Oracle® Containers for J2EE

サーブレット開発者ガイド

10*g*（10.1.3.1.0）

部品番号：B31859-01

2006 年 12 月

ORACLE®

Oracle Containers for J2EE サーブレット開発者ガイド , 10*g*（10.1.3.1.0）

部品番号 : B31859-01

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

原本部品番号 : B28959-01

原本著者 : Alfred Franci

原本協力者 : Bonnie Vaughan, Brian Wright, Tim Smith, Dana Singleterry, Olaf Heimburger, James Kirsch, Bryan Atsatt, Ashok Banerjee, Bill Bishop, Olivier Caudron, Cania Chung, Gerald Ingalls, 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, Tug Grall, Mike Lehmann, Steve Button

Copyright © 2002, 2006 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 USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

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

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

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

目次

はじめに ix

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

1 サーブレットの概要

サーブレットおよび J2EE テクノロジのサマリー 1-2
サーブレットの基礎知識 1-2
サーブレットを使用する理由 1-3
サーブレットのライフ・サイクル 1-4
JSP ページおよびその他の J2EE コンポーネント・タイプ 1-4
サーブレット・モデルの主要コンポーネントおよび API 1-5
サーブレット・インタフェースの主要メソッド 1-5
サーブレット通信 : 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 1-6
HttpServletRequest インタフェースの主要メソッド 1-6
HttpServletResponse インタフェースの主要メソッド 1-8
サーブレット・コンテナでのサーブレット実行 1-9
サーブレット構成オブジェクト 1-11
サーブレット構成オブジェクトの取得 1-11
サーブレット構成の主要メソッド 1-11
サーブレット・コンテキスト : アプリケーション・コンテナ 1-11
サーブレット・コンテキストの基本 1-11
サーブレット・コンテキストの取得 1-12
サーブレット・コンテキストの主要メソッド 1-12
サーブレット・セッション（ユーザー・セッション）の使用目的 1-13
サーブレットのスレッド・モデル 1-13
サーブレットの機能の表 1-14

2 サーブレットのデプロイおよび起動

初期の注意事項および OC4J の使用例 2-2
OC4J 管理の概要 2-2
スタンドアロンの OC4J と Oracle Application Server 環境の OC4J 2-3
OC4J および Oracle Application Server 管理ツール 2-3
URL 構成要素のサマリー 2-4

Web アプリケーションの OC4J へのデプロイ 2-7
アプリケーション構造 2-7
WAR ファイルをデプロイする一般的な手順のサマリー 2-8
EAR ファイルをデプロイする一般的な手順のサマリー 2-9
OC4J でのサーブレットの起動 2-10
スタンドアロン OC4J 環境でのサーブレットの起動 2-10
OC4J 開発時におけるクラス名によるサーブレットの起動 2-10
Oracle Application Server 環境でのサーブレットの起動 2-11
単純なサーブレットの例のデプロイおよび起動 2-12
サーブレットの例の WAR ファイルとしてのデプロイ 2-12
web.xml ファイルの作成 2-12
WAR ファイルの作成 2-12
WAR ファイルのデプロイおよび Web アプリケーションのバインド 2-13
サーブレットの例の EAR ファイルとしてのデプロイ 2-13
web.xml ファイルおよび WAR ファイルの作成 2-14
application.xml ファイルの作成 2-14
EAR ファイルの作成 2-14
EAR ファイルのデプロイおよび組み込まれている Web アプリケーションのバインド 2-15
サーブレットの例の起動 2-15
サーブレットの事前ロード 2-16

3 サーブレット・セッションの理解および使用方法

セッション・トラッキングの概要 3-2
セッション・オブジェクト 3-2
セッション ID 3-2
Cookie および永続セッション・データ 3-3
Cookie またはセッション属性を使用する場面 3-3
OC4J でのセッション・トラッキングの使用方法 3-3
セッション・トラッキングの構成と OC4J での Cookie の有効化または無効化 3-3
OC4J がセッション・トラッキングに Cookie を使用する方法 3-4
セッション・トラッキングのための URL リライティングの使用 3-4
保護された接続によるセッション・トラッキング 3-5
サーブレットでのセッション・オブジェクトの使用方法 3-6
HttpSession メソッドのサマリー 3-6
セッション属性の追加および取得 3-7
セッション・オブジェクトの例 3-8
サーブレットでの Cookie の使用方法 3-11
Cookie の構成 3-11
Cookie メソッドのサマリー 3-12
Cookie の取得、表示および追加 3-13
Cookie の例 3-14
セッションのキャンセル 3-16
タイムアウトによるセッションのキャンセル 3-16
セッションの明示的なキャンセル 3-17

4 サーブレット・フィルタの理解および使用方法

フィルタ機能の概要 4-2
サーブレット・コンテナによるフィルタの起動 4-2
一般的なフィルタ処理 4-3
標準 Filter インタフェース 4-4
Filter インタフェースのメソッド 4-4
FilterChain インタフェースのメソッド 4-5
FilterConfig インタフェースのメソッド 4-5
フィルタの実装および構成 4-5
フィルタのコードの実装 4-6
フィルタの構成 4-7
フィルタ・チェーンの構造 4-8
単純なフィルタの例 4-8
単純なフィルタのコードの作成 4-8
ターゲット JSP ページの作成 4-9
単純なフィルタの構成 4-9
単純なフィルタの例のパッケージ化 4-9
単純なフィルタの例の起動 4-10
転送またはインクルード・ターゲットのフィルタ 4-10
web.xml の <dispatcher> 要素 4-10
転送またはインクルード・ターゲットのフィルタの構成 4-11
フィルタを使用したリクエストまたはレスポンスのラップおよび変更 4-11
レスポンス・フィルタの例 4-12
カスタム出力ストリーム・コードの作成 4-12
レスポンス・ラッパー・コードの作成 4-13
ベース・フィルタのコードの作成 4-14
レスポンス・フィルタのコードの作成 4-14
ターゲット HTML ページの作成 4-15
レスポンス・フィルタの構成 4-15
レスポンス・フィルタの例のパッケージ化 4-15
レスポンス・フィルタの例の起動 4-16
フォーム認証フィルタ 4-16

5 イベント・リスナーの理解および使用方法

イベント・リスナー機能の概要 5-2
イベント・リスナー・インタフェース 5-3
ServletContextListener メソッド、ServletContextEvent クラス 5-3
ServletContextAttributeListener メソッド、ServletContextAttributeEvent クラス 5-3
HttpSessionListener メソッド、HttpSessionEvent クラス 5-4
HttpSessionAttributeListener メソッド、HttpSessionBindingEvent クラス 5-4
HttpSessionActivationListener メソッド 5-5
HttpSessionBindingListener メソッド 5-5
ServletRequestListener メソッド、ServletRequestEvent クラス 5-6
ServletRequestAttributeListener メソッド、ServletRequestAttributeEvent クラス 5-6
イベント・リスナーの実装および構成 5-7
リスナーのコードの実装 5-7

リスナーの構成 5-8
初期ファイルに必要な物理ファイル 5-8
セッションのライフ・サイクル・リスナーの例 5-9
JSP「ようこそ」ページの作成 5-9
セッション作成サーブレットの作成 5-10
セッション無効化サーブレットの作成 5-11
セッションのライフ・サイクル・リスナーのコードの作成 5-11
セッションのライフ・サイクル・リスナーの例の構成 5-12
セッションのライフ・サイクル・リスナーの例のパッケージ化 5-13
セッションのライフ・サイクル・リスナーの例の起動 5-13

6 サーブレットの開発

基本サーブレットの作成 6-2
サーブレット・インタフェースのメソッドを実装する場面 6-2
init() メソッドをオーバーライドする場面 6-3
doGet() または doPost() メソッドをオーバーライドする場面 6-3
doPut() メソッドをオーバーライドする場面 6-3
doDelete() メソッドをオーバーライドする場面 6-3
getServletInfo() メソッドをオーバーライドする場面 6-3
destroy() メソッドをオーバーライドする場面 6-4
レスポンスの設定 6-4
単純なサーブレットの作成手順 6-5
単純なサーブレットの例 6-6
サンプル・コードの作成 6-6
サンプル・コードのコンパイル 6-6
HTML フォームおよびリクエスト・パラメータの使用方法 6-7
HTML フォームによるユーザー入力 6-8
ユーザー入力で指定されたリクエスト・パラメータ・データの表示 6-8
フォームおよびリクエスト・パラメータを使用する例の全体 6-9
URL セキュリティのための POST メソッドの使用 6-11
リクエスト・オブジェクトの情報メソッドのコール 6-12
リクエスト情報を取得する例の全体 6-12
インクルードおよび転送による他のサーブレットへのディスパッチ 6-13
インクルードおよび転送の基本 6-13
インクルードおよび転送を使用する理由 6-14
インクルードまたは転送プロセスの手順 6-14
サーブレット・インクルードの例の全体 6-15
前処理および後処理のためにフィルタを使用する場面 6-17
サーブレット通知にイベント・リスナーを使用する場面 6-17
スタック・トレースの表示方法 6-18
アプリケーションの Apache Tomcat から OC4J への移行 6-19
Tomcat から OC4J への移行の指針 6-19
概要 6-19
サーブレットの移行手順 6-20
単純なサーブレットの移行 6-20
WAR ファイルの移行 6-21
展開 Web アプリケーションの移行 6-21
フィールドからのヒント 6-22

Tomcat および OC4J での JNDI ルックアップ 6-23
Tomcat から OC4J への JSP コンパイルの問題 6-23
Tomcat から OC4J へのクラスタリングの問題 6-24
Tomcat および OC4J の基本構成 6-24
Tomcat および OC4J のネットワークの考慮事項 6-24
Tomcat および OC4J の状態永続性メカニズム 6-25
Tomcat および OC4J のレプリケーション・アルゴリズム 6-26
状態レプリケーション送信 6-26
Tomcat および OC4J でのアプリケーション設計 6-26
Tomcat および OC4J のロード・バランシング 6-26

7 サービスおよびリソース参照の注釈の使用

注釈機能の概要 7-2
注釈および挿入 7-2
OC4J の注釈 7-4
EJB 注釈 7-4
Resource 注釈 7-5
Resources 注釈 7-5
PostConstruct 注釈 7-6
PreDestroy 注釈 7-6
PersistenceUnit(s) 注釈 7-6
PersistenceContext(s) 注釈 7-7
WebServiceRef 注釈 7-7
DeclaresRoles 注釈 7-7
RunAs 注釈 7-8
注釈ルールとガイドライン 7-8
サーブレットのバージョン 2.5 のパフォーマンスへの注釈の影響 7-9
注釈の例 7-9

8 JDBC または Enterprise JavaBeans の使用方法

サーブレットでの JDBC の使用方法 8-2
JDBC を使用する理由 8-2
データソースおよびリソース参照の構成 8-2
データソースの構成 8-3
リソース参照の構成 8-4
JDBC コールの実装 8-4
データベース問合せサーブレットの例 8-5
問合せサーブレットのデータソースの構成 8-5
HTML「ようこそ」ページの作成 8-6
問合せサーブレットの作成 8-6
サーブレットおよび JNDI リソース参照の構成 8-8
問合せの例のパッケージ化 8-8
問合せの例の起動 8-9
TopLink サーブレットの例 8-9
Enterprise JavaBeans の概要 8-10
Enterprise JavaBeans を使用する理由 8-10
OC4J および Oracle Application Server での EJB のサポート 8-11

サーブレットと EJB 間のルックアップの使用例 8-11
EJB ローカル・インタフェースとリモート・インタフェース 8-12
リモート・フラグによる同一アプリケーション内のリモート・ルックアップ 8-13

9 ベスト・プラクティスおよびパフォーマンス

セッションのベスト・プラクティス 9-2
セキュリティのベスト・プラクティス 9-2
スレッド・モデルの考慮事項 9-3
カスタム・スレッド・プール 9-4
パフォーマンスのベスト・プラクティス 9-5
パフォーマンスの監視 9-6
Oracle Application Server のダイナミック・モニタリング・サービス 9-6

A Web モジュールの管理

Application Server Control コンソール トップレベルの「Web モジュール」ページ A-2
「Web モジュール」ホームページへのアクセス方法 A-2
トップレベルの「Web モジュール」ページのサマリー A-3
Application Server Control Web モジュールの構成ページ A-4
「構成プロパティ」ページ A-5
デプロイメント・ディスクリプタの表示ページ A-7
「サーブレット・マッピング」ページ A-7
「フィルタ・マッピング」ページ A-7
「リソース参照マッピング」ページ A-9
「EJB 参照マッピング」ページ A-10
「環境エントリ・マッピング」ページ A-10
「リソース参照の参照コンテキスト」ページ A-11
Web モジュールの MBean および管理のサマリー A-12
OC4J の MBean 管理の一般的な概要 A-12
OC4J の Web モジュールの MBeans のサマリー A-13

B Web モジュールの構成ファイル

Web アプリケーションの構成ファイルの概要 B-2
標準 web.xml 構成ファイル B-2
Oracle の global-web-application.xml 構成ファイル B-3
Oracle の orion-web.xml 構成ファイル B-3
Web アプリケーションの構成ファイル間の関係のサマリー B-4
orion-web.xml および global-web-application.xml の階層 B-4
orion-web.xml および global-web-application.xml の要素と属性 B-5
<access-mask> B-5
<classpath> B-6
<context-attribute> B-7
<context-param-mapping> B-7
<ejb-ref-mapping> B-8
<env-entry-mapping> B-8
<expiration-setting> B-9
<group> B-9
<host-access> B-10

<ip-access> B-10
<jazn-web-app> B-11
<lookup-context> B-13
<mime-mappings> B-14
<ojsp-init> B-14
<orion-web-app> B-18
<request-tracker> B-24
<resource-env-ref-mapping> B-24
<resource-ref-mapping> B-25
<security-role-mapping> B-26
<service-ref-mapping> B-27
<servlet-chaining> B-27
<session-tracker> B-28
<session-tracking> B-29
<user> B-31
<virtual-directory> B-31
<web-app> B-32
<web-app-class-loader> B-33

C　サード・パーティ・ライセンス

ANTLR C-2
ANTLR ライセンス C-2
Apache C-2
Apache ソフトウェア・ライセンス C-3
Apache SOAP C-7
Apache SOAP ライセンス C-7

索引

はじめに

この開発者ガイドでは、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 Containers for J2EE（OC4J）と呼ばれています。

Oracle Application Server 10*g*（10.1.3.1.0）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 2.4 の完全実装です。

この章には、次の項目が含まれます。

- 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 モジュールを使用するアプリケーションを開発する場合は、『Oracle 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。

Oracle サポート・サービスへの TTY アクセス

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

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 Containers for J2EE 開発者ガイド』

  このマニュアルでは、OC4J 上で動作するアプリケーションを作成する開発者向けの一般的な項目（クラスのロードなど）について説明します。サーブレット、EJB、JSP コンテナなどの特定コンテナに特化した内容ではありません。

- 『Oracle Containers for J2EE デプロイメント・ガイド』

  このマニュアルでは、アプリケーションを OC4J 環境にデプロイするための情報を提供し、その手順について説明します。Oracle Enterprise Manager 10*g* に付属のデプロイ・プラン・エディタについても説明しています。

- 『Oracle Containers for J2EE 構成および管理ガイド』

  このマニュアルでは、OC4J 用アプリケーションの構成および管理方法について説明します。これには、Oracle Enterprise Manager 10*g* Application Server Control コンソールの使用方法、OC4J に付属の標準準拠 MBean の使用方法、OC4J 固有の XML 構成ファイルの直接使用の方法（該当する場合）などが含まれます。

- 『Oracle 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 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 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 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。Java Authentication and Authorization Service（JAAS）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 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 Containers for J2EE リソース・アダプタ管理者ガイド』

  このマニュアルは、J2EE コネクタ・アーキテクチャ機能の概要を示します。OC4J でのリソース・アダプタの構成および監視方法についても説明しています。

Oracle Application Server Web Services のドキュメント :

- 『Oracle Application Server Web Services 開発者ガイド』

  このマニュアルでは、OC4J および Oracle Application Server での Web サービスの開発および構成について説明します。

- 『Oracle Application Server Web Services アドバンスト開発者ガイド』

  このマニュアルでは、Web サービス・アセンブリに関する高度な項目について説明します。たとえば、一般的な相互運用性の問題を診断する方法、Web サービス管理機能（信頼

性、監査、ロギングなど）を有効化する方法、Java 値型のカスタム・シリアライズの使用方法などについて説明します。

このマニュアルでは、Web Service Invocation Framework（WSIF）、Web Service Provider API、メッセージ添付および管理機能（信頼性、監査およびロギング）の使用方法についても説明しています。また、代替 Web サービス計画（転送メカニズムとしての JMS の使用など）についても説明しています。

- 『Oracle Application Server Web Services セキュリティ・ガイド』

  このマニュアルでは、OC4J および Oracle Application Server での Web サービスのセキュリティおよび構成について説明します。

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 Application Server の追加ドキュメント :

- 『Oracle Application Server 管理者ガイド』
- 『Oracle Application Server パフォーマンス・ガイド』
- 『Oracle HTTP Server 管理者ガイド』
- 『Oracle Process Manager and Notification Server 管理者ガイド』

Application Server Control コンソールで利用できる Oracle Enterprise Manager 10*g* Application Server Control のオンライン・ヘルプ・トピック。

Java サーブレットおよび JavaServer Pages モジュールに関する次の Oracle Technology Network 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 サーブレット仕様を参照してください。

`http://java.sun.com/products/servlet/download.html#specs`

Sun 社のドキュメントは次のとおりです。

- Java サーブレットに関する Web サイト :

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

- JavaServer Pages に関する Web サイト :

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

- J2EE 1.4 Javadoc（`javax.servlet` および `javax.servlet.http` サーブレット・パッケージなど）:

  `http://java.sun.com/j2ee/1.4/docs/api/index.html`

- Java プラットフォーム、Enterprise Edition 仕様バージョン 5（Java EE 5 仕様）。

  `http://java.sun.com/javaee/5`

- Enterprise JavaBeans 仕様、バージョン 3.0（EJB 仕様）。

  `http://java.sun.com/products/ejb`

- Web Services for J2EE 1.2（Web Services 仕様）。

  `http://jcp.org/en/jsr/detail?id=109`

表記規則

このマニュアルでは次の表記規則を使用します。

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

サポートおよびサービス

次の各項に、各サービスに接続するための URL を記載します。

Oracle サポート・サービス

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

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

製品マニュアル

製品のマニュアルは、次の URL にあります。

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

研修およびトレーニング

研修に関する情報とスケジュールは、次の URL で入手できます。

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

その他の情報

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

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

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

1

サーブレットの概要

Oracle 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、リソース・アダプタ用の Resource Adapter Archive（RAR）ファイル、ならびに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*（10.1.3.1.0）では、OC4J は、Java 2 Platform Enterprise Edition バージョン 1.4 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 2.4 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。）

注意： このリリースのサーブレットは、HTTP/1.1 および Java 2 Standard Edition（J2SE）1.3 以上を必要とします。

この章では、サーブレット・テクノロジの概要を示し、最後にサーブレットの機能を表にまとめます。この章には、次の項が含まれます。

- サーブレットおよび J2EE テクノロジのサマリー
- サーブレット・モデルの主要コンポーネントおよび API
- サーブレットの機能の表

サーブレットの仕様は、次のサイトを参照してください。

```
http://java.sun.com/products/servlet/reference/api/index.html
```

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/
index.html
```

サーブレットおよび J2EE テクノロジのサマリー

次の項では、サーブレットおよびその他の J2EE テクノロジの概要を説明します。

- サーブレットの基礎知識
- サーブレットを使用する理由
- サーブレットのライフ・サイクル
- 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。

サーブレットの基礎知識

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 Enterprise JavaBeans（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。

サーブレットを使用する理由

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。2-16 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済みます。

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標準 `web.xml`Web モジュール構成ファイル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。1-13 ページの「サーブレットのスレッド・モデル」を参照してください。
- クライアント・リクエストは、サーブレットの主要メソッドである `service()` 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()`（HTTP `GET` リクエストの場合）、`doPost()`（HTTP `POST` リクエストの場合）、またはその他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、第 4 章「サーブレット・フィルタの理解および使用方法」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。6-13 ページの「インクルードおよび転送による他のサーブレットへのディスパッチ」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。

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。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。

サーブレットと JSP ページは、特に密接に対応しています。サーブレットと JSP ページは、ともに「Web コンポーネント」と呼ばれ、どちらも `web.xml` ファイルを使用して構成されます。JSP ページの変換時に JSP コンテナによって作成される JSP ページ実装クラスは、実際にはサーブレットであり、他のサーブレットと同じように `javax.servlet.Servlet` インタフェースを実装しています。JSP ページとサーブレットは、Web アプリケーションを作成する際に、透過的に併用することができます。

サーブレットまたは JSP ページは、しばしば、後続の処理を実行するために EJB をコールします。一般的な J2EE アプリケーションは、ユーザー・インタフェースやユーザー・リクエストの初期処理にサーブレットまたは JSP ページを使用し、EJB をコールしてビジネス・ロジックやデータベース・アクセスを実行します。

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

サーブレット・モデルの主要コンポーネントおよび API

この項では、サーブレット・モデルの重要なコンポーネントとプログラミング・インタフェースをまとめます。この項には、次の内容が含まれます。

- サーブレット・インタフェースの主要メソッド
- サーブレット通信 : 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・コンテナでのサーブレット実行
- サーブレット構成オブジェクト
- サーブレット・コンテキスト : アプリケーション・コンテナ
- サーブレット・セッション（ユーザー・セッション）の使用目的
- サーブレットのスレッド・モデル

この項で説明する API の詳細は、次のサイトで Sun 社の javax.servlet および javax.servlet.http パッケージの Javadoc を参照してください。

http://java.sun.com/j2ee/1.4/docs/api/index.html

サーブレット・インタフェースの主要メソッド

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、およびサービスからのサーブレット・インスタンスの削除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。このクラスは、Web サイトに適した HTTP サーブレット用で、Servlet インタフェースを実装する javax.servlet.GenericServlet クラスを拡張します。

HttpServlet クラスには、次のメソッドが含まれています。これらのメソッドは、サーブレットの実行時に、必要に応じて、OC4J サーブレット・コンテナによってコールされます（この章の後半を参照）。また、これらのどのメソッドについても、必要に応じて、コードを作成してオーバーライドすることによって、サーブレットで機能を提供することができます。6-2 ページの「サーブレット・インタフェースのメソッドを実装する場面」を参照してください。

- `void init(ServletConfig config)`

  リクエストを処理できるように、サーブレットを初期化します。入力として、サーブレット構成オブジェクト（1-11 ページの「サーブレット構成オブジェクト」を参照）を取得します。サーブレットのすべての特別な起動要件に対して、このコードを実装します。

- `void destroy()`

  サービスからサーブレットを削除します。サーブレットのすべての特別なシャットダウン要件（リソースの解放など）に対して、このコードを実装します。

- `void doGet(HttpServletRequest req, HttpServletResponse resp)`

  HTTP GET リクエストを実行するには、このコードを実装します。HTTP のリクエスト・オブジェクトおよびレスポンス・オブジェクトについては、次項の「サーブレット通信 : 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」を参照してください。

- `void doPost(HttpServletRequest req, HttpServletResponse resp)`

  HTTP POST リクエストを実行するには、このコードを実装します。

- `void doPut(HttpServletRequest req, HttpServletResponse resp)`

  HTTP PUT リクエストを実行するには、このコードを実装します。

- ```
  void doDelete(HttpServletRequest req,
                HttpServletResponse resp)
  ```

  HTTP DELETE リクエストを実行します。
  ```

- `String getServletInfo()`

  サーブレットに関する情報（作成者、リリース日など）を取得します。

次のメソッドもあります。

- ```
  void service(HttpServletRequest req,
               HttpServletResponse resp)
  ```

  これは、サーブレットの主要メソッドです。HTTP リクエストを受信し、デフォルトで、そのリクエストを、定義済の適切な do*XXX*() メソッドに送信します。通常、このメソッドをオーバーライドする必要はありません。

- `ServletConfig getServletConfig()`

  初期化および起動パラメータを含むサーブレット構成オブジェクトを取得します。

サーブレット通信 : 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

前の項で説明した、HTTP 操作の `doGet()`、`doPost()`、`doPut()` および `doDelete()` を処理するサーブレット・メソッドは、HTTP リクエスト・オブジェクト（`javax.servlet.ServletRequest` インタフェースを拡張する `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 HTTP レスポンス・オブジェクト（`javax.servlet.ServletResponse` インタフェースを拡張する `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）を入力として取得します。

リクエスト・オブジェクトは、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、リクエストを受信したサーバー名などが含まれます。レスポンス・オブジェクトは、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。

HttpServletRequest インタフェースの主要メソッド

この項では、HTTP リクエスト・オブジェクトに関係のあるメソッドをまとめます。いくつかのメソッドの使用例は、6-7 ページの「HTML フォームおよびリクエスト・パラメータの使用方法」を参照してください。

次のメソッドは、`HttpServletRequest` インタフェースで定義されます。

- `HttpSession getSession()`

  このリクエストに関連付けられたクライアント・セッションを表すオブジェクトを（必要に応じて作成してから）返します。1-13 ページの「サーブレット・セッション（ユーザー・セッション）の使用目的」を参照してください。オプションで、ブールの `true` または `false` を入力して、セッションがあらかじめ存在しない場合に新しいセッションを作成するかどうかを指定できます。

- `Cookie[] getCookies()`

  セッション・トラッキングに使用される、このリクエストとともに送信された Cookie の一連の Cookie オブジェクトを返します。3-11 ページの「サーブレットでの Cookie の使用方法」を参照してください。

- `java.lang.StringBuffer getRequestURL()`

  このリクエストに使用された URL を再作成します。

- `String getContextPath()`

  このリクエストの URL のコンテキスト・パス部分を返します。これは、Web アプリケーションのサーブレット・コンテキストのルート・パスに対応します。2-4 ページの「URL 構成要素のサマリー」を参照してください。
  ```

- `String getServletPath()`

  このリクエストの URL のサーブレット・パス部分を返します。これは、標準 `web.xml` ファイルでの構成に従って、リクエストされている特定のサーブレットを起動する部分です。

- `String getRequestURI()`

  このリクエストの URL の、ホストおよびポートの後ろの部分（問合せ文字列がある場合は問合せ文字列まで）を返します。これは、通常、コンテキスト・パスおよびサーブレット・パスです。

- `String getQueryString()`

  「?」デリミタの後に、URL（該当する場合）に追加される問合せ文字列を返します。

- `String getMethod()`

  このリクエストとともに使用される `GET`、`POST` または `PUT` などの HTTP メソッドを返します。

- `String getProtocol()`

  プロトコル（通常、HTTP）と使用されているバージョンを返します。

次のメソッドは、`ServletRequest` インタフェースから継承されます。

- `String getParameter(String name)`

  名前によって指定されるリクエスト・パラメータの値を示す文字列（検出されない場合は、`null`）を返します。

- `java.util.Enumeration getParameterNames()`

  このリクエストのすべてのリクエスト・パラメータの名前を示す文字列を含む列挙オブジェクトを返します。

- `javax.servlet.ServletInputStream getInputStream()`

  リクエストのボディをバイナリ形式で取得します。

- `java.io.BufferedReader getReader()`

  リクエストのボディを文字形式で取得します。

- `String getContentType()`

  このリクエストのボディの MIME タイプを示す文字列（MIME タイプが不明の場合は、`null`）を返します。

- `void setCharacterEncoding(String charset)`

  オーバーライドしないと、このリクエストのボディおよびパラメータの解析に使用される文字コード（MIME キャラクタ・セット）をオーバーライドします。

- `String getCharacterEncoding()`

  このリクエストのボディおよびパラメータの解析に使用される文字コードを示す文字列を返します。

- `RequestDispatcher getRequestDispatcher(String path)`

  指定されたパスにあるリソースのラッパーとして使用されるリクエスト・ディスパッチャを返します。6-13 ページの「インクルードおよび転送による他のサーブレットへのディスパッチ」を参照してください。

HttpServletResponse インタフェースの主要メソッド

この項では、HTTP レスポンス・オブジェクトに関係のあるメソッドをまとめます。詳細は、6-4 ページの「レスポンスの設定」を参照してください。

次のメソッドは、`HttpServletResponse` インタフェースで定義されます。

- `void sendRedirect(String location)`

  リダイレクトについて、クライアントがリダイレクトされる代替の場所（URL）を指定します。（リダイレクションは、たとえば、ロード・バランシングのために行われます。また、ドキュメントが別の URL に移動したことなどが理由で行われます。）

- `String encodeURL(String url)`

  セッション・トラッキングについて、Cookie が無効になっている場合、指定された URL をセッションの ID によってエンコードするために URL リライティングで使用されます。エンコードされた URL を返します。3-4 ページの「セッション・トラッキングのための URL リライティングの使用」を参照してください。

- `String encodeRedirectURL(String url)`

  リダイレクトについて、`encodeURL()` と同じ機能を実行します。

- `void addCookie(javax.servlet.http.Cookie cookie)`

  セッション・トラッキングについて、Cookie が有効になっている場合、指定された Cookie をレスポンスに追加します。3-11 ページの「サーブレットでの Cookie の使用方法」を参照してください。

- `void sendError(int code)`
  `void sendError(int code, String msg)`

  エラー・レスポンスを、指定された整数エラー・コードとともにクライアントに送信します。オプションで、説明メッセージを指定することもできます。

次のメソッドは、`ServletResponse` インタフェースから継承されます。

- `javax.servlet.ServletOutputStream getOutputStream()`

  クライアントへのレスポンスにバイナリ・データを書き込むために使用することができるストリーム・オブジェクトを返します。

- `java.io.PrintWriter getWriter()`

  クライアントへのレスポンスに文字データを書き込むために使用することができる PrintWriter オブジェクトを返します。

- `void setContentType(String type)`

  このレスポンスのボディの MIME タイプを指定します。オプションで、文字コード（MIME キャラクタ・セット）を指定することもできます。たとえば、「`text/html;charset=UTF-8`」と指定します。

- `String getContentType()`

  このレスポンスのボディの MIME タイプを示す文字列を返します。このメソッドは、文字コード（MIME キャラクタ・セット）も返します（指定されている場合）。

- `void setCharacterEncoding(String charset)`

  このレスポンスのボディの文字コードを指定します。文字コードは、`setContentType()` メソッドを使用して指定することもできます。`setCharacterEncoding()` での設定は、`setContentType()` で設定されるすべての文字コードをオーバーライドします。

- `String getCharacterEncoding()`

  このレスポンスのボディの文字コードを示す文字列を返します。

サーブレット・コンテナでのサーブレット実行

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。

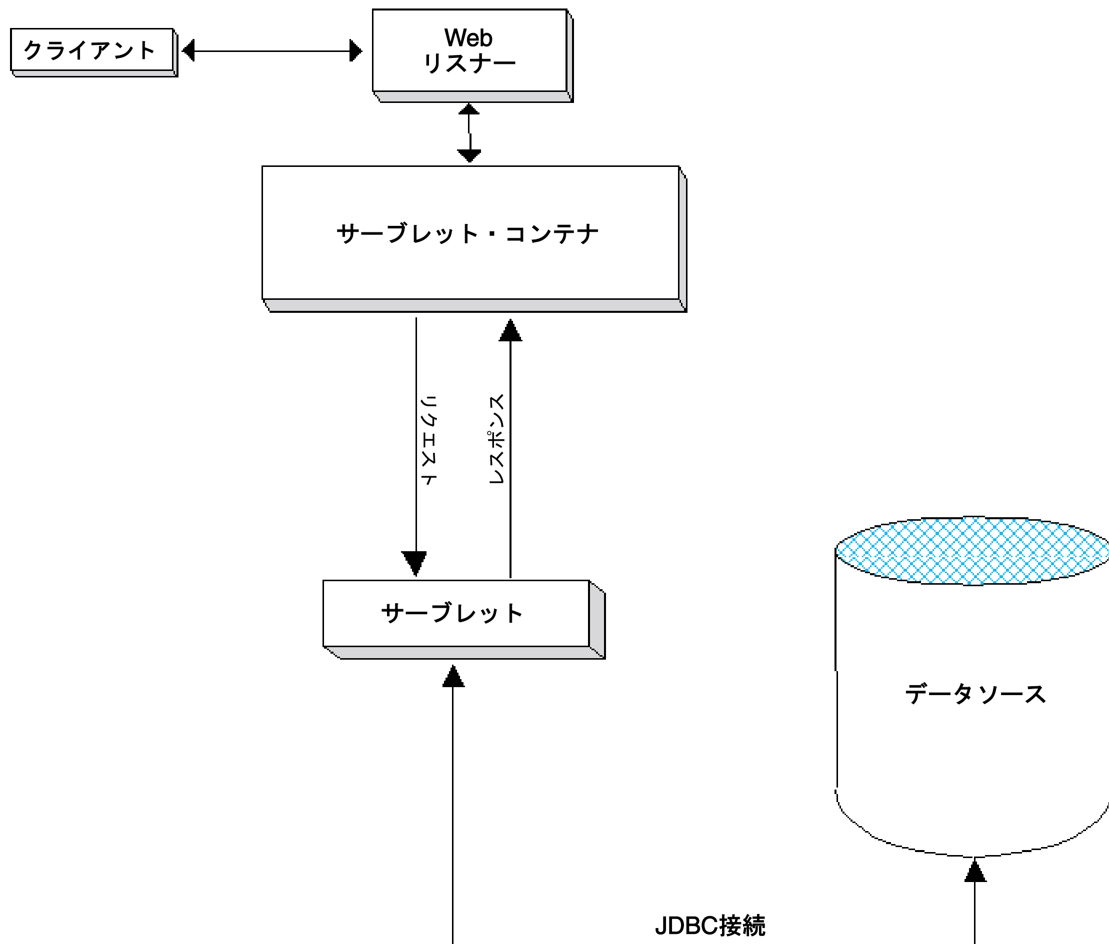
サーブレット・コンテナは、サーブレット・エンジンとも呼ばれ、サーブレットの実行と管理を行います。サーブレット・コンテナは、サーブレット・メソッドをコールし、サーブレットの実行中に必要なサービスを提供します。サーブレット・コンテナは通常 Java で作成し、Web サーバーに組み込むか（Web サーバーも Java で作成されている場合）、Web サーバーに関連付けて使用します。OC4J には、完全な標準準拠のサーブレット・コンテナが組み込まれます。

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。
- `HttpServlet` クラスに実装されている、サーブレットの `service()` メソッドを起動します。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。

図 1-1 サーブレットおよびサーブレット・コンテナ



Java オブジェクトおよび Enterprise JavaBeans（EJB）をリレーショナル・データベースに格納、および Java オブジェクトと XML 文書（JAXB）の間の変換用メカニズムを提供する Java のオブジェクトとリレーショナル間の永続性アーキテクチャである Oracle TopLink を使用することにより、サーブレットで J2EE 永続性を使用できます。TopLink を使用して、永続性およびオブジェクト変換をアプリケーションに統合する方法の詳細は、『Oracle TopLink 開発者ガイド』を参照してください。

サーブレット構成オブジェクト

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。

サーブレット構成オブジェクトの取得

サーブレットは、サーブレットの getServletConfig() メソッドを使用して、サーブレット構成オブジェクトを取得することができます。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。

サーブレットの init() メソッドは、入力として ServletConfig オブジェクトを取得します。このため、init() メソッドをオーバーライドすると、サーブレットは、サーブレット・コンテナがサーブレットの実行時に作成して渡すサーブレット構成オブジェクトにアクセスします。

サーブレット構成の主要メソッド

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。次項の「サーブレット・コンテキスト : アプリケーション・コンテナ」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)

  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・コンテキスト : アプリケーション・コンテナ

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

サーブレットは、サーブレット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することができます。1-11 ページの「サーブレット構成オブジェクト」を参照してください。

サーブレット・コンテキストの主要メソッド

`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。

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

  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)`

  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()`

  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)`

  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)`

  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()`

  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 getRequestDispatcher(String path)`

  指定されたパスにあるリソースのラッパーとして動作するリクエスト・ディスパッチャを戻します。6-13 ページの「インクルードおよび転送による他のサーブレットへのディスパッチ」を参照してください。

- `RequestDispatcher getNamedDispatcher(String name)`

  指定したサーブレットのラッパーとして動作するリクエスト・ディスパッチャを戻します。

- `String getRealPath(String path)`

  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)`

  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()`

  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()`

  `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・セッション（ユーザー・セッション）の使用目的

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・トラッキングは、ユーザー・セッションを識別し、ユーザーのすべてのリクエストをそのユーザーのセッションに適切に関連付けます。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。

標準サーブレット API では、各ユーザー・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。

詳細は、第 3 章「サーブレット・セッションの理解および使用方法」を参照してください。

サーブレットのスレッド・モデル

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・コンテナが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主要メソッドの `service()` 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。詳細は、9-3 ページの「スレッド・モデルの考慮事項」を参照してください。

サーブレットの機能の表

表 1-1 は、このマニュアルで説明しているサーブレット開発機能（前述のものを含む）と情報の相互参照をまとめたものです。サーブレット 2.4 仕様で追加された機能が注記されています。

表 1-1 最新リリースのサーブレットの機能

機能	情報
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	1-6 ページの「サーブレット通信 : 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・コンテナ	1-9 ページの「サーブレット・コンテナでのサーブレット実行」
サーブレット構成オブジェクト	1-11 ページの「サーブレット構成オブジェクト」
サーブレット・コンテキスト	1-11 ページの「サーブレット・コンテキスト : アプリケーション・コンテナ」
セッション	1-13 ページの「サーブレット・セッション（ユーザー・セッション）の使用目的」の概要、第 3 章「サーブレット・セッションの理解および使用方法」の詳細
インクルードおよび転送	6-13 ページの「インクルードおよび転送による他のサーブレットへのディスパッチ」
サーブレット・フィルタ	6-17 ページの「前処理および後処理のためにフィルタを使用する場面」の概要、第 4 章「サーブレット・フィルタの理解および使用方法」の詳細 **注意 :** インクルードまたは転送ターゲットとともにフィルタを使用する機能は、サーブレット 2.4 仕様で追加されました。
イベント・リスナー	6-17 ページの「サーブレット通知にイベント・リスナーを使用する場面」の概要、第 5 章「イベント・リスナーの理解および使用方法」の詳細 **注意 :** （サーブレット・コンテキストやセッション・リスナーとは異なる）リクエスト・リスナーのサポートは、サーブレット 2.4 仕様で追加されました。
JDBC およびデータソースの使用	8-2 ページの「サーブレットでの JDBC の使用方法」
EJB の使用	8-10 ページの「Enterprise JavaBeans の概要」

2 サーブレットのデプロイおよび起動

サーブレットをデプロイし、クライアントからサーブレットにリクエストが届くと、OC4J によりサーブレットが起動されます。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またはリクエスト・インクルード・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、サーブレットの構成およびデプロイ方法に基づく URL マッピングを通じてリクエストされます。その際には、標準 `web.xml` ファイルで指定される URL の一部（サーブレット・パス）と、デプロイ時に決定されるか標準 `application.xml` ファイルに基づく（デプロイ方法によって異なります）URL の別の部分（コンテキスト・パス）が使用されます。

次の各項で、サーブレットのデプロイと起動について説明します。

- 初期の注意事項および OC4J の使用例
- URL 構成要素のサマリー
- Web アプリケーションの OC4J へのデプロイ
- OC4J でのサーブレットの起動
- 単純なサーブレットの例のデプロイおよび起動
- サーブレットの事前ロード

初期の注意事項および OC4J の使用例

OC4J でのサーブレットのデプロイおよび起動について説明する前に、いくつかの初期注意事項と使用例をまとめます。

- OC4J 管理の概要
- スタンドアロンの OC4J と Oracle Application Server 環境の OC4J
- OC4J および Oracle Application Server 管理ツール

OC4J 管理の概要

OC4J は、J2EE 環境におけるアプリケーションのデプロイおよび管理についての次の標準をサポートしています。

- Java Management Extensions（JMX）1.2 仕様は、J2EE 環境でサービスやアプリケーションなどのリソースを管理する標準インタフェースの作成を可能にします。JMX の OC4J 実装は、OC4J サーバーやそこで動作するアプリケーションの完全な管理のために使用できるユーザー・インタフェースを提供します。
- Java 2 Platform, Enterprise Edition Management Specification（JSR-77）は、J2EE 環境のアプリケーションの実行時管理のために MBean（Managed Bean）と呼ばれるオブジェクトを作成することを可能にします。OC4J では、Oracle Enterprise Manager 10*g* のシステム MBean ブラウザを使用して MBean に直接アクセスできますが、そのプロパティの多くは、Enterprise Manager の他の機能によってユーザーによりわかりやすい方法で公開されます。
- Java 2 Enterprise Edition Deployment API Specification（JSR-88）は、J2EE 互換環境で J2EE アプリケーションとモジュールを構成およびデプロイするための標準 API を定義します。OC4J 実装には、コンポーネントを OC4J にデプロイするために必要な OC4J 固有の構成データを含むデプロイ・プランを作成または編集する機能が含まれています。

デプロイ・プランは、アーカイブを OC4J にデプロイするために必要なすべての構成データのクライアント・サイドの集約です。デプロイ・プラン・エディタを使用すると、デプロイ・プランをデプロイ中に編集できます。

OC4J デプロイ・プラン・エディタおよびシステム MBean ブラウザは、Application Server Control と呼ばれる Oracle Enterprise Manager 10*g* Application Server Control を通じて公開されます。このためのユーザー・インタフェースは、Application Server Control コンソールです。さらに、Application Server Control コンソールの他のページを通じて公開されている、MBean プロパティ（Web モジュール関連の主要プロパティを含む）に対応する多くのパラメータが役立ちます。

一般に、可能であれば、OC4J MBean または OC4J 固有の XML 構成ファイルを直接操作しないでください。XML ファイルは、Application Server Control コンソールを使用すると、OC4J によって自動的に更新されます。このため、付録 B「Web モジュールの構成ファイル」に参照情報がありますが、OC4J 固有の XML 構成については、このマニュアルには比較的少数の例しか含まれていません。ただし、デプロイの状況によっては、Application Server Control コンソールを通じて `orion-web.xml` プロパティが公開されない可能性もあります。このような状況では、XML ファイルを直接操作する以外に手段がない場合があります。

OC4J のデプロイ、構成および管理に関する一般的な情報については、『Oracle Containers for J2EE デプロイメント・ガイド』および『Oracle Containers for J2EE 構成および管理ガイド』を参照してください。Application Server Control の詳細は、『Oracle Application Server 管理者ガイド』で示されている管理ツールの概要を参照することもできます。

スタンドアロンの OC4J と Oracle Application Server 環境の OC4J

開発時は、一般に、Oracle Application Server 環境の外で OC4J を単独で使用します。これをスタンドアロン OC4J と呼びます（管理されない OC4J と呼ぶこともあります）。この使用例では、OC4J は、独自の Web リスナーを使用することができ、どの外部 Oracle Application Server プロセスによっても管理されません。

一方、完全な Oracle Application Server 環境（管理された OC4J と呼ばれることもあります）では、Web リスナーとして Oracle HTTP Server が使用され、環境を管理するために Oracle Process Manager and Notification Server（OPMN）が使用されます。

Oracle Application Server 環境とスタンドアロン環境に関する追加情報や OC4J による Oracle HTTP Server および OPMN の使用に関する追加情報は、『Oracle Containers for J2EE 構成および管理ガイド』を参照してください。

Oracle HTTP Server および関連 `mod_oc4j` モジュールに関する一般的な情報は、『Oracle HTTP Server 管理者ガイド』を参照してください。（Oracle HTTP Server から OC4J サーブレット・コンテナへの接続にはこのモジュールが使用されます。）OPMN に関する一般的な情報は、『Oracle Process Manager and Notification Server 管理者ガイド』を参照してください。

OC4J および Oracle Application Server 管理ツール

Oracle Application Server 環境またはスタンドアロン環境のいずれでも、Application Server Control（概要は、2-2 ページの「OC4J 管理の概要」を参照）を使用して OC4J で J2EE アプリケーションをバインド、構成および管理できます。これは、一般に、アプリケーションを管理するための推奨される方法ですので、このマニュアルでもこの方法を重視しています。アプリケーションは、OC4J ホームページからアクセスできる「アプリケーション」タブにある Application Server Control コンソールの「デプロイ」機能を使用してデプロイすることができます。Web モジュールを構成するための Application Server Control コンソールのページについては、付録 A「Web モジュールの管理」で説明します。

スタンドアロン OC4J では、OC4J の `admin_client.jar` コマンドライン・ツールを使用して J2EE アプリケーションをデプロイおよびバインドすることができます。

また、Oracle JDeveloper ツールを使用してアプリケーションを開発する場合は、このツールを使用してアプリケーションをデプロイおよびバインドすることもできます。

場合により、特に開発時に、OC4J 固有の XML ファイルを直接操作して OC4J アプリケーションの各部を構成する必要があることもあります。このため、OC4J のドキュメント・セットにはこれらのファイルに関する参照ドキュメントが含まれています。OC4J 固有 Web モジュール構成ファイルの `global-web-application.xml`（グローバル）および `orion-web.xml`（アプリケーション・レベル）の要素と属性については、付録 B「Web モジュールの構成ファイル」で説明します。

Application Server Control コンソールまたは `admin_client.jar` ツールによるアプリケーションのデプロイおよび管理に関する一般的な情報は、『Oracle Containers for J2EE デプロイメント・ガイド』および『Oracle Containers for J2EE 構成および管理ガイド』を参照してください。Application Server Control コンソールの詳細なオンライン・ヘルプもあります。

URL 構成要素のサマリー

サーブレットのデプロイと起動について説明する前に、URL の構成要素とその値を決定するもの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*?*param1*=*value1*...

表 2-1 に、一般的な構成要素を示します。

表 2-1 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。一般には、HTTP、HTTPS、FTP、ORMI（EJB で使用）などがあります。スタンドアロン環境では、OC4J は、通常、独自の Web リスナーを通じて直接 HTTP プロトコルを使用します。Oracle Application Server 環境では、Oracle HTTP Server が Web リスナーであり、Oracle HTTP Server は Apache JServ Protocol（AJP）を使用して OC4J と通信します。ただし、AJP はエンド・ユーザーには表示されません。 OC4J Web サイトのプロトコルは、default-web-site.xml（通常）などの Web サイトの XML ファイルに含まれる <web-site> 要素の protocol 属性に反映されます。protocol="http"（HTTP の場合）または protocol="ajp13"（AJP の場合）を使用します。（これらは、通常、デフォルトで適切に設定されています。）
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サーバー・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 Web サイトのポート番号は、Web サイトの XML ファイルに含まれる <web-site> 要素の prot 属性に反映されます。スタンドアロン OC4J の場合、これは、通常、default-web-site.xml ファイルであり、デフォルトのポートは 8888 です。Oracle Application Server 環境の場合、これは、通常、default-web-site.xml ファイルですが、port 設定により、実際のポート番号は OPMN によって決定されることがあります。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。 OC4J Web サイト構成および Web サイトの XML ファイルに関する一般的な情報は、『Oracle Containers for J2EE 構成および管理ガイド』を参照してください。

表 2-1 URL の構成要素（続き）

<table>
<tr><th>構成要素</th><th>説明</th></tr>
<tr><td>contextpath（コンテキスト・ルートとも呼ばれる）</td><td>サーブレット・コンテキストの指定されたルート・パス。Application Server Control コンソールを使用して EAR ファイルをデプロイする場合、これは、2-14 ページの「application.xml ファイルの作成」に示すように、EAR ファイル内の標準 application.xml ファイルの <context-root> 要素に従います。

Application Server Control コンソールを使用して WAR ファイルをデプロイする場合は、デプロイ時にコンテキスト・パスを指定できます。admin_client.jar を使用して EAR ファイルをデプロイする場合は、アプリケーションに含まれる Web モジュールをバインドする際にコンテキスト・パスを指定します。（詳細は、『Oracle Containers for J2EE デプロイメント・ガイド』を参照してください。）

OC4J では、指定されたコンテキスト・パスは、Web サイトの XML ファイルに含まれる該当する Web モジュールの <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、デプロイ時に指定する J2EE アプリケーション名（および EAR ファイル名）を反映し、name 属性によって、指定する Web モジュール名（および WAR ファイル名）を反映します。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" />

WAR ファイルは、単独でデプロイすると、OC4J のデフォルト J2EE アプリケーションに関連付けられます。</td></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。Web モジュールの 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。

<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></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。

次の URL を例にして説明します。

http://www.example.com:*port*/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。

この例で示すように、コンテキスト・パス、サーブレット・パスおよびその他のいかなるパス情報でも、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。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：

- 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。
- 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。

Web アプリケーションの OC4J へのデプロイ

OC4J では、J2EE 仕様に従って、J2EE アプリケーションを EAR ファイルとしてデプロイすることができます。EAR ファイルのコンテンツには、J2EE アプリケーション全体の一部である Web アプリケーション（サーブレットと JSP ページの組合せ）の WAR ファイルが 0 個以上含まれます。

Web アプリケーションのみをデプロイする場合は、J2EE ラッパー・アプリケーションを効率的に定義して EAR ファイル内に WAR ファイルをパッケージするか、WAR ファイルを直接デプロイすることができます。

次の項では、標準アプリケーション構造を説明した後に、各アプローチの一般的な手順を示します。

- アプリケーション構造
- WAR ファイルをデプロイする一般的な手順のサマリー
- EAR ファイルをデプロイする一般的な手順のサマリー

ここでは、詳細ではなく、簡単な説明のみを行います。OC4J にデプロイするための固有の情報や手順は、『Oracle Containers for J2EE デプロイメント・ガイド』を参照してください。

アプリケーション構造

Java サーブレット仕様で規定されているように、標準 Web アプリケーション構造は次のようなものです。

