BEA WebLogic Server™

WebLogic XML 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 : 2003 年 5 月 22 日

WebLogic XML プログラマーズ ガイド

パート番号	マニュアルの日付	ソフトウェアのバージョン
なし	2003 年 5 月 22 日	BEA WebLogic Server バージョン 7.0

目次

このマニュアルの内容

1. XML の概要

2. WebLogic Server による XML アプリケーションの開発

3. XML アプリケーション スコーピング

4. WebLogic XML Streaming API の使い方

5. XML プログラミングのベストプラクティス

6. XML プログラミング手法

7. WebLogic Server XML の管理

8. XML リファレンス

このマニュアルの内容

このマニュアルでは、BEA WebLogic Server™ XML ソフトウェアの使い方について説明します。XML ソフトウェアの使用に関連する概念を定義し、XML アプリケーションの開発プロセスについて述べています。また、このマニュアルでは、アプリケーション プログラミング インタフェース (API)、管理タスク、および XML ツールについても説明します。

このマニュアルの構成は次のとおりです。

- 第 1 章「XML の概要」は、XML ソフトウェアとそのコンポーネントの基本的な説明です。
- 第 2 章「WebLogic Server による XML アプリケーションの開発」では、WebLogic Server と XML ツールを使用して XML アプリケーションを開発する方法について説明します。
- 第 3 章「XML アプリケーション スコーピング」では、パーサ、トランスフォーマ、および特定のエンタープライズ アプリケーション用外部エンティティのコンフィグレーション方法について説明します。
- 第 4 章「WebLogic XML Streaming API の使い方」では、XML ドキュメントを解析する Java アプリケーションで WebLogic XML ストリーミング API を使用する方法に関する詳細を説明します。
- 第 5 章「XML プログラミングのベストプラクティス」では、XML ドキュメントを処理する Java アプリケーションを作成する場合のベスト プラクティスをいくつか説明します。
- 第 6 章「XML プログラミング手法」では、XML ドキュメントにおけるメッセージ駆動型 Bean および JMS キューの使用といった、特定のタスクを処理するためのプログラミング技術について説明します。
- 第 7 章「WebLogic Server XML の管理」では、Administration Console の XML レジストリ、および XML のコンフィグレーション タスクを行う方法について説明します。

- 第 8 章「XML リファレンス」では、この XML ソフトウェアでサポートしている仕様およびアプリケーション プログラミング インタフェースへのリンクを提供しています。

対象読者

このマニュアルは、XML アプリケーションを設計、開発、コンフィグレーション、および管理するシステム管理者およびプログラマを対象としています。Web 技術、XML、XSLT、Java プログラミング言語、サーブレット、および J2EE 仕様の JSP API 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

XML の詳細については、1-18 ページの「XML について学習するには」および第 8 章「XML リファレンス」を参照してください。

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。
斜体	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:
	`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
{ }	構文の中で複数の選択肢を示す。

表記法	適用
[ ]	構文の中で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 XML の概要

以下の節では XML テクノロジと WebLogic Server XML サブシステムについて概説します。

XML とは

Extensible Markup Language (XML) とは、ドキュメント内のデータの内容および構造を示すのに使用するマークアップ言語です。XML は Standardized General Markup Language (SGML) を簡素化したものです。XML は、インターネット上でコンテンツを配信するための業界標準です。また、新しいタグを定義する機能があるので、拡張が可能です。

HTML と同じように、XML でもタグを使用してコンテンツを記述します。しかし、XML のタグはコンテンツの表示方法ではなく、データの意味と階層構造を表します。この機能を使用すれば、異なるプログラムやシステム間でのデータ交換を効率化するために必要な高度なデータ型を定義できます。さらに、XML ではコンテンツと表現を分離できるため、コンテンツ (データ) を異種システム間で移植できます。

XML 構文では、一対の開始タグと終了タグ (<name> と </name> など) を使用して情報をマークアップします。タグで区切られる情報のことを、要素と呼びます。どの XML ドキュメントにもルート要素が 1 つあります。ルート要素は、他のすべての要素を含む最上位の要素です。他の要素内に含まれる要素をしばしば下位要素と言います。要素には、名前と値のペアという構造の属性が付くことがあります。属性は、要素の一部として要素をさらに定義するのに使用します。

次のサンプル XML ファイルはアドレス帳の内容を説明したものです。

```
<?xml version="1.0"?>

<address_book>
  <person gender="f">
    <name>Jane Doe</name>
    <address>
      <street>123 Main St.</street>
      <city>San Francisco</city>
      <state>CA</state>
      <zip>94117</zip>
    </address>
    <phone area_code=415>555-1212</phone>
  </person>
  <person gender="m">
    <name>John Smith</name>
    <phone area_code=510>555-1234</phone>
    <email>johnsmith@somewhere.com</email>
```

```
  </person>
</address_book>
```

XML ファイルのルート要素は address_book です。アドレス帳には現在、person 要素の形で「Jane Doe」と「John Smith」の 2 つのエントリがあります。Jane Doe のエントリには住所と電話番号、John Smith のエントリには電話番号と電子メール アドレスがあります。この XML ドキュメントの構造では、要素の本体の下位要素ではなく、phone 要素に area_code 属性を指定して市外局番を格納するように定義しています。すべての下位要素が person 要素で必要なわけではないことにも注意してください。

XML ドキュメントの記述方法

XML ドキュメントの記述方法には、DTD および XML スキーマの 2 種類があります。

DTD (Document Type Definition : 文書型定義) は、特定の XML ドキュメントの構造で基本的に必要なものを定義します。DTD では、XML ドキュメントで有効な要素と属性、およびそれらが有効となるコンテキストを定義します。言い換えれば、DTD では特定のタグ内で有効なタグ、およびオプションのタグと属性を指定します。

次の例では、上のアドレス帳のサンプル XML ドキュメントを定義する DTD を示します。

```
<!DOCTYPE address_book [
<!ELEMENT person (name, address?, phone?, email?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (street, city, state, zip)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT zip (#PCDATA)>

<!ATTLIST person gender CDATA #REQUIRED>
<!ATTLIST phone area_code CDATA #REQUIRED>
]>
```

スキーマは、XML の仕様で最近開発されたもので、DTD に取って代わるものとされています。スキーマでは、DTD および DTD 以外の XML ドキュメント自体より柔軟かつ詳細に XML ドキュメントを記述します。スキーマ仕様は、現在 World Wide Web Consortium (W3C) によって開発中であり、DTD の制限の多くを克服するものとされています。XML スキーマの詳細については、http://www.w3.org/TR/xmlschema-0/ を参照してください。

次の例では、上のアドレス帳のサンプル XML ドキュメントを定義するスキーマを示します。

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">

<xsd:element     name="address_book" type="bookType"/>

<xsd:complexType name="bookType">
  <xsd:element name=name="person"     type="personType"/>
</xsd:complexType>

<xsd:complexType name="personType">
  <xsd:element   name="name"     type="xsd:string"/>
  <xsd:element   name="address"  type="addressType"/>
  <xsd:element   name="phone"    type="phoneType"/>
  <xsd:element   name="email"    type="xsd:string"/>
  <xsd:attribute name="gender"   type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="addressType">

  <xsd:element   name="street"  type="xsd:string"/>
  <xsd:element   name="city"    type="xsd:string"/>
  <xsd:element   name="state"   type="xsd:string"/>
  <xsd:element   name="zip"     type="xsd:string"/>
</xsd:complexType>

<xsd:simpleType name="phoneType">
  <xsd:restriction base="xsd:string"/>
  <xsd:attribute name="area_code" type="xsd:string"/>
</xsd:simpleType>

</xsd:schema>
```

XML ドキュメントでは、ドキュメント自体の一部に DTD またはスキーマを含めることも、DOCTYPE 宣言で外部 DTD またはスキーマを参照することも、DTD やスキーマをまったく含めないまたは参照しないようにすることもできます。次の XML ドキュメントの抜粋では、`address.dtd` という外部 DTD を参照する方法を示しています。

```
<?xml version=1.0?>
<!DOCTYPE address_book SYSTEM "address.dtd">
<address_book>
...
```

XML ドキュメントは、パーサで検証する場合、または複雑なタイプが含まれる場合にのみ DTD またはスキーマを付ける必要があります。XML ドキュメントは、1) 関連する DTD またはスキーマがある場合、および 2) 関連する DTD またはスキーマに定義されている制約に準拠している場合に有効であると見なされます。ただし、XML ドキュメントが整形式でなければならない場合は、ドキュメントに DTD またはスキーマを付ける必要はありません。W3C の XML 1.0 勧告のすべてのルールに従っている場合、ドキュメントは整形式と見なされます。XML 1.0 仕様の詳細な説明については、http://www.w3.org/XML/ を参照してください。

XML を使用する理由

通常、それぞれの業界はその業界にとって意味をもつ固有のデータ交換方式を使用しています。電子商取引の誕生により、企業ではさまざまな業界と関係を結ぶようになっています。このため、こうした企業は、これらの業界で使用されるさまざまな電子通信プロトコルに関する専門知識を取得しなければなりません。

XML は拡張性に優れているため、さまざまな業界間のデータ交換フォーマットを標準化するための非常に効果的なツールとなっています。たとえば、複数の業界間または一企業内の複数の部門間でメッセージ ブローカとワークフロー エンジンがトランザクションを調整しなければならない場合、XML を使用すると、異なるソースからのデータを組み合わせて、すべての当事者が理解できる 1 つのフォーマットを作成できます。

XSL と XSLT

Extensible Stylesheet Language (XSL) は、XML ドキュメントに適用する表示のルールを記述するための W3C の規格です。XSL には、変換言語 (XLST) とフォーマット言語の両方が含まれます。この 2 つの言語は、互いに独立して機能

します。XSLT は XML ベースの言語で、XML ドキュメントを別の XML ドキュメントに、または HTML、PDF など他の文書フォーマットに変換する方法を記述するための W3C の仕様です。

XSLT トランスフォーマは、XML ドキュメントと XSLT ドキュメントを入力として受け付けます。XSLT ドキュメントに定義されるテンプレート ルールには、そのルールの適用先となる XML ツリーを指定するパターンが含まれています。XSLT トランスフォーマは、XML ドキュメントをスキャンしてルールに一致するパターンを見つけ出し、元の XML ドキュメントの該当するセクションにテンプレートを適用します。

DOM と SAX とは

DOM および SAX は、XML データ解析のための 2 つの標準 Java アプリケーション プログラミング インタフェース (API) です。いずれも、WebLogic Server 組み込みパーサによりサポートされます。この 2 つの API は、解析に対して異なるアプローチをとっており、それぞれの API に長所と短所があります。

SAX

SAX は Simple API for XML の略語です。SAX は、イベント ベースの XML 解析用標準インタフェースで、プラットフォームや言語に依存しません。SAX は、パーサが要素の最初または最後などの XML ドキュメントを読み取る際に発生するイベントを定義します。プログラマは、ドキュメント解析時に異なるイベントを処理するハンドラを提供します。

XML ドキュメントを解析するために SAX API を使用するプログラマは、これらのイベントが発生したときに起こることを完全に制御する権限を持ち、解析プロセスをカスタマイズできます。たとえば、プログラマは、ドキュメントが不正であることを示すエラーがパーサで発生したときに、ドキュメントがすべて解析されるまで待つのではなく、すぐに XML ドキュメントの解析を中止できるので、パフォーマンスが向上します。

WebLogic Server の組み込みパーサ (Apache Xerces) は、SAX バージョン 2.0 をサポートしています。SAX バージョン 1.0 を使用して XML ドキュメントを解析するプログラムを作成した場合は、2 つのバージョンの相違点を把握し、それに従ってプログラムを更新する必要があります。2 つのバージョンの相違点については、http://www.saxproject.org/ を参照してください。

DOM

DOM は Document Object Model の略語です。DOM は、プラットフォームや言語に依存しないインタフェースであり、DOM を使用することにより、プログラムとスクリプトは XML ドキュメントのコンテンツ、構造、およびスタイルに動的にアクセスして、それらを更新できます。DOM は、XML ドキュメントをメモリに読み込んでツリー表示します。ツリーの各ノードは、元の XML ドキュメントからの特定のデータの一部を表します。ツリー構造は、データを表すための標準プログラミング メカニズムで、Java を使用したツリーのナビゲートおよび操作を簡単に素早く、かつ効率的に実行できます。しかし、主な欠点として、DOM でツリーを作成するために XML ドキュメント全体を読み取る必要があるため、XML ドキュメントのサイズが大きくなると、アプリケーションのパフォーマンスが低下するというものがあります。

WebLogic Server の組み込みパーサ (Apache Xerces) は、DOM Level 2.0 Core をサポートしています。DOM Level 1.0 を使用して XML ドキュメントを解析するプログラムを作成した場合は、2 つのバージョンの相違点を把握し、それに従ってプログラムを更新する必要があります。相違点の詳細については、http://www.w3.org/DOM/DOMTR を参照してください。

XML Streaming とは

SAX と DOM だけでなく、XML Streaming API を使用しても XML ドキュメントを解析できます。

WebLogic XML Streaming API は、XML ドキュメントの解析と生成を行うための簡単でわかりやすい方法を提供します。SAX API をベースにしていますが、複雑な XML ドキュメントを処理する際の煩雑さの原因になる SAX イベント ハンドラを書く必要はなく、手順の流れに従って XML ドキュメントを処理できます。つまり、Streaming API では、SAX API より解析を綿密に制御できます。

XML Streaming API は、ドキュメント解析の際に WebLogic FastParser を使用します。

WebLogic XML Streaming API の使用方法の詳細については、第 4 章「WebLogic XML Streaming API の使い方」を参照してください。

注意： DOM や SAX と異なり、XML Streaming は、まだ JAXP (Java API for XML Processing) の一部ではありません。

JAXP とは

前の節では、XML データの解析に使用できる API、SAX、DOM について説明しました。JAXP (Java API for XML Processing) は、これらのパーサにアクセスする手段を提供します。JAXP はまた、どのような仕様のパーサまたはトランスフォーマも使用できるプラグイン可能レイヤも定義します。

WebLogic Server には、XML アプリケーションの開発と、WebLogic Server で構築した XML アプリケーションを他の Web アプリケーション サーバに移植するのに必要な作業を容易にするために、JAXP が実装されています。JAXP は、XML アプリケーションの移植性を高めるために Sun Microsystems で開発されました。JAXP では、Java プラットフォームの API の標準セットを介して XML ドキュメントを解析および変換するための基本機能がサポートされています。WebLogic Server 配布キット内の JAXP 1.1 は、組み込みパーサを使用するコンフィグレーションになっています。したがって、WebLogic Server で構築される XML アプリケーションでは、デフォルトで JAXP を使用します。

WebLogic Server 配布キットには、JAXP 1.1 で必要なインタフェースとクラスがあります。JAXP 1.1 には、SAX バージョン 2 および DOM Level 2 に対するサポートがあります。JAXP 用の Javadoc は、WebLogic Server のオンライン リファレンス マニュアルに含まれています。

JAXP パッケージ

JAXP には、以下の 2 つのパッケージがあります。

- `javax.xml.parsers`
- `javax.xml.transform`

`javax.xml.parsers` パッケージには、SAX バージョン 2.0 および DOM Level 2.0 モードで XML データを解析するためのクラスがあります。XML ドキュメントを SAX モードで解析するには、まず `newInstance()` メソッドで新しい `SaxParserFactory` オブジェクトのインスタンスを作成します。このメソッドは、明確に定義された場所のリストを基に、ロードするパーサの特定の実装をルックアップします。次に、`SaxParserFactory` から `SaxParser` インスタンスを取得し、`parse()` メソッドを実行して、解析する XML ドキュメントにそれを渡します。XML ドキュメントを DOM モードで解析する場合も同様ですが、この場合は `DocumentBuilder` クラスと `DocumentBuilderFactory` クラスを使用します。

JAXP による XML ドキュメント解析の詳細については、2-2 ページの「XML ドキュメントの解析」を参照してください。

`javax.xml.transform` パッケージには、XML ドキュメント、DOM ツリー、または SAX イベントなどの XML データを別の形式に変換するクラスがあります。トランスフォーマ クラスも、パーサ クラスと同じように機能します。XML ドキュメントを変換するには、まず `newInstance()` メソッドで新しい `TransformerFactory` オブジェクトのインスタンスを作成します。このメソッドは、明確に定義された場所のリストを基に、ロードする XSLT トランスフォーマの特定の実装をルックアップします。次に、特定の XSLT スタイルシートを基に新しい `Transformer` オブジェクトのインスタンスを作成し、`transform()` メソッドを実行して、変換する XML オブジェクトにそれを渡します。XML オブジェクトは、XML ファイルでも DOM ツリーなどでもかまいません。

JAXP による XML オブジェクト変換の詳細については、2-13 ページの「JAXP による XML データの変換」を参照してください。

XML と XSLT の一般的な使い方

XML と XSLT をどのように使用するかは、ビジネス上のニーズによって異なります。

XML と XSLT を使用したコンテンツと表示の分離

XML と XSLT は、複数のクライアント タイプをサポートするアプリケーションでよく使用されます。たとえば、ブラウザ ベースのクライアントと Wireless Application Protocol (WAP) クライアントの両方をサポートする Web ベース アプリケーションを使用しているとします。これらのクライアントは、それぞれ HTML と Wireless Markup Language (WML) という異なるマーク付け言語を理解しますが、アプリケーションでは両方に適したコンテンツを提供しなければなりません。

これを実現するには、クライアントに送信するデータを表す XML ドキュメントを最初に作成するようにアプリケーションを記述します。次に、アプリケーションは、クライアントのブラウザ タイプに応じて、データを表す XML ドキュメントを HTML または WML に変換します。アプリケーションでは、HTTP リクエストの `User-Agent` リクエスト ヘッダを調べることによって、クライアントのブラウザ タイプを識別できます。クライアントのブラウザ タイプを認識すると、適切な XSLT スタイル シートを使用して、適切なマーク付け言語にドキュメントを変換します。このヘッダ情報へのアクセス方法の例については、WebLogic Server 配布キットの `examples/servlets` ディレクトリに収められている SnoopServlet サンプルを参照してください。

このように、クライアント タイプごとに異なるマーク付け言語を使用して同じ XML ドキュメントを表示する方法を用いると、複数のクライアント タイプのサポートに必要な努力を、適切な XSLT スタイル シートの開発に集中させることができます。さらに、必要な場合は、アプリケーションを他のクライアント タイプに簡単に適合させることができます。

XSLT の詳細については、8-2 ページの「その他の XML 仕様と情報」を参照してください。

企業間通信用メッセージ フォーマットとしての XML

企業間 (B2B) 環境で、企業 A と企業 B は両者間の電子商取引に関する情報の交換を望んでいます。企業 A は大手の電子商取引サイトです。企業 B は、企業 A の製品を最適な顧客集団に販売する小規模な子会社です。企業 B が企業 A に顧客情報を送ると、企業 B は企業 A から 2 通りの方法で対価を受け取ることができます。企業 B は、金銭、および企業 B が提供した顧客と同じ購買層の他の顧客に関する情報を企業 A から受け取ります。情報を交換するためには、企業 A と企業 B はマシンが理解でき、どちらの企業のシステムでも簡単に処理できるデータ フォーマットについて合意する必要があります。XML は、このシナリオで使用する論理データ フォーマットですが、このフォーマットの選択はほんの第一歩に過ぎません。次に両企業は、交換する XML メッセージのフォーマットについて合意しなければなりません。企業 A はその子会社と 1 対多の関係にあるため、やり取りする XML メッセージのフォーマットは企業 A 側で定義する必要があります。

XML メッセージ、つまり XML ドキュメントのフォーマットを定義するために、企業 A は 2 種類の文書型定義 (DTD) を作成します。1 つは企業 A が提供する顧客情報を記述するもので、もう 1 つは企業 A が受け取る新しい子会社の情報を記述するものです。企業 B も、2 種類の DTD を作成します。1 つは企業 A から受け取る XML ドキュメントを処理するためのもの、もう 1 つは企業 A 側で処理できるフォーマットで XML ドキュメントを作成するためのものです。

WebLogic Server XML の機能

WebLogic Server は、WebLogic Server と WebLogic Server に基づく XML アプリケーションに適用可能な XML テクノロジを統合します。WebLogic Server XML サブシステムを使用すると、顧客は、標準パーサ、WebLogic FastParser、XSLT トランスフォーマ、DTD、および XML スキーマを使用して XML ファイルを処理および変換できます。

WebLogic Server XML サブシステムには、以下の機能があります。

- XML ドキュメント パーサ
- XML ドキュメント トランスフォーマ
- WebLogic XML Streaming API
- JAXP プラグイン可能レイヤの実装
- WebLogic サーブレット属性
- WebLogic XSLT JSP タグ ライブラリ
- パーサおよびトランスフォーマのコンフィグレーション用 XML レジストリ
- 外部エンティティ解決をコンフィグレーションするための XML レジストリ
- XML ドキュメントの解析と変換のためのサンプル コード

XML ドキュメント パーサ

WebLogic Server には、以下の 2 つのパーサがあります。

表 1-1 WebLogic Server に含まれるパーサ

パーサ	説明
組み込み	Apache Xerces パーサ バージョン 1.4.4 に基づく検証パーサ。組み込みパーサは、Simple API For XML (SAX) モードまたは JAXP API を使用する Document Object Model (DOM) モードのいずれかで使用できる。

表 1-1 WebLogic Server に含まれるパーサ

パーサ	説明
WebLogic FastParser	WebLogic Web サービスに関連付けられた SOAP および WSDL ファイルなど、中小規模のドキュメントを処理するために特別に設計された高性能の非検証 XML パーサ。FastParser は、SAX スタイルの解析のみをサポートしている。アプリケーションが中小規模 (最大 10,000 要素) の XML ドキュメントを処理する場合、FastParser を使用するように WebLogic Server をコンフィグレーションする。 WebLogic FastParser の詳細については、2-10 ページの「WebLogic FastParser の使用」を参照。

Administration Console による XML レジストリのコンフィグレーションでは、他のあらゆる XML パーサを使用できます。単一の WebLogic Server のインスタンスをコンフィグレーションすることによって、あるパーサを特定のアプリケーション用に使用し、別のパーサを異なるアプリケーション用に使用できます。

XML ドキュメント トランスフォーマ

WebLogic Server には、Apache Xalan XSLT トランスフォーマ バージョン 2.2 をベースとした組み込み XSLT トランスフォーマがあります。XML アプリケーションでは、この組み込み XSLT トランスフォーマまたは他の XSLT トランスフォーマを使用して、XML ドキュメントを他の XML ドキュメント、HTML などに変換できます。XML ドキュメント変換の詳細については、2-13 ページの「JAXP による XML データの変換」を参照してください。

WebLogic XML Streaming API

WebLogic XML Streaming API は、XML ドキュメントの解析と生成を行うための簡単で直感的な方法を提供します。SAX API をベースにしていますが、複雑な XML ドキュメントを処理する際の煩雑さの原因になる SAX イベント ハンドラを書く必要はなく、もっと手順の流れに従って XML ドキュメントを処理できます。つまり、Streaming API では、SAX API より解析を綿密に制御できます。

WebLogic XML Streaming API の使用方法の詳細については、第 4 章「WebLogic XML Streaming API の使い方」を参照してください。

JAXP プラグイン可能レイヤの実装

Java API for XML Parsing (JAXP) 1.1 は、Java 標準で、パーサに依存しない XML 用 API です。JAXP の詳細については、1-8 ページの「JAXP とは」を参照してください。

注意： WebLogic Server は、Administration Console からアクセスされる XML レジストリを使用して、パーサとトランスフォーマをプラグインします。この点が、システム プロパティを使用してパーサとトランスフォーマをプラグインするように指定された JAXP 1.1 仕様との違いです。

WebLogic サーブレット属性

WebLogic Server は、以下の特殊なサーブレット属性をサポートします。

- `org.xml.sax.HandlerBase`
- `org.xml.sax.helpers.DefaultHandler`
- `org.w3c.dom.Document`

前述の属性を持つ `ServletRequest` オブジェクトで、`setAttribute` メソッド (SAX 解析の場合) および `getAttribute` メソッド (DOM 解析の場合) を呼び出すと、任意の XML ドキュメントを解析できます。

以下のコード セクションでは、これらのメソッドの使い方の例を示します。

