Oracle® Containers for J2EE

JavaServer Pages 開発者ガイド

10*g*（10.1.3.1.0）

部品番号 : B31860-01

2007 年 1 月

ORACLE®

Oracle Containers for J2EE JavaServer Pages 開発者ガイド , 10*g*（10.1.3.1.0）

部品番号 : B31860-01

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

原本部品番号 : B28961-01

原本著者 : Alfred Franci

原本協力者 : Bonnie Vaughan, Brian Wright, Dan Hynes, Dana Singleterry, Olaf Heimburger, Sumathi Gopalakrishnan, Jay Swaminathan, Ashok Banerjee, Ellen Barnes, Julie Basu, Matthieu Devin, Jose Alberto Fernandez, Ralph Gordon, Ping Guo, Hal Hildebrand, Susan Kraft, Sunil Kunisetty, Clement Lai, Qiang Lin, Song Lin, Jeremy Lizt, Angela Long, Sharon Malek, Sheryl Maring, Kuassi Mensah, Jasen Minton, Kannan Muthukkaruppan, John O'Duinn, Robert Pang, Olga Peschansky, Shiva Prasad, Jerry Schwarz, Sanjay Singh, Gael Stevens, Jayaram Swaminathan, Kenneth Tang, YaQing Wang, Alex Yiu, Shinji Yoshida, Helen Zhao

目次

2 Oracle JSP の実装

3 OC4J の JSP 環境の構成

4 ojspc による JSP ページのプリコンパイル

5 OC4J における JSP の変換

6 JSP での作業

7 カスタム・タグの使用

8 OC4J における JSP XML サポート

9 Oracle での JSP グローバリゼーション・サポート

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 社が先導する業界グループが提供する、JavaServer Pages（JSP）テクノロジの Oracle 実装について説明します。標準機能の概要を説明しますが、主として Oracle 実装の詳細と付加価値機能を中心に説明します。標準的な JSP テクノロジの概要の後に、OC4J 実装、JSP 構成、プログラミングに関する基本的な考慮事項、JSP の方針とヒント、変換とデプロイ、JSP タグ・ライブラリおよびグローバリゼーション・サポートについて説明します。

JavaServer Pages テクノロジは、標準の Java 2 Enterprise Edition（J2EE）のコンポーネントです。Oracle Application Server の J2EE コンポーネントは、Oracle Containers for J2EE（OC4J）です。

OC4J の Web コンテナは、JSP 仕様 2.0 とサーブレット仕様 2.4 の完全な実装です。

この章には、次の各項が含まれます。

対象読者

このマニュアルは、JavaServer Pages テクノロジに基づく Web アプリケーション作成に関心がある開発者を対象としています。Web およびサーブレット環境での作業経験があり、次の内容を理解していることが前提です。

- 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 サーバーおよびサーブレット環境の構成方法
- HTML
- Java
- Oracle JDBC（Oracle Database にアクセスする JSP アプリケーション用）

ドキュメントのアクセシビリティについて

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

ドキュメント内のサンプル・コードのアクセシビリティについて

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

外部 Web サイトのドキュメントのアクセシビリティについて

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

Oracle サポート・サービスへの TTY アクセス

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

関連ドキュメント

詳細は、Oracle Java Platform グループの次のマニュアルを参照してください。

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

  このマニュアルでは、OC4J をスタンドアロンの開発環境または本番環境で使用するためのガイドラインと手順について説明します。

- 『Oracle Containers for J2EE サーブレット開発者ガイド』

  このマニュアルは、基本的なサーブレットの開発、JDBC と EJB の使用、アプリケーションの構築とデプロイ、サーブレットと Web サイトの構成など、OC4J でのサーブレットとサーブレット・コンテナの使用に関する情報を、サーブレット開発者に提供します。開発用のスタンドアロン環境および本番用の Oracle Application Server における OC4J について考慮しています。

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

  このマニュアルは、タグ・ライブラリ、JavaBeans および OC4J が提供する他の Java ユーティリティに関する概念的な情報、詳細な構文および使用に関する情報を提供します。他の Oracle 製品グループのタグ・ライブラリのサマリーもあります。

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

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

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

  このマニュアル（『Oracle Application Server セキュリティ・ガイド』と混同しないでください）では、OC4J に固有のセキュリティ機能と実装について説明しています。JAAS（Java Authentication and Authorization Service）および他の Java セキュリティ・テクノロジの使用に関する情報も記載されています。

表記規則

このマニュアルでは次の表記規則を使用します。

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

サポートおよびサービス

次の各項に、各サービスに接続するための URL を記載します。

Oracle サポート・サービス

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

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

製品マニュアル

製品のマニュアルは、次の URL にあります。

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

研修およびトレーニング

研修に関する情報とスケジュールは、次の URL で入手できます。

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

その他の情報

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

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

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

1

JSP スタート・ガイド

この章では、JavaServer Pages（JSP）テクノロジの標準機能、および JSP 実行モデルについて説明します。一般的な情報の詳細は、Sun 社の JSP 仕様バージョン 2.0 を参照してください。

この章には、次の各項が含まれます。

- JavaServer Pages テクノロジの概要
- JSP 構文の要素の概要
- 式言語の使用による JSP 作成の単純化
- JSP の実行モデル

注意： 以前のリリースに含まれていたサンプル・アプリケーションに関する章はなくなりました。

サンプル・コードおよびアプリケーションは、OTN（Oracle Technology Network）の次の場所から入手できます。

`http://www.oracle.com/technology/sample_code/index.html`

JavaServer Pages テクノロジの概要

次の各項で、JSP の概要を説明します。

JavaServer Pages テクノロジの概要

単純にいえば、JavaServer Pages（JSP）テクノロジを使用すると、動的に生成されたコンテンツを Web ブラウザに表示できるようになります。JSP ページは、Oracle Application Server 環境で実行されているすべての Web ベース・アプリケーションのプレゼンテーション・レイヤーを構成し、アプリケーションのビジネス・ロジックおよび処理能力に対するインタフェースを提供します。

JSP ページは、次の 2 種類のマークアップ・テキストが含まれているテキスト・ファイルです。

- HTML または XML。ページのレイアウトおよびテンプレート・テキストなどの静的なコンテンツの書式設定に使用されます。
- JSP 構文の要素、および場合によっては埋込み Java コード。動的なコンテンツを提供します。

JSP は開発が容易なため、迅速に実装できます。最新のリリースでは、JSP ページの作成者には、Java に関する深い知識すら必要ありません。

JavaServer Pages は、JSP ページの変換と実行をサポートする Web コンテナを必要とします。この Web コンテナは、Oracle Containers for J2EE（OC4J）の一部として提供されています。OC4J の機能の詳細は、第 2 章「Oracle JSP の実装」を参照してください。

JSP は、Sun 社が指定する Java 2 プラットフォーム、Enterprise Edition（J2EE）アーキテクチャの主要なテクノロジです。OC4J の Web コンテナは、Sun 社の JSP 2.0 およびサーブレット 2.4 仕様に完全に準拠しています。

JSP の主な利点

ほとんどの場面で、JSP ページには、サーブレットと比べて少なくとも 3 つの利点があります。

- **コーディングの容易さ**

  JSP 構文を使用すると、動的な Web ページを簡単な方法でコーディングできるため、通常は、同様のサーブレット・コードより大幅に少ないコード量で済みます。また、標準の JSP またはサーブレット・インタフェースの実装、および HTTP セッションの作成など、一部のサーブレット・コーディングのオーバーヘッドが、JSP トランスレータによって自動的に処理されます。

- **静的なコンテンツと動的なコンテンツの分離**

  JSP テクノロジは、静的なコンテンツ用の HTML コードの開発と、ビジネス・ロジックおよび動的なコンテンツ用の Java コードの開発を、ある程度分離しようとするものです。このため、JSP プログラミングは Web 設計者にとって使用しやすく、魅力あるものになっています。プレゼンテーションやレイアウトの担当者と、Java 開発者との間で、メンテナンス作業を単純に分離できるためです。

- **ビジネス・ロジック・コンポーネントの再利用**

  JSP テクノロジは、JavaBeans や Enterprise JavaBeans（EJB）などの再利用可能なコンポーネントを使用しやすいよう設計されています。通常 J2EE アプリケーションとともに提供されている JSP タグ・ライブラリにより、コーディングはさらに容易になります。

JSP の仕組み

JSP ページの動的な性質は、Web ページの HTML（または XML などその他のマークアップ・コード）に埋め込まれている JSP 要素によって実現されます。これらの要素により、外部の Java コンポーネント（JavaBeans または Enterprise JavaBeans（EJB）など）へのアクセスが可能になります。これらの Java コンポーネントは Web アプリケーションのビジネス・ロジックと処理能力を提供します。同様に、これらのコンポーネントは、データベースまたは他の EIS に直接または間接的にアクセスできます。

JSP ページは、通常はクライアントからリクエストされた時点で、Java サーブレットに変換されます。JSP トランスレータは、URL 内の `.jsp` ファイル名拡張子によってトリガーされます。次に、変換されたページが実行され、普通のサーブレットと同様に、HTTP リクエストの処理やレスポンスの生成を実行します。JSP ページのコーディングは、同様のサーブレットをコーディングするよりも、かなり簡単です。

さらに、JSP ページは、サーブレットと完全に相互運用できます。つまり、JSP ページは、サーブレットの出力をインクルードしたり、サーブレットに出力を転送できます。一方、サーブレットは、JSP ページの出力をインクルードしたり、JSP ページに出力を転送できます。

次に、単純な JSP ページ `welcomeuser.jsp` のコードを示します。

```
<%@ page import="java.util.*" %>
<HTML>
<HEAD><TITLE>The Welcome User JSP</TITLE></HEAD>
<BODY>
<H3>Welcome ${param.user}!</H3>
<P><B> Today is ${Date}. Have a fabulous day! :-)</B></P>
<B>Enter name:</B>
<FORM METHOD=GET>
<INPUT TYPE="text" NAME="user" SIZE=5>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

ユーザーが「Amy」という名前を入力すると、この JSP ページでは、次のような出力が作成されます。

```
Welcome Amy!

Today is Wed Jun 2 3:42:23 PDT 2000. Have a fabulous day! :-)
```

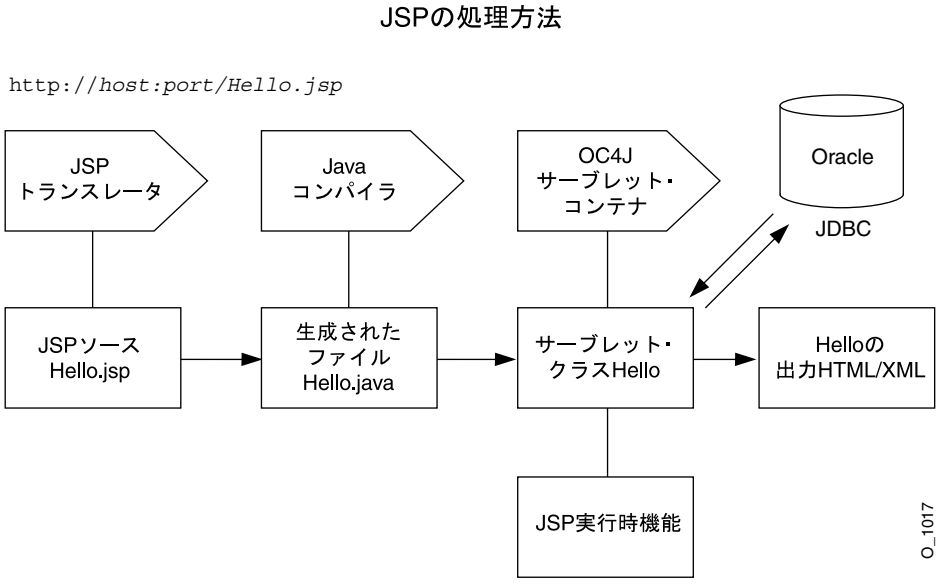
JSP の変換および実行時フロー

図 1-1 に、ユーザーがブラウザで URL を指定して JSP ページをコールした場合の、実行フローの概要を概念的に示します。Hello.jsp はデータベースにアクセスすると仮定します。

.jsp ファイル名拡張子により、次の手順が自動的に発生します。

1. JSP トランスレータが起動され、Hello.jsp を変換し、ファイル Hello.java を生成します。
2. Java コンパイラが起動され、Hello.class を作成します。
3. Hello.class がサーブレットとして実行され、JSP 実行時ライブラリを使用します。
4. Hello クラスが必要に応じて JDBC を介してデータベースにアクセスし、出力をブラウザに送信します。

図 1-1 JSP の変換および実行時フロー



JSP 構文の要素の概要

次の例では、一般的な JSP で、HTML マークアップと JSP 要素の両方を使用し、静的および動的なコンテンツを提供する方法を示します。動的なコンテンツは、OC4J の JSP コンテナが完全にサポートする、JSP 2.0 構文で作成されています。

注意： JSP 2.0 仕様では、従来の JSP 構文の代替として、XML 互換の JSP 構文をサポートしています。これにより、構文的に有効な XML 文書である JSP ページを作成できます。XML 互換の構文の詳細は、第 8 章「OC4J における JSP XML サポート」を参照してください。

JSP により、HTTP リクエストで指定された従業員について、データベースに保存されているすべての電話番号が表示されます。従業員の電話番号をキー / 値ペアのマップとして含む JavaBeans オブジェクトが、コードによって作成されます。JSP により電話番号が反復され、各キーとその値が HTML 表に表示されます。

```
<%@ page contentType="text/html; charset=UTF-8"; import="mypkg.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
    prefix="c" %>
<html>
 <head><title>Phone List</title></head>
 <body>

 <jsp:useBean id="employee" scope="application" class="mypkg.Employee"/>
  <jsp:setProperty name="employee" property="empUserId" param="employeeId"/>
 <c:set var="empName" value="${employee.fullName}" />
  <h2>Current Phone Numbers for ${empName}</h2>
   <c:if test ="${!empty employee.phoneNumbers}>
    <table>
      <tr>
       <th>Phone Type:</th><th>Number:</th>
      </tr>
     <c:forEach var="entry" items="{$employee.phoneNumbers}">
      <tr>
       <td>${entry.key}</td>
       <td>${entry.value}></td>
      </tr>
     </c:forEach>
    </table>
   </c:if>
   <c:if test="${empty employee.phoneNumbers}">
    <c:out value="No phone numbers were found for ${empName}">
   </c:if>
  </body>
</html>
```

次に、この例で使用されている JSP 要素を説明します。

- ディレクティブは、JSP の処理方法を OC4J コンテナに指示します。

  この例の `page` ディレクティブ（`<%@ page ... %>`）は、ページが返すコンテンツ・タイプを指定し、使用する `mypkg` パッケージをインポートします。`taglib` ディレクティブ（`<%@ taglib ... %>`）は、ページで使用するカスタム・タグ・ライブラリをインポートします。この例では、JavaServer Pages Standard Tag Library（JSTL）コア・ライブラリです。

- 標準アクション・タグは、いくつかの共通タスクを起動するための簡単なメカニズムを提供します。たとえば、実行を別の JSP に転送したり、オブジェクトを作成してそのプロパティにアクセスできます。

  この例では、`jsp:useBean` 標準アクションにより、`Employee` JavaBeans インスタンスが返されます。

`jsp:setProperty` タグにより、Bean プロパティの値が設定されます。この例では、`empUserId` プロパティに、`employeeId` リクエスト・パラメータの値が設定されます。

- カスタム・タグは標準アクション・タグと同様のものですが、ソフトウェア・ベンダーやページの作成者が作成およびパッケージ化したものである点が異なります。カスタム・タグを使用すると、再利用可能な Java クラスのロジックに JSP ページからアクセスでき、Java コードをページに直接埋め込む必要がなくなります。

  この例では、`c:forEach` タグによってユーザーの電話番号が反復され、`c:if` タグによってこのユーザーの電話番号が検出されたかどうかがテストされ、検出されなかった場合はメッセージが出力されます。

  すべてのカスタム・タグは、タグの接頭辞で示されるタグ・ライブラリに所属します。この例では、接頭辞 `c:` は、使用されているタグが JSTL の `core` タグ・ライブラリに所属することを示しています。カスタム・タグの作成および使用方法の詳細は、第 7 章「カスタム・タグの使用」を参照してください。

- `${ ... }` 構文によって識別される JSP の式言語（EL）の式を使用すると、JavaBeans やそのプロパティなどのオブジェクトに簡単にアクセスできます。

  EL は、従来の JSP 構文で使用される Java 式の代替手段です。たとえば、EL 式 `${employee.fullName}` は、JSP スクリプトの式 `<% = employee.getFullName() %>` と同等です。

  詳細は、1-18 ページの「式言語の使用による JSP 作成の単純化」を参照してください。

次の項では、ディレクティブ、スクリプト要素および標準アクション・タグを含む JSP の基本構文を説明し、いくつかの例を示します。また、Bean プロパティの変換についても説明します。JSP 2.0 構文の詳細は、Sun 社の JSP 仕様 2.0 を参照してください。

注意： この項では、標準の JSP 構文について説明します。JSP XML 構文および JSP XML 文書の詳細は、第 8 章「OC4J における JSP XML サポート」を参照してください。

ディレクティブ

ディレクティブは、JSP ページ全体に関する指示を Web コンテナに提供します。この情報は、ページの変換に使用されます。基本的な構文は、次のとおりです。

```
<%@ directive attribute="value" attribute2="value2"... %>
```

JSP 仕様では、次のディレクティブをサポートします。

- `page`
- `include`
- `taglib`

page ディレクティブ

ページに依存する属性を指定します。たとえば、スクリプト言語、コンテンツ・タイプ、文字エンコード、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・サイズ、バッファが満杯になったときにバッファを自動的にフラッシュするかどうかなどがあります。次に例を示します。

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

また、自動フラッシュを有効にし、JSP ページの出力バッファ・サイズを 20KB に設定する場合は、次のようにします。

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

次に、ページのバッファを無効にする例を示します。

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

include ディレクティブ

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します。次に例を示します。

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

リソースへのページ相対パスまたはコンテキスト相対パスのいずれかを指定します。ページ相対パスおよびコンテキスト相対パスの詳細は、1-23 ページの「JSP ページのリクエスト」を参照してください。

注意 :

- include ディレクティブは「静的なインクルード」と呼ばれ、この章で後述する jsp:include 操作に本質は類似していますが、jsp:include は変換時ではなくリクエスト時に効果があります。
- include ディレクティブは、同じサーブレット・コンテキスト（アプリケーション）内のファイル間でのみ使用できます。

---

taglib ディレクティブ

このディレクティブを使用して、JSP ページで使用するカスタム JSP タグのライブラリを指定します。ベンダーは、独自のタグ・セットを使用して、JSP 機能を拡張できます。このディレクティブには、タグ・ライブラリ・ディスクリプタへのポインタ、およびこのライブラリからタグの使用方法を識別するための接頭辞が含まれます。次に例を示します。

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

ライブラリ内のいずれかのタグをページの後のほうで使用するときは、常に oracust 接頭辞を使用します。このライブラリには、タグの dbaseAccess が含まれていると仮定します。

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

JSP タグ・ライブラリとタグ・ライブラリ・ディスクリプタの概要は、1-18 ページの「カスタム・タグ・ライブラリ」で説明します。詳細は、第 7 章「カスタム・タグの使用」を参照してください。

スクリプト要素

JSP スクリプト要素には、JSP ページに表示可能な Java コードの次のカテゴリが含まれます。

- 宣言
- 式
- スクリプトレット
- コメント

宣言

JSP ページで使用するメソッドまたはメンバー変数を宣言する文です。

JSP 宣言は、<%!...%> 宣言タグ内で標準の Java 構文を使用し、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、生成されたサーブレット・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 f=0.0; %>
```

この例では、メンバー変数の f を宣言します。JSP トランスレータで生成されたサーブレット・クラス・コードで、f はクラスのトップレベルで宣言されます。

注意： メンバー変数とは対照的に、メソッド変数は、後述するように、JSP スクリプトレット内で宣言されます。この 2 つの比較は、6-6 ページの「静的なインクルードと動的なインクルードの使用の比較」を参照してください。

式

評価され、必要に応じて文字列値に変換され、ページ内の検出された場所に表示される Java 式です。

JSP 式は、セミコロンで終了せず、`<%=...%>` タグ内に含まれます。次に例を示します。

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

注意： `jsp:setProperty` 文などに含まれるリクエスト時属性の JSP 式は、文字列値に変換する必要はありません。

スクリプトレット

ページのマークアップ言語内に記述される、Java コードの断片です。

スクリプトレットまたはコードの一部分は、1 行または複数行の Java コードで構成されている場合があります。このスクリプトレットを JSP ページの HTML コード内で使用すると、条件ブランチやループなどを設定できます。

JSP スクリプトレットは `<%...%>` スクリプトレット・タグ内に含まれ、通常の Java 構文を使用します。

次の例では、JavaBean インスタンスの `pageBean` を使用しています。

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

1 行で構成された JSP スクリプトレットが、2 行の HTML コードと交互に記述され、その 1 行にはセミコロンが不要な JSP 式が含まれています。JSP 構文によって、HTML コードは、`if` ブランチと `else` ブランチ（スクリプトレットに設定された Java の中カッコ内）内で条件付きで実行されます。

次の例では、Java コードがスクリプトレットに追加されています。

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

ここでは、JavaBean インスタンスの `pageBean` を使用し、すでにインスタンス化されているオブジェクトの `empmgr` には、既知または未知の従業員に対して適切な機能を実行するメソッドが設定されていることが前提になっています。

注意： メンバー変数とは対照的に、メソッド変数の宣言には、次の例のように JSP スクリプトレットを使用します。

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

このスクリプトレットは、メソッド変数の `f2` を宣言します。JSP トランスレータで生成されたサーブレット・クラス・コードでは、`f2` は、サーブレットのサービス・メソッド内で変数として宣言されます。

メンバー変数は、前述のように、JSP 宣言で宣言されます。

メソッド変数とメンバー変数の比較は、6-10 ページの「メソッド変数宣言とメンバー変数宣言の使用の比較」を参照してください。

コメント

Java コード内に埋め込まれたコメントと同様に、JSP コード内に埋め込まれた、開発者のコメントです。

コメントは、`<%--...--%>` 構文内に含まれます。次に例を示します。

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

HTML のコメントとは異なり、JSP のコメントは、ユーザーがブラウザでページのソースを表示しても参照できません。

JSP オブジェクトとスコープ

このマニュアルの JSP オブジェクトという用語は、JSP ページで宣言されるか、または JSP ページからアクセス可能な Java クラス・インスタンスを指します。JSP オブジェクトは、次のいずれかです。

- 明示的 : 明示的なオブジェクトは、JSP ページのコード内で宣言および作成され、選択した `scope` の設定に従って、その JSP ページおよびその他のページにアクセスできます。

または

- 暗黙的 : 暗黙的なオブジェクトは、基礎となる JSP 機能によって作成され、特定のオブジェクト型に固有の `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からアクセスできます。

この項には、次の項目が含まれます。

明示的なオブジェクト

通常、明示的なオブジェクトとは、`jsp:useBean` 操作文で宣言および作成された JavaBean インスタンスです。`jsp:useBean` 文などの操作文は、1-12 ページの「標準の JSP アクション・タグ」で説明しています。ここでは例を示します。

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

この文は、`mybeans` パッケージにある `NameBean` クラスの `pageBean` インスタンスを定義します。`scope` パラメータについては、1-11 ページの「オブジェクトのスコープ」を参照してください。

Java クラス・インスタンスを Java プログラム内で作成するように、オブジェクトは、Java スクリプトレットまたは宣言内でも作成できます。

暗黙的なオブジェクト

JSP テクノロジによって、JSP ページで一連の暗黙的なオブジェクトが使用可能になります。これらのオブジェクトは、Web コンテナによって自動的に作成される Java オブジェクトで、基礎となるサーブレット環境との相互作用を可能にします。

次に、使用可能な暗黙的なオブジェクトを示します。これらのオブジェクトで使用可能なメソッドについては、Sun 社の指定クラスとインタフェースに関する Javadoc を参照してください。

- `page`

  このオブジェクトは JSP ページ実装クラスのインスタンスで、ページの変換時に作成されます。ページ実装クラスは、インタフェース `javax.servlet.jsp.HttpJspPage` を実装します。`page` オブジェクトは、JSP ページ内では `this` と同じ意味であることに注意してください。

- `request`

  このオブジェクトは、HTTP リクエストを表し、`javax.servlet.ServletRequest` インタフェースを拡張する `javax.servlet.http.HttpServletRequest` インタフェースを実装するクラスのインスタンスです。

- `response`

  このオブジェクトは、HTTP レスポンスを表し、`javax.servlet.ServletResponse` インタフェースを拡張する `javax.servlet.http.HttpServletResponse` インタフェースを実装するクラスのインスタンスです。

  特定のリクエストに対する `response` オブジェクトと `request` オブジェクトは、相互に関連があります。

- `pageContext`

  このオブジェクトは、JSP ページのページ・コンテキストを表し、JSP ページ・インスタンスのすべての `page` スコープ・オブジェクトの格納およびアクセスで使用されます。`pageContext` オブジェクトは、JSP 2.0 の `javax.servlet.jsp.JspContext` を拡張する `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` スコープがあり、これによって、関連付けられた JSP ページ・インスタンスにのみアクセスできます。

- `session`

  このオブジェクトは、HTTP セッションを表す、`javax.servlet.http.HttpSession` インタフェースを実装するクラスのインスタンスです。

- `application`

  このオブジェクトは、Web アプリケーションのサーブレット・コンテキストを表し、`javax.servlet.ServletContext` インタフェースを実装するクラスのインスタンスです。

  `application` オブジェクトは、単一の JVM 内でアプリケーションのインスタンスの一部として実行している JSP ページ・インスタンスからアクセスできます。

- `out`

  コンテンツを JSP ページ・インスタンスの出力ストリームに書き込むためのオブジェクトです。このオブジェクトは、`java.io.Writer` クラスを拡張する `javax.servlet.jsp.JspWriter` クラスのインスタンスです。

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

- config

  このオブジェクトは、JSP ページのサーブレット構成を表し、javax.servlet.ServletConfig インタフェースを実装するクラスのインスタンスです。一般的に、サーブレット・コンテナは、ServletConfig インスタンスを使用して、初期化中の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 インスタンスです。

- 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（別の JSP ページから例外がスローされた場合に処理を転送するページ）にのみ適用されます。これらのエラー・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、別の JSP ページからスローされ、結果的に現行のエラー・ページを起動したキャッチされなかった例外を表す java.lang.Throwable インスタンスです。

  exception オブジェクトには、例外の発生時に処理が転送された JSP エラー・ページ・インスタンスからのみアクセスできます。JSP エラー処理の例と exception オブジェクトの使用方法は、6-24 ページの「実行時エラーの処理」を参照してください。

暗黙的なオブジェクトの使用

前項で説明した暗黙的なオブジェクトは、いずれも便利なオブジェクトです。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>
```

request オブジェクトは、他の暗黙的なオブジェクトと同様に、自動的に使用可能になり、明示的にインスタンス化されません。

オブジェクトのスコープ

JSP ページ内のオブジェクトは、明示的か暗黙的かに関係なく、特定のスコープ内でアクセス可能です。明示的なオブジェクト（jsp:useBean 操作で作成された JavaBean インタンスなど）の場合は、1-9 ページの「明示的なオブジェクト」の例に示すように、次の構文を使用して明示的にスコープを設定できます。

scope="*scopeValue*"

スコープには、次の 4 種類があります。

- scope="page"（デフォルトのスコープ）: オブジェクトには、そのオブジェクトが作成された JSP ページ内からのみアクセスできます。page スコープのオブジェクトは、暗黙的な pageContext オブジェクトに格納されます。page スコープは、ページの実行が停止すると終了します。

  JSP ページの実行中に、ユーザーがページをリフレッシュすると、すべての page スコープのオブジェクトについて新規インスタンスが作成されます。

- scope="request": オブジェクトには、そのオブジェクトを作成した JSP ページと同じ HTTP リクエスト・サービスを提供している JSP ページからアクセスできます。request スコープのオブジェクトは、暗黙的な request オブジェクトに格納されます。request スコープは、HTTP リクエストが完了すると終了します。

- scope="session": オブジェクトには、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 JSP ページからアクセスできます。session スコープのオブジェクトは、暗黙的な session オブジェクトに格納されます。session スコープは、HTTP セッションがタイムアウトになるか、または無効になると終了します。

- scope="application": オブジェクトには、単一の Java Virtual Machine 上で、そのオブジェクトを作成した JSP ページと同じ Web アプリケーションで使用される JSP ページからアクセスできます。これは、Java の静的変数の概念と同じです。application スコープのオブジェクトは、暗黙的な application サーブレット・コンテキスト・オブジェクトに格

納されます。application スコープは、アプリケーション自体が終了するか、Web コンテナまたはサーブレット・コンテナが停止すると終了します。

次のように、狭いスコープから広いスコープへ 4 つのスコープに区分けされます。

```
page  <request < session < application
```

1 つのアプリケーション内にある異なるページ間でオブジェクトを共有する場合、たとえば、あるページから別のページに実行を転送したり、あるページのコンテンツを別のページにインクルードする場合は、共有するオブジェクトに対して page スコープを使用できません。この場合は、ページごとに関連付けられた個別のオブジェクト・インスタンスを使用します。ページ間でのオブジェクト共有に使用できる最も狭いスコープは、request です。（ページをインクルードおよび転送する方法については、次の「標準の JSP アクション・タグ」を参照。）

注意： request、session および application スコープはサーブレットにも適用されます。

標準の JSP アクション・タグ

JSP 操作の要素により、JSP ページの実行中に発生する操作（Java オブジェクトをインスタンス化し、ページに対して使用可能にするなど）が行われます。次のような操作が含まれます。

- JavaBean インスタンスを作成し、そのプロパティにアクセスする操作
- 別の HTML ページ、JSP ページまたはサーブレットに実行を転送する操作
- 外部リソースを JSP ページにインクルードする操作

標準アクションについては、JSP 仕様に一連のタグが定義されています。JSP ページのコードは、この章で前述したディレクティブとスクリプト要素を使用して作成できますが、ここで説明する標準的なタグを使用すると、機能性や利便性が向上します。

JSP の標準アクションに対する一般的なタグの構文は、次のとおりです。

```
<jsp:tagattr="value" attr2="value2" ... attrN="valueN">
...body...
</jsp:tag>
```

ボディがない場合は、次のとおりです。

```
<jsp:tag attr="value", ..., attrN="valueN" />
```

次に、一般的に使用される JSP の標準アクション・タグについて簡単に説明します。

- jsp:usebean
- jsp:setProperty
- jsp:getProperty
- jsp:param
- jsp:include
- jsp:forward
- jsp:plugin

注意： 次のタグについては、このマニュアルの他の項で説明します。

- doBody タグについては、7-21 ページの「タグ・ボディの処理」を参照してください。
- 属性および起動タグについては、7-22 ページの「JSP フラグメントの使用」を参照してください。
- body、element および text タグについては、8-7 ページの「テンプレートおよび動的なテンプレート・コンテンツの含有」を参照してください。

jsp:useBean タグ

jsp:useBean タグは、Java 型のインスタンス（通常は JavaBean クラス）にアクセスし、Java 型のインスタンスを作成します。また、インスタンスを指定の名前または ID に関連付けます。インスタンスは、指定したスコープのスクリプト変数として、その ID を介して使用できます。スクリプト変数の概要は、1-18 ページの「カスタム・タグ・ライブラリ」を参照してください。スコープの詳細は、1-9 ページの「JSP オブジェクトとスコープ」を参照してください。

主な属性は、class、type、id および scope です。（使用頻度は多くありませんが、後述の beanName 属性もあります。）

id 属性を使用して、インスタンス名を指定します。Web コンテナは最初に、指定したスコープ内で、指定した型の指定の ID によってオブジェクトを検索します。オブジェクトが見つからない場合、コンテナはそのオブジェクトの作成を試みます。

class 属性は、Web コンテナが必要に応じてインスタンス化できるクラスを指定するために使用します。クラスは、抽象クラスにはできません。また、引数のないコンストラクタが必要です。

class 属性のかわりに、beanName 属性を使用することもできます。この場合は、クラス名のかわりにシリアライズ可能なリソースを指定するオプションがあります。beanName 属性を使用すると、Web コンテナは java.beans.Beans クラスの instantiate() メソッドを使用してインスタンスを作成します。

type 属性は、Web コンテナがインスタンス化できない型（インタフェース、抽象クラス、あるいは引数のないコンストラクタを持たないクラスのうちいずれか）を指定するために使用します。インスタンスがすでに存在する場合やインスタンス化可能なクラスのインスタンスが型に割り当てられる場合は、type を使用します。次に、典型的な 3 種類の使用例を示します。

- ターゲット・スコープにすでに存在しているインスタンスを指定するには、type と id を使用します。
- クラスのインスタンス名を指定するには、class と id を使用します。インスタンスは、ターゲット・スコープにすでに存在しているインスタンスか、Web コンテナによって新規作成されるインスタンスのいずれかです。
- インスタンス化するクラス、およびインスタンスを割り当てる型を指定するには、class、type および id を使用します。この場合、クラスは、型に対して正式に割当て可能であることが必要です。

scope 属性を使用して、インスタンスのスコープを指定します。ページ・コンテキスト・オブジェクトに関連付けるインスタンスの場合は page、HTTP リクエスト・オブジェクトに関連付けるインスタンスの場合は request、HTTP セッション・オブジェクトに関連付けるインスタンスの場合は session、サーブレット・コンテキストに関連付けるインスタンスの場合は application です。

次の例では、型 MyIntfc の request スコープ・インスタンス reqobj を使用しています。MyIntfc はインタフェースであり、直接インスタンス化できないため、reqobj がすでに存在している必要があります。

```
<jsp:useBean id="reqobj" type="mypkg.MyIntfc" scope="request" />
```

次の例では、クラス PageBean の page スコープ・インスタンス pageobj を使用しています（必要に応じて、このインスタンスを最初に作成します）。

```
<jsp:useBean id="pageobj" class="mybeans.PageBean" scope="page" />
```

次の例では、クラス SessionBean のインスタンスを作成し、そのインスタンスを型 MyIntfc の変数 sessobj に割り当てています。

```
<jsp:useBean id="sessobj" class="mybeans.SessionBean"
             type="mypkg.MyIntfc scope="session" />
```

jsp:setProperty タグ

jsp:setProperty タグは、1 つ以上の Bean プロパティを設定します。Bean は、jsp:useBean タグにすでに指定されている必要があります。指定のプロパティに値を直接指定したり、指定のプロパティの値を関連の HTTP リクエストのパラメータから取得したり、HTTP リクエストのパラメータから一連のプロパティと値を繰り返し取得できます。

次の例では、HTTP リクエストの username というパラメータの値セットに従って、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・パラメータが同じ名前（user）の場合は、次のように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で、Bean のプロパティ名とリクエストのパラメータ名を繰り返し一致させ、対応するリクエストのパラメータ値に従って Bean のプロパティ値を設定します。

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

jsp:setProperty タグを使用すると、文字列以外のプロパティの値を指定する場合でも、文字列入力を使用できます。これは、バックグラウンドで変換が行われるためです。追加情報は、1-17 ページの「文字列値から Bean プロパティへの変換」を参照してください。

重要： property="*" の場合は、次の点に注意してください。

- エラーが発生した場合に反復を継続するように指定するには、setproperty_onerr_continue 構成パラメータを true に設定します。このパラメータの詳細は、3-2 ページの「OC4J JSP コンテナの構成」を参照してください。（以前のリリースでは、デフォルトの動作は継続でした。ただし、OC4J 9.0.4 実装から、デフォルトの動作はエラー発生時に停止となっています。）

  setproperty_onerr_continue パラメータは Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。

- JSP 仕様では、プロパティの設定順序を規定していません。順序を指定し、JSP ページを移植可能にする場合は、各プロパティに対して異なる jsp:setProperty 文を使用する必要があります。また、jsp:setProperty 文を個別に使用する場合、JSP トランスレータは、対応する set*XXX*() メソッドを直接生成できます。この場合は、変換中にのみイントロスペクションが発生します。実行時は、Bean のイントロスペクションが発生しないので、コストがかかりません。

---

jsp:getProperty タグ

jsp:getProperty タグは、Bean のプロパティ値を読み取って Java 文字列に変換し、暗黙的な out オブジェクトに配置します。これによって、文字列値を出力として表示できます。Bean は、jsp:useBean タグにすでに指定されている必要があります。文字列に変換する場合、プリミティブ型は直接変換され、オブジェクト型は、java.lang.Object クラスに指定されている 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 オブジェクトに出力します。

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

jsp:param タグ

jsp:params タグは、jsp:include タグ、jsp:forward タグおよび jsp:plugin タグ（後述）とともに使用できます。

jsp:param タグを jsp:forward タグおよび jsp:include タグとともに使用すると、HTTP request オブジェクトのパラメータ値に対して名前 / 値ペアが必要に応じて提供されます。この操作で指定した新規のパラメータと値は、request オブジェクトに追加され、古い値よりも優先されます。

次の例では、request オブジェクトの username パラメータに、Smith という値を設定します。

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

jsp:include タグ

jsp:include タグは、リクエスト時にページが表示されると、静的または動的な追加リソースをページに挿入します。リソースは、相対 URL（ページ相対またはアプリケーション相対）を使用して指定します。次に例を示します。

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

flush 属性を「true」に設定すると、jsp:include 操作の実行時に、バッファがブラウザにフラッシュされます。JSP 仕様および OC4J Web コンテナは、「true」または「false」のいずれかの設定をサポートします。デフォルトは「false」です。

次の例に示すように、jsp:param タグを持つ操作ボディも指定できます。

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

次の構文は、前述の例の代替として機能します。

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

注意：

- 動的なインクルードと呼ばれる jsp:include タグは、この章で前述した include ディレクティブと特性は類似していますが、変換時ではなくリクエスト時に効果があります。この 2 つの比較は、6-6 ページの「静的なインクルードと動的なインクルードの使用の比較」を参照してください。
- jsp:include タグは、同じサーブレット・コンテキスト（アプリケーション）内のページ間でのみ使用できます。

jsp:forward タグ

jsp:forward タグは、現行のページの実行を事実上終了し、その出力を破棄し、新規ページ（HTML ページ、JSP ページまたはサーブレットのいずれか）をディスパッチします。

jsp:forward タグを使用するには、JSP ページをバッファする必要があります。page ディレクティブに buffer="none" は設定できません。この操作では、バッファがクリアされ、コンテンツはブラウザに出力されません。

jsp:include の場合と同様、次の 2 番目の例に示すように、jsp:param タグを持つ操作ボディも指定できます。

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

または

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

注意 :

- この jsp:forward の例と前述の jsp:include の例の違いは、jsp:include の例では、現行のページの出力内に userinfopage.jsp を挿入したのに対して、jsp:forward の例では、現行のページの実行を停止して userinfopage.jsp を表示していることです。
- jsp:forward タグは、同じサーブレット・コンテキスト内のページ間でのみ使用できます。
- jsp:forward タグによって、元の request オブジェクトは、ターゲット・ページに転送されます。request オブジェクトを転送しない場合は、標準の javax.servlet.http.HttpServletResponse インタフェースに指定されている sendRedirect(String) メソッドを使用できます。これによって、一時的なリダイレクト・レスポンスが、指定したリダイレクト場所の URL を使用して、クライアントに送信されます。相対 URL は開発者が指定できます。この相対 URL は、サーブレット・コンテナによって絶対 URL に変換されます。

jsp:plugin タグ

jsp:plugin タグによって、指定のアプレットまたは JavaBean がクライアント・ブラウザで実行されます。必要に応じて、Java プラグイン・ソフトウェアのダウンロードが先行して実行されます。

jsp:plugin 属性を使用して、実行するアプレットやコード・ベースなどの構成情報を指定します。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）を指定できます。

jsp:params の開始タグと終了タグの間にネストされている jsp:param タグを使用して、アプレットまたは JavaBean にパラメータを指定します。（jsp:params の開始タグと終了タグは、jsp:include 操作または jsp:forward 操作で jsp:param を使用する場合は含まれません。）

プラグインが実行できない場合は、jsp:fallback タグを使用して、実行する代替テキストを定めます。

次の例は、アプレット・プラグインの使用方法を示します。

```
<jsp:plugin type=applet code="Sample.class" codebase="/html" >
   <jsp:params>
      <jsp:param name="sample" value="samples/sample01" />
   </jsp:params>
   <jsp:fallback>
```

