



BEA WebLogic Server™ およ び WebLogic Express®

WebLogic HTTP サー
ブレット プログラ
マーズ ガイド

著作権

Copyright © 2002, BEA Systems, Inc. All Rights Reserved.

限定的権利条項

本ソフトウェアおよびマニュアルは、BEA Systems, Inc. 又は日本ビー・イー・エー・システムズ株式会社（以下、「BEA」といいます）の使用許諾契約に基づいて提供され、その内容に同意する場合にのみ使用することができ、同契約の条項通りにのみ使用またはコピーすることができます。同契約で明示的に許可されている以外の方法で同ソフトウェアをコピーすることは法律に違反します。このマニュアルの一部または全部を、BEA からの書面による事前の同意なしに、複写、複製、翻訳、あるいはいかなる電子媒体または機械可読形式への変換も行うことはできません。

米国政府による使用、複製もしくは開示は、BEA の使用許諾契約、および FAR 52.227-19 の「Commercial Computer Software-Restricted Rights」条項のサブパラグラフ (c)(1)、DFARS 252.227-7013 の「Rights in Technical Data and Computer Software」条項のサブパラグラフ (c)(1)(ii)、NASA FAR 補遺 16-52.227-86 の「Commercial Computer Software--Licensing」条項のサブパラグラフ (d)、もしくはそれらと同等の条項で定める制限の対象となります。

このマニュアルに記載されている内容は予告なく変更されることがあり、また BEA による責務を意味するものではありません。本ソフトウェアおよびマニュアルは「現状のまま」提供され、商品性や特定用途への適合性を始めとする（ただし、これらには限定されない）いかなる種類の保証も与えません。さらに、BEA は、正当性、正確さ、信頼性などについて、本ソフトウェアまたはマニュアルの使用もしくは使用結果に関していかなる確約、保証、あるいは表明も行いません。

商標または登録商標

BEA、Jolt、Tuxedo、および WebLogic は BEA Systems, Inc. の登録商標です。BEA Builder、BEA Campaign Manager for WebLogic、BEA eLink、BEA Manager、BEA WebLogic Commerce Server、BEA WebLogic Enterprise、BEA WebLogic Enterprise Platform、BEA WebLogic Express、BEA WebLogic Integration、BEA WebLogic Personalization Server、BEA WebLogic Platform、BEA WebLogic Portal、EA WebLogic Server、BEA WebLogic Workshop および How Business Becomes E-Business は、BEA Systems, Inc の商標です。

その他の商標はすべて、関係各社がその権利を有します。

WebLogic HTTP サーブレット 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6月28日	BEA WebLogic Server バージョン 7.0

目次

このマニュアルの内容

対象読者.....	vii
e-docs Web サイト.....	viii
このマニュアルの印刷方法.....	viii
関連情報.....	viii
サポート情報.....	ix
表記規則.....	x

1. HTTP サーブレットの概要

サーブレットとは.....	1-1
サーブレットの特長.....	1-2
サーブレットのデプロイメントの概要.....	1-3
サーブレットと J2EE.....	1-3
HTTP サーブレット API リファレンス.....	1-4

2. プログラミングの概要

単純な HTTP サーブレットの記述.....	2-1
高度な機能.....	2-3
HelloWorldServlet サンプルの全文.....	2-5

3. プログラミング タスク

サーブレットの初期化.....	3-1
WebLogic Server 起動時のサーブレットの初期化.....	3-2
init() メソッドのオーバーライド.....	3-3
HTTP 応答の提供.....	3-4
クライアント入力の取得.....	3-6
HTTP リクエストを使用するメソッド.....	3-8
例:クエリ パラメータによる入力の取得.....	3-9
サーブレットでのクライアント入力のセキュリティ.....	3-10
WebLogic Server ユーティリティ メソッドの使い方.....	3-11
サーブレットからのセッション トラッキング.....	3-12

セッショントラッキングの履歴	3-13
HttpSession オブジェクトを用いたセッションのトラッキング	3-14
セッションの有効期間.....	3-15
セッショントラッキングの仕組み.....	3-15
セッションの開始の検出.....	3-16
セッション名/値の属性の設定と取得	3-17
セッションのログアウトと終了	3-17
単一の Web アプリケーションに対する <code>session.invalidate()</code> の使用、 3-18	
複数のアプリケーションに対するシングルサインオンの実装.....	3-18
シングルサインオンからの Web アプリケーションの除外	3-19
セッショントラッキングのコンフィグレーション	3-19
クッキーに代わる URL 書き換えの使用.....	3-19
URL 書き換えと Wireless Access Protocol (WAP)	3-20
セッションの永続化.....	3-21
セッション使用時に避けるべき状況.....	3-22
シリアライズ可能な属性値の使い方.....	3-22
セッションの永続性のコンフィグレーション	3-23
サーブレットでのクッキーの使い方	3-23
HTTP サーブレットでのクッキーの設定.....	3-23
HTTP サーブレットでのクッキーの取得.....	3-24
HTTP と HTTPS の両方で送信されるクッキーの使い方	3-25
セキュアなクッキー.....	3-25
アプリケーションのセキュリティとクッキー.....	3-26
応答のキャッシュ.....	3-27
初期化パラメータ	3-32
HTTP サーブレットからの WebLogic サービスの使い方	3-33
データベースへのアクセス	3-34
JDBC 接続プールを用いたデータベースへの接続.....	3-34
サーブレットでの接続プールの使い方.....	3-35
DataSource オブジェクトを用いたデータベースへの接続.....	3-36
サーブレットでの DataSource の使用.....	3-36
JDBC ドライバを用いたデータベースへの直接接続.....	3-37
HTTP サーブレットにおけるスレッドの問題	3-37
SingleThreadModel	3-38

共有リソース.....	3-38
別のリソースへのリクエストのディスパッチ.....	3-39
リクエストの転送.....	3-40
リクエストのインクルード.....	3-41

4. 管理とコンフィグレーション

WebLogic HTTP サーブレットの管理の概要.....	4-1
サーブレットをコンフィグレーション、デプロイするためのデプロイメント記述子の使い方.....	4-2
web.xml (Web アプリケーション デプロイメント記述子).....	4-2
weblogic.xml (WebLogic 固有のデプロイメント記述子).....	4-3
WebLogic Server Administration Console.....	4-4
Web アプリケーションのディレクトリ構造.....	4-5
Web アプリケーションでのサーブレットの参照.....	4-5
URL パターン マッチング.....	4-6
サーブレットのセキュリティ.....	4-7
認証.....	4-8
認可 (セキュリティ制約).....	4-8
サーブレット開発のヒント.....	4-9
サーブレットのクラスタ化.....	4-10



このマニュアルの内容

このマニュアルでは、**WebLogic HTTP** サーブレットをプログラミングおよびデプロイする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「HTTP サーブレットの概要」では、**Hypertext Transfer Protocol (HTTP)** サーブレットのプログラミングについて概説し、**HTTP** サーブレットを **WebLogic Server** で使用する方法について説明します。
- 第 2 章「プログラミングの概要」では、基本的な **HTTP** サーブレットのプログラミングについて説明します。
- 第 3 章「プログラミング タスク」では、**WebLogic Server** 環境での **HTTP** サーブレットの記述方法について説明します。
- 第 4 章「管理とコンフィグレーション」では、**WebLogic Server** 環境での **HTTP** サーブレットの記述方法について説明します。

対象読者

このマニュアルは、**HTTP** サーブレットと **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を一度に 1 章ずつ印刷できます。

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>) で無料で入手できます。

関連情報

- javax.servlet パッケージ
(<http://java.sun.com/products/servlet/2.3/javadoc/javax/servlet/package-summary.html>)
- javax.servlet.http パッケージ
(<http://java.sun.com/products/servlet/2.3/javadoc/javax/servlet/http/package-summary.html>)
- サーブレット 2.3 仕様
(<http://java.sun.com/products/servlet/download.html#specs>)

-
- 『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>)
 - 「Web アプリケーションのデプロイメント記述子の記述」
(<http://edocs.beasys.co.jp/e-docs/wls/docs70/webapp/webappdeployment.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@beasys.com までお送りください。寄せられた意見については、WebLogic Server の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の名前とバージョン、およびドキュメントのタイトルと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]	複数のキーを同時に押すことを示す。
<i>斜体</i>	強調または書籍のタイトルを示す。
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、データ構造体とそのメンバー、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： <pre>import java.util.Enumeration; chmod u+w * config/examples/applications .java config.xml float</pre>
<i>斜体の等幅テキスト</i>	コード内の変数を示す。 例： <pre>String <i>CustomerName</i>;</pre>
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： <pre>LPT1 BEA_HOME OR</pre>
{ }	構文の中で複数の選択肢を示す。

表記法	適用
[]	<p>構文の中で任意指定の項目を示す。</p> <p>例：</p> <pre>java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]</pre>
	<p>構文の中で相互に排他的な選択肢を区切る。</p> <p>例：</p> <pre>java weblogic.deploy [list deploy undeploy update] password {application} {source}</pre>
...	<p>コマンドラインで以下のいずれかを示す。</p> <ul style="list-style-type: none"> ■ 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。
.	<p>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。</p> <p>.</p> <p>.</p> <p>.</p>



1 HTTP サーブレットの概要

以下の節では、Hypertext Transfer Protocol (HTTP) サーブレットのプログラミングについて概説し、HTTP サーブレットを WebLogic Server で使用する方法について説明します。

- サーブレットとは
- サーブレットの特長
- サーブレットのデプロイメントの概要
- サーブレットと J2EE
- HTTP サーブレット API リファレンス

サーブレットとは

サーブレットとは、Java に対応したサーバで実行される Java クラスです。HTTP サーブレットは、HTTP リクエストを処理し、通常は HTML ページの形式で HTTP 応答を送信する特殊なサーブレットです。WebLogic HTTP サーブレットの最も一般的な使い方は、クライアントサイドのプレゼンテーションに標準的な Web ブラウザを使い、WebLogic Server ではサーバサイドのプロセスとしてビジネスロジックを処理する、対話型アプリケーションを作成することです。WebLogic HTTP サーブレットは、データベース、エンタープライズ JavaBeans、メッセージング API、HTTP セッションなどの WebLogic Server の機能にアクセスできます。

WebLogic Server は、Sun Microsystems のサーブレット 2.3 仕様で定義されている HTTP サーブレットを完全にサポートしています。HTTP サーブレット形式は、Java 2 Enterprise Edition (J2EE) 規格の不可欠な部分です。

サーブレットの特長

- HTML フォームを使用してエンドユーザの入力を取得し、その入力に応答する HTML ページを表示する、動的な Web ページを作成できます。たとえば、オンラインショッピング カート、金融サービス、パーソナライズされたコンテンツなどに使用できます。
- オンライン会議などのコラボレーション システムを作成できます。
- **WebLogic Server** で実行されるサーブレットは、さまざまな API やサービスにアクセスできます。次に例を示します。
 - セッショントラッキング – Web サイトで複数の Web ページにわたるユーザの動きを追跡できます。この機能は、ショッピング カートを使用する e-コマース サイトなどの Web サイトをサポートします。**WebLogic Server** はデータベースへのセッション永続性をサポートしており、サーバのダウン タイム中のフェイルオーバー、およびクラスタ化されたサーバ間のセッションの共有を提供します。詳細については、3-12 ページの「サーブレットからのセッショントラッキング」を参照してください。
 - JDBC ドライバ (BEA のドライバを含む) – JDBC ドライバにより、基本的なデータベース アクセスが提供されます。**WebLogic Server** の多層 JDBC 実装により、接続プール、サーバサイドのデータ キャッシュ、およびトランザクションを利用できます。詳細については、3-34 ページの「データベースへのアクセス」を参照してください。
 - セキュリティ – 認証用の ALC やセキュアな通信を実現するセキュア ソケット レイヤ (SSL) の使用など、さまざまなタイプのセキュリティをサーブレットに適用できます。
 - エンタープライズ **JavaBeans** – サーブレットでエンタープライズ **JavaBean** (EJB) を使用して、セッション、データベースのデータ、その他の機能をカプセル化できます。
 - **Java Messaging Service (JMS)** – **JMS** を使用して、他のサーブレットや **Java** プログラムとメッセージを交換できます。
 - **Java JDK API** – サーブレットでは、標準的な **Java JDK API** を使用できます。
 - リクエストの転送 – 他のサーブレットやリソースへリクエストを転送できます。

- J2EE 準拠のサーブレット エンジン用に作成されたサーブレットであれば、WebLogic Server に簡単にデプロイできます。
- サーブレットと JavaServer Pages (JSP) を組み合わせてアプリケーションを作成できます。

