BEA WebLogic Server™

WebLogic Web サービス 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 : 2004 年 1 月 9 日

著作権

限定的権利条項

商標または登録商標

WebLogic Web サービス 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	2004 年 1 月 9 日	BEA WebLogic Server バージョン 7.0

目次

4. WebLogic Web サービスの設計

5. WebLogic Web サービスの実装

6. Ant タスクを使用した WebLogic Web サービスのアセンブル

7. WebLogic Web サービスの手動アセンブル

8. Web サービスの呼び出し

9. 非組み込みデータ型の使用法

10. SOAP メッセージをインターセプトする SOAP メッセージ ハンドラの作成

11. セキュリティのコンフィグレーション

12. JMS で実装された WebLogic Web サービスの作成

13. WebLogic Web サービスの管理

14. UDDI を使用した Web サービスのパブリッシュと検索

15. 相互運用性

16. 6.1 WebLogic Web サービスから 7.0 へのアップグレード

A. WebLogic Web サービス デプロイメント記述子の要素

B. Web サービス Ant タスクとコマンドライン ユーティリティ

C. WebLogic Web サービスのカスタマイズ

D. WebLogic Web サービスでサポートされている仕様

このマニュアルの内容

このマニュアルでは、BEA WebLogic Web サービスについて説明し、作成方法およびクライアント アプリケーションから呼び出す方法について説明します。

マニュアルの内容は以下のとおりです。

- 第 1 章「WebLogic Web サービスの概要」では、Web サービスの概念、WebLogic Web サービスの機能について説明します。
- 第 2 章「アーキテクチャの概要」では、WebLogic Web サービスのアーキテクチャの概要について説明します。
- 第 3 章「WebLogic Web サービスの作成 : 簡単な例」では、ステートレス セッション EJB に基づいた簡単な WebLogic Web サービスを作成するエンド ツー エンド プロセスを説明します。
- 第 4 章「WebLogic Web サービスの設計」では、WebLogic Web サービスの開発前に注意が必要な設計事項について説明します。
- 第 5 章「WebLogic Web サービスの実装」では、Web サービスを実装するバックエンド コンポーネントの作成方法について説明します。
- 第 6 章「Ant タスクを使用した WebLogic Web サービスのアセンブル」では、WebLogic Web サービスの Ant タスクを使用して自動的に Web サービスの最終的な部品 (非組み込みデータ型のシリアライゼーション情報およびクライアント JAR ファイルなど) を生成する方法、それらをデプロイ可能な EAR ファイルにパッケージ化する方法、およびその EAR ファイルを WebLogic Server にデプロイする方法について説明します。
- 第 7 章「WebLogic Web サービスの手動アセンブル」では、WebLogic Web サービスを Ant タスクを使用しないで手動でアセンブルする方法について説明します。
- 第 8 章「Web サービスの呼び出し」では、WebLogic Web サービスを呼び出すクライアント アプリケーションの作成方法について説明します。
- 第 9 章「非組み込みデータ型の使用法」では、XML と Java 表現間のユーザ定義データ型を変換するシリアライザおよびデシリアライザの作成方法について説明します。

- 第 10 章「SOAP メッセージをインターセプトする SOAP メッセージ ハンドラの作成」では、この後の処理のため SOAP メッセージをインターセプトするハンドラを作成する方法を説明します。
- 第 11 章「セキュリティのコンフィグレーション」では、WebLogic Web サービスのセキュリティをコンフィグレーションする方法について説明します。
- 第 12 章「JMS で実装された WebLogic Web サービスの作成」では、JMS メッセージ コンシューマまたはプロデューサを実装する WebLogic Web サービスの作成方法について説明します。
- 第 13 章「WebLogic Web サービスの管理」では、WebLogic Web サービスを管理する Administration Console の使い方を説明します。
- 第 14 章「UDDI を使用した Web サービスのパブリッシュと検索」では、WebLogic Server に含まれている UDDI 機能の使い方を説明します。
- 第 15 章「相互運用性」では、Web サービスにとって相互運用が意味するものを説明し、相互運用性の高い Web サービスを作成するためのヒントを紹介します。
- 第 16 章「6.1 WebLogic Web サービスから 7.0 へのアップグレード」では、WebLogic Server のバージョン 6.1 で作成された Web サービスをバージョン 7.0 へアップグレードする方法について説明します。
- 付録 A「WebLogic Web サービス デプロイメント記述子の要素」では、Web サービス デプロイメント記述子ファイル (`web-services.xml`) の要素について説明します。
- 付録 B「Web サービス Ant タスクとコマンドライン ユーティリティ」では、等価コマンドライン ユーティリティと一緒に WebLogic Web サービスをアセンブルするために使用される Ant タスクについて説明します。
- 付録 C「WebLogic Web サービスのカスタマイズ」では、Web アプリケーションのデプロイメント記述子ファイル `web.xml` をアップグレードして WebLogic Web サービスをカスタマイズする方法について説明します。
- 付録 D「WebLogic Web サービスでサポートされている仕様」では、SOAP 1.1、WSDL 1.1 など WebLogic Web サービスによってサポートされた仕様についての情報を提供します。

対象読者

このマニュアルは、WebLogic Server で実行する Web サービスを作成する Java 開発者を対象としています。

Web テクノロジ、XML、および Java プログラミング言語に読者が精通していることを前提として書かれています。

e-docs Web サイト

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

このマニュアルの印刷方法

Web ブラウザの [ファイル | 印刷] オプションを使用すると、Web ブラウザからこのマニュアルを一度に 1 章ずつ印刷できます。

このマニュアルの PDF 版は、WebLogic Server の 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。

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、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。

表記法	適用
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。
[ ]	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}`

表記法	適用
. . .	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　WebLogic Web サービスの概要

この章では、Web サービスの概要と WebLogic Server に実装する方法について説明します。

Web サービス

Web サービスは、分散型 Web ベース アプリケーションのコンポーネントとして共有および使用されるサービスのタイプです。これらのサービスは一般的に、CRM (又はカスタマ リレーションシップ マネージメント)、注文処理システムなどの既存のバックエンド アプリケーションとインタフェースします。

従来、ソフトウェア アプリケーション アーキテクチャは、メインフレーム上で実行する巨大なモノリシック システムまたはデスクトップ上で実行するクライアント アプリケーションの 2 つのカテゴリに分けられていました。これらのアーキテクチャはアプリケーションが扱うことを意図しているという目的においては効率よく機能しますが、閉鎖的で、Web の多様なユーザが簡単にアクセスすることができません。

ソフトウェア業界は、Web 上で動的に対話する疎結合なサービス指向アプリケーションの方向に向かっています。このアプリケーションは、大規模なソフトウェア システムを小規模なモジュラー コンポーネントまたは共有サービスに分割します。これらのサービスは別々のコンピュータ上に常駐することができ、多様なテクノロジを使用して実装できるにも関わらず、XML や HTTP などの標準の Web プロトコルを使用してパッケージ化され転送可能です。そのため、Web 上のすべてのユーザが簡単にアクセスできるようになります。

サービスの概念は新しいものではありません。RMI、COM、および CORBA は、すべてサービス指向テクノロジです。ただし、これらのテクノロジをベースにしたアプリケーションは、特定のベンダの特定のテクノロジを使用して記述されていることが要求されます。この要件は、一般的に Web 上のアプリケーションが広く受け入れられることの妨げになります。この問題を解決するために、Web サービスが異種環境から簡単にアクセスできるようにするための次の特性を共有するように定義されています。

- Web サービスが Web 上でアクセス可能であること。
- Web サービスが XML ベースの記述言語を使用して記述されていること。
- Web サービスが HTTP などの標準のインターネット プロトコルによって転送される XML メッセージを介してクライアント (エンドユーザ アプリケーションまたは Web サービスの両方) と通信すること。

Web サービスを使用する理由

Web サービスの主な利点は、次のとおりです。

- 多様なハードウェアおよびソフトウェア プラットフォームにわたる分散型アプリケーション間の相互運用性
- Web プロトコルを使用したファイアウォールによる、アプリケーションへの容易で幅広いアクセス
- 異機種分散型アプリケーションの開発を容易にするクロス プラットフォーム、クロス言語データ モデル (XML)

Web サービスは、XML や HTTP などの標準の Web プロトコルを使用してアクセスされるので、Web 上の多様な異種アプリケーション (一般的に XML および HTTP を理解する) は自動的に Web サービスにアクセスでき、相互の通信が可能になります。

これらの異なるシステムには Microsoft SOAP ToolKit クライアント、J2EE アプリケーション、レガシー アプリケーションなどがあります。アプリケーション作成に使用されているプログラミング言語は、Java、C++、Perl などです。アプリケーションの相互運用性が Web サービスの目標であり、それは、パブリッシュされている業界標準をサービス プロバイダがどれだけ忠実に遵守しているかによって左右されます。

Web サービス規格

Web サービスは、次のコンポーネントで構成されています。

- Web 上のサーバによってホストされる実装

  WebLogic Web サービスは、WebLogic Server によってホストされ、標準の J2EE コンポーネント (エンタープライズ Java Beans や JMS など) および Java クラスを使用して実装され、標準の J2EE エンタープライズ アプリケーションとしてパッケージ化されます。

- 標準のデータ転送および Web サービスは、Web サービスと Web サービスのユーザ間で呼び出します。

  WebLogic Web サービスでは、Simple Object Access Protocol (SOAP) 1.1 および 1.2 をメッセージ フォーマットとして使用し、HTTP を接続プロトコルとして使用します。1-4 ページの「添付ファイル付き SOAP 1.1」を参照してください。

  注意： WebLogic Web サービスは SOAP 1.1 と 1.2 両方のリクエストを受け付けますが、作成するのは SOAP 1.1 の応答だけです。

- Web サービスを起動できるようにクライアントに記述する標準的な方法

  WebLogic Web サービスは、XML ベースの仕様である Web Services Description Language (WSDL) 1.1 を使用して記述しています。1-5 ページの「WSDL 1.1」を参照してください。

- クライアント アプリケーションから Web サービスを呼び出す標準的な方法

  WebLogic Web サービスは、クライアント アプリケーションから WebLogic および非 WebLogic の Web サービスを呼び出すのに使用できる、クライアント JAR の一環として JAX-RPC (Java API for XML-based RPC) を実装しています。1-6 ページの「JAX-RPC」を参照してください。

- クライアント アプリケーションからの登録済み Ｗｅｂ サービスの検索および Web サービスの登録を行う標準的な方法

  WebLogic Web サービスは、UDDI (Universal Description, Discovery, and Integration) 仕様を実装しています。1-7 ページの「UDDI 2.0」を参照してください。

添付ファイル付き SOAP 1.1

SOAP (Simple Object Access Protocol) は、分散型環境で情報を交換するために使用する軽量 XML ベースのプロトコルです。WebLogic Server には、SOAP 1.1 仕様と添付ファイル付き SOAP 仕様のインプリメンテーションが含まれています。プロトコルの構成は以下のとおりです。

- SOAP メッセージを記述するエンベロープ。メッセージの本文を含むエンベロープは、処理するユーザを識別し、処理方法を記述します。
- アプリケーション固有のデータ型のインスタンスを表現するためのエンコーディング ルール セット。
- リモート プロシージャ呼び出しおよび応答を表す規則。

この情報は、HTTP またはその他の Web プロトコル上で転送可能な Multipurpose Internet Mail Extensions (MIME) エンコード パッケージ内に埋め込まれています。MIME は、非 ASCII メッセージをインターネット上で送信できるようにフォーマットするための仕様です。

注意： WebLogic Web サービスは SOAP 1.1 と 1.2 両方のリクエストを受け付けますが、作成するのは SOAP 1.1 の応答だけです。

次の例は、HTTP リクエスト内に埋め込まれている株取引情報用の SOAP リクエストを示しています。

```
POST /StockQuote HTTP/1.1
Host: www.sample.com
```

```
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
         SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
     <SOAP-ENV:Body>
          <m:GetLastStockQuote xmlns:m="Some-URI">
               <symbol>BEAS</symbol>
          </m:GetLastStockQuote>
     </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

WSDL 1.1

Web Services Description Language (WSDL) は、Web サービスを記述するための XML ベースの仕様です。WSDL ドキュメントは Web サービスによって提供されるメソッド、入力および出力パラメータ、および接続方法を記述します。

WebLogic Web サービスの開発者は、WSDL ファイルを作成する必要はありません。これらのファイルは WebLogic Web サービス開発プロセスの一部として自動的に生成されます。

次の例は参照用で GetLastStockQuote メソッドを含む株取引 Web サービス StockQuoteService を記述する WSDL ファイルを示しています。

```
<?xml version="1.0"?>
  <definitions name="StockQuote"
               targetNamespace="http://sample.com/stockquote.wsdl"
               xmlns:tns="http://sample.com/stockquote.wsdl"
               xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
               xmlns:xsd1="http://sample.com/stockquote.xsd"
               xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
               xmlns="http://schemas.xmlsoap.org/wsdl/">
      <message name="GetStockPriceInput">
          <part name="symbol" element="xsd:string"/>
      </message>
      <message name="GetStockPriceOutput">
          <part name="result" type="xsd:float"/>
      </message>
      <portType name="StockQuotePortType">
          <operation name="GetLastStockQuote">
             <input message="tns:GetStockPriceInput"/>
             <output message="tns:GetStockPriceOutput"/>
```

```
        </operation>
    </portType>
    <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
        <soap:binding style="rpc"
                      transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetLastStockQuote">
           <soap:operation soapAction="http://sample.com/GetLastStockQuote"/>
           <input>
             <soap:body use="encoded" namespace="http://sample.com/stockquote"
                    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
           </input>
           <output>
           <soap:body use="encoded" namespace="http://sample.com/stockquote"
                    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
           </output>
        </operation>>
    </binding>
    <service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
           <soap:address location="http://sample.com/stockquote"/>
        </port>
    </service>
</definitions>
```

JAX-RPC

JAX-RPC は Sun Microsystems の仕様で、Web サービスを呼び出すクライアント API を定義します。

次の表で、JAX-RPC の中心的なインタフェースとクラスを簡単に説明します。

表 1-1 JAX-RPC のインタフェースとクラス

javax.xml.rpc のインタフェースまたはクラス	説明
Service	主要なクライアント インタフェース。静的、動的呼び出しの両方で使用する。
ServiceFactory	`Service` インスタンスの生成に使用するファクトリ クラス。

表 1-1 JAX-RPC のインタフェースとクラス

javax.xml.rpc のインタフェースまたはクラス	説明
Stub	Web サービスのオペレーションを呼び出すためのクライアント プロキシを表す。通常は、Web サービスの静的呼び出しに使用する。
Call	Web サービスを動的に呼び出すのに使用する。
JAXRPCException	Web サービスの呼び出し中にエラーが発生した場合に送出される例外。

JAX-RPC についての詳細は、Web サイト、http://java.sun.com/xml/jaxrpc/index.html を参照してください。

JAX-RPC を使用して Web サービスを呼び出す方法に関するチュートリアルについては、http://java.sun.com/webservices/docs/ea1/tutorial/doc/JAXRPC.html を参照してください。

UDDI 2.0

UDDI 仕様では、Web サービスの記述、既知のレジストリへの Web サービスの登録、他の登録済み Web サービスの検索などの標準的な方法が定義されています。

http://www.uddi.org を参照してください。

WebLogic Web サービスの機能

WebLogic Web サービス サブシステムには、以下の機能があります。

- **標準仕様**

  1-3 ページの「Web サービス規格」を参照してください。

- **標準 J2EE コンポーネントの公開のサポート**

  WebLogic Web サービスは、ステートレス セッション EJB、JMS コンシューマ、JMS プロデューサなどの標準 J2EE コンポーネントの公開をサポートします。

- **Ant タスクおよびコマンドライン ユーティリティ**

  Ant タスクは、Web サービスの実装およびパッケージ化を容易にします。付録 B「Web サービス Ant タスクとコマンドライン ユーティリティ」を参照してください。

- **UDDI レジストリ、ディレトリ エクスプローラ、クライアント API**

  WebLogic Server には、UDDI レジストリ、UDDI ディレクトリ エクスプローラ、UDDI クライアント API 実装などが組み込まれています。第 14 章「UDDI を使用した Web サービスのパブリッシュと検索」を参照してください。

- **RPC 指向とドキュメント指向両方のオペレーションのサポート**

  WebLogic Web サービスのオペレーションは、RPC 指向 (SOAP メッセージがパラメータと戻り値を格納) またはドキュメント指向 (SOAP メッセージがドキュメントを格納) がサポートされています。詳細については、4-3 ページの「RPC 指向またはドキュメント指向の Web サービス」を参照してください。

- **ユーザ定義データ型のサポート**

  パラメータおよび戻り値に非組み込みデータ型が使用される WebLogic Web サービスを作成することができます。非組み込みデータ型は、サポートされている組み込みデータ型以外のデータ型として定義されます。組み込みデータ型の例としては、`int` や `String` があります。WebLogic Server の Ant タスクによって、非組み込みデータ型の使用が必須であるコンポーネントを生成することができます。この機能は オートタイピングと呼ばれます。また、これらのコンポーネントを手動で作成することができます。付録 B「Web サービス Ant タスクとコマンドライン ユーティリティ」と第 9 章「非組み込みデータ型の使用法」を参照してください。

- **SOAP メッセージにアクセスするための SOAP メッセージ ハンドラ**

  SOAP メッセージ ハンドラは、Web サービスの要求と応答の両方で SOAP メッセージとその添付ファイルにアクセスします。ハンドラは、Web サービスそのものと Web サービスを呼び出すクライアント アプリケーションの両

方で作成することができます。第 10 章「SOAP メッセージをインターセプトする SOAP メッセージ ハンドラの作成」を参照してください。

- **Web サービスを呼び出す Java クライアント**

  開発者は自動的に生成されるシン Java クライアントを使用して、WebLogic および非 WebLogic の Web サービスを呼び出す Java クライアント アプリケーションを作成することができます。第 8 章「Web サービスの呼び出し」を参照してください。

  注意： クライアント機能の現在の BEA のライセンス供与については、BEA eLicense の Web サイトを参照してください。

- **Web サービス ホーム ページ**

  デプロイされた WebLogic Web サービスにはすべて自動的にホーム ページが設けられ、その Web サービスの WSDL へのリンク、Web サービスを呼び出すためにダウンロードできるクライアント JAR ファイル、および Web サービスが適切に動作しているかを確認するテスト メカニズムが配置されます。8-23 ページの「WebLogic Web サービスのホーム ページおよび WSDL の URL」を参照してください。

- **ポイント ツー ポイント セキュリティ**

  WebLogic Server は、WebLogic Web サービス オペレーションに対して、Web サービス オペレーションの許可と認証をサポートするほか、接続指向のポイント ツー ポイント セキュリティもサポートします。第 11 章「セキュリティのコンフィグレーション」を参照してください。

- **相互運用性**

  WebLogic Web は、Microsoft .NET などの主な Web サービス プラットフォームと相互運用できます。

- **Java 2 Platform Micro Edition(J2ME) クライアント**

  WebLogic Server の `clientgen` Ant タスクは、J2ME で実行されるクライアント JAR ファイルを作成することができます。第 8 章「Web サービスの呼び出し」を参照してください。

Web サービスの作成例および呼び出し例

WebLogic Server の *WL_HOME*\\samples\server\src\examples\webservices ディレクトリには、WebLogic Web サービスの作成および呼び出しの方法として以下の例があります。*WL_HOME* は、WebLogic プラットフォームのメイン ディレクトリです。

- basic.statelessSession: パラメータと戻り値が組み込みデータ型の値となるステートレス セッション EJB バックエンド コンポーネントを使用します。
- basic.javaclass: パラメータと戻り値が組み込みデータ型の値となる Java クラス バックエンド コンポーネントを使用します。
- complex.statelessSession: パラメータと戻り値が非組み込みデータ型の値となるステートレス セッション EJB バックエンド コンポーネントを使用します。
- handler.log: ハンドラ チェーンとステートレス セッション EJB の両方を使用します。
- handler.nocomponent: バックエンド コンポーネントなしでハンドラ チェーンのみを使用します。
- client.static: 非 WebLogic Web サービスを呼び出す静的クライアント アプリケーションの作成方法を示します。
- client.static_out: 出力パラメータを使用する非 WebLogic Web サービスを呼び出す静的クライアント アプリケーションの作成方法を示します。
- client.dynamic_wsdl: WSDL を使用して非 WebLogic Web サービスを呼び出す動的クライアント アプリケーションの作成方法を示します。
- client.dynamic_no_wsdl: WSDL を使用しないで非 WebLogic Web サービスを呼び出す動的クライアント アプリケーションの作成方法を示します。
- message: JMS で実装される WebLogic Web サービスの作成方法を示します。
- multicomponent: 2 つの Java クラスで実装される WebLogic Web サービスの作成方法を示します。
- r4client: WebLogic Web サービスのクライアントを実行して SOAPBuilders Interoperability Lab Round 3 テストを行う方法を示します。

上記の例の構築と実行についての詳細な説明は、次の Web ページを参照してください。

WL_HOME\samples\server\src\examples\webservices\package-summary.html

WebLogic Web サービスの作成と呼び出しの例は、このほかにも、Web Services dev2dev Download Page の Web サービスのページに掲載されています。

WebLogic Web サービスの作成：主な手順

以下は、WebLogic Web サービスを作成する高度な手順です。ほとんどの手順については、後述の章で詳しく説明します。

第 3 章「WebLogic Web サービスの作成 : 簡単な例」で、Web サービス作成例を簡単に説明します。

1. WebLogic Web サービスを設計します。

   同期または非同期の Web サービスで、サービスを実装するバックエンド コンポーネントのタイプ、組み込みデータ型またはカスタム データ型のみを使用するサービスかどうか、送受信される SOAP メッセージをインターセプトする必要があるかどうか、などを決定します。

   第 4 章「WebLogic Web サービスの設計」を参照してください。

2. WebLogic Web サービスを実装します。

   Web サービスを構成するバックエンド コンポーネントの Java コードを記述します。必要であれば、SOAP メッセージをインターセプトする SOAP メッセージ ハンドラや XML と Java 間のデータ変換を行う独自のシリアライゼーション クラスなどを作成します。

   第 5 章「WebLogic Web サービスの実装」を参照してください。

3. WebLogic Web サービスをアセンブルし、パッケージ化します。

   すべての実装クラス ファイルを、適切なディレクトリ構造に収集して Web サービス デプロイメント記述子ファイルを作成し、サービスのサポート部分 (クライアント JAR ファイルなど) を作成し、すべてをデプロイメント EAR ファイルにパッケージ化します。

簡素な Web サービスを作成する場合は、すべてのアセンブリ ステップを実行する `servicegen` Ant タスクを使用してください。やや複雑な Web サービスを作成する場合は、他の Ant タスクも使用するか、手動でアセンブルしてください。

第 6 章「Ant タスクを使用した WebLogic Web サービスのアセンブル」を参照してください。

4. Web サービスにアクセスして Web サービスが期待通りに動作していることをテストするクライアントを作成します。Web サービス ホーム ページを使用して、作成した Web サービスをテストすることもできます。

   第 8 章「Web サービスの呼び出し」を参照してください。

5. WebLogic Web サービスをデプロイします。

   Web サービスをリモート クライアントで使用できるようにします。WebLogic Web サービスは標準 J2EE エンタープライズ アプリケーションとしてパッケージ化されているため、Web サービスのデプロイはエンタープライズ アプリケーションのデプロイと同じです。

   「デプロイメント」を参照してください。

6. 必要であれば、Web サービスを UDDI レジストリでパブリッシュすることができます。第 14 章「UDDI を使用した Web サービスのパブリッシュと検索」を参照してください。

バージョン 6.1、7.X の WebLogic Web サービスの相違点

WebLogic Web サービスは、WebLogic Server バージョン 6.1 とバージョン 7.X では異なっています。

- この 2 つのバージョンの実行時システムの相違により、WebLogic Server バージョン 6.1 で作成した Web サービスをバージョン 7.x で実行するには、アップグレードが必要です。バージョン 7.x の WebLogic Server インスタンスにバージョン 6.1 の Web サービスをデプロイしないでください。付録 16「6.1 WebLogic Web サービスから 7.0 へのアップグレード」を参照してください。

- WebLogic Server バージョン 6.1 に組み込まれている WebLogic Web サービス クライアント API は非推奨となっています。バージョン 6.1 のクライアント API は、`weblogic.soap.*` としてパッケージ化されていました。バージョン 7.x には、JAX-RPC をベースとした新しいクライアント API が組み込まれています。詳細については、第 8 章「Web サービスの呼び出し」を参照してください。
- バージョン 7.x Web サービスは、ステートレス セッション EJB、Java クラス、JMS リスナなどの複数のバックエンド コンポーネントを組み込んで実装することができます。バージョン 6.1 では、1 つの Web サービスの実装には、バックエンド コンポーネントは 1 つのみ組み込めました。このため、バージョン 7.x では、1 つの Web サービス全体を指す場合は、「rpc-style」および「message-style」という用語は使用していません。

  また、バージョン 7.x では、EJB のメソッドのうち、公開するものを選択することができます。バージョン 6.1 では EJB のパブリック メソッドはすべて公開しなければなりませんでした。

  第 2 章「アーキテクチャの概要」を参照してください。
- バージョン 7.x の Web サービスは、out および inout パラメータのほか、非組み込みデータ型のパラメータおよび戻り値をサポートします。バージョン 6.1 の WebLogic Web サービスでは、パラメータおよび戻り値として使用できるのは組み込みデータ型に限定されていました。第 9 章「非組み込みデータ型の使用法」を参照してください。
- バージョン 7.x では、SOAP メッセージをインターセプトしてさらに処理を加えるための SOAP メッセージ ハンドラとハンドラ チェーンが導入されています。バージョン 6.1 の WebLogic Web サービスは、SOAP の要求メッセージや応答メッセージにはアクセスできませんでした。第 10 章「SOAP メッセージをインターセプトする SOAP メッセージ ハンドラの作成」を参照してください。
- バージョン 7.x では、SOAP メッセージ ハンドラを使用して SOAP 添付ファイルにアクセスし、処理を加えることができます。バージョン 6.1 では添付ファイルにはアクセスできませんでした。第 10 章「SOAP メッセージをインターセプトする SOAP メッセージ ハンドラの作成」を参照してください。
- バージョン 7.x の Web サービスでは、`web-services.xml` という、各 Web サービス専用のデプロイメント記述子が用意されています。`web-services.xml` ファイルに関するリファレンス情報については、付録 A

「WebLogic Web サービス デプロイメント記述子の要素」を参照してください。ファイル例は、第 7 章「WebLogic Web サービスの手動アセンブル」を参照してください。

- WebLogic Server 7.x では、WebLogic Web サービスを短期間で実装、アセンブル、パッケージ化するための Ant タスクが新たにサポートされています。第 6 章「Ant タスクを使用した WebLogic Web サービスのアセンブル」を参照してください。
- バージョン 7.x では、Web サービスが非同期一方向サービスとなるように指定することができます。このように指定すると、Web サービスを呼び出すクライアント アプリケーションは、例外も含めていかなる応答も受け取りません。バージョン 6.1 では、すべての Web サービスは、同期要求応答方式であったため、クライアント アプリケーションは、サービスを呼び出すたびに応答を受け取りました。付録 A「WebLogic Web サービス デプロイメント記述子の要素」を参照してください。
- WebLogic Server バージョン 7.x の Administration Console には、Web サービスの管理を容易にするため、**Web サービス コンポーネント** デプロイメント ノードが新たに設けられています。第 13 章「WebLogic Web サービスの管理」を参照してください。
- WebLogic Server バージョン 7.x には、UDDI のサーバ機能とクライアント機能が組み込まれています。第 14 章「UDDI を使用した Web サービスのパブリッシュと検索」を参照してください。
- バージョン 7.x の Web サービス ホーム ページは、バージョン 6.1 のホーム ページから改善されました。新しいホーム ページには、Web サービスをテストできる機能が追加されています。8-23 ページの「WebLogic Web サービスのホーム ページおよび WSDL の URL」を参照してください。

サポートされていない機能

WebLogic Server では、以下の WSDL スキーマおよび XML スキーマの機能はサポートされていません。

- WSDL での過負荷オペレーション (SOAP の制約による)
- 制約ごとの XML スキーマ複合データ型の継承

- XML スキーマ ユニオンのシンプル データ型

XML ファイルの編集

WebLogic Web サービスを作成する際または呼び出す際は、`web-services.xml` Web サービス デプロイメント記述子ファイル、EJB デプロイメント記述子、Java Ant ビルド ファイルなどの XML ファイルを編集しなければならない場合があります。これらのファイルは、BEA XML エディタを使用して編集します。

注意： このマニュアルでは、ファイルおよび各要素で Web サービスがどのように記述されているかについて、プログラマの理解を助けるために、`web-services.xml` デプロイメント記述子を手動で作成、更新する方法について説明します。BEA XML エディタは、ファイルの更新にも使用できます。

BEA 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 Java ベースのツールです。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。このように、ドキュメントが 2 通りの方法で表示できるため、XML ドキュメントの編集には 2 つのオプションがあります。

- 階層ツリー表示では、構造化された制約のある編集が可能で、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ことができます。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 からダウンロードすることができます。

2　アーキテクチャの概要

この章では、WebLogic Web サービスのアーキテクチャの概要と WebLogic Web サービスの 3 種類のオペレーションについて説明します。

WebLogic Web サービスのアーキテクチャ

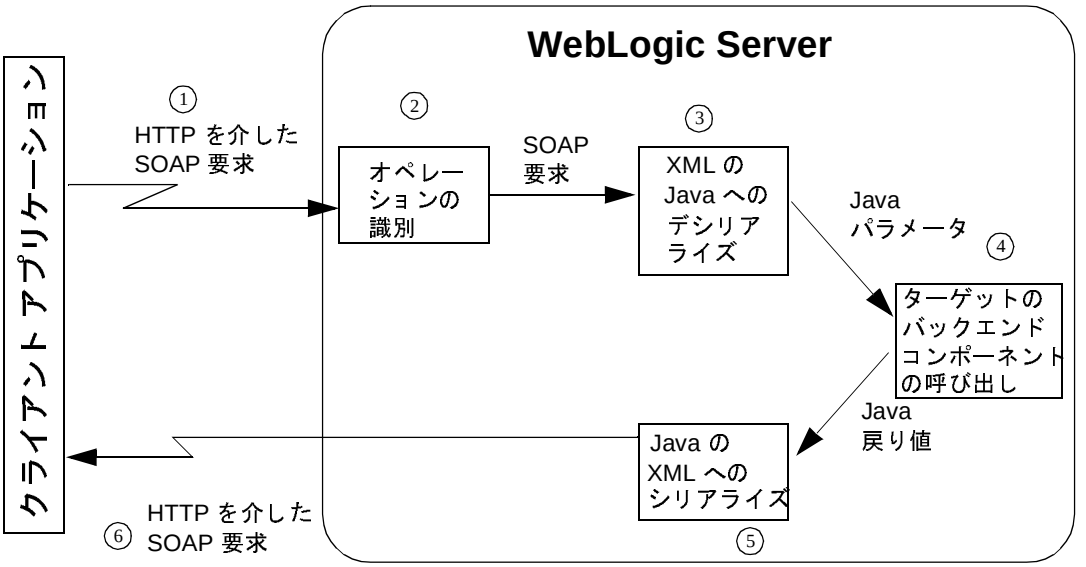
WebLogic Web サービスを開発するときは、ステートレス セッション EJB などの標準の J2EE コンポーネントと Java クラスを使用します。WebLogic Web サービスは J2EE プラットフォームに基づいているため、簡単で一般的なコンポーネント ベースの開発モデル、スケーラビリティ、トランザクションのサポート、ライフサイクル管理、既存のエンタープライズ システムへの J2EE API (JDBC および JTA など) を介した簡単なアクセス、簡素な統一セキュリティ モデルなど、すべての標準 J2EE の利点を自動的に継承します。

単一の WebLogic Web サービスは、1 つまたは複数のオペレーションで構成されています。各オペレーションは、異なるバックエンド コンポーネントと SOAP メッセージ ハンドラを使用して実装することができます。たとえば、オペレーションは、ステートレス セッション EJB の単一メソッドを使用して実装されている場合、または SOAP メッセージ ハンドラとステートレス セッション EJB を組み合わせて実装されている場合があります。

バックエンド コンポーネントのみのオペレーション

次の図は、ステートレス セッション EJB のメソッドなどのバックエンドコンポーネントのみで実装されている WebLogic Web サービス オペレーションのアーキテクチャを示しています。

図 2-1　バックエンド コンポーネントで実装された WebLogic Web サービス



クライアント アプリケーションがこのタイプの WebLogic Web サービス オペレーションを呼び出すと、次のような動作となります。

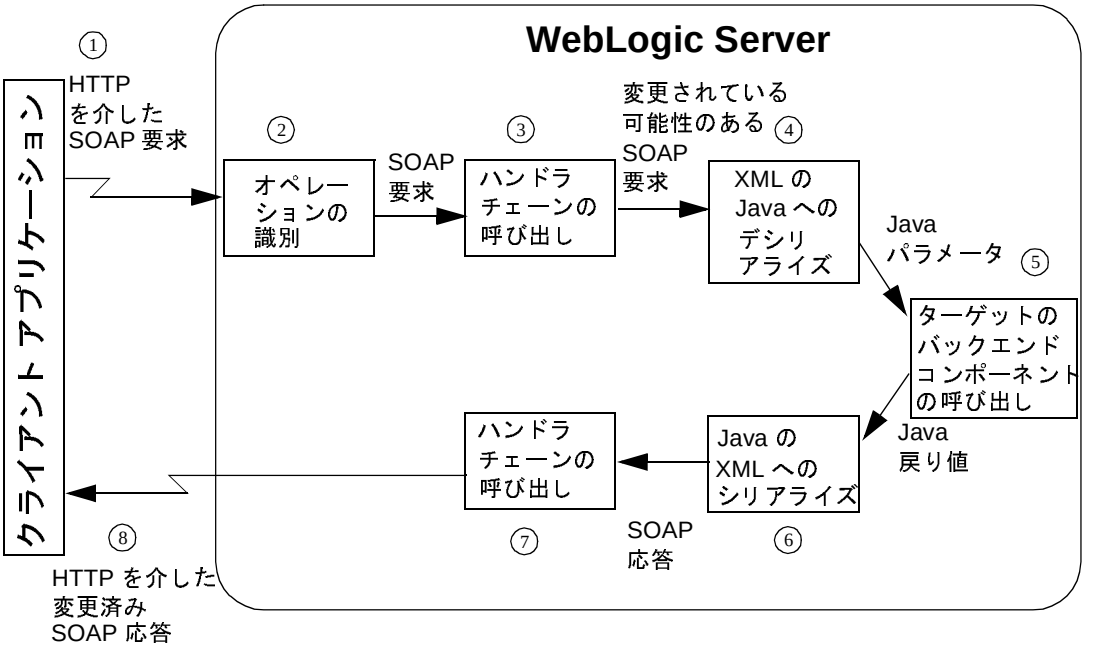
1. クライアント アプリケーションは、HTTP を介して SOAP 要求メッセージを WebLogic Server に送信します。WebLogic Server は、要求に含まれる URI に基づいて、呼び出されている Web サービスを識別します。
2. この Web サービスは、SOAP 要求メッセージを読み取って、実行すべきオペレーションを識別します。そのオペレーションは、ステートレス セッション EJB または Java クラスのメソッドの呼び出しに対応しています。

3. Web サービスは、適切なデシリアライザ クラスを使用して、SOAP メッセージにあるオペレーションのパラメータを XML 表現から Java 表現に変換します。デシリアライザ クラスには、組み込みデータ型用の WebLogic Server 指定のクラスと、非組み込みデータ型用のユーザ定義のクラスがあります。
4. Web サービスは適切なバックエンド コンポーネント メソッドを呼び出して Java パラメータに渡します。
5. Web サービスは、適切なシリアライザ クラスを使用してメソッドの戻り値を Java から XML に変換し、SOAP 応答メッセージにパッケージ化します。
6. Web サービスは、その Web サービスを呼び出したクライアント アプリケーションに SOAP 応答メッセージを戻します。

バックエンド コンポーネントと SOAP メッセージ ハンドラ チェーンのオペレーション

次の図は、SOAP メッセージ ハンドラ チェーンとバックエンド コンポーネントの両方を使用して実装された WebLogic Web サービス オペレーションを示しています。

図 2-2　バックエンド コンポーネントと SOAP メッセージ ハンドラ チェーンで実装された WebLogic Web サービス オペレーション



クライアント アプリケーションがこのタイプの WebLogic Web サービス オペレーションを呼び出すと、次のような動作となります。

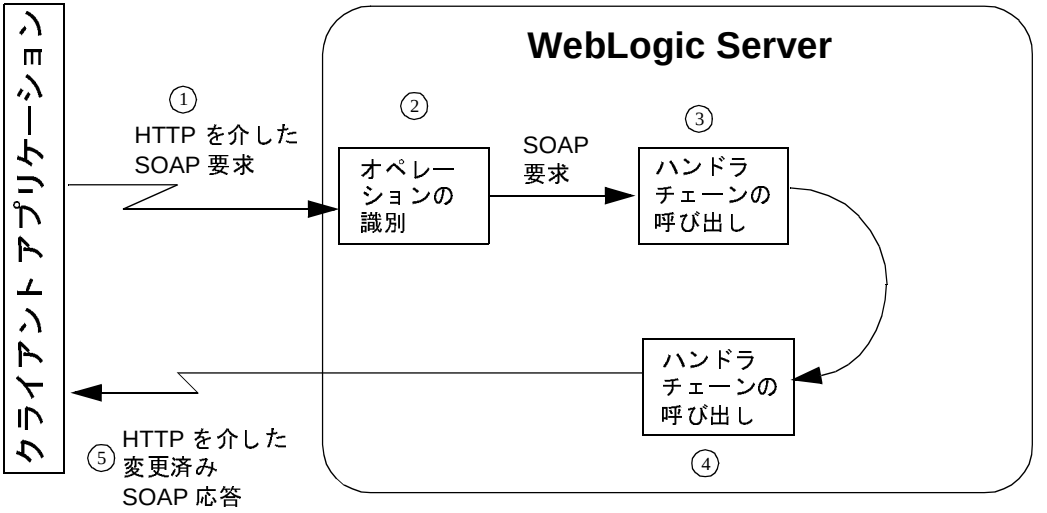
1. クライアント アプリケーションは、HTTP を介して SOAP 要求メッセージを WebLogic Server に送信します。WebLogic Server は、要求に含まれる URI に基づいて、呼び出されている Web サービスを識別します。

2. この Web サービスは、SOAP 要求メッセージを読み取って、実行すべきオペレーションを識別します。このケースのオペレーションは、ステートレス セッション EJB または Java クラスのメソッドの呼び出しを伴う SOAP メッセージ ハンドラ チェーンの呼び出しに対応しています。
3. Web サービスは適切なハンドラ チェーンを呼び出します。ハンドラ チェーンは、SOAP 要求メッセージの内容にアクセスし、おそらく、何らかの変更を加えます。
4. Web サービスは、適切なデシリアライザ クラスを使用して、SOAP メッセージ (変更されている可能性あり) にあるオペレーションのパラメータを XML 表現から Java 表現に変換します。デシリアライザ クラスには、組み込みデータ型用の WebLogic Server 指定のクラスと、非組み込みデータ型用のユーザ定義のクラスがあります。
5. Web サービスは適切なバックエンド コンポーネント メソッドを呼び出して Java パラメータに渡します。
6. Web サービスは、適切なシリアライザ クラスを使用してメソッドの戻り値を Java から XML に変換し、SOAP 応答メッセージにパッケージ化します。
7. Web サービスは適切な SOAP メッセージ ハンドラ チェーンを呼び出します。ハンドラ チェーンは、SOAP 応答メッセージの内容にアクセスし、何らかの変更を加える場合もあります。
8. Web サービスは、その Web サービスを呼び出したクライアント アプリケーションに SOAP 応答メッセージ (変更されている可能性あり) を戻します。

SOAP メッセージ ハンドラ チェーンのみのオペレーション

次の図は、SOAP メッセージ ハンドラ チェーンのみを使用して実装された WebLogic Web サービス オペレーションのアーキテクチャを示しています。

図 2-3　SOAP メッセージ ハンドラ チェーンのみで実装された WebLogic Web サービス オペレーション



クライアント アプリケーションがこのタイプの WebLogic Web サービス オペレーションを呼び出すと、次のような動作となります。

1. クライアント アプリケーションは、HTTP を介して SOAP 要求メッセージを WebLogic Server に送信します。WebLogic Server は、要求に含まれる URI に基づいて、呼び出されている Web サービスを識別します。
2. この Web サービスは、SOAP 要求メッセージを読み取って、実行すべきオペレーションを識別します。このケースのオペレーションは、SOAP メッセージ ハンドラ チェーンの呼び出しに対応しています。

3. Web サービスは適切なハンドラ チェーンを呼び出します。ハンドラ チェーンは、SOAP 要求メッセージの内容にアクセスし、おそらく、何らかの変更を加えます。
4. Web サービスは適切なハンドラ チェーンを呼び出します。このハンドラ チェーンは、SOAP 応答メッセージを作成します。
5. Web サービスは、その Web サービスを呼び出したクライアント アプリケーションに SOAP 応答メッセージを戻します。

3 WebLogic Web サービスの作成 : 簡単な例

この章では、WebLogic Web サービスの簡単な作成例について説明します。

サンプルの解説

この例では、*WL_HOME*\samples\server\src\examples\webservices\complex\statelessSession ディレクトリに製品例として格納されている WebLogic Web サービスの実装、アセンブル、およびデプロイのプロセスを始めから終わりまで説明します。*WL_HOME* は WebLogic プラットフォームのメイン ディレクトリです。

この例では、WebLogic Web サービスをステートレス セッション EJB に基づいて作成する方法を示します。この例では、Trader EJB (*WL_HOME*\samples\server\src\examples\ejb20\basic\statelessSession ディレクトリにある EJB 2.0 対応サンプルの 1 つ) を使用しています。

Trader EJB では、入力として株式シンボルである String と購入または売却する株数である int を受け取る、buy() と sell() という 2 つのメソッドが定義されています。これらのメソッドはともに、TraderResult という非組み込みデータ型を返します。

Trader EJB が Web サービスに変換されると、この 2 つのメソッドは、この Web サービスの WSDL で定義されるパブリック オペレーションとなります。Client.java アプリケーションは、JAX-RPC クライアント API を使用してオペレーションを呼び出す SOAP メッセージを作成します。

WebLogic Web サービスの作成例：主な手順

Trader WebLogic Web サービスのサンプルを作成するには、次の手順に従います。

1. 環境を設定します。

   Windows NT では、setExamplesEnv.cmd コマンドを実行します。このコマンドは、*WL_HOME*\samples\server\config\examples ディレクトリにあります。*WL_HOME* は WebLogic プラットフォームのメイン ディレクトリです。

   UNIX では、setEnv.sh コマンドを実行します。このコマンドは、*WL_HOME*/samples/server/config/examples ディレクトリにあります。*WL_HOME* は WebLogic プラットフォームのメイン ディレクトリです。

2. Trader ステートレス セッション EJB 用の Java インタフェースおよびクラスを記述します。

   3-4 ページの「EJB 用の Java コードの記述」を参照してください。

3. TradeResult 非組み込みデータ型の Java コードを記述します。

   3-9 ページの「非組み込みデータ型の Java コードの記述」を参照してください。

4. この Java コードをクラス ファイルにコンパイルします。

5. EJB デプロイメント記述子を作成します。

3-10 ページの「EJB デプロイメント記述子の作成」を参照してください。

6. EJB クラス ファイルおよびデプロイメント記述子を `trader.jar` アーカイブ ファイルにアセンブルします。

   3-11 ページの「EJB のアセンブル」を参照してください。

7. `build.xml` Ant ビルド ファイルを作成します。このファイルは、WebLogic Web サービスのアセンブルに使用する `servicegen` Ant タスクを実行します。

   3-12 ページの「build.xml Ant ビルド ファイルの作成」を参照してください。

8. ステージング ディレクトリを作成します。

9. EJB `trader.jar` ファイルおよび `build.xml` ファイルをステージング ディレクトリにコピーします。

10. Java Ant ユーティリティを実行して `Trader` Web サービスを `trader.ear` アーカイブ ファイルにアセンブルします。

    ```
    $ ant
    ```

11. テストするため、`trader.ear` アーカイブ ファイルを *`domain`*`/applications` ディレクトリにコピーして、`Trader` Web サービスを WebLogic Server に自動デプロイします。ここで、*`domain`* はユーザのドメインの場所を示しています。

12. ブラウザで次の URL を入力して `Trader` Web サービスのホーム ページを表示します。

    `http://`*`localhost`*`:`*`port`*`/webservice/TraderService`

    各要素の説明は次のとおりです。

    - *`localhost`* は、WebLogic Server が動作しているマシンの名前です。
    - *`port`* は、WebLogic Server がリスンしているポートです。

    Web サービスのホーム ページから、生成された WSDL を表示し、Web サービスが正しく機能するようテストすることができます。

Java クライアント アプリケーションから `Trader` Web サービスを呼び出す方法については、*`WL_HOME`*`\samples\server\src\examples\webservices\complex\statelessSession` ディレクトリにある `Client.java` ファイルを参照してください。

クライアントアプリケーションを作成、実行する方法については、*WL_HOME*\samples\server\src\examples\webservices\complex\statelessSession\package-summary.html Web ページをブラウザで呼び出してください。

EJB 用の Java コードの記述

サンプルの Trader ステートレス セッション EJB には、buy() と sell() という 2 つのパブリック メソッドがあります。Trader EJB では、入力として株式シンボルである String と購入または売却する株数である int を受け取る、buy() と sell() という 2 つのメソッドが定義されています。これらのメソッドはともに、TraderResult という非組み込みデータ型を返します。

次の Java コードは、Trader EJB のパブリック インタフェースです。

```
package examples.webservices.complex.statelessSession;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

/**
 * このインタフェース内のメソッドは TraderBean のパブリック インタフェース
 * これらのメソッドのシグネチャは EJBean のシグネチャと同じだが、
 * java.rmi.RemoteException を送出する点で異なる
 * EJBean はこのインタフェースを実装していないことに注意すること。対応する
 * コード生成の EJBObject である TraderBean は、このインタフェースを実装し
 * Bean に委託する
 *
 * @author Copyright (c) 1999-2002 by BEA Systems, Inc. All Rights Reserved.
 */

public interface Trader extends EJBObject {
  /**
   * 株式を購入する
   *
   * @param stockSymbol       String 株式シンボル
   * @param shares            int 売却する株数
   * @return                  TradeResult 取引結果
   * @exception               通信またはシステムに障害がある場合は
   *                          RemoteException を送出
   */
  public  TradeResult buy (String stockSymbol, int shares)
```

```
    throws RemoteException;
  /**
   * 株式を売却する
   *
   * @param stockSymbol       String 株式シンボル
   * @param shares            int 売却する株数
   * @return                  TradeResult 取引結果
   * @exception               通信またはシステムに障害がある場合は
   *                          RemoteException を送出
   */
  public TradeResult sell (String stockSymbol, int shares)
    throws RemoteException;
}
```

次の Java コードは、実際の ステートレス セッション EJB クラスです。

```
package examples.webservices.complex.statelessSession;

import javax.ejb.CreateException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 * TraderBean は、ステートレス セッション Bean。この Bean は以下を表す
 * <ul>
 * <li> セッション Bean への呼び出しと呼び出しの間に永続性はない
 * <li> 環境から値をルックアップ
 * </ul>
 *
 * @author Copyright (c) 1999-2002 by BEA Systems, Inc. All Rights Reserved.
 */
public class TraderBean implements SessionBean {

  private static final boolean VERBOSE = true;
  private SessionContext ctx;
  private int tradeLimit;

  // WebLogic のログ サービスの使用も検討すること
  private void log(String s) {
    if (VERBOSE) System.out.println(s);
  }

  /**
* このメソッドは EJB 仕様では必須だが、
* このサンプルでは使用されない
```

```
   *
   */
  public void ejbActivate() {
    log("ejbActivate called");
  }

  /**
* このメソッドは EJB 仕様では必須だが、
* このサンプルでは使用されない
   *
   */
  public void ejbRemove() {
    log("ejbRemove called");
  }

  /**
* このメソッドは EJB 仕様では必須だが、
* このサンプルでは使用されない
   *
   */
  public void ejbPassivate() {
    log("ejbPassivate called");
  }

  /**
* セッション コンテキストを設定
   *
   * @param ctx               SessionContext セッションのコンテキスト
   */
  public void setSessionContext(SessionContext ctx) {
    log("setSessionContext called");
    this.ctx = ctx;
  }

  /**
   * このメソッドは、ホーム インタフェース「WeatherHome.java」の create
   * メソッドに対応する
   * 2 つのメソッドのパラメータ セットは同じ。クライアントが
   * <code>TraderHome.create()</code> を呼び出すと、コンテナは、EJBean
   * のインスタンスを割り当てて <code>ejbCreate()</code> を呼び出す
   *
   * @exception               通信またはシステムに障害がある場合は
   *                          RemoteException を送出
   * @see                     examples.ejb11.basic.statelessSession.Trader
   */
  public void ejbCreate () throws CreateException {
    log("ejbCreate called");
    try {
```

```
      InitialContext ic = new InitialContext();
      Integer tl = (Integer) ic.lookup("java:/comp/env/tradeLimit");
      tradeLimit = tl.intValue();
    } catch (NamingException ne) {
      throw new CreateException("Failed to find environment value "+ne);
    }
  }

  /**
   * 指名された顧客に代わって株式を購入
   *
   * @param customerName      String 顧客の名前
   * @param stockSymbol       String 株式シンボル
   * @param shares            int 売却する株数
   * @return                  TradeResult 取引結果
   *                          株式の購入中にエラーがある場合
   */
  public TradeResult buy(String stockSymbol, int shares) {
    if (shares > tradeLimit) {
      log("Attempt to buy "+shares+" is greater than limit of "+tradeLimit);
      shares = tradeLimit;
    }
    log("Buying "+shares+" shares of "+stockSymbol);
    return new TradeResult(shares, stockSymbol);
  }

  /**
   * 指名された顧客に代わって株式を売却
   *
   * @param customerName      String 顧客の名前
   * @param stockSymbol       String 株式シンボル
   * @param shares            int 売却する株数
   * @return                  TradeResult 取引結果
   *                          株式の売却中にエラーがある場合
   */
  public TradeResult sell(String stockSymbol, int shares) {
    if (shares > tradeLimit) {
      log("Attempt to sell "+shares+" is greater than limit of "+tradeLimit);
      shares = tradeLimit;
    }
    log("Selling "+shares+" shares of "+stockSymbol);
    return new TradeResult(shares, stockSymbol);
  }
}
```

次の Java コードは、Trader EJB のホーム インタフェースです。

```
package examples.webservices.complex.statelessSession;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

