BEA WebLogic Server

WebLogic RMI
プログラマーズ ガイド

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

著作権

限定的権利条項

商標または登録商標

WebLogic RMI プログラマーズ ガイド

パート番号	マニュアルの日付	ソフトウェアのバージョン
なし	2002 年 6 月 24 日	BEA WebLogic Server バージョン 6.1

目次

このマニュアルの内容

1. WebLogic RMI について

2. プログラミングの考慮事項

3. WebLogic RMI の実装

このマニュアルの内容

このマニュアルでは、Sun Microsystems による JavaSoft Remote Method Invocation（RMI）の BEA WebLogic Server™ RMI 実装について説明します。BEA の実装は WebLogic RMI と呼ばれます。

このマニュアルの構成は次のとおりです。

- 第 1 章「WebLogic RMI について」では、WebLogic RMI の機能とアーキテクチャの概要について説明します。
- 第 2 章「プログラミングの考慮事項」では、WebLogic Server の RMI のプログラミングで使用する機能について説明します。
- 第 3 章「WebLogic RMI の実装」では、WebLogic RMI の一部として添付されているパッケージについて説明し、WebLogic RMI の実装方法を示します（パブリック API には、WebLogic で実装された RMI 基本クラス、レジストリ、およびサーバのパッケージが含まれています）。

対象読者

このマニュアルは、Remote Method Invocation（RMI）機能を使用して 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を対象としています。Web テクノロジ、オブジェクト指向プログラミング手法、および Java プログラミング言語に読者が精通していることを前提として書かれています。

e-docs Web サイト

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

このマニュアルの印刷方法

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

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

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

関連情報

BEA の Web サイトでは、WebLogic Server の全マニュアルを提供しています。このマニュアル以外に、『WebLogic RMI over IIOP プログラマーズ ガイド』を参照する場合があります。WebLogic RMI over IIOP を使用すると、クライアントは、Internet Inter-ORB Protocol（IIOP）を介して RMI リモート オブジェクトにアクセスできるため、RMI プログラミング モデルが拡張されます。

サポート情報

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

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

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

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

表記規則

このマニュアルでは、全体を通して以下の表記規則が使用されています。

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

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

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

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

1 WebLogic RMI について

以下の節では、WebLogic RMI の機能について概説します。

- WebLogic RMI とは
- WebLogic RMI の機能

WebLogic RMI とは

Remote Method Invocation （RMI）は、Java を使用した分散オブジェクト コンピューティングのための標準仕様です。RMI を使用すると、アプリケーションはネットワーク内の別の場所に存在するオブジェクトを参照し、そのオブジェクトのメソッドを、そのオブジェクトがあたかもクライアントの仮想マシンにローカルに存在するかのように呼び出すことができます。RMI は、分散 Java アプリケーションがどのように複数の Java 仮想マシン（JVM）で動作するかを指定するものです。

WebLogic は、JavaSoft RMI 仕様を実装しています。WebLogic RMI は、標準ベースの分散コンピューティングを実現します。WebLogic Server を使用すると、高速かつ信頼性の高い大規模なネットワーク コンピューティングが可能になります。さらに WebLogic RMI を使用すると、製品、サービス、リソースをネットワーク内のどこにでも配置しつつ、プログラマとエンド ユーザにはそれらをローカル環境の一部であるかのように見せることができます。

WebLogic RMI は直線的なスケーラビリティをサポートするだけでなく、コンフィグレーションされた数のサーバ スレッドに実行要求を分割できます。複数のサーバ スレッドの使用により、WebLogic Server はレイテンシと空いたプロセッサを活用できます。

WebLogic RMI は、標準に完全に準拠しています。RMI の他の実装を使用する場合は、import 文を変更するだけでプログラムを変換できます。JavaSoft の RMI の参照実装と WebLogic の RMI 製品には相違がありますが、これらの相違は開発者からはまったく見えません。

また、WebLogic RMI は WebLogic Java Naming and Directory Interface（JNDI）と完全に統合されています。WebLogic RMI では JNDI API またはレジストリ インタフェースを使用して、アプリケーションを意味のあるネーム スペースに分けることができます。

このマニュアルは WebLogic RMI の使い方について説明したものですが、リモート オブジェクトや分散アプリケーションの記述方法についての初心者向けチュートリアルではありません。RMI について知りたい場合は、JavaSoft Web サイトの「RMI tutorial」を参照してください。

WebLogic RMI の機能

JavaSoft の RMI 参照実装と同じように、WebLogic RMI は複数の JVM での透過的なリモート呼び出しを提供しています。RMI 仕様に書かれているリモート インタフェースおよび実装は、変更を加えずに WebLogic RMI で使用できます。

次の表に、RMI の WebLogic 実装の重要な機能を示します 。

表 1-1 WebLogic RMI のパフォーマンス

機能	WebLogic RMI
全体的なパフォーマンス	WebLogic Server フレームワークとの統合により、WebLogic RMI のパフォーマンスが向上する。WebLogic Server は、通信の基本サポート、スレッドとソケットの管理、効率的なガベージ コレクション、およびサーバ関連サポートを提供する。
スケーラビリティ	直線的なスケーラビリティをサポートする。JavaSort RMI に比べてスケーラビリティが極めて高い。PC クラスの比較的小規模な単一プロセッサのサーバであっても、1000 を超える RMI クライアントを同時にサポートできる（サーバの負荷とメソッド呼び出しの複雑さによって異なる）。

