Oracle® Application Server Containers for J2EE

サーブレット開発者ガイド

10*g* リリース 2(10.1.2)

部品番号 : B15633-02

2005 年 10 月

ORACLE®

Oracle Application Server Containers for J2EE サーブレット開発者ガイド , 10*g* リリース 2（10.1.2）

部品番号 : B15633-02

原本名 : Oracle Application Server Containers for J2EE Servlet Developer's Guide, 10*g* Release 2 (10.1.2)

原本部品番号 : B14017-02

原本著者 : Brian Wright

原本著者協力者 : Tim Smith

原本協力者 : Bryan Atsatt, Ashok Banerjee, Bill Bishop, Olivier Caudron, Cania Chung, Olaf Heimburger, Gerald Ingalls, James Kirsch, Sunil Kunisetty, Philippe Le Mouel, David Leibs, Sastry Malladi, Jasen Minton, Debu Panda, Lenny Phan, Shiva Prasad, Paolo Ramasso, Charlie Shapiro, JJ Snyder, Joyce Yang, Serge Zloto, Sheryl Maring, Ellen Siegal

Copyright © 2002, 2005, Oracle. All rights reserved.

制限付権利の説明

このプログラム（ソフトウェアおよびドキュメントを含む）には、オラクル社およびその関連会社に所有権のある情報が含まれています。このプログラムの使用または開示は、オラクル社およびその関連会社との契約に記された制約条件に従うものとします。著作権、特許権およびその他の知的財産権と工業所有権に関する法律により保護されています。

独立して作成された他のソフトウェアとの互換性を得るために必要な場合、もしくは法律によって規定される場合を除き、このプログラムのリバース・エンジニアリング、逆アセンブル、逆コンパイル等は禁止されています。

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

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

U.S. GOVERNMENT RIGHTS

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

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

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

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

目次

はじめに v

対象読者 vi
ドキュメントのアクセシビリティについて vi
関連ドキュメント vi
表記規則 viii
サポートおよびサービス ix

1 サーブレットの概要

サーブレットの概要 1-2
サーブレット・テクノロジのレビュー 1-2
サーブレットの利点 1-2
サーブレット・インタフェースとリクエスト・オブジェクトおよびレスポンス・オブジェクト 1-3
サーブレットおよびサーブレット・コンテナ 1-4
サーブレット・セッションの概要 1-5
サーブレット・コンテキストの概要 1-6
サーブレット構成オブジェクトの概要 1-8
サーブレット・フィルタの概要 1-8
イベント・リスナーの概要 1-9
JSP ページおよびその他の J2EE コンポーネント・タイプ 1-9

初めてのサーブレット作成の例 1-9
Hello World コード 1-9
サーブレットのコンパイルおよびデプロイ 1-10
サーブレットの実行 1-10

2 サーブレットの開発

開発用 OC4J スタンドアロン 2-2
概要 : OC4J スタンドアロンの使用 2-2
開発用の主な OC4J のフラグ 2-3
OC4J スタンドアロンからの tools.jar の削除 2-4

サーブレット開発の基本と主要な考慮事項 2-5
サンプルのコード・テンプレート 2-5
サーブレットのライフ・サイクル 2-6
サーブレットの事前ロード 2-6
サーブレットのクラスロードおよびアプリケーションの再デプロイ 2-7
サーブレット情報の交換 2-10
サーブレットのインクルードおよび転送 2-10

サーブレットのスレッド・モデルと関連する考慮事項 2-11
サーブレットのパフォーマンスと監視 2-12
JDK 1.4 の考慮事項 : パッケージ内に存在しないクラスを起動できない 2-13
追加の Oracle 機能 2-14
OC4J のロギング 2-14
サーブレットのデバッグ 2-16
Oracle JDeveloper によるサーブレット開発のサポート 2-18
オープン・ソース・フレームワークの OC4J サポートの概要 2-18
サーブレットの起動 2-19
URL 構成要素のサマリー 2-19
OC4J 開発時におけるクラス名によるサーブレットの起動 2-22
Oracle Application Server 本番環境でのサーブレットの起動 2-23
OC4J スタンドアロン環境でのサーブレットの起動 2-24
サーブレットのセッション 2-24
セッション・トラッキング 2-25
HttpSession インタフェースの機能 2-27
セッションのキャンセル 2-28
分散アプリケーションでのセッションのレプリケーション 2-28
セッション・サーブレットの例 2-30
サーブレットのセキュリティ 2-32
セキュリティ機能の使用 2-33
Oracle HTTP Server および OC4J での SSL の構成 2-36
SSL の一般的な問題と解決策 2-37
その他のセキュリティに関する考慮事項 2-38
サーブレットのベスト・プラクティスのサマリー 2-39
ベスト・プラクティス : 一般的なコーディング 2-39
ベスト・プラクティス : セッション 2-39
ベスト・プラクティス : 構成 2-40

3 サーブレット・フィルタとイベント・リスナー

サーブレット・フィルタ 3-2
サーブレット・フィルタの概要 3-2
サーブレット・コンテナによるフィルタの起動 3-2
転送またはインクルード・ターゲットのフィルタ 3-3
フィルタの例 3-4
イベント・リスナー 3-12
イベント・カテゴリとイベント・リスナー・インタフェース 3-12
代表的なイベント・リスナーの使用例 3-13
イベント・リスナーの宣言と起動 3-13
イベント・リスナーのコーディングとデプロイのガイドライン 3-14
イベント・リスナーのメソッドと関連クラス 3-14
イベント・リスナーの例 3-16

4 サーブレットからの JDBC および EJB コール

サーブレットでの JDBC の使用 4-2
データベース問合せサーブレット 4-2
データベース問合せサーブレットのデプロイおよびテスト 4-5

サーブレットからの EJB コール 4-6
サーブレットおよび EJB の概要 4-7
EJB のローカル・ルックアップ 4-8
同一アプリケーション内の EJB リモート・ルックアップ 4-14
アプリケーション外部の EJB リモート・ルックアップ 4-19

5 デプロイおよび構成の概要

OC4J のデプロイおよび構成の一般的な概要 5-2
概要 : OC4J スタンドアロンと Oracle Application Server 環境 5-2
OC4J のデプロイ方法の概要 5-4
Oracle のデプロイ・ツールの使用とエキスパート・モードの使用 5-5
構成ファイルの概要 5-5
OC4J および J2EE 構成ファイルの概要 5-6
OC4J のトップレベルのサーバー構成ファイル : server.xml 5-9
OC4J および J2EE アプリケーション・ディスクリプタ 5-11
OC4J および J2EE の Web ディスクリプタ 5-15
OC4J Web サイト・ディスクリプタ 5-18
例 : Web サイト・ディスクリプタへのマッピングと Web サイト・ディスクリプタからのマッピング 5-19
アプリケーションのパッケージング 5-20
J2EE アプリケーション構造 5-20
EAR ファイルおよび WAR ファイル構造 5-21
OC4J スタンドアロンへのデプロイ方法 5-22
管理ユーザーおよびパスワードの設定 5-23
OC4J スタンドアロンの起動および停止 5-24
OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーション 5-24
EAR ファイルの OC4J スタンドアロンへのデプロイ 5-26
OC4J スタンドアロンでの、J2EE アプリケーション構造へのファイルのデプロイ 5-30
OC4J スタンドアロンへの独立した WAR ファイルのデプロイ 5-31
OC4J スタンドアロンでの Web アプリケーション・ディレクトリ構造へのファイルのデプロイ 5-33
OC4J スタンドアロンにおけるアプリケーションのアンデプロイまたは再デプロイ 5-35
Oracle Application Server での OC4J のデプロイ 5-37
Oracle Application Server における OC4J のデプロイおよび構成の概要 5-38
Oracle Application Server での OC4J のデフォルト Web アプリケーション 5-39
Oracle Application Server でのアプリケーションのアンデプロイおよび再デプロイ 5-39

6 構成ファイルの説明

global-web-application.xml および orion-web.xml の構成 6-2
global-web-application.xml および orion-web.xml の要素の説明 6-2
global-web-application.xml および orion-web.xml の DTD 6-13
global-web-application.xml および orion-web.xml の階層表現 6-17
サンプルの global-web-application.xml の設定 6-18
Web サイトの XML ファイルの構成 6-19
Web サイト XML ファイルの要素の説明 6-19
Web サイトの XML ファイルの DTD 6-27

Web サイトの XML ファイルの階層表現 6-29
サンプルの default-web-site.xml ファイル 6-29

7 Enterprise Manager を使用した構成

Oracle Enterprise Manager 10*g* での Web モジュールの構成 7-2
Application Server Control コンソールのページの説明 7-2
Application Server Control コンソールの OC4J ホームページ 7-3
Application Server Control コンソールの OC4J の「アプリケーション」ページ 7-3
Application Server Control コンソールの「アプリケーションのデプロイ」ページ 7-4
Application Server Control コンソールの「Web アプリケーションのデプロイ」ページ 7-5
Application Server Control コンソールの OC4J の「管理」ページ 7-6
Application Server Control コンソールの「Web サイト・プロパティ」ページ 7-7
Application Server Control コンソールの「Web モジュール」ページ 7-8
Application Server Control コンソールの Web モジュールの「プロパティ」ページ 7-9
Application Server Control コンソールの Web モジュールの「マッピング」ページ 7-11
Application Server Control コンソールの Web モジュールの「フィルタ処理とチェーン」ページ 7-13
Application Server Control コンソールの Web モジュールの「環境」ページ 7-14
Application Server Control コンソールの Web モジュールの「拡張プロパティ」ページ 7-15

A オープン・ソース・フレームワークおよびユーティリティ

OC4J における Jakarta Struts の構成および使用 A-2
Jakarta Struts の概要 A-2
Struts バイナリ配布版のダウンロード A-2
Struts バイナリ配布版の解凍 A-3
Struts ドキュメントのインストールおよびアクセス A-3
Struts サンプル Web アプリケーションのインストール A-4
Struts フレームワークを使用したユーザー・アプリケーションのデプロイ A-5
OC4J における Jakarta log4j の構成および使用 A-7
Jakarta log4j の概要 A-7
log4j バイナリ配布版のダウンロード A-7
log4j バイナリ配布版の解凍 A-8
log4j ライブラリのインストール A-8
log4j 構成ファイルの使用 A-10
log4j デバッグ・モードの使用可能化 A-12

B サード・パーティ・ライセンス

Apache HTTP Server B-2
The Apache Software License B-2

索引

はじめに

このマニュアルでは、Sun 社主体の業界の協議により指定された、Java サーブレット・テクノロジの Oracle 実装について説明します。標準機能をまとめ、Oracle 実装の詳細と付加価値機能についても説明します。この説明には基本サーブレット、データ・アクセス・サーブレット、サーブレット・フィルタおよびイベント・リスナーも含まれます。

サーブレット・テクノロジは標準 Java 2 Enterprise Edition（J2EE）のコンポーネントです。Oracle Application Server の J2EE コンポーネントは、Oracle Application Server Containers for J2EE（OC4J）と呼ばれています。

Oracle Application Server 10*g* リリース 2（10.1.2）の OC4J サーブレット・コンテナは Sun 社の Java サーブレット仕様、バージョン 2.3 の完全実装です。

この章には、次の項が含まれます。

- 対象読者
- ドキュメントのアクセシビリティについて
- 関連ドキュメント
- 表記規則
- サポートおよびサービス

対象読者

このマニュアルは、サーブレットおよび（場合によっては）JavaServer Pages（JSP）を使用した Web アプリケーションを作成する、J2EE 開発者向けに作成されています。このマニュアルは、OC4J サーブレット・コンテナに関して必要となる基本情報を提供します。ここには、サーブレットのプログラミングに関する一般的な説明や、Java サーブレット API に関する詳細な説明は含まれていません。

Sun 社により提供される、Java サーブレット仕様の最新バージョンに関する知識が必要です。特に、分散 Web アプリケーションを開発していて、2 つ以上の Java Virtual Machine（JVM）の下で稼働している複数のサーバーにセッションをレプリケート可能な場合に、これらの知識が必要となります。

主に JavaServer Pages を使用するアプリケーションを開発する場合は、『Oracle Application Server Containers for J2EE JavaServer Pages 開発者ガイド』を参照してください。

ドキュメントのアクセシビリティについて

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

ドキュメント内のサンプル・コードのアクセシビリティについて

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

外部 Web サイトのドキュメントのアクセシビリティについて

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

関連ドキュメント

詳細は、次の Oracle ドキュメントを参照してください。

OC4J の追加ドキュメント :

- 『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』

  このマニュアルは、OC4J の概要と一般情報、サーブレット、JSP ページおよび EJB の入門に関する章、構成およびデプロイの一般手順を提供します。

- 『Oracle Application Server Containers for J2EE スタンドアロン・ユーザーズ・ガイド』

  このバージョンのユーザーズ・ガイドはスタンドアロン・バージョンの OC4J 専用で、OC4J からスタンドアロン・バージョンをダウンロードする際に入手できます。OC4J スタンドアロンは開発環境で使用するもので、本番環境で使用することはあまりありません。

- 『Oracle Application Server Containers for J2EE JavaServer Pages 開発者ガイド』

  このマニュアルは、OC4J での JavaServer Pages の開発と実装およびコンテナに関する情報を提供します。コマンドライン・トランスレータや OC4J 固有の構成パラメータなどの Oracle の機能についても説明しています。

- 『Oracle Application Server Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』

このマニュアルは、タグ・ライブラリ、JavaBeans および OC4J で提供される他の Java ユーティリティに関する概念的な情報、詳細な構文および使用に関する情報を提供します。他の Oracle 製品グループのタグ・ライブラリの要約も示します。

- 『Oracle Application Server Containers for J2EE サービス・ガイド』

  このマニュアルは、JTA、JNDI、JMS、JAAS および Oracle Application Server Java Object Cache など、OC4J で提供される標準的な Java サービスに関する情報を提供します。

- 『Oracle Application Server Containers for J2EE セキュリティ・ガイド』

  このマニュアル（『Oracle Application Server セキュリティ・ガイド』と混同しないでください）は、特に OC4J に関するセキュリティ機能および実装について説明します。JAAS（Java Authentication and Authorization Service）およびその他の Java セキュリティ・テクノロジに関する情報も提供します。

- 『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』

  このマニュアルは、OC4J での Enterprise JavaBeans の開発と実装およびコンテナに関する情報を提供します。

Oracle Application Server TopLink のドキュメント :

- 『Oracle Application Server TopLink スタート・ガイド』
- 『Oracle Application Server TopLink Mapping Workbench ユーザーズ・ガイド』
- 『Oracle Application Server TopLink アプリケーション開発者ガイド』

Oracle Database の Java 関連ドキュメント :

- 『Oracle Database Java 開発者ガイド』
- 『Oracle Database JDBC 開発者ガイドおよびリファレンス』
- 『Oracle Database JPublisher ユーザーズ・ガイド』

Oracle Application Server の追加ドキュメント :

- 『Oracle Application Server 管理者ガイド』
- 『Oracle Application Server セキュリティ・ガイド』
- 『Oracle Application Server Certificate Authority 管理者ガイド』
- 『Oracle Application Server パフォーマンス・ガイド』
- 『Oracle Enterprise Manager 概要』
- 『Oracle HTTP Server 管理者ガイド』
- 『Oracle Application Server グローバリゼーション・サポート・ガイド』
- 『Oracle Application Server Web Cache 管理者ガイド』
- 『Oracle Application Server Web Services 開発者ガイド』
- 『Oracle Application Server 高可用性ガイド』
- 『Oracle Application Server アップグレードおよび互換性ガイド』

Oracle JDeveloper のドキュメント :

- Oracle JDeveloper オンライン・ヘルプ
- OTN（Oracle Technology Network）上の Oracle JDeveloper マニュアル

  http://www.oracle.com/technology/products/jdev/index.html

Oracle Database の追加ドキュメント :

- 『Oracle XML Developer's Kit プログラマーズ・ガイド』
- 『Oracle Database XML C API リファレンス』

- 『Oracle Database アプリケーション開発者ガイド - 基礎編』
- 『Oracle Database PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』
- 『Oracle Database PL/SQL ユーザーズ・ガイドおよびリファレンス』
- 『Oracle Database SQL リファレンス』
- 『Oracle Database Net Services 管理者ガイド』
- 『Oracle Database Advanced Security 管理者ガイド』
- 『Oracle Database リファレンス』

Java サーブレットおよび JavaServer Pages 用の次の OTN Web サイトも使用できます。

http://www.oracle.com/technology/tech/java/servlets/index.html

サーブレットの詳細は、次の場所から入手可能な Java サーブレット仕様、バージョン 2.3 を参照してください。

http://jcp.org/aboutJava/communityprocess/first/jsr053/index.html

Sun 社のドキュメントは次のとおりです。

- Java サーブレット（最新仕様を含む）に関する Web サイト :

  http://java.sun.com/products/servlet/index.jsp

- JavaServer Pages（最新仕様を含む）に関する Web サイト :

  http://java.sun.com/products/jsp/index.jsp

- サーブレット API の Javadoc:

  http://java.sun.com/products/servlet/2.3/javadoc/index.html

表記規則

本文では、次の表記規則を使用します。

規則	意味
太字	太字は、操作に関連するグラフィカル・ユーザー・インタフェース要素、または本文中で定義されている用語および用語集に記載されている用語を示します。
イタリック体	イタリックは、特定の値を指定するプレースホルダ変数を示します。
`固定幅フォント`	固定幅フォントは、パラグラフ内のコマンド、URL、例に記載されているコード、画面に表示されるテキスト、または入力するテキストを示します。

サポートおよびサービス

次の各項に、各サービスに接続するための URL を記載します。

オラクル社カスタマ・サポート・センター

オラクル製品サポートの購入方法、およびオラクル社カスタマ・サポート・センターへの連絡方法の詳細は、次の URL を参照してください。

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

製品マニュアル

製品のマニュアルは、次の URL にあります。

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

研修およびトレーニング

研修に関する情報とスケジュールは、次の URL で入手できます。

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

その他の情報

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

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

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

1

サーブレットの概要

Oracle Application Server Containers for J2EE（OC4J）を使用すると、標準 J2EE 準拠のアプリケーションを開発およびデプロイできます。アプリケーションは、標準 Enterprise Archive（EAR）デプロイメント・ファイルにパッケージされており、Web モジュールをデプロイするための標準 Web Archive（WAR）ファイルと、アプリケーション内の Enterprise JavaBeans（EJB）およびアプリケーション・クライアント・モジュール用の Java archive（JAR）ファイルが含まれます。

Oracle Application Server 10*g* リリース 2（10.1.2）では、OC4J は、Java 2 Platform Enterprise Edition バージョン 1.3 に準拠し、OC4J コンテナ内では Sun 社の Java サーブレット仕様、バージョン 2.3 に完全に準拠しています。（特に記載のないかぎり、このマニュアルではサーブレット仕様はこのバージョンを指します）。

Web アプリケーションの構築およびデプロイ方法は OC4J におけるサーブレット開発を理解する上で、最も重要な概念です。サーブレットを初めて使用する場合は、第 2 章「サーブレットの開発」を参照してください。OC4J の開発環境を初めて使用する場合は、第 5 章「デプロイおよび構成の概要」を参照して、OC4J におけるアプリケーションのデプロイ方法を理解してください。

この章では、Java サーブレットの概要を説明し、基本サーブレットの例を示します。また、サーバー・サイドのプログラミングに関する様々な課題に対処するために J2EE アプリケーションでサーブレットを使用する方法を、簡単に説明します。

この章には、次の項が含まれます。

- サーブレットの概要
- 初めてのサーブレット作成の例

注意： サンプルのサーブレット・アプリケーションは OC4J のデモに含まれています。デモは、Oracle Technology Network の次のサイトから入手可能です（OTN のメンバーシップ登録が必要ですが、登録は無料です）。

`http://www.oracle.com/technology/tech/java/oc4j/demos/`

サーブレットの概要

次の項では、サーブレット・テクノロジの概要を説明します。

- サーブレット・テクノロジのレビュー
- サーブレットの利点
- サーブレット・インタフェースとリクエスト・オブジェクトおよびレスポンス・オブジェクト
- サーブレットおよびサーブレット・コンテナ
- サーブレット・セッションの概要
- サーブレット・コンテキストの概要
- サーブレット構成オブジェクトの概要
- サーブレット・フィルタの概要
- イベント・リスナーの概要
- JSP ページおよびその他の J2EE コンポーネント・タイプ

注意：「Web モジュール」、「Web アプリケーション」という用語はほとんどの場合が同義で、両方ともこのマニュアル全体を通して使用されています。区別されている場合、通常「Web モジュール」は独立したアプリケーションを構成するかしないかに関係なく単一のコンポーネントを示し、「Web アプリケーション」は複数のモジュールまたはコンポーネントから構成されている可能性のある動作中のアプリケーションを示します。

サーブレット・テクノロジのレビュー

近年、サーブレット・テクノロジは、動的 Web ページを使用して Web サーバー機能を拡張する強力な手段として登場しました。サーブレットは、Web サーバーで稼働する Java プログラムです。一方、アプレットはクライアント・ブラウザで稼働します。通常、サーブレットは、ブラウザから HTTP リクエストを取得し、データベース問合せなどによって動的コンテンツを生成し、HTTP レスポンスをブラウザに返します。または、直接別のアプリケーション・コンポーネントからアクセスされるか、または別のコンポーネントに出力を送信する場合もあります。ほとんどのサーブレットは HTML テキストを生成しますが、かわりに XML を生成してデータをカプセル化するサーブレットもあります。

具体的には、サーブレットは、OC4J などの J2EE アプリケーション・サーバーで稼働します。サーブレットは、JavaServer Pages（JSP）および EJB と同様、J2EE アプリケーションの主要なアプリケーション・コンポーネント・タイプの 1 つです。JSP や EJB も、サーバー・サイド J2EE コンポーネント・タイプです。サーブレットは、アプレット（Java 2 Platform, Standard Edition 仕様の一部）などのクライアント・サイド・コンポーネントやアプリケーションのクライアント・プログラムとともに使用されます。アプリケーションは、任意の数のコンポーネントで構成できます。

サーブレット以前は、動的コンテンツの作成に、Common Gateway Interface（CGI）が使用されていました。CGI プログラムは、Perl などの言語で記述され、Web アプリケーションによって Web サーバーを介してコールされます。しかし、CGI はアーキテクチャやスケーラビリティに制限があるため、理想的なプログラムでないことが実証されました。

サーブレットの利点

Java レルムでは、サーブレット・テクノロジは、データベースにアクセスするアプリケーションなどのサーバー集中型アプリケーションの場合、アプレット・テクノロジより利点があります。サーバーで実行することの利点の 1 つは、サーバーが通常多くのリソースを持つ堅牢なマシンであるため、プログラムがよりスケーラブルになることです。サーバーで実行すると、より直接的にデータにアクセスすることもできます。サーブレットが稼働中の Web サーバーは、アクセス対象データと同様に、ネットワーク・ファイアウォールの内側にあります。

サーブレットのプログラミングには、サーバー・サイドの Web アプリケーション開発における初期のモデルと比べると、次のような多くの利点もあります。

- サーブレットは、サーバー・サイド・インクルードや CGI スクリプトなど、動的 HTML 生成を行うための初期のテクノロジより優れています。一度メモリーにロードされたサーブレットは、軽量なシングル・スレッド上で実行できます。CGI スクリプトは、リクエストごとに別のプロセスにロードする必要があります。
- サーブレット・テクノロジには、スケーラビリティの向上に加えて、セキュリティ、堅牢さ、オブジェクト指向およびプラットフォームへの非依存など、よく知られた Java の利点が備わっています。
- サーブレットは、Java Database Connectivity（JDBC）などの Java 言語および Java の標準 API と完全に統合されています。
- サーブレットは、完全に J2EE フレームワークに統合されています。J2EE フレームワークには、作成する Web アプリケーションに使用可能なサービスの拡張セットが用意されています。たとえば、コンポーネントのネーミングおよびルックアップに使用する Java Naming and Directory Interface（JNDI）、トランザクションの管理に使用する Java Transaction API（JTA）、セキュリティに使用する Java Authentication and Authorization Service（JAAS）、分散アプリケーションに使用する Remote Method Invocation（RMI）および Java Message Service（JMS）などです。次の Web サイトには、J2EE フレームワークおよびサービスに関する情報が記載されています。

  http://java.sun.com/j2ee/docs.html

- サーブレットでは、スレッド・モデルに基づき、単一のサーブレット・インスタンスまたは複数のサーブレット・インスタンスを使用して同時リクエストを処理します。各サーブレットには明確に定義されたライフ・サイクルが存在します。また、オプションで、OC4J を起動するときにサーブレットをロードできます。そのため、初回のユーザー・リクエストに応じるのではなく、事前にすべての初期化が行われます。2-6 ページの「サーブレットの事前ロード」を参照してください。
- サーブレットのリクエストおよびレスポンス・オブジェクトを使用すると、HTTP リクエストの処理や、テキストとデータのクライアントへの返信が容易に行えます。

サーブレットは、Java プログラミング言語で記述されているため、Java Virtual Machine（JVM）を持つすべてのプラットフォーム、およびサーブレットをサポートしているすべての Web サーバーでサポートされます。サーブレットは、再コンパイルせずに別のプラットフォーム上で使用できます。サーブレットをグラフィックス、サウンドおよび他のデータなどの関連するファイルとともにパッケージングして、完全な Web アプリケーションを作成することができます。パッケージングによって、アプリケーションの開発とデプロイが簡素化されます。

さらに、サーブレット・ベースのアプリケーションを他の Web サーバーから OC4J に簡単に移植できます。J2EE 準拠の Web ブラウザ用に開発されたアプリケーションの場合、移植の手間は最小限ですみます。

サーブレット・インタフェースとリクエスト・オブジェクトおよびレスポンス・オブジェクト

Java サーブレットは、当然 `javax.servlet.Servlet` インタフェースを実装しています。このインタフェースは、サーブレットの初期化、リクエストの処理、サーブレットの構成情報や他の基本情報の取得、およびサーブレット・インスタンスの終了などを行うメソッドを指定します。

Web アプリケーションの場合は、`javax.servlet.http.HttpServlet` 抽象クラスを拡張して、`Servlet` インタフェースを実装できます。（一方、プロトコルに依存しないサーブレットの場合は、`javax.servlet.GenericServlet` クラスを拡張できます。）`HttpServlet` クラスには、次のメソッドが含まれています。

- `init(...)`: サーブレットを初期化します。
- `destroy(...)`: サーブレットを終了します。
- `doGet(...)`: HTTP `GET` リクエストを実行します。

- doPost(...): HTTP POST リクエストを実行します。
- doPut(...): HTTP PUT リクエストを実行します。
- doDelete(...): HTTP DELETE リクエストを実行します。
- service(...): HTTP リクエストを受信し、デフォルトで、そのリクエストを適切な do*XXX*() メソッドに送信します。
- getServletInfo(...): サーブレットに関する情報を取得します。

HttpServlet を拡張するサーブレット・クラスは、必要に応じてこれらのメソッドの一部またはすべてを実装し、元の実装を任意にオーバーライドしてリクエストを処理し、レスポンスを返します。たとえば、ほとんどのサーブレットは、doGet() メソッド、doPost() メソッドまたはこれらの両方をオーバーライドして、HTTP GET および POST リクエストを処理します。

各メソッドは、入力として HttpServletRequest インスタンス（javax.servlet.http.HttpServletRequest インタフェースを実装するクラスのインスタンス）と HttpServletResponse インスタンス（javax.servlet.http.HttpServletResponse インタフェースを実装するクラスのインスタンス）を取得します。

HttpServletRequest インスタンスは、サーブレットに HTTP リクエストに関する情報を提供します。情報には、リクエスト・パラメータ名と値、リクエストを作成したリモート・ホスト名およびリクエストを受信したサーバー名などが含まれます。HttpServletResponse インスタンスは、レスポンスの送信時に、コンテンツ長と MIME タイプの指定および出力ストリームの指定など、HTTP 固有の機能を提供します。

サーブレットおよびサーブレット・コンテナ

Java クライアント・プログラムとは異なり、サーブレットには静的な main() メソッドがありません。したがって、サーブレットは、外部コンテナの制御下で実行する必要があります。

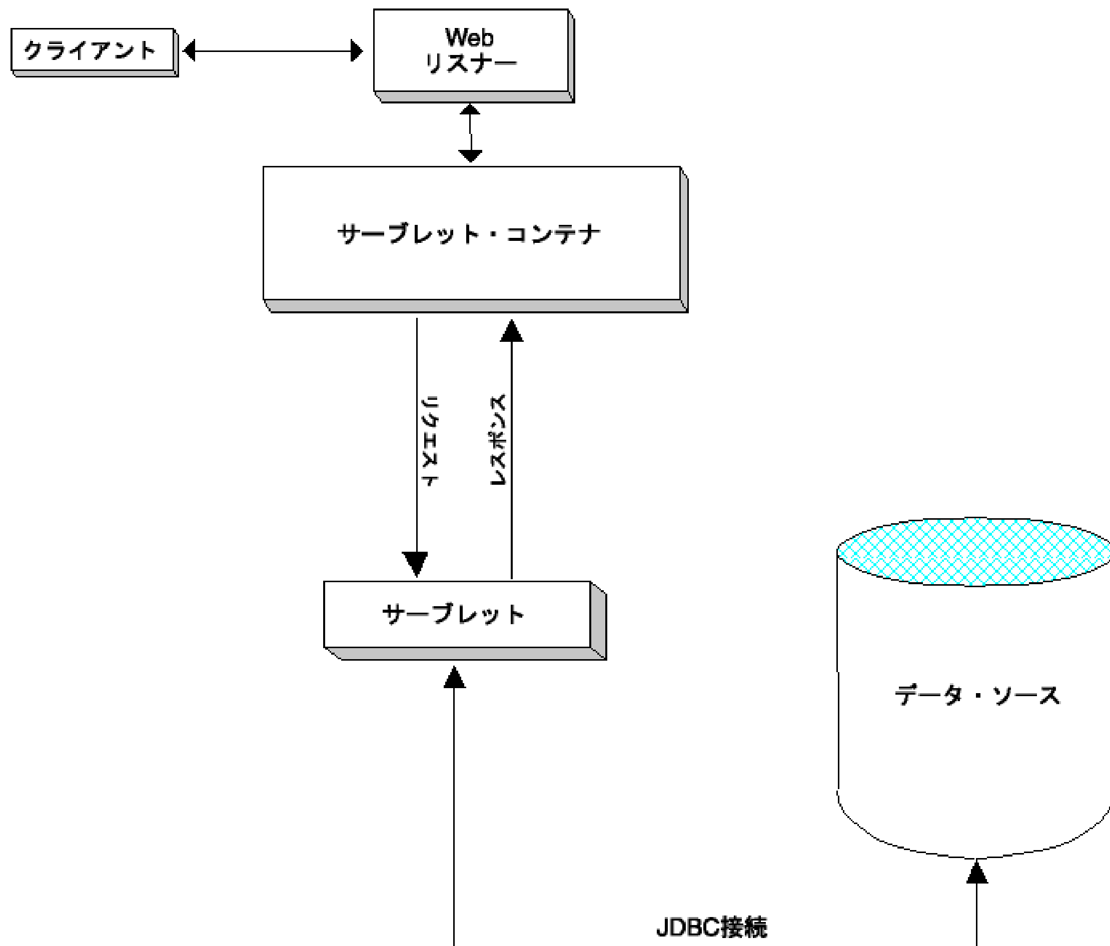
サーブレット・コンテナは、サーブレット・エンジンとも呼ばれ、サーブレットの実行と管理を行います。サーブレット・コンテナは、サーブレット・メソッドをコールし、サーブレットの実行中に必要なサービスを提供します。サーブレット・コンテナは通常 Java で作成し、Web サーバーに組み込むか（Web サーバーも Java で作成されている場合）、Web サーバーに関連付けて使用します。OC4J には、完全な標準準拠のサーブレット・コンテナが組み込まれます。

サーブレット・コンテナにより、サーブレットは、ヘッダーやパラメータなど HTTP リクエストのプロパティに容易にアクセスできます。URL による指定などでサーブレットがコールされると、Web サーバーは、HTTP リクエストをサーブレット・コンテナに渡します。コンテナは、次に、そのリクエストをサーブレットに渡します。サーブレットの管理で、サーブレット・コンテナは、次のタスクを実行します。

- サーブレットのインスタンスを作成し、その init() メソッドをコールして初期化します。
- リクエスト・オブジェクトを作成して、サーブレットに渡します。リクエストには、特に次のものが含まれます。
  - クライアントからの任意の HTTP ヘッダー
  - クライアントから渡されたパラメータと値（URL 内の問合せ文字列の名前および値など）
  - サーブレット・リクエストの完全な URI
- サーブレットのレスポンス・オブジェクトを作成します。
- サーブレットの service() メソッドを起動します。HTTP サーブレットの場合、汎用サービス・メソッドは、通常 HttpServlet クラスでオーバーライドされます。サービス・メソッドは、リクエスト内の HTTP ヘッダーに応じて（GET または POST）、サーブレットの doGet() または doPost() メソッドにリクエストを送ります。
- サーブレットの destroy() メソッドをコールして、必要に応じてそのサーブレットをガベージ・コレクションの対象とするために破棄します。（パフォーマンス上の理由から、通常サーブレット・コンテナはサーブレット・インスタンスをメモリー内に保持して再利用します。タスクが終了するたびにインスタンスを破棄することはありません。破棄されるのは、Web サーバー・シャットダウンなどの使用頻度が低いイベントの場合のみです。）

図 1-1 に、サーブレットがサーブレット・コンテナおよび Web ブラウザなどのクライアントとどのように関連するかを示します。Web リスナーが Oracle HTTP Server（powered by Apache）の場合、OC4J サーブレット・コンテナへの接続は、mod_oc4j モジュール経由になります。詳細は、『Oracle HTTP Server 管理者ガイド』を参照してください。

図 1-1 サーブレットおよびサーブレット・コンテナ



サーブレット・セッションの概要

サーブレットでは、HTTP セッションを使用して、各 HTTP リクエストの発信元ユーザーを追跡します。そのため、単一ユーザーからの複数のリクエストは、ステートフルな方法で管理できます。サーブレットのセッション・トラッキングは、CGI など以前のテクノロジで使用されたセッション・トラッキングと本質的に類似しています。

この項では、サーブレット・セッションの概要を説明します。詳細および例は、2-24 ページの「サーブレットのセッション」を参照してください。

セッション・トラッキングの概要

サーブレットは、クライアント・セッションとサーバー・セッションの同期を維持する便利な方法を備えています。これにより、ステートフルなサーブレットは、クライアントによるブラウズ・セッションの間中、サーバー上のセッションの状態を維持できます。

OC4J は次のセッション・トラッキング方法をサポートしています。詳細は、2-25 ページの「セッション・トラッキング」を参照してください。

- Cookie

  サーブレット・コンテナは、Cookie をクライアントに送信します。クライアントはその Cookie を HTTP リクエスト時にサーバーに返します。このプロセスでは、リクエストは Cookie が示すセッション ID に関連付けられます。これが最もよく使用される方法で、サーブレット仕様に準拠しているすべてのサーブレット・コンテナでサポートされます。

- URL リライティング

  Cookie を使用するかわりに、サーブレットは、レスポンス・オブジェクトの `encodeURL()` メソッドまたはリダイレクト用の `encodeRedirectURL()` メソッドをコールして、セッション ID を各リクエストの URL パスに追加することもできます。このプロセスでは、リクエストをセッションに関連付けることができます。これが、クライアントが Cookie を受け入れない場合に最もよく使用される方法です。

HttpSession インタフェースの概要

標準サーブレット API では、各クライアント・セッションは、`javax.servlet.http.HttpSession` インタフェースを実装するクラスのインスタンスによって表されます。サーブレットは、この `HttpSession` オブジェクト内のセッションに関する情報を設定および取得できます。このオブジェクトはアプリケーション・レベルのスコープを持つ必要があります。

サーブレットは、`HttpServletRequest` オブジェクトの `getSession()` メソッドを使用して、ユーザーの `HttpSession` オブジェクトを取得または作成します。このメソッドは、ブール引数を取得して、オブジェクト・セッションがアプリケーション内に存在しない場合に、クライアントに対して新規セッション・オブジェクトを作成する必要があるかどうかを指定します。

詳細は、2-27 ページの「HttpSession インタフェースの機能」を参照してください。

サーブレット・コンテキストの概要

サーブレット・コンテキストは、単一 JVM 内の Web アプリケーションの全インスタンス（つまり、Web アプリケーションに含まれる全サーブレットと JSP ページ・インスタンス）に関する情報を保持するために使用します。任意の JVM 内で稼働する Web アプリケーションごとに 1 つのサーブレット・コンテキストがあります。これは常に 1 対 1 の設定です。サーブレット・コンテキストは、特定のアプリケーションのコンテナと考えられます。

サーブレット・コンテキストの基本

サーブレット・コンテキストは、`javax.servlet.ServletContext` インタフェースを実装するクラスのインスタンスです。このクラスは、サーブレットをサポートする Web サーバーに含まれています。

`ServletContext` オブジェクトは、サーブレット環境に関する情報（サーバー名など）を提供し、単一 JVM 内のグループのサーブレット間でリソースを共有できるようにします。（同時に複数の JVM をサポートしているサーブレット・コンテナの場合、リソース共有の実装は一様ではありません。）

サーブレット・コンテキストは、アプリケーションの実行インスタンスに対してスコープを設定します。このメカニズムによって、各アプリケーションは別個のクラスローダーからロードされ、その実行時オブジェクトは他のアプリケーションのオブジェクトとは明確に区別されます。特に、`ServletContext` オブジェクトはアプリケーションごとに異なります。これは、各 `HttpSession` オブジェクトがそのアプリケーションのユーザーごとに異なるのと同じです。

サーブレット仕様のバージョン 2.2 から、ほとんどの実装で単一ホスト内に複数のサーブレット・コンテキストを設定できます。そのため、各 Web アプリケーションが独自のサーブレット・コンテキストを持つことができます（以前の実装では、任意のホストに設定できるサーブレット・コンテキストは 1 つのみでした）。

サーブレット・コンテキストの取得方法

サーブレットの構成オブジェクトの getServletContext() メソッドを使用して、サーブレット・コンテキストを取得します。1-8 ページの「サーブレット構成オブジェクトの概要」を参照してください。

サーブレット・コンテキスト・メソッド

ServletContext インタフェースは、サーブレットにそのサーブレットを実行するサーブレット・コンテナとの通信を許可するメソッドを指定します。この方法によって、サーブレットはアプリケーション・レベルの環境と状態情報を取得できます。ServletContext では、次のようなメソッドが指定されます。詳細は、次の場所で Sun 社の Javadoc を参照してください。

http://java.sun.com/products/servlet/2.3/javadoc/index.html

- `void setAttribute(String name, Object value)`

  このメソッドは、指定したオブジェクトをサーブレット・コンテキストの指定した属性名にバインドします。属性を使用すると、サーブレット・コンテナは、サーブレットに情報を提供できます。属性を使用しない場合、ServletContext インタフェースを介して情報を提供できません。

  > **注意：** サーブレット・コンテキストの場合、setAttribute() は、ローカル操作のみです。クラスタ内の他の JVM への配布を目的としていません。（これは、サーブレット仕様に従っています）。

- `Object getAttribute(String name)`

  このメソッドは、指定した名前を持つ属性を戻します。その名前の属性がない場合は、null を戻します。属性は、java.lang.Object インスタンスとして戻されます。

- `java.util.Enumeration getAttributeNames()`

  このメソッドは、サーブレット・コンテキストで使用可能な全属性の名前が含まれた java.util.Enumeration インスタンスを戻します。

- `void removeAttribute(String attrname)`

  このメソッドは、指定した属性をサーブレット・コンテキストから削除します。

- `String getInitParameter(String name)`

  このメソッドは、指定したコンテキスト全体の初期化パラメータの値を示す文字列を戻します。その名前のパラメータがない場合は、null を戻します。これによって、このサーブレット・コンテキストに関連付けられた Web アプリケーションに有用な構成情報にアクセスできます。

- `Enumeration getInitParameterNames()`

  このメソッドは、サーブレット・コンテキストの初期化パラメータ名が含まれた java.util.Enumeration インスタンスを戻します。

- `RequestDispatcher getNamedDispatcher(String name)`

  このメソッドは、指定したサーブレットのラッパーとして動作する javax.servlet.RequestDispatcher インスタンスを戻します。

- `RequestDispatcher getRequestDispatcher(String path)`

  このメソッドは、指定したパスにあるリソースのラッパーとして動作する javax.servlet.RequestDispatcher インスタンスを戻します。

- `String getRealPath(String path)`

  このメソッドは、指定した仮想パスに対する実際のパスを文字列で戻します。

- `URL getResource(String path)`

  このメソッドは、指定したパスにマップされたリソースに対する URL が含まれた `java.net.URL` インスタンスを戻します。

- `String getServerInfo()`

  このメソッドは、サーブレット・コンテナの名前とバージョンを戻します。

- `String getServletContextName()`

  このメソッドは、`web.xml` ファイルの `<display-name>` 要素に基づいて、サーブレット・コンテキストが関連付けられている Web アプリケーションの名前を戻します。

サーブレット構成オブジェクトの概要

サーブレット構成オブジェクトには、サーブレットの初期化および起動パラメータが含まれます。このオブジェクトは、`javax.servlet.ServletConfig` インタフェースを実装するクラスのインスタンスです。このクラスは、任意の J2EE 準拠の Web サーバーにより提供されます。

サーブレットの `getServletConfig()` メソッドをコールすることで、サーブレットのサーブレット構成オブジェクトを取得できます。このメソッドは `javax.servlet.Servlet` インタフェースで指定されており、`javax.servlet.http.HttpServlet` クラスでデフォルトが実装されています。

`ServletConfig` インタフェースは、次のメソッドを指定します。

- `ServletContext getServletContext()`

  アプリケーションのサーブレット・コンテキストを取得します。1-6 ページの「サーブレット・コンテキストの概要」を参照してください。

- `String getServletName()`

  サーブレットの名前を取得します。

- `Enumeration getInitParameterNames()`

  サーブレットの初期化パラメータの名前がある場合、その名前を取得します。名前は、`String` オブジェクトの `java.util.Enumeration` インスタンスに戻されます。(初期化パラメータがない場合、`Enumeration` インスタンスは空です。)

- `String getInitParameter(String name)`

  これは、指定した初期化パラメータの値を含む `String` オブジェクトを戻します。その名前のパラメータがない場合は、`null` を戻します。

サーブレット・フィルタの概要

リクエスト・オブジェクト(`HttpServletRequest` を実装するクラスのインスタンス)とレスポンス・オブジェクト(`HttpServletResponse` を実装するクラスのインスタンス)は、通常、サーブレット・コンテナとサーブレットとの間で直接受渡しされます。

ただし、サーブレット仕様では、サーブレット・フィルタを使用できます。このフィルタは、サーバーで実行され、特別なリクエストやレスポンスを処理するためにサーブレット(またはサーブレット・グループ)とサーブレット・コンテナの中間に置くことができる Java プログラムです。

サーブレットより先に起動するフィルタまたは一連のフィルタがある場合、これらのフィルタは、リクエスト・オブジェクトとレスポンス・オブジェクトをパラメータとして使用してコンテナからコールされます。フィルタは、`doChain()` メソッドを使用して、これらのオブジェクト(変更されている場合もある)を一連のフィルタ内の次のオブジェクトに渡します。新規オブジェクトを作成してそれを渡すこともあります。

詳細は、3-2 ページの「サーブレット・フィルタ」を参照してください。

イベント・リスナーの概要

サーブレット仕様には、イベント・リスナーを使用して Web アプリケーションの主要イベントを追跡する機能が追加されました。この機能を使用すると、イベントの状態に基づいたリソース管理と自動処理をより効果的に実行できます。

リスナー・クラスを作成する場合、サーブレット・コンテキストのライフ・サイクル・イベント、サーブレット・コンテキスト属性の変更、HTTP セッションのライフ・サイクル・イベント、および HTTP セッション属性の変更に対して標準インタフェースを実装できます。リスナー・クラスには、必要に応じて、これらのインタフェースの 1 つ、複数またはすべてを実装できます。

イベント・リスナー・クラスは、`web.xml` デプロイメント・ディスクリプタで宣言され、アプリケーションの起動時に起動され、登録されます。イベントが発生すると、サーブレット・コンテナは適切なリスナー・メソッドをコールします。

詳細は、3-12 ページの「イベント・リスナー」を参照してください。

JSP ページおよびその他の J2EE コンポーネント・タイプ

アプリケーションには、サーブレットに加えて、JSP ページや EJB などのサーバー・サイド・コンポーネントを組み込むことができます。特に、サーブレットは、Web アプリケーションの JSP ページと組み合せて使用するのが一般的です。サーブレットは、OC4J サーブレット・コンテナによって管理されます。一方、EJB は OC4J EJB コンテナ、JSP ページは OC4J JSP コンテナによって管理されます。これらのコンテナは OC4J の中核を形成します。

JSP ページにもサーブレット・コンテナが含まれています。これは、JSP コンテナ自体がサーブレットであり、サーブレット・コンテナによって実行されるためです。JSP コンテナは、JSP ページをページ実装クラスに変換します。このクラスは JSP コンテナによって実行され、基本的にはサーブレットでもあります。

注意： このマニュアルでサーブレットに適用する機能について説明している部分はすべて、特に記載がないかぎり JSP ページにも同様に適用されます。

JSP ページと EJB の詳細は、次のマニュアルを参照してください。

- 『Oracle Application Server Containers for J2EE JavaServer Pages 開発者ガイド』
- 『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』

初めてのサーブレット作成の例

サーブレットの作成に関する一般的なフレームワークを説明するには、基本的な例を参照するのが一番効率的です。

Hello World コード

このサーブレットはクライアント上に「Hi There!」を表示します。このコメントには、サーブレットの作成の基本的な側面の一部が記載されています。

```
// You must import at least the following packages for any servlet you write.
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Extend HttpServlet, the base servlet implementation.
public class HelloServlet extends HttpServlet {

  // Override the base implementation of doGet(), as desired.
  public void doGet (HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
```

```
    // Set the MIME type for the response content.
    resp.setContentType("text/html");

    // Get an output stream to use in sending the output to the client.
    ServletOutputStream out = resp.getOutputStream();
    // Put together the HTML code for the output.
    out.println("<html>");
    out.println("<head><title>Hello World</title></head>");
    out.println("<body>");
    out.println("<h1>Hi There!</h1>");
    out.println("</body></html>");
  }
}
```

サーブレットのコンパイルおよびデプロイ

サンプル・サーブレットのコードを OC4J スタンドアロン環境で使用する場合は、OC4J のデフォルトの Web アプリケーションの /WEB-INF/classes ディレクトリ内に HelloServlet.java として保存してください。（詳細は、5-24 ページの「OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーション」を参照してください。）

次に、サーブレットをコンパイルします。最初に、OC4J により提供される、servlet.jar が CLASSPATH 内にあることを確認してください。これには、Sun 社の javax.servlet および javax.servlet.http パッケージが含まれます。

注意： 開発時およびテスト時に、サーブレット・コードに対して OC4J の自動コンパイル機能を使用すると便利です。この機能は、OC4J 構成ファイル・ディレクトリにある global-web-application.xml ファイルの <orion-web-app> 要素の設定 development="true" を使用して有効化します。source-directory 属性も適切に設定する必要があります。自動コンパイルが有効な場合、変更したサーブレット・ソースを適切なディレクトリに保存すると、OC4J サーバーは、次の起動時にサーブレットを自動的にコンパイルし、再デプロイします。

development および source-directory の詳細は、6-2 ページの「global-web-application.xml および orion-web.xml の要素の説明」を参照してください。

サーブレットの実行

OC4J サーバーが起動されていて稼働中であり、クラス名による起動が /servlet/ の servlet-webdir の組込みデフォルト設定で有効になっている場合には、次のようにサーブレットを起動して、その出力を Web ブラウザに表示できます。*host* は、OC4J サーバーが実行されているホストの名前、*port* は、Web リスナーのポートです。

```
http://host:port/servlet/HelloServlet
```

（クラス名による起動および OC4J の servlet-webdir 属性の詳細は、2-22 ページの「OC4J 開発時におけるクラス名によるサーブレットの起動」を参照してください。）

OC4J スタンドアロン環境では、ポート 8888 を使用して OC4J Web リスナーに直接アクセスします。（概要は、2-2 ページの「開発用 OC4J スタンドアロン」を参照してください。）

この例では、「/」は Web アプリケーションのコンテキスト・パスです。OC4J スタンドアロンでは、これがデフォルトの Web アプリケーションに対するデフォルトのコンテキスト・パスです。

重要： ここに示すサーブレットの起動方法は、クラス名によって直接起動します。これは開発環境には適していますが、重大なセキュリティ上のリスクが発生します。本番環境では、OC4J をこのモードで動作するよう構成しないでください。詳細は、2-22 ページの「OC4J 開発時におけるクラス名によるサーブレットの起動」および 2-38 ページの「その他のセキュリティに関する考慮事項」を参照してください。

2

サーブレットの開発

この章では、OC4J および Oracle Application Server のサーブレット開発に関する基本情報を説明します。次の項が含まれます。最初の項では、開発およびテスト・フェーズで役立つ、OC4J のスタンドアロン・バージョンの使用方法を説明します。

- 開発用 OC4J スタンドアロン
- サーブレット開発の基本と主要な考慮事項
- 追加の Oracle 機能
- サーブレットの起動
- サーブレットのセッション
- サーブレットのセキュリティ
- サーブレットのベスト・プラクティスのサマリー

開発用 OC4J スタンドアロン

このマニュアルでは、OC4J スタンドアロンを、少なくとも初期開発フェーズで使用していることを前提としています。これは、OC4J スタンドアロンが、1 つの OC4J インスタンスを Oracle Application Server 環境および Oracle Enterprise Manager 10*g* 外で使用している場合を指します。OC4J スタンドアロンは、初期開発時に使用すると特に便利です。

次の各項で、概要と主な考慮事項を説明します。

- 概要 : OC4J スタンドアロンの使用
- 開発用の主な OC4J のフラグ
- OC4J スタンドアロンからの tools.jar の削除

OC4J スタンドアロンを取得するには、Oracle Technology Network（OTN）から `oc4j_extended.zip` ファイルをダウンロードします。サイトは次のとおりです。

`http://www.oracle.com/technology/tech/java/oc4j/index.html`

注意 :

- OC4J スタンドアロンを使用するには、Sun 社の JDK のサポートされているバージョンがインストールされている必要があります。JDK は OC4J スタンドアロン製品の付属ではありません。
- 開発時には、開発およびデプロイのためのビジュアル開発ツールである Oracle JDeveloper の使用も検討してください。このツールには、2-18 ページの「Oracle JDeveloper によるサーブレット開発のサポート」で説明するように数多くの利点があります。

概要 : OC4J スタンドアロンの使用

スタンドアロンの OC4J インスタンスの起動、管理および制御は、`oc4j.jar`（OC4J スタンドアロンの実行可能ファイル）と `admin.jar` コマンドライン・ユーティリティ（スタンドアロン製品に付属）を使用して行います。EAR ファイルをデプロイし、Web モジュールを `admin.jar` によってバインドすると、主要な構成ファイルが自動的に更新されます。

注意 : `admin.jar` ユーティリティの主要な側面については、第 5 章「デプロイおよび構成の概要」で説明しています。特に、5-26 ページの「EAR ファイルの OC4J スタンドアロンへのデプロイ」を参照してください。詳細は、『Oracle Application Server Containers for J2EE スタンドアロン・ユーザーズ・ガイド』を参照してください。

テスト中は、J2EE ディレクトリ構造に基づいて EAR ファイルまたは個別のファイルを手動でインストールし、手動で主要な構成ファイルを更新して処理を完了することも可能です。こうすると、OC4J が解凍およびアプリケーションのデプロイを行うよう、トリガーされます。

独立した Web アプリケーションが存在する場合、EAR ファイルを使用するかわりに、OC4J のデフォルトの J2EE アプリケーション内で WAR ファイル（またはディレクトリ構造）としてデプロイすることも可能です。

また、便利なテスト・モードで、OC4J のデフォルトの Web アプリケーションに個別のサーブレットまたは JSP ページをデプロイすることもできます。

OC4J スタンドアロン環境には、デフォルトで、次の主要ディレクトリが含まれています。

- J2EE ホーム : `j2ee/home`。OC4J のインストール先に相対的です。
- グローバル構成ファイルのディレクトリ : `j2ee/home/config`
- デフォルトの Web アプリケーションのルート・ディレクトリ : `j2ee/home/default-web-app`

- デプロイされたアプリケーションのルート・ターゲット・ディレクトリ：`j2ee/home/applications`
- デプロイメント・ディスクリプタ（`orion-web.xml` および `orion-application.xml` など）のルート・ターゲット・ディレクトリ：`j2ee/home/application-deployments`

最も単純な例では、テスト・サーブレットを OC4J のデフォルトの Web アプリケーションにデプロイするには、クラス・ファイルをデフォルトの Web アプリケーションのルート・ディレクトリ内の `/WEB-INF/classes` ディレクトリに置きます。

デプロイに関する詳細な考慮事項は、特に OC4J スタンドアロン・ユーザーを対象に、第 5 章で説明しています。特に次の項を参照してください。

- 「OC4J のデプロイおよび構成の一般的な概要」5-2 ページ
- 「OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーション」5-24 ページ
- 「OC4J スタンドアロンへのデプロイ方法」5-22 ページ

また、OC4J スタンドアロンでのサーブレット起動に関する情報は、2-24 ページの「OC4J スタンドアロン環境でのサーブレットの起動」を参照してください。

`admin.jar` およびスタンドアロン・プロセスの起動、停止、構成および管理方法の詳細は、『Oracle Application Server Containers for J2EE スタンドアロン・ユーザーズ・ガイド』を `OC4J_extended.zip` とともにダウンロードしてください。

開発用の主な OC4J のフラグ

開発時、通常は OC4J スタンドアロンの使用中に注意すべき OC4J のフラグがいくつかあります。これらのフラグは個別に独立して動作します。

- OC4J の `check-for-updates` フラグ

  OC4J の `server.xml` ファイルで、トップレベルの `<application-server>` 要素に `check-for-updates` 属性が含まれており、これは OC4J タスク・マネージャが XML 構成ファイル（`server.xml` 自体を含む）、ライブラリ JAR ファイルおよび JSP タグ・ライブラリの更新を自動的に確認するかどうかを決定します。これは、OC4J のポーリングと呼ばれます。開発時に使用するデフォルト設定は、`true` です。ポーリングを無効にするには、次のようにします。

  ```
  <application-server ... check-for-updates="false" ... >
     ...
  </application-server>
  ```

  たとえば、開発中に手動処理（エキスパート・モードとみなされます）を行う際、アプリケーションを手動でインストールし、Web アプリケーションの定義およびバインドを行うために、`server.xml` ファイル、グローバル `application.xml` ファイルおよび `http-web-site.xml` ファイルを必要に応じて手動で更新することが可能です。デフォルトの `check-for-updates="true"` 設定では、OC4J は自動的に変更を検出し、アプリケーションをデプロイします（必要に応じて処理中に EAR または WAR ファイルを解凍します）。

  このファイルの詳細は、5-9 ページの「OC4J のトップレベルのサーバー構成ファイル : server.xml」を参照してください。

  > **重要：** `check-for-updates` フラグは、OC4J スタンドアロンでのみ使用されます。Oracle Application Server 環境では、Oracle Process Management and Notification（OPMN）システムおよび Distributed Configuration Management（DCM）サブシステムが OC4J ファイルの更新機能を管理するため、無視されます。

- admin.jar の -updateConfig オプション

  check-for-updates="false" に設定されているときに OC4J の XML 構成ファイルを手動で更新する場合、admin.jar ユーティリティで -updateConfig オプションを使用して、更新のワンタイム・チェックを 1 回トリガーできます。

  ```
  % java -jar admin.jar -updateConfig
  ```

  重要： 確認機能を無効にした後に再度有効にする場合、OC4J が変更を検出するよう、check-for-updates="true" を設定した後で admin.jar の -updateConfig オプションを使用する必要があります。その後、自動確認は再度有効になります。

- サーブレットの development フラグ

  開発およびテスト中、global-web-application.xml ファイルまたは orion-web.xml ファイルの <orion-web-app> 要素の development="true" 設定を使用すると便利です。この設定を使用すると、特定のディレクトリ（通常は /WEB-INF/classes ディレクトリ、または <orion-web-app> の source-directory 属性に基づく）内のサーブレット・コードを更新すると、次に起動された際、サーブレットは自動的に再コンパイルおよび再デプロイされます。development および source-directory 属性の詳細は、6-2 ページの「global-web-application.xml および orion-web.xml の要素の説明」を参照してください。

- JSP の main_mode フラグ

  このフラグは、JSP コンテナの操作モードを送ります。特に、JSP ページの再変換、および変更された JSP 生成の Java クラスの再ロードに使用されます。開発中、recompile（デフォルト）設定を使用して JSP ページのタイムスタンプを確認し、前回ロードされた後に変更されている場合、再変換および再ロードします。（本番モードなどでタイムスタンプを確認しない場合、justrun 設定を使用します。）このフラグの詳細および設定方法は、『Oracle Application Server Containers for J2EE JavaServer Pages 開発者ガイド』を参照してください。

OC4J スタンドアロンからの tools.jar の削除

OC4J 9.0.3 スタンドアロン実装では、Sun 社の JDK 1.3.1 の tools.jar ファイルが提供されていました。このファイルには、たとえば、java フロントエンド実行可能ファイルや javac コンパイラ実行可能ファイルなどが他のコンポーネントとともに含まれています。

OC4J 10.1.2 スタンドアロンの実装には、tools.jar ファイルは含まれていません。そのため、OC4J そのものをインストールする前に、OC4J がサポートする JDK をインストールする必要があります。OC4J が OC4J 10.1.2 実装にサポートしている JDK のバージョンは、JDK 1.3.1（OC4J スタンドアロンのみ）と JDK 1.4 です。Oracle Application Server 10*g* リリース 2（10.1.2）には JDK 1.4 が含まれているので、通常は OC4J スタンドアロンにもこの JDK バージョンを使用します。ただし、移行の問題を考慮する必要があります。特に、すべての起動されたクラスがパッケージ内に存在する必要があるという JDK 1.4 の要件に注意してください。2-13 ページの「JDK 1.4 の考慮事項 : パッケージ内に存在しないクラスを起動できない」を参照してください。

注意： OC4J スタンドアロンでは、java -jar oc4j.jar コマンドを使用して java にアクセスするのと同じディレクトリから javac を使用するので、javac の適切なバージョンを使用できます。

サーブレット開発の基本と主要な考慮事項

ほとんどの HTTP サーブレットは、標準フォームに従って作成されます。HTTP サーブレットは、HttpServlet クラスを拡張するパブリック・クラスとして記述されます。コンテナがサーブレットをロードするときの初期化作業や、コンテナがサーブレットをシャットダウンするときのファイナライズ作業にコードが必要な場合、サーブレットは、init() メソッドと destroy() メソッドをオーバーライドします。ほとんどのサーブレットは、HTTP の GET または POST リクエストを適切に処理するために、HttpServlet の doGet() メソッドまたは doPost() メソッドをオーバーライドします。この 2 つのメソッドは、リクエスト・オブジェクトとレスポンス・オブジェクトをパラメータとして取得します。

この章では、1-9 ページの「初めてのサーブレット作成の例」の HelloWorldServlet よりも高度なサンプル・サーブレットについて説明します。

次の各項で、アプリケーションを開発する前に考慮する機能と問題点について説明します。

- サンプルのコード・テンプレート
- サーブレットのライフ・サイクル
- サーブレットの事前ロード
- サーブレットのクラスロードおよびアプリケーションの再デプロイ
- サーブレット情報の交換
- サーブレットのインクルードおよび転送
- サーブレットのスレッド・モデルと関連する考慮事項
- サーブレットのパフォーマンスと監視
- JDK 1.4 の考慮事項 : パッケージ内に存在しないクラスを起動できない

サンプルのコード・テンプレート

次に、サーブレット開発のサンプルのコード・テンプレートを示します。

```
public class myServlet extends HttpServlet {

  public void init(ServletConfig config) {
  }

  public void destroy() {
  }

  public void doGet(HttpServletRequest request, HttpServletResponse)
                  throws ServletException, IOException {
  }

  public void doPost(HttpServletRequest request, HttpServletResponse)
                  throws ServletException, IOException {
  }

  public String getServletInfo() {
    return "Some information about the servlet.";
  }
```

オプションで init()、destroy() および getServletInfo() メソッドをオーバーライドできますが、最も単純なサーブレットは、doGet() または doPost() のうちの一方のみオーバーライドします。

init() メソッドをオーバーライドするのは、サーブレットの存続期間中に 1 回のみ必要となる特別な処理を実行するためです。次のような処理が含まれます。

- データ・ソース接続を確立する。
- サーブレット構成オブジェクトから初期化パラメータを取得し、その値を格納する。

- サーブレットが必要とする永続データをリカバリする。
- ハッシュテーブルなど、重要なセッション・オブジェクトを作成する。
- ServletContext オブジェクトの log() メソッドでサーブレットのバージョンを記録する。

サーブレットのライフ・サイクル

サーブレットには、予測可能で管理可能なライフ・サイクルが存在します。

- サーブレットのロード時に、そのサーブレットの構成の詳細は web.xml から読み取られます。これらの詳細には初期化パラメータを含めることができます。
- シングルスレッド・モデルを使用している場合を除き、サーブレットのインスタンスは 1 つしか存在しません。2-11 ページの「サーブレットのスレッド・モデルと関連する考慮事項」を参照してください。
- クライアント・リクエストは、汎用サーブレットの service() メソッドを起動します。service() メソッドはそのリクエストを、リクエスト・ヘッダーの情報に基づいて、doGet() または doPost()（あるいは、別のオーバーライドされたリクエスト処理メソッド）に委任します。
- フィルタは、リクエスト時やレスポンス時にサーブレットの動作を変更するために、コンテナとサーブレットの間に置くことができます。詳細は、3-2 ページの「サーブレット・フィルタ」を参照してください。
- サーブレットは、他のサーブレットにリクエストを転送したり、他のサーブレットからの出力を含められます。2-10 ページの「サーブレットのインクルードおよび転送」を参照してください。
- レスポンスはレスポンス・オブジェクトによってクライアントに返されます。コンテナは、これを HTTP レスポンス・ヘッダーでクライアントに返します。サーブレットでは、java.io.PrintWriter または javax.servlet.ServletOutputStream オブジェクトを使用して、レスポンス・オブジェクトを書き込みます。
- コンテナは、サーブレットがアンロードされる前に destroy() メソッドをコールします。

サーブレットの事前ロード

通常、サーブレット・コンテナは、最初にリクエストされたときに、サーブレット・クラスをインスタンス化してロードします。ただし、server.xml ファイル、Web サイトの XML ファイル（default-web-site.xml または http-web-site.xml）と web.xml ファイルの設定を使用してサーブレットの事前ロードを設定できます。事前ロードされたサーブレットは、OC4J サーバーの起動時または Web アプリケーションのデプロイ時や再デプロイ時にロードされ、初期化されます。

事前ロードには、次の手順が必要です。

1. 使用する server.xml ファイル内の <application> 要素の属性が auto-start="true" に設定されていることを確認します。アプリケーションをデプロイすると、デフォルトで OC4J がこの設定を挿入します。
2. 属性設定 load-on-startup="true" を、使用する Web サイトの XML ファイルの <web-site> 要素の <web-app> サブ要素に指定します。Web サイトの XML ファイルの要素および属性の詳細は、6-19 ページの「Web サイトの XML ファイルの構成」を参照してください。
3. 事前ロードするサーブレットに対して、Web モジュールの web.xml ファイルの <servlet> 要素の下に <load-on-startup> サブ要素が存在する必要があります。

表 2-1 に、web.xml の <load-on-startup> 要素の動作を説明します。

表 2-1 web.xml ファイルの <load-on-startup> の動作

値の範囲	動作
0（ゼロ）未満（<0） 例： `<load-on-startup>-1</load-on-startup>`	サーブレットは事前ロードされません。
0（ゼロ）以上（>=0） 例： `<load-on-startup>1</load-on-startup>`	サーブレットは事前ロードされます。サーブレットのロードは、同じ Web アプリケーションに事前ロードされる他のサーブレットに関して load-on-startup の値に従い、低い数値から順に行われます。（たとえば、ゼロ（0）は 1 の前にロードされ、1 は 2 の前にロードされます。）
空要素 例： `<load-on-startup/>`	load-on-startup 値が `Integer.MAX_VALUE` の場合と同様に動作します。この場合、サーブレットは、load-on-startup 値がゼロ（0）以上のサーブレットの後にロードされます。

注意： OC4J では、startup クラスおよび shutdown クラスの指定をサポートしています。startup クラスは `server.xml` ファイルの `<startup-classes>` 要素によって指定され、OC4J の初期化直後にコールされます。shutdown クラスは `server.xml` ファイルの `<shutdown-classes>` 要素によって指定され、OC4J の終了直前にコールされます。

startup クラスは他の事前ロードされるサーブレットより先にコールされる点に注意してください。

startup クラスおよび shutdown クラスの詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