/**
 * このインタフェースは、TraderBean.java のホーム インタフェース
 * WebLogic では、コード生成のコンテナ クラスである TraderBeanC
 * によって実装される。ホーム インタフェースは、EJBean で「ejbCreate」
 * というメソッドに必ず対応する 1 つまたは複数の create メソッドをサポートできる
 *
 * @author Copyright (c) 1998-2002 by BEA Systems, Inc. All Rights Reserved.
 */
public interface TraderHome extends EJBHome {
  /**
   * このメソッドは、「WeatherBean.java」の ejbCreate
   * メソッドに対応する
   * 2 つのメソッドのパラメータ セットは同じ。クライアントが
   * <code>TraderHome.create()</code> を呼び出すと、コンテナは
   * EJBean のインスタンスを割り当てて <code>ejbCreate()</code> を呼び出す
   *
   * @return                    Trader
   * @exception                 通信またはシステムに障害がある場合は
   *                            RemoteException を送出
   * @exception                 Bean の作成時に問題が生じた場合は
   *                            CreateException を送出
   * @see                       examples.ejb11.basic.statelessSession.TraderBean
   */
  Trader create() throws CreateException, RemoteException;
}
```

非組み込みデータ型の Java コードの記述

EJB の 2 つのメソッドは、TraderResult という非組み込みデータ型を返します。次の Java コードは、この型を記述しています。

```
package examples.webservices.complex.statelessSession;

import java.io.Serializable;

/**
 * このクラスは、売買結果を反映する
 *
 * @author Copyright (c) 1999-2002 by BEA Systems, Inc. All Rights Reserved.
 */
public final class TradeResult implements Serializable {

  // 実際に売買された株数
  private int    numberTraded;

  private String stockSymbol;

  public TradeResult() {}

  public TradeResult(int nt, String ss) {
    numberTraded = nt;
    stockSymbol  = ss;
  }

  public int getNumberTraded() { return numberTraded; }

  public void setNumberTraded(int numberTraded) {
        this.numberTraded = numberTraded;
  }

  public String getStockSymbol() { return stockSymbol; }

  public void setStockSymbol(String stockSymbol) {
        this.stockSymbol = stockSymbol;
  }
}
```

EJB デプロイメント記述子の作成

BEA XML エディタを使用した ejb-jar.xml ファイルおよび weblogic-ejb-jar.xml ファイルの作成および編集については、1-15 ページの「XML ファイルの編集」を参照してください。

次の例は、Trader EJB に関する ejb-jar.xml デプロイメント記述子を示しています。

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

<!DOCTYPE ejb-jar PUBLIC
'-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN'
'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>TraderService</ejb-name>
      <home>examples.webservices.complex.statelessSession.TraderHome</home>
      <remote>examples.webservices.complex.statelessSession.Trader</remote>

<ejb-class>examples.webservices.complex.statelessSession.TraderBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <env-entry>
        <env-entry-name>WEBL</env-entry-name>
        <env-entry-type>java.lang.Double </env-entry-type>
        <env-entry-value>10.0</env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>INTL</env-entry-name>
        <env-entry-type>java.lang.Double </env-entry-type>
        <env-entry-value>15.0</env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>tradeLimit</env-entry-name>
        <env-entry-type>java.lang.Integer </env-entry-type>
        <env-entry-value>500</env-entry-value>
      </env-entry>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
```

```
        <ejb-name>TraderService</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

次の例は、Trader EJB に関する weblogic-ejb-jar.xml デプロイメント記述子を示しています。

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

<!DOCTYPE weblogic-ejb-jar PUBLIC
  '-//BEA Systems, Inc.//DTD WebLogic 7.0.0 EJB//EN'
'http://www.bea.com/servers/wls700/dtd/weblogic700-ejb-jar.dtd'>

<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>TraderService</ejb-name>
    <jndi-name>webservices-complex-statelessession</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

EJB のアセンブル

EJB クラス ファイルおよびデプロイメント記述子を trader.jar アーカイブ ファイルにアセンブルするには、次の手順に従います。

1. 一時的なステージング ディレクトリを作成します。
2. コンパイル済みの Java EJB クラス ファイルをステージング ディレクトリにコピーします。
3. ステージング ディレクトリに META-INF サブディレクトリを作成します。
4. META-INF サブディレクトリに ejb-jar.xml および weblogic-ejb-jar.xml デプロイメント記述子をコピーします。
5. jar ユーティリティを使用して pre_trader.jar アーカイブ ファイルを作成します。

   ```
   jar cvf pre_trader.jar -C staging_dir .
   ```

6. weblogic.ejbc ユーティリティを実行して、EJB 2.0 および 1.1 のコンテナクラスを生成、コンパイルして、最終的に trader.jar ファイルを作成します。

   ```
   java weblogic.ejbc pre_trader.jar trader.jar
   ```

build.xml Ant ビルド ファイルの作成

Ant ビルド ファイル build.xml には、servicegen Ant タスクへの呼び出しが入っています。このタスクは、trader.jar EJB ファイルを参照し、すべてのデータ型コンポーネント (シリアライゼーション クラスなど) を生成し、web-services.xml デプロイメント記述子ファイルを作成し、そのすべてをデプロイ可能な trader.ear ファイルにパッケージ化します。

次の build.xml ファイルには、EAR ファイルを一時的な build_dir ディレクトリに作成する指示が記述されています。

```
<project name="webServicesExample" default="build">
  <target name="build" >
    <delete dir="build_dir" />
    <mkdir dir="build_dir" />
    <copy todir="build_dir" file="trader.jar"/>
    <servicegen
      destEar="build_dir/trader.ear"
      warName="trader.war"
      contextURI="webservice">
      <service
         ejbJar="build_dir/trader.jar"
         targetNamespace="http://www.bea.com/examples/Trader"
         serviceName="TraderService"
         serviceURI="/TraderService"
         generateTypes="True"
         expandMethods="True" >
      </service>
    </servicegen>
  </target>
</project>
```

4　WebLogic Web サービスの設計

この章では、WebLogic Web サービスの実装に先立って検討すべき設計の問題について説明します。

同期オペレーション、非同期オペレーションのいずれかの選択

WebLogic Web サービス オペレーションは、同期要求応答方式または非同期一方向方式をとります。

同期要求応答方式 (デフォルトの動作) では、クライアント アプリケーションは、Web サービスを呼び出すたびに SOAP 応答を受け取ります。そのオペレーションを実装するメソッドが `void` を返す場合も同様です。非同期一方向方式では、障害や例外が発生した場合も含め、クライアントが SOAP 応答を受け取ることはありません。

このタイプの動作は、`web-services.xml` ファイルにある `<operation>` の `invocation-style` 属性を使用して指定します。

Web サービス オペレーションは一般的には同期要求応答方式で、典型的な RPC 方式の動作をミラーリングします。クライアントアプリケーションが、エラー発生時も含めて応答を受け取る必要がない場合は、非同期動作を実装した方がいい場合があります。非同期一方向 Web サービス オペレーションは、次の事項を念頭において設計してください。

- このオペレーションを実装するバックエンド コンポーネントは、明示的に `void` を返す必要があります。
- このオペレーションに対して out または inout パラメータを指定することはできません。指定できるのは入力パラメータのみです。

Web サービスのバックエンド コンポーネントの選択

WebLogic Web サービス オペレーションは、次のいずれかのタイプのバックエンド コンポーネントを使用して実装します。

- ステートレス セッション EJB のメソッド
- Java クラスのメソッド
- JMS メッセージのコンシューマまたはプロデューサ。詳細については、第 12 章「JMS で実装された WebLogic Web サービスの作成」を参照してください。

EJB バックエンド コンポーネント

ステートレス セッション EJB のメソッドを使用して実装された Web サービス オペレーションは、インタフェース駆動です。つまり、オペレーションが依拠しているステートレス セッション EJB のビジネス メソッドによって Web サービス オペレーションの動作が決定されます。クライアントが Web サービス オペレーションを呼び出すと、パラメータ値がメソッドに送信され、そのメソッドが実行されて戻り値が送り返されます。

ステートレス セッション EJB バックエンド コンポーネントは、アプリケーションが以下の特徴を備えている場合に使用してください。

- Web サービスの動作をインタフェースとして表現できる場合
- Web サービスがデータ指向でなく処理指向である場合
- 永続性、セキュリティ、トランザクション、同時実行性などの EJB の機能から Web サービスが恩恵を得られる場合。

このタイプの Web サービス オペレーションの実装例には、特定の地域の現在の天気情報の提供、指定株の現在の株価の表示、またはビジネス トランザクションが完了する前の取引先の信用状態のチェックなどがあります。

Java クラスのバックエンド コンポーネント

Java クラスを使用して実装された Web サービス オペレーションは、EJB メソッドを使用して実装されたオペレーションと類似しています。多くの場合、Java クラスの作成の方が EJB の作成より簡素であり、より短時間で作成できます。永続性、セキュリティ、トランザクション、同時実行性といった EJB 機能のオーバーヘッドが不要な場合は、Java クラスをバックエンド コンポーネントとして使用します。

ただし、Java クラスをバックエンド コンポーネントとして使用するには制限があります。詳細については、5-4 ページの「Java クラスを記述して Web サービスを実装する」を参照してください。

RPC 指向またはドキュメント指向の Web サービス

WebLogic Web サービスのオペレーションは、RPC 指向またはドキュメント指向のいずれかです。WSDL 1.1 仕様で説明されているように、RPC 指向のオペレーションでは SOAP メッセージにパラメータと戻り値が格納され、ドキュメント指向のオペレーションでは SOAP メッセージに XML ドキュメントが格納されます。

ドキュメント指向の WebLogic Web サービス オペレーションを実装するメソッドは、パラメータを 1 つだけ持つことができます (データ型はサポートされているものであればどれでもかまわない)。RPC 指向のオペレーションでは、パラメータの数に制限はありません。

RPC 指向の WebLogic Web サービス オペレーションでは、SOAP エンコーディングを使用します。ドキュメント指向の WebLogic Web サービス オペレーションでは、リテラル エンコーディングを使用します。

1 つの WebLogic Web サービスのすべてのオペレーションは、RPC 指向かドキュメント指向のいずれかでなければなりません。WebLogic Server では、同じ Web サービスで 2 つのスタイルをミックスすることはできません。

デフォルトでは、WebLogic Web サービスのオペレーションは RPC 指向です。オペレーションをドキュメント指向にするには、`servicegen` Ant タスクを使用して Web サービスをアセンブルするときに `<service>` 要素の `style="document"` 属性を使用します。生成される `web-services.xml` デプロイメント記述子では、適切な `<web-service>` 要素に、対応する `style="document"` 属性が設定されます。

ドキュメント指向の WebLogic Web サービスの実装については、5-6 ページの「ドキュメント指向の Web サービスの実装」を参照してください。`servicegen` Ant タスクを使用してドキュメント指向の Web サービスをアセンブルする方法の詳細については、6-3 ページの「servicegen Ant タスクを使用した WebLogic Web サービスのアセンブル」と B-19 ページの「servicegen」を参照してください。

データ型

WebLogic Web サービスは、Web サービス オペレーションのパラメータと戻り値として組み込みデータ型および非組み込みデータ型をサポートします。つまり、WebLogic Web サービスは、XML スキーマを使用して表現できるあらゆるデータ型を扱えます。

組み込みデータ型は、JAX-RPC 仕様で指定されているデータ型です。Web サービスが組み込みデータ型のみを使用する場合は、XML 表現と Java 表現間の変換は WebLogic Server によって自動的に処理されます。組み込みデータ型の詳細なリストは、5-13 ページの「組み込みデータ型の使用法」を参照してください。

より複雑で、非組み込みデータ型にパラメータや戻り値を使用する Web サービス オペレーションの場合は、以下を実行する必要があります。

a. XML 表現と Java 表現間のデータ変換を行うシリアライゼーション クラスを作成する

b. `web-services.xml` ファイルにあるデータ型の XML 表現を記述する (XML スキーマの表記法で)

c. そのデータ型の Java クラスファイルを作成する

d. `web-services.xml` ファイルにあるデータ型マッピングを記述する

WebLogic Server には、広く使用されているさまざまな XML および Java データ型でこれらのタスクを実行する Ant タスクが組み込まれています。この機能は、オートタイピングと呼ばれます。サポートされている非組み込みデータ型のリストは、6-14 ページの「servicegen および autotype Ant タスクでサポートされる非組み込みデータ型」を参照してください。Ant タスクの実行については、第 6 章「Ant タスクを使用した WebLogic Web サービスのアセンブル」を参照してください。

注意： オートタイピング Ant タスクを使用して Java クラスのデータ型情報を生成する場合、クラスは 5-5 ページの「非組み込みデータ型を実装する」のガイドラインに準拠している必要があります。

データ型が組み込みデータ型でもサポートされている非組み込みデータ型でもない場合は、シリアライゼーション クラスの作成などを手作業で行う必要があります。詳細については、第 9 章「非組み込みデータ型の使用法」を参照してください。

SOAP メッセージをインターセプトする SOAP メッセージ ハンドラの使用

Web サービスの中には、SOAP メッセージへのアクセスを必要とするものがあり、SOAP メッセージに対して SOAP メッセージ ハンドラを作成できます。

SOAP メッセージ ハンドラは、Web サービスの要求と応答の両方で SOAP メッセージをインターセプトするメカニズムです。ハンドラは、Web サービスそのものと Web サービスを呼び出すクライアント アプリケーションの両方で作成することができます。

簡単なハンドラ使用例としては、SOAP メッセージの本文でセキュアなデータを暗号化、復号化する場合があります。クライアント アプリケーションは、ハンドラを使用してデータを暗号化してから、Web サービスに SOAP 要求メッセージを送信します。Web サービスは、要求を受け取るとハンドラを使用してデータを復号化してから、そのデータを Web サービスを実装するバックエンド コンポーネントに送信します。SOAP 応答メッセージの場合は、同じ手順が逆の順序で実行されます。

別の例には、SOAP メッセージのヘッダー部分にある情報へのアクセスがあります。SOAP のヘッダーを使用して Web サービス固有の情報を格納しておいて、ハンドラを使用してその情報を操作することができます。

SOAP メッセージ ハンドラは、Web サービスのパフォーマンスを向上させるために使用することもできます。Web サービスがデプロイされてしばらくすると、同じパラメータを使用して Web サービスを呼び出すコンシューマが多いことに気付く場合があります。よく使用される Web サービス呼び出しの結果 (静的な結果を前提として) をキャッシュしておき、Web サービスを実装するバックエンド コンポーネントを呼び出すことなく、適宜、キャッシュしておいた結果をただちに返すことによって、Web サービスのパフォーマンスを向上させることができます。このパフォーマンスの向上は、ハンドラを使用して、SOAP 要求メッセージに頻繁に使用されるパラメータが含まれていないかチェックすることで達成されます。

ステートフル WebLogic Web サービス

WebLogic Web サービス オペレーションは、ステートレス セッション EJB または Java クラスを使用して実装するので、WebLogic Web サービス オペレーションはステートフルではありません。また、標準の要求応答モデルの範疇を超えて対話ができるサービスでもありません。

ただし、JDBC またはエンティティ Bean を使用して、対話が可能な Web サービスを模倣することはできます。たとえば、サービスを呼び出したクライアント アプリケーションが、自らを識別する一意の ID をステートレス セッション EJB のエントリ ポイントに渡すように、Web サービスを設計することができます。この EJB は、ID を使用して、何らかのタイプの永続ストレージで、エンティティ Bean または JDBC を使用して対話を続行します。このクライアント アプリケーションが再び同じ Web サービスを呼び出すと、ステートレス セッション EJB は、この一意の ID を使用して永続データを選択することによって、対話の前回のステートを復元することができます。

エンティティ Bean のプログラミングについては、『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』を参照してください。JDBC については、「WebLogic jDrivers」を参照してください。

5 WebLogic Web サービスの実装

この章では、WebLogic Web サービスを実装する方法について説明します。

WebLogic Web サービスの実装の概要

WebLogic Web サービスの実装とは、Web サービスを構成するバックエンド コンポーネントの Java コードを記述し、必要に応じて SOAP メッセージ ハンドラを作成することです。バックエンド コンポーネントには、ステートレス セッション EJB、Java クラス、および JMS メッセージ コンシューマ / プロデューサがあります。Web サービスは、これらのコンポーネントをさまざまに組み合わせて実装することができます。

単一の WebLogic Web サービスは、1 つまたは複数のオペレーションで構成されています。各オペレーションは、異なるバックエンド コンポーネントと SOAP メッセージ ハンドラのメソッドを使用して実装することができます。たとえば、オペレーションは、ステートレス セッション EJB の単一メソッドを使用して実装されている場合、または SOAP メッセージ ハンドラとステートレス セッション EJB を組み合わせて実装されている場合があります。

既存の WSDL ファイルから WebLogic Web サービスを実装する場合は、WebLogic Server に付属の `wsdl2Service` Ant タスクを利用して、必要な Java ソース コードの多くを生成できます。この生成されたファイルを土台として使用し、Web サービスを希望通りに機能させるために必要な残りの Java コードを追加できます。

ここでは、第 4 章「WebLogic Web サービスの設計」で説明した設計の問題を読み理解して Web サービスを設計していること、コード化する必要があるコンポーネントのタイプを基本的に理解していることを前提としています。

WebLogic Web サービスの実装：主な手順

以下は、WebLogic Web サービスを実装する高度な手順です。このマニュアルの後半で、この手順について詳細に説明します。実装する Web サービスのタイプにより、必須の手順もあれば省略可能なものもあります。

1. Web サービスを構成するバックエンド コンポーネントの Java コードを記述します。

   5-2 ページの「コンポーネントの Java コードの記述」を参照してください。

2. SOAP の要求メッセージまたは応答メッセージにある情報を処理したり、SOAP の添付ファイルにアクセスしたりする必要がある場合は、SOAP メッセージ ハンドラおよびハンドラ チェーンを作成します。

   第 10 章「SOAP メッセージをインターセプトする SOAP メッセージ ハンドラの作成」を参照してください。

3. バックエンド コンポーネントが、パラメータまたは戻り値として非組み込みデータ型を使用する場合は、XML と Java 間のデータ変換を行うシリアライゼーション クラスを生成、または作成します。

   5-5 ページの「非組み込みデータ型を実装する」を参照してください。

コンポーネントの Java コードの記述

WebLogic Web サービスを実装するときには、以下のいずれかのバックエンド コンポーネントの Java コードを記述します。

- スレートレス セッション EJB

  Java コードの記述については、5-4 ページの「ステートレス セッション EJB を使用して Web サービスを実装する」を参照してください。例については、3-4 ページの「EJB 用の Java コードの記述」を参照してください。

- Java クラス

  Java コードの記述については、5-4 ページの「Java クラスを記述して Web サービスを実装する」を参照してください。

- JMS メッセージのコンシューマまたはプロデューサ (一般的にはメッセージ駆動型 Bean)

  第 12 章「JMS で実装された WebLogic Web サービスの作成」を参照してください。

Web サービス オペレーションで非組み込みデータ型がパラメータまたは戻り値として使用される場合は、5-5 ページの「非組み込みデータ型を実装する」を参照してください。

RPC 指向ではなくドキュメント指向のオペレーションを使用する Web サービスを実装する場合は、5-6 ページの「ドキュメント指向の Web サービスの実装」を参照してください。

既存の WSDL ファイルに基づいて WebLogic Web サービスを実装し、かつその Web サービスを Java クラスを使用して実装する場合には、WebLogic Server に付属の `wsdl2Service` Ant タスクを利用することで必要な Java コードの多くを生成できます。この Ant タスクを使用する方法の詳細については、5-7 ページの「WSDL ファイルに基づく部分的な実装の生成」を参照してください。

Web サービス実装からの例外の送出については、5-11 ページの「SOAP 障害例外を送出する」を参照してください。

Web サービスのオペレーションで複数の値が返されるようにするには、5-8 ページの「複数の戻り値の実装」を参照してください。

ステートレス セッション EJB を使用して Web サービスを実装する

Web サービスのステートレス セッション EJB の Java コードを記述することは、以下の問題を除けば、スタンドアロン EJB の記述と比べて特に相違はありません。

- `web-services.xml` デプロイメント記述子で、Web サービス オペレーションが一方向となるように指定することができます。それにより、Web サービスを呼び出すクライアント アプリケーションは応答を求めません。このタイプのオペレーションを実装する EJB メソッドの Java コードを記述するときは、このメソッドから `void` が返されるように指定する必要があります。

  `web-services.xml` ファイルで Web サービス オペレーションが一方向となるように指定する方法の詳細は、A-10 ページの「operation」を参照してください。

- EJB メソッドのパラメータまたは戻り値のデータ型が組み込みデータ型のセットに含まれていない場合は、XML 表現と Java 表現間でデータ変換を行うシリアライゼーション クラスを生成、または作成する必要があります。組み込みデータ型のリストは、5-13 ページの「組み込みデータ型の使用法」を参照してください。

  5-5 ページの「非組み込みデータ型を実装する」を参照してください。

ステートレス セッション EJB の記述例については、3-4 ページの「EJB 用の Java コードの記述」を参照してください。一般的な情報については、『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』を参照してください。

Java クラスを記述して Web サービスを実装する

以下の規則に従い、Java クラスを使用して Web サービス オペレーションを実装することができます。

- スレッドを開始しないこと。この規則は、WebLogic Web Server で実行されるすべての Java コードに適用されます。
- デフォルトの引数のないコンストラクタを定義すること。

- Web サービス オペレーションとして公開される Java クラスのメソッドはパブリック メソッドとして定義すること。
- スレッド セーフの Java コードを記述すること。WebLogic Server は、Web サービス オペレーションを実装する Java クラスのインスタンスを 1 つのみ維持し、Web サービスの呼び出しはすべて同じインスタンスを使用するからです。

Java クラスを使用して WebLogic Web サービス オペレーションを実装する例は、*WL_HOME*\samples\server\src\examples\webservices\basic\javaclass ディレクトリにあります。*WL_HOME* は、WebLogic Server のメインのインストール ディレクトリです。

非組み込みデータ型を実装する

ステートレス セッション EJB あるいは Java クラスは、パラメータや戻り値として必ずしも組み込みデータ型を受け取るとは限りません。自作の Java データ型を使用する場合もあります。非組み込みデータ型の例は `TradeResult` です。このデータ型には、文字列の株式シンボルと売買される株数を表す整数の 2 つのフィールドがあります。組み込みデータ型のリストは、5-13 ページの「組み込みデータ型の使用法」を参照してください。

バックエンド コンポーネントが、パラメータまたは戻り値として非組み込みデータ型を使用する場合は、XML と Java 間のデータ変換を行うシリアライゼーション クラスを生成または作成する必要があります。それには、以下の 2 通りの方法があります。

- WebLogic Server の `servicegen` または `autotype` Ant タスクを使用して、EJB を参照し、シリアライゼーション クラスを自動的に生成させる方法。これらの Ant タスクは、データが XML フォーマットで表現されるように、対応する XML スキーマも作成し、該当するデータ型マッピング情報を使用して、`web-services.xml` デプロイメント記述子ファイルを更新します。6-5 ページの「servicegen Ant タスクを実行する」および 6-9 ページの「autotype Ant タスクを実行する」で説明する Web サービスのアセンブル作業の一環としてこれらの Ant タスクを実行します。

  警告： `autotype`、`servicegen`、および `clientgen` Ant タスクによって生成されたシリアライザ クラスと Java および XML 表現はラウンドトリッ

プできません。詳細については、6-18 ページの「生成されたデータ型コンポーネントの非ラウンドトリップ」を参照してください。

- 手動で、シリアライゼーション クラス、使用するデータ型の XML 表現、および Java 表現を作成する方法。この方法の方が、Ant タスクを使用して生成する方法より複雑で、時間がかかります。非組み込みデータ型を手動で処理する方法の詳細については、第 9 章「非組み込みデータ型の使用法」を参照してください。

Java データ型の XML 表現およびシリアライゼーション クラスを手動で作成する場合は、変換コードもすべて自分で記述するので、Java クラスを自在にコード化することができます。

`servicegen` または `autotype` Ant タスクを使用してデータ型のコンポーネントを自動的に生成させる場合は、そのデータ型の Java クラスを記述するときに、以下の要件に従います。

- デフォルト コンストラクタを定義する。これは、パラメータを受け取らないコンストラクタです。
- 公開しようとする各メンバー変数に対して `getXXX()` メソッドと `setXXX()` メソッドの両方を定義します。
- 公開される各メンバー変数のデータ型は、組み込みデータ型のいずれかか、または組み込みデータ型で構成され、対応するシリアライゼーション クラスと XML スキーマ表現を持つ非組み込みデータ型とします。

`servicegen` および `autotype` Ant タスクでは、ほとんどの一般的な XML および Java データ型のデータ型コンポーネントを生成できます。サポートされている非組み込みデータ型のリストは、6-14 ページの「servicegen および autotype Ant タスクでサポートされる非組み込みデータ型」を参照してください。

ドキュメント指向の Web サービスの実装

WebLogic Web サービスを作成するときには、その Web サービスがドキュメント指向 (SOAP メッセージにドキュメントが格納される) または RPC 指向 (SOAP メッセージにパラメータと戻り値が格納される) のどちらであるのかを指定できます。

ドキュメント指向の Web サービスを作成する場合は、以下の点に注意してください。

- Web サービスの各オペレーションを実装するメソッドでは、パラメータを 1 つしか使用できない。その 1 つのパラメータのデータ型は、サポートされているものならどれでもかまいません。詳細については、4-4 ページの「データ型」を参照してください。
- 各オペレーションを実装するメソッドでは、out パラメータおよび inout パラメータを使用できない。

WSDL ファイルに基づく部分的な実装の生成

`wsdl2Service` Ant タスクは既存の WSDL ファイルを入力とし、部分的に Web サービスを実装する Java ソース ファイルを生成します。

Java ソース ファイルには、Java クラスのみで実装される WebLogic Web サービスのテンプレートが格納されます。このテンプレートには、WSDL ファイルのオペレーションに対応する完全なメソッド シグネチャが含まれています。5-4 ページの「Java クラスを記述して Web サービスを実装する」のガイドラインに従って、希望通りに機能するようにそれらのメソッドの実際のコードを記述します。

`wsdl2Service` Ant タスクは、(`<service>` 要素で指定された) WSDL ファイルの 1 つの Web サービスのみで部分的な実装を生成します。`serviceName` 属性を使用すると、特定のサービスを指定できます。この属性を指定しない場合、`wsdl2Service` Ant タスクでは WSDL ファイルの最初の `<service>` 要素の部分的な実装が生成されます。

wsdl2Service Ant タスクの実行

`wsdl2Service` Ant タスクを実行するには、次の手順に従います。

1. `wsdl2Service` Ant タスクへの呼び出しの入った `build.xml` というファイルを作成します。詳細については、5-8 ページの「wsdl2Service Ant タスクのサンプル build.xml ファイル」を参照してください。
2. 環境を設定します。

   Windows NT では、`setWLSEnv.cmd` コマンドを実行します。*`WL_HOME`*`\server\bin` ディレクトリにあります。*`WL_HOME`* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

UNIX では、setWLSEnv.sh コマンドを実行します。*WL_HOME*/server/bin ディレクトリにあります。*WL_HOME* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

3. build.xml ファイルと同じディレクトリで ant と入力し、Ant タスクまたは build.xml ファイルで指定されたタスクを実行します。

```
prompt> ant
```

wsdl2Service Ant タスクのリファレンス情報については、B-33 ページの「wsdl2Service」を参照してください。

wsdl2Service Ant タスクのサンプル build.xml ファイル

以下は、簡単な build.xml ファイルの例です。

```
<project name="buildWebservice" default="generate-from-WSDL">
   <target name="generate-from-WSDL">
      <wsdl2service
           wsdl="c:\wsdls\myService.wsdl"
           destDir="c:\myService\implementation"
           typeMappingFile="c:\autotype\types.xml"
           packageName="example.ws2j.service" />
   </target>
</project>
```

例の中の wsdl2Service Ant タスクでは、WSDL ファイル c:\wsdls\myService.wsdl の最初の <service> 要素の Java ソース ファイルが生成されます。非組み込みデータ型のデータ型マッピング情報は、c:\autotype\types.xml ファイルから利用します。このファイルは、通常は事前に autotype Ant タスクを実行して生成されています。生成された Java ソース ファイルは、パッケージ example.ws2j.service にあります。

複数の戻り値の実装

WebLogic Web サービス オペレーションが返す値は一般的には 1 つで、その Web サービス オペレーションを実装する EJB メソッドまたは Java クラス メソッドの戻り値を返します。Web サービスのオペレーションで複数の値が返されるようにする場合は、以下のことができます。

- 戻り値のデータ型を、複数の部分または配列を持つオブジェクトなどの複合型として定義する

- Web サービス オペレーションの 1 つまたは複数のパラメータが、out パラメータまたは inout パラメータとなるように指定する

out および inout パラメータは、オペレーションのパラメータがパラメータと戻り値の両方の標準として機能できるメカニズムです。out パラメータは、オペレーションの呼び出し時には未定義ですが、オペレーションの完了時にはそのオペレーションを実装するメソッドによって定義されます。inout パラメータは、呼び出し時および完了時に定義されます。たとえば、Web サービス オペレーションに out パラメータがあり、そのオペレーションは EJB メソッドを使用して実装されるとします。この EJB メソッドは、out パラメータの値を設定し、オペレーションを呼び出したクライアント アプリケーションにこの値を返します。すると、クライアントアプリケーションは、戻り値の場合と同様に、この out パラメータの値にアクセスすることができます。inout パラメータは、メソッドに情報を送信する標準の in パラメータと out パラメータの両方の機能を備えたパラメータです。この節では、out または inout パラメータを使用する EJB メソッドまたは Java クラス メソッドを使用して Web サービスを実装する方法について説明します。

次の例は、第 2 のパラメータとして inout パラメータを持つメソッドを示しています。

```
public String myMethod( String param1,
        javax.xml.rpc.holders.IntHolder intHolder ) {

System.out.println ("The input value is: " + intHolder.value );
intHolder.value = 20;  // out パラメータの新しい値

return param1;
}
```

このメソッドは、文字列と整数の 2 つのパラメータを使用して呼び出します。このメソッドは、文字列 (標準の戻り値) と整数 (`IntHolder` ホルダ パラメータを介して) という 2 つの値を返します。

out パラメータおよび inout パラメータは、`javax.xml.rpc.holders.Holder` インタフェースを実装する必要があります。`Holder.value` フィールドを使用して、まず、inout パラメータの入力値にアクセスし、次に、out および inout パラメータの値を設定します。先の例で、第 2 のパラメータに値 40 を指定してメソッドを呼び出したものとします。このメソッドの完了時には、`intHolder` 値は 20 となります。

ホルダ クラスを使用した複数の戻り値の実装

out または inout パラメータが標準のデータ型である場合は、次の表に記載する JAX-RPC のホルダ クラスのいずれかを使用することができます。

表 5-1 WebLogic Server で使用できる組み込みホルダ クラス

組み込みホルダ クラス	保持される Java データ型
javax.xml.rpc.holders.BooleanHolder	boolean
javax.xml.rpc.holders.ByteHolder	byte
javax.xml.rpc.holders.ShortHolder	short
javax.xml.rpc.holders.IntHolder	int
javax.xml.rpc.holders.LongHolder	long
javax.xml.rpc.holders.FloatHolder	float
javax.xml.rpc.holders.DoubleHolder	double
javax.xml.rpc.holders.BigDecimalHolder	java.math.BigDecimal
javax.xml.rpc.holders.BigIntegerHolder	java.math.BigInteger
javax.xml.rpc.holders.ByteArrayHolder	byte[]
javax.xml.rpc.holders.CalendarHolder	java.util.Calendar
javax.xml.rpc.holders.QnameHolder	javax.xml.namespace.QName
javax.xml.rpc.holders.StringHolder	java.lang.String

パラメータのデータ型が用意されていない場合は、自分で実装を作成する必要があります。

自分で javax.xml.rpc.holders.Holder インタフェースの実装を作成するには、次のガイドラインに従います。

- *Type*Holder という実装クラスを指名します。ここで、*Type* は、複合型の名前です。たとえば、複合型の名前が Person なら、実装クラスの名前は PersonHolder となります。

- Holder 実装クラスは、保持しているクラスのパッケージの下にある holders サブ パッケージ内にパッケージ化する必要があります。

  たとえば、複合型 Person が examples.webservices パッケージ内にある場合、PersonHolder 実装クラスは examples.webservices.holders パッケージ内にパッケージ化します。

- value というパブリック フィールドを作成します。そのデータ型はパラメータと同じです。
- value フィールドをデフォルト値に初期化するデフォルト コンストラクタを作成します。
- value フィールドを渡されたパラメータの値に設定するコンストラクタを作成します。

次の例は、PersonHolder 実装クラスの概略を示しています。

```
package examples.webservices.holders;

public final class PersonHolder implements
            javax.xml.rpc.holders.Holder {

   public Person value;

   public PersonHolder() {

       //value 変数をデフォルト値に設定
   }

   public PersonHolder (Person value) {

       //value 変数を渡された値に設定
   }

}
```

SOAP 障害例外を送出する

ステートレス セッション EJB または Java クラスで javax.xml.rpc.soap.SOAPFaultException 例外を送出すると、WebLogic Server は、その例外を SOAP 障害にマッピングし、そのオペレーションを呼び出すクライアント アプリケーションに送信します。

次の抜粋は、SOAPFaultException クラスを記述しています。

```
public class SOAPFaultException extends java.lang.RuntimeException {
    public SOAPFaultException (QName faultcode,
                               String faultstring,
                               String faultactor,
                               javax.xml.soap.Detail detail ) {...}
    public Qname getFaultCode() {...}
    public String getFaultString() {...}
    public String getFaultActor() {...}
    public javax.xml.soap.Detail getDetail() {...}
}
```

EJB または Java クラスが他のタイプの Java 例外を送出する場合、WebLogic Server は、それをできるだけ SOAP 障害にマップしようとします。とはいえ、クライアント アプリケーションが最も有効な例外情報を受け取ることを保証するためには、`SOAPFaultException` 例外またはこの例外を拡張する例外を明示的に送出する必要があります。

`weblogic.webservice.util.FaultUtil.newDetail()` WebLogic Web サービスの API を使用して `javax.xml.soap.Detail` オブジェクトを作成してください。これは、アプリケーション固有の詳細なエラー情報を提供する、`DetailEntry` オブジェクトの JAX-RPC コンテナです。`javax.xml.soap.Detail.addDetailEntry()` メソッドを使用すると、`DetailEntry` を `Detail` オブジェクトに追加できます。

以下は、Web サービスの実装内から `SOAPFaultException` を作成および送出する場合の例です。

```
    throw new SOAPFaultException(
        new QName( "http://schemas.xmlsoap.org/soap/envelope/", "Server" ),
        "info on the fault",
        "info on the actor",
        weblogic.webservice.util.FaultUtil.newDetail() );
  }
```

警告： (`SOAPFaultException` を使用せずに) 独自の例外を作成して送出する場合、例外クラスのプロパティの 2 つ以上が同じデータ型であれば、JAX-RPC 仕様では要求されていませんが、これらのプロパティのセッター メソッドも作成する必要があります。これは、WebLogic Web サービスで SOAP メッセージ内の例外を受け取って XML を Java 例外クラスに変換する場合、対応するセッター メソッドがないと、どの XML 要素がどのクラス プロパティにマップするのかが分からないからです。

組み込みデータ型の使用法

次の節では、WebLogic Web サービスがサポートする組み込みデータ型およびその XML 表現と Java 表現間のマッピングについて説明します。Web サービスを実装するバックエンド コンポーネントのパラメータと戻り値のデータ型が、組み込みデータ型のセットの中にある限り、データは WebLogic Server によって、XML と Java 間で自動的に変換されます。

一方、非組み込みデータ型を使用する場合は、データの XML、Java 間のデータ変換を行うシリアライゼーション クラスを作成する必要があります。WebLogic Server には、ほとんどの非組み込みデータ型のシリアライゼーション クラスを生成できる `servicegen` および `autotype` Ant タスクが用意されています。サポートされている XML および Java データ型のリストは、6-14 ページの「servicegen および autotype Ant タスクでサポートされる非組み込みデータ型」を参照してください。`servicegen` および `autotype` の使い方の詳細については、第 6 章「Ant タスクを使用した WebLogic Web サービスのアセンブル」を参照してください。

サポートされていないデータ型については、シリアライゼーション クラスを手動で作成する必要があります。詳細については、第 9 章「非組み込みデータ型の使用法」を参照してください。

組み込みデータ型用の XML スキーマから Java へのマッピング

次の表に、XML スキーマ (ターゲット ネームスペース `http://www.w3.org/2001/XMLSchema`) が定義するすべての組み込みデータ型に対する定義済みマッピングと、対応する SOAP データ型 (ターゲット ネームスペース `http://schemas.xmlsoap.org/soap/encoding/`) を示します。

サポートされている非組み込み XML データ型のリストは、6-14 ページの「サポートされている XML 非組み込みデータ型」を参照してください。

表 5-2 組み込みデータ型用の XML スキーマから Java へのマッピング

XML スキーマ データ型	同等の Java データ型 (小文字はプリミティブ データ型を示す)
boolean	boolean
byte	byte
short	short
int	int
long	long
float	float
double	double
integer	java.math.BigInteger
decimal	java.math.BigDecimal
string	java.lang.String
dateTime	java.util.Calendar
base64Binary	byte[]
hexBinary	byte[]
duration	weblogic.xml.schema.binding.util.Duration
time	java.util.Calendar
date	java.util.Calendar
gYearMonth	java.util.Calendar `java.util.Calendar` Java データ型には `gYearMonth` データ型より多くのフィールドがあります。この追加情報には意味はありません。実際の XML データから生成されたものではなく、データ バインド機能によって生成された情報です。

表 5-2 組み込みデータ型用の XML スキーマから Java へのマッピング (続き)

XML スキーマ データ型	同等の Java データ型 (小文字はプリミティブ データ型を示す)
gYear	java.util.Calendar `java.util.Calendar` Java データ型には `gYearMonth` データ型より多くのフィールドがあります。この追加情報には意味はありません。実際の XML データから生成されたものではなく、データ バインド機能によって生成された情報です。
gMonthDay	java.util.Calendar `java.util.Calendar` Java データ型には `gYearMonth` データ型より多くのフィールドがあります。この追加情報には意味はありません。実際の XML データから生成されたものではなく、データ バインド機能によって生成された情報です。
gDay	java.util.Calendar `java.util.Calendar` Java データ型には `gYearMonth` データ型より多くのフィールドがあります。この追加情報には意味はありません。実際の XML データから生成されたものではなく、データ バインド機能によって生成された情報です。
gMonth	java.util.Calendar `java.util.Calendar` Java データ型には `gYearMonth` データ型より多くのフィールドがあります。この追加情報には意味はありません。実際の XML データから生成されたものではなく、データ バインド機能によって生成された情報です。
anyURI	java.lang.String
NOTATION	java.lang.String
token	java.lang.String
normalizedString	java.lang.String
language	java.lang.String
Name	java.lang.String
NMTOKEN	java.lang.String

表 5-2 組み込みデータ型用の XML スキーマから Java へのマッピング (続き)

XML スキーマ データ型	同等の Java データ型 (小文字はプリミティブ データ型を示す)
NCName	java.lang.String
NMTOKENS	java.lang.String[]
ID	java.lang.String
IDREF	java.lang.String
ENTITY	java.lang.String
IDREFS	java.lang.String[]
ENTITIES	java.lang.String[]
nonPositiveInteger	java.math.BigInteger
nonNegativeInteger	java.math.BigInteger
negativeInteger	java.math.BigInteger
unsignedLong	java.math.BigInteger
positiveInteger	java.math.BigInteger
unsignedInt	long
unsignedShort	int
unsignedByte	short
Qname	javax.xml.namespace.QName

組み込みデータ型用の Java から XML スキーマへのマッピング

サポートされている非組み込み Java データ型のリストは、6-16 ページの「サポートされている Java 非組み込みデータ型」を参照してください。

表 5-3 組み込みデータ型用の Java から XML スキーマへのマッピング

Java データ型 (小文字はプリミティブ データ型を示す)	同等の XML データ型
int	int
short	short
long	long
float	float
double	double
byte	byte
boolean	boolean
char	string (with facet of length=1)
java.lang.Integer	int
java.lang.Short	short
java.lang.Long	long
java.lang.Float	float
java.lang.Double	double
java.lang.Byte	byte
java.lang.Boolean	boolean
java.lang.Character	string (with facet of length=1)
java.lang.String	string
java.math.BigInteger	integer
java.math.BigDecimal	decimal
java.lang.String	string
java.util.Calendar	dateTime

表 5-3 組み込みデータ型用の Java から XML スキーマへのマッピング (続き)

Java データ型 (小文字はプリミティブ データ型を示す)	同等の XML データ型
java.util.Date	dateTime
byte[]	base64Binary
weblogic.xml.schema.binding.util.Duration	duration
javax.xml.namespace.QName	Qname

6 Ant タスクを使用した WebLogic Web サービスのアセンブル

この章では、さまざまな Ant タスクを使用して WebLogic Web サービスをアセンブルおよびデプロイする方法を説明します。

Ant タスクを使用した WebLogic Web サービスのアセンブルの概要

WebLogic Web サービスのアセンブルとは、サービスのすべてのコンポーネント (EJB JAR ファイル、SOAP メッセージ ハンドラ クラスなど) の収集、`web-services.xml` デプロイメント記述子ファイルの生成、および WebLogic Server にデプロイできるエンタープライズ アプリケーション アーカイブ (EAR) ファイルへのそれらすべてのパッケージ化を指します。

Ant タスクを使用した WebLogic Web サービスのアセンブルには次の 2 通りの方法があります。

- すべてのアセンブリ ステップを自動的に実行する `servicegen` Ant タスクの使用

  `servicegen` Ant タスクは、入力として EJB JAR ファイル (EJB 実装の Web サービスの場合) または Java クラスのリスト (Java クラス実装の Web サービスの場合) を受け取り、Java コードおよび Ant タスクの属性の参照後の情報に基づいて、WebLogic Web サービスを構成するすべてのコンポーネントを自動的に生成し、それらを EAR ファイルにパッケージ化します。

  詳細については、6-3 ページの「servicegen Ant タスクを使用した WebLogic Web サービスのアセンブル」を参照してください。

- `autotype`、`source2wsdd` といったさまざまな個別用途の Ant タスクの使用

  `servicegen` Ant タスクは、ほとんどの WebLogic Web サービスのアセンブルで適切に使用できます。ただし、Web サービスをどのようにアセンブルするかもっと細かくコントロールする場合は、個別用途の Ant タスクを使用できます。たとえば、`source2wsdd` で `web-services.xml` ファイルを生成し、情報を追加する必要がある場合にこのファイルを手動で更新できます。

  詳細については、6-6 ページの「他の Ant タスクを使用した WebLogic Web サービスのアセンブル」を参照してください。

Web サービスおよび Ant タスクの詳細なリファレンス情報は、付録 B「Web サービス Ant タスクとコマンドライン ユーティリティ」を参照してください。

注意 : WebLogic Server に付属の Java Ant ユーティリティでは、ANTCLASSPATH 変数を設定した際の *`WL_HOME`*`\server\bin` ディレクトリにある `ant` (UNIX) または `ant.bat` (Windows) コンフィグレーション ファイルを使用します。*`WL_HOME`* は、インストールした WebLogic Platform の最上位ディレクトリです。ANTCLASSPATH 変数を更新する必要がある場合は、使用しているオペレーティング システムの適切なファイルを変更してください。

servicegen Ant タスクを使用した WebLogic Web サービスのアセンブル

servicegen Ant タスクは、入力として EJB JAR ファイルまたは Java クラスのリストを受け取り、すべての必要な Web サービス コンポーネントを作成して、それらをデプロイ可能な EAR ファイルにパッケージ化します。

servicegen Ant タスクの機能

具体的には、servicegen Ant タスクは、次のタスクを実行します。

- Java コードを参照して、パブリック メソッドを検索し、Web サービス オペレーションおよびメソッドのパラメータおよび戻り値として使用される非組み込みデータ型に変換します。
- servicegen Ant タスクの属性と参照した EJB または Java クラスの情報に基づいて、web-services.xml デプロイメント記述子ファイルを作成します。
- 必要であれば、非組み込み型のデータを XML 表現から Java 表現へ、またはその逆へ変換するシリアライゼーション クラスを作成します。また、Java オブジェクトの XML スキーマ表現を作成し、それに対応して web-services.xml ファイルを更新します。サポートされている非組み込みデータ型のリストは、6-14 ページの「servicegen および autotype Ant タスクでサポートされる非組み込みデータ型」を参照してください。
- Web アプリケーション WAR ファイルにすべての Web サービス コンポーネントをパッケージ化し、次に、WAR ファイルと EJB JAR ファイルをデプロイ可能な EAR ファイルにパッケージ化します。

WebLogic Web サービスの自動アセンブル：主な手順

servicegen Ant タスクを使用して Web サービスを自動的にアセンブルするには次を行います。

1. 環境を設定します。

   Windows NT では、setWLSEnv.cmd コマンドを実行します。*WL_HOME*\server\bin ディレクトリにあります。*WL_HOME* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

   UNIX では、setWLSEnv.sh コマンドを実行します。*WL_HOME*/server/bin ディレクトリにあります。*WL_HOME* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

2. Web サービスのコンポーネントを格納するステージング ディレクトリを作成します。

3. Web サービス オペレーションが、EJB を使用して実装される場合は、サポートする EJB とともにそれらを EJB JAR ファイルにパッケージ化します。オペレーションが Java クラスを使用して実装される場合は、クラス ファイルにコンパイルします。

   詳細は、『WebLogic Server アプリケーションの開発』を参照してください。

4. EJB JAR ファイル、Java クラス ファイルのうちの一方または両方をステージング ディレクトリにコピーします。

5. ステージング ディレクトリに、servicegen Ant タスクへの呼び出しを含む Ant ビルド ファイル (デフォルトのファイル名は build.xml) を作成します。

   servicegen Ant タスクを指定する方法の詳細については、6-5 ページの「servicegen Ant タスクを実行する」を参照してください。

   Ant ビルド ファイルの一般的な情報については、http://jakarta.apache.org/ant/manual/ を参照してください。

6. ステージング ディレクトリで ant と入力し、必要であればこのコマンドにターゲットの引数を渡して、build.xml ファイルで指定された Ant タスク (1 つまたは複数) を実行します。

   ```
   prompt ant>
   ```

Ant タスクによってステージング ディレクトリに Web サービス EAR ファイルが生成されるので、WebLogic Server へのデプロイが可能になります。

servicegen Ant タスクを実行する

次の例、build.xml ファイルは、製品例、examples.webservices.basic.statelessession から引用したファイルで、servicegen Ant タスクを実行することを指定しています。

```
<project name="buildWebservice" default="ear">
  <target name="ear">
    <servicegen
       destEar="ws_basic_statelessSession.ear"
       contextURI="WebServices" >
       <service
         ejbJar="HelloWorldEJB.jar"
         targetNamespace="http://www.bea.com/webservices/basic/statelesSession"
         serviceName="HelloWorldEJB"
         serviceURI="/HelloWorldEJB"
         generateTypes="True"
         expandMethods="True"
         style="rpc" >
       </service>
     </servicegen>
  </target>
</project>
```

この例では、servicegen Ant タスクは、HelloWorldEJB という Web サービスを 1 つ作成します。この Web サービスを識別するための URI は /HelloWorldEJB です。Web サービスにアクセスするための完全 URL は次のとおりです。

```
http://host:port/WebServices/HelloWorldEJB
```

servicegen Ant タスクは、ws_basic_statelessSession.ear という EAR ファイルに Web サービスをパッケージ化します。EAR ファイルには、web-services.war (デフォルト名) という、web-services.xml デプロイメント記述子ファイルなどの Web サービスコンポーネントがすべて格納された WAR ファイルが含まれています。

generateTypes 属性は、True に設定されているので、WAR ファイルには、EJB メソッドに対するパラメータおよび戻り値として使用される非組み込みデータ型用のシリアライゼーション クラスも入っています。Ant タスクは、

HelloWorldEJB.jar ファイルにある EJB を参照して、パブリック オペレーションと非組み込みデータ型を探し、それに対応して、web-services.xml オペレーションとデータ型マッピングのセクションを更新します。expandMethods 属性も True に設定されているので、Ant タスクは、各パブリック EJB メソッドを別個のオペレーションとして web-services.xml ファイルにリストします。

style="rpc" 属性は、Web サービスのオペレーションがすべて RPC 指向であることを指定します。Web サービスのオペレーションがドキュメント指向の場合は、style="document" を指定します。

他の Ant タスクを使用した WebLogic Web サービスのアセンブル

servicegen Ant タスクは、ほとんどの WebLogic Web サービスのアセンブルで適切に使用できます。ただし、Web サービスをどのようにアセンブルするかもっと細かくコントロールする場合は、個別用途の Ant タスクを使用できます。たとえば、source2wsdd で web-services.xml ファイルを生成し、情報を追加する必要がある場合にこのファイルを手動で更新できます。

servicegen 以外の Ant タスクで WebLogic Web サービスをアセンブルするには、次の手順を行います。

1. Web サービスを実装するバックエンド コンポーネントを、それぞれのパッケージにパッケージ化またはコンパイルします。たとえば、ステートレス セッション EJB は EJB JAR ファイルにパッケージ化し、Java クラスはクラス ファイルにパッケージ化します。

   詳細は、「WebLogic Server アプリケーションのパッケージ化」を参照してください。

2. Web サービス デプロイメント記述子ファイル (web-services.xml) を作成します。

   Java クラスを使用して Web サービスを実装している場合は、source2wsdd Ant タスクを使用して web-services.xml ファイルを生成できます。詳細については、6-7 ページの「source2wsdd Ant タスクを実行する」を参照してください。wsdl2Service Ant タスクを使用して既存の WSDL ファイルから

Web サービスの部分的な実装を生成した場合は、その Ant タスクによって web-services.xml ファイルが既に生成されています。

EJB 実装の Web サービスなど、他のすべてのケースでは、web-services.xml ファイルを手作業で作成する必要があります。7-4 ページの「web-services.xml ファイルの手動作成：主な手順」を参照してください。

3. Web サービスで非組み込みデータ型が使用されている場合は、シリアライゼーション クラスなどの必要なコンポーネントをすべて作成します。それらのコンポーネントは、autotype Ant タスクで自動的に生成します (6-9 ページの「autotype Ant タスクを実行する」を参照)。
4. 必要であれば、clientgen Ant タスクを使用してクライアント JAR ファイルを作成します。

   6-10 ページの「clientgen Ant タスクを実行する」を参照してください。
5. wspackage Ant タスクを使用して、すべてのコンポーネントをデプロイ可能な EAR ファイルにパッケージ化します (6-11 ページの「wspackage Ant タスクを実行する」を参照)。

source2wsdd Ant タスクを実行する

source2wsdd Ant タスクを使用すると、Web サービスを実装する Java ソース ファイルから web-services.xml デプロイメント記述子ファイルを生成できます。

注意： この Ant タスクでは、EJB 実装の Web サービスの web-services.xml ファイルは生成できません。この Ant タスクは、Java クラス実装の Web サービスでのみ使用できます。

source2wsdd Ant タスクを実行するには、次の手順に従います。

1. source2wsdd Ant タスクへの呼び出しの入った build.xml というファイルを作成します。「source2wsdd Ant タスクのサンプル build.xml ファイル」を参照してください。
2. 環境を設定します。

Windows NT では、setWLSEnv.cmd コマンドを実行します。*WL_HOME*\server\bin ディレクトリにあります。*WL_HOME* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

UNIX では、setWLSEnv.sh コマンドを実行します。*WL_HOME*/server/bin ディレクトリにあります。*WL_HOME* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

3. build.xml ファイルと同じディレクトリで ant と入力し、Ant タスクまたは build.xml ファイルで指定されたタスクを実行します。

   ```
   prompt> ant
   ```

source2wsdd Ant タスクのリファレンス情報については、B-30 ページの「source2wsdd」を参照してください。

source2wsdd Ant タスクのサンプル build.xml ファイル

以下は、簡単な build.xml ファイルの例です。

