BEA WebLogic Server™

Web サービス プログラマーズ ガイド

BEA WebLogic Server バージョン 6.1
マニュアルの日付 : 2002 年 11 月 1 日

著作権

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

限定的権利条項

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

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

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

商標または登録商標

BEA、Jolt、Tuxedo、および WebLogic は BEA Systems, Inc. の登録商標です。BEA Builder、BEA Campaign Manager for WebLogic、BEA eLink、BEA Manager、BEA WebLogic Collaborate、BEA WebLogic Commerce Server、BEA WebLogic E-Business Platform、BEA WebLogic Enterprise、BEA WebLogic Integration、BEA WebLogic Personalization Server、BEA WebLogic Process Integrator、BEA WebLogic Server、E-Business Control Center、How Business Becomes E-Business、Liquid Data、Operating System for the Internet、および Portal FrameWork は、BEA Systems, Inc. の商標です。

その他の商標はすべて、関係各社がその権利を有します。

WebLogic Server 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	2002 年 11 月 1 日	BEA WebLogic Server バージョン 6.1

目次

このマニュアルの内容

対象読者 x
e-docs Web サイト x
このマニュアルの印刷方法 x
サポート情報 xi
表記規則 xii

1. WebLogic Web サービスの概要

Web サービス 1-1
Web サービスを使用する理由 1-2
Web サービス コンポーネント 1-3
添付ファイル付き SOAP 1.1 1-4
WSDL 1.1 1-5
WebLogic Web サービスの機能 1-6
Web サービス プログラミング モデル 1-6
RPC スタイル Web サービス 1-7
メッセージスタイル Web サービス 1-7
SOAP 1.1 の実装 1-8
Web サービス実行時コンポーネント 1-8
標準化 J2EE Web サービス アセンブリおよびデプロイメント 1-8
WSDL ファイルの生成 1-9
WebLogic Web サービスを呼び出す Java クライアント 1-9
Web サービスの作成および呼び出し例 1-9
WebLogic Web サービスのアーキテクチャ 1-10
RPC スタイル WebLogic Web サービス アーキテクチャ 1-11
メッセージスタイル WebLogic Web サービス アーキテクチャ 1-12
WebLogic Web サービスでサポートされていない SOAP および WSDL 機能 1-14
XML ファイルの編集 1-15

2. WebLogic Web サービスの開発

WebLogic Web サービスの開発 主な手順 2-1
WebLogic Web サービスの設計 2-3
RPC スタイルまたはメッセージスタイル Web サービスの選択 2-4
RPC スタイル Web サービスを使用する場合 2-4
メッセージスタイル Web サービスを使用する場合 2-4
RPC スタイル Web サービスを実装する EJB 2-5
既存の EJB アプリケーションの RPC スタイル Web サービスへの変換 . 2-5
ステートレス セッション EJB 内でのメソッドのオーバーロードの回避 2-6
メッセージスタイル Web サービスと JMS 2-6
キューまたはトピックの選択 2-7
ドキュメントの取得と処理 2-7
メッセージスタイル Web サービスの例 2-7
既存の JMS アプリケーションの Web サービスへの変換 2-9
WebLogic Web サービスのパラメータおよび戻り値でサポートされているデータ型 2-9
WebLogic Web サービスでの XML と Java 間の変換 2-12
セキュリティの問題 2-14
メッセージスタイル Web サービスのセキュリティ設定 2-14
RPC スタイル Web サービスのセキュリティ設定 2-16
WebLogic Web サービスを呼び出すときの相互 SSL の使い方 2-16
WebLogic Web サービスの実装 2-18
RPC スタイル Web サービスの実装 2-18
メッセージスタイル Web サービスの実装 2-19
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 2-20
WebLogic Web サービスのアセンブル 2-21
Java Ant タスクを使用した WebLogic Web サービスのアセンブル 2-21
Ant build.xml ファイルの例 2-23
build.xml Ant ビルド ファイルの作成 2-26
動的または静的 WSDL 2-28
WebLogic Web サービスのデプロイ 2-28
WebLogic Web サービスのデプロイ : 簡単な例 2-28

EJB 用の Java コードの記述 2-30
EJB デプロイメント記述子の作成 2-34
EJB のアセンブル 2-35
build.xml ファイルの作成..2-36

3. WebLogic Web サービスの呼び出し

WebLogic Web サービスの呼び出しの概要..3-2
WebLogic Web サービス クライアント API..3-3
WebLogic Web サービス クライアント API でサポートされているクライアント モード 3-3
WebLogic Web サービスを呼び出すクライアントの例 3-4
WebLogic Web サービス ホーム ページの呼び出し 3-5
Web サービス ホーム ページからの WSDL の取得...............................3-6
Java クライアント JAR ファイルの Web サービス ホーム ページからのダウンロード 3-7
WebLogic Web サービスを呼び出して WSDL を取得する URL 3-8
RPC スタイル WebLogic Web サービスを呼び出すクライアントの作成....3-9
Java クライアントの記述 3-10
静的 Java クライアントの記述 3-10
動的 Java クライアントの記述 3-13
Microsoft SOAP Toolkit クライアントの記述.......................................3-15
メッセージスタイル WebLogic Web サービスを呼び出す Java クライアントの作成 3-16
メッセージスタイル Web サービスへのデータの送信 3-18
メッセージスタイル Web サービスからのデータの受信 3-20
WebLogic Web サービスからの例外の処理..3-23
Web サービスを呼び出すための初期コンテキスト ファクトリ プロパティ . 3-24
WebLogic Web サービスを呼び出すクライアントに必要な追加クラス 3-25

4. WebLogic Web サービスの管理

WebLogic Web サービスの管理の概要 4-1
Administration Console の起動..4-1
WebLogic Server にデプロイされている Web サービスの表示 4-2

5. トラブルシューティング

vorbose モードのチューニング 5-1
java.io.FileNotFoundException....................5-2
解析不能例外 5-4
java.lang.NullPointerException....................5-6
java.net.ConnectException 5-7

6. 相互運用性

.NET クライアントと 6.1 WebLogic Web サービスとの間の相互運用性.....6-1
7.X WebLogic クライアントと 6.1 WebLogic Web サービスとの間の相互運用性 6-2

A. WebLogic Web サービスでサポートされている仕様

SOAP 1.1 の仕様 A-1
添付ファイル付き SOAP メッセージの仕様....................A-1
Web Services Description Language（WSDL）1.1 仕様 A-2

B. build.xml の要素と属性

build.xml ファイルの例....................B-1
build.xml 階層図 B-2
要素と属性の説明....................B-3
- wsgen....................B-3
- rpcservices....................B-5
- rpcservice B-6
- messageservices B-7
- messageservice....................B-7
- clientjar....................B-9
- manifest B-9
- entry....................B-10

C. Web サービス アーカイブ ファイルの手動によるアセンブル

始める前に C-1
Web サービス アーカイブ ファイルの説明....................C-2
RPC スタイル Web サービス アーカイブ ファイルの手動によるアセンブル . C-3

RPC スタイル Web サービスの web.xml ファイルの更新 C-7
RPC スタイル Web サービスの weblogic.xml ファイルの更新 C-10
RPC スタイル Web サービスの application.xml ファイルの更新 C-11
メッセージスタイル Web サービス アーカイブ ファイルの手動によるアセンブル C-12
メッセージスタイル Web サービス WSDL ファイルの作成 C-15
メッセージスタイル Web サービスの web.xml ファイルの更新 C-17
メッセージスタイル Web サービスの weblogic.xml ファイルの更新 C-21
メッセージスタイル Web サービスの application.xml ファイルの更新 . . C-22
client.jar ファイルの手動による作成 C-23

D. WSDL ファイルを使用しない Web サービスの呼び出し

用語集

索引

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。

このマニュアルの構成は次のとおりです。

- 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、およびそれらのアーキテクチャについて説明します。
- 第 2 章「WebLogic Web サービスの開発」では、WebLogic Web サービスの開発方法について説明します。
- 第 3 章「WebLogic Web サービスの呼び出し」では、クライアント アプリケーションから WebLogic Web サービスにアクセスする方法について説明します。
- 第 4 章「WebLogic Web サービスの管理」では、Administration Console を使用して Web サービスを管理する方法について説明します。
- 第 5 章「トラブルシューティング」では、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ときに発生する場合がある問題を解決する方法について説明します。
- 付録 A「WebLogic Web サービスでサポートされている仕様」では、WebLogic Web サービスでサポートされている仕様へのリンクを示します。
- 付録 B 「build.xml の要素と属性」では、Web サービスをエンタープライズ アプリケーション アーカイブ（`*.ear`）ファイルにアセンブルするために使用する `build.xml` Java ビルド ファイルについて説明します。
- 付録 C「Web サービス アーカイブ ファイルの手動によるアセンブル」では、**wsgen** Ant タスクを使用しないで手動で Web サービス アーカイブを作成する方法について説明します。
- 付録 D「WSDL ファイルを使用しない Web サービスの呼び出し」では、WSDL を使用しないで 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方法について説明します。

関連する用語の説明と索引が最後にあります。

対象読者

このマニュアルは、Web サービスとしてサードパーティ クライアントが利用できる WebLogic Server で現在実行している EJB の構築を目的としているアプリケーション開発者を対象としています。

Web テクノロジ、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、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）を書籍の形式で印刷できます。PDF を表示するには、WebLogic Server ドキュメントのホーム ページを開き、[ドキュメントのダウンロード] をクリックして、印刷するマニュアルを選択します。

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

サポート情報

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

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

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

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

表記規則

このマニュアルでは、全体を通して以下の表記規則が使用されています。

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

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

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

<table>
<tr><th>表記法</th><th>適用</th></tr>
<tr><td>[ ]</td><td>構文内の任意指定の項目を示す。
例 :

java utils.MulticastTest -n *name* -a *address*
     [-p *portnumber*] [-t *timeout*] [-s *send*]</td></tr>
<tr><td>|</td><td>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}</td></tr>
<tr><td>...</td><td>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。</td></tr>
<tr><td>.
.
.</td><td>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。</td></tr>
</table>

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。

- 1-1 ページの「Web サービス」
- 1-2 ページの「Web サービスを使用する理由」
- 1-3 ページの「Web サービス コンポーネント」
- 1-6 ページの「WebLogic Web サービスの機能」
- 1-10 ページの「WebLogic Web サービスのアーキテクチャ」
- 1-14 ページの「WebLogic Web サービスでサポートされていない SOAP および WSDL 機能」

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目的においては効率よく機能しますが、外部に対して閉鎖的で、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ますが、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されます。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 サービスを使用する主な理由は、以下の利点が得られることです。

- 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を通過するアプリケーションの利用可能性
- 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、異なるシステム間での通信の問題を解決します。

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。これらのシステムは、Java、C++、Perl などのさまざまな言語で記述されています。この機能を提供するアプリケーションが Web サービスとしてパッケージ化されている限り、それぞれのシステムは相互に通信できます。

Web サービス コンポーネント

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 JavaBean および JMS など）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 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。SOAP については、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。

- 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。WSDL の詳細については、1-5 ページの「WSDL 1.1」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。プロトコルの構成は次のとおりです。

- SOAP メッセージを記述するエンベロープ。特に、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。