サーブレットのクラスロードおよびアプリケーションの再デプロイ

次の各項で、サーブレットのクラスロードおよびアプリケーションのロードに関する OC4J の機能と重要な考慮点について説明します。

- OC4J の Web アプリケーションの再デプロイとクラスの再ロード機能
- OC4J におけるシステム・クラスより前の WAR ファイル・クラスのロード
- Oracle Application Server での OC4J サーブレット間におけるキャッシュ内の Java オブジェクトの共有

OC4J の Web アプリケーションの再デプロイとクラスの再ロード機能

OC4J では、次のような状況が発生すると、OC4J ポーリングによっては Web アプリケーションが再デプロイされ、リクエスト時に、サーブレット・クラスおよびすべての依存クラスが再ロードされます。

注意：

- OC4J ポーリングとは、OC4J タスク・マネージャによる、ライブラリ JAR ファイルおよび XML 構成ファイルの更新の自動確認機能のことです。Oracle Application Server 環境では、これは OPMN および DCM によって制御されます。OC4J スタンドアロンでは、`server.xml` の `check-for-updates` フラグ（デフォルトで `true` に設定）によって制御されます。2-3 ページの「開発用の主な OC4J のフラグ」を参照してください。
- ここでは、Web アプリケーションの再デプロイとは、OC4J による Web アプリケーションの実行領域からの削除、Web アプリケーションの実行に関連付けられたクラスローダーの削除、`web.xml` と `orion-web.xml` の再解析、サーブレット・リスナー、フィルタおよびマッピングの再初期化という一連の処理を指します。

- `/WEB-INF/classes` 内のサーブレットの `.class` ファイルが再コンパイルなどによって変更されると、次のサーブレットのリクエスト時には、関連付けられた Web アプリケーションが再デプロイされ、サーブレット・クラスおよびすべての依存クラスが再ロードされます。これは OC4J のポーリングには依存しません。サーブレットが次にリクエストされるまで何も発生しない点に注意してください。また、`/WEB-INF/classes` 内のサーブレット以外の `.class` ファイルが変更されても、何も再ロードされません。

注意： `global-web-application.xml` または `orion-web.xml` の `<classpath>` 要素で指定されたディレクトリ内のサーブレット・クラス・ファイルを変更した場合も、`/WEB-INF/classes` 内のサーブレット・クラス・ファイルを変更した場合と同じ影響があります。ただし、`<classpath>` 内の JAR ファイルまたは依存クラス・ファイル（JavaBeans のファイルなど）を変更しても、影響はありません。

- `web.xml` ファイルが変更されるか、または `/WEB-INF/lib` 内のライブラリ JAR ファイルが変更され、OC4J ポーリングが有効になっている場合、OC4J タスク・マネージャの次の実行時に、関連付けられた Web アプリケーションが再デプロイされます。デフォルトでは 1 秒につき 1 回です。Web アプリケーション内のすべてのサーブレット・クラスおよび依存クラスは、サーブレットの次のリクエスト時に再ロードされます。または、ポーリングが無効の場合、ワンタイム・ポーリングと、それに伴う再デプロイおよび再ロードを、`admin.jar` の `-updateConfig` オプションを使用してトリガーできます。

次の重要な考慮事項があります。

- 前述の使用例で、サーブレットが事前ロードされるよう設定されている場合は、サーブレットおよびその依存クラスは、次のリクエスト時ではなく、即時に再ロードされます。これは、`load-on-startup` 設定に基づきます。2-6 ページの「サーブレットの事前ロード」を参照してください。
- OC4J のサーブレット再ロード機能は、JSP ページ実装クラスには拡張されません。JSP ページ実装の `.class` ファイルを変更しても、再ロードは実行されません。JSP の再コンパイルおよび再ロード動作は、JSP の `main_mode` フラグによって決定されます。『Oracle Application Server Containers for J2EE JavaServer Pages 開発者ガイド』を参照してください。
- 親アプリケーションの Web モジュール内のクラスは子アプリケーションに表示されませんが、親アプリケーションのそれ以外のクラス（たとえば、EJB など）は表示されます。
- 共有コードのディレクトリまたは JAR、ZIP ファイルを指定するために、グローバル `application.xml` ファイルまたは `server.xml` ファイル内の `<library>` 要素を使用できます。OC4J は、起動時に、`<library>` 要素で指定されているすべての JAR または ZIP ファイル内の全クラス、および `<library>` 要素で指定されているディレクトリ内のすべての JAR または ZIP ファイル内の全クラスをロードします。

不要なオーバーヘッドを回避するには、`<library>` 要素をあまり使用せず、なるべくディレクトリ全体ではなく特定の JAR または ZIP ファイルを指定し、ディレクトリを指定する場合は、そのディレクトリ内の JAR または ZIP ファイルの数を少なくすることをお薦めします。

`application.xml` には、デフォルトで `j2ee/home/applib` ディレクトリの `<library>` 要素が含まれています。

注意： `<library>` 要素で指定されている JAR または ZIP ファイル、または `<library>` 要素で指定されているディレクトリにある JAR または ZIP ファイルを変更しても、それ自体では Web アプリケーションの再デプロイやクラスの再ロードは発生しません。OC4J タスク・マネージャはこれらの共有ライブラリはポーリングしません。

OC4J におけるシステム・クラスより前の WAR ファイル・クラスのロード

サーブレット仕様では、ローカル・クラス（WAR ファイル内のクラス）をシステム・クラス（環境内のその他のクラス）の前にロードすることを推奨しています。ただし必須ではありません。WAR ファイル内のクラスには、WAR ファイルのマニフェストの CLASSPATH 内のクラスも含まれます。デフォルトでは、OC4J サーブレット・コンテナはローカル・クラスを最初にロードしません。ただし、これは、`global-web-application.xml` または `orion-web.xml` の `<web-app-class-loader>` 要素を使用して構成可能です。この要素には次の 2 つの属性があります。

- `search-local-classes-first`: この属性を `true` に設定すると、システム・クラスの前に WAR ファイルのクラスが検索され、ロードされます。デフォルトの設定は、`false` です。
- `include-war-manifest-class-path`: この属性を `false` に設定すると、`search-local-classes-first` の設定に関係なく、WAR ファイル・クラスの検索時とロード時に、WAR ファイルのマニフェストの `Class-Path` 属性に指定された CLASSPATH は含まれません。デフォルトの設定は、`true` です。

注意：

- 両方の属性を `true` に設定すると、WAR ファイル内に物理的に常駐するクラスが、WAR ファイルのマニフェストの CLASSPATH のクラスより前にロードされるよう、全体の CLASSPATH が構成されます。したがって、競合が発生した場合、WAR ファイル内に物理的に常駐するクラスが優先されます。
- サーブレット仕様により、`search-local-classes-first` 機能は、`java.*` または `javax.*` パッケージでのクラスのロードには使用できません。

6-2 ページの「global-web-application.xml および orion-web.xml の要素の説明」も参照してください。

Oracle Application Server での OC4J サーブレット間におけるキャッシュ内の Java オブジェクトの共有

Oracle Application Server Java Object Cache の分散機能を活用する、つまりサーブレット間でキャッシュ内のオブジェクトを共有するには、アプリケーションのデプロイを一部変更する必要があります。サーブレット間で共有される、または JVM 間で配布されるユーザー定義オブジェクトは、システム・クラスローダーでロードされる必要があります。ただし、デフォルトでは、サーブレットによってロードされたオブジェクトは、コンテキスト・クラスローダーによってロードされます。コンテキスト・クラスローダーによってロードされたオブジェクトは、そのクラスローダーに対応するサーブレット・コンテキスト内のサーブレットに対してのみ参照可能です。オブジェクト定義は、他のサーブレットまたは別の JVM のキャッシュには使用できません。ただし、システム・クラスローダーでロードされたオブジェクトの場合、そのオブジェクト定義は、他のサーブレットや他の JVM 上のキャッシュに対して使用可能です。

OC4J では、システム CLASSPATH は、oc4j.jar ファイルのマニフェストおよび cache.jar などの関連 .jar ファイルから導出されます。環境内の CLASSPATH は無視されます。Oracle Application Server 環境で OC4J の CLASSPATH にキャッシュ内のオブジェクトを含めるには、.class ファイルを次のいずれかのように処理します。

- *ORACLE_HOME*/javacache/cachedobjects/classes ディレクトリにコピー
- *ORACLE_HOME*/javacache/cachedobjects/share.jar ファイルに追加

classes ディレクトリと share.jar ファイルは両方とも cache.jar のマニフェストに組み込まれます。したがって、システム CLASSPATH にも組み込まれます。

Oracle Application Server Java Object Cache の詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』を参照してください。

サーブレット情報の交換

サーブレットは、通常、次のような 1 つ以上のソースから情報を受け取ります。

- リクエスト・オブジェクトからのパラメータ
- HTTP セッション・オブジェクト
- サーブレット・コンテキスト・オブジェクト
- サーブレット外部のデータ・ソース（データベース、ファイル・システム、外部センサーなど）

サーブレットは、レスポンス・オブジェクトに情報を追加し、コンテナは、そのレスポンスをクライアントに返信します。

サーブレットのインクルードおよび転送

多くのサーブレットは、別のサーブレットをインクルードしたり、別のサーブレットに転送するなど、別のサーブレットを処理時に使用します。

サーブレットの用語では、サーブレットのインクルードは、サーブレットが別のサーブレットからのレスポンスを自らのレスポンスに含める処理を指します。処理およびレスポンスは、最初は元のクライアントによって処理され、次にインクルードされたサーブレットに渡され、インクルードされたサーブレットが完了した後に元のサーブレットに返されます。

サーブレットの転送では、処理は転送コールまでは元のサーブレットによって行われます。転送コールの時点で、レスポンスがリセットされ、ターゲット・サーブレットがリクエストの処理を引き継ぎます。レスポンスがリセットされると、出力ストリーム内の HTTP ヘッダー設定および情報は、すべてレスポンスから消去されます。転送後、元のサーブレットは、ヘッダーの設定やレスポンスへの書込みはできません。また、レスポンスがすでにコミットされている場合、サーブレットは転送または別のサーブレットのインクルードを行うことはできません。

別のサーブレットを転送またはインクルードするには、次のいずれかのサーブレット・コンテキスト・メソッドを使用して、そのサーブレットのリクエスト・ディスパッチャを取得する必要があります。

- `RequestDispatcher getRequestDispatcher(String path)`
- `RequestDispatcher getNamedDispatcher(String name)`

getRequestDispatcher() の場合、ターゲット・サーブレットのパスを入力します。getNamedDispatcher() の場合、web.xml ファイル内のそのサーブレットの <servlet-name> 要素に基づいて、ターゲット・サーブレットの名前を入力します。

いずれの場合も、返されるオブジェクトは、javax.servlet.RequestDispatcher インタフェースを実装するクラスのインスタンスです。（このクラスはサーブレット・コンテナによって提供されます。）リクエスト・ディスパッチャは、ターゲット・サーブレットのラッパーです。一般に、リクエスト・ディスパッチャの役割は、ラップ対象のリソースにリクエストをルーティングする際の媒介として機能することです。

リクエスト・ディスパッチャには次のメソッドがあり、インクルードまたは転送に使用します。

- ```
  void include(ServletRequest request,
               ServletResponse response)
  ```
- ```
  void forward(ServletRequest request,
               ServletResponse response)
  ```

このように、これらのメソッドをコールする際には、リクエストおよびレスポンス・オブジェクトを渡します。

注意：

- サーブレットが別のサーブレットを転送したりインクルードする際、デフォルトの OC4J の機能では、ターゲット・サーブレットおよび元のサーブレットに web.xml のセキュリティ制約が強制されます。サーブレット仕様に準拠しないことになりますが、global-web-application.xml または orion-web.xml の <authenticate-on-dispatch> 要素を使用して、この動作を無効化できます。この要素の詳細は、6-2 ページの「global-web-application.xml および orion-web.xml の要素の説明」を参照してください。
- サーブレットが別のサーブレットを転送したりインクルードする際、デフォルトの動作では、元のサーブレットに適用されているフィルタはターゲット・サーブレットでは実行されませんが、この動作は構成によって変更できます。3-3 ページの「転送またはインクルード・ターゲットのフィルタ」を参照してください。

サーブレットのスレッド・モデルと関連する考慮事項

非分散環境のサーブレットでは、サーブレット・コンテナは 1 回のサーブレット宣言当たり 1 つのサーブレット・インスタンスのみ使用します。分散環境では、コンテナは各 JVM の 1 回のサーブレット宣言当たり 1 つのサーブレット・インスタンスを使用します。したがって、OC4J サーブレット・コンテナを含めたサーブレット・コンテナは、サーブレットに対する同時リクエストを処理するには、一般的に複数のスレッドを使用してサーブレットの service() メソッドを同時に複数実行します。

サーブレットの開発者は、この点を念頭に置き、複数のスレッドによる同時処理用の措置を講じ、共有リソースへのアクセスが同期化または調整されるよう、サーブレットを設計する必要があります。共有リソースには、主に 2 種類あります。

- インスタンスまたはクラス変数などのメモリー内データ
- ファイル、データベース接続およびネットワーク接続などの外部オブジェクト

1 つの方法は、service() メソッド全体を同期化することです。ただし、一方で、これはパフォーマンスに影響を与えるおそれがあります。

よりよい方法は、同期ブロックにより、インスタンスまたはクラス・フィールドを選択的に保護したり、外部リソースへのアクセスを選択的に保護することです。

これらが実行できない場合のために、サーブレット仕様では、シングルスレッド・モデルをサポートしています。サーブレットが javax.servlet.SingleThreadModel インタフェースを実装する場合、サーブレット・コンテナは、サーブレットのどのインスタンスの service() メソッドにも、複数のリクエスト・スレッドが存在しないことを保証する必要があります。このために、OC4J では、通常サーブレット・インスタンスのプールを作成し、各同時リクエストを個別のインスタンスが処理します。ただし、このプロセスはサーブレット・コンテナのパフォーマンスに大きく影響するおそれがあり、なるべく回避する必要があります。さらに、SingleThreadModel インタフェースは、サーブレット仕様のバージョン 2.4 では使用されなくなります。

マルチスレッドの一般情報は、次の Web サイトで、Sun 社の「Java Tutorial on Multithreaded Programming」を参照してください。

http://java.sun.com/docs/books/tutorial/essential/threads/multithreaded.html

サーブレットのパフォーマンスと監視

次の各項で、サーブレットのパフォーマンスの考慮事項を示し、Oracle Application Server Dynamic Monitoring Service（DMS）の概要を説明します。

- 一般的なパフォーマンスに関する考慮事項
- Oracle Application Server ダイナミック・モニタリング・サービス

DMS およびパフォーマンス・メトリック用の `dmstool` の説明を含めた OC4J のパフォーマンスに関する一般情報は、『Oracle Application Server パフォーマンス・ガイド』を参照してください。

一般的なパフォーマンスに関する考慮事項

この項では、パフォーマンスに影響を与える可能性のある問題（主にこのマニュアルの他の部分で説明している内容）をまとめます。

- アプリケーション内の Web ページの最適な期限設定を考慮してください。`global-web-application.xml` または `orion-web.xml` 内の `<orion-web-app>` の `<expiration-setting>` サブ要素を使用して、指定した URL パターンに一致するページの期限を設定することが可能です。（詳細は、6-2 ページの「global-web-application.xml および orion-web.xml の要素の説明」を参照してください。）適切な設定により、アプリケーションへの負荷が減少し、パフォーマンスが向上します。
- 複数の同時リクエストの同期化または調整がパフォーマンスに与える影響や、スレッド・モデルに関する考慮事項にも注意してください。2-11 ページの「サーブレットのスレッド・モデルと関連する考慮事項」を参照してください。
- 分散環境におけるセッションの状態のレプリケートは、パフォーマンスと関連があります。セッション・オブジェクトへの `setAttribute()` コールごとにレプリケーションがトリガーされるため、サーブレットでコールの回数が多い場合、パフォーマンスに影響を与えるおそれがあります。また、パフォーマンス上の理由から、OC4J はセッションの状態のレプリケーションが正常に完了したかどうかを確認しません。2-28 ページの「分散アプリケーションでのセッションのレプリケーション」を参照してください。
- サーブレットの構成パラメータは、パフォーマンスに著しい影響を与えることがあります。`<orion-web-app>` 要素の `file-modification-check-interval` 属性の詳細は、6-2 ページの「global-web-application.xml および orion-web.xml の要素の説明」を参照してください。`<web-site>` 要素の `use-keep-alives` 属性の詳細は、6-19 ページの「Web サイト XML ファイルの要素の説明」を参照してください。
- 追加の JSP 関連構成パラメータは、パフォーマンスに著しい影響を与えることがあります。`<orion-web-app>` 要素の `simple-jsp-mapping` および `enable-jsp-dispatcher-shortcut` 属性の詳細は、6-2 ページの「global-web-application.xml および orion-web.xml の要素の説明」を参照してください。
- OC4J スタンドアロンは、1 つのアプリケーションを複数の Web サイトで共有する処理モードをサポートしています。この場合、サイトは特定のホストおよびポートとして定義します。この機能は、一部の通信でのみ HTTPS が必要なセキュアなアプリケーションで使用します。重要でない通信に HTTP ポートを使用することにより、パフォーマンスが向上します。`<web-app>` 要素の `shared` 属性の詳細は、6-19 ページの「Web サイト XML ファイルの要素の説明」を参照してください。
- OC4J スタンドアロンを本番環境として使用する場合（通常は使用しない）、`server.xml` の `check-for-updates` フラグを必ず無効にします。2-3 ページの「開発用の主な OC4J のフラグ」を参照してください。

Oracle Application Server ダイナミック・モニタリング・サービス

Oracle Application Server 環境では、DMS により、OC4J を含めた様々なコンポーネントにパフォーマンス監視機能が追加されます。DMS の目的は、組込みパフォーマンス測定により、実行時動作に関する情報をユーザーに提供し、ユーザーがパフォーマンスの問題を診断、分析およびデバッグできるようにすることです。DMS はこの情報をパッケージで提供し、この情報はライブ・デプロイ中を含め、いつでも使用できます。データは HTTP を通じて公開され、ブラウザで表示できます。

グローバル application.xml ファイルおよび default-web-site.xml ファイルに、Spy サーブレットおよび監視エージェントなどの DMS サーブレットの標準構成が存在します。

OC4J の application.xml ファイルでは、Web モジュール dms および dms0 と、それぞれの WAR ファイルへのパスが指定されています。default-web-site.xml ファイルは、これらの Web モジュールが OC4J デフォルト・アプリケーションにデプロイされていることを指定し、コンテキスト・パスにバインドします。これらの DMS 構成は、直接変更しないでください。

DMS へのアクセス、DMS 情報の表示および DMS の構成（適切な場合）には、Oracle Enterprise Manager 10*g* を使用してください。

JDK 1.4 の考慮事項 : パッケージ内に存在しないクラスを起動できない

Oracle Application Server 10*g* リリース 2（10.1.2）付属の環境である Sun 社の JDK 1.4 環境に移行する際の考慮事項の中には、特にサーブレットおよび JSP 開発者が注意する必要のある点があります。

以前の JDK バージョンにおけるセキュリティ上の問題やあいまいな点に対処するために、Sun 社は無名のネームスペースから型をインポートするインポート文を拒否するように Java コンパイラを変更しました。これで、パッケージ内に存在しないクラス（クラスのメソッド）は基本的に起動できません。起動しようとすると、コンパイル時に致命的エラーが発生します。

これにより、特に JSP ページから JavaBeans を起動する JSP 開発者が影響を受けます。このような場合、Bean はパッケージの外部に存在することが多いためです（ただし、JSP 仕様のバージョン 2.0 では、新規のコンパイラ要件を満たすために、Bean はパッケージ内に存在する必要があります）。パッケージ外部の JavaBeans が起動されると、OC4J 9.0.3/JDK 1.3.1 環境で構築および実行されていた JSP アプリケーションは、OC4J 10.1.2/JDK 1.4 環境では稼働しなくなります。

すべての JavaBeans と起動された他のクラスがパッケージ内に置かれるようにアプリケーションを更新するまでは、OC4J スタンドアロンを JDK 1.3.1 環境に戻すことでこの問題を防ぐことができます。JDK 1.3.x は、完全な Oracle Application Server 10.1.2 環境ではサポートされないので注意してください。

注意 :

- javac の -source コンパイラ・オプションを使用すると、JDK 1.3.1 コードを透過的に JDK 1.4 コンパイラで処理できますが、このオプションはクラスがパッケージ内に存在しない場合の問題には対処していません。
- OC4J では、JDK 1.3.1 および JDK 1.4 コンパイラのみがサポートおよび動作保証されています。<java-compiler> 要素を server.xml ファイルに追加すると、別のコンパイラを指定することができ、これによってクラスがパッケージ内に存在しない場合の問題を回避できる可能性があります。ただし、OC4J での他のコンパイラの使用は、オラクル社により動作保証およびサポートされていません。（さらに、Oracle Application Server 環境では、server.xml ファイルを直接更新しないでください。かわりに Oracle Enterprise Manager 10*g* を使用してください。）

クラスがパッケージ内に存在しない場合の問題や、その他の JDK 1.4 の互換性の問題の詳細は、次の Web サイトを参照してください。

http://java.sun.com/j2se/1.4/compatibility.html

特に、リンク「Incompatibilities Between Java 2 Platform, Standard Edition, v1.4.0 and v1.3」をクリックしてください。

追加の Oracle 機能

次の各項で、OC4J でのサーブレットの開発および実行時に考慮する必要のある、主に Oracle 固有の追加機能について説明します。

- OC4J のロギング
- サーブレットのデバッグ
- Oracle JDeveloper によるサーブレット開発のサポート
- オープン・ソース・フレームワークの OC4J サポートの概要

OC4J のロギング

次の各項で、OC4J のロギング機能の概要を説明します。

- OC4J のログ
- Oracle Diagnostic Logging とテキストベースのロギング
- 追加の Oracle Application Server ログ・ファイル

注意： この項で説明するロギング機能は、OC4J サーバーからのログ・メッセージ用の機能です。OC4J では、Apache Jakarta プロジェクトなどが提供しているオープン・ソース・フレームワークおよびユーティリティを使用することも可能です。これには、ユーザー・コードにログ文を挿入するために使用可能な補完テクノロジである log4j も含まれます。A-7 ページの「OC4J における Jakarta log4j の構成および使用」を参照してください。

OC4J のログ

OC4J では複数のログを使用できます。これらはサーブレット固有のものではないため、ここでは説明しませんが、この項ではサマリーおよび該当する相互参照を示します。各ログについて、テキストベースのロギングまたは ODL ロギングのいずれかを選択できます。（次項の「Oracle Diagnostic Logging とテキストベースのロギング」を参照してください。）ODL の場合、ログ・ファイル名の形式は必ず log*N*.xml（*N* は整数）です。テキストベースのロギングの場合、ユーザーがログ・ファイル名を指定する必要があります。

各ログには、該当する OC4J 構成ファイル内にテキストベースのロギングを有効にする構成要素、および ODL ロギングを有効にする別の要素が存在します。ロギング構成要素の存在により、関連付けられたロギング・タイプが有効になります。

OC4J では、次のログをサポートしています。

- アプリケーション・ログ

  デプロイされた各アプリケーション用のログが用意されています。orion-application.xml で構成されます。テキストベースのロギングの場合、一般的な名前は application.log です。

- グローバル・アプリケーション・ログ

  デフォルト・アプリケーションを含めたすべてのアプリケーションのグローバル・ロギング用のログが用意されています。OC4J のグローバル application.xml ファイルで構成されます。テキストベースのロギングの場合、一般的な名前は global-application.log です。

- JMS ログ

  Java Message Service（JMS）機能のログが用意されています。jms.xml で構成されます。テキストベースのロギングの場合、一般的な名前は jms.log です。

- RMI ログ

  Remote Method Invocation 機能のログが用意されています。rmi.xml で構成されます。テキストベースのロギングの場合、一般的な名前は rmi.log です。

- サーバー・ログ

  サーバー全体のログが用意されています。server.xml で構成されます。テキストベースのロギングの場合、一般的な名前は server.log です。

- Web サイト・アクセス・ログ

  Web サイトのアクセス・ログ（各 Web サイトごとに 1 つのログ・ファイルでサイトの全アクセスのログを記録）が用意されています。Web サイトの XML ファイルで構成されます。テキストベースのロギングの場合、一般的な名前は http-access.log です。

> **注意：** Web サイトのアクセス・ロギングでは、1 種類のロギングのみ使用可能で、両方を使用することはできません。

Web アクセス・ログの構成方法は、このマニュアルで説明されています。6-19 ページの「Web サイト XML ファイルの要素の説明」で、<web-site> 要素の <access-log> および <odl-access-log> サブ要素に関する説明を参照してください。

その他の OC4J ファイルのロギングを有効にする方法は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

Oracle Diagnostic Logging とテキストベースのロギング

前項「OC4J のログ」で示した各ログで、Oracle Diagnostic Logging（ODL）を使用できます。ODL を使用すると、テキストベースのロギングに比べ、いくつかの利点があります。

ODL は、OC4J の全コンポーネントにわたる標準化されたロギングを提供します。また、ログ・ファイルが XML 形式で作成され、リポジトリにロードしてレポートおよび表示に使用できます。たとえば、ODL ログを Oracle Enterprise Manager 10*g* で表示できます。

また、ODL を使用すると、ログ・ファイルのサイズおよび数の管理が容易です。多くの場合、テキストベースのロギングを使用していると、OC4J サーバーを定期的にシャットダウンし、手動でファイルをクリーン・アップする必要があります。

Web サイトのアクセス・ログ・ファイル用に ODL ロギングを有効にし、構成するには、Web サイトの XML ファイル内で、<web-site> 要素の <odl-access-log> サブ要素を使用します。テキストベースのロギングを有効にし、構成するには、かわりに <web-site> 要素の <access-log> サブ要素を使用します（ただし、どちらの要素の場合も、Web サイトの XML ファイル内で特定の Web アプリケーションの <web-app> 要素に access-log="false" を設定すると、その Web アプリケーションに対してオーバーライドされます）。

その他の OC4J ログに ODL ロギングを使用する場合、該当する XML 構成ファイル内の <log> 要素の <odl> サブ要素を使用します。テキストベースのロギングを使用するには、かわりに <log> 要素の <file> サブ要素を使用します。

> **注意：** Web サイトのアクセス・ログでは、一般的に標準 XLF または CLF 形式（拡張ログ・ファイル形式または共通ログ・ファイル形式）を使用します。ユーザーは、1 日の中の特定の時間または 1 か月の中の特定の日など、指定した期間によってファイルを分割できます。ただし、ODL の Web サイト・ログは XLF または CLF 形式をサポートしておらず、期間によってファイルを分割できません。ODL ログ・ファイルの最大サイズに達すると、自動的に新しいファイルが作成されます。（ログ・ファイル名は log1.xml、log2.xml などとなります。）

ODL に関する詳細な情報は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

追加の Oracle Application Server ログ・ファイル

Oracle Application Server では、前述の OC4J ログ・ファイル以外に、次のログ・ファイルをサポートしています。

- OPMN ログ・ファイル（各 OC4J インスタンス、Oracle Process Management and Notification 機能用に 1 つずつのログ・ファイル）
- ons.log（OPMN 通知システム・ログ。opmn.xml で構成）
- ipm.log（OPMN プロセス管理ログ。opmn.xml で構成）

OPMN は、アプリケーション・サーバー・インスタンス内で Oracle HTTP Server および OC4J プロセスを管理します。

Oracle Application Server ログ・ファイル管理の詳細は、『Oracle Application Server 管理者ガイド』を参照してください。

サーブレットのデバッグ

この項では、サーブレット開発者に関係のあるデバッグ機能および考慮事項と、追加情報の相互参照を示します。次の項が含まれます。

- OC4J のデバッグ・フラグ
- OC4J のデバッグ・フラグの設定
- Oracle Application Server でのデバッグのタイミング上の考慮事項
- JDeveloper およびその他の IDE を使用したデバッグ

OC4J のデバッグ・フラグ

OC4J では、サブシステムでのデバッグ出力を可能にする様々なフラグをサポートしています。

次に、HTTP のデバッグ・フラグを示します。

- `http.error.debug`（すべての HTTP エラーに使用。これを使用しない場合、一部のエラーはレポートされずに処理されることがあります）
- `http.cluster.debug`（HTTP のクラスタリングおよびセッション永続性に関するデバッグ文に使用）
- `http.session.debug`（HTTP セッション・エラーおよびライフ・サイクル文に使用）
- `http.request.debug`（HTTP リクエスト・ストリームからの情報に使用）
- `http.redirect.debug`（HTTP リダイレクトに関する情報に使用）
- `debug.http.contentLength`（明示的なコンテンツ長コールおよび余分な `sendError` 情報の出力に使用）
- `http.virtualdirectory.debug`（起動時の強制仮想ディレクトリ・マッピングの出力に使用）
- `http.method.trace.allow`（`traceHTTP()` メソッドを有効にするために使用）

次に、AJP のデバッグ・フラグを示します（Oracle HTTP Server とともに Oracle Application Server を使用している場合のみ）。

- `ajp.debug`（受信 AJP ストリームの出力に使用）
- `ajp.io.debug`（サーバーからの AJP レスポンスの出力に使用）

AJP フラグはユーザー・フレンドリな出力を生成しませんが、一部の AJP に関する問題のデバッグには必要です。

次に、JDBC のデバッグ・フラグを示します。

- `datasource.verbose`（データ・ソースおよびデータベース接続の作成に関する情報に使用）

- `jdbc.debug`（JDBC コールに関する詳細情報に使用）

次に、EJB のデバッグ・フラグを示します。

- `ejb.cluster.debug`（EJB クラスタリングに関する情報に使用）

次に、RMI のデバッグ・フラグを示します。

- `rmi.debug`（Remote Method Invocation に関する情報に使用）
- `rmi.verbose`（RMI コールに関する詳細情報に使用）

次に、Web サービスのデバッグ・フラグを示します。

- `ws.debug`（Web サービスに関する情報に使用）

OC4J のデバッグ・フラグの設定

デバッグ・フラグは、次のような Java オプション設定によって有効にします。

```
-Dhttp.session.debug=true
```

OC4J スタンドアロンを使用している場合、OC4J の起動時に Java コマンドラインでオプション設定を指定します。Oracle Application Server 環境では、Oracle Enterprise Manager 10*g* を使用します。OC4J インスタンスの Application Server Control コンソール「サーバー・プロパティ」ページの「コマンドライン・オプション」の下にある「Java オプション」フィールドでオプション設定を指定します。このページに移動するには、OC4J インスタンスの「管理」ページにある「インスタンス・プロパティ」の下の**「サーバー・プロパティ」**を選択します。7-6 ページの「Application Server Control コンソールの OC4J の「管理」ページ」を参照してください。詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

Oracle Application Server でのデバッグのタイミング上の考慮事項

OPMN の動作方法により、Oracle Application Server 環境でデバッグを行う際に、タイミングの問題を考慮する必要があります。特に、デバッグによってプロセスが中断された場合、中断がタイムアウトを超過すると、OPMN はそのプロセスを終了します。

これに対処するには、`opmn.xml` ファイルの `<timeout>` 要素を使用して、適切なタイムアウト値を設定する必要があります。

`opmn.xml`、特に Oracle Application Server の起動と停止の詳細は、『Oracle Application Server 管理者ガイド』を参照してください。

JDeveloper およびその他の IDE を使用したデバッグ

Oracle JDeveloper を開発環境に使用している場合、そのデバッグ機能を利用できます。

デバッグ用に、JDeveloper では JSP ページ、サーブレットおよびその他の Java ソース・ファイルのローカルおよびリモート・デバッグを提供しています。まず、JDeveloper 内のソース・ファイルにブレーク・ポイントを設定し、選択したソースのデバッグ・セッションを実行します。サーブレットなどのアプリケーションを JDeveloper でデバッグする場合、実行フローを完全に制御でき、変数値の表示および変更を行い、さらにクラス・インスタンス・カウントやメモリーの使用状況の表示など、アプリケーションの高度なパフォーマンスの監視を実行できます。JDeveloper は、アプリケーションからのコールを他のソース・ファイルまで追跡したり、使用不可のクラス・ソース用のスタブ・クラスの生成を提供します。デバッグするコードが起動され、JDeveloper のデバッガが付加された後は、リモート・デバッグはローカル・デバッグと似ています。

サーブレット開発用の JDeveloper の機能の概要は、2-18 ページの「Oracle JDeveloper によるサーブレット開発のサポート」を参照してください。

注意： 他の主要な IDE ベンダーは、OC4J との透過的な統合を可能にするプラグイン・モジュールを提供しています。これらを使用すると、開発者は、OC4J で実行されている J2EE アプリケーションを、直接 IDE 内から構築、デプロイおよびデバッグできます。詳細は、次の Web サイトを参照してください。

http://www.oracle.com/technology/products/ias/oracleas_partners.html

（Oracle Technology Network の会員登録が必要ですが、登録は無料です。）

Oracle JDeveloper によるサーブレット開発のサポート

ビジュアルな Java のプログラミング・ツールは通常、サーブレットのコーディングをサポートしています。特に、Oracle JDeveloper には、サーブレットの開発をサポートする次の機能が含まれています。

- サーブレット・コードの生成に役立つウィザード。
- アプリケーションの全開発サイクル（サーブレットの編集、デバッグおよび実行）をサポートする OC4J サーブレット・コンテナの統合機能。
- デプロイ済サーブレットのデバッグ。
- Web Bean と呼ばれるデータ対応および Web 対応の JavaBeans の拡張セット。
- カスタム JavaBeans を取り込むためのサポート。
- Oracle Application Development Framework（Oracle ADF）Business Components に依存するサーブレット・アプリケーションのデプロイ・オプション。

2-17 ページの「JDeveloper およびその他の IDE を使用したデバッグ」も参照してください。

JDeveloper に関する一般情報は、JDeveloper のオンライン・ヘルプか、Oracle Technology Network の次の Web サイトを参照してください。

http://www.oracle.com/technology/products/jdev/content.html

オープン・ソース・フレームワークの OC4J サポートの概要

OC4J では、いくつかの一般的なオープン・ソース・ユーティリティおよびフレームワークをサポートしています。Oracle Application Server 10*g* リリース 2（10.1.2）に関し、このマニュアルでは、特に次の 2 つに関するサポートについて説明します。

- Jakarta Struts
- Jakarta log4j

OC4J スタンドアロン環境での、これらのオープン・ソース・ユーティリティの構成および使用について説明しています。付録 A「オープン・ソース・フレームワークおよびユーティリティ」を参照してください。

サーブレットの起動

サーブレットは、クライアントからサーブレットにリクエストが届くと、コンテナにより起動されます。クライアント・リクエストは、Web ブラウザまたは Java クライアント・アプリケーションから、リクエスト転送メカニズムを使用してアプリケーションの別のサーブレットから、あるいはサーバーのリモート・オブジェクトから送信されます。サーブレットは、URL マッピングを通じてリクエストされます。

次の各項で、開発またはテスト環境でクラス名によってサーブレットを起動するための OC4J のいくつかの特別な機能を含め、サーブレットの起動について説明します。

- URL 構成要素のサマリー

- OC4J 開発時におけるクラス名によるサーブレットの起動
- Oracle Application Server 本番環境でのサーブレットの起動
- OC4J スタンドアロン環境でのサーブレットの起動

URL 構成要素のサマリー

サーブレットの起動について説明する前に、URL の構成要素のサマリーを示します。次に、一般的な構成を示します（ただし、通常は *pathinfo* は空です）。

protocol://*host*:*port*/*contextpath*/*servletpath*/*pathinfo*

また、デリミタの後に情報を追加することも可能です。たとえば、疑問符（?）デリミタの後にリクエスト・パラメータ設定を使用できます。

protocol://*host*:*port*/*contextpath*/*servletpath*/*pathinfo*?*param*=*value*

表 2-2 に、一般的な構成要素を示します。

表 2-2 URL の構成要素

構成要素	説明
protocol	Web アプリケーションの起動時に使用するネットワーク・プロトコル。たとえば、`http`、`https`、`ftp` および `ormi`（EJB で使用）などがあります。
host	Web アプリケーションが稼働しているサーバーのネットワーク名。Web クライアントがアプリケーション・サーバーと同じシステム上に存在する場合は、`localhost` を使用可能です。それ以外の場合は、ホスト名を使用します（たとえば UNIX システムの場合は、`/etc/hosts` に定義されています）。次に例を示します。 `www.example.com`
port	Web サーバーがリスニングしているポート。URL によってポートを指定しないと、HTTP プロトコルの場合はポート 80、HTTPS の場合はポート 443 が使用されます。 OC4J の場合、ポート番号は Web サイトの XML ファイル内の `<web-site>` 要素の `port` 属性で指定されます。たとえば、Oracle Application Server 環境の場合は `default-web-site.xml`、OC4J スタンドアロンの場合は `http-web-site.xml` です。（各ポートについて、`<web-site>` 要素の `protocol` 属性に基づく 1 つの関連付けられたプロトコルが必要です。）

contextpath	サーブレット・コンテキストの指定されたルート・パス。アプリケーションをデプロイする際、コンテキスト・パスを指定します。OC4J の場合、指定されたコンテキスト・パスは、Web サイトの XML ファイルの `<web-app>` 要素（`<web-site>` のサブ要素）の `root` 属性の設定に反映されます。
	各サーブレット・コンテキストは、サーバー・ファイル・システム内のディレクトリ・パスに関連付けられています。
	また、`<web-app>` 要素は、`application` 属性によって J2EE アプリケーション名（および EAR ファイル名）、`name` 属性によって Web モジュール名（および WAR ファイル名）も示します。J2EE アプリケーション名、Web モジュール名およびコンテキスト・パスは、すべてこのように一緒にマップされます。次に例を示します。
	`<web-app application="ojspdemos" name="ojspdemos-web"`
	`root="/ojspdemos" />`

表 2-2 URL の構成要素（続き）

<table>
<tr><th>構成要素</th><th>説明</th></tr>
<tr><td>servletpath</td><td>コンテキスト・パスより後で、起動するサーブレットを指定するパス。アプリケーションの web.xml ファイル内で標準マッピングによってサーブレット・パスを指定します。サーブレット・クラスは、<servlet> 要素の <servlet-class> および <servlet-name> サブ要素によって任意のサーブレット名にマップされます。サーブレット名は、<servlet-mapping> 要素の <servlet-name> および <url-pattern> サブ要素によってサーブレット・パスにマップされます。（1 つのサーブレット・クラスを複数のサーブレット名および複数のサーブレット・パスにマップできます。）次に例を示します。
<pre>
<web-app>
  ...
  <servlet>
    <servlet-name>logout</servlet-name>
    <servlet-class>
       oracle.security.jazn.samples.http.Logout
    </servlet-class>
  </servlet>
  ...
  <servlet-mapping>
    <servlet-name>logout</servlet-name>
    <url-pattern>/logout/*</url-pattern>
  </servlet-mapping>
  ...
</web-app>
</pre></td></tr>
<tr><td>pathinfo</td><td>（通常はこれは空です。）コンテキスト・パスおよびサーブレット・パスより後に、URL に HTTP リクエスト・オブジェクトを介してサーブレットに提供される追加情報を含めることが可能です。この情報は、サーブレットによって理解されるとみなされます。この情報は、疑問符などのデリミタに続くリクエスト・パラメータ設定またはその他の URL の構成要素とは異なるものです。デリミタは、パス情報の後に続きます。</td></tr>
</table>

注意： `<servlet-name>` 要素で指定される名前は、サーブレットのリクエスト・ディスパッチャを使用する場合にサーブレット・コンテキストの `getNamedDispatcher()` メソッドに入力する名前です。

表 2-2 で説明した OC4J の構成要素および属性の詳細は、6-19 ページの「Web サイト XML ファイルの要素の説明」を参照してください。web.xml ファイルの要素および属性の詳細は、サーブレット仕様を参照してください。

次の URL を例にして説明します。

```
http://www.example.com:8888/foo/bar/mypath/MyServlet/info1/info2?user=Amy
```

クライアント・ブラウザが提供した URL に基づいてサーブレットを起動する際、サーブレット・コンテナは次のステップを実行します。

1. URL のポート番号の後の部分全体を検証し、コンテキスト・パスを認識するためにコンテナ自体の構成設定（Web サイトの XML ファイル内など）を検証して、URL のどの部分がコンテキスト・パスであるかを決定します。

   この例では、コンテキスト・パスが /foo/bar であるとします。

2. URL のコンテキスト・パスの後の部分全体を検証し、web.xml ファイル内のサーブレット・マッピングで認識済のサーブレット・パスを検証して、URL のどの部分がサーブレット・パスであるかを決定します。

この時点で、サーブレットは起動できます。サーブレット・コンテナは、サーブレット・パスより後ろの情報は使用しません。

この例では、サーブレット・パスが /mypath/MyServlet であるとします。

3. URL の中で、サーブレット・パスより後、URL デリミタ（この例ではリクエスト・パラメータ設定を区切る「?」）より前に残っている部分がある場合、URL のその部分は追加情報とみなされ、HTTP リクエスト・オブジェクトを介してサーブレットに渡されます。

   この例では、追加パス情報が /info1/info2 であるとします。

コンテキスト・パス、サーブレット・パスおよびその他のどんなパス情報でも、1 つ以上のスラッシュが間にある複合構成要素である可能性があります。前の例がこれに該当します。多くの場合、コンテキスト・パスは foo のみのように単純で、サーブレット・パスも MyServlet のみのように単純であり、パス情報も単純なことがよくあります。ただし、URL を見ただけでは、どの部分がコンテキスト・パス、サーブレット・パスまたはその他のパス情報（存在する場合）であるかはわかりません。これを判断するには、Web サイトの XML ファイルおよび web.xml ファイル内の構成を検証する必要があります。

注意：

- 定義済のポート、そのマップ先リスナー、およびこれらの設定の変更方法に関する詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』または、『Oracle Application Server Containers for J2EE スタンドアロン・ユーザーズ・ガイド』を参照してください。
- Cookie の名前は、ホスト名、ポート番号およびパス（デフォルトではコンテキスト・パスのみだが、サーブレット・パスも含めることが可能）に基づいて決定されます。
- サーブレット・コンテキストおよびコンテキスト・パスの概念は、サーブレット仕様のバージョン 2.2 で導入されました。
- コンテキスト・パス、サーブレット・パスおよびパス情報は、HTTP リクエスト・オブジェクトの getContextPath()、getServletPath() および getPathInfo() メソッドを使用して取得できます。

OC4J 開発時におけるクラス名によるサーブレットの起動

OC4J の開発環境またはテスト環境には、クラス名でサーブレットを起動する簡易的なメカニズムがあります。セキュリティ上の理由から、このメカニズムはアプリケーションの開発時にのみ使用してください。OC4J のデフォルトの構成では無効になっています。

global-web-application.xml ファイルまたは orion-web.xml ファイルの <orion-web-app> 要素内の servlet-webdir 属性によって、クラス名によるサーブレット起動に使用される特別な URL コンポーネントが定義されます。この URL コンポーネントは URL のコンテキスト・パスに続き、この URL コンポーネントに続くものは、適切なサーブレット・コンテキスト内のサーブレット・クラス名（該当するパッケージ情報も含め）とみなされます。URL には、サーブレット・パスのかわりにサーブレット・クラス名が使用されます。（サーブレット・パスは servlet-webdir 値で、サーブレット自体として動作し、実行するサーブレットのクラス名がパス情報として取得されます。）

一般に、すべてのアプリケーションでは、クラス名による起動に対する OC4J の動作は、アプリケーションの orion-web.xml ファイル内の servlet-webdir 設定で決まります（設定されている場合）。ただし、次の点に注意してください。

- global-web-application.xml ファイル内の servlet-webdir 要素の任意の設定は、デフォルト値として使用されます（これは、global-web-application.xml 内の構成設定全般に当てはまります）。ただし、global-web-application.xml に servlet-webdir が設定されていない場合、デフォルト値は ""（空の引用符）です。この設定では、クラス名による起動は無効です。デフォルト値が使用されるのは、アプリケーションのデプロイ時に orion-web.xml が提供されない場合や、servlet-webdir が設定されていない場合です。

- クラス名によるサーブレットの起動は、次のいずれかの方法で無効にできます。
  - システム・プロパティ `http.webdir.enable` の値を `false` に設定。この結果、`servlet-webdir` 設定は無視されます。
  - `global-web-application.xml` または `orion-web.xml` を使用して、`servlet-webdir` の値を `""`（空の引用符）に設定。

OC4J システム・プロパティの詳細は、『Oracle Application Server Containers for J2EE スタンドアロン・ユーザーズ・ガイド』を参照してください。Oracle Application Server 環境については、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

次に示す URL は、クラス名によりサーブレット `SessionServlet` を起動します。設定を `servlet-webdir="/servlet/"` と仮定します。この例では、`SessionServlet` がパッケージ `foo.bar` に存在し、OC4J のデフォルトの Web アプリケーションで実行されることを前提としています。また、コンテキスト・パスは「`/`」（OC4J スタンドアロンのデフォルトの Web アプリケーションのデフォルト）です。

```
http://www.example.com:8888/servlet/foo.bar.SessionServlet
```

このメカニズムは、デフォルトの Web アプリケーションのみではなく、すべてのサーブレット・コンテキストに適用されます。たとえば、コンテキスト・パスが `foo` の場合、クラス名で起動する URL は、次のようになります。

```
http://www.example.com:8888/foo/servlet/foo.bar.SessionServlet
```

> **重要：** クラス名によるサーブレットの起動を許可すると、重大なセキュリティ上のリスクが発生する可能性があります。本番環境では、OC4J をこのモードで動作するよう構成しないでください。詳細は、2-38 ページの「その他のセキュリティに関する考慮事項」を参照してください。

Oracle Application Server 本番環境でのサーブレットの起動

次の各項で、Oracle Application Server 環境でのサーブレット起動に関する Oracle HTTP Server および OC4J の機能について説明します。

- Oracle Application Server での起動の主要な機能
- OC4J による認識可能なフロントエンド・ホストの使用

Oracle Application Server での起動の主要な機能

Oracle Application Server 本番環境では、OC4J は必ず Oracle HTTP Server を通じてアクセスされます。Oracle HTTP Server は OC4J との通信に AJP（Apache JServ Protocol）を使用しますが、これはエンド・ユーザーには表示されません。

サーブレットがリクエストされると、OC4J サーブレット・コンテナは 2-19 ページの「URL 構成要素のサマリー」で説明されているように URL を解析します。

使用するポート番号は、`default-web-site.xml` ファイルの `<web-site>` 要素によって AJP プロトコルにマップされます。（これは一般的な名前ですが、Web サイトの XML ファイル名は `server.xml` ファイルの設定に基づいており、変更可能です。）ポート・マッピングは、`<web-site>` 要素の `port` および `protocol` 属性で定義され、`port` は指定どおり、`protocol` は `ajp13` に設定されます。デフォルトでは、Oracle Application Server Web Cache を有効にした Oracle HTTP Server によるアクセスには、ポート 7777 が使用されます。

Enterprise Manager を使用してアプリケーションをデプロイすると、URL のマッピングの指定を求められます。指定するマッピングによって、`mod_oc4j.conf` での新しい OC4J のマウント・ポイントが決まります。たとえば、URL マッピングに `/mypath` を指定すると、これが Web アプリケーションのコンテキスト・パスとなり、新規 OC4J マウント・ポイントとして定義されます。そして、次の URL でサーブレットを起動します。

```
http://www.example.com:7777/mypath/MyServlet
```

Enterprise Manager の EAR および WAR デプロイ・ページの詳細は、7-4 ページの「Application Server Control コンソールの「アプリケーションのデプロイ」ページ」および 7-5 ページの「Application Server Control コンソールの「Web アプリケーションのデプロイ」ページ」を参照してください。

Oracle Application Server へのデプロイの概要は、5-37 ページの「Oracle Application Server での OC4J のデプロイ」を参照してください。詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。Enterprise Manager に関する一般的な情報は、『Oracle Enterprise Manager 概要』を参照してください。

Oracle HTTP Server の構成、マウント・ポイントおよび `mod_oc4j.conf` ファイルの詳細は、『Oracle HTTP Server 管理者ガイド』を参照してください。

OC4J による認識可能なフロントエンド・ホストの使用

`default-web-site.xml` ファイル（または他の Web サイトの XML ファイル）にも、サーブレット起動に関連のある追加要素があります。`<web-site>` 要素のサブ要素 `<frontend>` は、HTTP クライアントによって表示される Web サイトの、認識可能なフロントエンド・ホストおよびポートを指定できます。サイトがロード・バランサやファイアウォールの後ろにある場合は、`<frontend>` を指定して、URL リライティングなどの機能のための Web アプリケーションに適切な情報を与える必要があります。属性は、フロントエンド・サーバーのホスト名（`www.example.com` など）に対しては `host` を、フロントエンド・サーバーのポート番号（`8080` など）に対しては `port` を指定します。このフロントエンド情報によって、実際にアプリケーションを実行中のバックエンド・サーバーは、URL リライティングでサーバー自体ではなく参照先の `www.example.com` を認識します。したがって、後続のリクエストは、バックエンドに直接アクセスすることなく、フロントエンドから適切に受信されます。

また、指定したフロントエンドの `host` および `port` 設定、および HTTP リクエスト・オブジェクトの `getServerName()` または `getServerPort()` メソッドをコールして受信した値も、サーブレットに反映されます。

OC4J スタンドアロン環境でのサーブレットの起動

OC4J スタンドアロンの Web サイトでは、Oracle HTTP Server と AJP を経由せずに、HTTP プロトコルを使用します。このサイトは、`http-web-site.xml` ファイルの設定に従って構成されます。（これは一般的な名前ですが、Web サイトの XML ファイル名は `server.xml` ファイルの設定に基づいており、変更可能です。）

サーブレットがリクエストされると、OC4J サーブレット・コンテナは 2-19 ページの「URL 構成要素のサマリー」で説明されているように URL を解析します。

使用するポート番号は、`http-web-site.xml` ファイル（または、必要に応じて Web サイトの XML ファイル）の `<web-site>` 要素によって HTTP プロトコルにマップされます。ポート・マッピングは、`<web-site>` 要素の `port` および `protocol` 属性で定義され、`port` は指定どおり、`protocol` は `http` に設定されます。デフォルトでは、独自の Web リスナー経由で OC4J にアクセスするために、ポート 8888 が設定されています。

OC4J スタンドアロンでは、OC4J のデフォルト Web アプリケーションにデプロイされているアプリケーションに HTTP プロトコルを使用するためのデフォルトのコンテキスト・パスは「`/`」です。次に例を示します。

```
http://www.example.com:8888/MyServlet
```

デフォルトの Web アプリケーションを使用しない場合、アプリケーションのデプロイ時にコンテキスト・パスを指定します。これを行うには、`admin.jar` ユーティリティを使用するか、`http-web-site.xml` ファイルを手動でデプロイおよび編集します（この方法はお薦めしません）。OC4J スタンドアロンのデプロイについては 5-22 ページの「OC4J スタンドアロンへのデプロイ方法」で説明していますが、完全な情報は、『Oracle Application Server Containers for J2EE スタンドアロン・ユーザーズ・ガイド』を参照してください。このマニュアルには、OC4J のポート設定とその他のデフォルト設定に関する情報も含まれています。

たとえば、`/mypath` をコンテキスト・パスに指定した場合、次の URL でサーブレットを起動します。

```
http://www.example.com:7777/mypath/MyServlet
```

サーブレットのセッション

サーブレット・セッションの概要は、1-5 ページの「サーブレット・セッションの概要」を参照してください。次の各項で、詳細に説明し、例を示します。

- セッション・トラッキング
- HttpSession インタフェースの機能
- セッションのキャンセル
- 分散アプリケーションでのセッションのレプリケーション
- セッション・サーブレットの例

重要： 新しい Web モジュールをアクティブな OC4J インスタンスにデプロイすると、デフォルトでは、サーバー・インスタンス内で実行されているすべての Web アプリケーションの HTTP セッションが失われます。クラスタリングされていない OC4J インスタンスでは、各 Web アプリケーションの orion-web.xml ファイルで永続性ディレクトリを定義することにより、この問題を回避できます（詳細は、6-2 ページの「global-web-application.xml および orion-web.xml の要素の説明」に記載されている <orion-web-app> の persistence-path 属性を参照してください）。既存の HTTP セッションは、アプリケーション・デプロイメント全体で、この一時ロケーションに格納されます。

この機能は、クラスタ環境内の OC4J インスタンスには使用できません。クラスタ環境でセッション状態の問題を解決するガイドラインは、『Oracle Application Server 高可用性ガイド』を参照してください。

セッション・トラッキング

この項では、サーブレットのセッション・トラッキングと機能の概要を説明し、次に OC4J 実装を説明します。

セッション・トラッキングの概要

HTTP プロトコルは、設計上ステートレスです。これは、単にリクエストを取得し、簡単な処理を行って結果を出力した後に消滅するような、ステートレスなサーブレットについては問題ありません。ただし、ほとんどのサーバー・サイド・アプリケーションでは、状態情報を保持してクライアントとの対話を続ける必要があります。このような場合の最も一般的な例は、ショッピング・カートのアプリケーションです。クライアント・ユーザーは、同じブラウザから何度かサーバーにアクセスして、いくつかの Web ページを表示します。クライアント・ユーザーは、Web サイトで販売されているいくつかのアイテムの購入を決め、**「製品の購入」**ボタンをクリックします。各トランザクションがステートレスなサーバー・サイド・オブジェクトで処理されており、各リクエストに対してクライアントから識別情報が提供されない場合、クライアントからの複数の HTTP リクエストに渡ってショッピング・カートの中身を維持することはできません。このケースでは、クライアントをサーバー・セッションに関連付ける手段がないため、ステートレスなトランザクション・データを永続性のある記憶域に書き込んでも問題は解決されません。

セッション・トラッキングでは、ID 番号を使用してユーザー・セッションを識別し、リクエストをユーザー・セッションに関連付けます。このプロセスは通常 Cookie または URL リライティングによって実行されます。

OC4J のサーブレット・コンテナは、サーブレット仕様に従い、HTTP セッション・オブジェクト（javax.servlet.http.HttpSession インタフェースを実装するクラスのインスタンス）によってセッション・トラッキングを実装します。

サーブレットで HTTP セッション・オブジェクトを作成（リクエスト・オブジェクト getSession() メソッドを使用）する場合、クライアントの相互作用はステートフルであるとみなされます。

HTTP セッション・オブジェクトのスコープに含まれるのは、Web アプリケーションのみです。セッション・オブジェクトを使用してアプリケーション間でデータを共有することはできません。また、セッション・オブジェクトを使用して同一アプリケーションの異なるクライアント間でデータを共有することもできません。1 つのアプリケーションの 1 つのクライアント当たり 1 つの HTTP セッション・オブジェクトが存在します。

注意： クライアントまたはアプリケーション間で情報を共有する場合、保護、トランザクションの安全性、およびデータベースによるバックアップが必要であれば、その永続データをデータベースに格納できます。かわりに、永続情報をファイル・システムやリモート・オブジェクトに保存することもできます。

Cookie

HTTP プロトコルでステートフルを実現するための方法は多数あります。最も広く使用されている方法は、Cookie を使用してサーバーとクライアント間で識別子を送信し、ステートフル・サーブレットを使用してセッション・オブジェクトを維持する方法です。セッション・オブジェクトは、関連付けられたキー（Java 文字列）とともに値（Java オブジェクト）を格納するディクショナリにすぎません。

次に、Cookie の使用方法を示します。

1. サーバー（コンテナ）では、セッション作成後のステートフル・サーブレットからの最初のレスポンスとともに、セッション識別子を含む Cookie をクライアントに返信します。多くの場合、Cookie にはその他の少量の有用な情報（すべて 4 KB 未満）も含まれています。コンテナは、`JSESSIONID` という名前の Cookie を HTTP レスポンス・ヘッダーで送信します。
2. その後、同じ Web クライアント・セッションから後続のリクエストを受信するたびに、クライアントはその Cookie をリクエストの一部としてサーバーに返信します（クライアントが Cookie をサポートしている場合)。サーバーでは Cookie の値を使用してセッションの状態情報をルックアップし、サーブレットに渡します。
3. コンテナは、後続のレスポンスとともに、更新済の Cookie をクライアントに返信します。

Cookie を送信するためのサーブレット・コードは不要です。送信はコンテナで処理されます。Cookie のサーバーへの返信は、ユーザーが Cookie を無効化していないかぎり、Web ブラウザで自動的に処理されます。

コンテナは、Cookie をセッションの維持に使用します。サーブレットは、`HttpServletRequest` オブジェクトの `getCookies()` メソッドを使用して Cookie を取り出し、`javax.servlet.http.Cookie` オブジェクトのアクセッサ・メソッドを使用して、Cookie 属性を調べることができます。

URL リライティング

Cookie の使用にかわるものとして、レスポンス・オブジェクトの `encodeURL()` メソッドを使用する URL リライティングがあります。この場合、セッション ID は、リクエストの URL パス内にエンコードされます。URL リライティングの例は、2-30 ページの「セッション・サーブレットの例」を参照してください。

パス・パラメータの名前は、`jsessionid` です。次に例を示します。

http://*host*:*port*/myapp/index.html?jsessionid=6789

Cookie の機能と同様に、サーバーでは再書込みされた URL の値を使用してセッションの状態情報をルックアップし、サーブレットに渡します。

通常、Cookie は有効になっていますが、セッション・トラッキングを確実に実行する唯一の方法は、サーブレットで `encodeURL()` を使用するか、またはリダイレクト用の `encodeRedirectURL()` を使用することです。

注意： サーブレット仕様に従い、Cookie が有効な場合、`encodeURL()` メソッドと `encodeRedirectURL()` メソッドのコールによってアクションは実行されません。

他のセッション・トラッキング方法

従来は、クライアントとサーバー・セッションを関連付けるために、サーバーの隠しフォーム・フィールドや、追加情報を格納するためのユーザー認証メカニズムなど、他のテクニックが使用されてきました。これらのテクニックを OC4J アプリケーションで使用することはお薦めしません。このような方法には、パフォーマンスが悪化したり機密性が失われるなど多くのデメリットがあるためです。

OC4J でのセッション・トラッキング

OC4J でのセッション・トラッキングでは、サーブレット・コンテナは、最初に Cookie を使用してトラッキングを試行します。Cookie が無効な場合、セッション・トラッキングの維持に使用できるのは、レスポンス・オブジェクトの `encodeURL()` メソッドまたはリダイレクト用の `encodeRedirectURL()` メソッドのみです。Cookie が無効化されている場合は、サーブレットに `encodeURL()` コールまたは `encodeRedirectURL()` コールを含める必要があります。

セッション Cookie の使用を無効化するには、`global-web-application.xml` ファイルまたは `orion-web.xml` ファイルで次のように設定します。

```
<session-tracking cookies="disabled" ... >
   ...
</session-tracking>
```

デフォルトでは、Cookie は有効です。

注意：

- OC4J では、サーブレット・コンテナでセッション ID を自動的に URL にエンコードする、自動エンコーディングはサポートしていません。これは、コストのかかる標準外の処理です。
- Cookie が適切に動作している場合、`encodeURL()` コールまたは `encodeRedirectURL()` コールは、セッション ID を URL にエンコードすることはありません。
- サーブレット 2.0 の `encodeUrl()` メソッドは使用できないため、`encodeURL()` メソッド（大文字化に注意）に置換されました。

HttpSession インタフェースの機能

サーブレット・コンテナはユーザー・セッションのトラッキングおよび管理に、HTTP セッション・オブジェクトを使用します。HTTP セッション・オブジェクトは、`javax.servlet.http.HttpSession` インタフェースを実装するクラスのインスタンスです。`HttpSession` インタフェースは、次のパブリック・メソッドを指定して、セッション情報の取得と設定を行います。

- `void setAttribute(String name, Object value)`

  このメソッドは、指定したオブジェクトを指定した名前でセッションにバインドします。

- `Object getAttribute(String name)`

  このメソッドは、指定した名前（名前が一致しない場合は `null`）で、セッションにバインドされたオブジェクトを取得します。

サーブレット・コンテナとサーブレット自体の構成に基づいて、セッションを設定時間後自動的に期限切れにできます。あるいはサーブレットで明示的に無効化できます。サーブレットでは、次のメソッドを使用してセッションのライフ・サイクルを管理できます。メソッドは `HttpSession` インタフェースで指定します。

- `void invalidate()`

  このメソッドは、セッションを即時に無効化し、すべてのオブジェクトのバインディングをセッションから解除します。

- `void setMaxInactiveInterval(int interval)`

  このメソッドは、セッションのタイムアウト時間（秒単位）を int 型で設定します。負の値は、タイムアウトしないことを示します。値が 0 の場合は即時タイムアウトします。

- `boolean isNew()`

  このメソッドは、セッションを作成したリクエスト内にある場合は `true` を戻します。それ以外の場合は `false` を戻します。

- `long getCreationTime()`

  このメソッドは、セッション・オブジェクトの作成時間を、1970 年 1 月 1 日午前 0 時を基点としたミリ秒単位で戻します。

- `long getLastAccessedTime()`

  このメソッドは、クライアント・セッションに関連付けられた最新のリクエストの時刻を、1970 年 1 月 1 日午前 0 時を基点としたミリ秒単位で返します。クライアント・セッションがアクセスされていない場合、このメソッドはセッションが作成された時刻を返します。

サーブレットが HTTP セッション・オブジェクトを使用する例は、2-30 ページの「セッション・サーブレットの例」を参照してください。

`HttpSession` メソッドの詳細は、次の場所で Sun 社の Javadoc を参照してください。

http://java.sun.com/products/servlet/2.3/javadoc/index.html

セッションのキャンセル

HTTP セッション・オブジェクトは、サーバー・サイド・セッションが存続する間、継続します。セッションは、サーブレットにより明示的に終了されるか、あるいは一定期間後にタイムアウトになり、コンテナによりキャンセルされます。

タイムアウトによるキャンセル

OC4J サーバーのデフォルトのセッション・タイムアウトは、20 分です。`web.xml` の `<session-config>` 要素の `<session-timeout>` サブ要素を設定することにより、特定のアプリケーション用に、この期間を変更することができます。タイムアウトを分単位で、整数で指定します。たとえば、セッションのタイムアウトを 5 分に短縮するには、アプリケーションの `web.xml` ファイルに次の行を追加します。

```
<session-config>
  <session-timeout>5</session-timeout>
</session-config>
```

サーブレット仕様では、負の値によって、セッションがタイムアウトにならないデフォルトの動作が指定されます。次に例を示します。

```
<session-config>
  <session-timeout>-1</session-timeout>
</session-config>
```

値が 0 の場合は即時タイムアウトします。

サーブレットによるキャンセル

サーブレットでは、セッション・オブジェクト上で `invalidate()` メソッドを起動することで、明示的にセッションをキャンセルできます。`HttpServletRequest` オブジェクトの `getSession()` メソッドを起動して、新規セッション・オブジェクトを取得する必要があります。

分散アプリケーションでのセッションのレプリケーション

ステートフルなサーブレットのセッション・オブジェクトは、ロード・バランスされているクラスタ・アイランド内の他の OC4J サーバーにレプリケートできます。サーブレットへのリクエストを処理中のサーバーがエラーになると、そのリクエストは、同一クラスタ・アイランド内の別のサーバー上の他の JVM にフェイルオーバーし、セッション状態は維持されます。

次の各項で、詳細に説明します。

- セッションのレプリケーションの概要と要件
- 発生可能なクラスタリングのエラーおよび関連する環境フラグ
- セッションのレプリケーションの詳細およびロジスティックス

セッションのレプリケーションの概要と要件

OC4J サーバー間でアプリケーションのセッション状態のレプリケーションを有効にするには、Web アプリケーションを分散アプリケーションとしてマークする必要があります。これには、`web.xml` ファイルの標準 `<distributable>` 要素を使用します。次のように、`<web-app>` 要素にこのサブ要素が存在する場合、アプリケーションが分散アプリケーションであると指定されます。

```
<web-app ... >
   ...
   <distributable/>
   ...
</web-app>
```

注意： Oracle Application Server 環境では、Oracle Enterprise Manager 10*g* を使用してこれを行います。詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』のクラスタリングに関する説明を参照してください。

サーブレットによって `HttpSession` オブジェクトに格納されたオブジェクトは、レプリケートされます。レプリケーションを適切に機能させるには、シリアライズ可能（`java.io.Serializable` インタフェースを直接または間接的に実装）またはリモート化可能（`java.rmi.Remote` インタフェースを直接または間接的に実装）なオブジェクトを使用する必要があります。さらに、セッション・オブジェクト内のオブジェクトによって参照されるオブジェクト自体も、シリアライズ可能またはリモート化可能である必要があります。

発生可能なクラスタリングのエラーおよび関連する環境フラグ

レプリケートされたデータは、クラスタ・アイランド内の他の OC4J サーバーに非同期式に送信されます。パフォーマンス上の理由で、OC4J はレプリケーションの成功を確認するまで待機しません。そのため、可能性はきわめて低いですが、次の事態が発生することがあります。

