がスローされます。	−

項番	アノテーションクラス	指定可能な変数の型	指定できない変数の型にアノテーションを指定した場合の動作	備考
5	Min	• float/java.lang.Float • double/java.lang.Double • java.lang.String	javax.validation.UnexpectedTypeException がスローされます。	−
6	Max	• java.math.BigDecimal • java.math.BigInteger • byte/java.lang.Byte • short/java.lang.Short • int/java.lang.Integer • long/java.lang.Long • float/java.lang.Float • double/java.lang.Double • java.lang.String	javax.validation.UnexpectedTypeException がスローされます。	−
7	DecimalMin	• java.math.BigDecimal • java.math.BigInteger • java.lang.String • byte/java.lang.Byte • short/java.lang.Short • int/java.lang.Integer • long/java.lang.Long • float/java.lang.Float • double/java.lang.Double	javax.validation.UnexpectedTypeException がスローされます。	value 属性に対して java.math.BigDecimal で解析できない値を指定した場合，javax.validation.ValidationException がスローされます。
8	DecimalMax	• java.math.BigDecimal • java.math.BigInteger • java.lang.String • byte/java.lang.Byte • short/java.lang.Short • int/java.lang.Integer • long/java.lang.Long • float/java.lang.Float • double/java.lang.Double	javax.validation.UnexpectedTypeException がスローされます。	value 属性に対して java.math.BigDecimal で解析できない値を指定した場合，javax.validation.ValidationException がスローされます。
9	Size	• java.lang.String • java.util.Collection • java.util.Map • 配列	javax.validation.UnexpectedTypeException がスローされます。	max 属性と min 属性に対して負の値を指定した場合，javax.validation.ValidationException がスローされます。

項番	アノテーションクラス	指定可能な変数の型	指定できない変数の型にアノテーションを指定した場合の動作	備考
9	Size	• java.lang.String • java.util.Collection • java.util.Map • 配列	javax.validation.UnexpectedTypeException がスローされます。	min の値に max より大きい値をした場合，javax.validation.ValidationException がスローされます。
10	Digits	• java.math.BigDecimal • java.math.BigInteger • java.lang.String • byte/java.lang.Byte • short/java.lang.Short • int/java.lang.Integer • long/java.lang.Long • float/java.lang.Float • double/java.lang.Double	javax.validation.UnexpectedTypeException がスローされます。	integer 属性と fraction 属性に対して負の値を指定した場合，javax.validation.ValidationException がスローされます。
11	Past	• java.util.Date • java.util.Calendar	javax.validation.UnexpectedTypeException がスローされます。	－
12	Future	• java.util.Date • java.util.Calendar	javax.validation.UnexpectedTypeException がスローされます。	－
13	Pattern	• java.lang.String	javax.validation.UnexpectedTypeException がスローされます。	regexp 属性に対して指定した値が正規表現として誤っていた場合，javax.validation.ValidationException 例外がスローされます。 正規表現の妥当性は java.util.regex.Pattern の仕様に依存します。

（凡例）－：該当しない。

10.5　Bean Validation の利用手順

この節では，Bean Validation の利用手順について説明します。

10.5.1　JSF から Bean Validation の利用手順

ここでは，JSF から Bean Validation を利用するための手順を示します。

(1)　使用前提条件

JSF で Bean Validation による検証処理を行うためには，次の条件を満たす必要があります。

- Web ページは Facelets で作成してください。
- ManagedBean の中で Bean Validation のアノテーションを利用して，制約を定義してください。

(2)　検証処理の動作

Bean Validation の検証処理の動作は，JSF が提供するコンテキストパラメタの値によって異なります。javax.faces.validator.DISABLE_DEFAULT_BEAN_VALIDATOR の設定値と Bean Validation のアノテーションを定義している ManagedBean に対する<f:validateBean>タグの有無によって，次の表に示すように動作が変わります。

表 10-4　javax.faces.validator.DISABLE_DEFAULT_BEAN_VALIDATOR の設定値と検証処理の関係

javax.faces.validator.DISABLE_DEFAULT_BEAN_VALIDATOR の設定値	入力値 <f:validateBean>タグの指定の有無（disabled 属性を指定していない場合）	Bean Validation による検証処理の有無
true	あり	あり
	なし	なし
false	あり	あり
	なし	あり
値を指定しなかった場合（デフォルト）	false の場合と同様	false の場合と同様

<f:validateBean>タグの使用方法の詳細に関しては，JSF の標準仕様を参照してください。ManagedBean の変数に対するバリデーション定義の方法は，Bean Validation の標準仕様を参照してください。

(3)　実装例

JSF から Bean Validation を使用する場合の実装例を示します。

まず，検証が必要な情報を登録する Facelets ページの実装例を示します。

```
<f:view>
  <h:form>
      IDの入力<br/>
      <h:inputText id="IDBox" value="#{personalData.id}" /><br/>
      <h:message for="IDBox"/><br/>
      <br/>
    <f:validateBean disabled="true">
      名前の入力（検証処理は行いません）<br/>
      <h:inputText id="NameBox" value="#{personalData.name}" /><br/>
    </f:validateBean >
      <br/>
```

```
        年齢の入力<br/>
        <h:inputText id="AgeBox" value="#{personalData.age}" /><br/>
        <h:message for="AgeBox"/><br/>
        <br/>
      <br/>
        :
        :
    </h:form>
  </f:view>
```

! 注意事項

<f:validateBean disabled="true">を指定している<inputText>タグは，対応する ManagedBean の変数にバリデーションの定義をしていますが，検証処理は行われません。

次に，検証対象のデータを格納する ManagedBean に対するバリデーション定義の実装例を示します。

```
@ManagedBean(name="personalData")
@SessionScoped
public class PersonalData
{
    @Size(min=8,max=12, message="8から12文字までの文字列を入力してください。")
    private String id = "";

    @Size(min=1,message="名前を入力してください。")
    private String name = "";

    @Max(value = 150, message="年齢の入力が正しいか確認して下さい。")
    @Min(value = 0, message="０歳以上の年齢を入力してください。")
    private int age = 0;
      :
setter/gettterメソッド
      :
}
```

10.5.2　CDI から Bean Validation の利用手順

ここでは，CDI から Bean Validation を利用するための手順を示します。

(1)　利用前に必要な手順

CDI で Bean Validation による検証処理を行うためには，次の手順が必要になります。

1. ユーザアプリケーションクラスで@Inject アノテーションを使って，ValidatorFactory を注入してください。

   例:@Inject private ValidatorFactory validatorFactory

2. Validator オブジェクトを得るために，Bean Validation からユーザアプリケーションの validatorFactory.getValidator()を呼びます。
3. 最後に，ユーザアプリケーションから validator.validate()メソッドを呼び，検証する Bean クラスを渡します。

(2)　実装例

CDI から Bean Validation を使用する場合の実装例を示します。

まず，検証が必要な情報を登録するサーブレットの実装例を示します。

```
public class EmployeeServBv extends HttpServlet{
@Inject  private ValidatorFactory validatorFactory;
@Inject BV_CDI bean;

   public void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException{
   Validator validator = validatorFactory.getValidator();
```

```
        validator.validate(bean);
    }
}
```

この例では，BV_CDI bean で Bean Validation のアノテーションを適用しています。

検証対象のデータを格納する Bean に対する検証定義の実装例を次に示します。

```
import javax.validation.constraints.NotNull;
public class BV_CDI{
        @NotNull
        private String name;
        public String getName(){
                return name;
        }
        public void setName(String name){
                this.name = name;
        }
}
```

10.6　デバッグ用ログ（開発調査ログ）の利用

この節では，Bean Validation のデバッグ用ログ（開発調査ログ）について説明します。

Bean Validation では，開発に関するメッセージが開発調査ログに出力されます。アプリケーションで利用している Bean Validation のデバッグをする場合に確認が必要なメッセージは，メッセージに出力されたクラス名から判断できます。メッセージを出力するクラス名について，次の表に示します。

表 10-5　Bean Validation の開発調査ログに出力されるクラス名

コンポーネント名	出力されるクラス名	出力内容
Bean Validation	com.hitachi.software.beanvalidation～	• 障害に関するメッセージ • 機能の動作状況に関するメッセージ • 内部状態に関するメッセージ

なお，開発調査ログはデフォルトの設定では出力されません。必要に応じて設定を変更してください。

開発調査ログを出力するための設定，および開発調査ログの出力先，出力形式，ログレベルなどについては，マニュアル「アプリケーションサーバ メッセージ(構築／運用／開発用)」の「24.4　開発調査ログに出力されるメッセージ」を参照してください。

10.7　validation.xml の内容が不正な場合の情報の出力

この節では，Bean Validation との連携で validation.xml の内容が不正な場合に出力されるメッセージについて説明します。

Bean Validation と連携するアプリケーション別に，出力されるメッセージ ID と内容，および出力先と対処方法について次の表に示します。

表 10-6　validation.xml の内容が不正な場合に出力されるメッセージ，出力先，および対処方法

Bean Validation と連携するアプリケーション	出力されるメッセージ ID，および内容	出力先，および対処方法
JSF アプリケーション	Web コンテナから KDJE39056-E が出力されます。そのメッセージには，リソース名と例外名が出力されています。	出力先： web_servlet ログ 対処方法： validation.xml の内容を確認し，問題点を修正してください。
CDI アプリケーション	KDJE59312-E が出力されます。そのメッセージには，例外の詳細が出力されています。	出力先： cjexception ログ 対処方法： validation.xml の内容を確認し，問題点を修正してください。
ユーザアプリケーション（Web アプリケーション）	Web コンテナから KDJE39045-E が出力されます。そのメッセージには，J2EE サーバモードの場合は，J2EE アプリケーション名とコンテキストルート名が出力されています。サーブレットエンジンモードの場合は，空文字列とコンテキストルート名が出力されています。	出力先： web_servlet ログ 対処方法： validation.xml の内容を確認し，問題点を修正してください。
ユーザアプリケーション（EJB アプリケーション）	－	出力先： cjexception ログ 対処方法： validation.xml の内容を確認し，問題点を修正してください。

（凡例）

－：該当しない。

10.8　Bean Validation 実装時の注意事項

この節では，Bean Validation 実装時の注意事項について説明します。

Bean Validation 1.0 仕様での注意事項

Bean Validation 1.0 仕様では，クラスパス上に配置できる validation.xml は一つだけです。J2EE アプリケーションのクラスパス上に複数の validation.xml を配置しないでください。

Message Interpolator のデフォルトのメッセージに関する注意事項

Message Interpolator のデフォルトのメッセージは，標準仕様に記載されている内容と異なります。アプリケーションサーバにおける Message Interpolator のメッセージのデフォルト値を，次の表に示します。

表 10-7　アプリケーションサーバにおける Message Interpolator のメッセージのデフォルト値

メッセージプロパティのキー	デフォルト値
javax.validation.constraints.AssertFalse.message	must be false
javax.validation.constraints.AssertTrue.message	must be true
javax.validation.constraints.Digits.message	numeric value out of bounds (<{integer} digits>.<{fraction} digits> expected)
javax.validation.constraints.Future.message	must be in the future
javax.validation.constraints.Max.message	must be less than or equal to {value}
javax.validation.constraints.Min.message	must be greater than or equal to {value}
javax.validation.constraints.NotNull.message	may not be null
javax.validation.constraints.Null.message	must be null
javax.validation.constraints.Past.message	must be in the past
javax.validation.constraints.Pattern.message	must match "{regexp}"
javax.validation.constraints.Size.message	size must be between {min} and {max}

11 アプリケーションの属性管理

この章では，アプリケーションの属性管理について説明します。アプリケーションを作成するに当たり，アプリケーションを構成する要素である，EJB-JAR 属性，WAR 属性，リソースなどを定義する必要があります。

アプリケーションサーバでは，それぞれの属性ファイルにアプリケーションを使用する場合に必要な情報を定義します。また，cosminexus.xml にアプリケーションサーバ独自の情報を定義してアプリケーションに含めることで，標準 DD の定義とアプリケーションサーバ独自の定義を別々に管理できます。アプリケーションを設定するに当たって必要な属性ファイル，および cosminexus.xml を使ったアプリケーションの運用などについて説明します。

11.1　この章の構成

アプリケーションの属性管理で説明する内容と参照先を次の表に示します。

表 11-1　アプリケーションの属性管理で説明する内容と参照先

機能	参照先
属性の管理	11.2
cosminexus.xml を含むアプリケーション	11.3
DD の省略	11.4

11.2　属性の管理

アプリケーションサーバでは，DD とアプリケーションサーバ独自の情報を定義するファイル（cosminexus.xml）を使ってアプリケーションを定義できます。

アプリケーションサーバでの DD のサポート範囲を次の表に示します。

表 11-2　アプリケーションサーバでの DD のサポート範囲

DD の種類	バージョン	サポート有無
application.xml	1.2	△※
	1.3	△※
	1.4	○
	5.0	○
	6.0	○
	DD なし（5.0）	○
	DD なし（6.0）	○
ejb-jar.xml	1.1	△
	2.0	○
	2.1	○
	3.0	○
	3.1	○
	DD なし（3.0）	○
	DD なし（3.1）	○
web.xml	2.2	△
	2.3	○
	2.4	○
	2.5	○
	3.0	○
	DD なし（2.5）	○
	DD なし（3.0）	○
ra.xml	1.0	○
	1.5	○

（凡例）

○：サポートする。

△：サポートするが，インポート時に DD のバージョンを書き換える。

注※

application.xml のバージョンが 1.2 または 1.3 の場合，<context-root>が EAR ファイル内で一意でないときにも，インポート時にバージョンとして 1.4 が設定されます。

! 注意事項

バージョン 3.0 の ejb-jar.xml に指定できる要素

EJB-JAR の<display-name>，インターセプタに関する要素，およびアプリケーション例外に関する要素が指定できます。これら以外の要素や属性を指定しても，無視されます。

具体的には，次の要素を指定できます。

- /ejb-jar/display-name
- /ejb-jar/assembly-descriptor/interceptor-binding，およびその配下の要素
- /ejb-jar/assembly-descriptor/application-exception，およびその配下の要素

ejb-name 要素に*を指定した interceptor-binding 要素が複数ある場合，いちばん上に記述された内容だけが使用されます。2 番目以降の内容は無視されます。

<ejb-name>，<named-method>，およびその配下の要素すべてが一致する<interceptor-binding>が複数存在する場合，いちばん上に記述された内容だけが使用されます。2 番目以降の内容は無視されます。

バージョン 3.1 の ejb-jar.xml に指定できる要素

バージョン 3.1 の ejb-jar.xml では，バージョン 3.0 の ejb-jar.xml で指定できる要素に加え，次の要素が指定できます。

- /ejb-jar/module-name

アプリケーションサーバでは，DD と cosminexus.xml を別々に管理して J2EE アプリケーションに含めることができます。cosminexus.xml を J2EE アプリケーションに含めることで，アプリケーションをインポートしたあとの属性ファイルの設定が不要になります。そのため，cosminexus.xml を含むアプリケーションは，インポートしたあとそのまま開始して使用できます。

また，アプリケーションサーバでは，アプリケーションの属性を定義する DD（application.xml，ejb-jar.xml および web.xml）を省略できます。

11.3 以降では，cosminexus.xml を含むアプリケーションの作成のしかたおよび運用のしかたについて説明します。また，11.4 以降では，DD の省略について説明します。

11.3　cosminexus.xml を含むアプリケーション

この節では，cosminexus.xml を含むアプリケーションの概要について説明します。

この節の構成を次の表に示します。

表 11-3　この節の構成（cosminexus.xml を含むアプリケーション）

分類	タイトル	参照先
解説	cosminexus.xml とは	11.3.1
	cosminexus.xml を含むアプリケーションを使用する利点	11.3.2
	cosminexus.xml を含むアプリケーションの作成	11.3.3
実装	cosminexus.xml の作成	11.3.4
	cosminexus.xml の作成例	11.3.5
運用	cosminexus.xml を含むアプリケーションの運用	11.3.6
	cosminexus.xml を含まないアプリケーションから移行する手順	11.3.7

注　「設定」，および「注意事項」について，この機能固有の説明はありません。

11.3.1　cosminexus.xml とは

cosminexus.xml は，アプリケーションに関するアプリケーションサーバ独自の定義情報を記載する属性ファイルです。cosminexus.xml は，アプリケーションごとに作成できます。

cosminexus.xml をアーカイブ形式のアプリケーションに含めてインポートまたはリデプロイしたり，展開ディレクトリ形式のアプリケーションに含めてアプリケーションを開始したりすることで，アプリケーションサーバ独自の定義情報を設定した状態でアプリケーションを実行できます。

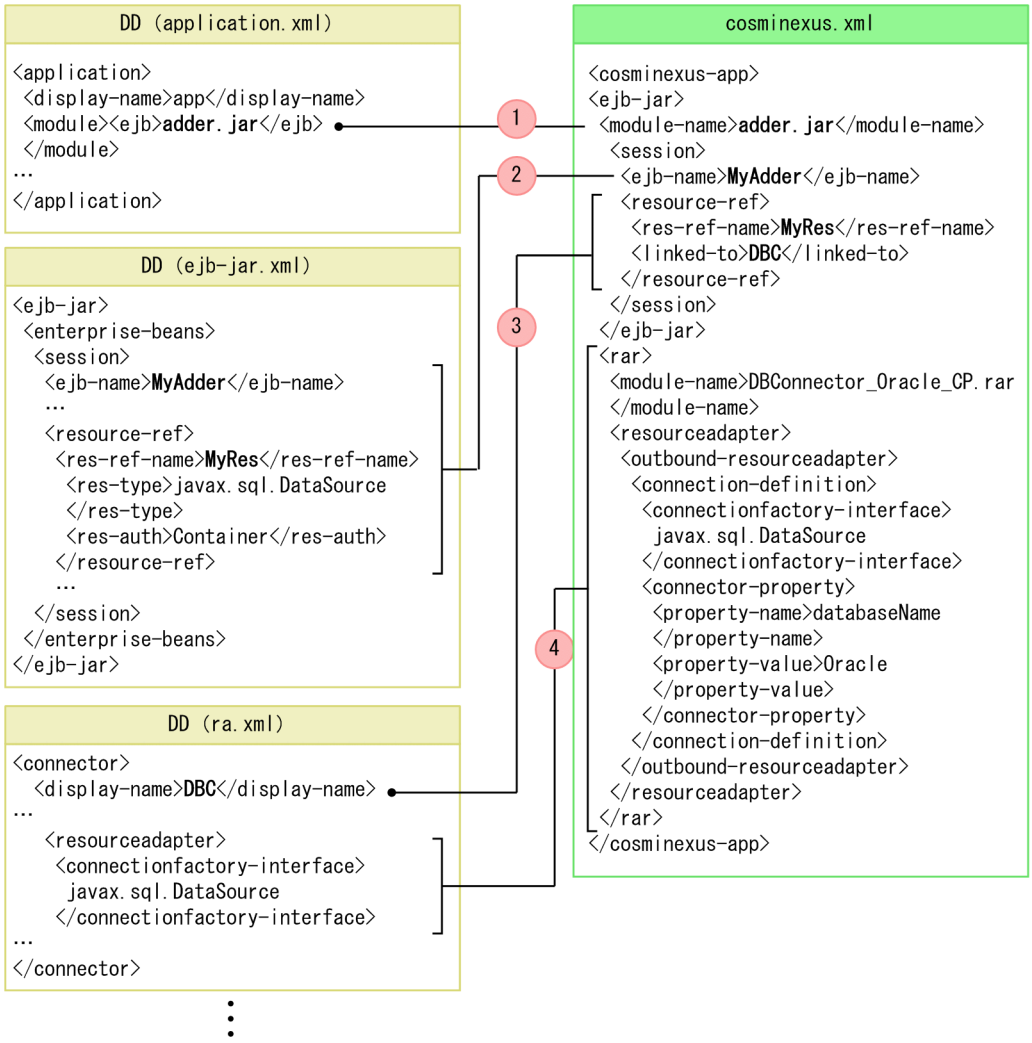
cosminexus.xml を含むアプリケーションの場合，DD の情報とアプリケーションサーバ独自の情報が別々に管理されます。そのため，アプリケーションサーバ独自の情報を編集したときに，DD の情報を編集する必要がありません。cosminexus.xml を含むアプリケーションでは，J2EE サーバ上でのコマンドによるアプリケーションの属性の設定が不要です。

cosminexus.xml を含むアプリケーションの概念を図に示します。

図 11-1　cosminexus.xml を含むアプリケーションの概念図

cosminexus.xml と DD の定義を関連づける例を次の図に示します。

図 11-2　cosminexus.xml と DD の定義を関連づける例



図について説明します。この例では，cosminexus.xml によって，EJB-JAR からのリソースの参照を解決して，リソースのプロパティを設定します。なお，説明の番号は図中の番号と対応しています。

1. <module-name>で対象とする EJB-JAR を特定します。ここでは，adder.jar が対象になります。
2. adder.jar の DD（ejb-jar.xml）と cosminexus.xml を関連づけます。ここでは，<ejb-name>で対象となる Session Bean（MyAdder）を特定します。
3. Session Bean（MyAdder）で参照しているリソース（MyRef）のリソース参照を解決します。<res-ref-name>に指定した名称に対する参照先として，リソースの表示名（DBC）を<linked-to>に指定します。
4. リソース（DBC）に対するプロパティを設定します。なお，cosminexus.xml で設定できるのは，J2EE アプリケーションに含まれるリソースアダプタのプロパティです。

この cosminexus.xml を J2EE アプリケーションに含めてインポートすることで，J2EE サーバ上でのリソース参照解決やリソースアダプタのプロパティ設定が不要になります。

11.3.2　cosminexus.xml を含むアプリケーションを使用する利点

cosminexus.xml を含むアプリケーションを使用する利点を次に示します。

(1)　定義情報を最小限にできる

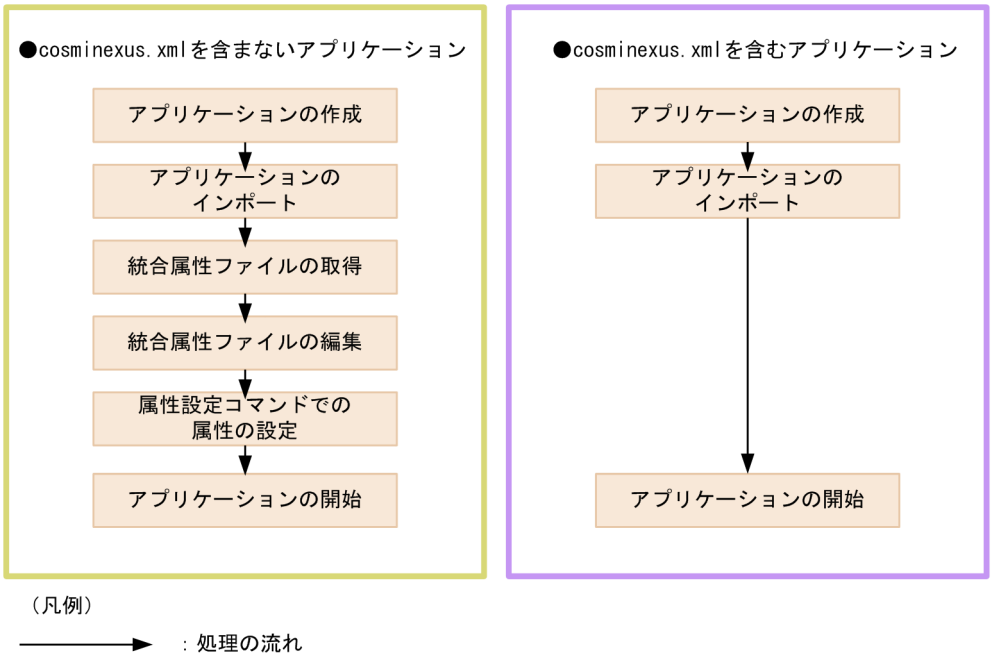
cosminexus.xml を含むアプリケーションには，アプリケーションサーバ独自の定義情報のうち，ユーザがカスタマイズしたい情報だけを記述します。省略した場合はデフォルト値が使用されます。

(2)　アプリケーションのインポートから開始までの手順を簡略化できる

アプリケーションをインポートしたあと，アプリケーションの属性設定コマンド（cjgetappprop コマンドおよび cjsetappprop コマンド）による操作をしないでアプリケーションを実行できます。

cosminexus.xml を含むアプリケーションと cosminexus.xml を含まないアプリケーションでのアプリケーションのインポートから開始までの流れについて図で示します。

図 11-3　cosminexus.xml を含むアプリケーションと cosminexus.xml を含まないアプリケーションのインポートから開始までの流れ



(3)　DD を変更した際の属性ファイルの変更が不要

cosminexus.xml を含まないアプリケーションの場合，標準 DD を変更してアプリケーションを入れ替えた場合，サーバ管理コマンドを使ってアプリケーションサーバ独自の定義を設定し直す必要があります。しかし，cosminexus.xml を含むアプリケーションの場合，アプリケーションサーバ独自の定義は DD と別のファイルで管理されているため，実行環境でサーバ管理コマンドを使って設定し直す必要がありません。

ただし，次に示す場合には，cosminexus.xml を変更する必要があります。

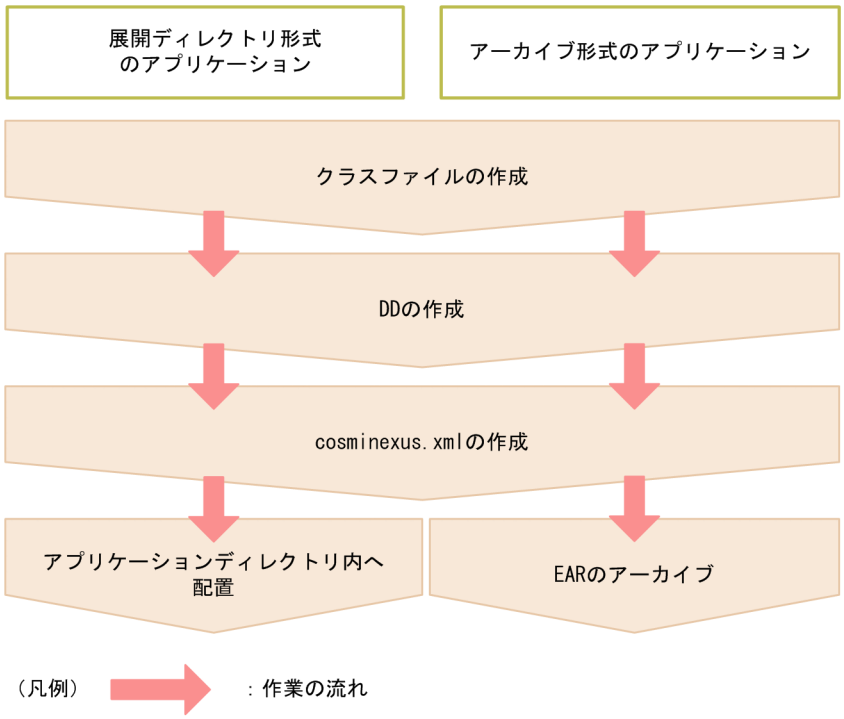
- アプリケーションサーバ独自の定義情報と標準 DD およびアノテーションの定義情報を関連づける情報（J2EE リソースのモジュール名など）を変更した場合
- アプリケーションを開始するためにリンク解決が必要な定義情報（リソースなど）を追加した場合

cosminexus.xml に記述した定義情報と，標準 DD およびアノテーションの定義情報を関連づける定義情報およびリンク解決が必要な定義情報については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「2.　アプリケーション属性ファイル（cosminexus.xml）」を参照してください。

11.3.3　cosminexus.xml を含むアプリケーションの作成

ここでは，cosminexus.xml を含むアプリケーションの作成について説明します。cosminexus.xml を含むアプリケーションを作成するには，あらかじめクラスファイル，DD，および cosminexus.xml を作成する必要があります。cosminexus.xml を含むアプリケーションを作成する流れを図で示します。

図 11-4　cosminexus.xml を含むアプリケーションを作成する流れ



なお，WAR ファイルまたは WAR ディレクトリを指定してインポートした WAR アプリケーションの場合，図 11-4 は次のようになります。WAR アプリケーションの詳細については，「13.9　WAR アプリケーション」を参照してください。

展開ディレクトリ形式の場合

　アプリケーションは WAR ディレクトリ内に配置します。

アーカイブ形式の場合

　アプリケーションは WAR のアーカイブを作成します。

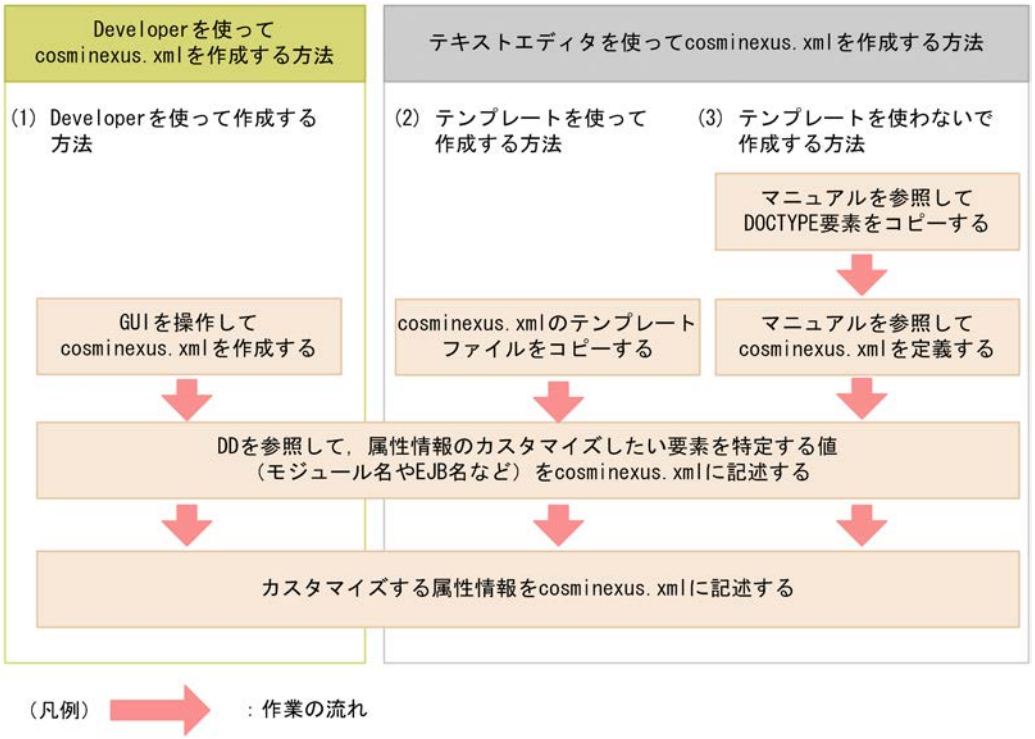
クラスファイルの作成，DD の作成，および cosminexus.xml を含むアプリケーションの作成についてはマニュアル「アプリケーションサーバ アプリケーション開発ガイド」を参照してください。

11.3.4 以降では，cosminexus.xml の作成について説明します。

11.3.4　cosminexus.xml の作成

cosminexus.xml を含むアプリケーションを作成するには，あらかじめ作成した cosminexus.xml をアプリケーションに含める必要があります。cosminexus.xml を作成する方法を次に示します。

図 11-5　cosminexus.xml を作成する方法



cosminexus.xml を作成する方法には，Developer の WTP を使って作成する方法とテキストエディタを使って作成する方法があります。それぞれの方法を，図中の（1）～（3）に沿って説明します。

(1)　Developer を使って作成する方法

アプリケーションサーバでは，Developer の WTP を使って cosminexus.xml を作成できます。Developer の WTP を使って cosminexus.xml を作成する方法については，マニュアル「アプリケーションサーバ アプリケーション開発ガイド」の「5.3.1　cosminexus.xml の作成」を参照してください。

(2)　テンプレートを使って作成する方法

テキストエディタを使って cosminexus.xml を作成する場合，テンプレートファイルを使用すると容易に作成できます。

テンプレートの格納場所を次に示します。

テンプレートの格納場所

- Windows の場合
  <Application Serverのインストール先>¥CC¥admin¥templates¥cosminexus.xml
- UNIX の場合
  /opt/Cosminexus/CC/admin/templates/cosminexus.xml

テンプレートをコピーしてアーカイブファイルまたは展開先ディレクトリ内に格納します。

cosminexus.xml の格納先を次に示します。

EAR ファイルをインポートする場合

- EAR 内の格納先（アーカイブ形式のアプリケーションの場合）

<EARのルート>/META-INF/cosminexus.xml

- アプリケーションディレクトリ内の格納先（展開ディレクトリ形式のアプリケーション場合）

  展開ディレクトリ形式のアプリケーションでの cosminexus.xml の格納先については,「13.4.2　アプリケーションディレクトリの構成」を参照してください。

WAR ファイルをインポートする場合

cosminexus.xml の格納先は，cjimportwar コマンドで指定します。cjimportwar コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjimportwar（WAR アプリケーションのインポート)」を参照してください。

! 注意事項

テンプレートには，DOCTYPE 要素に指定するスキーマ定義ファイルの格納先として，アプリケーションサーバのインストール先の書き換えを前提とした URI「file:///<アプリケーションサーバのインストール先>/CC/admin/dtds/cosminexus_8_0.dtd」が記載されています。J2EE サーバはこの URI を参照しないため，書き換える必要はありません。ただし，XML エディタなどで使用する場合は，使用する環境に合わせてこの URI を書き換えてください。

(3)　テンプレートを使わないで作成する方法

テンプレートを使わないで cosminexus.xml を作成する場合は，マニュアルを基に，テキストエディタを使ってアプリケーションサーバ独自の情報を定義します。テキストエディタを使って cosminexus.xml の作成する場合に必要な DOCTYPE 要素，および cosminexus.xml の定義内容については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「2.　アプリケーション属性ファイル（cosminexus.xml)」を参照してください。

11.3.5　cosminexus.xml の作成例

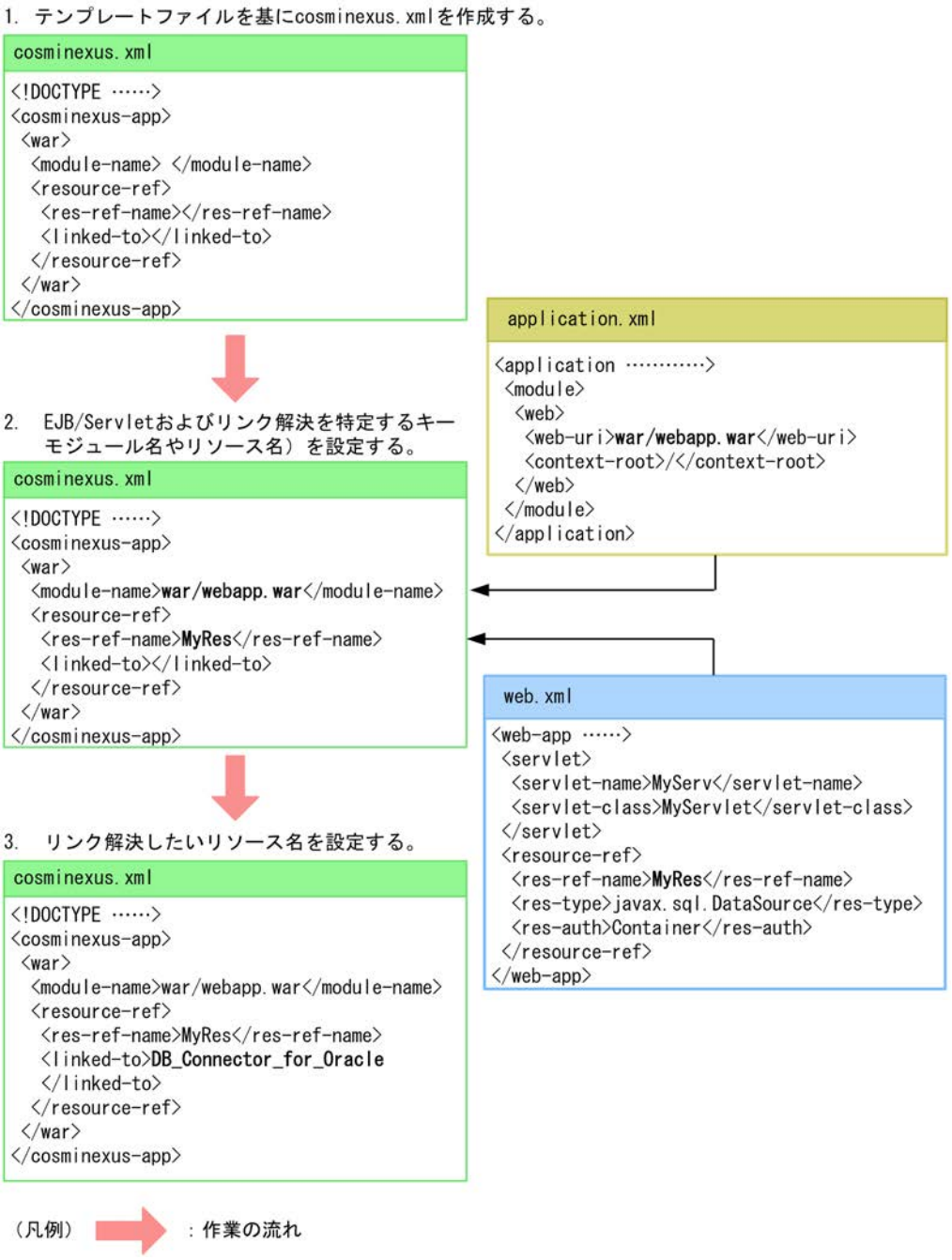
テキストエディタを使った cosminexus.xml の作成例を示します。ここでは，次の DD の定義内容を基に cosminexus.xml を作成します。

標準 DD の定義内容

- Web アプリケーションのモジュール名：war/webapp.war
- Servlet 名：MyServ
- Servlet（MyServ）から参照するリソース名：DB_Connector_for_Oracle

cosminexus.xml の作成例を図で説明します。

図 11-6　cosminexus.xml の作成例



図中の 1.～3.について説明します。

1. テンプレートファイルを基に，cosminexus.xml を作成する。

   テンプレートファイルをコピーして EAR またはアプリケーションディレクトリ内に格納します。コピーしたテンプレートファイルは，テキストエディタを使って編集します。

2. EJB／Servlet およびリンク解決を特定するキー（モジュール名やリソース名）を設定する。

   DD で定義した情報と関連づけるキーを設定します。ここでは，Web アプリケーションのモジュール名，および参照しているリソース名を設定しています。

3. リンク解決したいリソース名を設定する。

2.で設定したキーのリンク解決をするリソースを設定します。

なお，cosminexus.xml の定義内容の詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「2.1　アプリケーション属性ファイル（cosminexus.xml）の指定内容」を参照してください。

11.3.6　cosminexus.xml を含むアプリケーションの運用

ここでは，cosminexus.xml を含むアプリケーションの運用について説明します。

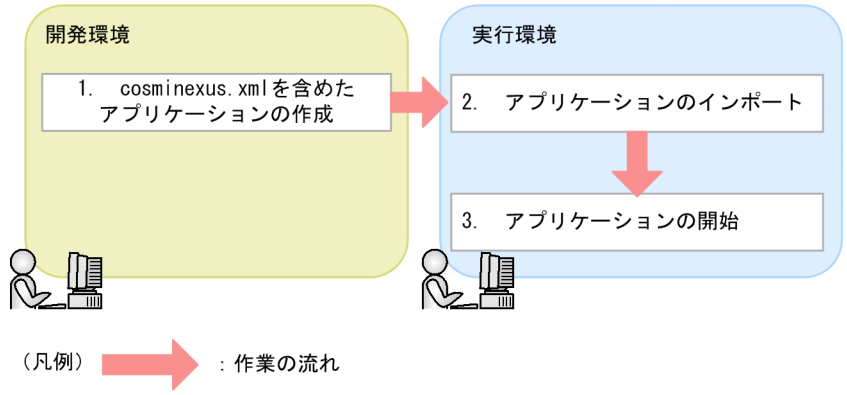
(1)　cosminexus.xml を含むアプリケーションを新規作成する流れ

cosminexus.xml を含むアプリケーションを新規で作成する流れについて説明します。

cosminexus.xml を含むアプリケーションは，開発環境で作成します。作成したアプリケーションを実行環境へインポートし，アプリケーションを開始します。

開発環境での cosminexus.xml を含むアプリケーションの作成から，実行環境でのアプリケーションの開始までの流れを図で説明します。

図 11-7　cosminexus.xml を含むアプリケーションの作成から実行環境でのアプリケーションの開始までの流れ



図中の 1.～3.について説明します。

1. cosminexus.xml を含めたアプリケーションの作成

   クラスファイル，標準 DD および cosminexus.xml を作成し，アプリケーションの形式に合わせてアーカイブしたりアプリケーションディレクトリに配置したりします。クラスファイル，DD，および cosminexus.xml を含むアプリケーションの作成については，マニュアル「アプリケーションサーバ アプリケーション開発ガイド」を参照してください。また，テキストエディタを使った cosminexus.xml の作成については，「11.3.4　cosminexus.xml の作成」を参照してください。

2. アプリケーションのインポート

   開発環境で作成したアプリケーションを実行環境へインポートします。アプリケーションのインポートについては，「(2)　アプリケーションのインポート」を参照してください。

3. アプリケーションの開始

   インポートしたアプリケーションを開始します。アプリケーションの開始方法については，「(3)　アプリケーションの開始」を参照してください。

(2)　アプリケーションのインポート

cosminexus.xml を含むアプリケーションのインポートについて説明します。

cosminexus.xml を含むアプリケーションの場合，インポートするタイミングで cosminexus.xml の情報が読み込まれます。J2EE アプリケーションをインポートするには，cjimportapp コマンドを実行します。cjimportapp コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjimportapp（J2EE アプリケーションのインポート）」を参照してください。J2EE アプリケーションのインポートに関する注意事項については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「8.1　J2EE アプリケーションのインポート」を参照してください。

WAR アプリケーションをインポートするには cjimportwar コマンドを実行します。cjimportwar コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjimportwar（WAR アプリケーションのインポート）」を参照してください。WAR アプリケーションについては，「13.9　WAR アプリケーション」を参照してください。

(3)　アプリケーションの開始

cosminexus.xml を含むアプリケーションの開始について説明します。

cosminexus.xml を含むアプリケーションを開始する手順は，cosminexus.xml を含まないアプリケーションを開始する手順と変わりません。アプリケーションを開始するには cjstartapp コマンドを実行します。cjstartapp コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjstartapp（J2EE アプリケーションの開始）」を参照してください。また，J2EE アプリケーションの開始に関する注意事項については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「10.2.1　J2EE アプリケーションの開始」を参照してください。

cosminexus.xml を含むアプリケーションを実行環境で開始したあとで一度アプリケーションを停止し，再度，アプリケーションを開始する場合のアプリケーションの動作について説明します。

ここでは，アプリケーションを再開始した際のアプリケーションの形式が，アーカイブ形式の場合，展開ディレクトリ形式の場合，cosminexus.xml を含まないアプリケーションの場合に分けて説明します。

- アーカイブ形式のアプリケーションの場合

  アーカイブ形式のアプリケーションに cosminexus.xml が含まれている場合，アプリケーションに含まれている cosminexus.xml の情報が読み込まれます。アーカイブ形式のアプリケーションには，停止する前のアプリケーションに対するサーバ管理コマンドを使った定義情報の変更は反映されません。

- 展開ディレクトリ形式の場合

  展開ディレクトリ形式のアプリケーションに cosminexus.xml が含まれている場合，アプリケーションを開始するときに，実行環境で変更したアプリケーションサーバ独自のアプリケーションに関する情報を上書きします。つまり，展開ディレクトリ形式のアプリケーションの場合には，サーバ管理コマンドを使った定義情報の変更がファイルに反映されます。

- アプリケーションに cosminexus.xml が含まれていない場合

  アプリケーションに cosminexus.xml が含まれていない場合は，前回アプリケーションを開始した際のアプリケーションサーバ独自の定義内容でアプリケーションを開始します。

(4)　アプリケーションの開始後に定義情報を変更する流れ

開発環境からインポートしたアプリケーションを実行環境で開始したあとで，アプリケーションの定義情報を変更したい場合の手順について説明します。

cosminexus.xml を含むアプリケーションの定義を変更する場合，アプリケーションの形式がアーカイブ形式か，または展開ディレクトリ形式かによって手順が異なります。

アプリケーションの形式ごとに，定義内容の変更手順について説明します。

参考

cosminexus.xml を含まないアプリケーションと同じように，サーバ管理コマンド（cjgetappprop コマンドおよび cjsetappprop コマンド）を使ったアプリケーションの定義内容の変更もできます。

！ 注意事項

cosminexus.xml を含むアプリケーションの定義情報を変更する場合の注意事項を示します。

- アプリケーションサーバ独自の定義情報をすべてデフォルト値に戻したい場合は，cosminexus.xml のファイルから，<cosminexus-app>要素以外すべての要素を削除してください。
- サーバ管理コマンドを使ってアプリケーションの定義を変更した場合の注意事項を，アプリケーションの形式ごとに説明します。

アーカイブ形式の場合

リデプロイ機能による定義情報の変更では，アプリケーションサーバのアプリケーションの実行時情報をいったんデフォルトに戻したあとで，入れ替え後の cosminexus.xml に定義されている情報を読み込みます。そのため，サーバ管理コマンドで設定した定義情報はすべて失われます。

！ 注意事項

WAR アプリケーションの場合はリデプロイ機能で定義情報の変更はできません。サーバ管理コマンド（cjgetappprop コマンドおよび cjsetappprop コマンド）を使用してください。

展開ディレクトリ形式の場合

サーバ管理コマンドを使用して変更したアプリケーションの属性の内容は，そのままアプリケーションディレクトリ内の cosminexus.xml に反映されます。

なお，リロード機能では，cosminexus.xml の定義情報は読み込まれません。変更した定義情報を反映したい場合は，アプリケーションの再開始が必要です。アプリケーション統合属性ファイルとサーバ管理コマンドを使用して定義情報を変更する場合については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「9. J2EE アプリケーションのプロパティ設定」を参照してください。

(a) アーカイブ形式の場合

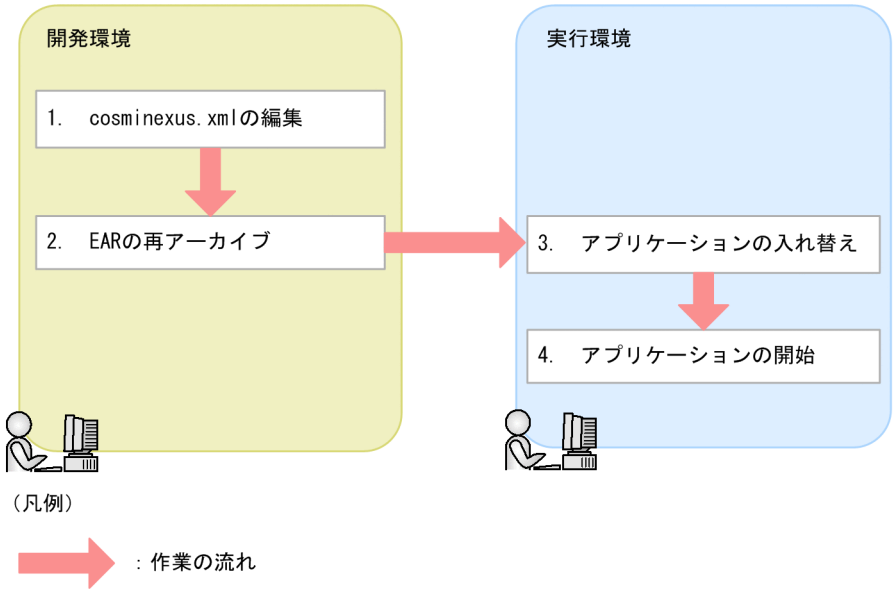
cosminexus.xml を含むアプリケーションがアーカイブ形式の場合の変更手順について説明します。なお，cosminexus.xml を含むアプリケーションの定義情報は，ここで示した手順だけでなく，サーバ管理コマンドと属性ファイルを使う手順でも実行できます。

cosminexus.xml を含むアプリケーションを実行環境で開始したあとで，定義内容を変更したい場合は，開発環境でアプリケーションの定義内容を変更します。開発環境で定義内容を変更してアーカイブし直したあと，再度アプリケーションを実行環境へインポートし，アプリケーションを入れ替えます。

また，稼働中の J2EE アプリケーションを実行時情報付き ZIP 形式にしてエクスポートできます。実行時情報付き ZIP 形式のアプリケーションの操作手順は，cosminexus.xml を含むアプリケーションと cosminexus.xml を含まないアプリケーションとで差異はありません。アプリケーションのエクスポートについては「(6)　アプリケーションのエクスポート」を参照してください。

cosminexus.xml を含むアプリケーションの定義内容の変更手順について図で説明します。

図 11-8　cosminexus.xml を含むアプリケーションの定義内容の変更手順（アーカイブ形式の場合）



図中の 1.～4.について説明します。

1. cosminexus.xml の編集

   cosminexus.xml を含むアプリケーションを実行環境へインポートしたあとで，定義情報を変更する場合，開発環境で cosminexus.xml を編集します。cosminexus.xml の編集については，マニュアル「アプリケーションサーバ アプリケーション開発ガイド」の「5.3.2　cosminexus.xml エディタの操作方法」を参照してください。

2. EAR の再アーカイブ

   編集した cosminexus.xml をアプリケーションに含めてアーカイブします。

3. アプリケーションの入れ替え

   開発環境で作成し直したアプリケーションを，実行環境で使用しているアプリケーションと入れ替えます。アプリケーションの入れ替えについては，「(5)　アプリケーションの入れ替え」を参照してください。

4. アプリケーションの開始

   アプリケーションを開始します。アプリケーションの開始方法については，「(3)　アプリケーションの開始」を参照してください。

ポイント

サーバ管理コマンドを使って cosminexus.xml の定義情報を変更した場合，アプリケーションを入れ替えたときに入れ替え後の J2EE アプリケーションに含まれる cosminexus.xml の定義情報が上書きされます。cosminexus.xml の定義情報を変更する場合は，開発環境で cosminexus.xml を編集し，アプリケーションをリデプロイまたは再度インポートし直してください。

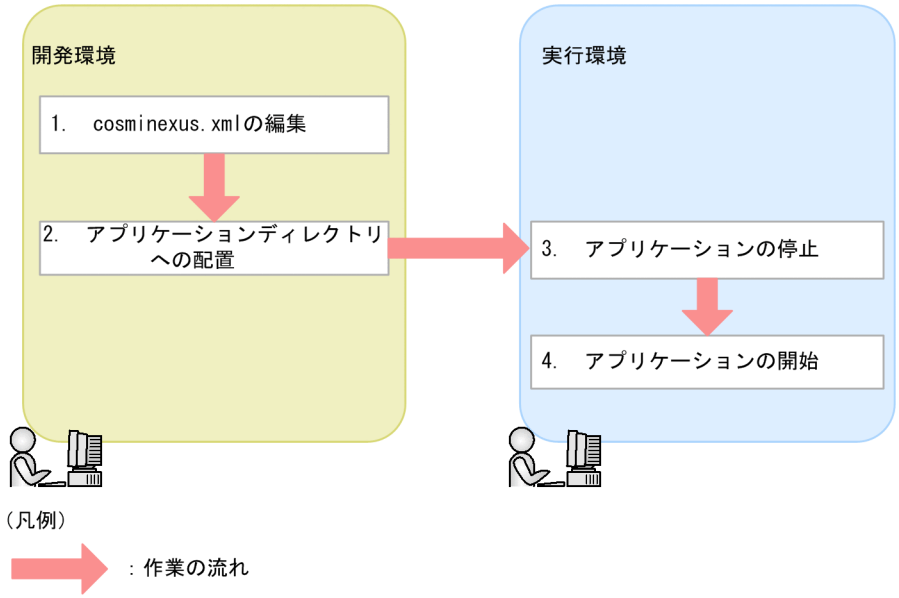
サーバ管理コマンドを使用して定義情報を変更する方法については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「9.　J2EE アプリケーションのプロパティ設定」を参照してください。

(b)　展開ディレクトリ形式の場合

cosminexus.xml を含むアプリケーションが展開ディレクトリ形式の場合の変更手順について説明します。

cosminexus.xml を含むアプリケーションの定義内容の変更手順について図で説明します。

図 11-9　cosminexus.xml を含むアプリケーションの定義内容の変更手順（展開ディレクトリ形式）



図中の 1.～4.について説明します。

1. cosminexus.xml の編集

   cosminexus.xml を含むアプリケーションを実行環境へインポートしたあとで，定義情報を変更する場合，開発環境で cosminexus.xml を編集します。cosminexus.xml の編集については，マニュアル「アプリケーションサーバ アプリケーション開発ガイド」の「5.3.2　cosminexus.xml エディタの操作方法」を参照してください。

2. アプリケーションディレクトリへの配置

   編集した cosminexus.xml をアプリケーションディレクトリへ配置します。アプリケーションディレクトリへ配置については，マニュアル「アプリケーションサーバ アプリケーション開発ガイド」の「1.4.1　展開ディレクトリ形式の J2EE アプリケーション」を参照してください。

3. アプリケーションの停止

   実行環境で使っているアプリケーションを停止します。アプリケーションの停止については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「10.2.2　J2EE アプリケーションの停止」を参照してください。

4. アプリケーションの開始

   インポートしたアプリケーションを開始します。アプリケーションの開始については「(3)　アプリケーションの開始」を参照してください。

(5)　アプリケーションの入れ替え

cosminexus.xml を含むアプリケーションの入れ替えについて説明します。

アーカイブ形式のアプリケーションを入れ替えるには，リデプロイ機能を使用します。リデプロイ機能を使ってアプリケーションを入れ替える場合，入れ替え後のアプリケーションに含まれる cosminexus.xml の情報が再度読み込まれます。開発環境で cosminexus.xml を変更し，EAR ファイルにアーカイブし直してリデプロイを実行することで，アプリケーションサーバ独自の定義情報が変更されます。

! 注意事項

WAR アプリケーションの場合，アプリケーションの入れ替えによる定義情報の変更はできません。サーバ管理コマンド（cjgetappprop コマンドおよび cjsetappprop コマンド）を使用してください。

なお，入れ替えるアプリケーションに cosminexus.xml が含まれていない場合，入れ替え前の cosminexus.xml の定義情報を引き継ぎます。※

リデプロイ機能を使用するには，cjreplaceapp コマンドを使用します。cjreplaceapp コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjreplaceapp（アプリケーションの入れ替え）」を参照してください。また，リデプロイ機能に関する注意事項については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「10.5.1　アーカイブ形式のアプリケーション」を参照してください。

注※　入れ替えるアプリケーションの cosminexus.xml の有無にかかわらず，DD の入れ替えはできません。

! 注意事項

アーカイブ形式のアプリケーションを入れ替える場合の注意事項を示します。

- cosminexus.xml を含むアプリケーションに対して，サーバ管理コマンドでアプリケーションサーバ独自の定義情報を変更したあとでリデプロイ機能を使用すると，サーバ管理コマンドで変更したアプリケーションサーバ独自の情報は失われます。
- アプリケーションサーバ独自の情報をすべてデフォルト値に戻したい場合は，cosminexus.xml から <cosminexus-app>以外の要素をすべて削除してから，cosminexus.xml を含むアプリケーションを入れ替えてください。

---

(6)　アプリケーションのエクスポート

cosminexus.xml を含むアプリケーションのエクスポートについて説明します。

アプリケーションをエクスポートするには，cjexportapp コマンドを使用します。cjexportapp コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjexportapp (J2EE アプリケーションのエクスポート)」を参照してください。また，アプリケーションのエクスポートに関する注意事項については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「8.2　J2EE アプリケーションのエクスポート」を参照してください。

! 注意事項

WAR アプリケーションの場合，アプリケーションのエクスポートはできません。

---

cosminexus.xml を含むアプリケーションをアーカイブ形式でエクスポートした場合，エクスポートされた EAR ファイルには cosminexus.xml が含まれます。エクスポートの対象となるアプリケーションおよびエクスポートされた EAR ファイルの組み合わせを次に示します。

表 11-4　エクスポートの対象となるアプリケーションおよびエクスポートされた EAR ファイルの組み合わせ

エクスポートの対象となるアプリケーションの形式	エクスポートされた EAR ファイルの形式	
	cjexportapp コマンドに-normal オプションを指定した場合	cjexportapp コマンドに-raw オプションを指定した場合
cosminexus.xml を含むアプリケーション	cosminexus.xml を含む実行時情報付き ZIP 形式	cosminexus.xml を含む EAR

エクスポートの対象となるアプリケーションの形式	エクスポートされた EAR ファイルの形式	
	cjexportapp コマンドに-normal オプションを指定した場合	cjexportapp コマンドに-raw オプションを指定した場合
cosminexus.xml を含まないアプリケーション	cosminexus.xml を含まない実行時情報付き ZIP 形式	cosminexus.xml を含まない EAR

! 注意事項

エクスポートしたアプリケーションに含まれる cosminexus.xml に関する注意事項を示します。

- コメントは保持されません。
- 文字エンコードは UTF-8 になります。
- DOCTYPE に記載された DTD のバージョンが古いバージョンの cosminexus.xml を含む J2EE アプリケーションをエクスポートした場合, DTD のバージョンはアプリケーションサーバでサポートされる最新のバージョンに変更されます。
- エクスポートした EAR/ZIP に含まれるのは J2EE サーバが最後に読み込みに成功した cosminexus.xml です。J2EE サーバが cosminexus.xml を読み込んだあとにアプリケーションディレクトリ内の cosminexus.xml を変更しても，変更後の定義情報はエクスポートした EAR/ZIP に含まれません。例えば，展開ディレクトリ形式のアプリケーションをインポートしたあと，アプリケーションディレクトリ内の cosminexus.xml を書き換え，アプリケーションを開始しないでアプリケーションを EAR 形式でエクスポートした場合，EAR にはインポートした時と同様の cosminexus.xml が含まれます。インポート後に更新したアプリケーションディレクトリ内の cosminexus.xml は含まれません。

(7)　アプリケーションからの J2EE リソースの削除

アプリケーションから J2EE リソースを削除する場合，cosminexus.xml に削除する J2EE リソースの情報が含まれていても，その情報は削除されません。

アプリケーション内にない要素をカスタマイズする定義情報が cosminexus.xml にある場合，cosminexus.xml を読み込むサーバ管理コマンドの実行時に警告メッセージが出力されます。

11.3.7　cosminexus.xml を含まないアプリケーションから移行する手順

ここでは，cosminexus.xml を含まないアプリケーションから cosminexus.xml を含むアプリケーションに移行する手順について説明します。

次の手順でアプリケーションを移行してください。

1. 移行前のアプリケーションからアプリケーション統合属性ファイルを取得します。
   取得にはサーバ管理コマンド（cjgetappprop）を使用します。
2. cosminexus.xml を作成します。
   cosminexus.xml とアプリケーション統合属性ファイルの対応については, マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「2.2.1　アプリケーション統合属性の詳細」を参照してください。
3. EAR（アーカイブ形式の場合）またはアプリケーションディレクトリ（展開ディレクトリ形式の場合）に手順 2.で作成した cosminexus.xml を含めます。
4. アプリケーションを入れ替えます。
5. 正しくアプリケーションについての定義情報が移行できているかを検証します。

入れ替え後のアプリケーションからアプリケーション統合属性ファイルを取得して，手順 1.で取得したファイルと比較することで検証できます。

11.4　DD の省略

アプリケーションサーバでは，application.xml，ejb-jar.xml および web.xml を省略できます。ただし，モジュールや DD のバージョンの組み合わせによっては省略できないものもあります。

この節の構成を次の表に示します。

表 11-5　この節の構成（cosminexus.xml を含むアプリケーション）

分類	タイトル	参照先
解説	アプリケーションサーバで実行できるアプリケーションの構成	11.4.1
	application.xml の有無による機能の違い	11.4.2
	application.xml がある場合のモジュールの決定規則	11.4.3
	application.xml がない場合のモジュールの決定規則	11.4.4
	web.xml を省略した Web アプリケーションに対する操作	11.4.5
	DD を省略した場合に設定される表示名	11.4.6
	application.xml を省略した場合に application.xml が作成される操作	11.4.7
注意事項	application.xml を省略した場合に J2EE アプリケーションにリソースを追加するときの注意	11.4.8

注　「設定」，「実装」および「運用」について，この機能固有の説明はありません。

11.4.1　アプリケーションサーバで実行できるアプリケーションの構成

アプリケーションサーバでは，次の DD を省略できます。

- Java EE 5，Java EE 6 の application.xml
- EJB 3.0，EJB 3.1 の ejb-jar.xml
- Servlet 2.5，Servlet 3.0 の web.xml

application.xml の DD のバージョンと DD の省略，および各モジュールの DD（ejb-jar.xml，web.xml，ra.xml）のバージョンと DD の省略の組み合わせを表で示します。

表 11-6　DD の組み合わせ

DD の種別		application.xml			
		1.4 以前※1	Java EE 5	Java EE 6	DD なし
ejb-jar.xml	2.0 以前※1	○	○	○	○
	2.1	○	○	○	○
	3.0	×	○	○	○
	3.1	×	×	○	○

DD の種別		application.xml			
		1.4 以前※1	Java EE 5	Java EE 6	DD なし
ejb-jar.xml	DD なし（3.0 以降）	○（3.0）	○（3.0）	○（3.1）	○（3.1）※2
web.xml	2.3 以前※1	○	○	○	○
	2.4	○	○	○	○
	2.5	×	○	○	○
	3.0	×	×	○	○
	DD なし（2.5 以降）	×	○（2.5）	○（3.0）	○（3.0）※2
ra.xml	1.0	○	○	○	○
	1.5	○	○	○	○

（凡例）○：サポートする　×：サポートしない

注※1　バージョンアップ，またはバージョンアップしたあとのサーバ管理コマンド実行時に，次に示すバージョンに変更になります。

注※2　08-70 以前のアプリケーションサーバでインポート済みの J2EE アプリケーションは，09-00 以降にアップグレードしても，ejb-jar.xml のバージョンは 3.0，web.xml のバージョンは 2.5 として動作します。

表 11-7　DD のバージョンアップ

DD	バージョンアップ前のバージョン	バージョンアップ後のバージョン
application.xml	1.2	1.4
	1.3	
ejb-jar.xml	1.1	2.0
web.xml	2.2	2.3

参考

次のコマンドを実行したとき，J2EE アプリケーションがサポートしていない構成の場合はメッセージ KDJE42361-E を出力し，エラーとなります。

- cjimportapp
- cjreplaceapp
- cjaddapp

11.4.2　application.xml の有無による機能の違い

application.xml を省略した場合，コマンドを実行したときのアセンブル機能およびデプロイ機能に違いが出ます。J2EE アプリケーションに application.xml がある場合，および application.xml がない場合に分けて，サーバ管理コマンドの機能の違いを説明します。

表 11-8　application.xml の有無による機能の違い一覧

J2EE アプリケーションを操作するコマンド	application.xml あり	application.xml なし
cjimportapp	**モジュールの決定** application.xml によってモジュールを決定します。 **J2EE アプリケーションの表示名** <display-name>タグによって決定します。<display-name>タグがなければ EAR ファイル名によって決定します。 **EJB-JAR ディレクトリ，WAR ディレクトリの決定** 展開ディレクトリ形式（-a オプション指定）の場合，application.xml の<module>タグに記述されたパス名から拡張子を除いた名称がとなります。	**モジュールの決定** ファイルの位置，拡張子，およびファイルの内容によってモジュールを決定します。 **J2EE アプリケーションの表示名** EAR ファイル名によって決定します。 **EJB-JAR ディレクトリ，WAR ディレクトリの決定** 展開ディレクトリ形式の場合，EJB-JAR ディレクトリ名の最後は「_jar」，WAR ディレクトリ名の最後は「_war」となります。
cjexportapp	application.xml ありの J2EE アプリケーションをエクスポートします。	application.xml なしの J2EE アプリケーションをエクスポートします。
cjaddapp	J2EE アプリケーションへリソースを追加します。application.xml にモジュールの情報を追加します。追加できるリソースファイルの拡張子に制限はありません。	J2EE アプリケーションへリソースを追加します。その際，リソースファイルの拡張子が application.xml 省略時の Java EE 仕様に準拠していない場合はエラーとなります。
cjsetappprop	属性ファイルに指定した内容を設定します。	属性ファイルに指定した内容を設定します。設定する内容によって application.xml を作成します。
cjrenameapp	J2EE アプリケーション名を変更します。	J2EE アプリケーション名を変更します。application.xml を作成します。

参考

次に示すサーバ管理コマンドの機能には，application.xml がある場合と application.xml がない場合とで差異はありません。

- cjdeleteapp
- cjstartapp
- cjstopapp
- cjchmodapp
- cjgencmpsl
- cjgetappprop
- cjgetstubsjar
- cjimportlibjar
- cjdeletelibjar
- cjlistlibjar
- cjlistapp
- cjreloadapp
- cjreplaceapp

11.4.3　application.xml がある場合のモジュールの決定規則

アプリケーション内に application.xml がある場合，ライブラリ JAR 以外は Java EE 仕様に従って application.xml の内容からモジュールを決定します。ただし，application.xml のバージョンが 1.4 以前の場合，ライブラリ JAR はアプリケーションサーバ独自の仕様に従って決定します。ライブラリ JAR は，application.xml のバージョンが Java EE 5 以降の場合と，J2EE1.4 以前の場合とでモジュールの決定規則が異なります。ライブラリ JAR のモジュールの決定規則について，application.xml のバージョンが Java EE 5 以降の場合と，J2EE1.4 以前の場合とに分けて説明します。

(1)　application.xml のバージョンが Java EE 5 以降の場合

application.xml のバージョンが Java EE 5 以降の場合，次に示すファイルを除いて，ライブラリディレクトリ直下と，J2EE アプリケーションのルート直下にある JAR ファイル（拡張子が小文字の.jar のファイル）がライブラリ JAR と見なされます。

除外されるファイル

- META-INF ディレクトリ直下の application.xml の<module>タグに書かれているファイル
- J2EE アプリケーションのルート直下の hitachi-runtime.jar

application.xml のバージョンが Java EE 5 以降の場合のモジュールの決定規則を次に示します。

表 11-9　application.xml のバージョンが Java EE 5 以降の場合のモジュールの決定規則

<library-directory>タグの値		ライブラリ JAR※
存在するディレクトリ	/	<ファイル名>.jar
	/以外	• <ファイル名>.jar • <library-directory>タグの値/<ファイル名>.jar
存在しないディレクトリ		<ファイル名>.jar
存在するファイル		インポート時に KDJE42360-E を出力してエラーにする。
空文字		<ファイル名>.jar
タグ自体が存在しない		• <ファイル名>.jar • /lib（小文字）/<ファイル名>.jar

注※
「除外されるファイル」で示したファイルは除きます。

(2)　application.xml のバージョンが J2EE1.2，1.3，1.4 の場合

次に示すファイル以外の JAR ファイル（拡張子が小文字の.jar のファイル）がライブラリ JAR と見なされます。

- META-INF ディレクトリ直下の application.xml の<module>タグに書かれているファイル
- J2EE アプリケーションのルート直下の hitachi-runtime.jar

(3)　ライブラリ JAR として扱われる JAR ファイルの例

META-INF ディレクトリ直下の application.xml の<module>タグに<ライブラリディレクトリ>/<ファイル名>.jar と書いた場合，その JAR ファイルはライブラリ JAR と認識されません。また，hitachi-

runtime.jar もライブラリ JAR として認識されません。ライブラリ JAR として扱われる JAR ファイルの例を次に示します。

表 11-10　ライブラリ JAR として扱われる JAR ファイルの例

EAR ファイル内のパス	application.xml のバージョン			
	J2EE1.4	Java EE 5 以降		
		<library-directory>の値		
		lib	library	タグなし
lib1.jar	ライブラリ JAR	ライブラリ JAR	ライブラリ JAR	ライブラリ JAR
lib/lib2.jar	ライブラリ JAR	ライブラリ JAR	−	−
library/lib3.jar	ライブラリ JAR	−	ライブラリ JAR	−

（凡例）−：該当なし

11.4.4　application.xml がない場合のモジュールの決定規則

J2EE アプリケーションに application.xml が存在しない場合のモジュールの決定規則について説明します。J2EE アプリケーションに application.xml が存在しない場合は，アプリケーションの形式ごとにライブラリ JAR の決定規則が異なります。アプリケーションの形式ごとにモジュールの決定規則を説明します。

(1)　アーカイブ形式または-d オプションを指定してインポートした展開ディレクトリ形式のアプリケーションの場合

application.xml が含まれていないアプリケーションを，アーカイブ形式でインポートした場合，または展開ディレクトリ形式（cjimportapp コマンドに-d オプションを指定）でインポートした場合，Java EE 仕様に従ってモジュールが決定されます。モジュールが決定される順番を示します。

1. ファイルの拡張子が.war のファイルすべてを，Web モジュールと見なします。
2. ファイルの拡張子が.rar のファイルすべてをリソースアダプタと見なします。
3. lib ディレクトリをライブラリディレクトリと見なします。
4. lib ディレクトリ以下を除き，ファイルの拡張子が.jar のファイルについては次のように決定します。
   - META-INF/ejb-jar.xml ファイルを含む，または EJB コンポーネントアノテーションを含む JAR ファイルは EJB モジュールと見なします。
   - 「hitachi-runtime.jar」を除いた J2EE アプリケーションのルート直下の JAR ファイルをライブラリ JAR と見なします。

これらのルールに該当しないすべてのファイルは無視されます。

ポイント

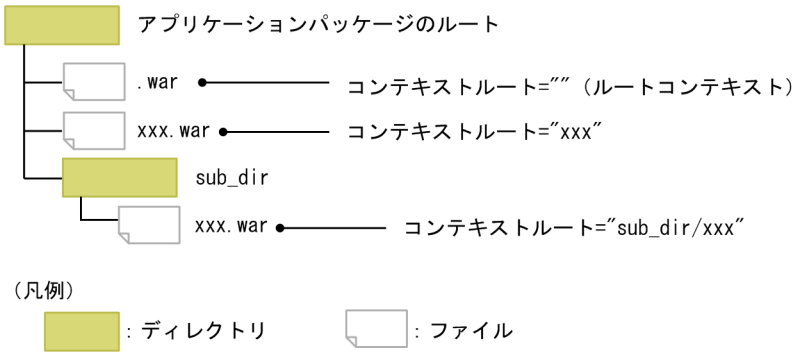
Web モジュールのコンテキストルート

Web モジュールのコンテキストルートは，アプリケーションパッケージのルートから WAR ファイルへの相対パス名から，拡張子である.war を除いたものになります。

このため，application.xml を省略する場合，WAR ファイルへのパスのディレクトリ名，および WAR ファイル名は，URI（RFC3986）で使用できる文字で指定してください。

Web モジュールの構成とコンテキストルートの例を次の図に示します。

図 11-10　Web モジュールの構成とコンテキストルートの例（アーカイブ形式または-d オプションを指定してインポートした展開ディレクトリ形式のアプリケーションの場合）



なお，WAR ファイルへのパス名によっては，複数の WAR ファイルに対するコンテキストルートが重複することがあります。例えば，「sub/.war」という名称の WAR ファイルと，「sub.war」という名称の WAR ファイルがある場合，どちらのコンテキストルートも「sub」になります。

(2)　-a オプションを指定してインポートした展開ディレクトリ形式のアプリケーションの場合

application.xml が含まれていないアプリケーションを，展開ディレクトリ形式（cjimportapp コマンドに-a オプションを指定）でインポートした場合，次の規則に従って順番に WAR ディレクトリ，EJB-JAR ディレクトリ，モジュールが決定されます。次に，モジュールが決定する順番を記します。

1. アプリケーションディレクトリのルート以下の階層に「_war」で終わるディレクトリがある場合，そのディレクトリを WAR ディレクトリと見なします。ただし，次の条件に当てはまる場合は，WAR ディレクトリとは見なしません。
   - WAR ディレクトリ以下のディレクトリ
   - EJB-JAR ディレクトリ以下のディレクトリ
   - lib ディレクトリ
2. アプリケーションディレクトリのルート以下の階層に「_jar」で終わるディレクトリがある場合，そのディレクトリを EJB-JAR ディレクトリと見なします。ただし，次の条件に当てはまる場合は，EJB-JAR ディレクトリとは見なしません。
   - WAR ディレクトリ以下のディレクトリ
   - EJB-JAR ディレクトリ以下のディレクトリ
   - lib ディレクトリおよび lib ディレクトリ以下のディレクトリ
3. WAR ディレクトリ以下，および EJB-JAR ディレクトリ以下のファイルを除いて，ファイル名の拡張子が.rar のファイルすべてをリソースアダプタと見なします。
4. lib ディレクトリをライブラリディレクトリと見なします。
5. WAR ディレクトリ以下，EJB-JAR ディレクトリ以下，および lib ディレクトリ以下のファイルを除き，ファイル名の拡張子が.jar のファイルについては「hitachi-runtime.jar」を除いた J2EE アプリケーションのルート直下の JAR ファイルをライブラリ JAR と見なします。

これらのルールに該当しないすべてのファイルは無視されます。

ポイント

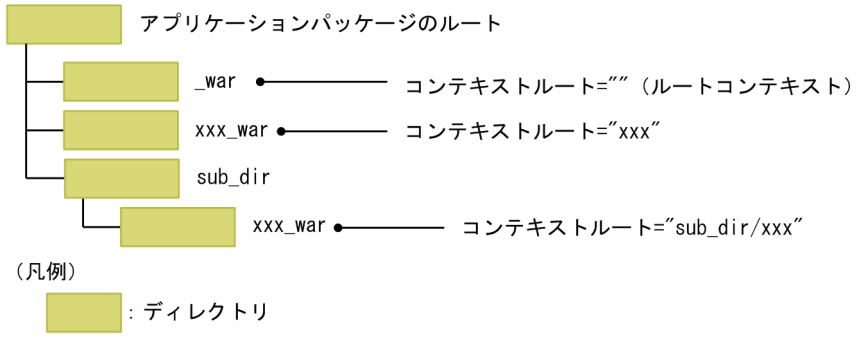
Web モジュールのコンテキストルート

Web モジュールのコンテキストルートは，アプリケーションディレクトリから WAR ディレクトリへの相対パス名の後ろから，「_war」を除いたものになります。

このため，application.xml を省略する場合，WAR ディレクトリへのパスのディレクトリ名，および WAR ディレクトリ名は，URI（RFC3986）で使用できる文字で指定してください。

Web モジュールの構成とコンテキストルートの例を次の図に示します。

図 11-11　Web モジュールの構成とコンテキストルートの例（-a オプションを指定してインポートした展開ディレクトリ形式のアプリケーションの場合）



なお，WAR ディレクトリへのパス名によっては，複数の WAR ディレクトリに対するコンテキストルートが重複することがあります。例えば，「sub/_war」という名称の WAR ディレクトリと，「sub_war」という名称の WAR ディレクトリがある場合，どちらのコンテキストルートも「sub」になります。

