BEA WebLogic Server™

アプリケーションの開発

BEA WebLogic Server バージョン 6.1
マニュアルの日付：2002 年 6 月 24 日

著作権

限定的権利条項

商標または登録商標

WebLogic Server アプリケーションの開発

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 6 月 24 日	BEA WebLogic Server バージョン 6.1

目次

このマニュアルの内容

1. WebLogic Server J2EE アプリケーションについて

2. WebLogic Server J2EE アプリケーションの開発

3. WebLogic Server J2EE アプリケーションのパッケージ化

B. クライアント アプリケーションのデプロイメント記述子の要素

索引

このマニュアルの内容

このマニュアルでは、BEA WebLogic Server™ アプリケーションの開発環境を紹介します。開発環境の構築方法と、WebLogic Server プラットフォームでのデプロイメント用にアプリケーションをパッケージ化する方法について説明します。

このマニュアルの構成は次のとおりです。

- 第 1 章「WebLogic Server J2EE アプリケーションについて」では、WebLogic Server アプリケーションのコンポーネントについて説明します。
- 第 2 章「WebLogic Server J2EE アプリケーションの開発」では、WebLogic Server アプリケーションの高レベルな作成手順と、Java プログラマがプログラミング環境を構築する際に役立つ情報について概説します。
- 第 3 章「WebLogic Server J2EE アプリケーションのパッケージ化」では、WebLogic Server コンポーネントとアプリケーションを、配布およびデプロイメント用に標準の JAR ファイルにまとめる方法について説明します。
- 第 4 章「プログラミング トピック」では、メッセージのロギング方法やスレッドの使い方など、一般的な WebLogic Server アプリケーションのプログラミングの問題について説明します。
- 付録 A「application.xml デプロイメント記述子の要素」は、標準 J2EE エンタープライズ アプリケーション デプロイメント記述子（`application.xml`）のリファレンスです。
- 付録 B「クライアント アプリケーションのデプロイメント記述子の要素」は、標準 J2EE クライアント アプリケーション デプロイメント記述子（`application-client.xml`）、および WebLogic 固有のクライアント アプリケーション デプロイメント記述子のリファレンスです。

対象読者

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

e-docs Web サイト

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

このマニュアルの印刷方法

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

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

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

関連情報

BEA の Web サイトでは、WebLogic Server の全マニュアルを提供しています。以下の WebLogic Server ドキュメントには、WebLogic Server アプリケーションのコンポーネントの作成に関連する情報が含まれています。

- 『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』
- 『WebLogic HTTP サーブレット プログラマーズ ガイド』
- 『WebLogic JSP プログラマーズ ガイド』
- 『Web アプリケーションのアセンブルとコンフィグレーション』
- 『WebLogic JDBC プログラミング ガイド』
- 『WebLogic Server Web サービス プログラマーズ ガイド』
- 『WebLogic J2EE コネクタ アーキテクチャ』

Java アプリケーションの開発に関する一般情報については、Sun Microsystems, Inc. の Java 2, Enterprise Edition Web サイト（http://java.sun.com/products/j2ee/）を参照してください。

サポート情報

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

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

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

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

表記規則

このマニュアルでは、全体を通して以下の表記規則が使用されています。

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

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

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

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

1 WebLogic Server J2EE アプリケーションについて

以下の節では WebLogic Server J2EE アプリケーションとアプリケーション コンポーネントについて概説します。

- WebLogic Server J2EE アプリケーションおよびコンポーネント
- Web アプリケーション コンポーネント
- エンタープライズ JavaBean コンポーネント
- WebLogic Server コンポーネント
- コネクタ コンポーネント
- エンタープライズ アプリケーション
- クライアント アプリケーション

WebLogic Server J2EE アプリケーションおよびコンポーネント

BEA WebLogic Server™ アプリケーションは、WebLogic Server で動作する 1 つまたは複数の J2EE コンポーネントで構成されるアプリケーションです。J2EE コンポーネントには、次のものが含まれます。

- Web コンポーネント ― HTML ページ、サーブレット、JavaServer Pages、および関連ファイル
- EJB コンポーネント ― エンティティ Bean、セッション Bean、およびメッセージ駆動型 Bean
- WebLogic Server コンポーネント ― スタートアップ クラスとシャットダウン クラス
- コネクタ コンポーネント ― リソース アダプタ

Web デザイナ、アプリケーション開発者、およびアプリケーション アセンブル担当者は、JavaServer Pages、サーブレット、およびエンタープライズ JavaBean、リソース アダプタ などの J2EE 技術を使用してアプリケーションとそのコンポーネントを作成します。

コンポーネントは、Java ARchive (JAR) ファイルにパッケージ化します。JAR ファイルは、Java `jar` ユーティリティで作成されるアーカイブです。JAR ファイルでは、ディレクトリ内のすべてのコンポーネント ファイルがディレクトリ構造を維持しながら 1 つのファイルにまとめられます。JAR ファイルには、WebLogic Server にコンポーネントのデプロイ方法を指示する XML 記述子も格納されます。

Web アプリケーションは、拡張子が `.war` の JAR ファイルにパッケージ化されます。エンタープライズ Bean、WebLogic コンポーネント、およびクライアント アプリケーションは、拡張子が `.jar` の JAR ファイルにパッケージ化されます。リソース アダプタは、拡張子が `.rar` の JAR ファイルにパッケージ化されます。

アセンブル済みの Web アプリケーション、EJB コンポーネント、リソース アダプタで構成されるエンタープライズ アプリケーションは、拡張子が .ear の JAR ファイルにまとめられます。.ear ファイルには、アプリケーションのすべての .jar、.war、および .rar コンポーネント アーカイブ ファイルおよびコンポーネントを記述する XML 記述子が格納されます。

コンポーネント、アプリケーション、またはリソース アダプタをデプロイするには、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、JAR ファイルを対象の WebLogic Server にアップロードします。

Web ブラウザではないクライアント アプリケーションは、Remote Method Invocation（RMI）を使用して WebLogic Server に接続する Java クラスです。Java クライアントでは、エンタープライズ JavaBean、JDBC 接続、JMS メッセージなどのサービスに RMI を使用してアクセスできます。

J2EE プラットフォーム

WebLogic Server には、Java 2 Platform、Enterprise Edition（J2EE）技術が組み込まれています。J2EE は、Java プログラミング言語に基づいた多層エンタープライズ アプリケーションを開発するための標準プラットフォームです。J2EE を構成する技術は、BEA Systems をはじめとするソフトウェア ベンダと Sun Microsystems によって共同開発されました。

J2EE アプリケーションは、標準化され、モジュール化されたコンポーネントに基づいています。WebLogic Server では、これらのコンポーネント用にあらゆるサービスが用意され、細かなアプリケーションの動作を、プログラミングを必要とせずに自動的に処理します。

WebLogic Server 6.1 と J2EE 1.2 および J2EE 1.3

BEA WebLogic Server 6.1 は、高度な J2EE 1.3 の機能を実装する最初の e- コマース トランザクション プラットフォームです。J2EE のルールに準拠するために、BEA Systems では 2 つの別個のダウンロードを用意しています。1 つは J2EE 1.3 の機能が有効になっているもの、1 つは J2EE 1.2 の機能に制限されているものです。いずれのダウンロードもコンテナは同じですが、利用可能な API だけ異なります。

注意： J2EE コンポーネントのコンパイルの CLASSPATH 設定は、J2EE 1.2 完全準拠のコンポーネントを作成するか、J2EE 1.3 の機能を採用したコンポーネントを作成するかどうかによって異なります。詳細については、2-19 ページの「コンパイル用のクラスパスの設定」を参照してください。

J2EE 1.2 の機能に加えて J2EE 1.3 の機能を備える WebLogic Server 6.1

このダウンロードでは、WebLogic Server はデフォルトで J2EE 1.3 の機能を使用して動作します。それらの機能には、EJB 2.0、JSP 1.2、サーブレット 2.3、および J2EE コネクタ アーキテクチャ 1.0 が含まれます。J2EE 1.3 の機能を有効にして WebLogic Server 6.1 を実行しても、J2EE 1.2 アプリケーションはそのままフルサポートされます。J2EE 1.3 機能の実装では、適切な API 仕様の最終ではないバージョンが使用されます。したがって、J2EE 1.3 の新機能を使用する BEA WebLogic Server 6.1 用に開発されたアプリケーション コードは、BEA WebLogic Server の今後のリリースでサポートされる J2EE 1.3 プラットフォームとは互換性を持たない場合があります。

J2EE 1.2 認定の WebLogic Server 6.1

このダウンロードでは、WebLogic Server はデフォルトで J2EE 1.3 機能が無効な状態で動作し、J2EE 1.2 の仕様と規定に完全に準拠します。

Web アプリケーション コンポーネント

Web アーカイブには、Web アプリケーションを構成するファイルが格納されます。`.war` ファイルは、1 つまたは複数の WebLogic Server にユニットとしてデプロイされます。

WebLogic Server の Web アーカイブには、常に以下のファイルが含まれています。

- ヘルパー クラスとともに最低 1 つのサーブレットまたは JSP ページ
- `web.xml` デプロイメント記述子（`.war` ファイルの内容を記述する J2EE 標準の XML ドキュメント）
- `weblogic.xml` デプロイメント記述子（Web アプリケーションの WebLogic Server 固有の要素が格納される XML ドキュメント）

Web アーカイブには、HTML ページまたは XML ページ、およびそれらに付属する画像やマルチメディア ファイルなどのサポート ファイルが含まれている場合もあります。

サーブレット

サーブレットは、WebLogic Server で実行される Java クラスであり、クライアントから要求を受け取り、その要求を処理して、必要に応じてクライアントに応答を返します。GenericServlet は、プロトコルに依存せず、他の Java クラスからアクセスされるサービスを実装するために J2EE アプリケーションで使用できます。HttpServlet は、HTTP プロトコルのサポートで GenericServlet を拡張します。HttpServlet は主に、Web ブラウザの要求に応えて動的な Web ページを生成するために使用します。

JavaServer Pages

JSP ページは、Java コードを Web ページに埋め込むことができる拡張 HTML で記述された Web ページです。JSP ページでは、HTML に似たタグを使用して、taglibs と呼ばれるカスタム Java クラスを呼び出すことができます。WebLogic JSP コンパイラの `weblogic.jspc` では、JSP ページがサーブレットに変換されます。WebLogic Server では、サーブレット クラス ファイルが存在しないか、または JSP ソース ファイルよりもタイムスタンプが古い場合に JSP ページが自動的にコンパイルされます。

サーバでのコンパイルを避けるために、あらかじめ JSP ページをコンパイルし、サーブレット クラスを Web アーカイブにパッケージ化することもできます。サーブレットと JSP ページは、Web アプリケーションと一緒にデプロイしなければならないヘルパー クラスに依存する場合があります。

Web アプリケーションのディレクトリ構造

Web アプリケーション コンポーネントは、`jar` コマンドを使って作成される `.war` ファイルをステージングするために各ディレクトリにアセンブルします。HTML ページ、JSP ページといったこれらのコンポーネントから参照される Java クラス以外のファイルは、ステージング ディレクトリの最上位から順にアクセスされます。

XML 記述子およびコンパイル済みの Java クラスと JSP taglibs は、ステージング ディレクトリの最上位に位置する WEB-INF サブディレクトリに格納します。Java クラスとしては、サーブレット、ヘルパー クラス、およびコンパイル済みの JSP ページ（必要な場合）などがあります。

ステージングが終了したら、jar コマンドを使用してディレクトリ全体を .war ファイルにまとめます。.war ファイルは、それだけでデプロイすることも、他の Web アプリケーション、EJB コンポーネント、WebLogic コンポーネントといった他のアプリケーション コンポーネントと一緒にエンタープライズ アーカイブ（.ear ファイル）にパッケージ化することもできます。

Web アプリケーション コンポーネントの詳細について

Web アプリケーション コンポーネントの作成の詳細については、以下のマニュアルを参照してください。

- 『WebLogic HTTP サーブレット プログラマーズ ガイド』
- 『WebLogic JSP プログラマーズ ガイド』
- 『JSP Tag Extensions プログラマーズ ガイド』
- 『Web アプリケーションのアセンブルとコンフィグレーション』

エンタープライズ JavaBean コンポーネント

エンタープライズ JavaBean（EJB）は、ビジネス タスクまたはエンティティを実装するサーバサイド Java コンポーネントで、EJB 仕様に基づいて記述されています。エンタープライズ Bean には、セッション Bean、エンティティ Bean、およびメッセージ駆動型 Bean の 3 種類があります。

EJB の概要

セッション Bean は、単一のセッション時に単一のクライアントに代わって特定のビジネス タスクを実行します。セッション Bean は、ステートフルにもステートレスにもなりますが、永続的ではありません。クライアントでセッション Bean の利用が終わると、その Bean は消えてなくなります。

エンティティ Bean は、データ ストア（通常はリレーショナル データベース システム）のビジネス オブジェクトを表します。永続性（データのロードと保存）は、Bean で管理される場合とコンテナで管理される場合があります。データ オブジェクトをメモリ内で表現するだけではなく、エンティティ Bean にはそれらが表すビジネス オブジェクトの動作をモデル化するメソッドがあります。エンティティ Bean は、複数のクライアントで同時にアクセスでき、当然のごとく永続的です。

メッセージ駆動型 Bean は、EJB コンテナで動作し、JMS キューからの非同期メッセージを処理するエンタープライズ Bean です。JMS キューでメッセージが受信されると、メッセージ駆動型 Bean ではメッセージを処理するためにそれ自身のインスタンスがプールから割り当てられます。メッセージ駆動型 Bean は、クライアントとは関連付けられません。メッセージ駆動型 Bean では、到着したメッセージが処理されるだけです。JMS ServerSessionPool も同様の機能を備えていますが、EJB コンテナで動作するメリットを利用できません。

エンタープライズ Bean は、それらのコンパイル済みクラスと XML デプロイメント記述子が格納される JAR ファイルにまとめられます。

EJB のインタフェース

エンティティ Bean とセッション Bean には、Bean の開発者から提供されるリモート インタフェース、ホーム インタフェース、および実装クラスがあります。メッセージ駆動型 Bean は、EJB コンテナの外からはアクセスできないので、ホーム インタフェースまたはリモート インタフェースを必要としません。

リモート インタフェースでは、エンティティ Bean またはセッション Bean でクライアントが呼び出すことができるメソッドが定義されます。実装クラスは、リモート インタフェースのサーバサイドの実装です。ホーム インタフェースは、エンタープライズ Bean を作成、破棄、および検索するためのメソッドを備えています。クライアントでは、Bean のホーム インタフェースを通じてエンタープライズ Bean のインスタンスにアクセスします。

EJB のホーム インタフェースとリモート インタフェースおよび実装クラスは、EJB 仕様が実装されているどの EJB コンテナにも移植できます。EJB 開発者は、コンパイル済みの EJB インタフェースとクラスおよびデプロイメント記述子だけが格納されている JAR ファイルを提供できます。

EJB と WebLogic Server

J2EE では、EJB 仕様をサポートする EJB サーバの間でコンポーネントを確実に移植できるように、開発とデプロイメントのロールが明確に区別されます。WebLogic Server でエンタープライズ Bean をデプロイするときには、エンタープライズ Bean をリモートで実行できるようにするスタブ クラスとスケルトン クラスを生成するために WebLogic EJB コンパイラ `weblogic.ejbc` が動作している必要があります。

WebLogic のスタブとスケルトンでは、エンタープライズ Bean のロードバランシングとフェイルオーバを実現する WebLogic クラスタもサポートできます。`weblogic.ejbc` を実行してスタブ クラスとスケルトン クラスを生成し、それらを EJB JAR ファイルに追加するか、または `WebLogic Server` でデプロイメント時にコンパイラを実行して生成することもできます。

J2EE 指定のデプロイメント記述子 `ejb-jar.xml` では、EJB JAR ファイルにパッケージ化されたエンタープライズ Bean が記述されます。この記述子では、Bean のタイプと名前、そしてホーム インタフェースとリモート インタフェースおよ

び実装クラスの名前が定義されます。また、ejb-jar.xml デプロイメント記述子では、Bean のセキュリティ ロールおよび Bean のメソッドのトランザクション動作も定義されます。

追加のデプロイメント記述子では、WebLogic 固有のデプロイメント情報が提供されます。コンテナ管理のエンティティ Bean の weblogic-cmp-rdbms-jar.xml デプロイメント記述子では、Bean がデータベースのテーブルにマップされます。weblogic-ejb-jar.xml デプロイメント記述子では、クラスタ化やキャッシュのコンフィグレーションといった WebLogic Server 環境に固有の追加情報が提供されます。

エンタープライズ JavaBean の作成とデプロイメントについては、『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 Server コンポーネント

WebLogic Server コンポーネントとしては、スタートアップ クラスとシャットダウン クラスがあります。これらのクラスは、それぞれデプロイメントまたは停止の時点で実行される Java クラスです。

スタートアップ クラスには、WebLogic Server ネーミング ツリーに自動的に登録される RMI クラス、または WebLogic Server で実行できる他の Java クラスを使用できます。スタートアップ クラスを使用すると、WebLogic Server で新しいサービスを実装できます。たとえば、レガシー アプリケーションやリアルタイム フィーダーにアクセスできるようにするスタートアップ クラスを作成することもできます。

シャットダウン クラスは、WebLogic Server が停止するときに実行され、通常はスタートアップ クラスで取得されたリソースを解放するために使用します。

スタートアップ クラスとシャットダウン クラスは、WebLogic Server で Administration Console からコンフィグレーションできます。これらの Java クラスは、サーバの CLASSPATH で設定されていなければなりません。

コネクタ コンポーネント

WebLogic J2EE コネクタ アーキテクチャで中心となるコンポーネントは、「コネクタ」として機能するリソース アダプタです。コネクタ アーキテクチャを使用すると、エンタープライズ情報システム（EIS : Enterprise Information System）のベンダおよびサードパーティ アプリケーションの開発者は、Sun Microsystems の J2EE 1.3 仕様をサポートしているアプリケーション サーバでデプロイできるリソース アダプタを開発できます。リソース アダプタには、Java、および必要に応じて EIS との対話に必要なネイティブ コンポーネントが含まれます。

リソース アダプタが WebLogic Server 環境でデプロイされた場合、リモート EIS システムへのアクセス権がある堅牢な J2EE アプリケーションの開発が可能になります。WebLogic Server アプリケーションの開発者は、HTTP サーブレット、JavaServer Pages（JSP）、エンタープライズ JavaBean（EJB）、およびその他の API を使用して、EIS のデータとビジネス ロジックを使う統合アプリケーションを開発できます。

基本のリソース アーカイブ（`.rar`）またはデプロイメント ディレクトリは、そのままでは WebLogic Server にデプロイできません。最初に `weblogic-ra.xml` ファイルで、WebLogic Server 固有のデプロイメント プロパティを作成およびコンフィグレーションし、その XML ファイルをデプロイメントに追加する必要があります。

リソース アダプタの作成とデプロイメントについては、『WebLogic J2EE コネクタ アーキテクチャ』を参照してください。

エンタープライズ アプリケーション

エンタープライズ J2EE アプリケーションには、Web コンポーネントと EJB コンポーネント、デプロイメント記述子、およびアーカイブ ファイルが含まれます。エンタープライズ アーカイブ（`.ear`）ファイルには、Web アーカイブと EJB アーカイブが格納されます。`META-INF\application.xml` デプロイメント記述子には、各 Web コンポーネントおよび EJB コンポーネントのエントリのほか、セキュリティ ロールやアプリケーション リソース（データベースなど）を記述する追加エントリがあります。