- ブロードキャストの待機時間 : セッションのレプリケーション・メッセージは、クライアントが再ルーティングされるまで、他の OC4J サーバーによって受信および処理されません。
  1. クライアントがリクエストを送信し、OC4J サーバーからレスポンスを受信します。
  2. サーバーはクラスタ・アイランド内の他の OC4J サーバーに、クライアントの更新された状態のレプリケーション・メッセージをブロードキャストします。
  3. 更新状態がすべての OC4J サーバーによって受信および処理される前に、クライアントが別のリクエストを送信します。
  4. 元のサーバーに障害が発生し、新しい状態情報を受信していない OC4J サーバーの 1 つにクライアントが再ルーティングされ、その結果、クライアントは古いデータを受信します。
- クライアントへのレスポンス前の障害 : サーバーが他のサーバーにレプリケーション・メッセージをブロードキャストした後、クライアントへのレスポンスが完了する前に、サーバーに障害が発生します。

1. クライアントが OC4J サーバーにリクエストを送信します。
2. サーバーはクラスタ・アイランド内の他の OC4J サーバーに、クライアントの更新された状態のレプリケーション・メッセージをブロードキャストします。
3. ただし、サーバーがクライアントへのレスポンスを完了する前に、サーバーに障害が発生します。この結果、他の OC4J サーバーは処理が完了したことを認識していますが、クライアントはサーバーによって処理が完了していることを検出できません。

エラーの可能性があるため、OC4J および Oracle HTTP Server は、セッション・アフィニティを維持します。つまり、常にリクエストおよびレスポンスを、同一の OC4J JVM を通じてルーティングするようにします。セッション Cookie `JSESSIONID` は、HTTP リクエスト間で必須の詳細なルーティング情報を維持し、これによって、Oracle HTTP Server を介した後続のリクエストが、可能なかぎり元の JVM に送信されるようにします。

さらに、OC4J 10.1.2 実装では、これらのエラーの発生リスクを削減できる、2 つの環境フラグをサポートしています。

- `cluster.thread.priority`: デフォルトで、OC4J クラスタリング・スレッドは、他の主要な OC4J スレッドと同じ優先順位で実行されます。ただし、このフラグを 6 から 10 の任意の整数値に設定することにより、クラスタリング・スレッドの優先順位を上げることが可能です。10 が最高の優先順位です。
- `cluster.failover.delay`: OC4J サーバーに障害が発生すると、このフラグにより、クライアントが代替サーバーに再ルーティングされるまで、指定されたミリ秒の遅延が発生します。デフォルトでは、遅延は発生しません。前述のエラー状況の 1 つ目を回避するには、7000 ～ 9000 の値を設定します。

セッションのレプリケーションの詳細およびロジスティックス

分散アプリケーションの場合、セッション・オブジェクトに `setAttribute()` がコールされるたびに、セッション・レプリケーションがトリガーされます。コールで指定された名前および値はシリアライズおよびレプリケートされ、シリアライズされた値は指定された名前をキーとして格納されます。値は、フェイルオーバーされたサーブレットによってアクセスされたときにのみデシリアライズされます。

セッション・オブジェクトに属するデータ項目を更新する場合、毎回明示的に `setAttribute()` をコールする必要があります。たとえば、Bean を取得するためにセッション・オブジェクトに対して `getAttribute()` をコールし、次に Bean の状態を変更するために Bean にメソッドをコールする場合、セッション内の Bean を更新するために、セッション・オブジェクトに `setAttribute()` をコールする必要があります。これは、非分散環境の場合に、Bean にメソッドをコールすると Bean は参照によって即時渡され、直接セッション・オブジェクト内で更新されるのとは異なります。

また、この機能がパフォーマンスに与える影響にも注意してください。多数の `setAttribute()` コールのあるサーブレットは、状態のレプリケーションの実行時に発生する小さなオーバーヘッドにより、パフォーマンスが低下する可能性があります。

注意： OC4J のデバッグ用フラグ `http.session.debug` および `http.cluster.debug` を有効にすることにより、レプリケーションの実行時の状態およびセッション状態の更新を監視できます。2-16 ページの「OC4J のデバッグ・フラグ」および 2-17 ページの「OC4J のデバッグ・フラグの設定」を参照してください。

セッション・サーブレットの例

次の SessionServlet コードで実装されるサーブレットは、HttpSession オブジェクトを確立して、リクエスト・オブジェクトとセッション・オブジェクトによって維持されているデータを出力します。

SessionServlet コード

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Date;

public class SessionServlet extends HttpServlet {

  public void doGet (HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

    // Get the session object. Create a new one if it doesn't exist.
    HttpSession session = req.getSession(true);

    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    out.println("<head><title> " + "SessionServlet Output " +
                "</title></head><body>");
    out.println("<h1> SessionServlet Output </h1>");

    // Set up a session hit counter. "sessionservlet.counter" is just the
    // conventional way to create a key for the value to be stored in the
    // session object "dictionary".
    Integer ival =
      (Integer) session.getAttribute("sessionservlet.counter");
    if (ival == null) {
      ival = new Integer(1);
    }
    else {
      ival = new Integer(ival.intValue() + 1);
    }

    // Save the counter value.
    session.setAttribute("sessionservlet.counter", ival);

    // Report the counter value.
    out.println(" You have hit this page <b>" +
                ival + "</b> times.<p>");

    // This statement provides a target that the user can click
    // to activate URL rewriting. It is not done by default.
    out.println("Click <a href=" +
                res.encodeURL(HttpUtils.getRequestURL(req).toString()) +
                ">here</a>");
    out.println(" to ensure that session tracking is working even " +
                "if cookies aren't supported.<br>");
    out.println("Note that by default URL rewriting is not enabled" +
                " due to its large overhead.");

    // Report data from request.
    out.println("<h3>Request and Session Data</h3>");
    out.println("Session ID in Request: " +
                req.getRequestedSessionId());
    out.println("<br>Session ID in Request is from a Cookie: " +
                req.isRequestedSessionIdFromCookie());
```

```
    out.println("<br>Session ID in Request is from the URL: " +
                req.isRequestedSessionIdFromURL());
    out.println("<br>Valid Session ID: " +
                req.isRequestedSessionIdValid());

    // Report data from the session object.
    out.println("<h3>Session Data</h3>");
    out.println("New Session: " + session.isNew());
    out.println("<br> Session ID: " + session.getId());
    out.println("<br> Creation Time: " + new Date(session.getCreationTime()));
    out.println("<br>Last Accessed Time: " +
                new Date(session.getLastAccessedTime()));
    out.println("</body>");
    out.close();
  }

  public String getServletInfo() {
    return "A simple session servlet";
  }
}
```

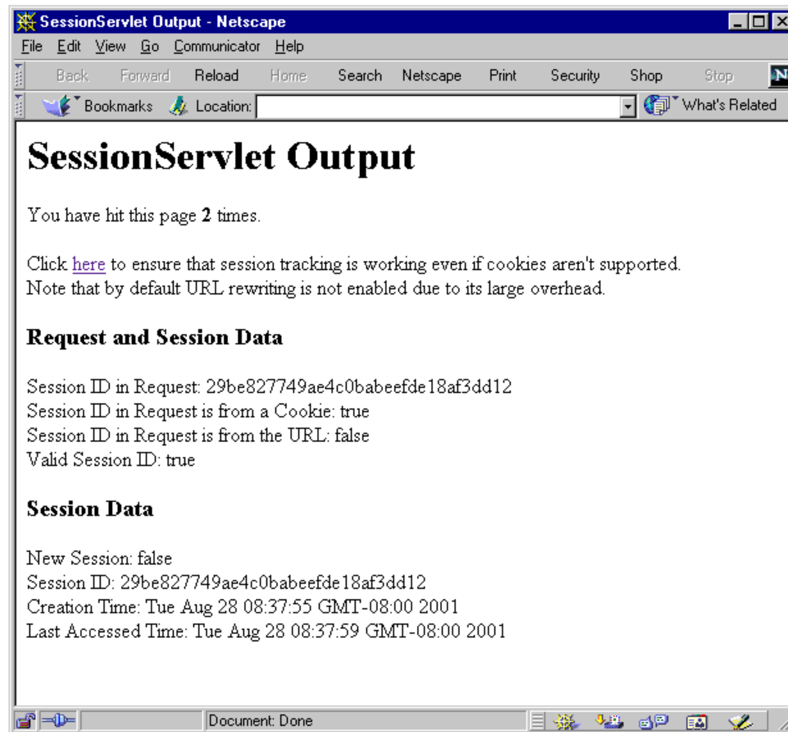
デプロイおよびテスト

OC4J スタンドアロンでは、前述のコードを OC4J のデフォルト Web アプリケーションの /WEB-INF/classes ディレクトリにあるファイル SessionServlet.java に保存します。デフォルトで、デフォルトの Web アプリケーションのルート・ディレクトリは j2ee/home/default-web-app です。（詳細は、5-24 ページの「OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーション」を参照してください。）

global-web-application.xml ファイルの <orion-web-app> 要素の development="true" 設定を使用すると便利です。development フラグの詳細は、6-2 ページの「global-web-application.xml および orion-web.xml の要素の説明」を参照してください。

図 2-1 に、Cookie が有効になっている Web ブラウザによって、サーブレットがセッション内で 2 回目に起動されたときの出力を示します。Cookie を無効にしてから、URL リライティングを行う HREF を選択するなど、Web ブラウザの設定を変更して、違いを試してみてください。

図 2-1 セッション・サーブレットの表示



サーブレットのセキュリティ

OC4J では、Oracle Application Server 環境下の Oracle HTTP Server と OC4J 間でセキュアな AJP を使用した Secure Socket Layer（SSL）通信をサポートしています。これは、Oracle HTTP Server が OC4J との通信に使用しているプロトコルである Apache JServ Protocol のセキュアなバージョンです。次の各項で、詳細を説明します。

- セキュリティ機能の使用
- Oracle HTTP Server および OC4J での SSL の構成
- SSL の一般的な問題と解決策

続いて、一般的なセキュリティ上の考慮事項を説明します。

- その他のセキュリティに関する考慮事項

注意：

- クライアントと Oracle HTTP Server 間のセキュアな通信は、Oracle HTTP Server と OC4J 間のセキュアな通信からは独立したものです。（また、Oracle HTTP Server と OC4J 間で使用されるセキュアな AJP プロトコルは、エンド・ユーザーには表示されません。）この項では、Oracle HTTP Server と OC4J 間のセキュアな通信についてのみ説明します。
- さらに、OC4J スタンドアロンでは、クライアントと OC4J 間での HTTPS を使用した直接の SSL 通信をサポートしています。スタンドアロン・バージョンをダウンロードする際に OTN-J から入手可能な『Oracle Application Server Containers for J2EE スタンドアロン・ユーザーズ・ガイド』を参照してください。

Oracle Application Server のセキュリティおよび Oracle HTTP Server に関する追加情報は、次のマニュアルを参照してください。

- 『Oracle Application Server セキュリティ・ガイド』（クライアントと Oracle HTTP Server 間のセキュアなプロトコルに関する情報を含む）
- 『Oracle Application Server Containers for J2EE セキュリティ・ガイド』（SSL 鍵、証明書および関連概念の概要を含む）
- 『Oracle HTTP Server 管理者ガイド』

セキュリティ機能の使用

次の各項で、SSL 機能を OC4J および Oracle HTTP Server で使用する方法について説明します。

- OC4J および Oracle HTTP Server での証明書の使用
- クライアント認証のリクエスト

OC4J および Oracle HTTP Server での証明書の使用

次に、OC4J で SSL 通信を行う際に鍵および証明書を使用するための手順を説明します。これらはサーバーレベルの手順で、通常はセキュアな通信を必要とするアプリケーションのデプロイ前に実行します。たとえば、Oracle Application Server インスタンスを最初に設定する際に実行します。

キーストアは、すべての信頼されたパーティの証明書を含め、プログラムで使用する証明書を格納するために使用される点に注意してください。キーストアを使用することにより、OC4J などのエンティティは、他のパーティを認証すると同時に、自らを他のパーティに対して認証することができます。Oracle HTTP Server では、同じ目的で Wallet と呼ばれるものを使用します。

Java では、キーストアは `java.security.KeyStore` インスタンスで、Sun 社の JDK とともに提供される `keytool` ユーティリティを使用して作成および操作できます。このオブジェクトの基礎となる物体はファイルです。`keytool` の詳細は、次の Web サイトを参照してください。

http://java.sun.com/j2se/1.3/docs/tooldocs/win32/keytool.html

Oracle Wallet に対する Oracle Wallet Manager の機能は、キーストアに対する `keytool` の機能と同じです。

次に、OC4J および Oracle HTTP Server 間で証明書を使用するための手順を示します。

1. `keytool` を使用して、秘密鍵、公開鍵および署名されていない証明書を生成します。この情報を、新規のキーストアまたは既存のキーストアに入れます。
2. 次のいずれかの方法で、証明書の署名を取得します。

   自分の署名を生成します。

   a. `keytool` を使用して、証明書を自己署名します。クライアントによって、自らが認証局のように信頼される場合、この方法を使用できます。

   または、認識されている認証局から署名を取得します。

   a. 手順 1 の証明書を使用し、`keytool` を使用して証明書リクエストを生成します。これは、証明書を認証局によって署名してもらうためのリクエストです。
   b. 証明書リクエストを認証局に送信します。
   c. 署名を認証局から受領し、再度 `keytool` を使用してキーストアにインポートします。キーストア内で、署名は関連付けられた証明書に対してマッチされます。

注意： Oracle Application Server には、Oracle Application Server Certificate Authority（OCA）が含まれています。OCA を使用すると、証明書をユーザー自身およびそのユーザーに対して作成および発行できますが、事前の手配なしではユーザーの組織外ではこれらの証明書は認識されません。OCA の詳細は、『Oracle Application Server Certificate Authority 管理者ガイド』を参照してください。

署名のリクエストおよび受領のプロセスは、認証局によって異なります。これは Oracle Application Server の対象および制御の範囲外であるため、Oracle Application Server のドキュメントでは説明されていません。詳細は、認証局の Web サイトを参照してください。（すべてのブラウザに、信頼できる認証局のリストが含まれています。）次に、VeriSign 社および Thawte 社の Web アドレスを示します。

http://www.verisign.com/

http://www.thawte.com/

OC4J と Oracle HTTP Server 間の SSL 通信には、必ず前述の手順を Oracle HTTP Server に対して実行しますが、キーストアおよび keytool ユーティリティのかわりに、Wallet および Oracle Wallet Manager を使用します。Wallet および Oracle Wallet Manager の詳細は、『Oracle Application Server セキュリティ・ガイド』を参照してください。

前述の手順 1 および 2 に加え、必要に応じて次の手順を実行します。

1. **Oracle HTTP Server がまだ信頼していないエンティティによって OC4J の証明書が署名されている場合**、そのエンティティの証明書を取得して Oracle HTTP Server にインポートします。OC4J の証明書が自己署名であるかどうかによって、詳細は次のように異なります。

   OC4J の証明書が自己署名である場合（基本的に、Oracle HTTP Server はまだ OC4J を信頼していません）、次の手順を実行します。

   a. keytool を使用して、OC4J から OC4J の証明書をエクスポートします。この手順により、証明書が Oracle HTTP Server によってアクセス可能なファイルに置かれます。

   b. Oracle Wallet Manager を使用して、Oracle HTTP Server に OC4J の証明書をインポートします。

   OC4J の証明書が別のエンティティによって署名されている場合（Oracle HTTP Server はまだ信頼していません）、次の手順を実行します。

   a. エンティティからエクスポートするなど、適切な方法でそのエンティティの証明書を取得します。実際の手順は、エンティティによって大きく異なります。

   b. Oracle Wallet Manager を使用して、Oracle HTTP Server にエンティティの証明書をインポートします。

2. **Oracle HTTP Server の証明書が、OC4J がまだ信頼していないエンティティによって署名され、OC4J がクライアント認証を必要とするモードで実行されている場合、次の手順を実行します**（2-35 ページの「クライアント認証のリクエスト」を参照）。

   a. エンティティからエクスポートするなど、適切な方法でそのエンティティの証明書を取得します。実際の手順は、エンティティによって大きく異なります。

   b. keytool を使用して、OC4J にエンティティの証明書をインポートします。

注意： Oracle HTTP Server と OC4J 間の SSL 上の通信で、2 者間の通信チャネル上の全データが暗号化されます。次の手順が実行されます。1）暗号化チャネルの設立時に、Oracle HTTP Server によって OC4J の証明連鎖が認証されます。2）オプションで、OC4J がクライアント認証モードの場合、Oracle HTTP Server が OC4J によって認証されます。このプロセスも、暗号化チャネルの設立時に行われます。3）この最初の交換以降に行われる通信は、すべて暗号化されます。

例 : SSL 証明書の作成と自らの署名の生成　この例は、前述の手順 2 における、keytool を使用して証明書を自己署名することにより、自らの署名を生成するモードに対応します。

まず、keytool コマンドを使用し、RSA 秘密鍵と公開鍵のペアを持つキーストアを作成します。次の例では（% がシステム・プロンプト）、RSA 鍵ペア生成アルゴリズムを使用して、mykeystore というファイルに常駐し、パスワードは 123456 で 21 日間有効のキーストアが生成されます。

```
% keytool -genkey -keyalg "RSA" -keystore mykeystore -storepass 123456 -validity 21
```

次の点に注意してください。

- keystore オプションにより、鍵が格納されるファイルの名前を指定します。
- storepass オプションにより、キーストアを保護するパスワードを設定します。
- validity オプションにより、証明書の有効日数を設定します。

keytool により、次のように、さらに情報を入力するようプロンプトが表示されます。

```
What is your first and last name?
  [Unknown]:  Test User
What is the name of your organizational unit?
  [Unknown]:  Support
What is the name of your organization?
  [Unknown]:  Oracle
What is the name of your City or Locality?
  [Unknown]:  Redwood Shores
What is the name of your State or Province?
  [Unknown]:  CA
What is the two-letter country code for this unit?
  [Unknown]:  US
Is <CN=Test User, OU=Support, O=Oracle, L=Reading, ST=Berkshire, C=GB> correct?
  [no]:  yes

Enter key password for <mykey>
        (RETURN if same as keystore password):
```

注意 :　2 文字の国コードを調べるには、次の URL の ISO 国コード・リストを使用してください。

http://www.bcpl.net/~j1m5path/isocodes.html

---

mykeystore ファイルが現行ディレクトリに作成されます。鍵のデフォルトの別名は mykey です。

クライアント認証のリクエスト

OC4J では、サーバーがクライアントと通信を開始する前に、クライアントからの認証を明示的にリクエストする、クライアント認証モードをサポートしています。Oracle Application Server 環境では、Oracle HTTP Server が OC4J のクライアントとして動作します。

クライアント認証を行うために、Oracle HTTP Server は証明書を必要とし、証明書と、ルート証明書で終わる証明連鎖を送信することにより、自らを認証します。OC4J は、クライアントの信頼の連鎖を設立する際、指定されたリストに含まれるルート証明書のみ受領するよう構成できます。

OC4J が信頼する証明書は、トラスト・ポイントと呼ばれます。Oracle HTTP Server からの証明連鎖では、トラスト・ポイントは、OC4J 内のキーストアに一致する最初の証明書です。信頼を設立する方法は 3 つあります。

- クライアント証明書がキーストア内に存在する。
- Oracle HTTP Server からの証明連鎖内の中間 CA 証明書がキーストア内に存在する。
- Oracle HTTP Server からの証明連鎖内のルート CA 証明書がキーストア内に存在する。

OC4J は、偽造された証明書を防ぐため、トラスト・ポイントを含め、そこまでの証明連鎖全体が有効であることを検証します。

`needs-client-auth` 属性を使用してクライアント認証をリクエストする場合、次の手順を実行します。この属性の構成方法は、2-36 ページの「OC4J での SSL の構成手順」を参照してください。

1. Oracle HTTP Server からの連鎖内のどの証明書をトラスト・ポイントにするかを決定します。このトラスト・ポイントを使用する証明書の発行を制御できるか、または認証局を発行者として信頼できることを確認します。
2. クライアント証明書の認証のトラスト・ポイントとして、サーバーのキーストアに中間またはルート証明書をインポートします。

> **注意：** OC4J によって使用しないトラスト・ポイントが存在する場合、これらのトラスト・ポイントがキーストアに含まれていないことを確認してください。

3. クライアント証明書の作成手順を実行します（2-33 ページの「OC4J および Oracle HTTP Server での証明書の使用」を参照）。クライアント証明書には、サーバーにインストールされている中間またはルート証明書が含まれます。別の認証局をも信頼する場合、その認証局から証明書を取得します。
4. 証明書を Oracle HTTP Server 内のファイルに保存します。
5. Oracle HTTP Server がセキュアな AJP 接続を開始する際、証明書を提供します。

Oracle HTTP Server および OC4J での SSL の構成

Oracle HTTP Server および OC4J 間でセキュアな通信を行うには、次のように、両方で構成手順を実行する必要があります。

- Oracle HTTP Server での SSL の構成手順
- OC4J での SSL の構成手順

Oracle HTTP Server での SSL の構成手順

Oracle HTTP Server では、セキュアな通信のために、`mod_oc4j.conf` で適切な SSL の設定を検証します。次のように、SSL が有効で、Wallet ファイルおよびパスワード指定されている必要があります。

```
Oc4jEnableSSL on
Oc4jSSLWalletFile wallet_path
Oc4jSSLWalletPassword pwd
```

`wallet_path` 値は、Wallet ファイルへのディレクトリ・パスで、ファイル名を含みません。（Wallet ファイル名はすでに認識されています。）*`pwd`* 値は Wallet のパスワードです。

`mod_oc4j.conf` ファイルの詳細は、『Oracle HTTP Server 管理者ガイド』を参照してください。

OC4J での SSL の構成手順

`default-web-site.xml` ファイル（または、必要に応じて他の Web サイトの XML ファイル）内で、`<web-site>` 要素の下で適切な SSL 設定を指定する必要があります。

1. 次のように、`secure` フラグをオンにしてセキュアな通信を指定します。

```
<web-site ... secure="true" ... >
   ...
</web-site>
```

secure="true" に設定すると、AJP プロトコルで SSL ソケットを使用するよう指定されます。

2. 次のように、<ssl-config> サブ要素とその keystore および keystore-password 属性を使用して、キーストアのパスおよびパスワードを指定します。

```
<web-site ... secure="true" ... >
   ...
   <ssl-config keystore="path_and_file" keystore-password="pwd" />
</web-site>
```

secure フラグが true に設定されている場合、<ssl-config> 要素は必須です。

path_and_file 値では、絶対または相対ディレクトリ・パスのどちらでも指定でき、ファイル名を含めます。相対パスは、Web サイトの XML ファイルの位置に対して相対的です。

3. オプションで、クライアント認証を必要とすることを指定するために、needs-client-auth フラグをオンにします。これは、<ssl-config> 要素の属性です。

```
<web-site ... secure="true" ... >
   ...
   <ssl-config keystore="path_and_file" keystore-password="pwd"
               needs-client-auth="true" />
</web-site>
```

この手順により、OC4J が、Oracle HTTP Server などのクライアント・エンティティの識別により、セキュアな通信を許容または拒絶するモードが設定されます。needs-client-auth フラグにより、OC4J が接続時にクライアントの証明連鎖をリクエストするよう指定されます。OC4J がクライアントのルート証明書を認識すると、クライアントが許容されます。

<ssl-config> 要素で指定されたキーストアには、セキュアな AJP および SSL による OC4J への接続を認可されているクライアントの証明書が含まれている必要があります。

次に、クライアント認証でセキュアな通信を設定する例を示します。

```
<web-site display-name="OC4J Web Site" protocol="ajp13" secure="true" >
   <default-web-app application="default" name="defaultWebApp" root="/j2ee" />
   <access-log path="../log/default-web-access.log" />
   <ssl-config keystore="../keystore" keystore-password="welcome"
               needs-client-auth="true" />
</web-site>
```

太字の部分のみが、セキュリティ固有の箇所です。セキュアな通信の使用または不使用にかかわらず、Oracle HTTP Server を介した通信では、プロトコル値は必ず ajp13 です。プロトコル値が ajp13 で secure="false" の場合は AJP プロトコル、ajp13 で secure="true" の場合はセキュアな AJP プロトコルを示します。

<web-site> 要素および <ssl-config> 要素の要素および属性の詳細は、6-19 ページの「Web サイト XML ファイルの要素の説明」を参照してください。

また、関連情報は、2-35 ページの「クライアント認証のリクエスト」を参照してください。

SSL の一般的な問題と解決策

この項では、一般的な SSL のエラー、その原因および解決策と、一般的な SSL のデバッグについて説明します。

一般的な SSL のエラー

SSL 証明書の使用時に、次のエラーが発生する場合があります。

Keytool Error: java.security.cert.CertificateException: Unsupported encoding

原因： 文末に空白があります。keytool ユーティリティでは、文末に空白は使用できません。

処置： 文末の空白をすべて削除してください。それでもエラーが発生する場合、証明書の返信ファイルに改行を追加します。

Keytool Error: KeyPairGenerator not available

原因： 以前のバージョンの JDK の keytool ユーティリティを使用していると考えられます。

処置： システム上の最新の JDK の keytool ユーティリティを使用してください。確実に最新の JDK を使用するには、JDK のフルパスを指定してください。

Keytool Error: Failed to establish chain from reply

原因： keytool ユーティリティがキーストア内でルート CA 証明書を検出できないため、サーバーの鍵から信頼されたルート認証局への証明連鎖を構築できません。

処置： 次のコマンドを実行します。

```
keytool -keystore keystore -import -alias cacert -file cacert.cer
(keytool -keystore keystore -import -alias intercert -file inter.cer)
```

中間 CA keytool ユーティリティを使用している場合、次のコマンドを実行します。

```
keystore keystore -genkey -keyalg RSA -alias serverkey
keytool -keystore keystore -certreq -file my.host.com.csr
```

Certificate Signing Request（CSR）から証明書を取得し、次のコマンドを実行します。

```
keytool -keystore keystore -import -file my.host.com.cer -alias serverkey
```

No available certificate corresponds to the SSL cipher suites that are enabled

原因： 証明書に問題があります。

処置： 問題を決定し、修正してください。

一般的な SSL のデバッグ

OC4J スタンドアロンで開発している際、Java Secure Socket Extension（JSSE）実装からの冗長デバッグ情報を表示できます。オプションのリストを取得するには、次のように OC4J を起動します。

```
java -Djavax.net.debug=help -jar oc4j.jar
```

次のように起動すると、冗長性が完全に有効になります。

```
java -Djavax.net.debug=all -jar oc4j.jar
```

これにより、ブラウザのリクエスト・ヘッダー、サーバーの HTTP ヘッダー、サーバーの HTTP ボディ、（暗号化前後の）コンテンツ長、および SSL のバージョンが表示されます。

その他のセキュリティに関する考慮事項

前述の SSL 機能に加え、OC4J サーブレット・コンテナで稼働する Web アプリケーションのセキュリティに関して、次の点を考慮する必要があります。

- global-web-application.xml ファイルまたは orion-web.xml ファイルにおいて、<orion-web-app> の <jazn-web-app> サブ要素を使用して OracleAS JAAS Provider および Single Sign-On（SSO）プロパティをサーブレットの実行用に構成します。特定のセキュリティ・サブジェクトの権限を使用してサーブレットを起動するには、これらの機能を適切に設定する必要があります。この要素は、6-2 ページの「global-web-application.xml および orion-web.xml の要素の説明」で説明されています。
- OC4J には、web.xml デプロイメント・ディスクリプタの <security-role> 要素を使用したセキュリティ制約とセキュリティ・ロールの標準サポートが用意されています。一般情報については、サーブレット仕様を参照してください。OC4J には、global-web-application.xml ファイルの <security-role-mapping> 要素を使用した関連サポートも備わっています。global-web-application.xml の要素および属性の詳細は、6-2 ページの「global-web-application.xml および orion-web.xml の構成」を参照してください。

- クラス名による起動は、開発環境でのみ使用してください。クラス名によるサーブレットの起動をユーザーに許可すると、重大なセキュリティ上のリスクが発生するためです。

  クラス名による起動では、web.xml ファイルで特に対処しないかぎり、標準のセキュリティ制約が無視されるおそれがあります。また、サーブレットをクラスで起動すると、スローされる例外によって、サーブレットの場所の物理パスが公開される可能性があり、非常に危険です。

  特に本番環境でのセキュリティの問題を解決するには、次のいずれかの方法で、クラス名によるサーブレット起動を無効にします。

  – システム・プロパティ http.webdir.enable の値を false に設定。この結果、servlet-webdir 設定は無視されます。

  – global-web-application.xml または orion-web.xml を使用して、servlet-webdir の値を ""（空の引用符）に設定。

  （servlet-webdir の設定に関する追加情報を含む、クラス名による起動の詳細は、2-22 ページの「OC4J 開発時におけるクラス名によるサーブレットの起動」を参照してください。）

  たとえば、orion-web.xml で次のような構成を行うと、クラス名による起動が無効になります。

```
<orion-web-app ... servlet-webdir="" ... >
   ...
</orion-web-app>
```

- 破壊的な目的によるセッション ID 番号の推測やハッキングを防ぐために、OC4J では java.security.SecureRandom 機能を使用して、ランダムなセッション ID 番号を生成します。

サーブレットのベスト・プラクティスのサマリー

この項では、OC4J サーブレットのコーディングと構成に役立つベスト・プラクティスをまとめます。

ベスト・プラクティス : 一般的なコーディング

一般的なコーディングには次の方法をお薦めします。

- 柔軟性を考慮し、リソース接続の URL はハードコーディングしないようにします。データ・ソース構成の詳細は、4-2 ページの「サーブレットでの JDBC の使用」を参照してください。
- 効率性を考慮し、可能な場合はサーブレット・フィルタを使用します。3-2 ページの「サーブレット・フィルタ」を参照してください。
- リソースやクラスをロードする場合は、必ずコンテキスト・クラスローダーを使用します。次のコールは、現在実行しているスレッドのコンテキスト・クラスローダーを返します。

```
Thread.currentThread().getContextClassLoader();
```

ベスト・プラクティス : セッション

HTTP セッションには次の方法をお薦めします。関連情報については、2-24 ページの「サーブレットのセッション」を参照してください。

- コードがセッションによって無効になったら、セッションも無効になるようにします。
- セッション状態の永続化が適切および実行可能な場合は永続化し、永続化が実行不可能な場合はセッションをレプリケートします。
- 共有リソースはセッションに格納しません。
- 適切なセッション・タイムアウト値を設定します。

- クラスタリングとセキュリティは、初期の開発プロセスで考慮するようにします。

ベスト・プラクティス : 構成

本番環境には次の構成方法をお薦めします。

- パフォーマンスを改善するために、開発時の `check-for-updates` フラグは無効にしておきます。このフラグの詳細は、2-3 ページの「開発用の主な OC4J のフラグ」を参照してください。
- クラス名によるサーブレットの起動はセキュリティ上のリスクが大きいため、開発時に使用するには便利ですが、本番環境では無効にします。詳細は、2-22 ページの「OC4J 開発時におけるクラス名によるサーブレットの起動」を参照してください。
- ライブラリをサーバー・レベルで共有するか、アプリケーション固有のライブラリとして使用するかを十分に検討する必要があります。必要がない場合は、共有しないでください。サーバー・レベルのライブラリとして使用する場合は、OC4J `server.xml` ファイルまたはグローバル `application.xml` ファイルの `<library>` 要素を使用します。アプリケーション・レベルのライブラリとして使用する場合は、アプリケーションの `orion-application.xml` ファイルの `<library>` 要素を使用します。これらのファイルの詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。
- ライブラリにシステムのクラスパスは使用しないようにします。

3 サーブレット・フィルタとイベント・リスナー

この章では、サーブレットに関する次の機能について説明します。

- サーブレット・フィルタ
- イベント・リスナー

サーブレット・フィルタ

サーブレット・フィルタは、次の各項で説明するように、Web アプリケーションのリクエストの前処理とレスポンスの後処理に使用します。

- サーブレット・フィルタの概要
- サーブレット・コンテナによるフィルタの起動
- 転送またはインクルード・ターゲットのフィルタ
- フィルタの例

サーブレット・フィルタの概要

サーブレット・コンテナが、クライアントにかわってサーブレット内のメソッドをコールすると、クライアントが送信した HTTP リクエストは、デフォルトで、サーブレットに直接渡されます。サーブレットが生成するレスポンスは、コンテナによる内容の修正なしに、デフォルトでクライアントに直接返されます。この場合、サーブレットはリクエストを処理し、アプリケーションに必要な分のレスポンスを生成します。

ただし、サーブレットに対するリクエストの前処理が役立つ場合が多くあります。さらに、サーブレットのクラスからのレスポンスに対する変更が役立つ場合があります。暗号化がその一例です。アプリケーション内のサーブレットまたはサーブレット・グループは、機密に関わるレスポンス・データを生成することがあります。このようなデータは、特に HTTP などのセキュアでないプロトコルを使用して接続が行われている場合は、クリアテキスト・フォームでネットワーク上に発信すべきではありません。フィルタによって、レスポンスを暗号化できます。この場合は当然、クライアント側でレスポンスを復号化できる必要があります。

通常フィルタが使用されるのは、1 つのサーブレットではなく、サーブレット・グループのリクエストまたはレスポンスに対して前処理または後処理を適用する場合です。リクエストまたはレスポンスの変更が必要なサーブレットが 1 つのみの場合、フィルタの作成は不要です。必要な変更をサーブレット自体で直接実行します。

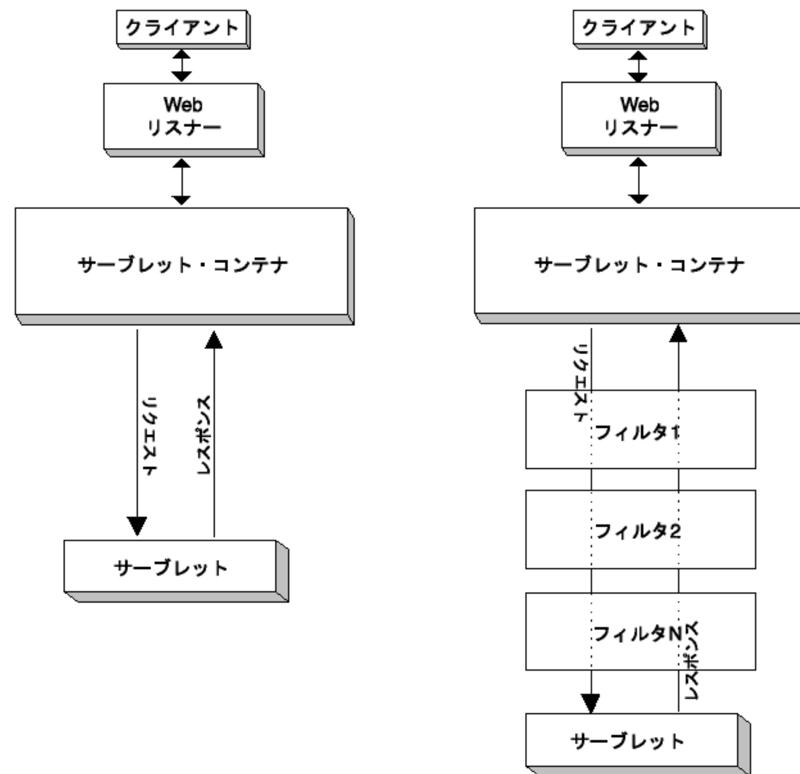
フィルタはサーブレットではない点に注意してください。フィルタは、`doGet()` や `doPost()` などの `HttpServlet` メソッドの実装およびオーバーライドは行いません。フィルタが実装するのは、`javax.servlet.Filter` インタフェースのメソッドです。メソッドは次のとおりです。

- `init()`
- `destroy()`
- `doFilter()`

サーブレット・コンテナによるフィルタの起動

図 3-1 に、サーブレット・コンテナによるフィルタの起動方法を示します。左側の例では、コール対象のサーブレットに対してフィルタが構成されていません。右側の例では、複数のフィルタ（1、2、...、N）が連鎖して構成され、サーブレットがコールされる前とレスポンスした後にコンテナによって起動されます。`web.xml` ファイルに、コンテナを使用してフィルタを起動するサーブレットを指定します。

図 3-1 フィルタ使用およびフィルタなしのサーブレットの起動



フィルタの起動順序は、web.xml ファイル内でのフィルタの構成順序に従います。web.xml 内の最初のフィルタがリクエスト時に最初に起動され、web.xml 内の最後のフィルタがレスポンス時に最初に起動されます。レスポンス時には順序が逆になります。

注意： 複数のフィルタを使用する際、機能の重複や、フィルタが上書きする対象など、組合せに注意してください。

転送またはインクルード・ターゲットのフィルタ

OC4J 10.1.2 実装では、サーブレットがフィルタされると、そのフィルタされたサーブレットに転送またはインクルードされているサーブレットは、デフォルトでフィルタされません。ただし、次の環境設定により、この動作を変更できます。

```
oracle.j2ee.filter.on.dispatch=true
```

このフラグは、デフォルトで false に設定されます。

注意： このフラグは、現行のリリースでは一時的なメカニズムです。今後のリリースではサーブレット仕様のバージョン 2.4 に準拠する予定です。この仕様では、フィルタされたサーブレットに転送またはインクルードされるサーブレットは、デフォルトでフィルタされません。ただし、仕様に従い、この動作は web.xml ファイルで構成できます。

インクルードおよび転送に関する一般的な情報は、2-10 ページの「サーブレットのインクルードおよび転送」を参照してください。

フィルタの例

この項では、3 つのサーブレット・フィルタの例を説明します。

フィルタの例 1

この項では、単純なフィルタの例を説明します。すべてのフィルタは、javax.servlet.Filter インタフェースの 3 つのメソッドを実装するか、または 3 つのメソッドを実装するクラスを拡張する必要があります。したがって、最初の手順では、これらのメソッドを実装するクラスを作成します。このクラスは、MyGenericFilter と呼ばれ、他のフィルタによって拡張できます。

汎用フィルタ 次に汎用フィルタ・コードを示します。この汎用フィルタは、com.example.filter パッケージに含まれており、対応するディレクトリ構造を設定することを前提としています。

これは空（またはパススルー）フィルタの要素の例で、テンプレートとして使用できます。

```
package com.example.filter;
import javax.servlet.*;

public class MyGenericFilter implements javax.servlet.Filter {
  public FilterConfig filterConfig;                                  //1

  public void doFilter(final ServletRequest request,                 //2
                       final ServletResponse response,
                       FilterChain chain)
      throws java.io.IOException, javax.servlet.ServletException {
    chain.doFilter(request,response);                                //3
  }

  public void init(final FilterConfig filterConfig) {               //4
    this.filterConfig = filterConfig;
  }

  public void destroy() {                                            //5
  }
}
```

このコードをパッケージ・ディレクトリ内の MyGenericFilter.java というファイルに保存します。コード内の番号に対する注は、次のとおりです。

1. このコードは、フィルタの構成オブジェクトを保存するための変数を宣言します。
2. doFilter() メソッドには、フィルタを実装するコードが含まれます。
3. 汎用の場合は、フィルタ・チェーンのみをコールします。
4. init() メソッドは、フィルタの構成を変数で保存します。
5. destroy() メソッドをオーバーライドすると、必要なファイナライズを行うことができます。

フィルタ・コード : HelloWorldFilter.java このフィルタは、前述の MyGenericFilter クラスの doFilter() メソッドをオーバーライドします。このフィルタは最初にコールされると、コンソールにメッセージを出力し、次に新規属性をサーブレットのリクエストに追加してから、フィルタ・チェーンをコールします。この例のチェーンには他のフィルタがありません。したがって、コンテナはリクエストをサーブレットに直接渡します。次のコードを HelloWorldFilter.java というファイルに入力します。

```
package com.acme.filter;

import javax.servlet.*;

public class HelloWorldFilter extends MyGenericFilter {
```

```
  private FilterConfig filterConfig;

  public void doFilter(final ServletRequest request,
                       final ServletResponse response,
                       FilterChain chain)
       throws java.io.IOException, javax.servlet.ServletException  {
    System.out.println("Entering Filter");
    request.setAttribute("hello","Hello World!");
    chain.doFilter(request,response);
    System.out.println("Exiting HelloWorldFilter");
  }
}
```

JSP コード : filter.jsp 例を単純にするために、フィルタ出力を処理するサーブレットを JSP ページとして作成します。次のように作成します。

```
<HTML>
<HEAD>
<TITLE>Filter Example 1</TITLE>
</HEAD>
<BODY>
<HR>
<P><%=request.getAttribute("hello")%></P>
<P>Check your console output!</P>
<HR>
</BODY>
</HTML>
```

この JSP ページは、フィルタによって追加された新規リクエスト属性 `hello` を取得し、その値をコンソールに出力します。`filter.jsp` ページを OC4J スタンドアロンのデフォルトの Web アプリケーションのルート・ディレクトリに置き、ブラウザから `filter.jsp` を起動したときにコンソール・ウィンドウに表示されるようにします。

例 1 の設定 この章では、OC4J スタンドアロンのデフォルトの Web アプリケーションを使用して、フィルタの例をテストします。フィルタは、デフォルトの Web アプリケーションの `/WEB-INF` ディレクトリ（デフォルトは `j2ee/home/default-web-app/WEB-INF`）にある `web.xml` ファイルに構成します。

`<web-app>` 要素には次のエントリが必要です。

```
  <!-- Filter Example 1 -->
  <filter>
    <filter-name>helloWorld</filter-name>
    <filter-class>com.acme.filter.HelloWorldFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>helloWorld</filter-name>
    <url-pattern>/filter.jsp</url-pattern>
  </filter-mapping>
  <!-- end Filter Example 1 -->
```

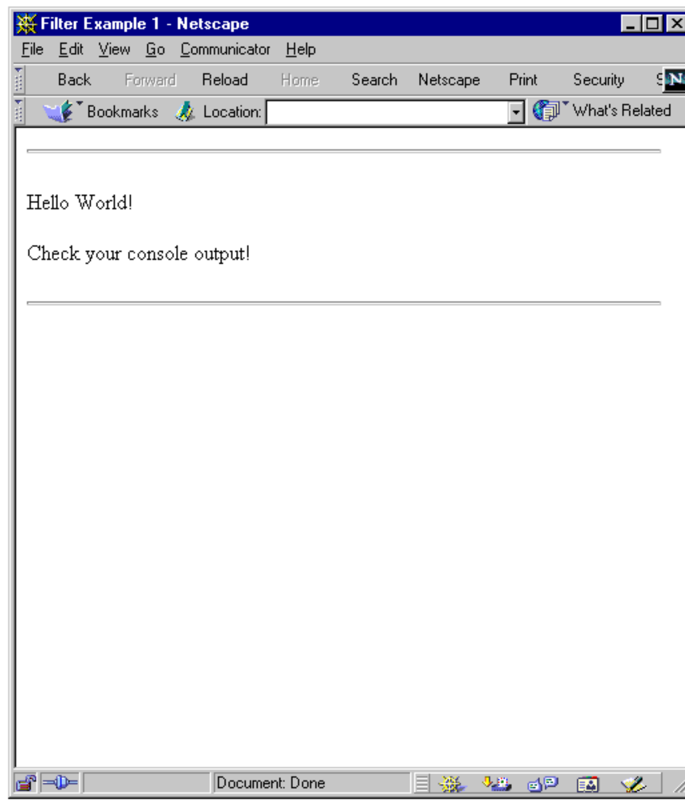
`<filter>` 要素によって、フィルタ名とそのフィルタを実装する Java クラスが定義されます。`<filter-mapping>` 要素によって、`<filter-name>` が適用されるターゲットを指定する URL パターンが定義されます。この単純な例では、フィルタが適用されるターゲットは、`filter.jsp` 内の JSP コードのみです。

例 1 の実行 Web ブラウザから `filter.jsp` を起動します。コンソールへの出力は、次のようになります。

```
hostname% Entering Filter
Exiting HelloWorldFilter
```

Web ブラウザには、図 3-2 に示すような出力が表示されます。

図 3-2 例 1 の出力



フィルタの例 2

web.xml ファイル内の初期化パラメータを使用してフィルタを構成できます。この項では、次の web.xml エントリを使用したフィルタの例を説明します。このエントリによってフィルタはパラメータ化されます。

```
<!-- Filter Example 2 -->
<filter>
  <filter-name>message</filter-name>
  <filter-class>com.acme.filter.MessageFilter</filter-class>
  <init-param>
    <param-name>message</param-name>
    <param-value>A message for you!</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>message</filter-name>
  <url-pattern>/filter2.jsp</url-pattern>
</filter-mapping>
<!-- end Filter Example 2 -->
```

この時点で、message という名前のフィルタが初期化パラメータによって構成されました。このパラメータも message と呼ばれます。message パラメータの値は、A message for you! です。

フィルタ・コード : MessageFilter.java 次に、message フィルタの例を実装するコードを示します。3-4 ページの「フィルタの例 1」の MyGenericFilter クラスを使用します。

```
package com.acme.filter;
import javax.servlet.*;
```

```
public class MessageFilter extends MyGenericFilter {
  public void doFilter(final ServletRequest request,
                       final ServletResponse response,
                       FilterChain chain)
       throws java.io.IOException, javax.servlet.ServletException {
    System.out.println("Entering MessageFilter");
    String message = filterConfig.getInitParameter("message");
    request.setAttribute("message",message);
    chain.doFilter(request,response);
    System.out.println("Exiting MessageFilter");
  }
}
```

このフィルタは、汎用フィルタに保存された filterConfig オブジェクトを使用します。filterConfig.getInitParameter() メソッドは、初期化パラメータの値を返します。

JSP コード : filter2.jsp 最初の例と同様に、この例では JSP ページを使用して、フィルタをテストするサーブレットを実装します。前述の <url-pattern> タグで名前を付けたフィルタは、filter2.jsp です。次に示すコードは、OC4J スタンドアロンのデフォルトの Web アプリケーションのルート・ディレクトリにある filter2.jsp ファイルに入力できます。

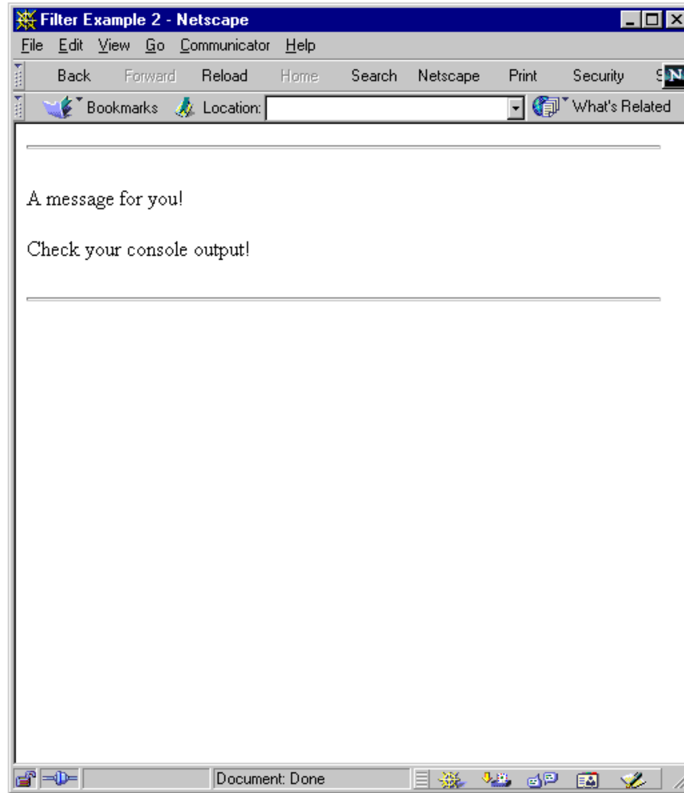
```
<HTML>
<HEAD>
<TITLE>Lesson 2</TITLE>
</HEAD>
<BODY>
<HR>
<P><%=request.getAttribute("message")%></P>
<P>Check your console output!</P>
<HR>
</BODY>
</HTML>
```

例 2 の実行 フィルタ構成が前述のとおり web.xml ファイルに入力されていることを確認してください。次に、ブラウザで JSP ページにアクセスします。コンソールへの出力は、次のようになります。

```
Auto-deploying file:/private/tssmith/appserver/default-web-app/ (Assembly had been
updated)...
Entering MessageFilter
Exiting MessageFilter
```

web.xml ファイルの編集後にデフォルトの Web アプリケーションが再度デプロイされたことを示すサーバーからのメッセージに注意してください。また、入力時と終了時のフィルタからのメッセージにも注意してください。Web ブラウザには、図 3-3 に示すような出力が表示されます。

図 3-3 例 2 の出力



フィルタの例 3

特に役立つフィルタの機能は、リクエストに対するレスポンスを操作できることです。操作するには、標準的な javax.servlet.http.HttpServletResponseWrapper クラス、ユーザー定義の javax.servlet.ServletOutputStream オブジェクトおよびフィルタを使用します。フィルタのテストには、フィルタで処理するターゲットも必要です。この例では、フィルタ処理の対象になるターゲットは、JSP ページです。

この例を実装するために、新規クラスを 3 つ作成します。

- FilterServletOutputStream: このクラスは、レスポンス・ラッパーに対する ServletOutputStream の新規実装です。
- GenericResponseWrapper: このクラスは、レスポンス・ラッパー・インタフェースの基本実装です。
- PrePostFilter: このクラスはフィルタを実装します。

この例では、HttpServletResponseWrapper クラスを使用して、ターゲットに送信される前のレスポンスをラップします。このクラスは、ServletResponse オブジェクトのラッパーとして動作するオブジェクトです（ソフトウェア・デザイン関係の書籍に記載されている Decorator パターンを使用）。このオブジェクトは実際のレスポンスのラップに使用されるため、リクエストのターゲットがレスポンスを送信した後で、変更することができます。

この例で作成された HTTP サーブレットのレスポンス・ラッパーは、ユーザー定義のサーブレット出力ストリームを使用します。この出力ストリームによって、サーブレット（この例では、JSP ページ）がレスポンスを作成して送信した後でも、ラッパーは、レスポンス・データを操作することができます。通常、この操作は、サーブレット出力ストリームがクローズした後（基本的には、サーブレットが終了をコミットした後）では実行できません。この例にある ServletOutputStream クラスに対してフィルタ固有の拡張機能を実装するのはこのためです。

出力ストリーム : FilterServletOutputStream.java 別のリソースのレスポンスを操作するために FilterServletOutputStream クラスを使用します。このクラスは、標準的な java.io.OutputStream クラスの 3 つの write() メソッドをオーバーライドします。

次に、新規出力ストリームのコードを示します。

```
package com.acme.filter;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class FilterServletOutputStream extends ServletOutputStream {

  private DataOutputStream stream;

  public FilterServletOutputStream(OutputStream output) {
    stream = new DataOutputStream(output);
  }

  public void write(int b) throws IOException  {
    stream.write(b);
  }

  public void write(byte[] b) throws IOException  {
    stream.write(b);
  }

  public void write(byte[] b, int off, int len) throws IOException  {
    stream.write(b,off,len);
  }

}
```

このコードをデフォルトの Web アプリケーションのルート・ディレクトリ（デフォルトは j2ee/home/default-web-app）の下にある次のディレクトリに保存し、コンパイルします。

```
/WEB-INF/classes/com/acme/filter
```

サーブレット・レスポンス・ラッパー : GenericResponseWrapper.java ユーザー定義の ServletOutputStream クラスを使用するには、レスポンス・オブジェクトとして動作可能なクラスを実装します。このラッパー・オブジェクトは、生成された元のレスポンスのかわりにクライアントに返信されます。

ラッパーは、コンテンツのタイプや長さを取得するためのユーティリティ・メソッドを実装する必要があります。GenericResponseWrapper クラスでは、次のように行われます。

```
package com.acme.filter;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class GenericResponseWrapper extends HttpServletResponseWrapper {
  private ByteArrayOutputStream output;
  private int contentLength;
  private String contentType;

  public GenericResponseWrapper(HttpServletResponse response) {
    super(response);
    output=new ByteArrayOutputStream();
  }

  public byte[] getData() {
    return output.toByteArray();
```

```
  }

  public ServletOutputStream getOutputStream() {
    return new FilterServletOutputStream(output);
  }

  public PrintWriter getWriter() {
    return new PrintWriter(getOutputStream(),true);
  }

  public void setContentLength(int length) {
    this.contentLength = length;
    super.setContentLength(length);
  }

  public int getContentLength() {
    return contentLength;
  }

  public void setContentType(String type) {
    this.contentType = type;
    super.setContentType(type);
  }


  public String getContentType() {
    return contentType;
  }
}
```

このコードをデフォルトの Web アプリケーションのルート・ディレクトリ（デフォルトは `j2ee/home/default-web-app`）の下にある次のディレクトリに保存し、コンパイルします。

```
/WEB-INF/classes/com/acme/filter
```

フィルタの作成 このフィルタは、ターゲットの起動後にレスポンスにコンテンツを追加します。このフィルタは、3-4 ページの「汎用フィルタ」のフィルタを拡張します。

次に、フィルタ・コード `PrePostFilter.java` を示します。

```
package com.acme.filter;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class PrePostFilter extends MyGenericFilter {

  public void doFilter(final ServletRequest request,
                       final ServletResponse response,
                       FilterChain chain)
       throws IOException, ServletException {
  OutputStream out = response.getOutputStream();
  out.write("<HR>PRE<HR>".getBytes());
  GenericResponseWrapper wrapper =
         new GenericResponseWrapper((HttpServletResponse) response);
  chain.doFilter(request,wrapper);
  out.write(wrapper.getData());
  out.write("<HR>POST<HR>".getBytes());
  out.close();
  }
}
```

このコードをデフォルトの Web アプリケーションのルート・ディレクトリ（デフォルトは j2ee/home/default-web-app）の下にある次のディレクトリに保存し、コンパイルします。

/WEB-INF/classes/com/acme/filter

前述の例と同様に、単純な JSP ページを作成します。

```
<HTML>
<HEAD>
<TITLE>Filter Example 3</TITLE>
</HEAD>
<BODY>
This is a testpage. You should see<br>
this text when you invoke filter3.jsp, <br>
as well as the additional material added<br>
by the PrePostFilter.
<br>
</BODY>
</HTML>
```

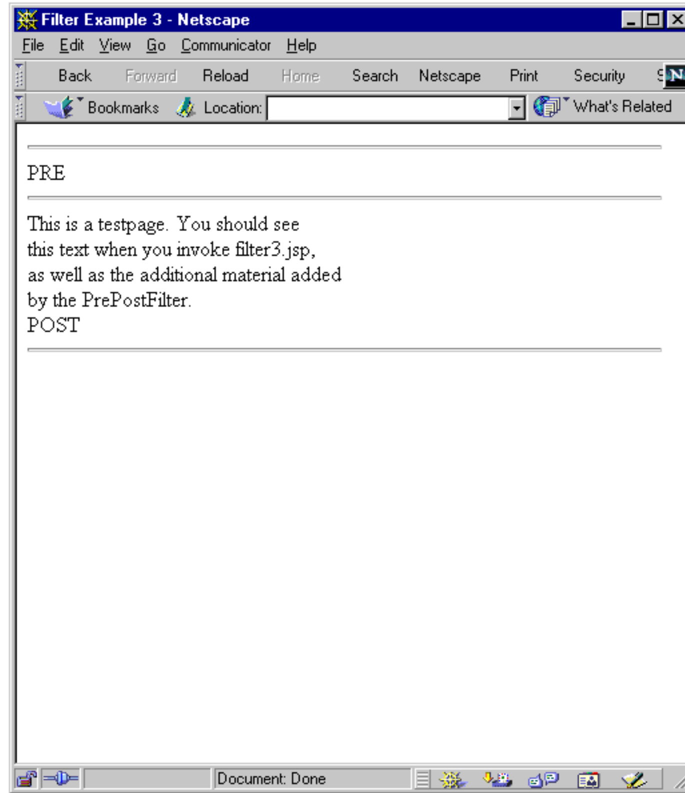
この JSP コードをデフォルトの Web アプリケーションのルート・ディレクトリにある filter3.jsp に保存します。

フィルタの構成 message フィルタの構成後に、次の <filter> 要素を web.xml に追加します。

```
<!-- Filter Example 3 -->
<filter>
  <filter-name>prePost</filter-name>
  <display-name>prePost</display-name>
  <filter-class>com.acme.filter.PrePostFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>prePost</filter-name>
  <url-pattern>/filter3.jsp</url-pattern>
</filter-mapping>
<!-- end Filter Example 3 -->
```

例 3 の実行 Web ブラウザでサーブレットを起動します。図 3-4 に示すようなページが表示されます。

図 3-4 例 3 の出力



イベント・リスナー

サーブレット仕様には、イベント・リスナーを使用して Web アプリケーションの主要イベントを追跡する機能が含まれています。この機能を使用すると、イベントの状態に基づいたリソース管理と自動処理をより効果的に実行できます。次の各項で、サーブレットのイベント・リスナーについて説明します。

- イベント・カテゴリとイベント・リスナー・インタフェース
- 代表的なイベント・リスナーの使用例
- イベント・リスナーの宣言と起動
- イベント・リスナーのコーディングとデプロイのガイドライン
- イベント・リスナーのメソッドと関連クラス
- イベント・リスナーの例

イベント・カテゴリとイベント・リスナー・インタフェース

サーブレットのイベントには次の 2 つのレベルがあります。

- サーブレット・コンテキスト・レベル（アプリケーション・レベル）のイベント

  このイベントには、アプリケーションのサーブレット・コンテキスト・オブジェクトのレベルで保持されたリソースまたは状態が含まれます。

- セッション・レベルのイベント

  このイベントには、単一ユーザー・セッションからの一連のリクエスト、つまり HTTP セッション・オブジェクトに関連付けられたリソースまたは状態が含まれます。

これら 2 つの各レベルには、次の 2 つのイベント・カテゴリがあります。

- ライフ・サイクルの変更
- 属性の変更

4 つのイベント・カテゴリごとに 1 つ以上のイベント・リスナー・クラスを作成できます。1 つのリスナー・クラスは、複数のイベント・カテゴリを監視できます。

イベント・リスナー・クラスを作成するには、`javax.servlet` パッケージまたは `javax.servlet.http` パッケージの適切なインタフェース（複数可）を実装します。表 3-1 は、4 つのカテゴリと関連するインタフェースをまとめたものです。

表 3-1　イベント・リスナーのカテゴリとインタフェース

イベント・カテゴリ	イベントの説明	Java インタフェース
サーブレット・コンテキストのライフ・サイクルの変更	サーブレット・コンテキストの作成（最初のリクエストに対応できる時点） サーブレット・コンテキストの緊急停止	javax.servlet.ServletContextListener
サーブレット・コンテキスト属性の変更	サーブレット・コンテキスト属性の追加 サーブレット・コンテキスト属性の削除 サーブレット・コンテキスト属性の置換	javax.servlet.ServletContextAttributeListener
セッション・ライフ・サイクルの変更	セッションの作成 セッションの無効化 セッションのタイムアウト	javax.servlet.http.HttpSessionListener
セッション属性の変更	セッション属性の追加 セッション属性の削除 セッション属性の置換	javax.servlet.http.HttpSessionAttributeListener

代表的なイベント・リスナーの使用例

データベースにアクセスするサーブレットで構成された Web アプリケーションについて考えます。イベント・リスナーの代表的な使用例は、サーブレット・コンテキストのライフ・サイクル・イベント・リスナーを作成し、データベース接続を管理することです。このリスナーは次のように機能します。

1. リスナーにアプリケーションの起動が通知されます。
2. アプリケーションはデータベースにログインし、接続オブジェクトをサーブレット・コンテキストに格納します。
3. サーブレットは、データベース接続を使用して SQL 操作を実行します。
4. リスナーにアプリケーションの緊急停止（Web サーバーのシャットダウンまたは Web サーバーからのアプリケーションの削除）が通知されます。
5. アプリケーションがシャットダウンする前に、リスナーはデータベース接続をクローズします。

イベント・リスナーの宣言と起動

イベント・リスナーの宣言は、アプリケーションの `web.xml` デプロイメント・ディスクリプタで、トップレベルの `<web-app>` 要素の下にある `<listener>` 要素を使用して行います。リスナーごとに独自の `<listener>` 要素と、そのクラス名を指定する `<listener-class>` サブ要素があります。各イベント・カテゴリ内に、イベント・リスナーをアプリケーションの実行時に起動する順序で指定する必要があります。

アプリケーションの起動後およびアプリケーションが最初のリクエストを処理する前に、サーブレット・コンテナは、宣言した各リスナー・クラスのインスタンスを作成および登録します。各イベント・カテゴリのリスナーは、宣言された順序で登録されます。その結果、アプリケーションの実行時に、各カテゴリのイベント・リスナーは、登録順に起動されます。すべてのリスナーは、アプリケーションの最後のリクエストが処理されるまで、アクティブ状態のままです。

アプリケーションのシャットダウン時は、宣言順序とは逆の順序で、最初にセッション・イベント・リスナーに通知され、次に、アプリケーション・イベント・リスナーに通知されます。

次に、Sun 社の Java サーブレット仕様、バージョン 2.3 によるイベント・リスナーの宣言の例を示します。

```
<web-app>
   <display-name>MyListeningApplication</display-name>
   <listener>
      <listener-class>com.acme.MyConnectionManager</listenerclass>
   </listener>
   <listener>
      <listener-class>com.acme.MyLoggingModule</listener-class>
   </listener>
   <servlet>
      <display-name>RegistrationServlet</display-name>
      ...
   </servlet>
</web-app>
```

MyConnectionManager および MyLoggingModule の両方が ServletContextListener インタフェースを実装し、MyLoggingModule は HttpSessionListener インタフェースも実装していると想定します。

アプリケーションの実行時、両方のリスナーにサーブレット・コンテキストのライフ・サイクル・イベントが通知され、MyLoggingModule リスナーにはセッション・ライフ・サイクル・イベントも通知されます。サーブレット・コンテキストのライフ・サイクル・イベントは、最初に MyConnectionManager リスナーに通知されます。これは、宣言順序によるためです。

イベント・リスナーのコーディングとデプロイのガイドライン

イベント・リスナー・クラスに対する次のルールとガイドラインに注意してください。

- マルチスレッド・アプリケーションでは、属性の変更が同時に発生する可能性があります。その結果生じた通知をサーブレット・コンテナで同期化する必要はありません。このような状況では、リスナー・クラス自体がデータの整合性を維持します。
- 各リスナー・クラスに、パブリックの 0（ゼロ）引数コンストラクタを設定する必要があります。
- 各リスナー・クラス・ファイルは、/WEB-INF/classes の下、または /WEB-INF/lib の JAR ファイル内のいずれかにあるアプリケーションの WAR ファイルにパッケージする必要があります。

> **注意：** 分散環境では、イベント・リスナーのスコープは、各 JVM の各デプロイメント・ディスクリプタの宣言に対するスコープです。分散 Web コンテナでは、サーブレット・コンテキスト・イベントまたはセッション・イベントをその他の JVM に伝播する必要はありません。サーブレット仕様ではこの点について説明しています。

イベント・リスナーのメソッドと関連クラス

この項では、サーブレット・コンテキスト・イベントまたはセッション・イベントの発生時にサーブレット・コンテナがコールするイベント・リスナーのメソッドを示します。これらのメソッドは、様々なタイプのイベント・オブジェクトを入力として取得するため、各イベント・クラスとそのメソッドについても説明します。

ServletContextListener メソッド、ServletContextEvent クラス

ServletContextListener インタフェースは、次のメソッドを指定します。

- void contextInitialized(ServletContextEvent sce)

  サーブレット・コンテナはこのメソッドをコールして、サーブレット・コンテキストの作成が完了し、アプリケーションでリクエストを処理できることをリスナーに通知します。

- void contextDestroyed(ServletContextEvent sce)

  サーブレット・コンテナはこのメソッドをコールして、アプリケーションが停止されることをリスナーに通知します。

サーブレット・コンテナは、ServletContextListener メソッドのコールに対して入力される javax.servlet.ServletContextEvent オブジェクトを作成します。ServletContextEvent クラスには、リスナーがコール可能な次のメソッドが含まれています。

- ServletContext getServletContext()

  このメソッドを使用して、作成済または破棄対象のサーブレット・コンテキス・オブジェクトを取得します。このオブジェクトから必要な情報を取得できます。javax.servlet.ServletContext インタフェースの詳細は、1-6 ページの「サーブレット・コンテキストの概要」を参照してください。

ServletContextAttributeListener メソッド、ServletContextAttributeEvent クラス

ServletContextAttributeListener インタフェースは、次のメソッドを指定します。

- void attributeAdded(ServletContextAttributeEvent scae)

  サーブレット・コンテナはこのメソッドをコールして、属性がサーブレット・コンテキストに追加されたことをリスナーに通知します。

- void attributeRemoved(ServletContextAttributeEvent scae)

  サーブレット・コンテナはこのメソッドをコールして、属性がサーブレット・コンテキストから削除されたことをリスナーに通知します。

- void attributeReplaced(ServletContextAttributeEvent scae)

  サーブレット・コンテナはこのメソッドをコールして、属性がサーブレット・コンテキストで置換されたことをリスナーに通知します。

サーブレット・コンテナは、ServletContextAttributeListener メソッドのコールに対して入力される javax.servlet.ServletContextAttributeEvent オブジェクトを作成します。ServletContextAttributeEvent クラスには、リスナーがコール可能な次のメソッドが含まれています。

- String getName()

  このメソッドを使用して、追加、削除または置換された属性の名前を取得します。

- Object getValue()

  このメソッドを使用して、追加、削除または置換された属性の値を取得します。置換された属性の場合、このメソッドは置換後の値ではなく、置換前の値を戻します。

HttpSessionListener メソッド、HttpSessionEvent クラス

HttpSessionListener インタフェースは、次のメソッドを指定します。

- void sessionCreated(HttpSessionEvent hse)

  サーブレット・コンテナはこのメソッドをコールして、セッションが作成されたことをリスナーに通知します。

- `void sessionDestroyed(HttpSessionEvent hse)`

  サーブレット・コンテナはこのメソッドをコールして、セッションが破棄されたことをリスナーに通知します。

サーブレット・コンテナは、`HttpSessionListener` メソッドのコールに対して入力される `javax.servlet.http.HttpSessionEvent` オブジェクトを作成します。`HttpSessionEvent` クラスには、リスナーがコール可能な次のメソッドが含まれています。

- `HttpSession getSession()`

  このメソッドを使用して、作成済または破棄済のセッション・オブジェクトを取得します。このオブジェクトから必要な情報を取得できます。`javax.servlet.http.HttpSession` インタフェースの詳細は、1-5 ページの「サーブレット・セッションの概要」を参照してください。

HttpSessionAttributeListener メソッド、HttpSessionBindingEvent クラス

`HttpSessionAttributeListener` インタフェースは、次のメソッドを指定します。

- `void attributeAdded(HttpSessionBindingEvent hsbe)`

  サーブレット・コンテナはこのメソッドをコールして、属性がセッションに追加されたことをリスナーに通知します。

- `void attributeRemoved(HttpSessionBindingEvent hsbe)`

  サーブレット・コンテナはこのメソッドをコールして、属性がセッションから削除されたことをリスナーに通知します。

- `void attributeReplaced(HttpSessionBindingEvent hsbe)`

  サーブレット・コンテナはこのメソッドをコールして、属性がセッションで置換されたことをリスナーに通知します。

サーブレット・コンテナは、`HttpSessionAttributeListener` メソッドのコールに対して入力される `javax.servlet.http.HttpSessionBindingEvent` オブジェクトを作成します。`HttpSessionBindingEvent` クラスには、リスナーがコール可能な次のメソッドが含まれています。

- `String getName()`

  このメソッドを使用して、追加、削除または置換された属性の名前を取得します。

- `Object getValue()`

  このメソッドを使用して、追加、削除または置換された属性の値を取得します。置換された属性の場合、このメソッドは置換後の値ではなく、置換前の値を戻します。

- `HttpSession getSession()`

  このメソッドを使用して、属性の変更が発生したセッション・オブジェクトを取得します。

イベント・リスナーの例

この項では、サーブレット・コンテキストのライフ・サイクル・イベント・リスナーおよびセッションのライフ・サイクル・イベント・リスナーを使用したサンプル・コードを示します。次のコンポーネントが含まれます。

- `SessionLifeCycleEventExample`: イベント・リスナー・クラス。`ServletContextListener` インタフェースおよび `HttpSessionListener` インタフェースを実装します。
- `SessionCreateServlet`: このサーブレットは、HTTP セッションを作成します。
- `SessionDestroyServlet`: このサーブレットは、HTTP セッションを破棄します。
- `index.jsp`: アプリケーションの「ようこそ」ページ（ユーザー・インタフェース）。このページから `SessionCreateServlet` または `SessionDestroyServlet` を起動できます。

- web.xml: デプロイメント・ディスクリプタ。ここでサーブレットとリスナーが宣言されます。

このアプリケーションをダウンロードして実行するには、次のリンク先で、サーブレットのライフ・サイクル・イベントの使用方法に関するドキュメントを参照してください。

http://www.oracle.com/technology/tech/java/oc4j/htdocs/oc4j-how-to.html

(これらの例は、旧バージョンの OC4J について記載されていますが、機能的には 10.1.2 の実装と変わりません。)

Oracle Technology Network の会員登録が必要ですが、登録は無料です。

「ようこそ」ページ : index.jsp

次に「ようこそ」ページを示します。このユーザー・インタフェースを使用すると、「**Create New Session**」リンクをクリックしてセッション作成サーブレットを起動でき、「**Destroy Current Session**」リンクをクリックしてセッション破棄サーブレットを起動できます。

