BEA WebLogic Server™

WebLogic Server アプリケーションの開発

BEA WebLogic Server バージョン 7.0
マニュアルの改訂 : 2004 年 4

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	マニュアルの改訂	ソフトウェアのバージョン
なし	2002 年 8 月 20 日	BEA WebLogic Server バージョン 7.0

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

5. WebLogic Server デプロイメント

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

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。

マニュアルの内容は以下のとおりです。

- 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 クラスローディング」では、Java クラスローダの概要および WebLogic Server J2EE アプリケーションのクラスローディングについて説明します。
- 第 4 章「WebLogic Server 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 5 章「WebLogic Server デプロイメント」では、WebLogic Server J2EE アプリケーション デプロイメントについて説明します。
- 第 6 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「アプリケーション 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 (`WebLogic application.xml`)、および WebLogic 固有のアプリケーション デプロイメント記述子 (`weblogic-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) を使用して e- コマース アプリケーションを構築するアプリケーション開発者向けのマニュアルです。このマニュアルは、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 [Product Documentation 製品のドキュメント] をクリックします。

このマニュアルの印刷方法

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

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

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

関連情報

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』({DOCROOT}/ejb/index.html)
- 『WebLogic HTTP サーブレット プログラマーズ ガイド』({DOCROOT}/servlet/index.html)
- 『WebLogic JSP プログラマーズ ガイド』({DOCROOT}/jsp/index.html)
- 『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』({DOCROOT}/webapp/index.html)
- 『WebLogic JDBC プログラマーズ ガイド』({DOCROOT}/jdbc/index.html)
- 『WebLogic Web サービス プログラマーズ ガイド』({DOCROOT}/webserv/index.html)
- 『WebLogic J2EE コネクタ』({DOCROOT}/jconnector/index.html)

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@beasys.com までお送りください。寄せられた意見については、ドキュメントを作成および改訂する BEA の専門の担当者が直に目を通します。

電子メールのメッセージには、ご使用のソフトウェアの名前とバージョン、およびドキュメントのタイトルと日付をお書き添えください。本バージョンの BEA WebLogic Server について不明な点がある場合、または BEA WebLogic Server の

インストールおよび動作に問題がある場合は、BEA WebSupport (www.bea.com) を通じて BEA カスタマ サポートまでお問い合わせください。カスタマ サポートへの連絡方法については、製品パッケージに同梱されているカスタマ サポート カードにも記載されています。

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

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

表記規則

このマニュアルでは、全体を通して以下の表記規則が使用されています。

表記法	適用
〔Ctrl〕+ 〔Tab〕	複数のキーを同時に押すことを示す。
斜体	強調または書籍のタイトルを示す。

表記法	適用

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

斜体の等幅テキスト	コード内の変数を示す。 例： `String CustomerName;`
すべて大文字のテキスト	デバイス名、環境変数、および論理演算子を示す。 例： LPT1 BEA_HOME OR
{ }	構文の中で複数の選択肢を示す。
[ ]	構文の中で任意指定の項目を示す。 例： `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`
|	構文の中で相互に排他的な選択肢を区切る。 例： `java weblogic.Deployer [list|deploy|undeploy|update] password {application} {source}`

表記法	適用
. . .	コマンドラインで以下のいずれかを示す。 ■ 引数を複数回繰り返すことができる。 ■ 任意指定の引数が省略されている。 ■ パラメータや値などの情報を追加入力できる。
. . .	コード サンプルまたは構文で項目が省略されていることを示す。

1 WebLogic 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

BEA WebLogic Server™ J2EE アプリケーションは、WebLogic Server 上で動作する以下のコンポーネントのいずれかを含みます。

- 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
- エンタープライズ Java Bean (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
- コネクタ コンポーネント — リソース アダプタ

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.Deployer` 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 インスタンス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 などのアクセス メソッドを使用してリモート アクセスできます。

J2EE プラットフォーム

WebLogic Server では、Java 2 Platform, Enterprise Edition (J2EE) バージョン 1.3 の技術 (`http://java.sun.com/j2ee/sdk_1.3/index.html`) を実装しています。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。

注意： J2EE には下位互換性があるため、WebLogic Server 7.0 上で引き続き J2EE 1.2 を実行できます。

Web アプリケーション コンポーネント

Web アーカイブ (WAR) ファイルには、`.war` 拡張子が付いており、Web アプリケーションを構成するコンポーネントが格納されています。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 アプリケーションには、以下のファイルが含まれています。

- 少なくとも 1 つのサーブレットまたは JSP、およびヘルパー クラス

- `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 アプリケーションには、HTML ページおよび 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、クライアントからリクエストを受け取り、そのリクエストを処理して、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 ブラウザのリクエストに応えて動的な Web ページを生成するために使用します。

JavaServer Pages

JavaServer Pages (JSP) 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 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、`weblogic.jspc` によって JSP 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、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、JSP 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 をコンパイルし、サーブレット クラスを WAR ファイルにパッケージ化することもできます。サーブレットおよび JSP では、アプリケーションと共にデプロイするヘルパー クラスがさらに必要な場合があります。

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 コンポーネントを、ディレクトリ内でアセンブルし、それらを jar コマンドを使用して WAR ファイルにパッケージ化します。

これらのコンポーネントが参照する HTML ページ、JSP、および 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 クラスには、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 (必要に応じて) 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 Server 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 ディレクトリ構造の詳細については、『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』の「Web アプリケーションのディレクトリ構造」を参照してください。

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』({DOCROOT}/servlet/index.html)
- 『WebLogic JSP プログラマーズ ガイド』({DOCROOT}/jsp/index.html)
- 『WebLogic JSP Tag Extensions プログラマーズ ガイド』({DOCROOT}/taglib/index.html)

- 『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』 ({DOCROOT}/webapp/index.html)

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、当然ですが永続的です。

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 にも同様の機能がありますが、EJB コンテナでは動作しません。

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 (拡張子 .jar) 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 にエンタープライズ Bean をデプロイするには、その EJB のセキュリティ、トランザクション、およびライフ サイクルの各ポリシーを実行するクラスを生成する WebLogic EJB コンパイラ、`weblogic.ejbc` が動作している必要があります。

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、{DOCROOT}/ejb/index.html の『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。

コネクタ コンポーネント

WebLogic Server J2EE 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) 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。

WebLogic Server 環境にデプロイされたリソース アダプタを使用すると、J2EE アプリケーションでリモート EIS システムにアクセスできるようになります。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 ファイル) もデプロイメント ディレクトリも、そのままでは WebLogic Server にデプロイできません。最初に weblogic-ra.xml ファイルに 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、この XML ファイルをデプロイメント ディレクトリに追加する必要があります。

リソース アダプタのコンフィグレーション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』({DOCROOT}/jconnector/index.html) 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。これらのコンポーネントは、.ear 拡張子の付いた EAR ファイルにパッケージ化されています。

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.Deployer コマンドライン ユーティリティを使用します。

WebLogic 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して使用されます。これらのサービスは一般的に、CRM (カスタマ リレーションシップ マネージメント) システム、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。Web サービスは別々のコンピュータ上に常駐でき、多様なテクノロジを使用して実装することができますが、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 上のどのようなユーザでも簡単にアクセスできます。

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 Web サービス実装

WebLogic Web サービスは、WebLogic Server によってホストされます。WebLogic Web サービスは、エンタープライズ Java Beans などの標準 J2EE コンポーネントを使用して実装され、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。

- 標準のデータ転送および Web サービスと Web サービスのユーザ間での Web サービス起動呼び出し

  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。

- 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。

WebLogic Web サービスの設計、開発、および起動の詳細については、{DOCROOT}/webServices/index.html の『WebLogic Web サービス プログラマーズ ガイド』を参照してください。

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 リクエストを使用して、WebLogic Server コンポーネントを間接的に利用します。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 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、クライアントのクラスパスに追加できる JAR ファイルに `weblogic.jar` ファイルとともにパッケージ化したほうがよいでしょう。

WebLogic Server では、(単純な Java プログラムとは対象的に) 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で 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されています。`weblogic.ClientDeployer` コマンドライン ユーティリティは、この仕様に基づいてパッケージ化されたクライアント アプリケーションを実行するためにクライアント コンピュータ上で実行されます。J2EE クライアント アプリケーションの詳細については、4-26 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。

命名規約

WebLogic Server では、WAR、EAR、JAR、および RAR アーカイブ ファイルおよび展開ディレクトリについて、以下のプログラム上の命名規約に従う必要があります。

- エンタープライズ Java Beans の JAR アーカイブ ファイルの末尾には `.jar` 拡張子を付けます。
- リソース アダプタ RAR アーカイブ ファイルの末尾には `.rar` 拡張子を付けます。
- Web アプリケーション WAR アーカイブ ファイルの末尾には `.war` 拡張子を付けます。
- エンタープライズ アプリケーション EAR アーカイブ ファイルの末尾には `.ear` 拡張子を付けます。
- 上記のすべてのアーカイブ ファイルついて、展開されたそれぞれの非アーカイブ バージョン ファイルの末尾には、拡張子 `.jar`、`.rar`、`.war`、`.ear` を付けないでください。

2 WebLogic Server J2EE アプリケーションの開発

以下の節では、さまざまなタイプの WebLogic Server J2EE アプリケーションの作成、開発環境の設定、および Java プログラムのコンパイル準備の手順について説明します。

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。

Web アプリケーションの作成：主な手順

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 (JavaServer Page) を作成します。通常、Web デザイナが、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、JSP で参照される JSP taglibs 用の Java コードを記述します。通常、Java プログラマが、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 ファイルへサーブレットをコンパイルします。

   コンパイルの詳細については、2-16 ページの「Java コードのコンパイル」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か、WebLogic Server に含まれる Java ベースのユーティリティを使用して自動的に生成します。

   これらのデプロイメント記述子の要素、およびこれらの記述子を手動で作成する方法の詳細については、4-5 ページの「デプロイメント記述子の自動生成」、および『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 ファイル、および weblogic.xml ファイルを WAR ファイルにパッケージ化します。

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し、JSP、HTML ページ、およびこれらのページで参照されるマルチメディア ファイルをこのステージング ディレクトリの最上位に保存します。

コンパイルされたサーブレット クラス、taglibs を格納します。必要に応じて JSP ページからコンパイルされたサーブレットは、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、4-17 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. テストを行うため、この WAR ファイルを 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、4-6 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、『管理者ガイド』を参照してください。

7. WAR ファイルをプロダクション用に 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場合はエンタープライズ 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、第 5 章「WebLogic Server デプロイメント」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、WebLogic Server にデプロイするため、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、標準コンパイラを使用してクラス ファイルにコンパイルします。

   コンパイルの手順については、2-16 ページの「Java コードのコンパイル」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 コンテナ管理の永続性 (CMP) Bean ごとに異なる名前を持ちます。このファイル名は、`weblogic-ejb.jar` ファイル内の Bean のエントリで指定されます。

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。

XML デプロイメント記述子は、手動で作成、編集するか、WebLogic Server に含まれる Java ベースのユーティリティを使用して自動的に生成します。これらのファイルの自動生成の詳細については、4-5 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 JAR ファイルにパッケージ化します。

   EJB ステージング ディレクトリを作成します。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。`weblogic.ejbc` EJB コンパイラを実行して、EJB のセキュリティ、トランザクション、およびライフサイクルの各ポリシーを実行するクラスを生成します。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 *
   ```

   EJB JAR ファイル作成の詳細については、4-19 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. テストを行うため、この EJB JAR ファイルを 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、4-6 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、『管理者ガイド』を参照してください。

6. JAR ファイルをプロダクション用に 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場合はエンタープライズ 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、第 5 章「WebLogic Server デプロイメント」を参照してください。

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、さらに WebLogic Server にデプロイするため、それらをすべてリソース アダプタ アーカイブ (RAR) ファイルにパッケージ化します。

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ときは、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。例 :

   - <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、標準コンパイラを使用してクラス ファイルにコンパイルします。

   コンパイルの手順については、2-16 ページの「Java コードのコンパイル」を参照してください。

3. 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』を参照してください。

4. Java クラスを Java アーカイブ (JAR) ファイルにパッケージ化します。

   JAR ファイルの作成では、まず、ハード ディスクの任意の場所にコネクタのステージング ディレクトリを作成します。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 RAR アーカイブ ファイルの作成については、4-22 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。

5. テストを行うため、RAR リソース アダプタ アーカイブ ファイル を WebLogic Server に自動デプロイします。

   テスト時に、デプロイメント記述子の編集が必要になる場合があります。編集は手動、または Administration Console のデプロイメント記述子エディタを使用して行います。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、4-6 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。

6. 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場合はエンタープライズ アーカイブ (EAR) ファイルに含めます。

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、第 5 章「WebLogic Server デプロイメント」を参照してください。

既存のリソース アダプタ (RAR) の変更

以下に示すのは、既存のリソース アダプタ (RAR) を、WebLogic Server にデプロイするための変更方法の例です。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. ハード ディスクの任意の場所にリソース アダプタのステージング用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.rar
          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』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.rar -C c:/stagedir
```

7. WebLogic Server にリソース アダプタをデプロイします。デプロイメント ツールは、いくつかの種類があります。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、第 5 章「WebLogic Server デプロイメント」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、EJB、およびコネクタ (リソース アダプタ) の各コンポーネント、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、EJB、コネクタの各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、EJB、およびコネクタを作成します。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、EJB、およびコネクタ コンポーネント作成の概要については、それぞれ 2-2 ページの「Web アプリケーションの作成 : 主な手順」、2-4 ページの「エンタープライズ JavaBean の作成 : 主な手順」、2-6 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。

Web、EJB、およびコネクタ コンポーネントを構成する Java コードの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』、『WebLogic JSP プログラマーズ ガイド』、および『WebLogic J2EE コネクタ』を参照してください。

2. Web、EJB、およびコネクタの各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`、`web.xml`、`ra.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、4-5 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、EJB、およびコネクタ デプロイメント記述子の手動作成の詳細については、『Web アプリケーションのアセンブルとコンフィグレーション』、『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』、および『WebLogic J2EE コネクタ』を参照してください。

3. Web、EJB、およびコネクタ 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 コンポーネントは WAR ファイルに、EJB コンポーネントは EJB JAR ファイルに、コネクタ コンポーネントは RAR 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、4-17 ページの「Web アプリケーションのパッケージ化」、4-19 ページの「エンタープライズ JavaBeans のパッケージ化」、および 4-22 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. 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。

   `application.xml` デプロイメント記述子は、手動で作成するか、WebLogic Server に含まれる Java ベースのユーティリティを使用して自動的に生成します。このファイルの自動生成の詳細については、4-5 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 EAR ファイルに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共に、Web、EJB、およびコネクタ コンポーネントのアーカイブを EAR (`.ear` 拡張子) ファイルにパッケージ化します。このファイルは 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。

   EAR ファイル作成の詳細については、4-23 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. テストのため、この EAR エンタープライズ アプリケーションを WebLogic Server に自動デプロイします。

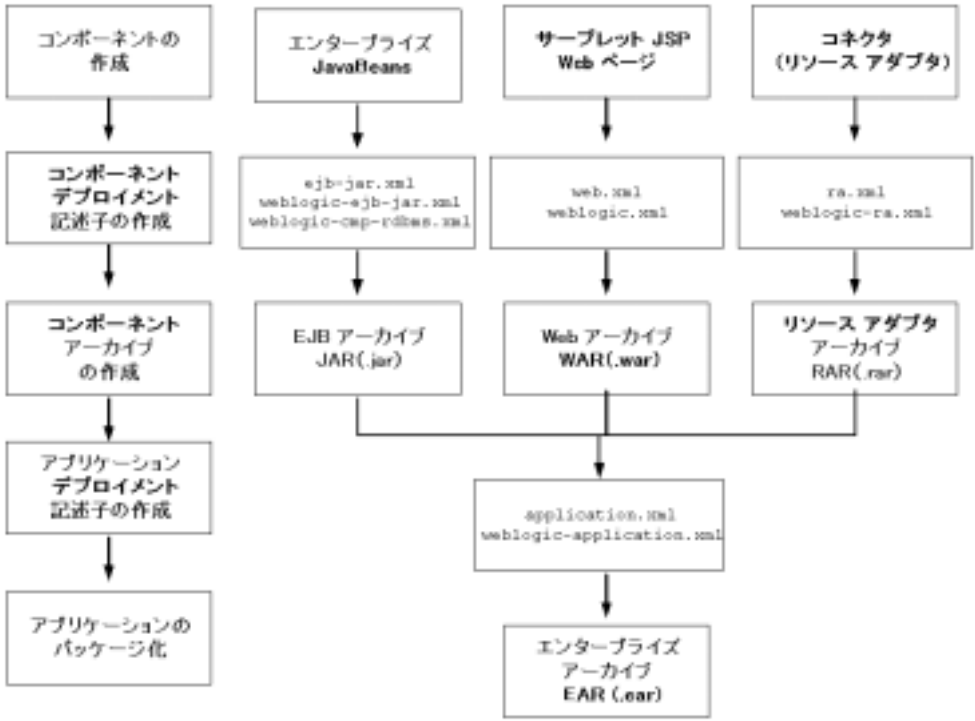
   エンタープライズ アプリケーションのテストでは、`application.xml` デプロイメント記述子の編集が必要になる可能性があります。ファイルの編集は手動、または Administration Console のデプロイメント記述子エディタを使用して行います。デプロイメント記述子エディタの使用方法の詳細については、4-6 ページの「デプロイメント記述子の編集」を参照してください。

7. プロダクション用に Administration Console を使用して EAR ファイルを WebLogic Server にデプロイします。

   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、第 5 章「WebLogic Server デプロイメント」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



開発環境の構築

WebLogic Server アプリケーションを開発するには、まず必要なソフトウェア ツールをアセンブルし、コードの作成、コンパイル、デプロイ、テスト、およびデバッグのための環境を設定しておきます。

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 または 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 ページおよび 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、4-6 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 Java 実行時環境 (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。未テスト コードに対しては、編集およびコンパイルに使用しているのと同じコンピュータに開発用 WebLogic Server インスタンスを設定するか、ネットワーク上の別の場所に WebLogic Server の開発場所を指定してください。