(3)　EAR ファイル内またはアプリケーションディレクトリのライブラリディレクトリの扱われ方

ここでは，application.xml がない場合の EAR ファイル内またはアプリケーションディレクトリのライブラリディレクトリの扱われ方について示します。ライブラリディレクトリの扱われ方は，EAR ファイル内またはアプリケーションディレクトリに，lib というディレクトリまたはファイルがある場合と，どちらもない場合で異なります。

表 11-11　application.xml がない場合の EAR ファイル内またはアプリケーションディレクトリのライブラリディレクトリの扱われ方

EAR ファイル内の lib またはアプリケーションディレクトリ内の lib	扱われ方
ディレクトリがある場合	lib がライブラリディレクトリとして扱われます。
ファイルがある場合	インポート時に KDJE42360-E が出力され，エラーになります。
ディレクトリまたはファイルがない場合	ライブラリディレクトリはないものとして動作します。 （ほかの個所にある lib は，ライブラリディレクトリとして扱われません）。

11.4.5　web.xml を省略した Web アプリケーションに対する操作

web.xml を省略した場合，サーバ管理コマンドでの Web アプリケーションに対する操作に制限があります。web.xml 省略時の Web アプリケーションに対する操作の可否について，次の表に示します。

表 11-12　web.xml 省略時の Web アプリケーションに対する操作の可否

操作				操作の可否
コマンド	引数	操作内容		
cjaddapp	-type filter	フィルタの追加		×※1
	-type war	WAR ファイルの追加		○
cjcopyres	-type war	WAR ファイルのコピー		○
cjdeleteapp	-type filter	フィルタの削除		×※2
	-type war	WAR ファイルの削除		○
	なし	アプリケーションの削除		○
cjdeleteres	-type war	WAR ファイルの削除		○
cjexportapp	-raw	アプリケーションのエクスポート（実行時情報を含まない）		○※3
	-normal	アプリケーションのエクスポート（実行時情報を含む）		○※3
cjgetappprop	-type war	WAR ファイルの属性の取得		○
	-type all	アプリケーションの属性の取得		○
cjgetresprop	-type war	WAR ファイルの属性の取得		○
cjimportapp		J2EE アプリケーションのインポート		○※4
cjimportres	-type war	WAR ファイルのインポート		○※4
cjlistapp	-type war	WAR ファイルの一覧表示		○
cjlistres	-type war	WAR ファイルの一覧表示		○
cjreloadapp		アプリケーションのリロード		○
cjreplaceapp	-replaceDD	アプリケーションの入れ替え（DD の定義を引き継がない）		○
	なし	アプリケーションの入れ替え（DD の定義を引き継ぐ）		○
cjsetappprop	-type war	属性設定	標準 DD に関する情報	×※5
			実行時情報	○
	-type all	属性設定	標準 DD に関する情報	×※5
			実行時情報	○
cjsetresprop	-type war	属性設定	標準 DD に関する情報	×※5
			実行時情報	○
cjstartapp		アプリケーションの開始		○
cjstopapp		アプリケーションの停止		○

（凡例）　○：操作できる　×：操作できない

注※1　コマンド実行時にエラーが発生し，メッセージ KDJE37606-E が出力されます。
注※2　web.xml を省略したアプリケーションにはフィルタを追加できないため，フィルタの削除もできません。
注※3　エクスポートしたファイルに web.xml を含みません。
注※4　インポートした WAR は，JSP および静的コンテンツだけが使用できます。
注※5　読み取り専用の属性として扱われるため，設定が無視されます。

11.4.6　DD を省略した場合に設定される表示名

アプリケーションサーバでは, DD を省略した場合インポートしたタイミングで表示名が補完されます。また，J2EE サーバが表示名の妥当性確認を実施します。インポート時の表示名の補完規則，および表示名の妥当性の確認について説明します。

(1)　インポート時の表示名の補完規則

EJB-JAR，WAR，RAR，または EAR の各 J2EE コンポーネントの DD の<display-name>タグが設定されていない場合, J2EE サーバによって表示名が設定されます。表示名は, J2EE コンポーネントのファイル名，およびディレクトリ名を基に決められます。DD を省略した場合も同様に J2EE サーバによって設定されます。その場合にファイル名，およびディレクトリ名として使用できる文字は半角英数字（0～9，A～Z，a～z），およびアンダースコア（_）です。EJB-JAR，WAR，RAR，または EAR の表示名を特定したい場合は，<display-name>タグを設定することをお勧めします。

(2)　表示名の妥当性の確認

J2EE アプリケーション，EJB-JAR，WAR，RAR およびその構成要素（Enterprise Bean，Servlet，JSP，Filter）の lang 属性「en」の表示名に使用できる文字は，英数字（0～9，A～Z，a～z），およびアンダースコア（_）です。J2EE サーバは表示名に半角英数字以外の文字が含まれている場合は，その文字をアンダースコア（_）に変換して使用します。J2EE サーバは J2EE アプリケーション，EJB-JAR，WAR，RAR をインポートする際に表示名の妥当性を確認して, 表示名に使用できない文字列が使用されている場合は警告メッセージ KDJE42374-W メッセージを出力します。

ただし，表示名が「TP1/Message Queue - Access」の場合は警告メッセージを出力しません。

表示名の妥当性を確認するサーバ管理コマンドについて説明します。

- cjimportapp コマンド
  アプリケーションの表示名およびアプリケーションに含まれる J2EE リソース（EJB-JAR，WAR，RAR，Enterprise Bean，Servlet/JSP，Filter）の表示名の妥当性が確認されます。実行時情報付き ZIP 形式では妥当性は確認されません。
- cjimportwar コマンド
  アプリケーションに含まれる J2EE リソース（WAR，Servlet/JSP，Filter）の表示名の妥当性が確認されます。
- cjimportres コマンド
  J2EE リソース（EJB-JAR，WAR，RAR，Enterprise Bean，Servlet/JSP，Filter）の表示名の妥当性が確認されます。実行時情報付き RAR 形式では妥当性は確認されません。

妥当性の確認が実施される表示名を次の表に示します。

表 11-13　妥当性の確認を行う表示名の一覧

項番	表示名を表す DD の要素	定義の種別
1	<application><display-name>	application.xml の定義
2	<ejb-jar><display-name>	ejb-jar.xml の定義
3	<ejb-jar><enterprise-beans><session><display-name>	ejb-jar.xml の定義
4	<ejb-jar><enterprise-beans><entity><display-name>	ejb-jar.xml の定義
5	<ejb-jar><enterprise-beans><message-driven><display-name>	ejb-jar.xml の定義
6	<web-app><display-name>	web.xml の定義
7	<web-app><filter><display-name>	web.xml の定義
8	<web-app><servlet><display-name>	web.xml の定義
9	<connector><display-name>	ra.xml の定義

11.4.7　application.xml を省略した場合に application.xml が作成される操作

インポートした J2EE アプリケーションに application.xml がない場合でも，次の操作を実行すると，application.xml が作成されます。

- cjsetappprop コマンドによって，各属性ファイルの次のどれかのタグの値を変更した場合

  アプリケーション属性ファイル

  <hitachi-application-property>/<description>
  <hitachi-application-property>/<icon>/<small-icon>
  <hitachi-application-property>/<icon>/<large-icon>

  WAR 属性ファイル

  <hitachi-war-property>/<war-runtime>/<context-root>

- cjrenameapp コマンドを実行した場合

11.4.8　application.xml を省略した場合に J2EE アプリケーションにリソースを追加するときの注意

インポート済みの EJB-JAR ファイル，WAR ファイル，または RAR ファイルを application.xml を省略した J2EE アプリケーションに追加する場合の注意事項を説明します。

J2EE アプリケーションに application.xml が存在しない場合，リソースファイルの拡張子は Java EE 仕様によって次のように決められています。

- EJB-JAR ファイルの拡張子は.jar
- WAR ファイルの拡張子は.war
- RAR ファイルの拡張子は.rar

リソースファイルの拡張子がこれらの拡張子ではない場合，J2EE アプリケーションへの追加はできません。追加時にメッセージ KDJE42366-E を出力し，エラーとなります。

12 アノテーションの使用

この章では，アノテーションの機能について説明します。
Servlet 2.5 以降または EJB 3.0 以降のアプリケーションでアノテーションを使用する場合に参照してください。

12.1　この章の構成

アノテーションとは，ソースコードにクラスやメソッドの付加情報などを埋め込むための記述方式です。アノテーションを使用することで，従来，DD（web.xml または ejb-jar.xml）で指定していた情報を，Servlet や Enterprise Bean のソースコード上に指定できます。

この章の構成を次の表に示します。

表 12–1　この章の構成（アノテーションの使用）

タイトル	参照先
アノテーションの指定	12.2
ロード対象のクラスとロード時に必要なクラスパス	12.3
DI の使用	12.4
アノテーションの参照抑止	12.5
アノテーションで定義した内容の更新	12.6
アノテーション使用時の注意事項	12.7

12.2　アノテーションの指定

この節では，アプリケーションサーバで指定できるアノテーションの種類，およびアノテーションを指定したアプリケーションの実装のポイントについて説明します。

この節の構成を次の表に示します。

表 12-2　この節の構成（アノテーションの指定）

分類	タイトル	参照先
解説	アノテーションを使用するメリットと指定できるアノテーション	12.2.1
	アノテーションを宣言したライブラリ JAR のクラスの使用	12.2.2
実装	アノテーションを指定する場合の Enterprise Bean の実装	12.2.3

注　「設定」「運用」および「注意事項」について，この機能固有の説明はありません。

12.2.1　アノテーションを使用するメリットと指定できるアノテーション

ここでは，アノテーションを使用するメリットと指定できるアノテーションについて説明します。

(1)　アノテーションを使用するメリット

アノテーションを使用するメリットを次に示します。

- ソースコードと DD に分散していた情報を，ソースコード上に集約できます。
- Web アプリケーションや Enterprise Bean の DD を作成する必要がありません。
- DI によって，ほかの Enterprise Bean やリソースへの参照を取得できます。

DI については，「12.4　DI の使用」を参照してください。

(2)　Web アプリケーションで指定できるアノテーション

Web アプリケーションにアノテーションを指定できるのは，Servlet 2.4，Servlet 2.5，または Servlet 3.0 の場合です。ただし，指定できるアノテーションは，Servlet のバージョンによって異なります。

指定できるアノテーションについては，マニュアル「アプリケーションサーバ リファレンス API 編」の「2.1　対応するアノテーションのサポート範囲」を参照してください。

なお，@Resource アノテーションで指定できるリソースのタイプについては，「12.4.1　@Resource アノテーションで指定できるリソースのタイプ」を参照してください。また，@EJB および@Resource アノテーションの name 属性は，web.xml のタグの要素と対応しています。対応については，標準仕様を確認してください。

ポイント

Servlet 2.3 以前のバージョンの web.xml がある場合，アノテーションの指定は無効になります。

(3)　Enterprise Bean で指定できるアノテーション

Enterprise Bean にアノテーションを指定できるのは，EJB 3.0 の場合です。

指定できるアノテーションについては，マニュアル「アプリケーションサーバ リファレンス API 編」の「2.1　対応するアノテーションのサポート範囲」を参照してください。

なお，@RemoteHome または@LocalHome アノテーションを使用してホームインタフェースを指定する場合，指定されたホームインタフェースの create メソッドの戻り値がコンポーネントインタフェースとみなされます。

12.2.2　アノテーションを宣言したライブラリ JAR のクラスの使用

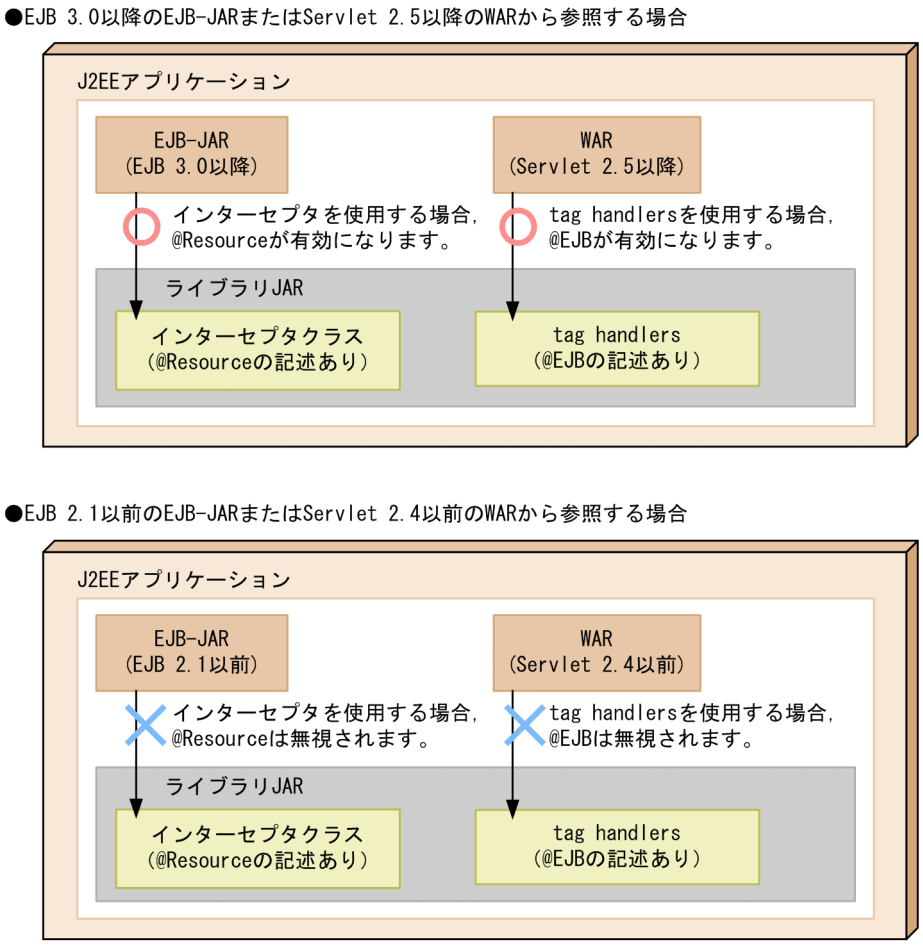
ライブラリ JAR に含まれるクラスは，各コンポーネントから使用されます。ライブラリ JAR のクラスに記述できるのは，そのクラスを使用するコンポーネントでサポートされているアノテーションです。

ライブラリ JAR で指定できるアノテーションには，次の制限があります。

- ライブラリ JAR では，コンポーネントを宣言するアノテーションは指定できません。指定しても無視されます。
- ライブラリ JAR 内のコンポーネントであるクラスに定義されたアノテーションは，サポート対象外とします。
- 次の EJB-JAR や WAR から参照された場合，ライブラリ JAR で宣言されているアノテーションは無視されます。
  - EJB 2.1 以前のモジュールを含む EJB-JAR
  - Servlet 2.4 以前のモジュールを含む WAR
- J2EE 1.4 の J2EE アプリケーションに含まれる EJB 3.0（DD なし）から参照された場合，ライブラリ JAR で宣言されているアノテーションは無視されます。

ライブラリ JAR 内のアノテーションを記述したクラスの使用例を次の図に示します。

図 12-1　ライブラリ JAR 内のアノテーションを記述したクラスの使用例



EJB 3.0 以降または Servlet 2.5 以降から参照した場合，ライブラリ JAR に含まれる「@Resource」や「@EJB」が有効になります。EJB 2.1 以前または Servlet 2.4 以前のコンポーネントから参照した場合，ライブラリ JAR に含まれる「@Resource」や「@EJB」は無視されます。ただし，参照元の J2EE アプリケーションのバージョンは，Java EE 5 以降であることが前提です。

12.2.3　アノテーションを指定する場合の Enterprise Bean の実装

Enterprise Bean の場合，アノテーションは，Session Bean で指定できます。ここでは，アノテーションを指定して，Stateless Session Bean，または Stateful Session Bean を実装する場合のポイントについて説明します。

(1)　Stateless Session Bean の実装

- Enterprise Bean の種類（Stateless Session Bean）は，@Stateless アノテーションで指定するか，または DD（ejb-jar.xml）に指定します。
- ビジネスインタフェースを使用して Session Bean を実装できます。
- javax.ejb.SessionBean インタフェースをインプリメントすることは，必須ではありません。
- インターセプタクラスは，必要に応じて実装します。
- Stateless Session Bean には，次のコールバックを定義できます。
  - @PostConstruct

- @PreDestroy

コールバック，またはメソッドが呼び出されるタイミングについては，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「2.2.3　Enterprise Bean のライフサイクル」を参照してください。

(2)　Stateful Session Bean の実装

- Enterprise Bean の種類（Stateful Session Bean）は，@Stateful アノテーションで指定するか，または DD（ejb-jar.xml）に指定します。
- ビジネスインタフェースを使用して Session Bean を実装できます。
- javax.ejb.SessionBean インタフェース，および java.io.Serializable インタフェースをインプリメントすることは，必須ではありません。
- インターセプタクラスは，必要に応じて実装します。
- 次のコールバックを定義できます。
  - @PostConstruct
  - @PreDestroy
- 実装クラスのメソッドに@Init アノテーションおよび@Remove アノテーションを指定できます。

コールバック，またはメソッドが呼び出されるタイミングについては，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「2.2.3　Enterprise Bean のライフサイクル」を参照してください。

12.3　ロード対象のクラスとロード時に必要なクラスパス

アノテーション情報は，次のような操作を実行したときに読み込まれます。

- リソースの追加（cjaddapp）※
- インポート時（cjimportapp）
- インポート時（cjimportwar）
- インポート時（cjimportres）
- インポート時（cjimportlibjar）※
- 開始時（cjstartapp）
- リプレース時（cjreplaceapp）
- リロード時（cjreloadapp）

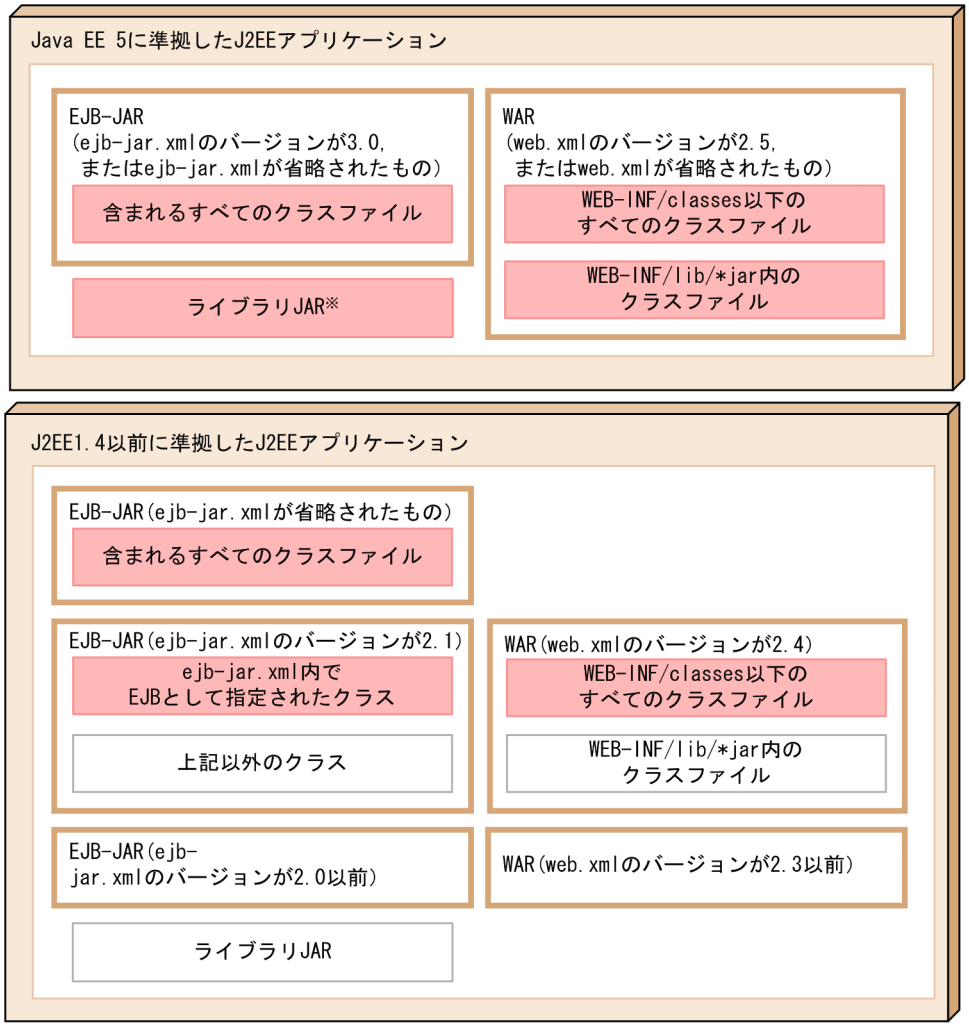
注※

対象となるのはライブラリ JAR のクラスです。

アノテーション情報を読み込むためには，まずクラスをロードする必要があります。

アノテーション情報を読み込むためにロードするクラスを次の図に示します。

図 12-2　アノテーション情報を読み込むためにロードするクラス



注※　プロパティejbserver.deploy.annotations.load_libjars.enabledの値にfalseが設定されている場合，ライブラリJARのクラス内のアノテーション情報を取得しません。

なお，EJB-JAR（ejb-jar.xml のバージョンが 2.1）に含まれるクラスファイルまたは WAR（web.xml のバージョンが 2.4）に含まれるクラスファイルからアノテーション情報を読み込む場合のアノテーションは，アプリケーションサーバ独自の仕様です。ただし，この機能は，旧バージョンとの互換機能です。詳細は，マニュアル「アプリケーションサーバ 機能解説 互換編」の「5.4　EJB 2.1 と Servlet 2.4 でのアノテーションの利用」を参照してください。

なお，J2EE 1.4 以前に準拠した J2EE アプリケーションの場合，EJB-JAR（ejb-jar.xml のバージョンが 2.1，または ejb-jar.xml が省略されたもの）や WAR（web.xml のバージョンが 2.4）からライブラリ JAR を使用することはできません。指定しても無視されます。

アノテーション情報を取得するためにクラスをロードするかどうかは，各モジュールのバージョン，および DD 内の属性定義に依存します。

モジュールのバージョンが EJB-JAR（3.0）または WAR（2.5）の場合のアノテーション情報を取得するかどうかの条件を次の表に示します。

表 12-3　モジュールのバージョンが EJB-JAR（3.0）または WAR（2.5）の場合

モジュール	DD の metadata-complete タグ		DD 省略
	false またはタグを省略	true	
EJB-JAR（3.0）	○	−	○
WAR（2.5）	○	−	○

（凡例）

○：アノテーション情報を取得する

−：アノテーション情報を取得しない

ライブラリ JAR のアノテーション情報を取得するかどうかを次の表に示します。

表 12-4　ライブラリ JAR の場合

モジュール	Java EE 5 以降に準拠する J2EE アプリケーション			J2EE 1.4 以前に準拠する J2EE アプリケーション		
	DD の metadata-complete タグ		DD 省略※	DD の metadata-complete タグ		DD 省略
	false またはタグを省略※	true		false またはタグを省略	true	
EJB-JAR（3.0）	○	−	○	×	×	−
WAR（2.5）	○	−	○	×	×	×

（凡例）

○：アノテーション情報を取得する（J2EE アプリケーションに一つでも該当するモジュールが含まれている場合，取得する）

−：アノテーション情報を取得しない

×：サポートされていない組み合わせである

注※　プロパティ ejbserver.deploy.annotations.load_libjars.enabled の値に false が設定されている場合，ライブラリ JAR のクラス内のアノテーション情報を取得しません。

! 注意事項

アノテーション情報を読み込むために必要なクラスが不足している場合，または不正なクラスがある場合は，アノテーション情報の取得処理中に例外が発生します。クラスのロード時に例外が発生した場合の挙動について次に示します。

アプリケーションサーバのバージョンが 08-00 以降の場合

例外ログ，およびメッセージ KDJE42380-W を出力し処理を続行します。なお，例外が発生した場合にエラーにしたいときは，プロパティ ejbserver.deploy.annotations.load_check.enabled の値を false から true へ変更してください。

アプリケーションサーバのバージョンが 08-00 よりも前の場合

エラーとなり処理を中断します。

12.4　DI の使用

Dependency Injection（DI）とは，EJB やインターセプタクラスのフィールドや setter メソッドに，@EJB や@Resource アノテーションを指定することで，Enterprise Bean やリソースへの参照を EJB コンテナが自動的にセットする機能です。DI を使用すると，Enterprise Bean やリソースへの参照を，JNDI を使用してルックアップする必要がなくなります。ここでは，@Resource アノテーションで指定できるリソースのタイプ，@Resource アノテーションを使用したリソースの参照解決，および DI に関する注意事項について説明します。

この節の構成を次の表に示します。

表 12-5　この節の構成（DI の使用）

分類	タイトル	参照先
解説	@Resource アノテーションで指定できるリソースのタイプ	12.4.1
	@Resource アノテーションを使用したリソースの参照解決	12.4.2
	DI 失敗時の動作	12.4.3
注意事項	注意事項	12.4.4

注　「実装」,「設定」および「運用」について，この機能固有の説明はありません。

12.4.1　@Resource アノテーションで指定できるリソースのタイプ

@Resource アノテーションを使用して，リファレンスを定義できます。@Resource アノテーションで指定できるリソースのタイプを次の表に示します。

表 12-6　@Resource アノテーションで指定できるリソースのタイプ

リソースのタイプ	指定可否
javax.sql.DataSource※1	○
javax.mail.Session	○
java.net.URL	×
javax.jms.ConnectionFactory	○
javax.jms.QueueConnectionFactory※2	○
javax.jms.TopicConnectionFactory	○
javax.jms.Queue※2	○
javax.jms.Topic	○
javax.resource.cci.ConnectionFactory※3	○
javax.resource.cci.InteractionSpec	×
javax.transaction.UserTransaction	○※4

リソースのタイプ	指定可否
org.omg.CORBA_ 2_3.ORB	○※5
javax.xml.rpc.Service	×
javax.xml.ws.Service	×
javax.jws.WebService	×
javax.ejb.EJBContext	○※6
javax.ejb.SessionContext	○※6
javax.ejb.TimerService	○※6※7
JavaBeans リソース独自	○
java.lang.String	○※8
java.lang.Character	○※8
java.lang.Integer	○※8
java.lang.Boolean	○※8
java.lang.Double	○※8
java.lang.Byte	○※8
java.lang.Short	○※8
java.lang.Long	○※8
java.lang.Float	○※8

（凡例）　○：指定できる　×：指定できない

注※1　DB Connector が該当します。

注※2　TP1/Message Queue - Access，Reliable Messaging が該当します。

注※3　TP1 Connector が該当します。

注※4　CMT で動作する Enterprise Bean またはインターセプタでは使用できません。

注※5　ORB の shareable 属性は true が指定されているものとして動作します。なお，注入される ORB オブジェクトは，ほかのコンポーネントでも使用される共有のインスタンスです。

注※6　Web コンテナ上で動作するクラスでは使用できません。

注※7　Stateful SessionBean や Stateful SessionBean に適用されたインターセプタでは使用できません。

注※8　<env-entry-value>タグに，DI またはルックアップで取得できる値を設定できません。

12.4.2　@Resource アノテーションを使用したリソースの参照解決

@Resource アノテーションを使用してリソースの参照解決をする方法には，次の 2 種類があります。

- name 属性にリソースの別名を指定する方法
- mappedName 属性に参照先リソースを指定する方法

@Resource アノテーションに属性を指定しない場合は，cosminexus.xml または cosminexus.xml 以外の属性ファイルによってリンクを解決する必要があります。ここでは，それぞれの方法について説明します。

(1)　@Resource アノテーションの name 属性で指定する方法

@Resource アノテーションの name 属性にリソースの別名を指定します。

@Resource アノテーションの name 属性に指定できるリソースを次に示します。

- javax.sql.DataSource
- javax.jms.ConnectionFactory
- javax.jms.QueueConnectionFactory
- javax.jms.TopicConnectionFactory
- javax.resource.cci.ConnectionFactory

コーディング例を次に示します。この例は，@Resource アノテーションの name 属性に別名「jdbc/ds」を設定する場合の例です。

```
package sample;

@Stateless public class MySessionBean implements MySession {
 @Resource(name="jdbc/ds") public DataSource customerDB;
    ...
}
```

@Resource アノテーションの name 属性にリソースアダプタの別名を指定した場合，参照解決が自動で実施されます。そのため，属性ファイルを編集する必要はありません。属性ファイルを取得した場合は，次のようになります。

```
<resource-ref>
  <description xml:lang="en"></description>
  <res-ref-name>jdbc/ds</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
  <injection-target>
     <injection-target-class>sample.MySessionBean</injection-target-class>
     <injection-target-name>customerDB</injection-target-name>
  </injection-target>
  <linked-to></linked-to>
</resource-ref>
```

(2)　@Resource アノテーションの mappedName 属性で指定する方法

@Resource アノテーションの mappedName 属性にリソースの表示名やキュー表示名を指定します。

@Resource アノテーションの mappedName 属性に記述したリソース表示名が参照解決に使用されます。

コーディング例を次に示します。この例は，@Resource アノテーションの mappedName 属性にリソース表示名「DB_Connector_for_Oracle」を設定する場合の例です。

```
package sample;

@Stateless public class MySessionBean implements MySession {
 @Resource(mappedName="DB_Connector_for_Oracle") public DataSource customerDB;
    ...
}
```

ここでは，@Resource の name 属性を省略しているため，デフォルト値「クラス名/フィールド名」が使用されます。この場合のデフォルト値は「sample.MySessionBean/customerDB」となります。属性ファイルを取得した場合は，次のようになります。

```
<resource-ref>
  <description xml:lang="en"></description>
  <res-ref-name>sample.MySessionBean/customerDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
  <mapped-name>DB_Connector_for_Oracle</mapped-name>
  <injection-target>
     <injection-target-class>sample.MySessionBean</injection-target-class>
     <injection-target-name>customerDB</injection-target-name>
  </injection-target>
  <linked-to></linked-to>
</resource-ref>
```

(3)　cosminexus.xml またはそれ以外の属性ファイルで指定する方法

@Resource アノテーションの属性を指定しない場合，cosminexus.xml や cosminexus.xml 以外の属性ファイルを使って参照解決をする必要があります。cosminexus.xml，または cosminexus.xml 以外の属性ファイルの<linked-to>に，リソースの表示名を指定します。参照先リソースがキューの場合は，<linked-queue>にリソースアダプタの表示名とキュー表示名を記述します。参照先リソースが管理対象オブジェクトの場合は，<linked-adminobject>にリソースアダプタの表示名と管理対象オブジェクト名を記述します。

コーディング例を次に示します。この例では，@Resource アノテーションの name 属性を省略しているため，別名としてデフォルト値の「クラス名/フィールド名」が使用されます。ここでは，「sample.MySessionBean/customerDB」が設定されます。

```
package sample;

@Stateless public class MySessionBean implements MySession {
 @Resource public DataSource customerDB;
    ...
}
```

cosminexus.xml で参照解決をする場合は，<link-to>，<linked-queue>または<linked-adminobject>にリソースの表示名を指定してから，アプリケーションをインポートします。cosminexus.xml 以外の属性ファイルで参照解決をする場合は，アプリケーションをインポートしたあとで，属性ファイル（アプリケーション統合属性ファイルなど）を取得します。取得した属性ファイルに，<linked-to><linked-queue>または<linked-adminobject>に参照先となるリソース表示名を記述します。

属性を設定したあとで属性ファイルを取得した場合は，次のようになります。

```
<resource-ref>
  <description xml:lang="en"></description>
  <res-ref-name> sample.MySessionBean/customerDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
  <injection-target>
     <injection-target-class>sample.MySessionBean</injection-target-class>
     <injection-target-name>customerDB</injection-target-name>
  </injection-target>
  <linked-to>DB_Connector_for_Oracle</linked-to>
</resource-ref>
```

(4)　参照解決方法の優先順位

@Resource アノテーションを使用してリソースの参照解決をする場合に，複数の方法を同時に設定したときには，次の優先順位で設定が有効になります。

1. cosminexus.xml または cosminexus.xml 以外の属性ファイルによる設定
2. @Resource アノテーションの mappedName 属性の設定

3. @Resource アノテーションの name 属性の設定

J2EE リソースに異なる別名が同時に設定された場合に有効となる別名の設定について表に示します。

表 12–7　複数の方法を同時に設定した場合に有効となる参照先の設定

name 属性の設定の有無※1	mappedName 属性の設定の有無	属性ファイル※2 の<linked-to>などの設定の有無	有効となる別名の設定※3
あり	あり	あり	属性ファイル※2 の<linked-to>などの設定
		なし	@Resource アノテーションの mappedName 属性の設定
	なし	あり	属性ファイル※2 の<linked-to>などの設定
		なし	@Resource アノテーションの name 属性の設定
なし	@Resource アノテーションがフィールド・メソッドに指定されていればデフォルト値が使用されて，「あり」の場合と同様になります※4。 また，@Resource アノテーションがクラスに指定された場合は，@Resource アノテーションの name 属性は省略できません。クラスに指定された@Resource の name 属性がない場合，アノテーションの読み込みに失敗して，インポート時にエラーになります。※5		

（凡例）

あり：アノテーションや属性ファイルで属性が設定されていることを示します。

なし：「なし」の場合の説明を次に示します。

- @Resource アノテーションの name 属性および@Resource アノテーションの mappedName 属性そのものの記述がない場合を示します。または，cosminexus.xml または cosminexus.xml 以外の属性ファイルの<linked-to>のタグそのものがない場合を示します。
- 属性またはタグに指定された値がない（空文字）場合を示します。

注※1　@Resource アノテーションの name 属性に対応する標準 DD の<resource-ref>または<res-ref-name>にもリソースの別名を指定できます。@Resource アノテーションの mappedName 属性，および属性ファイルの<linked-to>などに指定がない場合は，標準 DD の<resource-ref>または<res-ref-name>に指定した別名を持つリソースが参照先となります。

注※2　cosminexus.xml および cosminexus.xml 以外の属性ファイルが該当します。

注※3　別名の設定がない場合，アプリケーションの開始に失敗します。

注※4　@Resource アノテーションの mappedName 属性と属性ファイル（cosminexus.xml および cosminexus.xml 以外の属性ファイル）の<linked-to>などに指定がなく，name 属性のデフォルト値，またはクラスに指定した @Resource アノテーションの name 属性の値とリソースの別名が一致した場合は，その別名を持つリソースが参照先となります。

注※5　mappedName 属性と属性ファイル（cosminexus.xml および cosminexus.xml 以外の属性ファイル）の<linked-to>などに指定がなく，クラスに指定した@Resource の name 属性の値とリソースの別名が一致した場合は，その別名を持つリソースが参照先となります。

12.4.3　DI 失敗時の動作

DI が失敗した時の動作について説明します。

DD の<injection-target-class>タグに指定したクラスがない場合，または<injection-target-name>タグに指定したメソッドおよびフィールドがない場合，DI に失敗します。この場合のアプリケーション開始時，および DI 実行時の挙動を次に示します。

(1)　DD の<injection-target-class>タグに指定したクラスがない場合

挙動を次に示します。

アプリケーション開始時

メッセージ（KDJE53905-W）を出力します。アプリケーション開始処理は続行します。

DI 実行時

<injection-target-class>タグに指定したクラスがないため，DI は実行できません。

(2)　DD の<injection-target-name>タグに指定したメソッドおよびフィールドがない場合

挙動を次に示します。

アプリケーション開始時

メッセージ（KDJE53905-W）を出力します。アプリケーション開始処理は続行します。

DI 実行時

メッセージ（KDJE53900-E）を出力して，DI に失敗します。

12.4.4　注意事項

DI を使用した場合の注意事項を示します。

- DI を使用した場合の，Stateful Session Bean のリファレンス取得時の注意事項を次に示します。
  - J2EE アプリケーションがサーブレットから Stateful Session Bean を呼び出す構成の場合，サーブレットで Stateful Session Bean のリファレンスを取得するときは，DI を使用しないで，JNDI 経由で行ってください。
  - J2EE アプリケーションが Stateless Session Bean または Singleton Session Bean から Stateful Session Bean を呼び出す構成の場合に，DI を使用して Stateful Session Bean のビジネスインタフェースを Stateless Session Bean に注入するとき，Stateless Session Bean のビジネスメソッド呼び出し，またはタイムアウトメソッド呼び出しのたびに DI が実行されます。
- @Resource アノテーションで指定できないリソースのタイプを指定した場合，インポート時に例外が発生します。
- @EJB および@Resource アノテーションの name 属性は，次の表に示す ejb-jar.xml のタグの要素と対応しています。このため，アノテーションで Enterprise Bean およびリソースへの参照を定義する場合は，@EJB および@Resource アノテーションの name 属性と，name 属性に対応する ejb-jar.xml のタグに指定する要素は，重複しないように設定してください。name 属性に対応する ejb-jar.xml のタグを次の表に示します。

表 12–8　name 属性に対応する ejb-jar.xml のタグ

アノテーションと属性	name 属性に対応する ejb-jar.xml のタグ
@EJB　name()	<ejb-ref>タグ下の<ejb-ref-name>タグ
	<ejb-local-ref>タグ下の<ejb-ref-name>タグ
@Resource　name()	<env-entry>タグ下の<env-entry-name>タグ
	<resource-ref>タグ下の<res-ref-name>タグ
	<resource-env-ref>タグ下の<resource-env-ref-name>タグ

- DI を使用する Web アプリケーションのリロード時には，例外が発生することがあります。例外が発生する場合を次に示します。
  - DI ターゲット（DI 機能を使用して参照を注入する対象）を定義しているクラス，または参照先クラスのロードに失敗した場合
  - DI ターゲット名に対応したフィールド，メソッドがリロード後に削除されて，存在しない場合

  例外が発生すると，メッセージログに KDJE53904-W のメッセージが出力され，処理が続行されます。処理は続行されますが，対象となるインスタンスに対する DI が実行できない状態になり，対象のインスタンスへのリクエストが実行できなくなります。このため，このメッセージが出力された場合は，Web アプリケーションを修正して再度リロードを実施してください。

12.5　アノテーションの参照抑止

この節では，**アノテーションの参照抑止機能**について説明します。

アノテーション参照抑止機能を使用すると，アノテーションを使用していないモジュールに対する解析処理を抑止できます。これによって，不要なオーバーヘッドや不要な解析エラーの発生を防げます。

この節の構成を次の表に示します。

表 12–9　この節の構成（アノテーション参照抑止機能）

分類	タイトル	参照先
解説	アノテーション参照抑止機能の目的と適用範囲	12.5.1
	アノテーションを参照するタイミング	12.5.2
実装	DD での定義（モジュール単位の設定）	12.5.3
運用	アノテーション参照抑止機能の設定変更	12.5.4

注　「設定」および「注意事項」について，この機能固有の説明はありません。

12.5.1　アノテーション参照抑止機能の目的と適用範囲

ここでは，アノテーション参照抑止機能の目的と適用範囲について説明します。

(1)　アノテーション参照抑止機能の目的

アノテーション参照抑止機能は，J2EE アプリケーション中のアノテーションの参照（解析処理）を実施するかどうかを設定する機能です。モジュール内のアノテーションの有無を判断するためには，アプリケーションの解析が必要です。しかし，すべてのアプリケーションを解析すると，アノテーションを使用していないアプリケーションの場合，不要なオーバーヘッドが発生します。また，リソースやほかのアプリケーションを参照するアプリケーションの場合，アノテーション解析時にクラスファイルの参照エラーが発生して，デプロイに失敗することがあります。

アノテーション参照抑止機能を使用すると，アノテーションを使用しない場合に，アノテーションの解析処理を抑止できます。

アノテーションの参照抑止機能は，Servlet 2.5 以降の場合に使用できます。EJB 3.0 以降の場合，アノテーション参照抑止機能は使用できません。

Servlet 2.5 以降の場合の動作を次に示します。

Servlet 2.5 以降の場合

Java EE 5 以降のアプリケーションでは，標準仕様でアノテーションを使用できます。Servlet 2.5 以降の DD には，アノテーションの参照を抑止するための属性である metadata-complete 属性が追加されています。metadata-complete 属性を使用して，モジュール単位にアノテーションの参照を抑止できます。

ただし，DD が省略されているアプリケーションの場合は，モジュール定義情報を読み込むために必ずアノテーションが参照されるため，参照抑止機能は使用できません。

モジュール単位のアノテーション参照抑止機能は，Java EE 5 以降の標準仕様に準拠した機能です。

参考

アプリケーションサーバでは，EJB 3.0 の DD には<display-name>要素のほか，インターセプタおよびアプリケーション例外に関する定義だけが定義できます。EJB 3.1 ではこれらに加え，<module-name>要素が定義できますが，それ以外の EJB 3.0 または EJB 3.1 のモジュール定義情報は，すべてアノテーションで定義する必要があります。このため，EJB 3.0 以降ではアノテーション参照抑止機能を使用できません。

! 注意事項

アノテーションを記述した EJB 2.1 または Servlet 2.4 のモジュールを含む J2EE アプリケーションがインポートされた状態で，アノテーション参照抑止機能を無効から有効に変更した場合，次の現象が発生します。

- 属性ファイルにアノテーション情報が出力される
- @Resource または@Resources を使用していた場合，属性ファイルの設定および J2EE アプリケーションの開始に失敗する

(2)　適用範囲

アノテーション参照抑止機能の適用範囲は，アノテーションを含むモジュールの種類によって異なります。

アノテーション抑止機能の適用範囲を次の表に示します。

表 12-10　アノテーション抑止機能の適用範囲

モジュールのバージョン		DD の有無と指定内容		参照抑止の範囲	
				J2EE サーバ単位	モジュール単位
EJB	1.1，2.0	DD あり（metadata-complete 属性なし）		－	－
	2.1※1	DD あり（metadata-complete 属性なし）		○※2	－
	3.0 以降	DD なし		×	○
		DD あり（metadata-complete 属性あり）	省略	×	○
			指定あり	×	○※3
Servlet	2.2，2.3	DD あり（metadata-complete 属性なし）		－	－
	2.4※1	DD あり（metadata-complete 属性なし）		○※2	－
	2.5 以降	DD なし		×	○※4
		DD あり（metadata-complete 属性あり）	省略	×	○※4
			指定あり	×	○※4

（凡例）○：有効になる　×：使用できない　－：該当しない

注※1　EJB 2.1 および Sevlet 2.4 でのアノテーションの利用は，旧バージョンとの互換用の機能です。詳細は，マニュアル「アプリケーションサーバ 機能解説 互換編」の「5.4　EJB 2.1 と Servlet 2.4 でのアノテーションの利用」を参照してください。

注※2　有効にすると，アノテーションを記述した EJB 2.1 または Servlet 2.4 のモジュールを含む J2EE アプリケーションは使用できなくなります。

注※3　値に true は指定できません。true を指定した場合，J2EE サーバへのアプリケーションのインポートおよびリデプロイに失敗します。

注※4　Servlet3.0 で追加された@HandlesTypes アノテーションはアノテーション参照抑止機能の対象になりません。

モジュール単位の設定は DD の metadata-complete 属性で設定します。J2EE サーバ単位の設定は，ejbserver.deploy.applications.metadata_complete プロパティで設定します。モジュール単位の設定と J2EE サーバ単位の設定の組み合わせた場合にアノテーションが参照されるかどうかを次の表に示します。

表 12-11　J2EE サーバ単位の設定とモジュール単位の設定の組み合わせ

モジュールのバージョン	モジュール単位の設定		J2EE サーバ単位の設定		
			プロパティの指定なし	ejbserver.deploy.applications.metadata_complete プロパティ	
				false	true
EJB 3.0 以降	DD なし		○	○	○
	DD あり（metadata-complete 属性あり）	省略	○	○	○
		false	○	○	○
		true	―※	―※	―※
Servlet 2.5 以降	DD なし		○	○	○
	DD あり（metadata-complete 属性あり）	省略	○	○	○
		false	○	○	○
		true	×	×	×

（凡例）○：参照する　×：参照しない　－：対象外

注※　DD の metadata-complete 属性に true を指定した EJB 3.0 を含むアプリケーションは，インポートおよびリデプロイに失敗します。

12.5.2　アノテーションを参照するタイミング

アノテーションを参照するタイミングを，アプリケーション参照抑止機能を使用する場合としない場合に分けて次の表に示します。なお，J2EE サーバ単位，モジュール単位どちらの参照抑止機能でも，アノテーションを参照するタイミングは同じです。

表 12-12　アノテーションを参照するタイミング

タイミング（実行したコマンド）	参照抑止機能の設定	アプリケーションの形式	
		アーカイブ形式	展開ディレクトリ形式
インポート時（cjimportapp）	使用する	×	×
	使用しない	○	○
インポート時（cjimportwar）	使用する	×	×
	使用しない	○	○
インポート時（cjimportres※1）	使用する	×	－
	使用しない	○	－
開始時（cjstartapp※2）	使用する	×	×
	使用しない	○※3	○

タイミング（実行したコマンド）	参照抑止機能の設定	アプリケーションの形式	
		アーカイブ形式	展開ディレクトリ形式
リプレース時（cjreplaceapp）	使用する	×	−
	使用しない	△	−
リロード時（cjreloadapp）	使用する	−	×
	使用しない	−	○
属性取得時（cjgetappprop）	使用する	×	×
	使用しない	×	×
属性設定時（cjsetappprop）	使用する	×	×
	使用しない	×	×

（凡例）

使用しない：アプリケーション参照抑止機能を使用しない設定

使用する：アプリケーション参照抑止機能を使用する設定

○：アノテーションを参照する

×：アノテーションを参照しない

△：アノテーションを参照するが変更は反映されない

−：該当しない

注※1

cjimportres コマンドに引数「-type ejb」または「-type war」を指定して実行した場合です。

注※2

J2EE サーバの再起動時も含みます。

注※3

その前に J2EE サーバを起動したときと設定が異なる場合だけアノテーションをロードします。

参考

展開ディレクトリ形式のアプリケーションをインポートしたあとでアノテーション情報を変更した場合，cjgetappprop コマンドによって属性ファイルを取得しても変更後のアノテーション情報は取得できません。これは，属性ファイルを取得する時にはアノテーション情報がクラスファイルからロードされないためです。

変更後のアノテーション情報を属性ファイルに取得するためには，cjstartapp コマンドなど，アノテーション情報をクラスファイルからロードするコマンドを実行してから，属性ファイルを取得してください。

12.5.3　DD での定義（モジュール単位の設定）

Servlet 2.5 以降のモジュールに対するアノテーション参照抑止機能の定義は，web.xml に指定します。なお，EJB 3.0 以降の場合，<ejb-jar>タグの metadata-complete 属性には false だけが指定できます。

DD でのアノテーション参照抑止機能の定義について次の表に示します。

表 12-13　DD でのアノテーション参照抑止機能の定義

項目	DD の種類	指定するタグおよび属性	設定内容
Servlet 2.5 以降	web.xml	<web-app>タグの metadata-complete 属性	アノテーション参照を抑止する場合は true を設定します。

項目	DD の種類	指定するタグおよび属性	設定内容
Servlet 2.5 以降	web.xml	<web-app>タグの metadata-complete 属性	アノテーションを参照する場合は false を設定します。
EJB 3.0 以降	ejb-jar.xml	<ejb-jar>タグの metadata-complete 属性	アノテーションを参照する場合は false を設定します。

設定例を次に示します。

設定例（Servlet 2.5 の場合）

```
<web-app metadata-complete="true"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd"  version="2.5">
  :
</web-app>
```

太字の部分がアノテーション参照抑止機能の設定です。

DD の metadata-complete 属性の指定を省略した場合，または DD を省略した場合は，metadata-complete 属性に false を指定した場合と同様に動作します。

12.5.4　アノテーション参照抑止機能の設定変更

モジュール単位の設定（DD の metadata-complete 属性の値）を変更する場合は，metadata-complete 属性を変更したアプリケーションを作成して，J2EE サーバに再度インポートする必要があります。

metadata-complete 属性を変更した場合，リデプロイ機能（-replaceDD オプションを指定した cjreplaceapp コマンド）によってアプリケーションを入れ替えようとすると，エラーになり，入れ替え処理は中断されます。

12.6　アノテーションで定義した内容の更新

この節では，アノテーションで定義した内容の更新方法について説明します。

アノテーションを更新した場合，アプリケーションをインポートし直すか，リロード機能を使用することで，アプリケーションに変更を反映できます。また，Java EE 5 以降の標準仕様では，アノテーションで定義した情報を標準 DD で上書きできることが決められています。アプリケーションサーバでは，標準 DD によるアノテーションでの定義内容の上書き，およびサーバ管理コマンドを使用した定義内容の更新ができます。

この節の構成を次の表に示します。

表 12-14　この節の構成（アノテーションで定義した内容の更新）

分類	タイトル	参照先
解説	アノテーションの更新	12.6.1
	DD によるアノテーションの上書き	12.6.2
	サーバ管理コマンドを使用した定義の参照と更新	12.6.3

注　「実装」,「設定」および「運用」について，この機能固有の説明はありません。

12.6.1　アノテーションの更新

アノテーションの定義内容を更新した場合は，次のどちらかの方法でアプリケーションに反映できます。

アプリケーションをインポートし直す方法

通常の入れ替えによって更新内容を反映する方法です。アプリケーションを停止してから入れ替えます。

リロード機能を使用する方法（展開ディレクトリ形式のアプリケーションの場合）

更新検知または cjreloadapp コマンドの実行によって，デプロイ済みのアプリケーションの内容を動的に入れ替える方法です。展開ディレクトリ形式のアプリケーションの場合に使用できます。

ただし，リロード機能で更新できるアノテーションには制限があります。各アノテーションのリロード機能での更新可否を次の表に示します。

表 12-15　各アノテーションのリロード機能での更新可否

アノテーション名	更新可否
@PostConstruct	○
@PreDestroy	○
@Resource	×
@Resources	×
@RunAs	×
@DeclareRoles	○
@RolesAllowed	×
@PermitAll	○

アノテーション名	更新可否
@DenyAll	○
@Stateless	×
@Stateful	×
@Singleton	×
@DependsOn	○
@Startup	○
@AccessTimeout	○
@Lock	○
@ConcurrencyManagement	○
@Init	○
@Remove	○
@AfterBegin	○
@BeforeCompletion	○
@AfterCompletion	○
@Remote	×
@Local	×
@RemoteHome	×
@LocalHome	×
@LocalBean	×
@TransactionManagement	○
@TransactionAttribute	○
@PostActivate	○
@PrePassivate	○
@Interceptors	○
@AroundInvoke	○
@ExcludeDefautInterceptors	○
@ExcludeClassInterceptors	○
@Timeout	×
@Schedule	×
@Schedules	×
@Asynchronous	○

アノテーション名	更新可否
@ApplicationException	○
@EJB	×
@EJBs	×
@WebService	○
@WebServiceProvider	○
@PersistenceContext	○
@PersistenceContexts	○
@PersistenceProperty	○
@PersistenceUnit	○
@PersistenceUnits	○
@WebEndpoint	×
@WebServiceRef	×

（凡例）○：更新できる　×：更新できない

12.6.2　DD によるアノテーションの上書き

アノテーションに定義した情報は，DD で上書きできます。DD によって上書きできるのは，**Servlet 2.5 以降および EJB3.0 以降のモジュール**に定義されたアノテーションです。

アノテーションを上書きする方法には，次の 2 種類があります。

DD を編集して上書きする方法

DD を直接編集する方法です。

サーバ管理コマンドを使用して属性を変更して上書きする方法

J2EE サーバにインポート後のアプリケーションに対して，cjsetappprop コマンドまたは cjsetresprop コマンドを使用して属性を変更する方法です。ただし，この方法で上書きできるアノテーションには条件があります。詳細については「12.6.3　サーバ管理コマンドを使用した定義の参照と更新」を参照してください。

参考

アプリケーションサーバにインポートする EJB 3.0 以降の DD には次の定義ができます。

- <display-name>要素
- <interceptor-binding>要素，およびその配下の要素（インターセプタに関する定義）
- <application-exception>要素，およびその配下の要素（アプリケーション例外に関する定義）

! 注意事項

@WebServlet アノテーションの name 属性には，別の@WebServlet アノテーションと同じ値を設定できません。また，@WebFilter アノテーションの filtername 属性には，別の@WebFilter アノテーションと同じ値を設定できません。別のアノテーションと同じ値を設定した場合，DD での上書き時に次のメッセージが出力され，該当するアノテーションは無視されます。

アノテーション	属性	メッセージ
@WebServlet	name 属性	KDJE42397-W
@WebFilter	filtername 属性	KDJE42398-W

上書き方法は，対象となる DD の要素および子要素が複数回出現できる要素かどうかによって異なります。

DD の要素が 0 回（出現なし）または 1 回しか出現しない要素の場合

アノテーションの情報は DD の情報で上書きされます。

DD の要素が 0 回以上または 1 回以上出現する要素の場合

DD とアノテーションを関連づける要素（**キー**）の有無，および DD の要素内の子要素の出現回数で上書き規則が異なります。

ここでは，DD によるアノテーションの上書き規則を次の四つのパターンに分けて説明します。

表 12-16　DD によるアノテーションの上書き規則のパターン

パターン	アノテーションと対応する DD の要素の出現回数	キーの有無	DD の子要素の出現回数	参照先
パターン 1	0 回または 1 回	キーがない場合	－	(1)
パターン 2	0 回以上または 1 回以上	キーがない場合	－	(2)
パターン 3		キーがある場合	0 回または 1 回	(3)
パターン 4			0 回以上または 1 回以上	(4)

（凡例）－：キーの有無による違いがない

アノテーションと DD の要素の対応，各要素のキーの有無，およびキーとなる要素については，標準仕様を参照してください。

それぞれのパターンについて説明します。

(1)　パターン 1（0 回または 1 回出現する要素の場合）

出現回数が 0 回または 1 回出現する要素の場合は，アノテーションに対応する要素が DD に定義されているかどうかで上書きするかどうかが決まります。キーの値は関係ありません。

アノテーションに対応する要素が DD に定義されている場合

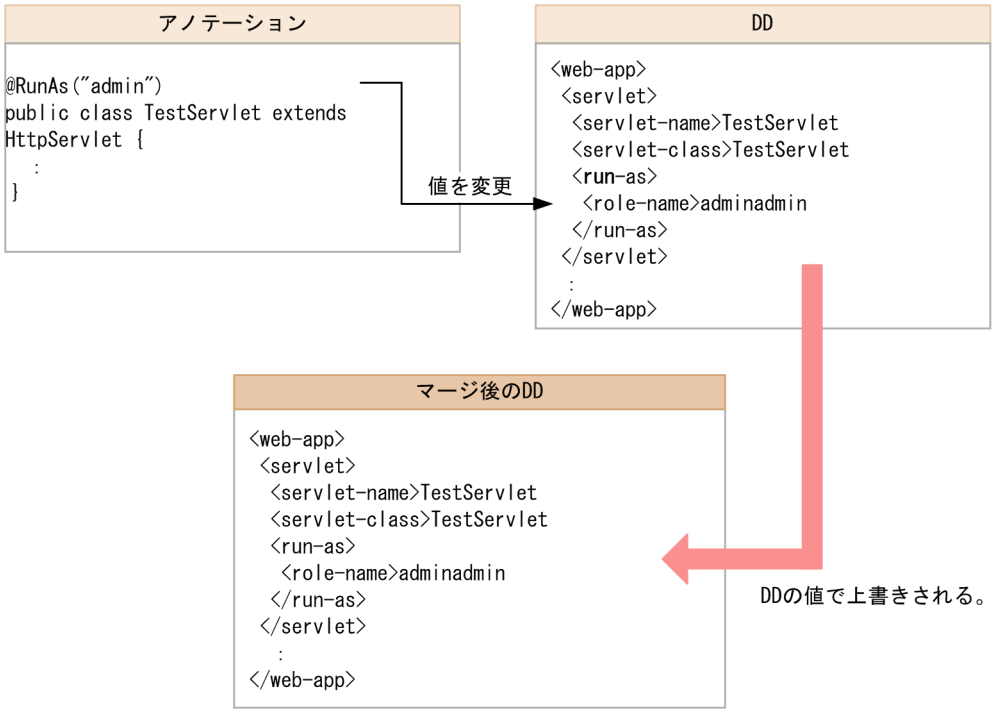
DD の定義でアノテーションの定義が上書きされます。

アノテーションに対応する要素が DD に定義されていない場合

アノテーションの定義が有効になります。

出現回数が 0 回または 1 回の要素の例を次の図に示します。キーがある場合の例です。

図 12-3　0 回または 1 回出現する要素の例

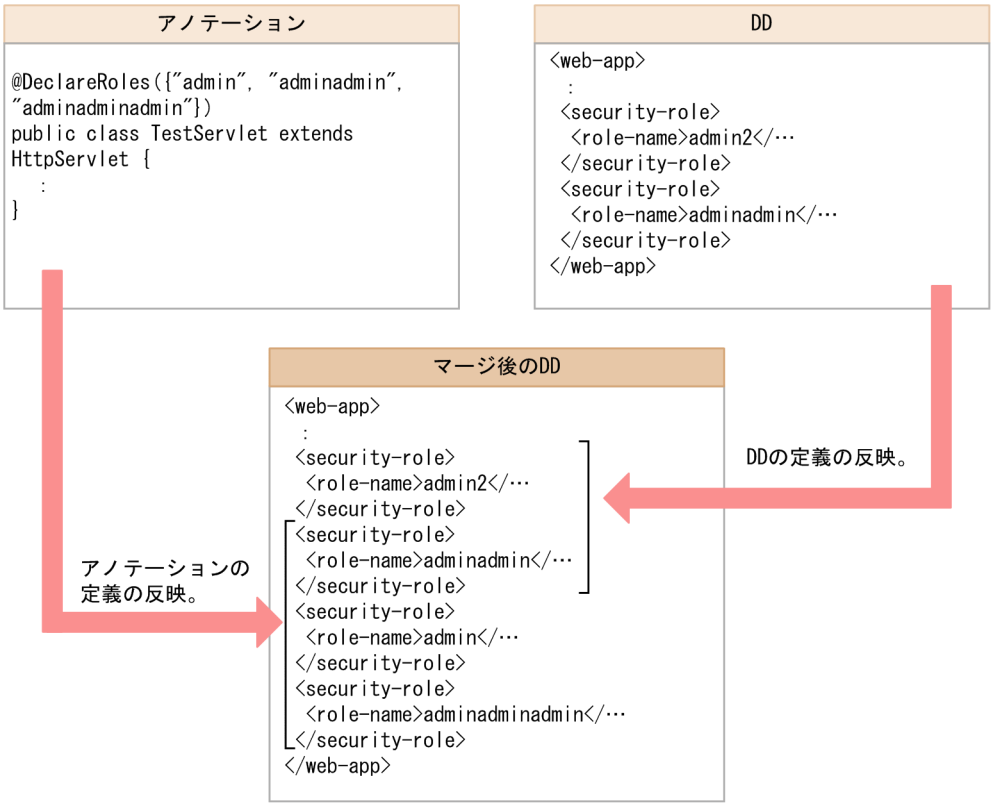


(2)　パターン 2（0 回以上または 1 回以上出現する要素でキーがない場合）

DD とアノテーションの両方の定義が有効になります。

キーがない場合の例を次の図に示します。「@DeclareRoles」で定義したセキュリティロールに加えて，DD で新しいセキュリティロールを定義します。

図 12-4　キーがない場合の例



この例では，アノテーションで「admin」「adminadmin」および「adminadminadmin」の三つのセキュリティロールを定義しています。これに加えて，DD で定義した「admin2」が追加されてマージ後の DD に反映されます。なお，「adminadmin」はアノテーションおよび DD の両方で定義されているセキュリティロールです。アプリケーションは，マージ後の DD が定義されたものとして動作します。

なお，このパターンに該当する場合，アノテーションの定義はサーバ管理コマンドで変更できません。アノテーションに対応する情報を属性ファイルで削除・変更して cjsetappprop コマンドを実行しても，そのあとで cjgetappprop コマンドによって取得した属性ファイルには，アノテーションで定義された情報が出力されます。

(3)　パターン 3（0 回以上または 1 回以上出現する要素で子要素が 0 回または 1 回出現する場合）

DD の要素内の子要素が 0 回または 1 回出現する場合は，キーの値によって上書きの方法が異なります。

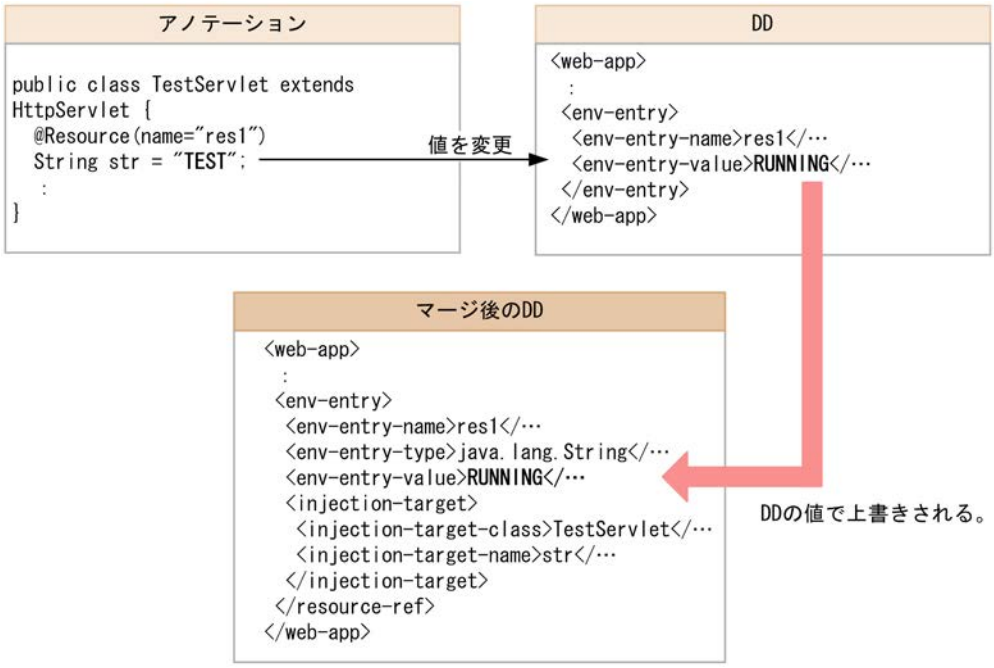
キーの値が DD とアノテーションで一致している場合

アノテーションの定義に DD の要素を上書きします。

キーの値が DD とアノテーションで一致しない場合

上書きされません。DD とアノテーションの両方の定義が有効になります。

図 12-5　子要素が 0 回または 1 回出現する例



この例では，「res1」の環境エントリの値をアノテーションで定義した「TEST」から DD で定義した「RUNNING」に変更します。アプリケーションは，マージ後の DD が定義されたものとして動作します。

(4)　パターン 4 (0 回以上または 1 回以上出現する要素で子要素が 0 回以上または 1 回以上出現する場合)

DD の要素内の子要素が 0 回以上または 1 回以上出現する場合は，キーの値によって上書きの方法が異なります。

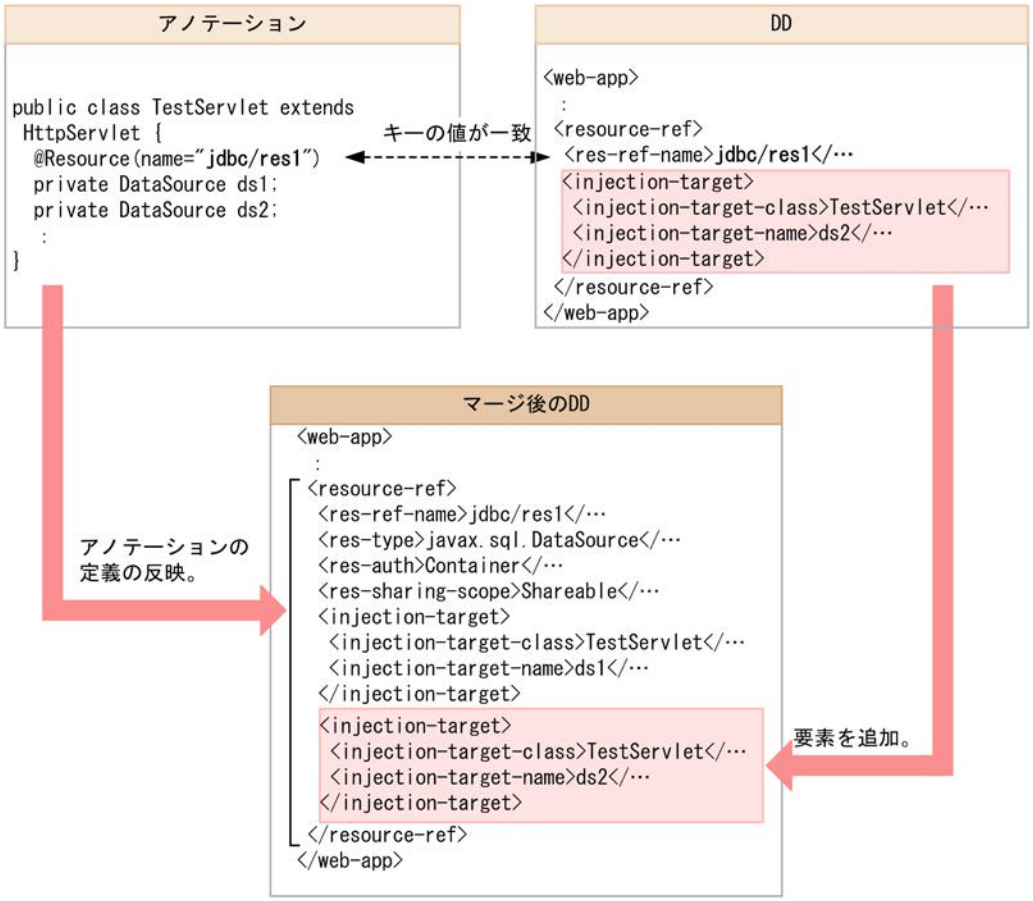
キーの値が DD とアノテーションで一致している場合

アノテーションの定義に DD の要素が追加されます。

キーの値が DD とアノテーションで一致しない場合

上書きされません。DD とアノテーションの両方の定義が有効になります。

図 12-6　子要素が 0 回以上または 1 回以上出現する例



この例では，アノテーションの定義に対して，DD で定義した要素「injection-target」が追加されます。アプリケーションは，マージ後の DD が定義されたものとして動作します。

12.6.3　サーバ管理コマンドを使用した定義の参照と更新

アノテーションで定義した要素は，サーバ管理コマンドの cjgetappprop コマンドまたは cjgetresprop コマンドを使用して参照できます。また，一部の定義は cjsetappprop コマンドまたは cjsetresprop コマンドを使用して更新できます。

(1)　アノテーションで定義した要素を参照・更新した場合の動作

サーバ管理コマンドを使用してアノテーションで定義した要素を参照･更新した場合の動作について次の表に示します。動作は，DD がある場合とない場合で異なります。

表 12-17　サーバ管理コマンドを使用してアノテーションで定義した要素を参照・更新した場合の動作

要素の種類		cjgetappprop または cjgetresprop による参照	cjsetappprop または cjsetresprop による更新
DD がある場合	DD で定義した要素	DD で定義した内容が出力されます。	DD の内容が更新されます。
	アノテーションで定義した要素	アノテーションで定義した内容が出力されます。	更新はできません。

要素の種類		cjgetappprop または cjgetresprop による参照	cjsetappprop または cjsetresprop による更新
DD がある場合	アノテーションの定義を DD で上書きした要素	アノテーションと DD の定義をマージした結果が出力されます。	「(2) サーバ管理コマンドによる定義の更新」を参照してください。
DD がない場合	アノテーションで定義した要素	アノテーションで定義した内容が出力されます。	更新はできません。

(2)　サーバ管理コマンドによる定義の更新

cjsetappprop コマンドまたは cjsetresprop コマンドを使用してアノテーションで定義した属性を上書き更新できるかどうかは，更新する情報がどこに定義されているかによって異なります。

定義されている場所と上書き可否の対応を次の表に示します。

表 12–18　定義されている場所と上書き可否の対応

定義されている場所	上書き可否
DD だけに定義されている要素	可
アノテーションだけに定義されている要素	不可
アノテーションの定義を DD で上書きされている要素	可※

注※　更新後にアプリケーションをエクスポートした場合，エクスポートしたアプリケーションに含まれる WAR ファイル内の DD（web.xml）に，アノテーションと DD で定義した内容がマージされて出力されます。展開ディレクトリ形式のアプリケーションの場合は，<WAR ディレクトリ>/WEB-INF/web.xml に，アノテーションと DD で定義した内容がマージされて出力されます。

● 注意事項

アノテーションの定義を DD で上書きした要素を，cjsetappprop コマンドまたは cjsetresprop コマンドを使用して上書き更新した場合，アプリケーションディレクトリ内の DD には DD とアノテーションの情報がマージされて出力されます。この場合，アノテーションで定義した内容が DD に反映されるため，それ以降にクラスファイル内のアノテーションを変更したときは，DD に残っている定義が優先されて変更したアノテーションの定義は有効になりません。

このような場合，DD に反映されているアノテーションの定義を削除して，アプリケーションを再インポートすることでアノテーションの定義が有効になります。また，属性ファイルで定義を追加，変更した場合は，アプリケーションを再インポートする必要はありません。

12.7　アノテーション使用時の注意事項

アノテーションを使用する場合の注意事項を示します。

(1)　インポート時の注意

- クラスに宣言されたアノテーションが，外部のアーカイブファイルのクラスを参照していると，それらを単体でインポートできません。
  - EJB（ejb-jar.xml のバージョン 2.1 以降または ejb-jar.xml なし）
  - Servlet（web.xml のバージョン 2.4 以降または web.xml なし）
  - ライブラリ JAR

  外部参照は，次のどれかの方法で実施してください。
  - 参照するクラスをすべて EAR に含めてインポートする。
  - 参照するクラスをすべてアプリケーションディレクトリに含めてインポートする。
  - 外部参照しているクラスが含まれる JAR ファイルをコンテナ拡張ライブラリとして設定する。また，インポートするプロセスのクラスパスにも設定する。

  操作方法ごとの設定例を示します。

  例 1：サーバ管理コマンドによって操作する場合

  J2EE サーバ用オプション定義ファイルのキー（add.class.path）に参照している JAR ファイルを指定します。また，サーバ管理コマンド用オプション定義ファイルのキー（USRCONF_JVM_CLASSPATH）に参照している JAR ファイルを指定します。

  例 2：Management Server によって操作する場合

  運用管理ポータルによる論理 J2EE サーバの J2EE コンテナの設定で，拡張パラメタに"add.class.path=<参照している JAR ファイル>"を指定します。

  また，adminagentuser.cfg の add.class.path キーに参照している JAR ファイルを指定します。さらに，サーバ管理コマンド用オプション定義ファイルのキー（USRCONF_JVM_CLASSPATH）に参照している JAR ファイルを指定します。
- 次の場合はサーバ管理コマンド実行時にエラーになります。
  - @Resource の要素 type()にアプリケーションサーバでサポートしていない Java Type を設定した場合。
  - EJB-JAR に ejb-jar.xml がなく，かつ@Stateless または@Stateful アノテーションを宣言したクラスが一つもない場合。
  - DD のスキーマ定義の出現回数を上回るアノテーションを宣言していた場合。
- WAR ファイルから EJB-JAR ファイルを参照している場合，WAR ファイルと EJB-JAR ファイルの両方を含んだアプリケーションにしてからインポートしてください。ただし，参照するインタフェースを WAR ファイルに含めている場合は，WAR ファイルを個別にインポートすることもできます。
- アノテーションの属性にライブラリ JAR 内のクラスを記述している場合，ライブラリ JAR を含んだ形のアプリケーションとしてインポートしてください。
- ほかの EJB-JAR を参照する EJB-JAR が存在する場合，両方の EJB-JAR を含んだ形のアプリケーションとしてインポートしてください。ただし，参照する側の EJB-JAR に，参照される側のインタフェースを含めている場合は，EJB-JAR ごとにインポートすることもできます。

(2)　@Resource の mapped-name 属性指定時の注意

アプリケーションサーバでは，@Resource で指定された mapped-name 属性だけ処理対象とします。@Resource に指定した mapped-name 属性は，属性ファイルの<resource-ref>タグ下の<linked-to>タグ，<resource-env-ref>タグ下の<linked-to>タグ，および<resource-env-ref>タグ下の<linked-queue>タグに対応するものとして処理します。ただし，属性ファイルの<linked-to>タグおよび<linked-queue>タグと mapped-name 属性が両方指定されていた場合は，<linked-to>タグおよび<linked-queue>タグの指定値を優先します。

(3)　EJB のリンク解決に関する注意

- @EJB は DD の<ejb-ref>タグに対応するアノテーションです。同一アプリケーションの範囲内で EJB のリンクを解決します。@EJB に beanName 属性が指定されている場合，対応する EJB 名を持つ EJB を検索してリンクを解決します。EJB 名とは ejb-jar.xml の場合，<session>タグ，<entity>タグ，および<message-driven>タグ下の<ejb-name>タグを指します。また，アノテーションであれば @Stateless および@Stateful の name 属性で指定されたものを指します。
- @EJB に beanName 属性が指定されていない場合，@EJB の beanInterface 属性と適合する型の EJB を検索してリンクを解決します。@EJB の beanInterface 属性と適合する型の EJB を次に示します。
  - @EJB の beanInterface 属性で指定したものと同じクラス型のホームインタフェースを持つ EJB
  - @EJB の beanInterface 属性で指定したものと同じクラス型のビジネスインタフェースを持つ EJB

  一つのビジネスインタフェースを複数の EJB が実装しているようなケースでは，@EJB および@EJBs に適合する EJB が複数存在することがあります。参照先を一つに絞り込めない場合はリンクを解決しません。

(4)　@RemoteHome または@LocalHome 使用時の注意

アプリケーションサーバでは，@RemoteHome または@LocalHome で指定したホームインタフェースの create メソッドの戻り値の型となるインタフェースは，コンポーネントインタフェースと見なされます。ホームインタフェースとビジネスインタフェースの組み合わせでは使用できません。