```
request.setAttribute("org.xml.sax.helpers.DefaultHandler", new DefHandler());

org.w3c.dom.Document = (Document)request.getAttribute("org.w3c.dom.Document");
```

注意： `setAttribute` メソッドと `getAttribute` メソッドは便利な機能として用意されているだけであり、サーブレットから XML を解析する際には必須というわけではありません。

WebLogic XSLT JSP タグ ライブラリ

JSP タグ ライブラリは、WebLogic Server で動作する JavaServer Pages (JSP) 内から組み込み XSLT トランスフォーマにアクセスするための単純なタグを提供します。現時点では、このタグは組み込み XSLT トランスフォーマだけをサポートしています。このタグを使用して、XML ドキュメントを JSP 内から異なるトランスフォーマを使って変換することはできません。

JSP タブ ライブラリは、`xmlx-tags.jar` に収められています。このファイルは、WebLogic Server 配布キットのインストール時に自動的にインストールされます。

注意： JSP タグ ライブラリは便利な機能として用意されているだけであり、JSP 内から XSLT トランスフォーマへのアクセスに必要というわけではありません。

パーサおよびトランスフォーマのコンフィグレーション用 XML レジストリ

XML レジストリは、管理タスクとコンフィグレーション タスクを XML アプリケーションから切り離すことによって、これらのタスクを簡素化します。WebLogic Server のインスタンスに対するパーサおよびトランスフォーマのコンフィグレーションには、Administration Console (WebLogic Server 管理用のグラフィカル ユーザ インタフェース (GUI)) を使用します。

注意： 各 WebLogic Server ドメインにはレジストリをいくつでも組み込むことができ、ドメイン内の各 WebLogic Server インスタンスにはゼロまたは 1 つのレジストリを割り当てることができます。

XML レジストリを使用すると

- 構築時ではなくデプロイメント時にパーサまたはトランスフォーマを指定できます。
- アプリケーションにパーサ依存コードまたはトランスフォーマ依存コードを組み込む必要がありません。

- 単一のサーバで複数のパーサおよびトランスフォーマをより便利にサポートできます。

XML レジストリを使用すると、以下のタスクを行うことができます。

- WebLogic Server の本バージョンに付属の組み込みパーサの代わりに別の XML パーサをコンフィグレーションします。
- WebLogic Server の本バージョンに付属の組み込みトランスフォーマの代わりに別の XSLT トランスフォーマをコンフィグレーションします。
- 特定のアプリケーションを処理するために XML パーサをコンフィグレーションします。

上記のすべての機能は、アプリケーションが WebLogic Server の本バージョンに付属の標準 Java API for XML Processing (JAXP) を使用する場合に利用できます。これらの機能は、サーバサイドだけで使用できます。

外部エンティティ解決をコンフィグレーションするための XML レジストリ

WebLogic XML では、XML レジストリを使用して外部エンティティを解決できます。外部エンティティは、XML ドキュメント内に記述されていないものの、XML ドキュメント内で参照されるさまざまなテキストです。実際のテキストは、同じコンピュータの他のファイル、Web 上など、どこにあってもかまいません。外部エンティティの例が、XML ドキュメントの検証に使用する DTD ファイルです。この機能を使用するには、Administration Console を開き、XML レジストリで、外部エンティティに関連付けるパブリック `ID` またはシステム `ID` を入力します。

外部エンティティをローカルに格納することに加えて、URL などの HTTP インタフェースをサポートする外部レポジトリから外部エンティティを取得およびキャッシュするように WebLogic Server をコンフィグレーションできます。WebLogic Server のコンフィグレーションで、メモリやディスクにある外部エンティティをキャッシュし、有効期限が過ぎたと判断するまでエンティティをキャッシュ内にどの程度の期間とどめるかを指定できます。

XML レジストリを使用した外部エンティティの解決の詳細については、7-12 ページの「外部エンティティのコンフィグレーション タスク」を参照してください。

XML ドキュメントの解析と変換のためのサンプル コード

WebLogic Server には、XML ドキュメントの解析および変換のサンプルがあります。

このサンプルは、*WL_HOME*\samples\server\src\examples\xml ディレクトリにあります。*WL_HOME* は最上位 WebLogic Platform ディレクトリです。

このサンプルを作成して実行する方法の詳細については、ブラウザで Web ページ *WL_HOME*\samples\server\src\examples\xml\package-summary.html を呼び出してください。

XML ファイルの編集

XML ファイルを編集するには、完全な Java ベースの XML スタンドアロン エディタである BEA XML エディタを使用します。この XML エディタは、XML ファイルを作成および編集するためのシンプルでユーザフレンドリなツールです。このツールでは、XML ファイルの中身を、階層的な XML ツリー構造と実際の XML コードの両方で表示します。したがって、XML ドキュメントの編集方法を選択できます。

- 階層ツリー表示では、階層 XML ツリー構造の各ポイントでいくつかの指定可能な機能を使用する形で、構造化された制約のある編集が可能です。指定可能な機能は、構文的に決定されており、指定されているものがある場合は XML ドキュメントの DTD またはスキーマに従っています。
- XML コード表示では、データを自由に編集できます。

BEA XML エディタは、指定した DTD または XML スキーマを基に XML コードを検証します。

使用方法の詳細については、BEA XML エディタのオンライン ヘルプを参照してください。

BEA XML エディタは、dev2dev Online からダウンロードできます。

XML について学習するには

XML について学習するには、以下のオンラインコースおよびチュートリアルを参照してください。第 8 章「XML リファレンス」にリンクと詳細情報があります。

- A Technical Introduction to XML
- XML Authoring Tutorial
- Working with XML and Java
- Tutorials for using the Java 2 platform and XML technology
- XML, Java, and the Future of the Web
- 『XML Bible』第 14 章「XSL Transformations」
- 『XSL Tutorial』、Miloslav Nic 著
- SAX 2.0: The Simple API for XML
- Document Object Model (DOM)

2 WebLogic Server による XML アプリケーションの開発

以降の節では、Java プログラミング言語と WebLogic Server を使用して XML アプリケーションを開発する方法について説明します。ここでは、Java サーブレットと JavaServer Pages (JSP) を使用して Java アプリケーションを記述する方法について理解していることを前提にしています。サーブレットと JSP アプリケーションの記述方法については、『WebLogic HTTP サーブレット プログラマーズ ガイド』と『WebLogic JSP プログラマーズ ガイド』を参照してください。

XML アプリケーションの開発：主な手順

WebLogic Server XML サブシステムを使用して XML アプリケーションを開発する場合、通常は以下のプログラミング タスクの一部または全部を実行します。

1. XML ドキュメントを解析します。

   XML ドキュメントは、いくつかのソースから構成されています。たとえば、クライアントから XML ドキュメントを受信するサーブレットを開発し、サーブレットまたはその他の EJB などから XML ドキュメントを受信する EJB を記述できます。各インスタンスで、データを操作できるように XML ドキュメントを解析しておくこともできます。

   このタスクの詳細については、2-2 ページの「XML ドキュメントの解析」を参照してください。

2. 新しい XML ドキュメントを生成します。

   サーブレットまたは EJB が XML ドキュメントを受信および解析し、何らかの形でデータを操作したら、サーブレットまたは EJB は、新しい XML ドキュメントを生成してクライアントに返すか、他の EJB に渡さなければならないことがあります。

   このタスクの詳細については、2-10 ページの「新しい XML ドキュメントの生成」を参照してください。

3. XML データを他の形式に変換します。

   XML ドキュメントを解析するか、新しい XML ドキュメントを生成したら、サーブレットまたは EJB は、その XML を HTML、WML、またはプレーンテキストなどの形式に変換しなければならないことがあります。

   このタスクの詳細については、2-13 ページの「JAXP による XML データの変換」を参照してください。

XML ドキュメントの解析

この節では、JAXP により DOM モードと SAX モードで XML ドキュメントを解析する方法、およびサーブレットから XML ドキュメントを解析する方法について説明します。

注意： WebLogic XML ストリーミング API により XML ドキュメントを解析する方法の詳細については、第 4 章「WebLogic XML Streaming API の使い方」を参照してください。

前述のように、Administration Console の XML レジストリを使用して以下の項目をコンフィグレーションします。

- doctype ごとのパーサ。指定した doctype の組み込みパーサの代わりに使用されます。
- 外部エンティティの解決。XML ドキュメントの解析中に外部ファイルを見つけるように要求された場合に XML パーサが実行する処理です。

Administration Console でこれらのタスクを実行する方法については、第 7 章「WebLogic Server XML の管理」を参照してください。

SAX モードによる XML ドキュメント解析の詳細なサンプルについては、*WL_HOME*\samples\server\src\examples\xml\sax ディレクトリを参照してください。*WL_HOME* は、最上位 WebLogic Platform ディレクトリです。

SAX モードで JAXP を使用した XML ドキュメントの解析

次のコード例は、SAX パーサ ファクトリをコンフィグレーションして検証パーサを作成する方法を示したものです。また、MyHandler クラスをパーサに登録する方法も示しています。MyHandler クラスは、SAX 解析イベントまたはエラーのカスタム動作を提供するために DefaultHandler クラスのあらゆるメソッドをオーバライドできます。

```
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

...
MyHandler handler = new MyHandler();
// MyHandler は org.xml.sax.helpers.DefaultHandler を拡張

  //SAXParserFactory のインスタンスを取得
  SAXParserFactory spf = SAXParserFactory.newInstance();
  // 検証パーサを指定
  spf.setValidating(true); // DTD をロードする必要がある
  //ファクトリから SAX パーサのインスタンスを取得
  SAXParser sp = spf.newSAXParser();
  // ドキュメントを解析
  sp.parse("http://server/file.xml", handler);
...
```

注意： 組み込みパーサ以外のパーサを使用する場合は、WebLogic Server Administration Console を使用して XML レジストリでパーサを指定します。指定しなかった場合、SaxParserFactory.newInstance メソッドは組み込みパーサを返します。組み込みパーサ以外のパーサを使用する WebLogic Server のコンフィグレーション手順については、7-5 ページの「組み込み以外のパーサまたはトランスフォーマのコンフィグレーション」を参照してください。

SAX モードによる XML ドキュメント解析の詳細なサンプルについては、*WL_HOME*\samples\server\src\examples\xml\sax ディレクトリを参照してください。*WL_HOME* は、最上位 WebLogic Platform ディレクトリです。

DOM モードで JAXP を使用した XML ドキュメントの解析

次のコード例では、XML ドキュメントを解析し、DocumentBuilder オブジェクトから org.w3c.dom.Document ツリーを作成する方法を示します。

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;

...
//DocumentBuilderFactory のインスタンスを取得
DocumentBuilderFactory dbf =
                  DocumentBuilderFactory.newInstance();
// 検証パーサを指定
dbf.setValidating(true); // DTD をロードする必要がある
// ファクトリから DocumentBuilder のインスタンスを取得
DocumentBuilder db = dbf.newDocumentBuilder();
// ドキュメントを解析
Document doc = db.parse(inputFile);
...
```

注意： 組み込みパーサ以外のパーサを使用する場合は、WebLogic Server Administration Console でパーサを指定します。指定しなかった場合、DocumentBuilderFactory.newInstance メソッドは組み込みパーサを返します。組み込みパーサ以外のパーサを使用する WebLogic Server のコンフィグレーション手順については、7-5 ページの「組み込み以外のパーサまたはトランスフォーマのコンフィグレーション」を参照してください。

DOM モードによる XML ドキュメント解析の詳細なサンプルについては、*WL_HOME*\samples\server\src\examples\xml\dom ディレクトリを参照してください。*WL_HOME* は、最上位 WebLogic Platform ディレクトリです。

サーブレットでの XML ドキュメントの解析

Java サーブレット仕様バージョン 2.2 で、setAttribute メソッドと getAttribute メソッドに対するサポートが追加されました。属性は、リクエストに関連付けられたオブジェクトです。リクエスト オブジェクトは、クライアント リクエストからの全情報をカプセル化します。HTTP プロトコルでは、この情報は、リクエストの HTTP ヘッダとメッセージ本文を基にクライアントからサーバに転送されます。

WebLogic Server では、setAttribute メソッドと getAttribute メソッドを使用して XML ドキュメントを解析できます。SAX モード解析には setAttribute メソッドを使用し、DOM モード解析には getAttribute メソッドを使用します。

org.xml.sax.DefaultHandler 属性を使用したドキュメントの解析

次のサンプル コードでは、setAttribute メソッドの使い方を示します。

```
import weblogic.servlet.XMLProcessingException;
import org.xml.sax.helpers.DefaultHandler;
...
public void doPost(HttpServletRequest request,
                              HttpServletResponse response)
   throws ServletException, IOException  {
   try {
     request.setAttribute("org.xml.sax.helpers.DefaultHandler",
                          new DefaultHandler());
    } catch(XMLProcessingException xpe) {
      System.out.println("Error in processing XML");
      xpe.printStackTrace();
      return;
    }
...
```

また、現在は非推奨となっているものの、org.xml.sax.HandlerBase 属性を使用して XML ドキュメントを解析することもできます。

```
request.setAttribute("org.xml.sax.HandlerBase",
                     new HandlerBase());
```

注意： このサンプル コードは、SAX と `setAttribute` メソッドでドキュメントを解析する単純な方法を示したものです。ドキュメントを解析するこのメソッドは、他のサーブレット ベンダではサポートされていない WebLogic Server の便利な機能です。したがって、アプリケーションを他のサーブレット プラットフォームで実行する場合は、この機能を使用しないでください。

org.w3c.dom.Document 属性を使用したドキュメントの解析

次のサンプル コードでは、`getAttribute` メソッドの使い方を示します。

```
import org.w3c.dom.Document;
import weblogic.servlet.XMLProcessingException;

...

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

try {
   Document doc = request.getAttribute("org.w3c.dom.Document");
 } catch(XMLProcessingException xpe) {
   System.out.println("Error in processing XML");
   xpe.printStackTrace();
   return;
 }
...
```

注意： このサンプル コードは、DOM と `getAttribute` メソッドでドキュメントを解析する単純な方法を示したものです。ドキュメントを解析するこのメソッドは、他のサーブレット ベンダではサポートされていない WebLogic Server の便利な機能です。したがって、アプリケーションを他のサーブレット プラットフォームで実行する場合は、この機能を使用しないでください。

非検証パーサの検証

前述のとおり、整形式ドキュメントは、W3C の XML 1.0 勧告のルールに従っている、構文的に正しいドキュメントのことです。有効なドキュメントは、DTD またはスキーマで指定した制約に従っているドキュメントです。

非検証パーサは、ドキュメントが整形式かどうかを検証しますが、有効かどうかは検証しません。2-10 ページの「WebLogic FastParser の使用」で説明している WebLogic FastParser は、非検証パーサです。

(検証パーサの使用を前提に) ドキュメント解析時の検証を有効にするには、以下を実行する必要があります。

- 以下の例で示すように、`SAXParserFactory.setValidating()` メソッドを true に設定します。

  ```
  SAXParserFactory factory = SAXParserFactory.newInstance();
  factory.setValidating(true);
  ```

- 解析する XML ドキュメントで、(インラインまたは参照で) DTD またはスキーマが定義されているようにします。

XML ドキュメント解析時のエンティティ解決の処理

この節では、XML パーサによる外部エンティティの識別および解決方法、および XML アプリケーションによる外部エンティティ解決のパフォーマンスを向上させる WebLogic Server の機能など、外部エンティティに関する一般的な情報について説明します。

XML ドキュメントを解析しながら外部エンティティを解決する場合の詳細なサンプルについては、*`WL_HOME`*`\samples\server\src\examples\xml\entityresolution` ディレクトリを参照してください。*`WL_HOME`* は、最上位 WebLogic Platform ディレクトリです。

外部エンティティに関する一般的な情報

外部エンティティは、XML ドキュメント内に記述されていないものの、XML ドキュメント内で参照されるさまざまなテキストです。実際のテキストは、同じコンピュータの他のファイル、Web 上など、どこにあってもかまいません。
パーサは、ドキュメントの解析時に外部エンティティ参照に出会うと、参照されたテキストをフェッチし、テキストを XML ドキュメント内に配置してから、解

析を続行します。外部エンティティの例が DTD です。XML ドキュメント内に DTD のテキストがすべて記述されているのではなく、別のファイルに格納されている DTD への参照が XML ドキュメント内にあります。

外部エンティティの識別方法には、システム ID とパブリック ID の 2 つがあります。システム ID は、URI で指定された位置を基に外部エンティティを参照します。パブリック ID は、外部で宣言されている名前を使用して情報を参照します。

次の例では、パブリック ID を使用して、J2EE アプリケーション アーカイブ (*.ear) を示す `application.xml` ファイルの DTD を参照する方法を示します。

```
<!DOCTYPE application PUBLIC "-//Sun Microsystems,
Inc.//DTD J2EE Application 1.2//EN">
```

次の例は、システム ID のみによる外部 DTD の参照を示します。

```
<!DOCTYPE application SYSTEM
"http://java.sun.com/j2ee/dtds/application_1_2.dtd">
```

次の例は、パブリック ID およびシステム ID の両方を使用する参照です。SYSTEM キーワードが省略されています。

```
<!DOCTYPE application
PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application 1.2//EN"
"http://java.sun.com/j2ee/dtds/application_1_2.dtd">
```

WebLogic Server のエンティティ解決機能の使用

以下の WebLogic Server 機能を使用すると、XML アプリケーションの外部エンティティ解決のパフォーマンスを向上させることができます。

- 外部エンティティのコピーを、WebLogic 管理サーバのホストとなるコンピュータに永続的に格納します。
- WebLogic Server が、URL などの HTTP インタフェースをサポートする外部レポジトリにある外部エンティティを自動的に取得およびキャッシュするように指定します。エンティティをキャッシュ先をメモリにするかディスクにするかを指定し、キャッシュされているエントリが自動的に更新されるキャッシュの有効期限を指定できます。

  この取り出し / キャッシュ機能を使用すると、実際に外部エンティティをローカル コンピュータにコピーする必要はなくなります。XML アプリケーションは、ドキュメントを解析するたびにではなく、指定した期間でのみ外

部エンティティを参照するので、アプリケーションのパフォーマンスを大幅に向上させることができ、必要に応じて外部エンティティを最新の状態で参照できます。

XML レジストリで、外部エンティティの場所 (ローカルまたは URL)、およびどのキャッシュ オプションが Web 上のエンティティ用かを識別するエンティティ解決エントリを作成します。システム ID またはパブリック ID を使用して外部エンティティ エントリを識別します。次に、XML ドキュメントで、この外部エンティティを参照すると、WebLogic Server は、ドキュメントの解析時にローカル コピーまたはキャッシュされたコピー (コンフィグレーションによる) を取り出します。

