

Oracle8*i*

JavaServer Pages 開発者ガイドおよびリファレンス

リリース 8.1

2000 年 11 月

部品番号 : J02316-01

ORACLE®

Oracle®i JavaServer Pages 開発者ガイドおよびリファレンス, リリース 8.1

部品番号 : J02316-01

原本名 : JavaServer Pages Developer's Guide and Reference, Release 8.1.7

原本部品番号 : A83726-01

原本著者 : Brian Wright

原本協力者 : Michael Freedman, Julie Basu, Alex Yiu, Sunil Kunisetty, YaQing Wang, Hal Hildebrand, Jasen Minton, Matthieu Devin, Jose Alberto Fernandez, Olga Peschansky, Jerry Schwarz, Clement Lai, Shinji Yoshida, Robert Pang, Ralph Gordon, Shiva Prasad, Sharon Malek, Jeremy Lizt, Kuassi Mensah, Susan Kraft, Sheryl Maring, Ellen Barnes, Angie Long, Sanjay Singh, Olaf van der Geest

Copyright © 2000, Oracle Corporation. All rights reserved.

Printed in Japan.

制限付権利の説明

プログラム（ソフトウェアおよびドキュメントを含む）の使用、複製または開示は、オラクル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に関する法律により保護されています。

当プログラムのリバース・エンジニアリング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ことがあります。オラクル社は本ドキュメントの無謬性を保証しません。

* オラクル社とは、Oracle Corporation（米国オラクル）または日本オラクル株式会社（日本オラクル）を指します。

危険な用途への使用について

オラクル社製品は、原子力、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を用途として開発されておりません。オラクル社製品を上述のようなアプリケーションに使用することについての安全確保は、顧客各位の責任と費用により行ってください。万一かかる用途での使用によりクレームや損害が発生いたしましても、日本オラクル株式会社と開発元である Oracle Corporation（米国オラクル）およびその関連会社は一切責任を負いかねます。当プログラムを米国国防総省の米国政府機関に提供する際には、『Restricted Rights』と共に提供してください。この場合次の Notice が適用されます。

Restricted Rights Notice

Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

このドキュメントに記載されているその他の会社名および製品名は、あくまでその製品および会社を識別する目的にのみ使用されており、それぞれの所有者の商標または登録商標です。

目次

はじめに	ix
対象読者	x
このマニュアルの構成	x
関連マニュアル	xii
その他のリソース	xiv
表記規則	xiv

1 概要

JavaServer Pages の概要	1-2
JSP ページの内容	1-2
JSP コードとサーブレット・コードの利便性の比較	1-3
ページ表現とビジネス・ロジックの分離 - JavaBeans のコール	1-4
JSP ページと代替マークアップ言語	1-5
JSP の実行	1-6
JSP コンテナの概要	1-6
JSP ページとオンデマンド変換	1-6
JSP ページの要求	1-7
JSP 構文要素の概要	1-8
ディレクティブ	1-9
スクリプト要素	1-11
JSP のオブジェクトと有効範囲	1-13
JSP のアクションと <jsp: > タグ・セット	1-16
タグ・ライブラリ	1-21

2 Oracle の JSP 実装の概要

サードパーティ環境間の移植性と機能性	2-2
OracleJSP の移植性	2-2
サードパーティ 2.0 環境向けの OracleJSP の拡張機能	2-2
Oracle 環境における OracleJSP のサポート	2-3
Oracle Servlet Engine (OSE) の概要	2-3
Oracle Internet Application Server の概要	2-4
Apache により駆動される Oracle HTTP Server のロール	2-5
Oracle Web Application のデータベース・アクセス方法	2-7
他の Oracle JSP 環境の概要	2-8
非 Oracle 環境における OracleJSP のサポート	2-10
OracleJSP のプログラム拡張機能の概要	2-10
移植可能な OracleJSP の拡張機能の概要	2-11
Oracle 固有の拡張機能の概要	2-13
OracleJSP と Oracle PL/SQL Server Pages の併用	2-15
OracleJSP のリリースと機能セットの概要	2-16
Oracle プラットフォームで提供される OracleJSP のリリース	2-16
リリース 1.0.0.6.x に関する OracleJSP の機能上の注意	2-17
OracleJSP の実行モデル	2-18
オンデマンド変換モデル	2-18
Oracle Servlet Engine の事前変換モデル	2-18
OracleJSP に対する Oracle JDeveloper のサポート	2-20

3 基本事項

予備的な考慮事項	3-2
インストールと構成の概要	3-2
開発環境と配布環境	3-2
クライアント側の考慮事項	3-3
アプリケーション・ルート機能とドキュメント・ルート機能	3-3
サードパーティ 2.2 環境におけるアプリケーション・ルート	3-3
サードパーティ 2.0 環境における OracleJSP のアプリケーション・ルート機能	3-4
JSP アプリケーションおよびセッションの概要	3-5
一般的な OracleJSP アプリケーションおよびセッションのサポート	3-5
JSP のデフォルトのセッション要求	3-5

JSP とサーブレットの相互作用	3-6
JSP ページからのサーブレットの起動	3-6
JSP ページから起動したサーブレットへのデータの受渡し	3-7
サーブレットからの JSP ページの起動	3-7
JSP ページとサーブレット間でのデータの受渡し	3-8
JSP とサーブレットの相互作用の例	3-9
JSP のリソース管理	3-10
標準的なセッション・リソース管理 — HttpSessionBindingListener	3-10
リソース管理のための Oracle の拡張機能の概要	3-15
JSP のランタイム・エラー処理	3-16
JSP エラー・ページの使用	3-16
JSP エラー・ページの例	3-17
データベース・アクセスに関する JSP の初期サンプル	3-19

4 重要な考慮事項

一般的な JSP のプログラミング方法、ヒントおよび注意事項	4-2
JavaBeans とスクリプトレット	4-2
JSP ページにおける Enterprise JavaBeans の使用	4-2
JDBC のパフォーマンス強化機能の使用	4-5
静的挿入と動的挿入	4-8
JSP タグ・ライブラリの作成と使用を検討する場合	4-10
セントラル・チェッカ・ページの使用	4-11
JSP ページの大量の静的コンテンツに関する回避策	4-12
メソッド変数宣言とメンバー変数宣言	4-13
page ディレクティブの特性	4-14
JSP の空白保持とバイナリ・データでの使用	4-15
OracleJSP の構成上の重要な問題	4-18
JSP 実行の最適化	4-18
CLASSPATH とクラス・ローダーの問題（非 OSE のみ）	4-19
OracleJSP のランタイムの考慮事項（非 OSE のみ）	4-22
ページの動的再変換	4-22
ページの動的再ロード	4-23
クラスの動的再ロード	4-24

Oracle Servlet Engine に関する考慮事項	4-25
Oracle8i JVM の JVM とサーバー側 JDBC 内部ドライバの概要	4-26
Oracle8i JVM の接続	4-26
Oracle Servlet Engine による JNDI の使用	4-29
OracleJSP のランタイム構成パラメータの等価コード	4-29
Apache/JServ サブレット環境に関する考慮事項	4-30
Oracle Internet Application Server による Apache/JServ の使用	4-30
Apache/JServ における動的挿入および転送	4-31
Apache/JServ 用のアプリケーション・フレームワーク	4-33
JSP とサブレットのセッション共有	4-33
ディレクトリ別名の変換	4-34

5 OracleJSP の拡張機能

移植可能な OracleJSP プログラミング拡張機能	5-2
JML の拡張データ型	5-2
OracleJSP の XML および XSL サポート	5-8
Oracle Database-Access JavaBeans	5-12
OracleJSP の SQL 用タグ・ライブラリ	5-22
Oracle 固有のプログラミング拡張機能	5-29
OracleJSP のイベント処理 — JspScopeListener	5-30
OracleJSP による Oracle SQLJ のサポート	5-31
サブレット 2.0 に対する OracleJSP のアプリケーションおよびセッションのサポート	5-34
globals.jsa の機能の概要	5-35
globals.jsa の構文とセマンティックの概要	5-37
globals.jsa のイベント・ハンドラ	5-39
グローバル宣言およびディレクティブ	5-43

6 JSP の変換と配布

OracleJSP トランスレータの機能	6-2
生成されるコードの機能	6-2
生成されるパッケージとクラスの名前（オンデマンド変換）	6-4
生成されるファイルと位置（オンデマンド変換）	6-6
サンプル・ページ実装クラスのソース	6-7

Oracle8i への配布時の機能とロジスティックの概要	6-11
Java 用のデータベース・スキーマ・オブジェクト	6-12
フロントエンド Web サーバーとしての Oracle HTTP Server	6-14
Oracle Servlet Engine の URL	6-14
Oracle Servlet Engine 内の JSP アプリケーション用の静的ファイル	6-17
サーバー側変換とクライアント側変換の比較	6-18
Oracle8i にホットロードされるクラスの概要	6-20
変換と Oracle8i への配布に使用するツールとコマンド	6-22
ojspc 事前変換ツール	6-22
loadjava ツールの概要	6-34
sess_sh セッション・シェル・ツールの概要	6-36
サーバー側変換を伴う Oracle8i への配布	6-38
Oracle8i への未変換の JSP ページのロード (loadjava)	6-38
Oracle8i での JSP ページの変換と公開 (セッション・シェルの publishjsp)	6-39
クライアント側変換を伴う Oracle8i への配布	6-50
JSP ページの事前変換 (ojspc)	6-51
Oracle8i への変換済み JSP ページのロード (loadjava)	6-55
Oracle8i でのページ実装クラスのホットロード	6-58
Oracle8i での変換済み JSP ページの公開 (セッション・シェルの publishservlet)	6-59
JSP 配布に関するその他の考慮事項	6-62
Oracle Internet Application Server と Oracle Servlet Engine のドキュメント・ルートの比較	6-63
非 OSE 環境向けの事前変換のための ojspc の使用	6-64
実行なしの一般的な JSP 事前変換	6-64
バイナリ・ファイルのみの配布	6-65
WAR の配布	6-66
JDeveloper を使用した JSP ページの配布	6-67

7 JSP のタグ・ライブラリと Oracle の JML タグ

標準タグ・ライブラリのフレームワーク	7-2
カスタム・タグ・ライブラリの実装の概要	7-2
タグ・ハンドラ	7-4
スクリプト変数と Tag-Extra-Info クラス	7-7
外部タグ・ハンドラ・インスタンスへのアクセス	7-9
タグ・ライブラリ記述ファイル	7-10

タグ・ライブラリ用の web.xml の使用	7-11
taglib ディレクティブ	7-12
全体の例: カスタム・タグの定義と使用	7-13
JSP マークアップ言語 (JML) のサンプル・タグ・ライブラリの概要	7-18
JML タグ・ライブラリの原理	7-19
JML タグのカテゴリ	7-19
JML のタグ・ライブラリ記述ファイルと taglib ディレクティブ	7-20
JSP マークアップ言語 (JML) タグの説明	7-28
構文の表記規則と注意事項	7-28
Bean バインド・タグの説明	7-28
ロジックおよびフロー制御タグの説明	7-32

8 OracleJSP の NLS サポート

page ディレクティブでのコンテンツ型の設定	8-2
コンテンツ型の動的設定	8-3
マルチバイト・パラメータのコード化に対する OracleJSP の拡張サポート	8-4
非マルチバイト・サブレット・コンテナのオーバーライドにおける translate_params の効果 ..	8-5
translate_params 構成パラメータの等価コード	8-6
translate_params に依存する NLS のサンプル	8-6
translate_params に依存しない NLS のサンプル	8-9

9 サンプル・アプリケーション

基本的なサンプル	9-2
hello ページ — hellouser.jsp	9-2
usebean ページ — usebean.jsp	9-3
ショッピング・カート・ページ — cart.jsp	9-4
JDBC のサンプル	9-9
単純な問合せ — SimpleQuery.jsp	9-10
ユーザー指定の問合せ — JDBCQuery.jsp	9-11
問合せ Bean を使用した問合せ — UseHtmlQueryBean.jsp	9-13
接続キャッシュ — ConnCache3.jsp および ConnCache1.jsp	9-16
Database-Access JavaBeans のサンプル	9-20
DBBean を使用するページ — DBBeanDemo.jsp	9-21
ConnBean を使用するページ — ConnBeanDemo.jsp	9-22

CursorBean を使用するページ — CursorBeanDemo.jsp	9-23
ConnCacheBean を使用するページ — ConnCacheBeanDemo.jsp	9-25
カスタム・タグのサンプル	9-27
JML タグのサンプル — hellouser_jml.jsp	9-27
他のカスタム・タグ・サンプルのポインタ	9-29
Oracle 固有のプログラミング拡張機能のサンプル	9-29
JspScopeListener を使用するページ — scope.jsp	9-29
XML の問合せ — XMLQuery.jsp	9-33
SQLJ の問合せ — SQLJSelectInto.sqljsp および SQLJIterator.sqljsp	9-34
サーブレット 2.0 環境に globals.jsa を使用するサンプル	9-38
アプリケーション・イベント用の globals.jsa の例 — lotto.jsp	9-38
アプリケーションおよびセッション・イベント用の globals.jsa の例 — index1.jsp	9-41
グローバル宣言用の globals.jsa の例 — index2.jsp	9-43

A 一般的なインストールと構成

システム要件	A-2
OracleJSP のインストールと Web サーバーの構成	A-3
OracleJSP の必須ファイルとオプション・ファイル	A-3
OracleJSP を実行する Web サーバーおよびサーブレット環境の構成	A-6
OracleJSP の構成	A-13
OracleJSP の構成パラメータ（非 OSE）	A-13
OracleJSP の構成パラメータ設定	A-23

B サーブレットおよび JSP の技術的な背景情報

サーブレットの背景情報	B-2
サーブレット・テクノロジーの概要	B-2
サーブレット・インタフェース	B-3
サーブレット・コンテナ	B-3
サーブレット・セッション	B-4
サーブレット・コンテキスト	B-6
イベント・リスナーを介したアプリケーションのライフ・サイクル管理	B-7
サーブレットの起動	B-8
Web アプリケーションの階層	B-8
標準 JSP インタフェースおよびメソッド	B-11

C コンパイル時 JML タグ・サポート

JML のコンパイル時とランタイムの考慮事項とロジスティック C-2

 コンパイル時とランタイムの全般的な考慮事項 C-2

 コンパイル時の JML サポート用の taglib ディレクティブ C-2

JML コンパイル時 /1.0.0.6.x 構文のサポート C-3

 JML の Bean 参照と式、コンパイル時実装 C-3

 JML 式による属性設定 C-4

JML コンパイル時 /1.0.0.6.x タグのサポート C-6

 JML タグの概要、1.0.0.6.x/ コンパイル時と 1.1.0.0.0/ ランタイム C-6

 追加の JML タグの説明、コンパイル時実装 C-7

索引

はじめに

このマニュアルでは、Sun Microsystems が規定している JavaServer Pages (JSP) テクノロジーの Oracle による実装について説明します。Sun が規定している標準機能と、OracleJSP 製品における Oracle 固有の拡張機能について説明します。

注意： このマニュアルは、Oracle8i リリース 8.1.7 に付属の OracleJSP を対象としています。このリリースの OracleJSP は、OracleJSP リリース 8.1.7 または OracleJSP リリース 1.1.0.0.0 と呼ばれることがあります。これは、Sun Microsystems の『JavaServer Pages Specification, Version 1.1』の実装です。旧バージョンの OracleJSP には、『JavaServer Pages Specification, Version 1.0』が実装されていました。

対象読者

このマニュアルは、OracleJSP を使用して JavaServer Pages テクノロジーに準拠する Web アプリケーションを作成しようとする開発者を対象としています。すでに Web およびサーブレット環境が稼働していることと、次の知識をよく理解していることを前提としています。

- Web テクノロジー全般
- サーブレット・テクノロジー全般（技術的な背景情報の一部については、[付録 B](#) を参照）
- Web サーバーおよびサーブレット環境の構成方法
- HTML
- Java
- Oracle JDBC（JSP アプリケーションが Oracle データベースにアクセスする場合）
- Oracle SQLJ（JSP データベース・アプリケーションが SQLJ を使用する場合）

このマニュアルは、Oracle JSP 拡張機能と、Oracle Servlet Engine（Oracle8i 内の Web サーバーとサーブレットのコンテナ）で JSP ページを実行する場合に固有の機能とロジスティックに重点を置いています。

標準的な JSP 1.1 テクノロジーと構文に関する概要は、[第 1 章](#)および他の章に示しますが、詳しくは説明しません。標準的な JSP 1.1 機能の詳細は、Sun Microsystems の『JavaServer Pages Specification, Version 1.1』または他の該当するリファレンス・ドキュメントを参照してください。

JSP 1.1 仕様はサーブレット 2.2 環境に依存しているため、このマニュアルもサーブレット 2.2 環境に重点を置いています。ただし、OracleJSP は旧バージョンのサーブレット環境用に特殊機能を備えており、これらの機能はサーブレット 2.0 環境、特に Apache/JServ に関連しているため、この点については例外的に説明します。

このマニュアルの構成

このマニュアルの構成は、次のとおりです。

第 1 章「概要」

標準的な JSP 1.1 テクノロジーについて説明します。
（詳細なリファレンスではありません。）

第 2 章「Oracle の JSP 実装の概要」

Oracle および非 Oracle の JSP 環境における OracleJSP のサポートと、Oracle JSP の拡張機能および特性について説明します。

第 3 章「基本事項」

基本的な JSP プログラミングの考慮事項を説明し、データベース・アクセスの初期サンプルを示します。

第 4 章「重要な考慮事項」	開発者が注意する必要があるプログラミングと構成全般に関する様々な問題点について説明します。また、OSE および Apache/JServ 環境に固有の考慮事項も取り扱います。
第 5 章「OracleJSP の拡張機能」	OracleJSP の拡張機能、つまり、Oracle 固有の拡張機能と他の JSP 環境に移植可能な拡張機能について説明します。
第 6 章「JSP の変換と配布」	JSP ページを Oracle8i に配布して Oracle Servlet Engine で実行するための手順とロジスティックに重点を置き、一般的な JSP の変換および配布機能と問題点についても説明します。
第 7 章「JSP のタグ・ライブラリと Oracle の JML タグ」	カスタム・タグ・ライブラリに関する基本的な JSP 1.1 のフレームワークの概要を説明し、Oracle JML サンプル・タグ・ライブラリの JSP 1.1 (ランタイム) 実装の概要とタグについて説明します。
第 8 章「OracleJSP の NLS サポート」	標準と Oracle 固有の両方の各国語サポート機能について説明します。
第 9 章「サンプル・アプリケーション」	標準的な JSP テクノロジと Oracle の拡張機能を対象とする一連のサンプル・アプリケーションを示します。
付録 A「一般的なインストールと構成」	OracleJSP の必須ファイルとオプション・ファイル、Apache/JServ や Tomcat のような非 Oracle 環境向けの構成手順およびオンデマンド変換のための OracleJSP 構成パラメータについて説明します。
付録 B「サーブレットおよび JSP の技術的な背景情報」	サーブレット・テクノロジに関する簡潔な背景情報と、変換済みページのための標準 JSP インタフェースについて説明します。
付録 C「コンパイル時 JML タグ・サポート」	Oracle JML サンプル・タグ・ライブラリ (JSP リリース 1.1 以前でサポート) のコンパイル時実装の概要と、第 7 章に記載されているランタイム実装でサポートされないタグについて説明します。

関連マニュアル

次のドキュメントを参照してください。

- 『Oracle8i Java 開発者ガイド』

Oracle8i における Java の基本概念を説明し、サーバー側の構成と機能に関する一般情報を提供します。このドキュメントの内容は、特定の製品（JDBC、SQLJ または EJB など）ではなく、Oracle Java プラットフォーム全体に関するものです。

- 『Oracle8i Oracle Servlet Engine ユーザーズ・ガイド』

Oracle8i 内の Web サーバーとサーブレットのコンテナである Oracle Servlet Engine の使用方法を説明しています。

- 『Oracle8i Java Tools リファレンス』

Oracle8i で使用する、または Oracle8i へのアプリケーションの配布に使用する Java 関連のツールとユーティリティ（Oracle8i セッション・シェルや loadjava ツールなど）について説明しています。

- 『Oracle8i JDBC 開発者ガイドおよびリファレンス』

JDBC 規格（Java Database Connectivity 向け）の Oracle 実装のプログラム構文と機能について説明しています。Oracle JDBC ドライバの概要、JDBC 1.22 および 2.0 の機能の Oracle 実装の詳細、および Oracle JDBC 型の拡張機能とパフォーマンス拡張機能の説明が含まれています。

- 『Oracle8i JPublisher ユーザーズ・ガイド』

Oracle JPublisher ユーティリティを使用してオブジェクト型や他のユーザー定義型を Java のクラスに変換する方法を説明しています。オブジェクト型、VARRAY 型、ネストした表の型またはオブジェクト参照型を使用する SQLJ または JDBC アプリケーションを開発する場合は、JPublisher でカスタムの Java クラスを生成して型にマップできます。

- 『Oracle8i SQLJ 開発者ガイドおよびリファレンス』

SQLJ を使用して静的な SQL 操作を Java コードに直接埋め込む方法と、SQLJ 言語構文および SQLJ トランスレータのオプションと機能について説明しています。標準的な SQLJ 機能と Oracle 固有の SQLJ 機能の両方について説明します。

- 『Oracle8i Java ストアド・プロシージャ開発者ガイド』

Java ストアド・プロシージャについて説明しています。Java ストアド・プロシージャは、Oracle8i サーバー上で直接実行されるプログラムです。ストアド・プロシージャ（ファンクション、プロシージャ、データベース・トリガーおよび SQL メソッド）を使用すると、Java 開発者はビジネス・ロジックをサーバー・レベルで実装し、アプリケーションのパフォーマンス、拡張性およびセキュリティを向上できます。

- 『Oracle8i Enterprise JavaBeans 開発者ガイドおよびリファレンス』
Oracle Enterprise JavaBeans の実装と拡張機能について説明しています。
- 『Oracle8i CORBA 開発者ガイド』
Oracle での CORBA の実装と拡張機能について説明しています。

次のマニュアルは、OracleJSP が組み込まれた Oracle 製品に関するものです。各製品向けのインストールや構成など、JSP 情報が記載されています。

- 『Oracle Application Server JServlet および JSP アプリケーション開発者ガイド』
- Oracle JDeveloper オンライン・ヘルプ、特にトピック「JSP ページの作成」
- 『Web-to-go インプリメンテーション・ガイド』

以下のマニュアルも参照してください。

- 『Oracle8i アプリケーション開発者ガイド — XML』
Oracle XML-SQL ユーティリティに関する情報を提供しています。OracleJSP が提供する XML 関連のサポートに関する情報も、一部含まれています。
- 『Oracle8i アプリケーション開発者ガイド 基礎編』
Oracle8i データベースの使用とデータベース・アクセス・アプリケーションの作成に関する基本的な設計の概念とプログラミング機能を紹介しています。
- 『Oracle8i NLS ガイド』
NLS 環境変数、キャラクタ・セットおよび地域とロケールの設定情報が含まれています。また、一般的な NLS の問題、代表的な使用例および OCI と SQL のプログラマを対象とした NLS の考慮事項を説明しています。
- 『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
Oracle8i サーバーの一部として使用可能な PL/SQL パッケージを説明しています。その一部は、JDBC アプリケーションからのコールに役立ちます。
- 『PL/SQL ユーザーズ・ガイドおよびリファレンス』
SQL への Oracle のプロシージャ型言語拡張機能である PL/SQL の概念と機能を説明しています。
- 『Oracle8i SQL リファレンス』
Oracle データベースで情報管理に使用する SQL コマンドと機能の内容と構文の詳細を説明しています。
- 『Oracle8i Net8 管理者ガイド』
Oracle8 Connection Manager と Net8 ネットワーク管理の概要を説明しています。

- 『Oracle8i Advanced Security 管理者ガイド』
Oracle Advanced Security Option（旧称 ANO または ASO）の機能を説明しています。
- 『Oracle8i リファレンス・マニュアル』
Oracle8i サーバーに関する一般的な参照情報が記載されています。
- 『Oracle8i エラー・メッセージ』
Oracle8i サーバーから渡すことができるエラー・メッセージを説明しています。

その他のリソース

次のリソースは、Sun Microsystems から入手できます。

- JavaServer Pages に関する Javasoft の Web サイト
<http://www.javasoft.com/products/jsp/index.html>
- JavaServer Pages に関する jsp-interest ディスカッション・グループ
サブスクライブするには、メッセージ本体に次の行を挿入して
listserv@java.sun.com に電子メールを送信します。

subscribe jsp-interest yourlastname yourfirstname

ただし、投稿された電子メールの 1 日分のダイジェストのみを要求することをお勧めします。そのためには、メッセージ本体に次の行も追加します。

```
set jsp-interest digest
```

表記規則

このマニュアルでは、次の表記規則を使用しています。

表記規則	意味
...	サンプル・コード内の省略記号は、文またはその一部が省略されていることを示します。たとえば、通常は他にも文やコードが表示されている箇所でも、その文やコードが例と関係ない場合にこの記号が使用されます。
固定幅フォント	標準フォントのテキストに含まれる固定幅フォントは、コマンド、オプション名、パラメータ名、Java 構文、クラス名、オブジェクト名、メソッド名、変数名、Java の型、Oracle のデータ型、ファイル名およびディレクトリ名を示します。

表記規則	意味
イタリックの固定幅フォント	プログラムの文中のイタリックの固定幅フォントは、ユーザーが指定する必要がある要素を示します。
[イタリックの固定幅フォント]	プログラムの文中で大カッコで囲まれているイタリックの固定幅フォントは、ユーザーがオプションで指定できる要素を示します。

この章では、JavaServer Pages テクノロジーの標準的な機能を説明します。詳細は、Sun Microsystems の『JavaServer Pages Specification, Version 1.1』を参照してください。

(Oracle 固有の OracleJSP の機能の概要は、第 2 章「[Oracle の JSP 実装の概要](#)」を参照してください。付録 B「[サーブレットおよび JSP の技術的な背景情報](#)」にも、標準的なサーブレットおよび JSP テクノロジーに関連する背景情報が記載されています。)

説明するトピックは、次のとおりです。

- [JavaServer Pages の概要](#)
- [JSP の実行](#)
- [JSP 構文要素の概要](#)

JavaServer Pages の概要

JavaServer Pages は、Web アプリケーション（Web サーバー上で実行されるアプリケーション）から出力される動的なコンテンツ・ページを手軽に生成する方法として、Sun Microsystems が規定しているテクノロジーです。

このテクノロジーは Java サブレット・テクノロジーと密接に連携しており、Java コードの断片や外部 Java コンポーネントのコールを、Web ページの HTML コード（または XML のような他のマークアップ・コード）に含めることができます。JavaServer Pages (JSP) テクノロジーは、JavaBeans および Enterprise JavaBeans (EJB) のビジネス・ロジックおよび動的機能のフロントエンドとしても、優れた機能を発揮します。

JSP コードは、Web ページでは JavaScript のような他の Web スクリプト・コードとは異なります。標準の HTML ページに挿入できるものであれば、すべて JSP ページにも挿入できます。

データベース・アプリケーションの典型的な使用例では、JSP ページは JavaBeans や Enterprise JavaBeans などのコンポーネントをコールし、Bean は通常は JDBC または SQLJ を介してデータベースに直接または間接的にアクセスします。

JSP ページは、実行前（通常はオンデマンドで、場合によっては事前）に、Java サブレットに変換され、HTTP 要求を処理して他のサブレットに似た応答を生成します。JSP テクノロジーにより、サブレットのコーディングが容易になります。

さらに、JSP ページはサブレットと全面的に相互運用可能です。つまり、JSP ページとサブレットの間では、相互に出力を挿入または転送できます。

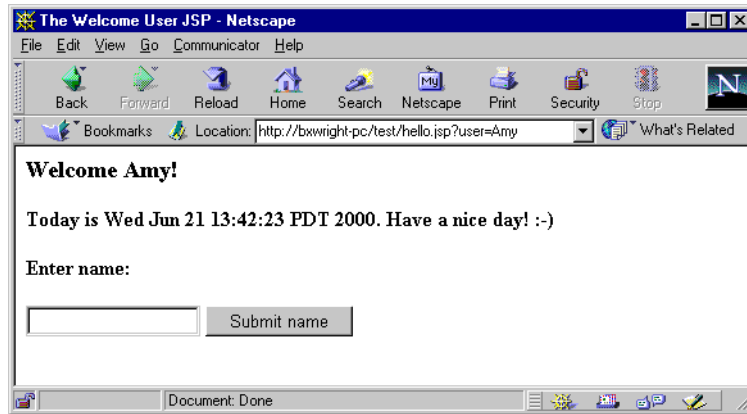
JSP ページの内容

次の例は、単純な JSP ページを示しています。（この例で使用している JSP 構文要素については、1-8 ページの「[JSP 構文要素の概要](#)」を参照してください）。

```
<HTML>
<HEAD><TITLE>The Welcome User JSP</TITLE></HEAD>
<BODY>
<% String user=request.getParameter("user"); %>
<H3>Welcome <%= (user==null) ? "" : user %>!</H3>
<P><B> Today is <%= new java.util.Date() %>. Have a nice day! :-)</B></P>
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

JSP ページでは、Java 要素は前述の例のように `<%` と `%>` のタグなどで設定されます。この例で、Java コードの断片は HTTP 要求オブジェクトからユーザー名を取得して出力し、現在の日付を取得します。

この JSP ページでは、ユーザーが「Amy」という名前を入力すると次の出力が生成されます。



JSP コードとサーブレット・コードの利便性の比較

サーブレットに単純な Java コードを使用するよりも、Java コードと HTML ページへの Java コールを組み合わせる方が便利です。JSP 構文には、動的な Web ページをコーディングするためのショートカットが用意されており、通常は、Java Servlet の構文より少数のコードですみます。次の例は、サーブレット・コードと JSP コードの比較を示しています。

サーブレット・コード

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

public class Hello extends HttpServlet
{
    public void doGet(HttpServletRequest rq, HttpServletResponse rsp)
    {
        rsp.setContentType("text/html");
        try {
            PrintWriter out = rsp.getWriter();
            out.println("<HTML>");
            out.println("<HEAD><TITLE>Welcome</TITLE></HEAD>");
            out.println("<BODY>");
```

```
        out.println("<H3>Welcome!</H3>");
        out.println("<P>Today is "+new java.util.Date()+".</P>");
        out.println("</BODY>");
        out.println("</HTML>");
    } catch (IOException ioe)
    {
        // (error processing)
    }
}
```

(標準的な `HttpServlet` 抽象クラス、`HttpServletRequest` インタフェースおよび `HttpServletResponse` インタフェースの背景情報については、B-3 ページの「[サーブレット・インタフェース](#)」を参照してください。)

JSP コード

```
<HTML>
<HEAD><TITLE>Welcome</TITLE></HEAD>
<BODY>
<H3>Welcome!</H3>
<P>Today is <%= new java.util.Date() %>.</P>
</BODY>
</HTML>
```

サーブレット・コードに比べ JSP 構文が単純であることに注意してください。特に、パッケージのインポートや `try...catch` ブロックなど、Java のオーバーヘッドが軽減されます。

また、JSP トランスレータにより出力される `.java` ファイル内では、サーブレットのコーディングのオーバーヘッドの大部分が自動的に処理されます。たとえば、標準的な `javax.servlet.jsp.HttpJspPage` インタフェースが直接または間接的に実装され (B-11 ページの「[標準 JSP インタフェースおよびメソッド](#)」を参照)、HTTP セッションを取得するためのコードが追加されます。

さらに、JSP ページの HTML はサーブレット・コードのように Java の `print` 文に埋め込まれていないため、HTML オーサリング・ツールを使用して JSP ページを作成できることに注意してください。

ページ表現とビジネス・ロジックの分離 - JavaBeans のコール

JSP テクノロジーにより、静的なページ表現を決定する HTML コードと、ビジネス・ロジックを処理して動的コンテンツを表示する Java コードの間で、開発作業を分離できます。したがって、Java ではなく HTML に精通した表現やレイアウトのスペシャリストと、HTML ではなく Java に精通したコードのスペシャリストの間で、メンテナンス作業を容易に分担できます。

通常の JSP ページでは、ほとんどの Java コードとビジネス・ロジックは、JSP ページに断片として埋め込まれるかわりに、JSP ページからコールされる JavaBeans または Enterprise JavaBeans に埋め込まれます。

JSP テクノロジーでは、JavaBeans クラスのインスタンスの定義と作成のために、次の構文が用意されています。

```
<jsp:useBean id="pageBean" class="mybeans.NameBean" scope="page" />
```

この例では、mybeans.NameBean クラスのインスタンス pageBean が作成されます (scope パラメータについては後述します)。

ページの後半で、この Bean インスタンスを次の例のように使用できます。

```
Hello <%= pageBean.getNewName() %> !
```

(この例では、ユーザー入力を介して発生するように、pageBean の newName 属性に「Julie」という名前があると、「Hello Julie!」と出力されます。)

ビジネス・ロジックをページ表現から分離することで、ビジネス・ロジックと動的コンテンツを担当する Java のエキスパートと、アプリケーション・ユーザーに表示される Web ページの静的表現とレイアウトを担当する HTML のエキスパートの間で、作業を容易に分担できます。Java のエキスパートは、NameBean クラスのコードを所有してメンテナンスする開発者で、HTML のエキスパートは、この JSP ページの .jsp ファイル内でコードを所有してメンテナンスする開発者です。

JavaBeans インスタンスを宣言する useBean や、Bean のプロパティにアクセスする getProperty および setProperty など、JavaBeans で使用するタグの詳細は、1-16 ページの「[JSP のアクションと<jsp:>タグ・セット](#)」を参照してください。

JSP ページと代替マークアップ言語

通常、JavaServer Pages テクノロジーは動的 HTML 出力に使用されますが、Sun Microsystems の『JavaServer Pages Specification, Version 1.1』は、他のタイプの構造化されたテキスト・ベースのドキュメント出力もサポートしています。JSP トランスレータでは、JSP 要素の外部のテキストは処理されないため、一般に Web ページに適したテキストは JSP ページにも適しています。

JSP ページは、HTTP 要求から情報を取り、データ・サーバーからの情報に (SQL データベースの問合せなどを介して) アクセスします。この情報が組み合されて処理され、動的コンテンツを持つ HTTP 応答に適切に組み込まれます。コンテンツの形式には、HTML、DHTML、XHTML または XML などを使用できます。

XML サポートの詳細は、5-8 ページの「[OracleJSP の XML および XSL サポート](#)」を参照してください。

JSP の実行

この項では、オンデマンド変換（JSP ページの初回実行時）、JSP コンテナとサーブレット・コンテナのロールおよびエラー処理など、JSP の実行方法の概要について説明します。

注意： JSP コンテナという用語は、Sun Microsystems の『JavaServer Pages Specification, Version 1.1』で、旧バージョンの仕様に使用されていた JSP エンジンという用語のかわりに使用されています。この 2 つの用語はシノニムです。

JSP コンテナの概要

JSP コンテナは、JSP ページを変換、実行および処理し、JSP ページに対する要求を配信するエンティティです。

JSP コンテナの正確な構成は実装ごとに異なりますが、サーブレットまたはその集合で構成されています。したがって、JSP コンテナはサーブレット・コンテナにより実行されます。（サーブレット・コンテナの概要は、B-3 ページの「[サーブレット・コンテナ](#)」を参照してください。）

Web サーバーが Java で記述されている場合は、JSP コンテナを Web サーバーに取り込むことができます。また、他の方法でコンテナを Web サーバーに関連付けて使用することも可能です。

JSP ページとオンデマンド変換

典型的なオンデマンド変換の使用例では、通常、JSP ページは次のステップで実行されます。

1. ユーザーが URL の末尾に `.jsp` ファイル名を指定して JSP ページを要求します。
2. URL の `.jsp` ファイル名拡張子に基づいて、Web サーバーのサーブレット・コンテナにより JSP コンテナが起動されます。
3. JSP コンテナにより JSP ページの位置が特定され、初めて要求された場合は変換されます。変換処理により、`.java` ファイル内でサーブレット・コードが生成され、`.java` ファイルがコンパイルされて、サーブレットの `.class` ファイルが生成されます。

JSP トランスレータにより生成されたサーブレット・クラスは、`javax.servlet.jsp.HttpJspPage` インタフェース（B-11 ページの「[標準 JSP インタフェースおよびメソッド](#)」を参照）を実装するクラス（JSP コンテナから提供されるクラス）がサブクラス化されます。サーブレット・クラスは、ページ実装クラスと呼ばれます。このマニュアルでは、ページ実装クラスのインスタンスを JSP ページ・インスタンスと呼びます。

JSP ページからサーブレットへの変換により、`javax.servlet.jsp.HttpJspPage` インタフェースの実装やサービス・メソッドのコードの生成など、標準的なサーブレッ

トのプログラミング・オーバーヘッドが、生成後のサーブレット・コードに自動的に取り込まれます。

4. JSP コンテナにより、ページ実装クラスのインスタンス化と実行がトリガーされます。

サーブレット（JSP ページ・インスタンス）により HTTP 要求が処理され、HTTP 応答が作成され、応答がクライアントに渡されます。

注意： 前述のステップは、概要を示しています。前述したように、各ベンダーが JSP コンテナの実装方法を決定していますが、いずれもサーブレットまたはその集合で構成されています。たとえば、JSP ページの位置を特定するフロントエンド・サーブレット、変換とコンパイルを処理する変換サーブレット、各ページ実装クラスによりサブクラス化される（変換後のページは純粋なサーブレットではなく、サーブレット・コンテナでは直接実行できないため）ラッパー・サーブレット・クラスなどがあります。これらの各コンポーネントを実行するには、サーブレット・コンテナが必要です。

JSP ページの要求

JSP ページは、URL を介して直接要求する方法と、他の Web ページやサーブレットを介して間接的に要求する方法があります。

JSP ページの直接要求

サーブレットや HTML ページの場合と同様に、エンド・ユーザーは URL を指定して JSP ページを直接要求できます。たとえば、次のように、Web サーバー上の myapp アプリケーションのルート・ディレクトリに、HelloWorld という JSP ページがあるとします。

```
myapp/dir1/HelloWorld.jsp
```

Web サーバーのポート 8080 が使用される場合は、次の URL を指定して要求できます。

```
http://hostname:8080/myapp/dir1/HelloWorld.jsp
```

（アプリケーションのルート・ディレクトリは、そのアプリケーションのサーブレット・コンテキストに指定します。サーブレット・コンテキストの概要は、B-6 ページの「[サーブレット・コンテキスト](#)」を参照してください。）

エンド・ユーザーが HelloWorld.jsp を初めて要求すると、JSP コンテナにより、そのページの変換と実行の両方がトリガーされます。2 度目以降の要求では、JSP コンテナによりページ実行のみがトリガーされ、変換ステップは不要になります。

JSP ページの間接要求

サーブレットと同様に、JSP ページも間接的に実行できます。つまり、標準の HTML ページからリンクするか、他の JSP ページまたはサーブレットから参照できます。

ある JSP ページを別の JSP ページの JSP 文から起動する場合は、アプリケーションのルートに対する相対パス（コンテキスト相対パスまたはアプリケーション相対パス）、または起動するページに対する相対パス（ページ相対パス）を指定できます。アプリケーション相対パスは「/」で始まりますが、ページ相対パスの先頭には「/」を使用しません。

通常、これらのパスは、URL や HTML リンクに使用されるパスとは異なるので注意してください。前項の例では、HTML リンクのパスは、次のように直接 URL 要求の場合と同じです。

```
<a href="/myapp/dir1/HelloWorld.jsp" /a>
```

JSP 文でのアプリケーション相対パスは、次のようになります。

```
<jsp:include page="/dir1/HelloWorld.jsp" flush="true" />
```

同じディレクトリ内の JSP ページから HelloWorld.jsp を起動するページの相対パスは、次のようになります。

```
<jsp:forward page="HelloWorld.jsp" />
```

(jsp:include および jsp:forward 文については、1-16 ページの「[JSP のアクションと <jsp:> タグ・セット](#)」を参照してください。)

JSP 構文要素の概要

JSP 構文の単純な例については、1-2 ページの「[JSP ページの内容](#)」で説明しました。ここでは、構文のカテゴリとトピックの概要を説明します。

- ディレクティブ。JSP ページ全体に関する情報を伝えます。
- スクリプト要素。宣言、式、スクリプトレットおよびコメントなどの Java コーディング要素です。
- オブジェクトと有効範囲。JSP オブジェクトを明示的または暗黙的に作成し、JSP ページまたはセッションのどこかなど、指定された有効範囲内でアクセスできます。
- アクション。オブジェクトを作成するか、JSP 応答での出力ストリームに影響します（またはその両方）。

この項では、各カテゴリについて基本構文と例を挙げて説明します。詳細は、Sun Microsystems の『JavaServer Pages Specification, Version 1.1』を参照してください。

注意： JSP のディレクティブ、宣言、式およびスクリプトレットの構文には、XML 互換の代替構文があります。5-9 ページの「[XML の代替構文](#)」を参照してください。

ディレクティブ

ディレクティブは、JSP ページ全体に関する指示を JSP コンテナに提供します。この情報は、そのページの変換または実行に使用されます。基本構文は、次のとおりです。

```
<%@ directive attribute1="value1" attribute2="value2" ... %>
```

JSP 1.1 仕様でサポートされるディレクティブは、次のとおりです。

- **page** — このディレクティブを使用して、使用するスクリプト言語、拡張するクラス、インポートするパッケージ、使用するエラー・ページまたは JSP ページの出力バッファ・サイズなど、多数のページ依存属性を任意に指定します。次に例を示します。

```
<%@ page language="java" import="packages.mypackage" errorPage="boof.jsp" %>
```

また、JSP ページの出力バッファ・サイズを 20KB（デフォルトは 8KB）に設定する場合は、次のようになります。

```
<%@ page buffer="20kb" %>
```

ページをバッファ解除する場合は、次のようになります。

```
<%@ page buffer="none" %>
```

注意：

- エラー・ページを使用する JSP ページは、バッファ処理する必要があります。エラー・ページに転送すると、ブラウザに出力されるのではなくバッファが消去されます。
 - OracleJSP では、デフォルトの言語設定は java です。ただし、プログラミング時には、この値を明示的に設定することをお勧めします。
-

- **include** — このディレクティブを使用して、変換時に JSP ページに挿入するテキストまたはコードを含むリソースを指定します。そのためには、リソースのパスを JSP ページの URL 指定への相対パスとして指定します。

次に例を示します。

```
<%@ include file="/jsp/userinfo.jsp" %>
```

include ディレクティブでは、ページ相対ロケーションまたはコンテキスト相対ロケーションを指定できます（関連情報については、1-7 ページの「[JSP ページの要求](#)」を参照してください）。

注意：

- include ディレクティブは「静的挿入」と呼ばれ、後述する `jsp:include` アクションに似た性格を持っていますが、要求時ではなく JSP 変換時に有効になります。4-8 ページの「[静的挿入と動的挿入](#)」を参照してください。
 - include ディレクティブを使用できるのは、同じサーブレット・コンテキスト内のみです。
-

- **taglib** — このディレクティブを使用して、JSP ページに使用するカスタム JSP タグのライブラリを指定します。ベンダーは、独自のタグ・セットを使用して JSP の機能を拡張できます。このディレクティブは、タグ・ライブラリ記述ファイルのロケーションと、そのライブラリからのタグの使用を区別する接頭辞を示します。

次に例を示します。

```
<%@ taglib uri="/oracustomtags" prefix="oracust" %>
```

ページの後半では、ライブラリ内のタグのいずれかを使用する必要が生じるたびに、`oracust` 接頭辞を使用します（このライブラリにタグ `dbaseAccess` が含まれているとします）。

```
<oracust:dbaseAccess>
...
</oracust:dbaseAccess>
```

前述のように、この例では XML スタイルの開始タグと終了タグの構文を使用しています。

JSP のタグ・ライブラリとタグ・ライブラリ記述ファイルの概要は 1-21 ページの「[タグ・ライブラリ](#)」、詳細は第 7 章「[JSP のタグ・ライブラリと Oracle の JML タグ](#)」を参照してください。

スクリプト要素

JSP のスクリプト要素には、JSP ページに表示できる次のカテゴリ Java コード断片が含まれます。

- 宣言 — JSP ページに使用されるメソッドまたはメンバー変数を宣言する文です。

JSP 宣言では、メンバー変数またはメソッドを宣言する `<%!...%>` 宣言タグ内で標準の Java 構文が使用されます。これにより、生成されるサーブレット・コードには対応する宣言が含まれます。次に例を示します。

```
<%! double f1=0.0; %>
```

この例では、メンバー変数 `f1` を宣言しています。JSP トランスレータにより生成されるサーブレット・クラス・コードでは、`f1` はクラスのトップレベルで宣言されます。

注意： メンバー変数とは異なり、メソッド変数は後述のように JSP スクリプトレット内で宣言されます。

- 式 — 評価され、該当する場合は文字列値に変換され、ページ上の使用箇所に表示される Java の式です。

JSP の式は、末尾にセミコロンがなく、`<%=...%>` タグに含まれています。

次に例を示します。

```
<P><B> Today is <%= new java.util.Date() %>. Have a nice day! </B></P>
```

注意： `jsp:setProperty` 文中など、要求時属性内の JSP 式は、文字列値に変換する必要はありません。

- スクリプトレット — Java コードのうちページのマークアップ言語に混在する部分です。

スクリプトレット、つまりコード・フラグメントは、Java コードの行の一部または複数行で構成できます。たとえば、JSP ページの HTML コード内で使用すると、条件付きのブランチやループなどを設定できます。

JSP スクリプトレットは、標準の Java 構文を使用して `<%...%>` スクリプトレット・タグで囲まれています。

例 1:

```
<% if (pageBean.getNewName().equals("")) { %>
    I don't know you.
```

```
<% } else { %>
    Hello <%= pageBean.getNewName() %>.
<% } %>
```

3 行の JSP スクリプトレットが 2 行の HTML と混在しています（そのうちの 1 行には JSP 式が含まれており、セミコロンを必要としません）。JSP 構文では、HTML コードを `if` および `else` ブランチ内（スクリプトレットで指定された Java の大カッコ内）で条件付きで実行されるコードとして使用できるので注意してください。

前述の例 1 では、JavaBeans インスタンス `pageBean` の使用を想定しています。

例 2:

```
<% if (pageBean.getNewName().equals("")) { %>
    I don't know you.
    <% empmgr.unknownemployee();
} else { %>
    Hello <%= pageBean.getNewName() %>.
    <% empmgr.knownemployee();
} %>
```

この例では、さらにスクリプトレットに Java コードが追加されています。JavaBeans インスタンス `pageBean` の使用と、一部のオブジェクト `empmgr` が以前にインスタンス化されており、既知の従業員または未知の従業員に対して適切な機能を実行するメソッドがあるものと想定しています。

注意： メンバー変数とは異なり、メソッド変数を宣言するには次の例のような JSP スクリプトレットを使用します。

```
<% double f2=0.0; %>
```

このスクリプトレットは、メソッド変数 `f2` を宣言しています。JSP トランスレータにより生成されるサーブレット・クラス・コードでは、`f2` はサーブレットの `service` メソッド内の変数として宣言されます。

メンバー変数は、前述したように JSP 宣言内で宣言されます。

メソッド変数とメンバー変数の比較については、4-13 ページの「[メソッド変数宣言とメンバー変数宣言](#)」を参照してください。

- コメント — Java コードに埋め込まれるコメントと同様に、開発者が JSP コードに埋め込むコメントです。

コメントは、`<!--...-->` タグで囲まれています。HTML でのコメントとは異なり、ユーザーがページ・ソースを表示しても、これらのコメントは表示されません。

次に例を示します。

```
<%-- Execute the following branch if no user name is entered. --%>
```

JSP のオブジェクトと有効範囲

このマニュアルでは、JSP オブジェクトという用語は、JSP ページ内で宣言されるか、JSP ページからアクセスできる、Java クラス・インスタンスを指しています。JSP オブジェクトには、次の 2 種類があります。

- 明示的 — 明示的オブジェクトは、JSP ページのコード内で宣言および作成され、選択した `scope` 設定に従ってそのページと他のページからアクセスできます。

または：

- 暗黙的 — 暗黙的オブジェクトは、基礎となる JSP メカニズムにより作成され、特定のオブジェクト型の本来の `scope` 設定に従って JSP ページ内の Java スクリプトレットまたは式からアクセスできます。

有効範囲については、後述の「[オブジェクトの有効範囲](#)」を参照してください。

明示的オブジェクト

明示的オブジェクトは、通常は `jsp:useBean` アクション文で宣言され作成される JavaBeans インスタンスです。`jsp:useBean` 文や他のアクション文の説明は、1-16 ページの「[JSP のアクションと <jsp:> タグ・セット](#)」を参照してください。次にこの文の例を示します。

```
<jsp:useBean id="pageBean" class="mybeans.NameBean" scope="page" />
```

この文では、`mybeans` パッケージ内の `NameBean` クラスのインスタンス `pageBean` を定義しています。`scope` パラメータについては、次の「[オブジェクトの有効範囲](#)」を参照してください。

Java プログラム内で Java クラス・インスタンスを作成する場合と同様に、Java スクリプトレットや宣言内でもオブジェクトを作成できます。

オブジェクトの有効範囲

明示的か暗黙的かに関係なく、JSP 内のオブジェクトには特定の有効範囲内でアクセスできます。`jsp:useBean` アクション文で作成される JavaBeans インスタンスなど、明示的オブジェクトの場合は、次の構文で（前述の「[明示的オブジェクト](#)」の例と同様に）有効範囲を明示的に設定できます。

```
scope="scopevalue"
```

次の 4 つの有効範囲が考えられます。

- `scope="page"` — オブジェクトには、作成元の JSP ページからのみアクセスできます。
ユーザーが JSP ページを実行中にリフレッシュすると、すべての `page` 有効範囲オブジェクトから新規インスタンスが作成されます。
- `scope="request"` — オブジェクトには、それを作成した JSP ページで処理される同じ HTTP 要求を処理する任意の JSP ページからアクセスできます。
- `scope="session"` — オブジェクトには、それを作成した JSP ページと同じ HTTP セッションを共有する任意の JSP ページからアクセスできます。
- `scope="application"` — オブジェクトには、それを作成した JSP ページと同じ Web アプリケーション（単一の Java 仮想マシン内）で使用される任意の JSP ページからアクセスできます。

暗黙的オブジェクト

JSP テクノロジーにより、どの JSP ページでも暗黙的オブジェクトのセットが使用可能になります。これは JSP メカニズムにより自動的に生成され、基礎となるサーブレット環境と相互作用できる Java クラス・インスタンスです。

次の暗黙的オブジェクトを使用できます。これらのオブジェクトで使用可能なメソッドの詳細は、次のロケーションにあるクラスとインタフェースの Sun Microsystems Javadoc を参照してください。

<http://java.sun.com/products/servlet/2.2/javadoc/index.html>

- `page`
これは、ページの変換時に作成された JSP ページ実装クラスのインスタンスです。
`page` は、JSP ページ内の `this` のシノニムです。
- `request`
これは、HTTP 要求を表し、`javax.servlet.ServletRequest` インタフェースに拡張される `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です。
- `response`
これは、HTTP 応答を表し、`javax.servlet.ServletResponse` インタフェースに拡張される `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です。

特定の要求の `response` および `request` オブジェクトは、相互に対応付けられています。

- `pageContext`

これは、JSP ページ・インスタンスのすべての `page` 有効範囲オブジェクトの格納とアクセスのために提供される、JSP ページのページ・コンテキストを表します。

`pageContext` オブジェクトは、`javax.servlet.jsp.PageContext` クラスのインスタンスです。

`pageContext` オブジェクトは `page` 有効範囲を持ち、アクセスできるのは対応付けられている JSP ページ・インスタンスのみです。

- `session`

これは HTTP セッションを表し、`javax.servlet.http.HttpSession` クラスのインスタンスです。

- `application`

これは、Web アプリケーションのサーブレット・コンテキストを表し、`javax.servlet.ServletContext` クラスのインスタンスです。

`application` オブジェクトには、単一 JVM 内のアプリケーションのいずれかのインスタンスの一部として実行される任意の JSP ページ・インスタンスからアクセスできます。（プログラマは、JVM の使用に関してサーバー・アーキテクチャに注意する必要があります。たとえば、Oracle Servlet Engine アーキテクチャでは、各ユーザーは自分の JVM で実行します。）

- `out`

これは、JSP ページ・インスタンスの出力ストリームにコンテンツを書き込むために使用されるオブジェクトです。また、`java.io.Writer` クラスに拡張される `javax.servlet.jsp.JspWriter` クラスのインスタンスです。

`out` オブジェクトは、特定の要求の `response` オブジェクトに対応付けられます。

- `config`

これは JSP ページのサーブレット構成を表し、`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（通常、サーブレット・コンテナは、`ServletConfig` インスタンスを使用して初期化中にサーブレットに情報を提供します。この情報の一部は、該当する `ServletContext` インスタンスです。）

- `exception` (JSP エラー・ページのみ)

この暗黙的オブジェクトは、JSP エラー・ページにのみ適用されます。JSP エラー・ページは、他の JSP ページからの例外発生時に処理が転送されるページであり、`page` ディレクティブの `isErrorPage` 属性を `true` に設定する必要があります。

暗黙的 `exception` オブジェクトは、他の JSP ページから発生した `uncaught` 例外を表し、このエラー・ページを起動させる `java.lang.Exception` インスタンスです。

`exception` オブジェクトにアクセスできるのは、例外の発生時に処理が転送された JSP エラー・ページ・インスタンスからのみです。

JSP エラー処理と `exception` オブジェクトの使用の例については、3-16 ページの「[JSP のランタイム・エラー処理](#)」を参照してください。

暗黙的オブジェクトの使用

前述の暗黙的オブジェクトは、いずれも役立ちます。次の例では、`request` オブジェクトを使用して HTTP 要求から `username` パラメータの値を取得し、表示します。

```
<H3> Welcome <%= request.getParameter("username") %> ! </H3>
```

JSP のアクションと `<jsp: >` タグ・セット

JSP アクション要素により、JSP ページの実行中にある種のアクションが発生します。たとえば、Java オブジェクトがインスタンス化されたり、そのページで使用可能になります。これには次のようなアクションが含まれます。

- JavaBeans インスタンスの作成とそのプロパティへのアクセス
- 他の HTML ページ、JSP ページまたはサーブレットへの実行の転送
- JSP ページへの外部リソースの挿入

アクション要素には、`<jsp:` 構文で始まる標準 JSP タグ・セットが使用されます。JSP ページをコーディングするには、前述の `<%` 構文で始まるタグで十分ですが、`<jsp:` タグには他にも機能があって役立ちます。

また、アクション要素には XML 文と同じ構文が使用され、次の例のように同様の「開始」タグと「終了」タグが使用されます。

```
<jsp:sampletag attr1="value1" attr2="value2" ... attrN="valueN">
...body...
</jsp:sampletag>
```

また、本体がない場合、アクション文は次のように空のタグで終了します。

```
<jsp:sampletag attr1="value1", ..., attrN="valueN" />
```

JSP 仕様には次の標準アクション・タグが含まれています。ここでは、各タグを簡単に説明します。

- `jsp:useBean`

`jsp:useBean` アクションでは、指定した JavaBeans クラスのインスタンスが作成され、指定した名前が与えられ、アクセス可能な有効範囲（現行の JSP ページ・インスタンス内の任意の場所など）が定義されます。

次に例を示します。

```
<jsp:useBean id="pageBean" class="mybeans.NameBean" scope="page" />
```

この例では、mybeans.NameBean クラスの page 有効範囲を持つインスタンス pageBean が作成されます。このインスタンスにアクセスできるのは、それを作成した JSP ページ・インスタンスのみです。

■ jsp:setProperty

jsp:setProperty アクションでは、1 つ以上の Bean プロパティが設定されます。(Bean は、useBean アクションで事前に指定する必要があります。) 特定のプロパティの値を直接指定する方法、対応する HTTP 要求のパラメータから取り込む方法、HTTP 要求のパラメータから一連のプロパティと値を反復する方法があります。

次の例では、pageBean インスタンス（前述の useBean の例で定義）の user プロパティを値「Smith」に設定しています。

```
<jsp:setProperty name="pageBean" property="user" value="Smith" />
```

次の例では、pageBean インスタンスの user プロパティを、HTTP 要求内のパラメータ username の設定値に従って設定しています。

```
<jsp:setProperty name="pageBean" property="user" param="username" />
```

また、Bean のプロパティと要求パラメータが同じ名前（user）の場合は、単にプロパティを次のように設定できます。

```
<jsp:setProperty name="pageBean" property="user" />
```

次の例では、HTTP 要求パラメータが反復され、Bean のプロパティ名が要求のパラメータ名と照合され、Bean のプロパティ値が対応する要求のパラメータ値に従って設定されます。

```
<jsp:setProperty name="pageBean" property="*" />
```

■ jsp:getProperty

jsp:getProperty アクションでは、Bean のプロパティ値が読み込まれ、Java 文字列に変換され、文字列値が出力として表示できるように暗黙的な out オブジェクトに置かれます。(Bean は、jsp:useBean アクションで事前に指定する必要があります。) 文字列変換の場合、プリミティブ型は直接変換され、オブジェクト型は java.lang.Object クラスで指定した toString() メソッドを使用して変換されます。

次の例では、pageBean Bean の user プロパティの値を out オブジェクトに置いています。

```
<jsp:getProperty name="pageBean" property="user" />
```

■ jsp:param

jsp:param アクションは、jsp:include、jsp:forward または jsp:plugin アクション（後述）とともに使用できます。

`jsp:forward` および `jsp:include` 文の場合、`jsp:param` アクションでは必要に応じて HTTP 要求オブジェクト内のパラメータ値にキーおよび値のペアが提供されます。このアクションで指定した新規のパラメータと値は要求オブジェクトに追加され、新しい値が古い値より優先されます。

次の例では、要求オブジェクトのパラメータ `username` を `Smith` の値に設定しています。

```
<jsp:param name="username" value="Smith" />
```

注意： `jsp:param` タグは、JSP 1.0 仕様での `jsp:include` または `jsp:forward` についてはサポートされません。

■ `jsp:include`

`jsp:include` アクションでは、ページが表示されているときの要求時に、追加の静的または動的リソースがページに挿入されます。リソースの指定には、相対 URL（ページ相対またはアプリケーション相対）を使用します。

Sun Microsystems の『JavaServer Pages Specification, Version 1.1』により、`jsp:include` アクションの実行時にバッファがブラウザにフラッシュされるように、`flush` を `true` に設定する必要があります。（`flush` 属性は必須ですが、`false` の設定は現時点では無効です。）

また、2 番目の例のように、アクション本体に `jsp:param` 設定を使用することもできます。

次に例を示します。

```
<jsp:include page="/templates/userinfopage.jsp" flush="true" />
```

または：

```
<jsp:include page="/templates/userinfopage.jsp" flush="true" >
  <jsp:param name="username" value="Smith" />
  <jsp:param name="userempno" value="9876" />
</jsp:include>
```

次の構文は、前述の例の代替として機能することに注意してください。

```
<jsp:include page="/templates/userinfopage.jsp?username=Smith&userempno=9876" flush="true" />
```

注意：

- `jsp:include` アクションは「動的挿入」と呼ばれ、前述の `include` ディレクティブに性質が似ていますが、変換時ではなく要求時に有効になります。4-8 ページの「静的挿入と動的挿入」を参照してください。
 - `jsp:include` アクションを使用できるのは、同じサーブレット・コンテキスト内のみです。
-

- `jsp:forward`

`jsp:forward` アクションでは、カレント・ページの実行が効率的に終了し、その出力が破棄され、HTML ページ、JSP ページまたはサーブレットなどの新規ページがディスパッチされます。

`jsp:forward` アクションを使用するには、JSP ページをバッファリングする必要があります（`buffer="none"` には設定できません）。このアクションによりバッファが消去されます（コンテンツはブラウザに出力されません）。

`jsp:include` と同様に、2 番目の例のようにアクション本体に `jsp:param` 設定を使用することもできます。

次に例を示します。

```
<jsp:forward page="/templates/userinfopage.jsp" />
```

または：

```
<jsp:forward page="/templates/userinfopage.jsp" >
  <jsp:param name="username" value="Smith" />
  <jsp:param name="userempno" value="9876" />
</jsp:forward>
```

注意：

- この `jsp:forward` の例と前述の `jsp:include` の例の違いは、`jsp:include` の例では `userinfopage.jsp` がカレント・ページの出力に挿入されるのに対して、`jsp:forward` の例ではカレント・ページの実行が中止され、かわりに `userinfopage.jsp` が表示されることです。
 - `jsp:forward` アクションを使用できるのは、同じサーブレット・コンテキスト内のみです。
-

- `jsp:plugin`

`jsp:plugin` アクションでは、必要に応じて Java プラグイン・ソフトウェアがダウンロードされてから、指定したアプレットまたは JavaBeans がクライアント・ブラウザで実行されます。

`jsp:plugin` 属性を使用して、実行するアプレットやコードベースなどの構成情報を指定します。JSP コンテナからダウンロード用のデフォルト URL が提供される場合がありますが、属性 `nspluginurl="url"` (Netscape ブラウザ用) または `iepluginurl="url"` (Internet Explorer ブラウザ用) を指定することもできます。

アプレットまたは JavaBeans のパラメータを指定するには、開始タグ `<jsp:params>` と終了タグ `</jsp:params>` の間でネストした `jsp:param` アクションを使用します。(`jsp:include` または `jsp:forward` アクションに `jsp:param` を使用する場合には、これらの `jsp:params` 開始および終了タグは不要なので注意してください。)

`plugin` を実行できない場合に実行する代替テキストを区切るには、開始タグ `<jsp:fallback>` と終了タグ `</jsp:fallback>` を使用します。

次の例は Sun Microsystems の『JavaServer Pages Specification, Version 1.1』から抜粋したもので、アプレット・プラグインの使用法を示しています。

```
<jsp:plugin type=applet code="Molecule.class" codebase="/html" >
  <jsp:params>
    <jsp:param name="molecule" value="molecules/benzene.mol" />
  </jsp:params>
  <jsp:fallback>
    <p> Unable to start the plugin. </p>
  </jsp:fallback>
</jsp:plugin>
```

その他、`jsp:plugin` アクション文には、ARCHIVE、HEIGHT、NAME、TITLE および WIDTH など、多数のパラメータを使用できます。これらのパラメータの使用は、一般の HTML 仕様に準拠しています。

タグ・ライブラリ

前述した標準 JSP タグの他に、JSP 仕様ではベンダーが独自のタグ・ライブラリを定義できます。また、ベンダーは、顧客が独自のタグ・ライブラリを定義できるようにフレームワークを実装できます。

タグ・ライブラリではカスタム・タグのコレクションが定義されており、JSP のサブ言語と見なすことができます。開発者は、JSP ページを手動でコーディングするときにタグ・ライブラリを直接使用できますが、Java 開発ツールでも自動的に使用される場合があります。タグ・ライブラリは、様々な JSP コンテナ実装間で移植可能である必要があります。

1-9 ページの「[ディレクティブ](#)」で紹介した `taglib` ディレクティブを使用して、タグ・ライブラリを JSP ページにインポートします。

JSP タグ・ライブラリに対する標準的な JavaServer Pages サポートの主要概念には、次のトピックが含まれています。

- タグ・ハンドラ

タグ・ハンドラでは、カスタム・タグの使用によって生じるアクションのセマンティックが記述されます。タグ・ハンドラは、標準 `javax.servlet.jsp.tagext` パッケージ内で `Tag` または `BodyTag` インタフェースを実装する Java クラスのインスタンスです（どちらを実装するかは、開始タグと終了タグの間に本体を使用するかどうかによって決まります）。

- スクリプト変数

カスタム・タグ・アクションでは、タグ自体またはスクリプトレットのような他のスクリプト要素に使用可能なサーバー側のオブジェクトを作成できます。そのためには、スクリプト変数を作成または更新します。

カスタム・タグで定義するスクリプト変数の詳細は、標準の `javax.servlet.jsp.tagext.TagExtraInfo` 抽象クラスのサブクラスで指定する必要があります。このドキュメントでは、このようなサブクラスを `Tag-Extra-Info` クラスと呼びます。JSP コンテナでは、このクラスのインスタンスが変換中に使用されます。

- タグ・ライブラリ記述ファイル

タグ・ライブラリ記述（TLD）ファイルは、タグ・ライブラリとライブラリ内の個々のタグに関する情報を含む XML ドキュメントです。TLD のファイル名には、`.tld` 拡張子が付いています。

JSP コンテナでは、TLD ファイルを使用して、ライブラリからのタグが発生したときに実行するアクションが決定されます。

- タグ・ライブラリ用の `web.xml` の使用

Sun Microsystems の『[Java Servlet Specification, Version 2.2](#)』には、サーブレット用の標準ディプロイメント・ディスクリプタ、`web.xml` ファイルが記述されています。JSP アプリケーションは、このファイルを使用して、JSP タグ・ライブラリ記述ファイルのロケーションを指定できます。

JSP タグ・ライブラリの場合、web.xml ファイルには taglib 要素と taglib-uri および taglib-location という 2 つの副要素を挿入できます。

これらのトピックの詳細は、7-2 ページの「[標準タグ・ライブラリのフレームワーク](#)」を参照してください。

OracleJSP で提供されるサンプル・タグ・ライブラリの詳細は、7-18 ページの「[JSP マークアップ言語 \(JML\) のサンプル・タグ・ライブラリの概要](#)」を参照してください。

詳細は、Sun Microsystems の『JavaServer Pages Specification, Version 1.1』を参照してください。

Oracle の JSP 実装の概要

Oracle8i リリース 8.1.7 (OracleJSP 1.1.0.0.0) では、JavaServer Pages の Oracle 実装は Sun Microsystems の『JavaServer Pages Specification, Version 1.1』の全面的な実装です。

この章では、OracleJSP の機能と、様々な環境、特に Oracle Servlet Engine (OSE) での OracleJSP のサポートについて説明します。OSE は、Oracle8i JVM で提供される Web サーバー（正確にはサーブレット・コンテナ）です。

標準の JavaServer Pages 機能の概要は、[第 1 章「概要」](#)を参照してください。

説明するトピックは、次のとおりです。

- [サーブレット環境間の移植性と機能性](#)
- [Oracle 環境における OracleJSP のサポート](#)
- [非 Oracle 環境における OracleJSP のサポート](#)
- [OracleJSP のプログラム拡張機能の概要](#)
- [OracleJSP のリリースと機能セットの概要](#)
- [OracleJSP の実行モデル](#)
- [OracleJSP に対する Oracle JDeveloper のサポート](#)

サーブレット環境間の移植性と機能性

Oracle の JavaServer Pages 実装は、サーバー・プラットフォームやサーブレット環境間での高度な移植性を備えています。また、サーブレット・コンテキストの動作が十分に定義されていなかった従来のサーブレット環境における、Web アプリケーションのフレームワークも提供します。

OracleJSP の移植性

OracleJSP は、Sun Microsystems の Java サーブレット仕様バージョン 2.0 以上に準拠する、すべてのサーブレット環境で実行できます。これに対して、ほとんどの JSP 実装には、サーブレット 2.1 (b) 以上の実装が必要です。次の項で説明しますが、OracleJSP は、従来のサーブレット環境にはなかったサーブレット機能に相当する機能性を備えています。

さらに、OracleJSP コンテナはサーバー環境とそのサーブレット実装から独立しています。これは、自社の JSP 実装を、スタンドアロン製品としてではなくサーブレット実装の一部として出荷するベンダーとは対照的です。

この移植性により、サーバーまたはサーブレット・プラットフォームの制限事項のために、開発システム上で様々な JSP 実装を使用する必要はなく、OracleJSP は開発環境でもターゲット環境でも容易に実行できます。通常、ターゲット・サーバーと同じ JSP コンテナを持つシステム上で開発すると利点が得られますが、現実には環境間になんらかの相違があるのが普通です。

サーブレット 2.0 環境向けの OracleJSP の拡張機能

サーブレットの仕様と JSP の機能性の相互依存性により、Sun Microsystems は JavaServer Pages 仕様のバージョンを Java サーブレット仕様のバージョンと連動させています。Sun Microsystems によれば、JSP 1.0 にはサーブレット 2.1 (b) の実装、JSP 1.1 にはサーブレット 2.2 の実装が必要です。

サーブレット 2.0 仕様には、サーブレット・コンテキストがアプリケーションごとに提供されるのではなく、Java 仮想マシンごとに 1 つしか提供されないという制限事項がありました。サーブレット 2.1 仕様では、アプリケーションごとに別個のサーブレット・コンテキストが可能になりましたが、必須ではありませんでした。サーブレット 2.1 (b) とサーブレット 2.2 の仕様では、別個のサーブレット・コンテキストが必須となっています。(サーブレットとサーブレット・コンテキストの背景情報は、B-2 ページの「[サーブレットの背景情報](#)」を参照してください。)

ただし、OracleJSP コンテナは、サーブレット 2.1 (b) 仕様で提供されたアプリケーション・サポートをエミュレートする機能性を備えています。これにより、Apache/JServ などのサーブレット 2.0 環境に十分なアプリケーション・フレームワークが得られます。これには、個別の ServletContext および HttpSession オブジェクトを持つアプリケーションの提供が含まれます。

この拡張サポートはファイル `globals.jsa` を介して提供され、このファイルは JSP アプリケーション・マーカー、アプリケーションおよびセッション・イベント・ハンドラ、および

アプリケーションのグローバル宣言とディレクティブの一元的ロケーションとして機能します。（詳細は、5-35 ページの「[globals.jsa の機能の概要](#)」を参照してください。）

この拡張された機能性により、OracleJSP は基礎となるサーブレット環境の制限を受けません。

Oracle 環境における OracleJSP のサポート

ここでは、OracleJSP をサポートおよび提供する Oracle 環境の概要について、次の各トピックで説明します。

- [Oracle Servlet Engine \(OSE\) の概要](#)
- [Oracle Internet Application Server の概要](#)
- [Apache により駆動される Oracle HTTP Server のロール](#)
- [Oracle Web Application のデータベース・アクセス方法](#)
- [他の Oracle JSP 環境の概要](#)

Oracle Servlet Engine (OSE) は、Oracle8i データベース内で実行される Web サーバーおよびサーブレットのコンテナであり、JSP の事前変換モデルをサポートします。JSP ページは、データベースへの配布前または配布中にサーブレットに変換され、その後はデータベースのアドレス空間で実行されます。

他の Oracle 環境では、OracleJSP コンテナは代表的なオンデマンド変換モデルをサポートし、通常はページを実行時に変換します。OracleJSP は、選択したサーバーに関係なく、どの状況でも効率的に動作して一貫したセマンティクスを提供するように設計されています。

Oracle Servlet Engine (OSE) の概要

JSP ページから Oracle8i データベースへのアクセスを意図している場合は、JSP ページをデータベース内で直接実行できます。Oracle8i JVM に組み込まれている Oracle Servlet Engine (OSE) により、OracleJSP コンテナが取り込まれます。このため、中間層の JSP 実行に比べて通信オーバーヘッドが削減されます。データベースへのアクセスには、Oracle JDBC のサーバー側内部ドライバが使用されます。

OSE 実行モデルでは、開発者はなんらかの特殊なステップに従って JSP ページを Oracle8i データベースに配布する必要があります。このステップには、ページの変換、サーバーへのロードおよび実行を可能にするための「公開」が含まれます。

Oracle8i リリース 8.1.7 のインストール中に、Apache により駆動される Oracle HTTP Server がデフォルトの Web サーバーとして設定され、OSE で実行される JSP およびサーブレット・アプリケーションのフロントエンドとして機能します。この設定の変更が必要な場合は、インストール手順を参照してください。

Oracle Servlet Engine リリース 8.1.7 は、サーブレット 2.2 および JSP 1.1 仕様をサポートしており、OracleJSP リリース 8.1.7 (1.1.0.0.0) を取り込んでいます。

Oracle Internet Application Server の概要

Oracle Internet Application Server は、スケーラブルでセキュリティ機能を備えた中間層のアプリケーション・サーバーです。Web コンテンツの配信、Web アプリケーションのホスト処理、バック・オフィス・アプリケーションへの接続および、これらのサービスにどのクライアント・ブラウザからもアクセスできるようにする処理に使用できます。ユーザーは、インターネット、社内イントラネットまたはエクストラネット上で情報にアクセスし、ビジネス分析を実行して、ビジネス・アプリケーションを実行できます。

この範囲のコンテンツとサービスを配信するために、Oracle Internet Application Server リリース 1.0.x は、Oracle HTTP Server リリース 1.0.0 (Apache により駆動)、中間層におけるデータベース・キャッシュとアプリケーションのための iCache、Oracle Forms ベースのアプリケーションとレポート生成をサポートする Oracle Forms Services と Oracle Reports Services、および Enterprise JavaBeans、ストアド・プロシージャ、Oracle Business Components for Java をサポートする各種ビジネス・ロジック・ランタイム環境を取り込みます。

データベースへのアクセス用に、Oracle HTTP Server は HTTP 要求を次の使用例のどちらかで実行されているサーブレットまたは JSP ページにルーティングできます。

- Oracle8i で直接 (ルーティングは Apache の mod_ose モジュールを経由)
この使用例では、データベース・アクセスはサーバー側 JDBC 内部ドライバ (JDBC または SQLJ コードを使用) を経由します。
- Apache/JServ 環境で (ルーティングは Apache の mod_jserv モジュールを経由)
この使用例では、データベース・アクセスはクライアント側 / 中間層の JDBC ドライバ (JDBC または SQLJ コードを使用) を経由します。

Oracle Internet Application Server リリース 1.0.x は、サーブレットおよび JSP 環境を次のように提供します。

- リリース 1.0.0 と 1.0.1 には、サーブレット 2.0 仕様をサポートする Apache/JServ サーブレット環境が組み込まれています。
- リリース 1.0.0 には、JSP 1.0 仕様をサポートする OracleJSP リリース 1.0.0.6.1 が組み込まれています。
- リリース 1.0.1 には、JSP 1.1 仕様をサポートする OracleJSP リリース 1.1.0.0.0 が組み込まれています。

注意： Oracle Internet Application Server の今後のリリースでは、Apache/JServ 環境が代替サーブレット環境に置き換えられる可能性があります。

Apache により駆動される Oracle HTTP Server のロール

Oracle Internet Application Server リリース 1.0.x と Oracle8i リリース 8.1.7 には、Apache Web サーバーで駆動される Oracle HTTP Server リリース 1.0.0 が、Oracle8i データベースにアクセスする Web アプリケーションの HTTP エントリ・ポイントとして組み込まれています。

Oracle HTTP Server を使用すると、データベースの内外で実行されるアプリケーションから Oracle8i にアクセスできます。Oracle HTTP Server は、適切な Apache アドオン・モジュールを介してデータベースにアクセスします。

以降は、次のトピックについて説明します。

- [Apache の mod の使用](#)
- [mod_ose の詳細](#)
- [mod_jserv の詳細](#)

注意：

- Oracle Servlet Engine 自体は Web サーバーとして機能できますが、特に静的 HTML を使用するアプリケーションの場合は、Oracle HTTP Server とともにサーブレット・コンテナとして使用することをお勧めします。mod_ose モジュールにより、Oracle HTTP Server を介して OSE にアクセスできます。
 - Oracle Internet Application Server のフレームワークでは、Oracle HTTP Server を使用して Oracle iCache（キャッシュされた読取り専用データ用）またはバックエンドの Oracle8i データベースにアクセスできます。ただし、Oracle Internet Application Server リリース 1.0.0 および 1.0.1 の場合、このリリースの iCache には Oracle Servlet Engine が組み込まれていないため、mod_ose/OSE の使用例は該当しません。
-

Apache の mod の使用

Apache により駆動される Oracle HTTP Server を使用すると、動的コンテンツは Apache やオラクル社のような他のベンダーから提供される各種の Apache mod コンポーネントを介して配信されます。（通常、静的コンテンツはファイル・システムから配信されます。）

Apache の mod は通常は C コードのモジュールで、Apache アドレス空間で実行され、特定

の mod 固有のプロセッサに要求を渡します。(mod ソフトウェアは、特定のプロセッサ専用 に記述されます。)

OracleJSP 開発者にとって重要な Apache mod は、次のとおりです。

- `mod_ose` は、Oracle8i データベースに配布された JSP ページおよびサーブレット用にオラクル社から提供され、データベースのアドレス空間で Oracle Servlet Engine により実行されます。
- `mod_jserv` は Apache から提供され、中間層 JVM の Apache/JServ サブレット環境で実行される JSP ページまたはサーブレットから Oracle8i データベースへのアクセスに使用できます。

注意： 他の多数の Apache 「mod」 コンポーネントが Apache 環境用に用意されています。これらは、Apache から汎用として提供されるか、オラクル社から Oracle 固有の用途にあわせて提供されますが、JSP アプリケーションには関係しません。

mod_ose の詳細

オラクル社が提供する `mod_ose` コンポーネントは、HTTP 要求を OSE で実行される JSP ページまたはサーブレットに委譲します。Net8 プロトコル経由で HTTP を使用して OSE と通信し、ステートレス要求またはステートフル要求を処理できます。Oracle HTTP Server 内で構成されている各仮想ドメインは、要求を実行するための接続先を示すデータベース接続文字列 (Net8 の名前 - 値リスト) に対応付けられています。この接続では Net8 が直接使用され、OCI と同じロード・バランシングおよびホット・バックアップ機能を提供します。

Oracle Internet Application Server フレームワーク内で実行されるアプリケーションで `mod_ose` が使用される場合、Oracle Internet Application Server の Apache/JServ サブレット 2.0 環境は関与しません。かわりに、Oracle Servlet Engine 自体のサーブレット 2.2 環境が使用されます。

OSE 内で実行される JSP アプリケーションおよびサーブレットでは、高速データベース・アクセスのために Oracle JDBC のサーバー側内部ドライバが使用されます。OSE の概要は、2-3 ページの「[Oracle Servlet Engine \(OSE\) の概要](#)」を参照してください。

Oracle8i JVM のセッション・シェルの `exportwebdomain` コマンドを使用すると、データベース内で公開されているサーブレットと JSP ページを検索するように `mod_ose` を構成できます。

`mod_ose` の詳細と `exportwebdomain` コマンドについては、『Oracle8i Oracle Servlet Engine ユーザーズ・ガイド』を参照してください。

mod_jserv の詳細

Apache が提供する `mod_jserv` コンポーネントは、HTTP 要求を中間層 JVM 内の Apache/JServ サブレット・コンテナ内で実行される JSP ページまたはサーブレットに委譲します。Oracle Internet Application Server リリース 1.0.x には、サーブレット 2.0 仕様を

サポートする Apache/JServ サブレット・コンテナと、JDK 1.1.8 または 1.2.2 が組み込まれています。中間層環境は、バックエンドの Oracle8i データベースと同じ物理ホスト上にあっても、他のホスト上にあってもかまいません。

mod_jserv と中間層 JVM との通信には、TCP/IP を介した所有権付きの Apache/JServ プロトコルが使用されます。mod_jserv コンポーネントは、ロード・バランシングのために要求をプール内の複数の JVM に委譲できます。

中間層 JVM で実行される JSP アプリケーションでは、データベースへのアクセスに Oracle JDBC OCI ドライバまたは Thin ドライバが使用されます。

サブレット 2.0 環境には（サブレット 2.1 または 2.2 環境とは異なり）、特に考慮を必要とする問題があります。4-30 ページの「[Apache/JServ サブレット環境に関する考慮事項](#)」を参照してください。

mod_jserv の構成情報については、Apache のドキュメントを参照してください。（このドキュメントは、Oracle8i および Oracle Internet Application Server とともに提供されます。）

Oracle Web Application のデータベース・アクセス方法

JSP アプリケーションから Oracle8i データベースをターゲットとする開発者には、次のように多数のオプションがあります。

1. Oracle HTTP Server を介して（mod_jserv を使用して）Apache/JServ サブレット・コンテナ内で実行します。
2. Oracle HTTP Server を介して（mod_ose を使用して）Oracle Servlet Engine 内で実行します。
3. Oracle Servlet Engine 内で実行し、Web サーバーとして直接使用します（ただし、通常は Oracle HTTP Server を使用することをお勧めします）。

注意： Oracle HTTP Server を使用する場合は、Apache/JServ サブレット・コンテナの静的ファイルのデフォルトのドキュメント・ルートが、Oracle Servlet Engine とは異なることに注意してください。6-63 ページの「[Oracle Internet Application Server と Oracle Servlet Engine のドキュメント・ルートの比較](#)」を参照してください。

JDBC OCI ドライバを使用する場合や、Oracle8i JVM 環境では使用できない Java 機能（JNI など）がアプリケーションに必要な場合は、標準 JVM（現在は JDK 1.2.2 または 1.1.8）が使用されるため、Apache/JServ で実行する必要があります。

ただし、Apache/JServ で実行する場合は、複数の JVM のプールが必要であり、手動で構成する必要があるというデメリットがあります。（詳細は、Oracle8i または Oracle Internet Application Server とともに提供される Apache mod_jserv のドキュメントを参照してください。）

JNI などの Java 機能を必要としない場合、特に SQL 集約型アプリケーションの場合、通常は次のような理由から OSE 内で実行することをお勧めします。

- OSE は、Oracle8i のアドレス空間で動作し、Oracle JDBC のサーバー側内部ドライバが使用されるため、データベースへのアクセスが高速になります。
- OSE の方が強力なセキュリティ機能を備えています。
- OSE の方がステートフルに対する強力なサポート機能を備えています。

Oracle Servlet Engine を Web サーバーとして直接使用することも可能であり、ある種の状況では望ましい方法ですが、代表的な使用例は Oracle HTTP Server と `mod_ose` を介してアクセスする方法であり、この方法をお勧めします。

特に、Oracle HTTP Server と `mod_ose` では、OSE 単独では処理できない次の状況进行处理できます。

- Net8 証明済みのファイアウォールを介したデータベース・アクセス
- 複数のデータベースを使用するフォールト・トレラント・システムの実装
- ポート 80 を介したデータベース・アクセス

通常、OSE を Web サーバーとして直接使用する場合、このデータベース・アクセスは不可能です。たとえば、UNIX 環境では、ポート 80 にアクセスできるのは root アカウントからのみで、エンド・ユーザーには root アクセス権がありません。

- セッション起動時のオーバーヘッドのほとんどを回避するための、ステートレス・アプリケーションの接続プーリング

Oracle8i リリース 8.1.7 のデフォルト・インストールでは、OSE 内で実行される JSP ページおよびサーブレットのフロントエンド Web サーバーとして Oracle HTTP Server が使用されます。

他の Oracle JSP 環境の概要

Oracle Servlet Engine と Oracle Internet Application Server の他に、次の Oracle 環境で OracleJSP がサポートされます。

- [Oracle Application Server](#)
- [Oracle Web-to-go](#)
- [Oracle JDeveloper](#)

Oracle Application Server

Oracle Application Server (OAS) は、アプリケーション・ロジックのためのスケーラブルで規格に準拠する中間層環境であり、社内環境と E-Business 環境でビジネス・アプリケーションをサポートできるようにデータベースを統合します。

新規ユーザーは、OAS のかわりに前述の Oracle Internet Application Server を使用することになります。ただし、既存の OAS ユーザーのために、Oracle Application Server リリース 4.0.8.2 にはサーブレット 2.1 環境と OracleJSP リリース 1.0.0.6.0 (JSP 1.0 仕様をサポート) が組み込まれています。

詳細は、『Oracle Application Server JServlet および JSP アプリケーション開発者ガイド』を参照してください。

Oracle Web-to-go

Oracle Web-to-go は Oracle8i Lite のコンポーネントであり、モバイル Web アプリケーションの開発、配布および管理を容易にするモジュールとサービスのコレクションで構成されています。

Oracle Web-to-go により、開発者は Web ベースのアプリケーションを常時接続していないユーザーへと拡張でき、レプリケーション、同期化および他のネットワーク関連の問題に必要なインフラストラクチャのコーディングは不要です。従来のモバイル・コンピューティング・テクノロジーはカスタムまたは所有権付きのアプリケーション・プログラミング・インタフェース (API) に依存していましたが、Oracle Web-to-go には業界標準のインターネット・テクノロジーが採用されています。

Oracle Web-to-go リリース 1.3 は、サーブレット 2.1 環境と OracleJSP リリース 1.0.0.6.1 (JSP 1.0 仕様をサポート) を提供します。今後のリリースでは、サーブレット 2.2 環境と OracleJSP 1.1.x が提供される予定です。

詳細は、『Web-to-go インプリメンテーション・ガイド』を参照してください。

Oracle JDeveloper

JDeveloper は、前述の他の Oracle 製品のような「プラットフォーム」ではなく Java 開発ツールですが、Web リスナー、サーブレット・ランナーおよび OracleJSP コンテナを実行とテストのために取り込みます。

詳細は、2-20 ページの「[OracleJSP に対する Oracle JDeveloper のサポート](#)」を参照してください。

JDeveloper リリース 3.1 は、サーブレット 2.1 環境と OracleJSP リリース 1.0.0.6.1 (JSP 1.0 仕様をサポート) を提供します。今後のリリースでは、サーブレット 2.2 環境と OracleJSP 1.1.x が提供される予定です。

非 Oracle 環境における OracleJSP のサポート

サーブレット仕様 2.0 以上をサポートしていれば、どのサーバー環境にも OracleJSP コンテナをインストールして実行できるようにする必要があります。特に、OracleJSP は、リリース 8.1.7 現在で次の環境でのテストが完了しています。

- Apache Software Foundation Apache/JServ 1.1

これは、JSP 環境のない Web サーバーおよびサーブレット 2.0 環境です。JSP ページを実行するには、その最上位に JSP 環境をインストールする必要があります。

- Sun Microsystems JSWDK 1.0 (JavaServer Web Developer's Kit)

これは、サーブレット 2.1 と JavaServer Pages 1.0 のリファレンス実装を伴う Web サーバーです。ただし、JSWDK サーブレット環境の最上位に OracleJSP をインストールすると、オリジナルの JSP 環境を置き換えることができます。

- Apache Software Foundation Tomcat 3.1

この Sun Microsystems と Apache Software Foundation の共同作業の成果は、サーブレット 2.2 および JavaServer Pages 1.1 のリファレンス実装を伴う Web サーバーです。ただし、Tomcat サーブレット環境の最上位に OracleJSP をインストールすると、オリジナルの JSP 環境を置き換えることができます。また、Tomcat Web サーバーを使用するかわりに、Apache Web サーバーとともに Tomcat を実行することもできます。

