Oracle9*i*

JavaServer Pages サポート・リファレンス

リリース 2（9.2）

2002 年 7 月

部品番号 : J06310-01

ORACLE®

Oracle9*i* JavaServer Pages サポート・リファレンス , リリース 2（9.2）

部品番号 : J06310-01

原本名 ：Oracle9*i* Support for JavaServer Pages Reference, Release 2 (9.2)

原本部品番号 ：A96657-01

原本著者 ：Brian Wright

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

目次

2 Oracle JSP 実装の概要

3 基本事項

4 重要な考慮点

5 Oracle 固有のプログラミング拡張機能

6 JSP による変換および配置

9 Apache JServ での Oracle JSP

A サード・パーティ・ライセンス

索引

はじめに

このマニュアルでは、Sun 社が仕様を策定した JavaServer Pages（JSP）テクノロジの Oracle 実装について説明します。Sun 社が仕様を策定した標準機能の概要を示しますが、主に Oracle 実装の詳細および付加価値機能について説明します。

Oracle9*i* リリース 2（9.2）に付属の Oracle JSP コンテナは、Sun 社の JavaServer Pages 仕様バージョン 1.1 の完全な実装です。

ここでは、次の項目について説明します。

重要： Oracle9*i* リリース 2（9.2）には、バージョン 1.1.2 の Oracle JSP コンテナが付属しています。

対象読者

このマニュアルは、JavaServer Pages テクノロジに基づいた Web アプリケーションを作成する開発者を対象としています。動作中の Web 環境およびサーブレット環境が存在し、読者が次の内容を十分に理解していることを前提としています。

- 一般的な Web テクノロジ
- 一般的なサーブレット・テクノロジ
- Web サーバー環境およびサーブレット環境の構成方法
- HTML
- Java
- Oracle JDBC（Oracle データベースにアクセスする JSP アプリケーション用）
- Oracle SQLJ（SQLJ を使用する JSP データベース・アプリケーション用）

第 1 章などでは、標準の JSP 1.1 テクノロジおよび構文について簡単に説明していますが、詳細は説明しません。標準の JSP 1.1 機能の詳細は、Sun 社の JavaServer Pages 仕様バージョン 1.1 またはその他の該当するリファレンス情報を参照してください。

JSP 1.1 仕様は Servlet 2.2 環境に依存しているため、このマニュアルでは Servlet 2.2 の機能について説明しています。ただし、Oracle9*i* に付属の JSP コンテナには、2.2 より前のサーブレット環境用の特別な機能があります。Servlet 2.0 環境に関連するこれらの特別な機能（特に Oracle9*i* に付属の Apache JServ）については、第 9 章を参照してください。

このマニュアルの構成

このマニュアルは、次のように構成されています。

第 1 章「概要」

この章では、標準の JSP1.1 テクノロジの概要を示します。

第 2 章「Oracle JSP 実装の概要」

この章では、Oracle JSP の機能および拡張機能の概要を示し、Oracle9*i* リリース 2（9.2）に付属の JSP コンテナ、Servlet コンテナおよび Web サーバーについて説明します。

第 3 章「基本事項」

この章では、JSP プログラミングの基本的な考慮点について説明し、データベースにアクセスするためのスタータ・サンプルを示します。

第 4 章「重要な考慮点」

この章では、開発者が注意する必要がある様々なプログラミングおよび構成の一般的な問題について説明します。

第 5 章「Oracle 固有のプログラミング拡張機能」

この章では、Oracle9*i* リリース 2（9.2）に付属の Oracle JSP コンテナによって提供される Oracle 固有の（移植不可能な）拡張機能について説明します。

第 6 章「JSP による変換および配置」

この章では、Oracle JSP による変換および配置の機能および問題について説明し、事前変換ツール `ojspc` について説明します。

第 7 章「JSP のタグ・ライブラリ」

この章では、カスタム・タグ・ライブラリ用の基本の JSP 1.1 フレームワークについて説明します。

第 8 章「Oracle JSP のグローバリゼーション・サポート」

この章では、グローバリゼーション・サポート用の標準機能および Oracle 固有の機能について説明します。

第 9 章「Apache JServ での Oracle JSP」

この章では、JServ Servlet 2.0 環境で Oracle JSP コンテナを使用する方法を詳細に説明し、必須ファイル、配置、構成、およびプログラミングに関連する特別な考慮点を示します。

付録 A「サード・パーティ・ライセンス」

この付録では、Oracle9*i* リリース 2（9.2）に付属し、このマニュアルで説明されているサード・バーティ製品のサード・パーティ・ライセンスを示します。

関連文書

次の Oracle9*i* リリースのマニュアルも参照してください。

- 『Oracle9*i* Java Developer's Guide』

  このマニュアルでは、Oracle9*i* での Java の基本的な概念を説明し、サーバー側の構成および機能に関する一般的な情報を示します。JDBC や SQLJ などの特定の製品ではなく、Oracle データベースの Java 環境に関する一般的な情報を示します。

- 『Oracle9*i* JDBC 開発者ガイドおよびリファレンス』

  このマニュアルでは、（Java Database Connectivity 用の）JDBC 標準の Oracle 実装のプログラミング構文および機能について説明します。Oracle JDBC Drivers の概要、JDBC 1.22、2.0 および 3.0 機能の Oracle 実装の詳細、および Oracle JDBC の型の拡張とパフォーマンスの向上について説明します。

- 『Oracle9*i* SQLJ 開発者ガイドおよびリファレンス』

  このマニュアルでは、SQLJ を使用して Java コードに静的 SQL 操作を直接埋め込む方法を説明します。SQLJ 言語の構文、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ます。標準の SQLJ 機能および Oracle 固有の SQLJ 機能の両方について説明します。

- 『Oracle9*i* JPublisher ユーザーズ・ガイド』

  このマニュアルでは、Oracle JPublisher ユーティリティを使用してオブジェクト型およびその他のユーザー定義型を Java クラスに変換する方法について説明します。オブジェクト型、VARRAY 型、ネストした表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、JPublisher はこれらの型にマップするためのカスタム Java クラスを生成できます。

- 『Oracle9*i* Java Stored Procedures Developer's Guide』

  このマニュアルでは、Java ストアド・プロシージャ（Oracle9*i* で直接実行するプログラム）について説明します。ストアド・プロシージャ（ファンクション、プロシージャ、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・レベルで実装できるため、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が向上します。

次の Oracle9*i* Application Server リリースの Oracle9*i*AS Containers for J2EE（OC4J）のマニュアルも参照してください。

- 『Oracle9*i*AS Containers for J2EE User's Guide』

  このマニュアルでは、OC4J の概要および一般的な情報を示します。主に、サーブレット、JSP ページおよび EJB について説明し、一般的な構成および配置方法を示します。

- 『Oracle9*i*AS Containers for J2EE JavaServer Pages サポート・リファレンス』

  このマニュアルでは、OC4J でページを実行する必要がある JSP 開発者向けの情報を示します。JSP 標準の概要およびプログラミングの考慮点を示します。また、Oracle の付加価値機能および OC4J 環境で作業を開始する手順も説明します。

- 『Oracle9*i*AS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』

  このマニュアルでは、タグ・ライブラリ、JavaBeans および OC4J で提供される他の Java ユーティリティに関する概念情報、および構文と使用方法の詳細情報を示します。

- 『Oracle9*i*AS Containers for J2EE Servlet Developer's Guide』

  このマニュアルでは、OC4J でのサーブレットおよび Servlet コンテナの使用に関するサーブレット開発者向けの情報を示します。また、関連する OC4J 構成ファイルについても説明します。

- 『Oracle9*i*AS Containers for J2EE Services Guide』

  このマニュアルでは、OC4J で提供される基本の Java サービス（JTA、JNDI、Oracle9*i* Application Server Java Object Cache など）に関する情報を示します。

- 『Oracle9*i*AS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference』

  このマニュアルでは、OC4J での EJB 実装および EJB コンテナに関する情報を示します。

次のマニュアルも参照してください。

- 『Oracle9*i* XML Developer's Kit ガイド - XDK』
- 『Oracle9*i* アプリケーション開発者ガイド - 基礎編』
- 『Oracle9*i* Java パッケージ・プロシージャ・リファレンス』
- 『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』
- 『PL/SQL ユーザーズ・ガイドおよびリファレンス』
- 『Oracle9*i* SQL リファレンス』
- 『Oracle9*i* Net Services 管理者ガイド』
- 『Oracle9*i* Advanced Security 管理者ガイド』
- 『Oracle9*i* データベース・リファレンス』
- 『Oracle9*i* データベース・エラー・メッセージ』
- 『Oracle9*i* サンプル・スキーマ』
- 『Oracle9*i* Application Server Administrator's Guide』
- 『Oracle Enterprise Manager Administrator's Guide』
- 『Oracle HTTP Server Administration Guide』
- 『Oracle9*i* Application Server Performance Guide』
- 『Oracle9*i* Application Server Globalization Support Guide』
- 『Oracle9*i*AS Web Cache Administration and Deployment Guide』
- 『Oracle9*i* Application Server: Migrating from Oracle9*i* Application Server 1.x』
- JDeveloper のオンライン・ヘルプ
- Oracle Technology Network Japan（OTN-J）の JDeveloper に関するドキュメント :

  `http://otn.oracle.co.jp/products/jdev/index.html`

リリース・ノート、インストール・マニュアル、ホワイト・ペーパーまたはその他の関連文書は、OTN-J（Oracle Technology Network Japan）に接続すれば、無償でダウンロードできます。OTN-J を使用するには、オンラインでの登録が必要です。次の URL で登録できます。

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

すでに OTN-J のユーザー名およびパスワードを取得済であれば、次の OTN-J Web サイトの文書セクションに直接接続できます。

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

JavaServer Pages の詳細は、次の Oracle Technology Network Japan（OTN-J）リソースを参照してください。

- Java サーブレットおよび JavaServer Pages に関する OTN-J の Web サイト :

  `http://otn.oracle.co.jp/tech/java/servlets/`

- OTN-J の JSP Discussion Forum:

  `http://otn.oracle.co.jp/forum/index.html`

Sun 社の次のリソースも参照してください。

- JavaServer Pages の Web サイト（最新の仕様を含む）:

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

- Java サーブレット・テクノロジの Web サイト（最新の仕様を含む）:

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

- JavaServer Pages の `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、メッセージの本文に次の 1 文を入力して、`listserv@java.sun.com` に電子メールを送信します。

  `subscribe jsp-interest` *`yourlastname yourfirstname`*

  ただし、ポストされた電子メールのデイリー・ダイジェストのみを要求することをお薦めします。これを行うには、メッセージ本文に次の 1 行を追加します。

  `set jsp-interest digest`

表記規則

この項では、このマニュアルの本文およびコード例で使用される表記規則について説明します。この項の内容は次のとおりです。

本文中の表記規則

本文では、特別な用語をより迅速に識別できるように、様々な表記規則を使用します。次の表に、それらの表記規則を説明し、その使用例を示します。

表記規則	意味	例
固定幅フォントの大文字	固定幅フォントの大文字は、システムが提供する要素を示します。このような要素には、パラメータ、権限、データ型、Recovery Manager のキーワード、SQL キーワード、SQL*Plus またはユーティリティ・コマンド、パッケージおよびメソッドが含まれます。また、システムが提供する列名、データベース・オブジェクト、データベース構造、ユーザー名およびロールも含まれます。	NUMBER 列に対してのみに、この句を指定できます。 BACKUP コマンドを使用して、データベースのバックアップを取ることができます。 USER_TABLES データ・ディクショナリ・ビュー内の TABLE_NAME 列を問い合せます。 DBMS_STATS.GENERATE_STATS プロシージャを使用します。
固定幅フォントの小文字	固定幅フォントの小文字は、実行可能ファイル、ファイル名、ディレクトリ名およびユーザーが提供する要素のサンプルを示します。このような要素には、コンピュータ名、データベース名、ネット・サービス名および接続識別子が含まれます。また、ユーザーが提供するデータベース・オブジェクト、データベース構造、列名、パッケージ、クラス、ユーザー名、ロール、プログラム・ユニットおよびパラメータの値も含まれます。 **注意：**大文字と小文字を組み合せて使用するプログラム要素もあります。これらの要素は、記載されているとおり入力してください。	sqlplus と入力して、SQL*Plus をオープンします。 パスワードは、orapwd ファイルで指定します。 /disk1/oracle/dbs ディレクトリ内のデータ・ファイルおよび制御ファイルのバックアップを取ります。 hr.departments 表には、department_id、department_name および location_id 列があります。 QUERY_REWRITE_ENABLED 初期化パラメータを true に設定します。 oe ユーザーとして接続します。 JRepUtil クラスが次のメソッドを実装します。
固定幅フォントの小文字のイタリック	固定幅フォントの小文字のイタリックは、プレースホルダまたは変数を示します。	*parallel_clause* を指定できます。 *old_release*.SQL を実行します。ここで、*old_release* とはアップグレード前にインストールしたリリースを示します。

コード例中の表記規則

コード例では、SQL、PL/SQL、SQL*Plus または他のコマンドライン文を示します。コード例は、固定幅フォントで表示され、この例に示すとおり通常のテキストと区別されます。

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

次の表に、コード例で使用される表記規則を説明し、その使用例を示します。

表記規則	意味	例
[ ]	大カッコは、任意に選択する 1 つ以上の項目を囲みます。大カッコは入力しないでください。	DECIMAL (*digits* [, *precision*])
|	縦線は、大カッコまたは中カッコ内の 2 つ以上のオプションの選択項目を表します。いずれかのオプションを入力します。縦線は入力しないでください。	{ENABLE | DISABLE} [COMPRESS | NOCOMPRESS]
...	水平省略記号は、次のいずれかを示します。	
	■ 例に直接関連しないコードの一部が省略されている。	CREATE TABLE ... AS *subquery*;
	■ コードの一部を繰り返すことができる。	SELECT *col1*, *col2*, ... , *coln* FROM employees;
その他の句読点	大カッコ、中カッコ、縦線および省略記号以外の句読点は、表示されているとおり入力する必要があります。	acctbal NUMBER(11,2); acct CONSTANT NUMBER(4) := 3;
イタリック体	イタリック文は、プレースホルダまたは特定の値を指定する必要のある変数を示します。	CONNECT SYSTEM/*system_password* DB_NAME = *database_name*
大文字	大文字は、システムが提供する要素を示します。これらの用語は、ユーザー定義の用語と区別するために大文字で示されます。用語が大カッコ内にないかぎり、表示されているとおりの順序および綴りで入力します。ただし、これらの用語は大 / 小文字が区別されないため、小文字でも入力できます。	SELECT last_name, employee_id FROM employees; SELECT * FROM USER_TABLES; DROP TABLE hr.employees;
小文字	小文字は、ユーザーが提供するプログラム要素を示します。たとえば、表名、列名またはファイル名などです。 **注意 :** 大文字と小文字を組み合せて使用するプログラム要素もあります。これらの要素は、記載されているとおり入力してください。	SELECT last_name, employee_id FROM employees; sqlplus hr/hr CREATE USER mjones IDENTIFIED BY ty3MU9;

1

概要

この章では、JavaServer Pages テクノロジの標準機能の概要を説明します。詳細は、Sun 社の JavaServer Pages 仕様バージョン 1.1 を参照してください。

（Oracle 固有の JSP 機能の概要は、第 2 章「Oracle JSP 実装の概要」を参照してください。）

この章の内容は次のとおりです。

JavaServer Pages の概要

JavaServer Pages は、Web アプリケーション（Web サーバー上で実行しているアプリケーション）によって出力されるページに動的コンテンツを生成するための有効な方法として、Sun 社によって仕様が策定されたテクノロジです。

Java サーブレット・テクノロジと密接に組み合されたこのテクノロジを使用すると、Web ページの HTML コード（または XML などの他のマークアップ・コード）内に、Java コードのフラグメント、および外部の Java コンポーネントへのコールを含めることができます。JavaServer Pages（JSP）テクノロジは、JavaBeans および Enterprise JavaBeans（EJB）で、ビジネス・ロジックおよび動的な機能のフロントエンドとして有効に機能します。

JSP コードは、Web ページで、他の Web スクリプト・コード（JavaScript など）と区別されます。通常の HTML ページに含めることが可能なすべてのコードは、JSP ページにも含めることが可能です。

データベース・アプリケーションの代表的な使用例では、JSP ページによって JavaBeans、Enterprise JavaBeans などのコンポーネントがコールされ、通常、JDBC または SQLJ を介して、Bean がデータベースに直接的または間接的にアクセスします。

JSP ページは、実行前に Java サーブレットに変換され（通常はオンデマンド変換ですが、事前変換の場合もあります）、他のサーブレットと同様に HTTP リクエストを処理し、レスポンスを生成します。JSP テクノロジによって、サーブレットをより簡単にコーディングできるようになります。

また、JSP ページは、サーブレットとの完全な相互運用が可能です。JSP ページは、サーブレットからの出力のインクルード、およびサーブレットへのフォワードを行うことができます。また、サーブレットは、JSP ページからの出力のインクルード、および JSP ページへのフォワードを行うことができます。

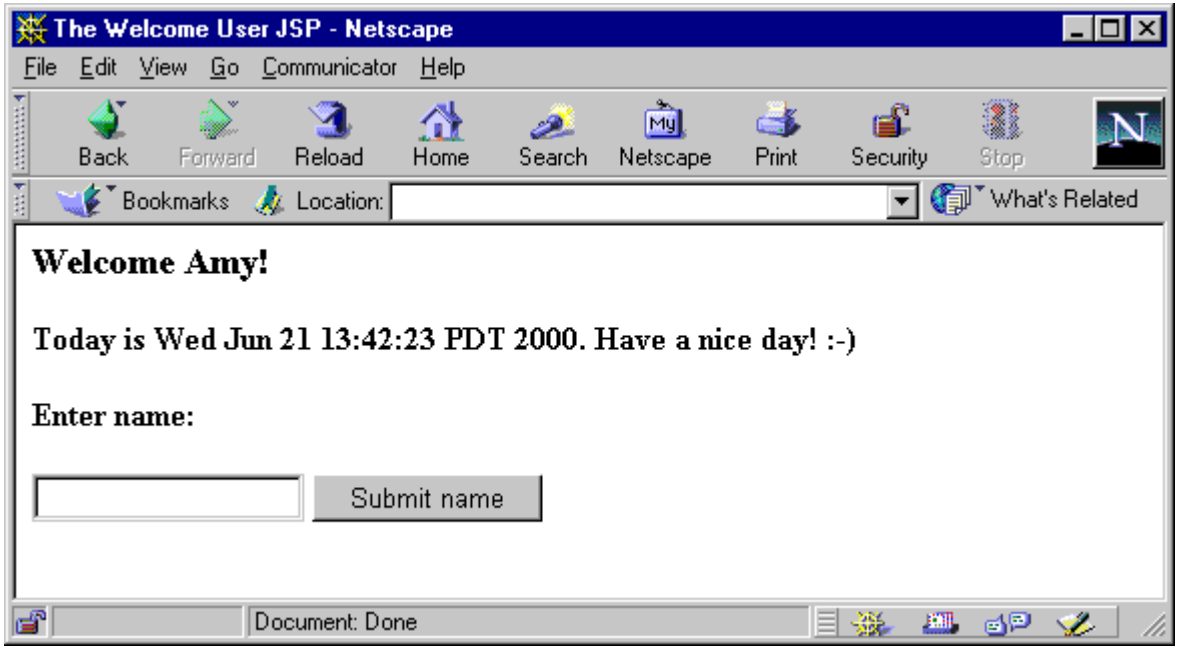
JSP ページの例

次に、単純な JSP ページの例を示します（この例で使用されている JSP 構文要素の説明については、1-9 ページの「JSP 構文要素の概要」を参照）。

```
<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>
```

前述の例に示したとおり、JSP ページでは、Java 要素が <%、%> などのタグで囲まれます。この例では、Java フラグメントによって HTTP request オブジェクトからユーザー名が取得され、出力されて、現在の日付が取得されます。

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 コーディングとサーブレット・コーディングの有効性の比較

Java コードと Java コールを組み合せて HTML ページに挿入する方法は、サーブレットで Java コードをそのまま使用する方法より有効です。JSP 構文を使用すると、動的な Web ページのコーディングが簡単になります。通常、JSP 構文で必要なコードは、Java サーブレット構文を使用する場合より大幅に少なくなります。次に、サーブレット・コードと JSP コードの比較例を示します。

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)
      }
   }
}
```

JSP コード

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

JSP 構文が非常に単純であることに注意してください。また、JSP 構文を使用すると、パッケージのインポート、`try...catch` ブロックなどの Java オーバーヘッドが減少します。

さらに、JSP トランスレータは、出力する `.java` ファイルでのサーブレット・コーディングの大量のオーバーヘッド（標準の `javax.servlet.jsp.HttpJspPage` インタフェースの直接的または間接的な実装、HTTP セッションを取得するためのコードの追加など）を自動的に処理します。

サーブレット・コードの場合とは異なり、JSP ページの HTML は Java の print 文に埋め込まれないため、HTML のオーサリング・ツールを使用して JSP ページの作成もできます。

ビジネス・ロジックのページ表示からの分離 : JavaBeans のコール

JSP テクノロジを使用すると、静的なページ表示を決定する 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 コードを分離して開発できます。そのため、表示およびレイアウトの専門家（HTML には精通しているが Java には精通していない開発者）とコーディングの専門家（Java には精通しているが HTML には精通していない開発者）の間で、メンテナンス時の役割を簡単に分担できるようになります。

通常の JSP ページでは、ほとんどの Java コードおよびビジネス・ロジックは、JSP ページに埋め込まれたフラグメント内ではなく、JSP ページから起動される JavaBeans または Enterprise JavaBeans に含まれます。

JSP テクノロジは、JavaBeans クラスのインスタンスを定義および作成するための次の構文を提供します。

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

この例では、`mybeans.NameBean` クラスの `pageBean` というインスタンスが作成されます（`scope` パラメータについては、この章の後半で説明します）。

ページの後半で、次の例に示すとおり、この Bean インスタンスを使用できます。

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

（たとえば、`pageBean` の `newName` 属性が「Julie」という名前の場合は、「Hello Julie !」と出力されます。この名前はユーザーから入力されます。）

ビジネス・ロジックをページ表示から分離すると、ビジネス・ロジックおよび動的コンテンツを担当する Java の専門家（`NameBean` クラス用のコードを所有およびメンテナンスする開発者）と、アプリケーション・ユーザーに表示される Web ページの静的表示およびレイアウトを担当する HTML の専門家（この JSP ページの `.jsp` ファイル内のコードを所有およびメンテナンスする開発者）の間で簡単に責任を分担できます。

JavaBeans と併用されるタグ（JavaBean インスタンスを宣言するための `useBean`、および Bean のプロパティにアクセスするための `getProperty` と `setProperty`）の詳細は、1-17 ページの「JSP アクションおよび <jsp: > タグ・セット」を参照してください。

JSP ページおよび代替マークアップ言語

通常、JavaServer Pages テクノロジは動的 HTML の出力に使用されますが、Sun 社の JavaServer Pages 仕様バージョン 1.1 は、その他の型の構造化されたテキスト・ベースのドキュメントの出力もサポートします。JSP トランスレータは、JSP 要素の外部に存在するテキストは処理しません。そのため、Web ページに適切なテキストは、通常、JSP ページにも適切です。

JSP ページは、HTTP リクエストから情報を取得し、（データベースへの SQL 問合せなどを介して）データ・サーバーの情報にアクセスします。JSP ページは、この情報を結合および処理し、動的コンテンツとともに HTTP レスポンスに適切に取り込みます。コンテンツは、HTML、DHTML、XHTML、XML などにフォーマットできます。

XML のサポートの詳細は、4-17 ページの「XML の代替構文」を参照してください。また、JML の `transform` タグの詳細は、『Oracle9*i*AS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

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 Servlet コンテナの役割およびエラー処理を含む、JSP の実行方法に関する主要な事項について説明します。

注意： Sun 社の JavaServer Pages 仕様バージョン 1.1 では、以前の仕様で使用されていた JSP エンジンという用語のかわりに、JSP コンテナという用語が使用されています。これらの 2 つの用語は同義です。

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 ページにリクエスト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によって異なりますが、1 つのサーブレットまたは複数のサーブレットの集まりで構成されます。そのため、JSP コンテナは、Servlet コンテナによって実行されます。

Web サーバーが Java で作成されている場合は、JSP コンテナを Web サーバーに組み込むことができます。それ以外の場合は、コンテナを Web サーバーに関連付けて、Web サーバーで使用できます。

JSP ページおよびオンデマンド変換

一般的な方法でオンデマンド変換を行う場合、通常、JSP ページは次のとおり実行されます。

1. ユーザーが、拡張子が .jsp のファイル名で終わる URL を介して、JSP ページをリクエストします。
2. URL 内の .jsp というファイル名拡張子が認識されると、Web サーバーの Servlet コンテナが JSP コンテナを起動します。
3. JSP コンテナは、JSP ページを検索し、リクエストが初めての場合、その JSP ページを変換します。変換では、.java ファイルにサーブレット・コードが生成され、次にその .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って生成されたサーブレット・クラスは、javax.servlet.jsp.HttpJspPage インタフェースを実装する（JSP コンテナによって提供された）クラスのサブクラスとなります。このサーブレット・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 ページのインスタンスと呼びます。

   JSP ページをサーブレットに変換すると、標準のサーブレット・プログラミングのオーバーヘッド（HttpJspPage インタフェースの実装、インタフェースのサービス・メソッドに対するコードの生成など）が、生成されるサーブレット・コードに自動的に取り込まれます。
4. JSP コンテナが、ページ実装クラスのインスタンス化および実行をトリガーします。

次に、サーブレット（JSP ページのインスタンス）が HTTP リクエストを処理し、HTTP レスポンスを生成して、クライアントにそのレスポンスを戻します。

注意： 前述の手順は、便宜上、詳細に説明していません。前述のとおり、JSP コンテナは、その実装方法はベンダーによって異なりますが、1 つのサーブレットまたは複数のサーブレットの集まりで構成されます。たとえば、JSP ページを検索するフロントエンドのサーブレット、変換およびコンパイルを処理する変換サーブレット、および（変換されたページが純粋なサーブレットではなく、Servlet コンテナによって直接実行できないため）各ページ実装クラスによってサブクラス化されたラッパー・サーブレット・クラスで構成される場合があります。これらの各コンポーネントを実行するには、Servlet コンテナが必要です。

JSP ページのリクエスト

JSP ページは、URL を介して直接リクエストするか、または他の Web ページまたはサーブレットを介して間接的にリクエストできます。

JSP ページの直接的なリクエスト

エンド・ユーザーは、サーブレットまたは HTML ページと同様に、URL で直接 JSP ページをリクエストできます。たとえば、次のとおり、Web サーバーの myapp アプリケーション・ルート・ディレクトリに含まれる HelloWorld という JSP ページが存在するとします。

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

Web サーバーのポート 8080 が使用されている場合は、次の URL を使用してこの JSP ページをリクエストできます。

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

(アプリケーション・ルート・ディレクトリは、アプリケーションのサーブレット・コンテキストに指定されています。)

エンド・ユーザーが HelloWorld.jsp を初めてリクエストした場合、JSP コンテナはページの変換と実行の両方をトリガーします。後続のリクエストでは、変換手順は不要のため、JSP コンテナはページの実行のみをトリガーします。

JSP ページの間接的なリクエスト

JSP ページは、サーブレットと同様に、通常の HTML ページからリンクするか、または他の JSP ページまたはサーブレットから参照することによって、間接的に実行することもできます。

他の JSP ページの JSP 文から JSP ページを起動する場合、パスは、アプリケーション・ルートからの相対パス（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ともいう)、または起動先のページからの相対パス（ページ相対パスともいう）に指定できます。アプリケーション相対パスは「/」で始まりますが、ページ相対パスは「/」で始まりません。

通常、これらのパスは、URL または HTML のリンクで使用されるパスとは異なることに注意してください。前述の項の例を使用すると、次に示すとおり、HTML のリンク内のパスは URL の直接リクエスト内のパスと同じです。

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

JSP 文のアプリケーション相対パスは、次のとおりです。

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

同じディレクトリに存在する JSP ページから HelloWorld.jsp を起動するページ相対パスは、次のとおりです。

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

（jsp:include 文および jsp:forward 文の詳細は、1-17 ページの「JSP アクションおよび <jsp: > タグ・セット」を参照してください。）

JSP 構文要素の概要

1-2 ページの「JSP ページの例」では JSP 構文の単純な例を示しましたが、この項では、構文のカテゴリおよび項目のトップレベルのリストを示します。

- ディレクティブ - JSP ページ全体に関する情報を伝達します。
- スクリプト要素 - 宣言、式、スクリプトレット、コメントなどの Java コーディング要素です。
- オブジェクトおよびスコープ - JSP オブジェクトは、明示的または暗黙的に作成できます。また、JSP ページまたはセッションの任意の位置など、指定されたスコープ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、または JSP レスポンスの出力ストリームに影響します（あるいはその両方）。

この項では、基本構文およびいくつかの例を含む、各カテゴリの概要を説明します。詳細は、Sun 社の JavaServer Pages 仕様バージョン 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 と互換性のある代替構文が存在します。4-17 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 コンテナに対して JSP ページ全体に関する指示を行います。この情報は、ページの変換時または実行時に使用されます。次に基本構文を示します。

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

JSP 1.1 仕様は、次のディレクティブをサポートします。

- page - 任意の数のページ依存属性（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、JSP ページの出力バッファのサイズなど）を指定するために使用します。次に例を示します。

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

JSP ページの出力バッファのサイズを 20KB（デフォルトは 8KB）に設定するには、次のディレクティブを使用します。

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

ページのバッファリングを行わないようにするには、次のディレクティブを使用します。

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

注意：

- エラー・ページを使用する JSP ページは、バッファリングする必要があります。（ブラウザに出力するのではなく）エラー・ページにフォワードすると、バッファが消去されます。
- Oracle JSP コンテナの場合、デフォルトの言語は `java` に設定されています。ただし、プログラミングを適切に行うには、言語を明示的に java に設定することをお薦めします。

- `include` - 変換時に JSP ページに挿入されるテキストまたはコードを含むリソースを指定するために使用します。JSP ページの URL 指定に関連したリソースのパスを指定します。

  例：

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

  `include` ディレクティブを使用すると、ページ相対またはコンテキスト相対のいずれかの位置を指定できます（詳細は、1-8 ページの「JSP ページのリクエスト」を参照）。

  注意：

  - `include` ディレクティブ（「静的インクルード」ともいう）の性質は、この章で後述する `jsp:include` アクションに類似していますが、リクエスト時ではなく JSP の変換時に機能します。4-6 ページの「静的インクルードと動的インクルードの比較」を参照してください。
  - `include` ディレクティブは、同じサーブレット・コンテキストのページ間のみで使用できます。

- taglib - JSP ページで使用されるカスタム JSP タグのライブラリを指定するために使用します。ベンダーは、独自のタグ・セットを使用して JSP 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位置、およびそのライブラリのタグを用途別に区別するための接頭辞を示します。

  例 :

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

  oracust 接頭辞は、ページの後半で、ライブラリ内の 1 つのタグを使用する必要がある場合に使用します（このライブラリに dbaseAccess タグが含まれているとします）。

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

  この例では、XML 形式の開始タグおよび終了タグ構文が使用されています。

  JSP タグ・ライブラリおよびタグ・ライブラリ記述ファイルの概要は、1-22 ページの「タグ・ライブラリ」を参照してください。詳細は、第 7 章「JSP のタグ・ライブラリ」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、次のカテゴリの Java コードのフラグメントが含まれます。これらのフラグメントは、JSP ページに表示できます。

- 宣言 - JSP ページで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、<%!...%> 宣言タグで囲まれた標準 Java 構文を使用して、メンバー変数またはメソッドを宣言します。これによって、生成されるサーブレット・コードで対応する宣言が行われます。次に例を示します。

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

  この例では、メンバー変数 f1 が宣言されます。JSP トランスレータによって生成されるサーブレット・クラス・コードで、f1 がクラスの最上位で宣言されます。

  > **注意 :** メンバー変数とは異なり、メソッド変数は、次に示すとおり JSP スクリプトレット内で宣言されます。詳細は、4-11 ページの「メソッド変数宣言とメンバー変数宣言の比較」を参照してください。

- 式 - Java の式です。これらの式は、評価され、必要に応じて文字列値に変換され、ページでの検出時に表示されます。

JSP の式は、セミコロンで終わりません。また、<%=...%> タグで囲まれます。

例 :

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

注意： jsp:setProperty 文などに含まれるリクエスト時属性の JSP の式は、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 - ページのマークアップ言語内に混在する Java コードの一部です。

  スクリプトレット（コードの一部分）は、Java コードの行の一部で構成される場合と、複数の行で構成される場合があります。たとえば、JSP ページの HTML コード内でスクリプトレットを使用して、条件付きブランチまたはループを設定できます。

  JSP スクリプトレットは、通常の Java 構文を使用して、<%...%> スクリプトレット・タグで囲まれます。

  例 1:

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

  1 行のコードで構成される 3 つの JSP スクリプトレットと、2 行の HTML が混在しています。（そのうち 1 つには JSP の式が含まれています。この場合、セミコロンは不要です。）JSP 構文では、HTML コードを、（スクリプトレットに設定された Java のカッコで囲まれる）if および else ブランチ内で条件付き実行されるコードとして使用できます。

  前述の例では、JavaBean 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、より多くの Java コードがスクリプトレットに追加されています。この例では、JavaBean インスタンス pageBean の使用を想定しています。また、事前にインスタンス化されたオブジェクト empmgr に、既知の従業員または不明な従業員に対して適切な機能を実行するためのメソッドが含まれていることを想定しています。

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 JSP スクリプトレットを使用します。次に例を示します。

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

このスクリプトレットによって、メソッド変数 f2 が宣言されます。JSP トランスレータによって生成されるサーブレット・クラス・コードで、f2 がサーブレットのサービス・メソッド内の変数として宣言されます。

メンバー変数は、前述のとおり JSP 宣言で宣言されます。

これらの比較については、4-11 ページの「メソッド変数宣言とメンバー変数宣言の比較」を参照してください。

- コメント - JSP コード内に埋め込まれた、開発者によるコメントです。任意の Java コードに埋め込まれたコメントに類似しています。

  コメントは、<%--...--%> タグで囲まれます。

  例：

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

JSP オブジェクトおよびスコープ

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ているか、または JSP ページにアクセス可能な Java クラスのインスタンスを示します。JSP オブジェクトは、次のいずれかになります。

- 明示的なオブジェクト - JSP ページのコード内で宣言および作成され、選択する scope の設定に従って、その JSP ページまたはその他のページにアクセスできます。

または

- 暗黙的なオブジェクト - 基礎となる JSP メカニズムによって作成され、特定のオブジェクト型の固有の scope 設定に従って、JSP ページの Java スクリプトレットまたは式にアクセスできます。

スコープの詳細は、1-14 ページの「オブジェクトのスコープ」を参照してください。

明示的なオブジェクト

通常、明示的なオブジェクトは、jsp:useBean アクション文で宣言および作成された JavaBean インスタンスです。次に例を示します。jsp:useBean 文およびその他のアクション文については、1-17 ページの「JSP アクションおよび <jsp: > タグ・セット」を参照してください。

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

この文では、mybeans パッケージの NameBean クラスのインスタンス pageBean が定義されます。scope パラメータについては、1-14 ページの「オブジェクトのスコープ」を参照してください。

任意の Java プログラムで Java クラスのインスタンスを作成する場合と同様に、Java スクリプトレットまたは宣言内にオブジェクトを作成することもできます。

オブジェクトのスコープ

JSP ページのオブジェクトは、明示的か暗黙的かにかかわらず、特定のスコープ内でアクセスできます。jsp:useBean アクション文で作成された JavaBean インスタンスなどの明示的なオブジェクトの場合は、(1-14 ページの「明示的なオブジェクト」の例に示すとおり) 次の構文を使用してスコープを明示的に設定できます。