XML レジストリでの外部エンティティ レジストリ作成の詳細については、7-12 ページの「外部エンティティのコンフィグレーション タスク」を参照してください。

組み込みパーサ以外のパーサの使用

JAXP で XML ドキュメントを解析する場合、WebLogic Server XML レジストリ (Administration Console でコンフィグレーション) には以下の選択肢があります。

- 組み込みパーサをサーバ全体のパーサとして使用します。
- WebLogic FastParser をサーバ全体のパーサとしてコンフィグレーションします。
- 選択した別のパーサ (別のバージョンの Apache Xerces パーサなど) をサーバ全体のパーサとしてコンフィグレーションします。
- システム ID、パブリック ID、またはルート タグに基づいた特定の DTD に対応するパーサをコンフィグレーションします。

XML レジストリで解析オプションをコンフィグレーションする手順については、7-4 ページの「XML パーサおよびトランスフォーマのコンフィグレーション タスク」を参照してください。

WebLogic FastParser の使用

WebLogic Server には、中小規模の XML ドキュメント (最大 10,000 個の要素) の検証用に特別に設計された高性能の非検証 XML パーサ (WebLogic FastParser) があります。大きなドキュメントになると、このパーサのパフォーマンスは、Apache Xerces などの標準パーサと変わらなくなります。

注意 : WebLogic FastParser は、SAX スタイルの解析のみをサポートしています。

WebLogic FastParser を WebLogic Server 全体のパーサとして使用するように指定することもできます。または、7-4 ページの「XML パーサおよびトランスフォーマのコンフィグレーション タスク」で説明されているように、XML レジストリを使用することで特定の DOCTYPE で使用するように指定することもできます。[SAX パーサ ファクトリ] フィールドを `weblogic.xml.babel.jaxp.SAXParserFactoryImpl` に設定します。

新しい XML ドキュメントの生成

この節では、DOM ドキュメント ツリーから JSP を使用して XML ドキュメントを生成する方法について説明します。

注意 : XML ストリーミング API により XML ドキュメントを生成する方法の詳細については、第 4 章「WebLogic XML Streaming API の使い方」を参照してください。

DOM ドキュメント ツリーからの XML の生成

この節では、DOM ドキュメント ツリーから XML ドキュメントを作成する以下の 2 つの方法について説明します。

- Apache `serialize` クラスを使用する場合
- JAXP `Transformer` クラスを使用する場合

Apache serialize クラスを使用する場合

DOM ドキュメント ツリーから XML ドキュメントを生成するには、`weblogic.apache.xml.serialize` クラスを使用すると、DOM ドキュメント ツリーを XML テキストに変換できます。このクラスの詳細については、`weblogic.apache.xml.serialize` の Javadoc を参照してください。

次のコード例では、このクラスの使い方を示します。

注意： 次の例では、JAXP を使用しない代わりに、Apache 固有の API を直接使用します。

```
package test;

import java.io.OutputStreamWriter;
import java.util.Date;
import java.text.DateFormat;

import org.w3c.dom.Document;
import org.w3c.dom.Element;

import weblogic.apache.xerces.dom.DocumentImpl;
import weblogic.apache.xml.serialize.DOMSerializer;
import weblogic.apache.xml.serialize.XMLSerializer;

public class WriteXML {

  public static void main(String[] args) throws Exception {

    // DOM ツリーを作成
    Document doc= new DocumentImpl();
    Element message = doc.createElement("message");
    doc.appendChild(message);
    Element text = doc.createElement("text");
    text.appendChild(doc.createTextNode("Hello world."));
    message.appendChild(text);
    Element timestamp = doc.createElement("timestamp");
    timestamp.appendChild(
      doc.createTextNode(
                DateFormat.getDateInstance().format(new Date()))
    );
    message.appendChild(timestamp);

    // DOM を XML テキストにシリアライズして stdout に出力
    DOMSerializer xmlSer =
      new XMLSerializer(new OutputStreamWriter(System.out),null);
    xmlSer.serialize(doc);
  }
}
```

JAXP Transformer クラスを使用する場合

コード例の次のセグメントで示すように、javax.xml.transform.Transformer クラスを使用して、DOM オブジェクトを XML ストリームにシリアライズできます。

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;

import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import java.io.*;

...

TransformerFactory trans_factory =
TransformerFactory.newInstance();
Transformer xml_out = trans_factory.newTransformer();
Properties props = new Properties();
props.put("method", "xml");
xml_out.setOutputProperties(props);
xml_out.transform(new DOMSource(doc), new
StreamResult(System.out));
```

例では、Transformer.transform() メソッドは、DOM オブジェクトを XML ストリームに変換します。transform() メソッドは、doc 変数に格納されている DOM ツリーから作成された javax.xml.transform.dom.DOMSource オブジェクトを入力としてとり、それを javax.xml.transform.stream.StreamResult オブジェクトに変換して、結果の XML ドキュメントを標準出力に書き出します。

JSP での XML ドキュメントの生成

通常、JSP は HTML の生成に使用しますが、JSP を使用して XML ドキュメントを生成することもできます。

JSP による XML の生成には、以下のように JSP ページのコンテンツ タイプを設定する必要があります。

```
<%@ page contentType="text/xml"%>
    ... XML document
```

次のコードでは、JSP による XML ドキュメントの生成例を示します。

```
<?xml version="1.0">

<%@ page contentType="text/xml"
import="java.text.DateFormat,java.util.Date" %>

<message>
  <text>
    Hello World.
  </text>
  <timestamp>
<%
out.print(DateFormat.getDateInstance().format(new Date()));
%>
  </timestamp>
</message>
```

JSP による XML の生成の詳細については、`http://java.sun.com/products/jsp/html/JSPXML.html` を参照してください。

XML ドキュメントの変換

変換とは、XML ドキュメント (変換元) を他の形式 (変換結果) に変えることです。通常は、別の XML ドキュメント、HTML、無線用マーク付け言語 (Wireless Markup Language : WML) に変換されます。この節では、JAXP を使用して、および JSP 内から JSP タグを使用して XML ドキュメントを変換する方法について説明します。

JAXP による XML データの変換

バージョン 1.1 の JAXP では、プラグイン可能の変換が提供されています。つまり、JAXP 準拠の任意のトランスフォーマ エンジンを使用できます。

JAXP は、XML データをさまざまな形式に変換するために以下のインタフェースを提供します。

- `javax.xml.transform`: このパッケージには、ドキュメントを変換するための汎用 API が入っています。SAX か DOM かに関係なく、変換元および変換結果の形式を推測し、できる限り少ない候補に絞り込みます。
- `javax.xml.transform.stream`: このパッケージは、ストリーム固有および URI 固有の変換 API を実装します。特に、`InputStreams` と URL を変換元、`OutputStreams` と URL を変換結果として指定できる `StreamSource` クラスおよび `StreamResult` クラスを定義します。
- `javax.xml.transform.dom`: このパッケージは、DOM 固有の変換 API を実装します。特に、DOM ツリーを変換元か変換結果、またはその両方として指定できる `DOMSource` クラスおよび `DOMResult` クラスを定義します。
- `javax.xml.transform.sax`: このパッケージは、SAX 固有の変換 API を実装します。特に、`org.xml.sax.ContentHandler` イベントを変換元か変換結果、またはその両方として指定できる `SAXSource` クラスおよび `SAXResult` クラスを定義します。

変換には、入力と出力の多くの組み合わせがあります。

XML ドキュメント変換の詳細な例については、*`WL_HOME`*`\samples\server\src\examples\xml\xslt` ディレクトリを参照してください。*`WL_HOME`* は、最上位 WebLogic Platform ディレクトリです。

JAXP による XML ドキュメント変換の例

抜粋した次のコード例では、JAXP を使用して `myXMLdoc.xml` を `mystylesheet.xsl` スタイルシートを使用する別の XML ドキュメントに変換する方法を示します。

```
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;

Transformer trans;
TransformerFactory factory = TransformerFactory.newInstance();
String stylesheet = "file://stylesheets/mystylesheet.xsl";
String xml_doc = "file://xml_docs/myXMLdoc.xml";
```

```
trans = factory.newTransformer(new StreamSource(stylesheet));
trans.transform(new StreamSource(xml_doc),
                new StreamResult(System.out));
```

DOM ドキュメントを XML ストリームに変換する方法の例については、2-12 ページの「JAXP Transformer クラスを使用する場合」を参照してください。

Xalan API 使用から JAXP 1.1 API への XML コードの変換

アプリケーションに Xalan 固有のコードが含まれている場合、それを JAXP を使用するように変更することをお勧めします。

この節では、Xalan API 使用から JAXP 使用に変換するために XML アプリケーションで必要な変更を簡単に説明します。この節では、よく似た変換タスクを実行する 2 つのコード セグメントを比較します。一方のコード セグメントでは Xalan API を直接使用し、もう一方では JAXP を使用します。

次の Java コード セグメントでは JAXP を使用しています。

```
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;

...

Transformer trans;
TransformerFactory factory = TransformerFactory.newInstance();
String stylesheet = "file://stylesheets/mystylesheet.xsl";
String xml_doc = "file://xml_docs/myXMLdoc.xml";

trans = factory.newTransformer(new StreamSource(stylesheet));
trans.transform(new StreamSource(xml_doc),
                new StreamResult(System.out));
```

次の Java コード セグメントでは Xalan API を直接使用しています。

```
 /*
 * このサンプル コードは、Apache Software Foundation により
 * 提供されているコードからの引用であり、
 * 多くの個人から有志の協力を得て完成した。
 * 元のソフトウェア著作権は以下のとおり
 * copyright (c) 1999, Lotus Development Corporation.,
 * http://www.lotus.com。Apache Software Foundation に関する情報について
は
```

```
 * <http://www.apache.org/> を参照
 */

import org.apache.xalan.xslt.XSLTProcessorFactory;
import org.apache.xalan.xslt.XSLTInputSource;
import org.apache.xalan.xslt.XSLTResultTarget;
import org.apache.xalan.xslt.XSLTProcessor;

...

XSLTProcessor processor = XSLTProcessorFactory.getProcessor();

String stylesheet = "file://stylesheets/mystylesheet.xsl";
String xml_doc = "file://xml_docs/myXMLdoc.xml";

processor.process(new XSLTInputSource(xml_doc),
                  new XSLTInputSource(stylesheet),
                  new XSLTResultTarget(System.out));
```

次の表では、前の例で XML ドキュメントの変換に使用した Xalan と JAXP のインタフェースおよびメソッドの名前を示します。この表を基に、既存の Xalan アプリケーションを完全な JAXP アプリケーションに変換してください。

注意： この表は、Xalan と JAXP の間のマッピング全体ではなく、前の例で使用したメイン クラスおよびメソッドのみを示します。各 API の詳細については、Apache および Sun の Web サイト (http://www.apache.org および http://java.sun.com/xml/index.html) を参照してください。

表 2-1 Xalan と JAXP のクラスとインタフェースの対応

クラスまたはインタフェースの説明	Xalan 1.X	JAXP 1.1
XML ドキュメントの変換に使用するメイン クラス	`XSLTProcessor`	`Transformer`
トランスフォーマ オブジェクトの作成に使用するファクトリ クラス	`XSLTProcessorFactory`	`TransformerFactory`
ファクトリの新しいインスタンスの作成に使用するメソッド	なし	`TransformerFactory.newInstance()`
新しいトランスフォーマ オブジェクトの作成に使用するメソッド	`XSLTProcessorFactory.getProcessor()`	`TransformerFactory.newTransformer()`

表 2-1 Xalan と JAXP のクラスとインタフェースの対応

クラスまたはインタフェースの説明	Xalan 1.X	JAXP 1.1
XML ドキュメントまたは XSL スタイルシートなどの変換元を保持するクラス	`XSLTInputSource`	`StreamSource`
変換結果を保持するクラス	`XSLTResultTarget`	`StreamResult`
変換を実行するメソッド	`XSLTProcessor.process()`	`Transformer.transform()`

JSP タグによる XML ドキュメントの変換

WebLogic Server では、JSP 内から XSLT トランスフォーマに簡単にアクセスするための小規模な JSP タグ ライブラリが提供されます。このタグを使用すると XML ドキュメントを HTML、WML などに変換できますが、必須ではありません。

JSP タグ ライブラリは、メイン タグの `x:xslt` と `x:xslt` タグの内部で使用できる 2 つのサブタグ (`x:stylesheet` および `x:xml`) で構成されます。

XSLT JSP タグ構文

XSLT JSP タグ構文は XML を基にしています。JSP タグは、開始タグ、本文 (省略可能)、対応する終了タグで構成されます。開始タグには要素名と属性 (省略可能) が含まれます。

次の構文では、WebLogic Server で提供される 3 つの XSLT JSP タグを JSP で使用する方法について示します。属性は省略可能です。また、`x:stylesheet` および `x:xml` タグも省略可能です。構文に続く表では、`x:xslt` および `x:stylesheet` タグの属性について示します。`x:xml` タグには属性はありません。

```
<x:xslt  [xml="uri of XML file"]
         [media="media type to determine stylesheet"]
         [stylesheet="uri of stylesheet"]
    <x:xml>In-line XML goes here
```

```
        </x:xml>
        <x:stylesheet [media="media type to determine stylesheet"]
                      [uri="uri of stylesheet"]
        </x:stylesheet>
    </x:xslt>
```

次の表に、x:xslt タグの属性を示します。

表 2-2 x:xslt JSP タグの属性

x:xslt タグの属性	必須	データ型	説明
xml	No	String	変換する XML ファイルの場所を指定する。場所は、タグが使用される Web アプリケーションのドキュメント ルートへの相対パス。
media	No	String	HTML または WML など、ドキュメントの出力タイプを定義する。このタイプによって、XML ドキュメントの変換時に使用するスタイルシートが決定される。 この属性は、x:xslt タグの本文の内部で x:stylesheet タグで囲まれた media 属性と組み合わせて使用できる。x:xslt タグの media 属性の値は x:stylesheet タグに囲まれた media 属性の値と比較される。値が等しい場合、x:stylesheet タグの uri 属性で指定されたスタイルシートが XML ドキュメントに適用される。 **注意 :** 同じ x:xslt タグ内で media 属性と stylesheet 属性の両方を設定するとエラーが発生する。
stylesheet	No	String	XML ドキュメントの変換に使用するスタイルシートの場所を指定する。場所は、タグが使用される Web アプリケーションのドキュメント ルートへの相対パス。 **注意 :** 同じ x:xslt タグ内で media 属性と stylesheet 属性の両方を設定するとエラーが発生する。

次の表に、x:stylesheet タグの属性を示します。

表 2-3 x:stylesheet JSP タグの属性

x:stylesheet タグの属性	必須	データ型	説明
media	No	String	HTML または WML など、ドキュメントの出力タイプを定義する。このタイプによって、XML ドキュメントの変換時に使用するスタイルシートが決定される。 この属性は x:xslt タグに囲まれた media 属性と組み合わせて使用する。x:xslt タグの media 属性の値は x:stylesheet タグに囲まれた media 属性の値と比較される。値が等しい場合、x:stylesheet タグの uri 属性で指定されたスタイルシートが XML ドキュメントに適用される。
uri	No	String	media 属性の値が、x:xslt タグに囲まれた media 属性の値と一致する場合に使用するスタイルシートの場所を指定する。場所は、タグが使用される Web アプリケーションのドキュメント ルートへの相対パス。

XSLT JSP タグの使い方

x:xslt タグは本文の有無に関係なく使用でき、属性は省略可能です。この節では、本文や 1 つまたは複数の属性を指定したかどうかによってタグの動作を決定するルールについて説明します。

x:xslt JSP タグが空タグ (本文なし) の場合、以下の説明が適用されます。

- 属性が設定されていない場合、XML ドキュメントはサーブレットのパスとデフォルトのメディア スタイルシートで処理されます。XML ファイルで、<?xml-stylesheet> 処理指示を使用してデフォルトのメディア スタイルシートを指定します。デフォルトのスタイルシートには media 属性はありません。

  この処理のタイプでは、XSLT 処理を実行するファイル サーブレットとしてタグ拡張を含む JSP ページを登録できます。
- media 属性のみが設定されている場合、XML ドキュメントは、サーブレットのパスと指定されたメディア タイプを使用して処理されます。

x:xslt タグの media タイプ属性の値は、XML ドキュメントにある <?xml-stylesheet> 処理指示の media 属性の値と比較されます。一致した場合は対応するスタイルシートが適用されます。一致しない場合はデフォルトのメディア スタイルシートが使用されます。メディア タイプ属性は、ドキュメントの出力タイプ (XML、HTML、ポストスクリプト、WML など) の定義に使用します。この機能を使用すると、ドキュメントの出力タイプごとにスタイルシートをまとめることができます。

- xml 属性のみが設定されている場合、指定した XML ドキュメントはデフォルトのメディア スタイルシートで処理されます。
- media および xml 属性を設定すると、指定した XML ドキュメントは指定したメディア タイプを使用して処理されます。
- stylesheet 属性が定義されている場合、XML ドキュメントは指定したスタイルシートで処理されます。

警告 : 同じ x:xslt タグ内で media 属性と stylesheet 属性の両方を設定するとエラーが発生します。

本文がある XSLT JSP タグは、<x:xml> タグまたは <x:stylesheet> タグを含んでいる場合があります。以下の説明が適用されます。

- <x:xml> タグを使用すると、XML ドキュメントをインライン処理用に指定できます。このタグには属性はありません。
- <x:stylesheet> タグは、属性を指定しない場合、デフォルトのスタイルシートをインラインで指定できます。
- <x:stylesheet> タグの uri 属性を使用して、デフォルト スタイルシートの場所を指定します。
- さまざまなメディア タイプごとに異なるスタイルシートを指定する場合は、media 属性に異なる値を指定して、複数の <x:stylesheet> タグを使用できます。タグの本文でメディア タイプごとのスタイルシートを指定したり、uri 属性を使ってスタイルシートの場所を指定したりできます。

XSLT JSP タグによる XML ドキュメントの変換

XSLT JSP タグで XML ドキュメントを変換するには、以下の手順を実行します。

1. *WL_HOME*\server\ext ディレクトリの xmlx.zip ファイルを開いて、xmlx-tags.jar ファイルを Web アプリケーションの /lib ディレクトリに移動します。*BEA Home* は、WebLogic Server 配布キットをインストールした最上位ディレクトリです。
2. <taglib> エントリを web.xml ファイルに追加します。次に例を示します。

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

3. タグを使用するには、次の行を JSP ページに追加します。

```
<%@ taglib uri="xmlx.tld" prefix="x"%>
```

4. トランスフォーマをコンフィグレーションします。以下の手順では、トランスフォーマのコンフィグレーションの一般的な方法を示します。
   a. 次のコード行を入力して xslt.jsp ファイルを作成します。

```
<%@ taglib uri="xmlx.tld" prefix="x"%><x:xslt/>
```

   b. 次のように、xslt.jsp ファイルを web.xml ファイルに登録します。

```
<servlet>
   <servlet-name>myxsltinterceptor</servlet-name>
   <jsp-file>xslt.jsp</jsp-file>
</servlet>
<servlet-mapping>
   <servlet-name>myxsltinterceptor</servlet-name>
   <url-pattern>/xslt/*</url-pattern>
</servlet-mapping>
```

   c. XML、DTD、XSL ドキュメントまたはサーブレットを Web アプリケーションに配置します。
   d. xslt プレフィックスを XML ドキュメントのパス名に追加 (たとえば、docs/fred.xml を xslt/docs/fred.xml に変更) し、ドキュメントにアクセスします。web.xml ファイルの <url-pattern> エントリによって、WebLogic Server は、XML ドキュメントで自動的に XSLT トランスフォーマを実行し、ドキュメントにデフォルトのスタイルシートを設定します。

e. メディア タイプを定義するには、XML ドキュメントのメディア タイプと出力のコンテンツ タイプを指定するコードを JSP に追加します。

f. メディア タイプを `xslt` タグに渡して、応答オブジェクトのコンテンツ タイプを設定します。

注意： 他の形の XSLT JSP タグは、スタイルシートが XML ドキュメントで指定されていない場合または XML スタイルシートがインラインで生成される場合に使用されます。

JSP での XSLT JSP タグの使用例

JSP から抜粋した以下のコードは、XSLT JSP タグを使用し、JSP を要求するクライアントのタイプにしたがって、XML を HTML または WML に変換する方法を示します。JSP は、クライアントがブラウザの場合は HTML を返し、無線デバイスの場合は WML を返します。

最初に、JSP は `HttpServletRequest` オブジェクトの `getHeader()` メソッドを使用して、JSP を要求するクライアントのタイプを識別します。次に、*`myMedia`* 変数を `wml` または `html` に適切に設定します。JSP で *`myMedia`* 変数を `html` に設定した場合は、*`content`* 変数に含まれる XML ドキュメントに `html.xsl` スタイルシートが適用されます。同様に、JSP で *`myMedia`* 変数を `wml` に設定した場合は、`wml.xsl` スタイルシートが適用されます。

```
<%
   String clientType = request.getHeader("User-Agent");
   // デフォルトは WML クライアント
   String myMedia = "wml";

   // クライアントが HTML ブラウザの場合
   if (clientType.indexOf("Mozilla") != -1) {
      myMedia = "http"
   }
%>

