Borland AppServer™ 6.7
開発者ガイド

Borland Software Corporation
20450 Stevens Creek Blvd., Suite 800
Cupertino, CA 95014 USA
www.borland.com

BAS67DevGuide
2006 年 12 月

Borland®

目次

第 46 章

管理オブジェクトの要素および属性 419

第 1 章

Borland AppServer の概要

Borland AppServer（AppServer）は、企業環境において分散エンタープライズ アプリケーションの開発、デプロイメント、管理を行うための、サービスやツールのセットです。

AppServer は J2EE 1.4 標準の先進実装製品であり、EJB 2.1、JMS 1.1、Servlet 2.4、JSP 2.0、CORBA 2.6、XML、SOAP などの最新の業界標準技術をサポートします。ボーランドは、2 つのバージョンの AppServer を提供しており、これには、「Java メッセージング サービス（JMS）」に対する最先端のエンタープライズ メッセージング ソリューション（Tibco と OpenJMS）がそれぞれ同梱されています。ユーザーは、AppServer で必要とする機能やサービスのレベルを選択することができ、それらを変更する必要がある場合には、ライセンスをアップグレードすることにより容易に対応できます。

AppServer を利用することにより、J2EE 1.4 プラットフォーム標準を実装した分散 Java/CORBA アプリケーションを安全にデプロイし、さまざまな側面から管理することができます。

AppServer では、インストールごとのサーバー インスタンスの数は無制限です。そのため、同時接続ユーザーの数は無制限です。

AppServer は次のコンポーネントを備えています。

- J2EE 1.4 の実装。
- Apache Web Server バージョン 2.2.3。
- Borland Security。AppServer のセキュリティのためのフレームワークを提供します。
- 先進の集中管理型 JMS 管理ソリューション（Tibco および OpenJMS）。AppServer に同梱されています。
- 分散コンポーネントのための強力な管理ツール群。AppServer の外部で開発されたアプリケーションも含まれます。

AppServer の機能

AppServer では次の機能が提供されます：

- BAS プラットフォームに対するサポート（AppServer に対してサポートされているプラットフォームのリストについては、http://support.borland.com/kbcategory.jspa?categoryID=389 を参照してください）。
- クラスタリング トポロジーに対する完全サポート。
- VisiBroker ORB インフラストラクチャとのシームレスな統合。
- Borland JBuilder 統合開発環境（IDE）との統合。
- 他のボーランド製品（Borland Optimizeit Profiler や ServerTrace など）との統合の強化。
- AppServer により、既存のアプリケーションを Web サービスとして公開したり、新しいアプリケーションや追加 Web サービスと統合することができます。Borland Web サービスは、Apache Axis 1.2 テクノロジー（SOAP 1.2 をサポートする次世代 Apache SOAP サーバー）をベースとしています。

Borland AppServer のドキュメント

AppServer 関連のドキュメントには次のものがあります：

- **『Borland AppServer インストール ガイド』**：AppServer をネットワーク上にインストールする方法について説明されています。これは、Windows、UNIX の各オペレーティング システムに精通しているシステム管理者の方を対象に書かれています。
- **『Borland AppServer 開発者ガイド』**：運用環境における分散オブジェクト ベース アプリケーションのパッケージング、デプロイメント、管理についての詳細情報が記載されています。
- **『Borland 管理コンソール ユーザーズ ガイド』**：Borland 管理コンソール GUI の使用方法についての情報が記載されています。
- **『Borland セキュリティ ガイド』**：VisiSecure for VisiBroker for Java や VisiSecure for VisiBroker for C++ など、AppServer のセキュリティを確保するためのボーランドのフレームワークについて説明されています。
- **『Borland VisiBroker for Java 開発者ガイド』**：Java による VisiBroker アプリケーションの開発方法について説明されています。本書により VisiBroker ORB の設定と管理、プログラミング ツールの使用方法に精通できるよう、記載されています。また、IDL コンパイラ、スマート エージェント、ロケーション サービス、ネーミング サービス、イベント サービス、オブジェクト アクティベーション デーモン（OAD）、サービス品質（QoS: Quality of Service）、インターフェース リポジトリについても説明されています。
- **『Borland VisiBroker VisiTransact ガイド』**：OMG オブジェクト トランザクション サービス仕様に対するボーランドの実装、および、ボーランドのトランザクション サービス統合コンポーネントについて説明されています。

通常、ドキュメントにアクセスするには、AppServer 製品と共にインストールされるヘルプ ビューアを使用します。ユーザーは、スタンドアロンのヘルプ ビューアから、もしくは AppServer GUI ツールから、ヘルプを参照することができます。どちらの場合も、独立したウィンドウ内にヘルプ ビューアが起動されるため、ナビゲーション ペインを利用できるだけでなく、ナビゲーションや印刷のためのヘルプ ビューアのメイン ツールバーも利用することができます。ヘルプ ビューアのナビゲーション ペインには、すべての AppServer ドキュメントや参考ドキュメントの目次、インデックス、包括的な検索を実行できるページがあります。

PDF 形式の『Borland AppServer 開発者ガイド』や『Borland 管理コンソール ユーザーズ ガイド』は、http://info.borland.com/techpubs/appserver より入手可能です。

AppServer オンライン ヘルプ トピックへのアクセス

オンライン ヘルプにアクセスするには（次のいずれかの方法を利用）:

Windows の場合

- [スタート | すべてのプログラム | Borland AppServer | Help Topics] を選択。
- または、Web ブラウザを起動し、`<AppServer_Home>/doc/index.html` を開く。

UNIX の場合

- Web ブラウザを起動し、`<AppServer_Home>/doc/index.html` を開く。

AppServer GUI ツールから AppServer オンライン ヘルプ トピックへのアクセス

AppServer GUI ツールからオンライン ヘルプにアクセスするには（次のいずれかの方法を利用）:

- Borland 管理コンソールから、[Help | Help Topics] を選択。
- Borland デプロイメント ディスクリプタ エディタ（DDEditor）から、[Help | Help Topics] を選択。

ドキュメント表記規則

AppServer のドキュメントでは、文中の特定の部分を表すために、次の表に示す書体や記号を使用しています :

表記規則	用途
ボールド	新規の用語およびドキュメント名に使用されます。
`computer`	ユーザーやアプリケーションが提供する情報、 サンプル コマンドライン、 およびコードです。
`bold computer`	本文では、 ユーザーが入力する情報を示します。 サンプル コードでは、 重要な文章を強調表示します。
[ ]	省略可能な項目であることを示します。
...	直前の引数が繰り返し可能であることを示します。
|	二者択一であることを示します。

プラットフォームの表記規則

AppServer のドキュメントでは、プラットフォーム固有の情報を表すために、次の記号を使用しています :

記号	意味
Windows	サポートされているすべての Windows プラットフォーム
Win2003	Windows 2003 のみ
WinXP	Windows XP のみ
Win2000	Windows 2000 のみ
UNIX	サポートされているすべての UNIX プラットフォーム
Solaris	Solaris のみ

Borland サポートへのお問い合わせ

ボーランド社は各種のサポート オプションを提供しています。それらには、インターネット上からの無償サービスもあり、大規模な情報データベースを検索したり、他のボーランド製品ユーザーからの情報を得たりすることが可能です。また、ボーランド製品のインストールに関するサポートから、有償のコンサルタント レベルのサポート、および高レベルなアシスタンスに至るまでの複数のカテゴリから、電話サポートの種類を選択できます。

ボーランドのサポート サービスについての詳細情報の入手や、実際にテクニカル サポートへお問い合わせいただくには、Web サイト http://support.borland.com を参照の上、製品をお使いになっている地域を選択してください。

ボーランド社のサポートへの連絡にあたっては、次の情報をご用意ください。

- 名前
- 会社名およびサイト ID
- 電話番号
- ユーザー ID（米国のみ）
- オペレーティング システムおよびバージョン
- ボーランド製品名およびバージョン
- 適用済みのパッチまたはサービス パック
- クライアントの言語とそのバージョン（使用している場合）
- データベースとそのバージョン（使用している場合）
- 発生した問題の詳細な内容と経緯
- 問題を示すログファイル
- 発生したエラー メッセージまたは例外の詳細な内容

オンライン リソース

ネットワーク上の次のサイトから情報を得ることができます。

ワールド ワイド ウェブ： http://www.borland.com
オンライン サポート： http://support.borland.com （ユーザー ID が必要）

ワールド ワイド ウェブ

http://www.borland.com は、定期的にご確認ください。AppServer 製品チームによる、ホワイト ペーパー、競合製品の分析、FAQ への回答、サンプル アプリケーション、更新ソフトウェア、更新ドキュメント、および新旧製品に関する情報が掲載されています。

特に、次の URL を確認されることをお勧めします：

- http://www.borland.com/downloads/download_appserver.html （AppServer ソフトウェアおよびその他のファイル）
- http://support.borland.com （AppServer FAQ）

Borland ニュースグループ

AppServer を対象とした数多くのスレッド化されたディスカッション グループに参加することができます。Enterprise Server やその他のボーランド製品に関する、ユーザー主体のニュースグループへ参加するには、http://www.borland.com/newsgroups を参照してください。

メモ これらのニュースグループはユーザーによって管理されているものであり、ボーランド社の公式サイトではありません。

第 2 章

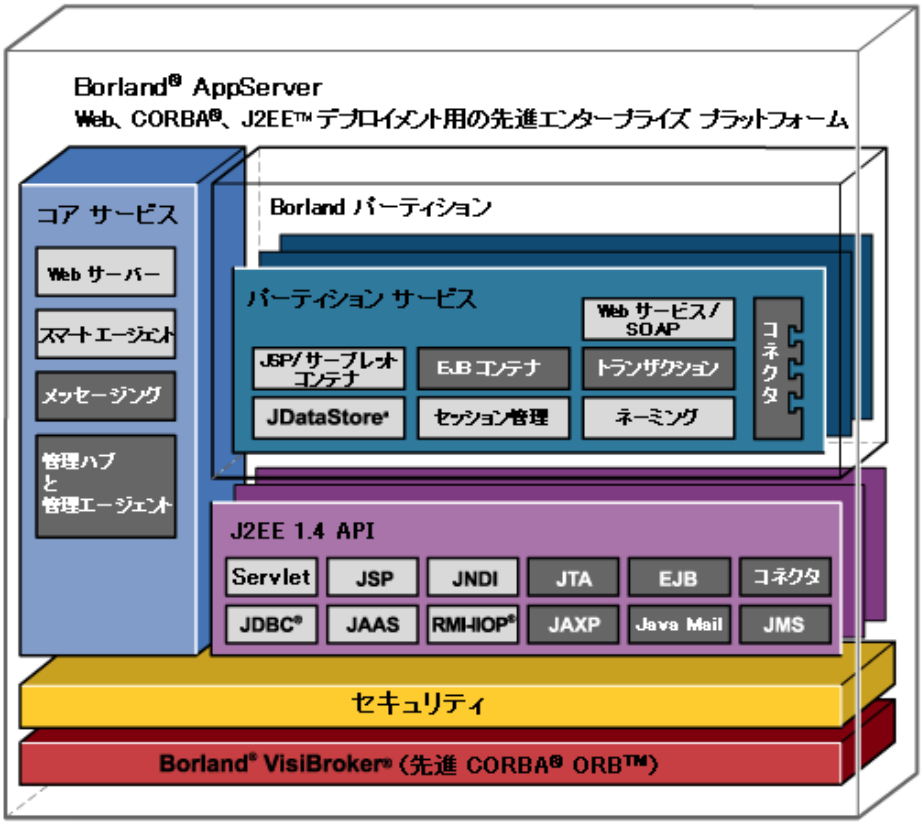
Borland AppServer の概要とアーキテクチャ

ここでは、Borland AppServer（AppServer）の製品について概説します。

AppServer アーキテクチャの概要

AppServer は CORBA ベースで、アーキテクチャを介して分散オブジェクトを利用する J2EE サーバーです。さらに、企業のメインフレームから、小規模なビジネスアプリケーションやリモート型データベースを搭載した異種システムまで、あらゆるプラットフォームと接続することができます。エンタープライズアプリケーションのパッケージングや、デプロイメントデスクリプタによるアプリケーションモジュールの記述内容に基づいて、AppServer のコンポーネントはエンタープライズアプリケーションを処理します。

次の図のアーキテクチャ構造で、AppServer の 1 番上がエンタープライズアプリケーションです。アプリケーションサーバーインストールには、AppServer *コアサービス*と*パーティション*があります。



AppServer サービスの概要

AppServer サービスは、AppServer でサービスを受けるすべてのアプリケーションに共通して利用できるサービスです。

- Web サーバー
- Java メッセージ（JMS）
- スマートエージェント
- 2PC トランザクションサービス

Web サーバー

AppServer には、Apache Web Server バージョン 2.3 が組み込まれています。Apache Web Server は、堅固な市販製品クオリティの HTTP プロトコルリファレンスインプリメンテーションです。サードパーティのモジュールを追加することで、Apache Web Server は高度に設定および拡張することができます。Apache は洗練された機能性でクライアントをサポートするだけでなく、クライアントエンドまでのコンテンツネゴシエーションもサポートしています。Apache では、さらに URL エリアスも無制限で提供します。

Borland では Apache Web サーバーに IIOP プラグインを追加しました。IIOP プラグインでは、インターネットインター ORB プロトコル（IIOP）を介して Apache と Borland Web コンテナが通信でき、従来にない方法で CORBA 機能を Web アプリケーションに追加できます。また、IIOP は VisiBroker ORB のプロトコルで、Web アプリケーションで Borland の ORB が提供するサービスをフルに活用できます。

JMS

AppServer は、標準 JMS 接続性をサポートし、現在は Tibco メッセージサービスがバンドルされています。JMS サービスに関するベンダー固有の情報については、221 ページの「JMS プロバイダの接続性」を参照してください。

スマートエージェント

スマートエージェントは、BES で使用している VisiBroker ORB が提供する分散ディレクトリサービスです。スマートエージェントは、クライアントプログラムとオブジェクトインプリメンテーションの両方で使用する機能で、ローカルサーバーネットワークにあるホストの少なくとも 1 つで起動する必要があります。

メモ Web サーバーと Web コンテナを HTTP やその他の Web プロトコルで通信させる場合は、Web Edition でスマートエージェントを使用する必要はありません。ただし、IIOP プラグイン（および広い意味では Web Edition の提供する ORB）を活用するには、スマートエージェントをオンにする必要があります。

ネットワーク上で実行できるように、複数のスマートエージェントを設定できます。スマートエージェントが複数のホストで起動されると、各スマートエージェントは、利用できるオブジェクトのサブセットを認識します。また、ほかのスマートエージェントと通信して、発見できないオブジェクトを検索します。さらに、スマートエージェントのプロセスの 1 つが予期しないで終了すると、そのスマートエージェントに登録されていたすべてのインプリメンテーションがこのイベントを発見し、使用可能な別のスマートエージェントに自動的に登録されます。必要とするサービス検索の付加が高い場合は、ネーミングサービス（VisiNaming）を使用することをお勧めします。VisiNaming には永続的ストレージ機能とクラスタ負荷分散機能がありますが、スマートエージェントでは osagent 単位の単純なラウンドロビンだけが提供されます。

2PC トランザクションサービス

2 フェーズコミット（ 2PC）のトランザクションサービスは、分散トランザクションを扱う CORBA アプリケーションに対する完全に回復可能なソリューションです。2PC トランザクションサービスは VisiBroker ORB 上に実装され、単一の統合アーキテクチャで基本的なサービスを提供して、分散トランザクションを単純化します。提供されるサービスには、トランザクションサービス、回復、ログ、データベースとの統合、管理機能などがあります。

パーティションとサービス

パーティションは、アプリケーションのデプロイメント先です。パーティションは、すべての J2EE 1.3 アプリケーションをサポートするときに必要な J2EE サーバー側実行時環境を提供します。パーティションは、単一のネイティブプロセスとして実装されますが、その中心インプリメンテーションは Java です。パーティションが起動すると、パーティションは内部に埋め込み Java Virtual Machine（JVM）を作成し、パーティションインプリメンテーションと J2EE アプリケーションコードを実行します。

パーティションは、AppServer の各エディションと製品に用意されていますが、Web Services、Team、および Visibroker Edition で使用できるアーカイブは限られています。ここでは、すべての Borland Enterprise Server に備わっている全機能パーティションについて説明します。各パーティションのインスタンスには次の機能があります。

- 接続サービス
- EJB コンテナ
- JDataStore サーバー
- 存続期間インターセプタマネージャ
- ネーミングサービス
- セッションストレージサービス
- トランザクションマネージャ
- Web コンテナ

接続サービス

VisiConnect とも呼ばれるコネクタサービスは Borland によるコネクタ 1.0 規格のインプリメンテーションで、さまざまな EIS を AppServer に統合するための簡潔な環境です。コネクタは、J2EE プラットフォームのアプリケーションサーバーと EIS を統合するためのソリューションを提供することにより、J2EE プラットフォームの利点である接続、トランザクション、およびセキュリティ基盤を活用できるようにして、EIS の統合という課題に対応しています。詳細については、261 ページの「VisiConnect の概要」を参照してください。

EJB コンテナ

AppServer は、統合された EJB コンテナサービスを提供します。こういったサービスを使用して、複数のパーティション上で統合された EJB コンテナや EJB コンテナを作成したり、管理することができます。このサービスを使用して、EJB をデプロイメント、実行、および監視します。ツールには、デプロイメントデスクリプタエディタ（DDEditor）と、EJB とその関連デスクリプタファイルをパッケージングおよびデプロイメントするタスクウィザードが含まれています。また、EJB コンテナは J2EE コネクタアーキテクチャを使用することもできるため、J2EE アプリケーションから企業の情報システム（Enterprise Information Systems、EIS）にアクセスできるようになります。

JDataStore サーバー

Borland の JDataStore は、完全に Java で記述されたリレーショナルデータベースサービスです。JDataStore は、必要な数だけ作成して管理できます。JDataStore の詳細については、JDatastore のオンラインマニュアル（www.borland.com/techpubs/jdatastore/）を参照してください。

存続期間インターセプタマネージャ

存続期間インターセプタを使用して、インプリメンテーションをさらにカスタマイズできます。パーティション存続期間インターセプタを使用すると、パーティションの存続期間内の特定のポイントでオペレーションを実行できます。詳細については、273 ページの「パーティションインターセプタの実装」を参照してください。

ネーミングサービス

ネーミングサービスは、VisiBroker ORB によって提供されます。このサービスを使用して、開発者、アセンブラ、デプロイヤが 1 つ以上の論理名をオブジェクトリファレンスに関連付け、その名前を VisiBroker の名前空間に保存することができます。また、このサービスにより、クライアントアプリケーションは、オブジェクトに割り当てられた論理名を使用してそのオブジェクトリファレンスを取得できます。オブジェクトインプリメンテーションは名前空間にあるオブジェクトの 1 つに名前をバインドできます。クライアントアプリケーションはこの名前空間で、`resolve()` メソッドを使って名前を解決します。メソッドはネーミングコンテキストやオブジェクトに対してオブジェクトリファレンスを返します。

セッションストレージサービス

Java セッションサービス（JSS）は特定のユーザーセッションに関係する情報を格納するサービスです。JSS を使用すると、セッション情報をデータベースに簡単に保存することができます。たとえば、ショッピングカートの場合、JSS はログイン名、ショッピングカート内の品目数などのセッション情報を取得して保存します。これにより、Borland Web コンテナの予定外のシャットダウンでセッションが中断されても、JSS を介して別の Tomcat インスタンスからにセッション情報を回復できます。JSS はローカルネットワークで実行してください。クラスタ設定内の Web コンテナインスタンスは、JSS を検索して接続し、セッション管理を続行します。詳細については、51 ページの「Java セッションサービス（JSS）の設定」の「設定」を参照してください。

トランザクションマネージャ

パーティショントランザクションマネージャは、AppServer の各パーティション内にあります。これは、CORBA トランザクションサービス仕様の Java によるインプリメンテーションです。パーティショントランザクションマネージャは、トランザクションタイムアウトと 1 フェーズコミットプロトコルをサポートします。特殊な環境では、2 フェーズコミットプロトコルでも使用できます。詳細については、153 ページの「トランザクション管理」を参照してください。

Web コンテナ

Web コンテナは、Web アプリケーションやその他のアプリケーション（サーブレット、JSP ファイルなど）の Web コンポーネントのデプロイメントをサポートするように設計されています。AppServer は、Tomcat 5.5.17 に基づく Borland Web コンテナを提供します。Tomcat は、サーブレット、JavaServer Pages、HTTP をサポートする、先進的で高い柔軟性を備えたオープンソースのツールです。また、Borland は、Web コンテナを含む IIOP プラグインも提供しており、厳密な HTTP ではなく、IIOP 上でアプリケーションコンポーネントと Web サーバーの通信を可能にします。このほかにも、Web コンテナには次のような機能が搭載されています。

- EJB リファレンシング
- データソースリファレンシング
- 環境リファレンシング
- 業界標準の Web サーバーへの統合

詳細については、27 ページの「Web コンポーネント」を参照してください。

Borland AppServer と J2EE API

AppServer は、J2EE 1.4 に完全準拠しており、次のような J2EE 1.4 API を使用できます。

- JNDI：Java ネーミングとディレクトリインターフェース
- RMI-IIOP：IIOP（Internet Inter-ORB Protocol）経由で実行される RMI（Remote Method Invokation）。
- JDBC：データベースに接続したり、データベースからデータをモデル化
- EJB 2.1：Enterprise JavaBeans 2.1 API
- Servlets 1.0：Sun Microsystems サーブレット API
- JSP：JavaServer Pages API
- JMS：Java メッセージサービス
- JTA：Java トランザクション API
- Java Mail：Java 電子メールサービス
- コネクタ 1.5：J2EE コネクタアーキテクチャ
- JAAS：Java 認証と承認サービス
- JAXP：XML 解析用の Java API

JDBC

Borland は、Sun Microsystems の Java DataBase Connection API を実装します。JDBC は、データベースドライバを記述する API と、独自のドライバを開発するための完全なサービスプロバイダサービス（SPI）を提供します。また、接続プールと分散トランザクション機能もサポートしています。詳細については、「トランザクション管理と JDBC」の「JDBC API の変更」を参照してください。

Java Mail

Java Mail は、Sun の Java Mail API のインプリメンテーションです。これはメールシステムをモデル化する抽象 API のセットで、Java 技術ベースの電子メールクライアントアプリケーションを構築するプラットフォームやプロトコルに依存しないフレームワークを提供します。

JTA

JTA（Java Transactional API）は、トランザクションの開始、停止、ロールバック、および実行するためのアプリケーションコンポーネントで必要な UserTransaction インターフェースを定義します。ほかのコンポーネントが JNDI 検索を使用するのに対して、EJB は `getUserTransaction` メソッドを使ってトランザクション関与を確立します。また、JTA は、アプリケーションサーバーのトランザクションマネージャと通信するコネクタやリソースマネージャで必要なインターフェースも指定します。

JAXP

JAXP（XML 解析用 Java API）は、DOM、SAX、および XSLT 解析のインプリメンテーションを使った XML 文書の処理を可能にします。開発者は、API のリファレンスインプリメンテーションとともに提供されるパーサを使用して、Java アプリケーションから簡単に XML を使用するようにできます。

JNDI

Java ネーミングとディレクトリインターフェースは、開発者がアプリケーションコンポーネントをアセンブリとデプロイメント時にコンポーネントのソースコードに変更を加えずにカスタマイズできるようにすることを目的として使用されます。コンテナはコンポーネントに対する実行時環境を実装し、その環境を JNDI ネーミングコンテキストとしてコンポーネントに提供します。コンポーネントのメソッドは、JNDI インターフェースを介してその環境にアクセスします。アプリケーション環境情報は JNDI ネーミングコンテキスト自体に格納され、実行時にすべてのアプリケーションコンポーネントで利用できるようにします。

RMI-IIOP

VisiBroker ORB は、RMI-over-IIOP プロトコルをサポートします。Apache と Borland Web コンテナの IIOP コネクタモジュールとともに使用することで、CORBA をベースとして分散 Web アプリケーションを構築できます。詳細については、『VisiBroker for Java 開発者ガイド』の「IIOP を介した RMI の使い方」を参照してください。

その他の技術

ほかの技術を取り込んでサービスとして提供し、AppServer で実行することもできます。

Optimizeit Profiler と Optimizeit ServerTrace

Borland の Optimizeit Profiler（別売り）では、Java アプリケーションを開発する際のメモリと CPU の利用率に関する問題を追跡できます。Optimizeit ServerTrace は、複雑な分散 J2EE ／ SOA 対応システムにおいてパフォーマンスの問題を解決する包括的で高レベルなアプリケーションパフォーマンス分析と根本原因診断の機能を提供します。AppServer は、Optimizeit Profiler と Optimizeit ServerTrace をパーティションレベルで実行します。

API の詳細については、『Sun Java Center』を参照してください。

第 3 章

パーティションの使い方

ここでは、管理コンソールを使用して、Borland AppServer（AppServer）のパーティションを操作する方法を説明します。具体的には、次の作業について説明します。

- パーティションの作成、クローンの作成、パーティションの削除
- パーティションへのモジュールのデプロイメント
- 既存のパーティションの設定
- パーティション情報の表示
- パーティションのパフォーマンスの調整
- ログ ファイルへのパーティションのスタック トレースのダンプ
- パーティションと Optimizeit Profiler または ServerTrace の実行

パーティションの作成、クローンの作成、パーティションの削除

管理コンソールを使用して、アプリケーションのパーティションの作成、クローン作成、および削除を行うことができます。パーティションをテンプレートから作成、または既存のパーティションからクローンとして作成できます。パーティションは、管理コンソールの ［Hubs］ビューのナビゲーション ペインに、親設定の子ノードとして表示されます。

新しいパーティションの作成

新しいパーティションを作成するには：

1. ナビゲーション ペインで、新しいパーティションが所属する設定を選択します。
2. 設定を右クリックし、［Add Managed Object］を選択します。
3. 管理オブジェクトの ［Template Gallery］が開きます。
4. AppServer カテゴリまたは OpenJMS Partition カテゴリで、次のパーティション テンプレートから選択します。
   - **AppServer 6.7 Partition**：AppServer 6.7 のデフォルトの `partition.config` を使って管理パーティションを生成します。
   - **Standard Partition**：管理パーティションを作成します。

- **Explicitly Pathed Partition**：既存のパーティションへのパスを作成します。既存のパーティションを現在の設定の管理下に移動する場合は、このテンプレートを使用します。
- **JBuilder Partition**：非管理パーティションを作成します。**JBuilder パーティションは、JBuilder によるローカル サーバーのデバッグに使用されます。デバッグ目的で JBuilder を AppServer とともに使用すると、自動的に作成されます。JBuilder パーティションをほかの目的で使用しないでください。**
- **Partition with Embedded OpenJMS**：OpenJMS を含むパーティションを生成します。

5 パーティション テンプレートを選択して、［Add］をクリックします。

［Add From Template］ダイアログ ボックスが表示されます。

メモ ［Add From Template］ダイアログ ボックスに表示される情報は、前の手順で選択したテンプレートによって異なります。

6 ダイアログで必要な情報（太字）を入力します。

このダイアログには、選択したテンプレートによって異なるプロパティが表示されます。テンプレート間で共通のプロパティは次のとおりです。

- **Name**：［Name］フィールドに一意のパーティション名を指定します。この名前は、ダイアログのほかの部分で文字列置換変数 `${mo.name}` の値として使用され、パーティションの表示名として使用されます。表示名を実際のパーティション名と違う名前にする場合は、［Display Name］フィールドの値を変更します。
- **Management Agent**：ドロップダウン リストから有効な管理エージェントを選択して、パーティションを作成するホストを選択します。文字列置換変数 `${hub.name}` で表される現在の管理エージェントを使用する場合、このフィールドを変更する必要はありません。
- **Display Name**：オプションで、管理コンソールに表示されるフレンドリ名を入力できます。文字列置換変数 `${mo.name}` には、［Name］フィールドの値が表示されます。
- **Smart Agent Port**：このフィールドに有効なポート番号を入力して、デフォルトの osagent（スマート エージェント）ポート番号を変更できます。
- **HTTP Connector Port**：このフィールドでデフォルトの HTTP コネクタ ポート番号を変更できます。
- **Data Directory**：明示的にパスが指定されるパーティションの場合は、既存のパーティションのパスを入力する必要があります。

7 その他のパーティション設定プロパティを表示する場合は、［Show hidden properties］チェック ボックスをオンにします。

8 終了したら、［OK］をクリックします。

既存のパーティションのクローン作成

既存のパーティションのクローンを作成するには：

1 ナビゲーション ペインから、クローンを作成するパーティションを選択します。

2 パーティションを右クリックし、［Clone］を選択します。

［Clone Managed Object］ダイアログが表示されます。

3 ［Target Configuration］ドロップダウン リストから、新たにクローンとして作成されたパーティションをターゲットにする設定を選択します。

4 ［New Name］フィールドに、クローンとして作成されるパーティションに指定する名前を入力します。

この名前は、ダイアログのほかの部分で文字列置換変数 `${mo.name}` の値として使用され、クローンとして作成されるパーティションの表示名として使用されます。表示名を実際のパーティション名と違う名前にする場合は、［New Display Name］フィールドの値を変更します。

5 必要に応じて、［Description］に入力します。この情報は省略できます。

6 ［Data Directory］（`${config.path}/mos/${mo.name}`）は、エージェントからの相対パスでパーティションの場所を識別します。

7 ［Target Agent］ドロップダウン リストから、クローンとして作成されるパーティションをホストする管理エージェントを選択します。

8 ［Target Group］ドロップダウン リストから、クローンとして作成されるパーティションが所属するグループを選択します。

9 終了したら、［OK］をクリックします。

パーティションの削除

パーティションを削除するには：

1 ナビゲーション ペインから、削除するパーティションを選択します。

2 パーティションを右クリックし、［Remove］を選択します。

パーティションへのモジュールとライブラリのデプロイメント

パーティションにモジュールをデプロイするには：

1 ナビゲーション ペインから、デプロイメント先のパーティションを選択します。

2 パーティションを右クリックし、[Deploy modules] を選択します。

モジュールとライブラリのデプロイメント ウィザードが表示されます。

3 モジュールを追加するには、[Add] をクリックします。

[Add J2EE Module] ダイアログ ボックスが表示されます。

a パーティションにデプロイするモジュールを参照し、[OK] をクリックします。

b この手順を繰り返して、すべてのアプリケーション モジュールを追加します。

間違えた場合は、そのモジュールをリストで強調表示し、[Remove] をクリックすると削除できます。

4 チェック ボックスを使用して、追加のオプションを選択します。次のオプションがあります。

- **Restart partitions on deploy (cold deploy) :** " コールド " デプロイメントを実行し、デプロイメント操作が完了したらパーティションを再起動します。実行中のパーティションへの " ホット " デプロイメントを実行し、パーティションを再起動しない場合は、この項目をオフのままにします。
- **Verify deployment descriptors :** Borland 固有のデスクリプタを含むすべてのデプロイメント デスクリプタが適切に作成されているかどうかを確認する検証ツールを実行します。これは推奨のオプションです。
- **Generate stubs :** デプロイメント時にアプリケーション スタブを生成する場合は、このチェック ボックスをオンにします。これは推奨のオプションです。

5 [Advanced Options] をクリックして、このデプロイメントの詳細なプロパティを設定します。詳細については、17 ページの「デプロイメントの詳細オプション」を参照してください。

6 [Next] をクリックして続行します。

ウィザードのステップ 2 が表示されます。

7 モジュールのデプロイメント先になるパーティションを選択します。

選択可能なパーティションが自動的にリストに表示されます。パーティションが表示されない場合は、[Refresh List] をクリックしてください。**Shift** キーまたは **Ctrl** キーを押したままクリックすると、複数のパーティションを選択できます。

8 終了したら、[Finish] をクリックします。

9 モジュールのデプロイメント中は、[Deploying Modules] ダイアログに進行状況が表示されます。

10 終了したら、[Close] をクリックします。

メモ アプリケーション モジュール（EAR）の仕様により、マニフェスト ファイルで指定したサブモジュールのクラスパス エントリはアプリケーションのクラスパスに追加する必要があります。この機能は、多くの場合使用しません。使用する場合は、VM プロパティ `enable.add.classpath.entries` を対応する設定ファイル（`partition.config` や `iastool.config` など）に指定してください。

デプロイメントの詳細オプション

デプロイメント ウィザードのステップ 1 では、スタブ生成と検証ツールの詳細オプションを設定できます。

詳細設定を行うには：

1 デプロイメント ウィザードのステップ 1 で、［Advanced Options］をクリックします。

［Advanced Deployment Options］ダイアログが表示されます。

2 ［Stub Generator］タブで次のオプションを設定できます。

- **Generate stubs：**このチェック ボックスをオンにすると、デプロイメント時にスタブの生成が有効になります。
- **Classpath：**ドロップダウン リストからクラスパスを選択したり、［Edit］をクリックし、クラスパスのリストにアーカイブを追加します。
- **Edit：**［Edit］をクリックするとクラスパス エディタが開き、そこでアーカイブとパスの追加と削除ができます。
- **Java2IIOP arguments：**このフィールドに `java2iiop` コマンドライン引数を入力します。有効なコマンドライン引数のリストについては、［More Info］をクリックするか、**『Borland VisiBorker for Java 開発者ガイド』**の「Java 対応プログラマ ツール」を参照してください。
- **More Info：**［More Info］をクリックすると、スタブ ジェネレータのコンパイル フラグの使い方が表示されます。
- **Javac arguments：**このフィールドに `javac` コマンドライン引数を入力します。

3 ［Verifier］タブでは、チェック ボックスを使って検証のレベルを選択できます。

- **Verify deployment descriptors：**Borland と標準のすべてのデスクリプタを確認する場合は、このチェック ボックスをオンにします。
- **Show all warnings：**デスクリプタに関するすべての問題のログ情報を受け取るには、このチェック ボックスをオンにします。
- **Use strict (pedantic) checks：**デスクリプタの厳密なチェックを行います。

4 終了したら、［OK］をクリックします。

追加モジュールのホスト

パーティションのフットプリント上にない " デプロイメント前 " のモジュールをホストすることもできます。さらに、" 展開されたアーカイブ " と呼ばれるパス（アーカイブ形式に変換されていないアプリケーション）をホストすることもできます。

パーティションでアーカイブをホストするには：

1 ナビゲーション ペインから、デプロイメント先のパーティションを選択します。

2 パーティションを右クリックし、［Host additional module］を選択します。

［Host Additional Module］ダイアログ ボックスが表示されます。

3 アーカイブをホストするには、［Select File］を選択します。展開されたアーカイブをホストするには、［Select Directory］を選択します。

4 ホストするアーカイブまたは展開されたアーカイブを指定するには、［Browse］をクリックします。

5 オプションで、ホストされるモジュールにわかりやすいモジュール名を指定できます。

6 終了したら、［OK］をクリックします。

パーティションの設定

ナビゲーション ペインでパーティションを選択したときに内容ペインに表示されるプロパティなどのパーティションのプロパティを設定できます。パーティション、パーティションのプロパティ、統計情報の収集の設定、JMX エージェントの設定、ログの設定、JMX クライアントの設定、時間ルール、管理の詳細オプションなどの一般情報を指定できます。

パーティションのプロパティを編集するには：

1 ナビゲーション ペインでパーティションを選択し、右クリックします。

2 ［Properties］を選択します。

［Partition Properties］ダイアログが表示されます。

3 以下で説明するタブを使用して、設定を編集します。

4 変更が完了したら、［OK］をクリックします。

一般プロパティ

［General］タブのプロパティでは、パーティションの一般情報を指定できます。

次のオプションを編集できます。

- **Display name：**管理コンソールに表示されるパーティションの名前を入力します。
- **Data directory：**パーティションのフットプリントの場所（エージェントからの相対パス）を入力します。
- **Version：**管理システムによって作成および管理されるバージョン番号。
- **Vendor：**管理オブジェクトのベンダー。標準パーティションのベンダーは、Borland Software Corporation です。
- **Description：**パーティションの説明（オプション）。

パーティション設定のプロパティ

［Partition Settings］タブのプロパティを使用して、パーティションのコマンドライン引数の指定と、JPDA デバッグの設定を行います。

次のオプションを設定できます。

- **Arguments：**パーティション実行可能ファイルのコマンドライン引数をスペース区切りリストで入力するか、［Edit］ボタンを押してコマンドライン引数をリストに保存できます。
- **Enable JPDA remote debugging：**パーティションでデバッグを有効にするかどうかを指定します。
- **JPDA debugging transport address：**パーティションに接続するために JPDA デバッガによって使用されるポート。ランダムにポートを割り当てる場合は、このフィールドを空白にします。
- **Suspend partition until debugger attaches：**オンにされた場合は、デバッガがアタッチに成功するまでパーティションを「Running（実行中）」としてマークしません。

統計情報のプロパティ

［Statistics］タブのプロパティでは、統計のタブに表示される統計情報を管理コンソールから収集できます。統計情報の収集は、デフォルトで有効になります。統計情報の収集を無効にすると、パーティションのパフォーマンスが向上します。

有効にすると、統計情報の収集について次の設定を行うことができます。

- **Enable Agent Statistics**：このチェック ボックスをオンにして、パーティションの統計情報エージェントを有効にします。

  ［Statistics level］ドロップダウン リストを使用して、ログのレベルを設定できます。また、［Snapshot period］フィールドに値を入力して、ポーリング間隔を設定できます。

- **Enable Agent Statistics Reaping**：このチェック ボックスをオンにして、保存されている統計情報を定期的に削除し、ディスク スペースを確保します。

  ［Reap older than］に値を入力して、統計情報を保存する期間を指定します。［Reap period］に値を入力して、統計情報ログを削除する頻度を設定できます。

JMX エージェントのプロパティ

［JMX Agent］タブのプロパティを使用すれば、パーティションに実装される JMX MBean サーバー、RMI-IIOP アダプタと HTTP アダプタ、および MLet サービスの一部のオプションを設定できます。

次のオプションを編集できます。

- **Enable JMX**：このチェック ボックスをオンにすると、JMX MBean サーバーが有効になります。

  MBean サーバーは、JMX のエージェント仕様レベルで定義されるインターフェイスとファクトリ オブジェクトです。このオプションは、JMX コンソールを起動するために有効にする必要があります（**『管理コンソール ユーザーズ ガイド』**の「JMX コンソールの使い方」を参照）。

- **Enable HTTP Adaptor**：このチェック ボックスをオンにすると、HTTP アダプタが有効になります。

  HTTP アダプタは、HTML 3.2 準拠のブラウザまたはアプリケーションを使ってパーティションを管理するための HTTP プロトコルのアダプタです。

  この設定タブでは、HTTP アダプタが監視するポートの番号（デフォルト値 8082）を設定し、XSLT プロセッサを有効にできます。Web ブラウザを使ってパーティションを監視する場合は、XSLT プロセッサを有効にして、HTTP アダプタの出力を未編集の XML から HTML に変換する必要があります。ホスト名（デフォルト名 `localhost`）を設定するには、`partition.xml` を編集する必要があります（詳細については、345 ページの「パーティション XML リファレンス」を参照）。

- **Enable RMI-IIOP adaptor**：このチェック ボックスをオンにすると、RMI-IIOP アダプタが有効になります。

  RMI-IIOP アダプタは、JMX クライアント フレームワークに基づくので、管理アプリケーションが RMI を使って MBean サーバーと通信する場合に役立ちます。

- **Enable mlet service**：このチェック ボックスをオンにすると、MLet サービスが有効になります。

  MLet サービスによって、MBean サーバーの JVM 内の MBean クラスとリソースを 1 つの操作で簡単にリモート ホストからロードして登録できます。

JMX エージェントのプロパティ設定の詳細については、345 ページの「パーティション XML リファレンス」を参照してください。JMX コンソール（パーティションに関連付けられている MBean の監視に使用できる JMX クライアント）の使い方については、**『管理コンソール ユーザーズ ガイド』**の「JMX コンソールの使い方」を参照してください。

JDK のプロパティ

［JDK］タブのプロパティでは、パーティションに JDK のプロパティを設定できます。

次のオプションを編集できます。

- **Select the JDK to be used by this partition：**クリックしてリストから適切な JDK を選択します。
- **Heap and Thread Stack Sizes：**［Initial heap size］フィールドに値を入力して、初期ヒープサイズを設定します。

  ［Maximum heap size］フィールドに値を入力すると、最大ヒープサイズを制御できます。VM によるスレッドの管理方法を制御するには、［Java thread stack size］フィールドを使用します。
- **Java VM Type：**適切なラジオボタンを選択して、Java VM タイプを選択します。値は、［Server］、［Client］、［Other］です（ドロップダウン リストからカスタマイズ）。

その他の JDK オプション

［Advanced Configuration］をクリックして、編集ウィンドウに `partition_server.config` ファイルを開きます。必要に応じて、ファイルにエントリを追加します。終了したら、［OK］をクリックします。

パフォーマンス調整のヒント

［Performance Tuning Hints］をクリックして、パーティションのパフォーマンスの最適化に関するヒントを取得します。終了したら、［OK］をクリックします。パフォーマンス調整の詳細については、24 ページの「パーティションのパフォーマンスの調整」を参照してください。

VisiBroker のプロパティ

［VisiBroker］タブのプロパティでは、パーティションで使用される VisiBroker ORB のプロパティの一部を調整できます。

次のオプションを編集できます。

- **Select a server connection manager：**ドロップダウン リストからサーバー接続マネージャを選択します。
- **Listener port：**サーバー接続マネージャのリスナー ポートを設定します。
- **Connection Pool：**これらのフィールドで、許容される最大接続数と最大接続アイドル時間を設定します。
- **Thread Dispatcher Pool：**これらのフィールドで、許容されるスレッドの範囲と最大スレッド アイドル時間を設定します。

その他の VisiBroker オプション

［Advanced］をクリックして、編集ウィンドウに `vbroker.properties` ファイルを開きます。必要に応じてファイルを編集します。終了したら、［OK］をクリックします。

セキュリティのプロパティ

［Security］タブのプロパティでは、パーティションのセキュリティ情報を設定できます。

セキュリティを有効にするには、ドロップダウン リストから［Security profile］を選択します。パーティションで SSL を有効にするには、`ssl_enabled` オプションを選択します。`ssl_enabled` を選択した場合は、次のオプションも設定できます。

- **SSL Listener Settings：**SSL リスナー ポートを設定し、そのポートを介して接続するクライアントの信頼を有効にします。
- **SSL Connection Pool：**これらのフィールドで、許容される最大接続数と最大接続アイドル時間を設定します。
- **SSL Thread Dispatcher Pool：**これらのフィールドで、許容されるスレッドの範囲と最大スレッド アイドル時間を設定します。

ログ設定のプロパティ

［Log Settings］タブのプロパティでは、パーティションにログのプロパティを設定できます。

次のオプションを編集できます。

- **Partition log format：**ドロップダウン リストからログ形式（XML またはテキスト）を選択します。
- **Trace Level Settings：**トレースが提供されている各サービスにトレース レベル（必要最小限（minimal）または詳細（verbose））を選択します。

時間ルールのプロパティ

［Time Rules］タブでは、パーティションの実行と停止の時刻に関するルールを設定できます。

次のプロパティを編集できます。

- **Default state：**パーティションのデフォルトの状態を実行中にするか、停止状態にするかをこのドロップダウン リストで指定します。
- **Rules：**パーティションの管理オブジェクトを利用できる時間に関するルールを設定します。

詳細設定のプロパティ

［Advanced］タブのプロパティでは、パーティションの管理方法に関する詳細情報を設定できます。

2 種類の管理情報（「管理オブジェクトの設定」と「管理アクションの設定」）を設定できます。

管理オブジェクトの設定

次の管理オブジェクトの設定を編集できます。

- **Local restart：**チェック ボックスをオンにすると、リモート ハブが要求を送信するのではなく、ローカル エージェントがパーティションを再起動します。
- **Escalate stop：**チェック ボックスをオンにすると、パーティションの停止処理がタイムアウトになった場合に、強制終了に移行します。
- **Ping policy：**［Always］に設定すると、パーティションは、実行中かどうかにかかわらず常にステータスを ping されます。［Not-when-stopped］に設定すると、パーティションは、実行中とわかる場合にだけ ping されます。
- **Ping strategy：**この操作に使用するアクション ストラテジ。
- **Ping interval：**パーティションの状態をチェックする間隔（秒）。

- **Max failure retries**：管理オブジェクトが起動、停止などの操作を再試行できる最大回数。
- **Failure retry interval**：管理オブジェクトが操作を再試行する間隔。

管理アクションの設定

ここには、パーティションの起動、停止、強制終了を設定する 3 つのタブがあります。起動、停止、強制終了の各操作に対して、次の項目を設定できます。

- **Strategy**：この操作に使用する AppServer のアクション ストラテジ。
- **First ping delay**：ping を開始するまでの時間。最初の ping 操作に遅延時間を設定しない場合は、このフィールドを空白にします。
- **Ping interval**：操作が成功したかどうかを判断するためにパーティションの状態をチェックする間隔。
- **Retry interval**：（停止と強制終了のタブのみ）再試行の間隔。
- **Max retries**：最初に操作が失敗してから管理オブジェクトが操作を再試行する最大回数。
- **Timeout**：成功するまで操作を続行する時間。

パーティション情報の表示

ナビゲーション ペインでパーティションを選択すると、管理コンソールの右側の内容ペインにさまざまな情報が表示されます。ここでは、内容ペインに表示されるタブと、タブに表示される情報について説明します。

[General] タブ

［General］タブには、パーティションに関する基本的な情報が表示されます。一般プロパティ、パーティションのプロパティ、セキュリティ設定、Web コンテナ ルート コンテキストの 4 つのカテゴリの情報が表示されます。

一般プロパティ

- **Display Name**：管理コンソールに表示されるパーティションの名前。
- **Name**：パーティションの論理名。
- **Agent name**：パーティションのホストエージェント名。
- **Data directory**：パーティションのフットプリントの場所（エージェントからの相対パス）。
- **Version**：ユーザーがオプションで指定したバージョン番号。
- **Vendor**：ユーザーが指定したパーティションのベンダー。
- **Description**：ユーザーがオプションで指定した説明。