```
scope="scopevalue"
```

次の 4 つのスコープに設定できます。

- scope="page" - オブジェクトは、そのオブジェクトが作成された JSP ページ内のみからアクセス可能です。

  JSP ページの実行中にユーザーがそのページをリフレッシュすると、page スコープが設定されたすべてのオブジェクトに対して新しいインスタンスが作成されることに注意してください。

- scope="request" - オブジェクトは、そのオブジェクトを作成した JSP ページによって処理される HTTP リクエストと同じリクエストを処理する、すべての JSP ページからアクセス可能です。

- scope="session" - オブジェクトは、そのオブジェクトを作成した JSP ページと同じ HTTP セッションを共有する、すべての JSP ページからアクセス可能です。

- scope="application" - オブジェクトは、そのオブジェクトを作成した JSP ページと同じ (1 つの Java Virtual Machine 内の) Web アプリケーションで使用される、すべての JSP ページからアクセス可能です。

暗黙的なオブジェクト

JSP テクノロジによって、すべての JSP ページで一連の暗黙的なオブジェクトを使用できるようになります。これらは JSP メカニズムによって自動的に作成される Java クラスのインスタンスで、基礎となるサーブレット環境との対話を可能にします。

次の暗黙的なオブジェクトが使用可能です。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 URL にある、代表的なクラスおよびインタフェースに関する Sun 社の Javadoc を参照してください。

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

- `page`

  ページの変換時に作成される JSP ページ実装クラスのインスタンスです。`javax.servlet.jsp.HttpJspPage` インタフェースを実装します。`page` は、JSP ページ内の `this` と同義です。

- `request`

  HTTP リクエストを表します。`javax.servlet.ServletRequest` インタフェースの拡張である、`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`

  HTTP レスポンスを表します。`javax.servlet.ServletResponse` インタフェースの拡張である、`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。

  特定のリクエストに対する `response` と `request` オブジェクトは、相互に関連付けられます。

- `pageContext`

  JSP ページのページ・コンテキストを表します。JSP ページ・インスタンスのすべての `page` スコープ・オブジェクトを格納するため、また、これらのオブジェクトにアクセスするために提供されます。`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` オブジェクトには `page` スコープが設定されています。これによって、関連付けられた JSP ページ・インスタンスのみへのアクセスが可能になります。

- `session`

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

- `application`

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

  `application` オブジェクトは、1 つの JVM 内に存在するアプリケーションのインスタンスの一部として実行されているすべての JSP ページの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のアーキテクチャに注意する必要があります。）

- `out`

  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。`java.io.Writer` クラスの拡張である、`javax.servlet.jsp.JspWriter` クラスのインスタンスです。

  `out` オブジェクトは、特定のリクエスト用の `response` オブジェクトに関連付けられます。

- `config`

  JSP ページのサーブレット構成を表します。`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、Servlet 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。この情報の一部が、適切な `ServletContext` インスタンスになります。

- `exception`（JSP エラー・ページのみ）

  この暗黙的なオブジェクトは、JSP エラー・ページにのみ適用されます。他の JSP ページで例外が発生すると、JSP エラー・ページに処理が転送されます。JSP エラー・ページでは、`page` ディレクティブの `isErrorPage` 属性が `true` に設定されている必要があります。

  暗黙的な `exception` オブジェクトは `java.lang.Exception` のインスタンスで、他の JSP ページで発生し、現行のエラー・ページが実行される原因となったキャッチされなかった例外を表します。

  `exception` オブジェクトは、例外の発生時に処理が転送された JSP エラー・ページのインスタンスからのみアクセス可能です。

  JSP のエラー処理および `exception` オブジェクトの使用の例は、3-16 ページの「JSP のランタイム・エラーの処理」を参照してください。

暗黙的なオブジェクトの使用

前述の項で説明したすべての暗黙的なオブジェクトは、有用な場合があります。次の例では、`request` オブジェクトを使用して、HTTP リクエストから `username` パラメータの値を取得し、表示します。

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

JSP アクションおよび <jsp: > タグ・セット

JSP アクション要素を使用すると、JSP ページの実行中に、Java オブジェクトのインスタンス化、ページでの Java オブジェクトの有効化などのアクションが発生します。次のアクションも指定できます。

- JavaBean インスタンスの作成および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

アクション要素では、「<jsp:」構文で始まる、一連の標準 JSP タグが使用されます。JSP ページのコーディングを行うには、この章で前述した「<%」構文で始まるタグで十分ですが、「<jsp:」タグを使用すると、機能性および利便性が向上します。

アクション要素では、XML 文の構文に類似した構文も使用されます。この構文では、次の例に示すような開始タグおよび終了タグが使用されます。

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

本体が存在しない場合、アクション文は空タグで終わります。

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

JSP 仕様には、次の標準アクション・タグが含まれます。この項では、これらのタグの概要を説明します。

- `jsp:useBean`

  指定された JavaBean クラスのインスタンスを作成し、そのインスタンスに名前を指定し、アクセス可能なスコープ（現行の JSP ページ・インスタンス内のすべての場所など）を定義します。

  例 :

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

  この例では、`mybeans.NameBean` クラスの page スコープが設定された `pageBean` インスタンスを作成します。このインスタンスには、このインスタンスが作成された JSP ページ・インスタンスからのみアクセス可能です。

- `jsp:setProperty`

  1 つ以上の Bean のプロパティを設定します。Bean は、`useBean` アクションで指定済である必要があります。指定されたプロパティの値を直接指定するか、関連付けられた HTTP リクエストのパラメータから指定されたプロパティの値を取得するか、または HTTP リクエストのパラメータの一連のプロパティおよび値を繰り返し設定できます。

次の例では、(前述の useBean の例で定義された) pageBean インスタンスの user プロパティの値を「Smith」に設定します。

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

次の例では、HTTP リクエストで username というパラメータに設定されている値に従って、pageBean インスタンスの user プロパティを設定します。

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

Bean のプロパティとリクエストのパラメータの名前が同じ (user) 場合、次のとおりプロパティを設定できます。

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

次の例では、HTTP リクエストのパラメータを繰り返し設定します。Bean のプロパティの名前とリクエストのパラメータの名前を照合し、対応するリクエストのパラメータの値に従って Bean のプロパティを設定します。

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

重要: property="*" である場合、JSP 1.1 仕様では、プロパティの設定順序が保持されません。順序が重要な場合、および JSP ページの移植性の確認が必要な場合は、各プロパティに個別の jsp:setProperty 文を使用する必要があります。

個別の jsp:setProperty 文を使用する場合、Oracle JSP トランスレータは、対応する setXXX() メソッドを直接生成できます。この場合、イントロスペクションは変換時にのみ発生します。実行時に、コストが高い Bean のイントロスペクションを行う必要はありません。

- jsp:getProperty

  Bean のプロパティ値を読み込み、その値を Java 文字列に変換し、その文字列値を出力として表示できるように暗黙的な out オブジェクト内に配置します。Bean は、jsp:useBean アクションで指定済の必要があります。文字列の変換では、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 クラスで指定された 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 オブジェクトに挿入します。

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

- jsp:param

  jsp:include、jsp:forward または jsp:plugin アクション（後続の説明を参照）と併用できます。

  jsp:forward および jsp:include 文と併用する場合は、オプションで、HTTP リクエストのオブジェクトのパラメータ値にキーと値の組を提供します。このアクションで指定された新しいパラメータおよび値は、request オブジェクトに追加されます。この場合、古い値より新しい値が優先されます。

  次の例では、request オブジェクトのパラメータ username を Smith という値に設定します。

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

  注意： JSP 1.0 仕様では、jsp:include または jsp:forward に対して jsp:param タグがサポートされていません。

- jsp:include

  ページの表示のリクエスト時に、追加の静的または動的なリソースをページに挿入します。相対 URL（ページ相対またはアプリケーション相対）を使用してリソースを指定します。

  Sun 社の JavaServer Pages 仕様バージョン 1.1 では、flush を true に設定する必要があります。これによって、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ます。（flush 属性は必須ですが、false の設定は現在無効です。）

  2 番目の例に示すとおり、アクション本体に jsp:param を設定することもできます。

  例：

```
<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>
```

  前述の構文のかわりに、次の構文も使用できます。

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

注意：

- jsp:include アクション（「動的インクルード」ともいう）の性質は、この章で前述した include ディレクティブに類似していますが、変換時ではなくリクエスト時に機能します。4-6 ページの「静的インクルードと動的インクルードの比較」を参照してください。
- jsp:include アクションは、同じサーブレット・コンテキストのページ間のみで使用できます。

- jsp:forward

  現行のページの実行を効果的に終了し、出力を破棄し、新しいページ（HTML ページ、JSP ページまたはサーブレットのいずれか）にディスパッチします。

  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。このアクションは、コンテンツをブラウザに出力するのではなく、バッファを消去します。

  jsp:include と同様に、2 番目の例に示すとおり、アクション本体に jsp:param を設定することもできます。

  例：

```
<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：

- jsp:forward の例と前述の jsp:include の例の相違点は、jsp:include の例では現行のページの出力に userinfopage.jsp が挿入されるのに対して、jsp:forward の例では現行のページの実行が停止され、かわりに userinfopage.jsp が表示されることです。
- jsp:forward アクションは、同じサーブレット・コンテキストのページ間のみで使用できます。
- jsp:forward アクションによって、元の request オブジェクトがターゲット・ページにフォワードされます。request オブジェクトをフォワードする必要がない場合は、かわりに、標準の javax.servlet.http.HttpServletResponse インタフェースで指定された sendRedirect(String) メソッドを使用できます。これによって、指定されたリダイレクト位置の URL を使用して、一時的なリダイレクト・レスポンスがクライアントに送信されます。相対 URL を指定できます。その相対 URL は、Servlet コンテナによって絶対 URL に変換されます。

- jsp:plugin

  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た後に、クライアントのブラウザで指定されたアプレットまたは JavaBeans を実行します。

  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 コンテナがダウンロード用のデフォルトの URL を提供する場合がありますが、nspluginurl="*url*" 属性（Netscape ブラウザの場合）または iepluginurl="*url*" 属性（Internet Explorer ブラウザの場合）を指定することもできます。

  アプレットまたは JavaBeans にパラメータを指定するには、<jsp:params> 開始タグと </jsp:params> 終了タグの間にネストした jsp:param アクションを使用します。（jsp:include または jsp:forward アクションで jsp:param を使用する場合、これらの jsp:params 開始タグおよび終了タグは不要です。）

  プラグイン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、<jsp:fallback> 開始タグおよび </jsp:fallback> 終了タグを使用します。

Sun 社の JavaServer Pages 仕様バージョン 1.1 に記載されている次の例は、アプレットのプラグインの使用方法を示します。

```
<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>
```

jsp:plugin アクション文では、ARCHIVE、HEIGHT、NAME、TITLE、WIDTH など、他の多くのパラメータも使用できます。これらのパラメータは通常の HTML 仕様に従って使用します。

タグ・ライブラリ

JSP 1.1 仕様では、この項で前述した標準 JSP タグの他に、ベンダーが独自のタグ・ライブラリを定義できます。また、ベンダーがフレームワークを実装し、顧客が独自のタグ・ライブラリを定義できるようにすることも可能です。

タグ・ライブラリは、カスタム・タグのコレクションを定義します。タグ・ライブラリは、JSP のサブ言語とみなすことができます。タグ・ライブラリは、開発者が JSP ページの手動コーディングに直接使用できますが、Java 開発ツールによって自動的に使用される場合もあります。標準タグ・ライブラリは、JSP コンテナの異なる実装間で移植可能である必要があります。

taglib ディレクティブを使用した JSP ページへのタグ・ライブラリのインポートの概要は、1-9 ページの「ディレクティブ」を参照してください。

標準 JavaServer Pages による JSP タグ・ライブラリのサポートの基本概念には、次の項目が含まれます。

- タグ・ハンドラ

  タグ・ハンドラは、カスタム・タグの使用によるアクションのセマンティクスを記述します。タグ・ハンドラは、標準 javax.servlet.jsp.tagext パッケージの Tag または BodyTag インタフェースのいずれか（タグが開始タグと終了タグの間の本体を使用するかどうかによって異なる）を実装する Java クラスのインスタンスです。

- スクリプト変数

  カスタム・タグのアクションでは、タグ自体、またはスクリプトレットなどの他のスクリプト要素で使用可能なサーバー側のオブジェクトを作成できます。これを実行するには、スクリプト変数を作成または更新します。

  カスタム・タグによって定義され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に指定されている必要があります。このマニュアルでは、このようなサブクラスをタグ追加情報クラスと呼びます。JSP コンテナは、変換時にこれらのクラスのインスタンスを使用します。

- 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・ライブラリおよびライブラリの個別のタグに関する情報が含まれている XML 文書です。TLD のファイル名には、`.tld` という拡張子が付きます。

  JSP コンテナは、TLD ファイルを使用して、ライブラリのタグを検出した場合に実行するアクションを決定します。

- タグ・ライブラリに対する `web.xml` の使用

  Sun 社の Java Servlet 仕様バージョン 2.2 には、サーブレットの標準デプロイメント・ディスクリプタである `web.xml` ファイルについて記載されています。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位置の指定できます。

  JSP タグ・ライブラリの場合、`web.xml` ファイルに `taglib` 要素、および `taglib-uri` と `taglib-location` という 2 つのサブ要素を含めることができます。

これらの項目の詳細は、7-2 ページの「標準のタグ・ライブラリ・フレームワーク」を参照してください。詳細は、Sun 社の JavaServer Pages 仕様バージョン 1.1 を参照してください。

Oracle によって提供されるタグ・ライブラリの詳細は、『Oracle9*i*AS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

2

Oracle JSP 実装の概要

Oracle JSP リリース 1.1.x.x は、Sun 社の JavaServer Pages 仕様バージョン 1.1 の完全実装です。

この章では、プログラム的な拡張機能を含む Oracle JSP 実装の概要、および Oracle と Oracle 以外の両方の環境でのサポートについて説明します。

この章の内容は次のとおりです。

重要： Oracle9*i* リリース 2（9.2）には、バージョン 1.1.2 の Oracle JSP コンテナが付属しています。

Oracle9*i* に付属の JSP コンテナ、Servlet コンテナおよび Web サーバーの概要

この項では、Oracle9*i* に付属の Oracle JSP コンテナおよびサーブレット環境の概要を説明します。また、データベースにアクセスする Web アプリケーション実行時の Oracle HTTP Server および `mod_jserv` コンポーネントの役割についても説明します。

Oracle9*i* データベースでの J2EE の非サポートに関する注意事項： 軽量で簡単に使用可能な、高速かつ認定済の新しい J2EE コンテナである Oracle9*i* Application Server Containers for J2EE（OC4J）の導入に併せて、Oracle9*i* データベース リリース 2（9.2）以上では、データベースでの Java 2 Enterprise Edition（J2EE）および CORBA スタックがサポートされません。ただし、データベース埋込み型の JVM（Oracle JVM）はサポートされ、継続して拡張されて、データベースに Java 2 Standard Edition（J2SE）の機能（Java ストアド・プロシージャ、JDBC および SQLJ）を提供します。Oracle9*i* データベース リリース 2（9.2）以上では、データベースで次のテクノロジがサポートされません。

- 次のコンポーネントで構成される J2EE スタック
  Enterprise Beans（EJB）コンテナ
  JavaServer Pages（JSP）コンテナ
  Oracle9*i* Servlet Engine（OSE）
- Visibroker for Java に基づく埋込み型の Common Object Request Broker Architecture（CORBA）フレームワーク

現在、Oracle データベースにサーブレット、JSP ページ、EJB および CORBA オブジェクトは配置できません。Oracle9*i* データベース リリース 1（9.0.1）が、J2EE および CORBA スタックをサポートする最後のデータベースです。データベースで実行している既存の J2EE アプリケーションは、OC4J に移行することをお薦めします。

Oracle9*i* に付属の JSP コンテナおよびサーブレット環境

Oracle9*i* リリース 2（9.2）には、バージョン 1.1.2 の Oracle JSP コンテナが付属しています。このバージョンは、Sun 社の JavaServer Pages 仕様バージョン 1.1 に完全に準拠します。

Oracle9*i* リリース 2（9.2）に付属のサーブレット環境は、Servlet 2.0 環境の Apache JServ です。JSP 1.1 仕様の要件では Servlet 2.1（b）以上の環境が必要ですが、Oracle JSP コンテナのバージョン 1.1.2 は、Servlet 2.0 以上のすべての環境で使用できます。JSP コンテナには、Servlet 2.2 の一部の機能をエミュレートする拡張機能が含まれています。詳細は、2-5 ページの「サーブレット環境間での移植性および機能性」を参照してください。

Servlet 2.0 環境で Oracle JSP コンテナを使用する場合の特別な考慮点については、第 9 章「Apache JServ での Oracle JSP」を参照してください。この章では、JSP および JServ 構成についても説明します。

他のサーブレット環境

JServ 以外に使用可能なサーブレット環境があります。次の選択肢があります。

- Oracle9*i*AS Containers for J2EE（OC4J）

  OC4J は、Oracle9*i* Application Server に付属しています。また、独自のバージョンは Oracle Technology Network Japan（`http://otn.oracle.co.jp`）から入手することもできます。

- Tomcat

  Apache Software Foundation の Tomcat には、Servlet 2.2 環境が含まれています。JSP 1.1 の参照実装も含まれていますが、それに加えて Oracle JSP コンテナを使用できます。

Oracle9*i* リリース 2（9.2）に付属の環境は Servlet 2.0 ですが、このマニュアルでは、Servlet 2.2 の多くの機能についても説明します。これは、Servlet 2.2 環境が Oracle JSP コンテナで使用可能で、また JSP コンテナ自体が Servlet 2.2 の一部の機能をエミュレートするためです。

OC4J のドキュメントは、Oracle9*i* Application Server リリースおよび Oracle Technology Network Japan から入手できます。

Oracle HTTP Server の役割

Apache Web サーバーを備えた Oracle HTTP Server は、データベースにアクセスする Web アプリケーションへの HTTP エントリ・ポイントとして、Oracle9*i* データベースに付属しています。データベース・アクセスは、Apache のアドオン・モジュールを介して行われます。

この項の内容は次のとおりです。

- Apache mod の使用
- mod_jserv の詳細

Apache mod の使用

Oracle HTTP Server を使用する場合、Apache、またはオラクル社などの他のベンダーのいずれかによって提供される様々な Apache mod コンポーネントを介して、動的コンテンツが配信されます（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）。通常、Apache mod は Apache のアドレス空間で実行する C コードのモジュールで、特定の mod 固有のプロセッサにリクエストを渡します。mod ソフトウェアは、特定のプロセッサとの使用のために作成されます。

Oracle9*i* に付属の JServ サーブレット環境で実行している JSP ページまたはサーブレットから Oracle9*i* のデータにアクセスするには、`mod_jserv` を使用します。この mod は Apache によって開発されたもので、Oracle9*i* に付属しています。

注意： Apache 環境では、他の多くの Apache mod コンポーネントを使用可能です。これらは、Apache が一般的な用途のために提供するか、またはオラクル社が Oracle 固有の用途のために提供しています。ただし、これらのコンポーネントは、JSP アプリケーションには関係ありません。

---

mod_jserv の詳細

`mod_jserv` コンポーネントは、中間層の JVM に存在する JServ サーブレット・コンテナで実行される JSP ページまたはサーブレットに、HTTP リクエストを委譲します。Oracle9*i* リリース 2（9.2）では、Servlet 2.0 仕様をサポートする JServ Servlet コンテナが提供されています。中間層環境は、バックエンドの Oracle9*i* データベースと物理的に同じホスト上に存在する必要はありません。

`mod_jserv` と中間層の JVM 間の通信では、TCP/IP ではなく独自の 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・バランシングの目的で、プール内の複数の JVM にリクエストを委譲できます。

中間層の JVM で実行している JSP アプリケーションでは、データベースにアクセスするために Oracle JDBC OCI ドライバまたは Thin ドライバが使用されます。

（Servlet 2.1 または 2.2 環境とは異なり）Servlet 2.0 環境では、特別な配慮が必要です。9-18 ページの「JServ サーブレット環境についての考慮点」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、Oracle9*i* に付属しています）。

サーブレット環境間での移植性および機能性

Oracle JavaServer Pages 実装には、サーバー・プラットフォーム間およびサーブレット環境間での高い移植性があります。また、サーブレット・コンテキストの動作の定義が不十分な古いサーブレット環境に存在する Web アプリケーション用のフレームワークも提供します。

Oracle JSP の移植性

Oracle JSP コンテナは、Sun 社の Java Servlet 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。これが、Servlet 2.1（b）以上の実装を必要とするほとんどの JSP 実装との相違点です。Oracle JSP コンテナは、古いサーブレット環境に欠落している機能と同等の機能を提供します。

また、Oracle JSP コンテナは、サーバー環境およびそのサーブレット実装に依存しません。これが、独自の製品としてではなく、サーブレット実装の一部として JSP 実装を提供するベンダーとの相違点です。

この移植性によって、サーバーまたはサーブレット・プラットフォームによる制限のために開発システムで異なる JSP 実装を使用する必要がなくなり、開発環境およびターゲット環境の両方でより簡単に JSP ページを実行できるようになります。通常、システム上でターゲット・サーバーと同じ JSP コンテナを使用して開発を行えるというメリットがありますが、環境によって多少の違いはあります。

Servlet 2.0 環境に対する Oracle JSP の拡張機能

サーブレット仕様と JSP 機能間に相互依存性があるため、Sun 社では、JavaServer Pages 仕様のバージョンを Java Servlet 仕様の特定のバージョンに関連付けています。Sun 社によると、JSP 1.0 には Servlet 2.1（b）実装が必要で、JSP 1.1 には Servlet 2.2 実装が必要です。

Servlet 2.0 仕様では、各アプリケーションにサーブレット・コンテキストが提供されるのではなく、1 つの Java Virtual Machine に対して 1 つのみのサーブレット・コンテキストが提供されるという制限がありました。Servlet 2.1 仕様では各アプリケーションに対する個別のサーブレット・コンテキストの提供が可能になりましたが、必須ではありませんでした。Servlet 2.1（b）および Servlet 2.2 仕様では、個別のサーブレット・コンテキストの提供が必須になりました。

これに対し、Oracle JSP コンテナは、Servlet 2.1（b）仕様によるアプリケーション・サポートをエミュレートする機能を提供します。この機能によって、JServ などの Servlet 2.0 環境で完全なアプリケーション・フレームワークを使用できるようになります。アプリケーションに対して個別の `ServletContext` および `HttpSession` オブジェクトも提供できます。

この拡張サポートは、globals.jsa ファイルを介して提供されます。このファイルは、JSP アプリケーション・マーカー、アプリケーションとセッションのイベント・ハンドラ、およびアプリケーションのグローバル宣言とグローバル・ディレクティブを集中管理する場所として機能します（詳細は、9-23 ページの「Oracle JSP アプリケーションおよび JServ でのセッション・サポート」を参照）。

この拡張機能のため、Oracle JSP コンテナは、基礎となるサーブレット環境による制限を受けません。

Oracle9*i* JDeveloper による Oracle JSP コンテナのサポート

現在、一部のビジュアル 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ています。特に、Oracle9*i* JDeveloper では Oracle JSP コンテナがサポートされ、次の機能が含まれています。

- アプリケーション開発のすべてのサイクル（JSP ページの編集、デバッグ、実行）をサポートするための Oracle JSP コンテナの統合
- 配置された JSP ページのデバッグ
- データ対応および Web 対応の JavaBeans の拡張セット（JDeveloper Web Beans と呼ばれる）
- JSP エレメント・ウィザード（事前定義された Web Beans をページに追加するための便利な方法を提供）
- 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置オプション

JSP の配置サポートの詳細は、6-28 ページの「Oracle9i JDeveloper を使用した JSP ページの配置」を参照してください。

デバッグに関しては、JDeveloper で、JSP ページのソース内にブレーク・ポイントを設定し、JSP ページからのコールを JavaBeans に渡すことができます。これは、JSP ページ内に print 文を追加して、（ブラウザに表示するために）状態をレスポンス・ストリームに出力する方法、（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・ログに出力する方法などの、手動でのデバッグ方法より効率的です。

JDeveloper の詳細は、オンライン・ヘルプまたは次の Web サイトを参照してください。

http://otn.oracle.co.jp/products/jdev/index.html

Oracle 以外の環境での Oracle JSP コンテナのサポート

Servlet 仕様 2.0 以上をサポートするすべてのサーバー環境で、Oracle JSP コンテナをインストールおよび実行できます。特に、次の環境では、リリース 1.1.2 以下の Oracle JSP コンテナが動作することはテスト済です。

- Apache Software Foundation の Apache JServ 1.1（Oracle9*i* に付属）

  JSP 環境を持たない、Web サーバーおよび Servlet 2.0 の環境です。JSP ページを実行するには、追加で JSP 環境をインストールする必要があります。

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

  Servlet 2.1 および JavaServer Pages 1.0 の参照実装が含まれる Web サーバーです。ただし、JSWDK サーブレット環境に加えて Oracle JSP コンテナをインストールし、元の JSP 環境に置き換えることもできます。

- Apache Software Foundation の Tomcat 3.1

  Sun 社と Apache Software Foundation が共同開発した、Servlet 2.2 および JavaServer Pages 1.1 の参照実装が含まれた Web サーバーです。ただし、Tomcat サーブレット環境に加えて Oracle JSP コンテナをインストールし、元の JSP 環境に置き換えることもできます。Tomcat Web サーバーを使用するかわりに、Tomcat を Apache Web サーバーと併用して実行することもできます。

Oracle JSP のプログラム的な拡張機能の概要

この項では、Oracle JSP コンテナによってサポートされる次の Oracle 固有のプログラム的な拡張機能の概要を説明します。

- 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。
- 拡張グローバリゼーション・サポート。
- イベント処理のための `JspScopeListener`。
- Servlet 2.0 環境でのアプリケーション・サポートのための `globals.jsa` ファイル。

Oracle JSP コンテナは、通常は他の JSP 環境に移植可能なカスタム・タグ・ライブラリおよびカスタム JavaBeans を介して、次の拡張機能も提供します。これらの拡張機能については、『Oracle9*i*AS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

- 指定されたスコープを持つ JavaBeans として実装された拡張型
- XML および XSL との統合
- データ・アクセス JavaBeans
- Oracle JSP Markup Language（JML）カスタム・タグ・ライブラリ（JSP の開発が簡略化されます。）
- SQL 機能用のカスタム・タグ・ライブラリ

次の項では、これらすべての機能の概要を示します。

Oracle 固有の拡張機能の概要

この項に示す Oracle JSP の拡張機能は、他の JSP 環境には移植できません。

Oracle JSP コンテナでの SQLJ のサポート

一般的に、動的サーバー・ページにはデータベースから抽出されたデータが含まれますが、JavaServer Pages テクノロジは、データベース・アクセスを簡単にするための組込みサポートを提供しません。通常、JSP 開発者は、標準の Java Database Connectivity（JDBC）API またはデータベース JavaBeans のカスタム・セットを使用する必要があります。

SQLJ は、静的 SQL 命令を Java コードに直接埋め込むための標準構文です。これによって、データベース・アクセスのプログラミングが大幅に簡略化されます。Oracle JSP コンテナおよびそのトランスレータは、JSP スクリプトレットでの SQLJ プログラミングをサポートします。

SQLJ 文は、`#sql` トークンによって示されます。JSP ソース・コード・ファイルに `.sqljsp` というファイル名の拡張子を使用することによって、Oracle JSP 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、5-3 ページの「Oracle JSP による Oracle SQLJ のサポート」を参照してください。

Oracle JSP コンテナでの拡張グローバリゼーション・サポート

Oracle9*i* リリース 2（9.2）は、マルチバイトのリクエスト・パラメータおよび Bean プロパティ設定をエンコードできないサーブレット環境に対して、拡張グローバリゼーション・サポートを提供します。

このような環境では、Oracle JSP コンテナが `translate_params` 構成パラメータをサポートします。このパラメータを有効にすると、Servlet コンテナに優先して JSP 自体がエンコーディングを行うように、JSP コンテナに指示できます。

詳細は、8-5 ページの「Oracle JSP によるマルチバイト・パラメータ・エンコーディングの拡張サポート」を参照してください。

イベント処理のための JspScopeListener

Oracle9*i* リリース 2（9.2）は、JSP アプリケーションに含まれる様々なスコープの Java オブジェクトのライフ・サイクル管理用に、`JspScopeListener` インタフェースを提供します。

標準サーブレットおよび JSP のイベントの処理は、`javax.servlet.http.HttpSessionBindingListener` インタフェースを介して行われます。ただし、このインタフェースによって処理されるのは、セッション・ベースのイベントのみです。Oracle の `JspScopeListener` は、ページ・ベース、リクエスト・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、5-2 ページの「JspScopeListener を使用した Oracle JSP によるイベント処理」を参照してください。

アプリケーション・サポートのための globals.jsa ファイル（Servlet 2.0）

サーブレット・コンテキストの定義が完全ではない Servlet 2.0 環境では、Oracle JSP コンテナが `globals.jsa` というファイルを定義して、サーブレットのアプリケーション・サポートを拡張します。

1 つの Java Virtual Machine 内で、アプリケーションごとに（サーブレット・コンテキストごとに）`globals.jsa` ファイルを定義できます。このファイルは、アプリケーションのロケーション・マーカーとして使用されることによって、Web アプリケーションの概念をサポートします。`globals.jsa` の機能に基づいて、Oracle JSP コンテナは、サーブレット・コンテキストおよび HTTP セッションの動作が十分に定義されていないサーブレット環境に対して、その動作を擬似実行することもできます。

`globals.jsa` ファイルによって、グローバル Java 宣言および JSP ディレクティブは、アプリケーションのすべての JSP ページ間で送信されます。

詳細は、9-23 ページの「Oracle JSP アプリケーションおよび JServ でのセッション・サポート」を参照してください。

Oracle9*i* に付属の JSP タグ・ライブラリおよび JavaBeans の概要

この項で説明する Oracle の拡張機能は、標準タグ・ライブラリまたはカスタム JavaBeans を介して実装され、通常、他の JSP 環境に移植可能です。

これらの機能については、『Oracle9*i*AS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

拡張型の JavaBeans

通常、JSP ページでは、スカラー値を表すために基本 Java 型が使用されます。ただし、次の型のカテゴリは、いずれも JSP ページでの使用には適していません。

- プリミティブ型（int、float、double など）

  これらの型の値には、スコープを指定できません。スコープ・オブジェクトに格納できるのはオブジェクトのみであるため、これらの型の値は、（page、request、session または application スコープの）JSP スコープ・オブジェクトには格納できません。

- 標準 java.lang パッケージのラッパー・クラス（Integer、Float、Double など）

  これらの型の値はオブジェクトであるため、理論上は、JSP スコープ・オブジェクトに格納できます。ただし、ラッパー・クラスは JavaBean モデルに準拠せず、引数なしのコンストラクタを提供しないため、これらの型の値を jsp:useBean アクション内で宣言することはできません。

  また、ラッパー・クラスのインスタンスは変更できません。値を変更するには、新しいインスタンスを作成して、適切な値を割り当てる必要があります。

これらの制限に対処するために、Oracle9*i* リリース 2（9.2）は、oracle.jsp.jml パッケージの JavaBean クラス JmlBoolean、JmlNumber、JmlFPNumber および JmlString を提供して、ほとんどの一般的な Java 型をラップします。

XML および XSL との統合

JSP 構文を使用すると、HTML コードのみでなく、任意のテキスト・ベースの 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 文書を生成する場合は、XML データがクライアントに送信される前に、その XML データにスタイルシートを適用する必要がある場合があります。JavaServer Pages テクノロジでは、JSP ページに使用される標準出力ストリームがサーバーを介して直接書き込まれるため、スタイルシートの適用は困難です。

Oracle9*i* リリース 2（9.2）には、特別なタグがサンプル JML タグ・ライブラリに含まれています。そのタグを使用して、すべてまたは一部の JSP ページを、出力前に XSL スタイルシートを介して変換する必要があることを指定します。ページの個別の部分に対して異なるスタイルシートを指定する必要がある場合は、単一の JSP ページ内でこの JML タグを複数回使用できます。

また、Oracle JSP トランスレータは、Sun 社の JavaServer Pages 仕様バージョン 1.1 で指定されている XML の代替構文をサポートします。詳細は、4-17 ページの「XML の代替構文」を参照してください。

カスタム・データ・アクセス JavaBeans

Oracle9*i* リリース 2（9.2）は、データベース・アクセスに使用するための一連のカスタム JavaBeans を提供します。次の Bean は、`oracle.jsp.dbutil` パッケージに含まれます。

- `ConnBean` - 単純なデータベース接続をオープンします。
- `ConnCacheBean` - データベース接続に Oracle の接続キャッシュ実装を使用します。
- `DBBean` - データベース問合せを実行します。
- `CursorBean` - 問合せおよび `UPDATE`、`INSERT` および `DELETE` 文に対する一般的な DML サポートを提供します。

SQL カスタム・タグ・ライブラリ

Oracle9*i* リリース 2（9.2）は、SQL 機能のカスタム・タグ・ライブラリを提供します。次のタグが提供されます。

- `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` - 問合せのカーソルをクローズします。
- `dbNextRow` - 結果セットの次の行に移動します。
- `dbExecute` - すべての SQL DML または DDL 文を実行します。

Oracle JSP Markup Language（JML）カスタム・タグ・ライブラリ

Sun 社の JavaServer Pages 仕様バージョン 1.1 では Java 以外のスクリプト言語もサポートされていますが、使用される主な言語は Java です。JavaServer Pages テクノロジは、静的な HTML 開発と動的な Java 開発を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 の専門家がいない少人数の開発グループでは）開発は困難になります。

Oracle9*i* リリース 2（9.2）は、かわりにカスタム・タグの JSP Markup Language（JML）を提供します。Oracle JML サンプル・タグ・ライブラリは追加の JSP タグ・セットを提供するため、Java 文を使用せずに JSP ページを作成できます。JML 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へのコールのためのタグを提供します。JML タグ・ライブラリは、前述のとおり、XML 機能もサポートします。

次の例では、jml:for タグを使用して、大きいヘッダーから小さいヘッダーの順（H1、H2、H3、H4、H5）に、「Hello World」と繰り返し出力します。

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

注意： JSP 1.1 仕様以前の Oracle JSP バージョンでは、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す。この実装は、現在も、標準の実行時実装の代替としてサポートされています。

JSP の実行モデル

前述のとおり、Oracle JSP フレームワークは様々なサーバー環境で使用できます。Oracle JSP コンテナは、2 つの個別の実行モデルを提供します。

- 通常、JSP コンテナは、実行をトリガーする前にページをオンデマンド変換します。他のほとんどのベンダーによる JSP 実装でも同様です。
- ただし、開発者が事前にページを変換し、変換およびコンパイルされた結果を配置する場合もあります。ページを変換するには、ojspc コマンドライン・ツールを使用できます。エンド・ユーザーが JSP ページをリクエストすると、JSP ページが直接実行されます。変換は不要です。

オンデマンド変換モデル

通常、JSP ページは、オンデマンド変換モデルで実行します。これには、JServ サーブレット環境での一般的な使用方法が含まれます。

Oracle JSP コンテナが組み込まれた Web サーバーから JSP ページがリクエストされた場合（Web サーバーの構成は適切であるとする）、サーブレット oracle.jsp.JspServlet がインスタンス化および起動されます。このサーブレットは、Oracle JSP コンテナのフロントエンドとみなすことができます。

JspServlet は、JSP ページを検索し、必要に応じて（ページ実装クラスが存在しないか、または JSP ページのソースより前のタイムスタンプを持つ場合）その JSP ページを変換およびコンパイルして、実行をトリガーします。

Web サーバーを適切に構成して、JspServlet に（URL 内で）*.jsp ファイル名拡張子をマップする必要があります。JServ にこの操作を行う手順は、9-6 ページの「JServ 用の JSP ファイル名拡張子のマッピング」を参照してください。

事前変換モデル

次に示すメリットのために、開発者は JSP ページを配置する前に事前変換する場合があります。

- 実行時の変換が必要ないため、エンド・ユーザーが JSP ページをリクエストしたときの時間が短縮されます。
- 独自のソフトウェアの使用時にコードの公開を回避するために、バイナリ・ファイルのみを配置する必要がある場合にも有効です。

詳細は、6-13 ページの「事前変換用 ojspc の一般的な使用方法」および 6-27 ページの「バイナリ・ファイルのみの配置」を参照してください。

Oracle9*i* リリース 2（9.2）は、JSP ページを事前変換するための `ojspc` コマンドライン・ユーティリティを提供します。このユーティリティには、アプリケーションの配置方法によって、出力ファイルに適切なベース・ディレクトリを設定できるオプションが含まれています。`ojspc` ユーティリティについては、6-14 ページの「ojspc 事前変換ツールの詳細」を参照してください。

3

基本事項

この章では、JSP 開発の主な基本事項について説明し、データにアクセスするための JSP スタータ・サンプルを示します。

この章の内容は次のとおりです。

- アプリケーション・ルートおよびドキュメント・ルートの機能
- 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・エラーの処理
- データにアクセスするための JSP スタータ・サンプル