OracleJSP のプログラム拡張機能の概要

この項では、OracleJSP でサポートされる拡張プログラミング機能の概要について説明します。

OracleJSP は、カスタム・タグ・ライブラリとカスタム JavaBeans を介して、次の拡張機能を提供します。これらの拡張機能は、いずれも他の JSP 環境に移植できます。

- 有効範囲を指定できる JavaBeans として実装される拡張データ型
- XML および XSL との統合
- Database-Access JavaBeans
- JSP 開発に要求される Java のスキル・レベルを軽減する、Oracle JSP マークアップ言語 (JML) のカスタム・タグ・ライブラリ
- SQL 機能向けのカスタム・タグ・ライブラリ

OracleJSP は、次の Oracle 固有の拡張機能も提供します。

- SQL 文を Java コードに直接埋め込むための標準構文である SQLJ のサポート
- 拡張 NLS サポート

- イベント処理用の `JspScopeListener`
- アプリケーション・サポート用の `globals.jsa` ファイル

各トピックについて説明した後で、OracleJSP ページと Oracle PL/SQL Server Pages との相互作用について簡潔に説明します。

移植可能な OracleJSP の拡張機能の概要

この項で説明する Oracle の拡張機能は、OracleJSP の JML サンプル・タグ・ライブラリまたはカスタム JavaBeans を介して実装されます。いずれも、他の JSP 環境に移植可能です。

OracleJSP の拡張データ型

通常、JSP ページは Java データ型に依存してスカラー型の値を表現しています。そのための次の標準的アプローチは、どちらも JSP ページでの使用には適していません。

- `int`、`float` および `double` などのプリミティブ型
- `Integer`、`Float` および `Double` など、標準的な `java.lang` パッケージ内のラッパー・クラス

プリミティブ型の値には、有効範囲を指定できません。有効範囲オブジェクトに格納できるのはオブジェクトのみのため、プリミティブ型の値は JSP の有効範囲オブジェクト (`page`、`request`、`session` または `application` 有効範囲など) に格納できません。

ラッパー型の値はオブジェクトなので、理論上は JSP の有効範囲オブジェクトに格納できます。ただし、ラッパー・クラスは JavaBeans モデルに準拠せず、引数 0 のコンストラクタを提供しないため、`jsp:useBean` アクションでは宣言できません。

また、ラッパー・クラスのインスタンスはイミュータブルです。値を変更するには、新規インスタンスを作成して適切に割り当てる必要があります。

これらの制限事項を回避するために、OracleJSP では、パッケージ `oracle.jsp.jml` に、`JmlBoolean`、`JmlNumber`、`JmlFPNumber` および `JmlString` の JavaBeans クラスが用意されており、ほとんどの一般的な Java データ型がラップされます。

詳細は、5-2 ページの「[JML の拡張データ型](#)」を参照してください。

XML および XSL との統合

JSP 構文を使用して、HTML コードのみでなくテキスト・ベースのあらゆる MIME タイプを生成できます。特に、XML 出力を動的に作成できます。ただし、JSP ページを使用して XML ドキュメントを作成する場合、通常はクライアントに送信する前に XML データにスタイルシートを適用する必要があります。JSP ページに使用される標準出力ストリームはサーバーを介して直接書き込まれるため、これは JavaServer Pages テクノロジーでは困難です。

OracleJSP のサンプル JML タグ・ライブラリには特殊なタグが用意されており、JSP ページ全体またはその一部を出力前に XSL スタイルシートを介して変換するように指定されます。ページの各部分に異なるスタイルシートを指定する必要がある場合は、この JML タグを単

一の JSP ページ内で必要な回数だけ使用できます。JML タグ・ライブラリは、他の JSP 環境に移植できることに注意してください。

また、OracleJSP トランスレータでは、Sun Microsystems の『JavaServer Pages Specification, Version 1.1』で指定されている XML 代替構文がサポートされます。

詳細は、5-8 ページの「[OracleJSP の XML および XSL サポート](#)」を参照してください。

カスタム Database-Access JavaBeans

OracleJSP は、Oracle データベースへのアクセスに使用するカスタム JavaBeans のセットを提供します。oracle.jsp.dbutil パッケージには、次の Bean が用意されています。

- ConnBean は、単一のデータベース接続をオープンします。
- ConnCacheBean は、データベース接続に Oracle の接続キャッシング実装を使用します。
- DBBean はデータベース問合せを実行します。
- CursorBean は、問合せのみでなく、UPDATE、INSERT および DELETE 文について一般的な DML サポートを提供します。

詳細は、5-12 ページの「[Oracle Database-Access JavaBeans](#)」を参照してください。

OracleJSP の SQL カスタム・タグ・ライブラリ

リリース 8.1.7 の OracleJSP には、SQL 機能のためのカスタム・タグ・ライブラリが用意されています。提供されるタグは、次のとおりです。

- dbOpen — データベース接続をオープンします。
- dbClose — データベース接続をクローズします。
- dbQuery — 問合せを実行します。
- dbCloseQuery — 問合せ用のカーソルをクローズします。
- dbNextRow — 結果セットの次行に移動します。
- dbExecute — 任意の SQL DML または DDL 文を実行します。

詳細は、5-22 ページの「[OracleJSP の SQL 用タグ・ライブラリ](#)」を参照してください。

Oracle JSP マークアップ言語（JML）のカスタム・タグ・ライブラリ

Sun Microsystems の『JavaServer Pages Specification, Version 1.1』は、Java 以外のスクリプト言語もサポートしていますが、Java は主流となっている言語であり、多くの場合は唯一の検討対象です。JavaServer Pages テクノロジは動的な Java 開発作業を静的な HTML 開発作業から分離するように設計されていますが、Web 開発者に Java の知識がない場合、特に、Java のエキスパートが参加していない小規模な開発グループでは、障害となることに変わりはありません。

OracleJSP は、代替となるカスタム・タグ、つまり JSP マークアップ言語 (JML) を提供します。Oracle JML のサンプル・タグ・ライブラリには、追加の JSP タグ・セットが含まれているため、Java の文を使用しなくても独自の JSP ページを記述できます。JML には、変数宣言、制御フロー、条件付きブランチ、反復ループ、パラメータ設定およびオブジェクトのコールに使用するタグがあります。

JML タグ・ライブラリでは、前述のように XML 機能もサポートされます。

次の例は、`jml:for` および `jml:print` タグの使用方法を示しています。

```
<jml:for id="i" from="1" to="5" >
  <H<jml:print eval="i" />
    Hello World!
  </H<jml:print eval="i" />
</jml:for>
```

詳細は、7-18 ページの「[JSP マークアップ言語 \(JML\) のサンプル・タグ・ライブラリの概要](#)」を参照してください。

注意： JSP 1.1 仕様以前の OracleJSP のバージョンには、JML タグ・ライブラリの Oracle 固有のコンパイル時実装が使用されていました。この実装は、標準ランタイム実装の代替として引き続きサポートされます。詳細は、[付録 C「コンパイル時 JML タグ・サポート」](#)を参照してください。

Oracle 固有の拡張機能の概要

この項で説明する OracleJSP の拡張機能は、他の JSP 環境には移植できません。

OracleJSP における SQLJ のサポート

一般に、動的サーバー・ページにはデータベースから抽出された情報が含まれていますが、JavaServer Pages テクノロジにはデータベース・アクセスを容易にするサポートは組み込まれていません。通常、JSP 開発者は、標準的な Java データベース接続 (JDBC) API またはカスタムのデータベース JavaBeans セットに依存する必要があります。

SQLJ は、静的な SQL 指示を Java コードに直接埋め込むための標準構文であり、データベース・アクセスのプログラミングを大幅に簡素化します。OracleJSP と OracleJSP トランスレータでは、JSP スクリプトレットでの SQLJ プログラミングがサポートされます。

SQLJ 文は、`#sql` トークンで示されます。JSP のソース・コード・ファイルのファイル拡張子 `.sqljsp` を使用すると、OracleJSP トランスレータをトリガーして Oracle SQLJ トランスレータを起動できます。

詳細は、5-31 ページの「[OracleJSP による Oracle SQLJ のサポート](#)」を参照してください。

OracleJSP における拡張 NLS サポート

OracleJSP は、マルチバイトの要求パラメータと Bean のプロパティ設定をコード化できないサーブレット環境に、拡張 NLS サポートを提供します。

このような環境向けに、OracleJSP には `translate_params` 構成パラメータが用意されています。このパラメータを使用可能にすると、OracleJSP に対して、サーブレット・コンテナをオーバーライドしてそれ自体をコード化するように指示できます。

詳細は、8-4 ページの「[マルチバイト・パラメータのコード化に対する OracleJSP の拡張サポート](#)」を参照してください。

イベント処理用の JspScopeListener

OracleJSP には、JSP アプリケーション内で様々な有効範囲の Java オブジェクトのライフ・サイクルを管理できるように、`JspScopeListener` インタフェースが用意されています。

標準的なサーブレットおよび JSP のイベント処理は

`javax.servlet.http.HttpSessionBindingListener` インタフェースを介して提供されますが、このインタフェースで処理されるのはセッション・ベースのイベントのみです。Oracle の `JspScopeListener` では、ページ・ベース、要求ベースおよびアプリケーション・ベースのイベントも処理できます。

詳細は、5-30 ページの「[OracleJSP のイベント処理 — JspScopeListener](#)」を参照してください。

アプリケーション・サポート用の globals.jsa ファイル（サーブレット 2.0）

サーブレット・コンテキストが十分に定義されていないサーブレット 2.0 環境では、OracleJSP により、サーブレット・アプリケーション・サポートを拡張するための `globals.jsa` ファイルが定義されます。

どの単一 Java 仮想マシン内でも、アプリケーションごと（つまりサーブレット・コンテキストごと）に `globals.jsa` ファイルを使用できます。このファイルでは、アプリケーション・ロケーション・マーカースとしての使用を通じて Web アプリケーションの概念がサポートされます。OracleJSP コンテナは、`globals.jsa` の機能性に基づいて、サーブレット環境に対するサーブレット・コンテキストと HTTP セッションの動作を疑似実行することもできます。これらの動作は、サーブレット環境では十分に定義されていません。

また、`globals.jsa` ファイルは、アプリケーションのすべての JSP ページ間で、グローバルな Java 宣言と JSP ディレクティブのための媒介を提供します。

OracleJSP と Oracle PL/SQL Server Pages の併用

オラクル社は、PL/SQL Server Pages (PSP) と呼ばれる製品を提供しています。PSP テクノロジーにより、PL/SQL スクリプトレットとストアード・プロシージャ・コールを HTML ページに埋め込むことができ、JSP テクノロジーの場合と同様の開発上のメリットが得られます。つまり、ページの動的部分と静的部分のコーディングについて、開発作業を大きく分離できます。HTML のエキスパートはページの静的部分をコーディングし、PL/SQL のエキスパートは動的部分をコーディングできます。PSP ページ内で PL/SQL スクリプトレットを区別するための構文は、JSP ページで Java スクリプトレットの区別に使用されるものと同じです。

これ以降は、JSP-PSP の相互作用のサポートと、PSP の URL に関する背景情報について説明します。

PL/SQL Server Pages の概要は、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。

JSP ページと PSP ページ間でサポートされる相互作用

エンド・ユーザーが PSP アプリケーションを実行すると、Web ブラウザへの出力の生成中に PL/SQL Gateway で実行できるように、PSP ページがストアード・プロシージャに変換されます。Oracle8i の PL/SQL Gateway はサーブレット・ラッパー内で動作するため、Oracle Servlet Engine 内で実行される JSP ページは PSP ページと次のように対話できます。

- JSP ページから PSP ページを動的に挿入できます (jsp:include)。
- JSP ページから PSP ページに動的に転送できます (jsp:forward)。

ただし、PSP ページには、JSP ページの動的挿入機能や JSP ページへの動的転送機能はありません。また、JSP ページから PSP ページを静的に挿入することはできません (変換中にファイルを挿入する `<%@ include %>` ディレクティブ)。

PSP ページの URL

各 PSP ページは、データベースにロードされてコンパイルされると、PL/SQL ストアド・プロシージャになります。PSP ページのストアード・プロシージャ名は、そのページ内で `<%@ plsql procedure="proc-name" %>` 構文を使用して明示的に宣言されるか、PSP ファイル名から導出されます。

PL/SQL ストアド・プロシージャ名に基づき、次の一般構文に従って URL が決定されます。

```
http://host[:port]/some-prefix/dad/[schema.]proc-name
```

`<some-prefix>` は埋込み PL/SQL モジュールの `plsql`、`<dad>` はストアード・プロシージャを実行するデータベース・アクセス記述子です。

詳細は、『Oracle8i アプリケーション管理者ガイド 基礎編』を参照してください。

OracleJSP のリリースと機能セットの概要

OracleJSP リリース 1.1.0.0.0 は、JSP 1.1 仕様を全面的にサポートしており、Oracle8i リリース 8.1.7 で提供されます。このマニュアルでは、「OracleJSP リリース 8.1.7」は「OracleJSP リリース 1.1.0.0.0」のシノニムです。

OracleJSP をサポートしている他の Oracle プラットフォームには、まだ最新の OracleJSP リリースを取り込んでいないものもありますが、JSP 1.0 の実装だった OracleJSP リリース 1.0.0.6.1 または 1.0.0.6.0 が統合されます。

Oracle プラットフォームで提供される OracleJSP のリリース

表 2-1 は、このマニュアルの発行時の Oracle プラットフォームのリリースと、そのプラットフォームで提供される OracleJSP のリリースを示しています。

「OracleJSP の機能上の注意」列は、このマニュアルに記載されている OracleJSP リリース 1.1.0.0.0 の機能のうち、特定の Oracle プラットフォーム向けの OracleJSP リリースに制限事項があるもの、またはプラットフォームに特に重要なものを指しています。詳細は、2-17 ページの「[リリース 1.0.0.6.x に関する OracleJSP の機能上の注意](#)」を参照してください。

表 2-1 Oracle プラットフォームのリリースと OracleJSP のリリース

Oracle プラット フォーム	サブレット環境	OracleJSP のリリース	OracleJSP の機能上の注意
Oracle Servlet Engine (Oracle8i)、リリース 8.1.7	サブレット 2.2	OracleJSP 1.1.0.0.0 (JSP 1.1)	該当なし
Oracle Internet Application Server リリース 1.0.1	サブレット 2.0 (Apache/JServ)	OracleJSP 1.1.0.0.0 (JSP 1.1)	該当なし
Oracle Internet Application Server リリース 1.0.0	サブレット 2.0 (Apache/JServ)	OracleJSP 1.0.0.6.0 (JSP 1.0)	globals.jsa 構成パラメータの JML の制限事項
Oracle Application Server リリース 4.0.8.2	サブレット 2.1	OracleJSP 1.0.0.6.0 (JSP 1.0)	構成パラメータの JML の制限事項
Oracle Web-to-go リリース 1.3	サブレット 2.1	OracleJSP 1.0.0.6.1 (JSP 1.0)	構成パラメータの JML の制限事項
Oracle JDeveloper リリース 3.1	サブレット 2.1	OracleJSP 1.0.0.6.1 (JSP 1.0)	構成パラメータの JML の制限事項

最新バージョンの OracleJSP をダウンロードして取り込み、前述の Oracle プラットフォームで使用できます。前述の OracleJSP のバージョンは、製品に付属のバージョンです。

特定の環境で使用する OracleJSP のリリースを確認するには、次のように JSP ページ内で暗黙的な application オブジェクトからリリース番号を取り出します。

```
<%= application.getAttribute("oracle.jsp.versionNumber") %>
```

リリース 1.0.0.6.x に関する OracleJSP の機能上の注意

ここでは、OracleJSP リリース 1.0.0.6.x に関する表 2-1 の「[OracleJSP の機能上の注意](#)」列の重要性について説明します。

- サブレット 2.0 仕様では、Web アプリケーション向けの完全なフレームワークは提供されていませんでした。Apache/JServ や Oracle Internet Application Server (Apache/JServ を使用) などのサブレット 2.0 環境では、どのリリースの OracleJSP でも `globals.jsa` メカニズムを介して拡張機能が提供され、より完成度の高いアプリケーション・フレームワークがサポートされます。詳細は、5-34 ページの「[サブレット 2.0 に対する OracleJSP のアプリケーションおよびセッションのサポート](#)」を参照してください。
- リリース 1.1.0.0.0 でサポートされる OracleJSP 構成パラメータには、リリース 1.0.0.6.x ではサポートされていなかったものがあります。A-13 ページの「[構成パラメータの概要表](#)」を参照してください。
- リリース 1.0.0.6.x の OracleJSP は JSP 1.1 仕様ではなく JSP 1.0 仕様に準拠していたため、JSP 1.1 のカスタム・タグ・ライブラリのメカニズムはサポートできませんでした。したがって、このリリースの OracleJSP では、OracleJSP トランスレータへの拡張機能を使用し、Oracle 固有のコンパイル時実装を介して JML タグがサポートされていました。

OracleJSP リリース 1.0.0.6.x で JML を使用するには、`taglib` ディレクティブ (JSP 1.1 用に指定され、OracleJSP 1.1.0.0.0 でサポート) が必要ですが、このディレクティブではライブラリを含むクラスを次のように指定する必要があります。

```
<%@ taglib uri="oracle.jsp.parse.OpenJspRegisterLib" prefix="jml" %>
```

これに対して、OracleJSP 1.1.0.0.0 のように JSP 1.1 仕様に準拠する JSP 実装を使用する場合は、`taglib` ディレクティブでタグ・ライブラリ記述ファイルを (`.tld` ファイルまたは `.jar` ファイル内で) 次のように指定します。

```
<%@ taglib uri="/WEB-INF/tlds/jmltags.tld" prefix="jml" %>
```

JML のコンパイル時実装の詳細は、[付録 C 「コンパイル時 JML タグ・サポート」](#) を参照してください。

OracleJSP の実行モデル

前述のように、OracleJSP のフレームワークは様々なサーバー環境で使用できます。OracleJSP には、次の 2 つの個別実行モデルが用意されています。

- Oracle Servlet Engine 以外の環境では、通常、OracleJSP コンテナにより各ページがオンデマンドで変換されてから、実行がトリガーされます。これは、他のほとんどのベンダーの JSP 実装と同様です。
- Oracle Servlet Engine 環境は、Oracle8i データベース内で実行される JSP ページ用であり、開発者が各ページを事前に変換し、稼働用サブリットとして Oracle8i データベースにロードします。（ページの変換、ロードおよび実行可能にするための「公開」には、コマンドライン・ツールを使用できます。変換処理は、クライアント上でもサーバー上でも実行できます。）エンド・ユーザーが JSP ページを要求すると直接実行され、変換は不要です。

オンデマンド変換モデル

OracleJSP では、Oracle Servlet Engine ではなく OracleJSP をサポートするすべてのサーバー環境に、代表的なオンデマンド変換モデルが使用されます。これには、様々な Oracle 環境のみでなく、OracleJSP、Apache Web サーバーおよび JServ の併用などが含まれます。

OracleJSP コンテナを取り込む Web サーバーから JSP ページが要求されると、`oracle.jsp.JspServlet` サブリットがインスタンス化されて起動されます（適切な Web サーバー構成の場合）。このサブリットは、OracleJSP コンテナのフロントエンドと見なすことができます。

JspServlet により JSP ページの位置が特定され、必要であれば変換されてコンパイルされ（ページ実装クラスが存在しない場合や、JSP ページ・ソースより古いタイムスタンプが付いている場合）、その実行がトリガーされます。

*.jsp ファイル拡張子（URL 内）を JspServlet にマップするには、Web サーバーが適切に構成されている必要があります。Apache/JServ、Sun Microsystems JWSKD および Tomcat に関する Web サーバーの構成ステップの詳細は、A-6 ページの「[OracleJSP を実行する Web サーバーおよびサブリット環境の構成](#)」を参照してください。

Oracle Servlet Engine の事前変換モデル

Oracle Servlet Engine (OSE)、Oracle8i 内の Web サーバーおよびサブリット・コンテナで実行するように意図されている JSP ページは、事前に変換され、Oracle8i に稼働用のサブリットとして配布されます。OSE は OracleJSP ランタイムを取り込みます。

Oracle Servlet Engine で JSP ページを実行するための配布ステップ

JSP ページを Oracle8i データベースに配布する手順は、次のとおりです。

1. JSP ページを事前変換します（通常はコンパイル作業が含まれます）。JSP トランスレータにより生成されるページ実装クラスは、実際には稼働用サブリットです。
2. 変換後の JSP ページを Oracle8i データベースにロードします。
3. 生成されたページ実装クラスを必要に応じて「ホットロード」します。
4. JSP ページを「公開」し、データベースからアクセスして実行できるようにします。

ページの変換、ロードおよび公開には、コマンドライン・ツールを使用できます。トランスレータにより、.java ファイル内でページ実装クラスが作成され、.class ファイルにコンパイルされます。

ホットロード機能を使用可能にして実行するには、追加のステップが必要です。これは、ページ実装クラス内で生成される HTML タグなど、リテラル文字列の使用効率を高めるための機能です。

Oracle8i への配布は、サーバー上またはクライアント上で変換を行って実行できます。これらの使用例と必要なステップの詳細は、6-38 ページの「[サーバー側変換を伴う Oracle8i への配布](#)」および 6-50 ページの「[クライアント側変換を伴う Oracle8i への配布](#)」を参照してください。

Oracle Servlet Engine の JSP コンテナ

Oracle Servlet Engine は独自の OracleJSP コンテナを取り込みます。このコンテナは、OracleJSP コンテナのほぼ全体からなり、OracleJSP トランスレータはありません（OSE 環境で実行される JSP ページはすべて事前変換されるためです）。

OSE には、オンデマンド変換モデルの JspServlet に似た機能を持つ、フロントエンドの JSP 処理が含まれています。

フロントエンド・コンポーネントは、公開時に Oracle8i の JNDI ネームスペースに入力されたサブリット・パス（通常は「仮想パス」と呼ばれます）に従って、JSP ページを検索して実行します。サブリットのパス名は、JSP ページの公開時に指定します。

OracleJSP に対する Oracle JDeveloper のサポート

ビジュアル Java プログラミング・ツールでは、JSP コーディングがサポートされるようになります。特に、Oracle JDeveloper では OracleJSP がサポートされ、次の機能 (JDeveloper リリース 3.1 現在) が組み込まれています。

- OracleJSP コンテナの統合によるアプリケーション開発サイクル全体のサポート — JSP ページの編集、デバッグおよび実行
- 配布された JSP ページのデバッグ
- JDeveloper Web Bean と呼ばれる、データと Web を使用可能な JavaBeans の広範囲なセット
- 事前定義の Web Bean をページに簡単に追加できるようにする JSP Element Wizard
- カスタム JavaBeans の取込みのサポート
- JDeveloper Business Components for Java (BC4J) に依存する JSP アプリケーション用の配布オプション

JSP 配布サポートの詳細は、6-67 ページの「[JDeveloper を使用した JSP ページの配布](#)」を参照してください。

JDeveloper では、JSP ページのソースにデバッグ用のブレーク・ポイントを設定し、JSP ページから JavaBeans までコールをたどることができます。これは、JSP ページに出力文を追加して状態を応答ストリームや (暗黙的な `application` オブジェクトの `log()` メソッドを介して) サーバー・ログに出力するなど、手動によるデバッグ技法に比べてはるかに便利な方法です。

JDeveloper の詳細は、JDeveloper オンライン・ヘルプ、特にトピック「JSP ページの作成」を参照してください。

基本事項

この章では、アプリケーションとセッション、JSP サーブレットの相互作用、リソース管理およびアプリケーションのルートとドキュメントのルートなど、基本的な事項について説明します。データベース・アクセスに関する JSP の初期サンプルも記載されています。

説明するトピックは、次のとおりです。

- 予備的な考慮事項
- アプリケーション・ルート機能とドキュメント・ルート機能
- JSP アプリケーションおよびセッションの概要
- JSP とサーブレットの相互作用
- JSP のリソース管理
- JSP のランタイム・エラー処理
- データベース・アクセスに関する JSP の初期サンプル

予備的な考慮事項

この項では、開発を開始する前の注意事項について説明します。説明するトピックは、次のとおりです。

- [インストールと構成の概要](#)
- [開発環境と配布環境](#)
- [クライアント側の考慮事項](#)

インストールと構成の概要

主に主要な非 Oracle 環境に関するインストールと構成については、[付録 A「一般的なインストールと構成」](#)を参照してください。

OracleJSP をサポートする Oracle 環境のインストールと構成については、該当する Oracle 製品のドキュメントを参照してください。

Oracle8i 内では、Oracle Servlet Engine (OSE) により OracleJSP が取り込まれ、Oracle8i JVM とともに提供されます。

開発環境と配布環境

通常、Apache/JServ のような非 Oracle 環境をターゲットとする JSP 開発者は、ターゲット環境と同じ環境で開発作業を行います。この場合、[付録 A「一般的なインストールと構成」](#)のインストールおよび構成手順は開発環境と配布環境の両方に適用されますが、開発中にのみ重要な構成パラメータがあります。

Oracle Servlet Engine や他の Oracle 環境をターゲットとする JSP 開発者には、少なくとも次の 2 つのオプションが用意されています。

- 開発と配布に Oracle JDeveloper を使用します。

JDeveloper により、開発中のテストに使用できるように OracleJSP とサーブレット・コンテナが取り込まれます。また、完成した製品をターゲット・ロケーションに配布するときに役立つ機能も取り込まれます。

JDeveloper における OracleJSP のサポートの概要は、2-20 ページの「[OracleJSP に対する Oracle JDeveloper のサポート](#)」を参照してください。インストールおよび構成手順については、JDeveloper のドキュメントを参照してください。

- Apache/JServ などの非 Oracle 環境で開発とテストを行ってから、ターゲットとなる Oracle 環境に配布して最終テストを行い、本番への使用を開始します。

この場合は、[付録 A](#) の情報が開発環境に重要になります。

開発環境でのテスト完了後は、JSP ページを事前変換し、OracleJSP のインストールで利用可能になるコマンドライン・ツールを使用して、Oracle8i データベースに配布できます。OracleJSP のコマンドライン・トランスレータには、関連する変換時構成パラメー

タと同等のオプションが用意されています。詳細は、6-22 ページの「[ojspc 事前変換ツール](#)」および 6-50 ページの「[クライアント側変換を伴う Oracle8i への配布](#)」を参照してください。

OracleJSP をサポートする Oracle 環境のインストールと構成の詳細は、該当する製品のドキュメントを参照してください。

クライアント側の考慮事項

JSP ページは、HTTP 1.0 以上をサポートしている標準ブラウザで動作します。

JSP ページの Java コードはすべて Web サーバーまたはデータ・サーバーで実行されるため、エンド・ユーザーが使用する Web ブラウザの JDK や他の Java 環境は無関係です。

アプリケーション・ルート機能とドキュメント・ルート機能

この項では、サーブレット 2.2 の機能性とサーブレット 2.0 の機能性の違いである、アプリケーション・ルートとドキュメント・ルートの概要について説明します。

サーブレット 2.2 環境におけるアプリケーション・ルート

前述のように、サーブレット 2.2 仕様では、アプリケーションごとに独自のサーブレット・コンテキストが提供されます。各サーブレット・コンテキストは、アプリケーションの各モジュールのベース・パスであり、サーバー・ファイル・システムのディレクトリ・パスに対応付けられています。これを、アプリケーション・ルートと呼びます。各アプリケーションには、独自のアプリケーション・ルートがあります。

これは、Web サーバーで Web アプリケーションに属する HTML ページや他のファイルのルート位置としてドキュメント・ルートが使用されるのに似ています。

サーブレット 2.2 環境のアプリケーションの場合、アプリケーション・ルート（サーブレットと JSP ページ用）とドキュメント・ルート（HTML ファイルなどの静的ファイル用）には 1 対 1 のマッピングがあり、両者は実際には同じものです。

サーブレットの URL の一般的なフォームは、次のとおりです。

```
http://host[:port]/contextpath/servletpath
```

サーブレット・コンテキストの作成時には、アプリケーション・ルートと URL のコンテキスト・パス部分とのマッピングを指定します。

たとえば、アプリケーションのアプリケーション・ルートが /home/dir/mybankappdir で、コンテキスト・パス mybank にマップされているとします。また、このアプリケーションには、サーブレット・パス loginervlet を持つサーブレットが組み込まれているとします。このサーブレットは、次のように起動できます。

`http://host[:port]/mybank/loginservlet`

(アプリケーション・ルートのディレクトリ名自体は、エンド・ユーザーには表示されません。)

この例で、このアプリケーションの HTML ページに関して、次の URL がファイル `/home/dir/mybankappdir/dir1/abc.html` を指しているとします。

`http://host[:port]/mybank/dir1/abc.html`

サーブレット環境ごとに、デフォルトのサーブレット・コンテキストもあります。このコンテキストの場合、コンテキスト・パスは単に「/」であり、デフォルトのサーブレット・コンテキストのアプリケーション・ルートにマップされます。たとえば、デフォルト・コンテキストのアプリケーション・ルートが `/home/mydefaultdir` で、サーブレット・パス `myservlet` を持つサーブレットがデフォルト・コンテキストを使用するとします。その URL は、次の例のようになります。(この場合も、アプリケーション・ルートのディレクトリ名自体は、ユーザーには表示されません。)

`http://host[:port]/myservlet`

(デフォルト・コンテキストは、URL で指定されたコンテキスト・パスとの一致がない場合にも使用されます。)

HTML ファイルに関する前述の例で、次の URL がファイル `/home/mydefaultdir/dir2/def.html` を指しているとします。

`http://host[:port]/dir2/def.html`

サーブレット 2.0 環境における OracleJSP のアプリケーション・ルート機能

Apache/JServ や他のサーブレット 2.0 環境には、アプリケーション環境が 1 つしかないため、アプリケーション・ルートの概念はありません。Web サーバーのドキュメント・ルートは、実際にはアプリケーション・ルートです。

Apache の場合、ドキュメント・ルートは通常は `.../htdocs` ディレクトリです。また、`httpd.conf` 構成ファイル内で `alias` 設定を介して「仮想」ドキュメント・ルートを指定できます。

サーブレット 2.0 環境では、OracleJSP はドキュメント・ルートとアプリケーション・ルートに関して次の機能を提供します。

- デフォルトでは、OracleJSP ではドキュメント・ルートがアプリケーション・ルートとして使用されます。
- OracleJSP の `globals.jsa` メカニズムを介して、ドキュメント・ルートのディレクトリを、指定したアプリケーションのアプリケーション・ルートとして機能するように指定できます。そのためには、`globals.jsa` ファイルを必要なディレクトリにマーカーとして配置します。(5-35 ページの「[globals.jsa の機能の概要](#)」を参照してください。)

JSP アプリケーションおよびセッションの概要

この項では、OracleJSP による JSP アプリケーションおよびセッションのサポートの概要について説明します。

一般的な OracleJSP アプリケーションおよびセッションのサポート

OracleJSP では、アプリケーションとセッションの管理に基礎となるサーブレット・メカニズムが使用されます。これらのメカニズムの詳細は、B-4 ページの「[サーブレット・セッション](#)」および B-6 ページの「[サーブレット・コンテキスト](#)」を参照してください。サーブレット 2.1 およびサーブレット 2.2 環境の場合は、これらの基礎となるメカニズムで十分であり、JSP アプリケーションごとに個別のサーブレット・コンテキストとセッション・オブジェクトを提供します。

ただし、Apache/JServ のようなサーブレット 2.0 環境では、サーブレット・メカニズムの使用には問題があります。サーブレット 2.0 仕様では Web アプリケーションの概念が詳細に定義されていないため、サーブレット 2.0 環境にはサーブレット・コンテナごとにサーブレット・コンテキストが 1 つしか存在しません。また、セッション・オブジェクトも、サーブレット・コンテナごとに 1 つのみです。ただし、Apache/JServ や他のサーブレット 2.0 環境の場合、OracleJSP は必要に応じてアプリケーションごとに個別のサーブレット・コンテキストやセッション・オブジェクトを使用できる拡張機能を備えています。（これは、単一アプリケーションのみのホストとして機能する Web サーバーの場合は不要です。）

注意： Apache/JServ や他のサーブレット 2.0 環境に関連する追加情報は、4-30 ページの「[Apache/JServ サーブレット環境に関する考慮事項](#)」および 5-35 ページの「[globals.jsa の機能の概要](#)」を参照してください。

JSP のデフォルトのセッション要求

一般に、サーブレットはデフォルトでは HTTP セッションを要求しません。ただし、JSP ページ実装クラスは、デフォルトで HTTP セッションを要求します。この動作は、次のように JSP の page ディレクティブで session パラメータを false に設定するとオーバーライドできます。

```
<%@ page ... session="false" %>
```

JSP とサーブレットの相互作用

JSP ページは様々な方法でコーディングできますが、状況によってはサーブレットが必要です。その一例は、4-17 ページの「[JSP ページ内でバイナリ・データの使用を回避する理由](#)」で説明するようにバイナリ・データを出力する場合です。

したがって、アプリケーション内でサーブレットと JSP ページの間のやりとりが必要になる場合があります。この項では、その方法を次のトピックで説明します。

- [JSP ページからのサーブレットの起動](#)
- [JSP ページから起動したサーブレットへのデータの受渡し](#)
- [サーブレットからの JSP ページの起動](#)
- [JSP ページとサーブレット間でのデータの受渡し](#)
- [JSP とサーブレットの相互作用の例](#)

重要： この説明では、サーブレット 2.2 環境を想定しています。Apache/JServ と他のサーブレット 2.0 環境に関連する考慮事項については、このドキュメントの他の項を参照してください。

JSP ページからのサーブレットの起動

ある JSP ページを他の JSP ページから起動する場合と同様に、`jsp:include` および `jsp:forward` アクション・タグを介して JSP ページからサーブレットを起動できます。(1-16 ページの「[JSP のアクションと <jsp:> タグ・セット](#)」を参照してください。) 次に例を示します。

```
<jsp:include page="/servlet/MyServlet" flush="true" />
```

ページの実行中にこの文が検出されると、ページ・バッファがブラウザに出力され、サーブレットが実行されます。サーブレットの実行が完了すると、制御は JSP ページに戻され、そのページが引き続き実行されます。これは、JSP ページ間の `jsp:include` アクションの場合と同じ機能です。

JSP ページ間の `jsp:forward` アクションの場合と同様に、次の文でページ・バッファが消去され、JSP ページの実行が終了し、サーブレットが実行されます。

```
<jsp:forward page="/servlet/MyServlet" />
```

重要： Apache/JServ または他のサーブレット 2.0 環境では、サーブレットの挿入やサーブレットへの転送はできません。かわりに JSP ラッパー・ページを記述する必要があります。詳細は、4-31 ページの「[Apache/JServ における動的挿入および転送](#)」を参照してください。

JSP ページから起動したサーブレットへのデータの受渡し

JSP ページから動的にサーブレットを挿入するか、サーブレットに転送する場合は、`jsp:param` タグを使用してサーブレットにデータを渡すことができます（他の JSP ページを挿入するか、他の JSP ページに転送する場合と同じです）。

`jsp:param` タグは、`jsp:include` または `jsp:forward` タグ内で使用します。次に例を示します。

```
<jsp:include page="/servlet/MyServlet" flush="true" >
  <jsp:param name="username" value="Smith" />
  <jsp:param name="userempno" value="9876" />
</jsp:include>
```

`jsp:param` タグの詳細は、1-16 ページの「[JSP のアクションと <jsp:> タグ・セット](#)」を参照してください。

また、適切な有効範囲を指定した JavaBeans、または HTTP 要求オブジェクトの属性を介して、JSP ページとサーブレットの間でデータをやりとりすることもできます。要求オブジェクトの属性の使用方法については、3-8 ページの「[JSP ページとサーブレット間でのデータの受渡し](#)」を参照してください。

注意： `jsp:param` タグは、JSP 1.1 仕様で導入されました。

サーブレットからの JSP ページの起動

JSP ページは、標準の `javax.servlet.RequestDispatcher` インタフェースの機能を介してサーブレットから起動できます。このメカニズムを使用するには、コードに次のステップを挿入します。

1. サーブレット・インスタンスからサーブレット・コンテキスト・インスタンスを取得します。

```
ServletContext sc = this.getServletContext();
```

2. `getRequestDispatcher()` メソッドへの入力としてターゲット JSP ページのページ相対パスまたはアプリケーション相対パスを指定し、サーブレット・コンテキスト・インスタンスから要求ディスパッチャを取得します。

```
RequestDispatcher rd = sc.getRequestDispatcher("/jsp/mypage.jsp");
```

このステップの実行前または実行中に、オプションで HTTP 要求オブジェクトの属性を介して、データを JSP ページに使用することができます。詳細は、次の「[JSP ページとサーブレット間でのデータの受渡し](#)」を参照してください。

3. 引数として HTTP 要求または応答オブジェクトを指定し、要求ディスパッチャの `include()` または `forward()` メソッドを起動します。次に例を示します。

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

または

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

この 2 つのメソッドの機能は、`jsp:include` および `jsp:forward` アクションと同じです。`include()` メソッドでは、制御が一時的に移ります。その後、実行は起動側のサーブレットに戻ります。

`forward()` メソッドにより出力バッファが消去されるため注意してください。

注意：

- 要求オブジェクトと応答オブジェクトは、`javax.servlet.http.HttpServlet` クラスで指定されている `doGet()` メソッドなど、標準サーブレット機能を使用して先に取得されています。
 - この機能は、サーブレット 2.1 仕様で導入されました。
-

JSP ページとサーブレット間でのデータの受渡し

前項「[サーブレットからの JSP ページの起動](#)」では、要求ディスパッチャを介してサーブレットから JSP ページを起動する場合に、オプションで HTTP 要求オブジェクトを介してデータを渡せることを説明しました。

そのためには、次の 2 通りのアプローチがあります。

- 要求ディスパッチャを取得するときに、`name=value` のペアを含む「？」構文を使用して URL に問合せ文字列を追加できます。

次に例を示します。

```
RequestDispatcher rd =  
    sc.getRequestDispatcher("/jsp/mypage.jsp?username=Smith");
```

ターゲット JSP ページ（またはサーブレット）内では、暗黙的な `request` オブジェクトの `getParameter()` メソッドを使用して、この方法で設定したパラメータの値を取得できます。

- HTTP 要求オブジェクトの `setAttribute()` メソッドを使用できます。

次に例を示します。

```
request.setAttribute("username", "Smith");  
RequestDispatcher rd = sc.getRequestDispatcher("/jsp/mypage.jsp");
```

ターゲット JSP ページ（またはサーブレット）内では、暗黙的な `request` オブジェクトの `getAttribute()` メソッドを使用して、この方法で設定したパラメータの値を取得できます。

注意：

- この機能は、サーブレット 2.1 仕様で導入されました。サーブレット 2.1 仕様とサーブレット 2.2 仕様ではセマンティクスが異なることに注意してください。サーブレット 2.1 環境では、特定の属性を設定できるのは1度のみです。
 - `jsp:param` タグのかわりに、この項で説明したメカニズムを使用して JSP ページからサーブレットにデータを渡すことができます。
-
-

JSP とサーブレットの相互作用の例

この項では、前述した機能を使用する JSP ページとサーブレットについて説明します。JSP ページ `Jsp2Servlet.jsp` にはサーブレット `MyServlet` が含まれており、このサーブレットには別の JSP ページ `welcome.jsp` が含まれています。

`Jsp2Servlet.jsp` のコード

```
<HTML>  
<HEAD> <TITLE> JSP Calling Servlet Demo </TITLE> </HEAD>  
<BODY>  
<!-- Forward processing to a servlet -->  
<% request.setAttribute("empid", "1234"); %>  
<jsp:include page="/servlet/MyServlet?user=Smith" flush="true"/>  
</BODY>  
</HTML>
```

`MyServlet.java` のコード

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

```
public class MyServlet extends HttpServlet {

    public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException {
        PrintWriter out= response.getWriter();
        out.println("<B><BR>User:" + request.getParameter("user"));
        out.println
            ("", Employee number:" + request.getAttribute("empid") + "</B>");
        this.getServletContext().getRequestDispatcher("/jsp/welcome.jsp").
            include(request, response);
    }
}
```

welcome.jsp のコード

```
<%-----
    Copyright © 1999, Oracle Corporation. All rights reserved.
-----%>

<HTML>
<HEAD> <TITLE> The Welcome JSP </TITLE> </HEAD>
<BODY>

<H3> Welcome! </H3>
<P><B> Today is <%= new java.util.Date() %>. Have a nice day! </B></P>
</BODY>
</HTML>
```

JSP のリソース管理

javax.servlet.http パッケージには、標準的なセッション・リソース管理メカニズムが用意されています。また、Oracle は、application、session、page および request リソースを管理するための拡張機能を備えています。

標準的なセッション・リソース管理 — HttpSessionBindingListener

JSP ページでは、JDBC 接続、文および ResultSet オブジェクトなど、実行中に取得するリソースを適切に管理する必要があります。標準の javax.servlet.http パッケージには、session 有効範囲を持つリソースを管理できるように、HttpSessionBindingListener インタフェースと HttpSessionBindingEvent クラスが用意されています。たとえば、このメカニズムを介して、session 有効範囲を持つ問合せ Bean は、インスタンス化されるときにデータベース・カーソルを取得し、HTTP セッションの終了時にカーソルをクローズできま

す。(3-19 ページの「[データベース・アクセスに関する JSP の初期サンプル](#)」の例では、問合せごとに接続をオープンし、クローズしているため、オーバーヘッドが増大します。)

この項では、`HttpSessionBindingListener` の `valueBound()` および `valueUnbound()` メソッドの使用方法について説明します。

注意： Bean インスタンスは自分自身を HTTP セッション・オブジェクトのイベント通知リストに登録する必要がありますが、`jsp:useBean` 文ではこの登録が自動的に処理されます。

valueBound() メソッドと valueUnbound() メソッド

`HttpSessionBindingListener` インタフェースを実装するオブジェクトは、`valueBound()` メソッドと `valueUnbound()` メソッドを実装できます。この 2 つのメソッドは、それぞれ入力として `HttpSessionBindingEvent` インスタンスを取ります。この 2 つのメソッドは、サーブレット・コンテナによりコールされます。つまり、オブジェクトがセッションに格納されるときには `valueBound()` メソッド、オブジェクトがセッションから削除されるとき、またはセッションがタイムアウトになるか、無効になるとときには `valueUnbound()` メソッドがコールされます。通常、開発者は `valueUnbound()` を使用して、オブジェクトが保持しているリソースを解放（後述の例では、データベース接続を解放）します。

注意： OracleJSP には追加のリソース管理用の拡張機能があり、`session` 有効範囲を持つリソースのみでなく、`page` 有効範囲、`request` 有効範囲または `application` 有効範囲を持つリソースを管理するように `JavaBeans` を記述できます。5-30 ページの「[OracleJSP のイベント処理 — JspScopeListener](#)」を参照してください。

次の「[JDBCQueryBean の JavaBeans コード](#)」では、`HttpSessionBindingListener` を実装するサンプル `JavaBeans` と、その Bean をコールするサンプル JSP ページについて説明します。

JDBCQueryBean の JavaBeans コード

次の例は、`HttpSessionBindingListener` インタフェースを実装する `JavaBeans` である `JDBCQueryBean` のサンプル・コードを示しています。（データベース接続には `JDBC OCI` ドライバが使用されます。この例を自分で実行する場合は、適切な `JDBC` ドライバと接続文字列を使用してください。）

`JDBCQueryBean` は、HTML 要求を介して検索条件を取得し（3-14 ページの「[UseJDBCQueryBean の JSP ページ](#)」を参照）、検索条件に基づいて動的問合せを実行し、結果を出力します。

また、このクラスでは、セッション終了時にデータベース接続をクローズする `valueUnbound()` メソッド (`HttpSessionBindingListener` インタフェース内で指定) も実装されます。

```
package mybeans;

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

public class JDBCQueryBean implements HttpSessionBindingListener
{
    String searchCond = "";
    String result = null;

    public void JDBCQueryBean() {
    }

    public synchronized String getResult() {
        if (result != null) return result;
        else return runQuery();
    }

    public synchronized void setSearchCond(String cond) {
        result = null;
        this.searchCond = cond;
    }

    private Connection conn = null;

    private String runQuery() {
        StringBuffer sb = new StringBuffer();
        Statement stmt = null;
        ResultSet rset = null;
        try {
            if (conn == null) {
                DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
                conn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                                    "scott", "tiger");
            }

            stmt = conn.createStatement();
            rset = stmt.executeQuery ("SELECT ename, sal FROM scott.emp "+
                                     (searchCond.equals("") ? "" : "WHERE " + searchCond ));
            result = formatResult(rset);
            return result;
        }
    }
}
```



```
    } catch (SQLException e) {
        return ("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
    }
    finally {
        try {
            if (rset != null) rset.close();
            if (stmt != null) stmt.close();
        }
        catch (SQLException ignored) {}
    }
}

private String formatResult(ResultSet rset) throws SQLException {
    StringBuffer sb = new StringBuffer();
    if (!rset.next())
        sb.append("<P> No matching rows.<P>\n");
    else {
        sb.append("<UL><B>");
        do { sb.append("<LI>" + rset.getString(1) +
            " earns $ " + rset.getInt(2) + "</LI>\n");
        } while (rset.next());
        sb.append("</B></UL>");
    }
    return sb.toString();
}

public void valueBound(HttpSessionBindingEvent event) {
    // do nothing -- the session-scoped bean is already bound
}

public synchronized void valueUnbound(HttpSessionBindingEvent event) {
    try {
        if (conn != null) conn.close();
    }
    catch (SQLException ignored) {}
}
}
```

注意： 前述のコードは、あくまでもサンプルです。大規模な Web アプリケーション内でのデータベース接続プーリングの処理に適した方法であるとは限りません。

UseJDBCQueryBean の JSP ページ

次の JSP ページでは、前項（「[JDBCQueryBean の JavaBeans コード](#)」）で定義した JDBCQueryBean JavaBeans を使用しています。起動時には、session 有効範囲を指定しています。この Bean では、ユーザーが入力した検索条件と一致する従業員名を表示するために、JDBCQueryBean が使用されています。

JDBCQueryBean は、この JSP ページ内の jsp:setProperty コマンドを介して検索条件を取得します。この Bean の searchCond プロパティは、ユーザーが HTML フォームを介して入力した searchCond 要求パラメータの値に従って設定されます。（HTML の INPUT タグは、フォームに入力される検索条件の名前が searchCond になるように指定するタグです。）

```
<jsp:useBean id="queryBean" class="mybeans.JDBCQueryBean" scope="session" />
<jsp:setProperty name="queryBean" property="searchCond" />

<HTML>
<HEAD> <TITLE> The UseJDBCQueryBean JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">

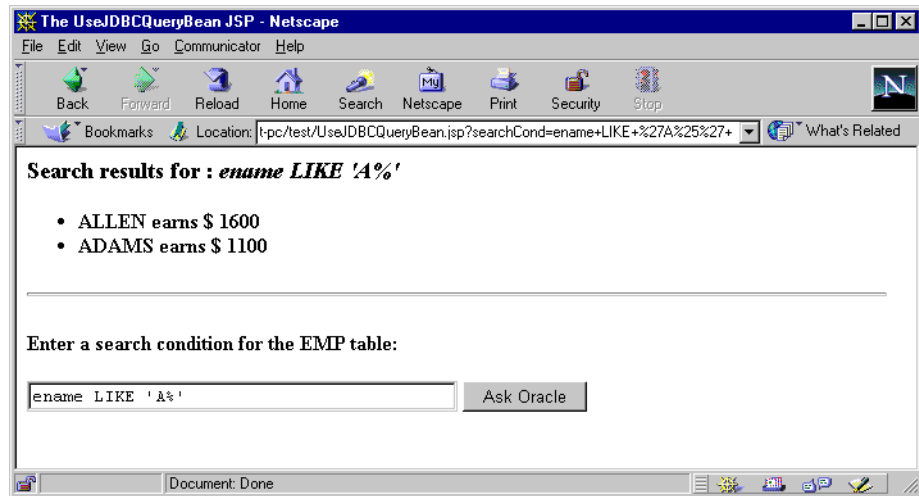
<% String searchCondition = request.getParameter("searchCond");
   if (searchCondition != null) { %>
       <H3> Search results for : <I> <%= searchCondition %> </I> </H3>
       <%= queryBean.getResult() %>
       <HR><BR>
   <% } %>

<B>Enter a search condition for the EMP table:</B>

<FORM METHOD="get">
<INPUT TYPE="text" NAME="searchCond" VALUE="ename LIKE 'A%' " SIZE="40">
<INPUT TYPE="submit" VALUE="Ask Oracle">
</FORM>

</BODY>
</HTML>
```

次の例は、このページのサンプル入出力を示しています。



HttpSessionBindingListener のメリット

前述の例では、HttpSessionBindingListener メカニズムのかわりに、JavaBeans の finalize メソッド内で接続がクローズされます。finalize メソッドは、セッション終了後に Bean のガベージが収集されるときにコールされます。ただし、finalize メソッドに比べると、HttpSessionBindingListener インタフェースの動作は予測可能です。ガベージ・コレクションの頻度は、アプリケーションのメモリー消費パターンに応じて異なります。これに対して、HttpSessionBindingListener インタフェースの valueUnbound() メソッドは、セッション終了時に確実にコールされます。

リソース管理のための Oracle の拡張機能の概要

Oracle には、page および request リソースのみでなく、application および session リソースを管理できるように、次の拡張機能が用意されています。

- JspScopeListener — application、session、page または request のリソース管理用
詳細は、5-30 ページの「[OracleJSP のイベント処理 — JspScopeListener](#)」を参照してください。
- globals.jsa application および session イベント — 通常は Apache/JServ などのサーバー プレット 2.0 環境における、application と session のイベントの開始および終了用
詳細は、5-39 ページの「[globals.jsa のイベント・ハンドラ](#)」を参照してください。

JSP のランタイム・エラー処理

JSP ページが実行され、クライアント要求が処理される間に、ページの内側または外側（コール側 JavaBeans 内など）でランタイム・エラーが発生することがあります。この項では、単純な例を挙げて JSP のエラー処理メカニズムについて説明します。

JSP エラー・ページの使用

JSP ページの実行中に発生するランタイム・エラーは、いずれも標準の Java 例外メカニズムを使用して次のどちらかの方法で処理されます。

- 標準の Java 例外処理コードを使用して、JSP ページ自体に含まれる Java スクリプトレット内で例外を捕捉し、処理できます。
- JSP ページ内で捕捉しなかった例外があると、要求と未捕捉の例外がエラー・ページに転送されます。JSP エラーの処理には、この方法をお勧めします。

発信側 JSP ページ内で `page` ディレクティブの `errorPage` パラメータを設定し、エラー・ページの URL を指定できます。（`page` ディレクティブなど、JSP ディレクティブの概要は、1-9 ページの「[ディレクティブ](#)」を参照してください。）

サーブレット 2.2 環境では、次のような指示を使用して、`web.xml` ディプロイメント・ディスクリプタ内でデフォルトのエラー・ページを指定することもできます。

```
<error-page>
  <error-code>404</error-code>
  <location>/error404.html</location>
</error-page>
```

（デフォルトのエラー・ページの詳細は、Sun Microsystems の『Java Servlet Specification, Version 2.2』を参照してください。）

エラー・ページの `page` ディレクティブでは、`isErrorPage` パラメータを `true` に設定する必要があります。

エラーを記述する例外オブジェクトは、暗黙的な `exception` オブジェクトを介してエラー・ページ内でアクセスできる `java.lang.Exception` インスタンスです。

暗黙的な `exception` オブジェクトにアクセスできるのは、エラー・ページのみです。（`exception` オブジェクトなど、JSP の暗黙的オブジェクトの詳細は、1-14 ページの「[暗黙的オブジェクト](#)」を参照してください。）

エラー・ページの使用例については、次の「[JSP エラー・ページの例](#)」を参照してください。

注意： JSP 1.1 仕様には、JSP メカニズムを介して処理できる例外タイプに関して曖昧な点があります。

OracleJSP トランスレータにより生成されるページ実装クラスでは、`java.lang.Exception` クラスまたはサブクラスのインスタンスを処理できますが、`java.lang.Throwable` クラスや `Exception` 以外のサブクラスのインスタンスは処理できません。`Throwable` インスタンスは、OracleJSP コンテナによりサーブレット・コンテナに発生します。

この曖昧さは、JSP 1.2 仕様で対処される予定です。OracleJSP の動作は、今後のリリースで適切に修正されます。

JSP エラー・ページの例

次の例の `nullpointer.jsp` では、エラーが生成され、エラー・ページ `myerror.jsp` を使用して暗黙的な `exception` オブジェクトのコンテンツが出力されます。

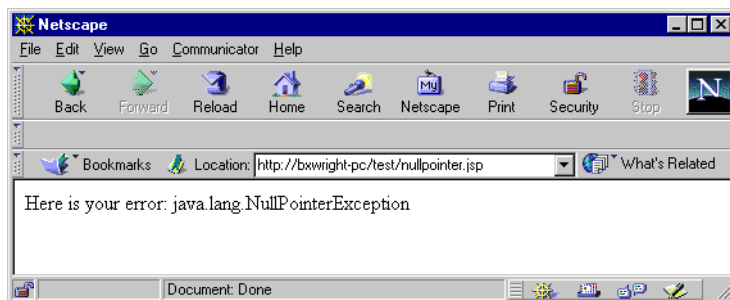
`nullpointer.jsp` のコード

```
<HTML>
<BODY>
<%@ page errorPage="myerror.jsp" %>
Null pointer is generated below:
<%
    String s=null;
    s.length();
%>
</BODY>
</HTML>
```

`myerror.jsp` のコード

```
<HTML>
<BODY>
<%@ page isErrorPage="true" %>
Here is your error:
<%= exception %>
</BODY>
</HTML>
```

この例の出力は次のようになります。



注意： 処理がエラー・ページに転送される場合、`nullpointer.jsp` の行「Null pointer is generated below:」は出力されません。これは、JSP の「挿入」機能と「転送」機能の違いを示しています。「転送」の場合は、「転送先」ページからの出力で「転送元」ページからの出力が置換されます。

データベース・アクセスに関する JSP の初期サンプル

第1章「概要」では単純な JSP の例を示しましたが、OracleJSP を使用して、Oracle データベースへアクセスすることももちろん可能です。この項では、JSP ページ内で標準 JDBC コードを使用して問合せを実行する、より重要な例について説明します。

JDBC API は Java インタフェースのセットにすぎないため、JavaServer Pages テクノロジーでは JSP スクリプトレット内での使用が直接サポートされます。

注意：

- Oracle JDBC には、代替ドライバとして以下に示すドライバが用意されています。
 - 1) Oracle クライアント・インストール用の JDBC OCI ドライバ
 - 2) 事実上どんなクライアント（アプレットなど）でも使用できる 100% Java の JDBC Thin ドライバ
 - 3) ある Oracle データベースに別の Oracle データベースからアクセスするサーバー側 JDBC Thin ドライバ
 - 4) Java コードを実行中のデータベースに（Java ストアド・プロシージャや Enterprise JavaBeans などから）アクセスするサーバー側 JDBC 内部ドライバOracle JDBC の詳細は、『Oracle8i 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。
 - OracleJSP では、静的な SQL 操作用の SQLJ（Java の埋込み SQL）もサポートされ、データベース・アクセス用のカスタム JavaBeans とカスタム SQL タグが用意されています。これらの機能については、[第5章「OracleJSP の拡張機能」](#)を参照してください。
-
-

次の例では、ユーザーが HTML フォームを介して入力（ボックスに入力して「Ask Oracle」ボタンをクリック）した検索条件から、問合せを動的に作成しています。指定した問合せを実行するために、JSP 宣言内で定義されたメソッド `runQuery()` で JDBC コードを使用します。また、JSP 宣言内では、出力を生成するための `formatResult()` メソッドも定義されています。`runQuery()` メソッドは、パスワード `tiger` で `scott` スキーマを使用します。（問合せは動的に生成されるため、SQLJ のかわりに JDBC が使用されます。SQLJ は静的 SQL 用です。）

HTML の `INPUT` タグは、フォームに入力される文字列の名前が `cond` になるように指定するタグです。したがって、`cond` は、この HTTP 要求の暗黙的な `request` オブジェクトの `getParameter()` メソッドの入力パラメータおよび `runQuery()` メソッド（`cond` 文字列を問合せの `WHERE` 句に挿入するメソッド）の入力パラメータでもあります。

注意:

- この例のもう 1 つのアプローチは、runQuery() メソッドを `<%!...%>` 宣言構文のかわりに `<%...%>` スクリプトレット構文で定義することです。
 - この例では、Oracle クライアントのインストールを必要とする JDBC OCI ドライバを使用しています。このサンプルを実行する場合は、適切な JDBC ドライバと接続文字列を使用してください。
-

```
<%@ page language="java" import="java.sql.*" %>

<HTML>
<HEAD> <TITLE> The JDBCQuery JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">

<% String searchCondition = request.getParameter("cond");
   if (searchCondition != null) { %>
       <H3> Search results for <I> <%= searchCondition %> </I> </H3>
       <B> <%= runQuery(searchCondition) %> </B> <HR><BR>
<% } %>
<B>Enter a search condition:</B>
<FORM METHOD="get">
<INPUT TYPE="text" NAME="cond" SIZE=30>
<INPUT TYPE="submit" VALUE="Ask Oracle");
</FORM>
</BODY>
</HTML>

<!-- Declare and define the runQuery() method. --%>
<%! private String runQuery(String cond) throws SQLException {
    Connection conn = null;
    Statement stmt = null;
    ResultSet rset = null;
    try {
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
        conn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                           "scott", "tiger");

        stmt = conn.createStatement();
        // dynamic query
        rset = stmt.executeQuery ("SELECT ename, sal FROM scott.emp "+
                                   (cond.equals("") ? "" : "WHERE " + cond ));
        return (formatResult(rset));
    } catch (SQLException e) {
        return ("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
    } finally {
```



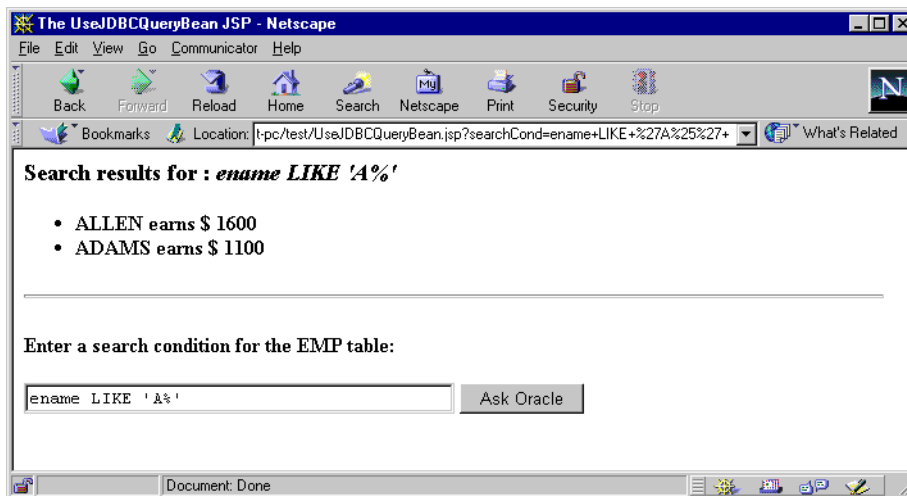
```

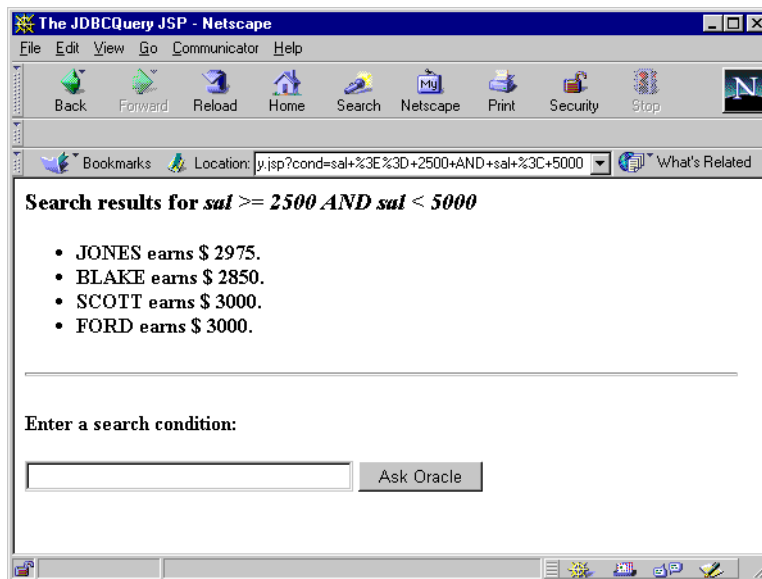
        if (rset!= null) rset.close();
        if (stmt!= null) stmt.close();
        if (conn!= null) conn.close();
    }
}

private String formatResult(ResultSet rset) throws SQLException {
    StringBuffer sb = new StringBuffer();
    if (!rset.next())
        sb.append("<P> No matching rows.<P>\n");
    else {
        sb.append("<UL>");
        do {
            sb.append("<LI>" + rset.getString(1) +
                " earns $ " + rset.getInt(2) + ".</LI>\n");
        } while (rset.next());
        sb.append("</UL>");
    }
    return sb.toString();
}
%>

```

次の図は、サンプル入出力を示しています。





重要な考慮事項

この章では、プログラミング、構成およびランタイムの重要な考慮事項と、特定の実行環境に関する特記事項について説明します。

説明するトピックは、次のとおりです。

- 一般的な JSP のプログラミング方法、ヒントおよび注意事項
- OracleJSP の構成上の重要な問題
- OracleJSP のランタイムの考慮事項（非 OSE のみ）
- Oracle Servlet Engine に関する考慮事項
- Apache/JServ サーブレット環境に関する考慮事項

一般的な JSP のプログラミング方法、ヒントおよび注意事項

この項では、OracleJSP コンテナ内で実行される JSP ページのプログラミング時の考慮事項について、ターゲット環境に関係なく説明します。

注意： この項のトピックのみでなく、OracleJSP の変換と配布に関する問題と動作にも注意する必要があります。第 6 章「JSP の変換と配布」を参照してください。

JavaBeans とスクリプトレット

JavaServer Pages テクノロジーの重要なメリットについては、1-4 ページの「[ページ表現とビジネス・ロジックの分離 - JavaBeans のコール](#)」を参照してください。ビジネス・ロジックを含み、動的コンテンツを決定する Java コードを、要求処理、表現ロジックおよび静的コンテンツを含む HTML コードから分離できます。このように分離することで、HTML のエキスパートは JSP ページ自体の表現ロジックに専念し、Java のエキスパートは JSP ページからコールされる JavaBeans のビジネス・ロジックに専念できます。

典型的な JSP ページには、Java コードの単純な断片のみが含まれています。これは、通常は要求の処理や表現のための Java の機能です。3-19 ページの「[データベース・アクセスに関する JSP の初期サンプル](#)」に示したサンプル・ページは、具体例ではありますが理想的な設計とはいえません。このサンプルの `runQuery()` メソッドに使用されるようなデータベース・アクセスは、通常は JavaBeans で使用の方が適切です。ただし、サンプルの `formatResult()` メソッドは、出力をフォーマットするもので、JSP ページ自体に使用する方が適切です。

JSP ページにおける Enterprise JavaBeans の使用

JSP ページで Enterprise JavaBeans (EJB) を使用するには、次のどちらかのアプローチを選択します。

- EJB 用の JavaBeans ラッパーを使用して、他の JavaBeans の場合と同様に JSP ページから JavaBeans コールします。
- EJB を JSP ページから直接コールします。

この項では、JSP ページから EJB をコールする例を 2 つ示します。一方の例では JSP ページが中間層環境で実行され、他方の例では Oracle Servlet Engine 内で実行されます。

この 2 つの例は、Oracle Servlet Engine を使用する重要なメリットを示しています。

Oracle EJB 実装の概要は、『Oracle8i Enterprise JavaBeans 開発者ガイドおよびリファレンス』を参照してください。

中間層の JSP ページからの EJB のコール

次の JSP ページは、Oracle Internet Application Server のような中間層環境から EJB をコールします。この場合、サービスの URL は `sess_iiop://localhost:2481:ORCL` として指定されています（独自のホスト名、IIOP ポート番号および Oracle インスタンス名を使用するには、修正が必要な場合があります）。JNDI のネーミング・コンテキストは `new InitialContext(env)` 構成を介して設定されており、`env` はこのコンテキストのパラメータを定義するハッシュ表です。初期コンテキスト (`ic`) が作成されると、コードはサービス URL と EJB の JNDI 名を使用して EJB のホーム・オブジェクトを検索します。

```
EmployeeHome home = (EmployeeHome) ic.lookup (surl + "/test/employeeBean");
```

次に、`home.create()` メソッドがコールされて Bean のインスタンスが作成され、Bean の `query()` メソッドがコールされて、JSP ページの HTML フォームを介して番号が入力された従業員の氏名と給与額が取得されます。

次にサンプル・コードを示します。

```
<HTML>
<%@ page import="employee.Employee, employee.EmployeeHome,
employee.EmpRecord, oracle.aurora.jndi.sess_iiop.ServiceCtx,
javax.naming.Context, javax.naming.InitialContext, java.util.Hashtable"
%>

<HEAD> <TITLE> The CalleJB JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">
<BR>
<% String empNum = request.getParameter("empNum");
String surl = request.getParameter("surl");
if (empNum != null) {
    try {
        Hashtable env = new Hashtable();
        env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put(Context.SECURITY_PRINCIPAL, "scott");
        env.put(Context.SECURITY_CREDENTIALS, "tiger");
        env.put(Context.SECURITY_AUTHENTICATION,
            ServiceCtx.NON_SSL_LOGIN);
        Context ic = new InitialContext (env);
        EmployeeHome home = (EmployeeHome)ic.lookup (surl +
            "/test/employeeBean");
        Employee testBean = home.create();
        EmpRecord empRec = testBean.query (Integer.parseInt (empNum));

    %>
<h2><BLOCKQUOTE><BIG><PRE>
    Hello, I'm an EJB in Oracle8i.
    Employee <%= empRec.ename %> earns $ <%= empRec.sal %>
<% } catch (Exception e) { %>
```

```
                Error occurred: <%= e %>
<%    }
    } %>
</PRE></BIG></BLOCKQUOTE></h2>
<HR>

<P><B>Enter an employee number and EJB service URL:</B></P>
<FORM METHOD=get>
<INPUT TYPE=text NAME="empNum" SIZE=10 value="7654">
<INPUT TYPE=text NAME="surl" SIZE=40 value="sess_iiop://localhost:2481:ORCL">
<INPUT TYPE=submit VALUE="Ask Oracle">
</FORM>
</BODY>
</HTML>
```

Oracle Servlet Engine 内の JSP ページからの EJB のコール

JSP ページを Oracle8i の Oracle Servlet Engine に配布すると、EJB の検索と起動がはるかに簡略化され、高度に最適化されます。この場合、Bean の検索は Oracle8i の JNDI ネームスペース内でローカルに実行されます。サービスの URL を明示的に指定する必要はありません。ネーミング・コンテキストは、カレント・セッション用に次のような単純なコールで初期化されます。

```
Context ic = new InitialContext();
```

中間層の例とは異なり、この場合のコンストラクタには引数が不要なので注意してください。Bean は、その JNDI 名のみを使用して（サービスの URL を使用せずに）検索されます。

```
EmployeeHome home = (EmployeeHome)ic.lookup ("/test/employeeBean");
```

次にサンプル・コードを示します。

```
<HTML>
<%@ page import="employee.Employee, employee.EmployeeHome,
employee.EmpRecord, oracle.aurora.jndi.sess_iiop.ServiceCtx,
javax.naming.Context, javax.naming.InitialContext,
java.util.Hashtable" %>

<HEAD> <TITLE> The CalleJB JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">
<BR>
<%    String empNum = request.getParameter("empNum");
    if (empNum != null) {
        try {
            Context ic = new InitialContext();
            EmployeeHome home = (EmployeeHome)ic.lookup("/test/employeeBean");
            Employee testBean = home.create();
```

```

        EmpRecord empRec = testBean.query (Integer.parseInt (empNum));
    %>
<h2><BLOCKQUOTE><BIG><PRE>
        Hello, I'm an EJB in Oracle8i.
        Employee <%= empRec.ename %> earns $ <%= empRec.sal %>
    % } catch (Exception e) { %>
        Error occurred: <%= e %>
    % }
    } %>
</PRE></BIG></BLOCKQUOTE></h2>
<HR>

<P><B>Enter an employee number URL:</B></P>
<FORM METHOD=get>
<INPUT TYPE=text NAME="empNum" SIZE=10 value="7654">
<INPUT TYPE=submit VALUE="Ask Oracle">
</FORM>
</BODY>
</HTML>

```

JDBC のパフォーマンス強化機能の使用

OracleJSP により実行される JSP アプリケーションでは、次のパフォーマンス強化機能を使用できます。

- データベース接続のキャッシュ（Oracle の拡張機能を使用）
- JDBC 文のキャッシュ（Oracle の拡張機能を使用）
- 文のバッチ更新（Oracle の拡張機能を使用）
- 問合せ中の行のプリフェッチ（Oracle の拡張機能を使用）
- 行セットのキャッシュ（Sun Microsystems の拡張機能を使用）

これらのパフォーマンス機能のほとんどは、ConnBean および ConnCacheBean Database-Access JavaBeans でサポートされます（ただし、DBBean ではサポートされません）。これらの Bean については、5-12 ページの「[Oracle Database-Access JavaBeans](#)」を参照してください。

データベース接続のキャッシュ

新規データベース接続の作成は高コストの操作であり、できるだけ回避する必要があります。かわりに、データベース接続のキャッシュを使用してください。JSP アプリケーションは、物理接続の既存のプールから論理接続を取得し、終了後にプールに戻すことができます。

接続プールの作成時には、application、session、page または request という 4 つの JSP 有効範囲のいずれか 1 つを指定できます。最も効率的なのは、最大限の有効範囲を使用する方法です。つまり、Web サーバーにより許可されている場合は application 有効範囲、許可されていない場合は session 有効範囲を使用します。

Oracle JDBC 接続キャッシュ・スキーマは、JDBC 2.0 標準拡張機能で指定された標準接続プーリングに基づいて構築されており、OracleJSP で提供される ConnCacheBean Database-Access JavaBeans 内で実装されます。ほとんどの OracleJSP 開発者は、この方法で接続キャッシュを使用します。詳細は、5-15 ページの「[接続キャッシュ用の ConnCacheBean](#)」を参照してください。

また、次の例のように、Oracle JDBC の OracleConnectionCacheImpl クラスを、JavaBeans であるかのように直接使用することもできます（ただし、すべての OracleConnectionCacheImpl 機能は ConnCacheBean を介して使用できます）。

```
<jsp:useBean id="occi" class="oracle.jdbc.pool.OracleConnectionCacheImpl"
             scope="session" />
```

これと同じプロパティを、ConnCacheBean の場合と同様に OracleConnectionCacheImpl でも使用できます。どちらも、jsp:setProperty 文を介して、またはクラスの設定メソッドを介して直接設定できます。

OracleConnectionCacheImpl を直接使用する例は、9-16 ページの「[接続キャッシュ — ConnCache3.jsp および ConnCache1.jsp](#)」を参照してください。

Oracle JDBC の接続キャッシュ・スキームと OracleConnectionCacheImpl クラスの詳細は、『Oracle8i 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 文のキャッシュ

文のキャッシュは、リリース 8.1.7 で使用可能な Oracle JDBC の拡張機能です。この機能により、ループ内や繰り返しコールされるメソッド内など、1 つの物理接続で繰り返し使用される実行文をキャッシュしてパフォーマンスを改善できます。文をキャッシュすると、キャッシュされた文を再び解析したり、文オブジェクトを再作成したり、文が実行されるたびにパラメータのサイズ定義を再計算する必要はありません。

Oracle JDBC の文のキャッシュ・スキームは、OracleJSP で提供される ConnBean および ConnCacheBean Database-Access JavaBeans 内で実装されます。これらの各 Bean は stmtCacheSize プロパティを持ち、jsp:setProperty 文または Bean の setStmtCacheSize() メソッドを介して設定できます。詳細は、5-13 ページの「[データベース接続用の ConnBean](#)」および 5-15 ページの「[接続キャッシュ用の ConnCacheBean](#)」を参照してください。

文のキャッシュは、Oracle JDBC の OracleConnection および OracleConnectionCacheImpl クラスを介して直接使用することもできます。Oracle JDBC の文キャッシュ・スキームと OracleConnection および OracleConnectionCacheImpl クラスの詳細は、『Oracle8i 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。

重要： 文をキャッシュできるのは、単一の物理接続内のみです。接続キャッシュ用に文キャッシュを使用可能にすると、単一のプーリングされた接続オブジェクトからの複数の論理接続オブジェクトに対して文をキャッシュできますが、複数のプーリングされた接続オブジェクトに対してキャッシュすることはできません。

バッチ更新

Oracle JDBC のバッチ更新機能により、プリコンパイルされた各 SQL 文オブジェクトにバッチ値（制限）が対応付けられます。バッチ更新を使用すると、プリコンパイルされた SQL 文は `executeBatch()` メソッドがコールされるたびに JDBC ドライバにより実行されるかわりに、累積された実行要求のバッチに追加されます。ドライバは、バッチ値に達すると、すべての操作をデータベースに渡して実行させます。たとえば、バッチ値が 10 であれば、10 の操作からなる各バッチが一度にデータベースに送信されて処理されます。

OracleJSP では、Oracle JDBC のバッチ更新が `ConnBean Database-Access JavaBeans` の `executeBatch` プロパティを介して直接サポートされます。このプロパティは、`jsp:setProperty` 文または `Bean` の設定メソッドを介して設定できます。かわりに `ConnCacheBean` を使用する場合は、作成する接続オブジェクトと文オブジェクト内で Oracle JDBC 機能を介してバッチ更新を有効化できます。これらの JavaBeans の詳細は、5-13 ページの「[データベース接続用の ConnBean](#)」および 5-15 ページの「[接続キャッシュ用の ConnCacheBean](#)」を参照してください。

Oracle JDBC のバッチ更新の詳細は、『Oracle8i 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。

行のプリフェッチ

Oracle JDBC の行のプリフェッチ機能を使用すると、結果セットが問合せ中に移入される間に、データベースへの各トリップ中にクライアントにプリフェッチされる行数を設定し、サーバーへのラウンドトリップ数を削減できます。

OracleJSP では、Oracle JDBC の行のプリフェッチが `ConnBean Database-Access JavaBeans` の `preFetch` プロパティを介して直接サポートされます。このプロパティは、`jsp:setProperty` 文または `Bean` の設定メソッドを介して設定できます。かわりに `ConnCacheBean` を使用する場合は、作成する接続オブジェクトと文オブジェクト内で Oracle JDBC 機能を介して行のプリフェッチを有効化できます。これらの JavaBeans の詳細は、5-13 ページの「[データベース接続用の ConnBean](#)」および 5-15 ページの「[接続キャッシュ用の ConnCacheBean](#)」を参照してください。

Oracle JDBC の行のプリフェッチの詳細は、『Oracle8i 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。

行セットのキャッシュ

キャッシュされた行セットにより、データベースから取得されるデータ用のシリアル化可能でスクロール可能な非接続コンテナが提供されます。この機能は、あまり変更しない少数のデータ・セット、特にクライアントが頻繁に情報を要求したり、継続的に情報にアクセスする場合に役立ちます。これに対して、通常の結果セットを使用するには、基礎となる接続や他のリソースを保持する必要があります。ただし、大量にキャッシュされた行セットはクライアント上で大量のメモリーを消費するため注意してください。

リリース 8.1.7 時点では、Oracle JDBC にはキャッシュされた行セットの実装は用意されていませんが、Sun Microsystems からリファレンス実装を入手できます。Sun Microsystems の Web サイトからファイル `rowset.jar` をダウンロードし、Web サーバーの `CLASSPATH` に挿入して、パッケージ `sun.jdbc.rowset.*` をコードにインポートしてください。次に、コードを使用し、次の例のようにキャッシュされた行セットを作成して移入します。

```
CachedRowSet crs = new CachedRowSet();  
crs.populate(rset); // rset is a previously created JDBC ResultSet object.
```

行セットの移入後は、元の結果セットの取得時に使用した接続オブジェクトと文オブジェクトをクローズできます。

静的挿入と動的挿入

1-9 ページの「[ディレクティブ](#)」で説明した `include` ディレクティブは、挿入ページのコピーを作成し、変換中に JSP ページ（「挿入先ページ」）にコピーします。これは静的挿入（または変換時挿入）と呼ばれ、次の構文が使用されます。

```
<%@ include file="/jsp/userinfopage.jsp" %>
```

1-16 ページの「[JSP のアクションと <jsp:> タグ・セット](#)」で説明した `jsp:include` アクションでは、挿入ページからの出力が実行時に挿入先ページの出力に動的に挿入されます。これは動的挿入（またはランタイム挿入）と呼ばれ、次の構文が使用されます。

```
<jsp:include page="/jsp/userinfopage.jsp" flush="true" />
```

C 構文をよく理解しているユーザーにとっては、静的挿入は `#include` 文に似ています。動的挿入はファンクション・コールに似ています。どちらも便利ですが、その用途に違いがあります。

注意： 静的挿入も動的挿入も、使用できるのは同じサーブレット・コンテキスト内のみです。

静的挿入のロジスティック

静的挿入では、変換中に挿入ページのテキストが（include ディレクティブの位置で）挿入先ページに物理的にコピーされるかのように、挿入先 JSP ページに関して生成されるコードのサイズが大きくなります。挿入先ページにページが複数回挿入される場合は、複数のコピーが作成されます。

静的に挿入される JSP ページは、独立した変換可能エンティティでなくてもかまいません。この JSP ページは、単に、挿入先ページにコピーされるテキストで構成されます。挿入テキストがコピーされた挿入先ページは、変換可能である必要があります。また、実際には、挿入ページがコピーされる前は、挿入先ページは変換可能でなくてもかまいません。静的に挿入されるページの順序によっては、それぞれを単体でない断片部分に分割できます。

動的挿入のロジスティック

動的挿入では、挿入先ページ用に生成されるコードのサイズが大幅に増大することはありませんが、要求ディスパッチャなどを対象とするメソッドのコールが追加されます。動的挿入では、挿入ページのテキストが挿入先ページに物理的にコピーされるのに対して、ランタイム処理が挿入先ページから挿入ページに切り替わります。

動的挿入では、処理のオーバーヘッドが増大し、要求ディスパッチャへの追加コールが必要になります。

動的に挿入されるページは、独立したエンティティとして単独で変換して実行できる必要があります。同様に、挿入先ページも、独立したエンティティとして、動的挿入なしで変換して実行できる必要があります。

メリット、デメリットおよび典型的な使用方法

静的挿入はページ・サイズに影響し、動的挿入は処理のオーバーヘッドに影響します。静的挿入では、動的挿入に必要な要求ディスパッチャのオーバーヘッドは回避されますが、大量のファイルが関与する場合は問題が生じることがあります。（生成されるページの実装クラスのサービス・メソッドには、64KB のサイズ制限があります。4-12 ページの「[JSP ページの大量の静的コンテンツに関する回避策](#)」を参照してください。）

また、静的挿入を過剰に使用すると、JSP ページのデバッグが困難になり、プログラムの実行をトレースしにくくなることもあります。静的に挿入されるページ間では、微妙な相互依存性を回避してください。

通常、静的挿入は、コンテンツが複数の JSP ページで繰り返し使用される小さいファイルの挿入に使用します。たとえば、次のような場合に使用します。

- アプリケーションの各ページの最上部または最下部に、ロゴや著作権メッセージを静的に挿入する場合
- 複数ページに必要な宣言やディレクティブ（Java クラスのインポートなど）を含むページを静的に挿入する場合

- アプリケーションの各ページからセントラル・ステータス・チェッカ・ページを静的に挿入する場合（4-11 ページの「[セントラル・チェッカ・ページの使用](#)」を参照）

動的挿入は、モジュール形式のプログラミングに役立ちます。1つのページが単独で実行される場合と、他のページの出力の一部を生成するために使用される場合があります。動的に挿入されたページは複数の挿入先ページで再使用でき、挿入先ページのサイズは増大しません。

JSP タグ・ライブラリの作成と使用を検討する場合

状況によっては、開発チームによるカスタム・タグの作成と使用の検討が必要となる場合があります。特に、次の状況を考慮してください。

- 他の方法では、JSP ページに出力の表現とフォーマットに関する大量の Java ロジックを挿入する必要がある場合
- JSP 出力に特別な操作やリダイレクションが必要な場合

Java 構文の置換

JSP 開発者は、Java のプログラミング経験があるとは限らないため、JSP 出力の表現とフォーマットを指定するロジックなど、ページ内の Java ロジックをコーディングする作業の担当者候補としては理想的でない場合があります。

このような場合は、JSP タグ・ライブラリが役立つ可能性があります。多数の JSP ページが出力の生成時にこのようなロジックを必要とする場合は、JSP 開発者にとって Java ロジックを置換するタグ・ライブラリがあれば非常に便利です。

その一例が、OracleJSP で提供される JML サンプル・タグ・ライブラリです。このライブラリには、Java のループと条件と同等のロジックをサポートするタグが含まれています。（詳細は、7-18 ページの「[JSP マークアップ言語 \(JML\) のサンプル・タグ・ライブラリの概要](#)」を参照してください。）

JSP 出力の操作またはリダイレクト

カスタム・タグに共通するもう 1 つの状況は、応答出力の特殊なランタイム処理が必要な場合です。目的の機能性を得るには、余分な処理ステップや、ブラウザ以外への出力のリダイレクションが必要となる可能性があります。

その一例が、出力がブラウザではなくログ・ファイルにリダイレクトされるテキスト本体を囲むために、カスタム・タグを作成するステップです。次にこの例を示します（cust はタグ・ライブラリの接頭辞、log はライブラリのタグの 1 つです）。

```
<cust:log>
  Today is <%= new java.util.Date() %>
  Text to log.
```

```

    More text to log.
    Still more text to log.
</cust:log>

```

タグ本体の処理については、7-4 ページの「[タグ・ハンドラ](#)」を参照してください。

セントラル・チェッカ・ページの使用

JSP アプリケーション全般の管理または監視用に、アプリケーション内の各ページから挿入するセントラル・チェッカ・ページを使用すると役立つ場合があります。セントラル・チェッカ・ページでは、各ページの実行中に次のようなタスクを実行できます。

- セッション・ステータスのチェック
- ログイン・ステータスのチェック（有効なログインが実行されたかどうかを調べるための cookie のチェックなど）
- 使用プロファイルのチェック（マウスのクリックやページへのアクセスなど、必要なイベントを記録するためにロギング・メカニズムが実装されている場合）

その他にも、様々な用途が考えられます。

たとえば、`HttpSessionBindingListener` インタフェースを実装するセッション・チェッカ・クラス `MySessionChecker` を考えます。（3-10 ページの「[標準的なセッション・リソース管理 — HttpSessionBindingListener](#)」を参照してください。）

```

public class MySessionChecker implements HttpSessionBindingListener
{
    ...

    valueBound(HttpSessionBindingEvent event)
    {...}

    valueUnbound(HttpSessionBindingEvent event)
    {...}

    ...
}

```

次のような要素を含む `centralcheck.jsp` などのチェッカ JSP ページを作成できます。

```
<jsp:useBean id="sessioncheck" class="MySessionChecker" scope="session" />
```

`centralcheck.jsp` を含むどのページでも、`sessioncheck` が（セッションの終了時に）有効範囲外になるとすぐに、サーブレット・コンテナは `MySessionChecker` クラスに実装されている `valueUnbound()` メソッドをコールします。これは、セッション・リソースを管理するためです。`centralcheck.jsp` は、アプリケーションの各 JSP ページの末尾に挿入できます。

JSP ページの大量の静的コンテンツに関する回避策

JSP ページに大量の静的コンテンツ（特に、実行時に変化するコンテンツを持たない大量の HTML コード）が含まれていると、変換速度や実行速度が低下することがあります。

この場合は、主に次の 2 つの回避策があります（どちらの回避策でも変換が高速になります）。

- 静的 HTML を別個のファイルに格納し、動的 include コマンド (`jsp:include`) を使用して、出力を実行時に JSP ページの出力に挿入します。`jsp:include` コマンドの詳細は、1-16 ページの「[JSP のアクションと <jsp:> タグ・セット](#)」を参照してください。

重要： 静的 `<%@ include... %>` コマンドは機能しません。その結果、挿入ファイルは変換時に挿入され、そのコードは実際には挿入先ページにコピーされます。これでは問題は解決しません。

- 静的 HTML を Java リソース・ファイルに格納します。

`external_resource` 構成パラメータを有効化すると、この処理が OracleJSP により実行されます。このパラメータについては、A-13 ページの「[OracleJSP の構成パラメータ \(非 OSE\)](#)」を参照してください。

Oracle8i に配布する場合は、`ojspc` 事前変換ツールの `-extres` および `-hotload` オプションと、`publishjsp` セッション・シェル・コマンドの `-hotload` オプションでもこの機能を実行できます。

注意： 静的 HTML をリソース・ファイルに格納すると、クラスがロードされるたびにページ実装クラスがリソース・ファイルをロードする必要があるため、前述の `jsp:include` による回避策に比べて、メモリーのフットプリントが大きくなる場合があります。

また、可能性は低いものの、大量の静的コンテンツを含む JSP ページで問題になるのは、ほとんど（すべてではない）の JVM では 1 つのメソッド内のコードに 64KB というサイズ制限が適用されることです。`javac` ではコンパイルできますが、JVM では実行できません。JSP トランスレータの実装によっては、実際に JSP ページのソース・ファイル全体から生成される Java コードは、ページ実装クラスのサービス・メソッドとなるため、これが JSP ページの問題になる可能性があります。（Java コードは静的 HTML をブラウザに出力する目的で生成され、すべてのスクリプトレットからの Java コードは直接コピーされます。）

その他に、きわめてまれですが、JSP ページ内の Java スクリプトレットが大きすぎるために、サービス・メソッドにサイズ制限の問題が生じる使用例があります。ただし、1 ページに含まれる Java コードが問題になるほど大きければ、そのコードは `JavaBeans` に移動する必要があります。

メソッド変数宣言とメンバー変数宣言

1-11 ページの「[スクリプト要素](#)」では、メンバー変数の宣言には JSP の `<%! ... %>` 宣言を使用しますが、メソッド変数は `<% ... %>` スクリプトレットで宣言する必要があると説明しました。

次のように、各宣言には変数の使用方法に応じて適切なメカニズムを使用するように注意してください。

- JSP の `<%! ... %>` 宣言構文で宣言される変数は、JSP トランスレータにより生成されるページ実装クラス内で、クラス・レベルで宣言されます。
- JSP の `<% ... %>` スクリプトレット構文で宣言される変数は、ページ実装クラスのサービス・メソッドに対してローカルです。

次の例の `decltest.jsp` を考えます。

```
<HTML>
<BODY>
<% double f2=0.0; %>
<%! double f1=0.0; %>
Variable declaration test.
</BODY>
</HTML>
```

この場合、ページ実装クラス内のコードは次のようになります。

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

public class decltest extends oracle.jsp.runtime.HttpJsp {
    ...

