BEA WebLogic Server™ および WebLogic Express®

WebLogic JSP プログラマーズ ガイド

BEA WebLogic Server バージョン 7.0
マニュアルの改訂 : 2002 年 8 月 28 日

WebLogic JSP プログラマーズ ガイド

パート番号	マニュアルの改訂	ソフトウェアのバージョン
なし	2002 年 8 月 28 日	BEA WebLogic Server バージョン 7.0

目次

6. WebLogic EJB-to-JSP 統合ツールの使い方

7. トラブルシューティング

このマニュアルの内容

このマニュアルでは、JavaServer Pages(JSP) および WebLogic Server を使用した e コマース アプリケーションのプログラミング方法について説明します。

このマニュアルの構成は次のとおりです。

- 第 1 章「JSP の概要」では、JSP の基本的な構文の概要とリファレンス、および WebLogic Server で JSP を使用する方法について説明します。
- 第 2 章「WebLogic JSP の管理」では、WebLogic JSP の管理およびコンフィグレーション タスクの概要について説明します。
- 第 3 章「WebLogic JSP リファレンス」では、JSP の記述に関するリファレンスを提供します。
- 第 4 章「カスタム WebLogic JSP タグ (cache、process、repeat) の使い方」では、WebLogic Server 配布キットで提供される 3 つのカスタム JSP タグ (`cache` タグ、`repeat` タグ、および `process` タグ) の使い方について説明します。
- 第 7 章「トラブルシューティング」では、JSP ファイルのデバッグ テクニックをいくつか説明します。

対象読者

このマニュアルは、JSP と Sun Microsystems の Java 2 Platform, Enterprise Edition (J2EE) を使用して e コマース アプリケーションを構築するアプリケーション開発者を対象としています。Web テクノロジ、オブジェクト指向プログラミング手法、および Java プログラミング言語に読者が精通していることを前提として書かれています。

e-docs Web サイト

BEA 製品のドキュメントは、BEA の Web サイトで入手できます。BEA のホーム ページで [製品のドキュメント] をクリックします。

このマニュアルの印刷方法

Web ブラウザの [ファイル | 印刷] オプションを使用すると、Web ブラウザからこのマニュアルのメイン トピックを一度に 1 つずつ印刷できます。

このマニュアルの PDF 版は、Web サイトで入手できます。PDF を Adobe Acrobat Reader で開くと、マニュアルの全体 (または一部分) を書籍の形式で印刷できます。PDF を表示するには、WebLogic Server ドキュメントのホーム ページを開き、[ドキュメントのダウンロード] をクリックして、印刷するマニュアルを選択します。

Adobe Acrobat Reader は、Adobe の Web サイト (http://www.adobe.co.jp) から無料で入手できます。

関連情報

- Sun Microsystems の JSP 1.1 仕様
  (`http://java.sun.com/products/jsp/download.htm`)
- 『WebLogic JSP Tag Extensions プログラマーズ ガイド』
  (`http://edocs.beasys.co.jp/e-docs/wls/docs70/taglib/index.html`)
- 『Web アプリケーションのアセンブルとコンフィグレーション』
  (`http://edocs.beasys.co.jp/e-docs/wls/docs70/webapp/index.html`)

サポート情報

BEA のドキュメントに関するユーザからのフィードバックは弊社にとって非常に重要です。質問や意見などがあれば、電子メールで docsupport-jp@bea.com までお送りください。寄せられた意見については、ドキュメントを作成および改訂する BEA の専門の担当者が直に目を通します。

電子メールのメッセージには、ご使用のソフトウェア名とバージョン名、およびマニュアルのタイトルと作成日付をお書き添えください。本バージョンの BEA WebLogic Server について不明な点がある場合、または BEA WebLogic Server のインストールおよび動作に問題がある場合は、BEA WebSupport (www.bea.com) を通じて BEA カスタマ サポートまでお問い合わせください。カスタマ サポートへの連絡方法については、製品パッケージに同梱されているカスタマ サポート カードにも記載されています。

カスタマ サポートでは以下の情報をお尋ねしますので、お問い合わせの際はあらかじめご用意ください。

- お名前、電子メール アドレス、電話番号、ファクス番号
- 会社の名前と住所
- お使いの機種とコード番号
- 製品の名前とバージョン
- 問題の状況と表示されるエラー メッセージの内容

表記規則

このマニュアルでは、全体を通して以下の表記規則が使用されています。

表記法	適用
〔Ctrl〕+ 〔Tab〕	同時に押すキーを示す。
斜体	強調または本のタイトルを示す。

等幅テキスト	コード サンプル、コマンドとそのオプション、Java クラス、データ型、ディレクトリ、およびファイル名とその拡張子を示す。等幅テキストはキーボードから入力するテキストも示す。
	例:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

斜体の等幅テキスト	コード内の変数を示す。 例: `String CustomerName;`
すべて大文字のテキスト	デバイス名、環境変数、および論理演算子を示す。 例: LPT1 BEA_HOME OR
{ }	構文内の複数の選択肢を示す。

表記法	適用
[ ]	構文内の任意指定の項目を示す。 例： `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`
|	構文の中で相互に排他的な選択肢を区切る。 例： `java weblogic.deploy [list|deploy|undeploy|update] password {application} {source}`
...	コマンドラインで以下のいずれかを示す。 ■ 引数を複数回繰り返すことができる。 ■ 任意指定の引数が省略されている。 ■ パラメータや値などの情報を追加入力できる。
. . .	コード サンプルまたは構文で項目が省略されていることを示す。

1　JSP の概要

このマニュアルでは、JavaServer Pages (JSP) の基本的な構文の概要とリファレンス、および WebLogic Server で JSP を使用する方法について説明します。JSP を使用したプログラミングに関する情報を網羅する目的で作成されたものではありません。

以下の節では、JSP の概要について説明します。

- JSP とは
- WebLogic における JSP の実装
- JSP リクエストの処理方法
- 補足情報

JSP とは

JavaServer Pages (JSP) は、Java を HTML と組み合わせて Web ページで動的コンテンツを提供するための Sun Microsystems の仕様です。動的コンテンツを作成する場合、JSP では HTML ページに直接 Java コードを埋め込むことができるのに対し、HTTP サーブレットでは HTML を Java コードに埋め込むので、JSP の方が HTTP サーブレットよりコーディングが容易です。JSP は、Java 2 Enterprise Edition (J2EE) の一部です。

JSP を使用すると、Web ページから動的コンテンツなどの表現要素を分離することができます。そのために、ページのグラフィック設計を担当する HTML 開発者と、動的コンテンツを構成するソフトウェア開発を担当する Java 開発者という 2 種類の開発者の要求を満たします。

JSP は J2EE 標準の一部であるので、WebLogic Server を含むさまざまなプラットフォームで JSP をデプロイすることが可能です。さらに、サードパーティ ベンダやアプリケーション開発者は、動的コンテンツを構成するために JSP ページから参照できるカスタム JSP タグを定義したり、JavaBean コンポーネントを配布したりできます。

WebLogic における JSP の実装

BEA WebLogic JSP は、Sun Microsystems の JSP 1.2 仕様をサポートしています。JSP 1.2 では、カスタム JSP タグ拡張の定義がサポートされています。詳細については、『WebLogic JSP Tag Extensions プログラマーズ ガイド』を参照してください。

JSP 1.3 仕様の WebLogic Server 実装では、JSP に文字を出力するために、`getWriter` ではなく `getOutputStream` を呼び出します。そのため、特定の拡張文字が削除されたり、不適切に表示されたりする可能性があります。オングストローム単位や温度記号のような拡張文字用の HTML コードを使用すると、WebLogic Server インスタンスで適切に処理されるようになります。

また、WebLogic Server は、Sun Microsystems の サーブレット 2.3 仕様をサポートしています。

JSP リクエストの処理方法

WebLogic Server は、次の順序で JSP 要求を処理します。

1. ブラウザが、`.jsp` ファイルを WebLogic Server に要求します。
2. WebLogic Server が要求を読み取ります。
3. WebLogic Server は、JSP コンパイラを使って JSP を `javax.servlet.jsp.JspPage` を実装するサーブレット クラスに変換します。JSP ファイルは、ページが初めて要求されたとき、あるいは、JSP ファ

イルが変更されたときなど、必要な場合にだけコンパイルされます。通常は、以前に編集された JSP サーブレット クラスが再利用されるので、以降の応答は非常に高速です。

4. 生成された `JspPage` サーブレット クラスが呼び出され、ブラウザからの要求を処理します。

また、ブラウザから要求せずに直接 JSP コンパイラを呼び出すこともできます。詳細については、「WebLogic JSP コンパイラの使い方」を参照してください。JSP コンパイラはまず Java サーブレットを作成するので、コンパイラが生成する Java ファイルを見ることもできますし、生成された `JspPage` サーブレット クラスを HTTP サーブレットとして登録することもできます。

補足情報

- Sun Microsystems の「JavaServer Pages Tutorial」(
- Sun Microsystems の JSP 製品の概要
- Sun Microsystems の JSP 1.1 仕様
- 『WebLogic JSP Tag Extensions プログラマーズ ガイド』『WebLogic HTTP サーブレット プログラマーズ ガイド』『Web アプリケーションのアセンブルとコンフィグレーション』

2 WebLogic JSP の管理

以下の節では、WebLogic JavaServer Pages (JSP) をデプロイするために必要な管理およびコンフィグレーション作業について説明します。

- WebLogic JSP 管理の概要
- JSP 処理パラメータの設定

JSP の管理とコンフィグレーションの詳細については、「JSP のコンフィグレーション」を参照してください。

WebLogic JSP 管理の概要

Java 2 Enterprise Edition 標準に従って、JSP は Web アプリケーションの一部としてデプロイされます。Web アプリケーションとは、HTTP サーブレット、JavaServer Pages (JSP)、静的 HTML ページ、画像、その他のリソースなどからなる一群のアプリケーション コンポーネントのことです。

Web アプリケーションのコンポーネントは、標準のディレクトリ構造に従って配置されます。このディレクトリ構造を使ってアプリケーションをデプロイしたり、ファイルをいったん Web アプリケーション アーカイブ (`.war`) と呼ばれる 1 つのファイルにアーカイブしてから `.war` ファイルをデプロイしたりできます。Web アプリケーションのファイルに含まれる 2 つのデプロイメント記述子を使用して、Web アプリケーションのリソースおよび処理パラメータに関する情報を定義します。詳細については、『Web アプリケーションのアセンブルとコンフィグレーション』を参照してください。

1 つ目のデプロイメント記述子 `web.xml` は、Sun Microsystems のサーブレット 2.2 仕様に定義されています。これは、Web アプリケーションを定義する標準フォーマットを提供します。第 2 のデプロイメント記述子、`weblogic.xml` は、`web.xml` ファイルで定義されているリソースを WebLogic Server 内で使用可能な

リソースにマップして、JSP パラメータと HTTP セッション パラメータを定義する WebLogic 固有のデプロイメント記述子です。詳細については、「Web プリケーションのデプロイメント記述子の記述」を参照してください。

JSP には、HTTP サーブレットとは違い、固有のマッピングは不要です。Web アプリケーションに JSP をデプロイするには、Web アプリケーションのルート ディレクトリ (またはルートのサブディレクトリ) に JSP を置くだけです。登録する必要はありません。サーブレットと JSP の両方を同じ Web アプリケーションにデプロイできます。

JSP 処理パラメータの設定

JSP の動作を制御するパラメータは、Web アプリケーションの WebLogic 固有のデプロイメント記述子、`weblogic.xml` 内に定義します。このファイルの編集の詳細については、『Web アプリケーションのアセンブルとコンフィグレーション』を参照してください。

WebLogic 固有のデプロイメント記述子における JSP パラメータ (デフォルト値を含む) の詳細については、「jsp-descriptor 要素」を参照してください。

`weblogic.xml` で設定するパラメータは次のとおりです。

- `compileCommand`
- `compileFlags`
- `compilerclass`
- `encoding`
- `keepgenerated`
- `packagePrefix`
- `pageCheckSeconds`
- `verbose`
- `workingDir`

3 WebLogic JSP リファレンス

以下の節では、JavaServer Pages (JSP) の作成に関するリファレンス情報を提供します。

- JSP タグ
- 暗黙的オブジェクト用の予約語
- WebLogic JSP のディレクティブ
- スクリプトレット
- 式
- HTML と埋め込み Java を使用した JSP の例
- アクション
- JSP におけるユーザ入力データのセキュリティ
- JSP でのセッションの使い方
- JSP からのアプレットのデプロイメント
- JSP のファイルのルックアップと大文字 / 小文字の区別
- WebLogic JSP コンパイラの使い方