表 1-1 WebLogic RMI のパフォーマンス（続き）

機能	WebLogic RMI
スレッドとソケットの管理	WebLogic RMI のクライアントからネットワークへのトラフィック用に、非同期、双方向の単一接続が使用される。この接続で、WebLogic JDBC 要求などのサービスもサポートできる。
シリアライゼーション	高性能なシリアライズにより、リモート クラスを 1 回しか使用しない場合でも大幅なパフォーマンス向上が実現される。
共存オブジェクトの解決	同じ場所にあるオブジェクトがリモートとして定義されている場合のパフォーマンスの低下がない。同じ場所にある「リモート」オブジェクトへの参照は、実装オブジェクトへの直接参照として解決される。
サービスをサポートするプロセス	WebLogic RMI レジストリは RMI レジストリ プロセスに取って代わるものである。WebLogic RMI は WebLogic Server 内で稼動する。他のプロセスを追加する必要はない。

表 1-2 WebLogic RMI の使いやすさ

機能	WebLogic RMI
rmic	プロキシとバイトコードは実行時に WebLogic RMI によって動的に生成される。このため、クラスタ対応クライアントまたは IIOP クライアントを除いて、明示的に rmic を実行する必要はない。
簡単に使用できる拡張機能	リモート インタフェースとコード生成のための、簡単に使用できる拡張機能を備えている。たとえば、インタフェースの各メソッドは、throws ブロック内で `java.rmi.RemoteException` を宣言する必要がない。アプリケーションが送出する例外は、そのアプリケーションに固有なものでもよく、`RuntimeException` を拡張することもできる。

表 1-2 WebLogic RMI の使いやすさ（続き）

機能	WebLogic RMI
プロキシ	リモート オブジェクトのクライアントが使用するクラス。RMI の場合はスケルトン クラスとスタブ クラスが使用される。スタブ クラスは、クライアントの Java 仮想マシン（JVM）で呼び出されるインスタンス。スケルトン クラスはリモート JVM に存在し、リモート JVM 上で呼び出されたメソッドと引数のマーシャリングを解除し、リモート オブジェクトのインスタンスのメソッドを呼び出した後、結果をマーシャリングしてクライアントに返す。
セキュリティ マネージャ	セキュリティ マネージャは不要。WebLogic Server で提供されるすべての WebLogic RMI サービスは、SSL や ACL などのより高度なセキュリティ オプションを備えている。RMI コードを WebLogic RMI コードに変換するときに、setSecurityManager() への呼び出しをコメントアウトできる。
継承	UnicastRemoteObject を拡張する必要がないため、論理オブジェクト階層が保持される。リモート クラスは、rmi.server パッケージ実装を継承するために UnicastRemoteObject を継承する必要がない。リモート クラスにアプリケーション階層内のクラスを継承させつつ、rmi.server パッケージの動作を保持できる。
ツールと管理	RMI レジストリをホストする WebLogic Server は、分散アプリケーションを開発およびデプロイメントするのに十分なツールを備えた環境を提供する。

表 1-3 WebLogic RMI のネーミングとルックアップ

機能	WebLogic RMI
ネーミング	WebLogic JNDI と完全に統合されている。JNDI API またはレジストリ インタフェースを使用して、アプリケーションを意味のあるネーム スペースに分けることができる。JNDI を使用すると、LDAP や NDS のようなエンタープライズ ネーミング サービスを介して RMI オブジェクトをパブリック化できる。
ルックアップ	URL では、標準の rmi:// 方式、https://、iiop://、または http:// を使用する。http:// は、WebLogic RMI の HTTP リクエストをトンネリングし、ファイアウォールを通る場合でも WebLogic RMI のリモート呼び出しを使用可能にする。
クライアントサイドの呼び出し	クライアントからサーバ、クライアントからクライアント、またはサーバからクライアントへの呼び出しをサポートする。クライアントとサーバが最適化済み、多重化、非同期、双方向の接続で接続されている適切な WebLogic Server 環境で動作する。このため、クライアント アプリケーションはレジストリを介してオブジェクトを発行でき、他のクライアントまたはサーバは、クライアント常駐オブジェクトをサーバ常駐オブジェクトとして使用できる。

2 プログラミングの考慮事項

以下の節では、WebLogic Server で使用するための RMI のプログラミングで用いる WebLogic RMI の機能について説明します。

- WebLogic RMI コンパイラ
- WebLogic RMI フレームワーク
- RMI の動的プロキシ
- ホット コード生成
- WebLogic RMI レジストリ
- WebLogic RMI 実装の機能
- RMI と T3 プロトコル

WebLogic RMI コンパイラ

WebLogic RMI コンパイラ（`weblogic.rmic`) は、クライアントサイドでカスタム リモート オブジェクト インタフェースのための動的プロキシを生成し、サーバサイド オブジェクトにホット コード生成を提供します。RMI オブジェクトがデプロイされている場合は、`rmic` を実行すると、実行時にホット コード生成機能によってバイトコードが動的に生成されます。

注意： クラスタ対応クライアントまたは IIOP クライアントの場合は、明示的に `rmic` を実行するだけです（WebLogic RMI over IIOP を使用すると、クライアントは、Internet Inter-ORB Protocol（IIOP）を介して RMI リモート オブジェクトにアクセスできるため、RMI プログラミング モデルが拡張されます）。RMI over IIOP の使用方法の詳細については、『WebLogic RMI over IIOP プログラマーズ ガイド』を参照してください。

動的プロキシ クラスは、クライアントに渡されるシリアライズ可能クラスです。ホット コード生成は、バイトコードを生成する RMI の機能です。このバイトコードは、クライアントの動的プロキシからの要求を処理するサーバサイド クラスです。クラスの実装は、Weblogic Server の RMI レジストリ内の名前に関連付けられます。

クライアントは、レジストリでクラスをルックアップすることによって、そのクラスのプロキシを取得します。クライアントは、あたかもローカルなクラスであるかのようにプロキシのメソッドを呼び出します。プロキシは、要求をシリアライズして WebLogic Server に送ります。動的に生成されたバイトコードはクライアント要求をデシリアライズし、実装クラスに対して実行します。次に結果をシリアライズしてクライアントのプロキシへ送り返します。