    // ** Begin Declarations
    double f1=0.0;           // *** f1 declaration is generated here ***
    // ** End Declarations

    public void _jspService
        (HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        ...

        try {
            out.println( "<HTML>");
            out.println( "<BODY>");
            double f2=0.0;    // *** f2 declaration is generated here ***
            out.println( "");
            out.println( "");
            out.println( "Variable declaration test.");
        }
```

```
        out.println( "</BODY>");
        out.println( "</HTML>");
        out.flush();
    }
    catch( Exception e) {
        try {
            if (out != null) out.clear();
        }
        catch( Exception clearException) {
        }
    }
    finally {
        if (out != null) out.close();
    }
}
}
```

注意： このコードは、概念を示すためにのみ使用しています。ほとんどのクラスは例を簡潔にするために削除されており、OracleJSP により生成されるページ実装クラスの実際のコードとは多少異なります。

page ディレクティブの特性

この項では、page ディレクティブの次の特性について説明します。

- page ディレクティブは静的であり、変換時に有効になります。実行時に評価されるパラメータ設定は指定できません。
- page ディレクティブの Java import 設定は、単一の JSP ページ内で累積されます。

page ディレクティブは静的

page ディレクティブは静的であり、変換中に解析されます。実行時に解析される動的設定は指定できません。次に例を示します。

例 1 次の page ディレクティブは有効です。

```
<%@ page contentType="text/html; charset=EUCJIS" %>
```

例 2 次の page ディレクティブは無効で、エラーとなります（この例では EUCJIS がハードコーディングされていますが、この例は実行時に動的に判別されるキャラクタ・セットすべてについて true が設定されています）。

```
<% String s="EUCJIS"; %>
<%@ page contentType="text/html; charset=<%=s%>" %>
```


page ディレクティブの一部の設定には、予備作業があります。例 2 には、8-3 ページの「コンテンツ型の動的設定」で説明したように、コンテンツ・タイプの動的設定を可能にする `setContentType()` メソッドがあります。

page ディレクティブの import 設定は累積型

page ディレクティブの Java import 設定は、単一の JSP ページ内で累積されます。

どの単一 JSP ページ内でも、次の 2 つの例は等価です。

```
<%@ page language="java" %>
<%@ page import="sqlj.runtime.ref.DefaultContext, java.sql.*" %>
```

または

```
<%@ page language="java" %>
<%@ page import="sqlj.runtime.ref.DefaultContext" %>
<%@ page import="java.sql.*" %>
```

最初の page ディレクティブの import 設定の後で、2 番目の page ディレクティブの import 設定は、インポートされるクラスやパッケージを置換するのではなく、インポートされるクラスやパッケージのセットに追加されます。

JSP の空白保持とバイナリ・データでの使用

OracleJSP（および JavaServer Pages の実装全般）では、ブラウザに出力されるソース・コード内で CR+LF などの空白が保たれます。このような空白の挿入は開発者が意図したことではなく、通常は JSP テクノロジーがバイナリ・データの生成に適さなくなります。

空白の例

次の 2 つの JSP ページでは、ソース・コードに改行が使用されているため異なる HTML 出力が生成されています。

例 1 — 改行なし (nowhisp.jsp)

次の JSP ページでは、`Date()` および `getParameter()` のコールの後に改行がありません。`(Date())` のコールで始まる 3 行目と 4 行目は、実際には 1 行のコードが折り返されて 2 行になったものです。）

```
<HTML>
<BODY>
<%= new java.util.Date() %> <% String user=request.getParameter("user"); %> <%=
(user==null) ? "" : user %>
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
```

```
</FORM>
</BODY>
</HTML>
```

この例では、次の HTML 出力がブラウザに送られます。（日付の後に空白行がないことに注意してください。）

```
<HTML>
<BODY>
Tue May 30 20:07:04 PDT 2000
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

例 2 — 改行 (whitesp.jsp)

次の JSP ページでは、Date() および getParameter() のコールの後に改行があります。

```
<HTML>
<BODY>
<%= new java.util.Date() %>
<% String user=request.getParameter("user"); %>
<%= (user==null) ? "" : user %>
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

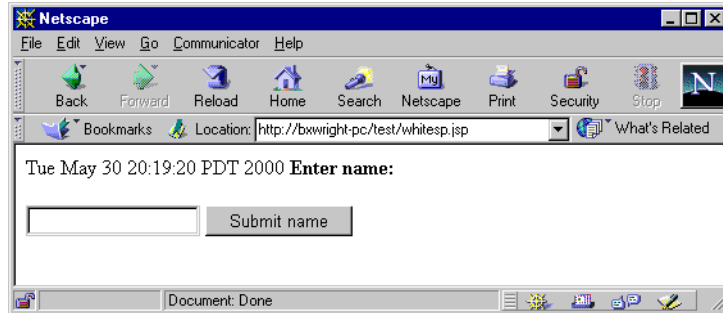
この例では、次の HTML 出力がブラウザに送られます。

```
<HTML>
<BODY>
Tue May 30 20:19:20 PDT 2000

<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
```

```
</BODY>
</HTML>
```

日付と「Enter name:」行の間に 2 つの空白行があることに注意してください。この場合、例 1 でも例 2 でもブラウザの表示内容は次のように同じのため、違いは重要ではありません。ただし、この説明は空白の保持に関するポイントを示しています。



JSP ページ内でバイナリ・データの使用を回避する理由

次の理由で、JSP ページはバイナリ・データの生成には適していません。通常はかわりにサブリットを使用する必要があります。

- JSP 実装は、バイナリ・データを処理するように設計されていません。JspWriter オブジェクトにはロー・バイトを記述するメソッドがありません。
- 実行中には、JSP コンテナにより空白が保たれます。空白が不要な場合もあり、JSP ページはブラウザへのバイナリ出力（.gif ファイルなど）の生成や空白が重要な他の用途には適していません。

次に例を示します。

```
...
<% out.getOutputStream().write(...binary data...) %>
<% out.getOutputStream().write(...more binary data...) %>
```

この場合、出力バッファのバッファリングに応じて、ブラウザはバイナリ・データの途中や最後に不要な改行文字を受け取ります。この問題は、コード行の間に改行を使用しないことで回避できますが、これは望ましいプログラミング・スタイルとは言えません。

JSP テクノロジは動的なテキスト・コンテンツのプログラミングを簡素化するためのものであり、JSP ページ内でバイナリ・データを生成しようとすると、そのポイントがいずれにせよ大きく失われることになります。

OracleJSP の構成上の重要な問題

この項では、主要な page ディレクティブ・パラメータと OracleJSP の構成パラメータの設定方法による重要な効果について説明します。ここでは、JSP ページの最適化、CLASSPATH およびクラス・ローダーの問題に重点を置きます。説明するトピックは、次のとおりです。

- [JSP 実行の最適化](#)
- [CLASSPATH とクラス・ローダーの問題（非 OSE のみ）](#)

JSP 実行の最適化

JSP のパフォーマンスを最適化する場合に、次のような設定を考慮します。

- [JSP ページのアンバッファリング](#)
- [再変換チェックの回避（非 OSE のみ）](#)（開発者モード）
- [HTTP セッション使用の回避](#)

JSP ページのアンバッファリング

デフォルトでは、JSP ページにはページ・バッファと呼ばれるメモリ領域が使用されます。このバッファ（デフォルトでは 8KB）が必要になるのは、ページで動的な NLS コンテンツ・タイプ設定、転送またはエラー・ページが使用される場合です。これらの機能を使用しない場合は、次のように page ディレクティブでバッファを使用禁止にすることができます。

```
<%@ page buffer="none" %>
```

これにより、メモリ使用量が減少し、出力ステップが省略されて（出力は最初にバッファに送られるかわりに、ブラウザに直接送られます）、ページのパフォーマンスが改善されます。

再変換チェックの回避（非 OSE のみ）

OracleJSP では、JSP ページの実行時に、デフォルトでページ実装クラスの有無がチェックされ、.class ファイルのタイムスタンプが .jsp ソース・ファイルのタイムスタンプと比較されて、.class ファイルの方が古ければページが再変換されます。

タイムスタンプの比較が不要な場合（ソース・コードが変化しない典型的な配布環境など）、OracleJSP の developer_mode フラグを無効化して（developer_mode=false）、タイムスタンプの比較を回避できます。

デフォルト設定は true です。このフラグを Apache/JServ、JSWDK および Tomcat 環境で設定する方法は、A-23 ページの「[OracleJSP の構成パラメータ設定](#)」を参照してください。

HTTP セッション使用の回避

JSP ページに HTTP セッションが不要な（実際には、セッション属性の格納や取得を必要としない）場合は、次の page ディレクティブを介してセッションの使用を回避できます。

```
<%@ page session="false" %>
```

これにより、セッションの作成や取得によるオーバーヘッドが排除され、ページのパフォーマンスが改善されます。

サーブレットではデフォルトでセッションは使用されませんが、JSP ページではデフォルトでセッションが使用されるため注意してください。背景情報は、B-4 ページの「[サーブレット・セッション](#)」を参照してください。

CLASSPATH とクラス・ローダーの問題（非 OSE のみ）

OracleJSP では、Web サーバーの CLASSPATH とは別個に独自の CLASSPATH が使用され、デフォルトでは独自のクラス・ローダーを使用してこの CLASSPATH からクラスがロードされます。これには重要なメリットとデメリットがあります。

OracleJSP の CLASSPATH は次の要素の組合せです。

- OracleJSP のデフォルトの CLASSPATH
- OracleJSP の classpath 構成パラメータに指定する追加の CLASSPATH

システムのクラス・ローダーのかわりに OracleJSP のクラス・ローダーでロードするクラスがある場合は、OracleJSP の classpath 構成パラメータを使用するか、該当のクラスを OracleJSP のデフォルトの CLASSPATH に挿入します。関連情報については、4-21 ページの「[OracleJSP クラス・ローダーのメリットとデメリット](#)」を参照してください。

OracleJSP のデフォルトの CLASSPATH

OracleJSP では、必要なクラス（JavaBeans など）の .class ファイルと .jar ファイルの位置を特定できるように、Web サーバー上の標準位置が定義されます。各ファイルはこれらの位置で検索され、Web サーバーの CLASSPATH 構成は使用されません。

これらの位置は次のとおりで、アプリケーション・ルートとの相対位置になっています。

```
/WEB-INF/classes
/Web-INF/lib
/_pages
```

重要： WEB-INF ディレクトリにあるクラスを OracleJSP のクラス・ローダーのかわりにシステムのクラス・ローダーでロードする場合は、クラスを Web サーバーの CLASSPATH にも挿入します。システムのクラス・ローダーの方が優先されます。両方の CLASSPATH に挿入されたクラスは、いずれも常にシステムのクラス・ローダーによりロードされます。

classes ディレクトリは、個々の Java .class ファイルの格納場所です。これらのクラスは、Java パッケージのネーミング規則に従って classes ディレクトリのサブディレクトリに格納する必要があります。

たとえば、LottoBean という JavaBeans があり、そのコードでロード先が oracle.jsp.sample.lottery パッケージとして定義されているとします。OracleJSP では、LottoBean.class がアプリケーション・ルートに対する次の相対位置で検索されます。

```
/WEB-INF/classes/oracle/jsp/sample/lottery/LottoBean.class
```

lib ディレクトリは .jar ファイルの格納場所です。Java パッケージの構造は .jar ファイル構造に指定されているため、.jar ファイルはすべて直接（サブディレクトリではなく）lib ディレクトリに格納されます。

前述の例では、LottoBean.class は、次のようにアプリケーション・ルートに対する相対位置にある lottery.jar に格納できます。

```
/WEB-INF/lib/lottery.jar
```

アプリケーション・ルート・ディレクトリは、次のどのディレクトリにでも配置できます（Web サーバーおよびサーブレット環境に応じて該当する場合）。各ディレクトリは検索順に記載されています。

- このアプリケーションのマップ先となる Web サーバーのディレクトリ
- Web サーバーのドキュメント・ルート・ディレクトリ
- globals.jsa ファイルが格納されているディレクトリ（該当する場合、通常はサーブレット 2.0 環境の場合）

注意：

- 一部の Web サーバー、特に、サーブレット 2.0 仕様をサポートしている Web サーバーには、全面的なサーブレット・コンテキスト機能など、十分なアプリケーション・サポート機能がありません。この場合や、アプリケーション・マッピングが使用されない場合、デフォルト・アプリケーションはサーバー自体で、アプリケーション・ルートは Web サーバーのドキュメント・ルートとなります。
 - 従来のサーブレット環境の場合、globals.jsa ファイルはアプリケーション・ルートを設定するためのアプリケーション・マーカーとして使用できる Oracle の拡張機能です。5-34 ページの「[サーブレット 2.0 に対する OracleJSP のアプリケーションおよびセッションのサポート](#)」を参照してください。
-

OracleJSP の classpath 構成パラメータ

OracleJSP の CLASSPATH に追加するには、その classpath 構成パラメータを使用します。

このパラメータの詳細は、A-13 ページの「[OracleJSP の構成パラメータ（非 OSE）](#)」を参照してください。

このパラメータを Apache/JServ、JSWDK および Tomcat 環境で設定する方法は、A-23 ページの「[OracleJSP の構成パラメータ設定](#)」を参照してください。

OracleJSP クラス・ローダーのメリットとデメリット

OracleJSP のクラス・ローダーを使用すると、次のようなメリットとデメリットがあります。

- 他のクラス・ローダーによりロードされたクラスから OracleJSP によりロードされたクラスへのアクセスの制限

クラスが OracleJSP のクラス・ローダーによりロードされた場合、その定義は OracleJSP のクラス・ローダーにのみ存在します。システムのクラス・ローダーや、サーブレットのような他のクラス・ローダーによりロードされたクラスの場合、アクセスが限定的になります。他のクラス・ローダーによりロードされたクラスでは、OracleJSP によりロードされたクラスをキャストできず、そのメソッドをコールできません。これは、状況に応じて望ましい場合とそうでない場合があります。

- クラスの自動再ロード

デフォルトでは、class ファイルや JAR ファイルがロード後に変更されると、OracleJSP のクラス・ローダーにより、クラスが OracleJSP の CLASSPATH に自動的に再ロードされます。（たとえば、JSP ページの場合は、動的再変換の結果としてクラスが自動的に再ロードされます。動的再変換は、ページの .jsp ソース・ファイルのタイムスタンプ

の方が、対応するページ実装の `.class` ファイルより新しい場合にデフォルトで発生します。）

通常、これは開発環境でのみメリットとなります。典型的な配布環境では、ソース、クラスおよび JAR ファイルは変更されず、その変更の有無をチェックするのは非効率的です。

詳細は、4-24 ページの「[クラスの動的再ロード](#)」を参照してください。

各ファイルの変更を配布環境でたどる場合、通常は OracleJSP の CLASSPATH を使用する必要はありません。デフォルトでは、`classpath` パラメータは空になっています。

OracleJSP のランタイムの考慮事項（非 OSE のみ）

この項では、実行時に OracleJSP によりページの再変換、ページの再ロードおよびクラスの再ロードが実行される条件について説明します。この説明は、Oracle Servlet Engine で実行される JSP ページには該当しません。

ページの動的再変換

Web アプリケーションの実行中は、JSP ページのソースが変更されるたびに、OracleJSP コンテナによりデフォルトでそのページが自動的に再変換され、再ロードされます。

OracleJSP では、OracleJSP のインメモリ・キャッシュに指定されたページ実装クラス・ファイルの最終変更時刻が、JSP ページのソース・ファイルの最終変更時刻より前かどうかチェックされます。

OracleJSP の `developer_mode` フラグを `false` に設定すると、OracleJSP で再変換のためにタイムスタンプがチェックされることによるオーバーヘッドを回避できます。これは、ソース・ファイルとクラス・ファイルが通常は変化しない配布環境ではメリットとなります。このフラグの詳細は、A-13 ページの「[OracleJSP の構成パラメータ（非 OSE）](#)」を参照してください。このフラグの設定方法は、A-23 ページの「[OracleJSP の構成パラメータ設定](#)」を参照してください。

注意：

- クラス・ファイルの最終変更時刻にはインメモリ値が使用されるため、ファイル・システムからページ実装クラス・ファイルを削除すると、OracleJSP では関連する JSP ページのソースが再変換されなくなるので注意してください。OracleJSP で再変換されるのは、JSP ページのソース・ファイルのタイムスタンプが変化した場合のみです。
 - クラス・ファイルは、キャッシュが消滅すると再生成されます。この再生成は、サーバーの再起動後や、このアプリケーションの別のページの再変換後に、このページに要求がダイレクトされるたびに発生します。
-

ページの動的再ロード

OracleJSP コンテナにより JSP ページが自動的に再ロード（つまり、生成されたページ実装クラスが再ロード）されるのは、次のような場合です。

- ページが再変換された場合
(前項「[ページの動的再変換](#)」を参照してください。)
- そのページでコールされ、OracleJSP のクラス・ローダー（システムのクラス・ローダーではなく）によりロードされた Java クラスが変更された場合
(次項「[クラスの動的再ロード](#)」を参照してください。)
- 同じアプリケーション内のいずれかのページが再ロードされる場合

JSP ページは、それが実行される Web アプリケーション全体に対応付けられています（特定のアプリケーションに対応付けられていない JSP ページも、「デフォルト・アプリケーション」の一部と見なされます）。

JSP ページが再ロードされるたびに、アプリケーションのすべての JSP ページが再ロードされます。

注意：

- OracleJSP では、静的に挿入されるファイルに変更があったという理由のみでは、ページは再ロードされません。（`<%@ include %>` 構文を介して静的に挿入されるファイルは、変換時に挿入されます。）
 - ページ再ロードとページ再変換は、異なる処理です。再ロードされても、再変換されるとは限りません。
-
-

クラスの動的再ロード

デフォルトでは、OracleJSP のクラス・ローダーでロードされた Java クラスを実行する要求が OracleJSP によりディスパッチされる前に、クラス・ファイルが初回ロード後に変更されたかどうかチェックされます。クラスに変更があった場合は、OracleJSP のクラス・ローダーにより再ロードされます。

これは、次のような OracleJSP の CLASSPATH 内のクラスにのみ適用されます。

- WEB-INF/lib ディレクトリ内の JAR ファイル
- WEB-INF/classes ディレクトリ内の .class ファイル
- OracleJSP の classpath 構成パラメータを介して指定されたパスにあるクラス
- _pages 出力ディレクトリ内で生成された .class ファイル

「[ページの動的再ロード](#)」で前述したように、クラスが再ロードされると、そのクラスを参照する JSP ページが動的に再ロードされます。

重要：

- クラスを動的に再ロードするには、システムの CLASSPATH ではなく JSP の CLASSPATH に置く必要があるため注意してください。システムの CLASSPATH にも含まれていると、システムのクラス・ローダーが優先され、JSP の自動再ロード機能の妨げとなる場合があります。
 - クラスの動的再ロードには、大量の CPU が使用されることがあります。この機能は、OracleJSP の `developer_mode` パラメータを `false` に設定すると無効化できます。これは、クラスの変化が予想されていない配布環境に適しています。
-
-

classpath および developer_mode 構成パラメータと設定方法の詳細は、A-13 ページの「[OracleJSP の構成パラメータ（非 OSE）](#)」および A-23 ページの「[OracleJSP の構成パラメータ設定](#)」を参照してください。

Oracle Servlet Engine に関する考慮事項

Oracle Servlet Engine (OSE) は、Oracle8i JVM 環境と統合されています。JSP ページを OSE 内で実行するには、データベースにロードして公開する必要があります。JSP ページを Oracle8i に配布する方法の詳細は、第 6 章「[JSP の変換と配布](#)」を参照してください。この項では、OSE 環境に関するプログラミング上の特記事項と、重要な OSE 特性の概要について説明します。

JSP アプリケーションを OSE で実行するには、Apache により駆動される Oracle HTTP Server をフロントエンド Web サーバーとして使用する方法（通常の推奨方法）と、OSE を Web サーバーとして直接使用する方法があります。2-7 ページの「[Oracle Web Application のデータベース・アクセス方法](#)」を参照してください。Oracle8i リリース 8.1.7 のインストール時には、Oracle HTTP Server がデフォルトの Web サーバーとして設定されます。この設定の変更が必要な場合は、インストール手順を参照してください。

Oracle Servlet Engine で実行される JSP ページがデータベース・アクセスを意図しているものと想定されるため、Oracle8i JVM でのデータベース接続に関する一部の背景情報も取り上げます。

通常、JSP コードは、OSE 環境と OracleJSP が使用される他の環境間で全面的に移植可能です。ただし、サーバー側 JDBC 内部ドライバを介した Oracle8i JVM での接続には違いがあります（接続文字列を必要としないなど）。4-26 ページの「[Oracle8i JVM の接続](#)」を参照してください。

Oracle8i JVM のデータベース接続コードや他の Oracle8i JVM 固有の機能が使用されることを除き、OSE 向けに記述された JSP ページは OracleJSP を実行中の他の環境に移植できます。オリジナル・コードの修正と再変換が必要になるのは、Oracle8i JVM 固有の機能が使用された場合のみです。

この章の内容は、次のとおりです。

- [Oracle8i JVM の JVM とサーバー側 JDBC 内部ドライバの概要](#)
- [Oracle8i JVM の接続](#)
- [Oracle Servlet Engine による JNDI の使用](#)
- [OracleJSP のランタイム構成パラメータの等価コード](#)

注意： この項では、OSE をターゲットとする開発上の考慮事項について説明します。ホットロードされるクラスや、クライアント側とサーバー側の変換の比較など、配布上の考慮事項については、6-11 ページの「[Oracle8i への配布時の機能とロジスティックの概要](#)」を参照してください。

Oracle8i JVM の JVM とサーバー側 JDBC 内部ドライバの概要

Oracle8i JVM のデータベース・セッションごとに、専用の Java 仮想マシンが起動します。このセッションと JVM 間の 1 対 1 の対応関係に注意する必要があります。

通常、ターゲットとなる Oracle8i データベースの JVM 内で実行される Java プログラムは、いずれもサーバー側 JDBC 内部ドライバを使用してローカル SQL エンジンにアクセスします。このドライバは、本来、Oracle8i データベースと JVM に連結されており、データベースと同じプロセスの一部として実行されます。また、デフォルトのデータベース・セッション、つまり JVM が起動されたのと同じセッション内でも実行されます。

サーバー側内部ドライバは、データベース・サーバー内で実行されるように最適化されており、ローカル・データベース上の SQL データと PL/SQL サブプログラムへの直接アクセスを提供します。JVM 全体が、データベースおよび SQL エンジンと同じアドレス空間で動作します。SQL エンジンへのアクセスはファンクション・コールであり、ネットワークは介在しません。これにより、JDBC プログラムのパフォーマンスが強化され、SQL エンジンにアクセスするためにリモート Net8 コールを実行する場合に比べてはるかに高速になります。

Oracle8i JVM の接続

サーバー側 JDBC 内部ドライバはデフォルトのデータベース・セッション内で実行されるため、すでにデータベースに暗黙的に「接続済み」になっています。デフォルト接続へのアクセスには、次のどちらかの JDBC メソッドを使用できます。

- `OracleDriver` クラスの `Oracle` 固有の `defaultConnection()` メソッドを使用します。（このメソッドは、コールされるたびに同じ接続オブジェクトを戻します。）
- URL 文字列として `jdbc:oracle:kprb` または `jdbc:default:connection` を指定し、静的な `DriverManager.getConnection()` メソッドを使用します。（このメソッドは、コールされるたびに異なる接続オブジェクトを戻します。）

通常は、`defaultConnection()` メソッドを使用することをお薦めします。

また、内部接続（Java コードを実行するデータベースへの接続）には、サーバー側 Thin ドライバを使用することもできますが、これは例外的です。

注意：

- その他、OracleJSP で提供されるカスタム JavaBeans を使用して接続する方法もあります。5-12 ページの「[Oracle Database-Access JavaBeans](#)」を参照してください。
 - サーバー側内部ドライバで接続するように `OracleDriver` クラスを登録する必要はありませんが、登録しても問題はありません。これは、接続に `getConnection()` を使用する場合も `defaultConnection()` を使用する場合も当てはまります。
-

Oracle JDBC を介したサーバー側接続の詳細は、『Oracle8i 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。

OracleDriver クラスの defaultConnection() メソッドを使用した接続

oracle.jdbc.driver.OracleDriver クラスの defaultConnection() メソッドは、内部データベース接続の確立に使用できる Oracle の拡張機能です。このメソッドでは、常に同じ接続オブジェクトが戻されます。戻される接続オブジェクトを異なる変数名に割り当てて、このメソッドを複数回コールする場合も、単一の接続オブジェクトが再使用されます。

defaultConnection() メソッドは、接続文字列を取りません。次に例を示します。

```
import java.sql.*;
import oracle.jdbc.driver.*;

class JDBCConnection
{
    public static Connection connect() throws SQLException
    {
        Connection conn = null;
        try {
            // connect with the server-side internal driver
            OracleDriver ora = new OracleDriver();
            conn = ora.defaultConnection();
        }

        } catch (SQLException e) {...}
        return conn;
    }
}
```

この例には conn.close() コールがないことに注意してください。JDBC コードがターゲット・サーバー内で実行される場合、接続はクライアントからのような明示的接続インスタンスではなく、暗黙的なデータ・チャネルとなります。通常、この接続はクローズしません。

close() メソッドをコールする場合の注意事項は、次のとおりです。

- defaultConnection() メソッドを介して取得されるすべての接続インスタンスは、実際には同じ接続オブジェクトを参照します。この接続インスタンスは、クローズされ、その後は使用できなくなります。必要に応じて状態とリソースがクリーン・アップされます。その後に defaultConnection() を実行すると、新規の接続オブジェクトと新規のトランザクションが取得されます。
- 接続オブジェクトがクローズされても、データベースへの暗黙的な接続はクローズされません。

DriverManager.getConnection() メソッドを使用した接続

defaultConnection() メソッドを使用して内部データベース接続を確立するかわりに、次のどちらかの接続文字列を指定して静的な DriverManager.getConnection() メソッドを使用できます。

```
Connection conn = DriverManager.getConnection("jdbc:oracle:kprb:");
```

または

```
Connection conn = DriverManager.getConnection("jdbc:default:connection:");
```

URL 文字列に挿入したユーザー名やパスワードは、サーバーのデフォルト接続への接続時には無視されます。

DriverManager.getConnection() メソッドは、コールされるたびに新規の Java Connection オブジェクトを戻します。このメソッドでは新規の物理接続が作成されるのではなく（単一の暗黙的接続のみが使用され）、新規オブジェクトが戻されることに注意してください。

「型マップ」と呼ばれるオブジェクト・マップを操作する場合は、DriverManager.getConnection() メソッドはコールされるたびに新規の接続オブジェクトを戻すという事実が重要になります。Oracle SQL のオブジェクト型を Java クラスにマップするための型マップは、特定の Connection オブジェクトおよびその一部であるステータスに対応付けられています。プログラムの一部として複数の型マップを使用する場合は、getConnection() をコールして型マップごとに新規の Connection オブジェクトを作成できます。型マップの概要は、『Oracle8i 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。

サーバー側 Thin ドライバを使用した接続

通常、Oracle JDBC のサーバー側 Thin ドライバは、あるデータベースから別のデータベースへの接続用に使用します。ただし、サーバー側 Thin ドライバは内部接続にも使用できます。Oracle JDBC Thin ドライバを使用する場合は接続文字列を指定してください。

この機能により、Oracle Servlet Engine と他のサーブレット環境間でコードの移植性というメリットが得られる場合があります。ただし、サーバー側内部ドライバの方がパフォーマンスは向上します。

サーバー側内部ドライバでの自動コミット機能の欠落

JDBC の自動コミット機能は、サーバー側内部ドライバでは使用できません。変更は手動でコミットまたはロールバックする必要があります。

サーバー側内部ドライバでの接続プーリングまたはキャッシュの欠落

サーバー側内部ドライバを使用する場合は、単一の暗黙的データベース接続が使用されるため、接続プーリングおよびキャッシュは使用できません。これらの機能を内部ドライバ経由で使用しようとする、実際にパフォーマンスが低下する場合があります。

Oracle Servlet Engine による JNDI の使用

Oracle Servlet Engine では JNDI メカニズムを使用して「公開」されている JSP ページとサーブレットが検索されますが、通常、このメカニズムは JSP 開発者やユーザーには表示されません。OSE の配布時に JSP ページを公開するには、Oracle セッション・シェルの `publishjsp` コマンド（サーバー側変換を伴う配布用）または `publishservlet` コマンド（クライアント側変換を伴う配布用）を実行する必要があります。

`publishservlet` コマンドを使用する場合は、ページ実装クラスの仮想パス名とサーブレット名を指定する必要があります。仮想パス名は、URL を介してページを起動するため、または OSE 内で実行中の他のページを挿入するためや他のページに転送するために使用されます。

`publishjsp` コマンドの場合は、コマンドラインで仮想パス名とサーブレット名を指定できます。コマンドラインで指定しなければ、指定した JSP ソース・ファイル名とディレクトリ・パスから推論されます。

サーブレット名と仮想パス名はどちらも Oracle8i JVM の JNDI ネームスペースに入力されますが、JSP 開発者やユーザーが注意する必要があるのは仮想パス名のみです。

OSE 用の JSP ページを公開する方法の詳細は、6-39 ページの「[Oracle8i での JSP ページの変換と公開（セッション・シェルの `publishjsp`）](#)」（サーバー側変換を伴う配布の場合）、または 6-59 ページの「[Oracle8i での変換済み JSP ページの公開（セッション・シェルの `publishservlet`）](#)」（クライアント側変換を伴う配布の場合）を参照してください。

Oracle Servlet Engine による JNDI の使用方法の概要は、『Oracle8i Oracle Servlet Engine ユーザーズ・ガイド』を参照してください。

OracleJSP のランタイム構成パラメータの等価コード

OracleJSP の構成パラメータには、変換時に有効になるものと実行時（ランタイム）に有効になるものがあります。JSP ページを Oracle8i データベースに配布して Oracle Servlet Engine 内で実行する場合は、OracleJSP の事前変換ツールのコマンドライン・オプションを介して、適切な変換時設定を指定できます。

ただし、Oracle Servlet Engine の実行時には、ランタイム構成パラメータはサポートされません。最も重要なランタイム構成パラメータは、NLS 関連の `translate_params` です。等価コードの詳細は、8-6 ページの「[translate_params 構成パラメータの等価コード](#)」を参照してください。

Apache/JServ サブレット環境に関する考慮事項

Oracle Internet Application Server リリース 1.0.x などの Apache/JServ ベース・プラットフォームで OracleJSP を実行する場合は、サブレット 2.0 環境であるため、特に考慮が必要な事項があります。サブレット 2.0 仕様では、サブレット 2.1 および 2.2 環境で使用可能な一部の重要な機能に対するサポートが欠落していました。

Apache/JServ 環境を OracleJSP 用に構成する方法については、次の各項を参照してください。

- [A-7 ページの「Web サーバーの CLASSPATH への OracleJSP 関連の JAR および ZIP ファイルの追加」](#)
- [A-9 ページの「Oracle JspServlet への JSP ファイル名拡張子のマッピング」](#)
- [A-23 ページの「Apache/JServ での OracleJSP パラメータの設定」](#)

(Oracle プラットフォームを介して Apache/JServ を使用する場合は、そのプラットフォームのインストールおよび構成ドキュメントを参照してください。)

この項の残りの部分では、Oracle Internet Application Server による Apache/JServ の使用の概要と、次の Apache 固有の考慮事項について説明します。

- [Apache/JServ における動的挿入および転送](#)
- [Apache/JServ 用のアプリケーション・フレームワーク](#)
- [JSP とサブレットのセッション共有](#)
- [ディレクトリ別名の変換](#)

Oracle Internet Application Server による Apache/JServ の使用

Oracle Internet Application Server リリース 1.0.0 および 1.0.1 では、この製品にはサブレット環境として Apache/JServ が使用されます。

Apache/JServ 環境や他のサブレット 2.0 環境の場合と同様に、Oracle Internet Application Server リリース 1.0.x を使用する場合には、サブレットと JSP の使用に関して特に考慮が必要な事項があります。ここでは、その詳細について説明します。

(Oracle Internet Application Server には、Apache により駆動される Oracle HTTP Server が Web サーバーとして組み込まれています。Oracle HTTP Server の `mod_ose` Apache mod を使用して Oracle Servlet Engine 内で JSP アプリケーションを実行する場合は、Oracle Internet Application Server の Apache/JServ サブレット 2.0 環境ではなく、OSE のサブレット 2.2 環境を使用していることに注意してください。)

注意： 将来のリリースの Oracle HTTP Server および Oracle Internet Application Server では、Apache/JServ 以外のサブレット環境も使用できるようになります。

Oracle Internet Application Server と Oracle HTTP Server 使用の概要は、2-3 ページの「[Oracle 環境における OracleJSP のサポート](#)」を参照してください。

Apache/JServ における動的挿入および転送

JSP の動的挿入 (`jsp:include` アクション) および転送 (`jsp:forward` アクション) は、サブレット 2.0 環境ではなくサブレット 2.1 および 2.2 環境に存在する要求ディスパッチ機能に依存します。

ただし、OracleJSP には、Apache/JServ および他のサブレット 2.0 環境で JSP ページ間、または JSP ページから静的 HTML ファイルへの、動的挿入および転送を可能にする拡張機能が用意されています。

ただし、この OracleJSP 機能では、サブレットへの動的転送や挿入はできません。(サブレットの実行は、OracleJSP コンテナではなく JServ または他のサブレット・コンテナにより制御されます。)

サブレットへの挿入または転送が必要な場合は、サブレットのラッパーとして動作する JSP ページを作成します。

次の例は、サブレットと、そのラッパーとして動作する JSP ページを示しています。Apache/JServ 環境では、JSP ラッパー・ページに挿入または転送すると、サブレットに効率的に挿入または転送できます。

サブレット・コード 次のサブレットに挿入または転送するとします。

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

public class TestServlet extends HttpServlet {

    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
        System.out.println("initialized");
    }

    public void destroy()
    {
        System.out.println("destroyed");
    }
}
```

```

public void service
    (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML><BODY>");
    out.println("TestServlet Testing");
    out.println("<H3>The local time is: " + new java.util.Date());
    out.println("</BODY></HTML>");
}
}

```

JSP のラッパー・ページ・コード 前述のサブレット用に次の JSP ラッパー (wrapper.jsp) を作成できます。

```

<!-- wrapper.jsp--wraps TestServlet for JSP include/forward --%>
<%@ page isThreadSafe="true" import="TestServlet" %>
<%!
    TestServlet s=null;
    public void jspInit() {
        s=new TestServlet();
        try {
            s.init(this.getServletConfig());
        } catch (ServletException se)
        {
            s=null;
        }
    }
    public void jspDestroy() {
        s.destroy();
    }
%>
<% s.service(request,response); %>

```

サブレット 2.0 環境で wrapper.jsp に挿入または転送すると、サブレット 2.1 または 2.2 環境で TestServlet に直接挿入または転送するのと同じ効果が得られます。

注意：

- ラッパーJSPページ内でisThreadSafeをtrueに設定するかfalseに設定するかは、元のサブレットがスレッド・セーフかどうかによって決まります。
 - この状況でラッパー JSP ページを使用するかわりに、元の JSP ページ (include または forward の発生元となるページ) に HTTP クライアント・コードを追加する方法もあります。標準 java.net.URL クラスのインスタンスを使用すると、元の JSP ページからサブレットへの HTTP 要求を作成できます。(この使用例では、セッション・データやセキュリティ資格証明は共有できないことに注意してください。) また、Innovation GmbH の HTTPClient クラスを使用する方法もあります。Oracle® JVM にはこのクラスの修正版が用意されており、URL に https:// を使用する場合は、SSL が直接、またはプロキシを介してサポートされます。(このクラスの概要は、www.innovation.ch/java/HTTPClient を参照してください。「Getting Started」をクリックすると、JDK の HTTP クライアントを HTTPClient クラスに置き換える方法などが表示されます。) これらの代替方法の詳細はこのマニュアルでは取り扱いませんが、通常このアプローチはお薦めしません。
-

Apache/JServ 用のアプリケーション・フレームワーク

サブレット 2.0 仕様では、後続の仕様で提供されるアプリケーション・サポートのための完全なサブレット・コンテキスト・フレームワークが提供されません。

Apache/JServ など、サブレット 2.0 環境の場合、OracleJSP では globals.jsa ファイルを使用して独自のアプリケーション・フレームワークが提供されます。このファイルは、アプリケーション・マーカーとして使用できます。

詳細は、5-36 ページの「globals.jsa を介した個別アプリケーションおよびセッション」を参照してください。

JSP とサブレットのセッション共有

Apache/JServ 環境の JSP ページとサブレット間で HTTP セッション情報を共有するには、oracle.jsp.JspServlet (OracleJSP コンテナのフロントエンドとして機能するサブレット) が、JSP ページのセッション共有対象となるサブレットと同じゾーンに含まれるように、環境を構成する必要があります。詳細は、Apache のドキュメントを参照してください。

ゾーン設定が適切かどうかを検証するために、一部のブラウザでは cookie に関する警告を有効化できます。Apache 環境では、cookie 名にゾーン名が含まれています。

また、`globals.jsa` ファイルを使用するアプリケーションの場合は、サブレットから JSP のセッション・データにアクセスできるように、OracleJSP の構成パラメータ `session_sharing` を `true` (デフォルト) に設定する必要があります。関連情報については、次の各項を参照してください。

- 5-34 ページの「サブレット 2.0 に対する OracleJSP のアプリケーションおよびセッションのサポート」
- A-13 ページの「OracleJSP の構成パラメータ (非 OSE)」
- A-23 ページの「OracleJSP の構成パラメータ設定」

ディレクトリ別名の変換

Apache では、`httpd.conf` 構成ファイルの `Alias` コマンドを介して「仮想ディレクトリ」を作成できるようにすることで、ディレクトリ別名機能をサポートしています。これにより、Web ドキュメントをデフォルトのドキュメント・ルート・ディレクトリ以外の場所に配置できます。(Web サーバーのドキュメント・ルートと各別名ルート用に、暗黙的アプリケーションが作成されます。)

サンプルとして、`httpd.conf` に次のエントリがあるとします。

```
Alias /icons/ "/apache/apache139/icons/"
```

このコマンドでは、`icons` を `/apache/apache139/icons/` パスの別名として使用できるようになります。このコマンドを使用して、たとえば次の URL を指定すると、ファイル `/apache/apache139/icons/art.gif` にアクセスできます。

```
http://host[:port]/icons/art.gif
```

ただし、現在、この機能はサブレットと JSP ページでは正常に機能しません。これは、Apache/JServ の `getRealPath()` メソッドが、別名ディレクトリにあるファイルを処理するときに不正な値を戻すためです。

OracleJSP には、Apache 固有の構成パラメータ `alias_translation` が用意されています。このパラメータにより、`alias_translation=true` に設定すると (デフォルト設定は `false`)、前述の制限事項が回避されます。

Apache/JServ 環境で OracleJSP の構成パラメータを設定する方法は、A-23 ページの「[Apache/JServ での OracleJSP パラメータの設定](#)」を参照してください。

OracleJSP の拡張機能

この章では、OracleJSP の拡張機能について次のトピックで説明します。

- [移植可能な OracleJSP プログラミング拡張機能](#)
- [Oracle 固有のプログラミング拡張機能](#)
- [サーブレット 2.0 に対する OracleJSP のアプリケーションおよびセッションのサポート](#)

移植可能な拡張機能は、Oracle の JSP マークアップ言語 (JML) カスタム・タグ、JML 拡張データ型、SQL カスタム・タグおよび Database-Access JavaBeans を介して提供されます。これらの機能は、他の JSP 環境で使用できます。

移植できない拡張機能とは、変換と実行に OracleJSP を必要とするものです。

サーブレット 2.0 環境向けの拡張アプリケーションおよびセッション・サポート機能は、Oracle `globals.jsa` の機能性を介して提供され、同じく OracleJSP が必要です。

移植可能な OracleJSP プログラミング拡張機能

この項で説明する Oracle の拡張機能は、Oracle JSP マークアップ言語 (JML) のサンプル・タグ・ライブラリまたはカスタム JavaBeans を介して実装されます。これらの拡張機能は、すべての標準 JSP 環境に移植可能です。たとえば、次の拡張機能があります。

- JML の拡張データ型
- XML および XSL のサポート (JML タグを含む)
- Database-Access JavaBeans
- JML SQL タグ

重要： JML 機能の使用方法は、7-18 ページの「[JSP マークアップ言語 \(JML\) のサンプル・タグ・ライブラリの概要](#)」を参照してください。

JML の拡張データ型

Java のプリミティブ型と `java.lang` のラッパー型 (2-11 ページの「[OracleJSP の拡張データ型](#)」を参照) における JSP の使用方法の欠点を補うために、OracleJSP では `oracle.jsp.jml` パッケージに次の JavaBeans クラスが用意されており、最も一般的な Java データ型のラッパーとして機能します。

- `boolean` 値を表す `JmlBoolean`
- `int` 値を表す `JmlNumber`
- `double` 値を表す `JmlFPNumber`
- `String` 値を表す `JmlString`

これらの各クラスは単一の属性である `value` を持ち、値の取得、様々な形式の入力からの値の設定、値が複数の形式のいずれかで指定された値と等しいかどうかのテスト、および値から文字列への変換を行うメソッドが含まれています。

また、`getValue()` および `setValue()` メソッドを使用するかわりに、他の Bean の場合と同様に `jsp:getProperty` および `jsp:setProperty` タグを使用できます。

次の例では、`application` 有効範囲を持つ `count` という名前の `JmlNumber` インスタンスを作成しています。

```
<jsp:useBean id="count" class="oracle.jsp.jml.JmlNumber" scope="application" />
```

この値が後に他の場所で設定された場合は、その値に次のようにアクセスできます。

```
<h3> The current count is <%=count.getValue() %> </h3>
```

次の例では、request 有効範囲を持つ maxSize という JmlNumber インスタンスを作成し、setProperty を使用して設定しています。

```
<jsp:useBean id="maxSize" class="oracle.jsp.jml.Number" scope="request" >
    <jsp:setProperty name="maxSize" property="value" value="<%= 25 %>" />
</jsp:useBean>
```

これ以降は、4つの拡張データ型クラスのパブリック・メソッドについて説明し、続いて例を示します。

JmlBoolean 型

JmlBoolean オブジェクトは、Java の boolean 値を表します。

getValue() および setValue() メソッドは、Bean の value プロパティを Java の boolean 値として取得または設定します。

- boolean getValue()
- void setValue(boolean)

setTypedValue() メソッドには複数のシグネチャがあり、Bean の value プロパティを文字列（「true」または「false」など）、java.lang.Boolean 値、Java の boolean 値または JmlBoolean 値から設定できます。文字列入力の場合、文字列の変換は標準の java.lang.Boolean.valueOf() メソッドと同じルールに従って実行されます。

- void setTypedValue(String)
- void setTypedValue(Boolean)
- void setTypedValue(boolean)
- void setTypedValue(JmlBoolean)

equals() メソッドは、Bean の value プロパティが、指定された Java の boolean 値と等しいかどうかをテストします。

- boolean equals(boolean)

typedEquals() メソッドには複数のシグネチャがあり、Bean の value プロパティの値が、指定された文字列（「true」または「false」など）、java.lang.Boolean 値または JmlBoolean 値と等しいかどうかをテストします。

- boolean typedEquals(String)
- boolean typedEquals(Boolean)
- boolean typedEquals(JmlBoolean)

toString() メソッドは、Bean の value プロパティを java.lang.String 値（「true」または「false」）として戻します。

- String toString()

JmlNumber 型

JmlNumber オブジェクトは、Java の `int` 値と等価の 32 ビット数を表します。

`getValue()` および `setValue()` メソッドは、Bean の `value` プロパティを Java の `int` 値として取得または設定します。

- `int getValue()`
- `void setValue(int)`

`setTypedValue()` メソッドには複数のシグネチャがあり、Bean の `value` プロパティを文字列、`java.lang.Integer` 値、Java の `int` 値または `JmlNumber` 値から設定できます。文字列入力の場合、文字列の変換は標準 `java.lang.Integer.decode()` メソッドと同じルールに従って実行されます。

- `void setTypedValue(String)`
- `void setTypedValue(Integer)`
- `void setTypedValue(int)`
- `void setTypedValue(JmlNumber)`

`equals()` メソッドは、Bean の `value` プロパティが、指定された Java の `int` 値と等しいかどうかをテストします。

- `boolean equals(int)`

`typedEquals()` メソッドには複数のシグネチャがあり、Bean の `value` プロパティの値が、指定された文字列（「1234」など）、`java.lang.Number` 値または `JmlNumber` 値と等しいかどうかをテストします。

- `boolean typedEquals(String)`
- `boolean typedEquals(Integer)`
- `boolean typedEquals(JmlNumber)`

`toString()` メソッドは、Bean の `value` プロパティを等価の `java.lang.String` 値（「1234」など）として戻します。このメソッドの機能性は、標準 `java.lang.Integer.toString()` メソッドと同じです。

- `String toString()`

JmlFPNumber 型

JmlFPNumber オブジェクトは、Java の `double` 値と等価の 64 ビット浮動小数点数を表します。

`getValue()` および `setValue()` メソッドは、Bean の `value` プロパティを Java の `double` 値として取得または設定します。

- `double getValue()`
- `void setValue(double)`

`setTypedValue()` メソッドには複数のシグネチャがあり、Bean の value プロパティを文字列（「3.57」など）、`java.lang.Integer` 値、Java の `int` 値、`java.lang.Float` 値、Java の `float` 値、`java.lang.Double` 値、Java の `double` 値または `JmlFPNumber` 値から設定できます。文字列入力の場合、文字列の変換は標準

`java.lang.Double.valueOf()` メソッドと同じルールに従って実行されます。

- `void setTypedValue(String)`
- `void setTypedValue(Integer)`
- `void setTypedValue(int)`
- `void setTypedValue(Float)`
- `void setTypedValue(float)`
- `void setTypedValue(Double)`
- `void setTypedValue(double)`
- `void setTypedValue(JmlFPNumber)`

`equals()` メソッドは、Bean の value プロパティが、指定された Java の `double` 値と等しいかどうかをテストします。

- `boolean equals(double)`

`typedEquals()` メソッドには複数のシグネチャがあり、Bean の value プロパティの値が、指定された文字列（「3.57」など）、`java.lang.Integer` 値、Java の `int` 値、`java.lang.Float` 値、Java の `float` 値、`java.lang.Double` 値、Java の `double` 値または `JmlFPNumber` 値と等価かどうかをテストします。

- `boolean typedEquals(String)`
- `boolean typedEquals(Integer)`
- `boolean typedEquals(int)`
- `boolean typedEquals(Float)`
- `boolean typedEquals(float)`
- `boolean typedEquals(Double)`
- `boolean typedEquals(JmlFPNumber)`

`toString()` メソッドは、Bean の value プロパティを `java.lang.String` 値（「3.57」など）として戻します。このメソッドの機能性は、標準 `java.lang.Double.toString()` メソッドと同じです。

- `String toString()`

JmlString 型

JmlString オブジェクトは、`java.lang.String` 値を表します。

`getValue()` および `setValue()` メソッドは、Bean の value プロパティを `java.lang.String` 値として取得または設定します。`setValue()` コールの入力が NULL であれば、value プロパティは空（長さ 0）の文字列に設定されます。

- `String getValue()`
- `void setValue(String)`

`toString()` メソッドの機能は、`getValue()` メソッドと等価です。

- `String toString()`

`setTypedValue()` メソッドは、指定された JmlString 値に従って Bean の value プロパティを設定します。JmlString 値が NULL であれば、value プロパティは空（長さ 0）の文字列に設定されます。

- `void setTypedValue(JmlString)`

`isEmpty()` メソッドは、Bean の value プロパティが空（長さ 0）の文字列 "" かどうかをテストします。

- `boolean isEmpty()`

`equals()` メソッドには 2 つのシグネチャがあり、Bean の value プロパティが指定された `java.lang.String` 値と等しいか、JmlString 値と等しいかをテストします。

- `boolean equals(String)`
- `boolean equals(JmlString)`

JML のデータ型の例

この例は、有効範囲内で単純なデータ型を管理するための、JML のデータ型の JavaBeans の使用方法を示しています。このページでは、JML の型ごとに 1 つずつ、4 つのセッション・オブジェクトが宣言されています。このページは、各型の値を入力できるフォームを表します。新規の値が発行されると、フォームには新規の値と以前に設定されていた値の両方が表示されます。この出力の生成プロセス中に、ページによりセッション・オブジェクトが新規フォームの値で更新されます。

```
<jsp:useBean id = "submitCount" class = "oracle.jsp.jml.JmlNumber" scope = "session" />
```

```
<jsp:useBean id = "bool" class = "oracle.jsp.jml.JmlBoolean" scope = "session" >  
    <jsp:setProperty name = "bool" property = "value" param = "fBoolean" />  
</jsp:useBean>
```

```
<jsp:useBean id = "num" class = "oracle.jsp.jml.JmlNumber" scope = "session" >  
    <jsp:setProperty name = "num" property = "value" param = "fNumber" />  
</jsp:useBean>
```

```

<jsp:useBean id = "fnum" class = "oracle.jsp.jml.JmlFPNumber" scope = "session" >
    <jsp:setProperty name = "fnum" property = "value" param = "fFPNumber" />
</jsp:useBean>

<jsp:useBean id = "str" class = "oracle.jsp.jml.JmlString" scope = "session" >
    <jsp:setProperty name = "str" property = "value" param = "fString" />
</jsp:useBean>

<HTML>

<HEAD>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html;CHARSET=iso-8859-1">
    <META NAME="GENERATOR" Content="Visual Page 1.1 for Windows">
    <TITLE>OracleJSP Extended Datatypes Sample</TITLE>
</HEAD>

<BODY BACKGROUND="images/bg.gif" BGCOLOR="#FFFFFF">

<% if (submitCount.getValue() > 1) { %>
    <h3> Last submitted values </h3>
    <ul>
        <li> bool: <%= bool.getValue() %>
        <li> num: <%= num.getValue() %>
        <li> fnum: <%= fnum.getValue() %>
        <li> string: <%= str.getValue() %>
    </ul>
<% }

    if (submitCount.getValue() > 0) { %>

        <jsp:setProperty name = "bool" property = "value" param = "fBoolean" />
        <jsp:setProperty name = "num" property = "value" param = "fNumber" />
        <jsp:setProperty name = "fnum" property = "value" param = "fFPNumber" />
        <jsp:setProperty name = "str" property = "value" param = "fString" />

        <h3> New submitted values </h3>
        <ul>
            <li> bool: <jsp:getProperty name="bool" property="value" />
            <li> num: <jsp:getProperty name="num" property="value" />
            <li> fnum: <jsp:getProperty name="fnum" property="value" />
            <li> string: <jsp:getProperty name="str" property="value" />
        </ul>
    <% } %>

```

```
<jsp:setProperty name = "submitCount" property = "value" value = "<%= submitCount.getValue() + 1 %>"
/>

<FORM ACTION="index.jsp" METHOD="POST" ENCTYPE="application/x-www-form-urlencoded">
<P> <pre>
  boolean test: <INPUT TYPE="text" NAME="fBoolean" VALUE="<%= bool.getValue() %>" >
    number test: <INPUT TYPE="text" NAME="fNumber" VALUE="<%= num.getValue() %>" >
fpnumber test: <INPUT TYPE="text" NAME="fFPNumber" VALUE="<%= fpnum.getValue() %>" >
  string test: <INPUT TYPE="text" NAME="fString" VALUE="<%= str.getValue() %>" >
</pre>

<P> <INPUT TYPE="submit">

</FORM>

</BODY>

</HTML>
```

OracleJSP の XML および XSL サポート

JSP テクノロジーを使用すると、動的な HTML ページのみでなく動的な XML ページも生成できます。OracleJSP では、次の 2 つの方法で JSP ページでの XML および XSL テクノロジーの使用がサポートされます。

- OracleJSP トランスレータには、標準 XML の代替 JSP 構文を認識するロジックが組み込まれています。
- OracleJSP には、JSP 出力ストリームに XSL スタイルシートを適用するための JML タグが用意されています。

また、Oracle8i では、データベース問合せ時の XML 機能に使用する XML-SQL ユーティリティの一部として、`oracle.xml.sql.query.OracleXMLQuery` クラスが提供されます。このクラスには、ファイル `xsu12.jar` (JDK 1.2.x 用) または `xsu111.jar` (JDK 1.1.x 用) が必要です。この 2 つのファイルは、OracleJSP の Database-Access JavaBeans での XML 機能にも必要で、Oracle8i リリース 8.1.7 で提供されます。

`OracleXMLQuery` を使用する JSP のサンプルについては、9-33 ページの「[XML の問合せ — XMLQuery.jsp](#)」を参照してください。

`OracleXMLQuery` クラスと他の XML - SQL ユーティリティの機能の詳細は、『Oracle8i アプリケーション開発者ガイド — XML』を参照してください。

XML の代替構文

スクリプトレットを表す `<%...%>`、宣言を表す `<%!...%>` および式を表す `<%=...%>` などの JSP タグの構文は、XML ドキュメント内では無効です。Sun Microsystems は、この問題に『JavaServer Pages Specification, Version 1.1』で XML 互換構文を使用して等価 JSP タグを定義することで対処しています。これは、XML ドキュメントの先頭の `jsp:root` 開始タグ内で指定できる標準 DTD を介して実装されます。

この機能により、たとえば XML オーサリング・ツールで XML ベースの JSP ページを作成できます。

OracleJSP では、この DTD が直接使用されることはなく、`jsp:root` タグを使用する必要もありませんが、OracleJSP トランスレータには、標準 DTD に指定された代替構文を認識するロジックが組み込まれています。表 5-1 は、この構文を示しています。

表 5-1 XML の代替構文

標準 JSP 構文	XML 代替 JSP 構文
<code><%@ directive ...%></code> 例： <code><%@ page ... %></code> <code><%@ include ... %></code>	<code><jsp:directive.directive ... /></code> 例： <code><jsp:directive.page ... /></code> <code><jsp:directive.include ... /></code>
<code><%! ... %></code> (宣言)	<code><jsp:declaration></code> <code>...declarations go here...</code> <code></jsp:declaration></code>
<code><%= ... %></code> (式)	<code><jsp:expression></code> <code>...expression goes here...</code> <code></jsp:expression></code>
<code><% ... %></code> (スクリプトレット)	<code><jsp:scriptlet></code> <code>...code fragment goes here...</code> <code></jsp:scriptlet></code>

ほとんどの部分では、`jsp:useBean` などの JSP アクション・タグに、すでに XML 準拠の構文が使用されています。ただし、引用符の表記規則や要求時属性の式などにより、変更が必要な場合があります。

XSL スタイルシート用の JML タグ

動的ページに XML と XSL を多数使用する場合は、結果がクライアントに戻される前にサーバー上で XSL 変換を実行する必要があります。

OracleJSP には、このプロセスを簡素化するために 2 つのシノニムの JML タグが用意されています。次の例のように、JML の `transform` タグまたは JML の `styleSheet` タグを使用します（どちらも効果は同じです）。

```
<jml:transform href="xslRef" >
```

```
    ...Tag body contains regular JSP commands and static text that  
    produce the XML code that the stylesheet is to be applies to...
```

```
</jml:transform >
```

（表記規則により `jml:` 接頭辞が使用されていますが、`taglib` ディレクティブではどの接頭辞でも指定できます。）

重要： JML タグを使用する場合は、7-18 ページの「[JSP マークアップ言語（JML）のサンプル・タグ・ライブラリの概要](#)」を参照してください。

`href` パラメータについては、次の点に注意してください。

- 静的な XSL スタイルシートまたは動的に生成された XSL スタイルシートを参照できます。たとえば、スタイルシートを生成する JSP ページまたはサブルーットを参照できます。
- 完全修飾の URL (`http://host[:port]/yourpath`)、アプリケーション相対の JSP 参照（「/」で始まる参照）またはページ相対の JSP 参照（「/」以外で始まる参照）を指定できます。アプリケーション相対パスとページ相対パスについては、1-8 ページの「[JSP ページの間接要求](#)」を参照してください。
- 動的に指定できます。デフォルトでは、`href` の値は静的な Java 文字列です。ただし、標準 JSP 式構文を使用して、動的に計算される値を指定できます。

通常は、`transform` または `styleSheet` タグを使用してページ全体を変換します。ただし、タグが適用されるのは、本体のうち開始タグと終了タグで囲まれた部分のみです。したがって、各ブロックの境界をそれぞれ `transform` または `styleSheet` タグ・セットで設定し、該当するスタイルシートへの固有の `href` ポインタを指定して、ページ内で個別 XSL ブロックを使用できます。

jml:transform を使用する XSL の例

この項では、サンプル XSL スタイルシートと、jml:transform タグを使用してスタイルシート経由で出力をフィルタ処理するサンプル JSP ページについて説明します。（この例は簡素化されており、ページ内の XML は静的です。より現実的な例では、JSP ページを使用して XML 全体またはその一部を動的に生成してから、変換を実行している場合があります。）

サンプル・スタイルシート : hello.xsl

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="page">
    <html>
      <head>
        <title>
          <xsl:value-of select="title"/>
        </title>
      </head>
      <body bgcolor="#ffffff">
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="title">
    <h1 align="center">
      <xsl:apply-templates/>
    </h1>
  </xsl:template>

  <xsl:template match="paragraph">
    <p align="center">
      <i>
        <xsl:apply-templates/>
      </i>
    </p>
  </xsl:template>

</xsl:stylesheet>
```

サンプル JSP ページ : hello.jsp

```
<%@ page session = "false" %>
<%@ taglib uri="/WEB-INF/jmltaglib.tld" prefix="jml" %>

<jml:transform href="style/hello.xsl" >

<page>
  <title>Hello</title>
  <content>
    <paragraph>This is my first XML/XSL file!</paragraph>
  </content>
</page>

</jml:transform>
```

この例の出力は次のようになります。



Oracle Database-Access JavaBeans

OracleJSP は、Oracle データベースへのアクセスに使用するカスタム JavaBeans のセットを提供します。次の Bean は、oracle.jsp.dbutil パッケージに含まれています。

- ConnBean は、単一のデータベース接続をオープンします。
- ConnCacheBean は、データベース接続に Oracle の接続キャッシング実装を使用します。(JDBC 2.0 が必要です。)
- DBBean はデータベース問合せを実行します。
- CursorBean は、問合せのみでなく、UPDATE、INSERT および DELETE 文とストアード・プロシージャ・コールについて一般的な DML サポートを提供します。

これらの Bean の使用例については、9-20 ページの「[Database-Access JavaBeans のサンプル](#)」を参照してください。

4 つの Bean は、いずれもイベント通知用の OracleJSP の `JspScopeListener` インタフェースを実装します。5-30 ページの「[OracleJSP のイベント処理 — JspScopeListener](#)」を参照してください。

この項は、Oracle JDBC の実務上の知識があることを前提としています。必要であれば、『Oracle8i 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。

重要： Oracle Database-Access JavaBeans を使用するには、ファイル `ojsputil.jar` をインストールして CLASSPATH に挿入します。このファイルは、OracleJSP のインストール時に提供されます。XML 関連のメソッドと機能を使用するには、Oracle8i リリース 8.1.7 で提供されるファイル `xsu12.jar` (JDK 1.2.x 用) または `xsu111.jar` (JDK 1.1.x 用) も必要です。

データベース接続用の ConnBean

単純なデータベース接続（接続プーリングまたはキャッシュを使用しない接続）を確立するには、`oracle.jsp.dbutil.ConnBean` を使用します。

注意：

- 問合せのみの場合は、独自の接続メカニズムを持つ `DBBean` を使用する方が単純です。
 - 接続キャッシュを使用するには、かわりに `ConnCacheBean` を使用します。
 - JSP ページに使用する JavaBeans と同様に、設定メソッドを直接使用するかわりに、`jsp:setProperty` アクションで `ConnBean` のプロパティのいずれかを設定できます。
-

`ConnBean` のプロパティは、次のとおりです。

- `user`（データベース・スキーマのユーザー ID）
- `password`（ユーザー・パスワード）
- `URL`（データベース接続文字列）
- `stmtCacheSize`（Oracle JDBC の文キャッシュ用のキャッシュ・サイズ）

`stmtCacheSize` を設定すると、Oracle JDBC の文キャッシュ機能が使用可能になります。文キャッシュ機能と制限事項の概要は、4-6 ページの「[JDBC 文のキャッシュ](#)」を参照してください。

- `executeBatch` (Oracle JDBC のバッチ更新用のバッチ・サイズ)

`executeBatch` を設定すると、Oracle JDBC のバッチ更新が使用可能になります。この機能の概要は、4-7 ページの「[バッチ更新](#)」を参照してください。

- `preFetch` (Oracle JDBC の行のプリフェッチでプリフェッチされる文の数)

`preFetch` を設定すると、Oracle JDBC の行のプリフェッチ機能が使用可能になります。この機能の概要は、4-7 ページの「[行のプリフェッチ](#)」を参照してください。

ConnBean には、次の設定メソッドと取得メソッドおよびプロパティがあります。

- `void setUser(String)`
- `String getUser()`
- `void setPassword(String)`
- `String getPassword()`
- `void setURL(String)`
- `String getURL()`
- `void setStmtCacheSize(int)`
- `int getStmtCacheSize()`
- `void setExecuteBatch(int)`
- `int getExecuteBatch()`
- `void setPreFetch(int)`
- `int getPreFetch()`

接続のオープンとクローズには、次のメソッドを使用します。

- `void connect()` — ConnBean プロパティ設定を使用してデータベース接続を確立します。
- `void close()` — 接続をクローズし、オープン・カーソルがあればそれもクローズします。

カーソルをオープンして CursorBean オブジェクトを戻すには、次のメソッドを使用します。

- `CursorBean getCursorBean(int, String)`

または

- `CursorBean getCursorBean(int)`

入力は、次のとおりです。

- 必要な JDBC 文の型を指定する int 定数、CursorBean.PLAIN_STMT (Statement オブジェクトの場合)、CursorBean.PREP_STMT (PreparedStatement オブジェクトの場合) または CursorBean.CALL_STMT (CallableStatement オブジェクトの場合) のいずれか 1 つ
- 実行する SQL 操作を指定する文字列 (オプション。SQL 操作は、文を実行する CursorBean メソッドのコールでも指定できます。)

CursorBean の機能の詳細は、5-18 ページの「[DML とストアド・プロシージャ用の CursorBean](#)」を参照してください。

接続キャッシュ用の ConnCacheBean

Oracle JDBC の接続キャッシュ・メカニズム (JDBC 2.0 の接続プーリングを使用) をデータベース接続に使用するには、oracle.jsp.dbutil.ConnCacheBean を使用します。接続キャッシュの概要は、4-5 ページの「[データベース接続のキャッシュ](#)」を参照してください。

注意:

- 単純な接続オブジェクト (接続プーリングまたはキャッシュ機能なし) を使用するには、かわりに ConnBean を使用します。
 - ConnCacheBean により OracleConnectionCacheImpl が拡張され、OracleConnectionCacheImpl により OracleDataSource が拡張されます (どちらも Oracle JDBC パッケージ oracle.jdbc.pool にあります)。これらの Oracle JDBC クラスの詳細は、『Oracle8i 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。
 - JSP ページに使用する JavaBeans と同様に、設定メソッドを直接使用するかわりに、jsp:setProperty アクションで ConnCacheBean のプロパティのいずれかを設定できます。
 - ConnBean とは異なり、ConnCacheBean を使用する場合は、標準 Connection オブジェクトの機能を使用して文オブジェクトを作成し、実行します。
-

ConnCacheBean のプロパティは、次のとおりです。

- user (データベース・スキーマのユーザー ID)
- password (ユーザー・パスワード)
- URL (データベース接続文字列)
- maxLimit (このキャッシュの最大許容接続数)

- minLimit (このキャッシュに存在する最小接続数。接続数がこの数値よりも少ない場合は、キャッシュの「アイドル・プール」にも接続が含まれます。)
- stmtCacheSize (Oracle JDBC の文キャッシュ用のキャッシュ・サイズ)
stmtCacheSize を設定すると、Oracle JDBC の文キャッシュ機能が使用可能になります。Oracle JDBC の文キャッシュ機能と制限事項の概要は、4-6 ページの「[JDBC 文のキャッシュ](#)」を参照してください。
- cacheScheme (次のいずれかのキャッシュ・タイプ)
 - DYNAMIC_SCHEME — 新規にプールされる接続を上限に関係なく作成できますが、提供された論理接続インスタンスが使用されなくなると、即時に自動的にクローズされて解放されます。
 - FIXED_WAIT_SCHEME — 上限に達すると、新規接続は既存の接続オブジェクトが解放されるまで待機します。
 - FIXED_RETURN_NULL_SCHEME — 上限に達すると、接続オブジェクトが解放されるまで、新規接続は失敗します (null が戻されます)。

ConnCacheBean クラスでは、プロパティの次の取得メソッドや設定メソッドなど、Oracle JDBC の OracleConnectionCacheImpl クラスで定義されたメソッドがサポートされます。

- void setUser(String)
- String getUser()
- void setPassword(String)
- String getPassword()
- void setURL(String)
- String getURL()
- void setMaxLimit(int)
- int getMaxLimit()
- void setMinLimit(int)
- int getMinLimit()
- void setStmtCacheSize(int)
- int getStmtCacheSize()
- void setCacheScheme(int)

ConnCacheBean.DYNAMIC_SCHEME、ConnCacheBean.FIXED_WAIT_SCHEME または ConnCacheBean.FIXED_RETURN_NULL_SCHEME を指定します。

- `int getCacheScheme()`

`ConnCacheBean.DYNAMIC_SCHEME`、`ConnCacheBean.FIXED_WAIT_SCHEME` または `ConnCacheBean.FIXED_RETURN_NULL_SCHEME` を返します。

また、`ConnCacheBean` クラスは、`oracle.jdbc.pool.OracleDataSource` クラスからプロパティおよび関連する取得メソッドと設定メソッドを継承します。これにより、プロパティ `databaseName`、`dataSourceName`、`description`、`networkProtocol`、`portNumber`、`serverName` および `driverType` の取得メソッドと設定メソッドが提供されます。これらのプロパティとその取得メソッドおよび設定メソッドの詳細は、『Oracle8i 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。

接続のオープンとクローズには、次のメソッドを使用します。

- `Connection getConnection()` — `ConnCacheBean` のプロパティ設定を使用して接続キャッシュから接続を取得します。
- `void close()` — すべての接続をクローズし、オープン・カーソルがあればそれもクローズします。

`ConnCacheBean` クラスでは、Oracle JDBC のバッチ更新と行のプリフェッチは直接サポートされませんが、`getConnection()` メソッドから取得する `Connection` オブジェクトの `setDefaultExecuteBatch(int)` および `setDefaultRowPrefetch(int)` メソッドをコールすると、これらの機能を有効化できます。また、`Connection` オブジェクトから作成する JDBC の文オブジェクトの `setExecuteBatch(int)` および `setRowPrefetch(int)` メソッドを使用することもできます（バッチ更新は、プリコンパイルされた SQL 文でのみサポートされます）。これらの機能の概要は、4-7 ページの「[バッチ更新](#)」および 4-7 ページの「[行のプリフェッチ](#)」を参照してください。

問合せ専用の DBBean

問合せのみを実行するには、`oracle.jsp.dbutil.DBBean` を使用します。

注意：

- `DBBean` には、独自の接続メカニズムがあります。`ConnBean` を使用しないでください。
 - 他のすべての DML 操作（`UPDATE`、`INSERT`、`DELETE` またはストアド・プロシージャ・コール）には、`CursorBean` を使用します。
 - JSP ページに使用する JavaBeans と同様に、設定メソッドを直接使用するかわりに、`jsp:setProperty` 文で `DBBean` のプロパティのいずれかを設定できます。
-

DBBean のプロパティは、次のとおりです。

- user (データベース・スキーマのユーザー ID)
- password (ユーザー・パスワード)
- URL (データベース接続文字列)

DBBean には、前述のプロパティ用に次の設定メソッドと取得メソッドがあります。

- void setUser(String)
- String getUser()
- void setPassword(String)
- String getPassword()
- void setURL(String)
- String getURL()

接続のオープンとクローズには、次のメソッドを使用します。

- void connect() — DBBean プロパティ設定を使用してデータベース接続を確立します。
- void close() — 接続をクローズし、オープン・カーソルがあればそれもクローズします。

問合せを実行するには、次のどちらかのメソッドを使用します。

- String getResultAsHTMLTable(String) — SELECT 文で文字列を入力します。
このメソッドは、結果セットを HTML 表として出力するために必要な HTML コマンドとともに文字列を戻します。表の列ヘッダーには、SQL 列名（または別名）が使用されます。
- String getResultAsXMLString(String) — SELECT 文で文字列を入力します。
このメソッドは、XML タグの SQL 名（または別名）を使用して、結果セットを XML 文字列として戻します。

DML とストアド・プロシージャ用の CursorBean

単純接続での SELECT、UPDATE、INSERT または DELETE 操作やストアド・プロシージャ・コールには、`oracle.jsp.dbutil.CursorBean` を使用します。接続には、以前に定義済みの ConnBean オブジェクトが使用されます。

SQL 操作は、ConnBean オブジェクトの `getCursorBean()` コールで指定するか、後述するように CursorBean オブジェクトの `create()`、`execute()` または `executeQuery()` メソッドのコールを介して指定できます。

CursorBean では、スクロール可能で更新可能なカーソル、バッチ更新、行のプリフェッチおよび問合せのタイムアウト制限がサポートされます。これらの Oracle JDBC 機能の詳細は、『Oracle8i 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。

注意：

- 接続キャッシュを使用するには、ConnCacheBean と標準の Connection オブジェクトの機能を使用します。CursorBean は使用しないでください。
 - JSP ページに使用する JavaBeans と同様に、設定メソッドを直接使用するかわりに、jsp:setProperty アクションで CursorBean プロパティのいずれかを設定できます。
-
-

CursorBean のプロパティは、次のとおりです。

- executeBatch (Oracle JDBC のバッチ更新用のバッチ・サイズ)
このプロパティを設定すると、Oracle JDBC のバッチ更新が使用可能になります。
- preFetch (Oracle JDBC の行のプリフェッチでプリフェッチされる文の数)
このプロパティを設定すると、Oracle JDBC の行のプリフェッチが使用可能になります。
- queryTimeout (タイムアウトを発行する前にドライバが文の実行を待機する秒数)
- resultSetType (結果セットのスクロール可能性)
 - TYPE_FORWARD_ONLY (デフォルト) — 前方にのみ (next () メソッドを使用して) スクロール可能で、位置指定はできない結果セット。
 - TYPE_SCROLL_INSENSITIVE — 前方または後方にスクロール可能で、位置指定できるが、基礎となるデータベース変更には影響を受けない結果セット。
 - TYPE_SCROLL_SENSITIVE — 前方または後方にスクロール可能で、位置指定でき、基礎となるデータベース変更に影響を受ける結果セット。

結果セットのスクロール可能性タイプの詳細は、『Oracle8i 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。

- resultSetConcurrency (結果セットの更新可能性)
 - CONCUR_READ_ONLY (デフォルト) — 読取り専用の (更新できない) 結果セット。
 - CONCUR_UPDATABLE — 更新可能な結果セット。

更新可能な結果セットの詳細は、『Oracle8i 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。

これらのプロパティを次のメソッドで設定し、Oracle JDBC の機能を必要に応じて有効化できます。

- `void setExecuteBatch(int)`
- `int getExecuteBatch()`
- `void setPrefetch(int)`
- `int getPrefetch()`
- `void setQueryTimeout(int)`
- `int getQueryTimeout()`
- `void setResultSetConcurrency(int)`
CursorBean.CONCUR_READ_ONLY または CursorBean.CONCUR_UPDATABLE を指定します。
- `int getResultSetConcurrency()`
CursorBean.CONCUR_READ_ONLY または CursorBean.CONCUR_UPDATABLE を戻します。
- `void setResultSetType(int)`
CursorBean.TYPE_FORWARD_ONLY、CursorBean.TYPE_SCROLL_INSENSITIVE または CursorBean.TYPE_SCROLL_SENSITIVE を指定します。
- `int getResultSetType()`
CursorBean.TYPE_FORWARD_ONLY、CursorBean.TYPE_SCROLL_INSENSITIVE または CursorBean.TYPE_SCROLL_SENSITIVE を戻します。

CursorBean インスタンスを `jsp:useBean` 文で定義した後に問合せを実行するには、CursorBean メソッドを使用して次のどちらかの方法でカーソルを作成できます。次のメソッドを使用すると、カーソルの作成と接続の提供を別々のステップで実行できます。

- `void create()`
- `void setConnBean(ConnBean)`

また、次のようにプロセスを 1 ステップにまとめることもできます。

- `void create(ConnBean)`

(ConnBean オブジェクトを 5-13 ページの「[データベース接続用の ConnBean](#)」の説明に従って設定してください。)

次のメソッドを使用し、問合せを指定して実行します。（裏側では JDBC の単純な Statement オブジェクトが使用されます。）

- `ResultSet executeQuery(String)`

SELECT 文で文字列を指定します。

また、結果セットを HTML 表または XML 文字列としてフォーマットする場合は、`executeQuery()` のかわりに次のどちらかのメソッドを使用します。

- `String getResultAsHTMLTable(String)`

文字列とともに、結果セットの HTML 表を作成する HTML 文を返します。SELECT 文で文字列を指定します。

- `String getResultAsXMLString(String)`

結果セット・データを XML 文字列で返します。SELECT 文で文字列を指定します。

CursorBean インスタンスを `jsp:useBean` アクションで定義した後に、UPDATE、INSERT または DELETE 文を実行するには、CursorBean メソッドを使用して次のどちらかの方法でカーソルを作成します。次のメソッドを使用すると、カーソルを作成して（文の型を integer、SQL 文を文字列として指定）接続を提供できます。

- `void create(int, String)`

- `void setConnBean(ConnBean)`

また、次のようにプロセスを 1 ステップにまとめることもできます。

- `void create(ConnBean, int, String)`

（ConnBean オブジェクトを 5-13 ページの「[データベース接続用の ConnBean](#)」で説明したように設定してください。）

`int` の入力値は、必要な JDBC 文の型を指定する定数、`CursorBean.PLAIN_STMT`（Statement オブジェクトの場合）、`CursorBean.PREP_STMT`（PreparedStatement オブジェクトの場合）または `CursorBean.CALL_STMT`（CallableStatement オブジェクトの場合）の 1 つを指定する値です。

`String` の入力値は、SQL 文を指定する値です。

次のメソッドを使用して INSERT、UPDATE または DELETE 文を実行します。（boolean の戻り値は無視してかまいません。）

- `boolean execute()`

また、バッチ更新の場合は、影響する行数を返す次のメソッドを使用します。（バッチ更新を有効化する方法については後述します。）

- `int executeUpdate()`

注意： `execute()` および `executeUpdate()` メソッドは、オプションで SQL 操作を指定する String を取ることができます。`create()` 対応するコールと、`ConnBean` 内の `getCursorBean()` コールは、オプションで SQL 操作を指定する String を取りません。文の作成時または実行時の操作を指定します。両方は指定できません。

また、`CursorBean` では、`registerOutParameter()` メソッド、`setXXX()` メソッドおよび `getXXX()` メソッドなど、Oracle JDBC の文および結果セット機能がサポートされます。

データベース・カーソルをクローズするには、次のメソッドを使用します。

- `void close()`

OracleJSP の SQL 用タグ・ライブラリ

リリース 8.1.7 では、OracleJSP は SQL 機能のためのカスタム・タグ・ライブラリを（JML のカスタム・タグ・ライブラリとは別個に）提供します。

提供されるタグは、次のとおりです。

- `dbOpen` — データベース接続をオープンします。
- `dbClose` — データベース接続をクローズします。
- `dbQuery` — 問合せを実行します。
- `dbCloseQuery` — 問合せ用のカーソルをクローズします。
- `dbNextRow` — 結果セットの各行を処理します。
- `dbExecute` — 任意の SQL 文（DML または DDL）を実行します。

ここでは、これらのタグについて説明します。例については、5-26 ページの「[SQL タグの例](#)」を参照してください。

SQL タグを使用する場合は、次の要件に注意してください。

- ファイル `ojsputil.jar` をインストールし、CLASSPATH に挿入します。このファイルは、OracleJSP のインストール時に提供されます。
- タグ・ライブラリ記述ファイル `sqltaglib.tld` がアプリケーションとともに配布され、次のように JSP ページの `taglib` ディレクティブで指定した場所に格納されていることを確認します。

```
<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
```

ライブラリ記述ファイルや taglib ディレクティブなど、JSP 1.1 のタグ・ライブラリの使用方法の概要は、7-2 ページの「[標準タグ・ライブラリのフレームワーク](#)」を参照してください。

SQL の dbOpen タグ

データベース接続をオープンするには、dbOpen タグを使用します。

```
<sql:dbOpen
  [ connId="connection-id" ]
  user="username"
  password="password"
  URL="databaseURL" >
...
</sql:dbOpen>
```

この接続で実行するネストしたコードは、dbOpen の開始タグと終了タグの間のタグ本体に挿入できます。(5-26 ページの「[SQL タグの例](#)」を参照してください。) オプションの connId パラメータを使用して接続識別子を設定する場合は、この接続を介して実行するコードで接続識別子を参照できます。dbOpen の開始タグと終了タグで囲む必要はありません。(接続識別子には任意の文字列を使用できます。)

パスワードを JSP ページにハードコーディングする必要はありません(セキュリティ上の問題になります)。パスワードはハードコーディングせずに、次のように request オブジェクトから他のパラメータとともに取得できます。

```
<sql:dbOpen connId="conn1" user=<%=request.getParameter("user")%>
  password=<%=request.getParameter("password")%> URL="url" />
```

(この例では、この接続を使用するコードのタグ本体は不要です。この接続を使用する文では、connId の conn1 値を介して参照できます。)

接続識別子を設定すると、dbClose タグで明示的にクローズするまで、接続はクローズされません。接続識別子を設定しない場合は、</sql:dbOpen> の終了タグが検出されると接続が自動的にクローズされます。

このタグでは、接続に ConnBean オブジェクトが使用されます。オプションで ConnBean のプロパティ stmtCacheSize、preFetch および batchSize を追加設定し、これらの Oracle JDBC 機能を有効化できます。詳細は、5-13 ページの「[データベース接続用の ConnBean](#)」を参照してください。

SQL の dbClose タグ

dbOpen タグで指定したオプションの connId パラメータに対応付けられた接続をクローズするには、dbClose タグを使用します。dbOpen タグ内で connId を使用しない場合は、dbOpen の終了タグに達すると接続が自動的にクローズされます。dbClose タグは不要です。

```
<sql:dbClose connId="connection-id" />
```

注意： OracleJSP 環境では、Oracle の JspScopeListener メカニズムを介して、セッション・ベースのイベント処理で接続を自動的にクローズさせることができます。5-30 ページの「[OracleJSP のイベント処理 — JspScopeListener](#)」を参照してください。

SQL の dbQuery タグ

問合せを実行し、結果を JDBC の結果セット、HTML 表または XML 文字列として出力するには、dbQuery タグを使用します。SELECT 文を、dbQuery の開始タグと終了タグの間のタグ本体に（1 つのみ）挿入します。

```
<sql:dbQuery
  [ queryId="query-id" ]
  [ connId="connection-id" ]
  [ output="HTML|XML|JDBC" ] >
  ...SELECT statement (one only) ...
</sql:dbQuery>
```

重要： リリース 8.1.7 (OracleJSP 1.1.0.0.0) では、SELECT 文の末尾にセミコロンを使用しないでください。末尾にセミコロンがあると構文エラーになります。

このタグのパラメータは、いずれも後述の用途に応じてオプションで使用できます。

dbNextRow タグを使用して結果セットを処理する場合は、queryId パラメータを使用して問合せ識別子を設定する必要があります。queryId には任意の文字列を使用できます。

また、queryId パラメータがあると、dbCloseQuery タグで明示的にクローズするまで、カーソルはクローズされません。問合せ識別子を設定しない場合は、</sql:dbQuery> の終了タグが検出されると、カーソルが自動的にクローズされます。

connId を指定しない場合は、dbQuery を dbOpen タグの本体内でネストする必要があります。dbOpen タグでオープンした接続が使用されます。

出力タイプの場合は、次の動作が発生します。

- HTML により、結果セットが HTML 表に挿入されます（デフォルト）。
- XML により、結果セットが XML 文字列に挿入されます。
- JDBC により、結果セットが dbNextRow タグを使用して処理できる JDBC の ResultSet オブジェクトに挿入され、行を介して反復されます。

このタグでは、カーソルに `CursorBean` オブジェクトが使用されます。`CursorBean` の機能の詳細は、5-18 ページの「[DML とストアド・プロシージャ用の `CursorBean`](#)」を参照してください。

SQL の `dbCloseQuery` タグ

`dbQuery` タグで指定したオプションの `queryId` パラメータに対応付けられたカーソルをクローズするには、`dbCloseQuery` タグを使用します。`dbQuery` タグ内で `queryId` を使用しない場合は、`dbQuery` の終了タグに達するとカーソルが自動的にクローズされます。`dbCloseQuery` タグは不要です。

```
<sql:dbCloseQuery queryId="query-id" />
```

注意： OracleJSP 環境では、Oracle の `JspScopeListener` メカニズムを介して、セッション・ベースのイベント処理でカーソルを自動的にクローズさせることができます。5-30 ページの「[OracleJSP のイベント処理 — `JspScopeListener`](#)」を参照してください。

SQL の `dbNextRow` タグ

`dbQuery` タグで取得し、指定した `queryId` に対応付けられている結果セットの各行を処理するには、`dbNextRow` タグを使用します。処理するコードを、`dbNextRow` の開始タグと終了タグの間のタグ本体に挿入します。本体が結果セットの行ごとに実行されます。

`dbNextRow` タグを使用する場合は、`dbQuery` タグで `output=JDBC` を指定し、`dbNextRow` タグで参照する `queryId` を指定する必要があります。

```
<sql:dbNextRow queryId="query-id" >
```

```
...Row processing...
```

```
</sql:dbNextRow >
```

`ResultSet` オブジェクトは、`dbQuery` タグの `Tag-Extra-Info` クラスのインスタンス内で作成されます（`Tag-Extra-Info` クラスの詳細は、7-10 ページの「[タグ・ライブラリ記述ファイル](#)」を参照してください）。

SQL の dbExecute タグ

DML または DDL 文を（1 つのみ）実行するには、dbExecute タグを使用します。文を、dbExecute の開始タグと終了タグの間のタグ本体に挿入します。

```
<sql:dbExecute
  [connId="connection-id"]
  [output="yes|no"] >
  ...DML or DDL statement (one only)...
</sql:dbExecute >
```

重要： リリース 8.1.7 では、DML または DDL 文の末尾にセミコロンを使用しないでください。末尾にセミコロンがあると構文エラーになります。

connId を指定しない場合は、dbExecute を dbOpen タグの本体内でネストし、dbOpen タグでオープンした接続を使用する必要があります。

output=yes に設定すると、DML 文の場合は、HTML 文字列「*number row[s] affected*」がブラウザに出力されます。これは、操作の影響を受けたデータベース行数を示します。DDL 文の場合は、文の実行ステータスが出力されます。デフォルト設定は no です。

このタグでは、カーソルに CursorBean オブジェクトが使用されます。CursorBean の機能の詳細は、5-18 ページの「[DML とストアド・プロシージャ用の CursorBean](#)」を参照してください。

SQL タグの例

次の例は、OracleJSP の SQL タグの使用方法を示しています。（自分で実行する場合は、URL、ユーザー名およびパスワードを適切に設定する必要があります。）

例 1: 接続 ID を使用した問合せ

```
<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
<HTML>
  <HEAD>
    <TITLE>A simple example with open, query, and close tags</TITLE>
  </HEAD>
  <BODY BGCOLOR="#FFFFFF">
    <HR>
    <sql:dbOpen URL="jdbc:oracle:thin:@dlsun991:1521:816"
      user="scott" password="tiger" connId="con1">
    </sql:dbOpen>
    <sql:dbQuery connId="con1">
      select * from EMP
    </sql:dbQuery>
```

```

        <sql:dbClose connId="con1" />
    <HR>
</BODY>
</HTML>

```

例 2: dbOpen タグ内でネストした問合せ

```

<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
<HTML>
    <HEAD>
        <TITLE>Nested Tag with Query inside Open </TITLE>
    </HEAD>
    <BODY BGCOLOR="#FFFFFF">
        <HR>
        <sql:dbOpen URL="jdbc:oracle:thin:@dlsun991:1521:816"
            user="scott" password="tiger">
            <sql:dbQuery>
                select * from EMP
            </sql:dbQuery>
        </sql:dbOpen>
        <HR>
    </BODY>
</HTML>

```

例 3: XML 出力を使用した問合せ

```

<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
<HTML>
    <HEAD>
        <TITLE>A simple tagLib with XML output</TITLE>
    </HEAD>
    <BODY BGCOLOR="#FFFFFF">
        <HR>
        <sql:dbOpen URL="jdbc:oracle:thin:@dlsun991:1521:816"
            user="scott" password="tiger">
            <sql:dbQuery output="xml">
                select * from EMP
            </sql:dbQuery>
        </sql:dbOpen>
        <HR>
    </BODY>
</HTML>

```

例 4: 結果セットの反復

```

<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
<HTML>
  <HEAD>
    <TITLE>Result Set Iteration Sample </TITLE>
  </HEAD>
  <BODY BGCOLOR="#FFFFFF">
    <HR>
    <sql:dbOpen connId="con1" URL="jdbc:oracle:thin:@dlsun991:1521:816"
      user="scott" password="tiger">
    </sql:dbOpen>
    <sql:dbQuery connId="con1" output="jdbc" queryId="myquery">
      select * from EMP
    </sql:dbQuery>
    <sql:dbNextRow queryId="myquery">
      <%= myquery.getString(1) %>
    </sql:dbNextRow>
    <sql:dbCloseQuery queryId="myquery" />
    <sql:dbClose connId="con1" />
    <HR>
  </BODY>
</HTML>

```

例 5: DDL 文と DML 文 この例では、ユーザーが、実行する DML または DDL 文の種類を指定できるように、HTML フォームを使用しています。

```

<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
<HTML>
<HEAD><TITLE>DML Sample</TITLE></HEAD>
<FORM METHOD=get>
<INPUT TYPE="submit" name="drop" VALUE="drop table test_table"><br>
<INPUT TYPE="submit" name="create"
  VALUE="create table test_table (col1 NUMBER)"><br>
<INPUT TYPE="submit" name="insert"
  VALUE="insert into test_table values (1234)"><br>
<INPUT TYPE="submit" name="select" VALUE="select * from test_table"><br>
</FORM>
<BODY BGCOLOR="#FFFFFF">
Result:

  <HR>
  <sql:dbOpen URL="jdbc:oracle:thin:@dlsun991:1521:816"
    user="scott" password="tiger">
    <% if (request.getParameter("drop")!=null) { %>
    <sql:dbExecute output="yes">
      drop table test_table
    </sql:dbExecute>

```



```

<% } %>
<% if (request.getParameter("create")!=null) { %>
<sql:dbExecute output="yes">
    create table test_table (col1 NUMBER)
</sql:dbExecute>
<% } %>
<% if (request.getParameter("insert")!=null) { %>
<sql:dbExecute output="yes">
    insert into test_table values (1234)
</sql:dbExecute>
<% } %>
<% if (request.getParameter("select")!=null) { %>
<sql:dbQuery>
    select * from test_table
</sql:dbQuery>
<% } %>
</sql:dbOpen>
<HR>

</BODY>
</HTML>

```

Oracle 固有のプログラミング拡張機能

この項で説明する OracleJSP の拡張機能は、他の JSP 環境には移植できません。たとえば、次の拡張機能があります。

- Oracle JspScopeListener メカニズムを介したイベント処理
- SQL 文を Java コードに直接埋め込むための標準構文である SQLJ のサポート
- JDBC のパフォーマンス強化機能の使用

注意：

- サーブレット 2.0 環境では、Web アプリケーション・フレームワークをサポートするために、OracleJSP は `globals.jsa` と呼ばれるメカニズム、移植できない拡張機能を提供します。このメカニズムについては、5-34 ページの「[サーブレット 2.0 に対する OracleJSP のアプリケーションおよびセッションのサポート](#)」を参照してください。
 - また、OracleJSP は拡張（移植できない）NLS のサポートも提供します。8-4 ページの「[マルチバイト・パラメータのコード化に対する OracleJSP の拡張サポート](#)」を参照してください。
-
-

OracleJSP のイベント処理 — JspScopeListener

標準のサーブレットおよび JSP テクノロジーでサポートされるのは、セッション・ベースのイベントのみです。OracleJSP では、このサポートが `oracle.jsp.event` パッケージ内の `JspScopeListener` インタフェースと `JspScopeEvent` クラスを介して拡張されます。OracleJSP のメカニズムでは、JSP アプリケーションで使用される Java オブジェクトすべてについて、イベント処理用に次の 4 つの標準 JSP 有効範囲がサポートされます。

- `page`
- `request`
- `session`
- `application`

アプリケーションに使用する Java オブジェクト用に、`JspScopeListener` インタフェースを適切なクラスに実装し、そのクラスのオブジェクトを `jsp:useBean` などのタグで JSP の有効範囲に連結します。

有効範囲の終わりに達すると、`JspScopeListener` を実装し、その有効範囲に連結されていたオブジェクトにその旨が通知されます。通知は、OracleJSP コンテナにより、`JspScopeListener` インタフェースに指定した `outOfScope()` メソッドを介して、該当するオブジェクトに `JspScopeEvent` インスタンスが送られて実行されます。

`JspScopeEvent` オブジェクトのプロパティは、次のとおりです。

- 終了する有効範囲（定数 `PAGE_SCOPE`、`REQUEST_SCOPE`、`SESSION_SCOPE` または `APPLICATION_SCOPE` のいずれか 1 つ）
- この有効範囲のオブジェクトのリポジトリであるコンテナ・オブジェクト（暗黙的オブジェクト `page`、`request`、`session` または `application` のいずれか 1 つ）
- 通知に関連するオブジェクトの名前（`JspScopeListener` を実装するクラスのインスタンスの名前）
- JSP の暗黙的な `application` オブジェクト

OracleJSP のイベント・リスナー・メカニズムは、`page` または `request` 有効範囲のオブジェクト・リソースを、エラー条件に関係なく常に解放する必要のある開発者にとって大きなメリットをもたらします。このような開発者は、ページ実装を Java の `try/catch/finally` ブロックで囲む必要がなくなります。

詳細なサンプルについては、9-29 ページの「[JspScopeListener を使用するページ — scope.jsp](#)」を参照してください。

OracleJSP による Oracle SQLJ のサポート

SQLJ は、静的な SQL 指示を Java コードに直接埋め込むための標準構文であり、データベース・アクセスのプログラミングを大幅に簡素化します。OracleJSP と OracleJSP トランスレータでは Oracle SQLJ がサポートされており、JSP のスクリプトレットに SQLJ 構文を使用できます。SQLJ 文は、`#sql` トークンで示されます。

SQLJ JSP コードの例

次の例は、サンプル SQLJ JSP ページを示しています。(page ディレクティブでは、一般に SQLJ に必要なクラスがインポートされます。)

```
<%@ page language="sqlj"
    import="sqlj.runtime.ref.DefaultContext,oracle.sqlj.runtime.Oracle" %>
<HTML>
<HEAD> <TITLE> The SQLJQuery JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">
<% String empno = request.getParameter("empno");
if (empno != null) { %>
<H3> Employee # <%=empno %> Details: </H3>
<%= runQuery(empno) %>
<HR><BR>
<% } %>
<B>Enter an employee number:</B>
<FORM METHOD="get">
<INPUT TYPE="text" NAME="empno" SIZE=10>
<INPUT TYPE="submit" VALUE="Ask Oracle");
</FORM>
</BODY>
</HTML>
<%!