ドメイン内の 1 つまたは複数の WebLogic Server に .ear ファイルをデプロイするには、WebLogic 管理サーバの 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ます。

クライアント アプリケーション

WebLogic Server コンポーネントにアクセスする、Java で記述されたクライアントサイド アプリケーションは、標準の I/O を使用する単純なコマンドライン ユーティリティから、Java Swing/AWT クラスを使用して構築された、高度な対話型の GUI アプリケーションまでさまざまです。

クライアント アプリケーションは、HTTP リクエストまたは RMI リクエストを使って、WebLogic Server コンポーネントを直接使用します。コンポーネントは、クライアント内ではなく、WebLogic Server 内で実行されます。

WebLogic Server Java クライアントを実行するには、クライアント コンピュータで、weblogic.jar ファイル、weblogic_sp.jar ファイル（WebLogic Server のサービス パック バージョンを使用している場合）、WebLogic Server 上のすべての RMI クラスとエンタープライズ Bean のリモート インタフェース、およびクライアント アプリケーション クラスが必要となります。

クライアントサイド アプリケーションは、クライアント コンピュータにデプロイできるようにパッケージ化します。保守とデプロイメントを簡略化するために、クライアントサイド アプリケーションは、weblogic.jar ファイルおよび weblogic_sp.jar ファイルと一緒にクライアントのクラスパスに追加できる JAR ファイルにパッケージ化したほうがよいでしょう。

WebLogic Server は、(単純な Java プログラムとは対照的に) 標準の XML デプロイメント記述子 (client-application.xml) および WebLogic 固有のデプロイメント記述子と一緒に JAR ファイルにパッケージ化される J2EE クライアント アプリケーションもサポートしています。この仕様に基づいてクライアント アプリケーションをパッケージ化するには、クライアント コンピュータで weblogic.ClientDeployer コマンドライン ユーティリティを実行します。J2EE クライアント アプリケーションの詳細については、3-15 ページの「クライアント アプリケーションのパッケージ化」を参照してください。

2 WebLogic Server J2EE アプリケーションの開発

以下の節では、エンタープライズ アプリケーション、Web アプリケーション、エンタープライズ JavaBean など、さまざまな WebLogic Server J2EE アプリケーションを作成する方法について説明します。

- アプリケーションの作成 : 主な手順
- エンタープライズ JavaBean の作成 : 主な手順
- WebLogic Server エンタープライズ アプリケーションの作成 : 主な手順
- リソース アダプタの作成 : 主な手順
- 開発環境の構築
- コンパイルの準備
- デプロイメント記述子の編集

WebLogic Server アプリケーションは、Java プログラマ、Web デザイナ、およびアプリケーション アセンブラによって作成されます。プログラマとデザイナは、アプリケーションのビジネス ロジックとプレゼンテーション ロジックを実装するコンポーネントを作成します。アプリケーション アセンブラは、コンポーネントをアセンブルして、WebLogic Server にデプロイ可能なアプリケーションを作成します。

アプリケーションの作成：主な手順

Web アプリケーションを作成するには、HTML ページ、JSP、サーブレット、JSP taglibs、および 2 つのデプロイメント記述子を作成してから、それらをすべて *.war ファイルにパッケージ化する必要があります。*.war ファイルは、Web アプリケーションとして WebLogic Server にデプロイされます。

Web アプリケーション作成の主な手順は次のとおりです。

1. Web アプリケーションの Web インタフェースを構成する HTML ページおよび JSP を作成します。通常、Web デザイナは、Web アプリケーションのこの部分を作成します。

   JSP 作成の詳細については、『WebLogic JSP プログラマーズ ガイド』を参照してください。

2. サーブレットと、JavaServer Pages（JSP）で参照される JSP taglibs 用の Java コードを記述します。通常、Java プログラマは、Web アプリケーションのこの部分を作成します。

   サーブレット作成の詳細については、『WebLogic HTTP サーブレット プログラマーズ ガイド』を参照してください。

3. クラス ファイルへのサーブレットのコンパイル

   コンパイルの詳細については、2-18 ページの「コンパイルの準備」を参照してください。

4. web.xml および weblogic.xml デプロイメント記述子を作成します。

   web.xml ファイルは各サーブレットと JSP ページを定義し、Web アプリケーションで参照されるエンタープライズ Bean を列挙します。weblogic.xml ファイルは、WebLogic Server 用の補足デプロイメント情報を追加します。

   手動で web.xml および weblogic.xml デプロイメント記述子を作成することも、WebLogic Server に含まれる Java ベース ユーティリティを使用して自動的に生成することもできます。これらのファイルの自動生成の詳細については、3-5 ページの「デプロイメント記述子の自動生成」を参照してください。

   デプロイメント記述子の要素、および手動で作成する方法については、『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。

5. HTML ページ、サーブレット クラス ファイル、JSP ファイル、web.xml、および weblogic.xml を Web アーカイブ（*.war）ファイルにパッケージ化します。

   .war ファイル作成の最初の手順は、Web アプリケーションのステージング ディレクトリを作成することです。JSP ページ、HTML ページ、およびそれらのページによって参照されるマルチメディア ファイルは、ステージング ディレクトリの最上位に保存されます。コンパイルされたサーブレット クラス、taglibs、および JSP ページからコンパイルされたサーブレット（必要な場合）は、ステージング ディレクトリの WEB-INF ディレクトリに格納されます。すべての Web アプリケーション コンポーネントをステージング ディレクトリに配置したら、JAR コマンドを実行して、.war ファイルを作成します。

   *.war ファイル作成の詳細については、3-7 ページの「Web アプリケーションのパッケージ化」を参照してください。

6. *.war ファイルをテスト目的で WebLogic Server に自動デプロイします。

   Web アプリケーションをテストしているときに、web.xml および weblogic.xml デプロイメント記述子を編集しなければならない場合があります。これは手動で編集することも、Administration Console のデプロイメント記述子エディタで編集することもできます。デプロイメント記述子エディタの使用方法の詳細については、2-22 ページの「デプロイメント記述子の編集」を参照してください。

   コンポーネントおよびアプリケーションの自動デプロイメントの詳細については、『BEA WebLogic Server 管理者ガイド』を参照してください。

7. プロダクション用に WebLogic Server に *.war ファイルをデプロイするか、エンタープライズ アプリケーションの一部としてデプロイするためにエンタープライズ アーカイブ（*.ear）ファイルに含めます。Administration Console を使用してアプリケーションおよびコンポーネントをデプロイします。

   コンポーネントおよびアプリケーションのデプロイメントの詳細については、『BEA WebLogic Server 管理者ガイド』を参照してください。

エンタープライズ JavaBean の作成 : 主な手順

エンタープライズ JavaBean を作成するには、特定の EJB（セッション、エンティティ、またはメッセージ駆動型）のクラスおよび EJB 固有のデプロイメント記述子を作成してから、それらをすべて `*.ear` ファイルにパッケージ化する必要があります。

エンタープライズ JavaBean 作成の主な手順は次のとおりです。

1. EJB の仕様に従って、各タイプの EJB（セッション、エンティティ、メッセージ駆動型）で必要なクラスの Java コードを記述します。たとえば、セッションおよびエンティティ EJB では、以下の 3 つのクラスが必要です。
   - EJB のホーム インタフェース
   - EJB のリモート インタフェース
   - EJB の実装クラス

   メッセージ駆動型 Bean では、実装クラスだけが必要となります。
2. インタフェースと実装の Java コードをクラス ファイルにコンパイルします。

   コンパイルの詳細については、2-18 ページの「コンパイルの準備」を参照してください。
3. EJB 固有のデプロイメント記述子を作成します。
   - `ejb-jar.xml` は、Sun Microsystems の標準 DTD を使用して、EJB のタイプとそのデプロイメント プロパティを記述します。
   - `weblogic.xml` ファイルは、WebLogic Server 固有のデプロイメント情報を追加します。
   - `weblogic-cmp-rdbms-jar.xml` は、コンテナ管理のエンティティ EJB をデータベース上のテーブルにマップします。このファイルは、JAR ファイルにパッケージ化される CMP Bean ごとに異なる名前を持たなければなりません。このファイル名は、`weblogic-ejb.jar` ファイル内の Bean のエントリに指定されます。

   手動で EJB のデプロイメント記述子を作成することも、WebLogic Server に含まれる Java ベース ユーティリティを使用して自動的に生成することもで

きます。これらのファイルの自動生成の詳細については、3-5 ページの「デプロイメント記述子の自動生成」を参照してください。

EJB 固有のデプロイメント記述子の要素、およびファイルの手動作成の方法については、『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。

4. クラス ファイルおよびデプロイメント記述子を Java アーカイブ ファイル `*.jar` にパッケージ化します。

   `*.jar` ファイル作成の最初の手順は、EJB のステージング ディレクトリを作成することです。コンパイルした Java クラスをステージング ディレクトリに置き、デプロイメント記述子を `META-INF` というサブディレクトリに置きます。次に、`weblogic.ejbc` EJB コンパイラを実行して、スタブおよびスケルトン クラスを生成してステージング ディレクトリに格納します。次に、ステージング ディレクトリで次のように `jar` コマンドを実行して、EJB アーカイブを作成します。

   ```
   jar cvf myEJB.jar *
   ```

   `*.jar` アーカイブ ファイル作成の詳細については、3-9 ページの「エンタープライズ JavaBeans のパッケージ化」を参照してください。

5. EJB アーカイブ ファイル `*.jar` をテスト目的で WebLogic Server に自動デプロイします。

   EJB をテストしているときに、EJB デプロイメント記述子を編集しなければならない場合があります。これは手動で編集することも、Administration Console のデプロイメント記述子エディタで編集することもできます。デプロイメント記述子エディタの使用方法の詳細については、2-22 ページの「デプロイメント記述子の編集」を参照してください。

   コンポーネントおよびアプリケーションの自動デプロイメントの詳細については、『BEA WebLogic Server 管理者ガイド』を参照してください。

6. プロダクション用に WebLogic Server に `*.jar` ファイルをデプロイするか、エンタープライズ アプリケーションの一部としてデプロイするためにエンタープライズ アーカイブ（`*.ear`）ファイルに含めます。Administration Console を使用してアプリケーションおよびコンポーネントをデプロイします。

   コンポーネントおよびアプリケーションのデプロイメントの詳細については、『BEA WebLogic Server 管理者ガイド』を参照してください。

WebLogic Server エンタープライズ アプリケーションの作成 : 主な手順

WebLogic Server エンタープライズ アプリケーションの作成では、Web コンポーネントと EJB コンポーネント、デプロイメント記述子、およびアーカイブ ファイルの作成が必要です。最終的にはエンタープライズ アプリケーション アーカイブ（.ear ファイル）になり、WebLogic Server にデプロイできます。

WebLogic Server エンタープライズ アプリケーション作成の主な手順は次のとおりです。

1. アプリケーションの Web コンポーネントと EJB コンポーネントを作成します。

   プログラマは、J2EE API を使用して、これらのコンポーネント用のサーブレットと EJB を作成します。Web デザイナは、HTML または XML、および JavaServer Pages を使用して Web ページを作成します。

   Web コンポーネントおよび EJB コンポーネント作成の詳細については、2-2 ページの「アプリケーションの作成 : 主な手順」および 2-4 ページの「エンタープライズ JavaBean の作成 : 主な手順」を参照してください。

   Web コンポーネントおよび EJB コンポーネントを構成する Java コード作成の詳細については、『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』、『WebLogic HTTP サーブレット プログラマーズ ガイド』、および『WebLogic JSP プログラマーズ ガイド』を参照してください。

2. Web コンポーネントと EJB コンポーネントのデプロイメント記述子を作成します。

   コンポーネントのデプロイメント記述子は、WebLogic Server でのアプリケーションのデプロイメントに必要な情報を提供する XML ドキュメントです。J2EE 仕様では、ejb-jar.xml や web.xml などのデプロイメント記述子の内容を定義しています。追加のデプロイメント記述子では、WebLogic Server でのコンポーネントのデプロイメントに必要な情報が提供され、J2EE 仕様の記述子を補足します。

   手動でこれらのデプロイメント記述子を作成することも、WebLogic Server に含まれる Java ベース ユーティリティを使用して自動的に生成することも

できます。これらのファイルの自動生成の詳細については、3-5 ページの「デプロイメント記述子の自動生成」を参照してください。

Web コンポーネントのデプロイメント記述子を手動で記述する際の詳細については、『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』、EJB コンポーネントのデプロイメント記述子を手動で記述する際の詳細については、『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。

3. Web コンポーネントおよび EJB コンポーネントをコンポーネント アーカイブ ファイルにパッケージ化します。

   コンポーネント アーカイブは、デプロイメント記述子など、すべてのコンポーネント ファイルが含まれる JAR ファイルです。Web コンポーネントを `*.war` ファイルに、EJB コンポーネントを EJB `*.jar` ファイルにパッケージ化します。

   コンポーネント アーカイブ作成の詳細については、3-7 ページの「Web アプリケーションのパッケージ化」および 3-9 ページの「エンタープライズ JavaBeans のパッケージ化」を参照してください。

4. エンタープライズ アプリケーションのデプロイメント記述子を作成します。

   エンタープライズ アプリケーションのデプロイメント記述子である `application.xml` では、アプリケーションに一緒にアセンブルされる個々のコンポーネントを示します。

   手動で `application.xml` のデプロイメント記述子を作成することも、WebLogic Server に含まれている Java ベース ユーティリティを使用して自動的に生成することもできます。このファイルの自動生成の詳細については、3-5 ページの「デプロイメント記述子の自動生成」を参照してください。

   `application.xml` ファイルの要素の詳細については、A-1 ページの「application.xml デプロイメント記述子の要素」を参照してください。

5. エンタープライズ アプリケーションをパッケージ化します。

   エンタープライズ アプリケーションのデプロイメント記述子とともに、Web コンポーネントおよび EJB コンポーネントのアーカイブをエンタープライズ アーカイブ（`*.ear`）ファイルにパッケージ化します。これは WebLogic Server にデプロイされるファイルです。WebLogic Server では `application.xml` デプロイメント記述子を使用して、EAR ファイルにパッケージ化された個々のコンポーネントを見つけてデプロイします。

エンタープライズ アプリケーションの *.ear アーカイブ ファイル作成の詳細については、3-13 ページの「エンタープライズ アプリケーションのパッケージ化」を参照してください。

6. *.ear エンタープライズ アプリケーションをテスト目的で WebLogic Server に自動デプロイします。

   エンタープライズ アプリケーションをテストしているときに、application.xml デプロイメント記述子を編集しなければならない場合があります。これは手動で編集することも、Administration Console のデプロイメント記述子エディタで編集することもできます。デプロイメント記述子エディタの使用方法の詳細については、2-22 ページの「デプロイメント記述子の編集」を参照してください。

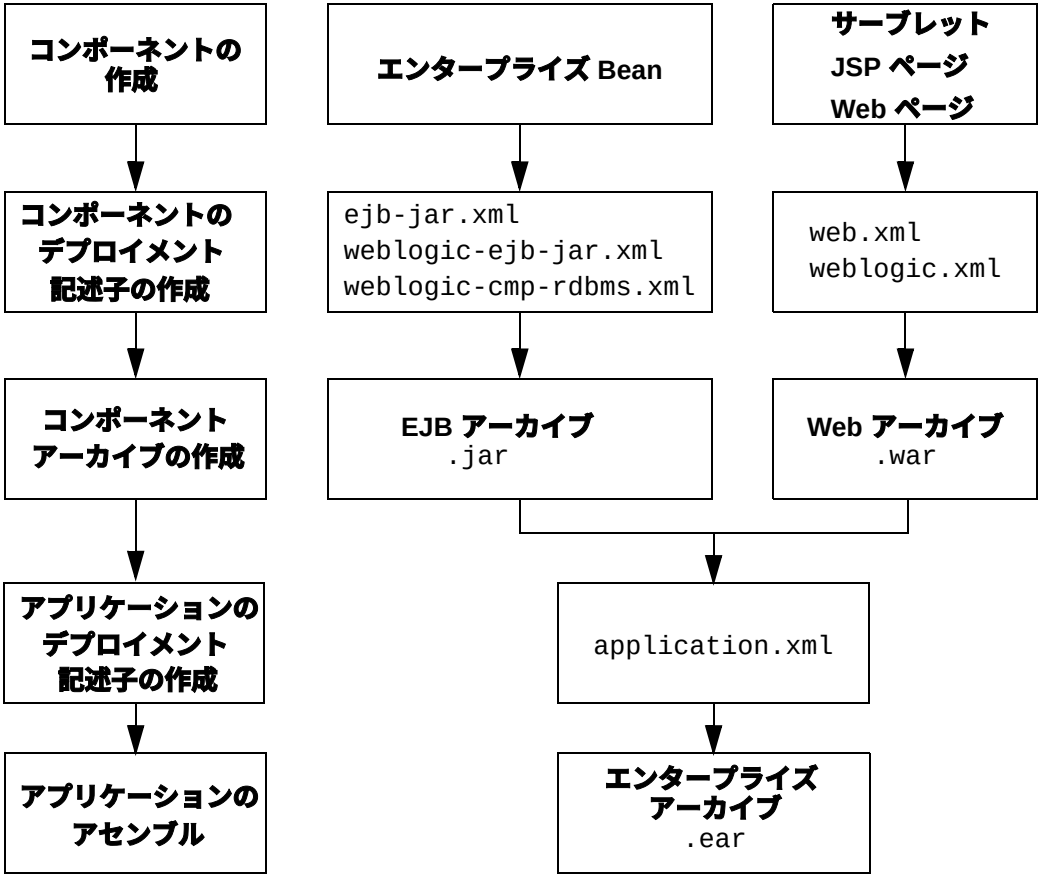
   コンポーネントおよびアプリケーションの自動デプロイメントの詳細については、『BEA WebLogic Server 管理者ガイド』を参照してください。

7. *.ear ファイルをプロダクション用に WebLogic Server にデプロイします。Administration Console を使用してアプリケーションおよびコンポーネントをデプロイします。

   コンポーネントおよびアプリケーションのデプロイメントの詳細については、『BEA WebLogic Server 管理者ガイド』を参照してください。

図 2-1 に、WebLogic Server エンタープライズ アプリケーションの開発とパッケージ化の手順を示します。

図 2-1 エンタープライズ アプリケーションの作成


コンポーネントの作成
エンタープライズ Bean
サーブレット
JSP ページ
Web ページ
コンポーネントのデプロイメント記述子の作成
ejb-jar.xml
weblogic-ejb-jar.xml
weblogic-cmp-rdbms.xml
web.xml
weblogic.xml
コンポーネントアーカイブの作成
EJB アーカイブ
.jar
Web アーカイブ
.war
アプリケーションのデプロイメント記述子の作成
application.xml
アプリケーションのアセンブル
エンタープライズアーカイブ
.ear

リソース アダプタの作成 : 主な手順

リソース アダプタを作成するには、リソース アダプタのクラスおよびコネクタ固有のデプロイメント記述子を作成してから、WebLogic Server にデプロイするために、それらをすべて .rar ファイルにパッケージ化する必要があります。