```
      <p>Unable to start the plugin.</p>
   </jsp:fallback>
</jsp:plugin>
```

ARCHIVE、HEIGHT、NAME、TITLE、WIDTH など、その他のパラメータも jsp:plugin タグ内で使用できます。これらのパラメータの使用方法は、一般的な HTML 仕様に従います。

文字列値から Bean プロパティへの変換

JSP ページで jsp:useBean タグを介して JavaBean を使用し、次に jsp:setProperty タグを使用して Bean プロパティを設定する場合は、バックグラウンドで変換が行われるため、文字列以外のプロパティの値を指定する場合でも、文字列入力を使用できます。変換には、2 つの手順があります。次の各項で説明します。

- 一般的なプロパティ変換
- プロパティ・エディタによるプロパティの型変換

一般的なプロパティ変換

関連するプロパティ・エディタがない Bean プロパティについて、文字列値を使用してプロパティを設定する場合の変換方法を表 1-1 に示します。

表 1-1 属性の変換方法

プロパティの型	変換
Boolean または boolean	Boolean クラスの valueOf(String) メソッドに基づきます。
Byte または byte	Byte クラスの valueOf(String) メソッドに基づきます。
Character または char	String クラスの charAt(0) メソッド（索引値として 0 を入力）に基づきます。
Double または double	Double クラスの valueOf(String) メソッドに基づきます。
Integer または int	Integer クラスの valueOf(String) メソッドに基づきます。
Float または float	Float クラスの valueOf(String) メソッドに基づきます。
Long または long	Long クラスの valueOf(String) メソッドに基づきます。
Short または short	Short クラスの valueOf(String) メソッドに基づきます。
Object	String のコンストラクタがコールされる場合と同じように、リテラルの文字列を入力します。 Object インスタンスとして String インスタンスが戻されます。

プロパティ・エディタによるプロパティの型変換

Bean プロパティには、関連するプロパティ・エディタを指定できます。このエディタは、`java.beans.PropertyEditor` インタフェースを実装するクラスです。このようなクラスは、プロパティの編集に使用する GUI をサポートします。一般的に、標準の Java 型には標準のプロパティ・エディタがあり、ユーザー定義型にはユーザー定義のプロパティ・エディタがあります。ただし、OC4J の JSP 実装で検索されるのは、ユーザー定義のプロパティ・エディタのみです。`sun.beans.editors` パッケージのデフォルトのプロパティ・エディタは考慮されません。

プロパティ・エディタの詳細、およびプロパティ・エディタを型に関連付ける方法については、Sun 社の JavaBeans API 仕様を参照してください。

プロパティ・エディタが関連付けられているプロパティを、JavaBeans 仕様で指定されているように設定するには、以前と同様に文字列値を使用できます。この場合は、入力された文字列を適切な型の値に変換する際に、`PropertyEditor` インタフェースで指定されている `setAsText(String text)` メソッドが使用されます。(`setAsText()` メソッドが `IllegalArgumentException` をスローした場合、変換は失敗します。)

カスタム・タグ・ライブラリ

JSP 仕様では、前述の標準的な JSP タグ以外に、ベンダーが独自のタグ・ライブラリを定義できます。また、ベンダーは、顧客が独自のタグ・ライブラリを定義できるフレームワークを実装することもできます。

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、JSP のサブ言語とみなすことができます。開発者は、JSP ページを手動でコーディングするときにタグ・ライブラリを直接使用できますが、Java 開発ツールで自動的に使用することもできます。標準のタグ・ライブラリは、異なる Web コンテナ実装間で移植可能であることが必要です。

JSP タグ・ライブラリに対する標準的な JavaServer Pages のサポートに関する主な概念には、次の項目が含まれます。

これらの項目については、第 7 章「カスタム・タグの使用」を参照してください。

OC4J が提供するタグ・ライブラリの詳細は、『Oracle Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

式言語の使用による JSP 作成の単純化

JSP 式言語（EL）を使用すると、JavaBeans に格納されているリクエスト・パラメータやアプリケーション・データにアクセスする際、埋込み Java スクリプトレットや式を使用する必要がなく、JSP の作成が非常に簡単になります。

EL は、元は JavaServer Pages 標準タグ・ライブラリ（JSTL）バージョン 1.0 の一部として導入されました。JSP 2.0 のリリースに伴い、EL は JSP 仕様に統合され、データへのアクセス機能が著しく改善されました。

JSP 2.0 準拠の OC4J コンテナでは、実装された EL 式を次のものとして解釈できます。

- 標準アクション（`jsp:useBean` など）または実行時の式を使用できるカスタム・タグの属性値。
- HTML、または JSP 要素以外のテキストなど、静的なテンプレート・テキスト。この用法では、テキスト内の式の値が評価され、現行の出力に挿入されます。ただし、タグ・ボディが `tagdependent` と宣言されている場合、式は評価されません。

次の例では、JSTL の `c:if` タグを使用して、企業のリストから製鉄業の企業を選択します。

```
<c:if test="${company.industry == 'steel'}">
   ...
</c:if>
```

式言語の構文の概要

この項では、式言語の構文の概要を示し、OC4J の JSP アプリケーションで EL 評価を使用する方法を説明します。

EL には独自の構文がありますが、一般的な目的で使用されるプログラミング言語ではありません。むしろ、JSP 作成者の作業を簡単にするためのデータ・アクセス・メカニズムです。

JSP 式言語の構文

式言語には、部分的に JavaScript の構文に基づいた、独自の構文があります。次に、JSP 式言語の主な構文機能の概要を示します。続いて、単純な例をいくつか示します。

- 起動

  式言語は、`${expression}` 構文によって起動されます。最も基本的なセマンティックとしては、指定された変数 `${foo}` を起動すると、メソッド・コール `PageContext.findAttribute(foo)` と同じ結果が得られます。

- データ構造へのアクセス

  JavaBeans や、リスト、マップおよび配列などのコレクション内の指定されたプロパティにアクセスするために、式言語では「`.`」および「`[]`」演算子をサポートしています。

  「`.`」演算子を使用すると、名前が標準の Java 識別子であるプロパティにアクセスできます。たとえば、`employee.phones.cell` は、Java 構文の `employee.getPhones().getCell()` と同等です。

  「`[]`」演算子は、配列またはリストへのアクセスなど、より一般的なアクセスに使用します。ただし、有効な Java 識別子に対しては、「`.`」演算子と同等です。たとえば、式 `employee.phoneNumbers` と `employee["phoneNumbers"]` は同じ結果を返します。

- 関係演算子

  式言語では、関係演算子 `==`（または `eq`）、`!=`（または `ne`）、`<`（または `lt`）、`>`（または `gt`）、`<=`（または `le`）、`>=`（または `ge`）をサポートしています。

- 算術演算子

  式言語では、算術演算子 `+`、`-`、`*`、`/`（または `div`）、`%`（または `mod`、剰余またはモジュロに使用）をサポートしています。

- 論理演算子

  式言語では、論理演算子 `&&`（または `and`）、`||`（または `or`）、`!`（または `not`）、`empty` をサポートしています。

基本的な例

次の例では、式言語の基本的な起動を示します。関係演算子 `<=`（以下を示す）が含まれています。

```
<c:if test="${auto.price <= customer.priceLimit}">
   The <c:out value="${auto.makemodel}"/> is in your price range.
</c:if>
```

コレクションへのアクセスの例

次の例では、「`.`」および「`[]`」演算子の使用方法を示します。`catalogue` は製品の説明が含まれている `Map` オブジェクト、`preferences` は特定のユーザーの設定が含まれている `Map` オブジェクトです。

```
Item:
<c:out value="${catalogue[productId]}"/>
   Delivery preference:
<c:out value="${user.preferences['delivery']}"/>
```

式言語の暗黙的なオブジェクト

式言語では、次の暗黙的なオブジェクトが用意されています。

- pageScope: page スコープ変数にアクセスできます。
- requestScope: request スコープ変数にアクセスできます。
- sessionScope: session スコープ変数にアクセスできます。
- applicationScope: application スコープ変数にアクセスできます。
- pageContext: JSP ページのページ・コンテキストのすべてのプロパティにアクセスできます。
- param: 通常 request.getParameter() メソッドによってアクセスされるリクエスト・パラメータを含んでいる、Java の Map オブジェクト。式 ${param["foo"]} または同等の ${param.foo} は、いずれもリクエスト・パラメータ foo に関連付けられた最初の文字列値を返します。
- paramValues: たとえば、paramValues["foo"] を使用すると、リクエスト・パラメータ foo に関連付けられたすべての文字列値の配列が返されます。
- header: param と同様、このオブジェクトを使用して、リクエスト・ヘッダーに関連付けられた最初の文字列値にアクセスできます。
- headerValues: paramValues を使用する際と同様に、このオブジェクトを使用して、リクエスト・ヘッダーに関連付けられたすべての文字列値にアクセスできます。
- initParam: コンテキスト初期化パラメータにアクセスできます。
- cookie: リクエスト内で受信した Cookie にアクセスできます。

式言語の追加機能

式言語では、次の追加機能も提供されています。

- 式の評価の失敗がリカバリ可能であるとみなされる場合、デフォルト値を提供できます。
- アプリケーション・データが、タグ属性または式言語演算子によって予期される型に完全に一致しない場合のために、結果の値の型を予期される型に変換するための規則が用意されています。

式言語の関数の作成および使用

式言語では、EL 式内で起動可能な、関数と呼ばれる静的なメソッドを定義できます。

関数の作成または使用は、カスタム・タグの作成または使用と似ています。現に、JSTL には、実際は式言語の関数であるカスタム・タグが 6 つ含まれています。カスタム・タグの実装の詳細は、第 7 章「カスタム・タグの使用」を参照してください。

関数は、PUBLIC な Java クラス内の PUBLIC な静的なメソッドとして実装する必要があります。次の例では、Jakarta Taglib 標準ライブラリから入手可能な、JSTL の fn:length 関数の静的なメソッドを説明しています。

```
public static int length(Object obj)
throws JspTagException {
...
}
```

関数メソッドが含まれているクラスは、カスタム・タグと同様、タグ・ライブラリとしてグループ化されます。各関数のシグネチャと、対応するメソッドが含まれている PUBLIC クラスへのマッピングが、タグ・ライブラリ・ディスクリプタ（TLD）ファイルに追加されます。次に例を示します。

```
<function>
 <description>
  Returns the number of items in a collection or the number of
  characters in a string.
```

```
 </description>
 <name>length</name>
 <function-class>
  org.apache.taglibs.standard.functions.Functions
 </function-class>
 <function-signature>
  int length(java.lang.Object)
 </function-signature>
</function>
```

EL の関数を使用するには、JSP では taglib ディレクティブを使用して適切なタグ・ライブラリをインポートする必要があります。次の例では、fn:length 関数を実装する Java クラスが含まれている、JSTL の functions ライブラリをページにインポートします。

```
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
```

これで、関数を JSP 内の EL 式から起動できます。次の例では、スコープ付き変数 employees は、Employee オブジェクトのコレクションです。

```
There are ${fn:length(employees)} employees listed in the database.
```

式言語の無効化

EL 構文で書かれたパターンを、EL 式として評価せずに JSP で使用できるよう、式言語を無効化または解除することが可能です。EL は、Web アプリケーションまたは個別の JSP レベルのいずれでも無効化できます。また、EL 式を無視するよう、タグ・ファイルに指示することも可能です。

パターン \$ は、EL が有効な場合は引用符として認識されますが、EL を無効化すると引用符として認識されないため、注意してください。

Web アプリケーション内のすべての JSP での EL の無効化

アプリケーション内のすべての JSP で EL を無効化するには、アプリケーションの web.xml Web アプリケーション・ディスクリプタ・ファイルに、次の <jsp-property-group> 要素を追加します。

```
<jsp-property-group>
  <url-patter>*.jsp</url-pattern>
  <el-ignored>true</el-ignored>
</jsp-property-group>
```

JSP 内での EL の無効化

JSP ページで EL 式を無効化するには、JSP で、page ディレクティブの isELIgnored 属性を true に設定します。

タグ・ファイルでの EL の無効化

タグ・ファイルで EL 評価を無効化するには、タグ・ファイルで、tag ディレクティブの isELIgnored 属性を true に設定します。

JSP の実行モデル

この項では、オンデマンド変換（JSP ページの初回実行時）およびエラー処理など、JSP ページの実行方法の概要を説明します。

JSP の実行モデル

JSP ページの実行モデルには、次の 2 種類があります。

- ほとんどの実装およびほとんどの状態で、Web コンテナは、ページを実行する直前に、オンデマンドで、つまり、ユーザーがリクエストしたときにページを変換します。
- ただし、開発者がページを事前に変換し、そのページを作業サーブレットとしてデプロイする場合もあります。コマンドライン・ツールを使用してページを変換、ロードおよび公開し、ページを実行可能にできます。変換は、クライアントまたはサーバーのいずれかで実行できます。ユーザーが JSP ページをリクエストすると、ページは直接実行されるため、変換は必要ありません。

オンデマンド変換モデル

JSP ページはオンデマンド変換で実行するのが一般的です。Web コンテナが取り込まれた Web サーバーから JSP ページがリクエストされると、フロントエンド・サーブレットがインスタンス化されて起動します（Web サーバーが正しく構成されていることが前提です）。このサーブレットは、Web コンテナのフロントエンドとみなすことができます。OC4J では、`oracle.jsp.runtimev2.JspServlet` です。

`JspServlet` は、必要な場合（変換したクラスが存在しない場合、または JSP ページ・ソースが更新されている場合）、JSP ページの検索、変換およびコンパイルを行い、そのページの実行をトリガーします。

Web サーバーは、ファイル名の拡張子 `*.jsp`（URL 内）を `JspServlet` にマッピングするために、正しく構成されている必要があります。

事前変換モデル

通常のオンデマンド変換とは別の方法として、開発者が JSP ページを事前に変換してからデプロイする場合があります。この事前変換には、次のようなメリットがあります。

- 実行時の変換が不要なため、ユーザーは、初めて JSP ページをリクエストする際に時間を節約できます。
- 専用ソフトウェアであるか、またはセキュリティ上の理由でコードを公開しないようにするために、バイナリ・ファイルのみをデプロイする場合に役立ちます。

Oracle には、JSP ページを事前に変換するための `ojspc` コマンドライン・ユーティリティが用意されています。このユーティリティには、出力ファイル用の適切なベース・ディレクトリを、アプリケーションのデプロイ方法に応じて設定できるオプションがあります。`ojspc` ユーティリティについては、第 4 章「ojspc による JSP ページのプリコンパイル」で説明します。

JSP ページとオンデマンド変換

典型的なオンデマンド変換では、JSP ページは、通常、次の手順で実行されます。

1. ユーザーは、ファイル名が .jsp で終わる URL により、JSP ページをリクエストします。
2. Web サーバーのサーブレット・コンテナは、URL 内のファイル名の拡張子 .jsp を認識すると、すぐに Web コンテナを起動します。
3. Web コンテナは、JSP ページが初めてリクエストされると、その JSP ページを検索して変換します。変換処理では、.java ファイルにサーブレット・コードが作成され、その .java ファイルがコンパイルされて、サーブレットの .class ファイルが作成されます。

   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クラス（Web コンテナによって提供されます）を拡張します。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・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、標準的なサーブレット・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、サービス・メソッドのコード生成など）が、生成されたサーブレット・コードに自動的に取り込まれます。
4. Web コンテナは、ページ実装クラスのインスタンス化と実行をトリガーします。

次に、JSP ページ・インスタンスは、HTTP リクエストを処理して HTTP レスポンスを生成し、そのレスポンスをクライアントに返信します。

注意： 前述の手順では、概略を説明しています。前述のように、各ベンダーは Web コンテナの実装方法を決定しますが、Web コンテナはサーブレットまたは複数サーブレットの集合で構成されます。たとえば、サーブレットには、JSP ページを検索するフロントエンド・サーブレット、変換とコンパイルを処理する変換サーブレット、各ページ実装クラスによって拡張されるラッパー・サーブレット・クラス（変換されたページは実際には純粋なサーブレットではなく、サーブレット・コンテナで直接実行できないため）などがあります。サーブレット・コンテナは、これらのコンポーネントを実行するために必要です。

JSP ページのリクエスト

JSP ページは、URL にアクセスして直接的にリクエストしたり、別の Web ページやサーブレットを使用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

サーブレットや HTML ページと同様に、ユーザーは、URL により JSP ページを直接リクエストできます。たとえば、次のように、myapp ディレクトリに HelloWorld JSP ページがあり、myapp は、Web サーバーの myapproot コンテキスト・パスにマッピングされていると仮定します。

```
myapp/dir/HelloWorld.jsp
```

たとえば、次の URL にアクセスしてこのページをリクエストできます。

```
http://host:port/myapproot/dir/HelloWorld.jsp
```

ユーザーが初めて HelloWorld.jsp をリクエストすると、Web コンテナは、このページの変換と実行をトリガーします。後続のリクエストでは、Web コンテナはページの実行のみトリガーし、変換手順は不要となります。

注意： 一般的なサーブレットと JSP の起動の詳細は、『Oracle Containers for J2EE サーブレット開発者ガイド』を参照してください。

JSP ページの間接的なリクエスト

サーブレットと同様に、JSP ページも、通常の HTML ページからリンクしたり、別の JSP ページやサーブレットから参照して、間接的に実行できます。

ある JSP ページを別の JSP ページにある JSP 文から起動する場合のパスは、アプリケーション・ルートに対して相対的なパス（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呼ばれます）、または起動ページに対して相対的なパス（ページ相対パスと呼ばれます）のいずれかになります。ページ相対パスと異なり、アプリケーション相対パスは、「/」で始まります。

通常、これらのパスは、URL リンクまたは HTML リンクで使用するパスと同じではありません。前項の例の場合、次のように、HTML リンクで使用するパスは、直接的な URL リクエストで使用するパスと同じです。

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

次に、JSP 文のアプリケーション相対パスを示します。

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

次に、同じディレクトリ内の JSP ページから `HelloWorld.jsp` を起動するページ相対パスを示します。

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

（`jsp:include` 文および `jsp:forward` 文については、1-12 ページの「標準の JSP アクション・タグ」を参照してください。）

2

Oracle JSP の実装

この章では、Oracle Application Server のコンポーネントである Oracle Containers for J2EE（OC4J）付属の Web コンテナが実装する機能の詳細を説明します。Oracle Application Server、OC4J、OC4J JSP の実装と機能、および提供されているカスタム・タグ・ライブラリとユーティリティ（『Oracle Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照）の概要を説明します。

次の項目について説明します。

OC4J の概要

Oracle Containers for J2EE 10*g* リリース 3（10.1.3）、略称 OC4J は、完全な Java 2 Enterprise Edition（J2EE）1.4 準拠の環境を提供します。スタンドアロン OC4J は、開発環境および小規模から中規模の本番環境で使用されます。

OC4J は、すべて Java で作成されており、標準の Java Development Kit（JDK）の Java 仮想マシン（JVM）で実行されます。ご使用のオペレーティング・システムの標準の JDK で OC4J を実行できます。

OC4J のドキュメントでは、Java プログラミング、J2EE テクノロジ、および Web と EJB アプリケーション・テクノロジについて、基本的な知識があることを前提としています。これには、`/WEB-INF` や `/META-INF` ディレクトリなどのデプロイ規則が含まれます。

OC4J の新機能

Oracle Containers for J2EE リリース 3（10.1.3）には、いくつかの新機能や拡張機能が含まれています。これらについて、次に説明します。

Web サービスのサポート

OC4J では、J2EE 1.4 標準（JAX-RPC 1.1 を含む）に基づいた Web サービスを完全にサポートしています。また、Web サービスの相互運用性もサポートしています。

- EJB 2.1 Web サービス・エンド・ポイント・モデル
- JSR 109 クライアント・サーバー・デプロイメント・モデル
- CORBA Web サービス : 既存の基本 CORBA サーバントの Web サービスとしてのラッピング、および IDL からの WSDL の自動生成のサポート
- 起動および終了スタイル（RPC/literal、RPC/encoded、Doc/literal）などの Web サービスの動作のカスタマイズ、Java から XML へのマッピングのカスタマイズ、およびセキュリティの強化のためのソース・コードの注釈のサポート
- データベースおよび JMS の Web サービス

新しい J2EE 1.4 のアプリケーション管理およびデプロイ仕様のサポート

OC4J は、J2EE 環境でアプリケーションをデプロイおよび管理するための新しい標準を定義している、次の仕様をサポートしています。

- Java Management Extensions（JMX）1.2 仕様。これにより、J2EE 環境でサービスやアプリケーションなどのリソースを管理するための標準インタフェースを作成できます。JMX の OC4J 実装では、OC4J サーバーおよび OC4J サーバー内で実行されているアプリケーションを完全に管理できる JMX クライアントを提供しています。
- J2EE Management Specification（JSR-77）。JMX のサブセットで、J2EE 環境でアプリケーションを管理するための標準インタフェースを作成できます。
- J2EE Application Deployment API（JSR-88）。J2EE アプリケーションやモジュールを J2EE 互換の環境用に構成およびデプロイする標準 API を定義します。OC4J 実装には、コンポーネントを OC4J にデプロイする際に必要な OC4J 固有の構成データが含まれているデプロイ・プランの作成または編集（あるいはその両方）機能が含まれています。

Oracle Application Server TopLink のサポート

Oracle Application Server TopLink は、高機能なオブジェクトの永続化フレームワークで、幅広い Java 2 Enterprise Edition（J2EE）や Java アプリケーションのアーキテクチャに使用できます。OracleAS TopLink は、OC4J のコンテナ管理の永続性（CMP）コンテナと、Bean 管理の永続性（BMP）開発を単純化するベース・クラスもサポートしています。

OracleAS Job Scheduler

OracleAS Job Scheduler は、J2EE アプリケーションに非同期スケジュール・サービスを提供します。主な機能には、ジョブの発行、制御および監視機能が含まれます。ジョブとは、作業が行われる際に実行される作業単位と定義されています。

新しい 2 フェーズ・コミット・トランザクション・コーディネータ機能

OC4J の新しい分散トランザクション・マネージャでは、あらゆる種類の XA リソース間で 2 フェーズ・トランザクションを調整できます。XA リソースには、Oracle データベースに加え、他のベンダーや JMS プロバイダのデータベース（IBM WebsphereMQ など）が含まれます。また、障害時の自動トランザクション・リカバリもサポートしています。

Generic JMS Resource Adapter の拡張

Generic JMS Resource Adapter が、OC4J の最新バージョンに同梱される OracleAS JMS、および IBM Websphere MQ JMS バージョン 5.3 用の OC4J プラグインとして使用できるようになりました。

JMS 接続をキャッシュできるようにするため、そして同時にグローバル・トランザクションにも正常に参加できるようにするため、遅延トランザクション登録に対するサポートが追加されました。

さらに、Generic JMS Resource Adapter のエラー処理機能が向上しました。エンドポイントは、プロバイダまたはシステム障害後、自動的に再試行を行い、`onMessage()` エラーが正しく処理されます。

OC4J の機能

OC4J の最新のリリースでは、次の機能が提供されます。

J2EE のサポート

OC4J は、標準の J2EE の API をサポートおよび認証しています。これらの API を表 2-1 に示します。

表 2-1 OC4J がサポートする J2EE の API

J2EE の標準 API	OC4J でサポートされているバージョン
JavaServer Pages（JSP）	2.0
サーブレット	2.4
Enterprise JavaBeans（EJB）	2.1
Java Transaction API（JTA）	1.0
Java Message Service（JMS）	1.1
Java Naming and Directory Interface（JNDI）	1.2
Java Mail	1.1.2
Java Database Connectivity（JDBC）	2.0 拡張機能
Oracle Application Server Java Authentication and Authorization Service（JAAS）Provider	1.0
J2EE Connector Architecture	1.5

表 2-1 OC4J がサポートする J2EE の API（続き）

J2EE の標準 API	OC4J でサポートされているバージョン
Java API for XML-Based RPC（JAX-RPC）	1.1
SOAP with Attachments API for Java（SAAJ）	1.2
Java API for XML Registries（JAXR）	1.0.5

OC4J の Web 通信

OC4J では、Oracle HTTP Server を使用しなくても HTTP および HTTPS 通信をネイティブにサポートしています。

デフォルトの Web サイトは `http-web-site.xml` ファイルで定義されます。このファイルでデフォルトの HTTP リスナーはポート `8888` に指定されます。このファイルを応用して、追加の Web サイトを別のポートに定義できます。OC4J で追加の Web サイトを作成する手順は、『Oracle Containers for J2EE 構成および管理ガイド』を参照してください。

クラスタリング

OC4J では、OC4J インスタンスのクラスタ間で、HTTP セッション、ステートフル・セッションでの Enterprise JavaBeans レプリケーションおよびロード・バランシングをサポートしています。ただしこのリリースでは、クラスタの構成および管理はすべて手動で、OC4J 固有のアプリケーション構成ファイルを編集することによって行う必要があります。詳細は、『Oracle Containers for J2EE 構成および管理ガイド』を参照してください。

JSP のための Oracle の付加価値機能

JSP ページに関する OC4J の付加価値機能は、次の 3 つのカテゴリにグループ化できます。

- Oracle 固有の機能。
- カスタム・タグ・ライブラリ、カスタム JavaBeans またはカスタム・クラスの実装によって提供される機能。通常、他の JSP 環境に移植できます。
- キャッシング・テクノロジをサポートする機能。

次の各項では、これらのカテゴリの機能の概要、および JavaServer Pages 標準タグ・ライブラリ（JSTL）に対する Oracle サポートの概要について説明します。JSTL サポートの詳細は、『Oracle Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

サポートしている仕様

OC4J 付属の JSP コンテナは、Sun 社による次の仕様に完全に準拠しています。

- JavaServer Pages 仕様、バージョン 2.0
- Java サーブレット仕様、バージョン 2.4
- JSR-045: Debugging Support for Other Languages

詳細は、これらの仕様を参照してください。

Oracle 固有の機能

この項では、OC4J の Web コンテナがサポートする Oracle 固有のプログラミング拡張機能の概要を説明します。

OC4J の構成可能な JSP 拡張機能

JSP 仕様 2.0 準拠に加え、OC4J Web コンテナには、注目すべき次の機能があります。

2-4 ページの「JSP のための Oracle の付加価値機能」も参照してください。

- `web.xml` ファイルと TLD の XML 妥当性チェック用の個別のモード切替え

  `web.xml` の妥当性チェックは、デフォルトでは無効ですが、有効にできます。TLD の妥当性チェックはデフォルトでは無効です。

- 追加インポート用のモード・フラグ

  このフラグを使用して、JSP のデフォルト設定よりも多い Java パッケージを自動的にインポートします。

- タグ・ライブラリを共有するための予約済の場所

  複数の Web アプリケーションで共有するタグ・ライブラリ JAR ファイルを配置するディレクトリを指定できます。

- 構成可能な JSP タイムアウト

  JSP ページに対してタイムアウト値を指定できます。この値が経過した後に再度リクエストされなかった場合、そのページはメモリーから削除されます。

次の機能もサポートされています。

- ページの自動再変換と自動再ロードのためのモード切替え

  選択できるモードは、1）自動再ロードまたは JSP ページの自動再変換を行わずに JSP ページを実行する、2）ページ実装クラス（JavaBeans クラスまたは他の依存クラスを除く）を自動的に再ロードする、3）変更された JSP ページを自動的に再変換する、のいずれかです。

- タグ・ハンドラ・インスタンスのプーリング

  タグ・ハンドラの作成とガベージ・コレクションの時間を節約するために、タグ・ハンドラ・インスタンスのプーリングを必要に応じて有効にできます。これらは、`application` スコープにプールされます。ページごとに、または同じページのセクションごとに、異なる設定を使用できます。7-17 ページの「タグ・ハンドラの再利用（タグ・プーリング）の無効化または有効化」を参照してください。

- NULL 出力に対する出力モード

  JSP ページからの NULL 出力に対して、デフォルトの NULL 文字列のかわりに空の文字列を出力できます。

グローバル・インクルード

OC4J の Web コンテナでは、グローバル・インクルードと呼ばれる機能が提供されています。この機能によって、仮想の JSP `include` ディレクティブを使用して、指定のディレクトリ内またはディレクトリ下の JSP ページに静的にインクルードする 1 つ以上のファイルを指定できます。変換時に、Web コンテナは、インクルード・ファイルとディレクトリをページに指定する構成ファイル `/WEB-INF/ojsp-global-include.xml` を検索します。

この拡張機能は、以前の Oracle JSP リリースで `globals.jsa` または `translate_params` 機能を使用していたアプリケーションを移行する場合に、特に便利です。詳細は、5-7 ページの「Oracle JSP のグローバル・インクルード」を参照してください。

ダイナミック・モニタリング・サービスのサポート

DMS によって、OC4J も含めた多数の Oracle Application Server コンポーネントにパフォーマンス監視機能が追加されます。DMS の目的は、組込みのパフォーマンス測定値を介して、実行時の動作に関する情報を提供することにあります。これによって、ユーザーは、パフォーマンスに関する問題を診断、分析およびデバッグできます。DMS は、この情報を、デプロイ中も含めていつでも使用できるパッケージで提供します。データは HTTP を通じて公開され、ブラウザで表示できます。

OC4J の Web コンテナは DMS 機能をサポートし、関連する統計値を計算し、スパイ・サーブレットなどの DMS サーブレットに情報を提供して、エージェントを監視します。統計には、次の内容（必要に応じて、平均、最大および最小を使用）が含まれます。時間はミリ秒単位で示されます。

- HTTP リクエストの処理時間
- JSP サービス・メソッドの処理時間
- 作成済または使用可能な JSP インスタンスの数

これらのサーブレットの標準構成は、OC4J の `application.xml` および `default-web-site.xml` 構成ファイルにあります。Oracle Enterprise Manager 10*g* Application Server Control コンソールを使用して DMS にアクセスし、DMS 情報を表示して、必要に応じて DMS 構成を変更します。

JSP メトリックの正確な定義と、JSP メトリックを表示および分析する詳細な手順については、『Oracle Application Server パフォーマンス・ガイド』を参照してください。

OC4J が提供する JSP ユーティリティとタグ・ライブラリ

この項では、標準準拠のカスタム・タグ・ライブラリ、カスタム JavaBeans およびその他のクラスを介して実装される、OC4J JSP 拡張機能の概要について説明します。これらの機能の詳細は、『Oracle Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

- スコープを指定できる JavaBeans として実装された拡張型。
- イベント処理用の `JspScopeListener`。
- XML と XSL の統合。
- データ・アクセス用タグ・ライブラリ（SQL タグとも呼ばれる）と JavaBeans。
- Web サービス用タグ・ライブラリ。
- タグ・ライブラリと JavaBeans。ファイルのアップロード、ファイルのダウンロード、アプリケーション内からの電子メール送信に利用します。
- EJB タグ・ライブラリ。
- その他のユーティリティ・タグ（指定したロケールに適した日付や通貨を表示するタグなど）。

キャッシング・サポートに関するタグと API

Web でのパフォーマンスの問題に対応するために、E-Business では、コスト効率の高いテクノロジとサービスを導入して、インターネット・サイトのパフォーマンスを改善する必要があります。Web キャッシングは、この問題を解決するための主要なテクノロジで、静的および動的な Web コンテンツをキャッシングします。Web キャッシングの利点には、パフォーマンス、スケーラビリティ、高可用性、コスト削減およびネットワークの通信量の軽減があります。

OC4J は、Web キャッシング・テクノロジに対して、次のサポートを提供します。

- Edge Side Includes（ESI）用の JESI タグ・ライブラリ。XML スタイルのマークアップ言語で、動的なコンテンツを Web サーバーから独立してアセンブリできます。

  Oracle Web Cache は、ESI エンジンを備えています。

- Web Object Cache 用のタグ・ライブラリとサーブレット API。アプリケーション・レベルのキャッシュで、Java Web アプリケーション内に埋め込まれて維持されます。

  Web Object Cache は、Oracle Application Server の Java Object Cache をデフォルト・リポジトリとして使用します。

これらの機能の詳細は、『Oracle Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

JavaServer Pages 標準タグ・ライブラリ（JSTL）のサポート

OC4J JSP 実装は、Sun 社の JSTL 仕様で指定されているとおり、JavaServer Pages 標準タグ・ライブラリ（JSTL）をサポートしています。

注意： JSTL は、OC4J 内の *ORACLE_HOME*`/j2ee/home/jsp/lib/taglib` ディレクトリにインストールされません。

デプロイされた Web アプリケーション間でタグ・ライブラリを共有する手順は、7-26 ページの「Web アプリケーション間でのタグ・ライブラリの共有」を参照してください。

JSTL は、Java などのスクリプト言語に不慣れな JSP ページの作成者を対象にしています。以前は、JSP ページで動的データを処理するには、スクリプトレットが使用されていました。JSTL タグを使用すると、スクリプトレットが不要になります。

JSTL の主な機能は、次のとおりです。

- 式言語サポート、条件付きロジックとフローの制御、イテレータ操作および URL ベース・リソースへのアクセス用のコア・タグ
- XML 処理、フロー制御および XSLT 変換用のタグ
- データベース・アクセス用の SQL タグ
- I18N 対応の国際化および書式設定用のタグ

  （「I18N」は、国際化標準を指します。）

タグのサポートは、前述の機能に基づいて 4 つの JSTL サブライブラリに編成されています。

JSTL サポートの詳細は、『Oracle Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。JSTL の詳細は、次の Web サイト上の仕様を参照してください。

`http://www.jcp.org/aboutJava/communityprocess/first/jsr052/index.html`

注意： OC4J が提供する JML、XML、データ・アクセス（SQL）およびその他のカスタム・タグ・ライブラリは JavaServer Pages 標準タグ・ライブラリ（JSTL）以前のライブラリで、重複した機能があります。

標準に準拠するには、カスタム・ライブラリではなく、原則として JSTL を使用することをお薦めします。

JSTL ではまだ使用できないカスタム・ライブラリの機能のうち、有用と考えられる機能については、必要に応じて JSTL 標準に採用される予定です。

Oracle JDeveloper の JSP サポート

現在の Visual Java プログラミング・ツールは、JSP コーディングをサポートしています。特に、Oracle JDeveloper は、JSP 開発をサポートし、次の機能を備えています。

- OC4J の Web コンテナの統合。アプリケーション開発サイクル（JSP ページの編集、デバッグおよび実行）を完全にサポートします。
- デプロイされた JSP ページのデバッグ。
- データと Web に対応した JavaBeans の拡張セット。JDeveloper Web Bean と呼ばれます。
- JSP 要素ウィザード。事前定義済 Web Bean をページに追加する便利な方法です。
- カスタム JavaBeans の取込みのサポート。

デバッグでは、JDeveloper によって JSP ページ内にブレーク・ポイントを設定し、JSP ページから JavaBeans へのコールを追跡できます。これは、手動によるデバッグと比較して大変便利な方法です。たとえば、JSP ページに出力文を追加し、状態をレスポンス・ストリーム（ブラウザ表示用）やサーバー・ログ（暗黙的な `application` オブジェクトの `log()` メソッドを使用）に出力できます。

JDeveloper については、JDeveloper のオンライン・ヘルプ、または次の OTN（Oracle Technology Network）のサイトを参照してください。

`http://otn.oracle.com/products/jdev/content.html`

（この Web サイトを参照するには、OTN に登録する必要があります。無償で登録できます。）JDeveloper が提供する JSP タグ・ライブラリの概要は、『Oracle Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

注意： 他の主要な IDE ベンダーが、OC4J とシームレスに統合できるプラグイン・モジュールを構築しています。これにより、開発者は、IDE 内から直接、OC4J で実行中の J2EE アプリケーションをビルド、デプロイおよびデバッグできます。

Oracle の JSP リソース管理機能

次の各項では、リソース管理に関する標準機能と Oracle の付加価値機能について説明します。

標準セッションのリソース管理 : HttpSessionBindingListener

JSP ページでは、実行中に取得したリソース（JDBC 接続、文、結果セット・オブジェクトなど）の適切な管理が必要です。標準の `javax.servlet.http` パッケージには、session スコープのリソースを管理するために、`HttpSessionBindingListener` インタフェースと `HttpSessionBindingEvent` クラスが用意されています。たとえば、session スコープの問合せ Bean はこの機能を使用して、Bean がインスタンス化されるときにデータベース・カーソルを取得し、HTTP セッションが終了するときにそのカーソルをクローズできます。

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

注意 : Bean インスタンスは、HTTP セッション・オブジェクトのイベント通知リストに登録する必要がありますが、`jsp:useBean` 文によって自動的に登録されます。

---

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

`HttpSessionBindingListener` インタフェースを実装するオブジェクトは、`valueBound()` メソッドと `valueUnbound()` メソッドを実装できます。いずれのメソッドも、`HttpSessionBindingEvent` インスタンスを入力として取得します。これらのメソッドは、サーブレット・コンテナによってコールされます。`valueBound()` メソッドは、オブジェクトがセッションに格納されるときにコールされ、`valueUnbound()` メソッドは、オブジェクトがセッションから削除されるか、あるいはセッションがタイムアウトまたは無効になるとコールされます。開発者は通常、オブジェクトが保持するリソースを解放するために `valueUnbound()` メソッドを使用します（後述の例では、データベース接続の解放で使用します）。

次の「JDBCQueryBean JavaBean コード」の項では、`HttpSessionBindingListener` を実装する JavaBean のサンプルとその Bean をコールする JSP ページのサンプルを示します。

JDBCQueryBean JavaBean コード

次に、`JDBCQueryBean` のサンプル・コードを示します。これは、`HttpSessionBindingListener` インタフェースを実装する JavaBean です。ここでは、データベース接続用に JDBC OCI ドライバを使用しています。このサンプル・コードを実行する場合は、適切な JDBC ドライバと接続文字列を使用してください。

`JDBCQueryBean` は、HTML リクエストから検索条件を取得し（2-11 ページの「UseJDBCQueryBean JSP ページ」を参照）、その検索条件に基づいて動的なクエリーを実行し、結果を出力します。

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

```
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() +
          " 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-scope 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、サンプル用です。大規模な Web アプリケーションでデータベース接続のプーリングを処理する方法としては、必ずしもお薦めできません。

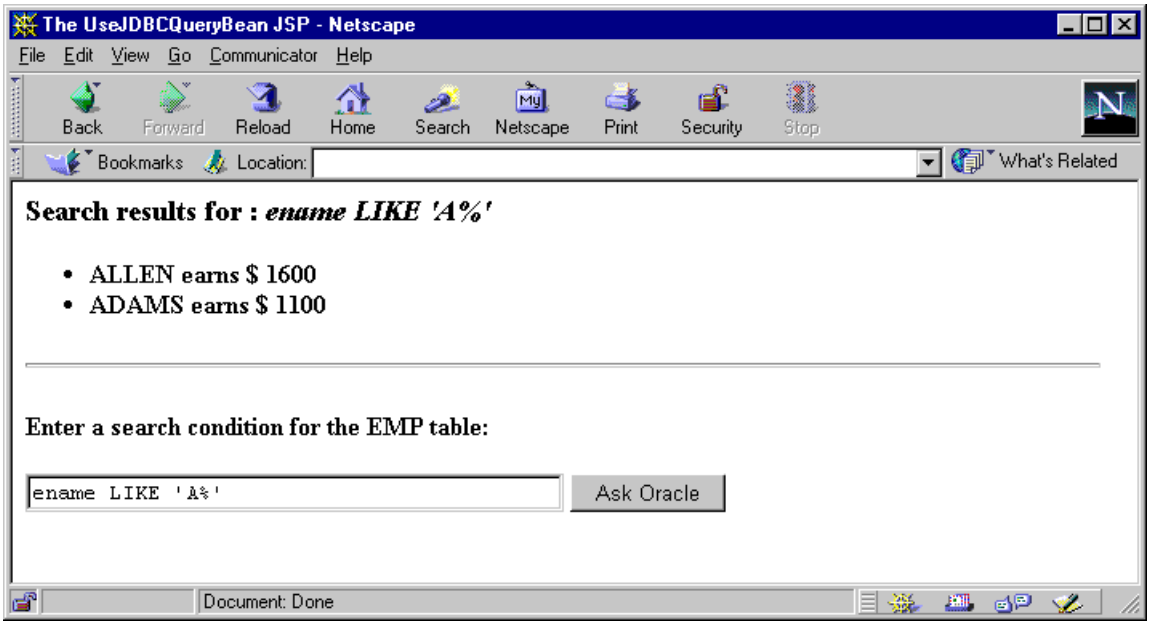
UseJDBCQueryBean JSP ページ

次の JSP ページでは、前の「JDBCQueryBean JavaBean コード」の項で定義した JDBCQueryBean JavaBean を使用し、session スコープを使用して Bean を起動します。ここでは、JDBCQueryBean を使用して、ユーザーが入力した検索条件と一致する従業員名を表示します。