サーブレットのデプロイメントの概要

- HTTP サーブレットのプログラマは、JavaSoft の標準 API である `javax.servlet.http` を利用して対話型のアプリケーションを作成します。
- HTTP サーブレットは HTTP ヘッダを読み取り、HTML コードを書き出してブラウザ クライアントへ応答を送り出すことができます。
- サーブレットは、Web アプリケーションの一部として WebLogic Server にデプロイされます。Web アプリケーションとは、サーブレット クラス、JavaServer Pages (JSP)、静的な HTML ページ、画像、セキュリティなどのアプリケーション コンポーネントをグループ化したものです。詳細については、4-1 ページの「管理とコンフィグレーション」を参照してください。

サーブレットと J2EE

Java 2 Platform, Enterprise Edition の一部であるサーブレット 2.3 仕様は、サーブレット API の実装と、エンタープライズ アプリケーションでのサーブレットのデプロイ方法を定義しています。WebLogic Server など J2EE 準拠のサーバでサーブレットをデプロイするには、エンタープライズ アプリケーションを構成するサーブレットなどのリソースを Web アプリケーションという 1 つの単位にパッケージ化します。Web アプリケーションでは、リソースを格納する特定のディレクトリ構造と、これらのリソースが対話する方法や、クライアントによるアプリケーションへのアクセス方法を定義する、デプロイメント記述子を利用します。また、Web アプリケーションは `.war` ファイルと呼ばれるアーカイブ ファイルとしてデプロイすることもできます。

Web アプリケーションの作成の詳細については、『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。サーブレットの管理およびデプロイメントに関する問題の概要は、4-1 ページの「管理とコンフィグレーション」を参照してください。

HTTP サーブレット API リファレンス

WebLogic Server では、Java サーブレット 2.3 API の `javax.servlet.http` パッケージがサポートされています。このパッケージについては、Sun Microsystems から、さらに以下のドキュメントが提供されています。

- API ドキュメント
 - `javax.servlet` パッケージ
 - `javax.servlet.http` パッケージ
- サーブレット 2.3 仕様

2 プログラミングの概要

以下の節では、基本的な HTTP サーブレットのプログラミングについて説明します。

- 単純な HTTP サーブレットの記述
- 高度な機能
- HelloWorldServlet サンプルの全文

単純な HTTP サーブレットの記述

この節では、Hello World というメッセージを出力する単純な HTTP サーブレットを記述する手順について説明します。これらの手順を示すサンプルコード (HelloWorldServlet) の全文がこの節の最後にあります。JDBC、RMI、JMS など各種の J2EE および WebLogic Server サービスをサーブレットで使用する方法的詳細については、このマニュアルで後述します。

1. 以下の適切なパッケージおよびクラスをインポートします。

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

2. javax.servlet.http.HttpServlet を拡張します。次に例を示します。

```
public class HelloWorldServlet extends HttpServlet{
```

3. service() メソッドを実装します。

サーブレットの主要な機能は、Web ブラウザからの HTTP リクエストを受け取って、HTTP 応答を返すことです。この処理は、サーブレットの service() メソッドによって行われます。サービス メソッドには、出力を生成する応答オブジェクトと、クライアントからのデータを受け取るリクエスト オブジェクトがあります。

これ以外にも、`doPost()` メソッドや `doGet()` メソッドを実装したサーブレットのサンプルを見たことがあるかもしれません。これらのメソッドは、**POST** または **GET** リクエストにのみ応答するものです。`service()` メソッドを実装しておけば、1つのメソッドですべてのリクエスト タイプを処理できます（ただし、`service()` メソッドを実装した場合、このメソッドの最初で `super.service()` を呼び出さない限り、`doPost()` メソッドや `doGet()` メソッドを実装することはできません）。**HTTP** サーブレットの仕様ではこれ以外に、他のリクエスト タイプの処理に使用されるメソッドも解説していますが、全部をまとめて、サービス メソッドと総称しています。

サービス メソッドはすべて、同じパラメータ引数を取ります。

`HttpServletRequest` は、リクエストの情報を提供し、

`HttpServletResponse` は **HTTP** クライアントに応答する際にサーブレットによって使用されます。サービス メソッドは次のようになります。

```
public void service(HttpServletRequest req,
                    HttpServletResponse res) throws IOException
{
```

4. 次のように、コンテンツ タイプを設定します。

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

5. 次のように、出力に使用する `java.io.PrintWriter` オブジェクトへの参照を取得します。

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

6. 次の例に示すように、`PrintWriter` オブジェクトに対し `println()` メソッドを使用して、**HTML** を生成します。

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

7. サーブレットを次のようにコンパイルします。

- a. 開発環境シェルの設定を、クラスパスとパスを正しく指定して行います。
- b. サーブレットの **Java** ソース コードが格納されているディレクトリから、サーブレットが格納されている **Web** アプリケーションの `WEB-INF/classes` ディレクトリに、サーブレットをコンパイルします。次に例を示します。

```
javac -d /myWebApplication/WEB-INF/classes myServlet.java
```

8. **WebLogic Server** にホストが配置される **Web** アプリケーションの一部として、サーブレットをデプロイします。サーブレットのデプロイメントの概要については、4-1 ページの「管理とコンフィグレーション」を参照してください。
9. ブラウザからサーブレットを呼び出します。
サーブレットの呼び出しに使用する **URL** は、(a) そのサーブレットが含まれる **Web** アプリケーション名、(b) **Web** アプリケーションのデプロイメント記述子にマップされるサーブレット名によって決まります。リクエストパラメータも、サーブレットを呼び出す **URL** に含まれることがあります。
一般的には、サーブレットの **URL** は以下のパターンに従います。

```
http://host:port/webApplicationName/mappedServletName?parameter
```

URL の各要素は次のように定義します。

- **host** は、**WebLogic Server** が稼動しているマシンの名前。
- **port** は、上記マシンが **HTTP** リクエストをリスンしているポート。
- **webApplicationName** は、サーブレットが含まれる **Web** アプリケーションの名前。
- **parameters** は、サーブレットで使用できるブラウザから送信された情報が含まれる、1 つまたは複数の名前と値の組み合わせ。

たとえば、**examplesWebApp** にデプロイされ、使用中のマシンで稼動している **WebLogic Server** から提供される **HelloWorldServlet** (このマニュアルで使われているサンプル) を **Web** ブラウザで呼び出す場合、次のような **URL** を入力します。

```
http://localhost:7001/examplesWebApp/HelloWorldServlet
```

URL の **host:port** 部分は、**WebLogic Server** にマップされている **DNS** 名に置き換えることもできます。

高度な機能

上記の手順で、基本的なサーブレットが作成できます。サーブレットでは、さらに高度な機能を使用することもできます。

- **HTML 形式のデータ処理** – HTTP サーブレットでは、ブラウザ クライアントからの **HTML** フォームによるデータを受け取り、処理できます。
 - 3-6 ページの「クライアント入力の取得」
- **アプリケーションの設計** – HTTP サーブレットでは、さまざまな方法でアプリケーションを設計できます。サーブレットの記述に関する詳細は、以下の節を参照してください。
 - 3-4 ページの「HTTP 応答の提供」
 - 3-37 ページの「HTTP サーブレットにおけるスレッドの問題」
 - 3-39 ページの「別のリソースへのリクエストのディスパッチ」
- **サーブレットの初期化** – サーブレットの初期化時に、データを初期化する、初期化引数を受け取るなどのアクションを実行する必要がある場合は、`init()` メソッドをオーバーライドできます。
 - 3-1 ページの「サーブレットの初期化」
- **サーブレットにおけるセッションおよび永続性の使用** – セッションと永続性により、**HTTP** セッション中、および **HTTP** セッション間でユーザを追跡できます。セッション管理には、クッキーの使用も含まれます。詳細については、以下の節を参照してください。
 - 3-12 ページの「サーブレットからのセッション トラッキング」
 - 3-23 ページの「サーブレットでのクッキーの使い方」
 - 3-23 ページの「セッションの永続性のコンフィグレーション」
- **サーブレットにおける WebLogic サービスの使用** – **WebLogic Server** が提供するさまざまなサービスや API を **Web** アプリケーションで使用できます。サービスには、**Java Database Connectivity (JDBC)** ドライバ、**JDBC** データベース接続プール、**Java Messaging Service (JMS)**、エンタープライズ **JavaBeans (EJB)**、**Remote Method Invocation (RMI)** などがあります。詳細については、以下の節を参照してください。
 - 3-33 ページの「HTTP サーブレットからの **WebLogic** サービスの使い方」
 - 4-7 ページの「サーブレットのセキュリティ」
 - 3-34 ページの「データベースへのアクセス」

HelloWorldServlet サンプルの全文

この節では、前述の手順で使用した Java ソース コード サンプルの全文を示します。このサンプルは HTTP リクエストに応答する簡単なサーブレットです。このマニュアルの後半では、このサンプルを拡張することにより、HTTP パラメータ、クッキー、およびセッショントラッキングの使い方を説明します。

コード リスト 2-1 HelloWorldServlet.java

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

public class HelloWorldServlet extends HttpServlet {
    public void service(HttpServletRequest req,
                        HttpServletResponse res)
        throws IOException
    {
        // 先にコンテンツ タイプを設定する
        res.setContentType("text/html");
        // ここで HTML を挿入する PrintWriter を取得する
        PrintWriter out = res.getWriter();

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

WebLogic Server 配布キットの `samples/examples/servlets` ディレクトリに、このソースコードと、このマニュアルで使われている全サンプルをコンパイルして実行するための手順説明があります。

3 プログラミング タスク

以下の節では、WebLogic Server 環境で HTTP サーブレットを記述する方法について説明します。

- サーブレットの初期化
- HTTP 応答の提供
- クライアント入力の取得
- サーブレットからのセッション トラッキング
- サーブレットでのクッキーの使い方
- 応答のキャッシュ
- HTTP サーブレットからの WebLogic サービスの使い方
- データベースへのアクセス
- HTTP サーブレットにおけるスレッドの問題
- 別のリソースへのリクエストのディスパッチ

サーブレットの初期化

通常、WebLogic Server によってサーブレットが初期化されるのは、そのサーブレットに対して最初のリクエストが出されたときです。その後、そのサーブレットが変更されると、既存のバージョンのサーブレットに対して `destroy()` メソッドが呼び出されます。変更後のサーブレットにリクエストが送信されると、変更後のサーブレットの `init()` メソッドが実行されます。詳細については、4-9 ページの「サーブレット開発のヒント」を参照してください。

サーブレットが初期化される時、WebLogic Server はサーブレットの `init()` メソッドを実行します。サーブレットは一度初期化されると、WebLogic Server を再起動するまで、またはサーブレットが変更された場合はサーブレットが呼び

出されるまで、再び初期化されることはありません。init() メソッドをオーバーライドすると、データベース接続の確立など、サーブレットの初期化時に特定のタスクを実行させることができます(3-3 ページの「init() メソッドのオーバーライド」を参照)。

WebLogic Server 起動時のサーブレットの初期化

サーブレットに最初のリクエストが送信された時に **WebLogic Server** がサーブレットを初期化するのではなく、サーバの起動時に初期化するように、最初に **WebLogic Server** をコンフィグレーションできます。そのためには、**Web** アプリケーションデプロイメント記述子の <load-on-startup> 要素内にサーブレットクラスを指定します。詳細については、「servlet 要素」を参照してください。サーブレットをロードする起動クラスの使用は、サポートされなくなりました。

初期化中に **HTTP** サーブレットにパラメータを渡すには、サーブレットを含む **Web** アプリケーションでパラメータを定義します。このパラメータを使用すると、サーブレットが初期化されるたびに書き換えることなく値を渡せます。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。

たとえば、**Web** アプリケーションデプロイメント記述子の以下のようなエントリの場合、次の 2 つの初期化パラメータが定義されます。1 つは `welcome` の値を持つ `greeting`、もう 1 つは `WebLogic Developer` の値を持つ `person` です。

```
<servlet>
  ...
  <init-param>
    <param-name>greeting</param-name>
    <param-value>Welcome</param-value>
    <description>The salutation</description>
  </init-param>
  <init-param>
    <param-name>person</param-name>
    <param-value>WebLogic Developer</param-value>
    <description>name</description>
  </init-param>
</servlet>
```