新しいリソース アダプタ（.rar）の作成

リソース アダプタ（.rar）を作成する主な手順を以下に説明します。

1. 「J2EE コネクタ仕様、バージョン 1.0、最終草案 2」（http://java.sun.com/j2ee/download.html#connectorspec）に準拠して、リソース アダプタ（ConnectionFactory や Connection など）に必要な各種クラスの Java コードを記述します。

   リソース アダプタを実装する場合は、以下のように ra.xml ファイルでクラスを指定しなければなりません。

   - <managedconnectionfactory-class>com.sun.connector.blackbox.LocalTxManagedConnectionFactory</managedconnectionfactory-class>
   - <connectionfactory-interface>javax.sql.DataSource</connectionfactory-interface>
   - <connectionfactory-impl-class>com.sun.connector.blackbox.JdbcDataSource</connectionfactory-impl-class>
   - <connection-interface>java.sql.Connection</connection-interface>
   - <connection-impl-class>com.sun.connector.blackbox.JdbcConnection</connection-impl-class>

2. インタフェースと実装の Java コードをクラス ファイルにコンパイルします。
3. Java クラスを Java アーカイブ ファイル（.jar）にパッケージ化します。

   *.jar ファイル作成の最初の手順は、コネクタのステージング ディレクトリを作成することです。.jar ファイルをステージング ディレクトリに置き、デプロイメント記述子を META-INF というサブディレクトリに置きます。

次に、ステージング ディレクトリで次のように jar コマンドを実行して、リソース アダプタ アーカイブを作成します。

```
jar cvf myRAR.rar *
```

リソース アダプタの .jar アーカイブ ファイルの作成については、3-11 ページの「リソース アダプタのパッケージ化」を参照してください。

4. リソース コネクタ固有のデプロイメント記述子を作成します。
   - ra.xml は、Sun Microsystems の標準 DTD を使用して、リソース アダプタ関連の属性タイプとそのデプロイメント プロパティを記述します。
   - weblogic-ra.xml ファイルは、WebLogic Server 固有のデプロイメント情報を追加します。

   コネクタ固有のデプロイメント記述子の作成の詳細については、『WebLogic J2EE コネクタ アーキテクチャ』を参照してください。
5. リソース アダプタ アーカイブ ファイル（.rar ファイル）を作成します。
   a. 最初に、空のステージング ディレクトリを作成します。
   b. リソース アダプタの Java クラスが入った .rar ファイルをステージング ディレクトリに格納します。
   c. デプロイメント記述子を META-INF というサブディレクトリに格納します。
   d. 次に、ステージング ディレクトリで次のように jar コマンドを実行して、リソース アダプタ アーカイブを作成します。

      ```
      jar cvf myRAR.rar *
      ```

      リソース アダプタ アーカイブ ファイルの作成の詳細については、3-11 ページの「リソース アダプタのパッケージ化」を参照してください。
6. リソース アダプタ アーカイブ ファイル .rar をテスト目的で WebLogic Server に自動デプロイします。

   リソース アダプタをテストしているときに、デプロイメント記述子を編集しなければならない場合があります。これは手動で編集することも、Administration Console のデプロイメント記述子エディタで編集することもできます。デプロイメント記述子エディタの使用方法の詳細については、2-22 ページの「デプロイメント記述子の編集」を参照してください。

コンポーネントおよびアプリケーションの自動デプロイメントの詳細については、『BEA WebLogic Server 管理者ガイド』を参照してください。

7. .rar リソース アダプタ ファイルを WebLogic Server にデプロイするか、またはエンタープライズ アプリケーションの一部としてデプロイするエンタープライズ アーカイブ（.ear）に含めます。

   コンポーネントおよびアプリケーションのデプロイメントの詳細については、『BEA WebLogic Server 管理者ガイド』を参照してください。

既存のリソース アダプタ（.rar）の変更

以下は、既存のリソース アダプタ（.rar）を、WebLogic Server にデプロイするために変更する方法の例です。この場合、デプロイメント記述子 weblogic-ra.xml を追加し、再パッケージ化する必要があります。

1. リソース アダプタをステージングするための一時ディレクトリを作成します。

   ```
   mkdir c:\stagedir
   ```

2. 一時ディレクトリにデプロイするリソース アダプタをコピーします。

   ```
   cp blackbox-notx.rar c:\stagedir
   ```

3. リソース アダプタ アーカイブの中身を展開します。

   ```
   cd c:\stagedir
   jar xf blackbox-notx.rar
   ```

   ステージング ディレクトリには、以下のものが格納されます。

   - リソース アダプタを実装する Java クラスが入った jar ファイル
   - Manifest.mf および ra.xml ファイルが入った META-INF ディレクトリ

以下のコマンドを実行してこれらのファイルを確認します。

```
c:\stagedir> ls
       blackbox-notx.jar
       META-INF
c:\stagedir> ls META-INF
       Manifest.mf
       ra.xml
```

4. weblogic-ra.xml ファイルを作成します。このファイルは、リソース アダプタ用の WebLogic 固有のデプロイメント記述子です。このファイルには、接続ファクトリ、接続プール、およびセキュリティ マッピングのパラメータを指定します。

   weblogic-ra.xml DTD の詳細については、『WebLogic J2EE コネクタ アーキテクチャ』を参照してください。

5. weblogic-ra.xml ファイルを一時ディレクトリの META-INF サブディレクトリにコピーします。META-INF ディレクトリは、.rar ファイルを展開した一時ディレクトリ、またはリソース アダプタを展開ディレクトリ形式で格納しているディレクトリ内にあります。次のコマンドを使用します。

   ```
   cp weblogic-ra.xml c:\stagedir\META-INF
   c:\stagedir> ls META-INF
          Manifest.mf
          ra.xml
          weblogic-ra.xml
   ```

6. リソース アダプタ アーカイブを作成します。

   ```
   jar cvf blackbox-notx.jar -C c:\stagedir
   ```

7. WebLogic Server にリソース アダプタをデプロイします。リソース アダプタの WebLogic Server へのデプロイメントについては、『WebLogic J2EE コネクタ アーキテクチャ』を参照してください。

開発環境の構築

WebLogic Server アプリケーションを開発するには、ソフトウェア ツールをアセンブルし、コードを作成、コンパイル、デプロイ、テスト、およびデバッグするための環境を設定しておく必要があります。この節では、ツールキットを構築し、開発用コンピュータにコンパイラ関連の環境を設定する方法について説明します。

ソフトウェア ツール

この節では、WebLogic Server アプリケーションの開発に必要なソフトウェア、および開発とデバッグに使用するオプション ツールについて説明します。

ソース コード エディタまたは IDE

Java ソース ファイル、コンフィグレーション ファイル、HTML/XML ページ、および JavaServer Pages の編集には、テキスト エディタが必要です。Windows と UNIX の行末の違いを適切に処理するエディタが望ましいですが、それ以外に特別な要件は何もありません。

WebGain VisualCafé などの Java 統合開発環境（IDE）には、通常 Java のカスタム サポートと共にプログラマ用エディタが付属しています。また IDE は、WebLogic Server でのサーブレットとエンタープライズ JavaBean の作成とデプロイをサポートしている場合もあります。その場合、アプリケーションの開発、テスト、およびデバッグが非常に簡単になります。

HTML/XML ページと JavaServer Pages は、通常のテキスト エディタか、または DreamWeaver などの Web ページ エディタで編集できます。

XML エディタ

EJB および Web アプリケーション デプロイメント記述子、config.xml ファイルなど、WebLogic Server で使用される XML ファイルを編集するには、XML エディタを使用します。WebLogic Server には、次の 2 つの XML エディタが付属しています。

- Administration Console のデプロイメント記述子エディタ
- Java ベースのスタンドアロン エディタの BEA XML エディタ

これらの XML エディタの使用方法の詳細については、2-22 ページの「デプロイメント記述子の編集」を参照してください。

Java コンパイラ

Java コンパイラは、ポータブル バイト コードで構成される Java クラス ファイルを Java ソースから生成します。Java コンパイラは、アプリケーション用に記述した Java コードと、WebLogic RMI、EJB、および JSP コンパイラによって生成されたコードをコンパイルします。

Sun Microsystems の Java 2, Standard Edition には、`javac` という Java コンパイラが付属しています。WebLogic Server をインストールしたときに付属の JRE をインストールした場合、`javac` コンパイラがインストールされます。

これ以外にも、さまざまなプラットフォームに対応した Java コンパイラを使用できます。標準 Java `.class` ファイルを生成する Java コンパイラであれば、どのようなコンパイラでも WebLogic Server アプリケーションの開発に使用できます。ほとんどの Java コンパイラは `javac` より何倍も高速であり、また IDE と緊密に統合されている Java コンパイラもあります。

コンパイラによって生成された最適化済みコードが、すべての Java 仮想マシン（JVM）で正常に動作しない場合もあります。問題をデバッグする場合は、最適化を無効にするか、異なる最適化セットを選択するか、または `javac` でコンパイルしてみて、使用している Java コンパイラが原因かどうかを調べてください。また、デプロイする前に、常に対象となる各 JVM でコードをテストしてください。

開発用 WebLogic Server

テストされていないコードを、製品アプリケーションのサーバとなる WebLogic Server にデプロイしないでください。つまり、開発用 WebLogic Server が環境に必要です。開発用 WebLogic Server は、編集とコンパイルを行うコンピュータで実行することも、ネットワークのどこかにデプロイされているコンピュータで実行することもできます。

Java はプラットフォームに依存しないので、任意のプラットフォームでコードの編集とコンパイルを行い、別のプラットフォームで稼働する開発用 WebLogic Server でアプリケーションをテストできます。たとえば、WebLogic Server アプリケーションを Windows または Linux が動作している PC で開発する場合、そのアプリケーションが最終的にどこにデプロイされるかを考慮する必要はありません。

開発用コンピュータで開発用 WebLogic Server を実行しない場合でも、WebLogic Server 配布キットにアクセスできなければプログラムをコンパイルできません。WebLogic または J2EE API を使用してコードをコンパイルするには、Java コンパイラが配布ディレクトリ内の `weblogic.jar` ファイルとその他の JAR ファイルにアクセスする必要があります。開発用コンピュータに WebLogic Server をインストールすると、これらのファイルがローカルに使用できます。

データベース システムと JDBC ドライバ

データベース システムは、ほぼすべての WebLogic Server アプリケーションで必要となります。標準 JDBC ドライバを介してアクセスできる任意の DBMS を使用できますが、WebLogic JMS などのサービスでは、Oracle、Sybase、Informix、Microsoft SQL Server、IBM DB2、または Cloudscape をサポートする JDBC ドライバが必要です。サポートされるデータベース システムと JDBC ドライバについては、プラットフォーム サポートの Web ページを参照してください。

JDBC 接続プールは非常に高いパフォーマンスを提供するので、2 層 JDBC ドライバを直接使用するアプリケーションの作成を検討する必要はほとんどありません。接続プールは、使用の準備ができたデータベース接続のコレクションです。接続プールはその起動時に、指定された数の同じ物理データベース接続を作成します。起動時に接続を確立することにより、接続プールは各アプリケーション用にデータベース接続を作成するオーバーヘッドを軽減します。BEA では、クライアント側とサーバ側の両方のアプリケーションが、JDBC ツリー上のデータソースを通じて接続プールから接続を取得する構成を推奨しています。接続を使った処理が終了したら、アプリケーションは接続を接続プールに返却します。

マルチプールは基本の接続プールを多重化したものです。マルチプールはアプリケーションにとっては基本のプールと同じように機能しますが、マルチプールを使用すると、接続プールのプールを確立できるようになります。そのとき、接続属性は接続プールごとに異なります。1 つの接続プール内の接続はすべて等質ですが、あるプールに予測される障害がマルチプール内の別のプールを巻き添えに

したりしないように、マルチプール内の接続プールごとに接続の性質はある程度異なっているのが一般的です。これらのプールは通常、同じデータベースの異なったインスタンスを接続先とします。

マルチプールが効果を発揮するのは、アプリケーションの接続を同じように処理する複数の異なったデータベース インスタンスが存在し、アプリケーションの作業が複数のデータベース間に分散されるときにアプリケーション システムがデータベースの同期を正しく処理する場合に限られます。まれに、同じデータベース インスタンスを指す複数のプールを異なったユーザとして用意すると役に立つ場合があります。これは、DBA が他のユーザを有効にしたままある特定のユーザを無効にする場合などに役立ちます。

デフォルトでは、クラスタ化されるマルチプールは高可用性（DBMS のフェイルオーバ）を実現します。必要に応じて、ロードバランシングにも対応するようにマルチプールをコンフィグレーションすることができます。

Web ブラウザ

ほとんどの J2EE アプリケーションは、Web ブラウザ クライアントによって実行されるように設計されています。WebLogic Server は HTTP 1.1 仕様をサポートしており、Netscape Communicator および Microsoft Internet Explorer ブラウザの現行バージョンでテストされています。

作成するアプリケーションの条件を書き出す場合、どの Web ブラウザ バージョンをサポートするかに留意してください。テスト プランは、サポートするバージョンごとに作成します。バージョン番号とブラウザ コンフィグレーションは明確に指定します。作成するアプリケーションは SSL をサポートしますか？ブラウザの代替セキュリティ設定をテストして、サポートしているセキュリティをユーザに知らせることができるようにします。

アプリケーションがアプレットを使用する場合、さまざまなブラウザに埋め込まれている JVM の違いのために、サポートするブラウザのコンフィグレーションをテストすることが特に重要です。解決策の 1 つは、Sun から Java Plug-in をインストールするようユーザに指示して、すべてのユーザが同じ Java ランタイム バージョンを持つようにすることです。

サードパーティ ソフトウェア

WebGain Studio、WebGain StructureBuilder、および BEA WebLogic Integration Kit for VisualAge for Java などのサードパーティ ソフトウェア製品を使用して、WebLogic Server 開発環境を強化できます。

詳細については、「BEA WebLogic Developer Tools Resources」Web ページを参照してください。BEA アプリケーション サーバをサポートする製品の開発者ツール情報を確認できます。

このツールの一部をダウンロードするには、「BEA WebLogic Server Downloads」Web ページを参照してください。

注意： ソフトウェア ベンダに問い合わせて、使用しているプラットフォームと WebLogic Server バージョンにソフトウェアが対応しているかどうかを確認してください。

コンパイルの準備

WebLogic Server 用の Java プログラムのコンパイルは、他の Java プログラムのコンパイルと同じです。適切にコンパイルを行うには、以下の準備が必要です。

- 検索パスに Java コンパイラを指定する
- クラスパスを設定して、Java コンパイラがすべての依存クラスを検索できるようにする
- コンパイルされたクラスの出力ディレクトリを指定する

環境を設定する方法の 1 つは、コマンド ファイルまたはシェル スクリプトを作成して環境変数を設定し、それをコンパイラに渡すことです。この方法の例として、`config\examples` ディレクトリに `setExamplesEnv.cmd`（Windows）ファイルと `setExamplesEnv.sh`（UNIX）ファイルがあります。

検索パスへの Java ツールの指定

オペレーティング システムがコンパイラとその他の JDK ツールを検索できるようにするには、そのコンパイラをコマンド シェルの PATH 環境変数に追加します。JDK を使用している場合、ツールは JDK ディレクトリの bin サブディレクトリに置かれています。javac 以外のコンパイラ（WebGain VisualCafé の sj コンパイラなど）を使用するには、そのコンパイラが格納されているディレクトリを検索パスに追加します。

たとえば、JDK が UNIX ファイル システムの /usr/local/java/java130 にインストールされている場合、Bourne シェルまたはシェル スクリプトで次のようなコマンドを使用して javac を検索パスに追加します。

```
PATH=/usr/local/java/java130/bin:$PATH; export PATH
```

WebGain sj コンパイラを Windows NT または Windows 2000 のパスに追加するには、コマンド シェルまたはコマンド ファイルで次のようなコマンドを使用します。

```
PATH=c:\VisualCafe\bin;%PATH%
```

IDE を使用している場合は、その IDE のドキュメントを参照して、検索パスの設定方法を調べてください。

コンパイル用のクラスパスの設定

ほとんどの WebLogic サービスは J2EE 仕様に基づいており、標準 J2EE パッケージを通じてアクセスします。WebLogic サービスを使用するプログラムのコンパイルに必要な Sun、WebLogic、およびその他の Java クラスは、インストールした WebLogic Server の lib ディレクトリの weblogic.jar ファイルにパッケージ化されます。weblogic.jar ファイル以外にも、以下のものをコンパイラのクラスパスに組み込みます。

- J2EE 1.2 の機能に限定された（つまり、J2EE 1.3 の機能を備えていない）WebLogic Server 6.1 のバージョンを使用している場合は、必ず j2ee12.jar ファイルを CLASSPATH に入れてから、weblogic.jar ファイルを指定する必要があります。j2ee12.jar ファイルは、CLASSPATH の先頭に入れることをお勧めします。

WebLogic Server インスタンスで実装する J2EE のバージョン（1.2 または 1.3）の詳細については、1-3 ページの「J2EE プラットフォーム」を参照してください。

- JDK ディレクトリ内の `lib\tools.jar` ファイル、または使用する Java 開発キットに必要なその他の標準 Java クラス。
- WebLogic Server サービス パック（存在する場合）と共に配布される `weblogic_sp.jar` ファイル。

  この `jar` ファイルは、クラスパス内で `weblogic.jar` より前に指定する必要があります。これにより、サービス パック クラスはそれらに取って代わられる `weblogic.jar` 内のクラスより先に検索されるようになります。
- サードパーティ Java ツールまたはプログラムがインポートするサービスのクラス。
- コンパイルするプログラムによって参照されるその他のアプリケーション クラス。

  クラスパスには、コンパイラによってコンパイルされたクラスの書き出し先となるディレクトリを指定します。これにより、コンパイラはアプリケーション内で依存し合うクラスをすべて検索できるようになります。次の節では、この出力ディレクトリについて詳しく説明します。

コンパイルされたクラスの出力ディレクトリの設定

コンパイルされたクラスの出力ディレクトリを指定しない場合、Java コンパイラは Java ソースと同じディレクトリにクラス ファイルを書き出します。出力ディレクトリを指定した場合、コンパイラはパッケージ名と同じディレクトリ構造にクラス ファイルを格納します。これにより、Java クラスはアプリケーションのパッケージ化に使用するステージング ディレクトリ内の適切な場所にコンパイルされます。出力先ディレクトリを指定しなかった場合、ファイルを移動してからでなければ、パッケージ化されたコンポーネントを含む `jar` ファイルを作成できません。

J2EE アプリケーションは、1 つのアプリケーションにアセンブルされ、1 つまたは複数の WebLogic Server または WebLogic クラスタにデプロイされるモジュールから構成されています。各モジュールは、独自のステージング ディレクトリを持つ必要があります。これにより、他のモジュールとは別個にコンパイル、

パッケージ化、およびデプロイできるようになります。たとえば、EJB、Web コンポーネント、およびその他のサーバサイド クラスをそれぞれ独立したモジュールにパッケージ化できます。

コンパイラの出力ディレクトリの設定例については、WebLogic Server 配布キットの config\examples ディレクトリにある setExamplesEnv スクリプトを参照してください。このスクリプトは、以下の変数を設定します。

CLIENT_CLASSES
: コンパイルされたクライアント クラスが書き出されるディレクトリ。これらのクラスは通常、WebLogic Server に接続するスタンドアロンの Java プログラムです。これらのクラスは WebLogic Server クラスパスに配置する必要はありません。