注意：

- JServ 環境固有の構成などの JSP 構成については、9-2 ページの「JServ 環境を使用する前に」を参照してください。
- JSP ページは、HTTP 1.0 以上をサポートするすべての標準ブラウザで表示できます。JSP ページのすべての Java コードは Web サーバーまたはデータ・サーバーで実行されるため、JDK またはエンド・ユーザーの Web ブラウザに存在するその他の Java 環境は関係ありません。

アプリケーション・ルートおよびドキュメント・ルートの機能

この項では、Servlet 2.2 の機能と Servlet 2.0 の機能を区別して、アプリケーション・ルートおよびドキュメント・ルートの概要を説明します。

Servlet 2.2 環境でのアプリケーション・ルート

前述のとおり、Servlet 2.2 仕様では、各アプリケーション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、サーバー・ファイル・システムのディレクトリ・パス（アプリケーションのモジュールのベース・パス）に関連付けられています。これをアプリケーション・ルートといいます。各アプリケーションは、独自のアプリケーション・ルートを持ちます。

これは、Web サーバーが、ドキュメント・ルートを Web アプリケーションに属する HTML ページおよび他のファイルのルートとして使用する方法に類似しています。

Servlet 2.2 環境に存在するアプリケーションの場合は、（サーブレットおよび JSP ページへの）アプリケーション・ルートと（HTML ファイルなどの静的ファイルへの）ドキュメント・ルート間で 1 対 1 のマッピングが行われます。アプリケーション・ルートとドキュメント・ルートは基本的に同じものです。

サーブレットの URL には、次の一般形式が使用されます。

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

サーブレット・コンテキストが作成されると、アプリケーション・ルートと URL の *contextpath* 部分の間でマッピングが指定されます。

たとえば、アプリケーション・ルート `/home/dir/mybankappdir` を持つアプリケーションで、このアプリケーション・ルートがコンテキスト・パス `mybank` にマップされる場合について考えてみます。また、このアプリケーションには `loginservlet` というサーブレット・パスを持つサーブレットが含まれているとします。このサーブレットは、次のとおり起動できます。

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

（エンド・ユーザーは、アプリケーション・ルートのディレクトリ名を参照できません。）

アプリケーションの HTML ページに対して引き続きこの例を適用すると、次の URL がファイル `/home/dir/mybankappdir/dir1/abc.html` を指します。

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

各サーブレット環境には、デフォルトのサーブレット・コンテキストも存在します。このコンテキストのコンテキスト・パスは「/」です。このパスは、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、デフォルトの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のコンテキストを使用するとします。この URL は、次のようになります（この場合も、ユーザーはアプリケーション・ルートのディレクトリ名を参照できません）。

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

（URL に指定されたコンテキスト・パスに一致するコンテキストが存在しない場合にも、デフォルトのコンテキストが使用されます。）

HTML ファイルに対して引き続きこの例を適用すると、次の URL がファイル `/home/mydefaultdir/dir2/def.html` を指します。

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

Servlet 2.0 環境でのアプリケーション・ルート機能の Oracle 実装

Apache JServ および他の Servlet 2.0 環境では、存在するアプリケーション環境が 1 つのみであるため、アプリケーション・ルートという概念は存在しません。Web サーバーのドキュメント・ルートがアプリケーション・ルートと同様の動作をします。

Apache の場合、通常、ドキュメント・ルートは `.../htdocs` ディレクトリとして表されます。また、`httpd.conf` 構成ファイルの `alias` の設定を介して、仮想のドキュメント・ルートを指定できます。

Servlet 2.0 環境では、Oracle JSP コンテナが、ドキュメント・ルートおよびアプリケーション・ルートに関する次の機能を提供します。

- Oracle JSP コンテナは、デフォルトでドキュメント・ルートをアプリケーション・ルートとして使用します。
- Oracle の `globals.jsa` メカニズムを介して、任意の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ドキュメント・ルート下のディレクトリを指定できます。これを実行するには、任意のディレクトリで `globals.jsa` ファイルをマーカーに設定します（9-24 ページの「globals.jsa の機能の概要」を参照）。

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、Oracle JSP コンテナによる 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を説明します。

Oracle JSP コンテナでの一般的なアプリケーションおよびセッションのサポート

Oracle JSP コンテナは、基礎となるサーブレット・メカニズムを使用して、アプリケーションおよびセッションを管理します。Servlet 2.1 および Servlet 2.2 環境の場合は、各 JSP アプリケーションに対して個別のサーブレット・コンテキストおよびセッション・オブジェクトが提供されるため、これらの基礎となるメカニズムを使用するのみで十分です。

ただし、JServ などの Servlet 2.0 環境では、サーブレット・メカニズムを使用すると問題が発生します。Servlet 2.0 仕様には Web アプリケーションの概念が明確に定義されていないため、Servlet 2.0 環境では、1 つのサーブレット・コンテナに対して 1 つのみのサーブレット・コンテキストが提供されます。また、1 つのサーブレット・コンテキストに対して提供されるセッション・オブジェクトも 1 つのみです。ただし、Oracle は、JServ および他の Servlet 2.0 環境に対して、個別のサーブレット・コンテキストおよびセッション・オブジェクトを各アプリケーションにオプションで提供できるようにするための拡張機能を提供します。（単一のアプリケーションのみをホスティングする Web サーバーの場合は不要です。）

注意： JServ および他の Servlet 2.0 環境の詳細は、9-18 ページの「JServ サーブレット環境についての考慮点」および 9-24 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・リクエスト

通常、サーブレットは、デフォルトでは HTTP セッションをリクエストしません。ただし、JSP ページ実装クラスは、デフォルトで HTTP セッションをリクエストします。このデフォルトをオーバーライドするには、次のとおり JSP の `page` ディレクティブの `session` パラメータを `false` に設定します。

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

JSP とサーブレットの相互作用

JSP ページのコーディングは多くの場合に有用ですが、サーブレットが必要な場合もあります。たとえば、4-16 ページの「JSP ページでバイナリ・データの使用を回避する理由」で説明する、バイナリ・データを出力する場合などです。

そのため、アプリケーション内で、サーブレットと JSP ページを交互に使用する必要がある場合があります。この項では、この交互に使用する方法を説明します。この項の内容は次のとおりです。

- JSP ページからのサーブレットの起動
- JSP ページから起動され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： この項では、Servlet 2.2 環境の使用を想定します。JServ および他の Servlet 2.0 環境の関連項目については、他の該当する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

JSP ページから別の JSP ページを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（1-17 ページの「JSP アクションおよび <jsp: > タグ・セット」を参照）。次に例を示します。

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

ページの実行時にこの文が検出されると、ブラウザにページ・バッファが出力され、サーブレットが実行されます。サーブレットの実行が終了すると、制御が JSP ページに戻され、そのページの実行が続行されます。これは、JSP ページから別の JSP ページに対して実行される `jsp:include` アクションと同様の機能です。

また、JSP ページから別の JSP ページに対して `jsp:forward` アクションを実行する場合と同様に、次の文によってページ・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： JServ または他の Servlet 2.0 環境に対して、インクルードまたはフォワードは実行できません。かわりに、JSP ラッパー・ページを作成する必要があります。詳細は、9-19 ページの「JServ での動的なインクルードおよびフォワード」を参照してください。

JSP ページから起動されたサーブレットへのデータの受渡し

JSP ページからサーブレットに対して動的なインクルードまたはフォワードを実行する場合は、`jsp:param` タグを使用して、（別の JSP ページに対してインクルードまたはフォワードを実行する場合と同様に）サーブレットにデータを渡すことができます。

`jsp:param` タグは、`jsp:include` または `jsp:forward` タグ内で使用されます。次の例について考えてみます。

```
<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、1-17 ページの「JSP アクションおよび <jsp: > タグ・セット」を参照してください。

また、スコープが適切に指定された JavaBeans または HTTP request オブジェクトの属性を介して、JSP ページとサーブレット間でデータの受渡しを行うことができます。request オブジェクトの属性の使用については、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

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 JSP ページを起動できます。このメカニズムを使用するには、コードで次の手順を実行します。

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. ターゲットの 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 `getRequestDispatcher()` メソッドへの入力に指定して、サーブレット・コンテキスト・インスタンスからリクエスト・ディスパッチャを取得します。

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

   この手順の実行前または実行中に、オプションで、HTTP request オブジェクトの属性を介して JSP ページでデータを使用できるようにすることができます。詳細は、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。

3. HTTP request オブジェクトおよび response オブジェクトを引数に指定して、リクエスト・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);
   ```

   これらのメソッドの機能は、`jsp:include` および `jsp:forward` アクションの機能に類似しています。`include()` メソッドでは、制御が一時的に譲渡された後で、起動先のサーブレットによる実行に戻ります。

   `forward()` メソッドでは出力バッファが消去されることに注意してください。

   注意：

   - request および response オブジェクトは、`javax.servlet.http.HttpServlet` クラスに指定された `doGet()` メソッドなどの標準サーブレット機能を使用して事前に取得されています。
   - この機能は、Servlet 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

3-6 ページの「サーブレットからの JSP ページの起動」で説明したとおり、リクエスト・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、オプションで、HTTP request オブジェクトを介してデータを渡すことができます。

この受渡しを実行するには、次のいずれかの方法を使用します。

- リクエスト・ディスパッチャの取得時に、*name*=*value* の組合せとともに「?」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");
  ```

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

- HTTP request 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");
  ```

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

注意：

- この機能は、Servlet 2.1 仕様で導入されました。Servlet 2.1 仕様と Servlet 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。Servlet 2.1 環境では、任意の属性を設定できるのは 1 回のみです。
- `jsp:param` タグのかわりにこの項で説明する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、前述の項で説明した機能を使用する JSP ページおよびサーブレットについて説明します。JSP ページ Jsp2Servlet.jsp は、サーブレット MyServlet をインクルードします。このサーブレットは、別の JSP ページ welcome.jsp をインクルードします。

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

```
<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 パッケージは、セッション・リソースを管理するための標準メカニズムを提供します。また、Oracle は、アプリケーション、セッション、ページおよびリクエストのリソースを管理するための拡張機能を提供します。

HttpSessionBindingListener を使用した標準セッション・リソース管理

JSP ページは、実行時に取得されるリソース（JDBC 接続、文、結果セット・オブジェクトなど）を適切に管理する必要があります。標準の javax.servlet.http パッケージは、session スコープが設定されたリソースを管理するための HttpSessionBindingListener インタフェースおよび HttpSessionBindingEvent クラスを提供します。このメカニズムでは、たとえば、session スコープが設定された問合せ Bean によって、Bean のインスタンス化時にデータベース・カーソルを取得し、HTTP セッションの終了時にデータベース・カーソルをクローズすることができます。(3-19 ページの「データにアクセスするための JSP スタータ・サンプル」に示す例では、問合せが実行されるたびに接続がオープンおよびクローズされるため、オーバーヘッドが増加します。)

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 インスタンスでは、HTTP セッション・オブジェクトのイベント通知リストにこのインスタンス自体を登録する必要がありますが、jsp:useBean 文ではこれが自動的に処理されます。

---

valueBound() および 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() メソッドおよび valueUnbound() メソッドを実装できます。これらの各メソッドは、HttpSessionBindingEvent インスタンスを入力に取ります。これらのメソッドは、Servlet コンテナによってコールされます。セッションにオブジェクトが格納される場合は valueBound() メソッド、セッションからオブジェクトが削除される場合、またはセッションがタイムアウトするか、または無効になった場合は 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するリソースを解放します（次の例では、データベース接続が切断されます）。

> **注意：** Oracle9*i* リリース 2（9.2）は、他のリソースを管理するための拡張機能を提供します。これによって、JavaBeans をプログラミングして、session スコープが設定されたリソースに加えて、page スコープ、request スコープまたは application スコープが設定されたリソースを管理できます。5-2 ページの「JspScopeListener を使用した Oracle JSP によるイベント処理」を参照してください。

次の「JavaBeans（JDBCQueryBean）のコード」の項に、HttpSessionBindingListener を実装する JavaBeans の例、および Bean をコールする JSP ページの例を示します。

JavaBeans（JDBCQueryBean）のコード

次に、HttpSessionBindingListener インタフェースを実装する JDBCQueryBean という JavaBeans のサンプル・コードを示します。（このサンプル・コードでは、データベース接続に JDBC OCI ドライバが使用されます。このサンプル・コードを実行するには、適切な JDBC ドライバおよび接続文字列を使用します。）

（3-14 ページの「JSP ページ（UseJDBCQueryBean）」で説明するとおり）JDBCQueryBean は、HTML リクエストを介して検索条件を取得し、その検索条件に基づいて動的問合せを実行して、結果を出力します。

このクラスは、（HttpSessionBindingListener インタフェースで指定されたとおり）valueUnbound() メソッドを実装します。これによって、セッション終了時にデータベース接続がクローズされます。

```
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、サンプルとしてのみ機能します。この方法は、大規模な Web アプリケーションのデータベース接続プーリングの処理には有効でない場合があります。

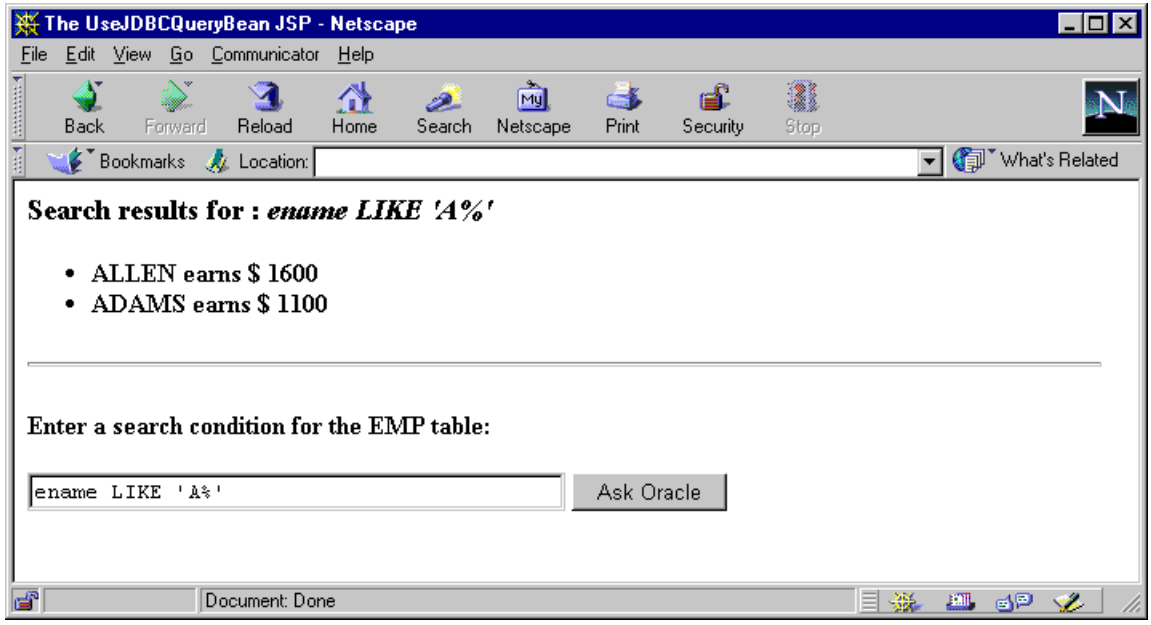
JSP ページ（UseJDBCQueryBean）

次の JSP ページは、3-11 ページの「JavaBeans（JDBCQueryBean）のコード」で定義された JDBCQueryBean という JavaBeans を使用して、session スコープが設定された Bean を起動します。JDBCQueryBean によって、ユーザーが入力した検索条件に一致する従業員名が表示されます。

JDBCQueryBean は、この JSP ページの jsp:setProperty コマンドを介して、検索条件を取得します。これによって、ユーザーが HTML 形式で入力した searchCond リクエスト・パラメータの値に従って、Bean の searchCond プロパティが設定されます。（HTML の INPUT タグは、この形式で入力された検索条件に searchCond という名前を指定します。）

```
<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>
```

次に、このページの入力および出力の例を示します。



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()` メソッドで接続をクローズしています。セッションを閉じた後に Bean のガベージ・コレクションが実行されると、`finalize()` メソッドがコールされます。ただし、`HttpSessionBindingListener` インタフェースの動作は、`finalize()` メソッドの動作より予測可能です。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よって異なります。一方、`HttpSessionBindingListener` インタフェースの `valueUnbound()` メソッドは、セッションの停止時に常にコールされます。

リソース管理に対する Oracle の拡張機能の概要

Oracle は、ページおよびリクエストのリソースに加えて、アプリケーションおよびセッションのリソースを管理するための次の拡張機能を提供します。

- `JspScopeListener` - アプリケーション、セッション、ページまたはリクエストのリソースの管理に使用します。

  詳細は、5-2 ページの「JspScopeListener を使用した Oracle JSP によるイベント処理」を参照してください。

- `globals.jsa` アプリケーションおよびセッション・イベント - 通常、JServ などの Servlet 2.0 環境で、アプリケーションおよびセッションの開始イベントおよび終了イベントのために使用します。

  詳細は、9-28 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・エラーの処理

JSP ページでクライアント・リクエストの実行および処理中に、ページの内側または外側（コールされた JavaBeans の場合など）のいずれかでランタイム・エラーが発生する可能性があります。この項では、JSP のエラー処理メカニズムについて説明し、簡単な例を示します。

JSP のエラー・ページの使用

JSP ページの実行時に発生したランタイム・エラーは、次の 2 つのいずれかの方法で標準の Java 例外メカニズムを使用して処理されます。

- 標準の Java 例外処理コードを使用して、JSP ページ内の Java スクリプトレットで例外を検出および処理できます。
- JSP ページで例外が検出されない場合は、リクエストおよび検出されなかった例外がエラー・ページに転送されます。JSP のエラーの処理には、この方法の使用をお薦めします。

発生元の JSP ページで `page` ディレクティブの `errorPage` パラメータを設定して、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の 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）。

エラー・ページでは、`page` ディレクティブの `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、`java.lang.Exception` インスタンスです。このインスタンスは、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可能です。

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 オブジェクトなどの JSP の暗黙的なオブジェクトの詳細は、1-15 ページの「暗黙的なオブジェクト」を参照）。

エラー・ページの使用方法の例は、3-17 ページの「JSP のエラー・ページの例」を参照してください。

注意： JSP 1.1 仕様では、JSP メカニズムを介して処理可能な例外のタイプに関して、明確に記載されていません。

Oracle JSP コンテナでは、トランスレータが生成したページ実装クラスによって、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。ただし、java.lang.Throwable クラスまたは 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、JSP コンテナによって Servlet コンテナにスローされます。

JSP 1.2 仕様では、JSP メカニズムを介して処理可能な例外のタイプに関して、明確に記載されています。Oracle の動作は、将来のリリースで改善されます。

JSP のエラー・ページの例

次の nullpointer.jsp の例では、エラーを生成し、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を出力します。

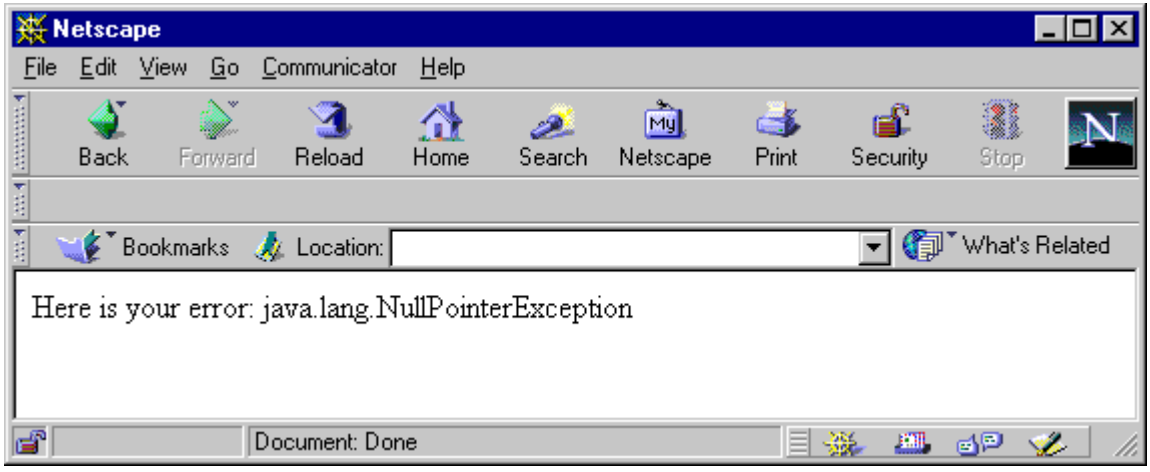
nullpointer.jsp のコード

```
<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>
```

この例の出力結果は次のとおりです。



注意： `nullpointer.jsp` の「Null pointer is generated below:」という行は、処理がエラー・ページにフォワードされた場合は出力されません。これが `jsp:include` と `jsp:forward` の機能の相違点です。`jsp:forward` では、「forward-from」ページからの出力が、「forward-to」ページからの出力に置換されます。

データにアクセスするための JSP スタータ・サンプル

第 1 章「概要」に、2 つの簡単な JSP の例を示しています。ただし、Oracle JSP コンテナを使用する場合は、Oracle データベースにアクセスする必要があります。この項では、JSP ページで標準の JDBC コードを使用して問合せを実行する例を示します。

JDBC API は単に Java インタフェースのセットであるため、JavaServer Pages テクノロジでは、JSP スクリプトレット内での JDBC API の使用が直接サポートされます。

注意：

- Oracle JDBC で使用可能なドライバは、次の 4 種類です。
  - 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
  - 基本的に Applet を含むすべてのクライアント環境で使用可能な 100% Java で記述された JDBC Thin ドライバ
  - Oracle データベース内から別の Oracle データベースにアクセスするための JDBC サーバー側 Thin ドライバ
  - 内部で（Java ストアド・プロシージャなどによって）Java コードが実行されているデータベースにアクセスするための JDBC サーバー側内部ドライバ

  Oracle JDBC の詳細は、『Oracle9*i* 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。
- Oracle JSP コンテナは、静的な SQL 操作に対する SQLJ（Java に埋め込まれた SQL）の使用もサポートします。詳細は、5-3 ページの「Oracle JSP による Oracle SQLJ のサポート」を参照してください。

次の例では、ユーザーが（ボックスに入力し、`Ask Oracle` ボタンを押すことによって）HTML 形式で入力した検索条件から、問合せが動的に作成されます。指定された問合せを実行するには、JSP 宣言に定義されている `runQuery()` というメソッドで JDBC コードを使用します。また、JSP 宣言内に `formatResult()` メソッドを定義して、出力を生成します。`runQuery()` メソッドには、パスワード `tiger` で `scott` スキーマを使用します。

HTML の `INPUT` タグは、この形式で入力された文字列に `cond` という名前を指定します。そのため、`cond` は、この HTTP リクエストに対する暗黙的な `request` オブジェクトの `getParameter()` メソッドへの入力パラメータ、および（`cond` 文字列を問合せの `WHERE` 句に挿入する）`runQuery()` メソッドへの入力パラメータにもなります。

注意：

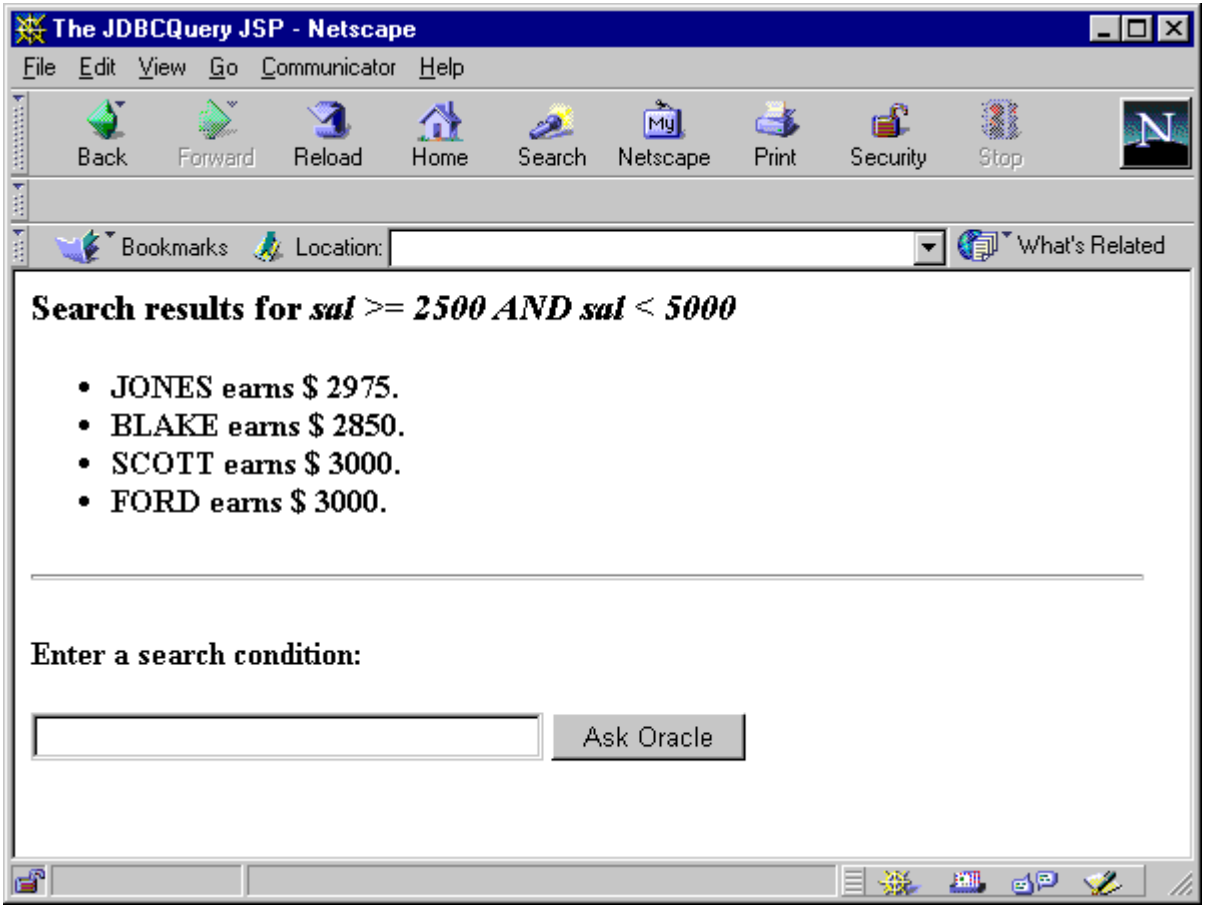
- この例は、<%!...%> 宣言構文ではなく、<%...%> スクリプトレット構文で runQuery() メソッドを定義して実行することもできます。
- この例では、JDBC OCI ドライバを使用します。そのため、Oracle クライアントをインストールする必要があります。このサンプルを実行する場合は、適切な JDBC ドライバおよび接続文字列を使用します。

```
<%@ 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();
  }
%>
```

次の図に、次の入力に対する出力例を示します。

```
sal >= 2500 AND sal < 5000
```



4

重要な考慮点

この章では、JSP アプリケーションの開発でのプログラミング、構成およびランタイムに関する重要な考慮点について説明します。この章の内容は次のとおりです。

- JSP プログラミングの一般的な方針、ヒントおよび注意事項
- JSP の構成に関する主な問題
- Oracle JSP による実行時のページおよびクラスの再ロード

JSP プログラミングの一般的な方針、ヒントおよび注意事項

この項では、ターゲット環境に関係なく、Oracle JSP コンテナで実行する JSP ページをプログラミングする場合に考慮する必要がある問題について説明します。この項の内容は次のとおりです。

> **注意：** この項で説明する事項に加えて、Oracle JSP による変換および配置に関する問題およびこれらの動作についても注意する必要があります。第 6 章「JSP による変換および配置」を参照してください。

JavaBeans とスクリプトレットの比較

1-5 ページの「ビジネス・ロジックのページ表示からの分離 : JavaBeans のコール」では、JavaServer Pages テクノロジの主なメリットについて説明しています。ビジネス・ロジックを含み、動的コンテンツを決定する Java コードは、リクエストの処理、プレゼンテーション・ロジックおよび静的コンテンツを含む HTML コードと分離できます。これによって、HTML の専門家は JSP ページ自体のプレゼンテーション・ロジックに集中し、Java の専門家は JSP ページからコールされる JavaBeans のビジネス・ロジックに集中して作業することができます。

通常の JSP ページには、Java コードの簡単なフラグメントのみが含まれます。これらは、通常、リクエストの処理または表示のための Java 機能で使用されます。3-19 ページの「データにアクセスするための JSP スタータ・サンプル」に示すサンプル・ページは説明用で、最適な設計ではありません。サンプルに含まれる `runQuery()` メソッドなどによるデータ・アクセスは、通常、JavaBeans での使用に適しています。一方、サンプルに含まれる `formatResult()` メソッド（出力の書式を設定）は、JSP ページ自体への使用に適しています。

JDBC のパフォーマンス改善機能の使用

Oracle JSP コンテナによって実行される JSP アプリケーションでは、次のパフォーマンス改善機能を使用できます。これらの機能は、Oracle JDBC の拡張機能を介してサポートされます。

- データベース接続のキャッシュ
- JDBC 文のキャッシュ
- UPDATE 文のバッチ処理
- 問合せ時の行のプリフェッチ
- 行セットのキャッシュ

これらのほとんどのパフォーマンス機能が、Oracle の `ConnBean` および `ConnCacheBean` データ・アクセス JavaBeans でサポートされます（`DBBean` ではサポートされません）。これらの Bean については、『Oracle9*i*AS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

データベース接続のキャッシュ

新しいデータベース接続の作成は高コストな操作であり、できるかぎり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ます。JSP アプリケーションでは、物理的な接続の既存のプールから論理的な接続を取得し、終了時に接続をそのプールに戻すことができます。

`application`、`session`、`page` または `request` の 4 つの JSP スコープのいずれかで、接続プールを作成できます。有効範囲が最も大きいスコープを使用することが最も効率的です。Web サーバーで許可されている場合は `application` スコープ、それ以外の場合は `session` スコープを使用します。

Oracle JDBC の接続キャッシュ・スキームは、JDBC 2.0 の標準拡張機能に指定されているとおり、標準の接続プーリングに基づいて構築されます。このスキームは、Oracle9*i* に付属の `ConnCacheBean` データ・アクセス JavaBeans に実装されています。通常、JSP 開発者は、この方法で接続キャッシュを使用します。この Bean については、『Oracle9*i*AS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

次の例に示すとおり、Oracle JDBC の `OracleConnectionCacheImpl` クラスは、（すべての `OracleConnectionCacheImpl` 機能は `ConnCacheBean` を介しても使用可能です）JavaBeans と同様に直接使用することもできます。

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

`OracleConnectionCacheImpl` では、`ConnCacheBean` と同様のプロパティが使用可能です。これらのプロパティは、`jsp:setProperty` 文を介して設定するか、またはクラス設定メソッドを介して直接設定できます。

Oracle JDBC の接続キャッシュ・スキームおよび OracleConnectionCacheImpl クラスの詳細は、『Oracle9*i* 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

Oracle JDBC の拡張機能である文キャッシュでは、単一の物理接続内（ループまたは繰り返しコールされるメソッド内など）で繰り返し使用される実行可能な文をキャッシュすることによって、パフォーマンスが改善されます。文をキャッシュすると、文が実行されるたびに、文の再解析、文オブジェクトの再作成およびパラメータ・サイズの定義の再計算を行う必要がなくなります。

Oracle JDBC の文キャッシュ・スキームは、Oracle9*i* に付属の ConnBean および ConnCacheBean データ・アクセス JavaBeans に実装されます。これらの各 Bean には、jsp:setProperty 文または Bean の setStmtCacheSize() メソッドを介して設定可能な stmtCacheSize プロパティが含まれます。これらの Bean については、『Oracle9*i*AS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

文キャッシュは、Oracle JDBC の OracleConnection および OracleConnectionCacheImpl クラスを介して直接実行することもできます。Oracle JDBC の文キャッシュ・スキーム、および OracleConnection と OracleConnectionCacheImpl クラスの詳細は、『Oracle9*i* 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： 文は、単一の物理接続内のみでキャッシュできます。接続キャッシュに対して文キャッシュを有効にすると、プールされた単一の接続オブジェクトと複数の論理接続オブジェクトの間にまたがって文をキャッシュできます。ただし、プールされた複数の接続オブジェクト間にまたがってはキャッシュできません。

バッチ更新

Oracle JDBC のバッチ更新機能によって、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）が関連付けられます。バッチ更新では、JDBC ドライバは、JDBC ドライバの実行メソッドがコールされるたびにプリコンパイルされた SQL 文を実行するのではなく、累積された実行リクエストのバッチに文を追加します。ドライバは、すべての操作をデータベースに渡し、バッチ値に達すると 1 回実行されます。たとえば、バッチ値が 10 の場合、10 の操作が 1 つのバッチとしてデータベースに送信され、1 回のトリップで処理されます。

Oracle JSP コンテナは、ConnBean データ・アクセス JavaBeans の executeBatch プロパティを介して、Oracle JDBC のバッチ更新を直接サポートします。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 を使用すると、作成した接続および文オブジェクトで、Oracle JDBC 機能を使用したバッチ更新を有効にできます。これらの Bean については、『Oracle9*i*AS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

Oracle JDBC のバッチ更新の詳細は、『Oracle9*i* 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。

行プリフェッチ

Oracle JDBC の行プリフェッチ機能を使用すると、問合せ時の結果セットの移入中に、データベースへの 1 回のトリップでクライアントにプリフェッチする行の数を設定できます。これによって、サーバーへのラウンドトリップの回数が減ります。

Oracle JSP コンテナは、ConnBean データ・アクセス JavaBeans の preFetch プロパティを介して、Oracle JDBC の行プリフェッチを直接サポートします。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 を使用すると、作成した接続および文オブジェクトで、Oracle JDBC 機能を使用した行プリフェッチを有効にできます。これらの Bean については、『Oracle9*i*AS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

Oracle JDBC の行プリフェッチの詳細は、『Oracle9*i* 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

行セットをキャッシュすると、取得されたデータに対して、シリアル化、スクロールおよび接続の切断が可能なコンテナが提供されます。この機能は、頻繁に変更されない小規模なデータ・セットに有効です。特に、クライアントが、その情報に頻繁または継続的にアクセスする必要がある場合に有効です。一方、通常の結果セットを使用する場合は、基礎となる接続およびその他のリソースを保持しておく必要があります。ただし、大規模な行セットをキャッシュすると、クライアント上で大量のメモリーが消費されることに注意してください。

Oracle9*i* では、Oracle JDBC が、キャッシュされた行セットの実装を提供します。Oracle JDBC ドライバを使用する場合は、JSP ページ内でコードを使用して、キャッシュされた行セットを次のとおり作成および移入します。

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

行セットを移入すると、元の結果セットの取得に使用された接続および文オブジェクトをクローズできます。

Oracle JDBC の行セット・キャッシュの詳細は、『Oracle9*i* 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。

静的インクルードと動的インクルードの比較

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、インクルード対象のページのコピーを作成し、変換時にそのコピーを JSP ページ（インクルード元のページ）に送信します。これは、静的インクルード（または変換時インクルード）と呼ばれます。静的インクルードでは、次の構文が使用されます。

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

1-17 ページの「JSP アクションおよび <jsp: > タグ・セット」で説明した jsp:include アクションでは、実行時に、インクルード元のページの出力内にインクルード対象のページの出力が動的にインクルードされます。これは、動的インクルード（または実行時インクルード）と呼ばれます。動的インクルードでは、次の構文が使用されます。

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

静的インクルードは、C 構文の #include 文と同等です。動的インクルードは、ファンクション・コールに類似しています。これらはいずれも有効ですが、目的は異なります。

注意： 静的インクルードおよび動的インクルードは、同じサーブレット・コンテキストのページ間のみで使用できます。

静的インクルードのプロセス

静的インクルードを実行すると、変換時（include ディレクティブが実行された時点）にインクルード対象のページのテキストがインクルード元のページに物理的にコピーされたものとして、インクルード元の JSP ページに対して生成されるコードのサイズが大きくなります。ページが、インクルード元のページ内に複数回インクルードされた場合は、複数のコピーが作成されます。

静的にインクルードされる JSP ページは、独立した変換可能なエンティティとして存在する必要はありません。JSP ページは、インクルード元のページにコピーされるテキストで構成されます。インクルード元のページは、インクルード対象のテキストのコピー後に、変換可能になる必要があります。実際は、インクルード対象のページがコピーされる前に、インクルード元のページが変換可能になる必要はありません。静的にインクルードされる一連のページは、それぞれが切り離すことができないフラグメントとして存在する場合もあります。