パーティションのプロパティ

- **Path of partition on server**：パーティションのフットプリントの物理的な場所。
- **Verify all modules when loaded**：パーティションの起動時に検証ツールを実行するかどうかを指定するフラグ。

セキュリティのプロパティ

- **Security profile**：パーティションで基本的なセキュリティ（認証 / 承認）を有効にするかどうかを指定するフラグ。
- **Secure transport enabled**：RPC が SSL を実装するかどうかを指定するフラグ。

Web コンテナ ルート コンテキスト

［General］タブのこのプロパティでは、Web コンテナのルート コンテキストが表示されます。

［Properties］タブ

［Properties］タブには、JPDA リモート デバッグとサーバー接続マネージャ設定に関する情報が表示されます。

仮想マシン

- **Enable JPDA remote debugging：**パーティションでデバッグを有効にするかどうかを指定します。
- **JPDA debugging transport address：**パーティションに接続するために JPDA デバッガによって使用されるポート。
- **Suspend partition until debugger attaches：**true の場合は、デバッガがアタッチに成功するまでパーティションを "Running（実行中）" としてマークしません。

サーバー接続マネージャ設定

- **Server connection manager name：**パーティションにサービスを提供するサーバー接続マネージャ。
- **Listener port：**サーバー接続マネージャが着信接続を監視するポート。
- **Connection Pool：**パーティションへの接続の許容される最少数と最大数の範囲を表示します。
- **Dispatcher Pool：**パーティション内のスレッドの許容される最少数と最大数の範囲を表示します。ほかに、最大スレッドアイドル時間も表示されます。

［XML］タブ

［XML］タブには、パーティションの管理プロパティを定義する XML データ ブロックが表示されます。このデータ ブロックは、`configuration.xml` ファイルから抜粋されます。

［Class Loading］タブ

［Class Loading］タブには、パーティションのクラスローディング ポリシーとクラスローダー階層が表示されます。

［Logs］タブ

［Logs］タブには、log4j、stderr、stdout のログが表示されます。タブの左上隅にあるドロップダウン リストを使用して、各ログの内容を表示します。log4j ログには、フィルタリングの機能があります。

［Status］タブ

パーティションの統計情報エージェントを有効にしている場合は、［Status］タブにパーティションに関する情報がリアルタイムで表示されます。統計収集を有効にするには、19 ページの「統計情報のプロパティ」を参照してください。

[JDBC Pool States] タブ

パーティションの統計情報エージェントを有効にしている場合は、[JDBC Pool States] タブに JDBC プールに関する情報がリアルタイムで表示されます。統計収集を有効にするには、19 ページの「統計情報のプロパティ」を参照してください。

パーティションのパフォーマンスの調整

パーティションには、パフォーマンスのチューニング ウィザードがあり、パーティションの操作に関する環境設定を行うことができます。パフォーマンス チューニングの詳細プロパティも設定できます。

パーティションを調整するには：

1 ナビゲーション ペインから、調整するパーティションを選択します。

2 パーティションを右クリックし、[Performance Tuning] を選択します。[Performance Tuning Wizard | Step 1] が表示されます。

このステップで、統計情報の調整の使用モデルを選択できます。ここで選択した内容は、ウィザードの次のステップにある統計情報の収集の設定に反映されます。次のモデルから選択します。

- **Developer：**最大レベルの統計情報の収集を有効にします。
- **System test：**中間レベルの統計情報の収集を有効にします。
- **Production：**最小レベルの統計情報の収集を有効にします。
- **Best performance：**このオプションを選択すると、統計情報の収集が無効になり、パーティションのパフォーマンスが向上します。
- **Custom：**目的の設定が上記の定義済みの使用モデルに該当しない場合は、このオプションを選択します。上記の使用モデルをカスタマイズすることもできますが、その場合、使用モデルは*カスタム*と表示されます。

3 終了したら、[Next] をクリックします。[Performance Tuning Wizard | Step 2] が表示されます。

このステップでは、パーティションが操作に関する統計情報を収集する方法を選択します。統計情報は、ディスクに書き込まれ、内容ペインのパーティションの [Statistics] タブに表示されます。パーティションの統計の詳細については、345 ページの「パーティション XML リファレンス」を参照してください。パフォーマンスを最大にするには、統計情報の収集を無効にします。このステップでは、次のオプションを指定できます。

- **Enable Agent Statistics：**このチェック ボックスをオンにして、パーティションの統計情報エージェントを有効にします。[Statistics level] ドロップダウン リストを使用して、ログのレベルを設定できます。また、[Snapshot period] フィールドに値を入力して、ポーリング間隔を設定できます。
- **Enable Agent Statistics Reaping：**このチェック ボックスをオンにして、保存されている統計情報を定期的に削除し、ディスク スペースを確保します。[Reap older than] に値を入力して、統計情報を保存する期間を指定します。[Reap period] に値を入力して、統計情報ログを削除する頻度を設定できます。

4 続行するには、[Next] をクリックします。ステップ 3 が表示されます。

不要または使用されていないデプロイメント モジュールをパーティションで有効にすると、パーティションの起動時間が長くなり、メモリのフットプリントが増加し、ガベージ コレクションにかかる時間が長くなります。無効にするモジュールをリストから選択します。**Shift** キーまたは **Ctrl** キーを押したままクリックすると、複数のモジュールを選択できます。

5 続行するには、[Next] をクリックします。ステップ 4 が表示されます。

アプリケーションで必要がないパーティション サービスを無効にできます。パーティション サービスを無効にするには、チェック ボックスをオフにします。有効にするには、チェック ボックスをオンにします。

6 続行するには、[Next] をクリックします。ステップ 5 が表示されます。

VM パラメータをアプリケーションに合わせて調整すれば、パフォーマンスを向上できます。このステップで、次の設定を行うことができます。

- **Select the JDK to be used by this partition**：リストから JDK をクリックして選択します。
- **Heap and Thread Stack Sizes**：[Initial heap size] フィールドに値を入力して、初期ヒープサイズを設定します。[Maximum heap size] フィールドに値を入力すると、最大ヒープサイズを制御できます。VM によるスレッドの管理方法を制御するには、[Java thread stack size] フィールドを使用します。
- **Java VM Type**：適切なオプションを選択します。有効な値は、[Server]、[Client]、[Other] です（ドロップダウン リストから選択）。

さらに、一部の詳細設定（25 ページの「パフォーマンスのチューニングの詳細オプション」を参照）を行い、パフォーマンス調整のヒントを参照することもできます。

7 終了したら、[Next] をクリックします。ステップ 6 が表示されます。

パフォーマンスに影響を与える VisiBroker パラメータの一部を設定できます。このステップで、次の設定を行うことができます。

- **Connection Pool**：適切なダイアログ ボックスで、許容される最大接続数と最大接続アイドル時間を設定します。
- **Dispatcher Pool**：[Minimum number of threads] フィールドと [Maximum number of threads] フィールドで、許容されるスレッド数の範囲を設定できます。[Maximum thread idle time] フィールドを設定します。
- **Security**：パーティションで基本的なセキュリティ（認証 / 承認）を有効にするかどうかを指定するフラグ。

8 続行するには、[Next] をクリックします。最後のステップが表示されます。

このステップでは、J2EE 固有のプロパティを調整できます。次の設定を設定できます。

- **EJB Container Tuning**：[Session passivation timeout] フィールドを設定します。Bean 内呼び出しで PBV を使用するには、チェック ボックスをオンにします。
- **Message-driven bean thread pool**：[Minimum number of threads] フィールドと [Maximum number of threads] フィールドに値を設定して、スレッドプールの範囲を設定します。[Maximum thread idle time] フィールドを設定します。
- **Web Container Tuning**：ドロップダウン リストから HTTP サービス コネクタを選択し、プロセッサと接続タイムアウトについて設定します。

9 終了したら、[Finish] をクリックします。

パフォーマンスのチューニングの詳細オプション

パフォーマンスのチューニングの詳細オプションにアクセスするには：

1 パフォーマンス チューニング ウィザードのステップ 4 で、[Advanced Configuration] をクリックします。

2 編集ウィンドウに `partition_server.config` ファイルが表示されます。

3 必要に応じてファイルを編集します。終了したら、[OK] をクリックします。

ログ ファイルへのパーティションのスタック トレースのダンプ

トラブルシューティングのために、ログ ファイルにパーティションのスタック トレースをダンプできます。

ログ ファイルにパーティションのスタック トレースをダンプするには：

1 ナビゲーション ペインでパーティションを選択します。

2 パーティションを右クリックし、[Dump stack to log] を選択します。

3 スタック トレースは、次の場所にある `stdout.log` にダンプされます。

```
<install_dir>/var/domains/<domain-name>/configurations/<configuration-name>/
    mos/<partition-name>/adm/logs/<partition_name>.stdout.log
```

パーティションと Optimizeit Profiler または ServerTrace の実行

Optimizeit ServerTrace と Optimizeit Profiler が統合された管理コンソールの設定と使い方については、**『管理コンソール ユーザーズ ガイド』**の「Optimizeit Profiler と ServerTrace の使い方」を参照してください。

第 4 章

Web コンポーネント

ここでは、Borland AppServer（AppServer）に含まれ、VisiBroker Edition の一部（Web サービスパック（VisiExchange）コンポーネント）としてオプションでインストールできる Web コンポーネントに関する情報を提供します。詳細については、Borland AppServer の『*インストールガイド*』の「Borlans AppServer の Windows へのインストール」または「Borland AppServer または Linux へのインストール」のセクションを参照してください。

Apache Web サーバーのインプリメンテーション

オープンソース Apache Web Server バージョン 2.3（httpd サーバー）の AppServer のインプリメンテーションは HTTP 1.1 準拠で、 Apache モジュールを介して高度にカスタマイズ可能です。

Apache 設定

Apache Web サーバーはあらかじめ設定済みで出荷されるので、起動してすぐに使用できます。Apache の起動時に、多くのモジュールが動的にロードされます。Apache は、後で、1 つ以上の Web コンテナで IIOP コネクタ、クラスタ、フェイルオーバー、負荷分散の設定をカスタマイズできます。管理コンソールを使って設定ファイルを変更できます。プレーンテキストの設定ファイル `httpd.conf` で、指示文を使って変更することもできます。

デフォルトでは、Apache `httpd.conf` ファイルは次のディレクトリにあります。

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/
mos/<apache_managedobject_name>/conf
```

また、`httpd.conf` ファイルの場所については、次の場所にある `configuration.xml` ファイルを参照してください。

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>
```

そして、Apache 管理オブジェクトの `apache-process` `httpd-conf` サブ要素の属性を検索します。

```
httpd-conf=
```

IIOP コネクタ／リダイレクタの `httpd.conf` ファイルの設定については、「Web サーバーと Web コンテナの接続」の 37 ページの「Apache における IIOP 設定の変更」を参照してください。

Apache 設定構文

httpd.conf ファイルを編集する際は、次に示す構文のガイドラインにしたがってください。

- httpd.conf ファイルは、1 行に 1 つの指示文からなります。
- 指示文が次の行に続くことを示すには、行の最後の文字としてバックスラッシュ（日本語環境では ¥ 記号）を使用します。
- 行の最後とバックスラッシュ「\」の間に、ほかの文字や空白は入れないでください。
- 指示文では大文字と小文字は区別しませんが、指示文の引数では大文字と小文字が区別される場合があります。
- ハッシュ文字「#」で始まる行はコメントとみなされます。
- 設定指示文の後の行には、コメントは挿入できません。
- 指示文の前にある空行と空白は無視されるため、わかりやすくするために指示文をインデントすることができます。

特権ポートでの Apache Web サーバーの実行

UNIX ホストの特権ポートにアクセスするプロセスは、適切なアクセス許可を持つ必要があります。たとえば、ユーザー root のアクセス許可付きでプロセスを開始する必要があります。通常、root のアクセス許可付きで開始する必要があるプロセスは、設定内の一部のプロセスだけです。setuser スクリプトは、Apache Web サーバーがルートまたはルート特権で起動できるように BES を設定します。setuser ツールの使い方とマルチユーザーモードについては、『インストール ガイドの』「setuser ツールによる所有権の管理」を参照してください。

設定を開始する前に、Apache をインストールするシステムについて、以下の情報を収集します。

1 AppServer インストールディレクトリ

2 ルート UID を放棄した後にエージェントが使用するアカウント、つまりインストールオーナーのユーザー名とグループ名。

3 システムルートアカウントのユーザー名とグループ名（通常は root/sys）

次の手順で、Apache Web サーバーがポート 80 で実行するように設定する方法を説明します。

1 管理ハブと Apache Web サーバーを含む設定ファイルが実行されていないことを確認します。

2 AppServer インストールでマルチユーザーモードを有効にします。

   a プロパティを編集します。

   ```
   agent.mum.enabled.root.mo=true
   ```

   次の場所にあります。

   ```
   <install_dir>/var/domains/base/adm/properties/agent.config
   ```

   b root ユーザーになります。

   c setuser スクリプトを実行します。

   ```
   setuser -u <user> -g <group> +m
   ```

   ここで、<user> と <group> はインストールオーナーのアカウント属性です（上記の B を参照）。

3 管理ハブを起動します。

4 管理コンソールにある Apache Web サーバーを編集します。

   a Apache Web サーバーの MO を右クリックし、［Properties］を選択します。

   b ［Properties］ダイアログボックスで［Apache Process Settings］タブを選択します。

c ［More settings］をクリックし、［Advanced Process Settings］ダイアログボックスを開きます。

d ［Platform Specific Settings］タブを選択します。

e ［Unix Settings］グループで、［Start as user］と［Start as group］フィールドにシステムルートアカウントのユーザー名とグループ名を入力します（上記の C を参照）。

f ［OK］をクリックして［Advanced Process Settings］ダイアログボックスを閉じます。

g ［Properties］ダイアログボックスで、［Files］タブを選択し、httpd.conf ファイルを選択します。

h ユーザーとグループの指示文を、AppServer インストールを所有するアカウントのユーザー名とグループ名の値に変更します（上記の B を参照）。

i Listen 指示文を「80」に変更します。

j ［OK］をクリックして［Apache Properties］ダイアログボックスを閉じます。

5 設定ファイルを起動します。

.htaccess ファイルの使い方

Apache Web サーバーでは、Web ツリーの中に置かれた `.htaccess` ファイルで、設定を分散的に管理します。これらのファイルは、`AccessFileName` 指示文で指定します。

`.htaccess` ファイルに書かれた指示文は、このファイルが置かれたディレクトリとそのすべてのサブディレクトリに適用されます。`.htaccess` ファイルの構文は、メインの設定ファイルと同じです。`.htaccess` ファイルは、すべての要求で読み取られるため、このファイルに対する変更はすぐに反映されます。`.htaccess` ファイルに組み込むことのできる指示文については、指示文のコンテキストを参照してください。`.htaccess` ファイルに組み込むことのできる指示文は、メインの設定ファイルに `AllowOverride` 指示文を設定して制御します。

Apache ディレクトリ構造

Apache Web サーバーをインストールすると、デフォルトでは、次の場所に Apache 固有のディレクトリ構造が作成されます。

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/
mos/<apache_managedobject_name>/
```

Apache 固有のディレクトリ名	説明
`cgi-bin`	すべての CGI スクリプトを格納します。
`conf`	すべての設定ファイルを格納します。
`error`	すべてのエラー html ドキュメントを格納します。
`htdocs`	すべての HTML ドキュメントと Web ページを格納します。
`icons`	`.gif` 形式のアイコン画像を格納します。
`logs`	すべてのログファイルを格納します。
`proxy`	Web アプリケーションのプロキシを格納します。

Borland Web コンテナのインプリメンテーション

Borland Web コンテナは、Web アプリケーションの開発とデプロイメントをサポートします。Tomcat 5.5.17 をベースにした Borland Web コンテナは AppServer Edition に含まれ、オプションで VisiBroker Edition の一部としてインストールできます（VisiExchange に含まれる）。詳細については、Borland AppServer の『*インストールガイド*』の「AppServer の Windows へのインストール」または「AppServer の Solaris または Linux へのインストール」のセクションを参照してください。

Borland Web コンテナは洗練された柔軟性のあるツールであり、Servlets 2.4 仕様や JSP 2.0 仕様をサポートします。

「パーティションサービス」として、Borland Web コンテナ設定ファイルはすべて、次の下のパーティションの各データディレクトリに置かれています。

```
adm/tomcat/conf/
```

デフォルトでは、パーティションのデータディレクトリは次の場所にあります。

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/
mos/<partition_name>/
```

たとえば、「standard」という名前のパーティションの場合、デフォルトでは、Borland Web コンテナ設定ファイルは次の場所にあります。

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/
mos/standard/adm/tomcat/conf/
```

また、パーティションのデータディレクトリの場所については、次の場所にある `configuration.xml` ファイルを参照してください。

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/
```

さらに、パーティション管理オブジェクトの `partition-process` サブ要素の次の `directory` 属性を探してください。

```
<partition-process directory=
```

サーブレットと JavaServer Pages

「サーブレット」とは、Web サーバーの機能を拡張する Java プログラムです。これらは、動的なコンテンツを生成したり、要求／応答の形式で Web クライアントとの通信を行います。

「JavaServer Pages（JSP）」は、サーブレットモデルよりさらに抽象的なものです。JSP は、テンプレートデータ、カスタム要素、スクリプト記述言語、サーバー側 Java オブジェクトを使用して、クライアントに動的コンテンツを返し拡張 Web 機能です。通常、テンプレートデータは HTML 要素または XML 要素であり、そのクライアントは多くの場合 Web ブラウザです。

サーブレットと JSP は、通常 Web サーバー内で実行されるサーバーコンポーネントです。サーブレットは、HTML ページなどとは別に、Web サーバーの拡張機能として開発しますが、JSP では Java コードが直接 HTML に埋め込まれます。実行時に、JSP Java コードは自動的にサーブレットに変換されます。

サーブレットは Web 要求を処理し、これらをバックエンドのエンタープライズアプリケーションシステムに渡し、処理結果を HTML や XML のクライアントインターフェースとして動的に表現します。また、サーブレットはクライアントセッション情報を管理するため、ユーザーは同じ情報を何度も入力する必要がありません。

一般的な Web アプリケーション開発プロセス

Web アプリケーションの一般的な開発プロセスは、次のような手順にしたがいます。

1. Web デザイナが JSP コンポーネントを記述し、ソフトウェア開発者が表示ロジックを処理するサーブレットを作成します。
2. さらに、サーバー側コンポーネント（EJB アプリケーション層、CORBA オブジェクト、JDBC オブジェクト）に送るクライアント要求を処理するために、その他のソフトウェアエンジニアが、サーブレットの Java ソースコードを記述したり、`.jsp` ファイルや `.html` ファイルを作成します。
3. Java クラスファイル、`.jsp` ファイル、および `.html` ファイルは、デプロイメントデスクリプタとバンドルされて WAR（Web ARchive）ファイルになります。
4. WAR ファイル（または Web モジュール）は、Web アプリケーションとして Borland Web コンテナにデプロイメントされます。

AppServer デプロイメントデスクリプタエディタ（DDE）を使って Web アーカイブ（WAR）ファイルを作成する方法については、『*ユーザーズガイド*』の「デプロイメントデスクリプタエディタの使い方」の「WAR 情報の追加」を参照してください。

Web アプリケーションアーカイブ（WAR）ファイル

Borland Web コンテナで Web アプリケーションをデプロイメントするには、Web アプリケーションを Web アーカイブ（WAR）ファイルにパッケージする必要があります。それには、一般的な Java アーカイブツール `jar` コマンドを使用します。

WAR ファイルには、`WEB-INF` ディレクトリを組み込みます。このディレクトリには、Web アプリケーションに関連したファイルが保存されます。Web アプリケーションのドキュメントルートディレクトリとは異なり、`WEB-INF` ディレクトリのファイルでは、クライアントとの直接的な対話機能がありません。`WEB-INF` ディレクトリには、次のものが入っています。

ディレクトリ／ファイル名	内容
`/WEB-INF/web.xml`	デプロイメントデスクリプタ
`/WEB-INF/web-borland.xml`	Borland 固有の DTD を持つデプロイメントデスクリプタ。
`/WEB-INF/classes/*`	サーブレットとユーティリティクラス。 アプリケーションクラスローダーは、 このディレクトリでクラスをロードします。
`/WEB-INF/lib/*.jar`	Web アプリケーションに役立つサーブレット、 Bean、 その他のユーティリティクラスを含む Java アーカイブ（JAR）ファイル。 すべての JAR ファイルは、クラスをロードするために Web アプリケーションのクラスローダーに使用します。

Borland 固有の DTD

`web.xml` ファイルには、Web アプリケーションのための標準的なデプロイメントデスクリプタ機能が含まれます。ただし、`web-borland.xml` には、Borland 固有の拡張機能を格納します。以下の表では、Borland 固有の要素とその使い方を説明します。一部の要素は標準構造体を強化したものであり、一部は新規の構造体です。

Web コンテナの環境変数の追加

パーティションの Web コンテナ環境変数は、パーティションサービスの環境変数を設定するのと同じ方法で追加できます。<env-vars> 要素を使用し、partition-process サブ要素内で xml コードを挿入します。

メモ　Web コンテナ環境変数を追加する際は、値のペアをスペースで区切って入力してください。

すべての configuration.xml は、デフォルトでは次のディレクトリに置かれています。

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/
```

パーティションの管理オブジェクトの Web コンテナ環境変数を追加するには、env-vars 要素と env-var サブ要素を次の構文で使用します。

```
<managed-object name="standard"> ...>
    <partition-process ...>
        <env-vars ...>
          <env-var name="name" value="value"/>
        </env-vars>
        ...
</managed-object>
```

ここで、<name> は環境変数名、<value> は指定された環境変数に設定する値です。

次に例を示します。

```
<managed-object name="standard"> ...>
    <partition-process ...>
        <env-vars ...>
          <env-var name="ABC" value="val_abc"/>
        </env-vars>
        ...
</managed-object>
```

Microsoft Internet Information Services（IIS）Web サーバー

Microsoft Internet Information Services（IIS）Web サーバーは、AppServer 製品には含まれていません。ただし、AppServer には、Borland の Tomcat ベースの Web コンテナから IIS Web サーバーへの接続と、IIS Web サーバーから CORBA サーバーへの接続を提供する IIOP リダイレクタが含まれています。IIOP リダイレクタは、次の IIS バージョンでサポートされています。

- Microsoft Windows 2000/IIS バージョン 5.0
- Microsoft Windows XP/IIS バージョン 5.1
- Microsoft Windows 2003/IIS バージョン 6.0

詳細については、46 ページの「IIS Web サーバーと Borland Web コンテナの接続」を参照してください。

IIS/IIOP リダイレクタのディレクトリ構造

AppServer 製品をインストールすると、デフォルトでは、次のような IIS/IIOP リダイレクタ固有のディレクトリ構造が作成されます。

```
<install_dir>/etc/iisredir2/
```

IIS/IIOP リダイレクタ固有のディレクトリ名	説明
conf	すべての設定ファイルを格納します。
logs	すべてのログファイルを格納します。

スマートエージェントのインプリメンテーション

スマートエージェントは、クライアントプログラムとオブジェクトインプリメンテーションの特定やマッピングに役立つサービスです。スマートエージェントは、デフォルトのプロパティで自動的に起動します。スマートエージェントの設定については、『VisiBroker for Java 開発者ガイド』の*「スマートエージェントの使い方」*を参照してください。

スマートエージェントは、動的な分散ディレクトリサービスであり、クライアントプログラムとオブジェクトインプリメンテーションの両方にサービスを提供します。スマートエージェントは、バインドするクライアントプログラムで使用するオブジェクト名やサービス名をオブジェクトインプリメンテーションに関連付けることで、クライアントプログラムを適切なオブジェクトインプリメンテーションにマッピングします。オブジェクトインプリメンテーションとは、Borland Web コンテナなど、サーバーが提供するオブジェクトリファレンスのことです。

ローカルネットワーク内の少なくとも 1 つのホストでスマートエージェントを起動する必要があります。クライアントプログラムが（`bind` メソッドで）オブジェクトを呼び出すと、スマートエージェントが自動的に問い合わせを受けます。スマートエージェントは、クライアントおよび指定されたオブジェクトインプリメンテーションの間に接続を確立するために、オブジェクトインプリメンテーションを検索します。スマートエージェントとの通信は、クライアントプログラムに対して完全に透過的です。

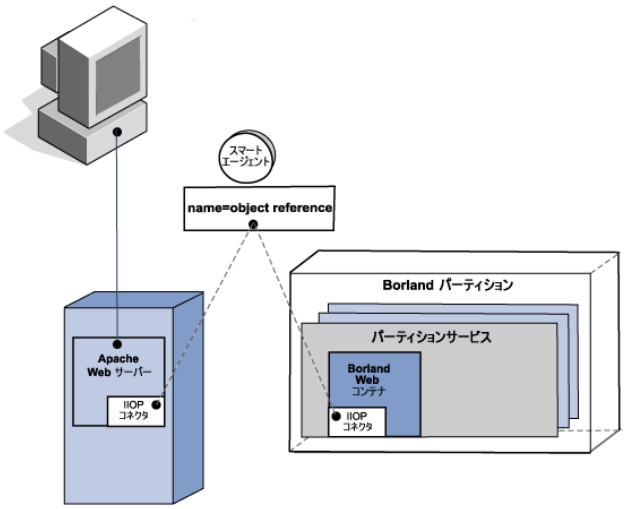
次に、AppServer Web コンポーネントによるスマートエージェントの使用例を示します。

- Apache Web サーバーの Borland Web コンテナへの接続
- Borland Web コンテナの Java セッションサービス（JSS）への接続

Apache Web サーバーの Borland Web コンテナへの接続

スマートエージェントは、分散ディレクトリサービスとして、オブジェクトリファレンスのアクティブな ID を登録し、クライアントプログラムで使用します。次の図では、スマートエージェントでバインドしたクライアントプログラムとオブジェクトとの間の対話を示します。この例では、Apache Web サーバーはクライアントとして機能し、Borland Web コンテナはサーバーとして機能し、オブジェクトリファレンスを提供します。

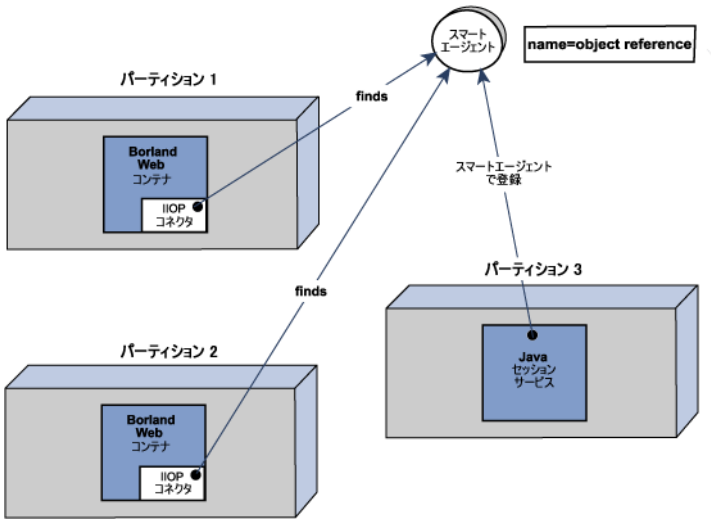
図 4.1　オブジェクトリファレンスにバインドしたクライアントプログラム



Borland Web コンテナの Java セッションサービスへの接続

この構成では、起動時に複数の Web コンテナを Java セッションサービスに接続する必要があります。クライアントとサーバーは、スマートエージェントで接続します。次の図では、Borland Web コンテナの複数のインスタンスを示します。それぞれの Web コンテナはクライアントとして機能します。起動時に、スマートエージェントはディレクトリサービスとして問い合わせを受け、JSS オブジェクトリファレンスを検索および接続します。Java セッションサービス（JSS）については、51 ページの「Java セッションサービス（JSS）の設定」を参照してください。

図 4.2　複数の Web コンテナを 1 つの JSS に接続する



第 5 章

Web サーバーと Web コンテナの接続

ここでは、Borland AppServer （AppServer）で提供され、VisiBroker Edition の一部（Web サービスパック（VisiExchange）コンポーネント）としてオプションでインストールできる Web サーバーと Web コンテナ IIOP の接続性について説明します。詳細については、Borland AppServer 『*インストールガイド*』の「Windows へのインストール」または「Solaris または Linux へのインストール」のセクションを参照してください。

Apache と CORBA の接続については、61 ページの「Apache Web サーバーから CORBA サーバーへの接続」を参照してください。

Apache Web サーバーと Borland Web コンテナの接続

ここでは以下の Web コンポーネントの機能について説明します。

- オープンソース Apache Web Server バージョン 2.2.3 のインプリメンテーション
- Tomcat ベースの Borland Web コンテナ
- Apache Web サーバーから Tomcat ベースの Borland Web コンテナへの接続を提供する IIOP コネクタ

これらの Web コンポーネントは、BorlandAppServer AppServer Edition に含まれ、オプションで VisiBroker Edition の一部としてインストールできます。詳細については、『*インストールガイド*』の「AppServer の Windows へのインストール」または「AppServer の Solaris または Linux へのインストール」のセクションを参照してください。

Borland Web コンテナの IIOP 設定の変更

`server.xml` は、Borland Web コンテナのメインの設定ファイルで、パーティションのデータディレクトリに格納されています。

```
adm/tomcat/conf/
```

詳細については、「Web コンポーネント」の 30 ページの「Borland Web コンテナのインプリメンテーション」を参照してください。

`server.xml` ファイルには、IIOP コネクタ設定に関する次のコード行があります。

```
<Connector className="com.borland.catalina.connector.iiop.IiopConnector"
 name="tc_inst1 debug="0" minProcessors="5"
 maxProcessors="75" enableChunking="false" port="0"
 canBufferHttp10Data="true" downloadBufferSize="4096" shortSessionId="false" />
```

このコード行と次の属性を使って Borland Web コンテナの IIOP コネクタを設定します。

属性	デフォルト値	説明
name	tc_inst1	Apache および IIS Web サーバーがこのコネクタに到達するために使用する名前。
debug	0	デバッグ情報のレベルを設定する整数。デフォルトの 0 に設定すると、デバッグはオフになります。デバッグをオンにするには、1 に設定します。詳細なデバッグメッセージを得るには、99 に設定します。
minProcessors	5	要求を処理するためにこのコネクタにあらかじめ作成する最小スレッド数。
maxProcessors	75	要求を処理するためにこのコネクタに作成できる最大スレッド数。
enableChunking	false	コネクタでチャンク化動作を有効にします。チャンク化を有効にするには、この属性を true に設定します。 **重要：**チャンク化を有効にする場合は、サーブレットの応答ヘッダー Transfer-Encoding の値も chunked に設定する必要があります。詳細については、「大量データのダウンロード」を参照してください。
downloadBufferSize	4096	enableChunking を true に設定した場合に使用される、「チャンク化」バッファサイズを定義します。この指示文は、0 より大きい数値を受け付けます。基本的に、この指示文に指定するバイト数を大きくすると、データを Apache または IIS に送信するために必要な CORBA RPC の数が減少します。ただし、この指示文の設定値を大きくすると、トランザクションを処理する際に消費されるメモリが増えます。このパラメータを調整してパフォーマンス特性を微調整できます。これにより、管理者は、メモリリソースの使用量より RPC コストを重視してシステム上のアップロードを最適化できます。 **メモ：**無効な値（数値以外の値や負数）が存在すると、デフォルト値の 4096 が使用されます。詳細については、「大量データのダウンロード」を参照してください。
port	0	IIOP コネクタポート。デフォルトの 0（0）に設定すると、ランダムポートが選択されます。**メモ：**Apache または IIS からこのコネクタを探すのに corbaloc メカニズムを使用する必要がある場合は、port を 0 以外の値に設定する必要があります。

属性	デフォルト値	説明
canBufferHttp10Data	true	HTTP プロトコルのバージョンが 1.1 より低く、コンテンツ長がサーブレットで設定されていない場合、Web コンテナは、次の 2 つのうちのいずれかを実行できます。 データをバッファに入れることができる場合は、コンテンツ長を計算した後、 応答を送信するか、 エラーメッセージを生成できます。 データをバッファに入れてメモリを消費するのを避けるには、 この属性を `false` に設定します。詳細については、「HTTP 1.0 プロトコルだけをサポートするブラウザ」を参照してください。
shortSessionId	false	IIOP コネクタセッション ID のサイズを縮小する Borland "collapsed locator" 機能を有効にするには、`true` に設定します。デフォルトでは、この属性は無効、 つまり `false` に設定されています。 IIOP コンテナは、 クライアントを指定された Borland Web コンテナ（サービスアフィニティ）に関連付けるために、 文字列化されたオブジェクトリファレンス（IOR）を使用します。 ネットワークルーターやブラウザによっては、Session Id Cookie に格納された大きなペイロードに対応できないものもあります。 この問題を解決するために、 Borland はサービスアフィニティをはるかに短いセッション ID 文字列を使って実装できる collapsed locator 文字列を開発しました。 `shortSessionId` を `true` に設定するとこの機能が有効になり、`false` に設定するか、またはパラメータを省略すると、デフォルトの従来の IOR ベースのソリューションになります。 **メモ：**"collapsed locator" は、基本的にコーディングされた CORBALOC 文字列です。CORBALOC 文字列をオブジェクトリファレンスに解決する処理は、IOR で同じ処理をするより負荷がかかります。 この余分な負荷を軽減するために、 Apache Web Server では、コーディングされた CORBALOC 文字列から解決済みオブジェクトリファレンスへのルックアサイドリストを保持します。 これにより、 Apache Web Server あたりのメモリ使用量はわずかに増加します。 **重要：** `shortSessionID` 機能が正常に動作するには、IIOP コネクタに `port` 値を指定する**必要があり**、デフォルトの `port` 値 （ゼロ）を使用することはできません。

Apache における IIOP 設定の変更

`httpd.conf` ファイルは、Apache Web サーバーのグローバル設定ファイルです。
`httpd.conf` ファイルには、IIOP コネクタに関する次の行があります。

Windows

```
    LoadModule iiop2_module <install_dir>/lib/<apache_managedobject_name>/
       mod_iiop2.dll
    IIopLogFile <install_dir>/var/domains/<domain_name>/
  configurations/<configuration_name>/mos/<apache_managedobject_name>/
  logs/mod_iiop.log
    IIopLogLevel error
    IIopClusterConfig <install_dir>/var/domains/<domain_name>/
  configurations/<configuration_name>/mos/<apache_managedobject_name>/
  conf/WebClusters.properties
    IIopMapFile <install_dir>/var/domains/<domain_name>/
  configurations/<configuration_name>/mos/<apache_managedobject_name>/
  conf/UriMapFile.properties
```

このコード行を使って Apache Web サーバーの IIOP コネクタを設定します。

指示文	デフォルト	説明
LoadModule	<install_dir>/lib/<apache_managedobject_name>/ mod_iiop2.dll	Apache 2.2 が IIOP コネクタをロードできるようにします。この指示文は、Apache mod_iiop2 モジュールを指定された場所からロードするように Apache Web サーバーに指示します。モジュールをロードすると、次の 4 つの指示文によって IIOP コネクタを有効にして、通信相手の Web コンテナまたは CORBA サーバーを検索し、その他の機能を実行します。
IIopLogFile	<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/mos/<apache_managedobject_name>/logs/mod_iiop.log	IIOP コネクタがログ出力を書き込む場所を指定します。
IIopLogLevel	error	書き込むログ情報のレベルを指定します。指示文の値は、debug、warn、info、error のいずれかになります。
IIopClusterConfig	<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/mos/<apache_managedobject_name>/conf/WebClusters.properties	「クラスタ」インスタンスファイルの場所を設定します。CORBA サーバーの場合、IIOP コネクタが認識するための「クラスタ」名を登録したファイル名を指定します。[1]
IIopMapFile	<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/mos/<apache_managedobject_name>/conf/UriMapFile.properties	URI 対インスタンスマッピングファイルの場所を設定します。CORBA サーバーの場合、IIOP コネクタが認識できる特定の「クラスタ」までの HTTP URI をマッピングします。

[1]「クラスタ」は、システムが 1 つの名前や URI で認識する CORBA サーバーインスタンスを表します。IIOP コネクタでは、複数のインスタンス間で負荷分散できるので、用語「クラスタ」を使用します。

Apache 2.2.3 の IIOP コネクタ向けのこの 5 行の代表的な設定例を次に示します。

Windows のサンプル

```
  LoadModule iiop2_module C:/Borland/BDP/lib/myapache/mod_iiop2.dll
  IIopLogFile C:/Borland/BDP/var/domains/base/configurations/j2ee/mos/
myapache/logs/mod_iiop.log
  IIopLogLevel error
  IIopClusterConfig C:/Borland/BDP/var/domains/base/configurations/j2ee/
mos/myapache/conf/WebClusters.properties
  IIopMapFile C:/Borland/BDP/var/domains/base/configurations/j2ee/mos/
myapache/conf/UriMapFile.properties
```

Solaris のサンプル

```
  LoadModule iiop2_module /opt/Borland/BDP/lib/myapache/mod_iiop2.so
  IIopLogFile /opt/Borland/BDP/var/domains/base/configurations/j2ee/mos/
myapache/logs/mod_iiop.log
  IIopLogLevel error
  IIopClusterConfig /opt/Borland/BDP/var/domains/base/configurations/j2ee/
mos/myapache/conf/WebClusters.properties
  IIopMapFile /opt/Borland/BDP/var/domains/base/configurations/j2ee/mos/
myapache/conf/UriMapFile.properties
```

Apache IIOP の追加指示文

Apache IIOP 設定をさらにカスタマイズするために、次に示すオプションの追加指示文を使用できます。

指示文	デフォルト	説明
IIopChunkedUploading	（コメントアウトされる）true	Apache が Borland Web コンテナの IIOP コネクタに対して「チャンク形式」アップロードを試みるかどうかを制御します。Apache が IIopUploadBufferSize の値より大きいサイズのデータを「チャンク化」できるようにするには、コメントをはずし、確実に true に設定します。 **メモ：**server.xml で属性 enablechunking="true" を設定して、「チャンク形式」アップロードを Web コンテナでも有効にする必要があります。 Apache が、すべてのデータを収集し終えるまで待機してから、CORBA RPC を呼び出してサイズが大きいデータを Borland Web コンテナに送信する場合は、この指示文をコメントアウトしたままにするか、false に設定します。詳細については、「チャンク形式アップロードの実装」を参照してください。
IIopUploadBufferSize	（コメントアウトされる）4096	IIopChunkedUploading を true に設定した場合に使用される、「チャンク化」バッファサイズを定義します。この指示文は、0 より大きい数値を受け付けます。基本的に、この指示文に指定するバイト数を大きくすると、データを Borland Web コンテナに送信するために必要な CORBA RPC の数が減少します。ただし、この指示文の設定値を大きくすると、トランザクションを処理する際に消費されるメモリが増えます。このパラメータを調整してパフォーマンス特性を微調整できます。これにより、管理者は、メモリリソースの使用量より RPC コストを重視してシステム上のアップロードを最適化できます。 **メモ：**無効な値（数値以外の値や負数）が存在すると、デフォルト値の 4096 が使用されます。詳細については、「チャンク形式アップロードの実装」を参照してください。

Apache IIOP コネクタの設定

Apache IIOP コネクタには、Web サーバーのクラスタ情報で更新する必要がある設定ファイルのセットがあります。これらの IIOP コネクタ設定ファイルは、デフォルトでは次のディレクトリに置かれています。

```
<install_dir>/var/domains/<domain_name>/configurations/
<configuration_name>/mos/<apache_managedobject_name>/conf
```

次の 2 つの設定ファイルがあります。

IIOP 設定ファイル	説明
WebClusters.properties	クラスタと、各クラスタに対応する Web コンテナを指定します。
UriMapFile.properties	WebClusters.properties ファイルで定義したクラスタに URI リファレンスをマッピングします。

メモ　この 2 つの設定ファイルを変更する場合は、Apache Web サーバーまたは CORBA サーバーを起動やシャットダウンは不要です。これは、IIOP コネクタによってファイルが自動的にロードされるためです。

新しいクラスタの追加

`WebClusters.properties` ファイルは、IIOP コネクタに次の情報を伝えます。

- 利用できる各クラスタの名前：（`ClusterList`）
- Web コンテナの ID
- 特定クラスタに自動負荷分散（`enable_loadbalancing`）を提供するかどうか

`WebClusters.properties` ファイルに新しいクラスタを追加するには、次の手順にしたがいます。

1 `ClusterList` に設定したクラスタの名前を追加します。次に例を示します。

```
ClusterList=cluster1,cluster2,cluster3
```

2 次の形式でクラスタ名、必須の `webcontainer_id` 属性、および追加属性（「クラスタ定義属性」の表を参照）を指定する行を追加して、各クラスタを定義します。次に例を示します。

```
<clustername>.webcontainer_id = <id> <attribute>
```

メモ　フェイルオーバーとスマートセッションは常に有効になっています。詳細については、55 ページの「Web コンポーネントのクラスタリング」を参照してください。

属性	必須	定義
webcontainer_id	はい	オブジェクトの「バインド」名、またはクラスタを実装する Web コンテナを識別する corbaloc 文字列。
enable_loadbalancing	いいえ	負荷分散を有効にするには、この属性を省略するか、またはこの属性を省略せずに `true` に設定します。 負荷分散は、デフォルトで有効になっています。負荷分散を無効にするには、`false` に設定して、このクラスタインスタンスが負荷分散技術を使用しないように指定します。 **警告：** enable_loadbalancing 属性の指定時には、正しい値（true か false）を指定してください。

次に例を示します。

```
ClusterList=cluster1,cluster2,cluster3
cluster1.webcontainer_id = tc_inst1
cluster2.webcontainer_id = corbaloc::127.20.20.2:20202,:127.20.20.3:20202/
   tc_inst2
cluster2.enable_loadbalancing = true
cluster3.webcontainer_id = tc_inst3
cluster3.enable_loadbalancing = false
```

上の例では、次の 3 つのクラスタが定義されています。

1 最初のクラスタは、osagent 命名方式を使用し、負荷分散が有効にされています。

2 2 番めのクラスタは、corbaloc 命名方式を使用し、負荷分散が有効にされています。

3 3 番めのクラスタは、osagent 命名方式を使用しますが、負荷分散が無効にされています。

メモ 特定のクラスタの使用を無効にするには、目的のクラスタ名を `ClusterList` リストから削除します。ただし、Web サーバー（アタッチユーザー）にアタッチされたアクティブ HTTP セッションは削除しないでください。これらの「ライブ」セッションを要求しても失敗するだけです。

メモ `WebClusters.properties` ファイルの変更結果は、次回の要求で自動的に有効になります。サーバーを再起動する必要はありません。

新しい Web アプリケーションの追加

重要 デフォルトでは、Apache からは Web アプリケーションを使用できません。Web アプリケーションを Apache から使用できるようにするには、Web アプリケーションデスクリプタにいくつかの情報を追加する必要があります。追加方法の具体的な手順については、『*管理コンソールユーザーズガイド*』の「デプロイメントデスクリプタエディタの使い方」の「Web デプロイメントパス」を参照してください。

Borland Web コンテナにデプロイメントした新しいアプリケーションの場合、次の手順で Apache Web サーバーで利用できるように設定します。`UriMapFile.properties` ファイルを使って `HTTP URI` 文字列を `WebClusters.properties` ファイルで設定されている Web クラスタ名にマッピングします（40 ページの「新しいクラスタの追加」を参照）。

- `UriMapFile.properties` ファイルで次のように入力します。

  ```
  <uri-mapping> = <clustername>
  ```

  ここで、`<uri-mapping>` は、標準 URI 文字列またはワイルドカード文字列です。`<clustername>` は、`WebClusters.properties` ファイルの `ClusterList` エントリに出現するクラスタ名です。

次に例を示します。

```
/examples = cluster1
/examples/* = cluster1

/petstore/index.jsp = cluster2
/petstore/servlet/* = cluster2
```

この例では、次のようになります。

- `/examples` で開始する URI は、Web コンテナや「cluster1」Web クラスタで実行する CORBA オブジェクトに転送されます。
- `/petstore/index.jsp` と同じ URI、または `/petstore/servlet` で始まる URI は、「cluster2」に転送されます。

メモ URI マッピングの場合、ワイルドカード "*" は、URI の**最後**のワードにだけ有効であり、次のような場合が考えられます。

- ワード全体（および下位のすべてのリファレンス）。例：`/examples/*`。
- ファイル指定文字列のファイル名の部分。例：`/examples/*.jsp`。

メモ `UriMapFile.properties` ファイルの変更結果は、次回の要求で自動的に有効になります。サーバーを再起動する必要はありません。

`WebCluster.properties` または `UriMapFile.properties` が変更された場合、それらのファイルは IIOP コネクタによって自動的にロードされます。つまり、Web アプリケーションを追加や削除、クラスタ設定の変更は、Apache Web サーバーや Borland Web コンテナの起動やシャットダウンなしで実行できるわけです。

大量データの転送

ここでは、クライアントと Borland Web コンテナ間で Apache 2.2.3 を通して大量のデータを転送するために使用できる AppServer のオプションについて説明します。転送されるデータは、次のいずれかになります。

- ファイルから取得された静的コンテンツ
- 動的に生成されるコンテンツ

通常、静的コンテンツでは、コンテンツ長があらかじめわかっていますが、動的コンテンツでは不明です。

大量データのダウンロード

Borland Web コンテナからブラウザへ大量のデータをダウンロードする際は、次のモードを利用できます。

- チャンク形式ダウンロード
- 非チャンク形式ダウンロード

チャンク形式ダウンロードの実装

「チャンク形式」ダウンロードモードでは、Borland Web コンテナは、送信するデータをすべて取得し終わるまで待機しません。Web コンテナは、サーブレットがデータを生成するとただちに、Apache を通して固定サイズバッファでのブラウザへのデータ送信を開始します。

データは利用可能になるとすぐに消去されるため、チャンク形式ダウンロード転送モードでは、Apache と Borland Web コンテナの両方で必要なメモリ量が少なくて済みます。ブラウザのユーザーは、転送がすべて完了してから大きなデータをまとめて表示するのではなく、データを受け取りながら表示できます。

チャンク形式ダウンロードの有効化

チャンク形式ダウンロードモードを有効にするには、次に示すパーティションのデータディレクトリに格納されている Borland Web コンテナの `server.xml` ファイルを更新します。