動的プロキシとバイトコード

WebLogic Server 6.1 より前のバージョンでは、`rmic` を実行すると、クライアントにはスタブが、サーバサイドにはスケルトン コードが生成されました。今回のバージョンでは、rmic は実行時にロードされる XML デプロイメント記述子を生成します。スタブの代わりに、クライアントは動的プロキシを使用してリモート オブジェクトと通信します。スケルトン クラスは必要時にメモリ内に作成されます。これにより、クラスの生成が不要になりました。

バージョン 6.1 より前の WebLogic RMI オブジェクトをバージョン 6.1 以降の WebLogic Server で実行できるようにするには、それらのオブジェクトに対して `rmic` を再度実行します。これにより、デプロイ済みの RMI オブジェクトを有効にする、必要なプロキシおよびバイトコードが生成されます。

リモート オブジェクトが EJB の場合は、`weblogic.ejbc` を再度実行すると、バージョン 6.1 より前の WebLogic Server オブジェクトがバージョン 6.1 以降で動作するようになります。`weblogic.ejbc` の使用方法の詳細については、『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』を参照してください。

リモート オブジェクトに対して `weblogic.rmic` または `weblogic.ejbc` を再度実行すると、そのオブジェクトのデプロイメント記述子ファイルが生成されます。

WebLogic RMI コンパイラのオプション

WebLogic RMI コンパイラは、Java コンパイラがサポートしているオプションをすべて受け入れます。たとえば、コマンド ラインのコンパイラ オプションに `-d \classes examples.hello.HelloImpl` を追加できます。ほかにも、Java コンパイラがサポートしているすべてのオプションを使用でき、これらのオプションは直接 Java コンパイラに渡されます。

次の表に、`java weblogic.rmic` オプションを示します。これらのオプションは、`java weblogic.rmic` の後、リモート クラス名の前に入力します。

表 2-1 WebLogic RMI コンパイラのオプション

オプション	説明
`-callRouter <callRouterClass>`	`-clusterable` と組み合わせた場合だけ使用可能。ルーティング メソッド呼び出し用に使われるクラスを指定する。このクラスは `weblogic.rmi.cluster.CallRouter` を実装する必要がある。指定した場合、各メソッド呼び出しの前にそのクラスのインスタンスが呼び出され、そのメソッド パラメータに基づいてルーティングするサーバを指定できる。サーバ名または null が返される。null は現在のロード アルゴリズムが使用されることを示す。
`-clusterable`	そのサービスをクラスタ化可能（WebLogic クラスタ内の複数のサーバがホストできる）として指定する。各ホスティング オブジェクト、またはレプリカは、共通名でネーミング サービスにバインドされる。そのサービス プロキシがネーミング サービスから取り出される場合、それはレプリカのリストを保持するレプリカ対応参照を含み、レプリカ間のロード バランシングやフェイルオーバを行う。
`-commentary`	注釈を出力する。
`-dispatchPolicy <queueName>`	サービスが WebLogic Server の実行スレッドを取得するために使う、コンフィグレーション済みの実行キューを指定する。詳細については、「実行キューによるスレッド使用の制御」を参照。

オプション	説明
-help	オプションの説明を表示する。
-idl	リモート インタフェース用の IDL を生成する。
-idlOverwrite	既存の IDL ファイルを上書きする。
-idlVerbose	IDL 情報についての冗長な情報を表示する。
-idlStrict	OMG 規格に従って IDL を生成する。
-idlNoFactories	valuetype のファクトリ メソッドを生成しない。
-idlDirectory <idlDirectory>	IDL ファイルを作成するディレクトリを指定する（デフォルトはカレント ディレクトリ）。
-iiop	サーバから IIOP プロキシを生成する。
-iiopDirectory	IIOP プロキシ クラスが作成されるディレクトリを指定する。
-keepgenerated	WebLogic RMI コンパイラを実行するときに、生成したプロキシ クラスとバイトコードのソースを保持できる。
-loadAlgorithm <algorithm>	-clusterable と組み合わせた場合だけ使用可能。ロード バランシングおよびフェイルオーバに使用する特定のアルゴリズムのサービスを指定する（デフォルトは weblogic.cluster.loadAlgorithm）。ラウンドロビン、ランダム、重みベースの中から 1 つ選択できる。
-methodsAreIdempotent	-clusterable と組み合わせた場合だけ使用可能。このクラスのメソッドが多重呼び出し不変であることを示す。これにより、リモート メソッドが呼び出される前に起きた障害かどうか保証できなくても、通信障害があれば、プロキシはその復旧を試みることができるようになる。_ デフォルトでは（このオプションが使われなければ）、プロキシはリモート メソッドが呼び出された前に起きたことが保証されている障害に関してだけ再試行する。

オプション	説明
-nomanglednames	コンパイル時にリモート クラスに固有のプロキシが作成される。
-replicaListRefreshInterval <seconds>	-clusterable と組み合わせた場合だけ使用可能。クラスタのレプリカ リストをリフレッシュする試みの最小間隔を指定する（デフォルトは 180 秒）。
-stickToFirstServer	-clusterable と組み合わせた場合だけ使用可能。セッション維持型ロード バランシングを有効にする。最初の要求をサービスするために選ばれたサーバが、後続のすべての要求にも使用される。
-version	バージョン情報を出力する。