初期化パラメータを取得するには、親 `javax.servlet.GenericServlet` クラスから `getInitParameter(String name)` メソッドを呼び出します。パラメータ名を渡されると、このメソッドはパラメータの値を文字列で返します。

init() メソッドのオーバーライド

init() メソッドをオーバーライドすることで、初期化時にサーブレットにタスクを実行させることができます。次のコードでは、Web アプリケーションデプロイメント記述子の中で挨拶文と名前を定義する <init-param> タグを読み込んでいます。

```
String defaultGreeting;
String defaultName;

public void init(ServletConfig config)
    throws ServletException {
    if ((defaultGreeting = getInitParameter("greeting")) == null)
        defaultGreeting = "Hello";

    if ((defaultName = getInitParameter("person")) == null)
        defaultName = "World";
}
```

各パラメータの値は、クラスインスタンス変数 defaultGreeting および defaultName に格納されます。最初にパラメータが null 値を持つかどうかをテストして、null 値が返されれば、適切なデフォルト値を提供します。

これで、service() メソッドを使用して、応答の中にこれらの変数を含めることができます。次に例を示します。

```
out.print("<body><h1>");
out.println(defaultGreeting + " " + defaultName + "!");
out.println("</h1></body></html>");
```

WebLogic Server 配布キットの samples/examples/servlets ディレクトリに、完全なソースコードと、HelloWorld2.java というサンプルをコンパイル、インストール、および試行するための手順説明があります。その中で、init() メソッドの使い方が解説されています。

サーブレットの init() メソッドは、WebLogic Server がサーブレットをロードするときに必要な初期化処理を行います。デフォルトの init() メソッドは、WebLogic Server が必要とするすべての初期化処理を行うので、特別な初期化要件がある場合を除き、このメソッドをオーバーライドする必要はありません。init() をオーバーライドする場合、デフォルトの初期化アクションが最初に実行されるように、まず super.init() を呼び出します。

HTTP 応答の提供

この節では、HTTP サーブレットでクライアントへの応答を提供する方法について説明します。応答はすべて、サーブレットの `service()` メソッドにパラメータとして渡される `HttpServletResponse` オブジェクトを使用して渡さなければなりません。

1. `HttpServletResponse` をコンフィグレーションします。

`HttpServletResponse` オブジェクトを使用して、HTTP ヘッダ情報に変換される複数のサーブレットプロパティを設定できます。

- 少なくとも、ページのコンテンツを書き込む出力ストリームを取得する前に、`setContentType()` メソッドを使用してコンテンツ タイプを設定します。HTML ページの場合、コンテンツ タイプは `text/html` に設定します。次に例を示します。

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

- (省略可能) `setContentType()` メソッドを使用して文字エンコードを設定することもできます。次に例を示します。

```
res.setContentType("text/html;ISO-8859-4");
```

- `setHeader()` メソッドを使用して、ヘッダ属性を設定します。動的な応答の場合は、`Pragma` 属性を `no-cache` に設定するとよいでしょう。こうするとブラウザは常にページを再ロードし、確実に最新データを表示します。次に例を示します。

```
res.setHeader("Pragma", "no-cache");
```

2. HTML ページを作成します。

サーブレットがクライアントに送り返す応答は、通常の HTTP コンテンツ (基本的には HTML 形式) のように見える必要があります。サーブレットは、`service()` メソッドの応答パラメータを使用して取得した出力ストリームを通じて、HTTP 応答を送り返します。HTTP 応答を送信するには、次の手順に従います。

- a. `HttpServletResponse` オブジェクトおよび次の例のいずれかのメソッドを使用して、出力ストリームを取得します。

- `PrintWriter out = res.getWriter();`

- `ServletOutputStream out = res.getOutputStream();`

同じサーブレット内 (または、サーブレットの中にインクルードされた別のサーブレット内) で、`PrintWriter` と `ServletOutputStream` を両方とも使用できます。どちらの出力も、同じバッファに書き込まれます。

- b. `print()` メソッドを使用して、応答の内容を出力ストリームに書き出します。これらの文では、**HTML** タグを使うことができます。次に例を示します。

```
out.print("<html><head><title>My Servlet</title>");
out.print("</head><body><h1>");
out.print("Welcome");
out.print("</h1></body></html>");
```

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、3-10 ページの「サーブレットでのクライアント入力のセキュリティ」を参照してください。

出力ストリームは `close()` メソッドでクローズしないでください。また、ストリームのコンテンツをフラッシュすることも避けてください。出力ストリームをクローズしたりフラッシュしたりしないことによって、**WebLogic Server** は次の手順で説明する永続的 **HTTP** 接続の利点を活かすことができます。

3. 応答を最適化します。

デフォルトでは、**WebLogic Server** は、可能な限り **HTTP** の永続的接続を使用しようとします。永続的接続は、クライアントとサーバ間の一連の通信のために、同一の **HTTP TCP/IP** 接続を再利用しようとします。各リクエストごとに新しい接続をオープンする必要がないので、アプリケーションの性能を高めることができます。永続的接続は、**HTML** ページにインライン画像が多く含まれる場合に便利です。接続を再利用しないと、画像が要求されるごとに新しい **TCP/IP** 接続が必要になるためです。

WebLogic Server Administration Console を使用すると、**WebLogic Server** が **HTTP** 接続をオープンに保つ時間をコンフィギュレーションできます。

永続的接続を確立するために、**WebLogic Server** は **HTTP** 応答の長さを知る必要があるため、**HTTP** 応答ヘッダに `Content-Length` プロパティを自動的に追加します。コンテンツ長を確認するために、**WebLogic Server** では応答をバッファリングする必要があります。ただし、サーブレットが `ServletOutputStream` を明示的にフラッシュすると、**WebLogic Server** は応

答の長さを判別できないため、永続的接続を使用できません。このため、サーブレットで **HTTP** 応答を明示的にフラッシュすることは避けます。

場合によっては、ページ完成前にクライアントに情報を表示するために応答を早くフラッシュした方がよいこともあります。たとえば、時間のかかるページのコンテンツを計算している途中で、バナー広告を表示する場合などです。逆に、サーブレットエンジンが使用するバッファサイズを増やして、フラッシュする前に、もっと長い応答を入れておきたいという場合もあります。 `javax.servlet.ServletResponse` インタフェースの関連メソッドを用いて、応答バッファのサイズを操作することができます。

WebLogic Server の応答バッファのデフォルト値は **12K** で、バッファサイズは、`CHUNK_SIZE` (`CHUNK_SIZE = 4088 or 4Kb`) に基づいて内部で計算されます。たとえば、ユーザが **5KB** に設定すると、バッファサイズは、それを切り上げた最も近い `CHUNK_SIZE` の倍数 (この場合は **2**) である **8176 (8KB)** に設定されます。

クライアント入力の取得

HTTP サーブレット API は、**Web** ページからユーザ入力を取得するためのインタフェースを提供しています。

Web ブラウザからの **HTTP** リクエストには、クライアント、ブラウザ、クッキー、およびユーザのクエリ パラメータに関する情報といった、**URL** 以外の情報も含めることができます。ブラウザからのユーザ入力を渡すには、クエリパラメータを使用します。**GET** メソッドは **URL** アドレスにパラメータを付加し、**POST** メソッドはそれらを **HTTP** リクエスト本文の中に含めます。

HTTP サーブレットは、これらの細部を扱う必要はありません。リクエストの中の情報は、送った方法に関係なく、`HttpServletRequest` オブジェクトを通じて取得され、`request.getParameters()` メソッドを使用してアクセスできるようになります。

クエリ パラメータをクライアントから送る方法については、以下を参照してください。

- ページのリンクの **URL** に、パラメータを直接エンコードします。この方法では、**GET** メソッドを使用してパラメータを送ります。パラメータは **URL** の後ろに `?` を付けてその後に追加します。パラメータが複数ある場合は `&` で

区切ります。パラメータは、常に名前 = 値の組で指定されます。指定される順序は重要ではありません。たとえば、次のようなリンクを **Web** ページに入れる場合、ここでは `ColorServlet` と呼ばれる **HTTP** サーブレットに、パラメータ `color` の値を `purple` にして送ります。

```
<a href=
  "http://localhost:7001/myWebApp/ColorServlet?color=purple">
  Click Here For Purple!</a>
```

- クエリ パラメータを付けた **URL** をブラウザの場所フィールドに手動で入力します。これは前の例で示したリンクをクリックするのと同じことです。
- **HTML** フォームで、ユーザに入力を要求します。このフォームの送信ボタンがクリックされると、フォーム上の各ユーザ入力フィールドの内容が、クエリ パラメータとして送られます。このフォームがクエリ パラメータを送るために使うメソッド (`POST` または `GET`) を、`METHOD="GET|POST"` 属性を使用して `<FORM>` タグに指定します。

クエリ パラメータは常に名前 = 値の組で送られ、`HttpServletRequest` オブジェクトを通じてアクセスされます。クエリのすべてのパラメータ名の `Enumeration` を取得し、そのパラメータ名を使用して各パラメータの値を取得できます。1 つのパラメータの値は通常 1 つだけですが、値の配列を持つこともできます。パラメータの値は常に `String` として読み取られるので、より適切な型にキャストすることが必要な場合もあります。

`service()` メソッドからの次のサンプルでは、フォームから得たクエリ パラメータ名とその値を調べます。`request` は `HttpServletRequest` オブジェクトであることに注意してください。

```
Enumeration params = request.getParameterNames();
String paramName = null;
String[] paramValues = null;

while (params.hasMoreElements()) {
    paramName = (String) params.nextElement();
    paramValues = request.getParameterValues(paramName);
    System.out.println("\nParameter name is " + paramName);
    for (int i = 0; i < paramValues.length; i++) {
        System.out.println("  value " + i + " is " +
            paramValues[i].toString());
    }
}
```

注意: ユーザによって入力されたデータを出力するたびに、以前に入力された 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、3-10 ページの「サーブレットでのクライアント入力のセキュリティ」を参照してください。

HTTP リクエストを使用するメソッド

この節では、リクエスト オブジェクトからデータを取得できる `javax.servlet.HttpServletRequest` インタフェースのメソッドを定義します。次の制限事項に注意してください。

- この節の `getParameter()` メソッドを使用してリクエスト パラメータを読み込んだ後に、`getInputStream()` メソッドでリクエストを読もうとすることはできません。
- `getInputStream()` でリクエストを読み込んだ後に、`getParameter()` メソッドの 1 つを使ってリクエスト パラメータを読み込もうとすることもできません。

上のいずれかの手順を実行しようとする、`illegalStateException` が送出されます。

`javax.servlet.HttpServletRequest` の次のメソッドを使用して、リクエスト オブジェクトからデータを取得できます。

`HttpServletRequest.getMethod()`
GET や POST などのリクエスト メソッドを決定できます。

`HttpServletRequest.getQueryString()`
クエリ文字列にアクセスできるようにします (クエリ文字列は、要求された URL で ? の後に続く部分です)。

`HttpServletRequest.getParameter()`
パラメータの値を返します。

`HttpServletRequest.getParameterNames()`
パラメータ名の配列を返します。

`HttpServletRequest.getParameterValues()`
パラメータの値の配列を返します。

```
HttpServletRequest.getInputStream()
```

リクエスト本体をバイナリ データとして読み込みます。リクエストパラメータを `getParameter()`、`getParameterNames()`、または `getParameterValues()` で読み込んだ後にこのメソッドを呼び出すと、`illegalStateException` が送出されます。

例：クエリ パラメータによる入力の取得

この例では、より個人的な挨拶文を表示するために、クエリ パラメータとしてユーザ名を受け付けるように `HelloWorld2.java` サブレットのサンプルを修正しています(コードの全文については、**WebLogic Server** 配布キットの `samples\examples\servlets` ディレクトリにある `HelloWorld3.java` サンプルを参照してください)。 `service()` メソッドを次に示します。

コードリスト 3-1 `service()` メソッドによる入力の取得

```
public void service(HttpServletRequest req,
                    HttpServletResponse res)
    throws IOException
{
    String name, paramName[];
    if ((paramName = req.getParameterValues("name"))
        != null) {
        name = paramName[0];
    }
    else {
        name = defaultName;
    }

    // まず、コンテンツ タイプを設定する
    res.setContentType("text/html");
    // PrintWriter を出力ストリームとして取得する
    PrintWriter out = res.getWriter();

    out.print("<html><head><title>" +
              "Hello World!" + </title></head>");
    out.print("<body><h1>");
    out.print(defaultGreeting + " " + name + "!");
    out.print("</h1></body></html>");
}
```