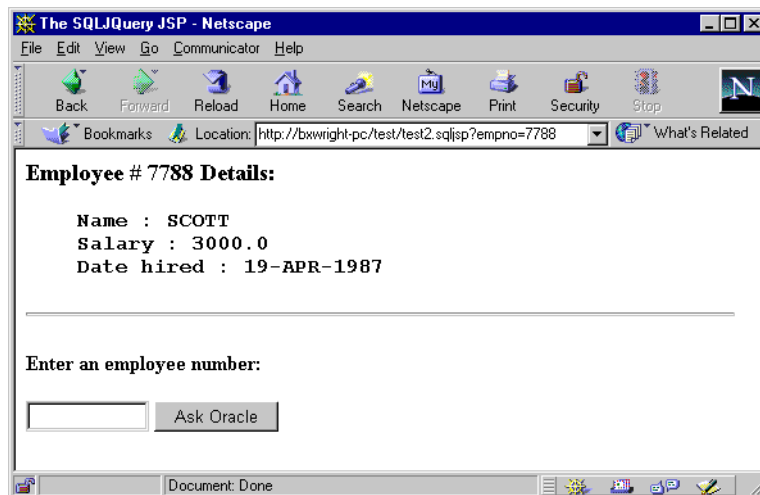
private String runQuery(String empno) throws java.sql.SQLException {
    DefaultContext dctx = null;
    String ename = null; double sal = 0.0; String hireDate = null;
    StringBuffer sb = new StringBuffer();
    try {
        dctx = Oracle.getConnection("jdbc:oracle:oci8:@", "scott", "tiger");
        #sql [dctx] {
            select ename, sal, TO_CHAR(hiredate,'DD-MON-YYYY')
            INTO :ename, :sal, :hireDate
            FROM scott.emp WHERE UPPER(empno) = UPPER(:empno)
        };
        sb.append("<BLOCKQUOTE><BIG><B><PRE>\n");
        sb.append("Name : " + ename + "\n");
        sb.append("Salary : " + sal + "\n");
        sb.append("Date hired : " + hireDate);
```

```
sb.append("</PRE></B></BIG></BLOCKQUOTE>");
} catch (java.sql.SQLException e) {
    sb.append("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
} finally {
    if (dctx!= null) dctx.close();
}
return sb.toString();
}

%>
```

この例では、JDBC OCI ドライバを使用しています。(JDBC OCI ドライバを使用するには、Oracle クライアントのインストールが必要です。) 接続の取得に使用される Oracle クラスは、Oracle SQLJ で提供されます。

従業員番号 7788 を入力すると、結果は次のように出力されます。



注意：

- JSP ページが同じ JVM 内で複数回起動される場合は、デフォルトの接続コンテキストのかわりに、常に明示的な接続コンテキストを使用することをお勧めします。この例では、`dctx` を使用しています。
(`dctx` は、ローカルのメソッド変数であることに注意してください。)
 - OracleJSP には、Oracle SQLJ リリース 8.1.6.1 以上が必要です。
 - 将来の OracleJSP では、JSP 変換中に Oracle SQLJ トランスレータをトリガーする `page` ディレクティブ内で、`language="sqlj"` がサポートされるようになります。上位互換性を保つために、プログラミング作業では、このディレクティブの使用をただちに開始することをお勧めします。
-

JSP ページでの SQLJ の使用例の詳細は、9-34 ページの「[SQLJ の問合せ — SQLJSelectInto.sqljsp および SQLJIterator.sqljsp](#)」を参照してください。

Oracle SQLJ のプログラミング機能と構文の概要は、『Oracle8i SQLJ 開発者ガイドおよびリファレンス』を参照してください。

SQLJ トランスレータの実行タイミング

JSP のソース・ファイルのファイル拡張子 `.sqljsp` を使用すると、OracleJSP トランスレータをトリガーして Oracle SQLJ トランスレータを起動できます。

これにより、OracleJSP トランスレータで `.java` ファイルのかわりに `.sqlj` ファイルが生成されます。次に Oracle SQLJ トランスレータが起動し、`.sqlj` ファイルが `.java` ファイルに変換されます。

SQLJ を使用すると、出力ファイルが追加生成されます。6-6 ページの「[生成されるファイルと位置 \(オンデマンド変換\)](#)」を参照してください。

重要：

- Oracle SQLJ を使用するには、(環境に応じて) 適切な SQLJ ZIP ファイルをインストールし、CLASSPATH に追加する必要があります。
A-3 ページの「[OracleJSP の必須ファイルとオプション・ファイル](#)」を参照してください。
 - 生成されるクラス名と `.java` ファイル名が同じになってしまうため、同一アプリケーション内では、`.jsp` ファイルと `.sqljsp` ファイルに同じベース・ファイル名を使用しないでください。
-

Oracle SQLJ オプションの設定

SQLJ の JSP ページを実行または事前変換する場合は、必要な Oracle SQLJ オプション設定を指定できます。これは、オンデマンド変換の使用例と事前変換の使用例の両方に、次のように当てはまります。

- オンデマンド変換の場合は、OracleJSP の `sqljcmd` 構成パラメータを使用します。このパラメータを使用すると、特定の SQLJ トランスレータの実行ファイルを指定できるのみでなく、SQLJ のコマンドライン・オプションも設定できます。（`sqljcmd` パラメータは、リリース 1.1.0.0.0 より前の OracleJSP では使用できません。）

詳細は、A-13 ページの「[OracleJSP の構成パラメータ（非 OSE）](#)」にある `sqljcmd` の説明を参照してください。この構成パラメータの設定方法は、A-23 ページの「[OracleJSP の構成パラメータ設定](#)」を参照してください。

- `ojspc` 事前変換ツールで事前変換する場合は、`ojspc -s` オプションを使用します。このオプションにより、SQLJ のコマンドライン・オプションを設定できます。

詳細は、6-25 ページの「[ojspc のコマンドライン構文](#)」および 6-26 ページの「[ojspc のオプションの説明](#)」を参照してください。

Oracle SQLJ オプションの概要は、『Oracle8i SQLJ 開発者ガイドおよびリファレンス』を参照してください。

サーブレット 2.0 に対する OracleJSP のアプリケーションおよびセッションのサポート

OracleJSP では、サーブレット 2.0 環境に JSP 仕様を実装するメカニズムとして、ファイル `globals.jsa` が定義されています。Web アプリケーション・コンテキストとサーブレット・コンテキストは、サーブレット 2.0 仕様では十分に定義されていませんでした。

この項では、`globals.jsa` のメカニズムについて次のトピックで説明します。

- [globals.jsa の機能の概要](#)
- [globals.jsa の構文とセマンティックの概要](#)
- [globals.jsa のイベント・ハンドラ](#)
- [グローバル宣言およびディレクティブ](#)

サンプル・アプリケーションについては、9-38 ページの「[サーブレット 2.0 環境に globals.jsa を使用するサンプル](#)」を参照してください。

重要： `globals.jsa` のファイル名には、小文字のみを使用してください。大 / 小文字区別のない環境では大文字も使用できますが、ページを大 / 小文字区別のある環境に移植した場合に生じるエラーを診断するのが難しくなります。

globals.jsa の機能の概要

どの単一 Java 仮想マシン内でも、アプリケーションごと（つまりサーブレット・コンテキストごと）に `globals.jsa` ファイルを使用できます。このファイルにより、次の分野で Web アプリケーションの概念がサポートされます。

- アプリケーションの配布 — アプリケーション・ルートを定義するアプリケーション・ロケーション・マーカースとしてのロール
- 個別のアプリケーションとセッション — アプリケーションごとに個別のサーブレット・コンテキストとセッション・オブジェクトを提供する際の OracleJSP による使用
- アプリケーションのライフ・サイクルの管理 — セッションとアプリケーションの開始イベントと終了イベント

また、`globals.jsa` ファイルは、アプリケーションのすべての JSP ページ間で、グローバルな Java 宣言と JSP ディレクティブのための媒介を提供します。

globals.jsa を介したアプリケーションの配布

サーブレットを取り込まない OracleJSP アプリケーションを配布するには、そのディレクトリ構造を Web サーバーにコピーし、ファイル `globals.jsa` を作成してアプリケーション・ルート・ディレクトリに配置します。

`globals.jsa` のファイル・サイズは 0（ゼロ）でもかまいません。このファイルの位置は OracleJSP コンテナにより識別され、ディレクトリ内での存在により、そのディレクトリが（URL の仮想パスからマップされ）アプリケーションのルート・ディレクトリとして定義されます。

また、OracleJSP では、JSP アプリケーション・リソースのデフォルト位置も定義されます。たとえば、アプリケーション相対の `WEB_INF/classes` および `WEB_INF/lib` ディレクトリにあるアプリケーションの Bean とクラスは、OracleJSP のクラス・ローダーにより自動的にロードされ、特定の構成を必要としません。

注意： サーブレットを取込むアプリケーションの場合、特にサーブレット 2.2 仕様より前のサーブレット環境では、サーブレット配布の場合と同様に手動構成が必要です。サーブレット 2.2 環境にあるサーブレットの場合は、必要な構成を標準の `web.xml` デプロイメント・ディスクリプタに挿入できます。

globals.jsa を介した個別アプリケーションおよびセッション

サーブレット 2.0 仕様では Web アプリケーションの概念が明確に定義されておらず、それ以後のサーブレット仕様のようにサーブレット・コンテキストとアプリケーション間の関連が定義されていません。Apache/JServ などのサーブレット 2.0 環境では、サーブレット・コンテキスト・オブジェクトは JVM ごとに 1 つずつです。また、セッション・オブジェクトも 1 つのみです。

ただし、globals.jsa ファイルは、特にサーブレット 2.0 環境で使用する場合は、Web サーバーにおける複数アプリケーションおよび複数セッションのサポートを提供します。

他の方法ではアプリケーションごとに個別のサーブレット・コンテキスト・オブジェクトを使用できない場合は、アプリケーションの globals.jsa ファイルの存在により、OracleJSP コンテナはアプリケーションに個別の ServletContext オブジェクトを提供できます。

また、他の方法ではセッション・オブジェクトが（単一のサーブレット・コンテキストまたは複数のサーブレット・コンテキスト間で）1 つしかない場合は、globals.jsa ファイルの存在により、OracleJSP コンテナはアプリケーションにプロキシ HttpSession オブジェクトを提供できます。（これにより、セッション変数名が他のアプリケーションと衝突することはなくなりますが、他のアプリケーションによるアプリケーション・データの検査または変更は防止できません。HttpSession オブジェクトは、一部の機能について基礎となるサーブレットのセッション環境に依存する必要があるためです。）

globals.jsa を介したアプリケーションとセッションのライフ・サイクルの管理

重要な状態推移が発生した場合、アプリケーションは通知を受け取る必要があります。たとえば、アプリケーションは HTTP セッションの開始時にリソースを取得して、セッション終了時にリソースを解放したり、アプリケーション自体の起動時または終了時に永続データをリストアまたは保存することがよくあります。

ただし、標準のサーブレットおよび JSP テクノロジでサポートされるのは、セッション・ベースのイベントのみです。

globals.jsa ファイルを使用するアプリケーションの場合は、OracleJSP により、この機能が次の 4 つのイベントで拡張されます。

- session_OnStart
- session_OnEnd
- application_OnStart
- application_OnEnd

サーバーによる応答を必要とする前述のイベントについて、globals.jsa ファイル内でイベント・ハンドラを記述できます。

session_OnStart イベントと session_OnEnd イベントは、それぞれ HTTP セッションの開始時と終了時にトリガーされます。

application_OnStart イベントは、どのアプリケーションの場合にも、単一の JVM 内でそのアプリケーションに対する最初の要求によりトリガーされます。application_OnEnd イベントは、OracleJSP コンテナによりアプリケーションがアンロードされる時にトリガーされます。

詳細は、5-39 ページの「[globals.jsa のイベント・ハンドラ](#)」を参照してください。

globals.jsa の構文とセマンティックの概要

この項では、globals.jsa ファイルの一般的な構文とセマンティックの概要を説明します。

globals.jsa ファイル内の各イベント・ブロック、つまり session_OnStart ブロック、session_OnEnd ブロック、application_OnStart ブロックまたは application_OnEnd ブロックには、イベント開始タグ、イベント終了タグ、およびイベント・ハンドラ・コードを含む本体（開始タグと終了タグで囲まれた部分全体）があります。

次の例は、このパターンを示しています。

```
<event:session_OnStart>
  <% This scriptlet contains the implementation of the event handler %>
</event:session_OnStart>
```

イベント・ブロックの本体には、標準タグやカスタム・タグ・ライブラリ内で定義されているタグなど、有効な任意の JSP タグを含めることができます。

ただし、イベント・ブロック内のどの JSP タグの有効範囲も、そのブロックのみに限定されます。たとえば、あるイベント・ブロック内の jsp:useBean タグで宣言される Bean は、その Bean を使用する他のイベント・ブロック内で再宣言する必要があります。ただし、この制限事項は、globals.jsa のグローバル宣言メカニズムを介して回避できます。5-43 ページの「[グローバル宣言およびディレクティブ](#)」を参照してください。

4 つのイベント・ハンドラの個々の詳細は、5-39 ページの「[globals.jsa のイベント・ハンドラ](#)」を参照してください。

重要： 標準の JSP ページに使用する静的テキストを格納できるのは、session_OnStart ブロックのみです。session_OnEnd、application_OnStart および application_OnEnd のイベント・ブロックに挿入できるのは、Java のスクリプトレットのみです。

JSP の暗黙的オブジェクトは、globals.jsa のイベント・ブロック内で次のように使用できます。

- application_OnStart ブロックには、application オブジェクトへのアクセス権があります。
- application_OnEnd ブロックには、application オブジェクトへのアクセス権があります。
- session_OnStart ブロックには、application、session、request、response、page および out オブジェクトへのアクセス権があります。
- session_OnEnd ブロックには、application および session オブジェクトへのアクセス権があります。

完全な globals.jsa ファイルの例 この例は、4 つのイベント・ハンドラすべてを使用する完全な globals.jsa ファイルを示しています。

```
<event:application_OnStart>

<!-- Initializes counts to zero -->
<jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
<jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
<jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

</event:application_OnStart>

<event:application_OnEnd>

<!-- Acquire beans -->
<jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
<jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
<% application.log("The number of page hits were: " + pageCount.getValue() ); %>
<% application.log("The number of client sessions were: " + sessionCount.getValue() ); %>

</event:application_OnEnd>

<event:session_OnStart>

<!-- Acquire beans -->
<jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
<jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />
<%
    sessionCount.setValue(sessionCount.getValue() + 1);
    activeSessions.setValue(activeSessions.getValue() + 1);
%>
<br>
Starting session #: <%=sessionCount.getValue() %> <br>
There are currently <b> <%= activeSessions.getValue() %> </b> active sessions <p>
```

```
</event:session_OnStart>

<event:session_OnEnd>

    <%-- Acquire beans --%>
    <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />
    <%
        activeSessions.setValue(activeSessions.getValue() - 1);
    %>

</event:session_OnEnd>
```

globals.jsa のイベント・ハンドラ

この項では、globals.jsa の 4 つのイベント・ハンドラの詳細を個別に説明します。

application_OnStart

application_OnStart ブロックの一般的な構文は、次のとおりです。

```
<event:application_OnStart>
    <% This scriptlet contains the implementation of the event handler %>
</event:application_OnStart>
```

application_OnStart イベント・ハンドラの本体は、OracleJSP がアプリケーション内に最初の JSP ページをロードするときに実行されます。通常、これが発生するのは、任意のクライアントから、アプリケーション内のいずれかのページに対して最初の HTTP 要求が発行されるときです。アプリケーションは、このイベントを使用して、データベース接続プールや、永続リポジトリからアプリケーション・オブジェクトに読み込まれたデータなど、アプリケーション単位のリソースを初期化します。

イベント・ハンドラには、JSP タグ（カスタム・タグなど）と空白のみを含める必要があります。静的テキストは使用できません。

このイベント・ハンドラで発生した、イベント・ハンドラ・コードで処理されないエラーは、OracleJSP コンテナにより自動的にトラップされ、アプリケーションのサーブレット・コンテキストを使用してログに記録されます。その後、イベント処理はエラーが発生しなかったかのように進行します。

例 : application_OnStart 次の application_OnStart の例は、9-38 ページの「[アプリケーション・イベント用の globals.jsa の例 — lotto.jsp](#)」から抜粋したものです。この例では、特定ユーザーに関して生成された抽選番号が 1 日だけキャッシュされます。ユーザーが抽選をもう一度要求すると、同じ番号セットを取得します。キャッシュは 1 日に一度リサイクルされ、各ユーザーには新しい抽選セットが与えられます。この意図のとおり機能させるには、番号くじアプリケーションはシャットダウン時にキャッシュを永続的なものにして、アプリ

セッションが再びアクティブになった時点でキャッシュをリフレッシュする必要があります。

`application_OnStart` イベント・ハンドラは、`lotto.che` ファイルからキャッシュを読み込みます。

```
<event:application_OnStart>

<%
    Calendar today = Calendar.getInstance();
    application.setAttribute("today", today);
    try {
        FileInputStream fis = new FileInputStream
            (application.getRealPath("/") + File.separator + "lotto.che");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Calendar cacheDay = (Calendar) ois.readObject();
        if (cacheDay.get(Calendar.DAY_OF_YEAR) == today.get(Calendar.DAY_OF_YEAR)) {
            cachedNumbers = (Hashtable) ois.readObject();
            application.setAttribute("cachedNumbers", cachedNumbers);
        }
        ois.close();
    } catch (Exception theE) {
        // catch all -- can't use persistent data
    }
%>

</event:application_OnStart>
```

application_OnEnd

`application_OnEnd` ブロックの一般的な構文は、次のとおりです。

```
<event:application_OnEnd>
    <% This scriptlet contains the implementation of the event handler %>
</event:application_OnEnd>
```

`application_OnEnd` イベント・ハンドラの本体は、OracleJSP が JSP アプリケーションをアンロードするときに実行されます。アンロードは、オンデマンドの動的再変換後に、以前にロードしたページを再ロードするたびに発生します (OracleJSP の `unsafe_reload` 構成パラメータが有効化されていない場合)。または、それ自体がサーブレットである OracleJSP コンテナの `destroy()` メソッドが、基礎となるサーブレット・コンテナによりコールされ、そのコンテナが終了するときに発生します。アプリケーションは、`application_OnEnd` イベントを使用してアプリケーション・レベルのリソースをクリーン・アップするか、アプリケーションの状態を永続ストアに書き込みます。

イベント・ハンドラには、JSP タグ (カスタム・タグなど) と空白のみを含める必要があります。静的テキストは使用できません。

このイベント・ハンドラに発生した、イベント・ハンドラ・コードで処理されないエラーは、OracleJSP コンテナにより自動的にトラップされ、アプリケーションのサーブレット・コンテキストを使用してログに記録されます。その後、イベント処理はエラーが発生しなかったかのように進行します。

例 : application_OnEnd 次の application_OnEnd の例は、9-38 ページの「[アプリケーション・イベント用の globals.jsa の例 — lotto.jsp](#)」から抜粋したものです。このイベント・ハンドラでは、キャッシュがアプリケーションの終了前にファイル lotto.che に書き込まれます。

```
<event:application_OnEnd>

<%
    Calendar now = Calendar.getInstance();
    Calendar today = (Calendar) application.getAttribute("today");
    if (cachedNumbers.isEmpty() ||
        now.get(Calendar.DAY_OF_YEAR) > today.get(Calendar.DAY_OF_YEAR)) {
        File f = new File(application.getRealPath("/") + File.separator + "lotto.che");
        if (f.exists()) f.delete();
        return;
    }

    try {
        FileOutputStream fos = new FileOutputStream
            (application.getRealPath("/") + File.separator + "lotto.che");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(today);
        oos.writeObject(cachedNumbers);
        oos.close();
    } catch (Exception theE) {
        // catch all -- can't use persistent data
    }
%>

</event:application_OnEnd>
```

session_OnStart

session_OnStart ブロックの一般的な構文は、次のとおりです。

```
<event:session_OnStart>
    <% This scriptlet contains the implementation of the event handler %>
    Optional static text...
</event:session_OnStart>
```

`session_OnStart` イベント・ハンドラの本体は、OracleJSP が JSP ページ要求への応答時に新規セッションを作成するときに実行されます。これは、アプリケーション内でセッションが使用可能な JSP ページに関する最初の要求を受け取るたびに、クライアントごとに発生します。

アプリケーションでは、このイベントを次の用途に使用できます。

- 特定のクライアントに連結されたリソースの初期化
- アプリケーションでのクライアントの起動位置の制御

`session_OnStart` では暗黙的 `out` オブジェクトが使用できるため、これは JSP タグに加えて静的テキストを使用できる `globals.jsa` の唯一のイベント・ハンドラです。

`session_OnStart` イベント・ハンドラは、JSP ページのコードが実行される前にコールされます。そのため、`session_OnStart` からの出力はページからの出力より優先されます。

`session_OnStart` イベント・ハンドラとイベントをトリガーした JSP ページは、同じ `out` ストリームを共有します。このストリームのバッファ・サイズは、JSP ページのバッファ・サイズにより制御されます。`session_OnStart` イベント・ハンドラでは、ストリームがブラウザに自動的にフラッシュされません。ストリームは、一般的な JSP ルールに従ってフラッシュされます。ヘッダーは、`session_OnStart` イベントをトリガーする JSP ページに引き続き書き込むことができます。

このイベント・ハンドラで発生した、イベント・ハンドラ・コードで処理されないエラーは、OracleJSP コンテナにより自動的にトラップされ、アプリケーションのサーブレット・コンテキストを使用してログに記録されます。その後、イベント処理はエラーが発生しなかったかのように進行します。

例 : `session_OnStart` 次の例では、各新規セッションがアプリケーションの初期ページ (`index.jsp`) で開始されることを確認しています。

```
<event:session_OnStart>

    <% if (!page.equals("index.jsp")) { %>
        <jsp:forward page="index.jsp" />
    <% } %>

</event:session_OnStart>
```

`session_OnEnd`

`session_OnEnd` ブロックの一般的な構文は、次のとおりです。

```
<event:session_OnEnd>
    <% This scriptlet contains the implementation of the event handler %>
</event:session_OnEnd>
```

`session_OnEnd` イベント・ハンドラの本体は、OracleJSP が既存セッションを無効にするときに実行されます。これは、次のどちらかの状況で発生します。

- アプリケーションで `session.invalidate()` メソッドをコールしてセッションを無効化する場合
- セッションがサーバー上で時間切れ（タイムアウト）になる場合

アプリケーションは、このイベントを使用してクライアントのリソースを解放します。

イベント・ハンドラには、JSP タグ（タグ・ライブラリのタグなど）と空白のみを含める必要があります。静的テキストは使用できません。

このイベント・ハンドラに発生したが、イベント・ハンドラ・コードで処理されないエラーは、OracleJSP コンテナにより自動的にトラップされ、アプリケーションのサーブレット・コンテキストを使用してログに記録されます。その後、イベント処理はエラーが発生しなかったかのように進行します。

例 : session_OnEnd 次の例では、セッション終了時に「アクティブなセッション」数を減らしています。

```
<event:session_OnEnd>

<!-- Acquire beans --%>
<jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

<%
    activeSessions.setValue(activeSessions.getValue() - 1);
%>

</event:session_OnEnd>
```

グローバル宣言およびディレクティブ

`globals.jsa` ファイルは、イベント・ハンドラを格納する以外に、JSP アプリケーションのディレクティブとオブジェクトをグローバルに宣言するために使用できます。`scope` パラメータを持つ JSP ディレクティブ（`jsp:useBean` など）、JSP 宣言、JSP コメントおよび JSP タグを挿入できます。

この項で説明するトピックは、次のとおりです。

- [グローバルな JSP ディレクティブ](#)
- [globals.jsa の宣言](#)
- [グローバルな JavaBeans](#)
- [globals.jsa の構造](#)
- [グローバル宣言およびディレクティブの例](#)

グローバルな JSP ディレクティブ

globals.jsa ファイル内で使用されるディレクティブは、次の 2 つの機能を果たします。

- globals.jsa ファイル自体の処理に必要な情報を宣言します。
- 後続ページのデフォルト値を設定します。

globals.jsa ファイル内のディレクティブはアプリケーション内のすべての JSP ページに対する暗黙的ディレクティブとなりますが、特定ページの globals.jsa ディレクティブを上書きできます。

globals.jsa ディレクティブは、JSP ページ内で属性ごとに上書きされます。たとえば、globals.jsa ファイルに次のディレクティブが含まれているとします。

```
<%@ page import="java.util.*" bufferSize="10kb" %>
```

さらに、JSP ページに次のディレクティブが含まれているとします。

```
<%@page bufferSize="20kb" %>
```

これは、次のディレクティブを含むページと等価です。

```
<%@ page import="java.util.*" bufferSize="20kb" %>
```

globals.jsa の宣言

globals.jsa ファイル内のいずれかのイベント・ハンドラ間で共有するメソッドまたはデータ・メンバーを宣言する場合は、globals.jsa ファイル内で JSP の `<%!... %>` 宣言を使用します。

アプリケーション内の JSP ページにはこれらの宣言へのアクセス権がないため、このメカニズムはアプリケーション・ライブラリの実装には使用できないので注意してください。宣言のサポートは、共通のファンクションをイベント・ハンドラ間で共有できるように、globals.jsa ファイルに用意されています。

グローバルな JavaBeans

globals.jsa ファイル内で宣言される最も一般的な要素は、グローバル・オブジェクトです。globals.jsa ファイル内で宣言されたオブジェクトは、globals.jsa イベント・ハンドラとアプリケーション内のすべての JSP ページの暗黙的オブジェクト環境の一部となります。

globals.jsa ファイル内で (jsp:useBean 文などにより) 宣言されたオブジェクトは、アプリケーションの個々の JSP ページ内で再宣言する必要はありません。

グローバル・オブジェクトの宣言には、jsp:useBean または jml:useVariable など、scope パラメータを持つ任意の JSP タグまたは拡張機能を使用できます。グローバルに宣言されたオブジェクトの場合、有効範囲は (page または request ではなく) session または application にする必要があります。

ネストしたタグがサポートされます。したがって、`jsp:useBean` 宣言内で `jsp:setProperty` コマンドをネストできます。(`jsp:setProperty` を `jsp:useBean` 宣言の外側で使用すると、変換エラーが発生します。)

globals.jsa の構造

`globals.jsa` イベント・ハンドラ内でグローバル・オブジェクトを使用する場合は、その宣言の位置が重要になります。特定のイベント・ハンドラの前に宣言されているオブジェクトのみが、そのイベント・ハンドラに暗黙的オブジェクトとして追加されます。このため、開発者には、`globals.jsa` ファイルを次の順序で構築することをお勧めします。

1. グローバル・ディレクティブ
2. グローバル・オブジェクト
3. イベント・ハンドラ
4. `globals.jsa` の宣言

グローバル宣言およびディレクティブの例

次のサンプル `globals.jsa` ファイルで実行される内容は、次のとおりです。

- `globals.jsa` ファイルと後続の全ページの JML タグ・ライブラリ（この場合は、コンパイル時実装）を定義します。

taglib ディレクティブを `globals.jsa` ファイルに挿入することで、アプリケーションの JSP ページにこのディレクティブを個別に挿入する必要がなくなります。
- すべてのページで使用する 3 つのアプリケーション変数を（`jsp:useBean` 文で）宣言します。

グローバル宣言に関する `globals.jsa` の他の使用例については、9-43 ページの「[グローバル宣言用の globals.jsa の例 — index2.jsp](#)」を参照してください。

```
<%-- Directives at the top --%>

<%@ taglib uri="oracle.jsp.parse.OpenJspRegisterLib" prefix="jml" %>

<%-- Declare global objects here --%>

<%-- Initializes counts to zero --%>
<jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
<jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
<jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

<%-- Application lifecycle event handlers go here --%>

<event:application_OnStart>
    <% This scriptlet contains the implementation of the event handler %>
```

```
</event:application_OnStart>

<event:application_OnEnd>
  <% This scriptlet contains the implementation of the event handler %>
</event:application_OnEnd>

<event:session_OnStart>
  <% This scriptlet contains the implementation of the event handler %>
</event:session_OnStart>

<event:session_OnEnd>
  <% This scriptlet contains the implementation of the event handler %>
</event:session_OnEnd>

<%-- Declarations used by the event handlers go here --%>
```

JSP の変換と配布

この章では、主に JSP アプリケーションを Oracle8i に配布して Oracle Servlet Engine で実行する場合の考慮事項と手順について説明します。また、OracleJSP の変換機能全般と、他の環境、特に Oracle Internet Application Server で使用する Apache/JServ 環境への配布についても説明します。

説明するトピックは、次のとおりです。

- [OracleJSP トランスレータの機能](#)
- [Oracle8i への配布時の機能とロジスティックの概要](#)
- [変換と Oracle8i への配布に使用するツールとコマンド](#)
- [サーバー側変換を伴う Oracle8i への配布](#)
- [クライアント側変換を伴う Oracle8i への配布](#)
- [JSP 配布に関するその他の考慮事項](#)

OracleJSP トランスレータの機能

JSP トランスレータでは、JSP ページ実装クラス用の標準 Java コードが生成されます。このクラスは、実際には JSP の機能でラップされたサーブレット・クラスです。

この項では、OracleJSP トランスレータの機能全般について、Apache/JServ や Oracle Internet Application Server など、オンデマンド変換環境での動作に重点を置いて説明します。説明するトピックは、次のとおりです。

- 生成されるコードの機能
- 生成されるパッケージとクラスの名前（オンデマンド変換）
- 生成されるファイルと位置（オンデマンド変換）
- サンプル・ページ実装クラスのソース

重要： この項で説明するパッケージとクラスのネーミング、出力ファイルの場所および生成されるコードに関する実装の詳細は、具体例を示すことのみを意図しています。正確な詳細は OracleJSP リリース 8.1.7 (1.1.0.0.0) にのみ適用され、該当する場合は JSP 1.1 仕様に準拠しています。また、このような詳細は将来のリリースで変更されることがあります。

Oracle Servlet Engine をターゲットとする JSP ページは、セッション・シェルの `publishjsp` コマンドを実行するか（サーバー側変換を伴う配布の場合）、`ojspc` 事前変換ツールを直接実行して（クライアント側変換を伴う配布の場合）、事前に変換する必要があります。どちらの場合も、出力ファイルの場所など、この項の説明とは機能に一部違いがあります。詳細は、6-39 ページの「[Oracle8i での JSP ページの変換と公開（セッション・シェルの `publishjsp`）](#)」および 6-22 ページの「[ojspc 事前変換ツール](#)」を参照してください。

生成されるコードの機能

この項では、JSP ソース（`.jsp` および `.sqljsp` ファイル）の変換時に、OracleJSP トランスレータにより生成されるページ実装クラス・コードの機能全般について説明します。

ページ実装クラス・コードの機能

OracleJSP トランスレータでは、ページ実装クラスのサーブレット・コードの生成時に、標準的なプログラミング・オーバーヘッドの一部が自動的に処理されます。オンデマンド変換モデルの場合も、事前変換モデルの場合も、生成されるコードには次の機能が自動的に組み込まれます。

- 標準 `javax.servlet.jsp.HttpJspPage` インタフェースを実装する OracleJSP コンテナで提供されるラッパー・クラス (`oracle.jsp.runtime.HttpJsp`) が拡張されます (このインタフェースでは、より汎用的な `javax.servlet.jsp.JspPage` インタフェースが拡張され、`javax.servlet.jsp.JspPage` インタフェースでは標準 `javax.servlet.Servlet` インタフェースが拡張されます)。
- `HttpJspPage` インタフェースで指定された `_jspService()` メソッドが実装されます。このメソッドは、通常は広義に「サービス」メソッドと呼ばれ、ページ実装クラスの中核的なメソッドです。JSP ページ内の Java スクリプトレットと式からのコードは、このメソッドの実装に取り込まれます。
- JSP ソース・コードで特に `session=false` と設定されていなければ、HTTP セッションを要求するコードが含まれます (この設定は、`page` ディレクティブで指定できます)。

主要な JSP およびサーブレットのクラスとインタフェースの概要は、[付録 B「サーブレットおよび JSP の技術的な背景情報」](#) を参照してください。

静的テキスト用のインナー・クラス

ページ実装クラスのサービス・メソッド `_jspService()` には、出力コマンドが含まれています。`out.print()` では、暗黙的 `out` オブジェクトがコールされ、JSP ページ内の静的テキストがすべて出力されます。ただし、OracleJSP トランスレータでは、静的テキスト自体はページ実装クラスのインナー・クラスに置かれます。サービス・メソッドの `out.print()` 文では、テキストを出力するインナー・クラスの属性が参照されます。

このインナー・クラスの実装により、ページが変換されコンパイルされるときに `.class` ファイルが追加されます。クライアント側事前変換の使用例では (通常、Oracle8i に配置する場合)、これが配布対象となる `.class` ファイルの追加を意味することに注意してください。

インナー・クラス名は、常に `.jsp` ファイルまたは `.sqljsp` ファイルのベース名に基づく名前になります。たとえば、`mypage.jsp` の場合、インナー・クラス (および `.class` ファイル) の名前には、常に「`mypage`」が含まれます。

注意： OracleJSP トランスレータでは、オプションで Java リソース・ファイルに静的テキストを挿入できます。これは、ページに大量の静的テキストが含まれている場合にメリットとなります。(4-12 ページの「[JSP ページの大量の静的コンテンツに関する回避策](#)」を参照してください。) この機能は、OracleJSP の `external_resource` 構成パラメータ (オンデマンド変換の場合) または `ojspc -extres` オプション (事前変換の場合) を介して要求できます。ホットロード機能 (Oracle8i に配布する場合) を有効化した場合も、静的テキストはリソース・ファイルに格納されます。

静的テキストがリソース・ファイルに格納されていても、インナー・クラスは生成され、その `.class` ファイルは配布する必要があります。(この点に注意する必要があるのは、クライアント側事前変換の使用例の場合のみです。)

生成されるパッケージとクラスの名前 (オンデマンド変換)

Sun Microsystems の『JavaServer Pages Specification, Version 1.1』では、JSP テキストの解析と変換に関する統一プロセスは定義されていますが、生成されるクラスのネーミングについては説明がなく、各 JSP 実装に委ねられています。

この項では、変換中のコード生成時に、OracleJSP でパッケージ名とクラス名がどのように作成されるかについて説明します。

重要： OracleJSP リリース 8.1.7 (1.1.0.0.0) では、URL のパス・ディレクトリと `.jsp` ファイル名 (パッケージ名とクラス名の生成に使用される名前) は、有効な Java パッケージおよびクラス識別子に限定されています。たとえば、パス・ディレクトリまたは `.jsp` 名の先頭には、数値を使用できません。また、`for` や `class` などの Java 予約語をパス・ディレクトリまたは `.jsp` 名に使用すると (`class.jsp` など)、無効になります。このような実装の詳細は、将来のリリースで変更される可能性があります。

パッケージのネーミング

オンデマンド変換の使用例では、ユーザーが JSP ページの要求時に指定する URL パス、特に、ドキュメント・ルートまたはアプリケーション・ルートへの相対パスにより、生成されるページ実装クラスのパッケージ名が決定されます。URL パスに含まれる各ディレクトリは、パッケージの階層レベルを表します。

ただし、生成されるパッケージ名には、URL での表記に関係なく常に小文字が使用されるため注意する必要があります。

たとえば、次の URL があるとします。

```
http://host[:port]/HR/expenses/login.jsp
```

OracleJSP リリース 8.1.7 (1.1.0.0.0) では、この URL により生成されるコードでのパッケージ仕様部は次のようになります（実装の詳細は、将来のリリースで変更されることがあります）。

```
package hr.expenses;
```

JSP ページがドキュメント・ルートまたはアプリケーション・ルート・ディレクトリにあると、URL は次のようになり、パッケージ名は生成されません。

```
http://host[:port]/login.jsp
```

クラスのネーミング

生成されるコード内のクラス名は、.jsp ファイル（または .sqljsp ファイル）のベース名によって決定されます。

たとえば、次の URL があるとします。

```
http://host[:port]/HR/expenses/UserLogin.jsp
```

OracleJSP リリース 8.1.7 (1.1.0.0.0) では、この URL により生成されるコードでのクラス名は次のようになります（実装の詳細は、将来のリリースで変更されることがあります）。

```
public class UserLogin extends ...
```

エンド・ユーザーが URL に入力する大文字 / 小文字の区別は、実際の .jsp または .sqljsp ファイル名のそれと一致するため注意してください。たとえば、UserLogin.jsp や userlogin.jsp が実際のファイル名であれば、そのとおり指定できますが、UserLogin.jsp が実際のファイル名の場合に userlogin.jsp は指定できません。

OracleJSP リリース 8.1.7 (1.1.0.0.0) では、クラス名の大 / 小文字表記はトランスレータによりファイル名の大 / 小文字表記に従って決定されます。次に例を示します。

- UserLogin.jsp の場合、クラス名は UserLogin となります。
- Userlogin.jsp の場合、クラス名は Userlogin となります。
- userlogin.jsp の場合、クラス名は userlogin となります。

クラス名の大 / 小文字表記が重要な場合は、それに従って .jsp ファイルまたは .sqljsp ファイルを命名する必要があります。ただし、ページ実装クラスはエンド・ユーザーには表示されないため、通常は重要ではありません。

生成されるファイルと位置（オンデマンド変換）

この項では、OracleJSP トランスレータにより生成されるファイルとその位置について説明します。事前変換の使用例では、ojspc ではファイルが様々な位置に格納され、独自の関連オプション・セットが用意されています。6-32 ページの「[ojspc の出力ファイル、位置および関連オプションの概要](#)」を参照してください。

ここでは、OracleJSP の複数の構成パラメータについて説明します。それぞれの詳細は、A-13 ページの「[OracleJSP の構成パラメータ（非 OSE）](#)」および A-23 ページの「[OracleJSP の構成パラメータ設定](#)」を参照してください。

OracleJSP により生成されるファイル

この項では、OracleJSP トランスレータにより生成されるファイルごとに、標準の JSP ページ（.jsp ファイル）と SQLJ の JSP ページ（.sqljsp ファイル）について説明します。ファイル名の例として、ファイル Foo.jsp または Foo.sqljsp が変換対象であるとしします。

ソース・ファイルは、次のとおりです。

- SQLJ の JSP ページ（Foo.sqlj など）の場合は、OracleJSP トランスレータにより .sqlj ファイルが生成されます。
- ページ実装クラスとインナー・クラス（Foo.java など）の場合は、.java ファイルが生成されます。このファイルは、OracleJSP トランスレータにより .jsp ファイルから直接生成されるか、SQLJ の JSP ページの場合は SQLJ トランスレータにより .sqlj ファイルから生成されます。（デフォルトでは現在インストールされている Oracle SQLJ トランスレータが使用されますが、OracleJSP の sqljcmd 構成パラメータを使用すると、Oracle SQLJ トランスレータの代替トランスレータまたは代替リリースを指定できます。）

バイナリ・ファイルは、次のとおりです。

- SQLJ の JSP ページの場合は、SQLJ プロファイル用の SQLJ 変換時に 1 つ以上のバイナリ・ファイルが生成されます。デフォルトでは、これらのファイルは .ser Java リソース・ファイルですが、SQLJ の -ser2class オプションを（OracleJSP の sqljcmd 構成パラメータを介して）有効化している場合は、.class ファイルとなります。リソース・ファイルや .class ファイルの名前には、「Foo」が含まれます。
- ページ実装クラスの場合は、Java コンパイラにより .class ファイルが生成されます。（デフォルトでは Java コンパイラは javac ですが、OracleJSP の javaccmd 構成パラメータを使用して代替コンパイラを指定できます。）
- ページ実装クラスのインナー・クラスの場合は、追加の .class ファイルが生成されます。このファイルの名前には「Foo」が含まれます。
- OracleJSP の external_resource 構成パラメータが有効化されている場合は、静的ページ・コンテンツ用にオプションで .res Java リソース・ファイルが生成されます（Foo.res など）。

注意： ページ実装クラス用に生成されるファイルの正確な名前は、将来のリリースで変更されることがありますが、一般的な書式は同じです。ファイル名には常にベース名（前述の例では「Foo」など）が含まれますが、`_Foo.java` や `_Foo.class` のように軽微な変形が含まれることがあります。

OracleJSP トランスレータの出力ファイルの位置

OracleJSP では、変換済み JSP ページの生成やロードに、Web サーバーのドキュメント・リポジトリが使用されます。

デフォルトでは、ルート・ディレクトリは、Web サーバーのドキュメント・ルート・ディレクトリ（Apache/JServ の場合）、またはページが属する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・ディレクトリです。

OracleJSP の `page_repository_root` 構成パラメータを使用すると、代替ルート・ディレクトリを指定できます。

OracleJSP リリース 8.1.7（1.1.0.0.0）では、生成されたファイルは次のように格納されます（実装の詳細は、将来のリリースで変更されることがあります）。

- `.jsp`（または `.sqljsp`）ファイルがルート・ディレクトリに直接格納されていれば、OracleJSP では生成されたファイルがルート・ディレクトリの真下の `_pages` サブディレクトリに格納されます。
- `.jsp`（または `.sqljsp`）ファイルがルート・ディレクトリのサブディレクトリにあれば、生成されるファイル用に `_pages` サブディレクトリに同等のディレクトリ構造が作成されます。

たとえば、`htdocs` ルート・ディレクトリを持つ Apache/JServ 環境を考えます。`.jsp` ファイルが次のディレクトリにあるとします。

```
htdocs/subdir/test
```

生成されたファイルは次のディレクトリに格納されます。

```
htdocs/_pages/subdir/test
```

サンプル・ページ実装クラスのソース

この項では、前項の内容を示す例を使用します。

次の使用例を考えます。

- JSP ページ・コードは、`hello.jsp` ファイルにあります。
- ページは Apache/JServ 環境で実行されます。

- hello.jsp ファイルは次のディレクトリにあります。

htdocs/test

重要： ここで説明するコード生成の詳細は、JSP 1.1 仕様の Oracle 実装に従っています。仕様変更や、Oracle による未指定の部分の実装方法の変更などの結果、詳細は将来変更される可能性があります。

サンプル・ページ・ソース :hello.jsp

次の例は、hello.jsp 内の JSP コードを示しています。

```
<HTML>
<HEAD><TITLE>The Hello User JSP</TITLE></HEAD>
<BODY>
<% String user=request.getParameter("user"); %>
<H3>Welcome <%= (user==null) ? "" : user %>!</H3>
<P><B> Today is <%= new java.util.Date() %>. Have a nice day! :-)</B></P>
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

サンプル：生成されるパッケージとクラス

hello.jsp はルート・ディレクトリ (htdocs) の test サブディレクトリにあるため、OracleJSP リリース 8.1.7 (1.1.0.0.0) では、ページ実装コード内で次のパッケージ名が生成されます。

```
package test;
```

Java クラス名は .jsp ファイルのベース名と (大 / 小文字区別も) 同じなので、ページ実装コード内で次のクラス定義が生成されます。

```
public class hello extends oracle.jsp.runtime.HttpJsp
{
    ...
}
```

(ページ実装クラスはエンド・ユーザーには表示されないため、名前が Java の大 / 小文字区別の規則に準拠していなくても、通常は問題ありません。)

サンプル : 生成されるファイル

hello.jsp は次のディレクトリにあります。

```
htdocs/test/hello.jsp
```

そのため、OracleJSP リリース 8.1.7 (1.1.0.0.0) では、出力ファイルが次のように生成されます (ページ実装クラスの .java ファイルと .class ファイル、およびインナー・クラスの .class ファイル)。

```
htdocs/_pages/test/hello.java
htdocs/_pages/test/hello.class
htdocs/_pages/test/hello$__jsp_StaticText.class
```

注意： これらのファイル名は、特に OracleJSP 1.1.0.0.0 の実装に準拠しており、正確な詳細は将来のリリースで変更されることがあります。ただし、すべてのファイル名には常にベースの「hello」が含まれます。

サンプル・ページ実装コード : hello.java

次の例は、OracleJSP リリース 8.1.7 (1.1.0.0.0) により生成されるページ実装クラスの Java コード (hello.java) を示しています。

```
package test;

import oracle.jsp.runtime.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import java.beans.*;

public class hello extends oracle.jsp.runtime.HttpJsp {

    public final String _globalsClassName = null;

    // ** Begin Declarations

    // ** End Declarations

    public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
```

```
/* set up the intrinsic variables using the pageContext goober:
** session = HttpSession
** application = ServletContext
** out = JspWriter
** page = this
** config = ServletConfig
** all session/app beans declared in globals.jsa
*/
JspFactory factory = JspFactory.getDefaultFactory();
PageContext pageContext = factory.getPageContext( this, request, response, null,
true, JspWriter.DEFAULT_BUFFER, true);
// Note: this is not emitted if the session directive == false
HttpSession session = pageContext.getSession();
if (pageContext.getAttribute(OracleJspRuntime.JSP_REQUEST_REDIRECTED,
PageContext.REQUEST_SCOPE) != null) {
    pageContext.setAttribute(OracleJspRuntime.JSP_PAGE_DONTNOTIFY, "true",
PageContext.PAGE_SCOPE);
    factory.releasePageContext(pageContext);
    return;
}

ServletContext application = pageContext.getServletContext();
JspWriter out = pageContext.getOut();
hello page = this;
ServletConfig config = pageContext.getServletConfig();

try {
    // global beans
    // end global beans

    out.print(__jsp_StaticText.text[0]);
    String user=request.getParameter("user");
    out.print(__jsp_StaticText.text[1]);
    out.print( (user==null) ? "" : user );
    out.print(__jsp_StaticText.text[2]);
    out.print( new java.util.Date() );
    out.print(__jsp_StaticText.text[3]);

    out.flush();

}
catch( Exception e) {
    try {
        if (out != null) out.clear();
    }
    catch( Exception clearException) {
    }
}
```

```

        pageContext.handlePageException( e);
    }
    finally {
        if (out != null) out.close();
        factory.releasePageContext (pageContext);
    }
}

private static class __jsp_StaticText {
    private static final char text [] []=new char[4] [];
    static {
        text[0] =
            "<HTML>\r\n<HEAD><TITLE>The Welcome User
JSP</TITLE></HEAD>\r\n<BODY>\r\n".toCharArray();
        text[1] =
            "\r\n<H3>Welcome ".toCharArray();
        text[2] =
            "!</H3>\r\n<P><B> Today is ".toCharArray();
        text[3] =
            ". Have a nice day! :-)</B></P>\r\n<B>Enter name:</B>\r\n<FORM
METHOD=get>\r\n<INPUT TYPE=\"text\" NAME=\"user\" SIZE=15>\r\n<INPUT TYPE=\"submit\"
VALUE=\"Submit name\">\r\n</FORM>\r\n</BODY>\r\n</HTML>".toCharArray();
    }
}
}

```

Oracle8i への配布時の機能とロジスティックの概要

この項では、JSP アプリケーションを Oracle8i に配布して Oracle Servlet Engine で実行する場合の考慮事項とロジスティックの概要を説明します。説明するトピックは、次のとおりです。

- [Java 用のデータベース・スキーマ・オブジェクト](#)
- [フロントエンド Web サーバーとしての Oracle HTTP Server](#)
- [Oracle Servlet Engine の URL](#)
- [Oracle Servlet Engine 内の JSP アプリケーション用の静的ファイル](#)
- [サーバー側変換とクライアント側変換の比較](#)
- [Oracle8i にホットロードされるクラスの概要](#)

Java 用のデータベース・スキーマ・オブジェクト

Oracle Servlet Engine 内で実行される Java コードには、データベース内の Oracle8i JVM が使用されます。コードは、特定のデータベース・スキーマに 1 つ以上のスキーマ・オブジェクトとしてロードする必要があります。

Java 用のスキーマ・オブジェクトには、次の 3 種類があります。

- ソース・スキーマ・オブジェクト (Java ソース・ファイルに対応)
- クラス・スキーマ・オブジェクト (Java クラス・ファイルに対応)
- リソース・スキーマ・オブジェクト (Java リソース・ファイルに対応)

各スキーマ・オブジェクトは、データベース内の別個のライブラリ・ユニットです。スキーマの ALL_OBJECTS 表を問い合わせると、Java のスキーマ・オブジェクトはそれぞれ JAVA_SOURCE、JAVA_CLASS または JAVA_RESOURCE 型として表示されます。

詳細は、『Oracle8i Java 開発者ガイド』を参照してください。

Java ファイルのロードによるスキーマ・オブジェクトの作成

Java ファイルをデータベースにスキーマ・オブジェクトとしてロードするには、Oracle8i JVM の loadjava ツールを使用します。(6-34 ページの「[loadjava ツールの概要](#)」を参照してください。)

クライアント上でコンパイルして .class ファイルを直接ロードすると、loadjava により .class ファイルはクラス・スキーマ・オブジェクトとしてデータベースに格納されます。

リソース・ファイル（静的 JSP コンテンツ用の .res ファイルや SQLJ 用の .ser プロファイル・ファイルなど）をロードすると、loadjava によりリソース・スキーマ・オブジェクトとしてデータベースに格納されます。

.java (または .sqlj) ソース・ファイルをロードすると、loadjava によりソース・スキーマ・オブジェクトとしてデータベースに格納され、オプションでデータベース内でコンパイルされて、1 つ以上のクラス・スキーマ・オブジェクトが作成されます。

.jsp または .sqljsp ページ・ソース・ファイル（サーバー側変換用）をロードすると、loadjava によりリソース・スキーマ・オブジェクトとして格納されます。サーバー側変換中に (Oracle8i JVM のセッション・シェルの publishjsp コマンドを使用)、サーバー側の loadjava が自動的に起動され、変換およびコンパイル中にソース・スキーマ・オブジェクト、クラス・スキーマ・オブジェクトおよびリソース・スキーマ・オブジェクトが作成されます。

(loadjava およびセッション・シェル・ツールの概要は、6-22 ページの「[変換と Oracle8i への配布に使用するツールとコマンド](#)」を参照してください。)

スキーマ・オブジェクトの完全名と短縮名

Oracle8i では、スキーマ・オブジェクト名に完全名および短縮名という 2 つの書式があります。

完全名は、該当する場合にスキーマ・オブジェクト名として使用される完全修飾名です。ただし、完全名が 31 文字を超えていたり、不正な文字またはデータベースのキャラクタ・セットの文字に変換できない文字が含まれていると、Oracle8i サーバーでは完全名が短縮名に変換され、スキーマ・オブジェクト名として使用され、両方の名前とその変換方法が追跡されます。完全名が 31 文字以内で、不正な文字や変換できない文字が含まれていなければ、完全名がスキーマ・オブジェクト名として使用されます。

これらの事項と、短縮名から完全名を取得し、完全名から短縮名を取得する DBMS_JAVA プロシージャなど、他のファイル命名に関する考慮事項の詳細は、『Oracle8i Java 開発者ガイド』を参照してください。

ロード中の Java スキーマ・オブジェクト・パッケージの決定

Java ファイルをデータベースにロードする間に、loadjava ツールでは次のロジックを使用して、作成する Java スキーマ・オブジェクト用のパッケージが決定されます。

- ソース・スキーマ・オブジェクト（.java および .sqlj ファイルから作成）とクラス・スキーマ・オブジェクト（.class ファイルから、または .java ファイルのコンパイラにより作成）の場合、スキーマ・パッケージは Java コード内のパッケージ情報により決定されます。

たとえば、dir1.dir2 パッケージが指定されていて SCOTT スキーマにロードされるクラス Foo は、このスキーマに次のように格納されます。

```
SCOTT:dir1/dir2/Foo
```

注意： ojspc ツール（クライアント側変換を伴う Oracle8i への配置用）を使用して JSP ページを事前変換する場合は、生成される .java ファイルのパッケージを ojspc -packageName オプションで指定できます。

- リソース・スキーマ・オブジェクト（.res および .ser Java リソース・ファイルから作成されるものなど）の場合、スキーマ・パッケージは loadjava コマンドライン上（Java リソース・ファイルを直接ロードする場合）または JAR ファイル内（Java リソース・ファイルを JAR ファイルの一部としてロードする場合）のパス情報により決定されます。

たとえば、SCOTT スキーマに dir3/dir4/abcd.res としてロードされる .res ファイルは、スキーマ・オブジェクトに次のように格納されます。

```
SCOTT:dir3/dir4/abcd.res
```

スキーマ・オブジェクトの公開

Oracle Servlet Engine 内で実行される JSP ページ（またはサーブレット）は、すべて「公開」する必要があります。これは、実行可能な Java コード（クラス・スキーマ・オブジェクト）に、Oracle8i JVM の JNDI ネームスペース内のエントリを介してアクセスできるようにするための処理です。

JSP ページを公開すると、そのページ実装クラスのスキーマ・オブジェクトが、サーブレットのパスにリンクされます（また、オプションでデフォルト以外のサーブレット・コンテキスト・パスにもリンクされます）。サーブレット・パス（および、該当する場合はコンテキスト・パス）は、エンド・ユーザーがページにアクセスして実行するときに指定する URL の一部となります。詳細は、6-14 ページの「[Oracle Servlet Engine の URL](#)」を参照してください。

JSP ページを公開するには、Oracle8i のセッション・シェルで、「サーバー側変換を伴う配布」使用例の場合は `publishjsp` コマンド、「クライアント側変換を伴う配布」使用例の場合は `publishservlet` コマンドを使用します。6-39 ページの「[Oracle8i での JSP ページの変換と公開（セッション・シェルの `publishjsp`）](#)」または 6-59 ページの「[Oracle8i での変換済み JSP ページの公開（セッション・シェルの `publishservlet`）](#)」を参照してください。

フロントエンド Web サーバーとしての Oracle HTTP Server

Oracle Servlet Engine 内で実行される JSP ページとサーブレットへのアクセスには、通常は Oracle HTTP Server（Apache により駆動）とその `mod_ose` モジュールが使用されますが、OSE 自体を Web サーバーとして使用することもできます。

Oracle HTTP Server と `mod_ose` のロールの詳細は、2-5 ページの「[Apache により駆動される Oracle HTTP Server のロール](#)」を参照してください。

Oracle Servlet Engine の URL

通常のサーブレットの URL と同様に、Oracle Servlet Engine 内で実行される JSP ページを起動する URL は、次の 2 つの構成要素（およびホスト名とポート）の組合せからなっています。

- サーブレット・コンテキストの作成時に決定された、OSE 内のサーブレット・コンテキストのコンテキスト・パス
- JSP ページの公開時に決定された、OSE 内の JSP ページのサーブレット・パス（通常は「仮想パス」と呼ばれます）

OSE のデフォルト・コンテキスト `/webdomains/contexts/default` のコンテキスト・パスは、単に次のようになります。

/

Oracle8i セッション・シェルの `createcontext` コマンドを使用して作成する他の OSE サーブレット・コンテキストのコンテキスト・パスは、作成時に `createcontext`

-virtualpath オプションで指定したパスになります。（このパスは表記規則どおりですが、コンテキスト・パスがコンテキスト名と同じになるように指定しなくてもかまいません。）

注意： createcontext コマンドを実行するたびに、-virtualpath オプションを指定する必要があります。

セッション・シェルの createcontext コマンドの概要は、『Oracle8i Java Tools リファレンス』を参照してください。Oracle8i のセッション・シェルの概要は、6-36 ページの「[sess_sh セッション・シェル・ツールの概要](#)」を参照してください。

サブレット・パス（JSP ページの「仮想パス」）は、次のように JSP ページの公開方法により決定されます。

- セッション・シェルの publishjsp コマンド（サーバー側変換用）を使用すると、publishjsp -virtualpath オプションにより決定されます。このオプションを指定しなければ、デフォルトで指定されるスキーマ・パスと同じになります。
- セッション・シェルの publishservlet コマンド（クライアント側変換後）を使用する場合は、publishservlet -virtualpath オプションにより決定されます（JSP ページに publishservlet を使用する場合は、このオプションを指定する必要があります）。

6-39 ページの「[Oracle8i での JSP ページの変換と公開（セッション・シェルの publishjsp）](#)」または 6-59 ページの「[Oracle8i での変換済み JSP ページの公開（セッション・シェルの publishservlet）](#)」を参照してください。

例 1 たとえば、次のように、サブレット・パス（仮想パス）で OSE のデフォルト・コンテキストに公開される JSP ページがあるとします。

mydir/mypage.jsp

このページへのアクセスは次のようになります。

http://host[:port]/mydir/mypage.jsp

このページには、mydir/mypage2.jsp など、アプリケーション内の他のページから次のどちらかの方法でアクセスできます（最初の例はページ相対パス、2 番目の例はアプリケーション相対パスです）。

```
<jsp:include page="mypage.jsp" flush="true" />
```

```
<jsp:include page="/mydir/mypage.jsp" flush="true" />
```

例 2 次のように作成されるサーブレット・コンテキストがあるとします（\$ はセッション・シェル・プロンプトです）。

```
$ createcontext -virtualpath mycontext /webdomains mycontext
```

この場合の操作は、次のとおりです。

- サーブレット・コンテキスト /webdomains/contexts/mycontext が作成されます（/webdomains ドメインにあるサーブレット・コンテキストは、すべて /webdomains/contexts にアクセスします）。
- コンテキスト名（mycontext）と同じになるようにコンテキスト・パスが指定されます。

mydir/mypage.jsp が mycontext サーブレット・コンテキストに公開される場合は、次のようにアクセスされます。

```
http://host[:port]/mycontext/mydir/mypage.jsp
```

このページには、mydir/mypage2.jsp など、アプリケーション内の他のページから次のどちらかの方法でアクセスできます（最初の例はページ相対パス、2 番目の例はアプリケーション相対パスです）。

```
<jsp:include page="mypage.jsp" flush="true" />
```

```
<jsp:include page="/mydir/mypage.jsp" flush="true" />
```

動的 jsp:include 文の構文は、例 1 の場合と同じです。異なるサーブレット・コンテキストを使用しても、ページのコンテキスト相対パスは変わりません。

例 3 次のように作成されるサーブレット・コンテキストがあるとします（\$ はセッション・シェル・プロンプトです）。

```
$ createcontext -virtualpath mywebapp /webdomains mycontext
```

この場合の操作は、次のとおりです。

- 例 2 と同様に、サーブレット・コンテキスト /webdomains/contexts/mycontext が作成されます。
- ただし、コンテキスト名とは異なるコンテキスト・パス mywebapp が定義されます。これは、コンテキスト名ではなく、URL に使用されるコンテキスト・パスです。

この場合、mydir/mypage.jsp が mycontext サーブレット・コンテキストに公開される場合は、次のようにアクセスされます。

```
http://host[:port]/mywebapp/mydir/mypage.jsp
```

このページには、mydir/mypage2.jsp など、アプリケーション内の他のページから次のどちらかの方法でアクセスできます（最初の例はページ相対パス、2 番目の例はアプリケーション相対パスです）。

```
<jsp:include page="mypage.jsp" flush="true" />
```

```
<jsp:include page="/mydir/mypage.jsp" flush="true" />
```

Oracle Servlet Engine 内の JSP アプリケーション用の静的ファイル

この項では、HTML ファイルなど、Oracle Servlet Engine 内で実行される JSP アプリケーションに使用される静的ファイルの必須位置について説明します。

この項の情報は、Oracle HTTP Server（Apache により駆動）を OSE のフロントエンド Web サーバーとして使用する場合も、OSE を直接使用する場合も適用されます。

動的挿入および転送の対象ファイル

Oracle Servlet Engine 内で実行される JSP アプリケーション内で、動的な include または forward のターゲット（jsp:include または jsp:forward）となる静的ファイルは、そのアプリケーションのサーブレット・コンテキストに対応する OSE のドキュメント・ルート・ディレクトリに、手動で移動またはコピーする必要があります。OSE サーブレット・コンテキストを（セッション・シェルの createcontext コマンドで）作成するときに、createcontext -docroot オプションでドキュメント・ルート・ディレクトリを指定します。各 OSE ドキュメント・ルート・ディレクトリは、Oracle8i JVM の JNDI ネームスペースにリンクされています。

OSE ドキュメント・ルート・ディレクトリは、データベースの外側にあります。静的ファイルに関する JNDI 検索メカニズムは、データベースが常駐するサーバーのファイル・システムのフロントエンドです。

OSE のデフォルト・サーブレット・コンテキスト /webdomains/contexts/default のドキュメント・ルートは、次のとおりです。

```
$ORACLE_HOME/jis/public_html
```

セッション・シェルの createcontext コマンドでサーブレット・コンテキストを追加作成するたびに、createcontext -docroot オプションでドキュメント・ルート・ディレクトリを指定できます。（セッション・シェルの createcontext コマンドの詳細は、『Oracle8i Java Tools リファレンス』を参照してください。）

注意： JSP アプリケーションを Apache から OSE に移行する場合は、OSE のサブレット・コンテキストのドキュメント・ルートを Apache のドキュメント・ルートにマップするのではなく、Apache のドキュメント・ルートから OSE のサブレット・コンテキストのドキュメント・ルートに、静的ファイルをコピーすることをお勧めします。ドキュメント・ルートをマップすると、混乱する恐れがあります。

静的挿入の対象ファイル

他の JSP ページの場合も HTML ファイルのような静的ファイルの場合も、JSP ページにより（include ディレクティブを介して）静的に挿入されるファイルは、すべて変換中に OracleJSP トランスレータからアクセス可能にする必要があります。

OSE のターゲットとなる JSP アプリケーションの場合は、次の 2 つの変換方法があります。

■ サーバー側変換

これは、.jsp ファイルを Java リソースとしてデータベースにロードし、publishjsp を使用してサーバー内で OracleJSP トランスレータを起動する場合です。（6-38 ページの「[サーバー側変換を伴う Oracle8i への配布](#)」を参照してください。）

この場合は、loadjava を使用し、事前に静的ファイルをリソース・スキーマ・オブジェクトとしてデータベースにロードする必要があります。

■ クライアント側変換

これは、ojspc を使用してクライアント上で .jsp ファイルを変換し、生成されたコンポーネントをデータベースにロードする場合です。

この場合、静的ファイルをサーバー側で処理する必要はありません。必要なのは、変換中にクライアント上の ojspc からアクセスできるようにすることのみです。（アプリケーション相対の静的 include ディレクティブについては、6-26 ページの「[ojspc のオプションの説明](#)」にある ojspc -appRoot オプションの説明を参照してください。）

サーバー側変換とクライアント側変換の比較

JSP ページを Oracle8i に配布して Oracle Servlet Engine 内で実行しようとする開発者は、サーバー側またはクライアント側で変換できます。

サーバー側変換を伴う配布には、次の 2 つの必須ステップがあります。

1. loadjava を実行し、JSP ページのソース（.jsp または .sqljsp ファイル）を Oracle8i にリソース・スキーマ・オブジェクトとしてロードします。（必須の Java クラスや他の必須の JSP ページをすべてロードする必要があります。）

2. セッション・シェルの `publishjsp` コマンドを実行します。これにより、次の処理が自動的に実行されます。
 - JSP ページのソースがページ実装クラス用の Java コードに変換されます (SQLJ の JSP ページの場合は、最初に SQLJ ソース・ファイルが生成され、SQLJ トランスレータが起動されます)。
 - Java コードがコンパイルされ、1 つ以上のクラス・ファイルが生成されます。
 - オプションでページ実装クラスがホットロードされます (`publishjsp -hotload` オプションを指定した場合)。
 - ページ実装クラスが、データベース内で実行できるように公開されます。

このステップでは、生成されたすべての `.java` ファイル (および `.sqljsp` ページの場合は `.sqlj` ファイル)、`.class` ファイルおよびリソース・ファイル用に、それぞれソース・スキーマ・オブジェクト、クラス・スキーマ・オブジェクトおよびリソース・スキーマ・オブジェクトも生成されます。

詳細は、6-38 ページの「[サーバー側変換を伴う Oracle8i への配布](#)」を参照してください。

クライアント側変換を伴う配布の場合、次のように必須ステップが 3 つ、オプション・ステップが 1 つあります。

1. OracleJSP の事前変換ツール `ojspc` を実行します。これにより次の処理が実行されます。
 - JSP ページ・ソースがページ実装クラス用の Java コードに変換されます。(SQLJ の JSP ページの場合は、`ojspc` により最初に SQLJ ソース・ファイルが生成されてから、SQLJ トランスレータが起動され、Java コードが生成されます。)
 - `ojspc -extres` および `-hotload` オプションに応じて、オプションで静的テキスト用の Java リソース・ファイルが生成されます。
 - Java コードがコンパイルされ、クラス・ファイルが生成されます。
2. Oracle8i の `loadjava` ユーティリティを実行し、クラス・ファイルとリソース・ファイルを Oracle8i にクラス・スキーマ・オブジェクトおよびリソース・スキーマ・オブジェクトとしてロードします。
3. オプションで、Oracle8i セッション・シェルの `java` コマンドを使用してページ実装クラスの `main()` メソッドを実行し、クラスをホットロードします (変換時に `ojspc -hotload` オプションを有効化した場合)。
4. セッション・シェルの `publishservlet` コマンドを実行し、ページ実装クラスをデータベース内で実行できるように公開します。

詳細は、6-50 ページの「[クライアント側変換を伴う Oracle8i への配布](#)」を参照してください。

Oracle JDeveloper を使用している場合は、JDeveloper で提供される OracleJSP トランスレータを使用してクライアント側で変換してから、生成されたクラスとリソースをステップ 2 ~ 4 のように配布する方が便利です。

ただし、JDeveloper を使用していない場合は、サーバー側で変換する方が便利です。これは、セッション・シェルの `publishjsp` コマンドでは、変換、オプションのホットロードおよび公開が一度に実行されるためです。

また、次の状況でも、サーバー側での変換が必要になる場合があります。

- 必須ライブラリをクライアント側で使用できない場合
- 実行時に使用される正確なクラス・セットに対してコンパイルする必要がある場合

Oracle8i にホットロードされるクラスの概要

Oracle8i JVM には、`static final` 変数（定数）の使用効率を高めるために、ホットロード・クラスと呼ばれる機能が用意されています。この機能は、ホットロードされるクラスが複数の同時ユーザーにより使用される場合に常に関連します。

Oracle8i JVM のデータベース・セッションごとに、別個の JVM が起動されます。通常、各セッションは、セッション・スペース内、またはリテラル文字列の場合は共有メモリー内のインターン表と呼ばれるハッシュ表に、すべての `static final` 変数の独自コピーを取り出します。インターン表でのリテラル文字列の使用は、セッション間で同期化されます。

リテラル文字列の処理は、特に JSP ページに関連しています。デフォルト（ホットロードなし）では、JSP ページ内の静的テキストは最終的にリテラル文字列で表されます。

注意： この項では、OracleJSP の事前変換ツール (`ojspc`)、Oracle のセッション・シェル・ツール (`sess_sh`) およびセッション・シェルの `publishjsp` コマンドに言及しています。各ツールの概要は、6-22 ページの「[変換と Oracle8i への配布に使用するツールとコマンド](#)」を参照してください。

ホットロードの有効化と実行

JSP ページのホットロード機能は、変換時に `ojspc -hotload` オプション（クライアント側変換用）または `publishjsp -hotload` オプション（サーバー側変換用）を介して有効化されます。

`-hotload` オプションを有効化すると、OracleJSP トランスレータにより次の処理が実行されます。

- ホットロード・メソッドと、それを起動する `main()` メソッドが作成され、ホットロードできるようにページ実装クラスにコードが生成されます。
- 静的テキストが Java リソース・ファイルに書き込まれます。（前述のオプションを有効化しなければ、静的テキストはページ実装クラスのインナー・クラスに書き込まれます。）

ホットロード自体は、次のように実行されます。

- クライアント側変換を伴う配布の場合は、追加の配布ステップとしてホットロードする必要があります。ojspc -hotload オプションを有効化し、ページをデータベースにロードして変換した後、ページを公開する前に、セッション・シェルの java コマンドを使用して、ページ実装クラスの main() メソッドを起動する必要があります。この処理の詳細は、6-50 ページの「[クライアント側変換を伴う Oracle8i への配布](#)」を参照してください。
- サーバー側変換を伴う配布の場合は、publishjsp -hotload オプションを有効化すると、ホットロードが publishjsp 機能の一部として自動的に実行されます。

セッション・シェルの java コマンドを介して直接、または publishjsp コマンドを介して間接的に、ページ実装クラスをホットロードすると、実際にはインナー・クラスの静的テキストがデータベース内の複数の JVM 間で共有可能になるのみです。

ホットロードの機能とメリット

クラスのホットロードには、次のロジスティック上の機能とメリットがあります。

- トランスレータにより、静的テキストを含む Java リソースを静的初期化機能に読み込むコードが生成され、静的テキストを表す char 配列が初期化されます。
- ホットロード中には、ホットロードされる各インナー・クラスが一度のみ初期化され、静的な JSP テキストが静的な Java の char 配列に一度のみ変換されます。

これらの char 配列は、同期化されたインターン表に格納されるかわりに、同期化なしに全セッション間で共有されるグローバル領域の他の場所に格納されます（これが可能なのは、変数に変更がないことが認識されるためです）。

ホットロードにより、同期化や高コストの他のオーバーヘッドが回避され、Oracle Servlet Engine 内で実行される JSP ページのランタイム・パフォーマンスと拡張性を大幅に改善できます。また、ホットロードされたクラスの参照時には、クラス初期化機能は再実行されません。セッションは、リテラル文字列と他の static final 変数への一時的なアクセス権を持ちます。

ホットロードにより、個々の JSP ページのパフォーマンスが改善されるのみでなく、サーバーの CPU 使用量全体が削減されます。

注意： 同時に複数のユーザーに使用されない JSP ページや、リテラル文字列が少数しか含まれていない JSP ページは、ホットロードしてもパフォーマンスはほとんどまたはまったく改善されません。

変換と Oracle8i への配布に使用するツールとコマンド

JSP ページを変換して Oracle8i に配布する場合に、必要に応じて使用できるように、次のツールが用意されています。実装方法は、オペレーティング・システムに応じて異なります (Solaris 用のシェル・スクリプトや Windows NT 用の .bat ファイルなど)。

- `ojspc` (OracleJSP の事前変換ツール)
- `loadjava` (JSP ページまたは Java ファイルをデータベースにロードするためのツール)
- `sess_sh` (Oracle8i のセッション・シェル・ツール)

クライアント側変換を伴う配布には、前述の 3 つのツールすべてが必要です。`ojspc` を使用してクライアント側で JSP ページを事前に変換し、`loadjava` を使用して変換済みのページを Oracle8i にロードし、セッション・シェルの `publishservlet` コマンドを使用して公開します。

サーバー側変換を伴う配布には、`ojspc` は不要です。`loadjava` を使用して未変換の JSP ページを Oracle8i にロードし、セッション・シェルの `publishjsp` コマンドを使用して変換および公開します。

`loadjava` および `sess_sh` ツールは Oracle8i JVM 環境向けの汎用ツールですが、`ojspc` は JSP ページ専用です。

注意：

- もう 1 つのツールである Oracle8i JVM Accelerator が関連するのは、アプリケーションをそのままコンパイルして Oracle8i で実行する必要がある場合です。このツールは `ncomp` として起動されます。詳細は、『Oracle8i Java Tools リファレンス』を参照してください。
 - この項で説明するツールは、`[ORACLE_HOME]/bin` ディレクトリにあります。
-
-

ojspc 事前変換ツール

クライアント側変換を行って JSP アプリケーションを Oracle8i に配布する最初のステップは、OracleJSP の事前変換ツール `ojspc` を実行することです。

その後、次の項で紹介する `loadjava` を使用して、変換後の .class ファイルとリソース・ファイル (存在する場合) を、それぞれクラス・スキーマ・オブジェクトおよびリソース・スキーマ・オブジェクトとしてデータベースにロードします。

説明するトピックは、次のとおりです。

- [ojspc の機能の概要](#)
- [ojspc のオプション概要一覧](#)
- [ojspc のコマンドライン構文](#)

- [ojspc のオプションの説明](#)
- [ojspc の出力ファイル、位置および関連オプションの概要](#)

注意： その他にも、JSP ページの事前変換における `ojspc` の使用については、中間層階層などの使用例があります。6-64 ページの「[非 OSE 環境向けの事前変換のための ojspc の使用](#)」を参照してください。

ojspc の機能の概要

単純な (SQLJ の JSP ではなく) JSP ページの場合、`ojspc` のデフォルトの機能は次のとおりです。

- 引数として `.jsp` ファイルを取ります。
- OracleJSP トランスレータを起動し、`.jsp` ファイルを Java のページ実装クラス・コードに変換し、`.java` ファイルを生成します。ページ実装クラスには、静的ページ・コンテンツ用のインナー・クラスが含まれます。
- Java コンパイラを起動して `.java` ファイルをコンパイルし、2 つの `.class` ファイル (ページ実装クラス自体に 1 つ、インナー・クラス用に 1 つ) を生成します。

また、SQLJ の JSP ページに対する `ojspc` のデフォルトの機能は次のとおりです。

- 引数として `.jsp` ファイルのかわりに `.sqljsp` ファイルを取ります。
- OracleJSP トランスレータを起動し、`.sqljsp` ファイルをページ実装クラス (およびインナー・クラス) 用の `.sqlj` ファイルに変換します。
- Oracle SQLJ トランスレータを起動し、`.sqlj` ファイルを変換します。これにより、ページ実装クラス (およびインナー・クラス) 用の `.java` ファイルと SQLJ の「プロファイル」ファイルが生成されます。SQLJ のデフォルトのプロファイル・ファイルは、`.ser` Java リソース・ファイルです。

SQLJ プロファイルの詳細は、『Oracle8i SQLJ 開発者ガイドおよびリファレンス』を参照してください。

- Java コンパイラを起動して `.java` ファイルをコンパイルし、2 つの `.class` ファイル (ページ実装クラス自体に 1 つ、インナー・クラス用に 1 つ) を生成します。

一部の状況では (後述の `-hotload` および `-extres` オプションの説明を参照)、`ojspc` のオプションは OracleJSP トランスレータに対して、静的ページ・コンテンツをページ実装クラスのインナー・クラスに挿入するかわりに、このコンテンツ用の `.res` Java リソース・ファイルを生成するように指示します。ただし、インナー・クラスも作成され、ページ実装クラスとともに配布する必要があります。

OracleJSP トランスレータの出力 (特に、オンデマンド変換の使用例での出力) の概要は、6-6 ページの「[生成されるファイルと位置 \(オンデマンド変換\)](#)」を参照してください。

注意： ojspc コマンドライン・ツールは、oracle.jsp.tool.Jspc クラスを起動するフロントエンド・ユーティリティです。

ojspc のオプション概要一覧

表 6-1 は、ojspc 事前変換ユーティリティでサポートされるオプションを示しています。これらのオプションの詳細は、6-26 ページの「[ojspc のオプションの説明](#)」を参照してください。

2 列目は、オンデマンド変換環境（Apache/JServ など）のための互換または関連 OracleJSP 構成パラメータを示しています。

注意： ブール ojspc オプションを有効化するには、true に設定するのではなく、オプション名のみを入力します。true に設定するとエラーになります。

表 6-1 ojspc 事前変換ユーティリティのオプション

オプション	関連する OracleJSP 構成パラメータ	説明	デフォルト
-addclasspath	classpath（関連はあるが異なる機能）	javac に関する追加の classpath エントリ	空（追加のパス・エントリなし）
-appRoot	該当なし	ページからのアプリケーション相対の静的 include ディレクティブのアプリケーション・ルート・ディレクトリ	カレント・ディレクトリ
-debug	emit_debuginfo	ojspc に対して、デバッグ用に元の .jsp ファイルへのライン・マップを生成するように指示するブール	false
-d	page_repository_root	ojspc で生成されたバイナリ・ファイル（.class およびリソース）を格納する場所	カレント・ディレクトリ
-extend	該当なし	生成されたページ実装クラスで拡張するクラス	空
-extres	external_resource	ojspc に対して、.jsp ファイルからの静的テキスト用に外部リソース・ファイルの生成を指示するブール	false

表 6-1 ojspc 事前変換ユーティリティのオプション (続き)

オプション	関連する OracleJSP 構成パラメータ	説明	デフォルト
-hotload (OSE 専用)	該当なし	ojspc に対して、ページ実装クラス内のコードを実装してホットロード可能にするように指示するブール	false
-implement	該当なし	生成されたページ実装クラスで実装するインタフェース	空
-noCompile	javaccmd	ojspc に対して、生成されたページ実装クラスをコンパイルしないように指示するブール	false
-packageName	該当なし	生成されるページ実装クラスのパッケージ名	空 (.jsp ファイルの位置ごとにパッケージ名を生成)
-S-<sqlj option>	sqljcmd	-s 接頭辞と後続の Oracle SQLJ オプション (.sqljsp ファイル用)	空
-srcdir	page_repository_root	ojspc で生成されたソース・ファイル (.java および .sqlj) を格納する場所	カレント・ディレクトリ
-verbose	該当なし	ojspc に対して、実行時にステータス情報を出力するように指示するブール	false
-version	該当なし	ojspc に対して、OracleJSP のバージョン・ナンバーを表示するように指示するブール	false

ojspc のコマンドライン構文

ojspc の一般的なコマンドライン構文は、次のとおりです (% は UNIX プロンプトとします)。

```
% ojspc [option_settings] file_list
```

ファイル・リストには、.jsp ファイルまたは .sqljsp ファイルを指定できます。

構文に関する注意事項は、次のとおりです。

- 複数の .jsp ファイルを変換する場合は、すべてのファイルで同じキャラクタ・セットを使用する必要があります (デフォルトで設定されるか、page ディレクティブの contentType 設定を介して指定します)。

- ファイル・リストでは、ファイル名を空白で区切ります。
- オプション名とオプション値は空白で区切ります。
- オプション名には大 / 小文字区別がありませんが、通常、オプション値（パッケージ名、ディレクトリ・パス、クラス名およびインタフェース名など）には大 / 小文字区別があります。
- デフォルトで無効化されているブール・オプションを有効化するには、オプション名のみを入力します。たとえば、`-hotload true` ではなく `-hotload` と入力します。

次に例を示します。

```
% ojspc -d /myapp/mybindir -srcdir /myapp/mysrcdir -hotload MyPage.sqljsp MyPage2.jsp
```

ojspc のオプションの説明

この項では、ojspc のオプションの詳細を説明します。

-addclasspath（完全修飾パス。ojspc のデフォルト：空）

このオプションを使用して、生成されたページ実装クラス・ソースのコンパイル時に `javac` で使用する `classpath` エントリを追加指定します。このオプションを使用しない場合は、`javac` ではシステムの `classpath` のみが使用されます。

（`-addclasspath` 設定は、SQLJ JSP ページ用の SQLJ トランスレータでも使用されます。）

注意： オンデマンド変換の使用例では、OracleJSP の `classpath` 構成パラメータにより、関連する異なる機能が提供されます。A-13 ページの「[OracleJSP の構成パラメータ（非 OSE）](#)」を参照してください。

-appRoot（完全修飾パス。ojspc のデフォルト：カレント・ディレクトリ）

このオプションを使用して、アプリケーション・ルート・ディレクトリを指定します。デフォルトは、ojspc を実行したときのカレント・ディレクトリです。

指定したアプリケーション・ルート・ディレクトリのパスは、次のように使用されます。

- 変換対象ページ内の静的 `include` ディレクティブに使用されます。指定したディレクトリ・パスは、変換済みページの `include` ディレクティブのすべてのアプリケーション相対（コンテキスト相対）パスに付加されます。
- ページ実装クラスのパッケージの決定に使用されます。このパッケージは、アプリケーション・ルート・ディレクトリに対する変換対象ファイルの相対位置に基づいています。このパッケージにより、出力ファイルの位置が決定されます。（6-32 ページの「[ojspc の出力ファイル、位置および関連オプションの概要](#)」を参照してください。）

このオプションは必須です。たとえば、ojspc を他のディレクトリから実行した場合にも、挿入ファイルを検索できるようにするためです。

次に例を示します。

- 次のファイルを変換する必要があるとします。

```
/abc/def/ghi/test.jsp
```

- ojspc をカレント・ディレクトリ /abc から次のように実行します（% は UNIX プロンプトとします）。

```
% cd /abc  
% ojspc def/ghi/test.jsp
```

- test.jsp ページには、次の include ディレクティブがあります。

```
<%@ include file="/test2.jsp" %>
```

- test2.jsp ページは、次のように /abc ディレクトリにあります。

```
/abc/test2.jsp
```

デフォルトのアプリケーション・ルート設定はカレント・ディレクトリ（この例では /abc ディレクトリ）のため、この場合は -appRoot 設定は不要です。include ディレクティブでは、アプリケーション相対の /test2.jsp 構文（先頭の「/」に注意）が使用されるため、挿入ページは /abc/test2.jsp として検出されます。

この例のパッケージは def.ghi で、単に ojspc を実行したときのカレント・ディレクトリに対する test.jsp の相対位置に基づいています（カレント・ディレクトリは、デフォルトのアプリケーション・ルートです）。出力ファイルは、この相対位置に従って格納されます。