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 Java Message Service (JMS) などのサービスでは、Oracle、Sybase、Informix、Microsoft SQL Server、IBM DB2、または PointBase をサポートする JDBC ドライバが必要です。サポートされているデータベース システムおよび JDBC ドライバについては、「動作確認状況」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。WebLogic Server クラスタでは、マルチプールを設定してください。これにより、クラスタ内の複数のサーバ上の JDBC 接続プールに対して負荷分散が提供されます。

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セキュア ソケット レイヤ (SSL) プロトコル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」を参照してください。このページには、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が記載されています。

ツールをダウンロードするには、「BEA WebLogic Server Downloads」(http://commerce.bea.com/downloads/weblogic_server_tools.jsp) 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。

Java コードのコンパイル

WebLogic Server 用の Java コードのコンパイルは、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。

- 検索パスに標準 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、`samples\server\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 システムによってコンパイラとその他の JDK ツールの検索を可能にするには、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 VisualCaf・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。

コード 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 `CLASSPATH` に組み込みます。

- 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
- 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た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 `samples\server\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`
: `samples\server\stage\examples\clientclasses`
コンパイルされたクライアント クラスがサンプル ドメインに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。

`SERVER_CLASSES`
: デフォルトでは `samples\server\stage\examples\serverclasses`。サーバサイド クラスがサンプル ドメインに書き出されるディレクトリ。起動クラスとその他の Java クラスがあり、サーバの起動時に WebLogic Server CLASSPATH 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できないため、アプリケーション クラスは、通常このディレクトリにはコンパイルしないでください。

`EX_WEBAPP_CLASSES`
: `samples\server\stage\examples\applications\examplesWebApp\WEB-INF\classes`
Web アプリケーションによって使用されるクラスがサンプル ドメインに書き出されるディレクトリ。

`APPLICATIONS`
: *SAMPLES_HOME*\server\config\examples\applications
サンプル ドメイン用のアプリケーション ディレクトリ。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、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。

3 WebLogic Server J2EE アプリケーション クラスローディング

以下の節では、Java クラスローダの概要、および WebLogic Server J2EE アプリケーションのクラスローディングについて説明します。

Java クラスローダの概要

クラスローダは、Java 言語の基本的コンポーネントです。クラスローダは、Java 仮想マシン (JVM) の構成要素で、クラスをメモリにロードし、実行時にクラスファイルを検索してロードする役割を持つクラスです。Java プログラミングを適切に行うためには、クラスローダとその機能をよく理解する必要があります。この節では、Java クラスローダの概要を説明します。

Java クラスローダの階層

クラスローダには、親クラスローダと子クラスローダからなる階層があります。親クラスローダと子クラスローダの関係は、オブジェクトのスーパークラスとサブクラスの関係に似ています。ブートストラップ クラスローダは Java クラスローダ階層の親です。Java 仮想マシン (JVM) では、ブートストラップ クラスローダを作成し、このクラスローダによって、Java 開発キット (JDK) の内部クラスおよび JVM に組み込まれている `java.*` パッケージがロードされます (`java.lang.String` などがロードされます)。

拡張クラスローダは、ブートストラップ クラスローダの子です。拡張クラスローダは、JDK の拡張ディレクトリにあるすべての JAR ファイルをロードします。この機能は、クラスパスにエントリを追加することなく JDK を拡張できるので便利です。ただし、拡張ディレクトリ内にあるものは、すべて完全に独立している必要があり、拡張ディレクトリに含まれているクラスまたは JDK クラスしか参照できません。

システム クラスパス クラスローダは、JDK 拡張クラスローダを拡張します。システム クラスパス クラスローダは、JVM のクラスパスからクラスをロードします。アプリケーション固有のクラスローダ (WebLogic Server のクラスローダなど) は、システム クラスパス クラスローダの子です。

クラスのロード

クラスローダでは、クラスのロード時に委託モデルを使用します。クラスローダの実装では、まず要求されたクラスが既にロードされていないかチェックします。クラスを確認することで、ディスクからクラスを繰り返しロードする代わり

に、キャッシュ メモリのコピーが使用されるのでパフォーマンスが向上します。メモリにクラスがないと、親クラスローダがクラスをロードします。親クラスローダがクラスをロードできない場合のみクラスローダはクラスのロードを試行します。親と子両方のクラスローダにクラスがあると、親のクラスがロードされます。

注意： クラスローダは、親クラスローダにクラスのロードを要求してから、クラスを実際にロードします。必要に応じて、ローカルでチェックをしてから親クラスローダをチェックするようにクラスローダをコンフィグレーションできます。

PreferWebInfClasses 要素

WebAppComponentMBean には、PreferWebInfClasses 要素があります。デフォルトでは、この要素は False に設定されています。この要素を True に設定すると、クラスローダの委託モデルが無効になり、Web アプリケーションおよびシステム クラスローダの両方にロードされる同じクラスの取得が容易になります。さらにそれにより ClassCastException の取得も容易になります。

この要素を True に設定して、さまざまなサービスの BEA 実装 (多くは XERCES などの XML 処理クラスの実装) をオーバーライドするユーザもいます。この要素を True に設定する場合には、異なるクラスローダからロードされたクラスのインスタンスが混在しないように注意してください。

コード リスト 3-1 PreferWebInfClasses 要素

```
/**
* true の場合、Web アプリケーションの WEB-INF ディレクトリ内のクラスが、アプリケーションまたはシステム クラスローダにロードされるクラスに優先してロードされる
* デフォルト値は false
*/
boolean isPreferWebInfClasses();
void setPreferWebInfClasses(boolean b);
```

実行中プログラムのクラス変更

WebLogic Server では、サーバが稼働中に EJB などのアプリケーション コンポーネントの新しいバージョンをデプロイできます。このプロセスは、ホットデプロイ、またはホット再デプロイと呼ばれ、クラスのロードと密接な関係があります。

新しいバージョンのアプリケーションをデプロイすると、新しいアプリケーション クラスローダが作成されます。この方法は、新しいクラスローダによってアプリケーション クラスがロードされている限り有効です。クラスがシステム クラスパスにある場合は、サーバの稼働中にクラスを変更することはできません。

注意： Java クラスローダには、クラスをアンデプロイしたり、アンロードしたりする標準メカニズムはありません。また、新しいバージョンのクラスをロードすることもできません。この問題を解決するため、クラスパス クラスローダの子としてアプリケーション固有のクラスローダを作成する方法があります。

WebLogic Server アプリケーションにおけるクラスローダの概要

この節では、WebLogic Server アプリケーション クラスローダの概要を説明します。

アプリケーションのクラスロード

WebLogic Server のクラスロードは、アプリケーションの概念に基づいています。アプリケーションは、通常、アプリケーション クラスを含むエンタープライズ アーカイブ (EAR) ファイルにパッケージ化されています。EAR ファイル内にあるものは、すべて同じアプリケーションの一部と見なされます。他に以下のアプリケーションがあります。

- エンタープライズ JavaBean (EJB) JAR ファイル
- Web アプリケーション WAR ファイル

注意： リソース アダプタ RAR ファイルとそのクラスロードについては、「リソース アダプタ クラス」を参照してください。

EJB の JAR ファイルと Web アプリケーションの WAR ファイルを別々にデプロイすると、これらは 2 つのアプリケーションと見なされます。2 つのファイルをまとめて EAR ファイル内にデプロイすれば、1 つのアプリケーションとなります。したがって、複数のコンポーネントが同一アプリケーションの一部と見なされるためには、複数のコンポーネントをまとめて EAR ファイルにデプロイします。

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。リソース アダプタ固有のクラスを Web コンポーネント (EJB、Web アプリケーションなど) と共に使用する必要がある場合、それらのクラスを該当するコンポーネントのアーカイブ ファイル (EJB に使用する JAR ファイルまたは Web アプリケーションに使用する WAR ファイルなど) にまとめます。

すべてのアプリケーションは、独自のクラスローダ階層を受け取ります。この階層の親はシステム クラスパス クラスローダです。したがって、アプリケーションどうしが隔離されているため、アプリケーション A からアプリケーション B のクラスローダおよびクラスを見ることはできません。クラスローダには、兄弟概念またはフレンド概念はありません。アプリケーション クラスローダから見ることができるのは、親クラスローダであるシステム クラスパス クラスローダだけです。したがって、WebLogic Server は同一の JVM 内の複数の隔離されたアプリケーションのホストになります。

アプリケーション クラスローダの階層

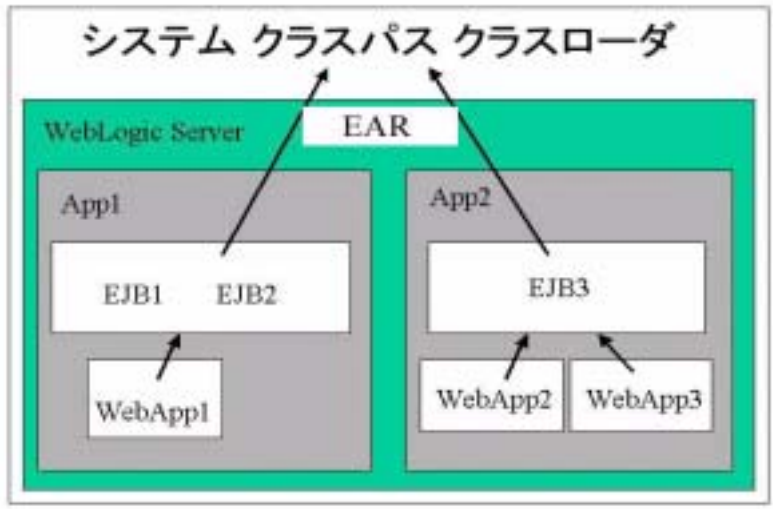
WebLogic Server では、アプリケーションのデプロイ時にクラスローダのセットが自動的に作成されます。基本アプリケーション クラスローダによって、アプリケーション内の EJB JAR ファイルがロードされます。また、Web アプリケーション WAR ファイルごとに子クラスローダが作成されます。

Web アプリケーションによる EJB の呼び出しがよくあるため、WebLogic Server アプリケーション クラスローダ アーキテクチャでは JavaServer Page (JSP) ファイルおよびサーブレットからその親クラスローダ内の EJB インタフェースを見ることができるようになっています。また、このアーキテクチャでは EJB 層を

再デプロイすることなく Web アプリケーションを再デプロイできます。実際には、EJB 層ではなく JSP ファイルおよびサーブレットを変更するほうが一般的です。

次の図は、WebLogic Server アプリケーションのクラスロード概念を示しています。

図 3-1 WebLogic Server のクラスロード



アプリケーションに EJB 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。
- この 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。
- パッケージ化した 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、まとめて 1 つの EAR ファイルにしてデプロイすることにより、サーブレットと JSP から EJB クラスを検索できるようにクラスローダが配置されます。WAR ファイルと JAR

ファイルを別々にデプロイすると、WebLogic Server ではそれらのファイルごとに兄弟のクラスローダを作成します。したがって、その 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。WebLogic Server では、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、EJB 呼び出しに RMI のスタブ クラスとスケルトン クラスを使用します。この概念については、次節の 3-7 ページの「アプリケーション クラスローダと、値渡しまたは参照渡し」で詳しく説明しています。

注意： Web アプリケーション クラスローダには、サーブレット実装クラスと JSP クラスを除く、Web アプリケーションのすべてのクラスが含まれます。各サーブレット実装クラスと JSP クラスは、独自のクラスローダを取得します。このクラスローダは Web アプリケーション クラスローダの子です。これにより、サーブレットと JSP を個別に再ロードできます。

アプリケーション クラスローダと、値渡しまたは参照渡し

最近のプログラミング言語では、パラメータを渡すときのモデルとして、値渡しと参照渡しがよく使用されます。値渡しの場合、パラメータおよび戻り値はメソッドの呼び出しごとにコピーされます。参照渡しの場合は、実際のオブジェクトを示すポインタ (または参照) がメソッドに渡されます。参照渡しでは、オブジェクトをコピーしなくてもよく、渡されたパラメータの状態を変更するメソッドも使用できるため、パフォーマンスが向上します。

WebLogic Server では、サーバ内のリモート メソッド インタフェース (RMI) 呼び出しのパフォーマンス向上のための最適化機能もあります。値渡しおよび RMI サブシステムのマーシャリングやアンマーシャリング機能を使用するのではなく、参照渡しを使用することによって、サーバがダイレクト Java メソッド呼び出しを行います。この方式により、パフォーマンスが大幅に改善され、EJB 2.0 ローカル インタフェースにもこの方式が採用されています。

RMI 呼び出しの最適化および参照呼び出しが使用されるのは、呼び出す側と呼び出される側が共に同じアプリケーションにある場合に限られます。通常、クラスローダが関係します。アプリケーションにはそれぞれ独自のクラスローダ階層があるため、どのアプリケーション クラスも両方のクラスローダに定義があり、複数のアプリケーション間で割り当てを行おうとすると ClassCastException エ

ラーを受け取ります。この問題を回避するため、WebLogic Server では複数のアプリケーション間での呼び出しには、同じ JVM 内であっても値呼び出しを使用しています。

注意： 複数アプリケーション間の呼び出しは、同一アプリケーション内の呼び出しに比べて遅くなります。したがって、コンポーネントは 1 つの EAR ファイルにまとめてデプロイし、高速の RMI 呼び出しおよび EJB 2.0 ローカル インタフェースを使用できるようにしてください。

以下に EJB バージョン 2.0 および 1.1 で呼び出されるパラメータの値渡しと参照渡しのリストを示します。

表 3-1 EJB バージョン 2.0 で呼び出されるパラメータ

パッケージ化	呼び出される側のインタフェース	デフォルトの呼び出しタイプ	enable-call-by-reference の影響
呼び出す側の WebApp/EJB が EAR ファイルにある	ローカル	参照渡し	False の場合、値渡し
	リモート	値渡し	True の場合、参照渡し
呼び出す側の EJB と呼び出される側の EJB が同じ EAR ファイルではなく、同じ JAR ファイルにある	ローカル	参照渡し	False の場合、値渡し
	リモート	値渡し	True の場合、参照渡し
呼び出す側の WebApp/EJB と呼び出される側の EJB が別々のデプロイメント モジュール (EAR/JAR) にある	ローカル	値渡し	なし
	リモート	値渡し	なし

表 3-2 EJB バージョン 1.1 で呼び出されるパラメータ

パッケージ化	呼び出される側のインタフェース	デフォルトの呼び出しタイプ	enable-call-by-reference の影響
呼び出す側の WebApp/EJB が EAR ファイルにある	リモート	参照渡し	False の場合、値渡し
呼び出す側の EJB と呼び出される側の EJB が同じ EAR ファイルではなく、同じ JAR ファイルにある	リモート	参照渡し	False の場合、値渡し
呼び出す側の WebApp/EJB と呼び出される側の EJB が別々のデプロイメント モジュール (EAR/JAR) にある	リモート	値渡し	なし

コンポーネントおよびアプリケーション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、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。

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。リソース アダプタ固有のクラスを Web コンポーネント (EJB、Web アプリケーションなど) と共に使用する必要がある場合、それらのクラスを該当するコンポーネントのアーカイブ ファイル (EJB に使用する JAR ファイルまたは Web アプリケーションに使用する WAR ファイルなど) にまとめます。

共有ユーティリティ クラスのパッケージ化

複数のアプリケーションでユーティリティ クラスを共有していることがよくあります。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、そうしたクラスは各アプリケーションと共に個別の JAR ファイルにパッケージ化します。JAR ファイルは、完全に独立している必要があり、EJB または Web コンポーネント内のクラスへの参照はありません。共有ユーティリティ クラスとして一般的なものはデータ転送オブジェクトまたは JavaBean で、Web 層と EJB 層の間で受け渡しされます。

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 クラスパスに共有ユーティリティ 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。

マニフェスト クラスパス

J2EE 仕様では、クラスの補助的な JAR ファイルを必要とすることをコンポーネントで指定する手段としてマニフェスト `Class-Path` エントリが用意されています。このマニフェスト `Class-Path` エントリを使用するのは、EJB JAR ファイルまたは WAR ファイルの一部として追加の補助的な JAR ファイルがある場合に限られます。その場合、JAR ファイルまたは WAR ファイルの作成時に、必要な JAR ファイルを参照する `Class-Path` 要素と共にマニフェスト ファイルを含める必要があります。

`utility.jar` ファイルを参照する簡単なマニフェスト ファイルを以下に示します。

```
Manifest-Version: 1.0 [CRLF]
Class-Path: utility.jar [CRLF]
```

このマニフェスト ファイルの先頭行には、常に `Manifest-Version` 属性を指定する必要があり、次に改行 (CR | LF |CRLF)、そして `Class-Path` 属性が続きます。マニフェスト フォーマットの詳細については、http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html#JAR を参照してください。

マニフェスト `Class-Path` エントリは、エントリが定義されている現在のアーカイブを基準として他のアーカイブを参照します。この構造により、複数の WAR ファイルおよび EJB JAR ファイルで、共通のライブラリ JAR を共有できます。たとえば、WAR ファイルに `y.jar` というマニフェスト エントリが含まれる場合、このエントリは、次のように WAR ファイルの次に (中ではない) 指定されなければなりません。

`/<directory>/x.war`

`/<directory>/y.jars`

マニフェスト ファイル自体は、`META-INF/MANIFEST.MF` のアーカイブに配置します。

詳細については、http://java.sun.com/docs/books/tutorial/jar/basics/manifest.html を参照してください。

4 WebLogic Server アプリケーションのパッケージ化

以下の節では、WebLogic Server コンポーネントのパッケージ化について説明します。コンポーネントは、WebLogic Server にデプロイする前にパッケージ化します。

パッケージ化の概要

WebLogic Server J2EE アプリケーションは、J2EE 仕様に従って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必要な場合があります。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。

詳細については、J2EE 1.3 仕様
(http://java.sun.com/j2ee/download.html#platformspec) を参照してください。