`getParameterValues()` メソッドは、HTTP クエリ パラメータから `name` パラメータの値を取得します。これらの値は、`String` 型の配列で取得します。このパラメータに 1 つの値が返され、`name` 配列の第 1 要素に割り当てられます。クエリ データにこのパラメータがなければ、戻り値は `null` になります。この場合は、`init()` メソッドによって `<init-param>` プロパティから読み込んだデフォルトの名前に `name` を割り当てます。

パラメータが HTTP リクエストに含まれていると仮定してサーブレット コードを記述しないでください。`getParameter()` メソッドは非推奨となったため、配列の添え字にその末尾までタグ付けすることによって、`getParameterValues()` メソッドを略記しようとする場合があります。しかし、このメソッドは指定したパラメータが有効でないと `null` を返すことがあり、`NullPointerException` が発生します。

たとえば、次のコードでは `NullPointerException` が発生します。

```
String myStr = req.getParameterValues("paramName")[0];
```

代わりに、次のコードを使用します。

```
if ((String myStr[] =
    req.getParameterValues("paramName"))!=null) {
    // これで myStr[0] を使うことができる
}
else {
    // paramName がクエリ パラメータの中になかった！
}
```

サーブレットでのクライアント入力のセキュリティ

ユーザによる入力データを取得して返す機能により、**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 3-1 に示した HTML 特殊文字を探します。該当する文字が見つければ、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 3-1 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 メソッドの使い方

WebLogic Server には、ユーザによって入力されたデータに含まれる特殊文字を置き換えるために、`weblogic.servlet.security.Utils.encodeXSS()` メソッドが用意されています。このメソッドを使用するには、ユーザによって入力されたデータを入力として指定します。例えば、コードリスト 3-1 に示したユーザ入力によるデータを保護するには、以下の行を

```
out.print(defaultGreeting + " " + name + "!");
```

次の行と置き換えます。

```
out.print(defaultGreeting + " " +
weblogic.security.servlet.encodeXSS(name) + "!");
```

3 プログラミング タスク

アプリケーション全体を保護するには、ユーザによる入力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。上記のコードリスト 3-1 の例は `encodeXSS()` メソッドを使用する必要がある場所を示し、表 3-2 はこのメソッドの使用を検討すべき場所を示します。

表 3-2 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	「 <code>username</code> 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

サーブレットからのセッション トラッキング

セッション トラッキングを使用すると、複数のサーブレットまたは HTML ページにわたって、本来はステートレスであるユーザの状況を追跡できます。セッションとは、ある期間中に同じクライアントから出される一連の関連性のあるブラウザ リクエストのことです。セッション トラッキングは、ショッピング カート アプリケーションのように、全体として何らかの意味を持つ一連のブラウザ リクエスト (これをページと見なす) を結合します。

以下の節では、HTTP サーブレットからのセッション トラッキングについて、さまざまな観点から説明します。

- セッション トラッキングの履歴
- `HttpSession` オブジェクトを用いたセッションのトラッキング
- セッションの有効期間
- セッション トラッキングの仕組み
- セッションの開始の検出

- セッション名 / 値の属性の設定と取得
- セッションのログアウトと終了
- セッション トラッキングのコンフィグレーション
- クッキーに代わる URL 書き換えの使用
- URL 書き換えと Wireless Access Protocol (WAP)
- セッションの永続化

セッション トラッキングの履歴

セッション トラッキングという概念が発展する前は、ページの非表示フィールドに情報を詰め込むか、長い文字列をリンクで使われる URL に追加してユーザの選択内容を埋め込むことで、ページに状態を組み込んでいました。このよい例が、サーチ エンジン サイトです。サーチ エンジン サイトの多くはいまだに CGI に依存しています。これらのサイトは、URL の後に HTTP で予約されている ? 記号を付け、その後続く URL パラメータの名前 = 値の組を使用して、ユーザの選択を追跡します。このようにすると、URL は非常に長いものになる場合があります、CGI スクリプトはこれを慎重に解析し、管理する必要があります。この手法の問題点は、情報をセッションからセッションへと渡せないことです。URL の制御を失うと、つまりユーザがページから離れると、ユーザ情報は永久に失われます。

その後、Netscape はブラウザのクッキーを発表しました。これを使用してサーバごとにクライアント上のユーザ関連情報を格納することができます。しかし、ブラウザによってはまだクッキーを完全にサポートしていないものもあり、またブラウザのクッキー オプションをオフにしておくことを選ぶユーザもいます。もう 1 つの考慮すべき要因は、ほとんどのブラウザではクッキーで格納できるデータ量を制限しているということです。

CGI の手法と異なり、HTTP サーブレットの仕様では、サーバが単一のセッションを超えるサーバ上のユーザの詳細情報を格納することを可能にし、コードがセッションのトラッキングにより複雑化することを防ぐ方法が定義されています。サーブレットは、HttpSession オブジェクトを使用して、単一のセッション期間中のユーザ入力を追跡し、セッションの詳細を複数サーブレットで共有することができます。セッション データは、WebLogic サービスで使用できるさまざまなメソッドにより保持できます。

HttpSession オブジェクトを用いたセッションの トラッキング

WebLogic が実装してサポートしている Java Servlet API では、各サーブレットは HttpSession オブジェクトを使用して、サーバサイドセッションにアクセスすることができます。HttpServletRequest オブジェクトを使用して、サーブレットの service() メソッド内の HttpSession オブジェクトにアクセスできます。次の例のように変数 request を使用します。

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

引数 true で request.getSession(true) メソッドが呼び出されると、そのクライアントに対して既存の HttpSession オブジェクトがない場合には、HttpSession オブジェクトが生成されます。セッション オブジェクトは、そのセッションの有効期間の間 WebLogic Server に存在し、そのクライアントに関連した情報を蓄積します。サーブレットは、必要に応じて、セッション オブジェクトで情報の追加や削除を行います。セッションは特定のクライアントに関連付けられます。クライアントがサーブレットを訪れるごとに、getSession() メソッドが呼び出されたときと同一の、関連付けられた HttpSession オブジェクトが取得されます。

HttpSession でサポートされるメソッドの詳細については、「HttpServlet API」を参照してください
techdocs/api/javax/servlet/http/HttpSession.html。

次の例では、service() メソッドは、セッション中にユーザがサーブレットを要求する回数を数えます。

```
public void service(HttpServletRequest request,
                    HttpServletResponse, response)
    throws IOException
{
    // セッションとカウンタ パラメータ属性を取得する
    HttpSession session = request.getSession (true);
    Integer ival = (Integer)
        session.getAttribute("simplesession.counter");
    if (ival == null) // カウンタを初期化する
        ival = new Integer (1);
    else // カウンタをインクリメントする
        ival = new Integer (ival.intValue () + 1);
    // セッションに新しい属性値を設定する
    session.setAttribute("simplesession.counter", ival);
    // HTML ページを出力する
    out.print("<HTML><body>");
}
```

```
out.print("<center> You have hit this page ");
out.print(ival + " times!");
out.print("</body></html>");
}
```

セッションの有効期間

セッションは、1つのトランザクションの一連のページにわたるユーザの選択を追跡します。1つのトランザクションは、ある品物を閲覧し、それをショッピングカートに追加した後、支払手続きをするといった複数のタスクで構成されます。セッションは永続的なものではなく、以下のいずれかの時点で、その有効期間は終了します。

- ユーザがサイトを離れ、ブラウザがクッキーを受け入れない場合。
- ユーザがブラウザを終了した場合。
- 動作がないため、セッションがタイムアウトになった場合。
- セッションが完了し、サーブレットによって無効にされた場合。
- ユーザがログアウトし、サーブレットによって無効にされた場合。

より永続的な長い時間データを保存するには、サーブレットは、JDBCまたはEJBを使用してデータベースに詳細を書き込み、存続期間の長いクッキーやユーザ名とパスワードによって、クライアントとこのデータを関連付ける必要があります。このマニュアルでは、セッションが内部的にクッキーや永続性を使用すると説明していますが、ユーザに関するデータの格納のための一般的なメカニズムとしてセッションを使用してはいけません。

セッション トラッキングの仕組み

WebLogic Server は、各クライアントにどのセッションが関連付けられているのかを、どのようにして認識するのでしょうか。HttpSession がサーブレットで生成されると、ユニークな ID と関連付けられます。ブラウザは、このセッション ID をそのリクエストに付与し、サーバが再びセッションデータを見つけられるようにしなければなりません。サーバは、クライアントにクッキーを設定することによって、この ID を保存しようとします。一度クッキーが設定されると、

ブラウザがサーバにリクエストを送るたびに、リクエストにはその ID を内包したクッキーが含まれます。サーバは自動的にクッキーを解析し、サーブレットが `getSession()` メソッドを呼び出すと、セッション データを提供します。

クライアントがクッキーを受け入れない場合、代替の方法としては、クライアントへ送り返されるページのなかで、URL リンクにその ID をエンコードするかありません。このため、サーブレットの応答に URL を入れる場合は、必ず `encodeURL()` メソッドを使用します。WebLogic Server はブラウザがクッキーを受け入れるかどうかを検出して、不必要な URL のエンコードは行いません。自動的に、エンコードされた URL からセッション ID を解析し、`getSession()` メソッドを呼び出すと、正しいセッション データを取得します。`encodeURL()` メソッドを使用すると、セッション トラッキングにどの手順を使用しても、サーブレットのコードが破壊されることはありません。詳細については、「3-19 ページの「クッキーに代わる URL 書き換えの使用」」を参照してください。

セッションの開始の検出

`getSession(true)` メソッドを使用してセッションを取得した後、`HttpSession.isNew()` メソッドを呼び出すことにより、そのセッションが生成されたばかりかどうかわかります。このメソッドが `true` を返した場合、クライアントはまだ有効なセッションを持っておらず、またこの時点では新規のセッションを認識しません。クライアントが新規のセッションを認識するのは、サーバから応答がポストされて戻ってからです。

ビジネス ロジックに合った方法で、新規または既存のセッションに適應するようにアプリケーションを設計してください。たとえば、セッションがまだ開始していないと判断すれば、次のサンプル コードのように、アプリケーションでクライアントの URL をログイン / パスワードのページにリダイレクトしてもよいでしょう。

```
HttpSession session = request.getSession(true);
if (session.isNew()) {
    response.sendRedirect(welcomeURL);
}
```

ログインのページでは、システムにログインするか、新規アカウントを作成するかを選択できるようにします。また、Web アプリケーションでログイン ページを指定することもできます。詳細については、「login-config」を参照してください。

セッション名 / 値の属性の設定と取得

名前 = 値の組み合わせを使用して、`HttpSession` オブジェクトにデータを格納できます。セッションに格納されたデータは、そのセッションを通じて利用することができます。セッションにデータを格納するには、次のメソッドを `HttpSession` インタフェースから使用します。

```
getAttribute()
getAttributeNames()
setAttribute()
removeAttribute()
```

次のコードでは、既存の名前 = 値の組み合わせをすべて取得する方法を示します。

```
Enumeration sessionNames = session.getAttributeNames();
String sessionName = null;
Object sessionValue = null;

while (sessionNames.hasMoreElements()) {
    sessionName = (String)sessionNames.nextElement();
    sessionValue = session.getAttribute(sessionName);
    System.out.println("Session name is " + sessionName +
        ", value is " + sessionValue);
}
```

名前の付いた属性を追加または上書きするには、`setAttribute()` メソッドを使用します。名前の付いた属性を完全に削除するには、`removeAttribute()` メソッドを使用します。

注意： Java の `Object` の子孫をセッション属性として追加し、それに名前を関連付けることができます。ただし、セッション永続性を利用している場合、属性 `value` のオブジェクトは `java.io.Serializable` を実装しなければなりません。

セッションのログアウトと終了

アプリケーションがデリケートな情報を扱う場合、セッションからログアウトする機能の提供を検討することがあります。これは、ショッピング カートやインターネットの電子メール アカウントを使用する際には一般的な機能です。同一のブラウザがサービスに戻るとき、ユーザはシステムにログインし直さなければなりません。

単一の Web アプリケーションに対する `session.invalidate()` の使用

ユーザ認証の情報は、ユーザのセッションデータ、およびサーバのコンテキストまたは Web アプリケーションにより割り当てられた仮想ホストのコンテキストの両方に格納されます。ユーザのログアウトに多く使われる

`session.invalidate()` メソッドを使用すると、ユーザの現在のセッションのみが無効になり、ユーザの認証情報は有効なまま、サーバまたは仮想ホストのコンテキストに格納されます。サーバまたは仮想ホストが 1 つの Web アプリケーションのみのホストである場合、実際には `session.invalidate()` メソッドでユーザをログアウトします。