動的インクルードのプロセス

動的インクルードでは、リクエスト・ディスパッチャなどに対するメソッドのコールが追加されても、インクルード元のページに対して生成されるコードのサイズが大幅に大きくなることはありません。動的インクルードを実行すると、インクルード対象のページのテキストがインクルード元のページに物理的にコピーされるのではなく、実行時処理がインクルード元のページからインクルード対象のページに切り替えられます。

動的インクルードでは、リクエスト・ディスパッチャに対する追加のコールが必要であるため、処理オーバーヘッドが増加します。

動的にインクルードされるページは、独立したエンティティとして存在し、個別に変換および実行できる必要があります。同様に、インクルード元のページも、動的インクルードを行うことなく変換および実行が可能な、独立したエンティティとして存在する必要があります。

メリット、デメリットおよび代表的な使用例

静的インクルードはページ・サイズに影響し、動的インクルードは処理オーバーヘッドに影響します。静的インクルードは、動的インクルードで発生するリクエスト・ディスパッチャによるオーバーヘッドを回避できますが、大きいファイルの処理には適していません。（生成されるページ実装クラスのサービス・メソッドに対する最大値は 64KB です。4-10 ページの「JSP ページに大量の静的コンテンツが含まれている場合の対策」を参照してください。）

静的インクルードを乱用すると、JSP ページのデバッグが困難になる可能性もあります。これによって、プログラム実行のトレースが困難になります。静的にインクルードしたページ間には、相互依存性が存在しないようにしてください。

通常、静的インクルードは、複数の JSP ページで繰り返し使用されるコンテンツを持つ小さいファイルのインクルードに使用されます。次に例を示します。

- アプリケーションの各ページの上部または下部へのロゴまたは著作権情報の静的インクルード。
- 複数のページで必要な宣言またはディレクティブ（Java クラスのインポートなど）を持つページの静的インクルード。
- アプリケーションの各ページの集中型の「状態チェック」ページの静的インクルード（4-9 ページの「集中チェック・ページの使用」を参照）。

動的インクルードは、モジュラー・プログラミングに有効です。個別に実行されるページが存在する場合も、他のページの出力を生成するために使用されるページが存在する場合もあります。動的にインクルードされたページは、インクルード元のページのサイズを大きくすることなく、複数のインクルード元のページで再利用できます。

JSP タグ・ライブラリの作成および使用が必要な場合

いくつかの状況では、開発チームがカスタム・タグの作成および使用を検討する必要があります。特に、次の状況について考えてみます。

- 出力の表示およびフォーマットに関する大量の Java ロジックを JSP ページにインクルードする必要がある場合。
- JSP 出力の特別な操作または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 プログラミングに精通しているとはかぎらないため、ページでの Java ロジック（JSP 出力の表示およびフォーマットを指定するロジックなど）のコーディングに適さない場合があります。

このような状況では、JSP タグ・ライブラリが有効な場合があります。多くの JSP ページの出力に Java ロジックが必要な場合は、Java ロジックのかわりにタグ・ライブラリを使用すると効率的です。

Oracle9*i* に付属の JML サンプル・タグ・ライブラリがその一例です。Java のループおよび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るこのライブラリの詳細は、『Oracle9*i*AS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

JSP 出力の操作またはリダイレクト

カスタム・タグを使用する一般的な状況の 1 つに、レスポンス出力の特別な実行時処理が必要な場合があります。出力の処理手順を追加するか、または出力をブラウザ以外の出力先へリダイレクトする機能が必要な場合などです。

たとえば、次の例は、テキストの本体を囲むことで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カスタム・タグです（ここで、`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>
```

タグ本体の処理の詳細は、7-4 ページの「タグ・ハンドラ」を参照してください。

集中チェック・ページの使用

JSP アプリケーションの通常の管理または監視には、アプリケーションの各ページからインクルードする集中チェック・ページが有効です。集中チェック・ページを使用すると、各ページの実行時に次のようなタスクを実行できます。

- セッション状態の確認。
- ログイン状態の確認（有効なログインが行われているかどうかを Cookie で確認するなど）。
- 使用方法のプロファイルの確認（ロギング・メカニズムが、マウスのクリックやページへのアクセスなど、特定のイベントを記録するために実装されているかどうか）。

この他にも、多くの使用方法があります。

たとえば、`HttpSessionBindingListener` インタフェースを実装するセッション・チェック用クラス `MySessionChecker` について考えてみます（3-10 ページの「HttpSessionBindingListener を使用した標準セッション・リソース管理」を参照）。

```
public class MySessionChecker implements HttpSessionBindingListener
{
   ...

   valueBound(HttpSessionBindingEvent event)
   {...}

   valueUnbound(HttpSessionBindingEvent event)
   {...}

   ...
}
```

次のようなコードを含むチェック用 JSP ページ（たとえば、`centralcheck.jsp`）を作成できます。

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

`centralcheck.jsp` がインクルードされるすべてのページで、Servlet コンテナは、`sessioncheck` が範囲外になった直後（セッションの終了時）に、`MySessionChecker` クラスに実装された `valueUnbound()` メソッドをコールします。これによって、セッション・リソースが管理されます。アプリケーションの各 JSP ページの最後に、`centralcheck.jsp` をインクルードできます。

JSP ページに大量の静的コンテンツが含まれている場合の対策

大量の静的コンテンツ（基本的に、実行時に変更されるコンテンツを持たない大量の HTML コード）を含む JSP ページは、変換および実行の速度が低下する場合があります。

この場合、主に 2 つの対策があります（いずれの対策を講じても変換が高速化されます）。

- 静的 HTML を個別のファイルに挿入し、動的な `include` コマンド（`jsp:include`）を使用して、実行時に出力が JSP ページの出力にインクルードされるようにします。`jsp:include` コマンドの詳細は、1-17 ページの「JSP アクションおよび <jsp: > タグ・セット」を参照してください。

  重要： 静的な `<%@ include... %>` コマンドは、機能しません。このコマンドを使用すると、インクルード対象のファイルが、変換時にそのコードをインクルード元のページにコピーしてインクルードされます。この方法では、問題を解決できません。

- 静的 HTML を Java リソース・ファイルに挿入します。

  JSP の `external_resource` 構成パラメータを有効にすると、Oracle JSP コンテナがこの操作を実行します。このパラメータの詳細は、9-7 ページの「Oracle JSP の構成パラメータ」を参照してください。

  事前変換の場合は、`ojspc` コマンドライン・ツールの `-extres` オプションも同様に機能します。

  注意： リソース・ファイルに静的 HTML を挿入すると、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ため、前述の `jsp:include` による対策よりメモリー・フットプリントが大きくなる場合があります。

発生する可能性は低いですが、大量の静的コンテンツを含む JSP ページには、ほとんどの JVM で、単一のメソッド内のコード・サイズが 64KB に制限されるという問題が発生する場合もあります。このコードは、`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、このことが JSP ページで問題になる場合があります。基本的に JSP ページのソース・ファイル全体から生成された Java コードは、ページ実装クラスのサービス・メソッドに送信されるためです。（ブラウザに静的 HTML を出力するための Java コードが生成され、すべてのスクリプトレットの Java コードが直接コピーされます。）

発生する可能性はほとんどありませんが、JSP ページの Java スクリプトレットのサイズが、サービス・メソッドのサイズ制限を超える問題が発生する場合もあります。ページに大量の Java コードが含まれることによって問題が発生する場合は、そのコードを JavaBeans に移す必要があります。

メソッド変数宣言とメンバー変数宣言の比較

1-11 ページの「スクリプト要素」で説明したとおり、メンバー変数を宣言するには JSP の `<%! ... %>` 宣言を使用し、メソッド変数を宣言するには `<% ... %>` スクリプトレットを使用します。

変数の使用方法に応じて、各宣言に適切なメカニズムを使用してください。

- JSP 宣言構文 `<%! ... %>` で宣言された変数は、JSP トランスレータによって生成されたページ実装クラスのクラス・レベルで宣言されます。
- JSP スクリプトレット構文 `<% ... %>` で宣言された変数は、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になります。

次の `decltest.jsp` の例について考えてみます。

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

この場合、ページ実装クラスに次のようなコードが生成されます。

```
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、概念の説明のみに使用されます。ほとんどのクラスが簡素化の目的で削除されているため、Oracle JSP トランスレータによって生成されるページ実装クラスの実際のコードは、このコードとは異なります。

ページ・ディレクティブの特長

この項では、page ディレクティブの次の特長について説明します。

- page ディレクティブは静的で、変換時に機能します。実行時に評価されるパラメータの設定は指定できません。
- page ディレクティブの Java import 設定は、JSP ページ内に累積されます。

静的な page ディレクティブ

page ディレクティブは静的で、変換時に解析されます。実行時に解析される動的設定は指定できません。次の例について考えてみます。

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、エラーが発生します（この例では、EUCJIS はハードコードですが、任意のキャラクタ・セットが実行時に動的に決定されるように設定されています）。

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

一部の page ディレクティブ設定には、対策があります。例 2 の場合は、8-4 ページの「コンテンツ・タイプの動的設定」で説明するとおり、setContentType() メソッドを使用すると、コンテンツ・タイプを動的に設定できます。

page ディレクティブの累積型の import 設定

単一の JSP ページ内の page ディレクティブの Java import 設定は、累積型です。

単一の JSP ページ内で、次の 2 つの例は同等です。

```
<%@ 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 設定後、2 番目の page ディレクティブの import 設定では、インポートされる一連のクラスまたはパッケージは置換されるのではなく、追加されます。

JSP による空白の保持およびバイナリ・データでの JSP の使用

Oracle JSP コンテナ（および、一般的な JavaServer Pages 実装）は、改行などのソース・コード内の空白を、ブラウザへの出力に保持します。このような空白の挿入は、開発者の意図とは異なる場合があるため、通常、JSP テクノロジは、バイナリ・データの生成には適していません。

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るかどうかによって、異なる HTML 出力が生成されます。

例 1 - 改行なし

次の JSP ページでは、Date() および getParameter() コールの後に改行が挿入されていません。(Date() コールで始まる 3 行目および 4 行目は、この例では行が折り返されていますが、実際は 1 行のコードです。)

nowhitsp.jsp:

```
<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>
```

これによって、次の HTML がブラウザに出力されます(日付の後に空白行が存在し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 - 改行あり

次の JSP ページでは、Date() および getParameter() コールの後に改行が挿入されています。

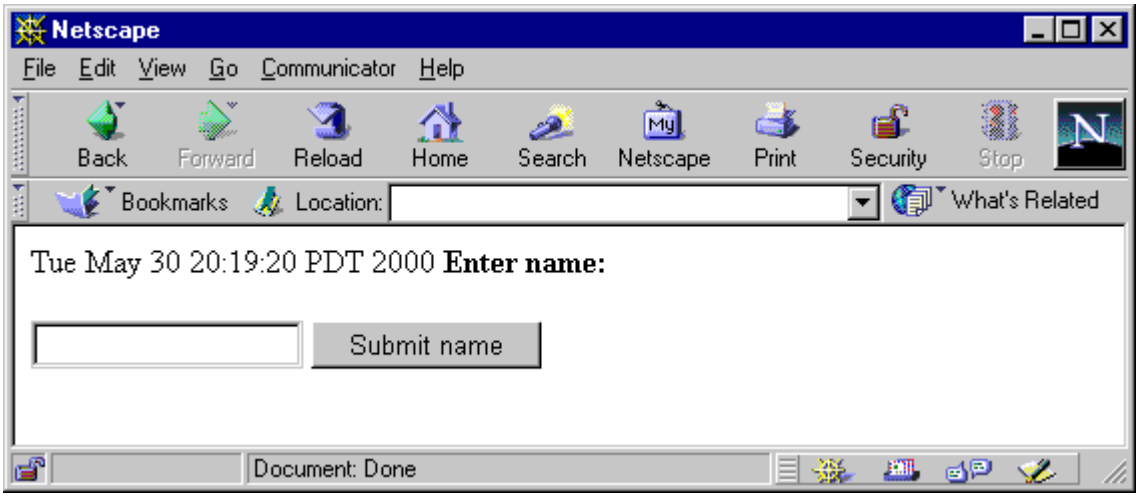
whitesp.jsp:

```
<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>
```

これによって、次の HTML がブラウザに出力されます。

```
<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 行の空白行があることに注意してください。この場合、両方の例で生成されるブラウザ上の表示が同じ外観であるため、違いは大きくありません。ただし、この説明では空白の保持の要点が示されています。



JSP ページでバイナリ・データの使用を回避する理由

次の理由から、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、JSP ページではなくサーブレ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、RAW バイトを書き込むためのメソッドが存在しません）。
- 実行時、JSP コンテナは空白を保持します。空白が不要な場合もあるため、ブラウザにバイナリ出力（`.gif` ファイルなど）を生成する場合または空白が重要なその他の場合に、JSP ページの使用は適していません。

  次の例について考えてみます。

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

  この場合、ブラウザは、バイナリ・データの途中または最後（出力バッファのバッファリングによって異なる）に含まれている不要な改行文字を受け取ります。この問題を回避するには、コードの行と行の間に改行を使用しないでください。ただし、これはプログラミング方法としては不適切です。

JSP ページでバイナリ・データの生成を試行すると、動的なテキスト・コンテンツのプログラミングを簡略化するという JSP テクノロジのメリットが大幅に失われます。

Oracle での XML のサポート

この項では、JSP ページで有効な場合がある XML 用の機能に対する Oracle のサポートについて説明します。

- XML の代替構文
- OracleXMLQuery クラス

XML および XSL の追加サポートの詳細は、『Oracle9*i*AS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

XML の代替構文

スクリプトレット用の <%...%>、宣言用の <%!...%>、式用の <%=...%> などの JSP タグは、XML 文書内では構文上有効ではありません。Sun 社では、JavaServer Pages 仕様バージョン 1.1 にこのことを記載しています。この JavaServer Pages 仕様には、XML と互換性のある構文を使用する同等の JSP タグが定義されています。この定義は、XML 文書の始めの jsp:root 開始タグ内に指定可能な標準 DTD を介して実装されます。

たとえば、この機能を使用すると、XML オーサリング・ツールに XML ベースの JSP ページを作成できます。

Oracle JSP コンテナでは、この DTD の直接使用および jsp:root タグの使用の要求は行われません。ただし、Oracle JSP 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ための論理は含まれています。表 4-1 に、この構文を示します。

表 4-1 XML の代替構文

標準 JSP 構文	XML の代替 JSP 構文
<%@ *directive* ...%>	<jsp:directive.*directive* ... />
例: <%@ page ... %> <%@ include ... %>	例: <jsp:directive.page ... /> <jsp:directive.include ... />
<%! ... %>（宣言）	<jsp:declaration> *...declarations go here...* </jsp:declaration>
<%= ... %>（式）	<jsp:expression> *...expression goes here...* </jsp:expression>
<% ... %>（スクリプトレット）	<jsp:scriptlet> *...code fragment goes here...* </jsp:scriptlet>

jsp:useBean などの JSP アクション・タグでは、すでに多くの部分で XML に準拠する構文が使用されています。ただし、引用規則またはリクエスト時の属性式によって、変更が必要な場合があります。

OracleXMLQuery クラス

oracle.xml.sql.query.OracleXMLQuery クラスは、データベース問合せで使用される XML 機能の XML SQL Utility の一部として、Oracle9*i* に付属しています。このクラスでは、xsu12.jar（JDK 1.2.x の場合）または xsu111.jar（JDK 1.1.x の場合）のいずれかのファイルが必要です。これらの両方のファイルは、Oracle9*i* に付属しています。

OracleXMLQuery クラスおよびその他の XML SQL Utility 機能の詳細は、『Oracle9*i* XML Developer's Kit ガイド - XDK』を参照してください。

JSP の構成に関する主な問題

この項では、主な page ディレクティブ・パラメータおよび JSP 構成パラメータの設定方法による重要な影響について説明します。主に、JSP ページの最適化、CLASSPATH の問題およびクラス・ローダーの問題について説明します。この項の内容は次のとおりです。

- JSP の実行の最適化
- CLASSPATH およびクラス・ローダーの問題

JSP の実行の最適化

いくつかの設定によって JSP のパフォーマンスを最適化できます。次に例を示します。

- JSP ページのバッファの無効化
- 再変換のための確認の回避
- HTTP セッションの使用の回避

JSP ページのバッファの無効化

デフォルトでは、JSP ページは、ページ・バッファと呼ばれるメモリー領域を使用します。ページで動的グローバリゼーション・サポートのコンテンツ・タイプの設定、フォワードまたはエラー・ページが使用される場合、このバッファ（デフォルトでは 8KB）が必要です。これらの機能が使用されない場合は、page ディレクティブでバッファを無効にできます。

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

これによって、メモリー使用量が減り、出力手順が短くなるため、ページのパフォーマンスが向上します。出力は、最初にバッファを介さずに、ブラウザに直接送信されます。

再変換のための確認の回避

Oracle JSP コンテナは、JSP ページの実行時、デフォルトで、ページ実装クラスがすでに存在しているかどうかを確認し、.class ファイルのタイムスタンプと .jsp ソース・ファイルのタイムスタンプを比較し、.class ファイルの方が古い場合、ページを再変換します。

タイムスタンプの比較が不要な場合（ソース・コードが変更されない一般的な配置環境の場合）、JSP の developer_mode フラグを false に設定して、タイムスタンプの比較を回避できます。デフォルトの設定は、true です。JServ 環境でこのフラグを設定する方法は、9-17 ページの「JServ での JSP パラメータの設定」を参照してください。

注意： この説明は、事前変換には該当しません。

HTTP セッションの使用の回避

JSP ページで HTTP セッションを使用する必要がない場合（基本的に、セッション属性を格納または取得する必要がない場合）、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" %>
```

これによって、セッションの作成または取得によるオーバヘッドがなくなるため、パフォーマンスが向上します。

デフォルトでは、サーブレットはセッションを使用しませんが、JSP ページはセッションを使用します。

CLASSPATH およびクラス・ローダーの問題

Oracle JSP コンテナは、Web サーバーの CLASSPATH とは異なる、独自の CLASSPATH を使用します。また、デフォルトで、独自のクラス・ローダーを使用して、この CLASSPATH からクラスをロードします。これには、重要なメリットおよびデメリットがあります。

JSP の CLASSPATH は、次の要素で構成されます。

- Oracle JSP のデフォルトの CLASSPATH
- JSP の classpath 構成パラメータに指定した追加の CLASSPATH

システム・クラス・ローダーではなく Oracle JSP クラス・ローダーでロードする必要があるクラスが存在する場合は、classpath 構成パラメータを使用するか、または Oracle JSP のデフォルトの CLASSPATH にクラスを配置します。詳細は、4-21 ページの「Oracle JSP クラス・ローダーのメリットおよびデメリット」を参照してください。

Oracle JSP のデフォルトの CLASSPATH

Oracle JSP は、必要なクラス（JavaBeans など）の .class ファイルおよび .jar ファイルの格納場所を設定するために、Web サーバー上に標準の格納場所を定義します。Oracle JSP コンテナは、Web サーバーの CLASSPATH 構成を使用せずに、これらの場所でファイルを検索します。

次に、これらの場所の例を示します。これらの場所は、アプリケーション・ルートからの相対パスです。

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

（WEB-INF ディレクトリは、JServ などの Servlet 2.0 環境には適していません。）

> **重要：** WEB-INF ディレクトリのクラスを、Oracle JSP クラス・ローダーではなく、システム・クラス・ローダーでロードする必要がある場合、これらのクラスを Web サーバーの CLASSPATH に配置します。システム・クラス・ローダーが優先されるため、両方の CLASSPATH に配置されているすべてのクラスは、常に、システム・クラス・ローダーによってロードされます。

_pages ディレクトリは、（JSP トランスレータによって出力された）変換およびコンパイル済の JSP ページがデフォルトで格納される場所です。

classes ディレクトリは、個別の Java .class ファイル用です。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、コードによって oracle.jsp.sample.lottery パッケージ内に存在するように定義されている、LottoBean という JavaBeans について考えてみます。Oracle JSP コンテナは、アプリケーション・ルートに関連した次の相対パスで LottoBean.class を検索します。

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

lib ディレクトリは、.jar ファイル用です。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、すべての .jar ファイルは、（サブディレクトリではなく）lib ディレクトリの直下に格納されます。

たとえば、LottoBean.class は、アプリケーション・ルートに関連した次の相対パスに存在する lottery.jar に格納される場合があります。

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

アプリケーション・ルート・ディレクトリは、（Web サーバーおよびサーブレット環境によって）次のいずれかの場所に、検索順に設定できます。

- アプリケーションがマップされる Web サーバー・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` ファイルが含まれるディレクトリ（該当する場合。通常、Servlet 2.0 環境）

注意：

- 一部の Web サーバー（特に、Servlet 2.0 仕様をサポートするサーバー）は、完全なアプリケーション・サポート（完全なサーブレット・コンテキスト機能など）を提供しません。この場合またはアプリケーション・マッピングが使用されない場合、デフォルトのアプリケーションはサーバー自体になり、アプリケーション・ルートは Web サーバーのドキュメント・ルートになります。
- 以前のサーブレット環境での `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可能な Oracle の拡張機能です。9-23 ページの「Oracle JSP アプリケーションおよび JServ でのセッション・サポート」を参照してください。

Oracle JSP の CLASSPATH 構成パラメータ

JSP の `classpath` 構成パラメータを使用して、Oracle JSP の CLASSPATH を追加します。

このパラメータの詳細は、9-7 ページの「Oracle JSP の構成パラメータ」を参照してください。JServ 環境でこのパラメータを設定する方法は、9-17 ページの「JServ での JSP パラメータの設定」を参照してください。

Oracle JSP クラス・ローダーのメリットおよびデメリット

Oracle JSP クラス・ローダーには、次のメリットおよびデメリットがあります。

- 他のクラス・ローダーによってロードされたクラスからの、Oracle JSP クラス・ローダーによってロードされたクラスへのアクセスの制限

  クラスが Oracle JSP クラス・ローダーによってロードされる場合、定義はそのクラス・ローダーのみに存在します。システム・クラス・ローダーまたはすべてのサーブレットを含む他のクラス・ローダーによってロードされたクラスでは、アクセスが制限されます。他のクラス・ローダーによってロードされたクラスは、Oracle JSP クラス・ローダーまたはこのクラス・ローダーに対するコール・メソッドによってロードされたクラスをキャストできません。この動作は状況によって、適切な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

  クラス・ファイルまたは JAR ファイルが最後にロードされた後に変更されている場合、Oracle JSP クラス・ローダーは、デフォルトで、Oracle JSP の CLASSPATH のクラスを自動的に再ロードします。たとえば、JSP ページの場合、動的な再変換の結果としてこのような状況が発生する場合があります。動的な再変換は、デフォルトでは、ページの `.jsp` ソース・ファイルのタイムスタンプが対応するページ実装 `.class` ファイルのタイムスタンプより新しい場合に実行されます。

  通常、この動作は、開発環境でのみメリットがあります。通常の配置環境では、ソース、クラスおよび JAR ファイルは変更されないため、このような変更の確認は効率的ではありません。

  詳細は、4-24 ページの「クラスの動的な再ロード」を参照してください。

通常、配置環境では、Oracle JSP の CLASSPATH を使用する必要がありません。デフォルトでは、`classpath` パラメータは空です。

Oracle JSP による実行時のページおよびクラスの再ロード

この項では、Oracle JSP コンテナが実行時にページの再変換、ページの再ロードおよびクラスの再ロードを実行する条件について説明します。

ページの動的な再変換

Web アプリケーションの実行時、デフォルトでは、ページのソースが変更されると、常に、Oracle JSP コンテナが JSP ページを自動的に再変換および再ロードします。

JSP コンテナは、Oracle JSP のメモリー内キャッシュに示されているページ実装クラス・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古いかどうかを確認します。

JSP の `developer_mode` フラグを `false` に設定すると、Oracle JSP コンテナが再変換のためにタイムスタンプを確認することによって発生するオーバーヘッドを回避できます。この操作は、ソースおよびクラス・ファイルが通常変更されない配置環境でメリットがあります。このフラグの詳細は、9-7 ページの「Oracle JSP の構成パラメータ」を参照してください。JServ 環境でこのフラグを設定する方法は、9-17 ページの「JServ での JSP パラメータの設定」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メモリー内の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・ファイルを削除しても、Oracle JSP コンテナは、関連付けられた JSP ページ・ソースを再変換しないことに注意してください。JSP コンテナは、JSP のページ・ソース・ファイルのタイムスタンプが変更された場合のみ、再変換を行います。
- キャッシュが消失すると、クラス・ファイルが再生成されます。サーバーの再起動後、またはアプリケーションの他のページの再変換後にページにリクエストが送信されると、このような状況が発生します。

ページの動的な再ロード

Oracle JSP コンテナは、次の状況で、JSP ページを自動的に再ロードします（生成されたページ実装クラスを再ロードします）。

- ページが再変換された場合

  （4-22 ページの「ページの動的な再変換」を参照してください。）

- ページによってコールされ、（システム・クラス・ローダーではなく）Oracle JSP クラス・ローダーによってロードされた Java クラスが変更された場合

  （4-24 ページの「クラスの動的な再ロード」を参照してください。）

- 同じアプリケーション内のページが再ロードされた場合

  JSP ページは、このページが実行される Web アプリケーション全体に関連付けられています（特定のアプリケーションに関連付けられていない JSP ページも、デフォルトのアプリケーションの一部であるとみなされます）。

  JSP ページが再ロードされると、常に、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：

- 静的にインクルードされたファイルが変更されるのみでは、Oracle JSP コンテナはページを再ロードしません。（`<%@ include ... %>` 構文を介して静的にインクルードされるファイルは、変換時にインクルードされます。）
- ページの再ロードとページの再変換は同じではありません。再ロードは、再変換を意味しません。

クラスの動的な再ロード

デフォルトでは、Oracle JSP コンテナは、Oracle JSP クラス・ローダーによってロードされた Java クラスを実行するリクエストをディスパッチする前に、クラス・ファイルが最初のロード後に変更されているかどうかを確認します。クラスが変更されている場合、Oracle JSP クラス・ローダーはそのクラスを再ロードします。

この動作は、次に示す Oracle JSP の CLASSPATH 内のクラスのみに適用されます。

- `/WEB-INF/lib` ディレクトリ（Servlet 2.2）の JAR ファイル
- `/WEB-INF/classes` ディレクトリ（Servlet 2.2）の `.class` ファイル
- Oracle JSP の `classpath` 構成パラメータを介して指定されたパス内のクラス
- `_pages` 出力ディレクトリに生成された `.class` ファイル

4-23 ページの「ページの動的な再ロード」で説明したとおり、クラスを再ロードすると、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：

- 動的に再ロードするクラスは、システムの CLASSPATH ではなく、JSP の CLASSPATH に存在する必要があります。システムの CLASSPATH にも存在する場合は、状況によって、システム・クラス・ローダーが優先される場合があります。この場合、JSP の自動再ロード機能が妨げられる可能性があります。
- クラスの動的な再ロードでは、CPU の使用率が大きくなる可能性があります。JSP の `developer_mode` パラメータを `false` に設定して、この機能を無効にできます。この操作は、クラスが変更されない配置環境に適しています。

`classpath` および `developer_mode` 構成パラメータ、および JServ 環境でこれらのパラメータを設定する方法は、9-7 ページの「Oracle JSP の構成パラメータ」および 9-17 ページの「JServ での JSP パラメータの設定」を参照してください。

5

Oracle 固有のプログラミング拡張機能

この章では、他の JSP 環境に移植不可能な、Oracle によって提供される JSP の拡張機能について説明します。この拡張機能には、Oracle の `JspScopeListener` メカニズムを使用したイベント処理、および SQLJ（Java コードに SQL 文を直接埋め込むための標準構文）のサポートが含まれます。この章の内容は次のとおりです。

注意：

- Servlet 2.0 環境の場合、Oracle JSP コンテナは、Web アプリケーションのフレーワークをサポートするための `globals.jsa` と呼ばれるメカニズムを介して、移植不可能な拡張機能をサポートします。このメカニズムについては、9-23 ページの「Oracle JSP アプリケーションおよび JServ でのセッション・サポート」を参照してください。
- Oracle JSP コンテナには、8-5 ページの「Oracle JSP によるマルチバイト・パラメータ・エンコーディングの拡張サポート」で説明する、拡張された（移植不可能な）グローバリゼーション・サポートも含まれています。

JspScopeListener を使用した Oracle JSP によるイベント処理

標準のサーブレットおよび JSP テクノロジでは、セッション・ベースのイベントのみがサポートされます。Oracle JSP コンテナは、oracle.jsp.event パッケージの JspScopeListener インタフェースおよび JspScopeEvent クラスを介して、このサポートを拡張します。このメカニズムは、次の 4 つの標準 JSP スコープをサポートして、JSP アプリケーションで使用されるすべての Java オブジェクトのイベント処理を行います。

- page
- request
- session
- application

アプリケーションで使用される Java オブジェクトに対して、適切なクラスの JspScopeListener インタフェースを実装し、そのクラスのオブジェクトを jsp:useBean などのタグを使用して JSP スコープに関連付けます。

スコープの終わりに達すると、そのスコープに関連付けられた JspScopeListener を実装するオブジェクトに通知されます。Oracle JSP コンテナは、JspScopeListener インタフェースに指定された outOfScope() メソッドを介して、これらのオブジェクトに JspScopeEvent インスタンスを送信することによって、この操作を実行します。

JspScopeEvent オブジェクトのプロパティには、次の項目が含まれます。

- スコープの終わり（PAGE_SCOPE、REQUEST_SCOPE、SESSION_SCOPE または APPLICATION_SCOPE のいずれかの定数）
- スコープのオブジェクト用リポジトリであるコンテナ・オブジェクト（page、request、session または application のいずれかの暗黙的なオブジェクト）
- 通知され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名）
- JSP の暗黙的な application オブジェクト

Oracle JSP のイベント・リスナー・メカニズムは、エラー条件に関係なく、page または request スコープが設定されたオブジェクト・リソースを常に解放する必要がある場合、大きなメリットがあります。このメカニズムによって、開発者は、Java の try/catch/finally ブロックでページ実装を囲む必要がなくなります。

Oracle JSP による Oracle SQLJ のサポート

SQLJ は、静的 SQL 命令を Java コードに直接埋め込むための標準構文です。これによって、データベース・アクセスのプログラミングが大幅に簡略化されます。Oracle JSP コンテナおよびその JSP トランスレータは Oracle SQLJ をサポートするため、JSP 文で SQLJ 構文を使用できます。SQLJ 文は、`#sql` トークンによって示されます。

Oracle SQLJ のプログラミング機能、構文およびコマンドライン・オプションの詳細は、『Oracle9*i* 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 コードの例

次に、SQLJ の JSP ページの例を示します。(page ディレクティブでは、通常、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");
```

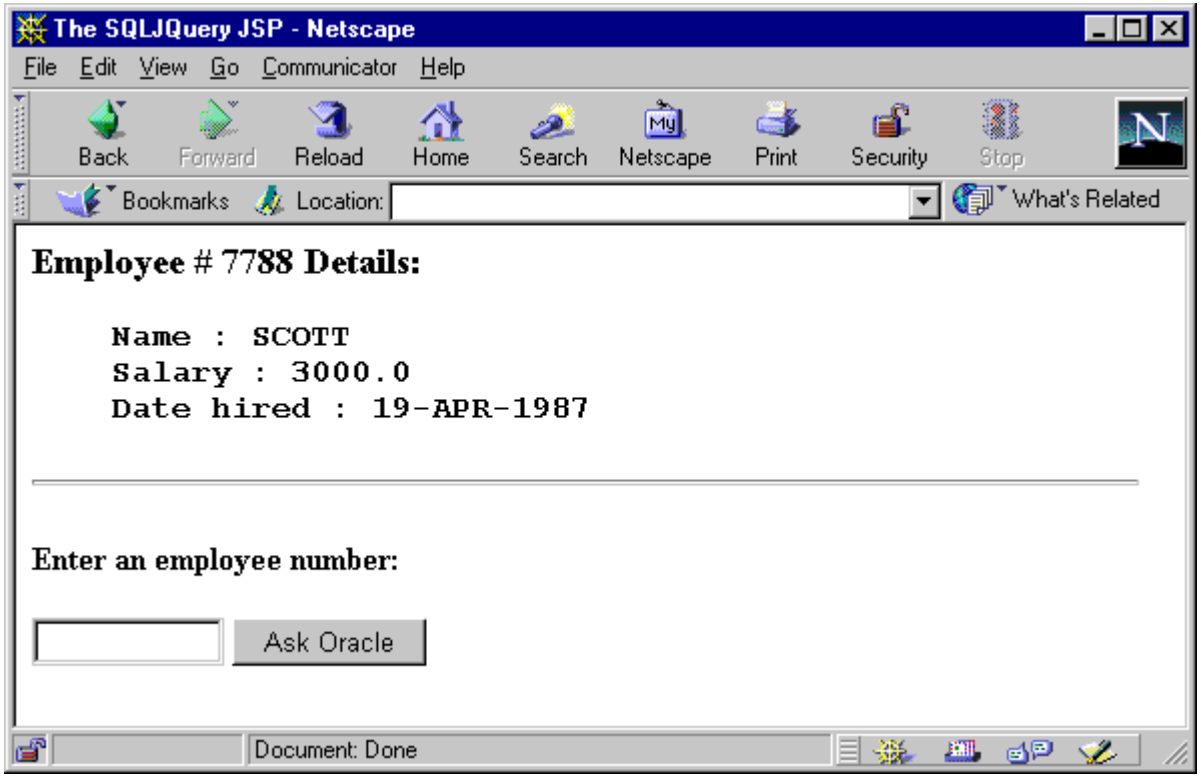
```
      sb.append("Name : " + ename + "\n");
      sb.append("Salary : " + sal + "\n");
      sb.append("Date hired : " + hireDate);
      sb.append("</PRE></B></BIG></BLOCKQUOTE>");
      } catch (java.sql.SQLException e) {
        sb.append("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
      } finally {
      if (dctx!= null) dctx.close();
      }
  return sb.toString();
}

%>
```

この例では、JDBC OCI ドライバを使用します。そのため、Oracle クライアントをインストールする必要があります。接続の確立に使用する `Oracle` クラスは、Oracle SQLJ に付属しています。

注意：

- 同じ JVM で JSP ページが複数回起動される場合、デフォルトの接続コンテキストではなく、常に、明示的な接続コンテキスト（前述の例の `dctx` など）を使用することをお薦めします（`dctx` はローカル・メソッド変数であることに注意してください）。
- Oracle JSP コンテナには、Oracle SQLJ リリース 8.1.6.1 以上が必要です。
- 将来のリリース（および Oracle9*i* Application Server リリース 2（9.0.2）では、Oracle JSP コンテナが、`page` ディレクティブで、JSP 変換時に Oracle SQLJ トランスレータをトリガーするための `language="sqlj"` をサポートします。将来の互換性のために、このディレクティブを使用してプログラミングすることをお薦めします。

この例で使用されるスキーマに従業員番号 7788 を入力すると、結果の出力は次のとおりです。



SQLJ トランスレータのトリガー

JSP ソース・ファイルに `.sqljsp` というファイル名の拡張子を使用することによって、Oracle JSP 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。

これによって、JSP トランスレータは、`.java` ファイルではなく `.sqlj` ファイルを生成します。次に、Oracle SQLJ トランスレータが起動されて、`.sqlj` ファイルを `.java` ファイルに変換します。

SQLJ を使用すると、追加の出力ファイルが生成されます。6-6 ページの「生成されるファイルおよび格納場所（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 JAR または ZIP ファイルをインストールし、これらのファイルを CLASSPATH に追加する必要があります。9-2 ページの「Oracle JSP 用の必須ファイルおよびオプションのファイル」を参照してください。
- 同じアプリケーション内の `.jsp` ファイルおよび `.sqljsp` ファイルには同じベース・ファイル名を使用しないでください。同じベース・ファイル名を使用すると、同じクラス名および `.java` ファイル名が生成されるためです。

Oracle SQLJ オプションの設定

SQLJ JSP ページを実行または事前変換する場合は、任意の Oracle SQLJ オプションを設定できます。これは、次に示すとおり、オンデマンド変換および事前変換の両方の場合にあてはまります。

- オンデマンド変換では、JSP 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可能ファイルを指定できるのみでなく、SQLJ のコマンドライン・オプションを設定できます。

  詳細は、9-7 ページの「Oracle JSP の構成パラメータ」の `sqljcmd` の説明を参照してください。JServ 環境で構成パラメータを設定する方法は、9-17 ページの「JServ での JSP パラメータの設定」を参照してください。