```
<project name="buildWebservice" default="generate-typeinfo">
   <target name="generate-typeinfot">
      <source2wsdd
           javaSource="c:\source\MyService.java"
           typesInfo="c:\autotype\types.xml"
           ddFile="c:\ddfiles\web-services.xml"
           serviceURI="/MyService" />
</project>
```

例の中の source2wsdd Ant タスクでは、c:\source\MyService.java という Java ソース ファイルから web-services.xml ファイルが生成されます。非組み込みデータ型の情報は、c:\autotype\types.xml ファイルから利用します。この情報には、Web サービスのパラメータまたは戻り値として使用される非組み込みデータ型の XML スキーマ表現や、シリアライゼーション クラスの位置を指定するデータ型マッピング情報などが含まれます。このファイルは、通常は autotype Ant タスクを使用して生成します。

source2wsdd Ant タスクは、生成されたデプロイメント記述子情報をファイル c:\ddfiles\web-services.xml に出力します。Web サービスの URI は /MyService で、これはデプロイされたこの Web サービスを呼び出す完全 URL で使用します。

autotype Ant タスクを実行する

autotype Ant タスクを使用すると、シリアライゼーション クラスなどの非組み込みデータ型コンポーネントを生成できます。サポートされている非組み込みデータ型のリストは、6-14 ページの「servicegen および autotype Ant タスクでサポートされる非組み込みデータ型」を参照してください。

autotype Ant タスクを実行するには、次の手順に従います。

1. autotype Ant タスクへの呼び出しの入った build.xml というファイルを作成します。詳細については、「Autotype Ant タスクのサンプル build.xml ファイル」を参照してください。
2. 環境を設定します。

   Windows NT では、setWLSEnv.cmd コマンドを実行します。*WL_HOME*\server\bin ディレクトリにあります。*WL_HOME* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

   UNIX では、setWLSEnv.sh コマンドを実行します。*WL_HOME*/server/bin ディレクトリにあります。*WL_HOME* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

3. build.xml ファイルと同じディレクトリで ant と入力し、Ant タスクまたは build.xml ファイルで指定されたタスクを実行します。

   ```
   prompt> ant
   ```

autotype Ant タスクのリファレンス情報については、B-7 ページの「autotype」を参照してください。

Autotype Ant タスクのサンプル build.xml ファイル

以下は、簡単な build.xml ファイルの例です。

```
<project name="buildWebservice" default="generate-typeinfo">
   <target name="generate-typeinfot">
     <autotype  javatypes="mypackage.MyType"
                targetNamespace="http://www.foobar.com/autotyper"
                packageName="a.package.name"
                destDir="d:\output" />
   </target>
</project>
```

例の中の autotype Ant タスクでは、mypackage.MyType という Java クラスの非組み込みデータ型コンポーネントが作成されます。生成されるシリアライゼーション クラスでは、a.package.name というパッケージ名が使用されます。生成されたシリアライゼーション Java クラスと XML スキーマ情報は、d:\output ディレクトリに配置されます。生成された XML スキーマおよび型マッピング情報は、この output ディレクトリの types.xml というファイルに格納されます。

次の build.xml ファイルのサンプルからの抜粋は、autotype タスクのもう一つの使用法を示しています。

```
<autotype  wsdl="file:\wsdls\myWSDL"
           targetNamespace="http://www.foobar.com/autotyper"
           packageName="a.package.name"
           destDir="d:\output" />
```

この例は最初の例と似ていますが、データ型の Java 表現ではなく、Web サービスの WSDL に埋め込まれた XML スキーマ表現で始まっています。この場合、タスクは対応する Java 表現を生成します。

clientgen Ant タスクを実行する

clientgen Ant タスクを実行し、クライアント JAR ファイルを自動的に生成するには次を行います。

1. clientgen Ant タスクへの呼び出しの入った build.xml というファイルを作成します。詳細については、「clientgen Ant タスクのサンプル build.xml ファイル」を参照してください。
2. 環境を設定します。

   Windows NT では、setWLSEnv.cmd コマンドを実行します。*WL_HOME*\server\bin ディレクトリにあります。*WL_HOME* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

   UNIX では、setWLSEnv.sh コマンドを実行します。*WL_HOME*/server/bin ディレクトリにあります。*WL_HOME* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。
3. build.xml ファイルと同じディレクトリで ant と入力し、Ant タスクまたは build.xml ファイルで指定されたタスクを実行します。

   ```
   prompt> ant
   ```

clientgen Ant タスクのリファレンス情報については、B-12 ページの「clientgen」を参照してください。

clientgen Ant タスクのサンプル build.xml ファイル

以下は、簡単な build.xml ファイルの例です。

```
<project name="buildWebservice" default="generate-client">
   <target name="generate-client">
      <clientgen ear="c:/myapps/myapp.ear"
                 serviceName="myService"
           packageName="myapp.myservice.client"
                 useServerTypes="True"
                 clientJar="c:/myapps/myService_client.jar" />
   </target>
</project>
```

この例では、clientgen Ant タスクは c:/myapps/myService_client.jar クライアント JAR ファイルを作成します。この JAR ファイルには、c:/myapps/myapp.ear EAR ファイルに含まれる myService という WebLogic Web サービスの呼び出しに使用する、サービス固有のクライアントインタフェースとスタブ、およびシリアライゼーション クラスが含まれます。クライアント インタフェースとスタブ ファイルは、myapp.myservice.client にパッケージ化されます。useServerTypes 属性は、clientgen Ant タスクが、この Web サービスで使用される非組み込みデータ型を実装する Java コードを生成するのではなく、それらすべてのデータ型の Java 実装を c:/myapps/myapp.ear ファイルから取得するように指定します。

wspackage Ant タスクを実行する

wspackage Ant タスクを使用すると、Web サービスのさまざまなコンポーネントをデプロイ可能な EAR ファイルにパッケージ化できます。

wspackage Ant タスクを実行するには、次の手順に従います。

1. wspackage Ant タスクへの呼び出しの入った build.xml というファイルを作成します。詳細については、「wspackage Ant タスクのサンプル build.xml ファイル」を参照してください。
2. 環境を設定します。

Windows NT では、setWLSEnv.cmd コマンドを実行します。*WL_HOME*\server\bin ディレクトリにあります。*WL_HOME* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

UNIX では、setWLSEnv.sh コマンドを実行します。*WL_HOME*/server/bin ディレクトリにあります。*WL_HOME* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

3. build.xml ファイルと同じディレクトリで ant と入力し、Ant タスクまたは build.xml ファイルで指定されたタスクを実行します。

```
prompt> ant
```

wspackage Ant タスクのリファレンス情報については、B-36 ページの「wspackage」を参照してください。

wspackage Ant タスクのサンプル build.xml ファイル

次の例は、Java クラスを使用して実装される Web サービスのデプロイ可能な EAR ファイルを作成するための単純な build.xml ファイルです。

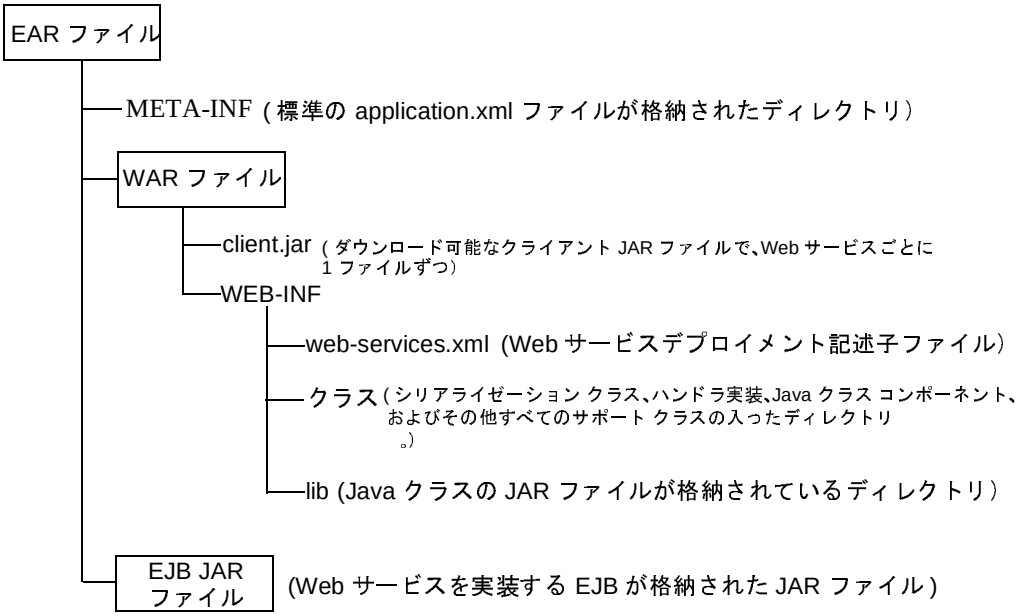
```
<project name="buildWebservice" default="generate-typeinfo">
   <target name="generate-typeinfot">
      <wspackage
           output="c:\myWebService.ear"
           contextURI="web_services"
           codecDir="c:\autotype"
           webAppClasses="example.ws2j.service.SimpleTest"
           ddFile="c:\ddfiles\web-services.xml" />

</project>
```

例の中の wspackage Ant タスクでは、c:\myWebService.ear という EAR ファイルが作成されます。Web サービスのコンテキスト URI は web_services で、これはその Web サービスを呼び出す完全 URL で使用されます。非組み込みデータ型のシリアライザ クラスを含むシリアライザ クラスは、c:\autotype ディレクトリに配置されます。Web サービスを実装する Java クラスは example.ws2j.service.SimpleTest という名前で、Web アプリケーションの WEB-INF/classes ディレクトリにパッケージ化されます。既存のデプロイメント記述子ファイルは c:\ddfiles\web-services.xml です。

Web サービス EAR ファイル パッケージ

Web サービスは、Web アプリケーションの WAR ファイルと EJB JAR ファイルが入っているエンタープライズ アプリケーションの標準の EAR ファイルにパッケージ化されています。

次の図は、典型的な WebLogic Web サービス EAR ファイル階層を示しています。



servicegen および autotype Ant タスクでサポートされる非組み込みデータ型

以降の節の表は、servicegen および autotype Ant タスクでデータ型コンポーネント (シリアライザ クラス、Java または XML 表現など) を生成できる非組み込み XML および Java データ型を示しています。

以降の表で示されていず、5-13 ページの「組み込みデータ型の使用法」の組み込みデータ型でもない XML または Java データ型については、非組み込みデータ型コンポーネントを手動で作成する必要があります。詳細については、第 9 章「非組み込みデータ型の使用法」を参照してください。

警告： autotype、servicegen、および clientgen Ant タスクによって生成されたシリアライザ クラスと Java および XML 表現はラウンドトリップできません。詳細については、6-18 ページの「生成されたデータ型コンポーネントの非ラウンドトリップ」を参照してください。

データ型に関して、WebLogic Web サービスが JAX-RPC 仕様にどのように準拠していないかについては、6-17 ページの「JAX-RPC へのデータ型の非準拠」を参照してください。

サポートされている XML 非組み込みデータ型

次の表は、サポートされている XML スキーマ非組み込みデータ型を示しています。XML データ型が表にある場合は、servicegen および autotype Ant タスクを使用して、XML 表現と Java 表現の間でデータを変換するシリアライザ クラスや、web-services.xml デプロイメント記述子の Java 表現および型マッピング情報を生成できます。

データ型の詳細や例については、JAX-RPC 仕様を参照してください。

表 6-1 サポートされている非組み込み XML スキーマ データ型

XML スキーマ データ型	同等の Java データ型またはマッピング メカニズム
列挙	型保障列挙パターン。詳細は、JAX-RPC 仕様のセクション 4.2.4 を参照。
単純型および複合型の両方の要素を持つ `<xsd:complexType>`	JavaBeans。
単純なコンテンツを含む `<xsd:complexType>`	JavaBeans。
`<xsd:complexType>` の `<xsd:attribute>`	JavaBeans のプロパティ。
既存の単純型の制限による新しい単純型の派生	同等の単純型の Java データ型。
制限要素で使用されるファセット **注意：** 基底のプリミティブ型は `string`、`decimal`、`float`、または `double` のいずれかでなければならない。パターン ファセットは任意。	シリアライゼーションおよびデシリアライゼーションで強制される制限。
`<xsd:list>`	リスト データ型の配列。
`wsdl:arrayType` 属性を使用した制限によって `soapenc:Array` から派生する配列	`arrayType` データ型と同等の Java データ型の配列。
制限によって `soapenc:Array` から派生する配列	同等の Java データ型の配列。
単純型からの複合型の派生	`String` 型の `simpleContent` というプロパティを持つ JavaBeans。
`<xsd:anyType>`	`java.lang.ObjectÅB`

表 6-1 サポートされている非組み込み XML スキーマ データ型

XML スキーマ データ型	同等の Java データ型またはマッピング メカニズム
<xsd:nil> および <xsd:nillable> 属性	Java の null 値。 XML データ型が組み込みデータ型で通常は Java プリミティブ データ型 (int や short など) にマップされる場合、その XML データ型は実際には同等のオブジェクト ラッパー型 (java.lang.Integer や java.lang.Short など) にマップされる。
拡張による複合型の派生	Java の継承を使用してマップされる。
抽象型	抽象 Java データ型。

サポートされている Java 非組み込みデータ型

次の表は、サポートされている Java 非組み込みデータ型を示しています。Java データ型がこの表にある場合、servicegen および autotype Ant タスクは Java 表現と XML 表現の間でデータを変換するシリアライザ クラスを生成できます。

表 6-2 サポートされている非組み込み Java データ型

Java データ型	同等の XML スキーマ データ型
サポートされているデータ型の配列。	SOAP 配列
サポートされているデータ型をプロパティとする JavaBean	<xsd:sequence>
java.util.List	SOAP 配列
java.util.ArrayList	SOAP 配列
java.util.LinkedList	SOAP 配列

表 6-2 サポートされている非組み込み Java データ型

Java データ型	同等の XML スキーマ データ型
java.util.Vector	SOAP 配列
java.util.Stack	SOAP 配列
java.lang.Object **注意 :** 実行時オブジェクトのデータ型は既知の型 (組み込みデータ型または型マッピング情報のあるデータ型のいずれか) でなければならない。	<xsd:anyType>
JAX-RPC スタイルの列挙値クラス	列挙ファセットを持つ <xsd:simpleType>

JAX-RPC へのデータ型の非準拠

autotype Ant タスクは、(そのために Java 表現を生成している) XML スキーマのデータ型に、以下の特性がすべて備わっている場合、JAX-RPC 仕様には準拠しません。

- データ型が complexType である。
- complexType に 1 つの sequence が含まれている。
- sequence に、maxOccurs が 1 より大きいか、または無限である、1 つの element が含まれている。

次の例に、上記のような XML スキーマのデータ型を示します。

```
<xsd:complexType name="Response">
    <xsd:sequence >
      <xsd:element name="code" type="xsd:string" maxOccurs="10" />
    </xsd:sequence>
</xsd:complexType>
```

autotype Ant タスクは、このタイプの XML スキーマのデータ型を、指定された要素の Java 配列に直接マップします。前述の例では、autotype Ant タスクは

Response XML スキーマ データ型を、java.lang.String[] Java 型にマップしています。これは、.NET が行うようなマッピングに類似しています。

一方、JAX-RPC 仕様では、このタイプの XML スキーマ データ型は、JavaBean クラス内のセッター メソッドとゲッター メソッドのペアを使用して、Java 配列にマップすべきであると述べています。WebLogic Web サービスは、仕様におけるこの最後の部分には、従っていません。

生成されたデータ型コンポーネントの非ラウンドトリップ

servicegen または autotype Ant タスクを使用して非組み込みデータ型のシリアライザ クラスおよび Java または XML 表現を作成する場合、このプロセスはラウンドトリップできないことに注意する必要があります。たとえば、autotype Ant タスクを使用して XML スキーマ データ型の Java 表現を生成し、次に、生成された Java 型から autotype を使用して XML スキーマ データ型を作成した場合、元の XML スキーマ データ型と生成された XML スキーマ データ型は、どちらも同じ XML データを記述しているにもかかわらず同じになるとは限りません。同様に、Java 型から XML スキーマを生成し、次に、生成されたその XML スキーマから新しい Java データ型を生成した場合も、元の Java 型と新しい Java 型が同じになるとは限りません。たとえば、元の Java 型と生成された Java 型では、コンストラクタのパラメータの列挙順序が異なる場合があります。

こうした動作は、さまざまな処理に影響を与えます。たとえば、非組み込みデータ型を使用する既存のステートレス セッション EJB から Web サービスを開発するとします。この場合、autotype Ant タスクを使用して非組み込みデータ型のシリアライザ クラスと Java および XML 表現を生成し、生成されたコードを、Web サービスを実装するサーバサイド コードで使用します。次に、clientgen Ant タスクを使用して Web サービス固有のクライアント JAR ファイルを生成します。このファイルにも非組み込みデータ型のシリアライザ クラスと Java 表現が組み込まれています。しかし、デフォルトによって clientgen はこれらのコンポーネントを Web サービスの WSDL (したがって XML スキーマ) から生成するので、clientgen によって生成されるクライアントサイド Java 表現は autotype によって生成されるサーバサイド Java コードと同じにならない場合があります。このため、クライアント アプリケーションではデータ型を処理する

サーバサイド コードを再利用できない場合があります。clientgen Ant タスクが WebLogic Web サービス EAR ファイル内の生成されたシリアライザ クラスとコードを常に使用するようにするには、useServerTypes 属性を指定します。

WebLogic Web サービスのデプロイ

WebLogic Web サービスのデプロイとは、リモート クライアントでそのサービスを使用できるようにすることです。WebLogic Web サービスは標準 J2EE エンタープライズ アプリケーションとしてパッケージ化されているため、Web サービスのデプロイはエンタープライズ アプリケーションのデプロイと同じです。

エンタープライズ アプリケーションのデプロイメントの詳細については、「WebLogic Server デプロイメント」を参照してください。

7 WebLogic Web サービスの手動アセンブル

この章では、WebLogic Web サービスの手動アセンブルに関する情報を提供します。

WebLogic Web サービスの手動アセンブルの概要

WebLogic Web サービスのアセンブルとは、サービスのすべてのコンポーネント (EJB JAR ファイル、SOAP メッセージ ハンドラ クラスなど) の収集、`web-services.xml` デプロイメント記述子ファイルの生成、および WebLogic Server にデプロイできるエンタープライズ アプリケーション EAR ファイルへのそれらすべてのパッケージ化を指します。

通常は、手順が複雑で時間がかかるので、WebLogic Web サービスは手動でアセンブルしません。そのかわりに、`servicegen`、`autotype`、`source2wsdd` などの WebLogic Ant タスクを使用して、自動的に必要なコンポーネントをすべて生成し、デプロイ可能な EAR ファイルにパッケージ化します。

ただし、Web サービスがあまりにも複雑で Ant タスクが必須コンポーネントを生成できないか、Web サービスのアセンブリのすべての側面を完全にコントロールする必要がある場合は、この章の内容に基づいて Web サービスを手動でアセンブルできます。

WebLogic Web サービスの手動アセンブル：主な手順

1. Web サービスを実装するバックエンド コンポーネントを、それぞれのパッケージにパッケージ化またはコンパイルします。たとえば、ステートレス セッション EJB は EJB JAR ファイルにパッケージ化し、Java クラスはクラス ファイルにパッケージ化します。

   詳細は、「WebLogic Server アプリケーションのパッケージ化」を参照してください。

2. Web サービス デプロイメント記述子ファイル (`web-services.xml`) を作成します。

   `web-services.xml` ファイルの詳細は、7-3 ページの「web-services.xml ファイルの概要」を参照してください。そのファイルを手動で作成する詳しい手順は、7-4 ページの「web-services.xml ファイルの手動作成：主な手順」を参照してください。

3. Web サービスで非組み込みデータ型が使用されている場合は、シリアライゼーション クラスなどの必要なコンポーネントをすべて作成します。

   必要なコンポーネントの手動作成の詳細は、第 9 章「非組み込みデータ型の使用法」を参照してください。

4. すべてのコンポーネントをデプロイ可能な EAR ファイルにパッケージ化します。

   EAR ファイルを手動でパッケージ化するときには、必ず適切な Web サービス コンポーネントを Web アプリケーション WAR ファイルに入れてください。WAR および EAR ファイル階層の詳細については、6-13 ページの「Web サービス EAR ファイル パッケージ」を参照してください。WAR ファイルと EAR ファイルを作成する手順については、「WebLogic Server アプリケーションのパッケージ化」を参照してください。

web-services.xml ファイルの概要

web-services.xml デプロイメント記述子ファイルには、Web サービスを実装するバックエンド コンポーネント、パラメータおよび戻り値として使用される非組み込みデータ型、SOAP メッセージをインターセプトする SOAP メッセージ ハンドラなど、1 つまたは複数の WebLogic Web サービスを記述する情報が入っています。他のすべてのデプロイメント記述子の場合と同様に、web-services.xml も XML ファイルです。

WebLogic Server は、web-services.xml デプロイメント記述子ファイルの内容に基づいて、デプロイされた WebLogic Web サービスの WSDL を動的に生成します。動的に生成された WSDL の URL を取得する方法の詳細については、8-23 ページの「WebLogic Web サービスのホーム ページおよび WSDL の URL」を参照してください。

単一の WebLogic Web サービスは、1 つまたは複数のオペレーションで構成されています。各オペレーションは、異なるバックエンド コンポーネントと SOAP メッセージ ハンドラのメソッドを使用して実装することができます。たとえば、オペレーションは、ステートレス セッション EJB の単一メソッドを使用して実装されている場合、または SOAP メッセージ ハンドラとステートレス セッション EJB を組み合わせて実装されている場合があります。

1 つの web-services.xml ファイルには、少なくとも 1 つの WebLogic Web サービスの記述があります。

Web サービスを手動でアセンブルする場合は (たとえば SOAP メッセージ ハンドラおよびハンドラ チェーンを使用する場合は必須)、web-services.xml ファイルを手動で作成する必要があります。WebLogic Web サービスを、servicegen Ant タスクを使用してアセンブルする場合は、web-services.xml ファイルを手動で作成する必要はありません。Ant タスクが、EJB、Ant タスクの属性などを参照してこのファイルを作成するからです。

Web サービスを手動でアセンブルする必要がある場合も、servicegen Ant タスクを使用して基本となるテンプレートを作成してから、このマニュアルを参照して、生成された web-services.xml を servicegen では提供されていない追加の情報で更新することができます。

web-services.xml ファイルの手動作成：主な手順

web-services.xml デプロイメント記述子ファイルは、1 つまたは複数の WebLogic Web サービスを記述するものです。このファイルには、Web サービスを構成するオペレーション、オペレーションを実装するバックエンド コンポーネント、そしてオペレーションのパラメータおよび戻り値として使用される非組み込みデータ型に関するデータ型マッピングなどについての情報が入っています。さまざまな WebLogic Web サービスを記述する web-services.xml ファイルの詳細な例については、7-11 ページの「Sample web-services.xml Files」を参照してください。web-services.xml ファイルは任意のテキスト エディタで作成することができます。

この節で説明する各要素の詳細な説明は、付録 A「WebLogic Web サービス デプロイメント記述子の要素」を参照してください。

次に、簡単な web-services.xml ファイルの例を示します。

```
<web-services>
     <web-service name="stockquotes" targetNamespace="http://example.com"
                  uri="/myStockQuoteService">
          <components>
               <stateless-ejb name="simpleStockQuoteBean">
                    <ejb-link path="stockquoteapp.jar#StockQuoteBean" />
               </stateless-ejb>
          </components>
          <operations>
               <operation method="getLastTradePrice"
                          component="simpleStockQuoteBean" />
          </operations>
     </web-service>
</web-services>
```

web-services.xml ファイルを手動で作成するには、次の手順に従います。

1. 他のすべての要素を含む、<web-services> ルート要素を作成します。

```
<web-services>
...
</web-servives>
```

2. 作成する Web サービスに、SOAP メッセージをインターセプトする SOAP メッセージ ハンドラを含むものがある場合は、<web-services> ルート要素の <handler-chains> 子要素を作成して、ハンドラの記述に関連する子要素をすべて、呼び出し順などでハンドラ チェーンに格納します。詳細については、10-17 ページの「SOAP メッセージ ハンドラ情報による web-services.xml ファイルの更新」を参照してください。

3. 定義する各 Web サービスについて、次の手順を行います。

   a. <web-services> 要素の <web-service> 子要素を作成します。name、targetNamespace、および uri 属性を使用して、Web サービスの名前、ターゲット ネームスペース、クライアントが Web サービスの呼び出しに使用する URI を指定します。次に例を示します。

```
<web-service name="stockquote"
             targetNamespace="http://example.com"
             uri="myStockQuoteService">
...
</web-service>
```

   Web サービスのオペレーションがすべてドキュメント指向であるように指定するには、style="document" 属性を使用します。style 属性のデフォルト値は rpc です。この場合はすべてのオペレーションが RPC 指向になります。

   b. Web サービスのオペレーションを実装するバックエンド コンポーネントをリストする <web-service> 要素の <components> 子要素を作成します。詳細については、7-6 ページの「<components> 要素を作成する」を参照してください。

   c. Web サービスを構成するオペレーションが、パラメータまたは戻り値として非組み込みデータ型を使用する場合は、<web-service> 要素の <types> 子要素および <type-mapping> 子要素を作成して、データ型マッピング情報を追加します。詳細については、9-11 ページの「データ型マッピング ファイルを作成する」を参照してください。

注意： Web サービスのオペレーションがパラメータおよび戻り値として組み込みデータ型のみを使用する場合は、この手順は必要ありません。サポートされている組み込みデータ型のリストは、5-13 ページの「組み込みデータ型の使用法」を参照してください。

d. Web サービスを構成するオペレーションをリストする <web-service> 要素の <operations> 子要素を作成します。

```
<operations xmlns:xsd="http://www.w3.org/2001/XMLSchema">
....
</operations>
```

e. <operations> 要素内で、Web サービス用に定義されたオペレーションをリストします。詳細については、7-7 ページの「<operation> 要素を作成する」を参照してください。

<components> 要素を作成する

Web サービスのオペレーションを実装するバックエンド コンポーネントをリストおよび記述する <web-service> 要素の <components> 子要素を作成します。各バックエンド コンポーネントに、後で、そのコンポーネントが実装するオペレーションを記述するのに使用する name 属性があります。

注意： ハンドラとハンドラ チェーンがすべての作業を行ってバックエンド コンポーネントを実行しない、SOAP メッセージ ハンドラのみのタイプの Web サービスを作成する場合は、web-services.xml ファイルでの <components> 要素の指定は行いません。それ以外のすべての Web サービスについては、<components> 要素を宣言する必要があります。

リストできるバックエンド コンポーネントのタイプは次のとおりです。

- <stateless-ejb>

  この要素は、ステートレス EJB バックエンド コンポーネントを記述します。<ejb-link> 子要素を使用して EJB 名とそれが格納されている JAR ファイルを指定するか、または <jndi-name> 子要素を使用して EJB の JNDI 名を指定します。次に例を示します。

```
<components>
  <stateless-ejb name="simpleStockQuoteBean">
     <ejb-link path="stockquoteapp.jar#StockQuoteBean" />
```

```
      </stateless-ejb>
    </components>
```

- <java-class>

  この要素は、Java クラスのバックエンド コンポーネントを記述します。class-name 属性を使用して、Java クラスの完全修飾パス名を指定します。次に例を示します。

```
<components>
   <java-class name="customClass"
               class-name="myclasses.MyOwnClass" />
</components>
```

<operation> 要素を作成する

<operation> 要素は、WebLogic Web サービスのパブリック オペレーションの実装方法を記述します。パブリック オペレーションとは、Web サービスの WSDL でリストされ、Web サービスを呼び出すクライアント アプリケーションによって実行されるオペレーションのことです。以下は、<operation> 宣言の例です。

```
<operation name="getQuote"
           component="simpleStockQuoteBean"
           method="getQuote">
   <params>
      <param name="in1" style="in" type="xsd:string" location="Header"/>
      <param name="in2" style="in" type="xsd:int"  location="Header"/>
      <return-param name="result" type="xsd:string"  location="Header"/>
   </params>
</operation>
```

一般的には、web-services.xml ファイルにある <operation> 要素の各インスタンスには、Web サービス オペレーションのパブリック名に変換される name 属性があります。唯一の例外は、method="*" 属性を使用して EJB または Java クラスのすべてのメソッドを 1 つの <operation> 要素に指定する場合です (この場合は、オペレーションのパブリック名がメソッド名となる)。

<operation> 要素の属性を組み合わせて使用し、各種オペレーションを指定します。詳細については、7-8 ページの「オペレーションのタイプを指定する」を参照してください。

必要であれば、<params> 要素を使用して、オペレーションのパラメータと戻り値をまとめます。詳細については、7-9 ページの「オペレーションのパラメータおよび戻り値を指定する」を参照してください。

オペレーションのタイプを指定する

<operation> 要素の属性をさまざまな組み合わせで使用して、オペレーションのタイプ、オペレーションを実装するコンポーネントのタイプ、一方向オペレーションかどうかなどを識別します。

注意： わかりやすくするため、次の節で取り上げる例では、パラメータの宣言はしていません。

以下の例は、さまざまなオペレーションを宣言する方法を示しています。

- オペレーションが、ステートレス セッション EJB のメソッドのみを使用して実装されるように指定するには、name 属性、component 属性、method 属性を使用します。次に例を示します。

```
<operation name="getQuote"
           component="simpleStockQuoteBean"
           method="getQuote">
</operation>
```

- 1 つの <operation> 要素のみを使用して、EJB または Java クラスのすべてのメソッドを指定するには、method="*" 属性を使用します。この場合、メソッド名がオペレーションのパブリック名となります。

```
<operation component="simpleStockQuoteBean"
           method="*">
</operation>
```

- オペレーションがデータを受け取るのみで、クライアント アプリケーションに何も返さないように指定するには、invocation-style 属性を追加します。

```
<operation name="getQuote"
           component="simpleStockQuoteBean"
           method="getQuote(java.lang.String)"
           invocation-style="one-way">
</operation>
```

この例は、メソッドのフル シグネチャを、method 属性を使用して指定する方法も示しています。メソッドのフル シグネチャは、EJB または Java クラスがメソッドに対して過負荷となり、Web サービス オペレーションとして

公開するメソッドを、あいまいさを排除した形で宣言しなければならない場合に限って指定する必要があります。

- オペレーションが、SOAP メッセージ ハンドラ チェーンとステートレス セッション EJB のメソッドを使用して実装されるように指定するには、name 属性、component 属性、method 属性、handler-chain 属性を使用します。

```
<operation name="getQuote"
           component="simpleStockQuoteBean"
               method="getQuote"
           handler-chain="myHandler">
</operation>
```

- オペレーションが SOAP メッセージ ハンドラ チェーンのみを使用して実装されるように指定するには、name 属性と handler-chain 属性を使用します。

```
<operation name="justHandler"
           handler-chain="myHandler">

</operation>
```

オペレーションのパラメータおよび戻り値を指定する

<params> 要素を使用して、オペレーションのパラメータと戻り値を明示的に宣言します。

オペレーションのパラメータや戻り値は明示的にリストする必要はありません。<operation> 要素に <params> 子要素がない場合、WebLogic Server はオペレーションを実装するバックエンド コンポーネントを参照してパラメータと戻り値を決定します。Web サービスの WSDL の生成時に、WebLogic Server は対応するメソッドのパラメータと戻り値の名前を使用します。

以下の場合には、オペレーションのパラメータと戻り値を明示的にリストしてください。

- 生成された WSDL のパラメータおよび戻り値の名前は、そのオペレーションを実装するメソッドの名前とは異なる名前とする必要がある場合。
- パラメータを SOAP の要求ヘッダーまたは応答ヘッダーにある名前にマップする必要がある場合。
- out または inout パラメータを使用する必要がある場合。

<params> 要素の <param> 子要素を使用して in パラメータを 1 つ指定し、<return-param> 子要素を指定して戻り値を指定します。in パラメータは、オペレーションを実装するメソッドを定義した順序と同じ順序で指定する必要があります。<param> 要素の数は、メソッドのパラメータ数と一致する必要があります。<return-param> 要素は、1 つのみ指定することができます。

<param> 要素と <return-param> 要素の属性を使用して、SOAP メッセージ内のパラメータの位置 (本文かヘッダー)、パラメータのタイプ (in、out、または inout) などを指定します。必ず、type 属性を使用してパラメータの XML スキーマのデータ型を指定する必要があります。次の例は、さまざまな in パラメータと戻り値を示しています。

- パラメータが、SOAP 要求メッセージのヘッダーにある標準の in パラメータであることを指定するには、次に示すように、style 属性と location 属性を使用します。

  ```
  <param name="inparam" style="in"
         location = "Header" type="xsd:string" />
  ```

- out および inout パラメータによって、(標準の <return-value> 要素を使用するほかに) オペレーションが複数の戻り値を返すことができます。次の <param> 要素のサンプルは、パラメータが inout パラメータであること、つまり、in パラメータと out パラメータの機能を兼ねることを指定する方法を示しています。

  ```
  <param name="inoutparam" style="inout"
         type="xsd:int" />
  ```

  location 属性のデフォルト値は Body なので、in パラメータ値と out パラメータ値は両方とも、SOAP メッセージの本文にあります。

- 次の例は、SOAP の応答メッセージのヘッダーにある標準の戻り値を指定する方法を示しています。

  ```
  <return-param name="result" location="Header"
                type="xsd:string" />
  ```

必要に応じて、<params> 要素の <fault> 子要素を使用して、オペレーションの呼び出し中にエラーが発生した場合に送信する独自の Java 例外を指定します。この例外は、java.rmi.RemoteException 例外に加える形で送出されます。たとえば、次のように送出されます。

```
<fault name="MyServiceException"
       class-name="my.exceptions.MyServiceException" />
```

Sample web-services.xml Files

以下の節では、下記のタイプの WebLogic Web サービスに使用する web-services.xml ファイルのサンプルについて説明します。

- 組み込みデータ型を使用した EJB コンポーネント Web サービス
- 非組み込みデータ型を使用した EJB コンポーネント Web サービス
- EJB コンポーネントおよび SOAP メッセージ ハンドラ チェーンを使用する Web サービス
- SOAP メッセージ ハンドラ チェーンのみの Web サービス

組み込みデータ型を使用した EJB コンポーネント Web サービス

ある種の WebLogic Web サービスは、パラメータと戻り値が組み込みデータ型であるステートレス セッション EJB を使用して実装されます。次の Java インタフェースはそのような EJB の例です。

```
public interface SimpleStockQuoteService extends javax.ejb.EJBObject {
 public float getLastTradePrice(String ticker) throws java.rmi.RemoteException;
}
```

このサンプル EJB を使用して実装された Web サービスの web-services.xml デプロイメント記述子は次のようになります。

```
<web-services>
     <web-service name="stockquotes" targetNamespace="http://example.com"
                  uri="/myStockQuoteService">
          <components>
               <stateless-ejb name="simpleStockQuoteBean">
                    <ejb-link path="stockquoteapp.jar#StockQuoteBean" />
               </stateless-ejb>
          </components>
          <operations>
               <operation method="getLastTradePrice"
                          component="simpleStockQuoteBean" />
          </operations>
```

```
    </web-service>
</web-services>
```

例では、stockquotes という Web サービスが使用されています。その Web サービスは、ejb-jar.xml ファイルにある <ejb-name> が StockQuoteBean で、stockquoteapp.jar という EJB JAR ファイルにパッケージ化されているステートレス セッション EJB を使用して実装されています。このコンポーネントの内部名は simpleStockQuoteBean です。この Web サービスにはオペレーションが 1 つあり、その名前は EJB メソッド名と同じく、getLastTradePrice です。in パラメータと out パラメータは、メソッド シグネチャから推測されるので、web-services.xml ファイルで明示的に指定する必要はありません。

注意： servicegen Ant タスクは、web-services.xml ファイルにあるオペレーションのリストを生成する際、EJBObject のメソッドは取り込みません。

先の例では、Web サービスのオペレーションを明示的にリストする方法を示しています。次の例で示すように、<operation method="*"> 要素を 1 つのみ取り込むことで、EJB のすべてのパブリック メソッドを暗黙的に公開することも可能です。

```
<operations>
     <operation method="*"
                component="simpleStockQuoteBean" />
</operations>
```

Web サービスが HTTP のみをサポートする場合は、次の例で示すとおり、<web-service> 要素の protocol 属性を使用します。

```
<web-service name="stockquotes"
               targetNamespace="http://example.com"
               uri="/myStockQuoteService"
             protocol="https" >
...
</web-service>
```

非組み込みデータ型を使用した EJB コンポーネント Web サービス

比較的複雑な Web サービスとして、オペレーションがパラメータおよび戻り値として非組み込みデータ型を受け取るタイプがあります。これらの非組み込みデータ型は、直接 XML または SOAP のデータ型にはマップしないので、web-services.xml ファイルでデータ型を記述する必要があります。

たとえば、次のインタフェースは、2 つのメソッドが TradeResult オブジェクトを返す EJB を記述しています。

```
public interface Trader extends EJBObject {
  public  TradeResult buy (String stockSymbol, int shares)
    throws RemoteException;
  public TradeResult sell (String stockSymbol, int shares)
    throws RemoteException;
}
```

TradeResult クラスは次のようになります。

```
public class TradeResult implements Serializable {

  private int    numberTraded;
  private String stockSymbol;

  public TradeResult() {}

  public TradeResult(int nt, String ss) {
    numberTraded = nt;
    stockSymbol  = ss;
  }

  public int getNumberTraded() { return numberTraded; }
  public void setNumberTraded(int numberTraded) {
     this.numberTraded = numberTraded; }

  public String getStockSymbol() { return stockSymbol; }
  public void setStockSymbol(String stockSymbol) {
     this.stockSymbol = stockSymbol; }
}
```

次の web-services.xml ファイルは、EJB を使用して実装された Web サービスを記述しています。

```
<web-services>

  <web-service name="TraderService"
               uri="/TraderService"
               targetNamespace="http://www.bea.com/examples/Trader">

    <types>
      <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
              xmlns:stns="java:examples.webservices"
              attributeFormDefault="qualified"
              elementFormDefault="qualified"
                   targetNamespace="java:examples.webservices">
        <xsd:complexType name="TradeResult">
           <xsd:sequence><xsd:element maxOccurs="1" name="stockSymbol"
                                      type="xsd:string" minOccurs="1">
                         </xsd:element>
                         <xsd:element maxOccurs="1" name="numberTraded"
                                      type="xsd:int" minOccurs="1">
```

```
                          </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:schema>
      </types>

    <type-mapping>
      <type-mapping-entry
            deserializer="examples.webservices.TradeResultCodec"
            serializer="examples.webservices.TradeResultCodec"
            class-name="examples.webservices.TradeResult"
            xmlns:p1="java:examples.webservices"
            type="p1:TradeResult" >
      </type-mapping-entry>
    </type-mapping>

    <components>
      <stateless-ejb name="ejbcomp">
        <ejb-link path="trader.jar#TraderService" />
      </stateless-ejb>
    </components>

    <operations>
      <operation method="*" component="ejbcomp">
      </operation>
    </operations>

  </web-service>

</web-services>
```

この例では、<types> 要素は、XML スキーマ表記法を使用して TradeResult データ型の XML 表現を記述しています。<type-mapping> 要素には、<types> 要素で記述されている各データ型に対するエントリが入っています (この場合は TradeResult が 1 つのみ)。<type-mapping-entry> は、Java オブジェクトを作成する Java クラス ファイルのほか、XML と Java 間でデータを変換するシリアライゼーション クラスをリストします。

EJB コンポーネントおよび SOAP メッセージ ハンドラ チェーンを使用する Web サービス

また、SOAP の要求メッセージおよび応答メッセージをインターセプトする、ステートレス セッション EJB バックエンド コンポーネントと SOAP メッセージ ハンドラ チェーンの両方を使用して実装されるタイプの Web サービスもあります。次の web-services.xml ファイルのサンプルでは、そのような Web サービスが記述されています。

```
<web-services>
    <handler-chains>
        <handler-chain name="submitOrderCrypto">
            <handler class-name="com.example.security.EncryptDecrypt">
                <init-params>
                   <init-param name="elementToDecrypt" value="credit-info" />
                   <init-param name="elementToEncrypt" value="order-number" />
                </init-params>
            </handler>
        </handler-chain>
    </handler-chains>

    <web-service targetNamespace="http://example.com" name="myorderproc"
                 uri="myOrderProcessingService">
        <components>
            <stateless-ejb name="orderbean">
               <ejb-link path="myEJB.jar#OrderBean" />
            </stateless-ejb>
        </components>
        <operations xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
            <operation name="submitOrder" method="submit"
                       component="orderbean"
                       handler-chain="submitOrderCrypto" >
               <params>
                 <param name="purchase-order" style="in" type="xsd:anyType" />
                 <return-param name="order-number" type="xsd:string" />
               </params>
            </operation>
        </operations>
    </web-service>
</web-services>
```

この例では、SOAP の要求メッセージおよび応答メッセージにある情報の暗号化および復号化に使用される、submitOrderCrypto という SOAP メッセージ ハンドラ チェーンの入った Web サービスを示しています。このハンドラ チェーンに

は、com.example.security.EncryptDecrypt Java クラスを使用して実装されたハンドラが 1 つ組み込まれています。ハンドラは、SOAP メッセージにある暗号化および復号化が必要な要素を指定する初期化パラメータを 2 つ受け取ります。

Web サービスは orderbean というステートレス セッション EJB バックエンド コンポーネントを 1 つ定義します。

submitOrder オペレーションは、method、component、および handler-chain の各属性を組み合わせて指定することによってハンドラ チェーンとバックエンド コンポーネントを結合する方法を示しています。クライアント アプリケーションが submitOrder オペレーションを呼び出すと、submitOrderCrypto ハンドラ チェーンはまず、SOAP 要求を処理してクレジット カード情報を復号化します。次に、ハンドラ チェーンは orderbean EJB の submit() メソッドを呼び出し、SOAP メッセージから受け取ったパラメータを、purchase-order 入力パラメータも含めて変更を加えてから渡します。次に、submit() メソッドは、order-number を返します。返された値はハンドラ チェーンによって暗号化され、最終的に、暗号化された情報の入った SOAP 応答が、最初に submitOrder オペレーションを呼び出したクライアント アプリケーションに送信されます。

SOAP メッセージ ハンドラ チェーンのみの Web サービス

WebLogic Web サービスを、SOAP メッセージ ハンドラ チェーンのみを使用して実装し、バックエンド コンポーネントをまったく呼び出さない場合もあります。このタイプの Web サービスは、たとえば、既存のワークフロー処理システムに対するフロントエンドとして便利な場合があります。ハンドラ チェーンは、単に、SOAP 要求メッセージを受け取って、それをワークフロー システムに渡すだけで、ワークフロー システムがその後の処理をすべて実行します。

次の web-services.xml ファイルのサンプルでは、そのような Web サービスが記述されています。

```
<web-services>
    <handler-chains>
       <handler-chain name="enterWorkflowChain">
           <handler class-name="com.example.WorkFlowEntry">
               <init-params>
                   <init-param name="workflow-eng-jndi-name"
```

```
                                value="workflow.entry" />
                </init-params>
            </handler>
        </handler-chain>
    </handler-chains>

    <web-service targetNamespace="http://example.com"
                 name="myworkflow" uri="myWorkflowService">
        <operations xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
            <operation name="enterWorkflow"
                       handler-chain="enterWorkflowChain"
                       invocation-style="one-way" />
        </operations>
    </web-service>
</web-services>
```

この例では、enterWorkflowChain という SOAP メッセージ ハンドラ チェーンが 1 つ入った Web サービスを示しています。このハンドラ チェーンにはハンドラ (com.example.WorkFlowEntry Java クラスを使用して実装されたもの) が 1 つあり、初期化パラメータとして、既存ワークフロー システムの JNDI 名を受け取ります。

Web サービスは、enterWorkflow というオペレーションを 1 つ定義します。クライアント アプリケーションがこのオペレーションを呼び出すと、enterWorkflowChain ハンドラ チェーンは、SOAP の要求メッセージを受け取って、WebLogic Server で稼働する、workflow.entry という JNDI 名を持つワークフロー システムに渡します。このオペレーションは、非同期一方向方式オペレーションとして定義されます。つまり、クライアント アプリケーションは SOAP 応答を受け取りません。

enterWorkflow オペレーションは、method 属性および component 属性を指定しないので、バックエンド コンポーネントが Web サービスから直接呼び出されることはありません。したがって、web-services.xml ファイルで <components> 要素を指定する必要もありません。

8　Web サービスの呼び出し

この章では、クライアント アプリケーションから WebLogic および非 WebLogic の Web サービスを呼び出す方法について説明します。

Web サービスの呼び出しの概要

Web サービスの呼び出しは、クライアント アプリケーションが Web サービスを使用するために実行するアクションです。Web サービスを呼び出すクライアント アプリケーションは、Java、Microsoft SOAP Toolkit、Microsoft .NET などの任意のテクノロジを使用して記述できます。

注意： この章では、クライアント アプリケーションという用語を、WebLogic シン クライアントを使用して Web サービスを呼び出すスタンドアロン クライアントと、WebLogic Server で実行されている EJB の内部で動作しているクライアントの両方の意味で使用しています。

以降の節では、JAX-RPC 仕様の BEA の実装を使用して Java クライアント アプリケーションから Web サービスを呼び出す方法について説明します。WebLogic Web サービスとそのホーム ページを呼び出すのに必要な URL についての節を除いては、WebLogic Server で実行されている Web サービスに限らず、あらゆる Web サービスを呼び出す場合を想定しています。

WebLogic Server では、Web サービスの呼び出しに必要なすべてのクラス、インタフェース、およびスタブが含まれたオプションの Java JAR ファイルを提供しています。クライアント JAR ファイルには、特定の Web サービスの呼び出しに必要な Java コードの量を最小化するための Web サービス固有の実装のほか、`webserviceclient.jar` および `webserviceclient+ssl.jar` と呼ばれる JAX-RPC 仕様のクライアント実行時実装も含まれています。

JAX-RPC API

JAX-RPC (Java API for XML based RPC) は Sun Microsystems の仕様で、Web サービスを呼び出すクライアント API を定義します。

次の表で、JAX-RPC の中心的なインタフェースとクラスを簡単に説明します。

表 8-1 JAX-RPC のインタフェースとクラス

javax.xml.rpc のインタフェースまたはクラス	説明
Service	主要なクライアント インタフェース。静的、動的呼び出しの両方で使用する。
ServiceFactory	`Service` インスタンスの生成に使用するファクトリ クラス。
Stub	Web サービスのオペレーションを呼び出すためのクライアント プロキシを表す。通常は、Web サービスの静的呼び出しに使用する。

表 8-1 JAX-RPC のインタフェースとクラス

javax.xml.rpc のインタフェースまたはクラス	説明
Call	Web サービスを動的に呼び出すのに使用する。
JAXRPCException	Web サービスの呼び出し中にエラーが発生した場合に送出される例外。

WebLogic Server には、JAX-RPC 仕様の実装が組み込まれています。

JAX-RPC の詳細については、以下の Web サイトを参照してください。
http://java.sun.com/xml/jaxrpc/index.html

JAX-RPC を使用して Web サービスを呼び出す方法に関するチュートリアルについては、http://java.sun.com/webservices/docs/ea1/tutorial/doc/JAXRPC.html を参照してください。

Web サービスを呼び出すクライアントの例

WebLogic Server の *WL_HOME*/samples/server/src/examples/webservices ディレクトリには、WebLogic Web サービスの作成および呼び出しの方法として以下の例があります。*WL_HOME* は、WebLogic プラットフォームのメイン ディレクトリです。

- basic.statelessSession : パラメータと戻り値が組み込みデータ型の値となるステートレス セッション EJB バックエンド コンポーネントを使用します。
- basic.javaclass : パラメータと戻り値が組み込みデータ型の値となる Java クラス バックエンド コンポーネントを使用します。
- complex.statelessSession : パラメータと戻り値が非組み込みデータ型の値となるステートレス セッション EJB バックエンド コンポーネントを使用します。
- handler.log :ハンドラ チェーンとステートレス セッション EJB の両方を使用します。

- `handler.nocomponent`：バックエンド コンポーネントなしでハンドラ チェーンのみを使用します。
- `client.static`：非 WebLogic Web サービスを呼び出す静的クライアント アプリケーションの作成方法を示します。
- `client.static_out`：出力パラメータを使用する非 WebLogic Web サービスを呼び出す静的クライアント アプリケーションの作成方法を示します。
- `client.dynamic_wsdl`:WSDL を使用して非 WebLogic Web サービスを呼び出す動的クライアント アプリケーションの作成方法を示します。
- `client.dynamic_no_wsdl`:WSDL を使用しないで非 WebLogic Web サービスを呼び出す動的クライアント アプリケーションの作成方法を示します。

上記の例の構築と実行についての詳細な説明は、次の Web ページを参照してください。

WL_HOME`/samples/server/src/examples/webservices/package-summary.html`

WebLogic Web サービスの作成と呼び出しの例は、このほかにも、Web Services dev2dev Download Page の Web サービスのページに掲載されています。

Web サービスを呼び出す Java クライアント アプリケーションの作成：主な手順

Web サービスを呼び出す Java クライアント アプリケーションを作成するには、次の手順に従います。

1. WebLogic Server にある Java クライアント JAR ファイルを取得して、CLASSPATH に追加します。

   クライアント アプリケーションが WebLogic Server で実行されている場合は、この手順は省略することができます。

   注意： クライアント機能に関する現在の BEA のライセンス供与については、Web サイト BEA eLicense を参照してください。

   詳細については、8-5 ページの「Java クライアント アプリケーション JAR ファイルの取得」を参照してください。

2. Java クライアント アプリケーションのコードを記述します。

   さまざまなクライアント アプリケーション (静的、動的、など) の記述方法の詳細については、8-8 ページの「Java クライアント アプリケーションのコードの記述」と 8-19 ページの「J2ME クライアントの記述」を参照してください。

3. Java クライアント アプリケーションをコンパイルし、実行します。

Java クライアント アプリケーション JAR ファイルの取得

WebLogic Server は、以下のクライアント JAR ファイルを提供しています。

- JAX-RPC のクライアント実行時実装が入った、`webserviceclient.jar` と呼ばれる実行時 JAR ファイル。この JAR ファイルは、WebLogic Server 製品の一環として配布されます。
- SSL の実行時実装が入った、`webserviceclient+ssl.jar` と呼ばれる実行時 JAR ファイル。この JAR ファイルは、WebLogic Server 製品の一環として配布されます。
- J2ME の CDC プロファイル用の SSL の実行時実装が入った、`webserviceclient+ssl_pj.jar` と呼ばれる実行時 JAR ファイル。この JAR ファイルは、WebLogic Server 製品の一環として配布されます。
- `clientgen` Ant タスクで生成する Web サービス固有の JAR ファイル。このファイルには、JAX-RPC 仕様で定義された Web サービス固有のスタブが含まれます。クライアント アプリケーションから `Stub` や `Service` などの (WebLogic または非 WebLogic の) Web サービスを静的に呼び出すために使用するファイルです。必要なコードのほとんどすべてが自動的に生成されます。

  注意： 動的クライアント アプリケーションを作成する場合は、この JAR ファイルは使用しません。BEA Systems は、静的クライアントを使用して Web サービスを呼び出す場合を考えこのファイルを提供しています。

クライアント機能に関する現在の BEA のライセンス供与については、Web サイト BEA eLicense を参照してください。

クライアント JAR ファイルを取得するには、次の手順に従います。

1. *WL_HOME*\server\lib\webserviceclient.jar ファイルを、クライアント アプリケーション開発用のコンピュータにコピーします。*WL_HOME* は、WebLogic プラットフォームの最上位ディレクトリです。この JAR ファイルには、JAX-RPC のクライアント実行時実装が入っています。

   注意： SSL を使用して Web サービスの安全性を確保する場合および WebLogic Server が提供するクライアント クラスの SSL 実装を使用する場合は、*WL_HOME*\server\lib\webserviceclient+ssl.jar ファイルをクライアント アプリケーション開発用のコンピュータにコピーします。この JAR ファイルには、SSL 実装のほか、webserviceclient.jar にあるのと同じクライアント ファイルも入っています。

   SSL を使用する J2ME クライアントを記述する場合は、ファイル *WL_HOME*\server\lib\webserviceclient+ssl_pj.jar をクライアント アプリケーションのコンピュータにコピーします。

2. clientgen Ant タスクを実行して Web サービス固有クライアント JAR ファイルを生成します。

   wsdl 属性を指定して、任意の (非 WebLogic Web サービスも含めた) Web サービス用の JAR ファイルを作成します。あるいは、WebLogic Web サービスの場合は、EAR ファイルにパッケージ化されている ear 属性を指定します。

   clientgen Ant タスク実行の詳細および例については、8-7 ページの「clientgen Ant タスクを実行する」を参照してください。リファレンス情報については、付録 B「Web サービス Ant タスクとコマンドライン ユーティリティ」を参照してください。

   注意： WebLogic Web サービスを呼び出すクライアント アプリケーションを作成する際、クライアント JAR ファイルをホーム ページからダウンロードすることもできます。詳細については、8-23 ページの「WebLogic Web サービスのホーム ページおよび WSDL の URL」を参照してください。

3. クライアント JAR ファイルをクライアント コンピュータに配置し、そのファイルを見つけるための CLASSPATH 環境変数を更新します。

clientgen Ant タスクを実行する

clientgen Ant タスクを実行し、クライアント JAR ファイルを自動的に生成するには、次のように行います。

1. clientgen Ant タスクへの呼び出しの入った build.xml というファイルを作成します。詳細については、「clientgen Ant タスクのサンプル build.xml ファイル」を参照してください。
2. 環境を設定します。

   Windows NT では、setWLSEnv.cmd コマンドを実行します。*WL_HOME*\server\bin ディレクトリにあります。*WL_HOME* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

   UNIX では、setWLSEnv.sh コマンドを実行します。*WL_HOME*/server/bin ディレクトリにあります。*WL_HOME* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。
3. build.xml ファイルと同じディレクトリで ant と入力し、Ant タスクまたは build.xml ファイルで指定されたタスクを実行します。

   ```
   prompt> ant
   ```

clientgen Ant タスクのリファレンス情報については、B-12 ページの「clientgen」を参照してください。

clientgen Ant タスクのサンプル build.xml ファイル

以下に、簡単な build.xml ファイルの例を示します。