JSP タグ

以下の表では、JSP ページで使用できる基本的なタグについて説明します。短縮形のタグには、それぞれ対応する XML タグがあります。

表 3-1 JSP ページの基本タグ

JSP タグ	構文	説明
Scriptlet	<% java_code %> ... または対応する以下の XML タグを使用する <jsp:scriptlet> java_code </jsp:scriptlet>	Java ソース コード スクリプトレットを、HTML 内に埋め込む。Java コードが実行され、その出力が、残りの HTML と一緒にページに順に挿入される。詳細については、「スクリプトレット」を参照。
Directive	<%@ *dir-type dir-attr* %> ... または対応する以下の XML タグを使用する <jsp:directive.*dir_type dir_attr* />	ディレクティブには、アプリケーション サーバへのメッセージが含まれる。 また、attr="value" という形式の名前 / 値のペアも含まれる。このペアは、アプリケーション サーバに対する追加の命令を提供する。「WebLogic JSP のディレクティブ」を参照。
Declarations	<%! declaration %> ... または対応する以下の XML タグを使用する <jsp:declaration> declaration; </jsp:declaration>	ページ内のほかの宣言、スクリプトレット、あるいは式によって参照されることがある変数またはメソッドを宣言する。「宣言」を参照。

表 3-1 JSP ページの基本タグ

JSP タグ	構文	説明
Expression	<%= expression %> ... または対応する以下の XML タグを使用する <jsp:expression> expression </expression>	ページ要求時に評価され、String に変換され、そして、JSP 応答の出力ストリームにインライン化されて送られる Java 式を定義する。「式」を参照。
Actions	<jsp:useBean ... > Bean をここでインスタンス化する場合は、ここに JSP 本文を入れる </jsp:useBean> <jsp:setProperty ... > <jsp:getProperty ... > <jsp:include ... > <jsp:forward ... > <jsp:plugin ... >	JSP の高度な機能へのアクセスを提供し、XML 構文だけを使用する。これらのアクションは、JSP 1.1 仕様に定義されているようにサポートされている。「アクション」を参照。

暗黙的オブジェクト用の予約語

JSP では、スクリプトレットや式内の暗黙的オブジェクトのための予約語が用意されています。この暗黙的オブジェクトとは、JSP ページに有用なメソッドや情報を提供する Java オブジェクトです。WebLogic JSP は、JSP 1.1 仕様に定義されている暗黙的オブジェクトをすべて実装しています。JSP API については、Sun Microsystems の JSP ホームページにある Javadoc で説明されています。

注意： これらの暗黙的オブジェクトは、スクリプトレットや式の内部でのみ使用できます。宣言で定義されるメソッドからこれらのキーワードを使用すると、未定義の変数を参照することになるため、変換時にコンパイルエラーが発生します。

`request`

`request` は `HttpServletRequest` オブジェクトを表します。ブラウザからの要求についての情報が含まれ、クッキー、ヘッダ、およびセッション データを取得するために便利なメソッドも用意されています。

`response`

`response` は、`HttpServletResponse` オブジェクト、JSP ページからブラウザに返送される応答の設定に便利な複数のメソッドを表します。この応答の例として、クッキーとその他のヘッダ情報があります。

警告：`response.getWriter()` メソッドを JSP ページ内で使用することはできません。使用した場合は、実行時に例外が送出されます。JSP 応答をブラウザに返送するには、スクリプトレット コード内で可能な限り `out` キーワードを使用します。WebLogic Server の `javax.servlet.jsp.JspWriter` の実装では、`javax.servlet.ServletOutputStream` を使用しています。これは、`response.getServletOutputStream()` を使用できることを暗黙的に表しますが、WebLogic Server に固有の実装です。しかし、この実装は WebLogic Server に固有のものであることに注意してください。このコードの保守容易性および移植性を保つには、`out` キーワードを使用します。

`out`

`out` は、`javax.jsp.JspWriter` のインスタンスで、ブラウザへの出力の返送に使えるメソッドを持っています。

出力ストリームを必要とするメソッドを使用している場合、`JspWriter` は動作しません。この制限を回避するには、バッファ化ストリームを提供して、このストリームを `out` に記述します。次の例に、例外スタックトレースを `out` に記述する方法を示します。

```
ByteArrayOutputStream ostr = new ByteArrayOutputStream();
exception.printStackTrace(new PrintWriter(ostr));
out.print(ostr);
```

`pageContext`

`pageContext` は、`javax.servlet.jsp.PageContext` オブジェクトを表します。さまざまなスコープのネームスペースやサーブレット関連オブジェクトにアクセスするために便利な API で、一般的なサーブレット関連機能のためのラッパー メソッドを提供します。`pageContext` の WebLogic Server 実装は、シリアライズされていないオブジェクトを受け容れません。

`session`

`session` は、リクエストに対する `javax.servlet.http.HttpSession` オブジェクトを表します。セッション ディレクティブはデフォルトでは true にされているので、`session` はデフォルトで有効です。JSP 1.1 使用では、セッション ディレクティブが `false` に設定されている場合、`session` キーワードを使用すると変換時に致命的なエラーが発生することが説明されています。サーブレットでのセッションの使い方については、『WebLogic HTTP サーブレット プログラマーズ ガイド』を参照してください。

`application`

`application` は、`javax.servlet.ServletContext` オブジェクトを表します。サーブレット エンジンやサーブレット環境に関する情報の検索に使用します。

リクエストを転送または取り込む場合は、`ServletContext` を使用してサーブレット `requestDispatcher` にアクセスできます。または、ほかのサーブレットへのリクエストの転送に JSP `forward` ディレクティブ、ほかのサーブレットからの出力の取り込みに JSP `include` ディレクティブを使用することもできます。

`config`

`config` は `javax.servlet.ServletConfig` オブジェクトを表し、サーブレット インスタンスの初期化パラメータへのアクセスを提供します。

`page`

`page` は、この JSP ページから生成されたサーブレット インスタンスを表します。スクリプトレット コード内の Java キーワード `this` と同義です。

`page` は `java.lang.Object` のインスタンスとして定義されるため、使用する際には JSP ページを実装しているサーブレットのクラス タイプにキャストする必要があります。デフォルトでは、このサーブレット クラスは JSP ファイル名を取って名付けられます。便宜上、`page` を使用する代わりに、Java キーワード `this` を使用してサーブレット インスタンスを参照し、初期化パラメータへアクセスすることをお勧めします。

基盤となる HTTP サーブレット フレームワークの詳細については、関連する開発者ガイド、『WebLogic HTTP サーブレット プログラマーズ ガイド』を参照してください。

WebLogic JSP のディレクティブ

関数を実行したり、JSP ページを特定の方法で解釈したりするよう WebLogic JSP に指示するには、ディレクティブを使用します。ディレクティブは、JSP ページのどこに挿入してもかまいません。通常、ディレクティブの位置は無関係で (include ディレクティブを除く)、複数のディレクティブ タグを使用できます。ディレクティブは、ディレクティブ タイプとその 1 つまたは複数の属性で構成されます。

構文は、次に示すように、短縮形と XML の 2 種類の構文を使用できます。

- 短縮形 :

  ```
  <%@ dir_type dir_attr %>
  ```

- XML :

  ```
  <jsp:directive.dir_type dir_attr />
  ```

dir_type はディレクティブのタイプ、dir_attr はそのディレクティブ タイプの 1 つまたは複数のディレクティブ属性のリストで置き換えます。

ディレクティブには、page、taglib、include の 3 種類があります。

ページ ディレクティブを使用した文字エンコーディングの設定

文字エンコーディング セットを指定するには、そのページの先頭で次のディレクティブを使用します。

```
<%@ page contentType="text/html; charset=custom-encoding” %>
```

custom-encoding は、標準の HTTP スタイルの文字セット名で置き換えます iana.org。

contentType ディレクティブで指定した文字セットは、JSP およびその JSP に含まれるすべての JSP で使用される文字セットを指定します。

Web アプリケーションの WebLogic 固有のデプロイメント記述子で、デフォルトの文字エンコーディングを指定できます。詳細については、「jsp-descriptor 要素」を参照してください。

taglib ディレクティブの使い方

taglib ディレクトリを使用して、JSP ページが、タグ ライブラリに定義されているカスタム JSP タグ拡張を使用することを宣言します。カスタム JSP タグの記述方法と使い方の詳細については、『JSP Tag Extensions プログラマーズ ガイド』を参照してください。

宣言

宣言を使用して、生成された JSP サーブレット内のクラス スコープ レベルの変数とメソッドを定義します。JSP タグの間に記述された宣言は、JSP ページのほかの宣言やスクリプトレットからアクセスできます。次に例を示します。

```
<%!
  int i=0;
  String foo= "Hello";
  private void bar() {
      // ... ここに Java コード ...
  }
%>
```

クラス スコープのオブジェクトは、サーブレットの同一のインスタンス内で実行中の複数のスレッド間で共有されます。共有違反を防ぐには、クラス スコープのオブジェクトを同期させます。スレッドセーフなコードの記述に自信がない場合は、次のディレクティブを使用して、サーブレットを非スレッドセーフとして宣言することができます。

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

デフォルトでは、この属性は true に設定されています。false に設定した場合、生成されたサーブレットは javax.servlet.SingleThreadModel インタフェースを実装します。これにより、同一のサーブレット インスタンス内で複数のスレッドが実行されることはありません。isThreadSafe を false に設定すると、メモリ消費量が増えるので、パフォーマンスが低下することがあります。

スクリプトレット

JSP スクリプトレットは、JSP サーブレットの HTTP 応答の Java 本文を構成します。次に示すように、短縮形または XML の scriptlet タグを使用して、JSP ページ内にスクリプトレットを包含します。

短縮形 :

```
<%
  // Java コードをここに書く
%>
```

XML :

```
<jsp:scriptlet>
  // Java コードをここに書く
</jsp:scriptlet>
```

スクリプトレットの特長は次のとおりです。

- Java コードとプレーンな HTML が混在する複数ブロックのスクリプトレットを作成できます。
- Java コンストラクトやブロックの内部であっても、任意の場所で、HTML と Java コードを切り替えることができます。3-9 ページの「HTML と埋め込み Java を使用した JSP の例」の例では、Java ループを宣言し、HTML に切り替え、再び Java コードに戻ってループを閉じています。ループ内部の HTML は、ループが反復する回数分出力として生成されます。
- あらかじめ定義された変数 `out` を使用して、Java コードからサーブレット出力ストリームに、HTML テキストを直接出力することができます。`print()` メソッドを呼び出して、HTTP ページ応答に文字列を追加します。

  ユーザによって入力されたデータを出力するたびに、入力された HTML 特殊文字を削除することをお勧めします。これらの文字を削除しないと、Web サイトがクロスサイト スクリプト攻撃を受ける可能性があります。詳細については、3-14 ページの「JSP におけるユーザ入力データのセキュリティ」を参照してください。
- この Java タグはインライン タグであり、強制的にパラグラフを分けることはありません。

式

JSP ファイルの中に式を含めるには、次のタグを使用します。

```
<%= expr %>
```

expr を Java 式で置き換えます。式が評価されるときに、その `string` 表現が HTML 応答ページ内にインラインで配置されます。このタグは次のタグの短縮形です。

```
<% out.print( expr ); %>
```

このテクニックを使用すると、JSP ページ内の HTML を読みやすくすることができます。次の節のサンプルでは expression タグの使い方を示します。

通常、式はユーザが入力したデータを返すために使用されます。ユーザによって入力されたデータを出力するたびに、入力された HTML 特殊文字を削除することをお勧めします。これらの文字を削除しないと、Web サイトがクロスサイトスクリプト攻撃を受ける可能性があります。詳細については、3-14 ページの「JSP におけるユーザ入力データのセキュリティ」を参照してください。

HTML と埋め込み Java を使用した JSP の例

次に、HTML と埋め込み Java を使用した JSP の例を示します。

```
<html>
  <head><title>Hello World Test</title></head>

<body bgcolor=#ffffff>
<center>
<h1> <font color=#DB1260> Hello World Test </font></h1>
<font color=navy>

<%

    out.print("Java-generated Hello World");
%>

</font>
<p> This is not Java!
```

```
<p><i>Middle stuff on page</i>
<p>
<font color=navy>

<%
     for (int i = 1; i<=3; i++) {
%>
        <h2>This is HTML in a Java loop! <%= i %> </h2>
<%
     }
%>

</font>
</center>
</body>
</html>
```