<x:xslt media="<%=myMedia%>">
  <x:xml><%=content%></x:xml>
  <x:stylesheet media="html" uri="html.xsl"/>
  <x:stylesheet media="wml"  uri="wml.xsl"/>
</x:xslt>
```

組み込みトランスフォーマ以外のトランスフォーマの使用

WebLogic Server の XML レジストリ (Administration Console でコンフィグレーション) では、以下の項目をコンフィグレーションします。

- 組み込みトランスフォーマをサーバ全体のトランスフォーマとして使用します。
- 組み込みトランスフォーマ以外のトランスフォーマをサーバ全体のトランスフォーマとして使用します。トランスフォーマは JAXP 準拠でなくてはなりません。

XML レジストリで変換オプションをコンフィグレーションする手順については、7-4 ページの「XML パーサおよびトランスフォーマのコンフィグレーション タスク」を参照してください。

3 XML アプリケーション スコーピング

以下の節では、特定のアプリケーションのためにパーサ、トランスフォーマ、外部エンティティ、外部エンティティ キャッシュをコンフィグレーションする方法について説明します。

- アプリケーション スコーピングの概要
- weblogic-application.xml ファイル
- エンタープライズ アプリケーション用のパーサまたはトランスフォーマのコンフィグレーション
- エンタープライズ アプリケーション用の外部エンティティのコンフィグレーション
- エンタープライズ アプリケーション用の外部エンティティ キャッシュのコンフィグレーション

アプリケーション スコーピングの概要

アプリケーション スコーピングとは、WebLogic Server コンフィグレーション全体ではなく、特定のエンタープライズ アプリケーション用のリソースをコンフィグレーションすることです。XML の場合、リソースとしては、パーサ、トランスフォーマ、外部エンティティ、外部エンティティ キャッシュ コンフィグレーションなどがあります。アプリケーション スコーピングの大きなメリットは、特定のアプリケーション用のリソースをアプリケーション自体から分離することです。アプリケーション スコーピングにより、アプリケーションごとに異なるパーサをコンフィグレーションすること、EAR ファイルや展開されたエンタープライズ ディレクトリ内のアプリケーション用の DTD を保管することなどができます。

アプリケーション スコーピングを使用するもう 1 つのメリットは、リソースを EAR ファイルと関連付けることにより、WebLogic Server の別のインスタンスで、そのサーバのリソースをコンフィグレーションせずにこの EAR ファイルを実行できることです。

特定のアプリケーション用の XML リソースをコンフィグレーションするには、EAR ファイルまたは展開されたエンタープライズ ディレクトリの META-INF ディレクトリにある weblogic-application.xml デプロイメント記述子ファイルに情報を追加します。

注意： WebLogic Server インスタンス用のパーサ、トランスフォーマ、外部エンティティ リソースをコンフィグレーションするには、第 7 章「WebLogic Server XML の管理」で説明するように、Administration Console を使用します。

weblogic-application.xml ファイル

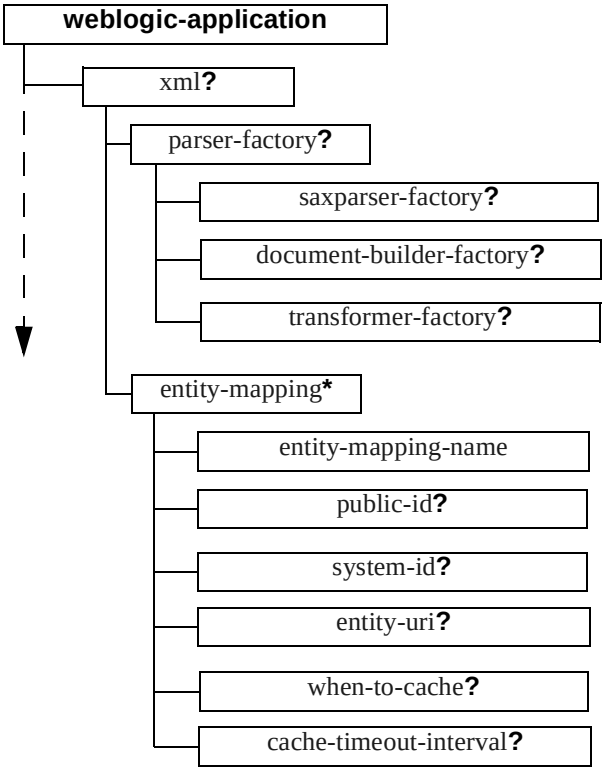
weblogic-application.xml ファイルは、エンタープライズ アプリケーション用の WebLogic Server 独特のデプロイメント記述子です。このファイルには、エンタープライズ アプリケーションにより使用される XML、JDBC、EJB などのリソースに関するコンフィグレーション情報が入れられます。標準 J2EE デプロイメント記述子は、application.xml と呼ばれます。

次のサンプル ファイル weblogic-application.xml は、あるエンタープライズ アプリケーション用の XML リソースをコンフィグレーションする方法を示しています。各種要素の本文は太字で示します。

```
<weblogic-application>
  ...
  <xml>
     <parser-factory>
        <saxparser-factory>
           weblogic.xml.babel.jaxp.SAXParserFactoryImpl
        </saxparser-factory>
        <document-builder-factory>
           org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
        </document-builder-factory>
        <transformer-factory>
          org.apache.xalan.processor.TransformerFactoryImpl
        </transformer-factory>
```

```
      </parser-factory>
      <entity-mapping>
        <entity-mapping-name>My Mapping</entity-mapping-name>
        <public-id>-//BEA Systems, Inc.//DTD for cars//EN</public-id>
        <system-id>http://www.bea.com/dtds/car.dtd</system-id>
        <entity-uri>dtds/car.dtd</entity-uri>
      </entity-mapping>
  </xml>
</weblogic-application>
```

XML リソースをコンフィグレーションするためのメイン要素は、<xml> です。次の図に、<xml> 要素のサブ要素を示し、その後の節で各要素について説明します。



? = 省略可能
+ = 1 つまたは複数
*** = ゼロまたは 1 つ以上**

xml

パーサ、トランスフォーマ、外部エンティティ、外部エンティティ キャッシュなど、エンタープライズ アプリケーション用 XML リソースをコンフィグレーションするためのメイン要素。

parser-factory

エンタープライズ アプリケーション用の特定のパーサまたはトランスフォーマを指定するための親要素。

saxparser-factory

このアプリケーションで SAX スタイル解析のために使用されるファクトリ クラスを指定する要素。この要素が指定されていない場合、WebLogic Server インスタンスに対して指定されたデフォルト SAX パーサ ファクトリが使用されます。

document-builder-factory

このアプリケーションで DOM スタイル解析のために使用されるファクトリ クラスを指定する要素。この要素が指定されていない場合、WebLogic Server インスタンスに対して指定されたデフォルト DOM パーサ ファクトリが使用されます。

transformer-factory

このアプリケーションで `javax.xml.transform` パッケージによりドキュメントを変換する場合に使用されるファクトリ クラスを指定する要素。この要素が指定されていない場合、WebLogic Server インスタンスに対して指定されたデフォルト XSLT トランスフォーマ ファクトリが使用されます。

entity-mapping

DTD やスキーマなど、XML ファイル内のエンティティ宣言をエンティティのローカル コピーにマッピングするための親要素。

entity-mapping-name

エンティティ マッピング宣言の名前を指定する要素。

public-id

次のようなエンティティのパブリック ID を指定する要素。

```
-//BEA Systems, Inc.//DTD for cars//EN.
```

system-id

次のようなエンティティのシステム ID を指定する要素。

```
http://www.bea.com/dtds/car.dtd
```

entity-uri

エンティティの URI を指定する要素。このパスは、EAR アーカイブまたは展開ディレクトリのメイン ディレクトリを基準とする相対パスです。

たとえば、`dtds/car.dtd` は、メイン EAR アーカイブ (`META-INF` ディレクトリと同じレベル) に `dtds` という名前のディレクトリがあり、そこに `car.dtd` という名前のファイルがあることを示します。

when-to-cache

外部エンティティをキャッシュする時期を指定する要素。有効な値は次のとおりです。

- [参照時] - WebLogic Server は、エンティティが XML ドキュメントで初めて参照されたときに、URL で参照される外部エンティティをキャッシュします。
- [初期化時] - WebLogic Server は、サーバ起動時にエンティティをキャッシュします。
- [キャッシュしない] - WebLogic Server は、外部エンティティをキャッシュしません。

デフォルト値は `cache-on-reference` です。

cache-timeout-interval

キャッシュされた外部エンティティが古くなる (期限切れになる) までの秒数を指定する要素。キャッシュされたコピーがこの期間を超えた場合、WebLogic Server は、EAR アーカイブまたは展開ディレクトリのメイン ディレクトリを基準に指定された相対 URL またはパス名から外部エンティティを再取得します。

このフィールドのデフォルト値は、120 秒です。

エンタープライズ アプリケーション用のパーサまたはトランスフォーマのコンフィグレーション

XML アプリケーションが、WebLogic Server に対してコンフィグレーションされた組み込みパーサまたはトランスフォーマと異なるパーサまたはトランスフォーマを使用するように指定できます。それには、XML アプリケーションの入っている EAR ファイルまたは展開ディレクトリの `weblogic-application.xml` を更新します。

エンタープライズ アプリケーション用に組み込み以外のパーサまたはトランスフォーマをコンフィグレーションするには、以下の手順を実行します。

1. <xml> 要素の <parser-factory> サブ要素を使用し、エンタープライズ アプリケーションのために、SAX および DOM スタイル解析用および XSLT 変換用のファクトリ クラスをコンフィグレーションします。次に例を示します。

```
<parser-factory>
   <saxparser-factory>
      weblogic.xml.babel.jaxp.SAXParserFactoryImpl
   </saxparser-factory>
   <document-builder-factory>
      org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
   </document-builder-factory>
   <transformer-factory>
     org.apache.xalan.processor.TransformerFactoryImpl
   </transformer-factory>
</parser-factory>
```

   この weblogic-application.xml ファイルに対応するアプリケーションでは、SAX スタイル解析には weblogic.xml.babel.jaxp.SAXParserFactoryImpl ファクトリ クラス、DOM スタイル解析には org.apache.xerces.jaxp.DocumentBuilderFactoryImpl ファクトリ クラス、XSLT 変換には org.apache.xalan.processor.TransformerFactoryImpl クラスが使用されます。

2. パーサ クラスまたはトランスフォーマ クラスを EAR アーカイブに対してローカルにするには、対象のクラスの入っている JAR ファイルを EAR ファイルの任意の場所に置き、META-INF/MANIFEST.MF ファイル内の Class-Path 変数を更新します。

   たとえば、ディレクトリ lib/xml にある myparser.jar という名前の JAR ファイルにパーサ クラスまたはトランスフォーマ クラスを置くには、以下に示すように MANIFEST.MF ファイルを更新します。

```
Manifest-Version: 1.0
Created-By: 1.3.1_01 (Sun Microsystems Inc.)
Class-Path: lib/xml/myparser.jar
```

3. パーサ クラスまたはトランスフォーマ クラスを EAR アーカイブ以外の場所に格納する場合は、クラスの入っている JAR ファイルの完全パス名を含むように WebLogic Server CLASSPATH 変数を必ず更新してください。

エンタープライズ アプリケーション用の外部エンティティのコンフィグレーション

DTD などの外部エンティティは、Web から取得するだけでなく、そのローカル コピーを EAR アーカイブまたは展開ディレクトリに格納しておくこともできます。

エンタープライズ アプリケーション用の外部エンティティをコンフィグレーションするには、以下の手順を実行します。

1. EAR アーカイブのメイン ディレクトリの下にディレクトリ `lib/xml/registry` を作成します。
2. DTD のような外部エンティティをそのディレクトリにコピーします。
3. `weblogic-application.xml` ファイルを更新し、`<xml>` 要素の `<entity-mapping>` サブ要素を使用して、アプリケーションにより処理される XML ファイルのエンティティ宣言にエンティティの名前をマップします。以下に例を示します。

   ```
   <entity-mapping>
     <entity-mapping-name>My Mapping</entity-mapping-name>
     <public-id>-//BEA Systems, Inc.//DTD for cars//EN</public-id>
     <system-id>http://www.bea.com/dtds/car.dtd</system-id>
     <entity-uri>dtds/car.dtd</entity-uri>
   </entity-mapping>
   ```

   この例では、`car.dtd` という DTD のローカル コピーが EAR アーカイブまたは展開ディレクトリのメイン ディレクトリの下の `lib/xml/registry/dtds` ディレクトリに格納されます。このエンティティのパブリック ID は `-//BEA Systems, Inc.//DTD for cars//EN` で、システム ID は `http://www.bea.com/dtds/car.dtd` です。アプリケーションが XML ファイルの解析中に、これらの ID のいずれかを使用しているエンティティ宣言を検出すると、その宣言は `car.dtd` ファイルに置き換えられます。

注意： エンティティのローカル コピーをマップする各エンティティ宣言について、`<entity-mapping>` 要素を指定してください。

エンタープライズ アプリケーション用の外部エンティティ キャッシュのコンフィグレーション

サーバ起動時、またはエンティティの最初の参照時のいずれかに、EAR アーカイブのメイン ディレクトリを基準にした相対 URL またはパス名で参照される外部エンティティを WebLogic Server がキャッシュするように指定できます。

外部エンティティをキャッシュすると、リモート アクセス時間が節約されるだけでなく、ネットワークまたは管理サーバのダウンにより XML ドキュメントの解析中に管理サーバにアクセスできなくなった場合には、ローカル バックアップを利用できます。

キャッシュされたエンティティについて、WebLogic Server が URL または EAR のディレクトリからエンティティを再取得して再キャッシュする間隔 (有効期限) をコンフィグレーションできます。

エンタープライズ アプリケーション用の外部エンティティ キャッシュをコンフィグレーションするには、`<entity-mapping>` 要素の `<when-to-cache>` サブ要素と `<cache-timeout-interval>` サブ要素を使用します。以下に例を示します。

```
<entity-mapping>
  <entity-mapping-name>My Mapping</entity-mapping-name>
  <public-id>-//BEA Systems, Inc.//DTD for cars//EN</public-id>
  <system-id>http://www.bea.com/dtds/car.dtd</system-id>
  <entity-uri>dtds/car.dtd</entity-uri>
 <when-to-cache>cache-at-initialization</when-to-cache>
  <cache-timeout-interval>300</cache-timeout-interval>
</entity-mapping>
```

この例では、`car.dtd` が EAR アーカイブまたは展開ディレクトリのメイン ディレクトリの下の `lib/xml/registry/dtds` ディレクトリに格納されます。WebLogic Server の最初の起動時に DTD のコピーがメモリにキャッシュされ、以後は、キャッシュされたコピーが 300 秒以上格納されている場合にリフレッシュされます。

4 WebLogic XML Streaming API の使い方

以下の節では、WebLogic XML Streaming API により XML ドキュメントの解析と生成を行う方法について説明します。

WebLogic XML Streaming API の概要

WebLogic XML Streaming API は、XML ドキュメントの解析と生成を簡単に直感的に行える方法を提供します。SAX API に似ていますが、複雑な XML ドキュメントを処理する際の煩雑さの原因になる SAX イベント ハンドラを書く必要はなく、手順の流れに従って XML ドキュメントを処理できます。つまり、Streaming API では、SAX API より解析を綿密に制御できます。

SAX を使用して XML ドキュメントを解析する場合、プログラムでは、イベント発生時に解析をリスンするイベント リスナを作成し、特定のイベントを請求するのではなく、イベントに対して対応する必要があります。一方、Streaming API を使用する場合は、XML ドキュメントを手順の流れに沿って処理すること、特定のタイプのイベント (要素の開始など) を請求すること、要素の属性を繰り返し処理すること、ドキュメントの先頭をスキップすること、任意の時点で処理を停止すること、特定の要素のサブ要素を取得すること、指定どおりに要素をフィルタすることができます。イベントに対応するのではなく、イベントを請求するので、Streaming API を使用する方法は、「プル解析」と呼ばれることもあります。

Streaming API により、オペレーティング システム上の XML ファイル、DOM ツリー、SAX イベントのセットなど、多くのタイプの XML ドキュメントを解析できます。これらの XML ドキュメントをイベントのストリーム、すなわち `XMLInputStream`、に変換し、このストリームを順に処理して、要素の開始、ドキュメントの終了などのイベントを必要に応じてスタックから取得します。

WebLogic Streaming API は、デフォルト パーサとして WebLogic FastParser を使用します。

Streaming API による XML ドキュメント解析の詳細なサンプルについては、*`WL_HOME`*`\samples\server\src\examples\xml\orderParser` ディレクトリを参照してください。*`WL_HOME`* は、最上位 WebLogic Platform ディレクトリです。

次の表で、WebLogic Streaming API の主なインタフェースとクラスについて説明します。

表 4-1 XML Streaming API のインタフェースとクラス

インタフェースまたはクラス	説明
`XMLInputStreamFactory`	XML ドキュメント解析用の `XMLInputStream` オブジェクトを作成するために使用されるファクトリ。
`XMLInputStream`	イベントの入力ストリームを入れるために使用されるインタフェース。
`BufferedXMLInputStream`	`XMLInputStream` インタフェースでストリームのマーキングとリセットを行えるようにするための拡張。
`XMLOutputStreamFactory`	XML ドキュメント生成用の `XMLOutputStream` オブジェクトを作成するために使用されるファクトリ。
`XMLOutputStream`	イベントを書き込むために使用されるインタフェース。
`ElementFactory`	この API で使用されるインタフェースのインスタンスを作成するためのユーティリティ。

表 4-1 XML Streaming API のインタフェースとクラス

インタフェースまたはクラス	説明
`XMLEvent`	要素の開始、要素の終了など、XML ドキュメントでのすべてのタイプのイベントのためのベース インタフェース。
`StartElement`	`XMLEvent` サブ インタフェースの最重要部。XML ドキュメントの開始要素に関する情報を取得するために使用される。
`AttributeIterator`	要素の属性のセットを繰り返し処理するために使用されるオブジェクト。
`Attribute`	要素の特定の属性を記述するオブジェクト。

WebLogic XML Streaming API の Javadoc

以下の Javadoc には、この章で説明した WebLogic XML Streaming API 機能に加え、詳しく説明されていない機能に関するリファレンスが掲載されています。

- weblogic.xml.stream
- weblogic.xml.stream.util

XML ドキュメントの解析：一般的手順

以下の手順は、WebLogic XML Streaming API により XML ドキュメントの解析と操作を行う一般的手順について説明したものです。

最初の 2 つの手順は必須です。その後の手順は、XML ファイルをどのように処理するかに応じて実行します。

1. `weblogic.xml.stream.*` クラスをインポートします。

2. XML ファイル、DOM ツリー、または SAX イベントのセットからイベントの XML ストリームを取得します。XML ストリームをフィルタすることにより、特定のタイプのイベント、特定の要素の名前などのみを取得することもできます。4-7 ページの「XML 入力ストリームの取得」を参照してください。
3. ストリームを繰り返し処理し、汎用 `XMLEvent` タイプを返します。4-12 ページの「ストリームの繰り返し処理」を参照してください。
4. 汎用 `XMLEvent` タイプのそれぞれについて、イベント タイプを判別します。イベント タイプには、XML ドキュメント、要素の終了、エンティティ参照などがあります。4-12 ページの「特定の XMLEvent タイプの判別」を参照してください。
5. 要素の属性を取得します。4-16 ページの「要素の属性の取得」を参照してください。
6. イベント全体のスキップ、特定のイベントのスキップなどにより、ストリームを位置決めします。4-18 ページの「ストリームの位置決め」を参照してください。
7. 要素の子を取得します。4-19 ページの「サブストリームの取得」を参照してください。
8. ストリームをクローズします。4-21 ページの「入力ストリームのクローズ」を参照してください。

XML ドキュメント解析の例

以下の節では、XML Streaming API により XML ドキュメントを解析する例を示します。

このプログラムは、XML ファイルを表わす 1 つのパラメータを取り、そのファイルを XML 入力ストリームに変換します。次に、ストリームを繰り返し処理し、XML 要素の開始、XML ドキュメントの終了など、各イベントのタイプを判別します。開始要素、終了要素、および要素の本文を形成する文字データの 3 つのタイプのイベントについては、情報がプリントアウトされます。コメントや XML ドキュメントの開始など、その他のタイプのイベントについては、何も行われません。

注意： 太字のコードについては、例の後の節で詳しく説明します。

```
package examples.xml.stream;

import weblogic.xml.stream.Attribute;
import weblogic.xml.stream.AttributeIterator;
import weblogic.xml.stream.ChangePrefixMapping;
import weblogic.xml.stream.CharacterData;
import weblogic.xml.stream.Comment;
import weblogic.xml.stream.XMLEvent;
import weblogic.xml.stream.EndDocument;
import weblogic.xml.stream.EndElement;
import weblogic.xml.stream.EntityReference;
import weblogic.xml.stream.ProcessingInstruction;
import weblogic.xml.stream.Space;
import weblogic.xml.stream.StartDocument;
import weblogic.xml.stream.StartPrefixMapping;
import weblogic.xml.stream.StartElement;
import weblogic.xml.stream.EndPrefixMapping;
import weblogic.xml.stream.XMLInputStream;
import weblogic.xml.stream.XMLInputStreamFactory;
import weblogic.xml.stream.XMLName;
import weblogic.xml.stream.XMLStreamException;

import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class ComplexParse {

  /**
   * ストリームでハンドルを取得するためのヘルパー メソッド
   * 名前で取得し、ストリームを戻す
   * このメソッドは、InputStreamFactory を使用して、
   * XMLInputStream のインスタンスを作成する
   * @param name 解析するファイル
   * @return XMLInputStream 解析するストリーム
   */
  public XMLInputStream getStream(String name)
    throws XMLStreamException, FileNotFoundException
  {
    XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
   XMLInputStream stream = factory.newInputStream(new FileInputStream(name));
    return stream;
  }