SERVER_CLASSES
: サーバサイド クラスが書き出されるディレクトリ。これらのクラスにはスタートアップ クラスとその他の Java クラスがあり、サーバの起動時に WebLogic Server クラスパスに配置されている必要があります。このディレクトリのクラスは WebLogic Server を再起動しないと再デプロイされないため、通常は、アプリケーション クラスをこのディレクトリにコンパイルしないでください。

EX_WEBAPP_CLASSES
: Web アプリケーションによって使用されるクラスが書き出されるディレクトリ。

APPLICATIONS
: サンプル ドメイン用の applications ディレクトリ。他の変数とは異なり、この変数は Java コンパイラの対象の指定には使用しません。この変数は、ファイルをソース ディレクトリから applications ディレクトリへ移動するコピー コマンドで、applications ディレクトリへの便利な参照として使用します。たとえば、ソース ツリーの中に .html、.jsp、および画像ファイルがある場合、コピー コマンドでこの変数を使用して、それらのファイルを開発用サーバにインストールできます。

これらの環境変数は、次のようなコマンドで（Windows の場合）コンパイラに渡されます。

```
javac -d %SERVER_CLASSES% *.java
```

IDE を使用しない場合、メイク ファイル、シェル スクリプト、またはコマンド ファイルを記述して、コンポーネントとアプリケーションをコンパイルおよびパッケージ化することを検討してください。構築スクリプトに変数を設定して、1 つのコマンドでコンポーネントを再構築できるようにします。

デプロイメント記述子の編集

以下のツールのいずれか 1 つを使用して、WebLogic アプリケーションおよびコンポーネントのデプロイメント記述子を編集できます。

- BEA XML エディタ
- Administration Console 内のデプロイメント記述子エディタ

いずれかのエディタを使用して、以下のデプロイメント記述子に対して、既存の要素の更新、新しい要素の追加、既存の要素の削除を行うことができます。

- `web.xml`
- `weblogic.xml`
- `ejb-jar.xml`
- `weblogic-ejb-jar.xml`
- `weblogic-cmp-rdbms-jar.xml`
- `ra.xml`
- `weblogic-ra.xml`
- `application.xml`

BEA XML エディタの使い方

XML ファイルを編集するには、BEA XML エディタを使います。これは、完全に Java ベースの XML スタンドアロン エディタで、XML ファイルの作成と編集の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 ファイルの内容を、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。

BEA XML エディタの使用方法の詳細については、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。

Administration Console のデプロイメント記述子エディタの使用方法

Administration Console のデプロイメント記述子エディタは、メインの Administration Console に非常に似ています。左側のペインでは、デプロイメント記述子ファイルの要素がツリー形式で表示され、右側のペインには、特定の要素を更新するためのフォームがあります。

エディタを使用する場合、インメモリ デプロイメント記述子のみを更新するか、またはインメモリおよびディスク ファイルの両方を更新することができます。特定の要素を更新してから [適用] ボタンをクリックするか、[作成] ボタンをクリックして新しい要素を作成すると、WebLogic Server のメモリ内のデプロイメント記述子のみが更新されます。変更はまだディスクには書き込まれていません。変更をディスクに書き込むためには、明示的に [永続化] ボタンをクリックする必要があります。変更を明示的にディスクに永続化しない場合、WebLogic Server を終了して再起動すると、変更は失われます。

EJB デプロイメント記述子の編集

この節では、Administration Console のデプロイメント記述子エディタを使用して以下の EJB デプロイメント記述子を編集する手順を説明します。

- `ejb-jar.xml`

- `weblogic-ejb-jar.xml`
- `weblogic-cmp-rdbms-jar.xml`

EJB 固有のデプロイメント記述子の要素の詳細については、『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。

EJB デプロイメント記述子を編集するには、次の手順に従います。

1. ブラウザで次の URL を指定して、Administration Console を起動します。

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

   `host` は、WebLogic Server が稼働するコンピュータの名前、*`port`* は WebLogic Server がリスンするポートの番号です。

2. 左ペインの [デプロイメント] ノードをクリックして展開します。
3. [デプロイメント] ノードの [EJB] ノードをクリックして展開します。
4. 編集対象のデプロイメント記述子がある EJB の名前を右クリックし、ドロップダウン メニューから [EJB 記述子の編集] を選択します。Administration Console ウィンドウが新しいブラウザに表示されます。

   左側のペインでは、3 つの EJB デプロイメント記述子のすべての要素がツリー形式で表示され、右側のペインには、`ejb-jar.xml` ファイルの説明要素のためのフォームがあります。

5. EJB デプロイメント記述子の要素を編集、削除、または追加するには、以下のリストで説明されているように、左側のペインで編集対象のデプロイメント記述子に対応するノードをクリックして展開します。
   - [ejb-jar] ノードには、`ejb-jar.xml` デプロイメント記述子の要素があります。
   - [weblogic-ejb-jar] ノードには、`weblogic-ejb-jar.xml` デプロイメント記述子の要素があります。
   - [CMP] ノードには、`weblogic-cmp-rdbms-jar.xml` デプロイメント記述子の要素があります。
6. いずれかの EJB デプロイメント記述子の既存の要素を編集するには、次の手順に従います。

   a. 左側のペインでツリーをナビゲートし、編集対象の要素が見つかるまで親要素をクリックします。

b. 要素をクリックします。右側のペインに、属性または下位要素のどちらかをリストするフォームが表示されます。

c. 右側のペインのフォームで、テキストを編集します。

d. [適用] をクリックします。

7. いずれかの EJB デプロイメント記述子の新しい要素を追加するには、次の手順に従います。

a. 左側のペインでツリーをナビゲートし、作成対象の要素の名前が見つかるまで親要素をクリックします。

b. 要素を右クリックして、ドロップダウン メニューから [新しい（要素名）のコンフィグレーション] を選択します。

c. 右側のペインに表示されるフォームで、要素情報を入力します。

d. [作成] をクリックします。

8. いずれかの EJB デプロイメント記述子の既存の要素を削除するには、次の手順に従います。

a. 左側のペインでツリーをナビゲートし、削除対象の要素の名前が見つかるまで親要素をクリックします。

b. 要素を右クリックして、ドロップダウン メニューから [（要素名）の削除] を選択します。

c. [はい] をクリックすると、要素の削除が確定されます。

9. EJB デプロイメント記述子への変更がすべて完了したら、左側のペインでツリーのルート要素をクリックします。ルート要素は、EJB の `*.jar` アーカイブ ファイルの名前または EJB の表示名です。

10. EJB デプロイメント記述子のエントリが有効かどうかを確認する場合は、[検証] をクリックします。

11. [永続化] をクリックして、デプロイメント記述子ファイルの編集を、WebLogic Server のメモリだけでなくディスクに書き込みます。

Web アプリケーションのデプロイメント記述子の編集

この節では、Administration Console のデプロイメント記述子エディタを使用して以下の Web アプリケーション デプロイメント記述子を編集する手順を説明します。

- `web.xml`
- `weblogic.xml`

Web アプリケーション 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。

Web アプリケーション デプロイメント記述子を編集するには、次の手順に従います。

1. ブラウザで次の URL を指定して、Administration Console を起動します。

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

   `host` は、WebLogic Server が稼働するコンピュータの名前、*`port`* は WebLogic Server がリスンするポートの番号です。

2. 左ペインの [デプロイメント] ノードをクリックして展開します。
3. [デプロイメント] ノードの [Web アプリケーション] ノードをクリックして展開します。
4. 編集対象のデプロイメント記述子がある Web アプリケーションの名前を右クリックし、ドロップダウン メニューから [Web アプリケーション記述子の編集] を選択します。Administration Console ウィンドウが新しいブラウザに表示されます。

   左側のペインでは、2 つの Web アプリケーション デプロイメント記述子のすべての要素がツリー形式で表示され、右側のペインには、`web.xml` ファイルの説明要素のためのフォームがあります。

5. Web アプリケーション デプロイメント記述子の要素を編集、削除、または追加するには、以下のリストで説明されているように、左側のペインで編集対象のデプロイメント記述子に対応するノードをクリックして展開します。
   - [Web App Descriptor] ノードには、`web.xml` デプロイメント記述子の要素があります。
   - [WebApp Ext] ノードには、`weblogic.xml` デプロイメント記述子の要素があります。

6. いずれかの Web アプリケーション デプロイメント記述子の既存の要素を編集するには、次の手順に従います。
   a. 左側のペインでツリーをナビゲートし、編集対象の要素が見つかるまで親要素をクリックします。
   b. 要素をクリックします。右側のペインに、属性または下位要素のどちらかをリストするフォームが表示されます。
   c. 右側のペインのフォームで、テキストを編集します。
   d. [適用] をクリックします。
7. いずれかの Web アプリケーション デプロイメント記述子の新しい要素を追加するには、次の手順に従います。
   a. 左側のペインでツリーをナビゲートし、作成対象の要素の名前が見つかるまで親要素をクリックします。
   b. 要素を右クリックして、ドロップダウン メニューから [新しい（要素名）のコンフィグレーション] を選択します。
   c. 右側のペインに表示されるフォームで、要素情報を入力します。
   d. [作成] をクリックします。
8. いずれかの Web アプリケーション デプロイメント記述子の既存の要素を削除するには、次の手順に従います。
   a. 左側のペインでツリーをナビゲートし、削除対象の要素の名前が見つかるまで親要素をクリックします。
   b. 要素を右クリックして、ドロップダウン メニューから [（要素名）の削除] を選択します。
   c. [はい] をクリックすると、要素の削除が確定されます。
9. Web アプリケーション デプロイメント記述子への変更がすべて完了したら、左側のペインでツリーのルート要素をクリックします。ルート要素は、Web アプリケーションの `*.war` アーカイブ ファイルの名前または Web アプリケーションの表示名です。
10. Web アプリケーション デプロイメント記述子のエントリが有効かどうかを確認する場合は、[検証] をクリックします。
11. [永続化] をクリックして、デプロイメント記述子ファイルの編集を、WebLogic Server のメモリだけでなくディスクに書き込みます。

リソース アダプタのデプロイメント記述子の編集

この節では、Administration Console のデプロイメント記述子エディタを使用して以下のリソース アダプタのデプロイメント記述子を編集する手順を説明します。

- `ra.xml`
- `weblogic-ra.xml`

リソース アダプタのデプロイメント記述子の要素の詳細については、『WebLogic J2EE コネクタ アーキテクチャ』を参照してください。

リソース アダプタのデプロイメント記述子を編集するには、次の手順に従います。

1. ブラウザで次の URL を指定して、Administration Console を起動します。

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

   `host` は、WebLogic Server が稼働するコンピュータの名前、*`port`* は WebLogic Server がリスンするポートの番号です。

2. 左ペインの [デプロイメント] ノードをクリックして展開します。
3. [デプロイメント] ノードの [コネクタ] ノードをクリックして展開します。
4. 編集対象のデプロイメント記述子があるリソース アダプタの名前を右クリックし、ドロップダウン メニューから [コネクタ記述子の編集] を選択します。Administration Console ウィンドウが新しいブラウザに表示されます。

   左側のペインでは、2 つのリソース アダプタのデプロイメント記述子のすべての要素がツリー形式で表示され、右側のペインには、`ra.xml` ファイルの説明要素のためのフォームがあります。

5. リソース アダプタのデプロイメント記述子の要素を編集、削除、または追加するには、以下のリストで説明されているように、左側のペインで編集対象のデプロイメント記述子に対応するノードをクリックして展開します。
   - [RA] ノードには、`ra.xml` デプロイメント記述子の要素があります。
   - [WebLogic RA] ノードには、`weblogic-ra.xml` デプロイメント記述子の要素があります。
6. いずれかのリソース アダプタ デプロイメント記述子の既存の要素を編集するには、次の手順に従います。

a. 左側のペインでツリーをナビゲートし、編集対象の要素が見つかるまで親要素をクリックします。

b. 要素をクリックします。右側のペインに、属性または下位要素のどちらかをリストするフォームが表示されます。

c. 右側のペインのフォームで、テキストを編集します。

d. [適用] をクリックします。

7. いずれかのリソース アダプタ デプロイメント記述子の新しい要素を追加するには、次の手順に従います。

a. 左側のペインでツリーをナビゲートし、作成対象の要素の名前が見つかるまで親要素をクリックします。

b. 要素を右クリックして、ドロップダウン メニューから [新しい（要素名）のコンフィグレーション] を選択します。

c. 右側のペインに表示されるフォームで、要素情報を入力します。

d. [作成] をクリックします。

8. いずれかのリソース アダプタ デプロイメント記述子の既存の要素を削除するには、次の手順に従います。

a. 左側のペインでツリーをナビゲートし、削除対象の要素の名前が見つかるまで親要素をクリックします。

b. 要素を右クリックして、ドロップダウン メニューから [（要素名）の削除] を選択します。

c. [はい] をクリックすると、要素の削除が確定されます。

9. リソース アダプタ デプロイメント記述子への変更がすべて完了したら、左側のペインでツリーのルート要素をクリックします。ルート要素は、リソース アダプタの `*.rar` アーカイブ ファイルの名前またはリソース アダプタの表示名です。

10. リソース アダプタ デプロイメント記述子のエントリが有効かどうかを確認する場合は、[検証] をクリックします。

11. [永続化] をクリックして、デプロイメント記述子ファイルの編集を、WebLogic Server のメモリだけでなくディスクに書き込みます。

エンタープライズ アプリケーションのデプロイメント記述子の編集

この節では、Administration Console のデプロイメント記述子エディタを使用してエンタープライズ アプリケーション デプロイメント記述子（application.xml）を編集する手順を説明します。

application.xml ファイルの要素の詳細については、A-1 ページの「application.xml デプロイメント記述子の要素」を参照してください。

注意： 以下の手順は、application.xml ファイルの編集方法のみを説明するものです。エンタープライズ アプリケーションを構成するコンポーネントのデプロイメント記述子を編集する場合は、2-23 ページの「EJB デプロイメント記述子の編集」、2-26 ページの「Web アプリケーションのデプロイメント記述子の編集」、または 2-28 ページの「リソース アダプタのデプロイメント記述子の編集」を参照してください。

エンタープライズ アプリケーション デプロイメント記述子を編集するには、次の手順に従います。

1. ブラウザで次の URL を指定して、Administration Console を起動します。

   http://*host*:*port*/console

   host は、WebLogic Server が稼働するコンピュータの名前、*port* は WebLogic Server がリスンするポートの番号です。

2. 左ペインの [デプロイメント] ノードをクリックして展開します。

3. [デプロイメント] ノードの [アプリケーション] ノードをクリックして展開します。

4. 編集対象のデプロイメント記述子があるエンタープライズ アプリケーションの名前を右クリックし、ドロップダウン メニューから [アプリケーション記述子の編集] を選択します。Administration Console ウィンドウが新しいブラウザに表示されます。

   左側のペインでは、application.xml ファイルのすべての要素がツリー構造で表示され、右側のペインには、表示名やアイコン ファイル名などの説明要素のためのフォームがあります。

5. application.xml デプロイメント記述子の既存の要素を編集するには、次の手順に従います。

a. 左側のペインでツリーをナビゲートし、編集対象の要素が見つかるまで親要素をクリックします。

b. 要素をクリックします。右側のペインに、属性または下位要素のどちらかをリストするフォームが表示されます。

c. 右側のペインのフォームで、テキストを編集します。

d. [適用] をクリックします。

6. `application.xml` デプロイメント記述子に新しい要素を追加するには、次の手順に従います。

a. 左側のペインでツリーをナビゲートし、作成対象の要素の名前が見つかるまで親要素をクリックします。

b. 要素を右クリックして、ドロップダウン メニューから [新しい（要素名）のコンフィグレーション] を選択します。

c. 右側のペインに表示されるフォームで、要素情報を入力します。

d. [作成] をクリックします。

7. `application.xml` デプロイメント記述子の既存の要素を削除するには、次の手順に従います。

a. 左側のペインでツリーをナビゲートし、削除対象の要素の名前が見つかるまで親要素をクリックします。

b. 要素を右クリックして、ドロップダウン メニューから [（要素名）の削除] を選択します。

c. [はい] をクリックすると、要素の削除が確定されます。

8. `application.xml` デプロイメント記述子への変更がすべて完了したら、左側のペインでツリーのルート要素をクリックします。ルート要素は、エンタープライズ アプリケーションの `*.ear` アーカイブ ファイルの名前またはエンタープライズ アプリケーションの表示名です。

9. `application.xml` デプロイメント記述子のエントリが有効かどうかを確認する場合は、[検証] をクリックします。

10. [永続化] をクリックして、デプロイメント記述子ファイルの編集を、WebLogic Server のメモリだけでなくディスクに書き込みます。

3 WebLogic Server J2EE アプリケーションのパッケージ化

以降の節では、WebLogic Server J2EE アプリケーションをパッケージ化およびデプロイする方法について説明します。

- パッケージ化の概要
- Web アプリケーションのパッケージ化
- エンタープライズ JavaBeans のパッケージ化
- リソース アダプタのパッケージ化
- エンタープライズ アプリケーションのパッケージ化
- **クライアント アプリケーションのパッケージ化**
- Apache ant を使った J2EE アプリケーションのパッケージ化

パッケージ化の概要

WebLogic Server J2EE アプリケーションは、J2EE 仕様で定義されている標準の方法でパッケージ化されます。J2EE では、コンポーネントの動作とパッケージ化が汎用的で移植性の高い方法で定義されています。このため、実行時コンフィグレーションはコンポーネントを実際にアプリケーション サーバにデプロイするときに行います。

J2EE には、Web アプリケーション、EJB モジュール、エンタープライズ アプリケーション、クライアント アプリケーション、およびリソース アダプタ用のデプロイメント仕様が含まれています。J2EE では、どのようにアプリケーションをターゲット サーバにデプロイするかは指定されておらず、標準のコンポーネントまたはアプリケーションをパッケージ化する方法だけが指定されています。

コンポーネントのタイプごとに、J2EE には必要なファイルとそれらのディレクトリ構造上の格納場所が定義されています。コンポーネントとアプリケーションは、多くの場合、EJB とサーブレットの Java クラス、リソース アダプタ、Web ページとサポート ファイル、XML 形式のデプロイメント記述子、およびその他のコンポーネントが格納された JAR ファイルで構成されています。

WebLogic Server にデプロイできる状態のアプリケーションには、WebLogic 固有のデプロイメント記述子が含まれています。また、WebLogic EJB、RMI、または JSP コンパイラで生成されたコンテナ クラス（オプション）が含まれることもあります。

JAR ファイル

Java の `jar` ユーティリティで作成される Java ARchive（JAR）ファイルには、1 つのディレクトリ内のファイルが、ディレクトリ構造を維持したまま統合されます。Java クラスローダは、クラスパス内のディレクトリを検索するのと同じように、JAR ファイル内の Java クラス ファイル（および他のファイル タイプ）を検索できます。クラスローダはディレクトリまたは JAR ファイルを検索できるので、「展開された」ディレクトリまたは JAR ファイルの形式で、J2EE コンポーネントを WebLogic Server にデプロイできます。

JAR ファイルは、コンポーネントとアプリケーションをパッケージ化して配布するのに役立ちます。簡単にコピーでき、展開されたディレクトリよりも処理するファイル数が少なく、ファイル圧縮によってディスク スペースも節約できます。管理サーバが複数の WebLogic Server を持つドメインを管理する場合、JAR ファイルしかデプロイできません。Administration Console は展開されたディレクトリを管理対象サーバにコピーしないからです。