上に示したコードがコンパイルされると、ブラウザには次のようなページが表示されます。

アクション

JSP アクションを使用して、JavaBean によって表されるオブジェクトの変更、使用、または作成を行います。アクションでは、XML 構文のみが使用されます。

JSP での JavaBean の使い方

<jsp:useBean> アクション タグを使用すると、JavaBean 仕様に準拠した Java オブジェクトをインスタンス化し、それを JSP ページから参照できます。

JavaBean 仕様に準拠するには、オブジェクトに次のものが必要です。

- 引数をとらないパブリック コンストラクタ
- 各 variable フィールドに対する setVariable() メソッド
- 各 variable フィールドに対する getVariable() メソッド

JavaBean オブジェクトのインスタンス化

<jsp:useBean> タグは、既存の名前付き Java オブジェクトを特定のスコープから検索しようと試みます。既存のオブジェクトが見つからなければ、新しいオブジェクトをインスタンス化し、id 属性で指定された名前にそのオブジェクトを関連付けようとします。オブジェクトは、オブジェクトの可用性を決定する scope 属性で指定された場所に格納されます。たとえば、次のタグでは、examples.jsp.ShoppingCart というタイプの Java オブジェクトを、名前 cart の下の HTTP セッションから検索しようとします。

```
<jsp:useBean id="cart"
    class="examples.jsp.ShoppingCart" scope="session"/>
```

そのようなオブジェクトがその時点で存在していなければ、JSP は、新しいオブジェクトを作成し、名前 cart の下の HTTP セッション内に格納しようとします。クラスは、WebLogic Server の起動に使用する CLASSPATH 内、または JSP を含む Web アプリケーションの WEB-INF/classes ディレクトリに入っている必要があります。

処理する必要がある実行時例外が存在するため、<jsp:useBean> タグと共に errorPage ディレクティブを使用するようにしてください。errorPage ディレクティブを使用しない場合、JavaBean で参照されるクラスを作成できず、InstantiationException が送出され、ブラウザにエラー メッセージが返されます。

Java 内で有効な型キャスト操作であれば、type 属性を使用して、その JavaBean タイプをほかのオブジェクトまたはインタフェースにキャストできます。class 属性なしで type 属性を使用する場合、JavaBean オブジェクトは、指定したスコープ内に既に存在している必要があります。有効でない場合は、InstantiationException が送出されます。

JavaBean インスタンス化のセットアップ作業

<jsp:useBean> タグ構文には、別の形式もあります。これを使用すると、オブジェクトがインスタンス化されるときに実行する JSP コードの本文を定義することができます。名前付き JavaBean が指定したスコープ内に既に存在する場合、本文は実行されません。この形式を使用すると、オブジェクトが最初に作成されるときのプロパティを設定することができます。次に例を示します。

```
<jsp:useBean id="cart" class="examples.jsp.ShoppingCart"
 scope=session>
    Creating the shopping cart now...
    <jsp:setProperty name="cart"
     property="cartName" value="music">
</jsp:useBean>
```

注意： class 属性なしで type 属性を使用すると、JavaBean オブジェクトはインスタンス化されません。したがって、本文を含めるようなタグ形式は使用しないでください。代わりに、単一のタグ形式を使用します。この場合、JavaBean が指定したスコープ内に存在していなければなりません。存在していない場合は InstantiationException が送出されます。errorPage ディレクティブを使用して、潜在的な例外を処理します。

JavaBean オブジェクトの使い方

JavaBean オブジェクトをインスタンス化したら、Java オブジェクトのように、JSP ファイルでその id 名によって参照できます。スクリプトレット タグや式評価タグ内で、その JavaBean オブジェクトを使うことができます。

<jsp:setProperty> タグを用いてその setXxx() メソッドを呼び出すこともできます。<jsp:getProperty> タグを用いて getXxx() メソッドを呼び出すこともできます。

JavaBean オブジェクトのスコープの定義

scope 属性を使用して、JavaBean オブジェクトの可用性とライフスパンを指定します。スコープは以下のいずれかです。

page
: JavaBean に対するデフォルトのスコープです。オブジェクトは現在のページの javax.servlet.jsp.PageContext に格納されます。この JSP ページの現在の呼び出しからのみ使用可能です。インクルードされた JSP ページでは使用できません。このページ要求の完了時には破棄されることになっています。

request
: request スコープを使用すると、オブジェクトは現在の ServletRequest 内に格納されます。同一の要求オブジェクトに渡された、インクルードされたほかの JSP ページで使用可能です。現在の要求が完了したときに破棄されます。

session
: 複数の HTTP ページにわたって JavaBean オブジェクトを追跡できるよう、session スコープを使用して HTTP セッション内に JavaBean オブジェクトを格納できます。JavaBean への参照は、このページの HttpSession オブジェクトに格納されます。このスコープを使用するには、JSP ページがセッションに参加できなければなりません。つまり、この page ディレクティブで、session を false に設定しないようにします。

application
: application- スコープ レベルでは、JavaBean オブジェクトは Web アプリケーションに格納されます。このスコープを使用すると、同じ Web アプリケーション内で実行中のほかのサーブレットや JSP ページからも、この JavaBean オブジェクトが使用可能になります。

JavaBean の使い方については、JSP 1.1 仕様を参照してください。

リクエストの転送

どのような種類の認証を使用している場合でも、<jsp:forward> タグでリクエストが転送されるときには、ユーザを再認証する必要がありません (デフォルト設定)。この動作を変更して、転送されたリクエストの認証を実行するには、<check-auth-on-forward/> 要素を WebLogic 固有のデプロイメント記述子 (weblogic.xml) の <container-descriptor> 要素に追加します。次に例を示します。

```
<container-descriptor>
   <check-auth-on-forward/>
</container-descriptor>
```

WebLogic 固有のデプロイメント記述子の編集方法については、「WebLogic 固有のデプロイメント記述子 (weblogic.xml) の記述」を参照してください。

リクエストのインクルード

<jsp:include> タグを使用すると、JSP に別のリソースを含めることができます。このタグは、次の 2 種類の属性を取ります。

page

ページ属性は、含めるリソースを指定するために使用します。次に例を示します。

```
<jsp:include page=”somePage.jsp”/>
```

flush

この属性を true に設定すると、ページ出力をバッファに格納し、リソースに含める前にそのバッファをフラッシュします。

<jsp:include> タグが JSP ページの別のタグの中に存在し、含めるリソースをそのタグで処理したい場合は、flush="false" に設定すると便利です。

JSP におけるユーザ入力データのセキュリティ

式とスクリプトレットを使用すると、JSP ではユーザからデータを受け取ったり、ユーザが入力したデータを返したりすることができます。たとえば コード リスト 3-1 のサンプル JSP では、文字列の入力をユーザに要求し、その文字列を

userInput というパラメータに割り当て、<%= request.getParameter("userInput")%> 式を使用してそのデータをブラウザに返します。

コード リスト 3-1 ユーザが入力した内容を返す式の使い方

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <body>
    <h1>My Sample JSP</h1>
      <form method="GET" action="mysample.jsp">
          Enter string here:
          <input type="text" name="userInput" size=50>
          <input type=submit value="Submit">
       </form>
    <br>
    <hr>
    <br>
    Output from last command:
    <%= request.getParameter("userInput")%>
  </body>
</html>
```

ユーザによる入力データを返す機能により、**クロスサイト スクリプト**と呼ばれるセキュリティの脆弱性がもたらされます。これは、ユーザのセキュリティ認可を盗用するために利用される可能性があります。クロスサイト スクリプトの詳細については、http://www.cert.org/tech_tips/malicious_code_mitigation.html の「Understanding Malicious Content Mitigation for Web Developers」(CERT のセキュリティ勧告) を参照してください。

セキュリティの脆弱性をなくすには、ユーザが入力したデータを返す前に、そのデータをスキャンして表 3-2 に示した HTML 特殊文字を探します。該当する文字が見つかれば、それらを HTML のエンティティまたは文字参照と置き換えます。文字を置換することによって、ブラウザがユーザの入力によるデータを HTML として実行することが回避されます。

表 3-2 置換が必要な HTML 特殊文字

置換が必要な特殊文字	置換後のエンティティ / 文字参照
<	<
>	>

表 3-2 置換が必要な HTML 特殊文字

置換が必要な特殊文字	置換後のエンティティ / 文字参照
(	&40;
)	&41;
#	&35;
&	&38;

WebLogic Server ユーティリティ メソッドの使用

WebLogic Server では、ユーザ供給データの特殊文字を置換する `weblogic.servlet.security.Utils.encodeXSS()` メソッドが提供されます。このメソッドを使用するには、ユーザ供給データを入力として提供します。次に例を示します。

```
<%= weblogic.servlet.security.Utils.encodeXSS(
request.getParameter("userInput"))%>
```

アプリケーション全体を保護するには、ユーザ供給データを返すたびに `encodeXSS()` メソッドを使用する必要があります。直前の例は `encodeXSS()` メソッドの使用が明らかに必要な場所であり、表 3-3 はメソッドを使用するかどうか検討すべき場所を示しています。

表 3-3 ユーザによる入力データを返すコード

ページのタイプ	ユーザによる入力データ	例
エラー ページ	入力エラー文字列、無効な URL、ユーザ名	「*username* はアクセスを許可されていない」ことを示すエラー ページ
ステータス ページ	ユーザ名、前のページの入力の要約	前のページで入力した内容の確認をユーザに求める要約ページ
データベース表示	データベースからのデータの提示	ユーザによって入力されたデータベース エントリのリストを表示するページ

JSP でのセッションの使い方

WebLogic JSP 内のセッションは、JSP 1.1 仕様に従って動作します。次に、セッションの使い方に関連する留意事項を挙げます。

- セッションには小さいオブジェクトを格納します。たとえば、セッションは EJB の格納ではなく、EJB プライマリ キーの格納に使用する方がよいでしょう。大量のデータは、データベースに格納します。セッションは、データの単純な文字列参照のみを保持する必要があります。
- サーブレットまたは JSP の動的な再ロードとともにセッションを使用する場合、サーブレット セッションに格納されるオブジェクトは、シリアライズ可能でなければなりません。サーブレットが新しいクラス ローダで再ロードされるため、シリアライゼーションが必要です。その結果、先に (サーブレットの古いバージョンから) ロードされたクラスと、新しいクラス ローダにロードされる (サーブレット クラスの新しいバージョンに対応した) クラスとの間で互換性がなくなります。この非互換性が原因で、サーブレットは `ClassCastException` エラーを返します。
- セッション データがユーザ定義型でなければならない場合、データ クラスはシリアライズできるようにする必要があります。さらに、セッションはシリアライズされた形のデータ オブジェクトを格納しなければなりません。このとき、シリアライゼーションはデータ クラスのすべてのバージョン間で互換性を持つ必要があります。
- 認証済みユーザをログアウトする必要がある場合は、『WebLogic HTTP サーブレット プログラマーズ ガイド』の「セッションのログアウトと終了」を参照してください。

JSP からのアプレットのデプロイメント

JSP を使用すると、適切なクライアント ブラウザ タグを含む HTML を生成することによって、Web ページに Java Plug-in を簡単に組み込むことができます。Java Plug-in では、クライアントの Web ブラウザにより実装された JVM の代わりに、Sun Microsystems から提供されている Java Runtime Environment (JRE) を

使用することができます。この機能により、アプレットと特定の種類の Web ブラウザとの間の非互換性の問題が回避できます。Java Plug-in は、Sun のサイト (http://java.sun.com/products/plugin/) から入手できます。

Internet Explorer と Netscape では使用している構文が異なるため、<jsp:plugin> アクションから生成されるサーブレット コードは、動的にブラウザ クライアントの種類を調べて、適切な <OBJECT> タグか、<EMBED> タグをその HTML ページに送ります。

<jsp:plugin> タグは、<APPLET> タグの属性と似た多くの属性、および使用する Java Plug-in のバージョンを設定するその他の属性を使用します。アプレットがサーバと通信する場合、アプレット コードを実行する JVM は WebLogic Server を実行する JVM と互換性がなければなりません。

次のサンプルでは、plugin アクションを使用してアプレットをデプロイします。

```
<jsp:plugin type="applet" code="examples.applets.PhoneBook1"
 codebase="/classes/" height="800" width="500"
 jreversion="1.1"
 nspluginurl=
 "http://java.sun.com/products/plugin/1.1.3/plugin-install.html"
 iepluginurl=
"http://java.sun.com/products/plugin/1.1.3/
  jinstall-113-win32.cab#Version=1,1,3,0" >

<jsp:params>
  <param name="weblogic_url" value="t3://localhost:7001">
  <param name="poolname" value="demoPool">
</jsp:params>