JDBCQueryBean は、この JSP ページの jsp:setProperty タグから検索条件を取得します。このタグは、ユーザーが HTML フォームを使用して入力した searchCond リクエスト・パラメータの値に従って、Bean の searchCond プロパティを設定します。HTML の INPUT タグによって、フォームに入力された検索条件は searchCond と命名されます。

```
<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、このページの入力と出力のサンプルを示します。



HttpSessionBindingListener のメリット

前述の例では、`HttpSessionBindingListener` 機能のかわりに、JavaBean の `finalize()` メソッドで接続をクローズできます。`finalize()` メソッドは、セッションのクローズ後に Bean のガベージ・コレクションが実行されるとコールされます。ただし、`HttpSessionBindingListener` インタフェースの動作は、`finalize()` メソッドよりも確実に予測できます。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。これに対して、`HttpSessionBindingListener` インタフェースの `valueUnbound()` メソッドは、セッションの停止時に確実にコールされます。

リソース管理のための Oracle の付加価値機能の概要

OC4J では、OC4J JSP は `JspScopeListener` インタフェースを提供し、application スコープ、session スコープ、request スコープまたは page スコープのリソースを管理します。

この機能は、サーブレットと JSP 標準に従って、`page`、`request`、`session` または `application` スコープのオブジェクトをサポートします。他のスコープとともに session スコープをサポートするクラスを作成するために、クラスに `JspScopeListener` インタフェースと `HttpSessionBindingListener` インタフェースを実装して、両方のインタフェースを統合できます。OC4J または JServ 環境での `page` スコープの場合は、Oracle 固有の実行時実装を使用することもできます。

`HttpSessionBindingListener` の構成と統合方法については、『Oracle Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

3

OC4J の JSP 環境の構成

この章では、JSP 環境での基本事項である、主なサポート・ファイル、主な OC4J 構成ファイルおよび Web コンテナの構成について説明します。また、アプリケーション・ルート機能、クラスパス機能、セキュリティ上の問題およびファイルのネーミング規則など、初期段階での考慮事項についても説明します。

この章には、次の項目が含まれます。

注意： JSP ページは、HTTP 1.0 以上をサポートする標準ブラウザを使用して実行します。JSP ページ内のすべての Java コードは Web サーバーで実行されるため、ユーザーの Web ブラウザの JDK または Java 環境は関係ありません。

OC4J JSP コンテナの構成

この項では、OC4J JSP コンテナの構成オプションを説明します。特に JSP 開発用にコンテナを構成する場合を中心に説明します。この項には、次の項目が含まれます。

JSP 構成パラメータの概要

OC4J では、JSP コンテナ環境を構成するためのパラメータがいくつか用意されています。表 3-1 に、サポートされている構成パラメータのサマリーを示します。

この表の「**Application Server Control コンソールの JSP プロパティ**」列に記されているパラメータは、OC4J 付属の Oracle Enterprise Manager 10*g* Application Server Control コンソール管理インタフェースの「JSP コンテナのプロパティ」ページで設定できます。詳細は、3-7 ページの「Application Server Control コンソールでの JSP パラメータの設定」を参照してください。

Application Server Control コンソールからアクセスできないパラメータは、デフォルトで *ORACLE_HOME*/j2ee/home/config ディレクトリ内にある global-web-application.xml 構成ファイルで直接設定できます。このファイルで持続された JSP パラメータは、OC4J サーバー・インスタンス内で実行されるすべての Web モジュールによって継承されるデフォルト値となります。

いずれのグローバル設定も、Web モジュールの web.xml または orion-web.xml デプロイメント・ディスクリプタを変更することにより、特定の Web アプリケーションでオーバーライドできます。これらのファイルは手動で編集する必要があります。Application Server Control コンソールでは編集機能を提供していません。

global-web-application.xml または orion-web.xml ファイルでパラメータを設定する方法の詳細は、3-8 ページの「XML 構成ファイルでの JSP パラメータの設定」を参照してください。

表 3-1 JSP 環境の構成パラメータ

Application Server Control コンソール の JSP プロパティ	XML パラメータ	説明
(なし)	check_page_scope <init-param>	true に設定すると、OC4J 内で JspScopeListener による page スコープ・チェックが有効になります。デフォルトは false です。 JspScopeListener ユーティリティの詳細は、『Oracle Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。
(なし)	debug <init-param>	JSR-45 デバッグ・サポートを有効にするために設定します。有効な値は次のとおりです。 ■ file: SMAP ファイルを生成します。 ■ class: 生成されたクラス・ファイルにデバッグを埋め込みます。 ■ none（デフォルト）: デバッグ情報は生成されません。
デバッグ・モード	debug_mode <init-param>	true に設定すると、実行時に特定の例外が発生した場合、スタック・トレースが出力されます。デフォルトは false です。 このパラメータが false に設定されていて、ファイルが見つからない場合、見つからないファイルのフルパスは表示されません。このことは、存在しない JSP ファイルがリクエストされたときにファイルの物理的なパスを表示しない場合、セキュリティ上の重要な考慮事項です。

表 3-1 JSP 環境の構成パラメータ（続き）

Application Server Control コンソールの JSP プロパティ	XML パラメータ	説明
静的コンテンツの外部リソース	external_resource <init-param>	true に設定すると、変換時に、JSP トランスレータは、生成されたページ実装クラスではなく Java リソース・ファイルに JSP ページ内の静的なコンテンツを格納します。デフォルトは false です。 external_resource パラメータは、Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。 トランスレータは、リソース・ファイルを、生成されたクラス・ファイルとして同じディレクトリに格納します。リソース・ファイル名は、JSP ページ名に拡張子 .res を付けた名前になります。 たとえば、MyPage.jsp を変換すると、通常の出力以外に、_MyPage.res が作成されます。（正確な実装は、今後のリリースで変更される場合があります。） 1 ページに大量の静的なコンテンツが存在する場合は、この技法によって変換速度が向上し、ページの実行速度も向上します。詳細は、6-9 ページの「大量の静的なコンテンツまたはタグ・ライブラリの使用の管理」を参照してください。
追加インポート・パッケージ・リスト	extra_imports <init-param>	6-3 ページの「OC4J でデフォルトでインポートされるパッケージ」で説明されている JSP のデフォルトより多いパッケージをインポートします。 extra_imports パラメータは、Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。 XML ファイルでは、名前には、カンマまたは空白のデリミタが使用できます。たとえば、次のサンプルは、いずれも有効です。 <init-param> <param-name>extra_imports</param-name> <param-value>java.util.* java.beans.*</param-value> </init-param> または <init-param> <param-name>extra_imports</param-name> <param-value>java.util.*,java.beans.*</param-value> </init-param>
重複するディレクティブ属性を受け入れます	forgive_dup_dir_attr <init-param>	true に設定すると、単一の JSP 変換単位内で同じディレクティブ属性の設定が重複していて値が異なる場合に、JSP 変換エラーを回避します。デフォルトは false です。 forgive_dup_dir_attr パラメータは、Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。
XML の検証	xml_validate <init-param>	このブール値を true に設定すると、web.xml ファイルの XML 妥当性チェックが実行されます。デフォルトは false で、web.xml の妥当性チェックは実行されません。 このパラメータは将来のリリースで削除される予定です。これは、リリース 10.1.3.1.0 でこの動作を実装する唯一の方法です。この動作を実装する場合は、このパラメータが削除されたリリースにアップグレードする際に、コードを変更する必要があります。
（なし）	iso-8859-1-convert <init-param>	このブール型パラメータを false に設定するとバイトの切捨てが指定され、パフォーマンスが向上します。文字列に文字が含まれていない一般的な使用例のほとんどに対し、このバイト切捨てを指定することをお薦めします。デフォルト値は true で、エンコーダを使用した完全な文字変換が実行されます。

表 3-1 JSP 環境の構成パラメータ（続き）

Application Server Control コンソール の JSP プロパティ	XML パラメータ	説明
JSP を変更するタイミング	`main_mode <init-param>`	変更が行われた場合に、JSP 生成クラスを自動的に再ロードするか、または JSP ページを自動的に再変換するかを指定するために使用します。 有効にすると、Web アプリケーションを再デプロイまたは再起動せずに、新規または変更された JSP を OC4J ランタイムにロードできます。追加情報は、3-11 ページの「OC4J での実行時の JSP 再変換および再ロードの構成」を参照してください。 有効な設定は次のとおりです。 ■ `recompile`（デフォルト）: コンテナは、JSP ページのタイムスタンプをチェックし、最後のロード以降に変更されている場合はそのページを再変換および再ロードします。`reload` で説明されている機能も実行されます。 ■ `reload`: コンテナは、ページ実装クラスなど、JSP トランスレータによって生成されたクラスのタイムスタンプをチェックして、最後のロード以降に変更または再デプロイされたクラスを再ロードします。これは、開発環境から本番環境に、JSP ページではなくコンパイル済クラスをデプロイまたは再デプロイする場合などに便利です。 ■ `justrun`: コンテナは、タイムスタンプ・チェックを実行しません。したがって、JSP ページの再変換または JSP 生成 Java クラスのリロードはありません。JSP ページを頻繁に変更しない本番環境では、このモードが最も効率的です。
（なし）	`no_tld_xml_validate <init-param>`	`true` に設定すると、TLD の XML 妥当性チェックは実行されません。デフォルトは `true` で、TLD の XML 構造の妥当性チェックは実行されません。 このパラメータは将来のリリースで削除される予定です。これは、リリース 10.1.3.1.0 でこの動作を実装する唯一の方法です。この動作を実装する場合は、このパラメータが削除されたリリースにアップグレードする際に、コードを変更する必要があります。
（なし）	`old_include_from_top <init-param>`	`true` に設定すると、Oracle9*i*AS リリース 2 より前のリリースの動作との下位互換性を維持するために、ネストされた `include` ディレクティブ内のページの場所がトップレベルのページに関連づけられます。デフォルトは `false` です。 `old_include_from_top` パラメータは、Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。
プリコンパイル・チェック	`precompile_check <init-param>`	このブール型パラメータを `true` に設定すると、標準の `jsp_precompile` 設定に対して HTTP リクエストがチェックされます。デフォルトは `false` です。 `precompile_check` が `true` で、リクエストによって `jsp_precompile` が使用可能な場合、JSP ページは実行されず、事前変換されるのみです。`precompile_check` を `false` に設定すると、パフォーマンスが改善され、リクエストの `jsp_precompile` の設定値は無視されます。
カスタム・タグのコード・サイズの削減	`reduce_tag_code <init-param>`	このブール型パラメータを `true` に設定すると、カスタム・タグを使用するために生成されるコードのサイズがさらに縮小します。デフォルトは `false` です。

表 3-1 JSP 環境の構成パラメータ（続き）

Application Server Control コンソール の JSP プロパティ	XML パラメータ	説明

（なし）	req_time_introspection <init-param>	このブール型パラメータを true に設定すると、コンパイル時にイントロスペクションが実行できない場合、リクエスト時に JavaBean のイントロスペクションが有効になります。ただし、有効なコンパイル時のイントロスペクションが正常終了した場合、このフラグの設定に関係なく、リクエスト時のイントロスペクションは実行されません。デフォルトは false です。
		
		リクエスト時のイントロスペクションの使用例として、タグ・ハンドラによって、タグ補足情報クラスの VariableInfo にある一般的な java.lang.Object インスタンスが変換時とコンパイル時に戻されますが、実際には、特定のオブジェクトが実行時に生成される場合を想定します。この場合、req_time_introspection が有効になっていると、Web コンテナはリクエスト時までイントロスペクションを遅延します。
		
		このフラグには、if..then..else ループの別の分岐などで、Bean を 2 回宣言できるという効果もあります。次の例を考えてみます。req_time_introspection のデフォルト値 false を使用すると、このコードにより解析例外が発生します。true 値を使用すると、コードはエラーなしで機能します。
		
		`<% if (cond) { %>`
		`  <jsp:useBean id="foo" class="pkgA.Foo1" />`
		`<% } else { %>`
		`  <jsp:useBean id="foo" class="pkgA.Foo2" />`
		`<% } %>`

（なし）	setproperty_onerr_continue <init-param>	このブール型パラメータを true に設定すると、property="*" の場合、jsp:setProperty の使用時にエラーが発生したとき、リクエスト・パラメータの反復と対応する Bean プロパティの設定が継続されます。デフォルトは false です。 setproperty_onerr_continue パラメータは Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。
静的テキストを文字として生成	static_text_in_chars <init-param>	このブール型パラメータを true に設定すると、JSP トランスレータは、JSP ページの静的なテキストをバイトではなく文字として生成します。デフォルトは false です。 次の例に示すように、実行時に文字エンコードをアプリケーションで動的に変更する必要がある場合は、このフラグを有効にします。 `<% response.setContentType("text/html; charset=UTF-8"); %>` デフォルト設定の false を使用すると、静的なテキスト・ブロックの出力で、パフォーマンスが改善されます。
（なし）	jsp-print-null <init-param>	このフラグを false に設定すると、JSP ページからの NULL 出力に対して、NULL 文字列ではなく空の文字列が出力されます。デフォルトは true です。 jsp-print-null パラメータは、Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。

表 3-1 JSP 環境の構成パラメータ（続き）

Application Server Control コンソール の JSP プロパティ	XML パラメータ	説明
デフォルトでタグを再利用	tags_reuse_default <init-param>	このパラメータを使用して、タグ・ハンドラの再利用（タグ・プーリングとも呼ばれる）のモードを指定します。 ■ タグ・ハンドラ再利用のコンパイル時モデルをその基本モードで有効にするには、compiletime に設定します。これがデフォルト値です。 ■ タグ・ハンドラ再利用のコンパイル時モデルを、その「解放」モードで有効にするには、compiletime_with_release に設定します。この場合、タグ・ハンドラの release() メソッドは、指定したページで指定したタグ・ハンドラが使用されている間にコールされます。 ■ タグ・ハンドラの再利用を無効にするには、none または false に設定します。この値は、特定の JSP ページで、JSP ページ・コンテキスト属性 oracle.jsp.tags.reuse を true の値に設定するとオーバーライドできます。 ■ runtime オプションと、同等の値 true は、Oracle Containers for J2EE 10*g*（10.1.3.1.0）からサポートされなくなりました。
JSP ページのタイムアウト	ルートの <orion-web-app> 要素の jsp-timeout 属性	時間を示す 1 以上の整数値（秒単位）を指定すると、この値が経過した後にリクエストされなかった場合、その JSP ページはメモリーから削除されます。デフォルトは 0 です。
永続 TLD キャッシング	ルートの <orion-web-app> 要素の jsp-cache-tlds 属性	OC4J には TLD の永続的なキャッシング機能があり、予約済のタグ・ライブラリの場所における TLD のグローバル・キャッシュ、および TLD を使用するアプリケーションに対するアプリケーション・レベルでのキャッシュを使用できます。詳細は、7-29 ページの「TLD キャッシング機能の有効化」を参照してください。 デフォルトでは、アプリケーションは、グローバル Web アプリケーション・レベルで設定された TLD キャッシングの値を継承します。ただし、TLD キャッシングがアプリケーション・レベルで設定されると、この値でグローバル設定がオーバーライドされます。 ■ standard（デフォルト）に設定すると、Web アプリケーションの /WEB-INF ディレクトリで、.tld 拡張子を持つファイルが検索されます。検出された TLD は、グローバル Web アプリケーションから継承された TLD のリストに追加されます。 /WEB-INF/lib および /WEB-INF/classes ディレクトリでは、.tld 拡張子を持つファイルは検索されません。 10*g* リリース 3（10.1.3）では、これがグローバル・レベルおよびアプリケーション・レベルの両方に設定されているデフォルト値です。 ■ on に設定すると、Web アプリケーション内のすべてのディレクトリで TLD が検索されます。検出された TLD は、グローバル Web アプリケーションから継承された TLD のリストに追加されます。 ■ off に設定すると、永続 TLD キャッシングが無効になります。/WEB-INF/lib および /WEB-INF/classes ディレクトリで、.tld 拡張子を持つファイルは検索されません。 このパラメータの値は、次の「タグ・ライブラリの場所」（jsp-taglib-locations）の解説で説明する複数の予約済のタグ・ライブラリを設定する際に影響を及ぼします。

表 3-1 JSP 環境の構成パラメータ（続き）

Application Server Control コンソール の JSP プロパティ	XML パラメータ	説明
タグ・ライブラリの場所リスト	ルートの <orion-web-app> 要素の jsp-taglib-locations 属性	JSP 仕様で規定されている標準的な JSP の予約済 URI 機能の拡張機能として、OC4J JSP コンテナは、1 つ以上の予約済のタグ・ライブラリの使用をサポートします。予約済のタグ・ライブラリはタグ・ライブラリ JAR ファイルを格納するディレクトリで、これによってライブラリを複数の Web アプリケーション間で共有できます。詳細は、7-29 ページの「予約済のタグ・ライブラリの場所の指定」を参照してください。 TLD キャッシングが無効になっている場合（off）、予約済のタグ・ライブラリの場所は、単一のディレクトリ（デフォルトで *ORACLE_HOME*/j2ee/home/jsp/lib/taglib）に制限されます。 TLD キャッシングが有効になっている場合（standard または on）、セミコロンをデリミタとして使用したディレクトリ・パスのリストを使用して、1 つ以上の予約済のタグ・ライブラリの場所を指定できます。 jsp-taglib-locations 属性は、global-web-application.xml のみで設定できます。個別の Web モジュールの orion-web.xml ファイルでは設定できません。そうした場合、場所は無視されます。 予約済のディレクトリ内に格納するタグ・ライブラリは、JAR ファイルにパッケージ化する必要があります。
（なし）	ルートの <orion-web-app> 要素の simple-jsp-mapping 属性	*.jsp ファイル拡張子が、アプリケーションに影響する Web ディスクリプタの <servlet> 要素（global-web-application.xml、web.xml および orion-web.xml）の oracle.jsp.runtimev2.JspServlet フロントエンド JSP サーブレットのみにマッピングされる場合は、true に設定します。これにより、JSP ページのパフォーマンスが改善されます。デフォルトの設定は true です。 *.jsp 拡張子を別のサーブレットにマップする必要がある場合は、このパラメータを false に設定します。

Application Server Control コンソールでの JSP パラメータの設定

Oracle Enterprise Manager 10*g* Application Server Control コンソールは、Oracle Application Server インスタンス用の管理ツールです。デフォルトで、OC4J とともにインストールされます。

ログイン後、**「管理」→「JSP プロパティ」**リンクから「JSP コンテナのプロパティ」ページにアクセスします。

次に、コンソールにアクセスする手順を説明します。

- スタンドアロン OC4J での Application Server Control コンソールへのアクセス
- Oracle Application Server での Application Server Control コンソールへのアクセス

スタンドアロン OC4J での Application Server Control コンソールへのアクセス

OC4J ソフトウェアをインストールすると、Application Server Control コンソールが自動的にインストールおよび構成されます。OC4J を起動すると、デフォルトでコンソールも起動されます。

コンソールには、ポート 8888 で HTTP リクエストをリスニングするよう構成されている `default` の Web サイトからアクセスします。コンソールにアクセスするには、Web ブラウザで次の URL を入力します。

```
http://hostname:8888/em
```

Oracle Application Server での Application Server Control コンソールへのアクセス

Oracle Universal Installer を使用して OC4J をインストールすると、Application Server Control コンソールが自動的にインストールおよび構成されます。

コンソールは、OPMN コマンドライン・ツール `opmnctl` を使用して、すべてのインストール済の Oracle Application Server コンポーネントと同時に起動されます。このツールは、各サーバー・ノードの *ORACLE_HOME*`/opmn/bin` ディレクトリにインストールされています。次のコマンドを発行して、すべてのインストール済のコンポーネントを起動します。

```
cd ORACLE_HOME/opmn/bin
opmnctl startall
```

一般的な Oracle Application Server インストールでは、Application Server Control コンソールを含めたすべての Web アプリケーションには、Oracle HTTP Server（OHS）からアクセスします。コンソールにアクセスするには、次の URL を使用します。

```
http://ohs_host_address:port/em
```

- *ohs_host_address* は、OHS のホスト・マシンのアドレスです。たとえば、`server07.company.com` のようになります。
- *port* は、OPMN によって OHS に割り当てられた HTTP リスナー・ポートです。OHS のホスト・マシンで次の `opmnctl` コマンドを実行すると、OPMN によって割り当てられたリスナー・ポートのリストを取得できます。

  ```
  opmnctl status -l
  ```

  OPMN のステータス出力で `http1` に指定されているポートを、*port* の値に指定します。

  ```
  HTTP_Server | HTTP_Server | 6412 | Alive | 1970872013 | 1
  6396 | 0:48:01 | https1:4443,http2:722,http1:7779
  ```

XML 構成ファイルでの JSP パラメータの設定

OC4J 開発環境では、JSP 構成パラメータをグローバル構成ファイルおよびモジュール固有の構成ファイルで直接設定できます。

`global-web-application.xml` および `orion-web.xml` の詳細は、『Oracle Containers for J2EE サーブレット開発者ガイド』を参照してください。

サーブレット初期化パラメータの設定

OC4J インスタンス内でインスタンス化された各 Web モジュールに対し、フロントエンド・サーブレット・クラス `oracle.jsp.runtimev2.JspServlet` のインスタンスが作成されます。OC4J でサポートされているサーブレットは、このサーブレット・クラスのみです。

サーブレット・インスタンスを初期化するためのデフォルト・パラメータは、`global-web-application.xml` 構成ファイル内の `<servlet>` 要素のサブ要素 `<init-param>` で指定されています。

Web アプリケーション・レベルのすべてのデフォルト・サーブレット・パラメータは、Web モジュールとともにインストールされている J2EE 標準の `web.xml` デプロイメント・ディスクリプタ内の対応する `<init-param>` 要素を指定することにより、オーバーライドできます。こ

のファイルのデフォルトの場所は、
ORACLE_HOME/home/j2ee/ *appName*/*webModuleName*/WEB-INF ディレクトリです。

次の例では、JSP フロントエンド・サーブレットの <servlet> 要素内で、サーブレット初期化パラメータが含まれている <init-param> 要素を設定する方法を示しています。このサンプルでは、precompile_check フラグを有効にし、タイムスタンプをチェックせずに実行するために main_mode フラグを設定して、Java コンパイラを冗長モードで実行します。

```
<servlet>
   <servlet-name>jsp</servlet-name>
   <servlet-class>oracle.jsp.runtimev2.JspServlet</servlet-class>
   <init-param>
      <param-name>precompile_check</param-name>
      <param-value>true</param-value>
   </init-param>
   <init-param>
      <param-name>main_mode</param-name>
      <param-value>justrun</param-value>
   </init-param>
   <init-param>
      <param-name>javaccmd</param-name>
      <param-value>javac -verbose</param-value>
   </init-param>
</servlet>
```

注意： javaccmd パラメータは、Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。

JSP 構成パラメータの設定

JSP のタイムアウトや、TLD キャッシングの有効性などの機能を制御するパラメータは、global-web-application.xml または Web モジュールの orion-web.xml ファイルのいずれかの、ルートの <orion-web-app> 要素の属性として設定されます。サーブレットの初期化パラメータと同様、orion-web.xml で設定された属性は、global-web-application.xml 内の対応するグローバル定義をオーバーライドします。デプロイされた Web モジュールのファイルは、デフォルトで、
ORACLE_HOME/j2ee/home/application-deployments/*appName*/*webModuleName* ディレクトリ内にあります。

次の例では、XML ファイル内でルートの <orion-web-app> 要素の属性を設定する方法を示しています。

```
<orion-web-app development="false" jsp-timeout="30" ... >
 ...
</orion-web-app>
```

<ojsp-init>

orion-web.xml ファイルでは、<ojsp-init> 要素を使用して、表 3-2 に示す JSP 構成パラメータを設定できます。

表 3-2 JSP サーブレットの構成パラメータ

<ojsp-init> 属性	同等の JSP サーブレット <init-param>	ojspc コマンドライン・オプション
debug-mode	debug_mode	n/a
iso-8859-1-convert	iso-8859-1-convert	n/a
jsr45-debug	debug	-debug
main-mode	main_mode	n/a
precompile-check	precompile_check	n/a
reduce-tag-code	reduce_tag_code	-reduceTagCode
req-time-introspection	req_time_instrospection	-reqTimeIntrospection
static-text-in-chars	static-text-in-chars	-staticTextInChars
tags-reuse	tags_reuse_default	-tagReuse

これらのパラメータの詳細は、3-2 ページの「JSP 構成パラメータの概要」を参照してください。

Web アプリケーションの orion-web.xml ファイルで <ojsp-init> を指定すると、デフォルト値を含めたこれらの属性値が、Web モジュールとともにインストールされた web.xml デプロイメント・ディスクリプタの <init-param> 要素で指定されている対応する JSP 構成パラメータの値をオーバーライドします。ojspc 事前変換ユーティリティの対応するコマンドライン・オプションは、<ojsp-init> 属性および web.xml の対応する設定をオーバーライドします。

注意： global-web-application.xml の <ojsp-init> 要素は使用できません。

<ojsp-config>

Oracle 独自の <ojsp-config> 要素は orion-web.xml ファイルに定義されています。<ojsp-config> のセマンティクスは、JSP 2.0 で導入された web.xml ファイルの <jsp-config> 要素に類似しています。

<default-buffer-size> プロパティは、<url-pattern> によって管理される一連の JSP の JSP ライター・オブジェクトにより使用されるデフォルトのバッファ・サイズを指定します。現在（リリース 10.1.3.1）、<ojsp-config> で設定できるものは、<default-buffer-size> のみです。<default-buffer-size> 設定は、<jsp-config> 要素では使用できません。

個々の JSP ページの <default-buffer-size> 設定は、page ディレクティブの buffer 属性を使用してオーバーライドできます。

page ディレクティブを使用したバッファ・サイズの設定例です。

```
<%@ page buffer="7kb" %>
```

orion-web.xml ファイルの <ojsp-config> 要素の構文の例です。

```
<ojsp-config>
   <ojsp-property-group>
       <url-pattern> ...</url-pattern>
       <default-buffer-size>..</default-buffer-size>
   </ojsp-property-group>
     ....
    multiple <ojsp-property-group> can exist
</ojsp-config>
```

OC4J での JSP のコンパイルの構成

Java コンパイラの起動方法には、OC4J と同じ JVM プロセス内で実行するインプロセス・コンパイルと、別の JVM プロセスで実行するアウトプロセス・コンパイルがあります。

デフォルトでは、OC4J でアウトプロセス・コンパイルを使用し、コンパイラは個別の実行可能ファイルとして起動します。使用されるデフォルトのコンパイラは、Sun 社の JDK 付属の `javac` です。

OC4J で別のコンパイラを使用するか、インプロセス・コンパイルを使用するには、`<java-compiler>` 要素を任意のオプション設定で OC4J の `server.xml` ファイルに追加します。詳細は、『Oracle Containers for J2EE 構成および管理ガイド』を参照してください。

OC4J での実行時の JSP 再変換および再ロードの構成

OC4J では、アプリケーションを再デプロイまたは再起動せずに、アクティブに稼働中の Web モジュールに新しい JavaServer Pages（JSP）を追加したり、既存の JSP を変更できます。

この機能を使用するには、OC4J インスタンス内で展開された WAR ファイル構造内の適切なディレクトリに、新規または更新済の JSP を追加します。追加先のディレクトリは、*`ORACLE_HOME`*`/j2ee/`*`instance`*`/applications/`*`appName`*`/`*`webModuleName`*`/` です。OC4J により、ページが変換およびロード（または再ロード）されます。

この機能は、`main_mode` サーブレット初期化パラメータによって制御されます。有効な設定は次のとおりです。

- `recompile`（デフォルト）: 変更された JSP ページを再変換します。
- `reload`: Web コンテナによって生成され、変更されたクラス（ページの実装クラスなど）を再ロードします。
- `justrun`: 再変換または再ロードせずに稼働します。本番環境などでは、最適なパフォーマンスを提供できる場合があります。

`main_mode` パラメータの追加情報は、3-2 ページの「JSP 構成パラメータの概要」を参照してください。

注意：

- メモリー内の値を使用して、クラス・ファイルの最終変更時間を保持するため、ページ実装クラス・ファイルをファイル・システムから削除しても、関連する JSP ページ・ソースは自動的に再変換されません。
- ページ実装クラス・ファイルは、メモリー・キャッシュが失われると再生成されます。この再生成は、サーバーの再起動後やこのアプリケーション内の別のページが再変換された後に、リクエストがこのページにダイレクトされるたびに発生します。
- OC4J では、静的にインクルードされたページ（つまり、`include` ディレクティブを使用してインクルードされたページ）が更新された場合、インクルード先のページは、次の起動時に自動的に再変換されます。

サーブレット・レイヤーでのクラス・ローダーの動作の詳細は、『Oracle Containers for J2EE サーブレット開発者ガイド』を参照してください。

OC4J が提供する主な JSP 関連のサポート・ファイル

この項では、Web コンテナまたは JSP アプリケーションで使用する JAR ファイルと ZIP ファイルの概要を説明します。これらのファイルは、システム上で OC4J のクラスパスにインストールされます。

- `ojsp.jar`: Web コンテナ用のクラス。
- `ojsputil.jar`: OC4J が提供するタグ・ライブラリとユーティリティ用のクラス。
- `xmlparserv2.jar`: XML 解析用。`web.xml` デプロイメント・ディスクリプタ・ファイル、タグ・ライブラリ・ディスクリプタ、および XML 関連タグ機能で必要です。
- `ojdbc14.jar` / `classes12.jar` / `classes111.jar`: Oracle JDBC ドライバ用（それぞれ JDK 1.4、1.2 以上用）。
- `runtime12.jar` / `runtime12ee.jar` / `runtime11.jar` / `runtime.jar` / `runtime-nonoracle.jar`: Oracle SQLJ ランタイム用（それぞれ JDK 1.2.x 以上と Oracle9*i* 以上の JDBC 用、JDK 1.2.x 以上の Enterprise Edition と Oracle9*i* 以上の JDBC 用、またはすべての JDK 環境と Oracle JDBC 以外のドライバ用）。
- `jndi.jar`: リソース（JDBC データ・ソース、Enterprise JavaBeans など）をルックアップするための JNDI サービス用。
- `jta.jar`: Java Transaction API 用。

特定のタグ・ライブラリなど、特定の領域に関連したファイルもあります。これには、次のファイルが含まれます。

- `mail.jar`: アプリケーション内での電子メール機能用（標準の `javax.mail` パッケージ）
- `activation.jar`: 電子メール機能用の Java アクティブ化ファイル
- `cache.jar`: Oracle Application Server の Java Object Cache 用（OC4J の Web Object Cache 用のデフォルトのバックエンド・リポジトリ）

4

ojspc による JSP ページのプリコンパイル

この章では、OC4J 付属の ojspc ユーティリティによって提供されている JSP の事前変換およびプリコンパイル機能について説明します。次の各項で、ojspc 機能について説明します。

ojspc ユーティリティの仕組み

この章では、ojspc によって提供されている基本的なプリコンパイル機能と、アーカイブ・ファイルを使用する JSP ページのバッチ・プリコンパイルについて説明します。次の項が含まれます。

基本的な ojspc 機能の概要

単純な JSP ページの場合、ojspc のデフォルトの機能は次のとおりです。

- 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、.java ファイルを作成します。
- Java コンパイラを起動して .java ファイルをコンパイルし、ページ実装クラスに対する .class ファイルを作成します。

ojspc は、デフォルトでは、JSP トランスレータがオンデマンド変換で生成したものと同じファイル・セットを生成し、ojspc を実行したときのカレント作業ディレクトリ内またはその下にそのファイル・セットを格納します。

出力には、次のファイルが含まれます。

- .java ソース・ファイル（バッチ事前変換の場合はコンパイル後に破棄されます。）
- ページ実装クラスに対する .class ファイル
- 必要に応じて、ページの静的なテキストの Java リソース・ファイル（.res）

特定の状況（この章で後述する -extres オプションの説明を参照）では、ojspc オプションによって、JSP トランスレータは、このコンテンツをページ実装クラスに格納せずに、静的なページのコンテンツに対する .res Java リソース・ファイルを作成します。

WAR ファイルのバッチ事前変換の概要

ojspc ユーティリティは、JSP ファイルの変換を行うのみでなく、WAR または EAR ファイルのバッチ事前変換も実行できます。処理後の .class ファイルとすべての Java リソース・ファイルは、WAR ファイル内のネストされた JAR ファイルに出力されます。ネストされた JAR ファイルは、WAR 内の WEB-INF\lib パス内に追加されるため、アーカイブはそのまま OC4J にデプロイできます。

ojspc のコマンドラインにアーカイブ・ファイル名が含まれている場合、ojspc は、デフォルトで次の手順を実行します。

1. アーカイブ・ファイルを開きます。
2. アーカイブ内の .jsp および .java ファイルをすべて変換およびコンパイルします。
3. アーカイブ内のタグ・ファイルも、アーカイブ内の JSP で使用されている場合、やはりコンパイルされます。
4. WEB-INF\lib パスに新しいネストされた JAR ファイルを追加してアーカイブを更新し、出力された .class ファイルとすべての Java リソース・ファイルをこのファイルに追加します。この過程で作成されたすべての .java ファイルは破棄されます。

ネストされた JAR ファイルの名前には、出力されたアーカイブ・ファイルのベース名が含まれ、.jar 拡張子が付加されます。たとえば、ojspc が sample.war に対して実行された場合、WAR 内のネストされた JAR ファイルの名前は __oracle_jsp_sample.jar となります。

ネストされた JAR ファイル内のファイル・パスは、Java パッケージ名と、JSP include 文と forward 文の指定したファイル・パスに基づきます。ネストされた JAR ファイル内の .class ファイルとリソース・ファイルは、元の JSP ファイルが抽出後に変換された場合と同じディレクトリ・パスに配置されます。

ojspc の使用

ojspc 実行可能ファイルの場所がホスト・マシンの PATH に追加されていれば、ojspc ユーティリティを任意のディレクトリから起動できます。一般的な ojspc のコマンドライン構文は、次のとおりです。

```
ojspc [option_settings] file_list
```

ソース・ファイル・リストには JSP ファイルと他のソース・ファイル（.java）、またはアーカイブ・ファイル（JAR、WAR、EAR または ZIP ファイル）を含めることができます。ディレクトリ内のすべての JSP ファイルをプリコンパイルするには、file_list の値に *.jsp を指定します。

個別の ojspc の使用例は、次の各項を参照してください。

- 1 つ以上の JSP のプリコンパイル
- WAR ファイル内の JSP のプリコンパイル

重要：

- ojspc を任意の作業ディレクトリから起動するには、ojspc.bat の場所を PATH に追加する必要があります。ojspc.bat のデフォルトのパスは次のとおりです。

  ORACLE_HOME/j2ee/home/jsp/bin/ojspc

次に、ファイルの生成および格納の制御に最も頻繁に使用される ojspc オプションを示します。オプションの完全なリストは、4-7 ページの「ojspc コマンドライン・オプションのサマリー表」を参照してください。

- -appRoot は、JSP ソース・ファイルが格納されているディレクトリと、ojspc が実行されている作業ディレクトリとが異なる場合に、アプリケーション・ルート・ディレクトリを指定します。
- -srcdir は、ソース・ファイルを指定の場所に格納します（バッチ事前変換の場合は関係ありません）。
- -dir は、バイナリ・ファイル（.class ファイルと Java リソース・ファイル）を指定の場所に格納します（バッチ事前変換の場合は関係ありません）。
- -noCompile は、生成されたページ実装クラスのソースをコンパイルしません。

  その結果、.class ファイルは作成されません。
- -extres は、静的なテキストを Java リソース・ファイル（.res）に格納します。

1 つ以上の JSP のプリコンパイル

この項では、JSP をプリコンパイルする使用例を示します。

単一の JSP のプリコンパイル

この例では、指定された JSP を変換およびコンパイルする、基本的な ojspc の使用方法を示します。

```
cd /source
ojspc index.jsp
```

この場合、次のファイルが /source 作業ディレクトリ内に生成されます。

```
_index.class
_index.java
```

複数の JSP のプリコンパイル

次の例では、myapp/mysrcdir/ 内のすべての JSP がコンパイルされます。

```
cd /source
ojspc myapp/mysrcdir/*.jsp
```

デフォルトでは、出力される .java および .class ファイルは、/source 作業ディレクトリ内の _myapp/_mysrcdir という名前の新しいサブディレクトリに生成されます。

異なるソース・ディレクトリ内の JSP のプリコンパイル

次のように、ソース・ファイルが格納されているディレクトリ構造の外にあるディレクトリから ojspc を実行すると仮定します。この場合、ファイルへの絶対パスに加え、-appRoot オプションを使用してアプリケーション・ルートも指定する必要があります。

```
cd /stuff
ojspc -appRoot /source D:/source/myapp/mysrcdir/*.jsp
```

出力されるファイルは /stuff/_source/_myapp/_mysrcdir に生成されます。

異なるディレクトリでの異なるファイル・タイプの生成

次の例では、-dir オプションで指定したディレクトリに Java クラスとファイルを生成し、-srcdir オプションで指定したディレクトリに .java ソース・ファイルを生成します。-extres オプションを使用すると、JSP 内の静的なテキストが _MyPage.res という Java リソース・ファイルに格納されます。これは、ページに大量の静的なコンテンツが含まれている場合に便利です。

```
ojspc -dir myapp/mybindir -extres -srcdir myapp/mysrcdir MyPage.jsp
```

ojspc を使用する際、次の点に注意してください。

- 特定の JSP をコンパイルするには、ファイル・リストの JSP ファイル名を空白で区切ります。ファイルが作業ディレクトリ内に存在しない場合は、各 JSP ごとにパスを指定する必要があります。
- ターゲット・ディレクトリ内のすべての JSP をコンパイルするには、*.jsp を指定します。
- オプション・リスト内のオプション名とオプション値の間のデリミタには空白を使用します。
- オプション名は大 / 小文字を区別しませんが、オプション値（パッケージ名、ディレクトリ名、クラス名、インタフェース名など）は、通常、大 / 小文字を区別します。
- コマンドラインにオプション名を入力して、デフォルトでは無効のブール型のオプション（フラグ）を有効にします。たとえば、-extres true ではなく、-extres と入力します。

WAR ファイル内の JSP のプリコンパイル

ojspc を WAR ファイルや、または 1 つ以上のネストされた WAR ファイルが格納されている EAR に対して実行し、中に格納されている JSP をプリコンパイルできます。ユーティリティにより、WAR 内の WEB-INF\lib パスに、コンパイル済の Java クラスを格納した __oracle_jsp*warFileName*.jar という JAR が作成されます。出力された WAR はそのままデプロイできます。

たとえば、ojspc を sample.ear に対して実行し、その中の sample-web.war には複数の JSP が含まれていると仮定します。出力ファイルを指定していないため、既存の WAR は新しい JAR ファイルで更新されます。

```
ojspc sample.ear
```

更新後の sample-web.war では、WEB-INF\lib パスに次のファイルが含まれます。

```
__oracle_jsp_sample-web.jar
```

使用の際は、次の点にも注意してください。

- デフォルトで、ojspc により、元のアーカイブ・ファイルが新しい JAR で更新されます。元のアーカイブ・ファイルを変更したくない場合、-output オプションを使用すると、新しいアーカイブ・ファイル名を指定できます。
- -batchMask オプションを使用すると、事前変換およびコンパイル用のファイル名拡張子を指定できます。指定した拡張子は、デフォルト（*.jsp および *.java）のかわりに使用されます。
- 出力されるアーカイブ・ファイルに処理されたソース・ファイルを含めない場合は、-deleteSource オプションを使用します。その場合、使用前に、4-7 ページの「ojspc コマンドライン・オプションのサマリー表」に記されているこのオプションの説明を確認してください。
- 変換中に作成されたすべての .java ファイルは更新後の WAR から破棄されます。

Ant タスクを使用した JSP のプリコンパイル

Ant タスクを使用して JSP をプリコンパイルできます。*ORACLE_HOME*/ant/ にある Ant の OC4J 実装を使用する場合は、Ant タスクを次のように設定できます。

```
 <project name="test" default="t1" basedir="." xmlns:oracle="antlib:oracle" >