ただし、ojspc を他のディレクトリ（この例では /home/mydir）から実行する場合は、-appRoot を次のように設定する必要があります。

```
% cd /home/mydir  
% ojspc -appRoot /abc /abc/def/ghi/test.jsp
```

パッケージは同じく def.ghi で、指定したアプリケーション・ルート・ディレクトリに対する test.jsp の相対位置に基づいています。

注意： 通常は、アプリケーション・ルート・ディレクトリを、変換済み JSP ページがあるディレクトリの親ディレクトリのなんらかのレベルに指定します。

-d (完全修飾パス。ojspc のデフォルト: カレント・ディレクトリ)

このオプションを使用して、ojspc で生成されるバイナリ・ファイル、つまり .class ファイルと Java リソース・ファイルを格納するベース・ディレクトリを指定します。

(SQLJ トランスレータにより SQLJ の JSP ページ用に生成される .ser プロファイル・ファイルと同様に、-extres および -hotload オプションにより静的コンテンツ用に生成される .res ファイルは Java リソース・ファイルです。)

指定したパスは、単に（アプリケーション相対またはページ相対パスではなく）ファイル・システム・パスとして使用されます。

Windows NT など、ディレクトリ名に空白を使用できる環境では、ディレクトリ名を引用符で囲みます。

パッケージに応じて、指定したディレクトリのサブディレクトリが必要な場合は自動的に作成されます。詳細は、6-32 ページの「[ojspc の出力ファイル、位置および関連オプションの概要](#)」を参照してください。

デフォルトでは、カレント・ディレクトリ（ojspc を実行したときのカレント・ディレクトリ）が使用されます。

生成されたファイルが一目でわかるように、このオプションを使用して、生成されたバイナリ・ファイルを空のディレクトリに格納することをお勧めします。

注意： オンデマンド変換の使用例では、OracleJSP の `page_repository_root` 構成パラメータにより関連機能が提供されます。A-13 ページの「[OracleJSP の構成パラメータ（非 OSE）](#)」を参照してください。

-debug（ブール。ojspc のデフォルト: false）

このフラグを有効化して、ojspc に対して、デバッグ用に元の .jsp ファイルへのライン・マップを生成するように指示します。このフラグを有効化しなければ、生成されたページ実装クラスにマッピングされます。

これは、Oracle JDeveloper の使用時など、ソース・レベルの JSP デバッグに役立ちます。

注意： オンデマンド変換の使用例では、OracleJSP の `emit_debuginfo` 構成パラメータにより同じ機能が提供されます。A-13 ページの「[OracleJSP の構成パラメータ（非 OSE）](#)」を参照してください。

-extend（完全修飾 Java クラス名。ojspc のデフォルト: 空）

このオプションを使用して、生成されたページ実装クラスで拡張される Java クラスを指定します。

-extres (ブール。ojspc のデフォルト: false)

このフラグを有効化し、ojspc に対して、生成された静的コンテンツ (静的 HTML コードを出力する Java 出力コマンド) を、生成されたページ実装クラスのインナー・クラスではなく、Java リソース・ファイルに格納するように指示します。

リソース・ファイル名は、JSP ページ名に基づいています。リリース 8.1.7 では、JSP 名と同じ名前になりますが、.res 接尾辞が付きます (たとえば、MyPage.jsp の変換により、通常の出力に加えて MyPage.res が作成されます)。ただし、正確な実装は将来のリリースで変更されることがあります。

リソース・ファイルは、.class ファイルと同じディレクトリに格納されます。

1 ページに大量の静的コンテンツが含まれている場合は、この技法を使用すると変換が高速になり、ページの実行も高速になることがあります。詳細は、4-12 ページの「[JSP ページの大量の静的コンテンツに関する回避策](#)」を参照してください。

注意:

- インナー・クラスも引き続き作成されるため、配布する必要があります。
 - オンデマンド変換の使用例では、OracleJSP の external_resource 構成パラメータにより同じ機能が提供されます。A-13 ページの「[OracleJSP の構成パラメータ \(非 OSE\)](#)」を参照してください。
-
-

-hotload (ブール。ojspc のデフォルト: false) (**OSE 専用**)

このフラグを有効化し、ホットロードを可能にします。このフラグが関連するのは、変換済みページを Oracle8i にロードして Oracle Servlet Engine で実行する場合のみです。

-hotload フラグは、ojspc に対して次の処理を指示します。

1. -extres 機能を実行し、静的出力を Java リソース・ファイルに書き込みます (前述の -extres の説明を参照)。
2. ホットロード可能になるように、生成されたページ実装クラスに main() メソッドとホットロード・メソッドを作成します。

ホットロードの概要は、6-20 ページの「[Oracle8i にホットロードされるクラスの概要](#)」を参照してください。ホットロード手順の実行方法 (ホットロードの有効化後) は、6-58 ページの「[Oracle8i でのページ実装クラスのホットロード](#)」を参照してください。

注意： ホットロードを有効化せずに静的コンテンツをリソース・ファイルに書き込むには（ページを OSE で実行しない場合など）、`-extres` オプションを使用します。

-implement（完全修飾 Java インタフェース名。ojspc のデフォルト：空）

このオプションを使用して、生成されたページ実装クラスで実装される Java インタフェースを指定します。

-noCompile（ブール、ojspc のデフォルト：false）

このフラグを有効化して、ojspc に対して、生成されたページ実装クラスのソースをコンパイルしないように指示します。これにより、後から代替 Java コンパイラでコンパイルできます。

注意：

- オンデマンド変換の使用例では、OracleJSP の javaccmd 構成パラメータにより関連機能が提供され、代替 Java コンパイラを直接指定できます。A-13 ページの「[OracleJSP の構成パラメータ（非 OSE）](#)」を参照してください。
 - SQLJ の JSP ページの場合は、`-noCompile` を有効化しても SQLJ 変換は禁止されず、Java でのコンパイルのみが禁止されます。
-
-

-packageName（完全修飾パッケージ名。ojspc のデフォルト：.jsp ファイルの位置に基づく）

このオプションで Java の「ドット」構文を使用して、生成されるページ実装クラスのパッケージ名を指定します。

このオプションを設定しない場合は、パッケージ名はカレント・ディレクトリ（ojspc を実行したディレクトリ）に対する .jsp ファイルの相対位置に従って決定されます。

例として、.jsp ファイルが /myapproot/src/jspsrc ディレクトリにあるときに、ojspc を /myapproot ディレクトリから実行するとします（% は UNIX プロンプトとします）。

```
% cd /myapproot
% ojspc -packageName myroot.mypackage src/jspsrc/Foo.jsp
```

これにより、パッケージ名として myroot.mypackage が使用されます。

この例で、`-packageName` オプションを使用しない場合は、OracleJSP リリース 8.1.7 (1.1.0.0.0) ではデフォルトで `src.jspsrc` がパッケージ名として使用されます。（このような実装の詳細は、将来のリリースで変更されることがあるため注意してください。）

-S-<sqlj option> <value> (-S と後続の SQLJ オプション設定。ojspc のデフォルト: 空)

SQLJ の JSP ページの場合は、ojspc の -S オプションを使用して Oracle SQLJ のオプションを SQLJ トランスレータに渡します。

SQLJ トランスレータを直接実行する場合とは異なり、SQLJ のオプションとその値は空白で区切ります (他の ojspc のオプションとの整合性を保つため)。

次に例を示します (UNIX プロンプトから) :

```
% ojspc -S-default-customizer mypkg.MyCust -d /myapproot/mybindir MyPage.jsp
```

この場合は、Oracle SQLJ の -default-customizer オプションが起動されて代替プロファイル・カスタマイザが選択されるのみでなく、ojspc の -d オプションが設定されます。

特定の Oracle SQLJ オプションに関する注意事項は、次のとおりです。

- SQLJ の -encoding オプションは使用しないでください。かわりに、JSP ページ内の page ディレクティブで contentType パラメータを使用します。
- ojspc の -addclasspath オプションを使用する場合は、SQLJ の -classpath オプションを使用しないでください。
- ojspc の -noCompile オプションを使用する場合は、SQLJ の -compile オプションを使用しないでください。
- ojspc の -d オプションを使用する場合は、SQLJ の -d オプションを使用しないでください。
- ojspc の -srcdir オプションを使用する場合は、SQLJ の -dir オプションを使用しないでください。

Oracle SQLJ トランスレータのオプションの詳細は、『Oracle8i SQLJ 開発者ガイドおよびリファレンス』を参照してください。

-srcdir (完全修飾パス。ojspc のデフォルト: カレント・ディレクトリ)

このオプションを使用して、ojspc で生成されるソース・ファイル、つまり .sqlj ファイル (SQLJ の JSP ページの場合) と .java ファイルを格納するベース・ディレクトリの位置を指定します。

指定したパスは、単に (アプリケーション相対またはページ相対パスではなく) ファイル・システム・パスとして使用されます。

Windows NT など、ディレクトリ名に空白を使用できる環境では、ディレクトリ名を引用符で囲みます。

パッケージに応じて、指定したディレクトリのサブディレクトリが必要な場合は自動的に作成されます。詳細は、6-32 ページの「[ojspc の出力ファイル、位置および関連オプションの概要](#)」を参照してください。

デフォルトでは、カレント・ディレクトリ（ojspc を実行したときのカレント・ディレクトリ）が使用されます。

生成されたファイルが一目でわかるように、このオプションを使用して、生成されたソース・ファイルを空のディレクトリに格納することをお勧めします。

注意： オンデマンド変換の使用例では、OracleJSP の `page_repository_root` 構成パラメータにより関連機能が提供されます。A-13 ページの「[OracleJSP の構成パラメータ（非 OSE）](#)」を参照してください。

-verbose（ブール。ojspc のデフォルト: false）

このオプションを有効化し、ojspc に対して変換ステップを実行時にレポートするように指示します。

次の例は、myerror.jsp の変換に関する -verbose の出力を示しています（この例で、ojspc は myerror.jsp があるディレクトリから実行されます。% は UNIX のプロンプトとします）。

```
% ojspc -verbose myerror.jsp
Translating file: myerror.jsp
1 JSP files translated successfully.
Compiling Java file: ./myerror.java
```

-version（ブール。ojspc のデフォルト: false）

このオプションを ojspc に対して有効化すると、OracleJSP のバージョン・ナンバーが表示されてから終了します。

ojspc の出力ファイル、位置および関連オプションの概要

デフォルトの ojspc では、オンデマンド変換の使用例で OracleJSP トランスレータにより生成されるのと同じファイル・セットが生成され、カレント・ディレクトリ（ojspc を実行したときのディレクトリ）またはそのサブディレクトリに格納されます。

生成されるファイルは、次のとおりです。

- .sqlj ソース・ファイル（SQLJ の JSP ページのみ）
- .java ソース・ファイル
- ページ実装クラス用の .class ファイル
- インナー・クラス用の .class ファイル

- Java リソース・ファイル、またはオプションで SQLJ プロファイル用の .class ファイル (SQLJ の JSP ページのみ)
- オプションでページの静的テキスト用の Java リソース・ファイル

OracleJSP トランスレータにより生成されるファイルの詳細は、6-6 ページの「[生成されるファイルと位置 \(オンデマンド変換\)](#)」を参照してください。

6-26 ページの「[ojspc のオプションの説明](#)」で説明した共通に使用される一部のオプションをまとめると、ojspc の次のオプションを使用してファイルの生成方法と格納場所を指定できます。

- -appRoot を使用して、アプリケーション・ルート・ディレクトリを指定します。
- -srcdir を使用して、ソース・ファイルを指定の代替ディレクトリに格納します。
- -d を使用して、バイナリ・ファイル (.class ファイルと Java リソース・ファイル) を指定の代替ディレクトリに格納します。
- -noCompile を使用して、生成されるページ実装クラス・ソースをコンパイルしないように指定します (このため、.class ファイルは生成されません)。
- -extres を使用して、静的テキストを Java リソース・ファイルに挿入します。
- -hotload を使用して、静的テキストを Java リソース・ファイルに挿入し、ホットロードを有効化します (Oracle Servlet Engine をターゲットとするページにのみ関連があります)。
- -S-ser2class (SQLJ の -ser2class オプション、SQLJ の JSP ページ専用) を使用して、SQLJ プロファイルを .ser Java リソース・ファイルではなく .class ファイル内で生成します。

出力ファイル位置の場合、カレント・ディレクトリ (または該当する -d および -srcdir オプションで指定したディレクトリ) のディレクトリ構造は、パッケージに基づいています。パッケージは、アプリケーション・ルートに対する変換対象ファイルの相対位置により決定されます。アプリケーション・ルートは、カレント・ディレクトリまたは -appRoot オプションで指定するディレクトリです。

たとえば、ojspc を次のように実行するとします (% は UNIX プロンプトとします)。

```
% cd /abc
% ojspc def/ghi/test.jsp
```

パッケージ名は def.ghi となり、出力ファイルはディレクトリ /abc/def/ghi に格納されます。

-d および -srcdir オプションで代替出力ディレクトリを指定すると、指定したディレクトリの下に def/ghi サブディレクトリが作成されます。

この時点で、次のように ojspc を他のディレクトリから実行するとします。

```
% cd /home/mydir
% ojspc -appRoot /abc /abc/def/ghi/test.jsp
```

パッケージは同じく def.ghi で、指定したアプリケーション・ルートに対する test.jsp の相対位置に基づいています。出力ファイルは、/home/mydir/def/ghi、または -d および -srcdir オプションで指定したディレクトリの def/ghi サブディレクトリに格納されます。

注意： 様々なディレクトリ内のファイルに必要なに応じて異なるパッケージ名が与えられるように、JSP アプリケーションのディレクトリごとに、ojspc を一度ずつ実行することをお勧めします。

loadjava ツールの概要

Oracle8i には、Java ファイルからスキーマ・オブジェクトを作成し、指定したデータベース・スキーマにロードできるように、loadjava コマンドライン・ツールが用意されています。

loadjava の詳細および関連の dropjava ツール（Java ソース、クラスおよびリソースのスキーマ・オブジェクトをデータベースから削除するツール）については、『Oracle8i Java Tools リファレンス』を参照してください。

通常（JSP アプリケーションに限らず）、Java 開発者は、クライアント側で Java ソースをコンパイルしてから生成されたクラス・ファイルをロードする方法と、Java ソースをロードし、サーバー側コンパイラにより Oracle8i 内で自動的にコンパイルさせる方法のどちらかを選択できます。前者の場合は、クラス・スキーマ・オブジェクトのみが作成されます。後者の場合は、ソース・スキーマ・オブジェクトとクラス・スキーマ・オブジェクトの両方が作成されます。どちらの場合も、開発者は Java リソース・ファイルもロードして、リソース・スキーマ・オブジェクトを作成できます。

loadjava ツールのコマンドラインでは、ソース・ファイル、クラス・ファイル、リソース・ファイル、JAR ファイルおよび ZIP ファイルを指定できます。ただし、ソース・ファイルとクラス・ファイルは同時にロードできません。JAR ファイル、ZIP ファイルまたは loadjava コマンドラインで扱えるのは、ソース・ファイルまたはクラス・ファイルのうち、どちらか一方のみです。（どちらの場合も、リソース・ファイルは扱うことができます。）

JAR ファイルまたは ZIP ファイルがオープンされ、処理されて、それぞれに含まれている各ファイルから 1 つ以上のスキーマ・オブジェクトが作成されます。

OracleJSP では、loadjava の使用方法は次のとおりです。

- クライアント側変換の場合は、すでに ojspc を使用して JSP ページを変換しています。これにより、デフォルトで変換済み Java ソースもコンパイルされています。次に、loadjava を使用して、生成された .class ファイルとリソース・ファイル（たとえ

ば、リソース・ファイルを生成する `ojspc -hotload` オプションなど) をロードします。通常、これらのファイルはすべて JAR ファイルにまとめられています。

または、コンパイル済みの `.class` ファイルではなく、変換済みの `.java` ファイルをロードする方法もあります。サーバー側コンパイラでは、ロード対象となる `.java` ファイルをコンパイルできます。

- サーバー側変換の場合は、`loadjava` を使用して、未変換の `.jsp` ファイルをリソース・スキーマ・オブジェクトとしてロードします。通常、これらのファイルは JAR ファイルにまとめられています。(これらのファイルは、後でサーバー側でセッション・シェルの `publishjsp` コマンドにより変換され、公開されます。)

`loadjava` オプションの構文の詳細は、次のとおりです。大カッコ `{...}` は、構文の一部ではなく、オプション入力の前にある 2 つのオプション書式の候補を囲むために使用しています。

```
loadjava {-user | -u} user/password[@database] [options]
file.java | file.class | file.jar | file.zip | file.sqlj | resourcefile
[-debug]
[-d | -definer]
[{-e | -encoding} encoding_scheme]
[-f | -force]
[{-g | -grant} user [, user]...]
[-o | -oci8]
[ -order ]
[-noverify]
[-r | -resolve]
[{-R | -resolver} "resolver_spec"]
[{-S | -schema} schema]
[ -stdout ]
[-s | -synonym]
[-t | -thin]
[-v | -verbose]
```

特に重要なのは `-user` および `-resolve` オプションです (それぞれの短縮形は `-u` および `-r` です)。`-user` オプションでは、スキーマ名とパスワードを指定します。`-resolve` オプションでは、コマンドラインで指定したクラスがすべてロードされた後に、ロードしたクラス内の外部参照を `loadjava` でコンパイルし (該当する場合)、解決するかどうかを指定します。

`.java` ソース・ファイルをロードし、ロード時にサーバー側コンパイラでコンパイルする場合は、`-resolve` オプションを有効化する必要があります。

次の例は、クライアント側変換の使用例に関するもので、JSP ページの変換とコンパイルはすでに完了しており、`HelloWorld.class` ファイルとページ実装インナー・クラス (「HelloWorld」で始まる名前を持つクラス) 用の `.class` ファイルが生成されています。% は UNIX のプロンプトとします。

```
% loadjava -u scott/tiger -r HelloWorld*.class
```

また、次のように、各ファイルを JAR ファイルにまとめることができます。

```
% loadjava -v -u scott/tiger -r HelloWorld.jar
```

loadjava -v (-verbose) オプションでは、ロード処理の進行中に詳細なステータスが出力されます。このオプションは、多数のファイルをロードする場合やサーバー側でコンパイルする場合に特に役立ちます。

次の例もクライアント側変換の使用例に関するもの（HelloWorld.java は JSP トランスレータの出力）ですが、クライアント側でのコンパイル・ステップをスキップし、かわりにサーバー側コンパイラで処理するように（ojspc -noCompile オプションで）選択しています。

```
% loadjava -v -u scott/tiger -r HelloWorld.java
```

次の例は、サーバー側変換の使用例に関するものです。

```
% loadjava -u scott/tiger -r HelloWorld.jsp
```

sess_sh セッション・シェル・ツールの概要

Oracle8i には、データベース・インスタンスのセッション・ネームスペースへの対話型インタフェースとして、sess_sh（セッション・シェル）ツールが用意されています。sess_sh の起動時に、データベース接続引数を指定します。これにより、\$ プロンプトが表示され、コマンドを入力できます。

このセッション・シェル・ツールには多数のトップレベル・コマンドがあり、それぞれに独自のオプション・セットを指定して \$ プロンプトから実行できます。OracleJSP 開発者にとって重要なコマンドは、publishservlet および unpublishservlet コマンド（クライアント側変換を伴う配布用）、publishjsp および unpublishjsp コマンド（サーバー側変換を伴う配布用）、createcontext コマンド（OSE サブレット・コンテキストの作成用）です。

このツールの起動に使用する sess_sh の主な構文要素は、次のとおりです。

```
sess_sh -user user -password password -service serviceURL
```

- -user では、スキーマのユーザー名を指定します。
- -password では、指定したユーザー名のパスワードを指定します。
- -service では、sess_sh でセッション・ネームスペースを「オープン」するデータベースの URL を指定します。serviceURL パラメータには、次の 3 つの書式のうち 1 つを使用する必要があります。

```
sess_iiop://host:port:sid  
jdbc:oracle:type:spec  
http://host[:port]
```

次に一般的な例を示します。

```
sess_iiop://localhost:2481:orcl
jdbc:oracle:thin:@myhost:1521:orcl
http://localhost:8000
```

次の例は、`sess_sh` のコマンドラインを示しています。

```
% sess_sh -user SCOTT -password TIGER -service jdbc:oracle:thin:@myhost:1521:orcl
```

`sess_sh` を起動すると、次のコマンド・プロンプトが表示されます。

```
$
```

このセッション・シェル・ツールには、`publishservlet` や `publishjsp` のようなオブジェクト公開コマンドに加えて、UNIX シェル (C シェルなど) の UNIX ファイル・システムのような操作をセッション・ネームスペースにもたらすシェル・コマンドが用意されています。たとえば、次の `sess_sh` コマンドでは、公開されているオブジェクトと、`/alpha/beta/gamma` 公開対象コンテキスト内の公開対象コンテキストが表示されます (公開対象コンテキストはセッション・ネームスペース内のノードで、ファイル・システムのディレクトリと同様です)。

```
$ ls /alpha/beta/gamma
```

前述のように、OracleJSP 開発者にとって重要な `sess_sh` コマンドは次のとおりです。

```
$ publishjsp ...
$ unpublishjsp ...
$ publishservlet ...
$ unpublishservlet ...
$ createcontext ...
```

`publishservlet` および `unpublishservlet` コマンドの詳細は、6-59 ページの「[Oracle8i での変換済み JSP ページの公開 \(セッション・シェルの publishservlet\)](#)」を参照してください。`publishjsp` および `unpublishjsp` コマンドの詳細は、6-39 ページの「[Oracle8i での JSP ページの変換と公開 \(セッション・シェルの publishjsp\)](#)」を参照してください。

各セッション・シェル・コマンドには、操作を記述する `-describe` オプション、構文の概要を表示する `-help` オプション、およびバージョン・ナンバーを表示する `-version` オプションがあります。

注意： このドキュメントで説明するのは、`sess_sh` の構文とオプションの概要のみです。ツールについては、最も単純な起動方法と使用方法のみを取り上げています。

たとえば、このマニュアルでは説明していませんが、`$` プロンプトのかわりに `sess_sh` のコマンドラインで、各コマンドを引用符で囲んで指定できます。

また、デフォルトのセッション IOP ではなくプレーンな IOP による接続、ロールの指定、SSL サーバー認証を使用するデータベースへの接続、および SID ではなくサービス名を含む URL の使用などに使用する、トップレベル・オプションもあります。

`sess_sh` ツールの詳細は、『Oracle8i Java Tools リファレンス』を参照してください。

サーバー側変換を伴う Oracle8i への配布

この項では、サーバー側変換を伴う Oracle8i への配布ステップについて説明します。

この場合のステップは次のとおりです。

1. `loadjava` を使用して、未変換の JSP ページまたは SQLJ の JSP ページのソース・ファイルを Oracle8i にロードします。
2. セッション・シェルの `publishjsp` コマンドを使用し、ページを変換して公開します。

`publishjsp` ステップでは、変換、コンパイル、ホットロード（該当する場合）および公開が自動的に処理されます。

Oracle8i への未変換の JSP ページのロード (loadjava)

サーバー側変換を伴う配布の最初のステップでは、Oracle `loadjava` ツールを使用して、未変換の `.jsp` または `.sqljsp` ファイルを Java リソース・ファイルとして Oracle8i にロードします。

複数のファイルをロードする場合は、各ファイルをロード用の JAR ファイルにまとめることをお勧めします。

Oracle8i には、Java ファイルをサーバーにロードするための汎用ツールとして、`loadjava` ツールが用意されています。概要は、6-34 ページの「[loadjava ツールの概要](#)」を参照してください。詳細は、『Oracle8i Java Tools リファレンス』を参照してください。

次の例は、未変換のページをロードする操作を示しています。

```
% loadjava -u scott/tiger Foo.jsp
```


この例では、`Foo.jsp` が SCOTT スキーマ（パスワード TIGER）に Java リソース・オブジェクトとしてロードされます。`loadjava -resolve (-r)` オプションを指定する必要はありません。

これにより、データベース内で次のリソース・スキーマ・オブジェクトが作成されます。

■ SCOTT:Foo.jsp

リソース・スキーマ・オブジェクトの位置は、JAR ファイル内または `loadjava` のコマンドラインで `.jsp` ファイル用に指定したパス情報により決定されるため注意してください。前述の例を次のように変更するとします。

```
% loadjava -u scott/tiger xxx/yyy/Foo.jsp
```

これにより、データベース内で次のリソース・スキーマ・オブジェクトが作成されます。

■ SCOTT:xxx/yyy/Foo.jsp

`loadjava` により生成されるスキーマ・オブジェクトの命名方法の概要は、6-12 ページの「[Java 用のデータベース・スキーマ・オブジェクト](#)」を参照してください。

次のように、`.sqljsp` ファイルもロードできます。

```
% loadjava -u scott/tiger Foo.sqljsp
```

この例では、`Foo.sqljsp` が SCOTT スキーマにロードされ、データベース内で次のリソース・スキーマ・オブジェクトが作成されます。

■ SCOTT:Foo.sqljsp

複数の `.jsp`（または `.sqljsp`）ファイルをロードする場合は、次のようにワイルドカード文字を使用できます（運用環境に依存します。`%` は UNIX プロンプトとします）。

```
% loadjava -u scott/tiger *.jsp
```

または、次のように `.jsp` ファイルを JAR ファイルにまとめるとします。

```
% loadjava -u scott/tiger myjspapp.jar
```

Oracle8i での JSP ページの変換と公開（セッション・シェルの `publishjsp`）

サーバー側変換を伴う配布の使用例では、Oracle8i JVM のセッション・シェルの `publishjsp` コマンドを実行すると、変換、コンパイル、ホットロード（該当する場合）および公開がすべて自動的に実行されます。セッション・シェルを起動してデータベースに接続する方法は、6-36 ページの「[sess_sh セッション・シェル・ツールの概要](#)」を参照してください。

`.jsp`（または `.sqljsp`）ファイルを Oracle8i にリソース・スキーマ・オブジェクトとしてロードしてから、`publishjsp` を実行します。（`.sqljsp` ファイルに対して `publishjsp`

を実行する操作については、結果にロジスティック上の違いがあるため、別途に説明します。)

注意： publishjsp で公開される JSP ページは、セッション・シェルの unpublishjsp コマンドを使用して「非公開」に（Oracle8i JVM の JNDI ネームスペースから削除）できます。6-50 ページの「[unpublishjsp を使用した JSP ページの非公開](#)」を参照してください。

publishjsp の構文とオプションの概要

sess_sh を起動すると、データベースへの接続が確立されます。sess_sh の起動後は、セッション・シェルの \$ プロンプトから publishjsp コマンドを実行できます。

publishjsp コマンドの一般構文は、次のとおりです。

```
$ publishjsp [options] path/name.jsp
```

次のいずれかのオプションを使用できます。

```
[-schema schemaname] [-virtualpath path] [-servletName name] [-packageName name]
[-context context] [-hotload] [-stateless] [-verbose] [-resolver resolver]
[-extend class] [-implement interface]
```

重要：

- -hotload などのプール・オプションを有効化するには、コマンドラインにオプション名のみを入力します（true に設定しないでください）。
 - 値を指定するオプションの場合、値を引用符で囲む必要はありません。
-

ファイル name.jsp（または、SQLJ の JSP ページの場合は name.sqljsp）は、loadjava でロードした JSP ページのリソース・スキーマ・オブジェクトで、関連スキーマの path 情報とともに指定する唯一の必須パラメータです。

デフォルトでは、-virtualpath オプションを指定しない場合は、path/name.jsp がサブレット・パスとなります。たとえば、dir1/foo.jsp（カレント・スキーマまたは指定したスキーマ内のパス）に対して publishjsp を実行すると、dir1/foo.jsp がサブレット・パスとなります。

デフォルトでは、-context オプションを指定しない場合は、OSE のデフォルトのサブレット・コンテキストが使用され、「/」がコンテキスト・パスとなります。

コンテキスト・パスとサーブレット・パス（およびホスト名とポート）により、ページを起動するための URL が決定されます。6-14 ページの「[Oracle Servlet Engine の URL](#)」を参照してください。

また、次の情報表示オプションも使用できます。

- `-showVersion` を単独で使用すると、OracleJSP のバージョン・ナンバーが表示されて終了します。
- `-usage` を単独で使用すると、publishjsp オプション・リストが表示されて終了します。

ここでは、各オプションについて説明します。

- `-schema schemaname`

このオプションを使用して、JSP ページのリソース・スキーマ・オブジェクトが、`sess_sh` を介してログインしたスキーマとは異なるスキーマにある場合に、そのスキーマを指定します。

このスキーマには、`sess_sh` のログイン・スキーマからアクセスできる必要があります。publishjsp コマンドでは、パスワードは指定できません。

- `-virtualpath path`

このオプションを使用すると、JSP ページの代替サーブレット・パスを指定できます。このオプションを使用しない場合は、サーブレット・パスは単に指定の .jsp ファイル名と指定のスキーマ・パスのみで構成されます。

次に例を示します。

```
-virtualpath altpath/Foo.jsp
```

または、単に次のように入力します。

```
-virtualpath mypath.jsp
```

- `-servletName name`

このオプションを使用すると、JSP ページの代替サーブレット名（OSE の `named_servlets`）を指定できます。ただし、サーブレット名は、ページの起動方法には関与しないため、ほとんどの場合は不要です。

デフォルトでは、サーブレット名は .jsp ファイル名と指定したパスで構成されます。たとえば、`SCOTT:dir1/Foo.jsp` に対して publishjsp を実行すると、OracleJSP リリース 8.1.7 (1.1.0.0.0) では、サーブレット名として `dir1.Foo` が使用されます（実装の詳細は将来のリリースで変更されることがあるため注意してください）。

- `-packageName name`

このオプションを使用して、生成されるページ実装クラスのパッケージ名を指定できます。このオプションを使用しない場合は、パッケージ名は publishjsp の実行時に

.jsp ファイルのパス指定により決定されます。たとえば、SCOTT:dir1/Foo.jsp に対して publishjsp を実行すると、ページ実装クラスのパッケージ名は dir1 となります。

-packageName オプションは、スキーマにあるスキーマ・オブジェクトの位置に影響しますが、JSP ページのサーブレット・パスには影響しません。

■ -context context

このオプションを使用すると、Oracle Servlet Engine 内のサーブレット・コンテキストを指定できます。このサーブレット・コンテキストのコンテキスト・パスは、ページの起動に使用される URL の一部となります。

このオプションを使用しない場合は、JSP ページは OSE のデフォルト・コンテキスト /webdomains/contexts/default に置かれます。コンテキスト・パスは単に「/」です。

コンテキストは、次のように /webdomains/contexts の下で指定する必要があります。

/webdomains/contexts/mycontext

重要： これは、コンテキスト名そのものではなく、ページへのアクセス用 URL に使用されるサーブレット・コンテキストのコンテキスト・パスであることに注意してください。

OSE 内でセッション・シェルの createcontext コマンドを使用してサーブレット・コンテキストを作成する場合は、コンテキスト・パス (createcontext -virtualpath オプションを使用) とコンテキスト名の両方を指定する必要があります。コンテキスト名とコンテキスト・パスを同じにすると便利のため、通常はこのように指定しますが、必須ではありません。

■ -hotload

このフラグを有効化し、ホットロードを有効化して実行します。その結果、publishjsp コマンドにより次のステップが実行されます。

1. 静的出力が、ページ実装クラスのスキーマ・オブジェクトではなく、リソース・スキーマ・オブジェクトに書き込まれます。
2. ホットロード可能になるように、生成されたページ実装クラスに main() メソッドとホットロード・メソッドが実装されます。
3. main() メソッドが実行され、ホットロードが実行されます。

-hotload を使用するには、Oracle8i JVM のホットローダーに対するパーミッションが必要です。このパーミッションは、次のように（たとえば、SCOTT スキーマの場合は SQL*Plus から）付与できます

```
dbms_java.grant_permission('SCOTT', 'SYS:oracle.aurora.security.JServerPermission', 'HotLoader', null);
```

ホットロードの概要は、6-20 ページの「[Oracle8i にホットロードされるクラスの概要](#)」を参照してください。

- -stateless

これは、Oracle Servlet Engine に対して、JSP ページをステートレスにするように指示するプール・オプションです。JSP ページには、実行中に HttpSession オブジェクトへのアクセス権を付与しないようにします。

このフラグは、mod_ose の最適化に使用します。Apache の mod_ose モジュールの詳細は、『Oracle8i Oracle Servlet Engine ユーザーズ・ガイド』を参照してください。

- -verbose

このオプションを true に設定し、publishjsp に対して実行時に変換ステップをレポートするように指示します。

- -resolver

このオプションを使用して、代替 Java クラス・リゾルバを指定します。リゾルバは、JSP ページに使用されるクラスの位置の特定など、loadjava を介して Java ソースをコンパイルし、解決するときに使用されます。

デフォルトのリゾルバは ((* user) (* PUBLIC)) です。たとえば、SCOTT スキーマの場合は次のようになります。

```
(( * SCOTT) ( * PUBLIC))
```

-resolver オプションの場合は、次のように値を引用符で囲んで指定する必要があります。

```
$ publishjsp ... -resolver "(( * BILL) ( * SCOTT) ( * PUBLIC))" ...
```

- -extend

このオプションを使用して、生成されたページ実装クラスで拡張される Java クラスを指定します。

- -implement

このオプションを使用して、生成されたページ実装クラスで実装される Java インタフェースを指定します。

例 : publishjsp を使用した JSP ページの公開

この項では、publishjsp を使用して Oracle8i 内で .jsp ページを変換し、公開する例について説明します。各ページは、すでにリソース・スキーマ・オブジェクトとして、SCOTT:Foo.jsp のような特定のスキーマにロードされています。

(.sqljsp ページに対して publishjsp を実行する方法の詳細は、6-48 ページの「[publishjsp を使用した SQLJ の JSP ページの公開](#)」を参照してください。)

サーブレット・パスとコンテキスト・パスの組合せによるページ起動用 URL の生成については、6-14 ページの「[Oracle Servlet Engine の URL](#)」を参照してください。

注意：

- 次の例では、SCOTT スキーマを使用しています。SCOTT は、sess_sh の起動時に指定したスキーマ、またはそのスキーマからアクセスできるスキーマである必要があります。
 - それぞれの例で、作成されるスキーマ・オブジェクトを示していますが、これは二次的な情報です。JSP ページの起動には、サーブレット・パスとコンテキスト・パスのみが使用されます。ページ実装クラスのスキーマ・オブジェクトは、publishjsp による公開ステップで自動的にマップされます。
 - Oracle8i 内では、どの JSP 環境の場合も、動的な jsp:include および jsp:forward 文のアプリケーション相対構文とページ相対構文は同じです。相対パスは、JSP ページの公開方法（後述の例を参照）に基づいています。
 - 生成されるスキーマ・オブジェクトの正確な名前は、将来のリリースで変更されることがありますが、一般的な書式は同じです。この名前には常にベース名（後述の例の「Foo」など）が含まれますが、Foo のかわりに _Foo のような変形が含まれる場合もあります。
 - \$ は sess_sh のプロンプトです。
-

例 1

```
$ publishjsp -schema SCOTT dir1/Foo.jsp
```

この例では、コンテキスト・パス「/」を持つデフォルトのサーブレット・コンテキストを使用しています。

デフォルトのサーブレット・パスは dir1/Foo.jsp です。

このコマンドの後で、次のように Foo.jsp を起動できます。

```
http://host[:port]/dir1/Foo.jsp
```

このファイルには、次のように（ページ相対構文、次にアプリケーション相対構文を使用して）アプリケーション内の別の JSP ページから動的にアクセスします。この JSP ページは、`dir1/Bar.jsp` として公開されているとします。

```
<jsp:include page="Foo.jsp" flush="true" />
```

または

```
<jsp:include page="/dir1/Foo.jsp" flush="true" />
```

デフォルトでは、`dir1` はページ実装クラスとインナー・クラスの Java パッケージです（SCOTT スキーマ内のパスを指定しているため）。

次のスキーマ・オブジェクトが作成されます。

- SCOTT:dir1/Foo ソース・スキーマ・オブジェクト
- SCOTT:dir1/Foo クラス・スキーマ・オブジェクト
- 静的テキストのインナー・クラス用の `dir1` の下のクラス・スキーマ・オブジェクト（SCOTT:dir1/Foo\$__jsp__StaticText のように、名前には「Foo」が含まれています）

例 2

```
$ publishjsp -schema SCOTT -context /webdomains/contexts/mycontext Foo.jsp
```

`mycontext` が次のように作成されているとします。

```
$ createcontext -virtualpath mycontext /webdomains mycontext
```

`publishjsp` コマンドにより、ページが `mycontext` サブレット・コンテキストに公開されます。このサブレット・コンテキストの作成時には、コンテキスト・パスとして `mycontext` も指定されています。

デフォルトのサブレット・パスは単に `Foo.jsp` です。

このコマンドの後で、次のように `Foo.jsp` を起動できます。

```
http://host[:port]/mycontext/Foo.jsp
```

このファイルには、次のように（ページ相対構文、次にアプリケーション相対構文を使用して）アプリケーション内の別の JSP ページから動的にアクセスします。この JSP ページは、`Bar.jsp` として公開されているとします。

```
<jsp:include page="Foo.jsp" flush="true" />
```

または

```
<jsp:include page="/Foo.jsp" flush="true" />
```

この例でデフォルト以外のサーブレット・コンテキストを指定しても、動的な `jsp:include` または `jsp:forward` コマンドには反映されません。反映されるのは、そのコンテキストに対するページの公開済み相対パスが単に `/Foo.jsp` であるということです。

デフォルトでは、ページ実装クラスとインナー・クラスの Java パッケージはありません (SCOTT スキーマ内でパスを指定していないため)。

次のスキーマ・オブジェクトが作成されます。

- SCOTT:Foo ソース・スキーマ・オブジェクト
- SCOTT:Foo クラス・スキーマ・オブジェクト
- 静的テキストのインナー・クラス用のクラス・スキーマ・オブジェクト
(SCOTT:Foo\$__jsp__StaticText のように、名前には「Foo」が含まれています)

例 3

```
$ publishjsp -schema SCOTT -context /webdomains/contexts/mycontext dir1/Foo.jsp
```

`mycontext` が次のように作成されているとします。

```
$ createcontext -virtualpath mywebapp /webdomains mycontext
```

`publishjsp` コマンドにより、ページが `mycontext` サーブレット・コンテキストに公開されます。このサーブレット・コンテキストの作成時には、コンテキスト・パスとして `mywebapp` が指定されています。

デフォルトのサーブレット・パスは `dir1/Foo.jsp` です。

このコマンドの後で、次のように `Foo.jsp` を起動できます。

```
http://host[:port]/mywebapp/dir1/Foo.jsp
```

このファイルには、次のように (ページ相対構文、次にアプリケーション相対構文を使用して) アプリケーション内の別の JSP ページから動的にアクセスします。この JSP ページは、`dir1/Bar.jsp` として公開されているとします。

```
<jsp:include page="Foo.jsp" flush="true" />
```

または

```
<jsp:include page="/dir1/Foo.jsp" flush="true" />
```

例 1 と **例 3** では異なるサーブレット・コンテキストを使用していますが、どちらの場合も、アプリケーション相対の `include` コマンドに関連があるのは、そのコンテキストに対するページの公開済み相対パスが `/dir1/Foo.jsp` であることです。

デフォルトでは、`dir1` はページ実装クラスとインナー・クラスの Java パッケージです。

次のスキーマ・オブジェクトが作成されます。

- SCOTT:dir1/Foo ソース・スキーマ・オブジェクト
- SCOTT:dir1/Foo クラス・スキーマ・オブジェクト
- 静的テキストのインナー・クラス用のクラス・スキーマ・オブジェクト
(SCOTT:dir1/Foo\$__jsp__StaticText のように、名前には「Foo」が含まれています)

例 4

```
$ publishjsp -schema SCOTT -hotload -packageName mypkg dir1/Foo.jsp
```

この例では、ホットロードを実行し、デフォルトのサーブレット・コンテキストを使用し、デフォルトの dir1 パッケージをオーバーライドしています。

コンテキスト・パスは「/」です。

-packageName オプションは、サーブレット・パスには反映されず、デフォルトで dir1/Foo.jsp のままです。

このコマンドの後で、次のように Foo.jsp を起動できます。

```
http://host[:port]/dir1/Foo.jsp
```

このファイルには、次のように（ページ相対構文、次にアプリケーション相対構文を使用し、アプリケーション内の別の JSP ページから動的にアクセスします。この JSP ページは、dir1/Bar.jsp として公開されているとします。

```
<jsp:include page="Foo.jsp" flush="true" />
```

または

```
<jsp:include page="/dir1/Foo.jsp" flush="true" />
```

次のスキーマ・オブジェクトが作成されます。

- SCOTT:mypkg/Foo ソース・スキーマ・オブジェクト
- SCOTT:mypkg/Foo クラス・スキーマ・オブジェクト
- インナー・クラス用の mypkg の下のクラス・スキーマ・オブジェクト
(SCOTT:mypkg/Foo\$__jsp__StaticText のように、名前には「Foo」が含まれています)
- 通常はインナー・クラスにある静的テキスト用の SCOTT:mypkg/Foo.res リソース・スキーマ・オブジェクト（リソースは publishjsp 機能の一部としてホットロードされます)

publishjsp を使用した SQLJ の JSP ページの公開

この項では、publishjsp を使用して Oracle8i 内で .sqljsp ページを変換し、公開する例について説明します。ページは、すでにリソース・スキーマ・オブジェクトとして、SCOTT:Foo.sqljsp のような特定のスキーマにロードされています。

その他の想定については、6-44 ページの「例: publishjsp を使用した JSP ページの公開」を参照してください。

サブレット・パスとコンテキスト・パスの組合せによるページ起動用 URL の生成については、6-14 ページの「Oracle Servlet Engine の URL」を参照してください。

.sqljsp ページに関する注意事項は、次のとおりです。

- .jsp ページの場合に作成されるスキーマ・オブジェクトに加えて、SQLJ プロファイル用のリソース・スキーマ・オブジェクトが作成されます。サーバー側での変換時には SQLJ の -ser2class オプションがないため、これはクラス・スキーマ・オブジェクトではなく、常に .ser リソース・スキーマ・オブジェクトです。

SQLJ プロファイルの詳細は、『Oracle8i SQLJ 開発者ガイドおよびリファレンス』を参照してください。

- 生成されるソース・スキーマ・オブジェクトは、Java ソースではなく SQLJ ソースです。
- SQLJ の場合、サーバー側でサポートされるオプションがきわめて限られています。

サーバー側の SQLJ オプション クライアント側の SQLJ オプションは、サーバー側での変換には使用できません（これは、JSP ページのみでなくすべてに該当します）。かわりに、dbms_java.set_compiler_option() ストアド・プロシージャで（SQL*Plus などを使用して）設定された標準的な Oracle8i JAVA\$OPTIONS 表を介して使用できるように、少数のオプション・セットが用意されています。その中で、JSP ページ用にサポートされているのは次のオプションのみです。

- online

これは、デフォルトの oracle.sqlj.checker.OracleChecker フロントエンドを介してオンライン意味検査を有効化するブール・オプションです。

サーバー側の SQLJ とセマンティクス・チェックの詳細は、『Oracle8i SQLJ 開発者ガイドおよびリファレンス』を参照してください。

SQLJ の JSP ページの公開例 次の例は、.sqljsp ページに対する publishjsp の使用例を示しています（\$ は sess_sh のプロンプトです）。

注意：

- この例では、SCOTT スキーマを使用しています。SCOTT は、sess_sh の起動時に指定したスキーマ、またはそのスキーマからアクセスできるスキーマである必要があります。
 - この例では、作成されるスキーマ・オブジェクトを示していますが、これは二次的な情報です。JSP ページの起動には、サーブレット・パスとコンテキスト・パスのみが使用されます。ページ実装クラスのスキーマ・オブジェクトは、publishjsp による公開ステップで自動的にマップされます。
 - 生成されるスキーマ・オブジェクトの正確な名前は、将来のリリースで変更されることがありますが、一般的な書式は同じです。この名前には常にベース名（後述の例の「Foo」など）が含まれますが、Foo のかわりに _Foo のような変形が含まれる場合もあります。
-
-

```
$ publishjsp -schema SCOTT dir1/Foo.sqljsp
```

この例では、デフォルトの OSE サーブレット・コンテキストとコンテキスト・パス「/」を使用しています。

デフォルトのサーブレット・パスは、dir1/Foo.sqljsp です。

このコマンドの後で、次のように Foo.sqljsp を起動できます。

```
http://host[:port]/dir1/Foo.sqljsp
```

このファイルには、次のように（ページ相対構文、次にアプリケーション相対構文を使用して）アプリケーション内の別の JSP ページから動的にアクセスします。この JSP ページは、dir1/Bar.jsp として公開されているとします。

```
<jsp:include page="Foo.sqljsp" flush="true" />
```

または

```
<jsp:include page="/dir1/Foo.sqljsp" flush="true" />
```

デフォルトでは、dir1 はページ実装クラスとインナー・クラスの Java パッケージです（SCOTT スキーマ内のパスを指定しているため）。

次のスキーマ・オブジェクトが作成されます。

- SCOTT:dir1/Foo ソース・スキーマ・オブジェクト
- SCOTT:dir1/Foo クラス・スキーマ・オブジェクト

- 静的テキストのインナー・クラス用の `dir1` の下のクラス・スキーマ・オブジェクト (SCOTT:dir1/Foo\$__jsp__StaticText のように、名前には「Foo」が含まれています)
- SQLJ プロファイル用の `dir1` の下のリソース・スキーマ・オブジェクト (SCOTT:dir1/Foo_SJProfile0.ser のように、名前には「Foo」が含まれています)

unpublishjsp を使用した JSP ページの非公開

`sess_sh` ツールには、JSP ページを Oracle8i JVM の JNDI ネームスペースから削除する `unpublishjsp` コマンドも用意されています。ただし、このコマンドでは、ページ実装クラスのスキーマ・オブジェクトはデータベースから削除されません。

`unpublishservlet` コマンドとは異なり、サーブレット名を指定する必要はありません (`publishjsp` の実行時に指定していない場合)。通常、必須入力はサーブレット・パスのみです (「仮想パス」と呼ばれることもあります)。

一般構文は次のとおりです。

```
$ unpublishjsp [-servletName name] [-context context] [-showVersion] [-usage] [-verbose] servletpath
```

`-servletName`、`-context`、`-showVersion`、`-usage` および `-verbose` オプションは、`publishjsp` の場合と同じです。6-40 ページの「[publishjsp の構文とオプションの概要](#)」を参照してください。

`unpublishjsp` を使用する場合は、`publishjsp` の使用時に指定した `-servletName` および `-context` の値を指定します。

たとえば、6-47 ページの例 4 で公開したページを非公開にするコマンドは次のとおりです。

```
$ unpublishjsp dir1/Foo.jsp
```

(例 4 で指定した `-packageName` オプションは、サーブレット・パスには無効であることに注意してください。)

クライアント側変換を伴う Oracle8i への配布

この項では、クライアント側変換を伴う Oracle8i への配布ステップについて説明します。

この場合のステップは次のとおりです。

1. `ojspc` を使用して、クライアント側で JSP ページまたは SQLJ の JSP ページを事前に変換します。
2. `loadjava` を使用して、ページ変換により生成されたファイルを Oracle8i にロードします。ページ変換の結果、`.class` ファイル (または、オプションで `.java` またはかわりに `.sqlj` ファイル) および Java リソース・ファイルが生成されています。

3. (オプション) ページを Oracle8i に「ホットロード」します (変換時にホットロードを有効化した場合)。ホットロードの背景情報については、6-20 ページの「[Oracle8i にホットロードされるクラスの概要](#)」を参照してください。
4. セッション・シェルの `publishservlet` コマンドを使用してページを公開します。

注意： 操作を簡素化し容易にするために、通常はサーバー側変換を伴う配布をお勧めします。6-38 ページの「[サーバー側変換を伴う Oracle8i への配布](#)」を参照してください。

JSP ページの事前変換 (ojspc)

JSP ページをクライアント側で事前変換するためには (通常は Oracle Servlet Engine 内で実行するページの場合)、`ojspc` コマンドライン・ツールを使用して OracleJSP トランスレータを起動します。

`ojspc` の概要とオプションの説明は、6-22 ページの「[ojspc 事前変換ツール](#)」を参照してください。

以降は、次のトピックについて説明します。

- [ojspc の最も単純な使用方法](#)
- [SQLJ の JSP ページ用の ojspc](#)
- [ojspc を使用したホットロードの有効化](#)
- [Oracle8i への配布に使用する ojspc のその他の主要機能とオプション](#)
- [ojspc の例](#)

注意： 生成されるファイルの正確な名前は、将来のリリースで変更されることがありますが、一般的なフォームは同じです。ファイル名には常にベース名 (前述の例では「Foo」など) が含まれますが、`_Foo.java` や `_Foo.class` のように少し変形したクラス名が含まれることがあります。

ojspc の最も単純な使用方法

次の例は、`ojspc` の最も単純な使用方法を示しています。

```
% ojspc Foo.jsp
```

この起動方法では、次のファイルが生成されます。

- `Foo.java`
- `Foo.class`

- 静的コンテンツのインナー・クラス用の .class ファイル（名前に「Foo」が含まれます）

デフォルトでは、すべての出力は ojspc を起動したときのカレント・ディレクトリに書き込まれます。

SQLJ の JSP ページ用の ojspc

ojspc ツールでは、次のように、SQLJ コードを使用する JSP ページ用の .sqljsp ファイルも指定できます。

```
% ojspc Foo.sqljsp
```

.sqljsp ファイルの場合は、ojspc により JSP トランスレータのみでなく SQLJ トランスレータも自動的に起動されます。

この起動方法では、次のファイルが生成されます。

- Foo.sqlj (JSP トランスレータにより Foo.sqljsp から生成)
- Foo.java (SQLJ トランスレータにより Foo.sqlj から生成)
- Foo.class
- 静的コンテンツのインナー・クラス用の .class ファイル（名前に「Foo」が含まれます）
- SQLJ の「プロファイル」に関する SQLJ の -ser2class オプションの設定に応じて、Java リソース・ファイル (.ser) またはクラス・ファイル (.class)（名前に「Foo」が含まれます）

SQLJ プロファイルの詳細は、『Oracle8i SQLJ 開発者ガイドおよびリファレンス』を参照してください。

デフォルトでは、すべての出力は ojspc を起動したときのカレント・ディレクトリに書き込まれます。

ojspc を使用したホットロードの有効化

ojspc -hotload オプションを使用してホットロードを有効化します。これにより（他の処理に加えて）、静的ページ・コンテンツは、ページ実装クラスのインナー・クラスではなく Java リソース・ファイルに書き込まれます。

次の例では、ページを変換し、OracleJSP トランスレータに対してホットロードを有効化するように指示しています。

```
% ojspc -hotload Foo.jsp
```

このコマンドでは、トランスレータにより次の出力が生成されます。

- Foo.java (通常どおり)
- Foo.class (通常どおり)
- Foo.res、静的ページ・コンテンツを含む Java リソース・ファイル
- インナー・クラス用の .class ファイル (通常どおり。名前に「Foo」が含まれていますが、静的コンテンツはインナー・クラスではなく Foo.res に書き込まれます)

ojspc -hotload オプションで有効化されるのは、ホットロードのみであることに注意してください。実際にページがホットロードされるわけではありません。ホットロード機能を使用するには、追加の配布ステップが必要です。6-58 ページの「[Oracle8i でのページ実装クラスのホットロード](#)」を参照してください。

ホットロードの概要は、6-20 ページの「[Oracle8i にホットロードされるクラスの概要](#)」を参照してください。

Oracle8i への配布に使用する ojspc のその他の主要機能とオプション

6-26 ページの「[ojspc のオプションの説明](#)」で詳しく説明したように、ojspc の次のオプションは特に役立ちます。

- -appRoot — デフォルト (ojspc を実行したときのカレント・ディレクトリ) を使用しない場合に、アプリケーション・ルート・ディレクトリを設定します。
- -noCompile — 変換時にコンパイルしない場合は、このフラグを有効化します。たとえば、このフラグを有効化するのは、変換済みページを .java ファイルとして Oracle8i にロードし、コンパイルはサーバー側コンパイラで実行している場合です。
- -d — ojspc により生成されたバイナリ・ファイル (.class ファイルと Java リソース・ファイル) が格納されるディレクトリを指定します。これにより、生成されたファイルと Oracle8i にロードする必要のあるファイルの確認が容易になります。
- -srcdir — ojspc により生成されたソース・ファイルが格納されるディレクトリを指定します。たとえば、-noCompile を有効化し、変換済みページを Oracle8i に .java ソースとしてロードする場合は、このオプションを -d のかわりに使用すると役立ちます。
- -extres — OracleJSP トランスレータに対して、静的コンテンツをページ実装クラスのインナー・クラスではなく Java リソース・ファイルに書き込むように指示します。
- -hotload — OracleJSP トランスレータに対して、静的コンテンツをページ実装クラスのインナー・クラスではなく Java リソース・ファイルに書き込み、ページ実装クラス内でコードを生成し、ホットロードを有効化するように指示します。
- -s — SQLJ の JSP ページの場合は、-s 接頭辞を使用して Oracle SQLJ のオプションを指定します。これらのオプション設定は、ojspc から Oracle SQLJ トランスレータに渡されます。

ojspc の例

次の例は、ojspc の主要オプションの使用方を示しています。

```
% ojspc -appRoot /myroot/pagesrc -d /myroot/bin -hotload /myroot/pagesrc/Foo.jsp
```

この例では、次の処理が実行されます。

- 変換済みページ内のアプリケーション相対の静的 include ディレクティブについて、アプリケーション・ルートが指定されます。
- ホットロードが有効化され、静的コンテンツ用の Java リソース・ファイル `Foo.res` が生成されます。
- `Foo.java` がデフォルトでカレント・ディレクトリに格納されます。`Foo.jsp` は指定したアプリケーション・ルート・ディレクトリにあるため、パッケージはありません。
- インナー・クラス用の `Foo.class`、`Foo.res` および `.class` ファイルが、`/myroot/bin` ディレクトリに格納されます。

```
% ojspc -appRoot /myroot/pagesrc -srcdir /myroot/gensrc -noCompile -extres /myroot/pagesrc/Foo.jsp
```

この例では、次の処理が実行されます。

- 変換済みページ内のアプリケーション相対の静的 include ディレクティブについて、アプリケーション・ルートが指定されます。
- 静的コンテンツ用の Java リソース・ファイル `Foo.res` が生成されます（ホットロードは有効化されません）。
- `Foo.java` が `/myroot/gensrc` ディレクトリに格納されます。`Foo.jsp` は指定したアプリケーション・ルート・ディレクトリにあるため、パッケージはありません。
- `Foo.java` はコンパイルされません（`.class` ファイルは生成されません）。
- `Foo.res` がデフォルトでカレント・ディレクトリに格納されます。

```
% ojspc -appRoot /myroot/pagesrc -d /myroot/bin -extres -S-ser2class true /myroot/pagesrc/Foo.sqljsp
```

この例では、次の処理が実行されます。

- 変換済みページ内のアプリケーション相対の静的 include ディレクティブについて、アプリケーション・ルートが指定されます。
- 静的コンテンツ用の Java リソース・ファイル `Foo.res` が生成されます（ホットロードは有効化されません）。
- `Foo.sqlj` および `Foo.java` がデフォルトでカレント・ディレクトリに格納されます。`Foo.jsp` は指定したアプリケーション・ルート・ディレクトリにあるため、パッケージはありません。
- インナー・クラス用の `Foo.class`、`Foo.res` および `.class` ファイルと、SQLJ プロファイル用の `.class` ファイルが、`/myroot/bin` ディレクトリに格納されます。

(SQLJ の `-ser2class` オプションが設定されていないため、プロファイルは `.class` ファイルではなく `.ser` Java リソース・ファイル内で生成されます。)

Oracle8i への変換済み JSP ページのロード (loadjava)

クライアント側での事前変換後に、Oracle の `loadjava` ツールを使用して、生成されたファイルを Oracle8i にロードします。次の 2 つの使用例のどちらかを使用できます。

- `.class` ファイルと Java リソース・ファイル（存在する場合）をロードします。
- 変換時に `ojspc -noCompile` オプションを使用してから、変換済みの `.java` ファイルとリソース・ファイル（存在する場合）をロードします。`.java` ファイルは、ロード時に Oracle8i のサーバー側コンパイラでコンパイルできます。

どちらの場合も、複数のファイルをロードするときには、各ファイルをロード用の JAR ファイルにまとめることをお勧めします。

Oracle8i には、Java ファイルをデータベースにロードするための汎用ツールとして、`loadjava` ツールが用意されています。概要は、6-34 ページの「[loadjava ツールの概要](#)」を参照してください。また、『Oracle8i Java Tools リファレンス』を参照してください。

重要： 次の 2 つの項（「[loadjava を使用したクラス・ファイルのロード](#)」および「[loadjava を使用したソース・ファイルのロード](#)」）では、次の重要な考慮事項に注意してください。

- 静的テキストをリソース・ファイルに格納するために `-extres` または `-hotload` オプションを有効化した場合も、ページ実装インナー・クラスは生成されるため、そのクラスをロードする必要があります。
 - Java コンパイラと同様に、`loadjava` ではリソースの参照ではなくクラスの参照が解決されます。リソース・ファイルを必要なクラスに適切にロードしてください。リソース・ファイルは、`.java` ファイルと同じパッケージに格納する必要があります。
-

loadjava を使用したクラス・ファイルのロード

`ojspc -extres` または `-hotload` オプションを有効化し、JSP ページ `Foo.jsp` を変換すると、次のファイルが生成されます。

- `Foo.java`
- `Foo.class`
- `Foo$__jsp__StaticText.class`
- `Foo.res`

注意： この例で生成される名前として使用しているのは、あくまでも一例です。このような実装の詳細は将来のリリースで変更されることがありますが、生成される名前の一部として常にベース名（この例の「Foo」など）が使用されます。

Foo.java は無視してもかまいませんが、バイナリ・ファイル（.class および .res）はすべて Oracle8i にロードする必要があります。通常は、Foo.class、Foo\$__jsp__StaticText.class および Foo.res を、Foo.jar のような 1 つの JAR ファイルにまとめて、次のようにロードします（% は UNIX プロンプトとします）。

```
% loadjava -v -u scott/tiger -r Foo.jar
```

-u (-user) オプションでは、データベース・スキーマのユーザー名とパスワードを指定します。-r (-resolve) オプションにより、ロードされるクラスが解決されます。オプションで、ステータス詳細が出力されるように -v (-verbose) オプションを使用します。

また、次のようにファイルを個別にロードすることもできます。（構文は運用環境に応じて異なります。各例では、% は UNIX プロンプトとします。）

```
% loadjava -v -u scott/tiger -r Foo*.class Foo.res
```

または

```
% loadjava -v -u scott/tiger -r Foo*.*
```

前述の例では、いずれもデータベース内に次のスキーマ・オブジェクトが作成されます（通常、知る必要があるのは、ページ実装クラスのスキーマ・オブジェクト名のみです）。

- SCOTT:Foo ページ実装クラス・スキーマ・オブジェクト

また、ojspc -packageName オプションや、ojspc を実行するときのカレント・ディレクトリに対する .jsp ファイルの相対位置によっては、パッケージ情報を追加指定できます。たとえば、-packageName に "abc.def" を設定すると、Foo クラスのパッケージになるため、SCOTT:abc/def/Foo クラス・スキーマ・オブジェクトが作成されます。

- SCOTT:Foo\$__jsp__StaticText クラス・スキーマ・オブジェクト

パッケージ指定はページ実装クラスと同じです。

- SCOTT:Foo.res リソース・スキーマ・オブジェクト

ロード時には、JAR ファイル内または loadjava コマンドラインでのパス指定に基づいて、パッケージ指定が使用されます。

loadjava により生成されるスキーマ・オブジェクトの命名方法の概要は、6-12 ページの「[Java 用のデータベース・スキーマ・オブジェクト](#)」を参照してください。

注意： 事前に変換された SQLJ の JSP ページをロードする場合は、生成されたプロファイル・ファイルもロードする必要があります。このプロファイル・ファイルは、SQLJ の `-ser2class` オプションに応じて `.ser` Java リソースファイルまたは `.class` ファイルです。スキーマ・オブジェクトの命名は、`.ser` ファイルの場合は、`.res` Java リソース・ファイルに使用されるものと同じです。`.class` ファイルの場合は、他の `.class` ファイルに使用されるものと同じです。

loadjava を使用したソース・ファイルのロード

`ojspc -noCompile` および `-extres` オプションを有効化し、JSP ページ `Foo.jsp` を変換すると、次のファイルが生成されます。

- `Foo.java` (サーバー側コンパイラでコンパイルするソースとして Oracle8i にロードするファイル)
- `Foo.res`

通常は、`Foo.java` および `Foo.res` を `Foo.jar` のような 1 つの JAR ファイルにまとめて、次のようにロードします。

```
% loadjava -v -u scott/tiger -r Foo.jar
```

`loadjava -r (-resolve)` オプションを有効化すると、ソース・ファイルはサーバー側コンパイラにより自動的にコンパイルされ、クラス・スキーマ・オブジェクトが生成されます。`-u (-user)` オプションでは、データベース・スキーマのユーザー名とパスワードを指定します。オプションで、ステータス詳細が出力されるように `-v (-verbose)` オプションを使用します。

また、次のようにファイルを個別にロードすることもできます。

```
% loadjava -v -u scott/tiger -r Foo.java Foo.res
```

または、次のようにワイルドカードを使用してロードします。

```
% loadjava -v -u scott/tiger -r Foo.*
```

前述の例では、いずれもデータベース内に次のスキーマ・オブジェクトが作成されます (通常、知る必要があるのは、ページ実装クラスのスキーマ・オブジェクト名のみです)。

- `SCOTT:Foo` ソース・スキーマ・オブジェクト

`loadjava` でソース・ファイルを Oracle8i にロードすると、サーバー側コンパイラにより生成されるクラス・スキーマ・オブジェクトに加えて、ソースがソース・スキーマ・オブジェクトとして別個に格納されます。

- SCOTT:Foo ページ実装クラス・スキーマ・オブジェクト

また、ojspc -packageName オプションや、ojspc を実行したときのカレント・ディレクトリに対する .jsp ファイルの相対位置によっては、Foo のクラスおよびソース・スキーマ・オブジェクトに関するパッケージ情報を追加指定できます。たとえば、-packageName に「abc.def」を設定すると、Foo クラスのパッケージになるため、SCOTT:abc/def/Foo クラス・スキーマ・オブジェクトが作成されます。

- SCOTT:Foo\$__jsp__StaticText クラス・スキーマ・オブジェクト

パッケージ指定はページ実装クラスと同じです。

- SCOTT:Foo.res リソース・スキーマ・オブジェクト

ロード時には、JAR ファイル内または loadjava コマンドラインでのパス指定に基づいて、パッケージ指定が使用されます。

loadjava により生成されるスキーマ・オブジェクトの命名方法の概要は、6-12 ページの「[Java 用のデータベース・スキーマ・オブジェクト](#)」を参照してください。

注意：

- この例で生成される名前として使用しているのは、あくまでも一例です。このような実装の詳細は将来のリリースで変更されることがありますが、生成される名前の一部として常にベース名（この例の「Foo」など）が使用されます。
 - SQLJ の JSP ページ用に変換済みのソース（.java）をロードする場合は、生成されたプロファイル・ファイルもロードする必要があります。このプロファイル・ファイルは、SQLJ の -ser2class オプションに応じて .ser Java リソース・ファイルまたは .class ファイルです。スキーマ・オブジェクトの命名は、.ser ファイルの場合は、.res Java リソース・ファイルに使用されるものと同じです。.class ファイルの場合は、他の .class ファイルに使用されるものと同じです。（ojspc -noCompile オプションでは、SQLJ 変換ではなく Java コンパイルが禁止されることに注意してください。）
-
-

Oracle8i でのページ実装クラスのホットロード

オプションで Oracle8i 内で変換済みの JSP ページを「ホットロード」するには、セッション・シェルの java コマンドを使用して、ページ実装クラス・スキーマ・オブジェクトの main() メソッドを起動します。このツールを起動してデータベースに接続する方法は、6-36 ページの「[sess_sh セッション・シェル・ツールの概要](#)」を参照してください。

変換中に、あらかじめ ojspc -hotload オプションでホットロードを有効化する必要があります。-hotload オプションを使用すると、main() メソッドとホットロード・メソッドがページ実装クラスに実装されます。main() メソッドを起動すると、ホットロード・メソッドがコールされ、ページ実装クラスがホットロードされます。

次に例を示します（\$ は `sess_sh` のプロンプトです）。

```
$ java SCOTT:Foo
```

-hotload オプションを有効化して変換し、loadjava で前述の例のように SCOTT スキーマにロードしたクラスを Foo とすると、このセッション・シェルの java コマンドでは、Foo ページ実装クラスがホットロードされます。

ホットロードの概要は、6-20 ページの「[Oracle8i にホットロードされるクラスの概要](#)」を参照してください。セッション・シェルの java コマンドの詳細は、『Oracle8i Java Tools リファレンス』を参照してください。

Oracle8i での変換済み JSP ページの公開（セッション・シェルの publishservlet）

「クライアント側変換を伴う配布」使用例の一部として変換済みページを公開するには、セッション・シェルの publishservlet コマンドを使用します。このツールを起動してデータベースに接続する方法は、6-36 ページの「[sess_sh セッション・シェル・ツールの概要](#)」を参照してください。

publishservlet コマンドは、OSE 内で実行するサーブレットを公開するための汎用コマンドですが、JSP ページ実装クラス（実際にはサーブレット）にも適用されます。

注意： publishservlet で公開される JSP ページは、セッション・シェルの unpublishservlet コマンドを使用して「非公開」に（Oracle8i JVM の JNDI ネームスペースから削除）できます。6-62 ページの「[unpublishservlet を使用した JSP ページの非公開](#)」を参照してください。

publishservlet の構文とオプションの概要

sess_sh を起動すると、Oracle8i データベースへの接続が確立されます。sess_sh の起動後は、セッション・シェルの \$ プロンプトから publishservlet コマンドを実行できます。

publishservlet コマンドの一般構文は、次のとおりです。

```
$ publishservlet context servletName className -virtualpath path [-stateless] [-reuse] [-properties props]
```

publishservlet を使用する場合は、次を指定する必要があります。

1. サーブレット・コンテキスト（前述のコマンドラインでは context）

これは、publishservlet に必須です。次のように、Oracle Servlet Engine のデフォルトのサーブレット・コンテキストを使用できます。

```
/webdomains/contexts/default
```

その結果、コンテキスト・パスは「/」となります。

他のサーブレット・コンテキストを指定すると、そのサーブレット・コンテキストのコンテキスト・パスが使用されます。

たとえば、次のように作成されたサーブレット・コンテキスト mycontext を指定します。

```
$ createcontext -virtualpath mywebapp /webdomains mycontext
```

公開される JSP ページのコンテキスト・パスは mywebapp となります。

2. サーブレット名（前述のコマンドラインでは *servletName*）

これは、publishservlet で named_servlets ディレクトリ内の JSP ページ名を指定するには必須ですが、JSP 開発者やユーザーにとっては非公開以外に実際的な用途はありません。任意の文字列を指定できます。

3. クラス名（前述のコマンドラインでは *className*）

これは、公開するページ実装クラス・スキーマ・オブジェクトの名前です。

4. サーブレット・パス（コマンドラインでは「仮想パス」と呼ばれます）

-virtualpath オプションを使用します。これは JSP ページには必須ですが、通常、サーブレットを公開する場合はオプションです。

コンテキスト・パスとサーブレット・パス（およびホスト名とポート）により、ページを起動するための URL が決定されます。6-14 ページの「[Oracle Servlet Engine の URL](#)」を参照してください。

重要：

- サーブレット・コンテキスト、サーブレット名およびクラス名の前には指定構文がないため、コマンドラインでは前述の相互の相対順で指定する必要があります。（ただし、どの publishservlet オプションも、これらのパラメータと併用できます。）
 - -stateless などのブール・オプションを有効化するには、コマンドラインにオプション名のみを入力します（true に設定しないでください）。
-
-

必須パラメータに加えて、次のいずれかのオプションを指定できます。

■ -stateless

これは、Oracle Servlet Engine に対して、JSP ページをステートレスにするように指示するブール・オプションです。JSP ページには、実行中に HttpSession オブジェクトへのアクセス権を付与しないようにします。

- `-reuse`

これは、JSP ページの新規サーブレット・パス（「仮想パス」とも呼ばれます）を指定するためのブール・オプションです。このオプションを有効化すると、指定したサーブレット・パスが JNDI ネームスペース内の指定したサーブレット名にリンクされ、`publishservlet` は公開プロセス全体を通過しません。

`-reuse` オプションを有効化する場合は、新規のサーブレット・パス、サーブレット・コンテキストおよび以前に公開されているサーブレット名を指定します。

- `-properties props`

このオプションを使用して、実行時に初期化パラメータとして JSP ページに渡すプロパティを指定します。

`publishservlet` コマンドの詳細は、『Oracle8i Java Tools リファレンス』を参照してください。

例 : `publishservlet` を使用した JSP ページの公開

次の例では、Oracle8i にロードされた JSP ページを公開しています（\$ は `sess_sh` のプロンプトです）。

```
$ publishservlet /webdomains/contexts/default -virtualpath Foo.jsp FooServlet SCOTT:Foo
```

例を単純化するために、OSE のデフォルトのサーブレット・コンテキストが指定されているため、コンテキスト・パスは「/」となります。

サーブレット・パスは `Foo.jsp` となります。（サーブレット・パスには必要な名前を指定できますが、元のソース・ファイル名に基づいて命名することをお勧めします。）

`FooServlet` は OSE の `named_servlets` ディレクトリ内のサーブレット名になりますが、通常、この名前は非公開にする場合以外は使用されません。

`SCOTT:Foo` は、公開するページ実装クラス・スキーマ・オブジェクトです。

前述の `publishservlet` コマンドの後に、エンド・ユーザーは次のように URL を指定して JSP ページを起動します。

```
http://host[:port]/Foo.jsp
```

このファイルには、次のように（ページ相対構文、次にアプリケーション相対構文を使用して）アプリケーション内の別の JSP ページから動的にアクセスします。この JSP ページは、`Bar.jsp` として公開されているとします。

```
<jsp:include page="Foo.jsp" flush="true" />
```

または

```
<jsp:include page="/Foo.jsp" flush="true" />
```

注意： `publishservlet` コマンドで指定したサーブレット・パスとサーブレット名は、Oracle8i JVM の JNDI ネームスペースに入力されますが、通常、JSP ユーザーにとって重要なのはサーブレット・パスのみです。OSE では、公開されている JSP ページやサーブレットの検索に JNDI が使用されます。

unpublishservlet を使用した JSP ページの非公開

`sess_sh` ツールには、サーブレットや JSP ページを Oracle8i JVM の JNDI ネームスペースから削除する `unpublishservlet` コマンドも用意されています。ただし、このコマンドでは、サーブレット・クラスのスキーマ・オブジェクトやページ実装クラスのスキーマ・オブジェクトは、データベースから削除されません。

コンテキスト、サーブレット・パス（コマンドラインでは「仮想パス」と呼ばれます）とサーブレット名を指定します。JSP ページを非公開にする一般構文は、次のとおりです。

```
$ unpublishservlet -virtualpath path context servletName
```

たとえば、前項で公開したページを非公開にするには、次のコマンドを使用します。

```
$ unpublishservlet -virtualpath Foo.jsp /webdomains/contexts/default FooServlet
```

JSP 配布に関するその他の考慮事項

データベース内での実行は、特別な考慮とロジスティックを必要とする特殊な状況のため、この章の大部分では、Oracle Servlet Engine をターゲットとする変換と配布を重点的に説明しています。

この項では、Oracle Servlet Engine をターゲットとしない場合を中心として、配布に関するその他の様々な考慮事項と使用例について説明します。

説明するトピックは、次のとおりです。

- [Oracle Internet Application Server と Oracle Servlet Engine のドキュメント・ルートの比較](#)
- [非 OSE 環境向けの事前変換のための ojspc の使用](#)
- [実行なしの一般的な JSP 事前変換](#)
- [バイナリ・ファイルのみの配布](#)
- [WAR の配布](#)
- [JDeveloper を使用した JSP ページの配布](#)

Oracle Internet Application Server と Oracle Servlet Engine のドキュメント・ルートの比較

Oracle Servlet Engine と Oracle Internet Application Server では、実際には Apache 環境である Oracle HTTP Server が、HTTP 要求の Web サーバーとして使用されます。ただし、各環境では独自のドキュメント・ルートが使用されます。

オラクル社が提供する Apache の `mod_ose` モジュールを介してルーティングされ、Oracle Servlet Engine 内で実行される JSP ページとサーブレットでは、関連サーブレット・コンテキストの OSE ドキュメント・ルートが使用されます。OSE のドキュメント・ルート・ディレクトリはファイル・システムにありますが、Oracle8i の JNDI メカニズムにリンクされています。

OSE 内で実行される JSP ページの場合、ドキュメント・ルートまたはサブディレクトリにあるのは静的ファイルのみなので注意してください。JSP ページはデータベース内にあります。

OSE ドキュメント・ルート・ディレクトリは、デフォルトのドキュメント・ルート `$ORACLE_HOME/jis/public_html`、またはサーブレット・コンテキストの作成時にセッション・シェルの `createcontext` コマンドの `-docroot` オプションで指定したドキュメント・ルートです。

JServ に付属の Apache `mod_jserv` モジュールを介してルーティングされ、Oracle Internet Application Server (リリース 1.0.x) の Apache/JServ 環境で実行される JSP ページとサーブレットには、Apache ドキュメント・ルートが使用されます。このドキュメント・ルート (通常は `htdocs`) は、Apache の `httpd.conf` 構成ファイルの `DocumentRoot` コマンドで設定します。

JServ 内で実行される JSP ページは、静的ファイルとともにドキュメント・ルートまたはサブディレクトリにあります。

Apache/JServ 環境と OSE 環境の間で移行する場合は、静的ファイルを該当のドキュメント・ルートに移動またはコピーします。

注意： Oracle HTTP Server とその `mod_ose` および `mod_jserv` モジュールのロールの概要は、2-5 ページの「[Apache により駆動される Oracle HTTP Server のロール](#)」を参照してください。

非 OSE 環境向けの事前変換のための ojspc の使用

通常、Oracle の ojspc ツールは、Oracle8i に配布するためのクライアント側 JSP 変換に使用します。このツールの詳細は、6-22 ページの「[ojspc 事前変換ツール](#)」を参照してください。ただし、ojspc を使用すると、どの環境でも JSP ページを事前変換できます。これは、ページを初めて実行するときにエンド・ユーザーを変換オーバーヘッドから解放するという点で役立つ場合があります。

他のターゲット環境で事前変換する場合は、ojspc -d オプションを指定して、生成されるバイナリ・ファイルを配置する適切なベース・ディレクトリを設定します。

たとえば、次の JSP ソース・ファイルを含む Apache/JServ 環境があるとします。

```
htdocs/test/foo.jsp
```

ユーザーが、次の URL を使用してこのファイルを起動します。

```
http://host[:port]/test/foo.jsp
```

実行時のオンデマンド変換中に、OracleJSP トランスレータは、生成されたバイナリ・ファイルの格納場所として、ベース・ディレクトリ htdocs/_pages を使用します。したがって、事前変換する場合は、次のように、htdocs/_pages をバイナリ出力のベース・ディレクトリとして設定する必要があります（% は UNIX プロンプトとします）。

```
% cd htdocs
% ojspc -d _pages test/foo.jsp
```

前述の URL は、アプリケーション相対パス test/foo.jsp を指定しているため、実行時には OracleJSP コンテナによりバイナリ・ファイルが htdocs/_pages ディレクトリの test サブディレクトリ内で検索されます。前述のように ojspc を実行すると、このサブディレクトリが自動的に作成されます。実行時には、OracleJSP コンテナにより事前変換済みのバイナリ・ファイルが検索され、事前変換後にソース・ファイルが変更されていなければ、変換を実行する必要はありません。（デフォルトでは、ソース・ファイルのタイムスタンプがバイナリ・ファイルのタイムスタンプより新しく、ソース・ファイルが使用可能で、bypass_source 構成パラメータが有効化されていなければ、ページが再変換されます。）

実行なしの一般的な JSP 事前変換

オンデマンド変換環境では、エンド・ユーザーのブラウザから JSP ページを起動するときに、jsp_precompile 要求パラメータを有効化して、JSP 事前変換のみを指定し、実行しないようにすることができます。

次に例を示します。

```
http://host[:port]/foo.jsp?jsp_precompile
```

詳細は、Sun Microsystems の『JavaServer Pages Specification, Version 1.1』を参照してください。

バイナリ・ファイルのみの配布

JSP ソースが所有権付きの場合は、JSP ページを事前変換し、変換およびコンパイル済みのバイナリ・ファイルのみを配布して、ソースの公開を回避できます。オンデマンド変換の使用例の前の実行で、または `ojspc` を使用して事前変換済みのページは、OracleJSP コンテナをサポートするどの環境にも配布できます。この使用例には、次の 2 つの側面があります。

- バイナリ・ファイルを適切に配布する必要があります。
- ターゲット環境では、`.jsp`（または `.sqljsp`）ソースが使用できない場合にページを実行するように、OracleJSP を適切に構成する必要があります。

バイナリ・ファイルの配布

JSP ページの変換後に、バイナリ出力ディレクトリの下のディレクトリ構造と内容をアーカイブし、必要に応じてターゲット環境にコピーします。次に例を示します。

- `ojspc` で事前変換する場合は、`ojspc -d` オプションでバイナリ出力ディレクトリを指定し、そのディレクトリの下のディレクトリ構造をアーカイブする必要があります。
- Apache/JServ（オンデマンド変換）環境で前回の実行中に生成されたバイナリ・ファイルをアーカイブする場合は、出力ディレクトリ構造をアーカイブします。このディレクトリ構造は、通常は `htdocs/_pages` ディレクトリにあります。

ターゲット環境では、アーカイブしたディレクトリ構造を、Apache/JServ 環境の `htdocs/_pages` ディレクトリなど、適切なディレクトリにリストアします。

バイナリ・ファイルのみで実行するための OracleJSP の構成

`.jsp` または `.sqljsp` ソースを使用できない場合に JSP ページを実行するように、OracleJSP の構成パラメータを次のように設定します。

- `bypass_source=true`
- `developer_mode=false`

このように設定しない場合は、OracleJSP では常に `.jsp` または `.sqljsp` ファイルが検索され、最終変更日時がページ実装の `.class` ファイルより新しいかどうかチェックされ、`.jsp` または `.sqljsp` ファイルが見つからない場合は「file not found」エラーを戻して異常終了します。

前述のパラメータを適切に設定すると、エンド・ユーザーはソース・ファイルが存在する場合に使用するのと同じ URL を指定してページを起動できます。たとえば、Apache/JServ 環境で、`foo.jsp` のバイナリ・ファイルが `htdocs/_pages/test` ディレクトリにあれば、`foo.jsp` が存在しなくても、次の URL を指定してページを起動できます。

```
http://host:[port]/test/foo.jsp
```

この構成パラメータの設定方法は、A-23 ページの「[OracleJSP の構成パラメータ設定](#)」を参照してください。

WAR の配布

Sun Microsystems の『Java Servlet Specification, Version 2.2』によれば、同社の『JavaServer Pages Specification, Version 1.1』では、JavaServer Pages などの Web アプリケーションのパッケージ化と配布がサポートされています。

典型的な JSP 1.1 実装では、JSP ページは Web アーカイブ (WAR) ・メカニズムを介して配布されます。WAR ファイルを作成するには、JAR ユーティリティを使用します。JSP ページをソース形式で配信し、他の必須サポート・クラスや静的 HTML ファイルとともに配布できます。

サーブレット 2.2 仕様によれば、Web アプリケーションにはディプロイメント・ディスクリプタ・ファイル `web.xml` が組み込まれており、このファイルにはアプリケーションの JSP ページや他のコンポーネントに関する情報が含まれています。WAR ファイルには、`web.xml` ファイルを挿入する必要があります。

また、サーブレット 2.2 仕様では、`web.xml` ディプロイメント・ディスクリプタ用の XML DTD が定義されており、サーブレット・コンテナで Web アプリケーションをディプロイメント・ディスクリプタに準拠するように配布する方法が正確に指定されています。

これらのロジスティックでは、WAR ファイルは Web アプリケーションが開発者が意図したとおり標準的なサーブレット環境に配布されるように保証するには最善の方法です。

`web.xml` ディプロイメント・ディスクリプタ内の配布構成には、サーブレット・パスと起動される JSP ページおよびサーブレットとのマッピングが含まれています。`web.xml` では、多数の機能の追加指定もできます。たとえば、アプリケーション・モジュールのタイムアウト値、ファイル拡張子から MIME タイプへのマッピング、エラー・コードから JSP エラー・ページへのマッピングなどです。

まとめると、WAR ファイルの内容は次のとおりです。

- `web.xml` ディプロイメント・ディスクリプタ
- JSP ページ
- 必須の JavaBeans および他のサポート・クラス
- 必須の静的 HTML ファイル

詳細は、Sun Microsystems の『Java Servlet Specification, Version 2.2』を参照してください。

注意： リリース 8.1.7 の OracleJSP では `web.xml` の用途が限定的であり、JSP のタグ・ライブラリ記述子とサープレットの URL ショートカットにのみ使用されます。

JDeveloper を使用した JSP ページの配布

Oracle JDeveloper リリース 3.1 には、JSP アプリケーション専用の配布オプション「Web Application to Web Server」が追加されています。

このオプションでは、次を指定する配布プロファイルが生成されます。

- JSP アプリケーションに必要な Business Components for Java (BC4J) クラスを含む JAR ファイル
- JSP アプリケーションに必要な静的 HTML ファイル
- Web サーバーへのパス

開発者は、プロファイルの作成と同時にアプリケーションを配布するか、後で使用できるように保存できます。

JSP のタグ・ライブラリと Oracle の JML タグ

この章では、カスタム・タグ・ライブラリと、ベンダーが独自のライブラリを提供し、OracleJSP でサンプルとして提供される JML タグ・ライブラリをドキュメント化するときに使用できる基本的なフレームワークについて説明します。説明するトピックは、次のとおりです。

- [標準タグ・ライブラリのフレームワーク](#)
- [JSP マークアップ言語 \(JML\) のサンプル・タグ・ライブラリの概要](#)
- [JSP マークアップ言語 \(JML\) タグの説明](#)

標準タグ・ライブラリのフレームワーク

標準の JavaServer Pages テクノロジーにより、ベンダーはカスタム JSP タグ・ライブラリを作成できます。

タグ・ライブラリでは、カスタム・アクションのコレクションを定義します。タグは、開発者が JSP ページをコーディングするときに手動で直接使用する方法と、Java 開発ツールで自動的に使用する方法があります。タグ・ライブラリは、様々な JSP コンテナ実装間で移植可能である必要があります。

タグ・ライブラリと標準的な JavaServer Pages タグ・ライブラリのフレームワークに関して、この章で説明しない情報は、次のドキュメントを参照してください。

- Sun Microsystems の『JavaServer Pages Specification, Version 1.1』
- 次の Web サイトからダウンロードできる、Sun Microsystems の `javax.servlet.jsp.tagext` パッケージの Javadoc

<http://java.sun.com/j2ee/j2sdkee/techdocs/api/javax/servlet/jsp/tagext/package-summary.html>

注意： カスタム・タグを使用する場合は、Tomcat 3.1 ベータのサーブレット (JSP 実装) の `servlet.jar` ファイルを使用しないでください。
`javax.servlet.jsp.tagext.TagAttributeInfo` クラスのコンストラクタ・シグネチャが変更されており、コンパイル・エラーになります。
かわりに、OracleJSP または本番バージョンの Tomcat 3.1 で提供される `servlet.jar` ファイルを使用してください。

カスタム・タグ・ライブラリの実装の概要

カスタム・タグ・ライブラリは、次の一般書式の `taglib` ディレクティブを使用して JSP ページにインポートします。

```
<%@ taglib uri="URI" prefix="prefix" %>
```

次の点に注意してください。

- ライブラリの各タグは、タグ・ライブラリ記述ファイルで定義します。7-10 ページの「[タグ・ライブラリ記述ファイル](#)」を参照してください。
- `taglib` ディレクティブ内の URI で、タグ・ライブラリ記述ファイルの検索先を指定します。7-12 ページの「[taglib ディレクティブ](#)」を参照してください。可能であれば、7-11 ページの「[タグ・ライブラリ用の web.xml の使用](#)」で説明するように、URI ショートカットを使用してください。
- `taglib` ディレクティブ内の接頭辞は、ライブラリ内のいずれかのタグを含む JSP ページに使用するよう選択する文字列です。

たとえば、`taglib` ディレクティブで接頭辞 `oracust` を指定するとします。


```
<%@ taglib uri="URI" prefix="oracust" %>
```

また、ライブラリにタグ mytag があるとします。mytag を次のように使用できます。

```
<oracust:mytag attr1="...", attr2="..." />
```

oracust 接頭辞を使用して、JSP トランスレータに対し、mytag がタグ・ライブラリ記述ファイルに定義されており、そのファイルが前述の taglib ディレクティブに指定されている URI で検索できることを通知します。

- タグ・ライブラリ記述ファイル内のタグ・エントリでは、タグで（mytag のように）属性を使用するかどうかなど、タグの使用方法和、各属性の名前が指定されています。
- タグの意味、つまり、そのタグを使用した結果として発生するアクションは、タグ・ハンドラ・クラスで定義されています。7-4 ページの「[タグ・ハンドラ](#)」を参照してください。各タグには独自のタグ・ハンドラ・クラスがあり、クラス名はタグ・ライブラリ記述ファイルに指定されています。
- タグ・ライブラリ記述ファイルは、タグで本体が使用されるかどうかを示します。

前述のように、本体を持たないタグは、次のように使用されます。

```
<oracust:mytag attr1="...", attr2="..." />
```

これに対して、本体を持つタグは、次のように使用されます。

```
<oracust:mytag attr1="...", attr2="..." >
    ...body...
</oracust:mytag>
```

- 1 つのカスタム・タグ・アクションで、タグ自体またはスクリプトレットのような他の JSP スクリプト要素で使用可能な、1 つ以上のサーバー側オブジェクトを作成できます。このようなオブジェクトは、スクリプト変数と呼ばれます。

カスタム・タグで使用されるスクリプト変数の詳細は、Tag-Extra-Info クラスで定義されています。7-7 ページの「[スクリプト変数と Tag-Extra-Info クラス](#)」を参照してください。