```
<%@page session="false" %>
<H2>OC4J - HttpSession Event Listeners </H2>
<P>
This example demonstrates the use of the HttpSession Event and Listener that is
new with the Java Servlet 2.3 API.
</P>
<P>
[<a href="servlet/SessionCreateServlet">Create New Session</A>]  
[<a href="servlet/SessionDestroyServlet">Destroy Current Session</A>]
</P>
<P>
Click the Create link above to start a new HttpSession. An HttpSession
listener has been configured for this application. The servler container
will send an event to this listener when a new session is created or
destroyed. The output from the event listener will be visible in the
console window from where OC4J was started.
</P>
```

注意： 同一のクライアントからセッションを作成した後に「**Create New Session**」リンクを再度クリックしても、新規セッション・オブジェクトは作成されません。ただし、セッションがタイムアウトになった場合や、明示的に破棄した場合は異なります。

デプロイメント・ディスクリプタ : web.xml

サーブレットとイベント・リスナーは、web.xml ファイルで宣言されます。その結果、アプリケーションの起動時に SessionLifeCycleEventExample がインスタンス化され、登録されます。このため、サーブレット・コンテキストまたはセッションのライフ・サイクル・イベントの発生時に、このクラスのメソッドが、必要に応じて、サーブレット・コンテナによって自動的にコールされます。次に、web.xml のエントリを示します。

```
<web-app>
   <listener>
      <listener-class>SessionLifeCycleEventExample</listener-class>
   </listener>
   <servlet>
      <servlet-name>sessioncreate</servlet-name>
      <servlet-class>SessionCreateServlet</servlet-class>
   </servlet>
   <servlet>
      <servlet-name>sessiondestroy</servlet-name>
      <servlet-class>SessionDestroyServlet</servlet-class>
   </servlet>
   <welcome-file-list>
```

```
      <welcome-file>index.jsp</welcome-file>
   </welcome-file-list>
</web-app>
```

リスナー・クラス : SessionLifeCycleEventExample

この項は、リスナー・クラスを示します。このクラスの sessionCreated() メソッドは、HTTP セッションの作成時にサーブレット・コンテナによってコールされます。このコールは、index.jsp の「**Create New Session**」リンクを選択すると実行されます。sessionCreated() のコール時に、log() メソッドがコールされ、新規セッションの ID を示す "CREATE" メッセージが出力されます。

sessionDestroyed() メソッドは、HTTP セッションの破棄時にコールされます。このコールは、「**Destroy Current Session**」リンクをクリックすると実行されます。sessionDestroyed() のコール時に、log() メソッドがコールされ、終了したセッションの ID と存続期間を示す "DESTROY" メッセージが出力されます。

```
import javax.servlet.http.*;
import javax.servlet.*;

public class SessionLifeCycleEventExample
   implements ServletContextListener, HttpSessionListener
{
   /* A listener class must have a zero-argument constructor: */
   public SessionLifeCycleEventExample()
   {
   }

   ServletContext servletContext;

   /* Methods from the ServletContextListener */
   public void contextInitialized(ServletContextEvent sce)
   {
      servletContext = sce.getServletContext();
   }

   public void contextDestroyed(ServletContextEvent sce)
   {
   }

   /* Methods for the HttpSessionListener */
   public void sessionCreated(HttpSessionEvent hse)
   {
      log("CREATE",hse);
   }
   public void sessionDestroyed(HttpSessionEvent hse)
   {

         HttpSession _session = hse.getSession();
         long _start = _session.getCreationTime();
         long _end = _session.getLastAccessedTime();
         String _counter = (String)_session.getAttribute("counter");
         log("DESTROY, Session Duration:"
                   + (_end - _start) + "(ms) Counter:" + _counter, hse);
   }

   protected void log(String msg, HttpSessionEvent hse)
   {
      String _ID = hse.getSession().getId();
      log("SessionID:" + _ID + "    " + msg);
   }

   protected void log(String msg)
   {
```

```
      System.out.println("[" + getClass().getName() + "] " + msg);
   }
}
```

セッション作成サーブレット : SessionCreateServlet.java

このサーブレットは、index.jsp の「**Create New Session**」リンクをクリックすると起動します。このサーブレットが起動すると、サーブレット・コンテナはリクエスト・オブジェクトおよび関連するセッション・オブジェクトを作成します。セッション・オブジェクトが作成されると、サーブレット・コンテナは、イベント・リスナー・クラスの sessionCreated() メソッドをコールします。

```
import java.io.*;
import java.util.Enumeration;
import java.util.Date;

import javax.servlet.*;
import javax.servlet.http.*;


public class SessionCreateServlet extends HttpServlet {

    public void doGet (HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException
      {
         //Get the session object
         HttpSession session = req.getSession(true);

        // set content type and other response header fields first
        res.setContentType("text/html");

        // then write the data of the response
        PrintWriter out = res.getWriter();

        String _sval = (String)session.getAttribute("counter");
        int _counter=1;

        if(_sval!=null)
        {
           _counter=Integer.parseInt(_sval);
           _counter++;
      }


      session.setAttribute("counter",String.valueOf(_counter));

      out.println("<HEAD><TITLE> " + "Session Created Successfully ..
         Look at OC4J Console to see whether the HttpSessionEvent invoked "
         + "</TITLE></HEAD><BODY>");
      out.println("<P>[<A HREF=¥"SessionCreateServlet¥">Reload</A>] ");
      out.println("[<A HREF=¥"SessionDestroyServlet¥">Destroy Session</A>]");
      out.println("<h2>Session created Successfully</h2>");
      out.println("Look at the OC4J Console to see whether the HttpSessionEvent
                  was invoked");
      out.println("<h3>Session Data:</h3>");
      out.println("New Session: " + session.isNew());
      out.println("<br>Session ID: " + session.getId());
      out.println("<br>Creation Time: " + new Date(session.getCreationTime()));
      out.println("<br>Last Accessed Time: " +
                   new Date(session.getLastAccessedTime()));
      out.println("<BR>Number of Accesses: " + session.getAttribute("counter"));

   }
}
```

セッション破棄サーブレット : SessionDestroyServlet.java

このサーブレットは、index.jsp の「**Destroy Current Session**」リンクをクリックすると起動します。このサーブレットが起動すると、セッション・オブジェクトの invalidate() メソッドがコールされます。次に、サーブレット・コンテナは、イベント・リスナー・クラスの sessionDestroyed() メソッドをコールします。

```
import java.io.*;
import java.util.Enumeration;

import javax.servlet.*;
import javax.servlet.http.*;

public class SessionDestroyServlet extends HttpServlet {

    public void doGet (HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException
      {
         //Get the session object
         HttpSession session = req.getSession(true);
         // Invalidate Session
         session.invalidate();

         // set content type and other response header fields first
         res.setContentType("text/html");

         // then write the data of the response
         PrintWriter out = res.getWriter();

         out.println("<HEAD><TITLE> " + "Session Destroyed Successfully ..
           Look at OC4J Console to see whether the HttpSessionEvent invoked "
           + "</TITLE></HEAD><BODY>");
         out.println("<P>[<A HREF=¥"../index.jsp¥">Restart</A>]");
         out.println("<h2> Session Destroyed Successfully</h2>");
         out.println("Look at the OC4J Console to see whether the
                      HttpSessionEvent was invoked");
         out.close();
   }
}
```

4

サーブレットからの JDBC および EJB コール

動的 Web アプリケーションは、通常、コンテンツを提供するために、データベースにアクセスします。この章では、データベース接続性の Java 標準である JDBC と、トランザクションによるセキュアなサーバー・サイド処理に使用される Enterprise JavaBeans をサーブレットで使用する方法について説明します。次の項が含まれます。

- サーブレットでの JDBC の使用
- サーブレットからの EJB コール

サーブレットでの JDBC の使用

サーブレットは、JDBC ドライバを使用してデータベースにアクセス可能です。JDBC の使用方法としては、OC4J データ・ソースを使用してデータベース接続を取得する方法をお薦めします。OC4J データ・ソースの詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』を参照してください。JDBC の詳細は、『Oracle Database JDBC 開発者ガイドおよびリファレンス』を参照してください。

データベース問合せサーブレット

サーブレットの利点の 1 つに、データベースからデータを取得できる点があげられます。サーブレットは、データベースから情報を取得して動的 HTML を生成し、それをクライアントに返すことが可能です。また、HTTP リクエスト内でサーブレットに渡された情報に基づき、データベースを更新することも可能です。

この例では、ユーザーから HTML フォーム経由で情報を取得し、その情報をサーブレットに渡すサーブレットを示します。このサーブレットは、SQL 文を完成させて実行し、サンプルの Human Resources（HR）スキーマに問合せを行って、そのリクエスト・データに基づき情報を取得します。

サーブレットは、様々な方法でクライアントから情報を取得できます。この例では、HTTP リクエストから問合せ文字列を読み取ります。

注意： この例では、内容をわかりやすくするために、次のことを前提としています。

- データベースがインストール済であり、`localhost` からポート 1521 でアクセス可能であること。
- OC4J スタンドアロンおよび OC4J のデフォルト Web アプリケーション（デフォルト・コンテキスト・ルートは「/」）を使用していること。

HTML フォーム

Web ブラウザは、Web リスナー経由で提供されるページ内のフォームにアクセスします。次の HTML をファイル `EmpInfo.html` にコピーします。

```
<html>

<head>
<title>Query the Employees Table</title>
</head>

<body>
<form method=GET ACTION="/servlet/GetEmpInfo">
The query is<br>
SELECT LAST_NAME, EMPLOYEE_ID FROM EMPLOYEES WHERE LAST NAME LIKE ?.<p>

Enter the WHERE clause ? parameter (use % for wildcards).<br>
Example: 'S%':<br>
<input type=text name="queryVal">
<p>
<input type=submit>
</form>

</body>
</html>
```

次に、このファイルを OC4J のデフォルトの Web アプリケーションのルート・ディレクトリ（デフォルトは `j2ee/home/default-web-app`）に保存します。

サーブレット・コード : GetEmpInfo

前述の HTML ページによってコールされるサーブレットは、問合せ文字列から入力を取得します。入力は、SELECT 文の WHERE 句により完了します。次に、サーブレットは、データベースへの問合せを構成するために、この入力を追加します。このサーブレットのコードの大半は、データベース・サーバーに接続して問合せ行を取得および処理するために必要な JDBC 文で構成されています。

このサーブレットは、JNDI を使用して、データ・ソースを 1 回だけルックアップするために init() メソッドを使用します。データ・ソースのルックアップでは、次のようなデータ・ソースが OC4J 構成ファイルのディレクトリにある data-sources.xml ファイルに定義されていることを前提としています。(URL に適切なサービス名が使用されていることを確認してください。)

```
<data-source
        class="com.evermind.sql.DriverManagerDataSource"
        name="OracleDS"
        location="jdbc/OracleCoreDS"
        xa-location="jdbc/xa/OracleXADS"
        ejb-location="jdbc/OracleDS"
        connection-driver="oracle.jdbc.driver.OracleDriver"
        username="hr"
        password="hr"
        url="jdbc:oracle:thin:@localhost:1521/myservice"
        inactivity-timeout="30"
/>
```

JNDI を使用したエミュレートされたデータ・ソースのルックアップで ejb-location の JNDI 名のみを指定することをお薦めします。データ・ソースの詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』を参照してください。

また、この例では、web.xml ファイルでデータ・ソース定義が次のとおりであることを前提としています。

```
  <resource-ref>
      <res-auth>Container</res-auth>
      <res-ref-name>jdbc/OracleDS</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
   </resource-ref>
```

次に、サーブレット・コードを示します。

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;  // for JNDI
import javax.sql.*;     // extended JDBC interfaces (such as data sources)
import java.sql.*;      // standard JDBC interfaces
import java.io.*;

public class GetEmpInfo extends HttpServlet {

  DataSource ds = null;
  Connection conn = null;

  public void init() throws ServletException {
    try {
      InitialContext ic = new InitialContext();  // JNDI initial context
      ds = (DataSource) ic.lookup("jdbc/OracleDS"); // JNDI lookup
      conn = ds.getConnection();  // database connection through data source
    }
    catch (SQLException se) {
      throw new ServletException(se);
    }
    catch (NamingException ne) {
      throw new ServletException(ne);
```

```
    }
  }

  public void doGet (HttpServletRequest req, HttpServletResponse resp)
                   throws ServletException, IOException {

/* Get the user-specified WHERE clause from the HTTP request, then   */
/* construct the SQL query.                                          */
    String queryVal = req.getParameter("queryVal");
    String query =
      "select last_name, employee_id from employees " +
      "where last_name like " + queryVal;

    resp.setContentType("text/html");

    PrintWriter out = resp.getWriter();
    out.println("<html>");
    out.println("<head><title>GetEmpInfo</title></head>");
    out.println("<body>");

/* Create a JDBC statement object, execute the query, and set up     */
/* HTML table formatting for the output.                             */
    try {
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery(query);

        out.println("<table border=1 width=50%>");
        out.println("<tr><th width=75%>Last Name</th><th width=25%>Employee " +
                   "ID</th></tr>");

/* Loop through the results. Use the ResultSet getString() and       */
/* getInt() methods to retrieve the individual data items.           */
        int count=0;
        while (rs.next()) {
         count++;
         out.println("<tr><td>" + rs.getString(1) + "</td><td>" +rs.getInt(2) +
                    "</td></tr>");

        }
         out.println("</table>");
         out.println("<h3>" + count + " rows retrieved</h3>");

      rs.close();
      stmt.close();
    }
    catch (SQLException se) {
      se.printStackTrace(out);
    }

    out.println("</body></html>");
  }

  public void destroy() {
    try {
      conn.close();
    }
    catch (SQLException se) {
      se.printStackTrace();
    }
  }
}
```

データベース問合せサーブレットのデプロイおよびテスト

この例をデプロイするには、HTML ファイルを OC4J のデフォルトの Web アプリケーションのルート・ディレクトリ（デフォルトは j2ee/home/default-web-app）に保存し、Java サーブレットをデフォルトの Web アプリケーションの /WEB-INF/classes ディレクトリに保存してください。GetEmpInfo.java ファイルは、フォームがサーブレットを起動したときに、自動的にコンパイルされます。

この例を、OC4J スタンドアロンなどで、OC4J リスナーを使用して直接テストするには、次のように、Web ブラウザから EmpInfo.html ページを起動します。

```
http://host:8888/EmpInfo.html
```

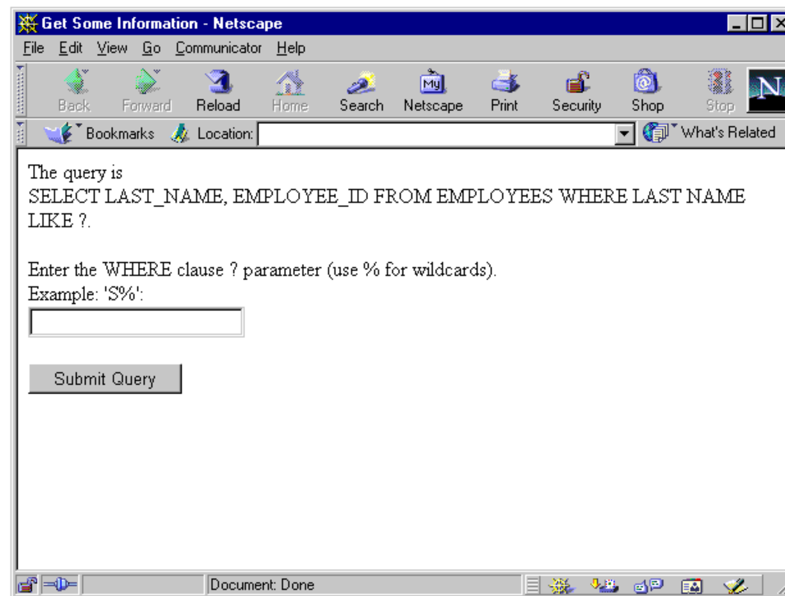
「/」は、OC4J スタンドアロンのデフォルトの Web アプリケーションのコンテキスト・パスとみなされます。

フォームを完成させて、「**Submit Query**」をクリックします。

注意： OC4J でのサーブレット起動に関する一般的な情報は、2-18 ページの「サーブレットの起動」を参照してください。

EmpInfo.html を起動すると、図 4-1 のようなブラウザ出力が表示されます。

図 4-1 社員情報の問合せ



フォームに S% と入力して「**Submit Query**」をクリックすると、GetEmpInfo サーブレットがコールされます。図 4-2 のような画面が表示されます。

図 4-2 社員情報の結果

Last Name	Employee ID
Sciarra	111
Stiles	138
Seo	139
Sully	157
Smith	159
Sewall	161
Smith	171
Sullivan	182
Sarchand	184

9 rows retrieved.

サーブレットからの EJB コール

サーブレットでは、Enterprise JavaBeans をコールして追加処理を実行することができます。一般的なアプリケーション設計では、サーブレットは、クライアント・リクエストの最初の処理を行うフロントエンドとして使用され、EJB は、データベースへのアクセスと更新を行うビジネス・ロジックを実行するためにコールされます。特に、コンテナによる永続性管理（CMP）の Entity Bean は、このような処理に向いています。

次の各項で、サーブレットから EJB を使用する代表的な例を説明します。

- サーブレットおよび EJB の概要
- EJB のローカル・ルックアップ
- 同一アプリケーション内の EJB リモート・ルックアップ
- アプリケーション外部の EJB リモート・ルックアップ

重要： この項の例では、開発時に OC4J をスタンドアロン・モードで使用していることを前提としています。これにより、JNDI ルックアップの URL が Oracle Application Server 環境での URL と異なる場合がありますが、それ以外の点ではサーブレット・コードに影響はありません。

注意：

- EJB の機能の詳細と、Oracle Application Server 環境でのサーブレットと EJB に関する例は、『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』を参照してください。
- OC4J では、JSP ページから EJB へのアクセスを便利にするために、EJB タグ・ライブラリが用意されています。詳細は、『Oracle Application Server Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

サーブレットおよび EJB の概要

次の各項で、サーブレットから EJB を使用する場合の考慮事項の概要を説明します。

- サーブレットおよび EJB の使用例
- EJB ローカル・インタフェースとリモート・インタフェース

サーブレットおよび EJB の使用例

この章のサーブレットと EJB の例では、次の 3 つの使用例について説明します。

- ローカル・ルックアップ : サーブレットは、同じ場所にある EJB、つまり同一のアプリケーションおよびホスト内に存在し、同一の JVM で実行されている EJB をコールします。サーブレットおよび EJB は、同一の EAR ファイル内、または親子関係にある EAR ファイル内でデプロイされています。この例では EJB ローカル・インタフェースを使用しています。これは、EJB 仕様のバージョン 2.0 で導入されています。4-8 ページの「EJB のローカル・ルックアップ」を参照してください。
- 同一アプリケーション内のリモート・ルックアップ : サーブレットは、同一アプリケーション内にあるけれども、別のホストに存在する EJB をコールします。（同一のアプリケーションが両方のホストにデプロイされています。）この場合、EJB リモート・インタフェースが必要です。サーブレットと EJB が同一アプリケーションの異なる層に存在する複数層のアプリケーションの場合、これに該当します。4-14 ページの「同一アプリケーション内の EJB リモート・ルックアップ」を参照してください。
- アプリケーション外のリモート・ルックアップ : サーブレットは、同一アプリケーションに存在しない EJB をコールします。これはリモート・ルックアップで、EJB リモート・インタフェースが必要です。EJB は同一のホストまたは異なるホストのいずれに存在する場合もありますが、同一の JVM では実行されていません。4-19 ページの「アプリケーション外部の EJB リモート・ルックアップ」を参照してください。

サーブレットと EJB 間の通信では、ルックアップに JNDI、EJB コールに RMI が使用され、ORMI（Oracle による RMI 実装）または IIOP（標準で相互操作可能な Internet Inter-ORB Protocol）のいずれかを介して通信が行われます。このマニュアルの例では、JNDI 初期コンテキスト・ファクトリには、`web.xml` で EJB 参照をサポートしている `ApplicationInitialContextFactory` クラス、およびサポートしていない `RMIInitialContextFactory` クラスが使用されています。場合によっては、`application-client.xml` ファイルで EJB 参照をサポートしている `ApplicationClientInitialContextFactory` も使用できます。JNDI と RMI を EJB とともに使用する方法の詳細は、『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』を参照してください。

リモート・ルックアップを行うには、JNDI 環境を設定する必要があります。これには、URL、ユーザー名およびパスワードが含まれます。この設定は、4-19 ページの「アプリケーション外部の EJB リモート・ルックアップ」で示されているように、通常はサーブレット・コードで行われますが、同一アプリケーション内のルックアップでは、`rmi.xml` ファイル内で設定される場合もあります。

同一アプリケーション内の、異なるホストでのリモート・ルックアップの場合、4-14 ページの「リモート・フラグの使用」に示すように、各ホストのアプリケーションの `orion-application.xml` ファイルで、`remote` フラグを正しく設定する必要もあります。

EJB が使用されているすべてのアプリケーションと同様、`ejb-jar.xml` ファイルに、EJB ごとにエントリが含まれている必要があります。

注意： この例では、ORMI での使用例のみを取り上げています。IIOP の使用方法の詳細は、『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』を参照してください。

EJB ローカル・インタフェースとリモート・インタフェース

EJB 仕様のバージョン 1.1 では、EJB には、必ず、`javax.ejb.EJBObject` インタフェースを拡張するリモート・インタフェースと `javax.ejb.EJBHome` インタフェースを拡張するホーム・インタフェースが備わっています。このモデルでは、すべての EJB はリモート・オブジェクトとして定義されているため、サーブレットまたは別のコール元モジュールが EJB と同じ場所へ配置されている場合は、EJB コールに不要なオーバーヘッドが追加されます。

注意： OC4J の `copy-by-value` 属性（`orion-ejb-jar.xml` ファイルの `<session-deployment>` 要素に属する）も、不要なオーバーヘッドの回避に関連しています。これは、EJB コールのすべての受信および送信パラメータをコピーするかどうかを指定します。詳細は、『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』を参照してください。

EJB 仕様のバージョン 2.0 では、同じ場所に配置されているルックアップのローカル・インタフェースに対するサポートが追加されています。この場合、EJB には、リモート・インタフェースではなく、`javax.ejb.EJBLocalObject` インタフェースを拡張するローカル・インタフェースが備わっています。そして、ホーム・インタフェースではなく、`javax.ejb.EJBLocalHome` インタフェースを拡張するローカル・ホーム・インタフェースが指定されます。

EJB リモート・インタフェースが関連するルックアップは RMI を使用するため、セキュリティなどの理由でオーバーヘッドが追加されます。ローカル・インタフェースを使用すると、RMI やその他のオーバーヘッドが回避されます。

注意：

- EJB は、ローカルおよびリモート・インタフェースの両方を備えることができます。この項の例では、ローカル・インタフェースまたはリモート・インタフェースのいずれかを使用しており、両方は使用していません。
- このマニュアルでは、ローカル・ルックアップという単語は、同一の JVM 内の同じ場所に配置されているルックアップを指します。「ローカル・ルックアップ」と「ローカル・インタフェース」を混同しないでください。通常ローカル・ルックアップではローカル・インタフェースが使用されますが、かわりにリモート・インタフェースが使用される場合もあります。（EJB 仕様のバージョン 2.0 より前のバージョンでは、この方法でローカル・ルックアップを実行する必要がありました。）

EJB のローカル・ルックアップ

この項では、ローカル・インタフェースを使用し、同じ場所に配置された単一の EJB（`HelloBean`）をコールする単一のサーブレット（`HelloServlet`）の例を示します。これは、最も単純なサーブレットと EJB の使用例です。

次に、サーブレット・コードの作成手順を示します。

1. Bean のホーム・インタフェースおよびリモート・インタフェースにアクセスするための EJB パッケージをインポートします。また、JNDI 用に `javax.naming`、RMI 用に `javax.rmi` をインポートする点に注意してください。
2. サーブレットからメッセージを出力します。
3. デフォルトのエラー・メッセージを持つ出力文字列を作成します。
4. JNDI を使用して EJB ローカル・ホーム・インタフェースをルックアップします。
5. ローカル・ホームから EJB ローカル・オブジェクトを作成します。

6. ローカル・オブジェクト上で helloWorld() メソッドを起動し、EJB メッセージを Java 文字列で出力します。
7. EJB からメッセージを出力します。

次の各項で、例のすべての内容を説明します。

- ローカル・ルックアップ用のサーブレットと EJB のアプリケーション・コード
- ローカル・ルックアップの構成およびデプロイ
- サーブレットと EJB のアプリケーションの起動

詳細な説明およびローカル・インタフェースの別の完全な使用例は、次の場所で EJB 2.0 ローカル・インタフェースの使用方法に関するドキュメントを参照してください。

http://www.oracle.com/technology/tech/java/oc4j/htdocs/oc4j-how-to.html

(これは元々、旧バージョンの OC4J について記載されていますが、機能的には 10.1.2 の実装と変わりません。)

ローカル・ルックアップ用のサーブレットと EJB のアプリケーション・コード

この項では、ローカル・インタフェースを使用し、同じ場所に配置された EJB をコールするサーブレットのコードを示します。これには、サーブレット・コード、EJB コードおよび EJB インタフェース・コードが含まれます。特に、太字の部分に注意してください。

サーブレット・コード : HelloServlet この項では、サーブレット・コードを示します。ローカル・ルックアップの場合、デフォルトの JNDI コンテキストが使用されます。

デフォルトでは、このサーブレットは JNDI 初期コンテキスト・ファクトリに ApplicationInitialContextFactory を使用します。したがって、web.xml ファイルでは EJB 参照が検索されます。JNDI ルックアップの java:comp 構文は、アプリケーション内に EJB 用に定義された参照が存在することを示します。この例では web.xml ファイルです。

JNDI 初期コンテキスト・ファクトリ・クラスの詳細は、『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』を参照してください。

```
package myServlet;

// Step 1: Import the EJB package.
import myEjb.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;       // for JNDI
import javax.rmi.*;          // for RMI, including PortableRemoteObject
import javax.ejb.CreateException;

public class HelloServlet extends HttpServlet {

  public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("<html><head><title>Hello from Servlet</title></head>");
    // Step 2: Print a message from the servlet.
    out.println("<body><h1>Hello from hello servlet!</h1></body>");

    // Step 3: Create an output string, with an error default.
    String s = "If you see this message, the ejb was not invoked properly!!";
    // Step 4: Use JNDI to look up the EJB local home interface.
    try {
      InitialContext ic = new InitialContext();
```

```
        HelloLocalHome hlh = (HelloLocalHome)ic.lookup
                              ("java:comp/env/ejb/HelloBean");

        // Step 5: Create the EJB local interface object.
        HelloLocal hl = (HelloLocal)hlh.create();
        // Step 6: Invoke the helloWorld() method on the local object.
        s = hl.helloWorld();
      } catch (NamingException ne) {
        System.out.println("Could not locate the bean.");
      } catch (CreateException ce) {
        System.out.println("Could not create the bean.");
      } catch (Exception e) {
        // Unexpected exception; send back to client for now.
        throw new ServletException(e);
      }
      // Step 7: Print the message from the EJB.
      out.println("<br>" + s);
      out.println("</html>");
  }
}
```

EJB コード : HelloBean ステートフル Session Bean ここで示す EJB は、コール元に挨拶文を返す `helloWorld()` という 1 つのメソッドを実装しています。ローカル・ホーム・インタフェースおよびローカル・インタフェースの EJB コードも次に示します。

```
package myEjb;

import javax.ejb.*;

public class HelloBean implements SessionBean
{
  public String helloWorld ()   {
    return "Hello from myEjb.HelloBean";
  }

  public void ejbCreate () throws CreateException {}
  public void ejbRemove () {}
  public void setSessionContext (SessionContext ctx) {}
  public void ejbActivate () {}
  public void ejbPassivate () {}
}
```

EJB インタフェース・コード : ローカル・ホーム・インタフェースおよびローカル・インタフェース 次にローカル・ホーム・インタフェースのコードを示します。

```
package myEjb;

import javax.ejb.EJBLocalHome;
import javax.ejb.CreateException;

public interface HelloLocalHome extends EJBLocalHome
{
  public HelloLocal create () throws CreateException;
}
```

次にローカル・インタフェースのコードを示します。

```
package myEjb;

import javax.ejb.EJBLocalObject;

public interface HelloLocal extends EJBLocalObject
{
  public String helloWorld ();
}
```

ローカル・ルックアップの構成およびデプロイ

この項では、サーブレットと EJB のローカル・ルックアップのサンプル・アプリケーションをデプロイおよび構成する手順を説明します。ディスクリプタ・ファイルで、特に太字の部分に注意してください。このアプリケーションをデプロイするには、次のものが含まれている EAR ファイルが必要です。

- サーブレット・コードおよび `web.xml` Web ディスクリプタが含まれている WAR（Web アーカイブ）ファイル
- EJB コードおよび `ejb-jar.xml` EJB ディスクリプタが含まれている EJB の JAR アーカイブ・ファイル
- アプリケーション・レベルのディスクリプタ `application.xml`

OC4J でのデプロイの概要は、第 5 章「デプロイおよび構成の概要」を参照してください。詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

Web ディスクリプタおよびアーカイブ 次のように、標準の `web.xml` Web ディスクリプタを作成します。`<ejb-local-ref>` 要素と、ローカル・インタフェースを使用するための `<local-home>` および `<local>` サブ要素に注意してください。

```
<?xml version="1.0"?>
<!DOCTYPE WEB-APP PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <display-name>HelloServlet</display-name>
  <description> HelloServlet </description>
  <servlet>
    <servlet-name>ServletCallingEjb</servlet-name>
    <servlet-class>myServlet.HelloServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>ServletCallingEjb</servlet-name>
    <url-pattern>/DoubleHello</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file> index.html </welcome-file>
  </welcome-file-list>
  <ejb-local-ref>
    <ejb-ref-name>ejb/HelloBean</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home>myEjb.HelloLocalHome</local-home>
    <local>myEjb.HelloLocal</local>
  </ejb-local-ref>
</web-app>
```

次に、Web アプリケーションのデプロイ用の標準 J2EE ディレクトリ構造を作成し、その中に、`web.xml` Web デプロイメント・ディスクリプタとコンパイル済のサーブレット・クラス・ファイルを移動します。ディレクトリ構造を作成し、移入した後、ファイルを入れる WAR ファイル（たとえば `myapp-web.war`）を作成します。次に WAR ファイルの内容を示します。

```
META-INF/
META-INF/MANIFEST.MF
WEB-INF/
WEB-INF/classes/
WEB-INF/classes/myServlet/
WEB-INF/classes/myServlet/HelloServlet.class
WEB-INF/web.xml
```

（JAR ユーティリティにより、自動的に `MANIFEST.MF` ファイルが作成されます。）

EJB ディスクリプタおよびアーカイブ 次のように、標準の ejb-jar.xml EJB ディスクリプタを作成します。前述の web.xml ファイルの <ejb-ref-name> の値は、この <ejb-name> の値に対応しています。この例では同じ名前を使用しています。このようにするとわかりやすいですが、必須ではありません。Web 層は JNDI 名から独立しており、任意の参照名を指定できます。

また、ローカル・インタフェースを使用するための <local-home> および <local> 要素に注意してください。これらは、web.xml ファイルの場合と同じエントリである必要があります。

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
  <enterprise-beans>
    <session>
      <description>Hello Bean</description>
      <ejb-name>ejb/HelloBean</ejb-name>
      <local-home>myEjb.HelloLocalHome</local-home>
      <local>myEjb.HelloLocal</local>
      <ejb-class>myEjb.HelloBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
  </assembly-descriptor>
</ejb-jar>
```

EJB コンポーネントを保持するための JAR ファイル（たとえば myapp-ejb.jar）を標準の J2EE 構造で作成します。次に JAR ファイルの内容を示します。

```
META-INF/
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
myEjb/
myEjb/HelloBean.class
myEjb/HelloLocalHome.class
myEjb/HelloLocal.class
```

（JAR ユーティリティにより、自動的に MANIFEST.MF ファイルが作成されます。）

アプリケーション・レベルのディスクリプタ アプリケーションをデプロイするには、標準のアプリケーション・デプロイメント・ディスクリプタ（application.xml）を作成します。このファイルには、アプリケーション内のモジュールが記述されています。

```
<?xml version="1.0"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application
 1.3//EN" "http://java.sun.com/dtd/application_1_3.dtd">

<application>
  <display-name>Servlet_calling_ejb_example</display-name>
  <module>
    <web>
      <web-uri>myapp-web.war</web-uri>
      <context-root>/myapp</context-root>
    </web>
  </module>
  <module>
    <ejb>myapp-ejb.jar</ejb>
  </module>
</application>
```

<context-root> 要素は必須ですが、OC4J スタンドアロン環境では無視されます。コンテキスト・パスは、次の「デプロイの構成」に示すように、実際は http-web-site.xml の該当する <web-app> 要素の root 属性で指定します。一貫性を保ち、混乱を避けるため、<context-root> 要素の設定と <web-app> の root 属性の設定は同じにしてください。

最後に、アプリケーション・コンポーネントを保持するための EAR ファイル（たとえば myapp.ear）を標準の J2EE 構造で作成します。次に EAR ファイルの内容を示します。

```
META-INF/
META-INF/MANIFEST.MF
META-INF/application.xml
myapp-ejb.jar
myapp-web.war
```

（JAR ユーティリティにより、自動的に MANIFEST.MF ファイルが作成されます。）

デプロイの構成 アプリケーションをデプロイするために、次のエントリが OC4J 構成ファイルのディレクトリ内の server.xml ファイルに追加され、適切なパス情報が指定されます。

```
<application
   name="myapp"
   path="your_path/lib/myapp.ear"
/>
```

アプリケーションのデプロイに admin.jar の -deploy オプションを使用すると、このエントリは自動的に作成されます。（詳細は、5-26 ページの「admin.jar を使用した EAR ファイルのデプロイ」を参照してください。）

次に、Web モジュールを Web サイトにバインドします。OC4J 構成ファイル・ディレクトリの Web サイトの XML ファイル（OC4J スタンドアロンでは通常 http-web-site.xml）には、次のエントリが必要です。

```
<web-app
   application="myapp"
   name="myapp-web"
   root="/myapp"
/>
```

アプリケーションのデプロイ後に admin.jar -bindWebApp オプションを使用すると、このエントリは自動的に作成されます。（詳細は、5-27 ページの「admin.jar を使用した Web アプリケーションのバインド」を参照してください。）

サーブレットと EJB のアプリケーションの起動

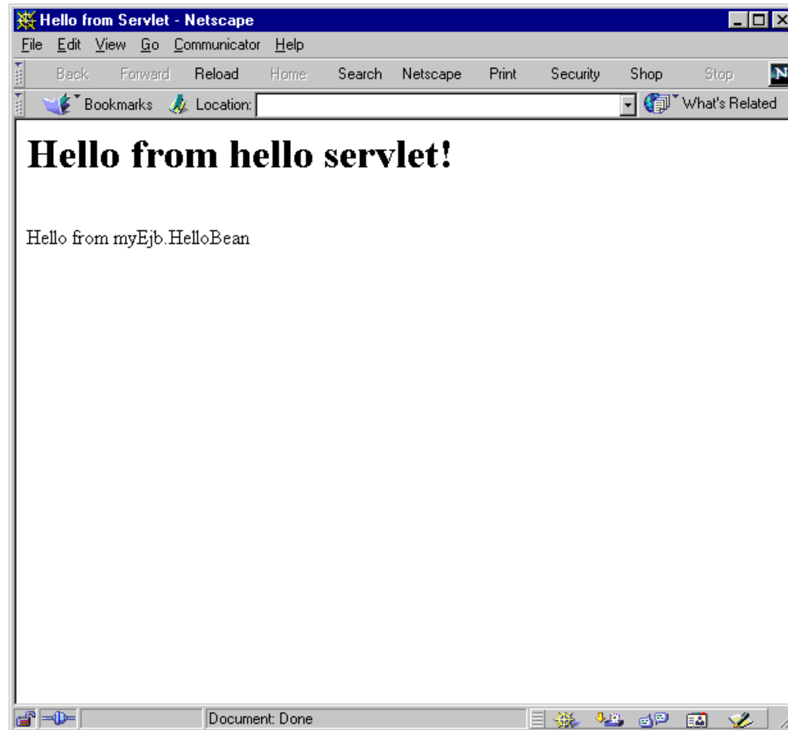
この例の構成の場合、次のようにしてサーブレットを起動できます。ホストを myhost とします。デフォルトで、OC4J スタンドアロンはポート 8888 を使用します。

```
http://myhost:8888/myapp/DoubleHello
```

コンテキスト・パス myapp は、関連する Web サイトの XML ファイル内の <web-app> 要素の root 設定によって決定されます。サーブレット・パス DoubleHello は、関連する web.xml ファイル内の <url-pattern> 要素によって決定されます。

次の図 4-3 に、Web ブラウザへのアプリケーション出力を示します。サーブレットからの出力は一番上に出力され（H1 の書式で）、EJB からの出力はその下にテキスト・フォーマットで出力されます。

図 4-3 HelloServlet からの出力



同一アプリケーション内の EJB リモート・ルックアップ

この項では、前述の HelloServlet/HelloBean の例を使用して、サーブレットと EJB が同一アプリケーションの異なる層に存在する、同一アプリケーション内のリモート・ルックアップの例を示します。ここでは、orion-application.xml ファイルの remote フラグの使用方法（EJB のデプロイ先と検索先を決定する）、およびコードとディスクリプタ・ファイルで必要な変更について説明します。

この例では、前述のローカル・ルックアップの例と同様に、JNDI コンテキストにデフォルトの ApplicationInitialContextFactory が使用されています。4-19 ページの「アプリケーション外部の EJB リモート・ルックアップ」の例に示すように、かわりに RMIInitialContextFactory を使用することも可能です。

リモート・フラグの使用

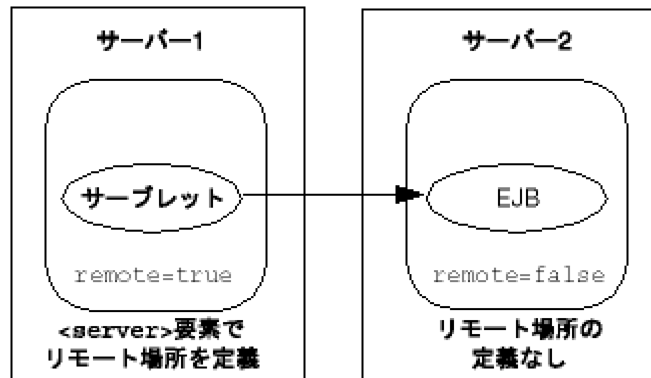
OC4J では、リモート EJB ルックアップを同一アプリケーションの異なる層で実行する場合（同一アプリケーションが両方の層でデプロイされている）、EJB の remote フラグを各層で正しく設定する必要があります。フラグが true に設定されている場合、Bean は、ローカル・サーバーで使用されている EJB サービスではなく、リモート・サーバーでルックアップされます。

remote フラグは、orion-application.xml ファイルにある <orion-application> 要素のサブ要素 <ejb-module> 内の属性です。デフォルトの設定は、remote="false" です。次に、true に設定する例を示します。

```
<orion-application ... >
   ...
   <ejb-module remote="true" ... />
   ...
</orion-application>
```

推奨される手順を図 4-4 に示し、次に説明します。

図 4-4 アプリケーション内のリモート・ルックアップの設定



1. アプリケーションの EAR ファイルを両サーバーにデプロイします。このとき、`remote` フラグの値は `false` です。`orion-application.xml` ファイルを提供する場合、`remote` フラグを明示的に `false` に設定しておくか、`remote` フラグ設定をまったく含めないことをお薦めします。フラグ設定を含めない場合、その値はデフォルトで `false` になります。`orion-application.xml` を提供しない場合は、OC4J によりこのファイルが自動的に生成されます。生成されたファイルの `remote` フラグは無効になっています。

2. サーブレット層であるサーバー 1 でアプリケーションの `orion-application.xml` ファイルを `remote="true"` に設定します。このように設定すると、サーブレットはサーバー 1 では EJB を検索しません。

3. EJB 層であるサーバー 2 で、アプリケーションの `orion-application.xml` ファイルが `remote="false"` に設定されていることを確認します。このように設定すると、サーブレットはサーバー 2 で EJB を検索します。

4. サーバー 1 の `rmi.xml` ファイル内の `<server>` 要素を使用し、サーブレットによる EJB の検索先として、リモート・ホストであるサーバー 2 を指定します。これには、ホスト名、ポートおよび認証用のユーザー名とパスワードが含まれます。

```
<rmi-server ... >
...
   <server host="remote_host" port="remote_port" username="user_name"
           password="password" />
...
</rmi-server>
```

複数のリモート・サーバーの `<server>` 要素が存在する場合、OC4J コンテナは、そのすべてでターゲット EJB を検索します。

注意： `rmi.xml` 構成では、リモート・ホストにはデフォルトの管理ユーザー名と、OC4J の `-install` オプションによってリモート・ホストで設定した管理パスワードを使用してください。このようにすることで、JAZN 構成に関する問題を回避できます。5-23 ページの「管理ユーザーおよびパスワードの設定」を参照してください。

---

5. デプロイおよび構成ファイルの変更が、各サーバーの OC4J に反映されていることを確認します。次のいずれかの方法で確認できます（サーバーごと）。

   - `check-for-updates` フラグが有効であるかどうか
   - `admin.jar` の `-updateConfig` オプションを使用
   - サーバーを再起動

   `check-for-updates` および `-updateConfig` の詳細は、2-3 ページの「開発用の主な OC4J のフラグ」を参照してください。

注意 : ここでは、開発時の OC4J 環境を前提として説明しています。Oracle Application Server 環境では、構成は常に Enterprise Manager または dcmctl コマンドライン・ユーティリティを使用して行う必要があります。デプロイ後に orion-application.xml を更新するのが不適切である場合は、一方の orion-application.xml ファイルでは remote="true" に設定し、もう一方の orion-application.xml ファイルでは remote="false" に設定した 2 種類の EAR ファイルを作成してデプロイする必要があります。

Oracle Application Server 環境でのサーブレットによる EJB コールの詳細は、『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』で説明されています。

同一アプリケーション内のリモート・ルックアップ用のサーブレットと EJB のアプリケーション・コード

この項では、同一アプリケーション内の EJB コンポーネントのリモート・ルックアップを使用したサーブレットと EJB のサンプル・コードを示します。これには、サーブレット・コード、EJB コードおよび EJB インタフェース・コードが含まれます。特に、太字の部分に注意してください。

サーブレット・コード : HelloServlet この項では、サーブレット・コードを示します。この例では、コードと構成は基本的にローカル・ルックアップの例と同じですが、次の 2 点が異なります。

- この例では、ローカル・インタフェースのかわりにリモート・インタフェースを使用します。
- この例では、javax.rmi.PortableRemoteObject.narrow() 静的メソッドを使用して、オブジェクトを確実に必要な型にキャストします。これはローカル・ルックアップの例では必要ありませんでしたが、リモート・ルックアップでは必須です。

この例でも、JNDI 初期コンテキスト・ファクトリにはデフォルトの ApplicationInitialContextFactory を使用し、ルックアップには java:comp 構文を使用し、web.xml ファイルで EJB 参照が検索されます。

この例では、サーブレット層の rmi.xml ファイルで、リモート・ルックアップのホスト、ポート、ユーザー名およびパスワードが指定されていることを前提としています。前項「リモート・フラグの使用」でこの点について説明しています。この前提があるため、サーブレット・コードで JNDI コンテキスト（URL、ユーザー名およびパスワード）を設定する必要はありません。

```
package myServlet;

// Step 1: Import the EJB package.
import myEjb.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;       // for JNDI
import javax.rmi.*;          // for RMI, including PortableRemoteObject
import javax.ejb.CreateException;
import java.rmi.RemoteException;

public class HelloServlet extends HttpServlet {

  public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
```

```
    out.println("<html><head><title>Hello from Servlet</title></head>");
    // Step 2: Print a message from the servlet.
    out.println("<body><h1>Hello from hello servlet!</h1></body>");

    // Step 3: Create an output string, with an error default.
    String s = "If you see this message, the ejb was not invoked properly!!";
    // Step 4: Use JNDI to look up the EJB home interface.
    try {
     InitialContext ic = new InitialContext();
     Object homeObject = ic.lookup("java:comp/env/ejb/HelloBean");
     HelloHome hh = (HelloHome)
        PortableRemoteObject.narrow(homeObject,HelloHome.class);

      // Step 5: Create the EJB local interface object.
      HelloRemote hr = (HelloRemote)
       PortableRemoteObject.narrow(hh.create(),HelloRemote.class);
      // Step 6: Invoke the helloWorld() method on the local object.
      s = hr.helloWorld();
     } catch (NamingException ne) {
         System.out.println("Could not locate the bean.");
     } catch (CreateException ce) {
         System.out.println("Could not create the bean.");
     } catch (RemoteException ce) {
         System.out.println("Error during execution of remote call.");
     } catch (Exception e) {
         // Unexpected exception; send back to client for now.
         throw new ServletException(e);
     }
    // Step 7: Print the message from the EJB.
    out.println("<br>" + s);
    out.println("</html>");
  }
}
```

EJB コード : HelloBean ステートフル Session Bean 同一アプリケーション内のリモート・ルックアップの EJB コードは、ローカル・ルックアップと非常に似ていますが、RemoteException を追加します。ホーム・インタフェースおよびリモート・インタフェースの EJB コードも次に示します。

```
package myEjb;

import javax.ejb.*;

public class HelloBean implements SessionBean
{
  public String helloWorld () {
    return "Hello from myEjb.HelloBean";
  }

  public void ejbCreate () throws CreateException {}
  public void ejbRemove () {}
  public void setSessionContext (SessionContext ctx) {}
  public void ejbActivate () {}
  public void ejbPassivate () {}
}
```

EJB インタフェース・コード : ホーム・インタフェースおよびリモート・インタフェース 次にホーム・インタフェースのコードを示します。EJBHome を拡張します（ローカル・インタフェースを使用する場合の EJBLocalHome ではない）。さらに、RemoteException も追加されています。

```
package myEjb;

import java.rmi.RemoteException;
import javax.ejb.EJBHome;
import javax.ejb.CreateException;

public interface HelloHome extends EJBHome
{
  public HelloRemote create () throws RemoteException, CreateException;
}
```

次にリモート・インタフェースのコードを示します。EJBObject を拡張します（ローカル・インタフェースを使用する場合の EJBLocalObject ではない）。前述のホーム・インタフェースの場合と同様、RemoteException の使用が追加されています。

```
package myEjb;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface HelloRemote extends EJBObject
{
  public String helloWorld () throws RemoteException;
}
```

同一アプリケーション内のリモート・ルックアップの構成

この項では、リモート・インタフェースで使用する web.xml および ejb-jar.xml エントリについて説明します。強調表示された箇所を、ローカル・インタフェースの例の該当する箇所と比較できます。

注意：

- 各ホストへのデプロイの際、web.xml および ejb-jar.xml ファイルは同じです。
- 各ホストの server.xml エントリは、4-13 ページの「デプロイの構成」で示したローカル・ルックアップの例の場合と同じです。アプリケーションのデプロイに admin.jar の -deploy オプションを使用すると、自動的にこの処理が行われます。サーブレット層の http-web-site.xml エントリはローカル・ルックアップの例の場合と同じですが、EJB 層では異なります。

また、4-14 ページの「リモート・フラグの使用」で示したように、remote フラグを各ホストの orion-application.xml で正しく設定する必要があります。

Web ディスクリプタ： この例の web.xml の内容は、次のとおりです。<ejb-ref> 要素と、リモート・インタフェースを使用するための <home> および <remote> サブ要素に注意してください。

```
<?xml version="1.0"?>
<!DOCTYPE WEB-APP PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <display-name>HelloServlet</display-name>
  <description> HelloServlet </description>
  <servlet>
    <servlet-name>ServletCallingEjb</servlet-name>
    <servlet-class>myServlet.HelloServlet</servlet-class>
  </servlet>
  <servlet-mapping>
```

```
    <servlet-name>ServletCallingEjb</servlet-name>
    <url-pattern>/DoubleHello</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file> index.html </welcome-file>
  </welcome-file-list>
  <ejb-ref>
    <ejb-ref-name>ejb/HelloBean</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>myEjb.HelloHome</home>
    <remote>myEjb.HelloRemote</remote>
  </ejb-ref>
</web-app>
```

EJB ディスクリプタ： この例での ejb-jar.xml の内容は次のとおりです。リモート・インタフェースを使用するための <home> および <remote> 要素に注意してください。

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
  <enterprise-beans>
    <session>
      <description>Hello Bean</description>
      <ejb-name>ejb/HelloBean</ejb-name>
      <home>myEjb.HelloHome</home>
      <remote>myEjb.HelloRemote</remote>
      <ejb-class>myEjb.HelloBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
  </assembly-descriptor>
</ejb-jar>
```

アプリケーション外部の EJB リモート・ルックアップ

この項では、前述の HelloServlet/HelloBean の例を使用して、異なるアプリケーション（異なる OC4J インスタンスにデプロイされている）へのリモート・ルックアップの例を示し、コードおよびディスクリプタ・ファイルで必要な変更を説明します。

デフォルトの JNDI 初期コンテキスト・ファクトリである ApplicationInitialContextFactory を使用するかわりに、この例では RMIInitialContextFactory を使用します。

前項 4-14 ページの「同一アプリケーション内の EJB リモート・ルックアップ」で説明した remote フラグは、ここでは必要ありません。

アプリケーション外部のリモート・ルックアップ用のサーブレットと EJB のアプリケーション・コード

この項では、アプリケーション外部のリモート・ルックアップを使用したサーブレットと EJB のサンプル・コードを示します。これには、サーブレット・コード、EJB コードおよび EJB インタフェース・コードが含まれます。特に、太字の部分に注意してください。

サーブレット・コード : HelloServlet この項では、サーブレット・コードを示します。この使用例では、URL、ユーザーおよびパスワードは、サーブレット・コードで指定する必要があります。（同一アプリケーション内のリモート・ルックアップの例では、この情報は rmi.xml ファイルで指定されていることが前提となっていました。）この場合、ルックアップ用の JNDI 環境の設定という手順がサーブレット・コードに追加されています。このコードで、次のものは、javax.naming.InitialContext クラスによって実装される、javax.naming.Context インタフェースの静的フィールドです。

- `INITIAL_CONTEXT_FACTORY` 設定は、使用する初期コンテキスト・ファクトリを指定します。この例では `RMIInitialContextFactory` です。
- `SECURITY_PRINCIPAL` 設定は、サービスのコール元を認証するプリンシパル（ユーザー名）の ID を指定します。
- `SECURITY_CREDENTIALS` 設定は、サービスのコール元を認証するプリンシパルのパスワードを指定します。
- `PROVIDER_URL` 設定は、ルックアップ用の URL またはカンマ区切りの URL のリストを指定します。ポート番号の後の情報は、`server.xml` ファイルで定義されているアプリケーション名に対応します。この例では、`myapp` です。

`RMIInitialContextFactory` が使用されている場合、接続先のリモート EJB コンポーネントの JNDI ルックアップに `java:comp` 構文が存在しないため、ルックアップでは、`ejb-jar.xml` ファイルで指定されている EJB 名を使用する必要があります。`web.xml` ファイルはアクセスされないため、そのファイル内の EJB 参照はルックアップでは無視されます。

```
package myServlet;

// Step 1: Import the EJB package.
import myEjb.*;
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;        // for JNDI
import javax.rmi.*;           // for RMI, including PortableRemoteObject
import javax.ejb.CreateException;
import java.rmi.RemoteException

public class HelloServlet extends HttpServlet {

  public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("<html><head><title>Hello from Servlet</title></head>");
    // Step 2: Print a message from the servlet.
    out.println("<body><h1>Hello from hello servlet!</h1></body>");

    //Step 2.5: Set up JNDI properties for remote call
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.evermind.server.rmi.RMIInitialContextFactory");
    env.put(Context.SECURITY_PRINCIPAL, "admin");
    env.put(Context.SECURITY_CREDENTIALS, "welcome");
    env.put(Context.PROVIDER_URL, "ormi://myhost:port/myapp");

    // Step 3: Create an output string, with an error default.
    String s = "If you see this message, the ejb was not invoked properly!!";
    // Step 4: Use JNDI to look up the EJB home interface.
    try {
      InitialContext ic = new InitialContext(env);
      Object homeObject = ic.lookup("ejb/HelloBean");
      HelloHome hh = (HelloHome)
        PortableRemoteObject.narrow(homeObject, HelloHome.class);

      // Step 5: Create the EJB remote interface.
      HelloRemote hr = (HelloRemote)
       PortableRemoteObject.narrow(hh.create(), HelloRemote.class);
       // Step 6: Invoke the helloWorld() method on the remote object.
       s = hr.helloWorld();
```

```
      } catch (NamingException ne) {
          System.out.println("Could not locate the bean.");
      } catch (CreateException ce) {
          System.out.println("Could not create the bean.");
      } catch (RemoteException ce) {
          System.out.println("Error during execution of remote call.");
      } catch (Exception e) {
          // Unexpected exception; send back to client for now.
          throw new ServletException(e);
      }
    // Step 7: Print the message from the EJB.
    out.println("<br>" + s);
    out.println("</html>");
  }
}
```

注意 :

- JNDI プロパティ設定では、リモート・ホストにはデフォルトの管理ユーザー名と、OC4J の `-install` オプションによってリモート・ホストで設定した管理パスワードを使用してください。このようにすることで、JAZN 構成に関する問題を回避できます。5-23 ページの「管理ユーザーおよびパスワードの設定」を参照してください。
- Oracle Application Server 環境の場合、OPMN の動的ポート割当てのため、ORMI の URL には `ormi://...` 構文ではなく、`opmn:ormi://...` 構文を使用してください。
- OC4J スタンドアロン・クラスタ・モードでは、`lookup:ormi://...` 構文を使用します。

EJB コード : HelloBean ステートフル Session Bean アプリケーション外部のリモート・ルックアップの EJB コードは、Bean コードとインタフェース・コードを含め、アプリケーション内のリモート・ルックアップと（`RemoteException` の使用方法を含め）同じです。

```
package myEjb;

import javax.ejb.*;

public class HelloBean implements SessionBean
{
  public String helloWorld () {
    return "Hello from myEjb.HelloBean";
  }

  public void ejbCreate () throws CreateException {}
  public void ejbRemove () {}
  public void setSessionContext (SessionContext ctx) {}
  public void ejbActivate () {}
  public void ejbPassivate () {}
}
```

EJB インタフェース・コード : ホーム・インタフェースおよびリモート・インタフェース 次にホーム・インタフェースのコードを示します。

```
package myEjb;

import java.rmi.RemoteException;
import javax.ejb.EJBHome;
import javax.ejb.CreateException;

public interface HelloHome extends EJBHome
{
```

```
  public HelloRemote create () throws RemoteException, CreateException;
}
```

次にリモート・インタフェースのコードを示します。

```
package myEjb;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface HelloRemote extends EJBObject
{
  public String helloWorld () throws RemoteException;
}
```

アプリケーション外部のリモート・ルックアップの構成およびデプロイ

この項では、リモート・ルックアップ固有の ejb-jar.xml エントリについて説明します。これらのエントリは、アプリケーション内のリモート・ルックアップで使用するエントリ同じです。強調表示された箇所を、他の例の該当する箇所と比較できます。

サーブレットは、JNDI 初期コンテキスト・ファクトリに RMIInitialContextFactory を使用するため、web.xml ファイルは関係がありません。

> **注意：** どのホストへのデプロイについても、ejb-jar.xml ファイルは同じです。
>
> ローカル・ホストの server.xml エントリは、4-13 ページの「デプロイの構成」で示したローカル・ルックアップの例の場合と同じです。アプリケーションのデプロイに admin.jar の -deploy オプションを使用すると、自動的にこの処理が行われます。リモート・ホストの server.xml ファイルは、リモート・アプリケーションにあわせて適切に構成されます。ローカル・ホストの http-web-site.xml エントリはローカル・ルックアップの例の場合と同じですが、リモート・ホストでは異なります。

EJB ディスクリプタおよびアーカイブ ejb-jar.xml の内容は次のとおりです。リモート・インタフェースを使用するための <home> および <remote> 要素に注意してください。