```
adm/tomcat/conf/
```

詳細については、「Web コンポーネント」の 30 ページの「Borland Web コンテナのインプリメンテーション」を参照してください。

チャンク形式ダウンロードを有効にするには、次の手順にしたがいます。

1. Borland Web コンテナの `server.xml` ファイルで、コードの `<Service name="IIOP">` セクションを見つけます。
2. チャンク形式ダウンロードを有効にする IIOP サービスに対して `enableChunking="true"` を指定します。
3. デフォルトでは、ダウンロードバッファサイズは `4096` に設定されています。IIOP サービスのこの値を変更するには、`downloadBufferSize` 属性を次のように使用します。

   ```
   downloadBufferSize=<value>
   ```

   ここで、`<value>` は、`0` より大きい数値です。

メモ　無効な値（数値以外の値や負数）が存在すると、デフォルト値の `4096` が使用されます。

チャンク形式ダウンロード転送モードでは、要求ごとの余分なスレッドによるオーバーヘッドが発生します。

既知のコンテンツ長と不明なコンテンツ長

コンテンツ長が事前にわかっているかどうかによって、チャンク形式ダウンロードモードでは、次の 2 つのいずれかが実行されます。

- コンテンツ長が既知のチャンク形式ダウンロード
- コンテンツ長が不明なチャンク形式ダウンロード

コンテンツ長が既知のチャンク形式ダウンロード

この場合、サーブレットまたは JSP は、転送前に、データのコンテンツ長を知っています。サーブレットは、データを書き出す前に `Content-Length` HTTP ヘッダーを設定します。Borland Web コンテナは、1 つの応答ヘッダーに続けて複数のデータチャンクを書き出します。Apache は、Web コンテナからこのデータを受け取ると、同じ方法でそれをブラウザに送信します。

応答ヘッダーには次のヘッダーが含まれます。

```
Content-Length=<actual data size>
```

コンテンツ長が不明なチャンク形式ダウンロード

HTTP プロトコルバージョン 1.1 では、事前にデータ長がわからない場合のデータ転送処理のために、新しい機能が追加されました。この機能を「HTTP チャンク化」といいます。この場合、サーブレットは、転送前に、データのコンテンツ長を知りません。サーブレットは、`Content-Length` HTTP ヘッダーを設定**しません**。

Borland Web コンテナは、コンテンツ長が事前にわかっているチャンク形式ダウンロードとまったく同じ方法で、データを Apache Web サーバーに送信します。つまり、1 つの応答ヘッダーに続けて複数のデータチャンクを送信します。応答ヘッダーには次のヘッダーが含まれます。

```
Transfer-Encoding="chunked"
```

ブラウザのプロトコルが HTTP 1.1 で、`Content-Length` ヘッダーがサーブレットによって設定されていない場合、Borland Web コンテナが自動的に `Transfer-Encoding="chunked"` ヘッダーを追加します。

Apache Web サーバーは、この `Transfer-Encoding` ヘッダーを認識すると、データを「HTTP チャンク」として送信開始します。つまり、応答ヘッダーに続けて、「チャンク化」ヘッダー、「チャンク化」データ、「チャンク化」トレーラの組合せを複数送信します。

メモ HTTP 1.1 仕様により、サーブレットが `Content-Length` と `Transfer-Encoding` ヘッダーの両方を設定した場合、`Content-Length` ヘッダーは、Borland Web コンテナによってドロップされます。

HTTP 1.0 プロトコルだけをサポートするブラウザ

ブラウザが HTTP プロトコルのバージョン 1.0 以下しかサポートしておらず、サーブレットが `Content-Length` ヘッダーを設定しない場合、Borland Web コンテナは `Transfer-Encoding` ヘッダーを自動的に追加できません。HTTP 1.0 プロトコルに対して、`Transfer-Encoding` ヘッダーを使用しても意味がないためです。この場合、Borland Web コンテナは、次のように動作します。

1. データがなくなるまで、すべてのデータをバッファに入れます。
2. コンテンツ長を計算します。
3. 自身で `Content-Length` ヘッダーを設定します。

Borland Web コンテナがこのバッファリング動作を実行しないようにするには、IIOP コネクタ属性 `canBufferHttp10Data="false"` を設定します。デフォルトでは、この属性は `true` に設定されています。

メモ `canBufferHttp10Data` 属性が `false` に設定されている場合、次のエラーメッセージがブラウザに送信されます。

```
Servlet did not set the Content-Length
```

非チャンク形式ダウンロードの実装

これは、IIOPコネクタのデフォルト のデータ転送モード です。「 非チャンク形式」ダウンロード モード では、Borland Webコンテナは、送信するデータをすべて取得し 終わるまで待機し ます。次に、コンテンツ長を計算し 、`Content-Length` ヘッダーに実際のコンテンツ長を設定し ます。そして、その応答ヘッダーに続けて 1 つの大きなデータブロックを送信し ます。

この転送モードでは、データがすべて利用可能になるまでデータをキャッシュするため、Apache Web サーバーと Borland Web コンテナの両方で必要メモリ量が増大し ます。また、データがすべて転送し 終わらないと、ブラウザのユーザーはデータを表示できません。

非チャンク形式ダウンロード転送モードでは、要求ごとの余分なスレッドによるオーバーヘッドは発生し ません。このダウンロードモードでは `Transfer-Encoding` ヘッダーがまったく設定されないため、このモードは、HTTP プロトコルのバージョン 1.0 および 1.1 でうまく機能し ます。

大量データのアップロード

クライアントによって開始された大量データのアップロードでは、次のモードを利用できます（この場合、クライアントとは、ブラウザ、または HTTP を使用するブラウザ以外のクライアント、たとえば Java などです）。

- チャンク形式アップロード
- 非チャンク形式アップロード

ブラウザは、常に、データを「チャンク化された」状態で Apache Web サーバーに送信します。この場合の「チャンク形式」アップロードおよび「非チャンク形式」アップロードは、Apache Web サーバーと Borland Web コンテナ間のデータ転送モードを指します。

チャンク形式アップロードの実装

デフォルトでは、Apache は、大量のデータを「チャンク」形式でアップロードしようとします。このモードでは、Apache は、ブラウザからすべてのデータを取得するまで待たずに、Borland Web コンテナへのデータの送信を開始します。Apache は、データをブラウザから取得できるようになると、固定サイズバッファでデータを送信します。

データは可能になるとすぐに消去されるため、チャンク形式アップロード転送モードでは、Apache と Borland Web コンテナの両方で必要なメモリ量が少なくて済みます。

チャンク形式転送モードでは、Borland Web コンテナで要求ごとの余分なスレッドによるオーバーヘッドが発生します。

チャンク形式アップロードの有効化

チャンク形式アップロードモードを有効にするには、次のファイルを**両方とも**更新する必要があります。

- Borland Web コンテナの `server.xml` ファイル。このファイルは、次に示すパーティションのデータディレクトリに格納されています。

  ```
  adm/tomcat/conf
  ```

  詳細については、30 ページの「Borland Web コンテナのインプリメンテーション」を参照してください。

- Apache の `httpd.conf` ファイル。このファイルは、デフォルトでは、次のディレクトリにあります。

  ```
  <install_dir>/var/domains/<domain_name>/configurations/
  <configuration_name>/mos/<apache_managedobject_name>/conf
  ```

  詳細については、27 ページの「Apache 設定」を参照してください。

チャンク形式アップロードを有効にするには、次の手順にしたがいます。

1 Borland Web コンテナの `server.xml` ファイルで、コードの `<Service name="IIOP">` セクションを見つけます。

2 デフォルトでは、enableChunking 属性は false に設定されています。この値を enableChunking="true" に変更します。

3 Apache httpd.conf ファイルで、次の IIOP 指示文を探し、コメントをはずします。

```
#IIopChunkedUploading true
```

メモ チャンク形式アップロード転送モードでは、Borland Web コンテナで要求ごとの余分なスレッドによるオーバーヘッドが発生します。

アップロードバッファサイズの変更

デフォルトでは、IIopUploadBufferSize は、4096 バイトに設定されています。この値を変更するには、次の手順にしたがいます。

1 Apache httpd.conf で、コメントアウトされている次の指示文を見つけます。

```
#IIopUploadBufferSize 4096
```

2 この指示文のコメントをはずし、次のように設定します。

```
IIopUploadBufferSize <value>
```

ここで、<value> は、0 より大きい数値です。

メモ 無効な値（数値以外の値や負数）を指定すると、デフォルト値の 4096 が使用されます。

既知のコンテンツ長と不明なコンテンツ長

コンテンツ長が事前にわかっているかどうかによって、チャンク形式アップロードモードでは、次の 2 つのいずれかが実行されます。

- コンテンツ長が既知のチャンク形式アップロード
- コンテンツ長が不明なチャンク形式アップロード

コンテンツ長が既知のチャンク形式アップロード

この場合、クライアントは、転送前に、データのコンテンツ長を知っています。クライアントは、データを書き出す前に Content-Length HTTP ヘッダーを設定します。クライアントは、1 つの応答ヘッダーに続けて複数のデータチャンクを書き出します。Apache は、ブラウザからこのデータを受け取ると、同じ方法でそれを Borland Web コンテナに送信します。

応答ヘッダーには次のヘッダーが含まれます。

```
Content-Length=<actual data size>
```

コンテンツ長が不明なチャンク形式アップロード

HTTP プロトコルバージョン 1.1 では、事前にデータ長がわからない場合のデータ転送処理のために、新しい機能が追加されました。この機能を「HTTP チャンク化」といいます。

この場合、クライアントは、転送前に、データのコンテンツ長を知りません。クライアントは、Content-Length HTTP リクエストヘッダーを設定**しません**。クライアントは、次に示すように、Transfer-Encoding HTTP リクエストヘッダーに chunked の値を設定します。

```
Transfer-Encoding="chunked"
```

クライアントは、データを Apache Web サーバーに「HTTP チャンク」として送信します。つまり、1 つのリクエストヘッダーに続けて、「チャンク化ヘッダー」、「チャンク化データ」、「チャンク化トレーラ」の組合せを複数送信します。

Apache Web サーバーは、この Transfer-Encoding ヘッダーを認識すると、チャンク化ヘッダーとチャンク化トレーラを削除し、データを通常のデータチャンクとして Borland Web コンテナに送信します。

現時点で、主要ブラウザで、コンテンツ長が不明なままでのデータのアップロードをサポートするものはありません。つまり、ブラウザが Transfer-Encoding="chunked" ヘッダーを HTTP 要求に追加することはありません。ただし、ブラウザ以外のクライアントは、このヘッダーを HTTP 要求に追加できます。

非チャンク形式アップロードの実装

これは、IIOP コネクタのデフォルトのデータ転送モードです。「非チャンク形式」アップロードモードでは、Apache Web サーバーは、送信するデータをすべて取得し終わるまで待機します。次に、コンテンツ長を計算し、`Content-Length` ヘッダーに実際のコンテンツ長を設定します。次に、Apache は、そのリクエストヘッダーに続けて 1 つの大きなデータブロックを送信します。

この転送モードでは、データがすべて利用可能になるまでデータをキャッシュするため、Apache Web サーバーと Borland Web コンテナの両方で必要メモリ量が増大します。

非チャンク形式アップロード転送モードでは、（Borland Web コンテナで）要求ごとの余分なスレッドによるオーバーヘッドが発生しません。このダウンロードモードでは `Transfer-Encoding` ヘッダーがまったく設定されないため、このモードは、HTTP プロトコルのバージョン 1.0 および 1.1 でうまく機能します。

IIS Web サーバーと Borland Web コンテナの接続

ここでは、Tomcat ベースの Borland Web コンテナ、その IIOP コネクタ、および IIS/IIOP リダイレクタについて説明します。IIS/IIOP リダイレクタは、Microsoft IIS (Internet Information Services) Web サーバー（BES 製品には含まれない）と Borland Web コンテナを接続します。以上の機能は AppServer とともに提供され、VisiBroker Edition の一部として任意でインストールできます（Web サービスパック（VisiExchange）コンポーネント）。詳細については、『*インストールガイド*』の「AppServer の Windows へのインストール」または「AppServer の Solaris または Linux へのインストール」のセクションを参照してください。

Borland Web コンテナにおける IIOP 設定の変更

`server.xml` は、Borland Web コンテナのメインの設定ファイルで、パーティションのデータディレクトリに格納されています。

```
adm/tomcat/conf/
```

`server.xml` ファイルには、IIOP コネクタ設定に関するセクションがあります。設定については、「*Apache と Borland Web コンテナの接続*」の 35 ページの「Borland Web コンテナの IIOP 設定の変更」を参照してください。

Microsoft Internet Information Services（IIS）サーバー固有の IIOP 設定

IIS/IIOP リダイレクタをシステムで使用する前に、以下の処理を完了する必要があります。

- IIS を実行している Windows 2003/XP/2000 システムの設定
- IIS/IIOP リダイレクタ設定

IIS を実行している Windows 2003/XP/2000 システムの設定方法

1 SYSTEM 環境に必要な環境変数 `OSAGENT_PORT` を追加します。

IISredirectory は VisiBroker に依存して IIS と Borland Web コンテナの間に IIOP 通信層を提供します。VisiBbroker が機能するには、以下のように環境変数を定義する必要があります。

環境変数	値	説明
OSAGENT_PORT	14000	AppServer が使用する OSAGENT ポート番号の数値。

重要 `OSAGENT_PORT` 環境変数を設定した後に IIS に認識させるために、Windows システムを再起動する必要があります。

2 IIS/IIOP リダイレクタを ISAPI フィルタとして追加します。

a ［マイコンピュータ］を右クリックして［管理］を選択します。

［コンピュータの管理］ダイアログボックスが表示されます。

b ツリーを展開し、［サービスとアプリケーション］ノードを展開します。

c ［インターネット インフォメーション サービス］ノードを展開します。

d ［既定の Web サイト］ノードを右クリックし、［プロパティ］を選択します。

［既定の Web サイトのプロパティ］ダイアログボックスが表示されます。

e ［ISAPI フィルタ］タブに移動します。

f ［追加］をクリックします。

g ［フィルタのプロパティ］ダイアログボックスで、フィルタ名と実行可能プログラムのパスを、対応するエントリボックスに入力します。

次の例のように［フィルタ名］は、対象のタスクを反映することになっています。

```
iisredir2
```

また、［実行ファイル］は、`<install_dir>¥bin` 内の `iisredir2.dll` を指定します。次に例を示します。

```
C:¥borland¥BDP¥bin¥iisredir2.dll
```

h ［OK］をクリックします。

新しい ISAPI フィルタがリストに表示されます。フィルタプロパティを変更する必要はありません。

i ［OK］をクリックします。

3 IIS Web サイトに「Borland」仮想ディレクトリを追加します。

a ［コンピュータの管理］ダイアログボックスで、［既定の Web サイト］を右クリックして、［新規作成｜仮想ディレクトリ］を選択します。

仮想ディレクトリ作成ウィザードが表示されます。

b ［次へ］をクリックします。

c ［エイリアス］に「Borland」と入力します。

IIS Web サーバーが URI、`http://localhost/borland/iisredir2.dll` に応答するとき、IIS/IIOP リダイレクタ拡張子を検索するには、`borland` 仮想ディレクトリが必要です。

d ［ディレクトリ］で、`<install_dir>¥bin` を参照します。

e ［次へ］をクリックして続行します。

f ［アクセス許可］で、デフォルトで選択される「読み取り」と「ASP などのスクリプトを実行する」のほかに「ISAPI アプリケーションや CGI などを実行する」を選択します。

g ［次へ］をクリックします。

h ［完了］をクリックします。

4 **Windows 2003 のみ：**Windows 2003 に対して ISAPI 拡張機能のアクセス許可を設定します。

IIS のバージョンによって、IIS にロードできるアプリケーションの拡張機能が制限されます。すべての拡張機能を有効にするか、または各自の IIS インストールで実行できる ISAPI 拡張機能だけを選択できます。以下の手順は、`iisredir2` 拡張機能だけを有効にします。

a ［コンピュータの管理］ダイアログボックスで［サービスとアプリケーション］を開きます。

b ［インターネット インフォメーション サービス］を開きます。

c ［Web サービス拡張］を開き、［Add a new Service Extension］をクリックします。

d "iisredir2.dll" 拡張機能に名前を付けます。

e ［追加］ボタンを使って <install>¥bin¥iisredir2.dll を探します。

f このファイルを選択します。

g ［Extension allowed］チェックボックスをチェックします。

h ［OK］をクリックします。

5 IIS Service をいったん終了してから再開して、IIS を再起動します。

a ［コンピュータの管理］ダイアログボックスで、［インターネット インフォメーション サービス］ノードを右クリックし、［すべてのタスク｜ IIS を再起動します］を選択します。

b ［終了／起動／再起動］ダイアログボックスで、ドロップダウンから「<name of your IIS web server> のインターネットサービスを終了」を選択します。

c ［OK］をクリックします。

Web サービスは、IIS 管理者がロードした dlls をアンロードします。

d サーバーのシャットダウンが完了したら、［インターネット インフォメーション サービス］ノードを右クリックして［すべてのタスク｜ IIS を再起動します］を選択します。

e ［停止／開始／再起動］ダイアログで、［<IIS Web サーバの名前 > のインターネットサービスを開始します］を選択します。

f ［OK］をクリックします。

Web サービスは、IIS 管理者がロードした dlls をアンロードします。

6 `iisredir2` フィルタが有効であるか確認します。

a ［コンピュータの管理］ダイアログボックスで、［既定の Web サイト］ノードを右クリックして、［プロパティ］を選択します。

b ［既定の Web サイトのプロパティ］ダイアログボックスで、［ISAPI フィルタ］タブに移動します。

c `iisredir2` フィルタに、有効であることを示す緑の上向き矢印が表示されます。

表示されない場合、`iisredir2.log` ファイルで、正しくロードされない理由を調べます。このファイルは次のサイトから入手できます。

`<install_dir>¥etc¥iisredir2¥logs`.

d ［OK］をクリックして終了します。

7 IIS Web サーバーから `¥examples` コンテキストにアクセスします。

先の手順を実行しておくと、IIS Server の再起動後に `¥examples` コンテキストにアクセスできます。

メモ このサンプルコードでは、Web サーバーのポート番号は、サイト用に設定した値に合わせます。たとえば、IIS 管理者がポート `6060` を監視するよう IIS を設定した場合、有効な URL は次のとおりです。

```
http://localhost:6060/examples
```

もちろん、IIS が Microsoft のデフォルトにしたがって設定されている場合、ポート `80` が監視対象となり、その場合はポート番号を省略できます。次に例を示します。

```
http://localhost/examples
```

IIS/IIOP リダイレクタ設定

IIS/IIOP リダイレクタには、Web サーバーのクラスタ情報で更新する必要がある設定ファイルのセットがあります。これらの IIOP リダイレクタ設定ファイルは、デフォルトでは次のディレクトリに置かれています。

```
<install_dir>/etc/iisredir2/conf
```

次の設定ファイルがあります。

IIOP 設定ファイル	説明
WebClusters.properties	クラスタと、各クラスタに対応する Web コンテナを指定します。
UriMapFile.properties	WebClusters.properties ファイルで定義したクラスタに URI リファレンスをマッピングします。

メモ この 2 つの設定ファイルを変更する場合は、IIS Web サーバーまたは Borland Web コンテナを起動やシャットダウンは不要です。これは、IIOP リダイレクタによってファイルが自動的にロードされるためです。

新しいクラスタの追加

`WebClusters.properties` ファイルは、IIOP リダイレクタに次の情報を伝えます。

- 利用できる各クラスタの名前：(`ClusterList`)
- Web コンテナの ID
- 特定クラスタに自動負荷分散（`enable_loadbalancing`）を提供するかどうか

`WebClusters.properties` ファイルに新しいクラスタを追加するには、次の手順にしたがいます。

1 `ClusterList` に設定したクラスタの名前を追加します。次に例を示します。

```
ClusterList=cluster1,cluster2,cluster3
```

2 次の形式でクラスタ名、必須の `webcontainer_id` 属性、および追加属性（「クラスタ定義属性」の表を参照）を指定する行を追加して、各クラスタを定義します。次に例を示します。

```
<clustername>.webcontainer_id = <id> <attribute>
```

メモ フェイルオーバーとスマートセッションは常に有効になっています。詳細については、55 ページの「Web コンポーネントのクラスタリング」を参照してください。

属性	必須	定義
webcontainer_id	はい	オブジェクトの「バインド」名、またはクラスタを実装する Web コンテナを識別する corbaloc 文字列。
enable_loadbalancing = true|false	いいえ	負荷分散を有効にするには、この属性を省略するか、またはこの属性を省略せずに `true` に設定します。負荷分散は、デフォルトで有効になっています。負荷分散を無効にするには、`false` に設定して、このクラスタインスタンスが負荷分散技術を使用しないように指定します。 **警告：** enable_loadbalancing 属性の指定時には、正しい値（true か false）を指定してください。

次に例を示します。

```
ClusterList=cluster1,cluster2,cluster3
cluster1.webcontainer_id = tc_inst1
cluster2.webcontainer_id = corbaloc::127.20.20.2:20202,:127.20.20.3:20202/
   tc_inst2
cluster2.enable_loadbalancing = true
cluster3.webcontainer_id = tc_inst3
cluster3.enable_loadbalancing = false
```

上の例では、次の 3 つのクラスタが定義されています。

1 最初のクラスタは、osagent 命名方式を使用し、負荷分散が有効にされています。

2 2 番めのクラスタは、corbaloc 命名方式を使用し、負荷分散が有効にされています。

3 3 番めのクラスタは、osagent 命名方式を使用しますが、負荷分散が無効にされています。

メモ 特定のクラスタの使用を無効にするには、目的のクラスタ名を `ClusterList` リストから削除します。ただし、Web サーバー（アタッチユーザー）にアタッチされたアクティブ HTTP セッションは削除しないでください。これらの「ライブ」セッションを要求しても失敗するだけです。

メモ `WebClusters.properties` ファイルの変更結果は、次回の要求で自動的に有効になります。サーバーを再起動する必要はありません。

新しい Web アプリケーションの追加

重要 デフォルトでは、Web アプリケーションは、IIS を通して利用できません。Web アプリケーションを IIS から使用できるようにするには、Web アプリケーションデスクリプタにいくつかの情報を追加する必要があります。追加方法の具体的な手順については、『*管理コンソールユーザーズガイド*』の「デプロイメントデスクリプタエディタの使い方」の「Web デプロイメントパス」を参照してください。

`¥examples` コンテキストは、IIS/IIOP インストール設定を確認するのに便利ですが、Borland Web コンテナにデプロイメントした新しいアプリケーションの場合、次の手順で IIS Web サーバーで利用できるように設定します。`UriMapFile.properties` ファイルを使って `HTTP URI` 文字列を `WebClusters.properties` ファイルで設定されている Web クラスタ名にマッピングします（40 ページの「新しいクラスタの追加」を参照）。

- `UriMapFile.properties` ファイルで次のように入力します。

  ```
  <uri-mapping> = <clustername>
  ```

  ここで、`<uri-mapping>` は、標準 URI 文字列またはワイルドカード文字列です。`<clustername>` は、`WebClusters.properties` ファイルの `ClusterList` エントリに出現するクラスタ名です。

次に例を示します。

```
/examples = cluster1
/examples/* = cluster1

/petstore/index.jsp = cluster2
/petstore/servlet/* = cluster2
```

この例では、次のようになります。

- `/examples` で開始する URI は、Web コンテナや「cluster1」Web クラスタで実行する CORBA オブジェクトに転送されます。
- `/petstore/index.jsp` と同じ URI、または `/petstore/servlet` で始まる URI は、「cluster2」に転送されます。

メモ URI マッピングの場合、ワイルドカード "*" は、URI の最後のワードにだけ有効であり、次のような場合が考えられます。

- ワード全体（および下位のすべてのリファレンス）。例：`/examples/*`。
- ファイル指定文字列のファイル名の部分。例：`/examples/*.jsp`。

メモ `UriMapFile.properties` ファイルの変更結果は、次回の要求で自動的に有効になります。サーバーを再起動する必要はありません。

`WebCluster.properties` または `UriMapFile.properties` が変更された場合、それらのファイルは IIOP リダイレクタによって自動的にロードされます。つまり、Web アプリケーションを追加や削除、クラスタ設定の変更は、IIS Web サーバーや Borland Web コンテナの起動やシャットダウンなしで実行できるわけです。

第 6 章

Java セッションサービス（JSS）の設定

Java セッションサービス（JSS）は特定のユーザーセッションに関係する情報を格納するサービスです。コンテナで障害が発生した場合の回復用のセッション情報を JSS に保存します。

Borland は、JSS を使用するためのインターフェース定義言語（IDL）インターフェースを提供しています。2 つのインプリメンテーション（DataExpress を使用する場合と JDBC 機能を持つ任意のデータベースを使用する場合）がバンドルされています。

JSS を使用すると、セッション情報をデータベースに簡単に保存することができます。たとえば、ショッピングカートの場合、JSS はショッピングカート内の品目数などのセッション情報を取得して保存します。これにより、Borland Web コンテナの予定外のシャットダウンでセッションが中断されても、JSS を介して別の Borland Web コンテナのインスタンスからセッション情報を回復できます。JSS はローカルネットワークで実行してください。クラスタ構成内の Web コンテナは、JSS を見つけ出して接続し、セッション管理を続行します。

Borland Web コンテナの詳細については、30 ページの「Borland Web コンテナのインプリメンテーション」を参照してください。

JSS によるセッション管理

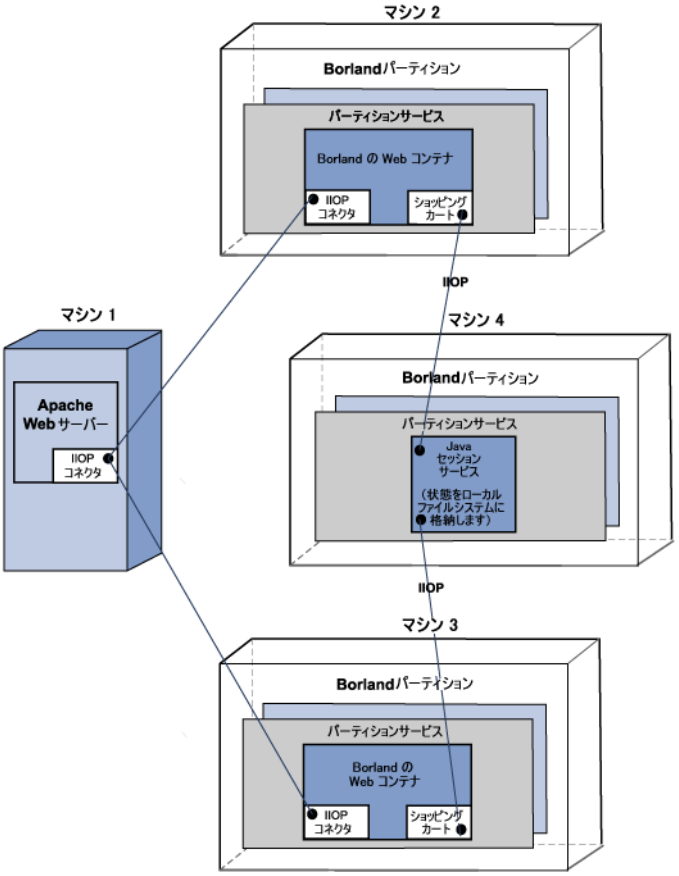
次の図は、Web コンポーネントの一般的な構造と、JSS でセッション情報を管理する方法を示します。JSS セッション管理はクライアントに対して完全に透過です。

一元化された JSS と 2 つの Web コンテナによる JSS 管理の図には、次の 4 つの仮想マシンがあります。

- 最初のマシンは、Apache Web サーバーをホストしています。
- ほかの 2 つのマシンには、Borland Web コンテナがあります。
- 4 番めのマシンは、JSS とリレーショナルデータベース（JDataStore または JDBC データソース）をホストしています。

Apache Web サーバー（マシン 1）から最初の Web コンテナのインスタンス（マシン 2）にクライアント要求を渡したときに障害が発生すると、第 2 の Web コンテナのインスタンス（マシン 3）は、JSS（マシン 4）からセッション情報を取得することで、クライアント要求の処理を続行できます。ショッピングカートの品目情報が保持され、クライアント要求の処理は続行されます。

図 6.1　一元化された JSS と 2 つの Web コンテナによる JSS 管理

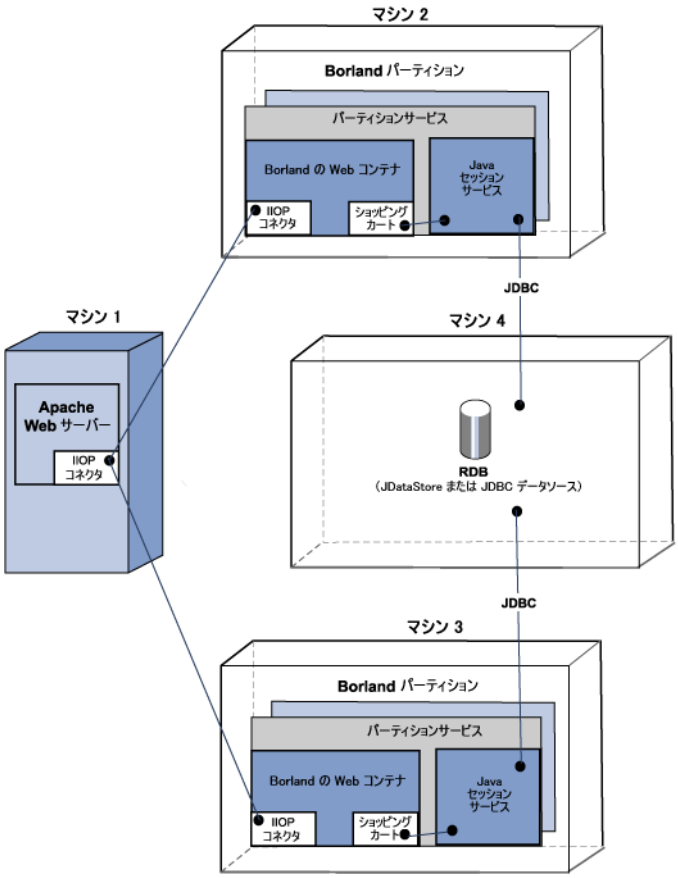


2 つの Web コンテナと一元化されたバックエンドデータストアによる JSS 管理には、次の 4 つの仮想マシンがあります。

- 最初のマシンは、Apache Web サーバーをホストしています。
- ほかの 2 つのマシンには、JSS をホストするマシンのほかに Borland Web コンテナのインスタンスも含まれています。
- 4 番めのマシンは、リレーショナルデータベース（JDataStore または JDBC データソース）をホストしています。

Apache Web サーバー（マシン 1）から最初の Web コンテナのインスタンス（マシン 2）にクライアント要求を渡したときに障害が発生すると、第 2 の Web コンテナのインスタンス（マシン 3）は、JSS（マシン 4）からセッション情報を取得することで、クライアント要求の処理を続行できます。ショッピングカートの品目情報が保持され、クライアント要求の処理は続行されます。

図 6.2　2 つの Web コンテナと一元化されたバックエンドデータストアによる JSS 管理



JSS の管理と設定

JSS 設定は自身のプロパティで定義されます。BES は 2 つのタイプの設定をサポートします。デフォルトでは JDatastore を使用しますが、任意の JDBC データソースをサポートしています。

- JSS は、JDataStore ファイルを使用する設定となり、データベーステーブルは JSS によって自動的に生成されます。
- JSS が、JDBC データソースを使用する設定の場合、システム管理者が次の SQL 文で 3 つのデータベーステーブルをバックエンドデータベースであらかじめ作成しておく必要があります。

```
CREATE TABLE "JSS_KEYS" ("STORAGE_NAME" java_string primary key, "KEY_BASE"
java_float);
CREATE TABLE "JSS_WEB" ("KEY" java_string primary key, "VALUE"
java_serializable, "EXPIRATION" java_float);
CREATE TABLE "JSS_EJB" ("KEY" java_string primary key, "VALUE"
java_serializable, "EXPIRATION" java_float);
```

メモ　前述の SQL ステートメントを使用するときは、使用するデータベースで使用される同等のデータ型に置き換えてください。

JSS は、パーティションの一部としてほかのパーティションサービスと並行して実行できます。

JSS パーティションサービスの設定

JSS 設定情報は、「パーティションサービス」として、各パーティションのデータディレクトリの `partition.xml` ファイルにあります。デフォルトでは、このファイルは次のディレクトリにあります。

```
<install_dir>/var/domains/base/configurations/<configuration_name>/mos/
<partition_name>/adm/properties.
```

たとえば、「MyPartition」というパーティションの場合、JSS 設定情報はデフォルトで次の場所にあります。

```
<install_dir>/var/domains/base/configurations/<configuration_name>/mos/
mypartition/adm/properties/partition.xml
```

詳細については、350 ページの「<services> 要素」を参照してください。

また、パーティションのデータディレクトリの場所については、次の場所にある `configuration.xml` ファイルを参照してください。

```
<install_dir>/var/domains/base/configurations/<configuration_name>/
```

そして、パーティション管理オブジェクトのディレクトリ属性を検索します。

```
<partition-process directory=
```

セッションサービス（JSS）レベルのプロパティのリストと説明については、「*EJB*、*JSS*、*および JTS のプロパティ*」の 364 ページの「Java セッションサービス（JSS）のプロパティ」を参照してください。

第 7 章

Web コンポーネントのクラスタリング

ここでは、Apache Web サーバーや Tomcat ベースの Borland Web コンテナを含む複数の Web コンポーネントのクラスタリングについて説明します。一般的なデプロイメント事例では、複数の Borland パーティションでスケーラブルな **n** 層ソリューションを提供します。

Borland パーティションごとに、同じサービスを設定したり、異なるサービスを設定することができます。クラスタリング方式に応じて、これらのサービスのオンとオフを切り替えてください。いずれにしても、このようなリソースをまとめて活用したり、クラスタリングすることで、Web アプリケーションのデプロイメント効率を高めることができます。Web コンポーネントのクラスタリングには、セッション管理、負荷分散、およびフォールトトレランス（フェイルオーバー）などが関係します。

ステートレスとステートフルの接続サービス

クライアントとサーバー間の対話には、ステートレスとステートフルという 2 種類のサービスがあります。*ステートレスサービス*ではクライアントとサーバー間の状態が保持されません。クライアント要求の処理中は、サーバーとクライアント間に「対話」はありません。*ステートフルサービス*では、クライアントとサーバーによって情報ダイアログが管理されます。

Borland Web コンテナの設定ファイルの場所については、30 ページの「Borland Web コンテナのインプリメンテーション」を参照してください。

Borland IIOP コネクタ

IIOP コネクタは、http Web サーバーで Borland Web コンテナに要求をリダイレクトするためのソフトウェアです。Borland AppServer（AppServer）には、Apache 2.2.3 と Microsoft Internet Information Server（IIS）バージョン 5.0、5.1、および 6.0 の Web サーバーの IIOP コネクタが含まれています。http 要求のリダイレクションを処理するジョブは、次の 2 つのコンポーネント間にまたがっています。

- Web サーバーで実行するネイティブライブラリ
- Web コンテナで実行する jar ファイル

AppServer は Web コンポーネントのクラスタリングをサポートします。Borland IIOP コネクタでは、IIOP プロトコルを使用します。次のような独自機能を備えています。

- 負荷分散サポート
- フォールトトレランス（フェイルオーバー）
- スマートセッション処理

負荷分散サポート

*負荷分散*は、http 要求を Web コンテナセット間で宛先を指定して転送する能力です。この能力は、システム管理者が http 通信の負荷を複数の Web コンテナ間に分散させるときに使用します。負荷分散技術により、システムのスケーラビリティを大幅に改善できます。Borland IIOP コネクタでは、次の 2 とおりの方法で負荷分散を設定できます。

- OSAgent 方式の負荷分散
- Corbaloc 方式の負荷分散

OSAgent 方式の負荷分散

この方法は処理が簡単で、設定作業も最小限で済みます。この設定では、多くの Borland Web コンテナのインスタンスを起動し、それらの Borland Web コンテナの IIOP コネクタに同じ名前を付けます。

`name` 属性の設定の詳細については、35 ページの「Borland Web コンテナの IIOP 設定の変更」を参照してください。

Apache では、要求ごとに Borland Web コンテナインスタンス間で負荷を分散します。基本的に、Apache は要求ごとに新しいバインドを実行します。新しく起動した Borland Web コンテナは動的に発見できます。

重要 すべての Borland Web コンテナと Apache は同じ ORB ドメインで実行しておきます。したがって、OSAgent 方式の負荷分散は、ORB ドメインごとにパーティションが異なる状況では使用できません。

Corbaloc 方式の負荷分散

この方式では、クラスタを構成する静的な Web コンテナを使用します。ただし、ORB ドメイン間にまたがることができます。この場合、CORBA corbaloc セマンティクスで、Web コンテナが実行する場所を指定します。次に例を示します。

```
corbaloc::172.20.20.28:30303,:172.20.20.29:30304/tc_inst1
```

上の corbaloc サンプル文字列で、

- 「`tc_inst1`」という名前の Web コンテナに 2 つの TCP/IP エンドポイントを設定します。
- 「`tc_inst1`」というオブジェクト名の Web コンテナを、ホスト `172.20.20.28` で、ポート `30303` に IIOP コネクタを割り当てて実行します。
- ホスト `172.20.20.29` には、ポート `30304` を監視する IIOP コネクタで、同じオブジェクト名で実行している Web コンテナがあります。

`port` 属性の設定の詳細については、35 ページの「Borland Web コンテナの IIOP 設定の変更」を参照してください。

Web サーバー側 IIOP コネクタは、`orb.string_to_object` で、この corbaloc 文字列を CORBA オブジェクトに変換し、VisiBroker の基底の機能により、corbaloc 文字列で指定したこれらの「エンドポイント」間に負荷を分散します。エンドポイントの数に制限はありません。

メモ リストされた Web コンテナのすべてが負荷分散のために実行する必要はありません。ORB は、有効な接続が得られるまで、次のエンドポイントに移行します。

ただし、corbaloc 方式の負荷分散では、corbaloc 式のオブジェクトネーミングで対応できるように、既知のポートで Web コンテナの IIOP コネクタを起動しておく必要があります。次に示すのは、必要な Web コンテナ IIOP コネクタ設定の一部です。

```
<Connector className="org.apache.catalina.connector.iiop.IiopConnector"
name="tc_inst1"  port="30303"/>
```

このコードでは、IIOP コネクタをポート `30303` で起動し、Borland Web コンテナオブジェクト「`tc_inst1`」を指定します。`port` 属性は省略できます。ただしポートを省略すると、ORB によってランダムポートが選択されるため、corbaloc 方式でオブジェクトを検索できなくなります。

組織によっては、使用する Web コンテナや IIOP ポート、あるいはポートの範囲の命名方法に規則を設けている場合があります。

フォールトトレランス（フェイルオーバー）

osagent バインド命名方式と corbaloc 命名方式を使用するフェイルオーバーは、どの場合も自動的に処理されます。corbaloc 命名方式では、corbaloc 名前文字列で次に設定したエンドポイントが処理対象となり、サイクル方式で次々処理され、最後は corbaloc 文字列のすべてのエンドポイントが処理されます。

osagent バインド命名方式では、osagent によってクライアントが代替（ただし等価）オブジェクトインスタンスに自動的にリダイレクトされます。

メモ osagent に利用できるオブジェクトがない場合、あるいは corbaloc 名前文字列で指定したエンドポイントが実行していない場合、http 要求は失敗します。

スマートセッション処理

セッションを指定しなければ、IIOP コネクタはラウンドロビン方式で無差別に処理します。ただし、セッションを指定するときは、セッションを開始した Web コンテナまで、Apache でセッション要求をルート指定することが大事です。

ほかの http 対サーブレットリダイレクタの場合（または IIOP コネクタの初期バージョンの場合）、これは、Web サーバーのキャッシュの sessions-ids-to-web-container-id's のリストを管理することで達成しています。ただしその場合、リストの状態管理に伴ってさまざまな問題が発生します。まず、リストのサイズが非常に大きくなるため、システムリソースが浪費されがちです。また、時代遅れの方式であり、セッションがタイムアウトになるなど、一般に、Web サーバーと Web コンテナ間のリダイレクションの枠組みの非効率的で問題の多い側面だと言えます。

IIOP コネクタでは、「スマートセッション ID」という技術でこの問題を解決しています。この場合、Web コンテナの IOR は、Web コンテナがセッションクッキー（URL 再書き込みの場合は URL）の一部として返すセッション ID 内に埋めこまれます。

セッション ID を生成するとき、Web コンテナは要求が IIOP コネクタを起点とする要求であるかどうかを判定します。IIOP コネクタが起点の場合、要求の取得元である IIOP コネクタの文字列化された IOR を取得します。Web コンテナは標準セッション ID を生成します。通常の ID のほかに、文字列化された IOR が前に追加されます。次に例を示します。

```
Stringified IOR: IOR:xyz
Normal session ID: abc
The new session ID: xyz_abc
```

元の Web コンテナが停止すると、フェイルオーバーが起動し、等価な Web コンテナの別のインスタンスを検索します。

corbaloc 識別 Web コンテナの場合、自動 osagent フェイルオーバーは不確実なので、IIOP コネクタは手動「リバインド」を実行して、実行中の等価 Web コンテナまでの有効なリファレンスを取得します。

ほかに実行している Web コンテナが明らかにない場合、http 要求は失敗します。

新しい Web コンテナは、セッションデータベースから古い状態を取得し、引き続き要求にサービスを提供します。応答を返すとき、新しい Web コンテナでは、その IOR を反映してセッション ID を変更します。ブラウザクライアントへの戻りで Apache はセッション ID を確認しないので、以上の処理は Apache には透過で処理されます。

JSS による Web コンテナの設定

セッションを呼び出すとき、正しくフェイルオーバーを適用するには、同じ JSS バックエンドで Web コンテナを設定します。

フェイルオーバーに使用する Borland Web コンテナの変更

Web アプリケーションごとに、次のサンプルコードと同様のエントリを、Borland Web コンテナの設定ファイル `server.xml` に追加します。`server.xml` ファイルの詳細については、35 ページの「Borland Web コンテナの IIOP 設定の変更」を参照してください。

```
<Manager className="org.apache.catalina.session.PersistentManager">
          <Store className="org.apache.catalina.session.BorlandStore"
          storeName="jss_factory"/>
      </Manager>
```

前のコードでは、ストレージクラス `BorlandStore` を持つ `PersistentManager` の使用を指定しています。`jss_factory` という名前の `BorlandStore` ファクトリとの接続も指定しています。そのファクトリ名で、ローカルネットワークで実行する JSS が必要です。

`jss.factoryName` については、364 ページの「Java セッションサービス（JSS）のプロパティ」を参照してください。

セッションストレージのインプリメンテーション

クラスタリングした Web コンポーネント用にセッションストレージを実装する方法は、次の 2 とおりがあります。

- プログラム的インプリメンテーション
- 自動的インプリメンテーション

プログラム的インプリメンテーション

*プログラム的インプリメンテーション*では、セッション属性を変更するたびに `session.SetAttribute()` を呼び出して、セッション属性の変更を Borland Web コンテナに通知します。

これは、サーブレットの開発では共通のオペレーションで、このオペレーションを実行する場合、`server.xml` ファイルを変更する必要はありません。セッションデータを変更するたびに、データは JSS を介してすぐにデータベースに書き込まれます。Web コンテナのインスタンスに予定外のシャットダウンが発生しても、セッションを収集するように割り当てられた次の Web コンテナのインスタンスがセッションデータにアクセスします。原則として、プログラム的インプリメンテーションでは、変更箇所がすみやかに保存されます。

自動的インプリメンテーション

*自動的インプリメンテーション*の場合、データが変更されたかどうかに関係なく、セッションデータは定期的に JSS に保存されます。自動的インプリメンテーションでは、セッション属性が変更されたことを Web コンポーネントに通知する必要はありません。

たとえば、次のサンプルコードのように、`setAttribute ()` を呼び出さなくても状態を変更できます。

```
Object myState = session.getAttribute("myState");

// Modify mystate here and do not call setAttribute ()
```

設定ファイル `server.xml` のコードは、次のようになります。