    <property name="oracle.home" value=ORACLE_HOME/>

    <target name="t1" >
       <oracle:compileJsp file="test/simplest.ear"
                          verbose="true"
                           output="test/out.ear" />
     </target>
    <target name="t2" >
      <oracle:compileJsp verbose="true"
                          appRoot="/scratch/ojspcAntTask/build"
                          dir="/scratch/ojspcAntTask/generatedClass"
                          srcDir="/scratch/ojspcAntTask/generatedJavaSources"

addClasspath="lib/a.jar;lib/b.jar:${oracle.home}/lib/some.jar">
        <fileset dir="/scratch/ojspcAntTask/build">
          <include name="**/*.jsp"/>
          <exclude name="**/*negativeExample.jsp*"/>
        </fileset>
      </oracle:compileJsp>
    </target>
  </project>
```

これで、ant t1 t2 を実行できます。

注意： この章に記載されている OC4J Ant タスクは、Apache Ant バージョン 1.6.5 での使用を意図しています。最新の Apache Ant 製品ドキュメントについては、`http://ant.apache.org/manual/` を参照してください。

Ant の OC4J 実装を使用しない場合は、『Oracle Containers for J2EE デプロイメント・ガイド』の手順に従ってください。

この Ant タスクの属性は、ojspc のコマンドライン・オプションに由来します。詳細は ojspc オプションを確認してください。名前 / 値オプションは、それぞれが Ant タスクの文字列またはファイルまたはパスの属性になります。ブール型オプション（名前のみで構成されるオプション）は、それぞれが Ant タスクのブール型属性になります。

処理するファイルの指定方法は 2 つあります。1 つは、t1 というターゲットが示すように、ファイル属性を使用する方法です。もう 1 つは、t2 というターゲットが示すように、標準の Ant の fileset 要素であるネストされた `<fileset>` 要素を使用する方法です。この方法では、コンパイル対象をより柔軟に指定できます。後者の場合、`appRoot` 属性を、ネストされた `<fileset>` 要素の `dir` 属性と同一にすることを強くお薦めします。

ojspc コマンドライン・オプションのサマリー表

表 4-1 に、ojspc 事前変換ユーティリティでサポートされているコマンドライン・オプションが要約されています。

表 4-1 ojspc 事前変換ユーティリティのオプション

オプション	説明
-addclasspath	生成されたページ実装クラス・ソースのコンパイル時に使用する、javac のための追加クラスパス・エントリを指定します。
-approot	アプリケーション・ルート・ディレクトリを指定します。変換対象のファイルが含まれているディレクトリとは別のディレクトリから ojspc を実行する場合のみ、このオプションを使用します。デフォルトは ojspc の作業ディレクトリです。 指定したアプリケーション・ルート・ディレクトリ・パスは、次の場合に使用されます。 ■ 変換するページの静的な include ディレクティブで使用 指定したディレクトリ・パスは、変換するページの include ディレクティブ内でアプリケーション相対（コンテキスト相対）パスの前に付加されます。 ■ ページ実装クラスのパッケージを決定するときに使用 パッケージのディレクトリは、変換されるファイルの場所に基づいて、アプリケーション・ルート・ディレクトリに対して相対的に決定されます。このパッケージによって、出力ファイルの格納場所も決定します。 次に例を示します。 ■ 次のファイルを変換すると仮定します。 /abc/def/ghi/test.jsp ■ 次のように、カレント・ディレクトリの /abc から ojspc を実行します。 cd /abc ojspc def/ghi/test.jsp ■ test.jsp ページには次の include ディレクティブがあります。 <%@ include file="/test2.jsp" %> ■ 次のように、test2.jsp ページは /abc ディレクトリ内にあります。 /abc/test2.jsp この例では、デフォルトのアプリケーション・ルートの設定がカレント・ディレクトリの /abc ディレクトリであるため、-appRoot の設定は不要です。include ディレクティブでは、アプリケーションに対して相対的な /test2.jsp 構文（「/」で始まります）を使用するため、インクルード・ページの場所は、/abc/test2.jsp になります。 この場合のパッケージは _def._ghi です。このパッケージは、ojspc を実行したときのカレント・ディレクトリに対して相対的な test.jsp の場所に基づいて決定されます。（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。）これに従って、出力ファイルが格納されます。 ただし、他のディレクトリ（/home/mydir など）から ojspc を実行する場合は、次の例に示すように、-appRoot の設定が必要です。 cd /home/mydir ojspc -appRoot /abc abc/def/ghi/test.jsp パッケージは _def._ghi のままですが、ディレクトリは、指定のアプリケーション・ルート・ディレクトリに対して相対的な test.jsp の場所に基づいて決定されます。

表 4-1 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	説明

-batchMask	バッチ事前変換の際、アーカイブ・ファイルで処理されるソース・ファイルを指定できます。デフォルトでは、.jsp および .java ファイルがすべて処理されます。-batchMask オプションによって指定されたファイル・マスクは、これらのデフォルトに追加するのではなく、デフォルトのかわりに使用されます。
	
	ファイル・マスクのリストの前後は引用符で囲み、リスト内ではデリミタにカンマまたはセミコロンを使用します。ファイル・マスクの前後にある空白は無視されます。マスクにディレクトリを含めることができます。このオプションで指定するファイル・マスクでは大 / 小文字を区別しません。
	
	-batchMask 実装では、標準のワイルドカード・パターン・マッチングが完全にサポートされています。
	
	デフォルト設定の場合、次の 2 つの例と同じです。
	
	`ojspc myapp.war`
	
	`ojspc -batchMask "*.jsp,*.java" myapp.war`
	
	この次の例では、.java ファイルの処理が削除されていますが、.jsph ファイルの処理が追加されています。
	
	`ojspc -batchMask "*.jsph,*.jsp" myapp.war`
	
	次の例では、.java ファイルは処理されていません。名前が「abc」で始まり、アーカイブ・ファイルの上位レベルのサブディレクトリにある .jsp ファイルのみが処理されます。
	
	`ojspc -batchMask "*/abc*.jsp" myapp.zip`
	
	次の例は、前述の例と同じですが、アーカイブ・ファイルの上位レベルにあり、名前が「abc」で始まる .jsp ファイルも処理されます。
	
	`ojspc -batchMask "abc*.jsp, */abc*.jsp" myapp.jar`
	
	最後の例では、ファイル a.jspc と、「My」で始まり、mydir/subdir のサブディレクトリであるディレクトリにあり、パターン「t?st」（2 番目の文字が任意の文字、「test」、「tast」、「tust」など）と一致する .jsp ファイルが明確に処理されます。
	
	`ojspc -batchMask "mydir/subdir/t?st/My*.jsp" myapp.ear`

-debug	生成後の Java クラス・ファイル内に SMAP デバッグ・データを生成する場合に使用します。 オプション設定の file により、作業ディレクトリ内の SMAP ファイルにデバッグ情報が生成されます。ファイル名は生成される Java クラスと同じで、.smap 拡張子が付加されます。たとえば、foo.jsp のデバッグ・データは _foo_jsp.smap に出力されます。

表 4-1 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	説明
-deleteSource	バッチ事前変換で、出力されるアーカイブ・ファイルに、処理されたソース・ファイルを含めない場合に使用します。デフォルトでは、.jsp および .java ファイルが対象となります。それ以外の場合は、-batchMask オプションのファイル・マスクと一致するファイルのみが対象となります。生成された .java ファイルも破棄されます。 -output オプションを使用しない場合、元のアーカイブ・ファイルのコンテンツが上書きされ、処理後に処理されたソース・ファイルが削除されます。-output オプションを使用すると、処理されたソース・ファイルは、指定の出力アーカイブ・ファイルにはコピーされません。（元のアーカイブ・ファイルは変更されません。） 使用上の注意 : ■ 名前がデフォルトのファイル拡張子と一致しない（-batchMask オプションを使用しない場合）ファイル、または -batchMask オプションを使用して指定した名前マスクと一致しないファイルは、-deleteSource オプションでは破棄されません。これらのファイルは、必要に応じて、出力されたアーカイブ・ファイルから削除する必要があります。これは特に、静的にインクルードされたソース・ファイルの場合に当てはまります。静的にインクルードされたソース・ファイルは、独自に変換できないため、.jsp 拡張子、またはファイルを独自に変換しようとする他の拡張子を使用できません。 ■ JSP ソース・ファイルをデプロイしない場合に、-deleteSource を使用するには、ターゲットの JSP 実行時環境を、使用可能なソース・ファイルがない場合でも正しく動作するように構成する必要があります。
-dir（または -d）	ojspc が生成されたバイナリ・ファイル（.class ファイルと Java リソース・ファイル）を格納するベース・ディレクトリを指定します。（-extres オプションによって静的なコンテンツに対して作成された .res ファイルは、Java リソース・ファイルです。）ショートカットとして、-d も使用できます。 指定されたパスは、ファイルのシステム・パスとして（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）取得されるため、ディレクトリはすでに存在している必要があります。 指定されたディレクトリ下のサブディレクトリは、パッケージに応じて、必要な場合に自動的に作成されます。 デフォルトでは、カレント・ディレクトリ（ojspc を実行したときのカレント・ディレクトリ）を使用します。 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格納することをお薦めします。これによって、作成されたファイルを簡単に識別できます。 **注意** ■ このオプションは、WAR または EAR ファイルのバッチ事前変換の際は無視されます。 ■ ディレクトリ名に空白を使用できる Windows および UNIX などの環境では、ディレクトリ名を引用符で囲んでください。
-extend	生成されたページ実装クラスが拡張するクラスを指定します。バッチ事前変換には、このオプションを使用しないでください。
-extraImports	OC4J によってインポートされるデフォルトの JSP パッケージよりも多くのインポートを追加します。追加のインポートのパッケージ名または完全修飾されたクラス名を指定します。名前を指定する場合は、次の例のように引用符で囲み、カンマまたはセミコロンで区切る必要があります。 ojspc -extraImports "java.util.*,java.io.*" foo.jsp -extraImports オプションは、Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。

表 4-1 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	説明
-extres	ページの静的なコンテンツを、生成されたページ実装クラスのサービス・メソッドではなく、Java リソース・ファイルに格納します。1 ページに大量の静的なコンテンツが存在する場合は、この技法によって変換速度が向上し、ページの実行速度も向上します。詳細は、6-9 ページの「大量の静的なコンテンツまたはタグ・ライブラリの使用の管理」を参照してください。 リソース・ファイルは、出力された .class ファイルと同じディレクトリに格納されます。 ファイル名は、JSP ページ名に基づいて付けられます。現在の OC4J の JSP 実装では、コア名が JSP 名と同じになりますが（JSP 名に特殊文字が含まれない場合）、接頭辞のアンダースコア（_）と接尾辞の .res が名前に付けられます。たとえば、MyPage.jsp を変換すると、通常の出力以外に、_MyPage.res が作成されます。 ただし、名前の生成に関する正確な実装は、今後のリリースで変更される場合があります。
-forgiveDupDirAttr	単一の JSP 変換単位内で同じディレクティブ属性に重複する設定がある場合に、JSP の変換エラーを回避します。 -forgiveDupDirAttr オプションは、Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。
-help（または -h）	コンソールに ojspc の使用情報が表示されます。
-ignoreErrors	JSP とともにパッケージ化されている XML ディスクリプタ（web.xml または orion-web.xml など）の 1 つによるエラーでないかぎり、エラーが発生した場合も処理を継続するよう ojspc に強制します。
-implement	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を指定します。バッチ事前変換には、このオプションを使用しないでください。
-noCompile	ojspc が生成されたページ実装クラスをコンパイルしないよう指示します。このオプションは、なんらかの理由で、後で代替の Java コンパイラを使用してコンパイルする場合などに使用します。
-oldIncludeFromTop	ネストされた include ディレクティブ内のページの場所をトップレベルのページに対して相対的にします。それ以外の場合は、1 つ上位の親ページに対して相対的になります。これは JSP 仕様に準拠しています。 このオプションは、Oracle9*i*AS リリース 2 より前の OC4J バージョンとの下位互換性を維持するために使用します。 -oldIncludeFromTop オプションは、Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。
-output	バッチ事前変換の場合に、出力アーカイブ・ファイル名を指定します。元のアーカイブ・ファイルのすべてのコンテンツが、指定したアーカイブ・ファイルにコピーされます。次に、事前変換で出力された .class ファイルおよびリソース・ファイルが、指定したファイル内のネストされた JAR ファイルに格納されます（-deleteSource が有効になっている場合、指定したファイルからソース・ファイルが削除されます）。 元のアーカイブ・ファイルは変更されません。デプロイには、元のファイルではなく新しいファイルを使用します。（ネストされた JAR ファイルの詳細は、4-2 ページの「WAR ファイルのバッチ事前変換の概要」を参照。） -output オプションを指定しないと、元のアーカイブ・ファイルが更新され、新しいアーカイブ・ファイルは作成されません。 次に、-output の使用例を示します。 ojspc -output myappout.war myapp.war

表 4-1 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	説明

-packageName	生成されたページ実装クラスのパッケージ名を指定します。指定されていない場合のパッケージ名は、ojspc を実行したときのカレント・ディレクトリに対して相対的な .jsp ファイルの場所に基づいて決定されます。
	
	たとえば、/myapproot ディレクトリから ojspc を実行し、.jsp ファイルが /myapproot/src/jspsrc ディレクトリに存在する場合について考えます（% は UNIX プロンプトを表します）。
	
	`% cd /myapproot`
	`% ojspc -packageName myroot.mypackage src/jspsrc/Foo.jsp`
	
	この結果、パッケージ名として myroot.mypackage が使用されます。
	
	この例で -packageName オプションが使用されなかった場合、JSP トランスレータは（現在の実装内で）デフォルトの _src._jspsrc をパッケージ名として使用します。（実装の詳細は、今後のリリースで変更される場合があります。）

-reduceTagCode	カスタム・タグを使用するために生成されたコードのサイズをさらに縮小します。デフォルトは false です。

-reqTimeIntrospection	コンパイル時にイントロスペクションが実行できなかった場合、リクエスト時に JavaBean のイントロスペクションを実行できます。ただし、有効なコンパイル時のイントロスペクションが正常終了した場合、このフラグの設定に関係なく、リクエスト時のイントロスペクションは実行されません。デフォルトは false です。
	
	リクエスト時のイントロスペクションの使用例として、タグ・ハンドラによって、タグ補足情報クラスの VariableInfo インスタンスにある一般的な java.lang.Object インスタンスが変換時とコンパイル時に戻される一方で、実際には、特定のオブジェクトがリクエスト時（実行時）に生成される場合を想定します。この場合、-reqTimeIntrospection が有効になっていると、Web コンテナはリクエスト時までイントロスペクションを遅延します。
	
	このフラグには、if..then..else ループの別の分岐などで、Bean を 2 回宣言できるという効果もあります。次の例を考えてみます。-reqTimeIntrospection が有効になっていないと、このコードにより、解析例外が発生します。有効になっている場合、コードはエラーなしで機能します。
	
	`<% if (cond) { %>`
	`  <jsp:useBean id="foo" class="pkgA.Foo" />`
	`<% } else { %>`
	`  <jsp:useBean id="foo" class="pkgA.Foo2" />`
	`<% } %>`

-setpropertyOnErrContinue	property="*" の場合、jsp:setProperty の使用時にエラーが発生したとき、リクエスト・パラメータの反復と対応する Bean プロパティの設定が継続されます。 -setpropertyOnErrContinue オプションは、Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。
-srcdir	生成されたソース・ファイルを未使用のディレクトリに格納します。これによって、作成されたファイルを簡単に識別できます。バッチ事前変換には、このオプションを使用しないでください。 ojspc が、生成された（.java）ソース・ファイルを格納する場所を指定します。指定するディレクトリはすでに存在している必要があります。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、ファイルのシステム・パスとして取得されます。 指定されたディレクトリ下のサブディレクトリは、パッケージに応じて、必要な場合に自動的に作成されます。 デフォルトでは、カレント・ディレクトリ（ojspc を実行したときのカレント・ディレクトリ）を使用します。

表 4-1 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	説明
-staticTextInChars	JSP トランスレータが、JSP ページの静的なテキストをバイトではなく文字として生成します。デフォルトの false では、静的なテキスト・ブロックの出力で、パフォーマンスが改善されます。 次の例に示すように、実行時に文字エンコードをアプリケーションで動的に変更する必要がある場合は、このフラグを有効にします。 <% response.setContentType("text/html; charset=UTF-8"); %>
-tagReuse	このオプションを使用して、タグ・ハンドラの再利用（タグ・ハンドラ・インスタンスのプーリング）のモードを指定します。有効な値は次のとおりです。 ■ runtime: タグ・ハンドラ再利用の実行時モデルを有効にします。この設定は、特定の JSP ページで、JSP ページ・コンテキスト属性 oracle.jsp.tags.reuse を false の値に設定するとオーバーライドできます。現在のリリースでは、このオプションは推奨していません。 ■ none: タグ・ハンドラの再利用を無効にします。この設定は、特定の JSP ページで、JSP ページ・コンテキスト属性 oracle.jsp.tags.reuse を true の値に設定するとオーバーライドできます。 ■ compiletime（デフォルト）: タグ・ハンドラ再利用のコンパイル時モデルをその基本モードで有効にします。 ■ compiletime_with_release: タグ・ハンドラ再利用のコンパイル時モデルを、その「解放」モードで有効にします。この場合、タグ・ハンドラの release() メソッドは、指定したページの指定したタグ・ハンドラの使用の合間にコールされます。
-validateXML	web.xml ファイルの XML 妥当性チェックをリクエストします。デフォルトでは、web.xml の妥当性チェックは実行されません。 このオプションは、TLD の XML 妥当性チェックを無効にします。デフォルトでは、TLD の妥当性チェックが実行されます。
-verbose	ojspc が実行時の状態情報を出力します。 次の例は、myerror.jsp の変換に対する -verbose の出力を示します。（この例では、ojspc は myerror.jsp が存在するディレクトリから実行されます。） > ojspc -verbose myerror.jsp Translating file: myerror.jsp JSP files translated successfully. Compiling Java file: ./_myerror.java
-version	JSP のバージョン番号を表示します。

5

OC4J における JSP の変換

この章では、JSP ページを Java サーブレット・コードに変換する、OC4J 内部の JSP トランスレータの操作について説明します。次の各項では、JSP トランスレータの全般的な機能について、Oracle Application Server でのオンデマンド変換の動作を中心に説明します。

- 生成されるコードの機能
- 出力名に関する一般規則
- 生成されるパッケージとクラスの名前
- 生成されるファイルとその格納場所
- Oracle JSP のグローバル・インクルード

重要： この項で説明するパッケージとクラスのネーミング、ファイルとディレクトリのネーミング、出力ファイルの格納場所、および生成されるコードに関する実装の詳細は、説明を目的としています。正確な詳細は、リリースごとに変更される場合があります。

生成されるコードの機能

OC4J の JSP トランスレータは、JSP ページ実装クラスの標準 Java コードを生成します。このクラスは、基本的に、JSP 機能が追加されたサーブレット・クラスです。

この項では、JSP トランスレータによって JSP ソース（通常は `.jsp` または `.jspx` ファイル）から生成され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

JSP トランスレータは、ページ実装クラスのサーブレット・コードを生成するとき、標準のプログラミング・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と事前変換モデルの両方について、生成されたコードには、自動的に次の機能が含まれます。

- `javax.servlet.jsp.HttpJspPage` インタフェースを実装した Web コンテナから提供されるラッパー・クラスを拡張します。これによって、より汎用的な `javax.servlet.jsp.JspPage` インタフェースが拡張され、さらに `javax.servlet.Servlet` インタフェースが拡張されます。
- `HttpJspPage` インタフェースで指定される `_jspService()` メソッドを実装します。このメソッドは、サービス・メソッドと呼ばれ、ページ実装クラスの中心的なメソッドです。Java スクリプトレットからのコード、式、および JSP ページ内の JSP タグは、このメソッド実装に組み込まれます。
- JSP ソース・コードの `page` ディレクティブで `session="false"` を特に設定しないかぎり、HTTP セッションをリクエストするコードが含まれます。

静的なテキスト用のメンバー変数

ページ実装クラスのサービス・メソッドの `_jspService()` には、JSP ページの静的なテキストを出力するための出力文（`out.print()` または暗黙的な `out` オブジェクトにある同等のコール）が含まれます。JSP トランスレータは、静的なテキスト自体をページ実装クラス内の一連のメンバー変数に格納します。サービス・メソッドの `out.print()` 文は、テキストを出力するために、これらのメンバー変数の属性を参照します。

注意：

- OC4J の JSP トランスレータは、静的なテキストを Java リソース・ファイルに必要に応じて格納できます。これは、大量の静的なテキストを含むページの場合にメリットがあります。6-9 ページの「大量の静的なコンテンツまたはタグ・ライブラリの使用の管理」を参照してください。この機能は、オンデマンド変換の場合は JSP `external_resource` 構成パラメータ、事前変換の場合は `ojspc -extres` フラグを介してリクエストできます。
- `external_resource` パラメータは、Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。

出力名に関する一般規則

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ネーミングを行います。ただし、この一連の規則と他の実装の詳細は、リリースによって変更される場合があります。

しかし、JSP ページのベース名に特殊文字が含まれていなければ、ベース名がそのまま出力クラス名とファイル名に含まれるという点は、変更されません。たとえば、`MyPage23.jsp` を変換すると、文字列「MyPage23」は常に、ページ実装クラス名、Java ソース・ファイル名およびクラス・ファイル名の一部になります。

ベース名の前にはアンダースコア（_）が付きます。`MyPage23.jsp` を変換すると、ページ実装クラスの `_MyPage23` がソース・ファイルの `_MyPage23.java` に作成され、`_MyPage23.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ている場合は、そのパスの各コンポーネントの前にアンダースコアが付きます。たとえば、`/jspdir/myapp/MyPage23.jsp` を変換すると、`_MyPage23` クラスは次のパッケージ内に含まれます。

```
_jspdir._myapp
```

パッケージ名は、出力の `.java` ファイルと `.class` ファイルのディレクトリの作成で使用されるため、出力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、`webapp/test` というディレクトリにある JSP ページを変換すると、JSP トランスレータは、デフォルトで、ページ実装クラス・ソースに対して `webappdeployment/_pages/_test` というディレクトリを作成します。5-5 ページの「生成されるファイルとその格納場所」で説明するように、すべての出力ディレクトリは、標準の `_pages` ディレクトリの下に作成されます。

JSP ページ名またはパス名に特殊文字を使用している場合、JSP トランスレータは、出力クラス名、パッケージ名およびファイル名に不正な Java 特殊文字が含まれていないことを確認します。

たとえば、`My-name_foo2.jsp` を変換すると、`_My_2d_name__foo2` という名前のクラスがソース・ファイルの `_My_2d_name__foo2.java` に作成されます。ハイフンは、英数字の文字列に変換されます。（「`foo2`」の前には追加のアンダースコアも挿入されます。）

この場合、JSP ページ名の英数字部分のみが、出力のクラス名とファイル名にそのまま含まれます。この例では、「`My`」、「`name`」または「`foo2`」が含まれます。

`.jspx` ファイルの場合に生成されるソース・ファイルとクラス・ファイルの名前も同様ですが、名前には _jspx が追加されます。たとえば、`MyPage.jsp` を変換すると、ソース・ファイルが `_MyPage_jspx.java` となり、これが `_MyPage_jspx.class` にコンパイルされます。

これらの規則の例は、この章で後述します。

生成されるパッケージとクラスの名前

JSP 仕様には、JSP テキストを解析して変換するための統一プロセスが定義されていますが、生成されるクラスの命名方法は説明されていません。クラスの命名方法は、各 JSP 実装によって異なります。

この項では、変換時にコードを生成するとき、OC4J の JSP トランスレータが行うパッケージ名とクラス名の作成方法について説明します。

注意： OC4J の JSP トランスレータが出力クラス、パッケージおよびファイルのネーミングで使用する一般規則については、5-3 ページの「出力名に関する一般規則」を参照してください。

パッケージのネーミング

オンデマンド変換の場合は、ユーザーが JSP ページのリクエスト時に指定する URL パス（具体的には、ドキュメント・ルートまたはアプリケーション・ルートに対して相対的なパス）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します。URL 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 内の文字の大小に関係なく、常に小文字になります。

次の URL の例を考えてみます。

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

この結果、現行の OC4J の JSP 実装における、生成されたコードのパッケージ仕様は次のとおりです。

```
package _hr._expenses;
```

（実装の詳細は、今後のリリースで変更される場合があります。）

JSP ページがアプリケーション・ルート・ディレクトリにある場合、パッケージ名は生成されません。この場合の URL は、次のとおりです。

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

クラスのネーミング

`.jsp` ファイルのベース名によって、生成されるコード内のクラス名が決定します。

次の URL の例を考えてみます。

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

現行の OC4J の JSP 実装における、生成されたコードのクラス名は次のとおりです。

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

（実装の詳細は、今後のリリースで変更される場合があります。）

ユーザーが URL に指定する文字は、実際の `.jsp` ファイル名と大 / 小文字が一致する必要があります。たとえば、実際のファイル名が `UserLogin.jsp` または `userlogin.jsp` である場合、エンド・ユーザーは、それぞれの名前をそのまま指定できますが、実際のファイル名が `UserLogin.jsp` の場合、`userlogin.jsp` は指定できません。

現在のトランスレータは、ファイル名の大 / 小文字に従ってクラス名の大 / 小文字を区別しています。次に例を示します。

- ファイル名 `UserLogin.jsp` は、クラスでは `_UserLogin` になります。
- ファイル名 `Userlogin.jsp` は、クラスでは `_Userlogin` になります。
- ファイル名 `userlogin.jsp` は、クラスでは `_userlogin` になります。

クラス名の大 / 小文字を区別する場合は、それに従って .jsp ファイル名を付ける必要があります。ただし、ページ実装クラスはエンド・ユーザーに表示されないため、通常は問題になりません。

生成されるファイルとその格納場所

この項では、オンデマンド変換の場合に JSP トランスレータで生成されるファイルとその格納場所について説明します。（プリコンパイルの場合は、ojspc によるファイルの格納方法が異なり、固有の関連オプションがあります。詳細は、第 4 章「ojspc による JSP ページのプリコンパイル」を参照してください。）

注意： 出力クラス、パッケージおよびファイルのネーミングで使用される一般規則については、5-3 ページの「出力名に関する一般規則」を参照してください。

JSP トランスレータで生成されるファイル

ファイル名の例では、変換されるファイルの名前として、Foo.jsp を使用します。

ソース・ファイル

- JSP トランスレータにより、.java ファイル（_Foo.java など）がページ実装クラスに対して作成されます。

バイナリ・ファイル

- .class ファイルは、Java コンパイラによって、ページ実装クラスに対して作成されます。デフォルトでの Java コンパイラは JDK javac ですが、JSP 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。

  javaccmd パラメータは、Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。

- .res Java リソース・ファイル（_Foo.res など）は、external_resource JSP 構成パラメータが有効な場合、静的なページのコンテンツに対して必要に応じて作成されます。

注意：

- ページ実装クラスに対して生成されるファイルの正確な名前は、今後のリリースで変更される場合がありますが、一般的なフォームは同じです。名前には、常にベース名（この例の場合は「Foo」）が含まれますが、詳細は変更される場合があります。
- external_resource parameter パラメータは、Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。

JSP トランスレータの出力ファイルの格納場所

JSP トランスレータは、生成された出力ファイルを _pages ディレクトリに格納します。このディレクトリは、JSP キャッシュ・ディレクトリの下に作成されます。JSP キャッシュ・ディレクトリは、global-web-application.xml ファイルまたはアプリケーションの orion-web.xml ファイルの <orion-web-app> 要素の jsp-cache-directory 属性に指定されます。jsp-cache-directory のデフォルトの ./persistence 値を想定している場合は、一般的に、次の場所が基本になります。

ORACLE_HOME/j2ee/home/app-deployment/app-name/web-app-name/persistence/_pages/...

スタンドアロン OC4J では、次の場所が OC4J がインストールされた場所に対して相対的になります。

j2ee/home/app-deployment/app-name/web-app-name/persistence/pages/...

次の点に注意してください。

- *app-deployment* ディレクトリは OC4J のデプロイ・ディレクトリで、OC4J の server.xml ファイルに指定されています。（スタンドアロン OC4J では、一般的には、application-deployments ディレクトリです。）
- *app-name* はアプリケーション名で、server.xml の <application> 要素に従います。
- *web-app-name* は対応する Web アプリケーション名で、OC4J Web サイトの XML ファイル（通常、Oracle Application Server では default-web-site.xml ファイル、スタンドアロン OC4J では http-web-site.xml ファイル）内の <web-app> 要素のアプリケーション名にマッピングされます。

_pages ディレクトリ下のパスは、アプリケーション・ルート・ディレクトリ下の .jsp ファイルのパスによって決まります。

たとえば、スタンドアロン OC4J で、スタンドアロン OC4J のデフォルトの Web アプリケーション・ディレクトリの下の examples/jsp サブディレクトリにある welcome.jsp ページについて考えてみます。このページへのパスは次のようになり、OC4J がインストールされた場所に対して相対的です。

j2ee/home/default-web-app/examples/jsp/welcome.jsp

デフォルトのアプリケーション・デプロイ・ディレクトリの場合、JSP トランスレータは、出力ファイル（_welcome.java および _welcome.class）を次のディレクトリに格納します。

j2ee/home/application-deployments/default/defaultWebApp/persistence/_pages/_examples/_jsp

.jsp ソース・ファイルは、アプリケーション・ルート・ディレクトリ下の examples/jsp サブディレクトリに格納されているため、JSP トランスレータは、パッケージ名として _examples._jsp を生成し、出力ファイルを _pages ディレクトリ下の _examples/_jsp サブディレクトリに格納します。

重要： 生成された出力ファイルの格納場所と出力ファイル名での「_」の使用など、実装の詳細は、今後のリリースで変更される場合があります。

Oracle JSP のグローバル・インクルード

OC4J の Web コンテナでは、グローバル・インクルードと呼ばれる機能が提供されています。この機能によって、仮想の JSP include ディレクティブを使用して、指定のディレクトリ内またはディレクトリ下の JSP ページに静的にインクルードする 1 つ以上のファイルを指定できます。変換時に、Web コンテナは、インクルード・ファイルとディレクトリをページに指定する構成ファイル /WEB-INF/ojsp-global-include.xml を検索します。

この機能は、以前の Oracle JSP リリースで globals.jsa または translate_params 機能を使用していたアプリケーションを移行する場合に、特に便利です。

Oracle のグローバル・インクルード機能は、JSP 2.0 仕様以前のものです。この機能は仕様に盛り込まれたため、移植性を考慮して、新規開発では JSP 仕様のメカニズムを使用することを強くお薦めします。

将来のリリースでは Oracle のグローバル・インクルード機能が推奨されなくなる可能性があります。

グローバルにインクルードされたファイルは、次の場合などに使用できます。

- グローバルな Bean 宣言（以前は、globals.jsa でサポートされていました。）
- 共通のページ・ヘッダーまたはフッター
- translate_params と同等の機能を持つコード

グローバル・インクルードのファイルと例

この項では、ojsp-global-include.xml ファイルの概要およびいくつかの例を示します。

ojsp-global-include.xml ファイル

ojsp-global-include.xml ファイルは、インクルードするファイルの名前、ファイルのインクルード先（JSP ページの最上部または最下部）、およびグローバル・インクルードを適用する JSP ページの場所を指定します。この項では、ojsp-global-include.xml の要素について説明します。

<ojsp-global-include>

ojsp-global-include.xml ファイルのルート要素です。属性はありません。

<ojsp-global-include> のサブ要素は次のとおりです。

<include>

<include ... >

<ojsp-global-include> の <include> サブ要素を使用して、インクルードするファイル、およびインクルード先が JSP ページの最上部かまたは最下部かを指定します。

<include> のサブ要素は次のとおりです。

<into>

<include> の属性は次のとおりです。

- file: /header.html や /WEB-INF/globalbeandeclarations.jsph などのインクルードするファイルを指定します。ファイル名の設定は、必ずスラッシュ（/）で始まります。つまり、ファイル名はページ相対ではなく、アプリケーション相対であることが必要です。
- position: ファイルのインクルード先が JSP ページの最上部かまたは最下部かを指定します。サポートされている値は、top（デフォルト）と bottom です。

<into ... >

<include> のこのサブ要素を使用して、指定ファイルのインクルード先 JSP ページの場所（ディレクトリ、場合によってはサブディレクトリ）を指定します。この要素にサブ要素はありません。

<into> の属性は次のとおりです。

- directory: ディレクトリを指定します。このディレクトリ内（必要に応じてそのサブディレクトリ内）にある JSP ページは、<include> 要素の file 属性に指定されているファイルを、静的にインクルードします。directory の設定値は、/dir のように、スラッシュ（/）で始まる必要があります。また、/dir/ のように、設定値のディレクトリ名の後にスラッシュを付けることができます。付けない場合は、変換時に内部で追加されます。
- subdir: directory のすべてのサブディレクトリ内にある JSP ページに、ファイルの静的なインクルードが必要かどうかを指定します。サポートされている値は、true（デフォルト）と false です。

グローバル・インクルードの例

この項では、グローバル・インクルードの例を示します。

例 : ヘッダー / フッター 次の ojsp-global-include.xml ファイルの例を考えてみます。

```
<?xml version="1.0" standalone='no'?>
<!DOCTYPE ojsp-global-include SYSTEM 'ojsp-global-include.dtd'>

<ojsp-global-include>
  <include file="/header.html">
     <into directory="/dir" />
  </include>
  <include file="/footer.html" position="bottom">
     <into directory="/dir" subdir="false" />
     <into directory="/dir/part/" subdir="false" />
  </include>
  <include file="/footer2.html" position="bottom">
     <into directory="/dir/part2/" subdir="false" />
  </include>
</ojsp-global-include>
```

この例には、次の 3 つの目的があります。

- header.html ファイルは、dir ディレクトリ内またはその下にある JSP ページの最上部にインクルードされます。結果は、このディレクトリ内またはその下にある各 .jsp ファイルのページの最上部に次の include ディレクティブがある場合と同じになります。

  ```
  <%@ include file="/header.html" %>
  ```

- footer.html ファイルは、dir ディレクトリまたはその part サブディレクトリにある JSP ページの最下部にインクルードされます。結果は、これらのディレクトリ内にある各 .jsp ファイルのページの最下部に次の include ディレクティブがある場合と同じになります。

  ```
  <%@ include file="/footer.html" %>
  ```

- footer2.html ファイルは、dir ディレクトリの part2 サブディレクトリにある JSP ページの最下部にインクルードされます。結果は、このディレクトリ内にある各 .jsp ファイルのページの最下部に次の include ディレクティブが含まれている場合と同じになります。

  ```
  <%@ include file="/footer2.html" %>
  ```

注意 : 複数のヘッダー・ファイルまたはフッター・ファイルを単一の JSP ページにインクルードする場合の順序は、ojsp-global-include.xml ファイルの <include> 要素の順序に従います。

例 : translate_params と同等のコード 次の ojsp-global-include.xml ファイルの例を考えてみます。

```
<?xml version=".0" standalone='yes'?>
<!DOCTYPE ojsp-global-include SYSTEM 'ojsp-global-include.dtd'>

<ojsp-global-include>
  <include file="/WEB-INF/nls/params.jsf">
     <into directory="/" />
  </include>
</ojsp-global-include>
```

params.jsf に、次のコードが含まれていると仮定します。

```
<% request.setCharacterEncoding(response.getCharacterEncoding()); %>
```

params.jsf ファイル（基本的に、setCharacterEncoding() メソッド・コール）は、アプリケーション・ルート・ディレクトリ内またはその下にある JSP ページの最上部にインクルードされます。つまり、このファイルは、アプリケーション内の JSP ページにインクルードされます。結果は、このディレクトリ内またはその下にある各 .jsp ファイルのページの最上部に次の include ディレクティブがある場合と同じになります。

```
<%@ include file="/WEB-INF/nls/parms.jsf" %>
```

6

JSP での作業

この章では、JSP ページのプログラミングに関する基本的な考慮事項について、例を示して説明します。JSP とサーブレット間の相互作用およびデータベース・アクセスについても説明します。

次の項目について説明します。

開始前の考慮点

次の各項では、OC4J 環境で JSP ページのコーディングまたは使用を開始する前に考慮しておく必要がある事項について説明します。

アプリケーション・ルート機能について

サーブレット仕様（サーブレット 2.2 以降）に従って、各 Web アプリケーションには独自のサーブレット・コンテキストがあります。各サーブレット・コンテキストは、サーバーのファイル・システム内でディレクトリ・パスに関連付けられています。このディレクトリ・パスは、Web アプリケーションのモジュール用のベース・パスです。このベース・パスがアプリケーション・ルートです。

各 Web アプリケーションには、独自のアプリケーション・ルートがあります。標準のサーブレット環境の Web アプリケーションの場合、サーブレット、JSP ページ、および HTML ファイルなどの静的なファイルは、すべてこのアプリケーション・ルートに基づいています。（これに対して、サーブレット 2.0 環境では、サーブレットと JSP ページのアプリケーション・ルートと、静的なファイルのドキュメント・ルートは異なります。）

サーブレット URL には、次の汎用的なフォームがあります。

```
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 のコンテキスト・パス部分との間にマッピングが指定されます。サーブレット・パスは、アプリケーションの `web.xml` ファイルに定義されます。`web.xml` 内の `<servlet>` 要素によって、サーブレット・クラスがサーブレット名に関連付けられます。`web.xml` 内の `<servlet-mapping>` 要素によって、URL パターンが指定のサーブレットに関連付けられます。サーブレットが実行されると、サーブレット・コンテナは、指定された URL パターンを既知のサーブレット・パスと比較し、一致したサーブレット・パスを選択します。詳細は、『Oracle Containers for J2EE サーブレット開発者ガイド』を参照してください。

たとえば、アプリケーション・ルートが `/home/dir/mybankapp/mybankwebapp` のアプリケーションが、コンテキスト・パス `/mybank` にマッピングされると仮定します。さらに、このアプリケーションには、サーブレット・パスが `loginservlet` のサーブレットが含まれると仮定します。このサーブレットは、次のように起動できます。

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

アプリケーション・ルートのディレクトリ名はユーザーには表示されません。

このアプリケーションの HTML ページについてこの例を使用すると、次の URL によって、`/home/dir/mybankapp/mybankwebapp/dir/abc.html` ファイルがポイントされます。

```
http://host:port/mybank/dir/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・コンテキストもあります。このコンテキストのコンテキスト・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・ルートにマッピングされている「/」のみです。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/dir/defaultapp/defaultwebapp` で、サーブレット・パスが `myservlet` のサーブレットでデフォルトのコンテキストを使用すると仮定します。その場合の URL は、次のとおりです。

```
http://host:port/myservlet
```

デフォルトのコンテキストは、URL に指定されているコンテキスト・パスと一致しない場合にも使用されます。

HTML ファイルについてこの例を使用すると、次の URL によって `/home/dir/defaultapp/defaultwebapp/dir2/def.html` ファイルがポイントされます。

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

OC4J のクラスパス機能について

OC4J の Web コンテナは、Web サーバー上の標準の場所を使用して、変換済 JSP ページおよび必須クラス（JavaBeans など）用の .class ファイルと .jar ファイルを検索します。コンテナは、Web サーバーのクラスパス構成を使用せずに標準の場所でファイルを検索します。

従属クラスの場所は次のとおりです。これらの場所はアプリケーション・ルートと相対的な関係があります。

```
/WEB-INF/classes/...
/WEB-INF/lib
```

JSP ページ実装クラス（変換済ページ）の場所は、次のとおりです。

```
.../_pages/...
```

/WEB-INF/classes ディレクトリは、Java の各 .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 ディレクトリ下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、oracle.jsp.sample.lottery パッケージに含まれるように定義した、LottoBean と呼ばれる JavaBean があると仮定します。Web コンテナは、アプリケーション・ルートに対して相対的な次の場所で、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（.jar）ファイル用です。Java パッケージ構造は 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、サブディレクトリ内ではなく、lib ディレクトリ内に直接格納されます。たとえば、LottoBean.class は、アプリケーション・ルートに対して相対的な次の場所にある lottery.jar に格納されます。

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

_pages ディレクトリは、OC4J の J2EE ホーム・ディレクトリの下にあり、jsp-cache-directory 構成パラメータの値によって決まります。詳細は、5-5 ページの「生成されるファイルとその格納場所」を参照してください。

重要： _pages ディレクトリのデフォルトの場所など、実装に関する詳細は、今後のリリースで変更される場合があります。

OC4J でデフォルトでインポートされるパッケージ

OC4J の Web コンテナは、JSP 仕様に基づいて次のパッケージを JSP ページにデフォルトでインポートします。JSP でこれらのパッケージを使用する場合、page ディレクティブの import 設定は不要です。

```
javax.servlet.*
javax.servlet.http.*
javax.servlet.jsp.*
```

以前のリリースでは、次のパッケージもデフォルトでインポートされました。