```
<project name="buildWebservice" default="generate-client">
   <target name="generate-client">
     <clientgen wsdl="http://example.com/myapp/myservice.wsdl"
           packageName="myapp.myservice.client"
                clientJar="c:/myapps/myService_client.jar"
/>

   </target>
</project>
```

この例では、clientgen タスクは、http://example.com/myapp/myservice.wsdl WSDL ファイルで記述されている Web サービスを呼び出すクライアント JAR ファイル (c:/myapps/myService_client.jar) を作成します。インタフェースとスタブファイルは myapp.myservice.client にパッケージ化されます。

Java クライアント アプリケーションのコードの記述

以下の節では、Java コードがほとんど不要な簡素な静的クライアントから、out パラメータを使用する比較的複雑なクライアントまで、Web サービスの呼び出しに使用されるさまざまな Java クライアント アプリケーションの記述方法について説明します。

すべての例で JAX-RPC API が使用されていて、BEA が提供する必要なクライアント JAR ファイルが CLASSPATH に追加されていることを前提としています。

Web サービスについての情報を取得

通常、クライアント コードの記述に先立って、Web サービスの名前とオペレーションのシグネチャを知る必要があります。

Web サービスの WSDL を参照してください。Web サービスの名前は、次の、TraderService WSDL の例に示すように、<service> 要素にあります。

```
<service name="TraderService">
  <port name="TraderServicePort"
       binding="tns:TraderServiceSoapBinding">
...
  </port>
</service>
```

この Web サービス用に定義されているオペレーションは、対応する <binding> 要素の下に記述されています。たとえば、次の WSDL の抜粋は、TraderService Web サービスには、buy と sell の 2 つのオペレーションがあることを示しています (わかりやすくするため、WSDL の関連部分のみを記載しています)。

```
<binding name="TraderServiceSoapBinding" ...>
  ...
  <operation name="sell">
  ...
  </operation>
  <operation name="buy">
  </operation>
</binding>
```

Web サービスのオペレーションのフル シグネチャを調べるには、Web サービス固有クライアント JAR ファイル (clientgen Ant タスクを使用して生成されています) を unJAR して、*.java ファイルそのものを調べてください。*ServiceName*Port.java ファイルには、Web サービスのインタフェース定義があります。*ServiceName* は Web サービスの名前です。たとえば、buy および sell オペレーションのシグネチャは、TraderServicePort.java ファイルにあります。

HTTP セッションの維持

クライアント アプリケーションが Web サービスのエンドポイントとの HTTP セッションに参加するかどうかを指定するには、アプリケーションで次のプロパティを設定します。

```
javax.xml.rpc.Call.SESSION_MAINTAIN_PROPERTY
```

クライアント アプリケーションが WebLogic Web サービスを呼び出すと、まず内部サーブレットが要求を処理し、クライアントごとに HttpSession オブジェクトを作成します。この HttpSession オブジェクトの有効期間は、標準の J2EE ガイドラインに従います。HttpSession オブジェクトの詳細については、「サーブレットからのセッション トラッキング」を参照してください。

Web サービスがクラッシュした場合の処理

WebLogic のクライアント JAR ファイルを使用しているクライアント アプリケーションは、Web サービスを最初に呼び出したときにその Web サービスが動作しているコンピュータの IP アドレスをキャッシュします。デフォルトでは、ここでキャッシュされたアドレスは新しく DNS がルックアップされるまでリフレッシュされません。そのため、Web サービスを呼び出した後にその Web サービスが動作しているコンピュータがクラッシュし、別の IP アドレスを持つ他のコンピュータでその機能を引き継いだ場合に、同じクライアント アプリケーションから再び Web サービスを呼び出そうとすると、呼び出しは失敗します。クライアント アプリケーションでは、以前にキャッシュした IP アドレスを持つコンピュータで引き続き Web サービスが動作していると見なすためです。つまり、新規に DNS をルックアップして再度 IP アドレスを解決しようとはせず、以前のルックアップでキャッシュされた情報を使用します。

この問題を解決するには、クライアント アプリケーションを更新して、JDK 1.4 のシステム プロパティの `sun.inetaddr.ttl` に、アプリケーションが IP アドレスをキャッシュしておく秒数を設定します。

警告 : この解決策はクライアント アプリケーションがスタンドアロンの場合にのみ有効です。クライアント アプリケーションが WebLogic Server 7.0 で動作している場合には、このプロパティを設定できません。このシステム プロパティは JDK バージョン 1.4 にのみ存在しており、WebLogic Server 7.0 ではバージョン 1.3.1 を使用しているためです。

簡素な静的クライアントの記述

動的クライアントの場合は、Web サービスのオペレーションとパラメータを間接的に参照しますが、静的クライアント アプリケーションを使用して Web サービスを呼び出す場合は、タイプ分けされた Java インタフェースを使用します。動的クライアントの使用は、Java reflection API を使用してメソッドを検索して呼び出すことに似ています。

Web サービスを静的に呼び出す場合は、Web サービス固有クライアント JAR を CLASSPATH に追加する必要があります。この JAR ファイルには、以下のクラスとインタフェースが入っています。

- Service インタフェースの Web サービス固有の実装。これはスタブ ファクトリとして機能します。スタブ ファクトリ クラスは、デフォルト コンストラクタにクライアント JAR ファイルを生成するのに使用した clientgen Ant タスクの wsdl 属性の値を使用します。
- WSDL の各 SOAP ポートのインタフェースと実装。
- 非組み込みデータ型とその Java 表現用のシリアライゼーション クラス。

以下のコードは、サンプルの TraderService Web サービスを呼び出すクライアント アプリケーションを記述する例です。この例では、TraderService はスタブ ファクトリで、TraderServicePort はスタブそのものです。

```
package examples.webservices.complex.statelessSession;

/**
 * このクラスは、JAX-RPC API を使用して TraderService Web サービスを呼び出し、
 * 以下のタスクを行う方法を示す
 * <ul>
 * <li> 株を 100 株購入する
 * <li> 株を 100 株売却する
 * </ul>
 *
 * TraderService Web サービスは、
 * ステートレス セッション EJB を使用して実装される
 *
 * @author Copyright (c) 1998-2002 by BEA Systems, Inc. All Rights Reserved.
 */

public class Client {

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

    //グローバルな JAXM メッセージ ファクトリをセットアップする
    System.setProperty("javax.xml.soap.MessageFactory",
      "weblogic.webservice.core.soap.MessageFactoryImpl");
  //グローバルな JAX-RPC サービス ファクトリをセットアップする
  System.setProperty( "javax.xml.rpc.ServiceFactory",
    "weblogic.webservice.core.rpc.ServiceFactoryImpl");

    //引数リストの解析
    Client client = new Client();
    String wsdl = (args.length > 0? args[0] :null);
    client.example(wsdl);
  }

  public void example(String wsdlURI) throws Exception {
```

```
    TraderServicePort trader = null;
    if (wsdlURI == null) {
      trader = new TraderService_Impl().getTraderServicePort();
    } else {
      trader = new TraderService_Impl(wsdlURI).getTraderServicePort();
    }
    String [] stocks = {"BEAS", "MSFT", "AMZN", "HWP" };

      // 購入の実行
    for (int i=0; i<stocks.length; i++) {
      int shares = (i+1) * 100;
      log("Buying "+shares+" shares of "+stocks[i]+".");
      TradeResult result = trader.buy(stocks[i], shares);
      log("Result traded "+result.getNumberTraded()
          +" shares of "+result.getStockSymbol());
    }
    // 売却の実行
    for (int i=0; i<stocks.length; i++) {
      int shares = (i+1) * 100;
      log("Selling "+shares+" shares of "+stocks[i]+".");
      TradeResult result = trader.sell(stocks[i], shares);
      log("Result traded "+result.getNumberTraded()
          +" shares of "+result.getStockSymbol());

    }

  }

  private static void log(String s) {
    System.out.println(s);
  }

}
```

上記の例で注目すべき主な点は以下のとおりです。

- 次のコードは、TraderServicePort スタブを作成する方法を示しています。

  ```
  trader = new TraderService_Impl().getTraderServicePort();
  ```

  TraderService_Impl スタブは JAX-RPC の Service インタフェースを実装します。TraderService_Impl のデフォルト コンストラクタは、clientgen Ant タスクによるクライアント JAR ファイルの作成時に指定された WSDL の URI に基づいてスタブを作成します。getTraderServicePort() メソッドは、TraderService スタブ実装のインスタンスを返すために使用される、Service.getPort() メソッドを実装します。

- 以下のコードは、TraderService Web サービスの buy オペレーションを呼び出す方法を示しています。

```
TradeResult result = trader.buy(stocks[i], shares);
```

trader Web サービスには、buy() と sell() の 2 つのオペレーションがあります。これらのオペレーションはともに、TradeResult という非組み込みデータ型を返します。

WSDL を使用する動的クライアントの記述

WSDL を使用する動的クライアントを作成するときは、まず、ServiceFactory.newInstance() メソッドを使用してサービス ファクトリを作成します。次に、そのファクトリから Service オブジェクトを作成し、それに WSDL および呼び出そうとする Web サービスの名前を渡します。次に、Service から Call オブジェクトを作成し、ポートの名前と実行しようとするオペレーションを渡し、最後に Call.invoke() メソッドを使用して、Web サービスのオペレーションを実際に呼び出します。

動的クライアントを記述するときは、clientgen Ant タスクを使用して生成された Web サービス固有クライアント JAR ファイルは使用しません。この JAR ファイルは静的クライアント専用です。ただし、WebLogic の JAX-RPC 仕様の実装が入った JAR ファイルは、CLASSPATH に含める必要があります。これらの JAR のファイルの詳細については、8-5 ページの「Java クライアント アプリケーション JAR ファイルの取得」を参照してください。

たとえば、WSDL を使用して次の URL にある Web サービスを呼び出す動的クライアント アプリケーションを作成するとします。

```
http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl
```

以下の Java コードは、その 1 つの方法を示しています。

```
/**
 * このクラスは、Web サービスを呼び出す Java クライアントを示す
 *
 * @author Copyright (c) 2002 by BEA Systems, Inc. All Rights Reserved.
 */

import java.net.URL;

import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;
```

```
import javax.xml.namespace.QName;

public class Main{

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

    //グローバルな JAXM メッセージ ファクトリをセットアップする
    System.setProperty("javax.xml.soap.MessageFactory",
      "weblogic.webservice.core.soap.MessageFactoryImpl");
  //グローバルな JAX-RPC サービス ファクトリをセットアップする
  System.setProperty( "javax.xml.rpc.ServiceFactory",
    "weblogic.webservice.core.rpc.ServiceFactoryImpl");

    //サービス ファクトリを作成
    ServiceFactory factory = ServiceFactory.newInstance();

    //QName を定義
    String targetNamespace =
      "http://www.themindelectric.com/"
      + "wsdl/net.xmethods.services.stockquote.StockQuote/";

    QName serviceName =
      new QName(targetNamespace,
        "net.xmethods.services.stockquote.StockQuoteService" );

    QName portName =
      new QName(targetNamespace,
        "net.xmethods.services.stockquote.StockQuotePort" );

    QName operationName = new QName( "urn:xmethods-delayed-quotes",
        "getQuote" );

    URL wsdlLocation =
      new
URL("http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl");

    //サービスを作成
    Service service = factory.createService( wsdlLocation, serviceName );

    //呼び出しを作成
    Call call = service.createCall( portName, operationName );

    //リモートの Web サービスを呼び出し
    Float result = (Float) call.invoke(new Object[] {
      "BEAS"
    });

    System.out.println("\n");
    System.out.println("This example shows how to create a dynamic client
                        application that invokes a non-WebLogic Web service.");
    System.out.println("The webservice used was:
```

```
http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl");
    System.out.println("The quote for BEAS is: ");
    System.out.println( result );
  }

}
```

注意： javax.xml.rpc.Call API を使用して WSDL を使用する動的クライアントを作成する場合、以下のメソッドはクライアント アプリケーションで使用できません。

- getParameterTypeByName()
- getReturnType()

また、getTargetEndpointAddress() メソッドを実行する必要がある場合は、たとえ WSDL で targetEndPointAddress を利用できる場合でも、事前に setTargetEndpointAddress() メソッドが実行されている必要があります。

WSDL を使用しない動的クライアントの記述

WSDL を使用しない動的クライアントも WSDL を使用するクライアントに似ていますが、オペレーションに対するパラメータ、ターゲットのエンドポイント アドレスなど、WSDL にある情報を明示的に設定する必要があります。

以下の例は、クライアント アプリケーションで WSDL を指定せずに Web サービスを呼び出すクライアント アプリケーションの作成方法を示しています。

```
/**
 * このクラスは、Web サービスを呼び出す Java クライアントを示す
 *
 * @author Copyright (c) 2002 by BEA Systems, Inc. All Rights Reserved.
 */

import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;

import javax.xml.namespace.QName;

public class Main{
```

```
public static void main(String[] args) throws Exception {
//グローバルな JAX-RPC サービス ファクトリをセットアップする
System.setProperty( "javax.xml.rpc.ServiceFactory",
  "weblogic.webservice.core.rpc.ServiceFactoryImpl");

  //サービス ファクトリを作成
  ServiceFactory factory = ServiceFactory.newInstance();

  //QName を定義
  String targetNamespace =
    "http://www.themindelectric.com/"
    + "wsdl/net.xmethods.services.stockquote.StockQuote/";

  QName serviceName =
    new QName(targetNamespace,
      "net.xmethods.services.stockquote.StockQuoteService" );

  QName portName =
    new QName(targetNamespace,
      "net.xmethods.services.stockquote.StockQuotePort" );

  QName operationName = new QName( "urn:xmethods-delayed-quotes",
      "getQuote" );

  //サービスを作成
  Service service = factory.createService( serviceName );

  //呼び出しを作成
  Call call = service.createCall();

  //ポートとオペレーション名を設定
  call.setPortTypeName( portName );
  call.setOperationName( operationName );
//パラメータを追加
  call.addParameter("symbol",
      new QName( "http://www.w3.org/2001/XMLSchema", "string" ),
                  ParameterMode.IN);

  call.setReturnType(new QName( "http://www.w3.org/2001/XMLSchema","float") );

  //エンドポイント アドレスの設定
  call.setTargetEndpointAddress("http://www.xmethods.com:9090/soap");

  //リモートの Web サービスを呼び出し
  Float result = (Float) call.invoke(new Object[] {
    "BEAS"
  });

  System.out.println("\n");
  System.out.println("This example shows how to create a dynamic client
```

```
                        application that invokes a non-WebLogic Web service.");
    System.out.println("The webservice used was:

http://www.themindelectric.com/wsdl/net.xmethods.services.stockquote.StockQuote
");
    System.out.println("The quote for BEAS is:");
    System.out.println( result );
  }

}
```

注意： WSDL を使用しない動的クライアントでは、`getPorts()` メソッドは常に `null` を返します。この動作は、WSDL を使用する動的クライアントとは異なります。WSDL を使用する動的クライアントの場合、メソッドは実際にポートを返します。

out または inout パラメータを使用するクライアントの記述

Web サービスは、複数の値を返す方法として out または inout パラメータを使用する場合があります。

out または inout パラメータを使用する Web サービスを呼び出すクライアントアプリケーションを記述する場合、out または inout パラメータのデータ型は、`javax.xml.rpc.holders.Holder` インタフェースを実装する必要があります。クライアント アプリケーションは、Web サービスを呼び出すと、`Holder` オブジェクトの out または inout パラメータを要求して、それらを標準の戻り値と同様に扱います。

たとえば、次の WSD L で記述される Web サービスには、`echoStructAsSimpleTypes()` という、標準の in パラメータを 1 つと out パラメータを 3 つ受け取るオペレーションがあります。

```
http://soap.4s4c.com/ilab/soap.asp?WSDL
```

以下の静的クライアント アプリケーションは、この Web サービスを呼び出す 1 つの方法を示しています。アプリケーションは、`clientgen` Ant タスクを使用して生成された、スタブ クラスの入った Web サービス固有クライアント JAR ファイルが CLASSPATH に追加されているものとみなします。

```
package websvc;
```

```
/**
 * このクラスは、Web サービスを呼び出す Java クライアントを示す
 *
 * @author Copyright (c) 2002 by BEA Systems, Inc. All Rights Reserved.
 */

public class Main{

  public static void main(String[] args) throws Exception {
  //グローバルな JAX-RPC サービス ファクトリをセットアップする
  System.setProperty( "javax.xml.rpc.ServiceFactory",
    "weblogic.webservice.core.rpc.ServiceFactoryImpl");

    InteropLab_Impl test = new InteropLab_Impl();
    InteropTest2PortType soap = test.getinteropTest2PortType();

    org.tempuri.x4s4c.x1.x3.wsdl.types.SOAPStruct inputStruct =
      new org.tempuri.x4s4c.x1.x3.wsdl.types.SOAPStruct();

    inputStruct.setVarInt( 10 );
    inputStruct.setVarFloat( 10.1f );
    inputStruct.setVarString( "hi there" );

    javax.xml.rpc.holders.StringHolder outputString =
         new javax.xml.rpc.holders.StringHolder();
    javax.xml.rpc.holders.IntHolder outputInteger =
         new javax.xml.rpc.holders.IntHolder();
    javax.xml.rpc.holders.FloatHolder outputFloat =
        new javax.xml.rpc.holders.FloatHolder();

    soap.echoStructAsSimpleTypes(inputStruct, outputString, outputInteger,
                                 outputFloat);

    System.out.println("This example shows how to create a static client
                        application that invokes a non-WebLogic Web service.");
    System.out.println("The webservice used was:
                      http://soap.4s4c.com/ilab/soap.asp?WSDL");
    System.out.println("This webservice shows how to invoke an operation that
                       uses out parameters.  The set parameters are below:");
    System.out.println("outputString.value: " + outputString.value);
    System.out.println("outputInteger.value: " + outputInteger.value);
    System.out.println("outputFloat.value: " + outputFloat.value);
  }

}
```

J2ME クライアントの記述

J2ME (Java 2 Platform, Micro Edition) Web サービス固有クライアント JAR ファイルを作成して、J2ME で実行されるクライアント アプリケーションと併用することができます。

注意： サポート対象の J2ME 環境は、CDC および Foundation プロファイルです。

Web サービスを呼び出す J2ME クライアント アプリケーションの作成方法は、非 J2ME クライアントの作成方法とほとんど同じです。たとえば、非 J2ME クライアント アプリケーションと同じ実行時クライアント JAR ファイル (*WL_HOME*\server\lib\webserviceclient.jar) を使用します。

J2ME クライアント アプリケーションを記述するには、8-4 ページの「Web サービスを呼び出す Java クライアント アプリケーションの作成 : 主な手順」の手順を行いますが、以下の変更点があります。

- `clientgen` Ant タスクを実行して Web サービス固有のクライアント JAR ファイルを生成するときは、次の例に示すように、必ず、`j2me="True"` 属性を指定します。

```
<clientgen wsdl="http://example.com/myapp/myservice.wsdl"
           packageName="myapp.myservice.client"
           clientJar="c:/myapps/myService_clients.jar"
           j2me="True"
/>
```

  注意： `clientgen` で生成された J2ME Web サービス固有クライアント JAR ファイルは、次の点で JAX-RPC 仕様には適合していません。

  - 生成されたスタブのメソッドは、`java.rmi.RemoteException` を送出しない。
  - 生成されたスタブは、`java.rmi.Remote` を拡張しない。

- Java クライアント アプリケーションを記述、コンパイル、実行する際は、必ず、J2ME の仮想マシンと API を使用します。

J2ME についての詳細は、http://java.sun.com/j2me/ を参照してください。

SSL を使用する J2ME クライアントの記述

WebLogic Server では、SSL を使用する J2ME クライアント アプリケーションを作成できます。SSL を使用する J2ME クライアントを記述する場合は、前節のガイドラインだけでなく以下のガイドラインにも従ってください。

- 以下のクラスとパッケージを使用する必要がある。
  - `java.math.BigInteger` (クラス)
  - `java.util.*` (パッケージ)
- ファイル *`WL_HOME`*`\server\lib\webserviceclient+ssl_pj.jar` をクライアント アプリケーションのコンピュータにコピーして、それを CLASSPATH に追加する。

  警告 : `weblogic.jar` ファイルは、CLASSPATH に設定しないでください。
- クライアント アプリケーションが WSDL ファイルを使用して Web サービスを呼び出す場合は、クライアント コンピュータに格納された WSDL ファイルのローカル コピーを使用する必要がある。`URLConnection` オブジェクトを使用して WSDL ファイルにアクセスすることはできません。

ポータブル スタブの作成と使い方

Web サービスのクライアント JAR ファイル (製品の一環として配布されたものでも `clientgen` Ant タスクで生成された Web サービス固有のものでも) を WebLogic Server で動作するアプリケーションの一部として使用する場合、JAR ファイルの Java クラスと WebLogic Server の Java クラスが衝突することがあります。クライアント JAR ファイルがデプロイされている WebLogic Server と、そのクライアント JAR ファイルが生成された WebLogic Server のバージョンが異なる場合、この問題はより頻発します。この問題を解決するために、ポータブル スタブを使用します。

注意 : ポータブル スタブを使用する必要があるのは、クライアント アプリケーションが WebLogic Server にデプロイされ、かつ動作している場合のみです。クライアント アプリケーションがスタンドアロンの場合、ポータブル スタブを使用する必要はありません。

クライアント アプリケーションでポータブル スタブを使用可能にする方法を、以下に示します。

1. クライアント アプリケーションにおいて、一般的な `webserviceclient.jar` クライアント JAR ファイルではなく、`wsclient70.jar` (WebLogic Server と共に *`WL_HOME`*`\server\lib` ディレクトリに配布) と呼ばれる WebLogic Server の各リリースに固有のクライアント JAR ファイルを使用します。`wsclient70.jar` には、標準のクライアント JAR ファイルと同じクラス ファイルを `weblogic70.*` というファイル名に変更したものが含まれています。これらのクラス ファイルはバージョンごとに固有のため、WebLogic Server の `weblogic.*` クラスと衝突することはありません。
2. `clientgen` Ant タスクで生成した Web サービス固有のクライアント JAR ファイルだけでなく、サポートするすべてのクライアント JAR ファイルを `VersionMaker` ユーティリティを介して実行します。このユーティリティを使用すると、クライアント JAR ファイルのクラスが以下のように変更されます。
   - すべての `weblogic.*` クラスが `weblogic70.*` に変更される。
   - すべての `weblogic.*` クラスへの参照が `weblogic70.*` への参照に変更される。

   こうして変更された新しいバージョン固有のクライアント JAR ファイルを、クライアント アプリケーションで使用します。

   `VersionMaker` の使用方法の詳細については、8-21 ページの「VersionMaker ユーティリティの使い方」を参照してください。

VersionMaker ユーティリティの使い方

`weblogic.webservice.tools.versioning.VersionMaker` ユーティリティでは、以下の引数を使用します。

- `destination_dir` : 新しいバージョン固有のクライアント JAR ファイルを含む送り先ディレクトリ。
- `client_jar_file` : `clientgen` Ant タスクで生成したクライアント JAR ファイルで、クラス ファイル名を `weblogic.*` から `weblogic70.*` に変更する必要があるもの。
- `other_jar_files` : サポートする JAR ファイル。

クライアント JAR ファイルを更新して、バージョン固有の WebLogic Server クラスを使用するには、以下の手順に従います。

1. 環境を設定します。

   Windows NT では、`setWLSEnv.cmd` コマンドを実行します。*`WL_HOME`*`\server\bin` ディレクトリにあります。*`WL_HOME`* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

   UNIX では、`setWLSEnv.sh` コマンドを実行します。*`WL_HOME`*`/server/bin` ディレクトリにあります。*`WL_HOME`* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

2. `weblogic.webservice.tools.versioning.VersionMaker` を以下の例のように実行します。

   ```
   java weblogic.webservice.tools.versioning.VersionMaker \
      new_directory myclient.jar supporting.jar
   ```

   この例では、`myclient.jar` および `supporting.jar` というクライアント JAR ファイルの `weblogic.*` クラスを `weblogic70.*` という名前に変更し、それらのクラスへの参照をすべて更新しています。カレント ディレクトリ内の `new_directory` というディレクトリに、新しいクライアント JAR ファイルが生成されます。

WebLogic Web サービス クライアントとのプロキシ サーバの使用

プロキシ サーバを使用すると、WebLogic Web サービスのクライアント アプリケーションから、Web サービスのホスト サーバ (WebLogic または非 WebLogic) にリクエストをプロキシできます。ただし、クライアント アプリケーションで必ず以下のすべてのシステム プロパティを設定する必要があります。

- `http.proxyHost`
- `http.proxyPort`
- `weblogic.webservice.transport.http.proxy.host`
- `weblogic.webservice.transport.http.proxy.port`

注意： Web サービスを呼び出すときのプロトコルとして HTTPS を使用する場合は、上記のプロパティの `http` を `https` に置き換えます。たとえば、`http.proxyHost` ではなく `https.proxyHost` を使用します。

以上のプロパティや、クライアント アプリケーションで設定できる他の WebLogic システム プロパティの詳細については、8-27 ページの「WebLogic Web サービスのシステム プロパティ」を参照してください。

また、プロキシ認証を使用するようにプロキシ サーバが設定されている場合は、クライアント アプリケーションから抜粋した次の例のように、クライアント アプリケーションのプロパティ `weblogic.net.proxyAuthenticatorClassName` を、`weblogic.common.ProxyAuthentication` インタフェースを実装する Java クラスの名前に設定することも必要です。

```
    System.setProperty("weblogic.net.proxyAuthenticatorClassName",
"my.ProxyAuthenticator");
```

この例の `my.ProxyAuthenticator` は、`weblogic.common.ProxyAuthentication` インタフェースを実装する、クライアント アプリケーションの CLASSPATH にあるクラスです。

`weblogic.common.ProxyAuthentication` インタフェースを使用すると、ユーザ認証を必要とするプロキシ サーバを介して WebLogic HTTP および SSL プロトコルをトンネリングする場合に、必要とされるユーザ認証情報をクライアント アプリケーションが提供できるようになります。このインタフェースの実装の詳細については、`weblogic.common.ProxyAuthentication` Javadoc を参照してください。

WebLogic Web サービスのホーム ページおよび WSDL の URL

WebLogic Server にデプロイされているすべての Web サービスには、ホーム ページがあります。ホーム ページからは、以下の操作ができます。

- サービスを記述する WSDL の表示

- クライアント アプリケーションからその Web サービスを呼び出すのに必要なインタフェース、クラス、スタブの入った Web サービス固有クライアント JAR ファイルのダウンロード

  注意： このクライアント JAR ファイルをダウンロードするためのリンクは、クライアント JAR ファイル名が *WebServiceName*_client.jar である場合にのみ、ホーム ページに表示されます。ここで、*WebServiceName* は、この Web サービスの名前で、web-services.xml ファイルにある、<web-service> 要素の name 属性で指定されています。これ以外の Web サービスについては、clientgen Ant タスクを使用して JAR ファイルを作成する必要があります。詳細については、6-10 ページの「clientgen Ant タスクを実行する」を参照してください。

- 各オペレーションの動作のテスト

  Web サービスのテストの一環として、SOAP 要求にある、非組み込みデータ型を記述する XML を編集して、相互運用性関連の問題をデバッグすることができます。

- オペレーションを実行した場合の SOAP の要求メッセージおよび応答メッセージの表示

以下の URL では、まず、Web サービスのホーム ページを呼び出す方法を示し、次に WSDL を表示します。

[*protocol*]://[*host*]:[*port*]/[*contextURI*]/[*serviceURI*]

[*protocol*]://[*host*]:[*port*]/[*contextURI*]/[*serviceURI*]?WSDL

各値の説明は次のとおりです。

- *protocol* は、サービスを呼び出すのに使用されるプロトコルで、HTTP か HTTPS のいずれかです。この値は、web-servicex.xml ファイルにある、Web サービスを記述する <web-service> 要素の protocol 属性の値と一致しています。servicegen Ant タスクを使用して Web サービスをアセンブルした場合は、この値は protocol 属性に対応します。
- *host* は、WebLogic Server が動作しているコンピュータの名前です。
- *port* は、WebLogic Server がリスンしているポート番号 (デフォルト値は 7001) です。
- *contextURI* は、Web アプリケーションのコンテキスト ルートで、EAR ファイルの application.xml デプロイメント記述子の <context-root> 要素に

対応しています。servicegen Ant タスクを使用して Web サービスをアセンブルした場合は、この値は contextURI 属性に対応します。

application.xml ファイルに <context-root> 要素がない場合は、contextURI の値は、Web アプリケーションのアーカイブ ファイルまたは展開ディレクトリの名前です。

- *serviceURI* は Web サービスの URI です。この値は、web-services.xml ファイルにある <web-service> 要素の uri 属性に対応します。servicegen Ant タスクを使用して Web サービスをアセンブルした場合は、この値は serviceURI 属性に対応します。

たとえば、次の build.xml ファイルを使用して、servicegen Ant タスクを使用して WebLogic Web サービスをアセンブルしたとします。

```
<project name="buildWebservice" default="build-ear">
  <target name="build-ear">
      <servicegen
         destEar="myWebService.ear"
         warName="myWAR.war"
         contextURI="web_services">
         <service
             ejbJar="myEJB.jar"
         targetNamespace="http://www.bea.com/examples/Trader"
         serviceName="TraderService"
         serviceURI="/TraderService"
         generateTypes="True"
             expandMethods="True" >
         </service>
     </servicegen>
  </target>
</project>
```

Web サービスが ariel というホストのデフォルトのポート番号で実行されているとすると、その Web サービスのホーム ページを呼び出す URL は次のとおりです。

```
http://ariel:7001/web_services/TraderService
```

この Web サービスの自動生成による WSDL を取得する URL は次のとおりです。

```
http://ariel:7001/web_services/TraderService?WSDL
```

Web サービス呼び出し中のエラーのデバッグ

Web サービス (WebLogic、非 WebLogic のいずれであっても) の呼び出し中にエラーが発生した場合、生成された SOAP の要求メッセージおよび応答メッセージを表示すると、多くの場合、問題が明らかになり便利です。

SOAP の要求メッセージおよび応答メッセージを表示するには、`-Dweblogic.webservice.verbose=true` フラグを使用してクライアント アプリケーションを実行します。以下に、`runService` というクライアント アプリケーションを実行する例を示します。

```
prompt> java -Dweblogic.webservice.verbose=true runService
```

SOAP の要求および応答メッセージが、クライアント アプリケーションを実行したコマンド ウィンドウにフル出力されます。

また、WebLogic Server の起動時に `-Dweblogic.webservice.verbose=true` フラグを指定することによって、デプロイされた WebLogic Web サービスが呼び出されるたびに、SOAP の要求メッセージおよび応答メッセージが出力されるように、WebLogic Server をコンフィグレーションすることもできます。SOAP メッセージは、WebLogic Server を起動したコマンド ウィンドウに出力されます。

注意 : デバッグ メッセージのコマンド ウィンドウへの出力という余分の作業によりパフォーマンスが低下することが考えられるので、この WebLogic Server のフラグは開発段階でのみ設定するようお勧めします。

WebLogic Web サービスのシステム プロパティ

次の表に、Web サービスを呼び出すクライアント アプリケーションで設定できる WebLogic Web サービスのシステム プロパティを示します。プロパティの設定には `System.setProperty()` メソッドを使用します。

表 8-2 WebLogic Web サービスのシステム プロパティ

システム プロパティ	説明	データ型
weblogic.webservice.transport.http.full-url	クライアント アプリケーションが呼び出している Web サービスの URL の指定は、HTTP リクエストの `Request-URI` フィールドで相対的に行うのではなく、完全な URL として行うことを指定する。 有効な値は `True` および `False`。デフォルト値は `False`。	Boolean
weblogic.webservice.transport.http.proxy.host	プロキシ サーバを使用して HTTP 接続を行う場合は、クライアント アプリケーションで、このシステム プロパティを使用してプロキシ サーバのホスト名を指定する。	String
weblogic.webservice.transport.http.proxy.port	プロキシ サーバを使用して HTTP 接続を行う場合は、クライアント アプリケーションで、このシステム プロパティを使用してプロキシ サーバのポートを指定する。	String
weblogic.webservice.transport.https.proxy.host	プロキシ サーバを使用して HTTPS (HTTP over SSL) 接続を行う場合は、クライアント アプリケーションで、このシステム プロパティを使用してプロキシ サーバのホスト名を指定する。	String

表 8-2 WebLogic Web サービスのシステム プロパティ

システム プロパティ	説明	データ型
weblogic.webservice.transport.https.proxy.port	プロキシ サーバを使用して HTTPS (HTTP over SSL) 接続を行う場合は、クライアント アプリケーションで、このシステム プロパティを使用してプロキシ サーバのポートを指定する。	String
weblogic.webservice.verbose	Web サービスの呼び出し時に verbose モードを有効にして、SOAP リクエストおよび応答のメッセージを表示できるようにする。 有効な値は `True` および `False`。デフォルト値は `False`。 詳細については、8-26 ページの「Web サービス呼び出し中のエラーのデバッグ」を参照。	Boolean
weblogic.webservice.client.ssl.strictcertchecking	WebLogic 提供の SSL 実装を使用する場合に厳密な証明書検証を有効または無効にする。 `True` に設定すると厳密な証明書検証が有効になり、`False` では無効になる。デフォルト値は `False`。 例については、11-8 ページの「WebLogic Server が提供する SSL 実装を使用する」を参照。	Boolean
weblogic.webservice.client.ssl.trustedcertfile	CA (Certificate Authority : 認証局) の証明書が格納されている、クライアント アプリケーションのコンピュータに配置されたファイルの名前。このファイルに証明書のある CA は、信頼を受けて WebLogic Server 証明書の発行が任されている。このファイルには、ユーザ自身が信頼する証明書も格納できる。	String

表 8-2 WebLogic Web サービスのシステム プロパティ

システム プロパティ	説明	データ型
weblogic.webservice.client.ssl.adapterclass	サードパーティの SSL 実装を使用するために実装されたアダプタ クラスの完全修飾名。 例については、11-12 ページの「サードパーティの SSL 実装を使用する」を参照。	String
weblogic.http.KeepAliveTimeoutSeconds	リクエストをタイム アウトするまで HTTP キープアライブを維持する秒数。HTTP キープアライブを使用しない場合は、このプロパティを 0 に設定する。 デフォルト値は 30 秒。	Integer.

9　非組み込みデータ型の使用法

この章では、WebLogic Web サービスにおける非組み込みデータ型の使用法について説明します。

- 9-1 ページの「非組み込みデータ型の使用法の概要」
- 9-2 ページの「非組み込みデータ型を手動で作成する方法 : 主な手順」

非組み込みデータ型の使用法の概要

パラメータおよび戻り値に非組み込みデータ型が使用される WebLogic Web サービスを作成することができます。非組み込みデータ型は、サポートされている組み込みデータ型以外のデータ型として定義されます。例としては、`int` や `String` があります。組み込みデータ型の完全なリストは、5-13 ページの「組み込みデータ型の使用法」を参照してください。

WebLogic Server は、組み込みデータ型の XML 表現と Java 表現間の変換を透過的に処理します。Web サービスのオペレーションが非組み込みデータ型を使用する場合は、WebLogic Server によって変換が実行されるように、以下の情報を入力する必要があります。

- データを XML 表現と Java 表現間で変換するシリアライゼーション クラス
- そのデータ型の Java 表現を格納する Java クラス
- そのデータ型の XML スキーマ表現
- `web-services.xml` デプロイメント記述子ファイルにあるデータ型マッピング情報

WebLogic Server には、`servicegen` Ant タスクおよび `autotype` Ant タスクが入っていて、作成した Web サービスのステートレス セッション EJB または Java クラス バックエンド コンポーネントを参照することにより、前述のコンポーネ

ントを自動的に生成します。これらの Ant タスクは、さまざまな非組み込みデータ型を処理できるので、一般的には、プログラマが手動でコンポーネントを作成する必要はありません。

非組み込みデータ型コンポーネントを手動で作成しなければならない場合もあります。データ型が複雑すぎて、Ant タスクが正しくコンポーネントを生成できないことがあります。あるいは、データの XML 表現と Java 表現間の変換方法に対して、WebLogic Server が使用するデフォルトの変換プロシージャに依存せず、より強力なコントロールを実装したいという場合もあるでしょう。

サポートされている非組み込みデータ型の詳細なリストは、6-14 ページの「servicegen および autotype Ant タスクでサポートされる非組み込みデータ型」を参照してください。

`servicegen` および `autotype` を使用する手順についての説明は、第 6 章「Ant タスクを使用した WebLogic Web サービスのアセンブル」を参照してください。リファレンス情報については、付録 B「Web サービス Ant タスクとコマンドライン ユーティリティ」を参照してください。

非組み込みデータ型を手動で作成する方法：主な手順

次の手順は、非組み込みデータ型の作成方法と、`servicegen` Ant タスクを使用したデプロイ可能な Web サービスの作成方法を示しています。

1. 使用するデータ型の XML スキーマ表現を記述します。9-4 ページの「XML スキーマ データ型表現を記述する」を参照してください。
2. 使用するデータ型の Java クラスを記述します。9-5 ページの「Java データ型表現を記述する」を参照してください。
3. XML 表現と Java 表現間のデータ変換を行うシリアライゼーション クラスを記述します。9-6 ページの「シリアライゼーション クラスを記述する」を参照してください。
4. Java コードをクラスにコンパイルします。CLASSPATH 変数でそれらのクラスの場所を特定できるようにしてください。

5. 非組み込みデータ型のデータ型マッピング情報が格納されたテキスト ファイルを作成します。9-11 ページの「データ型マッピング ファイルを作成する」を参照してください。
6. 6-3 ページの「servicegen Ant タスクを使用した WebLogic Web サービスのアセンブル」の説明に従って `servicegen` Ant タスクを使用して Web サービスをアセンブルします。`servicegen` Ant タスクを呼び出す `build.xml` ファイルを作成する場合は、必ず、`servicegen` の `typeMappingFile` 属性を指定し、それを前の手順で作成したデータ型マッピング ファイルの名前に設定してください。

   EAR ファイルではなく、`.ear` サフィックスを持たない値を `servicegen` の `destEar` 属性に指定して展開ディレクトリを作成することをお勧めします。その展開ディレクトリは、後で、Web サービスがデプロイできる状態になったときに EAR ファイルにパッケージ化できます。
7. 最初の手順で作成したデータ型の XML スキーマ表現を追加して、(`servicegen` Ant タスクで生成された) `web-services.xml` ファイルを更新します。9-12 ページの「XML スキーマ情報で web-services.xml ファイルを更新する」を参照してください。
8. 展開ディレクトリを Web サービスとしてデプロイするか、そのディレクトリを EAR ファイルにパッケージ化して、それを WebLogic Server にデプロイします。
9. `clientgen` Ant タスクを使用して Java クライアントを生成する必要がある場合は、6-10 ページの「clientgen Ant タスクを実行する」の手順に従います。ただし、`clientgen` を呼び出す `build.xml` ファイルについては以下のことを行います。
   - `ear` 属性を指定し、それを Web サービスの EAR ファイルの完全パス名に設定する。`wsdl` 属性は指定しないでください。
   - `useServerTypes` 属性を指定し、それを `True` に設定する。

XML スキーマ データ型表現を記述する

Web サービスは、サービスとそのサービスを呼び出すクライアント アプリケーション間のデータ送信のメッセージ フォーマットとして SOAP を使用します。SOAP は、XML ベースのプロトコルなので、Web サービスのオペレーションが使用する非組み込みデータ型の構造を記述するには、XML スキーマ表記法が必要です。

警告： XML スキーマは、強力かつ複雑なデータ記述言語であり、気の弱い方にはお勧めできません。

次の例では、`EmployBean` という非組み込みデータ型を記述する XML スキーマを示しています。

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
              xmlns:stns="java:examples.newTypes"
              attributeFormDefault="qualified"
              elementFormDefault="qualified"
            targetNamespace="java:examples.newTypes">
   <xsd:complexType name="EmployeeBean">
      <xsd:sequence>
        <xsd:element name="name"
                       type="xsd:string"
                       nillable="true"
                       minOccurs="1"
                     maxOccurs="1">
        </xsd:element>
        <xsd:element name="id"
                            type="xsd:int"
                       minOccurs="1"
                     maxOccurs="1">
        </xsd:element>
      </xsd:sequence>
   </xsd:complexType>
</xsd:schema>
```

次の XML は、`EmployeeBean` データ型のインスタンスを示しています。

```
<EmployeeBean>
  <name>Beverley Talbott</name>
  <id>1234</id>
</EmployeeBean>
```

XML スキーマ表記法で非組み込みデータ型を記述する方法の詳細については、「XML スキーマ仕様」を参照してください。

Java データ型表現を記述する

非組み込みデータ型の Java 表現は、Web サービスのオペレーションを実装する EJB または Java クラスで使用します。

以下は、前節でその XML 表現を説明した `EmployeeBean` データ型の Java 表現の例です。

```
package examples.newTypes;

/**
 * @author Copyright (c) 2002 by BEA Systems.All Rights Reserved.
 */

public final class EmployeeBean {

  private String name = "John Doe";
  private int id = -1;

  public EmployeeBean() {
  }

  public EmployeeBean(String n, int i) {
    name = n;
    id = i;
  }

  public String getName() {
    return name;
  }
  public void setName(String  v) {
    this.name = v;
  }

  public int getId() {
    return id;
  }
  public void setId(int  v) {
    this.id = v;
  }

  public boolean equals(Object obj) {
    if (obj instanceof EmployeeBean) {
      EmployeeBean e = (EmployeeBean) obj;
      return (e.name.equals(name) && (e.id == id));
    }
    return false;
```

```
  }
}
```

シリアライゼーション クラスを記述する

シリアライゼーション クラスは、データを XML 表現と Java 表現間で実際に変換するクラスです。データをシリアライズまたはデシリアライズするメソッドの入ったクラスを 1 つのみ記述します。このクラスでは、WebLogic XML Streaming API を使用して XML データを処理します。

WebLogic XML Streaming API は、XML ドキュメントの消費および生成を行う簡単で直観的な手段となります。この API により、一定の手順に従ったストリームベースの XML ドキュメントの扱いが可能になります。

WebLogic XML Streaming API の使用法の詳細については、『WebLogic XML プログラマーズ ガイド』参照してください。

次の例は、XML Streaming API を使用して、9-4 ページの「XML スキーマ データ型表現を記述する」および 9-5 ページの「Java データ型表現を記述する」で説明したデータ型をシリアライズおよびデシリアライズするクラスを示しています。この例を記したリストの後に、このようなクラスを作成する主な手順を示します。

```
package examples.newTypes;

import weblogic.webservice.encoding.AbstractCodec;

import weblogic.xml.schema.binding.DeserializationContext;
import weblogic.xml.schema.binding.DeserializationException;
import weblogic.xml.schema.binding.Deserializer;
import weblogic.xml.schema.binding.SerializationContext;
import weblogic.xml.schema.binding.SerializationException;
import weblogic.xml.schema.binding.Serializer;


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

```
import weblogic.xml.stream.XMLStreamException;


public final class EmployeeBeanCodec extends
  weblogic.webservice.encoding.AbstractCodec
{
  public void serialize(Object obj,
                XMLName name,
                XMLOutputStream writer,
                SerializationContext context)
    throws SerializationException
  {
    EmployeeBean emp = (EmployeeBean) obj;

    try {

      // 外部開始要素
      writer.add(ElementFactory.createStartElement(name));

      // 従業員名要素
      writer.add(ElementFactory.createStartElement("name"));
      writer.add(ElementFactory.createCharacterData(emp.getName()));
      writer.add(ElementFactory.createEndElement("name"));

      // 従業員 ID 要素
      writer.add(ElementFactory.createStartElement("id"));
      String id_string = Integer.toString(emp.getId());
      writer.add(ElementFactory.createCharacterData(id_string));
      writer.add(ElementFactory.createEndElement("id"));

      // 外部終了要素
      writer.add(ElementFactory.createEndElement(name));

    } catch(XMLStreamException xse) {
      throw new SerializationException("stream error", xse);
    }
  }

  public Object deserialize(XMLName name,
                    XMLInputStream reader,
                    DeserializationContext context)
    throws DeserializationException
  {
  // リーダから必要な情報を抽出
  // このデータ型を表現する要素全体を消費する
  // オブジェクトを作成し、返す
    EmployeeBean employee = new EmployeeBean();
```

```
    try {
      if (reader.skip(name, XMLEvent.START_ELEMENT)) {
        StartElement top = (StartElement)reader.next();

        //次の開始要素は従業員名
        if (reader.skip(XMLEvent.START_ELEMENT)) {
          StartElement emp_name = (StartElement)reader.next();

          //次の要素は名前の文字データとする
          CharacterData cdata = (CharacterData) reader.next();
          employee.setName(cdata.getContent());
    } else {
          throw new DeserializationException("employee name not found");
        }

        //次の開始要素は従業員 ID
        if (reader.skip(XMLEvent.START_ELEMENT)) {
          StartElement emp_id = (StartElement)reader.next();

          //次の要素は ID の文字データとする
          CharacterData cdata = (CharacterData) reader.next();
          employee.setId(Integer.parseInt(cdata.getContent()));
    } else {
          throw new DeserializationException("employee id not found");
        }

        //要素全体を消費しなければ、
        //このストリームを良好な状態で終了してその他のデシリアライザに移動できない
        if (reader.skip(name, XMLEvent.END_ELEMENT)) {
          XMLEvent end = reader.next();
    } else {
          throw new DeserializationException("expected end element not found");
        }
    } else {
        throw new DeserializationException("expected start element not found");
      }
    } catch (XMLStreamException xse) {
      throw new DeserializationException("stream error", xse);
    }
    return employee;
  }

  public Object deserialize(XMLName name,
                    Attribute att,
                    DeserializationContext context)
    throws DeserializationException
  {
     //注意 : この例では使用していない
```

```
    // 属性から必要な情報を抽出
  // このデータ型を表現する要素全体を消費する
  // オブジェクトを作成し、返す
    return new EmployeeBean();
  }
}
```

WebLogic XML Streaming API を使用してシリアライゼーション クラスを作成するには、次の手順に従います。

1. 以下のクラスをインポートします。これらは、シリアライゼーション クラスが拡張する抽象クラスから実装されます。

   ```
   import weblogic.xml.schema.binding.DeserializationContext;
   import weblogic.xml.schema.binding.DeserializationException;
   import weblogic.xml.schema.binding.Deserializer;
   import weblogic.xml.schema.binding.SerializationContext;
   import weblogic.xml.schema.binding.SerializationException;
   import weblogic.xml.schema.binding.Serializer;
   ```

2. 必要であれば、WebLogic XML Streaming API クラスをインポートします。先の例では、次のクラスがインポートされています。

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

3. Java クラスを記述して、次の抽象クラスを拡張します。

   ```
   weblogic.webservice.encoding.AbstractCodec
   ```

4. `serialize()` メソッドを実装します。データを Java から XML へ変換するのに使用するメソッドです。このメソッドのシグネチャは次のとおりです。

   ```
   void serialize(Object obj,
                  XMLName name,
                  XMLOutputStream writer,
                  SerializationContext context)
     throws SerializationException;
   ```

   作成した Java オブジェクトは、`Object` パラメータに格納されます。XML Streaming API を使用して、Java オブジェクトを `XMLOutputStream` パラ

メータに記述します。結果として作成された要素の名前として、XMLName パラメータを使用します。

警告： SerializationContext パラメータは、更新しないでください。WebLogic Server で内部的に使用されるパラメータです。

5. deserialize() メソッドを実装します。データを XML から Java へ変換するのに使用するメソッドです。このメソッドのシグネチャは次のとおりです。

```
Object deserialize(XMLName name,
                   XMLInputStream reader,
                   DeserializationContext context)
  throws DeserializationException;
```

デシリアライズする XML は、XMLInputStream パラメータにあります。WebLogic XML Streaming API を使用してこの XML を解析し、返される Object に変換します。XMLName パラメータには、この XML 要素名として予想される名前が格納されています。

deserialize() メソッドを再帰的に呼び出して、格納されている Object を構築します。

XML Streaming API を使用して XML ドキュメントを構成するイベントのストリームを読み込む際は、必ず、1 つの要素をその EndElement イベントまで完全に読み込んでください。実際のデータの終わりで読み込みを終了しないでください。EndElement イベントの手前で読み込みを終了すると、以降の要素のデシリアライゼーションが失敗するおそれがあります。

警告： DeserializationContext パラメータは、更新しないでください。WebLogic Server で内部的に使用されるパラメータです。

6. シリアライゼーション クラスを作成する対象となっているデータ型が XML ファイルで引数の値として使用されている場合は、次のような、deserialize() メソッドのバリエーションを実装します。