```
<Manager className=
"org.apache.catalina.session.PersistentManager"
         maxIdleBackup="xxx">
<Store className=
"org.apache.catalina.session.BorlandStore"
storeName="jss_factory">
</Manager>
```

ここで、`xxx` はセッションデータが保存される間隔を秒単位で示します。

`server.xml` ファイルの詳細については、35 ページの「Borland Web コンテナの IIOP 設定の変更」を参照してください。

メモ 自動的インプリメンテーションを使用する場合は、次の制限事項を考慮する必要があります。

1. Web コンテナが保存イベントと保存イベントの間に停止してしまうと、次の Web コンテナのインスタンスには最新の変更内容が伝えられません。ハートビートの時間間隔を定義するときは、この点に注意してください。
2. データが変更されたかどうかに関係なく、指定した間隔でデータは保存されます。セッションの変更頻度が低い場合に間隔値が短いと無駄になります。

HTTP セッションの使い方

HTTP（HyperText Transfer Protocol）はステートレスプロトコルです。クライアント／サーバー方式では、Apache Web サーバーが受け取るすべてのクライアント要求は、独立したトランザクションとして処理されます。クライアント要求の間には特に関係はありません。これは、クライアントおよびサーバーにある典型的なステートレス接続です。

ただし、クライアントが完全なトランザクション処理を行うためには、セッションの概念が必要な場合があります。*セッションという概念*は、通常クライアントとサーバー間にステートフルな対話を行うことを意味します。セッション概念のサンプルは、対話型のショッピングカートを使用したオンラインショッピングです。ショッピングカートに新しいアイテムを追加するたびに、前に追加したアイテムリストに新しいアイテムが追加され、表示されることが期待されます。HTTP は、通常ステートフルな方法でクライアント要求を処理しませんが、できないわけではありません。

AppServer は、インプリメンテーションの次の 2 つの方法を使って HTTP セッションをサポートします。

- Cookie：Web サーバーは Cookie を送信して、セッションを識別します。Web ブラウザは、その後の要求でも、同じ Cookie の送信を継続します。この Cookie は、サーバー側のコンポーネントが指定セッションのトランザクションを処理する方法を判定するときに便利です。
- URL の変更：ユーザーがクリックする URL は、セッション情報を保持するように動的に書き換えられます。

第 8 章

Apache Web サーバーから CORBA サーバーへの接続

Apache IIOP コネクタを設定することで、Web サーバーは `ReqProcessor` インターフェース定義言語（IDL）を実装するすべてのスタンドアロンの CORBA サーバーと通信できるようになります。これは、ほとんどすべての CORBA サーバーで Web ベースのフロントエンドを簡単に導入できることを意味します。

詳細については、Borland AppServer の『*インストールガイド*』の「Borland AppServer の Windows へのインストール」または「Borland AppServer の Solaris または Linux へのインストール」のセクションを参照してください。

Web 対応の CORBA サーバー

インターネットを介して CORBA サーバーにアクセスできるようにするには、次の手順にしたがいます。

- CORBA メソッドの URL の指定
- CORBA サーバーにおける ReqProcessor IDL の実装

CORBA メソッドの URL の指定

インターネットを介して CORBA サーバーにアクセスできるようにするには、次の手順にしたがいます。

1 公開するビジネスオペレーションを決定します。

2 ビジネスオペレーションの URL を指定します（CORBA メソッド）。

たとえば、銀行の CORBA サーバーは、メソッド `debit()`、`credit()`、`balance()` を実装し、これらのビジネスメソッドをインターネット経由でユーザーに公開します。CORBA サーバーの各オペレーションをユーザーがブラウザで入力する内容にマッピングします。

銀行の Web サイトは、`http://www.bank.com` です。

インターネットユーザーに公開する各ビジネスオペレーションに URL を指定するには、次のように操作します。

1 会社のルート URL に Web アプリケーション名を追加します。

次に例を示します。

```
http://www.bank.com/accounts
```

ここで、`accounts` は Web アプリケーション名です。

重要 デフォルトでは、Web サーバーからは Web アプリケーションを使用できません。Web アプリケーションを Apache から使用できるようにするには、Web アプリケーションデスクリプタにいくつかの情報を追加する必要があります。追加方法の具体的な手順については、『*管理コンソールユーザーズガイド*』の「デプロイメントデスクリプタエディタの使い方」の「Web デプロイメントパス」を参照してください。

2 公開する Web アプリケーションのメソッドに、ユーザーにわかりやすい名前を追加します。

次に例を示します。

```
http://www.bank.com/accounts/balance
```

ここで、`balance` は、`balance()` メソッドを表す名前です。

CORBA サーバーにおける ReqProcessor IDL の実装

`ReqProcessor` IDL によって、IIOP を使用した Web サーバーと CORBA サーバー間の通信が可能です。`ReqProcessor` IDL を CORBA サーバーに実装すると、Web サーバーから CORBA サーバーに http 要求を転送できます。

この IDL を実装するにあたり、要求 URL を `HttpRequest` の一部とみなし、その URL に呼応して適切な CORBA メソッドを呼び出す必要があります。

ReqProcessor インターフェースの IDL 仕様

```
*/
module apache {
        struct NameValue {
                string name;
                string value;
    };
        typedef sequence<NameValue> NVList;
    typedef sequence<octet> OctetSequence_t;

    struct HttpRequest {
                string  authType; // 認証タイプ（BASIC、FORM など）
                string  userid; // 要求に関連付けられているユーザー名
                string  appName; // アプリケーション名（コンテキストパス）
                string  httpMethod; // PUT、GET など
                string  httpProtocol; // プロトコル HTTP/1.0、HTTP/1.1 など
                string  uri; // 要求に関連付けられている URI
                string  args; // この要求に関連付けられているクエリー文字列
                string  postData; // 要求に関連付けられている POST（form）データ
                boolean isSecure; // クライアントで https または http が指定されているか
                string  serverHostname; // URI で指定されているサーバーのホスト名
                string  serverAddr; //  （オプション）URI で指定されているサーバーの IP アドレス
                long    serverPort; // URI で指定されているサーバーのポート番号
                NVList headers; // この要求形式に関連付けられているヘッダー（ヘッダー名：値）
        };

        struct HttpResponse {
                long    status; // HTTP ステータス（OK など）
                boolean isCommit; // サーバーがこの要求をコミットするかどうか
                NVList  headers; // ヘッダー配列
                OctetSequence_t data; // データバッファ
        };

interface ReqProcessor {
        HttpResponse process(in HttpRequest req);
  };
};
```

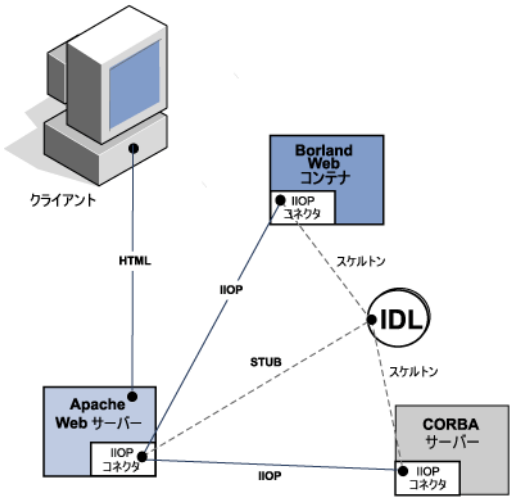
process() メソッド

ReqProcessor IDL には、process() メソッドが含まれています。これは、インターネット要求があると、Apache Web サーバーが呼び出すメソッドです。Web サーバーはユーザーの要求を引数で process() メソッドに渡します。基本的に、process() メソッドへの入力は、ブラウザからの要求 HttpRequest です。process() メソッドからの出力は、HttpResponse に格納された html ページです。

CORBA サーバーを呼び出すための Apache Web サーバーの設定

Apache Web サーバーが CORBA サーバーを呼び出すには、その前に httpd.conf ファイル内の IIOP コネクタに関するコード行を変更する必要があります。詳細については、35 ページの「Borland Web コンテナの IIOP 設定の変更」を参照してください。

図 8.1　Apache から CORBA サーバーへの接続



Apache IIOP 設定

Apache IIOP コネクタには、Web サーバーのクラスタ情報で更新する必要がある設定ファイルのセットがあります。これらの IIOP コネクタ設定ファイルは、デフォルトでは次のディレクトリに置かれています。

```
<install_dir>/var/domains/<domain_name>/configurations/
<configuration_name>/mos/<apache_managedobject_name>/conf
```

メモ　「クラスタ」は、システムが 1 つの名前や URI で認識する CORBA オブジェクトインスタンスを表します。IIOP コネクタでは、複数のインスタンス間で負荷分散できるので、用語「クラスタ」を使用します。

次の 2 つの設定ファイルがあります。

IIOP 設定ファイル	説明
WebClusters.properties	クラスタと、各クラスタに対応する CORBA サーバーを指定します。
UriMapFile.properties	WebClusters.properties ファイルで定義したクラスタに URI リファレンスをマッピングします。

この 2 つの設定ファイルを変更する場合は、Apache Web サーバーまたは CORBA サーバーを起動やシャットダウンは不要です。これは、IIOP コネクタによってファイルが自動的にロードされるためです。

新しい CORBA サーバー（クラスタ）の追加

CORBA サーバーは、IIOP コネクタにとって「クラスタ」です。CORBA サーバーを IIOP コネクタとともに使用するように設定するには、クラスタを定義して `WebClusters.properties` ファイルに追加します。

`WebClusters.properties` ファイルは、IIOP コネクタに次の情報を伝えます。

- 利用できる各クラスタの名前：（`ClusterList`）
- Web コンテナの ID
- 特定クラスタに自動負荷分散（`enable_loadbalancing`）を提供するかどうか

新しいクラスタを追加するには、次の手順にしたがいます。

- `WebClusters.properties` ファイルで：

  a `ClusterList` に設定したクラスタの名前を追加します。次に例を示します。

  ```
  ClusterList=cluster1,cluster2,cluster3
  ```

  b 次の形式でクラスタ名、必須の `webcontainer_id` 属性、および追加属性（「クラスタ定義属性」の表を参照）を指定する行を追加して、各クラスタを定義します。次に例を示します。

  ```
  <clustername>.webcontainer_id = <id> <attribute>
  ```

メモ フェイルオーバーとスマートセッションは常に有効になっています。詳細については、55 ページの「Web コンポーネントのクラスタリング」を参照してください。

属性	必須	定義
webcontainer_id	はい	オブジェクトの「バインド」名、またはクラスタを実装する Web コンテナを識別する corbaloc 文字列。
enable_loadbalancing	いいえ	負荷分散は、デフォルトで有効です。負荷分散を有効にするには、この属性を省略するか、この属性を true に設定します。負荷分散を無効にするには、false に設定して、このクラスタインスタンスが負荷分散技術を使用しないように指定します。 **警告：** enable_loadbalancing 属性の指定時には、正しい値（true か false）を指定してください。

次に例を示します。

```
ClusterList=cluster1,cluster2,cluster3
cluster1.webcontainer_id = tc_inst1
cluster2.webcontainer_id = corbaloc::127.20.20.2:20202,:127.20.20.3:20202/
 tc_inst2
cluster2.enable_loadbalancing = true
cluster3.webcontainer_id = tc_inst3
cluster3.enable_loadbalancing = false
```

上の例では、次の 3 つのクラスタが定義されています。

1. 最初のクラスタは、osagent 命名方式を使用し、負荷分散が有効にされています。
2. 2 番めのクラスタは、corbaloc 命名方式を使用し、負荷分散が有効にされています。
3. 3 番めのクラスタは、osagent 命名方式を使用しますが、負荷分散が無効にされています。

メモ 特定のクラスタの使用を無効にするには、目的のクラスタ名を `ClusterList` リストから削除します。ただし、CORBA サーバー（アタッチユーザー）にアタッチされたアクティブ HTTP セッションは削除しないでください。これらの「ライブ」セッションの要求が失敗します。

メモ `WebClusters.properties` ファイルの変更結果は、次回の要求で自動的に有効になります。サーバーを再起動する必要はありません。

定義済みクラスタへの URI のマッピング

クラスタエントリを定義すると、後は、Web サーバーが受け取る HTTP 要求のどれを CORBA サーバーに転送するか指定するだけです。`UriMapFile.properties` ファイルを使用して、`WebClusters.properties` ファイルで設定されている Web クラスタ名（CORBA インスタンス）に http URI 文字列をマッピングします。

- `UriMapFile.properties` ファイルで次のように入力します。

  ```
  <uri-mapping> = <clustername>
  ```

  ここで、`<uri-mapping>` は、標準 URI 文字列またはワイルドカード文字列です。`<clustername>` は、`WebClusters.properties` ファイルの `ClusterList` エントリに出現するクラスタ名です。

次に例を示します。

```
/examples = cluster1
/examples/* = cluster1

/petstore/index.jsp = cluster2
/petstore/servlet/* = cluster2
```

この例では、次のようになります。

- `/examples` で開始する URI は、CORBA サーバーや「cluster1」Web クラスタで実行する CORBA オブジェクトに転送されます。
- `/petstore/index.jsp` と同じ URI、または `/petstore/servlet` で始まる URI は、「cluster2」に転送されます。

メモ URI マッピングの場合、ワイルドカード "*" は、URI の**最後**のワードにだけ有効であり、次のような場合が考えられます。

- ワード全体（および下位のすべてのリファレンス）。例：`/examples/*`。
- ファイル指定文字列のファイル名の部分。例：`/examples/*.jsp`。

メモ `UriMapFile.properties` ファイルの変更結果は、次回の要求で自動的に有効になります。サーバーを再起動する必要はありません。

`WebCluster.properties` または `UriMapFile.properties` が変更された場合、それらのファイルは IIOP コネクタによって自動的にロードされます。したがって、これらのファイルを変更しても、Web サーバーまたは CORBA サーバーを起動したりシャットダウンする必要はありません。

第 9 章

Borland AppServer Web サービス

Borland AppServer（AppServer）は、すべての Borland パーティションでそのまま利用できる Web サービス機能を提供します。

Web サービスの概要

「Web サービス」は、標準 XML メッセージ通信でネットワーク上の記述、公開、検索、呼び出しができるアプリケーションコンポーネントです。Simple Object Access Protocol（SOAP）、Web Services Description Language（WSDL）、Universal Discovery、Description and Integration（UDDI）などの新しいテクノロジで定義した Web サービスは、World Wide Web でアクセスして再利用できるソフトウェアモジュールから e ビジネスアプリケーションを作成するための新しいモデルです。

Web サービスアーキテクチャ

標準 Web サービスアーキテクチャは、Web サービスの公開、検索、バインドを行う 3 つのロールからなります。

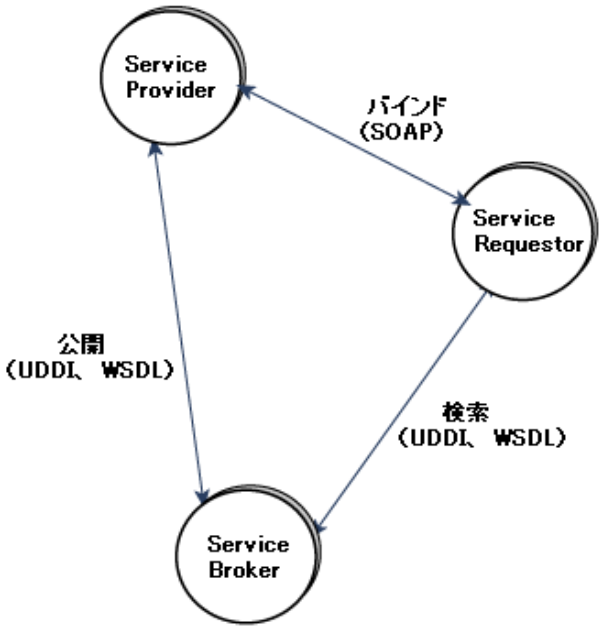
- 「*Service Provider*」は、利用できるすべての Web サービスを Service Broker に登録します。
- 「*Service Broker*」は、Service Requestor のアクセス用に Web サービスを公開します。公開される情報の内容は、Web サービスとその場所です。
- 「*Service Requestor*」は、Service Broker との対話から Web サービスを検索します。その結果を受けて、Service Requestor は、Web サービスをバインドまたは呼び出します。

Service Provider は Web サービスを処理し、Web 経由でクライアントに提供します。Service Provider は、Web サービス定義とバインド情報を、Universal Description、Discovery、Integration（UDDI）レジストリに公開します。Web Service Description Language（WSDL）ドキュメントには、受信メッセージと返信用の応答メッセージなど、Web サービスに関する情報が収められます。

Service Requestor は、Web サービスを利用するクライアントプログラムです。Service Requestor は、UDDI や電子メールなどの方法で Web サービスを検索します。その後、Web サービスをバインドして呼び出します。

Service Broker は、Service Provider と Service Requestor 間の対話を管理します。Service Broker では、すべてのサービス定義とバインド情報を提供します。現在は、SOAP（分散環境の情報通信向けの XML ベースのメッセージ通信、エンコードプロトコル形式）が Service Requestor と Service Broker 間の通信標準となっています。

図 9.1　標準 Web サービスアーキテクチャ



Web サービスとパーティション

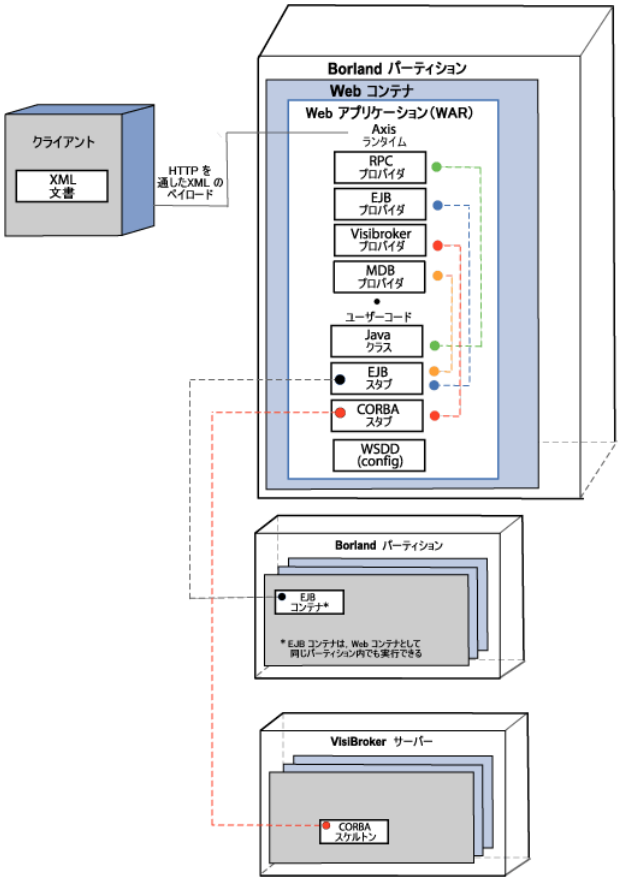
AppServer のパーティションは、いずれも Web サービスをサポートするように設定されています。必要な操作は、パーティションを起動し、Web サービスを収めた WAR（または WAR を収めた EAR）をデプロイメントするだけです。

また、以前にデプロイメントされたステートレスセッション Bean を Web サービスとして公開できます。詳細については、『*管理コンソールユーザーズガイド*』の「EJB を Web サービスとしてエキスポートウィザード」を参照してください。

Borland Web サービスは、Apache Axis テクノロジを基本に、受信 SOAP Web サービス要求を次の「Web サービスプロバイダ」に配信します。

- EJB プロバイダ
- RPC/Java プロバイダ
- RPC/Java プロバイダ

図 9.2　Borland Web サービスアーキテクチャ



Web サービスプロバイダ

Borland Web サービスエンジンには、多くのプロバイダが組み込まれています。「プロバイダ」は、クライアント Web サービス要求を、サーバー側のユーザークラスに接続するリンクです。

プロバイダにはいずれも以下のような機能があります。

- メソッドを呼び出せるオブジェクトのインスタンスを作成します。このオブジェクトの作成方法は、厳密にはプロバイダによって異なります。
- このオブジェクトでメソッドを呼び出し、XML クライアントが送信したすべてのパラメータを渡します。
- Axis Runtime エンジンに戻り値を渡します。XML に変換されてクライアントに返ります。

deploy.wsdd ファイルにおける Web サービス情報の指定

新しい Web サービスをインストールするときは、Web サービスの名前を指定し、サービスで利用するプロバイダを指定します。プロバイダによってパラメータが異なります。次の項では、各サービスプロバイダと、それぞれに必要なパラメータについて説明します。

Java:RPC プロバイダ

このプロバイダでは、Web サービスを処理するクラスがアプリケーションアーカイブ（WAR）にあるものとします。Web サービス要求を受信すると、RPC プロバイダは次の処理を行います。

1 サービスに関連付けられた Java クラスをロードします。

2 オブジェクトの新しいインスタンスを作成します。

3 reflection によって指定メソッドを呼び出します。

パラメータは次のとおりです。

- `className`：このサービスで要求を受信したときにロードされるクラスの名前。
- `allowedMethods`：このクラスで呼び出せるメソッド。このクラスが利用できるメソッドは、ここに示すメソッドだけにとどまりません。ここに紹介したメソッドはリモート呼び出しに適用できるものです。

例を次に示します。

```
<service name="Animal" provider="java:RPC">
 <parameter name="className" value="com.borland.examples.web
services.java.Animal"/>
 <parameter name="allowedMethods" value="talk sleep"/>
</service>
```

Java:EJB プロバイダ

このプロバイダで Web サービスを処理するクラスは EJB です。

メモ 以前にデプロイメントされたステートレスセッション Bean を Web サービスとして公開できます。詳細については、『*管理コンソールユーザーズガイド*』の「EJB を Web サービスとしてエキスポートウィザード」を参照してください。

Web サービス要求を受信すると次の処理が行われます。

1 EJB プロバイダは、JNDI 初期コンテキストで Bean 名を検索します。

2 ホームクラスを検索して Bean を作成します。

3 EJB スタブで reflection によって指定メソッドを呼び出します。

クライアントをアクセスするには、実 EJB 自体をパーティションのいずれかにデプロイメントします。

主なパラメータは次のとおりです。

- `beanJndiName`：JNDI の Bean の名前。
- `homeInterfaceName`：絶対パスで指定したホームインターフェースのクラス。このクラスの場所は WAR とします。
- `className`：EJB リモートインターフェースの名前。
- `allowedMethods`：この EJB で呼び出せるメソッド。スペースで区切ります。EJB が利用できるメソッドは、ここに示すメソッドだけにとどまりません。ここに紹介したメソッドはリモート呼び出しに適用できるものです。

例を次に示します。

```
  <service name="Animal" provider="java:EJB">
    <parameter name="beanJndiName" value="Animal"/>
    <parameter name="homeInterfaceName"
value="com.borland.examples.webservices.ejb.AnimalHome"/>
    <parameter name="className"
value="com.borland.examples.webservices.ejb.Animal"/>
    <parameter name="allowedMethods" value="talk sleep"/>
  </service>
```

Borland Web サービスのはたらき

1. Web サービスサーバーは、クライアントから XML SOAP メッセージを受信します。
2. その後、次の処理を行います。
   a. SOAP メッセージを解釈します。
   b. SOAP サービス名を展開します。
   c. サービスに応答できるプロバイダを決定します。
3. SOAP サービスとプロバイダのタイプ間のマッピングは、WAR デプロイメントの一部として Web サービスデプロイメントデスクリプタ（WSDD）から取得します。
4. メッセージは目的のプロバイダに転送されます。各プロバイダによるメッセージのさまざまな操作方法の詳細については、次の箇所を参照してください。70 ページの「Java:RPC プロバイダ」と 70 ページの「Java:EJB プロバイダ」

Web サービスのデプロイメント

Web サービスは、WAR の一部としてデプロイメントされます。1 つの WAR で複数の Web サービスを格納できます。また、複数の Web サービスを格納した WAR を複数デプロイメントすることもできます。

通常の WAR と、Web サービスを格納した WAR との違いは、`WEB-INF` ディレクトリに `server-config.wsdd` という名前のデスクリプタがある点です。`server-config.wsdd` ファイルには設定情報（Web サービス名、プロバイダ、対応する Java クラス）があります。

WAR 1 つにつき WSDD は 1 つあり、WAR で利用できるすべての Web サービスに関する情報が保存されています。

Web サービスを持つ WAR の代表的なコンポーネント構造には次の要素があります。

- `WEB-INF/web.xml`
- `WEB-INF/server-config.wsdd`
- `WEB-INF/classes/<` 各自の `Web` サービスに対応するクラスがここに配置されます。`>`
- `WEB-INF/lib/<` 各自の `Web` サービスに対応するクラスが圧縮された `JAR` 形式でここに配置されます。`>`

`WEB-INF/lib` には、Axis Runtime エンジンに必要な標準 JAR も組み込まれています。

Web サービスとして Java クラスを公開するには、パーティションにデプロイメントする項目を WSDD 形式で定義してください。たとえば「BankService」というサービスに対応するエントリは、次のようになります。

```
<service name="BankService" provider="java:RPC">
   <parameter name="allowedMethods" value="create_account query_account"/>
   <parameter name="className" value="com.fidelity.Bank"/>
</service>
```

この場合、`com.fidelity.Bank` Java クラスは Web サービス `BankService` にリンクします。クラス `com.fidelity.Bank` には、多くの public メソッドを設定できますが、Web サービスで利用できるのは、メソッド `create_account` とメソッド `query_account` だけです。

server-config.wsdd ファイルの作成

server-config.wsdd を作成するには

- JBuilder を利用して WAR の一部としてデプロイメントデスクリプタを生成します。

または

1. テキストエディタで deploy.wsdd ファイルを作成します。<install_dir>/examples/webservices/java/server にある deploy.wsdd ファイルを参照してください。
2. deploy.wsdd ファイルとともに 74 ページの「ツールの概要」を実行します。

   ```
   prompt>java org.apache.axis.utils.Admin server deploy.wsdd
   ```

   server-config.wsdd ファイルは、Web の一部としてパッケージされています。

WSDD プロパティの表示と編集

WAR ファイルにパッケージされている Web サービスデプロイメントデスクリプタ（WSDD）のプロパティ（server-config.wsdd ファイル）を表示および編集するには、Borland 管理コンソール または DDEditor を使用します。詳細については、『*管理コンソールユーザーズガイド*』の「J2EE コンポーネントの設定の表示」の「Web サービスデプロイメントデスクリプタプロパティの表示」、または「デプロイメントデスクリプタエディタの使い方」の「Web サービス」を参照してください。

Web Service アプリケーションアーカイブのパッケージ

Web サービスアーカイブにデプロイメントできる WAR ファイルを作成するには、次のように操作します。

1. Web サービスクラスが WEB-INF/classes または WEB-INF/lib にあることを確認します。
2. WEB-INF/lib に Axis ツールキットをコピーします。Axis ライブラリは、次のサイトから入手できます。<install_dir>/lib/axis
3. Axis ツールキットに必要な web.xml を WEB-INF ディレクトリにコピーします。web.xml は、次の場所にあります。<install_dir>/etc/axis
4. Web サービスに関するデプロイメント情報がある deploy.wsdd を作成します。
5. この deploy.wsdd で Axis 管理ツールを実行し、次のように server-config.wsdd を生成します。

   ```
   java org.apache.axis.utils.Admin server deploy.wsdd
   ```

6. この server-config.wsdd を WEB-INF にコピーします。
7. Web アプリケーションを WAR ファイルに JAR します。

Borland Web サービスのサンプル

Web サービスの開発とデプロイメント入門用に、Borland Web サービスエンジンのサンプルを用意しました。サンプルは、AppServer インストールの次の場所に収められています。

```
<install_dir>/examples/webservices/
```

さまざまな Web サービスプロバイダのサンプルが、Java、EJB、MDB、VisiBroker の各フォルダの Web サービスサンプルディレクトリにあります。

AppServer インストールの次の場所には、Apache Axis サンプルもあります。

```
<install_dir>/examples/webservices/axis/samples/
```

Web サービスプロバイダのサンプルの使用

AppServer のサンプルを実行するには、構築してからデプロイメントします。サンプルの構築では、必要な WSDL ファイルを生成し、アプリケーションのコードとデスクリプタをデプロイメント単位にパッケージします。この場合のデプロイメント単位は、WAR です。これで、WAR を Borland パーティションにデプロイメントできます。アプリケーションを実行するには、コマンドラインからクライアントを呼び出します。サンプルの構築と実行は、Apache ANT ユーティリティーで自動的に処理できます。ただし、デプロイメントには、AppServer に添付されているツールを使用します。

サンプルの構築、デプロイメント、実行手順

1 **構築。**サンプルの構築は、すべてまとめて同時に、または個別に行うことができます。サンプルをすべてまとめて同時に構築するには、次のディレクトリに移動して Ant コマンドを実行します。

```
/examples/webservices/
```

次に例を示します。

```
C:/BDP/examples/webservices>Ant
```

以上のコマンドを実行すると、すべてのサンプルが構築されます。

個別に構築するには、目的のディレクトリに移動し、Ant コマンドを実行します。

次に例を示します。

```
C:/BDP/examples/webservices/java>Ant
```

以上のコマンドを実行すると、Java Provider サンプルだけが構築されます。

2 **デプロイメント。**サンプルをデプロイメントして、AppServer の実行インスタンスにします。WAR と JAR をデプロイメントするには、`ant deploy` ターゲット、または次のいずれかを使用します。

- iastool コマンドラインユーティリティ。詳細については、315 ページの「iastool コマンドライン ユーティリティ」を参照してください。
- デプロイメントウィザード。詳細については、『*管理コンソールユーザーズガイド*』の「デプロイメントウィザード」を参照してください。

3 **実行。**サンプルを実行するには、そのディレクトリに移動し、`ant run-client` コマンドを実行します。

たとえば、Java Provider クライアントを実行するには、次のコマンドを実行します。

```
C:/BDP/examples/webservices/java>Ant run-client
```

Apache Axis Web サービスのサンプル

Apache Axis Web サービスのサンプルは、Borland パーティションにある `axis-samples.war` ファイルにデプロイメント済みです。以上のサンプルは、デプロイメント済みであり、『Apache Axis User's Guide』に記載された Apache Axis デプロイメントコマンドを実行する必要はありません。

『Apache Axis User's Guide』は、AppServer インストールの次の場所にあります。

```
<install_dir>/doc/axis/user-guide.html
```

または、*サードパーティドキュメント*の「Axis Documentation」にあります。

以上のサンプルでは、Axis の機能を紹介しています。元の Apache Axis インプリメンテーションからそのまま流用しているので、実行するかどうかは保証の限りではありません。

ツールの概要

ここでは、サンプルの各種ツールについて説明します。

Apache ANT ツール

Apache ANT ユーティリティは、プラットフォームに依存しない java ベースのビルドツールであり、サンプルの構築に使用します。

XML ビルドスクリプト `build.xml` でツールを実行します。`build.xml` ファイルは、プロジェクトに使用できるさまざまなターゲットと、それらのターゲットに呼応して実行されるコマンドを記述します。AppServer は、Apache Ant ツールを実行するための JAR とスクリプトを提供します。

Java2WSDL ツール

Java2WSDL は、Java クラスに対応した WSDL を生成する Apache Axis ユーティリティクラスです。このクラスでは、さまざまなコマンドライン引数を受け取ります。このユーティリティを引数なしで、次のように実行すると、対応するすべてのヘルプ情報が得られます。

```
java org.apache.axis.wsdl.Java2WSDL
```

メモ 次のコマンドを実行する前に、jar ファイルがすべて `<install-dir>¥lib¥axis` ディレクトリに組み込まれるよう CLASSPATH を設定してください。

WSDL2Java ツール

CLASSPATH は、WSDL ファイルから Java クラスを生成する Apache Axis ユーティリティクラスです。このツールでは、（クライアント側で使用する）Java スタブ、または（サーバー側で使用する）Java スケルトンを生成できます。生成されるファイルにより、所定の WSDL 用のクライアントやサーバーを簡単に開発できます。

このクラスでは、さまざまなコマンドライン引数を受け取ります。このユーティリティを引数なしで、次のように実行すると、対応するすべてのヘルプ情報が得られます。

```
java org.apache.axis.wsdl.WSDL2Java
```

メモ 次のコマンドを実行する前に、jar ファイルがすべて `<install-dir>¥lib¥axis` ディレクトリに組み込まれるよう CLASSPATH を設定してください。

Axis Admin ツール

Apache Admin ツールは、一部の Web サービス固有のデプロイメント情報から WAR レベルグローバル設定ファイルを生成するユーティリティクラスです。

このユーティリティの入力は、1 つ以上の Web サービスに関するデプロイメント情報を収めた XML ファイル（通常は `deploy.wsdd`）です。Apache Admin ユーティリティは、必要なグローバル定義を追加し、出力ファイルを書き込みます。このツールは、次のように使用します。

```
java org.apache.axis.utils.Admin server|client deployment-file
```

メモ 次のコマンドを実行する前に、`<install-dir>¥lib¥axis` ディレクトリの jar ファイルがすべて組み込まれるよう CLASSPATH を設定してください。

このツールは、選択したオプションに基づいて `server-config.wsdd` または `client-config.wsdd` を生成します。

第 10 章

エンタープライズ Bean クライアントの作成

エンタープライズ Bean のクライアントビュー

エンタープライズ Bean のクライアントは、アプリケーションか別のエンタープライズ Bean です。アプリケーションの場合、スタンドアロンアプリケーション、アプリケーションクライアントコンテナ、サーブレット、またはアプレットがあります。どのような場合でも、エンタープライズ Bean のクライアントがエンタープライズ Bean を使用するには、次の処理を行う必要があります。

- Bean のホームインターフェースを検索します。EJB 仕様でクライアントからホームインターフェースをアクセスする場合、JNDI（Java Naming and Directory Interface）API を使用します。
- エンタープライズ Bean オブジェクトのリモートインターフェースへのリファレンスを取得します。この作業の一環として、Bean のホームインターフェースで定義されているメソッドを使用します。セッション Bean を作成します。または、エンティティ Bean を作成するか、検索します。
- エンタープライズ Bean で定義された 1 つ以上のメソッドを呼び出します。エンタープライズ Bean で定義されたメソッドをクライアントから直接呼び出すことはありません。かわりにクライアントは、エンタープライズ Bean オブジェクトのリモートインターフェースのメソッドを呼び出します。リモートインターフェースには、エンタープライズ Bean がクライアントに公開するメソッドが定義されています。

クライアントの初期化

SortClient アプリケーションは、必要な JNDI クラスと、SortBean のホームインターフェースとリモートインターフェースをインポートします。クライアントは JNDI API でエンタープライズ Bean のホームインターフェースを検索します。

クライアントアプリケーションからは、データベース接続、リモートエンタープライズ Bean、環境変数などのリソースに、（各種 J2EE 仕様で推奨されているように）論理名でアクセスすることもできます。コンテナは、J2EE 仕様にしたがい、ローカルの JNDI 名前空間（`java:comp/env`）の管理対象のオブジェクトとしてこれらのリソースをエクスポーズします。

ホームインターフェースの検索

次のサンプルコードに示すように、クライアントは JNDI で Bean のホームインターフェースを検索します。クライアントは最初に JNDI 初期ネーミングコンテキストを取得します。このコードでは、新しい `javax.naming.Context` オブジェクトをインスタンス化します。このサンプルでは、initialContext を呼び出すオブジェクトです。次にクライアントは、コンテキストの `lookup()` メソッドにより、名前からホームインターフェースを識別します。初期ネーミングコンテキストファクトリの初期化方法は、EJB コンテナおよびサーバーによって異なるので注意してください。

クライアントアプリケーションからは、論理名でホームインターフェースなどのリソースにアクセスすることもできます。詳細については、75 ページの「クライアントの初期化」を参照してください。

コンテキストの `lookup()` メソッドは、`java.lang.Object` タイプのオブジェクトを返します。作成するコードでは、返されたオブジェクトを要求される型にキャストします。次のサンプルコードは、sort サンプルのクライアントコードの一部を示します。`main()` ルーチンでは、まず、JNDI ネーミングサービスとそのコンテキストの lookup メソッドを使用して、ホームインターフェースを検索します。リモートインターフェースの名前を（この例では `sort`）を `context.lookup()` メソッドに渡します。このサンプルプログラムは、最終的には `context.lookup()` メソッドの結果をホームインターフェース型 SortHome にキャストします。

```
// SortClient.java
import javax.naming.InitialContext;
import SortHome; // Bean のホームインターフェースを検索
import Sort; // Bean のリモートインターフェースを検索
public class SortClient {
   ...
   public static void main(String[] args) throws Exception {
      javax.naming.Context context;

// JNDI コンテキストルックアップの優先
// ローカル JNDI コンテキスト内の論理的な JNDI 名 (ejb-ref など) を使用した JNDI コンテキストの取得
      javax.naming.Context context = new javax.naming.InitialContext();
      Object ref = context.lookup("java:comp/env/ejb/Sort");
      SortHome home = (SortHome) javax.rmi.PortableRemoteObject.narrow
               (ref, SortHome.class);
      Sort sort = home.create();
      ... // ソートとマージ作業の実行
      sort.remove();
   }
}
```

このクライアントプログラムの `main()` ルーチンは、一般的な例外を生成します。今回のようなコーディングでは、発生する可能性のある例外を SortClient プログラムでキャッチする必要はありませんが、例外が発生した場合はプログラムが終了します。

リモートインターフェースの取得

エンタープライズ Bean のホームインターフェースを取得したら、そのエンタープライズ Bean のリモートインターフェースまでのリファレンスを取得できます。そこで、ホームインターフェースの作成メソッドまたは検索メソッドを使用します。どのメソッドを呼び出すかは、エンタープライズ Bean の種類と、エンタープライズ Bean プロバイダがホームインターフェースに定義したメソッドによって決まります。

たとえば、最初のサンプルコードには、Sort リモートインターフェースまでのリファレンスを SortClient で取得する方法が示されています。SortClient は、ホームインターフェースへのリファレンスを取得し、それを正しい型（SortHome）にキャストし、Bean のインスタンスを作成して、そのメソッドを呼び出します。SortClient は、ホームインターフェースの `create()` メソッドを呼び出します。呼び出されたメソッドは、Bean のリモートインターフェース Sort までのリファレンスを受け取ります。SortBean はステートレスセッション Bean なので、そのホームインターフェースに `create()` メソッドは 1 つしかなく、当然のことながらこのメソッドはパラメータを受け取りません。次に SortClient は、リモートインターフェースで定義されている `sort()` メソッドと `merge()` メソッドを呼び出して、ソートを実行します。ソートが終了すると、リモートインターフェースの `remove()` メソッドを呼び出して、このエンタープライズ Bean インスタンスを削除します。

セッション Bean

セッション Bean クライアントがセッション Bean のリモートインターフェースへのリファレンスを取得するには、ホームインターフェースのいずれかの作成メソッドを呼び出します。

すべてのセッション Bean には、少なくとも 1 つの `create()` メソッドがあります。ステートレスセッション Bean には `create()` メソッドを 1 つだけ定義する必要があり、このメソッドは引数を受け取りません。ステートフルセッション Bean には、`create()` メソッドを 1 つ定義できます。またさまざまなパラメータを持つ補助的 `create()` メソッドを定義できます。`create()` メソッドにパラメータがある場合は、それらのパラメータの値で、セッション Bean の初期化が行われます。

デフォルトの `create()` メソッドにパラメータはありません。sort サンプルでは、ステートレスセッション Bean が使用されています。当然ながら、ステートレスセッション Bean には、パラメータを受け取らない `create()` メソッドが 1 つだけあります。

```
Sort sort = home.create();
```

これに対して、cart サンプルでは、ステートフルセッション Bean を使用し、ホームインターフェース CartHome が複数の `create()` メソッドを実装しています。1 つの `create()` メソッドは 3 つのパラメータを受け取り、それらのパラメータからカート内容の購入者を識別して、Cart リモートインターフェースへのリファレンスを返します。CartClient は、3 つのパラメータ（`cardHolderName`、`creditCardNumber`、`expirationDate`）の値を設定し、`create()` メソッドを呼び出します。これを次のサンプルコードで示します。

```
Cart cart;
   {
      String cardHolderName = "Jack B. Quick";
      String creditCardNumber = "1234-5678-9012-3456";
      Date expirationDate = new GregorianCalendar(2001,
        Calendar.JULY, 1).getTime();
      cart = home.create(cardHolderName, creditCardNumber, expirationDate);
   }
```

セッション Bean に検索メソッドはありません。

エンティティ Bean

クライアントは、検索または作成オペレーションにより、エンティティオブジェクトへのリファレンスを取得します。ここで、エンティティオブジェクトはデータベースに保存された基底のデータを表すということを思い出してください。エンティティ Bean は永続的データを表すため、エンティティ Bean は長い期間存続します。エンティティ Bean を呼び出すクライアントアプリケーションの存続期間より大幅に長いのは間違いありません。したがって、クライアントでは、新しいエンティティオブジェクトを作成してデータを新規作成して基底のデータベースに格納するより、まずは目的の永続的データを表す既存のエンティティ Bean がないか探すのが普通です。

クライアントは、検索オペレーションにより、リレーショナルデータベーステーブル内の特定の 1 行などの既存のエンティティオブジェクトを検索します。つまり、検索オペレーションでは、データストレージの既存データエンティティを検索します。データには、エンティティ Bean によってデータストアに追加されるデータと、データベース管理システム（DBMS）などの EJB コンテキスト範囲外から直接追加されるデータがあります。レガシーシステムの場合、EJB コンテナをインストールする前からデータが存在する場合もあります。

クライアントは、エンティティ Bean オブジェクトの `create()` メソッドを使用して、基底のデータベースに格納される新しいデータエンティティを作成します。エンティティ Bean の `create()` メソッドを呼び出すと、エンティティの状態がデータベースに挿入され、`create()` メソッドのパラメータの値にしたがってエンティティの変数が初期化されます。エンティティ Bean の `create()` メソッドはリモートインターフェースを返しますが、対応する `ejbCreate()` メソッドはエンティティインスタンスの主キーを返します。

すべてのエンティティ Bean インスタンスに、自分を一意に識別する主キーが割り当てられます。さらに、特定のエンティティオブジェクトの検索に使用できる二次キーを持つ場合もあります。

検索メソッドと主キークラス

エンティティ Bean を検索するためのデフォルトの検索メソッドは `findByPrimaryKey()` です。このメソッドは、主キーの値を使ってエンティティオブジェクトを検索します。シグニチャは次のとおりです。

```
<remote interface> findByPrimaryKey( <key type> primaryKey )
```

エンティティ Bean は、`findByPrimaryKey()` メソッドを実装します。`primaryKey` パラメータは、デプロイメントデスクリプタの中で定義される別の主キークラスです。このキーの型は主キーの型であり、かつ RMI-IIOP の有効な値型である必要があります。Java クラスでも作成したクラスでも、任意のクラスを主キーのクラスとして使用できます。

たとえば、主キークラス AccountPK が定義されているエンティティ Bean の Account があるとします。AccountPK は String 型で、Account Bean の識別子を保持します。このとき、特定の Account エンティティ Bean インスタンスまでのリファレンスを取得するには、次のサンプルコードのように、AccountPK をアカウント識別子に設定し、`findByPrimaryKey()` メソッドを呼び出します。

```
AccountPK accountPK = new AccountPK("1234-56-789");
Account source = accountHome.findByPrimaryKey( accountPK );
```

Bean プロバイダは、クライアントが使用する追加の検索メソッドを定義できます。

作成メソッドと削除メソッド

クライアントでは、ホームインターフェースで定義されている作成メソッドを使用して、エンティティ Bean を作成することもできます。クライアントがエンティティ Bean に `create()` メソッドを呼び出すと、エンティティオブジェクトの新しいインスタンスがデータストアに保存されます。新しいエンティティオブジェクトには、識別子として主キーが割り当てられます。`create()` メソッドにパラメータを渡して、その値で新しいエンティティオブジェクトの状態を初期化することもできます。

エンティティ Bean は、そのデータがデータベース内にある限り存続します。エンティティ Bean の存続期間は、クライアントのセッションに結び付けられてはいません。エンティティ Bean は、`remove()` メソッドのいずれかを呼び出して削除します。これらのメソッドは、Bean を削除するとともに、エンティティデータの基底の表現をデータベースから削除します。また、DBMS や既存のアプリケーションなどでデータベースレコードを削除すれば、エンティティオブジェクトを直接削除することもできます。

メソッドの呼び出し

Bean のリモートインターフェースへのリファレンスを取得したら、クライアントはリモートインターフェースで定義されているメソッドを呼び出すことができます。クライアントにとって最も重要なメソッドは、Bean のビジネスロジックに関係したメソッドです。このほかにも、Bean とそのインターフェースに関する情報を取得するメソッド、Bean オブジェクトのハンドルを取得するメソッド、2 つの Bean が完全に同一かどうかを判定するメソッド、Bean インスタンスを削除するメソッドなどがあります。

次のサンプルコードは、クライアントがエンタープライズ Bean のメソッド（この例では cart セッション Bean）を呼び出す内容を示したものです。ここに紹介したのは、クレジットカードの所有者のために新しいセッション Bean インスタンスを作成し、Cart リモートインターフェースへのリファレンスを取得する箇所から始まるクライアントコードです。以上で、クライアントから Bean のメソッドを呼び出す準備が完了します。

クライアントでは、まず新しい book オブジェクトを作成し、title および price パラメータを設定します。次に、エンタープライズ Bean のビジネスメソッド `addItem()` を呼び出して、book オブジェクトをショッピングカートに追加します。`addItem()` メソッドは、CartBean セッション Bean で定義され、Cart リモートインターフェースによって公開されます。ここには示されていないほかの品目も追加し、クライアント自身の `summarize()` メソッドを呼び出して、品目をショッピングカートに記録します。さらに、`remove()` メソッドで、Bean インスタンスを削除します。なお、クライアントがエンタープライズ Bean のメソッドを呼び出す方法は、クライアント自身の `summarize()` などの任意のメソッドを呼び出すときと同じです。

```
...
Cart cart;
{
   ...
   // Bean のリモートインターフェースへのリファレンスを取得
   cart = home.create(cardHolderName, creditCardNumber, expirationDate);
}
// 新しい book オブジェクトを作成
Book knuthBook = new Book("The Art of Computer Programming", 49.95f);
// カートに新しい book 品目を追加
cart.addItem(knuthBook);