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（WDSL）は、Web サービスを記述するために使用される 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>
```

WebLogic Web サービスの機能

この章では、WebLogic Web サービス サブシステムの機能について説明します。

- Web サービス プログラミング モデル
- SOAP 1.1 の実装
- Web サービス実行時コンポーネント
- 標準化 J2EE Web サービス アセンブリおよびデプロイメント
- WSDL ファイルの生成
- WebLogic Web サービスを呼び出す Java クライアント
- Web サービスの作成および呼び出し例

Web サービス プログラミング モデル

プログラミング モデルには、WebLogic Server によってホストされる Web サービスの実装、アセンブル、デプロイ、および呼び出し方法が記述されています。Web サービスの実際の作業を実行するエンタープライズ JavaBean コードは別として、Web サービス自体のほとんどは、Web サービスのコンポーネントを生成してパッケージ化する Java Ant タスク `wsgen` を使用して開発できます。

WebLogic Server は、リモート プロシージャ呼び出し（RPC）スタイルとメッセージスタイルの 2 種類の Web サービスをサポートします。

RPC スタイル Web サービス

リモート プロシージャ呼び出し（RPC）スタイル 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ます。これは、クライアント アプリケーションへのリモート オブジェクトになります。

クライアントと RPC スタイル Web サービスとの対話は、サービス固有のインタフェースに集中します。クライアントが Web サービスを起動すると、パラメータ値が Web サービスに送信され、必要なメソッドが実行されて戻り値が送り返されます。クライアントと Web サービス間のこの対話により、RPC スタイル Web サービスは密に結合され、RMI または DCOM などの従来の分散型オブジェクト パラダイムと類似したものになります。

RPC スタイル Web サービスは同期です。つまり、クライアントがリクエストを送信したときは応答があるまで何もしないで待機します。

メッセージスタイル Web サービス

メッセージスタイル Web サービスは、メッセージ駆動型 Bean などの JMS メッセージ リスナを使用して実装され、JMS の送り先と関連付けられている必要があります。

メッセージスタイル Web サービスは疎結合で、サービス固有のインタフェースに関連付けられるのではなく、ドキュメント駆動型です。クライアントがメッセージスタイル Web サービスを起動すると、クライアントでは一般的にディスクリートなパラメータのセットではなく発注書などの全ドキュメントを送信します。Web サービスでは、ドキュメント全体を受け入れて処理し、結果メッセージを返す場合と返さない場合があります。クライアントと Web サービス間のリクエストと応答が密に結合されていないため、メッセージスタイル Web サービスではクライアントとサーバ間の結合が緩やかになります。

メッセージスタイル Web サービスは非同期です。Web サービスを呼び出すクライアントは、応答を待機しません。Web サービスからの応答がある場合は、数時間または数日後に示されることもあります。

クライアントは、メッセージスタイル Web サービスとの間でドキュメントの送信も受信も実行できますが、同じ Web サービスを使用して両方を行うことはできません。

SOAP 1.1 の実装

WebLogic Server には、開発者が Web サービスを呼び出すクライアントを作成するために使用できる独自の SOAP 1.1 および添付ファイル付き SOAP 1.1 仕様実装があります。

RPC スタイル Web サービスでは、SOAP 1.1 メッセージ フォーマットを使用し、メッセージスタイル Web サービスでは、添付ファイル付き SOAP 1.1 メッセージフォーマットを使用します。

注意 : 現在 WebLogic Web サービスでは、添付ファイル付き SOAP メッセージの実際の添付ファイルは無視されます。

Web サービス実行時コンポーネント

WebLogic Web サービス実行時コンポーネントは、Web サービスの作成に必要なサーブレットと関連インフラストラクチャのセットです。実行時の要素の 1 つは、クライアントからの SOAP リクエストを処理するサーブレットのセットです。これらのサーブレットは、WebLogic Server 配布キットに含まれているため、記述する必要はありません。実行時のもう一つの要素は、WebLogic Web サービスのすべてのコンポーネントを生成しアセンブルする Ant タスクです。

標準化 J2EE Web サービス アセンブリおよびデプロイメント

Web サービス開発者は、`wsgen` という Ant タスクと Administration Console を使用し、Web サービスを標準 J2EE エンタープライズ アプリケーションとして `*.ear` ファイルにアセンブルおよびデプロイします。`*.ear` ファイルには、EJB、SOAP サーブレットへの参照、`web.xml` ファイル、`weblogic.xml` ファイルなど、Web サービスのすべてのコンポーネントが含まれます。

WSDL ファイルの生成

WebLogic Web サービスを呼び出すクライアントを作成する開発者には、Web サービスを記述する WSDL が必要です。WebLogic Server では、デプロイされた Web サービスの WSDL を自動的に生成します。Web サービスの WSDL には、特別な URL を介してアクセスします。

WebLogic Web サービスを呼び出す Java クライアント

WebLogic Server は、開発者が Web サービスを呼び出す Java クライアントを開発するために使用するシン Java クライアントを自動的に生成できます。Java クライアント JAR ファイルには、Web サービスを呼び出すために必要なすべてのクラスがあります。これらのクラスには、Java クライアント API クラスおよびインタフェース、SOAP リクエストおよび応答を解析するパーサ、EJB への Java インタフェースなどがあります。この Java 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クライアント アプリケーションは、クライアント コンピュータ上に完全な WebLogic Server JAR ファイルを持っている必要はありません。

Java クライアント JAR ファイルは、WebLogic Web サービスのホーム ページからダウンロードできます。この Web ページの詳細については、第 3 章「WebLogic Web サービスの呼び出し」の 3-5 ページの「WebLogic Web サービス ホーム ページの呼び出し」を参照してください。

注意 : BEA では現在、クライアント機能をサーバ機能と別個にはライセンスしていないため、必要な場合には、この Java クライアント JAR ファイルをユーザ自身の顧客に再配布できます。

Web サービスの作成および呼び出し例

WebLogic Server には、RPC スタイル Web サービスおよびメッセージスタイル Web サービスを作成する両方の例と、Web サービスを呼び出す Java および Microsoft VisualBasic クライアント アプリケーションの両方の例があります。

これらの例は、***BEA_HOME*\samples\examples\webservices** ディレクトリにあります。***BEA_HOME*** は、メイン WebLogic Server インストール ディレクトリを指します。RPC スタイル Web サービスの例は **rpc** ディレクトリにあり、メッセージスタイル Web サービスの例は **message** ディレクトリにあります。

例を作成して実行する方法の詳細については、ブラウザで Web ページ *BEA_HOME*\samples\examples\webservices\package-summary.html を呼び出してください。

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、メッセージ駆動型 Bean、および JMS の送り先などの標準 J2EE コンポーネントを使用します。WebLogic Web サービスは完全に 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、優れたスケーラビリティ、トランザクションのサポート、自動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、簡単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。

WebLogic Server Web サービスは、次の特定のコンポーネントから構成される標準 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。

- 少なくとも SOAP メッセージをクライアントとの間で送受信するサーブレットを含む Web アプリケーション

  このサーブレットは、Web サービス開発プロセスの一部となっているため、開発者が自分で記述することはありません。

- RPC スタイル Web サービスを実装するステートレス セッション EJB またはメッセージスタイル Web サービス用の JMS リスナ（メッセージ駆動型 Bean など）

  RPC スタイル サービスでは、ステートレス セッション EJB が Web サービスの実際の作業のすべてを実行するか、または他の EJB に作業を分配します。Web サービスの実装によって、どの EJB が実際の作業を行うかが決まります。メッセージスタイル Web サービスでは、J2EE オブジェクト（一般

的にメッセージ駆動型 Bean）が JMS の送り先からメッセージを取得して処理します。

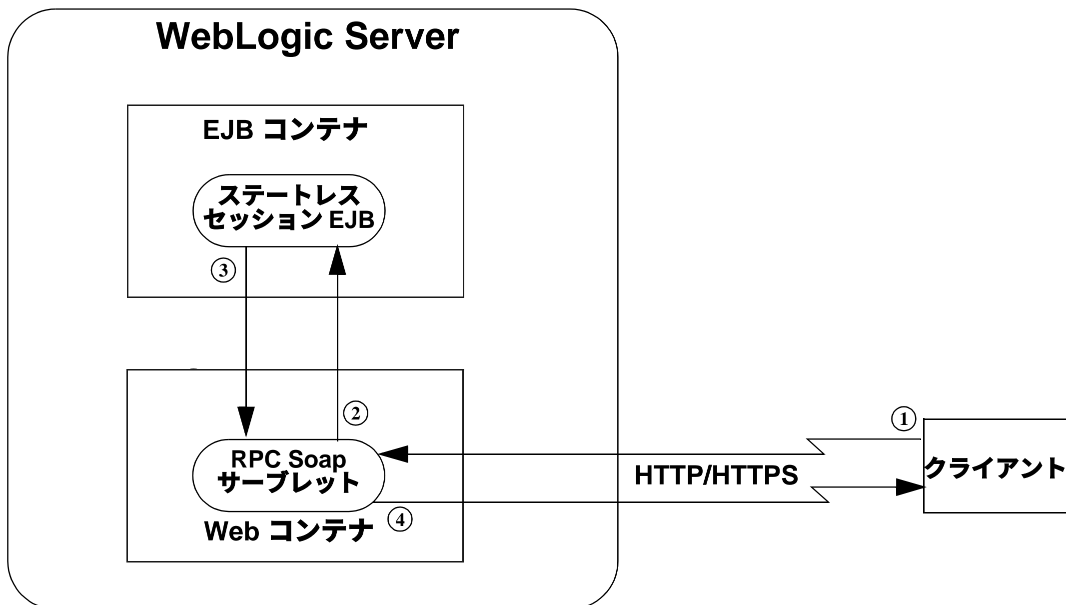
WebLogic Web サービスは、Web アプリケーションの Web アーカイブ（*.war）ファイルおよび EJB アーカイブ（*.jar）ファイルを含むエンタープライズアーカイブ（*.ear）ファイルとしてパッケージ化されます。

次の 2 つの節では、RPC スタイルおよびメッセージスタイル Web サービスのアーキテクチャについて説明します。

RPC スタイル WebLogic Web サービス アーキテクチャ

図 1-1 は、RPC スタイル WebLogic Web サービスのアーキテクチャを示しています。

図 1-1　RPC スタイル WebLogic Web サービス アーキテクチャ



クライアントが RPC スタイル WebLogic Web サービスを呼び出すときの動作を次に示します。

1. クライアントは HTTP/HTTPS を介して SOAP メッセージを WebLogic Server に送信します。SOAP メッセージには、RPC スタイル Web サービスを呼び出すための Web サービスの WSDL に準拠した指示が含まれます。

2. SOAP サーブレット（クライアントによって呼び出される Web アプリケーションの一部）は、RPC SOAP リクエストを処理するために設計されており、SOAP メッセージ エンベロープを開き、その情報に従って適切なステートレス セッション EJB ターゲットを識別します。次に、このサーブレットではパラメータのマーシャリングを解除して適切な Java オブジェクトにバインドし、ターゲット ステートレス セッション EJB を呼び出してパラメータに渡します。

   ステートレス セッション EJB では、Web サービス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。

3. 呼び出されたステートレス セッション EJB は、戻り値がある場合にはそれを RPC SOAP サーブレットに送信します。

4. RPC SOAP サーブレットは、戻り値をステートレス セッション EJB から SOAP メッセージにマーシャリングし、HTTP/HTTPS を介してクライアントに戻します。

   エラーが発生した場合、RPC SOAP サーブレットは SOAP エラー メッセージ（SOAP 障害）もクライアントに送信します。

メッセージスタイル WebLogic Web サービス アーキテクチャ

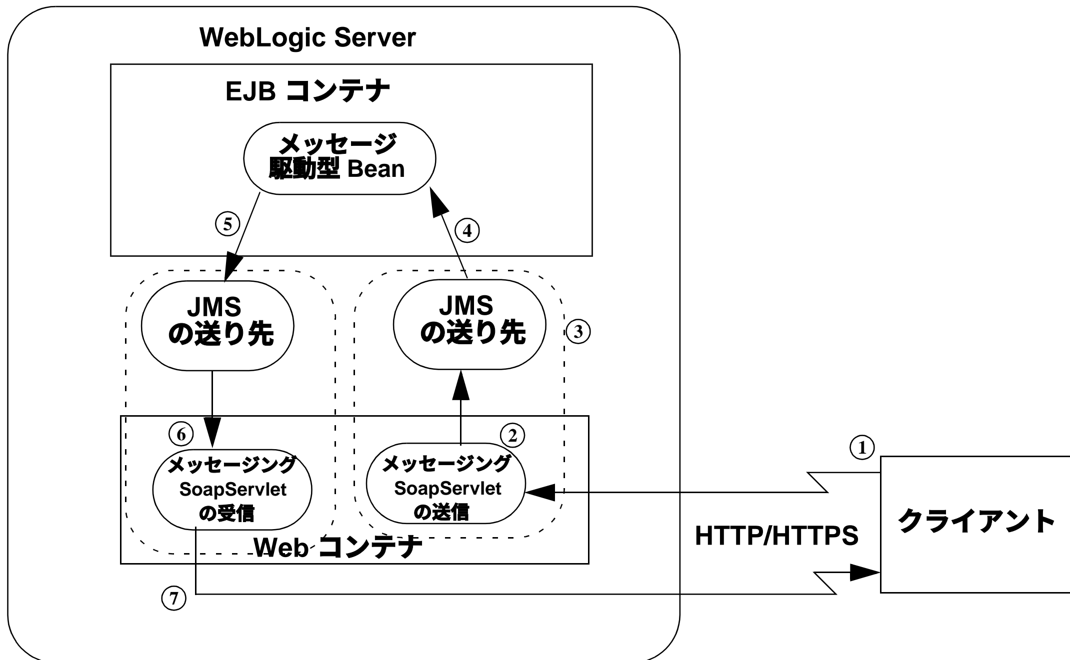
メッセージスタイル Web サービスでは、一方向通信をサポートします。クライアント アプリケーションは Web サービスからドキュメントを受け取るか、または Web サービスにドキュメントを送信しますが、単一メッセージスタイル Web サービスでクライアントが両方を行うことは許可されません。メッセージスタイル Web サービスを開発するときに、クライアントが Web サービスにメッセージを送信するか、または Web サービスからメッセージを受信するかを指定します。

送信用と受信用の 2 つのメッセージスタイル Web サービスを結合して双方向通信をサポートすることができます。同じクライアントが両方のタイプ、またはいずれかのタイプのサービスを使用することができます。

図 1-2 は、メッセージスタイル WebLogic Web サービスの両方のスタイルが一緒に機能するアーキテクチャを示しています。

注意 : 点線は、2 つの異なるメッセージスタイル Web サービスをカプセル化します。JMS の送り先からメッセージを取り出すのにメッセージ駆動型 Bean を使用する必要はありませんが、一般的には最適な方法です。

図 1-2 メッセージスタイル WebLogic Web サービス アーキテクチャ



クライアントがメッセージスタイル WebLogic Web サービスを呼び出すときの動作を次に示します。

1. クライアントは HTTP/HTTPS を介して SOAP メッセージを WebLogic Server に送信します。SOAP メッセージには、メッセージスタイル Web サービスを呼び出すための Web サービスの WSDL に準拠した指示が含まれます。

2. クライアントによって呼び出される Web アプリケーションの一部であるメッセージング SOAP サーブレットは、SOAP メッセージ エンベロープを開いて、メッセージの本文をデコードし、結果オブジェクトを適切な JMS の送り先に配置します。

   注意： WebLogic Server 6.1 では、添付ファイル付き SOAP 1.1 メッセージの添付ファイルの内容にアクセスすることはサポートしていません。

3. メッセージは、適切な JMS リスナ（一般的にはメッセージ駆動型 Bean）によって取り出されるまで JMS の送り先に残ります。
4. メッセージ駆動型 Bean は、メッセージを JMS の送り先から取り出します。メッセージ駆動型 Bean では、Web サービス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。
5. メッセージ駆動型 Bean は、クライアントでメッセージを受信するようにコンフィグレーションされた別のメッセージスタイル Web サービスに関連付けられている別の JMS の送り先に、結果ドキュメントを送信します。
6. 2 番目の Web サービスに関連付けられているメッセージング SOAP サーブレットは、メッセージを JMS の送り先から取り出します。
7. メッセージング SOAP サーブレットは、クライアントが 2 番目の受信 Web サービスを呼び出したときにドキュメントをクライアントに送り返します。

このサンプル アーキテクチャは、クライアントと情報を送受信するために一緒に機能する 2 つのメッセージスタイル Web サービスを示しています。クライアントは、2 つのメッセージスタイル Web サービスを呼び出す必要があります。

WebLogic Web サービスでサポートされていない SOAP および WSDL 機能

次の SOAP 機能は、WebLogic Web サービスではサポートされていません。

- `Header` 要素 - WebLogic Web サービスのクライアント API を使って、SOAP の `Header` 要素を設定または取得することはできません。さらに、内部の WebLogic Web サービス ランタイムは、SOAP の `Header` 要素を無視します。処理されるのは SOAP の `Body` 要素だけです。

- SOAP 添付ファイル

次の WSDL 機能は、WebLogic Web サービスではサポートされていません。

- `import` 要素
- `part` 要素の `element` 属性

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、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。これらのファイルを編集するには、BEA が提供する BEA XML エディタを使用します。BEA XML エディタは、完全な Java ベースのスタンドアロン XML エディタです。

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。ドキュメントを 2 通りに表示することにより、以下の 2 つの方法で XML ドキュメントを編集できます。

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

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

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

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

2 WebLogic Web サービスの開発

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

- 2-1 ページの「WebLogic Web サービスの開発 主な手順」
- 2-3 ページの「WebLogic Web サービスの設計」
- 2-18 ページの「WebLogic Web サービスの実装」
- 2-21 ページの「WebLogic Web サービスのアセンブル」
- 2-28 ページの「WebLogic Web サービスのデプロイ」
- 2-28 ページの「WebLogic Web サービスのデプロイ : 簡単な例」

WebLogic Web サービスの開発 主な手順

この章では、以下の手順についての詳細を説明します。

1. WebLogic Web サービスを設計します。

   Web サービスを RPC スタイルにするかメッセージスタイルにするか、どの EJB でサービスを実装するかなどを決定します。2-3 ページの「WebLogic Web サービスの設計」では、設計の考慮事項について説明します。

2. WebLogic Web サービスを実装します。

   WebLogic Web サービスの大部分を構成する EJB 用のビジネス ロジック Java コードを記述します。詳細については、2-18 ページの「WebLogic Web サービスの実装」を参照してください。

3. Web サービスを実装する EJB（RPC スタイル Web サービス用ステートレス セッション EJB およびメッセージスタイル Web サービス用メッセージ駆動型 Bean）をサポート EJB とともに EJB アーカイブ ファイル（`*.jar`）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。

4. WebLogic Web サービスをアセンブルします。

   サービスを構成するすべてのコンポーネント（ステートレス セッション EJB、SOAP サーブレットへの参照を含む Web アプリケーションなど）を J2EE エンタープライズ アプリケーション アーカイブ（`*.ear`）ファイルにパッケージ化して、WebLogic Server にデプロイできるようにします。Java Ant を使用して WebLogic Web サービスをアセンブルします。アセンブルでは、他の J2EE コンポーネント、メッセージスタイル Web サービスの JMS の送り先などの設定も参照します。

   詳細については、2-21 ページの「WebLogic Web サービスのアセンブル」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。詳細については、2-28 ページの「WebLogic Web サービスのデプロイ」を参照してください。

6. 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。詳細については、第 3 章「WebLogic Web サービスの呼び出し」を参照してください。

WebLogic Server には、RPC スタイル Web サービスおよびメッセージスタイル Web サービスを作成する両方の例と、Web サービスを呼び出す Java および Microsoft VisualBasic クライアント アプリケーションの両方の例があります。

これらの例は、*`BEA_HOME`*`\samples\examples\webservices` ディレクトリにあります。*`BEA_HOME`* は、メイン WebLogic Server インストール ディレクトリを指します。RPC スタイル Web サービスの例は `rpc` ディレクトリにあり、メッセージスタイル Web サービスの例は `message` ディレクトリにあります。

例を作成して実行する方法の詳細については、ブラウザで Web ページ *`BEA_HOME`*`\samples\examples\webservices\package-summary.html` を呼び出してください。

WebLogic Web サービスの設計

WebLogic Web サービスの大半は、SOAP リクエストが受信され、処理された後にバックグラウンドで作業を行う EJB です。

設計の最初の段階は、RPC スタイルまたはメッセージスタイルの Web サービスのどちらを作成するかを決定することです。詳細については、2-4 ページの「RPC スタイルまたはメッセージスタイル Web サービスの選択」を参照してください。

以下の節では、RPC スタイル設計の問題について説明します。

- RPC スタイル Web サービスを実装する EJB
- 既存の EJB アプリケーションの RPC スタイル Web サービスへの変換
- ステートレス セッション EJB 内でのメソッドのオーバーロードの回避

以下の節では、メッセージスタイル設計の問題について説明します。

- メッセージスタイル Web サービスと JMS
- 既存の JMS アプリケーションの Web サービスへの変換

以下の節では、両方のタイプの Web サービスに共通する問題について説明します。

- WebLogic Web サービスのパラメータおよび戻り値でサポートされているデータ型
- 次の表に、サポートされている XML データ型とそれに相当する Java との間のマッピングを示します。
- セキュリティの問題

RPC スタイルまたはメッセージスタイル Web サービスの選択

この節では、RPC スタイルまたはメッセージスタイルの Web サービスを使用する場合について説明します。

RPC スタイル Web サービスを使用する場合

RPC スタイル Web サービスは、インタフェース駆動型であり、基礎ステートレス セッション EJB のビジネス メソッドによって Web サービスの動作が決定されます。クライアントが Web サービスを起動すると、パラメータ値が Web サービスに送信され、対応するメソッドが実行されて戻り値が送り返されます。関係は同期で、クライアントが Web サービスからの応答を待機してから残りのアプリケーションに移行します。

アプリケーションが以下の特性を持っている場合は、RPC スタイル Web サービスを作成します。

- Web サービスを呼び出すクライアントが即時応答を必要としている場合
- クライアントと Web サービスが対話式に動作する場合
- 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

RPC スタイル Web サービスの例には、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。それぞれのケースで、情報が即時に返され、クライアントと Web サービス間の同期関係を示しています。

メッセージスタイル Web サービスを使用する場合

アプリケーションが次の特性を持っている場合は、メッセージスタイル Web サービスを作成します。

- クライアントが Web サービスに対して非同期関係の場合。つまり、クライアントが即時応答を期待していない場合。

- Web サービスが処理指向でなくデータ指向である場合。

メッセージスタイル Web サービスの例には、発注書の処理、新しい DSL ホームサービスに対する要求の受け入れ、または顧客からの見積り注文要求に対する応答などがあります。それぞれのケースで、クライアントは発注書などのドキュメント全体を Web サービスに送信して Web サービスで何らかの方法で処理されることを想定していますが、クライアントへの即時応答が必要ないか、応答の必要がまったくありません。Web サービスがこの非同期のドキュメント駆動型で機能する場合、メッセージスタイル Web サービスとして設計する必要があります。

RPC スタイル Web サービスを実装する EJB

Web サービスのすべての実際の作業を実行するか、または他の EJB に作業を分配するステートレス セッション EJB を使用して RPC スタイル Web サービスを実装します。この EJB は、クライアントが WebLogic Web サービスを呼び出すときに実行するメソッドを定義します。

クライアントと Web サービス間でやり取りするデータが最小限になるように EJB を設計します。この対話は同期で Web 上で行われるため、リクエストと応答が少ないほど全体のトランザクション速度が速くなります。

EJB のパラメータおよび戻り値のデータ型は、サポートされている Web サービス データ型のリストに制限されています。2-9 ページの「WebLogic Web サービスのパラメータおよび戻り値でサポートされているデータ型」を参照してください。このデータ型制限により、たとえば、Microsoft SOAP ToolKit などの Java および Java 以外の他の Web サービス実装との相互運用が容易になります。

既存の EJB アプリケーションの RPC スタイル Web サービスへの変換

パラメータおよび戻り値のデータ型が 2-9 ページの「WebLogic Web サービスのパラメータおよび戻り値でサポートされているデータ型」のサポートされている Web サービス データ型のリストにあれば、既存のステートレス セッション EJB を RPC スタイル Web サービスに変換できます。

既存の EJB を変換できない場合は、Web サービスを実装する新しいステートレス セッション EJB を作成し、サポートされているデータ型を使用してパラメータおよび戻り値をクライアントと送受信して、これらの値を正しいデータ型に変換してから値を既存のステートレス セッション EJB に渡す必要があります。

または、既存のステートレス セッション EJB を再プログラムして、サポートされているデータ型のみをパラメータおよび戻り値として受け取るようにすることもできます。

ステートレス セッション EJB 内でのメソッドのオーバーロードの回避

SOAP 仕様の制限により、SOAP メッセージは異なるシグネチャを持つ同じ名前のメソッド（オーバーロードされたメソッド）を明確に識別することができません。このため、WebLogic Server は RPC スタイル Web サービスを構成する EJB 内でオーバーロードされたメソッドをサポートしていません。各メソッドは固有の名前を持っている必要があります。

たとえば、ステートレス セッション EJB が文字列または整数をパラメータとして使用できる `myMethod()` というメソッドを定義したとします。SOAP 仕様では SOAP メッセージ内でパラメータのデータ型を宣言するように強制しないため、クライアントが呼び出したときに WebLogic Web サービスが `myMethod(String)` または `myMethod(int)` のどちらを実行するかを判断できない場合があります。混乱を避けるために、オーバーロードされたメソッドの 1 つを名前変更します。

メッセージスタイル Web サービスと JMS

メッセージスタイル Web サービスでは、エントリ ポイントとしてステートレス セッション EJB ではなく JMS リスナ（メッセージ駆動型 Bean など）を使用します。この節では、JMS と WebLogic Web サービスの関係およびメッセージスタイル Web サービスを開発するため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。

メッセージスタイル Web サービスを実装するときには、次の 2 つの点を決定する必要があります。

- 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、サービスにドキュメントを送信するか、またはサービスからドキュメントを受信するか。同じサービスで送信と受信の両方をサポートすることはできません。

ドキュメントの取得と処理

使用する JMS の送り先のタイプを決定した後で、JMS の送り先からドキュメント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、ドキュメント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 がドキュメントの処理を完了したら、Web サービスの実行は完了です。

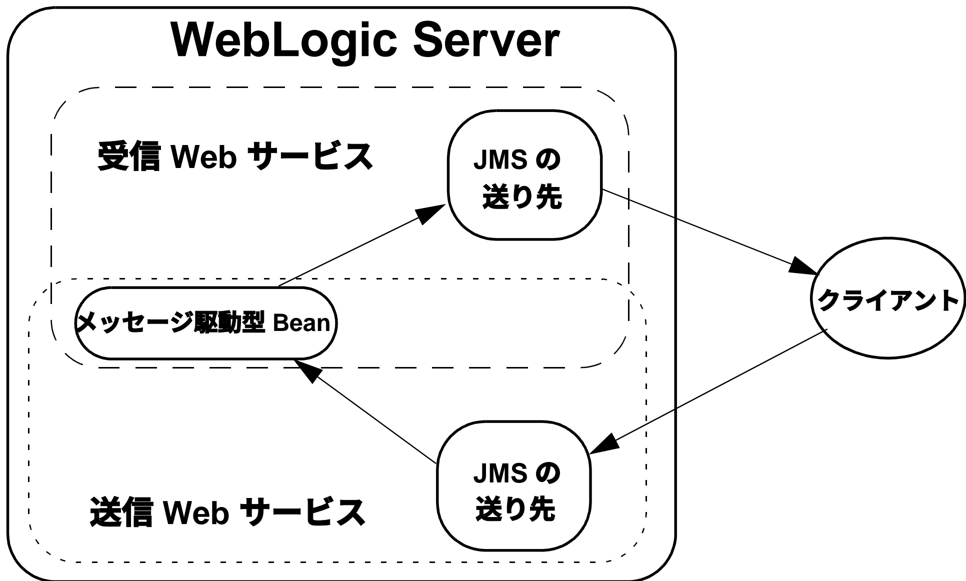
ドキュメントを送信して Web サービスを呼び出すクライアントが応答またはデータを受け取るようにするには、クライアントが応答を取得するために次に呼び出す 2 番目のメッセージスタイル Web サービスを作成する必要があります。2 番目の Web サービスは、オリジナルの Web サービスに関連しています。これは、ドキュメントを処理したオリジナルのメッセージ駆動型 Bean が結果情報または応答を 2 番目の Web サービスに対応する JMS の送り先に配置するためです。ここでも、2 番目の JMS の送り先がトピックまたはキューのどちらであるかを決定する必要があります。

メッセージスタイル Web サービスの例

図 2-1 は、クライアントからドキュメントを受信する Web サービスと、クライアントへドキュメントを送信する Web サービスの 2 つの簡単な例を示しています。2 つの Web サービスには、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 サービスを呼び出してドキュメントを WebLogic Server に送信し､次に 2 番目の Web サービスを呼び出して WebLogic Server からドキュメントを受信します。

図 2-1　メッセージスタイル Web サービスと JMS 間のデータ フロー



既存の JMS アプリケーションの Web サービスへの変換

JMS の送り先からメッセージを取得するメッセージ駆動型 Bean が JMS の送り先に送られる Java オブジェクトを処理できる場合は、既存の JMS アプリケーションをメッセージスタイル Web サービスに変換できます。たとえば、2-11 ページの「次の表に、サポートされている XML データ型とそれに相当する Java との間のマッピングを示します。」で説明しているように、WebLogic Web サービスではクライアントからの標準の XML ドキュメントを `org.w3c.dom.Document` オブジェクトに変換します。

既存の JMS アプリケーション内のメッセージ駆動型 Bean が他のタイプのドキュメント オブジェクトを予期している場合は、次の 2 つのいずれかを行うことができます。メッセージ駆動型 Bean を再プログラムして `org.w3c.dom.Document` オブジェクトを受け入れるようにするか、または `org.w3c.dom.Document` オブジェクトを受け入れる新しいメッセージ駆動型 Bean を作成し、オリジナルのメッセージ駆動型 Bean で使用できるデータ型に変換して、新しいオブジェクトを JMS の送り先に配置してオリジナルのメッセージ駆動型 Bean が取り出せるようにします。

WebLogic Web サービスのパラメータおよび戻り値でサポートされているデータ型

Java および Java 以外の他の Web サービス実装との相互運用性を容易にするため、WebLogic では Web サービスへのパラメータおよび戻り値として使用できるデータ型を制限しています。

次の表に、サポートされている Java データ型と相当する XML との間のマッピングを示します。

表 2-1 サポートされる Java データ型

Java データ型	対応する XML データ型
int	int
boolean	boolean
float	float
long	long
short	short
double	double
java.lang.Integer	int
java.lang.Boolean	boolean
java.lang.Float	float
java.lang.Long	long
java.lang.Short	short
java.lang.Double	double
java.lang.String	string
java.math.BigDecimal	decimal
java.util.Date	dateTime
byte[]	base64Binary
java.lang.String	string
この表内のサポートされている Java データ型のプロパティ、または別の JavaBean のプロパティを持つ JavaBean。	この表内のサポートされている XML データ型のメンバ、または別の複合構造のメンバを持つ複合構造。

表 2-1 サポートされる Java データ型

Java データ型	対応する XML データ型
この表にリストされているサポートされている Java データ型の配列（java.lang.Integer などのプリミティブ型と等価の参照を除く）。 一次元の配列のみ。	この表にリストされているサポートされている XML データ型の SOAP 配列。 一次元の配列のみ。
org.w3c.dom.Document	同等の XML なし
org.w3c.dom.DocumentFragment	同等の XML なし
org.w3c.dom.Element	同等の XML なし

次の表に、サポートされている XML データ型とそれに相当する Java との間のマッピングを示します。

表 2-2 XML から Java へのマッピング

XML データ型	対応する Java データ型
int	java.lang.Integer
boolean	java.lang.Boolean
float	java.lang.Float
long	java.lang.Long
short	java.lang.Short
double	java.lang.Double
decimal	java.math.BigDecimal
dateTime	java.util.Date
timeInstant	java.util.Date
byte	java.lang.Byte
base64Binary	byte[]

表 2-2 XML から Java へのマッピング

XML データ型	対応する Java データ型
hexBinary	byte[]
この表内のサポートされている **XML** データ型のメンバ、または別の複合構造のメンバを持つ複合構造。	この表内のサポートされている Java データ型のプロパティ、または別の JavaBean のプロパティを持つ JavaBean。
この表にリストされているサポートされている **XML** データ型の SOAP 配列。 一次元の配列のみ。	この表にリストされているサポートされている Java データ型の配列（java.lang.Integer などのプリミティブ型と等価の参照を除く）。 一次元の配列のみ。

WebLogic Web サービスでの XML と Java 間の変換

WebLogic Web サービスは、次の 2 つのエンコーディング スタイルをサポートしています。

- http://schemas.xmlsoap.org/soap/encoding/
- http://xml.apache.org/xml-soap/literalxml

注意 : 上記の URI は、ブラウザで実際には呼び出すことはできません。URI を使用したエンコーディング スタイルを示す標準の規則です。

WebLogic Web サービスがクライアントからデータを受信すると、SOAP メッセージで指定されているエンコーディング スタイルを使用してパラメータまたはメッセージのデータ型を識別して正しい Java オブジェクトに変換できるようにします。

注意 : WebLogic で生成した Java クライアント JAR ファイルを使用して Java クライアントを作成する場合、Java クライアント JAR ファイルには処理のためのコードが含まれているので特定のエンコーディング スタイルの知

識は必要ありません。この節は、WebLogic Web サービスを呼び出す Java 以外のクライアントを作成し、エンコーディング スタイルの処理方法を理解する必要があるプログラマのための説明です。

SOAP パケットが SOAP エンコーディング スタイルを指定する場合、Web サービスは SOAP メッセージの本文の XML データを 2-9 ページの「WebLogic Web サービスのパラメータおよび戻り値でサポートされているデータ型」にリストされている Java データ型の 1 つに変換します。

変換が失敗した場合（たとえば、パラメータの 1 つに対応する Java データ型が定義されていない場合）は、Web サービスは SOAP 障害を Web サービスを呼び出したクライアントに返します。

XML から Java への変換が正常に行われた場合は、Web サービスのスタイルによって次の異なるアクションが行われます。

- RPC スタイル Web サービスでは、結果 Java オブジェクトを適切なステートレス セッション EJB に渡します。
- メッセージスタイル Web サービスでは、Java オブジェクトを JMS `javax.jms.ObjectMessage` データ型にラップし、メッセージを適切な JMS の送り先に配置します。

SOAP パケットがリテラル XML エンコーディング スタイルを指定する場合、Web サービスでは Web サービスが RPC スタイルかメッセージスタイルかに応じて XML メッセージの本文の XML データを `org.w3c.dom.Element` データ型に変換して、ドキュメントをステートレス セッション EJB に送信するか、またはドキュメントを `javax.jms.ObjectMessage` データ型にラップしてメッセージを適切な JMS の送り先に配置します。

WebLogic Web サービスがデータをクライアントに戻したときには逆の動作が行われます。`org.w3c.dom.Element` 戻り値は、クライアントに戻される前にリテラル XML エンコーディング スタイルを使用してエンコードされ、その他の Java データ型は SOAP エンコーディング スタイルを使用してエンコードされます。

セキュリティの問題

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。

- SOAP サーブレット
- RPC スタイル Web サービスの基礎となるステートレス セッション EJB

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。ベーシック HTTP 認証および SSL の一般的な情報については、『WebLogic Security プログラマーズ ガイド』を参照してください。

WebLogic の Web サービスを呼び出すクライアントがデジタル証明書の提示を必要とするように相互 SSL を実装する方法については、2-16 ページの「WebLogic Web サービスを呼び出すときの相互 SSL の使い方」を参照してください。

メッセージスタイル Web サービスのセキュリティ設定

クライアントとサービス間の SOAP メッセージを処理する SOAP サーブレットのセキュリティを設定することによって、メッセージスタイル Web サービスのセキュリティを設定します。

注意 : また、このメソッドを使用して RPC スタイル Web サービスのセキュリティを設定することもできますが、2-16 ページの「RPC スタイル Web サービスのセキュリティ設定」で説明しているように、BEA では代わりに EJB にセキュリティを設定することをお勧めします。

`wsgen` Ant タスクを使用するかまたは手動で WebLogic Web サービスをアセンブルするときに、Web アプリケーションの `web.xml` ファイル内の SOAP サーブレットを参照します。これらの SOAP サーブレットでは、WebLogic Server とクライアント アプリケーション間の SOAP メッセージ を処理します。これらは常に WebLogic Server 上でデプロイされ、デプロイされたすべての WebLogic Web サービスで共有されます。

Web サービスによって参照される特定の SOAP サーブレットは、そのタイプ（RPC スタイルまたはメッセージスタイル）によって決定されます。次のリストは、各 SOAP サーブレットについて説明しています。

- `weblogic.soap.server.servlet.DestinationSendAdapter` — クライアント アプリケーションから JMS の送り先にデータを受信するメッセージスタイル Web サービス内の SOAP メッセージを処理します。
- `weblogic.soap.server.servlet.QueueReceiveAdapter` — 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送信するメッセージスタイル Web サービス内の SOAP メッセージを処理します。
- `weblogic.soap.server.servlet.TopicReceiveAdapter` — JMS トピックからクライアント アプリケーションにデータを送信するメッセージスタイル Web サービス内の SOAP メッセージを処理します。
- `weblogic.soap.server.servlet.StatelessBeanAdapter` — RPC スタイル Web サービスとクライアント アプリケーション間の SOAP メッセージを処理します。

たとえば、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メッセージスタイル Web サービスを作成した場合には、SOAP メッセージを処理する SOAP サーブレットは `weblogic.soap.server.servlet.DestinationSendAdapter` です。Web サービスをアセンブルするために使用された `wsgen` Ant タスクは、次の要素を Web アプリケーションの `web.xml` デプロイメント記述子に追加します。

```
<servlet>
  <servlet-name>sender</servlet-name>
  <servlet-class>
     weblogic.soap.server.servlet.DestinationSendAdapter
  </servlet-class>
  <init-param>
    <param-name>topic-resource-ref</param-name>
    <param-value>senderDestination</param-value>
  </init-param>
  <init-param>
    <param-name>connection-factory-resource-ref</param-name>
    <param-value>senderFactory</param-value>
  </init-param>
</servlet>
...