タグでは、次のような構文でスクリプト変数を作成できます。次の例では、オブジェクト myobj を作成しています。

```
<oracust:mytag id="myobj" attr1="...", attr2="..." />
```

- ネストしたタグのタグ・ハンドラでは、外部タグのタグ・ハンドラにアクセスできます。これは、ネストしたタグの処理や状態を管理するために必要です。7-9 ページの「[外部タグ・ハンドラ・インスタンスへのアクセス](#)」を参照してください。

以降の項では、このトピックで説明した内容について詳しく説明します。

タグ・ハンドラ

タグ・ハンドラでは、カスタム・タグの使用によって生じるアクションの意味が記述されます。タグ・ハンドラは、開始タグと終了タグで囲まれた文本体をタグで処理するかどうかに応じて、2つの標準 Java インタフェースのどちらかを実装する Java クラスのインスタンスです。

各タグには独自のハンドラ・クラスがあります。たとえば、表記規則により、タグ abc のタグ・ハンドラ・クラス名は `AbcTag` となります。

タグ・ライブラリのタグ・ライブラリ記述 (TLD) ファイルでは、ライブラリ内のタグごとにタグ・ハンドラ・クラスの名前が指定されます。(7-10 ページの「[タグ・ライブラリ記述ファイル](#)」を参照してください。)

タグ・ハンドラ・インスタンスは、要求時に使用されるサーバー側オブジェクトです。このタグ・ハンドラ・インスタンスには JSP コンテナで設定されるプロパティがあります。このプロパティには、カスタム・タグを使用する JSP ページのページ・コンテキスト・オブジェクトや、このカスタム・タグの使用を外部カスタム・タグ内でネストする場合の親タグ・ハンドラなどがあります。

タグ・ハンドラ・クラスのサンプル・コードは、7-14 ページの「[サンプル・タグ・ハンドラ・クラス: `ExampleLoopTag.java`](#)」を参照してください。

注意： Sun Microsystems の『JavaServer Pages Specification, Version 1.1』では、JSP ページ内で同じカスタム・タグを複数回使用する場合に、同じタグ・ハンドラ・インスタンスを使用する必要があるかどうかは強制されていません。この実装の詳細は、JSP ベンダーの裁量に委ねられています。OracleJSP では、タグの使用ごとに別個のタグ・ハンドラ・インスタンスを使用しています。

カスタム・タグの本体の処理

標準 JSP タグと同様に、カスタム・タグには本体があってもなくてもかまいません。また、カスタム・タグの場合は、本体があっても、タグ・ハンドラによる特殊な処理は必要ありません。

次の3つの場合があります。

- 本体がない場合

この場合は、開始タグと終了タグがある場合と異なりタグは1つしかありません。次に一般的な例を示します。

```
<oracust:abcdef attr1="...", attr2="..." />
```

- タグ・ハンドラによる特殊な処理を必要としない本体がある場合

この場合は、文の本体が開始タグと終了タグで囲まれています。タグ・ハンドラで本体を処理する必要はなく、本体の文に対して通常の JSP 処理のみが行われます。次に一般的な例を示します。

```
<foo:if cond="<%= ... %>" >
...body executed if cond is true, but not processed by tag handler...
</foo:if>
```

- タグ・ハンドラによる特殊な処理を必要とする本体がある場合

この場合も、文の本体が開始タグと終了タグで囲まれています。ただし、タグ・ハンドラで本体を処理する必要があります。

```
<oracust:ghijkl attr1="...", attr2="..." >
...body processed by tag handler...
</oracust:ghijkl>
```

本体処理のための整定数

以降の項で後述するタグ処理インタフェースは、doStartTag() メソッド（後述）を指定します。このメソッドは、状況に応じて適切な整定数を戻すように実装する必要があります。可能な戻り値は、次のとおりです。

- SKIP_BODY。本体がない場合、または本体の評価と実行をスキップする必要がある場合の戻り値です。
- EVAL_BODY_INCLUDE。タグ・ハンドラによる特殊な処理を必要としない本体がある場合の戻り値です。
- EVAL_BODY_TAG。タグ・ハンドラによる特殊な処理を必要とする本体がある場合の戻り値です。

本体を処理しないタグのハンドラ

本体を持たない、またはタグ・ハンドラによる特殊な処理を必要としない本体を持つカスタム・タグの場合は、タグ・ハンドラ・クラスにより次の標準インタフェースが実装されます。

- javax.servlet.jsp.tagext.Tag

次の標準サポート・クラスでは、Tag インタフェースが実装され、ベース・クラスとして使用できます。

- javax.servlet.jsp.tagext.TagSupport

Tag インタフェースは、doStartTag() メソッドと doEndTag() メソッドを指定します。タグ開発者は、開始タグと終了タグのそれぞれの検出時にこの 2 つのメソッドが実行されるよう、タグ・ハンドラ・クラス内でこれらのメソッドを適宜コーディングします。アクション

ン・タグで実行するアクションの処理は、すべて `doStartTag()` メソッド内で実装されます。`doEndTag()` メソッドでは、適切な後処理が実装されます。本体を持たないタグの場合、この2つのメソッドの実行間では実際にはなにも発生しません。

`doStartTag()` メソッドは整数値を戻します。Tag インタフェースを（直接または間接的に）実装するタグ・ハンドラ・クラスの場合、この値は `SKIP_BODY` または `EVAL_BODY_INCLUDE` である必要があります（7-5 ページの「[本体処理のための整数値](#)」を参照）。`EVAL_BODY_TAG` は、Tag インタフェースを実装するタグ・ハンドラ・クラスでは無効です。

本体を処理するタグのハンドラ

タグ・ハンドラによる特殊な処理が必要な本体を持つカスタム・タグの場合は、タグ・ハンドラ・クラスにより次の標準インタフェースが実装されます。

- `javax.servlet.jsp.tagext.BodyTag`

次の標準サポート・クラスでは、`BodyTag` インタフェースが実装され、ベース・クラスとして使用できます。

- `javax.servlet.jsp.tagext.BodyTagSupport`

`BodyTag` インタフェースは、Tag インタフェースで指定された `doStartTag()` および `doEndTag()` メソッドに加えて、`doInitBody()` メソッドと `doAfterBody()` メソッドを指定します。

Tag インタフェースを実装するタグ・ハンドラ（前述の「[本体を処理しないタグのハンドラ](#)」を参照）と同様に、タグ開発者はタグによるアクション処理用の `doStartTag()` メソッドと、後処理用の `doEndTag()` メソッドを実装します。

`doStartTag()` メソッドは整数値を戻します。`BodyTag` インタフェースを（直接または間接的に）実装するタグ・ハンドラ・クラスの場合、この値は `SKIP_BODY` または `EVAL_BODY_TAG` である必要があります（7-5 ページの「[本体処理のための整数値](#)」を参照してください）。`EVAL_BODY_INCLUDE` は、`BodyTag` インタフェースを実装するタグ・ハンドラ・クラスでは無効です。

タグ開発者は、`doStartTag()` および `doEndTag()` メソッドを実装するのみでなく、必要に応じて、本体の評価前にコールする `doInitBody()` メソッドと、本体が評価されるたびにコールする `doAfterBody()` メソッドをコーディングします。（本体は、ループの反復が終了するたびなど、複数回評価できます。）

`doStartTag()` メソッドの実行後に `EVAL_BODY_TAG` が戻された場合は、`doInitBody()` および `doAfterBody()` メソッドが実行されます。

`doEndTag()` メソッドは、本体処理後に終了タグが検出されると実行されます。

本体を処理する必要があるカスタム・タグの場合は、

`javax.servlet.jsp.tagext.BodyContent` クラスを使用できます。これは、本体評価を後から再抽出できるように処理する際に使用できる、`javax.servlet.jsp.JspWriter` のサブクラスです。`BodyTag` インタフェースには `setBodyContent()` メソッドが含まれて

います。JSP コンテナはこのメソッドを使用して、タグ・ハンドラ・インスタンスに `BodyContent` ハンドルを渡すことができます。

スクリプト変数と Tag-Extra-Info クラス

1つのカスタム・タグ・アクションで、スクリプト変数と呼ばれる1つ以上のサーバー側オブジェクトを作成できます。このオブジェクトは、タグ自体、またはスクリプトレットや他のタグなどの他のスクリプト要素で使用できます。

カスタム・タグで定義するスクリプト変数の詳細は、標準の `javax.servlet.jsp.tagext.TagExtraInfo` 抽象クラスのサブクラスで指定する必要があります。このマニュアルでは、このようなサブクラスを **Tag-Extra-Info クラス**と呼びます。

JSP コンテナでは、Tag-Extra-Info インスタンスが変換中に使用されます。（ライブラリを JSP ページにインポートする `taglib` ディレクティブで指定されたタグ・ライブラリ記述ファイルでは、該当する場合に特定のタグに使用する Tag-Extra-Info クラスを指定します。）

Tag-Extra-Info クラスには、HTTP 要求時に割り当てられるスクリプト変数の名前と型を取得する `getVariableInfo()` メソッドがあります。JSP トランスレータは、変換中にこのメソッドをコールし、標準 `javax.servlet.jsp.tagext.TagData` クラスのインスタンスに渡します。TagData インスタンスは、カスタム・タグを使用する JSP 文で属性値セットを指定します。

この項で説明するトピックは、次のとおりです。

- [スクリプト変数の定義](#)
- [スクリプト変数の有効範囲](#)
- [Tag-Extra-Info クラスと `getVariableInfo\(\)` メソッド](#)

スクリプト変数の定義

カスタム・タグで明示的に定義されているオブジェクトは、オブジェクト ID をハンドルとして使用し、ページ・コンテキスト・オブジェクトを介して他のアクション内で参照できます。次に例を示します。

```
<oracust:foo id="myobj" attr1="..." attr2="..." />
```

この文では、オブジェクト `myobj` が、タグからページの終わりまでのすべてのスクリプト要素で使用可能になります。`id` 属性は変換時属性です。タグ開発者は、JSP コンテナで 사용되는 Tag-Extra-Info クラスを提供します。特に、Tag-Extra-Info クラスでは、`myobj` オブジェクト用にインスタンス化するクラスを指定します。

`myobj` は JSP コンテナによりページ・コンテキスト・オブジェクトに挿入され、後から次のような構文を使用して他のタグやスクリプト要素で取得できます。

```
<oracust:bar ref="myobj" />
```

myobj オブジェクトは、foo および bar のタグ・ハンドラ・インスタンスを介して渡されます。知る必要があるのは、オブジェクト名 (myobj) のみです。

重要： id と ref は単なるサンプル属性名で、これらの属性には事前定義済みの特別な意味はないことに注意してください。属性名を定義し、ページ・コンテキストでオブジェクトを作成および取得するのは、タグ・ハンドラで実行します。

スクリプト変数の有効範囲

スクリプト変数の有効範囲は、その変数を作成するタグの Tag-Extra-Info クラスで指定します。次の整定数の 1 つを使用できます。

- NESTED — スクリプト変数が、定義された開始タグと終了タグの間で使用可能な場合
- AT_BEGIN — スクリプト変数が、開始タグからページの終わりまでの範囲内で使用可能な場合
- AT_END — スクリプト変数が、終了タグからページの終わりまでの範囲内で使用可能な場合

Tag-Extra-Info クラスと getVariableInfo() メソッド

スクリプト変数を作成するすべてのカスタム・タグ用に、Tag-Extra-Info クラスを作成する必要があります。このクラスはスクリプト変数を記述し、標準の

javax.servlet.jsp.tagext.TagExtraInfo 抽象クラスのサブクラスにする必要があります。

TagExtraInfo クラスの主要メソッドは getVariableInfo() です。このメソッドは、JSP トランスレータによりコールされ、標準

javax.servlet.jsp.tagext.VariableInfo クラスのインスタンスの配列（タグにより作成されるスクリプト変数ごとに 1 つの配列インスタンス）を戻します。

Tag-Extra-Info クラスでは、スクリプト変数に関する次の情報を使用して、各 VariableInfo インスタンスが構築されます。

- 名前
- Java の型
- 新規に宣言される変数かどうかを示すブール
- 有効範囲

重要： `getVariableInfo()` メソッドは、スクリプト変数の Java の型について、JML データ型などの完全修飾クラス名を戻す必要があります。(プリミティブ型はサポートされないため注意してください。)

Tag-Extra-Info クラスのサンプル・コードは、7-15 ページの「[サンプル Tag-Extra-Info クラス : ExampleLoopTagTEI.java](#)」を参照してください。

外部タグ・ハンドラ・インスタンスへのアクセス

ネストしたカスタム・タグが使用される場合、ネストしたタグのタグ・ハンドラ・インスタンスは外部タグのタグ・ハンドラ・インスタンスへのアクセス権を持ちます。これは、ネストしたタグにより実行される処理と状態管理に役立つ場合があります。

この機能は、`javax.servlet.jsp.tagext.TagSupport` クラスの静的 `findAncestorWithClass()` メソッドを介してサポートされます。外部タグ・ハンドラ・インスタンスがページ・コンテキスト・オブジェクト内で指定されていなくても、特定のタグ・ハンドラ・クラスの外側にある最も近いインスタンスであるためアクセス可能です。

たとえば、次の JSP コードがあるとします。

```
<foo:bar1 attr="abc" >
  <foo:bar2 />
</foo:bar1>
```

`bar2` タグ・ハンドラ・クラス（表記規則によりクラスの `Bar2Tag`）のコード内では、次のような文を使用できます。

```
Tag bar1tag = TagSupport.findAncestorWithClass(this, Bar1Tag.class);
```

`findAncestorWithClass()` メソッドは、次の入力を取ります。

- `findAncestorWithClass()` のコール元となったクラス・ハンドラ・インスタンス（この例では `Bar2Tag` インスタンス）である `this` オブジェクト
- `java.lang.Class` インスタンスとしての、`bar1` タグ・ハンドラ・クラスの名前（この例では `Bar1Tag`）

`findAncestorWithClass()` メソッドは、該当するタグ・ハンドラ・クラスのインスタンス（この例では `Bar1Tag`）を、`javax.servlet.jsp.tagext.Tag` インスタンスとして戻します。

`Bar2Tag` に `bar1` タグ属性の値が必要な場合や、`Bar1Tag` インスタンス上でメソッドをコールする必要がある場合に `Bar2Tag` インスタンスに外部 `Bar1Tag` インスタンスへのアクセス権があると役立ちます。

タグ・ライブラリ記述ファイル

タグ・ライブラリ記述（TLD）ファイルは、タグ・ライブラリとライブラリ内の個々のタグに関する情報を含む XML ドキュメントです。TLD ファイル名には、.tld 拡張子が付いています。

JSP コンテナでは、TLD ファイルを使用して、ライブラリからのタグが発生したときに実行するアクションが決定されます。

TLD ファイル内のタグ・エントリは、次のとおりです。

- カスタム・タグ名
- 対応するタグ・ハンドラ・クラスの名前
- 対応する Tag-Extra-Info クラスの名前（該当する場合）
- タグ本体（存在する場合）の処理方法を示す情報
- タグの属性情報（カスタム・タグを使用する場合に指定する属性）

次のサンプルは、タグ myaction の TLD ファイル・エントリを示しています。

```
<tag>
  <name>myaction</name>
  <tagclass>examples.MyactionTag</tagclass>
  <teiclass>examples.MyactionTagExtraInfo</teiclass>
  <bodycontent>JSP</bodycontent>
  <info>
    Perform a server-side action (one mandatory attr; one optional)
  </info>
  <attribute>
    <name>attr1</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>attr2</name>
    <required>false</required>
  </attribute>
</tag>
```

このエントリによれば、タグ・ハンドラ・クラスは MyactionTag で、Tag-Extra-Info クラスは MyactionTagExtraInfo です。属性 attr1 は必須で、属性 attr2 はオプションです。

bodycontent パラメータは、タグ本体（存在する場合）の処理方法を示します。有効な値は、次の3つです。

- 値 empty は、タグで本体を使用しないことを示します。
- 値 JSP は、タグ本体を JSP ソースとして処理し、変換する必要があることを示します。

- 値 `tagdependent` は、タグ本体を変換しないことを示します。本体に含まれるテキストは、すべて静的テキストとして処理されます。

JSP ページの `taglib` ディレクティブは、JSP コンテナに対して TLD ファイルの検索場所を示します。(7-12 ページの「[taglib ディレクティブ](#)」を参照してください。)

タグ・ライブラリ記述ファイルの詳細は、Sun Microsystems の『JavaServer Pages Specification, Version 1.1』を参照してください。

注意： Tomcat 3.1 のサーブレット /JSP 実装では、タグ自体 (JSP ページ内) に本体がないと、TLD ファイルでそのタグの `bodycontent` パラメータは読み込まれません。したがって、TLD ファイル内に意図せずに無効な `bodycontent` 値が生じる可能性があります (`empty` ではなく `none` など)。このファイルを OracleJSP のような他の JSP 環境で使用すると、エラーになります。

タグ・ライブラリ用の `web.xml` の使用

Sun Microsystems の『Java Servlet Specification, Version 2.2』では、サーブレット用の標準ディプロイメント・ディスクリプタ、`web.xml` ファイルを説明しています。JSP ページは、このファイルを使用して、JSP タグ・ライブラリ記述ファイルの位置を指定できます。

JSP タグ・ライブラリの場合、`web.xml` ファイルには `taglib` 要素と次の 2 つの副要素を挿入できます。

- `taglib-uri`
- `taglib-location`

`taglib-location` 副要素は、タグ・ライブラリ記述ファイルのアプリケーション相対位置 (「/」で始まる位置) を示します。

`taglib-uri` 副要素は、JSP ページの `taglib` ディレクティブで使用される「ショートカット」URI を示します。この URI は、添付の `taglib-location` 副要素で指定された TLD ファイル位置にマップされます。URI (universal resource indicator) という用語は URL (universal resource locator) という用語とほぼ同じ意味ですが、より汎用的です。

重要： JSP アプリケーションで `web.xml` ファイルが使用される場合は、そのアプリケーションとともに `web.xml` を配布する必要があります。このファイルは Java リソース・ファイルとして処理してください。

次のサンプルは、タグ・ライブラリ記述ファイルの `web.xml` エントリを示しています。

```
<taglib>
  <taglib-uri>/oracustomtags</taglib-uri>
```

```
<taglib-location>/WEB-INF/oracustomtags/tlds/MyTLD.tld</taglib-location>
</taglib>
```

これにより、/oracustomtags は、JSP ページの taglib ディレクティブ内の /WEB-INF/oracustomtags/tlds/MyTLD.tld と等価になります。例については、7-12 ページの「[TLD ファイルのショートカット URI の使用](#)」を参照してください。

web.xml デプロイメント・ディスクリプタおよびタグ・ライブラリ記述ファイルでのその使用方法の詳細は、Sun Microsystems の『Java Servlet Specification, Version 2.2』および『JavaServer Pages Specification, Version 1.1』を参照してください。

注意：

- Tomcat 3.1 のサーブレット（JSP 実装）からのサンプル web.xml ファイルは使用しないでください。標準 DTD XML 検証にパスしない新規要素が生じます。
 - web.xml ファイル内では、用語「uri」のかわりに「urn」を使用しないでください。「urn」の使用が許容されている JSP 実装（Tomcat 3.1 など）もありますが、「urn」を使用すると標準 DTD XML 検証にパスしなくなります。
-

taglib ディレクティブ

カスタム・ライブラリを JSP ページにインポートするには、次の書式で taglib ディレクティブを使用します。

```
<%@ taglib uri="URI" prefix="prefix" %>
```

URI には、次のオプションがあります。

- web.xml ファイルで定義されているショートカット URI を指定します（7-11 ページの「[タグ・ライブラリ用の web.xml の使用](#)」を参照してください）。
- タグ・ライブラリ記述（TLD）ファイル名と位置を完全に指定します。

TLD ファイルのショートカット URI の使用

タグ・ライブラリ記述ファイル MyTLD.tld 内で、タグ・ライブラリに関して次の web.xml エントリが定義されているとします。

```
<taglib>
  <taglib-uri>/oracustomtags</taglib-uri>
  <taglib-location>/WEB-INF/oracustomtags/tlds/MyTLD.tld</taglib-location>
</taglib>
```

この例では、JSP ページ内の次のディレクティブにより、JSP コンテナでは web.xml 内で /oracustomtags URI が検索されるため、タグ・ライブラリ記述ファイル (MyTLD.tld) の添付名と位置が検索されます。

```
<%@ taglib uri="/oracustomtags" prefix="oracust" %>
```

この文では、このカスタム・タグ・ライブラリの任意のタグを JSP ページに使用できます。

TLD ファイル名と位置の完全指定

タグ・ライブラリの使用に関して JSP アプリケーションを web.xml ファイルに依存させない場合は、次のように taglib ディレクティブでタグ・ライブラリ記述ファイル名および位置を完全に指定できます。

```
<%@ taglib uri="/WEB-INF/oracustomtags/tlds/MyTLD.tld" prefix="oracust" %>
```

位置は、アプリケーション相対位置として（この例のように先頭に「/」を付けて）指定します。関連情報については、1-7 ページの「[JSP ページの要求](#)」を参照してください。

または、taglib ディレクティブで .tld ファイルのかわりに .jar ファイルを指定できます。この場合、.jar ファイルにはタグ・ライブラリ記述ファイルが含まれています。タグ・ライブラリ記述ファイルの位置と名前は、JAR ファイルの作成時に次のように指定する必要があります。

```
META-INF/taglib.tld
```

これにより、次のような taglib ディレクティブを使用できます。

```
<%@ taglib uri="/WEB-INF/oracustomtags/tlds/MyTLD.jar" prefix="oracust" %>
```

全体の例：カスタム・タグの定義と使用

この項では、タグ本体を指定した回数だけ反復するために使用するカスタム・タグ loop の定義と使用に関する全体的な例を示します。

例に含まれる要素は、次のとおりです。

- タグを使用するページの JSP ソース
- タグ・ハンドラ・クラスのソース・コード
- Tag-Extra-Info クラスのソース・コード
- タグ・ライブラリ記述ファイル

サンプル JSP ページ : exampletag.jsp

次のサンプルは、loop タグを使用する JSP ページを示しており、外部ループを 5 回、内部ループを 3 回実行するように指定しています。

```
exampletag.jsp
<%@ taglib prefix="foo" uri="/WEB-INF/exampletag.tld" %>
<% int num=5; %>
<br>
<pre>
<foo:loop index="i" count="<%=num%>">
body1here: i expr: <%=i%> i property: <jsp:getProperty name="i" property="value" />
  <foo:loop index="j" count="3">
    body2here: j expr: <%=j%>
      i property: <jsp:getProperty name="i" property="value" />
      j property: <jsp:getProperty name="j" property="value" />
    </foo:loop>
  </foo:loop>
</pre>
```

サンプル・タグ・ハンドラ・クラス : ExampleLoopTag.java

次の例は、タグ・ハンドラ・クラス ExampleLoopTag のソース・コードを示しています。次の点に注意してください。

- doStartTag() メソッドは整定数 EVAL_BODY_TAG を戻すため、タグ本体（実際にはループ）が処理されます。
- ループが実行されるたびに、doAfterBody() メソッドによりカウンタ値が増分されます。まだ反復回数が残っている場合は EVAL_BODY_TAG が戻され、最後の反復後は SKIP_BODY が戻されます。

```
package examples;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.util.Hashtable;
import java.io.Writer;
import java.io.IOException;
import oracle.jsp.jml.JmlNumber;

public class ExampleLoopTag
    extends BodyTagSupport
{
    String index;
    int count;
    int i=0;
    JmlNumber ib=new JmlNumber();
```

```

public void setIndex(String index)
{
    this.index=index;
}
public void setCount(String count)
{
    this.count=Integer.parseInt(count);
}

public int doStartTag() throws JspException {
    return EVAL_BODY_TAG;
}

public void doInitBody() throws JspException {
    pageContext.setAttribute(index, ib);
    i++;
    ib.setValue(i);
}

public int doAfterBody() throws JspException {
    try {
        if (i >= count) {
            bodyContent.writeOut(bodyContent.getEnclosingWriter());
            return SKIP_BODY;
        } else
            pageContext.setAttribute(index, ib);
        i++;
        ib.setValue(i);
        return EVAL_BODY_TAG;
    } catch (IOException ex) {
        throw new JspTagException(ex.toString());
    }
}
}

```

サンプル Tag-Extra-Info クラス : ExampleLoopTagTEI.java

次の例は、loop タグで使用されるスクリプト変数を記述する Tag-Extra-Info クラスのソース・コードを示しています。

VariableInfo インスタンスは、次の変数情報を指定するように構築されます。

- 変数名は、index 属性に従って使用されます。
- 変数は oracle.jsp.jml.JmlNumber 型です（これは、完全修飾クラス名として指定する必要があります）。

- 変数は新規に宣言されます。
- 変数の有効範囲は NESTED です。

また、Tag-Extra-Info クラスには、カウント属性が有効かどうか（必ず整数）を判断する `isValid()` メソッドがあります。

```
package examples;

import javax.servlet.jsp.tagext.*;

public class ExampleLoopTagTEI extends TagExtraInfo {
    public VariableInfo[] getVariableInfo(TagData data) {
        return new VariableInfo[]
        {
            new VariableInfo(data.getAttributeString("index"),
                             "oracle.jsp.jml.JmlNumber",
                             true,
                             VariableInfo.NESTED)
        };
    }

    public boolean isValid(TagData data)
    {
        String countStr=data.getAttributeString("count");
        if (countStr!=null)    // for request time case
        {
            try {
                int count=Integer.parseInt(countStr);
            }
            catch (NumberFormatException e)
            {
                return false;
            }
        }
        return true;
    }
}
```

サンプル・タグ・ライブラリ記述ファイル : exampletag.tld

次の例は、タグ・ライブラリのタグ・ライブラリ記述（TLD）ファイルを示しています。この例では、ライブラリは単一のタグ loop のみで構成されています。

この TLD ファイルでは、loop タグに関して次が指定されています。

- examples.ExampleLoopTag はタグ・ハンドラ・クラスです。
- examples.ExampleLoopTagTEI は Tag-Extra-Info クラスです。
- bodycontent 仕様部は JSP で、JSP トランスレータで本体コードを処理して変換する必要があることを意味します。
- 2つの属性 index および count があり、どちらも必須です。count 属性には、要求時 JSP 式を使用できます。

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- a tab library descriptor -->

<taglib>
  <!-- after this the default space is
    "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
  -->

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>simple</shortname>
<!--
  there should be no <urn></urn> here
-->
  <info>
    A simple tab library for the examples
  </info>

  <!-- example tag -->
  <!-- for loop -->
  <tag>
    <name>loop</name>
    <tagclass>examples.ExampleLoopTag</tagclass>
    <teiclass>examples.ExampleLoopTagTEI</teiclass>
    <bodycontent>JSP</bodycontent>
    <info>for loop</info>
    <attribute>
      <name>index</name>
      <required>true</required>
```

```
</attribute>
<attribute>
  <name>count</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
</attribute>
</tag>

</taglib>
```

JSP マークアップ言語（JML）のサンプル・タグ・ライブラリの概要

OracleJSP には JSP マークアップ言語（JML）のサンプル・タグ・ライブラリが用意されており、どの標準 JSP 環境にも移植可能です。他の標準タグ・ライブラリの場合と同様に、JML タグは標準の JSP スクリプトと完全に互換し、すべての JSP ページで使用できます。

JML タグの多くは、Java をよく理解していない JSP 開発者のために、コードの構文を簡素化することを意図しています。XML 変換用（[第 5 章](#)を参照）、Bean のバインドおよび一般ユーティリティ用のタグもあります。

説明するトピックは、次のとおりです。

- [JML タグ・ライブラリの原理](#)
- [JML タグのカテゴリ](#)
- [JML のタグ・ライブラリ記述ファイルと taglib ディレクティブ](#)

JML タグを使用する場合は、次の要件に注意してください。

- ファイル `ojsputil.jar` をインストールし、CLASSPATH に含めます。このファイルは、OracleJSP のインストール時に提供されます。
- タグ・ライブラリ記述ファイル `jml.tld` がアプリケーションとともに配布され、JSP ページの `taglib` ディレクティブで指定した場所に格納されていることを確認します。7-20 ページの「[JML のタグ・ライブラリ記述ファイルと taglib ディレクティブ](#)」を参照してください。

注意：

- OracleJSP には、SQL 機能用のタグ・ライブラリも用意されています。5-22 ページの「[OracleJSP の SQL 用タグ・ライブラリ](#)」を参照してください。
 - リリース 1.1.0.0.0 以前の OracleJSP と JSP 1.1 仕様のリリースでは、OracleJSP では JML タグは Oracle の拡張機能としてのみサポートされていました。（タグ・ライブラリのフレームワークは、JSP 1.1 で JavaServer Pages 仕様に追加されました。）これらのリリースの場合、Oracle 固有の JML タグの処理は、OracleJSP トランスレータに組み込まれていました。この機能は「コンパイル時 JML サポート」と呼ばれます。[付録 C「コンパイル時 JML タグ・サポート」](#)を参照してください。
-

JML タグ・ライブラリの原理

JavaServer Pages のテクノロジーは、主に Java プログラミングの技能を持つ開発者と、主に静的コンテンツ、特に HTML の設計技能を持ち、スクリプトの作成経験が限られている開発者という、2 種類のユーザーを対象としています。

JML タグ・ライブラリは、Java の知識の有無を問わず、ほとんどの Web 開発者がプログラムのフロー制御機能を駆使して JSP アプリケーションを作成できるように設計されています。

このモデルでは、Java 開発者により別個に開発される JavaBeans にビジネス・ロジックが含まれているものとします。

JML タグのカテゴリ

JML タグ・ライブラリは、広範囲の機能セットに対応しています。[表 7-1](#) は、主要な機能カテゴリの概要を示しています。

表 7-1 JML タグの機能のカテゴリ

タグのカテゴリ	タグ	機能
Bean バインド・タグ	useVariable useForm useCookie remove	この種のタグでは、JavaBeans を指定の JSP 有効範囲で宣言するか、宣言を解除します。7-28 ページの「 Bean バインド・タグの説明 」を参照してください。

表 7-1 JML タグの機能のカテゴリ（続き）

タグのカテゴリ	タグ	機能
ロジック / フロー制御タグ	if choose..when..otherwise foreach return flush	この種のタグは、反復ループや条件 ブランチなど、コード・フローを定 義するための簡易構文を提供します。 7-32 ページの「 ロジックおよびフ ロー制御タグの説明 」を参照してく ださい。
XML 変換タグ	transform styleSheet	この種のタグは、XSL のスタイル シートを JSP ページの出力の全体ま たは一部に適用する処理を簡素化し ます。5-10 ページの「 XSL スタイル シート用の JML タグ 」を参照してく ださい。

JML のタグ・ライブラリ記述ファイルと taglib ディレクティブ

JSP 1.1 仕様に準拠するタグ・ライブラリと同様に、JML ライブラリのタグは XML 形式のタグ・ライブラリ記述（TLD）ファイルに指定されています。

この TLD ファイルは、OracleJSP のサンプル・アプリケーションとともに提供されます。このファイルは、JML タグを使用する JSP アプリケーションとともに配布し、JML タグを使用するページの taglib ディレクティブで指定する必要があります。

JML の taglib ディレクティブ

JML タグを使用する JSP ページでは、ファイル位置を識別する標準 URI（universal resource indicator）を指定する TLD ファイルを taglib ディレクティブ内で指定する必要があります。通常、URI 構文は次のようにアプリケーション相対です。

```
<%@ taglib uri="/WEB-INF/jml.tld" prefix="jml" %>
```

また、この例のように TLD ファイルへのフルパスを使用するかわりに、web.xml ファイル内で URI のショートカットを指定し、そのショートカットを taglib ディレクティブで使えます。7-11 ページの「[タグ・ライブラリ用の web.xml の使用](#)」を参照してください。

タグ・ライブラリ記述ファイルの概要は、7-10 ページの「[タグ・ライブラリ記述ファイル](#)」を参照してください。

JML の TLD ファイル・リスト

この項では、OracleJSP リリース 1.1.0.0.0 でサポートされる JML タグ・ライブラリの TLD ファイル全体を示します。

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- a tab library descriptor -->

<taglib>
  <!-- after this the default space is
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd"
  -->

<tlibversion>1.0</tlibversion>
<jspversion>1.1</jspversion>
<shortname>jml</shortname>
<info>
  Oracle's jml tag library.  Not all of the jml
  tag's available in the Oracle JSP environment
  are provided in this library.  No jsp: tags are
  duplicated, some tags are unavailable, and some tags
  have stricter syntax.  No bean expressions are supported.

  The differences are:
  *-jml:call - not available
  * jml:choose - works as documented
  * jml:flush - works as documented
  * jml:for - works as documented
  * jml:foreach - the type attribute is required, otherwise,
    as documented
  *!jml:forward - use jsp:forward
  *!jml:getProperty - use jsp:getProperty
  * jml:if - works as documented
  *!jml:include - use jsp:include
  *-jml:lock - not available
  *!jml:plugin - use jsp:plugin
  * jml:print - the expression to print must be supplied as
    an attribute.  i.e. the tag cannot have a body
  * jml:remove - works as documented
  * jml:return - works as documented
  *-jml:set - not available
  *!jml:setProperty - use jsp:setProperty
  * jml:styleSheet - works as documented
  * jml:transform - works as documented
```

```
*!jml:useBean - use jsp:useBean
* jml:useCookie - works as documented
* jml:useForm - works as documented
* jml:useVariable - works as documented
</info>

<!-- The choose tag -->
<tag>
  <name>choose</name>
  <tagclass>oracle.jsp.jml.tagext.JmlChoose</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>
    The outer tag of a multiple choice logic block,
    choose
      when condition1
      when condition2
      otherwise
    end choose
  </info>
</tag>

<!-- The flush tag -->
<tag>
  <name>flush</name>
  <tagclass>oracle.jsp.jml.tagext.JmlFlush</tagclass>
  <bodycontent>empty</bodycontent>
  <info>
    Flush the current JspWriter
  </info>
</tag>

<!-- The for tag -->
<tag>
  <name>for</name>
  <tagclass>oracle.jsp.jml.tagext.JmlFor</tagclass>
  <teiclass>oracle.jsp.jml.tagext.JmlForTEI</teiclass>
  <bodycontent>JSP</bodycontent>
  <info>
    A simple for loop
  </info>

  <attribute>
    <name>id</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>from</name>
```

```
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
  <name>to</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
</attribute>
</tag>

<!-- The foreach tag -->
<tag>
  <name>foreach</name>
  <tagclass>oracle.jsp.jml.tagext.JmlForeach</tagclass>
  <teiclass>oracle.jsp.jml.tagext.JmlForeachTEI</teiclass>
  <bodycontent>JSP</bodycontent>
  <info>
    A foreach loop for iterating arrays, enumerations,
    and vector's.
  </info>

  <attribute>
    <name>id</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>in</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>type</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>limit</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>

<!-- The if tag -->
<tag>
  <name>if</name>
  <tagclass>oracle.jsp.jml.tagext.JmlIf</tagclass>
  <bodycontent>JSP</bodycontent>
```

```
<info>
  A classic if
</info>

<attribute>
  <name>condition</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
</attribute>
</tag>

<!-- The otherwise tag -->
<tag>
  <name>otherwise</name>
  <tagclass>oracle.jsp.jml.tagext.JmlOtherwise</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>
    (optional) final part of a choose block
  </info>
</tag>

<!-- The print tag -->
<tag>
  <name>print</name>
  <tagclass>oracle.jsp.jml.tagext.JmlPrint</tagclass>
  <bodycontent>empty</bodycontent>
  <info>
    print the expression specified in the eval attribute
  </info>
  <attribute>
    <name>eval</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>

<!-- The remove tag -->
<tag>
  <name>remove</name>
  <tagclass>oracle.jsp.jml.tagext.JmlRemove</tagclass>
  <bodycontent>empty</bodycontent>
  <info>
    remove the specified object from the pageContext
  </info>
  <attribute>
    <name>id</name>
    <required>true</required>
```

```
</attribute>
<attribute>
  <name>scope</name>
  <required>false</required>
</attribute>
</tag>

<!-- The return tag -->
<tag>
  <name>return</name>
  <tagclass>oracle.jsp.jml.tagext.JmlReturn</tagclass>
  <bodycontent>empty</bodycontent>
  <info>
    Skip the rest of the page
  </info>
</tag>

<!-- The styleSheet tag -->
<tag>
  <name>styleSheet</name>
  <tagclass>oracle.jsp.jml.tagext.JmlStyleSheet</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>
    Transform the body of the tag using a stylesheet
  </info>
  <attribute>
    <name>href</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>

<!-- The transform tag -->
<tag>
  <name>transform</name>
  <tagclass>oracle.jsp.jml.tagext.JmlStyleSheet</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>
    Transform the body of the tag using a stylesheet
  </info>
  <attribute>
    <name>href</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

```
<!-- The useCookie tag -->
<tag>
  <name>useCookie</name>
  <tagclass>oracle.jsp.jml.tagext.JmlUseCookie</tagclass>
  <teiclass>oracle.jsp.jml.tagext.JmlUseTEI</teiclass>
  <bodycontent>empty</bodycontent>
  <info>
    create a jml variable and initialize it to a cookie value
  </info>
  <attribute>
    <name>id</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>scope</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>type</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>cookie</name>
    <required>true</required>
  </attribute>
</tag>

<!-- The useForm tag -->
<tag>
  <name>useForm</name>
  <tagclass>oracle.jsp.jml.tagext.JmlUseForm</tagclass>
  <teiclass>oracle.jsp.jml.tagext.JmlUseTEI</teiclass>
  <bodycontent>empty</bodycontent>
  <info>
    create a jml variable and initialize it to a parameter value
  </info>
  <attribute>
    <name>id</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>scope</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>type</name>
    <required>true</required>
  </attribute>
```



```
</attribute>
<attribute>
  <name>param</name>
  <required>true</required>
</attribute>
</tag>

<!-- The useVariable tag -->
<tag>
  <name>useVariable</name>
  <tagclass>oracle.jsp.jml.tagext.JmlUseVariable</tagclass>
  <teiclass>oracle.jsp.jml.tagext.JmlUseTEI</teiclass>
  <bodycontent>empty</bodycontent>
  <info>
    create a jml variable and initialize it to a parameter value
  </info>
  <attribute>
    <name>id</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>scope</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>type</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>value</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>

<!-- The when tag -->
<tag>
  <name>when</name>
  <tagclass>oracle.jsp.jml.tagext.JmlWhen</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>
    one part of a choose block, see choose
  </info>
  <attribute>
    <name>condition</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
```

```
</attribute>
</tag>

</taglib>
```

JSP マークアップ言語（JML）タグの説明

この項では、JSP 1.1 仕様に従って OracleJSP 1.1.0.0.0 ランタイム実装でサポートされる JML タグについて説明します。JML タグは、次の 2 つのカテゴリに分かれています。

- [Bean バインド・タグの説明](#)
- [ロジックおよびフロー制御タグの説明](#)

ここで説明するタグの初歩的な使用例は、9-27 ページの「[JML タグのサンプル — hellouser_jml.jsp](#)」を参照してください。

XML 変換用のタグについては、5-10 ページの「[XSL スタイルシート用の JML タグ](#)」を参照してください。

構文の表記規則と注意事項

タグの説明に含まれる構文では、次の点に注意してください。

- イタリック体は、値または文字列を指定する必要があることを示します。
- オプション属性は大カッコ [...] で囲まれています。
- オプション属性のデフォルト値は、**太字**で示されます。
- 属性の指定方法を示す選択枝は、縦線 | で区切られています。
- 接頭辞「jml:」が使用されています。これは、表記規則によるものですが必須ではありません。taglib ディレクティブでは任意の接頭辞を指定できます。

Bean バインド・タグの説明

この項では、Bean バインド操作に使用される次の JML タグについて説明します。

- [JML の useVariable タグ](#)
- [JML の useForm タグ](#)
- [JML の useCookie タグ](#)
- [JML の remove タグ](#)

JML の useVariable タグ

このタグは、単純な変数を宣言するときに `jsp:useBean` タグのかわりに使用できます。

構文

```
<jml:useVariable id = "beanInstanceName"
                [scope = "page | request | session | application"]
                type = "string | boolean | number | fpnumber"
                [value = "stringLiteral | <%= jspExpression %>"] />
```

属性

- `id` — 宣言する変数の名前を指定します。これは必須属性です。
- `scope` — 変数の継続時間または有効範囲を定義します (`jsp:useBean` タグと同じです)。これはオプション属性で、デフォルト有効範囲は `page` です。
- `type` — 変数の型を指定します (型仕様部は `JmlString`、`JmlBoolean`、`JmlNumber` または `JmlFPNumber` になります)。これは必須属性です。
- `value` — 変数を、文字列リテラルまたは `<%=... %>` 構文で囲まれた JSP の式として、宣言内で直接設定できるようにします。これはオプション属性です。この属性を指定しない場合は、値は最後に設定したとき (既存の場合) またはデフォルト値で初期化されたときそのまま変わりません。この属性を指定すると、この宣言でオブジェクトがインスタンス化されても指定の有効範囲から単に取得されても、常に値が設定されます。

例 次に例を示します。

```
<jml:useVariable id = "isValidUser" type = "boolean" value = "<%= dbConn.isValid() %>" scope = "session" />
```

この例は次の例と等価です。

```
<jsp:useBean id = "isValidUser" class = "oracle.jsp.jml.JmlBoolean" scope = "session" />
<jsp:setProperty name="isValidUser" property="value" value = "<%= dbConn.isValid() %>" />
```

JML の useForm タグ

このタグでは、変数を宣言し、その変数を要求から渡される値に設定する場合に便利な構文を提供します。

構文

```
<jml:useForm id = "beanInstanceName"
            [scope = "page | request | session | application"]
            [type = "string | boolean | number | fpnumber"]
            param = "requestParameterName" />
```

属性

- **id** — 宣言または参照する変数の名前を指定します。これは必須属性です。
- **scope** — 変数の継続時間または有効範囲を定義します (`jsp:useBean` タグと同じです)。これはオプション属性で、デフォルト有効範囲は `page` です。
- **type** — 変数の型を指定します (型仕様部は `JmlString`、`JmlBoolean`、`JmlNumber` または `JmlFPNumber` になります)。これはオプション属性で、デフォルト設定は `string` です。
- **param** — 値が変数設定に使用される要求パラメータの名前を指定します。これは必須属性です。要求パラメータがある場合は、この宣言によって変数が存在するようになるかどうかに関係なく、変数値が常に更新されます。要求パラメータがない場合は、変数値は変更されません。

例 次の例では、`string` 型のセッション変数 `user` を要求パラメータ `user` の値に設定しています。

```
<jml:useForm id = "user" type = "string" param = "user" scope = "session" />
```

この例は次の例と等価です。

```
<jsp:useBean id = "user" class = "oracle.jsp.jml.JmlString" scope = "session" />
<jsp:setProperty name="user" property="value" param = "user" />
```

JML の `useCookie` タグ

このタグでは、変数を宣言し、その変数を `cookie` に含まれる値に設定する場合に便利な構文を提供します。

構文

```
<jml:useCookie id = "beanInstanceName"
               [scope = "page | request | session | application"]
               [type = "string | boolean | number | fpnumber"]
               cookie = "cookieName" />
```

属性

- **id** — 宣言または参照する変数の名前を指定します。これは必須属性です。
- **scope** — 変数の継続時間または有効範囲を定義します。これはオプション属性で、デフォルト有効範囲は `page` です。
- **type** — 変数の型を識別します (型仕様部は `JmlString`、`JmlBoolean`、`JmlNumber` または `JmlFPNumber` となります)。これはオプション属性で、デフォルト設定は `string` です。

- **cookie** — 値がこの変数の設定に使用される cookie の名前を指定します。これは必須属性です。cookie がある場合は、この宣言で変数が存在するようになるかどうかに関係なく、変数値が常に更新されます。cookie がない場合は、変数値は変更されません。

例 次の例では、string 型の要求変数 user を cookie user の値に設定しています。

```
<jml:useCookie id = "user" type = "string" cookie = "user" scope = "request" />
```

この例は次の例と等価です。

```
<jsp:useBean id = "user" class = "oracle.jsp.jml.JmlString" scope = "request" />
<%
    Cookies [] cookies = request.getCookies();
    for (int i = 0; i < cookies.length; i++) {
        if (cookies[i].getName().equals("user")) {
            user.setValue(cookies[i].getValue());
            break;
        }
    }
%>
```

JML の remove タグ

このタグでは、その有効範囲からオブジェクトが削除されます。

構文

```
<jml:remove id = "beanInstanceName"
            [scope = "page | request | session | application" ] />
```

属性

- **id** — 削除する Bean の名前を指定します。これは必須属性です。
- **scope** — これはオプション属性です。指定しない場合は、有効範囲は 1) page、2) request、3) session、4) application の順に検索されます。名前が id と一致した最初のオブジェクトが削除されます。

例 次の例では、セッションの user オブジェクトが削除されます。

```
<jml:remove id = "user" scope = "session" />
```

この例は次の例と等価です。

```
<% session.removeValue("user"); %>
```

ロジックおよびフロー制御タグの説明

この項では、ロジックおよびフロー制御に使用される次の JML タグについて説明します。

- [JML の if タグ](#)
- [JML の choose...when...\[otherwise\] タグ](#)
- [JML の for タグ](#)
- [JML の foreach タグ](#)
- [JML の return タグ](#)
- [JML の flush タグ](#)

これらのタグは、Java に精通していない開発者を対象としており、反復ループや条件ブランチなど、Java のロジックおよびフロー制御構文のかわりに使用できます。

JML の if タグ

このタグでは、単一の条件文が評価されます。条件が `true` であれば、`if` タグの本体が実行されます。

構文

```
<jml:if condition = "<%= jspExpression %>" >  
    ...body of if tag (executed if the condition is true)...  
</jml:if>
```

属性

- `condition` — 評価する条件式を指定します。これは必須属性です。

例 次の E-Commerce の例では、ユーザーのショッピング・カートからの情報が表示されます。このコードでは、現在の T シャツの注文を保持する変数が空かどうかチェックされます。空でなければ、ユーザーが注文したサイズが表示されます。`currTS` は `JmlString` 型とします。

```
<jml:if condition = "<%= !currTS.isEmpty() %>" >  
    <S>(size: <%= currTS.getValue().toUpperCase() %>)</S>&nbsp;    
</jml:if>
```

JML の choose...when...[otherwise] タグ

choose タグは when および otherwise タグに対応付けて、複数の条件文を提供します。

choose タグの本体には 1 つ以上の when タグが含まれ、それぞれの when タグが 1 つの条件を表します。評価結果が true となる最初の when 条件の場合、その when タグの本体が実行されます。（最大 1 つの when 本体が実行されます。）

評価結果が true となる when 条件がなく、また、オプションの otherwise タグが指定されていれば、otherwise タグの本体が実行されます。

構文

```
<jml:choose>
  <jml:when condition = "<%= jspExpression %">" >
    ...body of 1st when tag (executed if the condition is true)...
  </jml:when>
  ...
  [...optional additional when tags...]
  [ <jml:otherwise>
    ...body of otherwise tag (executed if all when conditions false)...
  </jml:otherwise> ]
</jml:choose>
```

属性 when タグでは、次の属性が使用されます（choose および otherwise タグには属性はありません）。

- condition — 評価する条件式を指定します。これは必須属性です。

例 次の E-Commerce の例では、ユーザーのショッピング・カートからの情報が表示されます。このコードでは、注文の有無がチェックされます。注文があれば現在の注文が表示され、注文がなければ、再度ショッピングするかどうかをユーザーに確認するプロンプトが表示されます。（この例では、現在の注文を表示するコードは省略されています。） `orderedItem` は `JmlBoolean` 型とします。

```
<jml:choose>
  <jml:when condition = "<%= orderedItem.getValue() %">" >
    You have changed your order:
    -- output the current order --
  </jml:when>
  <jml:otherwise>
    Are you sure we can't interest you in something, cheapskate?
  </jml:otherwise>
</jml:choose>
```

JML の for タグ

このタグでは、Java の for ループと同様に、ループを反復できます。

id 属性は、現行範囲の要素の値を含む `java.lang.Integer` 型のローカル・ループ変数です。範囲は、from 属性で表される値から始まり、to 属性で表される値を超えるまで、ループ本体が実行されるたびに増分されます。

範囲の横断後は、制御は for 終了タグに続く最初の文に移ります。

注意： 降順の範囲はサポートされません。from 値は to 値以下にする必要があります。

構文

```
<jml:for id = "loopVariable"
    from = "<%= jspExpression %>"
    to = "<%= jspExpression %>" >
    ...body of for tag (executed once at each value of range, inclusive)...
</jml:for>
```

属性

- id — 範囲内の現在の値を保持するループ変数の名前を指定します。これは `java.lang.Integer` 型の値で、タグの本体内でのみ使用できます。これは必須属性です。
- from — 範囲の開始を指定します。これは、Java の `int` 型の値に評価される必要のある式です。これは必須属性です。
- to — 範囲の終わりを指定します。これは、Java の `int` 型の値に評価される必要のある式です。これは必須属性です。

例 次の例では、「Hello World」が1つずつ下位のヘッダー（H1、H2、H3、H4、H5）として繰り返し出力されます。

```
<jml:for id="i" from="<%= 1 %>" to="<%= 5 %>" >
    <H<%=i%>>
        Hello World!
    </H<%=i%>>
</jml:for>
```


JML の foreach タグ

このタグでは、同型の値セットを反復できます。

タグ本体は、そのセットの要素ごとに一度ずつ実行されます。(セットが空であれば、本体は実行されません。)

id 属性は、現在のセットの要素値を含むローカル・ループ変数です。型は、type 属性で指定します。(該当する場合は、セットの要素の型と一致する型を指定する必要があります。)

現在、このタグでは、次のタイプのデータ構造の反復がサポートされます。

- Java 配列
- java.util.Enumeration
- java.util.Vector

構文

```
<jml:foreach id = "loopVariable"
    in = "<%= jspExpression %>"
    limit = "<%= jspExpression %>"
    type = "package.class" >
    ...body of foreach tag (executes once for each element in data structure)...
</jml:foreach>
```

属性

- id — 各反復ステップで現在の要素値を保持するループ変数の名前を指定します。この属性は、タグ本体内でのみ使用できます。型は、type 属性で指定した型と同じです。id は必須属性です。
- in — Java 配列、Enumeration オブジェクトまたは Vector オブジェクトに評価される JSP の式を指定します。これは必須属性です。
- limit — セット内の要素数に関係なく、最大反復数を定義する Java の int 型の値に評価される JSP の式を指定します。これは必須属性です。
- type — ループ変数の型を指定します。該当する場合は、セットの要素の型と一致する型を指定する必要があります。これは必須属性です。

例 次の例は、要求パラメータに対して反復されます。

```
<jml:foreach id="name" in="<%= request.getParameterNames() %>" type="java.lang.String" >
    Parameter: <%= name %>
    Value: <%= request.getParameter(name) %> <br>
</jml:foreach>
```

また、複数の値を取るパラメータを処理する必要がある場合は、次のようになります。

```
<jml:foreach id="name" in="<%= request.getParameterNames() %>" type="java.lang.String" >
  Parameter: <%= name %>
  Value: <jml:foreach id="val" in="<%=request.getParameterValues(name)%>"
        type="java.lang.String" >
    <%= val %> :
  </jml:foreach>
<br>
</jml:foreach>
```

JML の return タグ

このタグに達すると、ページから実行が戻り、それ以上処理されません。

構文

```
<jml:return />
```

属性

なし。

例 次の例では、タイマーが時間切れになると、ページが処理されずに戻されます。

```
<jml:if condition="<%= timer.isExpired() %>" >
  You did not complete in time!
  <jml:return />
</jml:if>
```

JML の flush タグ

このタグでは、ページ・バッファの現在の内容がクライアントに書き戻されます。

これが適用されるのは、ページがバッファリングされる場合のみで、他の場合には無効です。

構文

```
<jml:flush />
```

属性

なし。

例 次の例では、現在のページのコンテンツがフラッシュされてから、高コストの操作が実行されます。

```
<jml:flush />  
<% myBean.expensiveOperation(out); %>
```

OracleJSP の NLS サポート

OracleJSP には、Sun Microsystems の『JavaServer Pages Specification, Version 1.1』に準拠する標準の各国語サポート（NLS）機能に加えて、マルチバイト・パラメータのコード化がサポートされないサーブレット環境向けの拡張サポート機能も用意されています。

ローカライズされるコンテンツに対する標準 Java サポートは、テキストの統一内部表現に Unicode 2.0 を使用することに依存しています。Unicode は、代替キャラクタ・セットへの変換用のベース・キャラクタ・セットとして使用されます。

この章では、OracleJSP による NLS サポート方法の主要な側面について説明します。説明するトピックは、次のとおりです（その他に、将来のリリースで取り上げられるトピックがあります）。

- [page ディレクティブでのコンテンツ型の設定](#)
- [コンテンツ型の動的設定](#)
- [マルチバイト・パラメータのコード化に対する OracleJSP の拡張サポート](#)

page ディレクティブでのコンテンツ型の設定

page ディレクティブの `contentType` パラメータを使用すると、JSP ページの MIME タイプと、オプションで JSP ページの文字コード化を設定できます。MIME タイプは、実行時に HTTP 応答に適用されます。文字コード化を設定すると、変換時のページ・テキストと実行時の HTTP 応答の両方に適用されます。

page ディレクティブには、次の構文を使用します。

```
<%@ page ... contentType="TYPE"; charset=character_set" ... %>
```

また、デフォルトのキャラクタ・セットを使用し、MIME タイプを設定するには、次の構文を使用します。

```
<%@ page ... contentType="TYPE" ... %>
```

TYPE は IANA (Internet Assigned Numbers Authority) の MIME タイプで、`character_set` は IANA のキャラクタ・セットです。(キャラクタ・セットを指定する場合、セミコロンの後の空白はオプションです。)

次に例を示します。

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
```

または

```
<%@ page language="java" contentType="text/html" %>
```

デフォルトの MIME タイプは `text/html` です。IANA は、MIME タイプのレジストリを次のサイトでメンテナンスしています。

<ftp://venera.isi.edu/in-notes/iana/assignments/media-types/media-types>

デフォルトの文字コードは ISO-8859-1 (Latin-1 と呼ばれます) です。IANA は、文字コードのレジストリを次のサイトでメンテナンスしています (指定の推奨 MIME 名が表示されている場合は、その MIME 名を使用してください)。

<ftp://venera.isi.edu/in-notes/iana/assignments/character-sets>

(Java と Web ブラウザでサポートされるキャラクタ・セットを使用している限り、IANA キャラクタ・セットを使用するための JSP 要件はありませんが、IANA のサイトには最も一般的なキャラクタ・セットのリストがあります。指定の推奨 MIME 名を使用することをお勧めします。)

page ディレクティブのパラメータは静的です。実行中に、ページで応答には異なる設定が必要であることが検出された場合は、次のどちらかの処理を実行できます。

- サブレット応答オブジェクト API を使用して、実行中にコンテンツ型を設定します。8-3 ページの「[コンテンツ型の動的設定](#)」を参照してください。
- 要求を別の JSP ページまたはサブレットに転送します。

注意：

- contentType を設定する page ディレクティブは、できるだけ JSP ページの先頭付近で使用する必要があります。
- ISO-8859-1 以外のキャラクタ・セットで記述された JSP ページでは、page ディレクティブで適切なキャラクタ・セットを設定する必要があります。ページは変換中に設定を認識するため、キャラクタ・セットを動的には設定できません。動的設定は実行時専用です。
- JSP 1.1 仕様では、JSP ページがそのコンテンツ配信時と同じキャラクタ・セットで記述されることを想定しています。
- このマニュアルでは、分かりやすく説明するために、ページのテキスト、要求パラメータおよび応答パラメータにすべて同じコード化を使用する典型的なケースを想定しています（ただし、他の使用例も技術的には可能です）。要求パラメータのコード化はブラウザにより制御されますが、Netscape ブラウザと Internet Explorer には応答パラメータに指定した設定が使用されます。

コンテンツ型の動的設定

HTTP 応答に該当するコンテンツ型が実行時まで判明しない状況では、コンテンツ型を JSP ページ内で動的に設定できます。このため、標準 `javax.servlet.ServletResponse` インタフェースでは、次のメソッドを指定します。

```
public void setContentType(java.lang.String contentType)
```

JSP ページの暗黙的 response オブジェクトは `javax.servlet.http.HttpServletResponse` インスタンスです。この `HttpServletResponse` インタフェースにより `ServletResponse` インタフェースが拡張されます。

page ディレクティブでの contentType 設定と同様に、`setContentType()` メソッドの入力には MIME タイプのみ、またはキャラクタ・セットと MIME タイプの両方を使用できます。次に例を示します。

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

または

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

page ディレクティブと同様に、デフォルトの MIME タイプは `text/html` で、デフォルトの文字コードは ISO-8859-1 です。

このメソッドは、変換時の JSP ページのテキストの解析では無効です。変換時に特定のキャラクタ・セットが必要な場合は、そのキャラクタ・セットを `page` ディレクティブで指定する必要があります。8-2 ページの「[page ディレクティブでのコンテンツ型の設定](#)」を参照してください。

使用上の重要な注意事項は、次のとおりです。

- `setContentType()` メソッドを使用する場合、JSP ページはアンバッファリングできません。（デフォルトではバッファリングされます。`page` ディレクティブで `buffer="none"` に設定しないでください。）
- `setContentType()` コールは、ページ内の、ブラウザへの出力や `jsp:include` コマンド（JSP バッファをブラウザにフラッシュするコマンド）の前に挿入する必要があります。
- サブレット 2.2 環境では、`response` オブジェクトには、指定されたロケールに基づいてデフォルトのキャラクタ・セットを設定し、以前のキャラクタ・セットをオーバーライドする `setLocale()` メソッドがあります。たとえば、次のメソッドのコールにより、キャラクタ・セット `Shift_JIS` が設定されます。

```
response.setLocale(new Locale("ja", "JP"));
```

マルチバイト・パラメータのコード化に対する OracleJSP の拡張サポート

要求パラメータの文字コード化は、HTTP 仕様では十分に定義されていません。ほとんどのサブレット・コンテナでは、その解析にサブレットのデフォルトの文字コード化 `ISO-8859-1` を使用する必要があります。

このような環境では、サブレット・コンテナはマルチバイトの要求パラメータと `Bean` のプロパティ設定をコード化できないため、OracleJSP は `translate_params` 構成パラメータを介して拡張サポートを提供します。

JSP ページで等価コードを使用することもできます。これは、Oracle Servlet 環境で必要です。

重要： 次の場合には、`translate_params` フラグを有効化しないでください。

- サブレット・コンテナでマルチバイト・パラメータのコード化自体が適切に処理される場合。この状況で `translate_params` を `true` に設定すると、不正な結果となります。ただし、このマニュアルの製作時点では、Apache/JServ、JSWDK および Tomcat では、いずれもマルチバイト・パラメータのコード化が正常に処理されないことが判明しています。
 - 要求パラメータに JSP の `page` ディレクティブまたは `setContentType()` メソッドで応答用に指定されているのとは異なるコード化が使用される場合。
 - `translate_params` の実行内容と等価の回避機能を持つコードがすでに JSP ページ内に存在している場合。(8-6 ページの「[translate_params 構成パラメータの等価コード](#)」を参照してください。)
-

非マルチバイト・サブレット・コンテナのオーバーライドにおける `translate_params` の効果

マルチバイトの要求パラメータと Bean のプロパティ設定をコード化できないサブレット・コンテナをオーバーライドするには、`translate_params` を `true` に設定します。(この OracleJSP 構成パラメータの設定方法については、A-23 ページの「[OracleJSP の構成パラメータ設定](#)」を参照してください。)

このフラグを有効化すると、OracleJSP では要求パラメータと Bean のプロパティ設定が、`response.getCharacterEncoding()` メソッドで指定されたとおり、`response` オブジェクトのキャラクタ・セットに基づいてコード化されます。

`translate_params` フラグは、パラメータ名と値、特に次の要素に影響します。

- 要求オブジェクトの `getParameter()` メソッドの出力
- 要求オブジェクトの `getParameterValues()` メソッドの出力
- 要求オブジェクトの `getParameterNames()` メソッドの出力
- Bean のプロパティ値の `jsp:setProperty` 設定

translate_params 構成パラメータの等価コード

translate_params 構成パラメータはランタイム構成パラメータで、Oracle Servlet Engine 環境では設定できません。(変換時の構成は、ojspc コマンドライン・オプションを介して OSE 環境用に設定できます。ランタイム構成パラメータの等価コードはありません。)

このため、または他の理由から、次のように、JSP ページのスクリプトレット・コードを介して実装できる等価機能を知っていると役立ちます。

```
<%@ page contentType="text/html; charset=EUC-JP" %>
...
String paramName="XXYYZZ";          // where XXYYZZ is a multibyte string
paramName =
    new String(paramName.getBytes(response.getCharacterEncoding()), "ISO8859_1");
String paramValue = request.getParameter(paramName);
paramValue= new String(paramValue.getBytes("ISO8859_1"), "EUC-JP");
...
```

このコードでは、次の処理が実行されます。

- 検索するパラメータ名として XXYYZZ が設定されます。(XX、YY および ZZ は3つの日本語文字とします。)
- パラメータ名が、サーブレット・コンテナで解析できるように、そのキャラクタ・セット ISO-8859-1 にコード化されます。(最初に、要求オブジェクトの文字コードを使用して、パラメータ名を表すバイト配列が作成されます。)
- パラメータ名に一致する値が検索され、要求オブジェクトからパラメータ値が取得されます。(要求オブジェクト内のパラメータ名にも ISO-8859-1 コード化が使用されているため、一致する値を検索できます。)
- パラメータ値が EUC-JP にコード化されて、処理が継続されるか、ブラウザに出力されます。

translate_params の有効化に依存する NLS のサンプルと、translate_params 設定に依存しないように等価コードを含む NLS サンプルについては、次の2つの項を参照してください。

translate_params に依存する NLS のサンプル

次のサンプルでは、日本語のユーザー名を使用でき、その名前がブラウザに正常に出力されます。マルチバイトの要求パラメータをコード化できないサーブレット環境では、このサンプルは OracleJSP の構成設定 translate_params=true に依存します。

XXYY はパラメータ名（日本語の「ユーザー名」の等価値）、AABB はデフォルト値（日本語の値）とします。

(translate_params 機能の等価の、translate_params 設定に依存しないコードを含むサンプルについては、次の項を参照してください。)

```
<%@ page contentType="text/html; charset=EUC-JP" %>

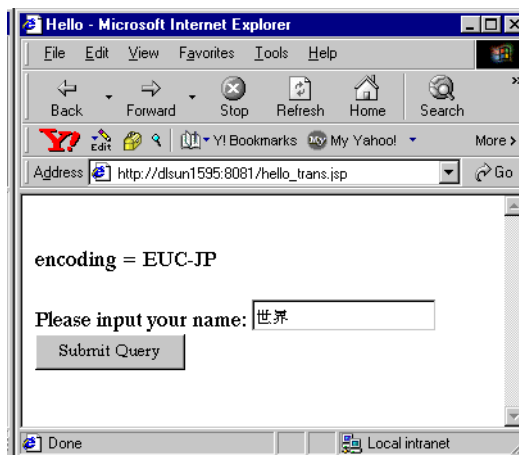
<HTML>
<HEAD>
<TITLE>Hello</TITLE></HEAD>
<BODY>
<%
    //charset is as specified in page directive (EUC-JP)
    String charset = response.getCharacterEncoding();
%>
    <BR> encoding = <%= charset %> <BR>

<%

String paramValue = request.getParameter("XXYY");

if (paramValue == null || paramValue.length() == 0) { %>
    <FORM METHOD="GET">
        Please input your name: <INPUT TYPE="TEXT" NAME="XXYY" value="AABB" size=20>
    <BR>
        <INPUT TYPE="SUBMIT">
    </FORM>
<% }
else
{ %>
    <H1> Hello, <%= paramValue %> </H1>
<% } %>
</BODY>
</HTML>
```

次にサンプル入力を示します。



サンプル出力は次のとおりです。



translate_params に依存しない NLS のサンプル

次のサンプルでは、前述のサンプルと同様に日本語のユーザー名を使用でき、その名前がブラウザに正常に出力されます。ただし、このサンプルには translate_params 機能の等価コードが含まれているため、translate_params 設定に依存しません。

重要： translate_params の等価コードを使用する場合は、translate_params フラグを有効化しないでください。このフラグを有効化すると不正な結果となります。（translate_params フラグがサポートされない OSE 環境では問題になりません。）

XXYY はパラメータ名（日本語の「ユーザー名」の等価値）、AABB はデフォルト値（日本語の値）とします。

このサンプル内の重要なコードについては、8-6 ページの「[translate_params 構成パラメータの等価コード](#)」を参照してください。

```
<%@ page contentType="text/html; charset=EUC-JP" %>

<HTML>
<HEAD>
<TITLE>Hello</TITLE></HEAD>
<BODY>
<%
    //charset is as specified in page directive (EUC-JP)
    String charset = response.getCharacterEncoding();
%>
    <BR> encoding = <%= charset %> <BR>
<%
String paramName = "XXYY";

paramName = new String(paramName.getBytes(charset), "ISO8859_1");

String paramValue = request.getParameter(paramName);

if (paramValue == null || paramValue.length() == 0) { %>
    <FORM METHOD="GET">
        Please input your name: <INPUT TYPE="TEXT" NAME="XXYY" value="AABB" size=20>
    <BR>
        <INPUT TYPE="SUBMIT">
    </FORM>
<% }
else
{
```

```
    paramValue= new String(paramValue.getBytes("ISO8859_1"), "EUC-JP"); %>
    <H1> Hello, <%= paramValue %> </H1>
<% } %>
</BODY>
</HTML>
```

サンプル・アプリケーション

この章では、JSP ページと JSP ページで使用する JavaBeans（該当する場合）について、様々なサンプル・コードを次のカテゴリに分けて説明します。

- [基本的なサンプル](#)
- [JDBC のサンプル](#)
- [Database-Access JavaBeans のサンプル](#)
- [カスタム・タグのサンプル](#)
- [Oracle 固有のプログラミング拡張機能のサンプル](#)
- [サーブレット 2.0 環境に globals.jsa を使用するサンプル](#)

基本的なサンプル

この項では、基本的な、Oracle JML のデータ型の使用例を示す JSP のサンプルについて説明します。ここでは、初歩的な「hello」のサンプル、JavaBeans の使用方法を示すサンプルおよび具体的なショッピング・カートの例を示します。提供されるサンプルは、次のとおりです。

- [hello ページ — hellouser.jsp](#)
- [usebean ページ — usebean.jsp](#)
- [ショッピング・カート・ページ — cart.jsp](#)

これらの例では標準データ型が代用されている場合がありますが、JML のデータ型には多数のメリットがあります。5-2 ページの「[JML の拡張データ型](#)」を参照してください。JML データ型は、他の JSP 環境にも移植可能です。

日本語環境では、Shift_JIS、EUC-JP などのキャラクタ・セットを指定する必要があります。たとえば、Shift_JIS の場合、以下の 1 行を始めに記述します。

```
<%@page Content-Type="text/html;charset=Shift_JIS"%>
```

hello ページ — hellouser.jsp

このサンプルは、初歩的な JSP の「hello」ページです。ユーザーには、ユーザー名の入力フォームが表示されます。名前を送ると、JSP ページの一番上にその名前を含むフォームが再表示されます。

```
<%-----
    Copyright (c) 1999, Oracle Corporation. All rights reserved.
-----%>

<%@page Content-Type="text/html;charset=Shift_JIS"%>
<%@page session="false" %>

<jsp:useBean id="name" class="oracle.jsp.jml.JmlString" scope="request" >
    <jsp:setProperty name="name" property="value" param="newName" />
</jsp:useBean>

<HTML>
<HEAD>
<TITLE>
Hello User
</TITLE>
</HEAD>

<BODY>
```



```

<% if (!name.isEmpty()) { %>
<H3>Welcome <%= name.getValue() %></H3>
<% } %>

<P>
Enter your Name:
<FORM METHOD=get>
<INPUT TYPE=TEXT name=newName size = 20><br>
<INPUT TYPE=SUBMIT VALUE="Submit name">
</FORM>

</BODY>
</HTML>

```

usebean ページ — usebean.jsp

このページでは、単純な JavaBeans である NameBean を使用して、jsp:useBean タグの使用方法を示しています。ページのコード以外に、Bean のコードも含まれています。

usebean.jsp のコード

```

<%-----
    Copyright (c) 1999, Oracle Corporation. All rights reserved.
-----%>

<%@page Content-Type="text/html;charset=Shift_JIS"%>
<%@ page import="beans.NameBean" %>

<jsp:useBean id="pageBean" class="beans.NameBean" scope="page" />
<jsp:setProperty name="pageBean" property="*" />

<jsp:useBean id="sessionBean" class="beans.NameBean" scope="session" />
<jsp:setProperty name="sessionBean" property="*" />

<HTML>
<HEAD> <TITLE> The UseBean JSP </TITLE> </HEAD>
<BODY BGCOLOR=white>

<H3> Welcome to the UseBean JSP </H3>
<P><B>Page bean: </B>
<% if (pageBean.getNewName().equals("")) { %>
    I don't know you.
<% } else { %>
    Hello <%= pageBean.getNewName() %> !
<% } %>

```

```
<P><B>Session bean: </B>
<% if (sessionBean.getNewName().equals("")) { %>
    I don't know you either.
<% } else {
    if ((request.getParameter("newName") == null) ||
        (request.getParameter("newName").equals(""))) { %>
        Aha, I remember you.
    } %>
    You are <%= sessionBean.getNewName() %>.
<% } %>

<P>May we have your name?
<FORM METHOD=get>
<INPUT TYPE=TEXT name=newName size = 20>
<INPUT TYPE=SUBMIT VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

NameBean.java のコード

```
package beans;

public class NameBean {

    String newName="";

    public void NameBean() { }

    public String getNewName() {
        return newName;
    }
    public void setNewName(String newName) {
        this.newName = newName;
    }
}
```

ショッピング・カート・ページー cart.jsp

このサンプルは、セッション状態を使用してショッピング・カートをメンテナンスする方法を示しています。ユーザーがTシャツまたはスウェットシャツを選択して注文すると、その注文が再表示されます。ショッピングを続けて注文が変更されると、その注文がページに再表示され、該当する場合は前回の選択項目に取消し線が引かれます。

cart.jsp ファイルは主ソース・ファイルであり、index.jsp を参照します。ここでは、両方のページのコードを示します。

```

<%
    Copyright (c) 1999-2000, Oracle Corporation. All rights reserved.
-----%>
<%@page Content-Type="text/html; charset=Shift_JIS"%>
<jsp:useBean id="currSS" scope="session" class="oracle.jsp.jml.JmlString" />
<jsp:useBean id="currTS" scope="session" class="oracle.jsp.jml.JmlString" />

<HTML>

<HEAD>
    <TITLE>Java Store</TITLE>
</HEAD>

<BODY BACKGROUND=images/bg.gif BGCOLOR=#FFFFFF>

<jsp:useBean id="sweatShirtSize" scope="page" class="oracle.jsp.jml.JmlString" >
    <jsp:setProperty name="sweatShirtSize" property="value" param="SS" />
</jsp:useBean>

<jsp:useBean id="tshirtSize" scope="page" class="oracle.jsp.jml.JmlString" >
    <jsp:setProperty name="tshirtSize" property="value" param="TS" />
</jsp:useBean>

<jsp:useBean id="orderedSweatshirt" scope="page" class="oracle.jsp.jml.JmlBoolean" >
    <jsp:setProperty name="orderedSweatshirt" property="value"
        value= '<%= !(sweatShirtSize.isEmpty() ||
sweatShirtSize.getValue().equals("none")) %>' />
</jsp:useBean>

<jsp:useBean id="orderedTShirt" scope="page" class="oracle.jsp.jml.JmlBoolean" >
    <jsp:setProperty name="orderedTShirt" property="value"
        value= '<%= !(tshirtSize.isEmpty() || tshirtSize.getValue().equals("none")) %>'
/>
</jsp:useBean>

<P>
<TABLE BORDER=0 CELLPADDING=0 CELLSPACING=0 WIDTH=100% HEIGHT=553>
    <TR>
        <TD WIDTH=33% HEIGHT=61>&nbsp;</TD>
        <TD WIDTH=67% HEIGHT=61>&nbsp;</TD>
    </TR>
    <TR>
        <TD WIDTH=33% HEIGHT=246>&nbsp;</TD>
        <TD WIDTH=67% HEIGHT=246 VALIGN=TOP BGCOLOR=#FFFFFF>
            <% if (orderedSweatshirt.getValue() || orderedTShirt.getValue()) { %>
                Thank you for selecting our fine JSP Wearables!<P>
            <% } %>
        </TD>
    </TR>
</TABLE>

```

```
<% if (!currSS.isEmpty() || !currTS.isEmpty()) { %>
You have changed your order:
  <UL>
    <% if (orderedSweatshirt.getValue()) { %>
      <LI>1 Sweatshirt
      <% if (!currSS.isEmpty()) { %>
        <S>(size: <%= currSS.getValue().toUpperCase() %>)</S>&nbsp;
        <% } %>
        (size: <%= sweatShirtSize.getValue().toUpperCase() %> )
      <% } else if (!currSS.isEmpty()) { %>
        <LI><S>1 Sweatshirt (size: <%= currSS.getValue().toUpperCase()
          %>)</S>
        <% } %>

        <% if (orderedTShirt.getValue()) { %>
          <LI>1 Tshirt
          <% if (!currTS.isEmpty()) { %>
            <S>(size: <%= currTS.getValue().toUpperCase() %>)</S>&nbsp;
            <% } %>
            (size: <%= tshirtSize.getValue().toUpperCase() %>)
          <% } else if (!currTS.isEmpty()) { %>
            <LI><S>1 Tshirt (size: <%= currTS.getValue().toUpperCase()
              %>)</S>
            <% } %>
          </UL>
        <% } else { %>
You have selected:
        <UL>
          <% if (orderedSweatshirt.getValue()) { %>
            <LI>1 Sweatshirt (size: <%= sweatShirtSize.getValue().toUpperCase()
              %>)
            <% } %>

            <% if (orderedTShirt.getValue()) { %>
              <LI>1 Tshirt (size: <%= tshirtSize.getValue().toUpperCase() %>)
              <% } %>
            </UL>
          <% } %>
        <% } else { %>
          Are you sure we can't interest you in something?
        <% } %>

      <CENTER>
        <FORM ACTION="index.jsp" METHOD="GET"
          ENCTYPE="application/x-www-form-urlencoded">
```

```

        <INPUT TYPE="IMAGE" SRC="images/shop_again.gif" WIDTH="91" HEIGHT="30"
            BORDER="0">
        </FORM>
    </CENTER>
</TD></TR>
</TABLE>

</BODY>

</HTML>

<%
if (orderedSweatshirt.getValue()) {
    currSS.setValue(sweatShirtSize.getValue());
} else {
    currSS.setValue("");
}

if (orderedTShirt.getValue()) {
    currTS.setValue(tshirtSize.getValue());
} else {
    currTS.setValue("");
}
%>

```

index.jsp のコード

```

<%-----
    Copyright (c) 1999-2000, Oracle Corporation. All rights reserved.
-----%>

<%@page Content-Type="text/html; charset=Shift_JIS"%>
<jsp:useBean id="currSS" scope="session" class="oracle.jsp.jml.JmlString" />
<jsp:useBean id="currTS" scope="session" class="oracle.jsp.jml.JmlString" />

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>

<HEAD>
    <TITLE>untitled</TITLE>
</HEAD>

<BODY BACKGROUND="images/bg.gif" BGCOLOR="#FFFFFF">

<FORM ACTION="cart.jsp" METHOD="POST" ENCTYPE="application/x-www-form-urlencoded">
<P>
<TABLE BORDER="0" CELLPADDING="0" CELLSPACING="0" WIDTH="100%" HEIGHT="553">

```

```
<TR>
  <TD WIDTH="33%" HEIGHT="61">&nbsp;</TD>
  <TD WIDTH="67%" HEIGHT="61">&nbsp;</TD>
</TR>
<TR>
  <TD WIDTH="33%" HEIGHT="246">&nbsp;</TD>
  <TD WIDTH="67%" HEIGHT="246" VALIGN="TOP" BGCOLOR="#FFFFFF">
    <TABLE BORDER="0" CELLPADDING="0" CELLSPACING="0" WIDTH="81%">
      <TR>
        <TD WIDTH="100%" BGCOLOR="#CCFFFF">
          <H4>JSP Wearables
        </TD>
      </TR>
    </TABLE>
  <TD WIDTH="100%" BGCOLOR="#FFFFFF">

    <BLOCKQUOTE>
    Sweatshirt
    <SPACER TYPE="HORIZONTAL" SIZE="10">($24.95)<BR>
    <SPACER TYPE="HORIZONTAL" SIZE="30">
      <INPUT TYPE="RADIO" NAME="SS" VALUE="xl"
        <%= currSS.getValue().equals("xl") ? "CHECKED" : "" %> >XL
    <SPACER TYPE="HORIZONTAL" SIZE="10">
      <INPUT TYPE="RADIO" NAME="SS" VALUE="l" <%= currSS.getValue().equals("l")
        ? "CHECKED" : "" %> >L
    <SPACER TYPE="HORIZONTAL" SIZE="10">
      <INPUT TYPE="RADIO" NAME="SS" VALUE="m" <%= currSS.getValue().equals("m")
        ? "CHECKED" : "" %> >M
    <SPACER TYPE="HORIZONTAL" SIZE="10">
      <INPUT TYPE="RADIO" NAME="SS" VALUE="s" <%= currSS.getValue().equals("s")
        ? "CHECKED" : "" %> >S
    <SPACER TYPE="HORIZONTAL" SIZE="10">
      <INPUT TYPE="RADIO" NAME="SS" VALUE="xs"
        <%= currSS.getValue().equals("xs") ? "CHECKED" : "" %> >XS
    <SPACER TYPE="HORIZONTAL" SIZE="10">
      <INPUT TYPE="RADIO" NAME="SS" VALUE="none"
        <%= currSS.getValue().equals("none") || currSS.isEmpty() ?
          "CHECKED" : "" %> >NONE
    <BR>
    <BR>
    T-Shirt<SPACER TYPE="HORIZONTAL" SIZE="10"> (14.95)<BR>
    <SPACER TYPE="HORIZONTAL" SIZE="30">
      <INPUT TYPE="RADIO" NAME="TS" VALUE="xl"
        <%= currTS.getValue().equals("xl") ? "CHECKED" : "" %> >XL
    <SPACER TYPE="HORIZONTAL" SIZE="10">
      <INPUT TYPE="RADIO" NAME="TS" VALUE="l" <%= currTS.getValue().equals("l")
        ? "CHECKED" : "" %> >L
```

```

<SPACER TYPE="HORIZONTAL" SIZE="10">
<INPUT TYPE="RADIO" NAME="TS" VALUE="m" <%= currTS.getValue().equals("m")
? "CHECKED" : "" %> >M
<SPACER TYPE="HORIZONTAL" SIZE="10">
<INPUT TYPE="RADIO" NAME="TS" VALUE="s" <%= currTS.getValue().equals("s")
? "CHECKED" : "" %> >S
<SPACER TYPE="HORIZONTAL" SIZE="10">
<INPUT TYPE="RADIO" NAME="TS" VALUE="xs"
<%= currTS.getValue().equals("xs") ? "CHECKED" : "" %> >XS
<SPACER TYPE="HORIZONTAL" SIZE="10">
<INPUT TYPE="RADIO" NAME="TS" VALUE="none"
<%= currTS.getValue().equals("none") || currTS.isEmpty() ?
"CHECKED" : "" %> >NONE
</BLOCKQUOTE>
</TD>
</TR>
<TR>
<TD WIDTH="100%">
<DIV ALIGN="RIGHT">
<P><INPUT TYPE="IMAGE" SRC="images/addtobkt.gif" WIDTH="103" HEIGHT="22"
ALIGN="BOTTOM" BORDER="0">
</DIV>
</TD>
</TR>
</TABLE>
</TD>
</TR>
</TABLE>

</FORM>

</BODY>

</HTML>

```

JDBC のサンプル

この項の例では、JDBC を使用してデータベースの問合せを実行しています。ほとんどの部分で標準 JDBC 機能を使用していますが、接続キャッシュの例では Oracle 固有の接続キャッシュの実装を使用しています。提供される例は、次のとおりです。

- [単純な問合せ — SimpleQuery.jsp](#)
- [ユーザー指定の問合せ — JDBCQuery.jsp](#)

- [問合せ Bean を使用した問合せ — UseHtmlQueryBean.jsp](#)
- [接続キャッシュ — ConnCache3.jsp および ConnCache1.jsp](#)

Oracle JDBC 全般と Oracle JDBC の接続キャッシュの実装については、『Oracle8i 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。

単純な問合せ — SimpleQuery.jsp

このページでは、scott.emp 表の単純な問合せを実行し、従業員とその給与額を HTML 表に（従業員名順で）出力しています。

```
<%@ page import="java.sql.*" %>

<!-------
 * This is a basic JavaServer Page that does a JDBC query on the
 * emp table in schema scott and outputs the result in an html table.
 *
-----!>

<HTML>
  <HEAD>
    <TITLE>
      SimpleQuery JSP
    </TITLE>
  </HEAD>
  <BODY BGCOLOR=EOFFFO>
    <H1> Hello
    <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "") %>
    ! I am SimpleQuery JSP.
  </H1>
  <HR>
  <B> I will do a basic JDBC query to get employee data
      from EMP table in schema SCOTT..
  </B>

  <P>
  <%
    try {
      // Use the following 2 files when running inside Oracle 8i
      // Connection conn = new oracle.jdbc.driver.OracleDriver().
      //                      defaultConnection ();
      Connection conn =
        DriverManager.getConnection((String)session.getValue("connStr"),
                                    "scott", "tiger");

      Statement stmt = conn.createStatement ();
      ResultSet rset = stmt.executeQuery ("SELECT ename, sal " +
                                          "FROM scott.emp ORDER BY ename");
```



```

        if (rset.next()) {
    %>
    <TABLE BORDER=1 BGCOLOR="C0C0C0">
    <TH WIDTH=200 BGCOLOR="white"> <I>Employee Name</I> </TH>
    <TH WIDTH=100 BGCOLOR="white"> <I>Salary</I> </TH>
    <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
        <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
    </TR>

    <%      while (rset.next()) {
    %>

    <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
        <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
    </TR>

    <% }
    %>
    </TABLE>
    <% }
    else {
    %>
    <P> Sorry, the query returned no rows! </P>

    <%
    }
    rset.close();
    stmt.close();
    } catch (SQLException e) {
    out.println("<P>" + "There was an error doing the query:");
    out.println ("<PRE>" + e + "</PRE> \n <P>");
    }
    %>

    </BODY>
</HTML>

```

ユーザー指定の問合せ — JDBCQuery.jsp

このページでは、ユーザー指定の条件に従って scott.emp 表の問合せを実行し、結果を出力しています。

```

<%@ page import="java.sql.*" %>

<HTML>
<HEAD> <TITLE> The JDBCQuery JSP </TITLE> </HEAD>
<BODY BGCOLOR=white>

```

```
<% String searchCondition = request.getParameter("cond");
    if (searchCondition != null) { %>
        <H3> Search results for : <I> <%= searchCondition %> </I> </H3>
        <%= runQuery(searchCondition) %>
        <HR><BR>
    <% } %>

<B>Enter a search condition:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="cond" SIZE=30>
<INPUT TYPE="submit" VALUE="Ask Oracle");
</FORM>
</BODY>
</HTML>
<%!
    private String runQuery(String cond) throws SQLException {
        Connection conn = null;
        Statement stmt = null;
        ResultSet rset = null;
        try {
            DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
            conn = DriverManager.getConnection((String) session.getValue("connStr"),
                                                "scott", "tiger");

            stmt = conn.createStatement();
            rset = stmt.executeQuery ("SELECT ename, sal FROM scott.emp "+
                                     (cond.equals("") ? "" : "WHERE " + cond ));
            return (formatResult(rset));
        } catch (SQLException e) {
            return ("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
        } finally {
            if (rset!= null) rset.close();
            if (stmt!= null) stmt.close();
            if (conn!= null) conn.close();
        }
    }

    private String formatResult(ResultSet rset) throws SQLException {
        StringBuffer sb = new StringBuffer();
        if (!rset.next())
            sb.append("<P> No matching rows.<P>\n");
        else {
            sb.append("<UL><B>");
            do {
                sb.append("<LI>" + rset.getString(1) +
                           " earns $ " + rset.getInt(2) + ".</LI>\n");
            } while (rset.next());
            sb.append("</B></UL>");
        }
    }
}
```

```

        return sb.toString();
    }
    %>

```

問合せ Bean を使用した問合せ — UseHtmlQueryBean.jsp

このページでは、JavaBeans である HtmlQueryBean を使用し、ユーザー指定の条件に従って scott.emp 表の問合せを実行しています。HtmlQueryBean では、クラス HtmlTable を使用して出力が HTML 表にフォーマットされます。このサンプルには、JSP ページ HtmlQueryBean と HtmlTable のコードが含まれています。

UseHtmlQueryBean.jsp のコード

```

<jsp:useBean id="htmlQueryBean" class="beans.HtmlQueryBean" scope="session" />
<jsp:setProperty name="htmlQueryBean" property="searchCondition" />

<HTML>
<HEAD> <TITLE> The UseHtmlQueryBean JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">

<% String searchCondition = request.getParameter("searchCondition");
   if (searchCondition != null) { %>
       <H3>Search Results for : <I> <%= searchCondition %> </I> </H3>
       <%= htmlQueryBean.getResult() %>
       <BR> <HR>
   <% } %>

<P><B>Enter a search condition:</B></P>
<FORM METHOD=get>
<INPUT TYPE=text NAME="searchCondition" SIZE=30>
<INPUT TYPE=submit VALUE="Ask Oracle">
</FORM>
</BODY>
</HTML>

```

HtmlQueryBean.java のコード