```
root_directory/
   Static files (for example, index.html)
   JSP pages
   WEB-INF/
      web.xml
      classes/
         servlet classes (directory substructure according to Java package)
      lib/
         JAR files (libraries and dependency classes)
```

この構造は、Web アプリケーションのデプロイに使用される標準 WAR ファイルの構造に反映されます。同様に Java サーブレット仕様で規定されている標準 `web.xml` では、（特に）サーブレットおよび JSP ページを構成します。WAR ファイルおよび `web.xml` に関する追加情報は、この少し後にある「WAR ファイルをデプロイする一般的な手順のサマリー」を参照してください。

注意： OC4J 固有の Web モジュール設定の場合は、`orion-web.xml` を、`web.xml` ファイルとともに `/WEB-INF` に組み込むことができます。また、OC4J で `orion-web.xml` ファイルを自動作成し、OC4J 固有の設定に Application Server Control コンソールを使用することができます（この設定は `orion-web.xml` に反映されます）。

Java 2 Enterprise Edition 仕様で規定されているように、標準 J2EE アプリケーション構造は、次のような、Web アプリケーション構造のスーパーセットです。

```
root_directory/
   META-INF/
      application.xml
   WebModule/
      Static files (for example, index.html)
      JSP pages
      WEB-INF/
         web.xml
         classes/
            servlet classes (directory substructure according to Java package)
         lib/
```

```
        JAR files (libraries and dependency classes)
EJBModule/...
ClientModule/...
ResourceAdapterModule/...
```

この構造は、J2EE アプリケーションや WAR およびその中に含まれる他のアーカイブ・ファイルのデプロイに使用される標準 EAR ファイルの構造に反映されます。同様に Java 2 Enterprise Edition 仕様で規定されている標準 application.xml ファイルでは、J2EE アプリケーションとそのモジュール（Web モジュールなど）を構成します。EAR ファイルには、任意の WAR ファイル、EJB の JAR ファイル、アプリケーション・クライアントの JAR ファイル、およびリソース・アダプタの RAR ファイル（アプリケーションのモジュールを含む）が組み込まれます。EAR ファイルおよび application.xml に関する追加情報は、この少し後にある「EAR ファイルをデプロイする一般的な手順のサマリー」を参照してください。

注意： OC4J 固有の J2EE アプリケーション設定の場合は、orion-application.xml ファイルを、標準 application.xml ファイルとともに /META-INF に組み込むことができます。ただし、一般的には、OC4J で orion-application.xml ファイルを自動作成し、OC4J 固有の設定に Application Server Control コンソールを使用することができます（この設定は orion-application.xml に反映されます)。orion-application.xml ファイルの詳細は、『Oracle Containers for J2EE 開発者ガイド』を参照してください。

WAR ファイルをデプロイする一般的な手順のサマリー

Web アプリケーションを WAR ファイルとして直接デプロイする（WAR ファイルを J2EE EAR ファイルに組み込まない）場合は、次の一般的な手順を実行します。

1. 標準 web.xml ファイルを作成して、Web アプリケーションを構成します。web.xml ファイルは、WAR ファイルに組み込まれている必要があります。トップレベルの <web-app> 要素内で、<servlet> および <servlet-mapping> サブ要素を使用して、サーブレットおよび JSP ページを構成します。

   <servlet> 要素の <servlet-name> および <servlet-class> サブ要素と <servlet-mapping> 要素の <servlet-name> および <url-pattern> サブ要素を使用して、サーブレット・クラスを URL サーブレット・パスにマップします。任意の名前を使用できますが、サーブレット・クラスをサーブレット・パスにマップすることが目的のため、論理的な名前を使用してください。

   ```
   <web-app>
   ...
     <servlet>
       <servlet-name>servletname</servlet-name>
       <servlet-class>package.Classname</servlet-class>
     </servlet>
     <servlet-mapping>
       <servlet-name>servletname</servlet-name>
       <url-pattern>servletpath</url-pattern>
     </servlet-mapping>
   ...
   </web-app>
   ```

2. WAR ファイルを作成し、2-7 ページの「アプリケーション構造」に示されている Web アプリケーション構造とパラレルなディレクトリ構造を持つルート・ディレクトリから、Web アプリケーション・コンポーネントおよび web.xml ファイルを組み込みます。

3. WAR ファイルを OC4J にデプロイします。Application Server Control コンソールの「デプロイ」機能（OC4J ホームページからアクセスできる「アプリケーション」タブにあります）を使用する場合は、ここで、標準 JSR-88 準拠デプロイ・プランを提供または作成することや、オプションで編集することができます。

4. Web アプリケーションをバインドします。これは、Web アプリケーションを OC4J Web サイトに関連付け、Web アプリケーションへのアクセスで使用する URL コンテキスト・パ

スを関連付けるプロセスです。Application Server Control コンソールを使用して WAR ファイルをデプロイする場合は、バインドはデプロイ手順に含まれており、コンテキスト・パスを指定することができます。

注意： WAR ファイルを単独でデプロイする際は、`orion-application.xml` ファイルに対応するパラメータを構成することはできません。これは、WAR ファイルに `orion-application.xml` ファイルを組み込むことができないためです。これらのパラメータを構成するには、WAR ファイルを EAR ファイルにパッケージする必要があります。関連情報については、2-7 ページの「アプリケーション構造」を参照してください。

EAR ファイルをデプロイする一般的な手順のサマリー

Web アプリケーションを EAR ファイル内の WAR ファイルとしてデプロイする（WAR ファイルを直接デプロイしない）場合は、次の手順を実行します。

1. 前項「WAR ファイルをデプロイする一般的な手順のサマリー」の説明に従って、Web アプリケーションの `web.xml` ファイルおよび WAR ファイルを作成します。
2. J2EE アプリケーションを構成する標準 `application.xml` ファイルを作成します。`application.xml` ファイルは、EAR ファイルに組み込まれている必要があります。特に、デプロイする Web アプリケーションは、URL コンテキスト・パスにマップしてください。このパスで、Application Server Control は、後で Web サイトの XML ファイルに書き込まれるコンテキスト・パス情報を取得します。

   `<web>` 要素内で、`<web-uri>` サブ要素を使用して WAR ファイル名を指定し、`<context-root>` サブ要素を使用してコンテキスト・パスを指定します。

   ```
   <application>
   ...
      <module>
         <web>
            <web-uri>warname.war</web-uri>
            <context-root>contextpath</context-root>
         </web>
      </module>
   ...
   </application>
   ```

3. EAR ファイルを作成し、2-7 ページの「アプリケーション構造」に示されている J2EE アプリケーション構造とパラレルなディレクトリ構造を持つルート・ディレクトリから、アプリケーション・コンポーネントおよび `application.xml` ファイルを組み込みます。
4. EAR ファイルを OC4J にデプロイします。Application Server Control コンソールの「デプロイ」機能（OC4J ホームページからアクセスできる「アプリケーション」タブにあります）を使用する場合は、ここで、標準 JSR-88 準拠デプロイ・プランを提供または作成することや、オプションで編集することができます。
5. 起動するすべての Web アプリケーションをバインドします。これは、Web アプリケーションを OC4J Web サイトに関連付け、Web アプリケーションへのアクセスで使用する URL コンテキスト・パスを関連付けるプロセスです。Application Server Control コンソールを使用して EAR ファイルをデプロイする場合は、Web アプリケーションのバインドはデプロイ手順に含まれており、コンテキスト・パスは、前述したように、EAR ファイルで提供した標準 `application.xml` ファイルに従います。

OC4J でのサーブレットの起動

この項では、スタンドアロン OC4J 環境と Oracle Application Server 環境におけるサーブレット起動方法を説明します。また、開発および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 の特別な機能についても説明します。

- スタンドアロン OC4J 環境でのサーブレットの起動
- 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 環境でのサーブレットの起動

スタンドアロン OC4J 環境でのサーブレットの起動

スタンドアロン OC4J 環境の Web サイトでは、Oracle HTTP Server を経由せずに、OC4J Web リスナーを直接経由して HTTP プロトコルを使用します。このサイトは、`default-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 URL を解析します（この解析については、コンテキスト・パスおよびサーブレット・パスの決定方法における注意点とともに、2-4 ページの「URL 構成要素のサマリー」で説明されています）。スタンドアロン OC4J のデフォルトでは、ポートは 8888 です。このポートは、このマニュアルの多くの例で使用されます（開発者向けのマニュアルであるため）。

たとえば、コンテキスト・パスが `/mypath`、サーブレット・パスが `/myservlet` の場合、次の URL でサーブレットを起動します。

```
http://www.example.com:8888/mypath/myservlet
```

OC4J 開発時におけるクラス名によるサーブレットの起動

スタンドアロン 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またはテスト時にのみ使用してください。

OC4J のシステム・プロパティ `http.webdir.enable` が `true` に設定されている場合（デフォルトは `false`）、`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、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。）

OC4J は、`/WEB-INF/classes` ディレクトリ（パッケージに基づくサブディレクトリ内にある）で、またはコンテキスト・パスに関連付けられている Web アプリケーションの `/WEB-INF/lib` ディレクトリにある JAR ファイルで、クラスを検索します。

注意 :

- Application Server Control のデプロイ・プラン・エディタを使用して、`servlet-webdir`（`servletWebdir`）を設定できます。
- OC4J のデフォルト Web アプリケーションでクラス名による起動のメカニズムを使用するには、クラスを `j2ee/home/default-web-app/WEB-INF/classes` ディレクトリに保存し、URL でデフォルト Web アプリケーションのコンテキスト・パス（デフォルトでは「/」）を使用してください。

一般に、すべてのアプリケーションでは、`http.webdir.enable` を `true` に設定した使用例の場合、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に `orion-web.xml` が含まれていない場合や、`servlet-webdir` が設定されていない場合です。
- 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できます。
  - システム・プロパティ `http.webdir.enable` のデフォルト値 `false` を使用します。この結果、`servlet-webdir` 設定は無視されます。（OC4J のシステム・プロパティに関する一般的な情報は、『Oracle Containers for J2EE 構成および管理ガイド』を参照してください。）
  - `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` 属性については、B-18 ページの「<orion-web-app>」でも説明します。

コンテキスト・パスを `/mypath`、設定を `servlet-webdir="/servlet/"` と仮定すると、次に示す URL は、クラス名によりサーブレット `foo.bar.SessionServlet` を起動します。

```
http://www.example.com:8888/mypath/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、9-2 ページの「セキュリティのベスト・プラクティス」を参照してください。

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

Oracle Application Server 環境では、OC4J は、OC4J との通信に AJP プロトコルを使用する Oracle HTTP Server を通じてアクセスされます。（AJP はエンド・ユーザーには表示されません。）Web サイトは、`default-web-site.xml` ファイルでの設定に基づいて構成されます。（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 URL を解析します（この解析については、ポート、コンテキスト・パスおよびサーブレット・パスの決定方法における注意点とともに、2-4 ページの「URL 構成要素のサマリー」で説明されています）。

たとえば、コンテキスト・パスが `/mypath`、サーブレット・パスが `/myservlet` の場合、次の URL（および適切なポート番号）でサーブレットを起動します。

```
http://www.example.com:port/mypath/myservlet
```

単純なサーブレットの例のデプロイおよび起動

この項では、6-6 ページの「単純なサーブレットの例」で示したサーブレットをデプロイして起動します。まず、サーブレットを WAR ファイルとして直接デプロイし、次に、EAR ファイルに組み込まれた WAR ファイルとしてデプロイします。

サーブレットの例の WAR ファイルとしてのデプロイ

この項では、最初に 2-8 ページの「WAR ファイルをデプロイする一般的な手順のサマリー」で概要が示された次の手順を実行して、単純なサーブレットの例を WAR ファイルとして直接デプロイします。

1. web.xml ファイルの作成
2. WAR ファイルの作成
3. WAR ファイルのデプロイおよび Web アプリケーションのバインド

サーブレットの実行や、デプロイ仕様の一部がサーブレットの URL に反映された結果は、2-15 ページの「サーブレットの例の起動」を参照してください。

web.xml ファイルの作成

次に、単純なサーブレットの例の web.xml ファイルを示します。<servlet-class> 要素は、6-6 ページの「サンプル・コードの作成」に示されている HelloWorld.java で指定されているパッケージ名とクラス名を反映します。<url-pattern> 要素は、サーブレットの起動で使用される URL のサーブレット・パス部分として myhello を指定します。サーブレット名は、クラス名をサーブレット・パスにマップします。

```
<?xml version="1.0"?>
<!DOCTYPE web-app (doctype...)>
<web-app>
  <servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>mytest.HelloWorld</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>hello</servlet-name>
    <url-pattern>myhello</url-pattern>
  </servlet-mapping>
</web-app>
```

WAR ファイルの作成

次に、標準 Web アプリケーション構造に基づく、単純なサーブレットの例のディレクトリ構造を示します。

```
root_directory/
   WEB-INF/
      web.xml
      classes/
         mytest/
            HelloWorld.class
            HelloWorld.java
```

WAR ファイルは、この構造を反映する必要があります。WAR ファイルを手動で作成する場合は、カレント・ディレクトリがルート・ディレクトリのときに JAR ユーティリティを使用して次のコマンドを発行します（% はシステム・プロンプト）。

```
% jar cvf MyHelloWorld.war .
```

これにより、次のコンテンツと構造を持つ WAR ファイルの MyHelloWorld.war が生成されます（ここで Manifest.mf ファイルおよび META-INF ディレクトリが自動的に作成されます)。

```
META-INF/Manifest.mf
WEB-INF/web.xml
WEB-INF/classes/mytest/HelloWorld.class
WEB-INF/classes/mytest/HelloWorld.java
```

WAR ファイル名は、主要な Web コンポーネント名を反映している必要があります。この場合は、サーブレットの例のクラス名を反映しています。

WAR ファイルのデプロイおよび Web アプリケーションのバインド

WAR ファイルをデプロイします。これには、通常、Application Server Control の「デプロイ」機能（OC4J ホームページからアクセスできる「アプリケーション」タブにあります）を使用します。

Application Server Control コンソールで、次の手順を実行します。

1. WAR ファイルを指定します。
2. Application Server Control で新しいデプロイ・プランを作成します。
3. アプリケーション（Web アプリケーション）名を指定します。この名前は、WAR ファイル名を反映する必要があります。
4. 親アプリケーション（デフォルトでは OC4J のデフォルト・アプリケーション）を指定します。
5. Web アプリケーションをバインドするための Web サイト（default-web-site など）を指定します。
6. コンテキスト・パスを指定します（または、WAR ファイル名から得られるデフォルトのコンテキスト・パスを使用します)。この例では、/mycontext を指定します。

注意 : Application Server Control コンソールのかわりに admin_client.jar を使用する場合は、デプロイとバインドが別の手順であり、バインドする際にコンテキスト・パスを指定します。詳細は、『Oracle Containers for J2EE デプロイメント・ガイド』を参照してください。

サーブレットの例の EAR ファイルとしてのデプロイ

この項では、最初に 2-9 ページの「EAR ファイルをデプロイする一般的な手順のサマリー」で概要が示された次の手順を実行して、単純なサーブレットの例を EAR ファイルに組み込まれた WAR ファイルとしてデプロイします。

1. web.xml ファイルおよび WAR ファイルの作成
2. application.xml ファイルの作成
3. EAR ファイルの作成
4. EAR ファイルのデプロイおよび組み込まれている Web アプリケーションのバインド

サーブレットの実行や、デプロイ仕様の一部がサーブレットの URL に反映された結果は、2-15 ページの「サーブレットの例の起動」を参照してください。

web.xml ファイルおよび WAR ファイルの作成

WAR ファイルを直接デプロイする場合と同じように、まず、web.xml ファイルと WAR ファイルを作成します。この手順は、2-12 ページの「web.xml ファイルの作成」および 2-12 ページの「WAR ファイルの作成」で説明されています。

application.xml ファイルの作成

次に、EAR ファイルに組み込まれた WAR ファイルにサーブレットをデプロイする際に使用する、単純なサーブレットの例の application.xml ファイルを示します。Application Server Control コンソールを使用してデプロイする場合は、そこでコンテキスト・パスが決定され、後で Web サイトの XML ファイルに書き込まれます。<web-uri> 要素は、WAR ファイルの名前を示します。<context-root> 要素は、WAR ファイルの Web アプリケーション（ここではサーブレットの例）と目的の URL コンテキスト・パスの /mycontext を結び付けます。

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<!DOCTYPE web-app (doctype...)>
<application>
   <module>
      <web>
         <web-uri>MyHelloWorld.war</web-uri>
         <context-root>/mycontext</context-root>
      </web>
   </module>
</application>
```

注意： admin_client.jar を使用して、EAR ファイルをデプロイし、組み込まれている Web アプリケーションをバインドする場合は、Web アプリケーションをバインドする際にコンテキスト・パスを直接指定します。

EAR ファイルの作成

EAR ファイルを作成する前に、次の手順を実行してディレクトリ構造を準備します。

1. 目的のルート・ディレクトリから、META-INF サブディレクトリを作成します。
2. 標準 J2EE アプリケーション構造に従って、application.xml ファイルを META-INF に配置します。
3. WAR ファイルをルート・ディレクトリに配置します。

これにより、次のディレクトリ構造が作成されます。

```
root_directory/
   META-INF/
      application.xml
   MyHelloWorld.war
```

EAR ファイルを手動で作成する場合は、カレント・ディレクトリがルート・ディレクトリのときに JAR ユーティリティを使用して次のコマンドを発行します（% はシステム・プロンプト）。

```
% jar cvf MyHelloWorld.ear .
```

これにより、次のコンテンツと構造を持つ EAR ファイルの MyHelloWorld.ear が生成されます（ここで Manifest.mf ファイルが自動的に作成されます）。

```
MyHelloWorld.war
META-INF/application.xml
META-INF/Manifest.mf
```

EAR ファイルのデプロイおよび組み込まれている Web アプリケーションのバインド

EAR ファイルをデプロイします。これには、通常、Application Server Control の「デプロイ」機能（OC4J ホームページからアクセスできる「アプリケーション」タブにあります）を使用します。

Application Server Control コンソールで、次の手順を実行します。

1. EAR ファイルを指定します。
2. Application Server Control で新しいデプロイ・プランを作成します。
3. J2EE アプリケーション名を指定します。この名前は、EAR ファイル名を反映する必要があります。
4. 親アプリケーション（デフォルトでは OC4J のデフォルト・アプリケーション）を指定します。
5. Web アプリケーションをバインドするための Web サイト（`default-web-site` など）を指定します。

注意： Application Server Control コンソールのかわりに `admin_client.jar` を使用する場合は、EAR ファイルのデプロイと組み込まれている Web アプリケーションのバインドは別の手順です。詳細は、『Oracle Containers for J2EE デプロイメント・ガイド』を参照してください。

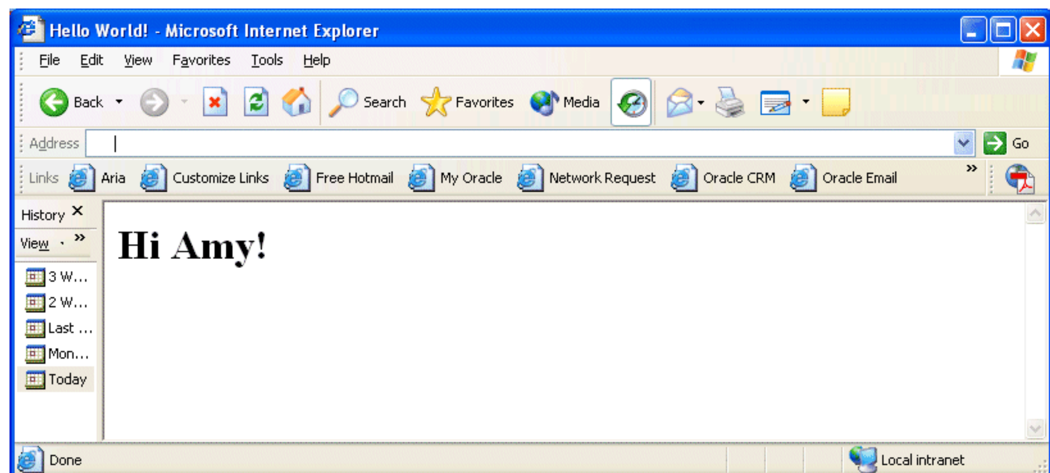
サーブレットの例の起動

前述の WAR ファイルのデプロイまたは EAR のデプロイ（どちらも同じサーブレット・パスおよびコンテキスト・パスを指定します）が完了したら、次のように、適切なホスト名を指定して、サーブレットを起動します。この URL は、OC4J のポート番号が 8888 の場合です。このポート番号は、スタンドアロン環境の OC4J のデフォルト設定です。

```
http://www.example.com:8888/mycontext/myhello
```

サーブレット・パスの `myhello` は、デプロイ時に提供した `web.xml` ファイルによって決定されます。コンテキスト・パスの `/mycontext` は、前述のように、`application.xml` ファイルによって、または Web アプリケーションのバインド時の指定によって決定されます。（コンテキスト・パスについては、2-4 ページの「URL 構成要素のサマリー」を参照してください。）

次に、サーブレットの出力を示します。



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、ブラウザ経由の直接リクエストやインクルードまたは転送などによって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、サーバーの起動後すぐにサーブレットを起動する場合は、server.xml ファイル、Web サイトの XML ファイル（default-web-site.xml など）および 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 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。（OC4J の Web サイトの XML ファイルの詳細は、『Oracle Containers for J2EE 構成および管理ガイド』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 2-2 に、web.xml の <load-on-startup> 要素の動作を説明します。

表 2-2 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。詳細は、『Oracle Containers for J2EE 開発者ガイド』を参照してください。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。

3

サーブレット・セッションの理解および使用方法

複数のリクエストおよびレスポンスについてユーザーの状態情報の追跡に使用されるサーブレット・セッションの概要は、1-13 ページの「サーブレット・セッション（ユーザー・セッション）の使用目的」で示しました。次の項では、セッション属性および Cookie の使用を含む詳細と例を提供します。

- セッション・トラッキングの概要
- OC4J でのセッション・トラッキングの使用方法
- サーブレットでのセッション・オブジェクトの使用方法
- サーブレットでの Cookie の使用方法
- セッションのキャンセル

セッション・トラッキングの概要

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。（このクラスは OC4J サーブレット・コンテナによって提供されます。）サーブレットで HTTP セッション・オブジェクトを作成する場合、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。

セッション・オブジェクトは、値（Java オブジェクト）を関連付けられたキー（Java 文字列）とともに格納したり、名前（Java 文字列）を格納したりするディクショナリと考えることができます。それぞれの名前 / 値ペアは、セッション属性を構成します。

次の項では、セッション・オブジェクトおよびその他のセッション・トラッキング機能について説明します。

- セッション・オブジェクト
- セッション ID
- Cookie および永続セッション・データ
- Cookie またはセッション属性を使用する場面

セッション・オブジェクト

サーブレットは、HTTP リクエスト・オブジェクトの `getSession()` メソッドを使用して、HTTP セッション・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。セッション・オブジェクトの属性を使用すると、`setAttribute()` および `getAttribute()` メソッドにより、ユーザー・セッションに関連するデータを格納および取得できます。3-6 ページの「サーブレットでのセッション・オブジェクトの使用方法」を参照してください。

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したり、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したりすることはできません。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。

セッション ID

特定のセッションと適切なセッション・オブジェクトとのマッピングを維持して、現在のセッションの情報に適切にアクセスできるようにするために、OC4J は、セッションごとに一意のセッション ID を生成します。ステートフル・サーブレット（基本的に、セッション・オブジェクトを作成したサーブレット）がクライアントにレスポンスを返す際に、OC4J によって生成されたセッション ID がレスポンスにインクルードされます。Cookie が有効な場合、OC4J は、セッション ID Cookie を使用してこれを実行します。3-4 ページの「OC4J がセッション・トラッキングに Cookie を使用する方法」を参照してください。

Cookie が無効な場合、セッション ID は URL リライティングによって通信され、HTTP レスポンス・オブジェクトの `encodeURL()` メソッド（インクルードまたは転送の場合は `encodeRedirectURL()` メソッド）をコールする必要があります。3-4 ページの「セッション・トラッキングのための URL リライティングの使用」を参照してください。

Cookie および永続セッション・データ

セッション・データを永続データにする場合、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機能が必要であれば、その永続データをデータベースに格納できます。データ量が少なく、データベース機能が不要なときは、独自の Cookie を作成して使用するか、多くの場合、ファイル・システムまたはその他のリモート・オブジェクトを使用することができます。

Cookie は名前と単一の関連値を持ち、HTTP のリクエスト・ヘッダーおよびレスポンス・ヘッダーの属性として格納されます。3-11 ページの「サーブレットでの Cookie の使用方法」を参照してください。

Cookie またはセッション属性を使用する場面

セッション・データは、一般に、セッション・オブジェクトの属性を使用して格納および操作されますが、データ量が少ない場合、特に永続データにしようとしている情報については、Cookie を作成して使用することもできます。セッション属性のかわりに Cookie を使用するかどうかは好みや意図の問題ですが、次の点を考慮してください。

- Cookie は、HTTP リクエストおよびレスポンスにおいてクライアントとサーバーの間で送受信されますが、セッション・オブジェクトは常にサーバー側に存在します。大量のデータをクライアントとサーバーの間で送受信することは明らかに非効率的であるため、Cookie はデータ量が少ない場合にのみ使用するようにしてください。
- Cookie はネットワークまたはインターネット経由で送受信されるため、一般に、セッション・オブジェクトよりも安全性が低くなります。

たとえば、業務関連のデータにセッション属性を使用し、プレゼンテーション関連のデータに Cookie を使用するという方針を立てることができます。Cookie は、表示内容や表示方法をサーブレットに通知し同じユーザーの後続のセッション間で変更されにくい情報（ユーザーに関する情報など）を示すことができます。

OC4J でのセッション・トラッキングの使用方法

次の項では、セッション・トラッキングのための OC4J の主な機能について説明します。

- セッション・トラッキングの構成と OC4J での Cookie の有効化または無効化
- OC4J がセッション・トラッキングに Cookie を使用する方法
- セッション・トラッキングのための URL リライティングの使用
- 保護された接続によるセッション・トラッキング

セッション・トラッキングの構成と OC4J での Cookie の有効化または無効化

OC4J は、`global-web-application.xml` または `orion-web.xml` ファイルの `<session-tracking>` 要素の設定に基づいてセッション・トラッキングを実行します。

サーブレット・コンテナは、最初に Cookie を使用してセッション・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、URL リライティング、レスポンス・オブジェクトの `encodeURL()` メソッド、もしくは転送またはインクルード用の `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。3-4 ページの「セッション・トラッキングのための URL リライティングの使用」を参照してください。

Cookies は、デフォルトでは、`<session-tracking>` の設定の `cookies="enabled"` を反映して有効になっています。`cookies="disabled"` に設定すると、Cookie は無効になります。

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

注意： Application Server Control デプロイ・プラン・エディタの sessionTracking プロパティによって、Cookie を明示的に有効または無効にすることができます（『Oracle Containers for J2EE デプロイメント・ガイド』を参照）。

また、<session-tracking> の <session-tracker> サブ要素の設定に基づいて、1 つ以上のセッション・トラッカ・サーブレットを指定することもできます。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・リスナーの詳細は、第 5 章「イベント・リスナーの理解および使用方法」を参照してください。

B-29 ページの「<session-tracking>」および B-28 ページの「<session-tracker>」も参照してください。

OC4J がセッション・トラッキングに Cookie を使用する方法

Cookie が有効な場合、OC4J は、次のように、セッション・トラッキング Cookie を使用します。

1. サーブレット・コンテナでは、セッション作成後のサーブレットへの最初のレスポンスとともに、セッション ID 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 4KB 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 Set-Cookie 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 HTTP Cookie リクエスト・ヘッダーでサーバーに返信します。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。

この Cookie を送信するためのサーブレット・コードは不要です。送信はコンテナで自動的に処理されます。Cookie のサーバーへの返信は、Web ブラウザで自動的に処理されます。

3-5 ページの「保護された接続によるセッション・トラッキング」も参照してください。

セッション・トラッキングのための URL リライティングの使用

Cookie にかわるものとして、HTTP レスポンス・オブジェクトの encodeURL() メソッド（インクルードまたは転送の場合は、同等の、encodeRedirectURL() メソッド）を使用する URL リライティングがあります。このメカニズムにより、OC4J は、Cookie が無効になっている場合に、セッション ID を URL パスにエンコードすることができます。次の条件が満たされている必要があります。

- セッションが必ず有効である。
- クライアントとの以前の交換で Cookie によってセッション ID を受け取っていない。
- URL がアプリケーション内の場所を指し示している。

サーブレット・コンテナが確実に URL リライティングを使用できるようにするために、出力ストリームに URL を書き込む際は、URL を直接書き込むのではなく、必ず、encodeURL() を使用してください。次に例を示します。

```
out.println("Click <a href=" + res.encodeURL(req.getRequestURL().toString()) +
            ">this link</a>");
out.println(" to access this page again.<br>");
```

OC4J は、次の例のように、パラメータの jsessionid（サーブレット仕様に準拠）を使用して、URL パスでセッション ID を示します。

```
http://www.example.com: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。

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。

注意：

- 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。
- 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、多くのリソースを消費する標準外の処理です。

保護された接続によるセッション・トラッキング

OC4J とクライアントの間での交換に機密情報が含まれる場合は、保護された接続を使用して送信する必要があります。これには、HTTPS を使用できます（HTTPS では SSL ソケットを使用して HTTP プロトコルが送信されます。詳細は、『Oracle Containers for J2EE セキュリティ・ガイド』を参照してください）。この場合、セッション ID が傍受されたり、なりすましが行われたりする可能性がある Cookie または URL リライティングは、セッション ID の送信には適していません。セッション ID の値の信頼性が失われると、関連するセッションが脆弱性を持ちます。

すべての送信に HTTPS が使用されるこの保護された送信状況では、OC4J は、セッション状態を取得するために必要な情報を、SSL セッションの属性（ユーザーには表示されない機能）として、直接 SSL 接続に格納します。これにより、セッション状態に対して最大限のセキュリティが提供されますが、セッション状態の存続期間も SSL 接続自体の存続期間に結び付けられます。SSL 接続が切断されると、セッション状態が失われます。

アプリケーションは、HTTP と HTTPS によって共有される可能性もあります。（詳細は、3-6 ページの「スタンドアロン OC4J の HTTP および HTTPS でアプリケーションを使用可能にする方法」を参照してください。）アプリケーションがこのように共有される場合は、OC4J が、アプリケーションへの送信を SSL 接続経由で行うことができるかどうかを判定します。セッション情報を格納できる保証された SSL 接続がない場合、OC4J は、前述のように、セッション・トラッキングに Cookie または URL リライティングを使用する状態に戻ります。

注意： 一部の古いブラウザでは、特定の状況で SSL 接続が切断され、セッション状態を継続して取得できなくなったり、場合によっては、セッション状態が突然消失することがあります。このような場合は、アプリケーションを共有するように指定して、セッション・トラッキングに Cookie または URL リライティングを強制的に使用することにより、問題を回避できます。安全性が低下しますが、これが実行可能な唯一の代替方法です。

スタンドアロン OC4J の HTTP および HTTPS でアプリケーションを使用可能にする方法

スタンドアロン OC4J において単一のアプリケーションを 2 つのプロトコル経由で使用できるようにするには、そのアプリケーションを 2 つの異なる Web サイトの XML ファイルで宣言し、共有のマークを付ける必要があります。この機能は、各 Web サイトの XML ファイルで、`<web-app>` 要素の `shared` 属性が `true` に設定されている場合に有効になります。アプリケーションを共有するように設定すると、アプリケーションが宣言されている各 Web サイトに対して定義されているプロトコル経由で単一のアプリケーションのデプロイが可能になります。HTTP と HTTPS で共有する場合、1 つの Web サイトは HTTPS 用に構成および保護され、別の Web サイトは HTTP 用に構成および保護されなくなります。

OC4J の共有アプリケーションに関する追加情報は、『Oracle Containers for J2EE 構成および管理ガイド』を参照してください。

サーブレットでのセッション・オブジェクトの使用方法

この項では、セッション・オブジェクトの属性の使用方法を示します。最初に `HttpSession` インタフェースの主要メソッドをまとめ、次に getter および setter メソッドによるセッション属性の作成および取得について説明し、最後に例を示します。

HttpSession メソッドのサマリー

サーブレット・コンテナはユーザー・セッションのデータの管理に、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 OC4J によって提供され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、セッション属性を処理します。

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

  指定したオブジェクトを指定した名前でセッション・オブジェクトに追加して、セッション属性を追加します。

- `Enumeration getAttributeNames()`

  すべてのセッション属性の名前について、`String` オブジェクトで構成される `java.util.Enumeration` オブジェクトを取得します。

- `Object getAttribute(String name)`

  指定した名前を持つセッション属性の値を取得します（名前が一致しない場合は `null` を返します）。

- `void removeAttribute(String name)`

  指定した名前を持つセッション属性を削除します。

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。または、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 setMaxInactiveInterval(int interval)`

  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。サーブレット・コンテナのデフォルトや `web.xml` の `<session-timeout>` 要素によって設定されたデフォルトはオーバーライドされます。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。3-16 ページの「セッションのキャンセル」も参照してください。

- `int getMaxInactiveInterval()`

  セッションのタイムアウト時間（秒単位）を示す値を取得します。`setMaxInactiveInterval()` によってタイムアウト値を指定している場合は、その値が返されます。それ以外の場合は、`web.xml` の `<session-timeout>` 要素の値が返されます（指定されている場合）。この値も指定されていない場合は、サーブレット・コンテナのデフォルトのタイムアウトが返されます。

- `void invalidate()`

  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。

サーブレット・コンテキストを取得するユーティリティ・メソッドや、セッションの作成やアクセスに関する情報を取得するユーティリティ・メソッドもあります。

- `ServletContext getServletContext()`

  セッションが属するサーブレット・コンテキストを取得します。

- `boolean isNew()`

  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()`

  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()`

  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。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間を返します。

開発のヒント：

`java.util.Date` クラスは、1970 年 1 月 1 日午前 0 時からの時間をミリ秒単位で示すコンストラクタを持ち、その値を、日付、時間、分および秒に変換します。

`HttpSession` メソッドの詳細は、次のサイトで Sun 社の `javax.servlet.http` パッケージの Javadoc を参照してください。