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
 1.12//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
  <enterprise-beans>
    <session>
      <description>Hello Bean</description>
      <ejb-name>ejb/HelloBean</ejb-name>
      <home>myEjb.HelloHome</home>
      <remote>myEjb.HelloRemote</remote>
      <ejb-class>myEjb.HelloBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
  </assembly-descriptor>
</ejb-jar>
```

アプリケーション外部のリモート・ルックアップのデプロイ時の注意 次の手順を実行します。

1. リモート EJB をデプロイするには、別の EAR ファイル内に置き、適切な OC4J サーバーにデプロイします。デプロイ先のサーバーは、サーブレット・コードの `PROVIDER_URL` に反映されます。
2. サーブレットのコールに対して、リモート・インタフェースおよびホーム・インタフェースが使用可能であることを確認します。処理を簡単にするため、次のいずれかの方法で EJB JAR ファイル全体を使用可能にできます。
   - WAR ファイルの `/WEB-INF/lib` ディレクトリに配置する。
   - 任意の場所に配置し、アプリケーションの `orion-application.xml` ファイルの `<library>` 要素を使用してこのファイルを指し示す。

5

デプロイおよび構成の概要

この章では、主に OC4J スタンドアロン環境における、サーブレット開発者向けの OC4J 構成、パッケージングおよびデプロイの概要を説明します。次の項が含まれます。

- OC4J のデプロイおよび構成の一般的な概要
- 構成ファイルの概要
- アプリケーションのパッケージング
- OC4J スタンドアロンへのデプロイ方法
- Oracle Application Server での OC4J のデプロイ

OC4J のデプロイおよび構成の一般的な概要

このマニュアルは開発者ガイドであり、主として、アプリケーション開発に便利な OC4J スタンドアロン環境の使用を目的としています。OC4J スタンドアロンは、Oracle Application Server 環境外で、単一の OC4J インスタンスにより構成されます。この章では主に、スタンドアロン環境での構成およびデプロイについて説明します。スタンドアロン環境では、開発しているシステム上にデプロイします。

アプリケーションがエンタープライズで使用できる段階に達したら、Oracle Application Server 環境にデプロイします。この章では、Oracle Application Server でのデプロイおよび構成の概要も説明します。また第 7 章「Enterprise Manager を使用した構成」では、追加情報を説明します。ただし、Oracle Application Server 環境における OC4J の使用の詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

次の項で、概要を説明します。

- 概要 : OC4J スタンドアロンと Oracle Application Server 環境
- OC4J のデプロイ方法の概要
- Oracle のデプロイ・ツールの使用とエキスパート・モードの使用

概要 : OC4J スタンドアロンと Oracle Application Server 環境

このマニュアルの、特にこの章で説明する OC4J 機能の多くは、開発段階において OC4J スタンドアロンでのみ使用されるものです。大規模な本番環境用のエンタープライズ管理機能が備わった Oracle Application Server では、サーバーの稼働に支障をきたす処理が行われないようにすることが非常に重要です。スタンドアロン環境での開発時にはこれを考慮する必要がないため、OC4J スタンドアロンで可能な処理、または必要な処理に対する制限はほとんどありません。

OC4J スタンドアロンでは、アプリケーションのデプロイ、構成および管理のための `admin.jar` コマンドライン・ユーティリティが提供されます。特に初期段階でのテストでは、手動でファイルをデプロイしたり、手動で構成ファイルを更新することも可能です。初期テスト用には、OC4J のデフォルトの Web アプリケーションがあり、これに対して個別にサーブレット・ファイル、JSP ページおよび依存クラスを使用することができます。

注意 : `admin.jar` の主要なコマンドの説明は、5-22 ページの「OC4J スタンドアロンへのデプロイ方法」を参照してください。

OC4J スタンドアロンで開発する際はまず、OC4J の `development` フラグを使用してサーブレット変更後の自動再コンパイルおよび再ロードをトリガーするなど、いくつかの点に注意が必要です。詳細は、2-2 ページの「開発用 OC4J スタンドアロン」を参照してください。

エンタープライズ本番環境では、OC4J は Oracle Application Server に組み込まれ、J2EE エンタープライズ・システムの管理が引き継がれます。Oracle Application Server は、クラスタリングされた複数の OC4J プロセスを監視し、Oracle Enterprise Manager 10*g* により管理されます。Enterprise Manager を使用すると、複数の Oracle Application Server インスタンスとホスト間で、OC4J プロセスを管理および構成できます。したがって、`admin.jar` ツールを使用したり、構成ファイルを手動で更新して、OC4J プロセスをローカルで管理することはできません。これは、Enterprise Manager による管理との間で競合が発生するためです。

表 5-1 は、OC4J のデプロイと構成の機能について、OC4J スタンドアロンと Oracle Application Server の OC4J とを比較し、まとめたものです。

表 5-1 スタンドアロンの OC4J と Oracle Application Server の OC4J: デプロイ

機能	OC4J スタンドアロン	Oracle Application Server の OC4J
デプロイ方法	`admin.jar` を使用するか、手動でファイルを配置する。	Enterprise Manager または `dcmctl` を使用する。
構成方法	`admin.jar` を使用するか、手動でファイルを更新する。	Enterprise Manager または `dcmctl` を使用する。手動ではファイルを更新しない。
デプロイのパッケージング	EAR ファイル、WAR ファイル、J2EE アプリケーション・ディレクトリ構造内の非アーカイブ・ファイルまたは Web アプリケーション・ディレクトリ構造内の非アーカイブ・ファイルを使用する。	EAR ファイルまたは WAR ファイルを使用する。
デフォルトの J2EE アプリケーションまたは J2EE アプリケーション・ラッパー	OC4J のデフォルトの J2EE アプリケーションに独立した WAR ファイルを含めることができる。単純な Web アプリケーションには、EAR ファイルの作成は不要。	独立した WAR ファイルのデプロイ時に、OC4J によって、WAR ファイルをラップする J2EE アプリケーションおよび EAR ファイルが自動的に作成される。
デフォルトの Web アプリケーション	OC4J のデフォルトの Web アプリケーションを使用すると、デフォルトのルート・ディレクトリの下にファイルを配置することで、サーブレットをデプロイできる。構成は不要。	該当なし。
アプリケーションまたは Web アプリケーションの自動再ロード	`application.xml`、`web.xml`、`/WEB-INF/classes` の下のサーブレット・コード、または `/WEB-INF/lib` の下の JAR ファイルを変更する。	該当なし。
自動再コンパイルおよび再ロード	`development="true"` を使用する（または JSP ページに対し、JSP の `main_mode` を `recompile` に設定する)。	該当なし。

表 5-2 は、Web サイト関連の OC4J の機能および具体的な手段について、OC4J スタンドアロンと Oracle Application Server の OC4J とを比較し、まとめたものです。

表 5-2 スタンドアロンの OC4J と Oracle Application Server の OC4J: Web サイト

機能	OC4J スタンドアロン	Oracle Application Server の OC4J
Web サーバー	OC4J	Oracle HTTP Server
protocol	HTTP/HTTPS	AJP/ セキュアな AJP
Web アプリケーションを起動するデフォルトのポート	8888	7777（Oracle HTTP Server および Oracle Application Server Web Cache 用）
Web サイトの XML ファイル	`http-web-site.xml`	`default-web-site.xml`

注意： Oracle Application Server では、Enterprise Manager または dcmctl のどちらかを使用してください。同じ OC4J インスタンスをターゲットとして両方を同時に使用したり、同じデプロイの異なる部分で両方を使用しないでください。

OC4J のデプロイ方法の概要

OC4J では、標準 J2EE アプリケーション構造およびそのデプロイ方法がサポートされます。Web モジュール、EJB モジュール、アプリケーション・クライアント・モジュールおよびリソース・アダプタ・モジュール（コネクタ・ファクトリに使用）を持つ完全な J2EE アプリケーションをデプロイするための、EAR ファイルの使用がサポートされます。各タイプのモジュールは、0（ゼロ）個以上含めることができます。また、独立した WAR ファイルを使用して、独立した Web アプリケーションをデプロイすることも可能です。（Web モジュールが含まれる完全なアプリケーションの場合、WAR ファイルは EAR ファイル内に含まれています。）これらの機能の詳細は、J2EE 仕様とサーブレット仕様を参照してください。次の Web サイトで入手可能です。

http://java.sun.com/j2ee/docs.html

http://jcp.org/aboutJava/communityprocess/first/jsr053/index.html

J2EE アプリケーションをデプロイする前に、次の手順を完了する必要があります。

1. 静的 HTML ファイル、サーブレット、JSP ページおよび EJB など、アプリケーションの全コンポーネントを作成します。
2. application.xml や web.xml などの J2EE のディスクリプタ、および必要に応じて orion-application.xml や orion-web.xml などの OC4J のディスクリプタを作成します。OC4J のディスクリプタを作成しない場合は、J2EE アプリケーションのデプロイ時に自動的に生成されます。デフォルト設定をそのまま使用する場合は、これで十分です。
3. J2EE アプリケーション構造に基づいて、アプリケーションのコンポーネントおよびディスクリプタをパッケージします。orion-application.xml を作成した場合は、application.xml とともにこれを配置します。orion-web.xml を作成した場合は、web.xml とともにこれを配置します。非アーカイブ・ファイルを該当するディレクトリ構造内にデプロイすることは可能ですが、アプリケーションを EAR または WAR ファイルにデプロイするのが一般的です。5-20 ページの「アプリケーションのパッケージング」を参照してください。

注意： Oracle Application Server では、Enterprise Manager を使用してデプロイおよび構成を行うと、適切な OC4J 構成ファイルが自動的に作成または更新されます。

アプリケーションのパッケージが完了した後は、デプロイ方法が複数あります。これについては、この章の後半で説明します。

OC4J スタンドアロン環境では、次の方法があります。

- EAR ファイルをデプロイする。5-26 ページの「EAR ファイルの OC4J スタンドアロンへのデプロイ」を参照してください。
- 完全な J2EE アプリケーションのファイルをアプリケーション・ディレクトリ構造に手動でデプロイする。5-30 ページの「OC4J スタンドアロンでの、J2EE アプリケーション構造へのファイルのデプロイ」を参照してください。
- 独立した WAR ファイルをデプロイする。5-31 ページの「OC4J スタンドアロンへの独立した WAR ファイルのデプロイ」を参照してください。

- 独立した Web アプリケーションのファイルを Web アプリケーション・ディレクトリ構造に手動でデプロイする。5-33 ページの「OC4J スタンドアロンでの Web アプリケーション・ディレクトリ構造へのファイルのデプロイ」を参照してください。これには、初期テスト時に、OC4J のデフォルト Web アプリケーションにサーブレットまたは JSP ページを容易にデプロイできるオプションがあります。(詳細は、5-24 ページの「OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーション」を参照してください。)

Oracle Application Server では、Enterprise Manager を使用して、アプリケーションをデプロイおよび構成します。5-38 ページの「Oracle Application Server における OC4J のデプロイおよび構成の概要」を参照してください。

注意： アプリケーションの開発、パッケージング、デプロイおよび構成には、Oracle JDeveloper などの IDE を使用することもできます。この概要は、2-18 ページの「Oracle JDeveloper によるサーブレット開発のサポート」を参照してください。

Oracle のデプロイ・ツールの使用とエキスパート・モードの使用

ここでは、OC4J のアプリケーションをデプロイおよび構成する操作の 2 つのモードについて説明します。1 つはサポート・クライアント・モードと呼ばれ、OC4J スタンドアロンまたは Oracle Application Server のいずれかで、提供されるツールを使用します。もう 1 つは、エキスパート・モードと呼ばれ、OC4J スタンドアロンでの開発およびテストのフェーズでのみ使用されます。エキスパート・モードでは、直接ファイルを操作します。EAR ファイル、WAR ファイルまたは非アーカイブ・ファイルを手動でシステム上に配置し、構成ディスクリプタを手動で更新します。表 5-3 に、これらのモードについてまとめます。

表 5-3 サポート・クライアント・モードおよびエキスパート・モードのツールまたは方法

モード	OC4J スタンドアロンのツールまたは方法	Oracle Application Server の OC4J のツールまたは方法
サポート・クライアント・モード	`admin.jar`	Enterprise Manager または `dcmctl`
エキスパート・モード	サーバー・ファイルの直接操作	該当なし

エキスパート・モードでは、OC4J および Oracle Application Server のツールの保護および制約の外での操作になります。

OC4J スタンドアロンでは、通常、`admin.jar` でデプロイし手動ではファイルを更新しないこと、または手動でファイルを更新し `admin.jar` ではデプロイしないことを前提としています。これらの使用方法を併用すると、予期せぬ結果を起こす可能性があります。

Oracle Application Server では、サーバー・ファイルを直接操作しないでください。Oracle Application Server の Distributed Configuration Management(DCM)サブシステムによって、構成情報のリポジトリが管理されています。このリポジトリ(ファイル・システムの構成ファイルではありません)には、実際の構成設定が含まれています。Enterprise Manager と `dcmctl` コマンドライン・ツールは DCM と同調して動作するため、これらのツールのいずれかを使用したときに、構成リポジトリが適切に更新されます。

構成ファイルの概要

OC4J および OC4J アプリケーションを構成する構成ファイルには、複数のカテゴリとレベルがあります(OC4J 固有と J2EE 標準、およびグローバル・レベルとアプリケーション・レベル)。

OC4J スタンドアロン環境でのアプリケーションの開発およびテスト時には、これらの構成ファイルを手動で、または OC4J より提供される `admin.jar` ユーティリティを使用して操作することができます。たとえば、`admin.jar -deploy` コマンドを使用すると、`server.xml` が自動的に更新されて、指定した J2EE アプリケーションが OC4J に追加されます。`admin.jar -bindWebApp` コマンドを使用すると、指定した Web サイトの XML ファイルが自動的に更新されて、指定した Web モジュールが Web サイトにバインドされます。

重要： Oracle Application Server 環境では、ほぼすべての構成が Oracle Enterprise Manager 10*g* を使用して行えます。ここで説明する構成ファイルを直接操作しないでください。直接操作すると、エンタープライズ管理が損なわれ、望ましくない結果を招く原因になります。

主要な `admin.jar` コマンドについては、この章の後半の該当箇所で説明します。`admin.jar` ユーティリティの詳細は、『Oracle Application Server Containers for J2EE スタンドアロン・ユーザーズ・ガイド』を参照してください。

次の項では、OC4J およびアプリケーション構成ファイルについて説明します。

- OC4J および J2EE 構成ファイルの概要
- OC4J のトップレベルのサーバー構成ファイル : server.xml
- OC4J および J2EE アプリケーション・ディスクリプタ
- OC4J および J2EE の Web ディスクリプタ
- OC4J Web サイト・ディスクリプタ
- 例 : Web サイト・ディスクリプタへのマッピングと Web サイト・ディスクリプタからのマッピング

追加情報は次を参照してください。

- Enterprise Manager を使用した、Oracle Application Server 環境での OC4J の構成については、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。
- Enterprise Manager を使用したサーブレットおよび Web モジュールの構成については、このマニュアルの後半の第 7 章「Enterprise Manager を使用した構成」を参照してください。
- OC4J スタンドアロンの構成については、『Oracle Application Server Containers for J2EE スタンドアロン・ユーザーズ・ガイド』を参照してください。このマニュアルは、OTN-J のサイトから OC4J のダウンロードで入手可能です。

OC4J および J2EE 構成ファイルの概要

OC4J および J2EE の構成ファイルは、次の 5 つのカテゴリに分類できます。

- サーバー構成（OC4J 固有）。全体におけるトップレベルの OC4J 構成ファイルと、セキュリティ、データ・ソース、RMI、JMS およびロード・バランシングに関するサーバー・レベルの構成ファイルが含まれます。
- グローバル構成（OC4J 固有）。グローバル・アプリケーション・ディスクリプタ、グローバル Web ディスクリプタ、およびリソース・アダプタ用のグローバル・ディスクリプタが含まれます。
- Web サイト構成（OC4J 固有）。
- J2EE アプリケーション・レベルの構成。標準 J2EE アプリケーション、Web、EJB、アプリケーション・クライアントおよびリソース・アダプタ（コネクタ・ファクトリ）・ディスクリプタが含まれます。
- OC4J 固有のアプリケーション・レベルの構成。OC4J アプリケーション、Web、EJB、アプリケーション・クライアントおよびリソース・アダプタ（コネクタ・ファクトリ）・ディスクリプタが含まれます。

グローバル・ファイルは、OC4J サーバー上で実行中の処理すべてに影響し、アプリケーション・レベルおよび Web サイト・レベルの J2EE と OC4J 固有の機能にデフォルトを確立します。J2EE ファイルは、標準 J2EE 機能のデフォルトをオーバーライドして、追加の J2EE 標準設定を確立します。OC4J 固有のアプリケーション・レベルのファイルは、対応するグローバル・ファイルのデフォルトをオーバーライドし、対応する J2EE ファイルの設定をオーバーライドして、OC4J 固有の機能および設定を追加します。

サーブレット開発者に特に関係があるのは、トップレベルの OC4J サーバー構成ファイル（server.xml）、アプリケーション・ディスクリプタ（OC4J グローバル、J2EE アプリケーション・レベルおよび OC4J アプリケーション・レベル）、Web ディスクリプタ（OC4J グローバル、J2EE アプリケーション・レベルおよび OC4J アプリケーション・レベル）、および OC4J Web サイト・ディスクリプタです。これらの詳細を簡潔に説明します。

サーバー・レベル構成ファイルおよびグローバル構成ファイルは、デフォルトで j2ee/home/config ディレクトリにあります。OC4J は、ここで server.xml ファイルを検索します。server.xml ファイルでは、他のサーバー・レベル・ファイルおよびグローバル・ファイルの場所を指定します（デフォルトは同じディレクトリ）。OC4J スタンドアロンでは、構成ファイル・ディレクトリは java -config コマンドライン・オプションを使用して指定できます。

前述の 5 つの構成ファイルのカテゴリについて、次にまとめます。

サーバー、グローバルおよび Web サイトの構成ファイルのサマリー

server.xml ファイルは j2ee/home/config ディレクトリにあり、その他のサーバー構成ファイルおよび Web サイト構成ファイルの場所を指定します。デフォルトでは、これらのファイルも j2ee/home/config にあります。

サーバー構成　このカテゴリのファイルは OC4J 固有であり、OC4J サーバーの様々な側面を構成します。

- server.xml: このファイルは全 OC4J 構成の親ファイルであり、他のサーバー・レベル構成ファイルとグローバル構成ファイル、サーバー上の全 Web サイト、およびサーバー上の全アプリケーション（OC4J のデフォルトのアプリケーションを含む）を指し示す要素が含まれています。5-9 ページの「OC4J のトップレベルのサーバー構成ファイル : server.xml」を参照してください。
- jazn.xml: Oracle Application Server Java Authentication and Authorization Service（JAAS）Provider（OracleAS JAAS Provider）のこの構成ファイルでは、サーバー・レベルの jazn-data.xml ファイルへのディレクトリ・パスを指定します。
- jazn-data.xml: このサーバー・レベルの JAAS ファイルには、XML ベースのプロバイダのユーザー名およびロール情報が含まれます。アプリケーション・レベルのバージョンもあります。OracleAS JAAS Provider で、かわりに LDAP ベースのプロバイダが使用されている場合、このファイルは使用されません。
- data-sources.xml: このファイルには、データベース接続に関するデータ・ソース定義が含まれます。
- rmi.xml: このファイルには、Remote Method Invocation の構成機能が含まれます。
- jms.xml: このファイルには、Java Message Service の構成機能が含まれます。

OC4J グローバル構成　このカテゴリのファイルは OC4J 固有であり、デフォルトのアプリケーションなどの OC4J グローバル機能の設定を定義し、対応するアプリケーション・レベルの構成ファイルのデフォルト設定を決定します。

- application.xml: このファイルは OC4J 固有のグローバル・アプリケーション・ディスクリプタです。5-11 ページの「OC4J および J2EE アプリケーション・ディスクリプタ」を参照してください。

> **注意：**　OC4J グローバル・アプリケーション・ディスクリプタと、J2EE 標準のアプリケーション・レベルのディスクリプタの application.xml を混同しないでください。両方とも Web モジュールの定義に使用され、その他にも類似した機能がありますが、OC4J グローバル・アプリケーション・ディスクリプタは OC4J 固有の DTD を使用します。

- global-web-application.xml: このファイルは、OC4J 固有のグローバル Web ディスクリプタです。5-15 ページの「OC4J および J2EE の Web ディスクリプタ」を参照してください。

- `oc4j-connectors.xml`: このファイルは、リソース・アダプタ（コネクタ・ファクトリ）用の OC4J 固有のグローバル・ディスクリプタです。

Web サイト構成 サーバーによって認識される各 Web サイトには、構成用の Web サイトの XML ファイルが 1 つあります。Oracle Application Server では、Web サイトは 1 つのみです。OC4J スタンドアロンでも Web サイトは通常 1 つですが、通信が HTTP 経由および一部 HTTPS 経由で行われる共有アプリケーションなどでは、追加の Web サイトを使用することがあります。また、OC4J インスタンスが Web モジュールで使用されていない場合は、Web サイトが存在しない可能性もあります。5-18 ページの「OC4J Web サイト・ディスクリプタ」を参照してください。

- `default-web-site.xml`: このファイルは、Oracle Application Server 環境でのデフォルトの Web サイト・ディスクリプタです。
- `http-web-site.xml`: このファイルは、OC4J スタンドアロン環境でのデフォルトの Web サイト・ディスクリプタです。
- 追加の Web サイトの XML ファイル : 追加の Web サイト用に、任意の名前を付けた Web サイトの XML ファイルを別に作成します。

アプリケーション・レベルの構成ファイルのサマリー

J2EE 構成ファイルは標準アプリケーション構造に組み込まれます。OC4J のアプリケーション・レベルの構成ファイルを EAR または WAR ファイルに組み込んだ場合、この構成ファイルはアプリケーション構造にも組み込まれます。5-20 ページの「J2EE アプリケーション構造」を参照してください。J2EE アプリケーションのデプロイ時に、OC4J ファイルはデプロイメント・ディレクトリ内にコピー（組み込んだ場合）または生成（組み込まなかった場合）されます。デプロイメント・ディレクトリは、通常は、`j2ee/home/application-deployments` の下にあります。

J2EE アプリケーション・レベルの構成 このカテゴリのファイルはすべてアプリケーション・レベルであり、J2EE 仕様で定義されます。

- `application.xml`: このファイルは J2EE 標準のアプリケーション・ディスクリプタです。5-11 ページの「OC4J および J2EE アプリケーション・ディスクリプタ」を参照してください。

> **注意 :** J2EE アプリケーション・ディスクリプタと、OC4J グローバル・アプリケーション・ディスクリプタの `application.xml` を混同しないでください。両方とも Web モジュールの定義に使用され、その他にも類似した機能がありますが、DTD は個別に異なります。

- `web.xml`: このファイルは、J2EE 標準の Web ディスクリプタです。5-15 ページの「OC4J および J2EE の Web ディスクリプタ」を参照してください。
- `ejb-jar.xml`: このファイルは、J2EE 標準の EJB ディスクリプタです。
- `application-client.xml`: このファイルは、アプリケーション・クライアント用の J2EE 標準のディスクリプタです。
- `ra.xml`: このファイルは、リソース・アダプタ（コネクタ・ファクトリ）用の J2EE 標準のディスクリプタです。

OC4J アプリケーション・レベルの構成 このカテゴリのファイルは、アプリケーション・レベルの、OC4J 固有のファイルです。これらのファイルは、OC4J 固有の機能を構成して、対応する J2EE ディスクリプタからの標準機能を補完し、対応するサーバー・レベル・ディスクリプタまたはグローバル・ディスクリプタのデフォルト設定をオーバーライドします。

- `orion-application.xml`: このファイルは、OC4J 固有のアプリケーション・ディスクリプタです。5-11 ページの「OC4J および J2EE アプリケーション・ディスクリプタ」を参照してください。

- orion-web.xml: このファイルは、OC4J 固有の Web ディスクリプタです。5-15 ページの「OC4J および J2EE の Web ディスクリプタ」を参照してください。
- orion-ejb-jar.xml: このファイルは、OC4J 固有の EJB ディスクリプタです。
- jazn-data.xml: このアプリケーション・レベルの JAAS ファイルには、XML ベースのプロバイダのユーザー名およびロール情報が含まれます。サーバー・レベルのバージョンもあります。OracleAS JAAS Provider で、かわりに LDAP ベースのプロバイダが使用されている場合、このファイルは使用されません。
- orion-application-client.xml: このファイルは、アプリケーション・クライアント用の OC4J 固有のディスクリプタです。
- oc4j-ra.xml: このファイルは、リソース・アダプタ（コネクタ・ファクトリ）用の OC4J 固有のディスクリプタです。

追加情報 前述の各ディスクリプタの詳細は、次のマニュアルを参照してください。

- server.xml およびロード・バランシングの詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。
- OC4J のデータ・ソース、RMI、JMS およびリソース・アダプタと、関連するディスクリプタの詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』を参照してください。
- OC4J のセキュリティおよび JAAS と、関連するディスクリプタの詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。
- EJB 開発と、J2EE および OC4J の EJB ディスクリプタの詳細は、『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』を参照してください。

OC4J のトップレベルのサーバー構成ファイル : server.xml

OC4J の server.xml ファイルは、デフォルトで OC4J の j2ee/home/config ディレクトリにあり、OC4J サーバーと、すべての J2EE アプリケーション、Web アプリケーションおよびサーバーの Web サイトの、構成の開始点となります。特に、次の点に注意してください。

- トップレベルの <application-server> 要素の属性は、特に、EAR ファイルがデプロイされ解凍されるターゲット・ディレクトリ（application-directory 設定により決定）、および OC4J ディスクリプタがコピーまたは生成されるターゲット・ディレクトリ（deployment-directory 設定により決定）を指定します。

> **注意：** OC4J 10.1.2 実装の主要な <application-server> 属性は、OC4J の XML 構成ファイルの更新を自動的にチェックする check-for-updates です。2-3 ページの「開発用の主な OC4J のフラグ」を参照してください。

- <global-application> 要素は、OC4J グローバル・アプリケーション（デフォルトのアプリケーション）を指定します。このアプリケーションは、デフォルトでは他のすべてのアプリケーションの親です。name 属性では名前を定義し、path 属性では OC4J グローバル・アプリケーション・ディスクリプタとして使用するものを指定します。OC4J のデフォルトのアプリケーションの詳細は、5-24 ページの「OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーション」を参照してください。
- <global-web-app-config> 要素では、path 属性を使用して、OC4J グローバル Web アプリケーション・ディスクリプタとして使用するものを指定します。
- サーバーにより認識される各 Web サイトには <web-site> 要素があり、path 属性を使用して、対応する Web サイトの XML ファイルとして使用するものを指定します。OC4J には、すでに構成された要素があります。

- サーバーにデプロイされた各 J2EE アプリケーションには `<application>` 要素があります。`name` 属性では任意の J2EE アプリケーション名を指定します。`path` 属性は、EAR ファイルをデプロイし解凍する場所、またはすでに解凍されて（または手動で配置されて）アプリケーション・ファイルが存在している場所を示します。いずれの場合も、`name` 属性は通常、`.ear` 拡張子のない EAR ファイル名と同じです。前者の場合、`path` 属性は、EAR ファイル名までを含む、EAR ファイルへのフルパスを指定します。後者の場合は、抽出されたファイルのトップレベル・ディレクトリを指定します。
- `<rmi-config>` 要素では、`path` 属性を使用して、OC4J の RMI ディスクリプタとして使用するものを指定します。
- `<jms-config>` 要素では、`path` 属性を使用して、OC4J の JMS ディスクリプタとして使用するものを指定します。

注意： Oracle Application Server では、RMI ディスクリプタ（デフォルトでは `rmi.xml`）および JMS ディスクリプタ（デフォルトでは `jms.xml`）のポート設定がオーバーライドされます。

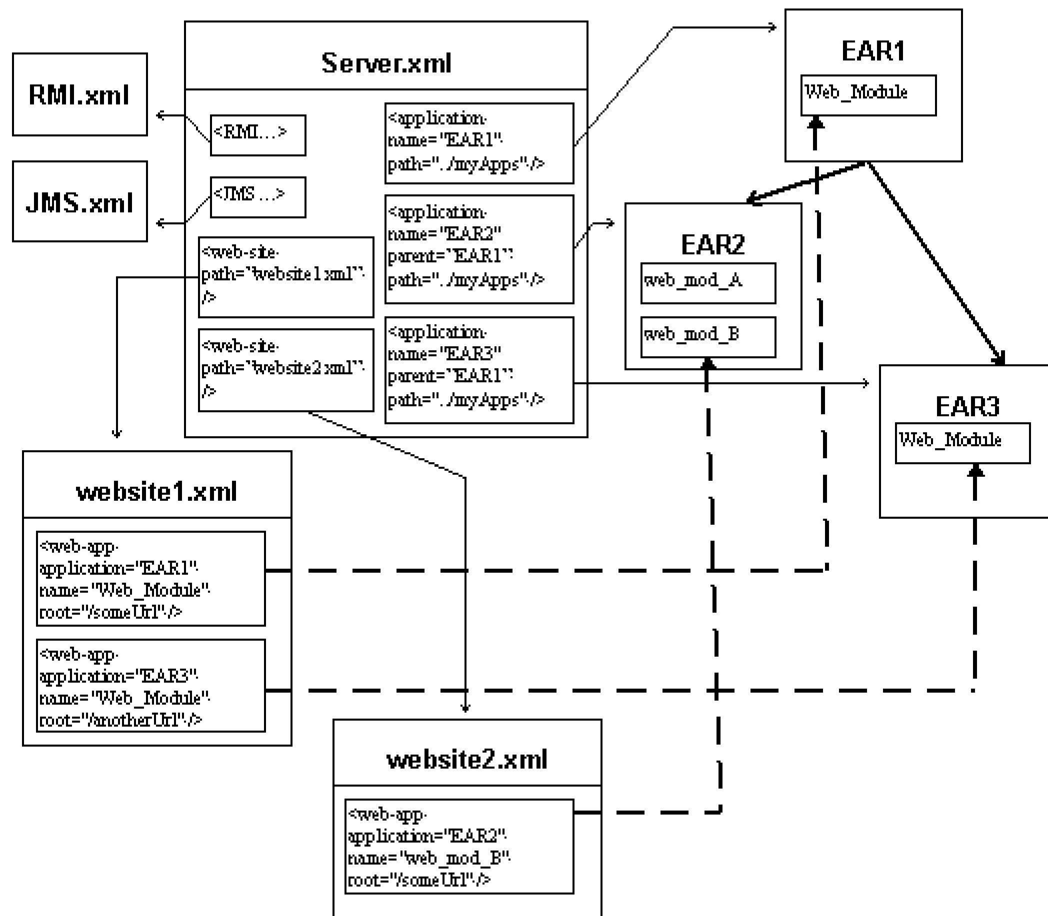
`server.xml` ファイルの詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。次に例を示します。

```
<?xml version="1.0"?>
<!DOCTYPE application-server PUBLIC "-//Evermind//DTD Orion
 Application-server//EN"
 "http://xmlns.oracle.com/ias/dtds/application-server.dtd">

<application-server application-directory="../applications"
                    deployment-directory="../application-deployments"
                    connector-directory="../connectors"
>
   <rmi-config path="./rmi.xml" />
   <jms-config path="./jms.xml" />
   <log>
      <file path="../log/server.log" />
   </log>
   <transaction-config timeout="250000" />
   <global-application name="default" path="application.xml" />
   <application name="petstore" path="../applications/petstore.ear" ... />
   <global-web-app-config path="global-web-application.xml" />
   <web-site default="true" path="./default-web-site.xml" />
   <web-site path="../myconfig/my-web-site.xml" />
   <cluster id="-1406559522" />
</application-server>
```

図 5-1 に、`server.xml` ファイル、Web サイトの XML ファイルを含むその他の XML ファイル、および J2EE の EAR ファイル間のマッピングを示します。この図では、`server.xml` の `<application>` 要素が、完全な形の各 EAR ファイルではなく、抽出された EAR ファイルのトップレベルのディレクトリを指し示していることに注意してください。

図 5-1 server.xml ファイルからのマッピング



OC4J および J2EE アプリケーション・ディスクリプタ

アプリケーション・ディスクリプタは、EJB や Web モジュールなどの J2EE アプリケーションのコンポーネントを指定し、さらにアプリケーションの追加構成も指定できます。

OC4J では、3 つのカテゴリのアプリケーション・ディスクリプタを使用します。次の項では、各カテゴリを説明し、互いの関係についてまとめています。

- 標準 J2EE アプリケーション・ディスクリプタ
- OC4J グローバル・アプリケーション・ディスクリプタ
- OC4J 固有のアプリケーション・ディスクリプタ
- 各アプリケーション・ディスクリプタの関係のサマリー

server.xml ファイルは、直接または間接的に、OC4J 上の各アプリケーションのアプリケーション・ディスクリプタを指し示します。通常の J2EE アプリケーションの場合、server.xml は EAR ファイル（または抽出された EAR のトップレベルのディレクトリ）を指し示すため、EAR ファイルに含まれる application.xml ファイルも指し示します。OC4J グローバル・アプリケーションの場合、server.xml ファイルは直接 OC4J グローバル・アプリケーション・ディスクリプタを指し示します。

OC4J のアプリケーション・ディスクリプタの詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

標準 J2EE アプリケーション・ディスクリプタ

J2EE 標準では、application.xml というアプリケーション・ディスクリプタの概要および DTD が定義されています。このディスクリプタは、J2EE アプリケーションの EAR ファイルの /META-INF ディレクトリに配置する必要があります。アプリケーション・ディスクリプタは、追加構成情報とともにアプリケーションに組み込まれているモジュールのマニフェストとして動作します。一部の開発環境では自動的に作成されます。詳細は、J2EE 仕様を参照してください。

次に、EJB モジュール、Web モジュールおよびクライアント・モジュールを使用したアプリケーションの例を示します。

```
<?xml version="1.0" ?>
<!DOCTYPE application (View Source for full doctype...)>
<application>
  <display-name>stateful, application:</display-name>
  <description>
    A sample J2EE application that uses a remote stateful session
    bean to call a local entity bean.
  </description>
  <module>
    <ejb>stateful-ejb.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>stateful-web.war</web-uri>
      <context-root>/stateful</context-root>
    </web>
  </module>
  <module>
    <java>stateful-client.jar</java>
  </module>
</application>
```

OC4J グローバル・アプリケーション・ディスクリプタ

OC4J 固有のグローバル・アプリケーション・ディスクリプタは、OC4J 固有の DTD である orion-application.dtd によって定義されます。これは、server.xml ファイルの <global-application> 要素で指定される、OC4J グローバル・アプリケーションのディスクリプタです。この要素は、グローバル・アプリケーション名をデフォルトで default に、グローバル・アプリケーション・ディスクリプタ名をデフォルトで application.xml（通常は server.xml と同じディレクトリに存在）に指定します。

OC4J グローバル・アプリケーションは、通常、デフォルトのアプリケーションとみなされ、OC4J インスタンス内のその他すべてのアプリケーションに対するデフォルトの親アプリケーションです。

> **注意：** 標準 J2EE アプリケーション・ディスクリプタおよび OC4J グローバル・アプリケーション・ディスクリプタは、異なる DTD で定義されますが、両方とも application.xml という名前です。この 2 つを混同しないでください。OC4J のデフォルトのアプリケーションに適用できる標準の application.xml ファイルはありません。

次に示すのは、OC4J のデフォルトのアプリケーション用の簡単なサンプル application.xml ファイルです。Web アプリケーションの名前に指定されている defaultWebApp は、通常、デフォルトの Web アプリケーションとして 1 つ以上の Web サイトにバインドされているものです。

OC4J のデフォルトの J2EE アプリケーションと、それに組み込まれているデフォルトの Web アプリケーションを混同しないでください。関連情報については、5-24 ページの「OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーション」を参照してください。

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE orion-application PUBLIC "-//Evermind//DTD J2EE Application runtime
 1.2//EN" "http://xmlns.oracle.com/ias/dtds/orion-application.dtd">

<!-- The global application config that is the parent of all the other
  applications in this server. -->

<orion-application autocreate-tables="true"
                   default-data-source="jdbc/OracleDS">
  <web-module id="defaultWebApp" path="../../home/default-web-app" />
  <connectors path="./oc4j-connectors.xml"/>

  <!-- Path to the libraries that are installed on this server.
  These will accesible for the servlets, EJBs etc -->
  <library path="../../home/lib" />
  <!-- Path to the taglib directory that is shared
        among different applications. -->
  <library path="../../home/jsp/lib/taglib" />

  <log>
    <file path="../log/global-application.log" />
  </log>

  <data-sources path="data-sources.xml" />
  <namespace-access>
    <read-access>
      <namespace-resource root="">
        <security-role-mapping>
          <group name="administrators" />
        </security-role-mapping>
      </namespace-resource>
    </read-access>
    <write-access>
      <namespace-resource root="">
        <security-role-mapping>
          <group name="administrators" />
        </security-role-mapping>
      </namespace-resource>
    </write-access>
  </namespace-access>
</orion-application>
```

OC4J 固有のアプリケーション・ディスクリプタ

標準アプリケーション・ディスクリプタである application.xml の他に、OC4J 固有のアプリケーション・レベルのアプリケーション・ディスクリプタである orion-application.xml があります。このディスクリプタは、OC4J グローバル・アプリケーション・ディスクリプタと同じ DTD によって定義されます。orion-application.xml ファイルは、application.xml ファイルと同じ、EAR ファイルの /META-INF ディレクトリに置くことができます。orion-application.xml ファイルで、OC4J 固有の構成を追加できます。

EAR ファイルへの orion-application.xml ファイルの組込みは、オプションです。組み込む場合は、デプロイ時に OC4J により、このファイルがデプロイメント・ディレクトリ（デフォルトでは j2ee/home/application-deployments ディレクトリの下）にコピーされます。組み込まない場合は、OC4J により、このファイルがデプロイメント・ディレクトリ内に生成されます。生成では、OC4J グローバル・アプリケーション・ディスクリプタ（デフォルトどおりに OC4J のデフォルトのアプリケーションが親アプリケーションとみなされます）と EAR ファイルの application.xml ファイルのデフォルト設定が使用されます。アプリケーション構造内で orion-application.xml が適合する位置については、5-20 ページの「J2EE アプリケーション構造」を参照してください。

注意： OC4J によりコピーされる際に orion-application.xml の内容が変更される場合がありますが、変更は透過的です。たとえば、デフォルト値を指定する属性設定が無視または削除される場合があります。

通常、orion-application.xml は、セキュリティ・ロール・マッピングなどの OC4J 固有の構成を定義するためにのみ使用します。また、OC4J でファイルが生成される場合は、J2EE の application.xml ファイルで指定されているモジュールを反映するために、<web-module> 要素が作成されることに注意してください。

次に、OC4J 固有の構成の例を示します。この例では、5-12 ページの「標準 J2EE アプリケーション・ディスクリプタ」にある、標準の application.xml ファイルの例での EJB、Web およびクライアント・モジュールと同じ定義を使用します。

```
<?xml version="1.0"?>
<!DOCTYPE orion-application PUBLIC "-//Evermind//DTD J2EE Application runtime
 1.2//EN" "http://xmlns.oracle.com/ias/dtds/orion-application.dtd">

<orion-application default-data-source="jdbc/OracleDS">
  <ejb-module remote="false" path="stateful-ejb.jar" />
  <web-module id="stateful-web" path="stateful-web.war" />
  <client-module path="stateful-client.jar" auto-start="false" />
  <persistence path="persistence" />
  <log>
    <file path="application.log" />
  </log>
  <namespace-access>
    <read-access>
      <namespace-resource root="">
        <security-role-mapping name="&lt;jndi-user-role&gt;">
          <group name="users" />
        </security-role-mapping>
      </namespace-resource>
    </read-access>
    <write-access>
      <namespace-resource root="">
        <security-role-mapping name="&lt;jndi-user-role&gt;">
          <group name="users" />
        </security-role-mapping>
      </namespace-resource>
    </write-access>
  </namespace-access>
</orion-application>
```

各アプリケーション・ディスクリプタの関係のサマリー

J2EE アプリケーション・ディスクリプタ、OC4J グローバル・アプリケーション・ディスクリプタ、および OC4J アプリケーション・レベルのアプリケーション・ディスクリプタ間の関係を次にまとめます。

- 典型的な J2EE アプリケーションでは、主要なアプリケーション・ディスクリプタは、標準 J2EE アプリケーション・ディスクリプタである application.xml です。このファイルは、J2EE アプリケーションのモジュールのマニフェストとして動作するため、J2EE アプリケーションの EAR ファイルの /META-INF ディレクトリに配置する必要があります。

- スタンドアロンの WAR ファイル（EAR ファイル内の WAR ファイルではありません）をデプロイする場合、ファイルを組み込むアプリケーションとして、OC4J のデフォルトのアプリケーションまたはグローバル・アプリケーションを使用できます。（詳細は、5-24 ページの「OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーション」を参照してください。）この場合、スタンドアロンの WAR ファイルには、関連付けられている J2EE 標準の application.xml ファイルがないため、OC4J グローバル・アプリケーション・ディスクリプタの application.xml（名前は同じですが、OC4J 固有の DTD で定義されています）が関連付けられます。

- オプションで、orion-application.xml ディスクリプタを組み込んで、OC4J 構成にセキュリティ・ロール・マッピングなどを追加できます。orion-application.xml ファイルでは、J2EE の application.xml ファイルで指定されているモジュール以外に追加モジュールも指定できます。また、application.xml で指定されているモジュールをオーバーライドすることもできます（ただし、これはお薦めできません）。orion-application.xml ファイルも、EAR ファイルの /META-INF ディレクトリに組み込まれます。このファイルを組み込まない場合は、デプロイ時に、OC4J グローバル・アプリケーション・ディスクリプタのデフォルトを使用してこのファイルが自動的に作成されます（デフォルトのアプリケーションがデプロイするアプリケーションの親であることを前提とします。デフォルトでは親となっています）。orion-application.xml ディスクリプタは、OC4J グローバル・アプリケーション・ディスクリプタと同じ DTD に基づいて定義されます。

OC4J および J2EE の Web ディスクリプタ

Web ディスクリプタは、静的ページ、サーブレットおよび JSP ページからなる J2EE の Web コンポーネントのセットを指定および構成します。各 Web コンポーネントは互いに組み合されて独立した Web アプリケーションを形成し、スタンドアロンの WAR ファイルにデプロイされます。ただし、通常は、これらのコンポーネントは J2EE アプリケーションの EAR ファイル内の WAR ファイルにデプロイされて、J2EE アプリケーション全体の一部を形成します。

OC4J では、3 つのカテゴリの Web ディスクリプタを使用します。次の項では、各カテゴリの説明と、カテゴリ間の関係についてまとめています。

- 標準 J2EE の Web ディスクリプタ
- OC4J グローバル Web アプリケーション・ディスクリプタ
- OC4J 固有の Web ディスクリプタ
- 各 Web ディスクリプタの関係のサマリー

標準 J2EE の Web ディスクリプタ

サーブレット仕様には、web.xml と呼ばれる Web ディスクリプタの概要および DTD が定義されています。このディスクリプタは、関連する WAR ファイルの /WEB-INF ディレクトリに配置する必要があります。web.xml ファイルは、WAR ファイルの Web コンポーネント、および Web コンポーネントによってコールされる EJB などの他のコンポーネントを指定および構成します。詳細は、サーブレット仕様を参照してください。

サンプル web.xml ファイルの中の、特にサーブレット、サーブレット・マッピングおよびローカル EJB ルックアップを指定している箇所を次に示します。

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
   <display-name>stateful, web-app:</display-name>
   <description>no description</description>
   <welcome-file-list>
      <welcome-file>index.html</welcome-file>
   </welcome-file-list>

   <ejb-local-ref>
      <ejb-ref-name>CartBean</ejb-ref-name>
      <ejb-ref-type>Session</ejb-ref-type>
      <local-home>cart.CartHome</local-home>
      <local>cart.Cart</local>
   </ejb-local-ref>

   <servlet>
      <servlet-name>cart</servlet-name>
      <servlet-class>cart.CartServlet</servlet-class>
```

```
        <init-param>
            <param-name>param1</param-name>
            <param-value>1</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>cart</servlet-name>
        <url-pattern>/cart</url-pattern>
    </servlet-mapping>
    <security-role>
        <role-name>users</role-name>
    </security-role>
</web-app>
```

OC4J グローバル Web アプリケーション・ディスクリプタ

server.xml ファイルでは、<global-web-app-config> 要素を使用して、OC4J グローバル Web アプリケーション・ディスクリプタが指定されます。通常、global-web-application.xml は、server.xml と同じディレクトリにあります。このディスクリプタでは、OC4J における Web アプリケーションのデフォルトの動作を定義します。

グローバル Web アプリケーション・ディスクリプタは、DTD の orion-web.dtd で定義されています。この DTD は、次項の「OC4J 固有の Web ディスクリプタ」で説明する、アプリケーション・レベルの OC4J 固有のディスクリプタ orion-web.xml の DTD と同一です。

orion-web.dtd は、web.xml 用の標準 DTD のスーパーセットです。orion-web.dtd には、トップレベル要素 <orion-web-app> のサブ要素 <web-app> があります。このサブ要素の仕様は、web.xml のトップレベル要素 <web-app> の仕様と同じです。<orion-web-app> には、この他にも OC4J 固有の機能を指定および構成するためのサブ要素が多くあります。

global-web-application.xml の <web-app> 要素内で指定するデフォルト設定はすべて、web.xml の <web-app> 設定を使用して、追加、またはオプションでオーバーライドできます。さらにその結果の設定は、orion-web.xml の <web-app> 設定を使用して、追加、またはオプションでオーバーライドできます。

> **注意：** global-web-application.xml や orion-web.xml 内で <web-app> 要素を使用しないでください。<web-app> エントリについては、通常、すべて web.xml を参照するため、このエントリが他にもあると混同しやすく、トラブルシューティングが困難になる場合があります。

global-web-application.xml の <web-app> 要素外で指定するデフォルト設定については、orion-web.xml のパラレル設定を使用して、追加、またはオプションでオーバーライドできます。

OC4J グローバル Web アプリケーション・ディスクリプタの要素と属性の詳細（DTD および階層表現を含む）は、6-2 ページの「global-web-application.xml および orion-web.xml の構成」を参照してください。

サンプル global-web-application.xml ファイル（抜粋）は、6-18 ページの「サンプルの global-web-application.xml の設定」を参照してください。

OC4J 固有の Web ディスクリプタ

標準 Web ディスクリプタ web.xml および OC4J グローバル Web アプリケーション・ディスクリプタ global-web-application.xml（デフォルトの動作を設定）の他に、OC4J 固有のアプリケーション・レベルの Web ディスクリプタ orion-web.xml があります。

orion-web.xml ディスクリプタは、DTD の orion-web.dtd で定義されています。この DTD は、前項の「OC4J グローバル Web アプリケーション・ディスクリプタ」で説明した、グローバル Web アプリケーション・ディスクリプタの DTD と同一です。

orion-web.xml ファイルは、web.xml ファイルと同じ、WAR ファイルの /WEB-INF ディレクトリに置くことができます。orion-web.xml を使用して、global-web-application.xml の任意のデフォルト設定に追加、またはオプションでオーバーライドできます。同様に、web.xml の設定も追加、またはオプションでオーバーライドできます。

WAR ファイル（EAR ファイル内）への orion-web.xml ファイルの組込みは、オプションです。組み込む場合は、デプロイ時に、OC4J により、このファイルがデプロイメント・ディレクトリ（デフォルトでは j2ee/home/application-deployments ディレクトリの下）にコピーされます。組み込まない場合は、OC4J により、orion-web.xml がデプロイメント・ディレクトリ内に生成されます。このファイルには、global-web-application.xml のデフォルト設定が使用されます。一部の web.xml 設定は、orion-web.xml の生成に影響を与えます。たとえば、web.xml 内の <resource-ref> エントリは、orion-web.xml 内の対応する <resource-ref-mapping> エントリに影響します。アプリケーション構造内で orion-web.xml が適合する位置については、5-20 ページの「J2EE アプリケーション構造」を参照してください。

注意： OC4J によりコピーされる際に orion-web.xml の内容が変更される場合がありますが、変更は透過的です。たとえば、デフォルト値を指定する属性設定が無視または削除される場合があります。

OC4J 固有の Web ディスクリプタの要素と属性の詳細（DTD および階層表現を含む）は、6-2 ページの「global-web-application.xml および orion-web.xml の構成」を参照してください。

次に、サンプルの orion-web.xml ファイルを示します。

```
<?xml version="1.0" ?>
<!DOCTYPE orion-web-app (View Source for full doctype...)>
<orion-web-app jsp-cache-directory="./persistence" temporary-directory="./temp"
              servlet-webdir="/servlet/" default-buffer-size="2048"
              development="false" directory-browsing="deny"
              file-modification-check-interval="1000" jsp-timeout="0 (never)">
  <ejb-ref-mapping name="EmployeeBean" />
  <security-role-mapping name="users">
     <group name="users" />
  </security-role-mapping>
  <!--
  <web-app>
      There are no <web-app> entries in this sample.
  </web-app>
  -->
</orion-web-app>
```

各 Web ディスクリプタの関係のサマリー

global-web-application.xml、web.xml および orion-web.xml 間の関係は、次のように考えることができます。

1. global-web-application.xml ファイルは、OC4J 内の Web アプリケーションのデフォルトを確立します。
2. web.xml ファイルは、global-web-application.xml の <web-app> 要素に定義されている設定すべてをオーバーレイします。この要素に定義されている Web コンポーネントおよびその他の設定を追加したりオーバーライドします。
3. orion-web.xml ファイルは、すべての設定をオーバーレイします。global-web-application.xml および web.xml の設定を追加したりオーバーライドします。

OC4J Web サイト・ディスクリプタ

OC4J の各 Web サイトの定義および構成には、Web サイトの XML ファイルを使用します。Web サイトの XML ファイルの主な機能は、次のとおりです。

- 指定された Web モジュールを Web サイトにバインドします。バインドする Web モジュール、Web モジュールが属する J2EE アプリケーション、および Web モジュールへのアクセスに使用する URL のコンテキスト・パス部分を識別します。
- ホスト名、ポート番号およびプロトコルなど、Web サイトの主要な設定を定義します。プロトコル設定は、Oracle Application Server 環境では AJP（Apache JServ Protocol）、スタンドアロン環境では HTTP を示す必要があります。

server.xml ファイルに含められた各サイトの <web-site> 要素によって、OC4J で認識される Web サイトの数が示されます。この要素は、それぞれ対応する Web サイトの XML ファイルのパスおよびファイル名を指定します。サンプルの server.xml エントリを次に示します。

```
...
<web-site path="./default-web-site.xml" />
<web-site path="mydir/my-web-site.xml" />
...
```

Oracle Application Server では、Web サイトは 1 つのみです。OC4J スタンドアロンでも Web サイトは通常 1 つですが、通信が HTTP 経由および一部 HTTPS 経由で行われる共有アプリケーションなどでは、追加の Web サイトを使用できます。（共有アプリケーションの詳細は、6-19 ページの「Web サイト XML ファイルの要素の説明」に記載されている <web-app> 要素の shared 属性の説明を参照してください。）

Web サイトの XML ファイルには、Web サイトにバインドする各 Web モジュールの <web-app> 要素が含まれます。各 <web-app> 要素には、少なくとも次が含まれます。

- application 属性 : Web モジュールが属する J2EE アプリケーションの名前（.ear 拡張子がない EAR ファイル名と同じ）を指定
- name 属性 : Web モジュールの名前（.war 拡張子がない WAR ファイル名と同じ）を指定
- root 属性 : この Web サイト上の、Web モジュールがバインドされる先のコンテキスト・パスを指定

この他に、デフォルトの Web アプリケーションの <default-web-app> 要素もあります。デフォルトの Web アプリケーションは、OC4J スタンドアロンでの開発時に役立ちます。これについては、5-24 ページの「OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーション」で説明します。Oracle Application Server では、デフォルトの Web アプリケーションは一部のシステム・レベルの機能で使用されますが、それ以外では利用されません。5-39 ページの「Oracle Application Server での OC4J のデフォルト Web アプリケーション」を参照してください。

重要 :

- root 設定は、組込み先の J2EE アプリケーションの application.xml ディスクリプタにある、この Web モジュールの <context-root> 要素の設定をオーバーライドします。<context-root> 要素は application.xml では必須ですが、OC4J では使用されません。後述の「例 : Web サイト・ディスクリプタへのマッピングと Web サイト・ディスクリプタからのマッピング」の項を参照してください。
- root 設定の「/」は、OC4J のデフォルトの Web アプリケーションをオーバーライドします。この設定（または NULL 設定。「/」に変換される）は、Web アプリケーションの Web サイトへのバインド時には、admin.jar ユーティリティによって使用が許可されません。

OC4J には、デフォルトで、Web サイトの XML ファイルが 1 つ構成されます。OC4J スタンドアロンでは `http-web-site.xml`、Oracle Application Server では `default-web-site.xml` です。

Web サイトの XML ファイルの要素と属性の詳細（DTD および階層表現を含む）は、6-19 ページの「Web サイトの XML ファイルの構成」を参照してください。

6-29 ページの「サンプルの default-web-site.xml ファイル」の例を参照してください。この例では、Oracle Application Server 環境におけるサンプルの `default-web-site.xml` ファイルを取り上げています。

例 : Web サイト・ディスクリプタへのマッピングと Web サイト・ディスクリプタからのマッピング

この例では、`server.xml` のエントリが Web サイト・ディスクリプタ（Web サイトの XML ファイル）を指し示す方法、および Web サイトの XML ファイルの `<web-app>` 要素が Web モジュールを指し示す方法を示しています。`<web-app>` 要素は、Web モジュールを Web サイトにバインドしています。Web モジュールは、組込み先の J2EE アプリケーションの `application.xml` ファイル（これも次に示します）に定義されています。

`server.xml` ファイルには、関連する（必要な Web モジュールを含む）J2EE アプリケーションの `<application>` 要素、および必要な Web サイトの Web サイト XML ファイルを指定する `<web-site>` 要素が含まれています。

```
<application-server ... >
   ...
   <application name="myear" path="../myapps/myear.ear" />
   ...
   <web-site path="my-web-site.xml" />
   ...
</application-server>
```

Web サイトの XML ファイルである `my-web-site.xml` によって、Web サイトが構成されます。このファイルには、Web モジュールを組み込む J2EE アプリケーション、Web モジュール自体の名前、および Web モジュールにアクセスするためのルート・コンテキスト・パスを指定した、`<web-app>` 要素が含まれています。

```
...
<web-site protocol="http" port="8888" display-name="My Web Site"
          host="[ALL]" log-request-info="false" secure="false">
  ...
  <web-app application="myear" name="mywebmod1" root="/someUrl"
           load-on-startup="false" max-inactivity-time="no shutdown"
           shared="false" />
  ...
</web-site>
```

ここで示した `<web-site>` および `<web-app>` 属性の詳細は、6-19 ページの「Web サイト XML ファイルの要素の説明」を参照してください。

注意 : EAR ファイルではなく、OC4J のデフォルトのアプリケーションにデプロイされる Web モジュール（WAR ファイル）の場合、`<web-app>` 要素の `application` 属性は、EAR ファイル名ではなく OC4J のデフォルトのアプリケーションの名前（デフォルトでは `default`）を示します。

デフォルトのアプリケーションに関する一般的な情報は、5-24 ページの「OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーション」を参照してください。

---

myear.ear 内の、J2EE の application.xml ファイルでは、Web モジュールが指定されています。

```
<application ... >
  ...
  <module>
    <web>
      <web-uri>mywebmod1.war</web-uri>
      <context-root>/someUrl</context-root>
    </web>
  </module>
  ...
</application>
```

注意：

- my-web-site.xml 内の <web-app> 要素の root 属性の値によって、application.xml 内の <context-root> 要素の値がオーバーライドされます。通常は、両方を同じ設定にしてください。
- OC4J のデフォルトのアプリケーションにデプロイする Web アプリケーションは、OC4J グローバル・アプリケーション・ディスクリプタで定義します。
- Oracle Application Server 環境では、default-web-site.xml ファイルによってデフォルトで設定される Web サイトは、OC4J へのアクセスに、Oracle HTTP Server および AJP（Apache JServ Protocol）を経由します。このプロトコル設定は ajp13 で、ポート設定は 0 です。ただし、このポート設定は、Oracle Process Manager and Notification Server（OPMN）によりオーバーライドされます。
- OC4J スタンドアロン環境では、http-web-site.xml ファイルによってデフォルトで設定される Web サイトは、OC4J リスナーに直接アクセスします。このプロトコル設定は http で、ポート設定は 8888 です。

アプリケーションのパッケージング

OC4J では、デプロイのための標準 J2EE アーカイブ・ファイルである、Web モジュール用の WAR ファイルと、全 J2EE アプリケーション用の EAR ファイルがサポートされます。次の項で、これらのファイルの構造について説明します。

- J2EE アプリケーション構造
- EAR ファイルおよび WAR ファイル構造

J2EE アプリケーション構造

この項では、標準 J2EE アプリケーション構造について説明します。この構造は、適切であれば開発構造として使用することができます。また、オプションの OC4J 固有のディスクリプタの相対的な位置も示します。デプロイに OC4J 固有のディスクリプタを含めない場合は、OC4J により、J2EE アプリケーションのデプロイ時にこのディスクリプタが生成されます。対応する OC4J グローバル・ディスクリプタおよび J2EE ディスクリプタの値がデフォルトとして使用されます。

```
J2EEAppName/

   META-INF/
      application.xml
      orion-application.xml (optional)

   EJBModuleName/
      (EJB classes, according to package)
```

```
    META-INF/
       ejb-jar.xml
       orion-ejb-jar.xml (optional)

 WebModuleName/
    (static files, such as index.html)
    (JSP pages)
    WEB-INF/
       web.xml
       orion-web.xml (optional)
       classes/
          (servlet classes, according to package)
       lib/
          (JAR files for dependency classes)

 ClientModuleName/
    (client classes, according to package)
    META-INF/
       application-client.xml
       orion-application-client.xml

 ResourceAdapterModuleName/
    META-INF/
       ra.xml
    (JAR files for required classes)
    (required static files or other files)
```

Web の部分は太字で示されています。この部分は、Web モジュールをデプロイする WAR ファイルの構造を示しています。トップレベルには静的ページ（index.html など）、JSP ページおよび /WEB-INF ディレクトリがあります。

注意 :

- この構造は、J2EE 仕様および関連する仕様で定義されています。J2EE 仕様は次の場所にあります。

http://java.sun.com/j2ee/docs.html

- application.xml および orion-application.xml の概要は、5-11 ページの「OC4J および J2EE アプリケーション・ディスクリプタ」を参照してください。
- web.xml および orion-web.xml の概要は、5-15 ページの「OC4J および J2EE の Web ディスクリプタ」を参照してください。
- ejb-jar.xml および orion-ejb-jar.xml の詳細は、『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』を参照してください。
- application-client.xml および orion-application-client.xml の詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

EAR ファイルおよび WAR ファイル構造

J2EE では、WAR ファイルは通常、EAR ファイル内に組み込まれます。前項の例では、EAR ファイル *J2EEAppName*.ear には、トップレベルの /META-INF ディレクトリとともに、Web モジュールの WAR ファイル、EJB モジュールの JAR ファイル、クライアント・アプリケーションの JAR ファイルおよびリソース・アダプタの RAR ファイル（各ファイルは、必要に応じてゼロ個以上）が含まれています。

```
META-INF/
   application.xml
   orion-application.xml (optional)
EJBModuleName.jar
WebModuleName.war
ClientModuleName.jar
ResourceAdapterModuleName.rar
```

サンプルの EAR および WAR ファイル

次の例は、単純な Web アプリケーションのアーカイブ・ファイルの構造を示しています。EAR ファイルには、単一のサーブレットを含む WAR ファイルが組み込まれています。

`utility.ear` の内容は次のとおりです。EJB、クライアント・アプリケーションまたはリソース・アダプタ・モジュールがある場合、関連する JAR ファイルは WAR ファイルと同じレベルになります。オプションで、`/META-INF` ディレクトリに `orion-application.xml` ファイルを組み込むことができます。この例では、組み込むかわりに、デプロイ時に OC4J によって生成されます。

```
META-INF/MANIFEST.MF
META-INF/application.xml
utility_web.war
```

`utility_web.war` の内容は次のとおりです。オプションで、`orion-web.xml` ファイルも `/WEB-INF` ディレクトリに組み込むことができます。この例では、組み込むかわりに、デプロイ時に OC4J によって生成されます。

```
META-INF/MANIFEST.MF
WEB-INF/classes/TestStatusServlet.class
WEB-INF/web.xml
index.html
```

注意：

- このマニュアルでは、ユーザーに J2EE の開発経験があり、EAR および WAR ファイルを作成する手段として、JAR ユーティリティを直接使用する方法、Oracle JDeveloper などの IDE を介する方法、または `ant` ユーティリティと `build.xml` ファイルを使用する方法のいずれかが使用可能であることを前提としています。`ant` の詳細は、次のサイトを参照してください。

http://ant.apache.org

- `MANIFEST.MF` ファイルは、JAR ユーティリティにより自動的に作成されます。

OC4J スタンドアロンへのデプロイ方法

この項では、予備的な考慮事項を検討した後、OC4J スタンドアロンへのいくつかのデプロイ方法について説明します。主なデプロイ方法は、アプリケーションを EAR ファイルにパッケージした後、`admin.jar` ユーティリティを使用することです。EAR ファイルには、オプションで、Web モジュールの WAR ファイル、EJB モジュールの JAR ファイル、クライアント・アプリケーションの JAR ファイル、およびリソース・アダプタの RAR ファイル（各ファイルはゼロ個以上）が組み込まれます。構造およびパッケージングの詳細は、5-20 ページの「アプリケーションのパッケージング」を参照してください。

OC4J のデプロイに対する EAR ファイルの使用および `admin.jar` ユーティリティの機能の詳細は、『Oracle Application Server Containers for J2EE スタンドアロン・ユーザーズ・ガイド』で広範囲にわたり説明されています。ここでは主な機能について説明します。

また、この項では、手動で J2EE アプリケーション構造を作成し移入する方法や、独立した WAR ファイルを OC4J のデフォルトのアプリケーションにデプロイする方法など、開発時に役立つ代替デプロイ方法についても説明します。

注意： これらの代替デプロイ方法では、手動でファイルを配置および更新するため、エキスパート・モードとみなされます。OC4J および Oracle Application Server のツールの保護および制約の外での操作となります。5-5 ページの「Oracle のデプロイ・ツールの使用とエキスパート・モードの使用」を参照してください。

この項には、次の項目が含まれます。

- 管理ユーザーおよびパスワードの設定
- OC4J スタンドアロンの起動および停止
- OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーション
- EAR ファイルの OC4J スタンドアロンへのデプロイ
- OC4J スタンドアロンでの、J2EE アプリケーション構造へのファイルのデプロイ
- OC4J スタンドアロンへの独立した WAR ファイルのデプロイ
- OC4J スタンドアロンでの Web アプリケーション・ディレクトリ構造へのファイルのデプロイ
- OC4J スタンドアロンにおけるアプリケーションのアンデプロイまたは再デプロイ

管理ユーザーおよびパスワードの設定

`admin.jar` ユーティリティを使用して OC4J スタンドアロンにアプリケーションをデプロイする前に、管理権限を持つユーザーを設定する必要があります。

`j2ee/home/config/jazn-data.xml` ファイルで、ユーザー・アカウントのセキュリティ権限が決定されます。デフォルトでは、管理権限を持つユーザー `admin` が存在します。サンプルの `jazn-data.xml` のエントリで、次のように指定されています。

```
<role>
   <name>administrators</name>
   <display-name>Realm Admin Role</display-name>
   <description>Administrative role for this realm.</description>
   <members>
      <member>
         <type>user</type>
         <name>admin</name>
      </member>
   </members>
</role>
```

デフォルトの管理ユーザー `admin` のデフォルトのパスワードは、`welcome` です。サンプルの `jazn-data.xml` のエントリで、次のように指定されています。

```
<users>
...
   <user>
      <name>admin</name>
      <display-name>OC4J Administrator</display-name>
      <description>OC4J Administrator</description>
      <credentials>!welcome</credentials>
   </user>
...
</users>
```

（ファイルは、指定したパスワードをわかりにくくするため、後で自動的に書き換えられます。）『Oracle Application Server Containers for J2EE セキュリティ・ガイド』で、`jazn-data.xml` ファイルの、特に `<credentials>` 要素に関する情報を参照してください。

重要： Oracle Application Server 環境では、admin.jar ユーティリティは使用できません。このユーティリティは OC4J スタンドアロン専用です。

注意： 廃止されている principals.xml ファイルをまだセキュリティに使用している場合は、管理アカウント・パスワードは OC4J の -install コマンドにより決定されます。

```
% java -jar oc4j.jar -install
```

(% はシステム・プロンプトで、j2ee/home はカレント・ディレクトリであるとします。）必要なパスワードの入力を促すプロンプトが表示されます。

OC4J スタンドアロンの起動および停止

この項では、OC4J スタンドアロンの起動および停止方法を簡潔にまとめます。% はシステム・プロンプトで、j2ee/home はカレント・ディレクトリであるとします。

次のコマンドを入力して、OC4J を起動します。

```
% java -jar oc4j.jar [options]
```

OC4J コマンドライン・オプションの詳細は、『Oracle Application Server Containers for J2EE スタンドアロン・ユーザーズ・ガイド』を参照してください。

次のコマンドを入力して、OC4J を停止します。

```
% java -jar admin.jar ormi://oc4j_host:oc4j_ormi_port
       adminuser adminpassword -shutdown
```

admin.jar の -shutdown コマンドに関して、次の点に注意してください。

- OC4J スタンドアロンでは、j2ee/home/config/rmi.xml ファイルから OC4J の ORMI ポート番号を取得できます。ファイル内の次のエントリにあります。

  ```
  <rmi-server port="23791" host="[ALL]">
  ```

- *adminuser* および *adminpassword* の詳細は、前項の「管理ユーザーおよびパスワードの設定」を参照してください。

OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーション

次の項では、OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーションの機能と構成について説明します。これらの機能は、後述する OC4J スタンドアロンのデプロイ方法の一部で使用されます。

- デフォルトのアプリケーションおよびデフォルトの Web アプリケーションの使用
- デフォルトのアプリケーションおよびデフォルトの Web アプリケーションの構成

注意： テスト段階のデプロイにデフォルトのアプリケーションおよびデフォルトの Web アプリケーションを使用できるのは、OC4J の便利な機能です。ただし、手動でアプリケーション・ファイルを配置し、場合によっては手動で構成ファイルを更新するため、エキスパート・モードでの操作となります。5-5 ページの「Oracle のデプロイ・ツールの使用とエキスパート・モードの使用」を参照してください。

デフォルトのアプリケーションおよびデフォルトの Web アプリケーションの使用

OC4J のインストールのデフォルト構成には、デフォルトのアプリケーション（グローバル・アプリケーションとも呼ばれます）が組み込まれています。デフォルトのアプリケーションは、OC4J の他のすべての J2EE アプリケーションのデフォルトの親です。

OC4J では、Web アプリケーションを親の J2EE アプリケーションに組み込む必要があります。通常、WAR ファイルは、親の J2EE アプリケーションを定義している EAR ファイルにデプロイします。独立した WAR ファイルをデプロイする場合は、かわりに OC4J のデフォルトのアプリケーションにデプロイできます。デフォルトでは、OC4J の `server.xml` ファイルには、デフォルトのアプリケーションを定義するグローバル・アプリケーション・ディスクリプタの場所と名前が指定されています。

通常の OC4J のインストールでは、デフォルトのアプリケーションにはデフォルトの Web アプリケーションが組み込まれています。デフォルトの Web アプリケーションの名前とルート・ディレクトリ・パスは、グローバル・アプリケーション・ディスクリプタに指定されています。デフォルトの Web アプリケーションは、OC4J スタンドアロンの `http-web-site.xml` ファイル（Oracle Application Server では `default-web-site.xml`）を使用して Web サイトにバインドされます。OC4J スタンドアロンでは、デフォルトの Web アプリケーションのデフォルトのコンテキスト・パスは「`/`」です。

また、OC4J スタンドアロンでは、デフォルトの Web アプリケーションのルート・ディレクトリは、デフォルトで `j2ee/home/default-web-app` です。デフォルトの Web アプリケーションにデプロイするには、JSP ページおよびクラス・ファイルを標準 Web アプリケーションのディレクトリ構造内のこのディレクトリの下に置きます。静的ページと JSP ページをトップレベルに、サーブレット・クラスを `j2ee/home/default-web-app/WEB-INF/classes` の下に、ライブラリ JAR ファイルを `j2ee/home/default-web-app/WEB-INF/lib` 内に置きます。5-33 ページの「デフォルトの Web アプリケーションでの Web アプリケーション・ディレクトリ構造の使用」も参照してください。

> **注意：** デフォルトの Web アプリケーションは、コンテキスト・パス「`/`」を使用して起動される他、リクエストされた URL のコンテキスト・パス・マッピングに失敗した場合にも起動されます。これは、リクエストされた URL に一致するコンテキスト・パスが、Web サイトの XML ファイル内にある、どの `<web-app>` 要素の `root` 属性にも含まれていない場合に発生します。

デフォルトのアプリケーションおよびデフォルトの Web アプリケーションの構成

この項では、OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーションのデフォルト構成について詳細に説明します。

デフォルトのアプリケーションの server.xml の構成 `server.xml` では、`<global-application>` 要素は OC4J のデフォルトのアプリケーションを指定します。`name` 属性は名前を指定し、`path` 属性は OC4J グローバル・アプリケーション・ディスクリプタとして使用するものを指定します。

```
<application-server ... >
   ...
   <global-application name="default" path="application.xml" />
   ...
</application-server>
```

デフォルトの Web アプリケーションの application.xml の構成 デフォルトのアプリケーション（グローバル・アプリケーション）用に指定されたディスクリプタ `application.xml` は、デフォルトのアプリケーションに組み込まれているデフォルトの Web アプリケーションの名前およびルート・ディレクトリ・パスを指定します。