<jsp:fallback>
  <font color=#FF0000>Sorry, cannot run java applet!!</font>
</jsp:fallback>

</jsp:plugin>
```

上のサンプルの JSP 構文は、ブラウザに対して、(すでにダウンロードされていなければ) Java Plug-in バージョン 1.3.1 をダウンロードし、code 属性で指定したアプレットを、codebase で指定された位置から実行するように指示しています。

jreversion 属性は、アプレットが要求している Java Plug-in のバージョンを同定します。Web ブラウザは、このバージョンの Java Plug-in を使おうとします。プラグインがブラウザにまだインストールされていなければ、nspluginurl 属性および iepluginurl 属性が URL を指定し、そこで、Sun の Web サイトから

その Java Pug-in をダウンロードできます。一度プラグインが Web ブラウザにインストールされれば、再度そのプラグインをダウンロードする必要はありません。

WebLogic Server では Java 1.3.x VM を使用するため、<jsp:plugin> タグ内に、Java Plug-in バージョン 1.3.x を指定する必要があります。上のサンプル コードで 1.3 JVM を指定するには、対応する属性値を次のように置き換えます。

```
jreversion="1.3"
nspluginurl=
"http://java.sun.com/products/plugin/1.3/plugin-install.html"
iepluginurl=
"http://java.sun.com/products/plugin/1.3/jinstall-131-win32.cab"
```

プラグイン アクションのほかの属性は、<APPLET> タグの属性に相当します。アプレット パラメータは、<params> タグのペアで指定し、<jsp:plugin> タグと </jsp:plugin> タグでネストします。

<jsp:fallback> タグでは、<jsp:plugin> アクションでサポートされていないブラウザ用に HTML を置換することができます。<fallback> タグと </jsp:fallback> タグでネストされた HTML が、プラグイン構文の代わりに送られます。

JSP のファイルのルックアップと大文字 / 小文字の区別

SecurityConfigMBean の WebAppFilesCaseInsensitive 属性では、win32 以外のすべてのプラットフォームにおいて、Java Server Pages (JSP) のファイル ルックアップで大文字 / 小文字を区別するかどうかを指定します。標準の win32 ファイル システムにおけるファイル ルックアップでは、常に大文字 / 小文字は区別されません。

win 32 以外の大文字 / 小文字を区別しないファイル システム (UNIX の NT Samba マウントや、大文字 / 小文字を区別しないモードでインストールされた Mac OS など) で、大文字 / 小文字を区別しないルックアップを指定すると (この属性を <code>false</code> に設定する)、JSP はそのソース コードを返さなくなります。

たとえば、JSP が Samba マウントから提供されていて、大文字 / 小文字を区別しないルックアップを指定した場合、WebLogic Server はリクエスト内のすべてのファイル名拡張子を小文字に変換してから JSP をルックアップします。

WebLogic JSP コンパイラの使い方

JSP サーブレットは自動的に WebLogic JSP コンパイラを呼び出して JSP ページを処理するので、通常は直接コンパイラを呼び出す必要はありません。しかし、デバッグする場合など、状況によっては、コンパイラを直接呼び出した方が便利です。この節はコンパイラのリファレンスです。

WebLogic JSP コンパイラでは、JSP ファイルを解析して `.java` ファイルに変換し、生成された `.java` ファイルを標準の Java コンパイラを使用して Java クラスにコンパイルします。

JSP コンパイラの構文

JSP コンパイラは、ほかの WebLogic コンパイラ (RMI コンパイラや EJB コンパイラなど) とほとんど同じ方法で動作します。JSP コンパイラを起動するには、次のコマンドを入力します。

```
$ java weblogic.jspc -options fileName
```

`fileName` を、コンパイルする JSP ファイルの名前と置き換えます。対象の *`fileName`* の前か後に *`options`* を指定することもできます。次の例では、`-d` オプションを使用して `myFile.jsp` をコンパイルし、出力先ディレクトリ `weblogic\classes` に出力します。

```
$ java weblogic.jspc -d /weblogic/classes myFile.jsp
```

注意 : Web アプリケーションの一部で、Web アプリケーションのリソース (JSP タグ ライブラリなど) を参照している JSP をプリコンパイルする場合は、`-webapp` フラグを使用して Web アプリケーション の場所を指定する必要があります。`-webapp` フラグは JSP コンパイラ オプションの次のリストで説明します。

JSP コンパイラのオプション

以下のオプションを任意に組み合わせて使用することができます。

`-classpath`

希望する CLASSPATH となるディレクトリのリストを (Windows NT/2000 プラットフォームではセミコロンで区切り、UNIX プラットフォームではコロンで区切って) 追加します。JSP に必要なすべてのクラスを格納するディレクトリを含めます。次に例を示します (実際には 1 行で入力します)。

```
$ java weblogic.jspc
   -classpath java/classes.zip;/weblogic/classes.zip
   myFile.JSP
```

`-charsetMap`

JSP contentType ディレクティブで使用される IANA または非公式の文字セット名から java 文字セット名へのマッピングを指定します。次に例を示します。

```
-charsetMap x-sjis=Shift_JIS,x-big5=Big5
```

最も一般的なマッピングは、JSP コンパイラに組み込まれています。このオプションは、希望の文字セット マッピングが認識されない場合にのみ使用します。

`-commentary`

JSP コンパイラが、生成された HTML ページに JSP からのコメントを含めます。このオプションを省略すると、コメントは生成された HTML ページには表示されません。

`-compileAll`

カレント ディレクトリ、または -webapp オプションで指定したディレクトリ内のすべての JSP を再帰的にコンパイルします。(このオプション リストにある -webapp の説明を参照してください)。サブディレクトリ内の JSP もコンパイルされます。

`-compileFlags`

1 つまたは複数のコマンドライン フラグをコンパイラに渡します。複数のフラグはスペースで区切り、引用符で囲みます。次に例を示します。

```
java weblogic.jspc -compileFlags "-g -v" myFile.jsp
```

`-compiler`
: 生成された Java ソース コードからクラス ファイルをコンパイルする際に使用する Java コンパイラを指定します。デフォルトでは、コンパイラとして `javac` が使用されます。コンパイラの絶対パスを明示的に指定する場合を除いて、Java コンパイラ プログラムは `PATH` の下になければなりません。

`-compilerclass`
: Java コンパイラを、ネイティブ実行ファイルとしてではなく、Java クラスとして実行します。

`-d <dir>`
: コンパイルされた出力 (クラス ファイル) の出力先ディレクトリを指定します。このオプションは、コンパイルされたクラスを、既に `CLASSPATH` に入っているディレクトリに置くためのショートカットとして使用します。

`-depend`
: JSP の以前に生成されたクラス ファイルのタイム スタンプが、JSP ソース ファイルよりも新しい場合、JSP は再コンパイルされません。

`-debug`
: デバッグ モードでコンパイルします。

`-deprecation`
: 生成された Java ソース ファイルをクラス ファイルにコンパイルしているときに、ソース ファイル中で非推奨のメソッドの使用があるとこれを警告します。

`-docroot directory`
: `-webapp` を参照してください。

`-encoding default|named character encoding`
: 有効な引数は、(a) JDK のデフォルト文字エンコーディングの使用を指定する `default`、または (b) `8859_1` などの名前付き文字エンコーディングです。`-encoding` フラグが指定されない場合、バイト配列が使用されます。

`-g`
: クラス ファイルにデバッグ情報も入れるよう、Java コンパイラに指示します。

`-help`
: JSP コンパイラで使用可能なオプションをすべて表示します。

`-J`
コンパイラに渡されるオプションのリストを取得します。

`-k`
1 つのコマンドで 複数の JSP をコンパイルする場合、1 つまたは複数の JSP がコンパイルに失敗しても、コンパイルを続行します。

`-keepgenerated`
コンパイル処理の途中で作成された Java ソース コード ファイルを削除せずに残します。通常、このファイルはコンパイル後に削除されます。

`-noTryBlocks`
JSP ファイルに、多数のまたは深くネストしたカスタム JSP タグが含まれていて、コンパイル時に `java.lang.VerifyError` 例外が発生する場合、JSP を正しくコンパイルするためにこのフラグを使用します。

`-nowarn`
Java コンパイラからの警告メッセージが出力されないようにします。

`-noPrintNulls`
JSP の式の "null" が "" （空の文字列）として示されます。

`-O`
最適化スイッチをオンにして、生成された Java ソース ファイルをコンパイルします。このオプションは、`-g` オプションをオーバライドします。

`-package` *`packageName`*
生成された Java HTTP サーブレットのパッケージ名の前に追加するパッケージ名を設定します。デフォルトは `jsp_servlet` です。

`-superclass` *`classname`*
生成されたサーブレットによって拡張されたスーパークラスのクラス名を設定します。スーパクラスの名前は、`HttpServlet` または `GenericServlet` から派生したものでなければなりません。

`-verbose`
`compiler` フラグで指定された Java コンパイラに、`verbose` フラグを渡します。詳細については、コンパイラのマニュアルを参照してください。デフォルトはオフです。

`-verboseJavac`
指定した JSP コンパイラによって生成されたメッセージを出力します。

`-version`
JSP コンパイラのバージョンを出力します。

`-webapp` *`directory`*

展開ディレクトリ形式で Web アプリケーション が含まれるディレクトリ名。JSP タグ ライブラリやその他の Java クラスなど、Web アプリケーション内のリソースへの参照が JSP に含まれる場合、JSP コンパイラはそのリソースをこのディレクトリで探します。Web アプリケーションのリソースを要求する JSP をコンパイルする場合、このフラグを省略するとコンパイルに失敗します。

JSP のプリコンパイル

`weblogic.xml` デプロイメント記述子の `<jsp-descriptor>` 要素の `precompile` パラメータを true に設定することで、Web アプリケーションをデプロイまたは再デプロイしたとき、あるいは WebLogic Server を起動したときに、JSP をプリコンパイルするよう、WebLogic Server をコンフィグレーションできます。サーバの再起動時や対象のサーバの追加時に JSP を再コンパイルしないようにするには、weblogic.jspc を使用して JSP をプリコンパイルしてから、`WEB-INF/classes` フォルダに配置し、.war ファイルにアーカイブします。

`web.xml` デプロイメント記述子の詳細については、『Web アプリケーションのアセンブルとコンフィグレーション』を参照してください。

Windows NT のコマンド長制限は、WebLogic JSP の新しい `compilerclass` オプションを使用して克服できます。このオプションは、weblogic.xml ファイルでコンフィグレーションできます。

インメモリの `compilerclass` オプションは、Sun で使用されるコンパイラ クラスを使用して内部的に Java ファイルをコンパイルします。この方法は新しいプロセスを作成する必要がないので、新しいプロセスを別に使用して各 Java ファイルをコンパイルするよりも効率的です。

compilerclass は、weblogic.xml に以下の行を追加することで使用できます。