JAR ファイル

Java の `jar` ツールで作成される JAR (Java ARchive: Java アーカイブ) 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 ファイルまたは EAR 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 (作成 (**c**reate)、展開 (e**x**tract)、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 では、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ときに使用する 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 4-1 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 4-1 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.xml`
	WebLogic	`weblogic.xml`
エンタープライズ Bean	J2EE	`ejb-jar.xml`
	WebLogic	`weblogic-ejb-jar.xml` `weblogic-cmp-rdbms-jar.xml`
リソース アダプタ	J2EE	`ra.xml`
	WebLogic	`weblogic-ra.xml`
エンタープライズ アプリケーション	J2EE	`application.xml`
	WebLogic	`weblogic-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 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、4-5 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、そのファイルには既に 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 JAR ファイルの内容をディレクトリに展開し、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。JAR ユーティリティには、「u」オプションが付いていて、このオプションを使用するとファイルを変更したり、既存の JAR ファイルに直接ファイルを追加できます。

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 2.0) などの J2EE コンポーネント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、および既存の記述子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.marathon.ddinit.WebInit`

  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.marathon.ddinit.EJBInit`

  エンタープライズ JavaBeans 2.0 のデプロイメント記述子を作成します。`ejb-jar.xml` が存在する場合、`DDInit` はそのデプロイメント情報を使用して `weblogic-ejb-jar.xml` を生成します。

DDInit の制限

DDInit 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。この推測が間違っていると、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 はエラーを返します。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、「デプロイメント記述子の編集」を参照してください。

エンティティ Bean 間の関係は DDInit 特有の問題です。DDInit は 1 対 1 の双方向の関係を断定することしかできないからです。「多」の側面を持つ関係の場合、DDInit は関係の性質を推測します。

例

WEB-INF ディレクトリ、JSP ファイル、および 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とします。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。（済）たとえば、DDInit では→削除 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。→デプロイメント記述子を作成していないとします。

```
java weblogic.marathon.ddinit.WebInit c:\stage
```

ユーティリティによって web.xml および weblogic.xml デプロイメント記述子が生成され、WEB-INF ディレクトリに配置されます (WEB-INF ディレクトリがない場合は DDInit によって作成されます)。

デプロイメント記述子の編集

BEA では、WebLogic Server 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の編集用に以下の 2 つのツールを提供しています。

- 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`
- `weblogic-application.xml`
- `application-client.xml`
- `client-application.runtime.xml`

BEA XML エディタの使い方

XML ファイルを編集するには、完全な Java ベースの XML スタンドアロン エディタである BEA XML エディタを使用します。この XML エディタは、XML ファイルを作成および編集するためのシンプルで使いやすいツールです。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。このようにドキュメントを 2 通りに表示することにより、以下のいずれかの編集方法を選択できます。

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

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

BEA XML エディタの使用法およびダウンロードの詳細については、`http://developer.bea.com/tools/utilities.jsp` の BEA dev2dev Online を参照してください。

EJBGen について

EJBGen は、Javadoc マークアップを使用して EJB デプロイメント記述子ファイルを生成するエンタープライズ JavaBeans 2.0 のコード ジェネレータ (コマンド ライン ツール) です。Bean クラス ファイルに javadoc タグで注釈を付け、EJBGen を使用してリモート クラスとホーム クラス、および EJB アプリケーション用のデプロイメント記述子ファイルを生成し、編集および保守が必要な複数の EJB ファイルを 1 つにまとめます。

EJBGen 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』({DOCROOT}/ejb/EJB_utilities.html) を参照してください。

Administration Console のデプロイメント記述子エディタの使い方

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できます。特定の要素を更新してから [Apply 適用] ボタンをクリックするか、[Create 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。ディスクに保存するには、[Persist 永続化] ボタンをクリックします。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手順は次のとおりです。

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 [Deployments デプロイメント] ノードをクリックして展開します。
3. [Deployments デプロイメント] ノードの [EJBEJB] 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 [Edit EJB DescriptorEJB 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。

   左側のペインには、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 JARejb-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 Jarweblogic-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。
   - [CMPCMP] (Container-Managed Persistence: コンテナ管理の永続性) 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移動し、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表示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. [Apply 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移動し、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 [Configure a New *Element* 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. [Create 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移動し、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 [Delete *Element*(要素名) の削除] を選択します。

c. [Yes 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、[Validate 検証] をクリックします。

11. [Persist 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内容を、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、`web.xml` および `weblogic.xml` 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 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手順は次のとおりです。

1. ブラウザで 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 [Deployments デプロイメント] ノードをクリックして展開します。
3. [Deployments デプロイメント] ノードの [Web ApplicationsWeb 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 [Edit Web Application DescriptorWeb 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。

   左側のペインには、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、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 Descriptorweb-app] 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 Extweblogic-web-app] 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手順は次のとおりです。

a. 左側のペインでツリーを移動し、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表示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. [Apply 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に新しい要素を追加する手順は次のとおりです。

a. 左側のペインでツリーを移動し、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 [Configure a New *Element* 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. [Create 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から既存の要素を削除する手順は次のとおりです。

a. 左側のペインでツリーを移動し、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 [Delete *Element*(要素名) の削除] を選択します。

c. [Yes 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、[Validate 検証] をクリックします。

11. [Persist 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内容を、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、`ra.xml` および `weblogic-ra.xml` リソース アダプタ デプロイメント記述子を編集する手順を説明します。

リソース アダプタ 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 デプロイメント記述子を編集する手順は次のとおりです。

1. ブラウザで 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 [Deployments デプロイメント] ノードをクリックして展開します。
3. [Deployments デプロイメント] ノードの [Connectors 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 [Edit Connector Descriptor 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。

   左側のペインには、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 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、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。
   - [RAconnector] ノードには、`ra.xml` デプロイメント記述子の要素が含まれています。
   - [WebLogic RAweblogic-connection-factory-dd] ノードには、`weblogic-ra.xml` デプロイメント記述子の要素が含まれています。
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手順は次のとおりです。

a. 左側のペインでツリーを移動し、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表示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. [Apply 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に新しい要素を追加する手順は次のとおりです。

a. 左側のペインでツリーを移動し、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 [Configure a New *Element* 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. [Create 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から既存の要素を削除する手順は次のとおりです。

a. 左側のペインでツリーを移動し、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 [Delete *Element*(要素名) の削除] を選択します。

c. [Yes 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、[Validate 検証] をクリックします。

11. [Persist 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内容を、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

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` および `weblogic-application.xml`) を編集する手順をこの節で説明します。

`application.xml` および `weblogic-application.xml` ファイルの詳細については、付録 A「アプリケーション 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。

注意： 以下の手順は、`application.xml` ファイルおよび `weblogic-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、4-8 ページの「EJB デプロイメント記述子の編集」、4-11 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 4-13 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手順は次のとおりです。

1. ブラウザで 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 [Deployments デプロイメント] ノードをクリックして展開します。
3. [Deployments デプロイメント] ノードの [Applications 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 [Edit Application Descriptor 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。

左側のペインには、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移動し、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表示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. [Apply 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手順は次のとおりです。
   a. 左側のペインでツリーを移動し、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 [Configure a New *Element* 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. [Create 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手順は次のとおりです。
   a. 左側のペインでツリーを移動し、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 [Delete *Element*(要素名) の削除] を選択します。
   c. [Yes 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、[Validate 検証] をクリックします。
10. [Persist 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内容を、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パッケージ化

Web アプリケーションがプログラムに基づく Java クライアントによってアクセスされる場合は、4-26 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。この節では、サーバによるアプリケーション クラスのロード方法を説明しています。

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. 一時ステージング ディレクトリをハードディスクの任意の場所に作成します。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. 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。
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、*server*\bin\setenv.cmd ディレクトリにある setenv.cmd コマンドを実行します。*server* は WebLogic Server がインストールされている最上位ディレクトリです。

   UNIX の場合は、*server*/bin/setenv.sh ディレクトリにある setenv.sh コマンドを実行します。*server* は WebLogic Server がインストールされている最上位ディレクトリです。

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.marathon.ddinit.WebInit 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、4-5 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。

   代わりに、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.Deployer 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できます。

   注意： これで、Web アプリケーションのパッケージ化が完了しました。WebLogic Server におけるアプリケーションのデプロイメントの手順については、「アプリケーションのデプロイメント」を参照してください。

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 JavaBean (EJB) 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 がプログラムに基づく Java クライアントによってアクセスされる場合は、4-26 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。この節では、WebLogic Server による EJB クラスのロード方法を説明しています。

EJB のステージングおよびパッケージ化

エンタープライズ JavaBean (EJB) をステージングおよびパッケージ化する手順は次のとおりです。

1. ハード ディスクの任意の場所に一時ステージング ディレクトリを作成します (`c:\stagedir` など)。
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、*`server`*`\bin\setenv.cmd` ディレクトリにある `setenv.cmd` コマンドを実行します。*`server`* は WebLogic Server がインストールされている最上位ディレクトリです。

   UNIX の場合は、*`server`*`/bin/setenv.sh` ディレクトリにある `setenv.sh` コマンドを実行します。*`server`* は WebLogic Server がインストールされている最上位ディレクトリで、*`domain`* はドメインの名前です。
5. 次のコマンドを実行して、`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します。

```
java weblogic.marathon.ddinit.EJBInit 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では、EJB 2.0 用のデプロイメント記述子が生成されます。

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、4-5 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 EJB 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したら、次のように `jar` コマンドを使用して EJB JAR ファイルを作成します。

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

   このコマンドによって、WebLogic Server にデプロイ可能な JAR ファイルが作成されます。

   `-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 コンパイラに関して不明な点が

あれば、『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。

注意： これで、EJB のパッケージ化が完了しました。WebLogic Server におけるアプリケーションのデプロイメントの手順については、「アプリケーションのデプロイメント」を参照してください。

ejb-client.jar の使用

WebLogic Server では、`ejb-client.jar` ファイルを使用できます。`ejb-client.jar` ファイルを作成するには、この機能を Bean の `ejb-jar.xml` デプロイメント記述子ファイルで指定してから、`weblogic.ejbc` を使用して `ejb-client.jar` ファイルを生成します。`ejb-client.jar` には、`ejb-jar` ファイルの EJB を呼び出すためにクライアント プログラムに必要なクラス ファイルが格納されます。これらのファイルは、クライアントをコンパイルするために必要なクラスです。この機能を指定した場合、WebLogic Server は `ejb-client.jar` を自動的に作成します。

詳細については、『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』の「WebLogic Server コンテナ用の EJB のパッケージ化」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た後で Java アーカイブ (JAR) 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、「WebLogic Server J2EE アプリケーション クラスローディング」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. 一時ステージング ディレクトリをハードディスクの任意の場所に作成します。
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. リソース アダプタの Java クラスを入れる JAR を作成します。この JAR をステージング ディレクトリの最上位に追加します。
4. 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。
5. `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、次の 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`)
6. `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。
7. 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 RAR を作成できます。

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

このコマンドによって作成された RAR は、WebLogic Server にデプロイすることも、またはアプリケーション アーカイブ (EAR) 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 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。

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。この EJB と Web モジュールは、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ファイルと共に、拡張子 .ear のついた別の JAR ファイルにパッケージ化されます。

エンタープライズ アプリケーション デプロイメント記述子ファイル

EAR ファイルの `META-INF` サブディレクトリには、`application.xml` デプロイメント記述子が入っています。このデプロイメント記述子のフォーマット定義は Sun Microsystems によって提供されています。`application.xml` デプロイメント記述子によって、EAR ファイルにパッケージ化されるモジュールが識別されます。

`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。

`application.xml` 内では、アプリケーションを構成するモジュールおよびアプリケーションで使用されるセキュリティ ロールなどの項目を定義します。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>
```

補足デプロイメント記述子、`weblogic-application.xml` には WebLogic 固有のデプロイメント情報が収められます。このデプロイメント記述子は省略可能で、アプリケーション スコープのコンフィグレーションを行うときにのみ必要です。

アプリケーション スコープとは、WebLogic Server のコンフィグレーション全体に対してではなく、特定のエンタープライズ アプリケーションに対してリソースをコンフィグレーションすることです。リソースの例としては、アプリケーションで使用される XML パーサ、EJB エンティティ キャッシュ、JDBC 接続プールなどがあります。アプリケーション スコープの最大の利点は、特定のアプリケーションに対するリソースがそのアプリケーション専用になることです。

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

アプリケーション スコーピングについては、{DOCROOT}/jdbc/programming.html#1050534 の「アプリケーション スコープの JDBC 接続プール」、および {DOCROOT}/xml/xml_appscop.html の「XML アプリケーション スコーピング」を参照してください。

weblogic-application.xml デプロイメント記述子要素については、付録 A「アプリケーション デプロイメント記述子の要素」の「weblogic-application.xml デプロイメント記述子の要素」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 : 主な手順

エンタープライズ アプリケーションがプログラムに基づく Java クライアントによってアクセスされる場合は、4-26 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。この節では、WebLogic Server によるエンタープライズ アプリケーション クラスのロード方法を説明しています。

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. 一時ステージング ディレクトリをハードディスクの任意の場所に作成します。
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、*server*\bin*setenv.cmd* ディレクトリにある setenv.cmd を実行します。ここで *server* は WebLogic Server がインストールされている最上位ディレクトリです。

   UNIX の場合は、*server*/bin/*setenv.sh* ディレクトリにある setenv.sh コマンドを実行します。ここで、*server* は WebLogic Server がインストールされているディレクトリです。
5. META-INF サブディレクトリに、エンタープライズ アプリケーションを記述する application.xml デプロイメント記述子を作成します。このファイルの要素の詳細については、付録 A「アプリケーション デプロイメント記述子の要素」を参照してください。
6. (省略可能) 付録 A「アプリケーション デプロイメント記述子の要素」の説明に従って、META-INF ディレクトリに weblogic-application.xml ファイルを手動で作成します。

7. 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.Deployer 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。

   注意 : これで、エンタープライズ アプリケーションのパッケージ化が完了しました。WebLogic Server におけるアプリケーションのデプロイメントの手順については、「アプリケーションのデプロイメント」を参照してください。

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 仮想マシン) で実行される Java クラスと、EJB (エンタープライズ 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 リソースに関するデプロイメント記述子が収められます。

J2EE クライアントに対しては、事実上の標準である Sun のデプロイメント記述子 application-client.xml が使用され、補足デプロイメント記述子には 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 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」の「application-client.xml のデプロイメント記述子の要素」を参照してください。

EAR ファイル内のクライアント アプリケーションの実行

アプリケーションの配布を簡素化するため、J2EE によってクライアントサイド コンポーネントを WebLogic Server で使用されるサーバサイド モジュールと共に 1 つの EAR ファイルに含める方法を定義します。これによって、サーバサイドとクライアントサイドのコンポーネントを 1 つのユニットとして配布できます。

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、そのアプリケーションが依存する Java クラス (WebLogic Server クラスなど) を見つけることができなければなりません。クライアント アプリケーションのステージングでは、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 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。最後に、`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` に追加します。また、Java Runtime Environment (JRE) を Java クライアント アプリケーションに組み込むこともできます。

注意 : `Class-Path` マニフェスト エントリのクライアント コンポーネント JAR での使用は、J2EE 標準ではないため移植性はありません。

JAR ファイル マニフェストの `Main-Class` 属性は、クライアント アプリケーションのメイン クラスを定義しています。クライアントは、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。デプロイヤが、クライアントの `Main-Class` 属性を呼び出す前に JNDI ルックアップ エントリの実行時値を指定し、コンポーネント ローカル JNDI ツリーに入力します。クライアント デプロイメント記述子で JNDI ルックアップ エントリを定義します (「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)。

J2EE EAR ファイルからクライアントサイド JAR ファイルを展開して、デプロイ可能な JAR ファイルを作成するには、`weblogic.ClientDeployer` を使用します。`weblogic.ClientDeployer` クラスは、Java コマンド ラインで次の構文を使用して実行します。

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

ear-file 引数は、1 つまたは複数のクライアント アプリケーション JAR ファイルが格納されている展開されたディレクトリか、または拡張子 .ear を持つ Java アーカイブ ファイルです。

例 :

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

ここで、app.ear は、myclient.jar にパッケージ化された J2EE クライアントを格納する EAR ファイルです。

EAR ファイルからクライアントサイドの JAR ファイルが展開されたら、weblogic.j2eeclient.Main ユーティリティを使用し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、次のように WebLogic Server インスタンスを示すようにします。

```
java weblogic.j2eeclient.Main clientjar URL [application args]
```

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

J2EE クライアント アプリケーションのデプロイメントで特に考慮すべき事項を以下に示します。

- クライアント デプロイメント ファイルの名前には、サフィックス .runtime.xml を付けます。
- weblogic.ClientDeployer クラスによって、client.properties ファイルが生成され、クライアント JAR ファイルに追加されます。一方、weblogic.j2eeclient.Main プログラムを使用して、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 マニフェスト ファイルで指定されたエントリ ポイントから実行します。

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

- application-client.xml ファイルで指定されたリソースが、以下のいずれかのタイプである場合、weblogic.j2eeclient.Main クラスによってそのリソースはサーバのグローバル JNDI ツリーから java:comp/env/ にバインドされます。

  ejb-ref

  javax.jms.QueueConnectionFactory

  javax.jms.TopicConnectionFactory

  javax.mail.Session

  javax.sql.DataSource

- weblogic.j2eeclient.Main クラスによって、UserTransaction は java:comp/UserTransaction にバインドされます。
- 残りのクライアント環境は、weblogic.ClientDeployer クラスによって作成された client.properties ファイルから java:comp/env/ にバインドされます。バインドができない、または不完全な場合、weblogic.j2eeclient.Main はエラー メッセージを送出します。
- アプリケーション デプロイメント ファイルの <res-auth> タグは、現在は無視されるので Application と入力します。現在、フォームベースの認証はサポートしていません。

注意： デプロイメントの詳細については、第 5 章「WebLogic Server デプロイメント」を参照してください。

Apache Ant による J2EE アプリケーションのパッケージ化

この節の各トピックでは、拡張可能 Java ベース ツール、Apache Ant を使用して J2EE アプリケーションを構築およびパッケージ化する方法を説明します。Ant は、make コマンドと似ていますが、Java アプリケーションを構築するためのツールです。Ant ライブラリは WebLogic Server に付属していて、製品パッケージから Java アプリケーションを簡単に構築できるようになっています。

開発者は、eXtensible Markup Language (XML) を使用して、Ant によるスクリプトを作成します。XML タグは構築対象、対象間の依存性、および対象を構築するための実行タスクを定義します。

Ant 機能の詳しい説明については、以下のサイト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 には、Java ソース ファイルをコンパイルする `javac` タスクがあります。現在のディレクトリのすべての Java ファイルを `classes` ディレクトリにコンパイルする例を以下に示します。