```

package beans;
import java.sql.*;

public class HtmlQueryBean {

    private String searchCondition = "";
    private String connStr = null;

```

```
public String getResult() throws SQLException {
    return runQuery();
}

public void setSearchCondition(String searchCondition) {
    this.searchCondition = searchCondition;
}

public void setConnStr(String connStr) {
    this.connStr = connStr;
}

private String runQuery() {
    Connection conn = null;
    Statement stmt = null;
    ResultSet rset = null;
    try {
        if (conn == null) {
            DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
            conn = DriverManager.getConnection(connStr,
                                              "scott","tiger");
        }
        stmt = conn.createStatement();
        rset = stmt.executeQuery ("SELECT ename as \"Name\", \" +
                                \"empno as \"Employee Id\", \" +
                                \"sal as \"Salary\", \" +
                                \"TO_CHAR(hiredate, 'DD-MON-YYYY') as \"Date Hired\" \" +
                                \"FROM scott.emp \" + (searchCondition.equals(\"\") ? \"\" :
                                \"WHERE \" + searchCondition ));
        return format(rset);
    } catch (SQLException e) {
        return ("<P> SQL error: <PRE> \" + e + \" </PRE> </P>\\n");
    }
    finally {
        try {
            if (rset != null) rset.close();
            if (stmt != null) stmt.close();
            if (conn != null) conn.close();
        } catch (SQLException ignored) {}
    }
}

public static String format(ResultSet rs) throws SQLException {
    StringBuffer sb = new StringBuffer();
    if (rs == null || !rs.next())
        sb.append("<P> No matching rows.<P>\\n");
    else {
```

```

        sb.append("<TABLE BORDER>\n");
        ResultSetMetaData md = rs.getMetaData();
        int numCols = md.getColumnCount();
        for (int i=1; i<= numCols; i++) {
            sb.append("<TH><I>" + md.getColumnLabel(i) + "</I></TH>");
        }
        do {
            sb.append("<TR>\n");
            for (int i = 1; i <= numCols; i++) {
                sb.append("<TD>");
                Object obj = rs.getObject(i);
                if (obj != null) sb.append(obj.toString());
                sb.append("</TD>");
            }
            sb.append("</TR>");
        } while (rs.next());
        sb.append("</TABLE>");
    }
    return sb.toString();
}
}

```

HtmlTable.java のコード

```

import java.sql.*;

public class HtmlTable {

    public static String format(ResultSet rs) throws SQLException {
        StringBuffer sb = new StringBuffer();
        if (rs == null || !rs.next())
            sb.append("<P> No matching rows.<P>\n");
        else {
            sb.append("<TABLE BORDER>\n");
            ResultSetMetaData md = rs.getMetaData();
            int numCols = md.getColumnCount();
            for (int i=1; i<= numCols; i++) {
                sb.append("<TH><I>" + md.getColumnLabel(i) + "</I></TH>");
            }
            do {
                sb.append("<TR>\n");

                for (int i = 1; i <= numCols; i++) {
                    sb.append("<TD>");
                    Object obj = rs.getObject(i);
                    if (obj != null) sb.append(obj.toString());
                    sb.append("</TD>");
                }
            } while (rs.next());
            sb.append("</TABLE>");
        }
    }
}

```

```
    }
    sb.append("</TR>");
  } while (rs.next());
  sb.append("</TABLE>");
}
return sb.toString();
}
}
```

接続キャッシュ — ConnCache3.jsp および ConnCache1.jsp

この項では、Oracle のキャッシュ実装を使用する接続キャッシュの2つの例を示します。この実装では、Oracle JDBC の `OracleConnectionCacheImpl` クラスを使用しています。概要は、4-5 ページの「データベース接続のキャッシュ」を参照してください。詳細は、『Oracle8i 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。

最初の例の `ConnCache3.jsp` では、独自のキャッシュ設定が実行されます。

2 番目の例の `ConnCache1.jsp` では、別のページ `setupcache.jsp` を使用して設定が実行されます。

3 つのページすべてのコードを示します。

注意： より便利な代替手法として、OracleJSP で提供される `ConnCacheBean` JavaBeans を使用できます。9-25 ページの「[ConnCacheBean](#) を使用するページ — `ConnCacheBeanDemo.jsp`」を参照してください。

ConnCache3.jsp のコード（キャッシュ設定あり）

このサンプル・ページでは、独自の接続キャッシュ設定が処理されます。

```
<%@ page import="java.sql.*, javax.sql.*, oracle.jdbc.pool.*" %>

<!-------
* This is a JavaServer Page that uses Connection Caching at Session
* scope.
-----!>

<jsp:useBean id="ods" class="oracle.jdbc.pool.OracleConnectionCacheImpl"
scope="session" />

<HTML>
<HEAD>
<TITLE>
  ConnCache 3   JSP
</TITLE>
```

```

</HEAD>
<BODY BGCOLOR=EOFFFO>
<H1> Hello
  <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "") %>
  ! I am Connection Caching JSP.
</H1>
<HR>
<B> Session Level Connection Caching.
</B>

<P>
<%
  try {
    ods.setURL((String)session.getValue("connStr"));
    ods.setUser("scott");
    ods.setPassword("tiger");

    Connection conn = ods.getConnection ();
    Statement stmt = conn.createStatement ();
    ResultSet rset = stmt.executeQuery ("SELECT ename, sal " +
                                         "FROM scott.emp ORDER BY ename");

    if (rset.next()) {
%>
      <TABLE BORDER=1 BGCOLOR="C0C0C0">
      <TH WIDTH=200 BGCOLOR="white"> <I>Employee Name</I> </TH>
      <TH WIDTH=100 BGCOLOR="white"> <I>Salary</I> </TH>
      <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
        <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
      </TR>

<%
      while (rset.next()) {
%>

      <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
        <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
      </TR>

<% }
%>
      </TABLE>
<% }
%>
      <P> Sorry, the query returned no rows! </P>

<%

```

```
    }
    rset.close();
    stmt.close();
    conn.close(); // Put the Connection Back into the Pool

} catch (SQLException e) {
    out.println("<P>" + "There was an error doing the query:");
    out.println ("<PRE>" + e + "</PRE> \n <P>");
}
%>

</BODY>
</HTML>
```

ConnCache1.jsp および setupcache.jsp のコード

このサンプル・ページでは、接続キャッシュ設定用に別のページ `setupcache.jsp` が静的に挿入されます。両方のページのコードを示します。

ConnCache1.jsp

```
<%@ include file="setupcache.jsp" %>
<%@ page import="java.sql.*, javax.sql.*, oracle.jdbc.pool.*" %>

<!-------
 * This is a JavaServer Page that uses Connection Caching over application
 * scope. The Cache is created in an application scope in setupcache.jsp
 * Connection is obtained from the Cache and recycled back once done.
-----!>

<HTML>
  <HEAD>
    <TITLE>
      ConnCache1 JSP
    </TITLE>
  </HEAD>
  <BODY BGCOLOR=EOFFFO>
    <H1> Hello
    <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "") %>
    ! I am Connection Caching JSP.
  </H1>
  <HR>
  <B> I get the Connection from the Cache and recycle it back.
  </B>

  <P>
  <%
```



```

try {
    Connection conn = cods.getConnection();

    Statement stmt = conn.createStatement ();
    ResultSet rset = stmt.executeQuery ("SELECT ename, sal " +
                                         "FROM scott.emp ORDER BY ename");

    if (rset.next()) {
%>
<TABLE BORDER=1 BGCOLOR="C0C0C0">
<TH WIDTH=200 BGCOLOR="white"> <I>Employee Name</I> </TH>
<TH WIDTH=100 BGCOLOR="white"> <I>Salary</I> </TH>
<TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
      <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
</TR>

<%
    while (rset.next()) {
%>

      <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
        <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
      </TR>

<% }
%>
</TABLE>
<% }
%>
    else {
%>
      <P> Sorry, the query returned no rows! </P>

<%
    }
    rset.close();
    stmt.close();
    conn.close(); // Put the Connection Back into the Pool
  } catch (SQLException e) {
    out.println("<P>" + "There was an error doing the query:");
    out.println ("<PRE>" + e + "</PRE> \n <P>");
  }
%>

</BODY>
</HTML>

```

setupcache.jsp

```
<jsp:useBean id="cods" class="oracle.jdbc.pool.OracleConnectionCacheImpl"
  scope="application">
<%
    cods.setURL((String)session.getValue("connStr"));
    cods.setUser("scott");
    cods.setPassword("tiger");
    cods.setStmtCache (5);
%>
</jsp:useBean>
```

Database-Access JavaBeans のサンプル

この項では、Oracle Database-Access JavaBeans の使用例を示します。これらの Bean は OracleJSP で提供されますが、通常は他の JSP 環境に移植可能です。ただし、接続キャッシュ Bean は接続キャッシュの Oracle JDBC 実装に依存することに注意してください。

DBBean はこの種の JavaBeans のうち最も単純で、独自の接続機能と問合せのサポートのみが含まれています。より複雑な操作の場合は、ConnBean（単純接続用）、ConnCacheBean（接続キャッシュ用）および CursorBean（一般的な SQL DML 操作用）を適切に組み合わせて使用します。

詳細は、5-12 ページの「[Oracle Database-Access JavaBeans](#)」を参照してください。

次の例を示します。

- [DBBean を使用するページ — DBBeanDemo.jsp](#)
- [ConnBean を使用するページ — ConnBeanDemo.jsp](#)
- [CursorBean を使用するページ — CursorBeanDemo.jsp](#)
- [ConnCacheBean を使用するページ — ConnCacheBeanDemo.jsp](#)

注意： Oracle には、他の状況で前述の JavaBeans を使用する SQL 機能用のカスタム・タグも用意されています。これらのタグの使用例については、5-26 ページの「[SQL タグの例](#)」を参照してください。

DBBean を使用するページー DBBeanDemo.jsp

このページでは、DBBean オブジェクトを使用してデータベースに接続し、問合せを実行し、結果を HTML 表として出力しています。

```
<%@ page import="java.sql.*" %>

<!-------
 * This is a basic JavaServer Page that uses a DB Access Bean and queries
 * dept and emp tables in schema scott and outputs the result in an html table.
 *
-----!>

<jsp:useBean id="dbbean" class="oracle.jsp.dbutil.DBBean" scope="session">
  <jsp:setProperty name="dbbean" property="User" value="scott"/>
  <jsp:setProperty name="dbbean" property="Password" value="tiger"/>
  <jsp:setProperty name="dbbean" property="URL" value=
    "<%= (String)session.getValue(\"connStr\") %>" />
</jsp:useBean>

<HTML>
  <HEAD>
    <TITLE>
      DBBeanDemo JSP
    </TITLE>
  </HEAD>
  <BODY BGCOLOR=EOFFFO>
    <H1> Hello
    <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "") %>
    ! I am DBBeanDemo JSP.
  </H1>
  <HR>
  <B> I'm using DBBean and querying DEPT & EMP tables in schema SCOTT.....
    I get all employees who work in the Research department.
  </B>

  <P>
  <%
    try {

      String sql_string = " select ENAME from EMP,DEPT " +
                          " where DEPT.DNAME = 'RESEARCH' " +
                          " and DEPT.DEPTNO = EMP.DEPTNO";

      // Make the Connection
      dbbean.connect();

      // Execute the SQL and get a HTML table
```

```
        out.println(dbbean.getResultAsHTMLTable(sql_string));

        // Close the Bean to close the connection
        dbbean.close();
    } catch (SQLException e) {
        out.println("<P>" + "There was an error doing the query:");
        out.println ("<PRE>" + e + "</PRE> \n <P>");
    }
}

%>

</BODY>
</HTML>
```

ConnBean を使用するページ — ConnBeanDemo.jsp

このページでは、ConnBean オブジェクト（単純接続用）を使用して CursorBean オブジェクトを取得し、CursorBean オブジェクトを使用して問合せの結果を HTML 表として出力しています。

```
<%@ page import="java.sql.* , oracle.jsp.dbutil.*" %>

<!-------
* This is a basic JavaServer Page that uses a Connection Bean and queries
* emp table in schema scott and outputs the result in an html table.
*
-----!>

<jsp:useBean id="cbean" class="oracle.jsp.dbutil.ConnBean" scope="session">
  <jsp:setProperty name="cbean" property="User" value="scott"/>
  <jsp:setProperty name="cbean" property="Password" value="tiger"/>
  <jsp:setProperty name="cbean" property="URL" value=
    "<%= (String)session.getValue(\"connStr\") %>"/>
  <jsp:setProperty name="cbean" property="PreFetch" value="5"/>
  <jsp:setProperty name="cbean" property="StmtCacheSize" value="2"/>
</jsp:useBean>

<HTML>
  <HEAD>
    <TITLE>
      Connection Bean Demo JSP
    </TITLE>
  </HEAD>
  <BODY BGCOLOR=EOFFFO>
    <H1> Hello
    <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "") %>
    ! I am Connection Bean Demo JSP.
  </H1>
```

```

<HR>
<B> I'm using connection and a query bean and querying employee names
      and salaries from EMP table in schema SCOTT..
</B>

<P>
<%
    try {

        // Make the Connection
        cbean.connect();

        String sql = "SELECT ename, sal FROM scott.emp ORDER BY ename";

        // get a Cursor Bean
        CursorBean cb = cbean.getCursorBean (CursorBean.PREP_STMT, sql);

        out.println(cb.getResultAsHTMLTable());

        // Close the cursor bean
        cb.close();
        // Close the Bean to close the connection
        cbean.close();
    } catch (SQLException e) {
        out.println("<P>" + "There was an error doing the query:");
        out.println ("<PRE>" + e + "</PRE> \n <P>");
    }
%>

</BODY>
</HTML>

```

CursorBean を使用するページー CursorBeanDemo.jsp

このページでは、ConnBean オブジェクト（単純接続用）と CursorBean オブジェクトを使用して PL/SQL 文を実行し、REF CURSOR を取得し、結果を HTML 表に変換しています。

```

<%@ page import="java.sql.* , oracle.jsp.dbutil.*" %>

<!-------
* This is a basic JavaServer Page that uses a Cursor and Conn Beans and queries
* dept table in schema scott and outputs the result in an html table.
*
-----!>

<jsp:useBean id="connbean" class="oracle.jsp.dbutil.ConnBean" scope="session">

```

```
<jsp:setProperty name="connbean" property="User" value="scott"/>
<jsp:setProperty name="connbean" property="Password" value="tiger"/>
<jsp:setProperty name="connbean" property="URL" value=
    "<%=      (String)session.getValue(\"connStr\") %>" />
</jsp:useBean>
<jsp:useBean id="cbean" class="oracle.jsp.dutil.CursorBean" scope="session">
    <jsp:setProperty name="cbean" property="PreFetch" value="10"/>
    <jsp:setProperty name="cbean" property="ExecuteBatch" value="2"/>
</jsp:useBean>

<HTML>
  <HEAD>
    <TITLE>
      CursorBean Demo JSP
    </TITLE>
  </HEAD>
  <BODY BGCOLOR=E0FFFF>
    <H1> Hello
    <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "") %>
    ! I am Cursor Bean JSP.
    </H1>
    <HR>
    <B> I`m using cbean and i`m quering department names from DEPT table
        in schema SCOTT..
    </B>

    <P>
    <%

        try {

            // Make the Connection
            connbean.connect();

            String sql = "BEGIN OPEN ? FOR SELECT DNAME FROM DEPT; END;";

            // Create a Callable Statement
            cbean.create ( connbean, CursorBean.CALL_STMT, sql);
            cbean.registerOutParameter(1,oracle.jdbc.driver.OracleTypes.CURSOR);

            // Execute the PLSQL
            cbean.executeUpdate ();

            // Get the Ref Cursor
            ResultSet rset = cbean.getCursor(1);

            out.println(oracle.jsp.dutil.BeanUtil.translateToHTMLTable (rset));
```

```

// Close the RefCursor
rset.close();

// Close the Bean
cbean.close();

// Close the connection
connbean.close();

} catch (SQLException e) {
    out.println("<P>" + "There was an error doing the query:");
    out.println ("<PRE>" + e + "</PRE> \n <P>");
}
%>

</BODY>
</HTML>

```

ConnCacheBean を使用するページ — ConnCacheBeanDemo.jsp

このページでは、ConnCacheBean オブジェクトを使用して接続キャッシュから接続を取得しています。その後、標準 JDBC 機能を使用して問合せを実行し、結果を HTML 表としてフォーマットします。

```

<%@ page import="java.sql.*, javax.sql.*, oracle.jsp.dbutil.ConnCacheBean" %>

<!-------
* This is a basic JavaServer Page that does a JDBC query on the
* emp table in schema scott and outputs the result in an html table.
* Uses Connection Cache Bean.
-----!>

<jsp:useBean id="ccbean" class="oracle.jsp.dbutil.ConnCacheBean"
    scope="session">
    <jsp:setProperty name="ccbean" property="user" value="scott"/>
    <jsp:setProperty name="ccbean" property="password" value="tiger"/>
    <jsp:setProperty name="ccbean" property="URL" value=
        "<%= (String)session.getValue(\"connStr\") %>" />
    <jsp:setProperty name="ccbean" property="MaxLimit" value="5" />
    <jsp:setProperty name="ccbean" property="CacheScheme" value=
        "<%= ConnCacheBean.FIXED_RETURN_NULL_SCHEME %>" />
</jsp:useBean>
<HTML>
<HEAD>
<TITLE>
    SimpleQuery JSP

```

```
</TITLE>
</HEAD>
<BODY BGCOLOR=EOFFFO>
<H1> Hello
  <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "") %>
  ! I am Connection Cache Demo Bean
</H1>
<HR>
<B> I will do a basic JDBC query to get employee data
    from EMP table in schema SCOTT. The connection is obtained from
    the Connection Cache.
</B>

<P>
<%
  try {
    Connection conn = ccbean.getConnection();

    Statement stmt = conn.createStatement ();
    ResultSet rset = stmt.executeQuery ("SELECT ename, sal " +
                                         "FROM scott.emp ORDER BY ename");

    if (rset.next()) {
%>
      <TABLE BORDER=1 BGCOLOR="C0C0C0">
      <TH WIDTH=200 BGCOLOR="white"> <I>Employee Name</I> </TH>
      <TH WIDTH=100 BGCOLOR="white"> <I>Salary</I> </TH>
      <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
        <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
      </TR>

      while (rset.next()) {
%>
        <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
          <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
        </TR>

      <% }
%>
      </TABLE>
      <% }
%>
      else {
%>
        <P> Sorry, the query returned no rows! </P>

      <%
        }
    }
  }
%>
```



```

        rset.close();
        stmt.close();
        conn.close();
        ccbean.close();
    } catch (SQLException e) {
        out.println("<P>" + "There was an error doing the query:");
        out.println ("<PRE>" + e + "</PRE> \n <P>");
    }
%>

</BODY>
</HTML>

```

カスタム・タグのサンプル

この項のトピックは、次のとおりです。

- いくつかの Oracle JSP マークアップ言語 (JML) カスタム・タグの使用例
- このマニュアルの他の項に記載されているカスタム・タグのサンプルの参照情報

JML タグのサンプル — hellouser_jml.jsp

この項では、いくつかの Oracle JML カスタム・タグの基本的な使用例を示します。

これは、前述した hellouser.jsp サンプルの修正版です。ただし、ここでは JML コードとオリジナル・コードの両方を示します。

JML タグ・ライブラリのランタイム実装は、他の JSP 環境に移植できることに注意してください。ランタイム実装の概要は、7-18 ページの「[JSP マークアップ言語 \(JML\) のサンプル・タグ・ライブラリの概要](#)」を参照してください。コンパイル時（移植できない）実装の詳細は、[付録 C「コンパイル時 JML タグ・サポート」](#)を参照してください。

hellouser_jml.jsp のコード (JML タグを使用)

```

<%-----
    Copyright (c) 1999, Oracle Corporation. All rights reserved.
    -----%>

<%@page session="false" %>
<%@ taglib uri="WEB-INF/jml.tld" prefix="jml" %>

<jml:useForm id="name" param="newName" scope="request" />

<HTML>
<HEAD>
<TITLE>

```

```
Hello User
</TITLE>
</HEAD>

<BODY>

<jml:if condition="!name.isEmpty()" >
<H3>Welcome <jml:print eval="name.getValue()" /></H3>
</jml:if>

<P>
Enter your Name:
<FORM METHOD=get>
<INPUT TYPE=TEXT name=newName size = 20><br>
<INPUT TYPE=SUBMIT VALUE="Submit name">
</FORM>

</BODY>
</HTML>
```

hellouser.jsp のコード（JML タグを不使用）

```
<%-----
    Copyright (c) 1999, Oracle Corporation. All rights reserved.
-----%>

<%@page session="false" %>

<jsp:useBean id="name" class="oracle.jsp.jml.JmlString" scope="request" >
    <jsp:setProperty name="name" property="value" param="newName" />
</jsp:useBean>

<HTML>
<HEAD>
<TITLE>
Hello User
</TITLE>
</HEAD>

<BODY>

<% if (!name.isEmpty()) { %>
<H3>Welcome <%= name.getValue() %></H3>
<% } %>

<P>
Enter your Name:
```

```
<FORM METHOD=get>
<INPUT TYPE=TEXT name=newName size = 20><br>
<INPUT TYPE=SUBMIT VALUE="Submit name">
</FORM>

</BODY>
</HTML>
```

他のカスタム・タグ・サンプルのポインタ

カスタム・タグのサンプルは、このドキュメントの他の箇所にも記載されています。

- 標準 JSP 1.1 準拠のカスタム・タグの定義および使用例の詳細は、7-13 ページの「[全体の例: カスタム・タグの定義と使用](#)」を参照してください。
- SQL 機能の Oracle カスタム・タグ・ライブラリを使用する例は、5-26 ページの「[SQL タグの例](#)」を参照してください。

Oracle 固有のプログラミング拡張機能のサンプル

この項では、Oracle 固有の拡張機能の様々な使用例について説明します。たとえば、次の拡張機能があります。

- [JspScopeListener](#) を使用するページ — [scope.jsp](#)
- XML の問合せ — [XMLQuery.jsp](#)
- SQLJ の問合せ — [SQLJSelectInto.sqljsp](#) および [SQLJIterator.sqljsp](#)

JspScopeListener を使用するページ — scope.jsp

このサンプルは、有効範囲に連結されている JSP オブジェクトが「有効範囲外」になるときに通知できるようにする、JspScopeListener 実装の使用方法を示しています。このサンプルでは、登録済みのオブジェクトまたはメソッドに有効範囲外通知を再ディスパッチする、汎用リスナーが実装されます。このリスナーを使用すると、要求およびページの有効範囲外通知のページ・イベント・ハンドラを、`scope.jsp` でシミュレーションできます。

このサンプルでは、リスナー・オブジェクトが作成され、`request` および `page` 有効範囲に連結されます。リスナーから転送された有効範囲外通知を処理するローカル・メソッドが登録されます。この具体例を示すために、サンプルでは2つのカウンタが維持されます。1つ目はページ数のカウンタで、2つ目は挿入ファイル数のカウンタです。

ページが有効範囲外になると、現在のページ数がログに記録されます。要求が有効範囲外になると、挿入ページ数がログに記録されます。その後、サンプル自体が5回挿入されます。

このサンプルでは、ページ数1を示す6つのメッセージに続いて、5回の `jsp:include` 操作が発生したことを示す単一のメッセージが出力されます。

JspScopeListener メカニズムの概要は、5-30 ページの「[OracleJSP のイベント処理 — JspScopeListener](#)」を参照してください。

リスナー実装 — PageEventDispatcher

PageEventDispatcher は、JspScopeListener インタフェースを実装する JavaBeans です。このインタフェースは `outOfScope()` イベント・メソッドを定義し、入力として `JspScopeEvent` オブジェクトを取ります。オブジェクトに対応付けられた有効範囲 (application、session、page または request) が終了すると、PageEventDispatcher オブジェクトの `outOfScope()` メソッドがコールされます。

このサンプルで、PageEventDispatcher オブジェクトは JSP ページのディスパッチャとして機能し、JSP ページはページおよび要求イベントに関する `globals.jsa` の「終了時」機能の等価を処理できます。JSP ページでは、イベント・ハンドラを提供する有効範囲ごとに、PageEventDispatcher オブジェクトが作成されます。その後、イベント・ハンドラ・メソッドが PageEventDispatcher オブジェクトに登録されます。PageEventDispatcher オブジェクトは、有効範囲外になったことを示す通知を受け取ると、ページの登録済みの「終了時」メソッドをコールします。

```
package oracle.jsp.sample.event;

import java.lang.reflect.*;
import oracle.jsp.event.*;

public class PageEventDispatcher extends Object implements JspScopeListener {

    private Object page;
    private String methodName;
    private Method method;

    public PageEventDispatcher() {
    }

    public Object getPage() {
        return page;
    }

    public void setPage(Object page) {
        this.page = page;
    }

    public String getMethodName() {
        return methodName;
    }

    public void setMethodName(String m)
        throws NoSuchMethodException, ClassNotFoundException {
```

```
        method = verifyMethod(m);
        methodName = m;
    }

    public void outOfScope(JspScopeEvent ae) {
        int scope = ae.getScope();

        if ((scope == javax.servlet.jsp.PageContext.REQUEST_SCOPE ||
            scope == javax.servlet.jsp.PageContext.PAGE_SCOPE) &&
            method != null) {
            try {
                Object args[] = {ae.getApplication(), ae.getContainer()};
                method.invoke(page, args);
            } catch (Exception e) {
                // catch all and continue
            }
        }
    }

    private Method verifyMethod(String m)
        throws NoSuchMethodException, ClassNotFoundException {
        if (page == null) throw new NoSuchMethodException
            ("A page hasn't been set yet.");

        /* Don't know whether this is a request or page handler so try one then
           the other
        */
        Class c = page.getClass();
        Class pTypes[] = {Class.forName("javax.servlet.ServletContext"),
            Class.forName("javax.servlet.jsp.PageContext")};

        try {
            return c.getDeclaredMethod(m, pTypes);
        } catch (NoSuchMethodException nsme) {
            // fall through and try the request signature
        }

        pTypes[1] = Class.forName("javax.servlet.http.HttpServletRequest");
        return c.getDeclaredMethod(m, pTypes);
    }
}
```

scope.jsp のソース

この JSP ページでは、前述の PageEventDispatcher クラス（JspScopeListener インタフェースを実装するクラス）を使用して、page または request 有効範囲のイベントを追跡しています。

```
<%-- declare request and page scoped beans here --%>

<jsp:useBean id = "includeCount" class = "oracle.jsp.jml.JmlNumber" scope = "request" />
<jsp:useBean id = "pageCount" class = "oracle.jsp.jml.JmlNumber" scope = "page" >
    <jsp:setProperty name = "pageCount"
        property = "value" value = "<%= pageCount.getValue() + 1 %>" />
</jsp:useBean>

<%-- declare the event dispatchers --%>
<jsp:useBean id = "requestDispatcher" class = "oracle.jsp.sample.event.PageEventDispatcher"
    scope = "request" >
    <jsp:setProperty name = "requestDispatcher" property = "page" value = "<%= this %>" />
    <jsp:setProperty name = "requestDispatcher" property = "methodName"
        value = "request_OnEnd" />
</jsp:useBean>

<jsp:useBean id = "pageDispatcher" class = "oracle.jsp.sample.event.PageEventDispatcher"
    scope = "page" >
    <jsp:setProperty name = "pageDispatcher" property = "page" value = "<%= this %>" />
    <jsp:setProperty name = "pageDispatcher" property = "methodName" value = "page_OnEnd" />
</jsp:useBean>

<%!
    // request_OnEnd Event Handler
    public void request_OnEnd(ServletContext application, HttpServletRequest request) {
        // acquire beans
        oracle.jsp.jml.JmlNumber includeCount =
            (oracle.jsp.jml.JmlNumber) request.getAttribute("includeCount");

        // now cleanup the bean
        if (includeCount != null) application.log
            ("request_OnEnd: Include count = " + includeCount.getValue());
    }

    // page_OnEnd Event Handler
    public void page_OnEnd(ServletContext application, PageContext page) {
        // acquire beans
        oracle.jsp.jml.JmlNumber pageCount =
            (oracle.jsp.jml.JmlNumber) page.getAttribute("pageCount");

        // now cleanup the bean -- uncomment code for real bean
        if (pageCount != null) application.log
```

```

        ("page_OnEnd: Page count = " + pageCount.getValue());
    }
%>

<!-- Page implementation goes here -->

<jsp:setProperty name = "includeCount" property = "value"
    value = '<%= (request.getAttribute("javax.servlet.include.request_uri")
        != null) ? includeCount.getValue() + 1 : 0 %>' />

<h2> Hello World </h2>

Included: <%= request.getAttribute("javax.servlet.include.request_uri") %>
Count: <%= includeCount.getValue() %> <br>

<% if (includeCount.getValue() < 5) { %>
    <jsp:include page="scope.jsp" flush = "true" />
<% } %>

```

XML の問合せ — XMLQuery.jsp

この例では、データベースに接続して、問合せを実行し、`oracle.xml.sql.query.OracleXMLQuery` クラスの機能を使用して結果を XML 文字列として出力しています。

これは、Oracle 固有の機能です。`OracleXMLQuery` クラスは、Oracle8i で XML-SQL ユーティリティの一部として提供されます。

JSP ページでの XML および XSL の使用方法の概要は、5-8 ページの「[OracleJSP の XML および XSL サポート](#)」を参照してください。

```

<!-------
    Copyright (c) 1999, Oracle Corporation. All rights reserved.
    -----%>

<%@ page import = "java.sql.*,oracle.xml.sql.query.OracleXMLQuery" %>
<html>
    <head><title> The XMLQuery Demo </title></head>
    <body>
        <h1> XMLQuery Demo </h1>
        <h2> Employee List in XML </h2>
        <b>(View Page Source in your browser to see XML output)</b>
        <% Connection conn = null;
            Statement stmt = null;
            ResultSet rset = null;
            try {

```

```
// determine JDBC driver name from session value
// if null, use JDBC kprb driver if in JServer, JDBC oci otherwise
String dbURL = (String)session.getValue("connStr");
if (dbURL == null)
    dbURL = (System.getProperty("oracle.jserver.version") == null?
        "jdbc:oracle:oci8:@" : "jdbc:oracle:kprb:@");

DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
conn = DriverManager.getConnection(dbURL, "scott", "tiger");
stmt = conn.createStatement ();
rset = stmt.executeQuery ("SELECT ename, sal " +
    "FROM scott.emp ORDER BY ename");

OracleXMLQuery xq = new OracleXMLQuery(conn, rset); %>
    <PRE> <%= xq.getXMLString() %> </PRE>
<% } catch (java.sql.SQLException e) { %>
    <P> SQL error: <PRE> <%= e %> </PRE> </P>
<% } finally {
    if (stmt != null) stmt.close();
    if (rset != null) rset.close();
    if (conn != null) conn.close();
} %>
</body>
</html>
```

SQLJ の問合せ — SQLJSelectInto.sqljsp および SQLJIterator.sqljsp

この項では、JSP ページ内で SQLJ を使用してデータベースの問合せを実行する例を示します。

最初の例の SQLJSelectInto.sqljsp では、SQLJ の SELECT INTO 構文を使用して 1 行を選択しています。

2 番目の例の SQLJIterator.sqljsp では、複数行を JDBC の結果セットに似た SQLJ イテレータを使用して選択しています。

JSP ページでの SQLJ の使用方法については、5-31 ページの「[OracleJSP による Oracle SQLJ のサポート](#)」を参照してください。

Oracle SQLJ のプログラミング機能と構文の概要は、『Oracle8i SQLJ 開発者ガイドおよびリファレンス』を参照してください。

SQLJSelectInto.sqljsp のコード (1 行選択)

この例では、SQLJ の SELECT INTO 構文を使用して、データベースから 1 行を選択しています。

```
<%@ page import="sqlj.runtime.ref.DefaultContext,oracle.sqlj.runtime.Oracle" %>
```

```
<HTML>
```

```
<HEAD> <TITLE> The SQLJSelectInto JSP </TITLE> </HEAD>
```

```
<BODY BGCOLOR=white>
```

```
<%
```

```
String connStr=request.getParameter("connStr");
```

```
if (connStr==null) {
```

```
    connStr=(String)session.getValue("connStr");
```

```
} else {
```

```
    session.putValue("connStr",connStr);
```

```
}
```

```
if (connStr==null) { %>
```

```
<jsp:forward page="../setconn.jsp" />
```

```
<%
```

```
}
```

```
%>
```

```
<%
```

```
String empno = request.getParameter("empno");
```

```
if (empno != null) { %>
```

```
    <H3> Employee # <%=empno %> Details: </H3>
```

```
    <%= runQuery(connStr,empno) %>
```

```
    <HR><BR>
```

```
<% } %>
```

```
<B>Enter an employee number:</B>
```

```
<FORM METHOD=get>
```

```
<INPUT TYPE="text" NAME="empno" SIZE=10>
```

```
<INPUT TYPE="submit" VALUE="Ask Oracle");
```

```
</FORM>
```

```
</BODY>
```

```
</HTML>
```

```
<%!
```

```
private String runQuery(String connStr, String empno) throws
java.sql.SQLException {
```

```
    DefaultContext dctx = null;
```

```
    String ename = null; double sal = 0.0; String hireDate = null;
```

```
    StringBuffer sb = new StringBuffer();
```

```
    try {
```

```
        dctx = Oracle.getConnection(connStr, "scott", "tiger");
```

```
        #sql [dctx] { SELECT ename, sal, TO_CHAR(hiredate, 'DD-MON-YYYY')}
```

```
        INTO :ename, :sal, :hireDate
        FROM scott.emp WHERE UPPER(empno) = UPPER(:empno)
    };
    sb.append("<BLOCKQUOTE><BIG><B><PRE>\n");
    sb.append("Name      : " + ename + "\n");
    sb.append("Salary    : " + sal + "\n");
    sb.append("Date hired : " + hireDate);
    sb.append("</PRE></B></BIG></BLOCKQUOTE>");

    } catch (java.sql.SQLException e) {
        sb.append("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
    } finally {
        if (dctx!= null) dctx.close();
    }
    return sb.toString();
}
%>
```

SQLJIterator.sqljsp のコード（複数行選択）

この例では、SQLJ イテレータを使用して、データベースから複数行を選択しています。

```
<%% page import="java.sql.*" %>
<%% page import="sqlj.runtime.ref.DefaultContext,oracle.sqlj.runtime.Oracle" %>

<!-------
 * This is a SQLJ JavaServer Page that does a SQLJ query on the
 * emp table in schema scott and outputs the result in an html table.
 *
-----!>

<%! #sql iterator Empiter(String ename, double sal, java.sql.Date hiredate) %>

<%
    String connStr=request.getParameter("connStr");
    if (connStr==null) {
        connStr=(String)session.getValue("connStr");
    } else {
        session.putValue("connStr",connStr);
    }
    if (connStr==null) { %>
<jsp:forward page="../setconn.jsp" />
<%
    }
%>

<%
```

```

DefaultContext dctx = null;
dctx = Oracle.getConnection(connStr, "scott", "tiger");
%>

<HTML>
<HEAD> <TITLE> The SqljIterator SQLJSP </TITLE> </HEAD>
<BODY BGCOLOR="#E0FFF0">
    <% String user;
        #sql [dctx] {SELECT user INTO :user FROM dual};
    %>

    <H1> Hello, <%= user %>!    </H1>
    <HR>
    <B> I will use a SQLJ iterator to get employee data
        from EMP table in schema SCOTT..
    </B>
    <P>
    <%
        Empiter emps;
        try {
            #sql [dctx] emps = { SELECT ename, sal, hiredate
                                FROM scott.emp ORDER BY ename};
            if (emps.next()) {
    %>
                <TABLE BORDER=1 BGCOLOR="C0C0C0">
                <TH WIDTH=200 BGCOLOR=white> Employee Name </TH>
                <TH WIDTH=100 BGCOLOR=white> Salary </TH>
                <TR> <TD> <%= emps.ename() %> </TD>
                    <TD> <%= emps.sal() %> </TD>
                </TR>

    <%         while (emps.next()) {
    %>
                <TR> <TD> <%= emps.ename() %> </TD>
                    <TD> <%= emps.sal() %> </TD>
                </TR>
    <% } %>
                </TABLE>
    <% } else { %>
                <P> Sorry, the query returned no rows! </P>
    <% }
                emps.close();
            } catch (SQLException e) { %>
                <P>There was an error doing the query:<PRE> <%= e %> </PRE> <P>
    <% } %>
    </BODY>
</HTML>

```

サーブレット 2.0 環境に globals.jsa を使用するサンプル

この項では、Oracle の globals.jsa メカニズムをサーブレット 2.0 環境で使用して、アプリケーション・フレームワークとアプリケーション・ベースおよびセッション・ベースのイベント処理を提供する方法の例を示します。提供される例は、次のとおりです。

- アプリケーション・イベント用の globals.jsa の例 — [lotto.jsp](#)
- アプリケーションおよびセッション・イベント用の globals.jsa の例 — [index1.jsp](#)
- グローバル宣言用の globals.jsa の例 — [index2.jsp](#)

globals.jsa の使用方法は、5-34 ページの「サーブレット 2.0 に対する OracleJSP のアプリケーションおよびセッションのサポート」を参照してください。

注意： この項の例では、アプリケーションのシャットダウンに基づいた機能があります。多くのサーバーでは、アプリケーションは手動でシャットダウンできません。この場合、globals.jsa はアプリケーション・マーカーとして機能できません。ただし、lotto.jsp ソースまたは globals.jsa ファイルを更新すると、アプリケーションのシャットダウンと再起動を自動的に（developer_mode=false の場合）実行できます（OracleJSP コンテナは常に、実行中のアプリケーションを終了してから、アクティブ・ページを再変換し、再ロードします）。

アプリケーション・イベント用の globals.jsa の例 — [lotto.jsp](#)

このサンプルでは、application_OnStart および application_OnEnd イベント・ハンドラを介した、OracleJSP の globals.jsa のイベント処理を示します。このサンプルでは、番号が 1 日の間にユーザー単位でキャッシュされます。その結果、特定の抽選画面では、ユーザーに 1 組の番号のみが表示されます。このサンプルでは、ユーザーは IP アドレスで識別されます。

application_OnStart および application_OnEnd に対し、キャッシュをアプリケーションのシャットダウン間で永続キャッシュにできるようにコードが書き込まれます。このサンプルでは、キャッシュされたデータは、終了時にファイルに書き込まれ、再起動時にファイルから読み込まれます（キャッシュへの書込みと同じ日にサーバーを再起動する場合）。

lotto.jsp の globals.jsa ファイル

```
<%@ page import="java.util.*, oracle.jsp.jml.*" %>

<jsp:useBean id = "cachedNumbers" class = "java.util.Hashtable" scope = "application" />

<event:application_OnStart>

<%
    Calendar today = Calendar.getInstance();
    application.setAttribute("today", today);
    try {
        FileInputStream fis = new FileInputStream
            (application.getRealPath("/") + File.separator + "lotto.che");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Calendar cacheDay = (Calendar) ois.readObject();
        if (cacheDay.get(Calendar.DAY_OF_YEAR) == today.get(Calendar.DAY_OF_YEAR)) {
            cachedNumbers = (Hashtable) ois.readObject();
            application.setAttribute("cachedNumbers", cachedNumbers);
        }
        ois.close();
    } catch (Exception theE) {
        // catch all -- can't use persistent data
    }
%>

</event:application_OnStart>

<event:application_OnEnd>

<%
    Calendar now = Calendar.getInstance();
    Calendar today = (Calendar) application.getAttribute("today");
    if (cachedNumbers.isEmpty() ||
        now.get(Calendar.DAY_OF_YEAR) > today.get(Calendar.DAY_OF_YEAR)) {
        File f = new File(application.getRealPath("/") + File.separator + "lotto.che");
        if (f.exists()) f.delete();
        return;
    }

    try {
        FileOutputStream fos = new FileOutputStream
            (application.getRealPath("/") + File.separator + "lotto.che");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(today);
        oos.writeObject(cachedNumbers);
        oos.close();
    } catch (Exception theE) {
```

```
        // catch all -- can't use persistent data
    }
%>

</event:application_OnEnd>
```

lotto.jsp のソース

```
<%@ page session = "false" %>
<jsp:useBean id = "picker" class = "oracle.jsp.sample.lottery.LottoPicker" scope = "page" />

<HTML>
<HEAD><TITLE>Lotto Number Generator</TITLE></HEAD>
<BODY BACKGROUND="images/cream.jpg" BGCOLOR="#FFFFFF">
<H1 ALIGN="CENTER"></H1>

<BR>

<!-- <H1 ALIGN="CENTER"> IP: <%= request.getRemoteAddr() %> <BR> -->
<H1 ALIGN="CENTER">Your Specially Picked</H1>
<P ALIGN="CENTER"><IMG SRC="images/winningnumbers.gif" WIDTH="450" HEIGHT="69" ALIGN="BOTTOM"
BORDER="0"></P>
<P>

<P ALIGN="CENTER">
<TABLE ALIGN="CENTER" BORDER="0" CELLPADDING="0" CELLSPACING="0">
<TR>
<%
    int[] picks;
    String identity = request.getRemoteAddr();

    // Make sure its not tomorrow
    Calendar now = Calendar.getInstance();
    Calendar today = (Calendar) application.getAttribute("today");
    if (now.get(Calendar.DAY_OF_YEAR) > today.get(Calendar.DAY_OF_YEAR)) {
        System.out.println("New day...");
        cachedNumbers.clear();
        today = now;
        application.setAttribute("today", today);
    }

    synchronized (cachedNumbers) {
        if ((picks = (int []) cachedNumbers.get(identity)) == null) {
            picks = picker.getPicks();
            cachedNumbers.put(identity, picks);
        }
    }
%>
```

```

        for (int i = 0; i < picks.length; i++) {
%>
        <TD>
        <IMG SRC="images/ball<%= picks[i] %>.gif" WIDTH="68" HEIGHT="76" ALIGN="BOTTOM" BORDER="0">
        </TD>

        <%
        }
%>
</TR>
</TABLE>

</P>

<P ALIGN="CENTER"><BR>
<BR>
<IMG SRC="images/playrespon.gif" WIDTH="120" HEIGHT="73" ALIGN="BOTTOM" BORDER="0">

</BODY>
</HTML>

```

アプリケーションおよびセッション・イベント用の globals.jsa の例 — index1.jsp

この例では、globals.jsa ファイルを使用して、アプリケーションとセッションのライフ・サイクル・イベントを処理しています。アクティブ・セッション数、合計セッション数およびアプリケーション・ページの合計ヒット回数がカウントされます。これらの値は、それぞれ application 有効範囲でメンテナンスされます。アプリケーション・ページ (index1.jsp) により、要求ごとにページ・ヒット数が更新されます。globals.jsa session_OnStart イベント・ハンドラにより、アクティブ・セッション数と合計セッション数が増分されます。globals.jsa session_OnEnd ハンドラでは、アクティブ・セッション数が1ずつ減少します。

ページ出力は単純です。新規セッションが開始すると、セッション・カウンタが出力されます。ページ・カウンタは、要求ごとに出力されます。それぞれの値の最終記録は、globals.jsa application_OnEnd イベント・ハンドラに出力されます。

この例では、次のことに注意してください。

- カウンタ変数の更新時には、値は application 有効範囲でメンテナンスされるため、アクセスを同期化する必要があります。
- 件数値には、OracleJSP の oracle.jsp.jml.JmlNumber 拡張データ型が使用されます。これは、application 有効範囲でのデータ値の使用を簡略化する組み込みの Bean です。(JML の拡張データ型については、5-2 ページの「[JML の拡張データ型](#)」を参照してください。)

index1.jsp の globals.jsa ファイル

```
<%@ taglib uri="oracle.jsp.parse.OpenJspRegisterLib" prefix="jml" %>

<event:application_OnStart>

    <!-- Initializes counts to zero --%>
    <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
    <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
    <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />
    <!-- Consider storing pageCount persistently -- If you do read it here --%>

</event:application_OnStart>

<event:application_OnEnd>

    <!-- Acquire beans --%>
    <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
    <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
    <% application.log("The number of page hits were: " + pageCount.getValue() ); %>
    <% application.log("The number of client sessions were: " + sessionCount.getValue() ); %>
    <!-- Consider storing pageCount persistently -- If you do write it here --%>

</event:application_OnEnd>

<event:session_OnStart>

    <!-- Acquire beans --%>
    <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
    <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

    <%
        synchronized (sessionCount) {
            sessionCount.setValue(sessionCount.getValue() + 1);
        %>

        <br>
        Starting session #: <%= sessionCount.getValue() %> <br>

    <%
    }
    %>

    <%
        synchronized (activeSessions) {
            activeSessions.setValue(activeSessions.getValue() + 1);
        %>

        There are currently <b> <%= activeSessions.getValue() %> </b> active sessions <p>

    <%
    }
    %>
```



```

    %>

</event:session_OnStart>

<event:session_OnEnd>

    <%-- Acquire beans --%>
    <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

    <%
        synchronized (activeSessions) {
            activeSessions.setValue(activeSessions.getValue() - 1);
        }
    %>

</event:session_OnEnd>

```

index1.jsp のソース

```

<%-- Acquire beans --%>
<jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />

<%
    synchronized(pageCount) {
        pageCount.setValue(pageCount.getValue() + 1);
    }
%>

This page has been accessed <b> <%= pageCount.getValue() %> </b> times.
<p>

```

グローバル宣言用の globals.jsa の例 — index2.jsp

この例では、globals.jsa ファイルを使用して変数をグローバルに宣言しています。これは、9-41 ページの「[アプリケーションおよびセッション・イベント用の globals.jsa の例 — index1.jsp](#)」にあるイベント・ハンドラのサンプルに基づいていますが、3つのアプリケーション・カウンタ変数がグローバルに宣言されている点が異なります。（これに対して、オリジナルのイベント・ハンドラのサンプルでは、各イベント・ハンドラと JSP ページ自体で jsp:useBean 文を提供して、アクセスする Bean をローカルに宣言する必要があります。）

Bean をグローバルに宣言すると、すべてのイベント・ハンドラと JSP ページ内で暗黙的に宣言されます。

index2.jsp の globals.jsa ファイル

```
<%-- globally declares variables and initializes them to zero --%>

<jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
<jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
<jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

<event:application_OnStart>

    <%-- Consider storing pageCount persistently -- If you do read it here --%>

</event:application_OnStart>

<event:application_OnEnd>

    <% application.log("The number of page hits were: " + pageCount.getValue() ); %>
    <% application.log("The number of client sessions were: " + sessionCount.getValue() ); %>

    <%-- Consider storing pageCount persistently -- If you do write it here --%>

</event:application_OnEnd>

<event:session_OnStart>

    <%
        synchronized (sessionCount) {
            sessionCount.setValue(sessionCount.getValue() + 1);
        %>

        <br>
        Starting session #: <%= sessionCount.getValue() %> <br>

    <%
        }
    %>

    <%
        synchronized (activeSessions) {
            activeSessions.setValue(activeSessions.getValue() + 1);
        %>

        There are currently <b> <%= activeSessions.getValue() %> </b> active sessions <p>

    <%
        }
    %>

</event:session_OnStart>
```

```
<event:session_OnEnd>

    <%
        synchronized (activeSessions) {
            activeSessions.setValue(activeSessions.getValue() - 1);
        }
    %>

</event:session_OnEnd>
```

index2.jsp のソース

```
<%-- pageCount declared in globals.jsa so active in all pages --%>

<%
    synchronized(pageCount) {
        pageCount.setValue(pageCount.getValue() + 1);
    }
%>

This page has been accessed <b> <%= pageCount.getValue() %> </b> times.

<p>
```

一般的なインストールと構成

この付録では、OracleJSP のインストール、OracleJSP を実行する Web サーバーの構成および OracleJSP の構成に関する一般情報について説明します。技術情報は、Apache/JServ、Sun Microsystems JavaServer Web Developer's Kit (JSWDK) および Tomcat (Sun Microsystems との協同開発による Apache 製) など、一般的な Web サーバーおよびサーバーレット環境に重点を置いています。OracleJSP をサポートする Oracle 環境の場合、インストールおよび構成指示については、該当する製品のドキュメントに言及しています。

Oracle Servlet Engine の場合、変換時構成は OracleJSP の事前変換ユーティリティのオプションを介して処理されます。6-22 ページの「[ojspc 事前変換ツール](#)」を参照してください。

説明するトピックは、次のとおりです。

- [システム要件](#)
- [OracleJSP のインストールと Web サーバーの構成](#)
- [OracleJSP の構成](#)

システム要件

OracleJSP は、pure Java 環境です。OracleJSP をインストールするシステムは、次の最小要件を満たしている必要があります。

オペレーティング・システム： JDK 1.1.x 以上をサポートする OS

Java のバージョン： JDK 1.1.x または 1.2.x
(プラットフォームで使用可能な最新バージョン、可能であれば JDK 1.1.8 以上を推奨)

Java コンパイラ： JDK に付属の標準 javac
(ただし、代替コンパイラを使用することも可能)

Web サーバー： サブレットをサポートする Web サーバー

サブレット環境： サブレット仕様 2.0 以上を実装しているサブレット・コンテナ

注意： サブレット環境に合ったサブレット・ライブラリをシステムにインストールし、Web サーバー構成ファイル内で CLASSPATH に挿入する必要があります。このライブラリには、`javax.servlet.*` パッケージが含まれています。

たとえば、Apache/JServ 環境（Oracle Internet Application Server を含む）の場合は、Sun Microsystems JSDK 2.0 で提供される `jsdk.jar` が必要です。Sun Microsystems JSWDK 環境の場合は、JSWDK 1.0 で提供される `servlet.jar`（サブレットバージョン 2.1）が必要です。Tomcat 環境の場合は、Tomcat 3.1 で提供される `servlet.jar`（サブレットバージョン 2.2）が必要です。JSDK（Java Servlet Developer's Kit）を JSWDK（JavaServer Web Developer's Kit）と混同しないでください。

Web サーバー構成ファイル内の CLASSPATH 設定の詳細は、A-6 ページの「[OracleJSP を実行する Web サーバーおよびサブレット環境の構成](#)」を参照してください。

OracleJSP のインストールと Web サーバーの構成

この項では、各種 JSP 環境向けの OracleJSP のインストールと関連 Web サーバーの構成について説明します。対象となる環境は、次のとおりです。

- Apache/JServ
- Sun Microsystems JSWDK
- Tomcat
- Oracle Servlet Engine (OSE)
- その他の Oracle 環境

ここでは、ターゲット・システムが A-2 ページの「[システム要件](#)」で指定した要件を満たしていることを前提としています。ターゲット・システムは、開発環境でも配布環境でもかまいません。また、次の操作上の知識も必要です。

- Java の実行
- Java コンパイラ（通常は標準の javac）の実行
- HTTP サブプレットの実行

注意：

- この付録の例は UNIX 環境向けですが、基本情報（ディレクトリ名やファイル名など）は他の環境にも適用されます。
 - Web サーバーの構成情報は、一般的な非 Oracle 環境に重点を置いています。Oracle 環境については、特定の製品（Oracle Internet Application Server や Oracle Web-to-go など）のドキュメントを参照してください。
-
-

OracleJSP の必須ファイルとオプション・ファイル

この項では、OracleJSP に必須の JAR および ZIP ファイルと、Oracle JDBC と SQLJ の機能、JML または SQL のカスタム・タグ、カスタムのデータ・アクセス JavaBeans などのオプションを使用するための JAR および ZIP ファイルについて説明します。ファイルの概要に続いて、OracleJSP ファイルを非 Oracle 環境にインストールする方法と、OracleJSP をすでに提供している Oracle 環境をリスト形式で説明します。

必須ファイルは、CLASSPATH にも追加する必要があります。（A-7 ページの「[Web サーバーの CLASSPATH への OracleJSP 関連の JAR および ZIP ファイルの追加](#)」を参照してください。）

ファイルの概要

注意： Oracle8i 製品 CD における前述のファイルの位置については、『Oracle8i Java 開発者ガイド』を参照してください。

OracleJSP に付属する次のファイルは、システムにインストールする必要があります。

- ojsp.jar (OracleJSP)
- xmlparserv2.jar (XML の解析用 — web.xml ディプロイメント・ディスクリプタとすべてのタグ・ライブラリ・ディスクリプタに必須)
- servlet.jar (標準サーブレット・ライブラリ、サーブレットバージョン 2.2)

また、JSP ページで Oracle JSP マークアップ言語 (JML) タグ、SQL タグまたは Database-Access JavaBeans を使用する場合は、次のファイルが必要です。

- ojsputil.jar
- xsu12.jar (JDK 1.2.x の場合) または xsu11.jar (JDK 1.1.x の場合) (OSE 内、またはクライアント側の XML 機能用)

Oracle Servlet Engine 内で実行するには、ojsputil.jar のインストール前または同時に、xsu12.jar または xsu11.jar をインストールする必要があります (これは、Oracle8i の標準インストールでは自動的に処理されます)。ただし、クライアント環境で実行するには、Database-Access JavaBeans で XML 機能 (結果セットを XML 文字列として取得するなど) を使用する場合には、xsu12.jar または xsu11.jar が必要です。xsu12.jar および xsu11.jar ファイルは、Oracle8i リリース 8.1.7 に組み込まれています。

サーブレット・ライブラリに関する注意 OracleJSP 8.1.7 には、サーブレット・ライブラリのバージョン 2.2 が必要で、このバージョンのサーブレット・ライブラリが付属しています。このライブラリには、標準 javax.servlet.* パッケージが含まれています。同様に、Web サーバー環境では、サーブレット・ライブラリの異なるバージョンが提供されます。CLASSPATH では、OracleJSP のバージョンの前に Web サーバーのバージョンを指定する必要があります。詳細は、A-7 ページの「[Web サーバーの CLASSPATH への OracleJSP 関連の JAR および ZIP ファイルの追加](#)」を参照してください。

表 A-1 は、サーブレット・ライブラリのバージョンをまとめたものです。Sun Microsystems JSDK (Java Servlet Developer's Kit) と混同しないでください。

表 A-1 サブレット・ライブラリのバージョン

サブレット・ライブラリのバージョン	ライブラリ・ファイル名	提供製品：
サブレット 2.2	servlet.jar	OracleJSP、Tomcat 3.1
サブレット 2.1	servlet.jar	Sun JSWDK 1.0
サブレット 2.0	jsdk.jar	Sun JSDK 2.0、Apache/JServ でも使用

(Apache/JServ の場合は、jsdk.jar を別途ダウンロードする必要があります。)

以降は、特定のオプション機能や拡張機能を使用する場合にのみ必要なファイルについて説明します。

JDBC 用のファイル (オプション) Oracle JDBC を使用する場合は、次のとおりです。(Oracle SQLJ では Oracle JDBC が使用されるため注意してください。)

- classes12.zip (JDK 1.2.x 環境の場合)

または

- classes111.zip (JDK 1.1.x 環境の場合)

SQLJ 用のファイル (オプション) JSP ページで Oracle8i アクセスに Oracle SQLJ を使用する場合は、次のとおりです。

- translator.zip (SQLJ トランスレータ用、JDK 1.2.x または 1.1.x の場合)

次の適切な SQLJ ランタイムも必要です。

- runtime12.zip (JDK 1.2.x で Oracle JDBC 8.1.7 を使用する場合は)

または

- runtime12ee.zip (JDK 1.2.x Enterprise Edition で Oracle JDBC 8.1.7 を使用する場合は)

または

- runtime11.zip (JDK 1.1.x で Oracle JDBC 8.1.7 を使用する場合は)

または

- runtime.zip (汎用:JDK 1.2.x または 1.1.x で Oracle JDBC の任意のバージョンを使用する場合)

(JDK 1.2.x Enterprise Edition では、SQLJ ISO 仕様に準拠するデータソース・サポートが可能です。)

OracleJSP を提供する Oracle 環境

次のバージョン以降の Oracle 環境では、OracleJSP と Web サーバーまたは Web リスナーが提供されます。

- Oracle Servlet Engine (OSE) 8.1.7
- Oracle Internet Application Server 1.0.0
- Oracle Application Server 4.0.8.2
- Oracle Web-to-go 1.3 (Oracle8i Lite と併用可能)
- Oracle JDeveloper 3.0

前述のどの環境でも、OracleJSP コンポーネントは製品インストールの対象に含まれています。

OSE をターゲットとしている場合は、クライアント側の開発およびテスト環境が必要です。通常は Oracle JDeveloper、または Oracle 以外の開発ツールを使用します。開発環境での予備テストを完了した後に、JSP ページを Oracle8i データベースに配布できます。第 6 章「JSP の変換と配布」を参照してください。

OracleJSP を実行する Web サーバーおよびサーブレット環境の構成

OracleJSP ページを実行できるように Web サーバーを構成するには、次の一般的なステップが必要です。

1. OracleJSP 関連の JAR および ZIP ファイルを Web サーバーの CLASSPATH に追加します。
2. JSP ファイル名拡張子 (.jsp と .JSP、および、オプションで .sqljsp と .SQLJSP) を、OracleJSP コンテナのフロントエンドの Oracle の JspServlet にマップするように、Web サーバーを構成します。

これらのステップはすべての Web サーバー環境に適用されますが、この項では最も一般的な非 Oracle 環境、つまり、Apache/JServ、Sun Microsystems JSWDK および Tomcat に重点を置いています。

Oracle8i JVM 環境で提供される Oracle Servlet Engine は、OracleJSP を実行するようにインストール時に自動的に構成されます。他の Oracle 環境については、手順に違いがあるため、該当する製品のドキュメントを参照してください。（インストールおよび構成手順の大部分は、自動的に実行できます。）

Web サーバーの CLASSPATH への OracleJSP 関連の JAR および ZIP ファイルの追加

Web サーバーの CLASSPATH を更新し、OracleJSP に必須の JAR および ZIP ファイルを正しい順序で追加する必要があります。（特に、後述するように、サーブレットバージョン 2.2 の `servlet.jar` を CLASSPATH に挿入する位置に注意する必要があります。）該当するファイルは、次のとおりです。

- `ojjsp.jar`
- `xmlparserv2.jar`
- `servlet.jar` (サーブレット バージョン 2.2)
(OracleJSP に付属する `servlet.jar` は、Tomcat 3.1 に付属する `servlet.jar` と同一です。)
- `ojjsputil.jar` (オプション。JML タグ、SQL タグおよび Database-Access JavaBeans 用)
- `xsu12.jar` (JDK 1.2.x の場合) または `xsu11.jar` (JDK 1.1.x の場合) (オプション。JML タグ、SQL タグおよび Database-Access JavaBeans 用。A-4 ページの「[ファイルの概要](#)」を参照)
- その他、必要に応じて JDBC および SQLJ 用のオプションの ZIP および JAR ファイル (A-4 ページの「[ファイルの概要](#)」を参照)

重要： OracleJSP で `javac` (または `javac`cmd 構成パラメータの設定によっては代替 Java コンパイラ) を検索できることも確認する必要があります。JDK 1.1.x 環境の `javac` の場合は、JDK の `classes.zip` ファイルを Web サーバーの CLASSPATH に指定する必要があります。JDK 1.2.x 環境の `javac` の場合は、JDK の `tools.jar` ファイルを Web サーバーの CLASSPATH に指定する必要があります。

Apache/JServ 環境 Apache/JServ 環境の場合は、JServ の `conf` ディレクトリにある `jserv.properties` ファイルに適切な `wrapper.classpath` コマンドを追加します。`jsdk.jar` はすでに CLASSPATH に存在しています。このファイルは Sun Microsystems JSDK 2.0 に付属しており、Apache/JServ に必要なサーブレットバージョン 2.0 の `javax.servlet.*` パッケージを提供します。JDK 環境用のファイルも、すでに CLASSPATH に存在します。

次の例 (UNIX のディレクトリ・パスを使用する場合) では、OracleJSP、JDBC および SQLJ 用のファイルを挿入しています。([`Oracle_Home`] を実際の Oracle ホームのパスに置き換えてください。)

```
# servlet 2.0 APIs (required by Apache/JServ, from Sun JSDK 2.0):
wrapper.classpath=jsdk2.0/lib/jsdk.jar
#
```

```
# servlet 2.2 APIs (required and provided by OracleJSP):
wrapper.classpath=[Oracle_Home]/ojsp/lib/servlet.jar
# OracleJSP packages:
wrapper.classpath=[Oracle_Home]/ojsp/lib/ojsp.jar
wrapper.classpath=[Oracle_Home]/ojsp/lib/ojsputil.jar
# XML parser (used for servlet 2.2 web deployment descriptor):
wrapper.classpath=[Oracle_Home]/ojsp/lib/xmlparserv2.jar
# JDBC libraries for Oracle database access (JDK 1.2.x environment):
wrapper.classpath=[Oracle_Home]/ojsp/lib/classes12.zip
# SQLJ translator (optional):
wrapper.classpath=[Oracle_Home]/ojsp/lib/translator.zip
# SQLJ runtime (optional) (for JDK 1.2.x enterprise edition):
wrapper.classpath=[Oracle_Home]/ojsp/lib/runtime12.zip
```

重要： Apache/JServ 環境の場合は、CLASSPATH 内で jsdk.jar を servlet.jar の前に挿入する必要があります。

次の useBean コマンドを実行した場合の例を考えます。

```
<jsp:useBean id="queryBean" class="mybeans.JDBCQueryBean" scope="session" />
```

次の wrapper.classpath コマンドを jserv.properties ファイルに追加できます（この例は、Windows NT 環境の場合です）。

```
wrapper.classpath=D:\¥Apache¥Apache1.3.9¥beans¥
```

また、JDBCQueryBean.class の位置を次のように識別する必要があります。

```
D:\¥Apache¥Apache1.3.9¥beans¥mybeans¥JDBCQueryBean.class
```

JSWDK 環境 jswdk-1.0 ルート・ディレクトリ内で startserver スクリプトを更新し、OracleJSP ファイルを環境変数 jspJars に追加します。UNIX の場合はコロン (:)、Windows NT の場合はセミコロン (;) など、オペレーティング・システムの適切なディレクトリ構文とセパレータ文字を使用して、それを最後に表示される .jar ファイルに追加します。次に例を示します。

```
jspJars=./lib/jspengine.jar:./lib/ojsp.jar:./lib/ojsputil.jar
```

この例（UNIX 構文）では、JAR ファイルが jswdk-1.0 ルート・ディレクトリの lib サブディレクトリにあるとします。

同様に、startserver スクリプトを更新し、次のように、環境変数 miscJars 内で必須ファイルを追加指定します。

```
miscJars=./lib/xml.jar:./lib/xmlparserv2.jar:./lib/servlet.jar
```

この例（UNIX 構文）でも、各ファイルは `jswdk-1.0` ルート・ディレクトリの `lib` サブディレクトリにあるとします。

重要： JSWDK 環境では、サーブレット バージョン 2.1 の `servlet.jar`（Sun JSWDK 1.0 に付属）を、CLASSPATH 内でサーブレット バージョン 2.2 の `servlet.jar`（OracleJSP に付属）より前に挿入する必要があります。

サーブレット バージョン 2.1 は、通常は環境変数 `jsdkJars` 内に挿入されています。CLASSPATH 全体は、`jsdkJars`、`jspJars` および `miscJars` など、様々な環境変数 `xxxJars` の組合せで構成されています。
`startserver` スクリプトを調べて、`miscJars` が CLASSPATH 内で `jsdkJars` の後に追加されていることを確認します。

Tomcat 環境 Tomcat の場合、CLASSPATH にファイルを追加する手順は、この項で説明する他のサーブレット環境に比べて、オペレーティング・システムへの依存度が高くなっています。

UNIX オペレーティング・システムの場合は、OracleJSP の JAR および ZIP ファイルを `[TOMCAT_HOME]/lib` ディレクトリにコピーします。このディレクトリは、Tomcat の CLASSPATH に自動的に挿入されます。

Windows NT オペレーティング・システムの場合は、`[TOMCAT_HOME]¥bin` ディレクトリにある `tomcat.bat` ファイルを更新し、各 OracleJSP ファイルを環境変数 CLASSPATH に個別に追加します。次の例は、各ファイルを `[TOMCAT_HOME]¥lib` ディレクトリにコピーした場合です。

```
set CLASSPATH=%CLASSPATH%;%TOMCAT_HOME%¥lib¥ojsp.jar;%TOMCAT_HOME%¥lib¥ojsputil.jar
```

サーブレット バージョン 2.2 の `servlet.jar`（OracleJSP に付属するものと同一バージョン）は、すでに Tomcat とともに挿入されているため、考慮する必要はありません。

Oracle JspServlet への JSP ファイル名拡張子のマッピング

Web サーバーを次のように構成する必要があります。

- 適切なファイル名拡張子を JSP ページとして認識させる
.`jsp` と `.JSP` をマップします。JSP ページで Oracle SQLJ が使用される場合は、`.sqljsp` と `.SQLJSP` もマップします。
- JSP ページの処理を開始するサーブレットを検索して実行する

OracleJSP では、これは `oracle.jsp.JspServlet` で、OracleJSP コンテナのフロントエンドと見なすことができます。

重要： 前述の構成では、OracleJSP により `.jsp` と `.JSP` のどちらのファイル名拡張子を使用するページ参照もサポートされますが、大 / 小文字区別のある環境では参照の大 / 小文字が実際のファイル名と一致する必要があります。ファイル名が `file.jsp` であれば、そのとおりに参照できますが、`file.JSP` としては参照できません。ファイル名が `file.JSP` であれば、そのとおりに参照できますが、`file.jsp` としては参照できません（これは、`.sqljsp` と `.SQLJSP` の場合も同様です）。

Apache/JServ 環境 Apache/JServ 環境では、各 JSP ファイル名拡張子を Oracle の JspServlet に一度にマップできます。JServ の `conf` ディレクトリ内で構成ファイル `jserv.conf` または `mod_jserv.conf` を更新し、このマッピングを実行する `ApJservAction` コマンドを追加します。

（旧バージョンの場合は、かわりに Apache の `conf` ディレクトリにある `httpd.conf` ファイルを更新する必要があります。新バージョンの場合は、`jserv.conf` または `mod_jserv.conf` ファイルが実行時に `httpd.conf` に「挿入」されます。`httpd.conf` ファイルを調べて、どちらが挿入されているかを確認してください。）

次に例を示します（UNIX 構文を使用する場合）。

```
# Map file name extensions (.sqljsp and .SQLJSP are optional).
ApJservAction .jsp /servlets/oracle.jsp.JspServlet
ApJservAction .JSP /servlets/oracle.jsp.JspServlet
ApJservAction .sqljsp /servlets/oracle.jsp.JspServlet
ApJservAction .SQLJSP /servlets/oracle.jsp.JspServlet
```

このコマンドで `oracle.jsp.JspServlet` に使用するパスは、ファイル・システム内のリテラルのディレクトリ・パスではありません。指定するパスは、サーブレット・ゾーンのマウント方法、ゾーン・プロパティ・ファイル名およびサーブレットのリポジトリとして指定されているファイル・システム・ディレクトリなど、Apache/JServ のサーブレット構成に応じて異なります。（「サーブレット・ゾーン」は、「サーブレット・コンテキスト」と同じ概念を表す Apache/JServ 用語です。）詳細は、Apache/JServ のドキュメントを参照してください。

JSWDK 環境 JSWDK 環境で、各 JSP ファイル名拡張子を Oracle の JspServlet にマップするには、2つのステップが必要です。

最初のステップでは、各サーブレット・コンテキストの `WEB-INF` ディレクトリにある `mappings.properties` ファイルを更新し、JSP ファイル拡張子を定義します。そのためには、次のコマンドを使用します。

```
# Map JSP file name extensions (.sqljsp and .SQLJSP are optional).
.jsp=jsp
.JSP=jsp
.sqljsp=jsp
.SQLJSP=jsp
```

2 番目のステップでは、各サーブレット・コンテキストの WEB-INF ディレクトリにある `servlet.properties` ファイルを更新し、JSP 処理を開始するサーブレットとして Oracle `JspServlet` を定義します。また、JSP 参照実装用に以前に定義したマッピングをコメント解除してください。そのためには、次のコマンドを実行します。

```
#jsp.code=com.sun.jsp.runtime.JspServlet (replacing this with Oracle)
jsp.code=oracle.jsp.JspServlet
```

Tomcat 環境 Tomcat 環境では、各 JSP ファイル名拡張子を Oracle の `JspServlet` に一度にマップできます。`web.xml` ファイルのサーブレット・マッピング・セクションを次のように更新します。

注意： グローバルな `web.xml` ファイルは、`[TOMCAT_HOME]/conf` ディレクトリにあります。このファイル内の設定を、特定のアプリケーションについてオーバーライドするには、特定のアプリケーション・ルートの WEB-INF ディレクトリにある `web.xml` ファイルを更新します。

```
# Map file name extensions (.sqljsp and .SQLJSP are optional).
```

```
<servlet-mapping>
  <servlet-name>
    oracle.jsp.JspServlet
  </servlet-name>
  <url-pattern>
    *.jsp
  </url-pattern>
</servlet-mapping>
```

```
<servlet-mapping>
  <servlet-name>
    oracle.jsp.JspServlet
  </servlet-name>
  <url-pattern>
    *.JSP
  </url-pattern>
</servlet-mapping>
```

```
<servlet-mapping>
  <servlet-name>
    oracle.jsp.JspServlet
  </servlet-name>
  <url-pattern>
    *.sqljsp
```

```
</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>
    oracle.jsp.JspServlet
  </servlet-name>
  <url-pattern>
    *.SQLJSP
  </url-pattern>
</servlet-mapping>
```

オプションで、次のように `oracle.jsp.JspServlet` クラス名の別名を設定できます。

```
<servlet>
  <servlet-name>
    ojsp
  </servlet-name>
  <servlet-class>
    oracle.jsp.JspServlet
  </servlet-class>
  ...
</servlet>
```

この別名を設定すると、次のように、他の設定にクラス名ではなく `ojsp` を使用できます。

```
<servlet-mapping>
  <servlet-name>
    ojsp
  </servlet-name>
  <url-pattern>
    *.jsp
  </url-pattern>
</servlet-mapping>
```


OracleJSP の構成

OracleJSP のフロントエンド・サーブレット JspServlet では、OracleJSP の操作を制御する多数の構成パラメータがサポートされます。各パラメータは、JspServlet のサーブレット初期化パラメータとして設定します。その設定方法は、使用中の Web サーバーおよびサーブレット環境に応じて異なります。

この項では、OracleJSP の構成パラメータと、最も一般的な Web サーバーおよびサーブレット環境での設定方法について説明します。

この種のパラメータのうち、OracleJSP が付属している Oracle 製品に重要なものは限られています。また、その設定方法も製品ごとに異なる場合があります。詳細は、各製品のドキュメントを参照してください。

通常、Oracle Servlet Engine に適用される構成設定は、OracleJSP の事前変換ツール (ojspc) の等価オプションとしてサポートされます。OSE では、JSP ページの変換や実行に Oracle の JspServlet は使用されません。

OracleJSP の構成パラメータ（非 OSE）

この項では、Apache/JServ、Sun Microsystems JSWDK または Tomcat などの環境向けに、Oracle の JspServlet でサポートされる構成パラメータについて説明します。（Oracle Internet Application Server では、Apache/JServ 環境が使用されるため注意してください。）

Oracle Servlet Engine 環境では、等価の構成機能の一部が等価の ojspc オプションを介してサポートされます。

構成パラメータの概要表

表 A-2 は、Oracle の JspServlet（OracleJSP コンテナのフロントエンド）でサポートされる構成パラメータの概要を示しています。この表には、パラメータごとに次の情報が含まれています。

- ページの変換中または実行中に使用されるかどうか。
- 通常、開発環境または配布環境、あるいはその両方に重要かどうか。
- Oracle Servlet Engine（JspServlet を使用しない環境）をターゲットとするページに対する等価の ojspc 変換オプション。

OSE では、ランタイム構成パラメータはサポートされません。

次の点に注意してください。

- パラメータ `bypass_source`、`emit_debuginfo`、`external_resource`、`javaccmd` および `sqljcmd` は、リリース 1.1.0.0.0 以前の OracleJSP ではサポートされていませんでした。
- パラメータ `alias_translation` は、Apache/JServ 環境専用です。

- パラメータ `session_sharing` は、`globals.jsa` 専用です（Apache/JServ などのサーブレット 2.0 環境の場合）。

注意：

- `ojspc` のオプションの詳細は、6-22 ページの「[ojspc 事前変換ツール](#)」を参照してください。
- 実行時と変換時の違いは、リアルタイムの変換環境では特に重要ではありませんが、OSE に関しては重要な場合があります。

表 A-2 OracleJSP の構成パラメータ

パラメータ	関連する ojspc のオプション	説明	デフォルト	JSP の変換または実行のどちらに使用するか	開発環境または配布環境のどちらで使用するか
alias_translation (Apache 固有)	該当なし	ブール。true に設定すると、JSP ページ参照についてディレクトリ別名に関する Apache/JServ の制限事項を回避できます。	false	実行	開発および配布
bypass_source	該当なし	ブール。true に設定すると、OracleJSP では .jsp ソースに対する FileNotFound 例外が無視されます。ソースが使用できない場合は、事前変換およびコンパイル済みのコードを使用します。	false	実行	配布（JDeveloper でも使用）
classpath	-addclasspath (関連はあるが異なる機能)	OracleJSP クラスのロード用の追加の CLASSPATH エントリです。	null (追加のパスなし)	変換または実行	開発および配布
developer_mode	該当なし	ブール。false に設定すると、ページが要求されたときにタイムスタンプはチェックされず、再変換が必要かどうかは検査されません。	true	実行	開発および配布

表 A-2 OracleJSP の構成パラメータ (続き)

パラメータ	関連する ojspc のオプション	説明	デフォルト	JSP の変換または実行のどちらに使用するか	開発環境または配布環境のどちらで使用するか
emit_debuginfo	-debug	ブール。true に設定すると、デバッグ用に元の .jsp ファイルへのライン・マップが生成されます。	false	変換	開発
external_resource	-extres	ブール。true に設定すると、OracleJSP により、変換中にページの静的コンテンツがすべて、別の Java リソース・ファイルに格納されます。	false	変換	開発および配布
javaccmd	-noCompile	Java コンパイラのコマンドライン — javac オプション、または別の JVM 内で実行される代替 Java コンパイラ (null は、JDK javac をデフォルト・オプションで使用することを意味します)。	null	変換	開発および配布
page_repository_root	-srcdir -d	OracleJSP で JSP ページのロードと生成に使用する代替ルート・ディレクトリ (完全修飾パス)。	null (デフォルト・ルートを使用)	変換または実行	開発および配布
session_sharing (globals.jsa で使用)	該当なし	ブール。true に設定すると、JSP セッション・データは、globals.jsa を使用してアプリケーションの基礎となるサーブレット・セッションに伝播されます。	true	実行	開発および配布

表 A-2 OracleJSP の構成パラメータ（続き）

パラメータ	関連する ojspc のオプション	説明	デフォルト	JSP の変換または 実行のどちらに 使用するか	開発環境または 配布環境のどちら で使用するか
sqljcmd	該当なし	SQLJ のコマンドライン —sqlj オプション、ま たは別の JVM 内で実行 される代替 SQLJ トラン スレータ（null は、 OracleJSP に付属する Oracle SQLJ バージョン をデフォルトのオプショ ン設定で使用することを 意味します）。	null	変換	開発および配布
translate_params	該当なし	ブール。true に設定す ると、マルチバイト・ コード化を実行しない サブリット・コンテナ がオーバーライドされま す。	false	実行	開発および配布
unsafe_reload	該当なし	ブール。true に設定す ると、JSP ページが再変 換され再ロードされて も、アプリケーションと セッションはそのたびに 再起動されません。	false	実行	開発

構成パラメータの説明

この項では、OracleJSP の構成パラメータについて詳しく説明します。

alias_translation（ブール。OracleJSP のデフォルト : false）（Apache 固有）

このパラメータにより、OracleJSP では Apache/JServ でのディレクトリ別名の処理方法に
対する制限事項を回避できます。現行の制限事項の詳細は、4-34 ページの「[ディレクトリ別
名の変換](#)」を参照してください。

httpd.conf ディレクトリ別名コマンド（次の例など）を Apache/JServ サブリット環境
で正常に機能させるには、alias_translation を true に設定する必要があります。

```
Alias /icons/ "/apache/apache139/icons/"
```

bypass_source (ブール。OracleJSP のデフォルト: false)

通常、JSP ページが要求された場合に、OracleJSP で対応する .jsp ソース・ファイルが見つからなければ、ページ実装クラスが見つかった場合でも、FileNotFound 例外が発生します。(デフォルトでは、OracleJSP でページ・ソースがチェックされ、ページ実装クラスの生成後に変更があったかどうかを検査されるためです。)

OracleJSP でページ・ソースが見つからない場合でも、処理を進行させ、ページ実装クラスを実行するには、このパラメータを true に設定します。

bypass_source が有効化されている場合は、ソースが使用可能で必要であれば、OracleJSP で再変換の要否がチェックされます (必要かどうかを決定する要素の 1 つは、developer_mode パラメータの設定です)。

注意:

- bypass_source オプションは、ソースではなく生成されるクラスのみが含まれる配布環境で役立ちます。(関連する説明については、6-65 ページの「[バイナリ・ファイルのみの配布](#)」を参照してください。)
 - OracleJDeveloper では、JSP ソースをファイルに保存する前に JSP ページを変換して実行できるように、bypass_source が有効化されます。
-
-

classpath (完全修飾パス。OracleJSP のデフォルト: null)

このパラメータを使用して、JSP ページの変換、コンパイルまたは実行中に使用される OracleJSP のデフォルトの classpath に classpath エントリを追加します。OracleJSP の CLASSPATH とクラス・ローダーの詳細は、4-19 ページの「[CLASSPATH とクラス・ローダーの問題 \(非 OSE のみ\)](#)」を参照してください。

正確な構文は、Web サーバー環境とオペレーティング・システムに応じて異なります。例については、A-23 ページの「[OracleJSP の構成パラメータ設定](#)」を参照してください。

OracleJSP では、通常クラスが独自の CLASSPATH (この classpath パラメータからのエントリなど)、システムの CLASSPATH、Web サーバーの CLASSPATH、ページ・リポジトリおよび JSP アプリケーションのルート・ディレクトリに対する事前定義済みの相対位置からロードされます。

classpath で指定したパスを介してロードされるクラスは、システムのクラス・ローダーではなく JSP のクラス・ローダーによりロードされるため注意してください。JSP の実行中には、JSP のクラス・ローダーによりロードされたクラスと、システムのクラス・ローダーや他のクラス・ローダーによりロードされたクラスの間では、相互にアクセスできません。

注意：

- OracleJSP のランタイム自動クラス再ロード機能が適用されるのは、OracleJSP の CLASSPATH 内のクラスのみです。これには、この classpath パラメータを介して指定したパスが含まれます。（この機能については、4-24 ページの「[クラスの動的再ロード](#)」を参照してください。）
 - Oracle Servlet Engine 内で実行するページを事前変換する場合は、ojspc の -addclasspath オプションにより、一部関連するが異なる機能が提供されます。6-26 ページの「[ojspc のオプションの説明](#)」を参照してください。
-

developer_mode (ブール。OracleJSP のデフォルト: true)

このフラグを false に設定し、OracleJSP に対して、ページが要求されたときに、ページ実装クラスのタイムスタンプを .jsp ソース・ファイルのタイムスタンプと定期的に比較しないように指示します。通常、OracleJSP では、ページ実装クラスの生成後にソースが変更されたかどうか、毎回チェックされます。変更された場合は、OracleJSP によりページが再変換されます。developer_mode=false に設定すると、ページまたはアプリケーションに関する初回要求時のみ、OracleJSP でチェックされます。後続の要求では、単に生成されたページ実装クラスが再実行されます。

通常、特にコード変更がなく、パフォーマンスが重要である配布環境では、developer_mode を false に設定することをお勧めします。

emit_debuginfo (ブール。OracleJSP のデフォルト: false) (開発者用)

このフラグを true に設定し、OracleJSP に対して、デバッグ用に元の .jsp ファイルへのライン・マップを生成するように指示します。このフラグを true に設定しない場合は、生成されたページ実装クラスにマップされます。

注意：

- Oracle JDeveloper では、emit_debuginfo が有効化されます。
 - Oracle Servlet Engine 内で実行するページを事前変換する場合は、ojspc の -debug オプションが等価になります。6-26 ページの「[ojspc のオプションの説明](#)」を参照してください。
-

external_resource (ブール。OracleJSP のデフォルト: false)

このフラグを true に設定し、OracleJSP トランスレータに対して、生成された静的コンテンツ (静的 HTML コードを出力する Java 出力コマンド) を、生成されたページ実装クラスのサービス・メソッドではなく、Java リソース・ファイルに格納するように指示します。

リソース・ファイル名は、JSP ページ名に基づいています。リリース 8.1.7 では、JSP 名と同じ名前になりますが、.res 接尾辞が付いています。(たとえば、MyPage.jsp を変換すると、標準出力に加えて MyPage.res が作成されます。ただし、正確な実装は将来のリリースで変更されることがあります。)

リソース・ファイルは、生成されたクラス・ファイルと同じディレクトリに格納されます。

1 ページに大量の静的コンテンツが含まれている場合は、この技法を使用すると変換が高速化され、ページの実行も高速化されることがあります。極端な場合は、サービス・メソッドが Java VM で適用される 64KB のメソッド・サイズ制限を超えなくなることもあります。詳細は、4-12 ページの「[JSP ページの大量の静的コンテンツに関する回避策](#)」を参照してください。

注意： Oracle Servlet Engine 内で実行するページを事前変換する場合は、ojspc の -extres オプションが等価です。

ojspc -hotload オプションも関連があります。このオプションでは、Oracle8i へのホットロードを可能にする追加ステップとともに -extres 機能が実行されます。6-26 ページの「[ojspc のオプションの説明](#)」を参照してください。

javacmd (コンパイラの実行可能ファイル。OracleJSP のデフォルト: null)

このパラメータは、次の 2 つの状況に役立ちます。

- javac コマンドライン・オプションを設定する場合 (ただし、通常はデフォルト設定で十分です)。
- javac 以外のコンパイラを使用する場合 (オプションでコマンドライン・オプションを含みます)。

代替コンパイラを指定すると、OracleJSP では、OracleJSP の JVM 内で JDK のデフォルト・コンパイラが起動されるかわりに、その実行可能ファイルが別の JVM 内で別個のプロセスとして起動されます。実行可能ファイルのフルパスを指定する方法と、実行可能ファイルのみを指定して OracleJSP にシステム・パス内で検索させる方法があります。

次の javacmd 設定では、javac -verbose オプションが有効化されます。

```
javacmd=javac -verbose
```

正確な構文は、サーブレット環境によって異なります。A-23 ページの「[OracleJSP の構成パラメータ設定](#)」を参照してください。

注意：

- 指定した Java コンパイラを CLASSPATH にインストールし、フロントエンド・ユーティリティ（該当する場合）はシステム・パスにインストールする必要があります。
 - Oracle Servlet Engine 内で実行するページを事前変換する場合は、ojspc の -noCompile オプションで同様の機能を実行できます。その場合、javac ではコンパイルされないため、必要なコンパイラを使用して変換済みクラスを手動でコンパイルできます。6-26 ページの「[ojspc のオプションの説明](#)」を参照してください。
-
-

page_repository_root（完全修飾ディレクトリ・パス。OracleJSP のデフォルト：null）

OracleJSP では、変換済み JSP ページの生成やロードに、Web サーバーのドキュメント・リポジトリが使用されます。オンデマンド変換の使用例の場合、デフォルトのルート・ディレクトリは Web サーバーのドキュメント・ルート・ディレクトリ（Apache/JServ の場合）、またはページが属するアプリケーションのサブレット・コンテキストのルート・ディレクトリです。JSP ページのソースは、ルート・ディレクトリまたはそのサブディレクトリに存在します。生成されたファイルは、_pages サブディレクトリまたは対応するサブディレクトリに書き込まれます。

OracleJSP に対して異なるルート・ディレクトリを使用するように指示するには、page_repository_root オプションを設定します。

ルート・ディレクトリおよび _pages サブディレクトリに対するファイルの相対位置については、6-7 ページの「[OracleJSP トランスレータの出力ファイルの位置](#)」を参照してください。

注意：

- 指定したディレクトリ、_pages サブディレクトリおよび該当するすべてのサブディレクトリは、存在しない場合自動的に作成されます。
 - Oracle Servlet Engine 内で実行するページを事前変換する場合は、ojspc のオプション -srcdir および -d で関連機能が提供されます。6-26 ページの「[ojspc のオプションの説明](#)」を参照してください。
-
-

session_sharing（ブール。OracleJSP のデフォルト：true）（globals.jsa で使用）

サブレット 2.0 環境など、アプリケーションに globals.jsa ファイルが使用される場合、各 JSP ページには個別の JSP セッション・ラッパーが使用されます。このラッパーは、サブレット・コンテナにより提供される単一のサブレット全体のセッション・オブジェクトに連結されています。

この状況では、`session_sharing` パラメータをデフォルトの `true` に設定すると、JSP セッションのデータは基礎となるサーブレット・セッションに伝播されます。これにより、アプリケーション内のサーブレットは、同一アプリケーション内の JSP ページのセッション・データにアクセスできます。

`session_sharing` が `false` の場合（ほとんどの JSP 実装での標準動作）、JSP セッションのデータはサーブレット・セッションに伝播されません。その結果、アプリケーションのサーブレットは JSP セッションのデータにアクセスできなくなります。

このパラメータは、`globals.jsa` を使用しない場合は無意味です。`globals.jsa` の詳細は、5-35 ページの「[globals.jsa の機能の概要](#)」を参照してください。

sqljcmd (SQLJ トランスレータの実行可能ファイル。OracleJSP のデフォルト: `null`)

このパラメータは、次の状況に役立ちます。

- SQLJ のコマンドライン・オプションを設定する場合
- OracleJSP に付属しているものとは異なる SQLJ トランスレータ（または少なくともバージョンが異なる）を使用する場合
- SQLJ を OracleJSP とは異なるプロセス内で実行する場合

SQLJ トランスレータの実行可能ファイルを指定すると、OracleJSP では、OracleJSP の JVM 内でデフォルトの SQLJ トランスレータが起動されるかわりに、その実行可能ファイルが別の JVM 内で別個のプロセスとして起動されます。

実行可能ファイルのフルパスを指定する方法と、実行可能ファイルのみを指定して OracleJSP にシステム・パス内で検索させる方法があります。

次の例は、`sqljcmd` の設定を示しています。