`jar` ユーティリティは、Java Development Kit の `bin` ディレクトリに格納されています。パスに `javac` が指定されている場合、`jar` も指定されています。`jar` コマンドの構文と動作は、UNIX `tar` コマンドとほぼ同じです。

`jar` コマンドの最も一般的な使い方は次のとおりです。

`jar cf` *`jar-file files ...`*

`jar-file` という名前の JAR ファイルを作成し、指定したファイルを統合します。ファイル リストにディレクトリを入れた場合、そのディレクトリとサブディレクトリ内のすべてのファイルが JAR ファイルに追加されます。

`jar xf` *`jar-file`*
現在のディレクトリ内の JAR ファイルを展開（分解）します。

`jar tf` *`jar-file`*
JAR ファイルの内容を一覧表示します。

最初のフラグは、操作（create、extract、または一覧表示（tell））を指定します。`f` フラグの後には、JAR ファイル名を指定しなければなりません。`f` フラグを指定しないと、`jar` は JAR ファイルの内容を *`stdin`* から読み込むか、または *`stdout`* に書き出します。このような処理は一般的ではありません。`jar` コマンド オプションの詳細については、JDK ユーティリティのドキュメントを参照してください。

XML デプロイメント記述子

コンポーネントとアプリケーションには、ディレクトリまたは JAR ファイルの内容について説明したデプロイメント記述子という XML ドキュメントが組み込まれています。デプロイメント記述子は、XML タグでフォーマットされたテキスト ドキュメントです。J2EE 仕様では、J2EE コンポーネントおよびアプリケーション用の標準的で移植性の高いデプロイメント記述子が定義されています。BEA は、コンポーネントまたはアプリケーションを WebLogic Server 環境にデプロイするために必要な WebLogic 固有のデプロイメント記述子をさらに定義しています。

表 3-1 に、コンポーネントとアプリケーションのタイプと、それらの J2EE 標準および WebLogic 固有のデプロイメント記述子を示します。

表 3-1 J2EE と WebLogic のデプロイメント記述子

コンポーネントまたはアプリケーション	スコープ	デプロイメント記述子
Web アプリケーション	J2EE	`WEB-INF\web.xml`
	WebLogic	`WEB-INF\weblogic.xml`

表 3-1 J2EE と WebLogic のデプロイメント記述子（続き）

コンポーネントまたはアプリケーション	スコープ	デプロイメント記述子
エンタープライズ Bean	J2EE	META-INF\ejb-jar.xml
	WebLogic	META-INF\weblogic-ejb-jar.xml META-INF\weblogic-cmp-rdbms-jar.xml
リソース アダプタ	J2EE	META-INF\ra.xml
	WebLogic	META-INF\weblogic-ra.xml
エンタープライズ アプリケーション	J2EE	META-INF\application.xml
クライアント アプリケーション	J2EE	application-client.xml
	WebLogic	client-application.runtime.xml

コンポーネントまたはアプリケーションをパッケージ化する場合は、デプロイメント記述子を格納するディレクトリ（WEB-INF または META-INF）を作成し、次にそのディレクトリ内に必要な XML デプロイメント記述子を作成します。

手動でデプロイメント記述子を作成することも、WebLogic 固有の Java ベース ユーティリティを使用して自動的に生成することもできます。デプロイメント記述子の生成の詳細については、3-5 ページの「デプロイメント記述子の自動生成」を参照してください。

開発者から J2EE 準拠の JAR ファイルを受け取った場合、そのファイルにはすでに J2EE 標準のデプロイメント記述子が組み込まれています。その JAR ファイルを WebLogic Server にデプロイするには、そのファイルの内容をディレクトリに展開し、必要な WebLogic 固有のデプロイメント記述子とその他の生成されたコンテナ クラスを追加して、新旧のファイルが入った新しい JAR ファイルを作成します。

デプロイメント記述子の自動生成

WebLogic Server には、Web アプリケーション、エンタープライズ JavaBean（バージョン 1.1 および 2.0）、エンタープライズ アプリケーションなどの J2EE コンポーネントまたはアプリケーションのデプロイメント記述子を自動的に生成する Java ベースのユーティリティがあります。

これらのユーティリティは、ステージング ディレクトリにアセンブルしたオブジェクトを検証し、サーブレット クラス、EJB クラスなどを基に適切なデプロイメント記述子を構築します。ユーティリティは、コンポーネントごとに標準 J2EE デプロイメント記述子と WebLogic 固有のデプロイメント記述子の両方を生成します。

WebLogic Server には、以下のユーティリティがあります。

- `weblogic.ant.taskdefs.ejb.DDInit`

  エンタープライズ JavaBean 1.1 のデプロイメント記述子を作成します。

- `weblogic.ant.taskdefs.ejb20.DDInit`

  エンタープライズ JavaBean 2.0 のデプロイメント記述子を作成します。

- `weblogic.ant.taskdefs.war.DDInit`

  Web アプリケーションのデプロイメント記述子を作成します。

- `weblogic.ant.taskdefs.ear.DDInit`

  エンタープライズ アプリケーションのデプロイメント記述子を作成します。

注意： これらのユーティリティは、ユーザのコンポーネントまたはアプリケーションに完全かつ厳密に従ったデプロイメント記述子ファイルを作成しようとしますが、必要な要素の多くに対して値を推測しなければなりません。この推測にはしばしば誤りがあり、コンポーネントまたはアプリケーションをデプロイするときに、WebLogic Server がエラーを返す原因になります。この場合、コンポーネントまたはアプリケーションをアンデプロイし、Administration Console のデプロイメント記述子エディタでデプロイメント記述子を編集してから、再デプロイする必要があります。デプロイメント記述子エディタの使用方法の詳細については、2-22 ページの「デプロイメント記述子の編集」を参照してください。

各ユーティリティは、単一のパラメータをとります。デプロイメント記述子の生成対象となるコンポーネントまたはアプリケーションのオブジェクトを格納するルート ディレクトリです。ルート ディレクトリは、WEB-INF または META-INF サブディレクトリを含むディレクトリです。

たとえば、WEB-INF ディレクトリ、JSP ファイル、または Web アプリケーションを構成するその他のオブジェクトを含む c:\stage というディレクトリを作成したものの、web.xml および weblogic.xml デプロイメント記述子を作成していない場合があります。それらを自動的に生成するには、次のコマンドを実行します。

```
$ java weblogic.ant.taskdefs.war.DDInit c:\stage
```

ユーティリティは、WEB-INF ディレクトリに web.xml および weblogic.xml デプロイメント記述子を作成します。

開発モードとプロダクション モード

WebLogic Server は、開発モードとプロダクション モードの 2 つの異なるモードで実行できます。このモードは、STARTMODE スクリプト変数をコンフィグレーションして決定します。この変数は、*domain_name*\startWebLogic にあって、変更が可能です。この STARTMODE 変数で、起動時のモードをプロダクション モードから開発 モードへと切り替えることができます。

開発モードを有効にするには、STARTMODE スクリプト変数を以下のようにコンフィグレーションします。

```
-Dweblogic.ProductionModeEnabled=false
```

プロダクション モードを有効にするには、この変数を以下のように設定します。

```
-Dweblogic.ProductionModeEnabled=true
```

注意： デフォルト設定は、false です。

WebLogic Server の開発モード、プロダクション モードでの起動についての詳細は、「WebLogic Server の起動と停止」を参照してください。

開発モードを指定すると、applications ディレクトリの自動デプロイ機能を使用できます。これは、WebLogic Server がインストールされている config/*domain_name*（*domain_name* は WebLogic Server ドメインの名前）にある管理サーバの applications ディレクトリに新しいファイルをコピーするということです。アプリケーションは自動的にデプロイされ更新されます。

プロダクション モードでは、WebLogic Server の Administration Console または weblogic.Deploy ツールを使用して、アプリケーションをデプロイする必要があります。どちらのデプロイメント方式も、ユーザ名とパスワードが必要です。このようにして、ファイル システムに対する書き込み権を持ち、サーバにアプリケーションをデプロイできる権限を持つユーザに関するセキュリティ上の懸念に対応します。

Web アプリケーションのパッケージ化

Web アプリケーションをパッケージ化する前に、3-15 ページの「クライアント アプリケーションのパッケージ化」を読み、WebLogic Server がアプリケーション クラスをどのようにロードするかを理解してください。

Web アプリケーションをステージングおよびパッケージ化するには、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。このディレクトリの名前は自由に付けることができます。
2. HTML ファイル、JSP ファイル、およびこれらの Web ページが参照する画像などのすべてのファイルを、ステージング ディレクトリにコピーします。その際、参照されるファイルのディレクトリ構造はそのまま維持します。たとえば、HTML ファイルに <img src="images/pic.gif"> というタグが定義されている場合、pic.gif ファイルはその HTML ファイルの images サブディレクトリに配置されなければなりません。
3. ステージング ディレクトリに WEB-INF および WEB-INF\classes サブディレクトリを作成して、デプロイメント記述子とコンパイル済みの Java クラスを格納します。
4. サーブレット クラスとヘルパー クラスを WEB-INF\classes サブディレクトリにコピーまたはコンパイルします。

5. サーブレットが使用するエンタープライズ Bean のホームおよびリモート インタフェース クラスを WEB-INF\classes サブディレクトリにコピーします。

   注意： 3-24 ページの「クラスローダの概要」を参照して、同じアプリケーションにあるサーブレットからの EJB 参照に影響する WebLogic Server のクラスロード メカニズムについて理解しておきます。

6. JSP タグ ライブラリを WEB-INF サブディレクトリにコピーします（タグ ライブラリは WEB-INF のサブディレクトリにインストールされます。.tld へのパスは .jsp ファイルにコーディングされています）。

7. 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。

8. WEB-INF サブディレクトリに web.xml および weblogic.xml デプロイメント記述子を自動的に生成する次のコマンドを実行します。

   ```
   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 ユーティリティ DDInit の詳細については、3-5 ページの「デプロイメント記述子の自動生成」を参照してください。

   または、WEB-INF サブディレクトリに web.xml および weblogic.xml ファイルを手動で作成することもできます。

   注意： web.xml および weblogic.xml ファイル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。

9. 次のような jar コマンドを実行して、ステージング ディレクトリを .war ファイルにパッケージ化します。

   ```
   jar cvf myapp.war -C staging-dir .
   ```

作成された .war ファイルは、エンタープライズ アプリケーション（.ear ファイル）に追加するか、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単独でデプロイすることができます。

エンタープライズ JavaBeans のパッケージ化

1 つまたは複数のエンタープライズ Bean を 1 つのディレクトリにステージングして、それらを EJB JAR ファイルにパッケージ化できます。

EJB をパッケージ化する前に、3-15 ページの「クライアント アプリケーションのパッケージ化」を読み、WebLogic Server が EJB クラスをどのようにロードするかを理解してください。

エンタープライズ Bean をステージングおよびパッケージ化するには、次の手順が必要です。

1. 一時的なステージング ディレクトリを作成します。
2. 対象となる Bean の Java クラスをステージング ディレクトリにコンパイルまたはコピーします。
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. 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。

5. META-INF サブディレクトリに ejb-jar.xml、weblogic-ejb-jar.xml、および（必要に応じて）weblogic-rdbms-cmp-jar-*bean_name*.xml デプロイメント記述子を自動的に生成する次のコマンドを実行します。

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

   staging-dir は、ステージング ディレクトリです。このユーティリティは EJB 1.1 用です。EJB 2.0 を作成する場合は、次のユーティリティを使用します。

   ```
   java weblogic.ant.taskdefs.ejb20.DDInit staging-dir
   ```

   デプロイメント記述子を生成する Java ベース ユーティリティ DDInit の詳細については、3-5 ページの「デプロイメント記述子の自動生成」を参照してください。

   または、EJB デプロイメント記述子ファイルを手動で作成することもできます。META-INF サブディレクトリに ejb-jar.xml および weblogic-ejb-jar.xml ファイルを作成します。この Bean がコンテナ管理される永続的なエンティティ Bean である場合、META-INF ディレクトリに、weblogic-rdbms-cmp-jar-*bean_name*.xml デプロイメント記述子を作成し、その Bean のエントリを追加します。weblogic-ejb-jar.xml ファイルの <type-storage> 属性を使用して、Bean をこの CMP デプロイメント記述子にマッピングします。

   注意： エンタープライズ Bean のコンパイルと EJP デプロイメント記述子の作成については、『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。

6. すべてのエンタープライズ Bean クラスとデプロイメント記述子をステージング ディレクトリに配置すると、次のような jar コマンドを使用して EJB JAR ファイルを作成できます。

   ```
   jar cvf jar-file.jar -C staging-dir .
   ```

   このコマンドによって作成された jar ファイルは、WebLogic Server にデプロイすることも、またはアプリケーション JAR ファイルにパッケージ化することもできます。

   -C *staging-dir* オプションを指定すると、jar コマンドはディレクトリを staging-dir に変更します。これにより、JAR ファイルに記録されるディレクトリ パスがエンタープライズ Bean のステージング ディレクトリを基準にした相対パスとなります。

エンタープライズ Bean には、コンテナ クラスが必要となります。コンテナ クラスとは、WebLogic EJB コンパイラによって生成され、エンタープライズ Bean を WebLogic Server にデプロイできるようにするためのクラスです。WebLogic EJB コンパイラは、EJB JAR ファイル内のデプロイメント記述子を読み取って、コンテナ クラスの生成方法を決定します。WebLogic EJB コンパイラは、エンタープライズ Bean をデプロイする前に JAR ファイル上で実行できます。また、デプロイメント時に WebLogic Server にコンパイラを実行させることもできます。WebLogic EJB コンパイラの詳細については、『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。

リソース アダプタのパッケージ化

1 つまたは複数のリソース アダプタを 1 つのディレクトリにステージングして、それらを JAR ファイルにパッケージ化できます。

リソース アダプタをパッケージ化する前に、3-15 ページの「クライアント アプリケーションのパッケージ化」を読み、WebLogic Server がクラスをどのようにロードするかを理解してください。

リソース アダプタをステージングおよびパッケージ化するには、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. 対象となるリソース アダプタの Java クラスをステージング ディレクトリにコンパイルまたはコピーします。
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 `ra.xml` デプロイメント記述子を作成して、そのリソース アダプタのエントリを追加します。

   注意： `ra.xml` の文書型定義の詳細については、以下の Sun Microsystems のドキュメントを参照してください。

   http://java.sun.com/dtd/connector_1_0.dtd

5. `META-INF` サブディレクトリに `weblogic-ra.xml` デプロイメント記述子を作成して、そのリソース アダプタのエントリを追加します。

注意 : `weblogic-ra.xml` 文書型定義の詳細については、『WebLogic J2EE コネクタ アーキテクチャ』を参照してください。

6. すべてのリソース アダプタ クラスとデプロイメント記述子をステージング ディレクトリに配置すると、次のような `jar` コマンドを使用してリソース アダプタ JAR ファイルを作成できます。

   ```
   jar cvf jar-file.jar -C staging-dir .
   ```

   このコマンドによって作成された `jar` ファイルは、WebLogic Server にデプロイすることも、またはアプリケーション JAR ファイルにパッケージ化することもできます。

   `-C staging-dir` オプションを指定すると、`jar` コマンドはディレクトリを `staging-dir` に変更します。これにより、JAR ファイルに記録されるディレクトリ パスがリソース アダプタのステージング ディレクトリを基準にした相対パスとなります。

   注意 : WebLogic Server へのデプロイメントに対するリソース アダプタの作成および既存のリソース アダプタの変更については、第 2 章「WebLogic Server J2EE アプリケーションの開発」の 2-10 ページの「リソース アダプタの作成 : 主な手順」を参照してください。

エンタープライズ アプリケーションのパッケージ化

エンタープライズ アーカイブには、関連するアプリケーションの一部である EJB および Web モジュールが格納されます。EJB および Web モジュールは .ear 拡張子を持つ別の JAR ファイルに一緒にまとめられます。

.ear ファイルの META-INF サブディレクトリには、.ear ファイルにパッケージ化されたモジュールを識別する application.xml デプロイメント記述子が含まれています。application.xml ファイルの DTD は、http://java.sun.com/j2ee/dtds/application_1_2.dtd で提供されています。エンタープライズ アーカイブには WebLogic 固有のデプロイメント記述子は必要ありません。

次に、Pet Sotre サンプルの application.xml ファイルを示します。

```
<?xml version="1.0"  encoding="UTF-8"?>

<!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>estore</display-name>
  <description>Application description</description>
  <module>
    <web>
      <web-uri>petStore.war</web-uri>
      <context-root>estore</context-root>
    </web>
  </module>
  <module>
    <ejb>petStore_EJB.jar</ejb>
  </module>
  <security-role>
    <description>the gold customer role</description>
    <role-name>gold_customer</role-name>
  </security-role>
  <security-role>
    <description>the customer role</description>
    <role-name>customer</role-name>
  </security-role>
</application>
```

エンタープライズ アプリケーションをパッケージ化する前に、3-15 ページの「クライアント アプリケーションのパッケージ化」を読み、WebLogic Server がエンタープライズ アプリケーション クラスをどのようにロードするかを理解してください。

エンタープライズ アプリケーションをステージングおよびパッケージ化するには、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. Web アーカイブ（.war ファイル）と EJB アーカイブ（.jar ファイル）をステージング ディレクトリにコピーします。
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. 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。

5. META-INF サブディレクトリに application.xml デプロイメント記述子を自動的に生成する次のコマンドを実行します。

   ```
   java weblogic.ant.taskdefs.ear.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 ユーティリティ DDInit の詳細については、3-5 ページの「デプロイメント記述子の自動生成」を参照してください。

   または、META-INF ディレクトリに application.xml ファイルを手動で作成することもできます。このファイルの要素の詳細については、付録 A「application.xml デプロイメント記述子の要素」を参照してください。

6. 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デプロイできます。

クライアント アプリケーションのパッケージ化

WebLogic Server アプリケーションには必要ありませんが、J2EE にはクライアント アプリケーションをデプロイするための仕様が定義されています。J2EE クライアント アプリケーション モジュールは、JAR ファイルにパッケージ化されます。この JAR ファイルには、クライアント JVM (Java Virtual Machine) で実行される Java クラスと、EJB (Enterprise JavaBeans) およびクライアントによって使用されるその他の WebLogic Server リソースを記述するデプロイメント記述子が収められています。

Sun が提供する事実上の標準デプロイメント記述子である application-client.xml が J2EE クライアントに対しても使用され、補足デプロイメント記述子には、その他の WebLogic 固有のデプロイメント情報が収められます。

注意： これらのデプロイメント記述子については、付録 A「application.xml デプロイメント記述子の要素」の 「application.xml デプロイメント記述子の要素」を参照してください。

EAR ファイルのクライアント アプリケーションの実行

アプリケーションの配布を簡略化するため、J2EE では、WebLogic Server が使用するサーバサイド モジュールに合わせて、EAR ファイルにクライアントサイド コンポーネントを入れる方式を定義しています。こうすることにより、サーバサイド、クライアントサイド両方のコンポーネントを 1 つにまとめて配布できます。

クライアント JVM は、アプリケーション用に作成した Java クラスと、WebLogic Server クラスなどアプリケーションが依存する Java クラスのすべての位置を把握できなければなりません。クライアント アプリケーションをステージングするには、クライアントで必要なすべてのファイルを 1 つのディレクトリにコピーし、そのディレクトリを 1 つの JAR ファイルにまとめます。クライアント アプリケーション ディレクトリの最上位には、アプリケーションを起動するバッチ ファイルまたはスクリプトを置くことができます。`classes` サブディレクトリを作成して、Java クラスと JAR ファイルを格納し、起動スクリプト内のクライアントの `Class-Path` にそれらを追加します。Java Runtime Environment (JRE) と Java クライアント アプリケーションのパッケージ化もできます。