```
<jsp-descriptor>
 <jsp-param>
   <param-name>compilerclass</jsp-param>
   <param-value>com.sun.tools.javac.Main</param-value>
 </jsp-param>
</jsp-descriptor>
```

4 カスタム WebLogic JSP タグ (cache、process、repeat) の使い方

以下の節では、WebLogic Server 配布キットで提供される 3 つのカスタム JSP タグ (`cache` タグ、`repeat` タグ、`process` タグ) の使い方について説明します。

- WebLogic カスタム JSP タグの概要
- Web アプリケーションでの WebLogic カスタム タグの使い方
- cache タグ
- process タグ
- repeat タグ

WebLogic カスタム JSP タグの概要

BEA では、JSP ページで使用できる 3 つの特殊な JSP タグを提供しています。これらは、`cache`、`repeat`、および `process` です。これらのタグは、`weblogic-tags.jar` というタグ ライブラリ jar ファイルにパッケージ化されています。この jar ファイルには、タグのクラスとタグ ライブラリ記述子 (TLD) が含まれています。これらのタグを使用するには、JSP を格納する Web アプリケーションにこの jar ファイルをコピーして、タグ ライブラリを JSP で参照します。

Web アプリケーションでの WebLogic カスタム タグの使い方

WebLogic カスタム タグを使うには、タグを Web アプリケーションに入れる必要があります。詳細については、『Web アプリケーションのアセンブルとコンフィグレーション』を参照してください。

これらのタグを JSP で使用するには、次の手順に従います。

1. WebLogic Server の `ext` ディレクトリの `weblogic-tags.jar` ファイルを、WebLogic カスタム タグを使用する JSP を格納する Web アプリケーションの `WEB-INF/lib` ディレクトリにコピーします。
2. このタグ ライブラリ記述子を、Web アプリケーション デプロイメント記述子 `web.xml` の `<taglib>` 要素で参照します。次に例を示します。

   ```
   <taglib>
     <taglib-uri>weblogic-tags.tld</taglib-uri>
     <taglib-location>
        /WEB-INF/lib/weblogic-tags.jar
     </taglib-location>
   </taglib>
   ```

   詳細については、「Web アプリケーションのデプロイメント記述子の記述」を参照してください。
3. `taglib` ディレクティブを使用して、JSP 内でタグ ライブラリを参照します。次に例を示します。

   ```
   <%@ taglib uri="weblogic-tags.tld" prefix="wl" %>
   ```

cache タグ

cache タグは、タグの本体内で行われた処理のキャッシングを有効にします。このタグは、出力 (変換) データと入力 (計算済み) データの両方をサポートします。出力キャッシングとは、タグ内のコードによって生成されたコンテンツを指します。入力キャッシングとは、タグ内のコードによって設定される変数の値を

指します。出力キャッシングは、コンテンツの最終的な形式をキャッシュする必要がある場合に便利です。入力キャッシングは、タグ内で計算されるデータと無関係にデータ表示が変化する場合に有用です。

あるクライアントがキャッシュのコンテンツを再計算しているときに、別のクライアントが同じコンテンツを要求した場合には、再計算の完了を待つことなく、すでにキャッシュ内にある情報が表示されます。これは、キャッシュの再計算が原因で、すべてのユーザに対して Web サイトが停止することを回避するための仕組みです。さらに、async 属性を使用して、すべてのユーザ (キャッシュの再計算を開始したユーザも含む) に対して再計算の完了を待たないことを指定できます。

キャッシュは、キャッシング システムがシステム メモリを使いすぎないようにソフト参照を使って保存されます。ただし、HotSpot VM と Classic VM との間で互換性がないために、WebLogic Server が HotSpot VM 内で動作しているときは、ソフト参照が使用されません。

キャッシュの更新

`_cache_refresh` オブジェクトを `true` に設定することで、対象スコープ内で強制的にキャッシュを更新できます。セッション スコープでキャッシュを更新するには、次のように指定します。

```
<% request.setAttribute("_cache_refresh", "true"); %>
```

すべてのキャッシュを更新する場合は、キャッシュを `application` スコープに設定します。ユーザ向けのすべてのキャッシュを更新する場合は、キャッシュを `session` スコープに入れます。現在のリクエスト内のすべてのキャッシュを更新する場合は、`_cache_refresh` オブジェクトをパラメータとして設定するか、リクエスト内に設定します。

`<wl:cache>` タグは、表示されるたびに更新する必要のあるコンテンツを指定します。`<wl:cache>` タグと `</wl:cache>` タグで囲まれた文は、キャッシュが期限切れになるか、key 属性値 (表「cache タグの属性」を参照) が変更された場合にのみ実行されます。

キャッシュのフラッシュ

キャッシュをフラッシュするとキャッシュされた値が強制的に消去され、次にキャッシュがアクセスされたときに値が再計算されます。キャッシュをフラッシュするには、flush 属性を true に設定します。キャッシュには、name 属性を使用して名前を付ける必要があります。キャッシュに size 属性が設定されている場合は、すべての値がフラッシュされます。キャッシュに key 属性が設定されていて size 属性が設定されていない場合は、キャッシュをユニークに識別するために必要なその他の属性 (scope や vars など) とともにその key 属性を指定すると、特定のキャッシュをフラッシュできます。

次に例を示します。

1. キャッシュを定義します。

```
<wl:cache name="dbtable" key="parameter.tablename"
scope="application">
// テーブルを読み込み、ページに出力する
</wl:cache>
```

2. キャッシュされたテーブル データを更新します。

3. 空タグ (/ で終了し、終了タグを使用しない) で flush 属性を使用して、キャッシュをフラッシュします。次に例を示します。

```
<wl:cache name="dbtable" key="parameter.tablename"
scope="application" flush="true"/>
```

表 4-1 cache タグの属性

属性	必須	デフォルト値	説明
timeout	いいえ	-1	キャッシュ タイムアウト プロパティ。cache タグ内の文が更新されるまでの時間 (秒単位)。この属性はプロアクティブではなく、この値は要求時にのみリフレッシュされる。秒単位より時間単位の方がよい場合は、使用する単位を値の後に付けて指定できる。 ms = ミリ秒 s = 秒 (デフォルト) m = 分 h = 時間 d = 日

表 4-1 cache タグの属性

属性	必須	デフォルト値	説明
scope	いいえ	application	データをキャッシュするスコープを指定する。有効なスコープは次のとおり。 ■ page (ページを要求する) ■ parameter (パラメータを要求する) ■ request (リクエストを要求する) ■ requestHeader (要求ヘッダを要求する) ■ responseHeader (応答ヘッダを要求する) ■ session (セッションを要求する) ■ application (アプリケーションを要求する) ほとんどのキャッシュは、session または application となる。

表 4-1 cache タグの属性

属性	必須	デフォルト値	説明
key	いいえ	--	タグ内に格納されている値をキャッシュするかどうかを評価する際に使用する値を指定する。指定されたキャッシュは通常、web.xml でコンフィグレーションしたキャッシュ名によって識別される。キャッシュ名が指定されていない場合は、リクエスト URI がキャッシュ名として使用される。タグを識別するために、キーを使用して追加の値を指定できる。たとえば、指定したエンド ユーザ用にキャッシュを分離する場合、キャッシュ名に加えて、ユーザ ID とクライアント IP (不要な場合もある) としてキーを指定できる。ユーザ ID は、リクエスト パラメータ スコープからキャッシュを取り出すための値 (パラメータのクエリ / ポスト)。したがって「`parameter.userid,parameter.clientip`」としてキーを指定する。「parameter」はスコープ (リクエスト パラメータ スコープ) であり、「userid」と「clientip」はそれぞれパラメータ、属性である。つまり、キャッシュの主キーはキャッシュ名 (この場合はリクエスト URI) + ユーザ ID リクエスト パラメータの値 + クライアント IP リクエスト パラメータの値になる。 キーのリストはカンマで区切る。この属性の値はキーとしてキャッシュに入れる値を持つ変数の名前となる。スコープ名を名前の前に付加することで、さらにスコープを指定できる。次に例を示す。 `parameter.key | page.key | request.key | application.key | session.key` デフォルトでは、上のリストの順でスコープ内を検索する。名前を付けた各キーは、cache タグ内でスクリプト変数として使用可能となる。キーのリストはカンマで区切る。

表 4-1 cache タグの属性

属性	必須	デフォルト値	説明
async	いいえ	false	async パラメータを true に設定すると、可能であれば、キャッシュは非同期で更新される。キャッシュ ヒットを開始するユーザには古いデータが表示される。
name	いいえ	--	複数の JSP ページ間でキャッシュを共有可能とするユニークなキャッシュの名前。同じバッファが、名前を付けたキャッシュを使用するすべてのページのデータを格納するために使用される。この属性は、キャッシュを共有する必要があり、本文をインクルードされるページで有用。この属性が設定されていない場合、ユニークな名前がキャッシュに割り当てられる。 キーの機能はどんな場合にも同じように使用できるので、タグの名前を手動で算出することは避ける。名前の算出方法は、weblogic.jsp.tags.CacheTag に URI を加え、さらにキャッシュするページ内のタグを表す生成済み番号を加える。別々の URI が同じ JSP ページに達する場合、キャッシュはデフォルトでは共有されない。名前付きのキャッシュはこうした場合に使用する。
size	いいえ	-1 (無制限)	キーを使用するキャッシュごとに許可されるエントリの数。デフォルトでは、キーのキャッシュを制限しない。キーの数を制限すると、タグはキャッシュに指示するために最小使用頻度方式 (least-used system) を使用する。使用中のキャッシュの size 属性の値を変更しても、そのキャッシュのサイズは変わらない。

表 4-1 cache タグの属性

属性	必須	デフォルト値	説明
vars	いいえ	--	変換されたキャッシュの出力をキャッシュするだけでなく、算出された値もブロック内にキャッシュできる。これらの変数はキャッシュキーと正確に同じように指定する。このタイプのキャッシュを入力キャッシュという。 変数は入力キャッシングを行うために使用される。キャッシュが取得されると、変数は指定したスコープに復元される。たとえば、データベースからの結果を取得するために、リクエスト パラメータから var1、セッションから var2 を使用したとする。キャッシュが作成されると、これらの変数の値はキャッシュで格納される。次回キャッシュにアクセスすると、これらの値が復元されるので、それぞれのスコープから値にアクセスできるようになる。たとえば、var1 はリクエストから、var2 はセッションから使用できるようになる。
flush	いいえ	なし	true に設定すると、キャッシュがフラッシュされる。この属性は、空タグ (/ で終了) 内で設定する必要がある。

次の例では、<wl:cache> タグの使い方を示します。

コード リスト 4-1 cache タグの使い方の例

```
<wl:cache>
<!--the content between these tags will only be
 refreshed on server restart-->
</wl:cache>

<wl:cache key="request.ticker" timeout="1m">
<!--get stock quote for whatever is in the request parameter ticker
 and display it, only update it every minute-->
</wl:cache>

<!--incoming parameter value isbn is the number used to lookup the
 book in the database-->
<wl:cache key="parameter.isbn" timeout="1d" size="100">
<!--retrieve the book from the database and display
```

```
the information -- the tag will cache the top 100
most accessed book descriptions-->
</wl:cache>

<wl:cache timeout="15m" async="true">
<!--get the new headlines from the database every 15 minutes and
 display them-->
<!--do not let anyone see the pause while they are retrieved-->
</wl:cache>
```

process タグ

<wl:process> タグは、クエリ パラメータベースのフロー制御用に使用します。4 つの属性を組み合わせて使用することで、<wl:process> タグと </wl:process> タグに囲まれた文を選択的に実行できます。process タグは、フォームの送信結果を説明を付けて処理する場合にも使用できます。リクエスト パラメータの値を基に条件を指定することで、JSP 構文をページに含めるかどうかを指定できます。

表 4-2 process タグの属性

タグ属性	必須	説明
name	いいえ	クエリ パラメータの名前
notname	いいえ	クエリ パラメータの名前
value	いいえ	クエリ パラメータの値
notvalue	いいえ	クエリ パラメータの値

次の例では、<wl:process> タグの使い方を示します。

コード リスト 4-2 process タグの使い方の例

```
<wl:process notname="update">
<wl:process notname="delete">
<!--Only show this if there is no update or delete parameter-->
<form action="<%= request.getRequestURI() %>">
```

```
  <input type="text" name="name"/>
  <input type="submit" name="update" value="Update"/>
  <input type="submit" name="delete" value="Delete"/>
</form>
</wl:process>
</wl:process>

<wl:process name="update">
<!-- do the update -->
</wl:process>


<wl:process name="delete">
<!--do the delete-->
</wl:process>

<wl:process name="lastBookRead" value="A Man in Full">
<!--this section of code will be executed if lastBookRead exists
 and the value of lastBookRead is "A Man in Full"-->
</wl:process>
```

repeat タグ

<wl:repeat> タグは、列挙値、イテレータ、コレクション、オブジェクト配列、ベクトル、結果セット、結果セットメタデータ、およびハッシュテーブル キーなど、さまざまなタイプの集合に対して処理を繰り返す場合に使用します。また、count 属性を使用して、一定回数だけループすることもできます。Java オブジェクトのタイプを指定するには、set 属性を使用します。

表 4-3 repeat タグの属性

タグ属性	必須	型	説明
set	いいえ	Object	以下を含むオブジェクトの集合。 ■ 列挙値 ■ イテレータ ■ コレクション ■ 配列 ■ ベクトル ■ 結果セット ■ 結果セット メタデータ ■ ハッシュテーブル キー
count	いいえ	int	集合内で最初の *count* エントリに対して処理を繰り返す。
id	いいえ	String	反復処理の対象となる現在のオブジェクトを格納する変数。
type	いいえ	String	渡された集合に対して処理を繰り返して得られるオブジェクトの型。デフォルトは Object。この型は完全修飾しなければならない。

次の例では、<wl:repeat> タグの使い方を示します。

コード リスト 4-3　repeat タグの使い方の例

```
<wl:repeat id="name" set="<%= new String[] { "sam", "fred", "ed" }
%>">
  <%= name %>
</wl:repeat>