```
sqljcmd=sqlj -user=scott/tiger -ser2class
```

（正確な構文は、サーブレット環境によって異なります。A-23 ページの「[OracleJSP の構成パラメータ設定](#)」を参照してください。）

注意：

- 適切な SQLJ ファイルを CLASSPATH に含めて、フロントエンド・ユーティリティ（この例の `sqlj` など）をシステム・パスにインストールする必要があります。（Oracle SQLJ の場合は、`translator.zip` と適切な SQLJ ランタイムの ZIP ファイルを CLASSPATH に指定する必要があります。A-4 ページの「[ファイルの概要](#)」を参照してください。）
 - 多くの OracleJSP 開発者は、JSP ページで SQLJ コードを使用する場合に、（他の SQLJ 製品ではなく）Oracle SQLJ を使用します。ただし、Oracle SQLJ の異なるバージョンを使用する場合（Oracle8i ドライバのかわりに Oracle JDBC 8.0.x/7.3.x ドライバを使用する場合など）や、SQLJ オプションを設定する場合などは、このオプションが役立ちます。
-
-

translate_params（ブール。OracleJSP のデフォルト：`false`）

このフラグを `true` に設定すると、マルチバイト（NLS）の要求パラメータや Bean のプロパティ設定をコード化しないサブレット・コンテナがオーバーライドされます。この設定では、OracleJSP により要求パラメータと Bean のプロパティ設定がコード化されます。`true` に設定しない場合は、OracleJSP ではサブレット・コンテナからのパラメータがそのまま戻されます。

Oracle Servlet Engine ではランタイム構成パラメータがサポートされないため、OSE 環境の場合は `translate_params` を設定できません。回避策については、8-6 ページの「[translate_params 構成パラメータの等価コード](#)」を参照してください。

使用しない状況などの、`translate_params` の機能と使用方法の詳細は、8-4 ページの「[マルチバイト・パラメータのコード化に対する OracleJSP の拡張サポート](#)」を参照してください。

unsafe_reload（ブール。OracleJSP のデフォルト：`false`）（開発者用）

デフォルトでは、JSP ページが動的に再変換され、再ロードされるたびに、OracleJSP によりアプリケーションとセッションが再起動されます（JSP ページの再変換と再ロードは、JSP トランスレータにより、対応するページ実装クラスより新しいタイムスタンプを持つ `.jsp` ソース・ファイルが検出されると発生します）。

OracleJSP に対して、動的な再変換と再ロードの後にアプリケーションを再起動しないように指示するには、このパラメータを `true` に設定します。これにより、既存のセッションが無効になることがなくなります。

特定の JSP ページに対して `developer_mode` が `false` に設定されている場合（OracleJSP では初回要求後は再変換されなくなります）、そのページに対する初回要求後は、このパラメータは無効です。

重要： このパラメータは開発者専用 intent されており、配布環境に使用することはお薦めしません。

OracleJSP の構成パラメータ設定

前項 (A-13 ページの「[OracleJSP の構成パラメータ \(非 OSE\)](#)」) で説明した JSP 構成パラメータの設定方法は、Web サーバーおよびサーブレット環境によって異なります。

非 Oracle 環境では、プロパティ・ファイルまたは同様の機能を介して構成パラメータ設定がサポートされます。

Oracle8i JVM 環境で提供される Oracle Servlet Engine では、OracleJSP の構成パラメータは直接サポートされません (これは JspServlet を使用しないためです)。ただし、一部の交換パラメータ設定には、OracleJSP トランスレータの等価オプションがあります。これらのオプションについては、A-13 ページの「[構成パラメータの概要表](#)」を参照してください。

OracleJSP をサポートする他の Oracle 製品には、構成設定用の独自メカニズムがあります。該当製品のドキュメントを参照してください。

以降は、Apache/JServ、Sun Microsystems JSWDK および Tomcat 環境における構成パラメータの設定方法について説明します。

Apache/JServ での OracleJSP パラメータの設定

Apache/JServ 環境の Web アプリケーションごとに、ゾーン・プロパティ・ファイルと呼ばれる独自のプロパティ・ファイルがあります。Apache 用語のゾーンは、実際にはサーブレット・コンテキストと同じ意味になります。

ゾーン・プロパティ・ファイルの名前は、ゾーンのマウント方法によって決まります。(ゾーンとマウントの詳細は、Apache/JServ のドキュメントを参照してください。)

Apache/JServ 環境で OracleJSP の構成パラメータを設定するには、次の例 (UNIX 構文の場合) のように、アプリケーションのゾーン・プロパティ・ファイル内で JspServlet の `initArgs` プロパティを設定します。

```
servlet.oracle.jsp.JspServlet.initArgs=developer_mode=false,  
sqljcmd=sqlj -user=scott/tiger -ser2class=true,classpath=/mydir/myapp.jar
```

(前述の例は、1 行のコードを折り返したものです。)

サーブレット・パス `servlet.oracle.jsp.JspServlet` も、ゾーンのマウント方法によって決まります。リテラルのディレクトリ・パスを表すものではありません。

注意： `initArgs` のパラメータはカンマで区切られるため、パラメータ設定にはカンマを使用できません。ただし、空白や他の特殊文字（この例の「=」など）はエラーになりません。

JSWDK での OracleJSP パラメータの設定

JSWDK 環境で OracleJSP の構成パラメータを設定するには、次の例（UNIX 構文を使用する場合）のように、アプリケーションのサーブレット・コンテキストの `WEB-INF` ディレクトリにある `servlet.properties` ファイル内で、`jsp.initparams` プロパティを設定します。

```
jsp.initparams=developer_mode=false,classpath=/mydir/myapp.jar
```

注意： `initparams` のパラメータはカンマで区切られるため、パラメータ設定にはカンマを使用できません。ただし、空白や他の特殊文字はエラーになりません。

Tomcat での OracleJSP パラメータの設定

Tomcat 環境で OracleJSP の構成パラメータを設定するには、次のように `web.xml` ファイルに `init-param` エントリを追加します。

注意： グローバルな `web.xml` ファイルは、`[TOMCAT_HOME]/conf` ディレクトリにあります。このファイル内の設定を特定のアプリケーションについてオーバーライドするには、特定のアプリケーション・ルートの `WEB-INF` ディレクトリにある `web.xml` ファイルを更新します。

```
<servlet>
  <init-param>
    <param-name>
      developer_mode
    </param-name>
    <param-value>
      true
    </param-value>
  </init-param>
  <init-param>
    <param-name>
      external_resource
    </param-name>
    <param-value>
      true
  </init-param>
</servlet>
```

```
        </param-value>
    </init-param>
    <init-param>
        <param-name>
            javaccmd
        </param-name>
        <param-value>
            javac -verbose
        </param-value>
    </init-param>
</servlet>
```

Oracle Servlet Engine の JSP 構成

Oracle Servlet Engine では OracleJSP の `JspServlet` フロントエンドを使用しないため、OracleJSP 構成設定用に他のメカニズムが必要です。

該当する変換時構成パラメータには、`ojspc` のコマンドライン・オプションを介した等価の機能のサポートがあります。`ojspc` は、OSE 環境向けに JSP ページを事前変換するユーティリティです。OracleJSP の構成パラメータと `ojspc` のオプションの相関関係については、A-13 ページの「[構成パラメータの概要表](#)」の表を参照してください。

ただし、ランタイム構成パラメータの場合、等価のサポートはありません。最も重要なのは `translate_params` で、要求パラメータのマルチバイト・コード化をサポートしないサーブレット環境で NLS を使用するには必須です。Oracle Servlet Engine にはこの機能が必要ですが、JSP ページ内で等価コードを記述するかどうかは、開発者が決定します。詳細は、8-6 ページの「[translate_params 構成パラメータの等価コード](#)」を参照してください。

サーブレットおよび JSP の技術的な背景情報

この付録では、サーブレットおよび JavaServer Pages の技術的な背景情報について説明します。このドキュメントはサーブレット・テクノロジーをよく理解しているユーザーを対象としていますが、ここで説明するサーブレット情報は再確認の意味でも役立ちます。

また、生成される JSP ページ実装クラスにより自動的に実装される標準 JavaServer Pages インタフェースについても簡単に説明します。ただし、この情報は、ほとんどの読者には不要です。

説明するトピックは、次のとおりです。

- [サーブレットの背景情報](#)
- [Web アプリケーションの階層](#)
- [標準 JSP インタフェースおよびメソッド](#)

サーブレットの背景情報

JSP ページは Java サーブレットに変換されるため、サーブレット・テクノロジーの概要を理解していると役立つ場合があります。ここで説明する概念の詳細は、Sun Microsystems の『Java Servlet Specification, Version 2.2』を参照してください。

この項で説明するメソッドの詳細は、次のサイトにある Sun Microsystems の Javadoc を参照してください。

<http://java.sun.com/products/servlet/2.2/javadoc/index.html>

サーブレット・テクノロジーの概要

近年、サーブレット・テクノロジーは、動的 HTML ページを介して Web サーバーの機能性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 サーバーで実行される Java プログラムです（これに対して、アプレットはクライアント・ブラウザで実行される Java プログラムです）。サーブレットは、ブラウザから HTTP 要求を受け取り、動的コンテンツを（データベースの問合せなどにより）生成し、HTTP 応答をブラウザに戻します。

サーブレット以前は、動的コンテンツには、CGI（Common Gateway Interface）テクノロジーが使用され、CGI プログラムは Perl などの言語で記述され、Web アプリケーションで Web サーバーを介してコールされていました。しかし、CGI のアーキテクチャと拡張性には制限があったため、最終的には理想的と言えないことが判明しました。

サーブレット・テクノロジーは、拡張性の改善に加えて、オブジェクト指向、プラットフォーム独立性、セキュリティおよび信頼性など、よく知られた Java のメリットを提供します。サーブレットでは、JDBC API（Java データベース接続用、特に、データベース・プログラマにとって重要）など、あらゆる標準 Java API を使用できます。

Java の領域では、データベースにアクセスするものなど、サーバー集中型アプリケーションには、アプレット・テクノロジーよりサーブレット・テクノロジーの方が大きなメリットをもたらします。その 1 つはサーブレットがサーバー上で実行されることです。通常、サーバーは多数のリソースを持つ信頼性の高いマシンであり、クライアントのリソース使用を最小限に抑えることができます。これに対して、アプレットはクライアントのブラウザにダウンロードされ、そこで実行されます。もう 1 つのメリットは、データへのダイレクト・アクセスです。サーブレットが実行される Web サーバーやデータ・サーバーは、ネットワーク・ファイアウォールに対してアクセス先データと同じ側にあります。ファイアウォールの外側にあるクライアント・マシンで実行されるアプレットの場合は、ダウンロード元とは異なるサーバーにアクセスできるように特別な手段（署名付きアプレットなど）が必要です。

サーブレット・インタフェース

定義によれば、Java サーブレットでは標準 `javax.servlet.Servlet` インタフェースが実装されます。このインタフェースでは、サーブレットの初期化、要求の処理、サーブレットの構成や他の基本情報の取得およびサーブレット・インスタンスの終了を行うメソッドが指定されています。

Web アプリケーションの場合は、標準 `javax.servlet.http.HttpServlet` 抽象クラスのサブクラス化により、`Servlet` インタフェースが間接的に実装されます。`HttpServlet` クラスには、次のメソッドが含まれています。

- `init(...)` および `destroy(...)`。それぞれサーブレットを初期化し、終了します。
- `doGet(...)`。HTTP の GET 要求用です。
- `doPost(...)`。HTTP の POST 要求用です。
- `doPut(...)`。HTTP の PUT 要求用です。
- `doDelete(...)`。HTTP の DELETE 要求用です。
- `service(...)`。HTTP 要求の受信用で、デフォルトでは適切な `doXXX()` メソッドにディスパッチします。
- `getServletInfo(...)`。サーブレットでそれ自体の情報を提供するために使用されます。

`HttpServlet` をサブクラス化するサーブレット・クラスでは、必要に応じて前述のメソッドのいくつかを実装する必要があります。各メソッドは、入力として標準 `javax.servlet.http.HttpServletRequest` インスタンスと標準 `javax.servlet.http.HttpServletResponse` インスタンスを取ります。

`HttpServletRequest` インスタンスは、要求パラメータの名前と値、要求を発行したりモート・ホストの名前および要求を受信したサーバーの名前など、HTTP 要求に関する情報をサーブレットに提供します。`HttpServletResponse` インスタンスは、コンテンツ長と MIME タイプの指定や、出力ストリームの提供など、応答送信に関する HTTP 固有の機能を提供します。

サーブレット・コンテナ

サーブレット・コンテナは、サーブレット・エンジンと呼ばれることもあり、サーブレットの実行と管理を行います。通常、サーブレット・コンテナは Java で記述され、Web サーバーに組み込まれるか（Web サーバーも Java で記述されている場合）、または Web サーバーに対応付けられて使用されます。

サーブレットがコールされると（サーブレットが URL で指定される場合など）、Web サーバーからサーブレット・コンテナに HTTP 要求が渡されます。次に、その要求がコンテナからサーブレットに渡されます。サーブレット管理中に、単純なコンテナにより次の処理が行われます。

- サーブレットのインスタンスが作成され、`init()` メソッドがコールされて初期化されます。
- サーブレットの `service()` メソッドがコールされます。
- サーブレットの `destroy()` メソッドがコールされ、ガベージを収集できるように必要に応じて破棄されます。

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、タスクを終了するたびにサーブレット・インスタンスを破棄するのではなく、再利用できるようにメモリに保存します。破棄されるのは、Web サーバーのシャットダウンなど、頻度の低いイベントの場合のみです。

サーブレットの実行中に追加のサーブレット要求があった場合、サーブレット・コンテナの動作は、サーブレットがシングル・スレッド・モデルを使用しているか、マルチ・スレッド・モデルを使用しているかに依存します。シングル・スレッドの場合、サーブレット・コンテナは複数の同時 `service()` コールが単一のサーブレット・インスタンスにディスパッチされないようにします。かわりに、複数の別個のサーブレット・インスタンスが起動されることがあります。マルチ・スレッド・モデルの場合、コンテナはコールごとに別個のスレッドを使用し、単一のサーブレット・インスタンスに対して複数の同時 `service()` コールを実行できます。ただし、サーブレット開発者は同期化を管理する必要があります。

サーブレット・セッション

サーブレットは、HTTP セッションを使用して、各 HTTP 要求を発行したユーザーを追跡します。このため、単一ユーザーからの要求グループをステートフルな方法で管理できます。サーブレット・セッション追跡は、CGI のような従来のテクノロジーにおける HTTP セッション追跡に性質が似ています。

HttpSession インタフェース

標準サーブレット API では、各ユーザーは標準 `javax.servlet.http.HttpSession` インタフェースを実装するオブジェクトで表されます。サーブレットは、この `HttpSession` オブジェクト内でセッション情報を設定および取得できます。このオブジェクトの有効範囲はアプリケーション・レベルで指定する必要があります。

サーブレットは、`HttpServletRequest` オブジェクト（HTTP 要求を表すオブジェクト）の `getSession()` メソッドを使用して、ユーザー用の `HttpSession` オブジェクトを取得または作成します。このメソッドは、ユーザー用のセッション・オブジェクトが存在しない場合に新規作成する必要があるかどうかを指定する、`boolean` 引数を取ります。

`HttpSession` インタフェースは、次のメソッドを指定してセッション情報を取得および設定します。

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

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

- `public Object getAttribute(String name)`

このメソッドでは、指定した名前セッションにバインドされたオブジェクト（または、一致するものがない場合は `null`）を取得します。

注意： 従来のサーブレット実装では、`setAttribute()` と `getAttribute()` のかわりに、`putValue()` と `getValue()` を同じグネチャで使用しています。

サーブレット・コンテナの実装とサーブレット自体に応じて、設定された時間間隔の後にセッションが自動的に時間切れになる場合と、サーブレットにより明示的に無効化される場合があります。サーブレットは、`HttpSession` インタフェースで指定した次のメソッドにより、セッションのライフ・サイクルを管理できます。

- `public boolean invalidate()`

このメソッドは、セッションを即時に無効にし、そこからオブジェクトをバインド解除します。

- `public boolean setMaxInactiveInterval(int interval)`

このメソッドは、秒単位のタイムアウト間隔を整数で設定します。

- `public boolean isNew()`

このメソッドは、セッションを作成した要求内では `true`、それ以外の場合は `false` を返します。

- `public boolean getCreationTime()`

このメソッドは、セッション・オブジェクトの作成時刻を、1970 年 1 月 1 日午前 0 時からのミリ秒数として返します。

- `public boolean getLastAccessedTime()`

このメソッドは、クライアントに対応付けられた最後の要求の時刻を、1970 年 1 月 1 日午前 0 時からのミリ秒数として返します。

セッションのトラッキング

`HttpSession` インタフェースでは、セッションをトラッキングするための代替メカニズムがサポートされます。どのメカニズムでも、セッション ID を割り当てるなんらかの手段が必要です。セッション ID は、サーブレット・コンテナにより割り当てられて使用される中間ハンドルです。同一ユーザーによる複数セッションでは、必要に応じて同じセッション ID を共有できます。

サポートされるセッションのトラッキング・メカニズムは、次のとおりです。

- cookie

サーブレット・コンテナはクライアントに cookie を送信し、クライアントは HTTP 要求ごとに cookie をサーバーに戻します。これにより、要求は cookie が示すセッション ID に対応付けられます。JSESSIONID は、cookie 名である必要があります。

これは最も使用頻度の高いメカニズムであり、サーブレット 2.2 仕様に準拠するサーブレット・コンテナすべてでサポートされます。

- URL のリライト

サーブレット・コンテナは、URL パスにセッション ID を追加します。パス・パラメータ名は、次のように jsessionid にする必要があります。

```
http://host[:port]/myapp/index.html;jsessionid=6789
```

これは、クライアントが cookie を受け入れない場合の、最も使用頻度の高いメカニズムです。

- SSL セッション

SSL (HTTPS プロトコルで使用される Secure Sockets Layer) には、クライアントから複数の要求を受け取って、それを単一セッションに属するものとして定義するメカニズムが組み込まれています。一部のサーブレット・コンテナは、独自のセッション追跡にも SSL メカニズムを使用します。

サーブレット・コンテキスト

サーブレット・コンテキストは、Web アプリケーションのすべてのインスタンスの状態情報を、単一の Java 仮想マシンに（つまり、Web アプリケーションの一部であるすべてのサーブレットと JSP ページのインスタンスに関して）メンテナンスするために使用されます。これは、セッションによりサーバー上で単一クライアントの状態情報がメンテナンスされる方法と同様ですが、サーブレット・コンテキストは単一ユーザーに固有ではなく、複数のクライアントを処理できます。通常は、特定の Java 仮想マシン内で実行される Web アプリケーションごとに、サーブレット・コンテキストが 1 つずつあります。サーブレット・コンテキストは、「アプリケーション・コンテナ」と見なすことができます。

サーブレット・コンテキストは、標準 `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あり、そのクラスはサーブレットをサポートするすべての Web サーバーで提供されます。

`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、単一の JVM 内でグループ内のサーブレット間でリソースの共有を可能にします。（複数の同時 JVM をサポートするサーブレット・コンテナの場合は、リソース共有の実装が異なります。）

サーブレット・コンテキストは、アプリケーションを実行するユーザーのセッション・オブジェクトをメンテナンスし、そのアプリケーションの実行中のインスタンスの有効範囲を提

供します。このメカニズムを介して、各アプリケーションが個別クラス・ローダーからロードされ、そのランタイム・オブジェクトが他のアプリケーションのオブジェクトと区別されます。特に、`ServletContext` オブジェクトは、各アプリケーション・ユーザー用の `HttpSession` オブジェクトと同様に、アプリケーションに対して個別になります。

Sun Microsystems の『Java Servlet Specification, Version 2.2』によれば、ほとんどの実装では単一ホスト内で複数のサーブレット・コンテキストを提供できます。これにより、Web アプリケーションごとに固有のサーブレット・コンテキストを使用できるようになります。(旧バージョンの実装では、通常、特定のホストに単一のサーブレット・コンテキストのみが提供されていました。)

`ServletContext` インタフェースは、サーブレットがそれを実行するサーブレット・コンテナと通信できるようにするメソッドを指定します。これは、サーブレットがアプリケーション・レベルの環境および状態情報を取得する方法の 1 つです。

注意： 旧バージョンのサーブレット仕様では、サーブレット・コンテキストの概念は十分に定義されていませんでした。ただし、バージョン 2.1 (b) からは、この概念がさらに明確化され、HTTP セッション・オブジェクトは複数のサーブレット・コンテキスト・オブジェクトにまたがって存在できないように指定されました。

イベント・リスナーを介したアプリケーションのライフ・サイクル管理

『Java Servlet Specification, Version 2.1』(以上) では、標準的な Java のイベント・リスナー・メカニズムを介して、限定的なアプリケーションのライフ・サイクル管理機能が提供されます。HTTP セッション・オブジェクトでは、イベント・リスナーを使用して、セッション・オブジェクトに格納されたオブジェクトに、それが追加または削除されるタイミングを認識させることができます。通常、セッション・オブジェクト内のオブジェクトは、セッションが無効になったために削除されるので、このメカニズムにより開発者はセッション・ベースのリソースを管理できます。同様に、イベント・リスナー・メカニズムにより、ページ・ベースと要求ベースのリソース管理も可能になります。

残念ながら、サーブレット・コンテキスト・オブジェクトでは、この種の通知はサポートされません。標準的なサーブレット・アプリケーション・サポートでは、アプリケーション・ベースのリソースを管理する手段は提供されません。

サーブレットの起動

HTML ページと同様に、サーブレットも URL を介して起動されます。サーブレットは、Web サーバー実装における URL へのマッピングに従って起動されます。考えられるマッピングは次のとおりです。

- 特定の URL を特定のサーブレット・クラスにマップできます。
- ディレクトリ内のどのクラスもサーブレットとして実行できるように、ディレクトリ全体をマップできます。たとえば、`/servlet/<servlet_name>` 形式のどの URL でもサーブレットが実行されるように、特殊な `/servlet` ディレクトリをマップできます。
- 名前に特定の拡張子が付いたファイルを指定する URL をサーブレットとして実行できるように、ファイル拡張子をマップできます。

このマッピングは、Web サーバー構成の一部として指定します。

Web アプリケーションの階層

Web アプリケーションに関連するエンティティ（サーブレットと JSP ページの組合せで構成）の場合、階層は単純ではありませんが、次の順序の階層と見なすことができます。

1. サーブレット・オブジェクト（ページ実装オブジェクトを含む）

サーブレットごと、および実行中のアプリケーション内の JSP ページ実装ごとに、1 つのサーブレット・オブジェクト（さらに、シングルのスレッド実行モデルとマルチ・スレッド実行モデルのどちらを使用するかに応じて複数のオブジェクト）が存在します。サーブレット・オブジェクトは、クライアントからの要求オブジェクトを処理して、クライアントに応答オブジェクトを戻します。サーブレット・コードの場合と同様に、JSP ページでは応答オブジェクトの作成方法を指定します。

単一のページまたはサーブレットを他のページまたはサーブレットに「挿入」または「転送」する場合など、状況によっては、複数のサーブレット・オブジェクトが単一の要求オブジェクトに存在するものと見なすことができます。

通常、ユーザーはセッション中に複数のサーブレット・オブジェクトにアクセスし、各サーブレット・オブジェクトがセッション・オブジェクトに対応付けられます。

ページ実装オブジェクトのみでなく、サーブレット・オブジェクトでも、標準 `javax.servlet.Servlet` インタフェースが間接的に実装されます。Web アプリケーション内のサーブレットの場合は、そのために標準

`javax.servlet.http.HttpServlet` 抽象クラスがサブクラス化されます。JSP ページ実装クラスの場合は、そのために標準 `javax.servlet.jsp.HttpJspPage` インタフェースが実装されます。

2. 要求オブジェクトと応答オブジェクト

これらのオブジェクトは、ユーザーがアプリケーションを実行すると生成される個々の HTTP 要求および応答を表します。

通常、ユーザーはセッション中に複数の要求を生成し、複数の応答を受け取ります。要求オブジェクトと応答オブジェクトは、セッションに「含まれる」のではなく、セッションに対応付けられます。

クライアントから発行された要求は、URL の仮想パスに従って適切なサーブレット・コンテキスト・オブジェクト（クライアントで使用中のアプリケーションに対応付けられているオブジェクト）にマップされます。仮想パスには、アプリケーションのルート・パスが挿入されます。

要求オブジェクトでは、標準 `javax.servlet.http.HttpServletRequest` インタフェースが実装されます。

応答オブジェクトでは、標準 `javax.servlet.http.HttpServletResponse` インタフェースが実装されます。

3. セッション・オブジェクト

セッション・オブジェクトには、特定のセッションのユーザーに関する情報が格納され、複数のページ要求間で単一ユーザーを識別する手段を提供します。ユーザーごとに 1 つずつセッション・オブジェクトがあります。

特定の時点で単一のサーブレットまたは JSP ページを複数のユーザーが使用している場合があり、各ユーザーは固有のセッション・オブジェクトで表されます。ただし、この種のセッション・オブジェクトはいずれも、アプリケーション全体に対応するサーブレット・コンテキストによりメンテナンスされます。実際には、各セッション・オブジェクトを、共通サーブレット・コンテキストに対応付けられた Web アプリケーションのインスタンスを表すものと見ることができます。

通常、セッション・オブジェクトは複数の要求オブジェクト、応答オブジェクトおよびページまたはサーブレット・オブジェクトを順番に使用し、他のセッションが同一オブジェクトを使用することはありません。ただし、セッション・オブジェクトに各オブジェクト自体が「含まれている」わけではありません。

特定ユーザーのセッションのライフ・サイクルは、そのユーザーからの初回要求で始まります。そのユーザーがセッションを終了するか（アプリケーションを終了するなど）、タイムアウトになると、ライフ・サイクルが終了します。

HTTP セッション・オブジェクトでは、`javax.servlet.http.HttpSession` インタフェースが実装されます。

注意： バージョン 2.1 (b) 以前のサーブレット仕様では、セッション・オブジェクトが複数のサーブレット・コンテキスト・オブジェクト間にあたることが許されていました。

4. サブレット・コンテキスト・オブジェクト

サブレット・コンテキスト・オブジェクトは、サーバー上の特定のパスに対応付けられています。これは、サブレット・コンテキストに対応付けられたアプリケーションのモジュール用のベース・パスであり、アプリケーション・ルートと呼ばれます。

特定の JVM には、アプリケーションの全セッション用に単一のサブレット・コンテキスト・オブジェクトがあり、そのアプリケーションを構成するサブレットと JSP ページにサーバーからの情報を提供します。また、サブレット・コンテキスト・オブジェクトにより、アプリケーション・セッションでは、他のアプリケーションから隔離された保護環境内でデータを共有できます。

サブレット・コンテナは、標準 `javax.servlet.ServletContext` インタフェースを実装するクラスを提供し、ユーザーからアプリケーションを初めて要求されたときに、このクラスをインスタンス化し、この `ServletContext` オブジェクトをアプリケーションの位置に関するパス情報とともに提供します。

通常、サブレット・コンテキスト・オブジェクトは、アプリケーションの複数の同時ユーザーを表すセッション・オブジェクトのプールを持ちます。

サブレット・コンテキストのライフ・サイクルは、対応するアプリケーションに関する（任意のユーザーからの）初回要求で始まります。このライフ・サイクルが終了するのは、サーバーがシャットダウンされるか、他の方法で終了した場合のみです。

（サブレット・コンテキストに関する追加情報は、B-6 ページの「[サブレット・コンテキスト](#)」を参照してください。）

5. サブレット構成オブジェクト

サブレット・コンテナは、サブレット構成オブジェクトを使用して、初期化時にサブレットに情報を渡します。`Servlet` インタフェースの `init()` メソッドは、入力としてサブレット構成オブジェクトを取ります。

サブレット・コンテナは、標準 `javax.servlet.ServletConfig` インタフェースを実装するクラスを提供し、必要に応じてインスタンス化します。サブレット構成オブジェクトには、サブレット・コンテキスト・オブジェクト（同じくサブレット・コンテナによりインスタンス化）が含まれています。

標準 JSP インタフェースおよびメソッド

JSP トランスレータにより生成されるコードに実装できる標準インタフェースが2つあり、どちらも `javax.servlet.jsp` パッケージにあります。

- `JspPage`
- `HttpJspPage`

`JspPage` は、特定のプロトコルとの併用を想定していない汎用インタフェースです。このインタフェースにより、`javax.servlet.Servlet` インタフェースが拡張されます。

`HttpJspPage` は、HTTP プロトコルを使用する JSP ページ用のインタフェースです。このインタフェースにより `JspPage` が拡張され、通常は JSP トランスレータで生成されるサーブレット・クラスにより直接、自動的に実装されます。

`JspPage` では、生成されるクラスのインスタンスの初期化と終了に使用する次のメソッドが指定されています。

- `jspInit()`
- `jspDestroy()`

これらのメソッドのコードは、次のように、すべて JSP ページのスクリプトレットに挿入する必要があります。

```
<%!
    void jspInit()
    {
        ...your implementation code...
    }
%>
```

(JSP の構文については後述します。1-11 ページの「[スクリプト要素](#)」を参照してください。)

`HttpJspPage` では、次のメソッドの仕様部が追加されます。

- `_jspService()`

通常、このメソッドのコードはトランスレータにより自動的に生成され、JSP ページのスクリプトレットからのコード、JSP ディレクティブにより生成されたコードおよびページの静的コンテンツが含まれます。(JSP ディレクティブは、スクリプトレット用 Java 言語の指定やパッケージ・インポートの提供など、ページに関する情報提供に使用されます。1-9 ページの「[ディレクティブ](#)」を参照してください。)

B-3 ページの「[サーブレット・インタフェース](#)」で説明した `Servlet` メソッドと同様に、`_jspService()` メソッドは入力として `HttpServletRequest` インスタンスと `HttpServletResponse` インスタンスを取ります。

JspPage および HttpJspPage インタフェースは、Servlet インタフェースから次のメソッドを継承します。

- `init()`
- `destroy()`
- `service()`
- `getServletConfig()`
- `getServletInfo()`

Servlet インタフェースとその主要メソッドについては、B-3 ページの「[サーブレット・インタフェース](#)」を参照してください。

コンパイル時 JML タグ・サポート

OracleJSP リリース 1.0.0.6.x は JSP 1.0 実装であったため、JML タグが Oracle 固有の拡張機能としてのみサポートされていました。（タグ・ライブラリのフレームワークは、JSP 1.1 までは JavaServer Pages 仕様に追加されませんでした。）これらのリリースの場合、JML タグの処理は、OracleJSP トランスレータに組み込まれていました。これは「コンパイル時 JML サポート」と呼ばれます。

リリース 1.1.0.0.0 でもコンパイル時 JML 実装は引き続きサポートされていますが、通常は、[第7章「JSP のタグ・ライブラリと Oracle の JML タグ」](#)で説明したように、できるだけランタイム実装を使用することをお勧めします。

この付録では、ランタイム実装にはないコンパイル時実装の機能について説明します。説明するトピックは、次のとおりです。

- [JML のコンパイル時とランタイムの考慮事項とロジスティック](#)
- [JML コンパイル時 /1.0.0.6.x 構文のサポート](#)
- [JML コンパイル時 /1.0.0.6.x タグのサポート](#)

JML のコンパイル時とランタイムの考慮事項とロジスティック

この項では、コンパイル時タグ・ライブラリの次の2つの側面を、ランタイム・タグ・ライブラリと比較して説明します。

- コンパイル時タグ・ライブラリ実装を使用するとメリットが得られる場合の全般的な考慮事項（JML のみでなくすべてのライブラリ）
- コンパイル時 JML 実装に特に必要な taglib ディレクティブ

コンパイル時とランタイムの全般的な考慮事項

Sun Microsystems の『JavaServer Pages Specification, Version 1.1』では、カスタム・タグ・ライブラリのランタイム・サポート・メカニズムについて説明しています。このメカニズムでは、XML 形式のタグ・ライブラリ記述ファイルを使用してタグが指定されます。7-2 ページの「[標準タグ・ライブラリのフレームワーク](#)」を参照してください。

このモデルに準拠するタグ・ライブラリを作成して使用すると、そのライブラリはすべての標準 JSP 環境に移植可能であると見なされます。

ただし、コンパイル時実装を考慮するには、次のような理由があります。

- コンパイル時実装の方が、効率的なコードが生成される場合があります。
- コンパイル時実装では、エンド・ユーザーが実行時に気づくのではなく、開発者が変換中とコンパイル中にエラーを捕捉できます。

将来、オラクル社は、コンパイル時タグ実装に、カスタム・タグ・ライブラリを作成するための、一般的なフレームワークを提供する可能性があります。この種の実装は OracleJSP トランスレータに依存するため、他の JSP 環境には移植できません。

コンパイル時実装の一般的なメリットとデメリットは、Oracle JML タグ・ライブラリにも適用されます。OracleJSP の旧バージョンに当初導入されたように、コンパイル時の JML 実装を使用する方がメリットが得られる場合があります。また、その実装には、他にも少数のタグとサポートされる式の構文があります。（C-3 ページの「[JML コンパイル時 /1.0.0.6.x 構文のサポート](#)」および C-6 ページの「[JML コンパイル時 /1.0.0.6.x タグのサポート](#)」を参照してください。）

ただし、通常は、JSP 1.1 仕様に準拠する JML ランタイム実装を使用することをお薦めします。

コンパイル時の JML サポート用の taglib ディレクティブ

OracleJSP 1.0.0.6x/ コンパイル時 JML サポートの実装では、オラクル社が提供するカスタム・クラス `OpenJspRegisterLib` を使用して JML タグ・サポートが実装されます。

コンパイル時実装で JML タグを使用する JSP ページでは、taglib ディレクティブで（標準的な JSP 1.1 のタグ・ライブラリの使用方法として TLD ファイルを指定するのではなく）このクラスの完全修飾名を指定する必要があります。

次に例を示します。

```
<%@ taglib uri="oracle.jsp.parse.OpenJspRegisterLib" prefix="jml" %>
```

JML ランタイム実装での taglib ディレクティブの使用方法については、7-12 ページの「[taglib ディレクティブ](#)」を参照してください。

JML コンパイル時 /1.0.0.6.x 構文のサポート

この項では、タグの属性値を指定する場合に、コンパイル時 JML 実装でサポートされる Oracle 固有の Bean 参照構文と式構文について説明します。説明するトピックは、次のとおりです。

- [JML の Bean 参照と式、コンパイル時実装](#)
- [JML 式による属性設定](#)

この機能は、他の JSP 環境には移植できません。

JML の Bean 参照と式、コンパイル時実装

一般に、Bean 参照は、Bean のプロパティまたはメソッドにアクセスする、JavaBeans インスタンス (Bean) の参照です。これには、それ自体が他の Bean のプロパティである Bean のプロパティまたはメソッドの参照が含まれます。

標準 JavaBeans 構文では、直接参照するのではなくアクセッサ・メソッドをコールしてプロパティにアクセスする必要があるため、これでは煩雑になります。たとえば、次の直接参照があるとします。

```
a.b.c.d.doIt()
```

この参照を標準 JavaBeans 構文では次のように表す必要があります。

```
a.getB().getC().getD().doIt()
```

ただし、Oracle のコンパイル時 JML 実装には、構文の省略形が用意されています。

JML の Bean 参照

コンパイル時 JML 実装によりサポートされる Oracle 固有の構文では、直接「.」(ドット) 表記法を使用して Bean 参照を表すことができます。標準 Bean プロパティのアクセッサ・メソッドの構文も有効であることに注意してください。

次の標準的な JavaBeans 参照があるとします。

```
customer.getName()
```

JML の Bean 参照構文では、この参照を次のどちらかの方法で表すことができます。

```
customer.getName()
```

または

```
customer.name
```

JavaBeans にはオプションでデフォルト・プロパティを指定できます。参照を明示的に指定しない場合はデフォルト・プロパティの参照と見なされます。デフォルト・プロパティ名は、JML の Bean 参照では省略できます。前述の例で、name がデフォルト・プロパティであれば、次の 3 つはいずれも有効な JML の Bean 参照になります。

```
customer.getName()
```

または

```
customer.name
```

または単に

```
customer
```

ほとんどの JavaBeans では、デフォルト・プロパティは定義されません。デフォルト・プロパティが定義される JavaBeans のうち、最も重要なのは JML データ型の JavaBeans です。5-2 ページの「[JML の拡張データ型](#)」を参照してください。

JML 式

コンパイル時 JML 実装によりサポートされる JML 式の構文は、標準 JSP 式構文のスーパーセットであり、前項で説明した JML の Bean 参照構文のサポートが追加されています。

JML 式に使用する JML の Bean 参照は、次の構文で囲む必要があります。

```
$(JML_bean_reference)
```

JML 式による属性設定

7-28 ページの「[JSP マークアップ言語 \(JML\) タグの説明](#)」のタグ属性の説明は、移植可能な標準構文を示しています。その説明のとおり、属性はランタイム用またはコンパイル時 JML 実装用、さらに非 Oracle JSP 環境用に設定できます。

ただし、Oracle 固有のコンパイル時実装のみを使用する場合は、JML の Bean 参照と JML 式の構文を使用して属性を設定できます。C-3 ページの「[JML の Bean 参照と式、コンパイル時実装](#)」を参照してください。

次の点に注意してください。

- 第7章で、文字列リテラルまたは式のいずれかが使用可能と記載されている属性の場合は、その `$(...)` 構文の JML 式を標準 JSP `<%=...%>` 構文内で使用できます。

JML の `useVariable` タグの使用例を考えます。ランタイム実装には、次のような構文を使用します。

```
<jml:useVariable id = "isValidUser" type = "boolean" value = "<%= dbConn.isValid() %>" scope = "session" />
```

また、コンパイル時実装には次のような構文を使用できます（`value` 属性は、文字列リテラルでも式でもかまいません）。

```
<jml:useVariable id = "isValidUser" type = "boolean" value = "<%= ${dbConn.valid} %>" scope = "session" />
```

- 第7章で、式のみ使用可能と記載されている属性の場合は、その `$(...)` 構文内の JML 式を `<%=...%>` 構文内でネストせずに使用できます。

JML の `choose...when` タグの使用例を考えます。ランタイム実装には、次のような構文を使用します（`orderedItem` は `JmlBoolean` インスタンスであるとしします）。

```
<jml:choose>
  <jml:when condition = "<%= orderedItem.getValue() %>" >
    You have changed your order:
    -- outputs the current order --
  </jml:when>
  <jml:otherwise>
    Are you sure we can't interest you in something?
  </jml:otherwise>
</jml:choose>
```

また、コンパイル時実装には次のような構文を使用できます（`condition` 属性に使用できるのは式のみです）。

```
<jml:choose>
  <jml:when condition = "${orderedItem}" >
    You have changed your order:
    -- outputs the current order --
  </jml:when>
  <jml:otherwise>
    Are you sure we can't interest you in something?
  </jml:otherwise>
</jml:choose>
```

JML コンパイル時 /1.0.0.6.x タグのサポート

この項のトピックは、次のとおりです。

- すべてのコンパイル時タグの概要と、ランタイム実装でサポートされないタグ
- コンパイル時実装でサポートされ、ランタイム実装ではサポートされないタグの説明（この種のタグは、7-28 ページの「[JSP マークアップ言語 \(JML\) タグの説明](#)」に記載されていないため）

注意： ほとんどの場合、ランタイム実装でサポートされない JML タグには、標準 JSP の等価タグがあります。ただし、JSP 1.1 仕様に準拠すると実装が困難な機能がある一部のコンパイル時タグは、非サポートになりました。

JML タグの概要、1.0.0.6.x/ コンパイル時と 1.1.0.0.0/ ランタイム

ほとんどの JML タグは、ランタイム・モデルでもコンパイル時モデルでも使用できますが、[表 C-1](#) のように例外があります。

表 C-1 サポートされる JML タグ：コンパイル時モデルとランタイム・モデルの比較

タグ	OracleJSP コンパイル時実装でのサポートの有無	OracleJSP ランタイム実装でのサポートの有無
Bean バインド・タグ：		
useBean	あり	なし（jsp:useBean を使用）
useVariable	あり	あり
useForm	あり	あり
useCookie	あり	あり
remove	あり	あり
Bean 操作タグ		
getProperty	あり	なし（jsp:getProperty を使用）
setProperty	あり	なし（jsp:setProperty を使用）
set	あり	なし
call	あり	なし
lock	あり	なし
制御フロー・タグ		
if	あり	あり

表 C-1 サポートされる JML タグ：コンパイル時モデルとランタイム・モデルの比較（続き）

タグ	OracleJSP コンパイル時実装でのサポートの有無	OracleJSP ランタイム実装でのサポートの有無
choose	あり	あり
for	あり	あり
foreach	あり（type 属性はオプション）	あり（type 属性は必須）
return	あり	あり
flush	あり	あり
include	あり	なし（jsp:include を使用）
forward	あり	なし（jsp:forward を使用）
XML タグ		
transform	あり	あり
styleSheet	あり	あり
ユーティリティ・タグ		
print	あり（文字列リテラルを指定するには二重引用符を使用）	なし（JSP の式を使用）
plugin	あり	なし（jsp:plugin を使用）

追加の JML タグの説明、コンパイル時実装

この項では、JML コンパイル時実装では引き続きサポートされますが、JML ランタイム実装ではサポートされない JML タグの詳細を説明します。これらのタグは、7-28 ページの「[JSP マークアップ言語（JML）タグの説明](#)」には記載されていません。

まとめると、これには次の JML タグが該当します。

- [JML の useBean タグ](#)
- [JML の getProperty タグ](#)
- [JML の setProperty タグ](#)
- [JML の set タグ](#)
- [JML の call タグ](#)
- [JML の lock タグ](#)
- [JML の include タグ](#)
- [JML の forward タグ](#)

- JML の `print` タグ
- JML の `plugin` タグ

タグの説明に含まれる構文では、次の点に注意してください。

- イタリックは、値または文字列を指定する必要があることを示します。
- オプション属性は大カッコ [...] で囲まれています。
- オプション属性のデフォルト値は、**太字**で示されます。
- 属性の指定方法を示す選択肢は、縦線 | で区切られています。
- 接頭辞「jml:」が使用されています。これは、表記規則によるものですが必須ではありません。taglib ディレクティブでは任意の接頭辞を指定できます。

JML の `useBean` タグ

このタグでは、ページに使用するオブジェクトが宣言され、以前にインスタンス化されているオブジェクトが存在する場合は、その位置が名前で指定した有効範囲で識別されます。存在しない場合は、このタグにより該当するクラスの新規インスタンスが作成され、名前で指定した有効範囲に連結されます。

構文と意味は標準 `jsp:useBean` タグと同じですが、`jsp:useBean` の使用方法で JSP 式が有効な場合は、JML の `useBean` の使用方法で JML 式または JSP 式が有効です。

構文

```
<jml:useBean id = "beanInstanceName"
             scope = "page | request | session | application"
             class = "package.class" | type = "package.class" |
             class = "package.class" type = "package.class" |
             beanName = "package.class" | <%= jmlExpression %>"
             type = "package.class" />
```

また、`setProperty` タグなどのネストしたタグを追加でき、`</jml:useBean>` 終了タグを使用できます。

属性

属性と構文の詳細は、Sun Microsystems の『JavaServer Pages Specification, Version 1.1』を参照してください。

例

```
<jml:useBean id = "isValidUser" class = "oracle.jsp.jml.JmlBoolean" scope = "session" />
```

JML の getProperty タグ

このタグの機能は、標準 `jsp:getProperty` タグと同じです。応答に Bean プロパティの値を出力します。

`getProperty` の使用方法の概要は、Sun Microsystems の『JavaServer Pages Specification, Version 1.1』を参照してください。

構文

```
<jml:getProperty name = "beanInstanceName" property = "propertyName" />
```

属性

- name — プロパティを取得する Bean の名前。これは必須属性です。
- property — 取得するプロパティの名前。これは必須属性です。

例 次の例では、`salary` プロパティの現在の値が出力されます。（`salary` は `JmlNumber` 型とします。）

```
<jml:getProperty name="salary" property="value" />
```

この例は次の例と等価です。

```
<%= salary.getValue() %>
```

JML の setProperty タグ

このタグは、標準 `jsp:setProperty` タグでサポートされる機能を含むうえ、JML の式をサポートする機能を追加します。特に、JML の Bean 参照を使用できます。

`setProperty` の使用方法の概要は、Sun Microsystems の『JavaServer Pages Specification, Version 1.1』を参照してください。

構文

```
<jml:setProperty name = "beanInstanceName"
    property = " * " |
    property = "propertyName" [ param = "parameterName" ] |
    property = "propertyName"
    value = "stringLiteral | <%= jmlExpression %>" />
```

属性

- `name` — プロパティを設定する Bean の名前。これは必須属性です。
- `property` — 設定するプロパティの名前。これは必須属性です。
- `value` — 値を要求パラメータから取得するのではなく、直接設定するためのオプション・パラメータ。JML の `setProperty` タグでは、値を指定する標準 JSP 式に加えて JML 式がサポートされます。

例 次の例では、`salary` が 6% の上昇率で更新されます。（`salary` は `JmlNumber` 型とします。）

```
<jml:setProperty name="salary" property="value" value="<%= $[salary] * 1.06 %>" />
```

この例は次の例と等価です。

```
<% salary.setValue(salary.getValue() * 1.06); %>
```

JML の set タグ

このタグは、`setProperty` タグより便利な構文を使用して、Bean プロパティの代替設定方法を提供します。

構文

```
<jml:set name = "beanInstanceName.propertyName"
        value = "stringLiteral | <%= jmlExpression %>" />
```

属性

- `name` — 設定する Bean プロパティの直接参照（JML の Bean 参照）。これは必須属性です。
- `value` — 新しいプロパティ値。文字列リテラル、JML 式または標準 JSP 式で表されます。これは必須属性です。

例 次の各例では、`salary` が 6% の上昇率で更新されます。（`salary` は `JmlNumber` 型とします。）

```
<jml:set name="salary.value" value="<%= salary.getValue() * 1.06 %>" />
```

または

```
<jml:set name="salary.value" value="<%= $[salary.value] * 1.06 %>" />
```

または

```
<jml:set name="salary" value="<%= $[salary] * 1.06 %>" />
```

この3つの例は次の例と等価です。

```
<% salary.setValue(salary.getValue() * 1.06); %>
```

JML の call タグ

このタグは、なにも戻さない Bean メソッドを起動するメカニズムを提供します。

構文

```
<jml:call method = "beanInstanceName.methodName(parameters)" />
```

属性

- **method** — スクリプトレットに記述するメソッドのコール。ただし、文の *beanInstanceName.methodName* 部分は、JML 式の `$[...]` 構文で囲むと JML の Bean 参照として記述できます。これは必須属性です。

例 次の例では、クライアントは別のページにリダイレクトされます。

```
<jml:call name='response.sendRedirect("http://www.oracle.com/")' />
```

この例は次の例と等価です。

```
<% response.sendRedirect("http://www.oracle.com/"); %>
```

JML の lock タグ

このタグでは、タグ本体内で名前付きオブジェクトを使用するコードについて、そのオブジェクトへの制御された同期アクセスが可能になります。

通常、JSP 開発者は並行性の問題を考慮する必要はありません。ただし、**application** 有効範囲を持つオブジェクトは、そのアプリケーションを実行中の全ユーザー間で共有されるため、重要なデータへのアクセスは制御して調整する必要があります。

JML の lock タグを使用すると、様々なユーザーによる同時更新を防止できます。

構文

```
<jml:lock name = "beanInstanceName" >  
    ...body...  
</jml:lock>
```

属性

- name — lock タグ本体のコードの実行中にロックする必要のあるオブジェクトの名前。これは必須属性です。

例 次の例で、pageCount は application 有効範囲を持つ JmlNumber の値です。この変数は、このコードが現在の値を取得してから新しい値を設定するまでに、他のユーザーにより値が更新されないようにロックされます。

```
<jml:lock name="pageCount" >
  <jml:set name="pageCount.value" value="<%= pageCount.getValue() + 1 %>" />
</jml:lock>
```

この例は次の例と等価です。

```
<% synchronized(pageCount)
{
    pageCount.setValue(pageCount.getValue() + 1);
}
%>
```

JML の include タグ

このタグでは、このページ（include を起動するページ）の応答に別の JSP ページ、サーブレットまたは HTML ページの出力が挿入されます。機能は標準 jsp:include タグと同じですが、page 属性は JML 式としても表すことができます。

構文

```
<jml:include page = "relativeURL | <%= jmlExpression %>" flush = "true" />
```

属性

include 属性と使用方法の概要は、Sun Microsystems の『JavaServer Pages Specification, Version 1.1』を参照してください。

例 次の例では、HTML 表を含むプレゼンテーション・コンポーネント table.jsp の出力が、問合せ文字列と要求属性のデータに基づいて挿入されます。

```
<jml:include page="table.jsp?maxRows=10" flush="true" />
```

JML の forward タグ

このタグでは、要求が別の JSP ページ、サーブレットまたは HTML ページに転送されます。機能は標準 `jsp:forward` タグと同じですが、`page` 属性は JML の式としても表すことができます。

構文

```
<jml:forward page = "relativeURL" | "<%= jmlExpression %>" />
```

属性

`forward` 属性と使用方法の概要は、Sun Microsystems の『JavaServer Pages Specification, Version 1.1』を参照してください。

例

```
<jml:forward page="altpage.jsp" />
```

JML の print タグ

このタグの機能は、実際には標準 JSP 式 `<%= expr %>` と同じです。指定した JML 式または文字列リテラルが評価され、結果が応答に出力されます。このタグでは、JML 式を `<%= ... %>` 構文で囲む必要はありませんが、文字列リテラルは二重引用符で囲む必要があります。

構文

```
<jml:print eval = '"stringLiteral"' | "jmlExpression" />
```

属性

`eval` — 評価して出力する文字列または式を指定します。これは必須属性です。

例 次のどちらの例でも、`salary` の現在の値が出力されます。値は `JmlNumber` 型です。

```
<jml:print eval="$[salary]" />
```

または

```
<jml:print eval="salary.getValue()" />
```

次の例では、文字列リテラルが出力されます。

```
<jml:print eval='"Your string here"' />
```

JML の plugin タグ

このタグの機能は、標準 `jsp:plugin` タグと同じです。

`plugin` 属性、使用方法および例については、Sun Microsystems の『JavaServer Pages Specification, Version 1.1』を参照してください。

記号

_jspService() メソッド, B-11

A

addclasspath、ojspc のオプション, 6-26

Apache/JServ

Apache の「mod」, 2-5

CLASSPATH の構成, A-7

config、ファイル名拡張子のマッピング, A-10

JSP とサーブレットのセッション共有の概要, 4-33

mod_jserv モジュール, 2-6

mod_ose モジュール, 2-6

Oracle Internet Application Server での使用, 4-30

OracleJSP のアプリケーション・フレームワーク,
4-33

OracleJSP のサポート, 2-10

OracleJSP の動的挿入のサポート, 4-31

構成パラメータの設定, A-23

特記事項の概要, 4-30

Apache/JServ 用のアプリケーション・フレーム
ワーク, 4-33

application_OnEnd タグ、globals.jsa, 5-40

application_OnStart タグ、globals.jsa, 5-39

application オブジェクト (暗黙的), 1-15

application 有効範囲 (JSP オブジェクト), 1-14

appRoot、ojspc のオプション, 6-26

B

Bean 参照、コンパイル時 JML, C-3

bypass_source 構成パラメータ, A-17

C

call タグ、コンパイル時 JML, C-11

choose タグ、JML, 7-33

classesXX.zip、JDBC 用の必須ファイル, A-5

CLASSPATH

CLASSPATH とクラス・ローダーの問題, 4-19

Web サーバーの CLASSPATH 構成, A-7

構成、Apache/JServ, A-7

構成、JSWDK, A-8

構成、Tomcat, A-9

classpath

classpath 構成パラメータ, A-17

config オブジェクト (暗黙的), 1-15

ConnBean JavaBeans

サンプル・アプリケーション, 9-22

使用方法 (接続用), 5-13

ConnCacheBean JavaBeans

サンプル・アプリケーション, 9-25

使用方法 (接続キャッシュ用), 5-15

context、publishjsp のオプション, 6-42

cookie, B-6

createcontext コマンド, 6-15, 6-16, 6-17, 6-42

CursorBean JavaBeans

サンプル・アプリケーション, 9-23

使用方法 (DML 用), 5-18

D

Database-Access JavaBeans

DML 用の CursorBean, 5-18

概要, 5-12

サンプル・アプリケーション, 9-20

接続キャッシュ用の ConnCacheBean, 5-15

- 接続用の ConnBean, 5-13
- 問合せ用の DBBean, 5-17
- DBBean JavaBeans
 - サンプル・アプリケーション, 9-21
 - 使用方法（問合せ用）, 5-17
- dbClose SQL タグ、接続のクローズ, 5-23
- dbCloseQuery SQL タグ、カーソルのクローズ, 5-25
- dbExecute SQL タグ、DML/DDDL, 5-26
- dbNextRow SQL タグ、結果の処理, 5-25
- dbOpen SQL タグ、接続のオープン, 5-23
- dbQuery SQL タグ、問合せ実行, 5-24
- developer_mode 構成パラメータ, A-18
- d、ojspc のオプション（バイナリ出力ディレクトリ）, 6-27

E

- EJB、JSP からのコール
 - Oracle Servlet Engine, 4-4
 - 概要, 4-2
 - 中間層, 4-3
- emit_debuginfo 構成パラメータ, A-18
- Enterprise JavaBeans、「EJB」を参照
- exception オブジェクト（暗黙的）, 1-15
- extend、ojspc のオプション, 6-28
- extend、publishjsp のオプション, 6-43
- external_resource 構成パラメータ, A-19
- extres、ojspc のオプション, 6-29

F

- fallback タグ（plugin タグと併用）, 1-20
- Feiner、Amy（welcome）, 1-3
- flush タグ、JML, 7-36
- foreach タグ、JML, 7-35
- forward タグ, 1-19
- forward タグ、コンパイル時 JML, C-13
- for タグ、JML, 7-34

G

- getProperty タグ, 1-17
- getProperty タグ、コンパイル時 JML, C-9
- globals.jsa
 - アプリケーション・イベント, 5-39
 - アプリケーションとセッションのライフ・サイクル, 5-36

- アプリケーションの配布, 5-35
- イベント処理, 5-39
- 機能の概要, 5-35
- グローバル宣言, 5-44
- グローバルな JavaBeans, 5-44
- グローバルな JSP ディレクティブ, 5-44
- 構文とセマンティックの概要, 5-37
- 個別アプリケーションおよびセッション, 5-36
- サプレット 2.0 向けの拡張サポート, 5-34
- サンプル・アプリケーション, 9-38
- サンプル・アプリケーション、アプリケーション・イベント, 9-38
- サンプル・アプリケーション、アプリケーションおよびセッション・イベント, 9-41
- サンプル・アプリケーション、グローバル宣言, 9-43
- セッション・イベント, 5-41
- ファイルのコンテンツ、構造, 5-45
- 例、宣言とディレクティブ, 5-45

H

- hotload、ojspc のオプション, 6-29
- hotload、publishjsp のオプション, 6-42
- HttpJspPage インタフェース, B-11
- HttpSessionBindingListener, 3-10
- HttpSession インタフェース, B-4

I

- if タグ、JML, 7-32
- implement、ojspc のオプション, 6-30
- implement、publishjsp のオプション, 6-43
- include タグ, 1-18
- include タグ、コンパイル時 JML, C-12
- include ディレクティブ, 1-9

J

- JavaBeans
 - Bean 参照、コンパイル時 JML, C-3
 - Database-Access JavaBeans のサンプル, 9-20
 - JML の Bean バインド・タグ, 7-28
 - Oracle データベース・アクセスの Beans, 5-12
 - useBean サンプル・アプリケーション, 9-3
 - useBean タグの併用, 1-16
 - グローバルな JavaBeans、globals.jsa, 5-44

- スクリプトレットとの比較, 4-2
- 問合せ Bean のサンプル・アプリケーション, 9-13
- ビジネス・ロジックの分離に使用, 1-4
- javaccmd 構成パラメータ, A-19
- java コマンド (セッション・シェル), 6-58
- JDeveloper
 - JSP ページの配布に使用, 6-67
 - OracleJSP のサポート, 2-20
- jml call タグ、コンパイル時 JML, C-11
- jml choose タグ, 7-33
- jml flush タグ, 7-36
- jml foreach タグ, 7-35
- jml forward タグ、コンパイル時 JML, C-13
- jml for タグ, 7-34
- jml getProperty タグ、コンパイル時 JML, C-9
- jml if タグ, 7-32
- jml include タグ、コンパイル時 JML, C-12
- jml lock タグ、コンパイル時 JML, C-11
- jml otherwise タグ, 7-33
- jml plugin タグ、コンパイル時 JML, C-14
- jml print タグ, C-13
- jml remove タグ, 7-31
- jml return タグ, 7-36
- jml setProperty タグ、コンパイル時 JML, C-9
- jml set タグ、コンパイル時 JML, C-10
- jml useCookie タグ, 7-30
- jml useForm タグ, 7-29
- jml useVariable タグ, 7-29
- jml when タグ, 7-33
- JmlBoolean 拡張データ型, 5-3
- JmlFPNumber 拡張データ型, 5-4
- JmlNumber 拡張データ型, 5-4
- JmlString 拡張データ型, 5-6
- JML タグ
 - Bean 参照、コンパイル時 JML, C-3
 - taglib ディレクティブ, 7-20
 - taglib ディレクティブ、コンパイル時 JML, C-2
 - 概要, 7-18
 - 概要、コンパイル時とランタイム, C-6
 - 原理, 7-19
 - サンプル・アプリケーション, 9-27
 - 式、コンパイル時 JML, C-4
 - 説明、Bean バインド・タグ, 7-28
 - 説明、XSL スタイルシート・タグ, 5-10
 - 説明、追加のコンパイル時タグ, C-7
 - 説明、ロジック / フロー制御タグ, 7-32
 - 属性設定、コンパイル時 JML, C-4
 - タグの概要、カテゴリ, 7-19
 - タグの説明、表記規則と注意事項, 7-28
 - タグ・ライブラリ記述ファイル, 7-21
 - 要件, 7-18
- JML の styleSheet タグ, 5-10
- JML の transform タグ, 5-10
- JML の式、コンパイル時 JML
 - 構文, C-4
 - 属性設定, C-4
- JML のデータ型
 - 説明, 5-2
 - 例, 5-6
- jsp fallback タグ (plugin タグと併用), 1-20
- jsp forward タグ, 1-19
- jsp getProperty タグ, 1-17
- jsp include タグ, 1-18
- jsp param タグ, 1-17
- jsp plugin タグ, 1-20
- jsp setProperty タグ, 1-17
- jsp useBean タグ
 - 構文, 1-16
 - サンプル・アプリケーション, 9-3
- JspPage インタフェース, B-11
- JspScopeEvent クラス、イベント処理, 5-30
- JspScopeListener
 - イベント処理での使用方法, 5-30
 - サンプル・アプリケーション, 9-29
- jspService() メソッド, B-11
- JSP からのサーブレットの起動、サーブレットから
 - JSP, 3-6
- JSP からのサーブレットのコール、サーブレットから
 - JSP, 3-6
- JSP とサーブレットの相互作用
 - JSP からのサーブレットの起動, 3-6
 - サーブレットからの JSP の起動、要求
 - ディスパッチャ, 3-7
 - サンプル・コード, 3-9
 - データの受渡し、JSP からサーブレット, 3-7
 - データの受渡し、サーブレットから JSP, 3-8
- JSP トランスレータ、「トランスレータ」を参照
- JSP ページの要求, 1-7
- JSP ファイル拡張子のマッピング, A-9
- JSP ページ内の JDBC
 - サーバー側内部ドライバ (OSE 用), 4-26
 - サンプル・アプリケーション, 9-9
 - パフォーマンス強化, 4-5
 - 必須ファイル, A-5

- JSP ページによるサーブレットのラッピング, 4-31
- JSP ページの実行, 1-6
- JSP マークアップ言語, 「JML」を参照
- JSWDK
 - CLASSPATH の構成, A-8
 - config、ファイル名拡張子のマッピング, A-10
 - OracleJSP のサポート, 2-10
 - 構成パラメータの設定, A-24

L

- loadjava ツール (Oracle8i へのロード)
 - オプション構文の詳細, 6-35
 - 概要, 6-34
 - クラス・ファイルとしての変換済みページのロード, 6-55
 - ソース・ファイルとしての変換済みページのロード, 6-57
 - 変換済みページのロード, 6-55
 - 未変換ページのロード, 6-38
- lock タグ、コンパイル時 JML, C-11

N

- NLS サポート
 - translate_params に依存しないサンプル, 8-9
 - translate_params に依存するサンプル, 8-6
 - 概要, 8-1
 - コンテンツ型の設定 (静的), 8-2
 - コンテンツ型の設定 (動的), 8-3
 - マルチバイト・パラメータのコード化, 8-4
- noCompile、ojspc のオプション, 6-30

O

- ojspc 事前変換ツール
 - Oracle8i に配布するための事前変換, 6-51
 - SQLJ の JSP ページ, 6-52
 - オプション概要一覧, 6-24
 - オプションの説明, 6-26
 - 概要, 6-22
 - 機能の概要, 6-23
 - コマンドライン構文, 6-25
 - 出力ファイル、位置、関連オプション, 6-32
 - 主要機能とオプション, 6-53
 - 非 OSE 環境向けの使用, 6-64
 - ホットロードの有効化, 6-52

- 最も単純な使用方法, 6-51
- 例, 6-54
- ojsp.jar、必須ファイル, A-4
- ojsputil.jar、必須ファイル, A-4
- Oracle Application Server、OracleJSP のサポート, 2-9
- Oracle HTTP Server
 - mod_jserv, 2-6
 - mod_ose, 2-6
 - OracleJSP でのロール, 2-5
 - 使用するメリット, 2-8
- Oracle Internet Application Server
 - Apache/JServ の使用, 4-30
 - OracleJSP のサポート, 2-4
- Oracle Servlet Engine
 - JNDI の使用, 4-29
 - JSP と PL/SQL Pages の統合, 2-15
 - OSE JSP コンテナ, 2-19
 - OSE 内の JSP からの EJB のコール, 4-4
 - URL, 6-14
 - 概要, 2-3
 - 仮想バス, 6-14
 - 構成の概要, A-25
 - 構成パラメータ、等価コード, 4-29
 - サーバー側 JDBC 接続, 4-26
 - 事前変換モデルの概要, 2-18
 - 使用するメリット, 2-8
 - 静的ファイル, 6-17
 - ドキュメント・ルート、iAS との比較, 6-63
 - 特記事項の概要, 4-25
- Oracle Servlet Engine の JNDI, 4-29
- Oracle Web-to-go、OracleJSP のサポート, 2-9
- Oracle8i JVM の JVM, 4-26
- Oracle8i に配布するための事前変換, 6-51
- Oracle8i への配布
 - loadjava ツールの概要, 6-34
 - ojspc 事前変換ツール, 6-22
 - 概要、機能とロジスティック, 6-11
 - クライアント側変換, 6-50
 - サーバー側変換, 6-38
 - サーバー側変換とクライアント側変換の比較, 6-18
 - 静的ファイルの位置, 6-17
 - セッション・シェル・ツールの概要, 6-36
 - ツールの概要, 6-22
 - ホットロードの概要, 6-20
- OracleJSP トランスレータ, 「トランスレータ」を参照
- OracleJSP の移植性, 2-2
- OracleJSP のシステム要件, A-2

OracleJSP の実行モデル, 2-18
OracleJSP をサポートする Oracle プラットフォーム
 JDeveloper, 2-20
 Oracle Application Server, 2-9
 Oracle Internet Application Server, 2-4
 Oracle Web-to-go, 2-9
 リリースの概要, 2-16
otherwise タグ、JML, 7-33
out オブジェクト (暗黙的), 1-15

P

packageName、ojspc のオプション, 6-30
packageName、publishjsp のオプション, 6-41
page_repository_root 構成パラメータ, A-20
pageContext オブジェクト (暗黙的), 1-15
page オブジェクト (暗黙的), 1-14
page ディレクティブ
 NLS 用の contentType 設定, 8-2
 概要, 1-9
 特性, 4-14
page 有効範囲 (JSP オブジェクト), 1-14
param タグ, 1-17
PL/SQL Server Pages、OracleJSP との併用, 2-15
plugin タグ, 1-20
plugin タグ、コンパイル時 JML, C-14
print タグ、JML, C-13
properties、publishservlet のオプション, 6-61
PSP ページ、OracleJSP との併用, 2-15
publishjsp コマンド
 SQLJ の JSP ページの公開, 6-48
 概要, 6-39
 構文とオプション, 6-40
 例, 6-44
publishservlet コマンド
 概要, 6-59
 構文とオプション, 6-59
 例, 6-61

R

remove タグ、JML, 7-31
RequestDispatcher インタフェース, 3-7
request オブジェクト (暗黙的), 1-14
request 有効範囲 (JSP オブジェクト), 1-14
resolver、publishjsp のオプション, 6-43
response オブジェクト (暗黙的), 1-14

return タグ、JML, 7-36
reuse、publishservlet のオプション, 6-61
runtimeXX.zip、SQLJ 用の必須ファイル, A-5

S

schema、publishjsp のオプション, 6-41
servlet.jar
 バージョン, A-5
 必須ファイル, A-4
servletName、publishjsp のオプション, 6-41
servletName、publishservlet のオプション, 6-60
sess_sh、「セッション・シェル」を参照
session_OnEnd タグ、globals.jsa, 5-42
session_OnStart タグ、globals.jsa, 5-41
session_sharing 構成パラメータ, A-20
session オブジェクト (暗黙的), 1-15
session 有効範囲 (JSP オブジェクト), 1-14
setContentType() メソッド、NLS, 8-3
setProperty タグ, 1-17
setProperty タグ、コンパイル時 JML, C-9
set タグ、コンパイル時 JML, C-10
showVersion、publishjsp のオプション, 6-41
SQLJ
 JSP コードの例, 5-31
 JSP 用の必須ファイル, A-5
 Oracle SQLJ オプションの設定, 5-34
 OracleJSP のサポート, 5-31
 publishjsp を使用した SQLJ の JSP ページの公開,
 6-48
 sqljcmd 構成パラメータ, A-21
 sqljsp ファイル, 5-33
 SQLJ オプション用の option の S オプション, 6-31
 SQLJ トランスレータの実行タイミング, 5-33
 サーバー側の SQLJ オプション, 6-48
 サンプル・アプリケーション, 9-34
sqljcmd 構成パラメータ, A-21
SQLJ の sqljsp ファイル, 5-33
SQL タグ
 概要、タグ・リスト, 5-22
 要件, 5-22
 例, 5-26
srcdir、ojspc のオプション, 6-31
SSL セッション, B-6
stateless、publishjsp のオプション, 6-43
stateless、publishservlet のオプション, 6-60
styleSheet タグ、JML, 5-10

Sun Microsystems JSWDK, 「JSWDK」を参照
S、option のオプション (SQLJ オプション用), 6-31

T

Tag-Extra-Info クラス (タグ・ライブラリ)
一般的な用途、getVariableInfo() メソッド, 7-8
サンプル Tag-Extra-Info クラス, 7-15
taglib ディレクティブ
Oracle JML タグ, 7-20
Oracle SQL タグ, 5-22
一般的な用途, 7-12
完全 TLD 名および位置の使用, 7-13
構文, 1-10
コンパイル時 JML, C-2
ショートカット URI の使用, 7-12
TLD ファイル, 「タグ・ライブラリ記述ファイル」を参照
Tomcat
CLASSPATH の構成, A-9
config、ファイル名拡張子のマッピング, A-11
OracleJSP のサポート, 2-10
構成パラメータの設定, A-24
transform タグ、JML, 5-10
translate_params 構成パラメータ
依存しない NLS サンプル, 8-9
依存する NLS サンプル, 8-6
概要, A-22
概要、マルチバイト・パラメータのコード化, 8-4
等価コード, 8-6
非マルチバイト・サーブレット・コンテナのオー
バーライドにおける効果, 8-5
translator.zip、SQLJ 用の必須ファイル, A-5

U

unpublishjsp コマンド, 6-50
unpublishservlet コマンド, 6-62
unsafe_reload 構成パラメータ, A-22
URL
Oracle Servlet Engine, 6-14
URL のリライト, B-6
コンテキスト・パス, 6-15
サーブレット・パス, 6-15
usage、publishjsp のオプション, 6-41

useBean タグ
構文, 1-16
サンプル・アプリケーション, 9-3
useBean タグ、コンパイル時 JML, C-8
useCookie タグ、JML, 7-30
useForm タグ、JML, 7-29
useVariable タグ、JML, 7-29

V

verbose、ojspc のオプション, 6-32
verbose、publishjsp のオプション, 6-43
version、ojspc のオプション, 6-32
virtualpath、publishjsp のオプション, 6-41
virtualpath、publishservlet のオプション, 6-60

W

WAR の配布, 6-66
web.xml、タグ・ライブラリ用の使用方法, 7-11
Web アプリケーションの階層, B-8
when タグ、JML, 7-33

X

xmlparserv2.jar、必須ファイル, A-4
XML/XSL のサポート
XML の代替構文, 5-9
XSL スタイルシート用の JML タグ, 5-10
XSL の変換例, 5-11
概要, 5-8
サンプル・アプリケーション, 9-33
xsu12.jar または xsu111.jar、オプション・ファイル,
A-4

あ

アクション・タグ
forward タグ, 1-19
getProperty タグ, 1-17
include タグ, 1-18
param タグ, 1-17
plugin タグ, 1-20
setProperty タグ, 1-17
useBean タグ, 1-16
概要, 1-16

アプリケーション・イベント
 globals.jsa を使用, 5-39
 JspScopeListener, 5-30
 サーブレット・アプリケーションのライフ・
 サイクル, B-7
アプリケーション階層, B-8
アプリケーション相対パス, 1-8
アプリケーションのサポート
 globals.jsa を使用, 5-36
 概要, 3-5
 サーブレット・アプリケーションのライフ・
 サイクル, B-7
アプリケーション・ルート機能, 3-3
暗黙的な JSP オブジェクト
 暗黙的オブジェクトの使用, 1-16
 概要, 1-14

い

イベント処理
 globals.jsa を使用, 5-39
 HttpSessionBindingListener, 3-10
 JspScopeListener, 5-30
 サーブレット・アプリケーションのライフ・
 サイクル, B-7

え

エラー処理 (ランタイム), 3-16

お

応答オブジェクト、サーブレット, B-9
オブジェクトと有効範囲 (JSP オブジェクト), 1-13
オンデマンド変換 (ランタイム), 1-6, 2-18

か

開発環境, 3-2
外部リソース・ファイル
 external_resource パラメータを使用, A-19
 ojspc の extres オプションを使用, 6-29
 静的テキスト, 4-12
拡張機能
 Database-Access JavaBeans の概要, 2-12
 globals.jsa の概要 (アプリケーション・サポート),
 2-14

JML タグ・ライブラリの概要, 2-12
JspScopeListener の概要, 2-14
Oracle 固有の拡張機能の概要, 2-13
PL/SQL Server Pages のサポートの概要, 2-15
SQLJ サポートの概要, 2-13
SQL タグ・ライブラリの概要, 2-12
XML/XSL サポートの概要, 2-11
移植可能な拡張機能の概要, 2-11
拡張 NLS サポートの概要, 2-14
拡張データ型の概要, 2-11
サーブレット 2.0 向けの拡張機能, 2-2
プログラム拡張機能の概要, 2-10
カスタム・タグ, 「タグ・ライブラリ」を参照
仮想パス (OSE の URL), 6-14
各国語サポート, 「NLS」を参照
環境、開発と配布, 3-2
完全名、スキーマ・オブジェクト, 6-13

き

行セットのキャッシュ, 4-8
行のプリフェッチ
 OracleJSP の ConnBean を使用, 5-14
 概要, 4-7
行のプリフェッチ, 「行のプリフェッチ」を参照

く

クライアント側の考慮事項, 3-3
クライアント側変換、Oracle8i への配布
 ojspc を使用した事前変換, 6-51
 publishservlet を使用したページの公開, 6-59
 概要, 6-50
 サーバー側変換との比較, 6-18
 ページ実装クラスのホットロード, 6-58
 変換済みページのロード, 6-55
クラスの再ロード、動的, 4-24
クラスの動的再ロード, 4-24
クラスのネーミング、トランスレータ, 6-5
クラス・ローダーの問題, 4-19

こ

更新のバッチ処理, 「バッチ更新」を参照

構成

CLASSPATH とクラス・ローダーの問題, 4-19

CLASSPATH, Apache/JServ, A-7

CLASSPATH, JSWDK, A-8

CLASSPATH, Tomcat, A-9

JSP ファイル名拡張子のマッピング, A-9

Web サーバーおよびサーブレット環境, A-6

Web サーバーの CLASSPATH, A-7

概要, OSE 構成, A-25

構成パラメータの設定, A-23

構成パラメータの説明 (OSE 以外), A-16

構成パラメータの等価コード, OSE, 4-29

構成パラメータ、概要表 (OSE 以外), A-13

実行の最適化, 4-18

パラメータの設定, Apache/JServ, A-23

パラメータの設定, JSWDK, A-24

パラメータの設定, Tomcat, A-24

ファイル名拡張子のマッピング, Apache/JServ,
A-10

ファイル名拡張子のマッピング, JSWDK, A-10

ファイル名拡張子のマッピング, Tomcat, A-11

構成パラメータ (非 OSE)

概要表と説明, A-13

設定, A-23

構文 (概要), 1-8

コード、トランスレータにより生成, 6-2

コメント (JSP コード内), 1-12

コンテキスト相対パス, 1-8

コンテキスト・パス、URL, 6-15

コンテナ

JSP コンテナ, 1-6

OSE JSP コンテナ, 2-19

サーブレット・コンテナ, B-3

コンテンツ型の設定

静的 (page ディレクティブ), 8-2

動的 (setContent-type メソッド), 8-3

コンパイル

javac 構成パラメータ, A-19

ojspc の noCompile オプション, 6-30

コンパイル時 JML タグ

taglib ディレクティブ, C-2

構文のサポート, C-3

タグの概要と説明, C-6

さ

サーバー側 JDBC ドライバ, 4-26

サーバー側変換、Oracle8i への配布

Oracle8i への未変換ページのロード, 6-38

概要, 6-38

クライアント側変換との比較, 6-18

変換と公開、publishjsp, 6-39

サービス・メソッド、JSP, B-11

サーブレット

JSP ページによるサーブレットのラッピング, 4-31

アプリケーションのライフ・サイクル管理, B-7

技術的背景情報, B-2

サーブレット・インタフェース, B-3

サーブレット・オブジェクト, B-8

サーブレット構成オブジェクト, B-10

サーブレット・コンテキスト, B-6

サーブレット・コンテキスト・オブジェクト, B-10

サーブレット・コンテナ, B-3

サーブレット・セッション, B-4

サーブレット・テクノロジーの概要, B-2

サーブレットの起動, B-8

セッション・オブジェクト, B-9

セッションの共有、JSP、Apache/JServ, 4-33

要求オブジェクトと応答オブジェクト, B-9

サーブレット 2.0 環境

globals.jsa サンプル・アプリケーション, 9-38

globals.jsa を介して追加されるサポート, 5-34

OracleJSP のアプリケーション・ルート機能, 3-4

OracleJSP の機能性の概要, 2-2

サーブレット・コンテキスト

概要, B-6

サーブレット・コンテキスト・オブジェクト, B-10

サーブレット・コンテナ, B-3

サーブレット・セッション

HttpSession インタフェース, B-4

セッションの追跡, B-5

サーブレットと JSP の相互作用

JSP からのサーブレットの起動, 3-6

サーブレットからの JSP の起動、要求

ディスパッチャ, 3-7

サンプル・コード, 3-9

データの受渡し、JSP からサーブレット, 3-7

データの受渡し、サーブレットから JSP, 3-8

サーブレット・パス、URL, 6-15

サーブレット・ライブラリ, A-4

最適化

- HTTP セッション使用の回避, 4-19
- JSP ページのアンバッファリング, 4-18
- 再変換チェックの回避, 4-18

サンプル・アプリケーション

- ConnBean のサンプル, 9-22
- ConnCacheBean のサンプル, 9-25
- CursorBean のサンプル, 9-23
- Database-Access JavaBeans のサンプル, 9-20
- DBBean のサンプル, 9-21
- globals.jsa のサンプル, 9-38
- globals.jsa、アプリケーション・イベント, 9-38
- globals.jsa、アプリケーションおよびセッション・イベント, 9-41
- globals.jsa、グローバル宣言, 9-43
- hello ページ, 9-2
- HttpSessionBindingListener のサンプル, 3-11
- JDBC のサンプル, 9-9
- JML タグのサンプル, 9-27
- JML のデータ型の例, 5-6
- JspScopeListener、イベント処理, 9-29
- JSP とサーブレットの相互作用, 3-9
- NLS、translate_params に依存, 8-6
- NLS、translate_params に依存しない, 8-9
- SQLJ の問合せ, 9-34
- SQLJ の例, 5-31
- SQL タグの例, 5-26
- useBean ページ, 9-3
- XML 問合せの出力, 9-33
- カスタム・タグの定義と使用, 7-13
- 基本的なサンプル, 9-2
- ショッピング・カート・ページ, 9-4
- 接続キャッシュ・ページ, 9-16
- データベース・アクセス、初期サンプル, 3-19
- 問合せ Bean, 9-13
- 問合せページ (単純), 9-10
- ページ実装クラス・コード, 6-7
- ユーザー指定の問合せページ, 9-11

し

式, 1-11

事前変換

- OSE (概要), 2-18
- クライアント側 (OSE), 6-51
- サーバー側 (OSE), 6-39
- 実行なし、一般, 6-64

事前変換ツール、ojspc, 6-22

出力ファイル

- ojspc の d オプション (バイナリ位置), 6-27
- ojspc の srcdir オプション (ソースの位置), 6-31
- page_repository_root 構成パラメータ, A-20
- 位置, 6-7
- 位置と関連オプション、ojspc, 6-32
- トランスレータにより生成, 6-6

す

スキーマ・オブジェクト

- Java ファイルのロードによる作成, 6-12
- Java 用, 6-12
- 完全名と短縮名, 6-13
- 公開, 6-14
- パッケージの決定, 6-13

スクリプト変数 (タグ・ライブラリ)

- 定義, 7-7
- 有効範囲, 7-8

スクリプト要素

- 概要, 1-11
- コメント, 1-12
- 式, 1-11
- スクリプトレット, 1-11
- 宣言, 1-11

スクリプトレット

- JavaBeans との比較, 4-2
- 概要, 1-11

せ

生成されるコード、トランスレータ, 6-2

静的挿入

- ディレクティブ, 1-9
- 動的挿入, 4-8
- ロジスティック, 4-9

静的テキスト

- external_resource パラメータ, A-19
- 外部リソース・ファイル, 4-12
- 外部リソース、ojspc の extres オプション, 6-29
- 生成されるインナー・クラス, 6-3
- 大量の静的コンテンツに関する回避策, 4-12

静的テキスト用のインナー・クラス, 6-3

静的ファイル、Oracle Servlet Engine, 6-17

セッション・イベント

- globals.jsa を使用, 5-41
- HttpSessionBindingListener, 3-10
- JspScopeListener, 5-30

セッション・オブジェクト、サーブレット、B-9

セッション・シェル・ツール

- createcontext コマンド, 6-15
- java コマンド, 6-58
- publishjsp コマンド, 6-39
- publishservlet コマンド, 6-59
- unpublishjsp コマンド, 6-50
- unpublishservlet コマンド, 6-62
- 概要, 6-36
- 主要コマンド, 6-37
- 主要な構文要素, 6-36

セッションの共有

- session_sharing 構成パラメータ, A-20
- 概要、JSP とサーブレット、Apache/JServ, 4-33

セッションのサポート

- globals.jsa を使用, 5-36
- 概要, 3-5
- デフォルトのセッション要求, 3-5

セッションの追跡, B-5

接続のキャッシュ

- ConnCacheBean JavaBeans を使用, 5-15
- 概要, 4-5
- サンプル・アプリケーション, 9-16

接続、サーバー側（OSE 用）、4-26

宣言

- グローバル宣言、globals.jsa, 5-44
- メソッド変数とメンバー変数, 4-13
- メンバー変数, 1-11

そ

相互作用、JSP とサーブレット, 3-6

ソースの位置、ojspc の srcdir オプション, 6-31

た

タグ・ハンドラ（タグ・ライブラリ）

- 外部タグ・ハンドラへのアクセス, 7-9
- 概要, 7-4
- サンプル・タグ・ハンドラ・クラス, 7-14
- 本体付きのタグ, 7-6
- 本体なしのタグ, 7-5

タグ・ライブラリ

- Oracle JML タグの説明, 7-28
- Oracle JML タグ、概要, 7-18
- Oracle SQL タグ, 5-22
- Tag-Extra-Info クラス, 7-7
- taglib ディレクティブ, 7-12
- web.xml の使用, 7-11
- 概要, 1-21
- スクリプト変数, 7-7
- タグ・ハンドラ, 7-4
- タグ・ライブラリ記述ファイル, 7-10
- 定義と使用、全体の例, 7-13
- 標準実装の概要, 7-2
- 標準フレームワーク, 7-2
- 方針、作成する場合, 4-10
- ランタイム実装とコンパイル時実装, C-2

タグ・ライブラリ記述ファイル

- Oracle JML タグ, 7-21
- Oracle SQL タグ, 5-22
- web.xml でのショートカット URI の定義, 7-11
- 一般的な機能, 7-10
- サンプル・ファイル, 7-17

短縮名、スキーマ・オブジェクト, 6-13

ち

チェッカ・ページ, 4-11

つ

ツール

- loadjava の概要（Oracle8i へのロード）、6-34
- Oracle8i への配布, 6-22
- クライアント側変換用の ojspc, 6-22
- セッション・シェルの概要, 6-36

て

ディレクティブ

- include ディレクティブ, 1-9
- page ディレクティブ, 1-9
- taglib ディレクティブ, 1-10
- 概要, 1-9
- グローバル・ディレクティブ、globals.jsa, 5-44
- ディレクトリ別名の変換、「別名の変換」を参照

データ型

- JmlBoolean 拡張型, 5-3
- JmlFPNumber 拡張型, 5-4
- JmlNumber 拡張型, 5-4
- JmlString 拡張型, 5-6
- JML のデータ型の例, 5-6
- Oracle JML の拡張データ型, 5-2
- OracleJSP の拡張機能の概要, 2-11

データベース・アクセス

- Database-Access JavaBeans, 5-12
- サーバー側 JDBC 接続, 4-26
- 初期サンプル, 3-19
- 方法, 2-7

データベース・スキーマ・オブジェクト, 「スキーマ・オブジェクト」を参照

デバッグ

- debug、ojspc のオプション, 6-28
- emit_debuginfo 構成パラメータ, A-18
- JDeveloper を使用, 2-20

バイナリ・ファイルの位置、ojspc の d オプション, 6-27

バイナリ・ファイルの配布, 6-65

配布環境, 3-2

配布、全般的な考慮事項

JDeveloper を使用したページの配布, 6-67

WAR の配布, 6-66

概要, 6-62

実行なしの一般的な事前変換, 6-64

ドキュメント・ルート、iAS と OSE, 6-63

バイナリ・ファイルのみの配布, 6-65

非 OSE 環境向けの ojspc, 6-64

パッケージのネーミング

ojspc の packageName オプション, 6-30

publishjsp の packageName オプション, 6-41

トランスレータ, 6-4

バッチ更新

OracleJSP の ConnBean を使用, 5-14

概要, 4-7

と

動的挿入

- Apache/JServ のための特殊サポート, 4-31
- アクション・タグ, 1-18
- 静的挿入, 4-8
- 大量の静的コンテンツ, 4-12
- ロジスティック, 4-9

動的転送、Apache/JServ のための特殊サポート, 4-31

ドキュメント・ルート

- iAS と OSE, 6-63
- 機能, 3-3

トランスレータ

- 出力ファイルの位置, 6-7
- 生成されるインナー・クラス、静的テキスト, 6-3
- 生成されるクラス名, 6-5
- 生成されるコードの機能, 6-2
- 生成されるコードのサンプル, 6-7
- 生成されるパッケージ名, 6-4
- 生成されるファイル, 6-6

は

バージョン・ナンバー、OracleJSP、表示するコード, 2-17

バイナリ・データ、JSP で回避する理由, 4-17

ひ

ヒント

JavaBeans とスクリプトレット, 4-2

JSP でのバイナリ・データの使用を回避, 4-17

JSP の空白表現, 4-15

page ディレクティブの特性, 4-14

回避策、大量の静的コンテンツ, 4-12

構成上の重要な問題, 4-18

サブレット・ラッパーとしての JSP ページ, 4-31

静的挿入と動的挿入, 4-8

タグ・ライブラリを作成する場合, 4-10

チェッカ・ページの使用, 4-11

メソッド変数宣言とメンバー変数宣言, 4-13

ふ

ファイル

OracleJSP の必須ファイル, A-4

位置、ojspc の d オプション, 6-27

位置、ojspc の srcdir オプション, 6-31

位置、page_repository_root 構成パラメータ, A-20

位置、トランスレータの出力, 6-7

トランスレータにより生成, 6-6

ファイル拡張子、マッピング, A-9

文のキャッシュ

OracleJSP の ConnBean を使用, 5-13

OracleJSP の ConnCacheBean を使用, 5-16

概要, 4-6

へ

ページ・イベント (JspScopeListener), 5-30

ページ実装クラス

概要, 1-6

サンプル・コード, 6-7

生成されるコード, 6-2

ページ相対パス, 1-8

ページの再変換、動的, 4-22

ページの再ロード、動的, 4-23

ページの動的再変換, 4-22

ページの動的再ロード, 4-23

別名の変換、Apache/JServ

alias_translation 構成パラメータ, A-16

概要, 4-34

変換

オンデマンド (ランタイム), 1-6

クライアント側での事前変換 (OSE), 6-51

サーバー側での事前変換 (OSE), 6-39

サーバー側とクライアント側の比較 (OSE の場合),
6-18

ほ

ホットロード (OSE)

ojspc の hotload オプション, 6-29

ojspc を介した有効化, 6-52

publishjsp の hotload オプション, 6-42

概要, 6-20

機能とメリット, 6-21

ページ実装クラスのホットロード, 6-58

有効化と実行, 6-20

ま

マルチバイト・パラメータのコード化, NLS, 8-4

め

明示的な JSP オブジェクト, 1-13

メソッド変数宣言, 4-13

メンバー変数宣言, 4-13

ゆ

有効範囲 (JSP オブジェクト), 1-13

よ

要求イベント (JspScopeListener), 5-30

要求オブジェクト、サーブレット, B-9

要求ディスパッチャ (JSP とサーブレットの
相互作用), 3-7

要件

OracleJSP のシステム要件, A-2

必須ファイルの概要, A-4

ら

ランタイムの考慮事項

クラスの動的再ロード, 4-24

ページの動的再変換, 4-22

ページの動的再ロード, 4-23

り

リソース管理

OracleJSP の拡張機能の概要, 3-15

アプリケーション (JspScopeListener), 5-30

セッション (JspScopeListener), 5-30

標準的なセッション管理, 3-10

ページ (JspScopeListener), 5-30

要求 (JspScopeListener), 5-30

リリース番号、OracleJSP、表示するコード, 2-17