`http://java.sun.com/j2ee/1.4/docs/api/index.html`

セッション属性の追加および取得

次に、セッション属性を作成および表示するためのいくつかの一般的な手順を示します。これらの手順の例の全体は、3-8 ページの「セッション・オブジェクトの例」に示されています。

1. リクエストからセッション・オブジェクトを取得します。

```
HttpSession session = request.getSession();
```

2. ユーザー入力を取得して、各属性の名前と値を指定します。この例では、サーブレットが、属性の名前と値を取得するフォームを持ち、それらを `dataname` および `datavalue` と呼ばれるリクエスト・パラメータに格納するものとします。（フォームは、例の全体が示されている項を参照してください。）

```
String dataName = request.getParameter("dataname");
String dataValue = request.getParameter("datavalue");
```

3. 指定された名前と値を使用して各属性を設定します。

```
if (dataName != null && dataValue != null) {
    session.setAttribute(dataName, dataValue);
}
```

4. すべての属性の出力をブラウザに戻す場合は、最初に `getAttributeNames()` をコールしてすべての名前を取得してください。

```
Enumeration attributeNames = session.getAttributeNames();
```

5. java.util.Enumeration オブジェクトによる反復によってそれぞれの名前を取得し、対応する各値を取得して、名前と値を出力します。この例では、すべての名前と値を Java 文字列として処理できるものとします。

```
while (attributeNames.hasMoreElements()) {
   String name = attributeNames.nextElement().toString();
   String value = session.getAttribute(name).toString();
   out.println(name + " = " + value + "<br>");
}
```

重要： クラスタリングされたアプリケーションでは、セッション属性として使用されるすべてのオブジェクトが java.io.Serializable インタフェースを実装する必要があります。

注意： HttpSession インタフェースでは、メソッドの setAttribute()、getAttributeNames() および getAttribute() が、使用されなくなった putValue()、getValueNames() および getValue() のかわりに使用されるようになりました。

セッション・オブジェクトの例

この例は、3-7 ページの「セッション属性の追加および取得」の手順をまとめたものです。この例には、属性の名前および値のユーザー入力を取得するフォームのコードと、セッション ID、作成された時間、および最後にアクセスした時間（この場合は最後に属性が追加された時間）を表示するコードも含まれています。

前の例のように、POST リクエストについては、doGet() メソッドが doPost() メソッドによってコールされます。

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

public class SessionExample extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body bgcolor=\"white\">");

        out.println("<h3>" + "My Session Example" + "</h3>");

/* Display session ID, creation time, and access time. */

        HttpSession session = request.getSession();
        out.println("Session ID: " + session.getId());
        out.println("<br>");
        out.println("Created: ");
        out.println(new Date(session.getCreationTime()) + "<br>");
        out.println("Last accessed: ");
        out.println(new Date(session.getLastAccessedTime()));
```

```
/* Set attribute, based on data from user (form is later in code). */

        String dataName = request.getParameter("dataname");
        String dataValue = request.getParameter("datavalue");
        if (dataName != null && dataValue != null) {
            session.setAttribute(dataName, dataValue);
        }

/* Display all attributes. */

        out.println("<P>");
        out.println("The following data is in the session: <br>");
        Enumeration attributeNames = session.getAttributeNames();
        while (attributeNames.hasMoreElements()) {
           String name = attributeNames.nextElement().toString();
           String value = session.getAttribute(name).toString();
           out.println(name + " = " + value + "<br>");
        }

/* Take user input for an attribute. */

        out.println("<P>");
        out.print("<form action=\"");
        out.print("SessionExample\" ");
        out.println("method=POST>");
        out.println("Specify attribute name: ");
        out.println("<input type=text size=20 name=dataname>");
        out.println("<br>");
        out.println("Specify attribute value: ");
        out.println("<input type=text size=20 name=datavalue>");
        out.println("<br><br>");
        out.println("<input type=submit>");
        out.println("</form>");

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

    }

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

}
```

サーブレットを初めて実行すると、次のような画面が表示されます。

My Session Example

Session ID: 8223afa422b8efe590201491464ea6c1e0ad554ef4d7
Created: Wed Apr 21 20:19:15 PDT 2004
Last accessed: Wed Apr 21 20:19:15 PDT 2004

The following data is in the session:

Specify attribute name:
Specify attribute value:

Submit Query

新しい属性の名前と値を入力して **「Submit Query」** をクリックするたびに、出力が更新され、新しい名前 / 値ペアが示されます。1 つの属性の名前と値として「Customer」と「Brian」を入力し、別の属性の名前と値として「SSN」と「123-45-6789」を入力して、ペアを入力するたびに **「Submit Query」** をクリックする場合、出力は、次のように更新されます。セッション・オブジェクトの最後のアクセス時間も変更されていることに注意してください。

My Session Example

Session ID: 8223afa422b8efe590201491464ea6c1e0ad554ef4d7
Created: Wed Apr 21 20:19:15 PDT 2004
Last accessed: Wed Apr 21 20:20:36 PDT 2004

The following data is in the session:
Customer = Brian
SSN = 123-45-6789

Specify attribute name:
Specify attribute value:

Submit Query

サーブレットでの Cookie の使用方法

永続データの量が少ないときは、Cookie が有効になっていれば、独自の Cookie を作成して使用することができます。Cookie は、javax.servlet.http.Cookie クラスのインスタンスによって表され、HTTP のリクエスト・ヘッダーおよびレスポンス・ヘッダーの属性としてクライアントとサーバーの間で受け渡されます。

サーブレットは、Cookie によって次のことを実行できます。

- Cookie クラス・コンストラクタを使用して、指定された名前および値で Cookie を作成する。
- レスポンス・オブジェクトの addCookie() メソッドを使用して、Cookie をレスポンスに追加する。
- リクエスト・オブジェクトの getCookies() メソッドを使用して、リクエストから Cookie を取得する。
- Cookie クラスのメソッド（この少し後にサマリーがあります）を使用して、Cookie の名前の取得や、Cookie の値やその他の情報を取得または設定する。

次の項では、Cookie の使用方法の詳細と例の全体を提供します。

- Cookie の構成
- Cookie メソッドのサマリー
- Cookie の取得、表示および追加
- Cookie の例

注意： Cookie が有効になっている場合、OC4J は、自動的に Cookie を使用して、セッション ID を追跡し続けます。詳細は、3-4 ページの「OC4J がセッション・トラッキングに Cookie を使用する方法」を参照してください。

Cookie の構成

デフォルトでは、Cookie は有効です。ただし、3-3 ページの「セッション・トラッキングの構成と OC4J での Cookie の有効化または無効化」で説明するように、global-web-application.xml または orion-web.xml ファイルの <session-tracking> 要素で、cookies="enabled" または cookies="disabled" を設定することにより、Cookie を明示的に有効または無効にできます。

<session-tracking> 要素には、次の追加の Cookie 設定もあります。

- cookie-domain: Cookie に必要なドメインです。一般に、これを使用して、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します。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 つの部分（.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。

  Cookie ドメイン機能を使用すると、たとえば、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。

  特定の Cookie については、その Cookie の setDomain() メソッドを使用して cookie-domain 設定をオーバーライドできます。

- 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 を保存する最大期間（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。このため永続性はありません。

  特定の Cookie については、その Cookie の setMaxAge() メソッドを使用して cookie-max-age 設定をオーバーライドできます。追加情報は、次項の「Cookie メソッドのサマリー」を参照してください。

B-29 ページの「<session-tracking>」も参照してください。

注意： Application Server Control デプロイ・プラン・エディタの sessionTracking プロパティの属性によって、Cookie を構成できます。詳細は、『Oracle Containers for J2EE デプロイメント・ガイド』を参照してください。

Cookie メソッドのサマリー

新しい Cookie を作成するには、Cookie コンストラクタを使用します。

- Cookie(String name, String value)

  指定された名前および値で、Java 文字列として、Cookie を作成します。

Cookie に関する情報を指定または取得するには、次のような Cookie の getter および setter メソッドを使用します。

- String getName()

  Cookie の名前を返します。

- void setValue(String value)

  Cookie の新しい値を Java 文字列として指定します。

- String getValue()

  Cookie の現在の値を Java 文字列として返します。

- void setDomain(String value)

  デフォルトでは、Cookie は送信元のサーバーにしか返されませんが、このメソッドを使用すると、Cookie が表示されるドメインまたはドメイン・ネーム・システム・ゾーンを指定できます。OC4J での Cookie ドメインの使用については、3-11 ページの「Cookie の構成」を参照してください。

- String getDomain()

  Cookie が表示されるドメインを返します。

- void setMaxAge(int maxAge)

  Cookie が期限切れになるまでの最大期間（秒単位）を指定します。値を 0 にすると、Cookie はすぐに削除されます。値を -1 にすると、Cookie はブラウザがシャットダウンされるまで存在します。このため、永続的に格納されることはありません。デフォルトは、-1 または OC4J 構成で指定された値です。3-11 ページの「Cookie の構成」を参照してください。

- int getMaxAge()

  Cookie が期限切れになるまでの最大期間（秒単位）または特別な値（setMaxAge() に関する説明を参照）を返します。

- void setComment(String comment)

  Cookie に関する情報を提供するコメントを作成します。

- String getComment()

  Cookie に関する情報を提供するコメント（コメントがない場合は null）を返します。

Cookie メソッドの詳細は、次のサイトで Sun 社の javax.servlet.http パッケージの Javadoc を参照してください。

http://java.sun.com/j2ee/1.4/docs/api/index.html

Cookie の取得、表示および追加

次に、Cookie を作成および表示するためのいくつかの一般的な手順を示します。これらの手順の例の全体は、3-14 ページの「Cookie の例」に示されています。

1. リクエストの現在の Cookie セットを含む一連の javax.servlet.http.Cookie オブジェクトを取得するために、リクエスト・オブジェクトの getCookies() メソッドをコールします。

```
Cookie[] cookies = request.getCookies();
```

2. Cookie の名前と値を出力するために、一連のオブジェクトを反復します（一連のオブジェクトが空の場合はユーザーに通知します）。

```
if (cookies != null && cookies.length > 0) {
   for (int i = 0; i < cookies.length; i++) {
      Cookie cookie = cookies[i];
      out.print("Cookie name: " + cookie.getName() + "<br>");
      out.println("Cookie value: " + cookie.getValue() + "<br><br>");
   }
} else {
   out.println("There are no cookies.");
}
```

3. 新しい Cookie を作成するために、ユーザー入力を取得して名前と値を指定することができます。この例では、サーブレットが、名前と値を取得するフォームを持ち、それらを cookiename および cookievalue と呼ばれるリクエスト・パラメータに格納するものとします。（フォームは、例の全体が示されている項を参照してください。）

```
String cookieName = request.getParameter("cookiename");
String cookieValue = request.getParameter("cookievalue");
```

4. 新しい Cookie をレスポンスに追加するために、まず、指定された名前と値を Cookie コンストラクタに入力して Cookie を作成し、レスポンス・オブジェクトの addCookie() メソッドをコールします。

```
if (cookieName != null && cookieValue != null) {
   Cookie cookie = new Cookie(cookieName, cookieValue);
   response.addCookie(cookie);
}
```

Cookie の例

この例は、3-13 ページの「Cookie の取得、表示および追加」の手順をまとめたものです。この例には、Cookie の名前と値のユーザー入力を取得するフォームのコードも含まれています。

前の例のように、POST リクエストについては、doGet() メソッドが doPost() メソッドによってコールされます。

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

public class CookieExample extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body bgcolor=\"white\">");

        out.println("<h3>" + "My Cookies Example" + "</h3>");

/* Show cookies currently in request. */

        Cookie[] cookies = request.getCookies();
        if (cookies != null && cookies.length > 0) {
            out.println("Your browser is sending the following cookies:<br><br>");
            for (int i = 0; i < cookies.length; i++) {
                Cookie cookie = cookies[i];
                out.print("Cookie name: " + cookie.getName() + "<br>");
                out.println("Cookie value: " + cookie.getValue() + "<br><br>");
            }
        } else {
            out.println("There are no cookies.");
        }

/* Add new cookie that was just specified by user (form code below), and output
   back to the browser to confirm what the user just added.                    */

        String cookieName = request.getParameter("cookiename");
        String cookieValue = request.getParameter("cookievalue");
        if (cookieName != null && cookieValue != null) {
            Cookie cookie = new Cookie(cookieName, cookieValue);
            response.addCookie(cookie);
            out.println("<P>");
            out.println
      ("You just created the following cookie for your browser to send:<br><br>");
            out.print("New cookie name: " + cookieName + "<br>");
            out.print("New cookie value: " + cookieValue + "<br>");
        }

/* Form to prompt user for a cookie name and value. */

        out.println("<P>");
        out.println("Create a new cookie for your browser to send:<br>");
        out.print("<form action=\"");
        out.println("CookieExample\" method=POST>");
        out.print("Specify cookie name: ");
        out.println("<input type=text length=20 name=cookiename><br>");
```

```
        out.print("Specify cookie value: ");
        out.println("<input type=text length=20 name=cookievalue><br><br>");
        out.println("<input type=submit></form>");

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

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

}
```

サーブレットを初めて実行すると、リクエストにすでに存在するすべての Cookie の名前と値が出力され、新しい Cookie の追加を求められます。次の出力例では、すでに 1 つの Cookie が存在します。

My Cookies Example

Your browser is sending the following cookies:

Cookie name: ORA_UCM_VER
Cookie value: %2FMP%2F8%60pg_l%2Cb_tgb%2Cupgefr%3Emp_ajc%2CamkMP%2F8%5Ene%5Dj*%60%5Dre%60*snecdp%3Ckn%5D_ha*_kiMP%2F8pckmrcGnMP%2F8naikpaEl

Create a new cookie for your browser to send:

Specify cookie name:
Specify cookie value:

Submit Query

Cookie の名前と値を指定して「**Submit Query**」をクリックすると、サーブレットは入力内容を確認します。

My Cookies Example

Your browser is sending the following cookies:

Cookie name: ORA_UCM_VER
Cookie value: %2FMP%2F8%60pg_l%2Cb_tgb%2Cupgefr%3Emp_ajc%2CamkMP%2F8%5Ene%5Dj*%60%5Dre%60*snecdp%3Ckn%5D_ha*_kiMP%2F8pckmrcGnMP%2F8naikpaEl

You just created the following cookie for your browser to send:

New cookie name: oreo
New cookie value: creamywhitefilling1234

Create a new cookie for your browser to send:

Specify cookie name:
Specify cookie value:

Submit Query

サーブレットをリロードすると、リクエストに現在含まれている Cookie のリストに新しい Cookie が追加されたことを確認できます。

My Cookies Example

Your browser is sending the following cookies:

Cookie name: oreo
Cookie value: creamywhitefilling1234

Cookie name: ORA_UCM_VER
Cookie value: %2FMP%2F8%60pg_l%2Cb_tgb%2Cupgefr%3Emp_ajc%2CamkMP%2F8%5Ene%5Dj*%60%5Dre%60*snecdp%3Ckn%5D_ha*_kiMP%2F8pckmrcGnMP%2F8naikpaEl

Create a new cookie for your browser to send:

Specify cookie name:
Specify cookie value:

Submit Query

セッションのキャンセル

HTTP セッション・オブジェクトは、一般に、サーバー・サイド・セッションが存続する間、継続します。ただし、サーブレットによってセッションを明示的に終了することや、サーブレット・コンテナによって一定の存続期間後にセッションをキャンセルすることができます。

タイムアウトによるセッションのキャンセル

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デフォルト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。

特定のサーブレット用に OC4J のタイムアウトまたは `<session-timeout>` 設定をオーバーライドするために、セッション・オブジェクトの `setMaxInactiveInterval()` メソッドを使用できます。このメソッドは、`HttpSession` インタフェースで指定されます。整数値を指定しますが、このメソッドでは、タイムアウト値が分単位ではなく秒単位で指定することに注意してください。繰り返しますが、負の値の場合はタイムアウトになりません。

次の例は、10 分のタイムアウトを指定します。

```
HttpSession session = request.getSession();
...
session.setMaxInactiveInterval(600);
```

getMaxInactiveInterval() メソッドは、タイムアウト時間（秒単位）を示す整数を返します。これは、setMaxInactiveInterval() で設定される値です（設定されている場合）。設定されていない場合は、<session-timeout> で指定されている値（秒単位に変換したもの）です（指定されている場合）。setMaxInactiveInterval() も <session-timeout> も使用されていない場合、getMaxInactiveInterval() は、OC4J のデフォルト・タイムアウト値（秒単位）の 1200 を返します。

セッションの明示的なキャンセル

サーブレットは、次の例のように、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。

```
HttpSession session = request.getSession();
...
session.invalidate();
```

これにより、セッションは即時に無効化され、セッションにバインドされているすべてのオブジェクトのバインディングが解除されます。

4 サーブレット・フィルタの理解および使用方法

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。

これに対して、サーブレット・フィルタを使用して、Web アプリケーション・リクエストの前処理やサーバー・レスポンスの後処理を実行できます。フィルタについては、6-17 ページの「前処理および後処理のためにフィルタを使用する場面」で簡単に説明しましたが、次の項で詳しく説明します。

- フィルタ機能の概要
- 標準 Filter インタフェース
- フィルタの実装および構成
- 単純なフィルタの例
- 転送またはインクルード・ターゲットのフィルタ
- フィルタを使用したリクエストまたはレスポンスのラップおよび変更
- レスポンス・フィルタの例
- フォーム認証フィルタ

フィルタ機能の概要

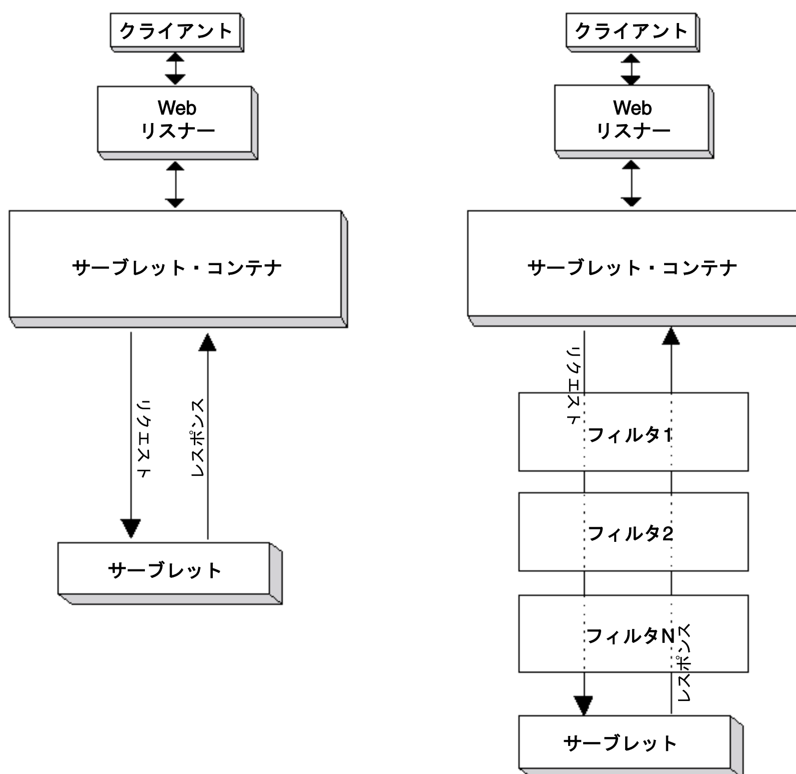
この項では、次の項目について簡単に説明します。

- サーブレット・コンテナによるフィルタの起動
- 一般的なフィルタ処理

サーブレット・コンテナによるフィルタの起動

図 4-1 は、左側が、リクエスト対象のサーブレットに対してフィルタが構成されていない例を示しています。右側の例では、複数のフィルタ（1、2、...、N）が構成されています。

図 4-1 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



各フィルタは、`javax.servlet.Filter` インタフェースを実装します。このインタフェースには、リクエストとレスポンスのペアをフィルタ・チェーンとともに入力として取得する `doFilter()` メソッドが含まれます。フィルタ・チェーンは、`javax.servlet.FilterChain` インタフェースを実装するクラス（サーブレット・コンテナによって提供されます）のインスタンスです。フィルタ・チェーンは、フィルタの順序を反映します。サーブレット・コンテナは、`web.xml` ファイルにおける構成順序に基づいて、マップされるフィルタを持つすべてのサーブレットまたはその他のリソースのフィルタ・チェーンを構成します。フィルタ・チェーンに含まれる各フィルタについては、フィルタに渡されるフィルタ・チェーン・オブジェクトが、順番にコールされる残りのフィルタを表します（これらのフィルタの後にターゲット・サーブレットが起動されます）。

`FilterChain` インタフェースも、`doFilter()` メソッドを指定します。このメソッドは、リクエストとレスポンスのペアを入力として取得し、各フィルタによってフィルタ・チェーンの次のエンティティを起動するために使用されます。

4-4 ページの「標準 Filter インタフェース」も参照してください。

たとえば、2 つのフィルタがある場合、このメカニズムの主要な手順は、次のとおりです。

1. ターゲット・サーブレットがリクエストされます。コンテナが、2 つのフィルタが存在することを検出し、フィルタ・チェーンを作成します。
2. チェーンの最初のフィルタが `doFilter()` メソッドによって起動されます。
3. 最初のフィルタがすべての前処理を完了し、フィルタ・チェーンの `doFilter()` メソッドをコールします。これにより、2 番目のフィルタが `doFilter()` メソッドによって起動されます。
4. 2 番目のフィルタがすべての前処理を完了し、フィルタ・チェーンの `doFilter()` メソッドをコールします。これにより、ターゲット・サーブレットが `service()` メソッドによって起動されます。
5. ターゲット・サーブレットの動作が完了すると、2 番目のフィルタのチェーン `doFilter()` コールが返され、2 番目のフィルタがすべての後処理を実行できます。
6. 2 番目のフィルタの動作が完了すると、最初のフィルタのチェーン `doFilter()` コールが返され、最初のフィルタがすべての後処理を実行できます。
7. 最初のフィルタの動作が完了すると、実行が完了します。

どのフィルタも、順序を認識していません。順序は、`web.xml` でフィルタが構成された順序に基づいて、フィルタ・チェーン全体によって処理されます。

一般的なフィルタ処理

フィルタの `doFilter()` メソッドの処理には、次のものが含まれます。

- リクエスト・オブジェクトのラッパーを作成して、入力フィルタリングを可能にします。必要に応じて、リクエスト・ラッパーのコンテンツまたはヘッダーを処理します。
- レスポンス・オブジェクトのラッパーを作成して、出力フィルタリングを可能にします。必要に応じて、レスポンス・ラッパーのコンテンツまたはヘッダーを処理します。
- `doFilter()` メソッドを使用して、リクエストとレスポンスのペア（またはラッパー）をチェーンの次のエンティティに渡します。または、リクエスト処理をブロックするために、チェーン `doFilter()` メソッドをコールしません。

ターゲット・リソースが起動する前に実行するすべての処理は、チェーン `doFilter()` コールの前にある必要があります。ターゲット・リソースが完了した後に実行するすべての処理は、チェーン `doFilter()` コールの後にある必要があります。これには、レスポンスのヘッダーの直接設定も含めることができます。

リクエスト・オブジェクトの前処理やレスポンス・オブジェクトの後処理では、元のリクエストまたはレスポンス・オブジェクトを直接操作できないことに注意してください。ラッパーを使用する必要があります。たとえば、レスポンスを後処理する場合、ターゲット・サーブレットはすでに動作を完了しており、フィルタがレスポンスになんらかの処理を実行できる時点ではレスポンスがすでにコミットされている可能性があります。元のレスポンスのかわりにレスポンス・ラッパーをチェーン `doFilter()` コールに渡す必要があります。4-11 ページの「フィルタを使用したリクエストまたはレスポンスのラップおよび変更」を参照してください。

標準 Filter インタフェース

サーブレット・フィルタは、javax.servlet.Filter インタフェースを実装します。このインタフェースの主要なメソッドである doFilter() は、フィルタ・チェーン全体を表すためにサーブレット・コンテナによって作成される javax.servlet.FilterChain インスタンスを入力として取得します。Filter インタフェースの初期化メソッドである init() は、javax.servlet.FilterConfig のインスタンスであるフィルタ構成オブジェクトを入力として取得します。この項では、これらのインタフェースで指定されるメソッドについて簡単に説明します。

この項で説明するインタフェースやメソッドに関する追加情報は、次のサイトで Sun 社の javax.servlet パッケージの Javadoc を参照してください。

http://java.sun.com/j2ee/1.4/docs/api/index.html

Filter インタフェースのメソッド

Filter インタフェースは、次のメソッドを指定して、フィルタに実装します。

- void init(FilterConfig filterConfig)

  サーブレット・コンテナは、フィルタが初めてインスタンス化されてサービスに組み込まれる際に、init() をコールします。このメソッドは、javax.servlet.FilterConfig インスタンスを入力として取得します。このインスタンスは、サーブレット・コンテナによって、初期化時に情報をフィルタに渡すために使用されます。すべての特別な初期化要件を実装にインクルードします。4-5 ページの「FilterConfig インタフェースのメソッド」も参照してください。

- ```
  void doFilter(ServletRequest request,
                ServletResponse response,
                FilterChain chain)
  ```

  ここで、フィルタ処理が実行されます。ターゲット・リソース（サーブレット、JSP ページなど）がリクエストされる（ここで、ターゲット・リソースが 1 つ以上のフィルタのチェーンにマップされる）たびに、サーブレット・コンテナは、web.xml フィルタ構成に基づく順序で、チェーンの各フィルタの doFilter() メソッドをコールします。（詳細は、4-8 ページの「フィルタ・チェーンの構造」を参照してください。）フィルタの doFilter() 処理内において、フィルタの doFilter() メソッドに渡されるフィルタ・チェーン・オブジェクトで doFilter() メソッドを起動します。（リクエスト処理をブロックする場合は例外です。）これにより、フィルタの動作が完了した後に、チェーン内の次のエンティティ（次のフィルタか、これがチェーン内の最後のフィルタである場合はターゲット・サーブレット）が起動します。

- destroy()

  サーブレット・コンテナは、フィルタのすべての動作が完了（doFilter() メソッドのすべてのスレッドが完了した場合またはタイムアウトが発生した場合）して、サービスからのフィルタの削除が開始される際、destroy() をコールします。すべての特別なクリーンアップ要件を実装にインクルードします。
  ```

FilterChain インタフェースのメソッド

FilterChain インタフェースは、次の 1 つのメソッドを指定します。

■ void doFilter(ServletRequest request,
                ServletResponse response)

  このメソッドを起動する（フィルタの doFilter() メソッドから実行）と、チェーン内の次のエンティティ（次のフィルタか、このメソッドがチェーン内の最後のフィルタからコールされている場合はサーブレットや JSP ページなどのターゲット・リソース）が起動します。

FilterConfig インタフェースのメソッド

FilterConfig インタフェースは、オプションで使用できる次のメソッドを指定します。

■ java.util.Enumeration getInitParameterNames()

  web.xml ファイルの <filter> 要素の下の <init-param> 要素を使用してフィルタの初期化パラメータを設定できます。（詳細は、4-7 ページの「フィルタの構成」を参照してください。）その後、フィルタで、FilterConfig オブジェクト（init() によって渡される）の getInitParameterNames() メソッドを使用して、初期化パラメータ名を含む Java 文字列の Enumeration オブジェクトを取得できます。（フィルタの初期化パラメータがない場合、Enumeration オブジェクトは空です。）

■ String getInitParameter(String paramname)

  初期化パラメータ名の取得の後に、getInitParameter() を使用して、指定されたパラメータの値を取得します。

■ ServletContext getServletContext()

  このメソッドを使用すると、リクエストされた（フィルタがフィルタリングする）サーブレットに関連付けられたサーブレット・コンテキストを取得できます。

■ String getFilterName()

  このメソッドを使用すると、web.xml ファイルの <filter-name> 要素に基づいて、フィルタの名前を取得できます。

フィルタの実装および構成

この項では、フィルタの実装および構成の基本手順を示します。これらの手順は、4-8 ページの「単純なフィルタの例」に示されている例の全体に含まれています。

また、web.xml でのフィルタ構成順序に基づくフィルタ・チェーンの構造を説明する項もあります。

フィルタのコードの実装

この項では、サーブレット・フィルタのコードを実装する手順を示します。

1. javax.servlet.Filter インタフェースを実装するクラスを作成します。次に例を示します。

```
public class TimerFilter implements javax.servlet.Filter { }
```

2. フィルタの初期化のために、Filter インタフェースで指定される init() メソッドを実装します。最初に、init() が入力として取得する javax.servlet.FilterConfig オブジェクトを作成または取得します。次に例を示します。

```
private FilterConfig filterConfig;
...
public void init(final FilterConfig filterConfig)
{
    this.filterConfig = filterConfig;
}
```

特別な初期化処理が必要な場合は、4-5 ページの「FilterConfig インタフェースのメソッド」を参照してください。

3. フィルタ処理のために、Filter インタフェースで指定される doFilter() メソッドを実装します。このメソッドは、リクエスト・オブジェクト、レスポンス・オブジェクトおよび javax.servlet.FilterChain オブジェクト（サーブレット・コンテナで作成される）を取得します。必要なすべての処理を実装し、（通常）フィルタ・チェーン・オブジェクトの doFilter() をコールしてチェーン内の次のエンティティを起動します。次に例を示します。

```
public void doFilter(ServletRequest request, ServletResponse response,
       FilterChain chain)
       throws java.io.IOException, javax.servlet.ServletException
{
    long start = System.currentTimeMillis();
    System.out.println("Milliseconds in: " + start);
    chain.doFilter(request, response);
    long end = System.currentTimeMillis();
    System.out.println("Milliseconds out: " + end);
}
```

最初の println() コールはチェーンの他の部分が起動される前に実行され、2 番目の println() コールは後で chain.doFilter() が返されたときに実行されます。

4. リソースをクリーンアップしたり、フィルタがサービスから削除される前に必要なすべての特別な処理を実行したりするために、Filter インタフェースで指定される destroy() メソッドを実行します。次に例を示します。

```
public void destroy()
{
    filterConfig = null;
}
```

注意： HTTP リクエストまたはレスポンスに変更を加えるフィルタの実装においては、追加の考慮事項があります。4-11 ページの「フィルタを使用したリクエストまたはレスポンスのラップおよび変更」を参照してください。

フィルタの構成

この項では、サーブレット・フィルタを構成する手順を示します。フィルタごとに、web.xml で次の手順を実行してください。

1. フィルタ・クラス（パッケージを含む）をフィルタ名にマップする <filter> 要素およびそのサブ要素を使用してフィルタを宣言します。次に例を示します。

```
<filter>
    <filter-name>timer</filter-name>
    <filter-class>filter.TimerFilter</filter-class>
</filter>
```

オプションで、サーブレットの場合と同様に、ここで初期化パラメータを指定できます。

```
<filter>
    <filter-name>timer</filter-name>
    <filter-class>filter.TimerFilter</filter-class>
    <init-param>
        <param-name>name</param-name>
        <param-value>value</param-value>
    <init-param>
</filter>
```

2. <filter-mapping> 要素およびそのサブ要素を使用して、フィルタ名をサーブレット名または URL パターンにマップし、フィルタを対応するリソース（サーブレット、JSP ページなど）に関連付けます。たとえば、myservlet という名前のサーブレットが起動されると必ずフィルタが起動されるようにするには、次のようにします。

```
<filter-mapping>
    <filter-name>timer</filter-name>
    <servlet-name>myservlet</servlet-name>
</filter-mapping>
```

また、URL パターンに基づいて、sleepy.jsp がリクエストされると必ずフィルタが起動されるようにするには、次のようにします。

```
<filter-mapping>
    <filter-name>timer</filter-name>
    <url-pattern>/sleepy.jsp</url-pattern>
</filter-mapping>
```

<url-pattern> 要素で特定のリソースを指定するかわりに、（例のように）ワイルド・カード文字を使用して複数のリソースを適合させることもできます。

```
<url-pattern>/mypath/*</url-pattern>
```

任意のフィルタ名を使用できますが、意味のある名前を使用することをお薦めします。このフィルタ名は、単に、フィルタ・クラスをサーブレット名または URL パターンにマップする際のリンケージとして使用されます。

リソースに適用する複数のフィルタを構成する場合、それらのフィルタは、web.xml での宣言順序に基づいてサーブレット・チェーンに入れられ、ターゲット・サーブレットがリクエストされたときにその順序で起動されます。次項の「フィルタ・チェーンの構造」を参照してください。

注意 : ターゲットの転送またはインクルードについては、フィルタを構成する追加手順があります。4-10 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。

フィルタ・チェーンの構造

web.xml でフィルタを宣言してマップすると、Web アプリケーションの各サーブレットまたはその他のリソース（JSP ページ、静的ページなど）に適用されるフィルタがサーブレット・コンテナによって決定されます。その後、サーブレットまたはリソースごとに、サーブレット・コンテナが、次のように、web.xml 構成順序に基づいて、適切なフィルタのチェーンを構築します。

1. 最初に、<url-pattern> 要素に基づいてサーブレットまたはリソースに適合するすべてのフィルタが、web.xml でのフィルタの宣言順に、チェーンに入れられます。
2. 次に、<servlet-name> 要素に基づいてサーブレットまたはリソースに適合するすべてのフィルタが、<url-pattern> に合致する最後のフィルタに続いて <servlet-name> に合致する最初のフィルタが入る順序で、チェーンに入れられます。
3. 最後に、<servlet-name> に合致する最後のフィルタの次に、ターゲット・サーブレットまたはその他のリソースがチェーンの最後に入れられます。

単純なフィルタの例

次の例は、JSP ページがリクエストされたときに起動するフィルタを示しています。JSP ページは、ブラウザにメッセージを書き込みます。フィルタは、OC4J コンソールに、JSP ページの実行の前後に 1 行ずつ、2 行のメッセージを書き込みます。

単純なフィルタのコードの作成

次に、単純なフィルタのコードの TimerFilter を示します。doFilter() メソッドは、OC4J コンソールに、ターゲット JSP の実行の前後に 1 行ずつ、2 行のメッセージを書き込みます。

```
package filter;

import javax.servlet.*;

public class TimerFilter implements javax.servlet.Filter
{
    private FilterConfig filterConfig;

    public void doFilter(ServletRequest request, ServletResponse response,
           FilterChain chain)
           throws java.io.IOException, javax.servlet.ServletException
    {
        long start = System.currentTimeMillis();
        System.out.println("Milliseconds in: " + start);
        chain.doFilter(request, response);
        long end = System.currentTimeMillis();
        System.out.println("Milliseconds out: " + end);
    }

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

    public void destroy()
    {
        filterConfig = null;
    }
}
```

ターゲット JSP ページの作成

次に、待機してから待機時間をブラウザに出力するターゲット JSP ページの `sleepy.jsp` を示します。

```
<%
  int sleeptime = 1000;
  Thread.sleep(sleeptime);
%>
<HTML>
<HEAD>
<TITLE>Sleepy JSP</TITLE>
</HEAD>
<BODY>
<HR>
<P><CENTER>Sleepy JSP slept for <%= sleeptime %> milliseconds!</CENTER></P>
<HR>
</BODY>
</HTML>
```

単純なフィルタの構成

次に、単純なフィルタを宣言してそのフィルタを `sleepy.jsp` のリクエストにマップする `web.xml` での構成を示します。

```
<?xml version="1.0" ?>
<!DOCTYPE web-app (doctype...)>
<web-app>
   <filter>
      <filter-name>timer</filter-name>
      <filter-class>filter.TimerFilter</filter-class>
   </filter>
   <filter-mapping>
      <filter-name>timer</filter-name>
      <url-pattern>/sleepy.jsp</url-pattern>
   </filter-mapping>
</web-app>
```

単純なフィルタの例のパッケージ化

この例の WAR ファイル（`filter.war`）は、次のコンテンツと構造を持ちます。

```
sleepy.jsp
META-INF/Manifest.mf
WEB-INF/web.xml
WEB-INF/classes/filter/TimerFilter.class
WEB-INF/classes/filter/TimerFilter.java
```

また、EAR ファイルは、次のとおりです。

```
filter.war
META-INF/Manifest.mf
META-INF/application.xml
```

（`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。）

単純なフィルタの例の起動

この例では、application.xml がコンテンツ・パスの /myfilter を filter.war にマップするものとします。この場合、デプロイ後、次のように、sleepy.jsp を起動します。その結果、フィルタが実行されます。

```
http://host:port/myfilter/sleepy.jsp
```

次のメッセージがブラウザに書き込まれます。

```
Sleepy JSP slept for 1000 milliseconds!
```

例を実行すると、次のメッセージが OC4J コンソール（たとえば、スタンドアロン環境で OC4J を実行している場合は、OC4J を起動した場所）に書き込まれます。

```
04/04/30 13:01:19 Milliseconds in: 1083355279136
04/04/30 13:01:20 Milliseconds out: 1083355280152
```

Milliseconds in 行は、JSP ページの起動前に書き込まれます。Milliseconds out 行は、JSP ページが実行され、実行がフィルタに返された後に書き込まれます。この例では、1016 ミリ秒の差がありますが、そのほとんどは、JSP ページの 1000 ミリ秒間の待機によるものです。

転送またはインクルード・ターゲットのフィルタ

直接リクエスト・ターゲットに対する動作に加えて（またはそのかわりに）転送ターゲットまたはインクルード・ターゲットに対して動作するようにフィルタを構成することができます。これについて、次の項で説明します。

- web.xml の <dispatcher> 要素
- 転送またはインクルード・ターゲットのフィルタの構成

> **注意：** インクルードおよび転送に関する一般的な情報は、6-13 ページの「インクルードおよび転送による他のサーブレットへのディスパッチ」を参照してください。

web.xml の <dispatcher> 要素

転送ターゲットまたはインクルード・ターゲットに対するフィルタを構成するには、web.xml の <filter-mapping> の <dispatcher> サブ要素を使用します。この要素については、次の 4 つの値がサポートされています。

- INCLUDE: この値は、指定されたサーブレット名に合致するインクルード・ターゲットまたは指定されたパターンに合致する URL を持つインクルード・ターゲットのすべてに適用されるフィルタに対して使用します。
- FORWARD: この値は、指定されたサーブレット名に合致する転送ターゲットまたは指定されたパターンに合致する URL を持つ転送ターゲットのすべてに適用されるフィルタに対して使用します。
- REQUEST: この値は、指定されたサーブレット名に合致する直接リクエスト・ターゲットまたは指定されたパターンに合致する URL を持つ直接リクエスト・ターゲットにも適用されるフィルタに対して、INCLUDE または FORWARD 設定（設定ごとに 1 つの <dispatcher> 要素）に加えて使用します。（REQUEST 値のみを使用することは無意味です。直接リクエストのみにフィルタを適用する場合は、<dispatcher> 要素を使用する必要はありません。）
- ERROR: この値は、エラー・ページ・メカニズムで適用されるフィルタに対して使用します。

例は、次項の「転送またはインクルード・ターゲットのフィルタの構成」を参照してください。

転送またはインクルード・ターゲットのフィルタの構成

この項では、転送ターゲットまたはインクルード・ターゲットに対して動作するようにフィルタを構成する場合のいくつかの例を示します。まずフィルタ宣言を示し、その後で代替フィルタ・マッピング構成を示します。

```
<filter>
    <filter-name>myfilter</filter-name>
    <filter-class>mypackage.MyFilter</filter-class>
</filter>
```

MyFilter を実行して includedservlet というインクルード・ターゲットをフィルタリングするには、次のようにします。

```
<filter-mapping>
   <filter-name>myfilter</filter-name>
   <servlet-name>includedservlet</servlet-name>
   <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
```

include() コールはアプリケーションの任意のサーブレット（またはその他のリソース）から発生していることに注意してください。また、値として REQUEST を持つ別の <dispatcher> 要素がないかぎり、MyFilter が includedservlet の直接リクエストに対して実行されないことに注意してください。

MyFilter を実行して、/mypath/ という URL パターンによって直接リクエストされるすべてのサーブレットや、/mypath/ という URL パターン文字列によって起動するすべての転送ターゲットをフィルタリングするには、次のようにします。

```
<filter-mapping>
   <filter-name>myfilter</filter-name>
   <url-pattern>/mypath/*</url-pattern>
   <dispatcher>FORWARD</dispatcher>
   <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

フィルタを使用したリクエストまたはレスポンスのラップおよび変更

特に役立つフィルタの機能は、リクエストまた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。リクエストまたはレスポンスを操作するには、ラッパーを作成する必要があります。次の一般的な手順を使用できます。

1. リクエストを操作するには、標準 javax.servlet.http.HttpServletRequestWrapper クラスを拡張するクラスを作成します。このクラスは、必要に応じてリクエストに変更を加えることを可能にするリクエスト・ラッパーになります。
2. レスポンスを操作するには、標準 javax.servlet.http.HttpServletResponseWrapper クラスを拡張するクラスを作成します。このクラスは、ターゲット・サーブレットまたはその他のリソースがレスポンスを提供し、多くの場合にそれをコミットした後に、レスポンスに変更を加えることを可能にするレスポンス・ラッパーになります。
3. オプションで、レスポンスの出力ストリームにカスタム機能を追加する場合は、標準 javax.servlet.ServletOutputStream クラスを拡張するクラスを作成します。
4. カスタム・クラスのインスタンスを使用するフィルタを作成し、必要に応じて、リクエストまたはレスポンスに変更を加えます。

次項の「レスポンス・フィルタの例」で、レスポンスに変更を加えるフィルタの例を提供します。

レスポンス・フィルタの例

この例では、カスタムのサーブレット出力ストリームを使用する HTTP サーブレット・レスポンス・ラッパーを使用します。この機能により、ラッパーは、ターゲット HTML ページがレスポンスを作成して送信した後でも、レスポンス・データを操作することができます。ラッパーを使用しないと、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、サーブレットがレスポンスをコミットした後）では、レスポンス・データを変更できません。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。

この例では、次のカスタム・クラスを使用します。

- GenericResponseWrapper: HTTP レスポンスの操作でカスタム機能を使用できるように HttpServletResponseWrapper を拡張します。
- FilterServletOutputStream: レスポンス・ラッパーで使用できるカスタム機能を提供するために ServletOutputStream を拡張します。
- MyGenericFilter: このクラスは、ベース・クラスとして使用される汎用の空（パススルー）フィルタです。
- PrePostFilter: MyGenericFilter を拡張し、HTTP レスポンスに変更を加えて HTML ページ出力の前後に 1 行ずつメッセージを挿入する doFilter() コードを実装します。

カスタム出力ストリーム・コードの作成

この FilterServletOutputStream というクラスは、標準 ServletOutputStream クラスを拡張して、レスポンス・ラッパーが使用する特別な機能を実装します。

```
package mypkg;

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);
  }

}
```

レスポンス・ラッパー・コードの作成

この GenericResponseWrapper というクラスは、標準 HttpServletResponseWrapper クラスを拡張して、HTTP レスポンスを操作するためのカスタム機能を実装します。

```
package mypkg;

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;
  }
}
```

ベース・フィルタのコードの作成

この MyGenericFilter というクラスは、空フィルタで、この例のレスポンス・フィルタによって拡張されるテンプレートを提供します。

```
package mypkg;

import javax.servlet.*;

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

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

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

  public void destroy() {
  }
}
```

レスポンス・フィルタのコードの作成

MyGenericFilter クラスを拡張するこの PrePostFilter というクラスは、ターゲット HTML ページのレスポンスに変更を加え、HTML ページ出力の前後にそれぞれ出力行を追加します。

```
package mypkg;

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(new String("<HR>PRE<HR>").getBytes());
  GenericResponseWrapper wrapper =
         new GenericResponseWrapper((HttpServletResponse) response);
  chain.doFilter(request,wrapper);
  out.write(wrapper.getData());
  out.write(new String("<HR>POST<HR>").getBytes());
  out.close();
  }
}
```

ターゲット HTML ページの作成

この prepostfilter.html という HTML ページは、この例でユーザーによってリクエストされます。フィルタは、このページの出力の前後に出力を挿入します。

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

レスポンス・フィルタの構成

次に、レスポンス・フィルタを宣言してそのフィルタを prepostfilter.html のリクエストにマップする web.xml での構成を示します。

```
<?xml version="1.0" ?>
<!DOCTYPE web-app (doctype...)>
<web-app>
   <filter>
      <filter-name>prepost</filter-name>
      <filter-class>mypkg.PrePostFilter</filter-class>
   </filter>
   <filter-mapping>
      <filter-name>prepost</filter-name>
      <url-pattern>/prepostfilter.html</url-pattern>
   </filter-mapping>
</web-app>
```

レスポンス・フィルタの例のパッケージ化

この例の WAR ファイル（myresponsewrapper.war）は、次のコンテンツと構造を持ちます。

```
prepostfilter.html
META-INF/Manifest.mf
WEB-INF/web.xml
WEB-INF/classes/mypkg/FilterServletOutputStream.class
WEB-INF/classes/mypkg/FilterServletOutputStream.java
WEB-INF/classes/mypkg/GenericResponseWrapper.class
WEB-INF/classes/mypkg/GenericResponseWrapper.java
WEB-INF/classes/mypkg/MyGenericFilter.class
WEB-INF/classes/mypkg/MyGenericFilter.java
WEB-INF/classes/mypkg/PrePostFilter.class
WEB-INF/classes/mypkg/PrePostFilter.java
```

また、EAR ファイルは、次のとおりです。

```
myresponsewrapper.war
META-INF/application.xml
META-INF/Manifest.mf
```

（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。）

レスポンス・フィルタの例の起動

この例では、`application.xml` がコンテンツ・パスの `/mywrapper` を `myresponsewrapper.war` にマップするものとします。この場合、デプロイ後、次のように、`prepostfilter.html` を起動します。その結果、フィルタが実行されます。

http://*host*:*port*/mywrapper/prepostfilter.html

次に、ブラウザへの出力を示します。フィルタによって、PRE、POST および横の罫線が追加されています。

PRE

This is a testpage. You should see
this text when you invoke prepostfilter.html,
as well as the additional material added
by the PrePostFilter class.