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

`javac` タスクに関するすべてのオプションを確認したい場合は、Apache Ant のオンライン マニュアルを参照してください。

WebLogic Server コンパイラの実行

Ant から任意の Java プログラムを実行するには、カスタム Ant タスクを作成するか、`java` タスクを使用してそのままプログラムを実行します。`ejbc` または `rmic` などのタスクは、以下に示す `java` タスクによって実行されます。

コード リスト 4-1 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>
```

上記の例では、fork システム呼び出しを使用して、ejbc を実行する Java プロセスを作成します。この例では、system プロパティを指定して weblogic.home を定義し、arg タグでコマンド ライン引数を指定しています。呼び出し元である Java プロセスのクラスパスは、classpath タグで指定します。

J2EE デプロイメント ユニットのパッケージ化

前述のように、J2EE アプリケーションは、そのコンポーネント タイプによって以下のような特定のファイル拡張子の付いた JAR ファイルにパッケージ化されます。

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

これらのコンポーネントは、J2EE 仕様に従って構造化されます。標準 XML デプロイメント記述子だけでなく、コンポーネントも WebLogic Server 固有の XML デプロイメント記述子によってパッケージ化されます。

Ant が提供するタスクによって、JAR ファイルの構築が容易になります。JAR コマンドの機能に加えて、Ant では、EAR ファイルと WAR ファイルを構築する特別なタスクも提供しています。Ant を使用すれば、パス名を JAR アーカイブに表示されるとおりに指定でき、ファイル システムの元のパスと異なるパスでもかまいません。この機能は、デプロイメント記述子 (この中で J2EE が指定するアーカイブ内の正確な場所は、ソース ツリー内の場所とは一致しない場合がある) のパッケージ化に便利です。関連情報については、ZipFileSet コマンドに関する Apache Ant オンライン マニュアルを参照してください。

WAR タスクの例を以下に示します。

コード リスト 4-2 WAR タスクの例

```
<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手順は以下のとおりです。

1. webxml パラメータを使用して標準 XML デプロイメント記述子を指定します。
2. war タスクにより、XML デプロイメント記述子が WAR アーカイブ WEB-INF/web.xml の標準名に自動的にマップされます。
3. Apache Ant により、manifest パラメータで指定された manifest ファイルが、標準名 META-INF/MANIFEST.MF の下に格納されます。
4. Apache Ant ZipFileSet コマンドを使用して、WEB-INF ディレクトリに格納される一連のファイル (この場合は、WebLogic Server 固有のデプロイメント記述子、weblogic.xml だけ) を定義します。
5. 次に、ZipFileSet コマンドを使用し、すべての画像を images ディレクトリにパッケージ化します。
6. classes タグは、WEB-INF/classes ディレクトリのサーブレット クラスをパッケージ化します。
7. 最後に、すべての .jsp ファイルと .html ファイルを現在のディレクトリからアーカイブに追加します。

構造が WAR ファイルと同じディレクトリ内にファイルをステージングし、そのディレクトリから JAR ファイルを作成しても、同じ結果が得られます。Ant JAR タスクの特別機能を使用することにより、ファイルを特定のディレクトリ階層にコピーする手間が省けます。

以下の例では、Web アプリケーションと EJB を 1 つずつ構築し、それらをまとめて EAR ファイルにパッケージ化しています。

コード リスト 4-3 パッケージ化の例

```
<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 では、Ant を実行する簡単なスクリプトを server/bin ディレクトリで提供しています。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 フラグを使用すればこの指定はオーバーライドできます。以下のコマンドで、上記のビルド スクリプトを使用してアプリケーションを構築し、デプロイできます。

ant -f yourbuildscript.xml

5 WebLogic Server デプロイメント

このリリースの WebLogic Server では、新しいデプロイメント プロトコルとして 2 フェーズ デプロイメントが導入されました。これを利用すると、サーバ間、特にクラスタ化されているサーバ間でデプロイメント ステートの一貫性を容易に維持できます。2 ページの「2 フェーズ デプロイメント」を参照してください。

アプリケーションは、WebLogic Server Administration Console、`weblogic.Deployer` ユーティリティ、WebLogic Builder、または自動デプロイメントを使用してデプロイできます。これらのデプロイメント ツールについては、「デプロイメント ツールおよび手順」で説明されています。

以下の節で WebLogic Server のデプロイメントについて説明します。

- 2 フェーズ デプロイメント
- リソースおよびアプリケーションのデプロイメント順序
- アプリケーションのステージング
- デプロイメント ツールおよび手順
- デプロイメント補足ドキュメント

2 フェーズ デプロイメント

新しい 2 フェーズ デプロイメント プロトコルを利用すると、ドメインの一貫性を容易に維持できます。旧バージョンの WebLogic Server では、アプリケーションのデプロイ時に、管理サーバによってアプリケーション ファイルのコピーがすべての対象サーバに送信され、その後、対象サーバにアプリケーションがロードされました。したがって、対象サーバへのデプロイメントが一部でも失敗すると、対象サーバ間のデプロイメント ステートにおける一貫性が失われました。

最新リリースの WebLogic Server では、まず、すべての対象サーバに対するアプリケーションが準備され、次に別フェーズでアプリケーションがアクティブ化されます。準備フェーズまたはアクティブ化フェーズでアプリケーションのデプロイメントが失敗すると、クラスタへのデプロイメントが失敗します。

従来の WebLogic Server デプロイメント プロトコルの使用については、「WebLogic Server 6.x のデプロイメント プロトコルの使用」を参照してください。

新しいデプロイメント プロトコルでは、デプロイされるアプリケーションに関して以下の新機能がサポートされています。

- **クラスタのデプロイメント ステートの一貫性。**クラスタを対象としたアプリケーションがクラスタ メンバーのいずれかに対して準備フェーズで失敗した場合、アクティブ化フェーズでは、どのクラスタ メンバーに対してもアプリケーションはアクティブ化されません。このため、クラスタの均一性が保証されます。
- **アプリケーションの順序設定。**サーバの起動時に、アプリケーションのアクティブ化順序を設定します。「リソースおよびアプリケーションのデプロイメント順序」を参照してください。
- **アプリケーション スコープのコンフィグレーション。**特定のリソースをアプリケーション用にコンフィグレーションし、スコープとして設定できます。これには、接続プール、セキュリティ レルム、および XML 関連リソースがあります。「アプリケーション スコーピングの概要」を参照してください。
- **再デプロイメントの改善。**再デプロイする前にアンデプロイを行う必要はありません。「Administration Console によるアプリケーションの更新」、「アーカイブされたアプリケーションのアンデプロイメントと再デプロイメント」、および「展開形式によるアプリケーションの再デプロイメント」を参照してください。

注意： 再デプロイ中には、アプリケーションがクライアントから使用できなくなります。そのため、プロダクション環境では再デプロイメントを使用しないことをお勧めします。

- **API の改善。**単純な API では、コンフィグレーションと実際のデプロイメント処理とが別になっています。デプロイメントが要求されると、この API によって必要なコンフィグレーション (MBean) が自動的に作成されます。また、デプロイメント処理は MBean 自体では行われないので、ユーザは、デプロイメント要求が開始されるまで、デプロイ済みのアプリケーションに影響を与えることなく、コンフィグレーション (対象リストなど) を変更できます。「デプロイメント管理 API」に加えて、`weblogic.management.deploy` で API ドキュメントも参照してください。
- **デプロイメント ステータス。**デプロイメントの進行状況の追跡が、特にデプロイ先が複数の場合に容易になりました。「weblogic.Deployer ユーティリティの使用例」、および Administration Console オンライン ヘルプの「タスク」を参照してください。

管理サーバの再起動

管理サーバを停止および再起動すると、保留中のデプロイメント要求が取り消されます。クラスタへのデプロイ中に、クラスタ内の対象サーバのいずれかが停止している場合、管理サーバを再起動すると、その対象サーバは復帰したときにデプロイメント要求を受け取れなくなります。

準備フェーズとアクティブ化フェーズ

2 フェーズ モデルでは、アプリケーションを対象サーバにデプロイする前に準備フェーズの成功を確認することで、クラスタ内でデプロイメント ステートの矛盾が発生しにくくなっています。準備フェーズで失敗したデプロイメントは、アクティブ化フェーズに進みません。

準備フェーズ

第 1 フェーズであるデプロイメントの準備フェーズでは、ファイルを配布またはコピーし、アプリケーションとそのコンポーネントを検証し、エラー チェックを実行してアクティブ化に備えます。準備フェーズの目的は、作成されたアプリケーションおよびそのコンポーネントがデプロイ可能な状態にあるか確認することです。

アクティブ化フェーズ

第 2 フェーズであるアクティブ化フェーズでは、対応するサーバのサブシステムを使用してアプリケーションとコンポーネントの実際のデプロイメントまたはアクティブ化が行われます。アプリケーションは、アクティブ化フェーズを経てクライアントに対して使用可能になります。

リソースおよびアプリケーションのデプロイメント順序

デフォルトでは、WebLogic Server は、サーバレベルのリソース (JDBC に続いて JMS) をデプロイしてからアプリケーションおよびスタンドアロン モジュールをデプロイします (その後に起動クラスが実行される)。起動クラスの実行の順序はコンフィグレーション可能です (5-5 ページの「起動クラスの実行とデプロイメントの順序設定」を参照)。

アプリケーションの順序の設定

アプリケーションは、コネクタ、EJB、Web アプリケーションの順にデプロイされます。WebLogic Server 7.0 では、アプリケーションについてロード順を選択できます。「Application」の `ApplicationMBean LoadOrder` 属性を参照してください。

アプリケーション コンポーネントの順序の設定

アプリケーションが EAR の場合、`application.xml` デプロイメント記述子に宣言されている順序で、個々のコンポーネントがロードされます。「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ファイル」を参照してください。

起動クラスの実行とデプロイメントの順序設定

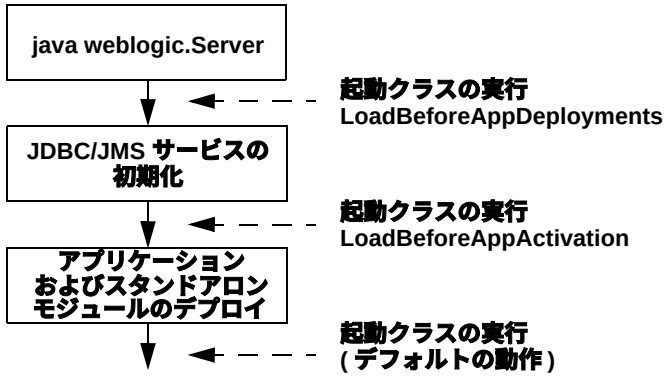
デフォルトでは、サーバで JMS サービスと JDBC サービスが初期化され、アプリケーションおよびスタンドアロン モジュールがデプロイされた後に WebLogic Server の起動クラスが実行されます。

JMS サービスと JDBC サービスが使用可能になった後、アプリケーションとモジュールがアクティブ化される前に起動タスクを実行する場合は、Administration Console の [Run Before Application Deployments アプリケーション デプロイメントの前に実行] オプションを選択します (または `StartupClassMBean` の `LoadBeforeAppActivation` 属性を「true」に設定する)。

JMS サービスと JDBC サービスが使用可能になる前に起動タスクを実行する場合は、Administration Console の [Run Before Application ActivationsRun Before Application Activations] オプションを選択します (または `StartupClassMBean` の `LoadBeforeAppDeployments` 属性を「true」に設定する)。不明 UI(2004/03/23)

次の図は、WebLogic Server でいつ起動クラスが実行されるかを簡単に示しています。

図 5-1 起動クラスの実行



詳細については、`StartupClassMBean` の Javadoc を参照してください。

アプリケーションのステージング

このリリースの WebLogic Server では、デプロイを行う前にアプリケーション ファイルをコピーするかどうか、およびコピーする場合のコピー先およびコピー主体を、ステージング モードで指定できます。ステージングとは、デプロイ元となる場所にアプリケーション ファイルをコピーすることです。サーバはそれぞれステージング モードを持っています。デプロイメントの実行後は、アプリケーションのステージング モードは変更できません。ステージング モードをいつ使用するかについては、「アプリケーション デプロイメントのベスト プラクティス」を参照してください。

ステージング モード

使用可能なステージング モードは以下のとおりです。

- nostage: アプリケーション ファイルを別の場所にコピーしません。

  nostage モードの場合、サーバはサーバにデプロイされたアプリケーションをソース ディレクトリから直接実行します。このモードでは、Web アプリケーション コンテナが JSP およびサーブレットに対する変更を検出します。

- stage : アプリケーション ファイルをデプロイメント対象のサーバにコピーします。

  stage モードでは、デプロイメント操作を実行すると、管理サーバは、ソースファイルを対象サーバのステージング ディレクトリに開きます。対象サーバは、このディレクトリからアプリケーションを初期化して実行します。

- external_stage : WebLogic Server ではなくユーザが、デプロイメントの前にアプリケーション ファイルをサーバのステージング ディレクトリにコピーします。

  デプロイメントは、各対象サーバのステージング ディレクトリのアプリケーション名と同じ名前を持つディレクトリにコピーされます。

  external stage モードは、外部エンティティがファイルの配布先として想定しているステージング ディレクトリからアプリケーションが起動することを意味します。このモードは、環境をサードパーティ ツールで管理している場合に便利です。

注意 : nostage または external_stage のいずれかを使用するには、デプロイするファイルが管理サーバ マシンからアクセス可能でなければなりません。管理サーバ マシンにファイルをコピーするか、またはファイルを管理サーバ マシンから使用可能な共有ディレクトリに入れます。

ステージング モードでは次のようにデフォルト設定されます。

- 管理対象サーバの場合、ステージング モードはデフォルトで stage となります。つまり、デフォルトのステージング動作ではアプリケーション ファイルが対象の管理対象サーバにコピーされます。
- 管理サーバの場合、ステージング モードはデフォルトで nostage となります。つまり、デフォルトのステージング動作では指定したソースの場所からのデプロイが行われます。

次の表では、ステージング属性およびパスの設定がアプリケーションのデプロイメントに与える影響について説明します。

表 5-1 デプロイメントのステージング モード

ステージング モード	ステージング ディレクトリ	管理サーバのデプロイメント動作	管理対象サーバのデプロイメント動作
stage	絶対パス、相対パス	アプリケーション ファイルは、サーバ ステージング ディレクトリ内のアプリケーション デプロイメントに基づいて付けられた名前のディレクトリにコピーされ、そのディレクトリ (たとえば、`server/stage/myapp/app.ear`) からアクティブ化される。 ステージング パスが相対パスの場合、パスはサーバのルート ディレクトリを基準にしている。	管理サーバと同じ。
nostage	絶対パス、相対パス	ステージング パスは無視され、ファイルはコピーされない。ファイルは、管理サーバ上の場所または管理サーバからアクセス可能な共有ディレクトリからデプロイされる。	管理サーバと同じ (デプロイするには、ソース ファイルが管理対象サーバ マシンからアクセス可能でなければならない)。

表 5-1 デプロイメントのステージング モード

ステージング モード	ステージング ディレクトリ	管理サーバのデプロイメント動作	管理対象サーバのデプロイメント動作
external_stage	絶対パス、相対パス	ファイルはコピーされないが、デプロイメント ファイルはサーバ ステージング ディレクトリ内のデプロイメントに基づいて付けられた名前のサブディレクトリにあるものと見なされ、そこからロードされる。たとえば、デプロイメント名「myextapp」を使用してアプリケーションをデプロイし、サーバ ステージング ディレクトリが `.\myserver\stage` の場合は、デプロイメント ファイルが `.\myserver\stage\myextapp` で使用可能であることをデプロイメント前に確認しておく必要がある。 ステージング パスが相対パスの場合、パスはサーバのルート ディレクトリを基準にしている。	管理サーバと同じ (デプロイメント ファイルが管理対象サーバのステージング ディレクトリの適切なサブディレクトリにコピーされていることを確認する必要がある)。

ステージング モードおよびディレクトリのコンフィグレーション

デフォルトでは、アプリケーションを管理対象サーバにデプロイすると、アプリケーションはデプロイ先サーバのステージング エリア (`ServerMBean.StagingDirectoryName`) にステージングされ、そこからデプロイされます。ステージングは、`ServerMBean.StagingMode` 属性または `ApplicationMBean.StagingMode` 属性を使用して無効にできます。`ServerMBean.StagingMode` 属性は、そのサーバにデプロイされているすべてのアプリケーションに適用されます。この属性は `ApplicationMBean.StagingMode` によってオーバライドされます。

ステージング モードをいつ使用するかについては、「アプリケーション デプロイメントのベスト プラクティス」を参照してください。

ここで取り上げた属性の Javadoc については、WebLogic クラスの Javadoc を参照してください。

ステージングのシナリオ

以下で説明する特定の共通タスクを実行するようにシステムをコンフィグレーションできます。

アプリケーションをソースの場所からデプロイする

アプリケーションまたは特定サーバの `StagingMethod` 属性を `nostage` にコンフィグレーションします。アプリケーションでこの属性をコンフィグレーションする場合、この設定は `weblogic.Deployer` ツールを使用してサーバでアプリケーションがコンフィグレーションされるときに行います。

このモードは、単一サーバで漸進的な開発を行う場合に便利です。また、複数のサーバがアプリケーションの同じコピーを使用する共有ディスク環境でも役立ちます。

アプリケーションを既知のステージング エリアからデプロイする

既知のディレクトリを指すように、各サーバの `StagingDirName` 属性をコンフィグレーションします。実際の EAR、JAR、WAR、および RAR ファイルは、そのディレクトリ内で対象アプリケーション名を持つディレクトリに配置します。

アプリケーション ファイルを管理対象サーバに配布する

`StagingMode` 属性を `stage` に設定した場合、WebLogic Server では、ソースからステージング ディレクトリにファイルがコピーされます。`stage` モードを使用してデプロイするには、次の手順に従います。

1. `stage` モードを使用するように管理対象サーバをコンフィグレーションし、各サーバが使用するステージング ディレクトリを指定します。

2. デプロイするファイルがドメインの管理サーバからアクセス可能であることを確認します。管理サーバ マシンにファイルをコピーするか、またはファイルを管理サーバ マシンから使用可能な共有ディレクトリに入れます。
3. Administration Console を使用して管理対象サーバにファイルをデプロイします。

external_stage モードを使用してアプリケーションをデプロイする

`external_stage` モードでは、デプロイメント ファイルを管理対象サーバに手動でコピーするか、またはファイルをコピーするアプリケーションあるいはスクリプトを用意する必要があります。`external_stage` モードを使用して、アプリケーションを管理対象サーバにデプロイするには、次の手順に従います。

1. `stage` モードを使用するように管理対象サーバをコンフィグレーションし、各サーバが使用するステージング ディレクトリを指定します。
2. 管理対象サーバのステージング エリアで、使用するデプロイメント名 (「mywar」など) を持つサブディレクトリを作成し、デプロイメント ファイルをそのサブディレクトリにコピーします。
3. デプロイするファイルがドメインの管理サーバからアクセス可能であることを確認します。管理サーバ マシンにファイルをコピーするか、またはファイルを管理サーバ マシンから使用可能な共有ディレクトリに入れます。
4. Administration Console で、同じデプロイメント名 (「mywar」) を使用して管理対象サーバにファイルをデプロイします。管理対象サーバは、上記の手順 2 でコピーしたデプロイメント ファイルを使用してデプロイします。

デプロイメント ツールおよび手順

WebLogic Server のデプロイメント ツールは、「デプロイメント管理 API」で説明したデプロイメント API へのインタフェースを提供します。

ここで説明するデプロイメント手順では、正しいディレクトリ構造を使用し、適切なデプロイメント記述子を含む、正常に機能する J2EE アプリケーションが作成されていることを前提にしています。デプロイメント記述子 (XML タグを使用して書式化されたテキスト ファイル) には、アプリケーションのディレクトリまたはアーカイブの内容を記述します。J2EE 仕様では、J2EE アプリケーションおよびそのコンポーネント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、アプリケーションおよびそのコンポーネントを WebLogic Server 環境にデプロイするための 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。詳細については、「XML デプロイメント記述子」を参照してください。

WebLogic Server デプロイメント ツールの一覧を以下に示します。

- weblogic.Deployer ユーティリティ
- wldeploy Ant タスク
- WebLogic Server Administration Console
- WebLogic Builder
- 自動デプロイメント (開発モードのみ)

weblogic.Deployer ユーティリティ

`weblogic.Deployer` ユーティリティは WebLogic Server 7.0 の新機能で、非推奨となった以前の `weblogic.deploy` ユーティリティに代わるものです。`weblogic.Deployer` ユーティリティは、WebLogic Server デプロイメント API にコマンドライン インタフェースを提供する、Java ベースの開発ツールです。このユーティリティは、コマンドライン、シェル スクリプト、または Java 以外の自動化された環境からデプロイメントを開始する必要がある管理者および開発者向けに開発されました。

この節では、`weblogic.Deployer` ユーティリティを使って以下のタスクを実行する方法について説明します。

- 新しいアプリケーションをデプロイする
- 新しいアプリケーションをクラスタにデプロイする
- アプリケーション全体を再デプロイする

- EAR に新しく追加されたモジュールをデプロイする
- 展開されたアプリケーションの一部を再デプロイ (更新) する
- すべてのアクティブな対象でアプリケーションを非アクティブ化する (使用不可にする)
- 非アクティブ化したアプリケーションを再アクティブ化する
- すべての対象サーバからアプリケーションを削除する
- デプロイメント タスクを取り消す
- すべてのデプロイメント タスクの表示
- 単一のサーバにアプリケーションをデプロイまたは再デプロイする
- アプリケーションを追加サーバにデプロイする

weblogic.Deployer ユーティリティによるデプロイ作業

`weblogic.Deployer` ユーティリティを使用してアプリケーションまたはそのコンポーネントをデプロイする手順は次のとおりです。

1. WebLogic Server クラスをシステムの `CLASSPATH` に入れ、JDK が利用可能になるようにローカル環境を設定します。`CLASSPATH` の設定には、サーバの `/bin` ディレクトリにある `setenv` スクリプトを使用できます。
2. 次のコマンド構文を使用します。

   ```
   % java weblogic.Deployer [options]
   [-activate|-deactivate|-remove|-cancel|-list] [files]
   ```

また、デプロイ (再デプロイ、アンデプロイ、非アクティブ化、準備解除、または削除) の対象となるアーカイブ内の特定の `-files` を一覧表示することもできます。ファイル リストには、ファイル名およびアプリケーションのルートを基準にしたディレクトリを含めることができます。ディレクトリを指定すると、サブツリー全体がデプロイまたは再デプロイされます。

weblogic.Deployer のアクションおよびオプション

表 5-2 weblogic.Deployer のアクション

アクション	説明
activate	-name で指定されたアプリケーションを -targets で指定されたサーバにデプロイまたは再デプロイする。
cancel	-id で識別されるタスクがまだ完了していない場合、そのタスクの取り消しを試みる。
deactivate	対象サーバのアプリケーションを非アクティブ化する。非アクティブ化されると、デプロイ済みのコンポーネントがサスペンドされ、ステージングされていたデータはそのまま残り、その後の再アクティブ化を待つ (このコマンドは 2 フェーズ デプロイメント プロトコルでのみ機能する)。
delete_files	ファイル リストに指定されたファイルを削除し、アプリケーションはアクティブ化したままにしておく。これは、アーカイブされていないアプリケーションにのみ適用される。対象のサーバは必ず指定する。
deploy	-activate の便利なエリアス。
examples	ツールの使用例を表示する。
help	ヘルプ メッセージを出力する。
list	-id で識別されるタスクのステータスを表示する。
remove	アプリケーションおよびステージング済みのデータを対象サーバから物理的に削除する。コンポーネントは非アクティブ化され、対象はアプリケーション コンフィグレーションから削除される。アプリケーションを完全に削除すると、関連する MBean もシステム コンフィグレーションから削除される (このコマンドは 2 フェーズ デプロイメント モデルでのみ機能する)。
undeploy	-unprepare の便利なエリアス。

アクション	説明
unprepare	対象サーバ上で -name によって識別されるアプリケーションのクラスを非アクティブ化し、アンロードする。ステージングされたアプリケーション ファイルは、編集またはすぐに再ロードできるような状態で残す。
upload	指定されたソース ファイルを管理サーバに転送する。このオプションは、リモート システムを使用しているときに、リモート システムに常駐するアプリケーションをデプロイする場合に使用する。アプリケーション ファイルは、指定された対象サーバに配布される前に WebLogic Server 管理サーバにアップロードされる。
version	バージョン情報を出力する。

weblogic.Deployer のオプションは以下のとおりです。

表 5-3 weblogic.Deployer のオプション

オプション	説明
adminurl	https://<server>:<port> が管理サーバの URL。デフォルトは http://localhost:7001。
debug	出力ログのデバッグ メッセージをオンにする。
enforceClusterConstraints	クラスタの 1 つまたは複数のメンバーが使用できない場合に、クラスタへのデプロイメントが成功するか、または失敗するかを指定する。このオプションを「true」に設定すると、デプロイメントはクラスタのすべてのメンバーが使用可能な場合にのみ成功するようになる。
external_stage	ユーザが自身で、またはサードパーティ ツールを使用してサーバのステージング エリアにアプリケーションをコピーすることを示す。このオプションを指定すると、WLS は (対象サーバの) StagingDirectoryName/applicationName の下でアプリケーションを探す。

オプション	説明
id	タスク識別子 -id はデプロイメント タスクでユニークな識別子である。-id は、-activate、-deactivate、または -remove コマンドで指定でき、-cancel または -list の引数として使用できる。-id は、既存のデプロイメント タスクすべてを通してユニークでなければならない。指定しない場合、システムが -id を自動的に生成する。
name	アプリケーションの -name には、デプロイされるアプリケーションの名前を指定する。この名前は、既存ないしコンフィグレーション済みのアプリケーション名、または新しいコンフィグレーション作成時に使用する名前でも可。
nostage	アプリケーションをステージングしないで、代わりに、-source オプションで指定した現在の場所からアプリケーションをデプロイする。 デフォルトは、管理サーバが nostage、管理対象サーバが stage。
nowait	アクションが開始されると、ツールがタスク ID を出力して終了する。これは複数のタスクを開始し、-list アクションを使用して後でモニタするために使用する。
output	raw または formatted を指定して、weblogic.Deployer 出力メッセージの表示方法を制御する。いずれの出力タイプでも同じ情報が表示されるが、raw 出力には埋め込みタグが含まれない。デフォルトでは、weblogic.Deployer は raw 出力を表示する。
password	パスワードをコマンドラインで指定する。パスワードを指定しないと、パスワードの入力が求められる。
remote	weblogic.Deployer が管理サーバと同じマシンで動作していないこと、およびソース パスがリモート サーバ上のパスなので変更せずに渡す必要があることを示す。

オプション	説明
source	デプロイするアーカイブ、ファイルまたはディレクトリの場所を指定する。このオプションは、アプリケーション パスの設定に使用する。source オプションはルート ディレクトリまたはデプロイされるアーカイブを参照する必要がある。upload コマンドと共に使用すると、ソース パスはカレント ディレクトリに対する相対パスになる。upload を使用しない場合は、管理サーバのルート ディレクトリ、すなわち config.xml ファイルのあるディレクトリに対する相対パスとなる。
stage	デプロイメント前にアプリケーションを対象サーバのステージング ディレクトリにコピーする必要があることを示す。デフォルトは、管理サーバが nostage、管理対象サーバが stage。アプリケーションの作成時に stagingMethod 属性を設定すると、アプリケーションは常にステージングされる。この値は対象サーバの stagingMethod 属性をオーバーライドする。
targets	対象サーバ名とクラスタ名のカンマ区切りのリストを表示する (<server 1>,...<component>@<server N>)。各対象は J2EE コンポーネント名を使用して修飾できる。これによりアーカイブの各種コンポーネントをさまざまなサーバにデプロイできる。現在デプロイされているアプリケーションの場合、デフォルトは現在のすべての対象。新しいアプリケーションの場合、デフォルトで管理サーバにデプロイされる。
timeout	秒数。デプロイメント タスクの完了までの最長時間 (単位 : 秒) を指定する。時間が経過すると、weblogic.Deployer はデプロイメントの現在のステータスを出力して終了する。
user	ユーザ名。
userconfigfile	管理ユーザ名およびパスワードに使用するユーザ コンフィグレーション ファイルの場所を指定する。このオプションは、自動化されたスクリプト、パスワードを画面上に表示しないようにする場合、または ps などのプロセスレベルのユーティリティにおいて、-user および -password オプションの代わりに使用する。-userconfigfile オプションを指定する前に、weblogic.Admin STOREUSERCONFIG コマンドを使用してファイルを生成しておく必要がある。

オプション	説明
userkeyfile	ユーザ コンフィグレーション ファイル (-userconfigfile オプション) に格納されたユーザ名およびパスワード情報の暗号化および解読に使用するユーザ キー ファイルの場所を指定する。-userkeyfile オプションを指定する前に、weblogic.Admin STOREUSERCONFIG コマンドを使用してファイルを生成しておく必要がある。
verbose	追加の進行状況メッセージを表示する。

weblogic.Deployer ユーティリティの使用例

weblogic.Deployer ユーティリティの使用例を以下に示します。

新しいアプリケーションをデプロイする