...
// 現在カート内にある品目を一覧
summarize(cart);
cart.removeItem(knuthBook);
...
```

Bean インスタンスの削除

The セッション Bean に対する `remove()` メソッドのはたらきは、エンティティ Bean の場合とは異なります。セッションオブジェクトは 1 つのクライアントごとに生成され、永続的ではないので、セッション Bean のクライアントは、セッションオブジェクトの処理が終了したら `remove()` メソッドを呼び出す必要があります。クライアントは、2 つの `remove()` メソッドを使用できます。セッションオブジェクトを削除するには `javax.ejb.EJBObject.remove()` メソッドを使用し、セッションハンドルを削除するには `javax.ejb.EJBHome.remove(Handle handle)` メソッドを使用します。ハンドルの詳細については、80 ページの「Bean のハンドルの使い方」を参照してください。

必ずしもクライアントでセッションオブジェクトを削除する必要はありませんが、プログラミングの練習としてお勧めします。クライアントがステートフルセッション Bean オブジェクトを削除しないと、タイムアウト値で指定した一定時間が経過した後で、コンテナがそのオブジェクトを削除します。タイムアウト値はデプロイメントのプロパティです。ただし、クライアントは、セッションのハンドルを将来参照するために保持し続けることもできます。

エンティティ Bean のクライアントがこの問題に対処する必要はありません。エンティティ Bean はトランザクションの持続期間に限ってクライアントに関連付けられ、エンティティ Bean のアクティブ化と非アクティブ化などの存続期間の管理は、コンテナが担当します。エンティティ Bean のクライアントが Bean の `remove()` メソッドを呼び出すのは、基底のデータベースからそのエンティティオブジェクトを削除する場合に限られます。

Bean のハンドルの使い方

エンタープライズ Bean はハンドルでも参照できます。ハンドルは、Bean までのリファレンスとしてシリアライズできます。ハンドルは Bean のリモートインターフェースから取得できます。取得したハンドルは、ファイルなどの永続的ストレージに保存でき、再びストレージから取り出して、エンタープライズ Bean へのリファレンスの再確立に使用できます。

ただし、リモートインターフェースのハンドルはその Bean へのリファレンスだけ再生でき、Bean 自体を再生できません。別のプロセスで Bean が削除された場合や、システムがクラッシュまたはシャットダウンしたために Bean インスタンスが削除された場合は、クライアントアプリケーションからハンドルで Bean までのリファレンスを再確立しても、例外が生成されます。

Bean インスタンスが残っているかどうかわからない場合は、リモートインターフェースのハンドルを使用しないでください。Bean のホームハンドルを保存しておけば、後で Bean の作成メソッドや検索メソッドを呼び出して Bean オブジェクトを再生できます。

Bean インスタンスを作成したクライアントは、`getHandle()` メソッドでこのインスタンスまでのハンドルを取得できます。取得したハンドルはシリアライズしたファイルに書き込むことができ、再びそのファイルを読み取ってオブジェクトを Handle 型にキャストできます。次に、ハンドルに `getEJBObject()` メソッドを呼び出して Bean までのリファレンスを取得し、`getEJBObject()` の結果をその Bean に応じた型にキャストします。

その例を次の CartClient プログラムで示します。ここでは、CartBean セッション Bean のハンドルを使って次の処理を行います。

```
import java.io;
import javax.ejb.Handle;
...
Cart cart;
...
cart = home.create(cardHolderName, creditCardNumber, expirationDate);
// cart オブジェクトで getHandle を呼び出して、そのハンドルを取得
cartHandle = cart.getHandle();
// シリアライズしたファイルにハンドルを書き込む
FileOutputStream f = new FileOutputStream ("carthandle.ser");
ObjectOutputStream o = new ObjectOutputStream(f);
o.writeObject(myHandle);
o.flush();
o.close();
...
// 後でファイルからハンドルを読み込む
FileInputStream fi = new FileInputStream ("carthandle.ser");
ObjectInputStream oi = new ObjectInputStream(fi);
// ファイルからオブジェクトを読み込み、ハンドルにキャスト
cartHandle = (Handle)oi.readObject();
oi.close();
...
// そのハンドルを使用して、Bean インスタンスへのリファレンスを取得
try {
   Object ref = context.lookup("cart");
   Cart cart1 = (Cart) javax.rmi.PortableRemoteObject.narrow(ref, Cart.class);
   ...
} catch (RemoteException e) {
   ...
}
...
```

セッション Bean のハンドルの役目が終了したら、クライアントは、`javax.ejb.EJBHome.remove(Handle handle)` メソッドで Bean を削除します。

トランザクション管理

クライアントプログラムのトランザクションは、エンタープライズ Bean（またはコンテナ）で管理させなくても、クライアントプログラム自体が管理できます。このようなクライアントは、トランザクションを自分で管理するセッション Bean と同じようにトランザクションを管理します。

トランザクションを管理するクライアントには、トランザクション境界を定める責任があります。つまり、このようなクライアントは、トランザクションの開始と終了（コミットまたはロールバック）を指示する必要があります。

クライアント でトランザクションを管理するには、`javax.transaction.UserTransaction` インターフェースを使用します。まず、JNDI で、`UserTransaction` インターフェースまでのリファレンスを取得します。`UserTransaction` コンテキストを取得したら、`UserTransaction.begin()` メソッドでトランザクションを開始します。その後、`UserTransaction.commit()` メソッドでトランザクションをコミットして終了します(または、`UserTransaction.rollback()` でロールバックしてトランザクションを終了します)。開始から終了までの間に、照会と更新を行います。

このサンプルコードでは、クライアント自身でトランザクションを管理するコードの例です。クライアントによるトランザクション管理に関係する部分が太字で強調されています。

```
...
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;
...
public class clientTransaction {
   public static void main (String[] argv) {
      UserTransaction ut = null;
      InitialContext initContext = new InitialContext();
      ...
      ut = (UserTransaction)initContext.lookup("java:comp/UserTransaction");
      // トランザクションを開始します。
      ut.begin();
      // トランザクション作業を実行します。
      ...
      // トランザクションをコミットまたはロールバックします。
      ut.commit(); // または ut.rollback();
      ...
   ]
]
```

エンタープライズ Bean に関する情報の取得

エンタープライズ Bean に関する情報は、メタデータと呼ばれます。クライアントでは、エンタープライズ Bean のホームインターフェースの `getMetaData()` メソッドで Bean のメタデータを取得できます。

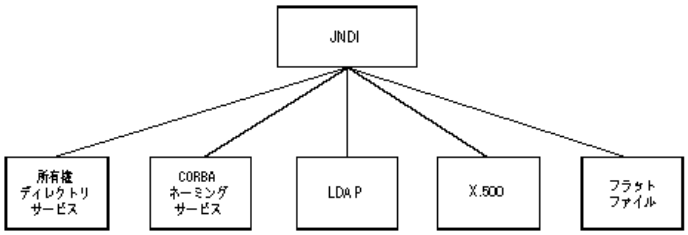
開発環境やツールビルダでは、`getMetaData()` メソッドを頻繁に使用します。これは、インストール済み Bean どうしのリンクなどを行う場合にエンタープライズに関する情報が必要なためです。スクリプト操作を行うクライアントも、Bean に関するメタデータを必要とする場合があります。

ホームインターフェースへのリファレンスを取得したクライアントは、ホームインターフェースの `getEJBMetaData()` メソッドを呼び出すことができます。その後、`EJBMetaData` インターフェースのメソッドを呼び出して、次の情報を取得します。

- `EJBMetaData.getEJBHome()` で Bean の `EJBHome` ホームインターフェースを取得。
- `EJBMetaData.getHomeInterfaceClass()` で Bean のホームインターフェースクラスオブジェクト（インターフェース、クラス、フィールド、メソッドなど）を取得。
- `EJBMetaData.getRemoteInterfaceClass()` で Bean のリモートインターフェースのクラスオブジェクト（すべてのクラス情報）を取得。
- `EJBMetaData.getPrimaryKeyClass()` で Bean の主キークラスオブジェクトを取得。
- Bean がセッション Bean かエンティティ Bean かを `EJBMetaData.isSession()` で判断。セッション Bean ならば true が返されます。
- セッション Bean がステートレスかステートフルを `EJBMetaData.isStatelessSession()` で判断。ステートレスセッション Bean ならば true が返されます。

JNDI のサポート

EJB 仕様では、ホームインターフェースを取得するための JNDI API を定義します。JNDI は、ほかのサービス（CORBA のネーミングサービス、LDAP/X.500、フラットファイル、専用のディレクトリサービスなど）の上に実装されます。次の図は、インプリメンテーションのさまざまな選択肢です。通常、EJB サーバーのプロバイダは、JNDI の特定のインプリメンテーションを選択します。



クライアントにとって、JNDI の下に実装されているテクノロジは、重要ではありません。クライアントは JNDI API だけ使用する必要があります。

EJB から CORBA へのマッピング

CORBA と Enterprise JavaBeans の関係にはさまざまな側面があります。その主な 3 つの側面として、ORB による EJB コンテナ／サーバーのインプリメンテーション、EJB 中間層への既存システムの統合、および非 Java コンポーネント（クライアント）からエンタープライズ Bean へのアクセスが挙げられます。ただし、現在の EJB 仕様は 3 番めの側面だけを定めています。

EJB の基本構造を実装するには、CORBA が最も適切で自然なプラットフォームです。EJB 仕様のすべての項目は、次の CORBA コア仕様または CORBA サービスを使って処理できます。

- **分散のサポート。**CORBA コアおよび CORBA ネーミングサービス。
- **トランザクションのサポート。**CORBA オブジェクトトランザクションサービス
- **セキュリティのサポート。**IIOP-over-SSL などの CORBA セキュリティ仕様。

CORBA を使用すると、非 Java コンポーネントでもアプリケーションに統合できます。既存のシステム、既存のアプリケーション、および各種のクライアントがこのようなコンポーネントに相当します。また、OTS と、IDL マッピングをサポートする任意のプログラミング言語を使用して、バックエンドシステムを容易に統合できます。この場合は、EJB コンテナが OTS と IIOP API を提供する必要があります。

EJB 仕様では、非 Java クライアントからエンタープライズ Bean へのアクセス可能性と、EJB から CORBA へのマッピングが規定されています。EJB/CORBA マッピングの目的は次のとおりです。

- CORBA をサポートするプログラミング言語で記述されたクライアントと、CORBA ベースの EJB サーバーで実行されるエンタープライズ Bean との相互運用性をサポートする。
- クライアントプログラムから CORBA オブジェクトへの呼び出しとエンタープライズ Bean への呼び出しが同一トランザクション内に混在できるようにする。
- さまざまなベンダーから提供された複数の CORBA ベース EJB サーバーで実行される複数のエンタープライズ Bean が関与する分散トランザクションをサポートする。

このマッピングのベースは、Java から IDL へのマッピングです。EJB 仕様では、分散、ネーミング、トランザクション、およびセキュリティにかかわるマッピングが定められています。以下では、これらのマッピングについて説明します。これらのマッピングでは、OMG の Object-by-Value 仕様で導入された新しい IDL 機能が使用されるため、ほかのプログラミング言語との相互運用性を維持するには、CORBA 2.3 準拠の ORB が必要です。

分散のためのマッピング

エンタープライズ Bean には、リモートにアクセスできるリモートインターフェースとホームインターフェースという 2 つのインターフェースがあります。これらのインターフェースに Java ／ IDL マッピングを適用すると、対応する IDL 仕様ができます。EJB 仕様で定義されているベースクラスは、同じ方法で IDL にマッピングできます。

たとえば、口座間で預金の転送するメソッドを備え、残高不足例外を生成する ATM エンタープライズセッション Bean の IDL インターフェースについて考えてみます。ホームインターフェースとリモートインターフェースに Java ／ IDL マッピングを適用すると、次の IDL インターフェースが得られます。

```
module transaction {
   module ejb {
   valuetype InsufficientFundsException : ::java::lang::Exception {};
   exception InsufficientFundsEx {
      ::transaction::ejb::InsufficientFundsException value;
   };
   interface Atm : ::javax::ejb::EJBObject{
   void transfer (in string arg0, in string arg1, in float arg2)
      raises (::transaction::ejb::InsufficientFundsEx);
   };
   interface AtmHome : ::javax::ejb::EJBHome {
      ::transaction::ejb::Atm create ()
      raises (::javax::ejb::CreateEx);
   };
};};};};
```

ネーミングのためのマッピング

任意の CORBA クライアントからエンタープライズ Bean にアクセスできる CORBA ベースの EJB 実行時環境を構築するには、CORBA ネーミングサービスで、エンタープライズ Bean のホームインターフェースの公開と解決を行う必要があります。ランタイムは直接 CORBA ネーミングサービスを使用するか、JNDI と、CORBA ネーミングサービスへの標準マッピングを使用して、間接的に CORBA ネーミングサービスを使用します。

JNDI 名の文字列表現は、「`directory1/directory2/.../directoryN/objectName`」という形式です。CORBA ネーミングサービスでは、複数の名前要素のシーケンスで名前を定義しています。

```
typedef string Istring;
   struct NameComponent {
      Istring id;
      Istring kind;
   };
typedef sequence<NameComponent> Name;
```

JNDI 文字列名の中の / で区切られた各名前は、それぞれ 1 つの名前要素にマッピングされます。左端の要素が CORBA ネーミングサービス名の最初のエントリに対応します。

JNDI 文字列名は、JNDI ルートコンテキストと呼ばれるネーミングコンテキストを基準とする相対名です。JNDI ルートコンテキストは、CORBA ネーミングサービスの初期コンテキストに対応します。CORBA ネーミングサービス名は、CORBA の初期コンテキストを基準とする相対名です。

CORBA プログラムは、ORB（擬似）オブジェクトの `resolve_initial_references("NameService")` で、初期 CORBA ネーミングサービスのネーミングコンテキストを取得します。CORBA ネーミングサービスはルートがなくてもネーミングコンテキストを構成できるため、ルートコンテキストの表記は不要です。ORB の初期化によって、`resolve_initial_references()` が返すコンテキストが決まります。

たとえば、JNDI 文字列名「transaction/corbaEjb/atm」で登録されている ATM セッション Bean のホームインターフェースを C++ クライアントから検索することを考えます。まず初期ネーミングコンテキストを取得します。

```
Object_ptr obj = orb->resolve_initial_refernces("NameService");
NamingContext initialNamingContext= NamingContext.narrow( obj );
if( initialNamingContext == NULL ) {
   cerr << "Couldn't initial naming context" << endl;
   exit( 1 );
}
```

次に、CORBA ネーミングサービス名を作成し、前に説明したマッピングにしたがって初期化します。

```
Name name = new Name( 1 );
name[0].id = "atm";
name[0].kind = "";
```

初期ネーミングコンテキストで名前を解決します。ここでは、初期化が正常に実行され、エンタープライズ Bean のネーミングドメインのコンテキストもあることを想定しています。得られた CORBA オブジェクトを予想される型にナローイングし、ナローイングが成功したかどうかを確認します。

```
Object_ptr obj = initialNamingContext->resolve( name );
ATMSessionHome_ptr atmSessionHome = ATMSessionHome.narrow( obj );
if( atmSessionHome == NULL ) {
   cerr << "Couldn't narrow to ATMSessionHome" << endl;
   exit( 1 );
}
```

トランザクションのためのマッピング

CORBA ベースのエンタープライズ Bean 実行時環境で、エンタープライズ Bean と CORBA クライアントを同じトランザクションに関与させるには、CORBA オブジェクトトランザクションサービスでトランザクションを制御します。

デプロイメントするエンタープライズ Bean は、それぞれ異なるトランザクションポリシーでインストールできます。ポリシーは、エンタープライズ Bean のデプロイメントデスクリプタの中で定義します。

トランザクション対応のエンタープライズ Bean について、次の規則が定義されています。CORBA クライアントは、エンタープライズ Bean のリモートおよびホームインターフェースに対応する IDL インターフェースが生成したスタブを介して、エンタープライズ Bean を呼び出します。クライアントがトランザクションに関与する場合は、CORBA オブジェクトトランザクションサービスが提供するインターフェースを使用します。たとえば、C++ クライアントは、先のサンプルの ATM セッション Bean を次のように呼び出すことができます。

```
try {
   ...
   // トランザクション current を取得
   Object_ptr obj = orb->resolve_initial_refernces("Current");
   Current current = Current.narrow( obj );
   if( current == NULL ) {
      cerr << "Couldn't resolve current" << endl;
      exit( 1 );
   }
// トランザクションを実行
   try {
      current->begin();
      atmSession->transfer("checking", "saving", 100.00 );
      current->commit( 0 );
   }   catch( ... ) {
      current->rollback();
   }
}
catch( ... ) {
   ...
}
```

セキュリティのためのマッピング

セキュリティについては、EJB 仕様は主にエンタープライズ Bean へのアクセス制限を規定します。CORBA では、次のような事例も含め、アイデンティティのさまざまな定義方法が定められています。

- **通常の IIOP。**CORBA の principal インターフェースは、1998 年初頭に使用されなくなりました。このインターフェースの目的は、クライアントのアイデンティティを判定することでした。しかし、これとは別に GIOP という CORBA セキュリティサービスが実装されました。
- GIOP 仕様には、サービスコンテキストと呼ばれるコンポーネントがあります。これは値のペアからなる配列です。識別子は CORBA long で、値はオクテットのシーケンスです。サービスコンテキスト内のエントリで、呼び出し元を識別できます。
- **セキュア IIOP。**CORBA セキュリティ仕様では、アイデンティティ用の不透過データ型が定義されています。これらのアイデンティティの実際のデータ型は、選択されたセキュリティメカニズム（GSS Kerberos、SPKM、CSI-ECMA など）によって決まります。
- **IIOP-over-SSL。**SSL では、X.509 認証でサーバーを識別します。オプションでクライアントも識別します。サーバーは、クライアントに証明書を要求し、それをクライアントのアイデンティティとして使用できます。

第 11 章

VisiClient コンテナの使い方

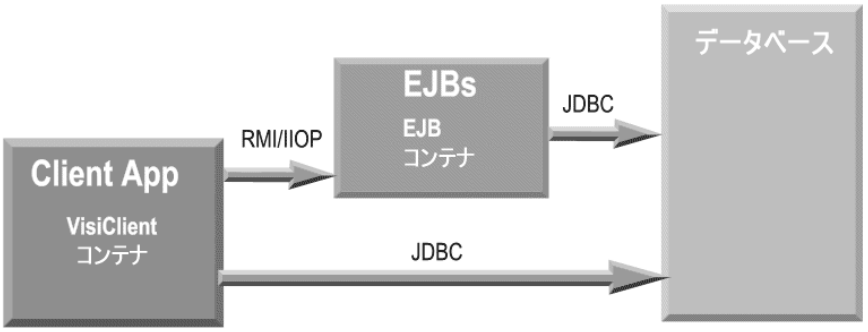
VisiClient は、アプリケーションクライアントのサービスに J2EE 環境を提供するコンテナです。

コンテナは J2EE アプリケーションにとって不可欠な要素で、ほとんどのアプリケーションはその種類に合ったコンテナを提供しています。アプリケーションクライアントは、システムサービスの提供をコンテナに依存します。これはすべての J2EE コンポーネントで共通です。

アプリケーションクライアントのアーキテクチャ

J2EE アプリケーションクライアントは、独自の Java 仮想マシン内で実行される第 1 階層クライアントプログラムです。アプリケーションクライアントは、Java 技術ベースのアプリケーションモデルにしたがって動作します。つまり、main メソッドによって起動され、仮想マシンが終了するまで動作します。ほかの J2EE アプリケーションコンポーネントと同様に、アプリケーションクライアントは、システムサービスの提供をコンテナに依存します。ただし、アプリケーションクライアントの場合、これらのサービスには制限があります。

図 11.1　VisiClient のアーキテクチャ



パッケージングとデプロイメント

アプリケーションクライアントのコンポーネントを VisiClient コンテナにデプロイメントするには、XML を使ったデプロイメントデスクリプタによる指定が必要です。アプリケーションクライアントと、それを J2EE 1.3 準拠のコンテナにデプロイメントする方法の詳細については、『J2EE Specification v1.3』を参照してください。

アプリケーションクライアントは、JAR ファイルにパッケージされ、デプロイメントデスクリプタを 1 つ格納します。これは、ほかの J2EE アプリケーションコンポーネントと同様です。デプロイメントデスクリプタは、アプリケーションから参照されるエンタープライズ Bean と外部リソースを定義します。アプリケーションクライアントコンポーネントのパッケージと編集には、Borland EAppServer（AppServer）のデプロイメントデスクリプタエディタを使用できます。デプロイメントデスクリプタの使い方の詳細については、『*管理コンソールユーザーズガイド*』の「デプロイメントデスクリプタエディタの使い方」を参照してください。

EJB やそのリソースに名前を割り当てるなど、デプロイメント時にさまざまな機能を設定するためにデプロイメントデスクリプタが必要になります。アプリケーションクライアントを VisiClient コンテナにデプロイメントするには、次の条件が必要です。

- クライアント側のすべてのクラスが 1 つの JAR にパッケージされていること。必要なクライアント JAR およびファイルについては、以下の項を参照してください。正しく作成された JAR には次の内容が含まれます。
  - アプリケーション固有のクラス。アプリケーションのエントリポイントを持つクラス（メインクラス）を含みます。
  - 上の JAR ファイル内に、次のファイルが格納されている META-INF サブディレクトリがあること。
    - マニフェストファイル
    - J2EE 1.3 仕様で要求されている標準 XML ファイル（application-client.xml）
    - ベンダー固有の XML ファイル（application-client-borland.xml）
- RMI-IIOP スタブを個別にパッケージすることもできます。その場合は、このファイルのマニフェストファイルのクラスパス属性を適切な値に設定する必要があります。このように作成された JAR は、スタンドアロンコンテナまたは EAR ファイルにデプロイメントできます。この章では、この手順について後述します。

VisiClient コンテナの利点

VisiClient は、J2EE アプリケーションを使用することで得られるさまざまな利点をユーザーにもたらします。次のような機能があります。

- **クライアントコードの可搬性：**アプリケーションは、J2EE 仕様で推奨されているように、論理名を使ってデータベース接続、リモート EJB、環境変数などのリソースにアクセスできます。コンテナは、J2EE 仕様にしたがい、これらのリソースを管理対象のオブジェクトとしてローカルの JNDI 名前空間（java:comp/env）にエクスポーズします。
- **JDBC 接続プール：**Borland AppServer 内のクライアントアプリケーションは、JDBC 2 ベースのデータソース（ファクトリ）を使用できます。VisiClient コンテナは、JDBC 2 ベースのデータソースを使用する AppServer 内のクライアントアプリケーションに接続プールを提供します。たとえば、VisiClient コンテナにより、アプリケーションは java.net.URL、JMS、およびメールのファクトリを使用できます。

データソースと URL ファクトリは、起動時にクライアントコンテナの仮想マシンに存在するインプロセスのローカル JNDI サブコンテキストにデプロイメントされます。その他の res-ref-type（JMS やメールなど）は、各製品のベンダーから提供される関連ツールを使って設定およびデプロイメントされます。設定とデプロイメントの詳細については、『Borland AppServer *開発者ガイド*』のデプロイメント、データソース、トランザクションに関する章を参照してください。

Document Type Definition（DTD）

J2EE 準拠の各アプリケーションクライアントモジュールには、それぞれ 2 種類のデプロイメントデスクリプタがあります。1 つは J2EE 標準のデプロイメントデスクリプタで、もう 1 つは J2EE 仕様にしたがったベンダー固有のファイルです。

J2EE アプリケーションクライアントデプロイメントデスクリプタに対する XML 文法は、J2EE アプリケーションクライアント DTD で定義します。アプリケーションクライアントデプロイメントデスクリプタのルート要素は、「application-client」です。

メモ 通常、XML 要素の内容は、大文字と小文字の区別があります。有効なアプリケーションクライアントデプロイメントデスクリプタ内には、次の DOCTYPE 宣言が必要です。

```
<!DOCTYPE application-client PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE
Application Client
 1.3//EN';';http://java.sun.com/j2ee/dtds/application-client_1_3.dtd'>
```

アプリケーションクライアントのベンダー固有デプロイメントデスクリプタ内には、次の DOCTYPE 宣言が必要です。

```
<!DOCTYPE application-client PUBLIC "-//Borland Corporation//DTD J2EE
Application Client
 1.3//EN""http://www.borland.com/devsupport/appserver/dtds/application-
client_1_3-borland.dtd">
```

Borland 固有のアプリケーションクライアント DTD の内容は次のとおりです。

```
        <!ELEMENT application-client (ejb-ref*, resource-ref*, property*)>
        <!ELEMENT ejb-ref (ejb-ref-name, jndi-name)>
        <!ELEMENT resource-ref (res-ref-name, jndi-name)>
        <!ELEMENT property (prop-name, prop-type, prop-value)>
        <!ELEMENT prop-name (#PCDATA)>
        <!ELEMENT prop-type (#PCDATA)>
        <!ELEMENT prop-value (#PCDATA)>
        <!ELEMENT ejb-ref-name (#PCDATA)>
        <!ELEMENT jndi-name (#PCDATA)>
        <!ELEMENT res-ref-name (#PCDATA)>
```

ここで、ejb-ref-name と res-ref-name は、J2EE XML ファイル内の対応する要素の名前です。また、オブジェクトが JNDI 内にデプロイメントされるときに使用される絶対 JNDI 名です。

DTD を使った XML のサンプル

上記のように、各アプリケーションクライアントには、標準ファイルとベンダー固有ファイルの 2 つの XML ファイルが必要です。

標準ファイルのサンプル：

```
<?xml version="1.0" encoding="ISO8859_1"?>

<!DOCTYPE application-client PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE
Application Client 1.3//EN' 'http://java.sun.com/j2ee/dtds/application-
client_1_3.dtd'>
<application-client>
  <display-name>SimpleSort</display-name>
  <description>J2EE AppContainer 仕様準拠の Sort クライアント</description>
  <env-entry>
    <description>
      環境エントリのテスト
    </description>
    <env-entry-name>myStringEnv</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>MyStringEnvEntryValue</env-entry-value>
  </env-entry>
  <ejb-ref>
    <ejb-ref-name>ejb/Sort</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>SortHome</home>
    <remote>Sort</remote>
    <ejb-link>sort</ejb-link>
```

```
  </ejb-ref>
  <resource-ref>
    <description>
       DD セクションで指定される JDBC データソースへのリファレンス
    </description>
    <res-ref-name>jdbc/CheckingDataSource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref></application-client>
```

ベンダー固有ファイルのサンプル：

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

<!DOCTYPE application-client PUBLIC "-//Borland Corporation//DTD J2EE
Application Client 1.3//EN"
 "http://www.borland.com/devsupport/appserver/dtds/application-client_1_3-
borland.dtd">
<application-client>
          <ejb-ref>
            <ejb-ref-name>ejb/Sort</ejb-ref-name>
            <jndi-name>sort</jndi-name>
          </ejb-ref>
          <resource-ref>
            <res-ref-name>jdbc/CheckingDataSource</res-ref-name>
<jndi-name>datasources/OracleDataSource</jndi-name>
          </resource-ref>
</application-client>
```

環境エントリ、ejb-ref、および resource-ref の詳細については、Sun Microsystem の EJB 2.0 仕様 www.java.sun.com/j2ee の関連セクションを参照してください。

サンプルコード

このサンプルは、論理ローカル JNDI ネーミングコンテキストの使い方を示します。ここでは、前のセクションで指定されたデプロイメントデスクリプタをクライアントがどのように使用するかを示しています。

```
// ネーミングサービスを使って JNDI コンテキストを取得し、リモートオブジェクト
を作成します。

    javax.naming.Context context = new javax.naming.InitialContext();
    Object ref = context.lookup("java:comp/env/ejb/Sort");
    SortHome home = (SortHome) javax.rmi.PortableRemoteObject.narrow(ref,
      SortHome.class);
    Sort sort = home.create();
    // JNDI を使って環境エントリの値を取得します。
    Object envValue = context.lookup("java:comp/env/myStringEnv");
    System.out.println("Value of env entry = "+ (java.lang.String) envValue );
    // UserTransaction オブジェクトを探します。
    javax.transaction.UserTransaction userTransaction =
      (javax.transaction.UserTransaction)  context.lookup("java:comp/
UserTransaction");

userTransaction.begin();
    // resource-ref 名を使ってデータソースを探します。
    Object resRef = context.lookup("java:comp/env/jdbc/CheckingDataSource");
    java.sql.Connection conn = ((javax.sql.DataSource)resRef).getConnection();
    // データベース作業を実行します。
    userTransaction.commit();
..............
```

リファレンスとリンクのサポート

アプリケーションのアセンブリとデプロイメントの間に、すべての EJB とリソースリファレンスが正しくリンクされていることを確認する必要があります。EJB とリソースリファレンスの詳細については、Sun Microsystem の EJB 2.0 仕様と J2EE 1.3 仕様を参照してください。

Borland AppServer のクライアントコンテナでは ejb-link を使用できます。スタンドアロン JAR ファイルの場合は、JAR がデプロイメントされる前に ejb-link を解決する必要があります。クライアントデプロイメントデスクリプタのベンダー固有のセクション内で、対象 Bean の JNDI 名が指定されていることが必要です。

クライアント JAR がエンタープライズアプリケーションアーカイブ（EAR）の一部である場合は、対象 EJB の JNDI 名が別の EJB JAR に存在することがあります。クライアント検証ツールは、ejb-link タグで指定された名前の対象 EJB が存在することを確認します。

実行時にコンテナは、ejb-link 名に対応する対象 EJB を EAR 内で解決（検索）し、そのエンタープライズ Bean の JNDI 名を使用します。アプリケーションクライアントが独自の Java 仮想マシンで実行されることに注意してください。アプリケーションクライアントの場合、ejb-link は最適化されません。これは、同じコンテナ内にある別の EJB を参照する EJB の場合とは異なります。

アプリケーションクライアントコンテナのデプロイメントデスクリプタで EJB リファレンスと ejb-link を使用する場合は、次の規則にしたがってください。

1. ejb-link ではない ejb-ref は、参照される（対象）EJB の JNDI 名を保持する Borland 固有ファイル内にエントリがある必要があります。
2. ejb-link の要素を持つ ejb-ref は、次の規則にしたがう必要があります。
   - その ejb-ref がスタンドアロン JAR のクライアント JAR 内にある場合は、最初の規則が適用されます。つまり、その ejb-ref の JNDI 名が同じ JAR のデプロイメントデスクリプタ内で解決されている必要があります。
   - その ejb-ref が、アプリケーションアーカイブ（EAR）に埋め込まれているクライアント JAR 内にある場合は、対象 EJB の JNDI 名が application-client-inprise.xml ファイル内に存在する必要はありません。この場合、ejb-link 要素の名前は、参照されたエンタープライズ Bean を含む ejb-jar の絶対パスを指定するパス名で構成されます。このパス名には対象となる bean の ejb-name が追加されており、「#」記号でパス名と分けられています。このパス名は、エンタープライズ Bean を参照するアプリケーションクライアントを含む JAR ファイルに関連付けられているため、複数のエンタープライズ Bean が同じ ejb-name を持っていても、それぞれを区別することができます。

   パスが指定されていない場合、コンテナは EAR にある EJB JAR リスト で最初に一致した EJB 名を選択し、ejb-link 要素に同じ名前の Bean が見つからない場合は例外を生成します。

VisiClient コンテナの使い方

次のコマンドラインは、VisiClient コンテナの使い方を示します。

```
Prompt% appclient <client-archive> [-uri <uri>] [client-arg1 client-arg2 ..]
```

次の表は、VisiClient コンテナのコマンドライン要素と定義をまとめたものです。

要素	定義
`<client-archive>`	スタンドアロンクライアント JAR、 またはクライアント JAR を含む EAR
`-uri`	EAR ファイル内のクライアント JAR の相対位置。 EAR 内にある JAR ファイルの場合、 この要素は必須です。
`<client-args>`	クライアントのメインクラスに渡すスペース区切りの引数リスト。

VisiClient コンテナの使い方のサンプル

次のコマンドラインは、アプリケーションクライアントの使い方を具体的に示したものです。これらのサンプルでは、appclient 起動プログラムは、VisiClient に必要なクラスパスを設定します。

このサンプルは、install_dir/examples/j2ee/hello ディレクトリ内の Hello サンプルの中にもあります。サーバー（EJB コンテナ）が動作している場合、EAR ファイルに埋め込まれているクライアントを実行するには、次のコマンドを使用します。

```
appclient me install_dir¥examples¥j2ee¥build¥hello¥hello.ear -uri
helloclient.jar
```

スタンドアロン JAR ファイル内のクライアントを実行するには、次のコマンドを使用します。

```
appclient me install_dir¥examples¥j2ee¥build¥hello¥client¥helloclient.jar
```

AppServer が動作していないマシン上での J2EE クライアントアプリケーションの実行

Borland Enterprise Server がインストールされていないクライアントマシン上で J2EE アプリケーションクライアントを実行するには、次に示す VisiClient ファイルをクライアントマシンにコピーし、次に示す処理を実行します。

1 次の JAR ファイルを <install_dir>/lib からクライアントマシンにコピーします。
   - lm.jar
   - xmlrt.jar
   - asrt.jar
   - vbjorb.jar
   - vbsec.jar
   - jsse.jar
   - jaas.jar
   - vbejb.jar

2 次の JAR ファイルを <install_dir>/jms/tibco/clients/java からクライアントマシンにコピーします。
   - tibjms.jar

3 <install_dir>/bin/appclient.config をクライアントマシンにコピーします。

4 <install_dir>/BES/bin/appclient.exe をクライアントマシンにコピーします。

appclient を使って J2EE クライアントを実行するには、次の手順にしたがいます。

1 appclient.exe および JDK への PATH を設定します。

2 appclient.config を編集して JAVA_HOME と lib PATH を変更します。

3 <client_application_folder>/client から J2EE クライアントを実行します。

既存のアプリケーションに VisiClient コンテナ機能を埋め込む

VisiClient コンテナでクライアントアプリケーションのデプロイメントや実行を行うかわりに、既存のアプリケーションにクライアントコンテナの機能を埋め込むというプログラム的なアプローチも可能です。この場合、クライアントアプリケーションは `main()` メソッドを実装するクラスを実行することで、一般的な Java と同じ方法で起動できます。

VisiClient コンテナ機能をアプリケーションに埋め込むには、次のメソッドを呼び出す必要があります。

```
public static void com.borland.appclient.Container.init
  (java.io.InputStream   deploymentDescriptorSun,
  java.io.InputStream deploymentDescriptorBorland)
throws IllegalArgumentException;
```

このメソッドは Sun と Borland のデプロイメントデスクリプタが提供する情報に基づいて、「java:comp/env」ネーミングコンテキストを作成および追加します。`deploymentDescriptorSun` および `deploymentDescriptorBorland` パラメータは、デプロイメントデスクリプタに対応するテキストの XML データである必要があります。提供されたデータが有効なデプロイメントデスクリプタと認識されない場合は、例外 `IllegalArgumntException` が返されます。

サンプルコード

次のサンプルでは、このメソッドの使い方を示します。

```
public static void main (String[] args) {
    . . .
      // デプロイメントデスクリプタファイルを読み込みます。
      java.io.FileInputStream ddSun = new
    java.io.FileInputStream("META-INF/application-client.xml");
      java.io.FileInputStream ddBorland = new
    java.io.FileInputStream("META-INF/application-client-borland.xml");
      // クライアントコンテナを初期化します。
      com.borland.appclient.Container.init(ddSun, ddBorland);
      // ejb-ref を使って JNDI で ejb を検索します。
    javax.naming.Context context = new javax.naming.InitialContext();
      Object ref = context.lookup ("java:comp/env/ejb/hello");
    . . .
}
```

メモ このメソッドでロードできるのは、アプリケーションクライアントデスクリプタだけです。したがって、すべての ejb-ref が解決されるか、Borland デスクリプタの jndi-name を指定して見つかる必要があります。Sun デスクリプタの ejb-link では実行できません。ejb-link を使用する場合、アプリケーションや EJB JAR デプロイメントデスクリプタを含めて、アプリケーション全体の完全な知識が必要とされるからです。

マニフェストファイルの使い方

VisiClient コンテナは、アプリケーションの起動に関する情報の取得をマニフェストファイルに依存します。マニフェストファイルは、クライアントアーカイブの META-INF サブディレクトリに保存する必要があります。VisiClient コンテナに関連する属性は次のとおりです。

- 起動時にコンテナによって呼び出されるメインクラス。つまり、マニフェストファイルにはアプリケーションのエントリポイントがあります。
- メインクラスが依存するクラスのクラスパス。クライアント JAR が外部に依存しない場合、またはアプリケーションの起動時にシステム CLASSPATH を使って依存関係が指定される場合、この属性は省略できます。

マニフェストファイルのサンプル

次に、マニフェストファイルのサンプルを示します。

```
Manifest-Version: 1.0
Main-Class: SortClient
Class-Path:
```

このサンプルは、マニフェストファイルの `Main-Class` 属性で指定されたクラスの main メソッドをロードすることによって実行を開始します。このサンプルでは、`SortClient` クラスが指定されています。コンテナは、このクラスに次のシグニチャを持つメソッドがあるものとみなします。

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

この main メソッドが見つからない場合、コンテナはエラーを報告して終了します。VisiClient 付属のクライアント検証ユーティリティは、メインクラスの検索を試み、見つからない場合はエラーを報告します。

例外処理

アプリケーションクライアントコードは、プログラムの実行時に生成される例外を処理する責任があります。処理されなかった例外はコンテナによってキャッチされます。コンテナは例外をログに記録し、JVM のプロセスを終了します。

リソースリファレンスファクトリタイプの使い方

クライアントコンテナにデプロイメントされたクライアントアプリケーションは、VisiTransact JDBC 接続プールと Prepared Statement の再利用の機能を使用できます。設定とデプロイメントの詳細については、『Borland AppServer *開発者ガイド*』のデプロイメント、データソース、トランザクションに関する章を参照してください。AppServer 内のクライアントアプリケーションは、JDBC 2 ベースのデータソースを使用できます。

`javax.sql.DataSource`（有効な res-ref-type の 1 つ）と同様に、VisiClient ではアプリケーションが resource-ref-type の種類として URL、JMS、および Mail ファクトリを使用できます。

`java.net.url` および `java_mail.session` ファクトリは、起動時にクライアントコンテナの仮想マシンに存在するインプロセスのローカル JNDI サブコンテキストにデプロイメントされます。JMS や Mail などのほかの res-ref-type では、それらの製品のベンダーから提供される関連ツールを使用して、設定およびデプロイメントを行う必要があります。

その他の機能

AppServer には、J2EE 仕様の要件を満たす機能のほかにもさまざまな機能が VisiClient に組み込まれています。次のような機能があります。

- **User Transaction インターフェース：**これは、java:comp/env 名前空間で利用でき、JNDI を使って検索できます。このインターフェースは、トランザクションの確立と伝達をサポートします。
- **クライアント検証ツール：**このツールは、スタンドアロンクライアント JAR、または EAR ファイルに埋め込まれたクライアント JAR 上で実行します。検証ツールは次の規則を適用します。
  - クライアント JAR 内のマニフェストファイルでメインクラスが指定されていること。
  - JAR/EAR が有効であること。つまり、必要な正しいマニフェストエントリを持っていること。
  - ejb-ref が有効であること。つまり、対象 EJB の JNDI 名が Borland 固有ファイル内で指定されていること。
  - ejb-ref が ejb-link である場合は、そのアーカイブが EAR ファイルであること。また、ejb-link 値と同じ名前を持つ EJB が EAR ファイル内に存在すること。
  - リソースリファレンスが有効であること。

クライアント検証ツールの使い方

次のコマンドラインは、クライアント検証ツールの使い方を示します。

```
iastool -verify -src <srcjar> -role <DEVELOPER| ASSEMBLER| DEPLOYER>
```

クライアント検証ツールの使い方のサンプル：

```
iastool -verify -src sort.jar -role DEVELOPER
iastool -verify -src sort.ear clients/sort_client.jar -role DEVELOPER
```

使用可能なオプションについては、iastool の 342 ページの「verify」を参照してください。

第 12 章

ステートフルセッション Bean のキャッシュ

EJB コンテナは、Java Session Service（JSS）ベースのハイパフォーマンスキャッシュアーキテクチャにより、ステートフルセッションエンタープライズ Bean をサポートします。オブジェクトプールには、準備完了プールと非アクティブプールという 2 つプールがあります。エンタープライズ Bean は設定可能なタイムアウト値が経過すると、準備完了プールから非アクティブプールに移動します。エンタープライズ Bean が非アクティブプールに移動すると、その Bean の状態がデータベースに保存されます。ステートフルセッションの非アクティブ状態には、次の 2 つの意味があります。

1 メモリリソースを有効活用する。

2 フェイルオーバーを実現する。

Borland の JSS のインプリメンテーションの設定については、51 ページの「Java セッションサービス（JSS）の設定」を参照してください。このマニュアルでは、個々のセッションオブジェクトの非アクティブ状態と永続性を制御するプロパティの使い方について説明しています。

セッション Bean の非アクティブ化

デプロイメント時には、特定のパーティションの EJB コンテナのタイムアウトを、Borland AppServer（AppServer）ツールで設定します。コンテナはアクティブセッション Bean を定期的にポーリングし、前回のアクセス時間を確認します。セッション Bean に対して、タイムアウトで指定した時間を超えてアクセスがなければ、その状態が永続的ストレージに転送され、Bean インスタンスはメモリから削除されます。

単純な非アクティブ化

非アクティブ化タイムアウトは、コンテナレベルで設定します。セッション Bean の状態が固定されてインスタンスがメモリから削除されるまで、アクセスがない状態を継続できる時間は、プロパティ `ejb.sfsd.passivation_timeout` で設定します。値は秒単位で設定します。デフォルト値は 5 秒です。このプロパティは、設定対象のパーティションの `partition.xml` プロパティファイルで設定します。この設定ファイルは、次の場所にあります。

```
<install_dir>/var/domains/base/configurations/<configuration_name>
/ mos/<partition_name>/adm/properties
```

このファイルを編集して `ejb.sfsb.passivation_timeout` プロパティを設定します。

このプロパティにゼロ以外の値を設定すると、デプロイメントデスクリプタにデプロイメント済みのセッション Bean ごとに整数プロパティ `ejb.sfsb.instance_max` も設定できます。このプロパティでは、EJB コンテナのメモリに同時に存在できる特定のステートフルセッション Bean の最大数を定義します。値が最大値に達した後にステートフルセッションの新しいインスタンスを割り振らなければならない状況になると、EJB コンテナからリソース不足を知らせる例外が生成されます。0 は特別な値です。これは最大値が設定されていないことを表します。

`ejb.sfsb.instance_max property` で定義したステートフルセッションの最大値に達すると、EJBコンテナは、新しいBeanの割り振りに対して、整数プロパティ `ejb.sfsb.instance_max_timeout` で定義した時間は、要求をブロックします。その間コンテナは、リソース不足を知らせる例外を生成する前の値まで値が下がるのを待ちます。このプロパティは、ms（1/1000秒）単位で設定します。0は特別な値です。0に設定すると待機時間が0となり、ただちにリソース不足を知らせる例外が生成されます。

積極的な非アクティブ化

JSS の主な利点は、フェイルオーバー能力にあります。JSS を実装するコンテナをいくつか集めて、同じ永続的ストアを使用するように設定すると、互いにフェイルオーバーする役割を設定できます。フェイルオーバーに対する JSS の設定方法については、51 ページの「Java セッションサービス（JSS）の設定」を参照してください。JSS のフェイルオーバー機能を活かすため、Borland では積極的非アクティブ化をオプションとして用意しました。

積極的な非アクティブ化は、タイムアウトに関係なくセッション状態をストレージに保存する機能です。積極的な非アクティブ化を使用するように設定した Bean は、ポーリングのたびにそのセッション状態を固定します。ただし、そのインスタンスはタイムアウトにならないとメモリから削除されません。これにより、あるクラスタでコンテナインスタンスに障害が発生しても、同じバックエンドと通信する同じ JSS インスタンスを利用するほかのコンテナは、前回保存された Bean のバージョンを利用できます。ただし、単純な非アクティブ化と同様に、Bean のタイムアウトに到達した場合はメモリから削除されます。

なお、積極的非アクティブ化は、論理プロパティ `ejb.sfsb.aggressive_passivation` でパーティション規模で設定します。プロパティを `true`（デフォルト）に設定すると、前回の非アクティブ化の要求前にアクセスがあったかどうかにかかわらず、セッションの状態が保存されます。プロパティを false に設定すると、コンテナでは単純な非アクティブ化だけが適用されます。なお、このプロパティは、次の場所にあるコンテナのプロパティファイル `partition.xml` に設定されます。

```
<install_dir>/var/domains/base/configurations/<configuration_name>
/ mos/<partition_name>/adm/properties
```

積極的な非アクティブ化を使用すると、フェイルオーバー面で有利ですが、コンテナからデータベースをアクセスする頻度が高くなり、パフォーマンスは低下します。ネイティブでないデータベースを使用するよう JSS を設定すると（つまり JDataStore を使用しない設定）、パフォーマンスの低下はさらに顕著になります。積極的な非アクティブ化の使用は、可用性とパフォーマンスのバランスを考慮して決めてください。

二次ストレージのセッション

タイムアウトになると、ほとんどのセッションは、永続的ストレージに永続的に保存されます。Borland では、データベースに保存したセッションで、**有効期限**で設定した時間を過ぎたものはデータベースから削除するメカニズムを用意しました。有効期限は、非アクティブ化になったセッションをステートフルストレージに保存する最短時間を秒数で指定したものです。未使用のセッションのためにデータベースを定期的にポーリングするのはパフォーマンス面で無駄なので、データベースに保存される実際の設定時間は状況に応じて設定します。セッションが持続する時間は、最低を有効期限の値とし、最高を有効期限の倍とします。

先に紹介したほかの非アクティブ化プロパティとは違って、有効期限の値の適用範囲には、パーティション規模とセッション Bean 単位のどちらでも設定できます。特定の Bean に有効期限を設定すると、コンテナ規模の値に優先します。有効期限を設定しない Bean があると、その Bean にはパーティション規模の値が適用されます。

コンテナでの有効期限の設定

Borland JSS インプリメンテーションでは、プロパティ `ejb.sfsb.keep_alive_timeout` で、非アクティブ化セッションをステートフルストレージに保存する時間を指定します。デフォルト値は 86,400 秒、または 24 時間です。先に紹介したほかのプロパティと同様に、有効期限はコンテナプロパティファイルに設定します。

```
<install_dir>/var/domains/base/configurations/<configuration_name>
/ mos/<partition_name>/adm/properties
```