<servlet-mapping>
  <servlet-name>sender</servlet-name>
```

```
  <url-pattern>/sendMsg</url-pattern>
</servlet-mapping>
```

DestinationSendAdapter SOAP サーブレットへのアクセスを制限するには、セキュリティ レルムの 1 つまたは複数のプリンシパルにマップされるロールを定義してから、この SOAP サーブレットに適用するセキュリティ制約を次の web-resources-collection 要素内の url-pattern 要素を Web アプリケーションの web.xml デプロイメント記述子に追加することによって指定します。

```
<url-pattern>/sendMsg</url-pattern>
```

wsgen Ant タスクで作成されたエンタープライズ アプリケーション アーカイブの構造の詳細については、付録 C「Web サービス アーカイブ ファイルの手動によるアセンブル」を参照してください。

サーブレットへのアクセスを制限する手順の詳細については、『Web アプリケーションのアセンブルとコンフィグレーション』を参照してください。

RPC スタイル Web サービスのセキュリティ設定

Web サービスを実装するステートレス セッション EJB へのアクセスを制限することによって RPC スタイル Web サービスへのアクセスを制限します。

RPC スタイル 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、常に Web アプリケーションおよび SOAP サーブレットへアクセスできますが、EJB を呼び出せない可能性があります。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。

EJB へのアクセスの制限については、『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。

WebLogic Web サービスを呼び出すときの相互 SSL の使い方

相互 SSL では、WebLogic Web サービスを呼び出すクライアント アプリケーションは、自分のデジタル証明書を WebLogic Server に提示する必要があります。WebLogic Server は、信頼性のある認証局のリストと照らし合わせてデジタル証明書を検証します。

相互 SSL を使用して WebLogic Web サービスを呼び出す Java クライアントの作成は、以下の手順で行います。

1. 相互 SSL プロトコル（相互認証とも呼ばれる）および証明書に基づく認証を利用するように WebLogic Server をコンフィグレーションします。

   詳細については、『WebLogic Server 管理者ガイド』の「SSL プロトコルのコンフィグレーション」および「相互認証のコンフィグレーション」を参照してください。

2. クライアント アプリケーションで、Web サービスのルックアップに使用するコンテキストを取得する処理の前に、以下の Java コードを追加します。

```
System.out.println("********************** loading client certs");

InputStream certs[] = new InputStream[3];
certs[0]=new PEMInputStream(new FileInputStream("sample_key.pem"));
certs[1]=new PEMInputStream(new FileInputStream("sample_cert.pem"));
certs[2]=new PEMInputStream(new FileInputStream("sample_ca.pem"));

h.put(SoapContext.SSL_CLIENT_CERTIFICATE, certs);

String prov = "weblogic.net";

String s = System.getProperty("java.protocol.handler.pkgs");
if (s == null) {
    s = prov;
} else if (s.indexOf(prov) == -1) {
    s += "|" + prov;
}

System.setProperty("java.protocol.handler.pkgs", s);
```

このコードについて説明します。

- `sample_key.pem` は、証明書に関連付けられているクライアントのプライベート キーを含むファイルの名前です。
- `sample_cert.pem` は、クライアントの証明書を含むファイルの名前です。
- `sample_ca.pem` は、クライアントの証明書を発行した認証局の証明書を含むファイルの名前です。

注意： SSL 接続を確立するときは、デジタル証明書の主体の DN が、SSL 接続を開始するサーバのホスト名と一致している必要があります。一致していないと、SSL 接続は確立されません。WebLogic Server に付属するデモ用証明書を使用している場合は、ホスト名が一致しません。

これを防ぐには、クライアント アプリケーションを実行するときに、またはこの設定を常に有効にする場合は WebLogic Server を起動するときにも、`-Dweblogic.security.SSL.ignoreHostnameVerification=true` フラグを使用します。このフラグを設定すると、主体の DN とホスト名を比較するホスト名検証が無効になります。この方法は、開発環境でのみ推奨されます。発信 SSL 接続を行うサーバ用に新しいデジタル証明書を取得する方が、対応策として一層安全です。

WebLogic Web サービスの実装

WebLogic Web サービスの実装では、Web サービスのエントリ ポイントとして定義されるステートレス セッション EJB（RPC スタイル Web サービス用）または JMS リスナ（メッセージスタイル Web サービス用）の Java コードを記述します。JMS リスナは一般的に、メッセージ駆動型 Bean です。ステートレス セッション EJB または JMS リスナでは、すべての Web サービス機能を含んでいるか、または他の EJB を呼び出して作業を分配する場合があります。

ここでは、2-3 ページの「WebLogic Web サービスの設計」で説明した設計の問題を読み、理解していること、Web サービスを設計し終えていること、コード化する必要があるコンポーネントのタイプを理解していることを前提としています。

RPC スタイル Web サービスの実装

RPC スタイル Web サービスを実装するには、ステートレス セッション EJB 用の Java コードを記述します。2-9 ページの「WebLogic Web サービスのパラメータおよび戻り値でサポートされているデータ型」にリストされている EJB のパラメータおよび戻り値としてサポートされている Java データ型のみを使用してください。

ステートレス セッション EJB 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。

メッセージスタイル Web サービスの実装

メッセージスタイル Web サービスには、2-6 ページの「メッセージスタイル Web サービスと JMS」に説明されているように Web サービスを呼び出すクライアントから XML データを受信するものと、XML データをクライアントに送信するものの 2 つのタイプがあります。

メッセージスタイル Web サービスを実装するには、次の手順に従います。

1. 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）。2-21 ページの「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、2-20 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。JMS の一般的な情報については、『WebLogic Server 管理者ガイド』を参照してください。

2. XML データをクライアントから受信するメッセージスタイル Web サービス用の JMS の送り先からメッセージを取り出すか、または XML データをクライアントに送信するメッセージスタイル Web サービス用の JMS の送り先にメッセージを送信する J2EE コンポーネント（一般的にメッセージ駆動型 Bean）の Java コードを記述します。

   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。

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、『WebLogic Server 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、4-1 ページの「Administration Console の起動」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. JMS サーバ 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。

WebLogic Web サービスのアセンブル

この節では、Web サービスのすべてのコンポーネントをアセンブルし、WebLogic Server にデプロイしてリモート クライアントでアクセスできるようにする方法について説明します。

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

WebLogic Web サービスのアセンブルでは、RPC スタイル Web サービスを実装する EJB、サポート EJB、SOAP サーブレットを含む Web アプリケーションなどの Web サービスのすべてのコンポーネントをエンタープライズ アプリケーション アーカイブ（`*.ear`）にパッケージ化して、WebLogic Server にデプロイできるようにします。

開発者は、`wsgen` と呼ばれる Java Ant タスクを使用して WebLogic Web サービスをアセンブルします。`wsgen` Ant タスクでは、SOAP サーブレットおよびエンタープライズ アプリケーション アーカイブに関する `application.xml` ファイルを含む Web アプリケーションなど、ほとんどの WebLogic Web サービス コンポーネントを生成します。先に作成する必要がある唯一のコンポーネントは、Web サービスを実装する EJB またはメッセージ駆動型 Bean です。

Ant の一般的な情報については、http://jakarta.apache.org/ant/index.html を参照してください。

注意： WebLogic Server に付属する Java Ant ユーティリティは、ANTCLASSPATH 変数を設定するときに、*BEA_HOME*`\bin` ディレクトリにあるコンフィグレーション ファイル `ant` (UNIX) または `ant.bat` (Windows) を使用します。*BEA_HOME* は、WebLogic Server のインストール ディレクトリです。ANTCLASSPATH 変数を変更する必要がある場合は、これらのファイルもそれに合わせて更新する必要があります。

WebLogic Web サービスを手動でアセンブルする詳しい手順については、付録 C「Web サービス アーカイブ ファイルの手動によるアセンブル」を参照してください。

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. 一時的なステージング ディレクトリを作成します。
2. RPC スタイル Web サービスをアセンブルしている場合は、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. 環境を設定します。

   Windows NT の場合は、`setEnv.cmd` コマンドを実行します。このコマンドは *BEA_HOME*`\config\`*domain* ディレクトリにあります。*BEA_HOME* は WebLogic Server がインストールされているディレクトリで、*domain* はドメインの名前です。

   UNIX の場合は、`setEnv.sh` コマンドを実行します。このコマンドは *BEA_HOME*`/config/`*domain* ディレクトリにあります。*BEA_HOME* は WebLogic Server がインストールされているディレクトリで、*domain* はドメインの名前です。

4. `build.xml` という名前のファイルを WebLogic Web サービスをアセンブルする Ant タスク要素を含むステージング ディレクトリ内に作成します。

   `build.xml` ファイルの作成の詳細については、2-23 ページの「Ant build.xml ファイルの例」を参照してください。

5. ディレクトリをステージング ディレクトリに変更して Ant ユーティリティを実行します。

   ```
   $ ant
   ```

   `wsgen` Ant タスクでは、ステージング ディレクトリ内にサービス コンポーネントを含む `*.ear` ファイルを作成します。以上でこの `*.ear` ファイルを WebLogic Server にデプロイできます。

Ant build.xml ファイルの例

WebLogic Server には、WebLogic Web サービスのコンポーネントをエンタープライズ アーカイブ ファイルに素早くアセンブルできるように wsgen Ant タスクが含まれています。

次の例は、1 つの RPC スタイル Web サービスと 2 つのメッセージスタイル Web サービス（メッセージの送信用と受信用）の 3 つの Web サービスをアセンブルする build.xml ファイルを示しています。表 2-3 では、ファイル要素について説明します。

コード リスト 2-1 WebLogic Web サービスをアセンブルする build.xml ファイルの例

```
<project name="myProject" default="wsgen">
   <target name="wsgen">
      <wsgen
             destpath="myWebService.ear"
             context="/myContext"
             protocol="http">
         <rpcservices path="myEJB.jar">
            <rpcservice
              bean="statelessSession"
              uri="/rpc_URI"/>
         </rpcservices>
         <messageservices>
           <messageservice
             name="sendMsgWS"
             action="send"
             destination="examples.soap.msgService.MsgSend"
             destinationtype="topic"
             uri="/sendMsg"
             connectionfactory="examples.soap.msgService.MsgConnectionFactory"/>
          <messageservice
             name="receiveMsgWS"
             action="receive"
             destination="examples.soap.msgService.MsgReceive"
             destinationtype="topic"
             uri="/receiveMsg"
             connectionfactory="examples.soap.msgService.MsgConnectionFactory"/>
      </messageservices>
      </wsgen>
   </target>
</project>
```

表 2-3 build.xml 例の説明

要素または属性	説明
wsgen 要素	Web サービスをアセンブルするために使用される wsgen Ant タスクを指定する。
destpath 属性	作成されるエンタープライズ アーカイブが myWebService.ear という名前になるように指定する。

要素または属性	説明
`context` 属性	Web サービスのコンテキスト ルートが `/myContext` という名前になるように指定する。このコンテキスト ルートは Web サービス用に生成された WSDL を表示するため、および Java クライアント JAR ファイルのダウンロードに使用される URL で後で使用する。
`protocol` 属性	クライアントが HTTP を使用してサービスを呼び出すように指定する。
`rpcservices` 要素	`/myContext` コンテキストと関連付けられている単一の RPC スタイル Web サービスを含む。
`path` 属性	EJB が `myEJB.jar` という名前の JAR ファイルにアーカイブされるように指定する。
`rpcservice` 要素	RPC スタイル Web サービスのプロパティを指定する。
`bean` 属性	RPC スタイル Web サービスを実装するステートレス セッション EJB の名前を `statelessSession` に指定する。 この名前は、EJB が収められている EJB アーカイブの `ejb-jar.xml` ファイル中の `ejb-name` 要素に対応する。EJB アーカイブのパスは、親の `rpcservices` 要素で `path` 属性を使って指定されている。
`uri` 属性	サービスの URI を `/rpc_URI` に指定する。この URI は、Web サービス の WSDL にアクセスするための URL で使用される。
`messageservices` 要素	/myContext コンテキストと関連付けられている 2 つのメッセージスタイル Web サービスを含む。
`messageservice` 要素	各メッセージスタイル Web サービスのプロパティを指定する。
`name` 属性	各サービスに固有の名前を割り当てる。`sendMsgWS` および `receiveMsgWS`。
`action` 属性	Web サービスを呼び出すクライアントが、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を指定する。最初のサービスは `send` を指定し、2 番目は、`receive` を指定する。

要素または属性	説明
destination 属性	メッセージを送信または受信する JMS の送り先の JNDI 名を指定する。最初のサービスは examples.soap.msgService.MsgSend を指定し、2 番目のサービスは examples.soap.msgService.MsgReceive を指定する。
destinationtype 属性	JMS の送り先がトピックまたはキューのどちらであるかを指定する。両方のサービスで topic を指定する。
uri 属性	サービスの URI がそれぞれ /sendMsg と /receiveMsg になるように指定する。URI は、Web サービス の WSDL への完全な URL を作成するために結合される。
connectionfactory 属性	JMS 接続を作成するために使用される接続ファクトリの JNDI 名を指定する。両方のサービスで同じファクトリの examples.soap.msgService.MsgConnectionFactory を使用する。

build.xml ファイルの要素および属性の詳細については、付録 B「build.xml の要素と属性」を参照してください。

build.xml Ant ビルド ファイルの作成

次の手順では、WebLogic Web サービスを正しくアセンブルするために build.xml ファイルに含める必要がある Ant タスク要素について説明します。前述の節の例をガイドとして使用してください。

次の手順で示される build.xml ファイルの要素および属性の詳細については、付録 B「build.xml の要素と属性」を参照してください。

BEA XML エディタを使用した build.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。

WebLogic Web サービスのアセンブル用の build.xml Ant ビルド ファイルを作成するには、次の手順に従ってください。

1. テキスト エディタを使用して build.xml という名前の空のファイルを作成します。
2. 次の 2 つの属性を持つ <project> 要素を 1 つ追加します。

- name ー プロジェクトの名前。
- default ー この属性を wsgen に設定します。

3. <project> 要素内で、1 つの属性（name）を持つ <target> 要素を追加し、name 属性を wsgen に設定します。
4. <target> 要素内で、次の属性を持つ <wsgen> 要素を追加します。
   - destpath
   - context
   - protocol
5. 1 つまたは複数の RPC スタイル Web サービスをアセンブルしている場合は、次の属性を持つ単一の <rpcservices> 要素を <wsgen> 要素内に追加します。
   - path
6. <rpcservices> 要素内で、アセンブルしている各 RPC スタイル Web サービスに対して次の属性を持つ <rpcservice> 要素を追加します。
   - bean
   - uri
7. 1 つまたは複数のメッセージスタイル Web サービスをアセンブルしている場合は、単一の <messageservices> 要素を <wsgen> 要素内に追加します。
8. <messageservices> 要素内で、アセンブルしている各メッセージスタイル Web サービスに対して、メッセージスタイル Web サービスに対して既に設定している JMS の送り先および接続ファクトリに関する次の属性を持つ <messageservice> 要素を追加します。
   - name
   - action
   - destination
   - destinationtype
   - uri
   - connectionfactory

動的または静的 WSDL

WebLogic Web サービスは、WSDL ファイルを JSP としてパブリッシュします。WSDL JSP は、特定の WebLogic Server のホストおよびポートをハードコード化したり、サービスをホストしている WebLogic Server に基づいてホストおよびポートを動的に生成したりできます。

一般的に、WebLogic Web サービスの WSDL でホストおよびポートを動的に生成する場合は、Web サービスのアセンブルに使用される `build.xml` Ant ファイル内の `wsgen` 要素の `host` および `port` 属性を指定しないで設定します。ただし、ホストおよびポートを WSDL JSP 内でハードコード化する場合は、`host` および `port` 属性を明示的に指定します。

WebLogic Web サービスのデプロイ

WebLogic Web サービスをデプロイすることで、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。

エンタープライズ アプリケーションのデプロイの詳細については、『WebLogic Server 管理者ガイド』を参照してください。

WebLogic Web サービスのデプロイ : 簡単な例

この節では、製品例として *BEA_HOME*`\samples\examples\rpc` ディレクトリに収められているサンプルの RPC スタイル WebLogic Web サービスの開発、アセンブル、デプロイの開始から終了までのプロセスについて説明します。

サンプルの Weather RPC スタイル WebLogic Web サービスを開発するには、次の基本手順に従います。

1. 環境を設定します。

   Windows NT の場合は、`setEnv.cmd` コマンドを実行します。このコマンドは *BEA_HOME*\config*domain* ディレクトリにあります。*BEA_HOME* は WebLogic Server がインストールされているディレクトリで、*domain* はドメインの名前です。

   UNIX の場合は、`setEnv.sh` コマンドを実行します。このコマンドは *BEA_HOME*/config/*domain* ディレクトリにあります。*BEA_HOME* は WebLogic Server がインストールされているディレクトリで、*domain* はドメインの名前です。

2. Weather 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。

   詳細については、2-30 ページの「EJB 用の Java コードの記述」を参照してください。

3. EJB Java コードをクラス ファイルにコンパイルします。

4. EJB デプロイメント記述子を作成します。

   詳細については、2-34 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。

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

   詳細については、2-35 ページの「EJB のアセンブル」を参照してください。

6. WebLogic Web サービスをアセンブルするために使用する `build.xml` Java Ant ビルド ファイルを作成します。

   詳細については、2-36 ページの「build.xml ファイルの作成」を参照してください。

7. ステージング ディレクトリを作成します。

8. EJB `weather.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。

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

   ```
   $ ant
   ```

10. weather.ear アーカイブ ファイルを *BEA_HOME*\config*domain*\applications ディレクトリにコピーして、Weather Web サービスをテスト目的のために自動デプロイします。*BEA_HOME* はメイン WebLogic Server インストール ディレクトリで、*domain* はドメインの名前です。

Weather Web サービスを Java および Visual Basic クライアント アプリケーションの両方から呼び出すには、*BEA_HOME*\samples\examples\webservices\rpc\javaClient および *BEA_HOME*\samples\examples\webservices\rpc\vbClient の例を参照してください。

クライアント アプリケーションのビルドおよび実行手順については、ブラウザで Web ページの *BEA_HOME*\samples\examples\webservices\package-summary.html を呼び出してください。

EJB 用の Java コードの記述

サンプル Weather ステートレス セッション EJB には、1 つのパブリック メソッド（getTemp()）が含まれます。このメソッドは単一の引数と郵便番号を持ち、郵便番号が 90210 で戻り値が -273.15 以外の場合に、77 の float 値を返します。

注意： このメソッドでは、指定した郵便番号地域の実際の気温を返す Web サービスをシミュレーションしています。

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

```
package examples.webservices.rpc.weatherEJB;

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

/**
 * このインタフェース内のメソッドは WeatherBean 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 WeatherBean は、このインタフェースを実装し
 * Bean に委託する
 *
```

```
 * @author Copyright (c) 1998 by WebLogic, Inc. All Rights Reserved.
 * @author Copyright (c) 2001 by BEA Systems, Inc. All Rights Reserved.
 */

public interface Weather extends EJBObject {
  /**
   * 指定した ZipCode 地域の気温を取得
   *
   * @param ZipCode           String ZipCode
   * @return                  double Temperature
   * @exception               通信またはシステムに障害がある場合は
   *                          RemoteException を送出
   */

  public float getTemp(String ZipCode) 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.rpc.weatherEJB;

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

/**
 * WeatherBean 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) 1998 by WebLogic, Inc. All Rights Reserved.
 * @author Copyright (c) 2001 by BEA Systems, Inc. All Rights Reserved.
 */

public class WeatherBean 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。
 * ただし、この例では使用されていない
 *
 */
public void ejbActivate() {
  log("ejbActivate called");
}
/**
 * このメソッドは EJB 仕様では必須。
 * ただし、この例では使用されていない
 *
 */
public void ejbRemove() {
  log("ejbRemove called");
}
/**
 * このメソッドは EJB 仕様では必須。
 * ただし、この例では使用されていない
 *
 */
public void ejbPassivate() {
  log("ejbPassivate called");
}
/**
 * セッションの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>WeatherHome.create()</code> を呼び出すと、コンテナが 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
 *                          javax.ejb.CreateException を送出
 * @see                     examples.ejb.basic.statelessSession.Weather
 */
```

```
  public void ejbCreate () throws CreateException {
    log("ejbCreate called");
    try {
      InitialContext ic = new InitialContext();
    } catch (NamingException ne) {
      throw new CreateException("Failed to find environment value "+ne);
    }
  }
  /**
   * 指定した ZipCode 地域の気温を取得
   *
   * @param ZipCode           String ZipCode
   * @return                  float Temperature
   * @exception               通信またはシステムに障害がある場合は
   *                          RemoteException を送出
   */
  public float getTemp(String ZipCode) {
    log("getTemp called");
    Float result;


    if (ZipCode.equals("90210")) {
      result = new Float(77.0);
    } else {
      result = new Float(-273.15);
    }
    return result.floatValue();
  }
}
```

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

```
package examples.webservices.rpc.weatherEJB;

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


/**
 * このインタフェースは、WeatherBean.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 WeatherBeanC
 * 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 by WebLogic, Inc. All Rights Reserved.
```

```
 * @author Copyright (c) 2001 by BEA Systems, Inc. All Rights Reserved.
 */
public interface WeatherHome 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>WeatherHome.create()</code> を呼び出すと、コンテナが EJBean
   * のインスタンスを割り当てて <code>ejbCreate()</code> を呼び出す
   *
   * @return                     Weather
   * @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.ejb.basic.statelessSession.WeatherBean
   */
  Weather create() throws CreateException, RemoteException;
}
```

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、Weather 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>statelessSession</ejb-name>
                   <home>
                       examples.webservices.rpc.weatherEJB.WeatherHome
                   </home>
                   <remote>
                       examples.webservices.rpc.weatherEJB.Weather
```

```
                    </remote>
                    <ejb-class>
                        examples.webservices.rpc.weatherEJB.WeatherBean
                    </ejb-class>
                    <session-type>Stateless</session-type>
                    <transaction-type>Container</transaction-type>
                </session>
        </enterprise-beans>
        <assembly-descriptor>
                <container-transaction>
                    <method>
                       <ejb-name>statelessSession</ejb-name>
                       <method-intf>Remote</method-intf>
                       <method-name>*</method-name>
                    </method>
                    <trans-attribute>Required</trans-attribute>
                </container-transaction>
        </assembly-descriptor>
</ejb-jar>
```

次の例は、Weather 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 5.1.0 EJB//EN'
          'http://www.bea.com/servers/wls510/dtd/weblogic-ejb-jar.dtd'>

<weblogic-ejb-jar>
        <weblogic-enterprise-bean>
        <ejb-name>statelessSession</ejb-name>
                <caching-descriptor>
                    <max-beans-in-free-pool>100</max-beans-in-free-pool>
                </caching-descriptor>
                <jndi-name>statelessSession.WeatherHome</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 `weather.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 `weather.jar` アーカイブ ファイルを作成します。

   ```
   jar cvf weather.jar -C staging_dir .
   ```

build.xml ファイルの作成

BEA XML エディタを使用した `build.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 `build.xml` ファイルは、`weather.jar` アーカイブ ファイルを WebLogic Web サービス `weather.ear` エンタープライズ アプリケーション アーカイブ ファイルにアセンブルする `wsgen` Java ant タスクを例示します。

```
<project name="weather-webservice" default="wsgen">
    <target name="wsgen">
      <wsgen
         destpath="weather.ear"
         context="/weather">
        <rpcservices path="weather.jar">
                <rpcservice bean="statelessSession" uri="/weatheruri"/>
        </rpcservices>
      </wsgen>
    </target>
</project>
```