`session.invalidate()` を呼び出した後、無効にされたセッションを参照しないでください。参照した場合、`IllegalStateException` が送出されます。ユーザが次に同じブラウザからサーブレットにアクセスしたときには、セッションデータは失われています。`getSession(true)` メソッドを呼び出すと、新しいセッションが作成されます。この時点で、ユーザに再度ログインページを送信できます。

複数のアプリケーションに対するシングル サインオンの実装

サーバまたは仮想ホストが複数の Web アプリケーションに割り当てられている場合、すべての Web アプリケーションからユーザをログアウトするには、別の方法が必要になります。サーブレット仕様では、すべての Web アプリケーションからユーザをログアウトするための API が用意されていないため、以下のメソッドを使用します。

```
weblogic.servlet.security.ServletAuthentication.logout()
    認証データをユーザのセッションデータから削除します。これにより、
    ユーザはログアウトされますが、セッションは引き続き有効です。

weblogic.servlet.security.ServletAuthentication.invalidateAll()
    すべてのセッションを無効にして、現在のユーザの認証データを削除し
    ます。クッキーも無効になります。

weblogic.servlet.security.ServletAuthentication.killCookie(
)
    応答がブラウザに送信された直後に有効期限が切れるようにクッキーを
    設定して、現在のクッキーを無効にします。このメソッドでは、応答が
```

ユーザのブラウザに正常に届く必要があります。セッションはタイムアウトするまで維持されます。

シングル サインオンからの Web アプリケーションの除外

シングル サインオンへの参加から Web アプリケーションを除外するには、除外する Web アプリケーションに異なるクッキー名を定義します。詳細については、「セッション クッキーのコンフィグレーション」を参照してください。

セッション トラッキングのコンフィグレーション

WebLogic Server では、セッション トラッキングの処理方法を決定するさまざまな属性をコンフィグレーションできます。これらのセッション トラッキング属性のコンフィグレーションの詳細については、「session-descriptor」を参照してください。

クッキーに代わる URL 書き換えの使用

状況によっては、ブラウザがクッキーを受け入れないこともあります。この場合、クッキーによるセッション トラッキングができません。代わりに URL 書き換えを使用すると、ブラウザがクッキーを受け入れないことを WebLogic Server が検出したときに、こうした状況を自動的に置き換えることができます。URL 書き換えでは、セッション ID を Web ページのハイパーリンクにエンコードし、サーブレットはそれらをブラウザに送り返します。ユーザが以後これらのリンクをクリックすると、WebLogic Server は URL からその ID を抽出し、適切な HttpSession を見つけ出します。その後、getSession() メソッドを使用して、セッション データにアクセスします。

WebLogic Server で URL 書き換えを有効にするには、WebLogic 固有のデプロイメント記述子の session-descriptor 要素で UrlRewritingEnabled 属性を true に設定します。

URL 書き換えをサポートするために、コードで URL を適切に処理するには、以下のガイドラインに従います。

- 次に示すように、URL を出力ストリームに直接書き出すことは避けます。

```
out.println("<a href=\" /myshop/catalog.jsp\">catalog</a>");
```

代わりに、`HttpServletResponse.encodeURL()` メソッドを使用します。次に例を示します。

```
out.println("<a href=\""+  
+ response.encodeURL("myshop/catalog.jsp")  
+ "\">catalog</a>");
```

- `encodeURL()` メソッドを呼び出すと、URL を書き換える必要があるかどうか調べられます。必要である場合、URL にセッション ID を組み込むことによって書き換えを行います。
- **WebLogic Server** への応答として返される URL に加えて、リダイレクトを送信する URL をエンコードします。次に例を示します。

```
if (session.isNew())  
    response.sendRedirect(response.encodeRedirectUrl(welcomeURL));
```

WebLogic Server はセッションが新しいときには、ブラウザがクッキーを受け入れる場合でも URL 書き換えを使用します。これは、セッションの最初ではサーバはブラウザがクッキーを受け入れるかどうかを判断できないからです。

サーブレットは、`HttpServletRequest.isRequestedSessionIdFromCookie()` メソッドから返されるブール値をチェックすることによって、所定のセッションがクッキーから返されたかを確認できます。**WebLogic Server** アプリケーションは適切に応答するか、**WebLogic Server** による URL 書き換えに依存します。

注意： **CISCO Local Director** ロード バランサでは、URL 書き換えの区切り記号として疑問符 (?) が想定されています。WLS の URL 書き換えメカニズムは区切り記号としてセミコロン (;) を使用するのので、WLS の URL 書き換えとこのロード バランサの間には互換性がありません。

URL 書き換えと Wireless Access Protocol (WAP)

WAP アプリケーションを作成する場合、WAP プロトコルはクッキーをサポートしていないため、URL を書き換える必要があります。また、一部の WAP デバイスでは、URL の長さが 128 文字 (パラメータも含む) に制限されます。これにより、URL 書き換えによって転送できるデータサイズが制限されます。パラメータ用の領域を増やすために、**WebLogic Server** によってランダムに生成され

るセッション ID のサイズを制限できます。そのためには、WebLogic 固有のデプロイメント記述子である `weblogic.xml` の `<session-descriptor>` 要素の `IDLength` 属性を使用してバイト数を指定します。

最小値は 8 バイト、デフォルト値は 52 バイトです。最大値は `Integer.MAX_VALUE` です。

セッションの永続化

WebLogic Server は、永続ストレージにセッション データを記録するように設定できます。セッション永続性を使用する場合、以下の特徴があります。

- フェイルオーバーのよさ。サーバに障害が発生してもセッションが保存されるためです。
- ロード バランスのよさ。どのサーバでも、セッションがいくつあってもそのリクエストを処理し、キャッシュを使用して、パフォーマンスを最適化できるためです。詳細については、にある「セッション永続性のコンフィグレーション」の `cacheEntries` プロパティを参照してください。
- セッションを、クラスタ化された WebLogic Server 間で共有できます。WebLogic クラスタではセッション永続性は必要条件ではなくなりました。その代わりに、ステートのインメモリ レプリケーションを使用します。詳細については、『WebLogic Server クラスタ ユーザーズ ガイド』を参照してください。
- 顧客が最高品質のサーブレット セッション永続性を求めている場合は、JDBC ベースの永続性が最適です。セッション永続性はある程度犠牲にしても、パフォーマンスを大幅に高めたい顧客の場合には、インメモリ レプリケーションが適切な選択です。JDBC ベースの永続性は、インメモリ レプリケーションよりも著しく低速です。サーブレット セッションに対して、インメモリ レプリケーションが JDBC ベースの永続性よりも 8 倍高いパフォーマンスを提供したケースもあります。
- セッションにはどの種類の Java オブジェクトを入れることもできますが、セッションに格納するオブジェクトには `java.io.Serializable` が実装されている必要があります。詳細については、「セッション永続性のコンフィグレーション」を参照してください。

セッション使用時に避けるべき状況

セッション永続性を、セッション間の長期データを格納する目的には使わないでください。つまり、後日クライアントがサイトに戻ったときに、アクティブなままのセッションがあっても、それに依存してはいけません。むしろアプリケーションでは、長期にわたる情報や重要な情報は、データベースの中に記録すべきです。

セッションはクッキーのコンビニエンス ラッパーではありません。セッションには長期間であろうと一定期間であろうと、クライアントデータを格納しようとしてはいけません。それよりも、アプリケーションのほうでクッキーをブラウザ上に作成し設定すべきです。例として、クッキーが長期間存続できる自動ログイン機能、クッキーが短期間で失効する自動ログアウト機能などがあります。この場合、**HTTP**セッションを使用しないでください。代わりに、アプリケーション固有のロジックを記述します。

セッション ID の形式は内部的に指定され、また **WebLogic Server** のバージョンごとに変更になる可能性があります。そのため、特定のセッション ID 形式を必要とするようなアプリケーションは作成しないことをお勧めします。

シリアライズ可能な属性値の使い方

永続セッションを使用する場合、セッションに追加するすべての属性 **value** オブジェクトは **java.io.Serializable** を実装する必要があります。シリアライズ可能なクラスの作成の詳細については、シリアライズ可能なオブジェクトに関するオンライン **Java** チュートリアルを参照してください ([providing.html](#))。独自のシリアライズ可能なクラスを永続セッションに加えるときは、クラスの各インスタンス変数もシリアライズ可能になっているか注意してください。シリアライズ可能でない場合は、それを **transient** として宣言できます。**WebLogic Server** は、その変数を永続ストレージには保存しません。

transient とすべきインスタンス変数の一般的な例としては、**HttpSession** オブジェクトが挙げられます (3-21 ページの「セッションの永続化」の、セッションにおけるシリアライズされたオブジェクトの使用に関する注意を参照してください)。

HttpServletRequest、**ServletContext**、および **HttpSession** 属性は、**WebLogic Server** インスタンスが、**Web** アプリケーションのクラスローダで変更を検出するとシリアライズ化されます。クラスローダが変更されるのは、**Web** アプリケーションが再デプロイされた場合、サーブレットで動的な変更があった場合、または **Web** アプリケーション間で転送やインクルードがあった場合です。

サーブレットの動的な変更時に属性がシリアライズされないようにするには、`weblogic.xml`にある `servlet-reload-check-secs` を無効にします。Web アプリケーション間の処理を行った場合や Web アプリケーションを再デプロイメントした場合に、属性のシリアライズ化を回避する方法はありません。

セッションの永続性のコンフィグレーション

永続セッションの設定の詳細については、「セッション永続性のコンフィグレーション」を参照してください。

サーブレットでのクッキーの使い方

クッキーは情報の一片です。サーバはこの情報をユーザのディスク上へローカルに保存するよう、クライアントブラウザに要求します。ブラウザは、同じサーバにアクセスするたび、HTTP リクエストと共にそのサーバに関連したクッキーをすべて送ります。クッキーは、クライアントからサーバに戻されるので、そのクライアントを識別するために便利なものです。

各クッキーは名前と値を持っています。通常、クッキーをサポートしているブラウザでは、各サーバのドメインに、1つあたり最大 4K のクッキーを 20 まで格納することができます。

HTTP サーブレットでのクッキーの設定

ブラウザ上にクッキーを設定するには、クッキーを作成し、値を与え、サーブレットのサービスメソッドの 2 番目のパラメータである `HttpServletResponse` オブジェクトに追加します。次に例を示します。

```
Cookie myCookie = new Cookie("ChocolateChip", "100");
myCookie.setMaxAge(Integer.MAX_VALUE);
response.addCookie(myCookie);
```

上記のサンプルでは、値が 100 の `ChocolateChip` と呼ばれるクッキーを、応答の送信時にブラウザクライアントに追加します。クッキーの有効期限は指定できる最大値に設定されているため、クッキーは永久に有効です。クッキーは文字列型の値のみを受け入れるので、クッキーに格納するための必要な型との間で

キャストします。EJB の場合、一般的にはクッキーの値に対する EJB インスタンスのホーム ハンドルを使用し、EJB にユーザの詳細情報を格納して、後で参照できるようにします。

HTTP サーブレットでのクッキーの取得

`service()` メソッドへの引数としてサーブレットに渡される `HttpServletRequest` から、クッキー オブジェクトを取得することができます。クッキーそのものは、`javax.servlet.http.Cookie` オブジェクトとして示されます。

サーブレット コードでは、`getCookies()` メソッドを呼び出すことにより、ブラウザから送られたすべてのクッキーを取得することができます。次に例を示します。

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

このメソッドは、ブラウザから送られたすべてのクッキーの配列を返すか、またはブラウザから送られたクッキーがない場合、`null` を返します。サーブレットは、その配列を処理して正しい名前のクッキーを探す必要があります。クッキーの名前は、`Cookie.getName()` メソッドを使って取得することができます。同一の名前で、パス属性の異なるクッキーが複数ある可能性があります。サーブレットが、同一の名前でパス属性の異なる複数のクッキーを設定した場合、`Cookie.getPath()` メソッドを使ってこれらと比較することも必要です。以下のコードは、ブラウザから送られたクッキーの詳細にアクセスする方法を示しています。このサーバに送られたクッキーはすべてユニークな名前を持ち、ブラウザクライアントで以前に設定したと考えられる `ChocolateChip` というクッキーを探しているという前提です。