ここで指定する値は、いずれも特定のセッション Bean に対する有効期限の設定値で上書きできます。

特定のセッション Bean に対する有効期限の設定

コンテナで管理する特定のセッション Bean に対する非アクティブ状態の保存時間を、ほかの Bean より長く、あるいは短く設定することができます。特定の Bean の有効期限は、`ejb-borland.xml` ファイルの **<timeout>** で指定できます。セッション Bean の DTD 要素にこの要素があります。

```
<!ELEMENT session (ejb-name, bean-home-name?, bean-local-home-name?, timeout?,
ejb-ref*, elb-local-ref*, resource-ref*, resource-env-ref*, property*)>
```

たとえば、ここに `personInfo` という名前のシンプルなステートフルセッション Bean があり、シンプルなメッセージフォーラムの個人情報が収集されているとします。積極的な非アクティブ化を適用せずにこのセッションの可用性を高めることにしました。非アクティブ状態にしたとしても、数分を超えてデータベースに保存する必要性はほとんどありません。ほかのセッション Bean は、非アクティブ状態にしたら、これらの Bean よりは長くストレージに保存しなければなりません。そこで、Borland 固有のデプロイメントデスクリプタを Bean の JAR に使用してより短い時間、たとえば 300 秒（5 分）に設定します。`ejb-borland.xml` デプロイメントデスクリプタに、次のように設定しました。

```
<ejb-jar>
 <enterprise-beans>
  <session>
   <ejb-name>personInfo</ejb-name>
   <timeout>300</timeout>
  </session>
 </enterprise-beans>
</ejb-jar>
```

この値は、`ejbcontainer.properties` ファイルで指定した値より優先しますが、ほかのサービス対象のセッションはそのファイルで指定したデフォルト値を使用します。

第 13 章

Borland AppServer のエンティティ Bean と CMP 1.1

ここでは、Borland AppServer（AppServer）にエンティティ Bean をデプロイメントする方法と永続性を管理する方法について説明します。ただし、これはエンティティ Bean そのものの入門書ではないのでそのようにお読みください。というより、ここでは Borland パーティション内におけるエンティティ Bean の使用時の背景説明が主になります。また、デスクリプタ、永続性オプション、その他コンテナの最適化について解説します。コンテナ管理永続性（CMP）の Borland 固有のデプロイメントデスクリプタとインプリメンテーションについては、一般に Sun Microsystems の J2EE 仕様から入手できる EJB 情報を優先して解説します。

エンティティ Bean

エンティティ Bean はデータベースに保存されるデータのビューを表します。エンティティ Bean は、エンティティ Bean とテーブル行が 1 対 1 の対応で、1 つのテーブルにマッピングされた細粒度エンティティの場合もあります。あるいは、複数のテーブルにまたがり、基底のデータベーススキーマとは無関係に存在するデータを表す場合もあります。エンティティ Bean どうしは相互に関係を持ち、クライアントから照会でき、さまざまなクライアント間で共有することができます。

AppServer パーティションのいずれかにエンティティ Bean をデプロイメントするには、JAR の一部としてパッケージにしておく必要があります。JAR には、`ejb-jar.xml` ファイルと独自の `ejb-borland.xml` ファイルの 2 つのデスクリプタを組み込みます。`ejb-jar.xml` デスクリプタについては、Sun Java Center を参照してください。このマニュアルでは `ejb-borland.xml` の DTD を転載しており、合わせてその使用方法についても紹介します。Borland 専用デスクリプタには、多くのプロパティが組み込まれており、その設定いかんで、コンテナパフォーマンスを最適化したり、エンティティ Bean の永続性を管理することができます。

コンテナ管理の永続性と関係

Borland の EJB コンテナには、エンティティ Bean をデプロイメントするときに、つまりエンティティ Bean をパーティションにインストールするときにデータベースアクセス呼び出しを生成するツールが組み込まれています。これらのツールはデプロイメントデスクリプタを使用します。これは、データベースアクセス呼び出しをどのインスタンスフィールドに生成すればよいかを判断するためです。この場合、データベースアクセスを Bean に直接コーディングすることはありません。コンテナツールでアクセス呼び出しを生成する対象となるインスタンスフィールドをコンテナ管理のエンティティ Bean の Bean プロバイダはデプロイメントデスクリプタで指定します。EJB コンテナには、エンティティ Bean のフィールドをデータソースにマッピングする先進的なデプロイメントツールが備わっています。

コンテナ管理の永続性には、Bean 管理の永続性に比べて多くの長所があります。コンテナ管理の永続性を使用すると、Bean プロバイダがデータベースアクセス呼び出しをコーディングする必要がないため、コーディングが簡単です。永続性の処理方法を変更する場合も、エンティティ Bean のコードを変更して再コンパイルする必要がありません。デプロイヤやアプリケーションアセンブラでエンティティ Bean のデプロイメント時にデプロイメントデスクリプタを変更することで、永続性の処理方法を変更できます。このようにデータベースアクセスと永続性の管理を EJB コンテナに任せると、Bean のコードを単純化でき、発生するエラーの範囲を狭めることができます。また、Bean プロバイダは基底のシステム関連の問題にとらわれずに Bean のビジネスロジックに集中できます。

EJB 2.0 仕様では、コンテナ管理の永続性を使用するエンティティ Bean どうしが、コンテナ管理の関係を持つことができます。コンテナは自動的に Bean 関係を管理し、これらの Bean 関係の参照の整合性を維持します。これは、Bean のリモートインターフェースによって Bean のインスタンス状態をエクスポーズすることしかできなかった EJB 1.1 仕様とは異なります。

EJB 2.0 仕様を使用すると、コンテナ管理の永続性フィールドを定義したように、コンテナ管理の関係フィールドも Bean のデプロイメントデスクリプタを定義できます。コンテナは、1 対 1 の関係と 1 対多、多対多など、さまざまなカーディナリティをサポートします。

エンティティ Bean の実装

エンティティ Bean の実装は、EJB 1.1 仕様と EJB 2.0 仕様の規則にしたがっています。ホームインターフェース、リモートインターフェースまたはローカルインターフェース（2.0 コンテナ管理の永続性を使用する場合）、およびエンティティ Bean インプリメンテーションクラスを実装する必要があります。エンティティ Bean クラスでは、リモートインターフェースまたはローカルインターフェースと、ホームインターフェースで宣言したメソッドに対応するメソッドを実装します。

パッケージ要件

セッション Bean のように、エンティティ Bean はリモートインターフェースやローカルインターフェースでのメソッドをエクスポーズします。エンティティ Bean は JAR のデプロイメントデスクリプタにも対応するエントリを持っています。標準デプロイメントデスクリプタ `ejb-jar.xml` には、原則として 3 種類のデプロイメント情報が収められます。次にそれらのデプロイメント情報について説明します。

1. **一般 Bean 情報：**これは、デスクリプタファイルにある `<enterprise-beans>` 要素に対応しており、3 種類すべての Bean に使用します。この情報には、Bean のインターフェースとクラス、セキュリティ情報、環境情報、さらには照会宣言まで含まれています。
2. **関係：**これは、デスクリプタファイルにある `<relationships>` 要素に対応しており、CMP だけを使用するエンティティ Bean に適用されます。ここに、コンテナ管理の関係を記述します。
3. **アセンブリ情報：**これは、デスクリプタファイルにある `<assembly-descriptor>` 要素に対応しており、Bean とアプリケーション間の関係を全体的に説明します。アセンブリ情報は 4 つのカテゴリに分類できます。
   - セキュリティロール：アプリケーションが使用するセキュリティロールの単純な定義。ユーザーが Bean に定義するセキュリティロールリファレンスも定義する必要があります。
   - メソッド許可：各 Bean のメソッドには、それぞれに実行に関する一定の規則が適用されます。規則はここで設定します。
   - コンテナトランザクション：トランザクションに関連するメソッドごとに、EJB 2.0 仕様にしたがってトランザクション属性を指定します。
   - 除外リスト：呼び出される相手がいないメソッド。

以上は、いずれもデプロイメントデスクリプタエディタからアクセスできます。DTD 情報とデスクリプタファイルの正しい使用方法については、EJB 2.0 仕様を参照してください。

エンティティ Bean の主キー

各エンティティ Bean には、Bean インスタンスを識別する一意の主キーを割り当てます。主キーは、RMI-IIOP で有効な値の型を備えた Java クラスで表すことができます。したがって、主キーは `java.io.Serializable` インターフェースを拡張します。また、主キーは、`Object.equals(Object other)` および `Object.hashCode()` メソッドのインプリメンテーションも提供する必要があります。

通常は、エンティティ Bean の主キーフィールドは、`ejbCreate()` メソッド内で設定します。これらのフィールドは、データベースに新しいレコードを挿入するときに使用します。ただし操作が難しく、メソッドも肥大化します。したがって、データベースの多くは、今では内蔵メカニズムによって適切な主キーの値を提供するようになっています。主キーの生成方法としてより洗練された方法には、主キーを生成するクラスをユーザーに別途実装する方法があります。このクラスでは、主キーを生成するためのデータベース固有のプログラミングロジックも生成できます。

ユーザークラスから主キークラスを生成

エンタープライズ Bean により、一意のデータを持つ Java クラスで主キーが表されます。この主キークラスは、RMI-IIOP の有効な値型であればクラスは問いません。したがって、主キークラスは `java.io.Serializable` インターフェースを拡張します。また、主キーは `Object.equals(Object other)` メソッドと `Object.hashCode()` メソッドのインプリメンテーションも提供します。この 2 つのメソッドは、当然ながらすべての Java クラスが継承します。

主キークラスは、特定のエンティティ Bean クラスでのみ使用される場合があります。言い換えると、各エンティティ Bean は、専用の主キークラスを定義する場合があります。これに対して、複数のエンティティ Bean が同じ主キークラスを共有する場合もあります。

bank アプリケーションでは、普通預金と当座預金の口座を表現するために、2 種類のエンティティ Bean を使用します。どちらの口座も、同じフィールドで口座レコードを一意に識別します。この場合、どちらの口座も同じ主キークラス AccountPK で、両方の口座の一意の識別子を表します。次のコードは、口座主キークラスの定義です。

```
public class AccountPK implements java.io.Serializable {
   public String name;
   public AccountPK() {}
   public AccountPK(String name) {
      this.name = name;
   }
}
```

カスタムクラスから主キークラスを生成

カスタムクラスから主キーを生成するには、`com.borland.ejb.pm.PrimaryKeyGenerationListener` インターフェースを実装するクラスを作成します。

複合キーのサポート

主キーは 1 列とは限りません。複数の列で構成されることもあります。たとえば、講座は単なる講座名だけでは識別できません。講座が設けられている学科と講座番号自体が各講座レコードの主キーになります。学科コードと講座番号は、Course テーブル内の別々の列です。特定の講座、またはある学生が登録されているすべての講座を取り出す select 文では、主キー全体を使用する必要があります。つまり、両方の列の主キーを考慮する必要があります。

Borland CMP エンジンは複合主キーをサポートします。select 文の where 節では、複数の列からなるキーを使用できます。select 文の select 節では、複合キーのすべてのフィールドを選択できます。

where 節では、単一フィールドの名前を指定する場合と同じ方法で、複数フィールドの名前を指定します。各フィールドは「and」で区切ります。次の形式を使用します。

```
<column> = :<parameter>[ejb/<entity bean>]
```

統合（=）は、有効な記号の 1 つです。このほか、より大きい（>）、より小さい（<）、以上（>=）、および以下（<=）を使用できます。コロン（:）はパラメータ置換を表します。パラメータフィールドを指定する場合は、最初に Bean 名、次にドット（.）、最後に Bean 属性を続けます。

たとえば、Art 205, Renaissance Art クラスに登録しているすべての学生を検索するとします。この講座は、学科名（Art）と講座番号（205）で識別します。この場合、検索メソッド `findByCourse()` に対して次の select 文を定義します。

```
SELECT sname FROM Enrollment WHERE course_department = :c.department[ejb/
Course] AND
   course_number = :c.number[ejb/Course]
```

select 文で、複合キーの複数のフィールドを返すこともできます。その場合は、select 文の select 節で、複数のフィールドをカンマで区切って指定します。パラメータと同様にドット表記を使用する必要があります。つまり、最初にエンティティ Bean 名、次にドット（.）、最後に属性を指定します。たとえば、検索メソッド `findByStudent()` に次の select 文を指定します。

```
SELECT c.department, c.number FROM Entrollment WHERE student_name = :s
```

リエントラント

デフォルトでは、エンティティ Bean はリエントラントではありません。同じトランザクションコンテキスト内でエンティティ Bean に呼び出しが到着すると、例外 `java.rmi.RemoteException` が生成されます。

デプロイメントデスクリプタの中で、エンティティ Bean をリエントラントとして宣言できます。ただし、その場合は注意が必要です。通常、コンテナは、同一トランザクション内でのループバック呼び出しと、同一トランザクションコンテキスト内での同一エンティティ Bean に対する同時呼び出しを区別できません。

エンティティ Bean をリエントラントとしてマークした場合、その Bean インスタンスに対して同一トランザクションコンテキスト内で同時呼び出しを行うことはできません。プログラマはこの規則を厳守してください。

AppServer におけるコンテナ管理の永続性

AppServer の EJB コンテナは、J2EE 1.3 完全準拠です。Bean プロバイダは、エンティティ Bean に永続性スキーマを設計し、コンテナ管理のフィールドと関係をアクセスするメソッドを決定し、これらのメソッドを Bean のデプロイメントデスクリプタに定義します。デプロイヤは、この永続性スキーマをデータベースにマッピングし、Bean のメンテナンスに必要なほかのクラスを作成します。

J2EE 1.3 エンティティ Bean と CMP 2.0 については、131 ページの「CMP 2.x の AppServer プロパティの使い方」を参照してください。

AppServer CMP エンジンの CMP 1.1 インプリメンテーション

Borland CMP エンジンのすべての面に精通していなければ、これを有効に使用できないというわけではありませんが、一定の領域である程度の知識があると役立ちます。ここでは、CMP エンジンを利用する上で理解しておくとよい分野について説明します。特に、デプロイメントデスクリプタファイルと、そのファイル内の XML 文を重点的に扱います。

ただし、1.1 コンテナ管理の永続性を持つエンティティ Bean のインプリメンテーションでは、次の点に注意する必要があります。

- このエンティティ Bean は検索メソッドを実装していません。コンテナ管理の永続性を持つエンティティ Bean の検索メソッドのインプリメンテーションは、EJB コンテナが提供します。したがって、エンティティ Bean クラスで検索メソッドを実装するかわりに、検索メソッドの実装方法をデプロイメントデスクリプタに記述して、コンテナに通知します。
- エンティティ Bean では、コンテナに管理させるすべてのフィールドを `public` として宣言します。CheckingAccount Bean では、`name` と `balance` が public フィールドとして宣言されています。
- エンティティ Bean クラスは、`EntityBean` インターフェースで宣言される 7 つのメソッド（`ejbActivate()`、`ebjPassivate()`、`ejbLoad()`、`ejbStore()`、`ejbRemove()`、`setEntityContext()`、`unsetEntityContext()`）を実装します。エンティティ Bean では、少なくともこれらのメソッドのインプリメンテーションスケルトンを提供する必要があり、さらに必要に応じてアプリケーション固有のコードを任意の場所に追加します。CheckingAccount Bean は、`setEntityContext()` が返したコンテキストを保存し、`unsetEntityContext()` でそのリファレンスを解放します。この点を除き、CheckingAccount Bean では、`EntityBean` インターフェースのメソッドにコードは追加されていません。
- このエンティティ Bean には、Bean の呼び出し元が新しい当座預金口座を作成できるように、`ejbCreate()` メソッドが実装されています。このインプリメンテーションは、引数にしたがって、インスタンスの口座名と預金残高を表す 2 つの変数を初期化します。コンテナ管理の永続性を使用する場合、クライアントに返すリファレンスはコンテナが作成するので、`ejbCreate()` メソッドは `null` 値を返します。
- `ejbPostCreate()` メソッドは、必要であればより高度な初期化作業を行うこともできますが、このエンティティ Bean は、`ejbPostCreate()` メソッドの最小限のインプリメンテーションの内容だけを提供します。これは、コンテナ管理の永続性を持つ

Bean の場合は、`ejbPostCreate()` がコールバック通知の役割を果たすためであり、`ejbPostCreate()` メソッドは最小限のインプリメンテーションで十分です。`EntityBean` インターフェースから継承されるメソッドについても同じことが言えます。

CMP メタデータのコンテナへの提供

EJB 仕様によって、デプロイヤは CMP のメタデータを EJB コンテナに提供する必要があります。Borland コンテナは、CMP 関連のメタデータを XML デプロイメントデスクリプタから取得します。具体的には、デプロイメントデスクリプタ内のベンダー固有の部分を利用して、メタデータを見つけます。

ここでは、特にコマンドラインレベルでコンテナ管理の検索メソッドを生成する場合に、そのメソッドのために提供する必要がある情報のいくつかを具体的に説明します。構文の詳細については、デプロイメントデスクリプタの DTD を調べることも役立ちます。検索メソッドと OR（Object-Relation）マッピングメタデータの構文を参照してください。

検索メソッドの生成

検索メソッドを生成する場合、実際には、where 節を持つ SQL select 文を生成することになります。select 文には、どのレコードまたはデータを検索して返すかを指定する節があります。たとえば、銀行の預金口座を検索して返すとします。select 文の where 節は、選択操作に対して制限事項を設定します。たとえば、指定した額より大きい残高のある口座や、月間に一定額以上の出し入れがあった口座だけを検索します。コンテナがコンテナ管理の永続性を使用する場合、デプロイメントデスクリプタで where 節の条件を指定する必要があります。

たとえば、`findAccountsLargerThan(int balance)` という名前の検索メソッドがあり、コンテナ管理の永続性を使用しているとします。この検索メソッドは、指定された値より大きい残高のあるすべての口座を検索します。コンテナがこの検索メソッドを実行した場合、実際には、このメソッドにパラメータとして渡された `int` 値と、口座残高を比較する where 節を持つ select 文を実行します。コンテナ管理の永続性を使用しているため、デプロイメントデスクリプタで where 節の条件を指定する必要があります。指定しなければ、完全な select 文を生成する方法がコンテナには伝わりません。

`findAccountsLargerThan(int balance)` メソッドの where 節の値は、「`balance > :balance`」です。これは、「balance 列の値は、`balance` というパラメータの値より大きい」という意味です（この検索メソッドの引数は、`int` 値だけです）。

コンテナ管理の永続性のデフォルトのインプリメンテーションは、次のような完全な SQL select 文を生成することにより、この検索メソッドをサポートします。

```
select * from Accounts where ? > balance
```

次に、CMP エンジンは「?」に `int` パラメータを代入します。最後に、エンジンは結果セットを主キーの `Enumeration` または `Collection` に変換します。これは、EJB 仕様で要求されているとおりです。

CMP インプリメンテーションが生成するさまざまな SQL 文を確認することができます。それには、コンテナの `EJBDebug` フラグを有効にします。そのフラグを有効にすると、コンテナによって生成されたとおりの SQL 文を出力します。

ほかの EJB コンテナ製品が CMP をサポートするためにコードを生成するのに対して、Borland コンテナ製品はコードの生成を使用しません。コードの生成には大きな限界があるからです。たとえば、コードを生成する方法では、「調整した更新」機能をサポートすることが困難になります。なぜなら、コンテナ管理のフィールドに対して異なる update 文が大量に必要になるからです。

where 節の生成

where 節は、取得するレコードの範囲を限定する場合に必要な select 文の一部です。where 節の構文はかなり複雑になることがあり、EJB コンテナがこの節を正しく生成できるように、XML デプロイメントデスクリプタファイルでは、一定の規則にしたがう必要があります。

まず、必ずしも <where-clause> で "where" リテラルを使用する必要はありません。このリテラルのない where 節を生成し、where 節記入はコンテナに任せることができます。ただし、コンテナは、<where-clause> 内が空文字列でない場合にだけ、これを行います。空文字列は空のままになります。たとえば、次のどちらの方法でも同じ where 節を定義できます。

```
<where-clause> where a = b </where-clause>
```

または

```
<where-clause> a = b </where-clause>
```

コンテナは、a = b を同じ where 節の where a = b に変換します。ただし、<where-clause> "" </where-clause> と定義されている空文字列は、変更されません。

メモ 空文字列を使用すると、簡単に findAll() メソッドを指定できます。空文字列だけを指定した場合、コンテナは、次のように文を生成します。

```
select [values] from [table];
```

このような select 文は、特定のテーブルのすべての値を返します。

パラメータ置換

パラメータ置換は、where 節の重要な部分です。Borland EJB コンテナは、標準の SQL 置換プレフィクスであるコロン（:）を見つけると、パラメータ置換を行います。置換される各パラメータは、XML デプロイメントデスクリプタにある検索メソッド仕様の中のパラメータの名前に対応します。

たとえば、XML デプロイメントデスクリプタで、パラメータ balance を受け取る次の検索メソッドを定義します。balance の前にコロンがあることに注意してください。

```
<finder>
   <method-signature>findAccountsLargerThan(float balance)</method-signature>
   <where-clause>balance > :balance</where-clause>
</finder>
```

コンテナは、次の where 節を持つ SQL select 文を生成します。

```
balance > ?
```

デプロイメントデスクリプタ内の :balance パラメータは、対応する SQL 文では疑問符（?）になることがわかります。呼び出し時に、コンテナはパラメータ :balance の値を where 節の ? に代入します。

複合パラメータ

コンテナは、複合パラメータもサポートします。複合パラメータは、テーブル名の後に、そのテーブル内の列が付きます。複合パラメータには、標準のドット（.）構文を使用します。この構文では、テーブル名と列名をドットで区切ります。複合パラメータでも、前にコロンを付けます。

たとえば、次の検索メソッドには複合パラメータ :address.city と :address.state があります。

```
<finder>
   <method-signature>findByCity(Address address)</method-signature>
   <where-clause>city = :address.city AND state = :address.state</where-clause>
</finder>
```

where 節は、address 複合オブジェクトの city フィールドと state フィールドを使用して、特定のレコードを選択します。基底の Address フィールドオブジェクトには、属性 city フィールドと state フィールドに対応する JavaBeans 形式のゲッターメソッドがあります。または、ゲッターメソッドのかわりに、それぞれの属性に対応する public フィールドを指定できます。

パラメータとなるエンティティ Bean

エンティティ Bean は、検索メソッドのパラメータとしても使用できます。エンティティ Bean は複合型として使用できます。その場合、SQL クエリーに渡すエンティティ Bean リファレンスのどのフィールドを使用するのかを CMP エンジンに指示する必要があります。複合型としてエンティティ Bean を使用しなければ、コンテナは、`where` 節にその Bean の主キーを代入します。

たとえば、`Order` エンティティオブジェクトに関連付けられている一連の `OrderItems` エンティティ Bean があるとします。次の検索メソッドを指定できます。

```
java.util.Collection OrderItemHome.findByOrder(Order order);
```

このメソッドは、特定の `Order` に関連付けられたすべての `OrderItems` を返します。その `where` 節に対応するデプロイメントデスクリプタのエントリは次のようになります。

```
<finder>
   <method-signature>findByOrder(Order order)</method-signature>
   <where-clause>order_id = :order[ejb/orders]</where-clause>
</finder>
```

この where 節を生成するために、コンテナは文字列 `:order[ejb/orders]` に `Order` オブジェクトの主キーを代入します。ブラケット内の文字列（この例では、`ejb/orders`）は、パラメータの型のホームに対応する `<ejb-ref>` です。この例では、`ejb/orders` は、`OrderHome` を宛先とする `<ejb-ref>` に対応します。

EJBObject を複合型として使用する（ドットを使用）場合、実際には、`<finder>` 定義内のフィールドに対応する基底のゲッターメソッドにアクセスします。たとえば、`<finder>` 定義の次のコマンドは、

```
order_id = :order.orderId
```

`EJBObject` の `getOrderId()` メソッドを呼び出し、その結果を選択条件の中で使用します。

エンティティ間の関係の指定

RDBMS（リレーショナルデータベース）では、あるテーブルのレコードを別のテーブルのレコードに関連付けることができます。そのために、RDBMS では、外部キーを使用します。つまり、一方のテーブルのレコードにある 1 つのフィールド（列）に、別のテーブルにある関連するレコードの外部キー、または主キーのリファレンスを保持します。エンティティ Bean 間にも同様のリファレンスをマッピングできます。

CMP エンジンでエンティティ Bean 間のリファレンスをマッピングするには、デプロイメントデスクリプタの `<ejb-link>` エントリを使用します。`<ejb-link>` は、対応するエンティティにフィールド名をマッピングします。CMP エンジンは、デプロイメントデスクリプタのこの情報で、特定のフィールドに関連するエンティティを検索します。`<ejb-link>` エントリの具体例については、pigs サンプルを参照してください。

コンテナ管理の永続性フィールドは、対応するテーブル内の外部キーフィールドに対応付けられます。エンティティ Bean のコードを見ると、これらの外部キー CMP フィールドは、オブジェクトリファレンスとして表示されます。

たとえば、`address` と `country` の 2 つのデータベーステーブルがあるとします。`address` テーブルには `country` テーブルへのリファレンスがあります。これらのテーブルの SQL `create` 文は、次のようになります。

```
create table address (
   addr_id         number(10),
   addr_street1    varchar2(40),
   addr_street2    varchar(40),
   addr_city       varchar(30),
   addr_state       varchar(20),
   addr_zip         varchar(10),
   addr_co_id       number(4)         * foreign key *
);
create table country (
   co_id            number(4),
   co_name          varchar2(50),
   co_exchange      number(8, 2),
   co_currency      varchar2(10)
);
```

address テーブルには addr_co_id フィールドがあり、これは country テーブルの主キーフィールド co_id を参照する外部キーです。

これらのテーブルに対応するエンティティを表すクラスには、Address クラスと Country クラスの 2 つがあります。Address テーブルには、Country テーブルへの直接ポインタ country があります。この直接的なポインタリファレンスは EJBObject リファレンスであり、インプリメンテーション Bean を指す直接的な Java リファレンスではありません。

では、以上 2 つのクラスについて、次のコードで確認してみます。

```
//Address クラス
public class Address extends EntityBean {
   public int id;
   public String street1;
   public String street1;
   public String city;
   public String state;
   public String zip;
   public Country country; // これは、Country への直接的なポインタです
}
//Country クラス
public class Country extends EntityBean {
   public int id;
   public String name;
   public int exchange;
   public String currency;
}
```

Address クラスから Country クラスのリファレンスをコンテナで解決するには、デプロイメントデスクリプタで Country クラスに関する情報を指定する必要があります。デプロイメントデスクリプタの <ejb-link> エントリを使用して、フィールド Address.country までのリファレンスをホームオブジェクト CountryHome の JNDI 名にリンクするようにコンテナに指示します。詳細については、pigs サンプルを参照してください。コンテナは、このエンティティ間リファレンスを最適化します。この最適化のため、エンティティ間リファレンスを使用しても、外部キーの値を格納する場合と速度は変わりません。

ただし、エンティティ間リファレンスを使用する場合と、外部キー値を格納する場合では、重要な違いが 2 つあります。

- 別のエンティティまでのクロスリファレンスポインタを使用するとき、エンティティのホームオブジェクト findByPrimaryKey() メソッドを呼び出さなくても、そのオブジェクトエンティティを取得できます。上の例でわかるように、Country オブジェクトを宛先とする Address.country ポインタで Country オブジェクトを直接取得しています。Country id に対応する Country オブジェクトを取得するには、必ずしも CountryHome.findByPrimaryKey(address.country) を呼び出す必要はありません。
- クロスリファレンスポインタの使用時には、参照される側のエンティティの状態は、実際に使用するときに初めてロードされます。ポインタを収めたエンティティをロードしても、参照される側のエンティティの状態が自動的にロードされるわけではありません。つまり、Address オブジェクトをロードするだけでは、実際には Country オブジェクトはロードされません。Address.country フィールドは、「怠惰な」リファレンスとみなすことができます。つまり、基底のオブジェクトが実際に使用された場合にだけ、「怠惰な」リファレンスは対応する状態をロードします。この「怠惰な」動作は EJB モデルの一部であることに注意してください。EJB モデルのこのような特徴から、Address.country と、Address Bean インスタンス自体の存続期間は、それぞれ独立しています。このモデルでは、Address.country は通常のエンティティ EJBObject リファレンスです。したがって、Address.country の状態は、それが使用されたときにだけロードされます。コンテナは、EJB モデルにしたがって、ほかの EJBObject と同様に AddressBean.country の状態を制御します。

コンテナ管理のフィールドの名前

Borland コンテナでは、コンテナ管理の永続性のフィールド名が、Java 向きに変更されています。SQL の列名は、各列の名前の前にテーブル名の短縮形とアンダースコアが付くのが普通です。たとえば、address テーブルには、addr_city という名前の都市の列があります。この列の完全なリファレンスは address.addr_city になります。Borland コンテナの場合、この列は、冗長で不便な Address.city ではなく、Java フィールド Address.addr_city にマッピングされます。

この Java 向きの列名とフィールド名のマッピングを行うには、デプロイメントデスクリプタを使用します。ここではデプロイメントデスクリプタを手動で編集する方法について説明しますが、この作業にはデプロイメントデスクリプタエディタの GUI を使用するのが最もよい方法です。GUI 画面を使用する手順については、『*管理コンソールユーザーズガイド*』の「デプロイメントデスクリプタエディタの使い方」を参照してください。

デプロイメントデスクリプタを手動で編集する場合は、<env-entry> タグ内の <env-entry-name>、<env-entry-type>、および <env-entry-value> サブタグを使用します。<env-entry-name> タグ内に Java 向きのフィールド名を入力します。それが JDBC 列を参照していることを確認してください。<env-entry-type> タグにフィールドの型を入力します。最後に、<env-entry-value> タグに、実際の SQL 列名を入力します。次に、デプロイメントデスクリプタコードの具体例を示します。

```
<env-entry>
   <env-entry-name>ejb.cmp.jdbc.column:city</env-entry-name>
   <env-entry-type>String</env-entry-type>
   <env-entry-value>addr_city</env-entry-value>
</env-entry>
```

プロパティの設定

Enterprise JavaBeans のほとんどのプロパティは、デプロイメントデスクリプタで設定できます。Borland デプロイメントデスクリプタエディタ（DDEditor）では、プロパティの設定やデスクリプタファイルの編集ができます。デプロイメントデスクリプタエディタの使用方法については、Borland AppServer の『*ユーザーズガイド*』を参照してください。デプロイメントデスクリプタのプロパティでは、エンティティ Bean のインターフェース、トランザクション属性などのほか、エンティティ Bean 固有の情報に関する情報を指定します。エンティティ Bean の一般的なデスクリプタ情報以外に、CMP インプリメンテーションをカスタマイズするために設定する 3 セットのプロパティがあります。それが、エンティティプロパティ、テーブルプロパティ、列プロパティです。エンティティプロパティは、［EJB Designer］で設定するか、XML で直接設定します。

デプロイメントデスクリプタエディタの使い方

Borland AppServer Edition に付属のデプロイメントデスクリプタエディタでは、コンテナ管理の永続性に関するすべての情報を設定できます。デプロイメントデスクリプタと関連ツールの使用方法の詳細については、『*管理コンソールユーザーズガイド*』を参照してください。

BMP または CMP 1.1 を使用する J2EE 1.2 エンティティ Bean

デスクリプタ要素	ナビゲーションツリーノード／パネル名	DDEditor のタブ
エンティティ Bean 名	Bean	［General］
エンティティ Bean クラス	Bean	［General］
ホームインターフェース	Bean	［General］
リモートインターフェース	Bean	［General］
JNDI 名	Bean	［General］
永続性タイプ（CMP または BMP）	Bean	［General］
主キークラス	Bean	［General］
リエントラント	Bean	［General］
アイコン	Bean	［General］
環境エントリ	Bean	［Environment］
その他の Bean への EJB リファレンス	Bean	［EJB References］
EJB リンク	Bean	［EJB References］
データオブジェクト／接続ファクトリへのリソースリファレンス	Bean	［Resource References］
リソースリファレンスの種類	Bean	［Resource References］
リソースリファレンス認証の種類	Bean	［Resource References］
セキュリティロールリファレンス	Bean	［Security Role References］
エンティティプロパティ	Bean	［Properties］
コンテナトランザクション	Bean：［Container Transactions］	［Container Transactions］
トランザクションメソッド	Bean：［Container Transactions］	［Container Transactions］
トランザクションメソッドインターフェース	Bean：［Container Transactions］	［Container Transactions］
トランザクション属性	Bean：［Container Transactions］	［Container Transactions］
メソッド許可	Bean：［Method Permissions］	［Method Permissions］
CMP 説明	Bean：CMP1.1	CMP 1.1
CMP テーブル	Bean：CMP1.1	CMP 1.1
コンテナ管理フィールドの説明	Bean：CMP1.1	CMP 1.1
ファインダ	Bean：CMP1.1	［Finders］
ファインダメソッド	Bean：CMP1.1	［Finders］
ファインダ WHERE 節	Bean：CMP1.1	［Finders］
ファインダ［Load State］オプション	Bean：CMP1.1	［Finders］

コンテナ管理データアクセスサポート

コンテナ管理の永続性には、Borland EJB コンテナがサポートするデータ型は、JDBC 仕様によってサポートされているデータ型のほか、JDBC でサポートされていないデータ型がいくつかあります。

次の表に、Borland EJB コンテナがサポートする基本型と複合型をまとめます。

- 基本型
  - `boolean Boolean`
  - `double Double`
  - `long Long`
  - `BigDecimaljava.util.Date`
  - `byte Byte`
  - `float Float`
  - `short Short`
  - `byte[]`
  - `char Character`
  - `int Integer`
  - `String java.sql.Date`
  - `java.sql.Time java.sql.TimeStamp`
- 複合型
  - `java.io.Serializable` を実装する任意のクラス（Vector や Hashtable など）
  - その他のエンティティ Bean リファレンス

Borland コンテナは、`java.io.Serializable` インターフェースを実装するクラス（Hashtable や Vector など）をサポートすることに注意してください。コンテナでは、Java コレクションやサードパーティコレクションなどもサポートしています。これらも `java.io.Serializable` を実装するからです。`Serializable` インターフェースを実装するクラスとデータ型に対して、Borland コンテナは、単にそれらの状態をシリアライゼーションし、その結果を `BLOB` に格納するだけです。Borland コンテナでは、これらのクラスや型に対して何らかのマッピングを行うことはなく、単にバイナリ形式でそれらの状態を格納します。Borland コンテナの CMP エンジンには、明示的にサポートされていないすべての型を `BLOB` としてシリアライズするという規則が適用されます。

そのことから、`BLOB` は、`LONGVARBINARY` がマッピングする型であるという JDBC 仕様にしたがいます。Oracle の場合、これは、`LONG RAW` です。

SQL キーワードの使用

Borland コンテナの CMP エンジンは、SQL92 標準に準拠するすべての SQL キーワードを処理できます。ただし、ベンダーによって独自のキーワードが追加されることはめずらしくないので注意してください。たとえば、Oracle はキーワード `VARCHAR2` を使用します。SQL 標準とは異なるベンダーのキーワードを CMP エンジンで確実に処理するには、デプロイメントデスクリプタで、CMP フィールド名を列名にマッピングするための環境プロパティを設定します。この種の環境プロパティを使用すれば、コードを変更する必要はありません。

たとえば、「select」という名前の CMP フィールドがあるとします。次に示すように、環境プロパティを使って「select」を「SLCT」という名前の列にマッピングできます。

```
<cmp-info>
         <database-map>
            <table>Data</table>
            <column-map>
               <field-name>select</field-name>
               <column-name>SLCT</column-name>
               </column-map>
            </database-map>
</cmp-info>
```

null 値の使い方

データベースの値が SQL null 値の場合があります。その場合は、Java データ型で Java null 値を保持できるようなフィールドに、それらの値をマッピングする必要があります。それには、通常、プリミティブ型ではなく Java 型を使用します。たとえば、プリミティブ `int` 型ではなく Java `Integer` 型をプリミティブ `float` 型ではなく Java `Float` 型を使用します。

データベース接続の確立

CMP エンジンでデータベース接続を開くには、DataSource を指定する必要があります。DataSource は、ユーザー名やパスワードなど、データベース接続の確立に必要な情報を定義します。DataSource を定義したら、Bean の XML デプロイメントデスクリプタの `resource-ref` で DataSource を参照します。これで CMP エンジンは、その DataSource を使用し、JDBC を介してデータベースにアクセスできます。

ベンダー固有の XML ファイルでは、`resource-ref` に `jndi` バインディングを指定する場所に、次の要素を追加します。

```
<cmp-resource>True</cmp-resource>
```

エンティティ Bean が `resource-ref` を 1 つだけ宣言している場合は、上の XML 要素を指定する必要はありません。エンティティ Bean に `resource-ref` が 1 つしかない場合、Borland コンテナは、その 1 つのリソースを `cmp-resource` として自動的に選択します。

コンテナによって作成されるテーブル

エンティティのコンテナ管理のフィールドに基づき、自動的にコンテナ管理のエンティティに対応するテーブルを作成するように Borland EJB コンテナに指示できます。テーブルの作成とデータ型のマッピングはベンダーによって異なるため、デプロイメントデスクリプタでコンテナに JDBC データベースダイアレクトを指定する必要があります。JDataStore 以外のデータベースでは、ダイアレクトを指定すると、コンテナがコンテナ管理のエンティティに対するテーブルを自動的に作成します。ダイアレクトを指定しない限り、コンテナはこれらのテーブルを作成しません。

ただし、JDataStore データベースの場合、コンテナは URL からダイアレクトを検出できます。したがって、JDataStore については、ダイアレクトを明示的に指定したかどうかに関係なく、コンテナはテーブルを作成します。

次の表に、さまざまなダイアレクト の名前と 値を示します。値の大文字と 小文字は区別しません。

データベース名	ダイアレクト値
JDataStore	jdatastore
Oracle	oracle
Sybase	sybase
MSSQLServer	mssqlserver
DB2	db2
Interbase	interbase
Informix	informix
データベースなし	なし。

Java 型から SQL 型へのマッピング

既存のデータベースに対応するエンタープライズ Bean を開発する場合は、データベーススキーマで指定されている SQL データ型を Java プログラミング言語のデータ型にマッピングする必要があります。

Borland EJB コンテナは、Java プログラミング言語の型を SQL 型にマッピングする JDBC 規則をサポートします。JDBC は、よく使用される SQL 型を表す共通の SQL 型識別子を定義しています。既存のデータベーステーブルをモデル化するエンタープライズ Bean を開発するときは、これらのデフォルトの JDBC マッピング規則を使用してください。これらのタイプは、クラス `java.sql.Types` で定義します。

以下の表では、JDBC 仕様で定義する SQL 型から Java 型へのデフォルトマッピングを示します。

Java 型	JDBC SQL 型
boolean/Boolean	BIT
byte/Byte	TINYINT
char/Character	CHAR(1)
double/Double	DOUBLE
float/Float	REAL
int/Integer	INTEGER
long/Long	BIGINT
short/Short	SMALLINT
String	VARCHAR
java.math.BigDecimal	NUMERIC
byte[]	VARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.util.Date	TIMESTAMP
java.io.Serializable	VARBINARY

自動テーブルマッピング

Borland EJB コンテナには、エンタープライズ Bean のコード内で定義されている Java 型をデータベーステーブルの型に自動的にマッピングする機能があります。ただし、そのように自動的にテーブルが作成されても、必ずしも最も適切なマッピングが使用されるとは限りません。実際、マッピングとテーブルをこのように自動生成することは、開発者にとってより便利なものです。

Borland によって生成されるテーブルは、パフォーマンスについては最適化されません。データベースリソースを過度に使用することはよくあります。たとえば、コンテナは、Java の `String` フィールドを対応する SQL の `VARCHAR` 型にマッピングします。ただし、このマッピングでは Java フィールドの実際の長さが考慮されないため、すべての文字列フィールドが最大長の `VARCHAR` にマッピングされます。したがって、2 文字の Java の `String` が `VARCHAR(2000)` 列にマッピングされます。

本稼動の状態では、データベース管理者がテーブルを作成し、型マッピングを行うことをお勧めします。データベース管理者は、デフォルトのマッピングより優先して、パフォーマンスとデータベースリソースの使用に関して最適化されたテーブルを生成できます。

すべてのリレーショナルデータベースが SQL 型を実装していますが、その実装方法は、各データベースによってさまざまです。同じセマンティクスを持つ SQL 型が、データベースによって異なる名前で識別されることもあります。たとえば、Java の `boolean` が、Oracle では `NUMBER(1,0)`、Sybase では `BIT`、DB2 では `SMALLINT` として実装されています。

Borland EJB コンテナがエンタープライズ Bean に対応するデータベーステーブルを作成する場合は、エンティティ Bean フィールドとデータベーステーブル列が自動的にマッピングされます。サポートする各データベースにテーブルを正しく作成するには、EJB コンテナが SQL 型の表現方法を知る必要があります。EJB コンテナは、使用されるデータベースに応じて、一部の Java 型を異なる方法でマッピングします。次の表は、Oracle、Sybase/MSSQL、および DB2 の場合のマッピングです。

Java 型	Oracle	Sybase ／ MSSQL	DB2
boolean/Boolean	NUMBER(1,0)	BIT	SMALLINT
byte/Byte	NUMBER(3,0)	TINYINT	SMALLINT
char/Character	CHAR(1)	CHAR(1)	CHAR(1)
double/Double	NUMBER	FLOAT	FLOAT
float/Float	NUMBER	REAL	REAL
int/Integer	NUMBER(10,0)	INT	INTEGER
long/Long	NUMBER(19,0)	NUMERIC(19,0)	BIGINT
short/Short	NUMBER(5,0)	SMALLINT	SMALLINT
String	VARCHAR(2000)	TEXT	VARCHAR(2000)
java.math.BigDecimal	NUMBER(38)	DECIMAL(28,28)	DECIMAL
byte[]	LONG RAW	IMAGE	BLOB
java.sql.Date	DATE	DATETIME	DATE
java.sql.Time	DATE	DATETIME	TIME
java.sql.Timestamp	DATE	DATETIME	TIMESTAMP
java.util.Date	DATE	DATETIME	TIMESTAMP
java.io.Serializable	RAW(2000)	IMAGE	BLOB

次の表は、JDatastore、Informix、および Interbase を使用する場合の Java 型から SQL 型へのマッピングです。

Java 型	JDatastore	Informix	Interbase
boolean/Boolean	BOOLEAN	SMALLINT	SMALLINT
byte/Byte	SMALLINT	SMALLINT	SMALLINT
char/Character	CHAR(1)	CHAR(1)	CHAR(1)
double/Double	DOUBLE	FLOAT	DOUBLE PRECISION
float/Float	FLOAT	SMALLFLOAT	FLOAT
int/Integer	INTEGER	INTEGER	INTEGER
long/Long	LONG	DECIMAL(19,0)	NUMBER(15,0)
short/Short	SMALLINT	SMALLINT	SMALLINT
String	VARCHAR	VARCHAR(2000)	VARCHAR(2000)
java.math.BigDecimal	NUMERIC	DECIMAL(32)	NUMBER(15,15)
byte[]	OBJECT	BYTE	BLOB
java.sql.Date	DATE	DATE	DATE
java.sql.Time	TIME	DATE	DATE
java.sql.Timestamp	TIMESTAMP	DATE	DATE
java.util.Date	TIMESTAMP	DATE	DATE
java.io.Serializable	OBJECT	BYTE	BLOB

第 14 章

エンティティ Bean と CMP 2.x のテーブルマッピング

ここでは、Borland Enterprise Server にエンティティ Bean をデプロイメントする方法と永続性を管理する方法について説明します。ただし、これはエンティティ Bean そのものの入門書ではないのでそのようにお読みください。というより、ここでは Borland パーティション内におけるエンティティ Bean の使用時の背景説明が主になります。また、デスクリプタ、永続性オプション、その他コンテナの最適化について解説します。コンテナ管理永続性（CMP）の Borland 固有のデプロイメントデスクリプタとインプリメンテーションについては、一般に Sun Microsystems の J2EE 仕様から入手できる EJB 情報を優先して解説します。

エンティティ Bean

エンティティ Bean はデータベースに保存されるデータのビューを表します。エンティティ Bean は、エンティティ Bean とテーブル行が 1 対 1 の対応で、1 つのテーブルにマッピングされた細粒度エンティティの場合もあります。あるいは、複数のテーブルにまたがり、基底のデータベーススキーマとは無関係に存在するデータを表す場合もあります。エンティティ Bean どうしは相互に関係を持ち、クライアントから照会でき、さまざまなクライアント間で共有することができます。

Borland AppServer パーティションのいずれかにエンティティ Bean をデプロイメントするには、JAR の一部としてパッケージにしておく必要があります。JAR には、`ejb-jar.xml` ファイルと独自の `ejb-borland.xml` ファイルの 2 つのデスクリプタを組み込みます。`ejb-jar.xml` デスクリプタについては、Sun Java Center を参照してください。このマニュアルでは `ejb-borland.xml` の DTD を転載しており、合わせてその使用方法についても紹介します。Borland プロプリエタリデスクリプタには、多くのプロパティの設定が可能であり、その設定いかんで、コンテナパフォーマンスを最適化したり、エンティティ Bean の永続性を管理することができます。

コンテナ管理の永続性と関係

Borland の EJB コンテナには、エンティティ Bean をデプロイメントするときに、つまりエンティティ Bean をパーティションにインストールするときに永続性呼び出しを生成するツールが組み込まれています。これらのツールではデプロイメントデスクリプタで、永続性を設定するインスタンスフィールドがどれであるかを決定します。この場合、データベースアクセスを Bean に直接コーディングすることはありません。コンテナツールでアクセス呼び出しを生成する対象となるインスタンスフィールドをコンテナ管理のエンティティ Bean の Bean プロバイダはデプロイメントデスクリプタで指定します。EJB コンテナには、エンティティ Bean のフィールドをデータソースにマッピングする先進的なデプロイメントツールが備わっています。