  /**
   *  要素の開始、ドキュメントの終了など、
   *  イベントのタイプを識別する
   *  イベントのタイプが START_ELEMENT, END_ELEMENT または
```

```
   *  CHARACTER_DATA である場合、メソッドは該当する情報を
   *  プリントアウトする。そうでない場合、何もしない
   *  @param event 解析された XML イベント
   */
  public void parse(XMLEvent event)
    throws XMLStreamException
  {
    switch(event.getType()) {
    case XMLEvent.START_ELEMENT:
      StartElement startElement = (StartElement) event;
      System.out.print("<" + startElement.getName().getQualifiedName() );
     AttributeIterator attributes = startElement.getAttributesAndNamespaces();
      while(attributes.hasNext()){
        Attribute attribute = attributes.next();
        System.out.print(" " + attribute.getName().getQualifiedName() +
                         "='" + attribute.getValue() + "'");
      }
      System.out.print(">");
      break;
    case XMLEvent.END_ELEMENT:
      System.out.print("</" + event.getName().getQualifiedName() +">");
      break;
    case XMLEvent.SPACE:
    case XMLEvent.CHARACTER_DATA:
      CharacterData characterData = (CharacterData) event;
      System.out.print(characterData.getContent());
      break;
    case XMLEvent.COMMENT:
      // コメントを出力
      break;
    case XMLEvent.PROCESSING_INSTRUCTION:
      // ProcessingInstruction を出力
      break;
    case XMLEvent.START_DOCUMENT:
      // StartDocument を出力
      break;
    case XMLEvent.END_DOCUMENT:
      // EndDocument を出力
      break;
    case XMLEvent.START_PREFIX_MAPPING:
      // StartPrefixMapping を出力
      break;
    case XMLEvent.END_PREFIX_MAPPING:
      // EndPrefixMapping を出力
      break;
    case XMLEvent.CHANGE_PREFIX_MAPPING:
```

```
      // ChangePrefixMapping を出力
      break;
    case XMLEvent.ENTITY_REFERENCE:
      // EntityReference を出力
      break;
    case XMLEvent.NULL_ELEMENT:
      throw new XMLStreamException("Attempt to write a null event.");
    default:
      throw new XMLStreamException("Attempt to write unknown event
                                    ["+event.getType()+"]");
    }
  }
  /**
   * ストリーム全体を繰り返し処理するヘルパー メソッド
   * @param name 解析するファイル
   */
  public void parse(XMLInputStream stream)
    throws XMLStreamException
  {
    while(stream.hasNext()) {
      XMLEvent event = stream.next();
      parse(event);
    }
    stream.close();
  }

  /** メイン メソッド。XML 入力ストリームに変換される
   *  XML ファイルを表わす 1 つの引数を取る
   */
  public static void main(String args[])
    throws Exception
  {
    ComplexParse complexParse= new ComplexParse();
    complexParse.parse(complexParse.getStream(args[0]));
  }
}
```

XML 入力ストリームの取得

XML Streaming API により、XML ファイル DOM ツリー、SAX イベントなど、さまざまなオブジェクトをイベントのストリームに変換できます。

次の例で、XML ファイルからイベントのストリームを変換する方法を示します。

```
XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
XMLInputStream stream = factory.newInputStream(new FileInputStream(name));
```

まず、XMLInputStreamFactory の新しいインスタンスを作成し、次に、このファクトリを使用して、name 変数で参照される XML ファイルから新しい XMLInputStream を作成します。

次の例では、DOM ツリーからストリームを作成する方法を示します。

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setValidating(false);
dbf.setNamespaceAware(true);
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse(new java.io.File(file));
XMLInputStream stream = XMLInputStreamFactory.newInstance().newInputStream(doc);
```

バッファされた XML 入力ストリームの取得

XMLInputStream オブジェクトの繰り返し処理を終了した後は、このストリームに再度アクセスすることはできません。ただし、再度ストリームを処理する必要がある場合、たとえば、ストリームを別のアプリケーションに送信したり、別の方法で再度繰り返し処理したりする場合は、XMLInputStream オブジェクトではなく、BufferedXMLInputStream オブジェクトを使用できます。

バッファされた XML 入力ストリームを作成するには、XMLInputStreamFactory クラスの newBufferedInputStream() メソッドを使用します。以下に例を示します。

```
XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
BufferedXMLInputStream bufstream =
    factory.newBufferedInputStream(factory.newInputStream(new
    FileInputStream(name)));
```

BufferedXMLInputStream オブジェクトの mark() メソッドと reset() メソッドにより、ストリームの特定のスポットをマークし、ストリームの処理を続け、マークしたスポットに戻るようにストリームをリセットできます。詳細については、4-20 ページの「バッファされた XML 入力ストリームのマーキングとリセット」を参照してください。

XML ストリームのフィルタリング

XML ストリームのフィルタリングとは、指定したタイプのオブジェクトのみの入ったストリームを作成することです。たとえば、開始要素、終了要素、および XML 要素の本文をマークアップする文字データのみの入ったストリームを作成できます。別の例では、指定した名前を持つ要素のみがストリームに出現するように XML ストリームをフィルタリングすることもできます。

XML ストリームをフィルタするには、`XMLInputStreamFactory.newInputStream()` メソッドに対する 2 番目のパラメータとしてフィルタ クラスを指定します。フィルタ クラスに対するパラメータとして XML ストリームに入れるイベントを指定します。次の例では、`TypeFilter` クラスにより、結果の XML ストリームに XML の開始要素と終了要素、および文字データのみを入れるように指定する方法を示します。

```
import weblogic.xml.stream.util.TypeFilter;

XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
XMLInputStream stream = factory.newInputStream(new FileInputStream(name),
                        new TypeFilter(XMLEvent.START_ELEMENT |
                                       XMLEvent.END_ELEMENT |
                                       XMLEvent.CHARACTER_DATA));
```

次の表で、WebLogic XML Streaming API に用意されているフィルタについて説明します。これらは、weblogic.xml.stream.util パッケージの一部です。

表 4-2

フィルタの名前	説明	使用例
TypeFilter	XMLEvent.START_ELEMENT、XMLEvent.END_ELEMENT など、指定されたイベント タイプに基づいて XML ストリームをフィルタする。イベント タイプの詳細なリストは、4-12 ページの「特定の XMLEvent タイプの判別」参照。 TypeFilter は、入力として整数ビットマスクを取る。例に示すとおり、このビットマスクを作成するには、値の OR (論理和) を取る。	new TypeFilter (XMLEvent.START_ELEMENT | XMLEvent.END_ELEMENT | XMLEvent.CHARACTER_DATA)
NameFilter	XML ドキュメント内の要素の名前に基づいて XML ストリームをフィルタする。	new NameFilter ("Book")
NameSpaceFilter	指定されたネームスペース URI に基づいて XML ストリームをフィルタする。	new NameSpaceFilter ("http://namespace.org")
NamespaceTypeFilter	指定されたイベント タイプとネームスペース URI に基づいて XML ストリームをフィルタする。このフィルタは、TypeFilter と NameSpaceFilter の機能を結合したものである。	new NamespaceFilter ("http://namespace.org", XMLEvent.START_ELEMENT) この例は、すべての開始要素が指定されたネームスペースを持っているストリームを返す。

カスタム フィルタの作成

API に入っているフィルタがニーズに合わない場合は、独自のフィルタを作成することもできます。

1. ElementFilter インタフェースを実装し、accept(XMLEvent) という名前のメソッドの入ったクラスを作成します。このメソッドは、特定のイベントをストリームに追加するかどうかを XMLInputStreamFactory.newInputStream() メソッドに通知します。以下に例を示します。

   ```
   package my.filters;

   import weblogic.xml.stream.XMLName;
   import weblogic.xml.stream.ElementFilter;
   import weblogic.xml.stream.events.NullEvent;

   public class SuperDooperFilter implements ElementFilter {

     protected String name;

     public SuperDooperFilter(String name)
     {
       this.name = name;
     }

     public boolean accept(XMLEvent e) {
       if (name.equals(e.getName().getLocalName()))
         return true;
       return false;
     }
   }
   ```

2. XML アプリケーションで、新しいフィルタ クラスをインポートしていることを確認します。

   ```
   import my.filters.SuperDooperFilter
   ```

3. newInputStream() メソッドに対する 2 番目のパラメータとしてフィルタを指定します。これは、XML ストリームに入れるイベントのタイプを、フィルタ クラスが必要とするフォーマットでクラス フィルタに渡します。

```
XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
XMLInputStream stream = factory.newInputStream(new FileInputStream(name),
                        new SuperDooperFilter(param));
```

ストリームの繰り返し処理

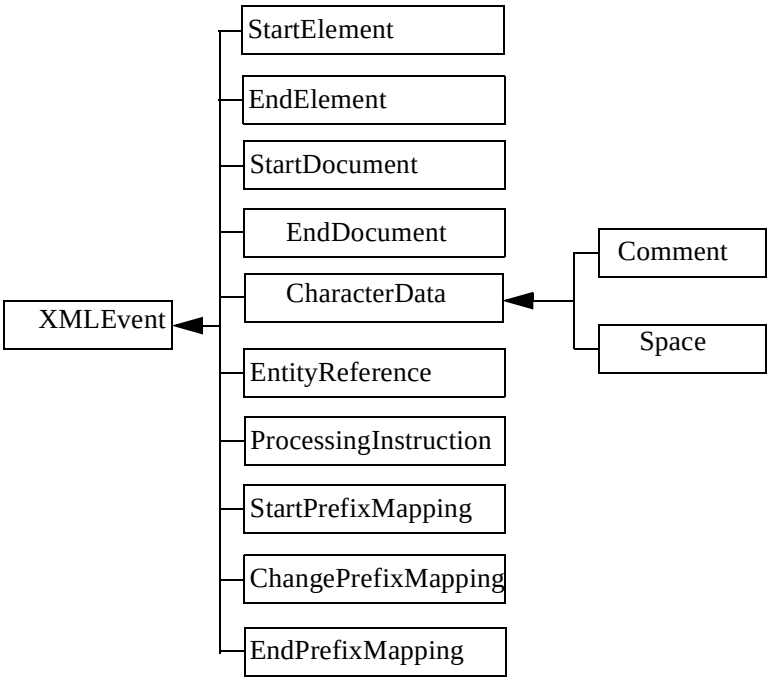
イベントのストリームが完成した後、次の手順は、XMLInputStream.next() メソッドと XMLInputStream.hasNext() メソッドにより、イベントのストリームを手順に沿って処理することです。以下に例を示します。

```
while(stream.hasNext()) {
  XMLEvent event = stream.next();
  System.out.print(event);
}
```

特定の XMLEvent タイプの判別

XMLInputStream.next() メソッドは、タイプ XMLEvent のオブジェクトを返します。XMLEvent には、このイベントが、XML ドキュメントの開始、要素の終了、エンティティ参照などのどれであるかをさらに分類するサブインタフェースがあります。XMLEvent インタフェースには、対応するフィールド、すなわち定

数、ならびに実際のイベントを識別するために使用できるメソッドのセットも入っています。次の図に、`XMLEvent` インタフェースとそのサブインタフェースの階層を示します。



次の表に、XML ストリームの解析時に特定のイベントを識別するために使用できる `XMLEvent` クラスのサブクラスとフィールドをリストします。

表 4-3 XMLEvent クラスのサブクラスとフィールド

XMLEvent サブクラス	サブクラスを識別するために使用される XMLEvent クラスのフィールド	サブクラスを識別するために使用されるメソッド	サブクラス イベントの説明
ChangePrefixMapping	CHANGE_PREFIX_MAPPING	isChangePrefixMapping	プレフィックス マッピングが古いネームスペースから新しいネームスペースに変更されたことを示す。

表 4-3 XMLEvent クラスのサブクラスとフィールド

XMLEvent サブクラス	サブクラスを識別するために使用される XMLEvent クラスのフィールド	サブクラスを識別するために使用されるメソッド	サブクラス イベントの説明
CharacterData	CHARACTER_DATA	isCharacterData	返された XMLEvent オブジェクトに要素の本文からの文字データが入っていることを示す。
Comment	COMMENT	isComment	返された XMLEvent オブジェクトに XML コメントが入っていることを示す。
EndDocument	END_DOCUMENT	isEndDocument	XML ドキュメントの終了を示す。
EndElement	END_ELEMENT	isEndElement	XML ドキュメントの要素の終了を示す。
EndPrefixMapping	END_PREFIX_MAPPING	isEndPrefixMapping	プレフィックス マッピングがスコープの範囲外になったことを示す。
EntityReference	ENTITY_REFERENCE	isEntityReference	返された XMLEvent オブジェクトにエンティティ参照が入っていることを示す。
ProcessingInstruction	PROCESSING_INSTRUCTION	isProcessingInstruction	返された XMLEvent オブジェクトに処理指示が入っていることを示す。
Space	SPACE	isSpace	返された XMLEvent オブジェクトにホワイトスペースが入っていることを示す。
StartDocument	START_DOCUMENT	isStartDocument	XML ドキュメントの開始を示す。

表 4-3 XMLEvent クラスのサブクラスとフィールド

XMLEvent サブクラス	サブクラスを識別するために使用される XMLEvent クラスのフィールド	サブクラスを識別するために使用されるメソッド	サブクラス イベントの説明
StartElement	START_ELEMENT	isStartElement	XML ドキュメントの要素の開始を示す。
StartPrefixMapping	START_PREFIX_MAPPING	isStartPrefixMapping	プレフィックス マッピングがそのスコープを開始したことを示す。

次の例では、Java case 文を使用して、`XMLInputStream.next()` メソッドにより返されたイベントのタイプを判別する方法を示します。簡単にするため、この例では、見つかったイベントをそのままプリントします。イベントをさらに処理する方法は、その後の節で示します。

```
switch(event.getType()) {
case XMLEvent.START_ELEMENT:
  // 要素の開始
  System.out.println ("Start Element\n");
  break;

case XMLEvent.END_ELEMENT:
  // 要素の終了
  System.out.println ("End Element\n");
  break;

case XMLEvent.PROCESSING_INSTRUCTION:
  // 処理指示
  System.out.println ("Processing instruction\n");
  break;

case XMLEvent.SPACE:
  // ホワイトスペース
  System.out.println ("White space\n");
  break;

case XMLEvent.CHARACTER_DATA:
  // 文字データ
  System.out.println ("Character data\n");
  break;
```

```
case XMLEvent.COMMENT:
  // コメント
  System.out.println ("Comment\n");
  break;

case XMLEvent.START_DOCUMENT:
  // XML ドキュメントの開始
  System.out.println ("Start Document\n");
  break;

case XMLEvent.END_DOCUMENT:
  // XML ドキュメントの終了
  System.out.println ("End Document\n");
  break;

case XMLEvent.START_PREFIX_MAPPING:
  // プレフィックス マッピング スコープの開始
  System.out.println ("Start prefix mapping\n");
  break;

case XMLEvent.END_PREFIX_MAPPING:
  // プレフィックス マッピング スコープの終了
  System.out.println ("End prefix mapping\n");
  break;

case XMLEvent.CHANGE_PREFIX_MAPPING:
  // プレフィックス マッピングがネームスペースを変更
  System.out.println ("Change prefix mapping\n");
  break;

case XMLEvent.ENTITY_REFERENCE:
  // エンティティ参照
  System.out.println ("Entity reference\n");
  break;
default:

  throw new XMLStreamException("Attempt to parse unknown event
                                [" + event.getType() + "]");
}
```

要素の属性の取得

XMLドキュメントの要素の属性を取得するには、まず、`XMLInputStream.next()` メソッドにより返された `XMLEvent` オブジェクトを `StartElement` オブジェクトに対してキャストする必要があります。

要素が何個の属性を持っているか分からないため、まず、属性のリスト全体を入れる AttributeIterator オブジェクトを作成してから、属性がなくなるまでリストを繰り返し処理します。次の例で、4-12 ページの「ストリームの繰り返し処理」に示した switch 文の START_ELEMENT case の一部としてこれを行う方法を示します。

```
case XMLEvent.START_ELEMENT:

    StartElement startElement = (StartElement) event;
    System.out.print("<" + startElement.getName().getQualifiedName() );
    AttributeIterator attributes = startElement.getAttributesAndNamespaces();
    while(attributes.hasNext()){
      Attribute attribute = attributes.next();
      System.out.print(" " + attribute.getName().getQualifiedName() +
                       "='" + attribute.getValue() + "'");
    }
    System.out.print(">");
    break;
```

この例では、まず、返された XMLEvent を StartElement にキャストすることにより、StartElement オブジェクトを作成します。次に、StartElement.getAttributesAndNamespaces() メソッドを使用して AttributeIterator オブジェクトを作成し、AttributeIterator.hasNext() メソッドを使用して属性を繰り返し処理します。Attribute それぞれについて、Attributes.getName().getQualifiedName() メソッドと Attribute.getValue() メソッドを使用し、属性の名前と値を返します。

getNamespace() メソッドと getAttributes() メソッドにより、ネームスペースまたは属性のみを返すこともできます。

ストリームの位置決め

次の表で、ストリームの特定の位置までスキップするために使用できる XMLInputStream インタフェースのメソッドについて説明します。

表 4-4 入力ストリームの位置決めのために使用されるメソッド

XMLInputStream のメソッド	説明
skip()	入力ストリームを次のストリーム イベントに位置決めする。 **注意：** 次のイベントは、XML ファイル内の実際の要素であるとは限らない。コメントやホワイトスペースの場合もある。
skip(int)	入力ストリームをこのタイプの次のイベントに位置決めする。 イベント タイプの例としては、XMLEvent.START_ELEMENT や XMLEvent.END_DOCUMENT などがある。イベント タイプの詳細なリストは、表 4-3 を参照。
skip(XMLName)	入力ストリームをこの名前の次のイベントに位置決めする。
skip(XMLName, int)	入力ストリームをこの名前とタイプの次のイベントに位置決めする。
skipElement()	次の要素にスキップする (現在の要素のサブ要素にはスキップしない)。
peek()	ストリームから実際には読み込まずに、次のイベントをチェックする。

次の例では、入力ストリームを繰り返し処理する基本コードを修正し、XML 要素の本文の文字データをスキップする方法を示します。

```
while(stream.hasNext()) {
  XMLEvent peek = stream.peek();
  if (peek.getType() == XMLEvent.CHARACTER_DATA ) {
    stream.skip();
    continue;
```

```
      }
      XMLEvent event = stream.next();
      parse(event);
    }
```

この例は、XMLInputStream.peek() メソッドを使用してストリーム上の次のイベントを判別する方法を示しています。イベントのタイプが XMLEvent.CHARACTER_DATA である場合、そのイベントはスキップされ、次のイベントに進みます。

サブストリームの取得

すべてのサブ要素も含め、次の要素のコピーを取得するには、XMLInputStream.getSubStream() メソッドを使用します。getSubstream() メソッドは、XMLInputStream オブジェクトを返します。親ストリーム (または getSubStream() を呼び出したストリーム) での位置は移動しません。親ストリームでは、getSubStream() で取得した要素をスキップしたい場合、skipElement() メソッドを使用します。

getSubStream() メソッドは、検出された START_ELEMENT イベントと END_ELEMENT イベントの数を記録し、その数が等しくなると (すなわち、完全な次の要素が見つかると) 停止して、結果のサブストリームを XMLInputStream オブジェクトとして返します。

たとえば、XML Streaming API を使用して以下の XML ドキュメントを解析するとします。ただし、<content> タグと </content> タグに囲まれたサブストリームのみを対象にします。

```
<book>
  <title>The History of the World</title>
  <author>Juliet Shackell</author>
  <publisher>CrazyDays Publishing</publisher>
  <content>
      <chapter title='Just a Speck of Dust'>
        <synopsis>The world as a speck of dust</synopsis>
        <para>Once the world was just a speck of dust...</para>
      </chapter>
      <chapter title='Life Appears'>
        <synopsis>Move over dust, here comes life.</synopsis>
        <para>Happily, the dust got a companion: life...</para>
      </chapter>
   </content>
</book>
```

次のコード (抜粋) は、<content> 開始要素タグをスキップし、サブストリームを取得し、独立した ComplexParse オブジェクトを使用してそのサブストリームを解析する方法を示しています。

```
if (stream.skip( ElementFactory.createXMLName("content")))
  {
     ComplexParse complexParse = new ComplexParse();
     complexParse.parse(stream.getSubStream());
  }
```

前の XML ドキュメントでこのメソッドを呼び出すと、以下の出力が得られます。

```
<content>
      <chapter title='Just a Speck of Dust'>
        <synopsis>The world as a speck of dust</synopsis>
        <para>Once the world was just a speck of dust...</para>
      </chapter>
      <chapter title='Life Appears'>
        <synopsis>Move over dust, here comes life.</synopsis>
        <para>Happily, the dust got a companion: life...</para>
      </chapter>
</content>
```

バッファされた XML 入力ストリームのマーキングとリセット

BufferedXMLInputStream オブジェクトを使用する場合、mark() メソッドと reset() メソッドにより、ストリームの特定のスポットをマークし、ストリームの処理を続けた後、マークしたスポットに戻るようにストリームをリセットできます。この 2 つのメソッドは、最初に繰り返し処理をした後で、ストリームをさらに操作したい場合に便利です。

注意 : バッファされたストリームをマーキングせずに読み取った場合、直前に読み取ったものにアクセスすることはできません。つまり、ストリームはバッファされるだけであり、自動的にストリームを再読み取りできることを意味するわけではありません。まずストリームをマークしておく必要があります。