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

for(int i=0; i < cookies.length; i++) {
    thisCookie = cookies[i];
    if (thisCookie.getName().equals("ChocolateChip")) {
        cookieFound = true;
        break;
    }
}

if (cookieFound) {
    // クッキーが見つかったので、その値を取得する
    int cookieOrder = String.parseInt(thisCookie.getValue());
}
```

クッキーの詳細については、以下を参照してください。

「Cookie API」

<techdocs/api/javax/servlet/http/Cookie.html>

HTTP と HTTPS の両方で送信されるクッキーの使い方

HTTP リクエストと HTTPS リクエストは異なるポートに送られるので、ブラウザによっては、HTTP リクエストに入れて送られてきたクッキーを、その後続の HTTPS リクエストに包含しない(あるいはその逆)ことがあります。このような場合には、サーブレット リクエストが HTTP と HTTPS の間で切り替わると、新しいセッションが作成されることとなります。セッション内でリクエストが行われるたびに、特定のドメインによって設定されるクッキーがサーバに送られるようにするには、CookieDomain 属性をドメイン名に設定します。CookieDomain 属性は、サーブレットが含まれる 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) の <session-descriptor> 要素で設定します。次に例を示します。

```
<session-descriptor>
  <session-param>
    <param-name>CookieDomain</param-name>
    <param-value>mydomain.com</param-value>
  </session-param>
</session-descriptor>
```

CookieDomain 属性は、mydomain.com によって指定されたドメイン内のホストに、すべてのリクエストについて適切なクッキーを入れるようブラウザに指示します。このプロパティやセッションクッキーのコンフィギュレーションの詳細については、「セッション管理の設定」を参照してください。

セキュアなクッキー

CookieSecure パラメータを使用すると、sessionCookie が安全であるとマークすることができます。このパラメータを設定すると、クライアントのブラウザは HTTPS 接続だけを使用してクッキーを送信します。これにより、クッキー ID がセキュリティで保護されるので、HTTPS だけを使用する Web サイトでのみ使用されます。この機能を有効にすると、HTTP 経由のセッションクッキーは無効に

なります。クライアントが **HTTPS** を使用していないサイトにリダイレクトされると、セッションは送信されません。この機能を使用する場合は **URLRewriting** を無効にすることを強くお勧めします。アプリケーションが **URL** をエンコードしようとする、セッション ID が **HTTP** 上で共有されてしまいます。この機能を使用するには、`weblogic.xml` の `<session-descriptor>` 要素に以下を追加します。

```
<session-param>
<param-name>CookieSecure</param-name>
<param-value>>true</param-value>
</session-param>
<session-param>
```

アプリケーションのセキュリティとクッキー

クッキーの使用は、マシン上での自動的なアカウント アクセスを可能にして便利ですが、セキュリティの観点からは望ましくない場合があります。クッキーを使用するアプリケーション設計時には、以下のガイドラインに従ってください。

- クッキーが常にユーザに対して正確であると考えないようにします。マシンが共有されている場合や、同一のユーザが異なるアカウントにアクセスしようとする場合もあります。
- ユーザが、サーバ上にクッキーを残すかどうか選択できるようにします。共有マシンでは、ユーザがそのアカウントに対する自動的なログインをそのままにしておくことを望まない場合があります。ユーザがクッキーについて理解していると仮定せずに、以下のような質問をします。
「このコンピュータから自動ログインしますか？」
- 注意の必要なデータを取得するためにログインするユーザに対しては、常にパスワードを要求します。ユーザがそれ以外の方法を要求しない限り、この選択結果とパスワードはユーザのセッション データに格納できます。セッションのクッキーは、ユーザがブラウザを終了したときに有効期限が切れるようにコンフィグレーションします。

応答のキャッシュ

キャッシュ フィルタは、次の例外を除いて、キャッシュ タグと同様に動作します。

- JSP フラグメント レベルではなく、ページ レベル (インクルードされたページ) でキャッシュ します。
- ドキュメント内でキャッシュ パラメータを宣言する代わりに、Web アプリケーションのコンフィグレーション内でパラメータを宣言できます。

キャッシュ フィルタには、別のページに含まれていなかったページのためのキャッシュ タグにはないデフォルト動作があります。キャッシュ フィルタは、応答ヘッダ **Content-Type** と **Last-Modified** を自動的にキャッシュ します。キャッシュ フィルタは、キャッシュ 内に存在しているページの リクエストを受け取ると、リクエストの **If-Modified-Since** ヘッダと **Last-Modified** ヘッダを比較して、実際にコンテンツを提供するか、302 `SC_NOT_MODIFIED` ステータスと空のコンテンツを送信するかを決定します。

表 3-3 応答キャッシュの属性

属性	必須 / 任意	デフォルト値	説明
<code>timeout</code>	いいえ	-1	<p>キャッシュ タイムアウト プロパティ。 <code>cache</code> 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。</p> <p>ms = ミリ秒 s = 秒 (デフォルト) m = 分 h = 時間 d = 日</p>

3 プログラミング タスク

属性	必須 / 任意	デフォルト値	説明
scope	いいえ	application	<p>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。</p> <ul style="list-style-type: none">■ 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■ cluster、クラスタ スコープ (対応するクラスタ リスナのコンフィグレーションが必要) <p>ほとんどのキャッシュは、session または application となる。</p>

属性	必須 / 任意	デフォルト値	説明
key	いいえ	--	<p>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。通常、キャッシュは <code>web.xml</code> でコンフィグレーションしたキャッシュ名で識別される。キャッシュ名が指定されていない場合、リクエスト URI がキャッシュ名として使用される。キーを使用すると、タグを識別するための値を追加できる。たとえば、特定のエンドユーザ用のキャッシュを取り出す場合、キャッシュ名以外に、<code>userid</code>、リクエストパラメータスコープ (クエリパラメータ / ポストパラメータ) から取り出す値にクライアント <code>ip</code> を加えてキーを指定できる。キーの指定は、</p> <p>「<code>parameter.userid,parameter.clientip</code>」となる。ここで、「<code>parameter</code>」はスコープ (リクエストパラメータのスコープ)、「<code>userid</code>」と「<code>clientip</code>」はパラメータまたは属性。つまり、キャッシュのプライマリキーはキャッシュ名 (このケースではリクエスト <code>uri</code>) + <code>userid</code> リクエストパラメータ + <code>clientip</code> リクエストパラメータとなる。</p> <p>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。</p> <pre>parameter.key page.key request.key application.key session.key</pre> <p>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、<code>cache</code> 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。</p>

3 プログラミング タスク

属性	必須 / 任意	デフォルト値	説明
<code>async</code>	いいえ	<code>false</code>	<code>async</code> パラメータを <code>true</code> 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。
<code>name</code>	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、 <code>weblogic.jsp.tags.CacheTag</code> 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。

属性	必須 / 任意	デフォルト値	説明
size	いいえ	-1 (無制限)	<p>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。</p>
vars	いいえ	--	<p>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。</p> <p>入力キャッシュには変数が使用される。キャッシュを取り出すと、変数は指定したスコープに戻される。たとえば、データベースから結果を取り出すために、リクエストパラメータから var1 を、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、これらの変数の値がキャッシュ内に格納される。次にキャッシュにアクセスする場合、これらの値は元に戻るのので、それぞれのスコープから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可能になる。</p>

属性	必須 / 任意	デフォルト値	説明
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。

次の例は、Web アプリケーション内のすべての HTML ページをキャッシュするキャッシュ フィルタの登録方法を示しています。

```
<filter>
  <filter-name>HTML</filter-name>
  <filter-class>weblogic.cache.filter.CacheFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>HTML</filter-name>
  <url-pattern>*.html</url-pattern>
</filter-mapping>
```

このキャッシュ システムは、ソフト リファレンスを使用してキャッシュを格納します。ガベージ コレクタは、キャッシュが作成または最終アクセスされてから経過した時間に基づいて、キャッシュを再要求するかどうか決定します。そして、`OutOfMemoryError` が発生しないように、ソフト リファレンスをクリアします。

初期化パラメータ

Web ページが更新されたときに新しいコピーをキャッシュに取り込みたい場合は、フィルタにタイムアウトを追加します。キャッシュ フィルタには、キャッシュ タグと同じような初期化パラメータが多数用意されています。

初期化パラメータは次のとおりです。

- **Name** — キャッシュの名前。*. 拡張子の URL パターンとの互換性を考慮して、リクエスト URI がデフォルト名になります。
- **Timeout** — キャッシュの最終更新時から、次にキャッシュ内のコンテンツを更新するまでの時間。デフォルトの単位は「秒」ですが、ms (ミリ秒)、s

(秒)、m(分)、h(時)、d(日)のいずれかの単位を指定することもできます。

- **Scope** – キャッシュのスコープには、*request*、*session*、*application*、*cluster* のいずれかを指定できる。*request* スコープは、ページ内にループ構造がある場合に便利です。デフォルトでは、スコープは *application* に設定されています。*cluster* スコープを使用するには、*ClusterListener* を設定する必要があります。
- **Key** – このパラメータは、キャッシュが、*name* だけでなく、スコープ内のさまざまなエントリの値で指定されていることを示す。*CacheTag* のように *page* スコープを使用することはできませんが、それ以外は *CacheTag* のキーと同じ要領で指定します。
- **Vars** – キャッシュしたいページに自動的に計算させる変数。一般に、入力パラメータに基づいてデータベースから情報を引き出すサブレットで使用されます。
- **Size** – キャッシュされるユニークなキー値の最大数。デフォルトでは、無制限になっています。

次の例は、フィルタ コード内の初期化パラメータの指定場所を示します。

```
<filter>
  <filter-name>HTML</filter-name>
  <filter-class>weblogic.cache.filter.CacheFilter</filter-class>
  <init-param>
```

HTTP サブレットからの WebLogic サービスの使い方

HTTP サブレットを記述する際には、JNDI、RMI、EJB、JDBC 接続など、WebLogic Server の豊富な機能を利用できます。

以下のマニュアルには、これらの機能の詳細が記載されています。

- 『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』
- 『WebLogic JDBC プログラマーズ ガイド』

- 『WebLogic JNDI プログラマーズ ガイド』
- 『WebLogic JMS プログラマーズ ガイド』

データベースへのアクセス

WebLogic Server は、サーバサイド Java クラス (サーブレットなど) からの Java Database Connectivity (JDBC) の使用をサポートしています。JDBC を使うと、Java クラスから SQL クエリを実行し、クエリの結果を処理できます。JDBC と WebLogic Server の詳細については、『WebLogic JDBC プログラマーズ ガイド』を参照してください。

以下の節で説明するように、JDBC はサーブレットで使用できます。

- 3-34 ページの「JDBC 接続プールを用いたデータベースへの接続」。
- 3-36 ページの「DataSource オブジェクトを用いたデータベースへの接続」。
- 3-37 ページの「JDBC ドライバを用いたデータベースへの直接接続」。

JDBC 接続プールを用いたデータベースへの接続

接続プールとは、接続プールが登録される時 (通常は WebLogic Server の起動時) に作成される、データベースへの同一 JDBC 接続のグループに名前を付けたものです。サーブレットはプールから接続を「借り」、使用後に接続を閉じることでプールに接続を返します。このプロセスは、接続プールの使用は、データベースへのアクセスが必要になるたびにクライアントごとに新しい接続を確立するよりもはるかに効率的です。もう 1 つの利点は、データベースについての詳細をサーブレットのコードに組み込む必要がないということです。

JDBC 接続プールに接続するには、次の多層 JDBC ドライバのうち 1 つを使用します。

- プール ドライバ。ほとんどのサーバサイド処理に使用します。
 - ドライバ URL : `jdbc:weblogic:pool`
 - ドライバ パッケージ名 : `weblogic.jdbc.pool.Driver`

- **JTS プール ドライバ**。データベース操作にトランザクションのサポートが必要な場合に使用します。
 - ドライバ URL :`jdbc:weblogic:jts`
 - ドライバ パッケージ名 :`weblogic.jdbc.jts.Driver`

サーブレットでの接続プールの使い方

次の例では、サーブレットからのデータベース接続プールの使い方を示します。

1. プール ドライバをロードし、`java.sql.Driver` にキャストします。ドライバの絶対パス名は、`weblogic.jdbc.pool.Driver` です。次に例を示します。

```
Driver myDriver = (Driver)
    Class.forName("weblogic.jdbc.pool.Driver").newInstance();
```

2. ドライバの URL を使用した接続に加えて、登録された接続プールの名前(省略可能)を作成します。プール ドライバの URL は、`jdbc:weblogic:pool` です。

プールは、以下の 2 通りの方法で識別できます。

- `connectionPoolID` キーを使用して、`java.util.Properties` オブジェクトで接続プール名を指定する。次に例を示します。