注意 : クライアント コンポーネント JAR 内で `Class-Path` manifest エントリを使用すると、J2EE 標準ではこれにまだ対応していないため、移植できません。

JAR ファイル manifest の `Main-Class` 属性は、クライアント アプリケーションのメイン クラスを定義します。クライアントでは、通常 、`java:/comp/env` JNDI ルックアップを使用して、`Main-Class` 属性を実行します。デプロイする側として、JNDI ルックアップ エントリの実行時の値を与え、クライアントの `Main-Class` 属性を呼び出す前にコンポーネント ローカルな JNDI ツリーを設定する必要があります。クライアントのデプロイメント 記述子に JNDI ルックアップ ツリーを定義します（「クライアント アプリケーションのデプロイメント記述子の要素」を参照してください）。

`weblogic.ClientDeployer` を使用して、J2EE EAR ファイルからクライアント サイド JAR ファイルを展開し、デプロイ可能な JAR ファイルを作成します。`weblogic.ClientDeployer` クラスは、以下の構文を使って Java コマンド ラインで実行します。

```
java weblogic.ClientDeployer ear-file client
```

`ear-file` 引数は、展開ディレクトリ（すなわち、拡張子が `.ear` となっている Java アーカイブ ファイル）で、その中に 1 つ以上のクライアント アプリケーション JAR ファイルが入っています。

例 :

```
java weblogic.ClientDeployer app.ear myclient
```

`app.ear` は、`myclient.jar` にパッケージ化された J2EE クライアントを含む EAR ファイル。

EAR ファイルからクライアントサイド JAR ファイルを展開したら、その weblogic.j2eeclient.Main ユーティリティを使って、クライアントサイド アプリケーションをブートストラップし、以下のように WebLogic Server を指すようにします。

```
java weblogic.j2eeclient.Main クライアントの jar URL [ アプリケーション変数 ]
```

例 :

```
java weblogic.j2eeclient.Main helloWorld.jar
t3://localhost:7001 Greetings
```

J2EE クライアント アプリケーションのデプロイメントに関する考慮事項

以下に、J2EE クライアント アプリケーションのデプロイメントに関する考慮事項を挙げます。

- WebLogic Server クライアント デプロイメント ファイルは、.runtime.xml というサフィックスを使って命名すること。
- weblogic.ClientDeployer クラスは、client.properties ファイルを生成し、それをクライアント JAR ファイルへ追加する処理を担当します。別プログラムの weblogic.j2eeclient.Main がローカル クライアント JNDI コンテキストを作成し、クライアント manifest ファイルで指定されているエントリポイントからクライアントを実行します。

  注意 : weblogic.ClientDeployer を使って J2EE クライアント アプリケーションを実行するには、(weblogic.jar ファイル内にある) weblogic.j2eeclient.Main クラスが必要です。

- application-client.xml ファイルに記述のあるリソースが、以下に示したタイプの 1 つである場合、weblogic.j2eeclient.Main クラスは、サーバ上のグローバルな JNDI ツリーからそれを取り出して、java:comp/env/ にバインドしようとします。

  ejb-ref

  javax.jms.QueueConnectionFactory

  javax.jms.TopicConnectionFactory

javax.mail.Session

javax.sql.DataSource

- weblogic.j2eeclient.Main クラスは、UserTransaction を java:comp/UserTransaction にバインドします。
- その他のクライアント環境は、weblogic.ClientDeployer クラスが作成した client.properties ファイルから java:comp/env/ にバインドされます。weblogic.j2eeclient.Main クラスは、バインドするものがなかったり、バインド処理が完了しない場合にエラー メッセージを出します。
- アプリケーション デプロイメント ファイル内の <res-auth> タグは、現在は無視され、Application として入力する必要があります。フォーム ベースの認証は、現在サポートしていません。

Apache ant を使った J2EE アプリケーションのパッケージ化

この節では、Apache の ant という拡張可能な Java ベースのツールを使った J2EE アプリケーションのビルドとパッケージ化について説明します。ant は、make コマンドに似ていますが、Java アプリケーションのビルド用に設計されたものです。ant ライブラリは WebLogic Server に同梱されており、カスタマがそのまま簡単に Java アプリケーションをビルドできるようになっています。

開発者は、eXtensible Markup Language (XML) を使って Ant ビルド スクリプトを作成します。XML タグで、ビルド対象、対象の依存関係、対象をビルドするために実行する処理を定義します。

ant で可能なことについての詳しい説明は、以下を参照してください。
http://jakarta.apache.org/ant/manual/index.html

Java ソースファイルのコンパイル

ant では、Java ソース ファイルのコンパイルのため、javac タスクを用意しています。以下の例では、カレント ディレクトリにあるすべての Java ファイルをコンパイルして classes ディレクトリに入れます。

```
<target name=”compile”>
  <javac srcdir=”.” destdir=”classes”/>
</target>
```

javac タスクに関連する全オプションについては、Apache ant のオンライン ドキュメントを参照してください。

WebLogic Server コンパイラの実行

ant からの任意の Java プログラムの実行は、カスタマイズした ant タスクを作成するか、そのまま java タスクを使ってプログラムを実行することにより可能です。ejbc または rmic などのタスクは、以下に示すように java タスクを使って実行できます。

コード リスト 3-1 WebLogic Server コンパイラの実行

```
<java classname="weblogic.ejbc" fork="yes" failonerror="yes">
  <sysproperty key="weblogic.home" value="${WL_HOME}"/>
    <arg line="-compiler java ${dist}/std_ejb_basic_containerManaged.jar
     ${APPLICATIONS}/ejb_basic_containerManaged.jar"/>
  <classpath>
     <pathelement path="${CLASSPATH}"/>
  </classpath>
</java>
```

上記のサンプルでは、fork システム コールを使って ejbc を実行するための Java プロセスを作成します。サンプルでは、system プロパティを提供して weblogic.home を定義し、arg タグを使ってコマンド ライン変数を用意します。呼び出される Java プロセスのためのクラスパスは、classpath タグを使って指定します。

J2EE デプロイメント ユニットのパッケージ化

すでに説明したように、J2EE アプリケーションはコンポーネントのタイプによってそれぞれのファイル拡張子をを持った JAR ファイルとしてパッケージ化されます。

- EJB は、JAR ファイルとしてパッケージ化される。
- Web アプリケーションは、WAR ファイルとしてパッケージ化される。
- リソース アダプタは、RAR ファイルとしてパッケージ化される。
- エンタープライズ アプリケーションは、EAR ファイルとしてパッケージ化される。

これらのコンポーネントは、J2EE 仕様に従って構成されます。標準の XML デプロイメント記述子に加えて、コンポーネントは WebLogic Server 固有の XML デプロイメント記述子を使用してパッケージ化することもできます。

ant は、こうした JAR ファイルの構築をより簡単に行えるタスクを提供します。JAR コマンドの機能の他に、ant は EAR ファイルおよび WAR ファイルをビルドするための特別なタスクを提供します。ant を使用して、JAR アーカイブ内に示されるパス名を指定できます。このパス名は、ファイル システム内の元のパスとは異なってもかまいません。この機能は、（J2EE がアーカイブ内の実際の位置を指定している）デプロイメント記述子のパッケージ化に便利です。この場合、ソース ツリー内の位置と対応しない場合もあります。ZipFileSet コマンドと関連情報については、Apache ant のオンライン ドキュメントを参照してください。

以下のリストを参照してください。

コード リスト 3-2 WAR タスクのサンプル

```
<war warfile="cookie.war" webxml="web.xml"
manifest="manifest.txt">
```

```
    <zipfileset dir="." prefix="WEB-INF" includes="weblogic.xml"/>
    <zipfileset dir="." prefix="images" includes="*.gif,*.jpg"/>
    <classes dir="classes" includes="**/CookieCounter.class"/>
    <fileset dir="." includes="*.jsp,*.html">
    </fileset>
</war>
```

J2EE デプロイメント ユニットをパッケージ化するには、以下のステップに従う必要があります。

1. webxml パラメータを使って、標準 XML デプロイメント記述子を指定します。
2. war タスクは、XML デプロイメント記述子 を WAR アーカイブ WEB-INF/web.xml 内の標準名に自動的にマップします。
3. Apache ant は manifest パラメータを使って指定した manifest ファイルを標準名 META-INF/MANIFEST.MF で格納します。
4. Apache ant の ZipFileSet コマンドを使って、WEB-INF ディレクトリに格納すべきファイルのセットを定義します（この場合は、WebLogic Server 固有のデプロイメント記述子 weblogic.xml のみ)。
5. 2 つめの ZipFileSet コマンドでは、images ディレクトリにあるすべてのイメージをパッケージ化します。
6. classes タグは、WEB-INF/classes ディレクトリにあるサーブレット クラスをパッケージ化します。
7. 最後に、カレント ディレクトリにあるすべての .jsp ファイルおよび .html ファイルをアーカイブに追加します。

WAR ファイルの構造にそのまま対応するディレクトリ内のファイルをステージングし、JAR ファイルをそのディレクトリから作成することにより、同じ結果が得られます。ant の JAR タスクの特別な機能を使用して、特定のディレクトリ構造にファイルをコピーしなくても済みます。

以下のサンプルでは、Web アプリケーションと EJB を 1 つずつビルドし、それらを 1 つの EAR ファイルにまとめてパッケージ化します。

コード リスト 3-3 パッケージ化のサンプル

```
<project name="app" default="app.ear">
   <property name="wlhome" value="/bea/wlserver6.1"/>
   <property name="srcdir" value="/bea/myproject/src"/>
   <property name="appdir" value="/bea/myproject/config/mydomain/applications"/>
   <target name="timer.war">
       <mkdir dir="classes"/>
      <javac srcdir="${srcdir}" destdir="classes" includes="myapp/j2ee/timer/*.java"/>
        <war warfile="timer.war" webxml="timer/web.xml"
manifest="timer/manifest.txt">
          <classes dir="classes" includes="**/TimerServlet.class"/>
       </war>
   </target>
   <target name="trader.jar">
   <mkdir dir="classes"/>
   <javac srcdir="${srcdir}" destdir="classes" includes="myapp/j2ee/trader/*.java"/>
   <jar jarfile="trader0.jar" manifest="trader/manifest.txt">
      <zipfileset dir="trader" prefix="META-INF" includes="*ejb-jar.xml"/>
      <fileset dir="classes" includes="**/Trade*.class"/>
      </jar>
      <ejbc source="trader0.jar" target="trader.jar"/>
   </target>
   <target name="app.ear" depends="trader.jar, timer.war">
      <jar jarfile="app.ear">
         <zipfileset dir="." prefix="META-INF" includes="application.xml"/>
         <fileset dir="." includes="trader.jar, timer.war"/>
      </jar>
```

```
   </target>
   <target name="deploy" depends="app.ear">
      <copy file="app.ear" todir="${appdir}/>
   </target>
</project>
```

ant の実行

BEA は、server/bin ディレクトリに ant 実行の簡単なスクリプトを用意しています。デフォルトでは、ant は build.xml ビルド ファイルをロードしますが、-f フラグを使って、これを変更できます。以下のコマンドを使って、上記のビルド スクリプトを使用してアプリケーションのビルドとデプロイメンとを行います。

```
ant -f yourbuildscript.xml
```

コンポーネント間のクラス参照の解決

アプリケーションでは、エンタープライズ Bean、サーブレットと JavaServer Pages、スタートアップ クラス、ユーティリティ クラス、およびサード パーティ製パッケージなど、さまざまな Java クラスを使用します。WebLogic Server では、個別のクラスローダにアプリケーションをデプロイして、独立を維持し、動的な再デプロイメントとアンデプロイメントを容易にします。このため、各コンポーネントが各クラスに個別にアクセスできるようにアプリケーションのクラスをパッケージ化する必要があります。場合によっては、一連のクラスを複数のアプリケーションまたはコンポーネントに格納する必要があります。この節では、アプリケーションを正常にステージングするために、WebLogic Server で複数のクラスローダを使用する方法について説明します。

クラスローダの概要

クラスローダは、要求されたクラスを見つけて Java 仮想マシン（JVM）にロードする Java クラスです。クラスローダは、クラスパスに示されたディレクトリまたは JAR ファイル内のファイルを検索して、参照を解決します。ほとんどの Java プログラムには 1 つのクラスローダがあります。JVM の起動時に作成されるシステム クラスローダです。WebLogic Server は、アプリケーションをデプロイするときに追加のクラスローダを作成します。これらのクラスローダはアプリケーションをアンデプロイするために破棄することができます。これにより、WebLogic Server では、変更されたアプリケーションをサーバを再起動せずに再デプロイできます。

クラスローダは階層的です。WebLogic Server の起動時に、Java システム クラスローダはアクティブになり、その後に WebLogic Server が作成するすべてのクラスローダの親になります。クラスローダは自身のクラスパスを検索する前に、常に親クラスローダにクラスを要求しますが、親クラスローダは子のクラスローダにはアクセスしません。検索はクラスローダの階層で上方向にのみ行われるためです。従って、子のクラスローダは兄弟クラスローダのクラスパス上にあるクラスを見つけることもできません。

検索プロトコルでは、重複したクラスが Java で処理される方法についても決められています。Java システム クラスパスで見つかったクラスは、子のクラスローダのクラスパスにある同じ名前のクラスよりも常に優先されます。このため、WebLogic Server を起動する前に、Java システム クラスパスにアプリケーションのクラスを配置することは避けてください。起動時に作成されたクラスローダは破棄されないため、このクラスローダに含まれるクラスは WebLogic Server を再起動しないと再デプロイできません。

アプリケーションのクラスローダ

WebLogic Server では、アプリケーションをデプロイするときに、EJB 用と Web アプリケーション用の 2 つの新しいクラスローダを作成します。EJB クラスローダは Java システム クラスローダの子、Web アプリケーション クラスローダは EJB クラスローダの子です。このため、Web アプリケーションのクラスは EJB クラスを見つけられますが、EJB クラスは Web アプリケーションのクラスを見つけられません。このクラスローダの階層には、サーブレットと JSP が EJB の

実装クラスに直接アクセスできるという効果的な作用もあります。EJB のクライアントと実装は同じ JVM にあるため、WebLogic Server は中間の RMI クラスを無視できます。

アプリケーションにエンタープライズ Bean を使用するサーブレットと JSP が含まれる場合は、次の手順に従います。

- サーブレットと JSP を .war ファイルにパッケージ化します。
- エンタープライズ Bean を EJB .jar ファイルにパッケージ化します。
- .war ファイルと .jar ファイルを .ear ファイルにパッケージ化します。
- .ear ファイルをデプロイします。

.war ファイルと .jar ファイルは別々にデプロイできますが、.ear ファイルに一緒にデプロイすると、サーブレットと JSP が EJB クラスを見つけられるようにクラスローダが配置されます。.war ファイルと .ejb ファイルを別々にデプロイすると、WebLogic Server はそれぞれのファイルごとに兄弟のクラスローダを作成します。つまり、.war ファイルには EJB ホームおよびリモート インタフェースを含める必要があります。EJB クライアントと実装クラスが異なる JVM にある場合と同じように、WebLogic Server は EJB 呼び出しで RMI のスタブ クラスとスケルトン クラスを使用する必要があります。

リソース アダプタ クラス

リソース アダプタ固有のクラスが WebLogic Server のシステム クラスパスにないことを確認してください。Web ドキュメントでリソース アダプタ固有のクラス（たとえば、EJB または Web アプリケーション）を使用する必要がある場合、これらのクラスを対応する各アーカイブ ファイル（たとえば、サーブレットの場合は .war の \classes ディレクトリ、EJB の場合は .jar の \classes ディレクトリ）にまとめる必要があります。

J2EE アプリケーションでの PreferWebInfClasses の使い方

デフォルトでは、Web アプリケーションのクラスローダは、Javasoft のドキュメントに記載されている標準の委託モデルに従います。サーブレット仕様では、Web アプリケーションは WAR ファイルからそのクラス定義を取得する必要があります。

この要件をサポートするため、Web アプリケーションのクラスローダは、WAR ファイルを検索してからそのクラスの親クラスローダに問い合わせるよう、BEA は Web アプリケーション用に委託モデルを変更するスイッチを組み込みました。このスイッチは `PreferWebInfClasses` といい、`WebAppComponentMBean` 上にあります。このスイッチの設定は、WebLogic Server コンソールで行えます。

`PreferWebInfClasses` を false （デフォルト）に設定すると、Web アプリケーションのクラスローダは標準の委託モデルに従います。true に設定すると、WAR ファイル内のクラス定義を検索してから、親にクラス定義を問い合わせます。

このスイッチは、仕様要件を満たします。しかし、親クラスローダ内にあるバージョンではなく、Web アプリケーション クラスローダ内にロードされている同じクラスの別のバージョンになる可能性があります。開発者がこの 2 つのインスタンスを切り分けるよう注意していないと、`ClassCastExceptions` を招きます。このため、標準の委託モデルを使用するよう、BEA はこの設定のデフォルトを false にしてあります。

共通ユーティリティ クラスとサード パーティ クラスのパッケージ化

複数のアプリケーションで使用するユーティリティ クラスを作成または取得する場合、それらのクラスを各アプリケーションに一緒にパッケージ化する必要があります。代わりに、WebLogic Server を実行するスクリプト内の java コマンドを編集して、Java システム クラスパスにそれらのクラスを追加できます。ただし、ユーティリティ クラスを変更し、そのクラスが Java システム クラスパスにある場合は、ユーティリティ クラスを変更した後に WebLogic Server を再起動する必要があります。

起動時に WebLogic Server が使用するクラスは Java システム クラスパスに存在しなければなりません。たとえば、接続プールに使用する JDBC ドライバは WebLogic Server の起動時にクラスパス内になければなりません。また、Java システム クラスパス内のクラスを変更する必要がある場合、またはクラスパスを変更した場合は、クラスまたはクラスパスを変更した後に WebLogic Server を再起動する必要があります。

スタートアップ クラスとアプリケーションの対話の処理

スタートアップ クラスはユーザが作成するクラスで、WebLogic Server の起動時に実行されます。スタートアップ クラスは Java システム クラスパスによって検索されるため、サーバを起動する前にシステム クラスパス内に配置しておく必要があります。また、スタートアップ クラスで必要なクラスは、すべてシステム クラスパスに含まれていなければなりません。

スタートアップ クラスがアプリケーション クラス（EJB インタフェースなど）を使用する場合、それらのクラスも WebLogic Server の起動クラスパスに追加しなければなりません。このため、それらのクラスは後でサーバを再起動しないと変更が反映されません。

アプリケーション オブジェクトを使用するスタートアップ クラスは、WebLogic Server でそのアプリケーションのデプロイメントが完了するまで、アプリケーション オブジェクトへのアクセスの試行を待つ必要があります。たとえば、スタートアップ クラスが EJB を使用する場合、ホームおよびリモート インタ

フェースをシステム クラスパス内に配置し、WebLogic Server が EJB アプリケーションのデプロイメントを完了するまで、スタートアップ クラスで EJB インスタンスを作成しないようにしておく必要があります。