```
java weblogic.Deployer -adminurl http://admin:7001 -name app
-source /myapp/app.ear -targets server1,server2 -activate
```

新しいアプリケーションをクラスタにデプロイする

```
java weblogic.Deployer -adminurl http://admin:7001 -name app
-source /myapp/app.ear -targets cluster1 -activate
-enforceClusterConstraints
```

アプリケーション全体を再デプロイする

```
java weblogic.Deployer -source /myapp/app.ear -adminurl
http://admin:7001 -name app -activate
```

注意: -source、つまり更新ファイルのリストを指定しない場合、アプリケーションに変更がないものと見なされるので、この処理は何の効果もありません。

Web アプリケーションを再デプロイするには、コマンドラインで -source および -target オプションの両方を指定する必要があります。

EAR に新しく追加されたモジュールをデプロイする

デプロイ済みのアプリケーション myapp.ear にモジュール newmodule.war を追加し、application.xml ファイルでそのモジュールを更新した場合は、次のコマンドを使用して newmodule.war を myapp.ear にデプロイできます。

```
java weblogic.Deployer -username myname -password mypassword
-name myapp.ear -activate -targets newmodule.war@myserver
-source /myapp/myapp.ear
```

このコマンドは、アプリケーション内のその他のモジュールは再デプロイせずに、新しいモジュールをデプロイします。

展開されたアプリケーションの一部を再デプロイ (更新) する

```
java weblogic.Deployer -adminurl http://admin:7001 -name app
-activate jsps/login.jsp
```

ここで、jsps は、展開された Web アプリケーションの最上位のディレクトリです。部分的な再デプロイメントは展開された WAR ファイルに関してのみサポートされています。パスは、元々デプロイされているアプリケーションのルートに対する相対パスです。たとえば、作業ディレクトリで login.jsp ファイルを変更した場合、weblogic.Deployer コマンドを入力する前に、更新したファイルを、展開されたアプリケーションの適切なソース ディレクトリにコピーする必要があります。

すべてのアクティブな対象でアプリケーションを非アクティブ化する (使用不可にする)

```
java weblogic.Deployer -adminurl http://admin:7001 -name app
-deactivate
```

非アクティブ化したアプリケーションを再アクティブ化する

```
 java weblogic.Deployer -adminurl http://7001 -name app
-activate
```

すべての対象サーバからアプリケーションを削除する

```
java weblogic.Deployer -adminurl http://admin:7001 -name app
-targets server -remove
```

デプロイメント タスクを取り消す

```
java weblogic.Deployer -adminurl http://admin:7001 -cancel -id
tag
```

すべてのデプロイメント タスクの表示

```
java weblogic.Deployer -adminurl http://admin:7001 -list
```

単一のサーバにアプリケーションをデプロイまたは再デプロイする

```
java weblogic.Deployer -activate -name ArchivedEarJar -source
C:/MyApps/JarEar.ear –target server1
```

アプリケーションを追加サーバにデプロイする

```
java weblogic.Deployer -activate -name ArchivedEarJar –target
server2
```

wldeploy Ant タスク

wldeploy Ant タスクを使用すると、Ant .xml ファイルで指定した属性を使用して weblogic.Deployer の機能を実行できます。他の WebLogic Server Ant タスクと一緒に wldeploy を使用して、次のような単一の Ant ビルド スクリプトを作成できます。

- wlserver および wlconfig Ant タスクを使用して、新しい WebLogic Server ドメインを作成、起動、およびコンフィグレーションする。
- wldeploy Ant タスクを使用して、コンパイル済みアプリケーションを、新しく作成されたドメインにデプロイする。

wlserver および wlconfig の詳細については、『管理者ガイド』の「Ant タスクを使用した WebLogic Server ドメインのコンフィグレーション」を参照してください。

注意： WebLogic Server Ant タスクは、1.5 より前のバージョンの Ant とは互換性がありません。また、WebLogic Server に含まれていないバージョンの Ant を使用する場合は、5-21 ページの「wldeploy を使用する基本手順」の説明に従って、`build.xml` ファイルで `wldeploy` タスク定義を指定する必要があります。

wldeploy を使用する基本手順

`wldeploy` Ant タスクを使用するには、次の手順に従います。

1. 環境を設定します。

   Windows NT では、`WL_HOME\server\bin` ディレクトリにある `setWLSEnv.cmd` コマンドを実行します。`WL_HOME` は WebLogic Platform がインストールされている最上位ディレクトリです。

   UNIX では、`WL_HOME/server/bin` ディレクトリにある `setWLSEnv.sh` コマンドを実行します。`WL_HOME` は WebLogic Platform がインストールされている最上位ディレクトリです。

2. ステージング ディレクトリで、Ant ビルド ファイル (デフォルトは `build.xml`) を作成します。WebLogic Server と一緒にインストールされているものとは異なる Ant を使用する場合は、最初に `wldeploy` Ant タスク定義を定義します。

   ```
   <taskdef name="wldeploy"
   classname="weblogic.ant.taskdefs.management.WLDeploy"/>
   ```

3. 必要に応じて、`wlserver` および `wlconfig` タスクのタスク定義と呼び出しをビルド スクリプトに追加して、新しい WebLogic Server ドメインを作成および起動します。`wlserver` および `wlconfig` の詳細については、『WebLogic Server コマンド リファレンス』の「Ant タスクを使用した WebLogic Server ドメインのコンフィグレーション」を参照してください。

4. `wldeploy` の呼び出しを追加して、1 つまたは複数の WebLogic Server インスタンスまたはクラスタにアプリケーションをデプロイします。5-22 ページの「wldeploy のサンプルの build.xml ファイル」および 5-22 ページの「wldeploy Ant タスクのリファレンス」を参照してください。

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

```
prompt> ant
```

wldeploy のサンプルの build.xml ファイル

次の出力は、アプリケーションを単一の WebLogic Server インスタンスにデプロイする wldeploy ターゲットを示しています。