表 2-2 クラスタ固有の WebLogic RMI コンパイラ オプション

オプション	説明
-callRouter <callRouterClass>	-clusterable と組み合わせた場合だけ使用可能。ルーティング メソッド呼び出し用に使われるクラスを指定する。このクラスは weblogic.rmi.cluster.CallRouter を実装する必要がある。指定した場合、各メソッド呼び出しの前にそのクラスのインスタンスが呼び出され、そのメソッド パラメータに基づいてルーティングするサーバを指定できる。サーバ名または null が返される。null は現在のロード アルゴリズムが使用されることを示す。
-clusterable	そのサービスをクラスタ化可能（WebLogic クラスタ内の複数のサーバがホストできる）として指定する。各ホスティング オブジェクト、またはレプリカは、共通名でネーミング サービスにバインドされる。そのサービス プロキシがネーミング サービスから取り出される場合、それはレプリカのリストを保持するレプリカ対応参照を含み、レプリカ間のロード バランシングやフェイルオーバを行う。

オプション	説明
-loadAlgorithm <algorithm>	-clusterable と組み合わせた場合だけ使用可能。ロード バランシングおよびフェイルオーバに使用する特定のアルゴリズムのサービスを指定する（デフォルトは weblogic.cluster.loadAlgorithm）。ラウンドロビン、ランダム、重みベースの中から 1 つ選択できる。 ロード アルゴリズム名は、-clusterable と組み合せた場合だけ使用可能。ロード バランシングとフェイルオーバを処理するプロキシによって使用されるサービス固有のアルゴリズムを指定する。この引数を指定しない場合、Administration Console で指定されたデフォルトのロード バランシング アルゴリズムが使用される。たとえば、重みベースのロード バランシングを指定するには、次のコマンドを使用する。 **$ java weblogic.rmic -clusterable -loadAlgorithm=weight-based**
-methodsAreIdempotent	-clusterable と組み合せた場合だけ使用可能。同じメソッドを何度も実行することになる場合でも、フェイルオーバの後に再試行ができることをプロキシに指示する。このフラグを指定しない場合、このプロキシ用のメソッドは多重呼び出し不変とは見なされない。これによって扱われる例外については、「クラスタ化されたオブジェクトのフェイルオーバ」を参照。
-replicaListRefreshInterval <seconds>	-clusterable と組み合わせた場合だけ使用可能。クラスタのレプリカ リストをリフレッシュする試みの最小間隔を指定する（デフォルトは 180 秒）。
-stickToFirstServer	-clusterable と組み合わせた場合だけ使用可能。セッション維持型ロード バランシングを有効にする。最初の要求をサービスするために選ばれたサーバが、後続のすべての要求にも使用される。

クラスタにおけるスタブのレプリケーション

レプリケートされないスタブをクラスタ内に生成することもできます。このようなスタブは、「固定」サービスと呼ばれています。これらのスタブは登録されたホストからのみ使用可能であり、透過的なフェイルオーバやロード バランシングは提供しません。

`weblogic.rmic` でクラスタ化オプションを使用して RMI オブジェクトをコンパイルし、3 ノードのサーバ クラスタ（A、B、C）の 2 つのノード（A および B）にオブジェクトをデプロイし、バインドを 3 つのノードすべてにレプリケートする場合、各サーバから同じビューが得られます。3 つのノードすべてに対して JNDI ルックアップを行うと同じスタブが得られ、メソッド呼び出しを行うと、サーバは最初の 2 つのノード間でロード バランシングを実行します。

したがって、クラスタ化可能オプションを指定して RMI オブジェクトをコンパイルし、バインド セットのレプリケーションを「false」に設定してそれらのオブジェクトを JNDI ツリーにバインドする場合、JNDI ルックアップを行うと次の結果が得られます。

- サーバ A 上では、サーバ A を指すスタブが得られる
- サーバ B 上では、サーバ B を指すスタブが得られる
- サーバ C 上では NameNotFoundException が返される

サーバ A に対してリモート呼び出しを行い、サーバが使用不可になっていることによって呼び出しが失敗する場合、クラスタ化可能スタブは再ルックアップを実行し、呼び出しのルーティング先に応じて次のいずれかの結果が発生します。

- サーバ B 上では、サーバ B を指すスタブが得られる
- サーバ C 上では NameNotFoundException が返される

クラスタ非対応の RMI オブジェクトを、バインド セットのレプリケーションを `false` に設定して JNDI ツリーにバインドする場合、JNDI ルックアップを行うと、固定スタブが返されフェイルオーバは行われません。固定サービスは、レプリケートされたクラスタ全体の JNDI ツリーにバインドされるため、クラスタ全体で利用可能です。ただし、固定サービスのホストである個別のサーバで障害が発生すると、クライアントは別のサーバへのフェイルオーバを行うことができません。

クラスタ非対応の RMI オブジェクトを、バインド セットのレプリケーションを `true` に設定して JNDI ツリーにバインドすると、バインドは失敗します。その理由は、オブジェクトがクラスタ非対応であり、クラスタ内でクラスタ非対応のサービスを提供できるのは 1 つのサーバに限られるためです。

WebLogic RMI フレームワーク

WebLogic RMI はクライアントとサーバのフレームワークに分けられています。実行時のクライアントはサーバのソケットを持たないため、接続をリスンしていません。クライアントはサーバを介して接続を取得します。サーバだけがクライアントのソケットを認識します。このため、クライアントにあるリモート オブジェクトをホストする場合、クライアントは WebLogic Server に接続していなければなりません。WebLogic Server はクライアントへのリクエストを処理して、クライアントへ情報を渡します。つまり、クライアントサイドの RMI オブジェクトには、クラスタ内であっても単一の WebLogic Server を介してのみアクセスできます。クライアント サイドの RMI オブジェクトが JNDI ネーミング サービスにバインドされている場合、そのオブジェクトへのアクセスは、そのバインドを実行したサーバにアクセスできる場合に限り可能です。