次の例で、BufferedXMLInputStream オブジェクトの一般的な使い方を示します。

```
XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
BufferedXMLInputStream bufstream =
factory.newBufferedInputStream(factory.newInputStream(new
            FileInputStream(name)));

// ストリームの開始をマーク
bufstream.mark();

// ストリームをローカルに処理
bufferedParse.parse(bufstream);

// ストリームをマークにリセット
bufstream.reset();

// 別のアプリケーションにストリームを送信
ComplexParse complexParse =  new ComplexParse();
complexParse.parse(bufstream);
```

入力ストリームのクローズ

プログラミング慣行として、XML 入力ストリームでの作業終了時に入力ストリームを明示的にクローズします。入力ストリームをクローズするには、`XMLInputStream.close()` メソッドを使用します。以下に例を示します。

```
// 入力ストリームのクローズ
input.close();
```

新しい XML ドキュメントの生成：一般的手順

次の手順では、WebLogic XML Streaming API により新しい XML ドキュメントを生成する一般的手順について説明します。

最初の 2 つの手順は必須です。その後の手順は、XML ファイルをどのように生成するかに応じて実行します。

1. `weblogic.xml.stream.*` クラスをインポートします。

2. XML ドキュメントを書き込むための XML 出力ストリームを作成します。4-25 ページの「XML 出力ストリームの作成」を参照してください。
3. XML 出力ストリームにイベントを追加します。4-26 ページの「出力ストリームへの要素の追加」を参照してください。
4. XML 出力ストリームに属性を追加します。4-26 ページの「出力ストリームの要素への属性の追加」を参照してください。
5. 出力ストリームに入力ストリームを追加します。4-27 ページの「出力ストリームへの入力ストリームの追加」を参照してください。
6. 出力ストリームを出力します。4-28 ページの「出力ストリームの出力」を参照してください。
7. 出力ストリームをクローズします。4-29 ページの「出力ストリームのクローズ」を参照してください。

XML ドキュメント生成の例

以下の節では、XML Streaming API により XML ドキュメントを生成する例を示します。

このプログラムでは、まず、`PrintWriter` オブジェクトに基づいて出力ストリームを作成し、次に、その出力ストリームに要素を追加して、プログラムのコメントに説明されているように、単純な XML 発注書を作成します。また、別の XML ファイルに基づく入力ストリームを出力ストリームに追加する方法も示します。

注意： サンプルの後の節に、さらに詳しい説明があります。

```
package examples.xml.stream;

import weblogic.xml.stream.XMLInputStream;
import weblogic.xml.stream.XMLOutputStream;
import weblogic.xml.stream.XMLInputStreamFactory;
import weblogic.xml.stream.XMLName;
import weblogic.xml.stream.XMLEvent;
import weblogic.xml.stream.StartElement;
import weblogic.xml.stream.EndElement;
import weblogic.xml.stream.Attribute;
import weblogic.xml.stream.ElementFactory;
import weblogic.xml.stream.XMLStreamException;
```

```
import weblogic.xml.stream.XMLOutputStreamFactory;


import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.PrintWriter;

/**
 * 次のような非常に単純な発注書を
 * プリントアウトするプログラム
 *
 * <purchase_order>
 *  <name>Juliet Shackell</name>
 *  <item id="1234" quantity="2">Fabulous Chair</item>
 *  <!-- this is a comment-->
 *  <another_file>
 *    This comes from another file called "another_file.xml"
 *  </another_file>
 * </purchase_order>
 *
 * 上の XML ファイルで、<another_file> 要素は、実際には
 * 別の XML ファイルであり、そのファイルからプログラムに引数が渡され、
 * XMLInputStream に変換された後、出力ストリームに追加される
 */
public class PrintPurchaseOrder {

  /**
   * ストリームでハンドルを取得するためのヘルパー メソッド
   * 名前で取得し、ストリームを戻す
   * このメソッドは、InputStreamFactory を使用して、
   * XMLInputStream のインスタンスを作成する
   * @param name 解析するファイル
   * @return XMLInputStream 解析するストリーム
   */
  public XMLInputStream getInputStream(String name)
    throws XMLStreamException, FileNotFoundException
  {
    XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
    XMLInputStream stream = factory.newInputStream(new FileInputStream(name));
    return stream;
  }
  public static void main(String args[])
    throws Exception
  {
    PrintPurchaseOrder printer = new PrintPurchaseOrder();
    //
    // 出力ストリームを作成する
    //
```

```
    XMLOutputStreamFactory factory = XMLOutputStreamFactory.newInstance();
    XMLOutputStream output = factory.newOutputStream(new
                                      PrintWriter(System.out,true));

   // <purchase_order> ルート要素を追加する
   output.add(ElementFactory.createStartElement("purchase_order"));
   output.add(ElementFactory.createCharacterData("\n"));

   // <name> 要素を追加する
   output.add(ElementFactory.createStartElement("name"));
   output.add(ElementFactory.createCharacterData("Juliet Shackell"));
   output.add(ElementFactory.createEndElement("name"));
   output.add(ElementFactory.createCharacterData("\n"));

   // ID 属性および数量属性の入った <item> 要素を追加する
   output.add(ElementFactory.createStartElement("item"));
   output.add(ElementFactory.createAttribute("id","1234"));
   output.add(ElementFactory.createAttribute("quantity","2"));
   output.add(ElementFactory.createCharacterData("Fabulous Chair"));
   output.add(ElementFactory.createEndElement("item"));
   output.add(ElementFactory.createCharacterData("\n"));

    // コメントを追加する
   output.add("<!-- this is a comment-->");
   output.add(ElementFactory.createCharacterData("\n"));

   // 各 XML ファイル引数から入力ストリームを作成し、それを出力に追加する
   for (int i=0; i < args.length; i++)
   //
   // 入力ストリームを取得し、それを出力ストリームに追加する
   //
   output.add(printer.getInputStream(args[i]));

    // 最後に、"purchase_order" ルート要素を終了する
   output.add(ElementFactory.createEndElement("purchase_order"));
   output.add(ElementFactory.createCharacterData("\n"));

    //
    // 結果を画面に出力する
    //
    output.flush();

   // 出力ストリームをクローズする
   output.close();
  }
}
```

上のプログラムは、次の出力を生成します。

```
<purchase_order>
  <name>Juliet Shackell</name>
  <item id="1234" quantity="2">Fabulous Chair</item>
  <!-- this is a comment-->
  <another_file>
     This is from another file.
  </another_file>
</purchase_order>
```

XML 出力ストリームの作成

WebLogic XML Streaming API により XML ドキュメントを生成する最初の手順の 1 つは、構築されるドキュメントを入れるための出力ストリームを作成することです。XML 出力ストリームの作成方法は、入力ストリームの作成方法と似ています。まず、`XMLOutputStreamFactory` のインスタンスを作成し、次に `XMLOutputStreamFactory.newOutputStream()` メソッドで出力ストリームを作成します。以下に例を示します。

```
      XMLOutputStreamFactory factory = XMLOutputStreamFactory.newInstance();
XMLOutputStream output = factory.newOutputStream(new
                          PrintWriter(System.out,true));
```

次の例では、DOM ツリーに基づいて `XMLOutputStream` を作成する方法を示します。

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setValidating(false);
dbf.setNamespaceAware(true);
Document doc = dbf.newDocumentBuilder().newDocument();
XMLOutputStream out =
     XMLOutputStreamFactory.newInstance().newOutputStream(doc);
```

`XMLOutputStreamFactory.newOutputStream()` メソッドを使用して、以下の 4 つの Java オブジェクトに基づく出力ストリームを作成できます。オブジェクトは、XML ドキュメントの最終形式を何にするか (オペレーティング システム上のファイル、DOM ツリーなど) に応じて決定します。

- `java.io.OutputStream`
- `java.io.Writer`
- `org.xml.sax.ContentHandler`
- `org.w3c.dom.Document`

出力ストリームへの要素の追加

出力ストリームに要素を追加するには、XMLOutputStream.add(XMLEvent) メソッドを使用します。特定の要素を作成するには、ElementFactory を使用します。

ElementFactory インタフェースには、各タイプの要素を作成するためのメソッドが入っています。一般的フォーマットは ElementFactory.create*XXX*() です。*XXX* は、createStartElement()、createCharacterData() などの特定の要素を表わします。String または XMLName のような名前を渡すことにより、大部分の要素を作成できます。

警告： XMLOutputStream は XML を検証しません。

注意： 開始要素を作成するたびに、終了要素もどこかに明示的に作成する必要があります。同じ規則が開始ドキュメントの作成にも適用されます。

たとえば、次のような小さな XML を作成するとします。

```
<name>Georgina Shackell Green</name>
```

出力ストリームにこの要素を追加する Java コードは次のようになります。

```
output.add(ElementFactory.createStartElement("name"));
output.add(ElementFactory.createCharacterData("Georgina Shackell Green"));
output.add(ElementFactory.createEndElement("name"));
output.add(ElementFactory.createCharacterData("\n"));
```

最後の createCharacterData() メソッドは、出力ストリームに改行文字を追加します。この指定は必須ではありませんが、読みやすい XML を作成するときに便利です。

出力ストリームの要素への属性の追加

作成した要素に属性を追加するには、XMLOutputStream.add(Attribute) を使用します。特定の属性を作成するには、ElementFactory.createAttribute() メソッドを使用します。

たとえば、次のような小さな XML を作成するとします。

```
<item id="1234" quantity="2">Fabulous Chair</item>
```

出力ストリームにこの要素を追加する Java コードは次のようになります。

```
output.add(ElementFactory.createStartElement("item"));
output.add(ElementFactory.createAttribute("id","1234"));
output.add(ElementFactory.createAttribute("quantity","2"));
output.add(ElementFactory.createCharacterData("Fabulous Chair"));
output.add(ElementFactory.createEndElement("item"));
output.add(ElementFactory.createCharacterData("\n"));
```

注意： 要素への属性の追加は、開始要素を作成した後、対応する終了要素の作成より前に行ってください。そうでない場合、コードは正常にコンパイルされますが、プログラム実行時にエラーが発生します。たとえば、次のコードでは、`<item>` 要素が明示的に終了した後に、この要素に属性が追加されているので、エラーが返されます。

```
output.add(ElementFactory.createStartElement("item"));
output.add(ElementFactory.createEndElement("item"));
output.add(ElementFactory.createAttribute("id","1234"));
output.add(ElementFactory.createAttribute("quantity","2"));
output.add(ElementFactory.createCharacterData("Fabulous Chair"));
output.add(ElementFactory.createCharacterData("\n"));
```

出力ストリームへの入力ストリームの追加

XML 出力ストリームを作成する際、XML ファイルや DOM ツリーのような既存の XML ドキュメントを出力ストリームに追加したいことがあります。これを行うには、まず、その XML ドキュメントを XML 入力ストリームに変換してから、`XMLOutputStream.add(XMLInputStream)` メソッドを使用して出力ストリームに入力ストリームを追加します。

次の例では、まず、XML ファイルから XML 入力ストリームを作成する getInputStream() というメソッドを示し、その後、このメソッドを使用して作成された入力ストリームを出力プログラムに追加する方法を示します。

```
/**
 * ストリームでハンドルを取得するためのヘルパー メソッド
 * 名前で取得し、ストリームを戻す
 * このメソッドは、InputStreamFactory を使用して、
 * XMLInputStream のインスタンスを作成する
```

```
   * @param name 解析するファイル
   * @return XMLInputStream 解析するストリーム
   */

  public XMLInputStream getInputStream(String name)
    throws XMLStreamException, FileNotFoundException
  {
    XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
    XMLInputStream stream = factory.newInputStream(new FileInputStream(name));
    return stream;
  }

....

   // 各 XML ファイル引数から入力ストリームを作成し、それを出力に追加する

   for (int i=0; i < args.length; i++)
   //
   // 入力ストリームを取得し、それを出力ストリームに追加する
   //
   output.add(printer.getInputStream(args[i]));
```

出力ストリームの出力

オブジェクトから作成した XML 出力ストリームをプリントアウトするには、`XMLOutputStream.flush()` メソッドを使用します。たとえば、`PrintWriter` オブジェクトから XML 出力ストリームを作成した場合、そのストリームは、`flush()` メソッドにより標準出力に出力されます。

注意： DOM ツリーに基づいて XMLOutputStream に書き込む場合、まず `flush()` メソッドを実行しなければ、DOM を操作することはできません。

次の例では、出力ストリームを出力する方法を示します。

```
//
// 結果を画面に出力する
//
output.flush();
```

出力ストリームのクローズ

プログラミング慣行として、XML 出力ストリームでの作業終了時に入力ストリームを明示的にクローズします。出力ストリームをクローズするには、`XMLOutputStream.close()` メソッドを使用します。以下に例を示します。

```
// 出力ストリームをクローズする
output.close();
```

5 XML プログラミングのベストプラクティス

以下の節では、XML データを処理する Java アプリケーションを作成する場合のプログラミング上の最良の方法について説明します。

- DOM、SAX、Streaming API を使用する場合
- XML 検証のパフォーマンスの改善
- XML スキーマまたは DTD を使用する場合
- パフォーマンスを最大にする外部エンティティ解決のコンフィグレーション
- SAX InputSource の使用
- 変換のパフォーマンスの向上

DOM、SAX、Streaming API を使用する場合

DOM、SAX、Streaming API により XML ドキュメントを解析できます。この節では、各 API の長所および短所について説明します。

DOM API は、小さなドキュメント、すなわち要素数が 1000 個以下のドキュメントに向いています。DOM は XML データのツリーを構成するので、要素の追加または削除により XML ドキュメントの構造を編集するには理想的です。

DOM API では、処理を開始する前に、まず、XML ドキュメント全体を解析し、DOM ツリーに変換します。このコストは、ドキュメント全体にアクセスする必要のあることが分かっている場合には有意義です。XML ドキュメントの一部に

しかアクセスする必要のない場合には、アプリケーションのパフォーマンスが低下するだけで、何のメリットもないことがあります。そのような場合は、SAX または Streaming API が適しています。

SAX API は最も軽い API です。ユニークな要素名の入った浅いドキュメント (要素のネストがあまりないドキュメント) の解析に理想的です。SAX では、コールバック構造が使用されます。つまり、API としてイベントの解析を処理するプログラマが XML ドキュメントを読み取ります。これは、比較的効率的で迅速な解析方法です。

ただし、SAX のコールバックの性質上、XML データの構造を変更する場合に必ずしも最良の API が使用されるとは限りません。また、コールバックを処理するアプリケーションのプログラミングは、分かりやすい直感的なものではありません。

Streaming API は SAX をベースにしているので、SAX を使用する理由はすべて Streaming API にもあてはまります。さらに、プログラマはイベント発生時にイベントに対応するのではなく、イベントを請求するので、Streaming API の方が SAX を使用するより直感的です。Streaming API は、パラメータとして XML ドキュメント全体を渡す場合にも最適です。入力ストリームを渡す方が、SAX イベントを渡すよりも簡単だからです。最後に、Streaming API は、もともと XML データを Java オブジェクトに バインドするために設計されたものです。

XML 検証のパフォーマンスの改善

パーサ検証問題により XML アプリケーションのパフォーマンスが低下していても、XML ドキュメントの検証を行う必要がある場合、`DocumentBuilderFactory` または `SaxParserFactory` の `setValidating()` メソッドを使用するのではなく、データを受信または解析するときに検証する独自のカスタム コードを書くことにより、アプリケーションのパフォーマンスを改善できる場合があります。

SAX または DOM で XML ドキュメント解析中の検証をオンにした場合、パーサはユーザが実際に必要とするより多くの検証を行うことがあり、その結果、アプリケーションの全体的パフォーマンスが低下します。XML ドキュメントが有効であるかをチェックする場合は、ドキュメントの解析中に適切なポイントを選択し、それらのポイントに独自の Java コードを追加することを検討してください。

たとえば、WebLogic XML Streaming API により XML 発注書を処理するアプリケーションを書くと仮定します。ドキュメントの最初の要素は `<purchase_order>` でなければならないことが分かっているので、ストリームから最初の要素を取り出し、その名前をチェックすることにより、ドキュメントが有効かを迅速に検証できます。もちろん、このチェックは XML ドキュメント全体が有効であることを保証するものではありませんが、ストリームから要素を取り出しながら、既知の要素についてさらにチェックを続けることができます。このようなクイック チェックは、標準の `setValidating()` メソッドを使用する場合よりはるかに高速です。

XML スキーマまたは DTD を使用する場合

XML ドキュメントの構造の記述方法には、DTD および XML スキーマの 2 種類があります。

最近は、スキーマにより XML ドキュメントを記述する傾向にあります。スキーマは、XML 要素を記述するために利用できるデータ タイプのセットが DTD よりはるかに豊富で、DTD より表現力があり、XML ドキュメントで何が有効であるかを詳しく記述できます。また、SOAP メッセージではスキーマのみを使用でき、DTD は使用できません。SOAP は Web サービスで使用される主要なメッセージング プロトコルであり、Web サービスに対する入力パラメータまたは出力パラメータとして使用される XML ドキュメントを記述するには、スキーマを使用することを検討してください。

DTD にもいくつかの利点があります。DTD は、急速に変化しつつあるスキーマより広い範囲でサポートされています。DTD はスキーマほどの表現力がないので、作成や管理が簡単です。

BEA Systems では、XML ドキュメントの記述にはスキーマを使用することをお勧めします。

パフォーマンスを最大にする外部エンティティ解決のコンフィグレーション

外部エンティティは、いつもネットワークから取得するのではなく、可能な限りローカルに格納しておくことをお勧めします。エンティティをネットワーク接続を介して探すよりも、WebLogic Server と同じマシン上で探す方がはるかに高速であるため、ローカルに格納することでアプリケーションのパフォーマンスが向上します。

WebLogic Server のために外部エンティティ解決をコンフィグレーションする方法の詳細については、7-12 ページの「外部エンティティのコンフィグレーション タスク」を参照してください。

SAX InputSource の使用

SAX API により XML ドキュメントを解析する場合、まず XML ドキュメントから `InputSource` オブジェクトを作成し、その `InputSource` オブジェクトを `parse()` メソッドに渡します。`InputSource` オブジェクトは、XML データに基づいて、`java.io.InputStream` オブジェクトまたは `java.io.Reader` オブジェクトから作成できます。

可能な限り、`java.io.InputStream` オブジェクトから `InputSource` を作成することをお勧めします。`InputStream` オブジェクトが渡されると、SAX パーサは、XML データの文字エンコーディングを自動検出し、正しい文字エンコーディングを使用して、`InputStreamReader` オブジェクトを自動的にインスタンス化します。つまり、ユーザに代わってパーサが文字エンコーディングのすべての作業を行うので、文字エンコーディングをユーザ自身が指定する場合より、実行時のエラーが大幅に減少します。

変換のパフォーマンスの向上

XSLT は、XML ドキュメントを、別の XML ドキュメント、HTML、WML など、異なるフォーマットに変換するための言語です。XSLT を使用するには、入力 XML ドキュメントの各要素を出力ドキュメントでどのように変換するかを定義するスタイルシートを作成します。

XSLT は強力な言語ですが、複雑な変換のためのスタイルシートの作成は非常に煩雑になることがあります。また、実際の変換には大量のリソースが必要であり、アプリケーションのパフォーマンスが低下することがあります。

したがって、変換が複雑な場合は、XSLT スタイルシートを使用するのではなく、アプリケーションで独自の変換コードを書くことを検討してください。また、DOM API の使用も検討してください。まず、XML ドキュメントを解析し、結果の DOM ツリーを必要に応じて操作し、それを最終フォーマットに変換するための カスタム Java コードを使用して、新しいドキュメントを書き出します。

6　XML プログラミング手法

以下の節では、XML データを処理する J2EE アプリケーションを開発するための特定の XML プログラミング手法について説明します。

- Java クライアントと WebLogic Server との間での XML データの転送
- JMS アプリケーションでの XML ドキュメントの処理
- HTTP インタフェースを持たない外部エンティティへのアクセス
- XML ドキュメント ヘッダ情報の取得

Java クライアントと WebLogic Server との間での XML データの転送

一般的な J2EE アプリケーションでは、クライアント アプリケーションは、XML データを処理するサーブレットまたは JSP に XML データを送信します。次に、サーブレットまたは JSP は、そのデータを JMS 送り先や EJB などの別の J2EE コンポーネントに送信するか、または処理した XML データを別の XML ドキュメントの形でクライアントに返します。

Java クライアントから WebLogic Server にホストされたサーブレットまたは JSP に XML データを送信し、クライアントにデータが返されるようにするには、`java.net.URLConnection` クラスを使用します。このクラスは、アプリケーションと URL (この場合は、サーブレットまたは JSP を呼び出す URL) の間の通信リンクを表します。`URLConnection` クラスのインスタンスは、HTTP POST メソッドを使用して XML ドキュメントを送信します。

WebLogic XML サンプルから抜粋した次の Java クライアント プログラムは、プログラムと JSP の間で XML データを転送する方法を示しています。

```
import java.net.*;
import java.io.*;
import java.util.*;

public class Client {

  public static void main(String[] args) throws Exception {
    if (args.length < 2) {
      System.out.println("Usage:  java examples.xml.Client URL Filename");
    }
    else {
      try {
        URL url = new URL(args[0]);
        String document = args[1];
        FileReader fr = new FileReader(document);
        char[] buffer = new char[1024*10];
        int bytes_read = 0;
        if ((bytes_read = fr.read(buffer)) != -1)
          {
            URLConnection urlc = url.openConnection();
            urlc.setRequestProperty("Content-Type","text/xml");
            urlc.setDoOutput(true);
            urlc.setDoInput(true);
            PrintWriter pw = new PrintWriter(urlc.getOutputStream());

            // XML を JSP に送信
            pw.write(buffer, 0, bytes_read);
            pw.close();

            BufferedReader in = new BufferedReader(new
InputStreamReader(urlc.getInputStream()));
            String inputLine;
            while ((inputLine = in.readLine()) != null)
                      System.out.println(inputLine);

            in.close();
            }
          }
          catch (Exception e) {
          e.printStackTrace();
          }
       }
    }
}
```

この例では、引数リストからの URL を使用して JSP への URL 接続をオープンし、その接続から出力ストリームを取得し、引数リストに指定された XML ドキュメントを出力ストリームに出力して、XML データを JSP に送信する方法を

示しています。次に、URLConnection クラスの getInputStream() メソッドで、JSAP がクライアント アプリケーションに返す XML データを読み取る方法を示します。

サンプル JSP から抜粋した次のコード セグメントでは、JSP がクライアント アプリケーションから XML データを受信し、XML ドキュメントを解析して、XML データを返す方法を示しています。

```
BufferedReader br = new BufferedReader(request.getReader());
DocumentBuilderFactory fact = DocumentBuilderFactory.newInstance();
DocumentBuilder db = fact.newDocumentBuilder();
Document doc = db.parse(new InputSource(br));

...