```
<target name="deploy">
   <wldeploy action="activate"
      source="${build}/ejb11_basic_statelessSession.ear"
name="ejbapp"
      user="a" password="a" verbose="true"
adminurl="t3://localhost:7001"
      debug="true" targets="myserver"/>
</target>
```

wldeploy Ant タスクのリファレンス

次の表では、wldeploy Ant タスクの属性について説明します。さまざまな用語の定義については、5-14 ページの「weblogic.Deployer のアクションおよびオプション」を参照してください。

表 5-4 wldeploy Ant タスクの属性

属性	説明	データ型	必須 / 省略可能
action	実行するデプロイメント アクション。有効な値は、activate、deactivate、remove、cancel、list および unprepare。	String	省略可能
adminurl	管理サーバの URL。	String	省略可能
debug	wldeploy デバッグ メッセージを有効にする。	boolean	省略可能
id	ステータスの取得やデプロイメントの取り消しに使用される ID。	String	省略可能
name	デプロイ済みのアプリケーションのデプロイメント名。	String	省略可能
nostage	デプロイメントで nostage デプロイメント モードを使用するかどうかを指定する。	boolean	省略可能

表 5-4 wldeploy Ant タスクの属性

属性	説明	データ型	必須 / 省略可能
nowait	wldeploy が (バックグラウンド タスクとしてデプロイすることによって) デプロイメント呼び出しの後直ちに復帰するかどうかを指定する。	boolean	省略可能
user	管理ユーザ名。	String	省略可能
password	管理パスワード。 ビルド ファイルまたは ps などのプロセス ユーティリティでプレーン テキストのパスワードを表示しないようにするには、まず、`weblogic.Admin STOREUSERCONFIG` コマンドを使用して、有効なユーザ名と暗号化されたパスワードをコンフィグレーション ファイルに格納する。次に、Ant ビルド ファイルで `username` および `password` 属性の両方を省略する。この属性を省略した場合、`wldeploy` はデフォルトのコンフィグレーション ファイルから取得した値を使用してログインしようとする。 デフォルト以外のコンフィグレーション ファイルおよびキー ファイルからユーザ名とパスワードを取得する場合は、`wldeploy` と一緒に `userconfigfile` および `userkeyfile` 属性を使用する。	String	省略可能
remote	サーバが別のマシンに配置されているかどうかを指定する。これはファイル名の転送方法に影響する。	boolean	省略可能
source	デプロイするソース ファイル。	ファイル	省略可能
targets	デプロイ先となる対象サーバのリスト。	String	省略可能
timeout	デプロイメントが正常に終了するまでの最長待ち時間。	int	省略可能

表 5-4 wldeploy Ant タスクの属性

属性	説明	データ型	必須 / 省略可能
userconfigfile	管理ユーザ名およびパスワードを取得するために使用するユーザ コンフィグレーション ファイルの場所を指定する。このオプションは、プレーン テキストのパスワードをインラインで表示しないようにする場合や ps などのプロセスレベルのユーティリティにおいて、ビルド ファイルで user および password 属性の代わりに使用する。「WebLogic Server コマンドライン リファレンス」の「STOREUSERCONFIG」で説明しているように、userconfigfile 属性を指定する前に、weblogic.Admin STOREUSERCONFIG コマンドを使用してファイルを生成しておく必要がある。	ファイル	省略可能
userkeyfile	ユーザ コンフィグレーション ファイル (userconfigfile 属性) に格納されたユーザ名およびパスワード情報の暗号化および解読に使用するユーザ キー ファイルの場所を指定する。userkeyfile 属性を指定する前に、weblogic.Admin STOREUSERCONFIG コマンドを使用してファイルを生成しておく必要がある。	ファイル	省略可能
verbose	wldeploy で冗長出力メッセージを表示するかどうかを指定する。	boolean	省略可能
failonerror	WebLogic Server Ant タスクで使用されるグローバル属性。ビルド中にエラーが発生した場合、タスクが失敗するかどうかを指定する。この属性はデフォルトで true に設定される。	Boolean	省略可能

WebLogic Server Administration Console

この節では、Administration Console によるデプロイメント タスクの実行について説明します。Administration Console は、weblogic.Deployer ユーティリティと同じ機能をサポートしています。この機能によって、デプロイヤは新しいまたは更新したアプリケーションの提出、ステータスのクエリ、および保留中のデプロイメントの削除を行うことができます。

Administration Console を使用した J2EE アプリケーション デプロイメントのコンフィグレーション

WebLogic Server Administration Console を使用して J2EE アプリケーションをコンフィグレーションする手順は次のとおりです。

1. WebLogic Server Administration Console を起動します。
2. 作業を行うドメインを選択します。
3. Administration Console の左ペインで、[Deployments デプロイメント] をクリックします。
4. Administration Console の左ペインで、[Applications アプリケーション] をクリックします。Administration Console の右ペインにテーブルが現れ、デプロイ済みのすべての J2EE アプリケーションが表示されます。
5. [Configure a new Application 新しい Application のコンフィグレーション] オプションを選択します。
6. コンフィグレーションするアーカイブ ファイル (WAR、EAR、RAR、JAR) の場所を確認します。

   注意 : 展開されたアプリケーションやコンポーネント ディレクトリのコンフィグレーションを行うこともできます。WebLogic Server では、指定されたディレクトリおよびその下位のディレクトリにあるコンポーネントがすべてデプロイされることに注意してください。
7. ディレクトリまたはファイルの左の [Select 選択] をクリックして選択し、次の操作に進みます。
8. [Available Servers 使用可能なサーバ] から対象サーバを選択します。
9. アプリケーションの名前を表示されたフィールドに入力します。
10. [Configure and Deploy コンフィグレーションとデプロイ] をクリックします。Administration Console によって [Deploy デプロイ] パネルが表示され、ここに対象 J2EE アプリケーションのデプロイメント ステータスおよびデプロイメント作業が示されます。
11. 使用できるタブで、以下の情報を入力します。
    - [Configuration コンフィグレーション] ― ステージング モードを編集し、デプロイメント順を入力します。

- [Targets 対象] ― デプロイ先のサーバを [Available 選択可] リストから [Chosen 選択済み] リストに移動して、コンフィグレーション済み J2EE アプリケーションの対象サーバを指定します。

注意： 対象を選択しない場合は、アプリケーションのコンフィグレーションだけが行われ、デプロイされません。`mydomain/applications` ディレクトリでこのアプリケーションにアクセスして、後で変更およびデプロイすることができます。

- [Deploy/Undeploy デプロイ / アンデプロイ] ― J2EE アプリケーションをすべてまたは選択した対象にデプロイするか、もしくは、すべてまたは選択した対象からアンデプロイします。J2EE アプリケーションをアンデプロイします。[Deploy/Undeploy デプロイ / アンデプロイ] はトグル オプションです。
- [Monitoring モニタ] ― J2EE アプリケーションのセッション モニタを有効にします。
- [Notes メモ] ― J2EE アプリケーションに関連するメモを入力します。

Administration Console を使用した J2EE アプリケーションのデプロイメント

WebLogic Server Administration Console を使用して J2EE アプリケーションをデプロイする手順は次のとおりです。

1. 左ペインの [Deployments デプロイメント] ノードを展開します。
2. [Applications アプリケーション] ノードを右クリックします。
3. [Configure a New Application 新しい Application のコンフィグレーション] を選択します。
4. コンフィグレーションする展開されたアプリケーションを格納するアーカイブ (WAR、EAR、RAR、JAR) またはディレクトリの場所を確認します。
5. ディレクトリまたはファイルの左の [Select 選択] をクリックして選択し、次の操作に進みます。ディレクトリを指定した場合、WebLogic Server は指定されたディレクトリおよびその下位のディレクトリにあるコンポーネントをすべてデプロイします。
6. [Available Servers 使用可能なサーバ] から対象サーバを選択します。

7. J2EE アプリケーションの名前を表示されたフィールドに入力します。
8. [Configure and Deploy コンフィグレーションとデプロイ] をクリックします。Administration Console によって [Deploy デプロイ] パネルが表示され、ここに対象 J2EE アプリケーションのデプロイメント ステータスおよびデプロイメント作業が示されます。
9. [Deploy デプロイ] ボタンを使って、その J2EE アプリケーションをすべてまたは選択した対象にデプロイするか、あるいは、すべてまたは選択した対象からアンデプロイします。
10. Web ブラウザからリソースにアクセスして、J2EE アプリケーションをテストします。次の URL からリソースにアクセスします。

    http://myServer:myPort/myApp/resource

    各値の説明は次のとおりです。

    - myServer は、WebLogic Server のホスト マシンの名前を表す。
    - myPort は、WebLogic Server がリクエストをリスンしているポート番号を表す。
    - myApp は、J2EE アプリケーション アーカイブ ファイルの名前 (myApp.ear など)、または J2EE アプリケーションを含むディレクトリの名前を表す。
    - resource は、JSP、HTTP サーブレット、HTML ページなどのリソースの名前を表す。

Administration Console を使用したデプロイ済みコンポーネントの参照

デプロイされたコンポーネントを Administration Console で表示する手順は次のとおりです。

1. Administration Console の左パネルから [Deployments デプロイメント] の下の [Component コンポーネント] を選択します。
2. 右ペインの [Deployments デプロイメント] テーブルでデプロイ済みのコンポーネントのリストを参照します。

Administration Console を使用したコンポーネントのアンデプロイメント

WebLogic Server Administration Console を使用してデプロイされているコンポーネントをアンデプロイする手順は次のとおりです。

1. Administration Console の左パネルから [Deployments デプロイメント] の下の [Component コンポーネント] を選択します。
2. コンポーネントの [Deployments デプロイメント] テーブルで、アンデプロイするコンポーネントを選択します。
3. [Apply 適用] をクリックします。

コンポーネントをアンデプロイしても、コンポーネント名は WebLogic Server から削除されません。コンポーネントは、Server セッションが終了するまでアンデプロイされた状態が続きます。ただし、アンデプロイ後にコンポーネントを変更した場合を除きます。サーバを再起動するまで、deploy 引数でデプロイメント名を再使用することはできません。次の節で説明するように、デプロイメントの更新にそのデプロイメント名を再使用できます。

Administration Console によるアプリケーションの更新

デプロイ済みの J2EE アプリケーションを更新する手順は次のとおりです。

1. Administration Console で、[Deployments デプロイメント] をクリックします。
2. [Applications アプリケーション] オプションをクリックします。
3. 表示されたテーブルで、更新するアプリケーションの名前をクリックします。
4. 必要に応じてアプリケーション名とデプロイ ステータスを更新します。

   注意： 再デプロイ中には、アプリケーションがクライアントから使用できなくなります。そのため、プロダクション環境では再デプロイメントを使用しないことをお勧めします。
5. [Apply 適用] をクリックします。

デプロイ済みアプリケーションにモジュールを追加し、追加したモジュールをデプロイする手順は次のとおりです。

1. 上記の手順 1 ～ 5 によって、アプリケーションを再デプロイしてモジュールを追加します。
2. [Deployments デプロイメント | Applications アプリケーション] タブで表示されるテーブルのモジュール名をクリックします。>
3. [Targets 対象] タブを選択します。
4. 使用したいサーバを [Available 選択可] 領域から [Chosen 選択済み] 領域に移動し、[Apply 適用] をクリックします。

WebLogic Builder

WebLogic Builder は、J2EE アプリケーションのデプロイメント記述子を生成および編集するための WebLogic Server ツールです。このツールでは、単一のサーバにアプリケーションをデプロイすることもできます。

「WebLogic Builder」を参照してください。

自動デプロイメント

自動デプロイメントは、管理サーバにアプリケーションを迅速にデプロイするための手段です。自動デプロイメントは、アプリケーションをテストするための単一サーバの開発環境でのみ使用してください。プロダクション環境で使用したり、管理対象サーバにコンポーネントをデプロイするために使用したりすることは避けてください。

自動デプロイメントが有効な場合は、アプリケーションが管理サーバの `\applications` ディレクトリにコピーされると、その管理サーバは新しいアプリケーションの存在を検出し、それを自動的にデプロイします (管理サーバが動作している場合)。アプリケーションを `\applications` ディレクトリにコピーしたときに WebLogic Server が稼働していない場合、そのアプリケーションは WebLogic Server が次に起動したときにデプロイされます。自動デプロイメントは管理サーバに対してのみデプロイします。

注意： Windows NT では厳格なファイル ロック制限により、アプリケーションが展開された場合、アプリケーション内のすべてのコンポーネントも展開されます。すなわち、WebLogic Server では展開されたアプリケーションまたはコンポーネント内に JAR ファイルを入れることはできません。

自動デプロイメントの有効化と無効化

WebLogic Server は、開発環境とプロダクション環境の 2 つの異なるモードで実行できます。開発モードは、アプリケーションをテストする場合に使用します。プロダクション環境への準備が整ったら、プロダクション モードで起動されたサーバにアプリケーションをデプロイします。

開発モードを使用すると、WebLogic Server は domain_name/applications ディレクトリ (domain_name は WebLogic Server のドメイン名) にあるアプリケーションを自動的にデプロイおよび更新します。つまり、開発モードでは自動デプロイを使用することになります。

プロダクション モードでは、自動デプロイメント機能は無効になります。代わりに、WebLogic Server Administration Console または `weblogic.Deployer` ツールを使用する必要があります。

デフォルトでは、WebLogic Server は開発モードで稼働します。サーバのモードを指定するには、以下のいずれかを実行します。

startWebLogic 起動スクリプトを使用する場合は、スクリプトを編集し、STARTMODE 変数を次のように設定します。

`STARTMODE` = false (開発モードを有効にする)

`STARTMODE` = true (プロダクション モードを有効にする)

コマンドラインで直接 `weblogic.Server` コマンドを入力してサーバを起動する場合、次のように -Dweblogic.ProductionModeEnabled オプションを使用します。

`-Dweblogic.ProductionModeEnabled=false` (開発モードを有効にする)

`-Dweblogic.ProductionModeEnabled=true` (プロダクション モードを有効にする)

開発モードおよびプロダクション モードにおける WebLogic Server の起動の詳細については、「WebLogic Server の起動と停止」を参照してください。

アプリケーションの自動デプロイメント

この機能は、開発中にアプリケーションをデプロイする場合に便利です。この機能を使用すると、あらかじめ定義しておいた自動デプロイメント ディレクトリにデプロイメントをコピーするだけで、アプリケーションまたは個々の J2EE モジュールが管理サーバにデプロイされます。このディレクトリはドメイン ディレクトリの下、たとえば、mydomain/applications にあります。

アーカイブされたアプリケーションのアンデプロイメントと再デプロイメント

自動デプロイされたアプリケーションまたはそのコンポーネントは、サーバの稼働時に動的に再デプロイできます。JAR、WAR、または EAR ファイルを動的に再デプロイするには、このファイルの新バージョンを、\applications ディレクトリ内の既存のファイルに上書きコピーするだけです。

この機能を使用すると、開発者はメイクファイルの最後のステップとして \applications ディレクトリへのコピーを追加するだけで、サーバを更新できます。

アプリケーションを \applications ディレクトリから削除すると、そのアプリケーションはアンデプロイされ、コンフィグレーションから削除されます。

展開形式によるアプリケーションの再デプロイメント

展開形式で自動デプロイされたアプリケーションまたはコンポーネントも、動的に再デプロイできます。アプリケーションが展開形式でデプロイされている場合、管理サーバは、展開されたアプリケーションのディレクトリ内で REDEPLOY というファイルを定期的に検索します。このファイルのタイムスタンプが変更されている場合、管理サーバは展開ディレクトリを再デプロイします。

展開されたアプリケーション ディレクトリ内のファイルを更新する場合は、次の手順に従います。

1. 展開されたアプリケーションを初めてデプロイする場合、REDEPLOY という名前の空のファイルを作成し、そのファイルを、デプロイするアプリケーションのタイプに応じて、WEB-INF または META-INF ディレクトリに配置します。

展開されたアプリケーションには、最上位にある META-INF ディレクトリが含まれ、このディレクトリには application.xml ファイルがあります。

展開された Web アプリケーションには、最上位にある WEB-INF ディレクトリが含まれ、このディレクトリには web.xml ファイルがあります。

展開された EJB アプリケーションには、最上位にある META-INF ディレクトリが含まれ、このディレクトリには ejb-jar.xml ファイルがあります。

展開されたコネクタには、最上位にある META-INF ディレクトリが含まれ、このディレクトリには ra.xml ファイルがあります。

注意： REDEPLOY ファイルは、デプロイされているアプリケーション全体またはデプロイされているスタンドアロン モジュールに対してのみ機能します。展開されたエンタープライズ アプリケーションをデプロイしてある場合、REDEPLOY ファイルは、エンタープライズ アプリケーション内の個々のモジュール (Web アプリケーションなど) ではなく、アプリケーション全体の再デプロイメントを制御します。Web アプリケーション自体を展開されたアーカイブ ディレクトリとしてデプロイする場合、REDEPLOY ファイルは Web アプリケーション全体の再デプロイメントを制御します。

2. 展開されたアプリケーションを更新するには、更新されたファイルをそのディレクトリ内の既存のファイルに上書きコピーします。
3. 新しいファイルをコピーしたら、展開されたディレクトリ内の REDEPLOY ファイルのタイムスタンプを更新します。

管理サーバは、タイムスタンプの変更を検出すると、展開されたディレクトリの内容を再デプロイします。

デプロイメント管理 API

デプロイメント タスクは、DeployerRuntimeMBean という、WebLogic 管理サーバに常駐するシングルトン (インスタンスが 1 つしかないオブジェクト) によって開始されます。この DeployerRuntimeMBean によって、アプリケーションをアクティブ化、非アクティブ化、および削除するためのメソッドが提供されます。これらのメソッドは、要求をカプセル化し、その進行状況を追跡する手段を提供する DeploymentTaskRuntimeMBean を返します。
DeploymentTaskRuntimeMBean は、TargetStatus オブジェクトによってデプロイ先ごとに要求に関する処理ステータスを示します。

WebLogic Server デプロイメント管理 API は、以下の WebLogic Server MBean によって定義されます。

- `DeployerRuntimeMBean` - デプロイメント要求へのプログラム的なインタフェース。`DeployerRuntimeMBean` によってデプロイメント要求が行われると、アプリケーションと該当するコンポーネント コンフィグレーション MBean に対して、コンフィグレーション ステートが示されます。これらの MBean は、アプリケーションのデプロイメント ステートを WebLogic Server ドメインに保持します。
- `DeploymentTaskRuntimeMBean` - さまざまなデプロイメント タスクへのインタフェース。

デプロイメント管理 API は非同期です。クライアントは、ステータスをポーリングするか、`ApplicationMBean` 通知を使用してタスクがいつ完了したかを判別します。

WebLogic Server のデプロイメント管理 API の詳細については、`weblogic.management.deploy` Javadoc を参照してください。

アプリケーション デプロイメントのベスト プラクティス

実際にアプリケーションをデプロイする場合のベスト プラクティスを以下に示します。

単一サーバでのデプロイメント

繰り返しデプロイメントが行われる環境では、すべてのファイルを展開形式で維持し、アプリケーションをソースの場所から直接デプロイする方法がもっとも効率的です。

単一サーバ環境 (管理サーバだけにデプロイ) で作業している場合、ディレクトリ ベースの自動デプロイメントを使用できます。すなわち、展開されたアプリケーションを mydomain/applications ディレクトリに配置するだけで、アプリケーションをデプロイできます。詳細については、このマニュアルの「自動デプロイメント」を参照してください。

Web アプリケーションまたは Web サービスの変更テスト

- mydomain/applications ディレクトリにあるアプリケーションの JSP ファイルまたは静的データ ファイルを変更して、アプリケーションで変更を表示できます。
- 更新したサーブレット クラスを、mydomain/applications ディレクトリの Web アプリケーション (mydomain/applications/exploded_web_app_dir/WEB-INF/classes など) に直接コンパイルすることもできます。新しいクラスは Web アプリケーションに組み込まれ、追加手順は必要ありません。
- Web アプリケーション デプロイメント記述子に対する変更を組み込むには、mydomain/applications ディレクトリの WEB-INF/REDEPLOY ファイルを変更します。これによって自動デプロイヤがその変更を検出します。Web アプリケーションが再デプロイされます。

EJB およびリソース アダプタの変更テスト

EJB またはリソース アダプタ デプロイメント記述子に対する変更を組み込むには、mydomain/applications ディレクトリの META-INF/REDEPLOY ファイルを変更します。これによって、自動デプロイヤがその変更を検出します。EJB またはリソース アダプタが再デプロイされます。

注意 : EJB がエンタープライズ アプリケーションの一部になっている場合、そのアプリケーション全体が再デプロイされます。

複数サーバでのデプロイメント

複数サーバ環境で作業している場合、weblogic.Deployer ツールを使用してアプリケーションをデプロイすることをお勧めします。

複数サーバ環境における変更テスト

アプリケーション ファイルに対して変更を行った場合、weblogic.Deployer ツールを使用してその変更をサーバに伝達する必要があります。サーバに伝達することにより、変更がデプロイ済みアプリケーションに組み込まれます。

Web アプリケーションに対する変更をサーバに伝達する方法を以下の手順で示します。この手順は、どのタイプのアプリケーションでも同じです。ただし、EJB がエンタープライズ アプリケーションに含まれる場合、アプリケーションおよびそのコンポーネント全体が再デプロイされます。

1. Administration Console または次のコマンドを使用して、Web アプリケーションをデプロイします。

   ```
   java weblogic.Deployer -adminurl http://adminAddr:7001 -name
   webapp -activate -source /myapp/webapp -targets
   managedserver1,managedserver2
   ```

2. JSP: /myapp/webapp/jsps/login.jsp に対して必要な変更を行います。

3. 行った変更を以下のように適用します。

   ```
   java weblogic.Deployer -adminurl http://adminAddr:7001 -name
   webapp -activate jsps/login.jsp
   ```

上記の例では、login.jsp が、Web アプリケーションがデプロイされているすべてのサーバに配布され、組み込まれます。

展開されたアプリケーションのファイル構造

デプロイ可能なユニットのパッケージ化の詳細については、「WebLogic Server アプリケーションのパッケージ化」を参照してください。

ディレクトリに複数のモジュールがあるが、全体的なアプリケーション記述子ファイルがない場合は、各モジュールを独自のディレクトリに格納し、独自の記述子ファイルを含める必要があります。つまり、非アーカイブ アプリケーションでは、アーカイブ内と同じく、各モジュールが他のモジュールと互いに独立していなければなりません。