```
Object deserialize(XMLName name,
                   Attribute att,
                   DeserializationContext context)
  throws DeserializationException;
```

Attribute パラメータには、デシリアライズすべき属性値が格納されています。XMLName 属性には、この XML 要素名として予想される名前が格納されています。

警告： DeserializationContext パラメータは、更新しないでください。WebLogic Server で内部的に使用されるパラメータです。

データ型マッピング ファイルを作成する

データ型マッピング ファイルは、web-services.xml デプロイメント記述子ファイルの一要素です。このファイルによって、非組み込みデータ型に関する情報の一部が 1 つの場所にまとめられます。たとえば、データの Java 表現を記述する Java クラスの名前や、XML と Java の間でデータを変換するシリアライゼーション クラスの名前などです。servicegen Ant タスクでは、非組み込みデータ型を使用する WebLogic Web サービスの web-services.xml デプロイメント記述子を作成するときにこのデータ型マッピング ファイルが使用されます。

データ型マッピング ファイルを作成するときには、次の手順に従います。

1. 任意の名前でテキスト ファイルを作成します。
2. そのテキスト ファイルで、次のように <type-mapping> ルート要素を追加します。

   ```
   <type-mapping>
   ...
   </type-mapping>
   ```

3. シリアライゼーション クラスを作成した非組み込みデータ型ごとに、<type-mapping> 要素の <type-mapping-entry> 子要素を追加します。属性には以下のものがあります。
   - xmlns:*name* - ネームスペースを宣言します。
   - class-name - Java クラスの完全修飾名を指定します。
   - type - このデータ型マッピング エントリが適用される XML スキーマ型を指定します。
   - serializer - データを Java 表現から XML 表現へ変換するシリアライゼーション クラスの完全修飾名です。このクラスを作成する方法の詳細については、9-6 ページの「シリアライゼーション クラスを記述する」を参照してください。
   - deserializer - データを XML 表現から Java 表現へ変換するシリアライゼーション クラスの完全修飾名です。このクラスを作成する方法の詳細については、9-6 ページの「シリアライゼーション クラスを記述する」を参照してください。

次の例は、9-12 ページの「XML スキーマ情報で web-services.xml ファイルを更新する」の XML スキーマ データ型用に <type-mapping> エントリが 1 つ追加されたデータ型マッピング ファイルを示しています。

```
<type-mapping>
    <type-mapping-entry
        xmlns:p2="java:examples.newTypes"
        class-name="examples.newTypes.EmployeeBean"
        type="p2:EmployeeBean"
        serializer="examples.newTypes.EmployeeBeanCodec">
        deserializer="examples.newTypes.EmployeeBeanCodec"
    </type-mapping-entry>
</type-mapping>
```

XML スキーマ情報で web-services.xml ファイルを更新する

servicegen で生成された web-services.xml ファイルには、独自にカスタム シリアライゼーション クラスを作成した非組み込みデータ型の XML スキーマ情報が格納されません。したがって、次の手順に従って、XML スキーマ情報を手動でデプロイメント記述子に追加する必要があります。

1. servicegen Ant タスクで生成された既存の web-services.xml ファイルで、<web-service> 要素の子要素 <types> を見つけます。

   ```
   <types>
   ...
   </types>
   ```

2. 9-4 ページの「XML スキーマ データ型表現を記述する」で作成した非組み込みデータ型の XML スキーマ表現を、次の例のように、<types> 要素内の既存の情報にマージします。

   ```
   <types>
     <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 xmlns:stns="java:examples.newTypes"
                 attributeFormDefault="qualified"
                 elementFormDefault="qualified"
                 targetNamespace="java:examples.newTypes">
        <xsd:complexType name="EmployeeBean">
           <xsd:sequence>
             <xsd:element name="name"
                          type="xsd:string"
   ```

```
                        nillable="true"
                        minOccurs="1"
                        maxOccurs="1">
            </xsd:element>
            <xsd:element name="id"
                            type="xsd:int"
                        minOccurs="1"
                        maxOccurs="1">
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
    </xsd:schema>
</types>
```

10 SOAP メッセージをインターセプトする SOAP メッセージ ハンドラの作成

この章では、WebLogic Web サービスの開発中に、SOAP メッセージ ハンドラを使用して SOAP の要求メッセージおよび応答メッセージをインターセプトする方法について説明します。

- 10-1 ページの「SOAP メッセージ ハンドラおよびハンドラ チェーンの概要」
- 10-2 ページの「SOAP メッセージ ハンドラの作成：主な手順」
- 10-3 ページの「SOAP メッセージ ハンドラおよびハンドラ チェーンの設計」
- 10-6 ページの「ハンドラインタフェースの実装」
- 10-17 ページの「SOAP メッセージ ハンドラ情報による web-services.xml ファイルの更新」
- 10-19 ページの「クライアント アプリケーションでの SOAP メッセージ ハンドラおよびハンドラ チェーンの使用」

SOAP メッセージ ハンドラおよびハンドラ チェーンの概要

SOAP メッセージ ハンドラは、Web サービスの要求と応答の両方で SOAP メッセージをインターセプトします。ハンドラは、Web サービスそのものと Web サービスを呼び出すクライアント アプリケーションの両方で作成することができます。ハンドラをどのような場合に使用するかについては、4-6 ページの「SOAP メッセージをインターセプトする SOAP メッセージ ハンドラの使用」を参照してください。

次の表は、javax.xml.rpc.handler API の主なクラスおよびインタフェースの説明です。それらを使用してハンドラを作成する方法についてはこの章で後述します。

表 10-1 JAX-RPC ハンドラのインタフェースとクラス

javax.xml.rpc.handler のクラスとインタフェース	説明
Handler	ハンドラ作成時に実装する主なインタフェース。SOAP の要求、応答、および障害を処理するメソッドが格納されている。
HandlerInfo	web-services.xml ファイルで指定されている初期化パラメータをはじめとする、ハンドラについての情報が格納されている。
MessageContext	ハンドラによって処理されたメッセージのコンテキストを抽出する。MessageContext プロパティにより、ハンドラ チェーン内のハンドラは処理ステートを共有できる。
soap.SOAPMessageContext	SOAP メッセージの取得または更新に使用される MessageContext インタフェースのサブインタフェース。
javax.xml.soap.SOAPMessage	実際の SOAP の要求メッセージまたは応答メッセージが入ったオブジェクト。ヘッダー、本文、添付ファイルが含まれる。

SOAP メッセージ ハンドラの作成：主な手順

WebLogic Web サービス開発中に、SOAP の要求メッセージおよび応答メッセージをインターセプトする SOAP メッセージ ハンドラを作成するには、以下の手順に従います。

1. ハンドラとハンドラ チェーンを設計します。10-3 ページの「SOAP メッセージ ハンドラおよびハンドラ チェーンの設計」を参照してください。

2. ハンドラ チェーン内の各ハンドラについて、`javax.xml.rpc.handler.Handler` インタフェースを実装する Java クラスを作成します。10-6 ページの「ハンドラインタフェースの実装」を参照してください。

   WebLogic Server には、ハンドラ クラスのコーディングの簡素化に役立つ、JAX-RP C ハンドラ API の拡張クラスが組み込まれています。`weblogic.webservice.GenericHandler` という抽象クラスです。10-14 ページの「GenericHandler Abstract クラスの拡張」を参照してください。

3. この Java コードをクラス ファイルにコンパイルします。第 6 章「Ant タスクを使用した WebLogic Web サービスのアセンブル」の説明にあるとおり、これらのクラス ファイルは、後で、デプロイ可能な Web サービスの EAR ファイルとしてパッケージ化します。

4. 適切な情報で、`web-services.xml` デプロイメント記述子ファイルを更新します。10-17 ページの「SOAP メッセージ ハンドラ情報による web-services.xml ファイルの更新」を参照してください。

クライアント サイドの SOAP メッセージ ハンドラおよびハンドラ チェーンの作成については、10-19 ページの「クライアント アプリケーションでの SOAP メッセージ ハンドラおよびハンドラ チェーンの使用」を参照してください。

SOAP メッセージ ハンドラおよびハンドラ チェーンの設計

SOAP メッセージ ハンドラを設計するときには、以下の事項を決める必要があります。

- すべての作業を実行するために必要なハンドラの数
- 実行の順序
- バックエンド コンポーネントを呼び出すかどうか、または Web サービスがハンドラ チェーンだけで構成されるのかどうか

ハンドラ チェーン内の各ハンドラには、SOAP の要求メッセージを処理するメソッドと SOAP の応答メッセージを処理するメソッドが 1 つずつあります。これらのハンドラは、`web-services.xml` デプロイメント記述子ファイルで指定します。順序付けされたハンドラのグループが、ハンドラ チェーンです。

Web サービスを呼び出すときに、WebLogic Server は次のようにハンドラを実行します。

1. ハンドラ チェーン内のハンドラの `handleRequest()` メソッドはすべて、`web-services.xml` ファイルで指定されている順序で実行されます。これらの `handleRequest()` メソッドのいずれもが、SOAP の要求メッセージを変更できます。
2. ハンドラ チェーン内の最後のハンドラの `handleRequest()` メソッドが実行されると、WebLogic Server は、Web サービスを実装するバックエンド コンポーネントを呼び出し、最終的な SOAP の要求メッセージを渡します。

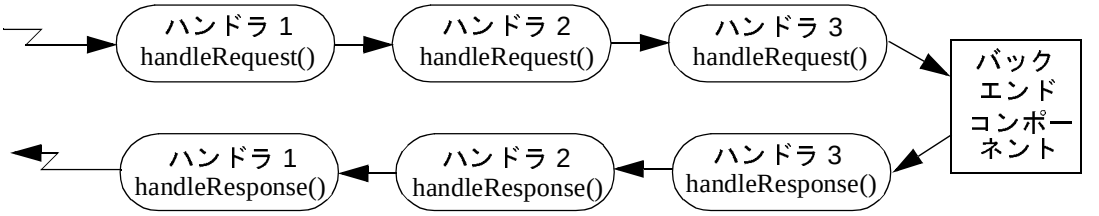
注意： この操作は、この Web サービスに対してバックエンド コンポーネントが実際に定義されている場合にのみ発生します。ハンドラ チェーンのみで構成される Web サービスを開発することもできます。

3. バックエンド コンポーネントの実行が終了したら、ハンドラ チェーン内のハンドラの `handleResponse()` メソッドが、`web-services.xml` ファイルで指定されているのとは逆の順序で実行されます。これらの `handleResponse()` メソッドのいずれもが、SOAP の応答メッセージを変更できます。
4. ハンドラ チェーン内の最初のハンドラの `handleResponse()` メソッドが実行されると、WebLogic Server は、Web サービスを呼び出したクライアント アプリケーションに最終的な SOAP 応答メッセージを返します。

たとえば、次の例のように、`web-services.xml` デプロイメント記述子で 3 つのハンドラが指定されている、`myChain` というハンドラ チェーンを指定したとしましょう。

```
<handler-chains>
   <handler-chain name="myChain">
     <handler class-name="myHandlers.handlerOne" />
     <handler class-name="myHandlers.handlerTwo" />
     <handler class-name="myHandlers.handlerThree" />
   </handler-chain>
</handler-chains>
```

次の図は、WebLogic Server で各ハンドラの `handleRequest()` メソッドと `handleResponse()` メソッドが実行される順序を示しています。



通常、SOAP の要求メッセージと応答メッセージの両方で同じタイプの処理が必要となるので、各 SOAP メッセージ ハンドラには、それぞれを処理する別個のメソッドがあります。たとえば、SOAP 要求にあるセキュア データを復号化する `handleRequest()` メソッドと、SOAP 応答を暗号化する `handleResponse()` メソッドを持つ暗号化ハンドラを設計する場合があります。

一方、SOAP 要求のみを処理し、それと同等の応答処理は行わないハンドラを設計することもできます。

また、ハンドラ チェーン内の次のハンドラを呼び出さず、そのまま、任意の時点でクライアント アプリケーションに応答を送信することも可能です。その方法は、以下の節で説明します。

最後に、ハンドラ チェーンにハンドラのみが存在し、バックエンド コンポーネントは 1 つも存在しない Web サービスを設計することができます。この場合、最後のハンドラにある `handleRequest()` メソッドが実行されると、`handleResponse()` メソッドが自動的に呼び出されます。`web-services.xml` ファイルを使用して、バックエンド コンポーネントは指定せず、ハンドラ チェーンのみで Web サービスを実装するように指定する例については、10-17 ページの「SOAP メッセージ ハンドラ情報による web-services.xml ファイルの更新」を参照してください。

ハンドラインタフェースの実装

SOAP メッセージ ハンドラ クラスは、次の例のように、`javax.rpc.xml.handler.Handler` インタフェースを実装する必要があります。この例は、SOAP の要求および応答メッセージを出力する簡単な方法を示しています。

```
package examples.webservices.handler.log;

import java.util.Map;

import javax.xml.rpc.handler.Handler;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;

import weblogic.logging.NonCatalogLogger;

/**
 * @author Copyright (c) 2002 by BEA Systems.All Rights Reserved.
 */

public final class LogHandler
  implements Handler
{
  private NonCatalogLogger log;

  private HandlerInfo handlerInfo;

  public void init(HandlerInfo hi) {
    log = new NonCatalogLogger("WebService-LogHandler");
    handlerInfo = hi;
  }

  public void destroy() {}

  public QName[ ] getHeaders() { return handlerInfo.getHeaders(); }

  public boolean handleRequest(MessageContext mc) {
    SOAPMessageContext messageContext = (SOAPMessageContext) mc;

    System.out.println("** Request:"+messageContext.getMessage().toString());
    log.info(messageContext.getMessage().toString());
    return true;
  }
```

```
  public boolean handleResponse(MessageContext mc) {

    SOAPMessageContext messageContext = (SOAPMessageContext) mc;

    System.out.println("** Response:"+messageContext.getMessage().toString());
    log.info(messageContext.getMessage().toString());
    return true;
  }

  public boolean handleFault(MessageContext mc) {
    SOAPMessageContext messageContext = (SOAPMessageContext) mc;

    System.out.println("** Fault:"+messageContext.getMessage().toString());
    log.info(messageContext.getMessage().toString());
  }
}
```

javax.xml.rpc.handler.Handler インタフェースには、実装すべき次のメソッドがあります。

- init()
- destroy()
- getHeaders()
- handleRequest()
- handleResponse()
- handleFault()

以降の節では、各メソッドを使用して実装をコーディングする方法を説明します。

Handler.init() メソッドを実装する

Handler.init() メソッドは、Handler オブジェクトのインスタンスを作成し、そのインスタンスが自動的に初期化されるようにするために呼び出されます。そのシグネチャは次のとおりです。

```
public void init(HandlerInfo config) throws JAXRPCException {}
```

HandlerInfo オブジェクトには、web-services.xml ファイルで指定されている初期化パラメータをはじめとする、SOAP メッセージ ハンドラについての情報が格納されています。HandlerInfo.getHandlerConfig() メソッドを使用してパラメータを取得すると、このメソッドは、名前と値の組み合わせの入った Map オブジェクトを返します。

初期化パラメータを処理する必要がある場合、または、他の初期化タスクを実行する必要がある場合は、init() メソッドを実装します。

初期化パラメータの使用例としては、デバッグのオン、オフ、メッセージやエラーを書き込むためのログ ファイルの名前の指定などがあります。

Handler.destroy() メソッドを実装する

Handler.destroy() メソッドは、Handler オブジェクトのインスタンスを破壊するために呼び出されます。そのシグネチャは次のとおりです。

```
public void destroy() throws JAXRPCException {}
```

destroy() メソッドを実装すると、ハンドラのライフサイクルを通じて取得されたあらゆるリソースを解放できます。

Handler.getHeaders() メソッドを実装する

Handler.getHeaders() メソッドは、このハンドラ インスタンスで処理されたヘッダー ブロックを取得します。そのシグネチャは次のとおりです。

```
public QName[ ] getHeaders() {}
```

Handler.handleRequest() メソッドを実装する

Handler.handleRequest() メソッドは、バックエンド コンポーネントによって処理される前に SOAP の要求メッセージをインターセプトするために呼び出されます。そのシグネチャは次のとおりです。

```
public boolean handleRequest(MessageContext mc) throws JAXRPCException  {}
```

このメソッドを実装して、バックエンド コンポーネントによって処理される前に SOAP メッセージにあるデータを復号化し、要求に含まれているパラメータの数が正しいかどうかなどの確認ができます。

`MessageContext` オブジェクトは、SOAP メッセージ ハンドラによって処理されたメッセージのコンテキストを抽出します。`MessageContext` プロパティにより、ハンドラ チェーン内のハンドラは処理ステートを共有できます。

`MessageContext` の `SOAPMessageContext` サブインタフェースを使用すると、SOAP の要求メッセージの内容を取得または更新することができます。SOAP の要求メッセージそのものは、`javax.xml.soap.SOAPMessage` オブジェクトに格納されます。このオブジェクトの詳細については、10-13 ページの「javax.xml.soap.SOAPMessage オブジェクト」を参照してください。

`SOAPMessageContext` クラスは、SOAP 要求を処理する次の 2 つのメソッドを定義します。

- `SOAPMessageContext.getMessage()`：SOAP の要求メッセージの入った `javax.xml.soap.SOAPMessage` オブジェクトを返します。
- `SOAPMessageContext.setMessage(javax.xml.soap.SOAPMessage)`：SOAP の要求メッセージを変更後、それを更新します。

SOAP 要求のすべての処理をコーディングしたら、次のいずれかを実行します。

- `true` を返すことにより、ハンドラ要求チェーン内の次のハンドラを呼び出します。

  要求チェーンの次のハンドラは、`web-services.xml` デプロイメント記述子にある `<handler-chain>` 要素の次の `<handler>` 下位要素として指定されています。チェーン内のすべてのハンドラが実行されたら、Web サービスのコンフィグレーションによって、メソッドは、バックエンド コンポーネントの呼び出し、最終的な SOAP の要求メッセージの送信、または最後のハンドラの `handleResponse()` メソッドの呼び出しを行います。

- `false` を返すことにより、ハンドラ要求チェーンの処理をブロックします。

  ハンドラ要求チェーンの処理をブロックすると、Web サービスのこの呼び出しについては、バックエンド コンポーネントは実行されなくなります。Web サービスの一部の呼び出しの結果をキャッシュしてあり、そのリストに現在の呼び出しも含まれている場合は、この操作が便利な場合があります。

  ハンドラ要求チェーンの処理を中断されますが、WebLogic Server が、現在のハンドラから順に、*response* チェーンを呼び出します。たとえば、ハンド

ラ チェーンに、handlerA と handlerB という 2 つのハンドラがあると想定します。ここで、handlerA の `handleRequest()` メソッドは、handlerB の対応するメソッドの前に呼び出されるものとします。handlerA で処理がブロックされると (したがって handlerB の `handleRequest()` メソッドは呼び出されない)、ハンドラ応答チェーンは handlerA で開始され、handlerB の `handleRequest()` メソッドも呼び出されません。

- `javax.xml.rpc.soap.SOAPFaultException` を送出して、SOAP 障害を示します。

  `handleRequest()` メソッドが `SOAPFaultException` を送出すると、WebLogic Server は、この例外を検出して、ハンドラ要求チェーンの後続の処理を終了し、このハンドラの `handleFault()` メソッドを呼び出します。

- あらゆるハンドラ固有実行時エラーに対して `JAXRPCException` を送出します。

  `handleRequest()` メソッドが `JAXRPCException` を送出すると、WebLogic Server は、この例外を検出して、ハンドラ要求チェーンの後続の処理を終了し、この例外を WebLogic Server のログファイルに記録し、このハンドラの `handleFault()` メソッドを呼び出します。

Handler.handleResponse() メソッドを実装する

`Handler.handleResponse()` メソッドは、バックエンド コンポーネントによって処理された SOAP の応答メッセージをインターセプトするために呼び出されますが、この Web サービスを呼び出したクライアント アプリケーションに返される前に呼び出されます。そのシグネチャは次のとおりです。

```
public boolean handleResponse(MessageContext mc) throws JAXRPCException  {}
```

このメソッドを実装すると、クライアント アプリケーションに返される前に SOAP メッセージに含まれているデータを暗号化したり、戻り値に処理を加えたりできます。

`MessageContext` オブジェクトは、SOAP メッセージ ハンドラによって処理されたメッセージのコンテキストを抽出します。`MessageContext` プロパティにより、ハンドラ チェーン内のハンドラは処理ステートを共有できます。

MessageContext の SOAPMessageContext サブインタフェースを使用すると、SOAP の応答メッセージの内容を取得または更新することができます。SOAP の応答メッセージそのものは、javax.xml.soap.SOAPMessage オブジェクトに格納されます。10-13 ページの「javax.xml.soap.SOAPMessage オブジェクト」を参照してください。

SOAPMessageContext クラスは、SOAP 応答を処理する次の 2 つのメソッドを定義します。

- SOAPMessageContext.getMessage()：SOAP の応答メッセージの入った javax.xml.soap.SOAPMessage オブジェクトを返します。
- SOAPMessageContext.getMessage(javax.xml.soap.SOAPMessage)：SOAP の応答メッセージを変更後、それを更新します。

SOAP 応答のすべての処理をコーディングしたら、次のいずれかを実行します。

- true を返すことにより、ハンドラ応答チェーン内の次のハンドラを呼び出します。

  ハンドラチェーンの次の応答は、web-services.xml デプロイメント記述子にある <handler-chain> 要素の先行する <handler> 下位要素として指定されています (先にも述べましたが、ハンドラチェーンにある応答は、web-services.xml ファイルで指定されているのとは逆の順序で実行されます。詳細は 10-3 ページの「SOAP メッセージ ハンドラおよびハンドラ チェーンの設計」を参照してください)。

  ハンドラ チェーン内のすべてのハンドラが実行されたら、メソッドは、この Web サービスを呼び出したクライアント アプリケーションに最終的な SOAP の応答メッセージを返します。

- false を返すことにより、ハンドラ応答チェーンの処理をブロックします。

  ハンドラ応答チェーンの処理をブロックすると、応答チェーンにある残りのハンドラが、Web サービスのこの呼び出しについては実行されなくなり、現在の SOAP メッセージがクライアント アプリケーションに返されます。

- あらゆるハンドラ固有実行時エラーに対して JAXRPCException を送出します。

  handleRequest() メソッドが JAXRPCException を送出すると、WebLogic Server は、この例外を検出して、ハンドラ要求チェーンの後続の処理を終了し、この例外を WebLogic Server のログファイルに記録し、このハンドラの handleFault() メソッドを呼び出します。

Handler.handleFault() メソッドを実装する

Handler.handleFault() メソッドは、SOAP メッセージ処理モデルに基づいて SOAP 障害を処理します。そのシグネチャは次のとおりです。

```
public boolean handleFault(MessageContext mc) throws JAXRPCException  {}
```

このメソッドを実装すると、バックエンドコンポーネントによって生成された障害のほか、handleResponse() メソッドおよび handleRequest() メソッドによって生成されたあらゆる SOAP 障害の処理も扱うことができます。

MessageContext オブジェクトは、SOAP メッセージ ハンドラによって処理されたメッセージのコンテキストを抽出します。MessageContext プロパティにより、ハンドラ チェーン内のハンドラは処理ステートを共有できます。

MessageContext の SOAPMessageContext サブインタフェースを使用すると、SOAP メッセージの内容を取得または更新することができます。SOAP メッセージそのものは、javax.xml.soap.SOAPMessage オブジェクトに格納されます。10-13 ページの「javax.xml.soap.SOAPMessage オブジェクト」を参照してください。

SOAPMessageContext クラスは、SOAP メッセージを処理する次の 2 つのメソッドを定義します。

- SOAPMessageContext.getMessage() : SOAP メッセージの入った javax.xml.soap.SOAPMessage オブジェクトを返します。
- SOAPMessageContext.getMessage(javax.xml.soap.SOAPMessage) : SOAP メッセージを変更後、それを更新します。

SOAP 障害のすべての処理をコーディングしたら、次のいずれかを実行します。

- true を返すことにより、ハンドラ チェーン内の次のハンドラの handleFault() メソッドを呼び出します。
- false を返すことにより、ハンドラ障害チェーンの処理をブロックします。

javax.xml.soap.SOAPMessage オブジェクト

javax.xml.soap.SOAPMessage 抽象クラスは、JAXM (Java API for XML Messaging) 仕様の構成要素です。このクラスは、SOAP メッセージ ハンドラ作成時に SOAP の要求メッセージおよび応答メッセージを操作するのに使用します。この節では、SOAPMessage オブジェクトの基本構造と SOAP メッセージの表示および更新に役立つメソッドのいくつかについて説明します。

SOAPMessage オブジェクトは、SOAPPart オブジェクト (SOAP XML ドキュメントそのものが入っている)、または同オブジェクトと添付ファイルで構成されています。

SOAPMessage クラスの詳細な説明については、JAXM API Javadocs を参照してください。JAXM の詳細については、http://java.sun.com/xml/jaxm/index.html を参照してください。

SOAPPart オブジェクト

SOAPPart オブジェクトには、SOAPEnvelope オブジェクトに格納された XML SOAP ドキュメントが入っています。このオブジェクトは、実際の SOAP のヘッダーと本文を取得するのに使用します。

次の Java コードのサンプルは、Handler クラスによって提供された MessageContext オブジェクトから SOAP メッセージを取り出して、その各部を参照する方法を示しています。

```
SOAPMessage soapMessage =  messageContext.getRequest();
SOAPPart soapPart = soapMessage.getSOAPPart();
SOAPEnvelope soapEnvelope = soapPart.getEnvelope();
SOAPBody soapBody = soapEnvelope.getBody();
SOAPHeader soapHeader = soapEnvelope.getHeader();
```

AttachmentPart オブジェクト

AttachmentPart オブジェクトには、SOAP メッセージに対するオプションの添付ファイルが入っています。添付ファイルは、SOAP メッセージの残りの部分とは異なり、XML フォーマットの必須部分ではないので、その形式は、単純なテキストからイメージ ファイルまで、さまざまです。

添付ファイルを操作するには、SOAPMessage クラスの以下のメソッドを使用します。

- countAttachments()：この SOAP メッセージに含まれている添付ファイルの数を返します。
- getAttachments()：すべての添付ファイルを取り出し、(AttachmentPart オブジェクトとして) Iterator オブジェクトに格納します。
- createAttachmentPart()：ほかのタイプの Object から AttachmentPart オブジェクトを作成します。
- addAttachmentPart()：作成された AttachmentPart オブジェクトを SOAPMessage に追加します。

GenericHandler Abstract クラスの拡張

WebLogic Server には、SOAP メッセージ ハンドラ クラスの Java コードの簡素化に役立つ、JAX-RP C ハンドラ API の拡張クラスが組み込まれています。この拡張クラスは、weblogic.webservices.GenericHandler という抽象クラスです。このクラスは、JAX-RPC javax.xml.rpc.handler.Handler インタフェースを実装します。

注意： weblogic.webservices.GenericHandler 抽象クラスは元々、JAX-RPC 仕様がまだ最終ではなく、その仕様にこの機能が含まれていなかったときに WebLogic Server 用に開発されたものです。ただし現在は、WebLogic Server のクラスとほぼ同じ独自の GenericHandler クラスが JAX-RPC に含まれているので、WebLogic 固有のクラスではなく標準の JAX-RPC 抽象クラスを使用することを強くお勧めします。この節の内容は、互換性のみを目的として提供されています。

JAX-RPC javax.xml.rpc.handler.GenericHandler 抽象クラスの詳細については、JAX-RPC の Javadoc を参照してください。

GenericHandler は抽象クラスなので、実装する必要があるのは、実際のコードが格納されるメソッドのみです。特に機能を持たないものも含めた、Handler インタフェースのメソッドをすべて実装する必要はありません。たとえば、ハンドラが初期化パラメータを使用せず、追加のリソースを割り当てる必要がない場合、init() メソッドは実装する必要がありません。

クラスは次のように定義されます。

```
package weblogic.webservice;

import javax.xml.rpc.handler.Handler;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.namespace.QName;

/**
 * @author Copyright (c) 2002 by BEA Systems.All Rights Reserved.
 */

public abstract class GenericHandler
  implements Handler
{

  private HandlerInfo handlerInfo;

  public void init(HandlerInfo handlerInfo) {
    this.handlerInfo = handlerInfo;
  }

  protected HandlerInfo getHandlerInfo() { return handlerInfo; }

  public boolean handleRequest(MessageContext msg) {
    return true;
  }

  public boolean handleResponse(MessageContext msg) {
    return true;
  }

  public boolean handleFault(MessageContext msg) {}

  public void destroy() {}
  public QName[ ] getHeaders() { return handlerInfo.getHeaders(); }

}
```

次のサンプルコードは examples.webservices.handler.nocomponent 製品サンプルからの抜粋ですが、GenericHandler 抽象クラスを使用して自分でハンドラを作成する方法を示しています。この例では、handleRequest() メソッドと handleResponse() メソッドのみを実装しています。init()、destroy()、getHeaders()、および handleFault() の各メソッドは実装していません (したがってコードには含まれていません)。

```
package examples.webservices.handler.nocomponent;

import java.util.Map;
```

```
import javax.xml.rpc.JAXRPCException;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.soap.*;

import weblogic.webservice.GenericHandler;

import weblogic.utils.Debug;

/**
 * @author Copyright (c) 2002 by BEA Systems.All Rights Reserved.
 */

public final class EchoStringHandler
  extends GenericHandler
{
  private int me = System.identityHashCode(this);

  public boolean handleRequest(MessageContext messageContext) {
    System.err.println("** handleRequest called in:"+me);
    return true;
  }

  public boolean handleResponse(MessageContext messageContext) {

    try {
      MessageFactory messageFactory = MessageFactory.newInstance();

      SOAPMessage m = messageFactory.createMessage();

      SOAPEnvelope env = m.getSOAPPart().getEnvelope();

      SOAPBody body = env.getBody();

      SOAPElement fResponse =
        body.addBodyElement(env.createName("echoResponse"));

      fResponse.addAttribute(env.createName("encodingStyle"),
        "http://schemas.xmlsoap.org/soap/encoding/");

      SOAPElement result =
        fResponse.addChildElement(env.createName("result"));

      result.addTextNode("Hello World");

      ((SOAPMessageContext)messageContext).setMessage(m);

      return true;

    } catch (SOAPException e) {
    e.printStackTrace();
      throw new JAXRPCException(e);
```

```
    }
  }
}
```

SOAP メッセージ ハンドラ情報による web-services.xml ファイルの更新

web-services.xml デプロイメント記述子ファイルは、Web サービス用に定義されている SOAP メッセージ ハンドラおよびハンドラ チェーンを、実行順に記述します。

ハンドラ情報で web-services.xml ファイルを更新するには、次の手順に従います。

1. この Web サービス用に定義されたすべてのハンドラ チェーンのリストが格納される <web-services> ルート要素の <handler-chains> 子要素を作成します。
2. <handler-chains> 要素の <handler-chain> 子要素を作成します。この要素内に、そのハンドラ チェーンにあるすべてのハンドラがリストされます。各ハンドラに対して、class-name 属性を使用してそのハンドラを実装する Java クラスの完全修飾名を指定します。<init-params> 要素を使用して、ハンドラの任意の初期化パラメータを指定します。

   次のサンプル コードからの抜粋は、3 つのハンドラ (最初のハンドラに初期化パラメータがある) の入った、myChain というハンドラ チェーンを示しています。

   ```
   <web-services>
      <handler-chains>
         <handler-chain name="myChain">
           <handler class-name="myHandlers.handlerOne" >
               <init-params>
                  <init-param name="debug" value="on" />
               </init-params>
           </handler>
           <handler class-name="myHandlers.handlerTwo" />
           <handler class-name="myHandlers.handlerThree" />
         </handler-chain>
      </handler-chains>
   ```

```
...
</web-services>
```

3. <operations> 要素 (この要素自体も <web-service> 要素の子要素) の <operation> 子要素を使用して、このハンドラ チェーンが、この Web サービスのオペレーションとなるように指定します。次の 2 つのシナリオのいずれかに従います。

   - ハンドラ チェーンは、ステートレス セッション EJB などのバックエンド コンポーネントとともに実行される。

     この場合は、次の web-services.xml ファイルからの抜粋のように、<operation> 要素の component、method、handler-chain の各属性を使用します。

     ```
     <web-service>
       <components>
         <stateless-ejb name="myEJB">
           ...
         </stateless-ejb>
       </components>
       <operations>
         <operation name="getQuote"
                    method="getQuote"
                    component="myEJB"
                    handler-chain="myChain" />
        </operations>
     </web-service>
     ```

     この例では、myChain ハンドラ チェーンの要求チェーンが最初に実行され、次に、myEJB ステートレスセッション EJB の getQuote() メソッドが実行され、最後に myChain の応答チェーンが実行されます。

   - ハンドラ チェーンは、バックエンド コンポーネントなしに単独で実行される。

     この場合、次の抜粋に示すように、<operation> 要素のハンドラチェーン属性のみを使用し、コンポーネントやメソッドの属性を明示的に指定することはしません。

     ```
     <web-service>
       <operations>
         <operation name="chainService"
                    handler-chain="myChain" />
        </operations>
     </web-service>
     ```

この例の Web サービスは、myChain ハンドラ チェーンのみを扱っています。

クライアント アプリケーションでの SOAP メッセージ ハンドラおよびハンドラ チェーンの使用

ここまで、WebLogic Server で動作する Web サービスの一環として実行される、ハンドラ チェーン内の SOAP メッセージ ハンドラを作成する方法について説明しました。一方、クライアント アプリケーション内で実行するハンドラを作成することも可能です。クライアント サイドのハンドラの場合、クライアント アプリケーションが Web サービスを呼び出す際に、以下のタイミングで 2 回実行されます。

- クライアント アプリケーションが Web サービスに SOAP 要求を送信する直前
- クライアント アプリケーションが Web サービスから SOAP 応答を受信した直後

クライアント サイド ハンドラを作成するには、サーバ サイド ハンドラの場合と同じように、javax.rpc.xml.handler.Handler インタフェースを実装する Java クラスを記述します。多くの場合は、WebLogic Server で動作する Web サービスと Web サービスを呼び出すクライアント アプリケーションの両方で、まったく同じハンドラ クラスを使用できます。たとえば、サーバおよびクライアント双方との SOAP メッセージの送受信をログに記録する汎用のロギング ハンドラ クラスを記述できます。ハンドラ Java クラスの記述の詳細については、10-6 ページの「ハンドラインタフェースの実装」を参照してください。

クライアント サイドのハンドラ クラスを作成したら、クライアント アプリケーションにハンドラを登録しますが、この手順はサーバ サイドの場合と異なります。クライアント アプリケーションにはデプロイメント記述子がないため、javax.xml.rpc.handler.HandlerInfo クラスおよび javax.xml.rpc.handler.HandlerRegistry クラスを使用して、プログラムで

ハンドラを登録する必要があります。以下のクライアント アプリケーションのサンプルで登録方法の例を示します。太字で示した部分については後ほど説明します。

```
import java.util.ArrayList;

import java.io.IOException;

import javax.xml.namespace.QName;

import javax.xml.rpc.ServiceException;

import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.HandlerRegistry;

public class Main{

  public static void main( String[] args ){

    if( args.length == 1 ){
      new Main( args[0] );
    } else {
      throw new IllegalArgumentException( "URL of the service not specified" );
    }
  }

  public Main( String wsdlUrl ){
    try{

      HelloWorldService service = new HelloWorldService_Impl( wsdlUrl );
      HelloWorldServicePort port = service.getHelloWorldServicePort();

      QName portName = new QName( "http://tutorial/sample4/",
        "HelloWorldServicePort");

      HandlerRegistry registry = service.getHandlerRegistry();

      List handlerList = new ArrayList();
      handlerList.add( new HandlerInfo( ClientHandler.class, null, null ) );

      registry.setHandlerChain( portName, handlerList );

      System.out.println( port.helloWorld() );
    }catch( IOException e ){
      System.out.println( "Failed to create web service client:" + e );
    }catch( ServiceException e ){
      System.out.println( "Failed to create web service client:" + e );
    }
  }
}
```

上記の例で注目すべき主な点は以下のとおりです。

- クライアント サイド ハンドラ クラスの登録に使用する JAX-RPC `HandlerInfo` クラスおよび `HandlerRegistry` クラスをインポートする。

  ```
  import javax.xml.rpc.handler.HandlerInfo;
  import javax.xml.rpc.handler.HandlerRegistry;
  ```

- Web サービス ポートの修飾名を格納する `QName` オブジェクトを作成する。

  ```
  QName portName = new QName( "http://tutorial/sample4/",
         "HelloWorldServicePort");
  ```

  ポートの名前およびそのネームスペース用に呼び出す Web サービスの WSDL を参照します。

- `HandlerRegistry` オブジェクトを作成する。

  ```
  HandlerRegistry registry = service.getHandlerRegistry();
  ```

- 登録するハンドラのリストを格納する `List` オブジェクトを作成する。このリストが、クライアント サイドのハンドラ チェーンとなります。Java ハンドラ クラスの名前を指定するには、`HandlerInfo` クラスを使用します。

  ```
  List handlerList = new ArrayList();
  handlerList.add( new HandlerInfo( ClientHandler.class, null,
  null ) );
  ```

  この例のハンドラ チェーンには `ClientHandler.class` という 1 つのハンドラしか含まれていませんが、ハンドラ チェーンに含めるハンドラの数に制限はありません。

  警告： `List` オブジェクトにハンドラを追加する順番によって、クライアント アプリケーションでのハンドラの実行順序が決まります。たとえば、最初に `HandlerA.class` を実行してから `HandlerB.class` を実行する場合は、`HandlerA.class` を `HandlerB.class` より前にリストに追加する必要があります。

- `HandlerRegistry.setHandlerChain()` メソッドを使用して、ハンドラ チェーンをクライアント アプリケーションに登録する。

  ```
  registry.setHandlerChain( portName, handlerList );
  ```

11 セキュリティのコンフィグレーション

この章では、WebLogic Web サービスのセキュリティをコンフィグレーションする方法について説明します。

セキュリティのコンフィグレーションの概要

WebLogic Web サービスに対するセキュリティのコンフィグレーションは、基本的には、WebLogic Server で実行される他のタイプのアプリケーションやコンポーネントに対するセキュリティのコンフィグレーションと同じです。Web サービス全体に対するセキュリティは、Web サービスおよび WSDL を呼び出す URL へのアクセスを制限することにより設定できます。Web サービス全体でセキュリティを設定すると、その Web サービスを構成するコンポーネントも自動的にセキュリティが設定されます。また、ステートレス セッション EJB、一部のメソッド、`web-services.xml` ファイルが入った Web アプリケーションなど、Web サービスの個々のコンポーネントに対してセキュリティを設定することもできます。

Web サービスまたはそのコンポーネントへのアクセスが保護されたら、その Web サービスを呼び出すときに HTTP または SSL を使用してそれ自身を認証するようにクライアント アプリケーションをコンフィグレーションします。

WebLogic Web サービスのセキュリティのさらなる例については、「Webservice Download Page」を参照してください。

セキュリティのコンフィグレーション：主な手順

WebLogic Web サービスおよび、サービスを呼び出すクライアントのセキュリティをコンフィグレーションするには、次の手順に従います。この手順の詳しい説明は後の節で行います。

1. ロールを作成し、それらを自分のレルムのプリンシパルにマップして、次に、ロール内のプリンシパルのみがアクセスできる、安全なコンポーネントがどれであるかを指定することにより、Web サービスまたはその一部のコンポーネントへのアクセスをコントロールします。

   11-3 ページの「WebLogic Web サービスへのアクセスのコントロール」を参照してください。

2. 必要であれば、`web-services.xml` ファイルを更新して、Web サービスが HTTPS のみによりアクセスできるように指定します。

   11-6 ページの「HTTPS プロトコルの指定」を参照してください。

3. クライアント アプリケーションが SSL を使用して認証される場合は、WebLogic Server に対して SSL をコンフィグレーションします。

   WebLogic Server がクライアント アプリケーションに対して証明の提示を要求される場合には一方向 SSL、またクライアント アプリケーションと WebLogic Server の双方が互いに証明書を提示する場合には、双方向 SSL をコンフィグレーションします。

   SSL の詳細、一方向と双方向の違い、および両者をコンフィグレーションするための手順については、「SSL のコンフィグレーション」を参照してください。

警告： WebLogic Web サービスを呼び出す際の接続のセキュリティを双方向 SSL を使用して設定する場合、WebLogic Server は常に証明書の ID をアサートすることで、有効なユーザにマップされるようにします。これは、Web サービスまたはステートレス EJB のバックエンド コンポーネントが特別な権限を必要としない場合も同様です。一方向 SSL の場合は、クライアント アプリケーションが証明書を送信しないため上記は適用されません。

4. WebLogic Web サービスの呼び出し時に、HTTP または SSL を使用して認証されるように、クライアントをコーディングします。

   11-7 ページの「セキュリティが設定された Web サービスを呼び出すためのクライアント アプリケーションのコーディング」を参照してください。

5. クライアント アプリケーションが SSL を使用する場合は、クライアント サイドで SSL をコンフィグレーションします。

   11-8 ページの「クライアント アプリケーションに対する SSL のコンフィグレーション」を参照してください。

WebLogic Web サービスへのアクセスのコントロール

前述のように、WebLogic Web サービスは、標準 J2EE エンタープライズ アプリケーションとしてパッケージ化されます。したがって、Web サービスへのアクセスにセキュリティを設定するには、Web サービスを構成する以下の標準 J2EE コンポーネントのいくつかまたはすべてに対するアクセスにセキュリティを設定する必要があります。

- Web サービスの URL
- Web サービスを実装するステートレス セッション EJB
- ステートレス セッション EJB のメソッドの一部
- WSDL および Web サービスのホーム ページ

ベーシック HTTP 認証または SSL を使用して、WebLogic Web サービスにアクセスするクライアントを認証できます。先のコンポーネントが標準の J2EE コンポーネントであるため、標準の J2EE セキュリティ プロシージャを使用してセキュリティを設定します。

注意： Web サービスを実装するバックエンド コンポーネントが Java クラスまたは JMS リスナである場合は、次の節で説明するとおり、その Web サービスに対しては、Web サービスを呼び出す URL へのセキュリティ制約の追加によるほかは、セキュリティを設定できません。

WebLogic のセキュリティのコンフィグレーション、プログラミング、および管理の詳細については、セキュリティに関するドキュメントを参照してください。

Web サービスの URL に対してセキュリティを設定する

8-23 ページの「WebLogic Web サービスのホーム ページおよび WSDL の URL」で説明するとおり、クライアント アプリケーションは、URL を使用して Web サービスにアクセスします。以下は、このような URL の例です。

```
http://ariel:7001/web_services/TraderService
```

URL へのアクセスを制限することにより、Web サービス全体へのアクセスを制限できます。そのためには、`web.xml` および `weblogic.xml` の両デプロイメント記述子ファイル (`web-services.xml` ファイルが格納されている Web アプリケーションにあります) をセキュリティ情報で更新します。

URL のアクセス制限の詳細については、「WebLogic リソースの保護」を参照してください。

ステートレス セッション EJB とそのメソッドに対してセキュリティを設定する

Web サービスを実装するステートレス セッション EJB に対してセキュリティを設定する場合は、そのサービスを呼び出すクライアント アプリケーションは、Web アプリケーション、WSDL および Web サービスのホーム ページにはアクセ

スできますが、オペレーションを実装するメソッドそのものを呼び出すことができない場合があります。このタイプのセキュリティは、EJB のビジネス ロジックに誰がアクセスしたかを細かくモニタするが、Web サービス全体へのアクセスはブロックしないという場合に便利です。

また、このタイプのセキュリティは、メソッドレベルで、Web サービスのさまざまなオペレーションにアクセスする場合にも使用できます。たとえば、情報を表示するメソッドは任意のユーザによって呼び出せるが、一部のユーザのみが情報を更新できるように指定することができます。

Adminstration Console を使用して EJB および EJB の個々のメソッドにセキュリティを設定する方法については、『WebLogic リソースのセキュリティ』を参照してください。

WSDL および Web サービスのホーム ページに対してセキュリティを設定する

WSDL または WebLogic Web サービスのホーム ページへのアクセスを制限するには、サービスを記述する `web-services.xml` デプロイメント記述子を更新します。以下の手順に従ってください。

1. 任意のエディタで `web-services.xml` ファイルを開きます。

   `web-services.xml` ファイルは、Web サービス EAR ファイルの Web アプリケーションの `WEB-INF` ディレクトリにあります。ファイルの格納場所の詳細については、6-13 ページの「Web サービス EAR ファイル パッケージ」を参照してください。

2. WSDL へのアクセスを制限する場合は、Web サービスを記述する `<web-service>` 要素に `exposeWSDL="False"` 属性を追加します。ホーム ページへのアクセスを制限するには、`exposeHomepage="False"` 属性を追加します。以下に、コード例の一部を抜粋します。

```
<web-service
   name="stockquotes"
          uri="/myStockQuoteService"
   exposeWSDL="False"
   exposeHomePage="False" >
    ...
</web-service>
```

3. 変更を有効にするために、Web サービスを再デプロイします。すべてのユーザが、WSDL および Web サービスのホーム ページにアクセスできなくなります。

HTTPS プロトコルの指定

Web サービスを HTTPS によってのみアクセスできるようにするには、次の例に示すように、その Web サービスを記述する `web-services.xml` ファイルにある `<web-service>` 要素の `protocol` 属性を更新します。

```
<web-services>
  <web-service name="stockquotes"
               targetNamespace="http://example.com"
               uri="/myStockQuoteService"
               protocol="https" >
     ...
  </web-service>
</web-services>
```

注意： WebLogic Server に対して SSL のコンフィグレーションは行うが、`web-services.xml` ファイルで HTTPS プロトコルを指定しない場合は、クライアント アプリケーションから HTTP および HTTPS を使用して Web サービスにアクセスすることができます。ただし、`web-services.xml` ファイルで HTTPS アクセスを指定した場合、クライアント アプリケーションは HTTP では Web サービスにアクセスできません。

`servicegen` Ant タスクを使用して Web サービスをアセンブルする場合は、次の `build.xml` ファイルのサンプルに示すように、`<service>` 要素の `protocol` 属性を使用して HTTPS プロトコルを指定してください。

```
<project name="buildWebservice" default="ear">
  <target name="ear">
    <servicegen
       destEar="ws_basic_statelessSession.ear"
       contextURI="WebServices"
       <service
         ejbJar="HelloWorldEJB.jar"
         targetNamespace="http://www.bea.com/webservices/basic/statelesSession"
         serviceName="HelloWorldEJB"
         serviceURI="/HelloWorldEJB"
         protocol="https"
```

```
        generateTypes="True"
        expandMethods="True">
      </service>
    </servicegen>
  </target>
</project>
```

セキュリティが設定された Web サービスを呼び出すためのクライアント アプリケーションのコーディング

Web サービスを呼び出す JAX-RPC クライアント アプリケーションを記述する際、クライアントの認証ができるように、サービスにユーザ名とパスワードを送信するための次の 2 つのプロパティを使用します。

- `javax.xml.rpc.security.auth.username`
- `javax.xml.rpc.security.auth.password`

次の例は JAX-RPC 仕様からの抜粋ですが、`javax.xml.rpc.Stub` インタフェースを使用して Web サービスを呼び出す場合の、これらのプロパティの使用法を示しています。

```
StockQuoteProviderStub sqp = // ... スタブを取得 ;
sqp._setProperty ("javax.xml.rpc.security.auth.username", "juliet");
sqp._setProperty ("javax.xml.rpc.security.auth.password", "mypassword");
float quote sqp.getLastTradePrice("BEAS");
```

WebLogic で生成したクライアント JAR ファイルを使用して Web サービスを呼び出す場合は、Stub クラスがすでに作成されているため、JAX-RPC 仕様から抜粋した次の例に示すように、`getServicePort()` メソッドのサービス固有の実装にユーザ名とパスワードを渡すことができます。

```
StockQuoteService sqs = // ... サービスへのアクセスを取得 ;
StockQuoteProvider sqp = sqs.getStockQuoteProviderPort ("juliet", "mypassword");
float quote = sqp.getLastTradePrice ("BEAS");
```

この例では、`getStockQuoteProvidePort()` メソッドの実装が、上記 2 つの認証プロパティを設定します。

JAX-RPC を使用してセキュリティが設定された Web サービスを呼び出すクライアント アプリケーションの記述に関する詳細は、http://java.sun.com/xml/jaxrpc/index.html を参照してください。

クライアント アプリケーションに対する SSL のコンフィグレーション

クライアント アプリケーションの SSL は、以下のいずれかを使用してコンフィグレーションします。

- WebLogic Server が提供する SSL 実装
- サードパーティの SSL 実装

双方向 SSL を使用している場合、クライアント アプリケーションも、WebLogic Server に対して証明書を提示する必要があります。詳細については、11-14 ページの「クライアント アプリケーションに対する双方向 SSL のコンフィグレーション」を参照してください。

この節で説明した API の詳細は、Web サービスのセキュリティに関する Javadocs を参照してください。

WebLogic Server が提供する SSL 実装を使用する

WebLogic Server は、`webserviceclient+ssl.jar` クライアント実行時 JAR ファイルで SSL の実装を提供しています。このクライアント JAR ファイルには、SSL 実装のほか、`webservicesclient.jar` にある、標準のクライアント JAX-PRC 実行時クラスも入っています。

注意： クライアント機能に関する現在の BEA のライセンス供与については、Web サイト BEA eLicense を参照してください。

クライアント アプリケーションに対して基本的な SSL サポート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. *WL_HOME*\server\lib\webserviceclient+ssl.jar ファイルを、クライアント アプリケーション開発用のコンピュータにコピーします。*WL_HOME* は、WebLogic プラットフォームの最上位ディレクトリです。このクライアント JAR ファイルには、SSL 実装および JAX-RPC のクライアント実行時実装が入っています。
2. クライアント JAR ファイルをクライアント アプリケーションの CLASSPATH 変数に追加します。
3. 信頼性のある CA (Certificate Authority : 認証局) 証明書のファイル名を設定します。その方法は次のいずれかです。
   - trustedfile System プロパティを、PEM エンコード済み証明書のコレクションを含むファイルの名前に設定する。
   - クライアント アプリケーションで BaseWLSSLAdapter.setTrustedCertificatesFile(String *ca_filename*) メソッドを実行する。
4. クライアントアプリケーションを実行する際、コマンド ラインで次の System プロパティを設定します。
   - bea.home=*license_file_directory*
   - java.protocol.handler.pkgs=com.certicom.net.ssl

   次の例に示すように、*licence_file_directory* は、BEA ライセンス ファイル license.bea が入っているディレクトリです。