(5)　@PostConstruct または@PreDestroy 使用時の注意

- @PostConstruct または@PreDestroy の定義が Java EE 仕様に沿っているかどうか，Web アプリケーション開始時にチェックされます。Java EE 仕様に沿わない定義があった場合は，アノテーションに関するメッセージが出力され，Web アプリケーションの開始が中断されます。
- web.xml で<post-construct>または<pre-destroy>タグを指定している場合，Web アプリケーション開始時にメソッドがチェックされます。指定されたクラス，メソッドが見つからない場合は，KDJE39332-E が出力されエラーとなります。
- 一つのクラス内で二つ以上のメソッドに@PostConstruct または@PreDestroy を指定した場合，Web アプリケーションの開始時に KDJE39327-E が出力されてエラーとなります。
- アプリケーションサーバでは，@PostConstruct または@PreDestroy を指定したメソッド定義をする場合，次の表に示す仕様に注意してください。@PostConstruct または@PreDestroy を指定したメソッド定義をする場合の仕様について，次の表に示します。

表 12-19　@PostConstruct または@PreDestroy を指定したメソッド定義をする場合の仕様

メソッド定義の仕様	Java EE 仕様との違い	説明
引数を持てません。	－	引数を持つメソッドの場合，Web アプリケーション開始時に KDJE39328-E のメッセージが出力され，エラーとなります。
検査例外をスローできます。非検査例外と同様に扱われます。	Java EE 仕様では，検査例外をスローできません。	throws の定義があるメソッドの場合，Web アプリケーション開始時に KDJE39329-W のメッセージが出力されます。
インスタンスメソッドおよびクラス（static）メソッドで使用できます。	Java EE 仕様では，static メソッドでは指定できません。	static メソッドの場合，Web アプリケーション開始時に KDJE39330-W のメッセージが出力されます。
戻り値があるメソッドを使用できます。	Java EE 仕様では，戻り値は void に限定する必要があります。	戻り値があるメソッドの場合，Web アプリケーション開始時に KDJE39331-W のメッセージが出力されます。
public，protected，パッケージプライベート，private のどれかを指定できます。	－	－
final として宣言できます。	－	－

（凡例）－：なし

- @PostConstruct アノテーションを指定したメソッドで例外が発生した場合，メッセージ KDJE53906-E が出力され，例外の発生したインスタンスが破棄されます。
- Web アプリケーション開始後に初期化するクラスの場合，初期化に失敗してもクラスを使用するたびに初期化が試行されます。例えば，サーブレットへの初回リクエスト時に，@PostConstruct を指定したメソッドで例外が発生した場合，次に同じサーブレットへアクセスする際にもサーブレットの初期化が試行されます。
- @PreDestroy アノテーションを指定したメソッドで例外が発生した場合，メッセージ KDJE53907-W が出力され，Web コンテナからのオブジェクトの破棄処理が継続されます。

(6)　そのほかの注意

- @Timeout を EJB の複数のメソッドに指定した場合，どのメソッドが対象となるかは特定できません。
- インターセプタとして指定したクラスのアノテーション情報が取得できない場合は，属性ファイル設定時にエラーになります。アノテーション情報を取得するには，対象のクラスとそのスーパークラス・インタフェース，およびそれらのクラス上で宣言されたすべてのフィールド・メソッドについて情報が取得できるようにクラスパスを指定する必要があります。
- @HandlesTypes アノテーションはアノテーション参照抑止機能の対象になりません。アノテーション参照抑止機能が有効な場合でも処理されます。

13 J2EE アプリケーションの形式とデプロイ

この章では，J2EE サーバで実行できる J2EE アプリケーションの形式と，形式ごとに実行できる J2EE アプリケーションのデプロイ，アンデプロイおよび入れ替え機能について説明します。

13.1　この章の構成

J2EE アプリケーションの形式とデプロイの機能と参照先を次の表に示します。

表 13-1　J2EE アプリケーションの形式およびデプロイ機能と参照先

機能	参照先
実行できる J2EE アプリケーションの形式	13.2
アーカイブ形式の J2EE アプリケーション	13.3
展開ディレクトリ形式の J2EE アプリケーション	13.4
J2EE アプリケーションのデプロイとアンデプロイ	13.5
J2EE アプリケーションの入れ替え	13.6
J2EE アプリケーションのリデプロイ	13.7
J2EE アプリケーションの更新検知とリロード	13.8
WAR アプリケーション	13.9

13.2　実行できる J2EE アプリケーションの形式

J2EE サーバ上で実行できるアプリケーションを，J2EE アプリケーションといいます。WAR 形式の Web アプリケーション，または EJB-JAR 形式の EJB アプリケーションを J2EE サーバで実行するためには，J2EE アプリケーションとしてデプロイする必要があります。

J2EE サーバ上で実行できる J2EE アプリケーションは，次に示す 2 種類です。

- **アーカイブ形式の J2EE アプリケーション**

  EJB やサーブレットなどのアプリケーションの実体を J2EE サーバの作業ディレクトリに持つ J2EE アプリケーションです。

- **展開ディレクトリ形式の J2EE アプリケーション**

  EJB やサーブレットなどのアプリケーションの実体を，J2EE サーバの外部にある一定のルールに従ったファイル／ディレクトリに持つ J2EE アプリケーションです。

なお，J2EE サーバでは WAR ファイルまたは WAR ディレクトリを指定してインポートできます。インポートしたアプリケーションを WAR アプリケーションといい，J2EE アプリケーションとしてデプロイできます。WAR アプリケーションについては，「13.9　WAR アプリケーション」を参照してください。

13.3 以降で，それぞれの形式の J2EE アプリケーションのデプロイについて説明します。また，J2EE アプリケーションの実行機能として，J2EE アプリケーションのリデプロイ，J2EE アプリケーションの更新検知とリロードについても説明します。

! 注意事項

UNIX の場合，LANG 環境変数の設定が，J2EE アプリケーションの開発環境と実行環境で異なると，J2EE サーバの起動時に，java.text.ParseException: Unparseable date 例外がスローされることがあります。そのあとの J2EE サーバの動作には影響ありません。

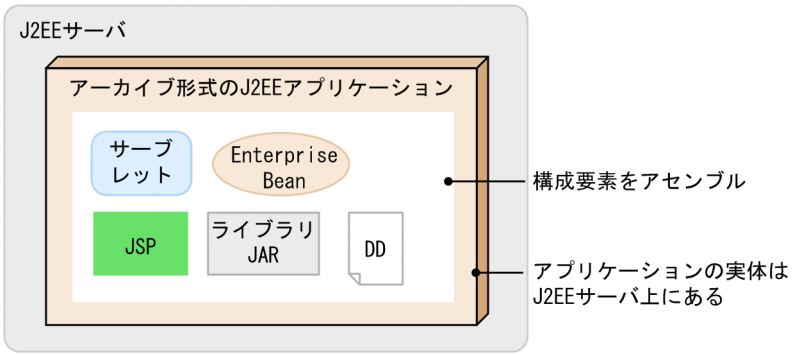
なお，システムの運用を開始したあとで J2EE アプリケーションの運用中に使用する機能については，マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「5.　J2EE アプリケーションの運用」を参照してください。

13.3　アーカイブ形式の J2EE アプリケーション

この節では，アーカイブ形式による J2EE アプリケーションのデプロイについて説明します。

J2EE サーバ上で実行できるアプリケーションの形式の一つに，**アーカイブ形式の J2EE アプリケーション**があります。アーカイブ形式の J2EE アプリケーションは，EJB やサーブレットなどのアプリケーションの実体を J2EE サーバの作業ディレクトリに持つアプリケーションです。アーカイブ形式の J2EE アプリケーションを次の図に示します。

図 13-1　アーカイブ形式の J2EE アプリケーション



アーカイブ形式の J2EE アプリケーションでは，アプリケーションの実体は J2EE サーバ上にあり，J2EE サーバの作業ディレクトリ下にアプリケーションの構成要素を保持します。

通常，アーカイブ形式の J2EE アプリケーションを入れ替える場合には，J2EE サーバ上で動作している J2EE アプリケーションの停止，新しい J2EE アプリケーションのインポート，デプロイなどの作業が発生します。このとき，リデプロイ機能を使用すると，通常の J2EE アプリケーションの入れ替えに比べて，少ない手順で入れ替えができるようになります。

リデプロイ機能を使用した J2EE アプリケーションの入れ替えでは，J2EE アプリケーションの属性情報を引き継ぐことができ，J2EE サーバ上で動作している J2EE アプリケーションの停止，新しい J2EE アプリケーションのインポート，デプロイなどの作業が不要になります。リデプロイ機能については，「13.7 J2EE アプリケーションのリデプロイ」を参照してください。

! 注意事項

アプリケーションのバージョンが Java EE 5 以降の場合，EAR ファイル内のエントリ名にアプリケーションパッケージのルートよりも上位を指定することはできません。指定した場合は，J2EE アプリケーションのインポート時に KDJE42387-E のメッセージが出力され，エラーになります。

13.4　展開ディレクトリ形式の J2EE アプリケーション

この節では，展開ディレクトリ形式による J2EE アプリケーションのデプロイについて説明します。

この節の構成を次の表に示します。

表 13–2　この節の構成（展開ディレクトリ形式の J2EE アプリケーション）

分類	タイトル	参照先
解説	展開ディレクトリ形式の概要	13.4.1
	アプリケーションディレクトリの構成	13.4.2
設定	展開ディレクトリ形式の J2EE アプリケーションを使用するための設定（セキュリティの設定変更）	13.4.3
注意事項	展開ディレクトリ形式を使用する場合の注意事項	13.4.4

注　「実装」および「運用」について，この機能固有の説明はありません。

13.4.1　展開ディレクトリ形式の概要

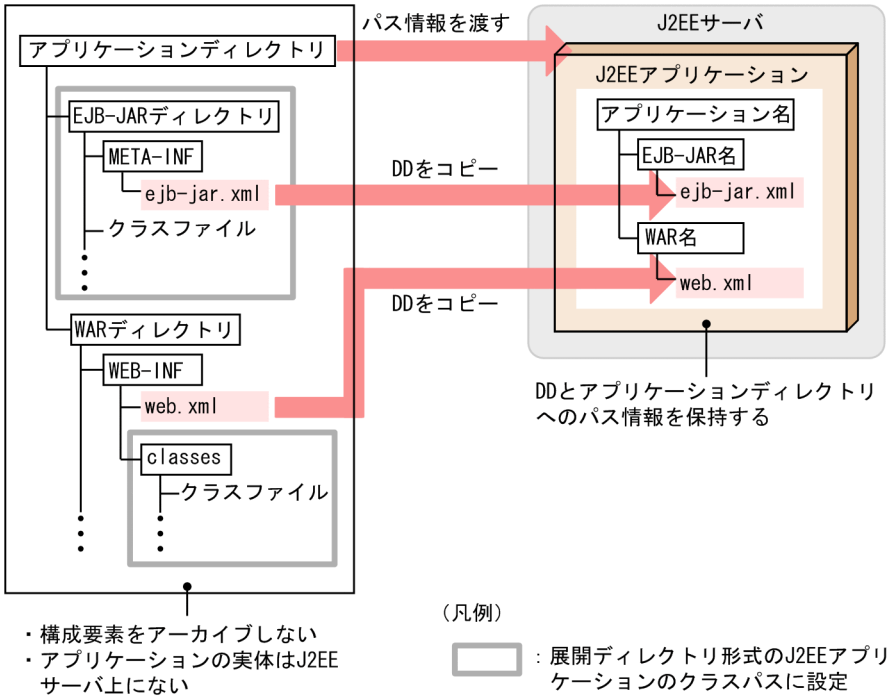
ここでは，展開ディレクトリ形式の概要，デプロイおよびアンデプロイの方法について説明します。

(1)　展開ディレクトリ形式の J2EE アプリケーションとは

J2EE サーバ上で実行できるアプリケーションの形式の一つに，**展開ディレクトリ形式の J2EE アプリケーション**があります。展開ディレクトリ形式の J2EE アプリケーションは，EJB やサーブレットなどのアプリケーションの実体を，J2EE サーバの外部にある一定のルールに従ったファイル／ディレクトリに持つアプリケーションです。**アプリケーションディレクトリ**というルートディレクトリを作成し，アプリケーションディレクトリ下に，EJB-JAR，Web アプリケーションなどの構成要素が格納されていれば，J2EE アプリケーションとして動作できます。アプリケーションディレクトリについては，「13.4.2　アプリケーションディレクトリの構成」を参照してください。

展開ディレクトリ形式の J2EE アプリケーションを次の図に示します。

図 13-2　展開ディレクトリ形式の J2EE アプリケーション



展開ディレクトリ形式の J2EE アプリケーションが保持するのは，DD とアプリケーションディレクトリへのパス情報です。J2EE サーバの作業ディレクトリ下に，EJB-JAR，Web アプリケーションなどの構成要素を保持しません。

展開ディレクトリ形式の J2EE アプリケーションの特長について説明します。

(a)　J2EE アプリケーションの入れ替えの容易化

展開ディレクトリ形式の J2EE アプリケーションでは，構成要素を EAR 形式にアーカイブする必要はありません。このため，アーカイブ形式の J2EE アプリケーションと比べて，少ない手順で J2EE アプリケーションの入れ替えができます。

また，リロード機能を使用することで，さらに少ない手順で J2EE アプリケーションを動的に入れ替えられます。リロード機能では，J2EE アプリケーションを構成するファイルの更新を検知し，更新した J2EE アプリケーションをリロードします。リロード機能を使用することで，J2EE サーバを再起動することなく，デプロイ済みのサーブレット，JSP や EJB-JAR を動的に入れ替えられるようになります。リロード機能については，「13.8　J2EE アプリケーションの更新検知とリロード」を参照してください。

(b)　複数の J2EE サーバでのアプリケーションディレクトリの共有

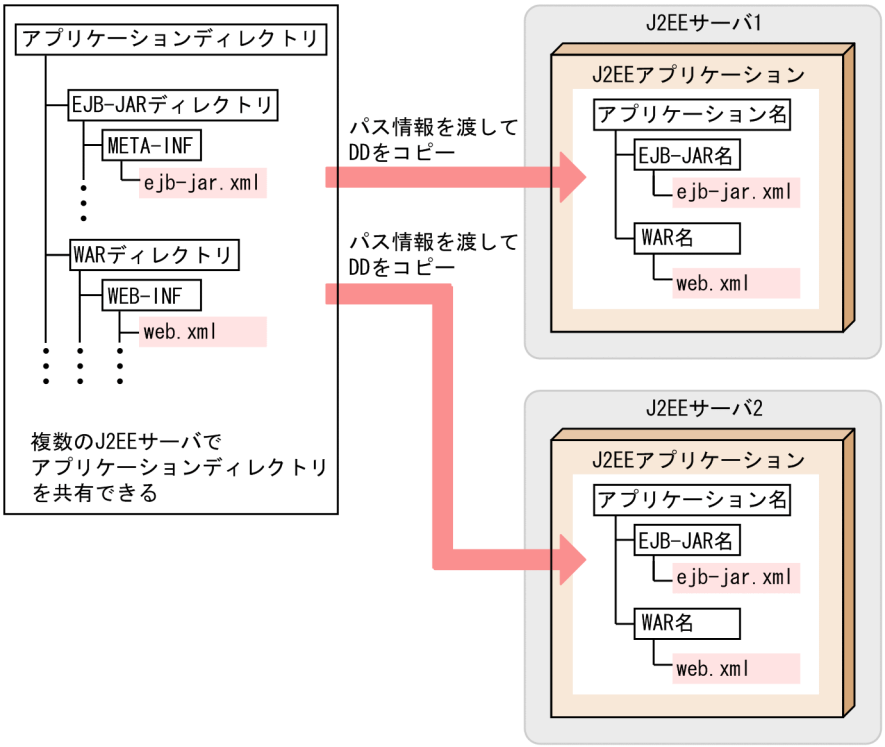
展開ディレクトリ形式の J2EE アプリケーションでは，複数の J2EE サーバでアプリケーションディレクトリを共有できます。

例えば，アーカイブ形式の J2EE アプリケーションを使用している場合に，同じ内容の J2EE アプリケーションを複数の J2EE サーバで実行させるときは，複数の J2EE サーバ上に，EAR ファイル／ZIP ファイルをそれぞれ配置する必要があります。また，インポートした J2EE アプリケーションを入れ替える場合には，J2EE サーバの数だけ，再インポートや再デプロイが必要になります。

しかし，展開ディレクトリ形式の J2EE アプリケーションでは，DD とアプリケーションディレクトリへのパス情報だけを保持するため，アプリケーションディレクトリ以下にある EJB-JAR，Web アプリケーションなどの構成要素を，複数の J2EE サーバで共有できます。

例えば，J2EE サーバをクラスタ構成で配置している場合にそれぞれの J2EE サーバで同じ J2EE アプリケーションを使用するときは，同一のディレクトリをアプリケーションディレクトリとして指定できます。同一のアプリケーションディレクトリを参照しているため，J2EE アプリケーションに変更が発生した場合には，そのアプリケーションディレクトリでクラスファイルなどを更新するだけで，すべての J2EE アプリケーションの入れ替えができるようになります。アプリケーションディレクトリの共有の例を次の図に示します。

図 13-3　アプリケーションディレクトリの共有の例



なお，J2EE アプリケーションを共有する場合には，それぞれの J2EE サーバから参照できる環境（共有ディスク装置など）にアプリケーションディレクトリを配置してください。

13.4.2　アプリケーションディレクトリの構成

アプリケーションディレクトリは，展開ディレクトリ形式の J2EE アプリケーションのルートディレクトリです。展開ディレクトリ形式の J2EE アプリケーションを作成する場合には，アプリケーションディレクトリを作成し，アプリケーションディレクトリ下に，EJB-JAR，Web アプリケーションなどの構成要素を格納します。アプリケーションディレクトリの構成を次の表に示します。

表 13-3　アプリケーションディレクトリの構成

ディレクトリ	ディレクトリの説明
<アプリケーションディレクトリ>	展開ディレクトリ形式の J2EE アプリケーションのルートディレクトリです。アプリケーションディレクトリの名称は任意です。

ディレクトリ			ディレクトリの説明
META-INF			J2EE アプリケーションの DD および cosminexus.xml の格納ディレクトリです。このディレクトリの直下に，application.xml および cosminexus.xml が格納されます。
<EJB-JAR ディレクトリ>			EJB アプリケーションのルートディレクトリです。
	META-INF		EJB-JAR の DD の格納ディレクトリです。このディレクトリの直下に，ejb-jar.xml が格納されます。
	<パッケージ名>		EJB-JAR のクラスファイルやプロパティファイルを格納するディレクトリです。 （例） MyEJB.class MyEJBHome.class MyEJBRemote.class MyResource.properties
<WAR ディレクトリ>			Web アプリケーションのルートディレクトリです。 なお，このディレクトリの直下に，JSP ファイルが格納されます。 （例） index.jsp
	WEB-INF		Web アプリケーションの DD の格納ディレクトリです。このディレクトリの直下に，web.xml が格納されます。
		classes	サーブレットクラスファイルやプロパティファイルを格納するディレクトリです。 （例） MyServlet.class
		lib	タグライブラリなどの JAR ファイルを格納するディレクトリです。 （例） MyTagLibrary.jar
RAR ファイル※			J2EE アプリケーションで使用するリソースアダプタです。
<ライブラリディレクトリ>			ライブラリ JAR を格納するディレクトリです。アプリケーションのバージョンが Java EE 5 以降の場合，このディレクトリの直下にライブラリ JAR が格納されます。

注※　リソースアダプタ（RAR ファイル）を J2EE アプリケーションに含めて使用する場合に配置します。RAR ファイルは，アーカイブファイルのまま格納します。リソースアダプタを J2EE リソースアダプタとしてデプロイする場合は，アプリケーションディレクトリにリソースアダプタを格納する必要はありません。

アプリケーションディレクトリの留意事項を次に示します。

- <EJB-JAR ディレクトリ>，<WAR ディレクトリ>，および RAR ファイルを配置する場合の留意点を application.xml がある場合と application.xml がない場合とに分けて説明します。
  - application.xml がある場合には，それぞれ application.xml の<module>タグ以下に記述した相対パスに従って作成してください。また，EJB-JAR ディレクトリと WAR ディレクトリの名称も application.xml の記述と合わせてください。application.xml の<module>タグ以下に記述した EAR 上の相対パス（EJB-JAR ファイル名，または WAR ファイル名）から拡張子（「.jar」または「.war」）を除いたものが，EJB-JAR ディレクトリと WAR ディレクトリの名称になります。

  - application.xml がない場合には，ディレクトリ名の最後は「_jar」または「_war」となります。
- application.xml の<module>/<ejb>，<module>/<web>，<module>/<connector>タグの値に（../）を含む相対パスは記述できません。記述した場合エラーになります。
- application.xml の<module>タグ以下で，EJB-JAR と WAR のモジュールが，「.jar」または「.war」以外の拡張子で宣言されていた場合，アプリケーションのインポートに失敗します。
- ライブラリ JAR の扱われ方は，アプリケーションのバージョンによって異なります。
  - アプリケーションのバージョンが J2EE1.4 以前の場合

    アプリケーションディレクトリ以下の JAR ファイルのうち，application.xml の<module>タグ以下で定義していない，拡張子が小文字（.jar）の JAR ファイルは，ライブラリ JAR として扱われます。
  - アプリケーションのバージョンが Java EE 5 以降の場合

    サブディレクトリを含まないライブラリディレクトリの直下，またはアプリケーションディレクトリの直下に置かれていて，application.xml の<module>タグに定義されていない JAR ファイルがライブラリ JAR として扱われます。ライブラリディレクトリには（../）を含むパスの記述はできません。記述した場合はアプリケーションをインポートするタイミングでエラーになります。
- アプリケーションディレクトリにない JAR を指定する場合は，参照ライブラリを使用します。
- アプリケーションディレクトリを変更する場合には，「(2)　アプリケーションディレクトリの変更」に示す規則に従ってください。

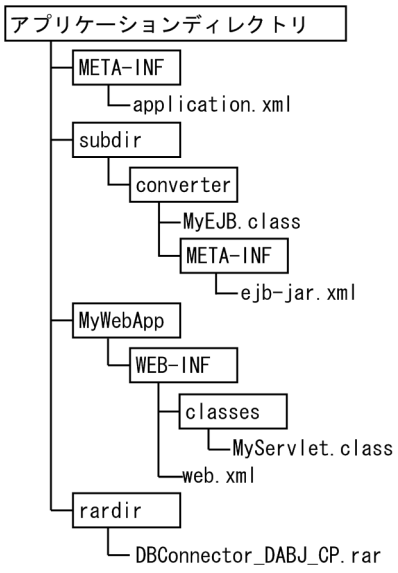
(1)　アプリケーションディレクトリの作成例

アプリケーションディレクトリの作成例を次に示します。

ここでは，EJB-JAR ディレクトリが「subdir/converter」，WAR ディレクトリが「MyWebApp」，RAR ファイルが「rardir/DBConnector_DABJ_CP.rar」の場合のアプリケーションディレクトリの構成例，application.xml の記述例を示します。

- **アプリケーションディレクトリの構成例**

  この例でのアプリケーションディレクトリの構成を次に示します。



- **application.xml の記述例**

  この例での application.xml を次に示します。

```
<?xml version="1.0" encoding="Shift_JIS" ?>
<application xmlns="http://java.sun.com/xml/ns/j2ee"
             version="1.4"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                         http://java.sun.com/xml/ns/j2ee/application_1_4.xsd">
  <display-name>converter</display-name>
  <module>
    <ejb>subdir/converter.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>MyWebApp.war</web-uri>
      <context-root>/</context-root>
    </web>
  </module>
  <module>
    <connector>rardir/DBConnector_DABJ_CP.rar</connector>
  </module>
</application>
```

(2)　アプリケーションディレクトリの変更

アプリケーションディレクトリを変更する場合，変更の対象となるディレクトリやファイルの種類と操作の内容によって，変更できなかったり，変更後にアプリケーションの再インポートが必要になったりします。次に，アプリケーションディレクトリを変更した場合に必要となる作業について説明します。

（例）

- ディレクトリの削除／名称変更

  アプリケーションディレクトリ，EJB-JAR ディレクトリ，および WAR ディレクトリは削除できません。また，ディレクトリ名は変更できません。削除した場合や名称を変更した場合には，アプリケーションを再インポートする必要があります。

- JAR ファイルの追加／変更／削除

  ライブラリ JAR の追加，削除，およびファイル名の変更はできません。ライブラリ JAR（application.xml で宣言していない JAR ファイル）を追加しても，その JAR ファイルは読み込まれません。アプリケーションのインポート時に存在したライブラリ JAR のファイル名を変更した場合や，ライブラリ JAR を削除した場合には，アプリケーションを再インポートする必要があります。

  なお，WAR ディレクトリ/WEB-INF/lib ディレクトリでは，JAR ファイルの追加／変更／削除はできます。

- DD の変更

  ユーザがアプリケーションディレクトリに存在する DD を変更しても，J2EE サーバは更新を検知しません。DD の定義を変更する場合，開発環境で定義を変更してからアプリケーションをインポートし直してください。または，サーバ管理コマンドを使用してください。ただし，クラスファイルに記述されたアノテーションまたは cosminexus.xml は，次回 J2EE アプリケーションを開始したときのクラスファイルの再読み込みで，更新を反映させることができます。

  なお，サーバ管理コマンド実行時にアプリケーションディレクトリの構成がチェックされます。

アプリケーションディレクトリの変更可否を次の表に示します。

表 13-4　アプリケーションディレクトリの変更可否

対象	ディレクトリおよびファイル	ディレクトリおよびファイルへの操作	アプリケーションの状態		説明
			停止	開始	
J2EE アプリケーション	アプリケーションディレクトリ	追加	－	－	－
		内容変更	－	－	－

対象	ディレクトリおよびファイル	ディレクトリおよびファイルへの操作	アプリケーションの状態		説明
			停止	開始	
J2EE アプリケーション	アプリケーションディレクトリ	名称変更	△	×	アプリケーションの構成変更はできない。
		削除	△	×	アプリケーションの構成変更はできない。
	application.xml	追加	−	−	−
		内容変更	○	×	DD の直接編集はできない。DD の定義の更新にはサーバ管理コマンドを使用する。※1
		名称変更	−	−	−
		削除	△	×	−
	cosminexus.xml	追加	◎	○	−
		内容変更	◎	○	−
		名称変更	−	−	−
		削除	◎	○	−
EJB アプリケーション	EJB-JAR ディレクトリ	追加	△	−	−
		内容変更	−	−	−
		名称変更	△	×	アプリケーションの構成変更はできない。
		削除	△	×	アプリケーションの構成変更はできない。
	EJB を構成するクラスファイル	追加	◎	×	クラスを追加したあとで，サーバ管理コマンドで属性を編集する必要がある。
		内容変更	◎	×	DD にメソッド情報（container-transaction や method-permission）が記述されている場合には，DD の情報と同期を取って修正する必要がある。
		名称変更	△	×	クラスファイルの内容を変更しないでクラスファイル名を変更しようとした場合，次回クラスローダ時にエラーになる。
		削除	◎	×	クラスを削除したあとで，サーバ管理コマンドで属性を編集する必要がある。

対象	ディレクトリおよびファイル	ディレクトリおよびファイルへの操作	アプリケーションの状態		説明
			停止	開始	
EJB アプリケーション	ほかのクラスファイル	追加	◎	×	－
		内容変更	◎	×	－
		名称変更	△	×	クラスファイルの内容を変更しないでクラスファイル名を変更しようとした場合，次回クラスローダ時にエラーになる。
		削除	◎	×	－
	ejb-jar.xml	追加	－	－	－
		内容変更	○	×	DD の直接編集はできない。DD の定義の更新にはサーバ管理コマンドを使用する。※1
		名称変更	－	－	－
		削除	△	×	－
Web アプリケーション	WAR ディレクトリ	追加	△	－	－
		内容変更	－	－	－
		名称変更	△	×	アプリケーションの構成変更はできない。
		削除	△	×	アプリケーションの構成変更はできない。
	WEB-INF/classes 下のクラスファイルやプロパティファイル	追加	◎	×	－
		内容変更	◎	×	－
		名称変更	－	－	クラスファイルの内容を変更しないでクラスファイル名を変更しようとした場合，次回クラスローダ時にエラーになる。
		削除	◎	×	－
	WEB-INF/lib 下の JAR ファイル	追加	◎	×	－
		内容変更	◎	×	－
		名称変更	◎	×	－
		削除	◎	×	－
	web.xml	追加	－	－	－
		内容変更	○	×	DD の直接編集はできない。DD の定義の更新に

対象	ディレクトリおよびファイル	ディレクトリおよびファイルへの操作	アプリケーションの状態		説明
			停止	開始	
Web アプリケーション	web.xml	内容変更	○	×	はサーバ管理コマンドを使用する。※1
		名称変更	－	－	－
		削除	△	×	－
	JSP ファイル，タグファイル，静的コンテンツ（HTML や JavaScript など），JSP ファイルまたはタグファイルが依存するファイル※2，JSP コンパイル結果	追加	◎	×	－
		内容変更	◎	×	－
		名称変更	－	－	－
		削除	◎	×	－
ライブラリ	ライブラリ JAR	追加	△	×	アプリケーションの構成変更はできない。ライブラリ JAR を追加しても無視される。
		内容変更	◎	×	－
		名称変更	△	×	アプリケーションの構成変更はできない。
		削除	△	×	アプリケーションの構成変更はできない。
	参照ライブラリ	追加	◎	×	サーバ管理コマンド（cjsetappprop）で定義を変更する。
		内容変更	◎	×	－
		名称変更	－	－	サーバ管理コマンド（cjsetappprop）で定義を変更する。開始状態では変更できない。
		削除	○	×	サーバ管理コマンド（cjsetappprop）で定義を変更する。開始状態では削除できない。
	参照ライブラリのディレクトリ	追加	◎	×	－
		内容変更	◎	×	－
		名称変更	◎	×	－
		削除	◎	×	－

対象	ディレクトリおよびファイル	ディレクトリおよびファイルへの操作	アプリケーションの状態		説明
			停止	開始	
リソースアダプタ	RAR ファイル	追加	△	×	－
		内容変更	○	×	RAR ファイルを直接変更できない。サーバ管理コマンドを使用してプロパティを変更する。
		名称変更	△	×	－
		削除	△	×	－

（凡例）

「アプリケーションの状態」が「停止」の場合

アプリケーションが停止状態の場合，ディレクトリおよびファイルの種類や操作の内容によって，アプリケーションの再インポートの要否が異なります。

◎：再インポートは不要（次回アプリケーション開始時に操作が反映される）。

○：条件を満たす場合には，再インポートは不要。条件は「説明」を参照。

△：再インポートが必要。

－：該当しない。

「アプリケーションの状態」が「開始」の場合

アプリケーションが開始状態でリロード機能が無効の場合，ディレクトリおよびファイルの種類や操作の内容によって，操作を許可するかどうかが異なります。

○：操作できる。

×：操作できない。

－：該当しない。

注※1

DD を変更しても J2EE サーバは更新を検知しないで，操作は無視されます。

次に示すコマンドを次回，実行したときに上書きされます。

cjsetappprop

cjaddapp（-type filter オプション指定時）

cjdeleteapp（-type filter オプション指定時）

cjrenameapp

cjstartapp

注※2

依存するファイルは，JSP ファイルまたはタグファイルの include ディレクティブでインクルードされるファイルや，web.xml の<include-prelude>または<include-coda>でインクルードされるファイルのことです。

13.4.3　展開ディレクトリ形式の J2EE アプリケーションを使用するための設定（セキュリティの設定変更）

ここでは，展開ディレクトリ形式の J2EE アプリケーションを使用するために必要な設定について説明します。

展開ディレクトリ形式の J2EE アプリケーションを使用する場合には，次のどちらかの方法でセキュリティの設定を変更することをお勧めします。

- SecurityManager の解除

- セキュリティポリシーの設定変更

それぞれの方法について説明します。

(1)　SecurityManager を解除する場合

J2EE サーバを起動するときに，cjstartsv コマンドに-nosecurity オプションを指定し，SecurityManager を解除します。

```
# cjstartsv 〈サーバ名称〉 -nosecurity
```

! 注意事項

SecurityManager を解除することで，リソースアクセス時の権限チェックのオーバーヘッドを削減できますが，J2EE アプリケーションは任意のリソースへのアクセスができるようになるため，セキュリティが低下します。

(2)　セキュリティポリシーの設定を変更する場合

アプリケーションディレクトリ以下の JAR ファイルやクラスファイルに，リソースに対するアクセス権を与えるために，server.policy を次のように編集します。

```
grant codeBase "file:/D:/MyApplicationDir/Web.war/-"{
    permission java.lang.RuntimePermission "loadLibrary.*";
    permission java.lang.RuntimePermission "queuePrintJob";
    permission java.net.SocketPermission "*", "connect";
    permission java.io.FilePermission "<<ALL FILES>>", "read, write";
    permission java.util.PropertyPermission "*", "read";
    permission javax.security.auth.AuthPermission "getSubject";
    permission javax.security.auth.AuthPermission "createLoginContext.*";
};
```

server.policy は，Smart Composer 機能のコマンドでシステムを構築したあとに設定してください。

13.4.4　展開ディレクトリ形式を使用する場合の注意事項

展開ディレクトリ形式を使用する場合の注意事項を次に示します。

- 展開ディレクトリ形式の J2EE アプリケーションでは，アーカイブ形式の J2EE アプリケーションと比べて，次の作業ができません。
  - EJB-JAR／WAR／RAR の追加
  - EJB-JAR／WAR／RAR の削除
  - ライブラリ JAR の追加
  - ライブラリ JAR の削除

  展開ディレクトリ形式の J2EE アプリケーションでこれらの作業を実施したい場合には，次の手順で実施してください。

  1. J2EE サーバから展開ディレクトリ形式の J2EE アプリケーションを削除します。
  2. アプリケーションディレクトリを修正します。
  3. 修正した展開ディレクトリ形式の J2EE アプリケーションをインポートします。
- 展開ディレクトリ形式の J2EE アプリケーションを使用する場合には，J2EE アプリケーションが任意のリソースへアクセスできるように，セキュリティの設定を変更することをお勧めします。セキュリティの設定の変更については，「13.4.3　展開ディレクトリ形式の J2EE アプリケーションを使用するための設定（セキュリティの設定変更)」を参照してください。

- 展開ディレクトリ形式の J2EE アプリケーションをインポートする場合に，アプリケーションディレクトリのパスとして UNC 名を含むパスは指定できません。UNC 名を含むパスを指定した場合には，コマンドの実行エラーになります。
- J2EE サーバの起動処理中やサーバ管理コマンドの実行中にアプリケーションディレクトリ以下のファイルおよびディレクトリを追加／削除／上書きしないでください。
- application.xml に<alt-dd>タグが指定されている J2EE アプリケーションは，展開ディレクトリ形式で使用できません。展開ディレクトリ形式の J2EE アプリケーションをインポートする場合，または EAR ファイルや，J2EE サーバからエクスポートした ZIP ファイルを展開ディレクトリ形式でインポートする場合に，J2EE アプリケーションの application.xml に<alt-dd>タグが指定されているときは，コマンドの実行エラーになります。
- J2EE サーバ内で，複数の J2EE アプリケーションが同一のディレクトリをアプリケーションディレクトリとして指定することはできません。展開ディレクトリ形式の J2EE アプリケーションをインポートする場合，または EAR ファイルや，J2EE サーバからエクスポートした ZIP ファイルを展開ディレクトリ形式でインポートする場合に，指定したディレクトリがすでにほかの J2EE アプリケーションのアプリケーションディレクトリであるときは，コマンドの実行エラーになります。
- 次のディレクトリをアプリケーションディレクトリまたは WAR ディレクトリとして持つ J2EE アプリケーションが，すでに J2EE サーバ内に存在する場合，インポートでコマンドの実行エラーになります。
  - cjimportapp コマンドの-a オプション，または-d オプションに指定したディレクトリの上位にあるディレクトリ
  - cjimportapp コマンドの-a オプション，または-d オプションに指定したディレクトリの下位にあるディレクトリ
- cjimportapp コマンドの-a オプションに Windows のドライブ直下（C:¥など）のディレクトリ，または UNIX のルートディレクトリ（/）を指定してインポートする場合は，application.xml ファイルの<display-name>タグにアプリケーション名を指定してください。
- インポート時に指定した JAR ファイル名は，作業ディレクトリ中のディレクトリ名として用いられます。作業ディレクトリのパス長がプラットフォームの上限に達しないように JAR ファイル名を指定してください。作業ディレクトリのパス長の見積もりについては，マニュアル「アプリケーションサーバシステム構築・運用ガイド」の「付録 C.1　J2EE サーバの作業ディレクトリ」を参照してください。
- アプリケーションディレクトリ以下に Java ソースファイルを置くような構成は推奨しません。同じディレクトリに置かれているクラスファイルと Java ソースファイルの同期が取れていない場合，アプリケーションの開始やリロードに失敗する場合があります。
- 次の半角記号はエスケープ文字として扱われるため，アプリケーションディレクトリ名およびモジュール名には指定しないでください。

  ! # % +
- アプリケーションディレクトリ以下に置かれている JAR ファイルはライブラリ JAR として扱われます。このため，アプリケーションディレクトリ以下には，cjgetstubsjar コマンドで取得したスタブおよびインタフェースを置かないでください。
- 展開ディレクトリ形式の J2EE アプリケーションをインポートする際，EJB-JAR ディレクトリと WAR ディレクトリを除くアプリケーションディレクトリ以下を対象にライブラリ JAR が検索されます。このため，EJB-JAR ディレクトリと WAR ディレクトリを除くアプリケーションディレクトリ以下に多数のファイルがある場合，インポートに時間が掛かることがあります。
- アプリケーションの開始時，インポート時に展開される WAR ファイル，EJB-JAR ファイル，または EAR ファイルに含まれるエントリの展開後のファイル絶対パスが，それぞれのルートディレクトリ内にない場合，KDJE42389-E のエラーメッセージを出力して，ファイルの展開処理を中断します。

13.5　J2EE アプリケーションのデプロイとアンデプロイ

この節では，J2EE アプリケーションのデプロイとアンデプロイについて説明します。

この節の構成を次の表に示します。

表 13-5　この節の構成（J2EE アプリケーションのデプロイとアンデプロイ）

分類	タイトル	参照先
解説	アーカイブ形式の J2EE アプリケーションのデプロイとアンデプロイ	13.5.1
	展開ディレクトリ形式の J2EE アプリケーションのデプロイとアンデプロイ	13.5.2
	EAR ファイル／ZIP ファイルの展開ディレクトリ形式でのデプロイ	13.5.3
設定	J2EE アプリケーションへのプロパティの設定	13.5.4

注　「実装」および「運用」について，この機能固有の説明はありません。

13.5.1　アーカイブ形式の J2EE アプリケーションのデプロイとアンデプロイ

アーカイブ形式の J2EE アプリケーションのデプロイとアンデプロイについて説明します。

(1)　J2EE アプリケーションのデプロイ

Web アプリケーション，または EJB アプリケーションを J2EE サーバで実行するためには，J2EE アプリケーションとしてデプロイする必要があります。J2EE アプリケーションを実行する J2EE サーバに EAR ファイルをインポートしてください。

(2)　J2EE アプリケーションのアンデプロイ

アーカイブ形式の J2EE アプリケーションを削除する場合は，J2EE アプリケーションに対してアンデプロイを実行します。アンデプロイの処理完了後に，J2EE サーバから J2EE アプリケーションが削除されます。

13.5.2　展開ディレクトリ形式の J2EE アプリケーションのデプロイとアンデプロイ

展開ディレクトリ形式の J2EE アプリケーションのデプロイとアンデプロイについて説明します。

(1)　J2EE アプリケーションのデプロイ

展開ディレクトリ形式の J2EE アプリケーションを J2EE サーバ内にインポートしてクライアントから実行可能な状態にするためには，デプロイが必要です。J2EE アプリケーションを実行する J2EE サーバに，展開ディレクトリ形式の J2EE アプリケーションをインポートしてください。アプリケーション開始時に生成される各クラスローダは，クラスパスとしてアプリケーションディレクトリを保持して動作します。

(2)　J2EE アプリケーションのアンデプロイ

展開ディレクトリ形式の J2EE アプリケーションを削除する場合は，展開ディレクトリ形式の J2EE アプリケーションに対してアンデプロイを実行します。アンデプロイの処理完了後に，J2EE サーバへの J2EE アプリケーションの登録が解除されます。なお，アプリケーションディレクトリは削除されません。

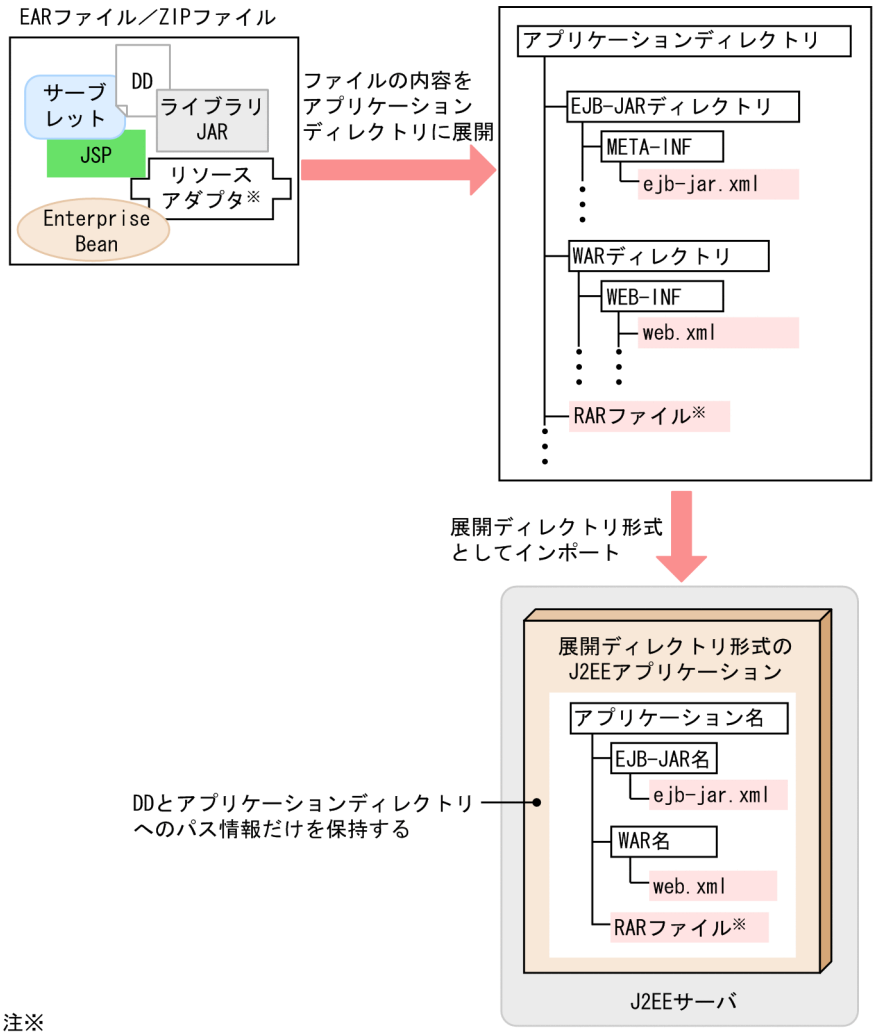
13.5.3　EAR ファイル／ZIP ファイルの展開ディレクトリ形式でのデプロイ

EAR ファイルや，J2EE サーバからエクスポートした ZIP ファイルを，展開ディレクトリ形式で J2EE サーバにインポートできます。この機能を使用することで，作成済みの EAR ファイル／ZIP ファイルを，展開ディレクトリ形式の J2EE アプリケーションとして動作させることが容易になります。

(1)　EAR ファイル／ZIP ファイルの展開

EAR ファイル／ZIP ファイルの内容を任意のディレクトリに展開して，展開ディレクトリ形式の J2EE アプリケーションとして J2EE サーバにインポートします。EAR ファイル／ZIP ファイルの展開を次の図に示します。

図 13-4　EAR ファイル／ZIP ファイルの展開



注※
リソースアダプタ（RARファイル）がEARファイル／ZIPファイル，および
アプリケーションディレクトリに含まれるのは，リソースアダプタを
J2EEアプリケーションに含めて使用する場合だけです。
J2EEリソースアダプタとしてデプロイして使用する場合には含まれません。

インポートするときに，コマンドの引数として，EAR ファイル／ZIP ファイルと，アプリケーションディレクトリとなる展開先のディレクトリを指定します。J2EE サーバは，展開先のディレクトリのパス情報をコマンドから受け取り，そのパス情報を基にアプリケーションディレクトリを生成して，アプリケーションディレクトリ下に EAR ファイル／ZIP ファイルの内容を展開します。

(2)　ディレクトリ名の生成規則

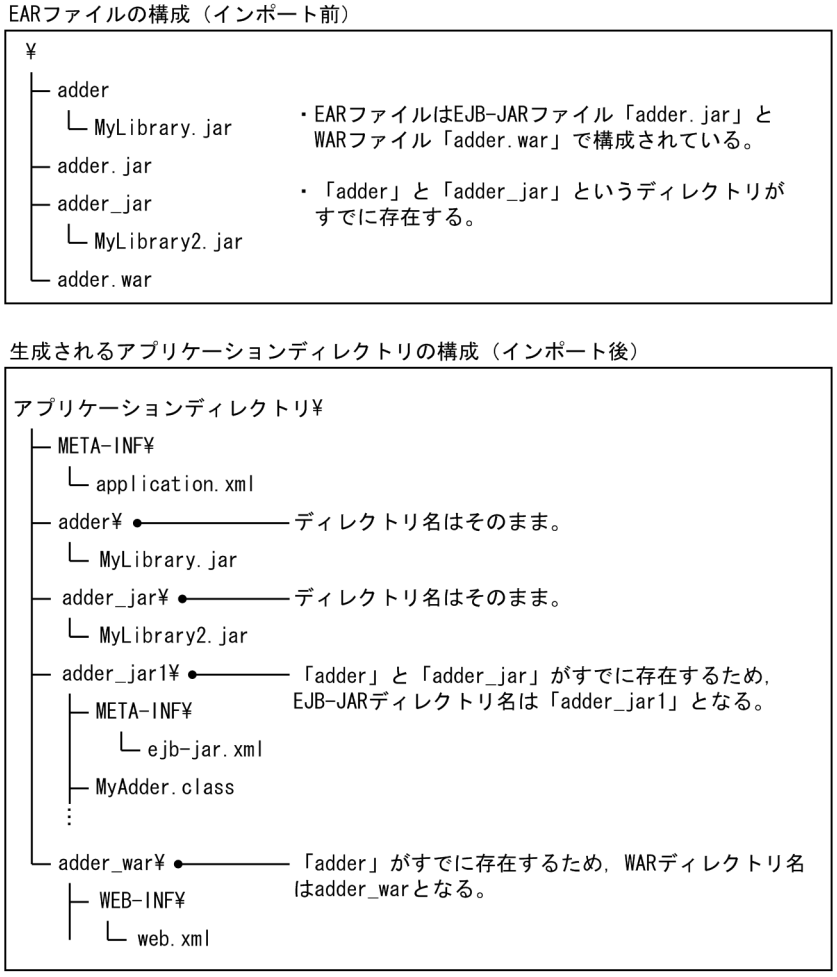
EAR ファイル／ZIP ファイルを展開ディレクトリ形式としてインポートする場合，ディレクトリ名の衝突が発生することがあります。ディレクトリ名の生成規則は，ディレクトリ名の衝突パターンによって異なります。EAR ファイル／ZIP ファイルを展開する場合の，EJB-JAR ディレクトリと WAR ディレクトリのディレクトリ名の生成規則を表に示します。

表 13-6　EAR ファイル／ZIP ファイルを展開する場合の，EJB-JAR ディレクトリと WAR ディレクトリのディレクトリ名の生成規則

ディレクトリ名の衝突パターン	ディレクトリ名の生成規則
application.xml があり，EAR ファイル内のディレクトリと衝突しない。	EJB-JAR ファイルと WAR ファイルから拡張子（「.jar」または「.war」）を除いたものが，EJB-JAR ディレクトリと WAR ディレクトリの名称になる。 EJB-JAR ファイルと WAR ファイルに拡張子がない場合は，ファイル名がそのままディレクトリ名になる。
• EJB-JAR ファイルと WAR ファイルで衝突する。 • EAR 内のディレクトリと衝突する。	EJB-JAR ファイルと WAR ファイルの拡張子（「.jar」または「.war」）をそれぞれ「_jar」または「_war」に置き換えたものが，EJB-JAR ディレクトリと WAR ディレクトリの名称になる。
「_jar」または「_war」に置き換えても衝突する。	「_jar」または「_war」の後ろに通し番号（1～2147483647）が付加される。

ディレクトリ名の衝突が発生する場合の EAR ファイルの構成と，生成されるアプリケーションディレクトリの構成を次の図に示します。

図 13-5　ディレクトリ名の衝突の例



注意事項

cjimportapp コマンドの-d オプションを使用して EAR ファイル／ZIP ファイルを展開して作成されたアプリケーションディレクトリが，次に示す条件のどれかに合致する場合，そのアプリケーションディレクトリを cjimportapp コマンドの-a オプションでインポートして展開ディレクトリ形式のアプリケーションディレクトリとして使用できません。

- EJB-JAR のモジュール名が「.jar」で終わっていない。
- WAR のモジュール名が「.war」で終わっていない。
- モジュールの拡張子を除いた名称が，ほかのモジュールの拡張子を除いた名称と重複する。
- モジュールの拡張子を除いた名称が，EAR ファイル内のディレクトリと重複する。

13.5.4　J2EE アプリケーションへのプロパティの設定

J2EE アプリケーションのプロパティ設定には，サーバ管理コマンドを使用します。

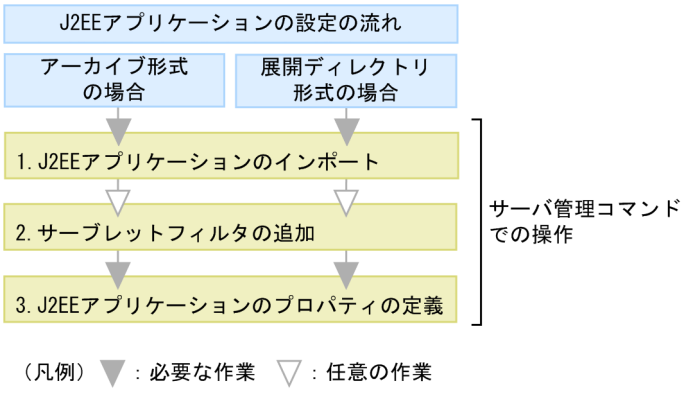
なお，ここでは，アーカイブ形式の J2EE アプリケーション（EAR 形式／ZIP 形式）や展開ディレクトリ形式の J2EE アプリケーションが作成済みであることを前提に，J2EE アプリケーションの設定方法について説明します。J2EE アプリケーションは，WTP などを使用して，アプリケーション開発時に作成しておいてください。アプリケーションの作成については，マニュアル「アプリケーションサーバ アプリケーション開発ガイド」の「4.　Eclipse を使用した J2EE アプリケーションの開発」を参照してください。

参考

サーバ管理コマンドでは，作成済みの EJB-JAR ファイルや WAR ファイルから，アーカイブ形式の J2EE アプリケーションを作成することもできます。サーバ管理コマンドでのアーカイブ形式の J2EE アプリケーションの作成については，「13.3　アーカイブ形式の J2EE アプリケーション」を参照してください。

J2EE アプリケーションの設定の流れを次の図に示します。

図 13-6　J2EE アプリケーションの設定の流れ



図中の 1.～3.について説明します。

1. サーバ管理コマンドを使用して J2EE アプリケーションをインポートします。

cjimportapp コマンドを使用して，J2EE アプリケーションをインポートします。

- アーカイブ形式の J2EE アプリケーションをインポートする場合には，-f オプションに EAR 形式／ZIP 形式のファイルを指定します。
- 展開ディレクトリ形式の J2EE アプリケーションをインポートする場合には，-a オプションにアプリケーションディレクトリを指定します。

- EAR 形式／ZIP 形式のファイルを展開ディレクトリ形式でインポートする場合には，-f オプションに EAR 形式／ZIP 形式のファイルを，-d オプションにアプリケーションディレクトリとなる展開先のディレクトリを指定します。

2. サーバ管理コマンドを使用して J2EE アプリケーションにサーブレットフィルタを追加します。

   サーブレットフィルタを追加する場合は，WAR ファイルにフィルタを登録したあと，フィルタのマッピングを定義します。サーブレットフィルタの追加については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「9.9　フィルタの設定」を参照してください。

3. サーバ管理コマンドを使用して J2EE アプリケーションのプロパティを定義します。

   cjgetappprop コマンドで各属性ファイルを取得し，ファイル編集後に，cjsetappprop コマンドで編集内容を反映させます。プロパティの設定内容については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「9.1　J2EE アプリケーションのプロパティ設定の概要」を参照してください。

サーバ管理コマンドでの操作については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「3.　サーバ管理コマンドの基本操作」を参照してください。また，コマンドについては，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「2.3　J2EE アプリケーションで使用するコマンド」を参照してください。属性ファイルについては，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「3.　J2EE アプリケーションの設定で使用する属性ファイル」を参照してください。

参考

次のような場合，実行時情報を含んだ J2EE アプリケーションをエクスポート・インポートすることで，効率良く J2EE アプリケーションを設定できます。

- 開発環境で作成した J2EE アプリケーションをエクスポートして，運用環境にインポートして使用する場合
- 運用環境ですでに動いている J2EE アプリケーションをエクスポートして，増設した J2EE サーバにインポートして使用する場合

なお，アプリケーションサーバのバージョンやプラットフォームが異なるホスト間では，J2EE アプリケーションをエクスポート・インポートして使用することはできません。J2EE アプリケーションをエクスポートするホストと，Application Server のバージョンやプラットフォームが異なるホストで J2EE アプリケーションを設定する場合は，J2EE アプリケーションを新規に作成および設定してください。

13.6　J2EE アプリケーションの入れ替え

システムの運用を開始したあとで，J2EE アプリケーションのバージョンアップやメンテナンスを実施するために，J2EE アプリケーションを入れ替えることがあります。

通常，J2EE アプリケーションを入れ替える場合には，J2EE サーバ上で動作している J2EE アプリケーションを停止したあと削除し，新しい J2EE アプリケーションをインポート，デプロイする必要があります。

通常の J2EE アプリケーションの入れ替え手順については，マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「5.6　J2EE アプリケーションの入れ替え」を参照してください。

13.7　J2EE アプリケーションのリデプロイ

リデプロイとは，少ない手順で高速に J2EE アプリケーションを入れ替えられるデプロイ方法です。この節では，リデプロイ機能を使用した J2EE アプリケーションの入れ替えについて説明します。

この節の構成を次の表に示します。

表 13–7　この節の構成（J2EE アプリケーションのリデプロイ）

分類	タイトル	参照先
解説	リデプロイによる J2EE アプリケーションの入れ替え	13.7.1
	J2EE アプリケーションの状態と入れ替え	13.7.2
注意事項	リデプロイによる J2EE アプリケーションの入れ替えの注意事項	13.7.3

注　「実装」,「設定」，および「運用」について，この機能固有の説明はありません。

リデプロイ機能を使用した J2EE アプリケーションの入れ替えでは，J2EE アプリケーションのすべての属性の情報を引き継ぐことができます。リデプロイ機能での J2EE アプリケーションの入れ替えは，次のような場合に利用できます。

J2EE アプリケーションのテスト時のロジックの修正による入れ替え

テスト実施中，J2EE アプリケーション内のロジックに問題を発見して J2EE アプリケーション内の EJB-JAR ファイルや WAR ファイルなどを修正した場合に，テスト中のアプリケーションを修正後の J2EE アプリケーションと入れ替えるときに使用できます。

運用中の J2EE アプリケーションと開発環境でテストした J2EE アプリケーションとの入れ替え

開発環境にある J2EE アプリケーションを，すでに運用中の J2EE アプリケーションと入れ替えるときに使用できます。

ただし，運用環境と開発環境のアプリケーションの構成は同じで，ロジックだけが異なる J2EE アプリケーションの場合に入れ替えできます。

リデプロイ機能を使用する場合には，通常の J2EE アプリケーションの入れ替えを実施するときに必要となる，入れ替え前の J2EE アプリケーションの停止と削除，入れ替える J2EE アプリケーションのインポートとデプロイなどの手順が不要になります。このため，通常の J2EE アプリケーションの入れ替えに比べて，少ない手順で入れ替えができるようになります。

なお，リデプロイ機能を使用した J2EE アプリケーションの入れ替えができるのは，アーカイブ形式の J2EE アプリケーションです。展開ディレクトリ形式の J2EE アプリケーションでは，リデプロイ機能は使用できません。

参考

リデプロイを実行する前に，JSP 事前コンパイル機能を実行しておくことをお勧めします。JSP 事前コンパイル機能は，Web アプリケーションに含まれる JSP ファイルをデプロイ前にコンパイルし，クラスファイルを生成する機能です。あらかじめ，クラスファイルの生成までを実施しておくので，JSP に最初にリクエストが到着したときのレスポンスタイムおよび Web アプリケーションの開始時間を短縮できます。JSP 事前コンパイル機能については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(Web コンテナ)」の「2.5　JSP 事前コンパイル機能とコンパイル結果の保持」を参照してください。

13.7.1　リデプロイによる J2EE アプリケーションの入れ替え

ここでは，リデプロイによる J2EE アプリケーションの入れ替えについて説明します。

リデプロイとは，アーカイブ形式の J2EE アプリケーションを入れ替える場合に，少ない手順で高速に入れ替えられるデプロイ方法です。ロジックだけを変更した J2EE アプリケーションを入れ替えたい場合などに利用できます。リデプロイは，サーバ管理コマンドを使用して実行できます。

リデプロイを実行できる条件を次に示します。

リデプロイを実行できる条件

- 入れ替えられるのは，実行時情報を含まない J2EE アプリケーションだけです。実行時情報を含む J2EE アプリケーション（ZIP ファイル）はリデプロイできません。
- 入れ替え前と入れ替え後の J2EE アプリケーションの構成が同じである必要があります。J2EE アプリケーションに含まれる EJB-JAR, リソースアダプタおよび WAR の数が異なったり, それらのファイル名称が異なったりする場合は，リデプロイはできません。また，J2EE アプリケーションの名称も同じである必要があります。
- 入れ替え前と入れ替え後の J2EE アプリケーションに含まれる EJB-JAR 内のホームインタフェース（ローカル，リモート），コンポーネントインタフェース（ローカル，リモート），ビジネスインタフェース（ローカル，リモート）のメソッド定義，およびアノテーションの値が同じである必要があります。
- ランタイム属性だけを引き継ぐ設定をしている場合に，アプリケーション開発環境で設定済みの DD ファイル（application.xml，ejb-jar.xml，ra.xml および web.xml）の定義内容が同じである必要があります。

また，J2EE アプリケーションを入れ替える時に，入れ替え前の J2EE アプリケーションを別な名称に変更して退避しておくと，名称による J2EE アプリケーションの世代管理が実現できます。ここでは，J2EE アプリケーションの名称変更についてもあわせて説明します。

リデプロイでは，入れ替え前の J2EE アプリケーションの情報を，入れ替え後の J2EE アプリケーションに引き継げます。次の情報が引き継げます。

Application Server のバージョンが 06-70 以降の場合

デフォルトの設定の場合，入れ替え後の J2EE アプリケーションには，入れ替え前の J2EE アプリケーションのすべての属性が引き継がれます。06-70 よりも前のバージョンと同じように，ランタイム属性だけを引き継ぎたい場合は，オプションを指定して cjreplaceapp コマンドを実行する必要があります。コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjreplaceapp（アプリケーションの入れ替え）」を参照してください。

Application Server のバージョンが 06-70 より前の場合

入れ替え前の J2EE アプリケーションの属性ファイルに設定したランタイム属性※が引き継がれます。

注※　属性ファイルには，DD(application.xml，ejb-jar.xml，ra.xml，web.xml)の定義と属性ファイル独自の定義が設定できます。属性ファイル独自の定義のことを，ランタイム属性といいます。

! 注意事項

入れ替える J2EE アプリケーションに cosminexus.xml が含まれている場合，入れ替える前のアプリケーションサーバ独自の定義情報をいったんデフォルト値に戻してから，cosminexus.xml の定義情報を読み込みます。入れ替えるアプリケーションに cosminexus.xml が含まれていない場合，入れ替え前の cosminexus.xml の定義情報を引き継ぎます。

リデプロイを実行するとき，J2EE アプリケーションは開始，停止どちらの状態でもかまいません。開始状態の J2EE アプリケーションを入れ替えた場合，J2EE アプリケーションは入れ替え後に自動的に開始されます。ただし，プールやキャッシュに格納されていた J2EE アプリケーション関連のオブジェクトは破棄されます。停止状態の J2EE アプリケーションを入れ替えた場合は，入れ替え後の J2EE アプリケーションも

停止した状態になります。J2EE アプリケーションの状態とリデプロイの関係については，「13.7.2　J2EE アプリケーションの状態と入れ替え」を参照してください。

ポイント

J2EE アプリケーションが開始されている状態でリデプロイを実行した場合，リデプロイ処理の中で J2EE アプリケーションは停止され，入れ替え後に再開始されます。このとき，停止処理の実行時間がサーバ管理コマンド（cjreplaceapp）で設定したタイムアウト時間を超過した場合，J2EE アプリケーションの強制停止が実行されます。タイムアウト時間を指定しなかった場合は，デフォルトのタイムアウト時間である 60 秒を超過すると，強制停止が実行されます。強制停止実行後にさらにタイムアウト時間を超過した場合は，コマンドが異常終了します。

また，入れ替え後に J2EE アプリケーションを再開始するとき，開始処理の実行時間がサーバ管理コマンド用の usrconf.properties の ejbserver.rmi.request.timeout キーに指定したタイムアウト時間を超過した場合も，コマンドが異常終了します。

入れ替え作業の実行形式と実行例を次に示します。

実行形式

```
cjreplaceapp 〈J2EEサーバ名〉 -name 〈J2EEアプリケーション名〉 -f 〈入れ替えるEARファイルのパス〉
```

実行例

```
cjreplaceapp MyServer -name App1 -f App1.ear
```

13.7.2　J2EE アプリケーションの状態と入れ替え

リデプロイ機能では，J2EE アプリケーションが開始・停止の状態に関係なく入れ替えできます。

開始状態のアプリケーションを入れ替えた場合，および停止状態のアプリケーションを入れ替えた場合について説明します。

(1)　開始状態の J2EE アプリケーションを入れ替えた場合

開始状態の J2EE アプリケーションを入れ替えた場合，入れ替え後の J2EE アプリケーションも開始状態になります。

開始状態の J2EE アプリケーションをリデプロイすると，次のような流れで入れ替え処理が実施されます。

1. J2EE アプリケーションを停止する
2. J2EE アプリケーションを入れ替える
3. J2EE アプリケーションを開始する

それぞれの処理については，タイムアウトがあります。

停止時のタイムアウト

J2EE アプリケーションの停止処理の実行時間が，サーバ管理コマンドでタイムアウトとして設定した時間を超えた場合，J2EE アプリケーションは強制停止されます。強制停止を実施後，強制停止で設定したタイムアウト値の時間を超えても J2EE アプリケーションが停止しない場合は，サーバ管理コマンドは異常終了します。

タイムアウトの設定については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjreplaceapp（アプリケーションの入れ替え）」を参照してください。

なお，タイムアウトの時間には，次に示す順序で，最初に取得できた値が適用されます。

1. サーバ管理コマンドに設定されたタイムアウト値

2. デフォルト値（60 秒）

入れ替え時および開始時のタイムアウト

入れ替え処理，および開始処理の実行時間でタイムアウト値として設定した時間を超えた場合，サーバ管理コマンドは異常終了します。なお，タイムアウト値は，RMI-IIOP 通信のタイムアウトに設定した時間が適用されます。RMI-IIOP 通信のタイムアウトについては，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「2.11.5　RMI-IIOP 通信のタイムアウト」を参照してください。

(2)　停止状態の J2EE アプリケーションを入れ替えた場合

停止状態の J2EE アプリケーションを入れ替えた場合，入れ替え後の J2EE アプリケーションも停止状態になります。停止状態の J2EE アプリケーションのリデプロイでは，アプリケーションの入れ替え処理だけが実施されます。

13.7.3　リデプロイによる J2EE アプリケーションの入れ替えの注意事項

リデプロイによる J2EE アプリケーションの入れ替えの注意事項について説明します。

- 入れ替え処理実行時に，J2EE サーバ上に同じ名称の J2EE アプリケーションがない場合は，エラーになります。入れ替えはできません。
- リデプロイによる J2EE アプリケーションの入れ替えでは，入れ替え前の J2EE アプリケーションの実行時情報を，入れ替え後の J2EE アプリケーション（新しい J2EE アプリケーション）に引き継ぐため，入れ替え後の J2EE アプリケーションに実行時情報が含まれていると入れ替えができません。入れ替えをしようとすると，エラー終了します。
- デフォルトの設定の場合，入れ替え後の J2EE アプリケーションには，入れ替え前の J2EE アプリケーションのすべての属性が引き継がれます。06-70 よりも前のバージョンと同じように，ランタイム属性※だけを引き継ぐ場合は，オプションを指定して cjreplaceapp コマンドを実行する必要があります。コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjreplaceapp（アプリケーションの入れ替え)」を参照してください。
- J2EE アプリケーションについて，入れ替え前と入れ替え後で次に示す内容に差異があると，入れ替え処理でエラー終了します。
  - J2EE アプリケーション内の EJB-JAR ファイルの数，リソースアダプタの数，および WAR の数が異なる。
  - J2EE アプリケーション内の EJB-JAR ファイル名，RAR ファイル名，および WAR ファイル名が，入れ替え前と入れ替え後とで異なる。
  - EJB-JAR ファイル内のホームインタフェース（ローカル／リモート），コンポーネントインタフェース（ローカル／リモート），およびビジネスインタフェース（ローカル／リモート）のメソッド定義が異なる。
  - ランタイム属性※だけを引き継ぐ設定をしている場合に，DD（application.xml，ejb-jar.xml，ra.xml，web.xml）の定義が異なる。
  - 設定しているアノテーションの値を変更した。

  また，-replaceDD オプションを指定した場合に，DD ファイル（application.xml，ejb-jar.xml，ra.xml，web.xml）も入れ替えるときには，さらに次の条件に一致するとエラーとなります。
  - DD ファイル（application.xml，ejb-jar.xml，ra.xml，web.xml）のタグが異なる。

  - DD ファイル（application.xml，ejb-jar.xml，web.xml）が存在しなくなる。または存在しなかった DD ファイルが存在するようになる。
- 開始状態の J2EE アプリケーションを入れ替えした場合，プールやキャッシュに格納された J2EE アプリケーション関連のオブジェクトは破棄されます。
- application.xml のバージョンが 1.4 以前の場合と，Java EE 5 以降の場合とではライブラリ JAR と判断する条件が異なります。ライブラリ JAR などのモジュールの決定規則については「11.4.3 application.xml がある場合のモジュールの決定規則」を参照してください。
- 入れ替え後の J2EE アプリケーションが，cosminexus.xml を含んでいて，CMP2.0 を利用する場合，cjreplaceapp コマンドは J2EE アプリケーションが停止状態の時に実行する必要があります。また，このコマンドを実行したあと，デプロイ前に cjgencmpsql コマンドを実行する必要があります。

注※

属性ファイルには，DD（application.xml，ejb-jar.xml，ra.xml，web.xml）の定義と属性ファイル独自の定義が設定できます。属性ファイル独自の定義のことをランタイム属性といいます。

13.8　J2EE アプリケーションの更新検知とリロード

展開ディレクトリ形式の J2EE アプリケーションの場合，構成するファイルを更新すると，更新した J2EE アプリケーションをリロードできます。リロード機能を使用することで，J2EE サーバを再起動することなく，デプロイ済みのサーブレット，JSP や EJB-JAR を動的に入れ替えられるようになります。

また，リロード機能を使用する場合には，通常の J2EE アプリケーションの入れ替えで必要となる，入れ替え前の J2EE アプリケーションの停止と削除，入れ替える J2EE アプリケーションのインポートとデプロイなどの手順が不要になります。このため，少ない手順で J2EE アプリケーションを動的に入れ替えられるようになります。

リロード機能は，アプリケーション開発でのテストやシステムの運用中に，修正した J2EE アプリケーションと動作中の J2EE アプリケーションを入れ替えたい場合に利用できます。

この節では，J2EE アプリケーションの更新検知とリロードについて説明します。

なお，cosminexus.xml を含むアプリケーションの場合，J2EE アプリケーションの更新検知とリロードを実行しても cosminexus.xml の情報は更新されません。

この節の構成を次の表に示します。

表 13-8　この節の構成（J2EE アプリケーションの更新検知とリロード）

分類	タイトル	参照先
解説	J2EE アプリケーションのリロード方法	13.8.1
	リロードの適用範囲	13.8.2
	リロード時のクラスローダの構成	13.8.3
	エラー発生時の動作	13.8.4
	更新検知の対象となるファイル	13.8.5
	J2EE アプリケーションの更新検知インターバル	13.8.6
	J2EE アプリケーションの構成ファイル更新用インターバル	13.8.7
	Web アプリケーションのリロード	13.8.8
	JSP のリロード	13.8.9
	ほかの機能との関係	13.8.10
	コマンドによる J2EE アプリケーションのリロード	13.8.11
設定	J2EE アプリケーションの更新検知とリロードの設定	13.8.12
注意事項	リロードの注意事項および制限事項	13.8.13

注　「実装」および「運用」について，この機能固有の説明はありません。

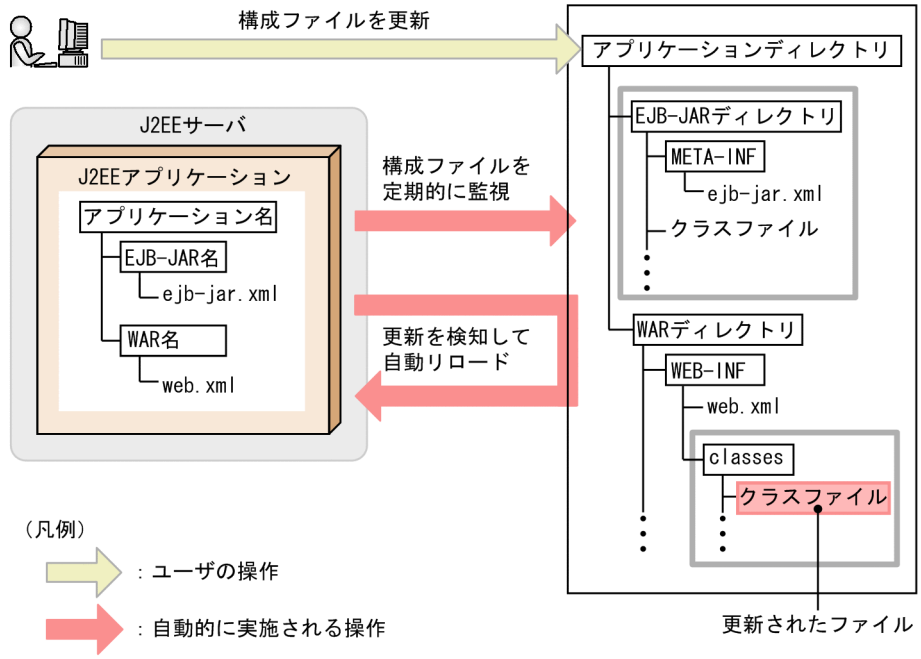
13.8.1　J2EE アプリケーションのリロード方法

J2EE アプリケーションのリロードには，更新検知によるリロードと，コマンドによるリロードの 2 種類の方法があります。

(1)　更新検知によるリロード

更新検知によるリロードは，J2EE アプリケーション開発時のテストを支援する機能として利用できます。更新検知によるリロードを次の図に示します。

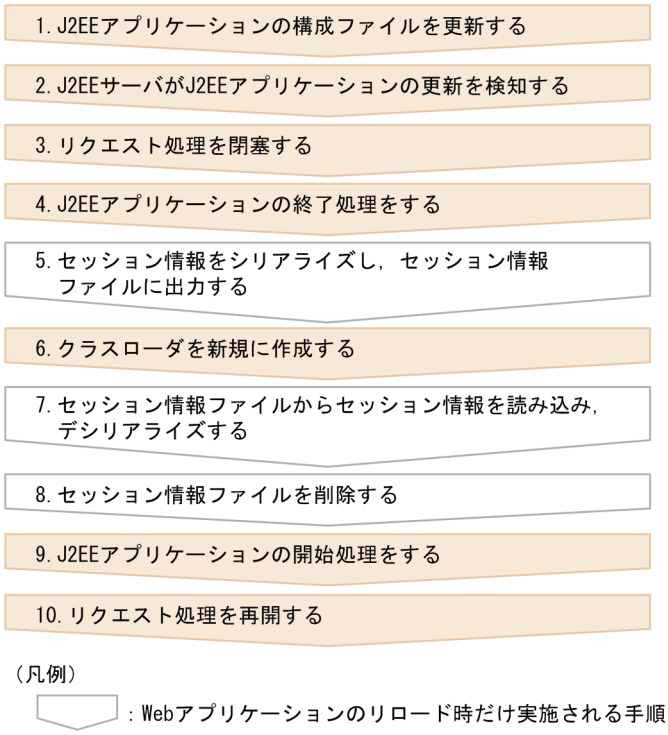
図 13-7　更新検知によるリロード



展開ディレクトリ形式の J2EE アプリケーションを構成する EJB アプリケーション（EJB-JAR）や Web アプリケーション（WAR）が更新された場合に，J2EE サーバが J2EE アプリケーションの更新を検知し，更新後の EJB-JAR や WAR を自動的にリロードします。

J2EE サーバは，J2EE アプリケーションの構成ファイルを定期的に監視し，構成ファイルの更新を検知すると，J2EE アプリケーションのリロードを実行します。J2EE アプリケーションの更新からリロードまでの処理の流れを次の図に示します。

図 13-8　更新検知によるリロードの処理の流れ



図中の 1.～10.について説明します。

1. J2EE アプリケーションの構成ファイルを更新します。

展開ディレクトリ形式の J2EE アプリケーションを構成する EJB-JAR や WAR を更新します。

2. J2EE サーバが J2EE アプリケーションの構成ファイルの更新を検知します。

- J2EE サーバは，J2EE アプリケーションの構成ファイルを定期的に監視していて，構成ファイルが更新されると更新を検知します。J2EE アプリケーションの構成ファイルを監視して更新を検知する間隔は，更新検知インターバルとして設定します。更新検知インターバルについては，「13.8.6 J2EE アプリケーションの更新検知インターバル」を参照してください。
- 更新を検知したあと，更新するファイルをロードします。このとき，ファイルのコピー中にロードが開始されてしまい，ロードに失敗することがあります。これを回避するため，構成ファイルの更新を検知してから処理中のリクエスト数の監視を開始するまでの時間を構成ファイル更新用インターバルとして設定しておくことができます。構成ファイル更新用インターバルについては，「13.8.7　J2EE アプリケーションの構成ファイル更新用インターバル」を参照してください。
- JSP を更新した場合には，JSP の再コンパイル，またはクラスファイルの監視によって更新が検知されます。JSP のリロードについては，「13.8.9　JSP のリロード」を参照してください。