```
Properties props = new Properties();
props.put("connectionPoolID", "myConnectionPool");
Connection conn =
    myDriver.connect("jdbc:weblogic:pool", props);
```

- プール名を URL の末尾に追加する。この場合、プールからの接続を使用するためにユーザ名とパスワードを設定していない限り、`Properties` オブジェクトは不要です。次に例を示します。

```
Connection conn =
    myDriver.connect("jdbc:weblogic:pool:myConnectionPool",
    null);
```

上の例では、`DriverManger.getConnection()` メソッドの代わりに、`Driver.connect()` メソッドが使われています。データベース接続を取得するために `DriverManger.getConnection()` を使用することもできますが、`Driver.connect()` を使用することをお勧めします。このメソッドは同期を取られることがなく、よりよいパフォーマンスを提供するためです。

`connect()` が返す **Connection** は、`weblogic.jdbc.pool.Connection` のインスタンスであることに注意してください。

3. JDBC の呼び出しが終わったら、**Connection** オブジェクトに対して `close()` メソッドを呼び出して、接続を正しくプールに戻します。コーディング方法としては、`try` ブロックに接続を作成してから、`finally` ブロックで接続を閉じ、いかなる場合でも確実に接続がクローズされるようにしてください。

```
conn.close();
```

DataSource オブジェクトを用いたデータベースへの接続

DataSource は、接続プールを参照するサーバサイドオブジェクトです。接続プールの登録により、**JDBC** ドライバ、データベース、ログインなど、データベース接続と関連するパラメータが定義されます。**DataSource** オブジェクトおよび接続プールは、**Administration Console** で作成します。**J2EE** 準拠のアプリケーションを作成する場合は、**DataSource** オブジェクトの使用をお勧めします。

サーブレットでの DataSource の使用

1. **Administration Console** を使って接続プールを登録します。詳細については、「**[JDBC 接続プール]**--**[コンフィグレーション]**--**[接続]**」を参照してください。
2. 接続プールを指す **DataSource** オブジェクトを登録します。詳細については、「**[JDBC データ ソース]**--**[コンフィグレーション]**」を参照してください。
3. **JNDI** ツリーで、**DataSource** オブジェクトをルックアップします。次に例を示します。

```
Context ctx = null;  
  
// JNDI ルックアップのためのコンテキストを取得する  
ctx = new InitialContext(ht);
```

```
// DataSource オブジェクトをルックアップする
javax.sql.DataSource ds
    = (javax.sql.DataSource) ctx.lookup ("myDataSource");
```

4. DataSource を使用して、JDBC 接続を作成します。次に例を示します。

```
java.sql.Connection conn = ds.getConnection();
```

5. 接続を使用して、SQL 文を実行します。次に例を示します。

```
Statement stmt = conn.createStatement();
stmt.execute("select * from emp");
```

```
...
```

JDBC ドライバを用いたデータベースへの直接接続

データベースへの直接接続は、データベース接続を確立する方法としては、最も効率の悪いものです。リクエストごとに新しいデータベース接続を確立しなければならないからです。データベースへの接続には、どの JDBC ドライバも使用できます。BEA では、Oracle および Microsoft SQL Server 用の JDBC ドライバを提供しています。詳細については、『WebLogic JDBC プログラマーズ ガイド』を参照してください。

HTTP サーブレットにおけるスレッドの問題

サーブレットの設計時に、高い負荷のもとで、WebLogic Server がサーブレットをどのように呼び出すか検討する必要があります。複数のクライアントが同時にサーブレットをヒットすることは避けられません。したがって、サーブレットのコードは、共有リソースやインスタンス変数の共有違反を防ぐように記述します。この問題を避けて設計するためのヒントを以下に示します。

SingleThreadModel

SingleThreadModel を実装したクラスのインスタンスは、同時に複数のスレッドで呼び出されることが保証されています。SingleThreadModel サブプレットの複数のインスタンスを使用して、それぞれをシングル スレッドで実行しながら、同時に発生するリクエストを処理します。

SingleThreadModel を効率的に使用するため、**WebLogic Server** は SingleThreadModel を実装する各サブプレットについて、サブプレット インスタンスのプールを作成します。サブプレットに対する最初のリクエストが行われると、**WebLogic Server** はサブプレット インスタンスのプールを作成し、必要に応じてプール内のサブプレット インスタンスの数を増やしていきます。

[シングル スレッド サブプレットのプール サイズ] 属性には、サブプレットに初めてリクエストが送られたときに作成されるサブプレット インスタンスの初期数を指定します。この属性は、SingleThreadModel サブプレットが処理する予定の同時リクエストの平均数に設定します。

サブプレットの設計時に、ファイルやデータベースへのアクセスのようなサブプレット クラスの外部の共有リソースの使い方に注意を払う必要があります。同一のサブプレット インスタンスが複数存在し、まったく同じリソースを使用する可能性があるため、SingleThreadModel を実装した場合でも、解決の必要がある同期と共有の問題が発生します。

共有リソース

共有リソースの問題は、個々のサブプレットごとに処理することをお勧めします。次のガイドラインを念頭に置いてください。

- 可能な限り同期を避ける。引き続き発生するサブプレットのリクエストが、現行のスレッドが完了するまで、ボトルネックになるためです。
- 各サブプレット リクエストに固有の変数は、サービス メソッドのスコープ内で定義する。ローカル スコープ変数は、スタックに保存されるため、同一のメソッド内で実行されている複数のスレッドで共有されることはありません。したがって、同期の必要性はなくなります。
- 外部リソースへのアクセスは、**Class** レベルで同期を取るか、トランザクションにカプセル化する。

別のリソースへのリクエストのディスパッチ

この節では、リクエストをサーブレットから別のリソースへディスパッチするのによく使用されるメソッドの概要を説明します。

サーブレットでは、リクエストを別のリソース(サーブレット、JSP、またはHTML ページなど)に渡すことができます。このプロセスは、リクエストのディスパッチと呼ばれます。リクエストをディスパッチする場合は、`RequestDispatcher` インタフェースの `include()` または `forward()` を使用します。`forward()` メソッドまたは `include()` メソッドを使用する場合、出力を応答オブジェクトに書き込める時期には制限があります。この制限についても、この節で説明します。

リクエストのディスパッチに関する詳細な説明については、Sun Microsystems が提供するサーブレット 2.3 仕様 (<http://java.sun.com/products/servlet/download.html#specs>) のセクション 8.1 を参照してください。

`RequestDispatcher` を使用すると、HTTP リダイレクト応答をクライアントに送り返す必要がなくなります。`RequestDispatcher` は、HTTP リクエストを要求されたリソースに渡します。

リソースを特定のリソースにディスパッチするには、以下の手順に従います。

1. 次のように、`ServletContext` への参照を取得します。

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

2. 以下のメソッドの1つを用いて、`RequestDispatcher` オブジェクトをルックアップします。

- `RequestDispatcher rd = sc.getRequestDispatcher(String path);`

`path` は、Web アプリケーションのルートに対する相対パスでなければなりません。

- `RequestDispatcher rd = sc.getNamedDispatcher(String name);`

`name` を、Web アプリケーションのデプロイメント記述子の中で `<servlet-name>` 要素によってそのサーブレットに割り当てられた名前で置き換えます。詳細については、「servlet 要素」を参照してください (`web.xml.html#web_xml_servlet`)。

- `RequestDispatcher rd = ServletRequest.getRequestDispatcher(String path);`

このメソッドは `RequestDispatcher` オブジェクトを返すものであって、`ServletContext.getRequestDispatcher(String path)` メソッドに似ています。ただし、ここでは、`path` を現在のサーブレットに対して相対的になるように指定することができます。「/」記号で始まるパスは、Web アプリケーションに対して相対的になるように解釈されます。

HTTP サーブレット、JSP ページ、通常の HTML ページなど、Web アプリケーション内のどのリソースについても、`getRequestDispatcher()` メソッドでリソースの適切な URL を要求することによって、`RequestDispatcher` を取得できます。返された `RequestDispatcher` を使用して、リクエストを別のサーブレットに転送します。

3. 適切なメソッドを使用して、リクエストを転送またはインクルードします。

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

これらのメソッドは、次の 2 つの節で説明しています。

リクエストの転送

一度、正しい `RequestDispatcher` が得られると、サーブレットは、引数として、`HttpServletRequest` と `HttpServletResponse` を渡し、`RequestDispatcher.forward()` メソッドを使用して、リクエストを転送します。出力が既にクライアントに送られた状態でこのメソッドを呼び出すと、`IllegalStateException` が送出されます。応答バッファの中に、コミットされていない保留中の出力がある場合には、バッファはリセットされます。

サーブレットは、応答に対する以前の出力を書き込もうとしてはいけません。リクエストを転送する前に、応答のためにサーブレットが `ServletOutputStream` または `PrintWriter` を取得すると、`IllegalStateException` が送出されます。

元のサーブレットからのそれ以外の出力はすべて、リクエストが転送された後は無視されます。

どのタイプの認証を使用する場合でも、転送されたリクエストは、デフォルトではユーザに再認証を要求しません。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>
```

デフォルトの動作は、サーブレット仕様 2.3 のリリースで変更されたことに注意してください。サーブレット 2.3 仕様では、転送されたリクエストの認証は要求されることが規定されています。

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

サーブレットには、`RequestDispatcher.include()` メソッドを使用し、引数として `HttpServletRequest` と `HttpServletResponse` を渡すことにより、他のリソースからの出力をインクルードすることができます。その場合、インクルードされたリソースは、リクエスト オブジェクトにアクセスできます。

インクルードされたリソースは、応答オブジェクトの `ServletOutputStream` または `Writer` オブジェクトにデータを書き戻すことができ、その後、応答バッファにデータを追加するか、または応答オブジェクトに対し `flush()` メソッドを呼び出すかのいずれかを行うことができます。応答のステータスコード、またはインクルードされたサーブレットの応答からの **HTTP** ヘッダ情報を設定しようとする、すべて無視されます。

実際には、`include()` メソッドを使用して、サーブレット コードから他の **HTTP** リソースの「サーバサイドインクルード」を実現できます。

4 管理とコンフィグレーション

以下の節では、**WebLogic HTTP** サーブレット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。サーブレットの管理とコンフィグレーションの詳細については、「サーブレットのコンフィグレーション」を参照してください。

この章では次の内容について説明します。

- **WebLogic HTTP** サーブレットの管理の概要
- **Web** アプリケーションでのサーブレットの参照
- **Web** アプリケーションのディレクトリ構造
- サーブレットのセキュリティ
- サーブレット開発のヒント
- サーブレットのクラスタ化

WebLogic HTTP サーブレットの管理の概要

Java 2 Enterprise Edition の規格に準拠するため、**HTTP** サーブレットは **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、サーブレット クラス、**JavaServer Pages (JSP)**、静的な **HTML** ページ、画像、ユーティリティクラスなどのアプリケーション コンポーネントをグループ化したものです。

Web アプリケーションでは、コンポーネントは標準的なディレクトリ構造を用いてデプロイされます。このディレクトリ構造は、**.war** ファイルと呼ばれるファイルにアーカイブされて、**WebLogic Server** 上にデプロイされます。**Web** ア

アプリケーションのリソースと操作パラメータに関する情報は、Web アプリケーションと共にパッケージ化されている 2 つのデプロイメント記述子で定義されます。

サーブレットをコンフィグレーション、デプロイするためのデプロイメント記述子の使い方

第 1 のデプロイメント記述子、`web.xml` は、Sun Microsystems のサーブレット 2.3 仕様に従って定義され、Web アプリケーションを記述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 の動作と 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。

web.xml (Web アプリケーション デプロイメント記述子)

Web アプリケーションのデプロイメント記述子では、HTTP サーブレットの以下の属性を定義します。

- サーブレット名
- サーブレットの Java クラス
- サーブレット初期化パラメータ
- サーブレットの `init()` メソッドを WebLogic Server の起動時に実行するかどうか
- 一致した場合には、サーブレットを呼び出す URL パターン
- 「セキュリティ」
- MIME タイプ
- エラー ページ
- EJB への参照
- 他のリソースへの参照

web.xml ファイルの作成に関する詳細については、「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。

weblogic.xml (WebLogic 固有のデプロイメント記述子)

WebLogic 固有のデプロイメント記述子では、HTTP サーブレットの以下の属性を定義します。

- HTTP セッションのコンフィグレーション
- クッキーのコンフィグレーション
- WebLogic Server に同梱されている SimpleApacheURLMatchMap ユーティリティなどの URL 照合ユーティリティで一致した場合に、このサーブレットを呼び出す URL パターン
- EJB リソースのマッピング
- JSP のコンフィグレーション

weblogic.xml ファイルの作成に関する詳細については、「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。