```
java -Dbea.home=c:\bea_home \
   -Djava.protocol.handler.pkgs=com.certicom.net.ssl  my_app
```

   注意 : クライアント アプリケーションが、WebLogic Server のホストであるコンピュータとは異なるコンピュータ上で実行されている場合 (これが一般的) は、BEA ライセンス ファイルをサーバ コンピュータからクライアント コンピュータ上のディレクトリにコピーし、次に bea.home システム プロパティをこのクライアントサイドのディレクトリに指定します。
5. 厳密な証明書検証を無効にするには、アプリケーション実行時にコマンド ラインで weblogic.webservice.client.ssl.strictcertchecking System プロパティを false に設定するか、プログラムで BaseWLSSLAdapter.setStrictCheckingDefault() メソッドを使用します。

詳細は、Web サービスのセキュリティに関する Javadocs を参照してください。

SSL をプログラムでコンフィグレーションする

WebLogic Server が提供する SSL 実装を、weblogic.webservice.client.WLSSLAdapter アダプタ クラスを使用してプログラムでコンフィグレーションすることもできます。このアダプタ クラスには、WebLogic Server の SSL 実装固有のコンフィグレーション情報が保持されていて、コンフィグレーションの参照と変更を可能にします。

次の抜粋は、ある WebLogic Web サービスに対する WLSSLAdapter クラスのコンフィグレーションの例です。太字の行については、例の後で説明します。

```
//アダプタのインスタンス化
WLSSLAdapter adapter  = new WLSSLAdapter();
adapter.setTrustedCertifcatesFile("mytrustedcerts.pem");

//任意に、アダプタ ファクトリが
//常にこのインスタンスを使用するように設定
SSLAdapterFactory.getDefaultFactory().setDefaultAdapter(adapter);
SSLAdapterFactory.getDefaultFactory().setUseDefaultAdapter(true);

//サービス ファクトリを作成
ServiceFactory factory = ServiceFactory.newInstance();

//サービスを作成
Service service = factory.createService( serviceName );

//呼び出しを作成
Call call = service.createCall();

call.setProperty("weblogic.webservice.client.ssladapter",
                 adapter);

try {

  //リモートの Web サービスを呼び出し
  String result = (String) call.invoke( new Object[ ]{ "BEAS" } );
  System.out.println( "Result: " +result);
} catch (JAXRPCException jre) {
...
}
```

この例では、まず、WebLogic Server が提供する WLSSLAdapter クラスをインスタンス化する方法を示しています。webserviceclient+ssl.jar ファイルにある SSL 実装をサポートするクラスです。次に、

setTrustedCertificatesFile(String) メソッドを使用して認証局証明書が入ったファイルの名前を選択して、アダプタ インスタンスをコンフィグレーションしています。この例では、ファイルの名前は mytrustedcerts.pem です。

次に、アダプタ ファクトリのデフォルトのアダプタとして WLSSLAdapter を設定する方法を示し、ファクトリが必ずこのデフォルト値を返すようにコンフィグレーションしています。

注意： これは任意の手順で、すべての Web サービスとそれぞれに対応するコンフィグレーションが、同じアダプタ クラスを共有できるようにします。

また、Web サービスの特定のポートや呼び出しに対してこのアダプタを設定することもできます。先の例では、このアダプタを設定し、Call クラスを使用して Web サービスを動的に呼び出す方法を示しています。

```
call.setProperty("weblogic.webservice.client.ssladapter", adapter);
```

weblogic.webservice.client.SSLAdapter インタフェース (この例では、WebLogic Server が提供する WLSSLAdapter クラス) を実装するオブジェクトにこのプロパティを設定してください。

次の例では、アダプタを設定し、Stub インタフェースを使用して Web サービスを静的に呼び出す方法を示しています。

```
((javax.xml.rpc.Stub)stubClass)._setProperty("weblogic.webservice.client.sslada
pter", adapterInstance);
```

次のメソッドを使用することにより、Web サービスの呼び出しまたはポートの特定のインスタンスに対するアダプタを取得して、Web サービスを動的に呼び出せます。

```
call.getProperty("weblogic.webservice.client.ssladapter");
```

Web サービスを静的に呼び出すには、次のメソッドを使用します。

```
((javax.xml.rpc.Stub)stubClass)._getProperty("weblogic.webservice.client.sslada
pter");
```

詳細は、Web サービスのセキュリティに関する Javadocs を参照してください。

サードパーティの SSL 実装を使用する

サードパーティの SSL 実装を使用する場合は、まず独自のアダプタ クラスを実装する必要があります。次の例は、JSSE のサポートを提供する簡単なクラスを示しています。自分でクラスを実装する主な手順は、例の後で説明します。

```
import java.net.URL;
import java.net.Socket;
import java.net.URLConnection;
import java.io.IOException;

public class JSSEAdapter implements weblogic.webservice.client.SSLAdapter {

  javax.net.SocketFactory factory =
    javax.net.ssl.SSLSocketFactory.getDefault();

  //weblogic.webservice.client.SSLAdapter インタフェースを実装

  public Socket createSocket(String host, int port) throws IOException  {
    return factory.createSocket(host, port);
  }

  public URLConnection openConnection(URL url) throws IOException {
    //java.protocol.handler.pkgs が正しく設定されていることを前提とする
    return url.openConnection();
  }

  // コンフィグレーション インタフェース

  public void setSocketFactory(javax.net.ssl.SSLSocketFactory factory) {
    this.factory = factory;
  }

  public javax.net.ssl.SSLSocketFactory getSocketFactory() {
    return (javax.net.ssl.SSLSocketFactory) factory;
  }
}
```

自分でアダプタ クラスを作成するには、次の手順に従います。

1. 次のインタフェースを実装するクラスを作成します。

   ```
   weblogic.webservice.client.SSLAdapter
   ```

2. 次のシグネチャを持つ createSocket メソッドを実装します。

   ```
   public Socket createSocket(String host, int port)
       throws IOException
   ```

このメソッドは、java.net.Socket を拡張するオブジェクトを返します。このオブジェクトは、Web サービスが呼び出されると、指定されたホスト名とポートに接続されます。

3. 次のシグネチャを持つ openConnection メソッドを実装します。

   ```
   public URLConnection openConnection(URL url) throws IOException
   ```

   このメソッドは、java.net.URLConnection を拡張するオブジェクトを返します。このオブジェクトは、指定された URL に接続されるようにコンフィグレーションされます。これらの接続は、Web サービスの WSDL のダウンロードなど、使用頻度の低いネットワーク オペレーションに使用されます。

4. クライアント アプリケーションの実行時に、次の System プロパティを、作成したアダプタ クラスの完全修飾名に設定します。

   ```
   weblogic.webservice.client.ssl.adapterclass
   ```

   デフォルトの SSLAdapterFactory クラスは、作成したアダプタ クラスをロードし、デフォルトの引数を持たないコンストラクタを使用してそのクラスのインスタンスを作成します。

5. 11-10 ページの「SSL をプログラムでコンフィグレーションする」で示したようにカスタム アダプタをコンフィグレーションし、WLSSLAdapter を自分のクラスに置き換えて、アダプタに定義されたコンフィグレーション方法を使用します。

詳細は、Web サービスのセキュリティに関する Javadocs を参照してください。

SSLAdapterFactory クラスを拡張する

SSLAdapterFactory クラスを拡張することによって、カスタムの SSL アダプタのファクトリ クラス (アダプタのインスタンスを作成するために使用) を作成することができます。ファクトリ クラスを拡張する理由の 1 つは、それにより、各アダプタの使用に先立ち、作成時に、カスタムのコンフィグレーションを実行できるようになることです。

カスタムの SSL アダプタのファクトリ クラスを作成するには、次の手順に従います。

1. 次のクラスを拡張するクラスを作成します。

   ```
   weblogic.webservice.client.SSLAdapterFactory
   ```

2. SSLAdapterFactory クラスの次のメソッドを書き換えます。

```
public weblogic.webservice.client.SSLAdapter createSSLAdapter();
```

アダプタ ファクトリによって、SSLAdapter またはこのインタフェースを実装するアダプタが新たに作成されたときは、必ず、このメソッドが呼び出されます。このメソッドを書き換えることによって、新しいアダプタを実際に使用し始める前にカスタムのコンフィグレーションを実行できます。

3. クライアント アプリケーションで、ファクトリのインスタンスを作成し、次のメソッドを実行して、そのファクトリをデフォルト ファクトリとして設定します。

```
SSLAdapterFactory.setDefaultFactory(factoryInstance);
```

詳細は、Web サービスのセキュリティに関する Javadocs を参照してください。

クライアント アプリケーションに対する双方向 SSL のコンフィグレーション

WebLogic Server に対して双方向 SSL をコンフィグレーションした場合は、一方向 SSL で要求されるようにクライアント アプリケーションに対して WebLogic Server が証明書を提示するのに加えて、クライアント アプリケーションが WebLogic Server に証明書を提示することも必要です。以下のサンプル Java コードでは、それを行うための方法の 1 つを示します。ここでは、クライアント アプリケーションがクライアント証明書ファイルを引数 (太字で示した該当コード) として受け取っています。

```
...

SSLAdapterFactory factory = SSLAdapterFactory.getDefaultFactory();
WLSSLAdapter adapter = (WLSSLAdapter) factory.getSSLAdapter();

if (argv.length > 1 ) {
  System.out.println("loading client certs from "+argv[1]);

FileInputStream clientCredentialFile  = new FileInputStream (argv[1]);
String pwd = "clientkey";

adapter.loadLocalIdentity(clientCredentialFile, pwd.toCharArray());
```

```
javax.security.cert.X509Certificate[] certChain =
adapter.getIdentity("RSA",0);
factory.setDefaultAdapter(adapter);
factory.setUseDefaultAdapter(true);
...
```

WebLogic Web サービスでの双方向 SSL の完全な使用例については、「Two-Way SSL Example」を参照してください。

プロキシ サーバを使用する

たとえば、クライアント アプリケーションがファイアウォールの内側で動作している状況で、プロキシ サーバを使用する必要がある場合は、以下の 2 つの System プロパティを使用してプロキシ サーバのホスト名とポートを設定します。

- **weblogic.webservice.transport.https.proxy.host**
- **weblogic.webservice.transport.https.proxy.port**

これらの System プロパティの詳細については、8-27 ページの「WebLogic Web サービスのシステム プロパティ」を参照してください。

12 JMS で実装された WebLogic Web サービスの作成

この章では、JMS で実装された WebLogic Web サービスを作成する方法について説明します。

JMS で実装された WebLogic Web サービスの作成方法の概要

Web サービスのオペレーションは、ステートレス セッション EJB または Java クラスを使用して実装するほか、メッセージ駆動型 Bean などの JMS メッセージのコンシューマまたはプロデューサを使用して実装することもできます。

JMS で実装されたオペレーションには、次に示す 3 つのタイプがあります。

- JMS の送り先にデータを送信するオペレーション

  このタイプのオペレーションは、JMS メッセージ コンシューマを使用して実装します。このメッセージ コンシューマは、Web サービス オペレーショ

ンを呼び出すクライアントが JMS 送り先にデータを送信した後にメッセージを消費します。

- JMS キューからデータを受信するオペレーション

  このタイプのオペレーションは、JMS メッセージ プロデューサを使用して実装します。このメッセージ プロデューサはメッセージを特定の JMS キューに入れ、このメッセージスタイルの Web サービス コンポーネントを呼び出すクライアントがポーリングを行ってメッセージを受信します。

- JMS トピックからデータを受信するオペレーション

  このタイプのオペレーションは、JMS メッセージ プロデューサを使用して実装します。このメッセージ プロデューサはメッセージを特定の JMS トピックにパブリッシュし、このメッセージスタイルの Web サービス コンポーネントを呼び出すクライアントがポーリングを行ってメッセージを受信します。

  注意： JMS トピックからのデータの受信は、このバージョンの WebLogic Server では非推奨となっています。つまり、現在この機能は動作しますが、将来のバージョンの WebLogic Server ではサポートされない可能性があります。

  現在この機能を使用している場合は、アプリケーションを更新して、ステートレス セッション EJB を JMS トピックとしてではなく Web サービスのバックエンド実装として使用するようにします。JMS API を使用した JMS トピックのリスンが、JMS を実装する Web サービスをクライアント アプリケーションがリスンしているのと同じ方法で実行できるように EJB をプログラムします。次に、`servicegen` Ant タスクを使用して Web サービスをアセンブルしなおします。最後に、クライアント アプリケーションを記述しなおして、JMS トピックからのメッセージに対して EJB を実装する新しい Web サービスがポーリングされるようにします。

クライアント アプリケーションが JMS で実装された Web サービス オペレーションにデータを送信する際、WebLogic Server は、まず、そのデータが組み込みデータ型かどうかによって、組み込みまたはカスタムのシリアライザを使用して XML データをその Java 表現に変換します。WebLogic Server は、変換の結果として取得した Java オブジェクトを `javax.jms.ObjectMessage` オブジェクトにラップし、JMS の送り先に配置します。次に、メッセージ駆動型 Bean などの

JMS リスナを記述して、ObjectMessage を受け取って処理することができます。クライアント アプリケーションが Web サービスを呼び出して JMS キューからデータを受信するときは、これと逆の順序で同様の動作が行われます。

非組み込みデータ型を使用している場合は、web-services.xml デプロイメント記述子ファイルを、正しいデータ型マッピング情報で更新する必要があります。Web サービスが当該データのデータ型マッピング情報を見つけられない場合は、データは、JAXM (Java API for XML Messaging) で定義されている javax.xml.soap.SOAPElement データ型に変換されます。

注意： JMS コンシューマまたはプロデューサで実装された Web サービス オペレーションへの in パラメータと戻り値は、java.io.Serializable インタフェースを実装する必要があります。

メッセージ駆動型 Bean のプログラミングの詳細については、『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』を参照してください。

JMS で実装された WebLogic Web サービスの設計

この節では、JMS と JMS コンシューマまたはプロデューサを使用して実装された WebLogic Web サービスのオペレーションとの関係、およびこれらのタイプの Web サービスを開発する際の設計上の考慮事項について説明します。

キューまたはトピックの選択

JMS キューでは、1 つの宛先に対してメッセージが配信されるポイント ツー ポイント メッセージング モデルを実装します。JMS トピックでは、複数の宛先に対してメッセージが配信されるパブリッシュ / サブスクライブ メッセージ モデルを実装します。

JMS コンシューマまたはプロデューサをバックエンド コンポーネントとして Web サービス オペレーションを実装する前に、以下の事項を決める必要があります。

- JMS キューまたはトピックのどちらを使用するか。
- Web サービスを呼び出すクライアント アプリケーションが、サービスにメッセージを送信するか、またはサービスからメッセージを受信するか。同じオペレーションで送信と受信の両方をサポートすることはできません。

メッセージの取得と処理

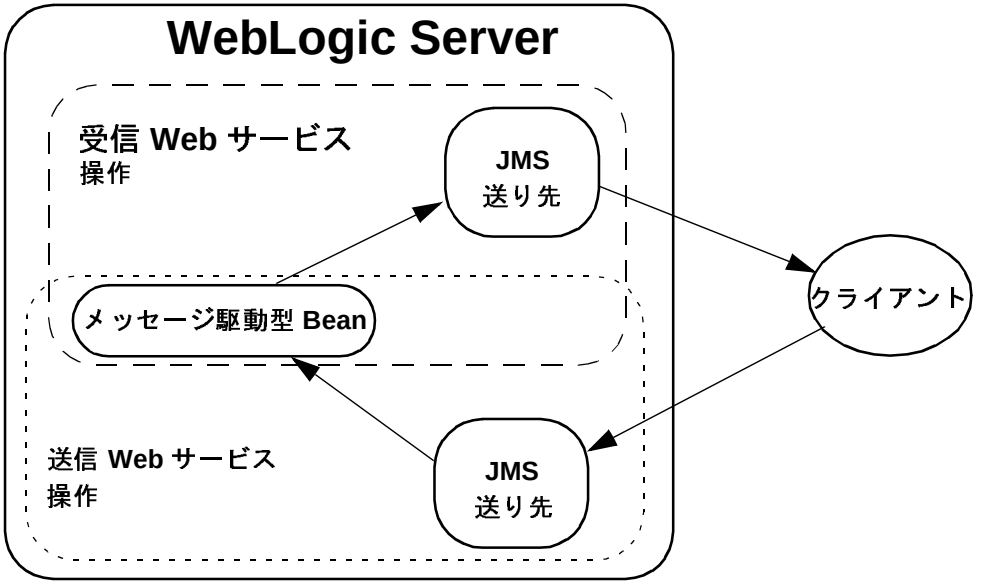
使用する JMS の送り先のタイプを決定した後で、JMS の送り先からメッセージを取得して処理する J2EE コンポーネントのタイプを決定する必要があります。一般的に、このタイプはメッセージ駆動型 Bean です。このメッセージ駆動型 Bean では、メッセージ処理のすべての作業を実行するか、部分的またはすべての作業を他の EJB に分配します。メッセージ駆動型 Bean がメッセージの処理を完了したら、Web サービスの実行は完了です。

1 つの Web サービス オペレーションで送信と受信の両方をサポートすることはできないため、クライアント アプリケーションでデータの送信と受信の両方ができるようにするには、Web サービス オペレーションを 2 つ作成する必要があります。送信機能を持つ Web サービス オペレーションは、受信機能を持つオペレーションに関連しています。これは、メッセージを処理したオリジナルのメッセージ駆動型 Bean が、受信機能を持つ Web サービス オペレーションに対応する JMS の送り先に応答を配置するためです。

JMS コンポーネントの使用例

図 12-1 は、クライアントからメッセージを受信するものと、クライアントへメッセージを送信するものの 2 つの Web サービス オペレーションを示しています。2 つの Web サービス オペレーションには、それぞれ独自の JMS の送り先があります。メッセージ駆動型 Bean は、最初の JMS の送り先からメッセージを取得して情報を処理し、メッセージを 2 番目の JMS の送り先に戻します。クライアントは、最初の Web サービス オペレーションを呼び出してメッセージを WebLogic Server に送信し、次に 2 番目の Web サービス オペレーションを呼び出して WebLogic Server からメッセージを受信します。

図 12-1 JMS 実装の Web サービス オペレーションと JMS 間のデータ フロー



JMS で実装された WebLogic Web サービスの実装

JMS メッセージのプロデューサまたはコンシューマを使用して Web サービスを実装するには、次の手順に従います。

1. JMS の送り先からのメッセージを消費する、または JMS の送り先にメッセージを生成する J2EE コンポーネント (一般的にはメッセージ駆動型 Bean) のコードを記述します。

   詳細については、『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』を参照してください。

2. Administration Console を使用して、以下の WebLogic Server の JMS コンポーネントをコンフィグレーションします。

- XML データをクライアントから受信する、またはクライアントに XML データを送信する JMS の送り先 (キューまたはトピック)。第 6 章「Ant タスクを使用した WebLogic Web サービスのアセンブル」に説明されているように後で Web サービスをアセンブルするときに、この JMS の送り先の名前を使用します。
- WebLogic Web サービスが JMS 接続を作成するために使用する JMS 接続ファクトリ。

この手順の詳細については、12-6 ページの「メッセージスタイル Web サービス用の JMS コンポーネントのコンフィグレーション」を参照してください。

メッセージスタイル Web サービス用の JMS コンポーネントのコンフィグレーション

この節では、既に JMS サーバをコンフィグレーションしていることを前提としています。JMS サーバのコンフィグレーションおよび JMS の一般的な情報については、『管理者ガイド』、および『WebLogic JMS プログラマーズ ガイド』を参照してください。

JMS の送り先 (キューまたはトピック) および JMS 接続ファクトリ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. ブラウザで Administration Console を起動します。詳細については、13-1 ページの「ブラウザで Administration Console を起動するには、次の URL を入力します。」を参照してください。
2. 左ペインで、[サービス] ノードをクリックして展開してから、[JMS] ノードを展開します。
3. [接続ファクトリ] ノードを右クリックして、ドロップダウン リストから [新しい JMSConnectionFactory のコンフィグレーション] を選択します。
4. 接続ファクトリの名前を [名前] フィールドに入力します。
5. 接続ファクトリの JNDI 名を [JNDI 名] フィールドに入力します。
6. [作成] をクリックします。
7. [対象] タブをクリックします。

8. サービスをホストしている WebLogic Server の名前がない場合は [選択済み] リスト ボックスにその名前を移動します。
9. [適用] をクリックします。
10. 左ペインで、[JMS] ノードの下の [サーバ] ノードをクリックして展開します。
11. 対象の JMS サーバのノードをクリックして展開します。
12. [送り先] ノードを右クリックして、次のいずれかを選択します。
    - トピックを作成する場合は、ドロップダウン リストから [新しい JMSTopic のコンフィグレーション] を選択します。
    - キューを作成する場合は、[新しい JMSQueue のコンフィグレーション] を選択します。
13. JMS の送り先の名前を [名前] テキスト フィールドに入力します。
14. 送り先の JNDI 名を [JNDI 名] フィールドに入力します。
15. [作成] をクリックします。

JMS で実装された WebLogic Web サービスの自動アセンブル

`servicegen` Ant タスクを使用して、JMS で実装された Web サービスを自動的にアセンブルできます。この Ant タスクは、`build.xml` ファイルの属性に基づいて `web-services.xml` デプロイメント記述子ファイルを作成し、必要に応じて (シリアライゼーション クラスなどの) 非組み込みデータ型コンポーネントを作成し、必要に応じて Web サービスを呼び出すために使用するクライアント JAR ファイルを作成し、すべてをデプロイ可能な EAR ファイルにパッケージ化します。

`servicegen` Ant タスクを使用して Web サービスを自動的にアセンブルするには次を行います。

1. Web サービスのコンポーネントを格納するステージング ディレクトリを作成します。

2. JMS メッセージのコンシューマとプロデューサ (メッセージ駆動型 Bean など) を JAR ファイルにパッケージ化します。

   この手順についての詳細は、『WebLogic Server アプリケーションの開発』を参照してください。

3. JAR ファイルをステージング ディレクトリにコピーします。

4. ステージング ディレクトリに、servicegen Ant タスクへの呼び出しを含む Ant ビルド ファイル (デフォルトのファイル名は build.xml) を作成します。

   servicegen Ant タスクを指定する方法の詳細については、12-8 ページの「servicegen Ant タスクを実行する」を参照してください。

   Ant ビルド ファイルの一般的な情報については、http://jakarta.apache.org/ant/manual/ を参照してください。

5. 環境を設定します。

   Windows NT では、setWLSEnv.cmd コマンドを実行します。*WL_HOME*\server\bin ディレクトリにあります。*WL_HOME* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

   UNIX では、setWLSEnv.sh コマンドを実行します。*WL_HOME*/server/bin ディレクトリにあります。*WL_HOME* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

6. ステージング ディレクトリで ant と入力し、必要であればこのコマンドにターゲットの引数を渡して、build.xml ファイルで指定された Ant タスク (1 つまたは複数) を実行します。

   ```
   prompt ant>
   ```

   Ant タスクによってステージング ディレクトリに Web サービス EAR ファイルが生成されるので、WebLogic Server へのデプロイが可能になります。

servicegen Ant タスクを実行する

以下のサンプル build.xml ファイルは、servicegen Ant タスクの実行方法を示しています。

```
<project name="buildWebservice" default="ear">
  <target name="ear">
    <servicegen
```

```
      destEar="jms_send_queue.ear"
      contextURI="WebServices" >
      <service
          JMSDestination="jms.destination.queue1"
          JMSAction="send"
          JMSDestinationType="queue"
          JMSConnectionFactory="jms.connectionFactory.queue"
          JMSOperationName="sendName"
          JMSMessageType="types.myType"
        generateTypes="True"
          targetNamespace="http://tempuri.org"
          serviceName="jmsSendQueueService"
          serviceURI="/jmsSendQueue"
          expandMethods="True">
      </service>
    </servicegen>
  </target>
</project>
```

この例では、servicegen Ant タスクは、jmsSendQueueService という Web サービスを 1 つ作成します。この Web サービスを識別するための URI は /jmsSendQueue です。Web サービスにアクセスするための完全 URL は次のとおりです。

```
http://host:port/WebServices/jmsSendQueue
```

servicegen Ant タスクは、jms_send_queue.ear という EAR ファイルに Web サービスをパッケージ化します。EAR ファイルには、web-services.war (デフォルト名) という、web-services.xml デプロイメント記述子ファイルなどの Web サービスコンポーネントがすべて格納された WAR ファイルが含まれています。

Web サービスは、sendName という 1 つのオペレーションをエクスポーズします。この Web サービスのオペレーションを呼び出すクライアント アプリケーションは、JNDI 名が jms.destination.queue1 である JMS キューにメッセージを送信します。このキューとの接続を作成するために使用される JMS ConnectionFactory は jms.connectionFactory.queue です。sendName オペレーションの 1 つのパラメータのデータ型は types.myType です。generateTypes 属性が True に設定されているので、servicegen Ant タスクはこのデータ型の非組み込みデータ型コンポーネント (シリアライゼーション クラスなど) を生成します。

注意： `types.myType` Java クラスは、`servicegen` の CLASSPATH で設定されている必要があります。設定されていないと、`servicegen` で適切なコンポーネントを生成できません。

JMS で実装された WebLogic Web サービスの手動アセンブル

JMS で実装された WebLogic Web サービスを手動でアセンブルするには、次の手順に従います。

1. Web サービスのアセンブルに関する JMS 固有情報についてのこの節の説明を読みます。
2. 適宜 JMS 固有情報を使用しながら、6-6 ページの「他の Ant タスクを使用した WebLogic Web サービスのアセンブル」で説明されている手順に従います。

以下の節では、Web サービスの手動アセンブルに関する JMS 固有情報について説明します。

JMS メッセージのコンシューマとプロデューサをパッケージ化する

JMS メッセージのコンシューマとプロデューサ (メッセージ駆動型 Bean など) を JAR ファイルにパッケージ化します。

Web サービス全体が入った EAR ファイルの作成時に、この JAR を、EJB JAR ファイルと同じく、最上位レベルのディレクトリに格納します。

コンポーネント情報で web-services.xml ファイルを更新する

Web サービスのオペレーションを実装する JMS バックエンド コンポーネントをリストおよび記述する <web-service> 要素の <components> 子要素を作成します。各バックエンド コンポーネントに、後で、そのコンポーネントが実装するオペレーションを記述するのに使用する name 属性があります。

例については、12-12 ページの「JMS コンポーネント Web サービス用の web-services.xml ファイルのサンプル」を参照してください。

JMS で実装された Web サービスに対しては、次のタイプのバックエンド コンポーネントをリストすることができます。

- <jms-send-destination>

  この要素は、クライアント アプリケーションからのデータの送信先である JMS バックエンド コンポーネントを記述します。このコンポーネントは、送信されたデータを JMS の送り先に配置します。この要素の connection-factory 属性を使用して、WebLogic Server が JMS 接続オブジェクトの作成に使用する JMS 接続ファクトリを指定します。次の例に示すように、<jndi-name> 子要素を使用して、送り先の JNDI 名を指定します。

  ```
  <components>
     <jms-send-destination name="inqueue"
                          connection-factory="myapp.myqueueCF">
         <jndi-name path="myapp.myqueueIN" />
     </jms-send-destination>
  </components>
  ```

- <jms-receive-queue>, <jms-receive-topic>

  これらの要素は、クライアント アプリケーションがデータを受信する、2 つの JMS バックエンド コンポーネントを記述します。第 1 のコンポーネントは JMS キューから、第 2 のコンポーネントは JMS トピックから受信します。connection-factory 属性を使用して、WebLogic Server が JMS 接続オブジェクトの作成に使用する JMS 接続ファクトリを指定します。次の例に示すように、<jndi-name> 子要素を使用して、キューまたはトピックの JNDI 名を指定します。

  ```
  <components>
     <jms-receive-queue name="outqueue"
                        connection-factory="myapp.myqueueCF">
  ```

```
                    <jndi-name path="myapp.myqueueOUT" />
                </jms-receive-queue>
            </components>
```

JMS コンポーネント Web サービス用の web-services.xml ファイルのサンプル

次の、web-services.xml ファイルのサンプルは、JMS メッセージのコンシューマまたはプロデューサを使用して実装される Web サービスを記述しています。

```
<web-services>
     <web-service targetNamespace="http://example.com"
                  name="myMessageService" uri="MessageService">

         <components>
            <jms-send-destination name="inqueue"
                                  connection-factory="myapp.myqueuecf">
                 <jndi-name path="myapp.myinputqueue" />
            </jms-send-destination>
            <jms-receive-queue name="outqueue"
                               connection-factory="myapp.myqueuecf">
                 <jndi-name path="myapp.myoutputqueue" />
            </jms-receive-queue>
         </components>

         <operations xmlns:xsd="http://www.w3.org/2001/XMLSchema">
              <operation invocation-style="one-way" name="enqueue"
                         component="inqueue" />
                 <params>
                   <param name="input_payload" style="in" type="xsd:anyType" />
                 </params>
              </operation>
              <operation invocation-style="request-response" name="dequeue"
                         component="outqueue" >
                 <params>
                   <return-param name="output_payload" type="xsd:anyType"/>
                 </params>
              </operation>
         </operations>
     </web-service>
</web-services>
```

この例では、クライアント アプリケーションが JMS の送り先にデータを送信する inqueue と、クライアント アプリケーションが JMS キューからデータを受信する outqueue という、2 つの JMS バックエンド コンポーネントが示されています。

enqueue と dequeue という、対応する 2 つの Web サービス オペレーションは、これらのバックエンド コンポーネントを使用して実装されます。

enqueue オペレーションは、inqueue コンポーネントを使用して実装されます。このオペレーションは、非同期一方向オペレーションとして定義されます。つまり、クライアント アプリケーションは、データを JMS の送り先に送信した後、SOAP 応答を (例外さえも) 受信しません。クライアントから送信されたデータは、input_payload パラメータに格納されます。

dequeue オペレーションは、outqueue コンポーネントを使用して実装されます。dequeue オペレーションは、JMS キューからデータを受信するのにクライアント アプリケーションによって呼び出されるので、同期要求応答として定義されます。応答データは、output_payload 出力パラメータに格納されます。

JMS で実装された WebLogic Web サービスのデプロイ

WebLogic Web サービスのデプロイとは、リモート クライアントでそのサービスを使用できるようにすることです。WebLogic Web サービスは標準 J2EE エンタープライズ アプリケーションとしてパッケージ化されているため、Web サービスのデプロイはエンタープライズ アプリケーションのデプロイと同じです。

エンタープライズ アプリケーションのデプロイメントの詳細については、「WebLogic Server デプロイメント」を参照してください。

JMS で実装された WebLogic Web サービスの呼び出し

この節では、JMS で実装された WebLogic Web サービスを呼び出すクライアント アプリケーションのサンプルを 2 つ示します。サービス オペレーションにデータを送信するクライアント アプリケーションのサンプルと、同じ Web サービス内の他のオペレーションからデータを受信するクライアント アプリケーションのサンプルです。この Web サービスを記述する、次の `web-services.xml` ファイルに示すように、第 1 のオペレーションは JMS の送り先を使用して実装されていて、第 2 のオペレーションは JMS キューを使用して実装されています。

```
<web-services xmlns:xsd="http://www.w3.org/2001/XMLSchema" >

  <web-service
    name="BounceService"
    targetNamespace="http://www.foobar.com/echo"
    uri="/BounceService">

      <components>

         <jms-send-destination name="inqueue"
                      connection-factory="weblogic.jms.ConnectionFactory">
              <jndi-name path="weblogic.jms.inqueue" />
        </jms-send-destination>
        <jms-receive-queue name="outqueue"
                     connection-factory="weblogic.jms.ConnectionFactory">
             <jndi-name path="weblogic.jms.outqueue" />
        </jms-receive-queue>
   </components>

   <operations xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <operation invocation-style="one-way" name="submit" component="inqueue" >
      </operation>

      <operation invocation-style="request-response"
                 name="query" component="outqueue" >
         <params>
             <return-param name="output_payload" type="xsd:string"/>
        </params>
      </operation>
   </operations>

 </web-service>
```

```
</web-services>
```

非同期 Web サービス オペレーションを呼び出してデータを送信する

次の例は、前節の web-services.xml ファイルで記述されている、submit オペレーションを呼び出す動的クライアント アプリケーションを示しています。submit オペレーションは、クライアント アプリケーションから JMS の送り先 weblogic.jms.inqueue にデータを送信します。このオペレーションは one-way と定義されているので非同期で、呼び出したクライアント アプリケーションにどのような値も返しません。

```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;

import javax.xml.namespace.QName;

/**
 * @author Copyright (c) 2002 by BEA Systems, Inc. All Rights Reserved.
 */

/**
 * send2WS - このモジュールは、特定の Web サービスに接続された JMS キューに送信する
 * メッセージが「中止」されると、このモジュールは終了する
 *
 * @返す
 * @例外を送出する
 */

public class send2WS{

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

  //グローバルな JAX-RPC サービス ファクトリをセットアップする
  System.setProperty( "javax.xml.rpc.ServiceFactory",
    "weblogic.webservice.core.rpc.ServiceFactoryImpl");

  ServiceFactory factory = ServiceFactory.newInstance();

  //QName を定義
  String targetNamespace = "http://www.foobar.com/echo";
```

```
  QName serviceName = new QName( targetNamespace, "BounceService" );
  QName portName = new QName( targetNamespace, "BounceServicePort" );

  //サービスを作成
    Service service = factory.createService( serviceName );

  // 発信呼び出しを作成
  Call Ws2JmsCall = service.createCall();

  QName operationName = new QName( targetNamespace, "submit" );

  //ポートとオペレーション名を設定
  Ws2JmsCall.setPortTypeName( portName );
  Ws2JmsCall.setOperationName( operationName );

  //パラメータを追加
  Ws2JmsCall.addParameter( "param",
      new QName( "http://www.w3.org/2001/XMLSchema", "string" ), ParameterMode.IN

);
  //エンドポイント アドレスの設定
  Ws2JmsCall.setTargetEndpointAddress(
        "http://localhost:7001/BounceBean/BounceService" );

  // ユーザからメッセージを取得
  BufferedReader msgStream =
      new BufferedReader(new InputStreamReader(System.in));
  String line = null;
  boolean quit = false;
  while (!quit) {
    System.out.print("Enter message (\"quit\" to quit): ");
    line = msgStream.readLine();
    if (line != null && line.trim().length() != 0) {
          String result = (String)Ws2JmsCall.invoke( new Object[ ]{ line } );
      if(line.equalsIgnoreCase("quit")) {
            quit = true;
            System.out.print("Done!");
        }
      }
    }
  }
}
```

同期 Web サービス オペレーションを呼び出してデータを送信する

次の例は、12-14 ページの「JMS で実装された WebLogic Web サービスの呼び出し」の web-services.xml ファイルで記述されている、query オペレーションを呼び出す動的クライアント アプリケーションを示しています。query オペレーションを呼び出すクライアント アプリケーションは、JMS キュー weblogic.jms.outqueue からデータを受信します。このオペレーションは request-response と定義されているので同期オペレーションで、JMS キューからクライアント アプリケーションにデータを返します。

```
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;

import javax.xml.namespace.QName;

/**
 * @author Copyright (c) 2002 by BEA Systems, Inc. All Rights Reserved.
 */

/**
 * fromWS - このモジュールは、JMS キューに関連付けられた Web サービスから受信する
 * メッセージが「中止」されると、このモジュールは終了する
 *
 * @返す
 * @例外を送出する
 */

public class fromWS {

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

  boolean quit = false;
  //グローバルな JAX-RPC サービス ファクトリをセットアップする
  System.setProperty( "javax.xml.rpc.ServiceFactory",
    "weblogic.webservice.core.rpc.ServiceFactoryImpl");

  ServiceFactory factory = ServiceFactory.newInstance();

  //QName を定義
  String targetNamespace = "http://www.foobar.com/echo";

  QName serviceName = new QName( targetNamespace, "BounceService" );
  QName portName = new QName( targetNamespace, "BounceServicePort" );
```

```
  //サービスを作成
    Service service = factory.createService( serviceName );

  //発信呼び出しを作成
  Call Ws2JmsCall = service.createCall();

  QName operationName = new QName( targetNamespace, "query" );

  //ポートとオペレーション名を設定
  Ws2JmsCall.setPortTypeName( portName );
  Ws2JmsCall.setOperationName( operationName );

  //パラメータを追加
  Ws2JmsCall.addParameter( "output_payload",
        new QName( "http://www.w3.org/2001/XMLSchema", "string" ),
                   ParameterMode.OUT);
  //エンドポイント アドレスの設定
  Ws2JmsCall.setTargetEndpointAddress(
        "http://localhost:7001/BounceBean/BounceService" );

  System.out.println("Setup complete.Waiting for a message...");

  while (!quit) {
        String result = (String)Ws2JmsCall.invoke( new Object[ ] {} );
        if(result != null) {
          System.out.println("TextMessage:" + result);
      if (result.equalsIgnoreCase("quit")) {
            quit = true;
        System.out.println("Done!");
          }
        continue;
        }
    try {Thread.sleep(2000);} catch (Exception ignore) {}
  }
 }
}
```

13 WebLogic Web サービスの管理

この章では、WebLogic Web サービスを管理するタスクについて説明します。

WebLogic Web サービスの管理の概要

WebLogic Web サービスを開発し、アセンブルしてデプロイしたら、Administration Console を使用して、WebLogic Server にデプロイされたその Web サービスを表示できます。Administration Console では、デプロイされた Web サービスで標準的な WebLogic 管理タスクを実行することもできます。

Web サービスは、EAR ファイルとしてパッケージ化します。EAR ファイルの構成要素は、`web-services.xml` ファイルおよび任意の Java クラス (Web サービス、ハンドラ、非組み込みデータ型のシリアライゼーション クラスを実装する Java クラスなど) が含まれる WAR ファイルと、Web サービスのオペレーションを実装するステートレス EJB または JMS コンシューマおよびプロデューサが含まれる任意の EJB JAR ファイルです。

Administration Console では、Web サービスはその WAR ファイルの内容で識別されます。つまり、EAR ファイル内の WAR ファイルに `web-services.xml` ファイルが含まれている場合、その WAR ファイルは Administration Console で Web サービスとしてリストされます。

Web アプリケーションに対する標準的管理タスクについては、「WebLogic Server Web コンポーネントのコンフィグレーション」を参照してください。

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、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、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。管理サーバのデフォルトのポート番号は 7001 です。

次の図は、Administration Console のメイン ウィンドウを示しています。

図 13-1　WebLogic Server Administration Console のメイン ウィンドウ

WebLogic Server にデプロイされている Web サービスの表示

WebLogic Web サービスは EAR ファイルとしてパッケージ化されていて、その中には次のコンポーネントが入っています。

- `web-services.xml` デプロイメント記述子、省略可能なサポート クラス ファイル (SOAP メッセージ ハンドラ 実装、シリアライゼーション クラスなど)、その他の入った Web アプリケーション WAR ファイル。Administration Console では、この WAR ファイルは、Web サービス コンポーネントと呼ばれています。
- Web サービスのオペレーションを実装する EJB が格納された任意の EJB JAR ファイル

WebLogic Server にデプロイされている Web サービスを表示するには、次の手順を行います。

1. ブラウザで Administration Console を起動します。13-1 ページの「WebLogic Web サービスの管理の概要」を参照してください。
2. 左ペインで、[デプロイメント] | [Web サービス コンポーネント] ノードを展開します。

   デプロイされた Web サービスのリストが、[Web サービス コンポーネント] ノードの下と右ペインのテーブルに表示されます。

   Web サービスのリストは、`web-services.xml` デプロイメント記述子の入っている、デプロイされた Web アプリケーションのリストに対応しています。`web-services.xml` が入っているほかは、Web サービス WAR ファイルは、標準の Web アプリケーション WAR ファイルと同じで、Administration Console でも同様に扱われます。

   右ペインのテーブルの [アプリケーション] カラムには、Web サービス コンポーネントが格納されているエンタープライズ アプリケーションがリストされます。

   次の図では、デプロイされた Web サービス コンポーネントが 1 つ示されています。これは `trader_service.war` というコンポーネント名で、

_appsdir_webservices_trader_ear というエンタープライズ アプリケーションの構成部分です。

図 13-2　デプロイされた Web サービス

3. EJB JAR ファイルなどの Web サービスを構成する他のコンポーネントを表示するには、左ペインの [デプロイメント] ノード下の [アプリケーション] ノードを展開します。

4. ここで使用した例にある _appsdir_webservices_trader_ear などの、Web サービス コンポーネントの入っているエンタープライズ アプリケーションの名前をクリックします。アプリケーションに格納されているすべてのコンポーネントのリストが、左ペインのアプリケーション名の下に表示されます。

先の図を見ると、_appsdir_webservices_trader_ear エンタープライズ アプリケーションには、trader_services.war コンポーネントのほかに、trader.jar という名前の EJB JAR ファイルも入っていることがわかります。この EJB JAR ファイルには、Web サービス オペレーションの一部または全部を実装する EJB バックエンド コンポーネントが入っています。

14 UDDI を使用した Web サービスのパブリッシュと検索

この章では、UDDI を使用した Web サービスのパブリッシュと検索に関する情報を提供します。

UDDI の概要

UDDI とは、Universal Description, Discovery and Integration を略したものです。UDDI プロジェクトは、企業間の相互の取引を迅速、簡単、かつダイナミックに発見、実行することを目指す、業界をリードするプロジェクトです。

UDDI レジストリには、ビジネスに関するカタログ化された情報、その企業が提供するサービス、および取引の際に使用する通信標準とインタフェースが格納されます。

UDDI は、SOAP (Simple Object Access Protocol) データ通信標準に基づいて構築されており、企業にとって有益な、グローバルでプラットフォームに依存しないオープン アーキテクチャ空間を作成します。

UDDI レジストリは、大きくは次の 2 つのカテゴリに分けることができます。

- UDDI と Web サービス
- UDDI とビジネス レジストリ

UDDI のデータ構造の詳細については、14-3 ページの「UDDI のデータ構造」を参照してください。

UDDI と Web サービス

Web サービスのオーナは、Web サービスを UDDI レジストリにパブリッシュします。パブリッシュ後は、Web サービスの説明およびサービスに対するポインタが UDDI レジストリに保持されます。

クライアントは、このレジストリを検索し、必要なサービスを見つけてその詳細を取り出すことができます。取り出せる詳細には、サービスの呼び出しポイントを始め、サービスやその機能性を確認するための情報が含まれています。

Web サービスの機能は、プログラミング インタフェースによってエクスポーズされており、通常は WSDL (Web Services Description Language) で記述されています。一般的な流れとしては、まずプロバイダがそのビジネスをパブリッシュしてサービスを登録し、Web サービスに関する技術的な情報をもとにバインディング テンプレートを定義します。バインディング テンプレートには、この Web サービスによって実装される抽象インタフェースを表す 1 つまたは複数の *tModel* への参照も格納されます。tModel は、プロバイダによってユニークにパブリッシュされるもので、WSDL ドキュメントに対するインタフェースや URL 参照に関する情報が格納されます。

続いて、以下のいずれかを目的として、クライアントが照会を行います。

1. 既知のインタフェースの実装を探す。

   つまり、クライアントが tModel ID を知っていて、その tModel ID を参照するバインディング テンプレートを探す場合です。

2. 既知のバインディング テンプレート ID の呼び出しポイント (たとえばアクセス ポイント) の更新値を調べる。

UDDI とビジネス レジストリ

UDDI によるビジネス レジストリ ソリューションとしては、企業が提供する製品やサービスを宣伝したり、Web 上での取引の方法などについて告知することなどが考えられます。UDDI をこうした用途に使用することで、企業間 (B2B) の電子コマースの発展を促進することができます。

ビジネスは、必要最低限の情報としてビジネスの名前さえあればパブリッシュできます。ビジネス エンティティに関する詳細な情報を記述してパブリッシュすれば、ビジネス エンティティ自体およびその製品やサービスを、正確かつアクセスしやすい形で宣伝できます。

ビジネス レジストリには以下を格納できます。

- **ビジネス ID**— ビジネスの複数の名前と説明、広範な連絡先情報、税金 ID 等の一般的なビジネス ID
- **カテゴリ**—標準的なカテゴリ情報 (たとえば D-U-N-S ビジネス カテゴリ番号)
- **サービスの説明** — サービスの複数の名前と説明。サービス情報のコンテナとして使用することで、さまざまなサービスについて、その所有権を明確に示しながら宣伝することができます。`bindingTemplate` 情報には、サービスへのアクセス方法を示します。
- **規格の準拠** — 状況によっては、準拠している規格を示すことが非常に重要になります。こうした規格によって、サービスを使用する上での技術的な要件が詳細に示されている場合もあります。
- **カスタム カテゴリ**—ビジネスやサービスを識別したり分類したりするための独自の仕様 (tModels) をパブリッシュできます。

UDDI のデータ構造

UDDI 内のデータ構造は、`businessEntity` 構造体、`businessService` 構造体、`bindingTemplate` 構造体、および `tModel` 構造体の 4 つから構成されます。

次の表に、これらの構造体を Web サービスやビジネス リポジトリのアプリケーションで使用する際の相違点をまとめます。

表 14-1 UDDI のデータ構造

データ構造	Web サービス	ビジネス レジストリ
businessEntity	Web サービス プロバイダを表す。 • 企業名 • 連絡先 • その他の企業情報	企業、および企業内の部門や部署を表す。 • 企業名 • 連絡先 • ID およびカテゴリ
businessService	1 つまたは複数の Web サービスの論理グループ。 子要素として格納された単一の名前を持つ API で、businessEntity に指定したビジネス エンティティに含まれる。	単一の `businessEntity` に含まれるサービスのグループ。 • 複数の名前と説明 • カテゴリ • 規格の準拠を示すインジケータ
bindingTemplate	単一の Web サービス。 ここに指定する情報によって、クライアント アプリケーションが Web サービスをバインドしたり、Web サービスと対話したりするのに必要な技術情報が提供される。 アクセス ポイント (Web サービスを呼び出すための URI など) も含まれる。	規格の準拠に関する詳細。 サービスのアクセス ポイント (URL、電話番号、電子メール アドレス、ファックス番号など)。
tModel	技術的な仕様 (仕様ポインタ、仕様書に関するメタデータなど) を表す。実際の仕様を指す名前や URL も含まれる。Web サービスの場合、実際の仕様書の形式は WSDL ファイル。	ユーザが特定の用途のために確立または登録した規格や技術仕様を表す。

WebLogic Server における UDDI の機能

WebLogic Server は、以下の UDDI 機能を提供します。

- UDDI ディレクトリ エクスプローラ
- UDDI レジストリ
- Web サービスの検索とパブリッシュをプログラムで設定できるクライアント サイド UDDI API の実装

UDDI ディレクトリ エクスプローラを使用すると、認可されたユーザは、WebLogic Server のプライベートな UDDI レジストリをパブリッシュしたり、すでに公開されている Web サービスの情報を変更することができます。

また、UDDI ディレクトリ エクスプローラを使用して、パブリックとプライベート両方の UDDI レジストリにある Web サービスおよびこれらの Web サービスを提供している企業や部課に関する情報を検索できます。さらに、ディレクトリ エクスプローラを使用して、Web サービスおよび関連付けられた WSDL ファイル (利用可能な場合) に関する詳細にアクセスすることもできます。

UDDI ディレクトリ エクスプローラの呼び出し

ブラウザで UDDI ディレクトリ エクスプローラ を起動するには、次の URL を入力します。