3 WebLogic 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 Web サービスを呼び出す方法について説明します。

- 3-2 ページの「WebLogic Web サービスの呼び出しの概要」
- 3-5 ページの「WebLogic Web サービス ホーム ページの呼び出し」
- 3-8 ページの「WebLogic Web サービスを呼び出して WSDL を取得する URL」
- 3-9 ページの「RPC スタイル WebLogic Web サービスを呼び出すクライアントの作成」
- 3-16 ページの「メッセージスタイル WebLogic Web サービスを呼び出す Java クライアントの作成」
- 3-23 ページの「WebLogic Web サービスからの例外の処理」
- 3-24 ページの「Web サービスを呼び出すための初期コンテキスト ファクトリ プロパティ」
- 3-25 ページの「WebLogic Web サービスを呼び出すクライアントに必要な追加クラス」

WebLogic Web サービスの呼び出しの概要

WebLogic 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。WebLogic Web サービスを呼び出すクライアント アプリケーションは、Java、Microsoft SOAP Toolkit などの任意のテクノロジを使用して記述できます。

クライアント アプリケーションは、呼び出す Web サービスに関する SOAP メッセージをアセンブルして必要なすべてのデータを SOAP メッセージの本文に含めます。次にクライアントは SOAP メッセージを HTTP/HTTPS を介して WebLogic Server に送信します。そこで Web サービスが実行され、SOAP メッセージが HTTP/HTTPS を介してクライアントに送り返されます。

注意 : クライアント アプリケーションを Java で記述する場合、WebLogic サーバはオプションの Java クライアント JAR ファイルを提供します。このファイルには WebLogic Web サービスを呼び出すために必要な WebLogic Web サービス クライアント API や WebLogic FastParser などが含まれます。他の Java WebLogic サーバ クライアントとは異なり、`weblogic.jar` ファイルを含める必要がなく、シンクライアントを作成できます。JAR ファイルのダウンロードの詳細については、3-7 ページの「Java クライアント JAR ファイルの Web サービス ホーム ページからのダウンロード」を参照してください。

各 Web サービスは独自のホーム ページを持っています。同じサーブレット コンテキストを共有する Web サービスはこの Web ページを共有します。この Web ページを使用して Web サービスの WSDL および Java クライアント JAR ファイルを取得します。この Web ページの詳細およびブラウザで呼び出す方法については、3-5 ページの「WebLogic Web サービス ホーム ページの呼び出し」を参照してください。

WebLogic Web サービス クライアント API

WebLogic サーバは、デプロイされた WebLogic Web サービスからダウンロード可能な Java クライアント JAR ファイル内にクライアント サイド Java SOAP API を持ちます。この API を使用して WebLogic 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します。このマニュアル内の例および製品に付属のサンプルでは、この API を使用しています。

警告： W3C または JavaSoft からの標準クライアント サイド Web Service API 仕様は利用できません。WebLogic Web サービス クライアント API が Java コミュニティ プロセスでは標準化されていないため、BEA Systems ではリリースごとに動作方法を変更する権利を保持し、下位互換性を保たない場合があります。

この章の例では、WebLogic Web サービス クライアント API のメイン クラス、インタフェース、およびメソッドについて簡単に説明します。API についての詳細なドキュメントは、「WebLogic Server API リファレンス」で `weblogic.soap` パッケージを検索してください。

WebLogic Web サービス クライアント API でサポートされているクライアント モード

WebLogic Web サービス クライアント API では、WebLogic 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次の 2 つのモードをサポートしています。

- **静的** : 静的クライアント アプリケーションは、Web サービスを構成する EJB および JavaBean インタフェースとクラスを明示的に使用します。これらのタイプのクライアント アプリケーションは、WebLogic サーバでサポートされている 2 つのモードの中で最も安全なタイプで、推奨されるタイプです。また、静的クライアント アプリケーションは、WebLogic 固有の Java コードを含みません。静的 Java クライアント アプリケーションの例については、3-10 ページの「静的 Java クライアントの記述」を参照してください。
- **動的** : 動的クライアント アプリケーションは、Web サービスの EJB インタフェースを明示的に参照しません。動的クライアント アプリケーションの例

については、3-13 ページの「動的 Java クライアントの記述」を参照してください。

この章で説明されている静的および動的クライアント アプリケーションはどちらも、Web サービスの WSDL を使用します。WSDL を使用しないクライアント アプリケーションの作成については、付録 D 「WSDL ファイルを使用しない Web サービスの呼び出し」を参照してください。

RPC スタイル Web サービスまたはメッセージスタイル Web サービスのいずれかを呼び出すために、静的クライアント アプリケーションと動的クライアント アプリケーションのどちらも使用できます。

WebLogic Web サービスを呼び出すクライアントの例

WebLogic Server には、RPC スタイル Web サービスおよびメッセージスタイル Web サービスを作成する両方の例と、Web サービスを呼び出す Java および Microsoft VisualBasic クライアント アプリケーションの両方の例があります。

これらの例は、`*BEA_HOME*\samples\examples\webservices` ディレクトリにあります。`*BEA_HOME*` は、メイン WebLogic Server インストール ディレクトリを指します。RPC スタイル Web サービスの例は `rpc` ディレクトリにあり、メッセージスタイル Web サービスの例は `message` ディレクトリにあります。

例を作成して実行する方法の詳細については、ブラウザで Web ページ `*BEA_HOME*\samples\examples\webservices\package-summary.html` を呼び出してください。

WebLogic Web サービス ホーム ページの呼び出し

WebLogic Web サービス ホーム ページは、特定のサーブレット コンテキスト用に定義された Web サービスを WSDL ファイルおよび各 Web サービスに関連付けられている Java クライアント JAR ファイルとともに一覧表示します。

次のテンプレート URL を使用してブラウザで WebLogic Web サービス ホーム ページを呼び出します。

```
[protocol]://[host]:[port]/[context]/index.html
```

この URL の説明は以下のとおりです。

- *protocol* は、Web サービスのビルドに使用される `build.xml` Ant ファイルの `<wsgen>` 要素の `protocol` 属性を指します。有効な値は、`http`（デフォルト）または `https` です。
- *host* は、Web サービスをホストしているコンピュータのホスト名を指します。
- *port* は、Web サービスをホストしている WebLogic Server インスタンスのポート番号を指します。
- *context* は、Web サービスのビルドに使用される `build.xml` Ant ファイルの `<wsgen>` 要素の `context` 属性を指します。

たとえば、次の `build.xml` ファイルを使用して Web サービスをビルドするとします。

```
<project name="myProject" default="wsgen">
   <target name="wsgen">
      <wsgen
            destpath="myWebService.ear"
            context="/myContext"
            protocol="http">
         <rpcservices path="myEJB.jar">
            <rpcservice
              bean="statelessSession"
              uri="/rpc_URI"/>
         </rpcservices>
         <messageservices>
           <messageservice
```

```
            name="sendMsgWS"
            action="send"
            destination="examples.soap.msgService.MsgSend"
            destinationtype="topic"
            uri="/sendMsg"
            connectionfactory="examples.soap.msgService.MsgConnectionFactory"/>
         <messageservice
            name="receiveMsgWS"
            action="receive"
            destination="examples.soap.msgService.MsgReceive"
            destinationtype="topic"
            uri="/receiveMsg"
            connectionfactory="examples.soap.msgService.MsgConnectionFactory"/>
      </messageservices>
      </wsgen>
   </target>
</project>
```

デフォルト ポート 7001 の myHost ホスト上の /myContext コンテキスト用の WebLogic Web サービス ホーム ページを呼び出す URL は、次のようになります。

```
http://www.myHost.com:7001/myContext/index.html
```

Web サービス ホーム ページからの WSDL の取得

Web サービス ホーム ページから Web サービスの WSDL を取得するには、次の手順に従います。

1. 3-5 ページの「WebLogic Web サービス ホーム ページの呼び出し」に記載されている手順に従って ブラウザでコンテキスト用の Web サービス ホーム ページを呼び出します。
2. Web サービスの名前をクリックします。
3. [**WSDL**] リンクをクリックします。指定した Web サービスの WSDL ファイルがブラウザにプレーン テキストで表示されます。

Java クライアント JAR ファイルの Web サービス ホーム ページからのダウンロード

WebLogic Server は、WebLogic 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ために必要な Java コード のほとんどを含む Java クライアント JAR ファイルを提供します。特に、JAR ファイルには クライアント サイド SOAP API の WebLogic 実装が含まれ、SOAP メッセージの作成および処理に低レベル Java コードを記述する必要がありません。

Java クライアント JAR ファイルには、次のオブジェクトが含まれます。

- WebLogic FastParser（ハイパフォーマンス XML パーサ）
- WebLogic Web サービス クライアント API
- RPC スタイル Web サービスを実装するステートレス セッション EJB のリモート インタフェース
- EJB パラメータまたは戻り値として使用される JavaBeans 用のクラス ファイル
- Web サービスのアセンブルに使用される `build.xml` Java ANT ビルド ファイルの `clientjar` 要素によって指定される追加クラス ファイル

注意： BEA では現在、クライアント機能をサーバ機能と別個にはライセンスしていないため、必要な場合には、この Java クライアント JAR ファイルをユーザ自身の顧客に再配布できます。

Java クライアント JAR ファイルをコンピュータにダウンロードするには、次の手順に従います。

1. 3-5 ページの「WebLogic Web サービス ホーム ページの呼び出し」に記載されている手順に従って、指定したコンテキスト用の Web サービス ホーム ページをブラウザで呼び出します。
1. Web サービスの名前をクリックします。
2. [**client.jar**] リンクをクリックします。
3. ローカル コンピュータ上で Java クライアント JAR ファイルを格納するディレクトリを指定します。

4. JAR ファイルを指定したディレクトリに格納します。
5. CLASSPATH を更新して Java クライアント JAR ファイルを含めます。

WebLogic Web サービスを呼び出して WSDL を取得する URL

WSDL は、呼び出す Web サービスを記述するためにクライアント アプリケーションによって使用されます。

WebLogic Web サービスの WSDL にアクセスするための URL は、次のとおりです。

```
[protocol]://[host]:[port]/[context]/[WSname]/[WSname].wsdl
```

この URL の説明は以下のとおりです。

- *protocol* は、Web サービスのビルドに使用される `build.xml` Ant ファイルの `<wsgen>` 要素の `protocol` 属性を指します。デフォルトでは、この値は `http` です。
- *host* は、Web サービスをホストしているコンピュータのホスト名を指します。
- *port* は、Web サービスをホストしている WebLogic Server インスタンスのポート番号を指します。
- *context* は、Web サービスのビルドに使用される `build.xml` Ant ファイルの `<wsgen>` 要素の `context` 属性を指します。
- *WSname* は、Web サービスの名前です。
  - RPC スタイル Web サービスの場合、Web サービスを実装するステートレス セッション EJB の JNDI 名です。

    たとえば、`build.xml` ファイルの `bean` 属性が `statelessSession` を指定した場合、`weblogic-ejb-jar.xml` には次のエントリがあります。

    ```
    <weblogic-enterprise-bean>
      <ejb-name>statelessSession</ejb-name>
    ```

```
  <jndi-name>statelessSession.WeatherHome</jndi-name>
</weblogic-enterprise-bean>
```

WSname 値は statelessSession.WeatherHome になります。

- メッセージスタイル Web サービスの場合、Web サービスの名前は build.xml ファイル内で Web サービスを定義する messageservice 要素の name 属性によって指定されます。

たとえば、3-5 ページの「WebLogic Web サービス ホーム ページの呼び出し」にリストされている build.xml サンプル ファイルを使用すると、RPC スタイル Web サービスの WSDL にアクセスする URL は次のようになります。

```
http://www.myHost.com:7001/myContext/statelessSession.WeatherHome/statelessSess
ion.WeatherHome.wsdl
```

同様に、2 つのメッセージスタイル Web サービス用の WSDL にアクセスする URL は、次のようになります。

```
http://www.myHost.com:7001/myContext/sendMsgWS/sendMsgWS.wsdl
http://www.myHost.com:7001/myContext/receiveMsgWS/receiveMsgWS.wsdl
```

RPC スタイル WebLogic Web サービスを呼び出すクライアントの作成

この節では、RPC スタイル WebLogic Web サービスを Java および Microsoft SOAP ToolKit の 2 つのタイプのクライアントから呼び出す方法について説明します。

この節で示す例では、examples.ejb.basic.statelessSession WebLogic Server 例で説明されている Trader ステートレス セッション EJB に基づいた RPC スタイル Web サービスを呼び出します。

Java クライアントの記述

必要な Java コード のほとんどすべてが WebLogic Server で提供され、クライアント コンピュータにダウンロードする Java クライアント JAR ファイルにパッケージ化されているため、WebLogic 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は簡単です。

この節では、クライアント アプリケーションの静的および動的の 2 つのモードについて説明します。EJB と JavaBean パラメータおよび戻り値の型の Java インタフェースがあり、クライアント Java コード内で直接使用する場合は、静的クライアントを使用します。インタフェースがない場合は、動的クライアントを使用します。

静的 Java クライアントの記述

次の例では、WebLogic Server の `examples.ejb.basic.statelessSession` EJB のサンプルに基づいた RPC スタイル Web サービスを呼び出す簡単な静的 Java クライアントを示します。

この例では、Web サービスの WSDL の取得に URL の http://www.myhost.com:7001/myContext/statelessSession/statelessSession.wsdl を使用します。この URL の作成方法および RPC スタイル Web サービスの作成に使用される `build.xml` ファイルの例については、3-8 ページの「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。

例の後に示す手順では、このクライアントを作成するために実行する基本手順で例に関連する部分について説明します。

```
import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;

import examples.ejb.basic.statelessSession.Trader;
import examples.ejb.basic.statelessSession.TradeResult;

public class Client{

  public static void main( String[] arg ) throws Exception

    Properties h = new Properties();

    h.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.soap.http.SoapInitialContextFactory");
```

```
    h.put("weblogic.soap.wsdl.interface",
         Trader.class.getName() );

    Context context = new InitialContext(h);

    Trader service = (Trader)context.lookup(
 "http://www.myHost.com:7001/myContext/statelessSession/statelessSession.wsdl"
);

    TradeResult result = (TradeResult)service.buy( "BEAS", 100 );

    System.out.print( result.getStockSymbol() );
    System.out.print( ":" );
    System.out.println( result.getNumberTraded() );
  }
}
```

WebLogic Web サービスを静的に呼び出す Java コードは、EJB を呼び出すリモート メソッド呼び出し（RMI）クライアント コードと似ています。主な違いは次のとおりです。

- サービスのホーム インタフェースをルックアップして呼び出す必要がありません。
- Web サービス クライアントは SOAP 固有の INITIAL_CONTEXT_FACTORY を使用します。
- Web サービス クライアントはパラメータ内のインタフェースを INITIAL_CONTEXT_FACTORY に指定します。

RPC スタイル WebLogic Web サービスを呼び出す静的 Java クライアントを作成するには、次の手順に従います。

1. WebLogic Web サービスをホストする WebLogic Server から、Java クライアント JAR ファイルを取得します。

   この手順の詳細については、3-7 ページの「Java クライアント JAR ファイルの Web サービス ホーム ページからのダウンロード」を参照してください。

2. クライアント コンピュータ上の CLASSPATH に、Java クライアント JAR ファイルを追加します。

3. クライアント Java プログラムを作成します。以降の手順では、Java コードの Web サービスに固有の部分について説明します。

a. クライアント アプリケーションの `main` メソッド内で、次の Java コードを追加してクライアントを初期化し、Web サービスと対話できるようにします。

```
Properties h = new Properties();

h.put(Context.INITIAL_CONTEXT_FACTORY,
     "weblogic.soap.http.SoapInitialContextFactory");

h.put("weblogic.soap.wsdl.interface",
     Trader.class.getName() );

Context context = new InitialContext(h);

Trader service = (Trader)context.lookup(
"http://www.myHost.com:7001/myContext/statelessSession/state
lessSession.wsdl" );
```

この例では、`Trader` は EJB へのパブリック インタフェースです。`context.lookup()` メソッドで使用される URL の作成方法の詳細については、3-8 ページの「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。

b. 次の例に示されているように EJB のパブリック メソッドを実行して Web サービス操作を呼び出します。

```
TradeResult result = (TradeResult)service.buy( "BEAS", 100 );
```

クライアントは、`Trader` EJB の `buy()` メソッドを実行します。戻り値は `TraderResult` JavaBean オブジェクトです。`Trader` EJB のパブリック メソッドを見つけるには、Web サービスの返された WSDL を検証するか、またはダウンロードした Java クライアント JAR ファイルを unJAR して **javap** ユーティリティを使用して `Trader` インタフェースのメソッドをリストします。

c. 返された `TraderResult` JavaBean の `get` メソッドを使用して返された結果を取得します。`TraderResult` クラスのメソッドを見つけるには、Java クライアント JAR ファイルを unJAR して **javap** ユーティリティを使用して `TraderResult` クラスのメソッドをリストします。

```
System.out.print( result.getStockSymbol() );
System.out.print( ":" );
System.out.println( result.getNumberTraded() );
```

4. 通常どおりにクライアント Java プログラムをコンパイルして実行します。

動的 Java クライアントの記述

次の例では、WebLogic Server の examples.ejb.basic.statelessSession EJB のサンプルに基づいた RPC スタイル Web サービスを呼び出す簡単な動的 Java クライアントを示します。

この例では、Web サービスの WSDL の取得に URL の http://www.myhost.com:7001/myContext/statelessSession/statelessSession.wsdl を使用します。この URL の作成方法および RPC スタイル Web サービスの作成に使用される build.xml ファイルの例については、3-8 ページの「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。

例の後に示す手順では、このクライアントを作成するために実行する基本手順で例に関連する部分について説明します。

```
import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;

import examples.ejb.basic.statelessSession.TradeResult;

import weblogic.soap.WebServiceProxy;
import weblogic.soap.SoapMethod;

public class DynamicClient{

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

    Properties h = new Properties();

    h.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.soap.http.SoapInitialContextFactory");

      Context context = new InitialContext(h);

    WebServiceProxy proxy = (WebServiceProxy)context.lookup(

"http://www.myHost.com:7001/myContext/statelessSession/statelessSession.wsdl" );

    SoapMethod method = proxy.getMethod( "buy" );

    TradeResult result = (TradeResult)method.invoke(
         new Object[]{ "BEAS", new Integer(100) } );

    System.out.print( result.getStockSymbol() );
    System.out.print( ":" );
    System.out.println( result.getNumberTraded() );
```

```
  }
}
```

RPC スタイル WebLogic Web サービスを呼び出す動的 Java クライアントを作成するには、次の手順に従います。

1. WebLogic Web サービスをホストする WebLogic Server から、Java クライアント JAR ファイルを取得します。

   この手順の詳細については、3-7 ページの「Java クライアント JAR ファイルの Web サービス ホーム ページからのダウンロード」を参照してください。

2. クライアント コンピュータ上の CLASSPATH に、Java クライアント JAR ファイルを追加します。

3. クライアント Java プログラムを作成します。以降の手順では、Java コードの Web サービスに固有の部分について説明します。

   a. クライアント アプリケーションの `main` メソッド内で、次の Java コードを追加してクライアントを初期化し、Web サービスと対話できるようにします。

   ```
   Properties h = new Properties();

   h.put(Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.soap.http.SoapInitialContextFactory");

   Context context = new InitialContext(h);

   WebServiceProxy proxy = (WebServiceProxy)context.lookup(
   "http://www.myHost.com:7001/myContext/statelessSession/state
   lessSession.wsdl" );
   ```

   この例では、`context.lookup()` メソッドは、特定の `Trader` オブジェクトではなく汎用 `WebServiceProxy` オブジェクトを返します。これにより、`WebServiceProxy` が任意の EJB オブジェクトを表すことが可能になるため、例がより動的になります。`context.lookup()` メソッドで使用される URL の作成方法の詳細については、3-8 ページの「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。

   b. 次の例に示されているように EJB のパブリック メソッドを実行して Web サービス操作を呼び出します。

   ```
   SoapMethod method = proxy.getMethod( "buy" );

   TradeResult result = (TradeResult)method.invoke(
        new Object[]{ "BEAS", new Integer(100) } );
   ```

クライアントは、`invoke()` メソッドを使用して間接的に `Trader` EJB の `buy()` メソッドを実行します。戻り値は `TraderResult` JavaBean オブジェクトです。`Trader` EJB のパブリック メソッドを見つけるには、Web サービスの返された WSDL を検証するか、またはダウンロードした Java クライアント JAR ファイルを unJAR して **javap** ユーティリティを使用して `Trader` インタフェースのメソッドをリストします。

c. 返された `TraderResult` JavaBean の `get` メソッドを使用して返された結果を取得します。`TraderResult` クラスのメソッドを見つけるには、Java クライアント JAR ファイルを unJAR して **javap** ユーティリティを使用して `TraderResult` クラスのメソッドをリストします。

```
System.out.print( result.getStockSymbol() );
System.out.print( ":" );
System.out.println( result.getNumberTraded() );
```

4. 通常どおりにクライアント Java プログラムをコンパイルして実行します。

Microsoft SOAP Toolkit クライアントの記述

WebLogic Web サービスは、Microsoft SOAP ToolKit で提供されるクライアントサイド コンポーネントを使用して Microsoft Visual Basic アプリケーションから呼び出すことができます。

注意 : WebLogic Server 6.1 では、Microsoft SOAP ToolKit のバージョン 2.0sp2 のみをサポートしています。

次の Visual Basic のサンプル コードは、`examples.webservices.rpc` 例で記述される WebLogic Web サービスを呼び出す簡単な例を示します。

```
SET soapclient = CreateObject("MSSOAP.SoapClient")

Call soapclient.mssoapinit(

"http://myhost:7001/weather/statelessSession.WeatherHome/statelessSession.Weath
erHome.wsdl", "Weather", "WeatherPort")

wscript.echo soapclient.getTemp(94117)
```

Microsoft SOAP ToolKit を使用して WebLogic Web サービスを Visual Basic アプリケーションから呼び出すには、次の手順に従います。

1. `SoapClient` オブジェクトを Visual Basic アプリケーション内でインスタンス化します。
2. `SoapClient.mssoapinit()` メソッドを実行することによって 次のパラメータを渡し、`SoapClient` オブジェクトを初期化します。
   - WebLogic Web サービスの WSDL の URL。この URL 作成の詳細については、3-8 ページの「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。
   - WSDL ファイル内の `service` 要素の `name` 属性で識別される Web サービスの名前。
   - WSDL ファイル内の `port` 要素の `name` 属性で識別される Web サービスのポート。

   `SoapClient` オブジェクトを初期化した後で、WSDL 内で定義されているすべてのメソッドは `SoapClient` オブジェクトに動的にバインドされます。
3. WebLogic Web サービス メソッドを実行します。

メッセージスタイル WebLogic Web サービスを呼び出す Java クライアントの作成

この節では、Java クライアント アプリケーションからメッセージスタイル Web サービスを呼び出す方法について説明します。

必要な Java コード のほとんどすべてが WebLogic Server で提供され、クライアント コンピュータにダウンロードする Java クライアント JAR ファイルにパッケージ化されているため、メッセージスタイル WebLogic 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は簡単です。

この節では、データを WebLogic Server に送信するメッセージスタイル Web サービスを呼び出す Java クライアントと、データを受信するメッセージスタイル Web サービスを呼び出す Java クライアントの 2 種類について説明します。両方の例で、動的 Java クライアントの作成方法を示します。

注意 : 送受信アクションは、クライアント アプリケーションの視点から見ています。

例で示す 2 つのメッセージスタイル Web サービスは、次の build.xml ファイルを使用してアセンブルされていると想定します。

```
<project name="myProject" default="wsgen">
   <target name="wsgen">
     <wsgen
       destpath="messageExample.ear"
       context="/msg"
       protocol="http" >
       <messageservices>
         <messageservice
           action="send"
           name="Sender"
           destination="examples.soap.msgService.MsgSend"
           destinationtype="topic"
           uri="/sendMsg"
           connectionfactory="examples.soap.msgService.MsgConnectionFactory"/>
         <messageservice
           action="receive"
           name="Receiver"
           destination="examples.soap.msgService.MsgReceive"
           destinationtype="topic"
           uri="/receiveMsg"
           connectionfactory="examples.soap.msgService.MsgConnectionFactory"/>


         </messageservices>
      </wsgen>
   </target>
</project>
```

build.xml ファイルは、次の 2 つのメッセージスタイル Web サービスを示しています。クライアント アプリケーションが JNDI 名 examples.soap.msgService.MsgSend を使用して JMS トピックにデータを送信するために使用する Sender と、クライアント アプリケーションが JNDI 名 examples.soap.msgService.MsgReceive を使用して JMS トピックからデータを受信するために使用する Receiver の 2 つです。両方のメッセージスタイル Web サービスは、同じ ConnectionFactory（examples.soap.msgService.MsgConnectionFactory）を使用して JMS 接続を作成します。