POST

フォーム認証フィルタ

10.1.3 で導入された OC4J 固有フィルタ・ディスパッチャにより、フィルタはフォーム認証を介して OC4J に渡されるユーザー名 / パスワード資格証明にアクセスできます。たとえば、外部リソースの認証を追加して実行する場合にこの機能を使用します。

この機能を使用可能にするには、FORMAUTH 値を `orion-web.xml` ファイル内のフィルタの `<dispatcher>` 要素で指定します。

次の例では、フィルタを宣言するコードおよび FORMAUTH 機能を指定するコードを示します。

> **注意：**
>
> `<filter>` 要素は、`orion-web.xml` ファイルまたは `web.xml` ファイルのいずれかで宣言できます。
>
> `<filter-mapping>` 要素は、`orion-web.xml` ファイルで宣言する必要があります。

`orion-web.xml` または `web.xml` の場合は次のとおりです。

```
<filter>
   <filter-name>MyFilter</filter-name>
   <filter-class>myFilterClass</filter-class>
</filter>
```

orion-web.xml の場合は次のとおりです。

```
<orion-web-app>
...
   <web-app>
   ...
      <filter-mapping>
         <filter-name>MyFilter</filter-name>
         <dispatcher>FORMAUTH</dispatcher>
      </filter-mapping>
```

```
   </web-app>
</orion-web-app>
```

この方法で宣言したフィルタはいずれも、認証フォームの発行後、OC4J での認証の実行前に実行されます。

フィルタは `request.getParameters()` をコールして、`j_username` および `j_password` パラメータをリクエスト・オブジェクトから取得します。

次に、コール側コードの例を示します。

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

public class MyFilter implements Filter {

    public MyFilter() {
        super();
    }

    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain filterChain)
        throws IOException, ServletExxception {
        HttpServletRequest req = (HttpServletRequest) request;

String username = req.getParameter("j_username");
String password = req.getParameter("j_password");
// use these credentials to access a remote DB
....
filterChain.doFilter(request, response);
    }

    public void init(FilterConfig filterConfig) throws ServletException {
    }

    public void destroy() {
    }
}
```

5 イベント・リスナーの理解および使用方法

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（イベント・リスナーの概要は、6-17 ページの「サーブレット通知にイベント・リスナーを使用する場面」を参照）。リスナーを使用すると、イベントの状態に基づいた自動処理やより効率的なリソース管理を実行できます。リスナーは、リクエスト・イベント、セッション・イベントおよびサーブレット・コンテキスト・イベントについて実装できます。これについて、次の項で説明します。

- イベント・リスナー機能の概要
- イベント・リスナー・インタフェース
- イベント・リスナーの実装および構成
- セッションのライフ・サイクル・リスナーの例

イベント・リスナー機能の概要

次の 8 つのイベント・リスナー・カテゴリがあります。

- サーブレット・コンテキストのライフ・サイクル（起動または停止）
- サーブレット・コンテキスト属性の変更（追加、削除または置換）
- セッション・ライフサイクル（起動、無効化またはタイムアウト）
- セッション属性の変更（追加、削除または置換）
- セッションの移行（アクティブ化または非アクティブ化）
- セッション・オブジェクトのバインド（オブジェクトのセッションへのバインドまたはセッションからのアンバインド）
- リクエストのライフ・サイクル（リクエスト処理の開始）
- リクエスト属性の変更（追加、削除または置換）

これらのイベント・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。また、1 つのリスナー・クラスが、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` または `javax.servlet.http` パッケージの適切なインタフェースを 1 つ以上実装します。表 5-1 は、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 5-1 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	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	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	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	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
セッションの移行	セッションのアクティブ化 セッションの非アクティブ化	javax.servlet.HttpSessionActivationListener
セッション・オブジェクトのバインド	セッションへのオブジェクトのバインド セッションからのオブジェクトのアンバインド	javax.servlet.HttpSessionBindingListener
リクエストのライフ・サイクル	リクエスト処理の開始	javax.servlet.ServletRequestListener
リクエスト属性	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.ServletRequestAttributeListener

リスナーは、`web.xml` ファイルの `<listener>` 要素（`<web-app>` 要素のサブ要素）を使用して構成します。5-8 ページの「リスナーの構成」を参照してください。

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、web.xml で宣言されている各リスナー・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。

イベント・リスナー・インタフェース

この項では、表 5-1 にまとめたインタフェースのメソッドについて説明します。この項で説明する各メソッドは、該当するイベントが発生したときにサーブレット・コンテナによってコールされます。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。このインタフェースは、サーブレット・コンテキストのライフ・サイクル・イベントを追跡するために使用するクラスに実装します。

- void contextInitialized(ServletContextEvent sce)

  サーブレット・コンテナは、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ようになると、このメソッドをコールします。

- void contextDestroyed(ServletContextEvent sce)

  サーブレット・コンテナは、アプリケーションが停止される直前に、このメソッドをコールします。

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、作成済または停止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、1-11 ページの「サーブレット・コンテキスト : アプリケーション・コンテナ」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。このインタフェースは、サーブレット・コンテキストの属性イベントを追跡するために使用するクラスに実装します。

- void attributeAdded(ServletContextAttributeEvent scae)

  サーブレット・コンテナは、属性がサーブレット・コンテキストに追加されると、このメソッドをコールします。

- void attributeRemoved(ServletContextAttributeEvent scae)

  サーブレット・コンテナは、属性がサーブレット・コンテキストから削除されると、このメソッドをコールします。

- void attributeReplaced(ServletContextAttributeEvent scae)

  サーブレット・コンテナは、属性がサーブレット・コンテキストで置換される（値が変化する）と、このメソッドをコールします。

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。このインタフェースは、セッションのライフ・サイクル・イベントを追跡するために使用するリスナー・クラスに実装します。

- `void sessionCreated(HttpSessionEvent hse)`

  サーブレット・コンテナは、セッションが作成されると、このメソッドをコールします。

- `void sessionDestroyed(HttpSessionEvent hse)`

  サーブレット・コンテナは、セッションが終了する直前に、このメソッドをコールします。

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、作成済または終了対象のセッション・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。

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。このインタフェースは、セッションの属性イベントを追跡するために使用するリスナー・クラスに実装します。

- `void attributeAdded(HttpSessionBindingEvent hsbe)`

  サーブレット・コンテナは、属性がセッションに追加されると、このメソッドをコールします。

- `void attributeRemoved(HttpSessionBindingEvent hsbe)`

  サーブレット・コンテナは、属性がセッションから削除されると、このメソッドをコールします。

- `void attributeReplaced(HttpSessionBindingEvent hsbe)`

  サーブレット・コンテナは、属性がセッションで置換される(値が変化する)と、このメソッドをコールします。

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。

HttpSessionActivationListener メソッド

`HttpSessionActivationListener` インタフェースは、次のメソッドを指定します。このインタフェースは、セッションの移行（アクティブ化または非アクティブ化）イベントを追跡するために使用するリスナー・クラスに実装します。

- `void sessionDidActivate(HttpSessionEvent hse)`

  サーブレット・コンテナは、セッションがアクティブ化されると、このメソッドをコールします。

- `void sessionWillPassivate(HttpSessionEvent hse)`

  サーブレット・コンテナは、セッションが非アクティブ化される直前に、このメソッドをコールします。

サーブレット・コンテナは、`HttpSessionActivationListener` メソッドのコールに対する入力として使用するために、`HttpSessionEvent` クラスのインスタンスを作成します。このクラスの詳細は、5-4 ページの「HttpSessionListener メソッド、HttpSessionEvent クラス」を参照してください。

HttpSessionBindingListener メソッド

`HttpSessionBindingListener` インタフェースは、次のメソッドを指定します。このインタフェースは、セッションにバインドされるインスタンスを持つクラスに実装します。

- `void valueBound(HttpSessionBindingEvent hsbe)`

  サーブレット・コンテナは、（`HttpSessionBindingListener` を実装する）オブジェクトが（識別される）セッションにバインドされると、このメソッドをコールします。

- `void valueUnbound(HttpSessionBindingEvent hsbe)`

  サーブレット・コンテナは、オブジェクトが（識別される）セッションからアンバインドされると、このメソッドをコールします。オブジェクトは、明示的にアンバインドされる場合や、セッションの無効化またはタイムアウトの結果としてアンバインドされる場合があります。

サーブレット・コンテナは、`HttpSessionBindingListener` メソッドのコールに対する入力として使用するために、`HttpSessionBindingEvent` クラスのインスタンスを作成します。このクラスの詳細は、5-4 ページの「HttpSessionAttributeListener メソッド、HttpSessionBindingEvent クラス」を参照してください。

ServletRequestListener メソッド、ServletRequestEvent クラス

`ServletRequestListener` インタフェースは、次のメソッドを指定します。このインタフェースは、リクエストのライフ・サイクル・イベントを追跡するために使用するリスナー・クラスに実装します。

- `void requestInitialized(ServletRequestEvent sre)`

  サーブレット・コンテナは、リクエストが Web アプリケーションのスコープに入る直前に、このメソッドをコールします。

- `void requestDestroyed(ServletRequestEvent sre)`

  サーブレット・コンテナは、リクエストが Web アプリケーションのスコープから出る直前に、このメソッドをコールします。

サーブレット・コンテナは、`ServletRequestListener` メソッドのコールに対して入力される `javax.servlet.ServletRequestEvent` オブジェクトを作成します。`ServletRequestEvent` 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。

- `ServletRequest getServletRequest()`

  このメソッドを使用して、ステータスが変化したサーブレット・リクエストを取得します。

- `ServletContext getServletContext()`

  このメソッドを使用して、Web 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。

ServletRequestAttributeListener メソッド、ServletRequestAttributeEvent クラス

`ServletRequestAttributeListener` インタフェースは、次のメソッドを指定します。このインタフェースは、リクエストの属性イベントを追跡するために使用するリスナー・クラスに実装します。

- `void attributeAdded(ServletRequestAttributeEvent srae)`

  サーブレット・コンテナは、属性がリクエストに追加されると、このメソッドをコールします。

- `void attributeRemoved(ServletRequestAttributeEvent srae)`

  サーブレット・コンテナは、属性がリクエストから削除されると、このメソッドをコールします。

- `void attributeReplaced(ServletRequestAttributeEvent srae)`

  サーブレット・コンテナは、属性がリクエストで置換される（値が変化する）と、このメソッドをコールします。

サーブレット・コンテナは、`ServletRequestAttributeListener` メソッドのコールに対して入力される `javax.servlet.ServletRequestAttributeEvent` オブジェクトを作成します。`ServletRequestAttributeEvent` 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。

イベント・リスナーの実装および構成

この項では、リスナーの実装および構成の基本手順を示します。例の全体は、5-9 ページの「セッションのライフ・サイクル・リスナーの例」を参照してください。

リスナーのコードの実装

リスナー・クラスは、5-2 ページの「イベント・リスナー機能の概要」でまとめたイベントのカテゴリのいずれかまたはすべてに使用できます。1 つのクラスによって、複数のリスナーを実装できます。次に、実装コードを示します。

- サーブレット・コンテキストのライフ・サイクル・リスナーについては、`ServletContextListener` インタフェースを実装し、必要に応じて、メソッドの `contextInitialized()`（アプリケーションの起動時の処理用）および `contextDestroyed()`（アプリケーションの停止時の処理用）のコードを作成します。
- サーブレット・コンテキストの属性リスナーについては、`ServletContextAttributeListener` インタフェースを実装し、必要に応じて、メソッドの `attributeAdded()`（属性の追加時の処理用）、`attributeRemoved()`（属性の削除時の処理用）および `attributeReplaced()`（属性値の変更時の処理用）のコードを作成します。
- セッションのライフ・サイクル・リスナーについては、`HttpSessionListener` インタフェースを実装し、必要に応じて、メソッドの `sessionCreated()`（セッション作成時の処理用）および `sessionDestroyed()`（セッション無効化時の処理用）のコードを作成します。簡単な例は、5-11 ページの「セッションのライフ・サイクル・リスナーのコードの作成」を参照してください。
- セッションの属性リスナーについては、`HttpSessionAttributeListener` インタフェースを実装し、必要に応じて、メソッドの `attributeAdded()`、`attributeRemoved()` および `attributeReplaced()` のコードを作成します。
- セッションの移行リスナーについては、`HttpSessionActivationListener` インタフェースを実装し、必要に応じて、メソッドの `sessionDidActivate()`（セッションのアクティブ化時の処理用）および `sessionWillPassivate()`（セッションの非アクティブ化時の処理用）のコードを作成します。
- セッションのバインド・リスナーについては、`HttpSessionBindingListener` インタフェースを実装し、必要に応じて、メソッドの `valueBound()`（セッションへのオブジェクトのバインド時の処理用）または `valueUnbound()`（セッションからのオブジェクトのアンバインド時の処理用）のコードを作成します。
- リクエストのライフ・サイクル・リスナーについては、`ServletRequestListener` インタフェースを実装し、必要に応じて、メソッドの `requestInitialized()`（リクエストがスコープに入ったときの処理用）および `requestDestroyed()`（リクエストがスコープから出たときの処理用）のコードを作成します。
- リクエストの属性リスナーについては、`ServletRequestAttributeListener` インタフェースを実装し、必要に応じて、メソッドの `attributeAdded()`、`attributeRemoved()` および `attributeReplaced()` のコードを作成します。

注意：

- 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。
- 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。

リスナーの構成

各リスナー・クラスを構成するには、アプリケーション web.xml ファイルの <listener> 要素（<web-app> のサブ要素）とその <listener-class> サブ要素を使用します。

```
<web-app>
   <listener>
      <listener-class>SessionLifeCycleEventExample</listener-class>
   </listener>
...
   <servlet>
      <servlet-name>name</servlet-name>
      <servlet-class>class</servlet-class>
   </servlet>
...
   <servlet-mapping>
      <servlet-name>name</servlet-name>
      <url-pattern>path</url-pattern>
   </servlet-mapping>
...
</web-app>
```

リスナーは、特定のサーブレットには関連付けられません。アプリケーションの起動時に、各イベント・カテゴリについて、サーブレット・コンテナは、web.xml で宣言された順序でリスナーを登録します。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、該当するイベントが発生するたびに登録順に起動されます。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。

ただし、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リスナーに通知され、サーブレット・コンテキスト・リスナーの前にリクエストおよびセッション・リスナーに通知されます。

初期ファイルに必要な物理ファイル

初期ファイルをサーブレットにディスパッチするためには、物理ファイルが必要です。JSP ページ /index.jsp にマップする /index.html にマップするサーブレットを作成し、それを初期ファイルとして使用する場合は、web.xml ファイルに次のエントリを含める必要があります。

```
<servlet>
   <servlet-name> index_jsp </servlet-name>
   <jsp-file> /index.jsp </jsp-file>
</servlet>

<servlet-mapping>
   <servlet-name>index_jsp</servlet-name>
   <url-pattern>/index.html</url-pattern>
</servlet-mapping>
```

Web アプリケーションに、物理ファイル、/index.html がある場合にのみ、これは有効です。ファイルの長さは 0（ゼロ）でもかまいません。ファイルが存在しさえすれば、このサーブレットは初期ファイルとしてロードされます。ファイルが存在しない場合は、java.lang.StringIndexOutOfBoundsException 例外がスローされます。

セッションのライフ・サイクル・リスナーの例

次に、セッションが作成または終了されるたびに OC4J コンソールにメッセージを書き込むセッションのライフ・サイクル・イベント・リスナーの簡単な例を示します。この例には、次のコンポーネントのコードが含まれます。

- `index.jsp`: アプリケーションの「ようこそ」ページ。このページには、`SessionCreateServlet` を起動して HTTP セッションを作成するためのリンクがあります。
- `SessionCreateServlet`: このサーブレットは、HTTP セッションを作成します。また、セッションを終了するための `SessionDestroyServlet` へのリンクを持ちます。
- `SessionDestroyServlet`: このサーブレットは、セッションを終了します。また、「ようこそ」ページに戻るためのリンクを持ちます。
- `SessionLifeCycleEventExample`: イベント・リスナー・クラス。`HttpSessionListener` インタフェースと、セッションの作成または終了時にコンソール・メッセージを書き込む `sessionCreated()` および `sessionDestroyed()` メソッドのコードを実装します。

JSP「ようこそ」ページの作成

次に、JSP「ようこそ」ページの `index.jsp` を示します。このページの **「Create New Session」** リンクをクリックすると、セッション作成サーブレットを起動できます。この例では、アプリケーションのコンテキスト・パスを `/mylistener` とし、`mysessioncreate` がセッション作成サーブレットのサーブレット・パスとして `web.xml` で構成されているものとします。

```
<%@page session="false" %>
<HTML>
<BODY>
<H2>OC4J Session Event Listener</H2>
<P>
This example demonstrates the use of a session event listener.
</P>
<P>
<a href="/mylistener/mysessioncreate">Create New Session</A><br><br>
</P>
<P>
Click the <b>Create</b> link above to start a new session.<br>
A session listener has been configured for this application.<br>
The servlet container will send an event to this listener when a new session is<br>
created or destroyed. The output from the event listener will be visible in the<br>
console window from where OC4J was started.
</P>
</BODY>
</HTML>
```

セッション作成サーブレットの作成

この SessionCreateServlet というサーブレットは、HTTP セッション・オブジェクトを作成し、作成されたセッションに関する情報を表示します。SessionDestroyServlet を起動する「**Destroy Session**」リンクをクリックすると、セッションを終了できます。この例では、アプリケーションのコンテキスト・パスを /mylistener とし、mysessiondestroy がセッション無効化サーブレットのサーブレット・パスとして web.xml で構成されているものとします。

```
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 for the response.
      res.setContentType("text/html");

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

      out.println("<HTML><BODY>");
      out.println("<A HREF=\"/mylistener/mysessiondestroy\">Destroy Session</A>");
      out.println("<h2>Session Created</h2>");
      out.println("Also check the OC4J console.");
      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("</BODY></HTML>");

   }
}
```

セッション無効化サーブレットの作成

この SessionDestroyServlet というサーブレットは、HTTP セッション・オブジェクトを破棄します。**「Reload Welcome Page」**リンクをクリックすると、JSP「ようこそ」ページに戻って新しいセッションを作成できます。この例では、アプリケーションのコンテキスト・パスを /mylistener とします。

```
import java.io.*;
import java.util.Enumeration;
import java.util.Date;
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 the session.
      session.invalidate();

      // Set content type for response.
      res.setContentType("text/html");

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

      out.println("<HTML><BODY>");
      out.println("<A HREF=\"/mylistener/index.jsp\">Reload Welcome Page</A>");
      out.println("<h2>Session Destroyed</h2>");
      out.println("Also check the OC4J console.");
      out.println("</BODY></HTML>");
      out.close();
   }
}
```

セッションのライフ・サイクル・リスナーのコードの作成

この項では、セッション・ライフ・サイクル・リスナー・クラスの SessionLifeCycleEventExample を示します。このクラスは、HttpSessionListener インタフェースを実装します。このクラスの sessionCreated() メソッドは、HTTP セッションが作成されるたびにサーブレット・コンテナによってコールされます。このコールは、JSP「ようこそ」ページの**「Create New Session」**をクリックすると実行されます。sessionCreated() のコール時に、新規セッションの ID を示す CREATED メッセージが OC4J コンソールに書き込まれます。

sessionDestroyed() メソッドは、HTTP セッションが破棄されるたびにサーブレット・コンテナによってコールされます。このコールは、セッション作成サーブレットで**「Destroy Session」**をクリックすると実行されます。sessionDestroyed() のコール時に、終了セッションの ID を示す DESTROYED メッセージが OC4J コンソールに出力されます。

(このクラスは、ServletContextListener インタフェースも実装し、contextInitialized() および contextDestroyed() メソッドも持ちますが、これらの機能は、この例では使用されません。)

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

public class SessionLifeCycleEventExample
       implements ServletContextListener, HttpSessionListener
```

```
{
   ServletContext servletContext;

   /* Methods for 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("CREATED",hse);
   }

   public void sessionDestroyed(HttpSessionEvent hse)
   {
      log("DESTROYED",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);
   }
}
```

セッションのライフ・サイクル・リスナーの例の構成

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、セッションのライフ・サイクル・イベントの発生時に、SessionLifeCycleEventExample 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(サーブレット・コンテキストのライフ・サイクル・イベントの発生時にもコールされますが、この例には関係ありません)。次に、web.xml のエントリを示します。

```
<?xml version="1.0" ?>
<!DOCTYPE web-app (doctype...)>
<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>
   <servlet-mapping>
      <servlet-name>sessioncreate</servlet-name>
      <url-pattern>mysessioncreate</url-pattern>
```

```
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>sessiondestroy</servlet-name>
        <url-pattern>mysessiondestroy</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

セッションのライフ・サイクル・リスナーの例のパッケージ化

この例の WAR ファイル（`sessionlistener.war`）は、次のコンテンツと構造を持ちます。

```
index.jsp
META-INF/Manifest.mf
WEB-INF/web.xml
WEB-INF/classes/SessionCreateServlet.class
WEB-INF/classes/SessionCreateServlet.java
WEB-INF/classes/SessionDestroyServlet.class
WEB-INF/classes/SessionDestroyServlet.java
WEB-INF/classes/SessionLifeCycleEventExample.class
WEB-INF/classes/SessionLifeCycleEventExample.java
```

また、EAR ファイルは、次のとおりです。

```
sessionlistener.war
META-INF/application.xml
META-INF/Manifest.mf
```

（`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。）

セッションのライフ・サイクル・リスナーの例の起動

この例では、`application.xml` がコンテンツ・パスの `/mylistener` を `sessionlistener.war` にマップするものとします。この場合、デプロイ後に、次のように、JSP「ようこそ」ページを起動します。

```
http://host:port/mylistener/index.jsp
```

次に、「ようこそ」ページの出力を示します。

OC4J Session Event Listener

This example demonstrates the use of a session event listener.

Create New Session

Click the **Create** link above to start a new session.
A session listener has been configured for this application.
The servlet 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.

「**Create New Session**」をクリックすると、セッション作成サーブレットが起動します。テスト実行では、これにより、次のように出力されます。

Destroy Session

Session Created

Also check the OC4J console.

Session Data:

New Session: true
Session ID: 8223afa422b84b94235252164cb9a7ad84089f1abe70
Creation Time: Thu May 13 15:56:25 PDT 2004

また、OC4J コンソールには、次のようにレポートされます。

```
04/05/13 15:56:25 SessionLifeCycleEventExample
Session ID: 8223afa422b84b94235252164cb9a7ad84089f1abe70     CREATED
```

「**Destroy Session**」をクリックすると、セッション終了サーブレットが起動します。これにより、次のように出力されます。

Reload Welcome Page

Session Destroyed

Also check the OC4J console.

また、テスト実行では、OC4J コンソールに、次のようにレポートされます。

```
04/05/13 15:58:08 SessionLifeCycleEventExample
Session ID: 8223afa422b84b94235252164cb9a7ad84089f1abe70     DESTROYED
```

「**Reload Welcome Page**」をクリックすると、JSP「ようこそ」ページに戻ります。このページでは、別のセッションを作成できます。

6 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。

- 基本サーブレットの作成
- 単純なサーブレットの例
- HTML フォームおよびリクエスト・パラメータの使用方法
- インクルードおよび転送による他のサーブレットへのディスパッチ
- 前処理および後処理のためにフィルタを使用する場面
- サーブレット通知にイベント・リスナーを使用する場面
- スタック・トレースの表示方法
- アプリケーションの Apache Tomcat から OC4J への移行

OC4J 開発に関するより一般的な情報については、『Oracle Containers for J2EE 開発者ガイド』を参照してください。

注意 : 指定されたディレクトリでのサーブレット・ソースの自動再コンパイルをトリガーする、開発時に使用できる便利なフラグがあります。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リロードされます。`development` フラグの詳細は、B-18 ページの「<orion-web-app>」を参照してください。

基本サーブレットの作成

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、javax.servlet.http.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。ほとんどのサーブレットは、HTTP の GET または POST リクエストを処理するために、HttpServlet の doGet() メソッドまたは doPost() メソッドをそれぞれオーバーライドします。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特別な処理が必要な場合は、init() メソッドと destroy() メソッドをオーバーライドすることもあります。

次の項では、これらのメソッドを実装する基本使用例を説明し、レスポンスの設定方法を示し、「Hello World」サーブレットのコードを順番に説明します。

- サーブレット・インタフェースのメソッドを実装する場面
- レスポンスの設定
- 単純なサーブレットの作成手順

サーブレット・インタフェースのメソッドを実装する場面

次に、サーブレット開発の基本コード・テンプレートを示します。

```
package ...;
import ...;

public class MyServlet extends HttpServlet {

  public void init(ServletConfig config) {
  }

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

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

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

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

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

  public void destroy() {
  }

}
```

次の項では、これらのメソッドをオーバーライドする例について説明します。

init() メソッドをオーバーライドする場面

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に、init() メソッドをオーバーライドすることができます。次のような処理が含まれます。

- データベ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。

たとえば、データソースを使用してデータベース接続を確立するには、次のようにします。

```
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
   }
   ...
}
```

doGet() または doPost() メソッドをオーバーライドする場面

ほとんどすべてのサーブレットは、大部分の処理について、HTTP の GET リクエストを処理するために doGet() メソッドを、または HTTP の POST リクエストを処理するために doPost() メソッドをオーバーライドします。GET と POST は、フォーム・データをサーバーに渡すための 2 つの HTTP メソッドです。どのような場面でどちらのメソッドを使用するのかについて、このマニュアルでは詳しく説明しませんが、セキュリティを特に考慮する場合は、doPost() メソッドの方が適していることがあります。これは、GET メソッドでは、フォーム・パラメータが URL 文字列に直接入力されたり、大きなデータ・シーケンスについて、クライアントがサーバーに送信できるデータの大きさが制限されなかったりするためです。

doGet() または doPost() の実装では、クライアントに渡すデータを生成するコードの作成に加えて、通常、HTTP リクエストからデータを読み取り、HTTP レスポンスを設定し、レスポンスを生成するコードを作成します。追加情報は、この少し後にある「レスポンスの設定」および 6-7 ページの「HTML フォームおよびリクエスト・パラメータの使用方法」を参照してください。

6-5 ページの「単純なサーブレットの作成手順」に、簡単な doGet() 実装の手順が示されています。

doPut() メソッドをオーバーライドする場面

このメソッドを使用して、HTTP PUT リクエストを実行します。これにより、ファイルがクライアントからサーバーに書き込まれます。doPut() メソッドは、コンテンツ・ヘッダーを処理できる必要があります（処理できない場合はエラー・メッセージが生成されます）。また、受け取ったコンテンツ・ヘッダーに変更を加えないようにする必要があります。

doDelete() メソッドをオーバーライドする場面

このメソッドを使用して、HTTP DELETE リクエストを実行します。これにより、ファイルまたは Web ページがサーバーから削除されます。

getServletInfo() メソッドをオーバーライドする場面

このメソッドを使用して、サーブレットから、作成者やバージョンなどの情報を取得します。デフォルトでは、このメソッドは空の文字列を返します。このため、このメソッドをオーバーライドして、意味のある情報を提供する必要があります。

destroy() メソッドをオーバーライドする場面

このメソッドは、サーブレットがシャットダウンされる直前にサーブレット・コンテナによってコールされます。次のように、サーブレットがシャットダウンの前にクリーンアップを必要とする場合は、このメソッドをオーバーライドすることができます。

- 永続データを更新して、最新の状態にする。
- データベース接続やファイル・ハンドルなどのリソースをクリーンアップする。

たとえば、6-3 ページの「init() メソッドをオーバーライドする場面」で開いたデータベース接続を閉じるには、次のようにします。

```
public void destroy() {
   try {
      conn.close();
   }
   ...
}
```

レスポンスの設定

サーブレットからレスポンスを送信するには、使用しているサーブレット・メソッド（通常、`doGet()` または `doPost()`）に渡される `HttpServletResponse` インスタンスを使用します。主な手順は、次のとおりです。

1. レスポンスのコンテンツ・タイプを設定し、オプションで、文字コード（MIME キャラクタ・セット）を指定します。

> **注意：** OC4J のデフォルト・コンテンツ・タイプが存在する場合は、OC4J の `global-web-application.xml`（グローバル）または `orion-web.xml`（アプリケーション・レベル）Web アプリケーション構成ファイルの `<default-mime-type>` 要素に反映されます。このコンテンツ・タイプは、Application Server Control コンソールのデプロイ・プラン・エディタ（概要は、2-2 ページの「OC4J 管理の概要」を参照）を使用して設定できます。

2. 文字データの場合は Writer オブジェクト（`java.io.PrintWriter`）、バイナリ・データの場合は出力ストリーム（`javax.servet.ServletOutputStream`）をレスポンス・オブジェクトから取得します。
3. レスポンス・データを Writer オブジェクトまたは出力ストリームに書き込みます。

次に、これらの手順に対応するコードを示します。

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException
{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html><body><h1>Hello World</h1></body></html>");
    ...
}
```

レスポンス・メソッドの要約は、1-8 ページの「HttpServletResponse インタフェースの主要メソッド」を参照してください。

> **開発のヒント：** サーブレット・コンテナは、レスポンスへのコミット後に Writer オブジェクトまたは出力ストリームを自動的にクローズしますが、明示的にクローズすることをお薦めします。

単純なサーブレットの作成手順

この項では、doGet() メソッドをオーバーライドする「Hello World」の例を示します。このサーブレットの全体は 6-6 ページの「単純なサーブレットの例」に示されていますが、この項でも順番に説明します。

サーブレットの例の最初の手順は、次のとおりです。

1. 必要に応じて、パッケージを宣言します。サーブレットの例では、mytest を宣言します。

```
package mytest;
```

2. 必要な Java パッケージ、特にサーブレット・パッケージをインポートします。一般に、次のコードが必要です。

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

3. HTTP 操作の HttpServlet を常に拡張するサーブレット・クラスを宣言します。

```
public class HelloWorld extends HttpServlet {
   ...
}
```

4. オーバーライドするすべてのサーブレット・メソッドを宣言します。HTTP 操作のサーブレット・メソッドはすべて、同じパラメータ（HTTP リクエスト・オブジェクトおよび HTTP レスポンス・オブジェクト）を取得し、同じ例外をスローします。サーブレットの例は、doGet() をオーバーライドします。

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

サーブレットの例の doGet() メソッドの手順は、次のとおりです。

1. レスポンス・オブジェクトのコンテンツ・タイプを設定します。この手順は不要な場合もありますが、一般に推奨されます。

```
response.setContentType("text/html");
```

   オプションで、文字コード（次の例の UTF-8 など）を指定することもできます。

```
response.setContentType("text/html; charset=UTF-8");
```

2. レスポンス・オブジェクトから Writer オブジェクトを取得します。

```
PrintWriter out = response.getWriter();
```

3. データをレスポンス・オブジェクトに書き込みます。

```
out.println("<html>");
out.println("<head>");
out.println("<title>Hello World!</title>");
out.println("</head>");
out.println("<body>");
out.println("<h1>Hi Amy!</h1>");
out.println("</body>");
out.println("</html>");
```

4. レスポンスにコミットしている出力ストリームをクローズします。

```
out.close();
```

（この単純な例では、リクエスト・データの操作は不要です。）

単純なサーブレットの例

この項では、前の項で手順を順番に説明した単純なサーブレットの例の全体を示します。この例は、2-12 ページの「単純なサーブレットの例のデプロイおよび起動」でデプロイされ、起動されます。

サンプル・コードの作成

次のコードを実行すると、ブラウザに「Hi Amy!」と表示されます。コードは、HelloWorld.java というファイルに入力します。package 文に従って、HelloWorld クラスは mytest パッケージに含まれます。

```
package mytest;

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

public class HelloWorld extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Hello World!</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Hi Amy!</h1>");
        out.println("</body>");
        out.println("</html>");

        out.close();
    }
}
```

サンプル・コードのコンパイル

サンプル・コードをコンパイルします。Sun 社の JDK とそのデフォルト・コンパイラを使用している場合は、次のように、.java ファイルのあるディレクトリからコンパイルします（% はシステム・プロンプトです）。

```
% javac HelloWorld.java
```

開発のヒント :

- Java 実行可能ファイル（JVM、Java コンパイラ、JAR ユーティリティなど）の位置を、システム・ファイル・パスに追加します。これにより、どこからでも実行できるようになります。たとえば、Sun 社の JDK 1.4.2、バージョン 4 の場合、*jdkroot*/j2sdk1.4.2_04/bin をファイル・パスに追加します。ここで、*jdkroot* は、JDK がインストールされているディレクトリのフルパスです。これにより、どこからでも、java、javac および jar を実行できるようになります。この手順は、オペレーティング・システムによって異なります。
- 標準サーブレット・クラスおよびインタフェースは、OC4J に付属しており、*oc4jroot*/j2ee/home/lib ディレクトリの servlet.jar というファイルに保存されています。ここで、*oc4jroot* は、OC4J がインストールされているディレクトリのフルパスです。servlet.jar を Java コンパイラで使用できるようにする必要があります。このための 1 つの方法は、*oc4jroot*/j2ee/home/lib/servlet.jar をシステムまたはユーザーの CLASSPATH 環境変数に追加するというものです。Sun 社の JDK を使用している場合は、servlet.jar を JDK の jre/lib/ext 拡張ディレクトリにコピーするという別の方法もあります。たとえば、JDK 1.4.2、バージョン 4 の場合は、*jdkroot*/j2sdk1.4.2_04/jre/lib/ext ディレクトリにコピーします。

HTML フォームおよびリクエスト・パラメータの使用方法

一般的なサーブレットは、表示または操作するサーブレットの情報の入力をユーザーに求めます。サーブレットは、HTML フォームを使用して情報を取得し、その情報を HTTP リクエスト・オブジェクトのパラメータに保存し、その情報をサーバーに送信することができます。または、リクエスト・オブジェクトからその他の情報（使用されているプロトコル、HTTP メソッド、リクエスト URI など）を取得することもできます。

次の各項で、いくつかの例を示します。

- HTML フォームによるユーザー入力
- ユーザー入力で指定されたリクエスト・パラメータ・データの表示
- フォームおよびリクエスト・パラメータを使用する例の全体
- URL セキュリティのための POST メソッドの使用
- リクエスト・オブジェクトの情報メソッドのコール
- リクエスト情報を取得する例の全体

HTTP リクエスト・パラメータは、静的リソース（たとえば、.html ファイルなど）へのリクエストのディスパッチ前に実行することになっているサーブレット・フィルタでは使用できません。サーブレットまたは JSP ページのような動的リソースの前に実行するフィルタからは、パラメータにアクセスが可能です。

リクエスト・メソッドの要約は、1-6 ページの「HttpServletRequest インタフェースの主要メソッド」を参照してください。

HTML フォームによるユーザー入力

サーブレットは、HTML フォームを使用してユーザーから入力を取得し、これらのデータを HTTP リクエスト・オブジェクトのパラメータとしてサーバーに送信することができます。次に例を示します。

```
PrintWriter out = response.getWriter();
...
out.print("<form action=\"");
out.print("RequestParamExample\" ");
out.println("method=GET>");
out.println("Enter a new first name: ");
out.println("<input type=text size=20 name=firstname>");
out.println("<br>");
out.println("Enter a new last name: ");
out.println("<input type=text size=20 name=lastname>");
out.println("<br>" + "<br>");
out.println("<input type=submit>");
out.println("</form>");
```

この例では、ユーザーが名前と名字を入力するように求められ、名前は `firstname` というリクエスト・パラメータに、名字は `lastname` というリクエスト・パラメータにそれぞれ格納されます。リクエスト・オブジェクトはサーバーに送信され、必要に応じてその情報がサーバーで処理されます（次の項を参照）。

ただし、この操作に `GET` メソッドを使用することの重大な短所は、パラメータ名および値がサーブレット URL 文字列に追加されることです。これを避けるために、6-11 ページの「URL セキュリティのための POST メソッドの使用」で示すように、かわりに `POST` メソッドを使用することができます。

ユーザー入力で指定されたリクエスト・パラメータ・データの表示

この項では、HTML フォームを使用してユーザーが指定したリクエスト・パラメータ・データ（前の項を参照）を表示するサンプル・コードを示します。

```
PrintWriter out = response.getWriter();
...
String firstName = request.getParameter("firstname");
String lastName = request.getParameter("lastname");
out.println("First and last name from request:" + "<br>" + "<br>");
if (firstName != null || lastName != null) {
    out.println("First name ");
    out.println(" = " + firstName + "<br>");
    out.println("Last name ");
    out.println(" = " + lastName + "<br>");
} else {
    out.println("(No names entered. Please enter first and last name.)");
}
```

リクエスト・パラメータの `firstname` と `lastname` の値は、文字列の `firstName` と `lastName` に格納され、ユーザーに出力されます。

フォームおよびリクエスト・パラメータを使用する例の全体

この項では、これまでの項で説明したコードからなるサーブレットの全体を示します。このサーブレットは、ユーザーに名前と名字の入力を求め、その情報はリクエスト・オブジェクトに書き込まれます。サーブレットは、リクエスト・オブジェクトから名前と名字を取得して、ユーザーに出力します。(出力コードが最初に実行されます。このコードは、ユーザーが最初に名前や名字を入力するまで、名前が入力されていないことを示す「No names entered」というメッセージを表示します。)

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

public class RequestParamExample extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");

        out.println("<h3>" + "My Request Parameter Example" + "</h3>");
        String firstName = request.getParameter("firstname");
        String lastName = request.getParameter("lastname");
        out.println("First and last name from request:" + "<br>" + "<br>");
        if (firstName != null || lastName != null) {
            out.println("First name ");
            out.println(" = " + firstName + "<br>");
            out.println("Last name ");
            out.println(" = " + lastName + "<br>");
        } else {
            out.println("(No names entered. Please enter first and last name.)");
        }
        out.println("<P>");
        out.print("<form action=\"");
        out.print("RequestParamExample\" ");
        out.println("method=GET>");
        out.println("Enter a new first name: ");
        out.println("<input type=text size=20 name=firstname>");
        out.println("<br>");
        out.println("Enter a new last name: ");
        out.println("<input type=text size=20 name=lastname>");
        out.println("<br>" + "<br>");
        out.println("<input type=submit>");
        out.println("</form>");

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

サーブレットを初めて起動すると、次の画面が表示されます。

My Request Parameter Example

First and last name from request:

(No names entered. Please enter first and last name.)