<% Vector v = new Vector();%>
<!--add to the vector-->

<wl:repeat id="item" set="<%= v.elements() %>">
<!--print each element-->
</wl:repeat>
```

5 WebLogic JSP フォーム検証タグの使い方

以下の節では、WebLogic JSP フォーム検証タグの使い方について説明します。

- WebLogic JSP フォーム検証タグの概要
- 検証タグ属性のリファレンス
- WebLogic JSP フォーム検証タグの JSP 内での使い方
- <wl:form> タグを使用した HTML フォームの作成
- カスタム バリデータ クラスの使い方
- バリデータ タグを使用した JSP のサンプル

WebLogic JSP フォーム検証タグの概要

WebLogic JSP フォーム検証タグは、JSP ページによって生成される HTML フォームのテキスト フィールドにユーザが入力するエントリを検証するための便利な方法を提供します。WebLogic JSP フォーム検証タグを使用すると、よく使用される検証ロジックを繰り返しコーディングする必要がなくなります。検証は、WebLogic Server 配布キットに含まれている複数のカスタム JSP タグによって行われます。JSP フォーム検証タグの機能は次のとおりです。

- 必須フィールドに値が入力されているかどうかを検証できます (`Required Field Validator` クラス)。
- フィールド内のテキストを正規表現と照らし合わせて検証できます (`Regular Expression Validator` クラス)。
- フォーム内の 2 つのフィールドを比較できます (`Compare Validator` クラス)。

- ユーザが記述した Java クラスによってカスタム検証を実行できます (`Custom Validator` クラス)。

WebLogic JSP フォーム検証タグには、以下のものがあります。

- <wl:summary>
- <wl:form>
- <wl:validator>

検証タグによってフィールド内のデータが正しく入力されていないことが判明した場合、そのページが再表示され、再入力が必要なフィールドがテキストまたは画像で示され、エンド ユーザに注意が促されます。フォームに正しく入力されると、エンド ユーザのブラウザには検証タグで指定された新しいページが表示されます。

検証タグ属性のリファレンス

この節では、WebLogic フォーム検証タグとその属性について説明します。タグを表すために使用されるプレフィックスは、JSP ページの `taglib` ディレクティブで定義できます。このマニュアルでは、理解しやすいように `wl` というプレフィックスを使用して WebLogic フォーム検証タグを表します。

<wl:summary>

`<wl:summary>` は、検証用の親タグです。`<wl:summary>` の開始タグは、JSP 内のすべての要素または HTML コードの前に置きます。終了タグの `</wl:summary>` は、終了タグ `</wl:form>` の後ろの任意の場所に置きます。

`name`

(省略可能) JSP ページの `<wl:validator>` タグによって生成されるすべての検証エラー メッセージを保持するベクトル変数の名前です。この属性を定義しなかった場合、デフォルトの `errorVector` が使用されます。エラー メッセージのテキストは、`<wl:validator>` タグの `errorMessage` 属性で定義されます。

このベクトル内の値を表示するには、<wl:errors/> タグを使用します。<wl:errors/> タグを使用するには、エラー出力を表示するページ上の場所にこのタグを配置します。次に例を示します。

```
<wl:errors color="red"/>
```

代わりに、スクリプトレットを使用することもできます。次に例を示します。

```
<% if (errorVector.size() > 0) {
   for (int i=0; i < errorVector.size(); i++) {
   out.println((String)errorVector.elementAt(i));
   out.println("<br>");
   }
} %>
```

ここで *errorVector* は、<wl:summary> タグの name 属性で割り当てられたベクトルの名前です。

name 属性は、1 つのページで複数のフォームを使用するときに必要となります。

headerText

ページに表示されるテキストを格納する変数です。ページでエラーが発生したときにのみこのテキストを表示させたい場合、スクリプトレットを使用してこの条件をテストできます。次に例を示します。

```
<% if(summary.size() >0 ) {
   out.println(headerText);
   }
%>
```

ここで *summary* は、<wl:summary> タグの name 属性で割り当てられたベクトルの名前です。

redirectPage

フォーム検証でエラーが返されない場合に表示されるページの URL です。この属性は、<wl:form> タグの action 属性で URL を指定する場合は必要ありません。

注意： redirectPage 属性を、<wl:summary> タグが配置されているページに設定しないでください。無限ループとなって StackOverFlow 例外が発生します。

<wl:form>

<wl:form> タグは HTML <form> タグとほぼ同じで、WebLogic JSP フォーム検証タグを使って検証できる HTML フォームを定義します。name 属性を使用して各フォームを一意に識別することによって、単一の JSP で複数のフォームを定義できます。

method

GET または POST を入力します。この機能は、HTML <form> タグの method 属性とまったく同じです。

action

フォーム検証でエラーが返されない場合に表示されるページの URL です。この属性の値は、<wl:summary> タグの redirectPage 属性の値に優先します。この属性は、単一の JSP ページに複数のフォームを定義する場合に便利です。

注意： action 属性を、<wl:form> タグが配置されているページに設定しないでください。無限ループとなって StackOverFlow 例外が発生します。

name

この機能は、HTML <form> タグの name 属性とまったく同じです。同じページで複数のフォームが使用される場合にフォームを識別します。また、name 属性はフォームへの JavaScript 参照にも便利です。

<wl:validator>

フォーム フィールドごとに、1 つまたは複数の <wl:validator> タグを使用します。たとえば、入力を正規表現と照らし合わせて検証し、フィールドに何かが入力されている必要がある場合、2 つの <wl:validator> タグを使用します。1 つは RequiredFieldValidator クラスを使用し、もう 1 つは RegExpValidator クラスを使用します(空白の値は ExpressionFieldValidator によって値として評価されるため、これらのバリデータを両方使用する必要があります)。

errorMessage

<wl:summary> タグの name 属性によって定義されるベクトル変数に格納される文字列です。

`expression`

`RegExpValidator` クラスを使用すると、正規表現が評価されるようになります。

`RegExpValidator` を使用しない場合、この属性は無視できます。

`fieldToValidate`

検証するフォーム フィールドの名前です。フィールド名は、HTML `<input>` タグの `name` 属性で定義されます。

`validatorClass`

検証ロジックを実行する Java クラスの名前です。3 つのクラスを使用できます。また、独自のバリデータ クラスを作成することもできます。詳細については、「カスタム バリデータ クラスの使い方」を参照してください。

使用できる検証クラスは以下のとおりです。

`weblogicx.jsp.tags.validators.RequiredFieldValidator`

テキストがフィールドに入力されているかどうかを検証します。

`weblogicx.jsp.tags.validators.RegExpValidator`

標準の正規表現を使用してフィールド内のテキストを検証します。

注意：空白の値も有効と評価されます。

`weblogicx.jsp.tags.validators.CompareValidator`

2 つのフィールドに同じ文字列が入力されているかどうかをチェックします。このクラスを使用する場合、`fieldToValidate` 属性をこれらの 2 つのフィールドに設定します。次に例を示します。

```
fieldToValidate="field_1,field_2"
```

注意：両方のフィールドが空白の場合でも、比較は有効と評価されます。

`myPackage.myValidatorClass`

カスタム バリデータ クラスを指定します。

WebLogic JSP フォーム検証タグの JSP 内での使い方

検証タグを JSP で使用するには、次の手順に従います。

1. JSP を記述します。

   a. `taglib` ディレクティブを入力して、WebLogic JSP フォーム検証タグ が存在するタグ ライブラリを参照します。次に例を示します。

      ```
      <%@ taglib uri="tagl" prefix="wl" %>
      ```

      プレフィックス属性では、JSP ページのすべてのタグを参照するために使用されるプレフィックスを定義します。プレフィックスは任意の値に設定できますが、このマニュアルでは `wl` というプレフィックスを使用してタグが参照されます。

   b. `<wl:summary> ... </wl:summary>` タグを入力します。

      開始タグの `<wl:summary ...>` を、ページ上のすべての HTML コード、JSP タグ、スクリプトレット、または式の前に配置します。

      終了タグの `</wl:summary>` を、`</wl:form>` の後ろの任意の場所に置きます。

   c. 付属のタグ ライブラリに登録されている `<wl:form>` JSP タグを使用して、HTML フォームを定義します。詳細については、「<wl:form>」と「<wl:form> タグを使用した HTML フォームの作成」を参照してください。フォーム ブロックは必ず `</wl:form>` タグを使用して閉じてください。フォームごとに `<wl:form>` タグの `name` 属性をユニークに定義する場合、単一のページに複数のフォームを作成できます。

   d. HTML `<input>` タグを使用して、HTML フォームのフィールドを作成します。

   e. `<wl:validator>` タグを追加します。このタグの構文については、「<wl:validator>」を参照してください。エラー メッセージまたは画像を表示するページ上の場所に `<wl:validator>` タグを配置します。同じページで複数のフォームを使用する場合、`<wl:validator>` タグを、検証するフォーム フィールドが存在する `<wl:form>` ブロックの中に配置します。

      次の例は、必須フィールドの検証を示したものです。

```
<wl:form name="FirstForm" method="POST" action="thisJSP.jsp">

<wl:validator
 errorMessage="Field_1 is required" expression=""
 fieldToValidate="field_1"
 validatorClass=
  "weblogicx.jsp.tags.validators.RequiredFieldValidator"
>
  <img src="images/warning.gif">
  <font color=red>Field 1 is a required field</font>
</wl:validator>

<p> <input type="text" name = "field_1">  </p>
<p> <input type="text" name = "field_2">  </p>
<p> <input type="submit" value="Submit FirstForm">  </p>
</wl:form>
```

ユーザが field_1 に値を入力しなかった場合は、ページが再表示され、warning.gif 画像と「Field 1 is a required field」というテキスト (赤色) が表示され、値を再入力するための空白フィールドが続いて表示されます。

2. WebLogic Server の ext ディレクトリの weblogic-vtags.jar ファイルを、使用する Web アプリケーションの WEB-INF\lib ディレクトリにコピーします。このディレクトリは作成が必要な場合があります。

3. Web アプリケーションの web.xml デプロイメント記述子に <taglib> 要素を追加することによって、Web アプリケーションがこのタグ ライブラリを使用するようコンフィグレーションします。次に例を示します。

```
<taglib>
  <taglib-uri>tagl</taglib-uri>
  <taglib-location>
    /WEB-INF/lib/weblogic-vtags.jar
  </taglib-location>
</taglib>
```

Web アプリケーションのデプロイメント記述子の詳細については、「Web アプリケーションのデプロイメント記述子の記述」を参照してください。

<wl:form> タグを使用した HTML フォームの作成

この節では、JSP ページに HTML フォームを作成する方法について説明します。ページに 1 つまたは複数のフォームを作成するには、<wl:form> タグを使用します。

単一のフォームの定義

weblogic-vtags.jar タグ ライブラリに含まれている <wl:form> タグを使用します。次に例を示します。

```
<wl:form method="POST" action="nextPage.jsp">
<p> <input type="text" name ="field_1">  </p>
<p> <input type="text" name ="field_2">  </p>
<p> <input type="submit" value="Submit Form">  </p>
</wl:form>
```

このタグの構文については、「<wl:form>」を参照してください。

複数のフォームの定義

ページで複数のフォームを使用する場合は、name 属性を使用して各フォームを識別します。次に例を示します。

```
<wl:form name="FirstForm" method="POST" action="thisJSP.jsp">
<p> <input type="text" name="field_1">  </p>
<p> <input type="text" name="field_2">  </p>
<p> <input type="submit" value="Submit FirstForm">  </p>
</wl:form>

<wl:form name="SecondForm" method="POST" action="thisJSP.jsp">
<p> <input type="text" name="field_1">  </p>
<p> <input type="text" name="field_2">  </p>
<p> <input type="submit" value="Submit SecondForm">  </p>
</wl:form>
```

検証によってエラーが返されたときにフィールド内の値を再表示する

バリデータ タグによってエラーが発見され、JSP ページが再表示されたときに、ユーザが入力した値を再表示して、フォーム全体を再び入力する必要がないようにしておくと便利です。この場合、HTML <input> タグの value 属性を使用するか、Apache Jakarta プロジェクトから使用できるタグ ライブラリを使用します。次に、これらの属性について説明します。

<input> タグを使用した値の再表示

javax.servlet.ServletRequest.getParameter() メソッドと HTML <input> タグの value 属性を一緒に使用すると、検証の失敗によってページが再表示されるときにユーザの入力を再表示できます。次に例を示します。

```
<input type="text" name="field_1"
 value="<%= request.getParameter("field_1") %>" >
```

クロスサイト スクリプトのセキュリティ脆弱性を回避するには、ユーザ提供のデータ内の HTML 特殊文字を HTML エンティティ参照と置き換えます。詳細については、「JSP におけるユーザ入力データのセキュリティ」を参照してください。

Apache Jakarta <input:text> タグを使用した値の再表示

Apache Jakarta プロジェクトの無償で使用できる JSP タグ ライブラリを使用することもできます。このタグ ライブラリには、HTML <input> タグの代わりとして <input:text> タグが用意されています。たとえば、次の HTML タグがあるとします。