```
http://host:port/uddiexplorer
```

各要素の説明は次のとおりです。

- *host* は、WebLogic Server が動作しているコンピュータの名前。
- *port* は、WebLogic Server が接続要求のリスニングを行っているポート番号。デフォルトのポート番号は 7001 です。

UDDI ディレクトリ エクスプローラを使用して、以下のタスクを実行できます。

- パブリック レジストリの検索
- プライベート レジストリの検索
- プライベート レジストリへのパブリッシュ
- プライベート レジストリの詳細の変更
- UDDI ディレクトリ エクスプローラのセットアップ

UDDI ディレクトリ エクスプローラの使用法の詳細については、メイン ページのヘルプのリンクをクリックしてください。

UDDI クライアント API の使用法

Web サービスの検索やパブリッシュに、Java クライアント アプリケーションで UDDI クライアント API を使用します。

UDDI クライアント API には、`Inquiry` と `Publish` という、2 つの主なクラスがあります。`Inquiry` クラスは、既知の UDDI レジストリにある Web サービスの検索に使用し、`Publish` クラスは、開発した Web サービスを既知のレジストレリに追加するために使用します。

WebLogic Server では、以下のクライアント UDDI API パッケージを実装しています。

- `weblogic.uddi.client.service`
- `weblogic.uddi.client.structures.datatypes`
- `weblogic.uddi.client.structures.exception`
- `weblogic.uddi.client.structures.request`
- `weblogic.uddi.client.structures.response`

これらのパッケージの使用法の詳細については、UDDI API に関する Javadocs を参照してください。

UDDI クライアント API の使用例については、Web Services dev2dev Download Page をスクロールして、以下の例を探してください。

- UDDI Client API Example

- UDDI Publish Example
- UDDI Inquire Example

15 相互運用性

この章では、Web サービスの相互運用性の概要を説明し、最大限の相互運用性を備えた Web サービス作成に役立つヒントを提供します。

相互運用性の概要

Web サービスの基本的な特徴は、相互運用性があることです。つまり、ハードウェアやソフトウェアの相違にかかわらず、クライアントから Web サービスを呼び出すことができます。特に、相互運用性が保証されるためには、以下の事項が異なっている場合も Web サービス アプリケーションの機能が同じであることが要求されます。

- BEA WebLogic Server、IBM Websphere、Microsoft .NET などのアプリケーション プラットフォーム
- Java、C++、C#、Visual Basic などのプログラミング言語
- メインフレーム、PC、周辺デバイスなどのハードウェア
- 各種のバリエーションを含めた、UNIX、Windows などのオペレーティング システム

- アプリケーションのデータ モデル

たとえば、Solaris が搭載された Sun Microsystems のコンピュータで稼働している WebLogic Server で実行されている相互運用性を持つ Web サービスを、Visual Basic で記述された Microsoft .NET の Web サービス クライアントから呼び出すことができます。

最大限の相互運用性を保証するため、WebLogic Server では、Web サービスの生成時に以下の仕様およびバージョンをサポートします。

- HTTP 1.1 (トランスポート プロトコル)
- XML スキーマ (データの記述)
- WSDL 1.1 (Web サービスの記述)
- SOAP 1.1 (メッセージ フォーマット)

以下の節では、相互運用性に関して、Web サービス アプリケーションを作成する際に役立つヒントと情報を提供します。

ベンダ固有の拡張機能を使用しないこと

SOAP、WSDL、HTTP などの、Web サービスで使用されている仕様に対するベンダ固有の拡張実装を使用しないでください。このような拡張実装に依存する Web サービスを、それらを使用していないクライアント アプリケーションから呼び出そうとした場合、呼び出しに失敗することがあります。

最新の相互運用性テスト情報の常時更新

公開の相互運用性テストにより、Web サービス仕様のさまざまなベンダによる実装間の相互運用性に関する情報が提供されています。この情報は、WebLogic Server で、たとえば、.NET による Web サービスなど、他のベンダが提供する Web サービスとの相互運用性を要件とする Web サービスを作成する場合に役立ちます。

警告： BEA もこうした相互運用性テストに参加しています。ただし、テストに参加する他のプラットフォームへの BEA Web サービスの実装を BEA が公式に保証しているわけではありません。

次の Web サイトに、公開相互運用性テスト結果が掲載されています。

- Web Service Interoperability Organization
- SoapBuilder Interoperability Lab

また、これらの Web サイトにリストされているベンダ実装を使用して、作成した Web サービスの相互運用性を徹底的にテストすることもできます。

作成したアプリケーションのデータ モデルの理解

Web サービスの効果的な利用法は、既存アプリケーションの統合に役立つクロスプラットフォーム技術の提供です。これらのアプリケーションは、データ モデルが大きく異なっているため、作成した Web サービスを調整しなければならない場合が一般的です。

たとえば、ある大企業の 2 つの会計システムを統合する Web サービス アプリケーションを作成しているとします。 各会計システムのデータ モデルは類似していても、データ フィールドの名前、各顧客について保存される情報量など、多少の相違は存在します。各データ モデルを理解した上で、両者を調和させる中間的なデータ モデルを作成することはプログラマの力量にかかっています。この中間的なデータ モデルは、XML スキーマを使用して XML で表現されます。ただ 1 つのデータ モデルに基づいて作成された Web サービス アプリケーションは、他のアプリケーションとの相互運用性が不十分となることが想像されます。

さまざまなデータ型の相互運用性の理解

Web サービス オペレーションのパラメータと戻り値のデータ型は、Web サービスの相互運用性を大きく左右します。次の表は、さまざまなデータ型の相互運用性の説明です。

表 15-1 各種データ型の相互運用性

データ型	説明
JAX-RPC 組み込みデータ型	プログラミングの追加なしで相互運用可能。 JAX-RPC 仕様では、XML スキーマの組み込みデータ型のサブセットを定義している。これらのデータ型はすべて、SOAP-ENC データ型に直接マップされるので、相互運用性がある。
WebLogic Server 組み込みデータ型	プログラミングの追加なしで相互運用可能。 WebLogic Server は、XML スキーマのすべての組み込みデータ型をサポートする。これらのデータ型はすべて、SOAP-ENC データ型に直接マップされるので、相互運用性がある。 WebLogic Server の組み込みデータ型の完全なリストは、5-13 ページの「組み込みデータ型の使用法」参照。

表 15-1 各種データ型の相互運用性

データ型	説明
非組み込みデータ型	プログラミングの追加やツールのサポートにより相互運用可能。 非組み込みデータ型の Web サービスを作成する場合は、データの XML 表現を記述する XML スキーマ、Java 表現を記述する Java クラス、XML 表現と Java 表現間でデータを変換するシリアライゼーション クラスを作成する必要がある。WebLogic Server には、これらのオブジェクトを自動的に生成する `servicegen` Ant タスクおよび `autotype` Ant タスクが組み込まれている。ただし、これらの Ant タスクが生成した XML スキーマの相互運用性が不十分な場合があり、また、Java データ型があまりにも複雑なものであった場合は、XML スキーマの作成に失敗するおそれもある。このような理由で、第 9 章「非組み込みデータ型の使用法」で説明するように、非組み込みデータ型が必要とするオブジェクトを手動で作成しなければならない場合がある。 さらに、Web サービスを呼び出すクライアント アプリケーションに、XML 表現とそのクライアントアプリケーションのプログラミング言語固有の表現間のデータ変換に必要なシリアライゼーション クラスが組み込まれていることを保証する必要がある。WebLogic Server では、Web クライアント アプリケーションが必要とするシリアライゼーション クラスを、`clientgen` Ant タスクを使用して生成することができる。ただし、作成した Web サービスを呼び出すクライアント アプリケーションが Java 以外の言語で記述されている場合は、シリアライゼーション クラスを手動で作成する必要がある。

SOAPBuilders Interoperability Lab Round 3 テストの結果

SOAPBuilders Interoperability Lab Round 3 テストに WebLogic Web サービスが参加した結果については、http://webservice.bea.com:7001 を参照してください。このテストは、バージョン 7.0.0.1 の WebLogic Server を使用して実行されました。

テストの結果については http://webservice.bea.com/index.html#qz41 を、テストのソース コードについては http://webservice.bea.com/index.html#qz40 を参照してください。

SOAPBuilder Interoperability テストの詳細については、http://www.whitemesa.net. を参照してください。

警告： BEA もこうした相互運用性テストに参加しています。ただし、テストに参加する他のプラットフォームへの BEA Web サービスの実装を BEA が公式に保証しているわけではありません。

.NET との相互運用

WebLogic Web サービスは、.NET Web サービスとシームレスに統合されます。

.NET Web サービスは、WebLogic Web サービスのクライアント アプリケーションから第 8 章「Invoking Web Services」で説明されているとおりに呼び出します。 `clientgen` Ant タスクを実行して Web サービス固有のクライアント JAR ファイルを生成する場合は、`wsdl` 属性を使用して、デプロイされている .NET Web サービスの WSDL の URL を指定します。

.NET クライアント アプリケーションから、デプロイされている WebLogic Web サービスを呼び出すには、Microsoft Visual Studio .NET を使用してアプリケーションを作成し、次に以下の例に従って Web 参照を追加して、デプロイされている WebLogic Web サービスの WSDL を指定します。Microsoft Visual Studio では、Web 参照の追加は、WebLogic `clientgen` Ant タスクの実行と同じです。

警告： 次の例は、WebLogic Web サービスを .NET クライアント アプリケーションから呼び出すための 1 つの方法を示したものです。Microsoft Visual Studio .NET を使用して WebLogic (および他の) Web サービスを呼び出す方法に関する最新情報と詳細情報については、Microsoft のドキュメント を参照してください。

1. Microsoft Visual Studio .NET を起動し、通常の方法でアプリケーションを作成します。
2. 右ペインのソリューション エクスプローラで、作成したアプリケーションを右クリックして [Web 参照の追加] を選択します。ソリューション エクスプローラ ブラウザが開きます。
3. ソリューション エクスプローラ ブラウザで、デプロイされている WebLogic Web サービスの WSDL を入力します。 ブラウザが WSDL を受け付けると、[参照の追加] ボタンがアクティブになります。

   デプロイされている WebLogic Web サービスの WSDL を取得する方法の詳細については、8-23 ページの「WebLogic Web サービスのホーム ページおよび WSDL の URL」を参照してください。
4. [参照の追加] ボタンをクリックします。ソリューション エクスプローラに、WebLogic Web サービスが表示されます。
5. Web サービスを呼び出すために使用するアプリケーション コンポーネント (ボタンなど) に、Web サービスの特定のオペレーションを呼び出すための Visual C# または Visual Basic コードを追加します。Visual Studio .NET は、ステートメント補完機能を使用してこのコードの記述を支援します。次の Visual C# コードは、`SoapInteropBaseService` Web サービスの `echoString` オペレーションを呼び出す単純な例を示しています。

```
WebReference1.SoapInteropBaseService s = new SoapInteropBaseService();
string s = s.echoString("Hi there!");
```

この例では、`WebReference1` は前の手順で追加した Web 参照の名前です。

16 6.1 WebLogic Web サービスから 7.0 へのアップグレード

この章では、6.1 WebLogic Web サービスを 7.0 にアップグレードする方法について説明します。

6.1 WebLogic Web サービスのアップグレードの概要

バージョン 6.1 と 7.0 の WebLogic Server 間の Web サービス実行時システムの変更に伴い、バージョン 6.1 で作成した Web サービスをバージョン 7.0 で実行するにはアップグレードが必要になります。この節では、アップグレードの手順について説明します。

6.1 Web サービスは、以下の 2 つの方法でアップグレードできます。

- `wsgen` Ant タスクを使用して行われる自動アップグレード。この Ant タスクは、対応する 6.1 属性を持たないすべての必須 7.0 `web-services.xml` 属性に対してデフォルト値を使用します。

詳細については、16-2 ページの「6.1 WebLogic Web サービスから 7.0 への自動アップグレード」を参照してください。

- 6.1 Web サービスの作成に使用した `build.xml` ファイルの書き換えによる手動アップグレード。この場合、`wsgen` Ant タスクではなく `servicegen` Ant タスクを呼び出します。

  詳細については、16-4 ページの「6.1 WebLogic Web サービスから 7.0 への手動アップグレード」を参照してください。

注意： 6.1 Web サービスを 7.0 WebLogic Server インスタンス上にデプロイすることはできません。

バージョン 6.1 の WebLogic Server に含まれていた WebLogic Web サービス クライアント API は非推奨になったため、これを使用して 7.0 Web サービスを呼び出さないでください。バージョン 7.0 には、Java API for XML based RPC (JAX-RPC) に基づいた新しいクライアント API が含まれています。6.1 Web サービス クライアントを使用しているクライアント アプリケーションは、JAX-RPC API を使用するよう書き換える必要があります。詳細については、第 8 章「Web サービスの呼び出し」を参照してください。

6.1 と 7.0 の Web サービス間の違いの詳細については、第 1 章「WebLogic Web サービスの概要」を参照してください。

6.1 WebLogic Web サービスから 7.0 への自動アップグレード

Web サービスを 6.1 から 7.0 に自動アップグレードするには `wsgen` Ant タスクを使用します。このタスクは、6.1 Web サービスのアセンブルに使用した `build.xml` ファイルを入力として受け取り、このファイル内の値に基づいて次の操作を行います。

- Web サービスを記述する `web-services.xml` ファイルを生成します。このタスクは、6.1 Web サービスに適用されていなかったデプロイメント記述子内のすべての必須要素または属性に対してデフォルト値を使用します。このデフォルト値の詳細については、付録 A「WebLogic Web サービス デプロイメント記述子の要素」を参照してください。

- Web サービスを呼び出すクライアント アプリケーションが必要とする Web サービス固有のクラス、スタブ、およびインタフェースを含むクライアント JAR ファイルを必要に応じて作成します。クラス、スタブ、インタフェースは JAX-RPC API に基づいています。
- Web アプリケーション WAR ファイルにすべての Web サービス コンポーネントをパッケージ化し、次に、WAR ファイルと EJB JAR ファイルをデプロイ可能な EAR ファイルにパッケージ化します。

6.1 WebLogic Web サービスを 7.0 に自動でアップグレードするには、次の手順に従います。

1. 一時的なステージング ディレクトリを作成します。
2. 6.1 RPC スタイル Web サービスをアップグレードしている場合は、サービスを実装する EJB を含む EJB JAR ファイルをサポート EJB とともにステージング ディレクトリにコピーします。
3. `wsgen` Ant タスクへの呼び出しを含む 6.1 `build.xml` ファイルをステージング ディレクトリにコピーします。
4. 6.1 `build.xml` ファイルで `<taskdef>` 要素を使用して `wsgen` Ant タスクを明示的に宣言している場合は (以下の例を参照)、

```
<taskdef name="wsgen" classname="weblogic.ant.taskdefs.ejb.WSGen"/>
```

   この `<taskdef>` 要素を `build.xml` ファイルから削除します。

5. 必要に応じて、`targetNameSpace` 属性と `packageName` 属性を `build.xml` ファイルの `<wsgen>` 要素に追加します。これらの省略可能な属性については、B-39 ページの「wsgen」を参照してください。
6. WebLogic Server 7.0 環境を設定します。

   Windows NT では、`setWLSEnv.cmd` コマンドを実行します。*`WL_HOME`*`\server\bin` ディレクトリにあります。*`WL_HOME`* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

   UNIX では、`setWLSEnv.sh` コマンドを実行します。*`WL_HOME`*`/server/bin` ディレクトリにあります。*`WL_HOME`* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

7. `build.xml` ファイルと同じディレクトリに `ant` を入力し、`build.xml` ファイル内の `wsgen` Ant タスクを実行します。

```
prompt> ant
```

Ant タスクにより、WebLogic Server 7.0 上でデプロイ可能な EAR ファイルが生成されます。

8. クライアント アプリケーションで、Web サービスまたは Web サービスの WSDL へのアクセスに使用する URL を 6.1 で使用されていた URL から 7.0 の URL に更新します。詳細については、16-7 ページの「Web サービスへのアクセスに使用される URL の更新」を参照してください。

6.1 WebLogic Web サービスから 7.0 への手動アップグレード

6.1 WebLogic Web サービスを 7.0 に手動でアップグレードするには、次の手順に従います。

1. `wsgen` Ant タスク で 6.1 Web サービスのアセンブルに使用した `build.xml` Ant ビルド ファイルを、`servicegen` Ant タスクを呼び出す 7.0 バージョンに変換します。

   詳細については、16-5 ページの「6.1 build.xml ファイルから 7.0 への変換」を参照してください。

2. 6.1 Web サービス EAR ファイルを unJAR し、ステートレス セッション EJB (6.1 RPC スタイル Web サービスの場合) またはメッセージ駆動型 Bean (6.1 メッセージスタイル Web サービスの場合) を含む EJB JAR ファイルを抽出します。サポート クラス ファイルがあればそれも抽出します。

3. 6.1 Web サービスが RPC スタイルの場合、`servicegen` Ant タスクの使用法については 6-3 ページの「servicegen Ant タスクを使用した WebLogic Web サービスのアセンブル」を参照してください。

   6.1 Web サービスがメッセージスタイルの場合、7.0 Web サービスを手動でアセンブルする必要があります。手順の詳細については、第 12 章「JMS で実装された WebLogic Web サービスの作成」を参照してください。

4. クライアント アプリケーションで、Web サービスまたは Web サービスの WSDL へのアクセスに使用する URL を 6.1 で使用されていた URL から 7.0 の URL に更新します。詳細については、16-7 ページの「Web サービスへのアクセスに使用される URL の更新」を参照してください。

6.1 build.xml ファイルから 7.0 への変換

Web サービスのアセンブルに使用される `build.xml` ファイルの 6.1 と 7.0 の主な違いは Ant タスクです。6.1 では、タスクは `wsgen` で呼び出され、7.0 では、`servicegen` で呼び出されます。`servicegen` Ant タスクは、`wsgen` と同じ要素と属性を多数使用します。ただし、一部は非適用になっています。`servicegen` Ant タスクにも、追加のコンフィグレーション オプションが含まれています。この節の最後にある表では、この 2 つの Ant タスクの要素間と属性間のマッピングについて説明します。

注意： 6.1 では `wsgen` Ant タスクを使用してメッセージスタイル Web サービスをアセンブルできましたが、`servicegen` Ant タスクは、JMS 実装されたバックエンド コンポーネントに対してこれに相当する 7.0 機能を装備していません。このため、6.1 メッセージスタイル Web サービスをアップグレードしている場合は、以前の `build.xml` ファイルの変換は不要です。

次の `build.xml` は、6.1 RPC スタイル Web サービスの例からの抜粋です。

```
<project name="myProject" default="wsgen">
        <target name="wsgen">
           <wsgen destpath="weather.ear"
                  context="/weather">
                <rpcservices path="weather.jar">
                   <rpcservice bean="statelessSession"
                               uri="/weatheruri"/>
                   </rpcservices>
           </wsgen>
        </target>
</project>
```

次の例では、これに相当する 7.0 `build.xml` ファイルを示します。

```
<project name="myProject" default="servicegen">
    <target name="servicegen">
       <servicegen
           destEar="weather.ear"
```

```
            contextURI="weather" >
            <service
                ejbJar="weather.jar"
                serviceURI="/weatheruri"
                includeEJBs="statelessSession" >
            </service>
        </servicegen>
    </target>
</project>
```

WebLogic Web サービス Ant タスクの詳細については、付録 B「Web サービス Ant タスクとコマンドライン ユーティリティ」を参照してください。

次の表では、6.1 `wsgen` 要素および属性と、それに相当する 7.0 `servicegen` 要素および属性との対応を示します。

表 16-1　6.1 から 7.0 への wsgen Ant タスクのマッピング

6.1 wsgen 要素	属性	相当する 7.0 servicegen 要素	属性
wsgen	basepath	相当する属性なし	相当する属性なし
	destpath	servicegen	destEar
	context	servicegen	contextURI
	protocol	servicegen.service	protocol
	host	相当する属性なし	相当する属性なし
	port	相当する属性なし	相当する属性なし
	webapp	servicegen	warName
	classpath	servicegen	classpath
rpcservices	module	相当する属性なし	相当する属性なし
	path	servicegen.service	ejbJar

表 16-1 6.1 から 7.0 への wsgen Ant タスクのマッピング (続き)

6.1 wsgen 要素	属性	相当する 7.0 servicegen 要素	属性
rpcservice	bean	servicegen.service	includeEJBS、excludeEJBs
	uri	servicegen.service	serviceURI
messageservices	なし	相当する属性なし	相当する属性なし
messageservice	全属性	相当する属性なし	相当する属性なし
clientjar	path	servicegen.service.client	clientJarName

Web サービスへのアクセスに使用される URL の更新

クライアント アプリケーションが WebLogic Web サービスへのアクセスに使用するデフォルトの URL とその WSDL は、バージョン 6.1 から 7.0 の WebLogic Server になったときに変更されました。

バージョン 6.1 のデフォルトの URL は次のとおりです。

```
[protocol]://[host]:[port]/[context]/[WSname]/[WSname].wsdl
```

詳細については、「WebLogic Web サービスを呼び出して WSDL を取得する URL」を参照してください。

たとえば、16-5 ページの「6.1 build.xml ファイルから 7.0 への変換」で示した `build.xml` ファイルで構築した 6.1 Web サービスを呼び出す URL は次のとおりです。

```
http://host:port/weather/statelessSession.WeatherHome/statelessSession.WeatherHome.wsdl
```

7.0 のデフォルトの URL は次のとおりです。

```
[protocol]://[host]:[port]/[contextURI]/[serviceURI]?WSDL
```

詳細については、8-23 ページの「WebLogic Web サービスのホーム ページおよび WSDL の URL」を参照してください。

たとえば、16-5 ページの「6.1 build.xml ファイルから 7.0 への変換」で示した 6.1 `build.xml` ファイルを変換して `wsgen` を実行した後で、相当する 7.0 Web サービスを呼び出す URL は次のとおりです。

```
http://host:port/weather/weatheruri?WSDL
```

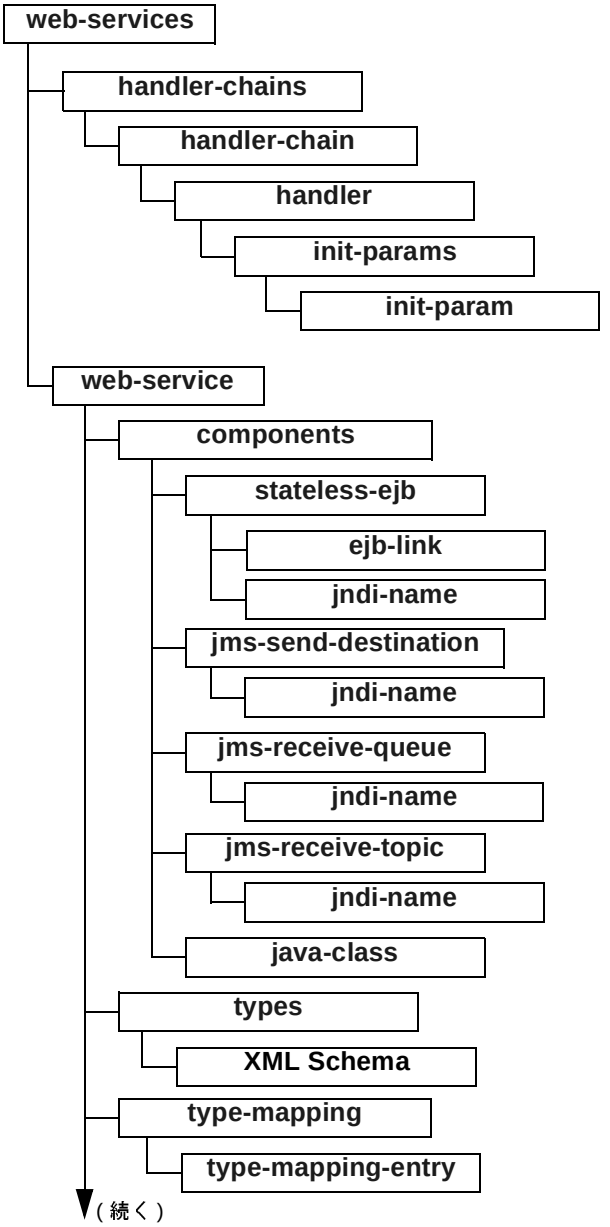
A WebLogic Web サービス デプロイメント記述子の要素

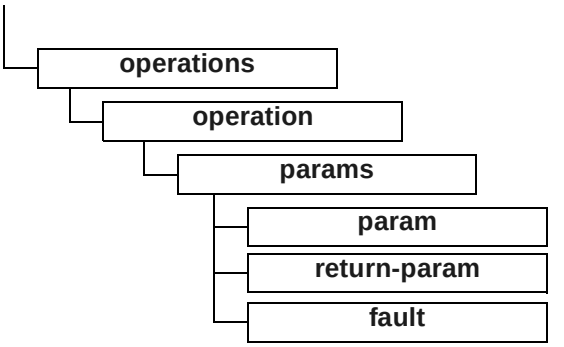
web-services.xml デプロイメント記述子ファイルには、1 つまたは複数の WebLogic Web サービスを記述する情報が格納されます。この情報には、Web サービスの操作を実装するバックエンド コンポーネント、パラメータおよび戻り値として使用される非組み込みデータ型、SOAP メッセージをインターセプトする SOAP メッセージ ハンドラなどの詳細情報が含まれます。他のすべてのデプロイメント記述子の場合と同様に、web-services.xml も XML ファイルです。

この章では、異なる形式を使用して web-services.xml ファイルについて説明します。

階層図

次の図は、web-services.xml 要素の階層を示しています。


web-services
handler-chains
handler-chain
handler
init-params
init-param
web-service
components
stateless-ejb
ejb-link
jndi-name
jms-send-destination
jndi-name
jms-receive-queue
jndi-name
jms-receive-topic
jndi-name
java-class
types
XML Schema
type-mapping
type-mapping-entry
(続く)



要素について

アルファベット順に構成された以降の節では、`web-services.xml` ファイルの各要素を説明します。

さまざまなタイプの WebLogic Web サービス用の Web サービスデプロイメント記述子ファイルのサンプルについては、7-11 ページの「Sample web-services.xml Files」を参照してください。

components

Web サービスを実装するバックエンド コンポーネントを定義します。

WebLogic Web サービスは、以下の 1 つまたは複数のコンポーネントを使用して実装できます。

- ステートレス セッション EJB
- JMS 送り先
- Java クラス

この要素には属性はありません。

ejb-link

EJB JAR ファイル内のどの EJB を使用してステートレス セッション EJB バックエンド コンポーネントを実装するかを指定します。

属性	説明	データ型	必須 / 任意
path	次の形式の EJB の名前。 *jar-name*#*ejb-name* *jar-name* は、ステートレス セッション EJB を含んでいる JAR ファイル (Web サービス EAR ファイル内に含まれている) の名前を指す。名前には、最上位の EAR ファイルへの相対パス名が含まれている必要がある。 *ejb-name* は、EJB JAR ファイル内の ejb-jar.xml デプロイメント記述子内の <ejb-name> 要素に対応するステートレス セッション EJB の名前を指す。 例 : myapp.jar#StockQuoteBean	String	必須

fault

この操作の呼び出しでエラーが発生した場合に送出される SOAP 障害を指定します。

この要素は必須ではありません。

属性	説明	データ型	必須 / 任意
name	パラメータの名前。	String	必須
class-name	SOAP ハンドラを実装する完全修飾 Java クラス。	String	必須

handler

ハンドラ チェーン内の SOAP メッセージハンドラを指定します。1 つのハンドラ チェーンは 1 つまたは複数のハンドラで構成できます。

ハンドラを実装する Java クラスに初期化パラメータが必要な場合は、<handler> 要素の <init-params> 子要素を任意に使用してパラメータを指定します。

属性	説明	データ型	必須 / 任意
class-name	SOAP メッセージ ハンドラを実装する完全修飾 Java クラス。	String	必須

handler-chain

特定のハンドラ チェーンを構成する SOAP メッセージ ハンドラをリストします。1 つの WebLogic Web サービスでは、任意数のハンドラ チェーンを定義できます (0 でもかまわない)。

ハンドラ (<handler> 子要素により定義される) がリストされる順番は重要です。デフォルトでは、ハンドラの handleRequest() メソッドは、<handler-chain> 要素の子要素としてリストされている順に実行されます。ハンドラの handleResponse() メソッドは、リストされているものとは逆の順序で実行されます。

属性	説明	データ型	必須 / 任意
name	このハンドラ チェーンの名前。	String	必須

handler-chains

この web-services.xml ファイルで記述された Web サービスに使用されている SOAP メッセージ ハンドラ チェーンを記述する <handler-chain> 要素のリストを含んでいます。1 つの WebLogic Web サービスでは、任意数のハンドラ チェーンを定義できます (0 でもかまわない)。

この要素は、属性を持ちません。

init-param

ハンドラの初期化パラメータのうちの 1 つを表す名前と値の組み合わせを指定します。

属性	説明	データ型	必須 / 任意
name	パラメータの名前。	String	必須
value	パラメータの値。	String	必須

init-params

ハンドラを実装する Java クラスに受け渡される初期化パラメータのリストを含んでいます。

この要素は、属性を持ちません。

java-class

Web サービスの 1 つまたは複数の操作を実装する Java クラス コンポーネントを記述します。

属性	説明	データ型	必須 / 任意
name	このコンポーネントの名前。	String	必須
class-name	このコンポーネントを実装する Java クラスの完全修飾名。	String	必須

jms-receive-queue

Web サービス内の操作のうちの 1 つが JMS キューにマップされることを指定します。この要素は、JMS キューからデータを受信する Web サービス操作の指定に使用されます。

通常は、メッセージ プロデューサがメッセージを特定の JMS キューに入れ、この Web サービス操作を呼び出すクライアントがポーリングを行ってメッセージを受信します。

属性	説明	データ型	必須 / 任意
name	このコンポーネントの名前。	String	必須
connection-factory	WebLogic Server が JMS 接続オブジェクトの作成に使用する JMS 接続ファクトリの JNDI 名。	String	必須
provider-url	WebLogic Server 以外の JMS 実装への接続に使用される URL。	String	任意
initial-context-factory	WebLogic Server 以外の JMS 実装のためのコンテキスト ファクトリ。	String	任意

jms-receive-topic

Web サービス内の操作のうちの 1 つが JMS トピックにマップされることを指定します。この要素は、JMS トピックからデータを受信する Web サービス操作の指定に使用されます。

通常は、メッセージ プロデューサがメッセージを特定の JMS トピックに入れ、この Web サービス コンポーネントを呼び出すクライアントがポーリングを行ってメッセージを受信します。

属性	説明	データ型	必須 / 任意
name	このコンポーネントの名前。	String	必須
connection-factory	WebLogic Server が JMS 接続オブジェクトの作成に使用する JMS 接続ファクトリの JNDI 名。	String	必須
provider-url	WebLogic Server 以外の JMS 実装への接続に使用される URL。	String	任意
initial-context-factory	WebLogic Server 以外の JMS 実装のためのコンテキスト ファクトリ。	String	任意

jms-send-destination

Web サービス内の操作のうちの 1 つが JMS 送り先 (キューまたはトピック) にマップされることを指定します。この要素は、JMS 送り先にデータを送信する Web サービス操作の指定に使用されます。

通常は、メッセージが JMS 送り先に送信された後に、メッセージ コンシューマ (メッセージ駆動型 Bean など) がそのメッセージを使用します。

属性	説明	データ型	必須 / 任意
name	このコンポーネントの名前。	String	必須
connection-factory	WebLogic Server が JMS 接続オブジェクトの作成に使用する JMS 接続ファクトリの JNDI 名。	String	必須
provider-url	WebLogic Server 以外の JMS 実装への接続に使用される URL。	String	任意
initial-context-factory	WebLogic Server 以外の JMS 実装のためのコンテキスト ファクトリ。	String	任意

jndi-name

JNDI ツリーにバインドされているオブジェクトへの参照を指定します。参照の対象は、ステートレス セッション EJB または JMS 送り先になります。

属性	説明	データ型	必須 / 任意
path	JNDI コンテキスト ルートからオブジェクトへのパス名。	String	必須

operation

Web サービスの 1 つの操作をコンフィグレーションします。この要素の属性の値と組み合わせによって、次の種類の操作をコンフィグレーションできます。

- ステートレス セッション EJB または Java クラスのメソッドの呼び出し。この種類の操作を指定するには、component 属性をステートレス セッション EJB または Java クラス コンポーネントの名前に設定し、method 属性をメソッドの名前に設定します。
- JMS バックエンド コンポーネントの呼び出し。この種類の操作を指定するには、component 属性を JMS コンポーネントの名前に設定します。
- ハンドラ チェーン上の SOAP メッセージ ハンドラの連続呼び出しとバックエンド コンポーネントの呼び出し。この種類の操作を指定するには、component 属性をコンポーネントの名前に設定し、handler-chain 属性を対象のハンドラ チェーンの名前に設定します。
- ハンドラ チェーン上の SOAP メッセージ ハンドラの連続呼び出し (ただし、バックエンド コンポーネントは呼び出さない)。この種類の操作を指定する場合は、component 属性と method 属性を設定せずに、handler-chain 属性を対象のハンドラ チェーンの名前に設定します。

操作のパラメータと戻り値を明示的に指定するには、<params> 子要素を使用します。

属性	説明	データ型	必須 / 任意
name	生成された WSDL で使用される操作の名前。 この属性が未指定の場合、操作の名前はメソッド名または SOAP メッセージ ハンドラ チェーン名にデフォルトで設定される。	String	任意
component	この操作を実装するコンポーネントの名前。 この属性の値は、該当する <component> 要素の name 属性に対応する。	String	任意

属性	説明	データ型	必須 / 任意

method	操作を実装する EJB または Java クラスのメソッドの名前 (`component` 属性で操作がステートレス セッション EJB または Java クラスで実装されることを指定した場合)。
	アスタリスク (*) を使用すると、全メソッドを指定できる。
	EJB または Java クラスがメソッドをオーバーロードしない場合は、次のように、メソッド名しか指定する必要がない。
	`method="sell"`
	ただし、EJB または Java クラスがメソッドをオーバーロードする場合は、次のように、全シグネチャを指定する。
	`method="sell(int)"`
		String	任意

handler-chain	この操作を実装する SOAP メッセージ ハンドラ チェーンの名前。 この属性の値は、該当する `<handler-chain>` 要素の `name` 属性に対応する。	String	任意
invocation-style	操作が SOAP 要求の受信と SOAP 応答の送信の両方を行うか、SOAP 応答の返信は行わずに SOAP 要求の受信のみを行うかを指定する。 この属性には、`request-response` (デフォルト値) または `one-way` のいずれかの値のみを指定できる。 **注意 :** この操作を実装するバックエンド コンポーネントが、ステートレス セッション EJB または Java クラスのメソッドである場合にこの属性を `one-way` に設定すると、メソッドにより `void` が返される。	String	任意

属性	説明	データ型	必須 / 任意
portTypeName	この操作が属する WSDL ファイルのポートの種類。ポートの種類から成るカンマ区切りリストを指定することにより、この操作を複数のポートの種類に含めることができる。この Web サービスの WSDL の生成時に、独立した <portType> 要素が指定した個々のポートの種類用に作成される。 デフォルト値は、<web-service> 要素の portType 属性の値。	String	任意

operations

<operations> 要素は、この Web サービスの明示的に宣言された操作をグループ化します。

この要素は、属性を持ちません。

param

<param> 要素は操作の単一パラメータを指定します。

パラメータは、オペレーションを実装するメソッドを定義した順序と同じ順序で指定する必要があります。<param> 要素の数は、メソッドのパラメータ数と一致する必要があります。

属性	説明	データ型	必須 / 任意
name	生成された WSDL で使用される in パラメータの名前。 この属性が未指定の場合、パラメータ名はパラメータのデータ型に基づいて、intvalue1、intvalue2、traderesult などになる。	String	任意
location	in パラメータの値を含む要求 SOAP メッセージの部分 (ヘッダー、本文、または添付ファイル)。 この属性の有効な値は、Body、Header、または attachment。デフォルト値は Body。 Body を指定すると、RPC 操作の呼び出しに関する通常の SOAP 規則に基づき、パラメータの値が SOAP 本文から抽出される。Header を指定すると、type 属性の値を名前に持つ SOAP ヘッダー要素から値が抽出される。 attachment を指定すると、SOAP エンベロープではなく SOAP 添付ファイルからパラメータの値が抽出される。JAX-RPC 仕様で指定されているように、SOAP 添付ファイルから抽出できるのは、以下の Java データ型のみ。 ■ java.awt.Image ■ java.lang.String ■ javax.mail.internet.MimeMultiport ■ javax.xml.transform.Source ■ javax.activation.DataHandler	String	任意

属性	説明	データ型	必須 / 任意
style	in パラメータ (標準入力パラメータ、戻り値として使用される out パラメータ、または値の入出力の両方に使用される inout パラメータ) のスタイル。 この属性の有効な値は、`in`、`out`、または `inout`。 パラメータを `out` または `inout` に指定すると、バックエンド コンポーネントのメソッド内のパラメータの Java クラスが `javax.xml.rpc.holders.Holder` インタフェースを実装する。	String	必須
type	パラメータの XML スキーマデータ型。	NMTOKEN	必須
class-name	パラメータのデータ型の Java 表現の Java クラス名。 この属性が未指定の場合、WebLogic Server は、パラメータの Java クラスの操作を実装するバックエンド コンポーネントを参照する。 この属性の指定が必要となるのは、パラメータの XML 表現と Java 表現間のマッピングをデフォルトとは異なるものにする場合のみである。たとえば、`xsd:int` はデフォルトでは Java プリミティブ `int` 型にマップされるため、これを `java.lang.Integer` にマップするにはこの属性を使用する。	NMTOKEN	通常は必須。属性の説明を参照。

params

<params> 要素は、操作の明示的に宣言されたパラメータとオペレーションの戻り値をグループ化します。

オペレーションのパラメータや戻り値は明示的にリストする必要はありません。<operation> 要素に <params> 子要素がない場合、WebLogic Server はオペレーションを実装するバックエンド コンポーネントを参照してパラメータと戻り値を決定します。Web サービスの WSDL ファイルの生成時に、WebLogic Server は対応するメソッドのパラメータと戻り値の名前を使用します。

以下の場合には、オペレーションのパラメータと戻り値を明示的にリストしてください。

- 生成された WSDL のパラメータおよび戻り値の名前は、そのオペレーションを実装するメソッドの名前とは異なる名前とする必要がある場合。
- パラメータを SOAP の要求ヘッダーまたは応答ヘッダーにある名前にマップする必要がある場合。
- out または inout パラメータを使用する必要がある場合。

操作のパラメータを指定するには、<param> 子要素を使用します。

操作の戻り値を指定するには、<return-param> 子要素を使用します。

<params> 要素は、属性を持ちません。

return-param

<return-param> 要素は、Web サービス操作の戻り値を指定します。

1 つの操作に指定できる <return-param> 要素は 1 つのみです。

属性	説明	データ型	必須 / 任意
name	生成された WSDL ファイルで使用される戻りパラメータの名前。 この属性が未指定の場合、戻りパラメータの名前は result になる。	String	任意
location	戻りパラメータの値を含む応答 SOAP メッセージの部分 (ヘッダーまたは本文)。 この属性の有効な値は、Body または Header。デフォルト値は Body。 Body を指定すると、戻りパラメータの値が SOAP 本文に追加される。Header を指定すると、type 属性の値を名前に持つ SOAP ヘッダー要素として値が追加される。	String	任意
type	戻りパラメータの XML スキーマ データ型。	NMTOKEN	必須

属性	説明	データ型	必須 / 任意
class-name	戻りパラメータのデータ型の Java 表現の Java クラス名。 この属性が未指定の場合、WebLogic Server は操作を実装するバックエンド コンポーネントを参照して戻りパラメータの Java クラスを決定する。 この属性は、次の場合に指定が必要になる。 ■ 操作を実装するバックエンド コンポーネントが `<jms-receive-queue>` または `<jms-receive-topic>` の場合。 ■ 戻りパラメータの XML 表現と Java 表現間のマッピングが不明瞭な場合 (たとえば、`xsd:int` を `int` Java プリミティブ型と `java.lang.Integer` のいずれかにマップする場合など)。	NMTOKEN	通常は必須。属性の説明を参照。

stateless-ejb

Web サービスの 1 つまたは複数の操作を実装するステートレス セッション EJB コンポーネントを指定します。

属性	説明	データ型	必須 / 任意
name	ステートレス EJB コンポーネントの名前。 **注意 :** 名前は `web-services.xml` ファイル内のもので、`ejb-jar.xml` ファイル内の EJB の名前は参照しません。	String	必須

type-mapping

<type-mapping> 要素には、<types> 要素に定義されている XML データ型とその Java 表現間のマッピングのリストが含まれています。

<types> 要素内の各データ型に対し、そのデータ型を実装する Java クラスや、データをシリアライズ / デシリアライズする方法などをリストした <type-mapping-entry> 要素があります。

この要素には属性はありません。

type-mapping-entry

<types> 要素内の単一 XML データ型とその Java 表現間のマッピングを指定します。

属性	説明	データ型	必須 / 任意
class-name	対応する XML データ型にマップされる Java クラスの完全修飾名。	String	必須
element	Java データ型にマップされる XML データ型の名前。データ型の XML スキーマ定義で <element> 要素を使用する場合にのみ指定する。	NMTOKEN	element または type のいずれか 1 つ (両方ではない) が必須。
type	Java データ型にマップされる XML データ型の名前。データ型の XML スキーマ定義で <type> 要素を使用する場合にのみ指定する。	NMTOKEN	element または type のいずれか 1 つ (両方ではない) が必須。

属性	説明	データ型	必須 / 任意
serializer	データを Java から XML に変換する Java クラスの完全修飾名。	String	データ型が、5-13 ページの「組み込みデータ型の使用法」にリストされている WebLogic Web サービス実行時でサポートされている組み込みデータ日付ではない場合にのみ必須。
deserializer	データを XML から Java に変換する Java クラスの完全修飾名。	String	データ型が、5-13 ページの「組み込みデータ型の使用法」にリストされている WebLogic Web サービス実行時でサポートされている組み込みデータ日付ではない場合にのみ必須。

types

XML スキーマ表記法に基づいて、Web サービス オペレーションのパラメータまたは戻り値の型として使用される非組み込みデータ型を記述します。

XML スキーマを使用して非組み込みデータ型の XML 表現を記述する詳細については、http://www.w3.org/TR/xmlschema-0/ を参照してください。

次の例では、`stockSymbol` (String データ型) と `numberTraded` (Integer 型) の 2 つの要素を含む `TradeResult` というデータ型の XML スキーマ宣言を示します。

```
<types>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
              xmlns:stns="java:examples.webservices"
              attributeFormDefault="qualified"
              elementFormDefault="qualified"
              targetNamespace="java:examples.webservices">
      <xsd:complexType name="TradeResult">
          <xsd:sequence>
              <xsd:element maxOccurs="1"
                           name="stockSymbol"
                           type="xsd:string" minOccurs="1">
              </xsd:element>
              <xsd:element maxOccurs="1"
                           name="numberTraded"
                           type="xsd:int"
                           minOccurs="1">
              </xsd:element>
          </xsd:sequence>
      </xsd:complexType>
  </xsd:schema>
</types>
```

web-service

単一 Web サービスを定義します。

Web サービスは次のコンポーネントで定義されます。

- 操作を実装するバックエンド コンポーネント(ステートレス セッション EJB、Java クラス、JMS コンシューマ、JMS プロデューサなど)。
- Web サービス オペレーションのパラメータまたは戻り値として使用される非組み込みデータ型のデータ型宣言の任意セット。
- 非組み込みデータ型のシリアライゼーション クラスと Java クラスを指定する XML/Java データ型マッピングの任意セット。
- Web サービスでサポートされる操作の宣言。

属性	説明	データ型	必須 / 任意
name	Web サービスの名前。	String	必須
targetNamespace	この Web サービスのネームスペース。	String	必須
uri	Web サービス の URI。Web サービスを呼び出す URL で後に使用される。 **注意：** `/TraderService` のように、先頭に「/」を必ず指定すること。	String	必須
protocol	サービスの呼び出しに使用されるプロトコル。 有効な値は `http` または `https`。デフォルトは `http`。	String	任意
exposeHomePage	Web サービスのホーム ページを外部にエクスポーズするかどうかを指定する。 この属性の有効な値は、`True` および `False`。デフォルト値は `True`。つまり、ホーム ページはデフォルトで外部からアクセス可能となる。	Boolean	任意

属性	説明	データ型	必須 / 任意
exposeWSDL	Web サービスの自動生成された WSDL を外部にエクスポーズするかどうかを指定する。 この属性の有効な値は、`True` および `False`。デフォルト値は `True`。つまり、WSDL はデフォルトで外部からアクセス可能となる。	Boolean	任意
style	Web サービスのオペレーションが RPC 指向かドキュメント指向かを指定する。 RPC 指向の WebLogic Web サービス オペレーションでは、SOAP エンコーディングを使用する。ドキュメント指向の WebLogic Web サービス オペレーションでは、リテラル エンコーディングを使用する。 有効な値は `rpc` および `document`。デフォルト値は `rpc`。 **警告：** この属性で `document` を指定した場合、Web サービスのオペレーションを実装するメソッドはすべて、パラメータが 1 つでなければならない。 **注意：** `style` 属性は Web サービス全体に適用されるので、1 つの `<web-service>` 要素で指定されたすべてのオペレーションが RPC 指向またはドキュメント指向のいずれかでなければならない。同じ Web サービスで 2 つのスタイルを混在させることはできない。	String	任意
portName	この Web サービスの動的に生成された WSDL の `<service>` 要素の `<port>` 子要素の名前。 デフォルト値は、この要素の `name` 属性に `Port` を付加した値。たとえば、この Web サービスの名前が `TraderService` の場合、ポート名は `TraderServicePort`。	String	任意

属性	説明	データ型	必須 / 任意
portTypeName	この Web サービスの動的に生成された WSDL で、デフォルトの <portType> 要素の名前。 デフォルト値は、この要素の name 属性に Port を付加した値。たとえば、この Web サービスの名前が TraderService の場合、portType 名は TraderServicePort。	String	任意
ignoreAuthHeader	SOAP リクエストの Authorization HTTP ヘッダが Web サービスで無視されることを指定する。 **注意：** この属性は、慎重に使用する必要がある。この属性の値を True に設定すると、Web サービスを構成する EJB、Web アプリケーション、またはエンタープライズ アプリケーションでアクセス制御のセキュリティ制約が定義されている場合でも、WebLogic Server は Web サービスを呼び出そうとしているクライアント アプリケーションの認証を絶対に行わない。つまり、認証資格を持たないクライアント アプリケーションでも、セキュリティ制約が定義されている Web サービスを呼び出せることになる。 有効な値は True および False。デフォルト値は False。	Boolean	任意

web-services

web-services.xml デプロイメント記述子のルート要素です。

この要素は、属性を持ちません。

B Web サービス Ant タスクとコマンドライン ユーティリティ

この章では、WebLogic Web サービスの Ant タスクと、それらの Ant タスクをベースとしたコマンドライン ユーティリティについて説明します。

WebLogic Web サービス Ant タスクとコマンドライン ユーティリティの概要

Ant は、`make` コマンドに似た Java ベースの構築ツールですが、その機能ははるかに強力です。Ant は、XML ベースのコンフィグレーション ファイル (デフォルトでは `build.xml`) を使用して Java で記述されたタスクを実行します。

BEA は、Web サービスの重要な部分 (シリアライゼーション クラス、クライアント JAR ファイル、`web-services.xml` ファイルなど) の生成や、WebLogic Web サービスのすべてのコンポーネントをデプロイ可能な EAR ファイルにパッケージ化するのに役立つ多数の Ant タスクを提供しています。

Apache Web サイトでは、EAR、WAR、EJB JAR ファイルをパッケージ化するための他の便利な Ant タスクも提供されています。詳細については、http://jakarta.apache.org/ant/manual/ を参照してください。

属性ではなくフラグを使用してユーティリティの動作を指定することにより、一部の Ant タスクをコマンドライン ユーティリティとして実行することも可能です。フラグの説明は、それに対応する属性の説明と完全に同じです。

警告 : Ant タスクのすべての属性が、それに相当するコマンドライン ユーティリティへのフラグとして使用可能なわけではありません。相当するコマンドラインを使用するときは、サポートされているフラグのリストを各 Ant タスクについて説明している節で確認してください。

これらの Ant タスクの使用法の詳しい例と説明については、第 6 章「Ant タスクを使用した WebLogic Web サービスのアセンブル」を参照してください。

Web サービス Ant タスクとコマンドライン ユーティリティのリスト

次の表では、BEA が提供する Web サービス Ant タスクの概要と、それに対応するコマンドライン ユーティリティの名前について説明します。

表 B-1 WebLogic Web サービス Ant タスク

Ant タスク	対応するコマンドライン ユーティリティ	説明
autotype	なし	WebLogic Web サービスに対するパラメータまたは戻り値として使用される非組み込みデータ型のシリアライゼーション クラス、Java 表現、XML スキーマ表現、およびデータ型マッピング情報を生成する。

表 B-1 WebLogic Web サービス Ant タスク

Ant タスク	対応するコマンドライン ユーティリティ	説明
clientgen	weblogic.webservice.clientgen	Web サービスの呼び出しに使用されるシン Java クライアントを含むクライアント JAR ファイルを生成する。
servicegen	weblogic.webservice.servicegen	Web サービスをアセンブルするために必要なすべての手順を実行するメイン Ant タスク。それらの手順は以下のとおり。 ■ Web サービス デプロイメント記述子 (`web-services.xml`) の作成 ■ EJB と Java クラスの参照およびコンポーネントのサポートに必要な非組み込みデータ型の生成 ■ クライアント JAR ファイルの生成 ■ デプロイ可能な EAR ファイルへの全コンポーネントのパッケージ化
source2wsdd	なし	Java クラスを使用して実装される WebLogic Web サービスの Java ソース ファイルから `web-services.xml` デプロイメント記述子ファイルを生成する。
wsdl2Service	なし	WSDL ファイルから WebLogic Web サービスのコンポーネントを生成する。それらのコンポーネントには、`web-services.xml` デプロイメント記述子ファイル、および Web サービスを実装する起点として使用できる Java ソース ファイルが含まれる。
wspackage	なし	WebLogic Web サービスのコンポーネントをデプロイ可能な EAR ファイルにパッケージ化する。
wsgen	なし	6.1 WebLogic Web サービスをバージョン 7.0 の WebLogic Server にアップグレードする。

Web サービス Ant タスクの使用

Ant タスクを使用するには、次の手順に従います。

1. Web サービス Ant タスクへの呼び出しを含む `build.xml` というファイルを作成します。

   次の例は、わかりやすくするために Web サービス Ant タスク `servicegen` と `clientgen` の詳細が省かれた、簡単な `build.xml` ファイルです。

   ```
   <project name="buildWebservice" default="build-ear">
     <target name="build-ear">
       <servicegen attributes go here...>
         ...
       </servicegen>
     </target>
     <target name="build-client" depends="build-ear">
       <clientgen attributes go here .../>
     </target>
     <target name="clean">
       <delete>
         <fileset dir="."
                  includes="example.ear,client.jar" />
       </delete>
     </target>
   </project>
   ```

   `build.xml` ファイルで Ant タスクを指定する例は後の節で紹介します。

2. 環境を設定します。

   Windows NT では、`setWLSEnv.cmd` コマンドを実行します。*`WL_HOME`*`\server\bin` ディレクトリにあります。*`WL_HOME`* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

   UNIX では、`setWLSEnv.sh` コマンドを実行します。*`WL_HOME`*`/server/bin` ディレクトリにあります。*`WL_HOME`* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

3. `build.xml` ファイルと同じディレクトリで `ant` と入力し、Ant タスクまたは `build.xml` ファイルで指定されたタスクを実行します。

   ```
   prompt> ant
   ```

WebLogic Ant タスク用のクラスパスを設定する

各 WebLogic Ant タスクには、新しいディレクトリまたは JAR ファイルを現在の CLASSPATH 環境変数に追加できるように、`classpath` 属性または要素を受け取ります。

次の例では、`servicegen` Ant タスクの `classpath` 属性を使用して CLASSPATH 変数に追加する方法を示します。

```
<servicegen destEar="myEJB.ear"
            classpath="${java.class.path};d:\my_fab_directory"
   ...
</servicegen>
```

次の例では、`<classpath>` 要素を使用して CLASSPATH に追加する方法を示します。

```
<servicegen ...>
   <classpath>
       <pathelement path="${java.class.path}" />
       <pathelement path="d:\my_fab_directory" />
   </classpath>
...
</servicegen>
```

次の例では、WebLogic Web サービス Ant タスク宣言の外で CLASSPATH 変数を構築してから、`<classpath>` 要素を使用してタスク内でその変数を指定する方法を示します。

```
<path id="myid">
   <pathelement path="${java.class.path}"/>
   <pathelement path="${additional.path1}"/>
   <pathelement path="${additional.path2}"/>
</path>

<servicegen ....>
   <classpath refid="myid" />
...
</servicegen>
```

注意： WebLogic Server に付属の Java Ant ユーティリティでは、ANTCLASSPATH 変数を設定した際の *`WL_HOME`*`\server\bin` ディレクトリにある `ant` (UNIX) または `ant.bat` (Windows) コンフィグレーション ファイルを使用します。*`WL_HOME`* は、インストールした WebLogic Platform の最上位ディレクトリです。ANTCLASSPATH 変数を更新する必要がある場合は、使用しているオペレーティング システムの適切なファイルを変更してください。

WSDL および XML スキーマ ファイルを操作する際のオペレーティング システムの大文字 / 小文字の区別の違い

WebLogic Web サービスの Ant タスクの多くには、WSDL や XML スキーマ ファイルなどのオペレーティング システム ファイルを指定するために使用できる属性があります。たとえば、`clientgen` Ant タスクの `wsdl` 属性を使用すると、Web サービスを記述する既存の WSDL ファイルから Web サービス固有のクライアント JAR ファイルを作成できます。

Ant タスクは、大文字と小文字を区別してそれらのファイルを処理します。つまり、たとえば名前が大文字と小文字の点でのみ異なる 2 つの複合型が XML スキーマ ファイルで指定されている場合 (たとえば `MyReturnType` と `MYRETURNTYPE`)、`clientgen` Ant タスクはその複合データ型の Java 表現に対応する 2 つの別々の Java ソース ファイル セットを適切に生成します (`MyReturnType.java` と `MYRETURNTYPE.java`)。

しかし、それらのソース ファイルをそれぞれのクラス ファイルにコンパイルすると、Ant タスクを Microsoft Windows で実行する場合には問題が生じます。なぜなら、Windows は大文字と小文字を区別しないオペレーティング システムだからです。つまり、Windows では `MyReturnType.java` と `MYRETURNTYPE.java` が同じ名前であると判断されるのです。したがって、Windows でこれらのファイルをコンパイルする際には、2 つ目のクラス ファイルが最初のファイルを上書きし、クラス ファイルが 1 つしか残りません。それでも Ant タスクは 2 つのクラスがコンパイルされていると想定しているので、次のようなエラーが生じることになります。