- `ojspc` 事前変換ツールを使用した事前変換では、`ojspc -S` オプションを使用します。このオプションを使用すると、SQLJ のコマンドライン・オプションを設定できます。

  詳細は、6-17 ページの「ojspc のコマンドライン構文」および 6-18 ページの「ojspc のオプションの説明」を参照してください。

6

JSP による変換および配置

この章では、Oracle JSP トランスレータの操作について説明し、次に ojspc ユーティリティおよび事前変換が有効な状況について説明します。その後、JSP による配置について、その他多くの一般的な考慮点を示します。

この章の内容は次のとおりです。

Oracle JSP トランスレータの機能

JSP トランスレータは、JSP ページ実装クラス用の標準 Java コードを生成します。基本的に、このクラスは、JSP 機能用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、主にオンデマンド変換に使用する場合の動作について、Oracle JSP トランスレータの一般的な機能を説明します。この項の内容は次のとおりです。

重要： パッケージとクラスの名前生成、ファイルとディレクトリの名前生成、出力ファイルの格納場所、および生成コードに関する実装の詳細は、この項では説明を目的として示されています。この項で示す詳細は、Oracle JSP リリース 1.1.x.x のみに適用されます。この詳細は、リリースによって異なります。

生成コードの機能

この項では、JSP ソース（`.jsp` および `.sqljsp` ファイル）の変換時に Oracle JSP トランスレータが作成するページ実装クラス・コードの一般的な機能について説明します。

ページ実装クラス・コードの機能

Oracle JSP トランスレータは、サーブレット・コードのページ実装クラスへの生成時に、標準プログラミングによるオーバーヘッドの一部を自動的に処理します。オンデマンド変換モデルおよび事前変換モデルの両方の場合に、生成コードには次の機能が自動的に含まれます。

- 標準の `javax.servlet.jsp.HttpJspPage` インタフェースを実装する、Oracle JSP コンテナに付属のラッパー・クラス（`oracle.jsp.runtime.HttpJsp`）を拡張します。（このインタフェースは、標準の `javax.servlet.Servlet` インタフェースを拡張する、より一般的な `javax.servlet.jsp.JspPage` インタフェースを拡張します。）
- `HttpJspPage` によって指定された `_jspService()` メソッドを実装します。このメソッド（通常、サービス・メソッドと呼ばれる）は、ページ実装クラスの主要なメソッドです。JSP ページの Java スクリプトレットおよび式から生成されたすべてのコードが、このメソッドの実装に組み込まれます。
- JSP ソース・コードで `session=false` が設定されていないかぎり（`page` ディレクティブで設定可能）、HTTP セッションをリクエストするコードをインクルードします。

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・メソッドである _jspService() には、JSP ページの静的テキストを出力するための出力コマンド（暗黙的な out オブジェクトに対する out.print() のコール）が含まれます。ただし、Oracle JSP トランスレータは、静的テキストをページ実装クラス内のインナー・クラスに配置します。サービス・メソッドの out.print() 文は、インナー・クラスの属性を参照して、テキストを出力します。

このインナー・クラス実装では、ページの変換およびコンパイル時に、追加の .class ファイルが生成されます。クライアント側で事前変換を行う場合は、配置の必要がある .class ファイルが増えることに注意してください。

インナー・クラスの名前は、常に、.jsp ファイルまたは .sqljsp ファイルのベース名に基づいて決定されます。たとえば、mypage.jsp の場合、インナー・クラス（およびその .class ファイル）の名前には、常に「mypage」が含まれます。

注意： Oracle JSP トランスレータは、オプションで、静的テキストを Java リソース・ファイルに配置できます。この機能は、ページに大量の静的テキストが含まれている場合に有効です（4-10 ページの「JSP ページに大量の静的コンテンツが含まれている場合の対策」を参照）。オンデマンド変換の場合は JSP の external_resource 構成パラメータを介して、また、事前変換の場合は ojspc -extres オプションを介して、この機能を要求できます。

静的テキストがリソース・ファイルに配置されている場合も、インナー・クラスが作成されるため、その .class ファイルを配置する必要があります。（この操作は、クライアント側で事前変換を行う場合のみ、注意が必要です。）

出力名に対する一般規則

Oracle JSP トランスレータは、一貫した一連の規則に従って、出力クラス、パッケージ、ファイルおよびディレクトリの名前を生成します。ただし、この一連の規則および実装に関するこの他の詳細は、リリースによって異なります。

JSP ページのベース名に特殊文字が含まれないかぎり、出力クラスおよびファイルの名前に、JSP ページのベース名がそのまま含まれます。この動作は、すべてのリリースに共通しています。たとえば、MyPage123.jsp を変換すると、ページ実装クラスの名前、Java ソース・ファイルの名前およびクラス・ファイルの名前に、常に「MyPage123」という文字列が含まれます。

Oracle JSP リリース 1.1.x.x（およびそれ以前のリリースの一部）では、ベース名の前にアンダースコア（「_」）が付きます。MyPage123.jsp を変換すると、ページ実装クラスは _MyPage123、このクラスが存在するリソース・ファイルは _MyPage123.java になり、_MyPage123.class にコンパイルされます。

同様に、Java パッケージ名の作成でパス名が使用される場合、パスの各構成要素の前にアンダースコアが付きます。たとえば、/jspdir/myapp/MyPage123.jsp を変換する場合、クラスは _MyPage123 となり、このクラスは次のパッケージに配置されます。

```
_jspdir._myapp
```

.java および .class 出力ファイル用のディレクトリの作成ではパッケージ名が使用されるため、出力先ディレクトリ名にもアンダースコアが付きます。たとえば、htdocs/test ディレクトリの JSP ページを変換する場合、Oracle JSP トランスレータは、デフォルトで、ページ実装クラスのソース用に htdocs/_pages/_test ディレクトリを生成します。

注意： 6-6 ページの「生成されるファイルおよび格納場所（オンデマンド変換）」に示すとおり、すべての出力ディレクトリは、デフォルトで、標準の _pages ディレクトリの下に作成されます。ただし、この動作は、9-7 ページの「Oracle JSP の構成パラメータ」で説明する page_repository_root 構成パラメータ、または 6-18 ページの「ojspc のオプションの説明」で説明する ojspc -d および -srcdir オプションを介して変更できます。

JSP ページの名前またはパス名に特殊文字が含まれている場合、Oracle JSP トランスレータは、出力されるクラス、パッケージおよびファイルの名前に Java で無効な文字が含まれないことを保証するための手順を実行します。たとえば、My-name_foo12.jsp を変換すると、クラス名は _My_2d_name__foo12、このクラスが存在するリソース・ファイルは _My_2d_name__foo12.java になります。ハイフン（-）は、英数字の文字列に変換されます。（「foo12」の前に追加のアンダースコアが挿入されます。）この場合、出力されるクラスおよびファイル名に、JSP ページ名を構成する英数字が含まれることのみが保証されます。たとえば、「My」、「name」または「foo12」を使用して検索できます。

これらの規則については、この項およびこの章の後半の例で説明します。

生成されるパッケージおよびクラスの名前（オンデマンド変換）

Sun 社の JavaServer Pages 仕様バージョン 1.1 では、JSP テキストの解析および変換のための統一されたプロセスが定義されていますが、生成されるクラスの名前生成方法（JSP 実装によって異なる）については記載されていません。

この項では、Oracle JSP コンテナが変換時にコードを生成する場合の、パッケージおよびクラスの名前の生成方法を説明します。

注意： Oracle JSP コンテナが、出力されるクラス、パッケージ、ファイルおよびスキーマ・パスの名前の生成に使用する一般規則の詳細は、6-3 ページの「出力名に対する一般規則」を参照してください。

パッケージ名の生成

オンデマンド変換の場合、ユーザーが JSP ページのリクエスト時に指定する 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・ルートからの相対パス）によって、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 のパス内の各ディレクトリは、パッケージ階層のレベルを表します。

ただし、生成されるパッケージ名は、URL で使用されている大 / 小文字に関係なく、常に小文字であることに注意してください。

例として、次の URL について考えてみます。

```
http://host[:port]/HR/expenses/login.jsp
```

Oracle JSP リリース 1.1.x.x では、これによって、パッケージが次のとおり生成コードに指定されます（将来のリリースでは、実装の詳細が変更されます）。

```
package _hr._expenses;
```

JSP ページがドキュメント・ルート・ディレクトリまたはアプリケーション・ルート・ディレクトリに存在し、URL が次のとおりである場合、パッケージ名は生成されません。

```
http://host[:port]/login.jsp
```

クラス名の生成

`.jsp` ファイル（または `.sqljsp` ファイル）のベース名によって、生成コードのクラス名が決定されます。

次の URL について考えてみます。

```
http://host[:port]/HR/expenses/UserLogin.jsp
```

Oracle JSP リリース 1.1.x.x では、これによって、次のクラス名が生成コードに生成されます（将来のリリースでは、実装の詳細が変更されます）。

```
public class _UserLogin extends ...
```

エンド・ユーザーが URL に入力する文字（大 / 小文字）を、実際の `.jsp` または `.sqljsp` ファイル名に一致させる必要があります。たとえば、`UserLogin.jsp` または `userlogin.jsp` が実際のファイル名の場合、これらのファイル名は指定できますが、実際のファイル名が `UserLogin.jsp` の場合、`userlogin.jsp` は指定できません。

Oracle JSP リリース 1.1.x.x では、ファイル名の大 / 小文字に基づいて、トランスレータによってクラス名の大 / 小文字が決定されます。次に例を示します。

- ファイル名が UserLogin.jsp の場合、クラス名は _UserLogin になります。
- ファイル名が Userlogin.jsp の場合、クラス名は _Userlogin になります。
- ファイル名が userlogin.jsp の場合、クラス名は _userlogin になります。

クラス名で大 / 小文字を区別する必要がある場合は、.jsp ファイルまたは .sqljsp ファイルの名前に適切な大 / 小文字を使用する必要があります。ただし、エンド・ユーザーはページ実装クラスを参照できないため、通常、クラス名の大 / 小文字は問題になりません。

生成されるファイルおよび格納場所（オンデマンド変換）

この項では、Oracle JSP トランスレータによって生成されるファイル、およびそれらのファイルが配置される場所について説明します。事前変換の場合、ojspc によってファイルが異なる場所に配置されます。また、このユーティリティには、一連の固有の関連オプションが存在します。6-24 ページの「ojspc の出力ファイル、格納場所および関連オプションの概要」を参照してください。

次の項では、いくつかの JSP 構成パラメータについて説明します。JSP 構成パラメータの詳細および JServ 環境での設定方法については、9-7 ページの「Oracle JSP の構成パラメータ」および 9-17 ページの「JServ での JSP パラメータの設定」を参照してください。

注意： Oracle JSP コンテナが出力クラス名の生成に使用する一般規則の詳細は、6-3 ページの「出力名に対する一般規則」を参照してください。

Oracle JSP コンテナによって生成されるファイル

この項では、Oracle JSP トランスレータによって生成されるファイルを列挙し、通常の JSP ページ（.jsp ファイル）および SQLJ JSP ページ（.sqljsp ファイル）の両方について説明します。ファイル名の例としては、Foo.jsp または Foo.sqljsp というファイルが変換されるとします。

ソース・ファイル

- ページが SQLJ JSP ページの場合は、Oracle JSP トランスレータによって .sqlj ファイル（_Foo.sqlj など）が作成されます。
- 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（_Foo.java など）が作成されます。Oracle JSP トランスレータが .jsp ファイルから直接作成するか、またはページが SQLJ JSP ページの場合は、SQLJ トランスレータが .sqlj ファイルから作成します。（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、JSP の sqljcmd 構成パラメータを使用して、別のトランスレータまたは別のリリースの Oracle SQLJ トランスレータを指定できます。）

バイナリ・ファイル

- SQLJ JSP ページの場合、ISO 規格の SQLJ コード生成（SQLJ の -codegen=iso 設定）を使用すると、SQLJ 変換時に、SQLJ プロファイル用の 1 つ以上のバイナリ・ファイルが作成されます。デフォルトでは、これらのファイルは Java リソース・ファイル .ser ですが、（JSP の sqljcmd 構成パラメータを介して）SQLJ の -ser2class オプションを有効にすると、.class ファイルになります。リソース・ファイルまたは .class ファイルの名前には「Foo」が含まれます。

  注意： SQLJ で Oracle 固有のコード生成（-codegen=oracle）を使用すると、プロファイルが作成されません。これは、Oracle9*i* リリース 2（9.2）での SQLJ のデフォルトのモードです。

- Java コンパイラによって、ページ実装クラス用の .class ファイルが作成されます。（Java コンパイラは、デフォルトでは javac ですが、JSP の javaccmd 構成パラメータを使用して別の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・クラス用の追加の .class ファイルが作成されます。このファイルの名前には「Foo」が含まれます。
- JSP の external_resource 構成パラメータが有効にされている場合は、オプションで、静的なページ・コンテンツ用の Java リソース・ファイル .res（_Foo.res など）が作成されます。

  注意： ページ実装クラスの生成ファイルの正確な名前は将来のリリースで変更される可能性がありますが、一般形式は変更されません。名前には、常に、ベース名（前述の例の「Foo」など）が含まれますが、それ以外の文字が含まれる場合があります。

Oracle JSP トランスレータによる出力ファイルの格納場所

Oracle JSP コンテナは、Web サーバーのドキュメント・リポジトリを使用して、変換された JSP ページを生成またはロードします。

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（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・ディレクトリになります。

JSP の page_repository_root 構成パラメータを介して、別のルート・ディレクトリを指定できます。

Oracle JSP リリース 1.1.x.x では、生成ファイルは次の場所に配置されます（将来のリリースでは、実装の詳細が変更される可能性があります）。

- .jsp（または .sqljsp）ファイルがルート・ディレクトリの直下に存在する場合、Oracle JSP コンテナは、生成ファイルをルート・ディレクトリの直下にあるデフォルトの _pages サブディレクトリに配置します。
- .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存在する場合、_pages サブディレクトリの下に、生成ファイル用のパラレル・ディレクトリ構造が作成されます。_pages ディレクトリのサブディレクトリ名は、ルート・ディレクトリのサブディレクトリ名に基づいて決定されます。

  たとえば、ドキュメント・ルート・ディレクトリが htdocs の JServ 環境について考えてみます。.jsp ファイルが次のディレクトリに存在するとします。

  ```
  htdocs/subdir/test
  ```

  生成ファイルは、次のディレクトリに配置されます。

  ```
  htdocs/_pages/_subdir/_test
  ```

ページ実装クラスのソースのサンプル

この項では、例を使用して、前述の項で示した情報について説明します。

次の場合について考えてみます。

- JSP ページ・コードが hello.jsp ファイルに存在します。
- ページが 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存在します。

  ```
  htdocs/test
  ```

重要： この項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。将来のリリースでは、仕様の変更、または Oracle による仕様に含まれない要素の実装方法の変更によって、詳細が変更される可能性があります。

ページのソースのサンプル : hello.jsp

次に、hello.jsp に含まれる JSP コードを示します。

```
<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>
```

生成されるパッケージおよびクラスのサンプル

hello.jsp はルート・ディレクトリ（htdocs）の test サブディレクトリに存在するため、Oracle JSP リリース 1.1.x.x では、ページ実装コードに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（大 / 小文字の区別する）によって決定されるため、ページ実装コードに次のクラス定義が生成されます。

```
public class _hello extends oracle.jsp.runtime.HttpJsp
{
   ...
}
```

（エンド・ユーザーはページ実装クラスを参照できないため、Java クラス名が Java の大文字化規則に従っていないことは、通常、問題にはなりません。）

生成ファイルのサンプル

hello.jsp が次の場所に存在するとします。

```
htdocs/test/hello.jsp
```

この場合、Oracle JSP リリース 1.1.x.x では、次のとおり出力ファイルが生成されます（それぞれ、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、Oracle JSP リリース 1.1.x.x 実装に基づいています。将来のリリースでは、詳細が変更される可能性があります。ただし、すべてのファイル名に、常に、「hello」というベース名が含まれます。

ページ実装コードのサンプル : _hello.java

次に、Oracle JSP リリース 1.1.x.x によって生成されるページ実装クラスの Java コード（_hello.java）を示します。

```
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();
    }
  }
}
```

JSP の事前変換および ojspc ユーティリティ

この項では、Oracle9*i* に付属の事前変換用 ojspc ユーティリティについて説明します。この項の内容は次のとおりです。

事前変換用 ojspc の一般的な使用方法

ojspc は、すべての環境で JSP ページの事前変換に使用できます。このユーティリティは、エンド・ユーザーが初めてページを実行する場合、変換オーバーヘッドの低減に有効な場合があります。

ターゲット環境以外の環境で事前変換を行う場合は、ojspc -d オプションを指定して、生成されたバイナリ・ファイルが配置される適切なベース・ディレクトリを設定します。

例として、次の JSP ソース・ファイルを持つ JServ 環境について考えてみます。

```
htdocs/test/foo.jsp
```

ユーザーは、次の URL を使用してこれを起動できます。

```
http://host[:port]/test/foo.jsp
```

実行時のオンデマンド変換では、Oracle JSP トランスレータは、デフォルトのベース・ディレクトリ htdocs/_pages を使用して、生成されたバイナリ・ファイルを配置します。そのため、事前変換を行う場合は、次の例に示すとおり、htdocs/_pages をバイナリ出力のベース・ディレクトリに設定する必要があります（% は UNIX プロンプトであるとします）。

```
% cd htdocs
% ojspc -d _pages test/foo.jsp
```

前述の URL によって、test/foo.jsp というアプリケーション相対パスが指定されます。そのため、Oracle JSP コンテナは、実行時に、デフォルトの htdocs/_pages ディレクトリの下にある _test サブディレクトリでバイナリ・ファイルを検索します。このサブディレクトリは、ojspc が前述の例のとおり実行されている場合、その ojspc によって自動的に生成されます。事前変換後にソース・ファイルが変更されていない場合、Oracle JSP コンテナは、実行時に事前変換されたバイナリ・ファイルを検索するため、変換を実行する必要がありません。（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でない場合、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い場合にページが事前変換されます。）

注意： 出力ディレクトリ名でのアンダースコア（「_」）の使用方法（前述の例では `_test`）などの Oracle JSP 実装の詳細は、リリースによって異なります。このマニュアルの説明は、Oracle JSP リリース 1.1.x.x を対象としています。

ojspc 事前変換ツールの詳細

この項の内容は次のとおりです。

- ojspc の機能の概要
- ojspc のオプションの一覧表
- 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、格納場所および関連オプションの概要

注意： JSP ページの事前変換に使用する `ojspc` には、中間層環境での使用など、他の使用例もあります。6-13 ページの「事前変換用 ojspc の一般的な使用方法」を参照してください。

ojspc の機能の概要

次に、簡単な（SQLJ JSP ではなく）JSP ページに対する、`ojspc` のデフォルトの機能を示します。

- `.jsp` ファイルを引数に取ります。
- Oracle JSP トランスレータを起動して、`.jsp` ファイルを Java ページ実装クラス・コードに変換します。これによって、`.java` ファイルが作成されます。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ます。これによって、2 つの `.class` ファイル（ページ実装クラス用のファイルおよびインナー・クラス用のファイル）が作成されます。

次に、SQLJ JSP ページに対する ojspc のデフォルトの機能を示します。

- .jsp ファイルではなく、.sqljsp ファイルを引数に取ります。
- Oracle JSP トランスレータを起動して、.sqljsp ファイルをページ実装クラス（およびインナー・クラス）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て、.sqlj ファイルを変換します。これによって、ページ実装クラス（およびインナー・クラス）用の .java ファイルが作成されます。また、ISO 規格の SQLJ コード生成（SQLJ で -codegen=iso に設定）を使用すると、SQLJ の「プロファイル」ファイル（デフォルトでは Java リソース・ファイル .ser）が作成されます。

  注意： Oracle 固有の SQLJ コード生成（SQLJ で -codegen=oracle に設定）を使用すると、プロファイルは作成されません。これは、Oracle9*i* リリース 2（9.2）のデフォルトのモードです。

  SQLJ のプロファイルおよび Oracle 固有のコード生成の詳細は、『Oracle9*i* 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ます。これによって、2 つの .class ファイル（ページ実装クラス用のファイルおよびインナー・クラス用のファイル）が作成されます。

状況（次の -extres オプションの説明を参照）によっては、ojspc の設定によって、Oracle JSP トランスレータが、静的ページ・コンテンツをページ実装クラスのインナー・クラスに挿入するのではなく、これらのコンテンツ用の Java リソース・ファイル .res を作成します。ただし、この場合もインナー・クラスが作成されるため、ページ実装クラスを使用して配置する必要があります。

ojspc によって Oracle JSP トランスレータが起動されるため、通常、ojspc の出力規則は、Oracle JSP コンテナの場合と同様になります（該当する場合）。生成コードの機能、出力名の一般規則、生成されるパッケージおよびクラスの名前、生成ファイル、格納場所などの Oracle JSP トランスレータによる出力の詳細は、6-2 ページの「Oracle JSP トランスレータの機能」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のユーティリティです。

ojspc のオプションの一覧表

表 6-1 に、ojspc 事前変換ユーティリティがサポートするオプションを示します。これらのオプションの詳細は、6-18 ページの「ojspc のオプションの説明」を参照してください。

2 列目には、オンデマンド変換環境（JServ など）の場合の同等の JSP 構成パラメータまたは関連する JSP 構成パラメータを示します。

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 JSP 構成パラメータ	説明	デフォルト
-addclasspath	classpath（関連性はあるが、異なる機能）	javac 用の追加の CLASSPATH エントリ	空（追加のパス・エントリはなし）
-appRoot	なし	アプリケーションに相対的な、ページからの静的 include ディレクティブに対するアプリケーション・ルート・ディレクトリ	カレント・ディレクトリ
-debug	emit_debuginfo	デバッグのために、元の .jsp ファイルへの行マッピングを生成するように ojspc に指示するブール値	false
-d	page_repository_root	ojspc が、生成されたバイナリ・ファイル（.class およびリソース）を配置する場所	カレント・ディレクトリ
-extend	なし	生成されたページ実装クラスが拡張するクラス	空
-extres	external_resource	.jsp ファイルから静的テキスト用の外部リソース・ファイルを生成するように ojspc に指示するブール値	false
-implement	なし	生成されたページ実装クラスが実装するインタフェース	空
-noCompile	javaccmd	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 ojspc に指示するブール値	false
-packageName	なし	生成されたページ実装クラスのパッケージ名	空（.jsp ファイルの各格納場所に従ってパッケージ名が生成される）

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 JSP 構成パラメータ	説明	デフォルト
-S-<sqlj option>	sqljcmd	-S という接頭辞とその後に続く Oracle SQLJ オプション（.sqljsp ファイルの場合）	空
-srcdir	page_repository_root	ojspc が、生成されたソース・ファイル（.java および .sqlj）を配置する場所	カレント・ディレクトリ
-verbose	なし	実行時にステータス情報を出力するように ojspc に指示するブール値	false
-version	なし	Oracle JSP のバージョン番号を表示するように ojspc に指示するブール値	false

ojspc のコマンドライン構文

次に、ojspc の一般的なコマンドライン構文を示します（% は UNIX プロンプトであるとします）。

```
% ojspc [option_settings] file_list
```

file_list には、.jsp ファイルまたは .sqljsp ファイルを指定できます。

構文の注意事項は次のとおりです。

- 複数の .jsp ファイルが変換される場合、これらのファイルで（デフォルトで、または page ディレクティブの contentType 設定を介して）同じキャラクタ・セットが使用される必要があります。
- *file_list* では、ファイル名とファイル名の間に空白を入れます。
- オプション・リストでは、オプション名とオプション値の間にセパレータとして空白を入れます。
- オプション名では大 / 小文字が区別されませんが、オプション値（パッケージ名、ディレクトリ・パス、クラス名、インタフェース名など）では大 / 小文字が区別されます。
- （デフォルトでは無効にされている）ブール値オプションを有効にするには、コマンドラインでオプション名を入力します。たとえば、-extres true ではなく、-extres と入力します。

次に例を示します。

```
% ojspc -d /myapp/mybindir -srcdir /myapp/mysrcdir -extres MyPage.sqljsp MyPage2.jsp
```

ojspc のオプションの説明

この項では、ojspc オプションの詳細を説明します。

-addclasspath（完全修飾されたパス。ojspc のデフォルトは空。）

このオプションを使用すると、javac に追加の CLASSPATH エントリ（生成されたページ実装クラスのソースのコンパイル時に使用）を指定できます。このオプションを使用しない場合、javac はシステムの CLASSPATH のみを使用します。

注意：

- オンデマンド変換の場合、JSP の classpath 構成パラメータが関連機能（同等機能ではない）を提供します。9-7 ページの「Oracle JSP の構成パラメータ」を参照してください。
- SQLJ トランスレータも、SQLJ JSP ページに対して -addclasspath の設定を使用します。

-appRoot（完全修飾されたパス。ojspc のデフォルトはカレント・ディレクトリ。）

このオプションを使用すると、アプリケーション・ルート・ディレクトリを指定できます。デフォルトは、ojspc が実行されているカレント・ディレクトリです。

指定されたアプリケーション・ルート・ディレクトリ・パスは、次のとおり使用されます。

- 変換されるページの静的 include ディレクティブに対して使用されます。指定されたディレクトリ・パスは、変換されるページの include ディレクティブのアプリケーション相対（またはコンテキスト相対）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。パッケージは、変換されるファイルのアプリケーション・ルート・ディレクトリからの相対パスに基づいて決定されます。次に、パッケージによって、出力ファイルの配置場所が決定されます（6-24 ページの「ojspc の出力ファイル、格納場所および関連オプションの概要」を参照）。

このオプションは、たとえば、他のディレクトリから ojspc を実行中、インクルード対象のファイルを検索する場合に必要です。

次の例について考えてみます。

- 次のファイルを変換します。

  ```
  /abc/def/ghi/test.jsp
  ```

- 次のとおり、カレント・ディレクトリ /abc から ojspc を実行します（% は UNIX プロンプトであるとします）。

```
% cd /abc
% ojspc def/ghi/test.jsp
```

- test.jsp ページには、次の include ディレクティブが含まれています。

```
<%@ include file="/test2.jsp" %>
```

- test2.jsp ページは、次のとおり /abc ディレクトリに存在します。

```
/abc/test2.jsp
```

この場合、デフォルトのアプリケーション・ルート設定がカレント・ディレクトリ（/abc ディレクトリ）であるため、-appRoot を設定する必要はありません。include ディレクティブではアプリケーションに相対的な /test2.jsp 構文（「/」で始まることに注意）が使用されるため、インクルード対象のページは /abc/test2.jsp として検出されます。

この場合、パッケージは、ojspc が実行されているカレント・ディレクトリからの test.jsp の相対パスに基づいて、_def._ghi になります（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）。出力ファイルも、同様に配置されます。

ただし、他のディレクトリ（/home/mydir と想定します）から ojspc を実行する場合は、次の例に示すとおり -appRoot を設定する必要があります。

```
% cd /home/mydir
% ojspc -appRoot /abc /abc/def/ghi/test.jsp
```

この場合も、パッケージは、指定されたアプリケーション・ルート・ディレクトリからの test.jsp の相対パスに基づいて、_def._ghi になります。

注意： 通常、指定されたアプリケーション・ルート・ディレクトリは、変換された JSP ページが存在するディレクトリよりいくつか上のレベルに存在する親ディレクトリです。

-d（完全修飾されたパス。ojspc のデフォルトはカレント・ディレクトリ。）

このオプションを使用すると、ojspc が、生成されたバイナリ・ファイル（.class ファイルおよび Java リソース・ファイル）を配置する場所のベース・ディレクトリを指定できます（-extres オプションによって静的コンテンツに生成された .res ファイルは、SQLJ トランスレータによって SQLJ JSP ページに生成された .ser プロファイル・ファイルと同様に、Java リソース・ファイルです）。

指定されたパスは、（アプリケーション相対パスまたはページ相対パスではなく）単にファイル・システムのパスとして使用されます。

指定されたディレクトリ下のサブディレクトリは、必要に応じて、パッケージによって自動的に作成されます。詳細は、6-24 ページの「ojspc の出力ファイル、格納場所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 が実行されているカレント・ディレクトリ）が使用されます。

このオプションを使用して、生成されたバイナリ・ファイルを未使用のディレクトリに配置し、作成されたファイルが簡単に識別できるようにすることをお薦めします。

注意：

- ディレクトリ名に空白を使用できる Windows NT などの環境では、ディレクトリ名を引用符で囲みます。
- オンデマンド変換の場合、JSP の page_repository_root 構成パラメータが関連機能を提供します。9-7 ページの「Oracle JSP の構成パラメータ」を参照してください。

-debug（ブール値。ojspc のデフォルトは false。）

このフラグを有効にすると、デバッグのために元の .jsp ファイルへの行マッピングを生成するように ojspc に指示できます。このフラグを有効にしない場合、行は生成されたページ実装クラスに行マップされます。

このオプションは、Oracle9*i* JDeveloper を使用する場合など、ソースレベルの JSP のデバッグに有効です。

注意： オンデマンド変換の場合、JSP の emit_debuginfo 構成パラメータが同等の機能を提供します。9-7 ページの「Oracle JSP の構成パラメータ」を参照してください。

-extend（完全修飾された Java クラス名。ojspc のデフォルトは空。）

このオプションを使用すると、生成されたページ実装クラスが拡張する Java クラスを指定できます。

-extres（ブール値。ojspc のデフォルトは false。）

このフラグを有効にすると、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・ファイル内に配置するように ojspc に指示できます。

リソース・ファイル名は、JSP ページ名に基づいて決定されます。Oracle JSP リリース 1.1.x.x の場合、（JSP 名に特殊文字が含まれないかぎり）コア名は、JSP 名と同じになりますが、先頭にアンダースコア（「_」）が付き、末尾に .res という接尾辞が付きます。たとえば、MyPage.jsp を変換すると、通常の出力の他に、_MyPage.res が作成されます。ただし、名前生成の正確な実装は、将来のリリースで変更される可能性があります。

リソース・ファイルは、.class ファイルと同じディレクトリに配置されます。

ページ内に多くの静的コンテンツが含まれている場合は、この方法によって変換が高速化され、ページの実行が高速化される場合があります。詳細は、4-10 ページの「JSP ページに大量の静的コンテンツが含まれている場合の対策」を参照してください。

注意：

- この場合もインナー・クラスが作成されるため、配置する必要があります。
- オンデマンド変換の場合、JSP の external_resource 構成パラメータが同等の機能を提供します。9-7 ページの「Oracle JSP の構成パラメータ」を参照してください。

-implement（完全修飾された Java インタフェース名。ojspc のデフォルトは空）。

このオプションを使用すると、生成されたページ実装クラスが実装する Java インタフェースを指定できます。

-noCompile（ブール値。ojspc のデフォルトは false。）

このフラグを有効にすると、生成されたページ実装クラスの Java ソースをコンパイルしないように ojspc に指示できます。これによって、後で別の Java コンパイラを使用してコンパイルできます。

注意：

- オンデマンド変換の場合、JSP の javaccmd 構成パラメータが関連機能を提供します。この機能を使用すると、別の Java コンパイラを直接指定できます。9-7 ページの「Oracle JSP の構成パラメータ」を参照してください。
- SQLJ JSP ページの場合、-noCompile を有効にすると Java のコンパイルのみが実行できなくなり、SQLJ による変換に影響はありません。

-packageName（完全修飾されたパッケージ名。ojspc のデフォルトは .jsp ファイルの格納場所に従う。）

このオプションを使用すると、Java のドット構文を使用して生成されたページ実装クラスにパッケージ名を指定できます。

このオプションを設定しない場合、パッケージ名は、（ojspc を実行している）カレント・ディレクトリからの .jsp ファイルの相対パスに従って決定されます。

.jsp ファイルが /myapproot/src/jspsrc ディレクトリに存在する場合に、/myapproot ディレクトリから ojspc を実行する例について考えてみます（% は UNIX プロンプトであるとします）。

```
% cd /myapproot
% ojspc -packageName myroot.mypackage src/jspsrc/Foo.jsp
```

これによって、myroot.mypackage がパッケージ名として使用されます。

この例で -packageName オプションが使用されない場合は、Oracle JSP リリース 1.1.x.x によって、デフォルトで、_src._jspsrc がパッケージ名として使用されます（これらの実装の詳細は、将来のリリースで変更される可能性があることに注意してください）。

-S-<sqlj option> <value>（-S とそれに続く SQLJ オプション設定。ojspc のデフォルトは空。）

SQLJ JSP ページの場合、Oracle SQLJ オプションを SQLJ トランスレータに渡すには、ojspc の -S オプションを使用します。-S を複数使用し、それぞれに 1 つずつ SQL オプションを設定できます。

SQLJ トランスレータを直接実行する場合とは異なり、（他の ojspc オプションとの一貫性を維持するために）SQLJ オプションとその値の間に空白を使用します。

次に（UNIX プロンプトから実行する場合の）例を示します。

```
% ojspc -S-codegen iso -d /myapproot/mybindir MyPage.jsp
```

これによって、デフォルトの Oracle 固有のコードではなく ISO 規格コードを生成するように SQLJ に指示します。

次に別の例を示します。

```
% ojspc -S-codegen iso -S-ser2class true -d /myapproot/mybindir MyPage.jsp
```

この場合も、ISO 規格コードを生成し、-ser2class オプションを有効にしてプロファイルを .class ファイルに変換するように SQLJ に指示します。

> **注意：** 前述の例に示すとおり、-S オプションの設定を使用した SQLJ のブール値オプションの有効化には、明示的な true 設定を使用できます。これが、明示的な true 設定を取らない ojspc ブール値オプションとの相違点です。

Oracle SQLJ オプションの場合、次のことに注意してください。

- SQLJ の -encoding オプションではなく、JSP ページの page ディレクティブの 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、SQLJ の -classpath オプションは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、SQLJ の -compile オプションは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、SQLJ の -d オプションは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、SQLJ の -dir オプションは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、『Oracle9*i* 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。

> **注意：** オンデマンド変換の場合、JSP の sqljcmd 構成パラメータが関連機能を提供します。この機能を使用すると、別の SQLJ トランスレータまたは SQLJ オプション設定を指定できます。9-7 ページの「Oracle JSP の構成パラメータ」を参照してください。

-srcdir（完全修飾されたパス。ojspc のデフォルトはカレント・ディレクトリ。）

このオプションを使用すると、ojspc が、生成されたソース・ファイル（SQLJ JSP ページの場合の .sqlj ファイル、および .java ファイル）を配置するためのベース・ディレクトリを指定できます。

指定されたパスは、（アプリケーション相対パスまたはページ相対パスではなく）単にファイル・システムのパスとして使用されます。

指定されたディレクトリ下のサブディレクトリは、必要に応じて、パッケージによって自動的に作成されます。詳細は、6-24 ページの「ojspc の出力ファイル、格納場所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 が実行されているカレント・ディレクトリ）が使用されます。

このオプションを使用して、生成されたソース・ファイルを未使用のディレクトリに配置し、作成されたファイルを簡単に識別できるようにすることをお薦めします。

注意：

- ディレクトリ名に空白を使用できる Windows NT などの環境では、ディレクトリ名を引用符で囲みます。
- オンデマンド変換の場合、JSP の page_repository_root 構成パラメータが関連機能を提供します。9-7 ページの「Oracle JSP の構成パラメータ」を参照してください。

-verbose（ブール値。ojspc のデフォルトは false。）

このオプションを有効にすると、実行時に変換手順を報告するように ojspc に指示できます。

次の例に、myerror.jsp の変換に対する -verbose の出力を示します。（この例では、myerror.jsp が存在するディレクトリから ojspc が実行されます。% は UNIX プロンプトであると想定します。）

```
% ojspc -verbose myerror.jsp
Translating file: myerror.jsp
1 JSP files translated successfully.
Compiling Java file: ./_myerror.java
```

-version（ブール値。ojspc のデフォルトは false。）

Oracle JSP のバージョン番号を表示し、次にこれを実行するには、ojspc に対してこのオプションを有効にします。

ojspc の出力ファイル、格納場所および関連オプションの概要

デフォルトでは、ojspc は、Oracle JSP トランスレータが生成する一連のファイルと同じファイルをオンデマンド変換で生成し、（ojspc が実行されている）カレント・ディレクトリまたはその下にこれらのファイルを配置します。

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 JSP ページのみ）
- .java ソース・ファイル
- ページ実装クラスの .class ファイル
- 静的テキスト用のインナー・クラスの .class ファイル