```
sourceDirectory\

               \Module1
                     WEB-INF\
                             web.xml
                             Module1FilesDir
```

```
                \Module2
                      META-INF\
                              ejb-jar.xml
                              Module2FilesDir
```

非 EAR デプロイメントを展開した場合、ソース ディレクトリは必ず A または B となる必要があります。

A.

```
 \Module1
         WEB-INF\
                web.xml
                Module1FilesDir
```

B.

```
 \Module1
         WEB-INF\
                web.xml
                Module1FilesDir

 \Module2
         META-INF\
                ejb-jar.xml
                Module2FilesDir
 \Module3
         META-INF\
                ra.xml
                Module3FilesDir
```

以下のディレクトリ構造はサポートされていないので、無効です。

```
sourceDirectory/

       META-INF\ejb-jar.xml

       WEB-INF\web.xml

       webfiles/

       ejbfiles/


moduledir/

       WEB-INF\web.xml

       webfiles

       ejb.jar

moduledir/

       WEB-INF\web.xml

       web1files
```

```
web2/WEB-INF/web.xml
web2/webfiles
```

ステージング モード

管理サーバの場合には nostage、管理対象サーバの場合には stage というデフォルト値を使用します。ただし、以下の場合を除きます。

- 管理サーバから管理対象サーバ マシンへのファイル コピーおよび配布管理にサードパーティのソリューションを使用する場合
- ドメイン内の異なるサーバ間でアプリケーション ソースを共有するため共有ファイル システムを使用する場合

自動デプロイメント

単一サーバ デプロイメントの開発設定では、自動デプロイメントのみ使用します。

展開エンタープライズ アプリケーション

複数モジュールによる展開デプロイメントでは、常に META-INF/application.xml で定義されたアプリケーション記述子を使用します。

部分的な再デプロイメント

モジュールまたはファイルを、アプリケーションの、参照を持つ他のモジュールに再デプロイする場合は、参照しているモジュールも再デプロイする必要があります。

エンタープライズ アプリケーションの一部であるコンポーネント間のクラス共有

エンタープライズ アプリケーションの一部であるコンポーネント間でクラスを共有するには、MANIFEST クラスパスを使用します。共有クラスを含む JAR ユーティリティは、その他のコンポーネント アーカイブの次に EAR にパッケージ化されます。これらのクラスを使用する各コンポーネントは、コンポーネント アーカイブ内の META-INF/MANIFEST.MF というファイルに Class-Path エントリを作成します。この方法は J2EE 標準の一部であり、アプリケーション サーバ間での移植性が必要な場合に使用します。

詳細については、3-11 ページの「マニフェスト クラスパス」を参照してください。

WebLogic Server 6.x のデプロイメント プロトコルの使用

デフォルトでは、利用可能なすべてのデプロイメント ツールによって新しいアプリケーションをデプロイする場合は、2 フェーズ デプロイメント プロトコルが使用されます。現在の管理サーバでも、WebLogic Server 6.x デプロイメント プロトコルはサポートされています。このプロトコルは以下の場合に使用します。

- コンフィグレーション済みのアプリケーションで、2 フェーズ デプロイメント プロトコルを指定 (ApplicationMBean.TwoPhase=true) しない場合
- アプリケーションに複数のモジュールがあり、かつ EAR ではない場合

6.x のプロトコルを使用するアプリケーションを 2 フェーズ プロトコルを使用するようにコンフィグレーションするには、次のように、ドメインからアプリケーションを削除 (コンフィグレーションを削除) してから、アプリケーションを再度アクティブ化します。

1. weblogic.Deployer を使用してアプリケーションを削除します。次の形式でコマンドを入力します。

```
java weblogic.Deployer -adminurl http://admin:7001 -name app
-targets server -remove
```

2. `weblogic.Deployer` を使用してアプリケーションを再度アクティブ化します。次の形式でコマンドを入力します。

```
java weblogic.Deployer -activate -name ArchivedEarJar -source
C:/MyApps/JarEar.ear –target server1
```

新しいプロトコルを使用してアプリケーションが再デプロイされます。

デプロイメント補足ドキュメント

WebLogic Server デプロイメントの詳細については、以下のマニュアルを参照してください。

マニュアル	デプロイメントのトピック
WebLogic Builder	WebLogic Builder を使用して、J2EE アプリケーションおよびそのコンポーネント用の XML デプロイメント記述子ファイルを編集、生成する方法
Administration Console オンラインヘルプ	デプロイメント タスクにおける Administration Console の使用方法
「クラスタのコンフィグレーションとアプリケーションのデプロイメント」	クラスタ化されたサーバへのデプロイ方法
WebLogic エンタープライズ Java Beans プログラマーズ ガイド	WebLogic Server EJB のデプロイ方法
WebLogic J2EE コネクタ アーキテクチャ	WebLogic Server J2EE コネクタのデプロイ方法

マニュアル	デプロイメントのトピック
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 Server Web アプリケーションのデプロイ方法
WebLogic JSP プログラマーズ ガイド	JSP からのアプレットのデプロイ方法
WebLogic Server 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 コンポーネントのパッケージ方法

6 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。

メッセージのロギング

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サイドのログ ファイルにメッセージを書き込むときにも使用できます。

詳細については、『WebLogic Server ロギング サービスの使い方』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アプリケーション コンポーネントを構築する必要があります。

通常、サーバサイド コンポーネントで新たなスレッドの作成が必要にならないアプリケーション設計をします。

- 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限できるようにスレッド プールを作成します。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も最適化できます。

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 (Internet Message Access Protocol) 対応および SMTP (Simple Mail Transfer Protocol) 対応のメール 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 (http://java.sun.com/products/javamail/index.html) 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` パッケージが入っており、また、`weblogic.jar` には、JavaMail で必要な JAF (Java Activation Framework) 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 Sun の手順に従います。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 を使用して 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 [Mail メール] ノードをクリックします。
2. [Create a New Mail Session 新しい Mail Session の作成] 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。
   - [Name 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。
   - [JNDINameJNDI 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。
   - [Properties 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ーライドできます。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。

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。
`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。

プロパティ	説明	デフォルト値
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 (ネーミング)、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を追加します。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、これらの例外を捕捉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 (ネーミング)、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を追加します。`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特定の要件に従う必要があります。詳細については、『WebLogic Server クラスタ ユーザーズ ガイド』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。クラスタにおけるさまざまな EJB タイプの機能に関する詳細については、『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』の「WebLogic Server EJB コンテナとサポートされるサービス」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 エンタープライズ JavaBeans プログラマーズ ガイド』の「weblogic-ejb-jar.xml 文書型定義」に、クラスタ化に関する XML デプロイメント要素が説明されています。

クラスタでデプロイするために EJB またはカスタム RMI オブジェクトのいずれかを開発する場合は、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、『WebLogic JNDI プログラマーズ ガイド』の「クラスタ環境での WebLogic JNDI の使い方」も参照してください。

A アプリケーション デプロイメント記述子の要素

以下の節では、WebLogic Server の J2EE 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。デプロイメント記述子は、`application.xml` という J2EE 標準デプロイメント記述子と、`weblogic-application.xml` という WebLogic 固有のアプリケーション デプロイメント記述子の 2 つが必要です。`weblogic-application.xml` ファイルは、WebLogic Server の拡張機能を使用していない場合は省略可能です。

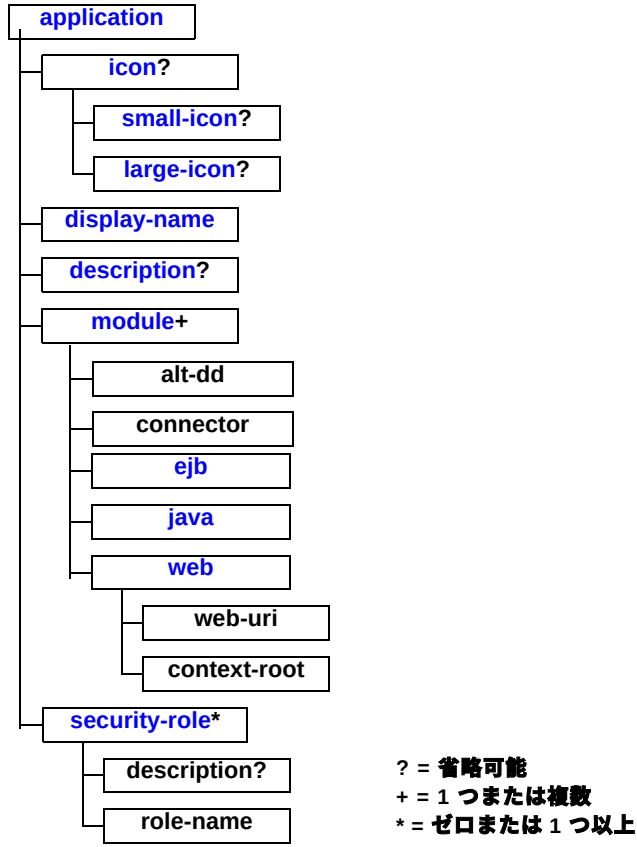
application.xml デプロイメント記述子の要素

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 1.3//EN"
"http://java.sun.com/dtd/application_1_3.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

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

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。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 URI を指定します。

alt-dd

特定の J2EE モジュールに対して、後でアセンブリするデプロイメント記述子ファイルへのオプション URI を指定します。この URI は、アプリケーションのルート ディレクトリに対する、デプロイメント記述子ファイルの絶対パス名を指定します。alt-dd を指定しない場合、各コンポーネントの仕様に必須であるデフォルトの場所およびファイル名からデプロイメント記述子が読み込まれます。

connector

リソース アダプタ (コネクタ) アーカイブ ファイルの URI (アプリケーション パッケージの最上位に対する相対位置) 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

application.xml ファイル内の 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 context-root 要素があります。context-root の値を宣言しない場合、web-uri 要素のベース名が Web アプリケーションのコンテキスト パスとして使用されます (コンテキスト パスは、Web サーバ内でユニークな値にします。これは、複数の Web アプリケーションで同一の Web サーバを使用する場合があり、それらの複数アプリケーション間でコンテキスト パスが競合しないようにする必要があるからです)。

web-uri

application.xml ファイル内の Web モジュールの場所を定義します。これは WAR ファイルの名前になります。

context-root

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 (省略可能) 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

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>
```

weblogic-application.xml デプロイメント記述子の要素

以下の節では、weblogic-application.xml ファイルについて説明します。weblogic-application.xml ファイルは、Sun Microsystems から提供された application.xml デプロイメント記述子を拡張した BEA WebLogic Server 固有のデプロイメント記述子です。このファイルによって、アプリケーション スコープの JDBC プールや EJB キャッシュなどの機能のコンフィグレーションを行います。

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 weblogic-application PUBLIC "-//BEA Systems, Inc.//DTD
WebLogic Application 7.0.0//EN"

"http://www.bea.com/servers/wls700/dtd/weblogic-application_1_0.d
td";>
```


weblogic-application
ejb?
entity-cache+
entity-cache-name
max-beans-in-cache?
max-cache-size?
bytes | megabytes
read-timeout-seconds?
caching-strategy?
start-mdbs-with-application?
xml?
parser-factory?
saxparser-factory?
document-builder-factory?
transformer-factory?
entity-mapping*
entity-mapping-name
public-id?
system-id?
entity-uri?
when-to-cache
cache-timeout-interval
jdbc-connection-pool*
data-source-name
connection-factory
factory-name
connection-properties?
user-name?
url?
driver-class-name?
connection-params*
parameter+
param-value
param-name
pool-params?
size-params?
initial-capacity?
max-capacity?
capacity-increment?
shrinking-enabled?
shrink-period-minutes?
xa-params?
debug-level?
keep-conn-until-tx-complete-enabled?
end-ordy-once-enabled?
resume-ordy-once-enabled?
tx-context-on-close-needed?
new-conn-for-commit-enabled?
prepared-statement-cache-size?
keep-logical-conn-open-on-release?
local-transaction-supported?
resource-health-monitoring-enabled?
login-delay-seconds?
leak-profiling-enabled?
connection-check-params?
table-name?
check-on-reserve-enabled?
check-on-release-enabled?
refresh-minutes?
driver-params?
profiling-enabled?
statement?
prepared-statement?
row-prefetch-enabled?
row-prefetch-size?
stream-chunk-size?
jndi-name?
application-param*
profiling-enabled?
cache-profiling-threshold?
cache-size?
parameter-logging-enabled?
max-parameter-length?
? = 省略可能
+ = 1 つまたは複数
* = 0 つまたは複数

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。

weblogic-application

weblogic-application 要素は、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。

ejb

省略可能。ejb 要素には、WebLogic アプリケーションの構成要素となる EJB モジュールに固有の情報が含まれます。現在、ejb 要素では、アプリケーションのエンティティ Bean によって使用される任意の数のアプリケーション レベル キャッシュを指定できます。

entity-cache

1 つまたは複数、指定します。entity-cache 要素は、実行時にエンティティ EJB インスタンスをキャッシュに入れるときに使用される名前付きアプリケーションレベル キャッシュの定義に使用されます。個々のエンティティ Bean は、使用するアプリケーションレベル キャッシュのキャッシュ名を参照します。個々のキャッシュを参照するエンティティ Bean の数に制限はありません。

アプリケーションレベル キャッシュは、エンティティ Bean が weblogic-ejb-jar.xml 記述子で独自のキャッシュを指定しない場合に、デフォルトとして使用されます。その場合、ExclusiveCache および MultiVersionCache という 2 つのデフォルト キャッシュが使用されます。アプリケーションの設定で、これらのデフォルト キャッシュを明示的に定義して、非デフォルト値を指定する場合もあります。キャッシング方式では、デフォルト キャッシュは変更できません。デフォルトで、キャッシュは最大サイズとして値 1000 を指定して max-beans-in-cache を使用します。

例 :

```
<entity-cache>
       <entity-cache-name>ExclusiveCache</entity-cache-name>
```

```
        <max-cache-size>
              <megabytes>50</megabytes>
        </max-cache-size>
</entity-cache>
```

entity-cache-name

entity-cache-name 要素は、エンティティ Bean キャッシュにユニークな名前を指定します。この名前は、ear ファイル内でユニークなものとし、空の文字列は使用できません。

例：