Enter a new first name:

Enter a new last name:

Submit Query

「Jimmy」および「Geek」と入力して「**Submit Query**」をクリックすると、次の画面が表示されます。

My Request Parameter Example

First and last name from request:

First name = Jimmy
Last name = Geek

Enter a new first name:

Enter a new last name:

Submit Query

開発のヒント： このサーブレットは HTTP `GET` メソッドを使用するため、リクエスト・パラメータ名および値がサーブレット URL に追加されます。この例では、文字列の「`?firstname=Jimmy&lastname=Geek`」が追加されます。これを避ける方法は、次項の 6-11 ページの「URL セキュリティのための POST メソッドの使用」を参照してください。

URL セキュリティのための POST メソッドの使用

前の例は、HTTP GET メソッドを使用するため、リクエスト・パラメータ名および値がサーブレット URL に追加されます。これを避ける（通常、セキュリティを考慮して）ために、かわりに POST メソッドを使用することができます。次のコードでは、フォームで POST メソッドを使用し、doPost() メソッドを使用して doGet() メソッドをコールするように、前の例が変更されています。変更箇所は、**太字**で強調されています。

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

public class RequestParamExample extends HttpServlet {


    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");

        out.println("<h3>" + "My Request Parameter Example" + "</h3>");
        String firstName = request.getParameter("firstname");
        String lastName = request.getParameter("lastname");
        out.println("First and last name from request:" + "<br>" + "<br>");
        if (firstName != null || lastName != null) {
            out.println("First name ");
            out.println(" = " + firstName + "<br>");
            out.println("Last name ");
            out.println(" = " + lastName + "<br>");
        } else {
            out.println("(No names entered. Please enter first and last name.)");
        }
        out.println("<P>");
        out.print("<form action=\"");
        out.print("RequestParamExample\" ");
        out.println("method=POST>");
        out.println("Enter a new first name: ");
        out.println("<input type=text size=20 name=firstname>");
        out.println("<br>");
        out.println("Enter a new last name: ");
        out.println("<input type=text size=20 name=lastname>");
        out.println("<br>" + "<br>");
        out.println("<input type=submit>");
        out.println("</form>");

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

    public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        doGet(request, response);
    }
}
```

開発のヒント： この例では、doPost() を直接使用するかわりに、依然として doGet() メソッドが使用されています。これは、ブラウザが GET リクエストを使用するためです。

リクエスト・オブジェクトの情報メソッドのコール

1-6 ページの「HttpServletRequest インタフェースの主要メソッド」には、HTTP リクエストに関する情報を取得するために使用できるリクエスト・オブジェクトのメソッドがいくつか示されています。この項では、リクエスト・オブジェクトの情報メソッドをコールして情報を出力するサンプル・コードを示します。

```
PrintWriter out = response.getWriter();
...
out.println("Method:");
out.println(request.getMethod());
out.println("Request URI:");
out.println(request.getRequestURI());
out.println("Protocol:");
out.println(request.getProtocol());
```

この例は、HTTP メソッド（GET、POST など）、リクエスト URI（この例では、コンテキスト・パスとサーブレット・パスで構成されます）、プロトコル（HTTP など）を取得して表示します。次の項には、例の全体が示されています。

リクエスト情報を取得する例の全体

この項では、HTTP メソッド、リクエスト URI およびプロトコルを取得して HTML 表に出力するサーブレットの全体を示します。

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

public class RequestInfoExample extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html;charset=UTF-8");

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");

        out.println("<h3>" + "My Request Info Example" + "</h3>");
        out.println("<table border=0><tr><td>");
        out.println("Method:");
        out.println("</td><td>");
        out.println(request.getMethod());
        out.println("</td></tr><tr><td>");
        out.println("Request URI:");
        out.println("</td><td>");
        out.println(request.getRequestURI());
        out.println("</td></tr><tr><td>");
        out.println("Protocol:");
        out.println("</td><td>");
        out.println(request.getProtocol());
        out.println("</td></tr>");
        out.println("</table>");
```

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

このコードを実行すると、次のような画面が表示されます。

My Request Info Example

Method: GET
Request URI: /servlet/RequestInfoExample
Protocol: HTTP/1.1

インクルードおよび転送による他のサーブレットへのディスパッチ

多くのサーブレットは、別のサーブレットのレスポンスをインクルードしたり、別のサーブレットにリクエストを転送するなど、別のサーブレットを処理時に使用します。次の項では、これらの機能について説明し、例を示します。

- インクルードおよび転送の基本
- インクルードおよび転送を使用する理由
- インクルードまたは転送プロセスの手順
- サーブレット・インクルードの例の全体

注意： サーブレットだけでなく JSP ページもインクルードまたは転送のターゲットとすることができます。これ以降、ターゲット・サーブレットについて説明している箇所は、ターゲット JSP ページにも当てはまります。

インクルードおよび転送の基本

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、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。リクエスト・ディスパッチャは、HTTP リクエストを代替サーブレットにディスパッチするメカニズムです。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 URI 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を実行します。

- `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、サーブレット・リクエストおよびレスポンス・オブジェクトを渡します。

インクルードおよび転送を使用する理由

サーブレット・インクルードは、次のことを実行する際に役立つ方法です。

- 既存のコードを変更せずに再利用する。
- サーブレットごとにコードを実装せずに、同じ処理または出力を複数のサーブレットにインクルードする。
- 静的ファイルのコンテンツをインクルードする。

元のサーブレットの出力に加えて、ターゲット・サーブレットの出力もインクルードします。

これらのことは、サーブレット転送にも同様に当てはまりますが、転送の場合は、元のサーブレットの出力に「加えて」ではなく、ターゲット・サーブレットの出力が元のサーブレットの出力の「かわり」であることに注意してください。

インクルードまたは転送プロセスの手順

この項では、インクルードまたは転送を実装する基本手順を示します。

1. サーブレット（`javax.servlet.Servlet` インタフェースで示される）の `getServletConfig()` メソッドを使用して、サーブレット構成オブジェクトを取得します。

   ```
   ServletConfig config = getServletConfig();
   ```

2. サーブレット構成オブジェクトの `getServletContext()` メソッドを使用して、サーブレットのサーブレット・コンテキスト・オブジェクトを取得します。

   ```
   ServletContext context = config.getServletContext();
   ```

3. サーブレット構成オブジェクトの `getRequestDispatcher()` または `getNamedDispatcher()` メソッドを使用して、`RequestDispatcher` オブジェクトを取得します。`getRequestDispatcher()` の場合は、ターゲット・サーブレットの URI パスを指定し、`getNamedDispatcher()` の場合は、`web.xml` ファイル内の関連する `<servlet-name>` 要素に基づいて、ターゲット・サーブレットの名前を指定します。

   ```
   RequestDispatcher rd = context.getRequestDispatcher("path");

   RequestDispatcher rd = context.getNamedDispatcher("name");
   ```

4. 必要に応じて、リクエスト・ディスパッチャの `forward()` または `include()` メソッドを使用して、インクルードまたは転送をそれぞれ実行します。サーブレット・リクエストおよびレスポンス・オブジェクトを渡します。

   ```
   rd.include(request, response);

   rd.forward(request, response);
   ```

次の例のように、4 つの手順すべてを単一の文に組み合せることができます。

```
getServletConfig().getServletContext().getRequestDispatcher
                              ("path").include(request, response);
```

注意： リクエスト・オブジェクト（HttpServletRequest インスタンス）の getRequestDispatcher() メソッドを使用してリクエスト・ディスパッチャを取得することもできます。

次の項には、サーブレット・インクルードの例の全体が示されています。

サーブレット・インクルードの例の全体

この項では、別のサーブレットの出力をインクルードするサーブレットの例の全体を示します。6-12 ページの「リクエスト情報を取得する例の全体」に示されている RequestInfoExample クラスが、6-6 ページの「単純なサーブレットの例」に示されている HelloWorld クラスを多少変更したものからの出力をインクルードするように更新されています。

次に、Hello World の例を多少変更したものを示します。このクラスの出力がインクルードされます。このクラスは、HelloIncluded と呼ばれます。パッケージには含まれません。

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

public class HelloIncluded extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Hello World!</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Hi Amy!</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

次に、更新されたリクエスト情報クラスの例を示します。このクラスは、RequestInfoWithInclude と呼ばれ、HelloIncluded からの出力をインクルードします。主要なコードは、**太字**で強調されています。

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

public class RequestInfoWithInclude extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html;charset=UTF-8");

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");

        out.println("<h3>" + "My Request Info Example" + "</h3>");
        out.println("<table border=0><tr><td>");
        out.println("Method:");
        out.println("</td><td>");
```

```
        out.println(request.getMethod());
        out.println("</td></tr><tr><td>");
        out.println("Request URI:");
        out.println("</td><td>");
        out.println(request.getRequestURI());
        out.println("</td></tr><tr><td>");
        out.println("Protocol:");
        out.println("</td><td>");
        out.println(request.getProtocol());
        out.println("</td></tr>");
        out.println("</table>");

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

        getServletConfig().getServletContext().getRequestDispatcher
                        ("/mypath/helloincluded").include(request, response);
    }
}
```

パスの `/mypath/helloincluded` は、コンテキスト・パスおよびサーブレット・パスで構成される URI です。アプリケーションは、次のように、`HelloIncluded` を直接リクエストすることもできるように構成されているものとします。

`http://host:port/mypath/helloincluded`

関連情報については、第 2 章「サーブレットのデプロイおよび起動」を参照してください。

同様に、次の例のように、サーブレットのかわりに JSP ページをインクルードすることもできます。

```
        getServletConfig().getServletContext().getRequestDispatcher
                        ("/mypath/hello.jsp").include(request, response);
```

`RequestInfoWithInclude` を実行すると、次のような画面が表示されます。

My Request Info Example

Method:	GET
Request URI:	/servlet/RequestInfoWithInclude
Protocol:	HTTP/1.1

Hi Amy!

前処理および後処理のためにフィルタを使用する場面

リクエスト・オブジェクトとレスポンス・オブジェクトは、通常、サーブレット・コンテナとサーブレット間で直接受け渡されます。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、リクエストをラップして前処理するため、またはレスポンスをラップして後処理するために、サーブレットとサーブレット・コンテナの中間に置くことができる Java プログラムです。フィルタは、サーブレット構成でフィルタがマップされているリソースについてのリクエストが存在する場合に起動されます。

フィルタは、リクエストやレスポンスを効率的に変換することができます。サーブレットのグループの前処理または後処理を適用する場合は、フィルタを使用してください。(1 つのサーブレットについてのみリクエストまたはレスポンスを変更する場合は、フィルタの作成は不要です。サーブレット自体に直接必要な変更を加えることができます。)

リソースにアクセスしたりそのリソースへのリクエストを起動前に前処理したりする場面や、カスタマイズされたリクエスト・オブジェクトまたはレスポンス・オブジェクトでリクエストやレスポンスをそれぞれラップする場面などでフィルタを使用できます。指定された順番で一連のフィルタを使用してサーブレットに対して操作することができます。

暗号化フィルタがその一例です。アプリケーション内のサーブレットは、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確立さている場合は、クリアテキスト形式でネットワーク上に発信しないようにする必要があります。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。)その他の例は、認証、ロギング、監査、データ圧縮およびキャッシュのためのフィルタです。

詳細は、第 4 章「サーブレット・フィルタの理解および使用方法」を参照してください。

サーブレット通知にイベント・リスナーを使用する場面

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。リスナーを実装して、アプリケーション・イベント、セッション・イベントまたはリクエスト・イベントをアプリケーションに通知することができます。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。

次のいずれかをアプリケーションに通知する理由がある場合は、イベント・リスナーを使用してください。

- サーブレット・コンテキストの場合 :
  - サーブレット・コンテキストが新たに作成されたか、シャットダウンされようとしている。
  - サーブレット・コンテキスト属性が追加、削除または置換された。
- セッションの場合 :
  - セッションが新たに作成されたか、新たに無効またはタイムアウトになった。
  - セッション属性が追加、削除または置換された。
  - セッションが新たにアクティブまたは非アクティブになった。
  - セッションからオブジェクトが新たにバインドまたはアンバインドされた。
- リクエストの場合 :
  - リクエストが新たに処理された。
  - リクエスト属性が追加、削除または置換された。

例として、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。サーブレット・コンテキストのライフ・サイクル・イベント・リスナーを作成して、データベース接続を管理することができます。このイベント・リスナーは次のように機能します。

1. イベント・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. イベント・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、イベント・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・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・コンテナは適切なイベント・リスナー・メソッドをコールします。

詳細は、第 5 章「イベント・リスナーの理解および使用方法」を参照してください。

スタック・トレースの表示方法

例外が発生しても処理するためのエラー・ページが存在しない場合に表示されるエラー・メッセージで、次の点が変更されました。

次のエラー・メッセージが、セキュリティ依存例外の場合に表示されます。

サーブレット・エラー : 例外が発生しました。セキュリティ上の理由により、このレスポンスには含まれません。詳細はアプリケーション・ログを確認してください。

その他の例外の場合は、次のエラー・メッセージが表示されます。

サーブレット・エラー : 例外が発生しました。現在のアプリケーション・デプロイメント・ディスクリプタでは例外をこのレスポンスに含めることはできません。詳細はアプリケーション・ログを確認してください。

実行するテストが、例外またはスタック・トレースの表示に依存する場合は、アプリケーションを開発者モードで実行すると、スタック・トレースを表示できます。

アプリケーションを開発モードで実行するには、orion-web.xml ファイルで <orion-web-app> 要素の development 属性を "true" に設定します。

次に例を示します。

```
<?xml version="1.0"?>
<orion-web-app
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion-web-10_0.x
sd"
      deployment-version="null"
      deployment-time="1152223108413"
      jsp-cache-directory="./persistence"
       jsp-cache-tlds="standard"
      temporary-directory="./temp"
      context-root="/DevelopmentThrowException"
      schema-major-version="10"
      schema-minor-version="0"
      development="true">
<!-- Uncomment this element to control web application class loader behavior.
 <web-app-class-loader search-local-classes-first="true"
include-war-manifest-class-path="true" />
 -->
```

```
    <web-app>
    </web-app>
</orion-web-app>}}
```

アプリケーションの Apache Tomcat から OC4J への移行

この項では、サーブレットまたは JSP モジュール、あるいはその両方をインクルードする Web アプリケーションの Apache Tomcat から OC4J への移行に関する情報について説明します。次の項目について説明します。

- Tomcat から OC4J への移行の指針
- Tomcat および OC4J での JNDI ルックアップ
- Tomcat から OC4J への JSP コンパイルの問題
- Tomcat から OC4J へのクラスタリングの問題

Tomcat から OC4J への移行の指針

この項では、Java サーブレットの Tomcat から Oracle Application Server への移行に関する指針を示します。この項には、次の項目が含まれます。

- 概要
- サーブレットの移行手順
- 単純なサーブレットの移行
- WAR ファイルの移行
- 展開 Web アプリケーションの移行
- フィールドからのヒント

概要

Java サーブレットは Tomcat から OC4J へ簡単に移行でき、Tomcat 環境で選択した内容に応じて、移行したサーブレットへのコードの変更はほとんど必要ないか、まったく必要ありません。

Oracle Application Server 10*g* リリース 3（10.1.3.x）は、Sun 社の J2EE サーブレット仕様、バージョン 2.4 に完全に準拠しています。Tomcat 5.5 も、バージョン 2.4 と互換性があります。

また、Oracle Application Server 10*g* リリース 3（10.1.3.x）は、サーブレット 2.3 と下位互換があります。つまり、標準 2.3 仕様に書き込まれたサーブレットは、OC4J で正常に作動し、移行の手間は最小限で済みます。

サーブレットを新しい環境に移行する場合の主なタスクは、構成とデプロイです。

サーブレットの移行に関係するタスクは、サーブレットのパッケージとデプロイの方法によって異なります。サーブレットは、単純なサーブレットとして、標準ディレクトリ構造内でその他のリソースとともにパッケージされる Web アプリケーションとして、または Web アーカイブ（WAR）ファイルとしてデプロイできます。

サーブレットの移行手順

次に、サーブレットを OC4J に移行する一般的な手順を示します。

1. 構成 : サーブレットに関する Oracle Application Server のデプロイメント・ディスクリプタを作成または変更します。
2. パッケージ :
   - 単純なサーブレットは個別にデプロイできます。
   - サーブレットは、WAR ファイル内の Web アプリケーションの一部としてパッケージできます。
3. デプロイ : Application Server Control コンソールを使用して、WAR ファイル内のサーブレットをデプロイできます。個別のサーブレットおよび展開 Web アプリケーション内のサーブレットは、適切なディレクトリにコピーすることで自動的にデプロイされます。

Oracle JDeveloper のツールとウィザードを使用すると、これらの手順を自動化できます。

単純なサーブレットの移行

単純なサーブレットは、簡単に OC4J で構成およびデプロイできます。手動でのサーブレットのデプロイ・プロセスは、Tomcat と OC4J で同一です。

注意 : サーブレットをデプロイする場合の推奨方法は、サーブレットを WAR または EAR ファイルにパッケージすることと、Oracle Enterprise Manager 10*g* Application Server Control コンソールを使用することです。

それ以外に、`admin_client.jar` コマンドライン・ユーティリティを使用して、手動でデプロイすることもできます。この章で説明する XML ファイルの編集、および Java コマンドを使用したコマンドラインでの OC4J の起動についての手動プロセスは、開発環境で使用してください。`admin_client.jar` の情報は、『Oracle Containers for J2EE 構成および管理ガイド』および『Oracle Containers for J2EE デプロイメント・ガイド』を参照してください。

サーブレットは、Web アプリケーションの一部として登録および構成する必要があります。サーブレットを登録および構成する場合は、Web アプリケーション・デプロイメント・ディスクリプタにエントリをいくつか追加する必要があります。

次に、単純なサーブレットをデプロイする一般的な手順を示します。

1. Web アプリケーション・デプロイメント・ディスクリプタ（`web.xml`）を、サーブレット・クラスの名前およびサーブレットのリクエストを解決する際に使用する URL パターンで更新します。
2. サーブレット・クラス・ファイルを、`WEB-INF/classes/` ディレクトリにコピーします。サーブレット・クラス・ファイルにパッケージ文が含まれる場合、パッケージ文の各レベル別に、追加のサブディレクトリを作成します。次に、サーブレット・クラス・ファイルは、そのパッケージのために作成した最下層のサブディレクトリに置く必要があります。
3. サポートするユーティリティ・クラス・ファイルおよびサーブレットで必要なその他のサポートするファイルを Oracle Application Server インストールの適切なディレクトリに展開してコピーします。
4. ホーム OC4J の home インスタンスを起動または再起動します。
5. URL を入力して、ブラウザからサーブレットを起動します。

WAR ファイルの移行

Web アプリケーションは、WAR ファイルとして構成およびデプロイできます。OC4J でこの処理を最も簡単に行うためには、Application Server Control コンソール管理 GUI を使用します。または、WAR ファイルを適切なディレクトリに手動でコピーします。これは、Tomcat でも同様です。

注意： WAR ファイルを適切なディレクトリに手動でコピーする場合は、OC4J がスタンドアロン・モード（Oracle Application Server インスタンスのコンポーネントではなく）である開発環境でのみ行うことができます。

本番 Web アプリケーションは、通常に WAR または EAR ファイルを使用して、Oracle Application Server コンソールまたはユーティリティを介してデプロイします。Web アプリケーションの開発中、展開ディレクトリ形式を使用すると編集済コードのデプロイおよびテストが短時間でできます。

デプロイの詳細は、『Oracle Containers for J2EE デプロイメント・ガイド』を参照してください。

次に、WAR ファイルを Tomcat から OC4J に移行する一般的な手順を示します。

1. サンプル・アプリケーションの WAR ファイルを作成します。
2. サンプル・アプリケーションを OC4J にデプロイします。
3. デプロイ済のアプリケーションをテストします。

展開 Web アプリケーションの移行

Web アプリケーションは、標準ディレクトリ構造または展開ディレクトリ形式に格納されたファイルの集合として構成およびデプロイもできます。OC4J でこの処理を行うためには、標準ディレクトリ構造のコンテンツを OC4J インストール内の適切なディレクトリに手動でコピーします。同じ方法を、Tomcat で使用できます。

主に Web アプリケーションの開発中に、Web アプリケーションを展開ディレクトリ形式でデプロイします。これは、変更を短時間で簡単にデプロイおよびテストできる方法です。本番 Web アプリケーションをデプロイする場合は、Web アプリケーションを WAR ファイルにパッケージし、Application Server Control コンソールで WAR ファイルをデプロイします。

次に、OC4J で展開 Web アプリケーションを手動でデプロイする一般的な手順を示します。

1. 展開 Web アプリケーションがある最上位のディレクトリを、使用している OC4J インストールの次のディレクトリにコピーします。
   `<ORACLE_HOME>/j2ee/home/applications`

   次に、アプリケーション・デプロイメント・ディスクリプタ `<ORACLE_HOME>/config/application.xml` を次のように変更して、Web アプリケーションをインクルードします。

   ```
   <web-module id="migratedHR" path="../applications/hrapp" />
   ```

2. 次のように、ディスクリプタ・ファイル `<ORACLE_HOME>/config/default-web-site.xml` にエントリを追加して、Web アプリケーションを Web サイトにバインドします。

   ```
   <web-app application="default" name="migratedHR " root="/hr" />
   ```

3. 最後に、`<ORACLE_HOME>/config/server.xml` ファイルに、新しい `<application>` タグ・エントリを追加して、新しいアプリケーションを登録します。

`server.xml` を変更して、それを保存する場合、OC4J はこのファイルのタイムスタンプの変更を検出し、アプリケーションを自動的にデプロイします。OC4J を再起動する必要はありません。

フィールドからのヒント

Tomcat から OC4J に移植する際に、留意すべき問題を説明します。

- パス名での開始のスラッシュ（/）の使用
- JNDI コンテキスト・ファクトリ・サマリー
- Xerces および Xalan で必要な追加手順

パス名での開始のスラッシュ（/）の使用

Tomcat は、`ServletContext` クラスの次のメソッドに関しては、サーブレット仕様に完全に準拠していません。

- `ServletContext.getResource()`
- `ServletContext.getResourceAsStream()`

メソッド `getResource()` および `getResourceAsStream()` は、パス・パラメータをとります。このパスにより、`WEB-INF/config.xml` のような Web アプリケーション・ディレクトリ構造内のファイルにアクセスできます。

J2EE API ドキュメントおよび仕様では、次のように記述されています。

「パスは（/）で始まる必要があり、これは現在のコンテキスト・ルートと相対的であると解釈されます。」

Tomcat では、サーブレット仕様とは異なり、パス名をスラッシュ（/）で始める必要がありません。

サーブレット仕様完全準拠の OC4J は、最初にスラッシュ（/）が必要です。

`ServletContext` クラスのファイルにアクセスする際には、パス名の最初に必ずスラッシュ（/）を付けるようにしてください。

JNDI コンテキスト・ファクトリ・サマリー

OC4J では、複数の異なる JNDI コンテキスト・ファクトリの使用が可能で、それぞれのファクトリは `InitialContext` を 1 つ作成し、異なる機能を持っています。

次のコンテキスト・ファクトリが、最も一般的に使用されます。

- 内部コンテキスト・ファクトリ : OC4J コンテナ内から `InitialContext` のデフォルト・コンストラクタをクラス・パスに `jndi.properties` を含めずにコールする場合に、内部コンテキスト・ファクトリを取得します。
- `RMIInitialContextFactory`: OC4J コンテナに接続する場合、このコンテキスト・ファクトリを一般的に使用します。すべての構成が無視され、`java:comp/env/` 接頭辞は必要ありません。
- `ApplicationClientContextFactory`: このコンテキスト・ファクトリは JNDI 環境（特に、`META-INF/application-client.xml` において）を処理し、`java:comp/env/` 環境接頭辞を必要とします。

Xerces および Xalan で必要な追加手順

コードが Xerces/Xalan に依存する場合、追加の手順が必要です。

この内容に関する詳細は、Metalink を参照してください。

Tomcat および OC4J での JNDI ルックアップ

Tomcat では、`java:`*`comp`*`/`*`env`*`/`*`ResourceName`* を使用して、`web.xml` 内の `<resource-ref>` 要素を定義せずに、リソースを参照できます。

OC4J では、`web.xml` 内の `<resource-ref>` 要素を必要とするか、またはルックアップは単に *`ResourceName`* のみです。

OC4J では、サーバーで定義されるデータソースに対して、デフォルトでこれを実行できます。

```
initialContext.lookup("jdbc/ScottDS")
```

一方、Tomcat では次のようになります。

```
initialContext.lookup("java:comp/env/jdbc/ScottDS")
```

コードを変更せずに Tomcat オプションを使用する必要がある場合、次のように `web.xml` を変更して、リソースに `<resource-ref>` エントリをインクルードします。

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

Tomcat から OC4J への JSP コンパイルの問題

一般的に、OC4J の J2EE 仕様の解釈は Tomcat よりも厳密です。

カスタム Tomcat JSP タグを標準 JSP タグに置き換えてから、OC4J で JSP ページをデプロイすると、コンパイルの際の問題を回避できます。

仕様に「should」および「may」の語が使用される場合は、注意してください。このような場合、選択が可能であり、その結果の動作が異なるため、このような場合は注意が必要です。

次に例を示します。JSP 仕様では、次のように述べられています。

> 指定したタイプ（指定した場合）またはクラス（タイプが指定されていない場合）のスクリプト言語変数は、スクリプト言語の現行の字句範囲内の指定の ID により定義されます。タイプ属性は、JavaBean としてインスタンス化できない Java タイプの指定に使用する必要があります（たとえば、抽象クラス、インタフェース、または引数のない public コンストラクタを持たないクラスである Java タイプ）。クラス属性が JavaBean としてインスタンス化できない Java タイプに使用される場合、コンテナはページが無効であると判断するので、変換時に致命的変換エラーまたはリクエスト時に java.lang.Instantiation-Exception を生成することが推奨されます（ただし、必須ではありません）。

この場合、OC4J では、JavaBean が `zero-args` コンストラクタを持たない場合、`class` 属性ではなく `type` 属性を使用する必要があるという推奨事項を実装します。Tomcat では、`type` 属性は要求されませんが、`class` 属性を受け入れます。この差異により、Tomcat から OC4J にアプリケーションを移行するときに次の問題が発生します。

`index.jsp` で `type` 属性を使用して、JavaBean（`my.MyClass`）が public の `no-args` コンストラクタを持たない場合、ページは Tomcat および OC4J の両方で正常に動作します。これが望ましい動作です。

望ましい使用方法の例を次に示します。

```
<jsp:useBean id="codeDesc" scope="session" type="my.MyClass"/>
```

一方、この状況で `class="my.MyClass"` 属性を使用すると、Tomcat はこの使用を受け入れ、正常に動作します。しかし、OC4J は仕様をより厳密に解釈するので、OC4J では `JSPCompilationException` がスローされます。

あまり厳密ではない使用方法の例を次に示します。

```
<jsp:useBean id="codeDesc" scope="session" class="my.MyClass"/>
```

Tomcat から OC4J へのクラスタリングの問題

この項では、OC4J と Tomcat のクラスタリングの相互関係の概要を説明します。次の項目について説明します。

- Tomcat および OC4J の基本構成
- Tomcat および OC4J のネットワークの考慮事項
- Tomcat および OC4J の状態永続性メカニズム
- Tomcat および OC4J のレプリケーション・アルゴリズム
- 状態レプリケーション送信
- Tomcat および OC4J でのアプリケーション設計
- Tomcat および OC4J のロード・バランシング

注意： この項の記述は、Tomcat 5 の動作を基にしています。

OC4J では、クラスタリングは、デプロイされている場合、アプリケーションごとの基準で定義されます。

構成の方法など、OC4J のクラスタリングの詳細は、『Oracle Containers for J2EE 構成および管理ガイド』の第 9 章「OC4J でのアプリケーションのクラスタリング」を参照してください。

ロード・バランシングの詳細は、『Oracle HTTP Server 管理者ガイド』の付録 D を参照してください。

Tomcat および OC4J の基本構成

Tomcat では、server.xml ファイル内の <cluster> 要素およびそのサブ要素を使用してクラスタリングを構成します。クラスタリングは、コンテナごとに有効になります。

OC4J では、デプロイされたアプリケーションの orion-application.xml ファイル内の <cluster> 要素を使用して、アプリケーションごとにクラスタリングを定義します。

これにより、有効化されたクラスタリングを使用および使用せずに、アプリケーションが同一の OC4J インスタンス内に共存できるようになります。対象のアプリケーションの orion-application.xml ファイルを使用して、OC4J 内のクラスタリングを構成します。デフォルト・アプリケーションのクラスタリングを有効にすると、すべてのデプロイ済のアプリケーションがその設定を継承するので、コンテナ・レベル・クラスタリングが有効になります。また、OC4J では、アプリケーションごとにクラスタリングを構成するため、異なるプロトコルおよび異なるオプションを使用し、異なるアプリケーションを構成してクラスタリングを有効化できます。

Tomcat および OC4J のネットワークの考慮事項

Tomcat では、IP マルチキャストおよび IP ソケットの両方を使用して、容易にクラスタリングできます。用語の面では、Tomcat には単一メモリー内のレプリケーション・オプションがあり、これにより 2 つのネットワーク・モデルを結合します。

- IP マルチキャストは、Tomcat インスタンスが互いに可用性を検出および確認する場合など、グループ・メンバーシップ操作を実行する場合に使用されます。
- IP ソケットは、1 つの Tomcat インスタンスから別の Tomcat インスタンスへのセッション状態の実際のレプリケーションを実行する場合に使用されます。通常、これはメモリー内の状態の永続性メカニズムのためにあります。

OC4J の場合は、メモリー内セッション状態レプリケーションの使用方法および用語が異なります。OC4J でセッション状態のメモリー内をレプリケートする方法は 2 つあり、IP マルチキャス

トまたは IP ソケットのいずれも使用できます。どちらの場合も、グループ・メンバーシップ・アクティビティは同一のネットワークを使用します。

- IP マルチキャスト

  OC4J を構成して、IP マルチキャストをレプリケーション・プロセスとして使用する場合、OC4J は IP マルチキャストをグループ・メンバーシップ・プロトコルに参加する手段として使用し、セッション状態をグループの他のメンバーに配布します。マルチキャスト・ネットワーク・アドレスは、既知の値をデフォルトにします。これは、<cluster> 要素の <multicast> サブ要素を使用して、任意に構成できます。OC4J ではマルチキャスト・レプリケーション・プロトコルには保証付き配信の追加された値があるため、実際に使用されているネットワーク・モデルに信頼性保証が備わっていない場合でも、OC4J での使用方法により、すべてのパケットが配信され、セッション状態が失われないことが保証されます。

- IP ソケット

  OC4J を構成して、ピアツーピアをレプリケーション・プロトコルとして使用する場合、OC4J は IP ソケットをグループ・メンバーシップ・プロトコルに参加する手段として使用し、セッション状態を選択したピアに配布します。OC4J を独立して使用する場合（スタンドアロン）、ピアのリストは構成ファイル内で静的に定義する必要があります。TCO ソケット・アドレスのデフォルトは、既知の値です。これは、クラスタ・プロトコル、タグのサブ要素を使用して構成できます。

  OC4J を Oracle Application Server 環境で使用する場合（クラスタ化）、ピアの初期リストが OPMN サーバーから提供されます。また、OPMN はピア・レプリケーション・プロトコルで使用されるポート番号を割り当てるため、複数の OC4J インスタンスを起動したサーバー上でポートの競合が発生しません。

Tomcat および OC4J の状態永続性メカニズム

Tomcat は次のメカニズムをサポートして、セッション状態のその他の Tomcat インスタンスへの配布を処理します。

- メモリー内
- データベース
- ファイルベース

OC4J は、次のセッション永続性メカニズムをサポートします。

- メモリー内
- データベース

OC4J がサポートするメモリー内ベースのレプリケーションに 2 つの方法（マルチキャストおよびピア）があることと、Tomcat は両方を組み合せて使用して、メモリー内レプリケーションをサポートすることを、混同しないようにしてください。

OC4J は、ファイルベース状態レプリケーション・プロトコルはサポートしません。

OC4J では、「状態永続性」ではなく「レプリケーション・プロトコル」の用語を使用します。

- Tomcat では、状態永続性メカニズムを定義するには、永続性メカニズムの実装に使用するクラス名を指定します。

- OC4J では、類似の概念であるレプリケーション・プロトコルは、`<protocol>` タグを使用し、`<multicast>`、`<peer>` または `<database>` サブタグのいずれか 1 つを指定してどのプロトコルを使用するかを示し、`orion-application.xml` ファイル内の値として構成されます。

  データベース・レプリケーション・プロトコルには、データソースの JNDI ロケーションが必要で、これが状態のレプリカが格納されるデータベース・インスタンスを示します。

Tomcat および OC4J のレプリケーション・アルゴリズム

Tomcat では、レプリケーションされた状態のタイプは、完全なセッションまたは変更セットだけで構成されます。レプリケーション・タスクの実行に使用されるクラスを指定して、構成します。

OC4J は、完全なセッションまたはデルタのみのいずれかを送信する同様の概念をサポートします。`<replication-policy>` 要素および `scope` 属性を使用して、構成することができます。オプションは、`modifiedAttributes` または `allAttributes` です。

状態レプリケーション送信

Tomcat では、状態レプリケーションの送信は、同期、非同期またはプールされるよう構成されます。

OC4J はデフォルトでは、非同期レプリケーション・モデルをレプリケーション送信に使用します。これは同期レプリケーション・オプションを使用して同期に変更可能で、レプリカを送信する OC4J インスタンスは、受信 OC4J からの確認を待ってから続行します。

Tomcat および OC4J でのアプリケーション設計

クラスタ化環境で実行するアプリケーションに対する、OC4J および Tomcat のアプリケーション設計要件は共通です。

Tomcat および OC4J のロード・バランシング

Tomcat は組込み HTTP サーバーを提供します。ただし、汎用 Apache HTTP サーバーを `mod_proxy` または `mod_rewrite`、または AJP ベースの `mod_jk` コネクタで使用して拡大し、複数の Tomcat インスタンス間をルーティングすることができます。

OC4J もまた組込み HTTP サーバーを提供します。ただし、提供される自動ロード・バランシング機能を拡大し、その恩恵を受けるには、Oracle HTTP サーバーを使用することをお薦めします。OC4J HTTP を使用するには、サード・パーティのロード・バランシング・コンポーネントが必要です。

Oracle Application Server 環境では、OC4J インスタンスで実行中のアプリケーションにリクエストをルーティングするために、Oracle HTTP Server が使用されます。Oracle HTTP Server は、同一のクラスタ・トポロジで実行中の OC4J インスタンスを検出するよう構成されます。アプリケーションは OC4J インスタンスにデプロイされるため、Oracle HTTP Server は新しいデプロイの更新を自動的に受信し、アプリケーション・コンテキスト・ルートをルーティング可能な URL のリストに自動的に追加します。

Oracle HTTP Server は、セッション・ベースのリクエストはフェイルオーバーが発生するまで常に同一の OC4J インスタンスにルーティングされるよう、スティッキー・セッションをサポートするクラスタ化されたアプリケーションの自動フェイルオーバーを提供します。

Oracle HTTP Server は、デフォルトでラウンドロビン・ロード・バランシング・アルゴリズムを使用します。ロード・バランシング・モデルは、いくつかの異なるオプションに変更できます。

詳細は、『Oracle HTTP Server 管理者ガイド』の付録 D「mod_oc4j を使用するロード・バランシング」を参照してください。

7 サービスおよびリソース参照の注釈の使用

OC4J は、サーブレット 2.5 仕様に記載される注釈をサポートし、コンポーネント・インスタンスがアプリケーションで使用可能になる前に、リソース参照を挿入します。次の項では、OC4J での注釈の使用方法について説明します。

- 注釈機能の概要
- 注釈および挿入
- OC4J の注釈
- 注釈ルールとガイドライン
- サーブレットのバージョン 2.5 のパフォーマンスへの注釈の影響
- 注釈の例

注釈機能の概要

J2SE 5.0 以上では、外部リソースの構成データおよび依存性を、Java コードで注釈と呼ばれるメタデータとして指定できます。そのようなデータは、構成ファイル内に定義するか、EJB や Web サービスなどのサービスまたはデータソースや JMS 宛先などのリソース参照の注釈内に定義することができます。

たとえば、J2EE 1.4 では、サーブレットが EJB を参照する前に、開発者は次のような `ejb-local-ref` 要素を定義する必要があります。

```
<ejb-local-ref>
 <ejb-ref-name>ejb/HelloWorld</ejb-ref-name>
 <local>oracle.ejb.HelloWorld</local>
</ejb-local-ref>
```

次に、コード内の EJB を参照するために、開発者は次のコードの一部分として JNDI を使用する必要があります。

```
Context ic = new InitialContext();

HelloWorld helloWorld = (HelloWorld)ic.lookup("java:comp/env/ejb/HelloWorld");
helloWorld.greet("Hello!");
```

サーブレット 2.5 では、クライアント・コードは単純化され、OC4J は正確なリソースまたはサービスを挿入します。クライアントは、リソースまたはサービスを、次のような注釈に指定する以外、何も必要ありません。

```
@EJB
private HelloWorld helloWorld;
helloWorld.greet("Hello!");
```

注釈および挿入

Web アプリケーションのデプロイメント・ディスクリプタ内の `web-app` 要素の `metadata-complete` 属性で、Web ディスクリプタおよびその他のこのモジュールの関連デプロイメント・ディスクリプタ（たとえば Web サービス・ディスクリプタ）が完了しているかどうかを指定します。`web.xml` ファイルで `version="2.5"` を設定してサーブレット 2.5 が使用されるか、サーブレット 2.5 スキーマ・ネームスペースが指し示される場合、OC4J サーブレット・コンテナは `metadata-complete` フラグを確認して、注釈を処理するかどうか判断します。`version` が `2.4` 以前のバージョンに設定されている場合、サーブレット・コンテナはいずれの注釈も処理しません。

`metadata-complete` がサーブレット 2.4 のデフォルト値である `true` に設定されている場合、サーブレット・コンテナはアプリケーションのクラス・ファイル内にあるすべてのサーブレット注釈を無視します。`metadata-complete` 属性がない、またはサーブレット 2.5 のデフォルト値である `false` に設定されており、`version` が `2.5` に設定されている場合、サーブレット・コンテナはサーブレット注釈のアプリケーションのクラス・ファイルを検証し、次のように注釈をサポートします。

1. OC4J サーブレット・コンテナは、`WEB-INF/` ディレクトリ内または `WEB-INF/lib/` ディレクトリの下の JAR ファイルにある Web アプリケーションのクラスに対する注釈参照のためにリソースまたはサービス参照を検査します。

   また、サーブレット・コンテナは、`MANIFEST.MF` 内にリストされる JAR ファイルへの注釈サポートも提供します。

2. OC4J サーブレット・コンテナは、Web アプリケーション・デプロイメント・ディスクリプタ内で宣言され、次のインタフェースを実装する管理されたコンポーネント・クラスに対する注釈サポートを提供します。

   - javax.servlet.Servlet
   - javax.servlet.Filter
   - javax.servlet.ServletContextListener
   - javax.servlet.ServletContextAttributeListener
   - javax.servlet.ServletRequestListener
   - javax.servlet.ServletRequestAttributeListener
   - javax.servlet.http.HttpSessionListener
   - javax.servlet.http.HttpSessionAttributeListener

3. OC4J サーブレット・コンテナは、すべてのライフサイクル・メソッドがインスタンスでコールされる前にリソースまたはサービス参照を挿入します。

   - javax.servlet.Servlet および javax.servlet.Filter に対する init()
   - javax.servlet.ServletContextListener に対する contextInitialized()
   - javax.servlet.ServletRequestListener に対する requestInitialized()

4. 注釈およびデプロイメント・ディスクリプタ・エントリの両方によって環境エントリが宣言されている場合、デプロイメント・ディスクリプタ・エントリの情報で Resource、EJB または WebServiceRef 注釈内の情報の一部をオーバーライドできます。

   次の仕様では、デプロイメント・ディスクリプタ・エントリを使用して注釈情報をオーバーライドする場合のルールが説明されています。

   - Resource 注釈をオーバーライドするルールは、Java EE 5 仕様を参照してください。
   - EJB 注釈をオーバーライドするルールは、EJB 仕様を参照してください。
   - WebServiceRef 注釈をオーバーライドするルールは、WebServiceRef 仕様を参照してください。

5. OC4J サーブレット・コンテナで、クラス用に挿入する必要があるリソースまたはサービスが見つからない場合は、クラスの初期化が失敗し、OC4J は次の警告メッセージを発行します。

   ```
   Some resource(s) and/or service(s) to be injected cannot be found for class name.
   class name will not be put to service.
   ```

6. すべてのリソースまたはサービス参照が挿入された後、いずれのライフサイクル・メソッドがインスタンスでコールされる前に、PostConstruct 注釈でマークされたメソッド（ある場合）を起動して、挿入したリソースを初期化する機会をサーブレットに与える必要があります。7-6 ページの「PostConstruct 注釈」を参照してください。

7. サーブレットをサービスから取り出す前に、PreDestroy 注釈でマークされたメソッド（ある場合）を起動して、挿入したリソースを解放する機会をサーブレットに与える必要があります。7-6 ページの「PreDestroy 注釈」を参照してください。

OC4J の注釈

この項では、OC4J がサポートする注釈を説明します。

EJB 注釈

Resource 注釈

Resources 注釈

PostConstruct 注釈

PreDestroy 注釈

PersistenceUnit(s) 注釈

PersistenceContext(s) 注釈

WebServiceRef 注釈

DeclaresRoles 注釈

RunAs 注釈

EJB 注釈

アプリケーション・コンポーネントのフィールドまたはメソッドの EJB 注釈は、デプロイメント・ディスクリプタの ejb-ref または ejb-local-ref 要素と同等です。フィールドに EJB 注釈がある場合、OC4J はフィールドに対応する EJB コンポーネントへの参照を挿入します。

EJB 注釈では、Bean のローカルまたはリモート・ホーム・インタフェースまたは EJB 3 Bean のビジネス・インタフェースを参照できます。EJB 3 のビジネス・インタフェースを参照する場合、OC4J は Enterprise Bean のインスタンスへの参照を挿入します。

短い EJB 注釈の例を次に示します。