```
<input type="text" name="field_1">
```

このタグは、Apache タグ ライブラリを使用すると次のように入力できます。

```
<input:text name="field_1">
```

詳細とマニュアルについては、「Input Tag library」を参照してください。

Apache タグ ライブラリを JSP で使用するには、次の手順に従います。

1. Input Tag Library 配布ファイルに含まれている input.jar ファイルを Web アプリケーションの WEB-INF\lib ディレクトリにコピーします。
2. 次のディレクティブを JSP に追加します。

   ```
   <%@ taglib uri="input" prefix="input" %>
   ```

3. 次のエントリを、Web アプリケーションの web.xml デプロイメント記述子に追加します。

   ```
   <taglib>
       <taglib-uri>input</taglib-uri>
       <taglib-location>/WEB-INF/lib/input.jar</taglib-location>
   </taglib>
   ```

カスタム バリデータ クラスの使い方

独自のバリデータ クラスを使用するには、次の手順に従います。

1. weblogicx.jsp.tags.validators.CustomizableAdapter 抽象クラスを拡張する Java クラスを記述します。詳細については、「CustomizableAdapter クラスの拡張」を参照してください。
2. validate() メソッドを実装します。このメソッドでは、次のことを行います。
   a. 検証するフィールドの値を ServletRequest オブジェクトからルックアップします。次に例を示します。

      ```
      String val = req.getParameter("field_1");
      ```

   b. フィールドが検証条件を満たしている場合、true を返します。
3. バリデータ クラスをコンパイルし、その .class ファイルを Web アプリケーションの WEB-INF\classes ディレクトリに格納します。
4. バリデータ クラス名を validatorClass 属性に指定することによって、そのクラスを <wl:validator> タグで使用します。次に例を示します。

   ```
   <wl:validator errorMessage="This field is required"
   fieldToValidate="field_1"
   validatorClass="mypackage.myCustomValidator">
   ```

CustomizableAdapter クラスの拡張

CustomizableAdapter クラスは、Customizable インタフェースを実装する抽象クラスで、以下のヘルパー メソッドを提供します。

getFieldToValidate()
: 検証するフィールド (<wl:validator> タグの fieldToValidate 属性によって定義される) の名前を返します。

getErrorMessage()
: <wl:validator> タグの errorMessage 属性で定義されるエラー メッセージのテキストを返します。

getExpression()
: <wl:validator> タグに定義される expression 属性のテキストを返します。

CustomizableAdapter クラスを拡張する代わりに、Customizable インタフェースを実装することができます。詳細については、weblogicx.jsp.tags.validators.Customizable の Javadoc を参照してください。

カスタム バリデータ クラスのサンプル

コード リスト 5-1 カスタム バリデータ クラスの例

```
import weblogicx.jsp.tags.validators.CustomizableAdapter;

public class myCustomValidator extends CustomizableAdapter{

    public myCustomValidator(){
       super();
    }

    public boolean validate(javax.servlet.ServletRequest req)
       throws Exception {
       String val = req.getParameter(getFieldToValidate());
        // 検証ロジックを実行
        // 検証が成功した場合は true を返す
         // それ以外の場合は false を返す

       if (true) {
           return true;
       }
```

```
        return false;
    }
}
```

バリデータ タグを使用した JSP のサンプル

このサンプル コードには、WebLogic JSP フォーム検証タグを使用する JSP の基本的な構造が示されています。WebLogic Server と一緒にサンプルをインストールした場合は、完全に機能するサンプル コードも使用できます。サンプル コードの実行手順については、インストールした WebLogic Server の `samples\examples\jsp\tagext\form_validation\package.html` を参照してください。

コード リスト 5-2 WebLogic JSP フォーム検証タグと JSP

```
<%@ taglib uri="tagl" prefix="wl" %>
<%@ taglib uri="input" prefix="input" %>


<wl:summary
name="summary"
headerText="<font color=red>Some fields have not been filled out
correctly.</font>"
redirectPage="successPage.jsp"
>

<html>
<head>
<title>Untitled Document</title>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1">
</head>

<body bgcolor="#FFFFFF">


<% if(summary.size() >0 )   {
    out.println("<h3>" + headerText + "</h3>");
} %>


<% if (summary.size() > 0) {
```

```
        out.println("<H2>Error Summary:</h2>");
        for (int i=0; i < summary.size(); i++) {
        out.println((String)summary.elementAt(i));
        out.println("<br>");
        }
} %>


<wl:form method="GET" action="successPage.jsp">

 User Name: <input:text name="username"/>
 <wl:validator
   fieldToValidate="username"

validatorClass="weblogicx.jsp.tags.validators.RequiredFieldValida
tor"
   errorMessage="User name is a required field!"
 >
   <img src=images/warning.gif> This is a required field!
 </wl:validator>

<p>

 Password: <input type="password" name="password">
 <wl:validator
   fieldToValidate="password"

validatorClass="weblogicx.jsp.tags.validators.RequiredFieldValida
tor"
   errorMessage="Password is a required field!"
 >
   <img src=images/warning.gif> This is a required field!
 </wl:validator>

 <p>

 Re-enter Password: <input type="password" name="password2">
 <wl:validator
   fieldToValidate="password,password2"
   validatorClass="weblogicx.jsp.tags.validators.CompareValidator"
   errorMessage="Passwords don't match"
 >
   <img src=images/warning.gif> Passwords don't match.
 </wl:validator>

 <p>

 <input type="submit" value="Submit Form">  </p>

</wl:form>



</wl:summary>
```

```
</body>
</html>
```

6 WebLogic EJB-to-JSP 統合ツールの使い方

以下の節では、WebLogic EJB-to-JSP 統合ツールを使用して、JavaServer Page (JSP) で EJB を呼び出すための JSP タグ ライブラリを作成する方法について説明します。ここでは、読者が少なくとも EJB と JSP の両方についてある程度の知識を持っていることを前提としています。

- WebLogic EJB-to-JSP 統合ツールの概要
- 基本的な処理
- インタフェース ソース ファイル
- ［Build Options］パネル
- トラブルシューティング
- JSP ページでの EJB タグの使い方
- EJB ホーム メソッド
- ステートフル セッション Bean とエンティティ Bean
- デフォルト属性

WebLogic EJB-to-JSP 統合ツールの概要

EJB jar ファイルを使用する場合、WebLogic EJB-to-JSP 統合ツールで JSP タグ拡張ライブラリを生成します。このライブラリのタグは、その jar ファイルの EJB を呼び出すようにカスタマイズされます。クライアントから見た場合、EJB はそのリモート インタフェースによって表されます。次に例を示します。

```
public interface Trader extends javax.ejb.EJBObject {
  public TradeResult buy(String stockSymbol, int shares);
```

```
  public TradeResult sell(String stockSymbol, int shares);
}
```

EJB を呼び出す Web アプリケーションの場合、一般的なモデルは JSP スクリプトレット (<% ... %>) 内から Java コードを使用して EJB を呼び出すことです。EJB 呼び出しの結果は HTML としてフォーマット化され、Web クライアントに提供されます。このアプローチは面倒で、エラーを引き起こしやすくなります。EJB の呼び出しに必要な Java コードは最も単純なケースでも非常に長く、HTML 表現を担当するほとんどの設計者のスキルを超えています。

EJB-to-JSP ツールは、Java コードの必要性を排除することによって、EJB 呼び出しプロセスを簡素化します。代わりに、EJB 用にカスタム生成される JSP タグライブラリを使用して、EJB を呼び出します。たとえば、上の Trader Bean のメソッドは、次のように JSP で呼び出されます。

```
<%@ taglib uri="/WEB-INF/trader-tags.tld" prefix="trade" %>
<b>invoking trade: </b><br>

<trade:buy stockSymbol="BEAS" shares="100"/>

<trade:sell stockSymbol="MSFT" shares="200"/>
```

この結果として生成される JSP ページは、より明確で直感的に理解できます。タグは EJB の各メソッドに対して (オプションで) 生成されます。これらのタグは、対応する EJB メソッド呼び出し用のパラメータに変換される属性を取ります。EJB 呼び出しの面倒な機構は隠され、生成されるタグ ライブラリのハンドラ コードの内部にカプセル化されます。生成されるタグ ライブラリは、ステートレス セッション Bean とステートフル セッション Bean、およびエンティティ Bean をサポートします。これらのケースでは、タグの使用方法にそれぞれ若干の相違があります (後述の説明を参照)。

基本的な処理

WebLogic EJB-to-JSP 統合ツールは、コマンドライン モードで、次のコマンドを使用して実行できます。

```
  java weblogic.servlet.ejb2jsp.Main
```

またはグラフィカル モードで実行できます。最も単純な EJB を除き、グラフィカル ツールの方が便利です。

グラフィカル ツールは、次のように起動します。

```
java weblogic.servlet.ejb2jsp.gui.Main
```

最初は、Web アプリケーションによって ejb2jsp プロジェクトがロードされません。新しいプロジェクトを作成するには、[**File** | **New**] メニューを選択し、ファイル選択欄を参照して EJB jar ファイルを見つけ、そのファイルを選択します。初期化が済んだら、ejb2jsp プロジェクトを修正、保存、および再ロードできます。

生成されるタグ ライブラリの構成は単純です。jar ファイル内の各 EJB のメソッドごとに、そのメソッドと同名の JSP タグが生成されます。各タグには、対応するメソッドのパラメータと同数の属性を指定できます。

インタフェース ソース ファイル

新しい EJB jar がロードされると、統合ツールは EJB のホーム インタフェースとリモート インタフェース用の Java ソース ファイルを見つけようとします。これは、変換ツールは EJB クラスを参照するだけでタグを生成できますが、それらのタグ (対応する EJB メソッドがパラメータを取る) に意味のある属性名を割り当てることができないからです。1 ページの「WebLogic EJB-to-JSP 統合ツールの概要」の **Trader** の例では、EJB jar がロードされると、変換ツールは **Trader.java** というソース ファイルを見つけようとします。このファイルは解析され、**buy()** メソッドが **stockSymbol** および **shares** パラメータを取ることを検出します。これにより、対応する JSP タグは適切な名前が付けられた属性 (**buy()** メソッドのパラメータに対応する) を持つようになります。

新しい EJB jar がロードされると、変換ツールは、EJB jar が存在するディレクトリとソース ディレクトリが同じであるという前提で動作します。この前提に当てはまらない場合でも、エラーは致命的ではありません。新しいプロジェクトのロード後に、[Project Build Options] パネルで、[EJB Source Path] 要素を調節して適切なディレクトリを指定できます。次に、[File | Resolve Attributes] メニューを選択して、解決プロセスを再実行します。

インタフェース クラスに対応する java ソース ファイルを探すときに、統合ツールは指定されたディレクトリと、そのインタフェースの Java パッケージによって示唆されるサブディレクトリの両方を検索します。たとえば、**my.ejb.Trader** の場合、指定されたディレクトリが **C:/src** であれば、統合ツールは **C:/src/Trader.java** と **C:/src/my/ejb/Trader.java** の両方を検索します。

ソース ファイルへのアクセスは必須ではありません。統合ツールを使用することにより、プロジェクトの各タグの属性名を常に変更できます。しかし、EJB のパブリック インタフェースのソース ファイルの解析は、意味のある属性名を割り当てるための最も迅速な方法として開発されました。

［Build Options］パネル

このパネルは、プロジェクトの構築に必要なローカル ファイル システムに関連するすべてのパラメータを設定するために使用します。Java コンパイラ、生成される JSP タグ ハンドラの Java パッケージ、およびプロジェクト構築後に生成される Java コードを保持するかどうか (デバッグに役立つ) を指定します。

また、このパネルでは、タグ ライブラリ出力の種類を指定できます。**J2EE Web** アプリケーションの場合、タグ ライブラリは次の 2 通りの方法のいずれかでパッケージ化する必要があります。1 つは別個のクラス ファイルとタグ ライブラリ記述子 (.tld) ファイルで、もう 1 つは単一の taglib jar ファイルです。いずれかの出力タイプを、［Output Type］プルダウン メニューで選択します。開発およびテストのために、**DIRECTORY** 出力をお勧めします。jar ファイルは、WebLogic Server の Web アプリケーションを再配布しなければ上書きできないからです。

DIRECTORY または **JAR** のいずれかについて、出力先を適切に選択して、タグ ライブラリが Web アプリケーションによって検索されるようにします。たとえば、ディレクトリ **C:\mywebapp** をルートとする Web アプリケーションでタグ ライブラリを使用する場合、［DIRECTORY classes］フィールドには次のように指定する必要があります。