```
c:\src\com\bea\order\MyReturnType.java:14:
class MYRETURNTYPE is public, should be declared in a file named
MYRETURNTYPE.java
public class MYRETURNTYPE
       ^
```

この問題を回避するには、この種の名前の衝突が起こらないように XML スキーマを書き換えます。もしそれが不可能である場合は、Unix などの大文字と小文字を区別するオペレーティング システムで Ant タスクを実行します。

Web サービス コマンド ライン ユーティリティを使用する

Ant タスクに相当するコマンドライン ユーティリティを使用するには、次の手順に従います。

1. コマンド シェル ウィンドウを開きます。
2. 環境を設定します。

   Windows NT では、`setWLSEnv.cmd` コマンドを実行します。*`WL_HOME`*`\server\bin` ディレクトリにあります。*`WL_HOME`* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。

   UNIX では、`setWLSEnv.sh` コマンドを実行します。*`WL_HOME`*`/server/bin` ディレクトリにあります。*`WL_HOME`* は WebLogic プラットフォームがインストールされている最上位ディレクトリです。
3. 次の例に示すように、`java` コマンドを使用してユーティリティを実行します。

```
prompt> java weblogic.webservice.clientgen \
               -ear c:\myapps\myapp.ear \
               -serviceName myService \
               -packageName myservice.client \
                -clientJar c:/myapps/myService_client.jar
```

   使い方メッセージを表示するには、引数なしでコマンドを実行します。

autotype

`autotype` Ant タスクは、Web サービス オペレーションのパラメータまたは戻り値として使用される非組み込みデータ型の以下のコンポーネントを生成します。

- データを XML 表現と Java 表現間で変換するシリアライゼーション クラス
- 入力が XML スキーマまたは WSDL ファイルの場合は、データ型の Java 表現が含まれる Java クラス
- 入力が非組み込みデータ型を表す Java クラスの場合は、データ型の XML スキーマ表現

- web-services.xml デプロイメント記述子ファイルに格納されるデータ型マッピング情報

autotype でデータ型コンポーネントを生成できる非組み込みデータ型のリストについては、6-14 ページの「servicegen および autotype Ant タスクでサポートされる非組み込みデータ型」を参照してください。

autotype Ant タスクでは、以下のいずれかの入力を指定できます。

- 非組み込みデータ型を表す Java クラス ファイル (javaTypes 属性を指定)。autotype Ant タスクは、web-services.xml ファイル用に、対応する XML スキーマ、シリアライザ クラス、およびデータ型マッピング情報を生成します。
- ステートレス セッション EJB などのバックエンド コンポーネントを格納する Java クラス ファイル (javaComponents 属性を指定)。autotype Ant タスクは、コンポーネントで使用される非組み込みデータ型を探し、それから web-services.xml ファイル用に、対応する XML スキーマ、シリアライザ クラス、およびデータ型マッピング情報を生成します。
- 非組み込みデータ型を表す XML スキーマ ファイル (schemaFile 属性を指定)。autotype Ant タスクは、web-services.xml ファイル用に、対応する Java 表現、シリアライザ クラス、およびデータ型マッピング情報を生成します。
- 非組み込みデータ型の記述を格納する WSDL ファイルの URL (wsdlURI 属性を指定)。autotype Ant タスクは、web-services.xml ファイル用に、対応する Java 表現、シリアライザ クラス、およびデータ型マッピング情報を生成します。

destDir 属性を使用すると、生成されたコンポーネントが格納されるディレクトリの名前を指定できます。XML スキーマとデータ型マッピング情報は、types.xml というファイルに生成されます。このファイルは、既存の web-services.xml ファイルを非組み込みデータ型マッピング情報で手動更新するために使用するか、あるいは servicegen または clientgen Ant タスクの typeMappingFile 属性、または source2wsdd Ant タスクの typesInfo 属性と関連して使用できます。

警告 : autotype、servicegen、および clientgen Ant タスクによって生成されたシリアライザ クラスと Java および XML 表現はラウンドトリップできません。詳細については、6-18 ページの「生成されたデータ型コンポーネントの非ラウンドトリップ」を参照してください。

注意： autotype Ant タスクの完全修飾名は、weblogic.ant.taskdefs.webservices.javaschema.JavaSchema です。

例

次の例は、Java クラス mypackage.MyType の非組み込みデータ型コンポーネントを作成する方法を示しています。

```
<autotype  javaTypes="mypackage.MyType"
           targetNamespace="http://www.foobar.com/autotyper"
           packageName="a.package.name"
           destDir="d:\output" />
```

次の例は、上の例に類似していますが、mypackage.MyType Java データ型の配列の非組み込みデータ型コンポーネントを作成する点が異なります。

```
<autotype  javaTypes="[Lmypackage.MyType;"
           targetNamespace="http://www.foobar.com/autotyper"
           packageName="a.package.name"
           destDir="d:\output" />
```

注意： [L*classname*; 構文は、java.lang.Class.getName() メソッドのドキュメントで概説されている Java クラスの命名規約に従っています。

次の例は、WSDL ファイルに対して autotype Ant タスクを使用する方法を示しています。

```
<autotype  wsdl="file:\wsdls\myWSDL"
           targetNamespace="http://www.foobar.com/autotyper"
           packageName="a.package.name"
           destDir="d:\output" />
```

属性

次の表では、autotype Ant タスクの属性について説明します。

属性	説明	必須 / 任意
schemaFile	非組み込みデータ型の XML スキーマ表現が格納されているファイルの名前。	schemaFile、wsdl、javaTypes、または javaComponents のいずれかの属性を 1 つだけ指定する必要がある。
wsdl	非組み込みデータ型の XML スキーマ表現が格納されている WSDL の完全パス名または URI。	schemaFile、wsdl、javaTypes、または javaComponents のいずれかの属性を 1 つだけ指定する必要がある。
javaTypes	非組み込みデータ型を表す Java クラス名のカンマ区切りリスト。Java クラスがコンパイルされ、CLASSPATH 内に存在する必要がある。 たとえば、次のように指定する。 `javaTypes="my.class1,my.class2"` **注意：** Java データ型の配列を指定するには、構文 `[Lclassname;` を使用する。例については、B-9 ページの「例」を参照。	schemaFile、wsdl、javaTypes、または javaComponents のいずれかの属性を 1 つだけ指定する必要がある。

属性	説明	必須 / 任意
javaComponents	Web サービス オペレーションを実装する Java クラス名のカンマ区切りリスト。Java クラスがコンパイルされ、CLASSPATH 内に存在する必要がある。 たとえば、次のように指定する。 `javaComponents="my.class1,my.class2"` `autotype` Ant タスクは、リストの Java クラスを参照し、見つかったすべての非組み込みデータ型のコンポーネントを自動的に生成する。	`schemaFile`、`wsdl`、`javaTypes`、または `javaComponents` のいずれかの属性を 1 つだけ指定する必要がある。
destDir	生成されたコンポーネントが格納されるディレクトリの完全パス名。XML スキーマとデータ型マッピング情報は、`types.xml` というファイルに生成される。	必須
typeMappingFile	必要なコンポーネントが既に生成されている非組み込みデータ型のデータ型マッピング情報が格納されているファイル。情報の形式は、`web-services.xml` ファイルの `<type-mapping>` 要素内のデータ型マッピング情報と同じ。 `autotype` Ant タスクは、このファイルにあるデータ型の非組み込みデータ型コンポーネントは生成しない。	任意
packageBase	Web サービスで戻り値またはパラメータとして使用される任意の非組み込みデータ型用に生成された Java クラスの基本パッケージ名。つまり、`autotype` Ant タスクは、指定されたパッケージの基本名に付加された個々の Java クラスに固有の名前を生成するが、生成された個々の Java クラスは同じパッケージ名の一部となる。 この属性を指定しない場合は、`autotype` Ant タスクが基本パッケージ名を生成する。 **注意：** BEA では、この属性よりも、`packageName` 属性を使用して完全パッケージ名を指定することを推奨。`packageBase` 属性は、JAX-RPC に準拠している。	任意 この属性を指定した場合、`packageName` を同時に指定することはできない。

属性	説明	必須 / 任意
packageName	Web サービスで戻り値またはパラメータとして使用される任意の非組み込みデータ型用に生成された Java クラスの完全パッケージ名。 この属性を指定しない場合は、`autotype` Ant タスクがパッケージ名を生成する。 **注意：** この属性は必須でないが、ほとんどの場合には指定することをお勧めする。 現時点で、この属性を指定すべきでないのは、`javaTypes` 属性を使用して、クラス名は同じだがパッケージ名が異なる Java データ型のリストを指定する場合のみである。この場合に `packageName` 属性も指定すると、`autotype` Ant タスクは最後のクラスでしかシリアライゼーション クラスを生成しない。	任意 この属性を指定した場合、`packageBase` を同時に指定することはできない。
targetNamespace	Web サービスのネームスペース URI。	必須

clientgen

`clientgen` Ant タスクは、クライアント アプリケーションが WebLogic Web サービスと非 WebLogic Web サービスの両方の呼び出しに使用できる Web サービス固有のクライアント JAR ファイルを生成します。通常、`clientgen` Ant タスクは既存の WSDL ファイルからクライアント JAR ファイルを生成するために使用します。ただし、WebLogic Web サービスの実装が含まれた EAR ファイルで使用することも可能です。

クライアント JAR ファイルの内容は次のとおりです。

- 静的モードでの Web サービスの呼び出しに使用されるクライアント インタフェースとスタブ ファイル (JAX-RPC 仕様に準拠)。
- 非組み込みデータをその XML 表現と Java 表現間で変換する省略可能なシリアライゼーション クラス。
- Web サービス WSDL ファイルの省略可能なクライアントサイドのコピー。

clientgen Ant タスクを使用すると、クライアント JAR ファイルを既存の Web サービス (WebLogic Server 上で実行されている必要はない) の WSDL ファイルから、または WebLogic Web サービス実装を含んでいる EAR ファイルから生成できます。

WebLogic Server 配布キットには、WebLogic Web サービス実行時コンポーネントのサポートに必要なクライアント サイド クラスを含んでいるクライアント実行時 JAR ファイルが含まれています。詳細については、8-5 ページの「Java クライアント アプリケーション JAR ファイルの取得」を参照してください。

警告： autotype、servicegen、および clientgen Ant タスクによって生成されたシリアライザ クラスと Java および XML 表現はラウンドトリップできません。詳細については、6-18 ページの「生成されたデータ型コンポーネントの非ラウンドトリップ」を参照してください。

警告： clientgen Ant タスクは要求 / 応答または通知 WSDL 操作をサポートしていません。つまり、これらの種類の操作を含む WSDL ファイルからクライアント JAR ファイルを作成しようとすると、Ant タスクは操作を無視します。

注意： clientgen Ant タスクの完全修飾名は、weblogic.ant.taskdefs.webservices.clientgen.ClientGenTask です。

例

```
<clientgen wsdl="http://example.com/myapp/myservice.wsdl"
           packageName="myapp.myservice.client"
           clientJar="c:/myapps/myService_client.jar"
/>
```

属性

次の表では、clientgen Ant タスクの属性について説明します。

属性	説明	必須 / 任意
wsdl	クライアント JAR ファイルの生成が必要な (WebLogic または非 WebLogic) Web サービスを記述する WSDL の完全パス名または URL。 クライアント JAR ファイル内の生成されたスタブ ファクトリ クラスは、デフォルト コンストラクタ内のこの属性の値を使用する。	wsdl または ear の指定が必須。
ear	クライアント JAR ファイルの生成が必要な WebLogic Web サービス実装を含んでいる EAR ファイルまたは展開ディレクトリの名前。 **注意：** clientgen の saveWSDL 属性が True (デフォルト値) に設定されている場合、clientgen Ant タスクは EAR ファイルの情報から WSDL ファイルを生成し、生成された JAR ファイルに格納する。clientgen は Web サービスをホストする WebLogic Server インスタンスのホスト名またはポート番号を認識しないため、clientgen は生成された WSDL の次のエンドポイント アドレスを使用する。 `http://localhost:7001/`*`contextURI`*`/`*`serviceURI`* *contextURI* と *serviceURI* は、8-23 ページの「WebLogic Web サービスのホーム ページおよび WSDL の URL」で説明した値と同じ。このエンドポイント アドレスが正しくなく、クライアント アプリケーションがクライアント JAR ファイルに格納されている WSDL ファイルを使用する場合、正確なエンドポイント アドレスを指定して手動で WSDL ファイルを更新する必要がある。	wsdl または ear の指定が必須。

属性	説明	必須 / 任意
warName	Web サービスを含んでいる WAR ファイルの名前。 デフォルト値は `web-services.war`。	任意 この属性は、`ear` 属性との組み合わせでのみ指定できる。
serviceName	対応するクライアント JAR ファイルの生成が必要な Web サービス名。 `wsdl` 属性を指定すると、Web サービス名は WSDL ファイル内の `<service>` 要素に対応する。`ear` 属性を指定すると、Web サービス名は `web-services.xml` デプロイメント記述子ファイル内の `<web-service>` 要素に対応する。 `serviceName` 属性が未指定の場合、`clientgen` タスクは WSDL または `web-services.xml` ファイル内で最初に見つかったサービス名のクライアント クラスを生成する。	任意
typeMappingFile	データ型マッピング情報を含むファイルで、JAX-RPC スタブの生成時に `clientgen` タスクにより使用される。情報の形式は、`web-services.xml` ファイルの `<type-mapping>` 要素内のデータ型マッピング情報と同じ。 `ear` 属性を指定すると、このファイル内の情報により、`web-services.xml` ファイル内のデータ型マッピング情報がオーバーライドされる。	任意
packageName	生成された JAX-RPC クライアント インタフェースとスタブ ファイルをパッケージ化するパッケージ名。	必須
autotype	`clientgen` タスクが、Web サービス オペレーションへのパラメータまたは戻り値として使用される非組み込みデータ型に対するシリアライゼーション クラスを生成してクライアント JAR ファイルに入れるかを指定する。 有効な値は `True` および `False`。デフォルト値は `True`。	任意

属性	説明	必須 / 任意
clientJar	生成されたクライアント インタフェース クラス、スタブ クラス、省略可能なシリアライゼーション クラスなどを clientgen タスクが入れる JAR ファイルまたは展開ディレクトリの名前。 JAR ファイルを作成または更新するには、JAR ファイルの指定時に .jar サフィックスを使用する (myclientjar.jar など)。属性値に .jar サフィックスがない場合、clientgen タスクはディレクトリ名が参照されていると想定する。 存在しない JAR ファイルまたはディレクトリを指定すると、clientgen タスクが新しい JAR ファイルまたはディレクトリを作成する。	必須
overwrite	既存のクライアント JAR ファイルを上書きするかを指定する。 有効な値は True および False。デフォルト値は True。	任意
useServerTypes	clientgen タスクが、Web サービスで使用されるいずれかの非組み込み Java データ型の実装を取得する場所を指定する。タスクは Java コードを生成するか、Web サービスの完全な実装を含む EAR ファイルからそれを取得する。 有効な値は True (EAR ファイル内の Java コードを使用) および False。デフォルト値は False。 clientgen でデータ型コンポーネントを生成できる非組み込みデータ型のリストについては、6-14 ページの「servicegen および autotype Ant タスクでサポートされる非組み込みデータ型」 を参照。	任意 ear 属性との組み合わせでのみ使用。
saveWSDL	True に設定されている場合、生成されたクライアント JAR ファイルに Web サービスの WSDL を保存するよう指定する。この場合、クライアント アプリケーションは Web サービスへのスタブを作成するたびに WSDL をダウンロードする必要がなく、ネットワーク使用量が削減されるためにパフォーマンスが一般に向上する。 有効な値は True および False。デフォルト値は True。	任意

属性	説明	必須 / 任意
j2me	clientgen Ant タスクが J2ME/CDC 準拠のクライアント JAR ファイルを作成するかどうかを指定する。 **注意：** 生成されたクライアント コードは JAX-RPC に準拠していない。 有効な値は True および False。デフォルト値は False。	任意
useLowerCaseMethodNames	True に設定されている場合、生成されたスタブのメソッド名の 1 文字目を小文字にするよう指定する。False に設定した場合、メソッド名はすべて WSDL ファイルのオペレーション名と同じになる。 有効な値は True および False。デフォルト値は True。	任意
typePackageName	Web サービスで戻り値またはパラメータとして使用される任意の非組み込みデータ型用に生成された Java クラスの完全パッケージ名を指定する。 この属性を指定した場合、typePackageBase を同時に指定することはできない。 この属性を指定しておらず、かつ WSDL ファイル内の XML スキーマがターゲット ネームスペースを定義している場合、clientgen Ant タスクが、ターゲット ネームスペースに基づき、パッケージ名を生成する。つまり、XML スキーマがターゲット ネームスペースを定義していない場合は、clientgen Ant タスクの typePackageName (推奨) 属性、または typePackageBase 属性のいずれかを指定する必要がある。 **注意：** この属性は必須でないが、指定することを推奨。	wsdl 属性を指定しており、WSDL ファイル内の XML スキーマでターゲット ネームスペースが定義されていない場合のみ、必須。

属性	説明	必須 / 任意
typePackageBase	Web サービスで戻り値またはパラメータとして使用される任意の非組み込みデータ型用に生成された Java クラスの基本パッケージ名を指定する。つまり、clientgen Ant タスクは、指定されたパッケージの基本名に付加された個々の Java クラスに固有の名前を生成するが、生成された個々の Java クラスは同じパッケージ名の一部となる。 この属性を指定した場合、typePackageName を同時に指定することはできない。 この属性を指定しておらず、かつ WSDL ファイル内の XML スキーマがターゲット ネームスペースを定義している場合、clientgen Ant タスクが、ターゲット ネームスペースに基づき、パッケージ名を生成する。つまり、XML スキーマがターゲット ネームスペースを定義していない場合は、clientgen Ant タスクの typePackageName (推奨) 属性、または typePackageBase 属性のいずれかを指定する必要がある。 **注意：** この属性を使用するよりも、typePackageName 属性で完全パッケージ名を指定する方が適切。typePackageBase 属性は、JAX-RPC に準拠している。	wsdl 属性を指定しており、WSDL ファイル内の XML スキーマでターゲット ネームスペースが定義されていない場合のみ、必須。
usePortNameAsMethodName	clientgen Ant タスクが、WSDL ファイルからクライアントを生成する際に操作の名前を取得する場所を指定する。 この値を True に設定した場合、操作は、WSDL ファイルの <port> 要素の name 属性で指定された名前を取る (<port> は <service> 要素の子要素)。この値を False に設定した場合、操作は、WSDL ファイルの <portType> 要素の name 属性で指定された名前を取る (<portType> は <definitions> 要素の子要素)。 有効な値は True および False。デフォルト値は False。	任意

相当するコマンドライン ユーティリティ

clientgen Ant タスクに相当するコマンドライン ユーティリティは weblogic.webservice.clientgen です。ユーティリティのフラグの説明は、前の節で説明した Ant タスク属性の説明と同じです。

weblogic.webservice.clientgen ユーティリティは次のフラグをサポートしています (フラグの説明は該当の属性を参照)。

- -wsdl *uri*
- -ear *pathname*
- -clientJar *pathname*
- -packageName *name*
- -warName *name*
- -serviceName *name*
- -typeMappings *pathname*
- useServerTypes

servicegen

servicegen Ant タスクは、EJB JAR ファイルまたは Java クラスのリストを入力として受け取り、必要な全 Web サービス コンポーネントを作成し、このコンポーネントをデプロイ可能な EAR ファイルにパッケージ化します。

具体的には、servicegen Ant タスクは、次のタスクを実行します。

- EJB と Java クラスを参照し、Web サービス オペレーションに変換するパブリック メソッドを検索します。
- servicegen Ant タスクの属性と参照した情報に基づいて、web-services.xml デプロイメント記述子ファイルを作成します。
- 必要であれば、非組み込みのデータ型を XML 表現から Java 表現へ、またはその逆へ変換するシリアライゼーション クラスを作成します。また、Java オブジェクトの XML スキーマ表現を作成し、それに対応して

web-services.xml ファイルを更新します。この機能は、オートタイピングと呼ばれます。

- Web サービスを呼び出すクライアント アプリケーションが必要とする Web サービス固有のクラス、スタブ、およびインタフェースを含むクライアント JAR ファイルを必要に応じて作成します。クラス、スタブ、インタフェースは JAX-RPC API に基づいています。
- Web アプリケーション WAR ファイルにすべての Web サービス コンポーネントをパッケージ化し、次に、WAR ファイルと EJB JAR ファイルをデプロイ可能な EAR ファイルにパッケージ化します。

警告： autotype、servicegen、および clientgen Ant タスクによって生成されたシリアライザ クラスと Java および XML 表現はラウンドトリップできません。詳細については、6-18 ページの「生成されたデータ型コンポーネントの非ラウンドトリップ」を参照してください。

注意： servicegen Ant タスクの完全修飾名は、weblogic.ant.taskdefs.webservices.servicegen.ServiceGenTask です。

例

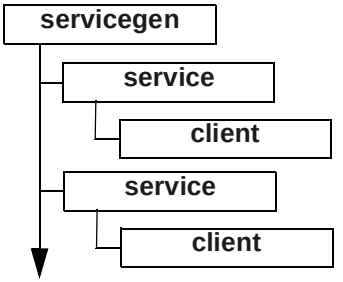
```
<servicegen
   destEar="c:\myWebService.ear"
   warName="myWAR.war"
   contextURI="web_services" >
   <service
       ejbJar="c:\myEJB.jar"
       targetNamespace="http://www.bea.com/examples/Trader"
       serviceName="TraderService"
       serviceURI="/TraderService"
       generateTypes="True"
       expandMethods="True" >
       </service>
</servicegen>
```

属性と子要素

1 つの EAR ファイルに定義する各 Web サービスに対して、servicegen Ant タスクには 4 つの属性と 1 つの子要素 (<service>) があります。1 つ以上の <service> 要素を指定する必要があります。

<service> 要素には、省略可能な <client> 要素が 1 つあります。

次の図は、servicegen Ant タスクの階層を示しています。



servicegen

servicegen Ant タスクは、Web サービスの全コンポーネントを自動的に生成およびアセンブルしてデプロイ可能な EAR ファイルにパッケージ化するメイン タスク です。

次の表では、servicegen Ant タスクの属性について説明します。

属性	説明	必須 / 任意
destEar	Web サービスとその全コンポーネントを含む EAR ファイルまたは展開ディレクトリのパス名。 EAR ファイルを作成または更新するには、EAR ファイルの指定時に .ear サフィックスを使用する (c:\mywebservice.ear など)。属性値に .ear サフィックスがない場合、servicegen タスクは展開ディレクトリを作成する。 存在しない EAR ファイルまたはディレクトリを指定すると、servicegen タスクが新しいファイルまたはディレクトリを作成する。	必須
overwrite	既存の EAR ファイルまたはディレクトリのコンポーネントを上書きするかを指定する。コンポーネントとしては、web-services.xml ファイル、シリアライゼーション クラス、クライアント JAR ファイルなどがある。 この属性の有効な値は、True および False。デフォルト値は True。 False を指定すると、servicegen Ant タスクは EAR ファイル / ディレクトリの内容と情報を web-services.xml ファイルにマージしようとする。	任意
warName	Web サービスの Web アプリケーションが書き込まれる WAR ファイルまたは展開ディレクトリの名前。WAR ファイルまたはディレクトリは EAR ファイルの最上位に作成される。 デフォルト値は web-services.war という WAR ファイル。 WAR ファイルを指定するには、.war サフィックスを使用する (mywebserviceWAR.war など)。属性値に .war サフィックスがない場合、servicegen タスクは展開ディレクトリを作成する。	任意

属性	説明	必須 / 任意
contextURI	Web サービスのコンテキスト ルート。この値は、Web サービスを呼び出す URL 内に使用する。 contextURI 属性のデフォルト値は warName 属性の値になる。	任意

service

<service> 要素は、ステートレス セッション EJB または Java クラスで実装される単一 Web サービスを記述します。

次の表では、servicegen Ant タスクの <service> 要素の属性について説明します。単一 EAR ファイルにパッケージ化する各 Web サービスに対して、1 つの <service> 要素を含めてください。

属性	説明	必須 / 任意
ejbJar	Web サービス オペレーションのバックエンド コンポーネントを実装する EJB を含む JAR ファイルまたは展開ディレクトリ。servicegen Ant タスクはこの EJB を参照して全コンポーネントを自動生成する。	ejbJar 属性、javaClassComponents 属性、または JMS* 属性の指定が必須。
javaClassComponents	Web サービス オペレーションを実装する Java クラス名のカンマ区切りリスト。Java クラスがコンパイルされ、CLASSPATH 内に存在する必要がある。 たとえば、次のように指定する。 javaClassComponents="my.FirstClass,my.SecondClass" **注意：** クラス名を指定するときには、拡張子 .class を含めない。 servicegen Ant タスクはこの Java クラス を参照して必要な全コンポーネントを自動生成する。	ejbJar 属性、javaClassComponents 属性、または JMS* 属性の指定が必須。

属性	説明	必須 / 任意
ignoreAuthHeader	SOAP リクエストの `Authorization` HTTP ヘッダが Web サービスで無視されることを指定する。 **注意：** この属性は、慎重に使用する必要がある。この属性の値を `True` に設定すると、Web サービスを構成する EJB、Web アプリケーション、またはエンタープライズ アプリケーションでアクセス制御のセキュリティ制約が定義されている場合でも、WebLogic Server は Web サービスを呼び出そうとしているクライアント アプリケーションの認証を絶対に行わない。つまり、認証資格を持たないクライアント アプリケーションでも、セキュリティ制約が定義されている Web サービスを呼び出せることになる。 有効な値は `True` および `False`。デフォルト値は `False`。	任意
includeEJBs	非組み込みデータ型コンポーネントの生成が必要な EJB 名のカンマ区切りリスト。 この属性を指定すると、`servicegen` タスクはこのリスト上の EJB のみを処理する。 EJB 名は、`ejbJar` 属性で指定される EJB JAR ファイル内の `ejb-jar.xml` デプロイメント記述子内の `<ejb-name>` 要素に対応する。	任意 `ejbJar` 属性との組み合わせでのみ使用される。
excludeEJBs	非組み込みデータ型コンポーネントを生成すべきでない EJB 名のカンマ区切りリスト。 この属性を指定すると、`servicegen` タスクはこのリスト上の EJB 以外の全 EJB を処理する。 EJB 名は、`ejbJar` 属性で指定される EJB JAR ファイル内の `ejb-jar.xml` デプロイメント記述子内の `<ejb-name>` 要素に対応する。	任意 `ejbJar` 属性との組み合わせでのみ使用される。

属性	説明	必須 / 任意
serviceName	WSDL でパブリッシュされる Web サービスの名前。 **注意：** `servicegen` を呼び出す `build.xml` ファイルに複数の `<service>` 要素を指定し、各要素の `serviceName` 属性を同じ値に設定すると、`servicegen` は複数の `<service>` 要素を単一 Web サービスにマージしようとする。	必須
serviceURI	クライアント アプリケーションが Web サービスの呼び出しに使用する URL の Web サービス URI 部分。 **注意：** `/TraderService` のように、先頭に「/」を必ず指定すること。 Web サービスを呼び出す完全 URL は次のとおり。 *`protocol://host:port/contextURI/serviceURI`* 各要素の説明は次のとおり。 ■ *`protocol`* は、`<service>` 要素の `protocol` 属性を指す。 ■ *`host`* は、WebLogic Server が動作しているコンピュータの名前を指す。 ■ *`port`* は、WebLogic Server がリスンしているポートを指す。 ■ *`contextURI`* は、メイン `servicegen` Ant タスクの `contextURI` 属性を指す。 ■ *`serviceURI`* は、この属性を指す。	必須
targetNamespace	Web サービスのネームスペース URI。	必須
protocol	この Web サービスのデプロイに使用されるプロトコル。 有効な値は `http` および `https`。デフォルト値は `http`。	任意

属性	説明	必須 / 任意
expandMethods	web-services.xml ファイルの生成時に、servicegen タスクが EJB または Java クラスの各メソッドに別々の <operation> 要素を作成するか、または method="*" 属性を含む 1 つの <operation> 要素のみを指定することで全メソッドを暗黙的に参照するかを指定する。 有効な値は True および False。デフォルト値は False。	任意
generateTypes	servicegen タスクが、パラメータまたは戻り値として使用される非組み込みデータ型に対してシリアライゼーション クラスと Java 表現を生成するかを指定する。 有効な値は True および False。デフォルト値は True。 servicegen でデータ型コンポーネントを生成できる非組み込みデータ型のリストについては、6-14 ページの「servicegen および autotype Ant タスクでサポートされる非組み込みデータ型」を参照。	任意
typeMappingFile	XML データ型のマッピングに関する追加情報を含むファイル。情報の形式は、web-services.xml 内のデータ型マッピング情報と同じ。 この属性は、操作を実装する EJB または Java クラスが使用する必須のデータ型の XML 記述だけでなく、追加の XML データ型情報も web-services.xml ファイルの <type-mapping> 要素に含める場合に使用する。servicegen タスクは、指定ファイル内の追加情報を、生成された web-services.xml ファイルに追加する。	任意

属性	説明	必須 / 任意
style	servicegen Ant タスクで RPC 指向またはドキュメント指向どちらの Web サービス オペレーションを生成するのかを指定する。 RPC 指向の WebLogic Web サービス オペレーションでは、SOAP エンコーディングを使用する。ドキュメント指向の WebLogic Web サービス オペレーションでは、リテラル エンコーディングを使用する。 この属性で document を指定した場合、生成される Web サービスのオペレーションを実装するメソッドは、パラメータが 1 つでなければならない。servicegen が複数のパラメータを持つメソッドに遭遇した場合、servicegen はそのメソッドを無視し、対応する Web サービス オペレーションが生成されない。 この属性の有効な値は、rpc および document。デフォルト値は rpc。 **注意：** style 属性は Web サービス全体に適用されるので、1 つの WebLogic Web サービスのすべてのオペレーションが RPC 指向またはドキュメント指向のいずれかでなければならない。同じ Web サービスで 2 つのスタイルを混在させることはできない。	任意
JMSDestination	JMS トピックまたはキューの JNDI 名。	必須 (JMS で実装される Web サービスを作成する場合)
JMSDestinationType	JMS 送り先のタイプ (キューまたはトピック)。 有効な値は topic または queue。	必須 (JMS で実装される Web サービスを作成する場合)

属性	説明	必須 / 任意
JMSAction	JMS で実装されるこの Web サービスを呼び出すクライアント アプリケーションが JMS 送り先にメッセージを送信するのか、それとも JMS 送り先からメッセージを受信するのかを指定する。 有効な値は `send` または `receive`。 クライアントが JMS 送り先にメッセージを送信する場合は `send`、クライアントが JMS 送り先からメッセージを受信する場合は `receive` を指定する。	必須 (JMS で実装される Web サービスを作成する場合)
JMSConnectionFactory	JMS の送り先への接続を作成するために使用される `ConnectionFactory` の JNDI 名。	必須 (JMS で実装される Web サービスを作成する場合)
JMSOperationName	生成された WSDL ファイルのオペレーションの名前。 デフォルト値は `send` または `receive` のいずれか (JMSAction 属性の値によって異なる)。	任意
JMSMessageType	送信または受信オペレーションの 1 つのパラメータのデータ型。 デフォルト値は `java.lang.String`。 この属性を使用して非組み込みデータ型を指定し、`generateTypes` 属性を `True` に設定する場合は、この非組み込みデータ型の Java 表現が必ず CLASSPATH で設定されているようにする。	任意

client

省略可能な `<client>` 要素は、クライアント アプリケーションが Web サービスの呼び出しに使用するクライアント JAR ファイルの作成方法を指定します。`servicegen` Ant タスクによりクライアント JAR ファイルを作成する場合にのみこの要素を指定します。

注意 : Web サービスのアセンブル時にクライアント JAR ファイルを作成する必要はありません。後で `clientgen` Ant タスクを使用して JAR ファイルを生成することができます。

次の表では、<client> 要素の属性について説明します。

属性	説明	必須 / 任意
clientJarName	生成されたクライアント JAR ファイルの名前。 servicegen タスクは、Web サービスをパッケージ化するときに、クライアント JAR ファイルを EAR ファイルの Web サービス WAR ファイルの最上位ディレクトリに入れる。 デフォルト名は *serviceName*_client.jar で、*serviceName* は Web サービスの名前 (serviceName 属性) を指す。 **注意 :** クライアント JAR ファイルへのリンクを Web サービス ホーム ページに自動的に表示する場合は、デフォルト名を変更しないこと。	任意
packageName	生成されたクライアント インタフェースとスタブ ファイルをパッケージ化するパッケージ名。	必須
useServerTypes	servicegen タスクが、Web サービスで使用されるいずれかの非組み込み Java データ型の実装を取得する場所を指定する。タスクは Java コードを生成するか、Web サービスの完全な実装を含む EAR ファイルからそれを取得する。 有効な値は True (EAR ファイル内の Java コードを使用) および False。デフォルト値は False。 servicegen でデータ型コンポーネントを生成できる非組み込みデータ型のリストについては、6-14 ページの「servicegen および autotype Ant タスクでサポートされる非組み込みデータ型」を参照。	任意
saveWSDL	True に設定されている場合、生成されたクライアント JAR ファイルに Web サービスの WSDL ファイルが保存される。この場合、クライアント アプリケーションは Web サービスへのスタブを作成するたびに WSDL ファイルをダウンロードする必要がなく、ネットワーク使用量が削減されるためにパフォーマンスが一般に向上する。 有効な値は True および False。デフォルト値は True。	任意

相当するコマンドライン ユーティリティ

servicegen Ant タスクに相当するコマンドライン ユーティリティは weblogic.webservice.servicegen です。ユーティリティのフラグの説明は、前の節で説明した Ant タスク属性の説明と同じです。

警告： weblogic.webservice.servicegen コマンドライン ユーティリティを使用して Web サービスを自動的にアセンブルする場合、web-services.xml ファイルに作成できる Web サービスは 1 つのみです。

weblogic.webservice.servicegen ユーティリティは次のフラグをサポートしています (フラグの説明は該当する属性を参照)。

- -destEar *pathname*
- -warName *name*
- -ejbJar *pathname*
- -javaClassComponents *list_of_classnames*
- -serviceName *name*
- -serviceURI *uri*
- -targetNamespace *uri*
- -protocol *protocol*
- -expandMethods
- -clientPackageName *name*
- -clientJarName *name*

source2wsdd

source2wsdd Ant タスクは、Java クラスを使用して実装される WebLogic Web サービスの Java ソース ファイルから web-services.xml デプロイメント記述子ファイルを生成します。

source2wsdd Ant タスクは、Java クラスのメソッドのパラメータまたは戻り値として使用される非組み込みデータ型のデータ型マッピング情報を生成しません。Java クラスで非組み込みデータ型が使用される場合は、まず autotype Ant タスクを実行して必要なコンポーネントを生成し、その後に source2wsdd Ant タスクの typesInfo 属性を、autotype Ant タスクで生成された types.xml ファイルに設定します。

Java クラスが他の Java クラス ファイルを参照する場合は、必ず、sourcePath 属性をそれらのファイルが格納されているディレクトリに設定してください。

注意： source2wsdd Ant タスクの完全修飾名は、weblogic.ant.taskdefs.webservices.autotype.JavaSource2DD です。

例

```
<source2wsdd
      javaSource="c:\source\MyService.java"
      typesInfo="c:\autotype\types.xml"
      ddFile="c:\ddfiles\web-services.xml"
      serviceURI="/MyService"
/>
```

属性

次の表では、source2wsdd Ant タスクの属性について説明します。

属性	説明	必須 / 任意
javaSource	Web サービスのコンポーネントを実装する Java ソース ファイルの名前。	必須
ddFile	生成されたデプロイメント記述子情報が格納される Web サービスのデプロイメント記述子ファイル (web-services.xml) の完全パス名。	必須

属性	説明	必須 / 任意
typesInfo	Web サービスのパラメータまたは戻り値として使用される非組み込みデータ型の XML スキーマ表現およびデータ型マッピング情報を格納するファイルの名前。 データ型マッピング情報の形式は、`web-services.xml` ファイルの `<type-mapping>` 要素内のそれと同じ。 通常は、既に `autotype` Ant タスクを実行し、この情報が `types.xml` というファイルに生成されている。	必須
serviceURI	クライアント アプリケーションが Web サービスの呼び出しに使用する URL の Web サービス URI 部分。 **注意：** `/TraderService` のように、先頭に「/」を必ず指定すること。 この属性の値は、生成された web-services.xml デプロイメント記述子の `<web-service>` 要素の uri 属性の値になる。	必須
sourcePath	javaSource 属性で指定された Java ソース ファイルによって参照される追加クラスが格納されているディレクトリの完全パス名。	任意

属性	説明	必須 / 任意
ignoreAuthHeader	SOAP リクエストの `Authorization` HTTP ヘッダが Web サービスで無視されることを指定する。 **注意：** この属性は、慎重に使用する必要がある。この属性の値を `True` に設定すると、Web サービスを構成する EJB、Web アプリケーション、またはエンタープライズ アプリケーションでアクセス制御のセキュリティ制約が定義されている場合でも、WebLogic Server は Web サービスを呼び出そうとしているクライアント アプリケーションの認証を絶対に行わない。つまり、認証資格を持たないクライアント アプリケーションでも、セキュリティ制約が定義されている Web サービスを呼び出せることになる。 有効な値は `True` および `False`。デフォルト値は `False`。	任意

wsdl2Service

`wsdl2Service` Ant タスクは既存の WSDL ファイルを入力として、以下の WebLogic Web サービス コンポーネントを生成します。

- `web-services.xml` デプロイメント記述子ファイル
- Web サービスを部分的に実装する Java ソース ファイル

Java ソース ファイルには、Java クラスのみで実装される WebLogic Web サービスのテンプレートが格納されます。このテンプレートには、WSDL ファイルのオペレーションに対応する完全なメソッド シグネチャが含まれています。希望通りに機能するようにそれらのメソッドの実際のコードを記述します。

wsdl2Service Ant タスクは、(<service> 要素で指定された) WSDL ファイルの 1 つのサービスのみで部分的な実装を生成します。serviceName 属性を使用すると、特定のサービスを指定できます。この属性を指定しない場合、wsdl2Service Ant タスクでは WSDL ファイルの最初の <service> 要素の部分的な実装が生成されます。

wsdl2Service Ant タスクは、WSDL ファイルのオペレーションのパラメータまたは戻り値として使用される非組み込みデータ型のデータ型マッピング情報を生成しません。WSDL で非組み込みデータ型が使用される場合は、まず autotype Ant タスクを実行してデータ型マッピング情報を生成し、その後に wsdl2Service Ant タスクの typeMappingFile 属性を autotype Ant タスクで生成された types.xml ファイルに設定します。

注意： wsdl2Service Ant タスクの完全修飾名は、weblogic.ant.taskdefs.webservices.wsdl2service.WSDL2Service です。

例

```
<wsdl2service
      wsdl="c:\wsdls\myService.wsdl"
      destDir="c:\myService\implementation"
      typeMappingFile="c:\autotype\types.xml"
      packageName="example.ws2j.service"
/>
```

属性

次の表では、wsdl2Service Ant タスクの属性について説明します。

属性	説明	必須 / 任意
wsdl	部分的な WebLogic Web サービスの実装が生成される Web サービスを記述する WSDL の完全パス名または URL。	必須

属性	説明	必須 / 任意
destDir	生成されたコンポーネント (`web-services.xml` ファイルおよび Web サービスを部分的に実装する Java ソース ファイル) が格納されるディレクトリの完全パス名。	必須
packageName	Web サービスを部分的に実装する、生成された Java ソース ファイルのパッケージ名。	必須
serviceName	部分的な WebLogic 実装が生成される WSDL ファイル内の Web サービスの名前。WSDL ファイル内の Web サービスの名前は、`<service>` 要素の `name` 属性の値。 この属性を指定しない場合、`wsdl2Service` Ant タスクは WSDL ファイルで見つかった最初の `<service>` 要素の部分的な実装を生成する。 **注意：** `wsdl2Service` Ant タスクは、WSDL ファイルの 1 つのサービスのみで部分的な WebLogic Web サービス実装を生成する。WSDL ファイルに複数の Web サービスが含まれている場合は、この属性の値を変えながら、`wsdl2Service` を複数回実行する必要がある。	任意
typeMappingFile	WSDL ファイルの Web サービスのオペレーションによって参照されるすべての非組み込みデータ型のデータ型マッピング情報が格納されているファイル。情報の形式は、`web-services.xml` ファイルの `<type-mapping>` 要素内のデータ型マッピング情報と同じ。 通常は、最初に `autotype` Ant タスク (`wsdl` 属性を指定) を同じ WSDL ファイルに対して実行し、すべての非組み込みデータ型コンポーネントを生成する。生成されるコンポーネントの 1 つは、非組み込みデータ型のマッピング情報が格納される `types.xml` というファイル。`typeMappingFile` 属性はこのファイルに設定する。	必須 (WSDL ファイルの Web サービスのオペレーションが非組み込みデータ型を参照する場合のみ)

wspackage

wspackage Ant タスクは、WebLogic Web サービスのさまざまなコンポーネントをデプロイ可能な EAR ファイルにパッケージ化します。これは、以下のようなコンポーネントが既に生成されていることを前提とします。

- web-services.xml デプロイメント記述子ファイル。
- Web サービスを実装する EJB が格納された EJB JAR ファイル。
- Web サービスを実装する Java クラス ファイル。
- Web サービスを呼び出すためにダウンロードして使用できるクライアント JAR ファイル。
- SOAP ハンドラの実装。
- Web サービスのパラメータおよび戻り値として使用される非組み込みデータ型のコンポーネント。それらのコンポーネントには、データ型の XML および Java 表現、およびそれら 2 つの表現の間でデータを変換するシリアライゼーション クラスが含まれる。

通常は、clientgen、autotype、source2wsdd、wsdl2Service といった他の Ant タスクを使用して前述のコンポーネントを生成します。

注意： wspackage Ant タスクの完全修飾名は、weblogic.ant.taskdefs.webservices.wspackage.WSPackage です。

例

```
<wspackage
      output="c:\myWebService.ear"
      contextURI="web_services"
      codecDir="c:\autotype"
      webAppClasses="example.ws2j.service.SimpleTest"
      ddFile="c:\ddfiles\web-services.xml"
/>
```

属性

次の表では、wspackage Ant タスクの属性について説明します。

属性	説明	必須 / 任意
output	Web サービスとその全コンポーネントを含む EAR ファイルまたは展開ディレクトリのパス名。 EAR ファイルを作成または更新するには、EAR ファイルの指定時に .ear サフィックスを使用する (c:\mywebservice.ear など)。属性値に .ear サフィックスがない場合、wspackage タスクは展開ディレクトリを作成する。 存在しない EAR ファイルまたはディレクトリを指定すると、wspackage タスクが新しいファイルまたはディレクトリを作成する。	必須
warName	Web サービスが書き込まれる WAR ファイルの名前。WAR ファイルは EAR ファイルの最上位に作成される。 デフォルト値は web-services.war。 **注意 :** 展開ディレクトリを output 属性を使用して指定した場合、warName 属性に .war サフィックスを指定しても、wspackage タスクは展開 Web アプリケーション ディレクトリを作成する。	任意
contextURI	Web サービスのコンテキスト ルート。この値は、Web サービスを呼び出す URL 内に使用する。 contextURI 属性のデフォルト値は warName 属性の値になる。	任意
ddFile	既存の Web サービスのデプロイメント記述子ファイル (web-services.xml) の完全パス名。	必須

属性	説明	必須 / 任意
filesToEar	EAR のルート ディレクトリにパッケージ化されるファイルのカンマ区切りリスト。 この属性を使用すると、他のサポート EJB JAR ファイルに加えて、Web サービスを実装する EJB JAR ファイルを指定できる。	任意
filesToWar	Web サービスの Web アプリケーションのルート ディレクトリにパッケージ化される (クライアント JAR ファイルなどの) 追加ファイルのカンマ区切りリスト。	任意
webAppClasses	Web サービスの Web アプリケーションの `WEB-INF/classes` ディレクトリにパッケージ化する必要のあるクラスファイルのカンマ区切りリスト。 この属性を使用すると、Web サービスを実装する Java クラス、SOAP ハンドラ クラスなどを指定できる。	任意
codecDir	Web サービスでパラメータまたは戻り値として使用される非組み込みデータ型のシリアライゼーション クラスが格納されているディレクトリの名前。	任意
overwrite	既存の EAR ファイルまたはディレクトリのコンポーネントを上書きするかを指定する。コンポーネントとしては、`web-services.xml` ファイル、シリアライゼーション クラス、クライアント JAR ファイルなどがある。 この属性の有効な値は、`True` および `False`。デフォルト値は `True`。 `False` を指定すると、`wspackage` Ant タスクは EAR ファイル / ディレクトリの内容と情報を `web-services.xml` ファイルにマージしようとする。	任意

wsgen

wsgen Ant タスクは、バージョン 6.1 の WebLogic Web サービスをバージョン 7.0 にアップグレードします。具体的には、次の操作を行います。

- アップグレードされた Web サービスを記述する web-services.xml ファイルを新規作成します。このタスクは 6.1 build.xml ファイル内の値とデフォルト値を使用してデプロイメント記述子を生成します。
- 必要であれば、Web サービスを呼び出すクライアント アプリケーションが必要とするすべてのクラス、スタブ、インタフェースを含むクライアント JAR ファイルを作成します。クラス、スタブ、インタフェースは JAX-RPC API に基づいています。
- web-services.war という WAR ファイルにすべての Web サービス コンポーネントを パッケージ化して、次に、WAR および EJB JAR ファイルをデプロイ可能な EAR ファイルにパッケージ化します。

このタスクは、バージョン 6.1 で使用されていたのと同じ build.xml ファイルを入力として受け取って Web サービスを作成します。B-40 ページの「その他の属性」に説明されているとおり、2 つの新しい属性を必要に応じて追加できます。このタスクは、6.1 Web サービスの作成に使用されていたものと同じアーカイブ ファイルまたは展開ディレクトリも入力として受け取ります。

警告： 新しい 7.0 Web サービスのアセンブルにはこの Ant タスクを使用しないでください。代わりに、servicegen Ant タスクを使用します。wsgen Ant タスクは Web サービスをバージョン 6.1 から 7.0 にアップグレードするためにのみ使用されるので、このタスクはこのバージョンの WebLogic Server では非推奨になっています。

注意： wsgen Ant タスクの完全修飾名は、weblogic.ant.taskdefs.webservices.wsgen.WSGenTask です。

例

次の build.xml は、6.1 RPC スタイル Web サービスの例からの抜粋です。この例には、新しい属性である targetNameSpace が含まれています。

```
<wsgen
   destpath="weather.ear"
   context="/weather">
   <rpcservices path="weather.jar">
      <rpcservice bean="statelessSession"
                  uri="/weatheruri"
                  targetNameSpace="http://www.bea.com/examples"/>
   </rpcservices>
</wsgen>
```

その他の属性

wsgen を使用して 6.1 Web サービスをバージョン 7.0 にアップグレードするときは 6.1 build.xml ファイルを使用するため、build.xml ファイルの属性と要素は、6.1 バージョンの『WebLogic Web サービス プログラマーズ ガイド』で説明されている属性と要素になります。詳細については、「build.xml の要素と属性」を参照してください。

次の表にリストされている 2 つの属性を 6.1 build.xml ファイルに追加でき、wsgen によって新しい web-services.xml デプロイメント記述子ファイルに適切な情報が追加されます。

属性	要素	説明
targetNameSpace	rpcservice、 messageservice	Web サービスのネームスペース URI。 この属性のデフォルト値は、既存の 6.1 build.xml ファイルに追加していない場合、http://example.org になる。
packageName	clientjar	生成されたクライアント インタフェースとスタブ ファイルをクライアント JAR ファイルにパッケージ化するパッケージ名。 この属性のデフォルト値は、既存の 6.1 build.xml ファイルに追加していない場合、org.example になる。

C　WebLogic Web サービスのカスタマイズ

この章では、Web サービス WAR ファイルの Web アプリケーション デプロイメント記述子ファイルを更新することにより WebLogic Web サービスをカスタマイズする方法について説明します。

- C-1 ページの「静的 WSDL ファイルのパブリッシュ」
- C-2 ページの「カスタム WebLogic Web サービス ホーム ページの作成」

静的 WSDL ファイルのパブリッシュ

デフォルトでは、WebLogic Server は WebLogic Web サービスの WSDL を `web-services.xml` デプロイメント記述子ファイルのコンテンツに基づいて動的に生成します。動的に生成された WSDL の URL を取得する方法の詳細については、8-23 ページの「WebLogic Web サービスのホーム ページおよび WSDL の URL」を参照してください。

静的な WSDL ファイルを Web サービス EAR ファイルに含め、その URL を Web サービスの正式な記述としてパブリッシュすることもできます。静的 WSDL をパブリッシュする理由の 1 つは、動的に生成された WSDL に含まれる以上のカスタム ドキュメントを追加できることです。

警告： 静的 WSDL を Web サービスの正式な記述としてパブリッシュする場合は、それが実際の Web サービスを反映して常に最新の状態に保たれるようにする必要があります。つまり、Web サービスを変更した場合は、Web サービスに行った変更を反映するよう静的 WSDL も手動で変更する必要があります。Weblogic が生成する動的 WSDL を使用するメリットの 1 つは、それが常に最新の状態に保たれることです。

動的に生成される WSDL を使わずに静的 WSDL ファイルを Web サービス EAR ファイルに含めてパブリッシュするには、次の手順に従います。

1. WebLogic Web サービス EAR ファイルを unJAR し、次に `web-services.xml` ファイルを含んでいる WAR ファイルを unJAR します。
2. 展開された Web アプリケーションのディレクトリに静的 WSDL ファイルを入れます。この手順では、ファイルを最上位ディレクトリに入れたことを前提にしています。
3. Web アプリケーションの `web.xml` ファイルを更新すると、WSDL ファイルの拡張子を XML MIME タイプにマップする `<mime-mapping>` 要素が追加されます。

   たとえば、静的 WSDL ファイルの名前が `myService.wsdl` の場合、`web.xml` ファイル内の対応するエントリは次のとおりです。

   ```
   <mime-mapping>
           <extension>wsdl</extension>
           <mime-type>text/xml</mime-type>
   </mime-mapping>
   ```

4. Web サービス WAR ファイルと EAR ファイルをもう一度 jar ファイルにパッケージングします。
5. 標準 URL を使用して静的 WSDL ファイルを呼び出し、Web アプリケーション内の静的ファイルを呼び出します。

   たとえば、`web_services` のコンテキスト ルートが入れられた Web アプリケーション内の `myService.wsdl` ファイルを呼び出すには以下の URL を使用します。

   `http://`*`host`*`:`*`port`*`/web_services/myService.wsdl`

カスタム WebLogic Web サービス ホームページの作成

すべての WebLogic Web サービスには、Web サービスの WSDL の表示、サービスのテスト、クライアント JAR ファイルのダウンロード、Web サービスを呼び出すクライアント アプリケーションの SOAP 要求および応答の表示を行うため

のリンクを含むデフォルトのホーム ページがあります。詳細については、8-23 ページの「WebLogic Web サービスのホーム ページおよび WSDL の URL」を参照してください。

WebLogic Server は Web サービス ホーム ページを動的に生成するので、このページはカスタマイズできません。カスタム ホーム ページを作成するには、Web サービス WAR ファイルに HTML ファイルまたは JSP ファイルを追加します。JSP の作成の詳細については、『WebLogic JSP プログラマーズ ガイド』を参照してください。

D WebLogic Web サービスでサポートされている仕様

WebLogic Web サービスでは、以下の仕様がサポートされています。

- JAX-RPC 1.0
- SOAP 1.1
- 添付ファイル付き SOAP メッセージ
- Web Services Description Language (WSDL) 1.1
- UDDI 2.0
- XML Schema Part 1: Structures
- XML Schema Part 2: Structures
- JSSE