コンテナ管理の永続性には、Bean 管理の永続性に比べて多くの長所があります。コンテナ管理の永続性を使用すると、Bean プロバイダがデータベースアクセス呼び出しをコーディングする必要がないため、コーディングが簡単です。永続性の処理方法を変更する場合も、エンティティ Bean のコードを変更して再コンパイルする必要がありません。デプロイヤやアプリケーションアセンブラでエンティティ Bean のデプロイメント時にデプロイメントデスクリプタを変更することで、永続性の処理方法を変更できます。このようにデータベースアクセスと永続性の管理を EJB コンテナに任せると、Bean のコードを単純化でき、発生するエラーの範囲を狭めることができます。また、Bean プロバイダは基底のシステム関連の問題にとらわれずに Bean のビジネスロジックに集中できます。

Borland の永続性マネージャ（PM）では、CMP フィールドの永続性を維持するだけでなく、CMP 関係の永続性も維持します。コンテナは Bean 関係を管理し、これらの関係の参照の整合性を維持します。EJB 2.0 仕様を使用すると、コンテナ管理の永続性フィールドを定義したように、コンテナ管理の関係フィールドも Bean のデプロイメントデスクリプタを定義できます。コンテナは、1 対 1 の関係と 1 対多、多対多など、さまざまなカーディナリティをサポートします。

パッケージ要件

セッション Bean と同様に、エンティティ Bean でもリモートインターフェースやローカルインターフェースで、メソッドをエクスポーズできます。リモートインターフェースは、ネットワークを介してほかのリモートコンポーネントに Bean のメソッドをエクスポーズします。ローカルインターフェースは、ローカルクライアント、つまり同じ EJB コンテナにあるクライアントにだけ Bean のメソッドをエクスポーズします。

EJB 2.0 コンテナ管理の永続性を使用するエンティティ Bean は、ローカルモデルを使用する必要があります。つまり、エンティティ Bean のローカルインターフェースは、`EJBLocalObject` インターフェースを拡張します。Bean のローカルホームインターフェースは、`EJBLocalHome` インターフェースを拡張します。これらのインターフェースも、Bean のクラスのインプリメンテーション同様デプロイメントする必要があります。

エンティティ Bean は JAR のデプロイメントデスクリプタにも対応するエントリを持っています。標準デプロイメントデスクリプタ `ejb-jar.xml` には、原則として 3 種類のデプロイメント情報が収められます。次にそれらのデプロイメント情報について説明します。

1. **一般 Bean 情報：**これは、デスクリプタファイルにある `<enterprise-beans>` 要素に対応しており、3 種類すべての Bean に使用します。この情報には、Bean のインターフェースとクラス、セキュリティ情報、環境情報、さらには照会宣言まで含まれています。
2. **関係：**これは、デスクリプタファイルにある `<relationships>` 要素に対応しており、CMP だけを使用するエンティティ Bean に適用されます。ここに、コンテナ管理の関係を記述します。
3. **アセンブリ情報：**これは、デスクリプタファイルにある `<assembly-descriptor>` 要素に対応しており、Bean とアプリケーション間の関係を全体的に説明します。アセンブリ情報は 4 つのカテゴリに分類できます。
   - **セキュリティロール：**アプリケーションが使用するセキュリティロールの単純な定義。ユーザーが Bean に定義するセキュリティロールリファレンスも定義する必要があります。
   - **メソッド許可：**各 Bean のメソッドには、それぞれに実行に関する一定の規則が適用されます。規則はここで設定します。

- **コンテナトランザクション：**トランザクションに関連するメソッドごとに、EJB 2.0 仕様にしたがってトランザクション属性を指定します。
- **除外リスト：**呼び出される相手がいないメソッド。

また、各エンティティ Bean では、Borland 固有のデスクリプタファイル `ejb-borland.xml` に永続性情報を保存します。このデスクリプタファイルでは、バッキングストアでエンティティの永続性を維持するために Borland CMP エンジンと PM が使用する情報を指定します。次の情報を指定します。

- **一般 Bean 情報：**Enterprise JavaBeans に関する情報。インターフェースの場所など。
- **テーブルプロパティと列プロパティ：**JAR のエンティティ Bean が使用するデータベーステーブルと列に関する情報。
- **セキュリティロール：**デプロイメント済み Enterprise JavaBeans の承認情報 。

以上は、いずれもデプロイメントデスクリプタエディタからアクセスできます。DTD 情報とデスクリプタファイルの正しい使用方法については、EJB 2.0 仕様を参照してください。

リエントラントに関する注意

デフォルトでは、エンティティ Bean はリエントラントではありません。同じトランザクションコンテキスト内でエンティティ Bean に呼び出しが到着すると、例外 `java.rmi.RemoteException` が生成されます。

デプロイメントデスクリプタの中で、エンティティ Bean をリエントラントとして宣言できます。ただし、その場合は注意が必要です。通常、コンテナは、同一トランザクション内でのループバック呼び出しと、同一トランザクションコンテキスト内での同一エンティティ Bean に対する同時呼び出しを区別できません。

エンティティ Bean をリエントラントとしてマークした場合、その Bean インスタンスに対して同一トランザクションコンテキスト内で同時呼び出しを行うことはできません。プログラマはこの規則を厳守してください。

App Server におけるコンテナ管理の永続性

Borland AppServer の EJB コンテナは、J2EE 1.3 完全準拠です。EJB 1.1 仕様と EJB 2.0 仕様の両方またはどちらかを実装する Enterprise JavaBeans のコンテナ管理の永続性（CMP）を実装します。Bean プロバイダは、エンティティ Bean に永続性スキーマを設計し、コンテナ管理のフィールドと関係をアクセスするメソッドを決定し、これらのメソッドを Bean のデプロイメントデスクリプタに定義します。デプロイヤは、この永続性スキーマをデータベースにマッピングし、Bean のメンテナンスに必要なほかのクラスを作成します。

Sun Microsystems の EJB 2.0 仕様は、第 10 章と第 11 章で述べた Bean とコンテナの協定の仕様の明細です。永続性スキーマの作成方法の説明については、このマニュアルでは触れません。Sun の仕様や Borland JBuilder のマニュアルを参照してください。また、『*Enterprise JavaBeans Developer's Guide*』と『*Distributed Application Developer's Guide*』には関連情報も記載されています。

永続性マネージャについて

永続性マネージャ（PM）は、エンティティ Bean の読み書き用のデータアクセス層を提供します。エンティティと EJB-QL の拡張機能間の関係のナビゲーションやメンテナンスサポートも提供します。現在、PM は JDBC により、リレーショナルデータベースのデータアクセスだけをサポートしています。PM では、楽観的同期方式でデータをアクセスします。リソース状態の競合は、トランザクションコミット前、またはロールバック前に、検査済み SQL 更新文と削除文で解決します。

PM はトランザクションを管理しませんが（コンテナで処理）、トランザクション状態と終了は認識しており、したがってエンティティ状態は管理できます。PM は、トランザクションライフサイクルで、`TxContext` クラスを利用して管理エンティティのルートを表します。コンテナによるトランザクション管理では、コンテナから PM に対して関連 `TxContext` インスタンスが要求されます。新しいトランザクションの開始時など関連インスタンスがない場合、PM が作成します。トランザクションが終了すると、コンテナはメソッド `TxContext.beforeCompletion()` を呼び出して、PM にエンティティ状態を検査するように警告します。

エンティティデータストレージの管理と、エンティティどうしの関係の状態の維持管理から PM は解放されます。関係の編集も PM によって管理されます。これにより、コンテナとの対話が簡素化でき、PM は読み書き操作を最適化できます。この方法では、要求されたエンティティに対して返される主キーの追跡により、`find` 要求が重複するのを避けることができます。重複した `find` 操作があると、そのデータはエンティティのデータの最初の読み込み時に返ることがあります。

Borland CMP エンジンの CMP 2.x インプリメンテーション

CMP 2.x では、ファインダと select メソッドの構築に関する詳細が EJB 2.0 仕様に移されました。データベース SQL のインプリメンテーション時の詳細については、仕様をよく確認する必要があります。Borland EJB コンテナは EJB 2.0 仕様に完全に準拠しており、その全機能をサポートしています。

2.0 コンテナ管理の永続性を使用したエンティティ Bean インプリメンテーションクラスは、2.0 コンテナ管理の永続性を使用したインプリメンテーションクラスとは異なります。主な相違点は次のとおりです。

- クラスが抽象クラスとして宣言されます。
- コンテナ管理フィールドであるフィールドには公開宣言はありません。かわりに、コンテナ管理フィールドには抽象的な `get` メソッドと `set` メソッドがあります。get メソッドと set メソッドが抽象である理由は、コンテナがこれらメソッドのインプリメンテーションを提供するためです。たとえば、フィールド `balance` と `name` を宣言するかわりに、`CheckingAccount` クラスに次のような get メソッドと set メソッドを組み込むことができます。

```
public abstract float getBalance();
public abstract void setBalance(float bal);
public abstract String getName();
public abstract void setName(String n);
```

- コンテナ管理の関係フィールドも同様に、インスタンス変数として宣言されません。クラスはかわりに抽象 get メソッドおよび set メソッドをコンテナ管理の関係フィールドに提供して、コンテナがこれらメソッドのインプリメンテーションを提供します。

CMP 2.x のテーブルマッピングは、ベンダー固有の `ejb-borland.xml` デプロイメントデスクリプタで行います。デスクリプタは、EJB 2.0 仕様で定めた ejb-jar.xml デスクリプタに添付されています。Borland では、必要に応じて XML タグ `<cmp2-info>` をテーブルマッピングデータのエンクロージャに使用します。`<table-properties>` 要素と、その関連 `<column-properties>` 要素で、エンティティ Bean のインプリメンテーションに関する特定の情報を指定します。DTD は、XML 文法の構文に使用します。

楽観的同期動作

コンテナは、楽観的または悲観的同期を使用して、同じデータにアクセスする複数のトランザクションの動作を制御します。AppServer には、テーブルプロパティとして指定されている 4 つの楽観的同期動作があります。次のような動作があります。

- SelectForUpdate
- SelectForUpdateNoWAIT
- UpdateAllFields
- UpdateModifiedFields
- VerifyModifiedFields
- VerifyAllFields

コンテナの動作は、optimisticConcurrencyBehavior テーブルプロパティの値に対応します。

悲観的同期

このモードでは、コンテナは、エンティティ Bean が保持するデータに一度に 1 つのトランザクションだけがアクセスできるようにします。同じデータを探しているほかのトランザクションは、最初のトランザクションがコミットまたはロールバックするまでブロックします。それには、SelectForUpdate テーブルプロパティを設定し、FOR UPDATE 文を含む調整した SQL 文を発行します。この SQL は、CMP エンジンによって生成される SQL の上書きによって発行できます。その行に対するほかの select 文は、それまでブロックされます。生成される調整 SQL は、次のようになります。

```
SELECT ID, NAME FROM EMP_TABLE WHERE ID=?FOR UPDATE
```

また、SelectForUpdateNoWAIT テーブルプロパティを指定することもできます。これを指定すると、データベースは、現在のトランザクションがコミットまたはロールバックされるまで、行をロックします。ただし、行内でほかを選択しようとすると、選択はブロックされるのではなく、失敗します。次の SQL 文は、このような SELECT 文の例を示しています。

```
SELECT ID, NAME FROM EMP_TABLE WHERE ID=?FOR UPDATE NOWAIT
```

これらのオプションは、注意して使用してください。データの整合性は保証されますが、アプリケーションのパフォーマンスがかなり低下する可能性があります。また、このモードではエンティティ Bean がメモリ内に留まり、ejbLoad() 呼び出しがトランザクション間で行われないため、オプション A キャッシュを使用すると、このオプションは機能しません。

楽観的同期

このモードでは、コンテナは、複数のトランザクションが同時に同じデータを操作するのを許可します。このモードは、パフォーマンスの点では優れていますが、データの整合性が損なわれる可能性があります。

AppServer には、テーブルプロパティとして指定されている 4 つの楽観的同期動作があります。次のような動作があります。

- SelectForUpdate
- SelectForUpdateNoWAIT
- UpdateAllFields
- UpdateModifiedFields
- VerifyModifiedFields
- VerifyAllFields

SelectForUpdate このオプションは、悲観的同期で使用します。このオプションを指定すると、現在のトランザクションがコミットまたはロールバックされるまで、データベースがその行をロックします。その行に対するほかの select 文は、それまでブロックされます。

SelectForUpdateNoWAIT このオプションは、悲観的同期で使用します。このオプションを指定すると、現在のトランザクションがコミットまたはロールバックされるまで、データベースがその行をロックします。その行に対するほかの select 文は、失敗します。

UpdateAllFields このオプションを指定した場合、コンテナは、フィールドが変更されたかどうかに関係なく、すべてのフィールドを更新します。たとえば、CMP エンティティ Bean に KEY、VALUE1、VALUE2 という 3 つのフィールドがあるとします。Bean が変更されたかどうかに関係なく、トランザクションが終了するたびに次の更新が発行されます。

```
UPDATE MyTable SET (VALUE1 = value1, VALUE2 = value2) WHERE KEY = key
```

UpdateModifiedFields このオプションは、デフォルトの楽観的同期動作です。コンテナは、トランザクションで変更されたフィールドだけを更新します。Bean が変更されていない場合は、更新をすべて抑止します。前述の例と同じ Bean について、トランザクションで VALUE1 だけが変更された場合を考えてみます。`UpdateModifiedFields` を使用すると、コンテナは、次のような更新を発行します。

```
UPDATE MyTable SET (VALUE1 = value1) WHERE KEY = key
```

このオプションを使用すると、アプリケーションのパフォーマンスが著しく向上する可能性があります。データアクセスは、多くの場合読み取り専用で行われます。その場合、トランザクションのたびにデータベースに更新情報を送信するのを避けたほうが、処理時間を大幅に節約できます。また、このような更新を抑止すると、データベースインプリメンテーションが更新を記録しないようにでき、パフォーマンスが向上します。JDBC ドライバの負担も、特に大規模な EJB アプリケーションで、大幅に減少します。よく調整されたドライバでも、ドライバ上での作業量が小さい方がパフォーマンスは上がります。

VerifyModifiedFields このオプションを有効にすると、CMP エンジンは、調整された更新を発行しますが、その際、更新されるフィールドが以前の値と一致するかどうかを検証します。トランザクションが初めに値を読み込んでから更新の準備ができるまでの間に値が変化した場合、トランザクションはロールバックします。このロールバックを適切に処理する必要があります。値が変化していない場合、トランザクションはコミットします。前述の例と同じテーブルを使用する場合を考えます。VALUE1 だけが更新された場合、`VerifyModifiedFields` を使用すると、CMP エンジンは次の SQL を生成します。

```
UPDATE MyTable SET (VALUE1 = value1) WHERE KEY = key AND VALUE1 = old-VALUE1
```

VerifyAllFields

このオプションは、すべてのフィールドが検証される点を除いて、`VerifyModifiedFields` によく似ています。前述の例と同じテーブルを使用する場合で考えると、このオプションを使用すると、CMP エンジンは次の SQL を生成します。

```
UPDATE MyTable SET (VALUE1 = value1) WHERE KEY = key AND VALUE1 = old-VALUE1
AND VALUE2 =  old-VALUE2
```

メモ 2 つの検証設定を使用すると、コンテナに `SERIALIZABLE` 分離レベルを複製できます。アプリケーションがシリアライズ可能な分離セマンティクスを必要とする場合があります。しかし、データベースに分離セマンティクスの実装を要求すると、パフォーマンスに大きな影響を及ぼす可能性があります。検証設定を使用すると、CMP エンジンは、フィールドレベルのロックを使って楽観的同期を実装できます。ロックの粒度が小さくなると、同期のパフォーマンスは向上します。

永続性スキーマ

Borland CMP 2.x エンジンでは、エンティティ Bean の構造や、エンティティ Bean のデプロイメントデスクリプタで提供される情報に基づいて、基底のデータベーススキーマを作成できます。このような場合、CMP マッピング情報を提供する必要はありません。次の「テーブルとデータソースの指定」の手順にしたがってください。あるいは、CMP エンジンにより、既存の基底のデータベーススキーマへの適合が行われます。ただしその場合は、データベーススキーマに関する情報を CMP エンジンに提供する必要があります。また、「テーブルとデータソースの指定」のケース 2 の場合は、124 ページの「列に対する CMP フィールドの基本マッピング」も参照してください。

テーブルとデータソースの指定

in ejb-borland.xml に必要な最小限の情報、エンティティ Bean 名と関連データソースです。データソースは、データベースへの接続を取得するために使用します。データソースの設定については、181 ページの「Borland AppServer を使用したリソースへの接続：定義アーカイブ（DAR）の使い方」を参照してください。この情報の提供手段は、2 つあります。

ケース 1：JDataStore データベースか Cloudscape データベースのどちらかを使用する、既存のデータベーステーブルがない開発環境。

この場合、Borland CMP エンジンは、エンティティ Bean 名が目的のテーブル名と同じであるとみなし、テーブルを自動的に作成します。指定する必要があるのは Bean の名前と、プロパティとしてのその関連データソースだけです。

```
<entity>
 <ejb-name>CustomerEJB</ejb-name>
 <property>
  <prop-name>ejb.datasource</property>
  <prop-value>serial://ds/myDatasource</prop-value>
 </property>
</entity>
```

Borland CMP エンジンは、Bean 名とフィールドから、このデータソースにテーブルを自動的に作成します。

ケース 2：サポートされているデータベースを使用する、既存のデータベーステーブルがないデプロイメント環境。

この場合、エンティティがマッピングするテーブルに関する情報を提供する必要があります。テーブル名は、デスクリプタの <entity> 部と、<table-properties> 部の何か所かで指定します。

```
<entity>
 <ejb-name>CustomerEJB</ejb-name>
 <cmp2-info>
  <table-name>CUSTOMER</table-name>
 </cmp2-info>
</entity>
.
.
<table-properties>
 <table-name>CUSTOMER</table-name>
  <property>
   <prop-name>datasource</prop-name>
   <prop-value>serial://ds/myDatasource</prop-value>
  </property>
</table-properties>
```

データソースプロパティは、<table-properties> 要素で指定する場合は datasource、<entity> 要素で指定する場合は ejb.datasource と呼ばれます。JDataStore や Cloudscape 以外のデータベースを使用し、Borland CMP エンジンでこのテーブルを自動的に作成する場合、次の <table-properties> 要素に XML を追加します。

```
.
.
<table-properties>
 <table-name>CUSTOMER</table-name>
  <property>
   <prop-name>create-tables</prop-name>
   <prop-value>True</prop-value>
  </property>
</table-properties>
```

列に対する CMP フィールドの基本マッピング

基本フィールドマッピングは、ejb-borland.xml デプロイメントデスクリプタの <cmp-field> 要素で行います。この要素の子ノードでは、フィールド名とマップ先の対応列を指定します。次に示す XML にある LineItem というエンティティ Bean を例に考えてみます。この Bean は、orderNumber と line という 2 つのフィールドを ORDER_NUMBER と LINE: という 2 つの列にマッピングします。

```
<entity>
 <ejb-name>LineItem</ejb-name>
 <cmp2-info>
  <cmp-field>
   <field-name>orderNumber</field-name>
   <column-name>ORDER_NUMBER</column-name>
  </cmp-field>
  <cmp-field>
   <field-name>line</field-name>
   <column-name>LINE</column-name>
  </cmp-field>
 </cmp2-info>
</entity>
```

1 つのフィールドを複数の列にマッピング

一般には粗粒度のエンティティ Bean で Java クラスを実装して細粒度のデータを表しています。たとえば、フィールドとして Address クラスを使用するエンティティ Bean があり、クラス要素（AddressLine1、AddressCity など）は基底のデータベースにマッピングする必要があるとします。それには、<cmp-field-map> 要素を使用します。これは細粒度クラスと、基底のそのデータベース表現のフィールドマップを定義する要素です。このようなクラスでは、java.io.Serializable を実装するものとし、そのすべてのデータメンバーはパブリックであるものとします。

Customer というエンティティ Bean について考えてみます。これは、クラス Address で顧客住所を表現します。Address クラスには、AddressLine、AddressCity、AddressState、AddressZip のフィールドがあります。次の XML では、対応する列とともに、データベースの表現にクラスをマッピングします。

```
<entity>
 <ejb-name>Customer</ejb-name>
 .
 .
 <cmp2-info>
  <cmp-field>
   <field-name>Address</field-name>
   <cmp-field-map>
    <field-name>Address.AddressLine</field-name>
    <column-name>STREET</column-name>
   </cmp-field-map>
   <cmp-field-map>
    <field-name>Address.AddressCity</field-name>
    <column-name>CITY</column-name>
   </cmp-field-map>
   <cmp-field-map>
    <field-name>Address.AddressState</field-name>
    <column-name>STATE</column-name>
   </cmp-field-map>
   <cmp-field-map>
    <field-name>Address.AddressZip</field-name>
    <column-name>ZIP</column-name>
   </cmp-field-map>
  </cmp-field>
 </cmp2-info>
 .
 .
</entity>
```

これで、データベース列 1 本につき、<cmp-field-map> 要素を 1 つ使用できます。

CMP フィールドを複数のテーブルにマッピング

複数のテーブルで永続性を備える情報を収めたエンティティを設定できます。このテーブルは、リンクテーブル内で外部キーを表す少なくとも 1 本の列でリンクする必要があります。たとえば、QUANTITY というテーブルに収められた外部キーである主キー LINE でテーブル LINE_ITEM にマッピングされた LineItem エンティティ Bean があるとします。LineItem エンティティには、QUANTITY テーブルのフィールドもいくつか収められており、これらは LINE_ITEM の LINE エントリに対応します。次に、LINE_ITEM テーブルのようすを示します。

LINE	ORDER_NO	ITEM	QUANTITY	COLOR	SIZE
001	XXXXXXX01	キティーセーター	2	赤	XL

QUANTITY、COLOR、SIZE は、次に示すようにすべて QUANTITY テーブルにも収められている値です。フィールドのいくつかには同じ値があります。これは、LINE_ITEM テーブルそのものに QUANTITY テーブルの情報が保存されており、LineItem エンティティで複合情報を提供しているからです。

LINE	QUANTITY	COLOR	SIZE
001	2	赤	XL

なお、<cmp-field> 要素と <table-ref> 要素の組み合わせで、以上の関係を記述できます。<cmp-field> 要素は、LineItem にあるフィールドを定義します。QUANTITY の情報が必要なフィールドがあるため、通常は TABLE_NAME.COLUMN_NAME 構文で指定します。たとえば、LINE_ITEM の COLOR 列を QUANTITY.COLOR と定義します。最後に、リンク列 LINE を指定します。これは、主キーと外部キーの関係を作成する列です。そのために、<table-ref> 要素を使用します。

では、XML で見てみましょう。まず、LineItem エンティティ Bean の CMP フィールドを定義します。

```
<entity>
 <ejb-name>LineItem</ejb-name>
 .
 .
 <cmp2-info>
  <cmp-field>
   <field-name>orderNumber</field-name>
   <column-name>ORDER_NO</column-name>
  </cmp-field>
  <cmp-field>
   <field-name>line</field-name>
   <column-name>LINE</column-name>
  </cmp-field>
  <cmp-field>
   <field-name>item</field-name>
   <column-name>ITEM</column-name>
  </cmp-field>
  <cmp-field>
   <field-name>quantity</field-name>
   <column-name>QUANTITY.QUANTITY</column-name>
  </cmp-field>
  <cmp-field>
   <field-name>color</field-name>
   <column-name>QUANTITY.COLOR</column-name>
  </cmp-field>
  <cmp-field>
   <field-name>size</field-name>
   <column-name>QUANTITY.SIZE</column-name>
  </cmp-field>
```

次に、<table-ref> 要素を使って LINE_ITEM と QUANTITY の間のリンク列を指定します。

```
    <table-ref>
     <left-table>
      <table-name>LINE_ITEM</table-name>
      <column-list>
       <column-name>LINE</column-name>
      </column-list>
     </left-table>
     <right-table>
      <table-name>QUANTITY</table-name>
      <column-list>
       <column-name>LINE</column-name>
      </column-list>
     </right-table>
    </table-ref>
   </cmp2-info>
  </entity>
```

テーブル間の関係の指定

テーブル間の関係を指定するには、ejb-borland.xml. で <relationships> 要素を使用します。<relationships> 要素内で、ロールのソース（エンティティ Bean）を保持する <ejb-relationship-role> と、関係を保持する <cmr-field> 要素を定義します。デスクリプタは、<table-ref> 要素で 2 テーブル（<left-table> と <right-table>）間の関係を指定します。次のカーディナリティができます。

- 方向ごとに 1 つの <ejb-relationship-role> ができます。双方向関係の場合、相互に関係のある Bean ごとに <ejb-relationship-role> を定義してください。
- 関係 1 つにつき、使用できる <table-ref> 要素は 1 つだけです。

<left-table> 要素と <right-table> 要素で、リンクする列名どうしを収めた列リストを指定します。列リストは、デスクリプタの <column-list> 要素に対応します。XML は次のとおりです。

```
<!ELEMENT column-list (column-name+)>
```

次に、この XML を実際にはどのように使用するか、いくつかの関係を見てみましょう。

ケース 1：単方向の 1 対 1 関係。

ここでは、主キー CUSTOMER_NO を持つ Customer エンティティ Bean を使用します。主キー CUSTOMER_NO は、エンティティ SpecialInfo の主キーとしても使用されており、このエンティティは別のテーブルに保存されている顧客の特別な情報を収めるエンティティです。この 2 つのエンティティの関係を指定する必要があります。Customer エンティティは、フィールド specialInformation で、SpecialInfo Bean にマッピングします。Bean ごとに 1 つ、合わせて 2 つの関係ロールを指定し、左右のテーブルに分けて割り当てます。次に、各テーブルで対応する列に名前を指定します。

```
   <relationships>
   <ejb-relation>
    <ejb-relationship-role>
     <relationship-role-source>
      <ejb-name>Customer</ejb-name>
     </relationship-role-source>
     <cmr-field>
      <cmr-field-name>specialInformation</cmr-field-name>
      <table-ref>
       <left-table>
        <table-name>CUSTOMER</table-name>
        <column-list>CUSTOMER_NO</column-list>
       </left-table>
       <right-table>
        <table-name>SPECIAL_INFO</table-name>
        <column-list>CUSTOMER_NO</column-list>
       </right-table>
      </table-ref>
     </cmr-field>
    </ejb-relationship-role>
```

次に、残り半分である SpecialInfo Bean を指定して、<ejb-relation> エントリを完成します。単方向関係なので、テーブル要素を指定する必要はありません。残りを追加して、関係のほかの半分とソースを定義します。

```
    <ejb-relationship-role>
     <relationship-role-source>
      <ejb-name>SpecialInfo</ejb-name>
     </relationship-role-source>
    </ejb-relationship-role>
   </ejb-relation>
  </relationships>
```

ケース 2: 双方向の 1 対多関係。

ここでは、主キー CUSTOMER_NO を持つ Customer エンティティ Bean を使用します。主キー CUSTOMER_NO は、Order エンティティ Bean の外部キーとしても使用されています。この関係を Borland EJB コンテナで管理します。Customer Bean は「orders」というフィールドを使用します。このフィールドは、顧客とその注文をリンクします。Order Bean は、フィールド「customers」を使用します。このフィールドは逆方向のリンクを行います。まず、最初の方向の関係とそのソースを定義し、Customer の注文のマッピングを設定します。

```
  <relationships>
   <ejb-relation>
    <ejb-relationship-role>
     <relationship-role-source>
      <ejb-name>Customer</ejb-name>
     </relationship-role-source>
     <cmr-field>
      <cmr-field-name>orders</cmr-field-name>
```

次に、テーブル間の関係を指定するテーブルリファレンスを指定します。この関係は、CUSTOMER_NO 列から抽出します。これは、Customer の主キーであり、Orders の外部キーです。

```
      <table-ref>
       <left-table>
        <table-name>CUSTOMER</table-name>
        <column-list>
         <column-name>CUSTOMER_NO</column-name>
        </column-list>
       </left-table>
       <right-table>
        <table-name>ORDER</table-name>
        <column-list>
         <column-name>CUSTOMER_NO</column-name>
        </column-list>
       </right-table>
      </table-ref>
     </cmr-field>
    </ejb-relationship-role>
```

ただし、これで関係が完成したわけではありません。残った方向の関係を指定しないと完成ではありません。

```
    <ejb-relationship-role>
     <relationship-role-source>
      <ejb-name>Customer</ejb-name>
     </relationship-role-source>
     <cmr-field>
      <cmr-field-name>customers</cmr-field-name>
      <table-ref>
       <left-table>
        <table-name>ORDER</table-name>
        <column-list>
         <column-name>CUSTOMER_NO</column-name>
        </column-list>
       </left-table>
       <right-table>
        <table-name>CUSTOMER</table-name>
        <column-list>
         <column-name>CUSTOMER_NO</column-name>
        </column-list>
       </right-table>
```

```
      </table-ref>
     </cmr-field>
    </ejb-relationship-role>
   </ejb-relation>
   .
   .
  </relationships>
```

ケース 3：多数対多関係。

多対多関係を定義する場合、CMP エンジンでクロステーブルを作成する必要があります。このテーブルは左右テーブルの関係をモデル化するテーブルです。これには、<cross-table> 要素を使用します。次に、その XML を示します。

```
<!ELEMENT cross-table (table-name, column-list, column-list)>
```

クロステーブルには、<table-name> 要素で好きな名前を付けることができます。2 つの <column-list> 要素は、関係モデルを作成する左右テーブルの列に対応します。たとえば、多対多の関係がある 2 つのテーブル EMPLOYEE と PROJECT があるとします。PROJECT テーブルには、プロジェクト ID 番号（PROJ_ID）、プロジェクト名（PROJ_NAME）と、担当の従業員の番号（EMP_NO）の列があります。EMPLOYEE テーブルには 3 つの要素があります。それは、従業員番号（EMP_NO）、姓（LAST_NAME）、プロジェクト ID 番号（PROJ_ID）です。PROJECT テーブルには、プロジェクト ID 番号（PROJ_ID）、プロジェクト名（PROJ_NAME）と、担当の従業員の番号（EMP_NO）の列があります。

以上 2 つのテーブルの関係モデルを作成するには、クロステーブルを作成する必要があります。たとえば、従業員名とその作業プロジェクトの名前を示すクロステーブルを作成するには、次のような <table-ref> 要素を作成します。

```
<table-ref>
    <left-table>
       <table-name>EMPLOYEE</table-name>
          <column-list>
                <column-name>EMP_NO</column-name>
         <column-name>LAST_NAME</column-name>
             <column-name>PROJ_ID</column-name>
          </column-list>
    </left-table>
    <cross-table>
        <table-name>EMPLOYEE_PROJECTS</table-name>
           <column-list>
                <column-name>EMP_NAME</column-name>
          <column-name>PROJ_ID</column-name>
           </column-list>
           <column-list>
            <column-name>PROJ_ID</column-name>
               <column-name>PROJ_NAME</column-name>
           </column-list>
    </cross-table>
    <right-table>
        <table-name>PROJECT</table-name>
           <column-list>
               <column-name>PROJ_ID</column-name>
            <column-name>PROJ_NAME</column-name>
            <column-name>EMP_NO</column-name>
           </column-list>
    </right-table>
</table-ref>
```

「二次テーブル」があり、そのために主キーがないので、PROJ_ID 列は両方の列リストに表示されます。これは、データのモデル化の方法によっては、共通列 EMP_NO になる場合もあります。

カスケード削除とデータベースカスケード削除の使用

<cascade-delete> は、エンティティ Bean オブジェクトを削除する場合に使用します。オブジェクトに対してカスケード削除を指定すると、コンテナは、そのオブジェクトの従属オブジェクトをすべて自動的に削除します。たとえば、Address Bean に対して 1 対多の単一方向の関係を持つ Customer Bean を作成する場合があります。Address インスタンスは、顧客に関連付ける必要があるため、顧客を削除すると、コンテナは、顧客に関連付けられているすべての住所を自動的に削除します。

カスケード削除を指定するには、次に示すように ejb-jar.xml ファイルで <cascade-delete> 要素を使用します。

```
<ejb-relation>
 <ejb-relation-name>Customer-Account</ejb-relation-name>
 <ejb-relationship-role>
  <ejb-relationship-role-name>Account-Has-Customer
  </ejb-relationship-role-name>
  <multiplicity>one</multiplicity>
  <cascade-delete/>
 </ejb-relationship-role>
</ejb-relation>
```

データベースカスケード削除のサポート

AppServer は、データベースカスケード削除機能をサポートします。この機能により、アプリケーションは、データベースに組み込まれているカスケード削除機能を利用できます。これにより、コンテナがデータベースに送信する SQL 操作の数が減少し、その結果パフォーマンスが向上します。

データベースカスケード削除を使用するには、それぞれのデータベースに、適切なテーブル制約を持つ、エンティティ Bean に対応するテーブルを作成する必要があります。たとえば、Order および LineItem エンティティ Bean の EJB 2.0 エンティティ Bean でカスケード削除を使用する場合は、テーブルを次のように作成する必要があります。

```
create table ORDER_TABLE (ORDER_NUMBER integer, LAST_NAME varchar(20),
FIRST_NAME varchar(20), ADDRESS varchar(48));
create table LINE_ITEM_TABLE (LINE integer, ITEM varchar(100), QUANTITY
numeric, ORDER_NUMBER integer CONSTRAINT fk_order_number REFERENCES
ORDER_TABLE(ORDER_NUMBER) ON DELETE CASCADE);
```

ejb-borland.xml ファイルの <cascade-delete-db> 要素は、カスケード削除操作がデータベースのカスケード削除機能を使用することを指定します。デフォルトでは、この機能はオフになっています。

メモ ejb-borland.xml ファイルで <cascade-delete-db> 要素を指定する場合、ejb-jar.xml で <cascade-delete> を指定する必要があります。

次の <relationships> 要素は、ejb-borland.xml の <cascade-delete-db> の XML の例を示します。

```
<relationships>
        <!--
        ONE-TO-MANY: Order LineItem
        -->
        <ejb-relation>
            <ejb-relationship-role>
                <relationship-role-source>
                    <ejb-name>OrderEJB</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>lineItems</cmr-field-name>
                    <table-ref>
                        <left-table>
                            <table-name>ORDER_TABLE</table-name>
                            <column-list>
                                <column-name>ORDER_NUMBER</column-name>
                            </column-list>
                        </left-table>
                        <right-table>
                            <table-name>LINE_ITEM_TABLE</table-name>
                            <column-list>
                                <column-name>ORDER_NUMBER</column-name>
```

```
                        </column-list>
                    </right-table>
                </table-ref>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <relationship-role-source>
                <ejb-name>LineItemEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>order</cmr-field-name>
                <table-ref>
                    <left-table>
                        <table-name>LINE_ITEM_TABLE</table-name>
                        <column-list>
                            <column-name>ORDER_NUMBER</column-name>
                        </column-list>
                    </left-table>
                    <right-table>
                        <table-name>ORDER_TABLE</table-name>
                        <column-list>
                            <column-name>ORDER_NUMBER</column-name>
                        </column-list>
                    <right-table>
                </table-ref>
            </cmr-field>
        </ejb-relationship-role>
<cascade-delete-db />
    </ejb-relation>
</relationships>
```

第 15 章

CMP 2.x の AppServer プロパティの使い方

プロパティの設定

Enterprise JavaBeans のほとんどのプロパティは、デプロイメントデスクリプタで設定できます。Borland デプロイメントデスクリプタエディタ（DDEditor）では、プロパティの設定やデスクリプタファイルの編集ができます。デプロイメントデスクリプタエディタの使用方法については、Borland 管理コンソール の『*ユーザーズガイド*』を参照してください。デプロイメントデスクリプタの使い方の詳細については、131 ページの「デプロイメントデスクリプタエディタの使い方」のセクションを参照してください。デプロイメントデスクリプタのプロパティでは、エンティティ Bean のインターフェース、トランザクション属性などのほか、エンティティ Bean 固有の情報に関する情報を指定します。エンティティ Bean の一般的なデスクリプタ情報以外に、CMP インプリメンテーションをカスタマイズするために設定する 3 セットのプロパティがあります。それが、エンティティプロパティ、テーブルプロパティ、列プロパティです。エンティティプロパティは、デプロイメントデスクリプタエディタの［EJB Designer］タブで設定するか、XML で直接設定します。

デプロイメントデスクリプタエディタの使い方

Borland AppServer（AppServer）に付属のデプロイメントデスクリプタエディタでは、コンテナ管理の永続性に関するすべての情報を設定できます。次の表は、デスクリプタに関する説明と、その情報を入力できるデプロイメントデスクリプタエディタの場所を示しています。

デプロイメントデスクリプタエディタおよびその他の関連ツールの使用の詳細については、『*Borland 管理コンソールユーザーズガイド*』の「デプロイメントデスクリプタエディタの使い方」のセクションを参照してください。

EJB Designer

CMP 2.x プロパティは、EJB Designer を使って設定します。EJB Designer の詳細については、『*Borland 管理コンソールユーザーズガイド*』の「デプロイメントデスクリプタエディタの使い方」の「EJB Designer」のセクションを参照してください。

J2EE 1.3 と 1.4 のエンティティ Bean

デスクリプタ要素	ナビゲーションツリーノード／パネル名	DDEditor のタブ
エンティティ Bean 名	Bean	［General］
エンティティ Bean クラス	Bean	［General］
ホームインターフェース	Bean	［General］
リモートインターフェース	Bean	［General］
ローカルホームインターフェース	Bean	［General］
ローカルインターフェース	Bean	［General］
JNDI 名	Bean	［General］
ローカルホーム JNDI 名	Bean	［General］
永続性タイプ（CMP または BMP）	Bean	［General］
CMP バージョン	Bean	［General］
主キークラス	Bean	［General］
リエントラント	Bean	［General］
アイコン	Bean	［General］
環境エントリ	Bean	［Environment］
その他の Bean への EJB リファレンス	Bean	［EJB References］
EJB リンク	Bean	［EJB References］
データオブジェクト／接続ファクトリへのリソースリファレンス	Bean	［Resource References］
リソースリファレンスの種類	Bean	［Resource References］
リソースリファレンス認証の種類	Bean	［Resource References］
セキュリティロールリファレンス	Bean	［Security Role References］
エンティティプロパティ	Bean	［Properties］
セキュリティ ID	Bean	［Security Identity］
名前 JAR の Bean までの EJB ローカルリファレンス	Bean	［EJB Local References］
EJB ローカルリンク	Bean	［EJB Local References］
JMS のリソース環境リファレンス	Bean	［Resource Env Refs］
コンテナトランザクション	Bean：［Container Transactions］	［Container Transactions］
トランザクションメソッド	Bean：［Container Transactions］	［Container Transactions］
トランザクションメソッドインターフェース	Bean：［Container Transactions］	［Container Transactions］
トランザクション属性	Bean：［Container Transactions］	［Container Transactions］
メソッド許可	Bean：［Method Permissions］	［Method Permissions］
エンティティ、テーブル、列プロパティ	JAR	EJB Designer（以下参照）

CMP 2.x プロパティの設定

AppServer では、EJB Designer（Deployment Descriptor Editor のコンポーネント）で、CMP 2.x プロパティを設定します。EJB Designer については、『*Borland 管理コンソールユーザーズガイド*』の「デプロイメントデスクリプタエディタの使い方」の「EJB Designer」のセクションに詳しい説明があります。

エンティティプロパティの編集

EJB Designer でエンティティプロパティを編集するには、次の手順にしたがいます。

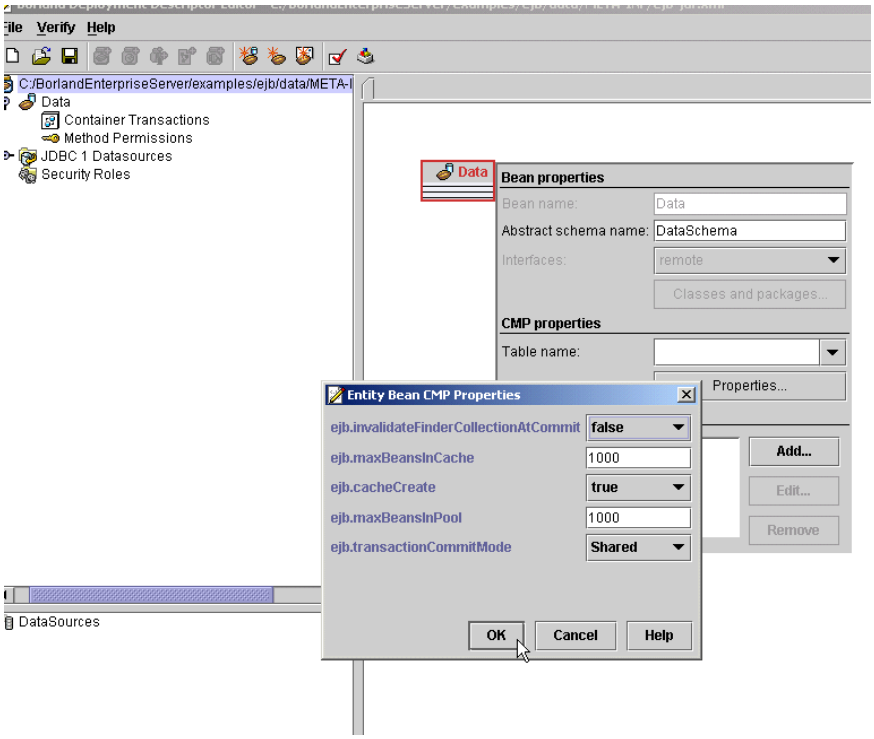
1 DDEditor を起動し、エンティティ Bean がある JAR のデプロイメントデスクリプタを開きます。

2 DDEditor のナビゲーションペインで、最上位のオブジェクトを選択します。プロパティペインに、［General］タブ、［XML］タブ、［EJB Designer］タブの 3 つが表示されます。

3 ［EJB Designer］タブを選択し、表示される Bean 表現のいずれかを左クリックします。［Properties］ボタンをクリックします。［Entity Beans Properties］ウィンドウが開きます。

4 目的のプロパティを編集し、［OK］をクリックします。プロパティそのものについては、次に説明します。

図 15.1 エンティティプロパティの編集



テーブルプロパティと列プロパティの編集

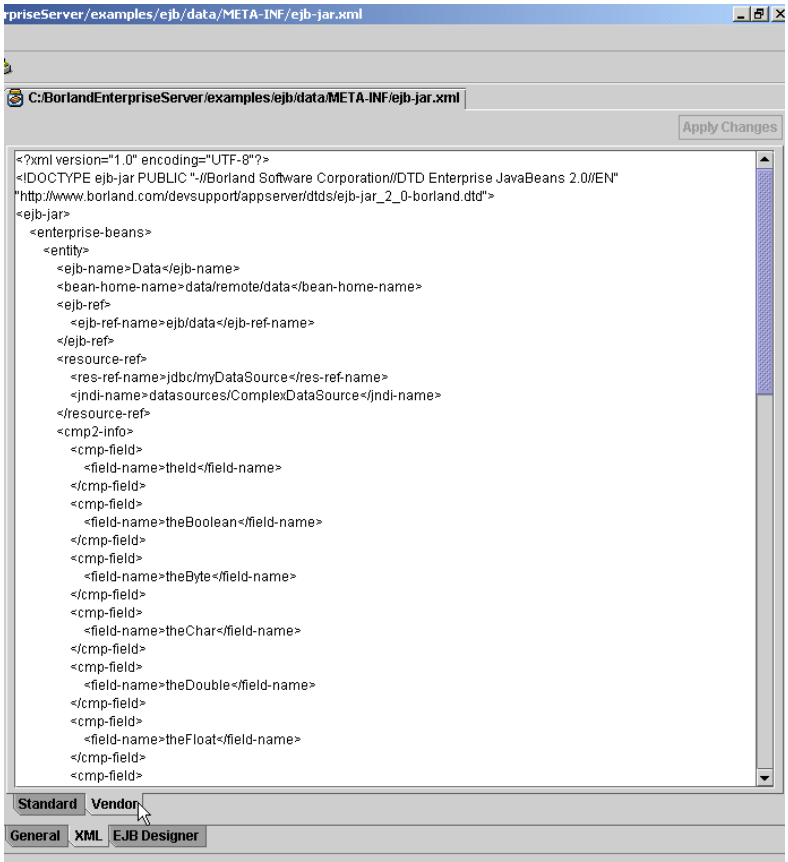
テーブルプロパティと列プロパティを設定するには、DDEditor の［XML］タブ内の［Vendor］タブから `ejb-borland.xml` デスクリプタファイルを編集するか、EJB Designer を使用します。テーブルプロパティと列プロパティを編集、または追加するには、次の手順にしたがいます。

1 DDEditor を起動し、エンティティ Bean がある JAR のデプロイメントデスクリプタを開きます。

2 DDEditor のナビゲーションペインで、最上位のオブジェクトを選択します。プロパティペインに、［General］タブ、［XML］タブ、［EJB Designer］タブの 3 つが表示されます。

3 ［XML］タブを選択します。プロパティペインには、［Standard］タブと［Vendor］タブという 2 つのタブが追加されました。［Vendor］を選択します。

図 15.2 テーブルプロパティと列プロパティの編集



4 `<column-properties>` 要素または `<table-properties>` 要素を探し、Borland 固有 DTD にしたがってプロパティを追加します（「ejb-borland.xml」を参照）。関係エントリは太字です。続けてエンティティ、テーブル、列プロパティの記述を指定します。データ型、デフォルト値、プロパティ記述も指定します。

エンティティプロパティ

次は、CMP 1.1 以上のインプリメンテーションのプロパティです。

プロパティ	型	デフォルト値	説明
`ejb.maxBeansInCache`	`lava.lang.Integer`	`1000`	このオプションは、トランザクションではなく主キーに関連付けられた Bean を保持するキャッシュ内の Bean の最大数を指定します。これは、オプション「A」と「B」に関係します（下記の `ejb.transactionCommitMode` を参照）。キャッシュがこの制限を超えた場合は、ejbPassivate が呼び出されて、エンティティが準備完了プールに移されます。
`ejb.maxBeansInPool`	`java.lang.Integer`	`1000`	準備完了プール内の最大の Bean 数。準備完了プールがこの制限を超えた場合は、`unsetEntityContext()` が呼び出されて、エンティティがコンテナから削除されます。