- Java リソース・ファイル（.ser）またはオプションで、SQLJ プロファイル用の .class ファイル（SQLJ JSP ページのみ）

  この場合、標準の SQLJ コード生成が行われるとします。Oracle 固有の SQLJ コード生成では、プロファイルは作成されません。

- ページの静的テキスト用の Java リソース・ファイル（.res）（オプション）

Oracle JSP トランスレータによって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および格納場所（オンデマンド変換）」を参照してください。

6-18 ページの「ojspc のオプションの説明」で説明したいくつかの一般的なオプションについてまとめると、ファイルの生成および配置には、次の ojspc オプションを使用できます。

- アプリケーション・ルート・ディレクトリを指定するには、-appRoot を使用します。
- 別の格納場所を指定してソース・ファイルを配置するには、-srcdir を使用します。
- 別の格納場所を指定してバイナリ・ファイル（.class ファイルおよび Java リソース・ファイル）を配置するには、-d を使用します。
- 生成されたページ実装クラスのソースがコンパイルされないようにするには、-noCompile を使用します（このオプションを使用すると、.class ファイルが作成されません）。

  SQLJ JSP ページの場合、変換された .java ファイルが作成されますが、コンパイルは行われません。

- Java リソース・ファイルに静的テキストを挿入するには、-extres を使用します。
- Java リソース・ファイル .ser ではなく .class ファイルに SQLJ プロファイルを生成するには、-S-ser2class を使用します（SQLJ の -ser2class オプションは、SQLJ JSP ページのみに、また、ISO 規格の SQLJ コード生成のみに使用できます）。

出力ファイルの配置では、カレント・ディレクトリの下のディレクトリ構造（可能な場合は、-d および -srcdir オプションによって指定されたディレクトリ）は、パッケージに基づいて決定されます。パッケージは、アプリケーション・ルート（カレント・ディレクトリまたは -appRoot オプションで指定されたディレクトリ）からの変換対象ファイルの相対パスに基づいて決定されます。

たとえば、次のとおり ojspc を実行するとします（% は UNIX プロンプトであるとします）。

```
% cd /abc
% ojspc def/ghi/test.jsp
```

これによって、パッケージは _def._ghi になり、出力ファイルは /abc/_def/_ghi ディレクトリに配置されます。プロセスの一部として、_def/_ghi サブディレクトリが作成されます。

-d および -srcdir オプションを介して出力に別の格納場所を指定する場合、指定したディレクトリに _def/_ghi サブディレクトリ構造が作成されます。

他のディレクトリから ojspc が実行されるとします。次に例を示します。

```
% cd /home/mydir
% ojspc -appRoot /abc /abc/def/ghi/test.jsp
```

この場合も、パッケージは、指定したアプリケーション・ルートからの test.jsp の相対パスに従って、_def._ghi になります。出力ファイルは、-d および -srcdir オプションを介して指定された格納場所の、/home/mydir/_def/_ghi ディレクトリまたは _def/_ghi サブディレクトリに配置されます。いずれの場合も、プロセスの一部として、_def/_ghi サブディレクトリ構造が作成されます。

注意： JSP アプリケーションの各ディレクトリに対して ojspc を 1 回実行することをお薦めします。これによって、異なるディレクトリに存在するファイルに対して、必要に応じて、異なるパッケージ名を付けることができます。

JSP による配置に関する他の考慮点

この項では、多くの場合ターゲット環境には関係のない、配置に関する様々な一般的考慮点および使用例について説明します。

この項の内容は次のとおりです。

実行を伴わない JSP による一般的な事前変換

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の JSP ページの起動時に、標準の jsp_precompile リクエスト・パラメータを有効にすることによって、実行することなく JSP による事前変換のみが行われるように指定できます。

次に例を示します。

```
http://host[:port]/foo.jsp?jsp_precompile
```

詳細は、Sun 社の JavaServer Pages 仕様バージョン 1.1 を参照してください。

バイナリ・ファイルのみの配置

独自の JSP ソースの場合は、JSP ページを事前変換し、変換およびコンパイル済のバイナリ・ファイルのみを配置することによって、ソースの公開を回避できます。オンデマンド変換で実行された以前の変換または ojspc を使用して事前変換されたページは、Oracle JSP コンテナをサポートするすべての環境に配置できます。この場合、次の 2 つの点に注意する必要があります。

- バイナリ・ファイルを適切に配置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が実行されるように、Oracle JSP コンテナを適切に構成する必要があります。

バイナリ・ファイルの配置

JSP ページの変換後、バイナリ出力ディレクトリの下に存在するディレクトリ構造およびコンテンツをアーカイブし、次に、ターゲット環境にディレクトリ構造およびコンテンツを適切にコピーします。次に例を示します。

- ojspc を使用して事前変換する場合は、ojspc の -d オプションを使用してバイナリの出力ディレクトリを指定し、次に、指定した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。
- JServ（オンデマンド変換）環境での以前の実行時に作成されたバイナリ・ファイルをアーカイブする場合は、通常は、デフォルトの htdocs/_pages ディレクトリの下に存在する出力ディレクトリ構造をアーカイブします。

ターゲット環境で、アーカイブされたディレクトリ構造を、適切なディレクトリ（JServ 環境の htdocs/_pages ディレクトリの下など）にリストアします。

バイナリ・ファイルのみを使用して実行される Oracle JSP コンテナの構成

.jsp または .sqljsp ソースが使用できない場合に JSP ページを実行するには、次のとおり JSP 構成パラメータを設定します。

- bypass_source を true に設定します。
- developer_mode を false に設定します。

これらの設定を行わない場合、Oracle JSP コンテナは、.jsp または .sqljsp ファイルがページ実装 .class ファイルより後に変更されているかどうかを確認するために、常に、これらのファイルを検索します。.jsp または .sqljsp ファイルを検出できない場合は、「file not found」というエラーが発生して異常終了します。

これらのパラメータが適切に設定されている場合、エンド・ユーザーは、これらのソース・ファイルが存在する場合に使用される URL と同じ URL を使用して、ページを起動できます。たとえば、JServ 環境について考えてみます。foo.jsp のバイナリ・ファイルが htdocs/_pages/_test ディレクトリに存在する場合、foo.jsp が存在しなくても、次の URL を使用してページを起動できます。

```
http://host:[port]/test/foo.jsp
```

JServ 環境での構成パラメータの設定方法については、9-17 ページの「JServ での JSP パラメータの設定」を参照してください。

Oracle9*i* JDeveloper を使用した JSP ページの配置

Oracle9*i* JDeveloper リリース 3.1 以上には、配置オプションの「Web Application to Web Server」が含まれます。このオプションは、JSP アプリケーション用に追加されています。

このオプションを使用すると、次の要素を指定する配置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時にアプリケーションを配置するか、または後での使用のためにプロファイルを保存できます。

JServ のドキュメント・ルート

Oracle9*i* に付属の JServ 環境で実行される JSP ページおよびサーブレット（Jserv に付属の Apache の mod_jserv モジュールを介してルーティングされる）では、Apache のドキュメント・ルートが使用されます。このドキュメント・ルート（通常 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 で JSP ページを実行する場合は、静的ファイルと同様に JSP ページも、ドキュメント・ルート内またはその下に存在します。

注意： Oracle HTTP Server の役割およびこのサーバーの mod_jserv モジュールの概要は、2-3 ページの「Oracle HTTP Server の役割」を参照してください。

7

JSP のタグ・ライブラリ

この章では、カスタム・タグ・ライブラリについて説明します。また、ベンダーが独自のライブラリを提供するために使用できる基本フレームワーク、および標準の実行時タグとベンダー固有のコンパイル時タグの比較についても説明します。この章の内容は次のとおりです。

- 標準のタグ・ライブラリ・フレームワーク
- コンパイル時タグ

2-9 ページの「Oracle9i に付属の JSP タグ・ライブラリおよび JavaBeans の概要」に示した Oracle9*i* リリース 2（9.2）に付属のタグ・ライブラリの詳細は、『Oracle9*i*AS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

注意： タグ・ライブラリ・フレームワークは、Oracle JSP コンテナによって、JServ（Servlet 2.0）環境でもサポートされます。ただし、Servlet 2.2 のタグ・ライブラリを完全にサポートするには、Oracle9*i*AS Containers for J2EE（推奨）、Tomcat などの Servlet 2.2 以上の環境を使用する必要があります。

標準のタグ・ライブラリ・フレームワーク

標準の JavaServer Pages テクノロジを使用すると、ベンダーは、JSP のカスタム・タグ・ライブラリを作成できます。

タグ・ライブラリは、カスタム・アクションのコレクションを定義します。これらのタグは、JSP ページのコーディング時に開発者が直接使用するか、Java 開発ツールで自動的に使用することができます。タグ・ライブラリは、異なる 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・ライブラリおよび標準の JavaServer Pages タグ・ライブラリ・フレームワークの詳細は、次のリソースを参照してください。

- Sun 社の JavaServer Pages 仕様バージョン 1.1
- 次の Web サイトにある Sun 社の `javax.servlet.jsp.tagext` パッケージ用の Javadoc

`http://java.sun.com/j2ee/j2sdkee/techdocs/api/javax/servlet/jsp/tagext/package-summary.html`

カスタム・タグ・ライブラリ実装の概要

カスタム・タグ・ライブラリには、次の一般的な形式の `taglib` ディレクティブを介して、JSP ページからアクセスできます。

```
<%@ taglib uri="URI" prefix="prefix" %>
```

次のことに注意してください。

- ライブラリのタグは、7-11 ページの「タグ・ライブラリ記述ファイル」に示すとおり、タグ・ライブラリ記述ファイルで定義されます。
- `taglib` ディレクティブ内の Uniform Resource Identifier（URI）は、7-14 ページの「taglib ディレクティブ」に示すとおり、タグ・ライブラリ記述ファイルの検出場所を指定します。7-12 ページの「タグ・ライブラリに対する web.xml の使用」に示すとおり、URI のショート・カットを使用できます。

- taglib ディレクティブ内の接頭辞には、ユーザーがライブラリ内の任意のタグとともに JSP ページ内で使用する任意の文字列を指定できます。

  taglib ディレクティブで、接頭辞 oracust が次のとおり指定されているとします。

  ```
  <%@ taglib uri="URI" prefix="oracust" %>
  ```

  さらに、ライブラリにタグ mytag が含まれているとします。この場合、mytag を次のとおり使用できます。

  ```
  <oracust:mytag attr1="...", attr2="..." />
  ```

  oracust 接頭辞を使用すると、mytag が、前述の taglib ディレクティブに指定した URI で検出可能なタグ・ライブラリ記述ファイルに定義されていることを、JSP トランスレータに通知できます。

- タグ・ライブラリ記述ファイル内のタグ用エントリは、そのタグが（mytag と同様に）属性を使用するかどうか、使用する場合のそれらの属性名など、そのタグの使用方法に関する仕様を指定します。

- タグのセマンティクス（タグの使用によって発生するアクション）は、7-4 ページの「タグ・ハンドラ」に示すとおり、タグ・ハンドラ・クラスで定義されます。各タグには独自のタグ・ハンドラ・クラスがあり、そのクラス名は、タグ・ライブラリ記述ファイルで指定されます。

- タグ・ライブラリ記述ファイルは、タグが本体を使用するかどうかを示します。

  前述のとおり、本体を持たないタグは、次の例に示すとおり使用されます。

  ```
  <oracust:mytag attr1="...", attr2="..." />
  ```

  それに対して、本体を持つタグは、次の例に示すとおり使用されます。

  ```
  <oracust:mytag attr1="...", attr2="..." >
     ...body...
  </oracust:mytag>
  ```

- カスタム・タグ・アクションは、タグ自身または他の JSP スクリプト要素（スクリプトレットなど）によって使用可能な 1 つ以上のサーバー側オブジェクトを作成できます。これらのオブジェクトを、スクリプト変数といいます。

  カスタム・タグが使用するスクリプト変数の詳細は、タグ追加情報クラスで定義されます。これについては、7-7 ページの「スクリプト変数およびタグ追加情報クラス」を参照してください。

  タグは、次の例に示すような構文を持つスクリプト変数を作成できます。この例では、myobj というオブジェクトが作成されます。

  ```
  <oracust:mytag id="myobj" attr1="...", attr2="..." />
  ```

- ネストしたタグのタグ・ハンドラは、そのタグの処理または状態管理に必要な場合、倍部のタグのタグ・ハンドラにアクセスできます。7-10 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。

これらの項目の詳細は、次の各項を参照してください。

タグ・ハンドラ

タグ・ハンドラは、カスタム・タグの使用によって発生するアクションのセマンティクスを記述します。タグ・ハンドラは、タグが（開始タグと終了タグの間にある）本体の文を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、タグ `abc` のタグ・ハンドラ・クラスの名前は、通常、`AbcTag` です。

タグ・ライブラリのタグ・ライブラリ記述（TLD）ファイルは、ライブラリに含まれる各タグのタグ・ハンドラ・クラスの名前を指定します（7-11 ページの「タグ・ライブラリ記述ファイル」を参照）。

タグ・ハンドラ・インスタンスは、リクエスト時に使用されるサーバー側オブジェクトです。このインスタンスは、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、親タグのハンドラ・オブジェクト（カスタム・タグを外部のカスタム・タグ内にネストして使用する場合）などの、JSP コンテナによって設定されるプロパティを持ちます。

タグ・ハンドラ・クラスのサンプル・コードについては、7-16 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 社の JavaServer Pages 仕様バージョン 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、同じタグ・ハンドラ・インスタンスを使用するか、または異なるタグ・ハンドラ・インスタンスを使用するかが規定されていません。この実装の詳細は、JSP ベンダーが決定します。Oracle JSP コンテナでは、タグを使用するたびに、個別のタグ・ハンドラ・インスタンスを使用します。

カスタム・タグの本体の処理

カスタム・タグは、標準の JSP タグと同様に、本体を持つ場合と持たない場合があります。カスタム・タグの場合は、本体が存在しても、タグ・ハンドラによる特別な処理が必要ない場合があります。

次の 3 つの場合が考えられます。

- 本体が存在しない場合

  この場合、開始タグと終了タグではなく、単一のタグのみが存在します。次に一般的な例を示します。

  ```
  <oracust:abcdef attr1="...", attr2="..." />
  ```

- タグ・ハンドラによる特別な処理を必要としない本体が存在する場合

  この場合、本体の文は開始タグと終了タグに囲まれていますが、タグ・ハンドラで本体を処理する必要がありません。本体の文は、通常の JSP 処理のためにのみ渡されます。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・ハンドラによる特別な処理を必要とする本体が存在する場合

  この場合も、本体の文は開始タグと終了タグに囲まれていますが、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>
  ```

本体を処理するための整定数

次の項で説明するタグ処理用のインタフェースは、状況に応じて適切な `int` 型定数を戻すように実装する必要がある `doStartTag()` メソッド（詳細は後述）を指定します。可能な戻り値は次のとおりです。

- `SKIP_BODY`: 本体が存在しない場合、または本体の評価および実行を省略する必要がある場合
- `EVAL_BODY_INCLUDE`: タグ・ハンドラによる特別な処理を必要としない本体が存在する場合
- `EVAL_BODY_TAG`: タグ・ハンドラによる特別な処理を必要とする本体が存在する場合

本体を処理しないタグのハンドラ

本体を持たないか、またはタグ・ハンドラによる特別な処理を必要としない本体を持つカスタム・タグの場合、タグ・ハンドラ・クラスは、次の標準インタフェースを実装します。

- `javax.servlet.jsp.tagext.Tag`

次の標準サポート・クラスは、`Tag` インタフェースを実装し、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()` メソッドおよび `doEndTag()` メソッドを指定します。タグ開発者は、必要に応じてタグ・ハンドラ・クラスのこれらのメソッドが、開始タグおよび終了タグそれぞれの検出時に実行されるように、コードを指定する必要があります。Oracle JSP トランスレータによって生成された JSP ページ実装クラスには、これらのメソッドへの適切なコールが含まれます。

アクションの処理は、アクション・タグを使用して何を実行する場合でも、`doStartTag()` メソッドで実装されます。`doEndTag()` メソッドは、すべての適切な後処理を実装します。本体を持たないタグの場合、基本的に、`doStartTag()` メソッドが実行されてから `doEndTag()` メソッドが実行されるまでの間に何も行われません。

`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`（7-5 ページの「本体を処理するための整定数」を参照）である必要があります。`EVAL_BODY_TAG` は、`Tag` インタフェースを実装するタグ・ハンドラ・クラスには無効です。

本体を処理するタグのハンドラ

タグ・ハンドラによる特別な処理を必要とする本体を持つカスタム・タグの場合、タグ・ハンドラ・クラスは、次の標準インタフェースを実装します。

- `javax.servlet.jsp.tagext.BodyTag`

次の標準サポート・クラスは、`BodyTag` インタフェースを実装し、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る `doStartTag()` メソッドおよび `doEndTag()` メソッドの他に、`doInitBody()` メソッドおよび `doAfterBody()` メソッドを指定します。

タグ開発者は、`Tag` インタフェースを実装するタグ・ハンドラの場合（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、タグによるアクション処理のために `doStartTag()` メソッドを実装し、すべての後処理のために `doEndTag()` メソッドを実装する必要があります。

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（7-5 ページの「本体を処理するための整定数」を参照）である必要があります。EVAL_BODY_INCLUDE は、BodyTag インタフェースを実装するタグ・ハンドラ・クラスには無効です。

doStartTag() メソッドおよび doEndTag() メソッドの実装の他に、タグ開発者は、必要に応じて本体の評価前に doInitBody() メソッドがコールされ、本体の評価後に doAfterBody() メソッドがコールされるように、コードを指定する必要があります（本体は、ループが終わるたびなど、複数回評価できます）。Oracle JSP トランスレータによって生成された JSP ページ実装クラスには、これらのすべてのメソッドへの適切なコールが含まれます。

doStartTag() メソッドの実行後、doStartTag() メソッドが EVAL_BODY_TAG を戻した場合は、doInitBody() メソッドおよび doAfterBody() メソッドが実行されます。

doEndTag() メソッドは、すべての本体処理が終了し、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、javax.servlet.jsp.tagext.BodyContent クラスを使用できます。このクラスは、javax.servlet.jsp.JspWriter のサブクラスで、本体の評価を後で再抽出できるように処理するために使用できます。BodyTag インタフェースには、タグ・ハンドラ・インスタンスに BodyContent ハンドルを与えるために JSP コンテナで使用可能な setBodyContent() メソッドが含まれます。

スクリプト変数およびタグ追加情報クラス

カスタム・タグ・アクションは、タグ自身または他のスクリプト要素（スクリプトレットや他のタグなど）によって使用可能な、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。

カスタム・タグが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タグ追加情報クラスといいます。

JSP コンテナは、変換中にタグ追加情報インスタンスを使用します（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は、可能な場合、指定されたタグに使用するタグ追加情報クラスを指定します）。

タグ追加情報クラスには、HTTP リクエスト中に割り当てられるスクリプト変数の名前および型を取得する getVariableInfo() メソッドが含まれます。JSP トランスレータは、変換中にこのメソッドをコールして、このメソッドに標準の javax.servlet.jsp.tagext.TagData クラスのインスタンスを渡します。TagData インスタンスは、カスタム・タグを使用する JSP 文内に設定された属性値を指定します。

この項の内容は次のとおりです。

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たオブジェクトは、オブジェクト ID をハンドルとして使用することによって、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で参照できます。次の例について考えてみます。

```
<oracust:foo id="myobj" attr1="..." attr2="..." />
```

この文を実行すると、タグとページの終わりの間に存在するすべてのスクリプト要素が、オブジェクト myobj を使用できるようになります。id 属性は、変換時属性です。タグ開発者は、JSP コンテナが使用するタグ追加情報クラスを指定する必要があります。タグ追加情報クラスは、特に、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。

JSP コンテナは、myobj をページ・コンテキスト・オブジェクトに挿入します。このオブジェクトでは、他のタグまたはスクリプト要素が、次のような構文を使用して後で myobj を取得できます。

```
<oracust:bar ref="myobj" />
```

myobj オブジェクトは、foo および bar 用のタグ・ハンドラ・インスタンスを介して渡されます。認識される必要があるのは、オブジェクトの名前（myobj）のみです。

重要： id および ref は単にサンプルの属性名であり、これらの属性に対して事前定義された特別なセマンティクスはないことに注意してください。属性名の定義、およびページ・コンテキストでのオブジェクトの作成と取得は、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タグ追加情報クラスで指定します。有効範囲には、次のいずれかの int 型定数を指定できます。

- NESTED: スクリプト変数を、それを定義するアクションの開始タグと終了タグの間で使用できる場合
- AT_BEGIN: スクリプト変数を、開始タグからページの終わりまでの間で使用できる場合
- AT_END: スクリプト変数を、終了タグからページの終わりまでの間で使用できる場合

タグ追加情報クラスおよび getVariableInfo() メソッド

タグ追加情報クラスは、スクリプト変数を作成するすべてのカスタム・タグに対して作成する必要があります。タグ追加情報クラスは、スクリプト変数を記述します。このクラスは、標準の `javax.servlet.jsp.tagext.TagExtraInfo` 抽象クラスのサブクラスである必要があります。

`TagExtraInfo` クラスの主要なメソッドは `getVariableInfo()` です。このメソッドは、JSP トランスレータによってコールされ、標準 `javax.servlet.jsp.tagext.VariableInfo` クラスのインスタンスの配列（タグが作成す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。

タグ追加情報クラスは、次のスクリプト変数情報を使用して、各 `VariableInfo` インスタンスを作成します。

- スクリプト変数名
- スクリプト変数の Java 型（プリミティブ型は使用不可）
- スクリプト変数が新しく宣言された変数であるかどうかを示すブール値
- スクリプト変数の有効範囲

重要： Oracle JSP リリース 1.1.x.x では、`getVariableInfo()` メソッドは、スクリプト変数の Java 型に対して完全修飾されたクラス名（FQCN）または部分修飾されたクラス名（PQCN）を戻すことができます。FQCN は、以前のリリースでは必須でしたが、今回のリリースでも、ご使用をお薦めします。パッケージ間で重複するクラス名が存在する場合に混同を回避できます。

プリミティブ型はサポートされていないことに注意してください。

タグ追加情報クラスのサンプル・コードについては、7-17 ページの「サンプル・タグ追加情報クラス : 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・タグを使用する場合、そのタグのタグ・ハンドラ・インスタンスは、外部タグのタグ・ハンドラ・インスタンスにアクセスできます。外部タグのタグ・ハンドラ・インスタンスは、ネストしたタグが実行する処理および状態管理に有効な場合があります。

この機能は、javax.servlet.jsp.tagext.TagSupport クラスのスタティック・メソッド findAncestorWithClass() を介してサポートされています。外部タグ・ハンドラ・インスタンスは、ページ・コンテキスト・オブジェクトで指定されていない場合でも、指定されたタグ・ハンドラ・クラスに最も近い囲みインスタンスであるため、アクセスが可能です。

次の JSP コードの例について考えてみます。

```
<foo:bar1 attr="abc" >
   <foo:bar2 />
</foo:bar1>
```

bar2 タグ・ハンドラ・クラス（通常、Bar2Tag クラス）のコード内に、次のような文を含めることができます。

```
Tag bar1tag = TagSupport.findAncestorWithClass(this, Bar1Tag.class);
```

findAncestorWithClass() メソッドは、入力として次のものを取ります。

- findAncestorWithClass() のコール元のクラス・ハンドラ・インスタンス（前述の例では、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（前述の例では、Bar1Tag とされます）

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を戻します。この例では、Bar1Tag が javax.servlet.jsp.tagext.Tag インスタンスとして戻されます。

Bar2Tag インスタンスが外部 Bar1Tag インスタンスにアクセスできると、Bar2Tag が bar1 タグ属性値を必要とするか、または Bar1Tag インスタンスでメソッドをコールする必要がある場合に有効です。

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・ライブラリおよびそのライブラリの個々のタグに関する情報を含む XML 文書です。TLD ファイルの名前には、.tld 拡張子が付きます。

JSP コンテナは、ライブラリのタグの検出時に実行するアクションを判別する場合、TLD ファイルを使用します。

TLD ファイル内のタグ用エントリには、次のものが含まれます。

- カスタム・タグ名
- 対応するタグ・ハンドラ・クラスの名前
- 対応するタグ追加情報クラス（該当する場合）の名前
- タグの本体（存在する場合）の処理方法を示す情報
- タグの属性（そのカスタム・タグを使用する場合に必ず指定する属性）に関する情報

次に、TLD ファイル内の myaction タグ用エントリの例を示します。

```
<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>
```

このエントリでは、タグ・ハンドラ・クラスは MyactionTag で、タグ追加情報クラスは 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、タグの本体（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。

- empty という値は、タグが本体を使用しないことを示します。
- JSP という値は、タグの本体を 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。
- tagdependent という値は、タグの本体を変換しないことを示します。本体内のすべてのテキストが、静的テキストとして処理されます。

JSP ページ内の taglib ディレクティブは、TLD ファイルの検出場所を JSP コンテナに通知します（7-14 ページの「taglib ディレクティブ」を参照）。

タグ・ライブラリ記述ファイルの詳細は、Sun 社の JavaServer Pages 仕様バージョン 1.1 を参照してください。

注意： Tomcat 3.1 サーブレット /JSP 実装では、指定されたタグ自体（JSP ページ内）が本体を持っていない場合、そのタグに対する TLD ファイルの bodycontent パラメータは読み込まれません。そのため、TLD ファイルに無効な bodycontent 値（empty ではなく none など）が含まれる場合があります。そのファイルを別の JSP 環境（Oracle JSP コンテナなど）で使用すると、エラーが発生します。

タグ・ライブラリに対する web.xml の使用

Sun 社の Java Servlet 仕様バージョン 2.2 には、サーブレット用の標準デプロイメント・ディスクリプタ（web.xml ファイル）が記載されています。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 ファイルは、1 つの taglib 要素および次の 2 つのサブ要素を含むことができます。

- taglib-uri
- taglib-location

taglib-location サブ要素は、タグ・ライブラリ記述ファイルの位置を、アプリケーション相対パス（「/」で始まる）で示します。

taglib-uri サブ要素は、JSP ページ内の taglib ディレクティブで使用するショート・カット URI を示します。この URI は、このサブ要素に伴う taglib-location サブ要素で指定された TLD ファイルの位置にマップされています。（Uniform Resource Identifier（URI）は、Uniform Resource Locator（URL）とほぼ同義ですが、より包括的な用語です。）

重要： JSP アプリケーションで web.xml ファイルを使用する場合は、そのアプリケーションで web.xml を配置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ます。

次に、web.xml 内のタグ・ライブラリ記述ファイル用エントリの例を示します。

```
<taglib>
   <taglib-uri>/oracustomtags</taglib-uri>
   <taglib-location>/WEB-INF/oracustomtags/tlds/MyTLD.tld</taglib-location>
</taglib>
```

これによって、/oracustomtags が、JSP ページの taglib ディレクティブ内の /WEB-INF/oracustomtags/tlds/MyTLD.tld と等しくなります。例については、7-14 ページの「TLD ファイルに対するショート・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それをタグ・ライブラリ記述ファイルに対して使用する方法の詳細は、Sun 社の Java Servlet 仕様バージョン 2.2 および JavaServer Pages 仕様バージョン 1.1 を参照してください。

注意：

- Tomcat 3.1 サーブレット /JSP 実装の web.xml サンプル・ファイルは使用しないでください。このサンプル・ファイルには、XML の標準 DTD 検証が正常に実行されない新しい要素が導入されています。
- web.xml ファイルでは、「uri」のかわりに「urn」という用語は使用しないでください。一部の JSP 実装（Tomcat 3.1 など）では「urn」を使用できますが、「urn」を使用すると、XML の標準 DTD 検証が正常に実行されません。

taglib ディレクティブ

taglib ディレクティブを次のとおり使用して、カスタム・ライブラリを JSP ページにインポートします。

```
<%@ taglib uri="URI" prefix="prefix" %>
```

URI には、次のオプションがあります。

- web.xml ファイルに定義されているとおり、ショート・カット URI を指定します（7-12 ページの「タグ・ライブラリに対する web.xml の使用」を参照）。
- タグ・ライブラリ記述（TLD）ファイルの名前および位置を完全に指定します。

TLD ファイルに対するショート・カット URI の使用

web.xml に、タグ・ライブラリ記述ファイル MyTLD.tld で定義されたタグ・ライブラリ用の次のエントリがあるとします。

```
<taglib>
   <taglib-uri>/oracustomtags</taglib-uri>
   <taglib-location>/WEB-INF/oracustomtags/tlds/MyTLD.tld</taglib-location>
</taglib>
```

この例では、JSP ページで次のディレクティブを使用すると、JSP コンテナが web.xml 内で /oracustomtags という URI を検索するため、それに伴うタグ・ライブラリ記述ファイル（MyTLD.tld）の名前および位置も検索します。

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

この文を使用すると、JSP ページ内のこのカスタム・タグ・ライブラリのすべてのタグを使用できます。

TLD ファイルの名前および位置の完全な指定

JSP アプリケーションで web.xml ファイルに依存せずにタグ・ライブラリを使用するには、次のとおり、taglib ディレクティブでタグ・ライブラリ記述ファイルの名前および位置を完全に指定します。

```
<%@ taglib uri="/WEB-INF/oracustomtags/tlds/MyTLD.tld" prefix="oracust" %>
```

位置は、アプリケーション相対パスとして指定します（この例に示すとおり「/」で始める）。アプリケーションに相対的な構文については、1-8 ページの「JSP ページのリクエスト」を参照してください。

または、taglib ディレクティブに .tld ファイルではなく .jar ファイルを指定することもできます。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ます。JAR ファイル（Servlet 2.2 用）を作成する場合は、タグ・ライブラリ記述ファイルの位置および名前を次のとおり指定する必要があります。

```
META-INF/taglib.tld
```

この場合、taglib ディレクティブは、次の例のようになります。

```
<%@ taglib uri="/WEB-INF/oracustomtags/tlds/MyTLD.jar" prefix="oracust" %>
```

例 : カスタム・タグの定義および使用

この項では、指定された回数にわたってタグの本体を反復処理するために使用する loop というカスタム・タグを定義および使用する例を示します。

この例には、次のものが含まれます。

- 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
- タグ追加情報クラスのソース・コード
- タグ・ライブラリ記述ファイル

注意 : 次に示すサンプル・コードでは、oracle.jsp.jml パッケージの拡張データ型を使用しています。これらの型については、『Oracle9*i*AS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

サンプル JSP ページ : exampletag.jsp

次に、loop タグを使用するサンプル JSP ページを示します。この例では、外部ループを 5 回、内部ループを 3 回実行することが指定されています。

```
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

この項では、ExampleLoopTag というタグ・ハンドラ・クラスのソース・コードを示します。次のこと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、タグの本体（基本的にループ）が処理されます。
- ループの処理ごとに、doAfterBody() メソッドがカウンタを増加させます。このメソッドは、反復が残っている場合は EVAL_BODY_TAG を戻し、最後の反復が終了すると SKIP_BODY を戻します。

次にコードを示します。

```
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・タグ追加情報クラス : ExampleLoopTagTEI.java

この項では、loop タグが使用するスクリプト変数を記述するタグ追加情報クラスのソース・コードを示します。

変数に対して、次のことを指定する VariableInfo インスタンスが作成されています。

- 変数名は、index 属性に基づく。
- 変数は、oracle.jsp.jml.JmlNumber タイプ（完全修飾されたクラス名として指定する必要がある）である。
- 変数は、新しく宣言されたものである。
- 変数の有効範囲は、NESTED である。

また、タグ追加情報クラスには、count 属性が有効（整数である必要がある）であるかどうかを判断する isValid() メソッドが含まれています。

```
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

この項では、タグ・ライブラリ用のタグ・ライブラリ記述（TLD）ファイルを示します。この例では、ライブラリは、loop という 1 つのタグのみで構成されます。

この TLD ファイルは、loop タグに対して、次のことを指定します。

- タグ・ハンドラ・クラスに examples.ExampleLoopTag を指定。
- タグ追加情報クラスに examples.ExampleLoopTagTEI を指定。
- JSP に bodycontent を指定。

  この操作は、JSP トランスレータが本体コードを処理および変換する必要があることを意味します。
- index 属性および count 属性は両方とも必須。

  count 属性には、リクエスト時に実行される JSP の式も指定できます。

次に TLD ファイルを示します。

```
<?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 tag 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 tag 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>
```

コンパイル時タグ

標準のタグ・ライブラリは、Sun 社の JavaServer Pages 仕様バージョン 1.1 に示されているとおり、実行時サポート・メカニズムを使用します。通常、標準のタグ・ライブラリは移植可能で、特定の JSP コンテナを必要としません。

ただし、ベンダーは、JSP トランスレータのベンダー固有の機能を介して、カスタム・タグをサポートすることもできます。このようなタグは、他のコンテナには移植できません。

通常、実行時メカニズムを使用する標準の移植可能タグを開発することをお薦めしますが、この項で説明するとおり、コンパイル時メカニズムを使用するタグの方が適切である場合があります。

コンパイル時および実行時に関する一般的な考慮点

JSP 1.1 仕様には、カスタム・タグ・ライブラリ用の実行時サポート・メカニズムについて記載されています。XML スタイルのタグ・ライブラリ記述ファイルを使用してタグを指定するこのメカニズムについては、7-2 ページの「標準のタグ・ライブラリ・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・ライブラリを作成および使用すると、そのライブラリは、任意の標準 JSP 環境に移植可能になります。

ただし、次の理由から、コンパイル時実装の使用の検討が必要な場合もあります。

- コンパイル時実装は、より効率的なコードを生成できます。
- コンパイル時実装を使用すると、開発者が変換およびコンパイル中にエラーをキャッチできるため、エンド・ユーザーによる実行時にエラーが発生しません。

将来的には、Oracle JSP コンテナで、コンパイル時タグ実装を使用してカスタム・タグ・ライブラリを作成するための汎用フレームワークがサポートされる予定です。コンパイル時タグ実装は Oracle JSP 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。

Oracle JML ライブラリ : コンパイル時および実行時

Oracle は、JSP Markup Language（JML）ライブラリという移植可能なタグ・ライブラリを提供します。このライブラリは、標準の JSP 1.1 実行時メカニズムを使用します。

ただし、JML タグは、コンパイル時メカニズムを介してもサポートされています。JML タグが最初に導入された JSP が、実行時メカニズムが導入された JSP 1.1 仕様より前のバージョンの JSP だったためです。コンパイル時タグは、下位互換性のために継続してサポートされています。

コンパイル時実装の一般的なメリットおよびデメリットは、Oracle JML タグ・ライブラリにも当てはまります。コンパイル時 JML 実装は、Oracle JSP の以前のバージョンで最初に導入されたため、コンパイル時 JML 実装を使用すると有効な場合もあります。コンパイル時 JML 実装にはいくつかの追加タグも含まれ、追加の式構文もサポートされています。

JML ライブラリの実行時バージョンおよびコンパイル時バージョンについては、『Oracle9*i*AS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

8

Oracle JSP のグローバリゼーション・サポート

Oracle JSP コンテナは、Sun 社の JavaServer Pages 仕様バージョン 1.1 に従って、標準のグローバリゼーション・サポート（National Language Support（NLS）ともいう）を提供します。また、マルチバイト・パラメータ・エンコーディングをサポートしないサーブレット環境に対する拡張サポートも提供します。

標準 Java でのローカライズされたコンテンツのサポートは、テキストの内部表現の統一に使用される Unicode 2.0 に依存します。Unicode は、代替キャラクタ・セットに変換する場合のベース・キャラクタ・セットとして使用されます。

この章では、Oracle JSP コンテナが Oracle グローバリゼーション・サポートを処理する方法の主要なポイントについて説明します。この章の内容は次のとおりです。

- page ディレクティブでのコンテンツ・タイプの設定
- コンテンツ・タイプの動的設定
- Oracle JSP によるマルチバイト・パラメータ・エンコーディングの拡張サポート

注意： Oracle グローバリゼーション・サポートの詳細は、『Oracle9*i* Database グローバリゼーション・サポート・ガイド』を参照してください。

---

page ディレクティブでのコンテンツ・タイプの設定

page ディレクティブの contentType パラメータを使用すると、MIME タイプの設定および JSP ページのキャラクタ・エンコーディングの設定（オプション）ができます。MIME タイプは、実行時の HTTP レスポンスに適用されます。キャラクタ・エンコーディング（設定した場合）は、変換中のページ・テキストおよび実行時の HTTP レスポンスの両方に適用されます。

次の page ディレクティブの構文を使用します。

```
<%@ page ... contentType="TYPE; charset=character_set" ... %>
```

または、デフォルトのキャラクタ・セットの使用中に 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 には Internet Assigned Numbers Authority（IANA）に登録済の MIME タイプを指定し、character_set には IANA に登録済のキャラクタ・セットを指定します（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、次のサイトで MIME タイプの登録を管理しています。

```
ftp://www.isi.edu/in-notes/iana/assignments/media-types/media-types
```