Pet Store アプリケーションにはスタートアップ クラスがあります。その中で、スタートアップ クラスがアプリケーションのデプロイメントの完了を待つために使用できるメソッドが示されています。
`com.bea.estore.startup.StartBrowser` スタートアップ クラスでは、Pet Store アプリケーションにアクセスする初期 URL が表示されます。また、Windows 上では、その URL と共にブラウザが起動します。`StartBrowser` は、アプリケーションがデプロイされてサーバが接続要求の受け入れを開始するまで、`while` ループを実行します。

このクラスから抜粋した次の部分は、この動作を示しています。

```
while (loop) {
      try {
        socket = new Socket(host, new Integer(port).intValue());
        socket.close();

        // ブラウザを起動
        String[] cmdArray = new String[3];
        cmdArray[0] = "beaexec.exe";
        cmdArray[1] = "-target:browser";
        cmdArray[2] = "-command:\"http://"+host+":"+port+"\"";
        try {
          Process p = Runtime.getRuntime().exec(cmdArray);
          p.getInputStream().close();
          p.getOutputStream().close();
          p.getErrorStream().close();
        }
        catch (IOException ioe) {
        }
        loop = false;
      } catch (Exception e) {
        try {
          Thread.sleep(SLEEPTIME); // 500 ミリ秒ごとに試行
        } catch (InterruptedException ie) {}
        finally {
          try {
            socket.close();
          } catch (Exception se) {}
        }
      }
    }
```

システムがソケットの作成に失敗した場合、または BEA で提供する `beaexec.exe` ユーティリティがエラーを返す場合は、500 ミリ秒間スリープしてからループが繰り返されます。スタートアップ クラスで EJB インスタンスを作成する必要がある場合、EJB インスタンスを作成するメソッドが正常に実行されるまでループするという同様の手法を使用できます。

4　プログラミング トピック

以下の節では、WebLogic Server 環境でのプログラミングに関する情報を提供し、WebLogic Server の便利な機能とさまざまなプログラミング手法の使い方について説明します。

- ログ メッセージ
- WebLogic Server でのスレッドの使い方
- WebLogic Server アプリケーションでの JavaMail の使い方
- WebLogic Server クラスタのアプリケーションのプログラミング

ログ メッセージ

各 WebLogic Server インスタンスには、サーバが生成するメッセージを格納するログ ファイルがあります。アプリケーションは、ローカライズされたメッセージ カタログにアクセスするインターナショナライゼーション サービスを使用して、ログ ファイルにメッセージを書き込むことができます。ローカライゼーションが不要な場合は、`weblogic.logging.NonCatalogLogger` クラスを使用して、メッセージをログに書き込みます。このクラスは、クライアント アプリケーションがクライアント側のログ ファイルにメッセージを書き込むためにも使用されます。

ここでは、`NonCatalogLogger` クラスの使い方について説明します。インターナショナライゼーション インタフェースの使い方の詳細については、『BEA WebLogic Server インターナショナライゼーション ガイド』を参照してください。

ログ ファイルの名前、場所、およびその他のプロパティは、Administration Console で管理できます。NonCatalogLogger クラスを使って書き込まれるログ メッセージの内容は次のとおりです。

表 4-1 ログ メッセージのフォーマット

プロパティ	説明
Localized Timestamp	メッセージが生成された日付と時刻。年、月、日、時、分、および秒が記載される。
millisecondsFromEpoch	メッセージの生成時刻のミリ秒部分。
ServerName、MachineName、ThreadId、TransactionId	メッセージの生成元。TransactionId は、トランザクションのコンテキストでロギングされたメッセージにのみ示される。
User Id	エラーが報告されたときに実行されていたシステムのユーザ。
Subsystem	EJB、JMS、または RMI のようなメッセージのソース。ユーザ アプリケーションは、NonCatalogLogger コンストラクタにサブシステム String を提供する。
Message Id	メッセージ固有の 6 桁の識別子。499000 までのすべてのメッセージ ID は WebLogic Server 用に予約されている。

表 4-1 ログ メッセージのフォーマット

プロパティ	説明	
Severity	重要度を示す次の値のいずれか。	
	Debug	サーバ / アプリケーションがデバッグ モードでコンフィグレーションされた場合にのみ出力される。サーバ / アプリケーションの処理または状態の詳細が示される場合がある。
	Informational	後で調べる場合に備えて正常の処理をログに記録するために使用される。
	Warning	サーバ / アプリケーションの正常の処理に影響しない要注意の処理、イベント、またはコンフィグレーション。
	Error	ユーザ レベルのエラー。システム / アプリケーションは、割り込みやサービスの停止をせずにエラーに対処できる。
	上記以外にも、WebLogic Server メッセージ用として次の重要度レベルが予約されている。	
	Notice	警告メッセージ。サーバの正常な処理に影響しない要注意の処理またはコンフィグレーション。
	Critical	システム / サービス レベルのエラー。システムは回復できるが、サービスが瞬間的に停止したり、低下したりする可能性がある。
	Alert	特定のサービスが使用不能な状態であることを示す。システムのその他の部分は機能し続ける。自動回復できないので、管理者が直ちに問題を解決する必要がある。
	Emergency	サーバが使用不能な状態であることを示す。重大なシステム障害または危機的状態を示すために使用される。
ExceptionName	メッセージに例外がロギングされた場合、このフィールドには例外の名前が示される。	
Message Text	WebLogic Server のメッセージの場合、このフィールドには、システム メッセージ カタログに定義されているメッセージの「短い説明」が示される。	

NonCatalogLogger を使用するには、weblogic.logging.NonCatalogLogger クラスをインポートし、サブシステム String でコンストラクタを呼び出します。「MyApp」というサブシステム名を使って例を示します。

```
import weblogic.logging.NonCatalogLogger;
...
NonCatalogLogger mylogger = new NonCatalogLogger("MyApp");
```

NonCatalogLogger は、debug()、info()、warn()、および error() メソッドを提供し、それぞれのメソッドによって、Debug、Informational、Warning、および Error という重要度の付いたメッセージが書き込まれます。各メソッドには 2 つの署名があり、一方は String メッセージ引数を、他方は String メッセージ引数と java.lang.Throwable 引数を取ります。2 番目の形式を使用する場合、ログ メッセージにはスタック トレースが含まれます。

ログに書き込まれた、スタック トレースのない通知メッセージを示します。

```
mylogger.info("MyApp initialized.");
```

Java クライアントで NonCatalogLogger を使用している場合は、weblogic.log.FileName Java システム プロパティを用いて、java コマンド ラインでログ ファイルの名前を指定します。次に例を示します。

```
java -Dweblogic.log.FileName=myapp.log myapp
```

ログ メッセージに特別な処理が必要な場合は、独自のメッセージ ハンドラを追加します。メッセージ ハンドラを使用すると、処理対象のメッセージをフィルタによって選択できます。WebLogic Server ロギング インフラストラクチャはログ メッセージごとに JMX 通知を生成します。この通知は、フィルタによってメッセージに対応する登録済みメッセージ ハンドラに送られます。

JMX 機能の使い方の詳細については、weblogic.management.logging.WebLogicLogNotification を参照してください。

WebLogic Server でのスレッドの使い方

WebLogic Server は高度なマルチスレッド アプリケーション サーバであり、ホストとなっているコンポーネントのリソースの割り当て、同時実行性、およびスレッドの同期化を慎重に管理します。WebLogic Server のアーキテクチャを最大限に活用するには、標準 J2EE API を使用して作成したコンポーネントで、アプリケーションを構築する必要があります。

サーバサイド コンポーネントで新しいスレッドの作成を必要とするアプリケーションの設計は、次の理由から避けることをお勧めします。

- 独自のスレッドを作成するアプリケーションはあまり効率がよくありません。JVM のスレッドは、慎重に割り当てる必要のある限られたリソースです。サーバの負荷が増大すると、アプリケーションは停止するか、WebLogic Server で障害を引き起こすことがあります。デッドロックやスレッドの不足のような問題は、アプリケーションに重い負荷がかかるまで発生しないことがあります。
- マルチスレッド コンポーネントは複雑なため、デバッグが困難になります。アプリケーションが生成したスレッドと WebLogic Server スレッドの間の対話の場合、特に予測や分析が難しくなります。

このような警告にもかかわらず、スレッドの作成が適切な状況もあります。たとえば、複数のリポジトリを検索して、結合された結果セットを返すアプリケーションの場合、メイン クライアント スレッドを同期的に使用する代わりに、各リポジトリの新しいスレッドを非同期的に使用して検索を実行すると、より速く結果が返されることがあります。

アプリケーション コードでのスレッドの使用を決定したら、アプリケーションが作成するスレッドの数を制御できるように、スレッド プールを作成する必要があります。JDBC 接続プールのように、指定した数のスレッドをプールに割り当て、実行可能なクラスのプールから利用できるスレッドを取得します。プール内のすべてのスレッドが使用中の場合は、スレッドが戻されるまで待機します。スレッド プールを使用すると、パフォーマンスの問題を回避し、WebLogic Server の実行スレッドとアプリケーションの間のスレッドの割り当てを最適化できます。

スレッドでデッドロックが発生しそうな場所を把握しておき、デッドロックが発生したら確実に処理します。設計を慎重に見直して、スレッドがセキュリティ システムを危険にさらすことのないようにしておきます。

WebLogic Server スレッドとの望ましくない対話を避けるには、WebLogic Server コンポーネントの中でスレッドの呼び出しを使用しないようにします。たとえば、作成するスレッドからは、エンタープライズ Bean やサーブレットを使用しないでください。アプリケーション スレッドは、TCP/IP 接続による外部サービス、またはファイルの適切なロックや読み書きを行う外部サービスとの対話のような、独立していて分離されたタスクに使用するのが最適です。存続期間が短く、1 つの目的を実行して終了する（スレッドをプールに返す）スレッドの方が、他のスレッドと競合しません。

障害のポイントにクライアントを追加して、負荷が次第に大きくなるような状況でマルチスレッドのコードをテストします。アプリケーションのパフォーマンスと WebLogic Server の動作を監視し、プロダクション環境で障害が発生しないことを確認しておきます。

WebLogic Server アプリケーションでの JavaMail の使い方

WebLogic Server には、Sun Microsystems の JavaMail API バージョン 1.1.3 参照実装が含まれています。JavaMail API を使用すると、WebLogic Server アプリケーションに E メール機能を追加できます。JavaMail を使用すると、自社ネットワークまたはインターネット上の IMAP 対応および SMTP 対応サーバに Java アプリケーションからアクセスできます。JavaMail はメール サーバ機能を持っていないので、JavaMail を使用するにはメール サーバが必要です。

JavaMail API の使い方について詳細に説明したドキュメントは、Sun の Web サイトにある JavaMail のページで入手できます。ここでは、WebLogic Server 環境で JavaMail を使用する方法について説明しています。

`weblogic.jar` ファイルには、Sun の `javax.mail` および `javax.mail.internet` パッケージが含まれています。`weblogic.jar` には、JavaMail で必要な Java Activation Framework（JAF）パッケージも含まれています。

javax.mail パッケージには、Internet Message Access Protocol（IMAP）および Simple Mail Transfer Protocol（SMTP）メール サーバのプロバイダが含まれています。Sun には、JavaMail 用に独自の POP3 プロバイダがあります。これは、weblogic.jar には含まれていません。使用する場合は、Sun からその POP3 プロバイダをダウンロードし、WebLogic Server のクラスパスに追加できます。

JavaMail コンフィグレーション ファイル

JavaMail は、システムのメール転送機能を定義するコンフィグレーション ファイルに依存します。weblogic.jar ファイルには、Sun の標準コンフィグレーション ファイルが入っています。このファイルにより、IMAP および SMTP メール サーバが JavaMail 向けに使用可能になり、JavaMail が処理できるデフォルトのメッセージ タイプが定義されます。

JavaMail を拡張してサポートする転送、プロトコル、およびメッセージのタイプを追加する場合を除いて、JavaMail コンフィグレーション ファイルを変更する必要はありません。JavaMail を拡張する場合は、Sun の Web サイトから JavaMail をダウンロードして、拡張を追加するための Sun による手順に従います。次に、拡張した JavaMail パッケージを weblogic.jar. の前の WebLogic Server クラスパスに追加します。

WebLogic Server 用の JavaMail のコンフィグレーション

WebLogic Server で使用するために JavaMail をコンフィグレーションするには、WebLogic Server Administration Console でメール セッションを作成します。これにより、あらかじめコンフィグレーションしておくセッション プロパティを使用して、サーバサイド コンポーネントとアプリケーションで JNDI を用いて JavaMail サービスにアクセスできるようになります。たとえば、メール セッションを作成すると、メール ホスト、転送および格納プロトコル、デフォルトのメール ユーザを Administration Console で指定できるため、JavaMail を使用するコンポーネントでこれらのプロパティを設定する必要はありません。WebLogic Server は単一のセッション オブジェクトを作成し、JNDI を通じてそ

のオブジェクトを必要とするすべてのコンポーネントで利用できるようにするため、多数の電子メール ユーザを持つアプリケーションではメリットが得られます。

1. Administration Console で、左ペインの [メール] ノードをクリックします。
2. [新しい Mail Session のコンフィグレーション] をクリックします。
3. 右ペインのフォームに次のように入力します。
   - [名前] フィールドに新しいセッションの名前を入力します。
   - [JNDI 名] フィールドに JNDI ルックアップ名を入力します。作成するコードでは、この文字列を使用して、`javax.mail.Session` オブジェクトをルック アップします。
   - [プロパティ] フィールドにプロパティを入力して、セッションをコンフィグレーションします。プロパティ名は、JavaMail API Design Specification で指定されています。JavaMail では各プロパティのデフォルト値が提示されますが、アプリケーションのコードでそれらの値をオーバライドできます。次の表は、このフィールドに設定可能なプロパティの一覧を示します。

表 4-2 メール セッションのプロパティ フィールド

プロパティ	説明	デフォルト値
`mail.store.protocol`	E メールを取り出すために使用するプロトコル。 例 : `mail.store.protocol=imap`	付属の JavaMail ライブラリは IMAP をサポートしている。
`mail.transport.protocol`	E メールを送信するために使用するプロトコル。 例 : `mail.transport.protocol=smtp`	付属の JavaMail ライブラリは SMTP をサポートしている。
`mail.host`	メール ホスト マシンの名前。 例 : `mail.host=mailserver`	デフォルトではローカル マシン。

表 4-2 メール セッションのプロパティ フィールド（続き）

プロパティ	説明	デフォルト値
mail.user	E メールを取り出すデフォルト ユーザの名前。 例 : mail.user=postmaster	デフォルトは、user.name Java システム プロパティの値。
mail.*protocol*.host	特定のプロトコルに対応するメール ホスト。たとえば、mail.SMTP.host と mail.IMAP.host を別々のマシン名に設定できる。 例 : mail.smtp.host=mail.mydom.com mail.imap.host=localhost	mail.host プロパティの値。
mail.*protocol*.user	メール サーバにロギングするプロトコル固有のデフォルト ユーザ名。 例 : mail.smtp.user=weblogic mail.imap.user=appuser	mail.user プロパティの値。
mail.from	デフォルトの返信アドレス。 例 : mail.from=master@mydom.com	username@host
mail.debug	JavaMail デバッグ出力を有効にするには True に設定する。	False

メール セッションで設定したプロパティ セットは、オーバーライドするプロパティを含む Properties オブジェクトを作成すると、コード内でオーバーライドできます。
メール セッション オブジェクトを JNDI でルックアップした後、Properties を使用して Session.getInstance() メソッドを呼び出し、カスタマイズしたセッションを取得します。

JavaMail を使用したメッセージの送信

WebLogic Server コンポーネント内から JavaMail を使用してメッセージを送信する手順は次のとおりです。

1. JNDI （naming）、JavaBean Activation、JavaMail パッケージをインポートします。`java.util.Properties:` もインポートします。

```
import java.util.*;
import javax.activation.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.naming.*;
```

2. JNDI でメール セッションを次のようにルックアップします。

```
InitialContext ic = new InitialContext();
Session session = (Session) ic.lookup("myMailSession");
```

3. Administration Console で設定したセッションのプロパティをオーバーライドする必要がある場合は、`Properties` オブジェクトを作成してオーバーライドするプロパティに追加します。`getInstance()` を呼び出して、新しいプロパティの新しいセッション オブジェクトを取得します。

```
Properties props = new Properties();
props.put("mail.transport.protocol", "smtp");
props.put("mail.smtp.host", "mailhost");
// HTML フォームからのメール アドレスを発信元アドレスに使用する
props.put("mail.from", emailAddress);
Session session2 = session.getInstance(props);
```

4. `MimeMessage` を作成します。次の例で、*`to`*、*`subject`*、および *`messageTxt`* は文字列の変数で、ユーザが入力した内容が入ります。

```
Message msg = new MimeMessage(session2);
msg.setFrom();
msg.setRecipients(Message.RecipientType.TO,
                  InternetAddress.parse(to, false));
msg.setSubject(subject);
msg.setSentDate(new Date());
// コンテンツは、本文が 1 つある MIME マルチパート メッセージに
// 格納される
MimeBodyPart mbp = new MimeBodyPart();
mbp.setText(messageTxt);
```

```
Multipart mp = new MimeMultipart();
mp.addBodyPart(mbp);
msg.setContent(mp);
```

5. メッセージを送信します。

```
Transport.send(msg);
```

JNDI ルックアップでは、エラーが発生すると NamingException 例外が送出されます。JavaMail では、転送クラスの特定またはメール ホストとの通信に失敗した場合に MessagingException が送出されます。コードを try ブロックの中に配置し、これらの例外を補足して処理するようにしておきます。

JavaMail を使用したメッセージの読み込み

JavaMail API を使用すると、メッセージ ストアに接続できます。メッセージ ストアは IMAP サーバまたは POP3 サーバとなります。メッセージはフォルダに保存されます。IMAP の場合、メッセージ フォルダはメール サーバ上に格納されます。メッセージ フォルダには、受信したメッセージが入るフォルダとアーカイブされたメッセージが入るフォルダがあります。POP3 の場合は、メッセージの到着時にメッセージを保存するフォルダをサーバが提供します。クライアントは、POP3 サーバに接続するときに、メッセージを取得してクライアントのメッセージ ストアに転送します。

フォルダは、ディスクのディレクトリと同じような階層構造になっています。フォルダにはメッセージまたは他のフォルダを格納できます。デフォルトのフォルダは構造の最上位にあります。特別なフォルダ名である INBOX は、ユーザのプライマリ フォルダのことを指し、デフォルト フォルダの内部にあります。受信したメールを読み込むには、ストアからデフォルト フォルダを取得して、次にデフォルト フォルダから INBOX フォルダを取得します。

API では、指定した数または範囲のメッセージを読み取る、またはメッセージの特定の部分をあらかじめ取り出してフォルダのキャッシュに入れるなど、メッセージの読み込みについて複数のオプションを提供しています。詳細については、JavaMail API を参照してください。

WebLogic Server コンポーネント内から POP3 サーバで受信したメッセージを読み込む手順は次のとおりです。

1. JNDI（naming）、JavaBean Activation、JavaMail パッケージをインポートします。java.util.Properties: もインポートします。

```
import java.util.*;
import javax.activation.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.naming.*;
```

2. JNDI でメール セッションを次のようにルックアップします。

```
InitialContext ic = new InitialContext();
Session session = (Session) ic.lookup("myMailSession");
```