WebLogic RMI コンパイラのその他の機能

WebLogic RMI コンパイラには、他にも以下のような機能があります。

- リモート メソッドのシグネチャは `RemoteException` を送出する必要がありません。
- リモートの例外は `RuntimeException` にマッピングできます。
- リモート クラスは、非リモート インタフェースも実装できます。

RMI の動的プロキシ

動的プロキシまたはプロキシは、リモート オブジェクトのクライアントが使用するクラスです。このクラスは、作成されるときに、実行時に指定されたインタフェースのリストを実装します。

RMI では、動的に生成されたバイトコードとプロキシ クラスが使用されます。プロキシ クラスは、クライアントの Java 仮想マシン（JVM）で呼び出されるインスタンスです。プロキシ クラスは、呼び出されたメソッド名とその引数をマーシャリングして、リモートの JVM に転送します。リモート呼び出しが終了して返された後に、プロキシ クラスはクライアント上でその結果のマーシャリングを解除します。生成されたバイトコードはリモート JVM に存在し、リモート JVM 上で呼び出されたメソッドと引数のマーシャリングを解除し、リモート オブジェクトのインスタンスのメソッドを呼び出した後、結果をマーシャリングしてクライアントに返します。

WebLogic RMI コンパイラとプロキシの使い方

WebLogic RMI コンパイラは、デフォルトの動作により、リモート インタフェース用のプロキシと、そのプロキシを共有するリモート クラス用のプロキシを作成します。プロキシは、リモート オブジェクトのクライアントが使用するクラスです。RMI では、動的に生成されたバイトコードとプロキシ クラスが使用されます。

たとえば、WebLogic RMI コンパイラでは、`example.hello.HelloImpl` と `counter.example.CiaoImpl` は、一対のプロキシ クラスとバイトコード、つまりリモート オブジェクト（このサンプルでは `example.hello.Hello`）によって実装されたリモート インタフェースに適合するプロキシで表わされます。

リモート オブジェクトが複数のインタフェースを実装する場合、プロキシの名前とパッケージは 1 組のインタフェースをエンコードすることによって決定されます。WebLogic RMI コンパイラの `-nomanglednames` というオプションを使って、デフォルトの動作をオーバーライドできます。このオプションを使用すると、コンパイル時にリモート クラスに固有のプロキシが作成されます。クラス固有のプロキシが検出された場合は、そのプロキシはインタフェース固有のプロキシに優先します。

さらに、WebLogic RMI のプロキシ クラスでは、プロキシは final ではありません。同じ場所に配置されたリモート オブジェクトへの参照は、プロキシではなくオブジェクトそのものへの参照です。

ホット コード生成

rmic を実行すると、WebLogic Server のホット コード生成機能により、メモリ内にサーバ クラス用のバイトコードが自動生成されます。バイトコードは、リモート オブジェクトの必要に応じて、動的に生成されます。現在のバージョンの WebLogic Server では、weblogic.rmic を実行しても、オブジェクトのスケルトン クラスは生成されません。

WebLogic RMI レジストリ

WebLogic Server は、RMI レジストリのホストとなり、RMI クライアント用のサーバ インフラストラクチャを提供します。レジストリ トラフィックは JDBC などのトラフィックと同じ接続で多重化されるので、RMI レジストリとサーバの通信オーバーヘッドは最小限に抑えられます。クライアントは RMI 用に単一のソケットを使用します。このため、WebLogic 環境での RMI クライアントのスケーラビリティは直線的です。

WebLogic RMI レジストリは WebLogic Server の起動時に作成され、新しいレジストリを作成するための呼び出しを行うだけで既存のレジストリの位置を見つけることができます。レジストリにバインドされているオブジェクトには、標準の rmi:// のほかに、http:// や https:// などのクライアント プロトコルを使ってアクセスできます。実際のところ、すべてのネーミング サービスで JNDI が使用されます。

WebLogic RMI 実装の機能

一般に、java.rmi パッケージに含まれているすべてのメソッドと同じ機能のメソッドが WebLogic RMI で提供されています（RMIClassLoader 内のメソッドと java.rmi.server.RemoteServer.getClientHost() メソッドを除く）。

他のすべてのインタフェース、例外、およびクラスは WebLogic RMI でサポートされています。以下の節に、特定の実装に関する注意事項を示します。

JNDI

WebLogic RMI では、オブジェクトのネーミングの望ましいメカニズムとして、Java Naming and Directory Interface（JDNI）を使用します。JNDI は、Java アプリケーションにネーミング サービスを提供するアプリケーション プログラミング インタフェース（API）です。JNDI は、Sun Microsystems の Java 2 Enterprise Edition（J2EE）技術の不可欠なコンポーネントです。ネーミング サービスは名前をオブジェクトに関連付けて、指定された名前に基づいてオブジェクトを見つけます（RMI レジストリは、ネーミング サービスの一例です）。

RMI で JNDI を使用すると、より効率的に分散プログラミングを行うことができます。ただし、リモート クライアントとサーバ間の往復回数には注意する必要があります。クライアントとサーバ間で JNDI ルックアップが繰り返されると、パフォーマンス上の問題が発生する場合があります。

rmi.RMISecurityManager

rmi.RMI SecurityManager は非最終クラスとして実装され、すべてのパブリック メソッドが WebLogic RMI でサポートされます。制約のある JavaSoft 参照実装とは異なり、まったく制約がありません。WebLogic RMI のセキュリティはより大きな WebLogic 環境の重要な一部で、SSL （Secure Socket Layer）と ACL（アクセス制御リスト）がサポートされています。

rmi.registry.LocateRegistry

rmi.registry.LocateRegistry は最終クラスとして実装され、すべてのパブリック メソッドがサポートされます。ただし、LocateRegistry.createRegistry(int port) を呼び出すと、レジストリが同じ場所に作成されるのではなく、JNDI を実装するサーバサイド インスタンスへの接続が試みられます。JNDI には属性によってホストとポートが指定されます。WebLogic RMI では、クライアントは、このメソッドを呼び出して WebLogic Server 上の JNDI ツリーを検索できます。