デフォルトのキャラクタ・エンコーディングは ISO-8859-1（Latin-1 ともいう）です。IANA は、次のサイトでキャラクタ・エンコーディングの登録を管理しています。「preferred MIME name」と示されている推奨 MIME 名（表示されている場合）を使用します。

```
http://www.iana.org/assignments/character-sets
```

(Java および Web ブラウザがサポートするキャラクタ・セットを使用するかぎり、JSP で IANA に登録済のキャラクタ・セットを使用する必要はありませんが、IANA のサイトには最も一般的なキャラクタ・セットがリストされています。IANA が示す推奨 MIME 名を使用することをお薦めします。)

page ディレクティブのパラメータは静的です。ページは、実行中、レスポンス用に異なる設定が必要であると認識した場合、次のどちらかの操作を実行します。

- 8-4 ページの「コンテンツ・タイプの動的設定」に示すとおり、サーブレットの response オブジェクト API を使用して、実行中にコンテンツ・タイプを設定します。
- リクエストを別の JSP ページまたはサー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、JSP ページのできるかぎり前の方で使用する必要があります。
- ISO-8859-1 以外のキャラクタ・セットで作成された JSP ページでは、該当するキャラクタ・セットを page ディレクティブに設定する必要があります。変換中に、ページがこのディレクティブの設定を認識する必要があるため、このディレクティブは動的には設定できません。動的設定は、実行時のみを対象としています。
- JSP 1.1 仕様は、JSP ページが、自身のコンテンツの配信に使用するキャラクタ・セットと同じキャラクタ・セットで作成されていることを前提としています。
- 技術的には他の使用例も考えられますが、このマニュアルでは、説明を簡単にするために、ページ・テキスト、リクエスト・パラメータおよびレスポンス・パラメータのいずれでも同じエンコーディングを使用する典型的なケースを想定しています。Netscape ブラウザおよび Internet Explorer は、ユーザーがレスポンス・パラメータに対して指定した設定に従って動作しますが、リクエスト・パラメータのエンコーディングはブラウザによって制御されます。

コンテンツ・タイプの動的設定

HTTP レスポンス用の適切なコンテンツ・タイプが実行時まで不明な場合は、コンテンツ・タイプを JSP ページで動的に設定できます。標準の javax.servlet.ServletResponse インタフェースは、この目的のために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。ここで、HttpServletResponse インタフェースは ServletResponse インタフェースの拡張です。

setContentType() メソッドの入力には、page ディレクティブで contentType を設定する場合と同様に、MIME タイプのみ、またはキャラクタ・セットおよび MIME タイプの両方を含むことができます。次に例を示します。

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

または

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

page ディレクティブの場合と同様に、デフォルトの MIME タイプは text/html で、デフォルトのキャラクタ・エンコーディングは ISO-8859-1 です。

このメソッドを使用しても、変換中の JSP ページのテキスト解析には影響はありません。変換中に特定のキャラクタ・セットが必要な場合は、8-2 ページの「page ディレクティブでのコンテンツ・タイプの設定」に示したとおり、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。

次の重要な使用方法に注意してください。

- setContentType() メソッドを使用している場合、JSP ページのバッファリングを行わないようにすることはできません。デフォルトでは、JSP ページのバッファリングが行われます。page ディレクティブに buffer="none" は設定しないでください。
- setContentType() のコールは、ページ内で、ブラウザへの出力または（JSP バッファをブラウザにフラッシュする）jsp:include コマンドより前に使用する必要があります。
- Servlet 2.2 環境では、response オブジェクトに setLocale() メソッドが含まれます。このメソッドは、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します。たとえば、次のメソッドをコールすると、Shift_JIS というキャラクタ・セットが設定されます。

  ```
  response.setLocale(new Locale("ja", "JP"));
  ```

Oracle JSP によるマルチバイト・パラメータ・エンコーディングの拡張サポート

リクエスト・パラメータのキャラクタ・エンコーディングは、HTTP 仕様では適切に定義されません。ほとんどの Servlet コンテナは、サーブレットのデフォルト・エンコーディング ISO-8859-1 を使用してリクエスト・パラメータを解析する必要があります。

Servlet コンテナがマルチバイト・リクエスト・パラメータおよび Bean プロパティの設定をエンコードできないこのような環境では、Oracle JSP コンテナが次の 2 つの方法で拡張サポートを提供します。

- setReqCharacterEncoding() メソッドを使用する方法

または

- translate_params 構成パラメータを使用する方法

setReqCharacterEncoding() メソッド

Oracle は、Servlet コンテナのデフォルトのエンコーディングが適切でない場合に有効な setReqCharacterEncoding() メソッドを提供します。このメソッドを使用して、マルチバイト・リクエスト・パラメータおよび Bean プロパティの設定のエンコーディング（Java コードでの getParameter() のコール、JSP コードで Bean プロパティを設定する jsp:setProperty 文など）を指定するには、デフォルトのエンコーディングがすでに適切な場合、このメソッドを使用する必要はありません。実際には、そのような場合にこのメソッドを使用すると、アプリケーションでパフォーマンスのオーバーヘッドが発生する場合があります。

setReqCharacterEncoding() メソッドは、oracle.jsp.util パッケージ内の PublicUtil クラスのスタティック・メソッドです。

このメソッドは、次のパラメータ名およびパラメータ値に影響します。

- request オブジェクトの getParameter() メソッドの出力
- request オブジェクトの getParameterValues() メソッドの出力
- request オブジェクトの getParameterNames() メソッドの出力
- Bean プロパティ値の jsp:setProperty 設定

このメソッドをコールする場合は、request オブジェクト、および使用するエンコーディングを指定する文字列を次のとおり入力します。

```
oracle.jsp.util.PublicUtil.setReqCharacterEncoding(myrequest, "EUC-JP");
```

注意：

- Oracle JSP リリース 1.1.2.x 以上では、8-6 ページの「translate_params 構成パラメータ」で説明する translate_params 構成パラメータではなく、setReqCharacterEncoding() メソッドを使用することをお薦めします。
- setReqCharacterEncoding() メソッドは、Servlet 2.3 API の request.setCharacterEncoding(encoding) メソッドと上位互換性があります。

translate_params 構成パラメータ

この項では、JSP の translate_params 構成パラメータを使用して、マルチバイト・リクエスト・パラメータおよび Bean プロパティの設定をエンコードする方法（Java コードでの getParameter() のコール、JSP コードで Bean プロパティを設定する jsp:setProperty 文の場合など）について説明します。

Oracle JSP リリース 1.1.2.x 以上では、この構成パラメータではなく、PublicUtil.setReqCharacterEncoding() メソッドを使用することをお薦めします。8-5 ページの「setReqCharacterEncoding() メソッド」を参照してください。

また、次のいずれの環境でも、translate_params を有効にしないでください。

- Servlet コンテナがマルチバイト・パラメータ・エンコーディング自体を適切に処理する場合

  この場合、translate_params を true に設定すると、不適切な結果となります。ただし、このマニュアルを作成した時点では、JServ、JSWDK および Tomcat はいずれもマルチバイト・パラメータ・エンコーディングを適切に処理しないことがわかっています。

- リクエスト・パラメータが、JSP の page ディレクティブまたは setContentType() メソッドでレスポンス用に指定されたエンコーディングとは異なるエンコーディングを使用する場合
- translate_params と同等に機能するコードがすでに JSP ページ内に存在する場合

  8-7 ページの「translate_params 構成パラメータと同等のコード」を参照してください。

非マルチバイト Servlet コンテナのオーバーライドにおける translate_params の影響

translate_params を true に設定すると、マルチバイト・リクエスト・パラメータおよび Bean プロパティの設定をエンコードできない Servlet コンテナがオーバーライドされます（JServ 環境で JSP の構成パラメータを設定する方法は、9-17 ページの「JServ での JSP パラメータの設定」を参照）。

このフラグを有効にすると、Oracle JSP コンテナは、response.getCharacterEncoding() メソッドで指定されているとおり、response オブジェクトのキャラクタ・セットに基づいて、リクエスト・パラメータおよび Bean プロパティの設定をエンコードします。

translate_params フラグは、次のパラメータ名およびパラメータ値に影響します。

- request オブジェクトの getParameter() メソッドの出力
- request オブジェクトの getParameterValues() メソッドの出力
- request オブジェクトの getParameterNames() メソッドの出力
- Bean プロパティ値の jsp:setProperty 設定

translate_params 構成パラメータと同等のコード

translate_params 構成パラメータを使用したくない場合、または使用できない場合があります。このような場合、JSP ページでスクリプトレット・コードを介して実装できる同等の機能を知っておくと有効です。次に例を示します。

```
<%@ 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");
...
```

このコードは、次の操作を実行します。

- XXYYZZ を、検索するパラメータ名として設定します。(XX、YY および ZZ は 3 つの日本語文字であるとします。)
- Servlet コンテナが解析できるように、パラメータ名を Servlet コンテナのキャラクタ・セット ISO-8859-1 にエンコードします。(最初に、request オブジェクトのキャラクタ・エンコーディングを使用して、パラメータ名に対するバイト配列が作成されます。)
- パラメータ名と一致する値を検索することによって、request オブジェクトからパラメータ値を取得します(request オブジェクト内のパラメータ名も ISO-8859-1 でエンコードされているため、一致するものを検索できます)。
- さらに処理を行うため、またはブラウザへの出力を行うために、パラメータ値を EUC-JP にエンコードします。

次の 2 つの項に、translate_params の有効化に依存するグローバリゼーションのサンプル、および同等のコードを含むため、translate_params の設定に依存しないグローバリゼーションのサンプルを示します。

translate_params に依存するグローバリゼーションのサンプル

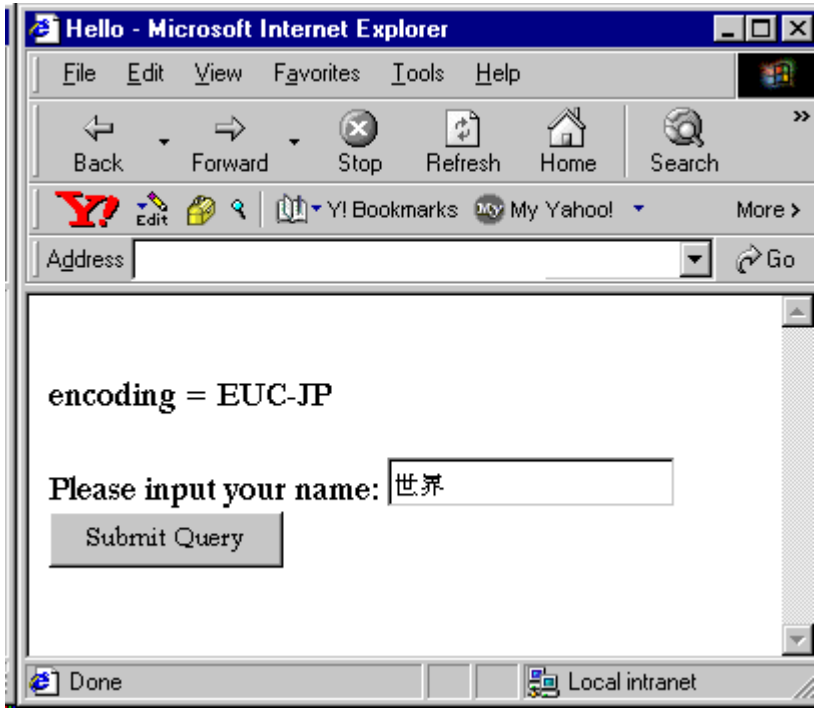
次のサンプルは、日本語文字のユーザー名を受け入れ、そのユーザー名を再度ブラウザに正しく出力します。マルチバイト・リクエスト・パラメータをエンコードできないサーブレット環境では、このサンプルは、translate_params=true という JSP 構成設定に依存します。

XXYY はパラメータ名（日本語で「ユーザー名」に相当するもの）で、AABB はデフォルト値（同様に、日本語での値）であるとします。

（translate_params と同等に機能するコードを含むため、translate_params の設定に依存しない例については、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、入力例を示します。


Hello - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Y! Bookmarks My Yahoo! More
Address
Go
encoding = EUC-JP
Please input your name:
世界
Submit Query
Done
Local intranet

次に、出力例を示します。

translate_params に依存しないグローバリゼーションのサンプル

次のサンプルは、前述のサンプルと同様に、日本語文字のユーザー名を受け入れ、そのユーザー名を再度ブラウザに正しく出力します。ただし、この例は、`translate_params` と同等に機能するコードを含んでいるため、`translate_params` の設定に依存しません。

重要： `translate_params` と同等のコードを使用する場合も、`translate_params` フラグを有効にしないでください。有効にすると、不適切な結果となります。

XXYY はパラメータ名（日本語で「ユーザー名」に相当するもの）で、AABB はデフォルト値（同様に、日本語での値）であるとします。

このサンプルに使用されている重要なコードについては、8-7 ページの「translate_params 構成パラメータと同等のコード」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>
```

9

Apache JServ での Oracle JSP

Oracle9*i* リリース 2（9.2）には、Apache JServ サーブレット環境が含まれています。この環境を使用する場合、すべての Servlet 2.0 環境の場合と同様に、サーブレットおよび JSP の使用方法についての特別な考慮点があります。

この章の内容は次のとおりです。

JServ 環境を使用する前に

この項では、JSP ページを実行するための JServ の構成について説明します。この項の内容は次のとおりです。

- Oracle JSP 用の必須ファイルおよびオプションのファイル
- JServ における Web サーバーの CLASSPATH へのファイルの追加
- JServ 用の JSP ファイル名拡張子のマッピング
- Oracle JSP の構成パラメータ
- JServ での JSP パラメータの設定

Oracle JSP 用の必須ファイルおよびオプションのファイル

この項では、Oracle JSP コンテナを使用するために必要な JAR ファイルおよび ZIP ファイルについて説明します。また、Oracle JDBC や Oracle SQLJ の機能、JML や SQL のカスタム・タグ、またはカスタム・データ・アクセス JavaBeans を使用するためのオプションの JAR ファイルおよび ZIP ファイルについても説明します。

必須ファイルは、CLASSPATH に追加する必要があります（9-5 ページの「JServ における Web サーバーの CLASSPATH へのファイルの追加」を参照）。

> **注意：** サーブレット環境用のサーブレット・ライブラリが、システムにインストールされており、Web サーバー構成ファイル内の CLASSPATH に含まれている必要があります。このライブラリには、標準の `javax.servlet.*` パッケージが含まれています。

次のファイルは、Oracle9*i* リリース 2（9.2）に添付されており、システムにインストールされている必要があります。

- `ojsp.jar`（Oracle JSP コンテナ）
- `xmlparserv2.jar`（XML 解析用で、Servlet 2.2 環境の `web.xml` デプロイメント・ディスクリプタ、およびすべてのタグ・ライブラリ・ディスクリプタに必要）
- `servlet.jar`（標準サーブレット・ライブラリ、Servlet 2.2）

また、JSP ページが Oracle JSP Markup Language（JML）タグ、SQL タグまたはデータ・アクセス JavaBeans を使用する場合は、次のファイルが必要です。

- `ojsputil.jar`（Oracle JSP ユーティリティ・ライブラリ）
- JDK 1.2.x 用の `xsu12.jar` または JDK 1.1.x 用の `xsu111.jar`（XML 機能用）

クライアント環境で実行するには、データ・アクセス JavaBeans で XML 機能を使用する場合（結果セットを XML 文字列として取得する場合など）のみ、`xsu12.jar` または `xsu111.jar` が必要です。`xsu12.jar` ファイルおよび `xsu111.jar` ファイルは、Oracle9*i* に含まれています。

Oracle データ・アクセスには、次のファイルも必要です。

- Oracle JDBC クラス・ファイル（すべての Oracle データ・アクセス用）
- Oracle SQLJ クラス・ファイル（JSP ページで SQLJ コードを使用する場合）

詳細は、9-4 ページの「JDBC 用のファイル（オプション）」および 9-4 ページの「SQLJ 用のファイル（オプション）」を参照してください。

JDBC データ・ソースまたは Enterprise JavaBeans を使用するには、次のファイルが必要です。

- `jndi.jar`

（このファイルは、Oracle JSP デモの一部に必要です。）

サーブレット・ライブラリの注意事項 Oracle JSP リリース 1.1.x.x では、必要なサーブレット・ライブラリのバージョン 2.2 が提供されています。バージョン 2.2 には、標準の `javax.servlet.*` パッケージが格納されています。Web サーバー環境では、多くの場合、サーブレット・ライブラリの別のバージョンが提供されています。CLASSPATH 内では、Web サーバー用のバージョンを Oracle JSP コンテナ用のバージョンより前に設定してください。詳細は、9-5 ページの「JServ における Web サーバーの CLASSPATH へのファイルの追加」を参照してください。

表 9-1 に、サーブレット・ライブラリのバージョンを示します。Sun 社の JavaServer Web Developer's Kit（JSWDK）を Sun 社の Java Servlet Developer's Kit（JSDK）と混同しないでください。

表 9-1 サーブレット・ライブラリのバージョン

サーブレット・ライブラリのバージョン	ライブラリ・ファイル名	そのバージョンが提供される製品
Servlet 2.2	`servlet.jar`	Oracle JSP、Tomcat 3.1
Servlet 2.1	`servlet.jar`	Sun 社の JSWDK 1.0
Servlet 2.0	`jsdk.jar`	Sun 社の JSDK 2.0（JServ でも使用）

（JServ の場合は、`jsdk.jar` を個別にダウンロードしてください。）

JDBC 用のファイル（オプション）データ・アクセスに Oracle JDBC を使用する場合は、次のファイルが必要です。（Oracle SQLJ は Oracle JDBC を使用することに注意してください。）

- `ojdbc14.jar` または `ojdbc14.zip`（JDK 1.4 環境用）

または

- `classes12.jar` または `classes12.zip`（JDK 1.2 または 1.3 環境用）

または

- `classes111.jar` または `classes111.zip`（JDK 1.1 環境用）

SQLJ 用のファイル（オプション）JSP ページがデータ・アクセスに Oracle SQLJ を使用する場合は、次のファイルが必要です。

- `translator.jar` または `translator.zip`（SQLJ トランスレータ用、JDK 1.2.x または 1.1.x 用）

また、次の該当する SQLJ ランタイムのファイルも必要です。

- `runtime12.jar` または `runtime12.zip`（Oracle9*i* JDBC を含む JDK 1.2.x 用）

または

- `runtime12ee.jar` または `runtime12ee.zip`（Oracle9*i* JDBC を含む JDK 1.2.x Enterprise Edition 用）

または

- `runtime11.jar` または `runtime11.zip`（Oracle9*i* JDBC を含む JDK 1.1.x 用）

または

- `runtime.jar` または `runtime.zip`（より一般的 : 任意の Oracle JDBC バージョンを含む JDK 1.2.x または 1.1.x 用）

または

- `runtime-nonoracle.jar` または `runtime-nonoracle.zip`（一般的 : Oracle JDBC Drivers 以外のドライバおよび任意の JDK 環境用）

（J2EE 1.2 以上では、ISO の SQLJ 仕様に従って、データ・ソース・サポートが可能です。）

JServ における Web サーバーの CLASSPATH へのファイルの追加

JServ 環境で Web サーバーの CLASSPATH にファイルを追加するには、適切な wrapper.classpath コマンドを JServ の conf ディレクトリにある jserv.properties ファイルに挿入します。jsdk.jar がすでに CLASSPATH に含まれている必要があることに注意してください。このファイルは、Sun 社の JSDK 2.0 に含まれており、JServ に必要な javax.servlet.* パッケージの Servlet 2.0 を提供します。また、JDK 環境用のファイルも、すでに CLASSPATH に含まれている必要があります。

次の例（UNIX ディレクトリ・パスを使用）には、JSP、JDBC および SQLJ 用のファイルが含まれています。[Oracle_Home] は、Oracle ホーム・パスに置き換えてください。

```
# servlet 2.0 APIs (required by JServ, from Sun JSDK 2.0):
wrapper.classpath=jsdk2.0/lib/jsdk.jar
#
# servlet 2.2 APIs (required and provided by OC4J):
wrapper.classpath=[Oracle_Home]/ojsp/lib/servlet.jar
# JSP 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：

- servlet.jar（javax.servlet.* パッケージの Servlet 2.2 用に Oracle JSP に付属）が JServ 環境の CLASSPATH に含まれている場合、それより前に jsdk.jar が設定されている必要があります。
- Oracle JSP コンテナが javac（または、javaccmd 構成パラメータの設定によっては、別の Java コンパイラ）を検出できることを確認する必要もあります。JDK 1.1.x 環境の javac の場合、JDK の classes.zip ファイルが Web サーバーの CLASSPATH に含まれている必要があります。また、JDK 1.2.x 以上の環境の javac では、JDK の tools.jar ファイルが Web サーバーの CLASSPATH に含まれている必要があります。

次の useBean コマンドを使用する例について考えてみます。

```
<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。（この例では、Windows NT 環境の場合を示しています。）

```
wrapper.classpath=D:¥Apache¥Apache1.3.9¥beans¥
```

次に、JDBCQueryBean.class を次のとおり設定する必要があります。

```
D:¥Apache¥Apache1.3.9¥beans¥mybeans¥JDBCQueryBean.class
```

JServ 用の JSP ファイル名拡張子のマッピング

JServ 環境では、各 JSP ファイル名拡張子を oracle.jsp.JspServlet（JServ 用の JSP フロントエンド・サーブレット）にマップするには、jserv.conf ファイルまたは mod_jserv.conf ファイルで ApJServAction コマンドを使用する必要があります。これらの構成ファイルは、JServ の conf ディレクトリに格納されています。

（以前のバージョンでは、かわりに、Apache の conf ディレクトリに格納されている httpd.conf ファイルを更新する必要がありました。新しいバージョンでは、実行時に jserv.conf または mod_jserv.conf ファイルが httpd.conf に格納されます。httpd.conf ファイルを参照して、どちらのファイルが格納されているかを確認してください。）

次に、UNIX 構文を使用した場合の例を示します。

```
# 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 に対して使用するパスは、ファイル・システムの実際のディレクトリ・パスではありません。指定するパスは、JServ サーブレットの構成（サーブレット・ゾーンのマウント方法、ゾーン・プロパティ・ファイルの名前、およびサーブレットのリポジトリとして指定されたファイル・システム・ディレクトリ）によって異なります。（「サーブレット・ゾーン」は、概念的に「サーブレット・コンテキスト」とほぼ同義の JServ 用語です。）詳細は、JServ のドキュメントを参照してください。

重要： 前述の構成を使用する場合、Oracle JSP はファイル名拡張子 .jsp または .JSP を使用するページ参照をサポートしますが、大 / 小文字を区別する環境では、参照内の大 / 小文字が実際のファイル名と一致している必要があります。ファイル名が file.jsp の場合、そのファイルは file.jsp として参照することはできますが、file.JSP としては参照できません。また、ファイル名が file.JSP の場合は、file.JSP として参照することはできますが、file.jsp としては参照できません。（.sqljsp と .SQLJSP の場合も同様です。）

Oracle JSP の構成パラメータ

この項では、Oracle JspServlet がサポートする構成パラメータについて説明します。

構成パラメータの一覧表

表 9-2 に、Oracle JspServlet（Oracle JSP コンテナのフロントエンド）がサポートする構成パラメータを示します。この表では、パラメータごとに次のことを示します。

- そのパラメータがページの変換時または実行時のいずれで使用されるか
- 通常、そのパラメータが開発環境または配置環境（あるいはその両方）のいずれで使用されるか
- 事前変換するページに対する同等の ojspc 変換オプション

  （ojspc ユーティリティは、JspServlet を使用しません。）

次のことに注意してください。

- debug_mode および send_error は、Oracle JSP リリース 1.1.2.x での新しいパラメータです。
- alias_translation パラメータは、JServ 環境でのみ使用されます。
- session_sharing パラメータは、（JServ などの Servlet 2.0 環境などで）globals.jsa でのみ使用されます。

注意：

- ojspc のオプションについては、6-14 ページの「ojspc 事前変換ツールの詳細」を参照してください。
- 実行時と変換時は、リアルタイム変換環境では特に大きな違いはありませんが、事前変換時にその違いが顕著になる場合があります。

表 9-2 Oracle JSP の構成パラメータ

パラメータ	関連する ojspc オプション	説明	デフォルト	使用時期	使用環境
alias_translation **(Apache 固有)**	なし	ブール値。JSP ページ参照に対するディレクトリ別名の JServ 制限を回避するには、`true` を指定します。	false	実行時	開発および配置環境
bypass_source	なし	ブール値。Oracle JSP コンテナで `.jsp` ソースに対する `FileNotFound` 例外を無視するには、`true` を指定します。ソースを使用できない場合は、事前変換されたコンパイル済のコードを使用します。	false	実行時	配置環境（JDeveloper でも使用）
classpath	-addclasspath（関連性はあるが、異なる機能）	Oracle JSP クラスをロードするための追加の CLASSPATH エントリ。	null（追加のパスなし）	変換時または実行時	開発および配置環境
debug_mode	なし	ブール値。実行時例外の発生時に Oracle JSP コンテナでスタック・トレースを出力するには、`true` を指定します。	true	実行時	開発環境
developer_mode	なし	ブール値。ページがリクエストされたときに、タイムスタンプを調べて、ページの再変換およびクラスの再ロードが必要かどうかを確認しない場合は、`false` を指定します。	true	実行時	開発および配置環境
emit_debuginfo	-debug	ブール値。開発中にデバッグ用に元の `.jsp` ファイルへの行マッピングを生成するには、`true` を指定します。	false	変換時	開発環境
external_resource	-extres	ブール値。Oracle JSP コンテナで変換時にページのすべての静的コンテンツを個別の Java リソース・ファイルに置くには、`true` を指定します。	false	変換時	開発および配置環境

表 9-2 Oracle JSP の構成パラメータ（続き）

パラメータ	関連する ojspc オプション	説明	デフォルト	使用時期	使用環境
javaccmd	-noCompile	Java コンパイラ・コマンドライン（javac のオプション）、または別の JVM で実行される代替 Java コンパイラ（デフォルトのオプションを含む JDK の javac の場合は null）。	null	変換時	開発および配置環境
page_repository_root	-srcdir -d	Oracle JSP コンテナが、JSP ページをロードおよび生成する際に使用する代替ルート・ディレクトリ（完全修飾されたパス）。	null（デフォルトのルートを使用）	変換または実行で使用	開発および配置環境
send_error	なし	ブール値。ファイルが見つからない場合は標準の「404」メッセージを出力し、コンパイル・エラーが発生した場合は標準の「500」メッセージを出力する（カスタマイズしたメッセージを出力しない）には、true を指定します。	false	実行時	配置環境
session_sharing **（Servlet 2.0 環境で globals.jsa とともに使用）**	なし	ブール値。globals.jsa を使用するアプリケーションで、JSP セッション・データを基礎となるサーブレット・セッションに伝播するには、true を指定します。	true	実行時	開発および配置環境

表 9-2 Oracle JSP の構成パラメータ（続き）

パラメータ	関連する ojspc オプション	説明	デフォルト	使用時期	使用環境
sqljcmd	-S	SQLJ コマンドライン（sqlj のオプション）、または別の JVM で実行される代替 SQLJ トランスレータ（Oracle9*i* に添付のデフォルトのオプション設定を含む Oracle SQLJ バージョンの場合は null）。	null	変換時	開発および配置環境
translate_params	なし	ブール値。マルチバイト・エンコーディングを実行しない Servlet コンテナをオーバーライドするには、true を指定します。	false	実行時	開発および配置環境
unsafe_reload **（開発のみで使用）**	なし	ブール値。JSP ページが再変換および再ロードされるたびにアプリケーションおよびセッションを再起動しない場合は、true を指定します。	false	実行時	開発環境

構成パラメータの説明

この項では、Oracle JSP 構成パラメータの詳細を説明します。

alias_translation（ブール値。デフォルトは false。）**（Apache 固有）**

このパラメータを使用すると、Oracle JSP コンテナは、JServ によるディレクトリ別名の処理方法に対する制限を回避できます。現行の制限については、9-21 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）が JServ サーブレット環境で正常に機能するには、alias_translation を true に設定する必要があります。

```
Alias /icons/ "/apache/apache139/icons/"
```

bypass_source（ブール値。デフォルトは false。）

通常、JSP ページがリクエストされると、Oracle JSP コンテナは、ページ実装クラスを検出できた場合でも、対応する .jsp ソース・ファイルを検出できなかった場合は、FileNotFound 例外を発生させます。（これは、デフォルトでは、JSP コンテナがページ・ソースを参照して、ページ実装クラスの生成後にページ・ソースが変更されたかどうかを確認するためです。）

ページ・ソースを検出できない場合でも、Oracle JSP コンテナの処理を続行し、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。

JSP コンテナは、`bypass_source` が有効にされている場合でも、ソースが使用可能で必要な場合は、再変換が必要かどうかを確認します。ソースが必要かどうかを判断する要因の 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、生成されたクラスのみが含まれ、ソースが含まれていない配置環境の場合に有効です（関連情報については、6-27 ページの「バイナリ・ファイルのみの配置」を参照）。
- Oracle9*i* JDeveloper は、JSP ソースをファイルに保存する前に JSP ページを変換および実行できるように、`bypass_source` を有効にします。

---

classpath（完全修飾されたパス。デフォルトは `null`。）

このパラメータは、JSP ページの変換、コンパイルまたは実行中に使用する Oracle JSP のデフォルトの CLASSPATH に CLASSPATH エントリを追加する場合に使用します。Oracle JSP の CLASSPATH およびクラス・ローダーについては、4-19 ページの「CLASSPATH およびクラス・ローダーの問題」を参照してください。

正確な構文は、Web サーバー環境およびオペレーティング・システムによって異なります。例については、9-17 ページの「JServ での JSP パラメータの設定」を参照してください。

概して、Oracle JSP コンテナは、独自の CLASSPATH（この `classpath` パラメータのエントリを含む）、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・ディレクトリからの相対的な事前定義済の位置からクラスをロードします。

`classpath` 設定パスで指定され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・ローダーによってロードされることに注意してください。JSP の実行中、JSP のクラス・ローダーによってロードされたクラスとシステムのクラス・ローダーまたは他のクラス・ローダーによってロードされたクラスの間で、アクセスを行うことはできません。

注意：

- Oracle JSP の実行時自動クラス再ロードは、Oracle JSP の CLASSPATH に含まれるクラスのみに適用されます。これには、この `classpath` パラメータを介して指定されたパスが含まれます（この機能については、4-24 ページの「クラスの動的な再ロード」を参照）。
- ページを事前変換する場合は、`ojspc` の `-addclasspath` オプションによって、関連性のある異なる機能が提供されます。6-18 ページの「ojspc のオプションの説明」を参照してください。

debug_mode（ブール値。デフォルトは `true`。）

ランタイム例外が発生するたびに、Oracle JSP コンテナがスタック・トレースを出力するようにするには、このフラグをデフォルトの `true` に設定します。この機能を無効にするには、このフラグを `false` に設定します。

developer_mode（ブール値。デフォルトは `true`。）

ページがリクエストされるたびに、Oracle JSP コンテナが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比較しないようにするには、このフラグを `false` に設定します。`developer_mode` が `true` に設定されている場合、Oracle JSP コンテナは、ページがリクエストされるたびに、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を確認します。ソースが変更された場合、JSP コンテナはページを再変換します。`developer_mode` が `false` に設定されている場合、JSP コンテナは、最初のページ・リクエスト時またはアプリケーション・リクエスト時にのみ確認を行います。その後のリクエストに対しては、生成されたページ実装クラスを再実行するのみです。

このフラグは、JSP ページによってコールされた JavaBeans および他のサポート・クラスの動的クラス再ロードにも影響します。`developer_mode` が `true` に設定されている場合、Oracle JSP コンテナは、このようなクラスが Oracle JSP のクラス・ローダーによってロードされた後に変更されたかどうかを確認します。

特に、コードが変更される可能性が低く、パフォーマンスが重要な問題である配置環境では、通常、`developer_mode` を `false` に設定することをお薦めします。

4-22 ページの「Oracle JSP による実行時のページおよびクラスの再ロード」も参照してください。

emit_debuginfo（ブール値。デフォルトは `false`。）

Oracle JSP コンテナが、開発時のデバッグ操作用に元の `.jsp` ファイルへの行マッピングを生成するようにするには、このフラグを `true` に設定します。`false` に設定した場合、行は生成されたページ実装クラスにマップされます。

注意：

- Oracle9*i* JDeveloper は、`emit_debuginfo` を有効にします。
- ページを事前変換する場合、`ojspc` の `-debug` オプションが同等の機能を提供します。6-18 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（ブール値。デフォルトは `false`。）

Oracle JSP トランスレータで、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・メソッド内ではなく Java ソース・ファイル内に配置するには、このフラグを `true` に設定します。

リソース・ファイル名は、JSP ページ名に基づいており、`.res` 接尾辞が付いています。Oracle9*i* では、たとえば、`MyPage.jsp` を変換すると、通常の出力の他に、`_MyPage.res` が作成されます。ただし、厳密な実装は、将来のリリースで変更される可能性があります。

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配置されます。

ページ内に多くの静的コンテンツが含まれている場合、この方法によって変換が高速化されます。また、ページの実行が高速化される場合もあります。場合によっては、サービス・メソッドが、JVM における 64KB のメソッド・サイズ制限を超えることも防止できます。詳細は、4-10 ページの「JSP ページに大量の静的コンテンツが含まれている場合の対策」を参照してください。

注意： ページを事前変換する場合、`ojspc` の `-extres` オプションが同等の機能を提供します。

javaccmd（コンパイラの実行ファイル。デフォルトは `null`。）

このパラメータは、次のいずれの場合にも有効です。

- デフォルト設定が一般的に十分な値であるにもかかわらず、`javac` のコマンドライン・オプションを設定する場合
- `javac`（オプションで、コマンドライン・オプションを含む）以外のコンパイラを使用する場合
- Oracle JSP コンテナとは別のプロセスで Java コンパイラを実行する場合

代替コンパイラを指定すると、Oracle JSP コンテナは、自身が実行している JVM 内で JDK のデフォルトのコンパイラを起動するのではなく、別の JVM 内で別のプロセスとしてその実行ファイルを起動します。実行ファイルのパスを完全に指定するか、または実行ファイルのみを指定して、Oracle JSP コンテナにシステム・パス内でそのファイルを検索させることができます。

次に、javac -verbose オプションを有効にするための javaccmd 設定の例を示します。

```
javaccmd=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。9-17 ページの「JServ での JSP パラメータの設定」を参照してください。

注意：

- 指定した Java コンパイラが CLASSPATH に含まれており、フロントエンド・ユーティリティ（該当する場合）がシステム・パスに含まれている必要があります。
- ページを事前変換する場合、ojspc の -noCompile オプションが同様の機能を提供します。その結果、javac によるコンパイルが行われないため、変換されたクラスを、適切なコンパイラを使用して手動でコンパイルできます。6-18 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（完全修飾されたディレクトリ・パス。デフォルトは null。）

Oracle JSP コンテナは、Web サーバーのドキュメント・リポジトリを使用して、変換された JSP ページを生成またはロードします。デフォルトでは、オンデマンド変換の場合、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（JServ の場合）、またはページが属しているアプリケーションのサーブレット・コンテキスト・ルート・ディレクトリです。JSP ページ・ソースは、ルート・ディレクトリまたは特定のサブディレクトリにあります。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特定のサブディレクトリに書き込まれます。

Oracle JSP コンテナで、別のルート・ディレクトリを使用するには、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 サブディレクトリからのファイルの相対位置については、6-7 ページの「Oracle JSP トランスレータによる出力ファイルの格納場所」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、およびこれらのディレクトリの下にある適切なサブディレクトリが存在しない場合は、自動的に作成されます。
- ページを事前変換する場合、ojspc の -srcdir オプションおよび -d オプションが関連機能を提供します。6-18 ページの「ojspc のオプションの説明」を参照してください。

send_error（ブール値。デフォルトは `false`。）

Oracle JSP コンテナで、ファイルが見つからない場合は標準の「404」メッセージを出力し、コンパイル・エラーが発生した場合は標準の「500」メッセージを出力するには、このフラグを `true` に設定します。

これは、より詳細な情報（見つからないファイルの名前など）を提供するカスタマイズされたメッセージの出力と相反するものです。JServ などの一部の環境では、「404」または「500」メッセージが出力される場合、カスタマイズしたメッセージは出力されません。

session_sharing（ブール値。デフォルトは `true`。）**（globals.jsa とともに使用）**

Servlet 2.0 環境で `globals.jsa` ファイルがアプリケーションに対して使用されている場合、各 JSP ページは、Servlet コンテナが提供する単一のサーブレット・セッション・オブジェクト全体に連結された個別の JSP セッション・ラッパーを使用します。

この場合、`session_sharing` パラメータをデフォルトの `true` に設定すると、JSP のセッション・データが基礎となるサーブレット・セッションに伝播されます。これによって、アプリケーションのサーブレットは、その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 実装で標準動作をパラレル化します）、JSP のセッション・データはサーブレット・セッションに伝播されません。そのため、アプリケーションのサーブレットは、JSP のセッション・データにアクセスできません。

`globals.jsa` を使用しない場合、このパラメータは意味がありません。`globals.jsa` については、9-23 ページの「Oracle JSP アプリケーションおよび JServ でのセッション・サポート」を参照してください。

sqljcmd（SQLJ トランスレータの実行ファイルおよびオプション。デフォルトは `null`。）

このパラメータは、次のいずれの場合にも有効です。

- SQLJ の 1 つ以上のコマンドライン・オプションを設定する場合

  （`sqljcmd` 設定で複数の SQLJ オプションを設定できます。）
- Oracle9*i* に付属のものとは異なる SQLJ トランスレータ（または異なるバージョン）を使用する場合
- Oracle JSP コンテナとは別のプロセスで SQLJ を実行する場合

SQLJ トランスレータの実行ファイルを指定すると、JSP コンテナは、自身が実行している JVM 内でデフォルトの SQLJ トランスレータを起動するのではなく、別の JVM 内で別のプロセスとしてその実行ファイルを起動します。

実行ファイルのパスを完全に指定するか、または実行ファイルのみを指定して、JSP コンテナにシステム・パス内でそのファイルを検索させることができます。

次に、オンライン・セマンティクス・チェックのために scott/tiger でログインし、ISO 規格の SQLJ コードを生成するための sqljcmd 設定の例を示します。