```
java.io.*
java.util.*
java.lang.reflect.*
java.beans.*
```

使用する未修飾のクラス名と、インポートしたパッケージ内の同じ名前のクラスとの間で発生する競合を最小化するために、インポートするデフォルトのパッケージ・リストが縮小されました。

ただし、このために、以前のバージョンの OC4J で使用していたアプリケーションで移行上の問題が発生する可能性があります。このようなアプリケーションは、正常にコンパイルされない場合があります。デフォルト・リストのパッケージより多くのパッケージをインポートする必要がある場合は、次の 2 つの方法があります。

- 追加のパッケージ名または完全修飾したクラス名を、1 つ以上の page ディレクティブの import 設定に指定します。詳細は、1-6 ページの「ディレクティブ」の項の page ディレクティブについての説明を参照してください。

  複数ページの場合、デフォルト・リストより多いパッケージのインポートは、グローバル・インクルード機能を使用して実行できます。5-7 ページの「Oracle JSP のグローバル・インクルード」を参照してください。

- 追加のパッケージ名または完全修飾されたクラス名は、JSP の extra_imports 構成パラメータまたは事前変換用の ojspc -extraImports オプションを使用して指定します。構文は、OC4J 構成パラメータ設定と ojspc オプション設定でそれぞれ異なります。必要に応じて次の項を参照してください。
  - 3-2 ページの「JSP 構成パラメータの概要」
  - 4-7 ページの「ojspc コマンドライン・オプションのサマリー表」

注意： extra_imports パラメータおよび -extraImports オプションは、Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。

JDK 1.4 に関する問題 : パッケージに含まれないクラスを起動できない

OC4J には、Sun 社の JDK1.4 および JDK 1.5 が同梱されています。したがって、以前のバージョンの JDK から移行する場合、次の点を考慮する必要があります。

Sun 社が述べているように、コンパイラでは、不特定の名前空間から型をインポートするインポート文は拒否されます。これは、JDK の以前のバージョンに関するセキュリティ上の問題とあいまい性に対処するための措置でした。基本的に、これはパッケージに含まれていないクラス（クラスのメソッド）を起動できないことを示します。パッケージに含まれていないクラスを起動しようとすると、コンパイル時に致命的エラーが発生します。

これは特に、JSP ページから JavaBeans を起動する JSP 開発者に影響します。このような Bean は、パッケージの外部にあることが多いためです（JSP 仕様 2.0 では、新しいコンパイラの要件を満たすために、Bean はパッケージ内に存在することが必要です）。

注意：

- javac -source コンパイラ・オプションは、JDK1.4 コンパイラによって JDK1.3 コードが透過的に処理されることを目的としていますが、このオプションは、パッケージに含まれないクラスの問題には対処していません。
- OC4J では、JDK1.4 と JDK1.5 コンパイラのみがサポートおよび認証されています。server.xml ファイルに <java-compiler> 要素を追加することで、別のコンパイラを指定できます。これがパッケージに含まれないクラスの問題の対処方法になることもありますが、OC4J とともに使用する場合、Oracle では他のコンパイラは認証またはサポートされていません。（また、Oracle Application Server 環境で server.xml ファイルを直接更新しないでください。Oracle Enterprise Manager 10*g* を使用してください。）

JDK1.4 の互換性の問題の詳細は、次の Web サイトを参照してください。

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

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

一般的な JSP プログラミングの方針

この項では、特定のターゲット環境に関係なく、JSP ページのプログラミング時に考慮する必要がある事項について説明します。次の項目について説明します。

- 従来型 JSP とスクリプトレス JSP の作成の比較
- JavaBeans とスクリプトレットの使用の比較
- 静的なインクルードと動的なインクルードの使用の比較
- JSP アプリケーションの監視
- 大量の静的なコンテンツまたはタグ・ライブラリの使用の管理
- メソッド変数宣言とメンバー変数宣言の使用の比較
- page ディレクティブの使用
- 生成したメソッドの 64K のサイズ制限に対する対処方法
- JSP ファイルのネーミング規則の順守
- JSP での空白の保持とバイナリ・データの使用

注意： この項で説明する内容以外に、JSP 変換に関連する事項と動作についても理解が必要です。第 5 章「OC4J における JSP の変換」を参照してください。

従来型 JSP とスクリプトレス JSP の作成の比較

JSP 開発の主眼は、スクリプトレスな JSP、つまりスクリプトレットや実行時の式などの埋込み Java スクリプト要素を含まないページの作成となっています。スクリプトレスな JSP は、従来のスクリプト・ベースの JSP 開発と比べ、いくつかの利点があります。

- JSP ページからの JSP スクリプトレットや式の排除。
- JSP 作成者と Java 開発者の間の作業の分離。JSP の主要な理念は、JSP ページを構成する HTML および JSP マークアップの作成を担当する JSP ページの作成者と、処理のロジックを提供する Java コンポーネントの実装を担当する Java 開発者との間で、作業と技術を分離することです。
- クリーンで可読性の高い JSP。
- メンテナンスの容易さ。

1-5 ページの JSP コードのサンプルは、スクリプトレスな JSP の例です。

JSP 1.1 以来、ページの作成者は、JavaBeans およびタグ・ハンドラ・インスタンスによって提供される Java 機能にアクセスするために、標準アクション・タグおよびカスタム・タグを活用することにより、ほとんど Java を使用しないページを作成することが可能でした。ただし、完全にスクリプトレスなページを作成するには多くの課題がありました。たとえば、データ・アクセスの制限や、カスタム・タグ・ハンドラ・クラスを作成する際の複雑さなどです。

JSP 2.0 リリースでは、いくつかの主要な改善点により、スクリプトレスなページの作成が大幅に容易になっています。たとえば、式言語（EL）機能を JSP 仕様に完全に統合したことにより、EL からすべての JSP ページのコンテキスト・オブジェクト、変数およびリクエスト・パラメータ、さらに JavaBeans のプロパティとコレクション要素にアクセスできるようになりました。EL を使用すると、Java スクリプトレットや式を使用せずに、アプリケーション・データにアクセスし、データを操作できます。式言語の詳細は、1-18 ページの「式言語の使用による JSP 作成の単純化」を参照してください。

また、カスタム・タグを作成し、JSP ページで使用する方法も容易になりました。JavaServer Pages 標準タグ・ライブラリ（JSTL）では、JSP 作成者が最も必要とする機能の大部分がカプセル化されているタグ・ライブラリを多数提供しています。新しい SimpleTag インタフェースにより、カスタム・タグ・ハンドラの作成が著しく単純化されました。実際、JSP 作成者は、完

全に JSP 構文で作成されているタグ・ファイルを使用して、まったく Java を使用しないタグ・ライブラリを作成できるようになりました。

JSP 2.0 は JSP 1.x との下位互換性を維持しています。つまり、JSP 2.0 構文で作成されたページでも、Java スクリプト要素を使用できます。

JavaBeans とスクリプトレットの使用の比較

JavaServer Pages テクノロジの主なメリットは、ビジネス・ロジックを含み、動的なコンテンツを決定する Java コードと、リクエスト処理、プレゼンテーション・ロジックおよび静的なコンテンツを含む HTML コードとを分離できる点です。この分離によって、HTML のエキスパートはプレゼンテーションに集中でき、Java のエキスパートは、JSP ページからコールされる JavaBeans のビジネス・ロジックに集中できます。

標準の JSP ページに含まれるのは、通常、リクエスト処理やプレゼンテーション用の Java 機能に関する簡単な Java コードのみです。たとえば、このサンプルの `runQuery()` メソッドでのデータ・アクセスは、JavaBean で実行する方が適切です。ただし、出力をフォーマットする `formatResult()` メソッドは、JSP ページの方が適切です。

静的なインクルードと動的なインクルードの使用の比較

JSP ページ内に JSP ページをインクルードする方法は、2 つあります。

`include` ディレクティブ（1-6 ページの「ディレクティブ」を参照）は、変換時にインクルード・ページのコピーを作成し、それを JSP ページ（インクルード先ページ）にコピーします。この機能は、静的なインクルード（または変換時インクルード）と呼ばれ、次の構文を使用します。

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

`<jsp:include>` タグ（1-12 ページの「標準の JSP アクション・タグ」を参照）は、実行時に、インクルード・ページの出力を、インクルード先ページの出力に動的に挿入します。この機能は、動的なインクルード（または実行時インクルード）と呼ばれ、次の構文を使用します。

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

C 構文の知識がある方にとって、静的なインクルードは `#include` 文に相当します。動的なインクルードは、ファンクション・コールと同じです。いずれのインクルードも便利ですが、異なる目的で使用されます。

注意： 静的なインクルードと動的なインクルードは、同じサーブレット・コンテキスト内のページ間でのみ使用できます。

静的なインクルードのロジック手法

静的なインクルードの場合は、インクルード先の JSP ページで生成されるコードのサイズが大きくなります。これは、変換時に `include` ディレクティブの時点で、インクルード・ページのテキストが、インクルード先ページに物理的にコピーされるためです。1 つのページが複数回、インクルード先ページにインクルードされると、複数のコピーが作成されます。

静的にインクルードされた JSP ページは、独立した変換可能なエンティティである必要はありません。このページは、インクルード先ページにコピーされるテキストのみで構成されます。インクルード・テキストがコピーされたインクルード先ページは、変換可能であることが必要です。インクルード先ページは、インクルード・ページがコピーされる前に変換可能である必要はありません。静的にインクルードされた一連のページは、それ自体では独立できないフラグメントとなる可能性があります。

動的なインクルードのロジック手法

動的なインクルードでは、リクエスト・ディスパッチャなどへのメソッド・コールが増加しますが、インクルード先ページで生成されたコードのサイズが大幅に増加することはありません。動的なインクルードによって、実行時の処理は、インクルード先ページからインクルード・ページに切り替えられます。これは、インクルード・ページのテキストをインクルード先ページに物理的にコピーする処理とは逆になります。

動的なインクルードの場合は、リクエスト・ディスパッチャへの追加コールのため処理のオーバーヘッドが増加します。

動的にインクルードされたページは、それ自体で変換および実行できる独立したエンティティであることが必要です。インクルード先ページも同様に、動的なインクルードなしに変換および実行できる独立したエンティティであることが必要です。

動的なインクルードと静的なインクルードのメリット、デメリットおよび代表的な使用例

静的なインクルードはページのサイズに影響を与え、動的なインクルードは処理のオーバーヘッドに影響を与えます。静的なインクルードの場合は、動的なインクルードに必要なリクエスト・ディスパッチャのオーバーヘッドを回避できますが、大規模なファイルの場合に問題が生じる可能性があります。（生成されたページ実装クラスのサービス・メソッドのサイズは、64KB に制限されています。）

また、静的なインクルードの過度の使用は、JSP ページのデバッグを困難にする可能性があるため、プログラムの実行をトレースすることがさらに困難になります。静的にインクルードしたページ間でのわかりにくい相互依存は避けてください。

静的なインクルードは通常、そのコンテンツが複数の JSP ページで繰り返し使用される、小規模なファイルをインクルードするために使用します。次に例を示します。

- ロゴまたは著作権メッセージをアプリケーションの各ページの最上部や最下部に静的にインクルードする場合。
- 複数のページに必要な宣言やディレクティブ（Java クラスのインポートなど）を含むページを静的にインクルードする場合。
- アプリケーションの各ページから、集中ステータス・チェッカ・ページを静的にインクルードする場合。（6-8 ページの「JSP アプリケーションの監視」を参照。）

動的なインクルードは、モジュール方式のプログラミングに役立ちます。1 つのページを、独立して実行したり、別のページの出力の一部を生成するために使用できます。動的にインクルードされたページは、インクルード先ページのサイズを増やさずに、複数のインクルード先ページで再利用できます。

JSP タグ・ライブラリでのサービスおよびリソース参照の注釈の使用

リリース 10.1.3.1 より、OC4J では、J2EE 5.0 および JSP 2.1 の仕様で定義されている JSP タグ・ライブラリでの注釈のサポートを開始しました。

J2SE 5.0 以上では、構成データおよび依存性を、Java コードの外部リソースにメタデータとして指定できます。これを、注釈とも言います。このようなデータを、構成ファイル、またはサービス（EJB や Web サービスなど）およびリソース参照（データソースや JMS 宛先など）の注釈で定義できます。

次の制約に注意してください。

- J2EE コンテナを JVM 1.5 以降で実行している場合にかぎり、注釈の使用とリソース・インジェクションが可能です。以前のバージョンの JVM では、リソース・インジェクションがサポートされていません。
- `tags_reuse_default` パラメータの `runtime` 値は推奨されていません。そのため、推奨されていない `runtime` の値を `tags_reuse_default` パラメータで使用する場合、注釈はサポートされません。
- インジェクションは JSP ページまたはタグ・ファイルではサポートされません。

- 注釈は、グローバル・タグ・ライブラリ（既知の場所にあるタグ・ライブラリまたは TLD キャッシュが有効化されている `jsp-taglib-locates` で定義されているタグ・ライブラリ）ではサポートされません。注釈は、アプリケーション内で使用されるタグ・ライブラリでのみサポートされています。
- JavaServer Pages でサポートされている一連の注釈は、サーブレットでサポートされている注釈のサブセットです。

  次の注釈は JavaServer Pages でサポートされています。

  - EJB 注釈
  - EJBs 注釈
  - PersistenceContext 注釈
  - PersistenceContexts 注釈
  - PersistenceUnit 注釈
  - PersistenceUnits 注釈
  - PostConstruct 注釈
  - PreDestroy 注釈
  - Resource 注釈
  - Resources 注釈
  - WebServiceRef 注釈
  - WebServiceRefs 注釈

  次の注釈は、サーブレットではサポートされていますが、JavaServer Pages ではサポートされていません。

  - DeclaresRoles 注釈
  - RunAs 注釈

前述の例を除き、JavaServer Pages でサポートされている注釈は、サーブレットでサポートされているものと同じです。これは『Oracle Containers for J2EE サーブレット開発者ガイド』の第 7 章「サービスおよびリソース参照の注釈の使用」に記載されています。

JSP アプリケーションの監視

JSP アプリケーションの全般的な管理や監視に、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カ・ページを使用すると便利です。集中チェッカ・ページは、各ページの実行中に次のタスクを実行できます。

- セッション・ステータスのチェック
- ログイン・ステータスのチェック（Cookie をチェックして有効なログインが行われたかどうかを確認するなど）
- 使用状況プロファイルのチェック（マウス・クリックやページへの接続など、対象イベントを記録するためのロギング機能を実装している場合）

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したセッション・チェッカ・クラスの MySessionChecker があります。

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

   valueBound(HttpSessionBindingEvent event)
   {...}

   valueUnbound(HttpSessionBindingEvent event)
   {...}

   ...
}
```

たとえば、次のような内容を含むチェッカ・ページ、centralcheck.jsp を作成できます。

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

centralcheck.jsp が含まれるページでは、セッション終了時に sessioncheck がスコープ外になると同時に、サーブレット・コンテナが、MySessionChecker クラスに実装されている valueUnbound() メソッドをコールします。これは、セッション・リソースを管理するために実行されます。centralcheck.jsp は、アプリケーションの各 JSP ページの最後に含めることができます。

大量の静的なコンテンツまたはタグ・ライブラリの使用の管理

JSP ページに大量の静的なコンテンツ（実行時に変更されるコンテンツのない大量の HTML コード）が含まれると、変換と実行の速度が低下する場合があります。

2 つの対処方法があり、いずれの対処方法でも変換速度は向上します。

- 静的な HTML コードを別のファイルに格納し、jsp:include タグを使用して、実行時にその出力を JSP ページ出力にインクルードします。jsp:include タグの詳細は、1-12 ページの「標準の JSP アクション・タグ」を参照してください。

  重要： 静的な include ディレクティブは有効ではありません。このディレクティブを使用すると、インクルード・ファイルが変換時にインクルードされ、そのコードが実際にインクルード先ページにコピーされます。これでは、問題の解決になりません。

- 静的な HTML コードを Java リソース・ファイルに格納します。

  external_resource 構成パラメータを有効にすると、JSP トランスレータがこの操作を実行します。このパラメータの詳細は、3-2 ページの「JSP 構成パラメータの概要」を参照してください。

  注意： external_resource parameter パラメータは、Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。

  事前に変換する場合は、ojspc ツールの -extres オプションが同じ機能を提供します。

  注意： 静的な HTML コードをリソース・ファイルに格納すると、前述の jsp:include による対処よりもメモリーのフットプリントが大きくなる場合があります。これは、クラスがロードされるたびに、ページ実装クラスはリソース・ファイルをロードする必要があるためです。

大量の静的なコンテンツを含む JSP ページ、または大量のタグ・ライブラリを使用する JSP ページで起こりうるもう 1 つの問題は、ほとんど（全部ではない場合）の JVM では、単一のメソッド内のコード・サイズが 64 KB に制限されていることです。`javac` によるコンパイルは可能ですが、JVM では実行できません。実質的に 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・メソッドに追加されるため、この問題は、JSP トランスレータの実装によっては、JSP ページに関する問題になる可能性があります。Java コードは静的な HTML をブラウザに出力するために生成され、スクリプトレットからの Java コードは直接コピーされます。

同様に、JSP ページの Java スクリプトレットのサイズが大きいため、サービス・メソッドでサイズ制限の問題が発生する場合があります。ページ内の Java コードが問題の原因の場合は、コードを JavaBeans に移動する必要があります。

大量のタグ・ライブラリの使用によって JSP ページのサイズが制限される場合、ページを複数のページに分割し、必要に応じて `jsp:include` タグを使用するという解決策が一般的です。

メソッド変数宣言とメンバー変数宣言の使用の比較

1-7 ページの「スクリプト要素」で、メンバー変数の宣言には JSP の `<%! ... %>` 宣言を使用し、メソッド変数は `<% ... %>` スクリプトレットで宣言する必要があることを説明しました。

変数の使用方法に応じて、適切な機能を使用して宣言するように注意してください。

- `<%! ... %>` JSP 宣言構文内で宣言される変数は、JSP トランスレータが生成したページ実装クラス内のクラス・レベルで宣言されます。この場合、オブジェクト・インスタンスを宣言すると、そのオブジェクトは複数のリクエストから同時にアクセスできます。したがって、`page` ディレクティブ内で `isThreadSafe="false"` が宣言されていないかぎり、オブジェクトはスレッド・セーフであることが必要です。
- `<% ... %>` JSP スクリプトレット構文内で宣言される変数は、ページ実装クラスのサービス・メソッドに対してローカルです。メソッドがコールされるたびに、変数またはオブジェクトのインスタンスが個別に作成されるため、スレッド・セーフティは必要ありません。

次に `decltest.jsp` の例を示します。

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

この場合、ページ実装クラスのコードは次のようになります。

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

public class decltest extends ... {
   ...

   // ** Begin Declarations
   double f=0.0;                  // *** f 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 : これは、概念を説明するためのコードです。クラスの大部分は簡素化のために削除されているため、JSP トランスレータで生成されるページ実装クラスの実際のコードとは異なります。

page ディレクティブの使用

この項では、次の page ディレクティブの特性について説明します。

- page ディレクティブは静的で、変換時に効果があります。パラメータ設定が実行時に評価されるようには指定できません。
- page ディレクティブの Java import 設定は、JSP ページまたは変換単位内に累積されます。

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。動的な設定を実行時に解析するようには指定できません。次に例を示します。

例 1 次の page ディレクティブは有効です。

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

例 2 次の page ディレクティブは無効で、エラーが発生します。(この例では EUCJIS がハードコードされていますが、実行時に動的に決定されるすべてのキャラクタ・セットに当てはまります。)

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

一部の page ディレクティブ設定には、代替策があります。例 2 の場合は、コンテンツ・タイプを動的に設定できる setContentType() メソッドを使用できます。

page ディレクティブ属性の重複設定の禁止

JSP 仕様では、複数のディレクティブ属性（page ディレクティブの import 属性を除く）が単一の JSP 変換単位（JSP ページおよび include ディレクティブを使用してインクルードされたページ）内で異なる値で再設定されていないことを、Web コンテナが確認する必要があることが記載されています。

ディレクティブ属性の重複設定が許可されている JSP 標準に対する下位互換性のために、OC4J では forgive_dup_dir_attr 構成パラメータが提供されています。JSP 2.0 では、異なる属性が異なる値を持つ場合のみ、このパラメータを設定する必要があります。このパラメータの詳細は、3-2 ページの「JSP 構成パラメータの概要」を参照してください。たとえば、以前にコーディングしたページには、page ディレクティブの language 属性がすべて java に設定されているセグメントが複数含まれている可能性があります。

> **注意：** forgive_dup_dir_attr パラメータは、Oracle Containers for J2EE 10*g*（10.1.3.1.0）では推奨されていません。

属性の重複設定については、次の点に注意してください。

- JSP 仕様では、異なる属性が設定されているかぎり、複数の page ディレクティブが許可されます。

  次の例は有効です。

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

  または

  ```
  ------------------------------
  <%@ page buffer="0kb" %>
  <%@ include file="b.jsp" %>
  ------------------------------
  ------------------------------
  b.jsp
  <%@ page session="false" %>
  ------------------------------
  ```

  ただし、この例では、forgive_dup_dir_attr パラメータを設定する必要があります。

  ```
  <%@ page buffer="none" %>
  <%@ page buffer="0kb" %>
  ```

  または

  ```
  <%@ page buffer="none" buffer="0kb" %>
  ```

  または

  ```
  ------------------------------
  <%@ page buffer="0kb" %>
  <%@ include file="b.jsp" %>
  ------------------------------
  ------------------------------
  b.jsp
  <%@ page buffer="3kb" %>
  ------------------------------
  ```

- 変換単位は、JSP ページおよび include ディレクティブを使用してインクルードされたページで構成されていますが、jsp:include タグを使用してインクルードされたページは含まれていません。jsp:include タグを使用してインクルードされたページは、変換時に静的にインクルードされるのではなく、実行時に動的にインクルードされます。詳細は、6-6 ページの「静的なインクルードと動的なインクルードの使用の比較」を参照してください。

したがって、次の場合は許可されます。

```
------------------------------
<%@ page buffer="0kb" %>
<jsp:include page="b.jsp" />
------------------------------
------------------------------
b.jsp
<%@ page buffer="3kb" %>
------------------------------
```

- 前述のように、page ディレクティブの import 属性には、属性の重複設定に対する制限は当てはまりません。

生成したメソッドの 64K のサイズ制限に対する対処方法

Java 仮想マシン（JVM）では、Java メソッド当たりのコード量を 64K（65536 バイト）に制限しています。アプリケーションで大きな JSP を使用すると、実行時にこの制限を超える可能性があります。一般的に、JSP のファイル・サイズは最小限にとどめる必要があります。

JSP で大量のタグ・ライブラリを使用する場合、**「アプリケーション」→「JSP コンテナのプロパティ」→「カスタム・タグのコード・サイズの削減」**プロパティ（または global-web-application.xml の reduce_tag_code 構成パラメータ）を有効にして、カスタム・タグから生成されるコードのサイズを削減します。ただし、これにより、JSP コンパイルのパフォーマンスが影響を受ける可能性があります。

JSP ファイルのネーミング規則の順守

サーブレット仕様では、JSP ページのファイル名に拡張子 .jsp が必要です。ただし、サーブレット仕様 2.3 では、個別に変換可能な完全なページと、個別に変換できないページ・セグメント（include ディレクティブを介して移入したファイルなど）の違いは識別されません。

JSP 仕様 2.0 では、次のことをお薦めします。

- トップレベルのページ、動的にインクルードされるページおよび転送されるページ（独自に変換可能なページ）には、.jsp 拡張子を使用します。
- include ディレクティブを介して導入したページ・セグメント（独自に変換できないファイル）には、.jsp を使用しないでください。このようなファイルに対する拡張子の指定はありませんが、.jsph、.jspf または .jsf の使用をお薦めします。

JSP での空白の保持とバイナリ・データの使用

Web コンテナは通常、ソース・コードの空白（改行を含む）をブラウザへの出力内に保持します。開発者が意図しない空白が挿入される場合があるため、一般的に、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・コード内での改行の使用に応じて、異なる HTML 出力が作成されます。

例 1: 改行なし

次の JSP ページでは、`Date()` コールと `getParameter()` コールの後に改行がありません。（`Date()` コールで始まる 3 行目と 4 行目は、実際には 1 行のコードが折り返されています。）

`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=5>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

このコードによるブラウザへの HTML 出力は、次のとおりです。日付の後に空白行はありません。

```
<HTML>
<BODY>
Tue May 30 20:07:04 PDT 2000
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=5>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

例 2: 改行あり

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

`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=5>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

このコードによるブラウザへの HTML 出力は、次のとおりです。

```
<HTML>
<BODY>
Tue May 30 20:9:20 PDT 2000


<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=5>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

SP ページでバイナリ・データを回避する理由

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。通常は、かわりにサーブレットを使用します。

- JSP 実装は、バイナリ・データを処理するよう設計されていません。つまり、`JspWriter` クラスには実際のバイトを書き込むためのメソッドがありません。
- 実行中、Web コンテナは空白を保持しています。空白は不要な場合があります。このため、JSP ページは、ブラウザへのバイナリ・データ出力（`.gif` ファイルなど）の生成など、空白が重要な場合には適していません。

  次に一般的な例を示します。

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

  この場合、ブラウザは、出力バッファのバッファリングに応じて、バイナリ・データの途中または最後にある不要な改行文字を受け取ります。コードの行間で改行を使用しないことによってこの問題は回避できますが、この方法は望ましいプログラミング・スタイルではありません。

  > **注意：** 前述の例は説明のみを目的としているため、将来の Oracle JSP バージョンや他の Web コンテナに移植できない場合があります。

JSP テクノロジは、動的なテキスト・コンテンツのプログラミングの簡素化を目的としているため、JSP ページでバイナリ・データの生成を試行すると、JSP テクノロジの特性を失うことになります。

JSP のベスト・プラクティス

次の各項では、OC4J にデプロイする JSP ページを開発する際、考慮すべきベスト・プラクティスについて説明します。

HTTP セッションに関する注意

HTTP セッションは、メモリーの使用量により、Web アプリケーションにパフォーマンス面でのオーバーヘッドを追加します。セッションは、JSP ではデフォルトで有効になっています。

不要な場合の HTTP セッションの不使用

HTTP セッション・オブジェクトが不要な場合は、使用しないでください。JSP ページで HTTP セッションが不要な場合（基本的に、セッション属性の格納または取得が不要な場合）は、セッションを使用しないように指定できます。page ディレクティブで、次のように指定します。

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

これによって、セッションの作成または取得のオーバーヘッドが減少するため、ページのパフォーマンスが改善されます。

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。

使用していないセッションの無効化

JSP で HTTP セッションを使用しない場合、必ず `javax.servlet.http.HttpSession.invalidate()` メソッドを使用して明示的に各セッションを取り消し、占有されているメモリーを解放します。

OC4J でのデフォルトのセッション・タイムアウトは 30 分です。アプリケーションの `web.xml` ファイルで、`<session-config>` 要素の `<session-timeout>` パラメータを設定することにより、特定のアプリケーションでこの値を変更できます。

ojspc ユーティリティを使用した JSP ページの事前変換

`ojspc` ユーティリティを使用した、デプロイ前の JSP ページの事前変換を検討してください。JSP ページにはユーザーが最初にアクセスするため、このユーティリティを使用すると、ページを変換するときのパフォーマンスの損失が発生しません。このユーティリティの詳細な使用方法は、第 4 章「ojspc による JSP ページのプリコンパイル」を参照してください。

HTTP セッションにおける更新オブジェクトの再設定の確認

分散可能な Web アプリケーションの JSP を作成している場合、HTTP セッションで変更されたオブジェクトを再設定するようページをコーディングし、セッション・オブジェクトの更新がクラスタ環境でレプリケートされることを確認してください。

OC4J では、セッションで保存されたセッション・オブジェクトをシリアライズ化しますが、オブジェクトのデータ・メンバーが変更されてもセッション・オブジェクトは再シリアライズ化されません。したがって、更新されたセッション状態はレプリケートされません。この問題は JSP 特有のものではなく、サーブレットなどでも、セッションで変更されたオブジェクトを再設定する必要があります。

JSP では、変更可能なセッション属性ごとに、`HttpSession` に対して `setAttribute()` をコールするスクリプトレットを含めることにより、セッション状態を確実にレプリケートする必要があります。

`<jsp:useBean>` タグを使用して session スコープ Bean を作成する場合、`setAttribute()` をコールしてセッションの更新された Bean を再設定します。Bean の作成時に Bean に設定されたプロパティはセッションに対して設定されますが、Bean のプロパティ値の更新は設定されません。

JSP ページのバッファの無効化

JSP ページのバッファを無効にします。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。このバッファ（デフォルトは 8 KB）は、動的な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、転送またはエラー・ページをページで使用する場合に必要です。このような機能をページで使用しない場合は、page ディレクティブのバッファを無効にできます。

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

これによって、メモリーの使用量が減少し、バッファをコピーする出力手順が不要になるため、パフォーマンスが改善されます。

JSP ページへのリダイレクトを使用しない転送

1 つの JSP ページから別のページに制御を渡す方法は 2 つあります。<jsp:forward> 標準アクションタグを使用する方法と、スクリプトレットでリダイレクト URL を response.sendRedirect() に渡す方法です。

<jsp:forward> オプションの方が速く、効率的です。この標準アクションを使用すると、転送されたターゲット・ページは JSP ランタイムによって内部的に起動され、継続してリクエストが処理されます。転送が実行されたことはブラウザに認識されず、ユーザーからは処理全体がシームレスに感じられます。

sendRedirect() を使用すると、ブラウザはリダイレクトされたページに新しいリクエストを送信する必要があります。ブラウザに表示される URL は、リダイレクトされたページの URL に変更されます。さらに、リダイレクトには新しいリクエストが含まれるため、すべての request スコープのオブジェクトが、リダイレクトされたページで使用できなくなります。

ユーザーがページを再ロードする場合に、実行されている実際のページが URL に反映されるようにする場合のみ、リダイレクトを使用してください。

アクセスを制限するために直接起動から JSP ページを除外する方法

一部の JSP ページをアプリケーションからのみアクセス可能にし、ユーザーが直接起動できないようにする場合があります。特に、Model-View-Controller（MVC）などのアーキテクチャで必要になる場合があります。

たとえば、フロントエンドまたは表示ページが index.jsp であるとします。ユーザーは、このページに直接アクセスする URL リクエストを介してアプリケーションを起動します。ここで、index.jsp には、2 番目のページの included.jsp が含まれ、3 番目のページの forwarded.jsp に転送されるとします。さらに、ユーザーが URL リクエストを介してこれらのページを直接起動できないようにする必要があるとします。

これを行うには、included.jsp と forwarded.jsp をアプリケーションの /WEB-INF ディレクトリに格納します。これらのページをこのディレクトリに格納すると、URL リクエストを介して直接起動できなくなります。直接起動しようとすると、ブラウザでエラー・レポートが生成されます。

index.jsp ページの文は次のようになります。

```
<jsp:include page="WEB-INF/included.jsp"/>
...
<jsp:forward page="WEB-INF/forwarded.jsp"/>
```

アプリケーション構造は次のようになります。構造には、サーブレット、JavaBeans または他のクラス用の標準の classes ディレクトリ、および JAR ファイル用の標準の lib ディレクトリが含まれます。

```
index.jsp
WEB-INF/
   web.xml
   included.jsp
   forwarded.jsp
   classes/
   lib/
```

効率的なメモリー使用のための JSP タイムアウトの利用

<orion-web-app> 要素の jsp-timeout 属性に、整数値（秒単位）を指定します。この値が経過した後にリクエストされなかった場合、その JSP ページはメモリーから削除されます。これによって、コール頻度の低いページに割り当てられているリソースが解放されます。デフォルト値は 0（ゼロ）で、タイムアウトはありません。<orion-web-app> 属性は、OC4J の global-web-application.xml ファイルおよび orion-web.xml ファイル内にあります。OC4J インスタンス内のすべてのアプリケーションにタイムアウトを適用する場合、global-web-application.xml ファイルを変更します。特定のアプリケーションの構成値を設定する場合、アプリケーション固有の orion-web.xml ファイル内のファイルを変更します。

デプロイ用の EAR ファイル内の JSP ファイルのパッケージ化

OC4J では、ファイルを直接適切な場所にコピーすることによる JSP ページのデプロイをサポートしています。ページの開発やテストの際、この機能は大変便利です。

ただし、JSP ベースのアプリケーションを本番用にリリースする場合、この方法はお薦めしません。必ず JSP ファイルを Enterprise Archive（EAR）ファイルにパッケージ化して、標準的なデプロイおよび複数のアプリケーション・サーバーに渡るデプロイを可能にしてください。

パフォーマンス改善のための動的キャラクタ・セットのチェックの無効化

デフォルトでは、JspWriter が出力 / 書込みのたびに動的なキャラクタ・セットをチェックします。パフォーマンス改善のため、OC4J 全体のシステム・プロパティを -Dcheck.dynamic.charset="false" に設定して、このチェックを無効にします。

これにより、キャラクタ・セットのチェックは、JspWriter での出力 / 書込みごとではなく、リクエストごとに 1 回実行されます。

JSP ファイルでの引用符の正しい使用

次の例には、JSP ファイル内での誤った引用符の使用例が含まれています。

```
    <td width="100%"
style="border-left: 1px solid #000; border-right: 1px solid #000; <dhv:evaluate if="<%=
!i.hasNext() %>">border-bottom: 1px solid
#000;</dhv:evaluate>">
```

問題は style 属性の 2 つ目の引用符（if= の後）です。

```
    style="border-left: 1px solid #000; border-right: 1px solid #000;
<dhv:evaluate if="<%= !i.hasNext() %>">border-bottom: 1px solid
#000;</dhv:evaluate>">
```

Tomcat はこの正しくないコードを受け入れ、処理を続行します。

OC4J は JSP およびサーブレットの仕様に従い、正しくないコードを受け入れず、エラーを発行します。

この問題を解決するには次を実行する必要があります。

- 引用文字（"）を括弧として読み取り、常に開く / 閉じるのセマンティクスに解釈します。
- 内部属性値に引用文字を正しく使用します。

  内部属性値に対する引用文字の正しい使用例です。

  ```
  <td width="100%" style="border-left: 1px solid #000; border-right: 1px solid
  #000; <dhv:evaluate if='<%= !i.hasNext() %>'>border-bottom: 1px solid
  #000;</dhv:evaluate>">
  ```

他にも引用符に関する問題があります。

プログラマがページの動的コンテンツに依存する外観を作成する場合があります。たとえば、個別に表の最終行を表示するとします。一部のライブラリはこれが可能であり、ライブラリをタグの属性内に含めて、周囲のタグがコールされる前に評価することができます。

Tomcat では次のシーケンスを使用できます。

```
attribute="text"another text"text"
```

Tomcat では、属性値が行の最初の引用符から最後の引用符までの単一の値として解析されます。

一方、OC4J では、属性値が最初の引用符と 2 つ目の引用符の間にある文字列 **text** として解析されます。後続の文字列 **another text"text"** はエラーとみなされます。

正しくない使用方法 :

```
<td width="100%"
   height="<dhv:evaluate if="<%= !i.hasNext() %>">20</dhv:evaluate>">
```

パーサーでの表示 :

```
<td width="100%"
   height="<dhv:evaluate if=" <<-- ERROR Tag not closed!!
   <%= !i.hasNext() %>">20</dhv:evaluate>">
```

正しい使用方法 :

```
<td width="100%"
   height="<dhv:evaluate if='<%= !i.hasNext() %>'>20</dhv:evaluate>">
```

サーブレットの使用

JSP ページのコーディングは多くの点で便利ですが、サーブレットのコールが必要な場合があります。その一例は、バイナリ・データを出力する場合です。

このため、サーブレットと JSP ページ間での往復が、1 つのアプリケーション内で必要になる場合があります。次の各項で、その方法を説明します。

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

JSP ページからのサーブレットの起動

ある 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` 操作タグと `jsp:forward` 操作タグを使用して、JSP ページからサーブレットを起動できます。（1-12 ページの「標準の JSP アクション・タグ」を参照。）次に例を示します。

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

ページの実行中にこの文が出現すると、ページ・バッファがブラウザに出力され、サーブレットが実行されます。サーブレットの実行が終了すると、制御が JSP ページに戻されて、ページの実行が続行されます。この機能は、JSP ページ間における `jsp:include` 操作タグと同じ機能です。

また、JSP ページ間における `jsp:forward` 操作タグと同様に、次の文は、ページ・バッファをクリアし、JSP ページの実行を終了して、サーブレットを実行します。

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

JSP ページから起動したサーブレットへのデータの受渡し

JSP ページからサーブレットに動的にインクルードまたは転送を行う場合は、jsp:param タグを使用して、サーブレットにデータを渡すことができます（別の JSP ページへのインクルードまたは転送でも同様です）。

jsp:param タグは、jsp:include タグまたは jsp:forward タグ内で使用できます。次に例を示します。

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

jsp:param タグの詳細は、1-12 ページの「標準の JSP アクション・タグ」を参照してください。

適切なスコープの JavaBean、または HTTP リクエスト・オブジェクトの属性を使用して、JSP ページとサーブレットとの間でデータの受渡しを行うこともできます。リクエスト・オブジェクトの属性の使用方法は、6-21 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」で説明します。

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使用すると、サーブレットから JSP ページを起動できます。この機能を使用するには、次の手順に従ってコードを作成します。

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. 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、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() メソッドへの入力として指定します。

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

   この手順の実行前か実行中に、HTTP リクエスト・オブジェクトの属性を必要に応じて使用して、JSP ページにデータを受け渡すことができます。詳細は、次項の「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. リクエスト・ディスパッチャの include() メソッドまたは forward() メソッドを起動し、HTTP リクエスト・オブジェクトとレスポンス・オブジェクトを引数として指定します。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 タグおよび jsp:forward タグの機能と同じです。include() メソッドは一時的に制御を移すのみで、後で、起動したサーブレットに実行の制御が戻されます。

   forward() メソッドは、出力バッファをクリアすることに注意してください。

   注意： リクエスト・オブジェクトとレスポンス・オブジェクトは、標準のサーブレット機能（javax.servlet.http.HttpServlet クラスに指定されている doGet() メソッドなど）を使用して、事前に取得されています。

JSP ページとサーブレット間でのデータの受渡し

前の項の「サーブレットからの JSP ページの起動」で説明したように、リクエスト・ディスパッチャを使用してサーブレットから JSP ページを起動するとき、HTTP リクエスト・オブジェクトを必要に応じて使用して、データを受け渡すことができます。この操作は、次のいずれかの方法で実行できます。

- *name=value* のペアを持つ「?」構文を使用してリクエスト・ディスパッチャを取得すると、問合せ文字列を URL に追加できます。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）では、暗黙的な `request` オブジェクトの `getParameter()` メソッドを使用すると、パラメータ・セットの値を取得できます。

- HTTP リクエスト・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は、暗黙的な `request` オブジェクトの `getAttribute()` メソッドを使用すると、パラメータ・セットの値を取得できます。

注意： `jsp:param` タグのかわりに、この項で説明する機能を使用して、JSP ページからサーブレットにデータを受け渡すことができます。

JSP とサーブレット間の相互作用のサンプル

この項では、前の各項で説明した機能を使用した JSP ページとサーブレットのサンプルを示します。JSP ページの `Jsp2Servlet.jsp` には、サーブレットの `MyServlet` がインクルードされ、このサーブレットには別の JSP ページの `welcome.jsp` がインクルードされます。

Jsp2Servlet.jsp のコード

```
<HTML>
<HEAD> <TITLE> JSP Calling Servlet Demo </TITLE> </HEAD>
<BODY>

<!-- Forward processing to a servlet -->
<% request.setAttribute("empid", "234"); %>
<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>
```

Apache Tomcat から OC4J への JSP ページの移行

この項では、JavaServer Pages を含むアプリケーションを Tomcat から Oracle Application Server（OC4J）に移行する場合に使用する JSP 固有のポインタを示します。Tomcat から OC4J への移行に関する主な説明は、『Oracle Containers for J2EE サーブレット開発者ガイド』の第 6 章のアプリケーションの Apache Tomcat から OC4J への移行に関する項に記載されています。これには、Tomcat から OC4J への JSP コンパイルの問題に関する項も含まれています。

概要

JSP ページは Tomcat から Oracle Application Server へ簡単に移行でき、Tomcat 環境で選択した内容に応じて、コードの変更はほとんど必要ないか、あるいはまったく必要ありません。

Oracle Application Server 10*g* リリース 3（10.1.3.1）は、Sun 社の JavaServer Page 仕様バージョン 2.0 に準拠しています。Tomcat 5.5 も、バージョン 2.0 と互換性があります。Tomcat と Oracle Application Server Containers for J2EE（OC4J）の両方に同じバージョンの Java Server Pages 仕様が実装されているため、コア JSP 仕様の領域でこの 2 つに違いはありません。

また、Oracle Application Server 10*g* リリース 3（10.1.3.1）は、バージョン 1.2 との下位互換性があります。そのため、標準のバージョン 1.2 仕様で作成された JSP ページは、Oracle Application Server で正常に作動し、移行の手間は最小限で済みます。

JSP ページを新しい環境に移行する場合の主なタスクは、構成とデプロイです。独自の拡張子を使用すると、追加のタスクが必要になり、移行が複雑になります。

JSP ページの移行に関係するタスクは、JSP ページのパッケージとデプロイの方法によっても異なります。JSP ページは、単純な JSP ページとして、標準ディレクトリ構造内でその他のリソースとともにパッケージされる Web アプリケーションとして（WAR ファイル）、または Enterprise Application Archive（EAR）ファイルとしてデプロイできます。

移行手順

Tomcat から OC4J への JSP ページの移行は簡単で、移行タスクには構成、(WAR ファイルへの) パッケージおよび (適切なデプロイ・ディレクトリへの) デプロイが含まれます。これらのタスクは、手動あるいは Oracle JDeveloper を使用して実行できます。

単純な JSP ページの移行

JSP ページには、HTTP サーブレットのような特定のマッピングは必要ありません。単純な JSP ページは、JSP ページおよび JSP ページに必要なすべてのファイルを適切なディレクトリにコピーすることでデプロイできます。その他の登録は必要ありません。

注意 : JSP を含むすべてのタイプのアプリケーションのデプロイには、Application Server Control コンソールを使用することをお薦めします。ただし、説明のため、次の例では JSP ページを Application Server Control コンソールを使用せずに手動で移行する方法を示します。

OC4J のデプロイ・プロセスは、J2EE Web アプリケーションと様々な構成ファイルをデフォルトで提供することで簡略化されています。

単純な JSP ページを Tomcat から OC4J へ移行する一般的な手順は、次のとおりです。

1. OC4J のインスタンスが実行されていない場合は、起動します。

   Oracle Enterprise Manager 10*g* Application Server Control コンソールの「管理」Web ページを使用するか、次の `opmnctl` コマンド (ローカルで実行) を使用します。

   ```
   opmnctl @instance startproc ias-component=OC4J
   ```

2. Tomcat のディレクトリから OC4J の適切なディレクトリに JSP ページをコピーします。
3. URL を使用し、Web ブラウザから JSP ページをリクエストします。次に例を示します。

   ```
   http://<hostname>:7777/j2ee/MyJspPage.jsp
   ```

   `<hostname>` は、JSP ファイルをコピーした Oracle Application Server のホストです。

JSP ページの構成とデプロイの詳細は、『Oracle Containers for J2EE 構成および管理ガイド』を参照してください。

JSP ページのプリコンパイル

JSP ページは JSP コンパイラによって自動的にコンパイルされます。ただし、JSP ページのテストおよびデバッグを行う場合は、直接 JSP コンパイラにアクセスします。

JSP コンパイラは .jsp ファイルを解析して .java ファイルにします。次に、標準の Java コンパイラを使用して .java ファイルを .class ファイルにコンパイルします。

次のツールのいずれかを使用して、JSP ページをプリコンパイルできます。

- JSP 仕様に記載されている標準の jsp_precompile メカニズム。
- 第 4 章「ojspc による JSP ページのプリコンパイル」に記載されている `ojspc` という名前の OC4J コマンドライン・ユーティリティ。

注意 : OC4J に JSP ページをデプロイする前に、カスタムの Tomcat JSP タグを標準の JSP タグに置き換えると、コンパイルの問題を回避できます。

実行時エラーの処理

JSP ページを実行してクライアント・リクエストを処理している間に、ページの内側または外側（コールされた JavaBean 内など）で実行時エラーが発生する場合があります。この項ではエラー処理機能について説明し、基本的な例を示します。

サーブレットと JSP の実行時エラー処理機能

この項では、実行時例外の処理機能について説明します。JSP エラー・ページを使用する場合についても説明します。

一般的なサーブレット実行時エラー処理機能

JSP ページの実行中に発生した実行時エラーは、標準の Java 例外処理機能によって、次のいずれかの方法で処理されます。

- 標準の Java 例外処理コードを使用して、JSP ページ内の Java スクリプトレットで例外を取得して処理できます。
- JSP ページで例外が取得されない場合、リクエストおよび取得されなかった例外（`java.lang.Throwable` インスタンス）は、エラー・リソースに転送されます。JSP エラーは、この方法で処理することをお薦めします。この場合、エラーの情報を持つ例外インスタンスは、名前に `javax.servlet.jsp.jspException` を使用し、`setAttribute()` コールを介して、`request` オブジェクトに格納されます。

エラー・リソースの URL は、元の JSP ページで `page` ディレクティブの `errorPage` 属性を設定すると指定できます。（`page` ディレクティブなどの JSP ディレクティブの概要は、1-6 ページの「ディレクティブ」を参照してください。）

デフォルトのエラー・リソースの詳細は、Sun 社の Java サーブレット仕様 2.4 を参照してください。

JSP エラー・ページ

オプションとして、別の JSP ページを、元の JSP ページからの実行時例外のエラー・リソースとして使用する方法があります。JSP エラー・ページには、`isErrorPage="true"` に設定する `page` ディレクティブが必要です。この方法で定義されたエラー・ページは、`web.xml` ファイルで宣言されているエラー・ページよりも優先されます。

エラーの情報を持つ `java.lang.Throwable` インスタンスは、エラー・ページで JSP の暗黙的な `exception` オブジェクトを介してアクセス可能です。このオブジェクトにアクセスできるのはエラー・ページのみです。`exception` オブジェクトなど、JSP の暗黙的なオブジェクトの詳細は、1-10 ページの「暗黙的なオブジェクト」を参照してください。