注意： デフォルト（rmi）以外のプロトコルを使うことも可能で、次に示すように方式、ホスト、およびポートを URL として使うこともできます。

```
LocateRegistry.getRegistry(https://localhost:7002);
```

この例では、標準 SSL WebLogic プロトコルを使用してローカル ホスト上のポート 7002 にある WebLogic Server レジストリの位置を検索します。

rmi.server クラス

rmi.server.LogStream は、write(byte[]) メソッドが WebLogic Server のログ ファイルを通してメッセージを記録する点で JavaSoft 参照実装とは異なります。

rmi.server.RemoteObject は、UnicastRemoteObject と同じ型を保存するために WebLogic RMI に実装されますが、その機能は WebLogic RMI の基本クラスである proxy によって提供されます。

rmi.server.RemoteServer は、rmi.server.UnicastRemoteObject の抽象スーパークラスとして実装され、getClientHost() を除くすべてのパブリック メソッドが WebLogic RMI でサポートされています。

rmi.server.UnicastRemoteObject は、リモート オブジェクトの基本クラスとして実装され、このクラスのすべてのメソッドは WebLogic RMI の基本クラスである Proxy という名前で実装されます。この結果、プロキシは final ではない Object メソッドをオーバーライドでき、実装についてまったく条件を付けずにこれらのメソッドを実装したのと同じ状態になります。

WebLogic RMI では、呼び出し側のオブジェクトが RMI オブジェクトと同じ Java 仮想マシン（JVM）内に存在しない場合、メソッド パラメータはすべて値渡しです。存在する場合、メソッド パラメータは参照渡しです。

注意： WebLogic RMI は、クライアントからのクラスのアップロードをサポートしません。つまり、リモート オブジェクトに渡されるどのクラスも、サーバの CLASSPATH 内に存在しなければなりません。

setSecurityManager

setSecurityManager() メソッドは、コンパイルの互換性を保持するためだけに WebLogic RMI に組み込まれています。WebLogic RMI は WebLogic Server 内のより一般的なセキュリティ モデルに依存しているため、このメソッドにはセキュリティは関連付けられていません。セキュリティ マネージャを設定する場合、1 つしか設定できません。セキュリティ マネージャを設定する前に、セキュリティ マネージャが既に設定されているかどうかをテストする必要があります。セキュリティ マネージャを重複して設定しようとすると、例外が送出されます。次に例を示します。

```
if (System.getSecurityManager() == null)
System.setSecurityManager(new RMISecurityManager());
```

使用されないクラス

以下のクラスは、実装されていますが、WebLogic RMI では使用されません。

- rmi.dgc.Lease
- rmi.dgc.VMID
- rmi.server.ObjID
- rmi.server.Operation
- rmi.server.RMIClassLoader
- rmi.server.RMISocketFactory
- rmi.server.UID

RMI と T3 プロトコル

WebLogic Server における RMI 通信では、T3 プロトコルが使用されます。T3 は、WebLogic Server とそれ以外の Java プログラム (クライアントや他の WebLogic Server など) の間でのデータ転送に用いられる最適化されたプロトコルです。サーバ インスタンスは、接続先の各 Java 仮想マシン (JVM) の動作を追跡し、JVM のすべてのトラフィックを伝送するための T3 接続を 1 つだけ作成します。

たとえば、Java クライアントが WebLogic Server 上のエンタープライズ Bean や JDBC 接続プールにアクセスする場合、WebLogic Server の JVM とクライアントの JVM との間に単一のネットワーク接続が確立されます。T3 プロトコルでは、単一接続上のパケットが目に見えない形で多重化されるため、あたかも専用のネットワーク接続を独占的に使用しているかのように、EJB サービスや JDBC サービスを記述することができます。

有効な T3 接続が確立された任意の 2 つの Java プログラム (たとえば、2 つのサーバ インスタンスや、サーバ インスタンスと Java クライアント) では、ポイント ツー ポイントの定期的な「ハートビート」を使用して、自分が引き続き利用可能であることの通知や、相手が利用可能かどうかの判断を行います。各エンドポイントでは、ピア (通信相手) に定期的にハートビートを送ると共に、ピアから引き続きハートビートを受信しているかどうかでピアがまだ利用可能かどうかを判断します。

サーバ インスタンスがハートビートを出す頻度は、*ハートビート間隔*で決まります。デフォルトでは 60 秒です。

ピアから何回ハートビートが届かなかったらピアが利用できないとサーバ インスタンス側が判断するかは、*ハートビート タイムアウト期間*で決まります。デフォルトでは 4 です。

したがって、各サーバ インスタンスでは、ピアからメッセージ (ハートビートかそれ以外の通信) がない場合、最大 240 秒間すなわち 4 分間待ってから、ピアが到達不能と判断します。

タイムアウトのデフォルト設定を変更することはお勧めしません。

3 WebLogic RMI の実装

以下の節では、WebLogic RMI API について説明します。

- WebLogic RMI API の概要
- WebLogic RMi の実装の手順

WebLogic RMI API の概要

WebLogic RMI の一部として、いくつかのパッケージが WebLogic Server に付属しています。パブリック API には以下のものが含まれています。

- RMI 基本クラスの WebLogic 実装
- レジストリ
- サーバのパッケージ
- WebLogic RMI コンパイラ
- パブリック API に含まれないサポート クラス

既に RMI クラスを記述している場合は、リモート インタフェースとそれを拡張するクラスで import 文を変えるだけで WebLogic RMI をインストールできます。クライアント アプリケーションにリモート呼び出しを追加するには、レジストリ内でオブジェクトを名前でルックアップします。