PrintWriter responseWriter = response.getWriter();
responseWriter.println("<?xml version='1.0'?>");
```

...
WebLogic サーブレットと JSP アプリケーションのプログラミングの詳細については、『WebLogic HTTP サーブレット プログラマーズ ガイド』と『WebLogic JSP プログラマーズ ガイド』を参照してください。

JMS アプリケーションでの XML ドキュメントの処理

WebLogic Server は、JMS アプリケーションで XML ドキュメントを処理するために一部の Java Message Service (JMS) クラスに対して次の拡張を提供しています。

- weblogic.jms.extensions.WLSession。JMS クラス javax.jms.Session を拡張します。
- weblogic.jms.extensions.WLQueueSession。JMS クラス javax.jms.QueueSession を拡張します。
- weblogic.jms.extensions.WLTopicSession。JMS クラス javax.jms.TopicSession を拡張します。

- `weblogic.jms.extensions.XMLMessage`。JMS クラス `javax.jms.XMLMessage` を拡張します。

より一般的な `TextMessage` クラスではなく、`XMLMessage` クラスを使用して、JMS アプリケーションで XML ドキュメントを送受信する場合、XML 固有のメッセージ セレクタを使用して不要なメッセージをフィルタ処理できます。特に、`JMS_BEA_SELECT` メソッドを使用すると、XML ドキュメントの XML フラグメントを検索するための XPath クエリを指定できます。クエリの結果に基づいて、メッセージ コンシューマがメッセージを受信しないよう決定することもあります。これにより、ネットワーク トラフィックを軽減し、JMS アプリケーションのパフォーマンスを向上させることができます。

`XMLMessage` クラスを使用して JMS アプリケーションに XML メッセージを含めるには、`WLQueueSession` オブジェクトまたは `WLTopicSession` オブジェクトのいずれかを作成する必要があります。どちらのオブジェクトを作成するかは、JMS `Connection` を作成した後に、汎用オブジェクト `QueueSession` または `TopicSession` ではなく、JMS キューまたはトピックを使用するかどうかによります。次に、`WLSession` インタフェースの `createXMLMessage()` メソッドを使用して `XMLMessage` オブジェクトを使用します。

JMS アプリケーションで `XMLMessage` オブジェクトを使用する際の詳細については、『WebLogic JMS プログラマーズ ガイド』を参照してください。

HTTP インタフェースを持たない外部エンティティへのアクセス

WebLogic Server は、外部レポジトリにおかれた外部エンティティに、エンティティを返す URL などの HTTP インタフェースがある場合に限り、その外部エンティティを自動的に取得およびキャッシュできます。XML レジストリで外部エンティティをコンフィグレーションする際の詳細については、7-12 ページの「外部エンティティのコンフィグレーション タスク」を参照してください。

HTTP インタフェースを持たないレポジトリに格納されている外部エンティティにアクセスする場合は、インタフェースを作成する必要があります。たとえば、システム ID、パブリック ID、DTD のテキストの列を持つデータベース テーブ

ルに XML ドキュメント用の DTD を格納しているとします。WebLogic XML アプリケーションから外部エンティティとして DTD にアクセスする場合、JDBC を使用してデータベースの DTD にアクセスするサーブレットを作成できます。

URL でサーブレットを呼び出すので、ここで外部エンティティに対する HTTP インタフェースを持っていることになります。XML レジストリでエンティティ レジストリ エントリを作成する場合、外部エンティティの場所としてサーブレットを呼び出す URL を指定します。WebLogic Server は、この外部エンティティへの参照を含む XML ドキュメントを解析しているときにサーブレットを呼び出し、サーブレットがデータベースのクエリに内部で使用するパブリック ID およびシステム ID をサーブレットに渡します。

XML ドキュメント ヘッダ情報の取得

ドキュメント内の実際のデータをすべて取得するのではなく、ルート要素、システム ID、パブリック ID などの XML ドキュメントに関するヘッダ情報のみ必要な場合を考えてみましょう。ドキュメントの詳細な解析は不要で、XML ドキュメントのサイズが非常に大きい場合は、アプリケーションのパフォーマンスが低下する可能性があります。

XML ドキュメントを解析する代わりに、`org.xml.sax.InputSource` クラスに対する WebLogic Server の拡張である `weblogic.xml.sax.XMLInputSource` クラスを使用すると、XML ドキュメントに関するヘッダー情報を取得できます。次のサンプル コード セグメントでは、このクラスの使い方を示します。

```
import weblogic.xml.sax.XMLInputSource;
...
    String inputXML = “file://xml_docs/myXMLdoc.xml”;
    XMLInputSource xis = new XMLInputSource(inputXML);
    String docType = xis.getRootTag();
    String publicID = xis.getPublicId();
    String systemID = xis.getSystemId();
    String namespaceURI = xis.getNamespaceURI();
```

`weblogic.xml.sax.XMLInputSource` クラスの詳細については、『WebLogic Server API リファレンス』を参照してください。

7 WebLogic Server XML の管理

以下の節では、WebLogic Server での XML の管理について説明します。

- WebLogic Server XML の管理の概要
- XML パーサおよびトランスフォーマのコンフィグレーション タスク
- 外部エンティティのコンフィグレーション タスク

WebLogic Server XML の管理の概要

Administration Console から XML レジストリにアクセスし、XML レジストリで WebLogic Server を XML アプリケーション用にコンフィグレーションします。

ブラウザで Administration Console を起動するには、次の URL を入力します。

```
http://host:port/console
```

各要素の説明は次のとおりです。

- *host* は、WebLogic 管理サーバが動作しているコンピュータの名前です。
- *port* は、WebLogic 管理サーバが接続リクエストのリスニングを行っているポート番号です。WebLogic 管理サーバのデフォルトのポート番号は 7001 です。

XML の管理タスク

XML レジストリの作成、コンフィグレーション、使用は、Administration Console を通じて行います。Administration Console XML レジストリの使用には、次のメリットがあります。

- XML アプリケーションで JAXP を使用している場合、XML レジストリのコンフィグレーションの変更が実行時に自動的に有効になります。
- XML レジストリを変更する場合、XML アプリケーション コードを変更する必要はありません。
- エンティティの解決がローカルで実行されます。XML レジストリでは、エンティティのローカル コピーを定義できます。または、WebLogic Server が Web からエンティティを指定した期間キャッシュし、Web 上のエンティティではなく、そのキャッシュしたコピーを使用するように指定することもできます。

XML レジストリを使用すると、以下の指定が可能になります。

- 組み込みパーサの代わりとなるサーバ全体の XML パーサ。
- ドキュメント タイプ単位の XML パーサ。
- 組み込みトランスフォーマの代わりとなるサーバ全体のトランスフォーマ。
- エンティティのローカル コピーで解決される外部エンティティ。外部エンティティを指定すると、管理サーバでは、エンティティのローカル コピーがファイル システムに保存され、解析時にサーバのパーサに自動的に配布されます。この機能を使用すれば、SAX EntityResolvers の作成と設定が不要になります。
- Web からの取得後に WebLogic Server によってキャッシュされる外部エンティティ。WebLogic Server が再取得するまでこれらの外部エンティティがキャッシュされる期間、および WebLogic が最初にエンティティを取得するタイミング (アプリケーションの実行時か、WebLogic Server の起動時か) を指定します。

これらの機能は、サーバサイドだけで使用できます。

XML レジストリの仕組み

XML レジストリは必要な数だけ作成できますが、WebLogic Server の特定のインスタンスに関連付けることができる XML レジストリの数は 1 つだけです。

WebLogic Server のインスタンスに関連付けられている XML レジストリがない場合、ドキュメントの解析および変換に組み込みパーサおよびトランスフォーマが使用されます。

XML レジストリを WebLogic Server のインスタンスに関連付けると、すべての XML コンフィグレーション オプションが、そのサーバを使用している XML アプリケーションで利用可能になります。

XML レジストリに対しては、以下のタイプのエントリを作成できます。

- パーサおよびトランスフォーマをコンフィグレーション
- 外部エンティティの解決のコンフィグレーション

注意： XML レジストリでは、大文字と小文字が区別されます。たとえば、ルート要素が `<CAR>` の XML ドキュメント タイプのパーサをコンフィグレーションしている場合、[ルート要素タグ] フィールドには、「`car`」または「`Car`」ではなく「`CAR`」と入力する必要があります。

XML レジストリ内のパーサの選択

JAXP で XML アプリケーションを記述する場合は、常に XML レジストリに自動的にアクセスします。WebLogic Server は、以下のようなルックアップ順序に従って、ロードするパーサのクラスを決定します。

1. 特定のドキュメント タイプ用に定義したパーサを使用します。
2. WebLogic Server インスタンスに関連付けられている XML レジストリで定義されたサーバ全体の代替パーサを使用します。
3. 組み込み Xerces パーサを使用します。

特定のドキュメント タイプに対応するトランスフォーマを定義することはできないので最初の手順は除きますが、それ以外ではこのプロセスはトランスフォーマにも当てはまります。

さらに、WebLogic Server の起動時に、SAX エンティティ リゾルバは、レジストリで宣言されたエンティティを解決するように自動的に設定されます。したがって、使用するパーサを制御したり、外部エンティティのローカル コピーの場所を設定したりするために、XML アプリケーション コードを変更する必要はありません。使用するパーサ、および外部エンティティの位置は、XML レジストリで制御します。

注意： パーサによって JAXP の代わりに提供された API を使用する場合、XML レジストリは XML ドキュメントの処理に影響を与えません。このため、XML アプリケーションではなるべく JAXP を使用してください。

XML パーサおよびトランスフォーマのコンフィグレーション タスク

デフォルトでは、WebLogic Server は、組み込みパーサとトランスフォーマで XML ドキュメントを解析および変換するようにコンフィグレーションされています。リリース 7.0 では、組み込み XML パーサは Apache Xereces、組み込みトランスフォーマは Apache Xalan です。デフォルト コンフィグレーションを使用する場合、XML アプリケーション用のコンフィグレーション タスクは不要です。組み込み以外のパーサまたはトランスフォーマを使用する場合は、以下の節で説明するように XML レジストリでパーサおよびトランスフォーマをコンフィグレーションする必要があります。

組み込み以外のパーサまたはトランスフォーマのコンフィグレーション

以下の手順ではまず、SAX および DOM パーサおよびトランスフォーマを定義する XML レジストリの作成方法を説明します。次に、サーバが新しいパーサとトランスフォーマを使用するために、新しい XML レジストリを WebLogic Server のインスタンスに関連付ける方法を説明します。

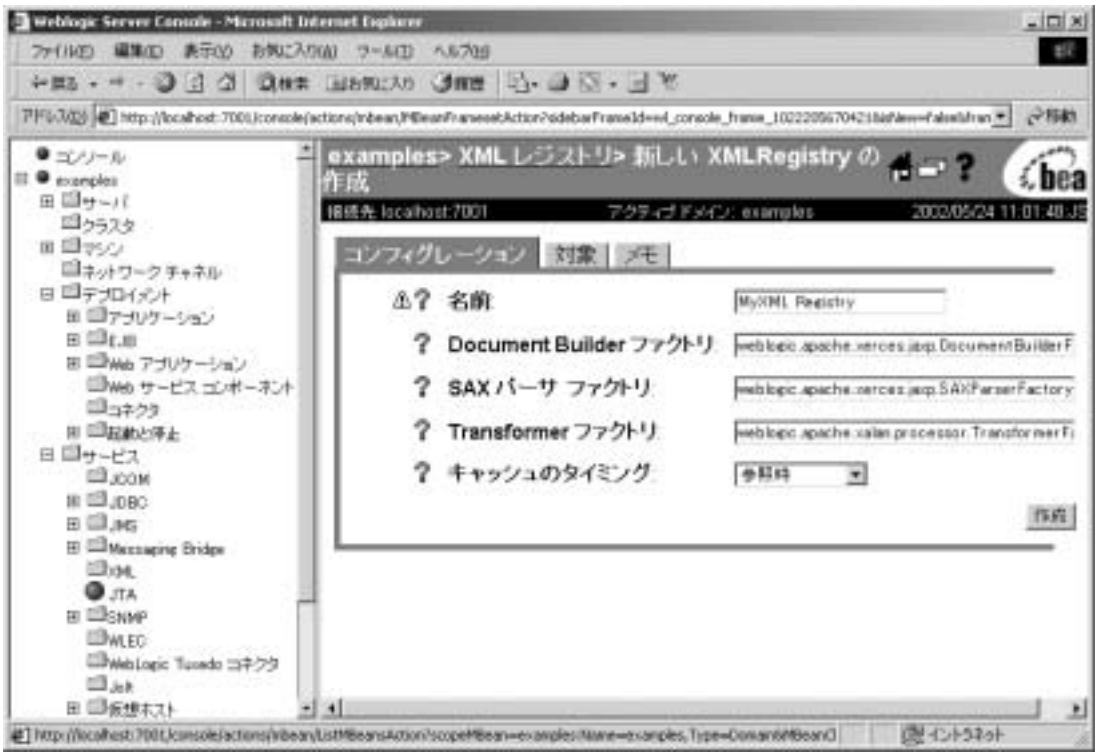
警告： WebLogic Server バージョン 7.0 では、Apache Xerces パーサの次のバージョンのみをプラグインできます。

- Xerces 1.2.2
- Xerces 1.2.3
- Xerces 1.3.0
- Xerces 1.3.1
- Xerces 1.4.0
- Xerces 1.4.1
- Xerces 1.4.2
- Xerces 1.4.3
- Xerces 1.4.4

また、Apache Xerces パーサの以前のバージョン対応の Apache Xalan トランスフォーマ のバージョンのみをプラグインできます。

1. WebLogic 管理サーバを起動し、Administration Console をブラウザで起動します。Administration Console 開始の詳細については、7-1 ページの「WebLogic Server XML の管理の概要」を参照してください。
2. 左ペインの [サービス] ノードの下にある [XML] ノードを右クリックし、ドロップダウン メニューから [新しい XML Registry のコンフィグレーション] を選択します。新しい XML レジストリ作成用のウィンドウが表示されます。

図 7-1 Administration Console のメイン XML レジストリ ウィンドウ



3. ユニークなレジストリ名を [名前] フィールドに入力し、[Document Builder ファクトリ] フィールド、[SAX パーサ ファクトリ] フィールド、および [Transformer ファクトリ] フィールドに適切なファクトリ パーサおよびトランスフォーマのクラスを設定します。

   たとえば、WebLogic FastParser の場合は、次の情報を入力します。

```
[ 名前 ] : WebLogic FastParser
[Document Builder ファクトリ ] :
[SAX パーサ ファクトリ ]: weblogic.xml.babel.jaxp.SAXParserFactoryImpl
[Transformer ファクトリ ] :
```

   上の例では、[Document Builder ファクトリ] および [Transformer ファクトリ] は空白のまま残されています。これは、DOM 解析および変換では、そ

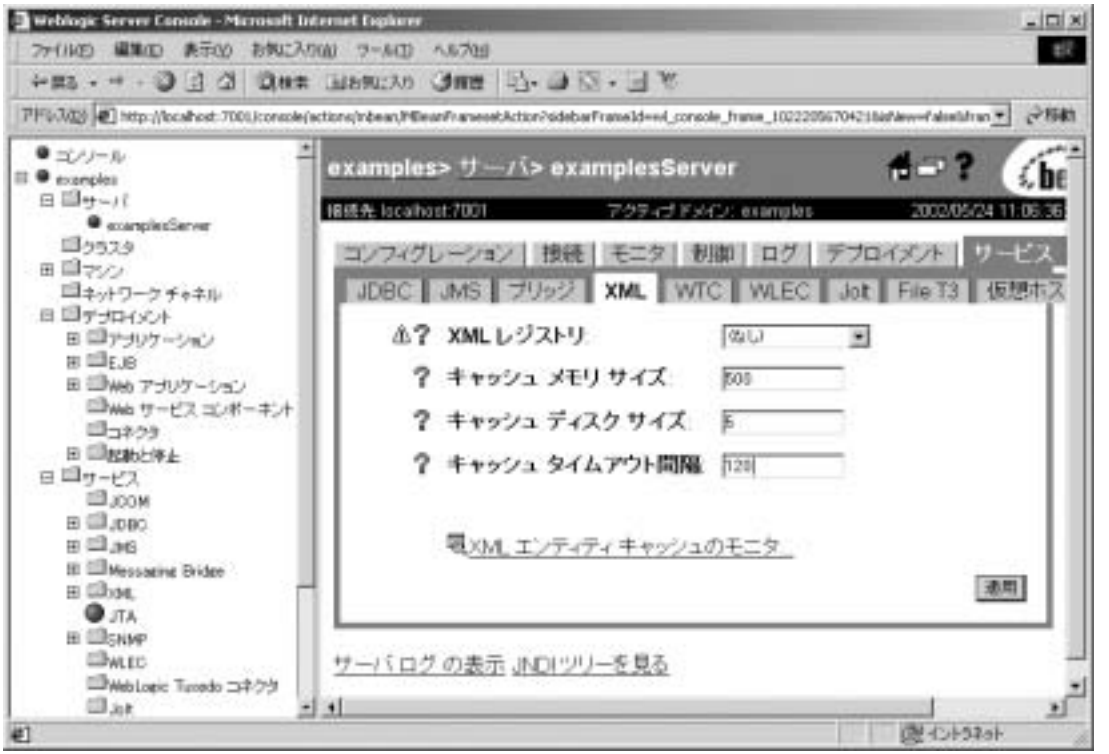
れぞれ組み込みパーサおよびトランスフォーマが使用されることを意味します。WebLogic FastParser は、SAX 解析でのみ使用されます。

Apache Xerces パーサおよび Xalan トランスフォーマを直接指定する場合、以下の情報のいずれかを指定します。

[名前] : Apache Xerces/Xalan Registry
[Document Builder ファクトリ] : org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
[SAX パーサ ファクトリ] : org.apache.xerces.jaxp.SAXParserFactoryImpl
[Transformer ファクトリ] : org.apache.xalan.processor.TransformerFactoryImpl

4. [作成] ボタンをクリックします。左ペインの [XML] ノードの下に、XML レジストリが作成されて表示されます。
5. 左ペインの [サーバ] ノード下で、新しい XML レジストリに関連付けるサーバの名前をクリックします。
6. 右ペインで、[サービス] タブを選択します。
7. [XML] タブを選択します。WebLogic Server の XML プロパティのコンフィグレーション用ウィンドウが右ペインに表示されます。

図 7-2 Administration Console の XML プロパティ コンフィグレーション用ウィンドウ



8. [XML レジストリ] フィールドで、このサーバに関連付ける XML レジストリ名を選択し、[適用] ボタンをクリックします。
9. サーバを再起動して新しい設定内容を有効にします。

特定のドキュメント タイプに対応したパーサのコンフィグレーション

特定のドキュメント タイプに対応したパーサをコンフィグレーションする場合、ドキュメントのシステム ID、パブリック ID、ルート要素タグを使用して、ドキュメント タイプを識別できます。

警告： WebLogic Server では、XML ドキュメントのタイプを識別する際、ドキュメントの先頭から 1000 バイトのみをサーチします。この先頭 1000 バイトで DOCTYPE ID が見つからない場合、ドキュメントのサーチは停止され、WebLogic Server インスタンス用にコンフィグレーションされたパーサを使用してドキュメントの解析が行われます。

注意： 次の手順では、これから新しい XML レジストリを作成して、必要なパーサ レジストリ エントリを追加し、サーバに関連付けることを前提としています。既存の XML レジストリを既にサーバに関連付けている場合は、手順 5. にスキップしてください。

特定のドキュメント タイプに対応したパーサをコンフィグレーションするには、以下の手順を実行します。

1. WebLogic 管理サーバを起動し、Administration Console をブラウザで起動します。

   Administration Console 開始の詳細については、7-1 ページの「WebLogic Server XML の管理の概要」を参照してください。

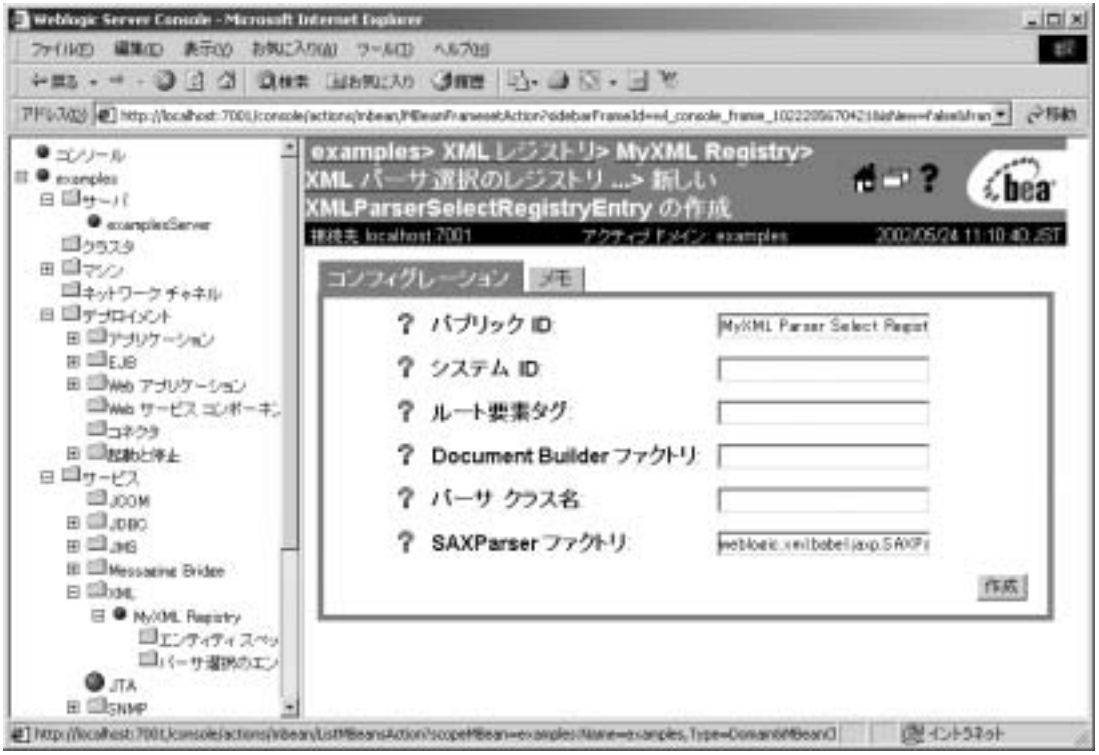
2. 左ペインの [サービス] ノードの下にある [XML] ノードを右クリックし、ドロップダウン メニューから [新しい XML Registry のコンフィグレーション] を選択します。図 7-1 のように、新しい XML レジストリの作成用ウィンドウが表示されます。

3. [名前] フィールドにユニークなレジストリ名を入力します。サーバに対してデフォルト パーサおよびトランスフォーマをコンフィグレーションする場合、[Document Builder ファクトリ]、[SAX パーサ ファクトリ]、および [Transformer ファクトリ] フィールドにファクトリ クラス名を入力します。そうでない場合は、フィールドを空白のままにします。

4. [作成] ボタンをクリックします。左ペインの [XML] ノードの下に、XML レジストリが作成されて表示されます。

5. 左ペインの [XML] ノード下で、XML レジストリの [XML パーサの登録] ノードを右クリックします。ドロップダウン メニューから [新しい XMLParserSelectRegistryEntry のコンフィグレーション] を選択します。ドキュメント タイプ情報入力用の空のウィンドウが右ペインに表示されます。

図 7-3 Administration Console による XML パーサのコンフィグレーション



6. 以下のいずれかの方法でドキュメント タイプ情報を入力します。

   a. [パブリック ID] フィールドまたは [システム ID] フィールドのいずれかに doctype を指定します。たとえば、`car.xml` (コード リスト 7-1 を参照) の場合、[パブリック ID] フィールドに「`-//BEA Systems, Inc.//DTD for cars//EN`」と入力します。

   b. [ルート要素タグ] にドキュメントのルート要素タグ名を指定します。`car.xml` の例では、[ルート要素タグ] フィールドに「`CAR`」と入力します。

      XML ドキュメントでネームスペースを定義する場合、`VEHICLES:CAR` のように完全修飾のルート要素タグを入力してください。

コード リスト 7-1 car.xml ファイル