```
@EJB private ShoppingCart myCart;
```

すべての注釈フィールドを使用する長い EJB 注釈の例を次に示します。

```
@EJB(
   name = "ejb/shopping-cart",
   beanName = "Cart1",
   beanInterface = ShoppingCart.class,
   description = "Items for purchase"
)
private ShoppingCart myCart;
```

EJB 注釈の詳細は、EJB 仕様を参照してください。

7-9 ページの「注釈の例」に示す例には、EJB 注釈が含まれています。

Resource 注釈

Resource 注釈を使用して、データソースや JMS 宛先、環境エントリなどのリソースに対する参照を宣言します。この注釈を使用する場合、デプロイメント・ディスクリプタ内の <resource-ref>、<message-destination-ref>、<env-ref>、または <resource-env-ref> 要素で参照を宣言する必要はありません。

Resource 注釈をフィールドまたは setter メソッドに適用する場合、OC4J は注釈で宣言されたリソースに対する参照を挿入し、リソースの JNDI 名に参照をマッピングします。注釈をクラスに適用する場合、注釈はアプリケーションが実行時に参照するリソースを宣言します。

挿入はそれぞれ JNDI ルックアップに対応します。注釈で JNDI 名を明示的に指定しない場合、クラスの完全修飾名と結合したフィールド名を JNDI 名として使用します。たとえば、パッケージ com.example 内のクラス MyApp の myDb という名のフィールドのデフォルト JNDI 名は、java:comp/env/com.example.MyApp/myDb です。すべての JNDI 名は、java:comp/env/ に対して相対的です。注釈を setter メソッドに適用する場合、クラス名で修飾されるメソッドに対応する JavaBean プロパティ名がデフォルトです。注釈をクラスに適用する場合、デフォルトは存在しないため、名前を指定する必要があります。

次の例では、JNDI 名が指定されていないため、OC4J はデフォルトの JNDI 名を使用します。

```
@Resource
private DataSource myDB;
```

次の例では、JNDI 名が明示的に指定されています。

```
@Resource(name="customerDB")
private DataSource myDB;
```

注釈に関する一般情報は、Java EE 5 仕様を参照してください。

7-9 ページの「注釈の例」に示す例には、Resources 注釈が含まれています。

Resources 注釈

注釈の繰返しは認められていないため、Resources 注釈は次の形式で複数の Resource 注釈のコンテナとして機能します。

```
public @interface Resources {
    Resource[] value;
}
// value - Array of multiple resources.
```

次の例では、データソースおよびコネクション・ファクトリのために 2 つの Resource 注釈をクラスに含む Resources 注釈を示します。

```
@Resources ({
    @Resource (name = "myDB", type=javax.sql.DataSource),
    @Resource (name = "myCF", type=javax.jms.ConnectionFactory)
)
public class MulResClass {
    //...
}
```

PostConstruct 注釈

すべてのその他のリソース挿入が完了した後、いずれかのライフサイクル・メソッドがインスタンスでコールされる前に、OC4J はメソッドを PostConstruct 注釈を使用して起動します。これにより、コンポーネントは作成後の処理が可能になります。クラスに他の注釈がない場合でも、OC4J はこのメソッドを起動します。

PostConstruct 注釈の例を次に示します。

```
@PostConstruct
void doPostInjectionProcessing {
   //...
```

注意： PostConstruct 注釈により、サーブレットの任意のメソッドが init() メソッドとして機能します。

PreDestroy 注釈

OC4J は、サーブレットをサービスから取り出す前に、PreDestroy 注釈を使用してメソッドを起動します。これにより、サーブレットは挿入されたリソースを解放します。クラスに他の注釈がない場合でも、OC4J はこのメソッドを起動します。

PreDestroy 注釈の例を次に示します。

```
@PreDestroy
void doPreDestroyProcessing {
   //...
   }
```

注意： PreDestroy 注釈により、サーブレットの任意のメソッドが destroy() メソッドとして機能します。

PersistenceUnit(s) 注釈

PersistenceUnit 注釈は、EJB 3.0 永続性を使用する際に必要です。永続性ユニットは、関連するエンティティ Bean のセットを管理するエンティティ・マネージャ（永続性コンテキスト）に関する構成の詳細を持ちます。永続性コンテキストに対する注釈は、7-7 ページの「PersistenceContext(s) 注釈」を参照してください。

サーブレットのフィールドまたはメソッドに、PersistenceUnit 注釈を使用して注釈を付けられます。論理永続性ユニット参照は、永続性ユニットのエンティティ・マネージャ・ファクトリを参照します。

単一の永続性ユニットを宣言する例を次に示します。

```
@PersistenceUnit
EntityManagerFactory emf;
```

サーブレットで複数の永続性ユニットを宣言する場合、次のように、各ユニットに明示的 unitName を指定する必要があります。

```
@PersistenceUnit(unitName="InventoryManagement")
   EntityManagerFactory emf;
```

PersistenceUnit 注釈は、persistence.xml 内の persistence-unit-ref 要素と同じです。

永続性ユニット参照の詳細は、Java EE 5 仕様を参照してください。

PersistenceContext(s) 注釈

PersistenceContext 注釈は、EJB 3.0 永続性を使用する際に必要です。永続性コンテキストは、一連の関連するエンティティ Bean を管理するエンティティ・マネージャです。永続性ユニットは、永続性コンテキストに関する構成の詳細を持ちます。永続性ユニットに対する注釈の詳細は、7-6 ページの「PersistenceUnit(s) 注釈」を参照してください。
PersistenceContext 注釈の例を次に示します。

```
@PersistenceContext
EntityManager em;
```

サーブレットで複数の永続性ユニットを宣言する場合、次のように unitName 属性で明示的なユニット名を指定する必要があります。

```
@PersistenceContext(unitName="InventoryManagement")
EntityManager em;
```

この注釈は、persistence.xml 内の persistence-context-ref 要素と同じです。

永続性コンテキスト参照の詳細は、Java EE 5 仕様を参照してください。

WebServiceRef 注釈

WebServiceRef 注釈を使用した注釈付けは、Web サービスへの参照を提供するデプロイメント・ディスクリプタ内の <resource-ref> の宣言と同じです。

この注釈には、2 つの主な用途があります。

- タイプが生成されるサービス・インタフェースである参照を定義します。
- タイプがサービス・エンドポイント・インタフェース（SEI）である参照を定義します。

WebServiceRef 注釈の例を次に示します。

```
// Generated Service Interface
@WebServiceRef
private StockQuoteService stockQuoteService;

// SEI
@WebServiceRef(StockQuoteService.class)
private StockQuoteProvider stockQuoteProvider;
```

WebServiceRef 注釈の詳細は、Java API for XML-Based Web Services 2.0（JSR 224）を参照してください。

DeclaresRoles 注釈

DeclaresRoles 注釈は、アプリケーションのセキュリティ・モデルを構成するすべてのセキュリティ・ロールを定義します。この注釈をクラスで指定し、注釈を付けたクラスのメソッド内からテスト可能なロールを isCallerInRole をコールして定義することができます。

DeclaresRoles 注釈の例を次に示します。

```
@DeclaresRoles("Manager")
public class CorporationServlet {
   //...
}
```

この注釈は、web.xml ファイルの次のコードと同じです。

```
<web-app>
   <security-role>
      <role-name>Manager</role-name>
   </security-role>
</web-app>
```

DeclaresRoles 注釈の詳細は、Common Annotations for the JavaTM Platform（JSR250）を参照してください。

RunAs 注釈

RunAs 注釈は、デプロイメント・ディスクリプタの <run-as> 要素と同じです。この注釈は、javax.servlet.Servlet インタフェースを実装するクラスまたはそのサブクラスでのみ使用できます。

RunAs 注釈の例を次に示します。

```
@RunAs("Admin")
public class CorporationServlet {
   //...
}
```

この注釈は、web.xml ファイルの次のコードと同じです。

```
<servlet>
   <servlet-name>CorporationServlet</servlet-name>
   <run-as>Admin</run-as>
</servlet>
```

RunAs 注釈の詳細は、Common Annotations for the JavaTM Platform（JSR250）を参照してください。

注釈ルールとガイドライン

注釈を使用する場合のルールをいくつか説明します。一般的な注釈のガイドラインの詳細は、Java EE 5 仕様を参照してください。

- 次のコンテナ管理コンポーネント・クラスのフィールドまたはメソッドには、コンポーネントの環境のエントリをクラスに挿入するリクエストをするように注釈を付けられます。

  ```
  javax.servlet.Servlet
  javax.servlet.Filter
  javax.servlet.ServletContextListener
  javax.servlet.ServletContextAttributeListener
  javax.servlet.ServletRequestListener
  javax.servlet.ServletRequestAttributeListener
  javax.servlet.http.HttpSessionListener
  javax.servlet.http.HttpSessionAttributeListener
  ```

- フィールドまたはメソッドには、任意のアクセス修飾子（たとえば、public、private など）を付けられます。
- フィールドまたはメソッドは静的にできませんが、挿入用に注釈を付けられているアプリケーション・クライアントのメイン・クラスのフィールドまたはメソッドは例外です。このようなフィールドまたはメソッドは静的にする必要があります。
- 前述のルールに違反すると、エラーが発生し、メッセージが記録されます。違反したクラスは、使用できません。
- クラスに注釈が適用されると、リソース挿入は行われません。かわりに、アプリケーション・コンポーネントの環境にアプリケーション・コンポーネントが JNDI またはコンポーネント・コンテキスト・ルックアップ・メソッドを通じて参照できるエントリを宣言します。
- Resource 注釈は、次の例に示すように、このリストの最初の項目のいずれかのクラス、またはそれらのスーパークラスのいずれかで指定できます。リソースの挿入は、フィールドおよびメソッドの可視性についての Java 言語オーバーライド・ルールに従います。

  ```
  Class A {

     @Resource
     private DataSource myDS; // Injected resource not visible in class B
     //...
  }
  ```

```
Class B extends Class A {
   public DataSource myDS; // Not a resource, needs to do JNDI lookup
   //...
}
```

サーブレットのバージョン 2.5 のパフォーマンスへの注釈の影響

注釈を、サーブレットのバージョン 2.5 で Web アプリケーションに使用するかどうかにかかわらず、`<web-app>` の `metadata-complete` 属性がデプロイメント・ディスクリプタにない場合または `false` に設定されている場合（サーブレット 2.5 のデフォルト値）、OC4J は `WEB-INF/` および `WEB-INF/lib` の下のすべてのクラスをロードして、注釈を検索する必要があります。ロードを行うと、起動時に Oracle Application Server のパフォーマンスに影響する場合があります。注釈を使用しない場合、`metadata-complete="true"` を指定することでパフォーマンスへの影響を回避できます。

起動時のパフォーマンスに影響を与えるのは、サーブレットのバージョン 2.5 を使用する Web アプリケーションのみです。サーブレットのバージョン 2.4 以下を使用する Web アプリケーションでは、パフォーマンスに影響はありません。

注釈の例

この項では、注釈および対応する `web.xml` ファイルを使用したサーブレットの例を示します。

サーブレットはサーブレット 2.5 の方法を示し、コメントアウトされた部分は比較のためのサーブレット 2.4 の方法です。サーブレットは、1 つの `@EJB` 注釈および 2 つの `@Resource` 注釈を示します。

`web.xml` ファイルは、注釈を使用できるようにするために必要な `version=` および `metadata-complete` 設定を説明します。

```
<web-app version="2.5" metadata-complete="false">
 ...
</web-app>
```

詳細は、次のサイトのドキュメント『How-To: Using Dependency Injection In Web Module』を参照してください。

http://www.oracle.com/technology/tech/java/oc4j/10131/how_to/index.html

次に、web.xml ファイルの例を示します。

```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
version="2.5"
        metadata-complete="false"
>

    <display-name>Annotation Example</display-name>
<description>A few examples of Servlet 2.5 Annotation and Resource
Injection</description>

    <servlet>
        <display-name>hello</display-name>
        <servlet-name>HelloServlet</servlet-name>
        <servlet-class>HelloServlet</servlet-class>
    </servlet>
```

```
    <servlet-mapping>
        <servlet-name>HelloServlet</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>

  <env-entry>
    <env-entry-name>EmpNo</env-entry-name>
    <env-entry-type>java.lang.Integer</env-entry-type>
    <env-entry-value>15</env-entry-value>
  </env-entry>
</web-app>
```

次に、HelloServlet.java サーブレットの例を示します。

```
import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.annotation.Resource;
import javax.sql.*;
import java.sql.*;
import javax.naming.*;
import org.acme.*;
import javax.rmi.*;
import javax.ejb.*;
public class HelloServlet extends HttpServlet {

    @EJB HelloObject bean;
    @Resource(name="EmpNo") int empNo;
    @Resource(name="jdbc/OracleDS") private DataSource db;

    public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
        res.setContentType("text/html");

        PrintWriter writer = res.getWriter();

        // ejb invocation
        writer.println(bean.sayHello()+"<br>");

        // fetch value set by deployer
        writer.println("EmpNo="+empNo+"<br>");

        // make a db connection
        writer.println("db="+getConnection()+"<br>");
    }

    public Connection getConnection() {
      Connection conn = null;

      try {
          if (db != null) {
              conn = db.getConnection();
          }
      } catch (Exception e) {
          e.printStackTrace();
      }
      return conn;
    }

    // The following is the pre-2.5 way to do it.
```

```
/*

   public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
        res.setContentType("text/html");

        PrintWriter writer = res.getWriter();

        InitialContext ctx = new InitialContext();
        Object obj = ctx.lookup("java:comp/env/ejb/Hello");

        HelloHome ejbHome = (HelloHome)
                    PortableRemoteObject.narrow(obj,HelloHome.class);
        HelloObject bean = ejbHome.create();

        // ejb invocation
        writer.println(bean.sayHello()+"<br>");

        // fetch value set by deployer
        Context myEnv = (Context) ctx.lookup("java:comp/env");
        Integer empNoInteger = (Integer) myEnv.lookup("EmpNo");
        int empNo = empNoInteger.intValue();

        writer.println("EmpNo="+empNo+"<br>");

        // make a db connection
        writer.println("db="+getConnection()+"<br>");
    }

    public Connection getConnection() {

        Context initCtx = new InitialContext();
        javax.sql.DataSource db =
           (javax.sql.DataSource) initCtx.lookup("java:comp/env/jdbc/OracleDS");

        Connection conn = null;

        try {
            if (db != null) {
                conn = db.getConnection();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return conn;
    }
*/
}
```

8 JDBC または Enterprise JavaBeans の使用方法

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 API である JDBC をサーブレットで使用する方法について説明します。また、Enterprise JavaBeans の概要も示します。Enterprise JavaBeans は、サーバー・サイド・ビジネス・ロジックを実行したり、アプリケーションのデータ永続性を管理するために、サーブレットからコールすることができます。この章には、次の項が含まれます。

- サーブレットでの JDBC の使用方法
- Enterprise JavaBeans の概要

サーブレットでの JDBC の使用方法

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、データベース接続に OC4J データソースを使用し、Java Naming and Directory Interface（JNDI）を使用してデータソースをルックアップすることをお薦めします。次の項では、関連する基本手順を説明し、この機能の例を示します。

- JDBC を使用する理由
- データソースおよびリソース参照の構成
- JDBC コールの実装
- データベース問合せサーブレットの例
- TopLink サーブレットの例

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。

JDBC を使用する理由

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して動的出力を作成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 を生成してそれをクライアントに返すことや、HTTP リクエスト内でサーブレットに渡された情報に基づいてデータベースを更新することが可能です。

JDBC は、データベースにアクセスするための標準 Java メカニズムです。

注意：

- 全般的な前提として、Oracle データベースと Oracle JDBC ドライバを使用するものとします。非 Oracle データベースへの接続には、Oracle Application Server で提供されている DataDirect JDBC ドライバを使用できます。
- サーブレットから直接 JDBC を使用するかわりに、EJB を使用してデータにアクセスできます。8-10 ページの「Enterprise JavaBeans の概要」も参照してください。

データソースおよびリソース参照の構成

データベース接続には、通常、標準データソースが使用されます。この項では、JNDI によって使用できるデータソースを構成する手順を説明します。

1. データソースの構成
2. リソース参照の構成

データソースとその OC4J での構成の詳細は、『Oracle Containers for J2EE サービス・ガイド』を参照してください。

データ ソースの構成

データソースを使用するには、データソースを主要 OC4J データソース構成に追加する必要があります。一般に、この手順は、Oracle Enterprise Manager 10*g* Application Server Control を使用して実行します。

Application Server Control コンソールで、次の手順を実行します。

1. 該当する「アプリケーション」ホームページから、または OC4J ホームページから、「管理」タブを選択します。
2. 「JDBC リソース」というタスクに移動します。
3. 「JDBC リソース」ページから、データソースを作成できます。前に作成したデータソースを編集することもできます。また、このページから接続プールを編集することもできます。

データソースを構成すると、次のフォームに従って、`j2ee/home/config/data-sources.xml` ファイルのエントリが（この例では、Oracle JDBC シン・ドライバを使用するために）新しく追加または更新されます。次の点に注意してください。

- `<connection-pool>` 要素は、JDBC 接続プール用の設定を持ち、プールの名前を指定します。（接続プール機能は、接続オブジェクトの既存のプールから接続を取得して、新しい接続オブジェクトを作成するオーバーヘッドを回避することにより、パフォーマンスを向上させます。）
- `<connection-pool>` の `<connection-factory>` サブ要素は、接続のファクトリ（この場合、通常、データソースを表すクラス）として使用するクラスとデータベース・ユーザー名、パスワードおよび接続文字列を指定します。
- `<managed-data-source>` 要素は、データソースの名前（`name`）と JNDI の場所（`jndi-name`）を指定し、`<connection-pool>` 要素で指定された接続プールを参照します。

8-5 ページの「問合せサーブレットのデータソースの構成」の例を参照してください。

```
<data-sources ... >
    <connection-pool name="poolname">
        <connection-factory factory-class="package.Classname"
                            user="user"
                            password="password"
                            url="jdbc:oracle:thin:@host:port/service"/>
    </connection-pool>
    <managed-data-source connection-pool-name="poolname"
                         jndi-name="jndiname"
                         name="name"/>
</data-sources>
```

注意： `url` エントリについては、*host*:*port*:*sid* 形式もまだサポートされていますが、廃止される可能性があります。

リソース参照の構成

データソースおよび JNDI ルックアップを使用するには、web.xml ファイルに適切なリソース参照エントリも存在する必要があります。次に、前の項で示したデータソース構成の例に対応する例を示します。

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

これにより、データソースとして使用するために、jdbc/OracleDS リソースが DataSource タイプであることが確立されます。

注意： <res-auth> 要素には必ず Container 設定を使用してください。これにより、アプリケーション・コンポーネント・コードではなくコンテナがリソースへのサインオンを実行することが示されます。

JDBC コールの実装

この項では、サーブレット・コードで JDBC によってデータベースにアクセスする一般的な手順を示します。例の全体は、8-6 ページの「問合せサーブレットの作成」を参照してください。

1. 必要なパッケージをインポートします。サーブレットおよび java.io パッケージに加えて、JDBC、データソースおよび JNDI のクラスを含むパッケージが存在します。

```
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.*;
```

2. データソースの JNDI ルックアップを実行し、データベース接続を確立するために、init() メソッドを try...catch ブロック内に実装します。このルックアップは、8-2 ページの「データソースおよびリソース参照の構成」で示した例に対応します。

```
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);
  }
}
```

3. 適切なサーブレットの do*XXX*() メソッド（doGet() など）を実装し、JDBC を使用して必要な SQL 操作を実行します。この例では、SQL 問合せ文字列が文字列の query で構成されているものとします。コードは、JDBC 文オブジェクトを作成し、問合せを実行し、データ・レコードを出力するために結果セットを調べて（ここで out は PrintWriter オブジェクト）、文および結果セット・オブジェクトをクローズします。SQL 操作も、try...catch ブロック内で実行されます。

```
try {
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery(query);
  while (rs.next()) {
     out.println(rs.getString(1) + rs.getInt(2));
  }
```

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

4. データベース接続を閉じる destroy() メソッドを（同じく try...catch ブロック内に）実装します。

```
    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例

この例には、次の問合せを完成する LIKE 仕様の入力をユーザーに求める HTML「ようこそ」ページが含まれています。

```
SELECT ename, empno FROM emp WHERE ename LIKE xxx
```

「ようこそ」ページは、問合せを実行して結果を出力するサーブレットを起動します。

次の項で、例の実装および構成方法を示します。

- 問合せサーブレットのデータソースの構成
- HTML「ようこそ」ページの作成
- 問合せサーブレットの作成
- サーブレットおよび JNDI リソース参照の構成
- 問合せの例のパッケージ化
- 問合せの例の起動

問合せサーブレットのデータソースの構成

次に、この例のデータソース構成を示します。この構成は、OC4J の data-sources.xml ファイルに反映されており、Application Server Control コンソールを使用して構成できます（8-3 ページの「データソースの構成」を参照）。この例は、Oracle JDBC シン・ドライバを使用し、ポート 5521 経由で、ホスト myhost 上のデータベースにアクセスします。その際、サービス名 myservice を使用し、ユーザー scott として接続します。（これは単純化された例です。data-sources.xml でのパスワードの公開を避ける方法もあります。）またこの例では、接続プール、および接続の取得先のデータソースを表すクラス OracleDataSource を使用します。jndi-name エントリの jdbc/OracleDS は、データソースの JNDI ルックアップのためにサーブレットによって使用されます。

```
<data-sources>
       <connection-pool name="ConnectionPool1">
             <connection-factory factory-class="oracle.jdbc.pool.OracleDataSource"
                                 url="jdbc:oracle:thin:@myhost:5521/myservice"
                                 user="scott" password="tiger"/>
       </connection-pool>
       <managed-data-source connection-pool-name="ConnectionPool1"
                            jndi-name="jdbc/OracleDS" name="OracleDS"/>
</data-sources>
```

HTML「ようこそ」ページの作成

次に、「ようこそ」ページの `empinfo.html` を示します。このページは、ユーザーに問合せの完成を求め、問合せサーブレットを起動します。この例の場合、サーブレットは、`/myquery/getempinfo` というコンテキスト・パスおよびサーブレット・パスで起動されるようにデプロイされます。

```
<html>
<head>
<title>Query the Employees Table</title>
</head>
<body>
<form method=GET ACTION="/myquery/getempinfo">
The query is<br>
SELECT ename, empno FROM emp WHERE ename LIKE xxx

<p>
Specify the WHERE clause xxx parameter.<br>
Enclose entry in single-quotes; use % for wildcard. Search is case-sensitive.<br>
Example: 'S%' (for all names starting with 'S').<br>
<input type=text name="queryVal">
<p>
<input type=submit>
</form>
</body>
</html>
```

問合せサーブレットの作成

次に、8-4 ページの「JDBC コールの実装」で示した手順を実装する問合せサーブレットの `GetEmpInfo` を示します。この例には、出力される HTML 表の書式設定と、取得された行の数のカウンタも含まれています。

```
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 LIKE specification for the WHERE clause from the user, through the */
/* HTTP request, then construct the SQL query.                                 */
```

```
    String queryVal = req.getParameter("queryVal");
    String query =
      "select ename, empno from emp " +
      "where ename like " + queryVal;

    resp.setContentType("text/html");

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

/* Create a JDBC statement object and execute the query.     */
    try {
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery(query);

/* HTML table formatting for the output.                     */
      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. Using 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();
    }
  }
}
```

サーブレットおよび JNDI リソース参照の構成

サーブレットの構成に加えて、web.xml ファイルには、データソースのリソース参照エントリが含まれている必要があります。また、初期ファイルとして empinfo.html を宣言するための構成も含まれています。次に、この例のファイルを示します。

```
<?xml version="1.0" ?>
<!DOCTYPE web-app (doctype...)>
<web-app>
   <servlet>
      <servlet-name>empinfoquery</servlet-name>
      <servlet-class>GetEmpInfo</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>empinfoquery</servlet-name>
      <url-pattern>getempinfo</url-pattern>
   </servlet-mapping>
   <resource-ref>
      <res-auth>Container</res-auth>
      <res-ref-name>jdbc/OracleDS</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
   </resource-ref>
   <welcome-file-list>
      <welcome-file>empinfo.html</welcome-file>
   </welcome-file-list>
</web-app>
```

問合せの例のパッケージ化

この例の WAR ファイル（empinfo.war）は、次のコンテンツと構造を持ちます。

```
empinfo.html
META-INF/Manifest.mf
WEB-INF/web.xml
WEB-INF/classes/GetEmpInfo.class
WEB-INF/classes/GetEmpInfo.java
```

また、EAR ファイルは、次のとおりです。

```
empinfo.war
META-INF/Manifest.mf
META-INF/application.xml
```

（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。）

問合せの例の起動

この例では、application.xml がコンテンツ・パスの /myquery を empinfo.war にマップするものとします。この場合、デプロイ後に、次のように、「ようこそ」ページの empinfo.html を起動することができます（web.xml で empinfo.html が「ようこそ」ページとして宣言されている必要があります)。

http://*host*:*port*/myquery

テスト実行では、S で始まるすべての名前を検索するために 'S%' を指定します。

The query is
SELECT ename, empno FROM emp WHERE ename LIKE xxx

Specify the WHERE clause xxx parameter.
Enclose entry in single-quotes; use % for wildcard. Search is case-sensitive.
Example: 'S%' (for all names starting with 'S').

'S%'

Submit Query

テスト実行に使用されたデータベースの場合は、これにより、2 つのエントリが返されました。

Last Name	Employee ID
SMITH	7369
SCOTT	7788

2 rows retrieved

TopLink サーブレットの例

サーブレットは、Oracle TopLink を使用して J2EE 永続性をアプリケーションに提供します。Oracle Technology Network の次の Web サイトで、TopLink サーブレットの例を参照できます。

http://www.oracle.com/technology/products/ias/toplink/examples/index.html

Enterprise JavaBeans の概要

サーブレットでは、Enterprise JavaBeans をコールして、データベースにアクセスすることや、追加のビジネス・ロジックを実行することができます。次の項で、EJB の概要と、サーブレットからの EJB の使用を説明します。

- Enterprise JavaBeans を使用する理由
- OC4J および Oracle Application Server での EJB のサポート
- サーブレットと EJB 間のルックアップの使用例
- EJB ローカル・インタフェースとリモート・インタフェース
- リモート・フラグによる同一アプリケーション内のリモート・ルックアップ

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 Containers for J2EE Enterprise JavaBeans 開発者ガイド』を参照してください。

注意 : 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 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。

Enterprise JavaBeans を使用する理由

EJB には、サーバー・サイド・ビジネス・ロジックに使用するセッション Bean やデータ永続性を管理するためのエンティティ Bean など、ビジネス・アプリケーションにおいて多くの用途があります。EJB テクノロジは、トランザクションによるセキュアなサーバー・サイド処理で使用するための、JSP またはサーブレット・テクノロジよりも堅牢なインフラストラクチャを提供します。

一般的なアプリケーション設計では、サーブレットは、HTTP リクエストを処理するフロントエンド・コントローラとして使用され、EJB は、データベースへのアクセスと更新を行うためにコールされ、最後に、別のサーブレットまたは JSP ページが、リクエスタのデータを表示するために使用されます。

EJB には、セッション Bean、エンティティ Bean およびメッセージドリブン Bean という 3 つのカテゴリがあります。特に、コンテナ管理の永続性エンティティ Bean は、永続データの管理に適しています。これは、データベースにアクセスする際に JDBC API を直接使用する必要がなくなるためです。かわりに、EJB コンテナで透過的にデータベース操作を処理できます。

セッション Bean は、ビジネス・ロジックのモデリングに役立ちます。また、ステートレスまたはステートフルのいずれかになります。ステートフルな Bean は、一般に、トランザクション状態がメソッド・コールまたはサーブレット・リクエスト間で維持される必要がある場合に使用されます。ステートレスな Bean には、アプリケーション状態とは関係のない個別のビジネス・ロジック・メソッドが含まれます。

OC4J および Oracle Application Server での EJB のサポート

OC4J は、セッション Bean、エンティティ Bean およびメッセージドリブン Bean を完全にサポートしています。エンティティ Bean 実装は、Bean 管理の永続性（BMP）、コンテナ管理の永続性（CMP）、ローカル・インタフェース、コンテナ管理の関連性、および EJB 問合せ言語による問合せ実行機能を提供します。

エンティティ Bean 実装内では、基本永続性マネージャが単純マッピングと複合マッピングの両方をサポートし、1 対 1、1 対多、多対 1 および多対多オブジェクト・リレーショナル・マッピングがサポートされます。また、エンティティ Bean のフィールドが対応するデータベース表に自動的にマップされます。

アプリケーションのメンテナンスとデプロイを容易にするために、Oracle Application Server は、動的 EJB スタブ生成などの多数の拡張機能を提供します。CORBA 相互運用性は、EJB を構築し、CORBA サービスとして CORBA クライアントから EJB にアクセスする機能を提供します。

サーブレットと EJB 間のルックアップの使用例

サーブレットからの EJB のコールには、3 つの使用例があります。

- 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。これには、EJB ローカル・インタフェースを使用します。
- 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（アプリケーションは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。
- 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。この場合、EJB リモート・インタフェースが必要です。EJB は異なるホストまたは同一の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。

サーブレットと EJB 間の通信では、ローカルおよびリモート EJB コールに JNDI が使用されます。リモート・ルックアップが実行される場合、JNDI は、Oracle による RMI 実装（ORMI）または標準で相互操作可能な Internet Inter-ORB Protocol（IIOP）のいずれかを使用します。3.0 よりも古いバージョンの EJB では、ホーム・インタフェースのみが JNDI ルックアップを必要としました。その後は、アプリケーションが使用する EJB を作成するために使用されています。J2EE コンポーネントは、デフォルトの `no-args` コンストラクタを使用して、同じアプリケーション内のオブジェクトをルックアップできます。リモート・ルックアップには、`RMIInitialContextFactory` または `IIOPInitialContextFactory` クラスを使用できます。OC4J の JNDI の詳細は、『Oracle Containers for J2EE サービス・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、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、各ホストで、アプリケーションの OC4J EJB `remote` フラグを正しく設定する必要もあります。8-13 ページの「リモート・フラグによる同一アプリケーション内のリモート・ルックアップ」を参照してください。

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。

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 仕様の初期バージョンでは、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` パラメータ（`orion-ejb-jar.xml` ファイルの `<session-deployment>` 要素にマップする）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 Containers for J2EE Enterprise JavaBeans 開発者ガイド』を参照してください。このパラメータが Application Server Control デプロイ・プラン・エディタで `copyByValue` として構成可能であることに注意してください。詳細は、『Oracle Containers for J2EE デプロイメント・ガイド』を参照してください。

より新しいバージョンの EJB 仕様は、同じ場所に配置されている EJB コールのローカル・インタフェースをサポートしています。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。

リモート・フラグによる同一アプリケーション内のリモート・ルックアップ

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）、OC4J EJB の `remote` フラグを各層で正しく設定する必要があります。サーバーでフラグが `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。

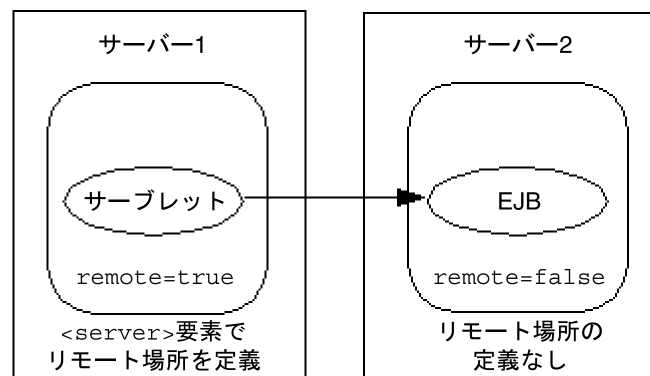
`remote` フラグは、`orion-application.xml` ファイルにある `<orion-application>` 要素のサブ要素 `<ejb-module>` 内の属性にマップします。デフォルトの設定は、`remote="false"` です。ファイルを更新して、次のように、このフラグを `true` に設定してください。

```
<orion-application ... >
   ...
   <ejb-module remote="true" ... />
   ...
</orion-application>
```

（Oracle Enterprise Manager 10*g* Application Server Control でこのフラグを設定することはできません。）

`false` のリモート・フラグ値で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てから、サーブレット層であるサーバー 1 でリモート・フラグ値を `true` に設定できます。図 8-1 にこれを示します。

図 8-1 アプリケーション内のリモート・ルックアップの設定



サーバー 2 で EJB を検索するように OC4J を設定するには、サーバー 2 をリモート・ホストとして正しく構成する必要があります。次のように、サーバー 1 の `rmi.xml` ファイルの該当する `<rmi-server>` 要素の `<server>` サブ要素で、`host`、`port`、`username` および `password` 設定を指定してください。

```
<rmi-server ... >
...
   <server host="remote_host" port="remote_port" username="user_name"
           password="password" />
...
</rmi-server>
```

リモート・ホストおよび `remote` フラグの使用方法の詳細は、『Oracle Containers for J2EE Enterprise JavaBeans 開発者ガイド』を参照してください。

注意： リモート・ホストのデフォルトの管理ユーザー名と、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。JAZN の詳細は、『Oracle Containers for J2EE セキュリティ・ガイド』を参照してください。

9

ベスト・プラクティスおよびパフォーマンス

この章では、サーブレットの効率とパフォーマンスを最大にするためのプログラミングのヒントを提供します。一般的な提案とともに、セッション、セキュリティおよびスレッド・モデルに関する提案を示します。また、パフォーマンスを監視するための Oracle ダイナミック・モニタリング・サービス（DMS）の概要も示します。次の項目について説明します。

- セッションのベスト・プラクティス
- セキュリティのベスト・プラクティス
- スレッド・モデルの考慮事項
- パフォーマンスのベスト・プラクティス
- パフォーマンスの監視

セッションのベスト・プラクティス

この項では、セッションを使用する際の考慮事項について説明します。

- 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。
- パフォーマンス上の理由で、OC4J はセッション状態のレプリケーションの成功を確認するまで待機しません。

OC4J でのクラスタリングの詳細は、『Oracle Containers for J2EE 開発者ガイド』を参照してください。

セキュリティのベスト・プラクティス

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は、次の点を考慮する必要があります。

- 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するために、`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>` サブ要素に適切な設定が反映されていることを確認します。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。Application Server Control デプロイ・プラン・エディタの `jaznWebApp` プロパティによって、これらを編集できます。詳細は、『Oracle Containers for J2EE デプロイメント・ガイド』を参照してください。
- 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。
- 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。（OC4J のシステム・プロパティに関する一般的な情報は、『Oracle Containers for J2EE 構成および管理ガイド』を参照してください。）

  – `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。Application Server Control デプロイ・プラン・エディタの `servletWebdir` プロパティによって、これを編集できます。

  （`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、2-10 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。

追加情報は、B-5 ページの「orion-web.xml および global-web-application.xml の要素と属性」に示されている参照ドキュメントも参照してください。

スレッド・モデルの考慮事項

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・コンテナが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主要メソッドの `service()` 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`

カスタム・スレッド・プール

1 つ以上のカスタム・スレッド・プールを作成して、デフォルト・スレッド・プールのかわりに選択したアプリケーションで使用できます。

server.xml ファイルでカスタム・スレッド・プールを作成し、それを 1 つ以上の *-web-site.xml ファイル内で参照してアプリケーションで使用できるようにします。

カスタム・スレッド・プールの作成

カスタム・スレッド・プールを作成するには、<custom-thread-pool> 要素を server.xml ファイルに追加します。<custom-thread-pool> 要素は、<application-server> 要素のサブ要素です。

名前の属性は必要ですが、それ以外の属性はすべてオプションです。<custom-thread-pool> 要素は、『Oracle Containers for J2EE 構成および管理ガイド』の第 10 章「タスク・マネージャとスレッド・プールの構成」で説明する他のスレッド・プール要素と同一の属性を持ちます。server.xml ファイルの詳細は、『Oracle Containers for J2EE 構成および管理ガイド』の付録 B「OC4J で使用される構成ファイル」の「OC4J サーバー構成ファイル（server.xml）の概要」を参照してください。

server.xml ファイルに mypool という名前のカスタム・スレッド・プールを作成する例を次に示します。この例では、mypool を次のように指定します。

- プール内に作成するスレッドの最小数は 10 です。
- プール内に作成できるスレッドの最大数は 100 です。
- それぞれのキュー内の未処理のリクエスト数は 50 です。
- アイドル・スレッドの存続期間は 700 秒です。

```
<application-server ...>
...
   <custom-thread-pool name="mypool" min="10" max="100"
    queue="50" keepAlive="700000" debug="true"/>
...
</application-server>
```

カスタム・スレッド・プールのアプリケーションへの割当て

アプリケーションにデフォルト・スレッド・プールのかわりにカスタム・スレッド・プールを使用するよ指示するには、1 つ以上の *-web-site.xml ファイル内の <web-site> 要素の custom-thread-pool 属性に対する該当するカスタム・スレッド・プールの参照を追加します。

それぞれの Web アプリケーションは、*-web-site.xml ファイルで指定された Web サイトおよびポートに、<web-app> 要素で命名されて割り当てられ、custom-thread-pool 属性で命名されたカスタム・スレッド・プールを使用します。

-web-site.xml ファイルについては、『Oracle Containers for J2EE 構成および管理ガイド』の付録 B「OC4J で使用される構成ファイル」の「Web サイト構成ファイル（-web-site.xml）の概要」を参照してください。

srdemo-web-site.xml ファイルの <web-app> 要素で命名するすべてのアプリケーションで使用される mypool カスタム・スレッド・プールの命名に使用する custom-thread-pool 属性の例を次に示します。

```
  <?xml version="1.0" ?>
- <web-site xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/web-site-10_0.xsd"
port="12501" protocol="ajp13" display-name="SRDEMO Web Site" custom-thread-pool="mypool"
schema-major-version="10" schema-minor-version="0">
  <default-web-app application="default" name="defaultWebApp" root="/j2ee" />
  <web-app application="system" name="dms" root="/dmsoc4j" access-log="false" />
  <web-app application="system" name="JMXSoapAdapter-web" root="/JMXSoapAdapter" />
  <web-app application="default" name="jmsrouter_web" load-on-startup="true" root="/jmsrouter" />
  <web-app application="ascontrol" name="ascontrol" load-on-startup="true" root="/em" />
  <web-app application="bc4j" name="webapp" load-on-startup="true" root="/webapp" />
  <web-app application="SRDEMO" name="SRDEMO-WEB" load-on-startup="true" root="/SRDEMO" />
  <access-log path="../log/default-web-access.log" split="day" />
  </web-site>
```

パフォーマンスのベスト・プラクティス

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> サブ要素に反映されているように、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。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。Application Server Control デプロイ・プラン・エディタの expirationSettings プロパティによって、これを編集できます。詳細は、『Oracle Containers for J2EE デプロイメント・ガイド』を参照してください。
- 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。9-3 ページの「スレッド・モデルの考慮事項」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。特に、global-web-application.xml または orion-web.xml 内の <orion-web-app> 要素の file-modification-check-interval 属性に反映されているように、ファイル変更チェック間隔の使用には注意が必要です。これは、Application Server Control コンソールの「構成プロパティ」ページ（A-5 ページの「「構成プロパティ」ページ」を参照）で、または Application Server Control デプロイ・プラン・エディタの fileModificationCheckInterval プロパティによって構成できます。

  また、default-web-site.xml 内の <web-site> 要素の use-keep-alives 属性にも注意が必要です。この属性については、『Oracle Containers for J2EE 構成および管理ガイド』を参照してください。
- 追加の JSP 関連構成パラメータ、特に、global-web-application.xml または orion-web.xml 内の <orion-web-app> 要素の simple-jsp-mapping および enable-jsp-dispatcher-shortcut 属性は、パフォーマンスに大きな影響を与える可能性があります。Application Server Control デプロイ・プラン・エディタの enableJspDispatcherShortcut および simpleJspMapping プロパティによって、これらを編集できます。
- スタンドアロン環境の OC4J 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。この機能は、default-web-site.xml 内の <web-app> 要素の shared 属性によって有効にすることができます。

- スタンドアロン OC4J を本番環境として使用する場合（通常は使用しない）は、`server.xml` の `<application-server>` 要素の `check-for-updates` フラグがオフになっていることを確認してください。このパラメータについては、『Oracle Containers for J2EE 構成および管理ガイド』および『Oracle Containers for J2EE デプロイメント・ガイド』を参照してください。

追加情報は、B-5 ページの「orion-web.xml および global-web-application.xml の要素と属性」に示されている参照ドキュメントも参照してください。

パフォーマンスの監視

この項では、サーブレットのパフォーマンスの監視について説明します。

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

Oracle Application Server 環境では、ダイナミック・モニタリング・サービス（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。

DMS モジュールの標準構成は、OC4J の `system-application.xml` ファイルおよび `default-web-site.xml` ファイルに反映されます。`system-application.xml` では、Web モジュールの `dms` と、その WAR ファイルへのパスが指定されます。`default-web-site.xml` ファイルは、この 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、Application Server Control を使用してください。

DMS の詳細は、『Oracle Application Server パフォーマンス・ガイド』を参照してください。グローバル `application.xml` ファイル（`orion-application.xml` ファイルと同じ仕様を使用）の詳細は、『Oracle Containers for J2EE 開発者ガイド』を参照してください。また、Web サイトの XML ファイルの詳細は、『Oracle Containers for J2EE 構成および管理ガイド』を参照してください。

A
Web モジュールの管理

この付録では、Oracle Enterprise Manager 10*g* Application Server Control を使用して Web モジュールを管理するための OC4J 機能の参照ドキュメントを提供します。

- Application Server Control コンソール トップレベルの「Web モジュール」ページ
- Application Server Control Web モジュールの構成ページ
- Web モジュールの MBean および管理のサマリー