```
<orion-application ... >
  ...
  <web-module id="defaultWebApp" path="../../home/default-web-app" />
  ...
</orion-application>
```

デフォルトの Web アプリケーションにデプロイするには、このディレクトリの下に、標準の Web アプリケーション構造に基づいてファイルを配置してください。

Web サイトの XML ファイル内でのデフォルトの Web アプリケーションのバインド デフォルトでは、デフォルトの Web アプリケーションは、OC4J スタンドアロンの `http-web-site.xml` ファイル（Oracle Application Server では、`default-web-site.xml`）の Web サイトにバインドされています。

ほとんどの OC4J の Web アプリケーションは、Web サイトの XML ファイルの `<web-site>` 要素の `<web-app>` サブ要素を使用して Web サイトにバインドされますが、デフォルトの Web アプリケーションは、かわりに `<web-site>` の `<default-web-app>` サブ要素を使用して構成されています。

OC4J スタンドアロンでは、デフォルトの Web アプリケーションのデフォルトのコンテキスト・パスは「/」で、`root` 属性は不要です。次に例を示します。

```
<web-site ... >
...
   <default-web-app application="default" name="defaultWebApp"
                    load-on-startup="true" shared="false" />
...
</web-site>
```

Web サイトの XML ファイルの要素および属性の詳細は、6-19 ページの「Web サイトの XML ファイルの構成」を参照してください。

注意： `<default-web-app>` 要素は、すべての Web サイトの XML ファイルに必要です。

EAR ファイルの OC4J スタンドアロンへのデプロイ

次の項では、`admin.jar` ユーティリティを使用して、OC4J スタンドアロンに EAR ファイルをデプロイする手順について説明します。

ここでは、コードを変更した場合、再パッケージおよび再デプロイすることを前提として説明します。

admin.jar を使用した EAR ファイルのデプロイ

アプリケーションを EAR ファイルにパッケージした後、OC4J の `admin.jar` ユーティリティを使用してデプロイできます。次の構文を使用します。

```
% java -jar admin.jar ormi://oc4j_host:oc4j_ormi_port
                      adminuser adminpassword
                      -deploy -file path/filename.ear
                      -deploymentName appname
```

このコマンドでは、OC4J との通信に RMI を使用します。次の点に注意してください。

- ORMI ポートの詳細は、5-24 ページの「OC4J スタンドアロンの起動および停止」を参照してください。
- *`adminuser`* および *`adminpassword`* の詳細は、5-23 ページの「管理ユーザーおよびパスワードの設定」を参照してください。
- `-file` には、EAR ファイルへのパスをファイル名を含めて指定してください。

- -deploymentName には、デプロイするアプリケーション名を指定してください。アプリケーション名は通常、.ear 拡張子のない EAR ファイル名と同じです。

> **注意：** 開発段階では、同じシステム上で開発と実行を行うものとして、ローカルにデプロイします。ただし、admin.jar ではリモートでデプロイすることも可能です。

デフォルトでは、デプロイの結果は次のようになります。

- EAR ファイルは j2ee/home/applications ディレクトリにコピーされます。このディレクトリは、server.xml ファイル内で、<application-server> 要素の application-directory 属性によってデフォルトに設定されています。
- EAR ファイルは j2ee/home/applications ディレクトリの下で解凍されます。
- OC4J 固有のディスクリプタ（最小限で、EAR ファイル内に、orion-application.xml および Web アプリケーションの orion-web.xml）が j2ee/home/application-deployments ディレクトリの下にコピーまたは生成されます。このディレクトリは、server.xml ファイル内で、<application-server> 要素の deployment-directory 属性によってデフォルトに設定されています。これらのディスクリプタは、EAR ファイルにある場合は EAR ファイルからコピーされますが、ない場合は OC4J により生成されます。
- <application> 要素が server.xml ファイルに追加されます。この要素は、admin.jar の -deploymentName の設定に基づいてアプリケーション名を指定し、EAR ファイルがデプロイされた場所へのパスを指定します。このパスは、デフォルトでは、j2ee/home/applications です。

5-28 ページの「サンプル・デプロイ」の例を参照してください。

> **注意：** ターゲット・ディレクトリは構成によって指定できます。-targetPath および -deploymentDirectory オプションなどの、admin.jar の追加情報は、『Oracle Application Server Containers for J2EE スタンドアロン・ユーザーズ・ガイド』を参照してください。

admin.jar を使用した Web アプリケーションのバインド

アプリケーションのデプロイ後、OC4J の admin.jar ユーティリティを使用して、関連する Web アプリケーションを Web サイトにバインドすることができます。

```
% java -jar admin.jar ormi://oc4j_host:oc4j_ormi_port
                      adminuser adminpassword
                      -bindWebApp appname webappname websitename contextpath
```

-bindWebApp コマンドでは、-deploy コマンドと同様に、OC4J との通信に RMI を使用します。次の点に注意してください。

- ORMI ポートの詳細は、5-24 ページの「OC4J スタンドアロンの起動および停止」を参照してください。
- *adminuser* および *adminpassword* の詳細は、5-23 ページの「管理ユーザーおよびパスワードの設定」を参照してください。
- *appname* は、デプロイ時の -deploymentName の設定に基づくアプリケーション名です。
- *webappname* は Web アプリケーションの名前です。この名前は、.war 拡張子のない WAR ファイル名です。
- *websitename* は、必要なサイトに対する Web サイトの XML ファイル名で示されます。ファイル名は、.xml 拡張子のない Web サイトの XML ファイル名（たとえば、OC4J スタンドアロンでは http-web-site）です。
- Web アプリケーションの起動に使用する URL のコンテキスト・パス部分を指定します。

注意： OC4J のデフォルトの Web アプリケーションをオーバーライドする「/」のコンテキスト・パス設定は、Web サイトへの Web アプリケーションのバインド時には、admin.jar ユーティリティによって使用できなくなります。NULL 設定も「/」に変換されるため使用できません。

このコマンドを実行すると、アプリケーション名、Web アプリケーション名およびコンテキスト・パスを示す <web-app> 要素が、指定した Web サイトの XML ファイルに追加されます。

次項の「サンプル・デプロイ」の例を参照してください。

サンプル・デプロイ

この例では、5-21 ページの「EAR ファイルおよび WAR ファイル構造」で説明した utility.ear ファイルのデプロイの結果とその Web アプリケーションのバインドについて示します。admin.jar コマンドを次に示します。% はシステム・プロンプトで、j2ee/home はカレント・ディレクトリ、EAR ファイルは j2ee/home/demo にあるものとします。

```
% java -jar admin.jar ormi://myhost:23791 admin welcome
  -deploy -file demo/utility.ear -deploymentName utility

% java -jar admin.jar ormi://myhost:23791 admin welcome
  -bindWebApp utility utility_web http-web-site /utilroot
```

次の点に注意してください。

- この例での OC4J のホスト名は myhost、ポート設定は j2ee/home/config/rmi.xml で設定されている 23791 です。
- この例では、管理アカウント名は admin、パスワードは welcome です。
- utility.ear 内の Web アプリケーション名は utility_web で、WAR ファイル名の utility_web.war に基づいています。
- -bindWebApp コマンドによって、j2ee/home/config/http-web-site.xml で定義された Web サイトに utility_web がバインドされます。ここでは、デフォルトと同様に、次のエントリが server.xml ファイルにあるものとしています。

  ```
  <web-site path="http-web-site.xml" />
  ```

- Web アプリケーションの起動に使用する URL のコンテキスト・パス部分は /utilroot です。

この -deploy コマンドを実行すると、server.xml 内に、トップレベルの <application-server> 要素のサブ要素として次のエントリが追加されます。

```
<application name="utility" path="../applications/utility.ear"
             auto-start="true" />
```

auto-start 属性は、OC4J の再起動のたびにこのアプリケーションを自動的に再起動するかどうかを指定します。

-bindWebApp コマンドを実行すると、http-web-site.xml 内に、トップレベルの <web-site> 要素のサブ要素として次のエントリが追加されます。

```
<web-app application="utility" name="utility_web" root="/utilroot"
 load-on-startup="false" max-inactivity-time="no shutdown" shared="false" />
```

（load-on-startup、max-inactivity-time および shared 属性の詳細は、6-19 ページの「Web サイト XML ファイルの要素の説明」を参照してください。）

注意：

- http-web-site.xml 内の <web-app> 要素の root 属性の値は、application.xml 内の <context-root> 要素の値をオーバーライドします。通常は、両方の設定を同じにしてください。
- server.xml および http-web-site.xml に追加されたエントリの情報は、バックグラウンド情報となります。admin.jar を使用するときは、どのような理由があっても、これらのファイルを手動で更新しないでください。

---

utility.ear のデプロイ後、主要なファイルのディレクトリ構造は次のようになります。ターゲット・ディレクトリはデフォルト設定であるとします。

```
j2ee/home/
   application-deployments/
      utility/
         orion-application.xml
         utility_web/
            orion-web.xml
   applications/
      utility.ear
      utility/
         utility_web.war
         META-INF/
            application.xml
         utility_web/
            index.html
            META-INF/
            WEB-INF/
               web.xml
               classes/
                  TestStatusServlet.class
```

server.xml および http-web-site.xml ファイルは、j2ee/home/config ディレクトリにあります。

orion-application.xml および orion-web.xml ファイルが EAR ファイル内に存在する場合は、これらのファイルが EAR ファイルからこの構造内のディレクトリにコピーされます。存在しない場合は、OC4J により、この構造内のディレクトリにファイルが生成されます。各ファイルの設定には、対応する OC4J グローバル・ディスクリプタおよび J2EE ディスクリプタのデフォルト設定が使用されます。

サンプルのデプロイのディスクリプタ

前項で説明したデプロイでは、次のディスクリプタが使用されます。サーブレット開発者にとって特に関係のある部分は、太字で強調されています。

application.xml ファイル　標準の application.xml ディスクリプタは、開発者が提供します。Oracle JDeveloper など一部の IDE 環境では、このファイルが作成されます。

```
<?xml version="1.0" ?>
<!DOCTYPE application (View Source for full doctype...)>
<application>
  <display-name>Web Services Demo</display-name>
  <module>
    <web>
      <web-uri>utility_web.war</web-uri>
      <context-root>/j2ee/utility</context-root>
    </web>
  </module>
</application>
```

この <context-root> 要素は、Web サイトの XML ファイルにある <web-app> 要素の root 属性によってオーバーライドされることに注意してください。

web.xml ファイル 標準の web.xml ディスクリプタは、開発者が提供します。

```
<?xml version="1.0" ?>
<!DOCTYPE web-app (View Source for full doctype...)>
<web-app>
  <display-name>Web Services Example</display-name>
  <description>A few examples of web service publication</description>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
  <servlet>
    <servlet-name>TestStatus</servlet-name>
    <servlet-class>TestStatusServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>TestStatus</servlet-name>
    <url-pattern>/TestStatusServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

orion-application.xml ファイル この例では、orion-application.xml ディスクリプタは EAR ファイルに含まれていないため、OC4J により生成されます。ここではファイルの大部分は省略されていますが、<web-module> 要素に application.xml ファイルのエントリが反映されていることに注意してください。

```
<?xml version="1.0" ?>
<!DOCTYPE orion-application (View Source for full doctype...)>
<orion-application deployment-version="10.1.2.0.0"
                   default-data-source="jdbc/OracleDS"
                   treat-zero-as-null="true" autocreate-tables="true"
                   autodelete-tables="false">
  ...
  <web-module id="utility_web" path="utility_web.war" />
  ...
</orion-application>
```

orion-web.xml ファイル この例では、orion-web.xml ディスクリプタは WAR ファイル（EAR ファイル内）に含まれていないため、OC4J により生成されます。この例に固有のエントリはないため、ここでは示されていません。

サンプルのアプリケーションの起動

前項で説明したサンプル・デプロイの場合、/utilroot が admin.jar の -bindWebApp コマンドでコンテキスト・パスとして指定され、/TestStatusServlet が web.xml でサーブレット・パスとして指定されています。このことから、アプリケーションの起動は、次のようになります。

http://*host*:*port*/utilroot/TestStatusServlet

OC4J スタンドアロンでの、J2EE アプリケーション構造へのファイルのデプロイ

EAR ファイルをデプロイするかわりに、手動でファイル構造をデプロイし、server.xml および Web サイトの XML ファイルを更新できます。これはエキスパート・モードです。（詳細は、5-5 ページの「Oracle のデプロイ・ツールの使用とエキスパート・モードの使用」を参照してください。）

再び、5-28 ページの「サンプル・デプロイ」を参照してください。ただし、j2ee/home/applications/utility ディレクトリを手動で作成し、その下にアプリケーション・ディレクトリ構造を手動で移入するものとします。

```
j2ee/home/applications/utility/
   META-INF/
      application.xml
   utility_web/
      index.html
      META-INF/
      WEB-INF/
         web.xml
         classes/
            TestStatusServlet.class
```

さらに、次のように server.xml ファイルを更新するものとします。

```
<application name="utility" path="../applications/utility"
             auto-start="true" />
```

そして、次のように http-web-site.xml ファイルを更新します。

```
<web-app application="utility" name="utility_web" root="/utilroot"
 load-on-startup="false" max-inactivity-time="no shutdown" shared="false" />
```

server.xml 内の <application> 要素の path 属性が、../applications/utility.ear ではなく ../applications/utility であることに注意してください。これは、EAR ファイルが存在せず、utility ディレクトリ下のディレクトリ構造であるためです。

OC4J スタンドアロンのデフォルトでは、server.xml を更新すると、OC4J によって変更が検出され、アプリケーションがデプロイされて、application-deployments ディレクトリの下に orion-web.xml ファイルおよび orion-application.xml ファイルが次のようにコピーまたは生成されます。

```
j2ee/home/
   application-deployments/
      utility/
         orion-application.xml
         utility_web/
            orion-web.xml
```

ただし、OC4J の更新チェックが無効化されている場合は、admin.jar -updateConfig オプションを使用して、OC4J に構成の更新を手動で通知する必要があります。更新チェックを有効化するには、server.xml の check-for-updates フラグを使用します。2-3 ページの「開発用の主な OC4J のフラグ」を参照してください。

OC4J スタンドアロンへの独立した WAR ファイルのデプロイ

5-26 ページの「EAR ファイルの OC4J スタンドアロンへのデプロイ」では、admin.jar ユーティリティを使用した EAR ファイル（Web モジュール用の WAR ファイルを含む）のデプロイについて説明しました。テスト時の便利な方法として、OC4J のデフォルトのアプリケーションを組込みアプリケーションとして使用し、独立した WAR ファイルを手動でデプロイすることも可能です。（Web アプリケーションは、OC4J では、常に親の J2EE アプリケーションの一部である必要があります。）これはエキスパート・モードです。5-5 ページの「Oracle のデプロイ・ツールの使用とエキスパート・モードの使用」を参照してください。

バックグラウンド情報は、5-24 ページの「OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーション」を参照してください。

次の手順で、独立した WAR ファイルを OC4J のデフォルトのアプリケーションにデプロイします。

1. WAR ファイルを目的のディレクトリに配置します。

2. OC4J グローバル application.xml ファイルに <web-module> 要素を追加し、Web アプリケーション名および WAR ファイルの場所を指定して更新します。
3. 対象となる Web サイトの XML ファイル（通常は、OC4J スタンドアロンの http-web-site.xml）に <web-app> 要素を追加し、Web アプリケーションを Web サイトにバインドして更新します。

次の例は、デフォルトのアプリケーションへの WAR ファイルのデプロイを示しています。

注意： Oracle Application Server では、すべてのアプリケーションが EAR ファイルにデプロイされます。Enterprise Manager で Application Server Control コンソールの「Web アプリケーションのデプロイ」ページを使用する場合、WAR ファイルが要求され、EAR ファイルが透過的に作成されて WAR ファイルが組み込まれます。

例 : OC4J のデフォルト・アプリケーションでの Web アプリケーション名

この例では、OC4J のデフォルトのアプリケーション内の Web アプリケーションである mywebmod1 に関する、server.xml、Web サイトの XML ファイルおよび OC4J グローバル application.xml ファイルのエントリを示します。次の点に注意してください。

- この例では、server.xml 内の <web-site> 要素の path 属性は、Web サイトの XML ファイルである http-web-site.xml のパスおよび名前を指定しています。
- server.xml 内の <global-application> 要素の name 属性は、OC4J のデフォルトのアプリケーションの名前を指定しており、http-web-site.xml 内の <web-app> 要素の application 属性に対応しています。
- server.xml 内の <global-application> 要素の path 属性は、OC4J グローバル application.xml ファイルを指し示しています。
- http-web-site.xml 内の <web-app> 要素の name 属性は、OC4J のデフォルトのアプリケーション内の Web アプリケーションである mywebmod1 を示しており、OC4J グローバル application.xml ファイル内の <web-module> 要素の id 属性に対応しています。通常、この 2 つの属性は、.war 拡張子のない WAR ファイル名に対応しています。
- グローバル application.xml ファイルでは、WAR ファイルの名前および場所の指定に、<web-module> 要素の path 属性を使用する必要があります。これは、WAR ファイルを組み込む EAR ファイルがないためです。

次のエントリは、server.xml ファイルにあります（変更はありません）。

```
<application-server ... >
   ...
   <global-application name="default" path="application.xml" />
   ...
   <web-site path="http-web-site.xml" />
   ...
</application-server>
```

太字で示されているエントリを、Web サイトの XML ファイル http-web-site.xml 内に挿入します。

```
<web-site protocol="http" port="8888" display-name="HTTP Web Site"
          host="[ALL]" log-request-info="false" secure="false">
  ...
  <web-app application="default" name="mywebmod1" root="/someUrl"
           load-on-startup="false" max-inactivity-time="no shutdown"
           shared="false" />
  ...
</web-site>
```

（ここで示した <web-site> および <web-app> 属性の詳細は、6-19 ページの「Web サイト XML ファイルの要素の説明」を参照してください。）

太字で示されているエントリを、OC4J グローバル application.xml ファイル内に挿入します。

```
<orion-application ... >
  ...
  <web-module id="mywebmod1" path="../myhome/mywebmod1.war" />
  ...
</orion-application>
```

注意： OC4J スタンドアロンのデフォルトでは、グローバル application.xml ファイルを編集すると、WAR ファイルが配置したディレクトリの下で自動的に解凍されます。orion-web.xml ファイルが組み込まれている場合は、WAR ファイルからデプロイメント・ディレクトリにコピーされます（デフォルトでは j2ee/home/application-deployments の下）。組み込まれていない場合は、orion-web.xml がデプロイメント・ディレクトリに生成されます。

ただし、構成変更の自動検出は、server.xml の check-for-updates フラグに依存します。これは、デフォルトでは true に設定されています。このフラグが無効化されている場合は、admin.jar の -updateConfig オプションを使用して、ワンタイム・チェックを 1 回トリガーできます。2-3 ページの「開発用の主な OC4J のフラグ」を参照してください。

OC4J スタンドアロンでの Web アプリケーション・ディレクトリ構造へのファイルのデプロイ

前項では、独立した WAR ファイルを OC4J のデフォルトのアプリケーションにデプロイする方法を説明しました。もう 1 つの方法として、WAR ファイルを使用するかわりに、J2EE の Web アプリケーションのディレクトリ構造を手動で設定することもできます。再び、OC4J のデフォルトのアプリケーションを使用します。ここでも、OC4J のデフォルトの Web アプリケーションを使用するのが最も簡単な方法ですが、オプションで、デフォルトのアプリケーションの下に新規の Web アプリケーションを定義することもできます。バックグラウンド情報は、5-24 ページの「OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーション」を参照してください。

これらの方法は、エキスパート・モードで行います。（詳細は、5-5 ページの「Oracle のデプロイ・ツールの使用とエキスパート・モードの使用」を参照してください。）次の項で、この 2 つの方法について説明します。

- デフォルトの Web アプリケーションでの Web アプリケーション・ディレクトリ構造の使用
- 代替 Web アプリケーションでの Web アプリケーション・ディレクトリ構造の使用

デフォルトの Web アプリケーションでの Web アプリケーション・ディレクトリ構造の使用

デフォルトでは、OC4J にはデフォルトのアプリケーションが構成されており、その中にデフォルトの Web アプリケーションが組み込まれています。テスト・ファイルにデフォルトの Web アプリケーションを使用するには、j2ee/home/default-web-app ディレクトリ（デフォルトの Web アプリケーションのデフォルト・ディレクトリ）の下の標準 Web アプリケーション・ディレクトリ構造にファイルを配置します。

次の手順を実行します。

1. server.xml ファイルで関連する構成をチェックします。まず、次のエントリを見て、デフォルトのアプリケーションの名前を定義し、アプリケーション・ディスクリプタを指定します。

```
<global-application name="default" path="application.xml" />
```

また、次のエントリも見て、Web サイトの XML ファイルも指定します。

```
<web-site path="./http-web-site.xml" />
```

2. OC4J グローバル `application.xml` ディスクリプタ内を参照します。次のエントリを見て、デフォルトの Web アプリケーション名を定義し、そのルート・ディレクトリを指定します。

```
<web-module id="defaultWebApp" path="../../home/default-web-app" />
```

3. `server.xml` に示されている Web サイトの XML ファイル内を参照します（デフォルトでは、OC4J スタンドアロンの `http-web-site.xml`）。次のエントリを見て、`defaultWebApp` を Web サイトのデフォルトの Web アプリケーションとして定義します。（前述したように、「/」は OC4J スタンドアロンにおけるデフォルトの Web アプリケーションのコンテキスト・パスであり、`<default-web-app>` 要素の `root` 属性は不要です。）

```
<default-web-app application="default" name="defaultWebApp" />
```

4. この構成を前提に、ファイルを次のようにデプロイします。

```
j2ee/home/default-web-app/
   index.html
   WEB-INF/
      web.xml
      classes/
         TestServlet.class
```

他に何も処理することなくサーブレットを起動できます。

代替 Web アプリケーションでの Web アプリケーション・ディレクトリ構造の使用

前項では、OC4J のデフォルトの Web アプリケーションである `defaultWebApp` は、Web アプリケーション・ディレクトリ構造のデプロイに使用されています。もう 1 つの方法として、その他の Web アプリケーションを定義して、OC4J のデフォルトのアプリケーションに組み込むこともできます。これは、デフォルトの Web アプリケーションと同様の機能は必要でも、未設定のディレクトリにデプロイする場合に便利です。

この項では、デフォルトのアプリケーション内に Web アプリケーション `myDefaultWebApp` を定義する手順を説明します。

1. OC4J グローバル `application.xml` ファイルに `<web-module>` 要素を追加します。この要素は、OC4J の新規 Web アプリケーションの名前を定義し、ルート・ディレクトリを指定します。OC4J のデフォルトの Web アプリケーションのエントリも、比較用に示しています。

```
<web-module id="defaultWebApp" path="../../home/default-web-app" />
<web-module id="myDefaultWebApp" path="../../home/my-default-web-app" />
```

2. Web サイトの XML ファイル `http-web-site.xml` に `<web-app>` 要素を追加します。この要素は、Web アプリケーションとコンテキスト・パスを結び付けます。OC4J のデフォルトの Web アプリケーションの `<default-web-app>` 要素も、比較用に示しています。

```
<default-web-app application="default" name="defaultWebApp" />
<web-app application="default" name="myDefaultWebApp" root="/mydefroot" />
```

3. この構成を前提に、ファイルを次のようにデプロイします。他に何も処理することなくサーブレットを起動できます。

```
j2ee/home/my-default-web-app/
   index.html
   WEB-INF/
      web.xml
      classes/
         TestServlet.class
```

OC4J スタンドアロンにおけるアプリケーションのアンデプロイまたは再デプロイ

テスト段階では、アプリケーションの変更および再デプロイが必要になります。次の項では、OC4J スタンドアロンのアンデプロイおよび再デプロイ機能について説明します。

- admin.jar を使用したアプリケーションのアンデプロイ
- admin.jar を使用したアプリケーションの再デプロイ
- 手動による WAR ファイルの再デプロイ
- ファイル操作後のアプリケーションの再デプロイのトリガー

admin.jar を使用したアプリケーションのアンデプロイ

アプリケーションの使用を終了した後、次のように admin.jar を使用して、アンデプロイすることができます。

```
% java -jar admin.jar ormi://oc4j_host:oc4j_ormi_port
                          adminuser adminpassword
                          -undeploy appname
```

これにより、server.xml および Web サイトの XML ファイルから関連するエントリが削除され、作成およびコピーされた全ディレクトリとファイルも削除されます。ORMI ポートの詳細は、5-24 ページの「OC4J スタンドアロンの起動および停止」を参照してください。 *adminuser* および *adminpassword* の詳細は、5-23 ページの「管理ユーザーおよびパスワードの設定」を参照してください。

たとえば、5-28 ページの「サンプル・デプロイ」で示した utility.ear アプリケーションをアンデプロイするには、次のようにします。

```
% java -jar admin.jar ormi://myhost:23791 admin welcome -undeploy utility
```

注意： 再デプロイする前のアプリケーションのアンデプロイは不要です。admin.jar の -undeploy オプションでは、永続的に削除されます。

---

admin.jar を使用したアプリケーションの再デプロイ

アプリケーションを最初に OC4J スタンドアロンにデプロイするときは、admin.jar の -deploy コマンドを使用して再デプロイします。ユーザーにとっては違いはありません。最初にユーティリティによって自動的に効率的にアンデプロイされるため、再デプロイに最適な状態から開始できます。-deploy コマンドの詳細は、5-26 ページの「EAR ファイルの OC4J スタンドアロンへのデプロイ」を参照してください。

再デプロイする前に、EAR ファイルを再パッケージして、更新済のファイルをすべて組み込むようにする必要があります。orion-web.xml や orion-application.xml などの OC4J ディスクリプタをサーバー上で更新し、変更を保持するには、これらのディスクリプタも EAR ファイルに組み込む必要があります。EAR ファイルが更新された場合、以前にコピーまたは生成された OC4J ディスクリプタは失われます。

手動による WAR ファイルの再デプロイ

OC4J のデフォルトのアプリケーションへの独立した WAR ファイルのデプロイについては、5-31 ページの「OC4J スタンドアロンへの独立した WAR ファイルのデプロイ」で説明しました。再デプロイする場合は、デプロイの手順をさかのぼって WAR ファイルを再パッケージし、デプロイの手順を繰り返します。これはエキスパート・モードです。（詳細は、5-5 ページの「Oracle のデプロイ・ツールの使用とエキスパート・モードの使用」を参照してください。）

次にまとめます。

1. グローバル application.xml ファイルおよび http-web-site.xml ファイル（または他の Web サイトの XML ファイル）への更新をコメントアウトします。

2. WAR ファイルを、その更新内容で更新または再パッケージします。orion-web.xml に保存するものがある場合は、WAR ファイルにそれを含めます。

3. 新規の WAR ファイルを元のターゲット・ディレクトリに戻します。ディレクトリに元のファイルが残っている場合は上書きします。

4. application.xml および http-web-site.xml の更新は、非コメント化しておきます。application.xml を更新すると、OC4J で、初期デプロイ時に WAR ファイルが解凍され、orion-web.xml ファイルがコピーまたは生成されます。

ファイル操作後のアプリケーションの再デプロイのトリガー

サーバー上の所定の位置でファイルが変更された場合にアプリケーションを再デプロイするトリガーは、OC4J のポーリング（OC4J スタンドアロンではデフォルトで有効）に応じて複数あります。（これはエキスパート・モードです。詳細は、5-5 ページの「Oracle のデプロイ・ツールの使用とエキスパート・モードの使用」を参照してください。）

- /WEB-INF/classes の下のサーブレット・クラス・ファイルへの変更

  /WEB-INF/classes にあるサーブレットの .class ファイルを更新すると、OC4J ポーリングが有効化されているかどうかに関係なく、次のリクエスト時に、サーブレットとその依存クラスが再ロードされ、Web アプリケーションが再デプロイされます。

注意：

- サーブレットが事前ロードに設定されている場合、サーブレットおよびその依存クラスは次のリクエストを待たずに即時に再ロードされます。これは、load-on-startup 設定に基づきます。2-6 ページの「サーブレットの事前ロード」を参照してください。

- global-web-application.xml または orion-web.xml の <classpath> 要素で指定されたディレクトリ内のサーブレット・クラス・ファイルを変更した場合も、/WEB-INF/classes 内のサーブレット・クラス・ファイルを変更した場合と同じ影響があります。ただし、<classpath> 位置にある JAR ファイルまたは依存クラス（JavaBean など）は、変更しても影響はありません。<classpath> 要素の詳細は、6-2 ページの「global-web-application.xml および orion-web.xml の要素の説明」を参照してください。

- 標準ディスクリプタへの変更（タイムスタンプを変更するものすべて）

  ポーリングが有効なときに web.xml を変更した場合、OC4J タスク・マネージャの次の実行時に Web アプリケーションが再デプロイされます。デフォルトでは 1 秒につき 1 回です。この Web アプリケーション内のサーブレットおよび依存クラスは、次のリクエスト時に再ロードされます。

  ポーリングが有効なときに application.xml を変更した場合、J2EE アプリケーションが再デプロイされます。この Web アプリケーション内のサーブレットおよび依存クラスは、次のリクエスト時に再ロードされます。

- /WEB-INF/lib の下のライブラリ JAR ファイルへの変更

  ポーリングが有効なときに /WEB-INF/lib 内の JAR ファイルを変更した場合、OC4J タスク・マネージャの次の実行時に Web アプリケーションが再デプロイされます。この Web アプリケーション内のサーブレットおよび依存クラスは、次のリクエスト時に再ロードされます。

- OC4J の development フラグの true の設定

  development フラグは、global-web-application.xml および orion-web.xml 内の <orion-web-app> 要素の属性です。development の設定が true の場合、特定のディレクトリ（デフォルトではアプリケーションの /WEB-INF/classes ディレクトリ）について、サーブレット・ソース・ファイルに更新があるかどうかが OC4J サーバーによってチェックされます。ソース・ファイルが前回のリクエスト以降に変更されていると、次のリクエスト時に、OC4J によってサーブレットが再コンパイルされ、Web アプリケーションが再デプロイされ、サーブレットとすべての依存クラスが再ロードされます。development の詳細は、6-2 ページの「global-web-application.xml および orion-web.xml の要素の説明」の説明を参照してください。

- JSP アプリケーションの場合、JSP の main_mode フラグの recompile の設定

  詳細は、『Oracle Application Server Containers for J2EE JavaServer Pages 開発者ガイド』を参照してください。

- OC4J の停止および再起動

  5-24 ページの「OC4J スタンドアロンの起動および停止」を参照してください。

OC4J スタンドアロンでは、ポーリングは server.xml の check-for-updates フラグによって制御されます。フラグは、デフォルトでは true に設定されています。もう 1 つの方法として、admin.jar の -updateConfig オプションを使用してワンタイム・ポーリングをトリガーできます。2-3 ページの「開発用の主な OC4J のフラグ」を参照してください。

注意： 前述の使用例について、次の点に注意してください。

- ここでは、Web アプリケーションの再デプロイとは、OC4J による Web アプリケーションの実行領域からの削除、Web アプリケーションの実行に関連付けられたクラスローダーの削除、web.xml と orion-web.xml の再解析、サーブレット・リスナー、フィルタおよびマッピングの再初期化という一連の処理を指します。
- 確実に正常に起動するために、再デプロイ後は OC4J をシャットダウンおよび再起動してください。5-24 ページの「OC4J スタンドアロンの起動および停止」を参照してください。
- 再デプロイによって、サーバーのデプロイディレクトリにある orion-application.xml や orion-web.xml などの OC4J ディスクリプタが著しく影響を受けることはありません。再ロードのトリガー後、以前にコピーまたは生成されたファイルには、指定したデフォルト以外の設定がすべて保持されます。

Oracle Application Server での OC4J のデプロイ

この項では、Oracle Application Server 環境における、OC4J へのデプロイおよび再デプロイについて説明します。

Oracle Application Server では、アプリケーションの起動、停止、構成およびデプロイに、Enterprise Manager または dcmctl コマンドライン・ユーティリティのパラレル・コマンドのいずれかを使用する必要があります。これらのツールは両方とも、Oracle Application Server の Distributed Configuration Management（DCM）サブシステムと組み合されています。OC4J スタンドアロンのユーティリティ admin.jar を使用して、Oracle Application Server インスタンス内で OC4J インスタンスを管理することはできません。また、Oracle Application Server では構成ファイルを手動で更新しないでください。（詳細は、5-5 ページの「Oracle のデプロイ・ツールの使用とエキスパート・モードの使用」を参照してください。）

注意： Oracle Application Server では、Enterprise Manager または dcmctl のどちらか一方を使用してください。両方を同時に使用して同じ OC4J インスタンスをターゲットにしたり、同じデプロイの異なる部分で両方を使用しないでください。

この項には、次の項目が含まれます。

- Oracle Application Server における OC4J のデプロイおよび構成の概要
- Oracle Application Server での OC4J のデフォルト Web アプリケーション
- Oracle Application Server でのアプリケーションのアンデプロイおよび再デプロイ

Oracle Application Server における OC4J のデプロイおよび構成の概要

Web モジュールのデプロイおよび構成に使用する Enterprise Manager ページの詳細は、第 7 章「Enterprise Manager を使用した構成」で説明されています。OC4J での、Enterprise Manager または `dcmctl` コマンドライン・ユーティリティの使用の詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

この章の前半で説明した、`admin.jar` を使用するデプロイ方法や手動でのアプリケーション・ファイルのデプロイ方法は、Oracle Application Server 環境には適用しないでください。Enterprise Manager には、EAR ファイルまたは WAR ファイルをデプロイするためのページがあります。これらのページについては、7-4 ページの「Application Server Control コンソールの「アプリケーションのデプロイ」ページ」および 7-5 ページの「Application Server Control コンソールの「Web アプリケーションのデプロイ」ページ」を参照してください。Oracle Application Server では、OC4J スタンドアロンで説明したような、手動による EAR または WAR ファイルのデプロイ、あるいは非アーカイブ・ファイルのデプロイは行わないでください。

Oracle Application Server では、EAR や WAR ファイルのコピー、これらのファイルのディレクトリ構造への解凍、および OC4J ディスクリプタ（`orion-web.xml` および `orion-application.xml` など）のコピーまたは生成は、通常、Enterprise Manager を通して自動的かつ透過的に処理されます。

Enterprise Manager または `dcmctl` を使用した Oracle Application Server へのデプロイでは、Web アプリケーションが Oracle HTTP Server に自動的に登録され、`mod_oc4j.conf` ファイルに新規のマウント・ポイントができます。Enterprise Manager で指定した URL マッピング（`/mypath/myapp` など）により、マウント・ポイントが定義されます。マウント・ポイントによって、処理のために Oracle HTTP Server から OC4J にルーティングする URL リクエストが判別されます。この場合では、`/mypath/myapp`（ホストとポートの後）で始まる URL リクエストがすべて OC4J に渡されます。

注意 :

- Oracle Application Server では、すべてのアプリケーションが EAR ファイルにデプロイされます。Enterprise Manager で Application Server Control コンソールの「Web アプリケーションのデプロイ」ページを使用する場合、WAR ファイルが要求され、EAR ファイルが透過的に作成されて WAR ファイルが組み込まれます。
- OC4J へのデプロイ時には、「/」のコンテキスト・パスを指定しないでください。

第 7 章で説明する Enterprise Manager の一部のページを使用して、サーブレットまたは Web サイトのパラメータを構成できます。これらのページで設定を操作することにより、適切な構成に自動的に更新されます。この章の前半で説明した構成ファイルは、Oracle Application Server では OC4J によって使用されますが、ほとんどが透過的です。次に、付加的な実際の操作について考えます。

- Oracle Process Manager and Notification Server（OPMN）サブシステムは、一部のシステム・プロパティや環境変数の他に、構成ファイルの設定の一部も動的にオーバーライドします。
- DCM サブシステムは、構成情報のリポジトリを管理しています。このリポジトリには、構成ファイルではなく、実際の構成設定が含まれています。

これらの理由により、Oracle Application Server では、構成の手動での更新は絶対に行わないでください。

なんらかの理由で Enterprise Manager を使用せずに構成ファイルを変更する必要がある場合、dcmctl の更新コマンドを実行して DCM に変更を通知することになります。これは、DCM が管理している OC4J のすべてのインスタンスに影響するため、避ける必要があります。

OPMN と DCM の基本については、『Oracle Application Server 管理者ガイド』を参照してください。『Oracle Application Server 管理者ガイド』には、dcmctl ツールに関する説明もあります。Enterprise Manager の詳細は、『Oracle Enterprise Manager 概要』も参照してください。

Oracle Application Server での OC4J のデフォルト Web アプリケーション

OC4J スタンドアロンのデフォルトの Web アプリケーションは、開発段階での使用を意図したものであり、これについては 5-24 ページの「OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーション」で説明しました。Oracle Application Server 環境でも、OC4J にデフォルトの Web アプリケーションはありますが、これは開発者用ではありません。

Oracle Application Server では、Web サイトは 1 つのみであり、そのルートのネームスペースは OC4J ではなく Oracle HTTP Server が所有しています。Oracle Application Server 環境における OC4J のデフォルトの Web アプリケーションの概念は、OC4J 自体が Web サイトを所有する OC4J スタンドアロンのものほど明確ではありません。さらに、OC4J スタンドアロンでは、デフォルトの Web アプリケーションを使用するために手動でファイルを操作しますが、これは Oracle Application Server の場合、適切ではありません。

Oracle Application Server では、前述したように、Oracle HTTP Server から OC4J へのルーティングが mod_oc4j.conf ファイル内のマウント・ポイントを介して実行されます。Oracle Application Server では、アプリケーションを OC4J にデプロイするたびに（7-4 ページの「Application Server Control コンソールの「アプリケーションのデプロイ」ページ」および7-5 ページの「Application Server Control コンソールの「Web アプリケーションのデプロイ」ページ」で説明）、コンテキスト・パスとして指定した URL が別のマウント・ポイントとして指定されます。

デフォルトの Web アプリケーション defaultWebApp のデフォルトの OC4J のマウント・ポイントとコンテキスト・パス /j2ee の 1 対は、OC4J システムが使用するためのものです。たとえば、リクエストの URL パターンが OC4J のマウント・ポイントと一致するため OC4J にルーティングはしても、指定された Web アプリケーションが見つからない場合、OC4J はこのデフォルトの Web アプリケーションを使用してエラー・メッセージを出力します。このコンテキスト・パスおよびデフォルトの Web アプリケーションは、default-web-site.xml の <default-web-app> 要素で指定されています。この要素は必須ですが、開発者が直接使用するものではありません。

Oracle Application Server でのアプリケーションのアンデプロイおよび再デプロイ

Oracle Enterprise Manager 10*g* には、アプリケーションをアンデプロイおよび再デプロイする機能があります。次の項でこれらの機能を説明します。

- Enterprise Manager を使用したアプリケーションのアンデプロイ
- Enterprise Manager を使用したアプリケーションの再デプロイ

注意： Oracle Application Server で Enterprise Manager を使用すると、Application Server Control コンソールの OC4J の「アプリケーション」ページで、アプリケーション・リストからアプリケーションを選択して適切なボタンをクリックすることで、アンデプロイおよび再デプロイできます。いずれの場合も、最初にスタンドアロンの WAR ファイルをデプロイしていれば、デプロイ処理時に WAR ファイルが EAR ファイルに自動的にラップされています。このため、アプリケーション・リストにアプリケーションが表示されます。

Enterprise Manager を使用したアプリケーションのアンデプロイ

Oracle Application Server の使用しなくなった J2EE アプリケーションは、Enterprise Manager で Application Server Control コンソールの OC4J の「アプリケーション」ページを使用してアンデプロイできます。アプリケーション・リストからアプリケーションを選択（対応するラジオ・ボタンを選択）し、**「アンデプロイ」**ボタンをクリックします。このプロセスにより、作成およびコピーされたすべてのディレクトリとファイルが削除され、サーバー構成が適切に更新されます。

このページの図と詳細は、7-3 ページの「Application Server Control コンソールの OC4J の「アプリケーション」ページ」を参照してください。

注意： 再デプロイする前のアプリケーションのアンデプロイは不要です。Enterprise Manager の `Undeploy` 機能では、永続的に削除されます。

Enterprise Manager を使用したアプリケーションの再デプロイ

Enterprise Manager で Application Server Control コンソールの OC4J の「アプリケーション」ページを使用し、Oracle Application Server において J2EE アプリケーションを再デプロイできます。アプリケーション・リストからアプリケーションを選択（対応するラジオ・ボタンを選択）し、**「再デプロイ」**ボタンをクリックします。EAR ファイルへのパスの入力を促すプロンプトが表示されます。

このページの図と詳細は、7-3 ページの「Application Server Control コンソールの OC4J の「アプリケーション」ページ」を参照してください。

前回のデプロイから EAR ファイルを変更していない場合は、再デプロイ時に、前回デプロイ時のサーバー構成の設定が保持されます。これは、標準構成（`application.xml` および `web.xml` などに基づく）が EAR ファイルに含まれる場合の OC4J ディスクリプタ（`orion-application.xml` および `orion-web.xml` など）の設定に、特に該当します。

ただし、EAR ファイルを変更している場合、前回のサーバー構成は廃棄されます。これは、OC4J ディスクリプタ（ある場合）や適用可能なデフォルト値などの、EAR ファイルからの情報によって置き換えられます。この使用例では、サーバー上で OC4J 固有の構成を変更した場合、EAR ファイルの OC4J ディスクリプタにも同じ変更を加えてその変更を維持する必要があります。

再デプロイ後、第 7 章「Enterprise Manager を使用した構成」で説明されている OC4J サーブレットおよび Web サイトの構成ページをチェックして、必要な構成の設定が保持されているかどうかを検証できます。

6 構成ファイルの説明

この章では、サーブレットと Web サイトを構成する、OC4J 構成ファイルの要素および属性について説明します。次の項が含まれます。

- global-web-application.xml および orion-web.xml の構成
- Web サイトの XML ファイルの構成

注意： この章での構成ファイルおよびその要素と属性に関する説明は、OC4J スタンドアロン開発環境を前提としています。Enterprise Manager を使用する Oracle Application Server 環境では、構成は、Application Server Control コンソールの Web モジュール・ページを使用して行われるため、ファイルとそのプロパティの多くは、ユーザーには表示されません。Oracle Application Server で Enterprise Manager を使用して、本番アプリケーションの構成とデプロイを行う場合の考慮事項は、第 7 章「Enterprise Manager を使用した構成」を参照してください。

---

global-web-application.xml および orion-web.xml の構成

次の項では、global-web-application.xml および orion-web.xml の構成ファイルの詳細を説明します。

- global-web-application.xml および orion-web.xml の要素の説明
- global-web-application.xml および orion-web.xml の DTD
- global-web-application.xml および orion-web.xml の階層表現
- サンプルの global-web-application.xml の設定

これらのファイルの概要は、5-15 ページの「OC4J および J2EE の Web ディスクリプタ」を参照してください。

global-web-application.xml および orion-web.xml の要素の説明

この項では、global-web-application.xml ファイルおよび orion-web.xml ファイルの要素と属性について説明します。

この項の要素の説明は、global-web-application.xml またはアプリケーション固有の orion-web.xml 構成ファイルのいずれかに適用可能です。global-web-application.xml ファイルでグローバル・アプリケーションを構成してデフォルトを設定し、必要に応じて、orion-web.xml ファイルで特定のアプリケーションのデプロイ用にこれらのデフォルトをオーバーライドします。

<orion-web-app ... >

Web アプリケーションの OC4J 固有の構成を指定するためのルート要素です。

注意：

- <orion-web-app> 要素の autoreload-jsp-pages 属性と autoreload-jsp-beans 属性は、OC4J JSP コンテナでは現在サポートされていません。autoreload-jsp-pages と同等の機能を持つ JSP の main_mode 構成パラメータを使用できます。このパラメータの詳細は、『Oracle Application Server Containers for J2EE JavaServer Pages 開発者ガイド』を参照してください。
- <servlet-filter> サブ要素と、document-root、get-locale-from-user、internationalize-resources および default-mime-type 属性はサポートされていません。

<orion-web-app> のサブ要素は次のとおりです。

<classpath>
<context-param-mapping>
<mime-mappings>
<virtual-directory>
<access-mask>
<cluster-config>
<servlet-chaining>
<request-tracker>
<session-tracking>
<resource-ref-mapping>
<env-entry-mapping>
<security-role-mapping>
<ejb-ref-mapping>
<expiration-setting>
<jazn-web-app>
<web-app-class-loader>
<authenticate-on-dispatch>
<web-app>

<orion-web-app> の属性は次のとおりです。

- default-buffer-size: サーブレットのレスポンスに関する出力バッファのデフォルトのサイズを、バイト単位で指定します。デフォルトは、2048 です。

> **注意：** default-buffer-size 属性は JSP のバッファ・サイズには影響を与えません。

- default-charset: デフォルトで使用する ISO キャラクタ・セットを指定します。デフォルトは、iso-8859-1 です。
- deployment-version: この Web アプリケーションがデプロイされている OC4J のバージョンを示します。この値が現在のバージョンと一致しない場合、アプリケーションが再度デプロイされます。これは内部サーバー値のため、変更しないでください。
- development: 開発時の便宜のために設けられたフラグです。development が true に設定されている場合、特定のディレクトリについて、サーブレット・ソース・ファイルに更新があるかどうかが OC4J サーバーによってチェックされます。ソース・ファイルが前回のリクエスト以降に変更されていると、次のリクエスト時に、OC4J によってサーブレットが再コンパイルされ、Web アプリケーションが再デプロイされ、サーブレットとすべての依存クラスが再ロードされます。

  対象のディレクトリは、source-directory 属性の設定により判断されます（次の説明を参照してください）。development でサポートされている値は、true および false（デフォルト）です。

> **注意：** OC4J JSP コンテナは development フラグを現在サポートしていません。このフラグはサーブレット専用です。JSP ページには、類似した機能を持つ、JSP の main_mode フラグを使用してください。これについては、『Oracle Application Server Containers for J2EE JavaServer Pages 開発者ガイド』で説明されています。development フラグに関連する旧 Orion JSP コンテナの機能は、OC4J JSP コンテナには適用されません。

- source-directory: development 属性が true に設定されている場合は、source-directory 設定で指定された場所で、自動コンパイルの対象となるサーブレット・ソース・ファイルが検索されます。デフォルトは、存在する場合は /WEB-INF/src で、それ以外の場合は /WEB-INF/classes です。
- directory-browsing: 「/」で終わる URL のディレクトリのブラウズの許可を指定します。サポートされている値は、allow および deny（デフォルト）です。

  次のような状況を仮定します。

  – アプリケーション・ルート・ディレクトリに index.html ファイルがありません。

  – web.xml ファイルに初期ファイルが定義されていません。

  このような状況で directory-browsing が allow に設定されている場合、「/」で終わる URL については、ユーザーのブラウザに、対応するディレクトリの内容が表示されます。

  同じ状況で directory-browsing が deny に設定されている場合は、「/」で終わる URL はエラーとなり、ディレクトリの内容は表示されません。

  アプリケーション・ルート・ディレクトリに、定義された初期ファイルまたは index.html ファイルがある場合、そのファイルの内容は directory-browsing 設定にかかわらず表示されます。

- file-modification-check-interval: HTML ファイルなどの静的ファイルに適用されます。ファイル・チェックを有効化する期間をミリ秒単位で指定します。前回のチェック以降、この期間内であれば、別途チェックを行う必要はありません。0（ゼロ）または負の数値を指定すると、常にチェックが発生します。デフォルトは、1000 です。パフォーマンス上の理由から、本番環境では非常に大きな値（たとえば、1000000）を指定することをお薦めします。

- jsp-print-null: このフラグを false に設定すると、JSP ページからの NULL 出力に対して、デフォルトの null 文字列ではなく、空の文字列が出力されます。デフォルトは、true です。
- jsp-timeout: 整数値を秒単位で指定します。この時間を経過すると、リクエストされなかった JSP ページがメモリーから削除されます。これによって、頻繁にコールされないページがある場合、リソースが解放されます。デフォルト値は 0（ゼロ）で、タイムアウトは発生しません。
- jsp-cache-directory: JSP キャッシュ・ディレクトリを指定します。このディレクトリは、JSP トランスレータからの出力ファイルに対するベース・ディレクトリとして使用されます。また、アプリケーション・レベルの TLD キャッシュのベース・ディレクトリとしても使用されます。デフォルト値は ./persistence で、これはアプリケーションのデプロイメント・ディレクトリからの相対パスです。
- jsp-cache-tlds: このフラグは、永続 TLD キャッシュの JSP ページにおける有効性を示します。TLD キャッシュは、TLD ファイルが指定されているタグ・ライブラリの場所にあるグローバル・レベル、および TLD ファイルが WEB-INF ディレクトリの下にあるアプリケーション・レベルの両方で実装されます。すべてのアプリケーション・ファイルから TLD ファイルを検索する場合は、true または on（デフォルト）に設定します。standard に設定すると、/WEB-INF および /WEB-INF/classes または /WEB-INF/lib 以外のサブディレクトリのみで TLD ファイルが検索されます。false または off に設定すると、この機能は無効になります。指定されている場所は jsp-taglib-locations 属性に基づきます。
- jsp-taglib-locations: 永続 TLD キャッシュが JSP ページで使用可能になっている場合（jsp-cache-tlds 属性で指定）、jsp-taglib-locations を使用して、指定の場所として使用する 1 つ以上のディレクトリを、セミコロン区切りのリストにして指定できます。タグ・ライブラリの JAR ファイルをこれらの場所に置いて、複数の JSP ページと Web アプリケーション間で共有したり、TLD キャッシュに使用することができます。

  ディレクトリの絶対パスまたは相対パスは、任意の組合せで指定できます。相対パスは、*ORACLE_HOME* が定義されている場合は *ORACLE_HOME* の下から、*ORACLE_HOME* が定義されていない場合は（OC4J プロセスが起動された）カレント・ディレクトリの下からになります。デフォルト値は次のとおりです。

  – *ORACLE_HOME* が定義されている場合は、*ORACLE_HOME*/j2ee/home/jsp/lib/taglib/

  または

  – *ORACLE_HOME* が定義されていない場合は、./jsp/lib/taglib

  重要： jsp-taglib-locations 属性は global-web-application.xml でのみ使用し、orion-web.xml では使用しないでください。

- simple-jsp-mapping: *.jsp がアプリケーションに影響を与える Web ディスクリプタ（global-web-application.xml、web.xml および orion-web.xml）の <servlet> 要素の oracle.jsp.runtimev2.JspServlet フロントエンド JSP サーブレットのみにマップされている場合は、このフラグを true に設定します。これにより、JSP ページのパフォーマンスが向上します。デフォルトの設定は、false です。
- enable-jsp-dispatcher-shortcut: デフォルトでは true に設定します。特に、simple-jsp-mapping 属性も true に設定している場合は、OC4J JSP コンテナのパフォーマンスが大幅に向上します。これは、多数の jsp:include 文を含む JSP ページに特に当てはまります。ただし、true の設定では、web.xml の <jsp-file> 要素を使用して JSP ファイルを定義する場合、それらの JSP ファイルに対応した <url-pattern> の仕様があることが前提になります。

注意： jsp-print-null、jsp-timeout、jsp-cache-directory、jsp-cache-tlds、jsp-taglib-locations、simple-jsp-mapping および enable-jsp-dispatcher-shortcut 属性に関連する処理は、OC4J JSP コンテナにより行われます。これらの属性および関連する機能の詳細は、『Oracle Application Server Containers for J2EE JavaServer Pages 開発者ガイド』を参照してください。

- persistence-path: サーバーの再起動またはアプリケーションの再デプロイ中に、サーブレットの HttpSession オブジェクトが OC4J によって永続的に保存される場所を示します。相対パスを指定します。これは、application-deployments ディレクトリの下にある OC4J の一時記憶領域からの相対パスです。デフォルト値はありません。値が定義されていない場合、再起動または再デプロイ中にセッション・オブジェクトの永続性は失われます。次に例を示します。

```
<orion-web-app ... persistence-path="persistdir" ... >
  ...
</orion-web-app>
```

  この機能を有効にするには、セッション・オブジェクトは、シリアライズ可能（直接または間接的に java.io.Serializable インタフェースを実装）、またはリモート可能（直接または間接的に java.rmi.Remote インタフェースを実装）である必要があります。

  <orion-web-app> 要素の <cluster-config> サブ要素に基づいて OC4J クラスタリングが有効な場合、persistence-path 属性は無視されます。

- servlet-webdir: サーブレットを起動するためのパスをクラス名で指定します。URL 内のこのパス以後に表示される名前は、すべてクラス名とみなされ、必要に応じて、パッケージが含まれます。

  この機能は通常、開発時およびテスト時に OC4J スタンドアロン環境で使用されます。デプロイ時は、標準の web.xml を使用してコンテキスト・パスおよびサーブレット・パスを定義してください。

  次に示すのは、クラス名によるサーブレットの起動例です。設定を servlet-webdir="/servlet/" と仮定します。

```
http://www.example.com:8888/servlet/foo.bar.SessionServlet
```

重要：

- servlet-webdir をスラッシュ（「/」）で始まる設定にすると、クラス名による起動が有効になります。これには重大なセキュリティ上のリスクがあるため、本番環境では使用しないでください。servlet-webdir=""（空の引用符）に設定、または OC4J のシステム・プロパティ http.webdir.enable の値を false に設定すると、クラス名による起動を無効にできます。
- アプリケーションの servlet-webdir 属性のデフォルト値には、global-web-application.xml のデフォルト値が使用されます（設定されている場合）。global-web-application.xml に値が設定されていない場合、デフォルト値は "" です。

2-22 ページの「OC4J 開発時におけるクラス名によるサーブレットの起動」および 2-38 ページの「その他のセキュリティに関する考慮事項」も参照してください。

- temporary-directory: サーブレットおよび JSP ページによりスクラッチ・ファイル用に使用される一時ディレクトリのパスです。パスは、絶対パスまたはデプロイメント・ディレクトリからの相対パスのいずれかにできます。デフォルトの設定は、./temp です。

サーブレットは、一時ディレクトリを使用する場合があります。たとえば、ユーザーがフォーム（情報がデータベースに書き込まれる前の、仮または短期間の格納用）にデータを入力するときに、ディスクに情報を書き込むために使用します。

次に、特定のディレクトリは、サーブレット・コンテキストから再コールされます。このコンテキストは、属性 javax.servlet.context.tempdir により使用できます。次に例を示します。

```
File file = (File)application.getAttribute("javax.servlet.context.tempdir");
```

java.io.File オブジェクトが戻されます。このオブジェクトから、ディレクトリ情報およびコンテンツを取得できます。

<classpath ... >

この要素を使用して、Web アプリケーションのクラスロードに対する追加コードの場所（ライブラリ・ファイル、または個別のクラス・ファイルの場所のいずれか）を OC4J に通知します。

<classpath> の属性は次のとおりです。

- path: カンマまたはセミコロンで区切られた、1 つ以上の場所を指定できます。場所は、次のいずれかになります。
  - ファイル名を含む、JAR または ZIP ファイルの完全なパス
  - ディレクトリ・パス

  いずれの場合も、絶対パス、または構成ファイルのある位置（global-web-application.xml または orion-web.xml の該当するもの）からの相対パスを使用できます。

  ディレクトリ・パスを指定した場合、クラスローダーでは、指定したディレクトリ内にある個別のクラス・ファイルのみが認識されます。JAR または ZIP ファイルは認識されません（個別に指定されている場合を除く）。

  たとえば、orion-web.xml が次のように設定されていると仮定します。

  ```
  <classpath path=/abc/def/lib1.jar,/abc/def/zip1.jar,/abc/def,mydir />
  ```

  クラスローダーでは、次が認識されます。

  - lib1.jar および zip1.jar ライブラリ（/abc/def のその他のライブラリは除く）
  - /abc/def 内のクラス・ファイルすべて
  - orion-web.xml からの相対位置で示されている mydir 内のクラス・ファイルすべて

<context-param-mapping ... >*deploymentValue*</context-param-mapping>

この要素は、orion-web.xml 内で、web.xml ファイルの context-param 設定の値をオーバーライドします。EAR の構成にデプロイ固有の値を直接反映しないようにするために使用します。タグのボディに新しい値を指定します。

<context-param-mapping> の属性は次のとおりです。

- name: オーバーライドする context-param 設定の名前を指定します。

<mime-mappings ... >

使用する MIME マッピングが入っているファイルのパスを定義します。

<mime-mappings> の属性は次のとおりです。

- path: ファイルのパスまたは URL を指定します。絶対パス（URL）または orion-web.xml ファイルがある場所からの相対パス（URL）のいずれかです。

<virtual-directory ... >

たとえば、UNIX システム上のシンボリック・リンクと理論的には類似した方法で動作し、静的コンテンツに仮想ディレクトリ・マッピングを追加します。仮想ディレクトリを使用すると、Web アプリケーションの WAR ファイルには物理的に存在しないアプリケーションに使用できる、実際のドキュメント・ルート・ディレクトリのコンテンツを作成できます。これは、たとえば企業全体のエラー・ページを複数の WAR ファイルにリンクさせる際に役立ちます。

`<virtual-directory>` の属性は次のとおりです。

- `real-path`: 実際のパス。UNIX の場合は `/usr/local/realpath`、Windows の場合は `C:¥testdir` など。
- `virtual-path`: 指定された実際のパスにマップする仮想パス。

<access-mask ... >

このアプリケーションにオプションのアクセス・マスクを指定するには、`<access-mask>` のサブ要素を使用します。クライアントをフィルタするには、`<host-access>` サブ要素でホスト名またはドメインを使用するか、`<ip-access>` サブ要素で IP アドレスとサブネットを使用します。両方を使用することも可能です。

`<access-mask>` のサブ要素は次のとおりです。

<host-access>
<ip-access>

`<access-mask>` の属性は次のとおりです。

- `default`: `<host-access>` または `<ip-access>` サブ要素で識別されないクライアントからのリクエストを、許可するかどうかを指定します。サポートされている値は、`allow`（デフォルト）および deny です。これらのサブ要素で識別されるクライアントからのリクエストを許可するかどうかを指定するには、`<host-access>` および `<ip-access>` サブ要素に別の `mode` 属性を使用します。

<host-access ... >

`<access-mask>` のこのサブ要素は、アクセスの許可または拒否の対象となるホスト名またはドメインを指定します。

`<host-access>` の属性は次のとおりです。

- `domain`: ホストまたはドメインを指定します。
- `mode`: 特定のホストやドメインからのアクセスを許可するか、または拒否するかを指定します。サポートされている値は、`allow`（デフォルト）または `deny` です。

<ip-access ... >

`<access-mask>` のこのサブ要素は、アクセスの許可または拒否の対象となる、IP アドレスおよびサブネット・マスクを指定します。

`<ip-access>` の属性は次のとおりです。

- `ip`: 32 ビット値の IP アドレスを指定します（例 : `123.124.125.126`）。
- `netmask`: 対応するサブネット・マスクを指定します（例 : `255.255.255.0`）。
- `mode`: 特定の IP アドレスおよびサブネット・マスクからのアクセスを許可するか、または拒否するかを指定します。サポートされている値は、`allow`（デフォルト）または `deny` です。

<cluster-config ... >

この要素は、OC4J クラスタリングを使用する場合のみ使用します。それ以外の場合は、削除またはコメントアウトします。クラスタリングされたアプリケーションでは、クラスタ・アイランド内のクラスタ間で、HTTP セッション・データがレプリケートされます。HTTP セッション・データのオブジェクトは、セッションのレプリケーションの動作に対してシリアライズ可能（直接的または間接的に `java.io.Serializable` インタフェースを実装）、またはリモート可能（直接的または間接的に `java.rmi.Remote` インタフェースを実装）である必要があります。

クラスタリングに関する一般的な情報は、『Oracle Application Server パフォーマンス・ガイド』を参照してください。

`<cluster-config>` の属性は次のとおりです。

- `host`: クラスタ・データの送受信を行う、マルチキャスト用ホストまたは IP。デフォルトは、`230.230.0.1` です。
- `id`: クラスタ内で、クラスタ自身を識別するためのクラスタ・ノードの ID（番号）。デフォルトは、ローカル・マシンの IP に基づいています。
- `port`: クラスタ・データの送受信を行うポート。デフォルトは、`9127` です。

<servlet-chaining ... >

現在のサーブレットのレスポンスが特定の MIME タイプに設定されているときにコールするサーブレットを指定します。指定されたサーブレットは、現在のサーブレットの後にコールされます。これは、サーブレット・チェーンと呼ばれ、特定の種類の出力をフィルタしたり変換します。

> **注意：** サーブレット・チェーンは、標準サーブレット・フィルタと基本的に同様の機能を持つ旧式のメカニズムです。これは、サーブレット仕様のバージョン 2.3 で説明されています。かわりにサーブレット・フィルタを使用します。3-2 ページの「サーブレット・フィルタ」を参照してください。

`<servlet-chaining>` の属性は次のとおりです。

- `mime-type`: `text/html` など、チェーンをトリガーする MIME タイプを指定します。
- `servlet-name`: 特定の MIME タイプの場合にコールするサーブレットを指定します。サーブレット名は、`global-web-application.xml`、`web.xml` または `orion-web.xml` の `<web-app>` 要素の定義によって、サーブレット・クラスに関連付けられます。

<request-tracker ... >

リクエストのトラッキングに使用するサーブレットを指定します。リクエスト・トラッキングは、ログを記録する場合などに役立ちます。

`global-web-application.xml` ではなく、`orion-web.xml` にすべてのリクエスト・トラッキングを定義する必要があります。これは、`<request-tracker>` 要素が同じアプリケーション内で定義される 1 つのサーブレットを指し示すためです。

リクエスト・トラッキングは、対応するレスポンスがコミットされるとき（レスポンスが実際に送信される直前）、ブラウザからサーバーに送信される各個別のリクエストに対して起動されます。

リクエスト・トラッキングが複数存在する場合がありますが、各トラッキングは個別の `<request-tracker>` 要素で定義されています。

`<request-tracker>` の属性は次のとおりです。

- `servlet-name`: 起動するサーブレットを指定します。`web.xml` ファイルの対応する `<servlet-name>` 要素または `<servlet-class>` 要素（両方とも `<servlet>` 要素のサブ要素）に基づいて、サーブレット名またはクラス名のいずれかを指定できます。

<session-tracking ... >

このアプリケーションに対してセッション・トラッキングの設定を指定します。セッション・トラッキングは、Cookie（Cookie 対応のブラウザの場合）を使用して行われます。

注意 :

- Cookie が無効な場合、セッション・トラッキングを実行できるのは、サーブレットが、レスポンス・オブジェクトの `encodeURL()` メソッドまたはリダイレクト用の `encodeRedirectURL()` メソッドを明示的にコールする場合のみです。
- OC4J では、サーブレット・コンテナでセッション ID を自動的に URL にエンコードする、自動エンコーディングはサポートしていません。これは、コストのかかる標準外の処理です。したがって、OC4J は `<session-tracking>` 属性の `autoencode-urls` と `autoencode-absolute-urls` をサポートしていません。2-26 ページの「OC4J でのセッション・トラッキング」も参照してください。

サーブレット・セッションに関する一般的な情報は、2-24 ページの「サーブレットのセッション」を参照してください。

セッション・トラッキングに使用するサーブレットは、サブ要素により指定します。

`<session-tracking>` のサブ要素は次のとおりです。

```
<session-tracker>
```

`<session-tracking>` の属性は次のとおりです。

- `autojoin-session`: ユーザーがアプリケーションにログインすると同時に、そのユーザーにセッションを割り当てるかどうかを指定します。サポートされている値は、`true` および `false`（デフォルト）です。
- `cookies`: セッション Cookie を送信するかどうかを指定します。サポートされている値は、`enabled`（デフォルト）および `disabled` です。
- `cookie-domain`: Cookie に必要なドメインを指定します。この属性を使用して、複数の Web サイト上で単一クライアントまたはユーザーを追跡できます。設定はピリオド（`.`）から始まる必要があります。次に例を示します。

  ```
  <session-tracking cookie-domain=".us.oracle.com" />
  ```

  この場合、ユーザーが `.us.oracle.com` ドメイン・パターンに一致する任意のサイト（`webserv1.us.oracle.com` または `webserv2.us.oracle.com` など）にアクセスすると、同じ Cookie が使用および再使用されます。

  ドメイン仕様は少なくとも 2 つの要素（`.us.oracle.com` または `.oracle.com` など）で構成する必要があります。たとえば、`.com` という設定は無効です。

  次に、Cookie ドメイン機能が役立つ例を 2 つ示します。

  – この機能を使用して、異なるホスト上で実行されている Web アプリケーションのノード間において、セッションの状態を共有できます。

  – OC4J スタンドアロンでは、この機能を Web サイトの XML ファイルの `<web-app>` 要素が `shared="true"` に設定されている共有アプリケーションに使用できます。このようなアプリケーションでは、リクエストに使用されるポートはセキュアである場合と、セキュアでない場合があります。ここでのポートは、別個の Web サイトを意味します。使用されるポートに関係なく、同じ Cookie を使用します。この使用例では `cookie-domain` を使用する必要はありませんが、これはデフォルト・ポートとして HTTP に 80、HTTPS には 443 を使用している場合についてです。クライアントでは、これらをすでに同じ Web サイトの異なるポートとして認識済であり、単一の Cookie のみが使用されます。