3. リクエスト処理を閉塞します。

構成ファイルの更新を検知し，構成ファイル更新用インターバルで指定した時間を経過すると，J2EE アプリケーションのリロードを実行するためにリクエスト処理を閉塞します。

- EJB アプリケーション（EJB-JAR）の場合

  新規リクエストが来たらエラーを返します。処理中のリクエストがある場合には，処理を続行します。ただし，Stateless Session Bean の場合は，CTM を使用することで，新規リクエストを実行待ちにできます。
- Web アプリケーション（WAR）の場合

新規リクエストが来たら実行待ちになります。処理中のリクエストがある場合には，処理を続行します。このとき，リロード遅延実行機能を使用すると，リロードの開始処理を遅らせることができます。リロード遅延実行については，「13.8.8(1)　Web アプリケーションのリロード遅延実行」を参照してください。

参考

処理中のリクエストの処理が完了しない場合には，J2EE アプリケーション実行時間の監視のメソッドタイムアウトおよびメソッドキャンセルを実行することで，リロード処理を開始できます。リロードと J2EE アプリケーション実行時間の監視との関係については，「13.8.10(2)　リロードと J2EE アプリケーション実行時間の監視との関係」を参照してください。

4. J2EE アプリケーションの終了処理をします。

リロードを実行するために，J2EE アプリケーションを終了します。終了処理では，次の処理が実施されます。

- リロード前のクラスローダにローディングされたサーブレットのインスタンスが破棄されます。サーブレットが destroy メソッドを実装している場合，destroy メソッドが実行されます。また，JSP ファイルから生成されたサーブレットのインスタンスも破棄されます。このとき，JSP ファイルが jspDestroy メソッドを実装していると，jspDestroy メソッドが実行されます。
- javax.servlet.ServletContext に登録されたオブジェクトは破棄されます。
- リロード前のクラスローダにローディングされた EJB のインスタンスが破棄されます。このとき，次の EJB のコールバックメソッドが実行されます。

分類	実行されるメソッド
Stateless Session Bean の場合	ejbRemove PreDestroy
Stateful Session Bean の場合	ejbRemove PreDestroy
Entity Bean の場合	unsetEntityContext
Message-driven Bean の場合	ejbRemove

5. セッション情報をシリアライズし，セッション情報ファイルに出力します。

Web アプリケーションをリロードする場合は，リロード実行前に生成したセッション情報を，リロード後も継続して利用できます。セッション情報の引き継ぎについては，「13.8.8(2)　Web アプリケーションのリロード時のセッション情報の引き継ぎ」を参照してください。

6. J2EE アプリケーション単位のクラスローダを新規に作成します。

J2EE アプリケーションのリロード処理が実行されると，J2EE アプリケーション単位のクラスローダが新たに作成され，リロード後のリクエスト処理で使用されます。

7. セッション情報ファイルからセッション情報を読み込み，デシリアライズします。

Web アプリケーションをリロードする場合は，セッション情報ファイルに出力されたセッション情報を新しいクラスローダに読み込みます。

8. セッション情報ファイルを削除します。

9. J2EE アプリケーションの開始処理をします。

- リロード後は，初回アクセス時に更新後のサーブレットのインスタンスが作成され，init メソッドが実行されます。

- 開始処理では，プールの最小値分の EJB を生成してプーリングします。このとき，次の EJB のコールバックメソッドが実行されます。

分類	実行されるメソッド
Stateless Session Bean の場合	setSessionContext ejbCreate PostConstruct
Entity Bean の場合	setEntityContext
Message-driven Bean の場合	setMessageDrivenContext ejbCreate

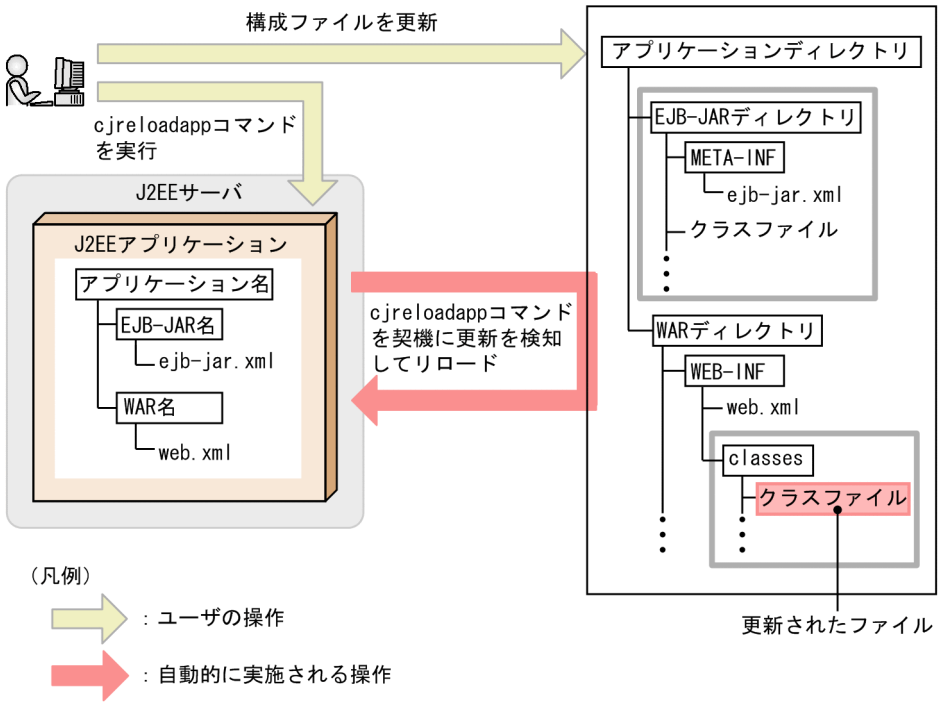
10. リクエストの閉塞を解除し，リクエストの処理を再開します。

3.で実行待ちにしていたリクエストの処理を再開します。

(2)　コマンドによるリロード

コマンドによるリロードは，J2EE アプリケーション開発時のテストを支援する機能，またはシステムの運用時に J2EE アプリケーションの入れ替えを支援する機能として利用できます。コマンドによるリロードを次の図に示します。

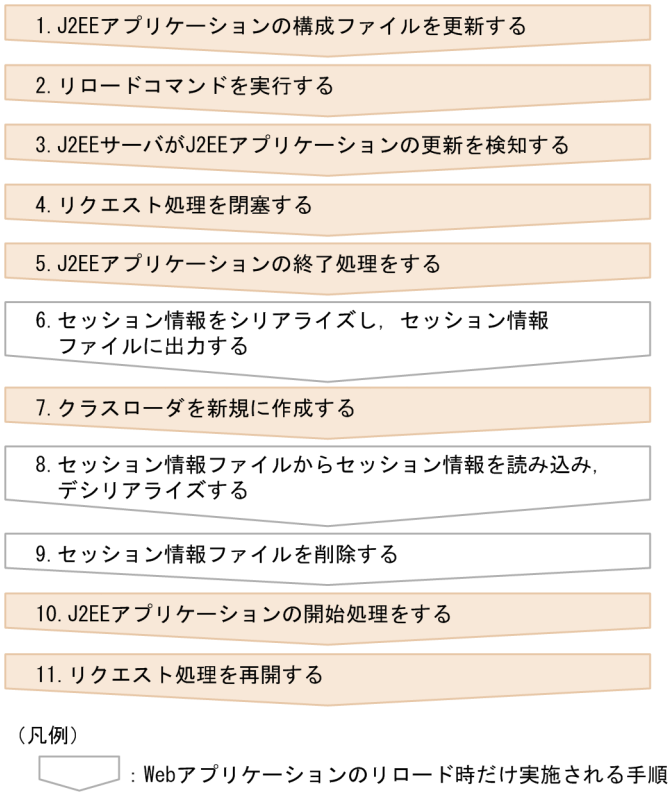
図 13-9　コマンドによるリロード



展開ディレクトリ形式の J2EE アプリケーションを構成する EJB アプリケーション（EJB-JAR）や Web アプリケーション（WAR）を更新した場合に，ユーザが cjreloadapp コマンドを実行します。J2EE サーバは，cjreloadapp コマンドを契機に J2EE アプリケーションの更新を検知し，更新後の EJB-JAR や WAR を自動的にリロードします。

J2EE アプリケーションの更新からリロードまでの処理の流れを次の図に示します。

図 13-10　コマンドによるリロードの処理の流れ



図中の 1.～4.について説明します。5.以降の手順については，更新検知によるリロードの場合と同じです。5.以降の手順については，「(1)　更新検知によるリロード」を参照してください。

1. J2EE アプリケーションの構成ファイルを更新します。

   展開ディレクトリ形式の J2EE アプリケーションを構成する EJB-JAR や WAR を更新します。

2. cjreloadapp コマンドを実行します。

3. cjreloadapp コマンドを契機に，J2EE サーバが J2EE アプリケーションの構成ファイルの更新を検知します。

4. リクエスト処理を閉塞します。

   J2EE アプリケーションのリロードを実行するためにリクエスト処理を閉塞します。

   - EJB アプリケーション（EJB-JAR）の場合

     新規リクエストが来たらエラーを返します。処理中のリクエストがある場合には，処理を続行します。ただし，Stateless Session Bean の場合は，CTM を使用することで，新規リクエストを実行待ちにできます。

   - Web アプリケーション（WAR）の場合

     新規リクエストが来たら実行待ちになります。処理中のリクエストがある場合には，処理を続行します。

   参考

   処理中のリクエストの処理が完了しない場合には，J2EE アプリケーション実行時間の監視のメソッドタイムアウトおよびメソッドキャンセルを実行することで，リロード処理を開始できます。リロードと J2EE アプリケーション実行時間の監視との関係については，「13.8.10(2)　リロードと J2EE アプリケーション実行時間の監視との関係」を参照してください。

13.8.2　リロードの適用範囲

リロードの対象として指定できるアプリケーションの種類を次の表に示します。

表 13-9　リロードの対象として指定できるアプリケーションの種類

アプリケーションの種類		適用の有無	制限事項
EJB アプリケーション（EJB-JAR）	Stateless Session Bean	△	リロード中の新規リクエストはエラーになります。なお，CTM を利用している場合は，リロード中の新規リクエストは実行待ちになります。
	Stateful Session Bean	△	リロード中の新規リクエストはエラーになります。また，リロードするとアプリケーションの状態が破棄されるため，アプリケーション開始前の状態になります。
	Singleton Session Bean	△	リロードすると Bean のインスタンスが破棄され，新しいインスタンスが作り直されるため，インスタンスの状態は保持されません。
	Entity Bean	△	リロード中の新規リクエストはエラーになります。
	Message-driven Bean	△	
Web アプリケーション（WAR）	サーブレット	○	－
	JSP	○	－
ライブラリ JAR	－	○	－

（凡例）

○：リロードの対象として指定できる

△：リロードの対象として指定できるが，制限事項がある

－：該当しない

リロードの適用範囲は，次の範囲で指定できます。

- app：EJB アプリケーション（EJB-JAR）と Web アプリケーション（WAR）をリロードの対象とする。
- web：Web アプリケーション（WAR）だけをリロードの対象とする。
- jsp：JSP だけをリロードの対象とする。

注　app，web，jsp は，usrconf.properties の ejbserver.deploy.context.reload_scope キーの指定値です。なお，none を指定した場合は，リロード機能は無効になります。

app を指定した場合

- EJB アプリケーションを更新すると，EJB アプリケーション，サーブレット，および JSP がリロードされます。
- サーブレットを更新するとサーブレットと JSP がリロードされます。
- JSP を更新すると JSP がリロードされます。

web を指定した場合

- サーブレットを更新するとサーブレットと JSP がリロードされ，JSP を更新すると JSP がリロードされます。

- サーブレットがあって JSP がない場合は，サーブレットだけがリロードされます。JSP があってサーブレットがない場合は，JSP だけがリロードされます。
- EJB アプリケーションを更新してもリロードは動作しません。

jsp を指定した場合

- JSP を更新すると JSP がリロードされます。
- EJB アプリケーションまたはサーブレットを更新しても，リロードは動作しません。

なお，リロード機能の有効／無効は，usrconf.properties の ejbserver.rmi.localinvocation.scope キーで指定するローカル呼び出し最適化機能の適用範囲と，リロード機能の適用範囲の組み合わせによって決まります。ローカル呼び出し最適化機能の適用範囲とリロード機能の適用範囲の対応を次の表に示します。

表 13-10　ローカル呼び出し最適化機能の適用範囲とリロード機能の適用範囲の対応

項目		ejbserver.rmi.localinvocation.scope キーの値		
		all	app	none
ローカル呼び出し最適化の範囲		同一 J2EE サーバ内となります。	同一アプリケーション内となります。	範囲はありません。
ejbserver.deploy.context.reload_scope キーの値	app	×※	○	○
	web	○	○	○
	jsp	○	○	○
	none	×	×	×

（凡例）○：リロード機能を使用できる　×：リロード機能を使用できない

注※　設定に誤りがあります。ejbserver.rmi.localinvocation.scope=all の場合に ejbserver.deploy.context.reload_scope=app を指定すると，J2EE サーバを起動するときにメッセージが出力されて起動に失敗します。

13.8.3　リロード時のクラスローダの構成

リロード時のクラスローダの構成は，ローカル呼び出し最適化の範囲によって異なります。ローカル呼び出し最適化の範囲とクラスローダの構成の対応を次の表に示します。なお，クラスローダの構成については，「付録 B　クラスローダの構成」を参照してください。

表 13-11　ローカル呼び出し最適化の範囲とクラスローダの構成の対応

項目	ejbserver.rmi.localinvocation.scope キー※の値		
	all	app	none
ローカル呼び出し最適化の範囲	同一 J2EE サーバ内となります。	同一アプリケーション内となります。	範囲はありません。
クラスローダ構成	ローカル呼び出し最適化時のクラスローダ構成になります。	デフォルトのクラスローダ構成になります。	デフォルトのクラスローダ構成になります。

注※　usrconf.properties に指定するキーです。

リロード機能では，ApplicationClassLoader 以下，または WebappClassLoader 以下のクラスローダを入れ替えます。EJB-JAR をリロードする場合，デフォルトのクラスローダ構成で，次のファイルをロードします。

- ApplicationClassLoader では，J2EE アプリケーションに含まれる EJB-JAR，ライブラリ JAR，および参照ライブラリをロードします。
- WebappClassLoader では，J2EE アプリケーションに含まれる WAR をロードします。
- JasperLoader では，J2EE アプリケーションに含まれる JSP をロードします。

EJB-JAR をリロードするために ApplicationClassLoader を入れ替える場合は，下位にある WebappClassLoader，および JasperLoader も入れ替える必要があります。したがって，EJB-JAR，ライブラリ JAR，参照ライブラリをリロードする場合は，WAR を含めたリロードになります。

13.8.4　エラー発生時の動作

リロード機能の使用中にエラーが発生した場合の動作を次の表に示します。

表 13-12　リロード機能でのエラー発生時の動作

エラーの内容	結果
リロード中（アプリケーション内で処理しているリクエストがない状態）にリクエストがサーバに来た場合	EJB-JAR の場合はクライアントにエラーが返ります。WAR の場合，新規リクエストはリロードが完了するまで実行待ちとなります。
リロード中に例外が発生した場合（クラスファイルのロードの失敗を含む）	更新検知は停止しないで，アプリケーションを停止します。更新を検知した場合，停止した J2EE アプリケーションは再び開始されます。
リロード以外の処理（更新チェック中，構成ファイル更新用インターバル中，リロード遅延実行中）で例外が発生した場合	更新検知は停止しないで，J2EE アプリケーションは開始状態のまま監視を続けます。
J2EE アプリケーションのメソッドがハングアップなどで終了しない場合	J2EE アプリケーション実行時間の監視でメソッドをキャンセルしないとリロードできません。また，cjreloadapp コマンドを使用しないとリロードできません。 J2EE アプリケーション実行時間の監視については，「13.8.10(2)　リロードと J2EE アプリケーション実行時間の監視との関係」を参照してください。

13.8.5　更新検知の対象となるファイル

クラスローダによってロードされるファイルのうち，監視対象のファイルが更新されたときに，J2EE サーバが更新を検知してリロードが実行されます。更新検知の対象となるタイミングは，J2EE アプリケーションの開始時です。

ただし，クラスローダでロードされていないファイルは，更新検知の対象になりません。アプリケーションディレクトリ以下にあってもロードされていないファイルや，アプリケーションディレクトリを使用していない J2EE アプリケーションのファイルは，更新検知の対象になりません。また，DD（ejb-jar.xml，web.xml）は，更新検知の対象になりません。

また，展開ディレクトリ形式で，更新後の JSP ファイルの更新日付が古い場合は JSP の再コンパイルが実行されません。

(1)　更新検知の対象ファイル

更新検知の対象ファイルを次の表に示します。

表 13–13　更新検知の対象ファイル

種別		更新検知の対象ファイル	リロードの適用範囲※1		
			app	web	jsp
アプリケーションディレクトリ以下のファイル		• ライブラリ JAR	○	−	−
EJB-JAR ディレクトリ以下の EJB アプリケーション（EJB-JAR）のファイル		• クラスファイル • java.util.ResourceBundle によってロードされたプロパティファイル	○	−	−
WAR ディレクトリ以下の Web アプリケーション（WAR）のファイル	サーブレット	• クラスファイル • JAR ファイル • java.util.ResourceBundle によってロードされたプロパティファイル	○	○	−
	JSP	• JSP ファイル	○※2	○※2	○※2
		• タグファイル • タグライブラリ記述子	○※3	○※3	○※3
		• JSP コンパイル結果 • タグファイルコンパイル結果	○※4	○※4	○※4
		• 静的コンテンツ	○※5	○※5	○※5
アプリケーションディレクトリ以外のファイル		• 参照ライブラリ	○	−	−

（凡例）○：更新検知の対象になる　−：更新検知の対象にならない

注※1　リロードの適用範囲の app, web, jsp は，usrconf.properties の ejbserver.deploy.context.reload_scope キーの指定値です。

注※2　JSP の事前コンパイルがされていない状態で，ロード済みの場合に，更新検知の対象になります。

注※3　JSP の事前コンパイルがされていない状態で，ロード済みの JSP ファイルまたはタグファイルで使用されている場合に，更新検知の対象になります。

注※4　JSP の事前コンパイルがされている場合に，更新検知の対象になります。

注※5　JSP ファイルまたはタグファイルが依存するファイルの場合に，更新検知の対象になります。依存するファイルは，JSP ファイルまたはタグファイルの include ディレクティブでインクルードされるファイルや，web.xml の <include-prelude>または<include-coda>でインクルードされるファイルのことです。

(2)　更新検知の対象ファイルの例

アプリケーションディレクトリ以下のファイルのうち，更新検知の対象となるファイルの例を次の表に示します。

表 13-14　更新検知の対象ファイルの例

ディレクトリ		ディレクトリ下のファイル	更新検知の対象
<アプリケーションディレクトリ>		ライブラリ JAR 例：MyLibrary.jar	○
META-INF		• J2EE アプリケーションの DD（application.xml） • アプリケーション属性ファイル（cosminexus.xml）	−
<EJB-JAR ディレクトリ>		−	−
	META-INF	EJB-JAR の DD（ejb-jar.xml）	−
	<パッケージ名>	Enterprise Bean クラス 例：MyEJB.class	○
		ホームインタフェースクラス 例：MyEJBHome.class	○
		コンポーネントインタフェースクラス 例：MyEJBRemote.class	○
		ビジネスインタフェースクラス 例：MyBusiness.class	○
		EJB から利用されるクラス 例：MyClass.class	○
		java.util.ResourceBundle によってロードされたプロパティファイル 例：MyResource.properties	○
<WAR ディレクトリ>		JSP ファイル 例：MyJsp.jsp	○
		静的コンテンツ 例：MyStatic.jspf	○
		HTML ファイル 例：MyIndex.html	−
	META-INF	−	−
	WEB-INF	Web アプリケーションの DD（web.xml）	−
		タグライブラリ記述子ファイル 例：MyTaglib.tld	○
		Web アプリケーションプロパティファイル（hitachi_web.properties）	−

ディレクトリ						ディレクトリ下のファイル	更新検知の対象
			classes			−	−
				<パッケージ名>		java.util.ResourceBundle によってロードされたプロパティファイル 例：MyResource.properties	○
						サーブレットクラス 例：MyServlet.class	○
			lib			JAR ファイル 例：MyLibrary.jar	○
			tags			タグファイル 例：MyTag.tag	○
			<JSP ワークディレクトリ>			JSP コンパイル結果 例：MyJsp$jsp.class	○
				WEB-INF	tags	タグファイルコンパイル結果 例：MyTag$tag.class	○
<アプリケーションディレクトリ以外のディレクトリ>						参照ライブラリ 例：MyRef.jar	○

（凡例）　○：更新検知の対象になる　−：更新検知の対象にならない

(3)　リロードの対象となる操作

監視対象のファイルを変更した場合に，更新が検知されてリロードが実行されるかどうかは，ファイルへの操作によって異なります。リロードの対象となる操作を次の表に示します。

表 13-15　リロードの対象となる操作

更新検知の対象ファイル		ファイルへの操作			注意点
		追加	削除	更新	
アプリケーションディレクトリ以下のファイル	ライブラリ JAR	×	×	○	• アノテーションも読み込まれます。※1 • ライブラリ JAR の名称は変更できません。 • ライブラリ JAR の構成は変更できます。
	アプリケーション属性ファイル	×	×	×	• リロードを実行する際，cosminexus.xml の内容が変更または削除されている場合は，定義情報の再読み込みは実施しません。
EJB-JAR ディレクトリ以下の EJB アプリケーション（EJB-JAR）のファイル	EJB を構成するクラスファイル	×	×	○	• アノテーションも読み込まれます。※1

更新検知の対象ファイル		ファイルへの操作			注意点
		追加	削除	更新	
EJB-JAR ディレクトリ以下の EJB アプリケーション（EJB-JAR）のファイル	EJB を構成するクラスファイル	×	×	○	• クラス名の変更はできません。 • これまで参照していなかったクラスを参照するように更新した場合，参照先のクラスファイルもリロードの対象となります。
	ほかのクラスファイル	×	×	○	• アノテーションも読み込まれます。※1 • これまで参照していなかったクラスを参照するように更新した場合，参照先のクラスファイルもリロードの対象となります。
WAR ディレクトリ以下の Web アプリケーション（WAR）のファイル	クラスファイル	△	○	○	－
	JAR ファイル	○	○	○	－
	プロパティファイル	△	○	○	－
	JSP ファイル	△	○※2	○※2	－
	タグファイル	△	○※2	○※2	• JAR ファイル内にあるタグファイルは除きます。
	JSP ファイルまたはタグファイルが依存するファイル	△	○※2	○※2	－
	静的コンテンツ	△	×	×	• リロード機能を使用した場合には，キャッシュが無効になります。
	JSP コンパイル結果（JSP，タグファイルをコンパイルして生成されたクラスファイル）	△	○	○	• JSP の事前コンパイル機能が有効である必要があります。
アプリケーションディレクトリ以外のファイル	参照ライブラリ	×	×	○	• 参照ライブラリの名称は変更できません。 • 参照ライブラリが指す JAR ファイルの構成およびクラスパス内のファイルの更新はできます。 • スタブおよびコンテナ生成クラスの再生成が必要な場合には，再生成されます。

（凡例）

○：リロードが実行される

△：操作はできるが，リロードは実行されない

×：リロードが実行されない

－：該当しない

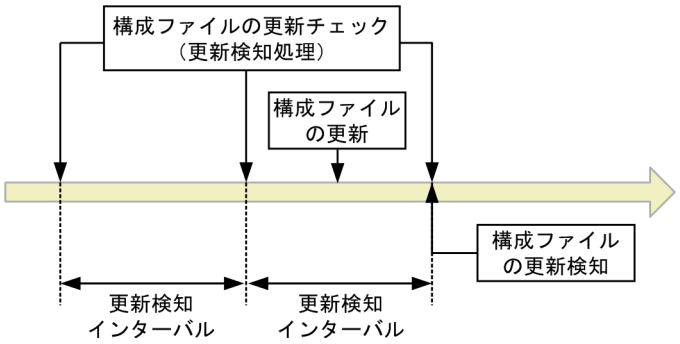
注※1　更新できるアノテーションと更新できないアノテーションがあります。アノテーションについては，「12.6.1 アノテーションの更新」を参照してください。

注※2　JSP の事前コンパイル機能が有効な場合は，△になります。

13.8.6　J2EE アプリケーションの更新検知インターバル

J2EE サーバは，J2EE アプリケーションの構成ファイルを定期的に監視していて，構成ファイルが更新されると更新を検知します。J2EE アプリケーションの構成ファイルを監視して更新を検知する間隔は，更新検知インターバルとして設定します。更新検知インターバルを次の図に示します。

図 13-11　更新検知インターバル



更新検知インターバルの値を大きくすると，構成ファイルを監視する間隔が長くなり，ファイル更新後のリロードの反映が遅くなります。また，値を小さくすると，リロードの反映が早くなります。

ポイント

更新検知の対象となるファイル数が増えると，更新検知処理のオーバーヘッドが大きくなり，CPU 使用率が高くなります。このような場合は，更新検知インターバルを変更することによって，性能への影響が小さくなります。更新検知インターバルの値は大きくすることをお勧めします。

更新検知インターバルは，J2EE アプリケーション，Web アプリケーション，および JSP でそれぞれ設定できます。設定値の関係を次に示します。

EJB アプリケーションの場合

J2EE アプリケーションの更新検知インターバル（ejbserver.deploy.context.check_interval）が使用されます。なお，J2EE アプリケーションの更新検知インターバルに 0 を指定している場合，EJB アプリケーションは更新検知されません。

サーブレットの場合

Web アプリケーションまたは J2EE アプリケーションの更新検知インターバルが使用されます。優先順位を次に示します。

1. Web アプリケーションの更新検知インターバル（webserver.context.check_interval）
2. J2EE アプリケーションの更新検知インターバル（ejbserver.deploy.context.check_interval）

なお，Web アプリケーションの更新検知インターバルを指定していない場合には，J2EE アプリケーションの更新検知インターバルが使用されます。

また，Web アプリケーションの更新検知インターバルに 0 を指定している場合，サーブレットは更新検知されません。

JSP の場合

JSP または J2EE アプリケーションの更新検知インターバルが使用されます。優先順位を次に示します。

1. JSP の更新検知インターバル（webserver.jsp.check_interval）
2. J2EE アプリケーションの更新検知インターバル（ejbserver.deploy.context.check_interval）

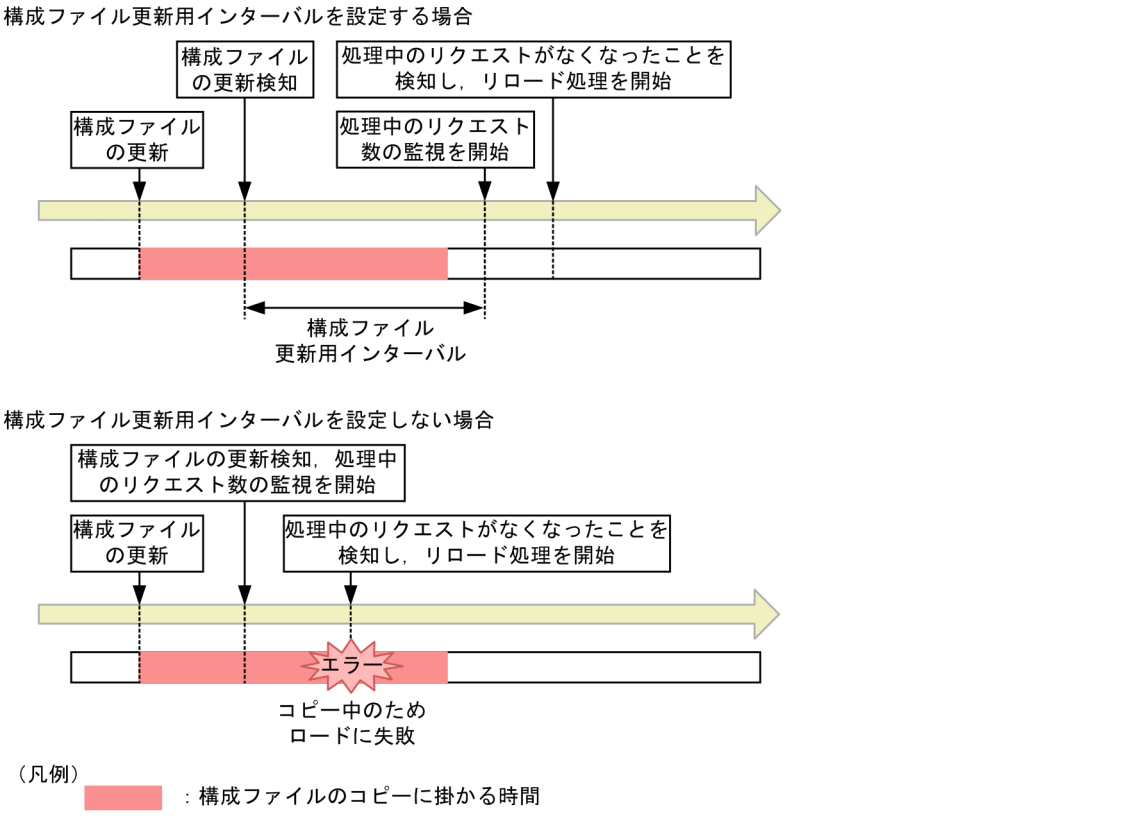
なお，JSP の更新検知インターバルを指定していない場合には，J2EE アプリケーションの更新検知インターバルが使用されます。

また，JSP の更新検知インターバルに 0 を指定している場合，JSP は更新検知されません。

13.8.7　J2EE アプリケーションの構成ファイル更新用インターバル

J2EE サーバは，構成ファイルの更新を検知すると処理中のリクエスト数の監視を開始し，処理中のリクエストの処理が完了するとファイルをロードします。ファイルの容量が大きい場合，ネットワークを経由した場合や，ファイルが複数ある場合には，ファイルコピーなどの処理時間が長くなることがあります。このとき，ファイルコピーが完了する前に処理中のリクエストがなくなると，ファイルコピー中にロードが開始されて，ロードに失敗するおそれがあります。これを回避するため，ファイルコピーに掛かる時間を考慮し，コピーが完了してからロードが開始されるように，構成ファイル更新用インターバルを設定しておくことができます。構成ファイル更新用インターバルでは，構成ファイルの更新を検知してから処理中のリクエスト数の監視を開始するまでの時間を設定します。構成ファイル更新用インターバルを次の図に示します。

図 13-12　構成ファイル更新用インターバル



ポイント

- 構成ファイル更新用インターバルには，実際に掛かるコピー時間にゆとりを持たせた時間を設定することをお勧めします。

- 複数のファイルを同時に更新する場合は，複数のファイルの更新に掛かる時間を算出して，構成ファイル更新用インターバルを設定してください。

構成ファイル更新用インターバルは，J2EE アプリケーション，Web アプリケーション，および JSP でそれぞれ設定できます。設定値の関係を次に示します。

EJB アプリケーションの場合

J2EE アプリケーションの構成ファイル更新用インターバル（ejbserver.deploy.context.update.interval）が使用されます。

サーブレットの場合

Web アプリケーションまたは J2EE アプリケーションの構成ファイル更新用インターバルが使用されます。優先順位を次に示します。

1. Web アプリケーションの構成ファイル更新用インターバル（webserver.context.update.interval）
2. J2EE アプリケーションの構成ファイル更新用インターバル（ejbserver.deploy.context.update.interval）

Web アプリケーションの構成ファイル更新用インターバルを指定していない場合には，J2EE アプリケーションの構成ファイル更新用インターバルが使用されます。

JSP の場合

JSP または J2EE アプリケーションの構成ファイル更新用インターバルが使用されます。優先順位を次に示します。

1. JSP の構成ファイル更新用インターバル（webserver.jsp.update.interval）
2. J2EE アプリケーションの構成ファイル更新用インターバル（ejbserver.deploy.context.update.interval）

JSP の構成ファイル更新用インターバルを指定していない場合には，J2EE アプリケーションの構成ファイル更新用インターバルが使用されます。

13.8.8　Web アプリケーションのリロード

Web アプリケーションをリロードする場合には，リロード遅延実行機能を使用することで，J2EE アプリケーションのサービス停止期間を最小限に抑えることができます。また，Web アプリケーションのリロードでは，リロード実行前に生成したセッション情報を引き継いで，リロード後も継続して利用することができます。ここでは，Web アプリケーションのリロード遅延実行とセッション情報の引き継ぎについて説明します。

(1)　Web アプリケーションのリロード遅延実行

デフォルトでは，構成ファイルの更新を検知すると，次のような手順でリロード処理を実行します。

1. **Web アプリケーションのリクエスト処理を閉塞します。**

   構成ファイルの更新を検知して構成ファイル更新用インターバルで指定した時間を経過すると，処理中のリクエスト数の監視を開始し，次のようにリクエストを処理します。

   - 新規リクエスト：実行待ちにします。
   - 処理中のリクエスト：処理を続行します。

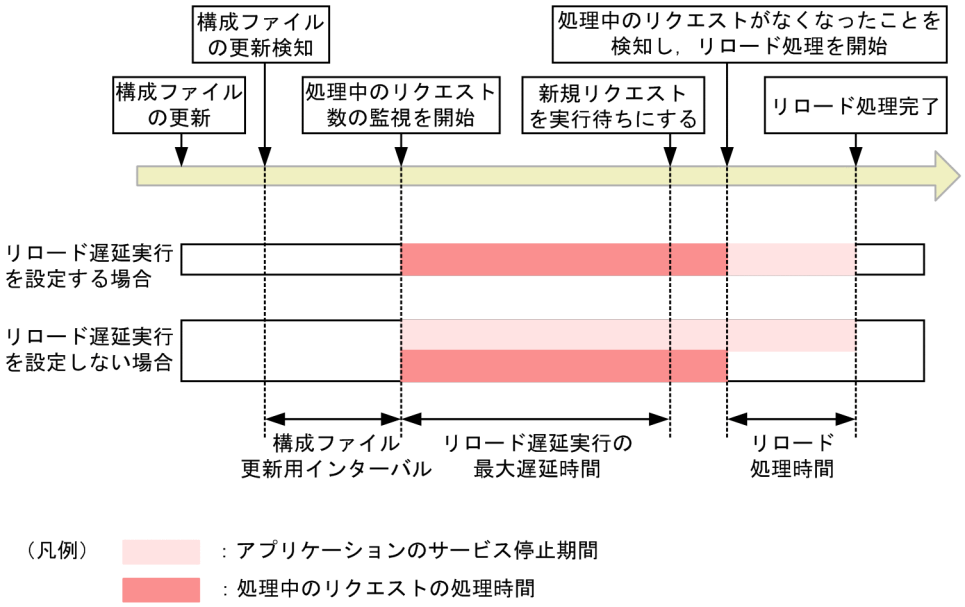
2. **処理中のリクエストの処理が完了すると，リロードを実行します。**

この場合，処理中のリクエストの処理時間が長くなるほど，新規リクエストの処理開始時間が遅くなり，新規リクエストに対するアプリケーションのサービス停止期間が長くなります。これを回避するために，リロード遅延実行を設定しておくことをお勧めします。リロード遅延実行機能を使用すると，処理中のリクエストの実行待ち時間を減らして，アプリケーションのサービス停止期間を最小限に抑えられます。

リロード遅延実行機能では，この機能を使用するかどうか，およびリロード処理を開始するまでの時間（最大遅延時間）を設定します。リロード遅延実行機能を使用すると，処理中のリクエストの処理が終わるまで，新規リクエストを受け付けることができます。処理中のリクエスト数が 0 になると，リロード処理を開始して新規リクエストを実行待ちにします。ただし，処理中のリクエスト数が 0 になるかどうかは，アクセス状況に依存します。リクエストが途切れない場合には，リロード処理を開始できません。このような場合には，構成ファイルの更新を検知してから，新規リクエストを実行待ちにするまでの最大遅延時間を設定します。最大遅延時間を経過すると，それ以降の新規リクエストは実行待ちとなり，処理中のリクエストの処理が完了するのを待ってリロード処理を開始します。

リロード遅延実行を設定した場合と設定しない場合の，アプリケーションのサービス停止期間の違いを，次の図に示します。

図 13-13　アプリケーションのサービス停止期間の違い



リロード遅延実行の最大遅延時間を設定している場合には，構成ファイルの更新を検知して構成ファイル更新用インターバルで指定した時間を経過すると，処理中のリクエスト数の監視を開始し，リロード遅延実行の最大遅延時間のカウントを開始します。例えば，最大遅延時間を 5 分と設定していた場合，アクセス状況によってリロード処理の開始時間が次のように異なります。

- 最大遅延時間を経過する前に処理中のリクエストが 0 になった場合

  最大遅延時間のカウントを開始してから 3 分後に，処理中のリクエストが 0 になった場合には，その時点でリロード処理を開始します。また，新規リクエストを実行待ちにします。

- 最大遅延時間を経過した場合

  リクエストが途切れないため処理中のリクエストが 0 にならない場合には，最大遅延時間のカウントを開始してから 5 分後に，新規リクエストを実行待ちにします。処理中のリクエストの処理が完了したら，リロード処理を開始します。

参考

処理中のリクエストの処理が完了しない場合には，J2EE アプリケーション実行時間の監視のメソッドタイムアウトおよびメソッドキャンセルを実行することで，リロード処理を開始できます。J2EE アプリケーション実行時間の監視については，「13.8.10(2)　リロードと J2EE アプリケーション実行時間の監視との関係」を参照してください。

(2)　Web アプリケーションのリロード時のセッション情報の引き継ぎ

Web アプリケーションをリロードする場合は，リロード実行前に生成したセッション情報を引き継いで，リロード後も継続して利用します。

(a)　セッション情報ファイル

Web アプリケーションをリロードする場合，セッション情報（javax.servlet.http.HttpSession オブジェクトに登録したオブジェクト）をシリアライズし，**セッション情報ファイル**に出力します。セッション情報ファイルからセッション情報を読み込み，リロード後のクラスローダ上でデシリアライズします。なお，シリアライズ処理，およびデシリアライズ処理の時間は，リロードを実行している Web アプリケーション上で生成されたセッション数，javax.servlet.http.HttpSession オブジェクトに登録したオブジェクトの容量などに依存します。

! 注意事項

シリアライズ対象となるセッションに登録したオブジェクトのクラス，およびそこから参照されるオブジェクトのクラスを，デシリアライズできない構成に更新してリロードを実行した場合，セッションのデシリアライズに失敗します。デシリアライズに失敗すると，Web アプリケーション上のすべてのセッション情報が破棄されます。

セッション情報ファイルは，リロード処理開始時に作成され，リロード処理完了時に削除されます。リロード実行時にセッションが存在しない場合でもセッション情報ファイルは作成されます。

セッション情報ファイルには，Web コンテナが自動生成する情報（2 キロバイト）とセッション情報が出力されます。セッション情報は，セッション数とセッションに登録したセッション情報（オブジェクト）に依存します。

セッション情報ファイルのファイル容量は，リロード時に出力されるシリアライズ処理完了のメッセージからファイル容量を確認し，セッション数に応じて算出してください。シリアライズ処理完了時のメッセージの出力例を次に示します。

```
KDJE39168-I The serialization of session objects has finished. (J2EE application = app1, context root = /
examples, number of sessions = 10, session information file size(byte) = 12345)
```

この出力例では，コンテキストルート名が「examples」の Web アプリケーション「app1」上に 10 個のセッションがあり，セッション情報ファイルのサイズは，12,345 バイトです。

(b)　セッション情報を引き継ぐ場合の注意事項

- リロード時のセッション引き継ぎ機能を使用する場合，javax.servlet.http.HttpSession に登録するセッション情報は，シリアライズできるオブジェクトである必要があります。
- HttpSession オブジェクトに登録したオブジェクトから参照するオブジェクトは，transient で宣言されているか，またはシリアライズできるオブジェクトである必要があります。以降，参照するオブジェクトも同様です。シリアライズできるオブジェクトについての詳細は，J2SE の仕様書を参照してください。

- セッション情報としてシリアライズできないオブジェクトを登録した場合は，エラーが発生します。エラーの内容はオブジェクトによって異なります。
  - javax.servlet.http.HttpSession に登録したオブジェクトの場合

    HttpSession オブジェクトに登録したオブジェクトが，シリアライズできないオブジェクトである場合，該当するオブジェクトだけを破棄します。破棄されたオブジェクトは，リロードが完了したあとに使用することはできません。また，該当するオブジェクトが，javax.servlet.http.HttpSessionBindingListener インタフェースを実装していても，Web コンテナからアンバインドされたイベントは通知されません。
  - javax.servlet.http.HttpSession に登録したオブジェクトから参照されるオブジェクトの場合

    HttpSession に登録したオブジェクトから参照されるオブジェクトがシリアライズできないオブジェクトの場合，すべてのセッション情報が破棄されます。HttpSession に登録したオブジェクトから参照されるオブジェクトが，HttpSession に登録したオブジェクトの場合も同様に，すべてのセッション情報が破棄されます。

13.8.9　JSP のリロード

JSP を更新した場合には，JSP の再コンパイル，またはクラスファイルの監視によって更新が検知され，JSP がリロードされます。JSP のリロード方法を次に示します。

- JSP の再コンパイルによるリロード

  Web コンテナは，ロードされた JSP ファイル，タグファイル，JSP ファイルまたはタグファイルが依存するファイルが更新されているかどうかをチェックし，更新日時がロード時と異なる場合には，再コンパイルして JSP をリロードします。

  更新を検知して JSP の構成ファイル更新用インターバルを経過したあとに，ファイルを再コンパイルします。コンパイルが完了し，処理中のリクエストがなくなったことを検知したら，リロード処理を開始します。
- クラスファイルの監視によるリロード

  Web コンテナは，Web コンテナにロードされた JSP ファイルから生成されたクラスファイルが更新されているかどうかをチェックし，更新日時がロード時と異なる場合に，JSP をリロードします。

  更新を検知して JSP の構成ファイル更新用インターバルを経過したあとに，処理中のリクエストがなくなったことを検知したら，リロード処理を開始します。

  クラスファイルの更新のタイミングは次のどちらかです。
  - JSP 事前コンパイルのコマンドを実行し，JSP ファイルをコンパイルしてクラスファイルを生成したとき
  - JSP 事前コンパイルでコンパイルされたクラスファイルを，JSP ワークディレクトリにコピーして上書きしたとき

  JSP の事前コンパイルについては，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(Web コンテナ)」の「2.5　JSP 事前コンパイル機能とコンパイル結果の保持」を参照してください。

JSP のリロード方法の差異を次の表に示します。

表 13-16　JSP のリロード方法の差異

項目	JSP の再コンパイルによるリロードの場合	クラスファイルの監視によるリロードの場合
JSP の事前コンパイル機能使用の有無	JSP 事前コンパイル機能は無効となります。	JSP 事前コンパイル機能は有効となります。

項目	JSP の再コンパイルによるリロードの場合	クラスファイルの監視によるリロードの場合
更新検知の対象ファイル※	• JSP ファイル • タグファイル • JSP ファイルまたはタグファイルが依存するファイル	• JSP 事前コンパイルで生成されたクラスファイル
更新検知後の処理	ファイルをコンパイルし，リロードします。	クラスファイルをリロードします。

注※　Web コンテナにロードされていないファイル，または更新検知の対象ではないファイルを更新しても検知されません。

どちらのリロード方法の場合でも，ファイルの更新を検知するために，JSP の更新検知インターバルと JSP の構成ファイル更新用インターバルを指定できます。更新検知インターバルについては，「13.8.6　J2EE アプリケーションの更新検知インターバル」を参照してください。構成ファイル更新用インターバルについては，「13.8.7　J2EE アプリケーションの構成ファイル更新用インターバル」を参照してください。

! 注意事項

ロード済みの JSP の破棄のタイミング，および JSP のリロード機能の監視対象について

Web アプリケーションのリロード機能と JSP のリロード機能を併用した場合に Web アプリケーションのリロード機能が実行されたとき，ロード済みの JSP が破棄されます。また，JSP のリロード機能の監視対象は Web アプリケーションのリロード機能実行時にロードされた JSP ファイルだけになります。

Web コンテナおよび Web アプリケーションの再起動をしたとき，ロード済みの JSP が破棄されます。また，JSP のリロード機能の監視対象は Web アプリケーションの開始後にロードされた JSP ファイルだけになります。

13.8.10　ほかの機能との関係

リロード機能と次の機能との関係について説明します。

- 同時実行スレッド数制御
- J2EE アプリケーション実行時間の監視

(1)　リロードと同時実行スレッド数制御との関係

Web アプリケーションのリロード機能での新規リクエストの実行待ちと，同時実行スレッド数制御との関係について説明します。なお，同時実行スレッド数の制御については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(Web コンテナ)」の「2.15　同時実行スレッド数の制御の概要」を参照してください。

(a)　Web アプリケーション単位，または URL グループ単位の同時実行スレッド数制御を使用する場合

リロード機能での新規リクエストの実行待ちは，Web アプリケーション単位，または URL グループ単位の同時実行スレッド数制御の実行待ちキューとは，別に制御されます。

- **Web アプリケーション単位，または URL グループ単位の同時実行スレッド数制御が設定されている場合**

  Web アプリケーション単位，または URL グループ単位の同時実行スレッド数制御が設定されている Web アプリケーションをリロードする場合，リロード機能での新規リクエストの実行待ちは次のように制御されます。

- リロード機能で実行待ちとなった新規リクエストは，Web アプリケーション単位，または URL グループ単位の実行待ちキューに登録されません。それらの実行待ちキューのサイズの上限に関係なく，実行待ちとなります。
- Web アプリケーション単位，または URL グループ単位の実行待ちキューに登録されたリクエストがある場合は，それらの実行待ちキューに登録されたリクエストの処理がすべて完了してから，リロードが開始されます。
- リロード処理完了後，リロード機能で実行待ちとなった新規リクエストの処理が開始されますが，同時実行スレッド数を超えたリクエストは，Web アプリケーション単位，または URL グループ単位の実行待ちキューに登録されます。それらの実行待ちキューのサイズの上限を超えたリクエストは，ステータスコード 503（Service Unavailable）のエラーを返します。

- **Web アプリケーション単位の同時実行スレッド数制御が設定されていない場合**

  Web アプリケーション単位の同時実行スレッド数制御が設定されていない Web アプリケーションをリロードする場合，リロード機能での新規リクエストの実行待ちは次のように制御されます。

  - リロード機能で実行待ちとなった新規リクエストは，デフォルトの実行待ちキューに登録されません。デフォルトの実行待ちキューサイズの上限に関係なく，実行待ちとなります。
  - デフォルトの実行待ちキューに登録されたリクエストがある場合は，その実行待ちキューに登録されたリクエストの処理がすべて完了してから，リロードが開始されます。
  - リロード処理完了後，リロード機能で実行待ちとなった新規リクエストの処理が開始されますが，同時実行スレッド数を超えたリクエストは，デフォルトの実行待ちキューに登録されます。デフォルトの実行待ちキューのサイズの上限を超えたリクエストは，ステータスコード 503（Service Unavailable）のエラーを返します。

(b)　Web コンテナ単位の同時実行スレッド数制御を設定する場合

Web コンテナ単位の同時実行スレッド数制御を設定している場合，リロード処理開始後に，Web コンテナ単位の同時実行スレッド数制御で実行できることになった，新規リクエストの処理スレッドが実行待ちとなります。

このため，リロード機能で実行待ちとなった新規リクエスト数が Web コンテナ単位の最大同時実行スレッド数まで達した場合，リロード処理を開始した Web アプリケーション以外の Web アプリケーションへのリクエストも，Web コンテナ単位の同時実行スレッド数制御機能によって実行待ちとなります。

(2)　リロードと J2EE アプリケーション実行時間の監視との関係

処理中のリクエストの処理が完了しない場合には，cjreloadapp コマンドで-t オプションを指定することで，処理中のリクエストを強制的に終了させて，リロード処理を開始できます。ただし，処理中のリクエストを強制的に終了できるのは，J2EE アプリケーション実行時間の監視機能のメソッドタイムアウトの適用対象メソッドだけです。また，メソッドタイムアウトの発生後にメソッドキャンセルが動作するように設定している場合は，タイムアウトの発生によって処理中のリクエストがキャンセルされて，リロードを開始できます。

J2EE アプリケーション実行時間の監視の詳細については，マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「5.3　J2EE アプリケーションの実行時間の監視とキャンセル」を参照してください。J2EE アプリケーション実行時間の監視の設定については，「13.8.12　J2EE アプリケーションの更新検知とリロードの設定」を参照してください。

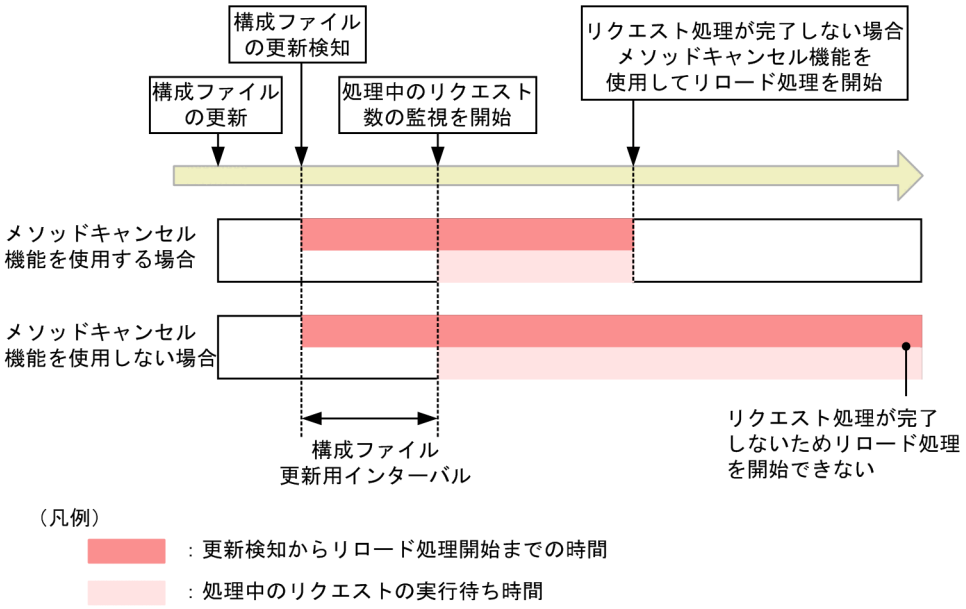
(a)　EJB アプリケーションの場合

EJB アプリケーションをリロードする場合について説明します。

構成ファイル更新用インターバルを経過すると，J2EE サーバは，処理中のリクエストの処理が完了するのを待ってリロード処理を開始します。ここで処理中のリクエストの処理が完了しないときは，J2EE アプリケーション実行時間の監視で，一定時間内に終了しなかったメソッド処理をタイムアウトとして通知し，メソッド処理をキャンセルすることで，リロード処理を開始できます。

メソッドキャンセル機能によるリロードを次の図に示します。

図 13-14　メソッドキャンセル機能によるリロード（EJB アプリケーションの場合）



構成ファイルの更新検知からリロード処理開始までの時間は次の式で求められます。

構成ファイル更新用インターバル（秒）＋処理中のリクエストの実行待ち時間（秒）

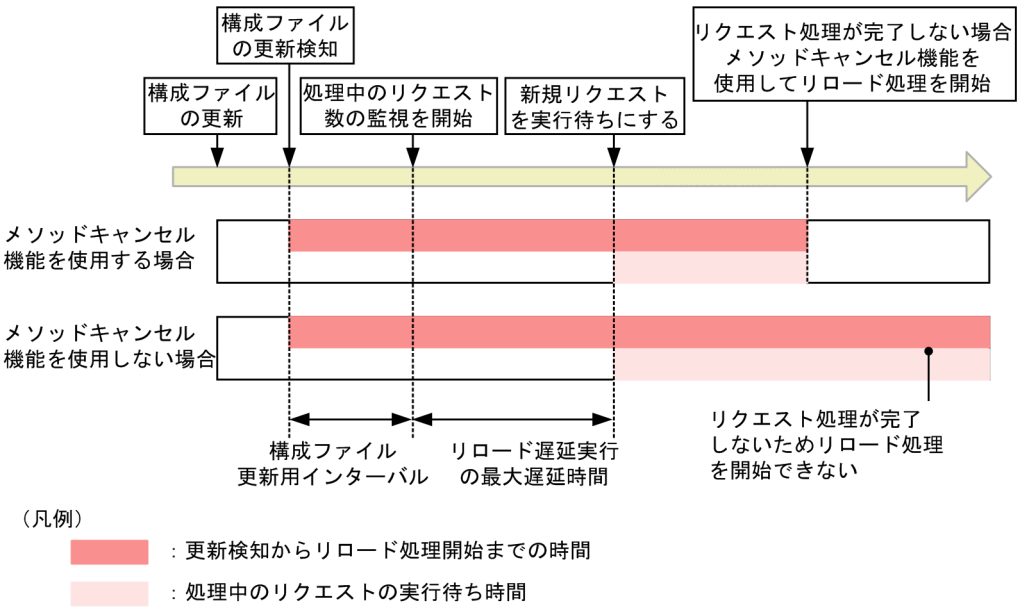
(b)　Web アプリケーションの場合

Web アプリケーションをリロードする場合について説明します。

リロード遅延実行を設定している場合，指定した最大遅延時間を経過すると，J2EE サーバは，新規リクエストを実行待ちにし，処理中のリクエストの処理が完了するのを待ってリロード処理を開始します。ここで処理中のリクエストの処理が完了しないときは，J2EE アプリケーション実行時間の監視で，一定時間内に終了しなかったメソッド処理をタイムアウトとして通知し，メソッド処理をキャンセルすることで，リロード処理を開始できます。

メソッドキャンセル機能によるリロードを次の図に示します。

図 13-15　メソッドキャンセル機能によるリロード（Web アプリケーションの場合）



構成ファイルの更新検知からリロード処理開始までの時間は次の式で求められます。

構成ファイル更新用インターバル（秒）＋リロード遅延実行の最大遅延時間（秒）＋処理中のリクエストの実行待ち時間（秒）

13.8.11　コマンドによる J2EE アプリケーションのリロード

ここでは，リロードによる J2EE アプリケーションの入れ替え方法について説明します。

リロードとは，少ない手順で展開ディレクトリ形式の J2EE アプリケーションの入れ替えを実行できる機能です。リロードによる J2EE アプリケーションの入れ替えでは，既存の J2EE アプリケーションの停止，削除，入れ替え後の J2EE アプリケーションのアーカイブ，インポート，再開などの作業が不要です。クラスファイルを更新してリロードを実行するだけで J2EE アプリケーションを更新できるため，メンテナンスが頻繁に発生するシステムの運用などで特に有効な機能です。

なお，リロードによる J2EE アプリケーションの入れ替えを実行するには，事前に設定が必要です。設定方法の詳細については，「13.8.12　J2EE アプリケーションの更新検知とリロードの設定」を参照してください。

リロードによる J2EE アプリケーションの入れ替えは，サーバ管理コマンド（cjreloadapp コマンド）を使用して実行できます。cjreloadapp コマンドの詳細については，マニュアル「アプリケーションサーバ アプリケーション設定操作ガイド」の「10.5.2　展開ディレクトリ形式のアプリケーション」を参照してください。

リロードは，次の手順で実行します。

1. メンテナンスの内容に従って Java ソースファイルを編集，または作成して，クラスファイルにコンパイルします。
2. J2EE アプリケーションのリロードを実行します。
   cjreloadapp コマンドを実行します。実行形式と実行例を次に示します。

実行形式

```
cjreloadapp 〈J2EEサーバ名〉 -name 〈J2EEアプリケーション名〉
```

実行例

```
cjreloadapp MyServer -name App1
```

! 注意事項

リロードに失敗したアプリケーションを削除するには，リロードの成功後，アプリケーションを停止，削除するか，J2EE サーバを再起動したあとにアプリケーションを削除してください。

13.8.12　J2EE アプリケーションの更新検知とリロードの設定

アプリケーション開発でのテストやシステムの運用中に，修正した J2EE アプリケーションと動作中の J2EE アプリケーションを入れ替えたい場合，リロード機能を使用した入れ替えができます。展開ディレクトリ形式の J2EE アプリケーションを構成するファイルを更新した場合に，更新検知やコマンド実行によって，更新した J2EE アプリケーションをリロードできます。リロード機能を使用することで，少ない手順で J2EE アプリケーションを動的に入れ替えられるようになります。

この項では，展開ディレクトリ形式の J2EE アプリケーションの更新検知とリロードをするための設定について説明します。コマンドやファイルのキーについては，マニュアル「アプリケーションサーバ リファレンス コマンド編」，およびマニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」を参照してください。

展開ディレクトリ形式の J2EE アプリケーションをリロードするために必要な設定は，構成ファイルの更新を検知してリロードするか，コマンドを実行してリロードするかによって異なります。展開ディレクトリ形式の J2EE アプリケーションをリロードするために必要な設定を次の表に示します。

表 13-17　リロードするために必要な設定

設定項目	更新検知によるリロードの場合	コマンドによるリロードの場合
リロード機能の適用範囲の設定	○	○
更新検知インターバルの設定	○	－
構成ファイル更新用インターバルの設定	○	－
リロード遅延実行の設定	○	－
セッション情報ファイル格納先ディレクトリの変更	△	△
JSP 事前コンパイルの設定	△	△
J2EE アプリケーション実行時間の監視の設定	△	△

（凡例）　○：設定する　△：必要に応じて設定する　－：該当しない

(1)　リロード機能の適用範囲の設定

リロード機能の適用範囲の設定は，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の <configuration>タグ内に指定します。

- ejbserver.deploy.context.reload_scope
  リロード機能を使用するかどうか，またリロードの対象を指定します。

- app：EJB アプリケーション（EJB-JAR）と Web アプリケーション（WAR）をリロードの対象とする。
- web：Web アプリケーション（WAR）だけをリロードの対象とする。
- jsp：JSP だけをリロードの対象とする。
- none：リロード機能を使用しない。

デフォルトの設定では，リロード機能は無効になっています。リロード機能を使用するためには，リロード機能を有効にして，適用範囲を設定する必要があります。

なお，リロード機能の有効／無効は，usrconf.properties の ejbserver.rmi.localinvocation.scope パラメタで指定するローカル呼び出し最適化機能の適用範囲と，リロード機能の適用範囲の組み合わせによって決まります。ローカル呼び出し最適化機能の適用範囲とリロード機能の適用範囲の対応については，「13.8.2　リロードの適用範囲」を参照してください。

(2)　更新検知インターバルの設定

更新検知インターバルの設定は，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に指定します。簡易構築定義ファイルでの更新検知インターバルの設定について次の表に示します。

表 13-18　簡易構築定義ファイルでの更新検知インターバルの設定

指定するパラメタ	設定内容
ejbserver.deploy.context.check_interval	J2EE アプリケーションの構成ファイルを監視して更新を検知する間隔（秒）を指定します。
webserver.context.check_interval	Web アプリケーションの構成ファイルを監視して更新を検知する間隔（秒）を指定します。
webserver.jsp.check_interval	JSP の構成ファイルを監視して更新を検知する間隔（秒）を指定します。

更新検知インターバルの設定値の関係を次に示します。

EJB アプリケーションの場合

ejbserver.deploy.context.check_interval の値が使用されます。なお，ejbserver.deploy.context.check_interval に 0 を指定している場合，EJB アプリケーションは更新検知されません。

サーブレットの場合

webserver.context.check_interval または ejbserver.deploy.context.check_interval の値が使用されます。優先順位を次に示します。

1. webserver.context.check_interval の値
2. ejbserver.deploy.context.check_interval の値

なお，webserver.context.check_interval を指定していない場合には，ejbserver.deploy.context.check_interval の値が使用されます。

また，webserver.context.check_interval に 0 を指定している場合，サーブレットは更新検知されません。

JSP の場合

webserver.jsp.check_interval または ejbserver.deploy.context.check_interval の値が使用されます。優先順位を次に示します。

1. webserver.jsp.check_interval の値
2. ejbserver.deploy.context.check_interval の値

なお，webserver.jsp.check_interval を指定していない場合には，ejbserver.deploy.context.check_interval の値が使用されます。

また，webserver.jsp.check_interval に 0 を指定している場合，JSP は更新検知されません。

(3)　構成ファイル更新用インターバルの設定

構成ファイル更新用インターバルは，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に指定します。簡易構築定義ファイルでの構成ファイル更新用インターバルの設定について次の表に示します。

表 13-19　簡易構築定義ファイルでの構成ファイル更新用インターバルの設定

指定するパラメタ	設定内容
ejbserver.deploy.context.update.interval	J2EE アプリケーションの構成ファイルのコピーに必要な時間（秒）を指定します。
webserver.context.update.interval	Web アプリケーションの構成ファイルのコピーに必要な時間（秒）を指定します。
webserver.jsp.update.interval	JSP の構成ファイルのコピーに必要な時間（秒）を指定します。

構成ファイル更新用インターバルの設定値の関係を次に示します。

EJB アプリケーションの場合

ejbserver.deploy.context.update.interval の値が使用されます。

サーブレットの場合

webserver.context.update.interval または ejbserver.deploy.context.update.interval の値が使用されます。優先順位を次に示します。

1. webserver.context.update.interval の値
2. ejbserver.deploy.context.update.interval の値

webserver.context.update.interval を指定していない場合には，ejbserver.deploy.context.update.interval の値が使用されます。

JSP の場合

webserver.jsp.update.interval または ejbserver.deploy.context.update.interval の値が使用されます。優先順位を次に示します。

1. webserver.jsp.update.interval の値
2. ejbserver.deploy.context.update.interval の値

webserver.jsp.update.interval を指定していない場合には，ejbserver.deploy.context.update.interval の値が使用されます。

(4)　リロード遅延実行の設定

リロード遅延実行は，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に次のパラメタを指定します。

- webserver.context.reload_delay_timeout
  Web アプリケーションをリロードする場合にリロード遅延実行を使用するかどうかを指定します。また，リロード遅延実行を使用する場合に，リロード処理を開始するまでの時間を設定するときは，最大遅延時間（秒）を指定します。

(5)　セッション情報ファイル出力先の変更

Web アプリケーションをリロードする場合は，リロード実行前に生成したセッション情報を引き継いで，リロード後も継続して利用します。セッション情報は，セッション情報ファイルに出力されます。

セッション情報ファイルの出力先

セッション情報ファイルは，デフォルトでは次の場所に出力されます。

- Windows の場合
  <Application Server のインストールディレクトリ>¥CC¥server¥repository¥<サーバ名>¥web¥<コンテキストルート名>¥cjwebsession.dat
- UNIX の場合
  /opt/Cosminexus/CC/server/repository/<サーバ名>/web/<コンテキストルート名>/cjwebsession.dat

Web アプリケーション単位のディレクトリ名

Web アプリケーション単位のディレクトリは，コンテキストルート名を基に規則に従ったディレクトリ名となります。コンテキストルート名にスラッシュ（/），ドル記号（$），パーセント（%），プラス記号（+）が含まれる場合は，次に示す文字に変換します。

変換前の文字	変換後の文字
/	$2f
$	$24
%	$25
+	$2b

出力先となるコンテキストルート名のディレクトリは，Web アプリケーションの開始時に作成されます。ただし，コンテキストルートがルートコンテキストの場合は，出力先ディレクトリのコンテキストルートは「$2f」として作成されます。作成されたディレクトリは，Web アプリケーションの終了時に削除されます。

セッション情報ファイルの出力先を変更する場合には，簡易構築定義ファイルで設定します。論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に，次のパラメタを設定します。

- ejbserver.deploy.session.work.directory
  Web アプリケーションのセッション情報ファイルの出力先を指定します。
  ejbserver.deploy.session.work.directory パラメタで指定したディレクトリ下に，「web¥<コンテキストルート名>」というディレクトリが作成されて，そのディレクトリ下にセッション情報ファイルが出力されます。

（例）セッション情報ファイルの出力先の設定例

```
<configuration>
  <logical-server-type>j2ee-server</logical-server-type>
  <param>
    <param-name>ejbserver.deploy.session.work.directory</param-name>
    <param-value>C:¥tmp¥session_work</param-value>
  </param>
   :
</configuration>
```

この設定例の場合，コンテキストルート名が「examples」の Web アプリケーションのセッション情報ファイルは，次の場所に出力されます。

C:¥tmp¥session_work¥web¥examples¥cjwebsession.dat

(6)　JSP 事前コンパイルの設定

JSP 事前コンパイルで JSP ファイルから生成されたクラスファイルを更新した場合も，J2EE アプリケーションの構成ファイルの更新を検知し，リロードが実行されます。

cjstartapp コマンドに-jspc オプションを指定して JSP 事前コンパイルを実行する場合には，usrconf.properties で JSP ワークディレクトリなどを設定しておく必要があります。JSP 事前コンパイルの設定については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(Web コンテナ)」の「2.5.8　実行環境での設定（J2EE サーバの設定)」を参照してください。

なお，アプリケーション開発時に cjjspc コマンドを使用して JSP 事前コンパイルを実行する場合には，JSP ワークディレクトリなどはコマンドのオプションで指定します。cjjspc コマンドを使用した JSP の事前コンパイルについては，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(Web コンテナ)」の「2.5　JSP 事前コンパイル機能とコンパイル結果の保持」を参照してください。

(7)　J2EE アプリケーション実行時間の監視の設定

構成ファイルの更新を検知した場合，処理中のリクエストの処理が完了するとリロード処理を開始しますが，処理中のリクエストの処理が完了しない場合には，J2EE アプリケーション実行時間の監視のメソッドタイムアウトおよびメソッドキャンセルを実施することで，リロード処理を開始できます。

必要に応じて，J2EE アプリケーション実行時間の監視を設定してください。J2EE アプリケーション実行時間の監視の設定については，マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「5.3.9　実行環境での設定」を参照してください。

13.8.13　リロードの注意事項および制限事項

リロードに関する注意事項および制限事項を次に示します。

(1)　リロード機能の使用に関する注意事項

- J2EE アプリケーションの削除について

  リロードに失敗している J2EE アプリケーションを削除する場合は，次のどちらかの方法で削除してください。

  - 更新検知によるリロード，または cjreloadapp コマンドによるリロードを再度実行してください。リロードが成功したら，J2EE アプリケーションを停止して削除してください。
  - J2EE サーバを再起動したあとで，J2EE アプリケーションを削除してください。

  削除後，J2EE アプリケーションを再インポートしてください。

- メモリ不足の発生について

リロード処理では，クラスローダを再作成してクラスをロードし直すため，メモリを多く消費します（消費するメモリ使用量は，J2EE アプリケーションの実装に依存します）。このため，例えば，複数の Web アプリケーションで同時にリロード処理を繰り返すとメモリ不足が発生する可能性があります。また，リロード処理によって不要となったリソースが JavaVM から解放されるタイミングは，JavaVM の GC の実行タイミングに依存します。リクエストのピーク時など，負荷が高いときにリロードを繰り返すと，GC を実行するタイミングがなく，メモリ不足が発生する可能性が高くなります。

このため，リクエストのピーク時など負荷が高い場合には，リソースの更新は避けてください。

- Metaspace 領域のメモリサイズについて

  リロードを使用する場合，Metaspace 領域に空きがない状態で新しいクラスをロードしようとすると，java.lang.OutOfMemoryError が発生します。そのため，リロード実行前に，Metaspace 領域に更新するアプリケーションが使用するサイズ以上の空き領域が確保されていることを確認する必要があります。十分な空き領域がない場合は，Metaspace 領域が不足しているおそれがあるので，Metaspace 領域の最大値を再設定してください。

  リロード実行時には，アプリケーションのクラスローダを破棄し，新しいクラスローダを生成します。そのとき，JavaVM の仕様とクラスローダの実装によって，破棄されたクラスローダは直ちに解放されないことがあります。その結果，リロード実行後は一時的にアプリケーションが必要とするサイズ以上に Metaspace 領域が占有されます。そのため，リロード機能が必要とする Metaspace 領域は，アプリケーションが必要とする Metaspace 領域の 3 倍程度に見積もることをお勧めします。

  Metaspace 領域の見積り式を次に示します。

  A ＝ B ＋ C ＋ （D×3） ＋ G

  - A：J2EE サーバが必要とする Metaspace 領域
  - B：コンテナが必要とする Metaspace 領域（120 メガバイト）
  - C：コンテナ拡張ライブラリが必要とする Metaspace 領域

    コンテナ拡張ライブラリに含まれる class ファイルのサイズの和となります。
  - G：JDK 提供の class ファイルのサイズ（90 メガバイト）
  - D：アプリケーションが必要とする Metaspace 領域

    クラスローダにロードされたクラスの class ファイルのサイズとなります。ただし，JAR ファイルに圧縮されている class ファイルは，解凍後のサイズで見積もる必要があります。また，アプリケーションに JSP が含まれている場合，JSP 事前コンパイルを実行して class ファイルを作成し，サイズを見積もる必要があります。見積もり式を次に示します。

    D ＝ E ＋ F ＋ H

    ・E：コンテナが作成するクラスの Metaspace 領域（アプリケーション開始後の Metaspace 領域－アプリケーション登録前の Metaspace 領域）。実際に J2EE サーバを起動し，Metaspace 領域を確認して算出してください。

    ・F：作成した J2EE アプリケーションの class ファイルのサイズ（J2EE アプリケーションを構成する class ファイルのサイズ＋ JAR に含まれる class ファイルのサイズ＋ JSP 事前コンパイルによって作成された class ファイルのサイズ）

    ・H：ライブラリ JAR，参照ライブラリを利用している場合に追加する JAR ファイルに含まれる class ファイルのサイズ

  Metaspace 領域の見積り例と変更例

  各 Metaspace 領域のメモリサイズが次の表に示す値の場合に，Metaspace 領域の最大値を変更する例を示します。

見積もり対象となる Metaspace 領域	メモリサイズ（単位：メガバイト）
コンテナが必要とする Metaspace 領域	120

見積もり対象となる Metaspace 領域	メモリサイズ（単位：メガバイト）
追加する JAR ファイルが必要とする Metaspace 領域	16
アプリケーションが必要とする Metaspace 領域	32
JDK 提供の class ファイルのサイズ	90

この例では，J2EE サーバが必要とする Metaspace 領域は，次の式で算出します。

120 ＋ 16 ＋（32×3）＋ 90 ＝ 322（メガバイト）

したがって，リロード機能を使用する場合は，割り当てる Metaspace 領域の最大値のデフォルト（128m）を，usrconf.cfg の add.jvm.arg キーで次のように変更する必要があります。

add.jvm.arg=-XX:MaxMetaspaceSize=322m

Metaspace 領域の確認方法

Metaspace 領域のメモリサイズは，JavaVM ログファイルを出力するか，またはリソース枯渇監視機能を使用するか，どちらかの方法で確認してください。

- **メソッドキャンセルの実行について**

処理中のリクエストの処理が完了しない場合には，J2EE アプリケーション実行時間の監視のメソッドタイムアウトおよびメソッドキャンセルを実行することで，リロード処理を開始できます。メソッドキャンセルを実行しても，処理をキャンセルできない場合には，J2EE サーバを再起動してください。

- **実行待ちとなっているリクエスト処理の再開について**

J2EE アプリケーションのリロードで EJB アプリケーションの開始，または停止に失敗した場合，Web アプリケーションのリロード処理開始以降の新規リクエストは実行待ちとなります。実行待ちとなっているリクエスト処理を再開するためには，EJB アプリケーションを更新してリロードを実行し，リロード処理を成功させてください。Web アプリケーションを更新しても，リロードは実行されません。

- **J2EE アプリケーションに含まれる RAR ファイルについて**

アプリケーションディレクトリに RAR ファイルを含めている場合，RAR ファイルはリロード機能の対象にはなりません。RAR ファイルを更新した場合は，更新した RAR ファイルを含む J2EE アプリケーションを，サーバ管理コマンドを使用して入れ替えてください。サーバ管理コマンドを使用して J2EE アプリケーションを入れ替える手順については，マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「5.6　J2EE アプリケーションの入れ替え」を参照してください。

- **CDI 機能を使用した J2EE アプリケーションのリロードについて**

J2EE アプリケーション内のコンポーネント（EJB-JAR，ライブラリ JAR，WAR）に CDI の機能を含む場合，更新検知によるリロードはできません。リロードを実行するには，cjreloadapp コマンドを使用してください。更新検知によるリロードが有効な設定の場合，CDI の機能を使用した J2EE アプリケーションの開始時に KDJE42393-W が出力され，更新検知によるリロードが無効であることが通知されます。

CDI の機能を使用した J2EE アプリケーションをリロードするには，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内で，ejbserver.deploy.context.reload_scope パラメタに「app」を設定してください。リロードを実行すると，更新したファイルが WAR ファイルまたは JSP ファイルの場合でも，J2EE アプリケーションすべてがリロードされます（WAR 単位，または JSP 単位でのリロードはできません）。このとき，更新したファイルが WAR ファイルまたは JSP ファイルの場合，KDJE42395-I が出力され，J2EE アプリケーションすべてがリロードされることが通知されます。

CDI の機能を使用した J2EE アプリケーションに対して cjreloadapp コマンドを実行してリロードした際に，簡易構築定義ファイルの論理 J2EE サーバ（j2ee-server）の<configuration>タグ内で，ejbserver.deploy.context.reload_scope パラメタに「web」または「jsp」を指定していた場合は，KDJE42394-E が出力され，リロードに失敗します。

CDI の機能を使用した J2EE アプリケーションに，cjreloadapp コマンドを実行してリロードした場合の実行結果を次の表に示します。

表 13-20　cjreloadapp コマンドの実行結果（CDI の機能を使用した J2EE アプリケーション）

更新したファイルを含むコンポーネント	ejbserver.deploy.context.reload.scope パラメタの値	cjreloadapp コマンドの実行結果
EJB-JAR またはライブラリ JAR	app	J2EE アプリケーションすべてがリロードされる。
	web	KDJE42394-E が出力され，コマンドの実行に失敗する。
	jsp	
WAR	app	KDJE42395-I が出力され，J2EE アプリケーションすべてがリロードされる。
	web	KDJE42394-E が出力され，コマンドの実行に失敗する。
	jsp	
JSP	app	KDJE42395-I が出力され，J2EE アプリケーションすべてがリロードされる。
	web	KDJE42394-E が出力され，コマンドの実行に失敗する。
	jsp	

- Inbound リソースアダプタについて

  アプリケーションのリロード中は，Inbound リソースアダプタからメッセージを送信しないでください。

- Message-driven Bean のリロードについて

  TP1 インバウンド連携機能使用時に Message-driven Bean をリロードする場合は，OpenTP1 からの RPC の発行を止めてからリロードしてください。リロード中に RPC を発行した場合は，Message-driven Bean は終了していると見なされ，RPC の発行元にはエラー応答が返ります。