元の JSP ページに `autoFlush="true"`（デフォルト設定）の `page` ディレクティブがあり、そのページの `JspWriter` オブジェクトのコンテンツがレスポンスの出力ストリームにすでにフラッシュされている場合、取得されなかった例外をエラー・ページに転送しようとしても、レスポンスをクリアできない可能性があることに注意してください。一部のレスポンスは、ブラウザがすでに受信している可能性があります。

エラー・ページの使用例は、次の「JSP エラー・ページの例」の項を参照してください。

JSP 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・ページの 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。

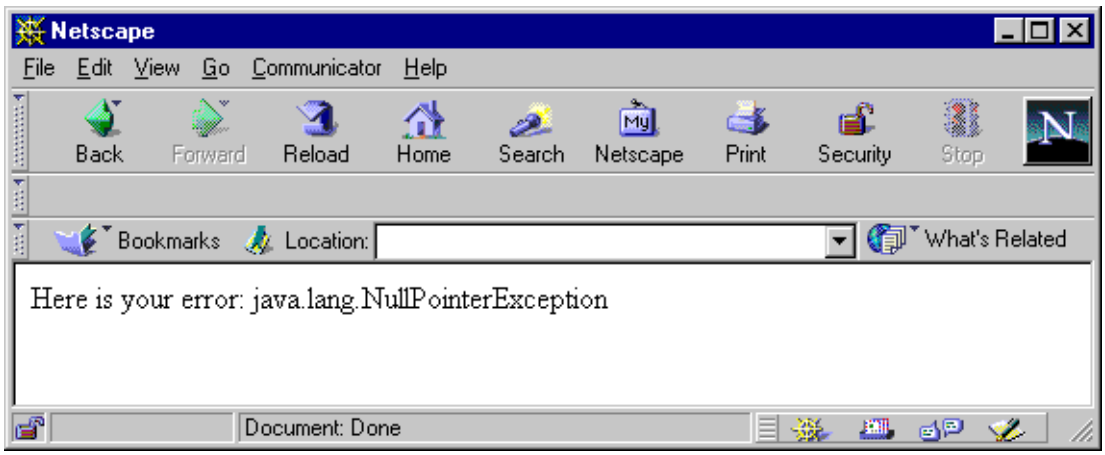
nullpointer.jsp のコード

```
<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>
```

この例では、次の内容が出力されます。



注意： 処理がエラー・ページに転送された場合、nullpointer.jsp の「Null pointer is generated below:」の行は出力されません。これは、jsp:include 機能と jsp:forward 機能の違いを示しています。jsp:forward の場合は、転送先ページの出力によって、転送元ページの出力が置換されます。

7

カスタム・タグの使用

この章では、カスタム・タグ、タグ・ライブラリ、およびベンダーが独自のライブラリの提供に使用できる基本的なフレームワークについて説明します。また、Oracle の拡張機能、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との比較についても説明します。この章には、次の各項が含まれます。

- カスタム・タグとは
- タグ・ハンドラの使用
- OC4J のタグ・ハンドラの機能
- タグ・ファイルの使用
- Web アプリケーション間でのタグ・ライブラリの共有

この章では、タグ・ライブラリの機能の概要について説明します。詳細は、Sun 社の JSP 仕様を参照してください。OC4J が提供するタグ・ライブラリについては、『Oracle Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

カスタム・タグの作成および実装方法については、多数の資料があります。このマニュアルで説明されている以外の情報は、次の資料を参照してください。

- Sun 社の JSP 仕様 2.0
- 次の Web サイトにある、`javax.servlet.jsp.tagext` パッケージに関する Sun 社の Javadoc

`http://java.sun.com/j2ee/1.4/docs/api/javax/servlet/jsp/tagext/package-summary.html`

カスタム・タグとは

カスタム・タグ（タグ拡張とも呼ばれる）は、他の Java コンポーネントによるカスタム・ロジックおよび出力を JSP ページに挿入できるようにする JSP 要素です。カスタム・タグを介して提供されるロジックは、タグ・ハンドラという Java オブジェクトによって実装されます。変換時に OC4J によって JSP 内のカスタム・タグが検出されると、タグ・ハンドラを取得して相互作用を行うコードが生成されます。

カスタム・タグは XML 構文を使用して JSP ページに含めます。タグにはボディを含めることも含めないこともできます。また、タグには、対応するタグ・ハンドラのプロパティと一致する XML 属性を含めることができます。

JSP 2.0 で、カスタム・タグの作成オプションは 2 つになりました。

- タグ・ハンドラ

  タグ・ハンドラ・クラスの作成を必要とするタグには、クラシックとシンプルの 2 種類があります。

  クラシック・タグ・ハンドラは、JSP 1.1 から導入されています。クラシック・タグは、それぞれのタグに対応するタグ・ハンドラ・クラスの実装に使用する Java インタフェースが複雑なため、作成がやや面倒です。また、動的な属性値については、Java の式に依存します。ただし、タグ・ボディに Java スクリプトレットまたは式を使用する場合は、これらのタグ・ハンドラを使用する必要があります。

  シンプル・タグ・ハンドラは JSP 2.0 で新しく導入され、クラシック・タグ・ハンドラに比べ、単純なライフサイクルとインタフェースを提供しています。タグのボディには JSP 式言語（EL）式を使用でき、まったくスクリプトを使用しないタグ開発が可能です。

- タグ・ファイル

  JSP 2.0 には、もう 1 つの革新的な新機能であるタグ・ファイルがあります。タグ・ファイルを使用すると、タグ・ハンドラ・クラスの作成およびコンパイルをせずに、タグ・ライブラリを JSP または XML 構文で完全に実装できます。かわりに、タグ・ファイルは OC4J JSP コンテナによってシンプル・タグ・ハンドラに変換され、コンパイルされます。実装が容易なため、タグ・ファイルはタグ・ハンドラ作成の魅力的な代替手段となります。

関連のあるタグ・ハンドラまたはタグ・ファイルは、タグ・ライブラリとしてまとめてパッケージングできます。タグ・ハンドラとして開発されたライブラリには、タグ・ライブラリ・ディスクリプタ（TLD）という XML 文書を含める必要があります。TLD は、各タグの構文を説明し、タグを対応するハンドラ・クラスにマップします。タグ・ファイル・ライブラリは TLD を必要としませんが、ライブラリをデプロイ用のアーカイブとしてパッケージングする場合はディスクリプタが必要です。

作成するカスタム・タグの種類は、必要に応じて異なります。Java 言語の柔軟性が必要な場合には、クラシックまたはシンプル・タグ・ハンドラを使用するタグが理想的です。プレゼンテーションが重視されるタグを作成する場合、または JSTL などの既存のタグ・ライブラリを利用する場合には、タグ・ファイルが便利です。

使用可能なタグ・ライブラリ

Oracle では、OC4J およびその他のコンポーネントとともに、幅広いタグ・ライブラリを提供しています。使用可能な Oracle のタグ・ライブラリの詳細は、『Oracle Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

さらに、他のタグ・ライブラリを様々なベンダーや組織から入手できます。たとえば、Jakarta Taglibs では、幅広いライブラリを `http://jakarta.apache.org/taglibs/` からダウンロードできます。

注意： JSTL は、OC4J 内の *`ORACLE_HOME`*`/j2ee/home/jsp/lib/taglib` ディレクトリにインストールされません。

デプロイされた Web アプリケーション間でタグ・ライブラリを共有する手順は、7-26 ページの「Web アプリケーション間でのタグ・ライブラリの共有」を参照してください。

カスタム・タグ・ライブラリの作成と使用を考慮する時期

カスタム・タグには、クリーンな JSP コードやコードの再利用性を含めた、様々な利点があります。状況によっては、カスタム・タグを作成または利用することがほぼ必須になります。特に、次の場合には必要です。

- カスタム・タグを作成しないと、プレゼンテーションと出力フォーマットに関する大量の Java ロジックをインクルードする必要が生じる場合
- 便利な JSP プログラミング・アクセスを機能に提供するには、カスタム・タグを作成しないと Java API の使用が必要となる場合
- JSP 出力の特殊な操作またはリダイレクションが必要な場合

多大な Java ロジックの削減

JSP 開発者に Java プログラミングの経験がない場合、ページで Java ロジック（たとえば、JSP 出力のプレゼンテーションとフォーマットを指示するロジック）をコーディングするのは困難な場合があります。

このような場合に、JSP タグ・ライブラリが役立ちます。出力を生成するこのようなロジックが多数の JSP ページで必要な場合、Java ロジックを置換するタグ・ライブラリは、JSP 開発者にとって非常に便利です。

API 機能に対する便利な JSP プログラミング・アクセスの提供

タグ・ライブラリの提供によって、Web アプリケーションのプログラマは、サーブレットまたは JSP スクリプトレットから製品の機能や拡張機能を使用するために、Java API に依存する必要がなくなります。タグ・ライブラリを使用すると、プログラマのタスクが大幅に軽減され、タグ・ハンドラによって適切な API コールが自動的に処理されます。

たとえば、OC4J では、電子メールとファイル・アクセス機能用に、タグと JavaBeans が提供されています。また、OC4J の Web Object Cache には、タグ・ライブラリと Java API が提供されています。同様に、Oracle Application Server Personalization には Java API が提供されていますが、OC4J にもパーソナライズ・アプリケーションをプログラミングする場合に使用できるタグ・ライブラリが提供されています。

JSP 出力の操作またはリダイレクト

カスタム・タグを使用するもう 1 つの状況は、レスポンス出力に関する特殊な実行時処理が必要な場合です。必要な機能を使用するには、処理手順を追加したり、出力をブラウザ以外の場所にリダイレクトする必要が生じる場合があります。

たとえば、タグ・テキストをブラウザではなくログ・ファイルにリダイレクトするカスタム・タグ `<cust:log>` を作成できます。

```
<cust:log>
   Text to log.
   More text to log.
   Yet more text to log.
</cust:log>
```

タグ・ハンドラの使用

次の各項では、カスタム・タグを使用することにより発生する操作のセマンティクスを定義するタグ・ハンドラについて説明します。

- クラシック・タグ・ハンドラとは
- シンプル・タグ・ハンドラとは
- タグ・ハンドラにおける属性の処理および文字列値の変換
- タグでのスクリプト変数の使用
- 外部タグ・ハンドラ・インスタンスへのアクセス
- タグ・ハンドラの実装

クラシック・タグ・ハンドラとは

クラシック・タグ・ハンドラは、標準の `javax.servlet.jsp.tagext.Tag` インタフェースを直接的または間接的に実装する Java クラスのインスタンスです。クラシック・タグ・ハンドラは作成が複雑だとみなされていますが、タグ・ボディか属性値に Java スクリプトレットまたは式を使用する場合は、これを使用する必要があります。

クラシック・タグ・ハンドラのインタフェース

タグ・ボディがあるかどうか、およびボディの処理方法に応じて、タグ・ハンドラは `javax.servlet.jsp.tagext` パッケージの次のいずれかのインタフェースを実装します。

- `Tag`: このインタフェースは、すべてのタグを処理するための基本メソッドを定義しますが、タグ・ボディの処理は含まれません。
- `IterationTag`: このインタフェースは、`Tag` を拡張し、タグ・ボディを反復するために使用されます。
- `BodyTag`: このインタフェースは、`IterationTag` を拡張し、タグ・ボディ・コンテンツにアクセスするために使用されます。

クラシック・タグ・ハンドラ・クラスは、これらのインタフェースのいずれかを直接実装したり、いずれかのインタフェースを実装するクラス（Sun 社が提供するサポート・クラスの 1 つなど）を拡張できます。

タグ・ハンドラは、必要に応じて、パラメータの受渡し、タグ・ボディの評価、および JSP ページ内の他のオブジェクト（他のタグ・ハンドラを含む）へのアクセスをサポートしています。

注意： JSP 仕様では、1 つの JSP ページ内で同じカスタム・タグを繰り返して使用する場合に、同じタグ・ハンドラ・インスタンスを使用するか、または別のインスタンスを使用するかは指示されていません。これは、JSP ベンダーが決定します。Oracle 実装については、7-17 ページの「OC4J のタグ・ハンドラの機能」を参照してください。

カスタム・タグの処理（タグ・ボディを使用する場合と使用しない場合）

カスタム・タグは、標準の JSP タグと同様に、ボディを使用する場合と使用しない場合があります。カスタム・タグでは、ボディを使用する場合でも、タグ・ハンドラによるボディ・コンテンツへのアクセスを必要としない場合があります。

次の 4 つの場合があります。

1. ボディがない場合

   この場合、開始タグと終了タグは使用せず、単一のタグのみが使用されます。次に、一般的な例を示します。

   ```
   <oracust:mytag attr="...", attr2="..." />
   ```

   このコードと次のコードは同等で、いずれも使用可能です。

   ```
   <oracust:mytag attr="...", attr2="..." ></oracust:abcdef>
   ```

   この場合、タグ・ハンドラは、Tag インタフェースを実装するか、または TagSupport を拡張する必要があります。

   TLD でのこのタグの <body-content> は、empty に設定する必要があります。

2. タグ・ハンドラによるボディ・コンテンツへのアクセスが不要で、1 回のみ実行されるボディを使用する場合

   この場合も、開始タグと終了タグの間に文のボディがありますが、タグ・ハンドラはボディを処理しません。ボディの文は、通常の JSP 処理に対してのみ渡されます。次に、一般的な例を示します。

   ```
   <foo:if cond="<%= ... %>" >
   ...body executed if cond is true, but body content not accessed by tag handler...
   </foo:if>
   ```

   この場合、タグ・ハンドラは、Tag インタフェースを実装する必要があります。

   TLD でのこのタグの <body-content> は、ボディ・コンテンツを変換する場合は JSP（デフォルト）、テンプレート・データとして処理する場合は tagdependent に設定する必要があります。

3. タグ・ハンドラによるボディ・コンテンツへのアクセスが不要で、反復して実行されるボディを使用する場合

   タグ・ボディが反復して処理されることを除いて、2 番目の場合と同じです。

   ```
   <foo:myiteratetag ... >
   ...body executed multiple times, according to attribute or other settings, but body
   content not accessed by tag handler...
   </foo:myiteratetag>
   ```

   この場合、タグ・ハンドラは、IterationTag インタフェースを実装する必要があります。

   TLD でのこのタグの <body-content> は、ボディ・コンテンツを変換する場合は JSP（デフォルト）、テンプレート・データとして処理する場合は tagdependent に設定する必要があります。

4. タグ・ハンドラによる処理が必要なボディを使用する場合

   この場合も、開始タグと終了タグの間に文のボディがありますが、タグ・ハンドラはそのボディ・コンテンツにアクセスする必要があります。

```
<oracust:mybodytag attr="...", attr2="..." >
...body accessed and processed by tag handler...
</oracust:mybodytag>
```

   この場合、タグ・ハンドラは、BodyTag インタフェースを実装する必要があります。

   TLD でのこのタグの <body-content> は、ボディ・コンテンツを変換する場合は JSP（デフォルト）、テンプレート・データとして処理する場合は tagdependent に設定する必要があります。

ボディ・コンテンツにアクセスするタグ・ハンドラ

タグ・ハンドラによるアクセスを必要とするボディ・コンテンツを使用するカスタム・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できます。

- javax.servlet.jsp.tagext.BodyTag

次の標準サポート・クラスは、BodyTag インタフェースおよび java.io.Serializable 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

このクラスは、Tag、IterationTag および BodyTag インタフェースから適切なメソッドを実装します。

注意： タグ・ハンドラが実際にボディ・コンテンツにアクセスする必要がない場合は、ほとんどの場合、BodyTag インタフェース（または BodyTagSupport クラス）を使用する必要はありません。使用すると、BodyContent オブジェクトを作成して維持するための不要なオーバーヘッドが発生することになります。ボディの反復が必要かどうかに応じて、Tag インタフェースまたは IterationTag インタフェース（または TagSupport クラス）を使用してください。

BodyTag インタフェースは、Tag インタフェースから基本的なタグ処理機能（doStartTag() メソッド、doEndTag() メソッド、およびその定義済の戻り値など）を継承します。このインタフェースは、IterationTag インタフェースからも機能（doAfterBody() メソッドおよびその定義済の戻り値など）を継承します。

BodyTag インタフェースには、継承した機能の他に、タグ・ボディから実行結果を取得する機能が追加されます。タグ・ボディの評価は、javax.servlet.jsp.tagext.BodyContent クラスのインスタンスにカプセル化されています。このインスタンスは、必要に応じて、ページ実装オブジェクトによって作成されます。

Tag インタフェースと同様に、BodyTag インタフェースに指定された doStartTag() メソッドは、int の戻り値として SKIP_BODY および EVAL_BODY_INCLUDE をサポートします。BodyTag インタフェースの場合、このメソッドは、int の戻り値として EVAL_BODY_BUFFERED もサポートします。次に、それぞれの値の概要を説明します。

- SKIP_BODY: ボディを評価しません。
- EVAL_BODY_INCLUDE: ボディを評価し、ボディ・コンテンツをタグ・ハンドラに対して使用可能にせずに、JSP out オブジェクトに渡します。これは、基本的に IterationTag インタフェースを実装するタグ・ハンドラを使用した場合の EVAL_BODY_INCLUDE の動作と同じです。
- EVAL_BODY_BUFFERED: タグ・ボディ・コンテンツを処理するための BodyContent オブジェクトを作成します。

BodyTag インタフェースには、次のメソッドの定義も追加されます。

- setBodyContent(): タグ・ハンドラの bodyContent プロパティ（BodyContent インスタンス）を設定します。
- doInitBody(): タグ・ボディの評価を準備します。

これらの手順は、タグ・ボディが評価される前に実行されます。ボディの評価中、JSP out オブジェクトは BodyContent オブジェクトにバインドされます。

各ボディの評価後に、IterationTag インタフェースを実装しているタグ・ハンドラに対して、ページ実装インスタンスはタグ・ハンドラの doAfterBody() メソッドをコールします。可能な戻り値は次のとおりです。

- SKIP_BODY: 反復を停止し、タグ・ボディを再評価しません。かわりに、doEndTag() をコールします。JSP out オブジェクトは、ページ・コンテキストからリストアされます。
- EVAL_BODY_AGAIN: 反復を継続し、タグ・ボディを再評価します。ボディは、評価時に現行の JSP out オブジェクトに渡されます。ボディの評価後、doAfterBody() メソッドが再度コールされます。

ボディの評価が完了しても多数の反復が必要な場合、ページ実装インスタンスは、タグ・ハンドラの doEndTag() メソッドを起動します。

BodyTag インタフェースを実装しているタグ・ハンドラでは、javax.servlet.jsp.tagext.BodyContent クラスのインスタンスを通じてタグ・ボディの評価結果にアクセスできます。このクラスは、javax.servlet.jsp.JspWriter クラスを拡張します。

BodyContent インスタンスは、JSP ページ・コンテキストの pushBody() メソッドを使用して作成されます。

BodyContent オブジェクトの一般的な使用方法は、次のとおりです。

- コンテンツを String インスタンスに変換し、その文字列を操作の値として使用します。
- コンテンツを JSP out オブジェクトに書き込みます。このオブジェクトは、開始タグが検出された時点でアクティブになっています。

シンプル・タグ・ハンドラとは

シンプル・タグ・ハンドラは JSP 2.0 の新機能で、クラシック・タグ・ハンドラと同様に強力な機能ですが、より単純な Java インタフェースとより簡単なライフサイクルによって、実装がはるかに容易です。

クラシック・タグ・ハンドラと異なり、シンプル・タグ・ハンドラ・オブジェクトは、OC4J JSP コンテナによってキャッシュされたり再利用されることはありません。そのかわり、オブジェクトはインスタンス化され、実行された後に破棄されます。このインタフェースでは、キャッシュされたり再利用されるものが存在しないため、使用時に複雑なキャッシング・セマンティクスがありません。この単純化されたライフサイクルにより、タグ・ハンドラの作成が容易になり、エラーも発生しにくくなります。

ここでは、シンプル・タグ・ハンドラを Java クラスとして実装する方法のみを説明しますが、シンプル・タグはタグ・ファイルを使用してすべて JSP 構文で実装することも可能です。詳細は、7-20 ページの「タグ・ファイルの使用」を参照してください。ただし、このオプションは、Java の柔軟性が不要な場合のみ使用してください。

SimpleTag インタフェース

シンプル・タグ・ハンドラ・クラスは、1 つのインタフェース `javax.servlet.jsp.tagext.SimpleTag` を実装します。このインタフェースに含まれるライフサイクル・メソッドは、`doTag()` の 1 つのみです。すべての反復、ボディ評価およびその他のタグ処理は、このメソッド内で実行されます。

理想としては、シンプル・タグ・ハンドラ・クラスで `javax.servlet.jsp.tagext.SimpleTagSupport` ユーティリティ・クラスを拡張します。このクラスは `SimpleTag` インタフェースを実装し、インタフェースのメソッドにデフォルトの実装を提供します。たとえば、このクラスは、ハンドラに渡されたタグ・ボディを返す `getJspBody()` を提供します。

タグ・ボディの評価は `setJspBody()` メソッドによって実行されます。このメソッドは、タグ起動のボディをカプセル化した `JspFragment` オブジェクトを使用して、OC4J JSP コンテナによって起動されます。（JSP フラグメントの詳細は、7-22 ページの「JSP フラグメントの使用」を参照してください。）タグ・ハンドラは、`JspFragment` に対して `invoke()` メソッドをコールし、必要な回数だけ何度でもボディを評価できます。このように、複数回の起動および評価が可能であるという性質は、フラグメントの主要な機能です。

クラシック・タグと異なり、シンプル・タグ拡張はコール元の JSP ページの基礎となるサーブレットの `PageContext` に依存せず、そのかわり、`PageContext` が拡張する `JspContext` に依存します。`JspContext` は、`JspWriter` の格納やスコープ付き属性の追跡などの汎用的なサービスを提供します。一方、PageContext はサーブレットのコンテキストでの JSP のサービスに固有の機能を持っています。シンプル・タグ・ハンドラは、これらのオブジェクトを介して、JSP ページから他の暗黙的なオブジェクト（`request`、`session` および `application`）を取得できます。

属性の使用

クラシック・タグと同様に、シンプル・タグの動作は、タグ・ハンドラ・クラスのプロパティに対応する属性を使用して制御できます。

属性には、単純、フラグメントおよび動的の 3 種類があります。

単純属性は、シンプル・タグ・ハンドラに渡される前に、タグの起動時に最初に OC4J JSP コンテナによって評価されます。属性は、構文 `attr="value"` を使用して開始タグ内で定義されます。値は、`String` 定数または EL 式として設定できます。

また、JSP 2.0 で導入された新しい要素である `<jsp:attribute>` 要素を使用して、カスタム・タグのボディで属性値を定義することも可能です。次の例では、`<jsp:attribute>` を使用して、`<my:helloWorld>` カスタム・タグの出力を使用し、Bean オブジェクトの `bar` プロパティの値を設定します。

```
<jsp:setProperty name="foo" property="bar">
   <jsp:attribute name="value">
      <my:helloWorld/>
   </jsp:attribute>
    </jsp:setProperty>
```

フラグメント属性は、名前付きフラグメントの作成に使用されます。JSP フラグメントの概要は、7-22 ページの「JSP フラグメントの使用」を参照してください。

動的な属性は、名前が示すとおり、タグ定義で指定されていない属性です。タグによって動的に属性を指定する機能は、JSP 2.0 の新機能で、シンプル・タグ・ハンドラおよびクラシック・タグ・ハンドラの両方で使用できます。同様の方法で処理される属性が多数含まれるタグの場合、動的な属性は特に便利です。

動的な属性をサポートするタグ・ハンドラは、TLD の tag 要素でそのことを宣言する必要があります。次に例を示します。

```
<tag>
  <description>
   Tag that echoes all its attributes and body content
  </description>
  <name>echoAttributes</name>
  <tag-class>jsp2.examples.simpletag.EchoAttributesTag</tag-class>
  <body-content>empty</body-content>
  <dynamic-attributes>true</dynamic-attributes>
</tag>
```

タグ・ハンドラにおける属性の処理および文字列値の変換

タグ・ハンドラ・クラスには、カスタム・タグの各属性について基礎となるプロパティがあります。このプロパティは、setter メソッドがある点などで、JavaBean のプロパティに類似しています。

タグ属性を設定するには、次の 2 つの方法があります。

- 属性がリクエスト時以外の属性の場合は、次のように、文字列リテラル値を使用して設定します。

  nrtattr="*string*"

  リクエスト時以外の属性の場合、基礎となるタグ・ハンドラ・プロパティが String 型でないと、Web コンテナは文字列値を適切な型の値に変換しようとします。

  タグ属性は Bean のプロパティに似ているため、その処理（文字列値から適切な型への変換など）も Bean のプロパティの処理に似ています。1-17 ページの「文字列値から Bean プロパティへの変換」を参照してください。

- 属性がリクエスト時の属性の場合は、次のいずれかの形式で、リクエスト時の式を使用して設定します。

  rtattr="<%=*expression*%>"

  rtattr="${ELexpression}"

  リクエスト時の属性の場合、変換は行われません。リクエスト時の式は、属性およびそれに対応するタグ・ハンドラ・プロパティ（任意のプロパティ型）に割り当てることができます。この操作は、ユーザー定義型のタグ属性などに適用できます。

タグでのスクリプト変数の使用

カスタム・タグに明示的に定義されたオブジェクトは、そのオブジェクトの ID をハンドルとして使用して、JSP ページ・コンテキストから他の操作で参照できます。次に例を示します。

```
<oracust:foo id="myobj" attr="..." attr2="..." />
```

この文によって、myobj オブジェクトは、myobj の宣言されたスコープに基づいて、ページ内のスクリプト要素で使用可能になります。id 属性は変換時の属性です。変数は次のいずれかの方法で指定できます。

- 変数用の <variable> 要素を TLD に指定し、変数の名前と型および追加情報を指定します。7-11 ページの「TLD の <variable> 要素を使用した変数宣言」を参照してください。
- タグ補足情報クラスを作成して、変数の名前と型、追加情報および関連ロジックを指定します。タグ補足情報クラスの名前は、TLD の <tei-class> 要素に指定します。7-12 ページの「タグ補足情報クラスを使用した変数宣言」を参照してください。

通常は、<variable> 機能を使用すると便利です。

Web コンテナによって、myobj がページ・コンテキストに入力されます。これによって、他のタグまたはスクリプト変数は、次の構文を使用して、後でこのオブジェクトを取得できます。

```
<oracust:bar ref="myobj" />
```

この myobj オブジェクトは、foo タグと bar タグのタグ・ハンドラ・インスタンスに渡されます。ここで必要なのは、オブジェクトの名前（myobj）のみです。

注意： id と ref はサンプルの属性名であるため、これらの属性名に対する事前定義の特別なセマンティクスはありません。属性名を定義し、ページ・コンテキスト内のオブジェクトを作成および取得するのは、タグ・ハンドラが行います。

スクリプト変数のスコープ

変数を作成するタグの <variable> 要素またはタグ補足情報クラスに、スクリプト変数のスコープを指定します。スコープには、次の int 定数のいずれかを指定できます。

- NESTED: スクリプト変数を定義する操作の開始タグと終了タグの間で使用可能なスクリプト変数の場合は、この設定を使用します。
- AT_BEGIN: 開始タグからそのページの最後まで使用可能なスクリプト変数の場合は、この設定を使用します。
- AT_END: 終了タグからそのページの最後まで使用可能なスクリプト変数の場合は、この設定を使用します。

TLD の <variable> 要素を使用した変数宣言

JSP 仕様 1.1 では、カスタム・タグでスクリプト変数を使用するには、タグ補足情報（TEI）クラスを作成する必要がありました。7-12 ページの「タグ補足情報クラスを使用した変数宣言」を参照してください。JSP 仕様 1.2 では、より簡単な方法が導入され、関連付けられたタグが定義されている TLD で <variable> 要素を使用できます。変数に関連するロジックが単純で、TEI クラスを使用する必要がない場合は、この方法で十分です。

<variable> 要素は、変数を使用するタグを定義する <tag> 要素内のサブ要素です。

変数の名前は、次のいずれかの方法で指定できます。

- <variable> 要素内の <name-given> サブ要素を使用して、変数の名前を直接指定します。

または

- <variable> 要素内の <name-from-attribute> サブ要素を使用して、タグ属性を指定します（タグ属性の値によって、変換時に変数の名前が指定されます）。

<name-given> および <name-from-attribute> 以外に、<variable> 要素には次のサブ要素があります。

- <variable-class> 要素は、変数のクラスを指定します。デフォルトは java.lang.String です。
- <declare> 要素は、その変数が新規に宣言される変数かどうかを指定します。この場合、JSP トランスレータが変数を宣言します。デフォルトは true です。false に設定すると、変数は、標準的な機能（jsp:useBean 操作、JSP スクリプトレット、JSP 宣言またはカスタム・アクションなど）を使用して、JSP ページですでに宣言されているとみなされます。
- <scope> 要素は、変数のスコープとして、NESTED、AT_BEGIN または AT_END のいずれかを指定します。これらの値の説明は、7-10 ページの「スクリプト変数のスコープ」を参照してください。デフォルトは NESTED です。

次の例では、タグ myaction に対して 2 つのスクリプト変数を宣言しています。<tag> 要素内で、変数の説明に直接関係ない部分は省略されていることに注意してください。

```
<tag>
   <name>myaction</name>
   ...
   <attribute>
     <name>attr2</name>
     <required>true</required>
   </attribute>
   <variable>
      <name-given>foo_given</name-given>
      <declare>false</declare>
      <scope>AT_BEGIN</scope>
   </variable>
   <variable>
      <name-from-attribute>attr2</name-from-attribute>
      <variable-class>java.lang.Integer</variable-class>
   </variable>
</tag>
```

最初の変数の名前は、foo_given にハードコードされています。デフォルトで、この変数の型は String になります。新規に宣言される変数ではなく、すでに宣言されている変数とみなされるため、そのスコープは開始タグからページの最後までとなります。

2 番目の変数の名前は、必須の attr2 属性の設定に基づきます。この変数の型は Integer です。デフォルトでは、この変数は新規に宣言され、そのスコープは NESTED（myaction の開始タグと終了タグの間）になります。

タグ補足情報クラスを使用した変数宣言

複雑な関連ロジックを持つスクリプト変数の場合、変数を宣言するために TLD 内で <variable> 要素を使用するだけでは不十分な場合があります。この場合は、スクリプト変数に関する詳細を javax.servlet.jsp.tagext.TagExtraInfo 抽象クラスのサブクラスに指定できます。このマニュアルでは、このようなサブクラスをタグ補足情報クラスと呼びます。タグ補足情報クラスは、タグ属性の追加の妥当性チェックをサポートし、スクリプト変数に関する追加情報を JSP 実行時機能に提供します。

Web コンテナは、タグ補足情報インスタンスを変換時に使用します。指定のタグのスクリプト変数で使用するタグ補足情報クラスは、TLD で指定します。次に、<tei-class> 要素の使用例を示します。

```
<tag>
  <name>loop</name>
  <tag-class>examples.ExampleLoopTag</tag-class>
  <tei-class>examples.ExampleLoopTagTEI</tei-class>
  <body-content>JSP</body-content>
  <description>for loop</description>
  <attribute>
     ...
  </attribute>
  ...
</tag>
```

次に、関連クラスを示します。これらのクラスも javax.servlet.jsp.tagext パッケージに含まれています。

- TagData: このクラスのインスタンスには、タグ・インスタンスの変換時の属性値情報が含まれます。
- VariableInfo: このクラスの各インスタンスには、実行時にタグによって宣言、作成または変更されたスクリプト変数の情報が含まれます。
- TagInfo: このクラスのインスタンスには、関連するタグに関する情報が含まれます。クラスは TLD からインスタンス化され、変換時のみ使用できます。TagInfo には、getTagName()、getTagClassName()、getBodyContent()、getDisplayName() および getInfoString() などのメソッドがあります。

詳細は、次の場所を参照してください。

http://java.sun.com/j2ee/1.4/docs/api/javax/servlet/jsp/tagext/package-summary.html

> **注意：** tag-extra-info 実装で TagInfo インスタンスを使用するのは、一般的ではありません。ただし、たとえば、複数のタグ・ライブラリと TLD に 1 つのタグ補足情報クラスをマッピングする場合には便利です。

TagExtraInfo クラスには、次のメソッドが関連しています。

- boolean isValid(TagData data)

  JSP トランスレータは、このメソッドをコールして、変換時にタグ属性の妥当性チェックを実行し、TagData インスタンスに渡します。

- VariableInfo[] getVariableInfo(TagData data)

  JSP トランスレータは、変換時にこのメソッドをコールして、TagData インスタンスに渡します。このメソッドは、VariableInfo インスタンスの配列を戻します（タグで作成されたスクリプト変数ごとに 1 つのインスタンスが含まれます）。

- void setTagInfo(TagInfo info)

  このメソッドをコールすると、TagInfo インスタンスがタグ補足情報クラスの属性として設定されます。このメソッドは通常、Web コンテナによってコールされます。

- TagInfo getTagInfo()

  このメソッドを使用して、タグ補足情報クラスの TagInfo 属性を取得します。TagInfo 属性は以前に設定されていることが前提です。

タグ補足情報クラスは、スクリプト変数に関する次の情報を使用して、各 VariableInfo インスタンスを構成します。

- 変数の名前
- 変数の Java 型（プリミティブ型以外）
- 新規に宣言された変数かどうかを示すブール値（この場合、JSP トランスレータが変数を宣言します）
- 変数のスコープ

重要： OC4J 9.0.4 実装では、getVariableInfo() メソッドは、スクリプト変数の Java 型について、完全修飾クラス名（FQCN）または部分修飾クラス名（PQCN）のいずれかを戻すことができます。以前のリリースでは FQCN が必須でしたが、パッケージ間でクラス名が重複するのを避けるため、現行のリリースでも FQCN の使用をお薦めします。プリミティブ型はサポートされていません。

外部タグ・ハンドラ・インスタンスへのアクセス

ネストされたカスタム・タグを使用する場合、ネストされたタグのタグ・ハンドラ・インスタンスは、外部タグのタグ・ハンドラ・インスタンスにアクセスできます。これは、ネストされたタグによる処理と状態管理に役立ちます。

この機能は、javax.servlet.jsp.tagext.TagSupport クラスの静的な findAncestorWithClass() メソッドを介してサポートされます。外部タグ・ハンドラ・インスタンスの名前が JSP ページ・コンテキストに指定されていない場合でも、その外部タグ・ハンドラ・インスタンスは、指定したタグ・ハンドラ・クラスのインスタンスの最も近くにある、引用符で囲まれたインスタンスであるため、アクセス可能です。

次の JSP コードの例を考えてみます。

```
<foo:bar attr="abc" >
   <foo:bar2 />
</foo:bar>
```

bar2 タグ・ハンドラ・クラス（表記規則に従って、クラス名は Bar2Tag になります）のコード内に、次の文を含めることができます。

```
Tag bartag = TagSupport.findAncestorWithClass(this, BarTag.class);
```

findAncestorWithClass() メソッドへの入力項目は、次のとおりです。

- findAncestorWithClass() がコールされたクラス・ハンドラ・インスタンスである this オブジェクト（この例では、Bar2Tag インスタンス）
- java.lang.Class インスタンスとしての、bar タグ・ハンドラ・クラス（この例では、BarTag とします）の名前

findAncestorWithClass() メソッドは、適切なタグ・ハンドラ・クラスのインスタンスを戻します。この例では、BarTag を javax.servlet.jsp.tagext.Tag インスタンスとして戻します。

これは、Bar2Tag インスタンスで bar タグ属性の値が必要な場合、または BarTag インスタンスについてのメソッドのコールが必要な場合に、Bar2Tag が外部の BarTag インスタンスにアクセスするために役立ちます。

タグ・ハンドラの実装

Java で実装されたクラシックまたはシンプル・タグ・ハンドラの主な作成手順は、次のとおりです。

- タグ・ハンドラ・クラスの作成およびコンパイル
- タグ・ハンドラのタグ・ライブラリへのパッケージング
- タグ・ライブラリ・ディスクリプタ（TLD）でのタグの定義

Oracle Technology Network（OTN）の Web サイトからダウンロード可能なサンプル・アプリケーションに、シンプル・タグ・ハンドラ実装の例がいくつか含まれています。サンプル・アプリケーションをダウンロードし、OC4J にデプロイした後、タグ・ハンドラの Java ソース・ファイルを開くと、どのように実装されているかを確認できます。

タグ・ハンドラ・クラスの作成

カスタム・タグごとに独自のハンドラ・クラスがあります。タグ・ハンドラ・クラス名は、表記規則によって決まります。たとえば、`abc` というタグは、`AbcTag` という名前になります。

タグ・ハンドラ・クラスは、PUBLIC な引数のないコンストラクタを必要とします。

タグ・ハンドラ・インスタンスは、通常、JSP ページ実装インスタンスによって引数が 0（ゼロ）のコンストラクタを使用して作成され、リクエスト時に使用されるサーバー・サイド・オブジェクトです。タグ・ハンドラには、Web コンテナで設定されたプロパティがあります。これには、カスタム・タグを使用する JSP ページのページ・コンテキスト・オブジェクトが含まれ、このタグを外部タグ内にネストして使用する場合には、タグ・ハンドラの親オブジェクトも含まれます。

TLD でのタグの定義

タグ・ハンドラが実装するそれぞれのタグは、ハンドラ・クラスを含むタグ・ライブラリとともにパッケージングされているタグ・ライブラリ・ディスクリプタ・ファイルで定義されている必要があります。TLD 内のタグ定義は、2 つの役割を果します。

- タグの構文の定義
- タグと対応するタグ・ハンドラ・クラス間のマッピングの提供

次のコードは、`ShuffleSimpleTag` ハンドラ・クラスが含まれているライブラリのディスクリプタである、`shuffle.tld` 内の `<my:shuffle>` タグの定義に基づいています。このハンドラは、シンプル・タグ・ハンドラの機能を示す「Random Tic-Tac-Toe」サンプル・アプリケーションに含まれています。このアプリケーションは、Oracle Technology Network の Web サイトからダウンロードできます。

次の例では、TLD の形式を示しています。すべての TLD ファイルは、TLD と JSP のバージョンを指定する XML スキーマを指定している、ルートの `<taglib>` 要素で始める必要があります。

```
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd"
    version="2.0">
  <tlib-version>1.0</tlib-version>
  <short-name>SimpleTagLibrary</short-name>
  <uri>/SimpleTagLibrary</uri>
  <tag>
   ...
  </tag>
</taglib>
```

次に、<tag> 要素で囲まれた <my:shuffle> タグの定義が示されます。

```
<tag>
  <name>shuffle</name>
   <tag-class>oracle.otnsamples.jsp20.simpletag.ShuffleSimpleTag</tag-class>
   <body-content>empty</body-content>
   <attribute>
     <name>fragment1</name>
     <required>true</required>
     <fragment>true</fragment>
   </attribute>
   <attribute>
     <name>fragment2</name>
     <required>true</required>
     <fragment>true</fragment>
   </attribute>
   <attribute>
     <name>fragment3</name>
     <required>true</required>
     <fragment>true</fragment>
   </attribute>
</tag>
```

<tag> 要素のサブ要素によって、次のようにタグを定義します。

- 必須の <name> サブ要素は、タグの名前を指定します。
- 必須の <tag-class> サブ要素は、対応するタグ・ハンドラ・クラスの名前を指定します。タグ・ハンドラ・クラスについては、7-4 ページの「タグ・ハンドラの使用」を参照してください。
- <body-content> サブ要素は、タグ・ボディ（ある場合）の処理方法を示します。
- 各 <variable> サブ要素（ある場合）およびその下のサブ要素は、スクリプト変数を定義します。スクリプト変数については、7-10 ページの「タグでのスクリプト変数の使用」を参照してください。<variable> 要素は、状況が比較的複雑ではなく、スクリプト変数のロジックでタグ補足情報クラスを必要としない場合に使用します。変数名を指定するには、名前を直接指定する場合は <name-given> サブ要素を使用し、変数名を指定するタグ属性の名前を指定する場合は <name-from-attribute> サブ要素を使用します。その他に、変数のクラスを指定する <variable-class> サブ要素、変数のスコープを指定する <scope> サブ要素、および変数が新規に定義されるものであるかどうかを指定する <declare> サブ要素があります。詳細は、7-11 ページの「TLD の <variable> 要素を使用した変数宣言」を参照してください。<variable> 内には、オプションの <description> 要素もあります。
- 各 <tei-class> サブ要素（ある場合）は、スクリプト変数を定義するタグ補足情報クラスの名前を指定します。これは、<variable> 要素のみでは変数を宣言できない場合に使用します。詳細は、7-12 ページの「タグ補足情報クラスを使用した変数宣言」を参照してください。
- 各 <attribute> サブ要素（ある場合）およびその下のサブ要素は、カスタム・タグの使用時に渡すことが可能なパラメータ（この例では指定されたフラグメント）に関する情報を提供します。<attribute> のサブ要素には、属性名を指定する <name> 要素、属性値の Java 型を示す <type> 要素（オプション）、属性が必須かどうかを指定する <required> 要素（デフォルト値は false）、および属性の値として実行時の式を受け入れるかどうかを指定する <rtexprvalue> 要素（デフォルト値は false）があります。例と関連情報は後述します。<attribute> 内には、オプションの <description> 要素もあります。

注意： Oracle Application Server 10*g*（9.0.4）では、OC4J JSP コンテナは <type> 要素を無視します。この要素は、TLD を調べる際に参照用としてのみ使用されます。次の点にも注意してください。

- <rtexprvalue> が false に指定されている場合、リテラル属性値に対する <type> の値（ある場合）は常に java.lang.String になります。
- <rtexprvalue> が true に指定されている場合は、このタグ属性に対応するタグ・ハンドラ・プロパティの型によって、指定する <type> の値（ある場合）が決まります。