すべてのリモート オブジェクトの基本単位は `java.rmi.Remote` インタフェースで、これにはメソッドが含まれていません。この「タグ付け」インタフェースを拡張し（リモート クラスを識別するタグとして機能するように）、リモート オブジェクトの構造を作成するメソッドを使って、使用するリモート インタフェースを作成します。次に、リモート クラスを使ってリモート インタフェースを実装します。この実装はレジストリ内の名前にバインドされ、クライアントやサーバはそこからオブジェクトをルックアップしてリモートで使用できます。

JavaSoft の RMI の参照実装の場合と同様に、`java.rmi.Naming` クラスは重要なクラスです。このクラスには、レジストリ内のリモート オブジェクトに名前をバインド、アンバインド、リバインドするメソッドが含まれています。また、クライアントからレジストリ内の名前付きリモート オブジェクトにアクセスするための `lookup()` メソッドも含まれています。

さらに、WebLogic JNDI はネーミング サービスとルックアップ サービスを提供します。WebLogic RMI は、JNDI によるネーミングとルックアップもサポートしています。

WebLogic RMI 例外は `java.rmi` 例外と同じ機能を備えているので、既存のインタフェースと実装で例外の処理方法を変える必要がありません。

WebLogic RMi の実装の手順

以下の節では WebLogic Server RMI の実装方法について説明します。

- リモートで呼び出すことができるクラスを作成する
  - 手順 1. リモート インタフェースを作成する
  - 手順 2. リモート インタフェースを実装する
  - 手順 3. Java クラスをコンパイルする
  - 手順 4. 実装クラスを RMI コンパイラでコンパイルする
  - 手順 5. リモート メソッドを呼び出すコードを記述する
- 完全なコード例

リモートで呼び出すことができるクラスを作成する

独自の WebLogic RMI クラスを簡単な手順で記述できます。次に、その単純な例を示します。

手順 1. リモート インタフェースを作成する

リモートで呼び出せるすべてのクラスは、リモート インタフェースを実装します。Java コードのテキスト エディタを使用し、以下のガイドラインに従ってリモート インタフェースを記述します。

- リモート インタフェースは、`java.rmi.Remote` インタフェースを拡張しなければなりませんが、これにはメソッド シグネチャが含まれていません。インタフェースを実装する各リモート クラスで実装されるメソッド シグネチャを含めます。インタフェースの作成方法については、Sun Microsystems JavaSoft チュートリアルの「Creating Interfaces」を参照してください。
- リモート インタフェースはパブリックでなければなりません。パブリックでない場合、クライアントはリモート インタフェースを実装するリモート オブジェクトをロードしようとするとエラーを受け取ります。
- JavaSoft の RMI とは異なり、インタフェース内の各メソッドがその `throws` ブロックで `java.rmi.RemoteException` を宣言する必要はありません。アプリケーションが送出する例外は、そのアプリケーションに固有なものでもよく、`RuntimeException` を拡張することも可能です。WebLogic RMI は、サブクラス `java.rmi.RemoteException` を作成するため、既存の RMI クラスがある場合は、例外処理を変更する必要がありません。
- リモート インタフェースにはコードがあまり記述されていない場合もあります。必要なのは、リモート クラスで実装するメソッドに対するメソッド シグネチャだけです。

  以下の例に、メソッド シグネチャ `sayHello()` が含まれたリモート インタフェースを示します。

```
package examples.rmi.multihello;
import java.rmi.*;
public interface Hello extends java.rmi.Remote {
  String sayHello() throws RemoteException;
}
```

JavaSoft の RMI では、リモート インタフェースを実装するすべてのクラスには、コンパイル済みのプロキシが存在しなければなりません。WebLogic RMI は、柔軟性の高い実行時のコード生成をサポートします。つまり、WebLogic RMI は、型さえ正しければ、そのほかはインタフェースを実装するクラスに依存しない動的プロキシと動的に生成されるバイトコードもサポートします。クラスが 1 つのリモート インタフェースを実装する場合、コンパイラが生成したプロキシとバイトコードは、リモート インタフェースと同じ名前になります。クラスが複数のリモート インタフェースを実装する場合、コード生成の結果できるプロキシとバイトコードの名前は、コンパイラが使う名前の区切り方によって変わります。

手順 2. リモート インタフェースを実装する

Java コードのテキスト エディタを使用して、リモートで呼び出されるクラスを記述します。このクラスは、手順 1 で記述したリモート インタフェースを実装する必要があります。これは、インタフェースに含まれるメソッド シグネチャを実装したことを意味します。現在のリリースでは、WebLogic RMI で行われるコード生成はこのクラスファイルに依存します。

WebLogic RMI では、クラスは JavaSoft RMI で必須の `UnicastRemoteObject` を拡張する必要はありません。このため、アプリケーションに適用しやすいクラス階層を維持できます。

クラスは、複数のリモート インタフェースを実装できます。また、クラスにはリモート インタフェースにないメソッドも定義できますが、このようなメソッドはリモートで呼び出せません。

次の例では、複数の `HelloImpls` を作成するクラスを実装し、それぞれをレジストリ内の固有の名前にバインドします。メソッド `sayHello()` は、ユーザに「Hello」とあいさつし、リモートで呼び出されたオブジェクトを識別します。

```
package examples.rmi.multihello;
import java.rmi.*;
public class HelloImpl implements Hello {
  private String name;
  public HelloImpl(String s) throws RemoteException {
    name = s;
  }
```

```
  public String sayHello() throws RemoteException {
    return "Hello!From " + name;
  }
```