(2)　EJB アプリケーション（EJB-JAR）の注意事項

- リロード後に EJB を呼び出す場合には，クライアント側のソースコードで EJBHome やビジネスインタフェースを取得し直す必要があります。リロード前に取得したインタフェース経由で EJB にアクセスした場合は例外が返ります。また，リロード実行中に EJB にアクセスした場合も例外が返ります。
- リモートインタフェースまたはリモートビジネスインタフェースを入れ替えた場合，クライアント側ではスタブを取得し直す必要があります。
- リロードを実行すると，Stateful Session Bean のクライアントとのセッションの状態は失われます。
- 非同期メソッドを展開ディレクトリ形式の J2EE アプリケーションでリロード機能を使用する場合，次の点に注意してください
  - 非同期メソッドが実行中の場合，非同期メソッドの処理が完了するまで，リロード処理は待機します。
  - リロード処理では，クリーンアップ処理が実行されるため，リロード処理前に取得した Future<V> オブジェクトによる操作は実行できなくなります。
- Singleton Session Bean を含む展開ディレクトリ形式の J2EE アプリケーションをリロードする場合，次の点に注意してください。

- @Startup アノテーションが指定された Singleton Session Bean は，アプリケーション開始時と同様に，リロード時にも初期化処理が実行されます。
- リロード処理の間，リロード対象の J2EE アプリケーション内の Singleton Session Bean のインスタンスはコンテナによって破棄されます。
- @DependsOn アノテーションの定義によって，リロードする Singleton Session Bean の依存関係を変更できます。ただし，循環依存があった場合も，リロード処理ではエラーになりません。このため，リロードの対象となる Singleton Session Bean では，循環依存を避けるため，@DependsOn アノテーションの使用を避けることをお勧めします。@DependsOn アノテーションを使用して，循環依存があった場合，動作は保証されません。
- Write ロックを取得待ちの状態のスレッドの処理は，リロード処理によって中断されません。

(3)　EJB アプリケーション（EJB-JAR）の制限事項

- これまで javax.ejb.TimedObject インタフェースを実装していなかった EJB で，javax.ejb.TimedObject インタフェースを実装するように変更して，リロードすることはできません。リロードした場合には，次のようになります。
  - 更新検知スレッドは停止させないで，J2EE アプリケーションを停止します。
  - 更新を検知した場合，停止した J2EE アプリケーションは再び開始されます。
- CMR を使用したアプリケーションはリロードできません。
- Singleton Session Bean の@DependsOn アノテーションの設定値を変更してリロードした場合，設定された依存関係が循環関係になっているときはリロードできません。

(4)　Web アプリケーション（WAR）の注意事項

- タグライブラリ記述子ファイルの listener タグの変更内容は，リロード後に反映されません。
- リロード時に利用できるサーブレットと JSP は，リロード後も状態を継続して利用できます。永久的に利用できないとされている場合は，リロード後も永久的に利用できません。一時的に利用できない場合は，リロードの実行とは関係なく，指定した期間後に利用できるようになります。
- Web アプリケーションでリロード監視対象になっているファイルを削除した場合，そのファイルは監視対象外となります。このため，再度ファイルを追加しても更新検知によるリロードは実行されません。ただし，/WEB-INF/lib 下に JAR ファイルを追加した場合は，その JAR ファイルが削除後に再度追加されたかどうかに関係なく，コマンドによるリロードまたは更新検知によるリロードを実行できます。
- JSP のリロード処理中に J2EE アプリケーションを停止した場合，または Web アプリケーションをリロードした場合，JSP のリロード処理は中止されます。
- JSP のリロード処理開始時，ログの出力レベルが Error の場合は，メッセージ KDJE39310-I に，更新されたファイル名とリロードを実行する JSP ファイルのファイル数が出力されます。ログの出力レベルが Warning の場合は，メッセージ KDJE39312-I に，リロードを実行するすべての JSP ファイル名が出力されます。
- リスナクラスのロード，インスタンス生成時に例外が発生した場合，Web コンテナは例外が発生したリスナを無効としてリロード処理を続行します。例外が発生したリスナを有効にするには，該当のリスナを修正して再度リロードを実行してください。
- フィルタクラスのロード，インスタンス生成，init メソッド実行で例外が発生した場合，Web コンテナは例外が発生したフィルタを無効としてリロード処理を続行します。例外が発生したフィルタを有効にするには，該当のフィルタを修正して再度リロードを実行してください。

- Web アプリケーションのリロード処理を実行すると，次の表に示すリスナインタフェースのメソッドが実行されます。

表 13-21　Web アプリケーションのリロード処理で実行されるリスナのメソッド

インタフェース名	メソッド名
javax.servlet.ServletContextListener	contextDestroyed()
	contextInitialized()
javax.servlet.ServletContextAttributeListener	attributeRemoved()
	attributeAdded()
javax.servlet.http.HttpSessionActivationListener※	sessionWillPassivate()
	sessionDidActivate()

注※　javax.servlet.http.HttpSession オブジェクトが存在しないときは実行しません。

- リロード処理時間は，Web アプリケーションの実装に大きく依存します。このため，次の点に注意が必要です。
  - セッション情報の容量は，シリアライズ処理時間，デシリアライズ処理時間に大きく影響します。シリアライズ対象となるセッション情報は，必要十分な容量にしてください。
  - リロード処理では，Web アプリケーションの終了処理，初期化処理が実行されます。終了処理，初期化処理が長い場合，リロードの処理時間も長くなり，Web アプリケーションのサービス停止期間にも影響があるため注意してください。ただし，リロード処理で初期化処理が実行されるのは，web.xml の<load-on-startup>タグを設定したサーブレットや JSP だけです。<load-on-startup>タグを設定していないサーブレットや JSP の初期化処理は，リロード完了後の初回アクセス時に実行されます。
- Web アプリケーションのリロード機能は，Web アプリケーション単位のクラスローダを入れ替えることによって，サーバ起動中の Web アプリケーションの入れ替えを実現しています。そのため，リロード後にリロード前の Web アプリケーション単位のクラスローダ，または Web アプリケーション単位のクラスローダでロードされたクラスが参照されるとメモリリークが発生します。

  次に，該当する場合を示します。リロード機能を使用する場合は，Web アプリケーション内でこれらのことをしないでください。
  - **スレッドの生成**

    スレッドはコンテキストクラスローダを保持しますが，デフォルトでは親スレッドのコンテキストクラスローダを保持します。Web アプリケーション実行時のスレッドは，Web アプリケーション単位のクラスローダをコンテキストクラスローダとしているため，生成されたスレッドは Web アプリケーション単位のクラスローダをコンテキストクラスローダとします。そのため，リロード後も該当スレッドが存在すると，リロード前の Web アプリケーション単位のクラスローダが解放されないでメモリリークが発生します。
  - **java.lang.Thread クラスまたはその派生クラスのインスタンス生成**

    java.lang.Thread クラスまたはその派生クラス（以降，スレッドクラス）のインスタンスを作成すると，所属するスレッドグループから該当スレッドクラスのインスタンスが参照されます。スレッドグループからの参照は，スレッドの実行を終了（run メソッドの完了）すると解放されます。そのため，Web アプリケーション内でスレッドクラスのインスタンスを生成し，run メソッドを実行しなかった場合は，該当スレッドクラスのインスタンスが解放されません。また，スレッドはコンテキストクラスローダとして Web アプリケーション単位のクラスローダを保持しているため，Web アプリケーション単位のクラスローダが解放されないでメモリリークが発生します。

- スレッドローカル変数の使用

  Web コンテナではリクエストを処理するスレッドはプールされていて，Web コンテナ終了時まで終了しません。そのため，スレッドローカル変数に Web アプリケーションに含まれるクラスのインスタンスを格納すると，メモリリークが発生します。

- JSP 事前コンパイル機能を使用している場合，JSP ファイルまたはタグファイルが依存するファイルは更新検知対象となったあと，次のどれかの条件を満たすときに更新検知対象から除外されます。
  - JSP ファイルまたはタグファイルを，依存するファイルを使用しないように変更したクラスファイルに更新し，依存するファイルを使用する JSP ファイルまたはタグファイルがほかにはない場合
  - JSP ファイルまたは JSP ファイルが依存するファイルから生成されたクラスファイルを更新し，JSP ファイルから生成されたクラスのロードでエラーが発生した際，依存するファイルを使用する JSP ファイルまたはタグファイルがほかにはない場合

  該当するファイルが更新検知対象から除外された場合，メッセージログに KDJE39319-I が出力されます。

  なお，cjjspc コマンドを実行すると依存するファイルを使用している JSP ファイルまたはタグファイルも再度コンパイルされます。クラスファイルを展開ディレクトリ以外で作成したあとに更新する場合は，cjjspc コマンドで生成されたクラスファイルをすべて更新してください。

- JSP 事前コンパイル機能を使用していない場合，JSP ファイルまたはタグファイルが依存するファイルは更新検知対象となったあと，次のどれかの条件を満たすときに更新検知対象から除外されます。
  - JSP ファイルまたはタグファイルを，依存するファイルを使用しないように修正し，コンパイルが正常に完了した際，依存するファイルを使用する JSP ファイルまたはタグファイルがほかにはない場合
  - JSP ファイル，または JSP ファイルが依存するファイルを更新し，JSP ファイルのコンパイルエラーが発生した際，依存するファイルを使用する JSP ファイルまたはタグファイルがほかにはない場合

  該当するファイルが更新検知対象から除外された場合，メッセージログに KDJE39318-I が出力されます。また，JSP ファイルのコンパイルエラーが発生し，かつ次の条件を満たすときは，更新検知対象から除外されたファイルだけを修正することで，コンパイルエラーとなった JSP ファイルを修正しないで，JSP のリロードを実行できます。

  - 依存するファイルがタグファイルの場合

    タグファイルの記述内容とタグファイルを使用するファイルの記述内容が矛盾している。タグファイルの次の属性のどれかが不正である。

    tag ディレクティブに定義した属性：body-content，または dynamic-attributes

    attribute ディレクティブに定義した属性：name，required，fragment，rtexprvalue，または type

    variable ディレクティブに定義した属性：name-given，variable-class，declare，または scope
  - 依存するファイルが web.xml の<include-prelude>，または<include-coda>に指定した暗黙的に include されるファイルの場合

    暗黙的に include されたファイルの記述内容が，include する JSP ファイルの記述内容と矛盾している。
  - 依存するファイルが include ディレクティブに指定したファイルの場合

    include ディレクティブに指定されたファイルの記述内容が，include ディレクティブを指定した JSP ファイルまたはタグファイルの内容と矛盾している。
  - 依存するファイルがタグライブラリ・ディスクリプタ（TLD）ファイルの場合

    TLD ファイルの記述内容と JSP ファイルの記述内容が矛盾している。

この場合，該当するファイルは更新検知の対象外となっているため，依存するファイルを更新したあと，次のどれかを実行し JSP のリロードを実行する必要があります。

- 依存するファイルを使用する JSP ファイルにブラウザなどからアクセスする。
- 依存するファイルを使用する JSP ファイルまたはタグファイルの更新日時を更新する。

- implicit.tld は更新検知対象ファイルではありません。implicit.tld とは，Servlet2.5 仕様で定義されたタグファイルのバージョンを示す TLD ファイルです。implicit.tld は JSP，またはタグファイルのリロード時に同時に再読み込みされます。
- Web アプリケーションのリロードでは，アノテーション情報は読み込みません。Web アプリケーションのクラスに定義したアノテーション情報だけを更新した場合，その更新情報を反映するには，J2EE アプリケーションをいったん停止して再度開始してください。

13.9　WAR アプリケーション

J2EE サーバでは，WAR ファイルまたは WAR ディレクトリを指定してインポートできます。インポートしたアプリケーションを WAR アプリケーションといいます。WAR アプリケーションは J2EE アプリケーションとしてデプロイできます。

WAR アプリケーションの機能と参照先を次の表に示します。

表 13–22　この節の構成（WAR アプリケーション）

機能	参照先
WAR アプリケーション操作コマンドのサポート範囲	13.9.1
実行できる WAR アプリケーションの形式	13.9.2
WAR アプリケーションの入れ替え	13.9.3
WAR アプリケーションのアプリケーション名	13.9.4
コンテキストルートの決定	13.9.5
cosminexus.xml ファイルの読み込み	13.9.6

13.9.1　WAR アプリケーション操作コマンドのサポート範囲

J2EE アプリケーションに対して実行できる操作コマンドのうち，WAR アプリケーションで実行できる操作コマンドには制限があります。

WAR アプリケーションに対する操作コマンドのサポート範囲を次の表に示します。

表 13–23　WAR アプリケーションに対する操作コマンドのサポート範囲

操作コマンド	サポート有無	実行結果
cjaddapp	×	KDJE42400-E のメッセージが出力されます。
cjchmodapp	×	KDJE42400-E のメッセージが出力されます。
cjdeleteapp	△	-type war，または-type filter を指定してコマンドを実行した場合，KDJE42401-E のメッセージが出力されます。そのほかの場合は，J2EE アプリケーションに対して実行した場合と同じです。
cjdeletelibjar	×	−
cjexportapp	×	KDJE42400-E のメッセージが出力されます。
cjgencmpsql	×	−
cjgetappprop	○	J2EE アプリケーションに対して実行したときと同じ。
cjgetstubsjar	×	−
cjimportapp	×	−
cjimportwar	○	WAR ディレクトリまたは WAR ファイルをインポートします。

操作コマンド	サポート有無	実行結果
cjimportlibjar	×	KDJE42400-E のメッセージが出力されます。
cjlistapp	○	WAR アプリケーションの場合，コンテキストルートが表示されます。
cjreloadapp	○	J2EE アプリケーションに対して実行したときと同じ。
cjrenameapp	×	KDJE42400-E のメッセージが出力されます。
cjreplaceapp	△	cosminexus.xml の再読み込みができません※。
cjsetappprop	○	J2EE アプリケーションに対して実行したときと同じ。
cjstartapp	○	J2EE アプリケーションに対して実行したときと同じ。
cjstopapp	○	J2EE アプリケーションに対して実行したときと同じ。
cjlistlibjar	×	－
cjlistpool	×	－
cjtestres	×	－
cjclearpool	×	－

（凡例）○：サポートあり　△：一部サポートあり　×：サポートなし　－：該当なし

注※　アプリケーションの属性を変更したい場合は，サーバ管理コマンド（cjgetappprop コマンドおよび cjsetappprop コマンド）を使用してください。

● WAR アプリケーションに対する操作の制限

WAR アプリケーションでは，次の操作は実行できません。

- リソースの追加・削除
  - WAR アプリケーションへの EJB-JAR／WAR／RAR の追加
  - WAR アプリケーションから WAR の削除
  - WAR アプリケーションへのライブラリ JAR の追加
  - WAR アプリケーションへのフィルタの追加
  - WAR アプリケーションからフィルタの削除
- テストモード
- アプリケーションのエクスポート
- アプリケーション名の変更

参考

リソースを追加・削除する場合，WAR アプリケーションを削除して，EAR ファイルの J2EE アプリケーションを作成してください。

13.9.2　実行できる WAR アプリケーションの形式

WAR アプリケーションは，アーカイブ形式または展開ディレクトリ形式でインポートできます。アプリケーションの形式については，「13.2　実行できる J2EE アプリケーションの形式」を参照してください。

アーカイブ形式の WAR アプリケーション

インポートする WAR ファイルは，cjimportwar コマンドを実行するユーザの読み込み権限が必要です。

なお，インポート時に指定した WAR ファイル名は作業ディレクトリ中のディレクトリ名として用いられます。作業ディレクトリのパス長がプラットフォームの上限に達しないように WAR ファイル名を指定してください。作業ディレクトリのパス長の見積もりについては，マニュアル「アプリケーションサーバ システム構築・運用ガイド」の「付録 C.1　J2EE サーバの作業ディレクトリ」を参照してください。

展開ディレクトリ形式の WAR アプリケーション

展開ディレクトリ形式の WAR アプリケーションは，WAR ディレクトリというルートディレクトリを作成して，展開ディレクトリ形式のアプリケーションとしてインポートします。WAR アプリケーションをインポートすると，WAR ディレクトリ下に Web アプリケーションなどの構成要素が格納されます。なお，展開ディレクトリ形式の WAR アプリケーションはリモート環境では実行できません。

● 展開ディレクトリ形式を使用する場合の注意事項

展開ディレクトリ形式を使用する場合の注意事項を次に示します。

- 展開ディレクトリ形式の WAR アプリケーションをインポートする場合に，WAR ディレクトリのパスとして UNC 名を含むパスは指定できません。UNC 名を含むパスを指定した場合，コマンドの実行エラーになります。
- Windows のドライブ直下（C:¥など）のディレクトリ，または UNIX のルートディレクトリ（/）を示すパス（相対パスも含む）を WAR ディレクトリに指定できません。cjimportwar コマンドの-a オプションにドライブ直下のディレクトリ，またはルートディレクトリを示すパスを WAR ディレクトリとして指定した場合，コマンドの実行エラーになります。
- cjimportwar コマンドの-a オプションに指定したディレクトリと同一のディレクトリを持つ WAR アプリケーションが，すでに J2EE サーバ内にある場合，インポートでコマンドの実行エラーになります。
- 次のディレクトリをアプリケーションディレクトリまたは WAR ディレクトリとして持つ J2EE アプリケーションが，すでに J2EE サーバ内に存在する場合，インポートでコマンドの実行エラーになります。
  - cjimportwar コマンドの-a オプションに指定したディレクトリの上位にあるディレクトリ
  - cjimportwar コマンドの-a オプションに指定したディレクトリの下位にあるディレクトリ

13.9.3　WAR アプリケーションの入れ替え

WAR アプリケーションは既存の J2EE アプリケーション（EAR 形式）と同じように，リデプロイ機能を使用して，開発環境にある WAR ファイルをすでに運用中の WAR アプリケーションと入れ替えることができます。J2EE アプリケーションの入れ替えについては，「13.7　J2EE アプリケーションのリデプロイ」を参照してください。

! 注意事項

WAR アプリケーションをリデプロイで入れ替える場合, cosminexus.xml ファイルの再読み込みはできません。WAR アプリケーションのアプリケーション属性を変更する場合は，サーバ管理コマンド（cjgetappprop コマンドおよび cjsetappprop コマンド）を使用してください。

13.9.4　WAR アプリケーションのアプリケーション名

アプリケーション名を cjimportwar コマンドの-name オプションで指定します。-name オプションに指定できる文字列は, application.xml の<application>タグ下の<display-name>タグに指定できる文字列

と同じです。使用できる文字については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjimportwar（WAR アプリケーションのインポート）」を参照してください。使用できない文字が使用されている場合，KDJE37206-E メッセージが出力されます。

cjimportwar コマンドに-name オプションを指定していない場合，J2EE サーバは WAR ファイル名（アーカイブ形式の場合），または WAR ディレクトリ名（展開ディレクトリ形式の場合）を基に，使用できない文字をアンダースコア(_)に置き換えてアプリケーション名を設定します。なお，J2EE サーバが設定したアプリケーション名が J2EE サーバ内で重複する場合，アプリケーション名が J2EE サーバ内で一意となるように，そのアプリケーション名の末尾に通し番号（1～2147483647）が追加されます。

アプリケーション名の設定例を次に示します。

表 13–24　アプリケーション名の設定例（アーカイブ形式の WAR アプリケーションの場合）

cjimportwar コマンドの-name オプション	WAR ファイル名	アプリケーション名
SampleTP	SampleWeb.war	SampleTP
（指定なし）	SampleWeb.war	SampleWeb

表 13–25　アプリケーション名の設定例（展開ディレクトリ形式の WAR アプリケーションの場合）

cjimportwar コマンドの-name オプション	WAR アプリケーションのディレクトリ名	アプリケーション名
SampleTP	SampleWeb	SampleTP
（指定なし）	SampleWeb	SampleWeb

13.9.5　コンテキストルートの決定

コンテキストルートを cjimportwar コマンドの-contextroot オプションで指定します。-contextroot オプションは，URI（RFC3986）で使用できる文字を指定してください。URI（RFC3986）で使用できる文字以外，または ejb/，web/，/ejb/，および/web/から始まる文字列を指定した場合，KDJE37206-E メッセージが出力されます。cjimportwar コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjimportwar（WAR アプリケーションのインポート）」を参照してください。

cjimportwar コマンドに-contextroot オプションを指定していない場合，J2EE サーバは WAR ファイル名（アーカイブ形式の場合），または WAR ディレクトリ名（展開ディレクトリ形式の場合）を基に，次のようにコンテキストルートを決定します。

- 拡張子を取り除いた文字列が ejb，または web だったときは，それぞれ ejb1，web1 に変換します。
- ejb または web 以外の文字列のときは，使用できない文字をアンダースコア(_)に置き換えてコンテキストルートを決定します。

コンテキストルートの決定例を次に示します。

表 13–26　コンテキストルートの決定例（アーカイブ形式の WAR アプリケーションの場合）

cjimportwar コマンドの-contextroot オプション	WAR ファイル名	コンテキストルート
example/Servlet	SampleWeb.war	example/Servlet

cjimportwar コマンドの-contextroot オプション	WAR ファイル名	コンテキストルート
（指定なし）	SampleWeb.war	SampleWeb

表 13–27　コンテキストルートの決定例（展開ディレクトリ形式の WAR アプリケーションの場合）

cjimportwar コマンドの-contextroot オプション	WAR アプリケーションのディレクトリ名	コンテキストルート
example/Servlet	SampleWeb	example/Servlet
（指定なし）	SampleWeb	SampleWeb

cjlistapp コマンドを実行すると，表示される文字列にコンテキストルートが表示されます。cjlistapp コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjlistapp (アプリケーションの一覧表示)」を参照してください。

13.9.6　cosminexus.xml ファイルの読み込み

読み込む cosminexus.xml ファイルを cjimportwar コマンドの-c オプションで指定します。-c オプションを省略した場合，cosminexus.xml ファイルは読み込まれません。cjimportwar コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「cjimportwar（WAR アプリケーションのインポート)」を参照してください。

cosminexus.xml を含むアプリケーションの運用については，「11.3.6　cosminexus.xml を含むアプリケーションの運用」を参照してください。

14 コンテナ拡張ライブラリ

この章では，コンテナ拡張ライブラリの機能について説明します。

アプリケーションサーバでは，Enterprise Bean やサーブレットから共通して使用するライブラリを，コンテナ拡張ライブラリとして利用できます。

14.1　この章の構成

コンテナ拡張ライブラリの機能と参照先を次の表に示します。

表 14–1　コンテナ拡張ライブラリの機能と参照先

機能	参照先
コンテナ拡張ライブラリの利用	14.2
コンテナ拡張ライブラリの機能	14.3
サーバ起動・停止フック機能	14.4
スマートエージェント経由での CORBA オブジェクトの呼び出し	14.5
コンテナ拡張ライブラリおよびサーバ起動・停止フック機能利用時の制限事項	14.6

14.2　コンテナ拡張ライブラリの利用

この節では，コンテナ拡張ライブラリおよびサーバ起動・停止フック機能の概要について説明します。

アプリケーションサーバでは，EJB-JAR と WAR 間で共通に利用したい処理がある場合や，異なる EAR 間で共通に利用したい処理がある場合に，ユーザ作成のライブラリを利用できます。ユーザ作成のライブラリを利用することで，サーブレット，JSP，および Enterprise Bean の機能を拡張できます。

サーブレット，JSP，および Enterprise Bean が共通に利用できるライブラリを**コンテナ拡張ライブラリ**といいます。このライブラリを利用することで，Enterprise Bean，サーブレット，JSP から共通して，ユーザ作成のライブラリを呼び出せるようになります。

また，**サーバ起動・停止フック機能**を利用することで，サーバの起動，終了時にコンテナ拡張ライブラリが呼び出されるようにできます。また，コンテナ拡張ライブラリで使用する JNI 機能の初期化などを行うことができます。

コンテナ拡張ライブラリを使用するためには，ライブラリを一つの JAR ファイルにまとめ，コンテナ拡張ライブラリを使用するための設定を usrconf.cfg で定義します。また，コンテナ拡張ライブラリが JNI を利用する場合は，サーバ起動・停止フック機能を使用するための設定も必要です。

コンテナ拡張ライブラリの設定については，「14.3.3　コンテナ拡張ライブラリの機能を使用するための設定」を参照してください。

14.3　コンテナ拡張ライブラリの機能

この節では，コンテナ拡張ライブラリの機能について説明します。

この節の構成を次の表に示します。

表 14–2　この節の構成（コンテナ拡張ライブラリの機能）

分類	タイトル	参照先
解説	コンテナ拡張ライブラリ利用の検討	14.3.1
	コンテナ拡張ライブラリの作成と利用の流れ	14.3.2
設定	コンテナ拡張ライブラリの機能を使用するための設定	14.3.3

注 1　「実装」および「運用」について，この機能固有の説明はありません。

注 2　コンテナ拡張ライブラリを利用する場合の制限事項については，「14.6　コンテナ拡張ライブラリおよびサーバ起動・停止フック機能利用時の制限事項」を参照してください。

14.3.1　コンテナ拡張ライブラリ利用の検討

処理の種類に応じて検討する方法と，処理の内容に応じて検討する方法について説明します。

(1)　処理の種類による検討

まず，処理を次の 3 種類に分類し，コンテナ拡張ライブラリを利用するかどうかを検討します。コンテナ拡張ライブラリを利用しない場合は，EJB-JAR ファイル，WAR ファイル，またはライブラリ JAR に共通のライブラリを含めます。

- **業務処理**

  業務ごとに処理が異なるため，EJB-JAR ファイル，または WAR ファイルに含めます。コンテナ拡張ライブラリを利用する必要はありません。

- **EJB-JAR ファイルおよび WAR ファイル間の共通処理**

  複数の EJB-JAR ファイルや WAR ファイルに含まれる，Enterprise Bean，サーブレット，JSP，業務処理が共通に利用できる処理がある場合，ライブラリ JAR を利用します。ライブラリ JAR を利用できない場合は，共通処理のクラスを作成して，コンテナ拡張ライブラリを利用します。

- **EAR 間の共通処理**

  複数の EAR に含まれる，EJB-JAR，WAR が共通に利用できる処理がある場合，ライブラリ JAR を利用します。ライブラリ JAR を利用できない場合は，共通処理のクラスを作成して，コンテナ拡張ライブラリを利用します。

(2)　処理の内容による検討

Enterprise Bean，サーブレット，JSP での操作内容を次のように分類し，それぞれコンテナ拡張ライブラリを使用するかどうかを検討します。コンテナ拡張ライブラリを利用するかどうかの指針を次の表に示します。

表 14–3　操作内容ごとのコンテナ拡張ライブラリの利用の指針

操作内容	コンテナ拡張ライブラリ	EJB-JAR ファイルまたは WAR ファイル	ライブラリ JAR
ファイルやディレクトリへのアクセス操作	○	×	×
JNI の利用による操作※	○	×	×
スマートエージェント経由の CORBA オブジェクトの呼び出し	○	×	×
J2EE コンテナ機能の操作	×	○	×
EJB-JAR ファイルおよび WAR ファイル内のクラス参照	×	−	×

（凡例）　○：含める　　×：含めない　　−：該当しない

注※　J2EE アプリケーションまたは Web アプリケーションで同一のネイティブライブラリをロードしようとした場合，JNI の仕様によって UnsatisfiedLinkError がスローされます。これは，アプリケーションで共通に使用するネイティブライブラリをコンテナ拡張ライブラリとして登録することで，回避できます。

! 注意事項

コンテナ拡張ライブラリには，次のアクセス権が付与されます。アクセス権は変更できません。

java.security.AllPermission

ただし，java.lang.RuntimePermission の setSecurityManager アクセス権は付与されません。

14.3.2　コンテナ拡張ライブラリの作成と利用の流れ

コンテナ拡張ライブラリの作成と利用の流れを次に示します。

1. **ユーザ作成のクラスを実装，コンパイルします。**

   サーバ起動・停止フック機能を使用する場合は，<Application Server のインストールディレクトリ>¥CC¥lib¥ejbserver.jar をクラスパスに指定してコンパイルします。なお，サーバ起動・停止フック機能を使用する場合は，IDE を使用しない方法で実装，コンパイルしてください。

   また，サーバ起動・停止フック機能では，TPBroker が提供するスマートエージェント経由の CORBA オブジェクトを呼び出す処理を実装することもできます。

2. **作成したクラスを，JAR ファイルにアーカイブします。**

   ユーザが作成したクラスを，コンテナ拡張ライブラリ用の JAR ファイルにアーカイブします。EJB-JAR ファイルや WAR ファイルには含めないでください。

3. **アーカイブした JAR ファイルを，J2EE サーバのシステムクラスパスに指定します。**

コンテナ拡張ライブラリを利用する場合，この手順のほかに J2EE サーバでの設定（定義ファイルの指定など）が必要になります。

J2EE サーバでの設定については，「14.3.3　コンテナ拡張ライブラリの機能を使用するための設定」を参照してください。

14.3.3　コンテナ拡張ライブラリの機能を使用するための設定

ここでは，コンテナ拡張ライブラリを使用するための設定について説明します。なお，コンテナ拡張ライブラリが JNI を利用する場合は，サーバ起動・停止フック機能を使用します。

コンテナ拡張ライブラリを使用するためには，次の設定が必要です。

1. コンテナ拡張ライブラリ用の JAR ファイルを作成します。
   コンテナ拡張ライブラリの利用については，「14.2　コンテナ拡張ライブラリの利用」を参照してください。
2. 簡易構築定義ファイルで，論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に，次のパラメタを指定します。
   - add.class.path
     add.class.path の設定値には，手順 1.で作成した JAR ファイルのパスを J2EE サーバのシステムクラスパスを指定します。

   add.class.path は，簡易構築定義ファイルの J2EE サーバの拡張パラメタに設定します。
3. コンテナ拡張ライブラリから JNI 機能を使用する場合は，簡易構築定義ファイルで，論理 J2EE サーバ（j2ee-server）の<configuration>タグ内に，次のパラメタを指定します。
   - add.library.path
     add.library.path の設定値には，JNI 用ライブラリの検索パスを指定します。
   - ejbserver.application.InitTermProcessClasses
     ejbserver.application.InitTermProcessClasses の設定値には，サーバ起動・停止フック機能のクラス名を指定します。

   add.library.path および ejbserver.application.InitTermProcessClasses は，簡易構築定義ファイルの，J2EE サーバの拡張パラメタに設定します。
4. サーバ管理コマンド用のファイルのキーにコンテナ拡張ライブラリ用の JAR ファイルを指定します。
   指定するファイルおよびキーは，OS によって異なります。
   - Windows の場合
     usrconf.bat の USRCONF_JVM_CLASSPATH キー
   - UNIX の場合
     usrconf の USRCONF_JVM_CLPATH キー
5. コンテナ拡張ライブラリから JNI 機能を使用する場合は，サーバ管理コマンド用のファイルのキーで，JNI 用ライブラリの検索パスを指定します。
   複数指定する場合は，セミコロン（;）で区切ってください。
   指定するファイルは，OS によって異なります（キーは共通です）。
   - Windows の場合
     usrconf.bat の USRCONF_JVM_LIBPATH キー
   - UNIX の場合
     usrconf の USRCONF_JVM_LIBPATH キー

ファイルの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.6　簡易構築定義ファイル」およびマニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「5.3　usrconf.bat（サーバ管理コマンド用オプション定義ファイル)」を参照してください。

簡易構築定義ファイルおよびユーザ定義ファイルの設定例を次に示します。この例では，コンテナ拡張ライブラリの JAR ファイルは「extended_container.jar」で，そのコンテナ拡張ライブラリが JNI を使用して「extended_container.dll」（UNIX の場合，extended_container）を呼び出します。

- Windows の場合

- 簡易構築定義ファイルの設定例

```
<configuration>
  <logical-server-type>j2ee-server</logical-server-type>
  <param>
    <param-name>add.class.path</param-name>
    <param-value>c:¥jar¥extended_container.jar</param-value>
    <param-name>add.library.path</param-name>
    <param-value>c:¥lib</param-value>
  </param>
   :
</configuration>
```

- サーバ管理コマンド用の usrconf.bat の設定例

```
rem system classpath
set USRCONF_JVM_CLASSPATH=c:¥jar¥extended_container.jar

rem library path
set USRCONF_JVM_LIBPATH=c:¥lib
```

- UNIX の場合

  - 簡易構築定義ファイルの設定例

```
<configuration>
  <logical-server-type>j2ee-server</logical-server-type>
  <param>
    <param-name>add.class.path</param-name>
    <param-value>/work/classes/extended_container.jar</param-value>
    <param-name>add.library.path</param-name>
    <param-value>/work/lib</param-value>
  </param>
   :
</configuration>
```

  - サーバ管理コマンド用の usrconf の設定例

```
#!/bin/csh -f
# system classpath
set USRCONF_JVM_CLPATH=/work/classes/extended_container.jar

# library path
set USRCONF_JVM_LIBPATH=/work/lib
```

14.4　サーバ起動・停止フック機能

この節では，サーバ起動・停止フック機能について説明します。

この節の構成を次の表に示します。

表 14-4　この節の構成（サーバ起動・停止フック機能）

分類	タイトル	参照先
解説	サーバ起動・停止フック処理の呼び出し順序	14.4.1
実装	サーバ起動・停止フック機能の実装方法	14.4.2
設定	サーバ起動・停止フック機能利用時のクラスパスの指定	14.4.3

注 1　「運用」について，この機能固有の説明はありません。

注 2　サーバ起動・停止フック機能を利用する場合の制限事項については，「14.6　コンテナ拡張ライブラリおよびサーバ起動・停止フック機能利用時の制限事項」を参照してください。

14.4.1　サーバ起動・停止フック処理の呼び出し順序

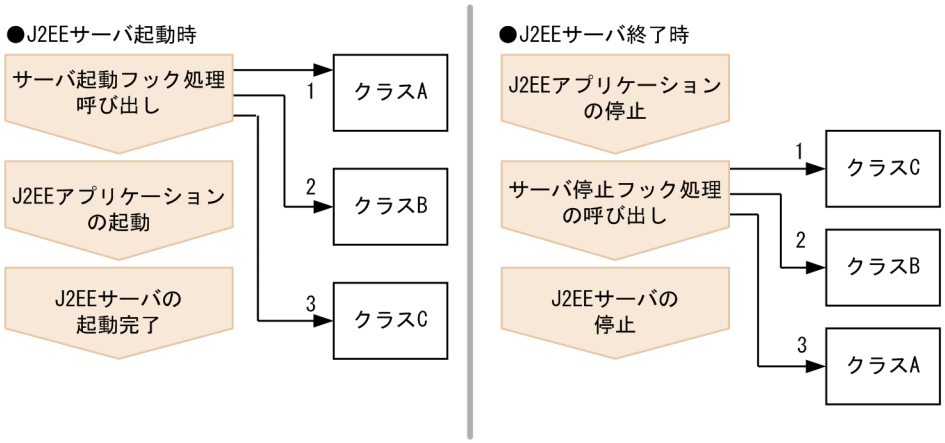
サーバ起動・停止フック機能を複数登録した場合の，サーバ起動・停止フック処理の呼び出し順序について説明します。

(1)　正常に起動，停止した場合

J2EE サーバ起動時には，登録されたクラス順にサーバ起動フック処理を行う serverInitializing メソッドが呼び出されます。また，J2EE サーバ停止時には，登録された逆順にサーバ停止フック処理を行う serverTerminating メソッドが呼び出されます。

J2EE サーバが正常起動した場合の，サーバ起動・停止フック機能の呼び出し順序を次の図に示します。

図 14-1　サーバ起動・停止フック処理の呼び出し順序（正常起動時）

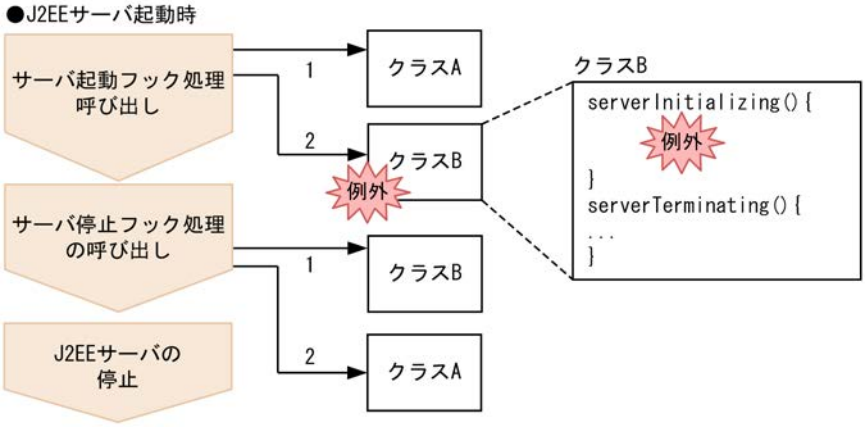


サーバ起動・停止フック処理を実装した A，B，C の三つのクラスがあり，クラス A，クラス B，クラス C の順でクラスが登録されていることとします。サーバ起動フック処理では，正常に呼び出されると，クラス A，クラス B，クラス C の順でクラスが呼び出されます。サーバ停止フック処理では，クラス C，クラス B，クラス A の順でクラスが呼び出されます。

(2)　サーバ起動フック処理中に例外が発生した場合

サーバ起動フック処理の実行中に例外が発生した場合の，サーバ起動・停止フック機能の呼び出し順序を次の図に示します。

図 14–2　サーバ起動・停止フック処理の呼び出し順序（起動フック処理中の例外発生時）

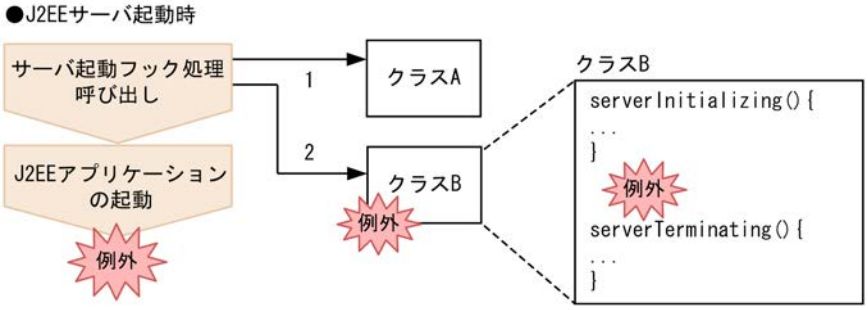


サーバ起動・停止フック処理を実装した A，B，C の三つのクラスがあり，クラス A，クラス B，クラス C の順でクラスが登録されていることとします。クラス B のサーバ起動フック処理（serverInitializing メソッド）の実行中に例外が発生した場合は，クラス C のサーバ起動フック処理は呼び出されません。また，クラス C のサーバ停止フック処理（serverTerminating メソッド）は呼び出されないで，クラス B，クラス A の順序でサーバ停止フック処理が呼び出されます。

(3)　サーバ起動フック処理後に例外が発生した場合

サーバ起動フック処理の実行後に例外が発生した場合の，サーバ起動・停止フック機能の呼び出し順序を次の図に示します。

図 14–3　サーバ起動・停止フック処理の呼び出し順序（起動フック処理後の例外発生時）



サーバ起動・停止フック処理を実装した A，B，C の三つのクラスがあり，クラス A，クラス B，クラス C の順でクラスが登録されていることとします。クラス B のサーバ起動フック処理（serverInitializing メソッド）の実行後に例外が発生した場合は，クラス C のサーバ起動フック処理は呼び出されません。また，サーバ停止フック処理は呼び出されません。

14.4.2　サーバ起動・停止フック機能の実装方法

サーバ起動・停止フック機能は，com.hitachi.software.ejb.application.InitTermProcess インタフェースを実装することで利用できます。サーバ起動フック処理は serverInitializing メソッド，停止フック処理は serverTerminating メソッドに実装します。次に，InitTermProcess インタフェースの実装例を示します。

```
package sample;

import com.hitachi.software.ejb.application.InitTermProcess;
import com.hitachi.software.ejb.application.InitTermException;

public class AppInitTerm implements InitTermProcess {
    public void serverInitializing() throws InitTermException {
        try {
            // サーバ起動フック処理
        } catch (Exception e) {
            throw new InitTermException("詳細メッセージ");
        }
    }
    public void serverTerminating() throws InitTermException {
        try {
            // サーバ停止フック処理
        } catch (Exception e) {
            throw new InitTermException("詳細メッセージ");
        }
    }
}
```

J2EE サーバは，起動時にデフォルトコンストラクタを使用してサーバ起動・停止フック機能のインスタンスを生成します。このため，クラスおよびデフォルトコンストラクタのアクセス指定子には，public を指定してください。

サーバ停止フック処理が呼び出されるタイミングは，アプリケーションの停止処理後になります。J2EE サーバでは，スレッドの停止処理を行っていないため，処理スレッドは残りますが，サーバ停止フック処理後に，新たにアプリケーションは実行されません。

ユーザ定義ファイル（usrconf.properties）のサーバ起動・停止フック機能用のプロパティキー（ejbserver.application.InitTermProcessClasses）に，サーバ起動・停止フックのクラス名を指定します。また，サーバ起動・停止フックのクラス名は複数指定できます。ユーザ定義ファイル（usrconf.properties）については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」を参照してください。

14.4.3　サーバ起動・停止フック機能利用時のクラスパスの指定

サーバ起動・停止フック機能を利用する場合，JAR ファイルのパスを，J2EE サーバのシステムクラスパスに追加する必要があります。システムクラスパスの追加は，ユーザ定義ファイル（usrconf.cfg）に指定します。

システムクラスパスの追加の方法については，「14.3.3　コンテナ拡張ライブラリの機能を使用するための設定」を参照してください。また，ユーザ定義ファイル（usrconf.cfg）については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」を参照してください。

14.5　スマートエージェント経由での CORBA オブジェクトの呼び出し

コンテナ拡張ライブラリを利用することで，J2EE アプリケーションまたは Web アプリケーションが TPBroker を使用して，スマートエージェント経由で CORBA オブジェクトを呼び出すことができます。スマートエージェントを利用して CORBA オブジェクトのオブジェクトリファレンスを取得し，メソッドを呼び出すときには，スマートエージェントが起動済みとなっており，スマートエージェントに，該当する CORBA オブジェクトのオブジェクトリファレンスが登録されている必要があります。また，コンテナ拡張ライブラリには，J2EE アプリケーションまたは Web アプリケーションから CORBA オブジェクトを呼び出す処理を含めます。

この節では，スマートエージェント経由での CORBA オブジェクトの呼び出しについて説明します。

この節の構成を次の表に示します。

表 14–5　この節の構成（スマートエージェント経由での CORBA オブジェクトの呼び出し）

分類	タイトル	参照先
実装	CORBA オブジェクト呼び出し処理の実装時の注意事項	14.5.1
	CORBA オブジェクト呼び出し処理のパッケージ化の注意事項	14.5.2

注　「解説」,「設定」, および「運用」について，この機能固有の説明はありません。

次に，CORBA オブジェクト呼び出し処理の実装時の注意事項，および CORBA オブジェクト呼び出し処理のパッケージ化の注意事項を示します。

14.5.1　CORBA オブジェクト呼び出し処理の実装時の注意事項

CORBA オブジェクトの呼び出し処理を実装するときには，次の点に注意してください。

- プロパティを使用して，org.omg.ORB をカスタマイズするには，org ORB init(String[] args, Properties props)メソッドの props パラメタにプロパティを指定してください。
  なお，J2EE サーバのシステムプロパティに指定できるキーは，vbroker.agent.port, vbroker.agent.enableLocator，および vbroker.agent.addr キーだけです。
- TPBroker のサーバ機能は利用できません。また，J2EE サーバ上で，CORBA オブジェクトを活性化することはできません。
- TPBroker のインターセプタは利用できません。
- TPBroker の DII 機能は利用できません。

14.5.2　CORBA オブジェクト呼び出し処理のパッケージ化の注意事項

CORBA オブジェクトの呼び出し処理をパッケージ化するときには，次の点に注意してください。

- org.omg.CORBA 配下のパッケージにあるインタフェース，クラスを利用するクラスは，EJB-JAR ファイルまたは WAR ファイルには含めないでください。これらのクラスは，コンテナ拡張ライブラリ用の JAR ファイルに含めます。

- IDL 定義から生成されたインタフェース，クラス，およびそれらを利用するクラスは，EJB-JAR ファイルまたは WAR ファイルには含めないでください。これらのクラスは，コンテナ拡張ライブラリ用の JAR ファイルに含めます。

14.6　コンテナ拡張ライブラリおよびサーバ起動・停止フック機能利用時の制限事項

コンテナ拡張ライブラリおよびサーバ起動・停止フック機能を利用する場合の制限事項を示します。

- アプリケーションのポータビリティについて

  コンテナ拡張ライブラリは J2EE の仕様外の機能です。したがって，コンテナ拡張ライブラリを使用した場合は，アプリケーションとしての移行性は低下します。

- コンテナ拡張ライブラリの呼び出しについて

  コンテナ拡張ライブラリは，サーブレット，JSP，および Enterprise Bean から呼び出されることを前提としています。次に示す利用形態は適用できないので，注意してください。

  - コンテナ拡張ライブラリおよびサーバ起動・停止フック機能から，EJB-JAR ファイル，WAR ファイルを参照する（EJB-JAR ファイル，WAR ファイルのクラスが，コンテナ拡張ライブラリのクラスを継承することも含む）。
  - コンテナ拡張ライブラリおよびサーバ起動・停止フック機能から，J2EE コンテナの機能を利用する（Enterprise Bean，JNDI，JDBC などの API を呼び出す）。
  - EJB-JAR ファイルおよび WAR ファイルのクラスから，直接サーバ起動・停止フック機能を参照する（コンテナ拡張ライブラリのクラスが，EJB-JAR ファイル，WAR ファイルのクラスを継承することも含む）。

- ファイル，ディレクトリのアクセスについて

  コンテナ拡張ライブラリおよびサーバ起動・停止フック機能から，次に示すファイルやディレクトリを操作しないでください。

  - Application Server のインストールディレクトリ以下のファイルやディレクトリ
  - J2EE サーバの作業ディレクトリ以下のファイルやディレクトリ

- JNI 機能の利用について

  コンテナ拡張ライブラリおよびサーバ起動・停止フック機能から JNI 機能を利用する場合，J2EE サーバでは，ネイティブメソッドでの処理を管理できません。例えば，ネイティブメソッドでメモリアクセス違反があった場合，J2EE サーバは，JavaVM のプロセスごと異常終了します。

- TPBroker の利用について

  - コンテナ拡張ライブラリおよびサーバ起動・停止フック機能では，TPBroker のサーバ機能を利用できません。これは，J2EE サーバ上に TPBroker の CORBA オブジェクトを活性化できないためです。
  - TPBroker の DII 機能は利用できません。

- インストール型オプションパッケージの使用について

  コンテナ拡張ライブラリおよびサーバ起動・停止フック機能は，インストール型オプションパッケージとして使用してはいけません。インストール型オプションパッケージとは，次のディレクトリに置かれたファイルです。

  - <Application Server のインストールディレクトリ>¥jdk¥jre¥lib¥ext に置かれた JAR ファイル
  - <Application Server のインストールディレクトリ>¥jdk¥jre¥bin に置かれたネイティブコードバイナリ

- シャットダウンフックの登録について

  コンテナ拡張ライブラリおよびサーバ起動・停止フック機能では，シャットダウンフックを登録しないでください。

- ハンドラ関数の設定について（Windows の場合）

  Windows でコンテナ拡張ライブラリを使用してプロセスのハンドラ関数を設定する場合，CTRL+BREAK 信号を処理するハンドラ関数では，TRUE を返したり，DLL の終了処理をしたり，ExitProcess 関数などを呼び出してプロセスを終了したりしないでください。

  J2EE サーバ，Component Container を利用するほかのプログラムなどが動作しなくなるおそれがあります。

- C++ライブラリについて（Linux の場合）

  Linux でコンテナ拡張ライブラリを C++で実装している場合，そのライブラリは，Red Hat Enterprise Linux 4 以降でビルドしておく必要があります。Red Hat Enterprise Linux 3 でビルドした C++のライブラリは実行できません。

15 アプリケーション実装時の注意事項

この章では，アプリケーション実装時の注意事項について説明します。

15.1　スレッドローカル変数使用時の注意事項

スレッドローカル変数へ格納したJ2EEアプリケーションのクラスのインスタンスを，J2EEアプリケーションの停止までに削除しないと，J2EEアプリケーションを停止してもクラスローダへの参照が残ることになり，メモリリークが発生します。

java.lang.ThreadLocal.remove()メソッドでスレッドローカル変数へ格納したJ2EEアプリケーションのクラスのインスタンスを削除してください。

フレームワークなどが格納したインスタンスをJ2EEアプリケーションで削除できない場合は，J2EEアプリケーションの開始および停止を繰り返さないでください。J2EEアプリケーションの開始および停止を繰り返す場合は，J2EEサーバを再起動してください。

15.2　Developer's Kit for Java に関する注意事項

ここでは，Developer's Kit for Java に関する共通の注意事項，JDK のバージョンの仕様差異に関する注意事項について説明します。

15.2.1　アプリケーションサーバのバージョン共通の注意事項

アプリケーションサーバのバージョンで共通の Developer's Kit for Java に関する注意事項について説明します。

(1)　OS 共通の場合

OS で共通の注意事項について説明します。

- Java SE の旧バージョンとの互換性について

  Java SE 6 は，Java SE（J2SE）の旧バージョンと，一部の機能に互換性がありません。詳細は，該当ページ（http://www.oracle.com/technetwork/java/javase/compatibility-137541.html）を参照してください。

- Java SE 6 でのそのほかの変更点について

  Java SE 6 でのそのほかの拡張および変更点については，該当ページ（http://www.oracle.com/technetwork/java/javase/releasenotes-136954.html）を参照してください。

- JVMPI と JVMDI について

  JVMPI と JVMDI を廃止しました。これらの代わりに JVM TI を利用してください。

- GC メモリ情報を取得する機能について

  GC メモリ情報を取得する JP.co.Hitachi.soft.jvm.MemoryInfo クラスは，Developer's Kit for Java で利用できます。ほかの製品では利用できません。

- URLClassLoader でのクラスのローディングについて

  URLClassLoader 経由で jar ファイルからクラスをローディングする場合，ローディング処理中に JavaVM の内部で作成されるオブジェクトが，JavaVM 終了時まで削除されないことがあります。適宜，Java アプリケーションを再起動してください。

- Java2D の使用について

  Java2D の DirectDraw および Direct3D の問題を回避するため，DirectDraw と Direct3D を無効にする，sun.java2d.noddraw プロパティに「true」を指定してください。

- メソッド長の上限について

  JavaVM の仕様によって，1 メソッドのバイトコードは，64 キロバイト以内にする必要があります。64 キロバイトを超えると，クラスファイル生成時にエラーとなるか，またはクラスのロード時に java.lang.LinkageError 例外が発生します。

  また，64 キロバイト以内であっても，非常に複雑で行数が多い場合は，次のような弊害が発生することがあります。

  - GC 処理の実行に非常に時間が掛かる。
  - JIT コンパイルに非常に時間が掛かる。
  - JIT コンパイルに非常に多くのメモリを消費する。

  さらに，ローカル変数情報出力機能が有効な場合は，次の弊害も発生することがあります。

  - 拡張スレッドダンプの出力に時間が掛かる。

- スレッドスタックトレースの取得に時間が掛かる。
- 例外発生時の例外オブジェクト生成処理に時間が掛かる。

そのため，Java ソース上の 1 メソッドの行数は，コメントや空行を除いて，およそ 500 行以内にすることをお勧めします。

- 直列化バージョン UID（serialVersionUID）について

  JDK 1.4 以降では，直列化可能な入れ子クラスにクラスオブジェクト（例えば，Class c = Object.class;）への明示的な参照が含まれている場合，入れ子クラスとそのクラスを包含するクラスの両方で，デフォルトの直列化バージョン UID の値が，JDK 1.4 より前のバージョンと異なります。これは，JDK 1.4 で実施された javac コマンドの変更によるものです。これらの条件をすべて満たす場合には，直列化可能クラスに直列化バージョン UID（serialVersionUID）を追加してください。

- ファイルディスクリプタのクローズについて

  java.lang.Runtime.exec()，および java.lang.ProcessBuilder.start()で起動した子プロセスは，Process.getInputStream，getErrorStream，または getOutputStream のそれぞれのメソッドで取り出したストリームを通じてプロセス間通信をします。

  親プロセスでは，これらのメソッドを使わない場合でも三つのファイルディスクリプタを消費することに注意してください。

  ファイルディスクリプタをクローズするのは次の場合です。

  - Process オブジェクトのファイナライズが完了した場合
  - Process.destroy()を呼び出した場合
  - これらストリームに対して明示的に close()メソッドを呼び出した場合

- システムプロパティ java.ext.dirs および java.library.path について

  システムプロパティ java.ext.dirs には，通常のアプリケーションクラスよりも優先してロードするクラスを含む jar ファイルを配置するディレクトリを指定します。デフォルト値を次に示します。

  Windows の場合

  <JDK インストールディレクトリ>¥jdk¥jre¥lib¥ext

  UNIX の場合

  /opt/Cosminexus/jdk/jre/lib/ext

  また，システムプロパティ java.library.path には，ユーザのネイティブライブラリの検索パスを指定します。デフォルト値を次に示します。

  Windows の場合

  "<JDK インストールディレクトリ>¥bin，<システムディレクトリ>，<Windows ディレクトリ>，%PATH%，カレントディレクトリ"

  AIX の場合

  環境変数 LD_LIBRARY_PATH の設定値，JDK 下のネイティブライブラリのディレクトリ，"/usr/lib:/lib"

  HP-UX の場合

  JDK 下のネイティブライブラリのディレクトリ，環境変数 LD_LIBRARY_PATH の設定値，"/usr/lib:/usr/lib/hpux64:/usr/ccs/lib/hpux64"

  Linux の場合

  JDK 下のネイティブライブラリのディレクトリ，環境変数 LD_LIBRARY_PATH の設定値，"/usr/lib64:/lib64:/lib:/usr/lib"

- ポジティブ DNS キャッシュの保持時間について

JDK は，DNS へのホスト名解決の問い合わせ結果をキャッシュする機能を持ちます。キャッシュ機能には，解決成功の結果を保持するポジティブキャッシュと，解決失敗の結果を保持するネガイティブキャッシュがあります。

このうち，セキュリティマネージャが無効な場合だけ，DNS への問い合わせ結果のポジティブキャッシュのデフォルト保持期間が，「永久にキャッシュ」から「実装依存」へ変更になりました。JDK 5.0 と同様の挙動にする場合は，セキュリティプロパティ networkaddress.cache.ttl の値を「-1」にしてください。詳細は，該当ページ（http://docs.oracle.com/javase/6/docs/api/java/net/InetAddress.html）を参照してください。

- インストゥルメンテーション機能が出力するメッセージについて

  インストゥルメンテーション機能の処理中にメモリ不足になった場合，次のメッセージが出力され，インストゥルメンテーション機能の処理が失敗することがあります。詳細は，該当ページ（http://docs.oracle.com/javase/jp/1.5.0/api/java/lang/instrument/package-summary.html）を参照してください。

  メッセージの内容

  ```
  *** java.lang.instrument ASSERTION FAILED ***
  ```

- java.net.Socket.connect()のタイムアウトについて

  java.net.Socket.connect()でのソケットの接続が，OS に設定されている TCP 通信のタイムアウト値によってタイムアウトすることがあります。TCP 通信のタイムアウト値になると，java.net.Socket.connect()に指定したタイムアウト値よりも前でも，タイムアウト値を指定していないときでも，接続がタイムアウトします。

  TCP 通信のタイムアウト値でタイムアウトした場合，次の詳細メッセージを含む java.net.ConnectException 例外がスローされます。

  メッセージの内容

  ```
  Connection timed out: connect [errno=10060, syscall=select]
  ```

  TCP 通信のタイムアウトについては，OS のドキュメントを参照してください。

- クラスロードのタイミングについて

  Java でのクラスファイルのメモリへの読み込み（クラスロード処理）は，プログラムの実行中に，そのクラスが初めて必要となったタイミングで実行されます。そのため，ある処理の初回実行でクラスロードの回数が多くなると，2 回目以降と比較して，処理時間が長くなることがあります。この場合，処理の実行に必要なクラスを事前にロードすることで，処理時間が改善します。

- 異なる OS 間で発生する文字化けについて

  文字コードの Unicode へのマッピングは，OS によって異なることがあります。このため，次の場合に文字化けが発生することがあります。文字化けが発生する代表的な文字には，—，～，‖，−，¢，£，¬があります。

  - UTF-8 などの Unicode のエンコーディングを使用して，異なる OS 間で上記文字データを受け渡す場合
  - 上記文字を含む文字列リテラルがあるソースプログラムから作成したクラスファイルを，異なる OS 上でリコンパイルしないでそのまま実行する場合

- Java API で使用するファイルについて

  Java API で使用するポリシーファイルや，ログイン構成ファイルなどのコンフィグレーションファイルは，UTF-8 エンコーディング方式でエンコーディングする必要があります。

- java.net.URL が扱う URL 文字列のエンコーディングについて

java.net.URL が扱う URL 文字列のエンコーディングは，標準 UTF-8 ではなく，Modified UTF-8 です。Modified UTF-8 でない文字列を与えた場合は，java.lang.IllegalArgumentException が発生します。

- JNI 関数で文字列操作をする場合の UTF-8 エンコーディングについて

  JNI の場合，次の文字列操作をする関数で使用するエンコーディングは，標準 UTF-8 ではなく，Modified UTF-8 です。

  - NewStringUTF
  - GetStringUTFLength
  - GetStringUTFChars
  - ReleaseStringUTFChars
  - GetStringUTFRegion
- if 文の判定でジャンプできる長さの上限について

  JavaVM の仕様によって，if 文の判定でジャンプできる長さは 32 キロバイトまでです。ジャンプ先が 32 キロバイトを超える場合は，java.lang.LinkageError となります。
- ファイルサイズの上限について

  java.util.zip，java.util.jar パッケージでは，4 ギガバイトを超える JAR ファイルや ZIP ファイルはサポートしていません。
- java.nio.channels.FileChannel クラスの map，transferFrom，および transferTo メソッドについて

  map メソッドでは，2 ギガバイトを超えるファイルはサポートしていません。

  また，transferFrom および transferTo メソッドでの count 指定では，2 ギガバイト以上の値はサポートしていません。
- スレッドのスタックサイズについて

  JNI 関数 AttachCurrentThread()などを使用して，ネイティブスレッドを JavaVM に接続する場合は，-Xss オプションの指定値よりも大きなスタックサイズのスレッドで接続してください。

(2)　Windows の場合

Windows での注意事項について説明します。

- AWT コンポーネントでの JIS X0213:2004 の第三水準，第四水準文字の非サポートについて

  AWT コンポーネントは JIS X0213:2004 の第三水準，第四水準文字をサポートしていません。AWT コンポーネントが出力する情報で該当する文字の部分は文字化けします。Windows Server 2012，Windows Server 2008，Windows 8，Windows 7，および Windows Vista で該当する文字を扱う場合には，Swing コンポーネントを使用してください。
- JavaVM ログファイルの GC 経過時間について

  JavaVM ログファイルの GC 経過時間は，秒単位で小数点以下第 7 位（100 ナノ秒）まで出力されます。

  Windows の場合，08-70 より前では，小数点以下第 7 位（100 ナノ秒）まで有効な数値が出力されていました。08-70 以降では，小数点以下第 6 位（1 マイクロ秒）まで有効な数値が出力され，小数点以下第 7 位（100 ナノ秒）には常に 0 が出力されます。
- JNI プログラム中で使用できないライブラリについて

  JavaVM 内で管理している情報が不正に更新されることがあるため，JNI プログラム中で次のライブラリは使用できません。

  - setjmp()

- longjmp()

- java.io.tmpdir プロパティについて

  java.io.tmpdir プロパティには，書き込み権限があり，かつ存在するディレクトリを指定してください。java.io.tmpdir プロパティの初期値は，Windows API の GetTempPath()関数で得られるディレクトリです。

  また，Java RMI の動的クラスローディング機能や Java API の java.io.File.createTempFile()では，java.io.tmpdir プロパティで指定されたディレクトリに一時ファイルを作成します。これらの機能を正常に動作させるため，JavaVM プロセス起動中は一時ファイルの作成先を削除しないでください。

- 管理者特権が必要な独立プロセスの起動について（Windows Server 2012，Windows Server 2008，Windows 8，Windows 7，および Windows Vista の場合）

  管理者特権のないユーザで起動した java アプリケーションから，java.lang.Runtime.exec()，または java.lang.ProcessBuilder.start()で，管理者特権のある独立プロセスを起動する場合は，次の手順で起動してください。

  1. 管理者特権として起動するコマンドにマニフェストを追加します。マニフェストの追加方法については，OS のドキュメントを参照してください。
  2. Windows の cmd.exe を介してプロセスを起動します。

     例えば，sample.exe を起動する場合には次のようになります。

     ・Runtime.getRuntime().exec("cmd.exe", "/c", "sample.exe");

     ・new ProcessBuilder("cmd.exe", "/c", "sample.exe").start();

     この java アプリケーションを起動すると，［ユーザー アカウント制御］ダイアログが表示されるので，［続行］をクリックしてアプリケーションを続行してください。

- java.util.prefs.Preferences.systemNodeForPackage によるデータ共有について（Windows Server 2012，Windows Server 2008，Windows 8，Windows 7，および Windows Vista の場合）

  管理者特権のないユーザが，java.util.prefs.Preferences.systemNodeForPackage を使用した場合，異なるユーザ間でのデータ共有機能は使用できません。

  例えば，同一のキーを引数に指定した場合，異なるユーザ間では異なったデータが返ります。

- デフォルトエンコーディングについて

  アプリケーションサーバのデフォルトエンコーディングは windows-31j（別名：MS932）です。

- エンコーディング時に発生する文字化けについて

  次に示す文字を java プログラムで扱う場合，エンコーディングに MS932，windows-31j，cswindows31j のどれかを指定してください。それ以外のエンコーディングを指定した場合，文字化けすることがあります。

  ・NEC拡張文字　①②...⑳，ⅠⅡ...Ⅹ，㈱，髙﨑㐂昭和平成，㍉㌔㌢㍍など
  ・NEC選定IBM拡張文字　ⅰⅱ...ⅹなど
  ・IBM拡張文字　ⅠⅡ...Ⅹ，ⅰⅱ...ⅹ，№，℡など

  （例）

```
import java.io.*;
class encode_eucjis {
  public static void main( String arg[] ) {
    try {
      String string_data = "ⅠⅡⅢ";
      byte[] data = string_data.getBytes();
      InputStreamReader isr =
        new InputStreamReader(
                    new ByteArrayInputStream( data ), "eucjis");
      char[] read_data = new char[4];
      isr.read( read_data, 0, 4 );
      System.out.println(new String(read_data));
    }
    catch ( Exception e ) {
      e.printStackTrace();
    }
  }
}
```

このプログラムは，IBM 拡張文字を eucjis でエンコーディングしているため，実行すると次のように文字化けします。

```
java encode_eucjis
???
```

文字化けを回避するためには，MS932，windows-31j，cswindows31j のどれかを指定してエンコーディングしてください。

- Java がサポートする UTF-8 エンコーディングについて

  Java がサポートする UTF-8 エンコーディングは，BOM なしの UTF-8 です。Windows のテキストエディタ（メモ帳）で「文字コード」に「UTF-8」を指定して保存したデータは，BOM ありの UTF-8 となるため，Java では扱えません。

(3)　UNIX 共通の場合

UNIX で共通の注意事項について説明します。

- fork システムコールについて

  JNI や JVMTI で呼び出されるネイティブメソッド，またはネイティブコードで，fork()システムコール発行だけで現行プロセスのコピーの子プロセスを生成，または実行した場合，その親子プロセスの動作は保証できません。fork()システムコール発行による子プロセス生成後は，必ず exec()システムコールで新規プログラムをローディングしてから起動してください。また，Java 環境下で子プロセスを生成する場合には，Java クラスライブラリの java.lang.Runtime.exec()メソッドを使用することをお勧めします。

- シグナルについて

  JNI や JVMTI で呼び出されるネイティブメソッド，またはネイティブコードで，次のシグナルに対してシグナルハンドラを登録した場合，動作は保証しません。

  SIGHUP，SIGINT，SIGQUIT，SIGILL，SIGFPE，SIGBUS，SIGSEGV，SIGPIPE，SIGTERM，SIGUSR2，SIGCHLD，SIGXCPU，SIGXFSZ，（_SIGRTMAX-2）番のシグナル

- システムライブラリ関数やシステムコール呼び出し中のシグナル受信について

  システムライブラリ関数やシステムコール呼び出し中に，次のシグナルを受信することがあります。

  SIGHUP，SIGINT，SIGQUIT，SIGILL，SIGFPE，SIGBUS，SIGSEGV，SIGPIPE，SIGTERM，SIGUSR2，SIGCHLD，SIGXCPU，SIGXFSZ，（_SIGRTMAX-2）番のシグナル

  システムライブラリ関数やシステムコールを呼び出す場合，該当する関数の処理が，これらのシグナル受信によって中断されて，エラーリターン（errno 値に EINTR が設定されるなど）することがあります。この場合は，適切な処置（例えば，再実行など）を実施してください。

- JNI プログラム中で使用できないライブラリについて

  JavaVM 内で管理している情報が更新されて不正な情報となる場合があるため，JNI プログラム中で次のライブラリは使用できません。

  - setjmp()
  - longjmp()
  - _setjmp()
  - _longjmp()
  - sigsetjmp()
  - siglongjmp()

- Windows との文字データの受け渡しについて

  Windows との文字データの受け渡しをする場合には，エンコーディングに MS932，windows-31j，cswindows31j のどれかを指定してください。

  Shift_JIS や SJIS を指定すると，次の文字が不正な文字コードに変換されます。

  ・NEC拡張文字　①②...⑳，ⅠⅡ...Ⅹ，㈱，髙﨑昭和平成，㍉㌔㌢㍍㌘など
  ・NEC選定IBM拡張文字　ⅰⅱ...ⅹなど
  ・IBM拡張文字　ⅠⅡ...Ⅹ，ⅰⅱ...ⅹ，№，℡など

- 生成できるスレッド数の上限値について

  システム全体で生成できるスレッド数の上限は，/proc/sys/kernel/threads-max です。また，1 ユーザ当たりの生成できるスレッド数の上限は，/etc/security/limits.conf の nproc の値（ulimit -u と同じ）となります。システムに応じて，これらの値を調整してください。

- 通知待ちのモニタ数について

  JVMTI インタフェースを使用して取得した，通知待ちのモニタ情報が不正となる場合があります。

(4)　AIX の場合

AIX での注意事項について説明します。

- Runtime.exec の例外メッセージについて

  Runtime.exec で例外が発生した場合，エラーメッセージが文字化けすることがあります。

- java.security.SecureRandom クラスについて

  SecureRandom クラスが使用する OS の乱数生成器のデフォルトは/dev/urandom ファイルです。java.security.egd プロパティに file:/dev/random を指定すると，/dev/random ファイルを使用できます。ただし，/dev/random ファイルを使用した場合，OS の乱数生成速度には限度があります。このため，SecureRandom クラスが/dev/random ファイルから乱数を取得する，次の処理を短い時間間隔で実施すると，OS による乱数生成完了まで処理が完了しないので注意してください。

  - SecureRandom クラスの generateSeed()メソッド呼び出し時
  - SecureRandom クラスの getSeed()メソッド呼び出し時

  java.security.egd プロパティについては，該当ページ（http://docs.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html）を参照してください。

- java.awt.print による印刷サポートについて

  Developer's Kit for Java の java.awt.print パッケージ印刷 API がサポートする AIX 印刷サブシステムは，System V 印刷サブシステムです。印刷 API を使用する場合は，次の方法で System V 印刷サブシステムを設定してください。

  1. smit printer コマンドを実行します。

2. [印刷スプーリング] − [現在の印刷サブシステムの変更／表示] − [現在の印刷サブシステム] を選択し，Tab キーで SystemV に変更します。

- 環境変数 LANG について

  環境変数 LANG に次の値以外の値を指定している場合，java.nio.channels パッケージで出力される例外メッセージが文字化けすることがあります。

  - C
  - POSIX

- java.io.tmpdir プロパティについて

  java.io.tmpdir プロパティには，書き込み権限があり，かつ存在するディレクトリを指定してください。java.io.tmpdir プロパティの初期値は/tmp です。

  また，Java RMI の動的クラスローディング機能や Java API の java.io.File.createTempFile()では，java.io.tmpdir プロパティで指定されたディレクトリに一時ファイルを作成します。これらの機能を正常に動作させるため，JavaVM プロセス起動中は一時ファイルの作成先を削除しないでください。

- デフォルトエンコーディングについて

  アプリケーションサーバのデフォルトエンコーディングを次に示します。

  LANG が C または POSIX のとき：ISO8859_1

  LANG が ja_JP，ja_JP.IBM-eucJP のどちらかのとき：Cp33722C

  LANG が Ja_JP，Ja_JP.IBM-943 のどちらかのとき：Cp943C

  LANG が Ja_JP.IBM-932 のとき：Cp942C

  LANG が JA_JP，JA_JP.UTF-8 のどちらかのとき：UTF8

(5)　HP-UX の場合

HP-UX での注意事項について説明します。

- AWT コンポーネントでの JIS X0213:2004 の第三水準，第四水準文字の非サポートについて

  AWT コンポーネントは JIS X0213:2004 の第三水準，第四水準文字をサポートしていません。AWT コンポーネントが出力する情報で該当する文字の部分は文字化けします。該当する文字を扱う場合には Swing コンポーネントを使用してください。

- Runtime.exec の例外メッセージについて

  Runtime.exec で例外が発生した場合，エラーメッセージが文字化けすることがあります。

- java.security.SecureRandom クラスについて

  /dev/random ファイルがあるプラットフォームでは，SecureRandom クラスの一部の API で/dev/random ファイルから OS が生成した乱数を取得します。/dev/random ファイルから乱数を取得するのは，次の条件がすべて重なる場合です。

  - java.security.egd プロパティまたは java.security ファイル中の securerandom.source に file:/dev/random/を指定する
  - SecureRandom クラスの generateSeed()メソッドまたは getSeed()メソッドを呼び出す

  OS の乱数生成速度には限度があります。このため，短い間隔で乱数の取得処理を実行すると，OS による乱数生成完了まで処理が完了しないので注意してください。

- java プログラム中でロードするライブラリパスについて

  java.lang.System.load メソッド，および java.lang.Runtime.load メソッドでロードするライブラリについては，指定されているパス名は無視され，環境変数 SHLIB_PATH，および環境変数 LD_LIBRARY_PATH で指定されたパスからロードされます。

- java.nio.MappedByteBuffer.isLoaded メソッドについて

  ダイレクトバッファが物理メモリ内にあるかどうかを問い合わせる isLoaded()メソッドを使用した場合，特定のメモリ領域が物理メモリ上にあるかどうかを問い合わせることができません。isLoaded()メソッドの返却値は常に false となります。

- java.lang.Thread.setPriority メソッドについて

  各スレッドの優先順位は OS による制御に依存します。このため，スレッドの優先順位を setPriority()メソッドで設定しても有効になりません。

- 環境変数 LANG について

  環境変数 LANG に次の値以外の値を指定している場合，java.nio.channels パッケージで出力される例外メッセージが文字化けすることがあります。

  - C
  - POSIX

- java.io.tmpdir プロパティについて

  java.io.tmpdir プロパティには，書き込み権限があり，かつ存在するディレクトリを指定してください。java.io.tmpdir プロパティの初期値は/var/tmp です。

  また，Java RMI の動的クラスローディング機能や Java API の java.io.File.createTempFile()では，java.io.tmpdir プロパティで指定されたディレクトリに一時ファイルを作成します。これらの機能を正常に動作させるため，JavaVM プロセス起動中は一時ファイルの作成先を削除しないでください。

- デフォルトエンコーディングについて

  アプリケーションサーバのデフォルトエンコーディングを次に示します。

  LANG が C または POSIX のとき：ISO8859_1

  LANG が ja_JP.SJIS のとき：SJIS

  LANG が ja_JP.eucJP のとき：EUC_JP

  LANG が ja_JP.kana8 のとき：ISO8859_1

  LANG が ja_JP.utf8 のとき：UTF8

(6)　Linux の場合

Linux での注意事項について説明します。

- 半角カタカナについて

  JIS X 201 で制定されている文字のうち，半角カタカナは JavaSE の API を使用した GUI の実装では利用できません。

- java.security.SecureRandom クラスについて

  /dev/random ファイルがあるプラットフォームでは，SecureRandom クラスの一部の API で/dev/random ファイルから OS が生成した乱数を取得します。/dev/random ファイルから乱数を取得するのは，次の条件がすべて重なる場合です。

  - java.security.egd プロパティまたは java.security ファイル中の securerandom.source に file:/dev/random/を指定する
  - SecureRandom クラスの generateSeed()メソッドまたは getSeed()メソッドを呼び出す

  OS の乱数生成速度には限度があります。このため，短い間隔で乱数の取得処理を実行すると，OS による乱数生成完了まで処理が完了しないので注意してください。

- java.io.tmpdir プロパティについて

java.io.tmpdir プロパティには，書き込み権限があり，かつ存在するディレクトリを指定してください。java.io.tmpdir プロパティの初期値は/tmp です。

また，Java RMI の動的クラスローディング機能や Java API の java.io.File.createTempFile()では，java.io.tmpdir プロパティで指定されたディレクトリに一時ファイルを作成します。これらの機能を正常に動作させるため，JavaVM プロセス起動中は一時ファイルの作成先を削除しないでください。

- Red Hat Enterprise Linux 5.1 で使用する場合について

  Red Hat Enterprise Linux 5.1 の IPv6 を有効にした環境で使用する場合は，java.net.preferIPv4Stack プロパティに true を設定してください。

- デフォルトエンコーディングについて

  アプリケーションサーバのデフォルトエンコーディングを次に示します。

  LANG が C または POSIX のとき：US-ASCII（別名 ASCII）

  LANG が jp_JP，jp_JP.eucJP，ja_JP.ujis，japanese，japanese.euc のどれかのとき：x-euc-jp-linux（別名 EUC_JP_LINUX）