- `cookie-max-age`: この数値は、セッション Cookie とともに送信され、ブラウザが Cookie をブラウザ Cookie キャッシュに保持する最大期間（秒単位）を指定します。デフォルトでは、ブラウザ・セッションの間 Cookie はメモリーに格納され、その後廃棄されます。

<session-tracker ... >

<session-tracking> のこのサブ要素は、セッション・トラッカとして使用するサーブレットを指定します。セッション・トラッキングは、たとえば、ログを記録する場合に役立ちます。

global-web-application.xml ではなく、orion-web.xml ですべてのセッション・トラッカを定義する必要があります。これは、<session-tracker> 要素が同じアプリケーション内で定義される 1 つのサーブレットを指し示すためです。

セッション・トラッカは、セッションが作成されると同時に起動されます。特に、HTTP セッション・リスナー（javax.servlet.http.HttpSessionListener インタフェースを実装するクラスのインスタンス）の sessionCreated() メソッドが起動されると同時に起動されます。

セッション・トラッカが複数存在することがありますが、各トラッカは個別の <session-tracker> 要素で定義されます。

<session-tracker> の属性は次のとおりです。

- servlet-name: 起動するサーブレットを指定します。web.xml ファイルの対応する <servlet-name> 要素または <servlet-class> 要素（両方とも <servlet> 要素のサブ要素）に基づいて、サーブレット名またはクラス名のいずれかを指定できます。

<resource-ref-mapping ... >

この要素を使用して、データ・ソースや JMS キュー、メール・セッションなどの外部リソースへの参照を宣言します。これは、デプロイ時に、リソース参照名を JNDI の場所に関連付けます。

<resource-ref-mapping> のサブ要素は次のとおりです。

<lookup-context>

<resource-ref-mapping> の属性は次のとおりです。

- location: リソースのルックアップを行う JNDI の場所を指定します。次に例を示します。

  ```
  location="jdbc/TheDS"
  ```

- name: リソースの参照名を指定します。web.xml ファイルの resource-ref 要素の名前と一致します。次に例を示します。

  ```
  name="jdbc/TheDSVar"
  ```

<lookup-context ... >

<resource-ref-mapping> のこのサブ要素は、リソースを取得するために使用するオプションの JNDI コンテキスト（javax.naming.Context インスタンス）を指定します。これは、サード・パーティ製のモジュール（サード・パーティ製の JMS サーバーなど）に接続する場合に役立ちます。リソース・ベンダーが提供する JNDI コンテキストの実装を使用するか、何も存在しない場合はベンダーのソフトウェアとのネゴシエーションを行う実装を作成します。

<lookup-context> のサブ要素は次のとおりです。

<context-attribute>

<lookup-context> の属性は次のとおりです。

- location: リソースの取得時に、外部（サード・パーティなど）の JNDI コンテキストを検索するための名前を指定します。

<context-attribute ... >

<lookup-context> のサブ要素（<resource-ref-mapping> のサブ要素）は、サード・パーティなどの外部の JNDI コンテキストに送信する属性を指定します。

JNDI の中で唯一の必須属性は java.naming.factory.initial で、これはコンテキスト・ファクトリの実装のクラス名です。

<context-attribute> の属性は次のとおりです。

- name: 属性の名前を指定します。
- value: 属性の値を指定します。

<env-entry-mapping ... >*deploymentValue*</env-entry-mapping>

orion-web.xml 内で、web.xml ファイルの env-entry 設定の値をオーバーライドします。EAR の構成にデプロイ固有の値を直接反映しないようにするために使用します。タグのボディに新しい値を指定します。

<env-entry-mapping> の属性は次のとおりです。

- name: オーバーライドする env-entry 設定の名前を指定します。

<security-role-mapping ... >

特定のユーザーとグループ、または全ユーザーに、セキュリティ・ロールをマップします。web.xml ファイル内の同じ名前のセキュリティ・ロールにマップします。impliesAll 属性または適切なサブ要素の組合せ（<group> または <user>、あるいはその両方）を使用する必要があります。

OC4J 構成ファイルの <security-role-mapping> 要素に関する追加情報は、『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』を参照してください。

<security-role-mapping> のサブ要素は次のとおりです。

<group>
<user>

<security-role-mapping> の属性は次のとおりです。

- impliesAll: このマッピングに全ユーザーを含めるかどうかを指定します。サポートされている値は、true または false（デフォルト）です。
- name: セキュリティ・ロールの名前を指定します。web.xml の <security-role> 要素の <role-name> サブ要素に指定されている名前と一致させてください。

重要： OC4J には自動セキュリティ・マッピング機能が備わっています。デフォルトでは、web.xml に定義されたセキュリティ・ロールが、jazn-data.xml に定義された OC4J グループ（または他の有効なユーザー・マネージャ）と同じ名前である場合、OC4J はそのセキュリティ・ロールをマップします。ただし、この機能は、<security-role-mapping> 要素を使用して、明示的なマッピングを行う場合は、完全に無効化されます。<security-role-mapping> を使用すると、OC4J では、明示的なマッピングのみが必要であるとみなされます。これは、ユーザーが明示的なマッピングを宣言しているときに、暗黙的なマッピングが実行されるのを防ぐためです。

<group ... >

<security-role-mapping> のサブ要素を使用して、親の <security-role-mapping> 要素のセキュリティ・ロールにマップするグループを指定します。指定されたグループの全メンバーがこのロールに含まれます。

<group> の属性は次のとおりです。

- name: グループの名前を指定します。

<user ... >

<security-role-mapping> のサブ要素を使用して、親の <security-role-mapping> 要素のセキュリティ・ロールにマップするユーザーを指定します。

<user> の属性は次のとおりです。

- name: ユーザーの名前を指定します。

<ejb-ref-mapping ... >

デプロイ時に、<ejb-ref> 要素に定義されている EJB 参照と JNDI の場所との間のマッピングを作成します。

<ejb-ref> 要素は、orion-web.xml または web.xml の <web-app> 要素内に置くことができ、EJB への参照を宣言するために使用します。

<ejb-ref-mapping> の属性は次のとおりです。

- location: EJB ホームのルックアップを行う JNDI の場所を指定します。
- name: EJB の参照名を指定します。<ejb-ref> 要素の <ejb-ref-name> 設定と一致します。

<expiration-setting ... >

指定されたリソースのセットに期限を設定します。これは、ブラウザでリソースが期限切れになるまでの期間です。(ブラウザでは、次回のリクエストに応じて期限切れのリソースを再ロードします。) これは、ドキュメントほど頻繁にイメージの再ロードを行わないなどのキャッシュ・ポリシーに役立ちます。

<expiration-setting> の属性は次のとおりです。

- expires: 期限切れまでの時間（秒単位）を指定します。期限切れを指定しない場合は never にします。即時期限切れのデフォルトの設定は 0（ゼロ）です。
- url-pattern: 期限切れが適用される URL パターンを指定します。次に例を示します。

  ```
  url-pattern="*.gif"
  ```

<jazn-web-app ... >

この要素を使用して、OracleAS JAAS Provider および Single Sign-On（SSO）のプロパティをサーブレットの実行用に構成します。特定のセキュリティ・サブジェクトの権限を使用してサーブレットを起動するには、これらの機能を適切に設定する必要があります。

<jazn-web-app> の属性は次のとおりです。

- auth-method: サポートされている値は、SSO（HTTP クライアント認証に Oracle Application Server Single Sign-On を使用）と、DIGEST（ダイジェスト認証メカニズムを使用）です。これらの設定は、orion-application.xml ファイルの <jazn-web-app> でもサポートされています。
- runas-mode: runas-mode を true に設定し、特定のサブジェクトの権限を使用してサーブレットを起動します。サブジェクトは javax.security.auth.Subject クラスのインスタンスで定義され、個人などの単一エンティティに関する一連のファクトが含まれます。ファクトには、パスワードや暗号鍵などの、認証およびセキュリティに関連する属性が含まれます。

  デフォルトの runas-mode="false" 設定では、doasprivileged-mode は無視されます。

- doasprivileged-mode: runas-mode="true" の場合は、doasprivileged-mode のデフォルトの true 設定を使用して、サーバーのアクセス制御の制限に縛られずに特定のサブジェクトの権限を使用します。

  サーブレットが起動されると、runas-mode="true" および doasprivileged-mode="true" の値から、静的な Subject.doAsPrivileged() メソッドが使用されます。runas-mode="true" および doasprivileged-mode="false" の値から、静的な Subject.doAs() メソッドが使用されます。いずれの場合も、JAAS Provider ではメソッドのコールにおける Subject インスタンスに渡します。

doAsPrivileged() メソッドが使用されると、JAAS Provider では NULL の java.security.AccessControlContext インスタンスを使用してメソッドを起動します。これは、現行サーバーの AccessControlContext インスタンスに制限されずに新規の処理を開始し、サーブレットを実行するためです。doAs() メソッドが使用されると、AccessControlContext インスタンスが現行スレッド（サーバー）から取得されます。

JAAS およびこの要素に関して説明した機能の追加情報は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。次のサイトで Sun 社のドキュメントも参照できます。

http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/JAASRefGuide.html

<web-app-class-loader ... >

この要素は、クラスロード指示に使用します。追加情報は、2-9 ページの「OC4J におけるシステム・クラスより前の WAR ファイル・クラスのロード」を参照してください。

<web-app-class-loader> の属性は次のとおりです。

- search-local-classes-first: この属性を true に設定すると、システム・クラスの前に WAR ファイルのクラスが検索され、ロードされます。デフォルトの設定は、false です。
- include-war-manifest-class-path: この属性を false に設定すると、search-local-classes-first の設定に関係なく、WAR ファイル・クラスの検索時とロード時に、WAR ファイルのマニフェストの Class-Path 属性に指定された CLASSPATH は含まれません。デフォルトの設定は、true です。

注意 :

- 両方の属性を true に設定すると、WAR ファイル内に物理的に常駐するクラスが、WAR ファイルのマニフェストの CLASSPATH のクラスより前にロードされるよう、全体の CLASSPATH が構成されます。したがって、競合が発生した場合、WAR ファイル内に物理的に常駐するクラスが優先されます。
- サーブレット仕様により、search-local-classes-first 機能は、java.* または javax.* パッケージでのクラスのロードには使用できません。

<authenticate-on-dispatch ... >

この要素は、転送またはインクルード・ターゲットの OC4J 認証の無効化に使用します。

<authenticate-on-dispatch> の属性は次のとおりです。

- value: この属性を false に設定すると、転送またはインクルード・ターゲットの認証が無効化されます。これは、サーブレット仕様に準拠しています。デフォルト値は true で、以前のバージョンの OC4J に対して開発されたアプリケーションでセキュリティ違反が発生するのを防ぎます。

<web-app ... >

この要素は、標準的な web.xml ファイルなどで使用します。詳細は、サーブレット仕様を参照してください。global-web-application.xml 内で、<web-app> 設定のデフォルトを設定できます。web.xml では、アプリケーション固有の <web-app> 設定によって、デフォルトをオーバーライドできます。orion-web.xml では、デプロイ固有の <web-app> 設定によって、web.xml の設定をオーバーライドできます。

global-web-application.xml および orion-web.xml の DTD

この項では、OC4J 10.1.2 実装の global-web-application.xml および orion-web.xml ファイルに使用する、DTD の OC4J 固有部分について説明します。ここには、web.xml ファイルの標準の <web-app> 要素に使用する DTD 部分は含まれません。

(global-web-application.xml および orion-web.xml の DTD は標準 web.xml の DTD のスーパーセットです。)

```
<!ENTITY % CHARSET "CDATA">

<!ENTITY % WEBPATH "CDATA">

<!ENTITY % NUMBER "CDATA">

<!ENTITY % HOST "CDATA">

<!ENTITY % PATH "CDATA">

<!ENTITY % CLASSNAME "CDATA">

<!-- A group that this security-role-mapping implies. Ie all the members of the
 specified group are included in this role. -->
<!ELEMENT group (#PCDATA)>
<!ATTLIST group name CDATA #IMPLIED
>

<!-- An attribute sent to the context. The only mandatory attribute in JNDI is
 the 'java.naming.factory.initial' which is the classname of the context factory
 implementation. -->
<!ELEMENT context-attribute (#PCDATA)>
<!ATTLIST context-attribute name CDATA #IMPLIED
value CDATA #IMPLIED
>

<!-- Defines the relative/absolute path to a file containing mime-mappings to
 use. -->
<!ELEMENT mime-mappings (#PCDATA)>
<!ATTLIST mime-mappings path CDATA #IMPLIED
>

<!-- Specifies a codebase where classes used by this application (such as
 servlets/beans) can be found. -->
<!ELEMENT classpath (#PCDATA)>
<!ATTLIST classpath path CDATA #REQUIRED
>

<!-- The specification of an optional javax.naming.Context implementation used
 for retrieving the resource. This is useful when hooking up with 3rd party
 modules, such as a 3rd party JMS server for instance. Either use the context
 implementation supplied by the resource vendor or if none exists write an
 implementation which in turn negotiates with the vendor software. -->
<!ELEMENT lookup-context (context-attribute+)>
<!ATTLIST lookup-context location CDATA #IMPLIED
>

<!-- Specifies a servlet to use as request-tracker; request-trackers are invoked
 for every request and are useful for logging purposes, for example -->
<!ELEMENT request-tracker (#PCDATA)>
<!ATTLIST request-tracker servlet-name CDATA #IMPLIED
>

<!-- The resource-ref element is used for the declaration of a reference to
 an external resource such as a datasource, JMS queue, mail session or similar.
 The resource-ref-mapping ties this to a JNDI-location when deploying. -->
<!ELEMENT resource-ref-mapping (lookup-context?)>
<!ATTLIST resource-ref-mapping location CDATA #IMPLIED
name CDATA #REQUIRED
>
```

```
<!-- Tag that is defined if the application is to be clustered. Clustered
 applications have their ServletContext and session data
 shared between the apps in the cluster, the values have to be either
 Serializable or be remote RMI-objects (implement java.rmi.Remote). -->
<!ELEMENT cluster-config (#PCDATA)>
<!ATTLIST cluster-config host %HOST; "230.0.0.1"
id CDATA "based on local IP"
port %NUMBER; "9127"
>

<!-- Specifies an optional access-mask for this application, hostnames and
 ip/subnets can be used to filter out allowed clients of this application. -->
<!ELEMENT access-mask (host-access*, ip-access*)>
<!ATTLIST access-mask default (allow|deny) "allow"
>

<!-- Overrides the value of an env-entry in the assembly descriptor. It is used
 to keep the .ear (assembly) clean from deployment-specific values. The body is
 the value. -->
<!ELEMENT env-entry-mapping (#PCDATA)>
<!ATTLIST env-entry-mapping name CDATA #IMPLIED
>

<!-- Specifies the Expires setting for a given set of resources, useful for
 caching policies (for instance for browsers not to reload images as frequently
 as documents). -->
<!ELEMENT expiration-setting (#PCDATA)>
<!ATTLIST expiration-setting expires CDATA #IMPLIED
url-pattern CDATA #IMPLIED
>

<!-- Overrides the value of a context-param in the assembly descriptor. It is
 used to keep the .ear (assembly) clean from deployment-specific values. The
 body is the value. -->
<!ELEMENT context-param-mapping (#PCDATA)>
<!ATTLIST context-param-mapping name CDATA #IMPLIED
>

<!-- Session-tracking settings for this application. -->
<!ELEMENT session-tracking (session-tracker*)>
<!ATTLIST session-tracking autoencode-absolute-urls (true|false) "false"
autoencode-urls (true|false) "true"
autojoin-session (true|false) "false"
cookie-domain CDATA #IMPLIED
cookie-max-age %NUMBER; "in memory only"
cookies (enabled|disabled) "enabled"
>

<!-- A user that this security-role-mapping implies. -->
<!ELEMENT user (#PCDATA)>
<!ATTLIST user name CDATA #IMPLIED
>

<!-- Adds a virtual directory mapping, used to include files that doesnt
 physically reside below the document root among the web-exposed files. -->
<!ELEMENT virtual-directory (#PCDATA)>
<!ATTLIST virtual-directory real-path %PATH; #IMPLIED
virtual-path %PATH; #IMPLIED
>

<!-- Specifies an ip/netmask who is allowed access. -->
<!ELEMENT ip-access (#PCDATA)>
```

```
<!ATTLIST ip-access ip CDATA #REQUIRED
mode (allow|deny) #REQUIRED
netmask CDATA #IMPLIED
>

<!-- Specifies a servlet to use as chainer for a specified mime-type. Useful to
 filter/transform certain kinds of output. -->
<!ELEMENT servlet-chaining (#PCDATA)>
<!ATTLIST servlet-chaining mime-type CDATA #IMPLIED
servlet-name CDATA #IMPLIED
>

<!-- Specifies a domain or netmask who is allowed access. -->
<!ELEMENT host-access (#PCDATA)>
<!ATTLIST host-access domain CDATA #REQUIRED
mode (allow|deny) #REQUIRED
>

<!-- The ejb-ref element is used for the declaration of a reference to
 another enterprise bean's home. The ejb-ref-mapping ties this to JNDI-location
 when deploying. -->
<!ELEMENT ejb-ref-mapping (#PCDATA)>
<!ATTLIST ejb-ref-mapping location CDATA #IMPLIED
name CDATA #REQUIRED
>

<!-- The runtime mapping (to groups and users) of a role. Maps to a
 security-role of the same name in the assembly descriptor. -->
<!ELEMENT security-role-mapping (group*, user*)>
<!ATTLIST security-role-mapping impliesAll CDATA #IMPLIED
name CDATA #IMPLIED
>

<!-- Specifies a servlet to use as session-tracker; session-trackers are invoked
 as soon as a session is created and are useful for logging purposes, for
 example -->
<!ELEMENT session-tracker (#PCDATA)>
<!ATTLIST session-tracker servlet-name CDATA #IMPLIED
>

<!-- JAZN configuration -->
<!ELEMENT jazn-web-app (#PCDATA)>
<!ATTLIST jazn-web-app auth-method CDATA #IMPLIED
runas-mode (true | false) "false"
doasprivileged-mode (true | false) "true"
>

<!-- Web-app classloader configuration -->
<!ELEMENT web-app-class-loader EMPTY>
<!ATTLIST web-app-class-loader
search-local-classes-first (true | false) "false"
include-war-manifest-class-path (true | false) "true"
>

<!-- Authentication of forward/include targets -->
<!ELEMENT authenticate-on-dispatch EMPTY>
<!ATTLIST authenticate-on-dispatch
value (true | false) "true"
>

<!-- This file contains the orion-specific configuration for a web-application.
 The path to the file is located at
 ORION_HOME/application-deployments/deploymentName/warname(.war)/orion-web.xml
```

```
 or (web-app-root/)WEB-INF/orion-web.xml if no deployment-directory is specified
 in server.xml. -->
<!ELEMENT orion-web-app ( classpath*, context-param-mapping*, mime-mappings*,
  virtual-directory*, access-mask?, cluster-config?, servlet-chaining*,
  request-tracker*, session-tracking?, resource-ref-mapping*,
  security-role-mapping*, env-entry-mapping*, ejb-ref-mapping*,
  expiration-setting*, web-app?, jazn-web-app?, web-app-class-loader?,
  authenticate-on-dispatch? )>
<!ATTLIST orion-web-app autoreload-jsp-beans (true|false) "true"
autoreload-jsp-pages (true|false) "true"
default-buffer-size CDATA "2048"
default-charset %CHARSET; "iso-8859-1"
deployment-version CDATA #IMPLIED
development (true|false) "false"
directory-browsing (allow|deny) "deny"
file-modification-check-interval %NUMBER; "1000"
jsp-cache-directory CDATA #IMPLIED
jsp-cache-tlds (true|on|standard|false|off) "true"
jsp-taglib-locations CDATA #IMPLIED
jsp-print-null (true|false) "true"
jsp-timeout %NUMBER; "0 (never)"
simple-jsp-mapping (true|false) "false"
enable-jsp-dispatcher-shortcut (true|false) "true"
persistence-path CDATA #IMPLIED
servlet-webdir %PATH; "/servlet/"
source-directory CDATA #IMPLIED
temporary-directory CDATA #IMPLIED
>
```

global-web-application.xml および orion-web.xml の階層表現

この項では、global-web-application.xml および orion-web.xml ファイルの階層表現について説明します。

注意： わかりやすく説明するために、終了タグは省略されています。

```
<orion-web-app default-buffer-size="..." default-charset="..."
               deployment-version="..." development="..."
               source-directory="..." directory-browsing="..."
               file-modification-check-interval="..."
               jsp-print-null="..." jsp-timeout="..." jsp-cache-directory="..."
               jsp-cache-tlds="..." jsp-taglib-locations="..."
               simple-jsp-mapping="..." enable-jsp-dispatcher-shortcut="..."
               persistence-path="..." servlet-webdir="..."
               temporary-directory="...">
     <classpath path="...">
     <context-param-mapping name="...">
     <mime-mappings path="...">
     <virtual-directory real-path="..." virtual-path="...">
     <access-mask default="...">
          <host-access domain="..." mode="...">
          <ip-access ip="..." netmask="..." mode="...">
     <cluster-config host="..." id="..." port="...">
     <servlet-chaining mime-type="..." servlet-name="...">
     <request-tracker servlet-name="...">
     <session-tracking autojoin-session="..." cookies="..."
                       cookie-domain="..." cookie-max-age="...">
          <session-tracker servlet-name="...">
     <resource-ref-mapping location="..." name="...">
          <lookup-context location="...">
               <context-attribute name="..." value="...">
```

```
<env-entry-mapping name="...">
<security-role-mapping impliesAll="..." name="...">
     <group name="...">
     <user name="...">
<ejb-ref-mapping location="..." name="...">
<expiration-setting expires="..." url-pattern="...">
<jazn-web-app auth-method="..." runas-mode="..."
              doasprivileged-mode="...">
<web-app-class-loader search-local-classes-first="..."
                      include-war-manifest-class-path="...">
<authenticate-on-dispatch value="...">
<web-app>   AS IN STANDARD WEB.XML
```

サンプルの global-web-application.xml の設定

これは、デフォルトの global-web-application.xml ファイルの簡単な例です。<orion-web-app> 属性の設定の一部、MIME マッピングの設定、および JSP と RMI のフロントエンド・サーブレットの設定とマッピングを次に示します（どの製品も出荷時に変更されることがあります)。

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE orion-web-app PUBLIC '//Evermind//Orion web-application'
 'http://xmlns.oracle.com/ias/dtds/orion-web.dtd'>

<orion-web-app
   jsp-cache-directory="./persistence"
   servlet-webdir="/servlet"
   development="false"
   jsp-timeout="0"
   jsp-taglib-locations="./jsp/lib/taglib"
>

   <!-- The mime-mappings for this server -->
   <mime-mappings path="./mime.types" />

   <web-app>

      <servlet>
         <servlet-name>jsp</servlet-name>
         <servlet-class>oracle.jsp.runtimev2.JspServlet</servlet-class>
         <load-on-startup>0</load-on-startup>
         <!-- you can disable page scope listener if you
              don't need this function. -->
         <init-param>
            <param-name>check_page_scope</param-name>
            <param-value>true</param-value>
         </init-param>
         <!-- you can set main_mode to "justrun" to speed up
              JSP dispatching, if you don't need to recompile
              your JSP anymore. You can always switch your
              main_mode. Please see our doc for details -->
         <!--
         <init-param>
            <param-name>main_mode</param-name>
            <param-value>justrun</param-value>
         </init-param>
         -->
      </servlet>

      <servlet-mapping>
         <servlet-name>jsp</servlet-name>
         <url-pattern>/*.jsp</url-pattern>
      </servlet-mapping>
```

```
    <servlet-mapping>
       <servlet-name>jsp</servlet-name>
       <url-pattern>/*.JSP</url-pattern>
    </servlet-mapping>

    <servlet>
       <servlet-name>rmi</servlet-name>
       <servlet-class>
          com.evermind.server.rmi.RMIHttpTunnelServlet
       </servlet-class>
    </servlet>
    <servlet-mapping>
       <servlet-name>rmi</servlet-name>
       <url-pattern>/*.tunnelrmi</url-pattern>
    </servlet-mapping>

 </web-app>
</orion-web-app>
```

Web サイトの XML ファイルの構成

次の項では、Oracle Application Server 環境用の `default-web-site.xml`、および OC4J スタンドアロン環境用の `http-web-site.xml` を含む、Web サイトの XML 構成ファイルの詳細を説明します。

- Web サイト XML ファイルの要素の説明
- Web サイトの XML ファイルの DTD
- Web サイトの XML ファイルの階層表現
- サンプルの default-web-site.xml ファイル

これらのファイルの概要は、5-18 ページの「OC4J Web サイト・ディスクリプタ」を参照してください。

Web サイト XML ファイルの要素の説明

この項の要素の説明は、`default-web-site.xml`（Oracle Application Server）、`http-web-site.xml`（OC4J スタンドアロン）および任意の OC4J Web サイトの XML ファイルに適用されます。

<web-site ... >

OC4J Web サイトを構成するルート要素です。

`<web-site>` のサブ要素は次のとおりです。

```
<description>
<frontend>
<web-app>
<default-web-app>
<user-web-apps>
<access-log>
<odl-access-log>
<ssl-config>
```

`<web-site>` の属性は次のとおりです。

- `cluster-island`: クラスタ・アイランドは、レプリケーションに対してセッション・フェイルオーバー時の状態を共有する、2 つ以上の Web サーバーです。Oracle Application Server の複数の OC4J インスタンス間で Web 層をクラスタリングする場合は、`cluster-island` 属性を使用します。この属性をクラスタ・アイランドの ID（1 から始まる数値）に設定すると、この Web サイトは、ID で指定されたアイランドのバックエンド・サーバーとして加わります。この ID は、使用しているクラスタ構成に基づいて選択された数値です。使用するアイランドが 1 つのみの場合、ID は常に 1 です。

クラスタリングに関する一般的な情報は、『Oracle Application Server パフォーマンス・ガイド』を参照してください。

- display-name: この属性を使用して、ユーザーにわかりやすい非公式の Web サイト名を指定できます。
- host: この Web サイトに、ホストを DNS ホスト名または IP アドレスのいずれかを指定します。サーバーがマルチ・ホーム・マシン（複数の IP アドレスを持つ）の場合、[ALL] 設定を使用してすべての IP アドレスをリスニングできます。これらはすべてこの単一の Web サイトに統合されます。
- log-request-info: エラーの発生時に、受信リクエストに関する情報を Web サイトのログに書き込むかどうかを指定します。サポートされている値は、true および false（デフォルト）です。Web サイトのログは、<access-log> または <odl-access-log> 要素を使用して、有効にします。これについては、この項の後半で説明します。（Web サイトのログを始めとする、ログの有効化に関する追加情報は、2-14 ページの「OC4J のロギング」を参照してください。）
- max-request-size: 受信リクエストの最大サイズを、バイト単位で設定します。この最大値を超えるリクエストをクライアントが送信すると、クライアントに「リクエスト・エンティティが大きすぎます」というエラーが返されます。デフォルトの最大値は、15000 です。
- secure: Secure Sockets Layer（SSL）機能をサポートするかどうかを指定します。サポートされている値は、true および false（デフォルト）です。ajp13 のプロトコル設定（Oracle Application Server 環境で使用）を true に設定すると、Oracle HTTP Server と OC4J 間のセキュアな AJP プロトコルとなります。http のプロトコル設定（OC4J スタンドアロンで使用）を true に設定すると、クライアントと OC4J 間の HTTPS プロトコルとなります。

  また、secure="true" 設定では、<ssl-config> 要素（<web-site> 要素の下のサブ要素）を使用して、キーストア・パスおよびパスワードを指定する必要があることに注意してください。この要素については、この項の後半で説明します。

  注意： SSL と HTTPS の機能は、Oracle HTTP Server 経由でも使用できます。これは、Oracle HTTP Server およびクライアント間の通信に使用します。詳細は、『Oracle Application Server セキュリティ・ガイド』を参照してください。

- protocol: Web サイトが使用しているプロトコルを指定します。可能な値は、http および ajp13（AJP ではデフォルト）です。Oracle Application Server の本番環境では、ajp13 設定のみを使用してください。AJP は、Oracle HTTP Server と mod_oc4j とともに使用します。各プロトコルには対応するポートが、各ポートには対応するプロトコルが必要です。

  http 設定は OC4J スタンドアロンで使用されます。

  ajp13 または http 設定のいずれかをセキュア・モード（SSL）で使用するには、secure フラグを true に設定し、<ssl-config> サブ要素を使用してキーストア・パスおよびパスワードを指定する必要があります。この要素については、この項の後半で説明します。

- port: この Web サイトのポート番号を指定します。各ポートには対応するプロトコルが、各プロトコルには対応するポートが必要です。OC4J スタンドアロンでは、デフォルトで port 設定 8888 を使用して、OC4J リスナーに直接アクセスします。これは、必要に応じて変更できます。

  Oracle Application Server 環境では、このポート設定は Oracle Process Management and Notification（OPMN）システムでオーバーライドされます。Oracle Application Server では、ポート 7777 がデフォルトで使用され、Oracle Application Server Web Cache が有効な Oracle HTTP Server を経由してアクセスします。

重要： UNIX 環境では、1024 未満のポート番号へのアクセスには root 権限が必要です。また、クライアント・ブラウザからポートの指定がない場合、ポート 80 は HTTP プロトコル用であり、ポート 443 は HTTPS 用となることに注意してください。

- `use-keep-alives`: サーブレット・コンテナの一般的な動作は、リクエストが完了した後に接続をクローズすることです。ただし、`use-keep-alives` 設定を `true` にすることにより、リクエスト間で接続が維持されます。AJP プロトコルの場合は、常に接続が維持され、この属性は無視されます。その他のプロトコルのデフォルトは `true` で、これを無効化するとパフォーマンスが大幅に低下します。
- `virtual-hosts`: このオプションの設定は、同じ IP アドレスを共有する仮想サイトに役立ちます。値は、この Web サイトに関連付けられているホスト名の、カンマ区切りのリストです。

<description>*Description here.*</description>

この要素のボディを使用して、Web サイトの簡単な説明を記述できます。

<frontend ... >

HTTP クライアントにより表示される Web サイトの、認識可能なフロントエンド・ホストおよびポートを指定します。サイトがロード・バランサまたはファイアウォールの後ろにある場合、URL リライティングなどの機能に対する Web アプリケーション・コードに適切な情報を与えるには、必ず `<frontend>` を指定する必要があります。アプリケーションを実行中のバックエンド・サーバーは、`<frontend>` 要素に指定されているホストとポートを使用し、URL リライティングではサーバー自体ではなく、参照先のフロントエンドを認識します。したがって、後続のリクエストは、バックエンドに直接アクセスすることなく、フロントエンドから適切に受信されます。

`<frontend>` の属性は次のとおりです。

- `host`: `www.acme.com` など、フロントエンド・サーバーのホスト名を指定します。
- `port`: `80` など、フロントエンド・サーバーのポート番号を指定します。

<web-app ... >

この要素は、特定の Web モジュールをこの Web サイトにバインドします。また、`server.xml` ファイルから J2EE アプリケーションのアーカイブの名前（`.ear` 拡張子がない EAR ファイル名）、および J2EE アプリケーション内の Web モジュールの名前を指定します。Web モジュールは、アプリケーションの EAR ファイル（または、EAR ファイルの `orion-application.xml` ファイル）にある、J2EE の `application.xml` ファイルで定義されます。Web モジュールは、`<web-app>` 要素の `root` 属性により指定された場所にバインドされます。

注意： EAR ファイル内の WAR ファイルではなく、WAR ファイル単独でデプロイできます。OC4J スタンドアロンでは、このような Web アプリケーションは OC4J のデフォルトのアプリケーションに追加されます。（OC4J では、常になんらかの親アプリケーションが存在する必要があります。）詳細は、5-24 ページの「OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーション」を参照してください。

この使用例では、Web サイトの XML ファイルの `<web-app>` 要素は、J2EE アプリケーションのアーカイブ名ではなくデフォルトのアプリケーション名を指定します。詳細は、次の属性の説明および例に示します。

特に application および name 属性に関する Web サイトの XML ファイルへのマッピングと Web サイトの XML ファイルからのマッピングについては、このマニュアルの他の章で例を示します。5-19 ページの「例 : Web サイト・ディスクリプタへのマッピングと Web サイト・ディスクリプタからのマッピング」(EAR ファイル内の WAR ファイルをデプロイする典型的な手順)、および 5-31 ページの「OC4J スタンドアロンへの独立した WAR ファイルのデプロイ」(WAR ファイル単独で OC4J のデフォルトのアプリケーションにデプロイする手順)を参照してください。

<web-app> の属性は次のとおりです。

- access-log: <access-log> 要素または <odl-access-log> 要素をオーバーライドし、この <web-app> 要素に関連付けられた Web アプリケーションのアクセス・ロギングのみを無効にするには、access-log="false" を設定します。デフォルトの設定は、true です(Web サイトの XML ファイルに <access-log> 要素または <odl-access-log> 要素がない場合は、アクセス・ロギングはすでに無効になっており、access-log 属性は影響しません)。
- application: J2EE アプリケーションのアーカイブ名を指定します。これは、.ear 拡張子がない EAR ファイル名であり、server.xml ファイルの <application> 要素の name 属性に対応します。

> **注意：** OC4J のデフォルトのアプリケーションを親として使用し、OC4J スタンドアロンに WAR ファイルを単独でデプロイする場合、application 属性は server.xml ファイルの <global-application> 要素に基づいて、デフォルトのアプリケーション名をかわりに反映します。

- load-on-startup: オプションの属性。この Web モジュールを、アプリケーションの起動時に事前にロードするかどうかを指定します。指定しない場合は、Web アプリケーションが最初にリクエストされた時点でロードされます。サポートされている値は、true および false(デフォルト)です。

  アプリケーションの web.xml ファイル内の <load-on-startup> 要素を使用した個別サーブレットの事前ロードを実行できるのは、この <web-app> 要素の load-on-startup 属性が有効な場合のみです。詳細は、2-6 ページの「サーブレットの事前ロード」を参照してください。
- max-inactivity-time: Web モジュールを非アクティブにしておく時間(分)を指定するための、オプションの整数属性です。この時間を経過すると、OC4J は Web モジュールをシャットダウンします。デフォルトでは、非アクティブが原因で Web モジュールがシャットダウンされることはありません。
- name: 特定の J2EE アプリケーション内の Web モジュール名を指定します。J2EE の application.xml ファイルにおける、<module> 要素の <web> サブ要素の <web-uri> 値(.war 拡張子がない)に対応します。J2EE の application.xml ファイルは EAR ファイルにあります。

注意：

- OC4J のデフォルトのアプリケーションを親として使用し、OC4J スタンドアロンに WAR ファイルを単独でデプロイする場合、name 属性は、OC4J グローバル application.xml ファイルにおける、<web-module> 要素の id 属性の値をかわりに反映します。これは OC4J のデフォルト・アプリケーションの application.xml ファイルですが、標準の J2EE ファイルではなく、OC4J 固有であることに注意してください。また、id 属性には、<web-app> 要素の name 属性と同様、.war 拡張子がありません。
- アプリケーションには、<web-module> 要素を含む、EAR ファイルの orion-application.xml ファイルもあります。この要素は、追加 Web モジュールを定義し、J2EE の application.xml ファイルで定義された Web モジュールのオーバーライドもします（ただし、オーバーライドはお薦めしません）。name 属性は、J2EE の application.xml ファイルの <web-uri> 値ではなく、orion-application.xml の <web-module> 要素の id 値を反映できます。
- orion-application.xml ファイルでは、OC4J グローバル application.xml ファイルと同じ DTD、すなわち、orion-application.dtd を使用します。

- root: Web モジュールをバインドするパスを指定します。モジュールの起動に使用される URL のコンテキスト・パス部分を定義します。たとえば、Web サイト www.example.com の Web モジュール CatalogApp が、root 設定の /catalog にバインドされる場合は、次のように起動できます。

  ```
  http://www.example.com/catalog
  ```

  重要：

  - root 属性は、J2EE の application.xml ファイルの対応する <web> 要素の <context-root> 値をオーバーライドします。<context-root> 要素は application.xml ファイルでは必須ですが、その値は OC4J では使用されません。
  - root 設定に「/」を指定すると、OC4J のデフォルトの Web アプリケーションをオーバーライドします。Web アプリケーションを Web サイトにバインドする場合、この設定または NULL 設定は、admin.jar ユーティリティで許可されません。

- shared: Web サイト間で公開された Web モジュールの共有を許可します。ここでは、Web サイトはプロトコルおよびポートの特定の組合せで定義されます。サポートされている値は、true および false（デフォルト）です。共有すると、セッション、サーブレット・インスタンスおよびコンテキスト値など、Web アプリケーションを構成するすべてのものが共有されることを意味します。たとえば、SSL がすべての通信ではなく一部の通信に必須の場合、同一のコンテキスト・パスにおいて、HTTP サイトおよび HTTPS サイト間で OC4J スタンドアロンの Web アプリケーションを共有する場合などです。（すべての情報ではなく、機密に関わる情報のみを暗号化することで、パフォーマンスが向上します。）

  HTTPS の Web アプリケーションに共有のマークが付いている場合、そのセッション・トラッキングの方針が、SSL セッション・トラッキングから、Cookie や URL リライティングのセッション・トラッキングに戻ります。これにより Web アプリケーションの保護が低下しますが、一部のブラウザでは適切にサポートされていない、SSL セッション・タイムアウトなどの問題に対処するためには必須です。

重要： shared="true" は、OC4J スタンドアロンのみで使用してください。

<default-web-app ... >

この Web サイトのデフォルトの Web アプリケーションに対する参照を作成します。ユーザーにとって、これは OC4J スタンドアロン環境でのみ役立ちます。詳細は、5-24 ページの「OC4J のデフォルトのアプリケーションおよびデフォルトの Web アプリケーション」を参照してください。

Oracle Application Server 環境では、OC4J のデフォルトの Web アプリケーションにはシステム・レベルの機能がありますが、この機能は、他には役立ちません。5-39 ページの「Oracle Application Server での OC4J のデフォルト Web アプリケーション」を参照してください。

`<default-web-app>` 要素では、前述の `<web-app>` 要素と同じ属性を使用しますが、`load-on-startup` のデフォルトの設定が `true` であることに注意してください。

<user-web-apps ... >

この要素は、ユーザー・ディレクトリとアプリケーションをサポートするために使用します。各ユーザーには、自分の Web モジュールおよび関連する `web-application.xml` ファイルがあります。ユーザー・アプリケーションは、サーバー・ルートの `/`*`username`*`/` に置かれます。

`<user-web-apps>` の属性は次のとおりです。

- `max-inactivity-time`: Web モジュールを非アクティブにしておく時間（分）を指定するための、オプションの整数属性。この時間を経過すると、OC4J は Web モジュールをシャットダウンします。デフォルトでは、非アクティブが原因で Web モジュールがシャットダウンされることはありません。
- `path`: ユーザー・アプリケーションのローカル・ディレクトリを指定するためのパスを指定します（ユーザー名にワイルド・カードも使用可能）。たとえば、UNIX でのデフォルトのパス設定は `/home/`*`username`* であり、*`username`* は、特定のユーザー名で置換されます。

<access-log ... >

この要素を使用して、この Web サイトのテキストベースのアクセス・ロギングを有効にし、アクセス・ログに関する情報（パス、ファイル名および含まれる情報など）を指定します。ログ・ファイルは受信リクエスト（Web サイトの各アクセス）のログが記録される場所にあります。

かわりに、ODL ロギングには `<odl-access-log>` 要素（次に説明します）を使用します。ODL の詳細は、2-15 ページの「Oracle Diagnostic Logging とテキストベースのロギング」を参照してください。

注意：

- どの Web アプリケーションでも、Web サイトの XML ファイルでその Web アプリケーションの `<web-app>` 要素に `access-log="false"` と設定すれば、`<access-log>` 要素をオーバーライドし、アクセス・ロギングを無効にできます。
- `<access-log>` および `<odl-access-log>` の両方を使用しないでください。ロギングに使用できるタイプは 1 つのみです。（Web サイトの XML ファイルの最後の要素は優先されますが、この動作には頼らないでください）。

<access-log> の属性は次のとおりです。

- format: 複数のサポートされている（ログ・エントリに追加される情報となる）変数を 1 つ以上指定します。サポートされる変数は、$time、$request、$ip、$host、$path、$size、$method、$protocol、$user、$status、$referer、$time、$agent、$cookie、$header および $mime です。変数の間には、ログ・メッセージの値の間に表示する、任意のセパレータを入力できます。デフォルトの設定は次のとおりです。

  ```
  "$ip - $user - [$time] '$request' $status $size"
  ```

  たとえば、これは次のようなログ・メッセージになります（2 番目のメッセージは 2 行目に折り返されています)。

  ```
  148.87.1.180 - - [06/Nov/2001:10:23:18 -0800] 'GET / HTTP/1.1' 200 2929
  148.87.1.180 - - [06/Nov/2001:10:23:53 -0800] 'GET
  /webservices/statefulTest HTTP/1.1' 200 301
  ```

  この例では、ユーザーは NULL、時刻は大カッコの中（format 設定の指定どおり)、リクエストは一重引用符の中（指定どおり)、そして最初のメッセージの状態とサイズはそれぞれ、200 および 2929 です。

- path: アクセス・ログのパスおよび名前を指定します。これは、絶対パスまたは j2ee/home/config ディレクトリからの相対パスにできます。default-web-site.xml のデフォルトの設定は次のとおりです。

  ```
  path="../log/default-web-access.log"
  ```

  > **注意：** パスおよびファイル名を指定する <access-log> の path 属性と、パスのみを指定する <odl-access-log> の path 属性の違いに注意してください。（ODL ログ・ファイル名は修正されます。）

- split: 新規アクセス・ログの開始頻度を指定します。サポートされている値は、none（デフォルトではなし)、hour、day、week、または month です。none 以外の値の場合、suffix 属性に従ってログに名前が付けられます。

- suffix: ファイル分割を使用した場合、各ファイル名を一意にするためにログのベース・ファイル名（path 属性で指定されたとおりの名前）に追加するタイムスタンプ情報を指定します。使用される形式は java.text.SimpleDateFormat で、suffix 設定で使用される記号は、そのクラスの記号表記に従います。SimpleDateFormat と、そこで使用される書式記号の詳細は、次の場所で Sun 社の Javadoc を参照してください。

  http://java.sun.com/j2se/1.4.2/docs/api/

  デフォルトの suffix 設定は、-yyyy-MM-dd です。SimpleDateFormat ドキュメントに記載されているとおり、これらの文字には、大 / 小文字の区別があります。

  たとえば、次のような <access-log> 要素を考えてみます（デフォルトの suffix 値を使用)。

  ```
  <access-log path="c:¥foo¥web-site.log" split="day" />
  ```

  ログ・ファイルには、次の例のような名前が付けられます。

  ```
  c:¥foo¥web-site-2001-11-17.log
  ```

<odl-access-log ... >

この要素を使用して、Web サイトに対する ODL ベースのアクセス・ロギングを有効にし、アクセス・ログに関する情報（パス、各ファイルのサイズの最大値およびログ・ディレクトリ内の全ファイルの合計サイズを含む）を指定します。ログ・ファイルは受信リクエスト（Web サイトの各アクセス）のログが記録される場所にあります。

かわりに、テキスト・ベースのロギングには（前述の）<access-log> 要素を使用します。

ODL の詳細は、2-15 ページの「Oracle Diagnostic Logging とテキストベースのロギング」を参照してください。

注意：

- どの Web アプリケーションでも、Web サイトの XML ファイルでその Web アプリケーションの `<web-app>` 要素に `access-log="false"` と設定すれば、`<odl-access-log>` 要素をオーバーライドし、アクセス・ロギングを無効にできます。
- `<access-log>` および `<odl-access-log>` の両方を使用しないでください。ロギングに使用できるタイプは 1 つのみです。（Web サイトの XML ファイルの最後の要素は優先されますが、この動作には頼らないでください）。

`<odl-access-log>` の属性は次のとおりです。

- `path`: アクセス・ログ・ディレクトリへのパスを指定します。これは、絶対パスまたは `j2ee/home/config` ディレクトリからの相対パスにできます。次に例を示します。

  ```
  path="../log/default-web-access"
  ```

  このディレクトリの初期ログ・ファイル名は `log1.xml` です。最大ファイル・サイズ（`max-file-size` 属性により指定）に到達すると、後続のログ・ファイルには、`log2.xml`、`log3.xml` などの名前が付けられます。

  注意： パスおよびファイル名を指定する `<access-log>` の `path` 属性と、パスのみを指定する `<odl-access-log>` の `path` 属性の違いに注意してください。（ODL ログ・ファイル名は修正されます。）

- `max-file-size`: 各ログ・ファイルの最大サイズを、KB 単位で指定します。
- `max-directory-size`: `path` 属性で指定されるディレクトリにある、すべてのログ・ファイルの最大合計サイズを、KB 単位で指定します。

<ssl-config ... >

この要素は SSL 構成設定を、必要に応じて指定します。`<web-site>` 要素の `secure` 属性を `true` に設定する場合は、これを常に使用する必要があります。

関連情報については、2-32 ページの「サーブレットのセキュリティ」を参照してください。

`<ssl-config>` のサブ要素は次のとおりです。

```
<property>
```

`<ssl-config>` の属性は次のとおりです。

- `keystore`: このインストールでユーザー・ベースの証明書およびキーを格納するためのこの Web サイトで使用されるキーストア・データベース（バイナリ・ファイル）への相対または絶対パス。パスの値にはファイル名が含まれます。相対パスは、Web サイトの XML ファイルの位置に対して相対的です。

  キーストアは `java.security.KeyStore` インスタンスであり、Sun 社の JDK で提供される `keytool` ユーティリティを使用して作成および保持できます。

- `keystore-password`: キーストアをオープンする必須パスワード。
- `needs-client-auth`: Oracle HTTP Server などの OC4J のクライアントであるエンティティで、OC4J と通信するための認可の証明書を発行する必要があるかどうかを示します。サポートされている値は、`true`（クライアント認証 - 証明書が必須の場合）および `false`（デフォルト - 証明書が必須でない場合）です。

- `provider`: JSSE（Java Secure Socket Extension）を使用している場合、この属性を使用してプロバイダを指定できます。デフォルトでは、OC4J は通常 Sun 社の SSL 実装を使用し、プロバイダに対し次のインスタンスを使用します。

  ```
  com.sun.net.ssl.internal.ssl.Provider
  ```

  ただし、SOAP および `http_client` などの場合、オラクル社の SSL 実装も使用する場合があります。

- `factory`: JSSE を使用していない場合、`factory` 属性を使用して `SSLServerSocketFactory` の実装を指定します。デフォルトの設定は、次のとおりです。

  ```
  "JSSE: com.evermind.ssl.JSSESSLServerSocketFactory"
  ```

  サード・パーティの `SSLServerSocketFactory` 実装を使用する場合、`<ssl-config>` 要素の `<property>` サブ要素を使用して、パラメータをファクトリに送信できます。

<property ... >

`<ssl-config>` 要素の `<property>` サブ要素を使用して、必要に応じてサード・パーティの `SSLServerSocketFactory` 実装にパラメータを渡します。

`<property>` の属性は次のとおりです。

- `name`: ファクトリに渡すパラメータの名前。
- `value`: 指定したパラメータの値。

Web サイトの XML ファイルの DTD

この項では、OC4J 10.1.2 実装の `default-web-site.xml` および `http-web-site.xml` を含む、Web サイトの XML 構成ファイルの DTD について説明します。

```
<!ENTITY % WEBPATH "CDATA">

<!ENTITY % NUMBER "CDATA">

<!ENTITY % HOST "CDATA">

<!ENTITY % BOOLEAN "true|false">

<!ENTITY % PATH "CDATA">

<!-- When enabled user dirs/apps will be supported. Each user has his own
 private web-application (and connected web-application.xml file).
 The user apps are reached at /~username/ from the server root. -->
<!ELEMENT user-web-apps (#PCDATA)>
<!ATTLIST user-web-apps max-inactivity-time CDATA "no shutdown"
path %PATH; #IMPLIED
>

<!-- Reference to the default <a class="link"
 href="web.xml.html">web-application</a> of this site. This application will be
 bound to the root of the site. -->
<!ELEMENT default-web-app (#PCDATA)>
<!ATTLIST default-web-app application CDATA #IMPLIED
load-on-startup (true|false) "true"
max-inactivity-time %NUMBER; #IMPLIED
name CDATA #IMPLIED
root %WEBPATH; #IMPLIED
shared (true|false) "false"
>

<!-- A short description of this web-site. -->
```

```
<!ELEMENT description (#PCDATA)>

<!-- Relative/absolute path to the access-log for this site, this is where
 incoming requests will be logged. -->
<!ELEMENT access-log (#PCDATA)>
<!ATTLIST access-log format CDATA "$ip - $user - [$time] '$request' $status
 $size"
path CDATA #IMPLIED
split (none|hour|day|week|month) "none"
suffix CDATA #IMPLIED
>

<!-- An ODL formated log file. The max-file-size is the maximum number of
 kilobytes a single log file is allowed to grow to. The max-directory-size is
 the maximum number of kilobytes that the directory is allowed to contain. -->
<!ELEMENT odl-access-log (#PCDATA)>
<!ATTLIST odl-access-log path CDATA #REQUIRED max-file-size CDATA #IMPLIED
 max-directory-size CDATA #IMPLIED>

<!-- Reference to a <a class="link" href="web.xml.html">web-application</a>.
 This application will be bound at the location specified by the 'root'
 attribute. -->
<!ELEMENT web-app (#PCDATA)>
<!ATTLIST web-app application CDATA #IMPLIED
load-on-startup (true|false) "false"
access-log (true|false) "true"
max-inactivity-time %NUMBER; "no shutdown"
name CDATA #IMPLIED
root %WEBPATH; #IMPLIED
shared (true|false) "false"
>

<!-- A configuration parameter. -->
<!ELEMENT property (#PCDATA)>
<!ATTLIST property name CDATA #IMPLIED
value CDATA #IMPLIED
>

<!-- Specifies SSL-configuration settings. These settings are used if
secure="true" is specified on the site.
If a 3rd party SSLServerSocketFactory implementation is used then x property
tags can be defined to send arbitary arguments to the factory. -->
<!ELEMENT ssl-config (property*)>
<!ATTLIST ssl-config factory CDATA
 "com.evermind.server.JSSESSLServerSocketFactory"
keystore CDATA #IMPLIED
keystore-password CDATA #IMPLIED
needs-client-auth (true|false) "false"
provider CDATA #IMPLIED
>

<!-- The frontend tag describes which IP, port, and so on that HTTP clients
 perceive this site to be. This is needed when acting behind a load balancer or
 firewall in order to provide the correct info to web-app code when rewriting
 URLs -->
<!ELEMENT frontend (#PCDATA)>
<!ATTLIST frontend host CDATA #IMPLIED
port CDATA #IMPLIED
>

<!-- This file contains the configuration for a web-site. -->
<!ELEMENT web-site (description?, frontend?, default-web-app, web-app*,
 user-web-apps?, access-log?, odl-access-log?, ssl-config?)>
```

```
<!ATTLIST web-site cluster-island CDATA #IMPLIED
display-name CDATA #IMPLIED
protocol CDATA #IMPLIED
host %HOST; "[ALL]"
log-request-info (true|false) "false"
max-request-size CDATA #IMPLIED
port %NUMBER; "80"
secure (true|false) "false"
use-keep-alives CDATA #IMPLIED
virtual-hosts CDATA #IMPLIED
>
```

Web サイトの XML ファイルの階層表現

この項では、default-web-site.xml および http-web-site.xml を含む、Web サイトの XML 構成ファイルの階層表現について説明します。

> **注意：** わかりやすく説明するために、終了タグは省略されています。

```
<web-site cluster-island="..." display-name="..." host="..."
          log-request-info="..." max-request-size="..." secure="..."
          protocol="..." port="..." use-keep-alives="..."
          virtual-hosts="...">
     <description>
     <frontend host="..." port="...">
     <web-app application="..." load-on-startup="..." access-log="..."
              max-inactivity-time="..." name="..." root="..." shared="...">
     <default-web-app application="..." load-on-startup="..."
              max-inactivity-time="..." name="..." root="..." shared="...">
     <user-web-apps max-inactivity-time="..." path="...">
     <access-log format="..." path="..." split="..." suffix="...">
     <odl-access-log path="..." max-file-size="..." max-directory-size="...">
     <ssl-config keystore="..." keystore-password="..."
                 needs-client-auth="..." provider="..." factory="...">
          <property name="..." value="...">
```

サンプルの default-web-site.xml ファイル

これは、Oracle Application Server 環境の OC4J で提供されるデフォルト・ファイルに類似した、サンプルの default-web-site.xml ファイルです。

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE web-site PUBLIC "Oracle Application Server XML Web-site"
 "http://xmlns.oracle.com/ias/dtds/web-site.dtd">

<web-site host="myhost" port="0"  protocol="ajp13"
          display-name="Default Oracle Application Server Java WebSite"
          cluster-island="1" >

   <!-- Uncomment the following line when using clustering -->
   <!-- <frontend host="your_host_name" port="80" /> -->
   <!-- The default web-app for this site, bound to the root -->
   <default-web-app application="default" name="defaultWebApp" root="/j2ee" />
   <web-app application="default" name="dms" root="/dmsoc4j" />
   <web-app application="default" name="admin_web" root="/adminoc4j" />

   <!-- Access Log, where requests are logged to -->
   <access-log path="../log/default-web-access.log" />

   <!-- Uncomment this if you want to use ODL logging capabilities
   <odl-access-log path="../log/default-web-access" max-file-size="1000"
```

```
                    max-directory-size="10000"/>
  -->

</web-site>
```

7 Enterprise Manager を使用した構成

Oracle Application Server 環境では、Oracle Enterprise Manager 10*g* を使用して Web モジュールを構成します。この章では、サーブレットおよび Web サイトの構成に使用する、Enterprise Manager の主な機能を説明します。次の項が含まれます。

- Oracle Enterprise Manager 10g での Web モジュールの構成
- Application Server Control コンソールのページの説明

Oracle Enterprise Manager 10*g* での Web モジュールの構成

第 6 章で説明されている global-web-application.xml、orion-web.xml および default-web-site.xml の各要素と属性を直接使用するのは、OC4J スタンドアロン環境で開発およびデプロイする場合です。Oracle Application Server 環境での本番デプロイなどでは、Enterprise Manager を使用して Web モジュールの構成およびデプロイを行います。

Oracle Enterprise Manager 10*g* Application Server Control コンソールは、Oracle Application Server インスタンス用の管理コンソールです。このコンソールを使用すると、パフォーマンスのリアルタイムの監視、Oracle Application Server のコンポーネントおよびインスタンスの管理、これらのコンポーネントおよびインスタンスの構成を実行できます。これには OC4J のインスタンスがすべて含まれています。特に、Application Server Control コンソールにはサーブレットおよび Web サイトを構成するためのページが含まれています。Application Server Control コンソールは Oracle Application Server とともにインストールされます。ias_admin ユーザーとしてログインしてください。

この章では、Oracle Application Server で OC4J インスタンス内の Web モジュールの管理および構成に関連する Application Server Control コンソールのページについて説明します。一部のページでは、global-web-application.xml、orion-web.xml および default-web-site.xml の各設定を変更できます。他のページに表示される web.xml の設定は、orion-web.xml の設定を使用してオーバーライドできます。

各ページの説明では、web.xml、orion-web.xml/global-web-application.xml または default-web-site.xml の対応する要素と属性についても記述しています。global-web-application.xml または orion-web.xml の要素と属性の詳細は、6-2 ページの「global-web-application.xml および orion-web.xml の要素の説明」で説明しています。default-web-site.xml 要素と属性の詳細は、6-19 ページの「Web サイト XML ファイルの要素の説明」で説明しています。web.xml 要素の情報については、サーブレット仕様を参照してください。

OC4J とともに Enterprise Manager を使用する場合の詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

Application Server Control コンソールのページの説明

次の各項で、サーブレットおよび Web サイトの構成およびデプロイに使用される、Enterprise Manager の Application Server Control コンソールの主要なページを説明します。

- Application Server Control コンソールの OC4J ホームページ
- Application Server Control コンソールの OC4J の「アプリケーション」ページ
- Application Server Control コンソールの「アプリケーションのデプロイ」ページ
- Application Server Control コンソールの「Web アプリケーションのデプロイ」ページ
- Application Server Control コンソールの OC4J の「管理」ページ
- Application Server Control コンソールの「Web サイト・プロパティ」ページ
- Application Server Control コンソールの「Web モジュール」ページ
- Application Server Control コンソールの Web モジュールの「プロパティ」ページ
- Application Server Control コンソールの Web モジュールの「マッピング」ページ
- Application Server Control コンソールの Web モジュールの「フィルタ処理とチェーン」ページ
- Application Server Control コンソールの Web モジュールの「環境」ページ
- Application Server Control コンソールの Web モジュールの「拡張プロパティ」ページ

Application Server Control コンソールの OC4J ホームページ

Enterprise Manager で Application Server Control コンソールから初めて Oracle Application Server インスタンスにアクセスすると、Oracle Application Server インスタンスのホームページが表示されます。このページからシステム・コンポーネント表内のインスタンス名（たとえば、home）を選択すると、実行中の任意の OC4J インスタンスにドリルダウンできます。Application Server Control コンソールで、そのインスタンスの OC4J ホームページが表示されます。

図 7-1 に、home インスタンスの OC4J ホームページの一部を示します。

図 7-1 Application Server Control コンソールの OC4J ホームページ

OC4J: home

Home Applications Administration

Page Refreshed Jul 3, 2003 3:00:50 PM

General

Status Up

Start Time Jul 2, 2003 6:47:35 PM

Virtual Machines 1

Stop Restart

Status

CPU Usage (%) 0.13

Memory Usage (MB) 67.98

Heap Usage (MB) 26.04

JDBC Usage

Open JDBC Connections 0

Total JDBC Connections 0

Active Transactions 0

Transaction Commits 0

Transaction Rollbacks 0

Response - Servlets and JSPs

Active Sessions 0

Active Requests 1

Request Processing Time (seconds) 0.005

Requests per Second 0.16

Response - EJBs

Active EJB Methods 0

Method Execution Time (seconds) 0.00

Method Execution Rate (per second) 0.00

Home Applications Administration

OC4J ホームページでは、次のことを実行できます。

- **「アプリケーション」**をクリックして、Application Server Control コンソールの OC4J の「アプリケーション」ページにアクセス
- **「管理」**をクリックして、Application Server Control コンソールの OC4J の「管理」ページにアクセス

Application Server Control コンソールの OC4J の「アプリケーション」ページ

図 7-2 に OC4J の「アプリケーション」ページを示します。ここからアプリケーションをデプロイできます。OC4J ホームページの**「アプリケーション」**をクリックすると、このページが表示されます。

このマニュアルで説明されている項目に関連して、特に次の点に注意してください。

- **「EAR ファイルのデプロイ」**ボタンをクリックすると、「アプリケーションのデプロイ」ページにアクセスします。
- **「WAR ファイルのデプロイ」**ボタンをクリックすると、「Web アプリケーションのデプロイ」ページにアクセスします。

図 7-2 Application Server Control コンソールの OC4J の「アプリケーション」ページ

OC4J: home

Home | Applications | Administration

Page Refreshed Jul 3, 2003 3:03:51 PM

Default Application Name default
Default Application Path application.xml

Deployed Applications

Deploy EAR file | Deploy WAR file

Edit | Undeploy | Redeploy

Select	Name	Path	Parent Application	Active Requests	Request Processing Time (seconds)	Active EJB Methods
◉	FAQAPP_SB	../applications/FAQAPP_SB.ear	default	0	0.00	0

Home | Applications | Administration

Application Server Control コンソールの「アプリケーションのデプロイ」ページ

図 7-3 に、Application Server Control コンソールの「アプリケーションのデプロイ」ページの主要部分を示します。これは、EAR ファイルのデプロイに使用するページです。OC4J インスタンスの「アプリケーション」ページからこのページにドリルダウンするには、**「EAR ファイルのデプロイ」**ボタンをクリックします。

図 7-3 Application Server Control コンソールの「アプリケーションのデプロイ」ページ

Deploy Application

For a J2EE application to be successfully deployed on the OC4J container, the application has to be assembled correctly as an Enterprise Archive (ear) file, with all the needed application and module deployment descriptors. The OC4J container generates default OC4J specific deployment descriptors when the application is deployed. If you have custom OC4J specific deployment descriptors that you wish to use, you need to include these in the ear file.

Select the J2EE application (.ear file) to be deployed.
J2EE Application [ ] Browse...

Specify a unique application name for this application.
Application Name [ ]

Select the parent for this application.
Parent Application [default]

Cancel | Continue

「アプリケーションのデプロイ」ページで、**「参照」**ボタンをクリックして、デプロイする EAR ファイルを選択し、次に、J2EE アプリケーション名を指定します。この名前は、通常、EAR ファイルの名前から `.ear` 拡張子を除外した部分と同じです。また、親アプリケーションも指定できますが、通常は OC4J のデフォルト・アプリケーションを親として使用します。

デプロイすると、新しい `<application>` 要素が `server.xml` ファイルに追加されます。

「続行」ボタンをクリックすると、「アプリケーションのデプロイ : Web モジュールの URL マッピング」ページが表示されます。このページでは、J2EE アプリケーションに含まれる Web アプリケーションの URL コンテキスト・パスを設定できます。図 7-4 にこのページを示します。`utility` という J2EE アプリケーションの Web アプリケーションのデフォルトのコンテキスト・パスが示されています。**「次へ」**ボタンをクリックすると、エントリを確認し、デプロイできます。

URL コンテキスト・パスを指定すると、Web アプリケーションを Web サイトにバインドするための要素が `default-web-site.xml` ファイルに追加されます。これは、`<web-site>` 要素の新しい `<web-app>` サブ要素を使用して行われます。さらに、Oracle HTTP Server の `mod_oc4j` Apache mod の `mod_oc4j.conf` 構成ファイルが適切なマウント・ポイントで更新されます。

注意： コンテキスト・パスの指定時には、次の書式は等価として扱われます。

```
someUrl
/someUrl
/someUrl/
```

図 7-4 Application Server Control コンソールの「アプリケーションのデプロイ：Web モジュールの URL マッピング」ページ

URL Mappings for Web Modules　Review

Deploy Application: URL Mapping for Web Modules

A web module needs to be mapped to an URL pattern in the default web site before it can be accessed. The following table lists all the web modules found in your application. Specify the URL mapping for each of these modules.

Name	URL Mapping
Web Services Example	* /j2ee/utility

Cancel　Step 1 of 2　Next　Finish

Application Server Control コンソールの「Web アプリケーションのデプロイ」ページ

図 7-5 に、Application Server Control コンソールの「Web アプリケーションのデプロイ」ページの主要部分を示します。これは、独立した WAR ファイルのデプロイに使用するページです。OC4J インスタンスの「アプリケーション」ページからこのページにドリルダウンするには、**「WAR ファイルのデプロイ」**ボタンをクリックします。

注意： 独立した WAR ファイルをデプロイする場合、これは透過的に EAR ファイルにラップされています。

図 7-5 Application Server Control コンソールの「Web アプリケーションのデプロイ」ページ

Deploy Web Application

Select the Web Application (.war file) you wish to deploy. This web application will be wrapped into a J2EE application (.ear file) before deployment.

Web Application　Browse...

Specify the name you would like this application to be called and the URL to map this web application to.

Application Name

Map to URL

Cancel　Deploy

「Web アプリケーションのデプロイ」ページで、**「参照」**ボタンをクリックして、デプロイするWAR ファイルを選択します。次に、J2EE アプリケーション名と URL コンテキスト・パスを指定し、Web アプリケーションにマップします。指定したアプリケーション名の J2EE アプリケーションが、Web アプリケーションを含めるために透過的に作成されます。OC4J では、すべての Web アプリケーションが J2EE アプリケーション内に存在する必要があります。

EAR ファイルのときと同様に、デプロイにより `server.xml` ファイルに新しい `<application>` 要素が追加されます。さらに、Web アプリケーションを Web サイトにバインドするために、`default-web-site.xml` ファイルの `<web-site>` 要素に新しい `<web-app>` サブ要素が追加されます。最後に、Oracle HTTP Server の `mod_oc4j` Apache mod の `mod_oc4j.conf` 構成ファイルが適切なマウント・ポイントで更新されます。

注意： コンテキスト・パスの指定時には、次の書式は等価として扱われます。

```
someUrl
/someUrl
/someUrl/
```

Application Server Control コンソールの OC4J の「管理」ページ

図 7-6 に、OC4J 管理ページを示します。ここから OC4J インスタンスのプロパティにアクセスできます。OC4J ホームページの**「管理」**をクリックすると、このページが表示されます。

「インスタンス・プロパティ」の下の**「Web サイト・プロパティ」**を選択すると、「Web サイト・プロパティ」ページにアクセスします。このページを経由して様々なページにアクセスし、Web モジュールのプロパティを更新できます。

図 7-6 Application Server Control コンソールの OC4J の「管理」ページ

OC4J: home

Home Applications **Administration**

Page Refreshed **Jul 3, 2003 3:02:48 PM**

Instance Properties

Server Properties
Website Properties
JSP Container Properties
Replication Properties
Advanced Properties

Application Defaults

Data Sources
Security
JMS Providers
Global Web Module

OC4J Inheritance

OC4J applications have a hierarchical parent-child relationship to facilitate administration through inheritance. A child application inherits certain attributes from its parent application such as principals and JNDI objects including data sources, JMS providers and EJBs. When an OC4J application is deployed, you specify the parent application. The Default Application is the top of the parent hierarchy.