```
<?xml version="1.0"?>
<!-- 以下の XML ドキュメントでは自動車を説明 -->
<!DOCTYPE CAR PUBLIC "-//BEA Systems, Inc.//DTD for cars//EN"
"http//www.beasys.co.jp/index.html">
<CAR>
<MAKE>Toyota</MAKE>
<MODEL>Corrolla</MODEL>
<YEAR>1998</YEAR>
<ENGINE>1.5L</ENGINE>
<HP>149</HP>
</CAR>
```

7. [Document Builder ファクトリ] または [SAX パーサ ファクトリ] フィールドに、適切なファクトリ パーサ クラスを指定します。

   たとえば、このドキュメント タイプを WebLogic FastParser で解析するように指定する場合は、[SAX パーサ ファクトリ] フィールドに「`weblogic.xml.babel.jaxp.SAXParserFactoryImpl`」と入力します。

   注意： [パーサ クラス名] フィールドに情報を入力しないでください。このフィールドは、旧バージョンの WebLogic Server との下位互換性を保つためのものです。

8. [作成] ボタンをクリックします。XMLParserSelect レジストリ エントリが作成されます。
9. 左ペインの [サーバ] ノード下で、新しい XML レジストリに関連付けるサーバの名前をクリックします。
10. 右ペインで、[サービス] タブを選択します。
11. [XML] タブを選択します。図 7-2 のように、WebLogic Server の XML プロパティのコンフィグレーション用ウィンドウが右ペインに表示されます。
12. [XML レジストリ] フィールドで、このサーバに関連付ける XML レジストリ名を選択し、[適用] ボタンをクリックします。
13. サーバを再起動して新しい設定内容を有効にします。

外部エンティティのコンフィグレーション タスク

XML レジストリを使用すると、外部エンティティ解決をコンフィグレーションし、外部エンティティ キャッシュをコンフィグレーションおよびモニタできます。

外部エンティティの解決のコンフィグレーション

WebLogic Server では、以下のいずれかの方法で外部エンティティの解決をコンフィグレーションできます。

- 物理的にエンティティ ファイルを、WebLogic 管理サーバからアクセス可能なディレクトリにコピーし、外部エンティティが XML ドキュメントで参照されている場合は常に管理サーバがそのローカル コピーを使用するように指定します。
- サーバ起動時、または外部エンティティの初めての参照時のいずれかに、管理サーバを基準にした相対 URL またはパス名で参照される外部エンティティを管理対象の Server がキャッシュするように指定します。

  外部エンティティを管理対象の Server にキャッシュすると、アクセス時間を節約できるだけでなく、ネットワークまたは管理サーバのダウンにより XML ドキュメントの解析中に管理サーバにアクセスできなくなった場合には、ローカル バックアップを利用できます。

  キャッシュされたエンティティに対して、WebLogic Server が URL または管理サーバからエンティティを再取得して再キャッシュするまでの間隔 (有効期限) をコンフィグレーションできます。

注意 : 次の手順では、これから新しい XML レジストリを作成して、必要な外部エンティティ解決のエントリを追加し、サーバに関連付けることを前提としています。既存の XML レジストリを既にサーバに関連付けている場合は、手順 5. にスキップしてください。

外部エンティティの解決をコンフィグレーションするには、以下の手順を実行します。

1. WebLogic 管理サーバを起動し、Administration Console をブラウザで起動します。

   Administration Console 開始の詳細については、7-1 ページの「WebLogic Server XML の管理の概要」を参照してください。

2. 左ペインの [サービス] ノードの下にある [XML] ノードを右クリックし、ドロップダウン メニューから [新しい XML Registry のコンフィグレーション] を選択します。図 7-1 のように、新しい XML レジストリの作成用ウィンドウが表示されます。

3. [名前] フィールドに、ユニークなレジストリ名を入力します。サーバに対してデフォルト パーサおよびトランスフォーマをコンフィグレーションする場合、[Document Builder ファクトリ]、[SAX パーサ ファクトリ]、および [Transformer ファクトリ] フィールドにファクトリ クラス名を入力します。そうでない場合は、フィールドを空白のままにします。

4. [作成] ボタンをクリックします。左ペインの [XML] ノードの下に、XML レジストリが作成されて表示されます。

5. 左ペインの [XML] ノード下で、XML レジストリの [XML エンティティ定義] ノードを右クリックします。ドロップダウン メニューから [新しい XMLEntitySpecRegistryEntry のコンフィグレーション] を選択します。エンティティ解決情報入力用の空のウィンドウが右ペインに表示されます。

図 7-4 Administration Console による外部エンティティのコンフィグレーション

6. XML ドキュメントで外部エンティティを参照するのに使用する [システム ID] または [パブリック ID] のどちらかを入力します。たとえば、次の car.xml ファイルの場合は、[システム ID] に対して「http://www.bea.com/dtds/car.dtd」と入力します。

コード リスト 7-2 car.xml ファイル

```
<?xml version="1.0"?>
<!-- 以下の XML ドキュメントでは自動車を説明 -->
<!DOCTYPE CAR PUBLIC "-//BEA Systems, Inc.//DTD for cars//EN"
"http//www.beasys.co.jp/index.html">
<CAR>
<MAKE>Toyota</MAKE>
<MODEL>Corrolla</MODEL>
```

```
<YEAR>1998</YEAR>
<ENGINE>1.5L</ENGINE>
<HP>149</HP>
</CAR>
```

7. 外部エンティティのローカル コピーをコンフィグレーションする場合、レジストリ エンティティ ディレクトリ *DOMAIN*/xml/registries/*reg_name* が存在することを確認してください。*DOMAIN* はドメイン ディレクトリ、*reg_name* は XML レジストリの名前です。ファイルが存在しない場合は、新たに作成します。
8. [エンティティ URI] フィールドで、以下のいずれかのエンティティ パスを入力します。
   a. 管理サーバ内のエンティティ ファイルのコピーのパス名。このパス名は、レジストリ エンティティ ディレクトリ *DOMAIN*/xml/registries/*reg_name* を基準にした相対パス名にする必要があります。

      たとえば、car.xml ファイルの場合、[エンティティ URI] フィールドに「dtds/car.dtd」と入力できます。

   b. Web 上の外部エンティティを示す URL、またはレポジトリに保存されているエンティティ。たとえば、「http://java.sun.com/j2ee/dtds/application_1_2.dtd」と入力して、J2EE エンタープライズ アプリケーションの説明に使用する application.xml ファイルの DTD を参照するか、jdbc: を使用してデータベース内の任意のエンティティを参照します。

      http://、file://、jdbc:、または ftp:// のプロトコル宣言を使用して、外部エンティティを指定します。
9. [キャッシュのタイミング] リスト ボックスから以下のオプションのいずれか 1 つを選択します。
   - [参照時] - WebLogic Server は、エンティティが最初に XML ドキュメントで参照されたときに URL で参照される外部エンティティをキャッシュします。
   - [初期化時] - WebLogic Server は、サーバ起動時にエンティティをキャッシュします。

- [レジストリの設定通り] - WebLogic Server は、デフォルトのキャッシュ設定を使用します。デフォルト キャッシュ設定のコンフィグレーションの詳細については、7-17 ページの「外部エンティティ キャッシュのコンフィグレーション」を参照してください。
- [キャッシュしない] - WebLogic Server は、外部エンティティをキャッシュしません。

10. [キャッシュ タイムアウト間隔] フィールドで、キャッシュされた外部エンティティが古くなる (期限切れになる) までの秒数を入力します。キャッシュされたコピーがこの期間を超えた場合に、WebLogic Server は、管理サーバを基準に指定した相対 URL またはパス名から外部エンティティを再取得します。

    このフィールドのデフォルト値は -1 で、WebLogic Server のグローバル タイムアウト値が使用されます。グローバル キャッシュ タイムアウト設定のコンフィグレーションの詳細については、7-17 ページの「外部エンティティ キャッシュのコンフィグレーション」を参照してください。

11. [作成] ボタンをクリックします。XMLEntitySpec レジストリ エントリが作成されます。
12. 左ペインの [サーバ] ノード下で、新しい XML レジストリに関連付けるサーバの名前をクリックします。
13. 右ペインで、[サービス] タブを選択します。
14. [XML] タブを選択します。図 7-2 のように、WebLogic Server の XML プロパティのコンフィグレーション用ウィンドウが右ペインに表示されます。
15. [XML レジストリ] フィールドで、このサーバに関連付ける XML レジストリ名を選択し、[適用] ボタンをクリックします。
16. サーバを再起動して新しい設定内容を有効にします。
17. Web からのキャッシュではなく、エンティティのローカル コピーを使用するように指定した場合は、エンティティ ファイルをエンティティ ディレクトリにコピーします。

    たとえば、`car.dtd` ファイルを *`DOMAIN`*`/xml/registries/`*`reg_name`*`/dtds` ディレクトリにコピーします。*`DOMAIN`* はドメイン ディレクトリ、*`reg_name`* は XML レジストリの名前です。

外部エンティティ キャッシュのコンフィグレーション

外部エンティティ キャッシュの以下のプロパティをコンフィグレーションできます。

- キャッシュ メモリのサイズ (単位 : KB)。このプロパティのデフォルト値は、500KB です。
- 永続ディスク キャッシュのサイズ (単位 : MB)。このプロパティのデフォルト値は、5MB です。
- 外部エンティティが WebLogic Server によってキャッシュされた後、キャッシュ内で陳腐化するまでの秒数。これは、サーバ全体のデフォルト値です。エンティティをコンフィグレーションする場合に、特定の外部エンティティに対応するようにこの値をオーバーライドできます。このプロパティのデフォルト値は、120 秒 (2 分) です。

外部エンティティ キャッシュをコンフィグレーションするには、次の手順に従います。

1. WebLogic 管理サーバを起動し、Administration Console をブラウザで起動します。

   Administration Console 開始の詳細については、7-1 ページの「WebLogic Server XML の管理の概要」を参照してください。

2. 左ペインの [サーバ] ノード下で、外部エンティティ キャッシュをコンフィグレーションする WebLogic Server の名前をクリックします。
3. 右ペインの [サービス] タブを選択します。
4. [XML] タブを選択します。図 7-2 のように、WebLogic Server の XML プロパティのコンフィグレーション用ウィンドウが右ペインに表示されます。
5. [キャッシュ メモリ サイズ] フィールドで、キャッシュ メモリのサイズ (KB) を入力します。
6. [キャッシュ ディスク サイズ] フィールドで、永続ディスク キャッシュのサイズ (MB) を入力します。

7. [キャッシュ タイムアウト間隔] フィールドで、エンティティが期限切れになるまでの秒数を入力します。
8. [適用] ボタンをクリックします。

外部エンティティ キャッシュのモニタ

外部エンティティ キャッシュを説明する一連の統計情報を使用して、キャッシュの効果をモニタできます。統計では、以下の情報を示します。

- キャッシュの現在の状態。
- 現在のセッションの累積アクティビティ。
- キャッシュ作成後、通常は WebLogic Server 起動後の累積アクティビティ。

外部エンティティ キャッシュをモニタするには、次の手順に従います。

1. WebLogic 管理サーバを起動し、Administration Console をブラウザで起動します。

   Administration Console 開始の詳細については、7-1 ページの「WebLogic Server XML の管理の概要」を参照してください。
2. 左ペインの [サーバ] ノード下で、外部エンティティ キャッシュをコンフィグレーションする WebLogic Server の名前をクリックします。
3. 右ペインの [サービス] タブを選択します。
4. [XML] タブを選択します。図 7-2 のように、WebLogic Server の XML プロパティのコンフィグレーション用ウィンドウが右ペインに表示されます。
5. 右ペインの [XML エンティティ キャッシュのモニタ] をクリックします。
6. キャッシュの現在の状態を表示するには [現在] タブを、現在のセッションの累積アクティビティを表示するには [セッション] タブを、キャッシュ作成以降、通常は WebLogic Server 起動以降の累積アクティビティを表示するには [履歴] タブを、それぞれクリックします。

以下の表で、外部エンティティ キャッシュの現在の状態を表示する場合のフィールドについて説明します。

表 7-1 キャッシュ統計の現在の状態

フィールド	タブ	説明
Total Current Entries	一般	キャッシュ内のエントリ合計数を返す。
Total Persistent Current Entries	一般	キャッシュ内の永続エントリ数を返す。
Total Transient Current Entries	一般	キャッシュ内の一時エントリ数を返す。
Avg Percent Transient	一般	エントリの中で一時エントリが占めるパーセントを返す。
Avg Percent Persistent	一般	エントリの中で永続エントリが占めるパーセントを返す。
Avg Timeout	一般	エントリの平均タイムアウト値を返す。
Min Entry Timeout	一般	現在のエントリの最小タイムアウト値を返す。
Max Entry Timeout	一般	現在のエントリの最大タイムアウト値を返す。
Avg Per Entry Memory Size	エントリ リソース使用状況	現在のエントリの平均メモリ サイズを返す。
Max Entry Memory Size	エントリ リソース使用状況	現在のエントリの最大メモリ サイズを返す。
Min Entry Memory Size	エントリ リソース使用状況	現在のエントリの最小メモリ サイズを返す。
Avg Per Entry Disk Size	エントリ リソース使用状況	現在のエントリの平均ディスク サイズを返す。
Memory Usage	合計リソース使用状況	すべてのメモリ常駐エントリの保存に使用するバイト数を返す。
Disk Usage	合計リソース使用状況	すべてのディスク常駐エントリの保存に使用するバイト数を返す。

以下の表で、外部エンティティ キャッシュの累積アクティビティを表示する場合のフィールドについて説明します。

表 7-2 キャッシュの累積アクティビティ

メソッド	タブ	説明
Total Current Entries	一般	キャッシュ内のエントリ合計数を返す。
Total Persistent Current Entries	一般	キャッシュ内の永続エントリ数を返す。
Total Transient Current Entries	一般	キャッシュ内の一時エントリ数を返す。
Avg Percent Transient	一般	エントリの中で一時エントリが占めるパーセントを返す。
Avg Percent Persistent	一般	エントリの中で永続エントリが占めるパーセントを返す。
Avg Timeout	一般	エントリの平均タイムアウト値を返す。
Min Entry Timeout	一般	現在のエントリの最小タイムアウト値を返す。
Max Entry Timeout	一般	現在のエントリの最大タイムアウト値を返す。
Avg Per Entry Memory Size	エントリ リソース使用状況	現在のエントリの平均メモリ サイズを返す。
Max Entry Memory Size	エントリ リソース使用状況	現在のエントリの最大メモリ サイズを返す。
Min Entry Memory Size	エントリ リソース使用状況	現在のエントリの最小メモリ サイズを返す。
Avg Per Entry Disk Size	エントリ リソース使用状況	現在のエントリの平均ディスク サイズを返す。
Total Number Of Rejections	拒否	拒否されたエントリ数を返す。
Total Size Of Rejections	拒否	拒否された全項目の合計サイズをバイト数で返す。

表 7-2 キャッシュの累積アクティビティ

メソッド	タブ	説明
Percent Rejected	拒否	拒否された挿入をパーセントで返す。
Total Number Of Renewals	拒否	期限切れエントリが更新された回数を返す。

8 XML リファレンス

以下の節では、WebLogic Server でサポートされる XML 仕様、アプリケーション プログラミング インタフェース (API)、およびツールに関する補足情報へのリンクを示します。

- XML API
- コード例
- 関連する WebLogic Server マニュアル
- チュートリアルとオンライン コース
- その他の XML 仕様と情報

XML API

- SAX 2.0 API
- DOM (Document Object Model) Level 2 仕様
- JAXP API 1.1 仕様
- Apache Xerces Java パーサ
- Apache Xalan XSLT トランスフォーマ

コード例

XML のサンプル コードとサポート マニュアルは、*WL_HOME*\samples\server\src\examples\xml\sax の WebLogic Server 配布キットにあります。*WL_HOME* は最上位 WebLogic Platform ディレクトリです。

関連する WebLogic Server マニュアル

- 『WebLogic Web サービス プログラマーズ ガイド』
- 『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』
- 『WebLogic JMS プログラマーズ ガイド』
- 『WebLogic JSP 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』
- 『WebLogic Server Wireless Application 開発プログラマーズ ガイド』

チュートリアルとオンライン コース

- A Technical Introduction to XML
- XML Authoring Tutorial
- Working with XML and Java
- Tutorials for using the Java 2 platform and XML technology
- Developing XML Solutions with JavaServer Pages Technology
- XML, Java, and the Future of the Web
- 『XML Bible』の第 14 章「XSL Transformations」
- 『XSL Tutorial』、Miloslav Nic 著
- XML Schema Part 0: Primer

その他の XML 仕様と情報

- XML 1.0 仕様

- XML Schema Part 1: Structures
- XML Schema Part 2: DatatypesW3C XML Namespaces 1.0 勧告
- Extensible Stylesheet LanguageXSL) 1.0 仕様
- JSR-000031 XML Data Binding 仕様
- XML Path Language (XPath) Version 1.0 仕様
- XML Linking Language (XLink) 仕様
- XML Pointer Language (XPointer) 仕様
- W3C (World Wide Web Consortium)
- XML.com
- XML FAQ
- XML.org, The XML Industry Portal

索引

T

U

W

X

い

か

く

さ

し

す

せ

と

は

ま

め

ゆ