次に、リモート オブジェクトのインスタンスを作成する main() メソッドを記述し、それを名前（オブジェクトの実装を指し示す URL）にバインドすることによって WebLogic RMI のレジストリに登録します。リモートでオブジェクトを使用するためにプロキシを取得しようとするクライアントは、オブジェクトを名前でルックアップできます。

RMI レジストリによって除外される文字列名の構文は次のとおりです。

```
rmi://hostname:port/remoteObjectName
```

hostname（ホスト名）と port（ポート）は RMI レジストリが動作しているマシンとポートを識別し、remoteObjectName はリモート オブジェクトの文字列名です。ホスト名、ポート、およびプレフィックスの「rmi:」は省略可能です。ホスト名を指定しない場合、WebLogic Server はデフォルトでローカル ホストになります。ポートを指定しない場合、1099 番ポートが使用されます。remoteObjectName を指定しない場合、名前が付けられているオブジェクトは RMI レジストリ自体です。

詳細については、RMI 仕様を参照してください。

次に、HelloImpl クラス用の main() メソッドの例を示します。この例では、WebLogic Server レジストリに MultiHelloServer という名前で HelloImpl オブジェクトを登録します。

```
  public static void main(String[] argv) {
    // 以下のコードは WebLogic RMI では不要
    // System.setSecurityManager(new RmiSecurityManager());
    // しかし、このコードを含める場合には、以下のように
    // 条件文にする必要がある
    // if (System.getSecurityManager() == null)
    // System.setSecurityManager(new RmiSecurityManager());
    int i = 0;
    try {
      for (i = 0; i < 10; i++) {
        HelloImpl obj = new HelloImpl("MultiHelloServer" + i);
        Naming.rebind("//localhost/MultiHelloServer" + i, obj);
```

```
        System.out.println("MultiHelloServer" + i + " created.");
      }
      System.out.println("Created and registered " + i +
                         " MultiHelloImpls.");
    }
    catch (Exception e) {
      System.out.println("HelloImpl error:" + e.getMessage());
      e.printStackTrace();
    }
  }
```

WebLogic RMI では、アプリケーションにセキュリティを統合するためにセキュリティ マネージャを設定する必要はありません。セキュリティは、WebLogic の SSL と ACL のサポートによって処理されます。必要な場合は独自のセキュリティ マネージャを使うこともできますが、それを WebLogic Server にインストールしないでください。

手順 3. Java クラスをコンパイルする

javac または他の Java コンパイラを使用して .java ファイルをコンパイルし、リモート インタフェースとそれを実装するクラス用の .class ファイルを作成します。

手順 4. 実装クラスを RMI コンパイラでコンパイルする

WebLogic RMI コンパイラ（weblogic.rmic）を実行するには、次のコマンド パターンを使用します。

```
$ java weblogic.rmic nameOfRemoteClass
```

nameOfRemoteClass は、リモート インタフェースを実装するクラスの完全なパッケージ名です。上で使用した例では、このコマンドは次のようになります。

```
$ java weblogic.rmic examples.rmi.hello.HelloImpl
```

生成したプロキシとバイトコードのソースを保持する場合は、WebLogic RMI コンパイラを実行するときに -keepgenerated フラグを設定します。使用可能な WebLogic RMI コンパイラ オプションの一覧については、2-3 ページの「WebLogic RMI コンパイラのオプション」を参照してください。

WebLogic RMI コンパイラを実行すると、プロキシとバイトコードが動的に生成されます。プロキシは、nameOfInterface_Proxy.class というクラスになります。作成される 3 つのファイル、つまりリモート インタフェース、リモート インタフェースを実装するクラス、およびプロキシは、WebLogic RMI コンパイラによって生成されるバイトコードとともに、オブジェクトの main() メソッドのネーミング方式で使用した URL を持つ WebLogic Server の CLASSPATH の該当ディレクトリに入れる必要があります。

手順 5. リモート メソッドを呼び出すコードを記述する

リモート クラス、リモート クラスが実装するインタフェース、およびそのプロキシとバイトコードをコンパイルして WebLogic Server にインストールしたら、Java コードのテキスト エディタを使用し、WebLogic クライアント アプリケーションにコードを追加してリモート クラスのメソッドを呼び出すことができます。

通常、そのためにはコードを 1 行だけ記述します。つまり、リモート オブジェクトへの参照を取得します。これは、Naming.lookup() メソッドで行います。次に、上の例で作成したオブジェクトを使用する短い WebLogic クライアント アプリケーションの例を示します。

```
package mypackage.myclient;
import java.rmi.*;

public class HelloWorld throws Exception {

  // WebLogic のレジストリ内の
  // リモート オブジェクトをルックアップする
  Hello hi = (Hello)Naming.lookup("HelloRemoteWorld");
  // メソッドをリモートで呼び出す
  String message = hi.sayHello();
  System.out.println(message);
}
```

この例では、クライアントとして Java アプリケーションを使用しています。

完全なコード例

次に、Hello インタフェースの完全なコードを示します。

```
package examples.rmi.hello;
import java.rmi.*;

public interface Hello extends java.rmi.Remote {

  String sayHello() throws RemoteException;

}
```

次に、このインタフェースを実装する HelloImpl クラスの完全なコードを示します。

```
package examples.rmi.hello;

import java.rmi.*;

public class HelloImpl
    // 以下は WebLogic RMI では不要
    // extends UnicastRemoteObject
    implements Hello {

  public HelloImpl() throws RemoteException {
    super();
  }

  public String sayHello() throws RemoteException {
    return "Hello Remote World!!";
  }

  public static void main(String[] argv) {
```

```
    try {
      HelloImpl obj = new HelloImpl();
      Naming.bind("HelloRemoteWorld", obj);
    }
    catch (Exception e) {
      System.out.println("HelloImpl error:" + e.getMessage());
      e.printStackTrace();
    }
  }
}
```