Home Applications **Administration**

Application Server Control コンソールの「Web サイト・プロパティ」ページ

図 7-7 に、OC4J インスタンスに対する Application Server Control コンソールの「Web サイト・プロパティ」ページの主要部分を示します。このページにドリルダウンするには、OC4J の「管理」ページの「インスタンス・プロパティ」の下の **「Web サイト・プロパティ」** をクリックします。

図 7-7 Application Server Control コンソールの「Web サイト・プロパティ」ページ

Website Properties

Default Web Module

Name **defaultWebApp**
Application **default**
Load on startup **true**

URL Mappings for Web Modules

Previous 1-3 of 3 Next

Name	Application	URL Binding	Load on startup
cabo	BC4J	/cabo	☐
dms	default	/dmsoc4j	☐
webapp	BC4J	/webapp	☐

Revert Apply

特に、このページでは、OC4J の起動時に各アプリケーションを自動的にロードするかどうかを指定できます。(指定しない場合、アプリケーションは最初にリクエストされるまでロードされません。) これは、`default-web-site.xml` ファイルにある `<web-site>` 要素の適切な `<web-app>` サブ要素の `load-on-startup` 属性に対応しています。(OC4J 起動時のアプリケーションのロードに関する一般的な情報は、2-6 ページの「サーブレットの事前ロード」を参照してください。)

「Web サイト・プロパティ」ページから特定の Web モジュールの「Web モジュール」ページにドリルダウンします。たとえば、このサンプル・ページでは、**`webapp`** を選択して、対象モジュールの「Web モジュール」ページにドリルダウンできます。

Application Server Control コンソールの「Web モジュール」ページ

図 7-8 に、webapp モジュールに対する Application Server Control コンソールの「Web モジュール」ページの主要部分を示します。特定のモジュールの「Web モジュール」ページにドリルダウンするには、「Web サイト・プロパティ」ページでモジュール名をクリックします。

図 7-8 Application Server Control コンソールの「Web モジュール」ページ

Web Module: webapp

Refreshed at Wednesday, July 17, 2002 6:11:35 PM PDT

General

Status **Loaded**
URL Binding **/webapp**
Referenced EJBs

Response and Load

Active Sessions **3**
Active Requests **0**
Request Client Time (secs) **0**
Request Load Time (secs) **0**
Request Overhead Time (secs) **0**
Requests per Second **0**
Requests Processed **11**

Servlets/JSPs

Return to Top

Previous 1-1 of 1 Next

Name	Status	Type	Source	Active Requests	Request Client Time (secs)	Requests per Second	Startup Priority
EMDServlet	Loaded	Servlet	oracle.jbo.server.emd.EMDServlet	0	0	0	

Administration

Return to Top

Properties

General
Mappings
Filtering and Chaining
Environment
Advanced Properties

Security

General

「Web モジュール」ページの「管理」セクションの「プロパティ」の下にある次のリンクを経由して、Web モジュール・プロパティの複数のカテゴリにアクセスできます。

- **「一般」**（Web モジュールの「プロパティ」ページにドリルダウンします）。
- **「マッピング」**（Web モジュールの「マッピング」ページにドリルダウンします）。
- **「フィルタ処理とチェーン」**（Web モジュールの「フィルタ処理とチェーン」ページにドリルダウンします）。
- **「環境」**（Web モジュールの「環境」ページにドリルダウンします）。
- **「拡張プロパティ」**（Web モジュールの「拡張プロパティ」ページにドリルダウンします）。

Application Server Control コンソールの Web モジュールの「プロパティ」ページ

図 7-9 および図 7-10 に、特定のモジュールに対する Application Server Control コンソールの Web モジュールの「プロパティ」ページの一部を示します。このページにドリルダウンするには、「Web モジュール」ページの「管理」セクションにある「プロパティ」の下の**「一般」**をクリックします。

図 7-9 Application Server Control コンソールの Web モジュールの「プロパティ」ページ（1/2）

Properties

General

Display Name
Description **BC4J Web Application**
Distributable **true**
Document Root **../**

Servlet Directory /servlet/
Temporary Directory ./temp
Response Buffer Size (bytes) 2048
File Check Interval (milliseconds) 1000

Session Configuration

Return to Top

Use Cookies Yes
Session Auto Join No
Session Timeout (minutes) 30
Cookie Max Age (seconds)
Cookie Domain
Session Storage Directory

図 7-10 Application Server Control コンソールの Web モジュールの「プロパティ」ページ（2/2）

Class Paths

Return to Top

Specifies where java classes used by this application can be found.

Previous Next

Select	Path
	(No items found)

Add Class Path

Session Trackers

Return to Top

Specifies a servlet to use as a session tracker. Session trackers are invoked as soon as a session is created and are useful for logging purposes.

Previous Next

Select	Servlet Name
	(No items found)

Add Session Tracker

Virtual Directories

Return to Top

Virtual directories may be used to expose web files that don't physically reside below the document root.

Previous Next

Select	Real Path	Virtual Path
	(No items found)	

Add Virtual Directory

Tag Libraries

Return to Top

A tag library is a collection of custom tags that encapsulate functionality used within JSP pages.

Previous Next

Name	Location
(No items found in J2EE deployment descriptor)	

これらの設定は、次のように、orion-web.xml 要素に対応しています。

「一般」セクション：

- 「サーブレット・ディレクトリ」は、<orion-web-app> 要素の servlet-webdir 属性に対応しています。
- 「一時ディレクトリ」は、<orion-web-app> 要素の temporary-directory 属性に対応しています。
- 「レスポンス・バッファ・サイズ (B)」は、<orion-web-app> 要素の default-buffer-size 属性に対応しています。
- 「ファイル・チェック間隔 (ミリ秒)」は、<orion-web-app> 要素の file-modification-check-interval 属性に対応しています。

「セッション構成」セクション：

- 「Cookie を使用」は、<orion-web-app> 要素のサブ要素である <session-tracking> 要素の cookies 属性に対応しています。
- 「セッション自動結合」は、<session-tracking> 要素の autojoin-session 属性に対応しています。
- 「セッション・タイムアウト (分)」は、標準 <web-app> 要素の <session-config> サブ要素の <session-timeout> サブ要素に対応しています。orion-web.xml の <orion-web-app> の下にある <web-app> サブ要素を使用すると、アプリケーションの web.xml ファイルの <web-app> 設定に対してデプロイ固有のオーバーライドを実行できます。
- 「Cookie 最大有効期間 (秒)」は、<session-tracking> 要素の cookie-max-age 属性に対応しています。

- 「Cookie ドメイン」は、<session-tracking> 要素の cookie-domain 属性に対応しています。
- 「セッション記憶域ディレクトリ」は、<orion-web-app> 要素の persistence-path 属性に対応しています。

「CLASSPATH」セクション：

- このセクションでの CLASSPATH の追加は、<orion-web-app> 要素の <classpath> サブ要素の path 属性の設定に対応しています。

「Session Tracker」セクション：

- このセクションでのセッション・トラッカの追加は、<session-tracking> 要素のサブ要素である <session-tracker> 要素の servlet-name 属性の設定に対応しています。

「仮想ディレクトリ」セクション：

- このセクションでの仮想ディレクトリの追加は、<orion-web-app> 要素の <virtual-directory> サブ要素の real-path 属性と virtual-path 属性の設定に対応しています。

「タグ・ライブラリ」セクション：

- このセクションでは、WAR ファイルの内容に基づいて、アプリケーションで使用される JSP タグ・ライブラリを表示します。

Application Server Control コンソールの Web モジュールの「マッピング」ページ

図 7-11 および図 7-12 に、特定の Web モジュールに対する Application Server Control コンソールの Web モジュールの「マッピング」ページの一部を示します。このページにドリルダウンするには、「Web モジュール」ページの「管理」セクションにある「プロパティ」の下の**「マッピング」**をクリックします。

図 7-11 Application Server Control コンソールの Web モジュールの「マッピング」ページ（1/2）

Mappings

Servlet Mappings

Defines a mapping between a servlet and a url pattern.

Previous Next

Servlet Name	URL Pattern
(No items found in J2EE deployment descriptor)	

MIME Mappings — Return to Top

Defines a mapping between and an extension and a mime type.

Previous 1-2 of 2 Next

MIME Type	Extension
text/html	html
text/plain	txt

図 7-12 Application Server Control コンソールの Web モジュールの「マッピング」ページ（2/2）

Welcome Files

Return to Top

Welcome files will be served to the user for incoming requests without a file specified (an existing directory is specified).

Previous Next

File Name
(No items found in J2EE deployment descriptor)

Error Pages

Return to Top

Defines a mapping between an error code or java exception type and the location of a resource.

Previous Next

Error Code/Exception Class	Location
(No items found in J2EE deployment descriptor)	

次の設定はすべて、web.xml ファイルにある <web-app> 要素のサブ要素に対応しています。orion-web.xml の <orion-web-app> の下にある <web-app> サブ要素を使用すると、これらの設定に対してデプロイ固有のオーバーライドを実行できます。このオーバーライドは、「拡張プロパティ」ページで実行できます。詳細は、7-15 ページの「Application Server Control コンソールの Web モジュールの「拡張プロパティ」ページ」を参照してください。

「サーブレット・マッピング」セクション：

- このセクションで指定するサーブレット名または URL パターンは、<web-app> 要素の <servlet-mapping> サブ要素の <servlet-name> または <url-pattern> サブ要素に対応します。

「MIME マッピング」セクション：

- このセクションで指定する MIME タイプと拡張子は、<web-app> 要素の <mime-mapping> サブ要素の <mime-type> サブ要素と <extension> サブ要素の設定に対応しています。

「初期ファイル」セクション：

- このセクションで指定するファイル名は、<web-app> 要素の <welcome-file-list> サブ要素の <welcome-file> サブ要素の設定に対応しています。

「エラー・ページ」セクション：

- このセクションで指定するエラー・コードと場所は、<web-app> 要素の <error-page> サブ要素の <error-code> サブ要素と <location> サブ要素の設定に対応しています。
- このセクションで指定する例外クラスと場所は、<web-app> 要素の <error-page> サブ要素の <exception-type> サブ要素と <location> サブ要素の設定に対応しています。

Application Server Control コンソールの Web モジュールの「フィルタ処理とチェーン」ページ

図 7-13 に、特定のモジュールに対する Application Server Control コンソールの Web モジュールの「フィルタ処理とチェーン」ページの主要部分を示します。このページにドリルダウンするには、「Web モジュール」ページの「管理」セクションにある「プロパティ」の下の **「フィルタ処理とチェーン」** をクリックします。

図 7-13 Application Server Control コンソールの Web モジュールの「フィルタ処理とチェーン」ページ

Filtering and Chaining

Refreshed at Sunday, July 14, 2002 3:42:38 PM PDT

Servlet Filtering

Specifies a servlet to use as a filter. Filters are invoked for every request and have the option of handling the request or simply ignoring it and passing it on for normal processing.

Previous Next

Select	URL Pattern	Servlet Name
	(No items found)	

Add Filter

Servlet Chaining

Specifies a servlet to use as chainer for a specified mime-type. Useful to filter/transform certain kinds of output.

Previous Next

Select	MIME Type	Servlet Name
	(No items found)	

Add Chain

Revert Apply

これらの設定は、次のように、orion-web.xml 要素に対応しています。

「サーブレットのフィルタ処理」セクション：

- このセクションでのフィルタの追加は、<web-app> 要素の下にある <filter-mapping> サブ要素の <servlet-name> または <url-pattern> サブ要素の設定に対応しています。指定するサーブレット名は、web.xml ファイルの標準構成を使用して、サーブレット・クラスに関連付けられます。

「サーブレットの連鎖」セクション：

- このセクションでの連鎖の追加は、<orion-web-app> 要素の <servlet-chaining> サブ要素の servlet-name 属性と mime-type 属性の設定に対応しています。指定するサーブレット名は、web.xml ファイルの標準構成を使用して、サーブレット・クラスに関連付けられます。

注意： サーブレット・チェーンは、標準サーブレット・フィルタと基本的に同様の機能を持つ旧式のメカニズムです。これは、サーブレット仕様のバージョン 2.3 で説明されています。かわりに、サーブレット・フィルタの使用をお薦めします。3-2 ページの「サーブレット・フィルタ」を参照してください。

Application Server Control コンソールの Web モジュールの「環境」ページ

図 7-14 に、特定の Web モジュールに対する Application Server Control コンソールの Web モジュールの「環境」ページの主要部分を示します。このページにドリルダウンするには、「Web モジュール」ページの「管理」セクションにある「プロパティ」の下の**「環境」**をクリックします。

図 7-14 Application Server Control コンソールの Web モジュールの「環境」ページ

Environment

Servlet Context Parameters

Overrides the value of the web application's servlet context initialization parameters.

Previous Next

Name	Value	Deployed Value
(No items found in J2EE deployment descriptor)		

Environment Entries

Return to Top

Overrides the value of environment entries specified in the assembly descriptor.

Previous Next

Name	Type	Description	Value	Deployed Value
(No items found in J2EE deployment descriptor)				

Resource References

Return to Top

Associates the declaration of a reference to an external resource such as a datasource, JMS queue or mail session with a JNDI-location when deploying.

Previous Next

Name	Type	Authorization	Description	JNDI Location	Lookup Context
(No items found in J2EE deployment descriptor)					

Revert Apply

このページには、サーブレット・コンテキスト・パラメータのオーバーライド、環境エントリのオーバーライドおよびリソース参照の設定が表示されます。このオーバーライドは、対応する web.xml 設定をオーバーライドする orion-web.xml ファイルの設定を示します。

これらの設定は、次のように、web.xml および orion-web.xml 要素に対応しています。

「サーブレット・コンテキスト・パラメータ」セクション：

- このセクションには、このデプロイに対してオーバーライドできる web.xml の <context-param> 要素の設定が、すでに指定済の「デプロイ済の値」とともに表示されます。「デプロイ済の値」列に新しい値を入力して、新規のオーバーライドを指定します。この指定によって、orion-web.xml に <context-param-mapping> エントリが作成されます。

「環境エントリ」セクション：

- このセクションには、このデプロイに対してオーバーライドできる web.xml の <env-entry> 要素の設定が、すでに指定済の「デプロイ済の値」とともに表示されます。「デプロイ済の値」列に新しい値を入力して、新規のオーバーライドを指定します。この指定によって、orion-web.xml に <env-entry-mapping> エントリが作成されます。

「リソース参照」セクション:

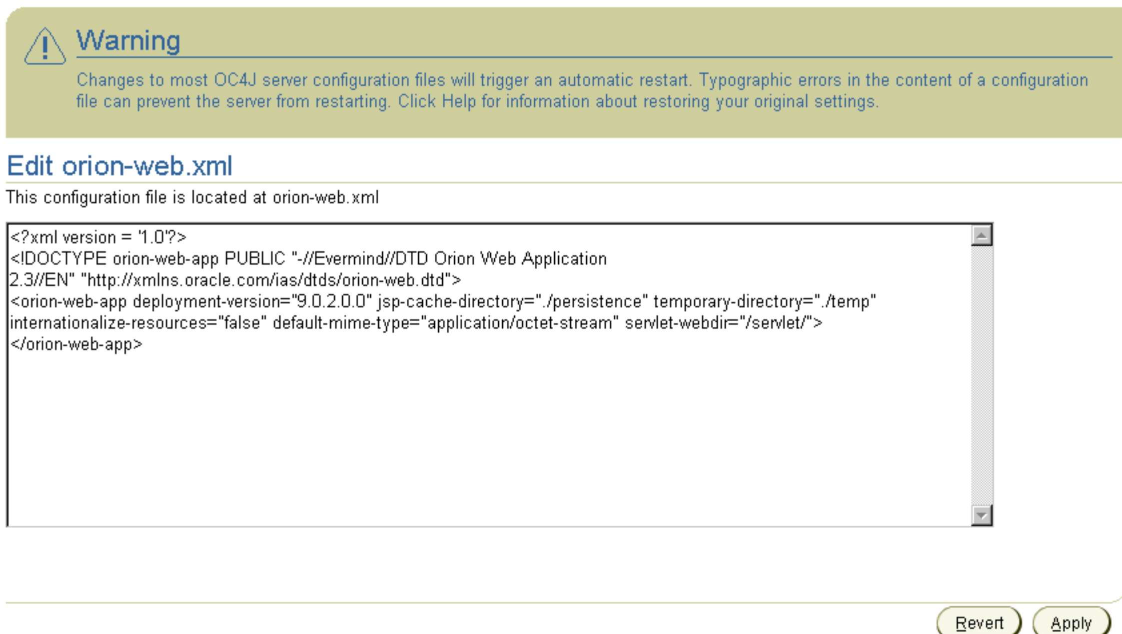
■ このセクションには、web.xml 設定と orion-web.xml 設定の組合せが表示されます。リソース参照名とタイプは、web.xml ファイルの <web-app> 要素の <resource-ref> サブ要素の下にある <res-ref-name> サブ要素と <res-type> サブ要素に対応しています。JNDI の場所と参照コンテキストは、orion-web.xml ファイルの <orion-web-app> 要素の下にある <resource-ref-mapping> 要素とその <lookup-context> サブ要素の設定に対応しています。

Application Server Control コンソールの Web モジュールの「拡張プロパティ」ページ

図 7-15 に、特定の Web モジュールに対する Application Server Control コンソールの Web モジュールの「拡張プロパティ」ページの主要部分を示します。このページにドリルダウンするには、「Web モジュール」ページの「管理」セクションにある「プロパティ」の下の**「拡張プロパティ」**をクリックします。

Web モジュールの「拡張プロパティ」ページでは、orion-web.xml または global-web-application.xml を編集して、前述の Application Server Control コンソールの Web モジュールの各ページでは処理されない設定を指定できます。実際には、「拡張プロパティ」ページを使用すると、orion-web.xml または global-web-application.xml のあらゆるエントリを作成できます。ただし、エラー処理機能とレポート作成機能があるため、可能なかぎり、前述の各ページを使用することをお薦めします。

図 7-15 Application Server Control コンソールの Web モジュールの「拡張プロパティ」ページ



A

オープン・ソース・フレームワークおよびユーティリティ

一般的に使用されているオープン・ソース・フレームワークおよびユーティリティには、Oracle Application Server 10*g* リリース 2（10.1.2）で OC4J とともに使用できるものがあります。この付録では、そのうちの 2 つ、Jakarta Struts および Jakarta log4j を構成および使用する方法を説明します。

ここでは、OC4J スタンドアロン環境での、これらのオープン・ソース・ユーティリティの構成および使用について説明します。次の各項で詳細に説明します。

- OC4J における Jakarta Struts の構成および使用
- OC4J における Jakarta log4j の構成および使用

重要：

- このマニュアルのパッケージおよび構成手順は、OC4J スタンドアロン・インストールの場合を前提として書かれています。Oracle Application Server 環境の場合、Enterprise Manager および `dcmctl` コマンドライン・ユーティリティなどの付属の管理ツールを使用して、これらの作業を行ってください。Oracle Application Server 環境では、構成ファイルを手動で変更しないでください。
- ここで説明しているオープン・ソース・ユーティリティおよびフレームワークは、オラクル社では直接サポートしていません。さらに、これらのユーティリティおよびフレームワークと OC4J 製品に関し、正式なテストや認定はされていません。これらのフレームワークを使用する際のサポートには、オープン・ソース・コミュニティの一般的なフォーラムを使用してください。
- ここでは、特定のバージョンの Struts と log4j について説明しますが、その内容の一般的なフレームワークはそれ以降のバージョンにも当てはまります。

OC4J における Jakarta Struts の構成および使用

次の各項で、Jakarta Struts を OC4J スタンドアロン環境で使用するための手順について説明します。

- Jakarta Struts の概要
- Struts バイナリ配布版のダウンロード
- Struts バイナリ配布版の解凍
- Struts ドキュメントのインストールおよびアクセス
- Struts サンプル Web アプリケーションのインストール
- Struts フレームワークを使用したユーザー・アプリケーションのデプロイ

注意： Oracle JDeveloper には、Struts を簡単に使用できるウィザードがあります。

Jakarta Struts の概要

Jakarta Struts は、オープン・ソース・フレームワークで、Java サーブレット、JavaServer Pages および XML などのオープン・スタンダードを使用した Web アプリケーションの開発を支援します。Struts は、Model-View-Controller（MVC）パターンに基づいたモジュール化アプリケーション開発モデルをサポートしています。Struts を使用することにより、業界標準や実証済設計モデルに基づき、アプリケーションの拡張可能な開発環境を作成できます。

次の各項で、Struts ライブラリ、ドキュメントおよびサンプル・アプリケーションを OC4J スタンドアロン環境にインストールする方法について説明します。このマニュアルでは、Struts を使用したアプリケーションの構築方法は説明しません。Struts の Web サイトにあるユーザー・ガイド、インストール・ガイドおよびその他のドキュメントを参照してください。次のサイトから入手可能です。

http://struts.apache.org

http://struts.apache.org/learning.html

注意： Struts は、Apache Jakarta プロジェクトの一部で、Apache Software Foundation によって運営されています。

Struts バイナリ配布版のダウンロード

Struts の配布版は、次のサイトから入手可能です。

http://struts.apache.org/

アーカイブ・ファイルをこのサイトからダウンロードします。プラットフォームに合ったフォーマット（ZIP ファイルまたは圧縮済の TAR ファイル）を選択し、ローカル・ファイル・システムに保存します。

注意： 次の項では、Struts 1.0.2 バージョンを使用する手順を示していますが、それ以降のバージョンの Struts でも同じ一般的な手順を使用して OC4J にデプロイできます。ただし、新しいバージョンには機能が追加され、ライブラリ・ファイルが増えている可能性があります。

Struts バイナリ配布版の解凍

WinZip または TAR など、プラットフォームに合ったツールを使用して、ダウンロードした Struts バイナリ配布版のアーカイブ・ファイルを解凍します。これにより、次のディレクトリ構造が作成されます（Struts 1.0.2 リリースの場合）。

```
jakarta-struts-1.0.2/INSTALL
jakarta-struts-1.0.2/LICENSE
jakarta-struts-1.0.2/README

jakarta-struts-1.0.2/lib/jdbc2_0-stdext.jar
jakarta-struts-1.0.2/lib/struts.jar
jakarta-struts-1.0.2/lib/struts.tld
jakarta-struts-1.0.2/lib/struts-bean.tld
jakarta-struts-1.0.2/lib/struts-config_1_0.dtd
jakarta-struts-1.0.2/lib/struts-form.tld
jakarta-struts-1.0.2/lib/struts-html.tld
jakarta-struts-1.0.2/lib/struts-logic.tld
jakarta-struts-1.0.2/lib/struts-template.tld
jakarta-struts-1.0.2/lib/web-app_2_2.dtd
jakarta-struts-1.0.2/lib/web-app_2_3.dtd

jakarta-struts-1.0.2/webapps/struts-blank.war
jakarta-struts-1.0.2/webapps/struts-documentation.war
jakarta-struts-1.0.2/webapps/struts-example.war
jakarta-struts-1.0.2/webapps/struts-exercise-taglib.war
jakarta-struts-1.0.2/webapps/struts-template.war
jakarta-struts-1.0.2/webapps/struts-upload.war
```

Struts ドキュメントのインストールおよびアクセス

Struts ドキュメントは、Web アプリケーションとして、Struts アーカイブの webapps ディレクトリの WAR ファイル内に用意されています。次の手順を、選択した Struts のバージョンに合せて使用し、Struts ドキュメント Web アプリケーションを OC4J デフォルト・アプリケーションにデプロイします。

構成ファイルは、j2ee/home/config ディレクトリに存在します。

1. OC4J グローバル・アプリケーション・ディスクリプタ application.xml に、struts-documentation.war ファイル用に新規の <web-module> 要素を追加します。この要素は、ファイル内に存在している任意の <web-module> 要素の後に置きます。

   Struts バイナリ配布版を解凍したディレクトリのパスを指定します。次にエントリの例を示します。

   ```
   <orion-application ... >
      ...
      <web-module id="struts-documentation"
      path="your_path/jakarta-struts-1.0.2/webapps/struts-documentation.war" />
   ...
   </orion-application>
   ```

2. Web サイトの XML ファイル http-web-site.xml に、ドキュメント Web アプリケーションを URL コンテキスト・パスにバインドするための新規 <web-app> 要素を追加します。この要素は、ファイル内に存在している任意の <web-app> 要素の後に置きます。次に、/struts/doc を Struts ドキュメントの URL コンテキスト・パスに指定するエントリの例を示します。

   ```
   <web-site ... >
      ...
      <web-app application="default" name="struts-documentation"
               root="/struts/doc" />
      ...
   </web-site>
   ```

OC4J のデフォルト・アプリケーションを使用するための `application="default"` 設定に注意してください。OC4J にデプロイされる Web アプリケーションは、すべて J2EE アプリケーション内に存在する必要があります。通常、このために、Web アプリケーションの WAR ファイルを J2EE アプリケーションの EAR ファイル内にパッケージ化します。ただし、この例のように、OC4J デフォルト・アプリケーションを使用してスタンドアロン WAR ファイルをデプロイして、手順を簡略化することもできます。

3. コマンドラインから OC4J を起動します。

```
% java -jar oc4j.jar
```

次のような出力が表示されます。

```
Auto-unpacking /java/jakarta-struts-1.0.2/webapps/struts-documentation.war
... done.
Oracle Application Server (10.1.2.0.0) Containers for J2EE initialized
```

`struts-documentation.war` を解凍すると、`jakarta-struts-1.0.2/webapps` ディレクトリの下に `struts-documentation` ディレクトリおよびサブディレクトリが作成および移入されます。

4. `http-web-site.xml` で指定した URL コンテキスト・パスを使用してドキュメントにアクセスします。

```
http://host:8888/struts/doc
```

Struts ドキュメントの Welcome ページが表示されます。

Struts サンプル Web アプリケーションのインストール

Struts バイナリ配布版には、サンプル Web アプリケーションが `webapps` ディレクトリの WAR ファイル内に用意されています。ドキュメント Web アプリケーションと同様、Struts サンプル Web アプリケーションも OC4J デフォルト・アプリケーションにデプロイできます。次の手順を、選択した Struts のバージョンに合せて使用します。構成ファイルは、`j2ee/home/config` ディレクトリに存在します。

1. OC4J グローバル・アプリケーション・ディスクリプタ `application.xml` に、`struts-example.war` ファイル用に新規の `<web-module>` 要素を追加します。Struts バイナリ配布版を解凍したディレクトリのパスを指定します。次にエントリの例を示します。

```
<web-module id="struts-example"
   path="your_path/jakarta-struts-1.0.2/webapps/struts-example.war" />
```

これを、`struts-documentation.war` 用に作成した `<web-module>` 要素の直後に追加します。

2. Web サイトの XML ファイル `http-web-site.xml` に、サンプル Web アプリケーションを URL コンテキスト・パスにバインドするための新規の `<web-app>` 要素を追加します。次に、`/struts/example` をサンプル Web アプリケーションの URL コンテキスト・パスに指定するエントリの例を示します。

```
<web-app application="default" name="struts-example"
         root="/struts/example" />
```

これを、ドキュメント Web アプリケーション用に作成した `<web-app>` 要素の直後に追加します。

ドキュメント Web アプリケーションと同様、`application="default"` 設定を使用すると、OC4J デフォルト・アプリケーション内にサンプル Web アプリケーションが含まれます。

3. コマンドラインから OC4J を起動します。

```
% java -jar oc4j.jar
```

次のような出力が表示されます。

```
Auto-unpacking /java/jakarta-struts-1.0.2/webapps/struts-example.war
...done.
Oracle Application Server (10.1.2.0.0) Containers for J2EE initialized
```

`struts-example.war` を解凍すると、`jakarta-struts-1.0.2/webapps` ディレクトリの下に `struts-example` ディレクトリおよびサブディレクトリが作成および移入されます。

4. `http-web-site.xml` で指定した URL コンテキスト・パスを使用してサンプル Web アプリケーションにアクセスします。

   ```
   http://host:8888/struts/example
   ```

   Struts サンプル・アプリケーションの Welcome ページが表示されます。

Struts フレームワークを使用したユーザー・アプリケーションのデプロイ

Struts フレームワークを使用してユーザー・アプリケーションをデプロイするには、Struts ライブラリから必要な部分をユーザー・アプリケーションの WAR ファイルにパッケージ化し、Struts コンポーネントの必須エントリを使用して、標準の `web.xml` デプロイメント・ディスクリプタを構成します。すると、Web アプリケーションが WAR ファイルとして作成およびパッケージ化されます。

次の手順を、選択した Struts のバージョンに合せて使用します。

注意： Struts を使用するために構成された WAR ファイルの例が、Struts アーカイブ・ファイルの `webapps` フォルダに `struts-blank.war` として提供されています。この例は、ユーザーが Web アプリケーションを作成する際に、テンプレートとして使用できます。

---

1. Struts の `lib` ディレクトリからユーザー・アプリケーションの `/WEB-INF/lib` ディレクトリに、Struts ライブラリをコピーします。次に、UNIX 環境（アーカイブ・ファイルを解凍したディレクトリから）の例を示します。「`%`」はシステム・プロンプトです。

   ```
   % cp jakarta-struts-1.0.2/lib/struts.jar web-inf/lib
   ```

2. Struts タグ・ライブラリ・ディスクリプタ・ファイル（JSP タグ・ライブラリの場合はすべての `.tld` ファイル）を、Struts の `lib` ディレクトリからユーザー・アプリケーションの `/WEB-INF` ディレクトリにコピーします。

   ```
   % cp jakarta-struts-1.0.2/lib/*.tld web-inf
   ```

注意： JSP 1.1 の手段を使用するこれらの手順は、JSP タグ・ライブラリ・ディスクリプタ・ファイルにアクセスする方法の 1 つにすぎません。OC4J など、JSP 1.2 環境では、他の手段もあります。詳細は、『Oracle Application Server Containers for J2EE JavaServer Pages 開発者ガイド』を参照してください。

---

3. Struts サーブレットおよびタグ・ライブラリ・エントリを `web.xml` ファイルに追加します。

   a. Struts コントローラのサーブレット定義要素を追加します。（オプションで、アプリケーション全体で使用する `MessageResource` ファイル、Struts 構成ファイルの名前と位置、およびデバッグ・レベルなどの追加プロパティも指定できます。）`<servlet>` 要素は、トップレベルの `<web-app>` 要素のサブ要素です。

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
  <init-param>
    <param-name>application</param-name>
    <param-value>ApplicationResources</param-value>
  </init-param>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
</servlet>
```

b. Struts コントローラ・サーブレットのサーブレット・マッピング要素を追加します。この手順では、サーブレット名（前述の `<servlet>` 要素でサーブレット・クラスにマップ済）を URL サーブレット・パスにマップします。`<servlet-mapping>` 要素は、トップレベルの `<web-app>` 要素のサブ要素です。

```
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

c. Struts タグ・ライブラリのエントリを追加します。これらのエントリは、手順 2 で示したように、TLD ファイルが `/WEB-INF` ディレクトリに置かれていることを前提としています。`<taglib>` 要素は、トップレベルの `<web-app>` 要素のサブ要素です。

```
<taglib>
  <taglib-uri>/WEB-INF/struts-bean.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/WEB-INF/struts-html.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/WEB-INF/struts-logic.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>
```

これで、Web アプリケーションが、Struts フレームワークを活用するアプリケーションのデプロイをサポートできます。

JSP ページ、サーブレット、Struts コンポーネントおよびその他のファイルを含めた Web アプリケーションの残りの部分を WAR ファイルに追加すると、アプリケーションを OC4J にデプロイできます。

注意： アプリケーション内で Struts を使用するには、追加手順が必要です。Struts が実行時に使用する Action クラスおよびその他のコンポーネントを作成し、Struts 構成ファイル `struts-config.xml` に対応するエントリを作成する必要があります。これらの点は OC4J 固有のものではないため、このマニュアルの対象外です。詳細は、Struts の Web サイトの Learning Guide を参照してください。

http://struts.apache.org/learning.html

OC4J における Jakarta log4j の構成および使用

次の各項で、Jakarta log4j を OC4J スタンドアロン環境で使用するための考慮事項について説明します。

- Jakarta log4j の概要
- log4j バイナリ配布版のダウンロード
- log4j バイナリ配布版の解凍
- log4j ライブラリのインストール
- log4j 構成ファイルの使用
- log4j デバッグ・モードの使用可能化

Jakarta log4j の概要

log4j フレームワークは、Java アプリケーションの実行時ロギング処理をサポートするための効率的で柔軟な API を提供するオープン・ソース・プロジェクトです。log4j を使用することにより、様々なレベルの警告でメッセージを取り込み、ログ文をコードに挿入することが可能になります。また、システム管理者は、提供されたアプリケーション・コードを変更することなく、アプリケーションの実行時に表示するロギングのレベルを個別に定義できます。

log4j の機能を使用すると、アプリケーションのバイナリ・ファイルを変更せずに、実行時にロギングを行うことが可能です。パフォーマンスを著しく低下せずに、文を出荷時のコードに残しておくことができます。ロギングは構成ファイルによって制御されるため、アプリケーションのバイナリを変更する必要がありません。

次の各項で、log4j ライブラリをインストールし、OC4J で使用するために構成する方法を説明します。幅広い OC4J API の使用方法は OC4J 固有ではないため、このマニュアルでは説明しません。公式の log4j の Web サイトにあるドキュメントを参照してください。次のサイトから入手可能です。

http://jakarta.apache.org/log4j/docs/index.html

http://jakarta.apache.org/log4j/docs/documentation.html

注意： log4j フレームワークは、Apache Jakarta プロジェクトの一部で、Apache Software Foundation によって運営されています。

---

log4j バイナリ配布版のダウンロード

log4j 配布版は、次のサイトから入手可能です。

http://jakarta.apache.org/log4j/docs/download.html

アーカイブ・ファイルをこのサイトからダウンロードします。プラットフォームに合ったフォーマット（ZIP ファイルまたは圧縮済の TAR ファイル）を選択し、ローカル・ファイル・システムに保存します。

注意： 次の項では、log4j 1.2.8 バージョンを使用する手順を示していますが、それ以降のバージョンの log4j でも同じ一般的な手順を使用して OC4J にデプロイできます。

---

log4j バイナリ配布版の解凍

WinZip または TAR など、プラットフォームに合ったツールを使用して、ダウンロードした log4j アーカイブ・ファイルを解凍します。これにより、次のディレクトリ構造が作成および移入されます。

```
jakarta-log4j-1.2.8/
   build/
   contrib/
      ...
   dist/
      classes/
         ...
      lib/
   docs/
      ...
   examples/
      ...
   src/
      ...
```

（一部のディレクトリ構造は示されていません。これ以外にもサブディレクトリがあります。）

log4j ライブラリのインストール

J2EE アプリケーションで log4j の機能を使用するには、log4j ライブラリが OC4J のクラスローダーによって使用できる必要があります。処理要件により、いくつかの方法があります。たとえば、log4j ライブラリをシステムまたはグローバル・アプリケーション・レベルでインストールすることにより、コンテナにデプロイされたすべてのアプリケーションで使用できるようにすることが可能です。また、log4j ライブラリを、特定のアプリケーションのライブラリとしてパッケージ化することも可能です。アプローチにより、構成ファイルの自動ロードの方法など、異なる処理特性が見られます。使用できるアプローチとその利点や弱点に関する詳細は、log4j の Web サイトおよびユーザー・メーリング・リストを参照してください。

次の各項で、log4j を OC4J で使用するための 3 種類の方法を説明します。

- log4j ライブラリのグローバル・アプリケーション・レベルでの使用
- Web アプリケーション・ライブラリとしての log4j ライブラリのパッケージ化
- EJB および Web アプリケーションの共有ライブラリとしての log4j ライブラリのパッケージ化

log4j ライブラリのグローバル・アプリケーション・レベルでの使用

log4j ライブラリを OC4J のグローバル・アプリケーション・レベルでインストールするには、log4j の `lib` ディレクトリの log4j JAR ファイル（`log4j-1.2.8.jar` など）を `j2ee/home/applib` ディレクトリにコピーします。デフォルトでは、`j2ee/home/config/application.xml` グローバル・アプリケーション・ディスクリプタの `<library>` 要素によって、OC4J インスタンスにデプロイされるすべてのアプリケーションが共有するライブラリで、このディレクトリを使用できるようになります。OC4J は、実行時に `applib` ディレクトリ内のすべてのライブラリを自動的にロードします。次に、UNIX 環境（アーカイブ・ファイルを解凍したディレクトリから）の例を示します。「`%`」はシステム・プロンプトです。

```
% cp jakarta-log4j-1.2.8/dist/lib/log4j-1.2.8.jar j2ee/home/applib
```

注意：

- この方法を使用すると、オーバーヘッドが発生します。log4j ライブラリを常時ロードしない場合は、`applib` ディレクトリを使用しないでください。
- Oracle Application Server 環境では、log4j に `applib` ディレクトリを使用しないでください。Oracle Enterprise Manager 10*g* も log4j を使用するため、コピーをグローバル・アプリケーション・レベルに置くと、Enterprise Manager とバージョンの競合が発生するおそれがあります。

Web アプリケーション・ライブラリとしての log4j ライブラリのパッケージ化

log4j ライブラリを特定の Web アプリケーション用にパッケージ化するには、log4j の `lib` ディレクトリの log4j JAR ファイル（`log4j-1.2.8.jar` など）を Web アプリケーションの `/WEB-INF/lib` ディレクトリにコピーします。Web アプリケーションのクラスローダーにより、実行時にサーブレット・コンテナが Web アプリケーションで log4j ライブラリを使用できるようにします。次に、UNIX 環境（アーカイブ・ファイルを解凍したディレクトリから）の例を示します。「`%`」はシステム・プロンプトです。

```
% cp jakarta-log4j-1.2.8/dist/lib/log4j-1.2.8.jar web-inf/lib
```

EJB および Web アプリケーションの共有ライブラリとしての log4j ライブラリのパッケージ化

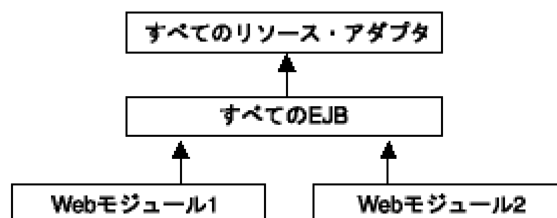
EJB コンポーネントおよび Web コンポーネントによって構成され、すべてが log4j を使用するアプリケーションが存在する場合、両方のコンポーネント・セットが使用できる 1 つの共有ライブラリとして log4j ライブラリをパッケージ化できます。

J2EE のクラスロード・メカニズムでは、EJB アプリケーションと同じ EAR ファイル内にデプロイされている Web アプリケーションは、EJB クラスローダーで使用可能なクラスにアクセスできます。log4j を EJB アプリケーションのライブラリにすると、Web アプリケーションのライブラリにもなります。

EJB クラスローダーおよび Web クラスローダーは、EAR ファイルの `META-INF/Manifest.mf` ファイルの `Class-Path` 属性で指定されているすべてのライブラリにアクセスできます。ライブラリの JAR ファイルは、`Class-Path` エントリを持つファイル（EAR ファイルなど）に対して相対的にロードされるため、そのファイルと同じディレクトリに格納されます。この機能を使用することにより、log4j の JAR ファイルを EJB の JAR ファイルと同じディレクトリ内に置き、マニフェスト・ファイルで必須ライブラリとして参照することが可能です。また、同じ EAR ファイル内の Web アプリケーションからも EJB コンポーネントのクラスを認識できるため、これらの Web アプリケーションからも log4j ライブラリにアクセスできるようになります。

図 A-1 に、J2EE アプリケーションのクラスロード階層を示します。

図 A-1 J2EE のクラスロード階層



log4j 構成ファイルの使用

log4j フレームワークを使用すると、外部構成ファイルで指定された設定によってロギング動作を制御し、アプリケーション・コードを変更せずにロギング動作に変更を加えることができます。

外部構成ファイルを使用する方法は、主に 3 種類あります。各アプローチにより、構成ファイルの名付け方、および実行時に J2EE アプリケーション・サーバーによってどのように検索されるかが異なります。

次の各項で、これら 3 種類の方法を説明します。

- デフォルト・ファイルを使用した自動 log4j 構成
- 代替ファイルを使用した自動 log4j 構成
- 外部構成ファイルをプログラム的に指定

デフォルト・ファイルを使用した自動 log4j 構成

デフォルトで、log4j はロギング動作の決定に、`log4j.properties` または `log4j.xml` という名前の構成ファイルを使用します。log4j は、実行時に使用可能なクラスローダーから、これらのファイルを自動的にロードします。両方のファイルが検出された場合、`log4j.xml` が優先されます。

自動構成ファイルを使用するには、OC4J が使用する CLASSPATH に含まれるディレクトリ内に入れます。次に、これに含まれるものを、ロードの優先順位によって示します。

1. グローバル・アプリケーション・レベル : `j2ee/home/applib`
2. Web アプリケーション・レベル : `/WEB-INF/classes`

注意 : log4j ランタイムは、`org.apache.log4j.Logger` クラスが初めてコールされた際に、自動構成ファイルを使用して 1 回のみ構成されます。log4j ライブラリを `j2ee/home/applib` ディレクトリに置くことによってグローバル・アプリケーション・レベルでインストールした場合、すべてのログ・レベル、アペンダ、およびサーバー上で実行中のすべてのアプリケーションの log4j プロパティの定義には、1 つの自動構成ファイルのみを使用できます。log4j ライブラリを各 Web アプリケーション用に、各 `/WEB-INF/lib` ディレクトリにインストールした場合、log4j ログ出力は、Web アプリケーションごとに個別に初期化されます。これにより、各 Web アプリケーションごとに、個別の自動 log4j 構成ファイルを使用できます。

代替ファイルを使用した自動 log4j 構成

log4j の自動構成に、デフォルト・ファイル名を使用するかわりに代替ファイル名を使用できます。これには、次のように、OC4J の起動時に追加実行時プロパティを指定します。「`%`」はシステム・プロンプトで、*`url`* は使用する構成ファイルの位置を指定します。

```
% java -Dlog4j.configuration=url
```

`log4j.configuration` プロパティの指定値が絶対 URL の場合、log4j は URL を直接ロードし、構成ファイルとして使用します。

指定値が絶対 URL でない場合、log4j は使用可能なクラスローダーからロードする構成ファイルの名前として指定値を使用します。

たとえば、OC4J を次のように起動したとします（折り返された 1 行のコマンドラインであるとします）。

```
% java -Dlog4j.debug=true -Dlog4j.configuration=file:///d:¥temp¥foobar.xml
      -jar oc4j.jar
```

この場合、log4j はファイル `d:¥temp¥foobar.xml` を構成ファイルとしてロードします。

別の例として、OC4J を次のように起動したとします。

```
% java -Dlog4j.debug=true -Dlog4j.configuration=foobar.xml -jar oc4j.jar
```

この場合、log4j は使用可能なクラスローダーから foobar.xml をロードします。これは、デフォルトの自動構成ファイル log4j.xml を使用する場合と同じ仕組みですが、かわりに指定されたファイル名を使用します。

注意： この方法は、より柔軟性がありますが、すべてのデプロイされたアプリケーションの外部構成ファイルが同じ名前である必要があります。

外部構成ファイルをプログラム的に指定

自動構成ファイルのローディング・メカニズムに依存するかわりに、一部のアプリケーションでは、外部構成ファイルのロードにプログラム的なアプローチを使用します。この場合、構成ファイルのパスは、アプリケーション・コード内で直接提供されます。これにより、アプリケーションごとに異なるファイル名を使用できます。log4j ユーティリティはロギング動作を決定するために、指定された構成ファイル（XML 文書またはプロパティ・ファイル）をロードおよび解析します。

次に例を示します。

```
public void init(ServletContext context) throws ServletException
{
  // Load the barfoo.xml file as the log4j external configuration file.
  DOMConfigurator("barfoo.xml");
  logger = Logger.getLogger(Log4JExample.class);
}
```

この場合、log4j は OC4J の起動ディレクトリから barfoo.xml をロードします。

プログラム的なアプローチを使用すると、開発者やシステム管理者は最大の柔軟性が得られます。構成ファイルには任意の名前を使用でき、任意の位置からロードできます。この場合でも、システム管理者は、アプリケーション・コードを変更せずに、外部構成ファイルによってロギング動作を変更できます。

さらに柔軟性を確保し、アプリケーションに特定の名前と位置をコードしない場合は、ファイル名と位置を標準 web.xml デプロイメント・ディスクリプタ内のパラメータとして提供すると便利です。サーブレットまたは JSP ページは、構成ファイルの位置と名前を指定するパラメータ値を読み取り、構成ファイルのロード元の位置を動的に構成します。このプロセスにより、システム管理者は、使用する構成ファイルの名前と位置の両方を選択できます。

次に、構成ファイルの名前と位置を指定している web.xml エントリの例を示します。

```
<context-param>
  <param-name>log4j-config-file</param-name>
  <param-value>/web-inf/classes/app2-log4j-config.xml</param-value>
</context-param>
```

アプリケーションは、デプロイメント・ディスクリプタから位置の値を読み取り、ローカル・ファイル・システム上でファイルへのフルパスを構成し、ファイルをロードします。次に、サンプル・コードを示します。

```
public void init(ServletContext context) throws ServletException
{
  /*
   * Read the path to the config file from the web.xml file,
   * should return something line /web-inf/xxx.xml or web-inf/classes/xxx.xml.
   */
  String configPath = context.getInitParameter("log4j-config-file");

  /*
   * This loads the file based on the base directory of the web application
   * as it is deployed on the application server.
```

```
  */
  String realPath = context.getRealPath(configPath);
   if(realPath!=null)
   DOMConfigurator.configure(realPath);
  _logger = Logger.getLogger(Log4JExample.class);
}
```

注意： 動作を定義するファイルで、HTTP リクエストからクライアントがアクセスすべきでないものは、Web アプリケーションの /WEB-INF ディレクトリに置くことをお薦めします。（/WEB-INF のサブディレクトリは使用しないでください。）たとえば、log4j.xml がこれに該当します。サーブレット仕様では、/WEB-INF ディレクトリの内部はクライアントからアクセス不可である必要があります。

log4j デバッグ・モードの使用可能化

log4j および外部構成ファイルを使用するアプリケーションを OC4J にデプロイする際、log4j がリクエストされた構成ファイルを検索およびロードする方法を表示すると便利な場合があります。これを行うため、log4j ではデバッグ・モードが用意されています。デバッグ・モードを使用すると、構成ファイルのロード方法が表示されます。

log4j のデバッグ・モードをオンにするには、次のようにして、OC4J の起動時に追加実行時プロパティを指定します（「%」はシステム・プロンプトです）。

```
% java -Dlog4j.debug=true -jar oc4j.jar
```

OC4J により、次のような出力が表示されます。

```
Oracle Application Server (10.1.2.0.0) Containers for J2EE initialized
log4j: Trying to find [log4j.xml] using context classloader [ClassLoader:
[[D:¥myprojects¥java¥log4j¥app1¥webapp1¥WEB-INF¥classes],
[D:¥myprojects¥java¥log4j¥app1¥webapp1¥WEB-INF¥lib/log4j-1.2.7.jar]]].
log4j: Using URL [file:/D:/myprojects/java/log4j/app1/webapp1/WEB-INF/classes/
log4j.xml] for automatic log4j configuration.
log4j: Preferred configurator class: org.apache.log4j.xml.DOMConfigurator
log4j: System property is :null
log4j: Standard DocumentBuilderFactory search succeded.
log4j: DocumentBuilderFactory is: oracle.xml.jaxp.JXDocumentBuilderFactory
log4j: URL to log4j.dtd is [classloader:/org/apache/log4j/xml/log4j.dtd].
log4j: debug attribute= "null".
log4j: Ignoring debug attribute.
log4j: Threshold ="null".
log4j: Level value for root is  [debug].
log4j: root level set to DEBUG
log4j: Class name: [org.apache.log4j.FileAppender]
log4j: Setting property [file] to [d:/temp/webapp1.out].
log4j: Setting property [append] to [false].
log4j: Parsing layout of class: "org.apache.log4j.PatternLayout"
log4j: Setting property [conversionPattern] to [%n%-5p %d{DD/MM/yyyy}
d{HH:mm:ss} [%-10c] [%r]%m%n].
log4j: setFile called: d:/temp/webapp1.out, false
log4j: setFile ended
log4j: Adding appender named [FileAppender] to category [root].
```

注意： 外部構成ファイルで log4j:configuration タグの debug 属性を使用して、デバッグ出力を使用することも可能です。ただしこの場合は、実行されたロード処理は表示されないため、構成ファイルのロード時の問題解決には不向きです。

B

サード・パーティ・ライセンス

この付録には、Oracle Application Server に付属するすべてのサード・パーティ製品のサード・パーティ・ライセンスが含まれます。この付録には次の項目が含まれています。

- Apache HTTP Server

Apache HTTP Server

Apache のライセンス条件に基づき、Oracle は次のライセンス文書を表示することが求められています。ただし、Oracle プログラム（Apache ソフトウェアを含む）を使用する権利は、この製品に付随する Oracle プログラム・ライセンスによって決定され、次のライセンス文書に含まれる条件でこの権利が変更されることはありません。反対の内容が Oracle プログラム・ライセンス内にあった場合でも、Apache ソフトウェアは現状のままで Oracle から提供されるものであり、いかなる種類の保証またはサポートも Oracle または Apache から提供されません。

The Apache Software License


```
/* ====================================================================
 * The Apache Software License, Version 1.1
 *
 * Copyright (c) 2000-2002 The Apache Software Foundation.  All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. The end-user documentation included with the redistribution,
 *    if any, must include the following acknowledgment:
 *       "This product includes software developed by the
 *        Apache Software Foundation (http://www.apache.org/)."
 *    Alternately, this acknowledgment may appear in the software itself,
 *    if and wherever such third-party acknowledgments normally appear.
 *
 * 4. The names "Apache" and "Apache Software Foundation" must
 *    not be used to endorse or promote products derived from this
 *    software without prior written permission. For written
 *    permission, please contact apache@apache.org.
 *
 * 5. Products derived from this software may not be called "Apache",
 *    nor may "Apache" appear in their name, without prior written
 *    permission of the Apache Software Foundation.
 *
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED.  IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR
 * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
 * USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 * ====================================================================
 *
 * This software consists of voluntary contributions made by many
 * individuals on behalf of the Apache Software Foundation.  For more
 * information on the Apache Software Foundation, please see
```

```
* <http://www.apache.org/>.
*
* Portions of this software are based upon public domain software
* originally written at the National Center for Supercomputing Applications,
* University of Illinois, Urbana-Champaign.
*/
```

索引

A

admin.jar ユーティリティ
  bindWebApp コマンド， 5-27
  アンデプロイ・コマンド， 5-35
  管理ユーザー / パスワード， 5-23
  再デプロイ・コマンド， 5-35
  デプロイ・コマンド， 5-26
AJP（Apache JServ Protocol）， 2-23
AJP（Apache JServ Protocol）、セキュアな， 2-32
AJPS， 2-32
Apache Jakarta プロジェクトのオープン・ソース・フレームワーク、構成および使用， A-1
Apache JServ Protocol（AJP）， 2-23
Apache JServ Protocol（AJP）、セキュアな， 2-32
Application Server Control
  J2EE「アプリケーションのデプロイ」ページ， 7-4
  OC4J の「アプリケーション」ページ， 7-3
  OC4J の「管理」ページ， 7-6
  OC4J ホームページ， 7-3
  「Web アプリケーションのデプロイ」ページ， 7-5
  「Web サイト・プロパティ」ページ， 7-7
  Web モジュールの「拡張プロパティ」ページ， 7-15
  Web モジュールの「環境」ページ， 7-14
  Web モジュールの「フィルタ処理とチェーン」ページ， 7-13
  Web モジュールの「プロパティ」ページ， 7-9
  Web モジュールの「マッピング」ページ， 7-11
  「Web モジュール」ページ， 7-8
  概要， 7-2
application.xml 構成ファイル
  OC4J/ グローバル， 5-12
  標準 / アプリケーション・レベル， 5-12
  例， 4-12, 5-29
autoencode-xxx 属性（未サポート）， 6-9
autoreload-jsp-xxx 属性（未サポート）， 6-2

C

Cookie， 1-6
Cookie、サーブレットでの使用， 2-25

D

DCM， 5-38
DCM（Distributed Configuration Management）， 5-5
dcmctl ユーティリティ、Oracle Application Server， 5-2, 5-3, 5-37
default-web-site.xml 構成ファイル
  DTD， 6-27
  階層表現， 6-29
  サンプル・ファイル， 6-29
  要素の説明， 6-19
destroy() サーブレット・メソッド， 1-3, 2-5
Distributed Configuration Management（DCM）， 5-5, 5-38
DMS（ダイナミック・モニタリング・サービス）， 2-13
doDelete() サーブレット・メソッド， 1-3
doFilter() フィルタ・メソッド， 3-2
doGet() サーブレット・メソッド， 1-3
doPost() サーブレット・メソッド， 1-3
doPut() サーブレット・メソッド， 1-3

E

EAR ファイル
  構造， 5-21
  デプロイ、スタンドアロン， 5-26
ejb-jar.xml 構成ファイル
  サーブレットの EJB コール用， 4-12, 4-19, 4-22
enable-jsp-dispatcher-shortcuts フラグ， 6-4
Enterprise Manager
  Application Server Control Web モジュールの「拡張プロパティ」ページ， 7-15
  Application Server Control Web モジュールの「環境」ページ， 7-14
  Application Server Control Web モジュールの「フィルタ処理とチェーン」ページ， 7-13
  Application Server Control Web モジュールの「プロパティ」ページ， 7-9
  Application Server Control Web モジュールの「マッピング」ページ， 7-11
  Application Server Control、概要， 7-2
  Application Server Control の J2EE「アプリケーションのデプロイ」ページ， 7-4
  Application Server Control の OC4J の「アプリケーション」ページ， 7-3
  Application Server Control の OC4J の「管理」ページ， 7-6
  Application Server Control の OC4J ホームページ， 7-3
  Application Server Control の「Web アプリケーションのデプロイ」ページ， 7-5
  Application Server Control の「Web サイト・プロパティ」ページ， 7-7
  Application Server Control の「Web モジュール」

ページ， 7-8
Web モジュールの構成， 7-2

G

GET、HTTP リクエスト， 2-5
getServletInfo() サーブレット・メソッド， 1-3, 2-5
global-web-application.xml 構成ファイル
DTD， 6-13
階層表現， 6-17
概要， 5-16
サンプル・ファイル， 6-18
要素の説明， 6-2

H

HttpServlet クラス， 1-3
HttpSessionAttributeListener インタフェース， 3-16
HttpSessionBindingEvent クラス， 3-16
HttpSessionEvent クラス， 3-15
HttpSessionListener インタフェース， 3-15
HttpSession インタフェース， 1-6
http-web-site.xml 構成ファイル
DTD， 6-27
階層表現， 6-29
要素の説明， 6-19

I

init() サーブレット・メソッド， 1-3, 2-5

J

J2EE， 1-3
JAAS， 1-3
Jakarta オープン・ソース・フレームワーク、構成および使用， A-1
Java Object Cache、オブジェクトの共有， 2-9
JDK 1.4 の考慮事項， 2-13
JMS， 1-3
JNDI， 1-3
JSP パラメータ
jsp-cache-directory， 6-4
jsp-cache-tlds， 6-4
jsp-print-null， 6-4
jsp-taglib-locations， 6-4
jsp-timeout， 6-4
simple-jsp-mapping， 6-4
JTA， 1-3

L

load-on-startup、OC4J， 2-6

M

mod_oc4j モジュール， 1-5

O

OC4J スタンドアロンのデフォルトのアプリケーション， 5-24
OC4J の起動， 5-24
OC4J の停止， 5-24
ODL（Oracle Diagnostic Logging）， 2-15
OPMN， 2-16, 5-38
Oracle Diagnostic Logging（ODL）， 2-15
Oracle Enterprise Manage、「Enterprise Manager」を参照
Oracle Process Management and Notification（OPMN）， 2-16, 5-38
orion-application.xml 構成ファイル
概要， 5-13
例， 5-30
orion-web-app 要素、構成， 6-2
orion-web.xml 構成ファイル
DTD， 6-13
階層表現， 6-17
概要， 5-16
要素の説明， 6-2
例， 5-30

P

POST、HTTP リクエスト， 2-5

R

RMI， 1-3

S

Secure Socket Laye、「SSL」を参照
server.xml 構成ファイル， 4-13
server.xml ファイル（構成）， 5-9
service() サーブレット・メソッド， 1-3
ServletContextAttributeEvent クラス， 3-15
ServletContextAttributeListener インタフェース， 3-15
ServletContextEvent クラス， 3-15
ServletContextListener インタフェース， 3-15
SSL， 2-32
Struts（Apache Jakarta プロジェクト）、構成および使用， A-2

U

URL の構成要素、サマリー， 2-19
URL リライティング， 1-6, 2-26

W

WAR ファイル
構造， 5-21
デプロイ、スタンドアロン， 5-31
web-app 要素、構成， 6-13
web-site 要素、構成， 6-19
web.xml 構成ファイル
イベント・リスナーの宣言， 3-13
概要および例， 5-15
サーブレットの EJB コール用， 4-11, 4-18
例， 5-30
Web サイト・ディスクリプタ， 5-18
Web サイトの XML 構成ファイル
DTD， 6-27
Web モジュールの Web サイトへのバインド， 4-13
階層表現， 6-29

概要， 5-18
マッピング先とマッピング元， 5-19
要素の説明， 6-19
Web ディスクリプタ， 5-15, 5-17
Web モジュールと Web アプリケーション， 1-2

あ

アプリケーション構造， 5-20
アプリケーション・ディスクリプタ， 5-11, 5-14
アプリケーションのパッケージング， 5-20
アンデプロイ
Enterprise Manager を使用した Oracle Application Server での， 5-40
スタンドアロン， 5-35
イベント・リスナー
イベント・カテゴリ， 3-12
イベント・リスナー・インタフェース， 3-12
概要， 1-9
コーディングとデプロイのガイドライン， 3-14
サンプル・コード， 3-16
宣言、起動、web.xml の使用， 3-13
代表的な使用例， 3-13
メソッドと関連クラス， 3-14
インクルード（別のサーブレットのインクルード）， 2-10
エキスパート・モード、構成およびデプロイ， 5-5
オープン・ソース・フレームワークおよびユーティリティ， A-1

か

キャッシュ、Java Object Cache オブジェクトの共有， 2-9
クラスタリング（OC4J）， 6-7, 6-19
クラスロード、サーブレット
OC4J クラスの再ロード， 2-7
OC4J サーブレット間におけるキャッシュ内の Java オブジェクトの共有， 2-9
システム・クラスより前の WAR クラスのロード， 2-9
構成
global-web-application.xml， 6-2
orion-web-app 要素， 6-2
orion-web.xml， 6-2
server.xml ファイル， 5-9
web-app 要素， 6-13
web-site 要素， 6-19
Web サイト・ディスクリプタ， 5-18
Web ディスクリプタ， 5-15, 5-17
アプリケーション・ディスクリプタ， 5-11, 5-14
概要、OC4J および J2EE 構成ファイル， 5-6
サーブレット起動用， 2-23
コード・テンプレート， 2-5
コンテキスト・パス， 2-19
コンテナ、サーブレット， 1-4

さ

サーブレット・インタフェース， 1-3
サーブレットからの EJB コール
アプリケーション外部のリモート・ルックアップ， 4-19
アプリケーション内のリモート・ルックアップ， 4-14
同じ場所への配置， 4-7
構成， 4-11, 4-18, 4-22
サーブレットおよび EJB の使用例， 4-7
デプロイ， 4-11
リモート・フラグの使用， 4-14
ルックアップ・カテゴリ， 4-7
ローカル（同じ場所に配置されている）ルックアップ， 4-8
ローカル・インタフェースとリモート・インタフェース， 4-8
サーブレット構成オブジェクト， 1-8
サーブレット・コンテキスト， 1-6
サーブレット・コンテナ， 1-4
サーブレット・チェーン， 6-8
サーブレットでの JDBC， 4-2
サーブレットでのスレッド・モデル， 2-11
サーブレットと EJB の同じ場所への配置， 4-7
サーブレットの起動
OC4J スタンドアロン， 2-24
OC4J によるフロントエンド・ホストの使用， 2-23
OHS 用のコンテキスト・パス・ルーティング， 2-23
Oracle Application Server 本番環境， 2-23
URL の構成要素のサマリー， 2-19
名前による（OC4J 固有）， 2-22
サーブレットのベスト・プラクティス， 2-39
サーブレット・パス， 2-19
サーブレット・フィルタ
HelloWorldFilter， 3-4
JSP ページの使用， 3-5
概要， 3-2
サーブレット・コンテナによる起動， 3-2
汎用コード， 3-4
フィルタの例 1， 3-4
フィルタの例 2， 3-6
フィルタの例 3， 3-8
再デプロイ
Enterprise Manager を使用した Oracle Application Server での， 5-40
アプリケーションの再ロードのトリガー、スタンドアロン， 5-36
手動による WAR の再デプロイ、スタンドアロン， 5-35
スタンドアロン， 5-35
サンプル・サーブレット
EJB のローカル・ルックアップ， 4-8
HelloWorldServlet， 1-9
JDBC 問合せ， 4-2
アプリケーション外部の EJB リモート・ルックアップ， 4-19
アプリケーション内の EJB リモート・ルックアップ， 4-14
イベント・リスナー， 3-16
セッション・サーブレット， 2-30
デモの場所、OTN， 1-1
フィルタの例 1， 3-4
フィルタの例 2， 3-6
フィルタの例 3， 3-8
シャットダウン、OC4J， 5-24
出力バッファ・サイズ， 6-3
シングルスレッド・モデル、サーブレット， 2-11
事前ロード、OC4J でのサーブレット， 2-6

自動エンコーディング（未サポート），2-27
セキュリティ
  OC4J および OHS での証明書の使用，2-33
  OC4J および OHS の構成，2-36
  SSL の一般的な問題と解決策，2-37
  SSL のデバッグ，2-38
  概要，2-32
  クライアント認証のリクエスト，2-35
  その他の考慮事項，2-38
セッション
  session-tracking 要素，6-8
  概要，1-5
  キャンセル，2-28
  詳細と例，2-24
  状態のレプリケーション，2-28
  セッション・サーブレットの例，2-30
  タイムアウト，2-28
  トラッキング，1-6, 2-25
  トラッキング、OC4J での，2-26
セッション状態のレプリケーション，2-28
セッションのキャンセル，2-28
セッションのタイムアウト，2-28
セッションのトラッキング，2-25

た

ダイナミック・モニタリング・サービス（DMS），2-13
チェーン、サーブレット，6-8
転送（別のサーブレットへの転送），2-10
テンプレート、サーブレット・コード，2-5
データ・ソース、OC4J，4-2
デバッグ
  JDeveloper を使用，2-17
  OC4J のデバッグ・フラグ，2-16
  Oracle Application Server でのタイミング上の考慮事項，2-17
  一般的な SSL のデバッグ，2-38
デフォルトの Web アプリケーション
  OC4J スタンドアロン，5-24
  Oracle Application Server での，5-39
  デプロイ、スタンドアロン，5-33
デプロイ
  EAR および WAR 構造，5-21
  EAR ファイル、スタンドアロン，5-26
  EJB サンプル・サーブレット，4-11
  J2EE アプリケーション構造への、スタンドアロン，5-30
  JDBC サンプル・サーブレット，4-5
  OC4J スタンドアロンの場合の方法，5-22
  Oracle Application Server への、概要，5-38
  WAR ファイル、スタンドアロン，5-31
  Web モジュール・ディレクトリ構造への、スタンドアロン，5-33
  アプリケーション構造，5-20
  アプリケーションのパッケージング，5-20
  管理ユーザー / パスワード，5-23
  概要、スタンドアロンと Oracle Application Server，5-2
  サンプル、EAR ファイル，5-28
  ツールとエキスパート・モード，5-5
  方法の概要，5-4
デモの場所、OTN，1-1

な

認証
  「セキュリティ」も参照
  転送 / インクルード・ターゲットの無効化，2-11

は

バッファ・サイズ、出力バッファ，6-3
パフォーマンス、サーブレット，2-12
フィルタ
  HelloWorldFilter，3-4
  JSP ページの使用，3-5
  概要，1-8, 3-2
  サーブレット・コンテナによる起動，3-2
  汎用コード，3-4
  フィルタの例 1，3-4
  フィルタの例 2，3-6
  フィルタの例 3，3-8
フロントエンド・ホスト、OC4J 機能，2-23
分散アプリケーション，2-28

ら

ライフ・サイクル、サーブレット，2-6
リスナー、「イベント・リスナー」を参照
リモート・フラグ、サーブレットと EJB コール用，4-14
ロギング
  log4j（Apache Jakarta プロジェクト）、構成および使用，A-7
  OC4J のログ・ファイル，2-14
  ODL（Oracle Diagnostic Logging），2-15
  追加の Oracle Application Server ログ・ファイル，2-16