メッセージスタイル Web サービスへのデータの送信

この節では、Web サービスを呼び出してデータを WebLogic Server に送信する動的 Java クライアント アプリケーションを作成する方法について説明します。要点を簡潔に示すため、この例ではデータを含める `String` データ型を送信します。

注意： `org.w3c.dom.Document`、`org.w3c.dom.DocumentFragment`、または `org.w3c.dom.Element` の各データ型を `send` メソッドに送信する方法を示すさらに複雑な例については、付録 D「WSDL ファイルを使用しない Web サービスの呼び出し」を参照してください。

データを WebLogic Server に送信するメッセージスタイル Web サービスは、単一のメソッド `send` を定義します。これは、ユーザの Java クライアント アプリケーションから呼び出す必要がある唯一のメソッドです。`send` メソッドで使用できるパラメータは実際のデータです。データ型は `String`（例で使用）、DOM ツリー、`InputStream` などの任意の型を使用できます。データは Web サービスをアセンブルするために使用される `build.xml` ファイル内に指定した JMS の送り先で最終的に終了します。

例では URL `http://localhost:7001/msg/Sender/Sender.wsdl` を使用して Web サービスの WSDL を取得します。この URL の作成方法の詳細については、3-8 ページの「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。

例の後の手順では、このクライアントを作成するために実行する基本手順の一部として、例に関連する節について説明します。

```
package examples.soap;

import java.util.Properties;
import java.net.URL;
import javax.naming.Context;
import javax.naming.InitialContext;

import weblogic.soap.WebServiceProxy;
import weblogic.soap.SoapMethod;
import weblogic.soap.SoapType;
import weblogic.soap.codec.CodecFactory;
import weblogic.soap.codec.SoapEncodingCodec;
```

```
public class ProducerClient{

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

    Properties h = new Properties();

    h.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.soap.http.SoapInitialContextFactory");
    h.put("weblogic.soap.verbose", "true" );

    CodecFactory factory = CodecFactory.newInstance();
    factory.register( new SoapEncodingCodec() );
    h.put( "weblogic.soap.encoding.factory", factory );

    Context context = new InitialContext(h);

    WebServiceProxy proxy = (WebServiceProxy)context.lookup(
    "http://localhost:7001/msg/Sender/Sender.wsdl" );
    SoapMethod method = proxy.getMethod( "send" );

    String toSend = arg.length == 0 ? "No arg to send" : arg[0];
    Object result = method.invoke( new Object[]{ toSend } );

  }

}
```

データを WebLogic Server に送信するメッセージスタイル WebLogic Web サービスを呼び出す動的 Java クライアントを作成するには、次の手順に従います。

1. WebLogic Web サービスをホストする WebLogic Server から、Java クライアント JAR ファイルを取得します。

   この手順の詳細については、3-7 ページの「Java クライアント JAR ファイルの Web サービス ホーム ページからのダウンロード」を参照してください。

2. クライアント コンピュータ上の CLASSPATH に、Java クライアント JAR ファイルを追加します。

3. クライアント Java プログラムを作成します。以降の手順では、Java コードの Web サービスに固有の部分について説明します。

   a. クライアント アプリケーションの `main` メソッドで、`Properties` オブジェクトを作成して初期コンテキスト プロパティを設定します。

```
Properties h = new Properties();

 h.put(Context.INITIAL_CONTEXT_FACTORY,
      "weblogic.soap.http.SoapInitialContextFactory");
 h.put("weblogic.soap.verbose", "true" );
```

b. エンコーディング スタイルのファクトリを作成して SOAP エンコーディング スタイルを登録します。

```
CodecFactory factory = CodecFactory.newInstance();
factory.register( new SoapEncodingCodec() );
h.put( "weblogic.soap.encoding.factory", factory );
```

c. 初期コンテキストを作成し、WSDL を使用して Web サービスをルックアップしてから `send` メソッドを取得します。

```
Context context = new InitialContext(h);

WebServiceProxy proxy = (WebServiceProxy)context.lookup(
"http://localhost:7001/msg/Sender/Sender.wsdl" );
SoapMethod method = proxy.getMethod( "send" );
```

d. `send` メソッドを呼び出して、データを Web サービスに送信します。この例では、クライアント アプリケーションは単純に最初の引数を受け取って `String` として送信します。ユーザーが特定の引数を指定しない場合、クライアント アプリケーションでは文字列 `No arg to send` を送信します。

```
String toSend = arg.length == 0 ? "No arg to send" : arg[0];
Object result = method.invoke( new Object[]{ toSend } );
```

4. 通常どおりにクライアント Java プログラムをコンパイルして実行します。

メッセージスタイル Web サービスからのデータの受信

この節では、Web サービスを呼び出してデータを WebLogic Server から受信する動的 Java クライアント アプリケーションを作成する方法について説明します。

データを WebLogic Server から受信するメッセージスタイル Web サービスは、`receive` という単一のメソッドを定義します。これは、ユーザの Java クライアント アプリケーションから呼び出す必要がある唯一のメソッドです。`receive` メソッドでは入力パラメータを使用できません。Web サービスをアセンブルするために使用された `build.xml` ファイル内に指定する JMS の送り先から Web サービスが取得したデータを含む汎用 Java オブジェクトを返します。

この節で示す例では URL の `http://localhost:7001/msg/Receiver/Receiver.wsdl` を使用して Web サービスの WSDL を取得します。この URL の作成方法の詳細については、3-8 ページの「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。

例の後の手順では、このクライアントを作成するために実行する基本手順の一部として、例に関連する節について説明します。

```
package examples.soap;

import java.util.Properties;
import java.net.URL;
import javax.naming.Context;
import javax.naming.InitialContext;

import weblogic.soap.WebServiceProxy;
import weblogic.soap.SoapMethod;
import weblogic.soap.SoapType;
import weblogic.soap.codec.CodecFactory;
import weblogic.soap.codec.SoapEncodingCodec;

public class ConsumerClient{

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

    Properties h = new Properties();
    h.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.soap.http.SoapInitialContextFactory");
    h.put("weblogic.soap.verbose", "true" );

    CodecFactory factory = CodecFactory.newInstance();
    factory.register( new SoapEncodingCodec() );
    h.put( "weblogic.soap.encoding.factory", factory );

    Context context = new InitialContext(h);

    WebServiceProxy proxy = (WebServiceProxy)context.lookup(
    "http://localhost:7001/msg/Receiver/Receiver.wsdl" );
    SoapMethod method = proxy.getMethod( "receive" );

    while( true ){
      Object result = method.invoke( null );
      System.out.println( result );
    }
  }
}
```

データを WebLogic Server から受信するメッセージスタイル WebLogic Web サービスを呼び出す動的 Java クライアントを作成するには、次の手順に従います。

1. WebLogic Web サービスをホストする WebLogic Server から、Java クライアント JAR ファイルを取得します。

   この手順の詳細については、3-7 ページの「Java クライアント JAR ファイルの Web サービス ホーム ページからのダウンロード」を参照してください。

2. クライアント コンピュータ上の CLASSPATH に、Java クライアント JAR ファイルを追加します。

3. クライアント Java プログラムを作成します。以降の手順では、Java コードの Web サービスに固有の部分について説明します。

   a. クライアント アプリケーションの `main` メソッドで、`Properties` オブジェクトを作成して初期コンテキスト プロパティを設定します。

   ```
   Properties h = new Properties();
   h.put(Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.soap.http.SoapInitialContextFactory");
   h.put("weblogic.soap.verbose", "true" );
   ```

   b. エンコーディング スタイルのファクトリを作成して SOAP エンコーディング スタイルを登録します。

   ```
   CodecFactory factory = CodecFactory.newInstance();
   factory.register( new SoapEncodingCodec() );
   h.put( "weblogic.soap.encoding.factory", factory );
   ```

   c. 初期コンテキストを作成し、WSDL を使用して Web サービスをルックアップしてから `receive` メソッドを取得します。

   ```
   Context context = new InitialContext(h);

   WebServiceProxy proxy = (WebServiceProxy)context.lookup(
   "http://localhost:7001/msg/Receiver/Receiver.wsdl" );
   SoapMethod method = proxy.getMethod( "receive" );
   ```

   d. `receive` メソッドを呼び出してデータを Web サービスから受信します。この例では、クライアント アプリケーションは無限 `while` ループを使用して連続して `receive` メソッドを呼び出します。つまり、メッセージの JMS の送り先をポーリングします。`receive` メソッドがデータを返すと、クライアント アプリケーションはその結果を標準の出力に出力します。

```
while( true ){
  Object result = method.invoke( null );
  System.out.println( result );
```

4. 通常どおりにクライアント Java プログラムをコンパイルして実行します。

WebLogic Web サービスからの例外の処理

WebLogic Server が Web サービスを実行中に例外が生成された場合、Web サービスを呼び出したクライアント アプリケーションが標準 SOAP 障害についての実行時 `weblogic.soap.SoapFault` 例外を受け取ります。

WebLogic Server の次のタイプの例外は、クライアント アプリケーション内で実行時 `SoapFault` 例外を生成する可能性があります。

- RPC スタイル Web サービスを実装するステートレス セッション EJB からの例外
- クライアント アプリケーションと WebLogic Web サービス間の SOAP メッセージ を処理する SOAP サーブレットからの例外
- JMS 例外

クライアント アプリケーションが `SoapFault` 例外を受け取った場合、`weblogic.soap.SoapFault` の次のメソッドを使用して検証してください。

- `getFaultCode()` ― SOAP faultcode を返します。
- `getFaultString()` ― WebLogic Server 内で例外を生成したクラスまたはインタフェースの名前を返します。たとえば、RPC スタイル Web サービスを構成するステートレス セッション EJB が例外を生成した場合、`getFaultString()` メソッドはこの EJB のインタフェースを返します。
- `printStackTrace()` ― 例外のスタック トレースを返します。

次の Java クライアント アプリケーションからの抜粋部分は、`weblogic.soap.SoapFault` を使用して WebLogic Server 上で発生したエラーを検証する例を示しています。

```
import weblogic.soap.SoapFault;

...
```

```
try {
   TradeResult result = (TradeResult)method.invoke(
      new Object[]{ "BEAS", new Integer(100) } );

   System.out.print( result.getStockSymbol() );
   System.out.print( ":" );
   System.out.println( result.getNumberTraded() );
} catch (SoapFault fault){
   System.out.println( "Ooops, got a fault: " + fault );
   fault.printStackTrace();
  }
```

Web サービスを呼び出すための初期コンテキスト ファクトリ プロパティ

次の表は、Java クライアント アプリケーション内で WebLogic で生成された Java クライアント JAR ファイルを使用して WebLogic Web サービスを呼び出すときに `Properties` オブジェクトを使用して設定できる Java プロパティを示しています。

注意： これらのプロパティは、初期コンテキスト ファクトリに渡されます。Java システム プロパティではありません。

表 3-1 Web サービスを呼び出すための初期コンテキスト ファクトリ プロパティ

プロパティ	説明
`weblogic.soap.wsdl.interface`	Web サービスの基礎になるステートレス セッション EJB のインタフェースを指定する。
`weblogic.soap.verbose`	`true` に設定されたときに、Java クライアントによって WebLogic Web サービスを呼び出すために生成された SOAP パケットは、クライアントへの出力になる。 有効な値は `true` および `false`（デフォルト）。
`weblogic.soap.encoding.factory`	XML と Java データ間で変換を行うエンコーダおよびデコーダを含む CodecFactory を指定する。 有効な値は `weblogic.soap.codec.CodecFactory` のインスタンス。

表 3-1 Web サービスを呼び出すための初期コンテキスト ファクトリ プロパティ（続き）

プロパティ	説明
`java.naming.factory.initial`	初期 SOAP コンテキスト ファクトリを指定する。 有効な値は `weblogic.soap.http.SoapInitialContextFactory` のインスタンス。
`java.naming.security.principal`	HTTP セキュリティを設定するときにユーザ名を指定する。
`java.naming.security.credentials`	HTTP セキュリティを設定するときにユーザ パスワードを指定する。

WebLogic Web サービスを呼び出すクライアントに必要な追加クラス

WebLogic Web サービスは、次の 2 つのエンコーディング スタイルをサポートしています。

- http://schemas.xmlsoap.org/soap/encoding/
- http://xml.apache.org/xml-soap/literalxml

Java クライアント アプリケーションが SOAP エンコーディングを使用する場合、WebLogic Server からダウンロードした Java クライアント JAR ファイルには WebLogic Web サービスを呼び出すために必要なすべてのクラスが含まれます。

ただし、クライアント アプリケーションが Apache のリテラル XML エンコーディングを使用している場合は、Java クライアント JAR ファイルには必要なファイルすべては含まれません。クライアント JAR ファイルのサイズはそれほど大きくないのが普通です。

以下に、追加で必要になる可能性があるクラスを示します。

- `weblogic.apache.Xerces.*`
- `weblogic.xml.jaxp.*`
- `org.w3c.dom.*`

- `org.w3c.sax.*`
- `javax.xml.parsers.*`

クライアント アプリケーションを実行するときに CLASSPATH 環境変数をそれぞれの場所に設定するか、または `wsgen` Java 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 `build.xml` ファイル内の `clientjar` 要素を使用して、これらのクラスを含めることができます。

クライアント アプリケーションで必要なクラスの完全なリストを取得するには、アプリケーションをコンパイルしてから `-verbose` フラグを指定して実行します。これにより必要なすべてのクラスがリストされます。

4 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。

- 4-1 ページの「WebLogic Web サービスの管理の概要」
- 4-2 ページの「WebLogic Server にデプロイされている Web サービスの表示」

WebLogic Web サービスの管理の概要

WebLogic Web サービスを開発、アセンブル、およびデプロイすると、Administration Console を使用して次の管理タスクを実行できます。

- WebLogic Server に現在デプロイされている Web サービスの表示

Administration Console の起動

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
```

この URL の説明は以下のとおりです。

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

次の図は、Administration Console のメイン ウィンドウを示しています。

WebLogic Server にデプロイされている Web サービスの表示

WebLogic Server 上にデプロイされているすべての Web サービスを表示して特定の Web サービスのプロパティを表示するには、次の手順に従います。

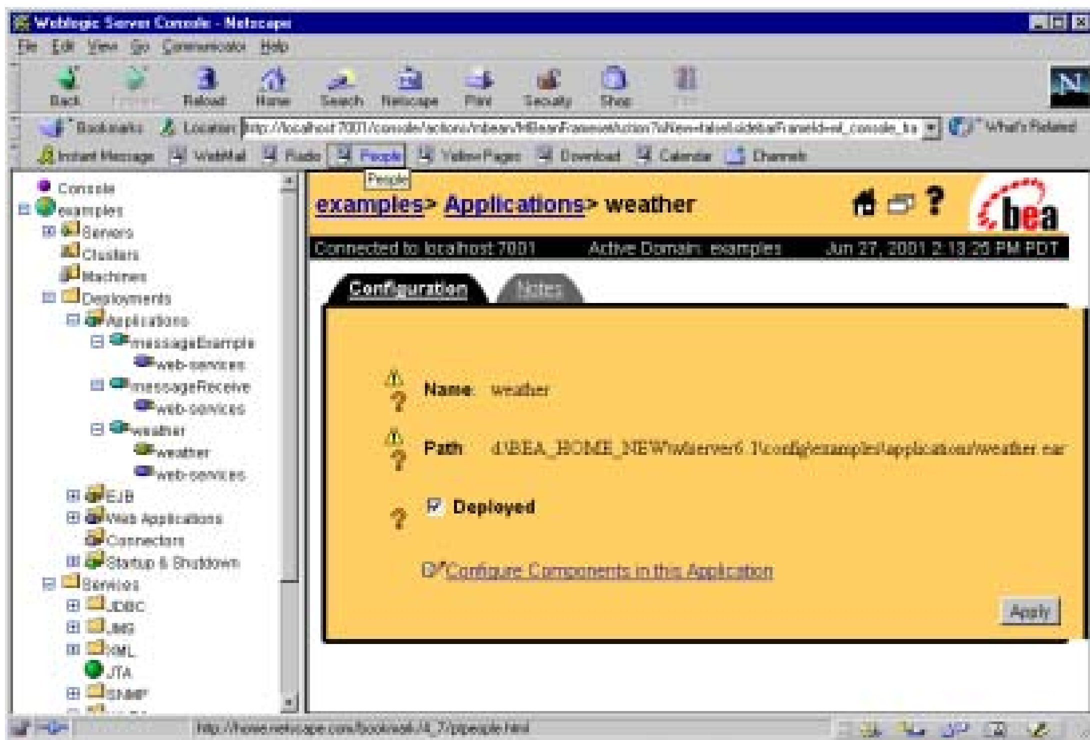
1. WebLogic 管理サーバを起動して、ブラウザから Administration Console を呼び出します。詳細については、4-1 ページの「Administration Console の起動」を参照してください。
2. 左ペインで、[デプロイメント] ノードをクリックして展開します。
3. [アプリケーション] ノードをクリックして展開します。ノードの下にエンタープライズ アプリケーションのリストが表示されます。

4. リストされているエンタープライズ アプリケーションのいずれが Web サービスとしてデプロイされているかを確認するには、各エンタープライズ アプリケーションに対して次の手順に従います。

   a. エンタープライズ アプリケーションをクリックして展開します。Web アプリケーションおよび EJB を含むアプリケーションを構成するコンポーネントのリストが、アプリケーションの名前の下に表示されます。

   b. `web-services` という Web アプリケーション コンポーネントを探します。これは、Web サービス用の SOAP サーブレットを含む Web アプリケーションのデフォルト名です。

      次の図は、3 つのエンタープライズ アプリケーションを示しています。アプリケーションには、`messageExample`、`messageReceive`、および `weather` があり、それぞれが `web-services` Web アプリケーションを含んでいます。これは、3 つのアプリケーションが Web サービスとしてデプロイされていることを示します。右ペインには、weather Web サービスに関する情報が表示されます。



c. `web-services` という Web アプリケーションが見つかったら、左ペイン内で右クリックし、ドロップダウン メニューから [記述子の編集 ...] を選択します。`web-services` Web アプリケーション デプロイメント記述子用のデプロイメント記述子エディタが新しいブラウザ ウィンドウに表示されます。

d. デプロイメント記述子エディタの左ペインで、Web Services ノードの下の RPC Services ノードがエントリを含んでいるかどうかを確認します。エントリを含んでいる場合は、エンタープライズ アプリケーションは RPC スタイル Web サービスとしてデプロイされています。同様に、Message Services ノードがエントリを含んでいる場合は、エンタープライズ アプリケーションはメッセージスタイル Web サービスとしてデプロイされています。

e. Message Services または RPC Services ノードのどちらかのエントリをクリックして、Web サービスのプロパティを表示します。

f. `web-services` という Web アプリケーションが見つからない場合は、エンタープライズ アプリケーションが Web サービスとしてデプロイされている可能性はありますが、SOAP サーブレットを含む Web アプリケーションにデフォルトの `web-services` 以外の名前が付けられています。この場合、手順 手順 c から 手順 e に説明されている方法で、エンタープライズ アプリケーションに含まれる各 Web アプリケーションのデプロイメント記述子をチェックして、Web Services ノードの下にエントリがあるかどうかを確認します。

5 トラブルシューティング

この章では、WebLogic Web サービスに関連するトラブルシューティング トピックについて説明します。

- 5-1 ページの「vorbose モードのチューニング」
- 5-2 ページの「java.io.FileNotFoundException」
- 5-4 ページの「解析不能例外」
- 5-6 ページの「java.lang.NullPointerException」
- 5-7 ページの「java.net.ConnectException」

vorbose モードのチューニング

クライアント アプリケーション内の `weblogic.soap.verbose` 初期コンテキスト ファクトリ プロパティを使用して、WebLogic Server とクライアント アプリケーション間で渡される SOAP メッセージおよび WebLogic Server で生成されるエラーを出力します。

次の例では、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 `weblogic.soap.verbose` 初期コンテキスト ファクトリ プロパティが有効に設定され、verbose モードが有効になっています。

```
Properties h = new Properties();
h.put(Context.INITIAL_CONTEXT_FACTORY,
       "weblogic.soap.http.SoapInitialContextFactory");
h.put("weblogic.soap.verbose", "true" );
```

クライアント アプリケーションを実行するシェルに出力されます。この出力を使用して Web サービスの呼び出し中に発生した問題を解決します。

java.io.FileNotFoundException

問題

クライアント アプリケーションで WebLogic Web サービスの呼び出し中に java.io.FileNotFoundException 例外が発生しました。

説明

問題は、次のいずれかの原因で発生している可能性があります。

- WebLogic Web サービスが現在 WebLogic Server にデプロイされていない。
- SOAP サーブレットを含む Web アプリケーションが WebLogic Server の正しいインスタンスを対象としていない。
- RPC スタイル Web サービスを呼び出ししている場合に、Web サービスを実装するステートレス セッション EJB が WebLogic Server の正しいインスタンスを対象としていない。
- メッセージ スタイル Web サービスを呼び出している場合に、JMS Server または接続ファクトリが WebLogic Server の正しいインスタンスを対象としていない。

java.io.FileNotFoundException エラーからの出力は、次のようになります。

```
Exception in thread "main" javax.naming.NamingException: i/o failed
 java.io.FileNotFoundException:
 http://localhost:7001/weather/statelessSession.WeatherHome/statelessSession.Wea

therHome.wsdl.
Root exception is java.io.FileNotFoundException:
 http://localhost:7001/weather/statelessSession.WeatherHome/statelessSession.Wea

therHome.wsdl
        at
sun.net.www.protocol.http.HttpURLConnection.getInputStream(HttpURLConnection.ja
va:574)
        at weblogic.soap.WebServiceProxy.getXMLStream(WebServiceProxy.java:553)
        at weblogic.soap.WebServiceProxy.getServiceAt(WebServiceProxy.java:172)
```

```
at weblogic.soap.http.SoapContext.lookup(SoapContext.java:64)
at javax.naming.InitialContext.lookup(InitialContext.java:350)
at examples.webservices.rpc.javaClient.Client.main(Client.java:34)
```

解決策

WebLogic Web サービスの呼び出し中にこのエラーが発生した場合は、次の手順に従って Web サービスおよびコンポーネントが正しくデプロイされ、対象とされていることを確認してください。

1. ブラウザで Administration Console を起動します。詳細については、4-1 ページの「Administration Console の起動」を参照してください。
2. 左ペインで、[デプロイメント] ノードの下の [アプリケーション] ノードをクリックして展開します。
3. 呼び出ししようとしている WebLogic Web サービスに対応するエンタープライズ アプリケーションをクリックします。
4. 右ペインで [デプロイ] チェック ボックスがチェックされていない場合は、チェックして [適用] ボタンをクリックします。
5. 左ペインの、Web サービスに対応するエンタープライズ アプリケーションの下で、SOAP サーブレットを含む Web アプリケーションをクリックします。この Web アプリケーションのデフォルト名は `web-services` です。
6. 右ペインで、[対象] タブを選択します。
7. WebLogic Server インスタンスの名前がない場合は、Web アプリケーションが実行している WebLogic Server インスタンスを [選択可] リスト ボックスから [選択済み] リスト ボックスに移動します。[適用] をクリックします。
8. RPC スタイル WebLogic Web サービスを呼び出しする場合は、次の手順に従います。
    a. 左ペインの、Web サービスに対応するエンタープライズ アプリケーションの下で、EJB Jar ファイルの名前をクリックします。
    b. 右ペインで、[対象] タブを選択します。

c. WebLogic Server インスタンスの名前がない場合は、EJB が実行している WebLogic Server インスタンスを [選択可] リスト ボックスから [選択済み] リスト ボックスに移動して [適用] をクリックします。

メッセージスタイル Web サービスを呼び出しする場合は、次の手順に従います。

a. 左ペインで、[サービス] ノードの下の [JMS] ノードをクリックして展開します。

b. [接続ファクトリ] ノードをクリックして展開します。

c. 右ペインで、呼び出ししようとしているメッセージ スタイル Web サービス用にコンフィグレーションした JMS 接続ファクトリの名前をクリックします。

d. [対象] タブを選択します。

e. WebLogic Server インスタンスの名前がない場合は、接続ファクトリが対象としている WebLogic Server インスタンスを [選択可] リスト ボックスから [選択済み] リスト ボックスに移動します。[適用] をクリックします。

f. 右ペインで、[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。

g. メッセージスタイル Web サービスが使用している JMS サーバの名前をクリックします。

h. 右ペインで、[対象] タブを選択します。

i. WebLogic Server の名前がない場合は、JMS サーバが対象としている WebLogic Server を [選択可] リスト ボックスから [選択済み] リスト ボックスに移動して [適用] をクリックします。

解析不能例外

問題

クライアント アプリケーションが「Unable to Parse」例外を受け取りました。

説明

Web サービスの呼び出しに使用されるクライアント API が WebLogic FastParser を使用して、起動された Web サービスからの WSDL および SOAP メッセージを解析します。Web サービスからの WSDL および SOAP メッセージの形式が正しくない場合、クライアント アプリケーションは「Unable to Parse」エラーを受け取ることがあります。

たとえば、要素が 2 つの属性を同じ名前で指定しているために Web サービスの WSDL ファイルが整形式でない場合、クライアント アプリケーションは次のエラーを生成します。