```
sqljcmd=sqlj -user=scott/tiger -codegen=iso
```

（正確な構文は、サーブレット環境によって異なります。9-17 ページの「JServ での JSP パラメータの設定」を参照してください。）

注意：

- 適切な SQLJ ファイルが CLASSPATH に含まれており、フロントエンド・ユーティリティ（例に示す sqlj など）がシステム・パスに含まれている必要があります。（Oracle SQLJ の場合、translator という ZIP ファイルまたは JAR ファイル、および適切な SQLJ ランタイムの ZIP ファイルまたは JAR ファイルが CLASSPATH に含まれている必要があります。9-2 ページの「Oracle JSP 用の必須ファイルおよびオプションのファイル」を参照してください。
- JSP ページで SQLJ コードを使用する場合、ほとんどの JSP 開発者は（他の SQLJ 製品ではなく）Oracle SQLJ を使用すると想定されます。ただし、このオプションは、別の Oracle SQLJ バージョン（Oracle9*i* ドライバではなく、Oracle JDBC Drivers 8.0.x/7.3.x 向けのバージョンなど）を使用する場合、または SQLJ オプションを設定する場合に有効です。
- ページを事前変換する場合、ojspc の -S オプションが関連機能を提供します。6-18 ページの「ojspc のオプションの説明」を参照してください。

---

translate_params（ブール値。デフォルトは false。）

注意： Oracle JSP リリース 1.1.2.x 以上では、translate_params パラメータではなく、PublicUtil.setReqCharacterEncoding() メソッドを使用することをお薦めします。8-5 ページの「setReqCharacterEncoding() メソッド」を参照してください。

マルチバイト（グローバリゼーション・サポート）・リクエスト・パラメータまたは Bean プロパティの設定をエンコードしない Servlet コンテナをオーバーライドするには、このフラグを true に設定します。true に設定すると、Oracle JSP コンテナは、リクエスト・パラメータおよび Bean プロパティの設定をエンコードします。false に設定すると、JSP コンテナは、Servlet コンテナから変更されないままのパラメータを戻します。

translate_params の機能および使用の詳細（このパラメータを使用できない場合など）は、8-5 ページの「Oracle JSP によるマルチバイト・パラメータ・エンコーディングの拡張サポート」を参照してください。

unsafe_reload（ブール値。デフォルトは false。）**（開発環境のみで使用）**

デフォルトでは、Oracle JSP コンテナは、JSP ページが動的に再変換および再ロードされるたびに、アプリケーションおよびセッションを再起動します（JSP トランスレータが、対応するページ実装クラスより後のタイムスタンプが付いた .jsp ソース・ファイルを検出した場合に実行されます）。

JSP コンテナが、動的な再変換および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するには、このパラメータを true に設定します。これによって、既存のセッションが無効になることが回避されます。

developer_mode を false に設定した場合、指定された JSP ページが最初にリクエストされた後、このパラメータは無効になります（この場合、JSP コンテナは最初のリクエスト後に再変換を行いません）。

重要： このパラメータは開発者専用であり、配置環境には使用しないでください。

JServ での JSP パラメータの設定

JServ 環境の各 Web アプリケーションには、「ゾーン・プロパティ・ファイル」という独自のプロパティ・ファイルが含まれています。Apache 用語では、「ゾーン」は基本的に「サーブレット・コンテキスト」と同じ意味です。

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異なります。（ゾーンおよびマウントについては、JServ のドキュメントを参照してください。）

JServ 環境で JSP の構成パラメータを設定するには、アプリケーション・ゾーン・プロパティ・ファイルの JspServlet initArgs プロパティを、次の例（UNIX 構文を使用）に示すとおり設定します。

```
servlet.oracle.jsp.JspServlet.initArgs=developer_mode=false,
sqljcmd=sqlj -user=scott/tiger -codegen=iso,classpath=/mydir/myapp.jar
```

（これは、実際には 1 行です。）

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異なります。このパスは、実際のディレクトリ・パスではありません。

次のことに注意してください。

- 複数の initArgs コマンドを使用した場合の結果は累積およびオーバーライドされます。たとえば、次の 2 つのコマンド（順序どおり）について考えてみます。

  ```
  servlet.oracle.jsp.JspServlet.initArgs=foo1=val1,foo2=val2
  servlet.oracle.jsp.JspServlet.initArgs=foo1=val3
  ```

  この 2 つのコマンドは、次の 1 つのコマンドと同等です。

  ```
  servlet.oracle.jsp.JspServlet.initArgs=foo1=val3,foo2=val2
  ```

  最初の 2 つのコマンドでは、foo1 の値 val3 が値 val1 をオーバーライドしますが、foo2 の設定には影響しません。

- initArgs パラメータはカンマで区切られているため、パラメータの設定内ではカンマを使用できません。ただし、空白およびその他の特殊文字（この例に示す「=」など）は、使用しても問題ありません。

JServ サーブレット環境についての考慮点

JServ ベースのプラットフォームで Oracle JSP コンテナを実行する場合、JServ が Servlet 2.0 環境であるため、特別な考慮点があります。Servlet 2.0 仕様は、Servlet 2.1 および 2.2 環境では使用可能ないくつかの重要な機能をサポートしていません。

Oracle JSP コンテナ用に JServ 環境を構成する方法については、次の項を参照してください。

- 9-5 ページの「JServ における Web サーバーの CLASSPATH へのファイルの追加」
- 9-6 ページの「JServ 用の JSP ファイル名拡張子のマッピング」
- 9-17 ページの「JServ での JSP パラメータの設定」

この項では、次の Apache 固有の考慮点について説明します。

- JServ での動的なインクルードおよびフォワード
- JServ 用のアプリケーション・フレームワーク
- JSP とサーブレットのセッション共有
- ディレクトリ別名の変換

JServ での動的なインクルードおよびフォワード

JSP の動的なインクルード（jsp:include アクション）およびフォワード（jsp:forward アクション）は、リクエスト・ディスパッチャ機能に依存します。この機能は、Servlet 2.1 および 2.2 環境には存在しますが、Servlet 2.0 環境には存在しません。

ただし、Oracle JSP コンテナを使用すると、JServ および他の Servlet 2.0 環境で 1 つの JSP ページから別の JSP ページまたは静的 HTML ファイルへの動的なインクルードおよびフォワードを可能にする拡張機能が提供されます。

ただし、この Servlet 2.0 環境用の Oracle JSP 機能では、サーブレットへの動的なフォワードまたはインクルードを行うことはできません。（サーブレットの実行は、JSP コンテナではなく、JServ または他の Servlet コンテナによって制御されます。）

JServ でサーブレットへのインクルードまたはフォワードを行う場合は、そのサーブレットのラッパーとして機能する JSP ページを作成できます。

次の例では、サーブレット、およびそのサーブレットのラッパーとして機能する JSP ページを示します。JServ 環境では、JSP ラッパー・ページにインクルードまたはフォワードすることによって、サーブレットに効果的にインクルードまたはフォワードできます。

サーブレットのコード 次のサーブレットにインクルードまたはフォワードするとします。

```
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 ラッパー・ページのコード 前述のサーブレット用に次の 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); %>
```

Servlet 2.0 環境で wrapper.jsp へのインクルードまたはフォワードを行うと、Servlet 2.1 または 2.2 環境で直接 TestServlet にインクルードまたはフォワードする場合と同じ結果になります。

動的なインクルードおよびフォワードについての注意事項

- ラッパーである JSP ページで isThreadSafe を true または false のどちらに設定するかは、元のサーブレットがスレッド・セーフかどうかに依存します。
- ラッパーである JSP ページを使用するかわりに、HTTP クライアント・コードを元の JSP ページ（include または forward の発生元である JSP ページ）に追加できます。標準の java.net.URL クラスのインスタンスを使用して、元の JSP ページからサーブレットへの HTTP リクエストを作成できます（この場合、セッション・データまたはセキュリティの資格証明を共有できないことに注意してください）。

JServ 用のアプリケーション・フレームワーク

Servlet 2.0 仕様では、Servlet 2.1 以上の仕様で提供される、アプリケーション・サポート用の完全なサーブレット・コンテキスト・フレームワークは提供されません。

Oracle JSP コンテナでは、JServ などの Servlet 2.0 環境用に、アプリケーション・マーカーとして使用できる `globals.jsa` というファイルを使用した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。

詳細は、9-25 ページの「globals.jsa を使用した個別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。

JSP とサーブレットのセッション共有

JServ 環境で JSP ページとサーブレットの間で HTTP セッション情報を共有するには、`oracle.jsp.JspServlet`（Oracle JSP コンテナのフロントエンドとして機能するサーブレット）が、JSP ページとセッションを共有する 1 つ以上のサー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。

ゾーン設定が適切であるかどうかを確認するために、一部のブラウザでは Cookie に対する警告を有効にできます。Apache 環境では、Cookie 名には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、サーブレットが JSP のセッション・データにアクセスできるようにするために、JSP 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、次の項を参照してください。

- 9-23 ページの「Oracle JSP アプリケーションおよび JServ でのセッション・サポート」
- 9-7 ページの「Oracle JSP の構成パラメータ」
- 9-17 ページの「JServ での JSP パラメータの設定」

ディレクトリ別名の変換

Apache 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」を作成可能にすることによって、ディレクトリ別名をサポートします。これによって、Web ドキュメントをデフォルトのドキュメント・ルート・ディレクトリの外に配置することができます。

次の `httpd.conf` エントリの例について考えてみます。

```
Alias /icons/ "/apache/apache139/icons/"
```

このコマンドを実行すると、icons が /apache/apache139/icons/ パスの別名として使用可能になります。この方法で、たとえば、/apache/apache139/icons/art.gif ファイルを次の URL によってアクセス可能にできます。

```
http://host[:port]/icons/art.gif
```

ただし、現在は、JServ の getRealPath() メソッドが別名ディレクトリの下にあるファイルの処理時に不適切な値を戻すため、この機能はサーブレットおよび JSP ページに対して正常に機能しません。

Oracle では、Apache 固有の JSP 構成パラメータである alias_translation が提供されます。alias_translation を true（デフォルト設定は false）に設定すると、この制限が回避されます。

alias_translation=true を設定すると、別名ディレクトリがアプリケーション・ルートになることに注意してください。そのため、ターゲット・ファイルの名前が「/」で始まる動的な include または forward コマンドでは、予期されるターゲット・ファイルの位置が別名ディレクトリからの相対位置になります。

/private/foo の下にあるすべての JSP および HTML ファイルをアプリケーション /mytest の下に効果的に配置する次の例について考えてみます。

```
Alias /mytest/ "/private/foo/"
```

また、次のとおり配置された JSP ページが存在するとします。

```
/private/foo/xxx.jsp
```

事実上、アプリケーション・ルートである別名ディレクトリ /private/foo の直下に xxx.jsp があるため、次の動的な include コマンドは機能します。

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

他のアプリケーションまたは一般的なドキュメント・ルート内の JSP ページは、JSP ページまたは HTML ファイルを /mytest アプリケーションの下にフォワードまたはインクルードできません。（Servlet 2.2 仕様に従って）ページまたは HTML ファイルを同じアプリケーション内でフォワードまたはインクルードすることのみ可能です。

注意：

- Web サーバーのドキュメント・ルートおよび各別名ルート用に、暗黙的アプリケーションが作成されます。
- JServ 環境で JSP の構成パラメータを設定する方法については、9-17 ページの「JServ での JSP パラメータの設定」を参照してください。

また、2 つの別名が同じ部分ディレクトリ・パスで始まる場合、問題となることにも注意してください。次に示す 2 つの別名の例について考えてみます。

```
Alias /foo/bar1 "/path/to/my/dir/x/bar1"
Alias /foo/bar2 "/path/to/my/dir/y/bar2"
```

最初の /foo/bar1/bar1.jsp リクエストは機能します。ただし、その後の /foo/bar2/bar2.jsp リクエストでは、/path/to/my/dir/x での bar2.jsp の検索が不適切に実行され、FileNotFound 例外が発生して検索が失敗します。これは、JServ の getRealPath() 実装でのさらなる制限が原因です。この実装は、不適切な情報を戻します。この場合、2 つの回避方法があります。

- 1 つの別名のみを使用し、その下に実ディレクトリを配置します。

  ```
  Alias /foo  "/path/to/my/dir"
  ```

  ここでは、bar1 および bar2 ディレクトリは、物理的に /path/to/my/dir/bar1 および /path/to/my/dir/bar2 として存在し、問題はありません。

または

- 複数の別名を使用しますが、共通のディレクトリ名は使用しません。

  ```
  Alias /foo/bar1  "/path/to/my/dir/x_bar1"
  Alias /foo/bar2  "/path/to/my/dir/y_bar2"
  ```

  物理ディレクトリは別名ディレクトリと同じ名前を持たないことに注意してください（別名ディレクトリと物理ディレクトリが bar1 および bar2 を共有している前述の例とは異なります）。

Oracle JSP アプリケーションおよび JServ でのセッション・サポート

Oracle JSP コンテナは、globals.jsa ファイルを、Servlet 2.0 環境で JSP 仕様を実装するためのメカニズムとして定義します。Servlet 2.0 仕様では、Web アプリケーションおよびサーブレット・コンテキストが完全には定義されていませんでした。

この項では、globals.jsa メカニズムについて説明します。この項の内容は次のとおりです。

- globals.jsa の機能の概要
- globals.jsa の構文およびセマンティクスの概要
- globals.jsa のイベント・ハンドラ
- グローバルな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、9-36 ページの「Servlet 2.0 環境で globals.jsa を使用する例」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 / 小文字を混在させて使用した場合、大 / 小文字を区別しない環境では機能しますが、大 / 小文字を区別する環境にページを移植する場合の問題の診断が困難になります。

---

globals.jsa の機能の概要

すべての単一の Java Virtual Machine 内では、アプリケーションごと（またはサーブレット・コンテキストごと）に globals.jsa ファイルを使用できます。このファイルは、次の場合で Web アプリケーションの概念をサポートします。

- アプリケーションの配置 - アプリケーション・ルートを定義するアプリケーションのロケーション・マーカーとして使用します。
- 個別のアプリケーションおよびセッション - 各アプリケーションに個別のサーブレット・コンテキストおよびセッション・オブジェクトを提供する場合に、Oracle JSP コンテナがこのファイルを使用します。
- アプリケーションのライフ・サイクル管理 - セッションおよびアプリケーションの開始イベントおよび終了イベントを使用します。

globals.jsa ファイルでは、アプリケーションのすべての JSP ページにわたるグローバルな Java 宣言および JSP ディレクティブを実現する手段も提供されます。

globals.jsa を使用したアプリケーションの配置

サーブレットを組み込まない Oracle JSP アプリケーションを配置するには、ディレクトリ構造を Web サーバー内にコピーし、globals.jsa というファイルを作成して、アプリケーション・ルート・ディレクトリに置きます。

globals.jsa ファイルは、サイズが 0（ゼロ）である場合があります。Oracle JSP コンテナがそのファイルを配置し、ディレクトリ内にそのファイルが存在することによって、そのディレクトリが（URL 仮想パスからマップされたとおり）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。

JSP コンテナは、JSP アプリケーション・リソースのデフォルトの位置も定義します。たとえば、アプリケーションから相対的な /WEB-INF/classes ディレクトリおよび /WEB-INF/lib ディレクトリ（Servlet 2.2 以上）に存在するアプリケーションの Bean およびクラスは、特定の構成なしで、Oracle JSP のクラス・ローダーによって自動的にロードされます。

注意： サーブレットを組み込むアプリケーションの場合、特に Servlet 2.1 以下の仕様のサーブレット環境では、すべてのサーブレット配置の場合と同様に、手動の構成が必要です。Servlet 2.2 以上の環境のサーブレットでは、必要な構成を標準の `web.xml` デプロイメント・ディスクリプタに含めることができます。

globals.jsa を使用した個別のアプリケーションおよびセッションのサポート

Servlet 2.0 仕様では、Servlet 2.1 以上の仕様とは異なり、Web アプリケーションの概念が明確に定義されておらず、サーブレット・コンテキストとアプリケーションの関係も定義されていません。JServ などの Servlet 2.0 環境では、サーブレット・コンテキスト・オブジェクトが JVM ごとに 1 つのみ存在します。また、Servlet 2.0 環境では、存在するセッション・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 Servlet 2.0 環境用に、Web サーバーでの複数のアプリケーションおよび複数のセッションをサポートします。

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、アプリケーション用の `globals.jsa` ファイルが存在すると、Oracle JSP コンテナは、その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、セッション・オブジェクトが（1 つのサーブレット・コンテキストまたは複数のサーブレット・コンテキスト全体で）1 つのみ存在する場合、`globals.jsa` ファイルが存在すると、Oracle JSP コンテナによって、アプリケーションにプロキシの `HttpSession` オブジェクトが提供されます。これによって、他のアプリケーションとのセッション変数名の衝突が回避されます。ただし、アプリケーション・データは、他のアプリケーションによって検査および変更される場合があります。これは、`HttpSession` オブジェクトが、その一部の機能について、基礎となるサーブレット・セッション環境に依存する必要があるためです。

globals.jsa を使用したアプリケーションおよびセッションのライフ・サイクル管理

重大な状態遷移が発生した場合、アプリケーションに通知される必要があります。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、セッションの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または終了時に永続データをリストアまたは保存する必要があります。

ただし、標準のサーブレットおよび JSP テクノロジでは、セッションベースのイベントのみがサポートされています。

globals.jsa ファイルを使用するアプリケーションの場合、Oracle JSP コンテナは、この機能を次の 4 つのイベントによって拡張します。

- session_OnStart
- session_OnEnd
- application_OnStart
- application_OnEnd

サーブレットがレスポンスする必要があるこれらのいずれのイベントに対しても、globals.jsa ファイルでイベント・ハンドラを作成できます。

session_OnStart イベントおよび session_OnEnd イベントは、それぞれ HTTP セッションの開始時および終了時にトリガーされます。

application_OnStart イベントは、任意の単一 JVM 内で任意のアプリケーションが最初にリクエストされたときに、そのアプリケーションに対してトリガーされます。application_OnEnd イベントは、Oracle JSP コンテナがアプリケーションをアンロードしたときにトリガーされます。

詳細は、9-28 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 の構文およびセマンティクスの概要

この項では、globals.jsa ファイルの一般的な構文およびセマンティクスの概要を示します。

globals.jsa ファイル内の各イベント・ブロック（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の間にあるすべてのもの）が含まれます。

次の例では、このパターンを示します。

```
<event:session_OnStart>
   <% This scriptlet contains the implementation of the event handler %>
</event:session_OnStart>
```

イベント・ブロックの本体は、すべての妥当な JSP タグ（標準のタグ、およびカスタム・タグ・ライブラリで定義済のタグ）を含むことができます。

ただし、イベント・ブロック内にあるすべての JSP タグの有効範囲は、そのブロックのみに限定されます。たとえば、イベント・ブロック内の jsp:useBean タグで宣言された Bean は、それを使用する他のすべてのイベント・ブロックで再宣言される必要があります。ただし、この制限は、globals.jsa グローバル宣言メカニズム（9-33 ページの「グローバルな宣言およびディレクティブ」を参照）を介して回避できます。

4 つのイベント・ハンドラの詳細は、9-28 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：** 通常の JSP ページで使用される静的テキストは、session_OnStart ブロック内のみに存在できます。session_OnEnd、application_OnStart および application_OnEnd のイベント・ブロックは、Java スクリプトレットのみを含むことができます。

JSP の暗黙的オブジェクトは、globals.jsa イベント・ブロックで次のとおり使用できます。

- application_OnStart ブロックは、application オブジェクトにアクセスできます。
- application_OnEnd ブロックは、application オブジェクトにアクセスできます。
- session_OnStart ブロックは、application、session、request、response、page および out オブジェクトにアクセスできます。
- session_OnEnd ブロックは、application および session オブジェクトにアクセスできます。

完全な globals.jsa ファイルの例 この例では、4 つのすべてのイベント・ハンドラを使用して、完全な globals.jsa ファイルを示します。

```
<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、4 つの globals.jsa イベント・ハンドラの詳細を説明します。

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、Oracle JSP コンテナがアプリケーション内の最初の JSP ページをロードしたときに実行されます。通常、任意のクライアントからアプリケーション内の任意のページに対する最初の HTTP リクエストが行われたときに実行されます。アプリケーションは、このイベントを使用して、アプリケーション全体のリソース（データベース接続プール、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）を初期化します。

このイベント・ハンドラは、JSP タグ（カスタム・タグなど）および空白のみを含む必要があり、静的テキストを含むことはできません。

このイベント・ハンドラで発生したにもかかわらず、イベント・ハンドラ・コードで処理されないエラーは、自動的に Oracle JSP コンテナによってトラップされ、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記録されます。その後、イベント処理は、エラーが発生していない場合と同様に続行されます。

例 : application_OnStart 次の application_OnStart の例は、9-36 ページの「アプリケーション・イベント用の globals.jsa の例 : lotto.jsp」からのものです。この例では、特定のユーザー用に生成された抽選番号が 1 日間キャッシュされます。ユーザーが抽選を再度リクエストすると、そのユーザーは同じ一連の番号を取得します。キャッシュは 1 日に 1 回リサイクルされ、各ユーザーに新しい抽選番号が与えられます。目的どおりに機能させるには、抽選アプリケーションの停止時にキャッシュを保持し、再起動時に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、Oracle JSP コンテナが JSP アプリケーションをアンロードしたときに実行されます。アンロードは、以前にロードされたページが、オンデマンド動的再変換後に再ロード（JSP の unsafe_reload 構成パラメータが有効にされていない場合）されるたびに実行されるか、またはサーブレットである JSP コンテナの destroy() メソッドを、基礎となる Servlet コンテナからコールすることによって、その JSP コンテナが終了したときに実行されます。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、またはアプリケーションの状態を永続ストアに書き込みます。

このイベント・ハンドラは、JSP タグ（カスタム・タグなど）および空白のみを含む必要があり、静的テキストを含むことはできません。

このイベント・ハンドラで発生したにもかかわらず、イベント・ハンドラ・コードで処理されないエラーは、自動的に Oracle JSP コンテナによってトラップされ、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記録されます。その後、イベント処理は、エラーが発生していない場合と同様に続行されます。

例 : application_OnEnd 次の application_OnEnd の例は、9-36 ページの「アプリケーション・イベント用の globals.jsa の例 : lotto.jsp」からのものです。このイベント・ハンドラでは、アプリケーションの終了前に、キャッシュが lotto.che ファイルに書き込まれます。

```
<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、Oracle JSP コンテナが JSP ページ・リクエストにレスポンスして新しいセッションを作成したときに実行されます。これは、アプリケーション内のセッション対応の JSP ページに対する最初のリクエストが受信されるたびに、各クライアントで実行されます。

アプリケーションは、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バインドされたリソースの初期化
- アプリケーション内でクライアントが起動される場所の制御

session_OnStart は暗黙的な out オブジェクトを使用できるため、これは、JSP タグの他に静的テキストを含むことができる唯一の globals.jsa 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 からの出力が、ページからの出力より先に実行されます。

session_OnStart イベント・ハンドラ、およびそのイベントをトリガーした JSP ページは、同じ出力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・サイズによって制御されます。session_OnStart イベント・ハンドラは、ストリームを自動的にブラウザにフラッシュしません。ストリームは、JSP の一般的な規則に従ってフラッシュされます。この場合でも、ヘッダーは、session_OnStart イベントをトリガーした JSP ページに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にもかかわらず、イベント・ハンドラ・コードで処理されないエラーは、自動的に Oracle JSP コンテナによってトラップされ、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記録されます。その後、イベント処理は、エラーが発生していない場合と同様に続行されます。

例 : session_OnStart 次の例では、新しい各セッションがアプリケーションの最初のページ（index.jsp）で開始されるようにします。

```
<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、Oracle JSP コンテナが既存のセッションを無効にしたときに実行されます。これは、次の場合に実行されます。

- アプリケーションが、session.invalidate() メソッドをコールすることによって、セッションを無効にした場合
- セッションが、サーバーで期限切れになった（タイムアウトした）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・リソースを解放します。

このイベント・ハンドラは、JSP タグ（タグ・ライブラリのタグなど）および空白のみを含む必要があり、静的テキストを含むことはできません。

このイベント・ハンドラで発生したにもかかわらず、イベント・ハンドラ・コードで処理されないエラーは、自動的に Oracle JSP コンテナによってトラップされ、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記録されます。その後、イベント処理は、エラーが発生していない場合と同様に続行されます。

例 : session_OnEnd 次の例では、セッションの終了時にアクティブなセッションの数を減らします。

```
<event:session_OnEnd>

     <%-- Acquire beans --%>
     <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

     <%
         activeSessions.setValue(activeSessions.getValue() - 1);
     %>

 </event:session_OnEnd>
```

グローバルな宣言およびディレクティブ

globals.jsa ファイルは、イベント・ハンドラを保持する他に、JSP アプリケーション用のディレクティブおよび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。JSP ディレクティブ、JSP 宣言、JSP コメント、および scope パラメータを含む JSP タグ（jsp:useBean など）を含めることができます。

この項の内容は次のとおりです。

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 つの目的で使用します。

- globals.jsa ファイル自体を処理するために必要な情報を宣言する。
- 後続のページのデフォルト値を設定する。

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にあるすべての JSP ページ用の暗黙的ディレクティブになります。ただし、特定のページでは、globals.jsa ディレクティブをオーバーライドできます。

globals.jsa ディレクティブは、JSP ページで属性ごとにオーバーライドされます。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" %>
```

また、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同等です。

```
<%@ page import="java.util.*" bufferSize="20kb" %>
```

globals.jsa 内の宣言

globals.jsa ファイル内のすべてのイベント・ハンドラ間で共有されるように、メソッドまたはデータ・メンバーを宣言する場合、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 ページはこれらの宣言にアクセスできないため、このメカニズムを使用してアプリケーション・ライブラリを実装することができないことに注意してください。宣言のサポートは、共通機能をイベント・ハンドラ間で共有させるために、globals.jsa ファイルで提供されます。

グローバルな JavaBeans

globals.jsa ファイル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 ファイルで宣言されたオブジェクトは、globals.jsa イベント・ハンドラ、およびアプリケーション内に存在するすべての JSP ページの暗黙的オブジェクト環境の一部になります。

globals.jsa ファイルで（jsp:useBean 文などによって）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 ページで再宣言される必要はありません。

scope パラメータを含む任意の JSP タグまたは拡張機能（jsp:useBean や jml:useVariable など）を使用して、グローバル・オブジェクトを宣言できます。グローバルに宣言されたオブジェクトは、page または request スコープではなく、session または application スコープである必要があります。

ネストしたタグがサポートされています。したがって、jsp:setProperty コマンドを jsp:useBean 宣言内にネストできます。（jsp:setProperty が jsp:useBean 宣言外で使用されると、変換エラーが発生します。）

globals.jsa の構造

グローバル・オブジェクトを globals.jsa イベント・ハンドラで使用する場合、その宣言の位置が重要です。特定のイベント・ハンドラより前に宣言されたオブジェクトのみが、暗黙的オブジェクトとしてそのイベント・ハンドラに追加されます。そのため、globals.jsa ファイルを次の順序で構成することをお薦めします。

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 宣言

グローバルな宣言およびディレクティブの例

この項で示す globals.jsa ファイルの例は、次の操作を実行します。

- globals.jsa ファイル用およびすべての後続ページ用の JML タグ・ライブラリ（この場合は、コンパイル時に実装）を定義します。

  taglib ディレクティブを globals.jsa ファイルに含めることによって、このディレクティブをアプリケーションの個々の JSP ページに含める必要がなくなります。

- すべてのページが使用できるように、3 つのアプリケーション変数を（jsp:useBean 文で）宣言します。

globals.jsa を使用してグローバル宣言を行うその他の例については、9-42 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 --%>
```

Servlet 2.0 環境で globals.jsa を使用する例

この項では、Servlet 2.0 環境で Oracle の globals.jsa メカニズムを使用して、アプリケーション・フレームワーク、およびアプリケーションベースや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。次の例を示します。

- 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・イベントおよびセッション・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 の使用方法については、9-23 ページの「Oracle JSP アプリケーションおよび JServ でのセッション・サポート」を参照してください。

注意 : この項に示す例の一部の機能はアプリケーションの停止に基づいています。多くのサーバーでは、アプリケーションを手動で停止できません。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ことによって、アプリケーションを自動的に停止および再起動できます（developer_mode が true に設定されている場合）。（Oracle JSP コンテナは、アクティブなページを再変換および再ロードする前に、必ず実行中のアプリケーションを終了します。）

アプリケーション・イベント用の globals.jsa の例 : lotto.jsp

この例では、application_OnStart および application_OnEnd イベント・ハンドラを使用した globals.jsa のイベント処理を示します。この例では、番号がユーザーごとに 1 日間キャッシュされます。そのため、特定の抽選では、決まった番号の組がユーザーに与えられます。この例では、ユーザーは IP アドレスで識別されます。

アプリケーションの停止中もキャッシュを保持させるために、application_OnStart および application_OnEnd 用のコードが作成されています。この例が停止されるときにキャッシュ済データをファイルに書き込み、再起動されるときにそのファイルからデータを読み込みます（キャッシュが書き込まれた日と同じ日にサーバーが再起動される場合）。

lotto.jsp 用の 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 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・イベントおよびセッション・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、アプリケーションおよびセッション・ライフ・サイクル・イベントが処理されます。この例では、アクティブなセッションの数、セッションの合計数、およびアプリケーション・ページがヒットした合計回数がカウントされます。これらの各値は、application スコープに保持されます。アプリケーション・ページ（index1.jsp）は、リクエストのたびに、ページがヒットした回数を更新します。globals.jsa session_OnStart イベント・ハンドラは、アクティブなセッションの数およびセッションの合計数を増やします。globals.jsa session_OnEnd ハンドラは、アクティブなセッションの数を 1 つずつ減らします。

ページの出力は単純です。新しいセッションが開始されると、セッション・カウンタが出力されます。ページ・カウンタは、リクエストのたびに出力されます。それぞれの値の最終集計が、globals.jsa application_OnEnd イベント・ハンドラで出力されます。

この例では、次のことに注意してください。

- カウンタ変数が更新されると、これらの値が application スコープに保持されるため、アクセスが同期化される必要があります。
- カウント値は、oracle.jsp.jml.JmlNumber 拡張データ型を使用します。このデータ型は、application スコープにあるデータ値を簡単に使用できるようにします。JML の拡張データ型については、『Oracle9*i*AS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

index1.jsp 用の 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 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、変数がグローバルに宣言されます。この例は、9-39 ページの「アプリケーション・イベントおよびセッション・イベント用の globals.jsa の例 : index1.jsp」に示すイベント・ハンドラの例を基にしていますが、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。(元のイベント・ハンドラの例では、各イベント・ハンドラおよび JSP ページ自身が jsp:useBean 文を指定して、それらがアクセスする 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・ハンドラおよび JSP ページでその Bean が暗黙的に宣言されます。

index2.jsp 用の 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 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>
```

A

サード・パーティ・ライセンス

この付録では、Oracle9*i* Application Server に含まれ、このマニュアルで説明するサード・パーティ製品のサード・パーティ・ライセンスを示します。この付録の内容は次のとおりです。

Apache HTTP Server

Apache のライセンス条件に基づき、オラクル社は次の情報を通知する必要があります。ただし、Apache ソフトウェアを含む Oracle プログラムを使用する権利は、この製品に付属の Oracle プログラム・ライセンスによって決定され、次の通知に含まれる条件はそれらの権利を変更するものではありません。また、Oracle プログラム・ライセンスにこれと異なる規定があっても、Apache ソフトウェアはオラクル社によって現状のまま提供され、オラクル社または Apache からいかなる保証またはサポートも行われません。

Apache ソフトウェア・ライセンス

```
/* ====================================================================
 * The Apache Software License, Version 1.1
 *
 * Copyright (c) 2000 The Apache Software Foundation.  All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. The end-user documentation included with the redistribution,
 *    if any, must include the following acknowledgment:
 *       "This product includes software developed by the
 *        Apache Software Foundation (http://www.apache.org/)."
 *    Alternately, this acknowledgment may appear in the software itself,
 *    if and wherever such third-party acknowledgments normally appear.
 *
 * 4. The names "Apache" and "Apache Software Foundation" must
 *    not be used to endorse or promote products derived from this
 *    software without prior written permission. For written
 *    permission, please contact apache@apache.org.
 *
 * 5. Products derived from this software may not be called "Apache",
 *    nor may "Apache" appear in their name, without prior written
 *    permission of the Apache Software Foundation.
 *
```

```
* THIS SOFTWARE IS PROVIDED ''AS IS'' AND ANY EXPRESSED OR IMPLIED
* WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
* OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
* DISCLAIMED.  IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
* LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
* USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
* ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
* OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
* OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
* ====================================================================
*
* This software consists of voluntary contributions made by many
* individuals on behalf of the Apache Software Foundation.  For more
* information on the Apache Software Foundation, please see
* <http://www.apache.org/>.
*
* Portions of this software are based upon public domain software
* originally written at the National Center for Supercomputing Applications,
* University of Illinois, Urbana-Champaign.
*/
```

Apache JServ

Apache のライセンス条件に基づき、オラクル社は次の情報を通知する必要があります。ただし、Apache ソフトウェアを含む Oracle プログラムを使用する権利は、この製品に付属の Oracle プログラム・ライセンスによって決定され、次の通知に含まれる条件はそれらの権利を変更するものではありません。また、Oracle プログラム・ライセンスにこれと異なる規定があっても、Apache ソフトウェアはオラクル社によって現状のまま提供され、オラクル社または Apache からいかなる保証またはサポートも行われません。

Apache JServ パブリック・ライセンス

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistribution of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistribution in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- All advertising materials mentioning features or use of this software must display the following acknowledgment:

  This product includes software developed by the Java Apache Project for use in the Apache JServ servlet engine project (http://java.apache.org/).
- The names "Apache JServ", "Apache JServ Servlet Engine" and "Java Apache Project" must not be used to endorse or promote products derived from this software without prior written permission.
- Products derived from this software may not be called "Apache JServ" nor may "Apache" nor "Apache JServ" appear in their names without prior written permission of the Java Apache Project.
- Redistribution of any form whatsoever must retain the following acknowledgment:

  This product includes software developed by the Java Apache Project for use in the Apache JServ servlet engine project (http://java.apache.org/).

THIS SOFTWARE IS PROVIDED BY THE JAVA APACHE PROJECT "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE JAVA APACHE PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

索引

A

B

C

D

E

F

G

P

R

S

T

U

V

W

X

あ

い

え

お

か

き

く

こ

さ

し

す

せ

そ

た

ち

て

と

は

ひ

ふ

へ

ま

め

り