JSP ページでのタグの宣言

タグ・ハンドラが含まれているタグ・ライブラリは、<taglib> ディレクティブを使用してファイル内で参照されます。<my:...> 接頭辞は、この接頭辞を含んでいるタグが対応する TLD 内で定義されていることを示します。

```
<%@ taglib uri="/WEB-INF/shuffle.tld" prefix="my" %>
```

JSP 仕様 1.1 で最初に定義されているように、JSP ページの taglib ディレクティブでは、特定のタグ・ライブラリを定義する TLD の名前、および WAR ファイル構造における物理的な場所を完全に指定できます。次に例を示します。

```
<%@ taglib uri="/WEB-INF/oracustomtags/tlds/mytld.tld" prefix="oracust" %>
```

アプリケーションに対して相対的な場所として指定します（この例に示すように、「/」で始まります）。アプリケーションに対して相対的な構文については、1-23 ページの「JSP ページのリクエスト」を参照してください。

TLD は、/WEB-INF ディレクトリまたはサブディレクトリ内に含まれている必要があることに注意してください。

かわりに、JSP 仕様 1.1 以降で定義されているように、taglib ディレクティブは、TLD ではなく JAR ファイルの名前、およびアプリケーション相対の物理的な場所を指定できます。JAR ファイルには、単一のタグ・ライブラリおよびそれを定義する TLD が含まれます。この場合、JSP 仕様 1.1 では、TLD は JAR ファイル内に次のように格納され、名前が付けられる必要がありました。

```
META-INF/taglib.tld
```

JSP 仕様 1.1 では、JAR ファイルは、/WEB-INF/lib ディレクトリ内に格納する必要もありました。

次に、タブ・ライブラリ JAR ファイルを指定する taglib ディレクティブの例を示します。

```
<%@ taglib uri="/WEB-INF/lib/mytaglib.jar" prefix="oracust" %>
```

複数の Web アプリケーションで共有できるようにタグ・ライブラリをパッケージングする方法を説明している、「単一の JAR ファイルへの複数のタグ・ライブラリと TLD のパッケージング」も参照してください。

JSP でのタグの使用

TictactoeManual.jsp ページには、<my:shuffle> タグが使用されており、これは 3 行、3 列からなる碁盤状の表を描きます。各列には特定の色のタイルが置かれます。これは、異なる 3 色のタイル（またはイメージ）を持つ <my:tile> タグをコールすることによって表示されます。これらのコールのラッピングには、<my:shuffle> タグが使用されます。タグ・ハンドラによってランダムに 3 つのタイル・フラグメントが実行され、ページがロードされるたびに各行の色付きのタイルをシャッフルします。

```
<my:shuffle>
   <jsp:attribute name="fragment1">
     <tr>
       <my:shuffle>
         <jsp:attribute name="fragment1">
           <my:tile imageVal="../images/blue_plain.gif" />
          </jsp:attribute>
          <jsp:attribute name="fragment2">
            <my:tile imageVal="../images/yellow_plain.gif" />
       </jsp:attribute>
       <jsp:attribute name="fragment3">
         <my:tile imageVal="../images/pink_plain.gif" />
       </jsp:attribute>
         </my:shuffle>
      </tr>
    </jsp:attribute>
      ...
</my:shuffle>
```

OC4J のタグ・ハンドラの機能

この項では、タグ・ハンドラのプーリング、および生成コードのサイズ縮小に関する OC4J の JSP 拡張機能について説明します。この項には、次の項目が含まれます。

- タグ・ハンドラの再利用（タグ・プーリング）の無効化または有効化
- タグ・ハンドラのコード生成

タグ・ハンドラの再利用（タグ・プーリング）の無効化または有効化

パフォーマンスを改善するために、各 JSP ページ内でタグ・インスタンスを再利用するように指定できます。この機能は、タグ・プーリングとも呼ばれます。

Oracle Containers for J2EE リリース 3（10.1.3）では、コンパイル時タグ・プーリング・モデルがサポートされています。タグ・ハンドラ再利用のロジックとパターンは、コンパイル時（JSP ページの変換時）に決定されます。この方法は、同じページ内に多数のタグがある（たとえば、数百のタグがある）アプリケーションの場合、パフォーマンスを効率的に改善できます。

OC4J JSP コンテナでは、タグ・プーリングは tags_reuse_default パラメータを使用して構成されます。このパラメータの設定方法の詳細は、3-2 ページの「JSP 構成パラメータの概要」を参照してください。

注意： 実行時タグ・プーリング・モデルは、以前はデフォルトのメカニズムでしたが、Oracle Containers for J2EE 10*g*（10.1.3.1.0）からはサポートされなくなりました。

タグ・ハンドラ再利用のコンパイル時モデルの有効化または無効化

次のいずれかの方法で、タグ・ハンドラ再利用をコンパイル時モデルに変更できます。

- 構成パラメータの tags_reuse_default を compiletime に設定します。
- 構成パラメータの tags_reuse_default を compiletime_with_release に設定します。

compiletime_with_release に設定すると、同じページ内で同じタグ・ハンドラが使用されるたびに、タグ・ハンドラの release() メソッドがコールされます。このメソッドは、タグ・ハンドラ実装の詳細に従って、状態情報を解放します。

たとえば、タグが使用されるたびに状態情報を解放するようにタグ・ハンドラがコーディングされている場合は、compiletime_with_release 設定が適切です。タグ・ハンドラの実装、および使用するコンパイル時設定に関する詳細が不明な場合は、それぞれの値を試してください。

注意 :

- タグ・ハンドラ再利用の設定を compiletime、compiletime_with_release または runtime モデル間で変更した場合は、JSP ページを再変換する必要があります。
- tags_reuse_default の設定が compiletime または compiletime_with_release の場合、ページ・コンテキストの oracle.jsp.tags.reuse 属性は無視されます。

コンパイル時タグ・プーリング・モデルを使用できる場面

JSP 2.0 仕様に準拠して、タグに対して定義されている属性セットが使用のたびに毎回類似している場合のみ、タグ・インスタンスを再利用できます。この制限は、compiletime および compiletime-with-release タグ・プーリング・モデルの両方に適用されます。

次の例では、JSTL の core ライブラリのタグを使用して、タグ・インスタンスの再利用を示しています。

```
<%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>

// New instance of c:forEach tag is created.
// Note the inclusion of the "var" attribute.
<c:forEach var='item' begin='2' end='10'>

// New instance of c:out tag is created.
<c:out value="foo"/>

</c:forEach>

<%out.println("****");%>

// The first c:forEach tag instance cannot be reused, since the "var"
// attribute does not exist. A new tag instance is therefore created.
<c:forEach begin='1' end='10'>

// The existing c:out tag instance is reused, as
// the set of attributes is identical in both usages.
<c:out value="bar"/>
```

compiletime タグ・プーリング・モデルのコード・パターン

次に、タグ・プーリングのデフォルトの compiletime モデルのコード・パターンを示します。

```
try {
  tag01.doStartTag();
  ...
  tag01.doEndTag();
  // reuse tag01 without calling release()
  tag01.doStartTag();
  ...
  tag01.doEndTag();
}
catch (Throwable e) {
  tag01.release();
}
```

compiletime-with-release タグ・プーリング・モデルのコード・パターン

次に、compiletime-with-release オプションのコード・パターンを示します。このオプションでは、同じ JSP 内の同じタグ・ハンドラの使用の合間に、タグ・ハンドラ・オブジェクトに対して release() メソッドをコールします。

```
try {
  tag01.setPageContext(pageContext);
  tag01.doStartTag();
  ...
  tag01.doEndTag();
  tag01.release();
  // reuse tag01 with calling release()
  tag01.doStartTag();
  ...
  tag01.doEndTag();
  tag01.release();
}
catch (Throwable e) {
  tag01.release();
}
```

タグ・ハンドラのコード生成

Oracle JSP 実装では、カスタム・タグを使用するための生成コードのサイズが縮小されています。また、JSP 構成フラグの reduce_tag_code を true に設定すると、さらにサイズを縮小できます。

ただし、このフラグを有効にすると、タグ・ハンドラを最大限に再利用するコード生成パターンになりません。

タグ・ファイルの使用

次の各項では、タグ・ファイルの概要と詳細を説明します。

タグ・ファイルとは

タグ・ファイルは JSP 2.0 で導入され、これを使用することで、JSP 作成者は Java の知識がまったくなくても、JSP 構文を使用してカスタム・タグ・ライブラリを完全に作成できます。実際のところ、タグ・ファイルの作成は JSP の作成とほぼ同じで、違いはわずかです。実装が容易なため、タグ・ファイルは従来のタグ・ハンドラに替わる魅力的な手段となります。

タグ・ファイルのソースは、JSP コードの再利用可能なフラグメントが含まれているテキスト・ファイルです。タグ・ファイルで使用できる有効なコンテンツは次のとおりです。

- 標準の JSP アクション（`<jsp:useBean>` など）
- カスタム・タグ
- JSTL タグ
- EL 式または関数
- テンプレート・テキスト

従来のカスタム・タグと異なり、タグ・ファイルは対応するタグ・ハンドラ・クラスを必要としません。かわりに、各ファイルは OC4J JSP コンテナによってシンプル・タグ・ハンドラに変換およびコンパイルされます。タグ・ファイルによって、Java プログラミングの必要性がなくなり、Java に関する詳しい作業知識を持たない JSP 作成者にもタグ・ライブラリの作成が可能になります。

JSP ページと同様に、タグ・ファイルでは、処理方法を指示するために標準の JSP ディレクティブが使用されます。唯一の例外は `page` ディレクティブで、これは使用できません。そのかわり、タグ・ファイルでは、タグ・ファイル処理に特有の属性が含まれている `tag` ディレクティブを使用します。

`attribute` および `variable` などのその他のディレクティブは、タグ・ファイルでのみ使用できます。詳細は、7-21 ページの「タグ・ファイルでの属性の使用」および 7-22 ページの「タグ・ファイルでの変数を介したデータの公開」を参照してください。

タグ・ボディの処理

カスタム・タグのボディの処理方法は、対応するタグ・ファイルの tag ディレクティブの body-content 属性で定義されます。デフォルト値は scriptless で、前述の JSP または静的なテキスト要素がすべてタグ・ボディで使用できることを示します。ただし、値から推測できるとおり、Java スクリプティング要素は使用できません。タグ・ボディを受け入れないタグは、次に示すように、empty として宣言する必要があります。

```
<%@ tag body-content="empty" %>
```

タグのボディは、タグ・ファイル内でのみ使用可能な <jsp:doBody> 標準アクションによって評価されます。タグ・ボディ内のすべての動的な JSP 要素がコールされ、生成された出力は、評価結果内のボディでテンプレート・テキストと組み合されます。

結果は変数に格納され、<jsp:doBody> タグの var または varReader 属性を使用して指定できます。var 属性は、結果を String として捕捉し、多くの場合に使用可能である必要があります。

varReader 属性は、結果を java.io.Reader オブジェクトとして捕捉します。標準またはカスタム・タグ、もしくは Reader から入力を読み込む EL 関数と組み合せて使用する場合、こちらの方が効率的なことがあります。変数が指定されていない場合、出力は暗黙的な JspWriter オブジェクトに送信されます。

SimpleTag インタフェースの説明にもあったとおり、タグ起動のボディは JspFragment オブジェクトに変換され、これがタグ・ハンドラに渡されて処理されます。この場合、text の値は、Web ブラウザで太字および斜体フォントで表示されます。次の「タグ・ファイルでの属性の使用」で説明するとおり、フラグメントはタグ・ファイルのコンテキストで非常に便利です。

タグ・ファイルでの属性の使用

属性は、attribute ディレクティブを使用してタグ・ファイル内で直接宣言されます。次の例では、required 属性が true に設定されています。これは、ページ作成者がタグ要素のボディを渡すときにこの属性の値を指定しない場合、エラーが返されることを示します。デフォルト値は false です。

```
<%@ attribute name="category" required="true" %>
```

クラシックおよびシンプル・タグ・ハンドラと同様、タグ・ファイルでも、宣言されていない動的な属性を使用できます。それには、タグ・ディレクティブの dynamic-attributes 属性に、タグ起動時に渡される動的な属性の名前と値を含むマップの名前を設定します。マップには、それぞれの動的な属性の名前と値がキー / 値のペアとして含まれている必要があります。

```
<%@ tag body-content="scriptless" dynamic-attributes="dynattrs" %>
```

タグ・ファイルでの変数を介したデータの公開

変数もまた、variableディレクティブを使用してタグ・ファイル内で直接宣言されます。これは、TLD内でタグ・ハンドラによって使用される、変数を宣言するための<variable>要素に類似しています。name-given属性は変数の名前を指定し、variable-class属性は型を定義します。

```
<%@ variable name-given="current" variable-class="java.lang.Object"
scope="NESTED" %>
```

タグ・ハンドラと同様、変数のディレクティブのscope属性は、AT_BEGIN、AT_ENDまたはNESTEDの3つの値のうちの1つを受け入れます。これらは、コール元のJSPが変数を検出する場所を制御します。

タグ・ファイルはローカルのpageスコープにアクセスできます。これは、コール元のJSPのpageスコープとは異なります。異なるスコープを使用することにより、コール元のページとタグ・ファイルのpageスコープ変数に同じ名前が使用されている場合の混乱を防ぎます。多くの場合、タグ・ファイルで変数名をハードコードするのではなく、タグ・ボディで提供される属性を使用して、コール元のJSPによって変数名を指定する方が便利です。

```
<%@ variable name-from-attribute="var" alias="current"
variable-class="java.lang.Object" scope="NESTED" %>
```

この例では、前の例で使用されていたname-given属性のかわりに、name-from-attributeおよびalias属性が使用されています。name-from-attribute値は、変数名を提供しているタグ属性の名前です。

alias属性値は、タグ・ファイルのローカルのpageスコープ変数の名前を宣言します。これは、OC4J JSPコンテナによってコール元のJSPのpageスコープにコピーされます。ページの作成者はカスタム・タグ内の変数に任意の名前を割り当てられますが、タグ・ファイルの開発時には固定された名前（ここではalias）が必要なため、この属性が必要になります。

JSP フラグメントの使用

JSPフラグメントは、カスタマイズされたコンテンツの作成に使用できるテンプレートとみなすことができます。JSP 2.0で導入されたJSPフラグメントは、シンプル・タグ拡張とともに使用する新しいオプションで、タグ・ファイルとシンプル・タグ・ハンドラの両方に関係があります。ただし、フラグメントの作成は、通常、タグ・ファイルのコンテキスト内で行われます。

タグ・ファイルと同様に、フラグメントは動的なJSP要素（標準のJSPアクション・タグ、カスタム・タグまたはEL式）と、タグ起動によってシンプル・タグ・ハンドラに渡されるオプションの静的なテンプレート・テキストをカプセル化したものです。

フラグメントは、必要に応じて、タグ・ハンドラによって0回以上起動および評価できます。フラグメント内のJSP要素はすべてのスコープ付き変数の現行の値にアクセスするため、通常、結果は起動のたびに異なります。これらの特性により、フラグメントは次のような場合に使用すると便利です。

- 条件付きデータまたはテンプレート・テキストの表示
- 反復内のデータ・セットの表示

JSP フラグメントの作成

タグ・ファイルで使用するための名前付きフラグメントの作成には、2 段階の手順が必要です。まずタグ・ファイル内でフラグメントを宣言し、次に、カスタム・タグのボディ内の <jsp:attribute> 標準アクション・タグでボディを定義する必要があります。

フラグメントは、タグ・ファイル内で attribute ディレクティブを使用して宣言します。ディレクティブの fragment 属性を true に設定すると、フラグメントはタグ・ハンドラによって評価されます。デフォルトの設定 false では、属性をタグ・ハンドラに渡す前に OC4J JSP コンテナで評価するように強制します。これにより、フラグメントを 0 回以上評価することが可能になります。

```
<%@ attribute name="frag1" fragment="true" %>
<%@ attribute name="frag2" fragment="true" %>
```

次に、JSP 構文を使用して、タグ・ファイル内の関連付けられたロジックを定義します。次のタグ・ファイルのコードは、JSTL タグを使用して実装された条件付きロジックを示します。テストの評価が true の場合、frag1 が起動されます。

```
<c:when test="${empRow[2] >= 10000}">
   <c:set var="name" value="${empRow[0]}"/>
   <c:set var="phone" value="${empRow[1]}"/>
   <c:set var="salary" value="${empRow[2]}"/>
   <jsp:invoke fragment="frag1"/>
</c:when>
```

フラグメントは、タグ・ファイル内の <jsp:invoke> 標準アクション・タグを使用して、名前で起動されます。<jsp:doBody> と同様に、このアクションはタグ・ファイル内でのみ使用できます。タグ・ボディ内のスコープ付き変数（この場合は EL 演算子）はタグ・ファイルによって設定されるため、起動のたびにフラグメントをタグ・ファイル・コードによってカスタマイズすることが可能になります。

フラグメントのコンテンツは、前述のとおり JSP 要素と静的な HTML で構成され、カスタム・タグのボディ内の <jsp:attribute> 標準アクション・タグのボディ内で定義されます。

```
<tags:EmpDetails deptNum="80">
 <jsp:attribute name="fragment1">
   <tr bgcolor="#FFFF99" align="center">
    <td><font color="#CC0033">${name}</font></td>
    <td><font color="#CC0033">${phone}</font></td>
    <td><font color="#CC0033">${salary}</font></td>
   </tr>
  </jsp:attribute>
</tags:EmpDetails>
```

タグ・ファイルの例

次に、タグ・ボディの評価の結果を格納する var 属性の使用方法を表した単純な例を示します。

サンプルの doBodyVarTest.tag ファイルは、Example.jsp 内の <tags:doBodyVarTest> タグのボディの評価結果を格納する text という名前の変数を定義します。また、EL 変数の評価結果を格納する frag1 と frag2 という 2 つのフラグメント属性も定義します。

タグ・ファイルが実行されると、Web コンテナにより、フラグメント属性とタグ・ボディという 2 種類のフラグメントがタグ・ファイルに渡されます。タグ・ファイル内では、<jsp:invoke> 要素を使用してフラグメント属性が評価され、<jsp:doBody> 要素を使用してタグ・ファイルのボディが評価されます。それぞれの種類のフラグメントの評価結果は、レスポンスに送信されるか、または後の操作のために EL 変数に格納されます。

```
<%@ attribute name="frag1" required="true" fragment="true" %>
<%@ attribute name="frag2" required="true" fragment="true" %>
<%@ variable name-given="text" scope="NESTED" %>
 <jsp:doBody var="text" />
  <TABLE border="0">
   <TR>
    <TD>
     <b><jsp:invoke fragment="frag1"/></b>
    </TD>
   </TR>
   <TR>
    <TD>
     <i><jsp:invoke fragment="frag2"/></i>
    </TD>
   </TR>
  </TABLE>
```

Example.jsp コードにより、doBodyVarTest タグのインスタンスが作成されます。<jsp:body> タグには、タグ・ハンドラによって text 変数に設定される文字列が含まれています。<jsp:body> タグは、シンプル・タグのボディを明示的に指定するために使用されます。また、タグ起動のボディに 1 つ以上の <jsp:attribute> 要素が使用される場合、タグ・ボディの指定に使用できるのはこのタグのみです。

```
<%@ taglib tagdir="/WEB-INF/tags" prefix="tags" %>

<html>
 <body>
  <tags:doBodyVarTest>
   <jsp:attribute name="frag1">
    ${text}
   </jsp:attribute>
   <jsp:attribute name="frag2">
    ${text}
   </jsp:attribute>
   <jsp:body>
    Have a great day!
   </jsp:body>
  </tags:doBodyVarTest>
 </body>
</html>
```

タグ・ファイルの実装

タグ・ライブラリをタグ・ファイルとして実装する手順は簡単です。この章で概要が説明されているとおりにタグ・ファイルを作成するのみです。

タグ・ファイルの作成

この章で前述したとおり、タグ・ファイルの作成は JSP の作成と非常に似ています。タグ・ファイルで使用できる要素のサマリーは、7-20 ページの「タグ・ファイルとは」を参照してください。

JAR ファイル内のタグ・ファイルはコンパイルされていないタグ・ファイルとして残るため、誰でもテキスト・エディタを使用してコードを参照できる点に注意してください。

タグ・ファイルのパッケージング

タグ・ファイルは次のいずれかの方法でデプロイできます。

- アプリケーションのディレクトリ構造内の `/WEB-INF/tags` ディレクトリに、タグを直接ドロップします。

  この方法でデプロイされた場合、タグ・ファイルはタグ・ライブラリ・ディスクリプタを必要としません。ただし、他のアプリケーションの `/WEB-INF/tags` ディレクトリにもタグをコピーしないかぎり、タグはそのアプリケーションからのみアクセス可能です。

- JAR ファイルにパッケージングします。

  この方法でタグ・ファイルをパッケージングするには、TLD が必要です。TLD で定義されていない JAR 内のタグ・ファイルは、OC4J JSP コンテナによって無視されます。

JAR にパッケージングされると、タグ・ファイルは JAR 内の `/META-INF/tags` の下に追加されます。その後、JAR は Web アプリケーションの `/WEB-INF/lib/` ディレクトリにインストールされます。

JAR 内の各タグ・ファイルは、TLD の `<tag-file>` 要素内で定義されます。これは、クラシックまたはシンプル・タグ・ハンドラの定義に使用される `<tag>` 要素と異なります。それぞれの `<tag-file>` 要素は 2 つのサブ要素を取ります。

- `<name>` 要素は、タグ名を定義し、一意である必要があります。
- `<path>` 要素はタグ・ファイルの絶対パスを指定するため、`/META-INF/tags` で始まる必要があります。

次の例は、配置用に JAR にパッケージングされた 2 つのタグ・ファイルを定義する TLD を示しています。

```
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd"
   version="2.0">
  <tlib-version>1.0</tlib-version>
  <short-name>MyTagFiles</short-name
  <uri>/MyTagFiles</uri>
  <description>JSP 2.0 tag files</description>
  <tlib-version>1.0</tlib-version>
  <short-name>My Tag Files</short-name>
  <tag-file>
    <name>EmpDetails</name>
    <path>/META-INF/tags/mytags/EmpDetails.tag</path>
  </tag-file>
  <tag-file>
    <name>ProductDetails</name>
    <path>/META-INF/tags/mytags/ProductDetails.tag</path>
  </tag-file>
</taglib
```

JSP でのタグ・ファイルの宣言

JSP ページは、taglib ディレクティブを使用して、タグ・ファイルが実装されたタグ・ライブラリをインポートします。タグ・ライブラリの接頭辞は、prefix 属性を使用して指定されます。ただし、これ以外に含まれる属性は、タグ・ファイルのデプロイ方法によって異なります。

タグ・ファイルが JAR にパッケージングされていない場合、tagdir 属性の値には、タグ・ファイルが格納されているディレクトリへのコンテキスト相対パスを設定する必要があります。

```
<%@ taglib tagdir="/WEB-INF/tags/mytags" prefix="mytags" %>
```

タグ・ファイルが JAR にパッケージングされている場合、uri 属性が使用され、TLD 内の <uri> 要素のコンテンツが設定されます。

```
<%@ taglib uri="/MyTagFiles" prefix="mytags" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

Web アプリケーション間でのタグ・ライブラリの共有

次の各項では、タグ・ライブラリと TLD のパッケージング、格納およびアクセスについて説明します。

- 単一の JAR ファイルへの複数のタグ・ライブラリと TLD のパッケージング
- 予約済のタグ・ライブラリの場所の指定
- TLD キャッシング機能の有効化

単一の JAR ファイルへの複数のタグ・ライブラリと TLD のパッケージング

JSP 仕様では、単一の JAR ファイル内に、複数のタグ・ライブラリおよびタグ・ライブラリを定義する TLD をパッケージングできます。

この項では、複数のタグ・ライブラリを単一の JAR ファイルにパッケージングする例を示します。JAR ファイルには、タグ・ハンドラ・クラス、タグ・ライブラリ・バリデータ（TLV）クラス、および複数のライブラリの TLD が含まれています。

次に、JAR ファイルの内容と構造を示します。複数の TLD が含まれている JAR ファイル内では、TLD は /META-INF ディレクトリまたはサブディレクトリ内に格納されている必要があります。

```
examples/BasicTagParent.class
examples/ExampleLoopTag.class
examples/BasicTagChild.class
examples/BasicTagTLV.class
examples/TagElemFilter.class
examples/TagFilter.class
examples/XMLViewTag.class
examples/XMLViewTagTLV.class
META-INF/xmlview.tld
META-INF/exampletag.tld
META-INF/basic.tld
META-INF/MANIFEST.MF
```

複数の TLD が含まれた JAR ファイルは、/WEB-INF/lib ディレクトリ内、または OC4J の予約済のタグ・ライブラリの場所（7-29 ページの「予約済のタグ・ライブラリの場所の指定」を参照）に格納する必要があります。JSP コンテナは、変換時に、この 2 つの場所で JAR ファイルを検索し、各 JAR ファイルで TLD を検索し、各 TLD にアクセスして <uri> 要素を検索します。

主要な TLD エントリ

各 TLD には、<taglib> 要素内に <uri> 要素があります。この機能は、次のように使用します。

- <uri> 要素に指定する値は、対応するタグ・ライブラリを使用する JSP ページ内で、taglib ディレクティブの uri 設定と一致する値であることが必要です。
- 予期しない結果を回避するために、<uri> の各値は、サーバー上のすべての TLD 内にある <uri> の値の中で一意であることが必要です。

<uri> 要素には任意の値を設定できますが、XML 名前空間の規則に従って設定する必要があります。この値はキーとしてのみ使用され、物理的な場所を示しません。ただし、表記規則上、値は物理的な場所と同じ形式になります。

basic.tld ファイルには次の内容が含まれます。

```
<taglib>

  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>basic</short-name>
  <uri>http://xmlns.oracle.com/j2ee/jsp/tld/demos/basic.tld</uri>

  ...

</taglib>
```

exampletag.tld ファイルには次の内容が含まれます。

```
<taglib>

  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>example</short-name>
  <uri>http://xmlns.oracle.com/j2ee/jsp/tld/demos/exampletag.tld</uri>

  ...

</taglib>
```

xmlview.tld ファイルには次の内容が含まれます。

```
<taglib>
  ...
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>demo</short-name>
  <uri>http://xmlns.oracle.com/j2ee/jsp/tld/demos/xmlview.tld</uri>
  ...
</taglib
```

主要な web.xml デプロイメント・ディスクリプタ・エントリ

この項では、web.xml デプロイメント・ディスクリプタの <taglib> 要素を示します。この要素は、URI のすべての値（前項で説明した TLD の <uri> 要素）を、ライブラリにアクセスする JSP ページで使用するショートカット URI 値にマッピングします。

<taglib> 要素には、2 つのサブ要素を含めることができます。

- <taglib-uri>

  タグを使用する JSP ページ内の taglib ディレクティブの uri 属性の値として使用されるショートカット URI が含まれます。

- <taglib-location>

  タグ・ライブラリの一意の識別子が含まれます。この場合、<taglib-location> 値は実際には場所ではなくキーを示し、任意のタグ・ライブラリの TLD にある <uri> 値に対応します。

各 TLD、または単一のタグ・ライブラリとその TLD が含まれる JAR ファイルを使用する場合、<taglib-location> サブ要素は、TLD またはタグ・ライブラリ JAR ファイルの、アプリケーション相対の物理的な場所（「/」で始まります）を示します。関連情報は、7-29 ページの「予約済のタグ・ライブラリの場所の指定」を参照してください。

複数のタグ・ライブラリとその TLD が含まれる JAR ファイルを使用する場合、<taglib-location> サブ要素は、タグ・ライブラリの一意の識別子を示します。この場合、<taglib-location> 値は実際には場所ではなくキーを示し、任意のタグ・ライブラリの TLD にある <uri> 値に対応します。関連情報は、7-26 ページの「単一の JAR ファイルへの複数のタグ・ライブラリと TLD のパッケージング」を参照してください。

```
<taglib>
  <taglib-uri>/oraloop</taglib-uri>
  <taglib-location>http://xmlns.oracle.com/j2ee/jsp/tld/demos/exampletag.tld
  </taglib-location>
</taglib>
<taglib>
  <taglib-uri>/orabasic</taglib-uri>
  <taglib-location>
     http://xmlns.oracle.com/j2ee/jsp/tld/demos/basic.tld
  </taglib-location>
</taglib>
<taglib>
  <taglib-uri>/oraxmlview</taglib-uri>
  <taglib-location>
     http://xmlns.oracle.com/j2ee/jsp/tld/demos/xmlview.tld
  </taglib-location>
</taglib>
```

複数のライブラリに対する JSP ページの taglib ディレクティブの例

この項では、適切な taglib ディレクティブを示します。このディレクティブは、前項でリストされている web.xml 要素に定義されたショートカット URI 値を参照します。

ページ basic1.jsp には、次のディレクティブが含まれます。

```
<%@ taglib prefix="basic" uri="/orabasic" %>
```

ページ exampletag.jsp には、次のディレクティブが含まれます。

```
<%@ taglib prefix="example" uri="/oraloop" %>
```

ページ xmlview.jsp には、次のディレクティブが含まれます。

```
<%@ taglib prefix="demo" uri="/oraxmlview" %>
```

予約済のタグ・ライブラリの場所の指定

JSP 仕様で規定されている標準的な予約済 URI 機能の拡張機能として、OC4J は、予約済のタグ・ライブラリの場所と呼ばれる 1 つ以上のディレクトリの使用をサポートします。このディレクトリには、複数の Web アプリケーション間で共有するタグ・ライブラリ JAR ファイルを格納できます。

デフォルトの予約済のタグ・ライブラリの場所は、*ORACLE_HOME*/j2ee/home/jsp/lib/taglib/ ディレクトリです。この場所にインストールされたタグ・ライブラリは、デフォルトで、OC4J インスタンスにデプロイされたすべての Web アプリケーションで使用可能になります。

また、共有タグ・ライブラリの場所を追加で定義し、これらのディレクトリに、アプリケーション間で共有するタグ・ライブラリ JAR ファイルをインストールできます。予約済のタグ・ライブラリの場所の定義には、2 つの手順が必要になります。

1. 各ディレクトリを、*ORACLE_HOME*/j2ee/home/config/global-web-application.xml ファイルの <orion-web-app> 要素の jsp-taglib-locations 属性で定義します。それぞれの場所は、セミコロンで区切ります。
2. 各ディレクトリに対し、<library> 要素を *ORACLE_HOME*/j2ee/home/config/application.xml に追加します。これは、default アプリケーションの構成ファイルです。path 属性を、タグ・ライブラリ JAR ファイルが含まれているディレクトリに設定します。

重要： 複数の共有タグ・ライブラリの場所を指定および利用できるかどうかは、<orion-web-app> 要素の jsp-cache-tlds 属性の値によって決定されます。詳細は、7-30 ページの表 7-1「TLD キャッシングのパラメータ」を参照してください。

TLD キャッシング機能の有効化

共有タグ・ライブラリのサポートの一環として、OC4J では TLD の永続的なキャッシング機能を提供しています。これには、すべての予約済のタグ・ライブラリの場所における TLD のグローバル・キャッシュ、および TLD キャッシングを使用するアプリケーションに対するアプリケーション・レベルでのキャッシュが含まれます。

TLD キャッシングの使用により、アプリケーションの起動時や JSP ページ変換時のパフォーマンス速度が向上します。ただし、次の状況では、TLD キャッシングをオフにしてください。

- アプリケーションでタグ・ライブラリが使用されていない。
- JSP ページの事前変換が完了しており、タグ・ライブラリのイベント・リスナーに <listener> 要素を使用している TLD がない。

TLD キャッシングは、<orion-web-app> 要素の jsp-cache-tlds 属性によって有効化または無効化されます。

- グローバル・レベルでは、TLD キャッシングは、グローバル Web アプリケーションの Web 構成ファイルである global-web-application.xml ファイルのこの属性で設定されます。このファイルで設定された値は、OC4J インスタンスにデプロイされた他のすべての Web アプリケーションによって継承されるデフォルト値になります。
- アプリケーション・レベルでは、キャッシングはアプリケーション固有の orion-web.xml ファイルで設定されます。このファイルの設定は、global-web-application.xml のデフォルト設定をオーバーライドします。

次の表で、jsp-cache-tlds 属性の値のサマリーを示します。

表 7-1 TLD キャッシングのパラメータ

jsp-cache-tlds の値	global-web-application.xml で設定される値
standard	TLD キャッシングは有効です。これは、グローバル・レベルおよびアプリケーション・レベルでのデフォルト設定です。 タグ・ライブラリ JAR をデフォルトの予約済のタグ・ライブラリの場所である *ORACLE_HOME*/j2ee/home/jsp/lib/taglib に追加します。タグ・ライブラリはすべての Web アプリケーションで使用可能になります。 TLD（*.tld）は、/WEB-INF ディレクトリに格納する必要があることに注意してください。TLD を /classes または /lib サブディレクトリに含めないでください。
on	TLD キャッシングは有効です。 タグ・ライブラリが含まれている JAR ファイルを次のいずれかに追加します。 ■ デフォルトの予約済のタグ・ライブラリの場所である *ORACLE_HOME*/j2ee/home/jsp/lib/taglib ディレクトリ ■ global-web-application.xml の <orion-web-app> 要素の jsp-taglib-locations 属性で指定された追加の場所 追加の場所の指定方法の詳細は、7-29 ページの「予約済のタグ・ライブラリの場所の指定」を参照してください。
off	TLD キャッシングは無効です。 タグ・ライブラリ JAR をデフォルトの予約済のタグ・ライブラリの場所である *ORACLE_HOME*/j2ee/home/jsp/lib/taglib のみに追加します。 異なる場所を、*ORACLE_HOME*/j2ee/home/config/global-web-application.xml の well_known_taglib_loc 初期化パラメータの値として指定できます。 初期化パラメータは、このファイルの <servlet>jsp</servlet> 表示の <init-param> サブ要素内で指定されます。詳細は、3-8 ページの「XML 構成ファイルでの JSP パラメータの設定」を参照してください。

重要：

- TLD が、予約済の場所とアプリケーションの /WEB-INF ディレクトリ下の両方にある場合、/WEB-INF のコピーが優先され、使用されます。
- URI 値が同じ TLD が、/WEB-INF ディレクトリと、/WEB-INF/lib ディレクトリの JAR ファイルにもある場合、どちらが使用されるかは確定できません。このような状況は避けてください。

TLD キャッシュ機能とファイルについて

TLD キャッシングを使用するアプリケーションの場合、グローバル・レベルまたはアプリケーション・レベルで有効になっているかどうかに関係なく、2 つのレベルのキャッシングがあり、各レベルのキャッシングには 2 つの機能があります。

キャッシング・レベル :

- TLD のグローバル・キャッシュがあります。これは、予約済のタグ・ライブラリの場所の JAR ファイルにあります。
- TLD のアプリケーション・レベルのキャッシュが、アプリケーションの `/WEB-INF` ディレクトリにあります。

  アプリケーション・レベルでは、タグ・ライブラリ JAR ファイルには TLD が含まれており、`/WEB-INF/lib` ディレクトリ内に格納する必要があります。

  個別の TLD は、`/WEB-INF` 内のディレクトリに直接、または任意のサブディレクトリに格納できますが、できれば、`/WEB-INF/lib` または `/WEB-INF/classes` には格納しないでください。`<orion-web-app>` 要素の `jsp-cache-tlds` 属性が `standard` に設定されている場合、TLD を `/WEB-INF/lib` または `/WEB-INF/classes` のいずれにも格納しないでください。

各レベルのキャッシング機能 :

- 関係する場所のリソース情報を含むファイル。グローバル・キャッシュの場合は予約済の場所、アプリケーション・レベル・キャッシュの場合は `/WEB-INF` または `/WEB-INF/lib` です。この機能によって、JAR ファイルを複数回スキャンする必要はありません。JAR ファイルには、2 つのタイプのエントリがあります。
  - リソースごとのタイムスタンプを含む全リソース（タグ・ライブラリ JAR ファイル）のリスト。これにより、リソースの変更が検出されます。また、各リソースに TLD が含まれているかどうかが、`true` または `false` でわかります。
  - TLD のリスト。リストの各エントリは、TLD 名、TLD URI 値（ある場合）およびライブラリ・リスナー（ある場合）で構成されます。
- 各 TLD のシリアライズ化された DOM 表現。この機能によって、TLD を複数回解析する必要はありません。

グローバル・キャッシュは、構成ディレクトリと並列の、`tldcache` というディレクトリに常に格納されます。`tldcache` ディレクトリには、次のものが含まれます。

- `_GlobalTldCache` ファイル。前述のように、予約済の場所に関するリソース情報が含まれています。
- 予約済の場所にある TLD の DOM 表現。予約済の場所の JAR ファイルに格納されている TLD ごとに、TLD の名前に基づいたファイル名とともに、JAR ファイルの名前に従って、サブディレクトリ内に DOM 表現が格納されます。たとえば、予約済の場所の `ojsputil.jar` に `email.tld` がある場合、DOM 表現は次のファイル（ディレクトリ `ojsputil_jar` のファイル名 `email`）に格納されます。

  ORACLE_HOME`/j2ee/home/jsp/lib/taglib/persistence/ojsputil_jar/email`

  これは、Oracle Application Server 環境で、*ORACLE_HOME* が定義されている場合です。スタンドアロン OC4J では、`j2ee` ディレクトリは、OC4J がインストールされた場所に対する相対パスになります。

アプリケーション・レベルのキャッシュは、global-web-application.xml または orion-web.xml の jsp-cache-directory 設定で指定されたディレクトリに格納されます。このディレクトリには、次のものが含まれます。

- _TldCache ファイル。前述のように、/WEB-INF ディレクトリの TLD に関するリソース情報が含まれています。/WEB-INF/lib の JAR ファイル内、または /WEB-INF や任意のサブディレクトリに個別に格納されますが、できれば、/WEB-INF/lib または /WEB-INF/classes は使用しないでください。

  <orion-web-app> 要素の jsp-cache-tlds 属性が standard に設定されている場合、TLD を /WEB-INF/lib または /WEB-INF/classes のいずれにも格納しないでください。

- /WEB-INF の TLD の DOM 表現。/WEB-INF/lib ディレクトリの JAR ファイルに格納されている TLD の場合、DOM 表現は、グローバル・キャッシュについて説明したものと同じスキーム・タイプで、jsp-cache-directory で指定されたディレクトリのサブディレクトリに格納されます。/WEB-INF の個別の TLD の場合、DOM 表現は、jsp-cache-directory の場所に直接格納されます。

注意：

- グローバル・レベルでの TLD の変更は、OC4J を再起動した後にのみ反映されます。
- アプリケーション・レベルでの TLD の変更は、スタンドアロン OC4J 環境ではただちに反映されますが、Oracle Application Server 環境では、アプリケーションを再起動した後にのみ反映されます。
- OC4J の冗長レベルを大きくすると、TLD キャッシュの構成、および重複している TLD URI に関する情報を確認できます。レベル 4 では一部の情報が提供され、レベル 5 では追加情報が提供されます。デフォルトのレベルは 3 です。

8

OC4J における JSP XML サポート

JavaServer Pages テクノロジは、XML 文書の作成に効果的なモデルであるとの認識が高まっています。JSP 2.0 でリリースされた、XML 構文の改善を含めた新しい拡張機能によって、JSP テクノロジは XML テクノロジをさらに補完するものとなり、XML ツールへのアクセスもより容易になりました。

OC4J JSP コンテナは、JavaServer Pages 2.0 仕様で説明されている JavaServer Pages による XML のサポートを完全に実装しています。これには、次のサポートが含まれます。

- JSP の構文要素に対応する XML スタイル
- JSP ページの XML ビューの概要
- すべての XML 要素をドキュメント・ルートとして使用可能（`<jsp:root>` 要素は不要）
- 新しい `<jsp:element>` および `<jsp:attribute>` 標準タグ要素のサポート

この章には、次の各項が含まれます。

- JSP ドキュメントと XML ビューの概要
- JSP ドキュメントの使用
- JSP XML ビューについて

カスタム・タグを介して OC4J で提供される XML と XSL の JSP サポートの詳細は、『Oracle Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

XML に関する一般情報は、次の Web サイトで XML 仕様を参照してください。

`http://www.w3.org/XML/`

JSP ドキュメントと XML ビューの概要

JSP ドキュメントという用語は、XML 構文で作成された JSP ページを指します。JSP ドキュメントは、純粋な XML 構文内で適切に構成されており、名前空間を識別します。JSP ドキュメントでは、JSP XML コア構文および使用する標準アクション・タグおよびカスタム・タグ・ライブラリの構文を指定するために、XML 名前空間が使用されます。対照的に、従来の JSP ページは通常、XML 文書ではありません。

JSP ドキュメントには、次のような利点があります。

- 既存の XML ツールと互換性のある純粋な XML 構文で、OC4J JSP コンテナに直接渡せる動的なドキュメントを作成できます。
- XHTML または SVG のような XML ベース言語で作成されたコンテンツを JSP ページで出力できます。
- XSLT などの XML 変換を適用することにより、テキスト表現から JSP ドキュメントを生成できます。
- 異なる Web アプリケーション間でのデータ交換に JSP ドキュメントを使用できます。

標準アクション・タグやカスタム・タグなどの多くの JSP 構文の要素は、すでに XML 構文で作成されています。互換性のない JSP 要素に対しては、対応する XML 要素が提供されています。詳細は、8-3 ページの「JSP ドキュメントの使用」を参照してください。

OC4J の JSP コンテナは、次のいずれかを検出すると、そのファイルを JSP ドキュメントとして識別します。

- Web モジュールの `web.xml` デプロイメント・ディスクリプタ内の `<is-xml>` 要素

  `<is-xml>` は、ファイルを JSP ドキュメントとして識別するために使用される、`<jsp-property-group>` 要素のサブ要素です。たとえば、この `web.xml` というコードでは、`.svg` 拡張子を持つファイル（`<url-pattern>` サブ要素で指定）を JSP ドキュメントとして識別します。