3. Administration Console で設定したセッションのプロパティをオーバーライドする必要がある場合は、`Properties` オブジェクトを作成してオーバーライドするプロパティに追加します。`getInstance()` を呼び出して、新しいプロパティの新しいセッション オブジェクトを取得します。

```
Properties props = new Properties();
props.put("mail.store.protocol", "pop3");
props.put("mail.pop3.host", "mailhost");
Session session2 = session.getInstance(props);
```

4. セッションから `Store` オブジェクトを取得し、`connect()` メソッドを呼び出してメール サーバに接続します。接続の認証を行うには、接続メソッドでメールホスト、ユーザ名、およびパスワードを提供する必要があります。

```
Store store = session.getStore();
store.connect(mailhost, username, password);
```

5. デフォルト フォルダを取得し、そのフォルダを使用して INBOX フォルダを取得します。

```
Folder folder = store.getDefaultFolder();
folder = folder.getFolder("INBOX");
```

6. フォルダ内のメッセージを、メッセージの配列に読み込みます。

```
Message[] messages = folder.getMessages();
```

7. メッセージの配列にあるメッセージを処理します。メッセージ クラスには、ヘッダ、フラグ、メッセージの内容など、メッセージのさまざまな部分にアクセスできるメソッドがあります。

IMAP サーバからのメッセージの読み込みは、POP3 サーバからのメッセージの読み込みと同様です。ただし IMAP の場合は、フォルダを作成および操作し、フォルダ間でメッセージを転送するメソッドが JavaMail API で提供されています。IMAP サーバを使用すると、POP3 サーバを使用する場合よりも少ないコードで、多機能な Web ベースのメール クライアントを実装できます。POP3 では、

おそらくデータベースまたはフォルダを表すためのファイル システムを使用して、WebLogic Server からメッセージ ストアを管理するコードを記述する必要があります。

WebLogic Server クラスタのアプリケーションのプログラミング

WebLogic Server のクラスタにデプロイされる JSP およびサーブレットは、セッション データを保持するために一部の要件を確認する必要があります。詳細については、『WebLogic Server クラスタ ユーザーズ ガイド』の「セッション プログラミングの必要条件」を参照してください。

WebLogic Server クラスタでデプロイされる EJB には、EJB のタイプに基づく制約があります。クラスタのさまざまな EJB タイプでの機能に関する詳細については、「WebLogic Server EJB コンテナ」を参照してください。EJB は、EJB デプロイメント記述子でプロパティを設定することでクラスタにデプロイできます。weblogic-ejb-jar.xml デプロイメント記述子は、クラスタ化に対応する XML デプロイメント要素を示します。

クラスタでデプロイするために EJB またはカスタム RMI オブジェクトのどちらかを開発する場合、クラスタ化されたオブジェクトの JNDI ツリーでのバインドについて理解するために、「クラスタ環境での WebLogic JNDI の使い方」も参照してください。

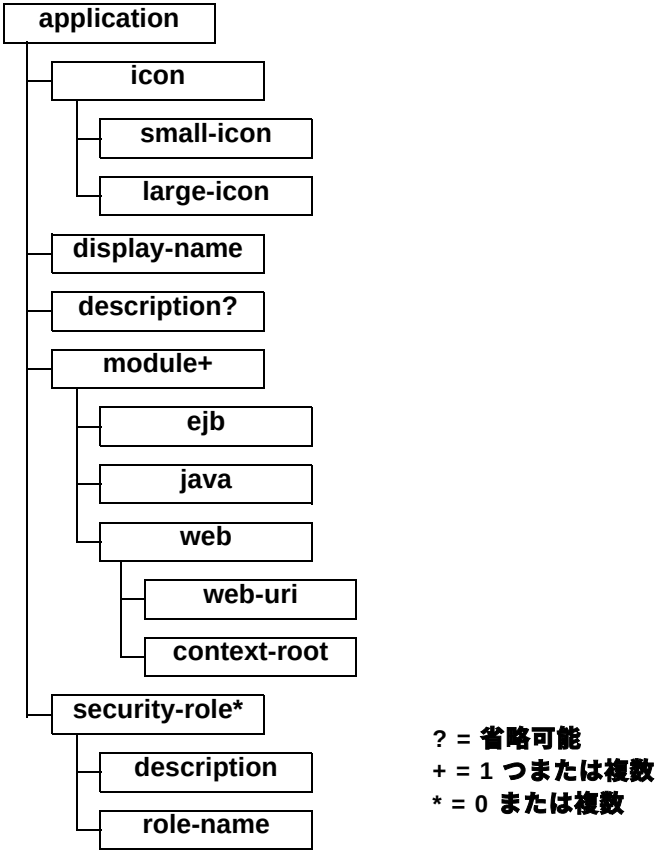
A application.xml デプロイメント記述子の要素

以下の節では、application.xml ファイルについて説明します。

application.xml ファイルは、エンタープライズ アプリケーション アーカイブのデプロイメント記述子です。ファイルは、アプリケーション アーカイブの META-INF サブディレクトリにあります。以下の DOCTYPE 宣言を最初に指定しなければなりません。

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

以下の図は、application.xml デプロイメント記述子の構造を示しています。



以降の節では、ファイル内に表示される各要素について説明します。

application

`application` は、アプリケーションのデプロイメント記述子のルート要素です。`application` 要素内の各要素については、以降の節で説明します。

icon

icon 要素は、GUI ツールでアプリケーションを表す小さい画像または大きい画像の位置を指定します。この要素は現在、WebLogic Server では使用されていません。

small-icon

省略可能。GUI ツールでアプリケーションを表す小さい（16x16 ピクセル）.gif 画像または .jpg 画像です。この要素は現在、WebLogic Server では使用されていません。

large-icon

省略可能。GUI ツールでアプリケーションを表す大きい（32x32 ピクセル）.gif 画像または .jpg 画像です。この要素は現在、WebLogic Server では使用されていません。

display-name

省略可能。display-name 要素は、アプリケーションの表示名（GUI ツールで表示することを想定した短い名前）を指定します。

description

省略可能です。アプリケーションに関する説明文です。

module

application.xml デプロイメント記述子には、エンタープライズ アーカイブ ファイル内の各モジュールに対する module 要素があります。各 module 要素には、アプリケーション内のモジュールのタイプと場所を示す ejb 要素、java 要素、または web 要素を指定できます。省略可能な alt-dd 要素は、後でアセンブリするデプロイメント記述子へのオプション URL を指定します。

ejb

アプリケーション ファイルの EJB モジュールを定義します。アプリケーションの EJB JAR ファイルへのパスも含まれます。

例 :

```
<ejb>petStore_EJB.jar</ejb>
```

java

アプリケーション ファイルのクライアント アプリケーション モジュールを定義します。

例 :

```
<java>client_app.jar</java>
```

web

アプリケーション ファイルの Web アプリケーション モジュールを定義します。web 要素には、web-uri 要素、およびオプションで context-root 要素があります。

web-uri

アプリケーション ファイルの Web モジュールの場所を定義します。この要素は、.war ファイル形式です。

context-root

省略可能。Web アプリケーションのコンテキスト ルートです。

例：

```
<web>
  <web-uri>petStore.war</web-uri>
  <context-root>estore</context-root>
</web>
```

security-role

security-role 要素には、アプリケーション共通のセキュリティ ロールが指定されます。各 security-role 要素には、description 要素（オプション）および role-name 要素があります。

description

省略可能。セキュリティ ロールの説明文です。

role-name

必須。アプリケーション内の認可に使用するセキュリティ ロールまたはプリンシパルの名前を定義します。各ロールは、weblogic-application.xml デプロイメント記述子の WebLogic Server ユーザまたはグループにマップされます。

例：

```
<security-role>
  <description>the gold customer role</description>
  <role-name>gold_customer</role-name>
</security-role>
<security-role>
  <description>the customer role</description>
  <role-name>customer</role-name>
</security-role>
```

B クライアント アプリケーションのデプロイメント記述子の要素

以下の節では、WebLogic Server の J2EE クライアント アプリケーションのデプロイメント記述子について説明します。application-client.xml という J2EE の標準デプロイメント記述子、およびクライアント アプリケーションの JAR ファイルから派生した名前を持つ WebLogic 固有の実行時デプロイメント記述子の 2 種類が必要です。

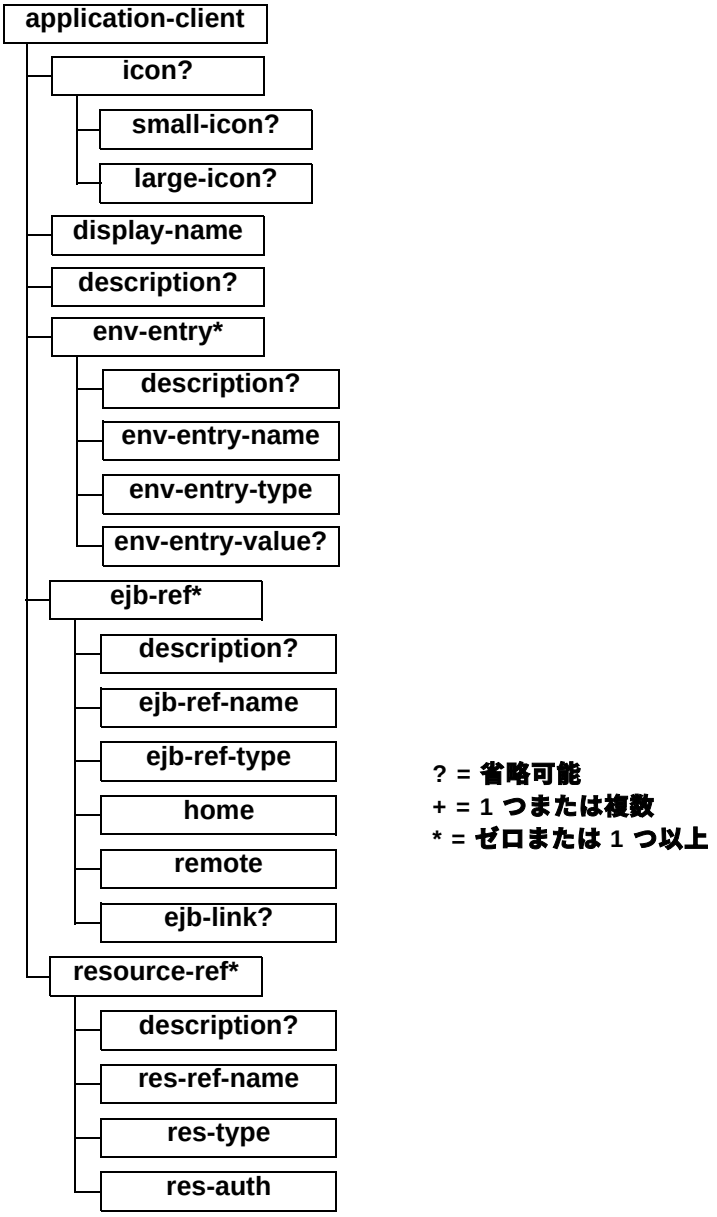
- application-client.xml のデプロイメント記述子の要素
- WebLogic クライアント アプリケーションの実行時デプロイメント記述子

application-client.xml のデプロイメント記述子の要素

application-client.xml ファイルは、J2EE クライアント アプリケーションのデプロイメント記述子です。以下の DOCTYPE 宣言を最初に指定しなければなりません。

```
<!DOCTYPE application-client PUBLIC "-//Sun Microsystems,
Inc.//DTD J2EE Application Client 1.2//EN"
"http://java.sun.com/j2ee/dtds/application-client_1_2.dtd">
```

以下の図は、application-client.xml デプロイメント記述子の構造を示しています。



以降の節では、ファイル内に表示される各要素について説明します。

application-client

application-client は、アプリケーション クライアントのデプロイメント記述子のルート要素です。アプリケーション クライアントのデプロイメント記述子は、クライアント アプリケーションで使用する EJB コンポーネントおよびその他のリソースを記述します。

application-client 要素内の各要素については、以降の節で説明します。

icon

省略可能。icon 要素は、GUI ツールでアプリケーションを表す小さい画像または大きい画像の位置を指定します。この要素は現在、WebLogic Server では使用されていません。

small-icon

省略可能。GUI ツールでアプリケーションを表す小さい（16x16 ピクセル）.gif 画像または .jpg 画像です。この要素は現在、WebLogic Server では使用されていません。

large-icon

省略可能。GUI ツールでアプリケーションを表す大きい（32x32 ピクセル）.gif 画像または .jpg 画像です。この要素は現在、WebLogic Server では使用されていません。

display-name

display-name 要素は、アプリケーションの表示名（GUI ツールで表示することを想定した短い名前）を指定します。

description

省略可能。description 要素は、クライアント アプリケーションの説明文を提供します。

env-entry

env-entry 要素には、クライアント アプリケーションの環境エントリの宣言が格納されます。

description

省略可能。description 要素には、特定の環境エントリの説明が格納されます。

env-entry-name

env-entry-name 要素には、クライアント アプリケーションの環境エントリの名前が格納されます。

env-entry-type

env-entry-type 要素には、Java タイプの環境エントリの完全修飾名が格納されます。java.lang.Boolean、java.lang.String、java.lang.Integer、java.lang.Double、java.lang.Byte、java.lang.Short、java.lang.Long、および java.lang.Float の値を指定できます。

env-entry-value

省略可能。env-entry-value 要素には、クライアント アプリケーションの環境エントリの値が格納されます。値には、指定した env-entry-type のコンストラクタで有効な文字列値を指定する必要があります。

ejb-ref

ejb-ref 要素は、クライアント アプリケーションで参照される EJB への参照の宣言に使用します。

description

省略可能。description 要素は、参照される EJB の説明文を提供します。

ejb-ref-name

ejb-ref-name 要素には、参照される EJB の名前が格納されます。通常、名前には、ejb/Deposit のように ejb/ というプレフィックスが付けられます。

ejb-ref-type

ejb-ref-type 要素には、参照される EJB で予期されるタイプ、Session または Entity が格納されます。

home

home 要素には、参照される EJB のホーム インタフェースの完全修飾名が格納されます。

remote

remote 要素には、参照される EJB のリモート インタフェースの完全修飾名が格納されます。

ejb-link

ejb-link 要素は、EJB 参照が J2EE アプリケーション パッケージのエンタープライズ JavaBean にリンクされるように指定します。ejb-link 要素の値は、同じ J2EE アプリケーションの ejb-name の名前と同じでなければなりません。

resource-ref

resource-ref 要素は、クライアント アプリケーションの外部リソースに対する参照の宣言が格納されます。

description

省略可能。description 要素は、参照される外部リソースの説明文を格納します。

res-ref-name

res-ref-name 要素は、リソース ファクトリ参照名を指定します。リソース ファクトリ参照名は、値にデータ ソースの JDNI 名が含まれるクライアント アプリケーションの環境エントリの名前です。

res-type

res-type 要素は、データ ソースのタイプを指定します。このタイプは、データ ソースによって実装されると予期される Java インタフェースまたはクラスによって指定されます。

res-auth

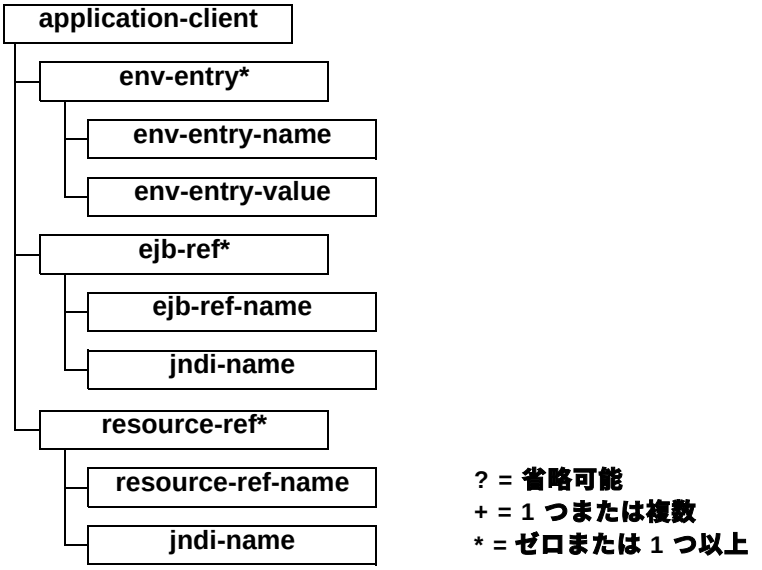
res-auth 要素は、EJB コードによってプログラムでリソース マネージャに対するサインオンを行うか、または EJB の代わりにコンテナによってリソース マネージャに対するサインオンを行うかを指定します。後者の場合、コンテナは、デプロイヤから提供される情報を使用します。res-auth 要素には、Application または Container のうち 1 つまたは複数を指定できます。

WebLogic クライアント アプリケーションの実行時デプロイメント記述子

この XML 形式のデプロイメント記述子は、他のデプロイメント記述子とは異なり、クライアント アプリケーションの JAR ファイル内には保存されませんが、クライアント アプリケーションの JAR ファイルと同じディレクトリに保存する必要があります。

デプロイメント記述子のファイル名は、JAR ファイルの基本名に .runtime.xml という拡張子が付けられます。たとえば、クライアント アプリケーションが c:\applications\ClientMain.jar というファイルにパッケージングされている場合、実行時デプロイメント記述子は c:\applications\ClientMain.runtime.xml というファイルにあります。

以下の図は、実行時デプロイメント記述子の各要素の構造を示しています。



application-client

application-client 要素は、WebLogic 固有のクライアント実行時デプロイメント記述子のルート要素です。

env-entry*

env-entry 要素は、デプロイメント記述子で宣言された環境エントリの値を指定します。

env-entry-name

env-entry-name 要素には、アプリケーション クライアントの環境エントリの名前が指定されます。

例 :

```
<env-entry-name>EmployeeAppDB</env-entry-name>
```

env-entry-value

env-entry-value 要素には、アプリケーション クライアントの環境エントリの値が指定されます。値には、単独の文字列パラメータをとる指定したタイプのコンストラクタで有効な文字列値を指定する必要があります。

ejb-ref*

ejb-ref 要素は、デプロイメント記述子の EJB 参照で宣言した JNDI 名を指定します。

ejb-ref-name

ejb-ref-name 要素には、EJB 参照の名前が指定されます。EJB 参照は、アプリケーション クライアントの環境のエントリです。名前には ejb/ というプレフィックスを付けることをお勧めします。

例 :

```
<ejb-ref-name>ejb/Payroll</ejb-ref-name>
```

jndi-name

jndi-name 要素は、EJB の JNDI 名を指定します。

resource-ref*

resource-ref 要素は、アプリケーション クライアントの外部リソースに対する参照を宣言します。要素には、リソース ファクトリの参照名が指定されます。この参照名は、アプリケーション クライアントのコードで予期されているリソース ファクトリのタイプ、および認証のタイプ（Bean かコンテナか）を示します。

例 :

```
<resource-ref>
  <res-ref-name>EmployeeAppDB</res-ref-name>
  <jndi-name>enterprise/databases/HR1984</jndi-name>
</resource-ref>
```

resource-ref-name

res-ref-name 要素は、リソース ファクトリ参照名を指定します。リソース ファクトリ参照名は、値にデータ ソースの JDNI 名が含まれるアプリケーション クライアントの環境エントリの名前です。

jndi-name

jndi-name 要素は、リソースの JNDI 名を指定します。

索引

記号

A

B

C

D

E

H

I

J

L

M

P

さ

し

す

せ

そ

て

ろ