  LANG が ja_JP.utf8 のとき：UTF-8（別名 UTF8）

15.2.2　アプリケーションサーバ Version 6 で提供していた JDK 1.4.2 との仕様差異に関する注意事項

アプリケーションサーバ Version 6 で提供していた JDK 1.4.2 との仕様差異に関する注意事項について説明します。

(1)　OS 共通の場合

OS で共通の注意事項について説明します。

- java.math.BigDecimal クラスの値表示の差異について

  JDK 5.0 および JDK 6 では，toString()メソッドによる文字列変換で指数表記が使用されるようになったため，値の出力形式が異なります。

  （例）1E-15 の値を toString()メソッドで出力する

  - JDK 1.4.2 以前の場合：0.000000000000001
  - JDK 5.0 および JDK 6 の場合：1E-15

  なお，JDK 1.4.2 以前の形式で値を出力したい場合は，toPlainString()メソッドを使用してください。

- Unicode のバージョンについて

  JDK 6 では，文字処理が Unicode 標準のバージョン 4.0 に基づくようになったため，文字コードを個々に参照して解析する場合は注意が必要です。

  詳細は，該当ページ（http://www.oracle.com/technetwork/articles/javase/supplementary-142654.html）の「Supporting Supplementary Characters in Your Application」を参照してください。

- java ソースプログラムの注釈の扱いについて

  JDK 6 では，ソースプログラムの注釈中の文字について，文字コードをチェックします。例えば，デフォルトエンコーディングが EUC の環境で，注釈に Shift-JIS コードを含むソースプログラムをコンパイルした場合には警告が出力されます。コンパイル時には適切なエンコーディングを-encoding オプションに指定してください。

- Java プログラムのコンパイルについて

J2SE 5.0 での Java 言語仕様の拡張に伴い，javac コマンドのデフォルトで以前はコンパイルできていた Java プログラムでも，J2SE 5.0 の Java 言語仕様に違反する構文について，注意または警告のメッセージが出力されたり，エラーになったりすることがあります。注意および警告のメッセージの場合，以前のバージョンで動作していたものはそのまま動作させて問題ありません。ただし，Java コードとしては，修正が望ましいコードと考えられます。なお，javac コマンドで-source オプションを指定すると，javac コマンドが受け付けるソースコード バージョン 1.4 を指定してコンパイルできます。これによって，注意および警告のメッセージを抑止できます。

- Java プログラムのコンパイル時に発生する注意および警告のメッセージについて

  J2SE 5.0 での Java 言語仕様の拡張に伴い，新しい言語仕様によるソース記述方法を推奨するため，次のような注意メッセージがコンパイル終了時に出力されることがあります。

  - 注: AccountEJB.java の操作は，未チェックまたは安全ではありません。
  - 注: 詳細については，-Xlint:unchecked オプションを指定して再コンパイルしてください。

  また，-Xlint オプション指定時には，次のような警告メッセージが該当個所ごとに出力されます。

  - 警告: [unchecked] raw 型 java.util.List のメンバとしての add(E) への無検査呼び出しです。

  これらの警告および注意のメッセージは，コンパイル時にチェックができないことを示すものです。実際にエラーがあった場合には，JDK 1.4 でも実行時に例外が発生します。したがって，移行前に稼働実績があるプログラムであれば，特に対処する必要はありません。

- JDK 1.4 から JDK 5.0 への変更での言語仕様に関する主な非互換項目について

  J2SE 5.0 での Java 言語仕様の拡張で，Java 言語に型保証された列挙型が追加されました。このため，enum はキーワードとなり，識別子として使用できません。

- java.net.Proxy クラスの追加による影響について

  JDK 5.0 では，既存の java.lang.reflect.Proxy クラスに加えて，java.net.Proxy クラスが追加されました。そのため，記述例 1 のように import 宣言をしたクラスで Proxy と記述した場合，どちらのパッケージの Proxy クラスかがあいまいであるため，コンパイルエラーとなります。

  記述例 1

  ```
  import java.lang.reflect.*;
  import java.net.*;
  ```

  この場合，記述例 2 のように import 宣言を変更すると，java.lang.reflect パッケージの Proxy クラスと明記されているので，コンパイルエラーを回避できます。

  記述例 2

  ```
  import java.lang.reflect.*;
  import java.net.*;
  import java.lang.reflect.Proxy;
  ```

- クラス初期化のタイミング変更について

  JDK 5.0 より前では，クラスリテラル（Foo.class など）を参照すると，クラスが初期化されていました。しかし，JDK 5.0 以降では初期化されません。クラスリテラルの参照時にクラスを初期化しようとしている場合には，初期化コードを追加する必要があります。記述例を次に示します。

  初期化コードの追加前の記述例

  ```
  Class cl = Foo.class;
  ```

  初期化コードの追加後の記述例

  ```
  Class cl = forceInit(Foo.class);
  public static <T> Class<T> forceInit(Class<T> klass) {
  try {
  ```

```
Class.forName(klass.getName(), true, klass.getClassLoader());
} catch (ClassNotFoundException e) {
throw new AssertionError(e); // Can't happen
}
return klass;
}
```

- java.lang.ClassLoader クラスの変更点について

  String クラス名を引数とする ClassLoader メソッドのクラス名には，API 仕様上，ピリオド（.）で区切られたバイナリ名を指定する必要があります。

  JDK 5.0 より前では，スラッシュ（/）で区切られた名称（例えば，java/lang/String など）でも問題なく動作しました。しかし，JDK 5.0 以降では，バイナリ名でないクラス名を指定した場合には，java.lang.ClassNotFoundException が発生します。ClassLoader クラスのメソッドには，API 仕様に従ってクラスのバイナリ名を指定してください。

- java.util.logging.Level の変更点について

  JDK 5.0 より前では，コンストラクタ java.util.logging.Level(String name, int value, String resourceBundleName)に null の name 引数を与えても例外は発生しません。しかし, JDK 5.0 以降では，NullPointerException が発生します。null 引数を与えないようにしてください。

- java.sql.Timestamp.compareTo の変更点について

  JDK 5.0 では，java.sql.Timestamp.compareTo(java.util.Date)が API に追加されました。これは，JDK1.4.2 の java.util.Date.compareTo(java.util.Date)とは精度が異なり，ナノ秒までの精度を持っています。JDK 1.4.2 ではこの API がないため，継承の関係から java.util.Date.compareTo(java.util.Date)が呼ばれていましたが，JDK 5.0 以降では，java.sql.Timestamp.compareTo(java.util.Date)が呼ばれます。

  次の条件をすべて満たす場合，JDK 1.4.2 と JDK 5.0 以降では比較結果が異なることがあります。

  - java.sql.Timestamp.compareTo(java.util.Date o)を呼び出す
  - java.sql.Timestamp オブジェクトの持つ時刻と引数 java.util.Date オブジェクトの持つ時刻で，ナノ秒単位の差がある

  JDK 1.4.2 と同じ精度で java.sql.Timestamp と java.util.Date の比較をするためには, java.util.Date の compareTo(java.util.Date)メソッドを使用して，java.sql.Timestamp を引数に渡す必要があります。

- 直列化バージョン UID について

  入れ子クラス内で包含クラス※のメソッド呼び出し（例えば, <入れ子クラスを包含するクラス名>.this.メソッド名(引数);）をして，その包含クラスが直列化可能な場合，JDK 5.0 以降では，包含クラスのデフォルトの直列化バージョン UID の値が，JDK 5.0 より前のバージョンと異なります。これは，JDK 5.0 で実施された javac コマンドの変更によるものです。これらの条件をすべて満たす場合には，直列化可能クラスに直列化バージョン UID を追加してください。

  注※　包含クラスとは，入れ子クラスを包含するクラスのことです。

  （記述例）

```
import java.io.*;
class A implements Serializable { // 直列化可能な包含クラス
void method1() {}
class B { // 入れ子クラス
void method2() { A.this.method1(); }
}
```

}

- メソッドキャンセル機能への影響について（Windows，AIX および Linux の場合）

  Windows，AIX および Linux の場合，アプリケーションサーバ 07-00 以降では，アプリケーションサーバ 06-70 と比較して，メソッドキャンセル機能が失敗しやすくなっています。

  これは，Component Container 06-70-/G での保護区の追加と，アプリケーションサーバ 07-00 での JDK 1.4.2 から JDK 5.0 へのバージョンアップによるものです。

  - Component Container 06-70-/G での保護区の追加

    Component Container 06-70-/G では，Java クラスライブラリ内の複数クラスを保護区に追加しました。保護区に追加した Java クラスライブラリ内のクラスでは，メソッドキャンセルが実行されると内部状態に不整合が生じます。

    Component Container 07-00 以降は，Component Container 06-70-/G 以降の保護区を引き継いでいるため，Component Container 06-70-/G より前と比較して，保護区が多くなっています。保護区として定義されている内容については，マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「付録 C　保護区リストの内容」を参照してください。

  - アプリケーションサーバ 07-00 での JDK 5.0 へのバージョンアップ

    Java の仕様や JDK 実装上の制限によって，メソッドキャンセルできないことがあります。このような領域は，クラス単位ではない特殊な保護区としています。Java の仕様や JDK 実装上の制限によって，メソッドキャンセルできないことがある領域の扱いについて，Component Container 06-70 と 07-00 以降の仕様差を次の表に示します。

    表 15–1　メソッドキャンセルできないことがある領域の扱い

Java プログラムの実行処理	Component Container 06-70 での扱い	Component Container 07-00 以降での扱い
synchronized でロック確保待ち	メソッドキャンセルできる	メソッドキャンセルできない
for や while ループの終端	メソッドキャンセルできる	メソッドキャンセルできない
オブジェクトの生成（new キーワード）	メソッドキャンセルできる	
上記以外の Java プログラム実行中	メソッドキャンセルできない（ただし，インタプリタによる実行中はメソッドキャンセルできる）	

    なお，Java の仕様や JDK 実装上の制限によってメソッドキャンセルできなかった場合は，KDJE52718-W セージの詳細情報として，クラス名（JP.co.Hitachi.soft.jvm.CriticalClass.dummy(Native Method)）が表示されます。

- UDP ソケット数の上限について

  セキュリティマネージャが有効な場合，JavaVM プロセス当たり 26 以上の UDP ソケットを同時に使用することはできません。

  26 以上の UDP ソケットを作成しようとした場合は，java.net.SocketException がスローされます。

(2)　HP-UX の場合

HP-UX での注意事項について説明します。

- スタックサイズ（-Xss オプション）の初期値について

  JDK 1.4.2 では，Java スレッドのスタックサイズの初期値は 1 メガバイトでしたが，JDK 6 以降では，Java スレッドのスタックサイズの初期値は 4 メガバイトとなっています。

(3)　Linux の場合

Linux での注意事項について説明します。

- java.lang.Process クラスの waitFor()および exitValue()メソッドについて
  waitFor()および exitValue()メソッドで取得できる子プロセスの終了コードが 0 以上 255 以下に変更されました。子プロセスが負の終了コードで終了した場合，その値は符号なし整数と見なされます。

15.2.3　アプリケーションサーバ Version 7 および Version 8 で提供していた JDK 5.0 との仕様差異に関する注意事項

アプリケーションサーバ Version 7 および Version 8 で提供していた JDK 5.0 との仕様差異に関する注意事項について説明します。

(1)　OS 共通の場合

OS で共通の注意事項について説明します。

- クラスファイルの変更点について
  クラスファイル形式の拡張に伴い，javac コマンドのデフォルトのコンパイルで生成されるクラスファイルのバージョンが 49.0 から 50.0 となります。また, このクラスファイルを JDK 5.0 以前の実行環境で実行すると，java.lang.UnsupportedClassVersionError がスローされます。
- java.io.File の変更点について
  java.io.File.deleteOnExit()の実装を変更しました。Java VM 終了時に削除するファイル情報を保持するヒープ領域が，C ヒープ領域から Java ヒープ領域に変更になりました。
- java アプリケーション起動ツールの非標準オプションについて
  次のオプションは，JDK 6 からサポートしていません。
  - -Xrunhprof
  - -Xdebug

  これらのオプションを指定した場合の動作は保証されません。指定しないようにしてください。なお，オプションを指定した場合はエラーにならないで実行されます。
- javac コマンドの変更点について
  javac コマンドを実行したときのリターンコードのうち，コンパイル対象ソースが見つからない場合のリターンコードが変更（JDK 5.0 では 1，JDK 6 では 2）されました。
- UDP ソケット数の上限について
  セキュリティマネージャが有効な場合，JavaVM プロセス当たり 26 以上の UDP ソケットを同時に使用することはできません。
  26 以上の UDP ソケットを作成しようとした場合は，java.net.SocketException がスローされます。

(2)　UNIX 共通の場合

UNIX で共通の注意事項について説明します。

- AWT の変更点について
  AWT の実装を Motif ベースの MAWT から X Window System ベースの XAWT に変更しました。

(3)　AIX の場合

AIX での注意事項について説明します。

- スタックサイズ（-Xss オプション）の初期値について
  JDK 5.0 では，Java スレッドのスタックサイズの初期値は 512 キロバイトでしたが，JDK 6 以降では，Java スレッドのスタックサイズの初期値は 1 メガバイトとなっています。

(4)　HP-UX の場合

HP-UX での注意事項について説明します。

- スタックサイズ（-Xss オプション）の初期値について
  JDK 5.0 では，Java スレッドのスタックサイズの初期値は 1 メガバイトでしたが，JDK 6 以降では，Java スレッドのスタックサイズの初期値は 4 メガバイトとなっています。

15.3　sun.rmi から始まるロガー使用時の注意事項

J2EE サーバでは，RMI レジストリを使用しています。

また，RMI 実装ログを取得するため次の名称のロガーを使用しています。

- sun.rmi.transport.tcp
- sun.rmi.server.call
- sun.rmi.client.call

J2EE アプリケーションで，これらのロガーを使用する場合は，次の点に注意してください。

- J2EE サーバの RMI 実装ログが出力されます。
- J2EE サーバでは，java.util.logging.Logger クラスの setUseParentHandlers(boolean)メソッドを使用して，これらのロガーの出力を親ロガーに送信しないように設定しています。そのため，親ロガーによる RMI 実装ログの出力はできません。
- J2EE サーバでは，ログレベル FINER で RMI 実装ログを出力しています。これらのロガーのログレベルを変更する場合，次のログレベルは設定しないでください。
  - SEVERE
  - WARNING
  - INFO
  - CONFIG
  - FINE
  - OFF

付録

付録 A　文字コード

HTTP は，主に Web サーバからブラウザへデータをダウンロードするために用いられるプロトコルですが，HTML のフォームの場合などではデータのアップロードにも使われます。

ダウンロードするコンテンツの文字コードを Web サーバからブラウザに伝えるための規格は，Servlet API でも明確に規定されています。しかし，過去の Servlet API（Servlet API 2.2 以前）では，HTTP クエリ文字列と HTTP リクエストボディの文字コードの扱いが明確ではなかったため，ベンダによって扱いが異なっていました。

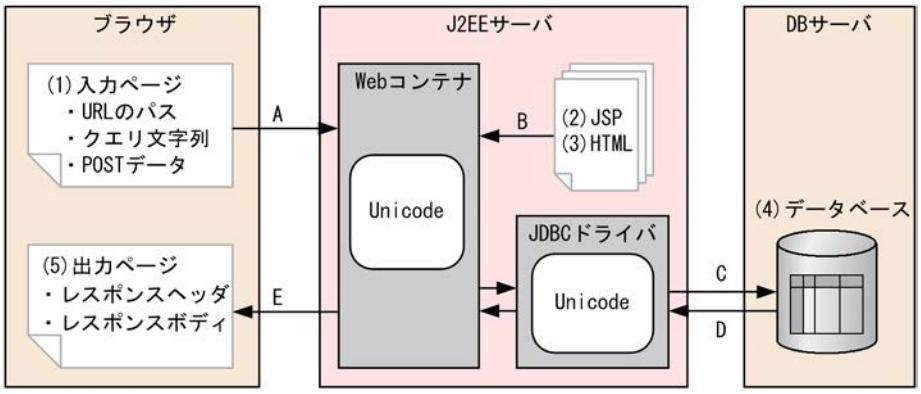
Servlet API 2.3 からは，Servlet API を通して HTTP クエリ文字列や HTTP リクエストボディを参照するときに，文字コードを指定できるようになりました（ただし，エンコードタイプが multipart/form-data の HTTP リクエストボディを除きます）。これらのデータは，指定された文字コードのデータと見なされて Java の内部表現である Unicode に変換されてからアプリケーションに渡されます。リソースの文字コードや Unicode への文字コード変換が間違っていると，「文字化け」を引き起こす原因になるため，アプリケーション開発時には，実行プロセスやリソースの文字コードを考慮する必要があります。

ここでは，アプリケーションで使用する文字コードと注意事項を示します。また，ブラウザとデータをやり取りする場合の，文字コード変換の注意事項についても示します。

付録 A.1　アプリケーションで扱う文字コード

アプリケーションの構成を基に，文字コード変換の流れを示します。

図 A-1　アプリケーションの文字コード変換の流れ



図中の番号（(1) ～ (5)）は，文字コードを考慮する必要のあるプログラムやリソースを表します。また，図中の英字 (A～E) は文字コード変換時のデータの流れを示します。プログラム，リソースで扱う文字コードと注意事項，および文字コード変換の動作と注意事項を，それぞれ表に示します。

表 A-1　プログラム，リソースで扱う文字コードと注意事項

プログラム，リソース名	項目	扱う文字コードおよび注意事項
(1) 入力ページ	URL のパス	URL のパスの文字コードは，ISO-8859-1 である必要があります。シフト JIS などの非 ASCII 文字は記述できません。
	クエリ文字列	HTML の<FORM>タグを使用して送信される文字は，フォームを表示している HTML ページと同じ文字コードで URL エンコードされて送出されます。
	POST データ	

プログラム，リソース名	項目	扱う文字コードおよび注意事項
(2) JSP	JSP	任意の文字コードで作成できます。 Web アプリケーションのバージョン 2.4 以降（JSP 2.0 仕様以降）で，JSP ドキュメントの文字エンコードを指定する場合は，XML 宣言で文字エンコードを指定してください。 JSP 1.2 仕様では，JSP の文字コードは，page ディレクティブの pageEncoding 属性に記述する必要があります。
(3) HTML	HTML	任意の文字コードで作成できます。
(4) データベース	データベース	データベースに格納するデータの文字コードは，ブラウザに表示する文字コードを考慮して決定してください。
(5) 出力ページ	レスポンスヘッダ	ISO-8859-1 である必要があります。非 ASCII 文字を使用する場合は，URL エンコードする必要があります。
	レスポンスボディ	ユーザプログラムで任意の文字コードを使用できます。

注意

Web コンテナでは英数字を 2 バイトで表現する文字コードは使用できません。英数字に 2 バイト使用する文字コードを次に示します。

- UCS-2（ISO/IEC 10646）
- UCS-4（ISO/IEC 10646-1）
- UTF-16

表 A-2　文字コード変換の動作と注意事項

変換個所	対象	文字コード変換時の動作および注意事項
A　ブラウザ～ J2EE サーバ	URL のパス	Web コンテナでは，URL のパスの文字コードは ISO-8859-1 として処理されます。
	クエリ文字列	クエリ文字列または POST データの文字コードは，アプリケーションで任意に決められます。サーブレット，JSP では Unicode で扱われるので，アプリケーション内で矛盾のないように文字コードを変換してください。
	POST データ	
B　J2EE サーバ内	JSP ファイル	page ディレクティブの pageEncoding 属性に記述されたエンコーディングでファイルが読み込まれます。pageEncoding 属性が省略されている場合は，contentType 属性が使用されます。
	HTML ファイル	HTML ファイルの文字コードのままブラウザに送信されます。
C　J2EE サーバ～ データベース	データベース	JDBC ドライバによって，Unicode がデータベース格納文字コードに変換されます。
D　データベース～ J2EE サーバ	データベース	JDBC ドライバによって，データベース格納文字コードが Unicode に変換されます。
E　J2EE サーバ～ ブラウザ	レスポンスヘッダ	Web コンテナによって，レスポンスヘッダの文字コードが ISO-8859-1 に変換されます。
	レスポンスボディ	サーブレットの場合 ServletResponse クラスの setContentType メソッドによって文字コードを指定します。

変換個所	対象	文字コード変換時の動作および注意事項
E　J2EE サーバ～ブラウザ	レスポンスボディ	JSP の場合 page ディレクティブの contentType 属性で文字コードを指定します。

参考

「文字化け」について

アプリケーションの実行環境では，文字コードは Unicode として扱われます。このため，ブラウザから送信された文字列は一度 Unicode に変換されます。また，データベースへのアクセス時には Unicode とデータベース格納文字コード間の文字コード変換，レスポンス時には Unicode からレスポンスの文字コードへの変換をする必要があります。これらの文字コード変換時に適切な変換をしないと，文字化けする原因になります。

これはシフト JIS と呼ばれる文字コードに機種依存文字が含まれることや，同じ文字でも Unicode への変換結果がほかの文字コードと異なっているもの※があるために発生します。例えば，ブラウザから機種依存文字を含んだ文字データが送信され，これを Unicode に変換した場合，レスポンス時にこの文字列をシフト JIS に変換すると，文字化けする結果となります。

クライアントの OS を Windows に限定できる場合は，文字コードをシフト JIS ではなく MS932 または Windows-31J と指定することで，文字化けを避けられます。

注※

－，～，‖，¢，£，¬などの文字が該当します。

付録 A.2　ブラウザとアプリケーション間の文字コード変換

ブラウザとアプリケーション間で文字コード変換をするときに使用するメソッド，および JSP と JavaScript の実装時の注意事項について説明します。

(1)　ブラウザとアプリケーション間の文字コード変換で使用するメソッド

ブラウザを使用して送信されたクエリ文字列，POST データは，HttpServletRequest クラスのメソッドを使用して取得します。HttpServletRequest クラスのうち，文字コード変換に関係のあるメソッドと使用時の注意事項を示します。

(a)　setCharacterEncoding メソッド

リクエストのメッセージボディで使われている文字コードを設定します。このメソッドは，getParameter メソッドや，getReader メソッドを使って入力ストリームから読み込む前に実行されなければなりません。

(a) setCharacterEncoding メソッド

リクエストのメッセージボディ，および GET リクエストのクエリで送信されたパラメタに使われている文字コードを設定します。このメソッドは，getParameter メソッドや，getReader メソッドを使って入力ストリームから読み込む前に実行されなければなりません。

setCharacterEncoding メソッドで設定した文字コードが使用される範囲を次の表に示します。

表 A-3　setCharacterEncoding メソッドで設定した文字コードが使用される範囲

メソッド	HTML のフォームから送信されたデータ（クエリも含む）	左記以外のメッセージボディ
getParameter	○	－
getParameterNames	○	－

メソッド	HTML のフォームから送信されたデータ（クエリも含む）	左記以外のメッセージボディ
getParameterValues	○	−
getParameterMap	○	−
getInputStream	×	×
getReader	×	△
getQueryString	×	−

（凡例）

○：取得した値は setCharacterEncoding メソッドで設定した文字コードとして Unicode に変換される。

△：文字データは setCharacterEncoding メソッドで設定した文字コードに変換される。

×：エンコードされたままの文字列が取得される。

−：取得できない。

(b)　getParameter メソッド

パラメタの名称を指定して，リクエストに含まれるパラメタの値を取得します。取得した値は URL デコードされ，Unicode に変換されます。パラメタの名称やパラメタの値を取得する前に，setCharacterEncoding メソッドを用いて文字コードを指定する必要があります。指定しなかった場合，ISO-8859-1 として Unicode に変換されます。HttpServletRequest クラスの getParameterNames メソッド，getParameterValues メソッド，getParameterMap メソッドも同様です。

(c)　getInputStream メソッド

リクエストのメッセージボディに含まれているバイナリデータを読み込むためのストリームを取得します。HTML のフォームから送信されたデータからはエンコードされたままの文字列が取得されるので，取得した文字列を適切な文字コードでデコードする必要があります

(d)　getReader メソッド

リクエストのメッセージボディに BufferedReader クラスを使い，文字データとして取り出します。文字データはメッセージボディと同じ文字コードに変換されます。HTML のフォームから送信されたデータからはエンコードされたままの文字列が取得されるので，取得した文字列を適切な文字コードでデコードする必要があります。

(e)　getQueryString メソッド

リクエストされた URL のパスの後ろに含まれているクエリ文字列を返します。HTML のフォームから送信されたデータからはエンコードされたままの文字列が取得されるので，取得した文字列を適切な文字コードでデコードする必要があります。

(2)　JSP のファイルインクルード時の注意事項

JSP ファイルの include ディレクティブでファイルをインクルードする場合には，インクルード元となる JSP ファイルで contentType 属性を使用してエンコードを指定してください。また，JSP ファイルの文字コードは pageEncoding 属性に指定してください。指定しない場合はインクルード先の文字が正常に表示されない場合があります。

(3)　JavaScript を使用してクエリ文字列を作成する場合の注意事項

- JavaScript の encodeURI 関数および encodeURIComponent 関数※を使用すると，UTF-8 で URL エンコードできます。この関数を用いて変換した文字列をクエリ文字列として使用してください。この場合，J2EE サーバ側で UTF-8 として Unicode に変換すると，日本語の文字列データを取得できます。

  注※

  encodeURI 関数と encodeURIComponent 関数の相違は，「,/?:@&=+$,#」などの特殊文字を変換するかどうかです。encodeURI 関数は「,/?:@&=+$,#」などの特殊文字を変換しません。

- JavaScript の escape 関数はブラウザの種類やバージョンによって動作が異なります。escape 関数で変換された文字列は，サーバ側で正常に扱えないため，escape 関数を使用しないで，encodeURI 関数または encodeURIComponent 関数を使用するようにしてください。

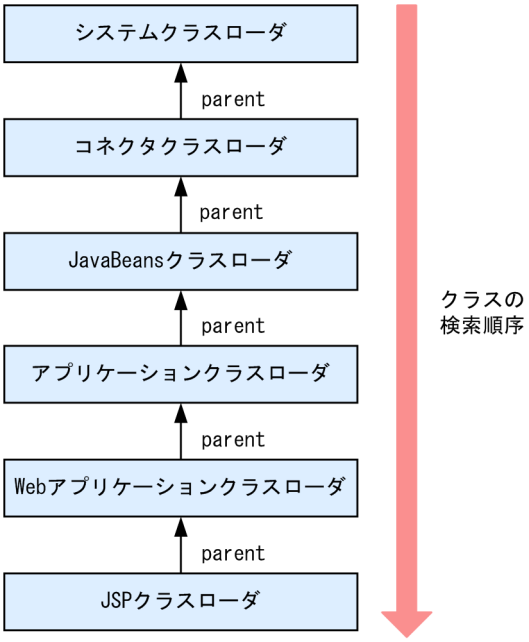
付録 B　クラスローダの構成

アプリケーションサーバのクラスローダ構成には，デフォルトのクラスローダ構成，ローカル呼び出し最適化機能を使用したときのクラスローダ構成，および下位互換用のクラスローダ構成の 3 種類があります。それぞれのクラスローダ構成およびクラスローダに設定されるクラスパスについて説明します。

付録 B.1　デフォルトのクラスローダ構成

新規インストール後のデフォルトのクラスローダ構成を次に示します。

図 B-1　デフォルトのクラスローダ構成



各クラスローダの内容を次に示します。

- システムクラスローダ
  Application Server のコンポーネントが提供するクラスをロードします。
  - 生成タイミング：J2EE サーバ起動時
  - 破棄タイミング：J2EE サーバ停止時
- コネクタクラスローダ
  単体デプロイされたリソースアダプタに含まれるクラスをロードします。J2EE サーバ内に一つだけ存在します。
  - 生成タイミング：J2EE サーバ起動時
  - 破棄タイミング：J2EE サーバ停止時
- JavaBeans クラスローダ
  JavaBeans リソースのクラスをロードします。
  - 生成タイミング：J2EE サーバ起動時
  - 破棄タイミング：J2EE サーバ停止時
- アプリケーションクラスローダ

アプリケーション内の EJB-JAR，ライブラリ JAR，リソースアダプタに含まれるクラスをロードします。アプリケーションごとに存在します。クラス名は com.hitachi.software.ejb.server.ApplicationClassLoader です。

- 生成タイミング：J2EE アプリケーション開始時
  cjmessage?.log（J2EE サーバの稼働ログ）に KDJE42143-I が出力されます（「?」はログの面数）。
- 破棄タイミング：J2EE アプリケーション停止時
  cjmessage?.log（J2EE サーバの稼働ログ）に KDJE42144-I が出力されます（「?」はログの面数）。

- Web アプリケーションクラスローダ

J2EE アプリケーション内の WAR ファイルに含まれるクラスをロードします。WAR ファイルごとに存在します。クラス名は org.apache.catalina.loader.WebappClassLoader です。

- 生成タイミング：J2EE アプリケーション開始時（Web アプリケーション開始時）
  cjmessage?.log（J2EE サーバの稼働ログ）に KDJE39219-I が出力されます（「?」はログの面数）。
- 破棄タイミング：J2EE アプリケーション停止時（Web アプリケーション停止時）
  cjmessage?.log（J2EE サーバの稼働ログ）に KDJE39220-I が出力されます（「?」はログの面数）。

- JSP クラスローダ

JSP ファイルおよびタグファイルをコンパイルしたときに生成されたクラスをロードします。JSP ごとに存在します。

- 生成タイミング：JSP クラスロード時
- 破棄タイミング：J2EE アプリケーション停止時（Web アプリケーション停止時）

ポイント

リソースアダプタのロードについて

J2EE リソースアダプタとしてデプロイしたリソースアダプタは，コネクタクラスローダでロードされます。J2EE アプリケーションに含めて使用するリソースアダプタは，アプリケーションクラスローダでロードされます。

! 注意事項

クラスローダの破棄について

J2EE アプリケーションの停止後，Web アプリケーションクラスローダまたはアプリケーションクラスローダのファイナライズ処理※（finalize メソッドの処理）が実行されたタイミングで，KDJE39220-I または KDJE42144-I のメッセージが出力されます。

このメッセージが出力されたあと，FullGC が発生したタイミングで，クラスローダとアプリケーションはヒープから解放されます。

注※　ファイナライズ処理は，ファイナライズ処理専用のスレッド（ファイナライザスレッド）で実行されます。ファイナライザスレッドは，JavaVM 実行時に常に一つあり，他の Java スレッドと並行して動作して，オブジェクトの finalize()メソッドを一つずつ処理します。

KDJE39220-I または KDJE42144-I のメッセージが出力されない場合，次の原因が考えられます。

原因

(C-1)：アプリケーションクラスローダやアプリケーションへのソフト参照が残っている。

(C-2)：Explicit ヒープに配置されたオブジェクトからアプリケーションクラスローダやアプリケーションへの参照が残っている。

(C-1)～(C-2)の原因について，それぞれ次の(I-1)～(I-2)の原因の切り分けを実施して，メッセージが出力されるか確認してください。

原因の切り分け

(I-1)：-XX:SoftRefLRUPolicyMSPerMB=0 を指定して実行する。

なお，SoftRefLRUPolicyMSPerMB オプションについては，マニュアル「アプリケーションサーバ システム設計ガイド」の「7.9　Java ヒープ内の Metaspace 領域のメモリサイズの見積もり」を参照してください。

(I-2)：明示管理ヒープ機能が有効の場合は，明示管理ヒープ機能を無効(-XX:-HitachiUseExplicitMemory)にする。

なお，明示管理ヒープ機能については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「16.2　JavaVM 拡張オプションの詳細」を参照してください。

(C-1)～(C-2)の原因のどれにも該当しない場合で，アプリケーションの開始または停止を繰り返しても KDJE39220-I または KDJE42144-I メッセージが出力されないときは，メモリリークが発生しているおそれがあります。

付録 B.2　ローカル呼び出し最適化時のクラスローダ構成

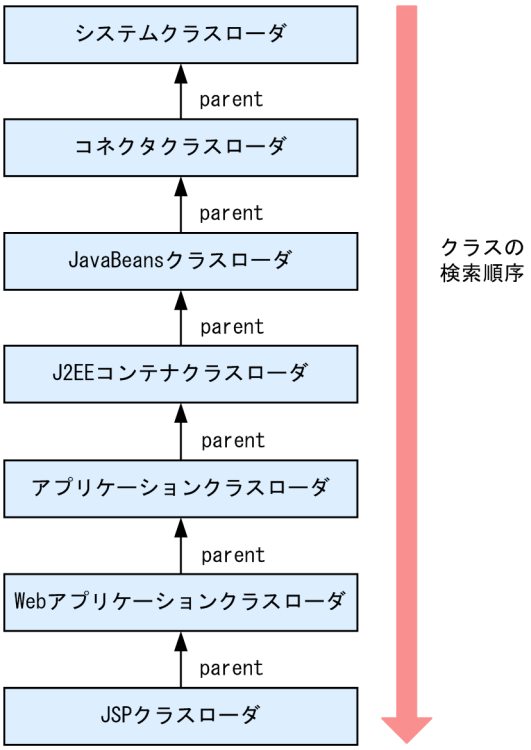
ローカル呼び出し最適化機能を利用する場合，デフォルトのクラスローダとはクラスローダ構成が異なります。ローカル呼び出し最適化機能は，別のアプリケーションの Enterprise Bean を高速で呼び出すことができる機能です。

ローカル呼び出し最適化機能を使用するには，J2EE サーバ用ユーザ定義ファイル（<Application Server のインストールディレクトリ>¥CC¥server¥usrconf¥ejb¥<サーバ名称>¥usrconf.properties）に次の設定をします。

```
ejbserver.rmi.localinvocation.scope=all
```

ローカル呼び出し最適化時のクラスローダ構成を次に示します。

図 B-2　ローカル呼び出し最適化時のクラスローダ構成



各クラスローダの内容を次に示します。

- システムクラスローダ

Application Server のコンポーネントが提供するクラスをロードします。
生成タイミング：J2EE サーバ起動時
破棄タイミング：J2EE サーバ停止時

- コネクタクラスローダ
  単体デプロイされたリソースアダプタに含まれるクラスをロードします。J2EE サーバ内に一つだけ存在します。
  生成タイミング：J2EE サーバ起動時
  破棄タイミング：J2EE サーバ停止時
- JavaBeans クラスローダ
  JavaBeans リソースのクラスをロードします。
  生成タイミング：J2EE サーバ起動時
  破棄タイミング：J2EE サーバ停止時
- J2EE コンテナクラスローダ
  すべてのアプリケーションクラスローダの親クラスローダとなり，すべてのアプリケーションクラスローダが持つクラスパスが設定されます。J2EE コンテナクラスローダは，J2EE サーバ内に一つだけ存在します。
  生成タイミング：J2EE サーバ起動時
  破棄タイミング：J2EE サーバ停止時
- アプリケーションクラスローダ
  アプリケーション内の EJB-JAR，ライブラリ JAR，およびリソースアダプタに含まれるクラスをクラスパスに持ちます。アプリケーションごとに存在します。ローカル呼び出し最適化時には親クラスローダである J2EE コンテナクラスローダにも，アプリケーションクラスローダが持つクラスパスが設定されているため，このクラスローダでは何もロードされません。クラス名は com.hitachi.software.ejb.server.ApplicationClassLoader です。
  - 生成タイミング：J2EE アプリケーション開始時
    cjmessage?.log（J2EE サーバの稼働ログ）に KDJE42143-I が出力されます（「?」はログの面数）。
  - 破棄タイミング：J2EE アプリケーション停止時
    cjmessage?.log（J2EE サーバの稼働ログ）に KDJE42144-I が出力されます（「?」はログの面数）。
- Web アプリケーションクラスローダ
  J2EE アプリケーション内の WAR ファイルに含まれるクラスをロードします。WAR ファイルごとに存在します。クラス名は org.apache.catalina.loader.WebappClassLoader です。
  - 生成タイミング：J2EE アプリケーション開始時（Web アプリケーション開始時）
    cjmessage?.log（J2EE サーバの稼働ログ）に KDJE39219-I が出力されます（「?」はログの面数）。
  - 破棄タイミング：J2EE アプリケーション停止時（Web アプリケーション停止時）
    cjmessage?.log（J2EE サーバの稼働ログ）に KDJE39220-I が出力されます（「?」はログの面数）。
- JSP クラスローダ
  JSP ファイルおよびタグファイルをコンパイルしたときに生成されたクラスをロードします。JSP ごとに存在します。
  - 生成タイミング：JSP クラスロード時
  - 破棄タイミング：J2EE アプリケーション停止時（Web アプリケーション停止時）

クラスローダの破棄についての注意事項は，「付録 B.1　デフォルトのクラスローダ構成」を参照してください。

ポイント

リソースアダプタのロードについて

J2EE リソースアダプタとしてデプロイしたリソースアダプタは，コネクタクラスローダでロードされます。J2EE アプリケーションに含めて使用するリソースアダプタは，アプリケーションクラスローダでロードされます。

! 注意事項

プロパティに ejbserver.rmi.localinvocation.scope=all を設定して実行するときには，次の点に注意してください。

- J2EE アプリケーションが持つ EJB-JAR やライブラリ JAR は同一のクラスローダでロードされるため，同名のクラスは J2EE アプリケーション間で共有されます。内容の異なる同名クラスが存在する場合は，Java の仕様上，先にロードされたクラスだけが有効となります。なお，WAR 内のクラスは WAR 単位に存在するクラスローダによってロードされますので共有の対象外となります。
- J2EE アプリケーションの EJB-JAR やライブラリ JAR を入れ替える場合は, J2EE アプリケーションを停止したあとに J2EE サーバを再起動してください。再起動後に有効となります。

付録 B.3　クラスローダに設定されるクラスパス

クラスローダに設定されるクラスパス, およびクラスローダがクラスをロードするときのクラスの検索順序を示します。

- コネクタクラスローダ

  コネクタクラスローダは単体 RAR のクラスをロードするクラスローダです。単体デプロイ済み RAR ファイルに含まれている JAR ファイルを，クラスパスに設定します。

- J2EE コンテナクラスローダ

  J2EE コンテナクラスローダは，ejbserver.rmi.localinvocation.scope=all（ローカル呼び出し最適化機能を使用する）を指定したときに有効になるクラスローダです。すべてのアプリケーションクラスローダの親クラスローダとなり，開始状態にあるすべてのアプリケーションクラスローダのクラスパスを自分のクラスパスに持ちます。

- アプリケーションクラスローダ

  アプリケーションクラスローダは，J2EE アプリケーション内の EJB-JAR，ライブラリ JAR，およびコンテナ生成クラスをロードします。設定されるクラスパスの順序を次に示します。

  1. J2EE アプリケーションに含まれる RAR に含まれている JAR ファイル
  2. J2EE アプリケーションに含まれる EJB-JAR
  3. 参照方式のライブラリ
  4. J2EE アプリケーションに含まれるライブラリ JAR
  5. Naming 切り替え機能によって参照している別アプリケーションの EJB-JAR ファイル（参照先のアプリケーションが同一 J2EE サーバに存在する場合）

付録 C　JPA プロバイダと EJB コンテナ間の規約

ここでは，JPA プロバイダと EJB コンテナ間の規約について説明します。

付録 C.1　ランタイムに関する規約

ランタイムに関する規約には，コンテナ側の責任と JPA プロバイダ側の責任があります。

(1)　コンテナ側の責任

- **トランザクションスコープの永続化コンテキストに関する規約**

  トランザクションスコープの永続化コンテキストが使用されている場合に，JTA トランザクションにエンティティマネージャが関連づいていないときには，コンテナは次の処理を実行します。

  - コンテナは次の場合に，EntityManagerFactory.createEntityManager を呼び出して新しいエンティティマネージャを作成します。

    JTA トランザクションのスコープ内のビジネスメソッドで，トランザクションスコープの永続化コンテキストを使用するエンティティマネージャのメソッドが初めて呼び出された場合

  - JTA トランザクションが決着（コミットまたはロールバック）したあとに，コンテナは EntityManager.close を呼び出して，エンティティマネージャをクローズします。

  また，コンテナは次の条件をすべて満たすと，TransactionRequiredException をスローします。

  - トランザクションスコープの永続化コンテキストが使用されている場合
  - トランザクションがアクティブでない場合
  - アプリケーションから EntityManager の persist，remove，merge，refresh メソッドが呼び出された場合

- **拡張永続化コンテキストに関する規約**

  拡張永続化コンテキストが使用されている場合は，コンテナは次の処理を行います。

  - コンテナは次の場合に，EntityManagerFactory.createEntityManager を呼び出して新しいエンティティマネージャを作成します。

    Stateful Session Bean のインスタンスが作成された際に，拡張永続化コンテキストを使用するエンティティマネージャへの参照が定義されている場合

  - コンテナは次のタイミングで，EntityManager.close を呼び出してエンティティマネージャをクローズします。

    エンティティマネージャを作成した Stateful Session Bean，および同じ永続化コンテキストを引き継いだ Stateful Session Bean が削除されたとき

  - コンテナ管理のトランザクションを使用する Stateful Session Bean のビジネスメソッドの呼び出し時に，エンティティマネージャが JTA トランザクションと関連づけられていない場合には，コンテナはエンティティマネージャを JTA トランザクションに関連づけ，EntityManager.joinTransaction を呼び出します。JTA トランザクションにすでに別のエンティティマネージャが関連づけられている場合には，コンテナは EJBException をスローします。

  - ビーン管理のトランザクションを使用する Stateful Session Bean のビジネスメソッド内で UserTransaction.begin が呼び出された場合には，コンテナはエンティティマネージャを JTA トランザクションと関連づけ，EntityManager.joinTransaction を呼び出します。なお，ビーン管理のトランザクションについては，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「2.7.2　BMT」を参照してください。

- **コンテナ管理のエンティティマネージャに関する規約**

  コンテナ管理のエンティティマネージャを使用している場合に，アプリケーションが EntityManager.close を呼び出すと，コンテナは IllegalStateException をスローします。

  @PersistenceCotnext または DD の<persistence-context-ref>タグで，プロパティが指定された場合には，コンテナは EntityManagerFactory.createEntityManager(Map map)メソッドを使用してエンティティマネージャを作成し，指定されたプロパティを map 引数に含めて JPA プロバイダに渡します。

- **コネクションの自動クローズ機能について**

  アプリケーションサーバの場合，Session Bean や Web コンポーネントの延長で JPA プロバイダが取得したコネクションは，コンテナの自動クローズ機能によって自動的にクローズされることがあります。コネクションの自動クローズ機能とは，コネクションのリークを防止するための機能です。

  コネクションは，次の条件を満たすと自動クローズの対象となります。

  - Stateless Session Bean で取得したコネクションは，ビジネスメソッドからリターンするときに，自動クローズの対象となります。
  - Stateful Session Bean で取得したコネクションは，Stateful Session Bean が破棄されるときに，自動クローズの対象となります。
  - Web コンポーネントで取得したコネクションは，サービスメソッドからリターンするときに，自動クローズの対象となります。

  ただし，これらの条件に当てはまる場合でも，コネクションが JTA トランザクションに参加しているときには，JTA トランザクションがコミットするまで自動クローズが保留されます。

(2)　JPA プロバイダ側の責任

アプリケーションで使用するエンティティマネージャが，トランザクションスコープの永続化コンテキストを使用するように定義されているか，拡張永続化コンテキストを使用するように定義されているかは，JPA プロバイダには伝わりません。JPA プロバイダでの責任は，コンテナが要求したときにエンティティマネージャを作成し，トランザクションからトランザクションの決着の通知を受け取るための Synchronization をトランザクションに登録することです。

- コンテナが EntityManagerFactory.createEntityManager を呼び出したときには，JPA プロバイダは新しいエンティティマネージャを作成し，それをコンテナに返す必要があります。JTA トランザクションがアクティブである場合は，JPA プロバイダは Synchronization を JTA トランザクションに登録する必要があります。
- コンテナが EntityManager.joinTransaction を呼び出したときには，JPA プロバイダは Synchronization を JTA トランザクションに登録する必要があります。ただし，以前に joinTransaction が呼び出されていて，JTA トランザクションに Synchronization を登録済みの場合には，何もする必要はありません。
- JTA トランザクションがコミットされるときには，JPA プロバイダはすべての変更されたエンティティの状態を，データベースにフラッシュする必要があります。
- JTA トランザクションがロールバックされるときには，JPA プロバイダは，すべての managed 状態のエンティティをデタッチする必要があります。
- JPA プロバイダがトランザクションのロールバックの原因になる例外をスローする場合には，JPA プロバイダがトランザクションをロールバックにマークする必要があります。
- コンテナが EntityManager.close を呼び出した場合，そのエンティティマネージャが関係したすべての未解決トランザクションが決着されたあとに，JPA プロバイダはすべての確保したリソースを解放する必要があります。エンティティマネージャがすでにクローズされている場合には，JPA プロバイダは IllegalStateException をスローする必要があります。

- コンテナが EntityManager.clear を呼び出した場合, JPA プロバイダはすべての managed 状態のエンティティをデタッチする必要があります。

(3)　javax.transaction.TransactionSynchronizationRegistry インタフェース

JPA プロバイダは，トランザクションに Synchronization を登録したり，トランザクションをロールバックにマークしたりするために，TransactionSynchronizationRegistry インタフェースを使用できます。TransactionSynchronizationRegistry のインスタンスは，JNDI を使用して「java:comp/TransactionSynchronizationRegistry」という名前でルックアップできます。

インタフェース定義を次に示します。

```
package javax.transaction;

/**
 * このインタフェースは，JPAプロバイダやリソースアダプタなど，
 * アプリケーションサーバのシステムレベルのコンポーネントから使用する
 * ためのものです。
 * このインタフェースを使って，
 * 特別な順序で呼ばれるシンクロナイゼーションの登録，
 * 現在のトランザクションへのリソースオブジェクトの登録，
 * 現在のトランザクションのコンテキストの取得，
 * 現在のトランザクションのステータスの取得，
 * 現在のトランザクションをロールバックにマークできます。
 *
 * このインタフェースはステートレスなサービスオブジェクトとして，
 * アプリケーションサーバによって実装されます。
 * 同じオブジェクトを複数のコンポーネントからマルチスレッドセーフに
 * 使用できます。
 *
 * 標準的なアプリケーションサーバでは，このインタフェースを実装した
 * インスタンスは，JNDIを使用して標準的な名前でルックアップできます。
 * 標準的な名前は
 * 「java:comp/TransactionSynchronizationRegistry」です。
 */
public interface TransactionSynchronizationRegistry {

    /**
     * このメソッドを呼び出した時に現在のスレッドに関連づいている
     * トランザクションを表現する一意のオブジェクトを返します。
     * このオブジェクトのhashCodeとequalsメソッドは
     * オーバーライドされており，
     * ハッシュマップのキーとして使用できます。
     * トランザクションが存在しない場合には，nullを返します。
     *
     * 同じアプリケーションサーバの同じトランザクションコンテキスト内で
     * このメソッドが呼ばれて返されたオブジェクトは，すべて同じhashCodeを
     * 持ち，equalメソッドでの比較結果はtrueとなります。
     *
     * toStringメソッドは，トランザクションコンテキストの情報を人が
     * 読みやすい形の文字列として返します。
     * ただし，toStringで返される文字列のフォーマットは規定されていません。
     * また，toStringの結果に関して，バージョン間での互換性は
     * 保障されていません。
     *
     * 取得したオブジェクトがシリアライズできる保障はなく，
     * JavaVMの外に出したときの動作は規定されていません。
     *
     * @return このメソッドが呼ばれたときにスレッドに関連づいている
     * トランザクションを一意に表現するオブジェクト
     */
    Object getTransactionKey();

    /**
     * このメソッドを呼び出した時のスレッドに関連づいているトランザクション
     * のリソースマップに，オブジェクトを追加またはリプレースします。
     * マップのキーは，コンフリクトが発生しないように，
     * このメソッドを呼び出す側で定義されたクラスである必要があります。
     * キーとして使用するクラスは，マップのキーとして適切なhashCodeとequals
     * メソッドを持っている必要があります。
```

```
 * マップのキーや値が，このクラスによって評価されたり，使用されたりする
 * ことはありません。
 * このメソッドの一般的な規約は，Mapのputメソッドと同じで，
 * キーはnull以外にする必要がありますが，値はnullにすることもできます。
 * すでにキーに関連づいた値が存在する場合，その値は置き換えられます。
 *
 * @param key マップのエントリのキー
 * @param value マップのエントリの値
 * @exception IllegalStateException アクティブなトランザクションが
 * 存在しない場合
 * @exception NullPointerException 引数のキーがnullである場合
 */
void putResource(Object key, Object value);

/**
 * このメソッドを呼び出した時のスレッドに関連づいているトランザクション
 * のリソースマップから，オブジェクトを取り出します。
 * キーは，同じトランザクション内で，事前にputResourceメソッドに
 * 指定したオブジェクトと同じである必要があります。
 * 指定されたキーが現在のリソースマップに存在しない場合は，
 * nullを返します。
 * このメソッドの一般的な規約は，Mapのputメソッドと同じで，
 * キーはnull以外にする必要がありますが，値はnullにすることもできます。
 * マップにキーが格納されていない場合や，キーに対してnullが格納
 * されている場合には，返り値はnullになります。
 * @param key マップのエントリのキー
 * @return キーに関連づけられた値
 * @exception IllegalStateException アクティブなトランザクションが
 * 存在しない場合
 * @exception NullPointerException 引数のキーがnullである場合
 */
Object getResource(Object key);

/**
 * 特別な順序で呼ばれるシンクロナイゼーションのインスタンスを登録します。
 * このメソッドで登録したSynchronizationのbeforeCompletionは，
 * すべてのSessionSynchronization.beforeCompletionや，
 * トランザクションに直接登録された
 * Synchronization.beforeCompletionが呼ばれたあとで，
 * 2フェーズコミット処理が開始される前に呼び出されます。
 * 同じように，このメソッドで登録した
 * SynchronizationのafterCompletionは，
 * 2フェーズコミット処理が完了したあとで，
 * すべてのSessionSynchronization.afterCompletionや，
 * トランザクションに直接登録されたSynchronization.afterCompletionが
 * 呼ばれる前に呼び出されます。
 *
 * beforeCompletionは，このメソッドを呼び出した時にスレッドに
 * 関連づいていたトランザクションのコンテキストで呼び出されます。
 * beforeCompletionでは，コネクタなどのリソースへのアクセスは
 * 許可されていますが，タイマーサービスやビーンのメソッドなどの
 * ユーザコンポーネントへのアクセスは許可されていません。
 * これは，呼び出し側によって管理されているデータや，
 * registerInterposedSynchronizationで登録された
 * ほかのSynchronizationによって
 * すでにフラッシュされたデータを変更してしまうおそれがあるためです。
 * 一般的なコンテキストは，registerInterposedSynchronizationを
 * 呼び出したコンポーネントのコンテキストになります。
 *
 * afterCompletionが呼び出される時のコンテキストは定義されていません。
 * なお，ユーザコンポーネントへのアクセスは許可されていません。
 * また，リソースをクローズすることはできますが，
 * リソースに対するトランザクショナルな操作はできません。
 *
 *トランザクションがアクティブでない場合にこのメソッドが呼び出されたとき，
 * IllegalStateExceptionがスローされます。
 *
 * このメソッドが2フェーズコミット処理の開始後に呼び出された場合は，
 * IllegalStateExceptionがスローされます。
 *
 * @param sync 登録するSynchronizationのインスタンス
 * @exception IllegalStateException アクティブなトランザクションが
 * 存在しない場合
 */
```

```
    void registerInterposedSynchronization(Synchronization sync);

    /**
     * このメソッドを呼んだ時にスレッドに関連づいているトランザクション
     * のステータスを返します。
     * このメソッドの返り値は,
     * TransactionManager.getStatus()の結果と同じです。
     *
     * @return このメソッドを呼んだときにスレッドに関連づいている
     * トランザクションのステータス
     */
    int getTransactionStatus();

    /**
     * このメソッドを呼んだ時にスレッドに関連づいているトランザクション
     * がロールバックされるようにマークします。
     *
     * @exception IllegalStateException アクティブなトランザクションが
     * 存在しない場合
     */
    void setRollbackOnly();

    /**
     * このメソッドを呼んだ時にスレッドに関連づいているトランザクション
     * がロールバックするようにマークされているかどうかを返します。
     *
     * @return ロールバックするようにマークされている場合はtrue
     * @exception IllegalStateException アクティブなトランザクションが
     * 存在しない場合
     */
    boolean getRollbackOnly();
}
```

付録 C.2　デプロイメントに関する規約

デプロイメントに関する規約には，コンテナ側の責任と JPA プロバイダ側の責任があります。

(1)　コンテナ側の責任

デプロイメント時に，コンテナはアプリケーション内の決められた場所にパッケージングされた persistence.xml を検索します。アプリケーション内に persistence.xml が存在する場合には，コンテナは persistence.xml に定義された永続化ユニットの定義を処理します。なお，コンテナが検索する場所については，「5.8　persistence.xml での定義」を参照してください。

コンテナは persistence.xml ファイルを persistence_1_0.xsd で検証します。検証の結果, エラーが発生した場合にはユーザに通知します。persistence.xml にプロバイダやデータソースの情報が指定されていない場合には，デフォルト値が使用されます。使用されるデフォルト値については，「5.8　persistence.xml での定義」を参照してください。コンテナが永続化ユニットのエンティティマネージャファクトリを作成するときには，JPA プロバイダにプロパティを渡すことがあります。

コンテナは，persistence.xml で永続化ユニットごとに定義された javax.persistence.spi.PersistenceProvider の実装クラスのインスタンスを作成し, createContainerEntityManagerFactory メソッドを呼び出して, コンテナ管理のエンティティマネージャを作成するための EntityManagerFactory を取得します。永続化ユニットのメタデータは，PersistenceUnitInfo オブジェクトとして，createContainerEntityManagerFactory メソッドの引数で JPA プロバイダに渡されます。コンテナは一つの永続化ユニット定義に対して，一つだけ EntityManagerFactory を作成し，その EntityManagerFactory から複数の EntityManager を作成します。

永続化ユニットが再デプロイされる場合には，コンテナはすでに取得した EntityManagerFactory の close メソッドを呼び出したあと，createContainerEntityManagerFactory を新しい PersistenceUnitInfo とともに呼び出します。

(2)　JPA プロバイダ側の責任

JPA プロバイダは PersistenceProvider SPI を実装し，PersistenceProvider の createContainerEntityManagerFactory メソッドが呼ばれたときに，引数で渡される永続化ユニットのメタデータ（PersisetnceUnitInfo）を使用して，EntityManagerFactory を作成し，コンテナに返す必要があります。

JPA プロバイダは，永続化ユニットに含まれるマネージドクラス（エンティティクラスなど）のメタデータアノテーションを処理します。また，永続化ユニットで O/R マッピングファイルが使用されている場合には，JPA プロバイダが解釈する必要があります。このとき，O/R マッピングファイルを orm_1_0.xsd を使用して検証し，エラーが発生した場合はユーザに通知する必要があります。

(3)　javax.persistence.spi.PersistenceProvider インタフェース

JPA プロバイダは，javax.persistence.spi.PersistenceProvider インタフェースを実装する必要があります。このインタフェースはコンテナによって呼び出されるものであり，アプリケーションから呼び出すものではありません。PersistenceProvider の実装クラスは，public で引数のないコンストラクタを持っている必要があります。

javax.persistence.spi.PersistenceProvider インタフェースを次に示します。

```
package javax.persistence.spi;

/**
* JPAプロバイダによって実装されるインタフェースです。
* このインタフェースはEntityManagerFactoryを作成するために使用されます。
* Java EE環境ではコンテナによって呼び出され，
* JavaSE環境ではPersistenceクラスによって呼び出されます。
*/
public interface PersistenceProvider {

  /**
  * PersistenceクラスがEntityManagerFactoryを作成する時に呼び出されます。
  *
  * @param emName 永続化ユニットの名前
  * @param map JPAプロバイダによって使用されるプロパティのMap。
  * ここに指定されたプロパティは，persistence.xmlファイルの対応する
  * プロパティを上書きするため，またはpersistence.xmlファイルに
  * 指定されていないプロパティを指定するために使用されます。
  * （プロパティを指定する必要がない場合はnullが渡されます）
  * @return 永続化ユニットのEntityManagerFactory
  * JPAプロバイダが正しくない場合は，nullを返します。
  */
  public EntityManagerFactory createEntityManagerFactory(String
  emName, Map map);

  /**
  * コンテナがEntityManagerFactoryを作成する時に呼びだされます。
  *
  * @param info JPAプロバイダが使用するためのメタデータ
  * @param map JPAプロバイダが使用するためのインテグレーションレベルの
  * プロパティ（指定しない場合はnullが渡されます）
  * @return メタデータで指定された永続化ユニットのEntityManagerFactory
  */
  public EntityManagerFactory createContainerEntityManagerFactory(
  PersistenceUnitInfo info, Map map);
}
```

(4)　javax.persistence.spi.PersistenceUnitInfo インタフェース

javax.persistence.spi.PersistenceUnitInfo インタフェースの定義を次に示します。

```
import javax.sql.DataSource;
/**
* このインタフェースはコンテナによって実装され，
```

```
* EntityManagerFactoryを作成する時にJPAプロバイダに渡されて使用される。
*/
public interface PersistenceUnitInfo {

  /**
  * @return persistence.xmlで定義された永続化ユニット名
  */
  public String getPersistenceUnitName();

  /**
  * @return persistence.xmlの<provider>エレメントに定義された,
  * JPAプロバイダ実装クラスの完全修飾クラス名
  */
  public String getPersistenceProviderClassName();

  /**
  * @return EntityManagerFactoryによって作成されるEntityManagerの
  * トランザクションのタイプを返す。
  * persistence.xmlのtransaction-type属性に指定されたタイプである。
  */
  public PersistenceUnitTransactionType getTransactionType();

  /**
  * @return JPAプロバイダが使用するためのJTAが有効な
  * データソースを返す。
  * persistence.xmlの<jta-data-source>エレメントで指定されたデータソースか,
  * デプロイメント時にコンテナによって決定されたデータソースである。
  */
  public DataSource getJtaDataSource();

  /**
  * @return JPAプロバイダがJTAトランザクションの外で
  * データにアクセスするための, JTAが無効なデータソースを返す。
  * persistence.xmlの<non-jta-data-source>エレメントで指定された
  * データソースか, デプロイメント時にコンテナによって決定された
  * データソースである。
  */
  public DataSource getNonJtaDataSource();

  /**
  * @return JPAプロバイダがエンティティクラスのマッピングを
  * 決定するためにロードする必要のある, マッピングファイル名のリスト
  * マッピングファイルは, 標準的なXML形式のマッピングフォーマット
  * でなければならない。
  * 一意の名前を持ち, アプリケーションのクラスパスからリソースとして
  * ロードできるものでなければならない。
  * それぞれのマッピングファイル名は, persistence.xmlの<mapping-file>タグ
  * に指定されたものである。
  */
  public List<String> getMappingFileNames();

  /**
  * JPAプロバイダが, 永続化ユニットのマネージドクラスを検索する必要のある,
  * JARファイルまたはJARファイルを展開したディレクトリのURLのリストを返す。
  * それぞれのURLはpersistence.xmlファイルの<jar-file>タグに指定された
  * ものである。
  * URLはJARファイルまたはJARファイルを展開したディレクトリを指す
  * ファイルURLか, またはJARのフォーマットのInputStreamを取得できる
  * そのほかのURL形式である。
  *
  * @return JARファイルまたはディレクトリを指すURLオブジェクトのリスト
  */
  public List<URL> getJarFileUrls();

  /**
  * 永続化ユニットルートであるJARファイルまたはディレクトリのURLを返す。
  * （永続化ユニットルートがWEB-INF/classesディレクトリである場合,
  * WEB-INF/classesディレクトリのURLを返す）
  * URLはJARファイルまたはJARファイルを展開したディレクトリを指す
  * ファイルURLか, またはJARのフォーマットのInputStreamを取得できる
  * そのほかのURL形式である。
  *
  * @return JARファイルまたはディレクトリを指すURLオブジェクト
  */
```

```
    public URL getPersistenceUnitRootUrl();

    /**
    * @return JPAプロバイダがマネージドクラスとして扱わなくてはならない
    * クラス名のリスト。
    * それぞれのクラス名はpersistence.xmlファイルの<class>タグに指定された
    * ものである。
    */
    public List<String> getManagedClassNames();

    /**
    * @return 永続化ユニットルートに配置されていて,
    * 明示的にマネージドクラスであると指定されていないクラスを,
    * マネージドクラスとして扱うかどうかを返す。
    * この値は, persistence.xmlファイルの<exclude-unlisted-classes>タグに
    * 指定されたものである。
    */
    public boolean excludeUnlistedClasses();

    /**
    * @return Propertiesオブジェクト。
    * それぞれのプロパティはpersistence.xmlファイルの<property>タグに
    * 指定されたものである。
    */
    public Properties getProperties();

    /**
    * @return JPAプロバイダがクラスやリソースをロードしたり,
    * URLをオープンしたりするために使用できるClassLoader。
    */
    public ClassLoader getClassLoader();

    /**
    * PersistenceUnitInfo.getClassLoaderメソッドによって返されるクラスローダで,
    * 新しいクラスの定義や, クラスの再定義のたびに呼ばれる,
    * JPAプロバイダのトランスフォーマーを登録する。
    * このトランスフォーマーは, PersistenceUnitInfo.getNewTempClassLoaderメソッドで返される
    * クラスローダでロードされるクラスには影響を与えない。
    * クラスローディングのスコープ内で永続化ユニットが幾つ定義されていても,
    * 同じクラスローディングのスコープ内でクラスが変換されるのは一度だけである。
    *
    * @param transformer コンテナがクラスの定義（再定義）時に呼び出す,
    * JPAプロバイダのトランスフォーマー
    */
    public void addTransformer(ClassTransformer transformer);

    /**
    * JPAプロバイダが一時的にクラスやリソースをロードしたり,
    * URLをオープンしたりするために使用できるクラスローダの
    * 新しいインスタンスを返す。
    * このクラスローダのスコープやクラスパスは,
    * PersistenceUnitInfo.getClassLoaderで返される
    * クラスローダと完全に同じである。
    * このクラスローダでロードしたクラスが,
    * アプリケーションのコンポーネントから参照できるようになることはない。
    * JPAプロバイダは, createContainerEntityManagerFactoryの
    * 呼び出しの延長でだけ, このクラスローダを使用できる。
    *
    * @return 現在のクラスローダと同じスコープ・クラスパスを持つ一時的な
    * クラスローダ
    */
    public ClassLoader getNewTempClassLoader();

}
```

参考

アプリケーションサーバでは，永続化ユニットで JTA データソース，非 JTA データソースが定義されていない場合には，getJtaDataSource()または getNonJtaDataSource()は null を返します。永続化ユニットで JTA データソース，非 JTA データソースが定義されていない場合とは，次の状態を指します。

- persistence.xml で<jta-data-source>，<non-jta-data-source>が省略されていて，かつデフォルト値がシステムプロパティ ejbserver.jpa.defaultJtaDsName，ejbserver.jpa.defaultNonJtaDsName で定義されていない場合
- システムプロパティ ejbserver.jpa.overrideJtaDsName，ejbserver.jpa.overrideNonJtaDsName も定義されていない場合

付録 D　JPQL の BNF

ここでは，JPA1.0 仕様で規定されている JPQL の BNF について示します。

表 D-1　BNF 表記の規則

表記	内容
{ }	グループを表します。
[ ]	オプションの構文を表します。
*	0 以上を表します。
A|B	A または B を表します。

なお，太字部分はキーワードを表します。

次に，BNF を示します。

```
QL_statement ::= select_statement | update_statement | delete_statement
select_statement ::= select_clause from_clause [where_clause] [groupby_clause]
[having_clause] [orderby_clause]
update_statement ::= update_clause [where_clause]
delete_statement ::= delete_clause [where_clause]
from_clause ::=
FROM identification_variable_declaration
{, {identification_variable_declaration | collection_member_declaration}}*
identification_variable_declaration ::= range_variable_declaration { join | fetch_join }*
range_variable_declaration ::= abstract_schema_name [AS] identification_variable
join ::= join_spec join_association_path_expression [AS] identification_variable
fetch_join ::= join_spec FETCH join_association_path_expression
association_path_expression ::=
collection_valued_path_expression | single_valued_association_path_expression
join_spec::= [ LEFT [OUTER] | INNER ] JOIN
join_association_path_expression ::= join_collection_valued_path_expression |
join_single_valued_association_path_expression
join_collection_valued_path_expression::=
identification_variable.collection_valued_association_field
join_single_valued_association_path_expression::=
identification_variable.single_valued_association_field
collection_member_declaration ::=
IN (collection_valued_path_expression) [AS] identification_variable
single_valued_path_expression ::=
state_field_path_expression | single_valued_association_path_expression
state_field_path_expression ::=
{identification_variable | single_valued_association_path_expression}.state_field
single_valued_association_path_expression ::=
identification_variable.{single_valued_association_field.}* single_valued_association_field
collection_valued_path_expression ::=
identification_variable.{single_valued_association_field.}*collection_valued_association_field
state_field ::= {embedded_class_state_field.}*simple_state_field
update_clause ::= UPDATE abstract_schema_name [[AS] identification_variable]
SET update_item {, update_item}*
update_item ::= [identification_variable.]{state_field | single_valued_association_field} =
new_value
new_value ::=
simple_arithmetic_expression |
string_primary |
datetime_primary |
boolean_primary |
enum_primary
simple_entity_expression |
NULL
delete_clause ::= DELETE FROM abstract_schema_name [[AS] identification_variable]
select_clause ::= SELECT [DISTINCT] select_expression {, select_expression}*
select_expression ::=
single_valued_path_expression |
```

```
aggregate_expression |
identification_variable |
OBJECT(identification_variable) |
constructor_expression
constructor_expression ::=
NEW constructor_name ( constructor_item {, constructor_item}* )
constructor_item ::= single_valued_path_expression | aggregate_expression
aggregate_expression ::=
{ AVG | MAX | MIN | SUM } ([DISTINCT] state_field_path_expression) |
COUNT ([DISTINCT] identification_variable | state_field_path_expression |
single_valued_association_path_expression)
where_clause ::= WHERE conditional_expression
groupby_clause ::= GROUP BY groupby_item {, groupby_item}*
groupby_item ::= single_valued_path_expression | identification_variable
having_clause ::= HAVING conditional_expression
orderby_clause ::= ORDER BY orderby_item {, orderby_item}*
orderby_item ::= state_field_path_expression [ ASC | DESC ]
subquery ::= simple_select_clause subquery_from_clause [where_clause]
[groupby_clause] [having_clause]
subquery_from_clause ::=
FROM subselect_identification_variable_declaration
{, subselect_identification_variable_declaration}*
subselect_identification_variable_declaration ::=
identification_variable_declaration |
association_path_expression [AS] identification_variable |
collection_member_declaration
simple_select_clause ::= SELECT [DISTINCT] simple_select_expression
simple_select_expression::=
single_valued_path_expression |
aggregate_expression |
identification_variable
conditional_expression ::= conditional_term | conditional_expression OR conditional_term
conditional_term ::= conditional_factor | conditional_term AND conditional_factor
conditional_factor ::= [ NOT ] conditional_primary
conditional_primary ::= simple_cond_expression | (conditional_expression)
simple_cond_expression ::=
comparison_expression |
between_expression |
like_expression |
in_expression |
null_comparison_expression |
empty_collection_comparison_expression |
collection_member_expression |
exists_expression
between_expression ::=
arithmetic_expression [NOT] BETWEEN
arithmetic_expression AND arithmetic_expression |
string_expression [NOT] BETWEEN string_expression AND string_expression |
datetime_expression [NOT] BETWEEN
datetime_expression AND datetime_expression
in_expression ::=
state_field_path_expression [NOT] IN ( in_item {, in_item}* | subquery)
in_item ::= literal | input_parameter
like_expression ::=
string_expression [NOT] LIKE pattern_value [ESCAPE escape_character]
null_comparison_expression ::=
{single_valued_path_expression | input_parameter} IS [NOT] NULL
empty_collection_comparison_expression ::=
collection_valued_path_expression IS [NOT] EMPTY
collection_member_expression ::= entity_expression
[NOT] MEMBER [OF] collection_valued_path_expression
exists_expression::= [NOT] EXISTS (subquery)
all_or_any_expression ::= { ALL | ANY | SOME} (subquery)
comparison_expression ::=
string_expression comparison_operator {string_expression | all_or_any_expression} |
boolean_expression { =|<>} {boolean_expression | all_or_any_expression} |
enum_expression { =|<>} {enum_expression | all_or_any_expression} |
datetime_expression comparison_operator
{datetime_expression | all_or_any_expression} |
entity_expression { = | <> } {entity_expression | all_or_any_expression} |
arithmetic_expression comparison_operator
{arithmetic_expression | all_or_any_expression}
comparison_operator ::= = | > | >= | < | <= | <>
arithmetic_expression ::= simple_arithmetic_expression | (subquery)
```

```
simple_arithmetic_expression ::=
arithmetic_term | simple_arithmetic_expression { + | - } arithmetic_term
arithmetic_term ::= arithmetic_factor | arithmetic_term { * | / } arithmetic_factor
arithmetic_factor ::= [{ + | - }] arithmetic_primary
arithmetic_primary ::=
state_field_path_expression |
numeric_literal |
(simple_arithmetic_expression) |
input_parameter |
functions_returning_numerics |
aggregate_expression
string_expression ::= string_primary | (subquery)
string_primary ::=
state_field_path_expression |
string_literal |
input_parameter |
functions_returning_strings |
aggregate_expression
datetime_expression ::= datetime_primary | (subquery)
datetime_primary ::=
state_field_path_expression |
input_parameter |
functions_returning_datetime |
aggregate_expression
boolean_expression ::= boolean_primary | (subquery)
boolean_primary ::=
state_field_path_expression |
boolean_literal |
input_parameter |
enum_expression ::= enum_primary | (subquery)
enum_primary ::=
state_field_path_expression |
enum_literal |
input_parameter |
entity_expression ::=
single_valued_association_path_expression | simple_entity_expression
simple_entity_expression ::=
identification_variable |
input_parameter
functions_returning_numerics::=
LENGTH(string_primary) |
LOCATE(string_primary, string_primary[, simple_arithmetic_expression]) |
ABS(simple_arithmetic_expression) |
SQRT(simple_arithmetic_expression) |
MOD(simple_arithmetic_expression, simple_arithmetic_expression) |
SIZE(collection_valued_path_expression)
functions_returning_datetime ::=
CURRENT_DATE|
CURRENT_TIME |
CURRENT_TIMESTAMP
functions_returning_strings ::=
CONCAT(string_primary, string_primary) |
SUBSTRING(string_primary,
simple_arithmetic_expression, simple_arithmetic_expression)|
TRIM([[trim_specification] [trim_character] FROM] string_primary) |
LOWER(string_primary) |
UPPER(string_primary)
trim_specification ::= LEADING | TRAILING | BOTH
```

付録 E　CJMS プロバイダのユースケース

ここでは，CJMS プロバイダを使用する場合のユースケースについて説明します。なお，ここで説明するユースケース以外の運用をした場合の動作は保証しません。

CJMS プロバイダを使用する場合は，ここで説明する手順に従って，環境構築および運用を実行してください。ここで示す以外の手順での構築・運用を実行した場合，CJMS プロバイダが正しく動作しないおそれがあります。

説明するユースケースの種類を次の表に示します。

表 E-1　CJMS プロバイダのユースケースの種類

分類	作業	作業内容	作業詳細	参照先
構築	新規環境構築	新規に環境を構築します。	CJMS プロバイダを使用する場合の環境構築	付録 E.3
	環境再構築	環境を再構築します。	CJMS プロバイダを使用するアプリケーションの追加	付録 E.4
			CJMS プロバイダを使用するアプリケーションの削除	付録 E.5
運用	定期運用	CJMS プロバイダを使用する場合に定期的（日々またはその他の運用タイミング）で必要となる運用を実行します。	CJMS プロバイダサービスの開始（初回起動時）	付録 E.6
			CJMS プロバイダサービスの開始（稼働中システムの再起動時）	付録 E.7
			CJMSP リソースアダプタと CJMSP ブローカーの状態確認	付録 E.8
			メッセージ配信状況の確認と滞留時の対処（CJMSP ブローカーを一時停止する方法）	付録 E.9
			メッセージ配信状況の確認と滞留時の対処（アプリケーションを停止する方法）	付録 E.10
			CJMS プロバイダサービスの終了	付録 E.11
	非定期運用	CJMS プロバイダを使用する場合に非定期で必要となる運用を実行します。	送信先の圧縮	付録 E.12
			送信先サイズの変更	付録 E.13
			永続化サブスクライバーの削除	付録 E.14
	監視作業	CJMS プロバイダで使用されるリソースを監視します。	CJMSP ブローカーの状態監視	付録 E.15
			CJMSP ブローカーの詳細情報確認	付録 E.16
			送信先の状態確認	付録 E.17

分類	作業	作業内容	作業詳細	参照先
運用	監視作業	CJMS プロバイダで使用されるリソースを監視します。	永続化サブスクライバーの状態確認	付録 E. 18
障害編	障害解析手順	CJMS プロバイダを使用中に障害が発生した場合に障害要因を解析します。	CJMSP リソースアダプタに問題が発生した場合の解析	付録 E. 19
			CJMSP ブローカーが障害によって停止したときの解析	付録 E. 20
			CJMS プロバイダサービス無応答時の解析	付録 E. 21
	障害回復手順	CJMS プロバイダを使用中に障害が発生した場合に回復します。	CJMSP リソースアダプタに問題が発生した場合の回復	付録 E. 22
			CJMSP ブローカーが障害によって停止したときの回復	付録 E. 23
			CJMS プロバイダサービス無応答時の回復	付録 E. 24
削除	環境削除	CJMS プロバイダの環境を削除します。	CJMS プロバイダサービスインスタンスの削除	付録 E. 25

付録 E.1　すべてのユースケースに共通の前提条件

すべてのユースケースに共通の前提条件を次に示します。

- CJMS プロバイダに関連するコマンドの操作でユーザ確認をしないで強制的に実行したい場合，cjmsbroker コマンドの場合は-force オプション，cjmsicmd コマンドの場合は-f オプションを指定してください。
- CJMS プロバイダに関連するコマンドの操作は CJMSP ブローカーが起動しているマシンで実行してください。
- CJMS プロバイダを使用する場合には，ユーザ権限に Administrator 権限が必要となります。
- CJMS プロバイダに関連するコマンドの操作は複数同時に実行しないでください。
- CJMS プロバイダのコマンドラインツール（cjmsbroker, cjmsicmd）に関するヘルプを表示するには，-h オプションまたは-help オプションを指定してください。
- J2EE サーバの起動オプションはセキュリティマネージャを使用しない設定にしてください。使用するに設定した場合，CJMSP リソースアダプタからリソースアクセスした際に権限不正のエラーが発生します。
- J2EE サーバで使用するコマンドの操作でプロバイダ URL を指定する場合は，次の例に従ってください。