```
<entity-cache-name>ExclusiveCache</entity-cache-name>
```

max-beans-in-cache

省略可能。max-beans-in-cache 要素は、キャッシュに入れることができるエンティティ Bean の最大数を指定します。限度に達すると、Bean に対してパッシベーションが行われる場合があります。このメカニズムでは、個々のエンティティ Bean が必要とする実際のメモリ サイズは考慮されません。この要素は、1 以上の値に設定できます。

デフォルト値：1000

max-cache-size

max-cache-size 要素は、エンティティ キャッシュのメモリ サイズの限度をバイト単位または MB 単位で指定するときに使用されます。max-cache-size 要素を使用して最大サイズが指定されたキャッシュを Bean が使用する場合、Bean プロバイダで weblogic-ejb-jar.xml 記述子の Bean の平均サイズを見積もる必要があります。デフォルトでは、Bean の平均サイズは 100 バイトと想定されています。

- bytes | megabytes ― バイト単位または MB 単位で示されるエンティティ キャッシュのメモリ サイズ。max-cache-size 要素で使用されます。

caching-strategy

省略可能。caching-strategy 要素は、EJB コンテナが特定のアプリケーション レベル キャッシュでエンティティ Bean インスタンスを管理するときに使用する一般的な方式を指定します。キャッシュによって、メモリ内のエンティティ Bean インスタンスがバッファに移され、対応する主キー値に関連付けられます。

caching-strategy 要素の値は、以下のいずれかに限られます。

- Exclusive ― 各主キー値ごとにメモリ内の Bean インスタンスを 1 つキャッシュに入れます。この固有のインスタンスは、通常、使用時に EJB コンテナの排他的ロックを使用してロックされ、同時にそのインスタンスを使用するトランザクションは 1 つだけになります。
- MultiVersion ― 各主キー値に対してメモリ内の複数の Bean インスタンスをキャッシュに入れます。各インスタンスを複数のトランザクションで同時に使用できます。

デフォルト値 : MultiVersion

例 :

```
<caching-strategy>Exclusive</caching-strategy>
```

start-mdbs-with-application

省略可能。アプリケーションを使用してメッセージ駆動型 Bean (MDBS) を起動できるように EJB コンテナをコンフィグレーションできます。true に設定されると、コンテナはアプリケーションの一部として MDBS を起動します。false に設定されると、コンテナは MDBS をキューに保持し、ポートでのリスンが開始されたときにサーバによって MDBS が起動されます。

xml

省略可能。xml 要素では、対象アプリケーションに固有な XML 処理のパーサおよびエンティティ マッピングに関する情報を指定します。

parser-factory

省略可能。parser-factory 要素には、saxparser-factory、document-builder-factory、および transformer-factory という 3 つの要素があります。

saxparser-factory

省略可能。saxparser-factory 要素では、対象アプリケーションだけで必要とされる XML 解析用の SAXParser ファクトリを設定できます。この要素によって、SAX スタイル解析に使用されるファクトリが指定されます。saxparser-factory 要素の設定をしないと、サーバ XML レジストリでコンフィグレーションされた SAXParser ファクトリ スタイルが使用されます。

デフォルト値 : サーバ XML レジストリの設定

document-builder-factory

省略可能。document-builder-factory 要素では、対象アプリケーションだけで必要とされる XML 解析用のドキュメント ビルダ ファクトリを設定できます。この要素によって、DOM スタイル解析に使用されるファクトリが決定されます。document-builder-factory 要素を設定しないと、サーバ XML レジストリでコンフィグレーションされた DOM スタイルが使用されます。

デフォルト値 : サーバ XML レジストリの設定

transformer-factory

省略可能。transformer-factory 要素では、対象アプリケーションだけで必要とされるスタイル シート処理用のトランスフォーマ エンジンを設定できます。この要素の値を指定しないと、サーバ XML レジストリでコンフィグレーションされた値が使用されます。

デフォルト値 : サーバ XML レジストリの設定

entity-mapping

ゼロまたは 1 つ以上、指定します。entity-mapping 要素はエンティティ マッピングの指定に使用されます。マッピングにより、特定のパブリック ID またはシステム ID の代替エンティティ URI が指定されます。このエンティティ URI を検索するデフォルトの場所は、lib/xml/registry ディレクトリです。

entity-mapping-name

entity-mapping-name 要素は、エンティティ マッピングの名前を指定します。

public-id

省略可能。public-id 要素は、マップされたエンティティのパブリック ID を指定します。

system-id

省略可能。system-id 要素は、マップされたエンティティのシステム ID を指定します。

entity-uri

省略可能。entity-uri 要素は、マップされたエンティティの EntityURI を指定します。

when-to-cache

省略可能。有効値は、次のとおりです。

- cache-on-reference
- cache-at-initialization
- cache-never

デフォルト値は cache-on-reference です。

cache-timeout-interval

省略可能。cache-timeout-interval 要素には、秒単位の整数値を指定できます。

jdbc-connection-pool

ゼロまたは 1 つ以上、指定します。jdbc-connection-pool 要素は、アプリケーション スコープの JDBC 接続プールを指定します。

data-source-name

data-source-name 要素は、アプリケーション固有の JNDI ツリーにある JNDI 名を指定します。

connection-factory

connection-factory 要素は、プールの初期化時に作成される物理的なデータベースの接続数を定義します。デフォルト値は 1 です。

factory-name

factory-name 要素は、config.xml ファイル内の JDBCDataSourceFactoryMBean の名前を指定します。

connection-properties

省略可能。connection-properties 要素は、デフォルトの接続ファクトリ設定のオーバーライドを定義する接続パラメータを指定します。

- user-name — 省略可能。user-name 要素は、JDBCDataSourceFactoryMBean の UserName のオーバーライドに使用されます。
- url — 省略可能。url 要素は、JDBCDataSourceFactoryMBean の URL のオーバーライドに使用されます。
- driver-class-name — 省略可能。driver-class-name 要素は、JDBCDataSourceFactoryMBean の DriverName のオーバーライドに使用されます。
- connection-params — ゼロまたは 1 つ以上。
  - parameter+ (param-value, param-name) — 1 つまたは複数。

pool-params

省略可能。pool-params 要素は、プール動作を変更するパラメータを定義します。

size-params

省略可能。size-params 要素は、プール内の接続数を変更するパラメータを定義します。

- initial-capacity ― 省略可能。initial-capacity 要素は、プールの初期化時に作成される物理的なデータベース接続数を定義します。デフォルト値は 1 です。
- max-capacity ― 省略可能。max-capacity 要素は、プールに含める物理的なデータベースの最大接続数を定義します。ただし、JDBC ドライバによってこの値がさらに制限されることがあります。デフォルト値は 1 です。
- capacity-increment ― 省略可能。capacity-increment 要素は、プール容量を拡張するときの増分を定義します。サービス要求のために利用可能となる物理的な接続が他にないとき、プールは必要なデータベースの物理的な接続数を作成し、プールに追加します。プール内の接続数は、max-capacity で設定された物理的な最大接続数を超えないように保持されます。デフォルト値は 1 です。
- shrinking-enabled ― 省略可能。shrinking-enabled 要素は、使用されていない接続が検出された場合に、プールの容量を initial-capacity に戻すかどうかを指定します。
- shrink-period-minutes ― 省略可能。shrink-period-minutes 要素は、要求を満たすためにインクリメンタルに増やした接続プールを縮小するまで待機する時間 (分単位) を定義します。縮小を可能にするには、shrinking-enabled 要素を true に設定しておく必要があります。

xa-params

省略可能。xa-params 要素は、XA DataSource のパラメータを定義します。

- debug-level ― 省略可能。整数。debug-level 要素は XA 処理のデバッグレベルを定義します。デフォルト値は 0 です。

- keep-conn-until-tx-complete-enabled ― 省略可能。ブール値。keep-conn-until-tx-complete-enabled 要素を true に設定すると、トランザクションが完了するまで、XA 接続プールによって同一の XA 接続が分散トランザクションに関連付けられます。
- end-only-once-enabled ―省略可能。ブール値。end-only-once-enabled 要素を true に設定すると、保留中の各 XAResource.start() メソッドに対する XAResource.end() メソッドの呼び出しが 1 度に限られます。
- recover-only-once-enabled ―省略可能。ブール値。recover-only-once-enabled 要素を true に設定すると、リソースに対する回復の呼び出しが 1 度に限られます。
- tx-context-on-close-needed ― 省略可能。tx-context-on-close-needed 要素は、さまざまな JDBC オブジェクト (例 : 結果セット、ステートメント、接続など) を閉じるときに XA ドライバで分散トランザクション コンテキストが必要な場合、true に設定します。true に設定されると、JDBC オブジェクトを閉じるときに送出される SQL 例外のうち、トランザクション コンテキストでないものは無条件で受信されます。
- new-conn-for-commit-enabled ―省略可能。ブール値。new-conn-for-commit-enabled 要素を true に設定すると、特定の分散トランザクションのコミットおよびロールバック処理に専用の XA 接続が使用されます。
- prepared-statement-cache-size ― 省略可能。prepared-statement-cache-size 要素は、Prepared Statement のキャッシュ サイズの設定に使用します。このキャッシュ サイズは、特定の接続から作成される Prepared Statement の数を表し、後で使用できるようにキャッシュに格納されます。Prepared Statement キャッシュ サイズを 0 に設定すると、キャッシュがオフになります。
- keep-logical-conn-open-on-release ― 省略可能。ブール値。keep-logical-conn-open-on-release 要素を true に設定すると、XA の物理的な接続が XA 接続プールに返されても JDBC の論理的な接続はオープンしたまま保持されます。デフォルト値は false です。
- local-transaction-supported ― 省略可能。ブール値。XA ドライバがグローバル トランザクションを使用しない SQL をサポートする場合には

local-transaction-supported を true に設定します。そうでない場合は、false に設定してください。デフォルト値は false です。

- resource-health-monitoring-enabled ― 省略可能。resource-health-monitoring-enabled 要素は、対象接続プールに対して JTA リソース状態モニタ機能を有効にする場合には true に設定します。

login-delay-seconds

省略可能。整数値。login-delay-seconds 要素は、各データベースの物理的な接続を作成するまで待機する遅延時間を秒単位で設定します。データベース サーバによっては、複数の接続リクエストが短い間隔で繰り返されると処理できないものもあります。このプロパティを使用すると、データベース サーバの処理が追いつくように、少しの間隔をあけることができます。この遅延は、データベースの物理的な接続が確立すると、プールの初期作成時とプールの有効期間中の両方で必ず行われます。

leak-profiling-enabled

省略可能。leak-profiling-enabled 要素は、JDBC 接続リーク プロファイルを有効にします。接続リークは、プールからの接続が close() メソッドの呼び出しで明示的にクローズされていない場合に発生します。接続リーク プロファイルがアクティブの場合、プールは接続オブジェクトがプールから割り当てられ、クライアントに与えられたときにスタック トレースを格納します。接続リークが検出されたとき (接続オブジェクトのガベージ コレクションが行われたとき) に、このスタック トレースが報告されます。

この要素はリソースを余計に使用し、接続プール処理を遅くする可能性があるので、プロダクション環境での使用はお勧めできません。

connection-check-params

省略可能。connection-check-params 要素は、プール内の接続がまだ有効であることを確認するチェックを行うかどうか指定し、行う場合の時機と方法を定義します。

- table-name ― 省略可能。table-name 要素は、スキーマ内でクエリできるテーブルを定義します。
- check-on-reserve-enabled ― 省略可能。check-on-reserve-enabled 要素が true に設定されると、ユーザに接続が渡される前に毎回接続がテストされます。

- check-on-release-enabled ― 省略可能。check-on-release-enabled 要素が true に設定されると、ユーザが接続をプールに返すときに毎回接続がテストされます。
- refresh-minutes ― 省略可能。refresh-minutes 要素が定義されると、定期的 (指定された分単位の時間) にトリガされます。トリガによって、プール内の各接続がまだ有効であることを確認するチェックが行われます。

driver-params

省略可能。driver-params 要素は、WebLogic Server ドライバ上の動作を設定します。

statement

省略可能。

- profiling-enabled ―省略可能。ブール値。profiling-enabled 要素は、JDBC SQL の応答プロファイルの実行を有効にします。有効化されると、後で分析するため SQL 文のテキスト、実行時間、その他のメトリックが外部的に格納されます。この機能では多くのリソースを使用するため、プロダクション環境のサーバではオフにすることをお勧めします。デフォルト値は false です。

prepared-statement

省略可能。profiling-enabled はブール値。prepared-statement 要素は、JDBC Prepared Statement のキャッシュ プロファイルの実行を有効にします。有効化されると、Prepared Statement のキャッシュ プロファイルが後で分析できるように外部ストレージに格納されます。この機能では多くのリソースを使用するため、プロダクション環境のサーバではオフにすることをお勧めします。デフォルト値は false です。

- profiling-enabled ― 省略可能。
- cache-profiling-threshold ― 省略可能。
  cache-profiling-threshold 要素は、Prepared Statement キャッシュの状態がログに記録されるステートメント要求の数を定義します。この要素は出力量を最小にします。この機能では多くのリソースを使用するため、プロダクション環境のサーバではオフにすることをお勧めします。

- `cache-size` ― 省略可能。`cache-size` 要素は、Prepared Statement キャッシュのサイズを返します。このキャッシュ サイズは、特定の接続から作成される Prepared Statement の数を表し、後で使用できるようにキャッシュに格納されます。
- `parameter-logging-enabled` ― 省略可能。SQL 応答プロファイリング時、Prepared Statement パラメータの値を格納できます。`parameter-logging-enabled` 要素を使用すると、ステートメント パラメータを格納できるようになります。この機能では多くのリソースを使用するため、プロダクション環境のサーバではオフにすることをお勧めします。
- `max-parameter-length` ― 省略可能。SQL 応答プロファイリング時、Prepared Statement パラメータの値を格納できます。`max-parameter-length` 要素は、JDBC SQL 応答プロファイリングのパラメータとして渡される文字列の最大の長さを定義します。この機能は多くのリソースを使用するため、パラメータのデータ長を制限して、出力量を減らすようにしてください。

row-prefetch-enabled

省略可能。

row-prefetch-size

省略可能。

stream-chunk-size

省略可能。

acl-name

省略可能。

application-param

ゼロまたは 1 つ以上、指定します。`application-param` 要素は、コンテナ動作に関係するさまざまなパラメータを定義します。定義可能なパラメータは以下のとおりです。

- webapp.encoding.usevmdefault
- webapp.encoding.default
- webapp.getrealpath.accept_context_path

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。J2EE アプリケーションの場合、ユーザの関心は通常サーバサイドのコンポーネント (Web アプリケーション、EJB、コネクタ) にあります。サーバサイドのコンポーネントのコンフィグレーションには、付録 A「アプリケーション デプロイメント記述子の要素」に説明されている `application.xml` デプロイメント記述子を使用します。

しかし、クライアント コンポーネント (JAR ファイル) を EAR ファイルに含めることもできます。この JAR ファイルは、クライアントサイドでのみ使用するもので、このクライアント コンポーネントのコンフィグレーションには `client-application.xml` デプロイメント記述子を使用します。この方法により、クライアントサイドとサーバサイド、両方のコンポーネントのパッケージ化が可能になります。サーバはサーバが関係する部分 (`application.xml` ファイルによる) だけを参照し、クライアントはクライアントが関係する部分 (`client-application.xml` ファイルによる) だけを参照します。

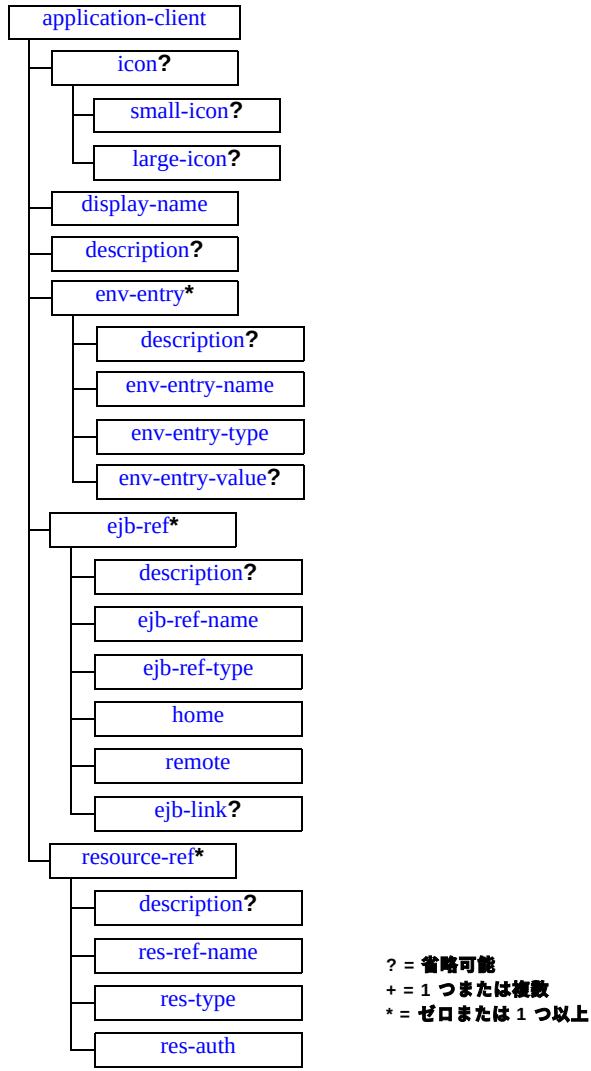
クライアントサイド コンポーネントには、`application-client.xml` という J2EE の標準デプロイメント記述子、およびクライアント アプリケーションの JAR ファイルから派生した名前を持つ WebLogic 固有の実行時デプロイメント記述子の 2 種類が必要です。

application-client.xml のデプロイメント記述子の要素

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

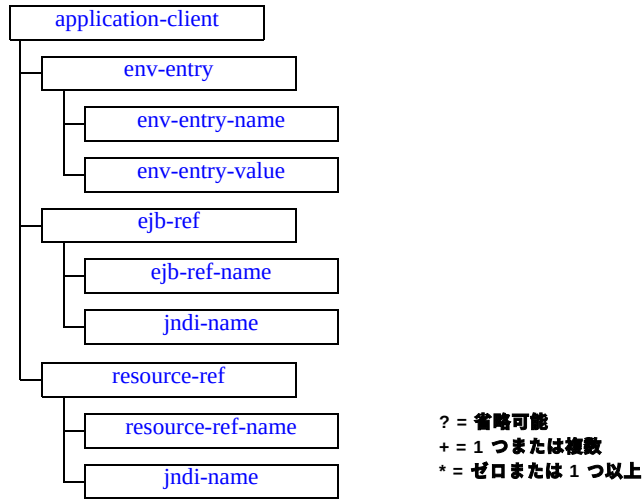
res-auth 要素は、EJB コードによってプログラムでリソース マネージャに対するサインオンを行うか、または EJB の代わりにコンテナによってリソース マネージャに対するサインオンを行うかを指定します。後者の場合、コンテナは、デプロイヤから提供される情報を使用します。res-auth 要素には、Application または Container のうち 1 つまたは複数を指定できます。

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

て

は

ふ

へ