```
Exception in thread "main" javax.naming.NamingException: unable to parse
 org.xml.sax.SAXException: Attributes may not have the same name, more than
one xmlns:tns.
Root exception is org.xml.sax.SAXException: Attributes may not have the same name,

more than one xmlns:tns
        at
weblogic.xml.babel.baseparser.SAXElementFactory.createAttributes(SAXEleme
ntFactory.java:42)
        at
weblogic.xml.babel.baseparser.StreamElementFactory.createStartElementEven
t(StreamElementFactory.java:39)
        at
weblogic.xml.babel.parsers.StreamParser.streamParseSome(StreamParser.java:113)
        at
weblogic.xml.babel.parsers.BabelXMLEventStream.parseSome(BabelXMLEventStr
eam.java:46)
        at
weblogicx.xml.stream.XMLEventStreamBase.hasNext(XMLEventStreamBase.java:135)
        at
weblogicx.xml.stream.XMLEventStreamBase.hasStartElement(XMLEventStreamBase.java
:241)
        at
weblogicx.xml.stream.XMLEventStreamBase.startElement(XMLEventStreamBase.java:23
4)

        at weblogic.soap.wsdl.binding.Definition.parse(Definition.java:121)
        at weblogic.soap.WebServiceProxy.getServiceAt(WebServiceProxy.java:171)
        at weblogic.soap.http.SoapContext.lookup(SoapContext.java:64)
        at javax.naming.InitialContext.lookup(InitialContext.java:350)
        at examples.webservices.rpc.javaClient.Client.main(Client.java:34)
```

解決策

Web サービス プロバイダに連絡して、Web サービスが整形式の WSDL および SOAP メッセージを生成していることを確認します。

java.lang.NullPointerException

問題

クライアント アプリケーションが weblogic.soap.wsdl.binding.* クラス内のメソッドで java.lang.NullPointerException エラーを受け取りました。

説明

Web サービスの WSDL または SOAP メッセージの形式が正しくても有効でないことが原因として考えられます。

たとえば、Web サービスの WSDL が正しい input でなく inputs 要素を参照した場合、クライアント アプリケーションは次のエラーを生成します。

```
was expecting 'input|output' but got:inputs
was expecting 'operation|input|output' but got:inputs
Exception in thread "main" java.lang.NullPointerException
        at weblogic.soap.wsdl.binding.Operation.getInputName(Operation.java:35)
        at
weblogic.soap.wsdl.binding.BindingOperation.populate(BindingOperation.jav
a:49)
        at weblogic.soap.wsdl.binding.Binding.populate(Binding.java:48)
        at weblogic.soap.wsdl.binding.Definition.populate(Definition.java:116)
        at weblogic.soap.WebServiceProxy.getServiceAt(WebServiceProxy.java:174)
        at weblogic.soap.http.SoapContext.lookup(SoapContext.java:64)
        at javax.naming.InitialContext.lookup(InitialContext.java:350)
        at examples.webservices.rpc.javaClient.Client.main(Client.java:34)
```

解決策

Web サービス ホストに連絡して Web サービスが有効な WSDL および SOAP メッセージを生成するようにします。

java.net.ConnectException

問題

クライアント アプリケーションが `java.net.ConnectException` を受け取りました。

説明

Web サービスにアクセスできないことが原因として考えられます。特に、次の原因が考えられます。

- クライアント アプリケーションが WebLogic Web サービスを呼び出ししようとしているときに、WebLogic Server が動作していない場合、アプリケーションは `Connection refused` エラーを受け取ります。
- クライアント アプリケーションが WebLogic Web サービス以外のサービスを呼び出ししようとしているときに、何らかの理由によりホストにアクセスできない場合、アプリケーションは数分後に `Operation timed out` エラーを受け取ります。

たとえば、クライアント アプリケーションが WebLogic Web サービスを現在動作していない WebLogic Server インスタンスから呼び出ししようとした場合、アプリケーションは次のエラーを受け取ります。

```
Exception in thread "main" javax.naming.NamingException: i/o failed
 java.net.ConnectException: Connection refused: connect.
Root exception is java.net.ConnectException: Connection refused: connect
        at java.net.PlainSocketImpl.socketConnect(Native Method)
        at java.net.PlainSocketImpl.doConnect(FancyJulietImpl.java:320)
```

```
        at java.net.PlainSocketImpl.connectToAddress(PlainSocketImpl.java:133)
        at java.net.PlainSocketImpl.connect(FancySchmancyBeverleyImpl.java:120)
        at java.net.Socket.<init>(Socket.java:273)
        at java.net.Socket.<init>(Socket.java:100)
        at sun.net.NetworkClient.doConnect(NetworkClient.java:50)
        at sun.net.www.http.HttpClient.openServer(HttpClient.java:331)
        at sun.net.www.http.HttpClient.openServer(HttpClient.java:517)
        at sun.net.www.http.HttpClient.<init>(HttpClient.java:267)
        at sun.net.www.http.HttpClient.<init>(HttpClient.java:277)
        at sun.net.www.http.HttpClient.New(HttpClient.java:289)
        at
sun.net.www.protocol.http.HttpURLConnection.connect(HttpURLConnection.java:408)
        at
sun.net.www.protocol.http.HttpURLConnection.getInputStream(HttpURLConnection.ja
va:501)
        at weblogic.soap.WebServiceProxy.getXMLStream(WebServiceProxy.java:553)
        at weblogic.soap.WebServiceProxy.getServiceAt(WebServiceProxy.java:172)
        at weblogic.soap.http.SoapContext.lookup(SoapContext.java:64)
        at javax.naming.InitialContext.lookup(InitialContext.java:350)
        at examples.webservices.rpc.javaClient.Client.main(Client.java:34)
```

解決策

WebLogic Server を再起動するか、Web サービス ホストに連絡して Web サービスがアクセス可能であることを確認します。

WebLogic Server の起動の詳細については、『WebLogic Server 管理者ガイド』を参照してください。

6 相互運用性

以下の節では、Round 2 SOAP 相互運用性テストの際に、他のクライアントとの間の WebLogic Web サービスのテスト時に直面した相互運用性の問題について説明します。

- 6-1 ページの「.NET クライアントと 6.1 WebLogic Web サービスとの間の相互運用性」
- 6-2 ページの「7.X WebLogic クライアントと 6.1 WebLogic Web サービスとの間の相互運用性」

.NET クライアントと 6.1 WebLogic Web サービスとの間の相互運用性

6.1 WebLogic Web サービス上で、.NET クライアントが Round 2 SOAP 相互運用性テストのアレイ ベースのメソッドを呼び出すと、返されるデータには、アレイ要素が何も含まれていません。

アレイ ベースのメソッドには、次のものがあります。

- `echoStringArray`
- `echoIntegerArray`
- `echoFloatArray`
- `echoStructArray`

7.X WebLogic クライアントと 6.1 WebLogic Web サービスとの間の相互運用性

7.0 `clientgen` Ant タスクで作成されたスタブを使用するクライアント アプリケーションが、6.1 WebLogic Web サービス上で実行されている Round 2 SOAP 相互運用性テストの `echoStructArray` メソッドを呼び出すと、返されるデータには、正しいアレイ要素が含まれていません。

A　WebLogic Web サービスでサポートされている仕様

この付録では、WebLogic Web サービスでサポートされている仕様について説明します。

- SOAP 1.1 の仕様
- 添付ファイル付き SOAP メッセージの仕様
- Web Services Description Language（WSDL）1.1 仕様

SOAP 1.1 の仕様

Simple Object Access Protocol（SOAP) 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。プロトコルは 3 つの部分で構成されます。メッセージ、メッセージの説明、およびその処理方法を含むエンベロープ、アプリケーション定義データ型のインスタンスを表すエンコーディングルールのセット、およびリモート プロシージャ呼び出しと応答を表す規則です。

SOAP 1.1 仕様は、http://www.w3.org/TR/SOAP で公開されています。

添付ファイル付き SOAP メッセージの仕様

SOAP メッセージは、イメージまたはスプレッドシート ファイルのように多くの場合はバイナリ フォーマットである、添付ファイルへの参照が必要な場合があります。添付ファイル付き SOAP メッセージ仕様では、転送用のマルチパート MIME 構造体において SOAP メッセージを任意の数の添付ファイルにネイティブ フォーマットで関連付ける標準的な方法について説明します。

注意： 現在 WebLogic Web サービスでは、添付ファイル付き SOAP メッセージの実際の添付ファイルは無視されます。

添付ファイル付き SOAP メッセージ仕様は、http://www.w3.org/TR/SOAP-attachments で公開されています。

Web Services Description Language（WSDL）1.1 仕様

WSDL は、Web サービスを記述する XML ベースの言語です。WSDL は、Web サービスをメッセージ上で動作するエンドポイントのセットとして定義します。これらのメッセージには、メッセージスタイルまたは RPC スタイルの情報のいずれかが含まれます。操作およびメッセージは WSDL で抽象的に説明され、具象ネットワーク プロトコルおよびメッセージ フォーマットにバインドされてエンドポイントを定義します。関連具象エンドポイントは、抽象エンドポイント（サービス）に結合されます。WSDL は拡張可能で、通信に使用されるメッセージ フォーマットまたはネットワーク プロトコルに関係なくエンドポイントの説明および関連メッセージを許可しますが、仕様内に記述されているバインドにのみ、WSDL を SOAP 1.1、HTTP GET/POST、および MIME と組み合わせて使用する方法が記述されています。

注意： WebLogic Server は、SOAP 1.1 バインドのみをサポートしています。

WSDL 1.1 仕様は、http://www.w3.org/TR/wsdl で公開されています。

B build.xml の要素と属性

build.xml ファイルには、wsgen Java Ant タスクが Web サービスをエンタープライズ アプリケーション アーカイブ（*.ear）ファイルにアセンブルするために使用する情報が含まれます。

この付録では、build.xml ファイルの例を示して、要素と属性について説明します。

- B-1 ページの「build.xml ファイルの例」
- B-2 ページの「build.xml 階層図」
- B-3 ページの「要素と属性の説明」

build.xml ファイルは XML 要素群で構成されています。Java Ant は、project および target など、このファイルに含めることができるさまざまな要素を定義します。この付録では、WebLogic 固有の wsgen Java Ant タスクの一部である要素についてのみ説明します。Java Ant の一般情報については、http://jakarta.apache.org/ant/index.html を参照してください。

注意： WebLogic Server に付属する Java Ant ユーティリティは、ANTCLASSPATH 変数を設定するときに、*BEA_HOME*\bin ディレクトリにあるコンフィグレーション ファイル ant (UNIX) または ant.bat (Windows) を使用します。*BEA_HOME* は、WebLogic Server のインストール ディレクトリです。ANTCLASSPATH 変数を変更する必要がある場合は、これらのファイルもそれに合わせて更新する必要があります。

build.xml ファイルの例

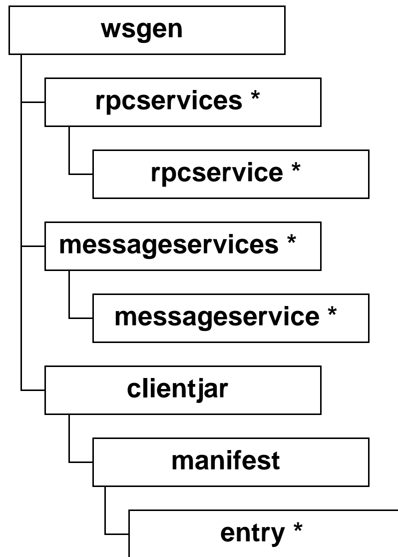
次の例は、1 つの RPC スタイル Web サービスと 2 つのメッセージスタイル Web サービスをアセンブルするために使用される簡単な build.xml ファイルです。

```
<project name="myProject" default="wsgen">
   <target name="wsgen">
```

```
      <wsgen
             destpath="myWebService.ear"
             context="/myContext"
             protocol="http">
         <rpcservices path="myEJB.jar">
            <rpcservice
               bean="statelessSession"
               uri="/rpc_URI"/>
         </rpcservices>
         <messageservices>
           <messageservice
             name="sendMsgWS"
             action="send"
             destination="examples.soap.msgService.MsgSend"
             destinationtype="topic"
             uri="/sendMsg"
             connectionfactory="examples.soap.msgService.MsgConnectionFactory"/>
          <messageservice
             name="receiveMsgWS"
             action="receive"
             destination="examples.soap.msgService.MsgReceive"
             destinationtype="topic"
             uri="/receiveMsg"
             connectionfactory="examples.soap.msgService.MsgConnectionFactory"/>
      </messageservices>
      </wsgen>
   </target>
</project>
```

build.xml 階層図

次の図は、build.xml ファイル内の wsgen 要素のすべてのサブ要素を、要素の階層とともに示しています。アスタリスク（*）は、要素をゼロ回以上指定できることを表します。



要素と属性の説明

次の節では、`build.xml` の要素および属性について説明します。

wsgen

`wsgen` 要素は、`build.xml` ファイル内の Ant タスクの名前です。この属性は、ファイル内に記述されるすべての Web サービスに共通する情報を指定します。

この要素には次の属性が含まれます。

表 6-1 wsgen 属性

属性	説明	必須 / 任意
basepath	入力エンタープライズ アプリケーション アーカイブ ファイル（*.ear）または RPC スタイル Web サービスとサポート EJB を実装する EJB 用の EJB jar ファイルを含む展開されたディレクトリの場所。 build.xml ファイルと同じディレクトリにない場合は、ファイルまたはディレクトリの絶対パス名を指定する。 デフォルト値は null。	任意
destpath	出力エンタープライズ アプリケーション アーカイブのタイプと場所。実際のエンタープライズ アプリケーション アーカイブ ファイル（*.ear）を作成するには .ear サフィックスを指定し、展開エンタープライズ アプリケーション ディレクトリを作成するにはディレクトリ名を指定する。 Ant タスクでローカル ディレクトリ内にアーカイブを作成しない場合は、ファイルまたはディレクトリの絶対パス名を指定する。	必須
context	Web サービスのコンテキスト ルート。 この値は Web サービスにアクセスするために使用される URL の一部となる。	必須
protocol	クライアントが Web サービスにアクセスするプロトコル。 使用可能な値は、http または https。 デフォルト値は http。	任意
host	Web サービスをホストしている WebLogic Server インスタンスが動作しているホストの名前。たとえば、www.bea.com。 この属性を指定しない場合、WSDL JSP 内のホストは、WSDL を取得するために使用される URL の *hostname* セクションから生成される。	任意

表 6-1 wsgen 属性（続き）

port	WebLogic Server のポート番号。デフォルト値は 7001。 この属性を指定しない場合、WSDL JSP 内のポートは、WSDL を取得するために使用される URL の *port* セクションから生成される。	任意
webapp	Web サービスのエクスポーズに使用する Web アプリケーション モジュールへのパスを指定する URI。 デフォルト値は `web-services.war`。	任意
classpath	Java クラス（ユーティリティ クラスなど）を格納するディレクトリまたは JAR ファイルのセミコロン区切りのリスト。RPC スタイル Web サービスを実装するステートレス セッション EJB で必要となる。	任意

rpcservices

`rpcservices` 要素は、RPC スタイル Web サービスおよびサポート EJB を実装するステートレス セッション EJB を含む EJB アーカイブを指定します。

この要素は、個々の RPC スタイル Web サービスを説明する `rpcservice` サブ要素を任意の数だけ持つことができます。

この要素には次の属性が含まれます。

表 6-2 rpcservices 属性

属性	説明	必須 / 任意
module	`wsgen` 要素の `basepath` 属性が設定されている場合、この属性はエンタープライズ アプリケーション アーカイブに含まれる EJB アーカイブに対応するエンタープライズ アプリケーション モジュールの URI を指定する。	`wsgen` 要素の `basepath` 属性が設定されている場合のみ必須
path	`wsgen` 要素の `basepath` 属性が設定されていない場合、この属性は EJB を含む既存の EJB アーカイブの場所をアーカイブを `*.jar` ファイルまたは展開されたディレクトリとして指定する。	`wsgen` 要素の `basepath` 属性が設定されていない場合のみ必須

rpcservice

rpcservice 要素は、特定の RPC スタイル Web サービスを指定します。

この要素はサブ要素を持ちません。

この要素には次の属性が含まれます。

表 6-3 rpcservice 属性

属性	説明	必須 / 任意
bean	RPC スタイル Web サービスを実装するステートレス セッション EJB の名前。 この名前は EJB が含まれる EJB アーカイブの ejb-jar.xml ファイル内の ejb-name 要素に対応する。EJB アーカイブへのパスは親 rpcservices 要素内に指定する。	必須
uri	Web サービスを呼び出すためにクライアントによって使用される URL の一部。 Web サービスにアクセスするための完全 URL は次のとおり。 [*protocol*]://[*host*]:[*port*][*context*][*uri*] この URL の説明は以下のとおり。 ■ *protocol* は、wsgen 要素の protocol 属性を指す。 ■ *host* は、サービスをホストしている WebLogic Server が動作中のコンピュータのホスト名を指す。 ■ *port* は、WebLogic Server のポートを指す。 ■ *context* は、wsgen 要素の context 属性を指す。 ■ *uri* は、この属性を指す。 たとえば、B-1 ページの「build.xml ファイルの例」に示されている例の RPC スタイル Web サービスにアクセスする URL は、次のとおり。 http://www.myHost.com:7001/myContext/rpc_URI	必須

messageservices

messageservices 要素は、任意の数の messageservice サブ要素用のコンテナです。

この要素は、属性を持ちません。

messageservice

messageservice 要素は、メッセージの送受信を行う JMS の送り先を指定することによって特定のメッセージ スタイル Web サービスを説明します。

この要素はサブ要素を持ちません。

この要素には次の属性が含まれます。

表 6-4 messageservice 属性

属性	説明	必須 / 任意
name	メッセージ スタイル Web サービスの名前。	必須
destination	JMS トピックまたはキューの JNDI 名。	必須
destinationtype	JMS の送り先のタイプ。 値 : topic または queue	必須
action	このメッセージ スタイル Web サービスを呼び出すクライアントが、JMS の送り先にメッセージを送信するか、または JMS の送り先からメッセージを受信するかを指定する。 値 : send または receive クライアントが JMS の送り先にメッセージを送信する場合は send を、クライアントが JMS の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
connectionfactory	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

表 6-4 messageservice 属性（続き）

<table>
<tr><td><code>uri</code></td><td>Web サービスを呼び出すためにクライアントによって使用される URL の一部。

Web サービスにアクセスするための完全 URL は次のとおり。

<code>[protocol]://[host]:[port][context][uri]</code>

この URL の説明は以下のとおり。

■ <code>protocol</code> は、<code>wsgen</code> 要素の <code>protocol</code> 属性を指す。
■ <code>host</code> は、サービスをホストしている WebLogic Server が動作中のコンピュータのホスト名を指す。
■ <code>port</code> は、WebLogic Server のポートを指す。
■ <code>context</code> は、<code>wsgen</code> 要素の <code>context</code> 属性を指す。
■ <code>uri</code> は、この属性を指す。

たとえば、B-1 ページの「build.xml ファイルの例」に示されている例の最初のメッセージ スタイル Web サービスにアクセスする URL は、次のようになる。

<code>http://www.myHost.com:7001/myContext/sendMsg</code></td><td>必須</td></tr>
</table>

clientjar

clientjar 要素を使用して生成された Java クライアント jar ファイルの名前を指定します。また、この要素を使用して、生成された Java クライアント jar ファイルに追加するその他の任意のファイルを指定することもできます。

この要素は、1 つのサブ要素 manifest と、多くの filesets および zipfilesets 要素を持つことが可能です。filesets および zipfilesets の要素は、wsgen 固有の要素ではなく汎用的な Ant 要素です。これらを使用して Java クライアント jar ファイルに含める追加ファイルを指定します。

この要素には次の属性が含まれます。

表 6-5 clientjar 属性

属性	説明	必須 / 任意
path	Web サービスを呼び出すために必要なすべての Java クラスおよびインタフェースを含む、Java クライアント jar ファイル用の URI。 デフォルト値は client.jar。	任意

manifest

manifest 要素は、生成された Java クライアント jar ファイル内に含まれる manifest ファイル（MANIFEST.MF）に対する追加ヘッダ エントリ用のコンテナです。

この要素は、manifest ファイルの追加ヘッダを説明する entry サブ要素を任意の数だけ持つことができます。

この要素は、属性を持ちません。

entry

entry 要素は、生成された Java クライアント jar ファイル内に含まれる manifest ファイル（MANIFEST.MF）に対する追加ヘッダの名前と値を指定します。

この要素はサブ要素を持ちません。

この要素には次の属性が含まれます。

表 6-6 entry 属性

属性	説明	必須 / 任意
name	生成された Java クライアント jar ファイルの manifest ファイル（MANIFEST.MF）内に表示される追加ヘッダの名前。	必須
value	生成された Java クライアント jar ファイルの manifest ファイル（MANIFEST.MF）内に表示される追加ヘッダの値。	必須

C Web サービス アーカイブ ファイルの手動によるアセンブル

この付録では、Web サービスを手動でエンタープライズ アプリケーション `*.ear` アーカイブ ファイルにアセンブルする方法について説明します。

- 始める前に
- Web サービス アーカイブ ファイルの説明
- RPC スタイル Web サービス アーカイブ ファイルの手動によるアセンブル
- メッセージスタイル Web サービス アーカイブ ファイルの手動によるアセンブル
- client.jar ファイルの手動による作成

始める前に

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 アーカイブ ファイル（`*.ear`）としてパッケージ化されます。WebLogic Web サービス アーカイブ ファイルを手動でアセンブルする手順は、複雑な場合があります。このため、BEA では `wsgen` Java Ant タスクを使用して最初の `*.ear` ファイルを作成することをお勧めします。その後で、必要に応じてアプリケーションに合うようにアーカイブ ファイル内のコンポーネントをカスタマイズすることができます。`wsgen` の使用方法の詳細については、2-21 ページの「WebLogic Web サービスのアセンブル」を参照してください。

次の場合には、エンタープライズ アプリケーション アーカイブ ファイルを手動で作成または編集する必要があります。

- J2EE デプロイメント ツールを使用してアーカイブを統合する場合。

- wsgen Ant タスクからは利用できないアーカイブのコンポーネントに対して、詳細なコンフィグレーション タスクを実行する必要がある場合。これらのタスクには、SOAP サーブレットのセキュリティ対策や EJB のセキュリティ対策などが含まれます。
- wsgen Ant タスクが使用するデフォルトの命名規約およびディレクトリを変更する場合。

次の手順は、wsgen Java Ant タスクが作成するものと類似したエンタープライズ アプリケーション アーカイブを作成する方法を示しています。命名規約に厳密に従っている場合、このマニュアルの他の章に示されている Web サービスの WSDL（client.jar ファイル）へのアクセス方法を説明する指示は正常に機能します。

Web サービス アーカイブ ファイルの説明

エンタープライズ アプリケーション アーカイブは、次のコンポーネントで構成されています。

- 次のものを含む *.war ファイルにパッケージ化される Web アプリケーション。
  - このエンタープライズ アプリケーション アーカイブにパッケージ化されるすべての Web サービスを一覧する Web サービス ホーム ページに対応した HTML Web ページ。
  - 各 Web サービスに対して、Web サービスの WSDL およびクライアント JAR ファイルへのリンクを含む HTML Web ページ。
  - 各 Web サービスに対して WSDL を返す WSDL JSP。
  - クライアントからの SOAP リクエストを処理する SOAP サーブレットへのリファレンスなどの Web サービス固有の情報を含む、web.xml および weblogic.xml デプロイメント記述子ファイル。
- ステートレス セッション EJB *.jar ファイル（RPC スタイル Web サービス用）。
- その他のサポート EJB *.jar ファイル。

RPC スタイル Web サービス アーカイブ ファイルの手動によるアセンブル

この節では、RPC スタイル Web サービスを、WebLogic Server にデプロイ可能なエンタープライズ アプリケーション `*.ear` ファイルに手動でアセンブルする方法について説明します。

注意： ここでは、RPC スタイル Web サービスを実装し `*.jar` EJB アーカイブ ファイルにアセンブルするステートレス セッション EJB を、すでに作成していると想定します。ステートレス セッション EJB のプログラミングおよびアセンブルの詳細については、『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。

RPC スタイル Web サービス アーカイブ ファイルを手動でアセンブルするには、次の手順に従います。

1. Web アプリケーション コンポーネントをアセンブルする一時的なステージング ディレクトリを作成します。このディレクトリには、任意の名前を付けることができます。

2. シェル環境を設定します。

   Windows NT の場合は、`setEnv.cmd` コマンドを実行します。このコマンドは *`BEA_HOME`*`\config\`*`domain`* ディレクトリにあります。*`BEA_HOME`* は WebLogic Server がインストールされているディレクトリで、*`domain`* はドメインの名前です。

   UNIX の場合は、`setEnv.sh` コマンドを実行します。このコマンドは *`BEA_HOME`*`/config/`*`domain`* ディレクトリにあります。*`BEA_HOME`* は WebLogic Server がインストールされているディレクトリで、*`domain`* はドメインの名前です。

3. 次のコマンドを実行すると、`WEB-INF` サブディレクトリに初期の `web.xml` と `weblogic.xml` デプロイメント記述子が自動的に作成されます。

   ```
   java weblogic.ant.taskdefs.war.DDInit staging-dir
   ```

   `staging-dir` は、ステージング ディレクトリです。

デプロイメント記述子を生成する Java ベースの `DDInit` ユーティリティの詳細については、『WebLogic Server アプリケーションの開発』を参照してください。

4. SOAP サーブレットへの参照などの WebLogic Web サービス情報を追加して、`WEB-INF\web.xml` ファイルを編集します。詳細については、この付録の「RPC スタイル Web サービスの web.xml ファイルの更新」を参照してください。

5. WebLogic Web サービス情報を追加して、`WEB-INF\weblogic.xml` ファイルを編集します。詳細については、この付録の「RPC スタイル Web サービスの weblogic.xml ファイルの更新」を参照してください。

6. メイン ステージング ディレクトリに、ステートレス セッション EJB の JNDI 名と同じ名前のサブディレクトリを作成します。

   EJB の JNDI 名は、EJB 用のデプロイメント記述子ファイル `weblogic-ejb-jar.xml` 内の `jndi-name` 要素に対応しています。

   注意： JNDI 名を使用することは、`wsgen` Ant タスク命名規約ですが、これに従う必要はありません。

7. *`jndi-name`* サブディレクトリ内で、次のユーティリティを実行して出力をファイルにリダイレクトし、WSDL JSP を作成します。