WebLogic Server Administration Console

WebLogic Server Administration Console を使用して、以下のパラメータを設定します。

- HTTP パラメータ
- ログ ファイル
- URL 書き換え
- キープアライブ
- デフォルト MIME タイプ
- クラスタ化パラメータ
- 仮想ホスティングのための URL マッピング

詳細については、以下のリソースを参照してください。

- Administration Console の「[Web アプリケーション]--[コンフィグレーション]--[ファイル]」
- Administration Console の「仮想ホスト」

Web アプリケーションのディレクトリ構造

すべての Web アプリケーションについて、以下のディレクトリ構造を使用します。

```

Default WebApp/(.jsp、.html、.jpg、.gif などの
                  公開されるファイル)
|
+WEB-INF/--+
|
+ classes/(Web アプリケーションで
            使用されるサーブレットなどの
            Java クラスを
            格納するディレクトリ)
|
+ lib/(Web アプリケーションで使用
       される jar ファイルを
       格納するディレクトリ)
|
+ web.xml
|
+ weblogic.xml

```

Web アプリケーションでのサーブレットの参照

Web アプリケーションでサーブレットを参照するための URL は、次のように構成されます。

```
http:// myHostName:port/myContextPath/myRequest/?myRequestParameters
```

URL の各要素は次のように定義します。

myHostName

WebLogic Server Administration Console で定義される、WebLogic Server にマップされる DNS 名です。

URL のこの部分は、`host:port` に置き換えることができます。host は、WebLogic Server が実行されているマシン名、port は WebLogic Server がリクエストをリスンしているポートです。

port

WebLogic Server がリクエストをリスンしているポートです。サーブレットは、Server MBean と SSL MBean 上の `listenPort` ポートを通じてのみ、プロキシと通信することができます。

myContextPath

`weblogic.xml` ファイルで指定されるコンテキスト ルート名、または `config.xml` ファイルで指定される Web モジュールの URI。

myRequest

`web.xml` で定義されるサーブレット名です。

myRequestParameters

URL にエンコードされる HTTP リクエスト パラメータです (省略可能)。HTTP サーブレットで読み取り可能です。

URL パターン マッチング

WebLogic Server には、J2EE のマッチング ルールに適合していない URL マッチング ユーティリティを実装する機能があります。マッチングユーティリティは、デフォルトで実装される `URLMatchMap` とは異なり、`web.xml` デプロイメント記述子ではなく、`weblogic.xml` デプロイメント記述子に指定します。

URL マッチング ユーティリティを WebLogic Server で使用するには、次のインタフェースを実装する必要があります。

```
Package weblogic.servlet.utils;

public interface URLMapping {

    public void put(String pattern, Object value);

    public Object get(String uri);

    public void remove(String pattern)

    public void setDefault(Object defaultObject);
```

```
public Object getDefault();

public void setCaseInsensitive(boolean ci);

public boolean isCaseInsensitive();

public int size();

public Object[] values();

public String[] keys();
}
```

SimpleApacheURLMatchMap ユーティリティ

同梱されている **SimpleApacheURLMatchMap** ユーティリティは、J2EE 固有のユーティリティではありません。このユーティリティは、**weblogic.xml** デプロイメント記述子ファイルに指定します。このユーティリティを使用すると、ユーザは、**web.xml** デプロイメント記述子に指定されたデフォルトの URL パターンマッチングではなく、**Apache** スタイルのパターン マッチングを指定することができます。

サーブレットのセキュリティ

サーブレットのセキュリティは、そのサーブレットが含まれる **Web** アプリケーションのコンテキストで定義されます。セキュリティは **WebLogic Server** で処理することも、プログラムによってサーブレット クラスに組み込むこともできます。

詳細については、「**Web** アプリケーションでのセキュリティのコンフィグレーション」を参照してください。

認証

次の3つの手法のうちいずれかを使用して、サーブレットにユーザ認証を組み込むことができます。

- **BASIC** – ブラウザを使って、ユーザ名とパスワードを収集します。
- **FORM** – HTMLのフォームを使って、ユーザ名とパスワードを収集します。
- **クライアント証明書** – デジタル証明書を使って、ユーザを認証します。詳細については、「デジタル証明書」を参照してください。

BASIC および **FORM** の手法は、ユーザとパスワードの情報が格納されたセキュリティロール内に呼び出しを行うものです。**WebLogic Server** に付属しているデフォルトのロールを使うことも、**Windows NT**、**UNIX**、**RDBMS** の各ロール、ユーザ定義のロールなど、既存のさまざまなロールを使うこともできます。セキュリティロールの詳細については、「セキュリティの基礎概念」を参照してください。

認可 (セキュリティ制約)

セキュリティ制約を使用すると、**Web** アプリケーションにおけるサーブレットなどのリソースへのアクセスを制限することができます。セキュリティ制約は、**Web** アプリケーション デプロイメント記述子 (**web.xml**) で定義されています。セキュリティ制約には、3つの基本的なタイプがあります。

- ロールやリソースによるリソースの制約
- セキュアソケットレイヤ (**SSL**) の暗号化
- プログラムによる認可

ロールは、プリンシパルにマップできます。特定リソースの制約は、**URL** パターンと **Web** アプリケーション内のリソースを一致させることにより実現します。また、セキュリティ制約としてセキュアソケットレイヤ (**SSL**) を使用できます。

`HttpServletRequest` インタフェースの次のいずれかのメソッドを使用してプログラミングし、認可を実行することも可能です。

- `getRemoteUser()`
- `isUserInRole()`
- `getUserPrincipal()`

詳細については、`javax.servlet API` を参照してください。

サーブレット開発のヒント

HTTP サーブレットを作成する際には、次のヒントを考慮してください。

- サーブレットは、Web アプリケーションの `WEB-INF/classes` ディレクトリにコンパイルします。
- サーブレットは、必ず Web アプリケーションのデプロイメント記述子 (`web.xml`) に登録します。
- サーブレットのリクエストに応答する際、**WebLogic Server** はサーブレットに関連付けられているフィルタを適用する前に、サーブレット クラス ファイルのタイム スタンプをチェックし、メモリ内にある既存のサーブレット インスタンスと比較します。バージョンの新しいサーブレット クラスがあれば、**WebLogic Server** は、フィルタ処理の前にすべてのサーブレット クラスを再ロードします。サーブレットが再ロードされると、そのサーブレットの `init()` メソッドが呼び出されます。すべてのサーブレットが再ロードされるのは、サーブレット クラスの変更が見つかり、サーブレット クラスが相互に依存している可能性がある場合です。

WebLogic Server がタイム スタンプをチェックする間隔 (秒単位) を [サーブレット再ロードのチェック間隔 (秒)] 属性で設定できます。この属性は、**Administration Console** で、Web アプリケーションの [ファイル] タブで設定します。この属性をゼロにすると、**WebLogic Server** はリクエストごとにタイム スタンプをチェックします。これはサーブレットの開発とテスト中には便利ですが、プロダクション環境では必要以上に時間を消費します。この属性を `-1` に設定すると、**WebLogic Server** は変更されたサーブレットについてはチェックを行いません。

サーブレットのクラスタ化

サーブレットをクラスタ化すると、フェイルオーバーとロード バランシングのメリットを活かせます。WebLogic Server のクラスタにサーブレットをデプロイするには、サーブレットを含む Web アプリケーションをクラスタ内の全サーバにデプロイします。手順については、『WebLogic Server クラスタ ユーザーズ ガイド』の「アプリケーションをクラスタにデプロイする」を参照してください。

サーブレットのクラスタ化に関する要件と、クラスタ化されたサーブレットに送られるリクエストのフェイルオーバープロセスの詳細については、『WebLogic Server クラスタ ユーザーズ ガイド』の「サーブレットと JSP のレプリケーションとフェイルオーバー」を参照してください。

注意： サーブレットの自動フェイルオーバーには、サーブレット セッション ステートをメモリ内にレプリケートする必要があります。手順については、『WebLogic Server クラスタ ユーザーズ ガイド』の「インメモリ HTTP レプリケーションをコンフィグレーションする」を参照してください。

サーブレットに関して WebLogic Server クラスタがサポートしているロード バランシングの詳細と、関連するプランニングとコンフィグレーションに関する開発者および管理者向け考慮事項については、『WebLogic Server クラスタ ユーザーズ ガイド』の「サーブレットと JSP のロード バランシング」を参照してください。

索引

A

addCookie() 3-23
Administration Console 4-4
API 1-4

C

contentType 2-2

D

DataSource 3-34, 3-36

E

EJB 3-33
encodeURL() 3-20

F

forward() 3-39

G

getAttribute() 3-17
getAttributeNames() 3-17
getCookies() 3-24
getParameterValues() 3-10
getSession() 3-14, 3-16

H

HelloWorldServlet 2-5
HTTP
 応答 3-4
HttpServletRequest 2-1
 メソッド 3-8

HttpServletResponse 2-1, 3-4
HttpSession オブジェクト 3-14

I

IDLength 3-21
IllegalStateException 3-17
include() 3-39
init パラメータ 3-2
init() メソッド 3-2, 3-3
init-param 3-3

J

J2EE 1-3
javax.servlet 1-4
JDBC 3-33, 3-34, 3-37
JDBC セッション永続性 3-21
JMS 3-33
JNDI 3-33
JTS プール ドライバ 3-35

P

PrintWriter オブジェクト 2-2

R

removeAttribute() 3-17
RequestDispatcher() 3-39

S

setAttribute() 3-17
SimpleApacheURLMatchMap ユーティリティ 4-7
SingleThreadModel 3-38

SingleThreadModelPoolSize 3-38

U

URL 4-5

URL 書き換え 3-19

WAP 3-21

Wireless Access Protocol 3-21

URL パターン マッチング 4-6

W

WAP 3-21

Web アプリケーション

URL 4-5

セキュリティ 4-7

ディレクトリ構造 4-5

デプロイメント記述子 4-2

web.xml 4-2

weblogic.xml 4-2

Wireless Access Protocol 3-21

い

インクルード 3-39

インクルード、リクエスト 3-41

印刷、製品のマニュアル viii

インポート 2-1

インメモリ レプリケーション 3-21

お

応答 3-4

最適化 3-5

バッファ 3-6

応答キャッシュの属性 3-27

応答のキャッシュ 3-27

か

開発 1-3

クラスパス 2-2

コンパイル 4-9

ヒント 4-9

開発環境 2-2

カスタマ サポート情報 ix

環境、開発

環境 2-2

管理

コンソール 4-4

き

キープ アライブ 3-5

く

クエリ パラメータ 3-6, 3-7, 3-9

クッキー 3-23

EJB 3-24

HTTP と HTTPS 3-25

サーブレットでの使い方 3-23

取得 3-24

ドメイン 3-25

パスワード 3-26

ログイン 3-26

クラスタ化 3-21, 4-10

クラスパス 2-2

こ

コンパイル 2-2

さ

サービス メソッド 2-1

サーブレット

クラスタ化 4-10

サーブレット 2.2 仕様 1-4

サポート

技術情報 ix

し

初期化

init() メソッド 3-2

パラメータ 3-2

す

スレッド 3-37

SingleThreadModel 3-38

せ

セキュリティ 4-7

制約 4-8

認可 4-8

認証 4-8

プログラムによる適用 4-8

レルム 4-8

セキュリティ制約 4-8

セッション

encodeURL() メソッド 3-20

HttpSession オブジェクトを用いたト
ラッキング 3-14

URL 書き換え 3-19

永続性 3-21

開始の検出 3-16

クッキー 3-16, 3-19

クラスタ 3-21

終了 3-17

トラッキング 3-12, 3-16

トラッキング、コンフィグレーション
3-19

トラッキングの履歴 3-13

名前/値の属性 3-17

有効期間 3-15

ログアウト 3-17

セッション永続性

JDBC 3-21

接続プール 3-34

DataSource 3-36

JDBC 3-34

使用 3-35

ドライバ 3-35

て

ディスパッチ 3-39

データベース 3-34

デプロイメント 2-3

デプロイメント記述子 4-2

転送 3-39, 3-40

な

名前と値の組み合わせ 3-17

に

入力

クエリ パラメータ 3-9

入力の取得 3-6

認証 4-8

は

パッケージ 2-1

ふ

プール ドライバ 3-34

ま

マニュアル、入手先 viii

り

リクエスト

インクルード 3-39, 3-41

ディスパッチ 3-39

転送 3-39, 3-40

ろ

ログアウト 3-17