```
C:/mywebapp/WEB-INF/classes
```

また、［**DIRECTORY .tld File**］フィールドには次のように指定する必要があります。

```
C:/mywebapp/WEB-INF/trader-ejb.tld
```

前述の**ソース パス**も［**Build Options**］パネルで編集します。タグ ライブラリがコア WebLogic Server または J2EE API に存在しない他のクラスに依存する場合は、［**Extra Classpath**］フィールドを使用できます。通常、このフィールドには何も追加する必要はありません。

トラブルシューティング

エラーや衝突によって、プロジェクトの構築に失敗する場合があります。この節では、これらのエラーの理由と解決方法について説明します。

- **構築情報の欠落。**［Build Options］パネル内の必須フィールドの 1 つに、java コンパイラ、コード パッケージ名、出力の保存先ディレクトリなどの値が指定されていない場合です。未指定のフィールドに値を指定しないと、構築を正常に行うことができません。

- **タグ名の重複。**EJB jar がロードされると、統合ツールは EJB のメソッドごとにタグを記録します。このタグ名は、メソッド名と同じです。EJB がオーバーロード メソッド (同名だがシグネチャが異なるメソッド) を持っている場合、タグ名が衝突します。この衝突を解決するには、これらのタグのいずれかの名前を変更するか、これらのタグの 1 つを無効にします。タグ名を変更するには、統合ツールの左ウィンドウのツリー階層を使用して問題のタグに移動します。右ウィンドウに表示されるタグ パネルで、［**Tag Name**］フィールドを変更します。タグを無効にするには、統合ツールの左ウィンドウのツリー階層を使用して問題のタグに移動します。右ウィンドウに表示されるタグ パネルで、［**Generate Tag**］ボックスのチェックをはずします。EJB jar に複数の EJB が含まれている場合、Bean 全体についてタグを無効にすることもできます。

- **意味のない属性名 arg0、arg1...。**このエラーは、タグの妥当な属性名が EJB のインタフェース ソース ファイルから推測できない場合に発生します。このエラーを修復するには、プロジェクトの階層ツリーで問題のタグに移動します。次に、そのタグの下にある各属性ツリー リーフを選択します。各属性について、ツールの右側に表示されるパネルの ［Attribute Name］フィールドに適切な名前を入力します。

- **属性名の重複。**これは、複数の属性を指定できる 1 つのタグに同名の属性が 2 つ存在する場合に発生します。問題の属性に移動して、そのタグでそれぞれユニークになるよう名前を変更します。

JSP ページでの EJB タグの使い方

生成された EJB タグを JSP ページで使用するには、そのページでタグ ライブラリを宣言し、他のタグ拡張と同じようにそのタグを呼び出します。

```
<% taglib uri="/WEB-INF/trader-ejb.tld"
 prefix="trade" %>
<trade:buy stockSymbol="XYZ" shares="100"/>
```

void 以外の戻り値の型を持つ EJB メソッドのために、「_return」という特別なオプションのタグが組み込まれています。このタグが存在する場合、このメソッドから返される値は正体の処理のためにページ上で使用できます。

```
<% taglib uri="/WEB-INF/trader-ejb.tld"
  prefix="trade" %>
<trade:buy stockSymbol="XYZ"
 shares="100" _return="tr"/>
<% out.println("trade result:" + tr.getShares()); %>
```

プリミティブ数値型を返すメソッドの場合、返される変数は、その型に対応する Java オブジェクトです (「int」-> java.lang.Integer、など)。

EJB ホーム メソッド

EJB 2.0 では、EJB ホーム インタフェースのメソッド (**create()** または **find()** 以外) を使用できます。これらのホーム インタフェース用のタグも生成されます。混乱を避けるため、新規プロジェクトがロードされるときに、EJB のホームの各メソッドに対応するタグの先頭に「**home-**」という文字列が付加されます。必要な場合は、これらのメソッドの名前を変更できます。

ステートフル セッション Bean とエンティティ Bean

「ステートフル」Bean の一般的な用途は、Bean のホーム インタフェースからそのインスタンスを取得し、単一の Bean インスタンスの複数のメソッドを呼び出すことです。このプログラミング モデルも、生成されるタグ ライブラリに保存できます。ステートフル EJB メソッド用のメソッド タグは、EJB ホーム インタフェースの (そのホームの **find()** または **create()** に対応する) タグの内部に配置する必要があります。find/create タグの内部に存在するすべての EJB メソッド タグは、終了タグによって検索または作成される Bean インスタンスで処理されます。ステートフル Bean のメソッド タグがそのホームの find/create タグで閉じられていない場合、実行時例外が発生します。たとえば、次の EJB が存在するとします。

```
public interface AccountHome extends EJBHome {

  public Account create(String accountId, double initialBalance);
  public Account findByPrimaryKey(String accountID);
    /* 残高がしきい値以上の口座をすべて検索する */
  public Collection findBigAccounts(double threshold);
}

public interface Account extends EJBObject {
  public String getAccountID();
  public double deposit(double amount);
  public double withdraw(double amount);
  public double balance();
}
```

この場合、正しいタグの使い方は次のようになります。

```
<% taglib uri="/WEB-INF/account-ejb.tld" prefix="acct" %>
<acct:home-create accountId="103"
  initialBalance="450.0" _return="newAcct">
 <acct:deposit amount="20"/>
 <acct:balance _return="bal"/>
 Your new account balance is: <%= bal %>
</acct:home-create>
```

find/create タグで「_return」属性が指定されている場合、検索 / 作成された EJB インスタンスを参照するページ変数が作成されます。エンティティ Bean の検索メソッドも、EJB インスタンスのコレクションを返す場合があります。Bean の

コレクションを返すメソッドを呼び出すホーム タグは、そのコレクションの Bean ごとにタグ本体で繰り返されます。「_return」が指定されている場合、それはその繰り返しの内部の現在の Bean に設定されます。

```
<b>Accounts above $500:</b>
<ul>
<acct:home-findBigAccounts threshold="500" _return="acct">
<li>Account <%= acct.getAccountID() %>
     has balance $<%= acct.balance() %>
</acct:home-findBigAccounts>
</ul>
```

上の例では、残高が $500 を超えるすべての口座 Bean の HTML リストが表示されます。

デフォルト属性

デフォルトでは、各メソッドのタグでは、そのすべての属性 (メソッド パラメータ) が各タグ インスタンスに設定される必要があります。しかし、統合ツールでは、JSP タグに属性が指定されていない場合に備えて、「デフォルト」のメソッド パラメータを指定することもできます。デフォルトの属性 / パラメータは、EJB-to-JSP ツールの［**Attribute**］ウィンドウで指定できます。パラメータのデフォルトは、単純な **EXPRESSION** から取得されるか、より複雑な処理が必要な場合、デフォルト **METHOD** 本文が記述されます。たとえば、1 ページの「WebLogic EJB-to-JSP 統合ツールの概要」の Trader の例で、何も指定されていない場合、株式シンボルの「XYZ」に対して「buy」タグを動作させたいとします。この場合、「buy」タグの「stockSymbol」属性の属性パネルで、[Default Attribute Value] フィールドを **EXPRESSION** に設定し、[**Default Expression**] フィールドに "XYZ" (引用符を含む) と入力します。この結果、buy タグは stockSymbol="XYZ" 属性が存在するかのように動作します (他の値が指定されていない場合)。

また、「buy」タグの株式属性を 0 ～ 100 のランダムな数値にしたい場合、[Default Attribute Value] を **METHOD** に設定し、[**Default Method Body**] 領域に、int 型 (「buy」メソッドの「shares」属性に対応する型) を返す Java メソッドの本文を記述します。

```
long seed = System.currentTimeMillis();
java.util.Random rand = new java.util.Random(seed);
int ret = rand.nextInt();
/* 正の数であることを確認 ...*/
```

```
ret = Math.abs(ret);
/* 100 より小さいことを確認 */<
return ret % 100;
```

デフォルトのメソッド本文は JSP タグ ハンドラの内部に現れるため、作成するコードは **pageContext** 変数にアクセスできます。SP PageContext から、現在の HttpServletRequest または HttpSession へのアクセスを取得し、セッション データまたはリクエスト パラメータを使用してデフォルト メソッド パラメータを生成できます。たとえば、ServletRequest パラメータから「buy」メソッドの「shares」パラメータを取得するには、次のコードを記述します。

```
HttpServletRequest req =
 (HttpServletRequest)pageContext.getRequest();
String s = req.getParameter("shares");
if (s == null) {
  /* webapp エラー ハンドラが、この例外のエラー ページに
   * リダイレクトされる
   */
  throw new BadTradeException("no #shares specified");
}
int ret = -1;
try {
  ret = Integer.parseInt(s);
} catch (NumberFormatException e) {
  throw new BadTradeException("bad #shares: " + s);
}
if (ret <= 0)
  throw new BadTradeException("bad #shares: " + ret);
return ret;
```

生成されるデフォルト メソッドは例外を送出するものと見なされます。処理中に発生した例外は、JSP の errorPage によって処理されるか、Web アプリケーションの登録済み例外処理ページによって処理されます。

7　トラブルシューティング

以下の節では、JSP ファイルのデバッグ テクニックをいくつか説明します。

- ブラウザ内のデバッグ情報
- ログ ファイルに見られる兆候

ブラウザ内のデバッグ情報

JSP ページをデバッグする際に、最も有用な機能は、デフォルトでブラウザに送られる出力です。これは、生成された HTTP サーブレット Java ファイル内のエラーの場所、エラーの説明、およびそのエラーコードに対応する元の JSP ファイルの大体の場所を表示してくれます。たとえば、コンパイルに失敗すると、ブラウザに次のメッセージが表示されます。

この機能を使用不可にするには、Web アプリケーションの WebLogic 固有のデプロイメント記述子の jsp-descriptor 要素で、`verbose` 属性を `false` に設定します。

Error 404—Not Found

JSP ファイルの URL を正しく入力したかどうか、URL が Web アプリケーションのルート ディレクトリの相対パスであるかどうかを確認します。

Error 500—Internal Server Error

WebLogic Server ログ ファイルのエラー メッセージをチェックします。3 ページの「ページのコンパイルに失敗した場合のエラー」を参照してください。一般に、このエラーは、JSP のコンパイル時に `ClasssNotFound` 例外が発生したことを示します。

Error 503—Service Unavailable

WebLogic Server が JSP をコンパイルするために必要なコンパイラを見つけられないことを示します。JSP コンパイラの定義については、「jsp-descriptor」の節を参照してください。

<jsp:plugin> タグの使用に関するエラー

JSP 内で `<jsp:plugin>` タグを使用し、アプレットをロードできない場合は、タグの構文を詳しくチェックします。生成された HTML ページを調べると、構文エラーがないかどうかチェックできます。ページ内のどこかに `<jsp:plugin ...` と表示されている場合は、タグの構文が間違っています。

ログ ファイルに見られる兆候

この節では、WebLogic Server ログ ファイル内の JSP 関連エラー メッセージについて説明します。WebLogic Server を実行すると、verbose メッセージが WebLogic ログ ファイルに保存されます。WebLogic ログ ファイルの詳細については、「ログ メッセージを使用した WebLogic Server の管理」を参照してください。

ページのコンパイルに失敗した場合のエラー

以下のエラーは、JSP コンパイラが JSP ページから Java ファイルへの変換に失敗した場合、あるいは、生成された Java ファイルをコンパイルできなかった場合に発生することがあります。ログ ファイル中の以下のエラー メッセージを確認してください。

`CreateProcess: ...`

Java コンパイラが見つからなかったか、または有効な実行可能ファイルではなかったことを示しています。Java コンパイラの指定については、「jsp-descriptor」の節を参照してください。

`Compiler failed:`

JSP ページから生成された Java コードが、指定した Java コンパイラでコンパイルできなかったことを示しています。JSP コンパイラだけを単独で使用し、生成された Java コードをより詳しく検査したりデバッグしたりできます。詳細については、3-20 ページの「WebLogic JSP コンパイラの使い方」を参照してください。

`Undefined variable or classname:`

カスタム変数を使用している場合は、スクリプトレットで使用する前に定義するか、あるいはそれを宣言で定義しておくようにします。宣言から暗黙的オブジェクトを使おうとすると、このエラーになる場合があります。宣言内での暗黙的オブジェクトの使用は、JSP ではサポートされていません。

索引