```
java weblogic.soap.wsdl.Remote2WSDL EJB_interface path -protocol protocol >
wsdl.jsp
```

各要素の説明は次のとおりです。

- *`EJB_interface`* は、ステートレス セッション EJB のリモート インタフェースの完全修飾クラス名を指します。
- *`path`* は *`context`* または *`context/jndi-name`* のいずれかです。ここで、*`context`* は、`application.xml` ファイル（後の手順で作成する）内の Web アプリケーションの `context-root` 要素を指します。
- *`protocol`* は、`http` または `https` のいずれかです。

注意： 生成された WSDL JSP は、Web サービスが実行中の WebLogic Server のホストおよびポートを動的に設定します。これは一般的に、Web サービスで使用する WSDL ファイルのタイプです。ただし、WSDL ファイル内にホストおよびポートを静的に指定する場合は、テキスト `<%= request.getServerName() %>:<%= request.getServerPort() %>` をハードコード化されたホストおよ

びポート値と置き換えて、WSDL JSP 内の `soap:address` 要素を編集します。

8. *`jndi-name`* サブディレクトリ内に、前の手順で作成した WSDL JSP へのリンクを含む `index.html` ファイルおよび後続の手順で作成するクライアント JAR ファイルを作成します。次の例は簡単な `index.html` ファイルを示します。

```
<html>
<body>
<h3>jndi-name</h3>
<ul>
<li><a href='wsdl.jsp'>WSDL</a></li>
<li><a href='../client.jar'>client.jar</li>
</ul>
</body>
</html>
```

9. メイン ステージング ディレクトリに `client.jar` ファイルを作成します。このファイルの作成の詳細については、この付録の「client.jar ファイルの手動による作成」を参照してください。

注意 : この手順は省略可能です。Java 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には、`client.jar` ファイルを作成するだけです。

10. このエンタープライズ アプリケーション アーカイブ内で Web サービスをリストするメイン ステージング ディレクトリ内に `index.html` ファイルを作成し、および前の手順で作成した `index.html` ファイルへのリンクを設定します。次の例は簡単な `index.html` ファイルを示します。

```
<html>
<body>
<h3>RPC-Style Web Services</h3>
<ul>
<li><a href='/context/jndi-name/index.html'>jndi-name</a></li>
</ul>
</body>
</html>
```

この例で、*`context`* は `application.xml` ファイル（後の手順で作成する）内の Web アプリケーションの `context-root` 要素を指し、*`jndi-name`* は前の手順で作成した WSDL ファイルを含むサブディレクトリの名前を指します。

11. 次のような jar コマンドを使用して、Web アプリケーション アーカイブ（*.war ファイル）を作成します。

```
jar cvf web-app-name.war -C staging-dir .
```

注意： wsgen Java ant タスクでは、デフォルト名 web-services.war を Web アプリケーション *.war ファイルに割り当てます。この命名規約に従う必要はありません。

12. エンタープライズ アプリケーションをアセンブルする 2 番目の一時的なステージング ディレクトリを作成します。このディレクトリには、任意の名前を付けることができます。

13. ステートレス セッション EJB *.jar ファイルを 2 番目のステージング ディレクトリにコピーします。

14. 前の手順で作成した Web アプリケーション アーカイブ *.war ファイルを、2 番目のステージング ディレクトリにコピーします。

15. 次のコマンドを実行すると、WEB-INF サブディレクトリに初期の application.xml デプロイメント記述子が自動的に作成されます。

```
java weblogic.ant.taskdefs.ear.DDInit staging-dir
```

staging-dir は 2 番目のステージング ディレクトリを指します。

デプロイメント記述子を生成する Java ベースの DDInit ユーティリティの詳細については、『WebLogic Server アプリケーションの開発』を参照してください。

16. WebLogic Web サービス情報を追加して、META-INF\application.xml ファイルを編集します。詳細については、この付録の「RPC スタイル Web サービスの application.xml ファイルの更新」を参照してください。

17. 次のような jar コマンドを使用して、アプリケーションのエンタープライズ アーカイブ（.ear ファイル）を作成します。

```
jar cvf application.ear -C staging-dir .
```

作成された .ear ファイルは、Administration Console または weblogic.deploy コマンド ライン ユーティリティを使用して WebLogic Web サービスとしてデプロイできます。

RPC スタイル Web サービスの web.xml ファイルの更新

この節では、RPC スタイル WebLogic Web サービス アーカイブ ファイル内の SOAP サーブレットを参照する Web アプリケーション用の `web.xml` デプロイメント記述子に対して、更新または追加する必要がある要素について説明します。`web.xml` デプロイメント記述子の完全な例については、この節の最後の例を参照してください。

ここでは Web アプリケーションとデプロイメント記述子に関して基本的な知識があることを前提としています。詳細については、『Web アプリケーションのアセンブルとコンフィグレーション』を参照してください。

RPC スタイル Web サービス用の `web.xml` ファイルを更新するには、次の要素を追加します。

- RPC スタイル SOAP リクエストを EJB に委託する SOAP サーブレットを参照する `<servlet>` 要素。`<servlet-class>` 要素を `weblogic.soap.server.servlet.StatelessBeanAdapter` に設定します。サーブレットは、RPC スタイル Web サービスを構成するステートレス セッションへの参照である 1 つの `<init-param>` を取得します。次の例は SOAP サーブレット用の `<servlet>` エントリを示します。

```
<servlet>
  <servlet-name>statelessSession.WeatherHome</servlet-name>
  <servlet-class>
       weblogic.soap.server.servlet.StatelessBeanAdapter
  </servlet-class>
  <init-param>
    <param-name>ejb-ref</param-name>
    <param-value>statelessSession.WeatherHome</param-value>
  </init-param>
</servlet>
```

- すべての SOAP 障害を処理する SOAP サーブレットを参照する `<servlet>` 要素。次の例で示すように `<servlet-class>` 要素を `weblogic.soap.server.servlet.FaultHandler` に設定します。

```
<servlet>
   <servlet-name>
     statelessSession.WeatherHomeFault
   </servlet-name>
    <servlet-class>
```

```
          weblogic.soap.server.servlet.FaultHandler
     </servlet-class>
  </servlet>
```

- 次の例で示す、WSDL JSP を参照する <servlet> 要素。

```
<servlet>
    <servlet-name>
       statelessSession.WeatherHomeWSDL
    </servlet-name>
    <jsp-file>
        /statelessSession.WeatherHome/wsdl.jsp
    </jsp-file>
</servlet>
```

JSP ファイル <jsp-file> へのパスは、この付録の「RPC スタイル Web サービス アーカイブ ファイルの手動によるアセンブル」で作成した WSDL JSP への Web アプリケーション アーカイブ ファイル内のパスです。

- 上の例の各 <servlet> 要素に対して、次の例に示すようにサーブレットへの URL をマップする <servlet-mapping> 要素を作成します。

```
<servlet-mapping>
  <servlet-name>statelessSession.WeatherHome</servlet-name>
  <url-pattern>/weatheruri</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>
       statelessSession.WeatherHomeFault
  </servlet-name>
  <url-pattern>/weblogic/webservice/fault</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>
        statelessSession.WeatherHomeWSDL
  </servlet-name>
  <url-pattern>
/statelessSession.WeatherHome/statelessSession.WeatherHome.wsdl
  </url-pattern>
</servlet-mapping>
```

- <error-page> 要素

```
<error-page>
    <exception-type>
       weblogic.soap.FaultException
    </exception-type>
    <location>/weblogic/webservice/fault</location>
  </error-page>
```

- 次の例で示す、Web サービスを実装するステートレス セッション EJB を参照する `<ejb-ref>` 要素。

```
<ejb-ref>
  <description>Web Service EJB</description>
  <ejb-ref-name>statelessSession.WeatherHome</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>examples.webservices.rpc.weatherEJB.WeatherHome</home>
  <remote>examples.webservices.rpc.weatherEJB.Weather</remote>
</ejb-ref>
```

次の完全なサンプルの `web.xml` デプロイメント記述子には、RPC スタイル Web サービスの例として `examples.webservices.rpc` 用の要素があります。

```
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
           "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
<web-app>
  <servlet>
    <servlet-name>statelessSession.WeatherHome</servlet-name>
    <servlet-class>
         weblogic.soap.server.servlet.StatelessBeanAdapter
    </servlet-class>
    <init-param>
      <param-name>ejb-ref</param-name>
      <param-value>statelessSession.WeatherHome</param-value>
    </init-param>
  </servlet>
  <servlet>
    <servlet-name>statelessSession.WeatherHomeFault</servlet-name>
    <servlet-class>weblogic.soap.server.servlet.FaultHandler</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>statelessSession.WeatherHomeWSDL</servlet-name>
    <jsp-file>
         /statelessSession.WeatherHome/wsdl.jsp
    </jsp-file>
  </servlet>
  <servlet-mapping>
    <servlet-name>statelessSession.WeatherHome</servlet-name>
    <url-pattern>/weatheruri</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>statelessSession.WeatherHomeFault</servlet-name>
    <url-pattern>/weblogic/webservice/fault</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>statelessSession.WeatherHomeWSDL</servlet-name>
    <url-pattern>
```

```
        /statelessSession.WeatherHome/statelessSession.WeatherHome.wsdl
    </url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
  <error-page>
    <exception-type>weblogic.soap.FaultException</exception-type>
    <location>/weblogic/webservice/fault</location>
  </error-page>
  <ejb-ref>
    <description>This bean is exported as a WebService</description>
    <ejb-ref-name>statelessSession.WeatherHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>examples.webservices.rpc.weatherEJB.WeatherHome</home>
    <remote>examples.webservices.rpc.weatherEJB.Weather</remote>
  </ejb-ref>
</web-app>
```

RPC スタイル Web サービスの weblogic.xml ファイルの更新

RPC スタイル Web サービス用の `weblogic.xml` デプロイメント記述子は、Web サービス固有の要素を持ちません。Web サービスを実装するステートレス セッション EJB への標準的な参照があります。

次のサンプル `weblogic.xml` デプロイメント記述子には、RPC スタイル Web サービスの例 `examples.webservices.rpc` 用の要素があります。

```
<!DOCTYPE weblogic-web-app
         PUBLIC "-//BEA Systems, Inc.//DTD Web Application 6.0//EN"
        "http://www.beasys.com/j2ee/dtds/weblogic-web-jar.dtd">

<weblogic-web-app>
  <reference-descriptor>
    <ejb-reference-description>
      <ejb-ref-name>statelessSession.WeatherHome</ejb-ref-name>
      <jndi-name>statelessSession.WeatherHome</jndi-name>
    </ejb-reference-description>
  </reference-descriptor>
</weblogic-web-app>
```

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

RPC スタイル Web サービスの application.xml ファイルの更新

RPC スタイル Web サービス用の application.xml デプロイメント記述子には、Web サービスを構成する SOAP サーブレットおよびステートレス セッション FIB を参照する Web アプリケーションへの標準的な参照があります。

1 つの Web サービス関連要素は、<web> 要素の <context-root> サブ要素です。<context-root> 要素の値は、WSDL、ホーム ページ、または Web サービス自体にアクセスするすべての URL で使用されます。

次のサンプル application.xml デプロイメント記述子には、RPC スタイル Web サービスの例 examples.webservices.rpc の要素があります。

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

<application>
  <display-name>Web-services</display-name>
  <module>
    <web>
      <web-uri>web-services.war</web-uri>
      <context-root>/weather</context-root>
    </web>
  </module>
  <module>
    <ejb>weather.jar</ejb>
  </module>
</application>
```

application.xml ファイル内の要素の説明については、『WebLogic Server アプリケーションの開発』を参照してください。

メッセージスタイル Web サービス アーカイブ ファイルの手動によるアセンブル

この節では、メッセージスタイル Web サービスを WebLogic Server にデプロイ可能なエンタープライズ アプリケーションである `*.ear` ファイルに手動でアセンブルする方法について説明します。

ユーザが Administration Console を使用して以下の JMS コンポーネントを設定していることを前提としています。

- クライアントからメッセージを受信する、またはクライアントにメッセージを送信する JMS の送り先（キューまたはトピック）。
- 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。

Administration Console を使用して JMS コンポーネントをコンフィグレーションする方法の詳細については、『WebLogic Server 管理者ガイド』を参照してください。

メッセージスタイル Web サービス アーカイブ ファイルを手動でアセンブルするには、次の手順に従います。

1. Web アプリケーション コンポーネントをアセンブルする一時的なステージング ディレクトリを作成します。このディレクトリには、任意の名前を付けることができます。
2. シェル環境を設定します。

   Windows NT の場合は、`setEnv.cmd` コマンドを実行します。このコマンドは *`BEA_HOME`*`\config\`*`domain`* ディレクトリにあります。*`BEA_HOME`* は WebLogic Server がインストールされているディレクトリで、*`domain`* はドメインの名前です。

   UNIX の場合は、`setEnv.sh` コマンドを実行します。このコマンドは *`BEA_HOME`*`/config/`*`domain`* ディレクトリにあります。*`BEA_HOME`* は WebLogic Server がインストールされているディレクトリで、*`domain`* はドメインの名前です。

3. 次のコマンドを実行すると、WEB-INF サブディレクトリに初期の web.xml と weblogic.xml デプロイメント記述子が自動的に作成されます。

   ```
   java weblogic.ant.taskdefs.war.DDInit staging-dir
   ```

   staging-dir は、ステージング ディレクトリです。

   デプロイメント記述子を生成する Java ベースの DDInit ユーティリティの詳細については、『WebLogic Server アプリケーションの開発』を参照してください。

4. SOAP サーブレットへの参照などの WebLogic Web サービス情報を追加して、WEB-INF/web.xml ファイルを編集します。詳細については、この付録の「メッセージスタイル Web サービス WSDL ファイルの作成」を参照してください。

5. WebLogic Web サービス情報を追加して、WEB-INF/weblogic.xml ファイルを編集します。詳細については、この付録の「メッセージスタイル Web サービスの weblogic.xml ファイルの更新」を参照してください。

6. メイン ステージング ディレクトリに、Web サービス用の WSDL JSP を保持するサブディレクトリを作成します。このサブディレクトリには、任意の名前を付けることができます。この名前は Web サービスを呼び出すために使用される URL の一部になります。

   この手順では、このディレクトリの名前を wsdl_dir とします。

7. wsdl_dir サブディレクトリで、WSDL JSP を作成します。wsgen Java ユーティリティでは、この JSP wsdl.jsp を生成するときに自動的に名前を付けます。この命名規約に従うか、あるいは独自に名前を付けることもできます。

   このファイルの作成の詳細については、この付録の「メッセージスタイル Web サービス WSDL ファイルの作成」を参照してください。

8. wsdl_dir サブディレクトリ内に、前の手順で作成した WSDL JSP へのリンクを含む index.html ファイルおよび後の手順で作成するクライアント JAR ファイルを作成します。次の例は簡単な index.html ファイルを示します。

   ```
   <html>
   <body>
   <h3>Web Service Name</h3>
   <ul>
   <li><a href='wsdl.jsp'>WSDL</a></li>
   <li><a href='../client.jar'>client.jar</li>
   </ul>
   ```

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

9. メイン ステージング ディレクトリに `client.jar` ファイルを作成します。このファイルの作成の詳細については、この付録の「client.jar ファイルの手動による作成」を参照してください。

   注意： この手順は省略可能です。Java 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には、`client.jar` ファイルを作成するだけです。

10. このエンタープライズ アプリケーション アーカイブ内で Web サービスをリストするメイン ステージング ディレクトリ内に `index.html` ファイルを作成し、および前の手順で作成した `index.html` ファイルへのリンクを設定します。次の例は簡単な `index.html` ファイルを示します。

```
<html>
<body>
<h3>Message-Style Web Services</h3>
<ul>
<li><a href='/context/wsdl_dir/index.html'>wsdl_dir</a></li>
</ul>
</body>
</html>
```

   この例では、*context* は `application.xml` ファイル（後の手順で作成する）内の Web アプリケーションの `context-root` 要素を指し、*wsdl_dir* は前の手順で作成した WSDL ファイルを含むサブディレクトリの名前を指します。

11. 次のような jar コマンドを使用して、Web アプリケーション アーカイブ（`*.war` ファイル）を作成します。

```
jar cvf web-app-name.war -C staging-dir .
```

   注意： `wsgen` Java ant タスクでは、デフォルト名 `web-services.war` を Web アプリケーション `*.war` ファイルに割り当てます。この命名規約に従う必要はありません。

12. エンタープライズ アプリケーションをアセンブルする 2 番目の一時的なステージング ディレクトリを作成します。このディレクトリには、任意の名前を付けることができます。

13. 前の手順で作成した Web アプリケーション アーカイブ `*.war` ファイルを手順 12. で作成したステージング ディレクトリにコピーします。

14. 次のコマンドを実行すると、WEB-INF サブディレクトリに初期の application.xml デプロイメント記述子が自動的に作成されます。

    ```
    java weblogic.ant.taskdefs.ear.DDInit staging-dir
    ```

    staging-dir は、手順 12. で作成したステージング ディレクトリを指します。

    デプロイメント記述子を生成する Java ベースの DDInit ユーティリティの詳細については、『WebLogic Server アプリケーションの開発』を参照してください。

15. WebLogic Web サービス情報を追加して、META-INF\application.xml ファイルを編集します。詳細については、この付録の「メッセージスタイル Web サービスの application.xml ファイルの更新」を参照してください。

16. 次のような jar コマンドを使用して、アプリケーションのエンタープライズ アーカイブ（.ear ファイル）を作成します。

    ```
    jar cvf application.ear -C staging-dir .
    ```

    作成された .ear ファイルは、Administration Console または weblogic.deploy コマンド ライン ユーティリティを使用して WebLogic Web サービスとしてデプロイできます。

メッセージスタイル Web サービス WSDL ファイルの作成

すべてのメッセージスタイル WebLogic Web サービス用の WSDL JSP ファイルは非常に似ています。これは、これらのタイプの Web サービスが実行するのは、データをクライアント アプリケーションとの間で送信または受信するという 2 つの操作だけだからです。

メッセージスタイル Web サービス用の WSDL JSP を作成するには、次の手順に従います。

1. テキスト エディタで wsdl.jsp という名前のファイルを作成します。
2. この節の最後にあるサンプル WSDL をコピーして wsdl.jsp ファイルに貼り付けて、以降の手順に従って編集します。

サンプル WSDL では、特定の Web サービスに合わせて変更する必要がある部分は太字で示されています。

3. `myService` への参照を自身の Web サービスの名前に置換します。
4. 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その Web サービスからメッセージを受け取る場合、`send` を `receive` に置換します。
5. `url:local` を自身の Web サービスの固有のネームスペースに置換します。
6. Web サービスを呼び出すために使用する URI を `/msg/sendMsg` から次の URI に置き換えます。

   /*context-root*/*url-pattern*

   context-root は `application.xml` デプロイメント記述子の `<context-root>` 要素を指し、*url-pattern* は `web.xml` デプロイメント記述子内の SOAP サーブレットの `<url-pattern>` を指します。
7. WSDL ファイルで Web サービスをホストする WebLogic Server のホストおよびポートを静的に指定する場合、テキスト `<%= request.getServerName() %>:<%= request.getServerPort() %>` をハードコード化されたホスト値およびポート値と置き換えて、WSDL JSP 内の `soap:address` 要素を編集します。

次のサンプル WSDL JSP を WSDL JSP の基本として使用してください。

```
<?xml version="1.0"?>
<definitions
    targetNamespace="urn:local"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="urn:local"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" >

  <types>
    <schema targetNamespace='urn:local'
            xmlns='http://www.w3.org/1999/XMLSchema'>
    </schema>
  </types>

  <message name="sendRequest">
    <part name="message" type="xsd:anyType" />
  </message>
  <message name="sendResponse">
  </message>
```

```
  <portType name="myServicePortType">
    <operation name="send">
      <input message="tns:sendRequest"/>
      <output message="tns:sendResponse"/>
    </operation>
  </portType>

  <binding name="myServiceBinding" type="tns:myServicePortType">
   <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http/"/>
    <operation name="send">
      <soap:operation soapAction="urn:send"/>
      <input>
        <soap:body use="encoded" namespace='urn:myService'
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded" namespace='urn:myService'
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
  </binding>

  <service name="myService">
    <documentation>todo</documentation>
    <port name="myServicePort" binding="tns:myServiceBinding">
      <soap:address location="http://<%= request.getServerName() %>:<%=
request.getServerPort() %>/msg/sendMsg"/>
    </port>
  </service>

</definitions>
```

メッセージスタイル Web サービスの web.xml ファイルの更新

この節では、メッセージスタイル WebLogic Web サービス アーカイブ ファイル内の SOAP サーブレットを参照する Web アプリケーション用の `web.xml` デプロイメント記述子に更新または追加する必要がある要素について説明します。`web.xml` デプロイメント記述子の完全な例については、この節の最後の例を参照してください。

ここでは Web アプリケーションとデプロイメント記述子に関して基本的な知識があることを前提としています。詳細については、『Web アプリケーションのアセンブルとコンフィグレーション』を参照してください。