Application Server Control コンソール トップレベルの「Web モジュール」ページ

Application Server Control コンソールは、アプリケーションをデプロイ、構成および監視できるだけでなく、アプリケーションによって使用される OC4J インスタンスおよび Web サービスを管理することもできる Web ベースのユーザー・インタフェースを提供します。このユーザー・インタフェースは、OC4J ソフトウェアを（スタンドアロン環境または Oracle Application Server 環境に）インストールすると自動的にインストールおよび事前構成され、起動されます。また、OC4J インスタンスが使用するポートにバインドされます。スタンドアロン環境では、通常、このポートは 8888 です。Oracle Application Server 環境では、通常、7777 です。

たとえば、スタンドアロン OC4J 環境では、適切なホストの 8888 ポートを使用してコンソールにアクセスできます。

```
http://host:1888
```

このインタフェースの使用方法の詳細は、Application Server Control コンソールのオンライン・ヘルプを参照してください。

コンソールは、アプリケーション、管理、パフォーマンスおよび Web サービスの機能エリアに編成されています。Web モジュールは、アプリケーション・エリアを使用して管理できます。

Web モジュールの構成は、Application Server Control コンソールの該当する「Web モジュール」ホームページから開始します。このページには、「一般」タブ、「パフォーマンス」タブおよび「管理」タブがあります。

この項の後半では、次の項目について説明します。

- 「Web モジュール」ホームページへのアクセス方法
- トップレベルの「Web モジュール」ページのサマリー

「Web モジュール」ホームページへのアクセス方法

OC4J ホームページから「Web モジュール」ホームページにアクセスするには、「アプリケーション」タブを選択します。このタブから、次のいずれかの方法で、「Web モジュール」ホームページにアクセスすることができます。

1. 「表示」ドロップダウン・メニューの「アプリケーション」を選択して、アプリケーションを表示します。
2. 目的のアプリケーションを選択します。
3. 「アプリケーション」ホームページが表示されたら、目的のモジュールを選択します。

または

1. 「表示」ドロップダウン・メニューの「モジュール」を選択して、すべてのアプリケーションのモジュールを表示します。
2. 目的のモジュールを選択します。

トップレベルの「Web モジュール」ページのサマリー

「Web モジュール」ホームページから、次のタブにアクセスできます。

- 「一般」タブ（「Web モジュール」ホームページ自体）: Web モジュールのホスト、ポートおよびコンテキスト・パスと、モジュールのアクティブ・サーブレットおよび JSP ページを一覧表示します。各サーブレットおよび JSP ページについては、「一般」タブ・ページに次のメトリックが一覧表示されます。
  - アクティブ・リクエスト
  - 現行クライアント処理時間（5 分前からの各リクエストの平均処理時間）
  - リクエスト数 / 秒
  - 処理済リクエスト
  - 合計クライアント処理時間（サーブレットまたは JSP ページの起動時以降）

  （OC4J が再起動されると、これらのメトリックは維持されません。）
- 「パフォーマンス」タブ : 実行されたサーブレットおよび JSP ページの次のメトリックを、グラフィカルに表示します。
  - アクティブ・セッション : 指定された期間にわたる各時点でアクティブなセッションの数
  - アクティブ・リクエスト : 指定された期間にわたる各時点でアクティブなリクエストの数
  - レスポンスと負荷 : 指定された期間にわたる、5 分前からの平均リクエスト処理時間と、5 分前からのリクエスト数（1 秒当たり）
- 「管理」タブ : このページから、特定の Web モジュール構成プロパティの表示または編集、`web.xml` または `orion-web.xml` ファイルの表示、Web モジュールに関連する様々な種類のマッピングの表示または編集が可能です。具体的には、次のページにアクセスできます。
  - 「構成プロパティ」ページ
  - 「デプロイメント・ディスクリプタの表示」ページ
  - 「固有デプロイメント・ディスクリプタの表示」ページ
  - 「サーブレット・マッピング」ページ
  - 「フィルタ・マッピング」ページ
  - 「リソース参照マッピング」ページ
  - 「EJB 参照マッピング」ページ
  - 「環境エントリ・マッピング」ページ

  これらのページについては、次項の「Application Server Control Web モジュールの構成ページ」で説明します。

前述のページの詳細は、Application Server Control のオンライン・ヘルプで、状況依存トピックの「「Web モジュール」ホームページ」、「Web モジュールの「パフォーマンス」ページ」および「Web モジュールの「管理」ページ」を参照してください。OC4J パフォーマンス・メトリックの詳細は、オンライン・ヘルプの「OC4J パフォーマンス・メトリックのサマリー」を参照してください。

Application Server Control Web モジュールの構成ページ

Web モジュールの「管理」ページから、「タスクに移動」を選択して、Web モジュールの構成に関連するいくつかの表示および編集機能を使用することができます。これらについて、次の項で説明します。

- 「構成プロパティ」ページ
- デプロイメント・ディスクリプタの表示ページ
- 「サーブレット・マッピング」ページ
- 「フィルタ・マッピング」ページ
- 「リソース参照マッピング」ページ
- 「EJB 参照マッピング」ページ
- 「環境エントリ・マッピング」ページ
- 「リソース参照の参照コンテキスト」ページ

注意：

- これらのページで表示または編集できる `orion-web.xml` プロパティについては、対応する要素および属性を B-5 ページの「orion-web.xml および global-web-application.xml の要素と属性」で説明します。
- プロパティが `<web-app>` 要素にあると記述されている場合、既存の値は、`web.xml` ファイルまたは `orion-web.xml` ファイルのいずれかから取得されます。`<web-app>` 要素は `web.xml` のルート要素であり、スキーマはサーブレット仕様で定義されます。この定義も、`orion-web.xml` スキーマ定義にインポートされます。`orion-web.xml` の `<web-app>` 要素で設定することにより、`web.xml` 内の `<web-app>` 要素のすべての同じ設定を効率的にオーバーライドすることができます。Application Server Control コンソールを使用してこれらのプロパティに加えられた変更は、`orion-web.xml` ファイルで維持されます。

「構成プロパティ」ページ

表 A-1 には、Application Server Control コンソールの「構成プロパティ」ページを使用して構成できる Web モジュール・プロパティが示されています。このページは、Web モジュールの「管理」ページからアクセスできます。プロパティに関する追加情報は、B-5 ページの「orion-web.xml および global-web-application.xml の要素と属性」に示されている該当する要素を参照してください。

「構成プロパティ」ページには、Web モジュール名とアプリケーション名も表示されます。

Application Server Control のオンライン・ヘルプで、状況依存トピックの「Web モジュールの「構成プロパティ」ページ」も参照してください。

表 A-1「構成プロパティ」ページのプロパティ

Application Server Control のプロパティ	対応する XML エントリ	説明
表示名	`<web-app>` の `<display-name>` 要素	ツールに表示される Web モジュールの短縮名です。 **注意 :** これは Application Server Control による読取り専用です。
説明	`<web-app>` の `<description>` 要素	Web モジュールの説明（オプション）です。 **注意 :** これは Application Server Control による読取り専用です。
分散可能	`<web-app>` の `<distributable>` 要素	サーブレット仕様に規定されているように、アプリケーションが分散可能かどうかを示します。このプロパティは、このページで編集できません。 **注意 :** これは Application Server Control による読取り専用です。
クラスパス	`<orion-web-app>` の `<classpath>` サブ要素	Web アプリケーションのクラスの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。
永続性パス	`<orion-web-app>` の `persistence-path` 属性	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。値が指定されていない場合、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。 **注意 :** OC4J クラスタリングが有効になっている場合、この属性は無視されます。

表 A-1「構成プロパティ」ページのプロパティ（続き）

Application Server Control のプロパティ	対応する XML エントリ	説明
一時ディレクトリ	`<orion-web-app>` の `temporary-directory` 属性	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。
ファイル変更チェック間隔	`<orion-web-app>` の `file-modification-check-interval` 属性	静的ファイル（HTML ファイルなど）について、タイムスタンプが変化しているかどうか、また、そのためにファイル・システムからリロードする必要があるかどうかをチェックするタイミングを、ミリ秒単位で決定します。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。
セッション・タイムアウト	`<web-app>` の `<session-config>` 要素の `<session-timeout>` サブ要素	Web アプリケーションで作成されるすべてのセッションのデフォルト・セッション・タイムアウトを定義します。値が 0 以下の場合、タイムアウトは発生しません。
デフォルト・バッファ・サイズ	`<orion-web-app>` の `default-buffer-size` 属性	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。指定のない場合、デフォルトは、`2048` です。
ディレクトリ参照を許可	`<orion-web-app>` の `directory-browsing` 属性	「/」で終わる URL のディレクトリ参照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。ディレクトリ参照に関する追加情報は、B-18 ページの「<orion-web-app>」を参照してください。

デプロイメント・ディスクリプタの表示ページ

web.xml ファイルと orion-web.xml ファイルは、それぞれ、Application Server Control コンソールの「デプロイメント・ディスクリプタの表示」ページと「固有デプロイメント・ディスクリプタの表示」ページを使用して表示できますが、編集はできません。

「サーブレット・マッピング」ページ

Application Server Control コンソールの「サーブレット・マッピング」ページでは、サーブレット名と URL パターンの間のマッピングを表示できますが、編集はできません。表 A-2 で、これらのマッピングについて説明します。

Application Server Control のオンライン・ヘルプで、状況依存トピックの「Web モジュールの「サーブレット・マッピング」ページ」も参照してください。

表 A-2「サーブレット・マッピング」ページのプロパティ

Application Server Control のプロパティ	対応する XML エントリ	説明
サーブレット名	<web-app> の <servlet-mapping> 要素の <servlet-name> サブ要素	<web-app> の対応する <servlet> 要素の <servlet-name> サブ要素で定義される、サーブレットの希望名への参照です。
URL パターン	<web-app> の <servlet-mapping> 要素の <url-pattern> サブ要素	対応するサーブレット名にマップするために必要な、サーブレットの URL パターン（サーブレット・パス）です。サーブレットを起動する URL には、このパターンが含まれます。

「フィルタ・マッピング」ページ

表 A-3 には、「フィルタ・マッピング」ページを使用してサーブレット・フィルタについて構成できるプロパティが示されています。

Application Server Control のオンライン・ヘルプで、状況依存トピックの「Web モジュールの「フィルタ・マッピング」ページ」も参照してください。

表 A-3「フィルタ・マッピング」ページの構成可能なプロパティ

Application Server Control のプロパティ	対応する XML エントリ	説明
フィルタ名	<web-app> の <filter-mapping> 要素の <filter-name> サブ要素	<web-app> の対応する <filter> 要素の <filter-name> サブ要素で定義される、フィルタの希望名への参照です。
URL パターンまたはサーブレット名	<web-app> の <filter-mapping> 要素の <url-pattern> または <servlet-name> サブ要素	URL パターンまたはサーブレット名の（両方ではなく）いずれかを対応するフィルタ名にマップします。
フォワードに適用	<web-app> の <filter-mapping> 要素の <dispatcher> サブ要素と、FORWARD という値	この値は、サーブレット名または URL パターンに合致する転送ターゲットのすべてに適用されるフィルタに対して使用します。

表 A-3「フィルタ・マッピング」ページの構成可能なプロパティ（続き）

Application Server Control のプロパティ	対応する XML エントリ	説明
リクエストに適用	`<web-app>` の `<filter-mapping>` 要素の `<dispatcher>` サブ要素と、`REQUEST` という値	この値は、サーブレット名または URL パターンに合致する直接リクエスト・ターゲットにも適用されるフィルタに対して、「フォワードに適用」または「インクルードに適用」設定に加えて使用します。
インクルードに適用	`<web-app>` の `<filter-mapping>` 要素の `<dispatcher>` サブ要素と、`INCLUDE` という値	この値は、サーブレット名または URL パターンに合致するインクルード・ターゲットのすべてに適用されるフィルタに対して使用します。
エラーに適用	`<web-app>` の `<filter-mapping>` 要素の `<dispatcher>` サブ要素と、`ERROR` という値	この値は、エラー・ページ・メカニズムで適用されるフィルタに対して使用します。

表 A-4 には、「フィルタ・マッピング」ページで参照用に表示される追加のサーブレット・フィルタ・プロパティが示されています。

表 A-4「フィルタ・マッピング」ページの参照プロパティ

Application Server Control のプロパティ	対応する XML エントリ	説明
名前	`<web-app>` の `<filter>` 要素の `<filter-name>` サブ要素	フィルタの希望名です。
クラス	`<web-app>` の `<filter>` 要素の `<filter-class>` サブ要素	フィルタ・コードを含むクラスの完全修飾名です。
説明	`<web-app>` の `<filter>` 要素の `<description>` サブ要素	フィルタの説明（オプション）です。

注意： 関連情報については、第 4 章「サーブレット・フィルタの理解および使用方法」を参照してください。

「リソース参照マッピング」ページ

「リソース参照マッピング」ページは、リソース（データソース、JMS キュー、メール・セッションなど）の JNDI の場所を指定するために使用します。表 A-5 に、このページで構成できるプロパティを示します。

Application Server Control のオンライン・ヘルプで、状況依存トピックの「Web モジュールのリソース参照マッピング」も参照してください。

「リソース参照マッピング」ページからは、「リソース参照の参照コンテキスト」ページ（A-11 ページの「「リソース参照の参照コンテキスト」ページ」を参照）に移動して、JNDI の参照コンテキストを編集することもできます。

表 A-5「リソース参照マッピング」ページのプロパティ

Application Server Control のプロパティ	対応する XML エントリ	説明
名前	<orion-web-app> の <resource-ref-mapping> サブ要素の name 属性	<web-app> の <resource-ref> の <res-ref-name> サブ要素の値を参照するリソースの名前です。より具体的には、リソース・マネージャのコネクション・ファクトリの参照の名前です。
タイプ	<web-app> の <resource-ref> の <res-type> サブ要素	データソースまたはその他のリソースのタイプです。これは、リソースによって実装される完全修飾 Java タイプです。
認可	<web-app> の <resource-ref> の <res-auth> サブ要素	リソース・マネージャへのサインオンがアプリケーション・コンポーネントでプログラム化されている（認可値は Application）か OC4J コンテナによって管理されている（認可値は Container）かを示します。
JNDI ロケーション	<orion-web-app> の <resource-ref-mapping> 要素の location 属性	リソースのルックアップを行う希望の JNDI の場所です。
参照コンテキスト	<orion-web-app> の <resource-ref-mapping> 要素の <lookup-context> サブ要素の location 属性	リソースのルックアップでデフォルトのコンテキストのかわりに使用されるオプションの JNDI コンテキストを指定します。

「EJB 参照マッピング」ページ

「EJB 参照マッピング」ページは、EJB の JNDI の場所を指定するために使用します。表 A-6 に、このページのプロパティを示します。

Application Server Control のオンライン・ヘルプで、状況依存トピックの「Web モジュールの「EJB 参照マッピング」ページ」も参照してください。

表 A-6「EJB 参照マッピング」ページのプロパティ

Application Server Control のプロパティ	対応する XML エントリ	説明
名前	<orion-web-app> の <ejb-ref-mapping> サブ要素の name 属性	<web-app> の <ejb-ref> の <ejb-ref-name> サブ要素の値を参照する EJB の名前です。
タイプ	<web-app> の <ejb-ref> 要素の <ejb-ref-type> サブ要素	EJB のタイプ（Entity または Session）です。
ホーム・インタフェース	<web-app> の <ejb-ref> 要素の <home> サブ要素	EJB のホーム・インタフェースの完全修飾名です。
リモート・インタフェース	<web-app> の <ejb-ref> 要素の <remote> サブ要素	EJB のリモート・インタフェースの完全修飾名です。
JNDI ロケーション	<orion-web-app> の <ejb-ref-mapping> サブ要素の location 属性	EJB のルックアップを行う希望の JNDI の場所です。

注意：

- 関連情報については、8-10 ページの「Enterprise JavaBeans の概要」を参照してください。

「環境エントリ・マッピング」ページ

「環境エントリ・マッピング」ページは、環境エントリの新しい値を設定するために使用します。表 A-7 に、このページのプロパティを示します。

Application Server Control のオンライン・ヘルプで、状況依存トピックの「Web モジュールの「環境エントリ・マッピング」ページ」も参照してください。

表 A-7「環境エントリ・マッピング」ページのプロパティ

Application Server Control のプロパティ	対応する XML エントリ	説明
名前	<orion-web-app> の <env-entry-mapping> サブ要素の name 属性	環境エントリの名前です。<web-app> の <env-entry> 要素の <env-entry-name> サブ要素の値を参照します。
タイプ	<web-app> の <env-entry> 要素の <env-entry-type> サブ要素	環境エントリの Java タイプです。
説明	<web-app> の <env-entry> 要素の <description> サブ要素	環境エントリの説明（オプション）です。
値	<web-app> の <env-entry> 要素の <env-entry-value> サブ要素	環境エントリのアセンブルされた値（通常、web.xml ファイルから）です。

表 A-7「環境エントリ・マッピング」ページのプロパティ（続き）

Application Server Control のプロパティ	対応する XML エントリ	説明
デプロイ済の値	<orion-web-app> の <env-entry-mapping> サブ要素	アセンブルされた値をオーバーライドするための、環境エントリの希望のデプロイ値です。<env-entry-mapping> 要素の値は、<env-entry-value> 要素の値をオーバーライドします。

「リソース参照の参照コンテキスト」ページ

このページは、「リソース参照マッピング」ページ（A-9 ページの「「リソース参照マッピング」ページ」を参照）からリンクされており、リソースのルックアップを行う新しいコンテキストを指定することができます。コンテキスト属性を編集することもできます。

Application Server Control のオンライン・ヘルプで、状況依存トピックの「「リソース参照の参照コンテキスト」ページ」も参照してください。

表 A-8「リソース参照の参照コンテキスト」ページのプロパティ

Application Server Control のプロパティ	対応する XML エントリ	説明
リソース参照名	<orion-web-app> の <resource-ref-mapping> サブ要素の name 属性	<web-app> の <resource-ref> の <res-ref-name> サブ要素の値を参照するリソースの名前です。より具体的には、リソース・マネージャのコネクション・ファクトリの参照の名前です。
参照コンテキストの場所	<orion-web-app> の <resource-ref-mapping> 要素の <lookup-context> サブ要素の location 属性	リソースのルックアップでデフォルトのコンテキストのかわりに使用されるオプションの JNDI コンテキストを指定します。
名前（コンテキスト属性の）	<lookup-context> の <context-attribute> サブ要素の name 属性	対応する参照コンテキストとして指定されているデフォルト以外（サード・パーティなど）の JNDI コンテキストに送信するための属性の名前です。
値（コンテキスト属性の）	<lookup-context> の <context-attribute> サブ要素の value 属性	属性の希望の値です。

Web モジュールの MBean および管理のサマリー

標準準拠の MBean は、OC4J のランタイム構成で使用できます。次の項で、概要を説明します。

- OC4J の MBean 管理の一般的な概要
- OC4J の Web モジュールの MBeans のサマリー

OC4J の MBean 管理の一般的な概要

OC4J では JMX 仕様がサポートされているので、J2EE 環境でリソース（リソース・アダプタ関連のリソースなど）を動的に管理するための標準インタフェースを作成することができます。OC4J の JMX 実装は、JMX クライアントのシステム MBean ブラウザを提供します。これにより、OC4J に付属の MBean を使用して OC4J インスタンスを管理することができます。

MBean は、JMX の管理可能なリソースを表す Java オブジェクトです。OC4J 内の各管理可能リソース（アプリケーション、リソース・アダプタなど）は、適切な MBean のインスタンスによって管理されます。OC4J に付属の各 MBean は、Application Server Control コンソールのシステム MBean ブラウザからアクセス可能な管理インタフェースを公開します。MBean 属性の設定、MBean でメソッドをコールする操作の実行、エラーまたは特定のイベントの通知のサブスクライブ、および実行統計の表示を行うことができます。

注意： この情報は参照用に提供されますが、Web モジュールの主要な構成設定は Application Server Control コンソールの他の機能によってユーザーによりわかりやすい方法で公開されます。A-4 ページの「Application Server Control Web モジュールの構成ページ」を参照してください。

OC4J ホームページからブラウザにアクセスするには、「管理」タブを選択し、タスクのリストから JMX タスクの「システム MBean ブラウザ」を選択します。ブラウザでは、次のことを実行できます。

- 左側のフレームで、目的の MBean を選択します。
- 右側のフレームにある「属性」タブを使用して、属性を表示または変更します。設定可能な属性にはフィールドがあり、新しい値を入力できます。その後、変更を適用します。
- 右側のフレームにある「操作」タブを使用して、MBean でメソッドを起動します。目的の操作を選択します。操作ウィンドウでは、パラメータ設定を指定して起動することができます。
- 右側のフレームにある「通知」タブ（使用できる場合）を使用して、通知をサブスクライブします。通知する各項目を選択することができます。選択したら、変更を適用します。
- 右側のフレームにある「統計」タブ（使用できる場合）を使用して、実行統計を表示します。

MBean およびその属性は場合によって変更が有効になる時点が異なることに注意してください。ランタイム・モデルでは、変更はただちに有効になります。構成モデルでは、一部の変更はリソースが再起動されると有効になり、一部の変更はアプリケーションが再起動されると有効になり、OC4J が再起動されたときに有効になるものもあります。変更が維持されるかどうかも、場合によって異なります。

詳細は、『Oracle Containers for J2EE 構成および管理ガイド』を参照してください。システム MBean ブラウザ自体も、MBean に関する情報を提供します。

OC4J の Web モジュールの MBeans のサマリー

OC4J は、アプリケーション実行時の管理を支援するために、各 Web モジュール（OC4J のデフォルト Web アプリケーションなど）の MBean セットをエクスポートします。J2EE 管理仕様をサポートするにはいくつかの OC4J MBean が必要ですが、それらによって拡張機能が提供される場合もあります。その他の OC4J MBean は、そのモデルへの Oracle の拡張機能です。

表 A-9 に、Web モジュールに関連し、JSR-77 によってアプリケーション・サーバーに必要とされる MBean の OC4J 実装をまとめます。これらの実装は、`oracle.oc4j.admin.management.mbeans` パッケージに含まれています。

注意 :

- MBean は、システム MBean ブラウザで自動文書化され、MBean 属性、操作および通知（該当する場合）に関するいくつかのドキュメントが提供されます。
- ほとんど（すべてではない）の MBean の統計プロパティは、DMS 統計から得られます。DMS の概要は、9-6 ページの「Oracle Application Server のダイナミック・モニタリング・サービス」を参照してください。
- デフォルトの Web アプリケーションについて : OC4J は、デフォルトの J2EE 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）を含むデフォルト構成とともにインストールされます。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（Application Server Control コンソールを除く）のデフォルトの親です。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およびルート・ディレクトリ・パスは、OC4J のグローバル `application.xml` ファイルで指定されます。スタンドアロン OC4J では、デフォルトの Web アプリケーションは、`default-web-site.xml` ファイルによって Web サイトにバインドされ、デフォルトのコンテキスト・パスは「/」です。

表 A-9 Web モジュールの必須システム MBean

MBean	説明
Servlet	`web.xml` ファイルの `<servlet>` 要素に対応するプロパティによって、サーブレットのインスタンスを管理します。
WebModule	`web.xml` ファイルの `<web-app>` 要素（`<servlet>` サブ要素の外）に対応するプロパティによって、Web モジュールの標準機能を管理します。 WebModule Mbean の loadAllServletMBeans() メソッドは、基礎となるサーブレットをロードせずにサーブレット Mbeans を作成します。OC4J は、起動時または事前ロードがリクエストされた場合に、サーブレットをロードします。詳細は 2-16 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。

表 A-10 に、Web モジュールに関連し、Oracle の拡張機能でもある OC4J MBean をまとめます。この MBean 実装も、oracle.oc4j.admin.management.mbeans パッケージに含まれています。

表 A-10 Web モジュールの追加システム MBean

MBean	説明
OC4JWebModule	orion-web.xml ファイルの <orion-web-app> 要素（<web-app> サブ要素の外）に対応するプロパティによって、Web モジュールの OC4J 固有の機能を管理します。

OC4J の Web モジュールの MBeans に関する注意

- <web-app> サブ要素の <context-param>、<servlet-mapping>、<filter-mapping> および <session-timeout> に対応する WebModule 属性には、Application Server Control コンソールからアクセスできます。
- <orion-web-app> 属性およびサブ要素の file-modification-check-interval、directory-browsing、<classpath>、<resource-ref-mapping>、<env-entry-mapping>、<ejb-ref-mapping> に対応する OC4JWebModule 属性には、Application Server Control コンソールからアクセスできます。
- JSP 関連の <orion-web-app> 属性の jsp-cache-directory、jsp-cache-tlds、jsp-taglib-locations、jsp-print-null、jsp-timeout は、本質的にグローバルであり、OC4JWebModule には適用されません。

B

Web モジュールの構成ファイル

この付録では、OC4J 固有の Web モジュール構成ファイルの `global-web-application.xml`（グローバルおよびデフォルト構成用）および `orion-web.xml`（アプリケーション・レベル構成用）に関する参照情報を提供します。各ファイルの概要と、それらの標準 `web.xml` ファイルとの関係についても説明します。

- Web アプリケーションの構成ファイルの概要
- orion-web.xml および global-web-application.xml の階層
- orion-web.xml および global-web-application.xml の要素と属性

Web アプリケーションの構成ファイルの概要

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、独立した 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 web.xml 構成ファイル
- Oracle の global-web-application.xml 構成ファイル
- Oracle の orion-web.xml 構成ファイル
- Web アプリケーションの構成ファイル間の関係のサマリー

標準 web.xml 構成ファイル

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 XSD 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。

次に、サンプル `web.xml` 構成の中の、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示します。

```
<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>
```

Oracle の global-web-application.xml 構成ファイル

OC4J の 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、XSD の orion-web.xsd で定義されています。この XSD は、次項の「Oracle の orion-web.xml 構成ファイル」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 XSD と同一です。

orion-web XSD は、web.xml 用の標準 XSD のスーパーセットです。orion-web XSD 内のトップレベル要素 <orion-web-app> のサブ要素 <web-app> は、web.xml のトップレベル要素 <web-app> と同じ仕様です。<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（可能な場合）。<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は、B-5 ページの「orion-web.xml および global-web-application.xml の要素と属性」を参照してください。

Oracle の orion-web.xml 構成ファイル

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、対応する XSD で定義されています。この XSD は、前項の「Oracle の global-web-application.xml 構成ファイル」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 XSD 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。

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は、B-5 ページの「orion-web.xml および global-web-application.xml の要素と属性」を参照してください。

Web アプリケーションの構成ファイル間の関係のサマリー

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。

orion-web.xml および global-web-application.xml の階層

次に、global-web-application.xml ファイルと orion-web.xml ファイルの要素の階層の概要を示します。

```
<orion-web-app>
   <classpath>
   <context-param-mapping>
   <mime-mappings>
   <virtual-directory>
   <access-mask>
      <host-access>
      <ip-access>
   <servlet-chaining>
   <request-tracker>
   <session-tracking>
      <session-tracker>
   <resource-ref-mapping>
      <lookup-context>
         <context-attribute>
   <resource-env-ref-mapping>
   <security-role-mapping>
      <group>
      <user>
   <env-entry-mapping>
   <ejb-ref-mapping>
   <service-ref-mapping>
   <expiration-setting>
   <web-app>
   <jazn-web-app>
   <web-app-class-loader>
   <ojsp-init>
```

注意： global-web-application.xml の <ojsp-init> 要素は使用できません。

---

orion-web.xml および global-web-application.xml の要素と属性

この項では、orion-web.xml および global-web-application.xml ファイルの要素をアルファベット順に説明します。階層については、前項の「orion-web.xml および global-web-application.xml の階層」を参照してください。

この項の要素の説明は、一般に、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。サマリーは、B-4 ページの「Web アプリケーションの構成ファイル間の関係のサマリー」を参照してください。

注意 :

- 属性について説明している箇所では、属性値が常に引用符付きで設定されていることに注意してください（attribute="value"）。
- 関係する属性のほとんどは、Application Server Control デプロイ・プラン・エディタによって設定できます。詳細は、『Oracle Containers for J2EE デプロイメント・ガイド』を参照してください。

<access-mask>

親要素 : <orion-web-app>

子要素 : <host-access>、<ip-access>

必須 / オプション オプション、0 または 1 つ

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リングするには、<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。

表 B-1 <access-mask> 属性

名前	説明
default	値 : allow | deny デフォルト : allow <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。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。

<classpath>

親要素： <orion-web-app>

子要素： なし

必須 / オプション オプション、0 以上

この要素を使用して、Web アプリケーションのクラスの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。

表 B-2 <classpath> 属性

名前	説明

path	値 : 文字列
	
	デフォルト : n/a（必要）
	
	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-attribute>

親要素： <lookup-context>

子要素： なし

必須 / オプション `<lookup-context>` を使用する場合は必須、1 つ以上

この要素の各出現は、親の `<lookup-context>` 要素で名前が付けられているデフォルト以外（サード・パーティなど）の JNDI コンテキストに送信するために属性を指定します。

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。

表 B-3 <context-attribute> 属性

名前	説明
name	値 : 文字列 デフォルト : n/a（必要） 属性の名前を指定します。
value	値 : 文字列 デフォルト : n/a（必要） 属性の希望の値を指定します。

<context-param-mapping>

親要素： <orion-web-app>

子要素： なし

必須 / オプション オプション、0 以上

この要素は、次のように、情報を要素自体のコンテンツに挿入します。

`orion-web.xml` で、`<context-param-mapping>` 要素は、サーブレット・コンテキスト・パラメータについて、`web.xml` の対応する `<context-param>` 要素によって指定されている値をオーバーライドします。`name` 属性を使用して `web.xml` の `<param-name>` 設定に適合させ、要素の値を使用して新しい値を指定します。

```
<context-param-mapping name="..." >deploymentValue</context-param-mapping>
```

表 B-4 <context-param-mapping> 属性

名前	説明
name	値 : 文字列 デフォルト : n/a（必要） 新しい値を指定するパラメータの名前です。

<ejb-ref-mapping>

親要素： <orion-web-app>

子要素： なし

必須 / オプション　オプション、0 以上

この要素を使用して、EJB の JNDI の場所を宣言します。これは、web.xml ファイルで EJB を宣言するために、対応する <ejb-ref> または <ejb-local-ref> 要素とともに使用します。<ejb-ref-mapping> 要素の name 属性は、web.xml の <ejb-ref-name> 要素に対応し、location 属性は JNDI の場所を指定します。

表 B-5 <ejb-ref-mapping> 属性

名前	説明
name	値 : 文字列 デフォルト : n/a（必要） web.xml の <ejb-ref-name> から、EJB 参照名を指定します。
location	値 : 文字列 デフォルト : n/a（必要） EJB のルックアップを行う JNDI の場所を指定します。

<env-entry-mapping>

親要素： <orion-web-app>

子要素： なし

必須 / オプション　オプション、0 以上

orion-web.xml で、<env-entry-mapping> 要素は、環境エントリについて、web.xml の対応する <env-entry> 要素によって指定されている値をオーバーライドします。name 属性を使用して web.xml の <env-entry-name> 設定に適合させ、要素の値を使用して新しい値を指定します。

```
<env-entry-mapping name="..." >deploymentValue</env-entry-mapping>
```

表 B-6 <env-entry-mapping> 属性

名前	説明
name	値 : 文字列 デフォルト : n/a（必要） 新しい値を指定する環境エントリの名前です。

<expiration-setting>

親要素： <orion-web-app>

子要素： なし

必須 / オプション オプション、0 以上

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リロードします。）これは、ドキュメントほど頻繁にイメージのリ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。

表 B-7 <expiration-setting> 属性

名前	説明
expires	値 : 文字列（整数、秒） デフォルト : 0 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。デフォルト設定は、即時期限切れを要求します。
url-pattern	値 : 文字列 デフォルト : n/a（必要） 期限が設定される URL パターンを指定します。たとえば、次のようになります。 `url-pattern="*.gif"`

<group>

親要素： <security-role-mapping>

子要素： なし

必須 / オプション オプション、0 以上

`<security-role-mapping>` のサブ要素を使用して、親の `<security-role-mapping>` 要素で指定されているセキュリティ・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。

表 B-8 <group> 属性

名前	説明
name	値 : 文字列 デフォルト : n/a（必要） グループの名前を指定します。

<host-access>

親要素： <access-mask>

子要素： なし

必須 / オプション オプション、0 以上

`<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。

表 B-9 <host-access> 属性

名前	説明
domain	値 : 文字列 デフォルト : n/a（必要） ホストまたはドメインを指定します。
mode	値 : allow | deny デフォルト : n/a（必要） 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。

<ip-access>

親要素： <access-mask>

子要素： なし

必須 / オプション オプション、0 以上

`<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。

表 B-10 <ip-access> 属性

名前	説明
ip	値 : 文字列 デフォルト : n/a（必要） 32 ビット値の IP アドレスを指定します（例 : `123.124.125.126`）。
netmask	値 : 文字列 デフォルト : デフォルトなし 対応するサブネット・マスク（存在する場合）を指定します（例 : `255.255.255.0`）。
mode	値 : allow | deny デフォルト : n/a（必要） 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。

<jazn-web-app>

親要素： <orion-web-app>

子要素： なし

必須 / オプション オプション、0 または 1 つ

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。

Oracle Identity Management を Web アプリケーションのセキュリティ・プロバイダとして使用し、認証に SSO を使用している場合、`<jazn-web-app>` によってサーブレット・セッションを OracleAS JAAS Provider のユーザー・コンテキスト同期化できます。セッションをユーザー・コンテキストと同期化するには、`<jazn-web-app>` の `<property>` サブ要素で、`sso.session.synchronize` プロパティを `true`（デフォルト）に設定します。

```
<jazn-web-app ...>
<property name="sso.session.synchronize" value="true"/>
</jazn-web-app>
```

または、プロパティを false に設定することができます。

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 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 社の関連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`

表 B-11 <jazn-web-app> 属性

名前	説明
auth-method	値 : BASIC | SSO | COREIDSSO デフォルト : BASIC これは、HTTP クライアント認証のメソッドです。`BASIC` は、基本 J2EE 認証用です。`SSO` は、Oracle Single Sign-On 用です。`COREIDSSO` は、Oracle COREid Access and Identity をセキュリティ・プロバイダとして使用し、シングル・サインオンを使用する場合用です。 `auth-method` のもう 1 つの値である `DIGEST` は、OC4J 10.1.3 実装では使用されなくなりましたが、下位互換のためにサポートされています。かわりに、`web.xml` ファイルの `auth-method` の標準 `DIGEST` 設定を使用できます。 **注意 :** アプリケーションで、カスタムの `LoginModule` インスタンスを使用する場合は、`BASIC` を使用します。

表 B-11 <jazn-web-app> 属性（続き）

名前	説明
runas-mode	値 : ブール デフォルト : false このモードは、OC4J 10.1.3 実装では使用されなくなりましたが、下位互換のためにサポートされています。新しい統合 JAAS モードが、サーブレットの `runas-mode` および `doasprivileged-mode` と置換されます。 `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	値 : ブール デフォルト : true このモードは、OC4J 10.1.3 実装では使用されなくなりましたが、下位互換のためにサポートされています。新しい統合 JAAS モードが、サーブレットの `runas-mode` および `doasprivileged-mode` と置換されます。 `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。

run-as で機能しない servlet.init() に対して発行される警告

Web アプリケーションの場合、`web.xml` ファイル内で `run-as` *user* を指定すると、`Servlet.init()` メソッド以外のすべてのメソッドの起動は、指定されたユーザーとして起動します。JMS Router が OC4J のデフォルト・アプリケーションの場合、ルーターの EJB に対してコールが認可されている必要があります。これは、JAAS の `"oc4j-administrators"` ロールにマップされるアプリケーション・ロール `"jmsRouter"` を定義し、ルーターの EJB のすべてのメソッドに `<method-permission>` を指定して行います。

ルーターの Web モデル内のサーブレットの `init()` メソッドによって、ルーター EJB オブジェクトが作成されます。サーブレットに対して `web.xml` で `run-as` が指定されているかどうかにかかわらず、セキュリティ例外がスローされます。

```
@ oracle.oc4j.rmi.OracleRemoteException: anonymous is not allowed to call this EJB
method, check your security settings (method-permission in ejb-jar.xml and
security-role-mapping in orion-application.xml).
```

run-as 警告の回避策

セキュリティ警告を取り除くためには、次に示すように、ejb-jar.xml 内の <method-permission> の <method-name> 要素の * をコメントアウトし、jmsRouter ロールがアクセス可能な AdminMgrBean 内のすべてのメソッドを明示的に列挙します。

```
<!--
    <method-permission>
      <role-name>jmsRouter</role-name>
      <method>
         <ejb-name>AdminMgrBean</ejb-name>
         <method-name>*</method-name>
      </method>
    </method-permission>
-->
    <method-permission>
      <role-name>jmsRouter</role-name>
      <method>
         <ejb-name>AdminMgrBean</ejb-name>
         <method-name>getConfig</method-name>
      </method>
    </method-permission>
    ...
```

runAsRoleName が、正しく ServletDescriptor.java で解析され、HttpApplication.loadServlet() の info および thread に格納されます。

<lookup-context>

親要素： <resource-ref-mapping>

子要素： <context-attribute>

必須 / オプション　オプション、0 または 1 つ

この要素は、その location 属性により、親の <resource-ref-mapping> 要素でマップされるリソースのルックアップでデフォルトのコンテキストのかわりに使用されるオプションの JNDI コンテキストを指定します。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。）

表 B-12 <lookup-context> 属性

名前	説明
location	値 : 文字列 デフォルト : n/a（必要） デフォルト以外（サード・パーティなど）の JNDI コンテキストの名前を指定します。

<mime-mappings>

親要素： <orion-web-app>

子要素： なし

必須 / オプション　オプション、0 以上

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。

表 B-13 <mime-mappings> 属性

名前	説明
path	値 : 文字列 デフォルト : n/a（必要） 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。

<ojsp-init>

親要素： <orion-web-app>

子要素： なし

必須 / オプション　オプション、0 または 1 つ

この要素は、表 B-14 にリストで示す JSP 構成パラメータ（属性）を設定します。この要素を orion-web.xml ファイルで Web アプリケーションに対して指定する場合、デフォルト値を含むこれらの属性の値は、Web モジュールでインストールされた web.xml デプロイメント・ディスクリプタの <init-param> 要素に指定した対応するすべての JSP 構成パラメータの値をオーバーライドします。ojspc 事前変換ユーティリティの対応するすべてのコマンドライン・オプションは、<ojsp-init> 属性および web.xml 内の対応するすべての設定をオーバーライドします。

> **注意：** global-web-application.xml の <ojsp-init> 要素は使用できません。

表 B-14 <ojsp-init> 属性

名前	説明
debug-mode	値 : ブール デフォルト : false スタック・トレースを出力するかどうかを指定します。特定のランタイム例外が発生する場合にスタック・トレースを出力するには、true に設定します。 このパラメータが false で、ファイルが見つからない場合、見つからないファイルのフルパスは表示されません。存在しない JSP ファイルがリクエストされた場合に物理ファイル・パスの表示を抑止することは、重要なセキュリティ上の考慮事項です。

表 B-14 <ojsp-init> 属性（続き）

名前	説明
iso-8859-1-convert	値 : ブール デフォルト : true 文字列を iso-8859-1 キャラクタ・セットに変換するかどうかを指定します。iso-8859-1-convert の値が true の場合、エンコーダによって文字は完全に変換されます。iso-8859-1-convert の値が false の場合、バイト切捨てが発生し、パフォーマンスが向上します。文字列に文字が含まれない通常の使用状況の大部分に対して、このバイト切捨てをお薦めします。
jsr45-debug	値 : なし、クラス、ファイル デフォルト : なし JSR-045: Debugging Support for Other Languages に記述されている JSR-45 サポートのメソッドを指定します。jsr45-debug を "file" に設定すると、Java ラインにマップされた JSP ラインの SMAP を含む個別のファイルが生成されます。"class" 設定の場合、この情報はコンパイル済のクラス・ファイルに書き込まれます。jsr45-debug を "none" に設定すると、JSR-45 デバッグ・サポートがオフになります。
main-mode	値 : recompile、reload、justrun デフォルト : recompile JSP で生成されたクラスを自動的にリロードするかまたは JSP ページが変更された場合に自動的に変換するかを指定します。 この機能が有効な場合、新しいまたは変更された JSP ページが、Web アプリケーションを再デプロイまたは再起動の必要なく、OC4J ランタイムにロードされます。追加情報は、『Oracle Containers for J2EE JavaServer Pages 開発者ガイド』を参照してください。 main-mode の値が recompile の場合、コンテナは JSP ページのタイムスタンプをチェックして再変換し、最後にロードしてから変更されている場合はページをリロードします。reload で説明した機能も、同様に実行されます。 main-mode の値が reload の場合、コンテナは JSP トランスレータが生成したクラス（ページ実装クラスなど）のタイムスタンプをチェックし、最後にロードしてから変更または再デプロイされたクラスをリロードします。これは、JSP ページではなくコンパイル済のクラスを、開発環境から製品環境へデプロイまたは再デプロイする場合に、役立ちます。 main-mode の値が justrun の場合、コンテナはタイムスタンプに対して何も行わないため、JSP の再変換または JSP で生成された Java クラスのリロードは行われません。JSP ページは頻繁に変更されない製品環境で、このモードを使用すると効果的です。
precompile-check	値 : ブール デフォルト : false 標準 jsp_precompile-check 設定への HTTP リクエストをチェックするかどうか指定します。デフォルトは、false です。 precompile-check が true で jsp_precompile がリクエストによって有効になる場合、例外なく事前変換のみが JSP ページに実行されます。precompile-check を false に設定すると、パフォーマンスが向上しリクエスト内の jsp_precompile 設定はすべて無視されます。