  実行形式

  ```
  cjlistrar 〈サーバ名称〉 -nameserver 〈プロバイダURL〉
  ```

  実行例

  ```
  cjlistrar MyServer -nameserver corbaname::localhost:900
  ```

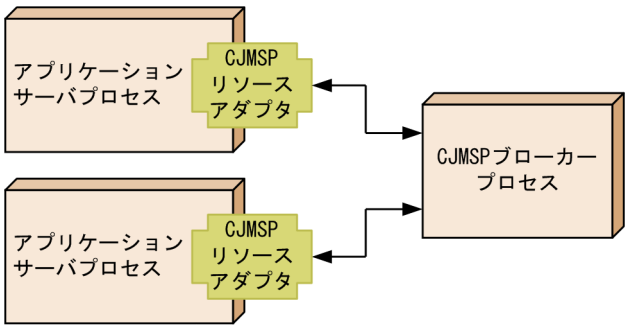
  <プロバイダ URL>には，<プロトコル名称>::<ホスト名称>:<ポート番号>を指定します。

- 説明中に記述されているコマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」を参照してください。

付録 E.2　前提とするプロセスモデル

ユースケースが前提とするプロセスモデルを次の図に示します。

図 E-1　ユースケースが前提とするプロセスモデル

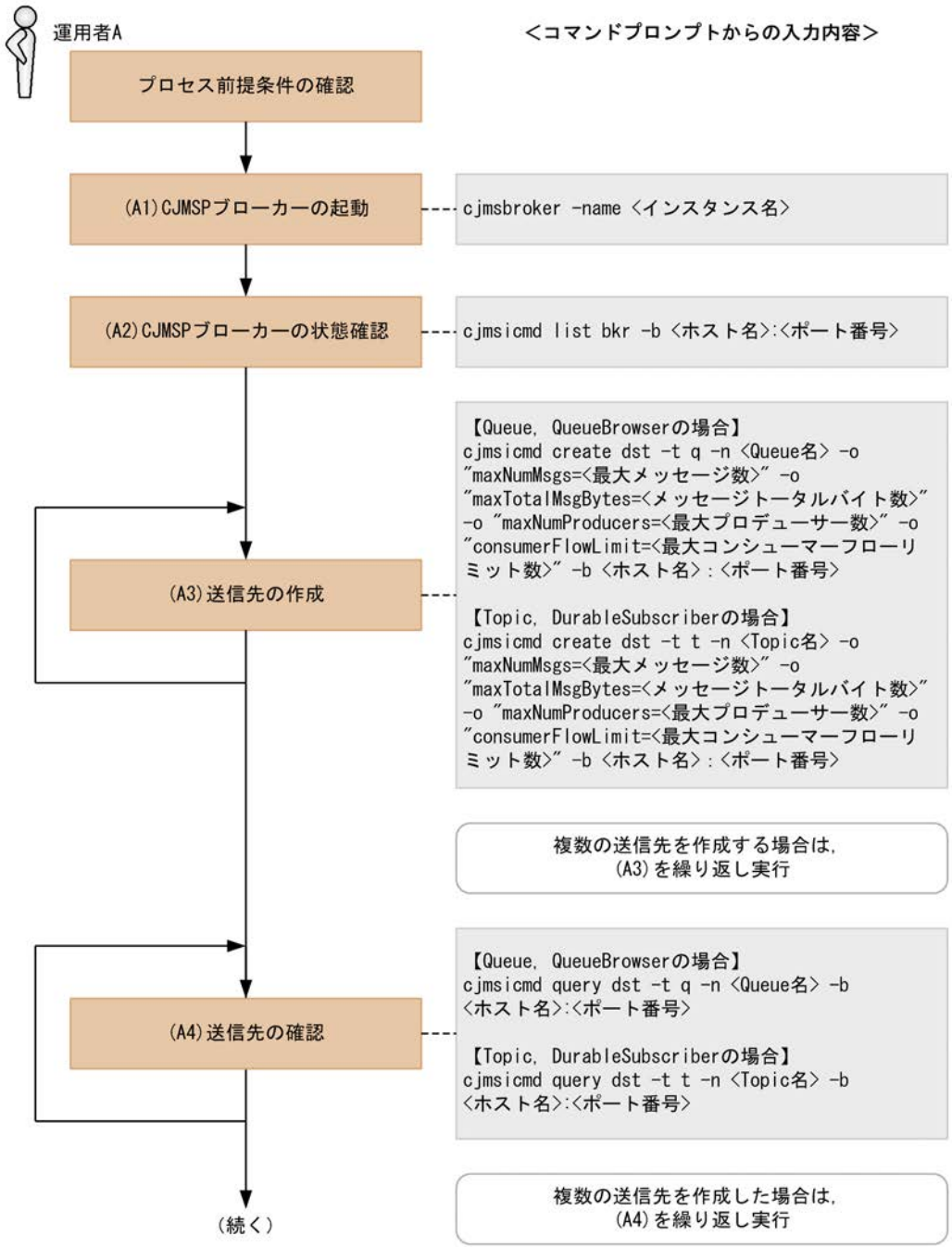


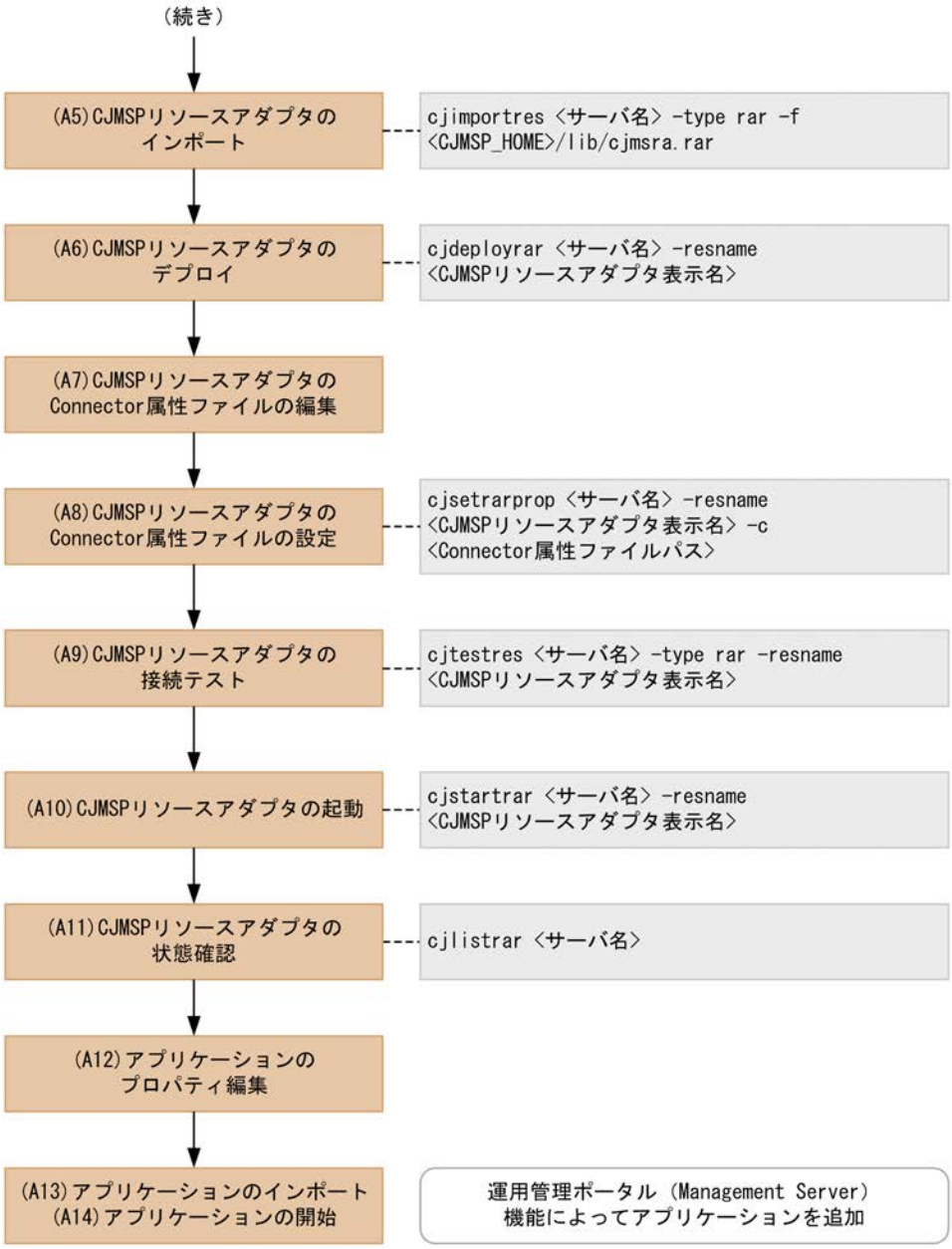
- 一つの CJMSP ブローカーに対して，複数の J2EE サーバからアクセスできます。
- 一つの J2EE サーバに定義できる CJMSP リソースアダプタは一つとします。

付録 E.3　CJMS プロバイダを使用する場合の環境構築

CJMS プロバイダを使用する場合の環境構築手順を次の図に示します。

図 E-2　CJMS プロバイダを使用する場合の環境構築手順





(1)　前提条件

- 運用管理ポータル（Management Server）によって，Application Server のインストールおよび J2EE サーバの構築が完了していること。
- CJMS プロバイダは，アプリケーションサーバのインストール完了後，次のディレクトリにインストールされていること。

  <COSMINEXUS_HOME>/CC/cjmsp

  手順実行前に，このディレクトリがあることを確認してください。

(2)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが未起動（インポートされていない状態）であること。

- CJMSP ブローカープロセスが未起動であること。
- アプリケーションが未起動（インポートされていない状態）であること。

(3)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

<インスタンス名>

省略した場合は，デフォルトで「cjmsbroker」を使用します。

複数インスタンスを使用したい場合には名称が重ならないように注意する必要があります。重なる場合には-name オプションを指定し，存在しない任意の名称を付与してください。

(A2)

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

CJMSP ブローカーの状態が「OPERATING」であることを必ず確認してください。

! 注意事項

CJMSP ブローカーが正常に起動されていない状態で CJMSP リソースアダプタを開始すると，例外が発生して CJMS プロバイダを使用することができません。

(A3)

<Queue 名>，<Topic 名>

Queue 名または Topic 名

<最大メッセージ数>

Queue または Topic に格納できる最大メッセージ数

<メッセージトータルバイト数>

Queue または Topic の最大トータルメッセージバイト数

<最大プロデューサー数>

送信先の最大プロデューサー数

<最大コンシューマーフローリミット数>

一つの処理単位でコンシューマーに配送可能な最大メッセージ数

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

(A4)

<Queue 名>，<Topic 名>

Queue 名または Topic 名

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

(A3)で作成した送信先が，指定したプロパティで正しく作られていることを確認してください。(A3)で指定した各引数と，表示内容の対応は次のとおりです。

- maxNumMsgs → Max Number of Messages
- maxNumProducers → Max Number of Producers
- maxTotalMsgBytes → Max Total Message Bytes
- consumerFlowLimit → Consumer Flow Limit

プロパティの表示例を次に示します。この例は，Queue を作成した場合の実行例です。

```
Destination Name                        TestQueue
Destination Type                        Queue
Destination State                       RUNNING
Created Administratively                true

Current Number of Messages
    Actual                              0
    Held in Transaction                 0
Current Message Bytes
    Actual                              0
    Held in Transaction                 0
Current Number of Producers             0
Current Number of Active Consumers      0

Max Number of Messages                  unlimited (-1)
Max Total Message Bytes                 unlimited (-1)
Max Number of Producers                 unlimited (-1)

Consumer Flow Limit                     100

KDAN34151-I Successfully queried the destination.
```

(A5)

<サーバ名>

Management Server で構築したサーバ名

<CJMSP_HOME>

<Application Server のインストールディレクトリ>/CC/cjmsp

(A6)

<サーバ名>

Management Server で構築したサーバ名

<CJMSP リソースアダプタ表示名>

CJMSP リソースアダプタの表示名

CJMS プロバイダの場合は，デフォルトで「Cosminexus_JMS_Provider_RA」が設定されています。

(A7)

Connector 属性ファイルは，テンプレートファイルをコピーして編集します。テンプレートファイルは，<CJMSP_HOME>/lib/templates/Cosminexus_JMS_Provider_RA_cfg.xml を使用します。

送信先を使用するため，CJMSP リソースアダプタが提供する管理対象オブジェクトについての定義を行います。

設定例を次に示します。番号を振っているタグを指定してください（番号は実際の定義には含まれません）。

```
<config-property>
  <description xml:lang="en"></description>
  <config-property-name>ConnectionURL</config-property-name>
```

```
     <config-property-type>java.lang.String</config-property-type>
(1)    <config-property-value>mq://localhost:7676/</config-property-value>
   </config-property>

    <config-property>
         <description xml:lang="en"></description>
         <config-property-name>clientId</config-property-name>
         <config-property-type>java.lang.String</config-property-type>
(2)     <config-property-value>Test</config-property-value>
    </config-property>

   <adminobject>
(3)  <adminobject-name>myQueue</adminobject-name>
(4)  <adminobject-interface>javax.jms.Queue</adminobject-interface>
(5)  <adminobject-class>com.cosminexus.jmsprovider.messaging.Queue</adminobject-class>
     <config-property>
       <description xml:lang="en"></description>
       <config-property-name>Name</config-property-name>
       <config-property-type>java.lang.String</config-property-type>
(6)    <config-property-value>TestQueue</config-property-value>
     </config-property>
     <config-property>
       <description xml:lang="en"></description>
       <config-property-name>Description</config-property-name>
       <config-property-type>java.lang.String</config-property-type>
       <config-property-value></config-property-value>
     </config-property>
   </adminobject>
```

各番号の指定内容を示します。

1. CJMSP ブローカーを起動するホスト名およびポート番号を指定します。
2. クライアント識別子を指定します。永続化サブスクライバーを使用する場合に必要になります。
3. <adminobject-name>に管理対象オブジェクト名を指定します。CJMSP リソースアダプタ内でユニークである必要があります。
4. <adminobject-interface>に管理対象オブジェクトのクラスが実装するインタフェースを指定します。

   Queue を使用する場合は，javax.jms.Queue を設定します。

   Topic を使用する場合は，javax.jms.Topic を設定します。
5. <adminobject-class>に管理対象オブジェクトのクラスを設定します。

   Queue を使用する場合は，com.cosminexus.jmsprovider.messaging.Queue を設定します。

   Topic を使用する場合は，com.cosminexus.jmsprovider.messaging.Topic を設定します。
6. (A3)で指定した送信先の名前を設定します。

   使用する送信先数分，<adminobject>タグを作成します。

! 注意事項

Message-driven Bean を使用する場合は，次の 2 点を設定してください。

- 次に示す(a)の値が(b)の値以上になるように設定してください。
- デプロイしているすべての Message-driven Bean の(a)の合計数の値が，次に示す(c)の値以下になるように設定してください

  アプリケーションプロパティファイルの Message-driven Bean のインスタンスプール最大数

  ```
  <pooled-instance>
  <minimum>1</minimum>
  (a) <maximum>2</maximum>
  </pooled-instance>
  ```

  アプリケーションプロパティファイルの Endpoint のインスタンスプール最大数

  ```
  <activation-config-property>
  ```

```
<activation-config-property-name>endpointPoolMaxSize</activation-config-property-name>
(b) <activation-config-property-value>1</activation-config-property-value>
</activation-config-property>
```

CJMSP リソースアダプタの Connector 属性ファイルの WorkManager のスレッドプール最大数

```
<property>
<property-name>MaxTPoolSize</property-name>
<property-type>int</property-type>
(c) <property-value>10</property-value>
<property-default-value>10</property-default-value>
</property>
```

(A8)

<サーバ名>

Management Server で構築したサーバ名

<CJMSP リソースアダプタ表示名>

CJMSP リソースアダプタの表示名

CJMS プロバイダの場合は，デフォルトで「Cosminexus_JMS_Provider_RA」が設定されています。

<Connector 属性ファイルパス>

属性ファイルの入力元パス

(A9)

<サーバ名>

Management Server で構築したサーバ名

<CJMSP リソースアダプタ表示名>

CJMSP リソースアダプタの表示名

CJMS プロバイダの場合は，デフォルトで「Cosminexus_JMS_Provider_RA」が設定されています。

CJMSP リソースアダプタの接続テストでは，次の 2 点を確認しています。

- 動作モードが 1.4 モードであること。
- CJMSP ブローカーとの接続を確立できること。

(A10)

<サーバ名>

Management Server で構築したサーバ名

<CJMSP リソースアダプタ表示名>

CJMSP リソースアダプタの表示名

CJMS プロバイダの場合は，デフォルトで「Cosminexus_JMS_Provider_RA」が設定されています。

(A11)

<サーバ名>

Management Server で構築したサーバ名

「running <CJMSP リソースアダプタ表示名>」と表示されていることを確認します。

(A12)

アプリケーションのプロパティに，CJMSP リソースアダプタのコネクションファクトリ，管理対象オブジェクトを関連づける設定をします。

設定例を次に示します。番号を振っているタグを編集してください（番号は実際の定義には含まれません）。

```
   <resource-ref>
(1)  <res-ref-name>jms/qcf</res-ref-name>
(2)  <res-type>javax.jms.QueueConnectionFactory</res-type>
(3)  <res-auth>Container</res-auth>
(4)  <res-sharing-scope>Unshareable</res-sharing-scope>
(5)  <linked-to>Cosminexus_JMS_Provider_RA!javax.jms.QueueConnectionFactory</linked-to>
   </resource-ref>
   <resource-env-ref>
(6)  <resource-env-ref-name>jms/TestQueue</resource-env-ref-name>
(7)  <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
        <linked-adminobject>
(8)  <resourceadapter-name>Cosminexus_JMS_Provider_RA</resourceadapter-name>
(9)  <adminobject-name>myQueue</adminobject-name>
        </linked-adminobject>
   </resource-env-ref>
```

各番号の指定内容を示します。

1. <res-ref-name>にリソース参照の名称を指定します。
2. <res-type>にリソース参照のタイプを指定します。コネクションファクトリの種類を指定します。
3. <res-auth>にリソースを使用するための認証をアプリケーション上で行うか，コンテナに任せるかを指定します。指定できる文字列を次に示します。

   Application

   Container
4. <res-sharing-scope>にリソース接続を共有するかどうかを指定します。指定できる文字列を次に示します。

   Shareable

   Unshareable
5. <linked-to>に対応する CJMSP リソースアダプタの表示名を指定します。次の文字列を指定します。

   <CJMSPリソースアダプタ表示名>!<コネクション定義識別子>
6. <resource-env-ref-name>にリソース環境変数参照の名称を指定します。
7. <resource-env-ref-type>に管理対象オブジェクトの型として，送信先のインタフェースの種類を指定します。
8. <resourceadapter-name>に CJMSP リソースアダプタの表示名を指定します。
9. <adminobject-name>に(A7)の(3)で設定した，管理対象オブジェクト名を指定します。

(A13)

特にありません。

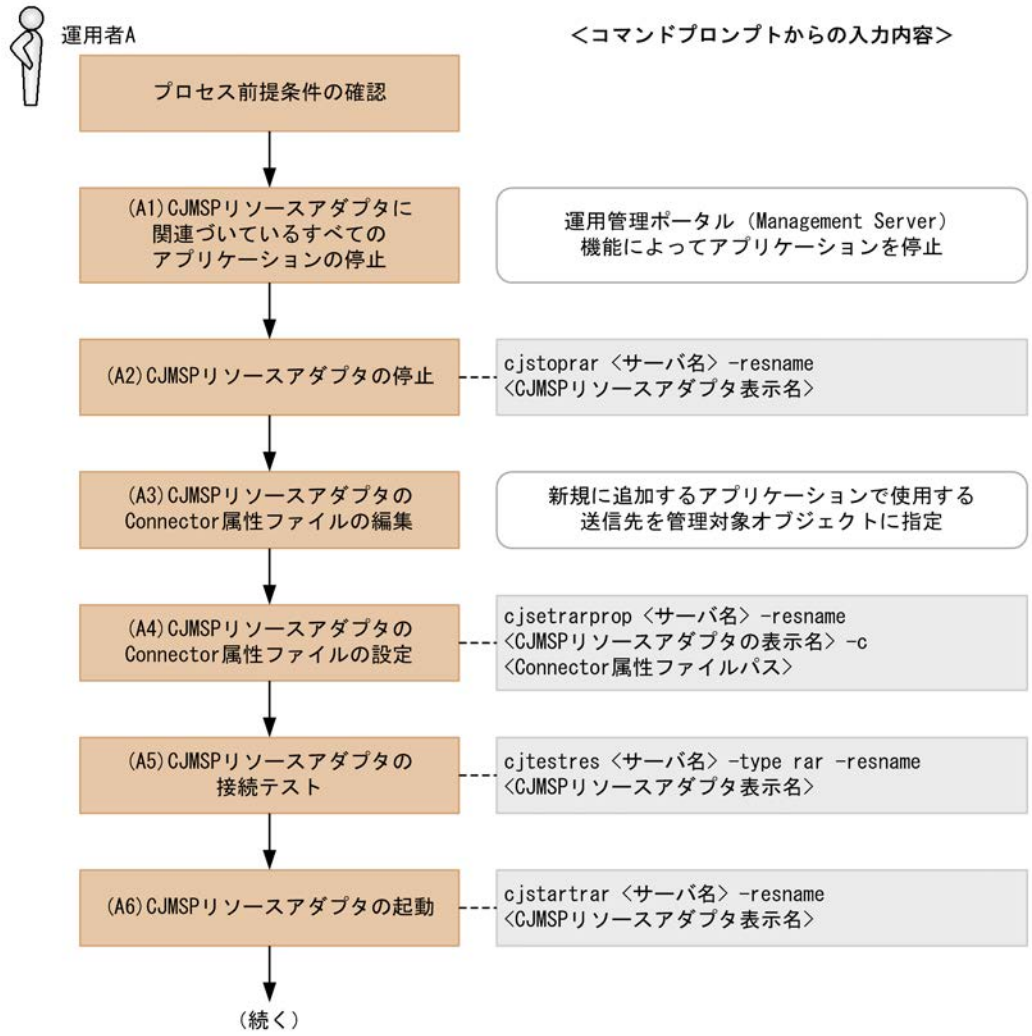
(A14)

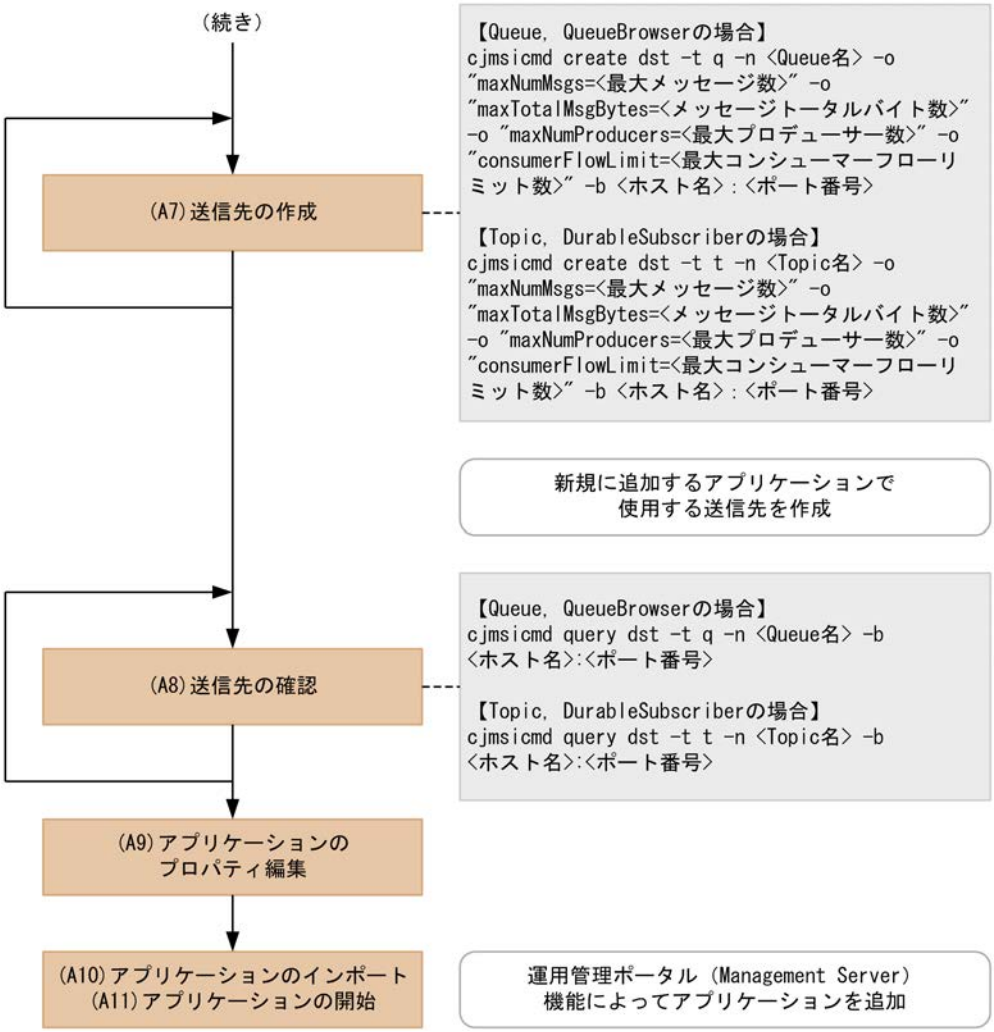
特にありません。

付録 E.4　CJMS プロバイダを使用するアプリケーションの追加

稼働中のシステムに対して新規にアプリケーションを追加する場合の手順を次の図に示します。

図 E-3　稼働中のシステムに対して新規にアプリケーションを追加する手順





(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動済みであること。
- アプリケーションが未起動（インポートされていない状態）であること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

運用管理ポータル（Management Server）機能でアプリケーションを停止します。

(A2)

<サーバ名>

Management Server で構築したサーバ名

<CJMSP リソースアダプタ表示名>

CJMSP リソースアダプタの表示名

CJMS プロバイダの場合は，デフォルトで「Cosminexus_JMS_Provider_RA」が設定されています。

(A3)

Connector 属性ファイルは，テンプレートファイルをコピーし，編集します。

テンプレートファイルは，<CJMSP_HOME>/lib/templates/Cosminexus_JMS_Provider_RA_cfg.xml を使用します。

送信先を使用するため，CJMSP リソースアダプタが提供する管理対象オブジェクトについての定義を行います。

設定例を次に示します。番号を振っているタグを指定してください（番号は実際の定義には含まれません）。

```
    <config-property>
      <description xml:lang="en"></description>
      <config-property-name>ConnectionURL</config-property-name>
      <config-property-type>java.lang.String</config-property-type>
(1)   <config-property-value>mq://localhost:7676/</config-property-value>
    </config-property>
     <config-property>
          <description xml:lang="en"></description>
          <config-property-name>clientId</config-property-name>
          <config-property-type>java.lang.String</config-property-type>
(2)   <config-property-value>Test</config-property-value>
    </config-property>

   <adminobject>
(3)    <adminobject-name>myQueue</adminobject-name>
(4)    <adminobject-interface>javax.jms.Queue</adminobject-interface>
(5)    <adminobject-class>com.cosminexus.jmsprovider.messaging.Queue</adminobject-class>
    <config-property>
      <description xml:lang="en"></description>
      <config-property-name>Name</config-property-name>
      <config-property-type>java.lang.String</config-property-type>
(6)    <config-property-value>TestQueue</config-property-value>
    </config-property>
    <config-property>
     <description xml:lang="en"></description>
     <config-property-name>Description</config-property-name>
     <config-property-type>java.lang.String</config-property-type>
    <config-property-value></config-property-value>
    </config-property>
   </adminobject>
```

各番号の指定内容を示します。

1. CJMSP ブローカーを起動するホスト名およびポート番号を指定します。
2. クライアント識別子を指定します。永続化サブスクライバーを使用する場合に必要になります。
3. <adminobject-name>に管理対象オブジェクト名を指定します。CJMSP リソースアダプタ内でユニークである必要があります。
4. <adminobject-interface>に管理対象オブジェクトのクラスが実装するインタフェースを指定します。

   Queue を使用する場合は，javax.jms.Queue を設定します。

   Topic を使用する場合は，javax.jms.Topic を設定します。
5. <adminobject-class>に管理対象オブジェクトのクラスを設定します。

   Queue を使用する場合は，com.cosminexus.jmsprovider.messaging.Queue を設定します。

   Topic を使用する場合は，com.cosminexus.jmsprovider.messaging.Topic を設定します。
6. 送信先の名前を設定します。

   使用する送信先数分，<adminobject>タグを作成します。

! 注意事項

Message-driven Bean を使用する場合は下記の 2 点を設定してください。

- 次に示す(a)の値が(b)の値以上になるように設定してください。
- デプロイしているすべての Message-driven Bean の(a)の合計数の値が，次に示す(c)の値以下になるように設定してください

アプリケーションプロパティファイルの Message-driven Bean のインスタンスプール最大数

```
<pooled-instance>
<minimum>1</minimum>
(a) <maximum>2</maximum>
</pooled-instance>
```

アプリケーションプロパティファイルの Endpoint のインスタンスプール最大数

```
<activation-config-property>
<activation-config-property-name>endpointPoolMaxSize</activation-config-property-name>
(b) <activation-config-property-value>1</activation-config-property-value>
</activation-config-property>
```

CJMSP リソースアダプタの Connector 属性ファイルの WorkManager のスレッドプール最大数

```
<property>
<property-name>MaxTPoolSize</property-name>
<property-type>int</property-type>
(c) <property-value>10</property-value>
<property-default-value>10</property-default-value>
</property>
```

(A4)

<サーバ名>

Management Server で構築したサーバ名

<CJMSP リソースアダプタ表示名>

CJMSP リソースアダプタの表示名

CJMS プロバイダの場合は，デフォルトで「Cosminexus_JMS_Provider_RA」が設定されています。

<Connector 属性ファイルパス>

属性ファイルの入力元パス

(A5)

<サーバ名>

Management Server で構築したサーバ名

<CJMSP リソースアダプタ表示名>

CJMSP リソースアダプタの表示名

CJMS プロバイダの場合は，デフォルトで「Cosminexus_JMS_Provider_RA」が設定されています。

CJMSP リソースアダプタの接続テストでは，次の 2 点を確認しています。

- 動作モードが 1.4 モードであること。
- CJMSP ブローカーとの接続を確立できること。

(A6)

<サーバ名>

Management Server で構築したサーバ名

<CJMSP リソースアダプタ表示名>

CJMSP リソースアダプタの表示名

CJMS プロバイダの場合は，デフォルトで「Cosminexus_JMS_Provider_RA」が設定されています。

(A7)

<Queue 名>，<Topic 名>

Queue 名または Topic 名

<最大メッセージ数>

Queue または Topic に格納できる最大メッセージ数

<メッセージトータルバイト数>

Queue または Topic の最大トータルメッセージバイト数

<最大プロデューサー数>

送信先の最大プロデューサー数

<最大コンシューマーフローリミット数>

一つの処理単位でコンシューマーに配送できる最大メッセージ数

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

<Queue 名> または<Topic 名>には，(A3)の(6)で設定した，送信先の名前を入力する。

(A8)

<Queue 名>，<Topic 名>

Queue 名または Topic 名

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

(A7)で作成した送信先が指定したプロパティで正しく作られていることを確認してください。(A7)で指定した各引数と，表示内容の対応は次のとおりです。

- maxNumMsgs → Max Number of Messages
- maxNumProducers → Max Number of Producers
- maxTotalMsgBytes → Max Total Message Bytes
- consumerFlowLimit → Consumer Flow Limit

指定したプロパティは次の表示で出力されます。この例は，Queue を作成した場合の実行例です。

```
Destination Name                TestQueue
Destination Type                Queue
Destination State               RUNNING
Created Administratively        true
```

```
Current Number of Messages
    Actual                          0
    Held in Transaction             0
Current Message Bytes
    Actual                          0
    Held in Transaction             0
Current Number of Producers         0
Current Number of Active Consumers  0

Max Number of Messages              unlimited (-1)
Max Total Message Bytes             unlimited (-1)
Max Number of Producers             unlimited (-1)

Consumer Flow Limit                 100

KDAN34151-I Successfully queried the destination.
```

(A9)

アプリケーションのプロパティに，CJMSP リソースアダプタのコネクションファクトリ，管理対象オブジェクトを関連づける設定をします。

設定例を次に示します。番号を振っているタグを指定してください（番号は実際の定義には含まれません）。

```
   <resource-ref>
(1)     <res-ref-name>jms/qcf</res-ref-name>
(2)     <res-type>javax.jms.QueueConnectionFactory</res-type>
(3)     <res-auth>Container</res-auth>
(4)     <res-sharing-scope>Unshareable</res-sharing-scope>
(5)     <linked-to>Cosminexus_JMS_Provider_RA!javax.jms.QueueConnectionFactory</linked-to>
   </resource-ref>
   <resource-env-ref>
(6)     <resource-env-ref-name>jms/TestQueue</resource-env-ref-name>
(7)     <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
        <linked-adminobject>
(8)       <resourceadapter-name>Cosminexus_JMS_Provider_RA</resourceadapter-name>
(9)       <adminobject-name>myQueue</adminobject-name>
        </linked-adminobject>
   </resource-env-ref>
```

各番号の指定内容を示します。

1. <res-ref-name>にリソース参照の名称を指定します。
2. <res-type>にリソース参照のタイプを指定します。コネクションファクトリの種類を指定します。
3. <res-auth>にリソースを使用するための認証元を，アプリケーション上で行うか，コンテナに任せるかを指定します。

   指定できる文字列を次に示します。

   Application

   Container
4. <res-sharing-scope>にリソース接続を共有するかどうかを指定します。

   指定できる文字列を次に示します。

   Shareable

   Unshareable
5. <linked-to>に対応する CJMSP リソースアダプタ表示名を指定します。

   次の文字列を指定します。

   <CJMSPリソースアダプタ表示名>!<コネクション定義識別子>
6. <resource-env-ref-name>にリソース環境変数参照の名称を指定します。
7. <resource-env-ref-type>に管理対象オブジェクトの型をします。送信先のインタフェースの種類を指定します。
8. <resourceadapter-name>に CJMSP リソースアダプタの表示名を指定します。

9. <adminobject-name>に(A3)の(3)で設定した，管理対象オブジェクト名を指定します。

(A10)

特にありません。

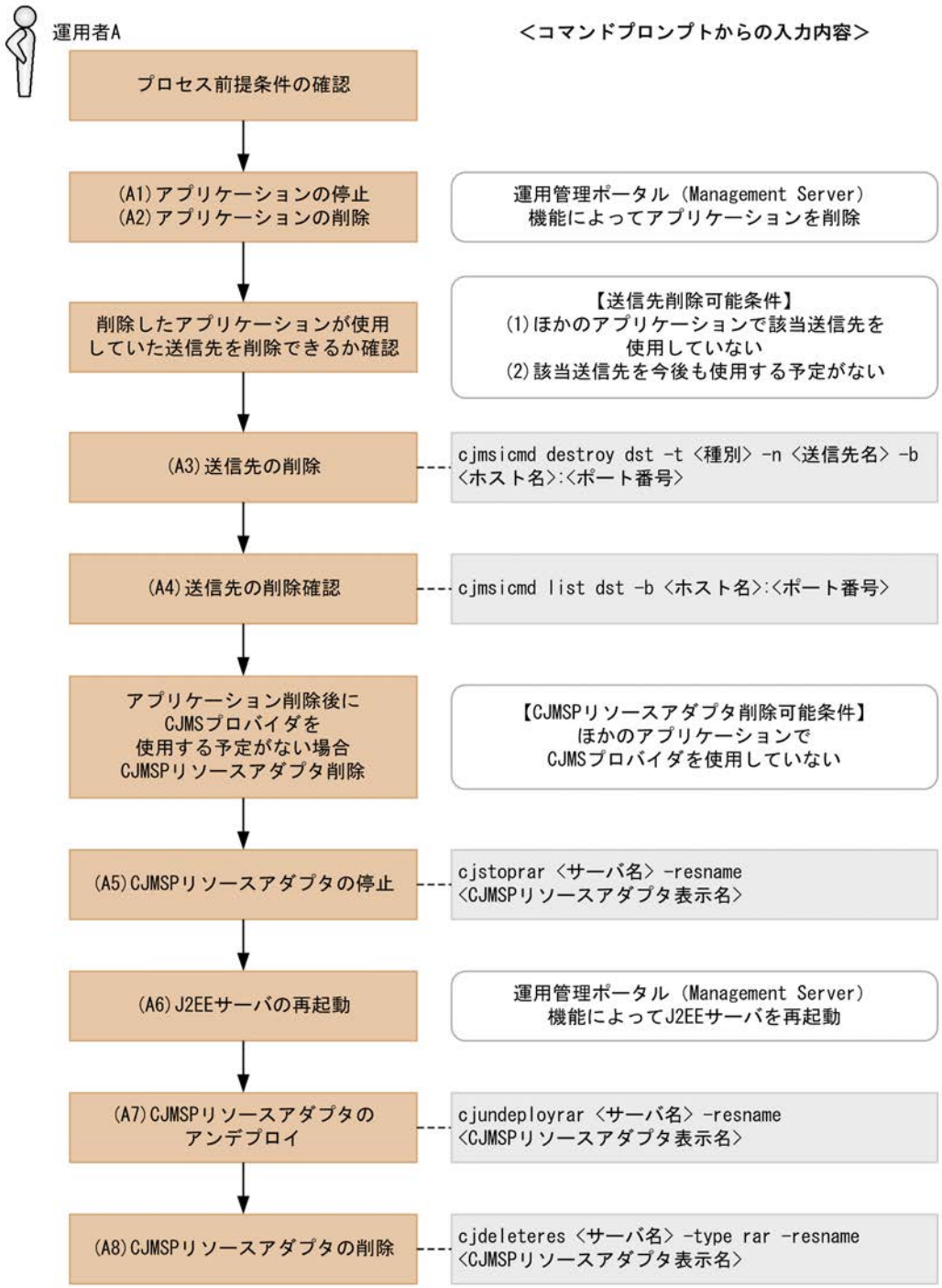
(A11)

特にありません。

付録 E.5　CJMS プロバイダを使用するアプリケーションの削除

稼働中のシステムからアプリケーションを削除する場合の手順を次の図に示します。

図 E-4　稼働中のシステムからアプリケーションを削除する場合の手順



(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動済みであること。
- アプリケーションが開始済みであること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

特にありません。

(A2)

運用管理ポータル（Management Server）機能によってアプリケーションを削除します。

(A3)

<種別>

q(Queue)または t(Topic)

<送信先名>

Queue 名または Topic 名

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

(A4)

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

(A3)で削除した送信先が正しく削除されていることを確認してください。

(A5)

<サーバ名>

Management Server で構築したサーバ名

<CJMSP リソースアダプタ表示名>

CJMSP リソースアダプタの表示名

CJMS プロバイダの場合は，デフォルトで「Cosminexus_JMS_Provider_RA」が設定されています。

(A6)

特にありません。

(A7)

<サーバ名>

Management Server で構築したサーバ名

<CJMSP リソースアダプタの表示名>

CJMSP リソースアダプタ表示名

CJMS プロバイダの場合は，デフォルトで「Cosminexus_JMS_Provider_RA」が設定されています。

(A8)

<サーバ名>

Management Server で構築したサーバ名

<CJMSP リソースアダプタ表示名>

CJMSP リソースアダプタの表示名

CJMS プロバイダの場合は，デフォルトで「Cosminexus_JMS_Provider_RA」が設定されています。

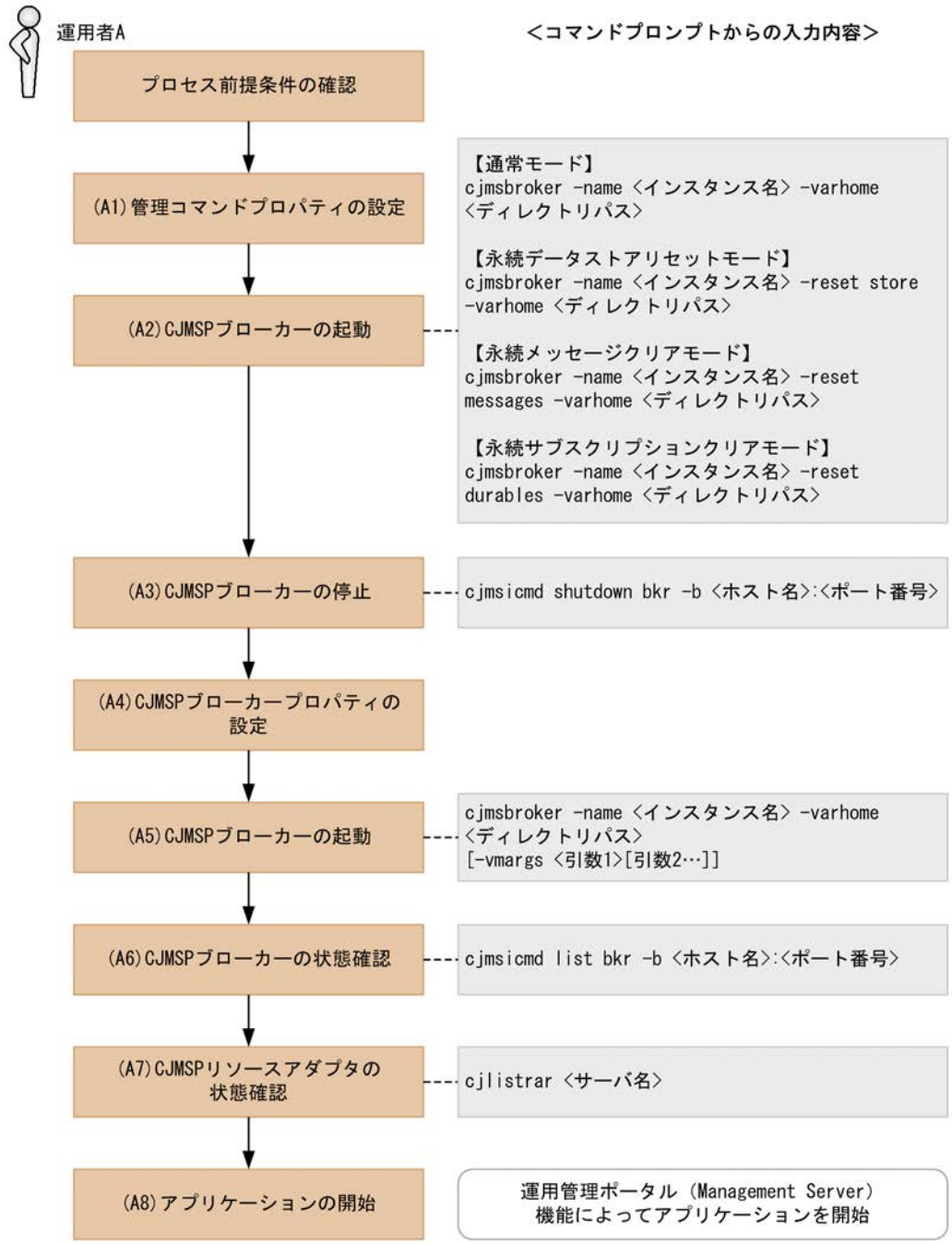
付録 E.6　CJMS プロバイダサービスの開始（初回起動時）

システム構築直後など，初回起動時に CJMS プロバイダのサービスを起動する手順について説明します。この手順で開始した場合は，「付録 E.11　CJMS プロバイダサービスの終了」で終了できます。

この手順に従って，CJMSP リソースアダプタの起動とアプリケーションの起動をした場合，J2EE サーバを再起動したときには CJMSP リソースアダプタおよびアプリケーションはどちらも起動状態となります（ただし，前提条件として CJMSP ブローカーが先に起動されていることが必須条件となります）。

CJMS プロバイダサービスの開始手順（初回起動時）を次の図に示します。

図 E-5　CJMS プロバイダサービスの開始手順（初回起動時）



(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが未起動であること。
- CJMSP ブローカープロセスが未起動であること。
- アプリケーションが未起動であること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

管理コマンドプロパティファイルは，<CJMSP_HOME>/var/admin/config/admin.properties に格納されています。

設定できるプロパティ名と説明を次に記述します。ファイルの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「7.2　admin.properties（管理コマンドプロパティファイル)」を参照してください。

admin.logger.MessageLogFile.trace.level

管理コマンドのメッセージログレベルを指定します。

admin.logger.MessageLogFile.filepath

管理コマンドのメッセージログを出力するファイルパスを指定します。

admin.logger.MessageLogFile.filenum

管理コマンドで作成される最大メッセージログファイル数を指定します。

admin.logger.MessageLogFile.filesize

管理コマンドのメッセージログファイルの最大サイズを指定します。

admin.logger.ExceptionLogFile.filepath

管理コマンドの例外ログを出力するファイルパスを指定します。

admin.logger.ExceptionLogFile.filenum

管理コマンドで作成される最大例外ログファイル数を指定します。

admin.logger.ExceptionLogFile.filesize

管理コマンドの例外ログファイルの最大サイズを指定します。

(A2)

<インスタンス名>

省略した場合は，デフォルトで「cjmsbroker」を使用します。

複数インスタンスを使用したい場合には名称が重ならないように注意する必要があります。

重なる場合には-name オプションを指定し存在しない任意の名称を付与してください。

<ディレクトリパス>

VAR_HOME を出力したいパスを指定します。

モードによるオプションの違い

- 通常モード

  オプション指定は行いません（状態をリセットしないで，継続状態で CJMS プロバイダのサービスを起動します)。

- 永続データストアリセットモード

  永続化メッセージ，永続化サブスクライバーが消去されます。

- 永続メッセージクリアモード

  すべての永続化メッセージが消去されます。

- 永続サブスクリプションクリアモード

  すべての永続化サブスクライバーが消去されます。

(A3)

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号を指定します。

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

(A4)

CJMSP ブローカープロパティファイルは，<VAR_HOME>/instances/<インスタンス名>/props/config.properties に格納されています。

このファイルは初めて CJMSP ブローカーを起動した時に作成されます。そのため，(A2)(A3)の手順が必要になります。

設定できるプロパティ名と説明を次に記述します。ファイルの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「7.4　config.properties (CJMSP ブローカー個別プロパティファイル)」を参照してください。

一時送信先を作成した場合，imq.autocreate.queue.consumerFlowLimit または imq.autocreate.topic.consumerFlowLimit の値が送信先のプロパティとして設定されます。

imq.hostname

すべての接続サービスのデフォルトのホスト名，または IP アドレスを指定します。

imq.portmapper.port

CJMSP ブローカーのポートマッパー用のポート番号を指定します。

imq.jms.tcp.port

jms サービスのポート番号を指定します。

imq.admin.tcp.port

admin サービスのポート番号を指定します。

imq.persist.file.sync.enabled

ディスク書き込み操作を同期にするか，または非同期にするかのフラグを指定します。

imq.autocreate.queue.consumerFlowLimit

一つの処理単位で Queue のコンシューマーに配送可能な最大メッセージ数を指定します。

imq.autocreate.topic.consumerFlowLimit

一つの処理単位で Topic のコンシューマーに配送可能な最大メッセージ数を指定します。

imq.metrics.interval

メトリクス情報をログとコンソールに出力する時間間隔を秒単位で指定します。

broker.logger.MessageLogFile.trace.level

CJMSP ブローカーのメッセージログレベルを指定します。

broker.logger.MessageLogFile.filenum

CJMSP ブローカーで作成される最大メッセージログファイル数を指定します。

broker.logger.MessageLogFile.filesize

CJMSP ブローカーのメッセージログファイルの最大サイズを指定します。

broker.logger.ExceptionLogFile.filenum

CJMSP ブローカーで作成される最大例外ログファイル数を指定します。

broker.logger.ExceptionLogFile.filesize

CJMSP ブローカーの例外ログファイルの最大サイズを指定します。

imq.instanceconfig.version

コンフィグプロパティのバージョンを指定します。

(A5)

<インスタンス名>

省略した場合は，デフォルトで「cjmsbroker」を使用します。

複数インスタンスを使用したい場合には名称が重ならないように注意する必要があります。

重なる場合には-name オプションを指定し存在しない任意の名称を付与してください。

<ディレクトリパス>

VAR_HOME を出力したいパスを指定します。

<引数>

-Xms（最大ヒープサイズ），-Xmx（最小ヒープサイズ），-XX:+HitachiVerboseGC（GC が発生したときの拡張 verbosegc 情報を出力するオプション）など JavaVM オプションを指定します。

(A6)

State が「OPERATING」であることを確認します。

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

(A7)

<サーバ名>

Management Server で構築したサーバ名

(A8)

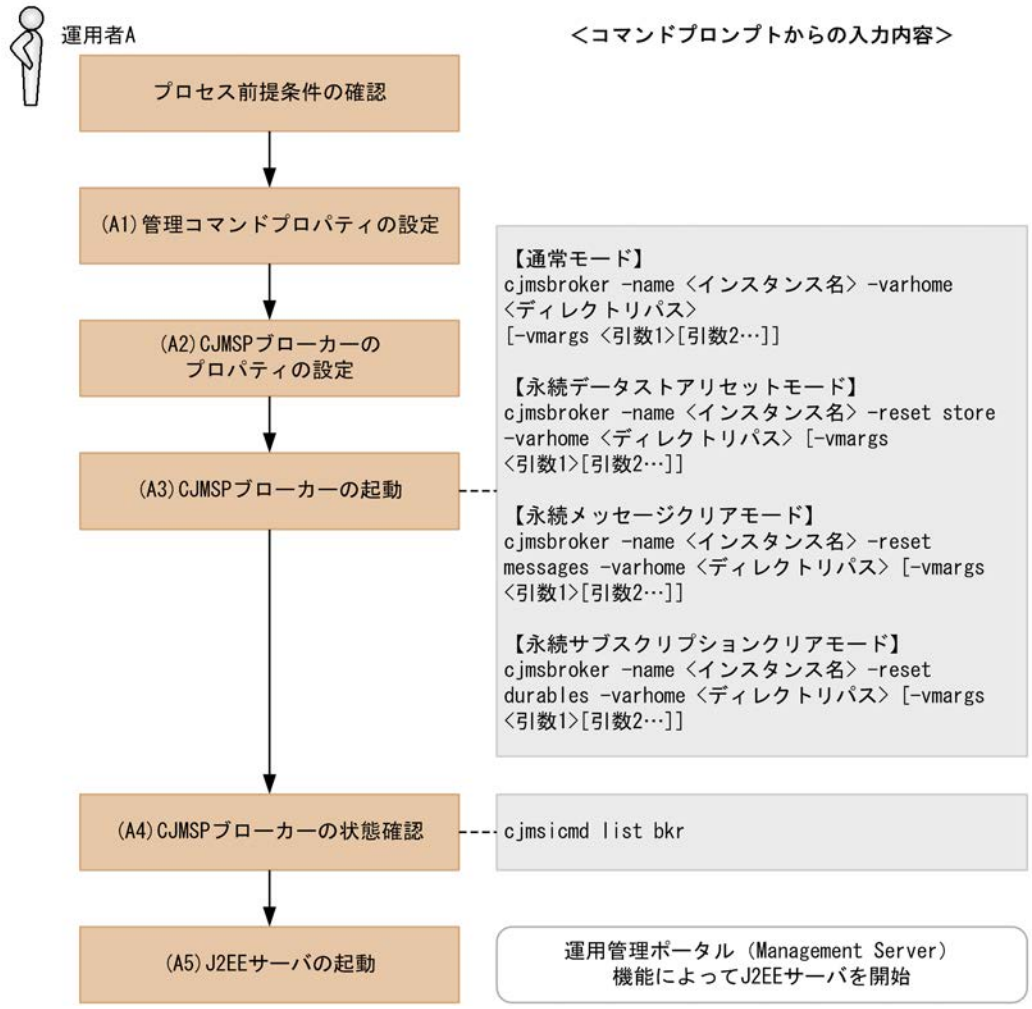
特にありません。

付録 E.7　CJMS プロバイダサービスの開始（稼働中システムの再起動時）

稼働中システムの再起動時に CJMS プロバイダのサービスを起動する手順について説明します。この手順で開始した場合は，「付録 E.11　CJMS プロバイダサービスの終了」で終了できます。

CJMS プロバイダサービスの開始手順（稼働中システムの再起動時）を次の図に示します。

図 E-6　CJMS プロバイダサービスの開始手順（稼働中システムの再起動時）



(1)　プロセス前提条件

- J2EE サーバプロセスが未起動であること。
- CJMSP リソースアダプタが未起動であること。
- CJMSP ブローカープロセスが未起動であること。
- アプリケーションが未起動であること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

管理コマンドプロパティファイルは，<CJMSP_HOME>/var/admin/config/admin.properties に格納されています。

設定できるプロパティ名と説明を次に記述します。ファイルの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「7.2　admin.properties（管理コマンドプロパティファイル)」を参照してください。

admin.logger.MessageLogFile.trace.level
　管理コマンドのメッセージログレベルを指定します。

admin.logger.MessageLogFile.filepath
　管理コマンドのメッセージログを出力するファイルパスを指定します。

admin.logger.MessageLogFile.filenum
　管理コマンドで作成される最大メッセージログファイル数を指定します。

admin.logger.MessageLogFile.filesize
　管理コマンドのメッセージログファイルの最大サイズを指定します。

admin.logger.ExceptionLogFile.filepath
　管理コマンドの例外ログを出力するファイルパスを指定します。

admin.logger.ExceptionLogFile.filenum
　管理コマンドで作成される最大例外ログファイル数を指定します。

admin.logger.ExceptionLogFile.filesize
　管理コマンドの例外ログファイルの最大サイズを指定します。

(A2)

CJMSP ブローカーのプロパティファイルは，<VAR_HOME>/instances/<インスタンス名>/props/config.properties に格納されています。

設定できるプロパティ名と説明を次に記述します。ファイルの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「7.4　config.properties (CJMSP ブローカー個別プロパティファイル)」を参照してください。

一時送信先を作成した場合，imq.autocreate.queue.consumerFlowLimit または imq.autocreate.topic.consumerFlowLimit の値が送信先のプロパティとして設定されます。

imq.hostname
　すべての接続サービスのデフォルトのホスト名，または IP アドレスを指定します。

imq.portmapper.port
　CJMSP ブローカーのポートマッパー用のポート番号を指定します。

imq.jms.tcp.port
　jms サービスのポート番号を指定します。

imq.admin.tcp.port
　admin サービスのポート番号を指定します。

imq.persist.file.sync.enabled
　ディスク書き込み操作を同期にするか，または非同期にするかのフラグを指定します。

imq.autocreate.queue.consumerFlowLimit
　一つの処理単位で Queue のコンシューマーに配送可能な最大メッセージ数を指定します。

imq.autocreate.topic.consumerFlowLimit
　一つの処理単位で Topic のコンシューマーに配送可能な最大メッセージ数を指定します。

imq.metrics.interval
　メトリクス情報をログとコンソールに出力する時間間隔を秒単位で指定します。

broker.logger.MessageLogFile.trace.level
　CJMSP ブローカーのメッセージログレベルを指定します。

broker.logger.MessageLogFile.filenum
　CJMSP ブローカーで作成される最大メッセージログファイル数を指定します。

broker.logger.MessageLogFile.filesize
　CJMSP ブローカーのメッセージログファイルの最大サイズを指定します。

broker.logger.ExceptionLogFile.filenum
　CJMSP ブローカーで作成される最大例外ログファイル数を指定します。

broker.logger.ExceptionLogFile.filesize
　CJMSP ブローカーの例外ログファイルの最大サイズを指定します。

imq.instanceconfig.version
　コンフィグプロパティのバージョンを指定します。

(A3)

<インスタンス名>
　省略した場合は，デフォルトで「cjmsbroker」を使用します。
　複数インスタンスを使用したい場合には名称が重ならないように注意する必要があります。
　重なる場合には-name オプションを指定し存在しない任意の名称を付与してください。

<ディレクトリパス>
　VAR_HOME を出力したいパスを指定します。

<引数>
　-Xms（最大ヒープサイズ），-Xmx（最小ヒープサイズ），-XX:+HitachiVerboseGC（GC が発生したときの拡張 verbosegc 情報を出力するオプション）など JavaVM のオプションを指定します。

モードによるオプションの違い

- 通常モード
  オプション指定は行いません（状態をリセットしないで，継続状態で CJMS プロバイダのサービスを起動します）。
- 永続データストアリセットモード
  永続化メッセージ，永続化サブスクライバーが消去されます。
- 永続メッセージクリアモード
  すべての永続化メッセージが消去されます。
- 永続サブスクリプションクリアモード
  すべての永続化サブスクライバーが消去されます。

(A4)

State が「OPERATING」であることを確認します。

(A5)

運用管理ポータル（Management Server）機能によって J2EE サーバを起動します。
CJMSP リソースアダプタとアプリケーションは J2EE サーバ起動時に開始されます。

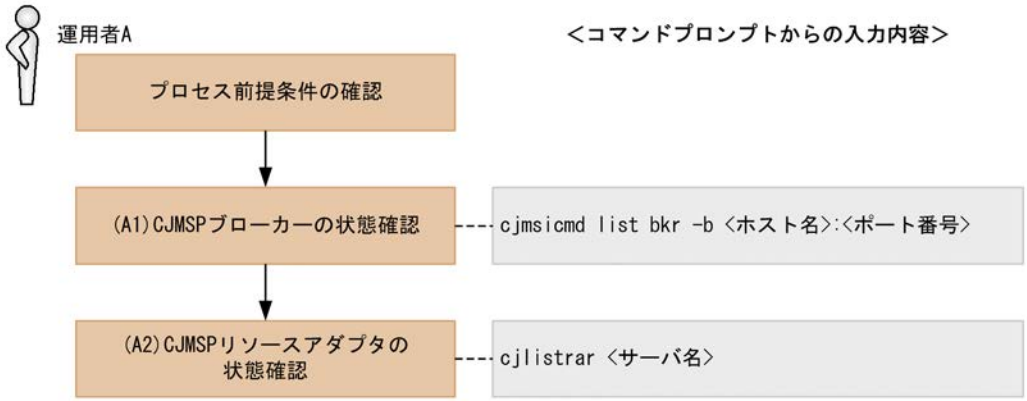
付録 E.8　CJMSP リソースアダプタと CJMSP ブローカーの状態確認

CJMS プロバイダのサービスが使用できる状態であるかどうかを確認する手順について説明します。

J2EE サーバとアプリケーションの状態については運用管理ポータル（Management Server）機能で確認してください。

CJMSP リソースアダプタと CJMSP ブローカーの状態確認手順を次の図に示します。

図 E-7　CJMSP リソースアダプタと CJMSP ブローカーの状態確認手順



(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動済みであること。
- アプリケーションが開始済みであること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

(A2)

<サーバ名>

Management Server で構築したサーバ名

CJMSP サービス起動状態

CJMSP ブローカーが「OPERATING」かつ CJMSP リソースアダプタが「running」の場合に CJMS プロバイダのサービスは開始されている状態です。

(A1)の結果例を次に示します。

```
----------------------
 Address       State
----------------------
localhost:7676   OPERATING
```

(A2)の結果例を次に示します。

```
running CJMS_Provider_RA
```

付録 E.9　メッセージ配信状況の確認と滞留時の対処（CJMSP ブローカーを一時停止する方法）

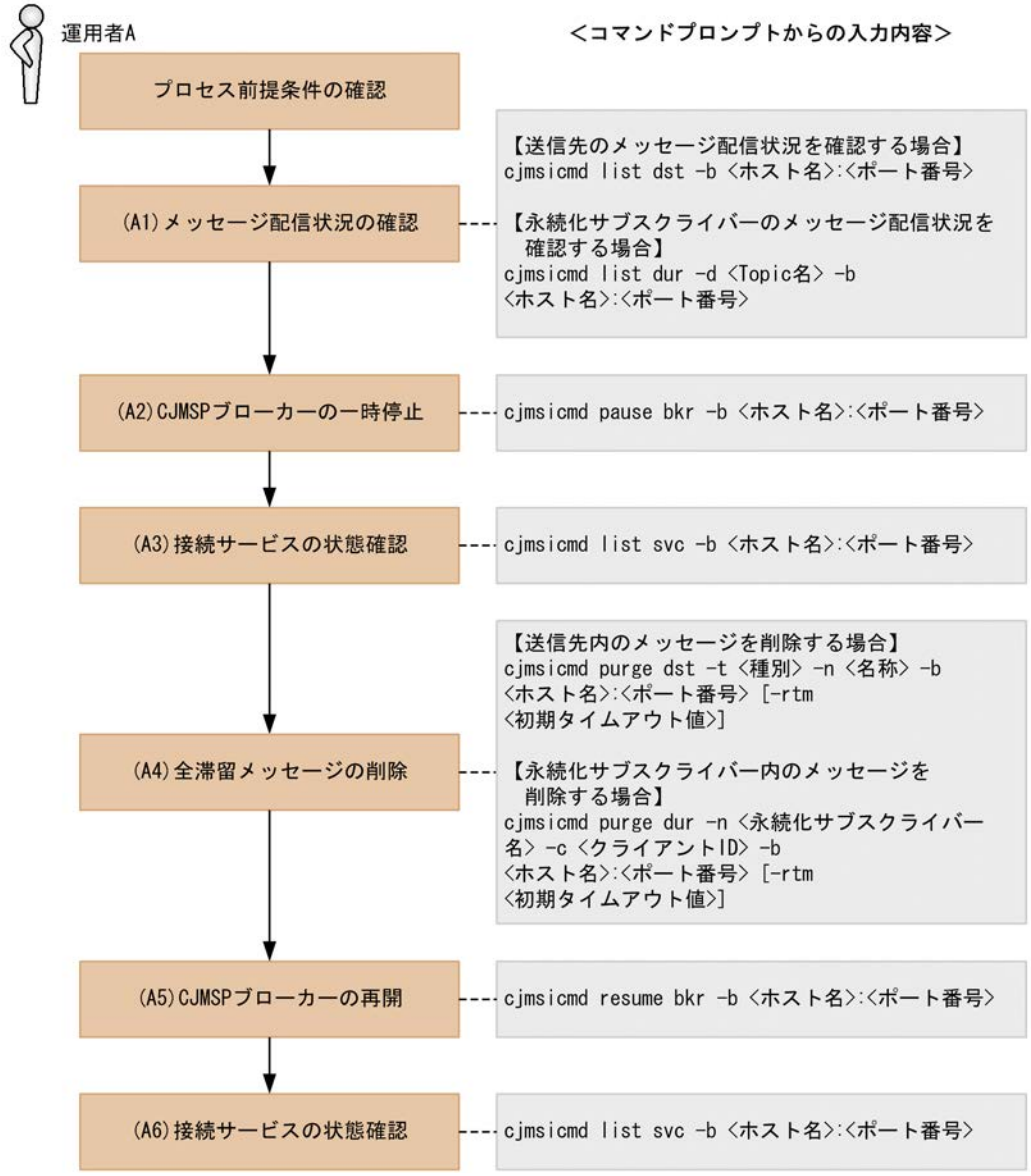
CJMS プロバイダのメッセージ配信状況を確認するとともに，メッセージが滞留していた場合は対処する手順について説明します。

この方法では，CJMSP ブローカーの一時停止後にアプリケーションでタイムアウトが発生することによって例外が発生する場合があります。

安全にメッセージを削除したい場合は，「付録 E.10　メッセージ配信状況の確認と滞留時の対処（アプリケーションを停止する方法）」を参照してください。

メッセージ配信状況の確認と滞留時の対処手順（CJMSP ブローカーを一時停止する方法）を次の図に示します。

図 E-8　メッセージ配信状況の確認と滞留時の対処手順（CJMSP ブローカーを一時停止する方法）



(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動済みであること。
- アプリケーションが開始済みであること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

送信先（または永続化サブスクライバー）名とメッセージ数を確認します。

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

<Topic 名>

確認しようとしている永続化サブスクライバーが存在する Topic 名

(A2)

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

(A3)

jms サービスが「PAUSED」の状態になっているか確認します。

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

出力例を次に示します。

```
-----------------------------------------------
Service Name    Port Number       Service State
-----------------------------------------------
admin           1248 (dynamic)         RUNNING
jms                0 (dynamic)         PAUSED

KDAN34113-I Successfully listed services.
```

(A4)

<種別>

q(Queue)または t(Topic)

<名称>

Queue 名または Topic 名

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

<初期タイムアウト値>

デフォルトの初期タイムアウト値は 10 秒に設定されています。デフォルト値で処理がタイムアウトしてしまう場合には値を調節してください。

<永続化サブスクライバー名>

(A1)で確認した永続化サブスクライバー名

<クライアント ID>

(A1)で確認したクライアント ID

(A5)

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

(A6)

jms サービスが「RUNNING」の状態になっているかを確認します。

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

出力例を次に示します。

```
-----------------------------------------------
Service Name    Port Number       Service State
-----------------------------------------------
admin           1248 (dynamic)         RUNNING
jms             1337 (dynamic)         RUNNING

KDAN34113-I Successfully listed services.
```

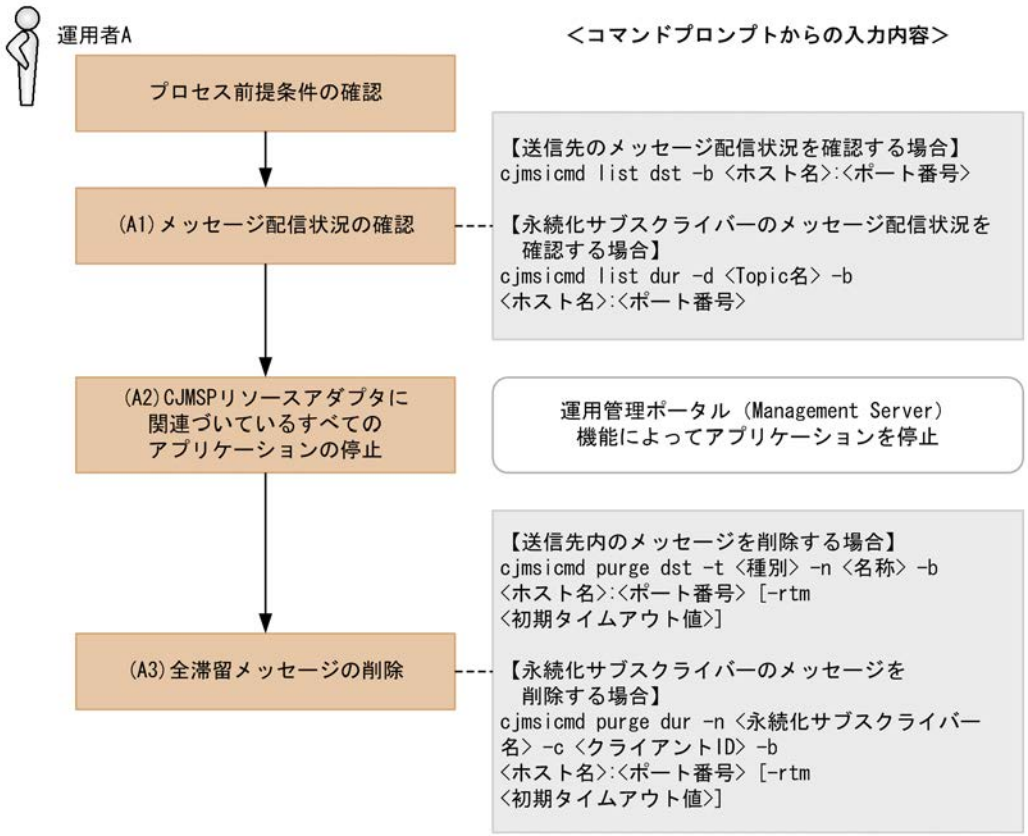
付録 E.10　メッセージ配信状況の確認と滞留時の対処（アプリケーションを停止する方法）

CJMS プロバイダのメッセージ配信状況を確認するとともに，メッセージが滞留していた場合に対処する手順について説明します。

アプリケーションを停止したくない場合は，「付録 E.9　メッセージ配信状況の確認と滞留時の対処 (CJMSP ブローカーを一時停止する方法)」を参照してください。

メッセージ配信状況の確認と滞留時の対処手順（アプリケーションを停止する方法）を次の図に示します。

図 E-9　メッセージ配信状況の確認と滞留時の対処手順（アプリケーションを停止する方法）



(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動済みであること。
- アプリケーションが開始済みであること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

送信先（または永続化サブスクライバー）名とメッセージ数を確認します。

＜ホスト名＞:＜ポート番号＞

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

＜Topic 名＞

確認しようとしている永続化サブスクライバーが存在する Topic 名

(A2)

特にありません。

(A3)

<種別>

q(Queue)または t(Topic)

<名称>

Queue 名または Topic 名

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

<初期タイムアウト値>

デフォルトの初期タイムアウト値は 10 秒に設定されています。デフォルト値で処理がタイムアウトしてしまう場合には値を調節してください。

<永続化サブスクライバー名>

(A1)で確認した永続化サブスクライバー名

<クライアント ID>

(A1)で確認したクライアント ID

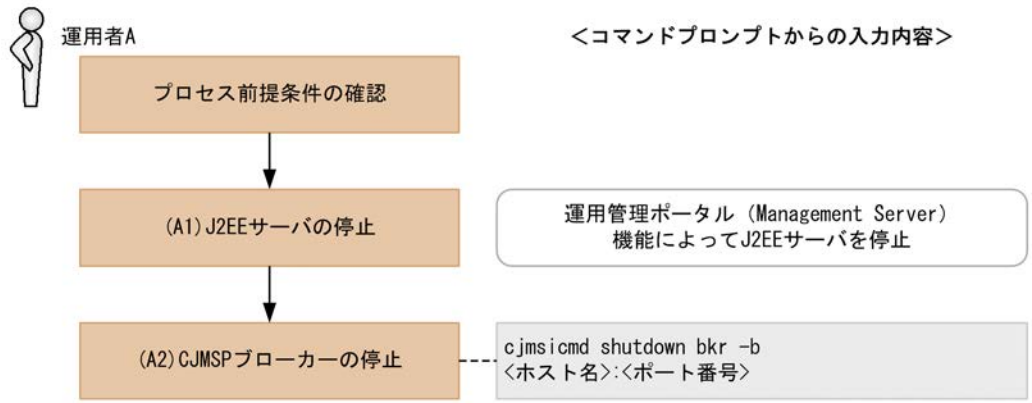
付録 E.11　CJMS プロバイダサービスの終了

CJMS プロバイダのサービスの終了の手順について説明します。

この手順で停止した場合は，「付録 E.7　CJMS プロバイダサービスの開始（稼働中システムの再起動時）」の手順によって再起動できます。

CJMS プロバイダのサービスの終了手順を次の図に示します。

図 E-10　CJMS プロバイダのサービスの終了手順



(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動済みであること。
- アプリケーションが開始済みであること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

特にありません。

(A2)

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

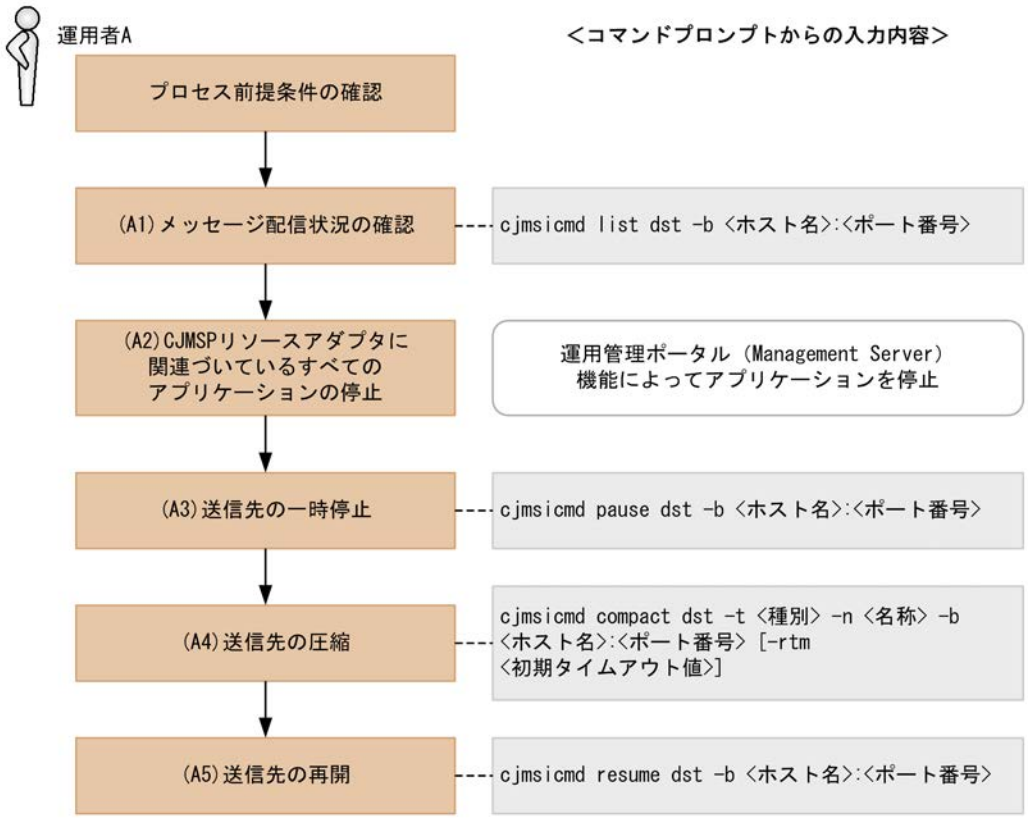
CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

付録 E.12　送信先の圧縮

CJMS プロバイダで使用する送信先の圧縮手順について説明します。

送信先の圧縮手順を次の図に示します。

図 E-11　送信先の圧縮手順



(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動済みであること。

- アプリケーションが開始済みであること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

送信先名とメッセージ数を確認します。

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

(A2)

特にありません。

(A3)

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

(A4)

<種別>

q(Queue)または t(Topic)

<名称>

Queue 名または Topic 名

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

<初期タイムアウト値>

デフォルトの初期タイムアウト値は 10 秒に設定されています。デフォルト値で処理がタイムアウトしてしまう場合には値を調節してください。

(A5)