メッセージスタイル Web サービス用の `web.xml` ファイルを更新するには、次の要素を追加します。

- メッセージスタイル Web サービスとクライアント アプリケーション間の SOAP メッセージを管理する SOAP サーブレットを参照する `<servlet>` 要素。`<servlet-class>` サブ要素を、JMS の送り先がトピックかキューか、またはサービスを呼び出すクライアントがメッセージを送信するか受信するかによって、次のサーブレット クラスの 1 つに設定します。
  - `weblogic.soap.server.servlet.DestinationSendAdapter` — サービスと、メッセージを JMS トピックまたはキューのどちらかに送信するクライアント アプリケーション間の SOAP メッセージを処理します。
  - `weblogic.soap.server.servlet.QueueReceiveAdapter` — サービスとメッセージを JMS キューから受信するクライアント アプリケーション間の SOAP メッセージを処理します。
  - `weblogic.soap.server.servlet.TopicReceiveAdapter` — サービスと、メッセージを JMS トピックから受信するクライアント アプリケーション間の SOAP メッセージを処理します。

  この `<servlet>` 要素には、JMS の送り先クラスを参照する要素と、JMS 接続ファクトリ クラスを参照する要素の 2 つの `<init-params>` 要素があります。

  次の例は SOAP サーブレットに対する `<servlet>` 参照を示します。

```
<servlet>
  <servlet-name>myService</servlet-name>
  <servlet-class>
       weblogic.soap.server.servlet.DestinationSendAdapter
  </servlet-class>
  <init-param>
    <param-name>topic-resource-ref</param-name>
    <param-value>myServiceDestination</param-value>
  </init-param>
  <init-param>
    <param-name>connection-factory-resource-ref</param-name>
    <param-value>myServiceFactory</param-value>
  </init-param>
</servlet>
```

- すべての SOAP 障害を処理する SOAP サーブレットを参照する <servlet> 要素。次の例で示すように <servlet-class> 要素を weblogic.soap.server.servlet.FaultHandler に設定します。

```
<servlet>
  <servlet-name>myServiceFault</servlet-name>
  <servlet-class>
        weblogic.soap.server.servlet.FaultHandler
  </servlet-class>
</servlet>
```

- 次の例で示す、WSDL JSP を参照する <servlet> 要素。

```
<servlet>
  <servlet-name>myServiceWSDL</servlet-name>
  <jsp-file>/myService/wsdl.jsp</jsp-file>
</servlet>
```

  JSP ファイル <jsp-file> へのパスは、この付録の「メッセージスタイル Web サービス アーカイブ ファイルの手動によるアセンブル」で作成した WSDL JSP への Web アプリケーション アーカイブ ファイル内のパスです。

- 上の例の各 <servlet> 要素に対して、次の例に示すようにサーブレットへの URL をマップする <servlet-mapping> 要素を作成します。

```
<servlet-mapping>
  <servlet-name>myServiceFault</servlet-name>
  <url-pattern>/weblogic/webservice/fault</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>myServiceWSDL</servlet-name>
  <url-pattern>/myService/myService.wsdl</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>myService</servlet-name>
  <url-pattern>/sendMsg</url-pattern>
</servlet-mapping>
```

- <error-page> 要素。正確には次のようになります。

```
<error-page>
    <exception-type>
       weblogic.soap.FaultException
    </exception-type>
    <location>/weblogic/webservice/fault</location>
  </error-page>
```

- 次の例に示すように、最初の <servlet> 要素内の JMS の送り先参照と接続ファクトリ参照を JNDI 内の Java オブジェクトにリンクする 2 つの <resource-ref> 要素。

```
<resource-ref>
  <res-ref-name>myServiceDestination</res-ref-name>
  <res-type>javax.jms.Destination</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
<resource-ref>
  <res-ref-name>myServiceFactory</res-ref-name>
  <res-type>javax.jms.ConnectionFactory</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

次のサンプル web.xml デプロイメント記述子には、メッセージスタイル Web サービスの例 examples.webservices.message 用の要素があります。

```
<!DOCTYPE web-app
          PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
          "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
  <servlet>
    <servlet-name>myService</servlet-name>
    <servlet-class>
         weblogic.soap.server.servlet.DestinationSendAdapter
    </servlet-class>
    <init-param>
      <param-name>topic-resource-ref</param-name>
      <param-value>myServiceDestination</param-value>
    </init-param>
    <init-param>
      <param-name>connection-factory-resource-ref</param-name>
      <param-value>myServiceFactory</param-value>
    </init-param>
  </servlet>
  <servlet>
    <servlet-name>myServiceFault</servlet-name>
    <servlet-class>weblogic.soap.server.servlet.FaultHandler</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>myServiceWSDL</servlet-name>
    <jsp-file>/myService/wsdl.jsp</jsp-file>
  </servlet>
  <servlet-mapping>
    <servlet-name>myServiceFault</servlet-name>
    <url-pattern>/weblogic/webservice/fault</url-pattern>
```

```
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>myServiceWSDL</servlet-name>
    <url-pattern>/myService/myService.wsdl</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>myService</servlet-name>
    <url-pattern>/sendMsg</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
  <error-page>
    <exception-type>weblogic.soap.FaultException</exception-type>
    <location>/weblogic/webservice/fault</location>
  </error-page>
  <resource-ref>
    <res-ref-name>myServiceDestination</res-ref-name>
    <res-type>javax.jms.Destination</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
  <resource-ref>
    <res-ref-name>myServiceFactory</res-ref-name>
    <res-type>javax.jms.ConnectionFactory</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
</web-app>
```

メッセージスタイル Web サービスの weblogic.xml ファイルの更新

メッセージスタイル Web サービス用の `weblogic.xml` デプロイメント記述子は、Web サービス固有の要素を持ちませんが、JMS の送り先と JMS 接続ファクトリへの標準的な参照があります。

次のサンプル `weblogic.xml` デプロイメント記述子には、メッセージスタイル Web サービスの例 `examples.webservices.message` 用の要素があります。

```
<!DOCTYPE weblogic-web-app
        PUBLIC "-//BEA Systems, Inc.//DTD Web Application 6.0//EN"
        "http://www.beasys.com/j2ee/dtds/weblogic-web-jar.dtd">

<weblogic-web-app>
  <reference-descriptor>
```

```
    <resource-description>
      <res-ref-name>myServiceDestination</res-ref-name>
      <jndi-name>examples.soap.msgService.MsgSend</jndi-name>
    </resource-description>
    <resource-description>
      <res-ref-name>myServiceFactory</res-ref-name>
      <jndi-name>examples.soap.msgService.MsgConnectionFactory</jndi-name>
    </resource-description>
  </reference-descriptor>
</weblogic-web-app>
```

メッセージスタイル Web サービスの application.xml ファイルの更新

メッセージスタイル Web サービス用の application.xml デプロイメント記述子には、SOAP サーブレットを含む Web アプリケーションへの標準的な参照が含まれます。

1 つの Web サービス関連要素は、<web> 要素の <context-root> サブ要素です。<context-root> 要素の値は、WSDL、ホーム ページ、または Web サービス自体にアクセスするすべての URL で使用されます。

次のサンプル application.xml デプロイメント記述子には、メッセージスタイル Web サービスの例 examples.webservices.message 用の要素があります。

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

<application>
  <display-name>Web-services</display-name>
  <module >
    <web>
      <web-uri>web-services.war</web-uri>
      <context-root>/msg</context-root>
    </web>
  </module>
</application>
```

client.jar ファイルの手動による作成

Java client.jar ファイルには、次のオブジェクトが含まれます。

- WebLogic FastParser（ハイパフォーマンス XML パーサ）。
- WebLogic Web サービス クライアント API。
- RPC スタイル Web サービスを実装するステートレス セッション EJB のリモート インタフェース。このオブジェクトは省略可能で、静的クライアントを使用してサービスを呼び出す場合にのみ必要になります。
- EJB パラメータまたは戻り値として使用される JavaBean 用のクラス ファイル。

BEA では `wsgen` Java Ant タスクを使用して最初の `*.ear` ファイルを作成し、次に `*.ear` ファイルに内に含まれている Java client.jar ファイルを抽出してユーザ固有の Web サービス用に修正することをお勧めします。`wsgen` の使用方法の詳細については、2-21 ページの「WebLogic Web サービスのアセンブル」を参照してください。

D WSDL ファイルを使用しない Web サービスの呼び出し

この付録では、WebLogic Web サービスを呼び出すときに WSDL ファイルを使用しない動的クライアント アプリケーションの例を示します。この例では特に、メッセージ スタイル Web サービスを呼び出して、データを WebLogic Server に送信します。

Web サービスの WSDL を使用しない動的クライアント アプリケーションは、すべてにおいて動的です。これは Web サービスのインタフェース、または戻り値およびパラメータの JavaBean インタフェースのいずれか、あるいは Web サービスを構成するメソッドの数およびシグネチャが不明でも Web サービスを呼び出すことができることによります。

この例では、Web サービスの呼び出しに URL `http://www.myHost.com:7001/msg/sendMsg` を使用します。この例では Web サービスの WSDL を使用しない動的クライアント アプリケーションを示しているので、前述の URL は Web サービスの *WSDL* ではなく、Web サービス自体を表します。

例の後の手順では、このクライアントを作成するために実行する基本手順の一部として、例に関連する節について説明します。

```
import java.util.Properties;
import java.net.URL;
import javax.naming.Context;
import javax.naming.InitialContext;

import weblogic.soap.WebServiceProxy;
import weblogic.soap.SoapMethod;
import weblogic.soap.SoapType;
import weblogic.soap.codec.CodecFactory;
import weblogic.soap.codec.SoapEncodingCodec;
import weblogic.soap.codec.LiteralCodec;

public class ProducerClient{

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

```
    CodecFactory factory = CodecFactory.newInstance();
    factory.register( new SoapEncodingCodec() );
    factory.register( new LiteralCodec() );

    WebServiceProxy proxy = WebServiceProxy.createService(
       new URL( "http://www.myHost.com:7001/msg/sendMsg" ) );
    proxy.setCodecFactory( factory );
    proxy.setVerbose( true );

    SoapType param = new SoapType( "message", String.class );
    proxy.addMethod( "send", null, new SoapType[]{ param } );
    SoapMethod method = proxy.getMethod( "send" );

    String toSend = arg.length == 0 ? "No arg to send" : arg[0];
    Object result = method.invoke( new Object[]{ toSend } );
  }
}
```

データを WebLogic Server に送信するメッセージ スタイル WebLogic Web サービスの呼び出しに WSDL を使用しない動的 Java クライアントを作成するには、次の手順に従ってください。

1. WebLogic Web サービスをホストする WebLogic Server から、Java クライアント JAR ファイルを取得します。

   この手順の詳細については、3-7 ページの「Java クライアント JAR ファイルの Web サービス ホーム ページからのダウンロード」を参照してください。

2. クライアント コンピュータ上の CLASSPATH に、Java クライアント JAR ファイルを追加します。

3. クライアント Java プログラムを作成します。次の手順では、Web サービス固有の Java コードについて説明します。

   a. クライアント アプリケーションの main メソッドで、エンコーディング スタイルのファクトリを作成し、WebLogic Server でサポートされる 2 つのスタイル（SOAP エンコーディング スタイルおよび Apache のリテラル XML エンコーディング スタイル）を登録します。

```
CodecFactory factory = CodecFactory.newInstance();
factory.register( new SoapEncodingCodec() );
factory.register( new LiteralCodec() );
```

   b. 次の Java コードを追加して Web サービスへの接続を作成し、エンコーディング スタイル ファクトリを設定します。

```
WebServiceProxy proxy = WebServiceProxy.createService(
      new URL( "http://www.myHost.com:7001/msg/sendMsg" ) );
```

```
proxy.setCodecFactory( factory );
proxy.setVerbose( true );
```

c. 次の Java コードを追加して、Web サービスの `send` メソッドを動的に取得します。

```
SoapType param = new SoapType( "message", String.class );
proxy.addMethod( "send", null, new SoapType[]{ param } );
SoapMethod method = proxy.getMethod( "send" );
```

d. `send` メソッドを呼び出して、データを Web サービスに送信します。この例では、クライアント アプリケーションは単純に最初の引数を受け取って `String` として送信します。ユーザーが特定の引数を指定しない場合、クライアント アプリケーションでは文字列 `No arg to send` を送信します。

```
String toSend = arg.length == 0 ? "No arg to send" : arg[0];
Object result = method.invoke( new Object[]{ toSend } );
```

4. 通常どおりにクライアント Java プログラムをコンパイルして実行します。

`org.w3c.dom.Document`、`org.w3c.dom.DocumentFragment`、または `org.w3c.dom.Element` の各データ型をパラメータとして受け付ける `send` メソッドの使い方を、以下のさらに複雑な例で示します。この例では、`send` メソッドのこのような考え方に沿ってリテラル エンコーディングを設定する方法を示しています。

```
import java.util.Properties;

import java.net.URL;
import java.io.File;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

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

import weblogic.apache.xml.serialize.OutputFormat;
import weblogic.apache.xml.serialize.XMLSerializer;

import weblogic.apache.xerces.dom.DocumentImpl;
```

```
import weblogic.soap.WebServiceProxy;
import weblogic.soap.SoapMethod;
import weblogic.soap.SoapType;

import weblogic.soap.codec.CodecFactory;
import weblogic.soap.codec.SoapEncodingCodec;
import weblogic.soap.codec.LiteralCodec;

public class ProducerClient{
  public static void main(String[] args) throws Exception{
    String url = "http://localhost:7001";
    // 引数リストを解析する
    if (args.length != 2) {
     System.out.println("Usage: java examples.webservices.message.ProducerClient
http://hostname:port \"message\"");
      return;
    } else if (args.length == 2) {
      url = args[0];
    }

    CodecFactory factory = CodecFactory.newInstance();
    factory.register(new SoapEncodingCodec());
    factory.register(new LiteralCodec());

    URL newURL = new URL(url + "/msg/sendMsg");

    WebServiceProxy proxy = WebServiceProxy.createService(newURL);
    proxy.setCodecFactory(factory);
    proxy.setVerbose(true);
    SoapType param = new SoapType( "message", Document.class );
    proxy.addMethod( "send", null, new SoapType[]{ param } );

    SoapMethod method = proxy.getMethod("send");

    // proxy を出力して、メソッド シグネチャが問題なさそうか確かめる
    System.out.println("Proxy:"+proxy);

    DocumentBuilderFactory dbf =
                     DocumentBuilderFactory.newInstance();
    // ファクトリから DocumentBuilder のインスタンスを取得する
    DocumentBuilder db = dbf.newDocumentBuilder();
    // ドキュメントを解析する
    Document w3cDoc = db.parse(new File("/test/fdr_nodtd.xml"));

    //Class parserClass = Class.forName("org.jdom.adapters.XercesDOMAdapter");
    //DOMAdapter da = (DOMAdapter)parserClass.newInstance();
    //Document w3cDoc = da.getDocument(new File("/test/fdr_nodtd.xml"),false);

    // XML を出力して、ドキュメントが正常に読み込まれたかどうか確かめる
    OutputFormat of = new OutputFormat();
```

```
    of.setEncoding("UTF-8");
    of.setLineWidth(40);
    of.setIndent(4);
    XMLSerializer xs = new XMLSerializer(System.out,of);
    xs.serialize(w3cDoc);

    System.out.println("Before Invoke");
    Object result = method.invoke( new Object[]{w3cDoc} );
    System.out.println("Done");
  }
}
```

用語集

Web サービスのアセンブル

Web サービスのすべてのコンポーネントをエンタープライズ アーカイブ ファイル（*.ear）にパッケージ化すること。Java Ant タスクを使用して WebLogic Web サービスをアセンブルします。

Web サービスのデプロイ

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すること。これは、全く同じではありませんが、EJB のデプロイに類似しています。Web サービスを構成する EJB をデプロイした後に Web サービスをデプロイします。Administration Console を使用して WebLogic Web サービスをデプロイします。

Web サービスの実装

Web サービスのエントリ ポイントとして定義されるステートレス セッション EJB（RPC スタイル Web サービス用）またはメッセージ駆動型 Bean（メッセージスタイル Web サービス用）の Java コードの記述。ステートレス セッション EJB またはメッセージ駆動型 Bean では、すべての Web サービス機能を含む場合と、他の EJB を呼び出して作業を分配する場合があります。

Web サービスの呼び出し

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。クライアントは、最初に、呼び出す Web サービスに関する SOAP メッセージをアセンブルし、必要なすべてのデータを SOAP 本文または添付ファイルに含めます。次にクライアントは SOAP メッセージを HTTP/HTTPS を介して WebLogic Server に送信します。そこで Web サービスが実行され、SOAP メッセージが HTTP/HTTPS を介してクライアントに返される場合と返されない場合があります。

Java Ant

WebLogic Web サービスをエンタープライズ アプリケーション アーカイブにアセンブルするために使用する Java ユーティリティ。

メッセージスタイル Web サービス

JMS の送り先をエントリ ポイントとして使用する Web サービスのタイプ。メッセージスタイル Web サービスは疎結合なドキュメント駆動型サービスです。これは、クライアントが一般にこのタイプの Web サービスを使用する場合、パラメータではなく、Web サービスによって処理されるドキュメント全体を送信して、戻り値を受け取ることを意味します。

Web サービスの発行

Web サービスをわかりやすい場所に登録して、ユーザが容易に見つけられるようにすること。これは、Web サービスを UDDI レジストリに登録し、Web サービスを呼び出す URL を希望するユーザに電子メールで知らせるなどの方法によって実行できます。

RPC スタイル Web サービス

ステートレス セッション EJB をエントリ ポイントとして使用する Web サービスのタイプ。RPC スタイル Web サービスは密結合なインタフェース駆動型サービスです。これは、クライアントが一般に Web サービスを使用する場合、Web サービスによって処理されるドキュメント全体を送信するのではなく、パラメータを送信して、戻り値を受け取ることを意味します。

SOAP

Simple Object Access Protocol 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。

添付ファイル付き SOAP

転送用のマルチパート MIME 構造体において SOAP メッセージを任意の数の添付ファイルにネイティブ フォーマットで関連付ける標準的な方法を説明する仕様。

Web サービス

異機種ユーザが Web 上でアクセスする、特定の機能をカプセル化する共有アプリケーション。

Web サービス ホーム ページ

特定のコンテキスト用に定義された Web サービスを WSDL ファイルおよび各 Web サービスに関連付けられている Java クライアント JAR ファイルとともに一覧表示する Web ページ。

WSDL

Web サービス記述言語。Web サービスを記述するために使用される XML ベースの言語です。

索引

A

Administration Console
  JMS コンポーネントのコンフィグレーション 2-19, 2-20
  Web サービスの表示 4-2
  呼び出し 4-1
Ant 1-6, 2-21, B-1

B

BEA XML エディタ 1-15
build.xml ファイル
  階層図 B-2
  作成 2-26
  要素と属性 B-1
  例 2-23, 2-36, 3-17, B-1
build.xml ファイルの要素
  clientjar B-9
  entry B-10
  manifest B-9
  messageservice B-7
  messageservices B-7
  rpcservice B-6
  rpcservices B-5
  wsgen B-3

C

clientjar、build.xml ファイルの要素 B-9

D

DestinationSendAdapter サーブレット 2-15

E

EJB
  RPC スタイル Web サービスでのセキュリティ 2-16
  RPC スタイル Web サービスの実装 1-10, 1-12, 2-5
  アーカイブ ファイル（*.jar）1-11
  アーカイブ ファイルへのアセンブル 2-35
  コード例 2-30
  デプロイメント記述子 2-34
ejb-jar.xml デプロイメント記述子 2-34, B-6, 2-25
entry、build.xml ファイルの要素 B-10

I

INITIAL_CONTEXT_FACTORY 3-11, 3-14, 3-19, 3-22

J

java.io.FileNotFoundException 5-2
java.lang.NullPointerException 5-6
java.net.ConnectException 5-7
javap ユーティリティ 3-12, 3-15
JMS
  destination 1-14, 2-19
  キューまたはトピックの選択 2-7
  コンポーネントのコンフィグレーション 2-20
  接続ファクトリ 2-19
  メッセージスタイル Web サービスとの関係 2-6
  リスナ 1-10

M

manifest、build.xml ファイルの要素 B-9

messageservice、build.xml ファイルの要素 B-7
messageservices、build.xml ファイルの要素 B-7
Microsoft SOAP Toolkit クライアント 3-15

Q

QueueReceiveAdapter サーブレット 2-15

R

RPC スタイル Web サービス
  EJB の設計 2-5
  Microsoft Toolkit クライアントからの呼び出し 3-15
  既存 EJB の変換 2-5
  実装 2-18
  使用する状況 2-4
  静的クライアントの記述 3-10
  セキュリティ 2-16
  説明 1-7
  動的クライアントの記述 3-13
  呼び出し 3-9
rpcservice、build.xml ファイルの要素 B-6
rpcservices、build.xml ファイルの要素 B-5
RPC スタイル Web サービス
  アーキテクチャ 1-11

S

SOAP
  エンコーディング 3-25
  エンコーディング スタイル 2-12
  サポートされていない機能 1-14
  仕様 A-1
  障害 3-23
  定義 1-4
  例 1-4
SOAP サーブレット
  セキュリティ 2-14
  説明 1-10
  ロール 1-12, 1-14
StatelessBeanAdapter サーブレット 2-15

T

TopicReceiveAdapter サーブレット 2-15

V

verbose モード 5-1

W

Web サービス
  コンポーネント 1-3
  使用する理由 1-2
  定義 1-1
web.xml 1-8, 2-14, 2-15
WebLogic FastParser 3-7
WebLogic Web サービス
  Administration Console を使用した表示 4-2
  Microsoft SOAP Toolkit クライアントからの呼び出し 3-15
  アーキテクチャ 1-10
  アセンブル 2-21
  エンコーディング スタイル 2-12, 3-25
  主な開発手順 2-1
  開発の例 2-28
  管理 4-1
  機能 1-6
  クライアント API を使用した呼び出し 1-9
  サポートされている仕様 A-1
  サポートされているデータ型 2-9
  実行時コンポーネント 1-8
  実装 2-18
  初期コンテキスト ファクトリ プロパティ 3-24
  セキュリティ 2-14
  設計 2-3
  デプロイメント 2-28
  標準アセンブリとデプロイメント 1-8
  プログラミング モデル 1-6

呼び出し 3-2, D-1
呼び出し用の URL 3-8
呼び出しを行うクライアントの例 3-4
例 1-9
例外処理 3-23
WebLogic Web サービス クライアント API
サポートされているモード 3-3
説明 3-3
WebLogic Web サービスのアセンブル 2-21
WebLogic Web サービス ホーム ページ
WSDL の取得 3-6
クライアント JAR ファイルの取得 3-7
呼び出し 3-5
weblogic.xml 1-8
weblogic-ejb-jar.xml デプロイメント記述子 3-8, C-4
Web アーカイブ ファイル（*.war）1-11
WSDL
WebLogic Web サービス ホーム ページからの取得 3-6
サポートされていない機能 1-15
取得用の URL 3-8
仕様 A-2
静的または動的 2-28
説明 1-5
例 1-5
wsgen Ant タスク
作成 2-36
説明 2-21
要素 B-1
wsgen、build.xml ファイルの要素 B-3

い

印刷、製品のマニュアル 1-x

え

エンコーディング スタイル
SOAP 2-12
リテラル XML 2-12
エンタープライズ アーカイブ ファイル（*.ear）1-11

お

オーバーロードされたメソッド、回避 2-6

か

カスタマ サポート情報 1-xi

き

キュー、JMS 2-7

く

クライアント JAR ファイル
ダウンロード 3-7
内容 3-7
必要な追加クラス 3-25

さ

サーブレット
DestinationSendAdapter 2-15
QueueReceiveAdapter 2-15
StatelessBeanAdapter 2-15
TopicReceiveAdapter 2-15
サポート
技術情報 1-xi

し

仕様
SOAP 1.1 A-1
WSDL 1.1 A-2
添付ファイル付き SOAP 1.1 A-1

せ

静的クライアント 3-3

と

動的クライアント 3-4
トピック、JMS 2-7

トラブルシューティング 5-1

ま

マニュアル、入手先 1-x

め

メッセージ駆動型 Bean 1-10, 1-14, 2-7, 2-19
メッセージスタイル Web サービス
JMS との関係 2-6
アーキテクチャ 1-13
既存の JMS アプリケーションの変換 2-9
キューまたはトピックの選択 2-7
実装 2-19
使用する状況 2-4
セキュリティ 2-14
説明 1-7
データ受信用のクライアントの記述 3-20
データ送信用のクライアントの記述 3-18
呼び出し 3-16
例 2-8

り

リテラル XML エンコーディング スタイル 2-12, 3-25

れ

例外
java.io.FileNotFoundException 5-2
java.lang.NullPointerException 5-6
java.net.ConnectException 5-7
解析不能 5-4
例外を解析不能 5-4