表 B-14 <ojsp-init> 属性（続き）

名前	説明
reduce-tag-code	値 : ブール デフォルト : false `true` に設定すると、カスタム・タグ使用のために生成されたコードのサイズをさらに縮小するように指定します。

req-time-introspection	値 : ブール
	デフォルト : false
	`true` に設定すると、コンパイル時イントロスペクションが不可能な場合は常に、リクエスト時イントロスペクションを有効化します。ただし、コンパイル時イントロスペクションが可能で成功した場合、このフラグの設定にかかわらずリクエスト時イントロスペクションは行われません。
	リクエスト時イントロスペクションの使用例として、タグ・ハンドラが、タグ追加情報クラスの `VariableInfo` 内の汎用 `java.lang.Object` インスタンスを変換およびコンパイル中に返しても、実際は、より特定的なオブジェクトがランタイム中に生成されることがあげられます。この時、`request_time_introspection` が有効な場合、Web コンテナはリクエスト時間までイントロスペクションを遅延させます。
	加えて、このフラグの働きにより、`if..then..else` ループの別のブランチ内などで、Bean を 2 回宣言できます。次の例で説明します。`req_time_introspection` のデフォルト値 `false` の場合、このコードは解析例外を発生させます。`true` 値の場合、エラーを発生せずにコードが機能します。
	`<% if (cond) { %>`
	      `<jsp:useBean id="foo" class="pkgA.Foo1" />`
	`<% } else { %>`
	      `<jsp:useBean id="foo" class="pkgA.Foo2" />`
	`<% } %>`

static-text-in-chars	値 : ブール デフォルト : false `true` に設定すると、JSP トランスレータに指示を出して、バイトではなく文字の静的テキストを JSP ページに生成します。デフォルトは、`false` です。 次の例に示すように、ランタイム中に文字コードを動的に変更する能力が使用するアプリケーションに必要な場合、このフラグを有効化します。 `<% response.setContentType("text/html; charset=UTF-8"); %>` デフォルトの `false` に設定すると、静的テキスト・ブロックの出力のパフォーマンスが向上します。

表 B-14 <ojsp-init> 属性（続き）

名前	説明
tags-reuse	値 : compiletime、compiletime-with-release、none デフォルト : compiletime タグ・プーリングと呼ばれるタグ・ハンドラのモードを指定します。 ■ `compiletime` に設定すると、タグ・ハンドラのコンパイル時モデルの基本モードを有効化します。これがデフォルト値です。 ■ `compiletime_with_release` に設定すると、タグ・ハンドラのコンパイル時モデルをリリース・モードで再使用できるようになり、指定するページにおける指定するタグ・ハンドラ使用の間でタグ・ハンドラ `release()` メソッドがコールされます。 ■ `none` または `false` に設定すると、タグ・ハンドラの再使用は無効になります。JSP ページ・コンテキスト属性 `oracle.jsp.tags.reuse` を `true` 値に設定すると、どのような JSP ページでもこの値をオーバーライドできます。 ■ `runtime` に設定すると、タグ・ハンドラの再使用のランタイム・モデルが有効になります。JSP ページ・コンテキスト属性 `oracle.jsp.tags.reuse` を `false` 値に設定すると、どのような JSP ページでもこの値をオーバーライドできます。 `runtime` オプションおよび同じ機能を持つ `true` は、今回の OC4J リリースでは推奨しないことに注意してください。

表 B-15 に `<ojsp-init>` 属性および対応する JSP サーブレットの `<init-param>` 要素および `ojspc` コマンドライン・オプションを示します。JSP サーブレット `<init-param>` 要素の詳細は、『Oracle Containers for J2EE JavaServer Pages 開発者ガイド』を参照してください。JSP サーブレット `ojspc` コマンドライン・オプションの詳細は、『Oracle 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。

表 B-15 JSP サーブレット構成パラメータ

<ojsp-init> 属性	同等の JSP サーブレット <init-param>	ojspc コマンドライン・オプション
debug-mode	debug_mode	n/a
iso-8859-1-convert	iso-8859-1-convert	n/a
jsr45-debug	debug	-debug
main-mode	main_mode	n/a
precompile-check	precompile_check	n/a
reduce-tag-code	reduce_tag_code	-reduceTagCode
req-time-introspection	req_time_instrospection	-reqTimeIntrospection
static-text-in-chars	static-text-in-chars	-staticTextInChars
tags-reuse	tags_reuse_default	-tagReuse

<orion-web-app>

親要素： n/a（ルート）

子要素： <access-mask>、<classpath>、<context-param-mapping>、<ejb-ref-mapping>、<env-entry-mapping>、<expiration-setting>、<jazn-web-app>、<mime-mappings>、<ojsp-init>、<request-tracker>、<resource-env-ref-mapping>、<resource-ref-mapping>、<security-role-mapping>、<service-ref-mapping>、<servlet-chaining>、<session-tracking>、<virtual-directory>, <web-app>、<web-app-class-loader>

必須 / オプション　必須、1 つのみ

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： `always-redeploy`、`deployment-time` および `deployment-version` 属性は、直接使用がサポートされていません。そのため、これらの属性については（`deployment-time` および `deployment-version` はデプロイ時間および OC4J バージョンのコンテナ生成値を受け取ることがありますが）説明していません。これらのパラメータを変更しても無効です。

表 B-16 <orion-web-app> 属性

名前	説明
autojoin-session	値 : ブール デフォルト : true 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。
default-buffer-size	値 : 負でない整数（バイト） デフォルト : 2048 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。
default-charset	値 : 文字列 デフォルト : iso-8859-1 10.1.3.1 では、JSP ページおよびサーブレット・コンテナ、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。一般に、JSP 2.0 ユーザーには、かわりに、標準 `<page-encoding>` 機能（JSP 2.0 仕様に準拠し、`web.xml` の `<jsp-config>` 要素にある）を使用して URL パターンに基づくキャラクタ・セットを指定することをお薦めします。ただし、多数の JSP ページを（特に複数のアプリケーションにわたって）持つ場合は、EAR ファイルで多数の変更を加える必要をなくすために `default-charset` が役立つ場合があります。また、`default-charset` を使用してベース・デフォルトを設定してから、`<page-encoding>` 機能を使用して特定の URL パターンのデフォルトをオーバーライドすることもできます。`<jsp-config>` および `<page-encoding>` 要素の詳細は、JSP およびサーブレット仕様を参照してください。
default-mime-type	値 : 文字列 デフォルト : デフォルトなし `setContentType()` メソッドがサーブレット実装からコールされない状況に対して、サーブレット・レスポンスのデフォルト・コンテンツ・タイプを指定します。`default-mime-type` が指定されていない場合、デフォルト・コンテンツ・タイプは存在しません。

表 B-16 <orion-web-app> 属性（続き）

名前	説明
development	値 : ブール デフォルト : false 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リロードされます。 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。 OC4J JSP コンテナは development フラグに対して特別な処理を実行しないことに注意してください。関連するすべての機能は、OC4J サーブレット・コンテナによって処理されます。
directory-browsing	値 : allow | deny デフォルト : deny 「/」で終わる URL のディレクトリ参照の許可を指定します。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。
enable-jsp-dispatcher-shortcut	値 : ブール デフォルト : true true 設定では、特に、simple-jsp-mapping 属性も true に設定している場合、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。

表 B-16 <orion-web-app> 属性（続き）

名前	説明
file-modification-check-interval	値 : 文字列（整数、ミリ秒） デフォルト : 1000 静的ファイル（HTML ファイルなど）について、タイムスタンプが変化しているかどうか、また、そのためにファイル・システムからリロードする必要があるかどうかをチェックするタイミングを決定します。静的ファイルに初めてアクセスする場合は、ファイルがファイル・システムからロードされ、キャッシュされます。各後続アクセスについては、次のロジックが使用されます。 ■ 最後にファイル・タイムスタンプをチェックしてからの経過時間が、指定されているファイル・チェック間隔よりも短い場合は、タイムスタンプがチェックされず、ファイルがキャッシュからロードされます。 ■ 最後にファイル・タイムスタンプをチェックしてからの経過時間が、指定されているファイル・チェック間隔よりも長い場合は、タイムスタンプがチェックされます。最後にチェックした後にタイムスタンプが変更されている場合は、ファイルがファイル・システムからロードされます。変更されていなければ、キャッシュからロードされます。 この値は、ミリ秒単位で指定します。0（ゼロ）または負の数値を指定すると、ファイル・タイムスタンプが常にチェックされます。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-cache-directory	値 : 文字列 デフォルト : ./persistence（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） 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。
jsp-cache-tlds	値 : on | off | standard デフォルト : standard 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。standard または on に設定すると、キャッシュが有効になります。また、どちらの場合も、.tld ファイルは、<orion-web-app> の jsp-taglib-locations 属性に基づいてグローバル TLD 位置から継承されます。 また、standard に設定すると、アプリケーションの /WEB-INF ディレクトリで .tld ファイルが検索され、これらのファイルが、グローバル・レベルから継承されたファイルに追加されます。/WEB-INF/lib および /WEB-INF/classes サブディレクトリは検索されないことに注意してください。 また、on に設定すると、アプリケーション内のすべてのファイルから .tld ファイルが検索され、これらのファイルが、グローバル・レベルから継承されたファイルに追加されます。 off 設定では、永続 TLD キャッシュが無効になります。
jsp-print-null	値 : ブール デフォルト : true 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。

表 B-16 <orion-web-app> 属性（続き）

名前	説明
jsp-taglib-locations	値 : 文字列 デフォルト : 次を参照 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/` ■ *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。
jsp-timeout	値 : 負でない整数（秒） デフォルト : 0（タイムアウトなし） 期間を指定します。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。
persistence-path	値 : 文字列 デフォルト : デフォルトなし（デフォルトでは永続性なし） 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。値が指定されていない場合、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。 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。 **注意 :** OC4J クラスタリングが有効になっている場合、この属性は無視されます。
schema-major-version	値 : 文字列 デフォルト : デフォルトなし `orion-web.xml` XSD のメジャー・バージョン番号です。`orion-web.xml` を手動で作成する場合は、OC4J 10.1.3 実装で使用するために、この実装を 10 に設定してください。 **注意 :** この属性は、`orion-web.xml` の XSD には直接表示されません。この属性は、トップレベルの OC4J XSD の `attributeGroup` 仕様に基づきます。

表 B-16 <orion-web-app> 属性（続き）

名前	説明
schema-minor-version	値 : 文字列 デフォルト : デフォルトなし orion-web.xml XSD のマイナー・バージョン番号です。orion-web.xml を手動で作成する場合は、OC4J 10.1.3 実装で使用するために、この実装を 0 に設定してください。 **注意 :** この属性は、orion-web.xml の XSD には直接表示されません。この属性は、トップレベルの OC4J XSD の attributeGroup 仕様に基づきます。
servlet-webdir	値 : 文字列 デフォルト : /servlet/（次の「注意」を参照） OC4J のシステム・プロパティの http.webdir.enable が true に設定されている場合は、この属性を使用して、スタンドアロン OC4J でのクラス名によるサーブレットの起動を有効にします。システム・プロパティを設定すると、スラッシュ（/）で始まるすべての servlet-webdir 設定によってこの機能が有効になり、OC4J にクラス名でサーブレットを起動させるために、コンテキスト・パスの後に挿入される特別な URL 部分が指定されます。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、パッケージが含まれます。追加情報は、2-10 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。 この機能は、通常、開発またはテスト時に OC4J スタンドアロン環境で使用されます。重大なセキュリティ上のリスクが発生するので、本番環境では使用しないでください。（本番デプロイでは、標準の web.xml メカニズム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。コンテキスト・パスを「/」、設定を servlet-webdir="/servlet/" と仮定します。 http://www.example.com:8888/servlet/foo.SessionServlet servlet-webdir=""（空の引用符）に設定、または OC4J のシステム・プロパティを http.webdir.enable=false に設定すると、クラス名による起動が無効になります。 **注意 :** デフォルトを含むすべての servlet-webdir 設定は、http.webdir.enable のデフォルトの false 設定によってオーバーライドされます。OC4J のシステム・プロパティに関する一般的な情報は、『Oracle Containers for J2EE 構成および管理ガイド』を参照してください。
simple-jsp-mapping	値 : ブール デフォルト : false *.jsp が 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、true に設定します。アプリケーションに影響を与えるすべての Web ディスクリプタ（global-web-application.xml、web.xml、orion-web.xml）の <servlet> 要素で指定されます。true に設定すると、JSP ページのパフォーマンスが向上します。

表 B-16 <orion-web-app> 属性（続き）

名前	説明
source-directory	値 : 文字列 デフォルト : /WEB-INF/src（存在する場合）または /WEB-INF/classes `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。デフォルトの場所を使用する場合、OC4J は、デプロイ後にアプリケーションの `/WEB-INF` ディレクトリの場所を追跡し続けます。修正されたソース・ファイルは、パッケージ名に基づいて、`source-directory` ディレクトリの任意の場所で検出されることに注意してください。

temporary-directory	値 : 文字列
	
	デフォルト : ./temp
	
	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。
	
	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。

注意 :

`enable-jsp-dispatcher-shortcut`、`jsp-cache-directory`、`jsp-cache-tlds`、`jsp-print-null`、`jsp-taglib-locations`、`jsp-timeout` および `simple-jsp-mapping` 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 Containers for J2EE JavaServer Pages 開発者ガイド』を参照してください。

<request-tracker>

親要素： <orion-web-app>

子要素： なし

必須 / オプション オプション、0 以上

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、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。

表 B-17 <request-tracker> 属性

名前	説明
servlet-name	値 : 文字列 デフォルト : n/a（必要） 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、示されている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-env-ref-mapping>

親要素： <orion-web-app>

子要素： なし

必須 / オプション オプション、0 以上

この要素を使用して、環境リソースの JNDI の場所を宣言します。これは、リソースを宣言する `web.xml` ファイルの対応する `<resource-env-ref>` 要素とともに使用します。`<resource-env-ref-mapping>` 要素の `name` 属性は、`web.xml` の `<resource-env-ref-name>` 要素に対応し、`location` 属性は JNDI の場所を指定します。

表 B-18 <resource-env-ref-mapping> 属性

名前	説明
name	値 : 文字列 デフォルト : n/a（必要） `web.xml` から、リソース名を指定します。
location	値 : 文字列 デフォルト : n/a（必要） リソースのルックアップを行う JNDI の場所を指定します。

<resource-ref-mapping>

親要素： <orion-web-app>

子要素： <lookup-context>

必須 / オプション　オプション、0 以上

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の JNDI の場所を宣言します。これは、リソースを宣言する web.xml ファイルの対応する <resource-ref> 要素とともに使用します。<resource-ref-mapping> 要素の name 属性は、web.xml の <res-ref-name> 要素に対応し、location 属性は JNDI の場所を指定します。

次に、この要素とそのサブ要素を使用した例を示します。

```
<resource-ref-mapping location="jdbc/HyperSonicDS" name="jdbc/myDS">
  <lookup-context location="foreign/resource/location">
    <context-attribute name="java.naming.factory.initial" value="classname" />
    <context-attribute name="name" value="value" />
  </lookup-context>
</resource-ref-mapping>
```

表 B-19 <resource-ref-mapping> 属性

名前	説明
name	値 : 文字列 デフォルト : n/a（必要） web.xml から、リソース名を指定します。次に例を示します。 name="jdbc/TheDSVar"
location	値 : 文字列 デフォルト : n/a（必要） 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"

<security-role-mapping>

親要素： <orion-web-app>

子要素： <group>、<user>

必須 / オプション　オプション、0 以上

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 ファイルの <security-role> 要素によって指定されている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）のいずれかを使用します。

OC4J 構成ファイルの <security-role-mapping> 要素に関する追加情報は、『Oracle Containers for J2EE Enterprise JavaBeans 開発者ガイド』を参照してください。

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、system-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。

表 B-20 <security-role-mapping> 属性

名前	説明
impliesAll	値 : ブール デフォルト : false このマッピングを全ユーザーに適用するかどうかを指定します。
name	値 : 文字列 デフォルト : n/a（必要） 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する、セキュリティ・ロールの名前を指定します。

<service-ref-mapping>

親要素： <orion-web-app>

子要素： その独自のスキーマ定義に基づく

必須 / オプション オプション、0 以上

この要素は、Web サービスを宣言する web.xml ファイルに出現する <service-ref> 要素とともに使用します。この要素を使用して、対応する Web サービス（セキュリティ、ロギング、監査など）の OC4J 固有のサービス品質機能を指定することができます。詳細は、『Oracle Application Server Web Services 開発者ガイド』を参照してください。

<service-ref> 要素が web.xml、ejb-jar.xml または application-client.xml ファイルに出現することが可能であるように、対応する <service-ref-mapping> 要素も orion-web.xml、orion-ejb-jar.xml または orion-application-client.xml ファイルに出現することが可能です。<service-ref-mapping> 要素のサポートされる機能は、orion-web、orion-ejb-jar、および orion-application-client XSD にインポートされる独自の XSD に基づきます。

<servlet-chaining>

親要素： <orion-web-app>

子要素： なし

必須 / オプション オプション、0 以上

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リングまたは変換します。

重要： サーブレット・チェーンは、標準サーブレット・フィルタの機能と似た機能を持つ旧式の固有のメカニズムです。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。OC4J の <servlet-chaining> 要素は、現在のリリースでは使用されなくなっており、次のリリースではサポートされなくなります。

表 B-21 <servlet-chaining> 属性

名前	説明
mime-type	値 : 文字列 デフォルト : n/a（必要） チェーンをトリガーする MIME タイプ（text/html など）を指定します。
servlet-name	値 : 文字列 デフォルト : n/a（必要） 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。

<session-tracker>

親要素： <session-tracking>

子要素： なし

必須 / オプション オプション、0 以上

<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。

表 B-22 <session-tracker> 属性

名前	説明
servlet-name	値 : 文字列 デフォルト : n/a（必要） 起動するサーブレットを指定します。web.xml ファイルの関連 <servlet> 要素の対応する <servlet-name> または <servlet-class> 要素に基づいて、示されている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。
set-secure	値 : ブール デフォルト : false アプリケーションについて OC4J によって生成されるすべてのセッション Cookie が、HTTPS プロトコルが使用されている場合にのみクライアントによって返されるかどうかを指定します。set-secure="true" の場合、すべてのセッション Cookie に secure 属性が含まれます。この属性により、ブラウザはセキュアな HTTPS プロトコル経由でのみ Cookie を返します。set-secure="false" の場合、ブラウザは、任意のプロトコル経由で Cookie を返します。

<session-tracking>

親要素： <orion-web-app>

子要素： <session-tracker>

必須 / オプション　オプション、0 または 1 つ

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。セッション・トラッキングに使用するサーブレットは、`<session-tracker>` サブ要素により指定します。

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。

表 B-23 <session-tracking> 属性

名前	説明
cookies	値 : enabled | disabled デフォルト : enabled セッション Cookie を送信するかどうかを指定します。セッション Cookie の名前は、`JSESSIONID` です。（`JSESSIONID` Cookie の詳細は、3-4 ページの「OC4J がセッション・トラッキングに Cookie を使用する方法」を参照してください。）

表 B-23 <session-tracking> 属性（続き）

名前	説明
cookie-domain	値 : 文字列 デフォルト : デフォルトなし JSESSIONID セッション Cookie に必要なドメインを指定します。これは、該当する Set-Cookie HTTP レスポンス・ヘッダーのすべての domain 設定をオーバーライドします。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、同じセッション 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 スタンドアロン環境では、Web サイトの XML ファイルの <web-app> 要素が shared="true" に設定されているときに、この機能を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-path	値 : 文字列 デフォルト : デフォルトなし オプションで、JSESSIONID セッション Cookie に適用される URL パスの値を指定するために使用できます。この属性は、（Cookie ドメイン設定に基づき）任意の該当ドメイン内でセッション Cookie が有効になる URL のサブセットを指定します。指定すると、該当する Set-Cookie HTTP レスポンス・ヘッダーのすべての path 設定をオーバーライドします。指定しない場合、Set-Cookie ヘッダーに path 設定がないときは、デフォルトの Cookie パスは Web アプリケーションのコンテキスト・パスです。
cookie-max-age	値 : 負でない整数（秒） デフォルト : デフォルトなし この数値は、JSESSIONID セッション Cookie とともに送信され、ブラウザが Cookie を保存する最大期間（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。

<user>

親要素： <security-role-mapping>

子要素： なし

必須 / オプション オプション、0 以上

<security-role-mapping> のサブ要素を使用して、親の <security-role-mapping> 要素で指定されているセキュリティ・ロールにマップするユーザーを指定します。

表 B-24 <user> 属性

名前	説明
name	値 : 文字列 デフォルト : n/a（必要） ユーザーの名前を指定します。

<virtual-directory>

親要素： <orion-web-app>

子要素： なし

必須 / オプション オプション、0 以上

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。

表 B-25 <virtual-directory> 属性

名前	説明
real-path	値 : 文字列 デフォルト : n/a（必要） 実際のパス。UNIX の場合は `/usr/local/realpath`、Windows の場合は `C:¥testdir` など。
virtual-path	値 : 文字列 デフォルト : n/a（必要） 指定された実際のパスにマップする仮想パス。

<web-app>

親要素： <orion-web-app>

子要素： サーブレット仕様でのその独自のスキーマ定義に基づく

必須 / オプション オプション、0 または 1 つ

この要素は、標準的な web.xml ファイルなどで使用します。概要は B-2 ページの「標準 web.xml 構成ファイル」を、詳細はサーブレット仕様を参照してください。global-web-application.xml の <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。

表 B-26 <web-app> 属性

名前	説明
id	値 : ID デフォルト :
metadata-complete	値 : ブール デフォルト : true（サーブレット仕様 2.4）、false（サーブレット仕様 2.5） このデプロイメント・ディスクリプタおよびこのモジュールに関連するその他のデプロイメント・ディスクリプタ（たとえば Web サービス・ディスクリプタ）が完了しているかどうか、またはこのモジュールで使用可能でこのアプリケーションにパッケージされたクラス・ファイルを、デプロイ情報を指定する注釈について検証するかどうかを指定します。metadata-complete が true に設定されている場合、OC4J サーブレット・コンテナはアプリケーションのクラス・ファイル内にあるすべての注釈を無視します。metadata-complete が指定されていないか false に設定されていて、バージョンが 2.5 に設定されているか web-xml がサーブレット 2.5 のスキーマ・ネームスペースを指し示す場合、サーブレット・コンテナは注釈についてアプリケーションのクラス・ファイルを検証します。
version	値 : web-app-versionType デフォルト : n/a（必要）

<web-app-class-loader>

親要素： <orion-web-app>

子要素： なし

必須 / オプション オプション、0 または 1 つ

この要素は、クラスのロードの指示に使用します。

表 B-27 <web-app-class-loader> 属性

名前	説明
search-local-classes-first	値 : ブール デフォルト : false 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。デフォルトでは、最初にシステム・クラスが検索され、ロードされます。
include-war-manifest-class-path	値 : ブール デフォルト : true 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。設定しない場合は、WAR ファイルのマニフェストの CLASSPATH は含まれません。

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。
- Oracle の XML パーサーまたは JDBC ドライバのかわりにアプリケーションにパッケージされている XML パーサーまたは JDBC ドライバを使用する場合は、`search-local-classes-first` 属性を `true` に設定してください。また、`orion-application.xml` の `<remove-inherited>` タグで、デフォルトの継承済 Oracle ライブラリも指定する必要があります。手順の詳細は、『Oracle Containers for J2EE 開発者ガイド』を参照してください。

C

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。

ANTLR

このプログラムには、ANTLR からのサード・パーティ・コードが含まれています。ANTLR のライセンス条件に基づき、オラクル社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（ANTLR 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ないことに注意してください。

ANTLR ライセンス

```
Software License

We reserve no legal rights to the ANTLR--it is fully in the public domain. An
individual or company may do whatever they wish with source code distributed with ANTLR
or the code generated by ANTLR, including the incorporation of ANTLR, or its output,
into commerical software.
We encourage users to develop software with ANTLR. However, we do ask that credit is
given to us for developing ANTLR. By "credit", we mean that if you use ANTLR or
incorporate any source code into one of your programs (commercial product, research
project, or otherwise) that you acknowledge this fact somewhere in the documentation,
research report, etc... If you like ANTLR and have developed a nice tool with the
output, please mention that you developed it using ANTLR. In addition, we ask that the
headers remain intact in our source code. As long as these guidelines are kept, we
expect to continue enhancing this system and expect to make other tools available as
they are completed.
```

Apache

このプログラムには、Apache Software Foundation（Apache）からのサード・パーティ・コードが含まれています。Apache のライセンス条件に基づき、オラクル社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ないことに注意してください。

Apache のライセンス契約は、次の組込み Apache コンポーネントに適用されます。

- Apache HTTP Server
- Apache JServ
- mod_jserv
- 正規表現パッケージ、バージョン 1.3
- commons-el.jar にパッケージされた Apache 式言語
- mod_mm 1.1.3
- Apache XML Signature および Apache XML Encryption バージョン 1.4 for Java および 1.0 for C++
- log4j 1.1.1
- BCEL バージョン 5
- XML-RPC バージョン 1.1
- Batik バージョン 1.5.1
- ANT 1.6.2 および 1.6.5
- Crimson バージョン 1.1.3
- ant.jar
- wsif.jar
- bcel.jar

- soap.jar
- Jakarta CLI 1.0
- jakarta-regexp-1.3.jar
- JSP Standard Tag Library 1.0.6 および 1.1
- Struts 1.1
- Velocity 1.3
- svnClientAdapter
- commons-logging.jar
- commons-el.jar
- standard.jar
- jstl.jar

Apache ソフトウェア・ライセンス

Apache Web Server 1.3.29 のライセンス


```
/* ====================================================================
 * 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.
```

Apache Web Server 2.0 のライセンス

```
Copyright (c) 1999-2004, The Apache Software Foundation
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this
file except in compliance with the License.  You may obtain a copy of the License at
://www.apache.org/licenses/LICENSE-2.0
Unless required by applicable law or agreed to in writing, software distributed under
the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
KIND, either express or implied.  See the License for the specific language governing
permissions and limitations under the License.
Copyright (c) 1999-2004, The Apache Software Foundation
                                Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
      otherwise, or (ii) ownership of fifty percent (50%) or more of the
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making modifications,
      including but not limited to software source code, documentation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Source or
```

Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all

```
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.
```

Apache SOAP

このプログラムには、Apache Software Foundation（Apache）からのサード・パーティ・コードが含まれています。Apache のライセンス条件に基づき、オラクル社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ないことに注意してください。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オラクル社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オラクル社または Apache から提供されません。

Apache SOAP ライセンス

Apache SOAP ライセンス 2.3.1

```
Copyright (c) 1999 The Apache Software Foundation.  All rights reserved.
TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION
1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
      otherwise, or (ii) ownership of fifty percent (50%) or more of the
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making modifications,
      including but not limited to software source code, documentation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Source or
      Object form, made available under the License, as indicated by a
      copyright notice that is included in or attached to the work
```

(an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works

```
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.
```

```
9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

索引

記号

<access-mask> 要素（orion-web.xml），B-5
<classpath> 要素（orion-web.xml），B-6
<context-attribute> 要素（orion-web.xml），B-7
<context-param-mapping> 要素（orion-web.xml），B-7
<ejb-ref-mapping> 要素（orion-web.xml），B-8
<env-entry-mapping> 要素（orion-web.xml），B-8
<expiration-setting> 要素（orion-web.xml），B-9
<group> 属性（orion-web.xml），B-9
<host-access> 要素（orion-web.xml），B-10
<ip-access> 要素（orion-web.xml），B-10
<jazn-web-app> 要素（orion-web.xml），B-11
<lookup-context> 要素（orion-web.xml），B-13
<mime-mappings> 要素（orion-web.xml），B-14
<ojsp-init> 要素（orion-web.xml），B-14
<orion-web-app> 要素（orion-web.xml），B-18
<request-tracker> 要素（orion-web.xml），B-24
<resource-env-ref-mapping> 要素（orion-web.xml），B-24
<resource-ref-mapping> 要素（orion-web.xml），B-25
<security-role-mapping> 要素（orion-web.xml），B-26
<service-ref-mapping> 要素（orion-web.xml），B-27
<servlet-chaining> 要素（orion-web.xml），B-27
<session-tracking> 要素（orion-web.xml），B-29
<user> 要素（orion-web.xml），B-31
<virtual-directory> 要素（orion-web.xml），B-31
<web-app-class-loader> 要素（orion-web.xml），B-33
<web-app> 要素（orion-web.xml），B-32

A

access-mask 要素（orion-web.xml），B-5
Application Server Control コンソール
  「EJB 参照マッピング」ページ，A-10
  web.xml および orion-web.xml の表示，A-7
  Web モジュールの管理，A-3
  Web モジュールの構成，A-4
  Web モジュールのパフォーマンス，A-3
  「Web モジュール」ホームページ , サマリー，A-3
  概要，2-2，A-2
  「環境エントリ・マッピング」ページ，A-10
  「構成プロパティ」ページ，A-5
  「サーブレット・マッピング」ページ，A-7
  トップレベルの「Web モジュール」ページ，A-3
  「フィルタ・マッピング」ページ，A-7
  「リソース参照の参照コンテキスト」ページ，A-11
  「リソース参照マッピング」ページ，A-9
Application Server Control コンソールの「Web モジュール」ホームページ , アクセス，A-2

C

classpath 要素（orion-web.xml），B-6
context-attribute 要素（orion-web.xml），B-7
context-param-mapping 要素（orion-web.xml），B-7
contextpath（URL の構成要素），2-5
Cookie
  Cookie メソッド , サマリー，3-12
  OC4J がセッション・トラッキングに使用，3-4
  構成，3-11
  サンプル・サーブレット，3-14
  取得 , 表示 , 追加，3-13
  有効化または無効化，3-3

D

DeclaresRoles 注釈，7-7
destroy() サーブレット・メソッド，1-5
DMS（ダイナミック・モニタリング・サービス），9-6
doDelete() サーブレット・メソッド，1-5
doGet() サーブレット・メソッド，1-5
doPost() サーブレット・メソッド，1-5
doPut() サーブレット・メソッド，1-5

E

ejb-ref-mapping 要素（orion-web.xml），B-8
EJB 注釈，7-4
Enterprise JavaBeans，「EJB」を参照
env-entry-mapping 要素（orion-web.xml），B-8
expiration-setting 要素（orion-web.xml），B-9

F

FilterChain インタフェース，4-5
FilterConfig インタフェース，4-5
Filter インタフェース，4-4

G

GET, HTTP リクエスト，6-2
getServletConfig() サーブレット・メソッド，1-5
getServletInfo() サーブレット・メソッド，1-5
global-web-application.xml ファイル
  web.xml との関係 , orion-web.xml，B-4

概要，B-3
要素の階層，B-4
要素の説明，B-5
group 属性（orion-web.xml），B-9

H

host（URL の構成要素），2-4
host-access 要素（orion-web.xml），B-10
HttpServletRequest インタフェース，1-6
HttpServletResponse インタフェース，1-6
HttpServlet クラス
メソッドの概要，1-5
メソッドをオーバーライドする場面，6-2
HttpSessionActivationListener インタフェース，5-5
HttpSessionAttributeListener インタフェース，5-4
HttpSessionBindingEvent クラス，5-4
HttpSessionBindingListener インタフェース，5-5
HttpSessionEvent クラス，5-4
HttpSessionListener インタフェース，5-4
HttpSession メソッド，3-6

I

init() サーブレット・メソッド，1-5
ip-access 要素（orion-web.xml），B-10

J

jazn-web-app 要素（orion-web.xml），B-11
JDBC ドライバ , パッケージされたものの使用，B-33
JNDI
初期コンテキスト・ファクトリ，8-11
データソースの名前および場所，8-3
データソースのルックアップ，8-4
JSP パラメータ
enable-jsp-dispatcher-shortcut，B-19
jsp-cache-directory，B-20
jsp-cache-tlds，B-20
jsp-print-null，B-20
jsp-taglib-locations，B-21
jsp-timeout，B-21
simple-jsp-mapping，B-22
JSR-77 のサポート，2-2
JSR-88 のサポート，2-2

L

load-on-startup, OC4J，2-16
lookup-context 要素（orion-web.xml），B-13

M

MBean
MBean ブラウザ，2-2
OC4J での管理，A-12
Web モジュールの MBean, サマリー，A-13
定義，2-2
mime-mappings 要素（orion-web.xml），B-14

O

OC4J のデフォルトの Web アプリケーション，A-13
OC4J のデフォルトのアプリケーション，A-13
ojsp-init 要素（orion-web.xml），B-14
Oracle Application Server 環境，2-3
OracleAS JAAS Provider のユーザー・コンテキスト，B-11
orion-web-app 要素（orion-web.xml），B-18
orion-web.xml ファイル
Application Server Control コンソールによる表示，A-7
web.xml との関係 , global-web-application.xml，B-4
概要，B-3
要素の階層，B-4
要素の説明，B-5

P

PersistenceContext(s) 注釈，7-7
PersistenceUnit(s) 注釈，7-6
POST, HTTP リクエスト，6-2，6-11
PostConstruct 注釈，7-6
PreDestroy 注釈，7-6
prot（URL の構成要素），2-4
protocol（URL の構成要素），2-4

R

request-tracker 要素（orion-web.xml），B-24
resource-env-ref-mapping 要素（orion-web.xml），B-24
resource-ref-mapping 要素（orion-web.xml），B-25
Resources 注釈，7-5
Resource 注釈，7-5
RunAs 注釈，7-8

S

security-role-mapping 要素（orion-web.xml），B-26
service() サーブレット・メソッド，1-4，1-5
service-ref-mapping 要素（orion-web.xml），B-27
servlet-chaining 要素（orion-web.xml），B-27
ServletContextAttributeEvent クラス，5-3
ServletContextAttributeListener インタフェース，5-3
ServletContextEvent クラス，5-3
ServletContextListener インタフェース，5-3
servlet.jar の CLASSPATH 設定，6-7
servletpath（URL の構成要素），2-5
ServletRequestAttributeEvent クラス，5-6
ServletRequestAttributeListener インタフェース，5-6
ServletRequestEvent クラス，5-6
ServletRequestListener インタフェース，5-6
session-tracking 要素（orion-web.xml），B-29

U

URL の構成要素 , サマリー，2-4
URL リライティング，3-4
user 要素（orion-web.xml），B-31

V

virtual-directory 要素（orion-web.xml），B-31

W

web-app-class-loader 要素（orion-web.xml），B-33
web-app 要素（orion-web.xml），B-32
WebServiceRef 注釈，7-7
web.xml ファイル
  Application Server Control コンソールによる表示，A-7
  orion-web.xml との関係，global-web-application.xml，B-4
  概要，B-2
Web サイト間でのアプリケーションの共有（スタンドアロン），3-6
Web モジュールと Web アプリケーション，1-2
Web モジュールの XML 構成ファイル，B-1

X

XML パーサー，パッケージされたものの使用，B-33

い

イベント・リスナー，「リスナー」を参照
インクルード
  インクルード・サーブレットのフィルタ，4-10
  基本，別のサーブレットのインクルード，6-13
  実装手順，6-14
  使用する理由，6-14
  例，6-15

え

永続セッション・データ，3-3

か

管理
  Application Server Control コンソールの Web モジュールの「管理」，A-3
  Application Server Control コンソール・ページ，A-4
  JSR-77 のサポート，2-2
  OC4J での MBean 管理，A-12
  OC4J の概要，2-2
  Web モジュールの MBean，サマリー，A-13
  Web モジュールの構成ファイル，B-1
管理された OC4J，2-3
管理されない OC4J，2-3

こ

構成
  Application Server Control コンソール・ページ，A-4
  Cookie，3-11
  Web モジュールの MBean，サマリー，A-13
  Web モジュールの構成ファイル，B-1
  セッション・トラッキング，3-3
コンテキスト・ルート，2-5
コンテナ，サーブレット，1-9

さ

サーブレット・インタフェース，1-5
サーブレットからの EJB
  Application Server Control コンソールの「EJB 参照マッピング」ページ，A-10
  OC4J および OracleAS のサポート，8-11
  ORMI, IIOP の使用，8-11
  同じ場所への配置，8-11
  使用する理由，8-10
  使用例，8-11
  リモート・ルックアップ用のリモート・フラグ，8-13
  ルックアップ・カテゴリ，8-11
  ローカル・インタフェースとリモート・インタフェース，8-12
サーブレットからの JDBC
  コードの実装，8-4
  サーブレットの例，8-5
  使用する理由，8-2
  データソースの構成，8-2
サーブレット構成オブジェクト，1-11
サーブレット・コンテキスト
  基本，1-11
  取得，1-12
  メソッド，1-12
  リスナーのインタフェース，属性の変更，5-3
  リスナーのインタフェース，ライフ・サイクルの変更，5-3
サーブレット・コンテナ，1-9
サーブレットと EJB の同じ場所への配置，8-11
サーブレットの HTML フォーム，6-7
サーブレットの起動
  Oracle Application Server 環境での，2-11
  URL の構成要素のサマリー，2-4
  クラス名による（OC4J 固有），2-10
  スタンドアロン OC4J での，2-10
サーブレットのフォーム，6-7
サーブレット・フィルタ，「フィルタ」を参照
サンプル・サーブレット
  cookie サーブレット，3-14
  HTML フォームおよびリクエスト・パラメータ，6-9
  JDBC 問合せ，8-5
  サーブレット・インクルード，6-15
  セキュリティ，POST の使用，6-11
  セッション・サーブレット，3-8
  セッションのライフ・サイクル・イベント・リスナー，5-9
  単純な例，6-6
  デモの場所，OTN，1-1
  フィルタ，単純，4-8
  フィルタ・レスポンス，4-12
  リクエスト情報の取得，6-12

し

事前ロード，OC4J でのサーブレット，2-16
自動エンコーディング，セッション（未サポート），3-5, B-29
シングルスレッド・モデル，サーブレット，9-3

す

スタンドアロン環境，2-3
スレッド・モデル , 概要，1-13

せ

セキュリティ
URL セキュリティのための POST メソッド，6-11
保護された接続によるセッション・トラッキング，3-5
セッション
Cookie, OC4J がセッション・トラッキングに使用，3-4
Cookie, サーブレットでの一般的な使用，3-11
Cookie, 有効化または無効化，3-3
Cookie とセッション属性，3-3
HttpSession メソッド , サマリー，3-6
OC4J でのセッション・トラッキング，3-3
URL リライティング , OC4J がセッション・トラッキングに使用，3-4
永続データ，3-3
概要 , 使用する場面，1-13
サンプル・サーブレット，3-8
セッション ID, 概要，3-2
セッション・オブジェクト , 概要，3-2
セッション・トラッキングの構成，3-3
属性 , 追加および取得，3-7
タイムアウト , 指定，3-16
保護された接続によるセッション・トラッキング，3-5
明示的なキャンセル，3-17
リスナーのインタフェース , 移行の変更，5-5
リスナーのインタフェース , オブジェクト・バインドの変更，5-5
リスナーのインタフェース , 属性の変更，5-4
リスナーのインタフェース , ライフ・サイクルの変更，5-4
セッションのタイムアウト , 指定，3-16

た

ダイナミック・モニタリング・サービス（DMS），9-6
他のサーブレットへのディスパッチ（インクルードまたは転送），6-13

ち

注釈
DeclaresRoles，7-7
EJB，7-4
PersistenceContext(s)，7-7
PersistenceUnit(s)，7-6
PostConstruct，7-6
PreDestroy，7-6
Resource，7-5
Resources，7-5
RunAs，7-8
WebServiceRef，7-7
概要，7-2

て

データソース
JNDI ルックアップ , データベース接続，8-4
構成，8-2
デプロイ
JSR-88 のサポート，2-2
デプロイ・プラン，2-2
デプロイ・プラン・エディタ，2-2
デモの場所 , OTN，1-1
転送
基本 , 別のサーブレットへの転送，6-13
実装手順，6-14
使用する理由，6-14
転送サーブレットのフィルタ，4-10
例，6-15

は

パスの設定，6-7
パフォーマンス , サーブレット，9-6，A-3

ひ

ヒント
1970 年 1 月 1 日を基点とするミリ秒単位の日付コンストラクタ，3-7
URL セキュリティのための POST の使用，6-10
パスの設定 , CLASSPATH，6-7
レスポンスの Writer オブジェクト / 出力ストリームのクローズは不要，6-4

ふ

フィルタ
Application Server Control コンソールの「フィルタ・マッピング」ページ，A-7
インクルード / 転送のフィルタ，4-10
インタフェース , 標準，4-4
概要，6-17
構成手順，4-7
コンテナによる起動，4-2
実装手順，4-6
処理 , 一般的なもの，4-3
フィルタ・チェーンの構造，4-8
リクエスト / レスポンスのラップ，4-11
例 , 単純なフィルタ，4-8
例 , レスポンス・フィルタ，4-12

め

メトリック , サーブレット，A-3

ゆ

ユーザーからの入力 , HTML フォーム，6-7

ら

ライフ・サイクル , サーブレット，1-4

り

リクエスト
　情報の取得 , リクエスト，6-12
　フォームからのユーザー入力のリクエスト・パラメータ，6-7
　リクエスト・オブジェクト，1-6
　リクエストのフィルタ / ラップ，4-11
　リスナーのインタフェース , 属性の変更，5-6
　リスナーのインタフェース , ライフ・サイクルの変更，5-6
　例 , フォームおよびリクエスト・パラメータ，6-9
　例 , リクエスト情報の取得，6-12
リクエスト・ディスパッチャ , 他のサーブレットへのインクルード / 転送，6-13
リスナー
　概要 , カテゴリ，5-2
　概要 , 使用する場面，6-17
　構成，5-8
　サーブレット・コンテキストの属性インタフェース，5-3
　サーブレット・コンテキストのライフ・サイクル・インタフェース，5-3
　実装するためのコード，5-7
　セッションの移行インタフェース，5-5
　セッションの属性インタフェース，5-4
　セッションのバインド・インタフェース，5-5
　セッションのライフ・サイクル・インタフェース，5-4
　リクエストの属性インタフェース，5-6
　リクエストのライフ・サイクル・インタフェース，5-6
　例，5-9
リダイレクト（代替 URL への），1-8
リモート・フラグ , リモート EJB ルックアップ用，8-13

れ

レスポンス
　例 , レスポンス・フィルタ，4-12
　レスポンス・オブジェクト，1-6
　レスポンスのフィルタ / ラップ，4-11