  ```
  <web-app ...>
   <jsp-config>
    <jsp-property-group>
     <description>Define files with SVG extension as JSP documents</description>
     <url-pattern>*.svg</url-pattern>
     <is-xml>true</is-xml>
    </jsp-property-group>
   </jsp-config>
  </web-app>
  ```

  `<is-xml>` 定義は、次に説明するすべてのインジケータをオーバーライドします。

- `.jspx` ファイル拡張子

  この拡張子は JSP 2.0 で新しくサポートされるようになりました。これは明示的にファイルを JSP ドキュメントとして定義します。タグ・ファイルも XML 構文で作成することが可能で、`.tagx` 拡張子によってコンテナに識別されます。タグ・ファイルの詳細は、7-20 ページの「タグ・ファイルとは」を参照してください。

- ドキュメント・ボディ内のトップ・レベル要素としての `<jsp:root>` 要素

  この要素には、JSP XML コア構文用の名前空間仕様、および使用するカスタム・タグ・ライブラリ用の名前空間仕様が含まれます。この要素は JSP 1.2 では必須でしたが、JSP 2.0 ではオプションになったため、ユーザーが自分でルート要素を指定できます。

JSP ドキュメントに対するセマンティック・モデルは従来のページと同様です。JSP ドキュメントは、対応する構文を使用した従来のページと同じ一連の操作と結果を指示します。空白の処理は XSLT 規則に従います。JSP ドキュメントのノードが識別されると、空白のみのテキスト・ノードは、テンプレート・データ用の `<jsp:text>` 要素内である場合を除き、文書から削除されます。`<jsp:text>` 要素の内容は、そのまま保持されます。

注意： テンプレート・データは、JSP トランスレータでは解析できないテキストで構成されています。

XML ビューは、JSP ページから導出される XML ドキュメントで、ページの妥当性チェックに使用されます。XML ビューは、JSP 変換時に OC4J JSP コンテナによって生成されます。JSP 2.0 仕様の中で、XML ビューは「XML 構文または従来の構文で記述された JSP ページと、それを説明する XML 文書との間のマッピング」として定義されています。

JSP 仕様 1.2 以降、すべてのタグ・ライブラリがその TLD 内に `<validator>` 要素を保持し、妥当性チェックを実行できるクラスを指定できます。このようなクラスは、タグ・ライブラリ・バリデータ（TLV）・クラスと呼ばれます。TLV クラスの目的は、タグ・ライブラリを使用する JSP ページが、実装した必要な制約に準拠しているかどうか検証することです。バリデータ・クラスは、その妥当性チェックのソースとして JSP XML ビューを使用します。

JSP ドキュメントの場合、JSP XML ビューはページ・ソースとほぼ同じです。相違点は、XML ビューが `include` ディレクティブに従って拡張される点です。他の相違点は、改善されたエラー・レポートに関する ID 属性がすべての XML 要素に追加される点です。

従来の JSP ページの場合、Web コンテナは、一連の変換を実行してページから XML ビューを作成します。詳細は、8-10 ページの「JSP XML ビューについて」を参照してください。

つまり、JSP XML 構文を必要に応じて使用して、XML 互換の JSP ページを作成できます。対照的に、JSP XML ビューは Web コンテナの機能で、ページの妥当性チェックに使用されます。

JSP ドキュメントの使用

ここでは、JSP ドキュメントの構文について、より詳しく説明します。全体の解説は、Sun 社の JSP 仕様を参照してください。

重要： JSP の従来の構文と JSP XML 構文を 1 つのファイルに混在させることはできません。ただし、`include` ディレクティブを使用すると、1 つの変換単位で両方の構文を使用できます。たとえば、従来の JSP ページに JSP ドキュメントを含めることができます。

JSP XML 構文には、次の要素を含めることができます。この構文には、`<%-- comment --%>` 以外のすべての JSP 要素について、XML の対応する要素または置換要素が含まれています。

- `<jsp:root ...>` 要素（オプション）: JSP XML コア構文用の名前空間仕様、および使用するカスタム・タグ・ライブラリ用の名前空間仕様が含まれます。この要素は JSP 2.0 ではオプションのため、ユーザーが自分でルート要素を指定できます。
- JSP の `page` ディレクティブおよび `include` ディレクティブ
- JSP の宣言要素
- JSP の式要素
- JSP のスクリプトレット要素
- JSP の標準アクションの要素
- JavaServer Pages 標準タグ・ライブラリ（JSTL）のタグ要素を含めた JSP カスタム・アクション要素
- 新しい `<jsp:attribute>` および `<jsp:element>` 要素 : 要素のボディおよび属性値に式言語（EL）式を含めることができます。
- テキスト要素 `<jsp:text>`: 静的なテンプレート・データ用
- テンプレート・データに関するその他の XML 要素（必要な場合）

前述のとおり、標準の JSP 構文の大半はすでに XML 互換のため、XML 要素形式で JSP ドキュメント内で使用できます。互換性のない JSP 要素に対しては、次の表 8-1 で示すとおり、対応する XML 要素のセットが提供されています。

表 8-1 標準の JSP 構文と XML 構文

構文要素	JSP 構文	XML 構文
コメント	`<%--.. --%>`	`<!-- .. -->`
宣言	`<%! ..%>`	`<jsp:declaration> ..</jsp:declaration>`
include ディレクティブ	`<%@ include .. %>`	`<jsp:directive.include .. />`
page ディレクティブ	`<%@ page .. %>`	`<jsp:directive.page .. />`
タグ・ライブラリ・ディレクティブ	`<%@ taglib .. %>`	`xmlns:prefix="tag library URL"`
式	`<%= .. %>`	`<jsp:expression> ..</jsp:expression>`
スクリプトレット	`<% .. %>`	`<jsp:scriplet> ..</jsp:scriplet>`

ドキュメントのルート要素の指定

JSP ドキュメントにはルート要素が必要です。JSP 2.0 より前のリリースでは、`<jsp:root>` 要素をルートとして使用する必要がありました。この要素はまだ使用できますが、必須ではなくなりました。したがって、ユーザーが自分でルート要素を指定できます。この柔軟性により、任意の XML 文書を JSP ドキュメントとして使用することが可能になります。

後述のドキュメントの例では、ルート要素は `html` です。また、次の項で説明するとおり、ルート内で、ドキュメントで使用するタグ・ライブラリを宣言できます。

JSP ドキュメントおよびそのリクエスト出力には、XML 宣言を含める必要はありません。それどころか、JSP ドキュメントが XML 出力を生成しない場合、ドキュメントに XML 宣言を含めることができません。

XML 名前空間を使用したタグ・ライブラリの宣言

JSP ドキュメントでは、ドキュメント内に標準アクション・タグおよびカスタム・タグを含めるために、XML 名前空間を使用します。XML 名前空間は、XML の要素タイプと属性名のコレクションです。

JSP ページと異なり、<jsp:useBean> などの標準アクション・タグは暗黙的に使用できません。かわりに、これらの要素に続けて、コア JSP XML 構文の名前空間を識別する名前空間を含める必要があります。

```
xmlns:jsp="http://java.sun.com/JSP/Page"
```

また、使用する各カスタム・タグ・ライブラリ用の `xmlns` 属性を含めて、タグ・ライブラリの接頭辞と名前領域を指定する必要があります。つまり、タグ使用の妥当性チェックに使用するための対応する TLD またはタグ・ファイル・ライブラリをポイントする必要があります。これらの `xmlns` 設定は、従来の JSP ページの `taglib` ディレクティブと等価です。

タグ・ライブラリをポイントするには、URN または URI のいずれかを使用できます。次の例では、JSTL の接頭辞 `c` とカスタム・タグの接頭辞 `my` の名前空間を `html` ルート要素で宣言しています。

```
<html     xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:c="http://java.sun.com/jsp/jstl/core"
          xmlns:my="urn:jsptagdir:/WEB-INF/tlds/mylib"
>

...body of document...

</html>
```

URN はアプリケーション相対パスを示し、次のいずれかの形式を使用する必要があります。

- "urn:jsptld:*path*"。パスは 1 つのタグ・ライブラリをポイントします。taglib ディレクティブの uri 属性の場合と同様です。
- "urn:jsptagdir:path"。パスは /WEB-INF/tags/ で始まり、WEB-INF/tags/ ディレクトリまたはそのサブディレクトリの 1 つにインストールされたタグ・ファイルとして実装されているタグ拡張子をポイントします。

URI は完全な URL の場合があります。そうでない場合は、web.xml ファイルの <taglib> 要素、または WEB-INF/lib の JAR ファイル内か、WEB-INF 内の TLD の <uri> 要素のマッピングに従います。7-28 ページの「主要な web.xml デプロイメント・ディスクリプタ・エントリ」および 7-26 ページの「単一の JAR ファイルへの複数のタグ・ライブラリと TLD のパッケージング」を参照してください。

前述の例のようにタグ・ライブラリをルート要素の xmlns 属性として宣言すると、ドキュメント内のすべての要素がそれらのライブラリにアクセスできます。ただし、これは必須ではありません。タグ・ライブラリをドキュメント内の使用箇所で参照し、タグ・ライブラリを特定の要素から使用するように有効範囲を設定できます。

たとえば、次の例では、c 接頭辞によって参照されるタグ・ライブラリの使用を <c:forEach 要素に制限しています。

```
<c:forEach xmlns:c="http://java.sun.com/jsp/jstl/core"
var="counter" begin="0" end=${4}>
...
</c:forEach>
```

JSP XML ディレクティブ要素の使用

page ディレクティブと include ディレクティブに対応する JSP XML 要素があります。page ディレクティブまたは include ディレクティブを、対応する JSP XML 要素に変換するのは簡単な作業です。次に例を示します。

例 : page ディレクティブ

次に page ディレクティブの例を示します。

```
<%@ page language="java"
    import="com.tks.ourpackage" %>
```

これは、次の JSP XML 要素に対応します。

```
<jsp:directive.page language="java"
    import="com.tks.ourpackage" />
```

例 : include ディレクティブ

次に include ディレクティブの例を示します。

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

これは、次の JSP XML 要素に対応します。

```
<jsp:directive.include file="/jsp/userinfopage.jsp" />
```

注意 : ページの XML ビューには、include 要素がありません。これは、静的にインクルードされたセグメントはビューに直接コピーされるためです。

JSP XML の宣言要素、式要素およびスクリプトレット要素の使用

JSP の宣言、式およびスクリプトレットに対応する JSP XML 要素があります。これらの構成を対応する JSP XML 要素に変換するのは簡単な作業です。次に例を示します。

例 : JSP の宣言

次に JSP の宣言例を示します。

```
<%! public String func(int myint) { if (myint<0) return("..."); } %>
```

これは、次の JSP XML 要素に対応します。

```
<jsp:declaration>
  <![CDATA[ public String func(int myint) { if (myint<0) return("..."); } ]]>
</jsp:declaration>
```

XML の CDATA（文字データ）の指示が使用されているのは、宣言に「<」文字が含まれているためです。この文字は XML パーサーに対して特別な意味を持ちます。(JSP XML ページの作成に XML エディタを使用すると、この文字は自動的に処理されます。)「<」のかわりに「<」エスケープ文字を使用して、次のように記述することもできます。

```
<jsp:declaration>
   public String func(int myint) { if (myint &lt; 0) return("..."); }
</jsp:declaration>
```

例 : JSP の式

次に JSP の式の例を示します。

```
<%= (user==null) ? "" : user %>
```

これは、次の JSP XML 要素に対応します。

```
<jsp:expression> (user==null) ? "" : user </jsp:expression>
```

例 : JSP のスクリプトレット

次に JSP のスクリプトレットの例を示します。

```
<% if (pageBean.getNewName().equals("")) { %>
   ...
```

これは、次の JSP XML 要素に対応します。

```
<jsp:scriptlet> if (pageBean.getNewName().equals("")) { </jsp:scriptlet>
   ...
```

JSP XML の標準アクションとカスタム・アクションの要素の使用

JSP の標準アクション（jsp:include、jsp:forward および jsp:useBean など）とカスタム・アクションに対する従来の構文は、すでに XML 互換です。ただし、JSP XML 構文で標準アクションやカスタム・アクションを実行する場合は、次の点に注意してください。

- リクエスト時の式の値を受け入れる属性を持つ標準アクションやカスタム・アクションの要素は、次の構文を使用してその値を取得できます。

  "%=*expression*%"

  この構文の前後には、「<」および「>」の山カッコがなく、*expression* の前後の空白は不要であることに注意してください。XML 文書の場合と同様に、規定の引用符の後にある *expression* の値は、JSP のリクエスト時の式の場合と同じです。

- 引用符は、XML 仕様に従う必要があります。

- テンプレート・データは、<jsp:text> 要素または選択した XML 要素（標準でもカスタムでもない要素）を使用して導入できます。詳細は、次項の「テンプレートおよび動的なテンプレート・コンテンツの含有」を参照してください。

テンプレートおよび動的なテンプレート・コンテンツの含有

JSP ドキュメントに静的なテンプレート・テキストを含めるには、空白を保持しない未解析の XML タグ、または JSP ドキュメントでテンプレート・データを示す <jsp:text> 要素を使用します。

<jsp:text> 要素を検出した OC4J JSP コンテナは、その内容を現行の JSP の out オブジェクトに渡します（XSLT の <xsl:text> 要素の処理とほぼ同じです）。

JSP 仕様では、<jsp:text> 要素が使用されている場所で、テンプレート・データ用に任意の要素（標準アクションの要素やカスタム・アクションの要素以外）を使用できます。これら任意の要素は、現行の JSP の out オブジェクトに送信されるコンテンツを使用し、<jsp:text> 要素と同じ方法で処理されます。

次のような JSP ドキュメントのソース・テキストについて考えてみます。

```
<hello><jsp:declaration>String n="Alfred";</jsp:declaration>
<morning>
<jsp:text> Good Morning
</jsp:text>${n}
</morning>
</hello>
```

このソース・テキストによって、次の内容が OC4J の JSP コンテナから出力されます。空白が保持されている点に注意してください。

```
<hello> <morning> Good Morning
Alfred </morning></hello>
```

適切に構成された XML ではないテンプレート・データの出力には、<jsp:text> も使用できます。たとえば、もし次の ${list}EL 式が <jsp:text> タグで囲まれていなければ、JSP ドキュメントでは無効となります。

```
<c:forEach var="list" begin="1" end="${4}">
  <jsp:text>${list}</jsp:text>
</c:forEach>
```

動的なテンプレート・コンテンツも、標準の JSP ページと同様に、EL 式、Java スクリプト要素（宣言、スクリプトレットまたは式）、標準アクション・タグおよびカスタム・タグを使用して JSP ドキュメントによって生成できます。

EL 式を <jsp:element> 標準アクションとともに JSP ドキュメントで使用すると、ハードコードせずにタグを動的に生成できます。このアクションは 1 つの String 属性を取り、これは生成された要素の名前として使用されます。

タグ要素には、オプションでボディを含めることができます。これは、次のもので構成されます。

- 属性を定義しないボディ
- 新しい要素の属性として設定される 1 つ以上の <jsp:attribute> 要素
- <jsp:attribute> を使用している場合、タグ・ボディを <jsp:body> 要素内に含めることが可能

次に、JSP ドキュメントのコードの例を示します。

```
<jsp:element name="${searchRequest.type}">
   <jsp:attribute name="language">${searchRequest.language}</jsp:attribute>
   <jsp:body>${searchRequest.content}</jsp:body>
</jsp:element>
```

これにより、次のものが生成されます。

```
<standardSearch language="English">What is an attribute?</standardSearch>
```

比較サンプル : 従来の JSP ページと JSP XML 文書との比較

ここでは、2 つのバージョン（従来の構文と XML 構文）の JSP ページを示します。

このサンプルのデプロイと実行の詳細は、次の Web サイトを参照してください。

http://otn.oracle.com/tech/java/oc4j/htdocs/how-to-jsp-xmlview.html

（この Web サイトを参照するには、OTN に登録する必要があります。無償で登録できます。）

従来の JSP ページのサンプル

従来の構文によるサンプル・ページを示します。

```
<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
<html>
<head>
<title>An eStore for all occasions</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body>
<table align="center" width="100%" height="100%" border="4" bgcolor="#FFFFFF"
 bordercolor="#FF0000">
 <tr>
  <td width="64%" valign="middle" align="center" bgcolor="#FFCCCC">
   <div align="center"><img src="gifs/christmas/Electronic.gif"></div>
   <tags:ProductDetails occasion="Christmas" category="Cards" thBgColor="#FF3366"
    thFontColor="#FFFFFF">
    <jsp:attribute name="normalPrice">
    <td width="50%" align="center"><font
     color="#CC3300"><b>${name}</b></font></td>
    <td width="30%" align="center"><font
     color="#CC3300"><b>${price}</b></font></td>
    <td width="20%" align="center"><img src=${image} width="100" height="40"></td>
  </tr>
  </jsp:attribute>
  <jsp:attribute name="onSale">
 <tr>
    <td width="50%" align="center"><font color="#CC3300"><b>${name}<img
     src="gifs/main/discount.gif"></b></font></td>
     <td width="30%" align="center"><font color="red"><b>
      <strike>Was:  ${price}</strike></b></font><br><font color="#996600"><b>
      <font color="#CC3300">Now: ${saleprice}</font></b></font>
     </td>
     <td width="20%" align="center"><img src=${image} width="100"
      height="40"></td>
    </tr>
```

```
    </jsp:attribute>
   </tags:ProductDetails>
 <br><br>
  </td>
 </tr>
</table>
</body>
</html>
```

JSP ドキュメントのサンプル

同じページを XML 構文で示します。ブラウザに渡される HTML が <jsp:text> タグで囲まれている点に注意してください。

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:tags="/WEB-INF/tags">
<jsp:directive.page contentType="text/html" />
<jsp:text>
<![CDATA[
<head><title>An eStore for all occasions</title></head>
<body>
<table align="center" width="100%" height="100%" border="4" bgcolor="#FFFFFF"
  bordercolor="#FF0000">
 <tr>
  <td width="64%" valign="middle" align="center" bgcolor="#FFCCCC">
   <div align="center"><img src="gifs/christmas/Electronic.gif"></div>
   ]]>
   </jsp:text>
   <tags:ProductDetails occasion="Christmas" category="Cards" thBgColor="#FF3366"
  thFontColor="#FFFFFF">
    <jsp:attribute name="normalPrice">
    <jsp:text>
    <![CDATA[<tr>
    <td width="50%" align="center"><font
     color="#CC3300"><b>${name}</b></font></td>
    <td width="30%" align="center"><font
     color="#CC3300"><b>${price}</b></font></td>
    <td width="20%" align="center"><img src=${image} width="100" height="40"></td>
  </tr>]]>
  </jsp:text>
  </jsp:attribute>
  <jsp:attribute name="onSale">
  <jsp:text>
  <![CDATA[<tr>
    <td width="50%" align="center"><font color="#CC3300"><b>${name}<img
     src="gifs/main/discount.gif"></b></font></td>
     <td width="30%" align="center"><font color="red"><b>
      <strike>Was:  ${price}</strike></b></font><br><font color="#996600"><b>
      <font color="#CC3300">Now: ${saleprice}</font></b></font>
     </td>
     <td width="20%" align="center"><img src=${image} width="100"
      height="40"></td>
    </tr>]]>
    </jsp:text>
    </jsp:attribute>
   </tags:ProductDetails>
  <jsp:text>
  <![CDATA[<br><br>
  </td>
 </tr>
</table>
</body>]]>
</jsp:text>
</html>
```

JSP XML ビューについて

OC4J で JSP ページを変換すると、XML ビューと呼ばれる、解析結果の XML バージョンが作成されます。JSP 仕様では、XML ビューを、JSP ページ（従来のページまたは JSP ドキュメントのいずれか）から、そのページについての記述がある XML 文書へのマッピングとして定義しています。

ページの XML ビューは、JSP XML 構文を使用してユーザー自身が記述した場合のページに類似していますが、主な相違点がいくつかあります。これについて簡単に説明します。

この項には、次の項目が含まれます。

- JSP ページから XML ビューへの変換
- 妥当性チェックにおけるエラー・レポートの jsp:id 属性
- 例 : 従来の JSP ページから XML ビューへの変換

詳細は、Sun 社の JSP 仕様を参照してください。

JSP ページから XML ビューへの変換

JSP ページを変換する場合、Web コンテナは、XML ビューを作成する際に従来の JSP ページと JSP XML 文書の両方に対して、次の変換を実行します。

- コンテナは、XML ビューを拡張して、`include` ディレクティブを使用して移入したファイルをインクルードします。
- オプションの `jsp:id` 属性（改善されたエラー・レポート用）をサポートしている Web コンテナは、その属性をページ内の各 XML 要素に挿入します。8-11 ページの「妥当性チェックにおけるエラー・レポートの jsp:id 属性」を参照してください。

JSP ドキュメントの場合は、これらの点が、XML ビューと元のページとの主な差異の原因となります。

Web コンテナは、従来の JSP ページに対して、次の追加変換を実行します。

- JSP XML 構文の場合は標準の `xmlns` 属性設定で、JSP バージョンの場合は `version` 属性で、`<jsp:root>` 要素を追加します。8-4 ページの「XML 名前空間を使用したタグ・ライブラリの宣言」を参照してください。
- 各 `taglib` ディレクティブを `<jsp:root>` 要素の追加の `xmlns` 属性に変換します。8-4 ページの「XML 名前空間を使用したタグ・ライブラリの宣言」を参照してください。
- 各 `page` ディレクティブを JSP XML 構文の対応する要素に変換します。8-5 ページの「JSP XML ディレクティブ要素の使用」を参照してください。
- 宣言、式、スクリプトレットを、それぞれ JSP XML 構文の対応する要素に変換します。8-6 ページの「JSP XML の宣言要素、式要素およびスクリプトレット要素の使用」を参照してください。
- リクエスト時の式を XML 構文に変換します。8-7 ページの「JSP XML の標準アクションとカスタム・アクションの要素の使用」を参照してください。
- テンプレート・データの `<jsp:text>` 要素を作成します。8-7 ページの「テンプレートおよび動的なテンプレート・コンテンツの含有」を参照してください。
- JSP の引用符を XML の引用符に変換します。
- JSP のコメント（`<%-- comment --%>`）を無視します。これらのコメントは XML ビューに表示されません。

注意 :

- XML ビューに DOCTYPE 文はありません。
- 8-7 ページの「テンプレートおよび動的なテンプレート・コンテンツの含有」に説明されているように、その他の XML 要素は XML ビューに表示されません。テンプレート・データで使用されるのは、<jsp:text> 要素のみです。

妥当性チェックにおけるエラー・レポートの jsp:id 属性

JSP 仕様には、Web コンテナが XML ビューの各 XML 要素に追加できる jsp:id 属性についての説明があります。jsp:id 属性は、ページの妥当性チェック時にタグ・ライブラリ・バリデータ・クラスによって使用されます。この属性の目的は、改善されたエラー・レポートを提供することです。これは、(Web コンテナによる jsp:id サポートの実装方法に従って) エラーの発生場所を開発者に正確に示すのに役立ちます。

jsp:id 属性の値は、それぞれの値または ID が、XML ビューのすべての要素間で一意となるように、コンテナによって生成される必要があります。

タグ・ライブラリ・バリデータ・オブジェクトは、戻された ValidationMessage オブジェクトで、これらの ID を使用できます。OC4J の JSP 実装では、ID を持つ ValidationMessage オブジェクトが戻されると、各 ID は、一致している要素のタグ名とソース位置を反映するために変換されます。

例 : 従来の JSP ページから XML ビューへの変換

この例では、従来のページ・ソースを示し、次に、OC4J の JSP トランスレータが生成するページの XML ビューを示します。このコードでは、Oracle JSP のバージョン番号と構成パラメータの値が表示されています。

従来の JSP ページ

従来の JSP ページを示します。

```
<%@ page import="java.util.*" %>
<HTML>
    <HEAD>
        <TITLE>JSP Information </TITLE>
    </HEAD>
    <BODY BGCOLOR="#FFFFFF">
         JSP Version:<BR>
            <%= application.getAttribute("oracle.jsp.versionNumber") %>
         <BR>
         JSP Init Parameters:<BR>
         <%
         for (Enumeration paraNames = config.getInitParameterNames();
              paraNames.hasMoreElements() ;) {
             String paraName = (String)paraNames.nextElement();
         %>
         <%=paraName%> = <%=config.getInitParameter(paraName)%>
         <BR>
         <%
         }
         %>
    </BODY>
</HTML>
```

JSP ページの XML ビュー

対応する XML ビューを示します。

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" jsp:id="0" version="1.2">
   <jsp:text jsp:id=""><![CDATA[   <HTML>
       <HEAD>
           <TITLE>JSP Information </TITLE>
       </HEAD>
       <BODY BGCOLOR="#FFFFFF">
            JSP Version:<BR>]]></jsp:text>
   <jsp:expression jsp:id="2">
      <![CDATA[ application.getAttribute("oracle.jsp.versionNumber") ]]>
   </jsp:expression>
   <jsp:text jsp:id="3"><![CDATA[
            <BR>
            JSP Init Parameters:<BR>
            ]]>
   </jsp:text>
   <jsp:scriptlet jsp:id="4"><![CDATA[
            for (Enumeration paraNames = config.getInitParameterNames();
                 paraNames.hasMoreElements() ;) {
                String paraName = (String)paraNames.nextElement();
            ]]></jsp:scriptlet>
   <jsp:text jsp:id="5"><![CDATA[
            ]]></jsp:text>
   <jsp:expression jsp:id="6"><![CDATA[paraName]]></jsp:expression>
   <jsp:text jsp:id="7"><![CDATA[ = ]]></jsp:text>
   <jsp:expression jsp:id="8">
      <![CDATA[config.getInitParameter(paraName)]]>
   </jsp:expression>
   <jsp:text jsp:id="9"><![CDATA[
            <BR>
            ]]></jsp:text>
   <jsp:scriptlet jsp:id="0"><![CDATA[
            }
            ]]></jsp:scriptlet>
   <jsp:text jsp:id=""><![CDATA[
       </BODY>
  </HTML>

]]></jsp:text>
</jsp:root>
```

9

Oracle での JSP グローバリゼーション・サポート

OC4J の Web コンテナは、JSP 仕様に従って、標準のグローバリゼーション・サポート（National Language Support（NLS）とも呼ばれます）を提供し、マルチバイト・パラメータ・エンコードをサポートしないサーブレット環境の拡張サポートも提供します。

ローカライズされたコンテンツの標準の Java サポートでは、テキストの内部表現を統一するため、Unicode を使用する必要があります。Unicode は、別のキャラクタ・セットに変換するためのベース・キャラクタ・セットとして使用されます。（Unicode のバージョンは、JDK のバージョンによって異なります。Sun 社の Javadoc で `java.lang.Character` クラスの Unicode のバージョンを確認できます。）

この章では、グローバリゼーションと国際化に対する JSP サポートについて、重要な事項を説明します。次の項目について説明します。

- コンテンツ・タイプの設定
- マルチバイト・パラメータ・エンコードに対する JSP サポート

> **注意：** Oracle Application Server のグローバリゼーション・サポートの詳細は、『Oracle Application Server グローバリゼーション・ガイド』を参照してください。

コンテンツ・タイプの設定

次の各項では、コンテンツ・タイプを JSP ページに静的または動的に指定する標準的な方法について説明します。また、IANA（Internet Assigned Numbers Authority）以外のキャラクタ・セットを JSP ライター・オブジェクトに対して指定できる Oracle の拡張メソッドについても説明します。

page ディレクティブでのコンテンツ・タイプの設定

page ディレクティブには、JSP のページ・ソース（変換時）またはレスポンス（実行時）の文字エンコードに影響を与える 2 つの属性（pageEncoding と contentType）があります。contentType 属性は、MIME タイプのレスポンスにも影響を与えます。各属性の機能は、次のとおりです。

- contentType を使用すると、ページ・ソースとレスポンス、および MIME タイプのレスポンスの文字エンコードを設定できます。
- pageEncoding を使用すると、ページ・ソースの文字エンコードを設定できます。この属性によって、レスポンスとページ・ソースで異なる文字エンコードを設定できます。ただし、キャラクタ・セットを指定する contentType 属性がない場合、この設定はレスポンスの文字エンコードに対するデフォルトになります。

contentType と pageEncoding の関連の詳細は、この項で後述します。

contentType には、次の構文を使用します。

```
contentType="TYPE; charset=character_set"
```

または、デフォルトのキャラクタ・セットを使用する場合は、次のように、MIME タイプを設定します。

```
contentType="TYPE"
```

pageEncoding には、次の構文を使用します。

```
pageEncoding="character_set"
```

すべての属性を設定するには、次の構文を使用します。

```
<%@ page ... contentType="TYPE; charset=character_set"
            pageEncoding="character_set" ... %>
```

TYPE は IANA の MIME タイプで、character_set は IANA のキャラクタ・セットです。contentType 属性を使用してキャラクタ・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、contentType と pageEncoding の設定例を示します。

```
<%@ page language="java" contentType="text/html" %>
```

または

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" %>
```

または

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
                         pageEncoding="US-ASCII" %>
```

page ディレクティブ設定がない場合のデフォルト設定は、次のとおりです。

- 従来の JSP ページに対するデフォルトの MIME タイプは text/html で、JSP XML 文書に対するデフォルトは text/xml です。
- ページ・ソースの文字エンコード（変換時）のデフォルトは、従来の JSP ページの場合は ISO-8859-1（Latin-1 とも呼ばれます）、JSP XML 文書の場合は UTF-8 または UTF-16 です。
- レスポンスの文字エンコードのデフォルトは、従来の JSP ページの場合は ISO-8859-1、JSP XML 文書の場合は UTF-8 または UTF-16 です。

UTF-8 と UTF-16 のいずれに設定するかは、http://www.w3.org/TR/REC-xml.html にある XML 仕様の「Autodetection of Character Encodings」に従います。

ただし、文字エンコードに関しては、pageEncoding と contentType には関連があります。次の表を参照してください。

	contentType エンコーディングを指定	**contentType エンコーディングを指定しない**
pageEncoding を指定	ページ・ソースのエンコーディングは pageEncoding に従う レスポンスのエンコーディングは contentType に従う	ページ・ソースのエンコーディングは pageEncoding に従う レスポンスのエンコーディングは pageEncoding に従う
pageEncoding を指定しない	ページ・ソースのエンコーディングは contentType に従う レスポンスのエンコーディングは contentType に従う	ページ・ソースのエンコーディングはデフォルトに従う レスポンスのエンコーディングはデフォルトに従う

使用方法については、次の重要な点に注意してください。

- contentType または pageEncoding を設定する page ディレクティブは、JSP ページ内のできるかぎり前の位置にある必要があります。
- ページが JSP XML 文書である場合、pageEncoding 設定は無視されます。かわりに、Web コンテナは、その文書の XML エンコード宣言を使用します。次に例を示します。

```
<?xml version="1.0" encoding="EUC-JP">
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
<jsp:directive.page contentType="text/html;charset=Shift_Jis" />
<jsp:directive.page pageEncoding="UTF-8" />
...
```

  有効なページ・エンコードは EUC-JP であり、UTF-8 ではありません。

- pageEncoding は、バイト・シーケンスがターゲット・キャラクタ・セット内の正当な文字を表しているページに対してのみ使用してください。
- contentType は、バイト・シーケンスがターゲット・キャラクタ・セット内の正当な文字を表しているページまたはレスポンス出力に対してのみ使用してください。
- たとえば、contentType で指定されているレスポンス出力のターゲット・キャラクタ・セットは、ページ・ソースのキャラクタ・セットに対するスーパーセットである必要があります。たとえば、UTF-8 は Big5 のスーパーセットですが、ISO-8859-1 はスーパーセットではありません。

- page ディレクティブのパラメータは静的です。レスポンスに対して異なるキャラクタ・セット仕様が必要であることが実行時に判明した場合、ページでは次のいずれかの方法で対処します。
  - サーブレットのレスポンス・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。詳細は、9-4 ページの「コンテンツ・タイプの動的な設定」を参照してください。

  または

  - 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。
- JSP XML 文書ではなく、ISO-8859-1 以外のキャラクタ・セットで記述された従来の JSP ページ・ソースの場合は、適切なキャラクタ・セットを、contentType 属性または pageEncoding 属性を使用して page ディレクティブに設定する必要があります。Web コンテナは変換時に設定を認識している必要があるため、ページ・エンコードのキャラクタ・セットは動的に設定できません。
- このマニュアルでは、わかりやすくするために、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・パラメータに、すべて同じエンコードを使用する一般的な場合を前提にしています。ただし、技術的には異なるエンコードも可能です。リクエスト・パラメータのエンコードは、ブラウザで制御されます。ただし、Netscape ブラウザと Internet Explorer ブラウザは、レスポンス・パラメータに指定した設定に従います。

IANA は、次のサイトで MIME タイプのレジストリを維持します。

```
ftp://www.isi.edu/in-notes/iana/assignments/media-types/media-types
```

IANA は、次のサイトで文字エンコードのレジストリを維持します。リストに「preferred MIME name」（推奨 MIME 名）と示されている場合は、それを使用します。

```
http://www.iana.org/assignments/character-sets
```

9-5 ページの「JSP ライター・オブジェクトのキャラクタ・セットに対する Oracle の拡張機能」で説明されている Oracle の追加拡張機能を除き、IANA リストのキャラクタ・セットのみを使用してください。

コンテンツ・タイプの動的な設定

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判明しない場合は、JSP ページでコンテンツ・タイプを動的に設定できます。標準の javax.servlet.ServletResponse インタフェースは、このために次のメソッドを指定します。

```
void setContentType(java.lang.String contenttype)
```

重要： OC4J 環境でコンテンツ・タイプの動的設定を使用するには、JSP の static_text_in_chars 構成パラメータを有効にする必要があります。

JSP ページの暗黙的な response オブジェクトは javax.servlet.http.HttpServletResponse インスタンスです。ここでは、HttpServletResponse インタフェースによって ServletResponse インタフェースが拡張されます。

page ディレクティブの contentType 設定のように、setContentType() メソッドの入力には、MIME タイプのみ、またはキャラクタ・セットと MIME タイプの両方を含めることができます。次に例を示します。

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

または

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

page ディレクティブと同様に、デフォルトの MIME タイプは、従来の JSP ページの場合は text/html、JSP XML 文書の場合は text/xml で、デフォルトの文字エンコードは ISO-8859-1 です。

JspWriter オブジェクトへの出力を記述する前に、ページのできるかぎり前の位置にコンテンツ・タイプを設定します。

setContentType() メソッドは、変換時に、JSP ページのテキストの解析に影響を与えません。変換時に特定のキャラクタ・セットが必要な場合は、page ディレクティブに指定する必要があります。詳細は、9-2 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」を参照してください。

JSP ライター・オブジェクトのキャラクタ・セットに対する Oracle の拡張機能

標準的な使用方法では、response オブジェクトのコンテンツ・タイプのキャラクタ・セットは、page ディレクティブの contentType パラメータまたは response.setContentType() メソッドで決定され、自動的に JSP ライター・オブジェクトのキャラクタ・セットになります。JSP ライター・オブジェクトは、javax.servlet.jsp.JspWriter インスタンスです。

ただし、一部のキャラクタ・セットは、IANA で認識されていないため、標準のコンテンツ・タイプ設定で使用できません。このため、OC4J には、oracle.jsp.util.PublicUtil クラスの静的な setWriterEncoding() メソッドが用意されています。

```
static void setWriterEncoding(JspWriter out, String encoding)
```

このメソッドによって、JSP ライターのキャラクタ・セットを直接指定し、response オブジェクトのキャラクタ・セットをオーバーライドできます。次の例では、コンテンツ・タイプのキャラクタ・セットとして Big5 を使用しますが、JSP ライターのキャラクタ・セットとして、Big5 の非 IANA 香港語キャラクタ・セットである MS950 を指定します。

```
<%@ page contentType="text/html; charset=Big5" %>
<% oracle.jsp.util.PublicUtil.setWriterEncoding(out, "MS950"); %>
```

注意： setWriterEncoding() メソッドは、JSP ページ内のできるかぎり前の位置で使用してください。

マルチバイト・パラメータ・エンコードに対する JSP サポート

サーブレット仕様では、javax.servlet.ServletRequest インタフェースに setCharacterEncoding() メソッドがあります。このメソッドは、サーブレット・コンテナのデフォルトのエンコードが、マルチバイトのリクエスト・パラメータと Bean プロパティの設定（Java コード内の getParameter() コールや JSP コードに Bean プロパティを設定する jsp:setProperty タグなど）に対して適切でない場合に役に立ちます。

setCharacterEncoding() メソッドと Oracle の同等の拡張機能は、特に次の場合に、パラメータの名前と値に影響を与えます。

- リクエスト・オブジェクト getParameter() メソッドの出力
- リクエスト・オブジェクト getParameterValues() メソッドの出力
- リクエスト・オブジェクト getParameterNames() メソッドの出力
- Bean プロパティ値に対する jsp:setProperty の設定

この項には、次の項目が含まれます。

標準の setCharacterEncoding() メソッド

サーブレット仕様 2.3 以上では、setCharacterEncoding() メソッドは、HTTP リクエストの読取りに使用するデフォルト以外の文字エンコードを指定する標準機能として、javax.servlet.ServletRequest インタフェースに指定されています。このメソッドのシグネチャは、次のとおりです。

```
void setCharacterEncoding(java.lang.String enc)
                          throws java.io.UnsupportedEncodingException
```

enc パラメータは、任意の文字エンコードの名前を指定する文字列で、デフォルトの文字エンコードをオーバーライドします。このメソッドは、リクエスト・パラメータを読み取る前、または getReader() メソッド（ServletRequest インタフェースに指定）を介して入力を読み取る前にコールしてください。

対応する次の getter メソッドもあります。

```
String getCharacterEncoding()
```

A

サード・パーティ・ライセンス

この付録には、Oracle Application Server に付属するすべてのサード・パーティ製品のサード・パーティ・ライセンスが含まれます。この章の内容は、次のとおりです。

Apache

Apache のライセンス条件に基づき、Oracle は次のライセンス文書を表示することが求められています。ただし、Oracle プログラム（Apache ソフトウェアを含む）を使用する権利は、この製品に付随する Oracle プログラム・ライセンスによって決定され、次のライセンス文書に含まれる条件でこの権利が変更されることはありません。反対の内容が Oracle プログラム・ライセンス内にあった場合でも、Apache ソフトウェアは現状のままで Oracle から提供されるものであり、いかなる種類の保証またはサポートも Oracle または Apache から提供されません。

The Apache Software License

License

"Contribution" shall mean any work of authorship, including
the original version of the Work and any modifications or additions
to that Work or Derivative Works thereof, that is intentionally
submitted to Licensor for inclusion in the Work by the copyright owner
or by an individual or Legal Entity authorized to submit on behalf of
the copyright owner. For the purposes of this definition, "submitted"
means any form of electronic, verbal, or written communication sent
to the Licensor or its representatives, including but not limited to
communication on electronic mailing lists, source code control systems,
and issue tracking systems that are managed by, or on behalf of, the
Licensor for the purpose of discussing and improving the Work, but
excluding communication that is conspicuously marked or otherwise
designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity
on behalf of whom a Contribution has been received by Licensor and
subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
this License, each Contributor hereby grants to You a perpetual,
worldwide, non-exclusive, no-charge, royalty-free, irrevocable
copyright license to reproduce, prepare Derivative Works of,
publicly display, publicly perform, sublicense, and distribute the
Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
this License, each Contributor hereby grants to You a perpetual,
worldwide, non-exclusive, no-charge, royalty-free, irrevocable
(except as stated in this section) patent license to make, have made,
use, offer to sell, sell, import, and otherwise transfer the Work,
where such license applies only to those patent claims licensable
by such Contributor that are necessarily infringed by their
Contribution(s) alone or by combination of their Contribution(s)
with the Work to which such Contribution(s) was submitted. If You
institute patent litigation against any entity (including a
cross-claim or counterclaim in a lawsuit) alleging that the Work
or a Contribution incorporated within the Work constitutes direct
or contributory patent infringement, then any patent licenses
granted to You under this License for that Work shall terminate
as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
Work or Derivative Works thereof in any medium, with or without
modifications, and in Source or Object form, provided that You
meet the following conditions:

   (a) You must give any other recipients of the Work or
   Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
   stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
   that You distribute, all copyright, patent, trademark, and
   attribution notices from the Source form of the Work,
   excluding those notices that do not pertain to any part of
   the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
   distribution, then any Derivative Works that You distribute must
   include a readable copy of the attribution notices contained
   within such NOTICE file, excluding those notices that do not
   pertain to any part of the Derivative Works, in at least one

of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
   boilerplate notice, with the fields enclosed by brackets "[]"
   replaced with your own identifying information. (Don't include
   the brackets!) The text should be enclosed in the appropriate
   comment syntax for the file format. We also recommend that a
   file or class name and description of purpose be included on the
   same "printed page" as the copyright notice for easier
   identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

Notice

This product includes software developed by
The Apache Software Foundation (http://www.apache.org/).

Portions of this software were developed at the National Center
for Supercomputing Applications (NCSA) at the University of
Illinois at Urbana-Champaign.

This software contains code derived from the RSA Data Security
Inc. MD5 Message-Digest Algorithm, including various
modifications by Spyglass Inc., Carnegie Mellon University, and
Bell Communications Research, Inc (Bellcore).

Regular expression support is provided by the PCRE library package,
which is open source software, written by Philip Hazel, and copyright
by the University of Cambridge, England. The original software is
available from
   ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/

索引

M

N

O

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

ね

は

ひ

ふ

へ

ま

め

よ

り