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

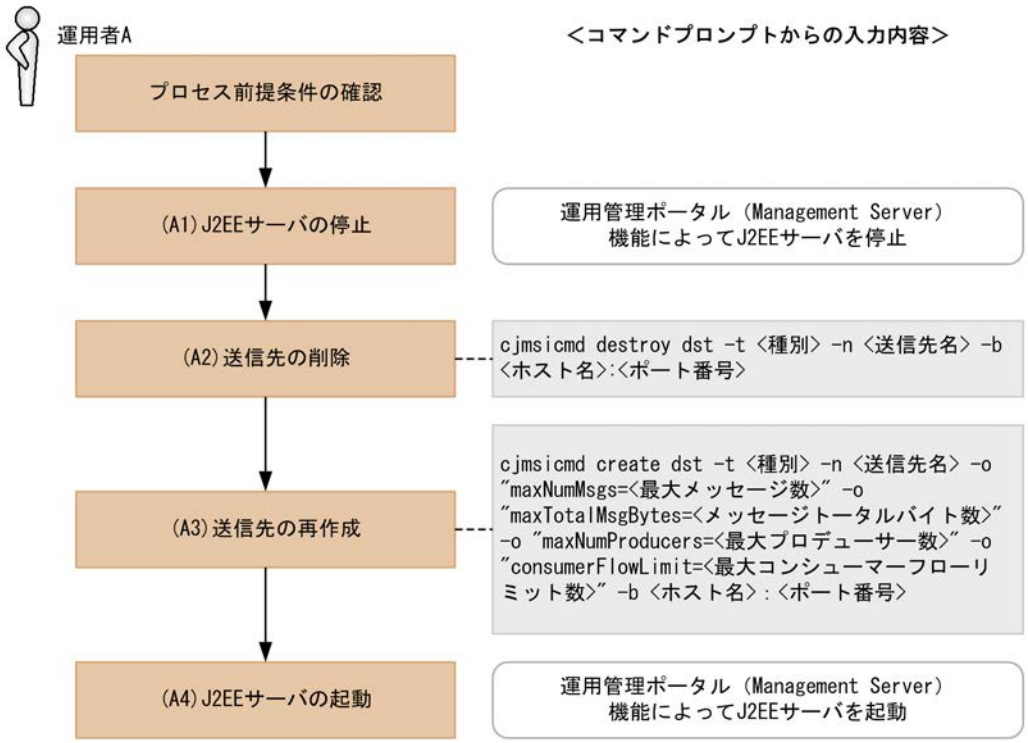
CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

付録 E.13　送信先サイズの変更

CJMS プロバイダで使用する送信先のサイズ変更手順について説明します。

送信先サイズの変更手順を次の図に示します。

図 E-12　送信先サイズの変更手順



(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動済みであること。
- アプリケーションが開始済みであること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

特にありません。

(A2)

＜種別＞

q(Queue)または t(Topic)

＜送信先名＞

削除する Queue 名または Topic 名

＜ホスト名＞:＜ポート番号＞

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

(A3)

<種別>

q(Queue)または t(Topic)

<送信先名>

再作成する Queue 名または Topic 名

<最大メッセージ数>

Queue または Topic に格納される最大メッセージ数

<メッセージトータルバイト数>

Queue または Topic に格納されるメッセージのトータルバイト数

<最大プロデューサー数>

送信先の最大プロデューサー数

<最大コンシューマーフローリミット数>

一つの処理単位でコンシューマーに配送可能な最大メッセージ数

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

実行結果の例を示します。

例：メッセージ最大数を 100000，メッセージのトータルバイト数を 12 メガバイト，最大プロデューサー数を 1000，最大コンシューマーフローリミット数を 1000 で Queue を作成した場合の例

```
C:¥>cjmsicmd create dst -t q -n Queue1 -o "maxNumMsgs=100000" -o "maxTotalMsgBytes=12m" -o
"maxNumProducers=1000" -o "consumerFlowLimit=1000" -b localhost:7676
KDAN34140-I Successfully created the destination.
```

(A4)

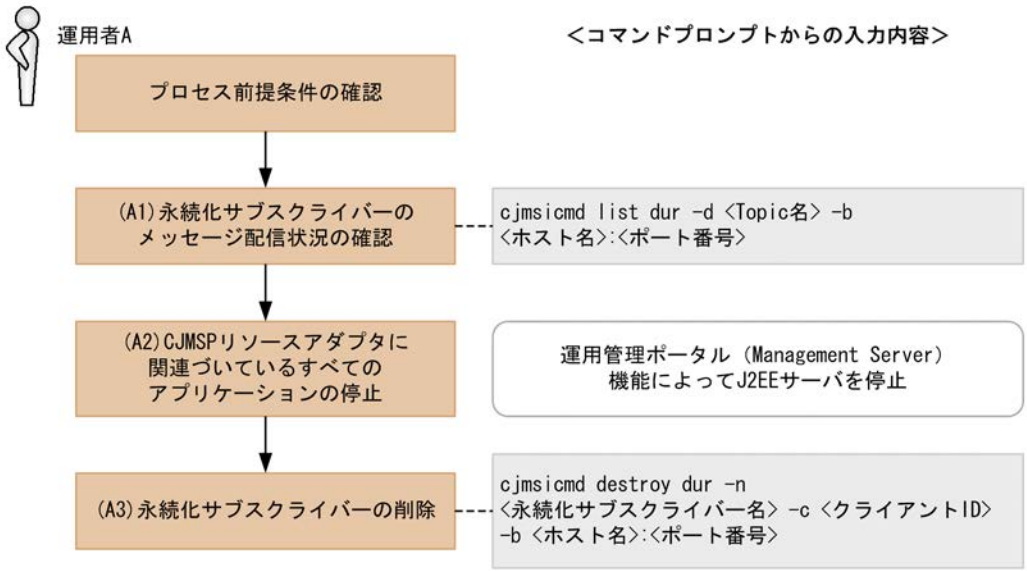
特にありません。

付録 E.14　永続化サブスクライバーの削除

CJMS プロバイダで使用する，永続化サブスクライバーの削除手順について説明します。

永続化サブスクライバーの削除手順を次の図に示します。

図 E-13　永続化サブスクライバーの削除手順



(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動済みであること。
- アプリケーションが開始済みであること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

<Topic 名>

確認しようとしている永続化サブスクライバーが存在する Topic 名

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

表示例を次に示します。

```
------------------------------------------------------------------------
Durable Sub. Name   Client ID   Number of Messages   Durable Sub. State
------------------------------------------------------------------------
DurableSUB1         clientId    9998                 INACTIVE
KDAN34323-I Successfully listed durable subscriptions.
```

(A2)

特にありません。

(A3)

<永続化サブスクライバー名>

(A1)で確認した永続化サブスクライバー名

<クライアント ID>

(A1)で確認したクライアント ID

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

付録 E.15　CJMSP ブローカーの状態監視

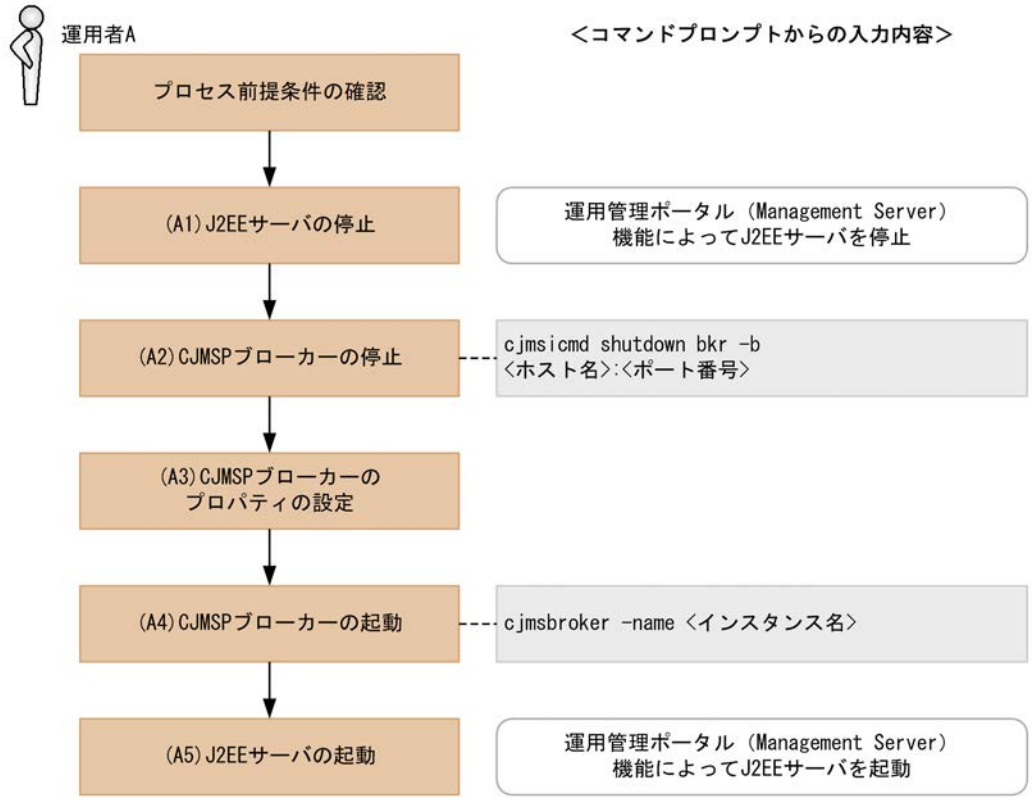
デバッグ，障害解析などによって，CJMSP ブローカーを監視モードで起動する場合の手順について説明します。

この手順で CJMSP ブローカーを再起動することによって，CJMSP ブローカーを実行したコマンドプロンプト上にメトリクス情報が出力されます。

この監視状態を終了する場合には，「付録 E.7　CJMS プロバイダサービスの開始（稼働中システムの再起動時)」を参照して，通常モードで CJMSP ブローカーを再起動してください。

CJMSP ブローカーの状態監視手順を次の図に示します。

図 E-14　CJMSP ブローカーの状態監視手順



(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動済みであること。

- アプリケーションが開始済みであること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

特にありません。

(A2)

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

(A3)

CJMSP ブローカーのプロパティファイルは，<VAR_HOME>/instances/<インスタンス名>/props/config.properties に格納されています。

状態監視モードで CJMSP ブローカーを起動する場合は，次のプロパティを設定してください。

なお，詳細はマニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「7.4 config.properties（CJMSP ブローカー個別プロパティファイル)」を参照してください。

imq.metrics.interval

メトリクス情報をログとコンソールに出力する時間間隔を秒単位で指定します。

(A4)

<インスタンス名>

省略した場合は，デフォルトで「cjmsbroker」を使用します。

複数インスタンスを使用したい場合には，名称が重ならないように注意する必要があります。

重なる場合には，-name オプションを指定し存在しない任意の名称を付与してください。

状態監視モードの表示例を次に示します。

```
  Connections: 0    JVM Heap: 2932736 bytes (731352 free) Threads: 0 (14-1010)
       In: 0 msgs (0 bytes)  0 pkts (0 bytes)
      Out: 0 msgs (0 bytes)  0 pkts (0 bytes)
  Rate In: 0 msgs/sec (0 bytes/sec)  0 pkts/sec (0 bytes/sec)
 Rate Out: 0 msgs/sec (0 bytes/sec)  0 pkts/sec (0 bytes/sec)
```

(A5)

特にありません。

付録 E.16　CJMSP ブローカーの詳細情報確認

CJMSP ブローカーの最新の詳細情報を確認する手順について，次の図に示します。

図 E-15　CJMSP ブローカーの詳細情報確認手順

(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動済みであること。
- アプリケーションが開始済みであること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

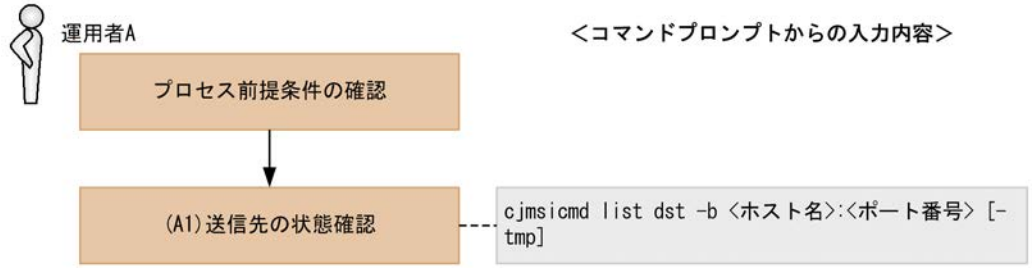
表示例を次に示します。

```
Instance Name                                                    cjmsbroker
Primary Port                                                       7676
Broker is Embedded                                               false
Instance Configuration/Data Root Directory                D:¥Hitachi¥CC¥cjmsp¥var
Current Number of Messages in System                      843
Current Total Message Bytes in System                      177030
Current Number of Messages in Dead Message Queue    0
Current Total Message Bytes in Dead Message Queue     0
KDAN34295-I Successfully queried the broker.
```

付録 E.17　送信先の状態確認

送信先の最新状態を確認する手順について，次の図に示します。

図 E-16　送信先の状態確認手順



(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動済みであること。
- アプリケーションが開始済みであること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

-tmp オプションを使用すると，一時送信先の状態を確認できます。

一時送信先の状態の確認ができるのは，アプリケーションを開始して一時送信先が有効になっている期間だけです。

-tmp オプション使用時の表示例を次に示します。

```
------------------------------------------------------------------------
   Name     Type    State   Producers  Consumers           Msgs
                                         Total    Count  UnAck  Avg Size
------------------------------------------------------------------------
Queue1         Queue  RUNNING  0          0          0      0      0.0
mq.sys.dmq  Queue  RUNNING  0          0          0      0      0.0
temporary_destination://queue/10.209.10.192/2139/1  Queue (temporary) 0          1          0
0      0.0
KDAN34110-I Successfully listed destinations.
```

付録 E.18　永続化サブスクライバーの状態確認

永続化サブスクライバーの最新状態を確認する手順について，次の図に示します。

図 E-17　永続化サブスクライバーの状態確認手順

(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動済みであること。
- アプリケーションが開始済みであること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

<Topic 名>

確認しようとしている永続化サブスクライバーが存在する Topic 名

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

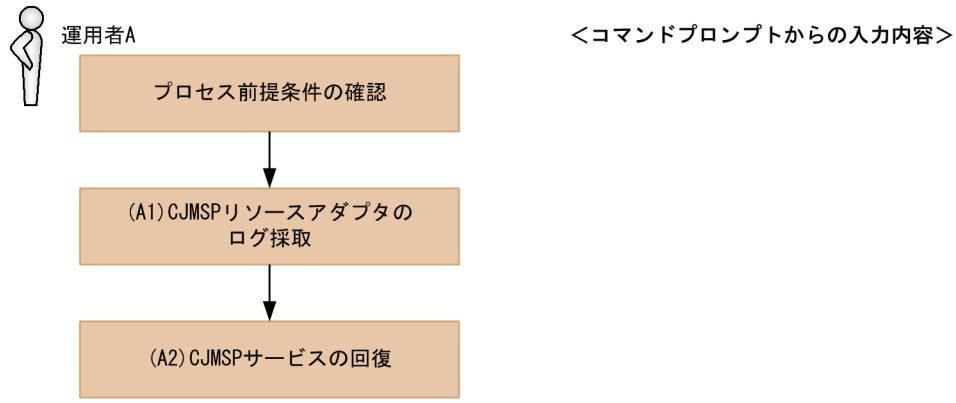
表示例を次に示します。

```
-------------------------------------------------------------------
Durable Sub. Name    Client ID    Number of Messages    Durable Sub. State
-------------------------------------------------------------------
DurableSUB1          clientId     9998                  INACTIVE
KDAN34323-I Successfully listed durable subscriptions.
```

付録 E.19　CJMSP リソースアダプタに問題が発生した場合の解析

CJMSP リソースアダプタで問題が発生した状態からの解析手順について，次の図に示します。

図 E-18　CJMSP リソースアダプタに問題が発生した場合の解析手順



(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動済みであること。
- アプリケーションが開始済みであること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

<Application Server のインストールディレクトリ>/CC/server/public/ejb/<サーバ名>/logs/cjms/<コネクタ名>下のログを確認し，問題の解析を行います。

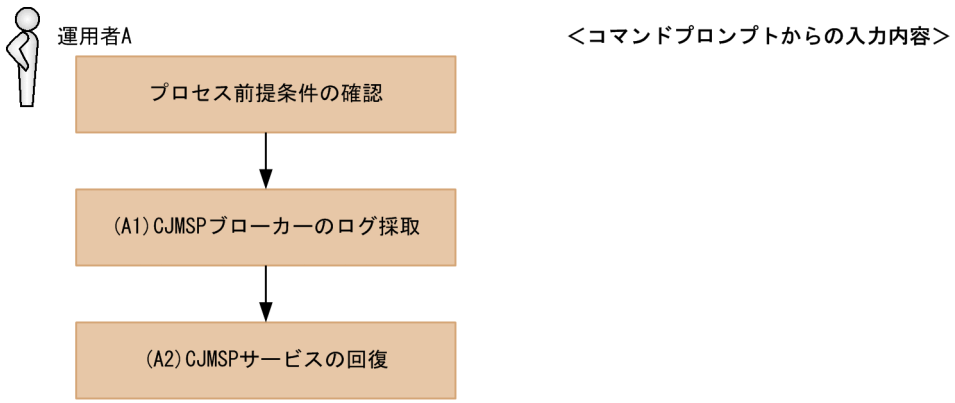
(A2)

「付録 E.22　CJMSP リソースアダプタに問題が発生した場合の回復」の手順に従って，CJMS プロバイダのサービスを回復してください。

付録 E.20　CJMSP ブローカーが障害によって停止したときの解析

CJMSP ブローカーが障害によって停止した状態からの解析手順について，次の図に示します。

図 E-19　CJMSP ブローカーが障害によって停止した状態からの解析手順



(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動済みであること。
- アプリケーションが開始済みであること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

<Application Server のインストールディレクトリ>/CC/cjmsp/var/instances/<インスタンス名>/log 下のログを確認し，問題の解析を行います。

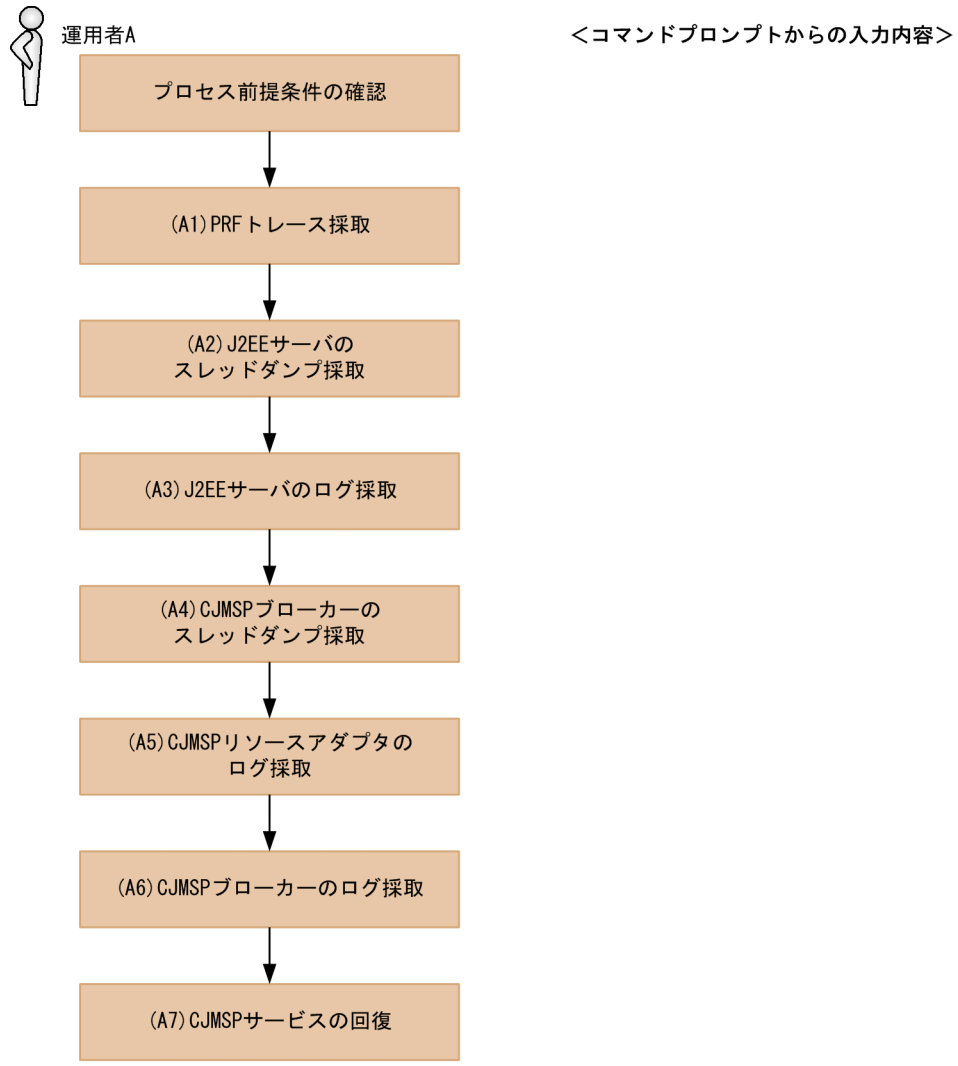
(A2)

「付録 E.23　CJMSP ブローカーが障害によって停止したときの回復」の手順に従って，CJMS プロバイダのサービスを回復します。

付録 E.21　CJMS プロバイダサービス無応答時の解析

CJMS プロバイダのサービス無応答時の解析手順を次の図に示します。

図 E-20　CJMS プロバイダサービス無応答時の解析手順



(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動済みであること。
- アプリケーションが開始済みであること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

PRF トレースを採取します。

(A2)

J2EE サーバのスレッドダンプを採取します。

(A3)

<Application Server のインストールディレクトリ>/CC/server/public/ejb/<サーバ名>/logs 下のログを確認し，障害を切り分けます。

(A4)

CJMSP ブローカーのスレッドダンプを採取します。

Windows の場合

CJMSP ブローカーのスレッドダンプの取得方法は二つあります。

- JavaVM のコマンド jheapprof を使用します。
  このコマンドの使用方法については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「jheapprof（クラス別統計情報付き拡張スレッドダンプの出力)」を参照してください。
- CJMSP ブローカーを起動しているコマンドプロンプト上で［Ctrl］＋［Break］キーを入力します。
  この場合，スレッドダンプを取得したあとに CJMSP ブローカーを停止すると，「バッチジョブを終了しますか(Y/N)?」と表示されます。その際，「N」を指定してください。

UNIX の場合

kill コマンドを使用します。

実行例

```
kill -3 <プロセスID>
```

(A5)

<Application Server のインストールディレクトリ>/CC/server/public/ejb/<サーバ名>/logs/cjms/<コネクタ名>下のログを確認して，障害を切り分けます。

(A6)

<VAR_HOME>/instances/<インスタンス名>/log 下のログを確認して，障害を切り分けます。

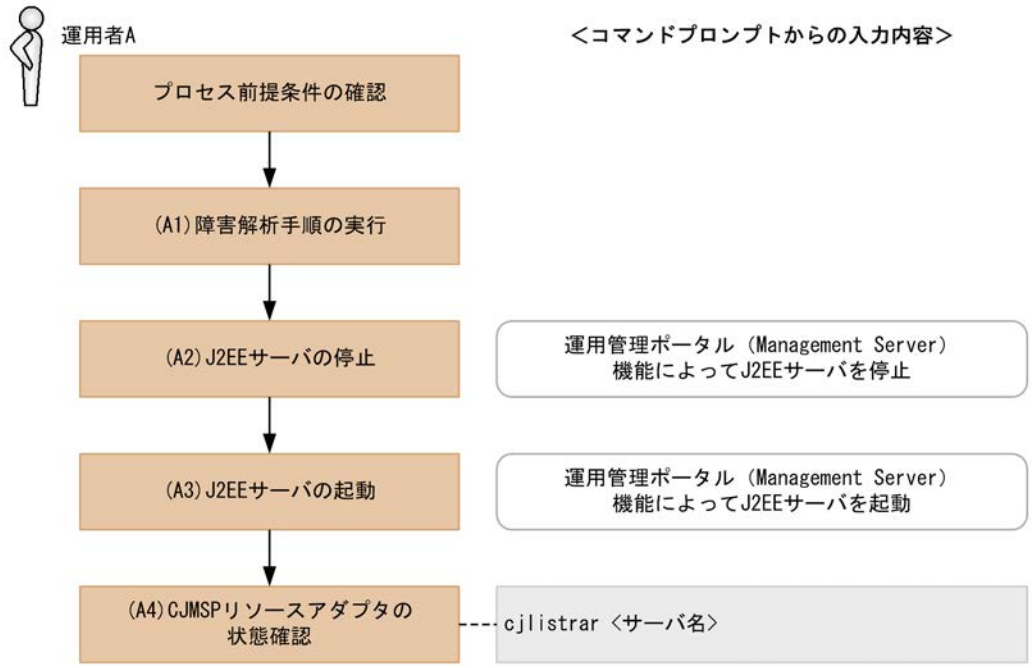
(A7)

「付録 E.24　CJMS プロバイダサービス無応答時の回復」の手順に従って，CJMS プロバイダのサービスを回復します。

付録 E.22　CJMSP リソースアダプタに問題が発生した場合の回復

CJMSP リソースアダプタに問題が発生した場合の回復手順を次の図に示します。

図 E-21　CJMSP リソースアダプタに問題が発生した場合の回復手順



(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動中であること。
- アプリケーションが未起動であること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

「付録 E.19　CJMSP リソースアダプタに問題が発生した場合の解析」を参照して，障害を解析します。

(A2)

特にありません。

(A3)

特にありません。

(A4)

<サーバ名>

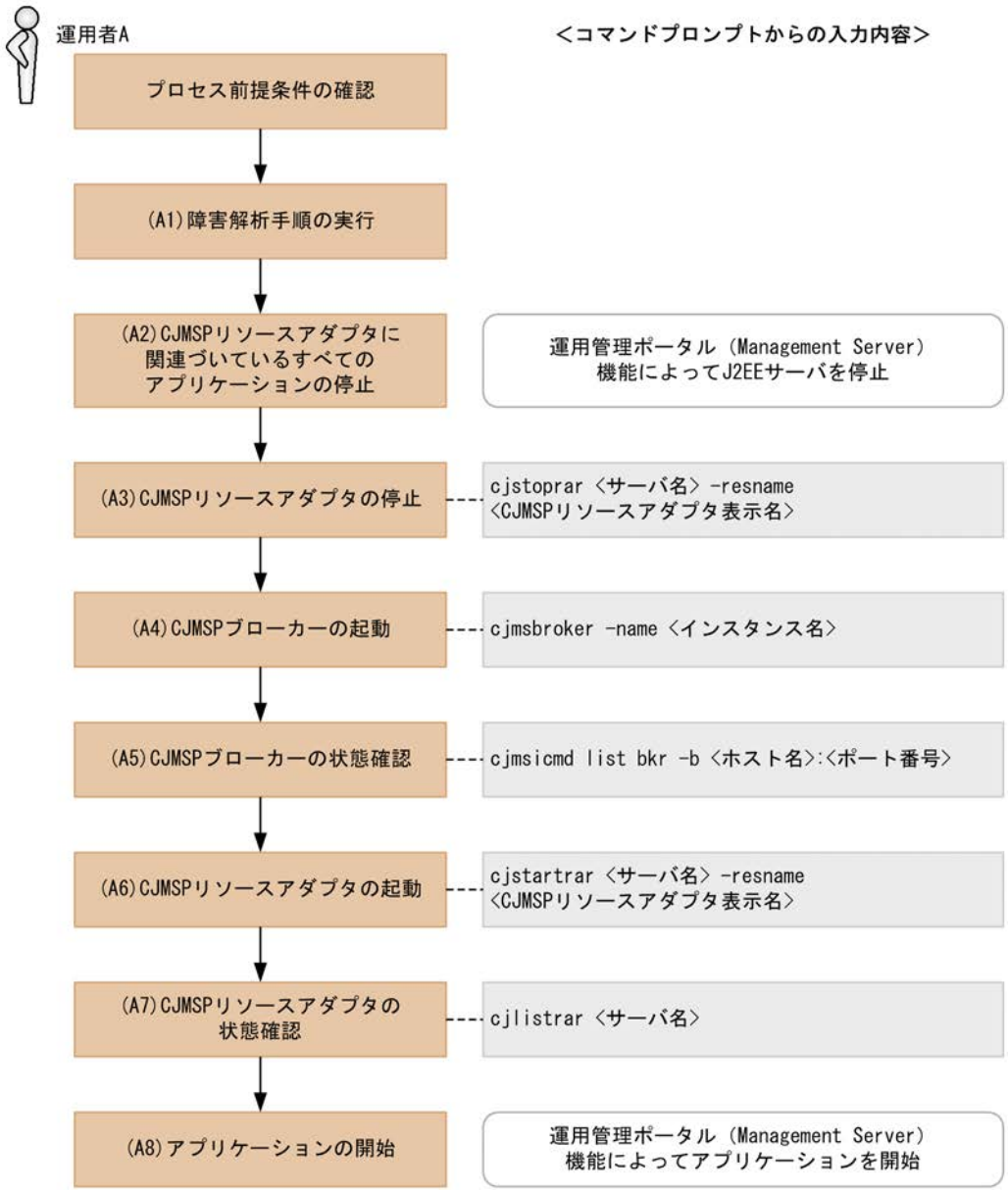
Management Server で構築したサーバ名

「running <CJMSP リソースアダプタ表示名>」と表示されていることを確認します。

付録 E.23　CJMSP ブローカーが障害によって停止したときの回復

CJMSP ブローカーが障害によって停止したときの回復手順を次の図に示します。

図 E-22　CJMSP ブローカーが障害によって停止したときの回復手順



(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが未起動であること。
- アプリケーションが起動済みであること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

「付録 E.20　CJMSP ブローカーが障害によって停止したときの解析」を参照して，障害を解析します。

(A2)

特にありません。

(A3)

<サーバ名>

Management Server で構築したサーバ名

<CJMSP リソースアダプタ表示名>

CJMSP リソースアダプタの表示名

CJMS プロバイダの場合は，デフォルトで「Cosminexus_JMS_Provider_RA」が設定されています。

(A4)

<インスタンス名>

省略した場合は，デフォルトで「cjmsbroker」を使用します。

複数インスタンスを使用したい場合には名称が重ならないように注意する必要があります。

重なる場合には-name オプションを指定し存在しない任意の名称を付与してください。

(A5)

State が「OPERATING」であることを確認します。

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

(A6)

<サーバ名>

Management Server で構築したサーバ名

<CJMSP リソースアダプタ表示名>

CJMSP リソースアダプタの表示名

CJMS プロバイダの場合は，デフォルトで「Cosminexus_JMS_Provider_RA」が設定されています。

(A7)

<サーバ名>

Management Server で構築したサーバ名

「running <CJMSP リソースアダプタ表示名>」と表示されていることを確認します。

(A8)

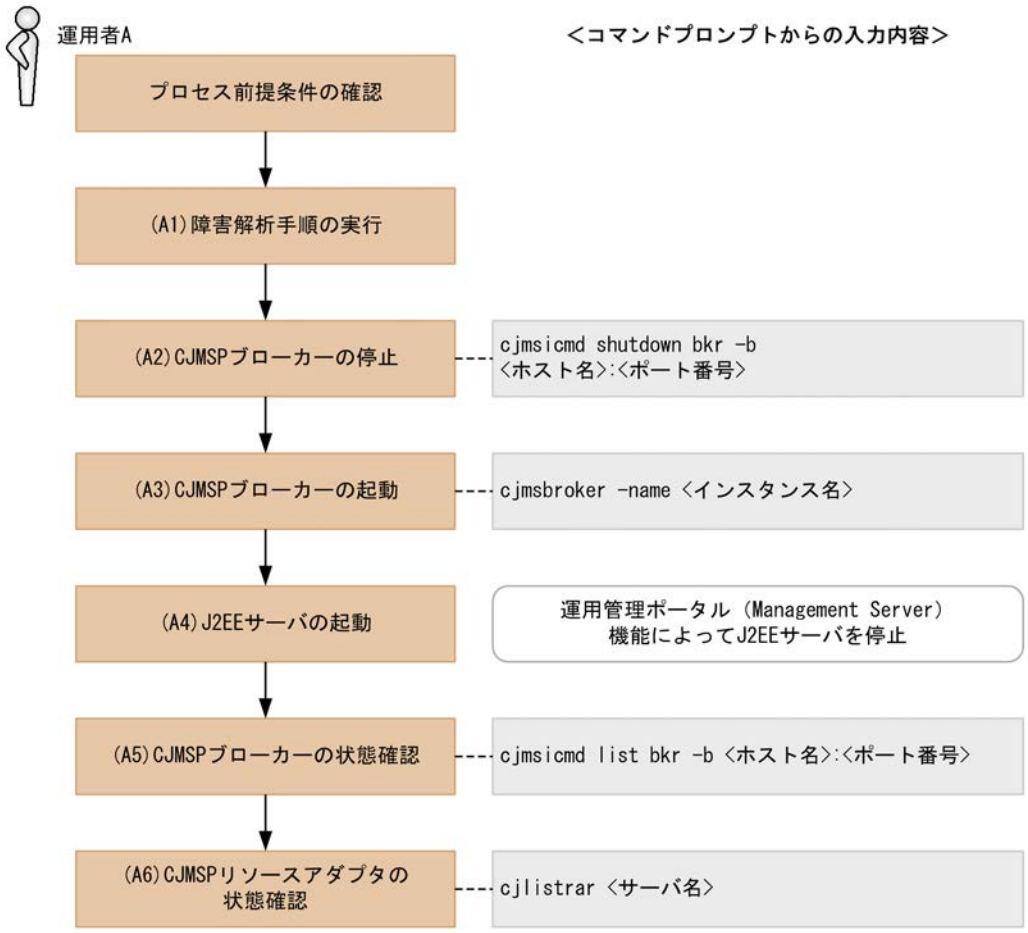
特にありません。

付録 E.24　CJMS プロバイダサービス無応答時の回復

CJMS プロバイダのサービス無応答状態からの回復手順について説明します。

CJMS プロバイダサービス無応答時の回復手順を次の図に示します。

図 E-23　CJMS プロバイダサービス無応答時の回復手順



(1)　プロセス前提条件

- J2EE サーバプロセスが起動済みであること。
- CJMSP リソースアダプタが開始済みであること。
- CJMSP ブローカープロセスが起動済みであること。
- アプリケーションが開始済みであること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

「付録 E.21　CJMS プロバイダサービス無応答時の解析」を参照して，障害を解析します。

(A2)

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

(A3)

<インスタンス名>

省略した場合は，デフォルトで「cjmsbroker」を使用します。

複数インスタンスを使用したい場合には名称が重ならないように注意する必要があります。

重なる場合には-name オプションを指定して存在しない任意の名称を付与してください。

(A4)

特にありません。

(A5)

<ホスト名>:<ポート番号>

CJMSP ブローカーを起動しているホスト名およびポート番号

CJMSP ブローカーの起動時にポート番号を省略した場合にはデフォルトの 7676 が使用されています。その場合，-b オプションの指定は省略できます。

State が「OPERATING」であることを確認します。

(A6)

<サーバ名>

Management Server で構築したサーバ名

「running <CJMSP リソースアダプタ表示名>」と表示されていることを確認します。

付録 E.25　CJMS プロバイダサービスインスタンスの削除

CJMS プロバイダのサービスの稼働中のインスタンスを削除する手順について説明します。

インスタンス名を変更したために以前のインスタンスが不要になった場合などに行います。

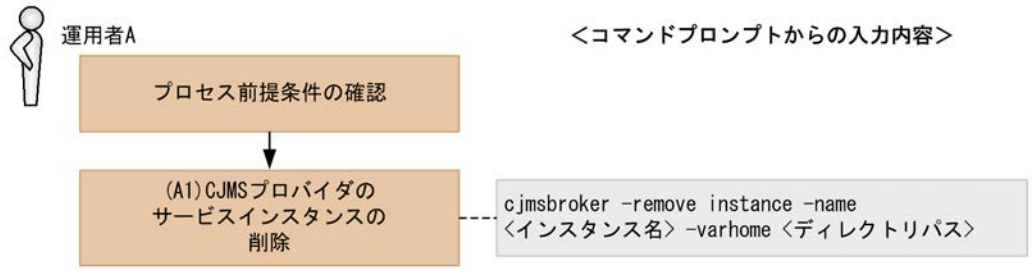
! 注意事項

この操作によって，送信先に存在するすべての情報が削除されます。

この操作を実行する前に，未処理の永続メッセージおよび永続サブスクライバーの情報が存在しないことを確認してください。

CJMS プロバイダのサービスインスタンス削除手順を次の図に示します。

図 E-24　CJMS プロバイダのサービスインスタンス削除手順



(1)　プロセス前提条件

- J2EE サーバプロセスが未起動であること。
- CJMSP リソースアダプタが未起動であること。
- CJMSP ブローカープロセスが未起動であること。

- アプリケーションが未起動であること。

(2)　実行に必要な情報および補足説明

(A1)などの記述は，図中の記述と対応しています。

(A1)

<インスタンス名>

省略した場合は，デフォルトで「cjmsbroker」を使用します。

複数インスタンスを使用したい場合には名称が重ならないように注意する必要があります。

重なる場合には-name オプションを指定し存在しない任意の名称を付与してください。

<ディレクトリパス>

削除したいインスタンスが存在する VAR_HOME のパス

CJMSP ブローカーインスタンス名に該当するディレクトリ（デフォルト名は cjmsbroker）を削除します。

この処理を行うと，prop と log ディレクトリ以外，すべてのインスタンスに関連するファイルとディレクトリを削除します。

インスタンスのディレクトリ自体が不要な場合，エクスプローラなどで次のディレクトリへ移動して，手動で削除してください。

```
<VAR_HOME>/instances
```

次回 CJMSP ブローカー起動時に削除したインスタンス名を使用して再度 CJMSP ブローカーを起動した場合には，インスタンスが再作成されます。

付録Ｆ　各バージョンでの主な機能変更

ここでは，09-70 よりも前のアプリケーションサーバの各バージョンでの主な機能の変更について，変更目的ごとに説明します。09-70 での主な機能変更については，「1.4　アプリケーションサーバ 09-70 での主な機能変更」を参照してください。

説明内容は次のとおりです。

- アプリケーションサーバの各バージョンで変更になった主な機能と，その概要を説明しています。機能の詳細については，「参照先マニュアル」の「参照個所」の記述を確認してください。「参照先マニュアル」および「参照個所」には，その機能についての 09-70 のマニュアルでの主な記載個所を記載しています。
- 「参照先マニュアル」に示したマニュアル名の「アプリケーションサーバ」は省略しています。

付録 F.1　09-60 での主な機能変更

(1)　標準機能・既存機能への対応

標準機能・既存機能への対応を目的として変更した項目を次の表に示します。

表 F-1　標準機能・既存機能への対応を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
G1GC への対応	G1GC を選択できるようになりました。	システム設計ガイド	7.15
		リファレンス 定義編(サーバ定義)	16.5
圧縮オブジェクトポインタ機能への対応	圧縮オブジェクトポインタ機能を使用できるようになりました。	機能解説 保守／移行編	9.18

(2)　信頼性の維持・向上

信頼性の維持・向上を目的として変更した項目を次の表に示します。

表 F-2　信頼性の維持・向上を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
ファイナライズ滞留解消機能の追加	ファイナライズ処理の滞留を解消でき，OS 資源の解放遅れなどの発生を抑止できるようになりました。	機能解説 保守／移行編	9.16

(3)　そのほかの目的

そのほかの目的で変更した項目を次の表に示します。

表 F-3　そのほかの目的による変更

項目	変更の概要	参照先マニュアル	参照個所
ログファイルの非同期出力機能の追加	ログのファイル出力を非同期でできるようになりました。	リファレンス 定義編(サーバ定義)	16.2

付録 F.2　09-50 での主な機能変更

(1)　開発生産性の向上

開発生産性の向上を目的として変更した項目を次の表に示します。

表 F-4　開発生産性の向上を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
Eclipse セットアップの簡略化	GUI を利用して Eclipse 環境をセットアップできるようになりました。	アプリケーション開発ガイド	1.1.5, 2.4
ユーザ拡張性能解析トレースを使ったデバッグ支援	ユーザ拡張性能解析トレース設定ファイルを開発環境で作成できるようになりました。	アプリケーション開発ガイド	1.1.3, 6.5

(2)　導入・構築の容易性強化

導入・構築の容易性強化を目的として変更した項目を次の表に示します。

表 F-5　導入・構築の容易性強化を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
仮想化環境でのシステム構成パターンの拡充	仮想化環境で使用できるティアの種類（http-tier, j2ee-tier および ctm-tier）が増えました。これによって，次のシステム構成パターンが構築できるようになりました。 • Web サーバと J2EE サーバを別のホストに配置するパターン • フロントエンド（サーブレット，JSP）とバックエンド（EJB）を分けて配置するパターン • CTM を使用するパターン	仮想化システム構築・運用ガイド	1.1.2

(3)　標準機能・既存機能への対応

標準機能・既存機能への対応を目的として変更した項目を次の表に示します。

表 F-6　標準機能・既存機能への対応を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
JDBC 4.0 仕様への対応	DB Connector で JDBC 4.0 仕様の HiRDB Type4 JDBC Driver，および SQL Server の JDBC ドライバに対応しました。	このマニュアル	3.6.3
Portable Global JNDI 名での命名規則の緩和	Portable Global JNDI 名に使用できる文字を追加しました。	このマニュアル	2.4.3
Servlet 3.0 仕様への対応	Servlet 3.0 の HTTP Cookie の名称，および URL のパスパラメタ名の変更が，Servlet 2.5 以前のバージョンでも使用できるようになりました。	機能解説 基本・開発編 (Web コンテナ)	2.7
Bean Validation と連携できるアプリケーションの適用拡大	CDI やユーザアプリケーションでも Bean Validation を使って検証できるようになりました。	このマニュアル	10 章

項目	変更の概要	参照先マニュアル	参照個所
JavaMail への対応	JavaMail 1.4 に準拠した API を使用したメール送受信機能を利用できるようになりました。	このマニュアル	8 章
javacore コマンドが使用できる OS の適用拡大	javacore コマンドを使って，Windows のスレッドダンプを取得できるようになりました。	リファレンス コマンド編	javacore（スレッドダンプの取得／Windows の場合）

(4)　信頼性の維持・向上

信頼性の維持・向上を目的として変更した項目を次の表に示します。

表 F-7　信頼性の維持・向上を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
コードキャッシュ領域の枯渇回避	システムで使用しているコードキャッシュ領域のサイズを確認して，領域が枯渇する前にしきい値を変更して領域枯渇するのを回避できるようになりました。	システム設計ガイド	7.2.6
		機能解説 保守／移行編	5.7.2, 5.7.3
		リファレンス 定義編 (サーバ定義)	16.1, 16.2, 16.4
明示管理ヒープ機能の効率的な適用への対応	自動解放処理時間を短縮し，明示管理ヒープ機能を効率的に適用するための機能として，Explicit ヒープに移動するオブジェクトを制御できる機能を追加しました。 • Explicit メモリブロックへのオブジェクト移動制御機能 • 明示管理ヒープ機能適用除外クラス指定機能 • Explicit ヒープ情報へのオブジェクト解放率情報の出力	システム設計ガイド	7.14.6
		機能解説 拡張編	8.2.2, 8.6.5, 8.10, 8.13.1, 8.13.3
		機能解説 保守／移行編	5.5
クラス別統計情報の出力範囲拡大	クラス別統計情報を含んだ拡張スレッドダンプに，static フィールドを基点とした参照関係を出力できるようになりました。	機能解説 保守／移行編	9.6

(5)　運用性の維持・向上

運用性の維持・向上を目的として変更した項目を次の表に示します。

表 F-8　運用性の維持・向上を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
EADs セッションフェイルオーバ機能のサポート	EADs と連携してセッションフェイルオーバ機能を実現する EADs セッションフェイルオーバ機能をサポートしました。	機能解説 拡張編	5 章, 7 章

項目	変更の概要	参照先マニュアル	参照個所
WAR による運用	WAR ファイルだけで構成された WAR アプリケーションを J2EE サーバにデプロイできるようになりました。	機能解説 基本・開発編 (Web コンテナ)	2.2.1
		このマニュアル	13.9
		リファレンス コマンド編	cjimportwar（WAR アプリケーションのインポート）
運用管理機能の同期実行による起動と停止	運用管理機能（Management Server および運用管理エージェント）の起動および停止を，同期実行するオプションを追加しました。	機能解説 運用／監視／連携編	2.6.1, 2.6.2, 2.6.3, 2.6.4
		リファレンス コマンド編	adminagentctl（運用管理エージェントの起動と停止），mngautorun（自動起動および自動再起動の設定／設定解除），mngsvrctl（Management Server の起動／停止／セットアップ）
明示管理ヒープ機能での Explicit メモリブロックの強制解放	javagc コマンドで，Explicit メモリブロックの解放処理を任意のタイミングで実行できるようになりました。	機能解説 拡張編	8.6.1, 8.9
		リファレンス コマンド編	javagc（GC の強制発生）

(6)　そのほかの目的

そのほかの目的で変更した項目を次の表に示します。

表Ｆ-9　そのほかの目的による変更

項目	変更の概要	参照先マニュアル	参照個所
定義情報の取得	snapshotlog（snapshot ログの収集）コマンドで定義ファイルだけを収集できるようになりました。	機能解説 保守／移行編	2.3
		リファレンス コマンド編	snapshotlog（snapshot ログの収集）
cjenvsetup コマンドのログ出力	Component Container 管理者のセットアップ（cjenvsetup コマンド）の実行情報がメッセージログに出力されるようになりました。	システム構築・運用ガイド	4.1.4
		機能解説 保守／移行編	4.20
		リファレンス コマンド編	cjenvsetup（Component Container 管理者のセットアップ）
BIG-IP v11 のサポート	使用できる負荷分散機の種類に BIG-IP v11 が追加になりました。	システム構築・運用ガイド	4.7.2
		仮想化システム構築・運用ガイド	2.1
明示管理ヒープ機能のイベントログへの CPU 時間の出力	Explicit メモリブロック解放処理に掛かった CPU 時間が，明示管理ヒープ機能のイベントログに出力されるようになりました。	機能解説 保守／移行編	5.11.3
ユーザ拡張性能解析トレースの機能拡張	ユーザ拡張性能解析トレースで，次の機能を追加しました。 • トレース対象の指定方法を通常のメソッド単位の指定方法に加えて，パッケージ単位またはクラス単位で指定できるようになりました。 • 使用できるイベント ID の範囲を拡張しました。 • ユーザ拡張性能解析トレース設定ファイルに指定できる行数の制限を緩和しました。 • ユーザ拡張性能解析トレース設定ファイルでトレース取得レベルを指定できるようになりました。	機能解説 保守／移行編	7.5.2, 7.5.3, 8.28.1
Session Bean の非同期呼び出し使用時の情報解析向上	PRF トレースのルートアプリケーション情報を使用して，呼び出し元と呼び出し先のリクエストを突き合わせることができるようになりました。	機能解説 基本・開発編 (EJB コンテナ)	2.17.3

付録 F.3　09-00 での主な機能変更

(1)　導入・構築の容易性強化

導入・構築の容易性強化を目的として変更した項目を次の表に示します。

表 F-10　導入・構築の容易性強化を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
仮想化環境での構築・運用の操作対象単位の変更	仮想化環境の構築・運用時の操作対象単位が仮想サーバから仮想サーバグループへ変更になりました。仮想サーバグループの情報を定義したファイルを使用して，複数の仮想サーバを管理ユニットへ一括で登録できるようになりました。	仮想化システム構築・運用ガイド	1.1.2
セットアップウィザードによる構築環境の制限解除	セットアップウィザードを使用して構築できる環境の制限が解除されました。ほかの機能で構築した環境があってもアンセットアップされて，セットアップウィザードで構築できるようになりました。	システム構築・運用ガイド	2.2.7
構築環境の削除手順の簡略化	Management Server を使用して構築したシステム環境を削除する機能（mngunsetup コマンド）の追加によって，削除手順を簡略化しました。	システム構築・運用ガイド	4.1.37
		運用管理ポータル操作ガイド	3.6，5.4
		リファレンス コマンド編	mngunsetup（Management Server の構築環境の削除）

(2)　標準機能・既存機能への対応

標準機能・既存機能への対応を目的として変更した項目を次の表に示します。

表 F-11　標準機能・既存機能への対応を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
Servlet 3.0 への対応	Servlet 3.0 に対応しました。	機能解説 基本・開発編 (Web コンテナ)	6 章
EJB 3.1 への対応	EJB 3.1 に対応しました。	機能解説 基本・開発編 (EJB コンテナ)	2 章
JSF 2.1 への対応	JSF 2.1 に対応しました。	機能解説 基本・開発編 (Web コンテナ)	3 章
JSTL 1.2 への対応	JSTL 1.2 に対応しました。	機能解説 基本・開発編 (Web コンテナ)	3 章
CDI 1.0 への対応	CDI 1.0 に対応しました。	このマニュアル	9 章

項目	変更の概要	参照先マニュアル	参照個所
Portable Global JNDI 名の利用	Portable Global JNDI 名を利用したオブジェクトのルックアップができるようになりました。	このマニュアル	2.4
JAX-WS 2.2 への対応	JAX-WS 2.2 に対応しました。	Web サービス開発ガイド	1.1, 16.1.5, 16.1.7, 16.2.1, 16.2.6, 16.2.10, 16.2.12, 16.2.13, 16.2.14, 16.2.16, 16.2.17, 16.2.18, 16.2.20, 16.2.22, 19.1, 19.2.3, 37.2, 37.6.1, 37.6.2, 37.6.3
JAX-RS 1.1 への対応	JAX-RS 1.1 に対応しました。	Web サービス開発ガイド	1.1, 1.2.2, 1.3.2, 1.4.2, 1.5.1, 1.6, 2.3, 11 章, 12 章, 13 章, 17 章, 24 章, 39 章

(3)　信頼性の維持・向上

信頼性の維持・向上を目的として変更した項目を次の表に示します。

表 F-12　信頼性の維持・向上を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
SSL/TLS 通信での TLSv1.2 の使用	RSA BSAFE SSL-J を使用して，TLSv1.2 を含むセキュリティ・プロトコルで SSL/TLS 通信ができるようになりました。	－	－

（凡例）－：09-70 で削除された機能です。

(4)　運用性の維持・向上

運用性の維持・向上を目的として変更した項目を次の表に示します。

表 F-13　運用性の維持・向上を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
Web コンテナ全体の実行待ちキューの総和の監視	Web コンテナ全体の実行待ちキューの総和を稼働情報に出力して監視できるようになりました。	機能解説 運用／監視／連携編	3 章
アプリケーションの性能解析トレース（ユーザ拡張トレース）の出力	ユーザが開発したアプリケーションの処理性能を解析するための性能解析トレースを，アプリケーションの変更をしないで出力できるようになりました。	機能解説 保守／移行編	7 章
仮想化環境でのユーザスクリプトを使用した運用	任意のタイミングでユーザ作成のスクリプト（ユーザスクリプト）を仮想サーバ上で実行できるようになりました。	仮想化システム構築・運用ガイド	7.8
運用管理ポータル改善	運用管理ポータルの次の画面で，手順を示すメッセージを画面に表示するように変更しました。 • ［設定情報の配布］画面 • Web サーバ，J2EE サーバおよび SFO サーバの起動画面 • Web サーバクラスタと J2EE サーバクラスタの一括起動，一括再起動および起動画面	運用管理ポータル操作ガイド	10.11.1, 11.9.2, 11.10.2, 11.11.2, 11.11.4, 11.11.6, 11.12.2, 11.13.2, 11.13.4, 11.13.6
運用管理機能の再起動機能の追加	運用管理機能（Management Server および運用管理エージェント）で自動再起動が設定できるようになり，運用管理機能で障害が発生した場合でも運用が継続できるようになりました。また，自動起動の設定方法も変更になりました。	機能解説 運用／監視／連携編	2.4.1, 2.4.2, 2.6.3, 2.6.4
		リファレンス コマンド編	mngautorun（自動起動および自動再起動の設定／設定解除）

(5)　そのほかの目的

そのほかの目的で変更した項目を次の表に示します。

表 F-14　そのほかの目的による変更

項目	変更の概要	参照先マニュアル	参照個所
ログ出力時のファイル切り替え単位の変更	ログ出力時に，日付ごとに出力先のファイルを切り替えられるようになりました。	機能解説 保守／移行編	3.2.1
Web サーバの名称の変更	アプリケーションサーバに含まれる Web サーバの名称を HTTP Server に変更しました。	HTTP Server	－
BIG-IP の API（SOAP アーキテクチャ）を使用した直接接続への対応	BIG-IP（負荷分散機）で API（SOAP アーキテクチャ）を使用した直接接続に対応しました。 また，API を使用した直接接続を使用する場合に，負荷分散機の接続環境を設定する方法が変更になりました。	システム構築・運用ガイド	4.7.3，付録 K
		仮想化システム構築・運用ガイド	2.1，付録 C

項目	変更の概要	参照先マニュアル	参照個所
BIG-IP の API（SOAP アーキテクチャ）を使用した直接接続への対応	BIG-IP（負荷分散機）で API（SOAP アーキテクチャ）を使用した直接接続に対応しました。 また，API を使用した直接接続を使用する場合に，負荷分散機の接続環境を設定する方法が変更になりました。	機能解説 セキュリティ管理機能編	8.2，8.4，8.5，8.6，18.2，18.3，18.4

（凡例）－：マニュアル全体を参照する

付録 F.4　08-70 での主な機能変更

(1)　導入・構築の容易性強化

導入・構築の容易性強化を目的として変更した項目を次の表に示します。

表 F-15　導入・構築の容易性強化を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
運用管理ポータル改善	運用管理ポータルの画面で，リソースアダプタの属性を定義するプロパティ（Connector 属性ファイルの設定内容）の設定，および接続テストができるようになりました。また，運用管理ポータルの画面で，J2EE アプリケーション（ear ファイルおよび zip ファイル）を Management Server にアップロードできるようになりました。	ファーストステップガイド	3.5
		運用管理ポータル操作ガイド	－
page/tag ディレクティブの import 属性暗黙インポート機能の追加	page/tag ディレクティブの import 属性暗黙インポート機能を使用できるようになりました。	機能解説 基本・開発編 (Web コンテナ)	2.3.7
仮想化環境での JP1 製品に対する環境設定の自動化対応	仮想サーバへのアプリケーションサーバ構築時に，仮想サーバに対する JP1 製品の環境設定を，フックスクリプトで自動的に設定できるようになりました。	仮想化システム構築・運用ガイド	7.7.2
統合ユーザ管理機能の改善	ユーザ情報リポジトリでデータベースを使用する場合に，データベース製品の JDBC ドライバを使用して，データベースに接続できるようになりました。 DABroker Library の JDBC ドライバによるデータベース接続はサポート外になりました。 簡易構築定義ファイルおよび運用管理ポータルの画面で，統合ユーザ管理機能に関する設定ができるようになりました。 また，Active Directory の場合，DN で日本語などの 2 バイト文字に対応しました。	機能解説 セキュリティ管理機能編	5 章，14.3
		運用管理ポータル操作ガイド	3.5，10.9.1
HTTP Server 設定項目の拡充	簡易構築定義ファイルおよび運用管理ポータルの画面で，HTTP Server の動作環境を定義するディレクティブ（httpsd.conf の設定内容）を直接設定できるようになりました。	システム構築・運用ガイド	4.1.21
		運用管理ポータル操作ガイド	10.10.1
		リファレンス 定義編 (サーバ定義)	4.13

（凡例）－：マニュアル全体を参照する

(2)　標準機能・既存機能への対応

標準機能・既存機能への対応を目的として変更した項目を次の表に示します。

表 F-16　標準機能・既存機能への対応を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
ejb-jar.xml の指定項目の追加	ejb-jar.xml に，クラスレベルインターセプタおよびメソッドレベルインターセプタを指定できるようになりました。	機能解説 基本・開発編 (EJB コンテナ)	2.15
パラレルコピーガーベージコレクションへの対応	パラレルコピーガーベージコレクションを選択できるようになりました。	リファレンス 定義編 (サーバ定義)	16.5
Connector 1.5 仕様に準拠した Inbound リソースアダプタのグローバルトランザクションへの対応	Connector 1.5 仕様に準拠したリソースアダプタで Transacted Delivery を使用できるようにしました。これによって，Message-driven Bean を呼び出す EIS がグローバルトランザクションに参加できるようになりました。	このマニュアル	3.16.3
TP1 インバウンドアダプタの MHP への対応	TP1 インバウンドアダプタを使用してアプリケーションサーバを呼び出す OpenTP1 のクライアントとして，MHP を使用できるようになりました。	このマニュアル	4 章
cjrarupdate コマンドの FTP インバウンドアダプタへの対応	cjrarupdate コマンドでバージョンアップできるリソースアダプタに FTP インバウンドアダプタを追加しました。	リファレンス コマンド編	2.2

(3)　信頼性の維持・向上

信頼性の維持・向上を目的として変更した項目を次の表に示します。

表 F-17　信頼性の維持・向上を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
データベースセッションフェイルオーバ機能の改善	性能を重視するシステムで，グローバルセッション情報を格納したデータベースのロックを取得しないモードを選択できるようになりました。また，データベースを更新しない，参照専用のリクエストを定義できるようになりました。	機能解説 拡張編	6 章
OutOfMemory ハンドリング機能の対象となる処理の拡大	OutOfMemory ハンドリング機能の対象となる処理を追加しました。	機能解説 保守／移行編	2.5.7
		リファレンス 定義編 (サーバ定義)	16.2
HTTP セッションで利用する Explicit ヒープの省メモリ化機能の追加	HTTP セッションで利用する Explicit ヒープのメモリ使用量を抑止する機能を追加しました。	機能解説 拡張編	8.11

(4)　運用性の維持・向上

運用性の維持・向上を目的として変更した項目を次の表に示します。

表 F-18　運用性の維持・向上を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
仮想化環境での JP1 製品を使用したユーザ認証への対応（クラウド運用対応）	JP1 連携時に，JP1 製品の認証サーバを利用して，仮想サーバマネージャを使用するユーザを管理・認証できるようになりました。	仮想化システム構築・運用ガイド	1.2.2，3 章，4 章，5 章，6 章，7.9

(5)　そのほかの目的

そのほかの目的で変更した項目を次の表に示します。

表 F-19　そのほかの目的による変更

項目	変更の概要	参照先マニュアル	参照個所
負荷分散機への API（REST アーキテクチャ）を使用した直接接続の対応	負荷分散機への接続方法として，API（REST アーキテクチャ）を使用した直接接続に対応しました。 また，使用できる負荷分散機の種類に ACOS（AX2500）が追加になりました。	システム構築・運用ガイド	4.7.2，4.7.3
		仮想化システム構築・運用ガイド	2.1
		リファレンス 定義編(サーバ定義)	4.5
snapshot ログ収集時のタイムアウトへの対応と収集対象の改善	snapshot ログの収集が指定した時間で終了（タイムアウト）できるようになりました。一次送付資料として収集される内容が変更になりました。	機能解説 保守／移行編	付録 A

付録 F.5　08-53 での主な機能変更

(1)　導入・構築の容易性強化

導入・構築の容易性強化を目的として変更した項目を次の表に示します。

表 F-20　導入・構築の容易性強化を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
さまざまなハイパーバイザに対応した仮想化環境の構築	さまざまなハイパーバイザを使用して実現する仮想サーバ上に，アプリケーションサーバを構築できるようになりました。 また，複数のハイパーバイザが混在する環境にも対応しました。	仮想化システム構築・運用ガイド	2 章，3 章，5 章

(2)　標準機能・既存機能への対応

標準機能・既存機能への対応を目的として変更した項目を次の表に示します。

表 F-21　標準機能・既存機能への対応を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
トランザクション連携に対応した OpenTP1 からの呼び出し	OpenTP1 からアプリケーションサーバ上で動作する Message-driven Bean を呼び出すときに，トランザクション連携ができるようになりました。	このマニュアル	4 章

項目	変更の概要	参照先マニュアル	参照個所
JavaMail	POP3 に準拠したメールサーバと連携して，JavaMail 1.3 に準拠した API を使用したメール受信機能を利用できるようになりました。	このマニュアル	8 章

(3)　信頼性の維持・向上

信頼性の維持・向上を目的として変更した項目を次の表に示します。

表 F-22　信頼性の維持・向上を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
JavaVM のトラブルシュート機能強化	JavaVM のトラブルシュート機能として，次の機能が使用できるようになりました。 • OutOfMemoryError 発生時の動作を変更できるようになりました。 • JIT コンパイル時に，C ヒープ確保量の上限値を設定できるようになりました。 • スレッド数の上限値を設定できるようになりました。 • 拡張 verbosegc 情報の出力項目を拡張しました。	機能解説 保守／移行編	4 章，5 章，9 章

(4)　運用性の維持・向上

運用性の維持・向上を目的として変更した項目を次の表に示します。

表 F-23　運用性の維持・向上を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
JP1/ITRM への対応	IT リソースを一元管理する製品である JP1/ITRM に対応しました。	仮想化システム構築・運用ガイド	1.3，2.1

(5)　そのほかの目的

そのほかの目的で変更した項目を次の表に示します。

表 F-24　そのほかの目的による変更

項目	変更の概要	参照先マニュアル	参照個所
Microsoft IIS 7.0 および Microsoft IIS 7.5 への対応	Web サーバとして Microsoft IIS 7.0 および Microsoft IIS 7.5 に対応しました。	－	－
HiRDB Version 9 および SQL Server 2008 への対応	データベースとして次の製品に対応しました。 • HiRDB Server Version 9 • HiRDB/Developer's Kit Version 9 • HiRDB/Run Time Version 9 • SQL Server 2008 また，SQL Server 2008 に対応する JDBC ドライバとして，SQL Server JDBC Driver に対応しました。	このマニュアル	3 章

（凡例）－：該当なし。

付録 F.6　08-50 での主な機能変更

(1)　導入・構築の容易性強化

導入・構築の容易性強化を目的として変更した項目を次の表に示します。

表 F-25　導入・構築の容易性強化を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
Web サービスプロバイダ側での web.xml の指定必須タグの変更	Web サービスプロバイダ側での web.xml で，listener タグ，servlet タグおよび servlet-mapping タグの指定を必須から任意に変更しました。	リファレンス 定義編(サーバ定義)	2.4
論理サーバのネットワークリソース使用	J2EE アプリケーションからほかのホスト上にあるネットワークリソースやネットワークドライブにアクセスするための機能を追加しました。	機能解説 運用／監視／連携編	1.2.3，5.2，5.7
サンプルプログラムの実行手順の簡略化	一部のサンプルプログラムを EAR 形式で提供することによって，サンプルプログラムの実行手順を簡略化しました。	ファーストステップガイド	3.5
		システム構築・運用ガイド	付録 M
運用管理ポータルの画面動作の改善	画面の更新間隔のデフォルトを「更新しない」から「3秒」に変更しました。	運用管理ポータル操作ガイド	7.4.1
セットアップウィザードの完了画面の改善	セットアップウィザード完了時の画面に，セットアップで使用した簡易構築定義ファイルおよび Connector 属性ファイルが表示されるようになりました。	システム構築・運用ガイド	2.2.6
仮想化環境の構築	ハイパーバイザを使用して実現する仮想サーバ上に，アプリケーションサーバを構築する手順を追加しました。※	仮想化システム構築・運用ガイド	3 章，5 章

注※

08-50 モードで構築する場合は，マニュアル「アプリケーションサーバ 仮想化システム構築・運用ガイド」の「付録 D　08-50 モードの仮想サーバマネージャを利用する場合の設定」を参照してください。

(2)　標準機能・既存機能への対応

標準機能・既存機能への対応を目的として変更した項目を次の表に示します。

表 F-26　標準機能・既存機能への対応を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
OpenTP1 からの呼び出しへの対応	OpenTP1 からアプリケーションサーバ上で動作する Message-driven Bean を呼び出せるようになりました。	このマニュアル	4 章
JMS への対応	JMS 1.1 仕様に対応した CJMS プロバイダ機能を使用できるようになりました。	このマニュアル	7 章

項目	変更の概要	参照先マニュアル	参照個所
Java SE 6 への対応	Java SE 6 の機能が使用できるようになりました。	機能解説 保守／移行編	5.5, 5.8.1
ジェネリクスの使用への対応	EJB でジェネリクスを使用できるようになりました。	機能解説 基本・開発編 (EJB コンテナ)	4.2.19

(3)　信頼性の維持・向上

信頼性の維持・向上を目的として変更した項目を次の表に示します。

表 F-27　信頼性の維持・向上を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
明示管理ヒープ機能の使用性向上	自動配置設定ファイルを使用して，明示管理ヒープ機能を容易に使用できるようになりました。	システム設計ガイド	7.2, 7.7.3, 7.11.5, 7.12.1
		機能解説 拡張編	8 章
データベースセッションフェイルオーバ機能の URI 単位での抑止	データベースセッションフェイルオーバ機能を使用する場合に，機能の対象外にするリクエストを URI 単位で指定できるようになりました。	機能解説 拡張編	5.6.1
仮想化環境での障害監視	仮想化システムで，仮想サーバ監視し，障害の発生を検知できるようになりました。	仮想化システム構築・運用ガイド	付録 D

(4)　運用性の維持・向上

運用性の維持・向上を目的として変更した項目を次の表に示します。

表 F-28　運用性の維持・向上を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
管理ユーザアカウントの省略	運用管理ポータル，Management Server のコマンド，または Smart Composer 機能のコマンドで，ユーザのログイン ID およびパスワードの入力を省略できるようになりました。	システム構築・運用ガイド	4.1.15
		運用管理ポータル操作ガイド	2.2，7.1.1，7.1.2，7.1.3，8.1，8.2.1，付録 F.2
		リファレンス コマンド編	1.4，mngsvrctl (Management Server の起動／停止／セットアップ)，mngsvrutil（Management Server の運用管理コマンド），8.3，cmx_admin_passwd (Management Server の管理ユーザアカウントの設定)
仮想化環境の運用	仮想化システムで，複数の仮想サーバを対象にした一括起動・一括停止，スケールイン・スケールアウト	仮想化システム構築・運用ガイド	4 章，6 章

項目	変更の概要	参照先マニュアル	参照個所
仮想化環境の運用	などを実行する手順を追加しました。※	仮想化システム構築・運用ガイド	４章，６章

注※

08-50 モードで運用する場合は，マニュアル「アプリケーションサーバ 仮想化システム構築・運用ガイド」の「付録Ｄ　08-50 モードの仮想サーバマネージャを利用する場合の設定」を参照してください。

(5)　そのほかの目的

そのほかの目的で変更した項目を次の表に示します。

表Ｆ-29　そのほかの目的による変更

項目	変更の概要	参照先マニュアル	参照個所
Tenured 領域内不要オブジェクト統計機能	Tenured 領域内で不要となったオブジェクトだけを特定できるようになりました。	機能解説 保守／移行編	9.8
Tenured 増加要因の基点オブジェクトリスト出力機能	Tenured 領域内不要オブジェクト統計機能を使って特定した，不要オブジェクトの基点となるオブジェクトの情報を出力できるようになりました。		9.9
クラス別統計情報解析機能	クラス別統計情報を CSV 形式で出力できるようになりました。		9.10
論理サーバの自動再起動回数オーバー検知によるクラスタ系切り替え	Management Server を系切り替えの監視対象としているクラスタ構成の場合，論理サーバが異常停止状態(自動再起動回数をオーバーした状態，または自動再起動回数の設定が 0 のときに障害を検知した状態)になったタイミングでの系切り替えができるようになりました。	機能解説 運用／監視／連携編	18.4.3, 18.5.3, 20.2.2, 20.3.3, 20.3.4
ホスト単位管理モデルを対象とした系切り替えシステム	クラスタソフトウェアと連携したシステム運用で，ホスト単位管理モデルを対象にした系切り替えができるようになりました。		20 章
ACOS（AX2000，BS320）のサポート	使用できる負荷分散機の種類に ACOS（AX2000，BS320）が追加になりました。	システム構築・運用ガイド	4.7.2, 4.7.3, 4.7.5, 4.7.6，付録 K，付録 K.2
		リファレンス 定義編(サーバ定義)	4.5, 4.6.2, 4.6.4, 4.6.5, 4.6.6, 4.10.1
CMT でトランザクション管理をする場合に Stateful Session Bean（SessionSynchronization）に指定できるトランザクション属性の追加	CMT でトランザクション管理をする場合に，Stateful Session Bean（SessionSynchronization）にトランザクション属性として Supports, NotSupported および Never を指定できるようになりました。	機能解説 基本・開発編(EJB コンテナ)	2.7.3

項目	変更の概要	参照先マニュアル	参照個所
OutOfMemoryError 発生時の運用管理エージェントの強制終了	JavaVM で OutOfMemoryError が発生したときに，運用管理エージェントが強制終了するようになりました。	機能解説 保守／移行編	2.5.8
スレッドの非同期並行処理	TimerManager および WorkManager を使用して，非同期タイマ処理および非同期スレッド処理を実現できるようになりました。	機能解説 拡張編	10 章

付録 F.7　08-00 での主な機能変更

(1)　開発生産性の向上

開発生産性の向上を目的として変更した項目を次の表に示します。

表 F-30　開発生産性の向上を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
ほかのアプリケーションサーバ製品からの移行容易化	ほかのアプリケーションサーバ製品からの移行を円滑に実施するため，次の機能を使用できるようになりました。 • HTTP セッションの上限が例外で判定できるようになりました。 • JavaBeans の ID が重複している場合や，カスタムタグの属性名と TLD の定義で大文字・小文字が異なる場合に，トランスレーションエラーが発生することを抑止できるようになりました。	機能解説 基本・開発編 (Web コンテナ)	2.3, 2.7.5
cosminexus.xml の提供	アプリケーションサーバ独自の属性を cosminexus.xml に記載することによって，J2EE アプリケーションを J2EE サーバにインポート後，プロパティの設定をしないで開始できるようになりました。	このマニュアル	11.3

(2)　標準機能への対応

標準機能への対応を目的として変更した項目を次の表に示します。

表 F-31　標準機能への対応を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
Servlet 2.5 への対応	Servlet 2.5 に対応しました。	機能解説 基本・開発編 (Web コンテナ)	2.2, 2.5.4, 2.6，6 章
JSP 2.1 への対応	JSP 2.1 に対応しました。	機能解説 基本・開発編 (Web コンテナ)	2.3.1, 2.3.3, 2.5，2.6, 6 章
JSP デバッグ	MyEclipse を使用した開発環境で JSP デバッグができるようになりました。※	機能解説 基本・開発編 (Web コンテナ)	2.4

項目	変更の概要	参照先マニュアル	参照個所
タグライブラリのライブラリ JAR への格納と TLD のマッピング	タグライブラリをライブラリ JAR に格納した場合に，Web アプリケーション開始時に Web コンテナによってライブラリ JAR 内の TLD ファイルを検索し，自動的にマッピングできるようになりました。	機能解説 基本・開発編 (Web コンテナ)	2.3.4
application.xml の省略	J2EE アプリケーションで application.xml が省略できるようになりました。	このマニュアル	11.4
アノテーションと DD の併用	アノテーションと DD を併用できるようになり，アノテーションで指定した内容を DD で更新できるようになりました。	このマニュアル	12.5
アノテーションの Java EE 5 標準準拠（デフォルトインターセプタ）	デフォルトインターセプタをライブラリ JAR に格納できるようになりました。また，デフォルトインターセプタから DI できるようになりました。	このマニュアル	11.4
@Resource の参照解決	@Resource でリソースの参照解決ができるようになりました。	このマニュアル	12.4
JPA への対応	JPA 仕様に対応しました。	このマニュアル	5 章, 6 章

注※　09-00 以降では，WTP を使用した開発環境で JSP デバッグ機能を使用できます。

(3)　信頼性の維持・向上

信頼性の維持・向上を目的として変更された項目を次の表に示します。

表 F-32　信頼性の維持・向上を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
セッション情報の永続化	HTTP セッションのセッション情報をデータベースに保存して引き継げるようになりました。	機能解説 拡張編	5 章, 6 章
FullGC の抑止	FullGC の要因となるオブジェクトを Java ヒープ外に配置することで, FullGC 発生を抑止できるようになりました。	機能解説 拡張編	8 章
クライアント性能モニタ	クライアント処理に掛かった時間を調査・分析できるようになりました。	－	－

（凡例）－：09-00 で削除された機能です。

(4)　運用性の維持・向上

運用性の維持・向上を目的として変更された項目を次の表に示します。

表 F-33　運用性の維持・向上を目的とした変更

項目	変更の概要	参照先マニュアル	参照個所
運用管理ポータルでのアプリケーション操作性向上	アプリケーションおよびリソースの操作について，サーバ管理コマンドと運用管理ポータルの相互運用ができるようになりました。	運用管理ポータル操作ガイド	1.1.3

(5)　そのほかの目的

そのほかの目的で変更された項目を次の表に示します。

表 F-34　そのほかの目的による変更

項目	変更の概要	参照先マニュアル	参照個所
無効な HTTP Cookie の削除	無効な HTTP Cookie を削除できるようになりました。	機能解説 基本・開発編 (Web コンテナ)	2.7.4
ネーミングサービスの障害検知	ネーミングサービスの障害が発生した場合に，EJB クライアントが，より早くエラーを検知できるようになりました。	このマニュアル	2.9
コネクション障害検知タイムアウト	コネクション障害検知タイムアウトのタイムアウト時間を指定できるようになりました。	このマニュアル	3.15.1
Oracle11g への対応	データベースとして Oracle11g が使用できるようになりました。	このマニュアル	3 章
バッチ処理のスケジューリング	バッチアプリケーションの実行を CTM によってスケジューリングできるようになりました。	機能解説 拡張編	4 章
バッチ処理のログ	バッチ実行コマンドのログファイルのサイズ，面数，ログの排他処理失敗時のリトライ回数とリトライ間隔を指定できるようになりました。	リファレンス 定義編 (サーバ定義)	3.6
snapshot ログ	snapshot ログの収集内容が変更されました。	機能解説 保守／移行編	付録 A.1，付録 A.2
メソッドキャンセルの保護区公開	メソッドキャンセルの対象外となる保護区リストの内容を公開しました。	機能解説 運用／監視／連携編	付録 C
統計前のガーベージコレクション選択機能	クラス別統計情報を出力する前に，ガーベージコレクションを実行するかどうかを選択できるようになりました。	機能解説 保守／移行編	9.7
Survivor 領域の年齢分布情報出力機能	Survivor 領域の Java オブジェクトの年齢分布情報を JavaVM ログファイルに出力できるようになりました。	機能解説 保守／移行編	9.11
ファイナライズ滞留解消機能	JavaVM のファイナライズ処理の状態を監視して，処理の滞留を解消できるようになりました。	－	－
サーバ管理コマンドの最大ヒープサイズの変更	サーバ管理コマンドが使用する最大ヒープサイズが変更されました。	リファレンス 定義編 (サーバ定義)	5.2，5.3
推奨しない表示名を指定された場合の対応	J2EE アプリケーションで推奨しない表示名を指定された場合にメッセージが出力されるようになりました。	メッセージ(構築／運用／開発用)	KDJE42374-W

（凡例）－：09-00 で削除された機能です。

付録 G　用語解説

マニュアルで使用する用語について

マニュアル「アプリケーションサーバ & BPM/ESB 基盤 用語解説」を参照してください。

索引

記号

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

う

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

つ

て

と

に

ね

は

ふ

へ

ま

み

め

も

ゆ

よ

ら

り

る

れ

ろ

わ