プロパティ	型	デフォルト値	説明
ejb.maxBeansInTransactions	lava.lang.Integer	500*（「説明」を参照）	1 つのトランザクションから、任意の数の多くのエンティティにアクセスできます。このプロパティにより、EJB コンテナが作成する物理的な Bean インスタンス数の上限を設定します。アクセスされるデータベースエンティティ／データベース行の数に関係なく、コンテナは、制限された数のエンティティオブジェクト（ディスパッチャ）でトランザクションを完了します。このデフォルトは、ejb.maxBeansInCache/2 という計算で求められます。ejb.maxBeansInCache プロパティが設定されていない場合は、500 になります。
ejb.TransactionCommitMode	Enumerated	Shared	トランザクションの面から見たエンティティ Bean の特性を指定します。次の値を指定できます。 ■ **Exclusive**：このエンティティは、データベース内の特定のテーブルに排他的にアクセスします。したがって、最後にコミットされたトランザクションの Bean の状態を次のトランザクションの最初の Bean の状態とみなすことができます。 ■ **Shared**：このエンティティは、データベース内の特定のテーブルまでのアクセスを共有します。ただし、パフォーマンス上の理由から、ejbActivate() と ejbPassivate() をトランザクション間で無駄に呼び出すことのないように、特定の Bean はトランザクション間で特定の主キーに関連付けられたままになります。これらの Bean はアクティブプールに残ります。 ■ **None**：このエンティティは、データベース内の特定のテーブルへのアクセスを共有します。トランザクション間で特定の主キーとの関連付けが解除され、トランザクションごとに準備完了プールに戻される Bean があります。

次は、CMP 2.x インプリメンテーションのみのプロパティです。

プロパティ	型	デフォルト値	説明
ejb.invalidateFinderCollectionAtCommit	java.lang.Boolean	False	ファインダコレクションを無効にしてトランザクションコミットを最適化するかどうか。CMP 2.x のみ。
ejb.cacheCreate	java.lang.Boolean	True	ejbPostCreate が処理されるまでエンティティ Bean の挿入をキャッシュするかどうか。
ejb.datasource	java.lang.String	N/A	table-properties を設定しない場合に使用されるデフォルトの JDBC データソース。CMP 2.x のみ。
ejb.truncateTableName	java.lang.Boolean	False	テーブル名を指定しない場合、CMP2.x エンジンは EJB 名をテーブル名として使用します。EJB 名は、長さが 30 文字を超える場合があります。しかし、一部のデータベースではテーブル名の長さが 30 文字以下に制限されています。このプロパティを使用すると、テーブル名が 30 文字以下になるように切り捨てられます。CMP 2.x のみ。
ejb.eagerLoad	java.lang.Boolean	False	行全体を eager ロードし、そのデータをトランザクションキャッシュに保持します。ロード後、すべてのデータベースリソースは解放されます。その後の getter では、キャッシュ内のデータを取得でき、データベースリソースを要求する必要はありません。CMP 2.x のみ。

テーブルプロパティ

次のプロパティは、CMP 2.x だけに適用されます。CMP 1.1 から CMP 2.x に移行する場合は、CMP プロパティを更新する必要があります。CMP 1.1 プロパティの正式なフォーマットは、ejb.<property-name> で、デプロイメントデスクリプタの <entity> 部分で指定されて

いました。CMP 2.x では、AppServer に永続性を管理するテーブルプロパティと列プロパティが追加されています。次のプロパティでは、移行に関する問題が発生する可能性があります。

プロパティ	型	デフォルト値	説明
datasource	java.lang.String	なし	現在のテーブルのデータベースの JNDI データソース名です。
optimisticConcurrencyBehavior	java.lang.String	UpdateModifiedFields	コンテナは、楽観的または悲観的同期を使用して、共有テーブルにアクセスする複数のトランザクション（更新）を制御します。次の値を指定できます。 ■ **SelectForUpdate**: 現在のトランザクションがコミットまたはロールバックされるまで、データベースがその行をロックします。その行に対するほかの select 文は、それまでブロック（ 待機）されます。 ■ **SelectForUpdateNoWAIT**：現在のトランザクションがコミットまたはロールバックされるまで、データベースがその行をロックします。その行に対するほかの select 文は、失敗します。 ■ **UpdateAllFields**：変更されているかどうかに関係なく、すべてのフィールドを更新します。 ■ **UpdateModifiedFields**：更新実行前に変更が確認されたフィールドのみ更新します。 ■ **VerifyModifiedFields**：エンティティが変更されたフィールドを更新前にデータベースと照合して確認します。 ■ **VerifyAllFields**：変更の有無に関係なく、すべてのエンティティのフィールドを更新前にデータベースを照合して確認します。 悲観的同期は、一度に 1 つのトランザクションだけがエンティティ Bean にアクセスできるようにコンテナに指示します。同じデータにアクセスしようとするほかのトランザクションは、最初のトランザクションが完了するまでブロック（ 待機）されます。このためには、エンティティ Bean がロードされるときに、`FOR UPDATE` を指定して設定された SQL が発行されます。悲観的同期を有効にする場合は、`SelectForUpdate` または `SelectForUpdateNoWAIT` を設定します。
useGetGeneratedKeys	java.lang.Boolean	False	JDBC3`java.sql.Statement.getGeneratedKeys()` メソッドで、autoincrement/sequence SQL フィールドから主キーに値を代入するかどうかを指定します。現在は、Borland JDataStore だけがこの文をサポートしています。
primaryKeyGenerationListener	java.lang.String	なし	`com.borland.ejb.pm.PrimaryKeyGenerationListener` インターフェースを実装し、主キーを生成するユーザー記述のクラスを指定します。
dbcAccesserFactory	java.lang.String	なし	アクセッサクラスインプリメンテーションを提供して `java.sql.ResultSet` から値を取得し、`java.sql.PreparedStatement` の値を設定できるファクトリクラス。
getPrimaryKeyBeforeInsertSql	java.lang.String	なし	主キー列名を提供する行を挿入する前に実行する SQL 文。
getPrimaryKeyAfterInsertSql	java.lang.String	なし	主キー列名を提供する行を挿入した後に実行する SQL 文。
useAlterTable	java.lang.Boolean	false	エンティティのテーブルを切り替えて、 一致列がないフィールドに列を追加する SQL ALTER 文を使用するかどうかを指定します。

プロパティ	型	デフォルト値	説明
createTableSql	java.lang.String	なし	テーブルを自動的に作成する必要がある場合、そのテーブルを作成するために使用される SQL 文。
create-tables	java.lang.Boolean	false	Borland CMP エンジンは、Cloudscape データベースと JDataStore Bean データベース、つまりデプロイメント環境で自動的に作成します。 ほかのデータベースで自動テーブルの作成を有効にするには、このフラグを True に設定します。

列プロパティ

プロパティ	型	デフォルト値	説明
ignoreOnInsert	java.lang.String	false	INSERT 文の実行時に設定しない列を指定します。 このプロパティは、getPrimaryKeyAfterInsertSql プロパティと組み合わせて使用します。
createColumnSql	java.lang.String	なし	標準データ型検索を上書きし、 手動でデータ型を指定するときにこのプロパティを使用します。 ■ javax.ejb.EJBContext の setRollbackOnly() および getRollbackOnly() メソッドをサポートします。 ■ データベース接続とトランザクションのタイムアウトをサポートします。 ■ パフォーマンスの観点から見ると軽量です。
columnJavaType	java.lang.String	なし	テーブルを自動的に作成するときに列を作成するための Java 型。 次の値を指定できます。 ■ java.lang.Boolean ■ java.lang.Byte ■ java.lang.Character ■ java.lang.Short ■ java.lang.Integer ■ java.lang.Long ■ java.lang.Float ■ java.math.BigDecimal ■ java.lang.String ■ java.sql.Time ■ java.sql.Date ■ java.sql.TimeStamp ■ java.io.Serializable createColumnSql を設定すると、このプロパティは無視されます。

セキュリティのプロパティ

次のセキュリティプロパティは、デプロイメントデスクリプタの <entity> 部分で指定します。

プロパティ	型	デフォルト値	説明
ejb.security.transportType	Enumerated	SECURE_ONLY	このプロパティは、 特定の EJB の保護品質を設定します。 CLEAR_ONLY に設定すると、クライアントは、この EJB に対してセキュリティで保護されていない接続だけを受け付けます。 EJB にメソッド許可が割り当てられていない場合は、これがデフォルト設定になります。 SECURE_ONLY に設定すると、クライアントは、この EJB に対してセキュリティで保護された接続だけを受け付けます。EJB に少なくとも 1 つのメソッド許可がある場合は、これがデフォルト設定になります。 ALL に設定すると、クライアントは、セキュリティで保護された接続とセキュリティで保護されていない接続の両方を受け付けます。 このプロパティの設定により、ServerQoPConfig ポリシーの転送値が制御されます。 詳細については、『プログラマーズリファレンス』の「セキュリティ API」の章を参照してください。
ejb.security.trustInClient	java.lang.Boolean	False	このプロパティは、 特定の EJB の保護品質を設定します。 true に設定すると、EJB コンテナは、クライアントに認証 ID を提供するように要求します。 メソッド許可が設定されていないメソッドが少なくとも 1 つ存在する場合は、このプロパティはデフォルトで false に設定されます。 そうでない場合は、 true に設定されます。 このプロパティの設定により、ServerQoPConfig ポリシーの転送値が制御されます。 詳細については、『プログラマーズリファレンス』の「セキュリティ API」の章を参照してください。

第 16 章

EJB-QL とデータアクセスサポート

EJB-QL では、オブジェクト指向クエリー言語の EJB-QL でクエリーを指定できます。Borland CMP エンジンはこれらのクエリーを SQL クエリーに変換します。Borland AppServer（AppServer）は、Sun Microsystems EJB 2.x 仕様に記載されている EJB-QL 機能のいくつかの拡張機能を提供します。

CMP フィールドまたは CMP フィールドのコレクションの選択

大きな EJB の 1 つの CMP フィールドだけが必要な場合は、EJB-QL を使用して、その CMP フィールドのコレクションの 1 つのインスタンスを選択できます。このように EJB-QL を使用することで、EJB 全体をロードする必要がなくなり、アプリケーションのパフォーマンスが向上します。たとえば、次のクエリーメソッドは、口座テーブルから残高フィールドだけを選択します。

```
<query>
   <query-method>
     <method-name>ejbSelectBalanceOfAccountLineItem</method-name>
      <method-params>
       <method-param>java.lang.Long</method-param>
   </method-params>
   </query-method>
   <result-type-mapping>Local</result-type-mapping>
   <ejb-ql>SELECT l.balance FROM Account a, IN (a.accountLineItem) l WHERE
l.lineItemId=?1</ejb-ql>
</query>
```

EJB-QL クエリーメソッドの戻り型は、次のようになります。

- CMP フィールドの Java 型がオブジェクト型の場合、クエリーメソッドは単一オブジェクトのクエリーメソッドであり、戻り型はそのオブジェクト型のインスタンスになります。
- CMP フィールドの Java 型がオブジェクト型であり、クエリーメソッドが複数のオブジェクトを返す場合、オブジェクト型のインスタンスのコレクションが返されます。
- CMP フィールドの Java 型がプリミティブ Java 型であり、SELECT メソッドが単一オブジェクトのメソッドである場合、戻り型はそのプリミティブ型になります。
- CMP フィールドの Java 型がプリミティブ Java 型であり、SELECT メソッドが複数のオブジェクトのメソッドである場合、ラップされた Java 型のコレクションが返されます。

結果セットの選択

1 つのクエリーメソッドで複数の CMP フィールドが返される場合、その戻り型は ResultSet 型にする必要があります。これにより、同じクエリーメソッド内の同じ EJB または複数の EJB から、複数の CMP フィールドを選択できます。続いて、その結果セットから目的のデータを抽出するコードを記述します。この機能は、Borland による CMP 2.x 仕様の拡張です。

EJB-QL の集計関数

集計関数には、MIN、MAX、SUM、AVG、および COUNT があります。MIN、MAX、SUM、および AVG 集計関数の場合、引数になるパス式は最後が CMP フィールドである必要があります。また、MAX、MIN、SUN、および AVG のデータベースクエリーは、その集計関数の引数に対応する行が存在しない場合は null 値を返します。戻り型がオブジェクト型の場合は、null が返されます。戻り型がプリミティブ型の場合、クエリー結果に値がないと、コンテナは `ObjectNotFoundException`（`FinderException` のサブクラス）を生成します。

COUNT 関数のパス式の最後は、CMP フィールドまたは CMR フィールドのいずれか、または、ID 変数にすることができます。

たとえば、次の EJB-QL 集計関数は、CMP フィールドで終了しています。

```
<query>
   <query-method>
     <method-name>ejbSelectMaxLineItemId</method-name>
      <method-params>
       <method-param>java.lang.String</method-param>
      </method-params>
   </query-method>
   <result-type-mapping>Local</result-type-mapping>
   <ejb-ql>SELECT MAX(l.lineItemId) FROM Account AS a, IN (a.accountLineItem) l
WHERE l.accountId=?1</ejb-ql>
</query>
```

集計関数には、次の制限があります。

- SUM および AVG 関数の引数には、数値（Integer、Byte、Long、Short、Double、Float、および BigDecimal 型）を指定する必要があります。
- MAX および MIN 関数の引数は、ソート可能な CMP フィールド型（数値、文字列、文字、および日付）に対応する必要があります。
- COUNT 関数の引数を形成するパス式は、CMP フィールドまたは CMR フィールドのいずれかで終了する必要があります。COUNT 関数を使って CMR フィールドのコレクションのサイズを決定すると、アプリケーションのパフォーマンスが大幅に向上します。

集計関数データの戻り型

次の表に、単一オブジェクトを選択する EJB-QL でさまざまな集計関数の引数として使用できるデータ型と返されるデータ型を示します。

複数のオブジェクトを選択する集計関数は、返される Java データ型のラップされたコレクションを返します。

集計関数	引数のデータ型	使用される戻り型
MIN, MAX, SUM	java.lang.Integer	java.lang.Integer
AVG	java.lang.Integer	java.lang.Double
COUNT	java.lang.Integer	java.lang.Long
MIN, MAX, SUM	java.lang.Integer	java.lang.Integer
AVG	java.lang.Integer	java.lang.Double
COUNT	java.lang.Integer	java.lang.Long
MIN, MAX, SUM	java.lang.Byte	java.lang.Byte

集計関数	引数のデータ型	使用される戻り型
AVG	java.lang.Byte	java.lang.Double
COUNT	java.lang.Byte	java.lang.Long
MIN, MAX, SUM	java.lang.Byte	java.lang.Byte
AVG	java.lang.Byte	java.lang.Double
COUNT	java.lang.Byte	java.lang.Long
MIN, MAX, SUM	java.lang.Long	java.lang.Long
AVG	java.lang.Long	java.lang.Double
COUNT	java.lang.Long	java.lang.Long
MIN, MAX, SUM	java.lang.Long	lonjava.lang.Long
AVG	java.lang.Long	java.lang.Double
COUNT	java.lang.Long	java.lang.Long
MIN, MAX, SUM	java.lang.Short	java.lang.Short
AVG	java.lang.Short	java.lang.Double
COUNT	java.lang.Short	java.lang.Long
MIN, MAX, SUM	java.lang.Short	java.lang.Short
AVG	java.lang.Short	java.lang.Double
COUNT	java.lang.Short	java.lang.Long
MIN, MAX, SUM	java.lang.Double	java.lang.Double
AVG	java.lang.Double	java.lang.Double
COUNT	java.lang.Double	java.lang.Long
MIN, MAX, SUM	java.lang.Double	java.lang.Double
AVG	java.lang.Double	java.lang.Double
COUNT	java.lang.Double	java.lang.Long
MIN, MAX, SUM	java.lang.Float	java.lang.Float
AVG	java.lang.Float	java.lang.Double
COUNT	java.lang.Float	java.lang.Long
MIN, MAX, SUM	java.lang.Float	java.lang.Float
AVG	java.lang.Float	java.lang.Double
COUNT	java.lang.Float	java.lang.Long
MIN, MAX, SUM	java.math.BigDecimal	java.math.BigDecimal
AVG	java.math.BigDecimal	java.lang.Double
COUNT	java.math.BigDecimal	java.lang.Long
MIN, MAX	java.lang.String	java.lang.String
COUNT	java.lang.String	java.lang.Long
MIN, MAX	java.util.Date	java.util.Date
COUNT	java.util.Date	java.lang.Long
MIN, MAX	java.sql.Date	java.sql.Date
COUNT	java.sql.Date	java.lang.Long
MIN, MAX	java.sql.Time	java.sql.Time
COUNT	java.sql.Time	java.lang.Long
MIN, MAX	java.sql.Timestamp	java.sql.Timestamp
COUNT	java.sql.Timestamp	java.lang.Long

ORDER BY のサポート

EJB 2.0 仕様は、EJB-QL で SELECT、FROM、および WHERE の 3 つの SQL 節をサポートします。

また、Borland CMP エンジンは、同じ EJB-QL 文の中で WHERE 節の**後**に置かれている SQL 節 ORDER BY をサポートします。これは、<ejb-ql> エンティティの標準の ejb-jar.xml デプロイメントデスクリプタでサポートされます。たとえば、次の EJB-QL 文は、Customer Bean からオブジェクトを個別に選択し、LNAME フィールドで並べ替えます。

```
<query>
 <description></description>
 <query-method>
  <method-name>findCustomerByNumber</method-name>
  <method-params />
 <ejb-ql>SELECT Distinct Object(c) from CustomerBean c WHERE c.no > 1000 ORDER
BY c.LNAME</eql-ql>
 </query-method>
<query>
```

EJB-QL では、ASC（昇順）または DESC（降順）も指定できます。どちらも指定しない場合は、デフォルトの昇順で並べられます。

たとえば、次の表を参照してください。

NAME（名前）	DEPARTMENT（部署）	SALARY（給与）	HIRE DATE（採用日）
Timmy Twitfuller	Mail Room	1000	1/1/01
Sam Mackey	The Closet with the Light Out	800	1/2/02
Ralph Ossum	Coffee Room	900	1/4/01

クエリー：

```
SELECT OBJECT(e) FROM EMPLOYEE e ORDER BY e.HIRE_DATE
```

上記のクエリーは、次の結果を生成します。

NAME（名前）	DEPARTMENT（部署）	SALARY（給与）	HIRE DATE（採用日）
Timmy Twitfuller	Mail Room	1000	1/1/01
Ralph Ossum	Coffee Room	900	1/4/01
Sam Mackey	The Closet with the Light Out	800	1/2/02

GROUP BY のサポート

GROUP BY 節は、SELECT オペレーションが実行される前に、結果テーブルの行をグループ化します。次の表を参照してください。

NAME（名前）	DEPARTMENT（部署）	SALARY（給与）	HIRE DATE（採用日）
Mike Miller	Mail Room	1200	11/18/99
Timmy Twitfuller	Mail Room	1000	1/1/01
Buddy	Coffee Room	1000	4/13/97
Sam Mackey	The Closet with the Light Out	800	1/2/02
Todd Whitmore	The Closet with the Light Out	900	4/12/01
Ralph Ossum	Coffee Room	900	1/4/01

次のように、単一クエリーメソッドを使用して、各部署の平均給与を取得できます。

```
SELECT e.DEPARTMENT, AVG(e.SALARY) FROM EMPLOYEE e GROUP BY e.DEPARTMENT
```

結果は次のようになります。

DEPARTMENT（部署）	AVG(SALARY)（平均月給）
Coffee Room	950
Mail Room	1100
The Closet with the Light Out	850

サブクエリー

サブクエリーは、クエリー対象のデータベースのインプリメンテーションで許可されている深さまで実行できます。たとえば、次の ejb-jar.xml で指定されるサブクエリー（太字）を使用できます。このサブクエリーには ORDER BY も含まれており、結果は降順（DESC）で返されます。

```
<query>
   <query-method>
     <method-name>findApStatisticsWithGreaterThanAverageValue</method-name>
     <method-params />
   </query-method>
     <ejb-ql>SELECT Object(s1) FROM ApStatistics s1 WHERE s1.averageValue >
SELECT AVG(s2.averageValue) FROM ApStatistics s2 ORDER BY s1.averageValue
DESC</ejb-ql>
</query>
```

サブクエリーの正しい使い方については、データベースインプリメンテーションのマニュアルを参照してください。

ダイナミッククエリー

場合によっては、さまざまな基準に基づいてデータを動的に検索する必要があります。しかし、EJB-QL クエリーはそのような状況をサポートしていません。EJB-QL クエリーはデプロイメントデスクリプタで指定されるため、クエリーを変更する場合は Bean を再デプロイメントする必要があります。AppServer には、Bean コードで EJB-QL クエリーをプログラムによって動的に作成および実行できるダイナミッククエリー機能が備わっています。

ダイナミッククエリーを使用することには、次の利点があります。

- EJB を更新してデプロイメントしなくても、新しいクエリーを作成して実行できます。
- EJB のデプロイメントデスクリプタファイルのサイズが小さくなります。これは、検索クエリーがデプロイメントデスクリプタで静的に定義されるのではなく、動的に作成されるからです。

ダイナミッククエリーは、デプロイメントデスクリプタに追加する必要はありません。これらは、動的な `ejbSelects` の Bean クラスで宣言するか、動的検索用のローカルまたはリモートホームインターフェースで宣言します。

ダイナミッククエリーの検索メソッドは、次のとおりです。

```
public java.util.Collection findDynamic(java.lang.String ejbql,
    Class[] types, Object[] args)
  throws javax.ejb.FinderException

public java.util.Collection findDynamic(java.lang.String ejbql,
    Class[] types, Object[] args, java.lang.String sql)
  throws javax.ejb.FinderException
```

ダイナミッククエリーの `ejbSelect` は、次のとおりです。

```
public java.util.Collection selectDynamicLocal(java.lang.String ejbql,
    Class[] types, Object[] params)
     throws javax.ejb.FinderException

public java.util.Collection selectDynamicLocal(java.lang.String ejbql, Class[]
types, Object[] params, java.lang.String sql)
     throws javax.ejb.FinderException

  public java.util.Collection selectDynamicRemote(java.lang.String ejbql,
    Class[] types, Object[] params)
     throws javax.ejb.FinderException

public java.util.Collection selectDynamicRemote(java.lang.String ejbql, Class[]
types, Object[] params, java.lang.String sql)
     throws javax.ejb.FinderException

  public java.sql.ResultSet selectDynamicResultSet(java.lang.String ejbql,
    Class[] types, Object[] params)
     throws javax.ejb.FinderException

public java.sql.ResultSet selectDynamicResultSet(java.lang.String ejbql,
     Class[] types, Object[] params, java.lang.String sql)
     throws javax.ejb.FinderException
```

次のように指定します。

- **`java.lang.String ejbql`**：これは、実際の EJB-QL 構文を表します。
- **`Class[] types`**：この配列は、選択メソッドまたは検索メソッドにパラメータのクラス型を提供します（パラメータがない場合は、空の配列を指定できます）。
- **`Object[] params`**：この配列は、パラメータの実際の値を提供します。これは、通常の選択メソッドまたは検索メソッドのパラメータ引数と同じです。

  動的な選択または検索メソッドの戻り型は、`selectDynamicResultSet` 以外は常に `java.util.Collection` です。クエリーから返されたオブジェクトのインスタンスまたは値の型が 1 つである場合は、それがコレクションの最初のメンバーになります。ダイナミッククエリーは、通常のクエリーと同じ規則にしたがいます。
- **`java.lang.String sql`**：ユーザー指定の SQL。指定すると、EJB-QL によって生成された SQL を上書きします。

メモ デプロイメントデスクリプタでは、ダイナミッククエリーに関連する 8 つのメソッドを使用した跡を残さないようにします。

EJB-QL から生成された SQL を CMP エンジンで上書きする

重要 この機能は、詳しい知識を持つユーザー向けです。

Borland CMP エンジンは、デプロイメントデスクリプタに入力された EJB-QL に基づいて、データベースに対する SQL 呼び出しを生成します。データベースインプリメンテーションによっては、生成される SQL が最適でない場合があります。生成された SQL は、補助ストアインプリメンテーションで提供されるツールまたは別の開発ツールを使って取得できます。生成された SQL が最適でない場合は、独自の SQL に置き換えることができます。ただし、ユーザー SQL に対する検証は行われません。

メモ SQL に問題があると、例外が生成され、それによってシステムがクラッシュすることがあります。

Borland 専用のデプロイメントデスクリプタ `ejb-borland.xml` で、独自の最適化された SQL を指定できます。XML の文法は `ejb-jar.xml` の文法と同じですが、`<ejb-ql>` 要素が `<user-sql>` 要素に置き換えられています。この専用の要素には、CMP エンジンが生成した SQL ではなく、データベースにアクセスするために使用される SQL-92 文（EJB-QL 文**ではない**）が含まれます。

重要 この文の `SELECT` 節は、Borland CMP エンジンによって生成された `SELECT` 節と同じである必要があります。

その後の節は、ユーザーが最適化できます。`SELECT` 節のフィールドの順序は、CMP エンジン固有の順序であり、この順序を維持する必要があります。

次に例を示します。

```
<entity>
 <ejb-name>EmployeeBean</ejb-name>
 ...
 <query>
  <query-method>
   <method-name>findWealthyEmployees</method-name>
   <method-params />
  </query-method>
  <user-sql>SELECT E.DEPT_NO, E.EMP_NO, E.FIRST_NAME, E.FULL_NAME,
          E.HIRE_DATE, E.JOB_CODE, E.JOB_COUNTRY,
          E.JOB_GRADE, E.LAST_NAME, E.PHONE_EXT, E.SALARY
       FROM EMPLOYEE E WHERE E.SALARY > 200000
  </user-sql>
 </query>
 ...
</entity>
```

メモ さまざまな `SELECT` 文により、CMP エンジンが生成する SQL の型が示されています。

CMP エンジンは、`ejb-jar.xml` デプロイメントデスクリプタ内で EJB-QL 文を検出すると、`ejb-borland.xml` をチェックして、同じ Bean のデスクリプタ内にユーザー SQL があるかどうかを確認します。

ユーザー SQL がない場合、CMP エンジンは独自の SQL を生成して実行します。

`ejb-borland.xml` デスクリプタは、クエリー要素がある場合、`<user-sql>` タグ内の SQL をかわりに使用します。

重要 `ejb-borland.xml` 内の `<query>` 要素は、標準の `ejb-jar.xml` デプロイメントデスクリプタ内の `<query>` 要素を**置き換えません**。CMP エンジンの SQL を上書きする場合は、**両方の**デスクリプタに要素を指定する必要があります。

コンテナ管理データアクセスサポート

CMP に対して、Borland EJB コンテナは、JDBC 仕様によってサポートされているすべてのデータ型をサポートします。また、JDBC によってサポートされていないデータ型もいくつかあります。

次の表に、Borland EJB コンテナがサポートする基本型と複合型をまとめます。

- 基本型
  - `boolean Boolean`
  - `double Double`
  - `long Long`
  - `BigDecimaljava.util.Date`
  - `byte Byte`
  - `float Float`
  - `short Short`
  - `byte[]`
  - `char Character`
  - `int Integer`
  - `String java.sql.Date`
  - `java.sql.Time java.sql.TimeStamp`
- 複合型
  - `java.io.Serializable` を実装する任意のクラス（Vector や Hashtable など）
  - その他のエンティティ Bean リファレンス

メモ 現在、Borland CMP エンジンは、日付に対して Long 値の型、`java.util.Date` に対して `java.sql.Date` をサポートしています。

Borland コンテナは、`java.io.Serializable` インターフェースを実装するクラス（Hashtable や Vector など）をサポートすることに注意してください。コンテナでは、Java コレクションやサードパーティコレクションなどもサポートしています。これらも `java.io.Serializable` を実装するからです。`Serializable` インターフェースを実装するクラスとデータ型に対して、Borland コンテナは、単にそれらの状態をシリアライゼーションし、その結果を `BLOB` に格納するだけです。Borland コンテナでは、これらのクラスや型に対して何らかのマッピングを行うことはなく、単にバイナリ形式でそれらの状態を格納します。Borland コンテナの CMP エンジンには、明示的にサポートされていないすべての型を `BLOB` としてシリアライズするという規則が適用されます。

データベースのインプリメンテーションによって、次のデータ型は列インデックスに基づいて取得する必要があります。

データベース	データ型
Oracle	■ LONG RAW
Sybase	■ NTEXT ■ IMAGE
MS SQL	■ NTEXT ■ IMAGE

メモ `BINARY`（MS SQL）または `RAW`（Oracle）の 2 つのデータ型のいずれかを主キーとして使用する場合は、サイズを明示的に指定する必要があります。

Oracle ラージオブジェクト（LOB）のサポート

ラージオブジェクト（LOB）には、バイナリラージオブジェクト（BLOB）とキャラクタラージオブジェクト（CLOB）の 2 種類があります。

BLOB は、次のデータ型で CMP フィールドにマッピングされます。

- byte[]
- java.io.Serializable
- java.io.InputStream

CLOB は、「*キャラクタ*」ラージオブジェクトなので、java.lang.String データ型を使って CMP フィールドにマッピングする必要があります。

デフォルトでは、Borland CMP エンジンは自動的に CMP フィールドを LOB にマッピングしません。LOB データ型を使用する場合は、`ejb-borland.xml` デプロイメントデスクリプタで明示的に CMP エンジンに通知する必要があります。通知するには、列プロパティ `createColumnSql` を設定します。次に例を示します。

```
<column-properties>
 <column-name>CLOB-column</column-name>
 <property>
  <prop-name>createColumnSql</prop-name>
  <prop-type>String</prop-type>
  <prop-value>CLOB</prop-value>
 </property>
</column-properties>

<column-properties>
 <column-name>BLOB-column</column-name>
 <property>
  <prop-name>createColumnSql</prop-name>
  <prop-type>String</prop-type>
  <prop-value>BLOB</prop-value>
 </property>
</column-properties>
```

コンテナによって作成されるテーブル

`create-tables` を有効にすることで、エンティティのコンテナ管理のフィールドに基づき、自動的にコンテナ管理のエンティティに対応するテーブルを作成するように Borland EJB コンテナに指示できます。テーブルの作成とデータ型のマッピングはベンダーによって異なるため、デプロイメントデスクリプタでコンテナに JDBC データベースダイアレクトを指定する必要があります。JDataStore 以外のデータベースでは、`create-tables` プロパティが `true` に設定されている場合、ダイアレクトを指定すると、コンテナがコンテナ管理のエンティティに対するテーブルを自動的に作成します。ダイアレクトを指定しない限り、コンテナはこれらのテーブルを作成しません。

次の表に、さまざまなダイアレクトの名前と値を示します。値の大文字と小文字は区別しません。

データベース名	ダイアレクト値
JDataStore	jdatastore
Oracle	oracle
Sybase	sybase
MSSQLServer	mssqlserver
DB2	db2
Interbase	interbase
Informix	informix

第 17 章

エンティティ Bean の主キーの生成

各エンティティ Bean には、Bean インスタンスを識別する一意の主キーを割り当てます。主キーは、RMI-IIOP で有効な値の型を備えた Java クラスで表すことができます。したがって、主キーは `java.io.Serializable` インターフェースを拡張します。また、主キーは、`Object.equals(Object other)` および `Object.hashCode()` メソッドのインプリメンテーションも提供する必要があります。

通常は、エンティティ Bean の主キーフィールドは、`ejbCreate()` メソッド内で設定します。これらのフィールドは、データベースに新しいレコードを挿入するときに使用します。ただし操作が難しく、メソッドも肥大化します。したがって、データベースの多くは、今では内蔵メカニズムによって適切な主キーの値を提供するようになっています。主キーの生成方法としてより洗練された方法には、主キーを生成するクラスをユーザーに別途実装する方法があります。このクラスでは、主キーを生成するためのデータベース固有のプログラミングロジックも生成できます。

主キーの生成方法としては、手動による方法、カスタムクラスを使用する方法、そしてコンテナを利用しデータベースツールで生成する方法があります。カスタムクラスを使用する場合は、次に説明する `com.borland.ejb.pm.PrimaryKeyGenerationListener` インターフェースを実装してください。データベースツールを使用する場合は、データベースベンダーに応じて CMP エンジンのプロパティを設定し主キーを生成します。

ユーザークラスから主キークラスを生成

エンタープライズ Bean により、一意のデータを持つ Java クラスで主キーが表されます。この主キークラスは、RMI-IIOP の有効な値型であればクラスは問いません。したがって、主キークラスは `java.io.Serializable` インターフェースを拡張します。また、主キーは `Object.equals(Object other)` メソッドと `Object.hashCode()` メソッドのインプリメンテーションも提供します。この 2 つのメソッドは、当然ながらすべての Java クラスが継承します。

カスタムクラスから主キークラスを生成

カスタムクラスから主キーを生成するには、`com.borland.ejb.pm.PrimaryKeyGenerationListener` インターフェースを実装するクラスを作成します。

メモ これは、主キー生成用の新しいインターフェースです。Borland AppServer の前バージョンでは、このクラスは `com.inprise.ejb.cmp.PrimaryKeyGenerator` でした。このインターフェースはまだサポートされていますが、できれば新しいインターフェースを使用してください。

次に、カスタムクラスを使用する意図をコンテナに伝え、エンティティ Bean の主キーを生成します。それには、プロパティ `primaryKeyGenerationListener` を主キージェネレータのクラス名に設定します。

CMP エンジンによる主キーの実装

主キーの生成は、CMP エンジンでも実装できます。Borland では 4 つのプロパティで、データベース固有の機能を利用した主キー生成をサポートしています。次にそれらのデプロイメント情報について説明します。

- `getPrimaryKeyBeforeInsertSql`
- `getPrimaryKeyAfterInsertSql`
- `ignoreOnInsert`
- `useGetGeneratedKeys`

列プロパティである `ignoreOnInsert` 以外はどのプロパティもテーブルプロパティです。

Oracle シーケンス：getPrimaryKeyBeforeInsertSql を使用

プロパティ `getPrimaryKeyBeforeInsertSql` は、一般には Oracle シーケンスと併用します。このプロパティの値は、シーケンスから生成される主キーを選択するための SQL 文です。たとえば、プロパティを次のように設定します。

```
SELECT MySequence.NEXTVAL FROM DUAL
```

CMP エンジンは、この SQL を実行し、ResultSet から適切な値を抽出します。この値は、後続の `INSERT` を実行するときの主キーとして使用します。ResultSet からの抽出は、主キーの型によって異なります。

SQL サーバー：getPrimaryKeyAfterInsertSql と ignoreOnInsert を使用

SQL サーバーを使用する場合は、プロパティを 2 つ指定します。`INSERT` の実行後に SQL を実行するように、`getPrimaryKeyAfterInsertSql` プロパティで指定しました。上記のように、CMP エンジンは、主キーの型に基づいて ResultSet から主キーを抽出します。プロパティ `ignoreOnInsert` も、アイデンティティ列の名前に設定します。`INSERT` にその列が設定されていないことが CMP エンジンに伝えられます。

JDataStore JDBC3: useGetGeneratedKeys を使用

Borland の JDataStore は、新しい JDBC3 メソッド `java.sql.Statement.getGeneratedKeys()` をサポートしています。このメソッドでは、新しく挿入された行から主キー値を取得します。これ以外のコーディングは必要ありませんが、このメソッドはほかのデータベースではサポートされていないので、使用するのは Borland JDataStore に限定してください。このメソッドを使用するには、論理プロパティ `useGetGeneratedKeys` を `True` に設定します。

名前付きシーケンステーブルを使用した主キーの自動生成

基底のデータベース（Oracle SEQUENCE など）と JDBC ドライバ（JDBC 3.0 での AUTOINCREMENT）がキーの生成をサポートしていない場合は、名前付きシーケンステーブルを使って主キーの自動生成をサポートします。名前付きシーケンステーブルでは、主キーの生成に使用するキーを保持するテーブルを指定できます。コンテナは、このテーブルを使ってキーを生成します。

テーブルは、列と行がそれぞれ 1 つである必要があります。

名前付きシーケンステーブルを使用するには、テーブルの行と列がそれぞれ 1 つで、値は（シーケンス値として）整数である必要があります。任意の整数値からなる「SEQUENCE」という 1 つの列を持つテーブルを作成する必要があります。次に例を示します。

```
CREATE TABLE TAB_A_SEQ (SEQUENCE int);
INSERT into TAB_A_SEQ values (10);
```

この例では、キーの生成は値 10 から始まります。

この機能を有効にするには、ejb-borland.xml の <column-properties> に次のように設定します。

```
<table-properties>
        <table-name>TABLE_A</table-name>
        <column-properties>
       <column-name>ID</column-name>
            <property>
                <prop-name>autoPkGenerator</prop-name>
                <prop-type>java.lang.String</prop-type>
                <prop-value>NAMEDSEQUENCETABLE</prop-value>
            </property>
       <property>
                <prop-name>namedSequenceTableName</prop-name>
                <prop-type>java.lang.String</prop-type>
                <prop-value>TAB_A_SEQ</prop-value>
            </property>
            <property>
                <prop-name>keyCacheSize</prop-name>
                <prop-type>java.lang.Integer</prop-type>
                <prop-value>2</prop-value>
            </property>
            </column-properties>
.....
    </table-properties>
```

「ID」は主キーの列です。これは、NAMEDSEQUENCETABLE を使って auto Pk Generation のマークが付けられています。使用するテーブルは TAB_A_SEQ です。

メモ getPrimaryKeyAfterInsert または useGetGeneratedKeys の使用時には、ejb.CacheCreate プロパティを false に設定します。コンテナは、Bean インスタンスの呼び出しをディスパッチするために主キーを知る必要があります。したがって、Create メソッドが戻ると同時に主キーを知る必要があります。

キーキャッシュサイズ

主キーの生成時に、コンテナはキーをデータベース内のテーブルから取得します。キーキャッシュサイズを指定すると、データベースへのアクセスが減るため、パフォーマンスを改善することができます。この機能を使用するには、データベースが取得する主キー値の数を指定するために、ejb-borland.xml ファイルで <key-cache-size> 要素を設定します。キャッシュサイズの値が 1 より大きい場合、コンテナは、このキーの数を使って主キーを生成します。

キーキャッシュのサイズが指定されていない場合のデフォルト値は 1 です。キーキャッシュのサイズはオプションですが、1 より大きい値を指定してパフォーマンスの最適化に利用することをお勧めします。

メモ コンテナが再起動された場合、またはクラスタモードで使用される場合、生成されるキーにギャップが生じる場合があります。

第 18 章

トランザクション管理

この章では、トランザクションを処理する方法について説明します。

トランザクションの概要

トランザクションをサポートする Java 2 Enterprise Edition（J2EE）などのプラットフォームを使用すると、アプリケーションを効率よく開発できます。トランザクション対応のシステムでは、障害回復やマルチユーザープログラミングなどの複雑な問題からプログラマが解放され、アプリケーション開発が簡単になります。トランザクションは、1 つのデータベースや 1 つのサイトに限定されません。分散トランザクションでは、複数のサイトにわたって複数のデータベースを同時に更新することができます。

通常、プログラマは、アプリケーションで行う作業を複数の単位に分割します。このそれぞれの作業単位がトランザクションのことです。アプリケーションの実行中、基底のシステムでは、各作業単位（各トランザクション）は、ほかの処理に邪魔されずに完了します。一部のトランザクションが完了しなかった場合、基盤システムはトランザクションをロールバックして、そのトランザクションで実行されたすべての作業を元に戻します。

トランザクションの特性

一般に、トランザクションとは、データベースなどの共有リソースにアクセスする操作を意味します。すべてのデータベースアクセスは、トランザクションのコンテキスト内で実行されます。どのトランザクションにも、次の特性が共通してあります。

- 原子性（Atomicity）
- 一貫性（Consistency）
- 分離性（Isolation）
- 耐久性（Durability）

これらの特性をまとめて、ACID という略称で呼びます。

多くの場合、トランザクションは複数の操作で構成されます。原子性とは、そのトランザクション内のすべての操作が実行されたか、または何も実行されないことで、トランザクションが完了したとみなされることです。トランザクションの一部の操作を実行できなかった場合は、すべての操作が実行されません。

一貫性とは、リソースの一貫性を意味します。データベースはトランザクション中に、ある一貫した状態から別の一貫した状態に移行する必要があります。トランザクションでは、データベースのセマンティクスの整合性と物理的な整合性を維持する必要があります。

分離性とは、各トランザクションが現在データベースを操作している唯一のトランザクションであるように見えることです。ほかのトランザクションが同時に実行されている可能性もあります。ただし、各トランザクションは、ほかのトランザクションが正常に完了して結果をコミットするまで、ほかのトランザクションが操作中のデータを見ることはありません。ほかのトランザクションによる更新の一部だけが見えると、更新内容が相互に依存する場合、データベースが一貫していないように見える可能性があります。分離性により、各トランザクションはこのようなデータの矛盾から保護されます。

トランザクションの分離性は、データベースで許可される並行処理のレベルによって変わります。分離レベルが高くなるほど、並行性が制限されます。すべてのトランザクションをシリアライゼーションできる場合、分離レベルは最大になります。この場合、データベースの内容は、各トランザクションがそれぞれ単独に実行され、ほかのトランザクションと重なることがないように見えます。しかし、アプリケーションによっては、分離レベルを下げて並行性を高めることができます。このようなアプリケーションでは多くのトランザクションが同時に実行されるため、各トランザクションは、部分的に更新された一貫しないデータを読み取る可能性があります。

耐久性とは、障害などが発生した場合でも、コミットされたトランザクションによる更新内容がデータベース中で存続する必要があるということです。耐久性により、コミットされた更新がコミット操作後の障害に関係なく存続することと、システムまたはメディアの障害からデータベースを回復できることが保証されます。

トランザクションのサポート

BorlandAppServer （AppServer）は、フラットなトランザクションをサポートしますが、ネストされたトランザクションはサポートしません。トランザクションは、暗黙的に伝達されます。つまり、ユーザーがトランザクションコンテキストをパラメータとして明示的に渡す必要はありません。J2EE コンテナがクライアントのためにこの作業を透過的に処理するからです。

トランザクション管理は、プログラムから標準の JTS または JTA API を呼び出すことで実行できます。また、Enterprise JavaBeans（EJB）などの J2EE コンポーネントを記述する際に推奨される別の方法としては、J2EE コンテナが透過的にトランザクションを開始および停止する宣言的なトランザクションを使用します。

トランザクションマネージャサービス

AppServer では、次の 2 つのトランザクションマネージャ（またはエンジン）を使用できます。

- トランザクションマネージャ（旧名：パーティショントランザクションサービス）
- OTS（旧名：2PC トランザクションサービス）

トランザクションマネージャは、各 AppServer パーティション内に存在します。これは、CORBA トランザクションサービス仕様の Java によるインプリメンテーションです。トランザクションマネージャは、トランザクションタイムアウトと 1 フェーズコミットプロトコルをサポートします。特殊な環境では、2 フェーズコミットプロトコルでも使用できます。

トランザクションマネージャは、次のような場合に使用してください。

- 1 フェーズコミットプロトコルを使用する場合。
- パフォーマンスを改善する場合。現在、インプロセスに設定できるのは、トランザクションマネージャだけです。トランザクション管理 API とその他のトランザクションコンポーネントは、インプロセスの JVM 呼び出しを行うため、トランザクションマネージャは OTS エンジンよりかなり高速になります。
- 2 フェーズコミットプロトコルを使用し、トランザクションの回復を考慮しない場合。Enterprise JavaBeans のデプロイメント時にビジネスロジックをチェックする場合などは、トランザクションの回復は必要ありません。2 フェーズコミットでトランザクションマネージャを使用する場合は、AppServer 管理コンソールのパーティション内に表示される［ Transaction Manager］の［ Properties］で、［ Allow unrecoverable completion］プロパティを true に設定する必要があります。または、パーティションの `EJBAllowUnrecoverableCompletion` システムプロパティを設定することもできます。

OTS エンジンは、独立したアドレス空間に存在します。分散トランザクション CORBA アプリケーションに完全なソリューションを提供します。OTS エンジンは VisiBroker ORB 上に実装され、単一の統合アーキテクチャで基本的なサービスを提供して、分散トランザクションを単純化します。提供されるサービスには、トランザクションサービス、回復、ログ、データベースとの統合、管理機能などがあります。

分散トランザクションと 2 フェーズコミット

Borland EJB コンテナでは、分散トランザクションを扱うことができます。分散トランザクションとは、複数のシステム、プラットフォーム、および Java 仮想マシン（JVM）にまたがるトランザクションです。

複数のリソースにわたってデータを操作するトランザクションでは、2 フェーズコミットプロセスを使用します。このプロセスでは、トランザクションに関連するすべてのリソースがトランザクションによって正しく更新されます。一部のリソースを更新できない場合は、どのリソースも更新されません。

メモ AppServer では、2 フェーズコミットトランザクションがサポートされていますが、リモートプロシージャコール（RPC）の数が多く、負荷が大きくなることが避けられないため、必要なときにだけ使用してください。次の節の 156 ページの「2 フェーズコミットトランザクションを使用する場合」を参照してください。

2 フェーズコミットには 2 つの手順があります。最初の手順は、準備フェーズです。このフェーズでは、トランザクションサービスは、トランザクションに関連する各リソースが更新の準備を行うように要求し、その更新をコミットできるかどうかをトランザクションサービスに通知します。2 番めのステップは、コミットフェーズです。トランザクションサービスは、すべてのリソースが更新プロセスを実行できると通知した場合にだけ、実際に更新を開始します。いずれかのリソースで更新を実行できないことが通知された場合、トランザクションサービスはほかのすべてのリソースに、そのトランザクションに関連するすべての更新をロールバックするように指示します。

トランザクションマネージャと OTS エンジンは、異種分散（2 フェーズコミット）トランザクション、および異種リソース用の 2 フェーズコミットの両方をサポートします。

デフォルトでは、トランザクションマネージャは、1 つのグローバルトランザクションに複数のリソースが関与することは許可しませんが、リカバリ不可トランザクションの実行をサポートすることにより、複数のリソースが関与できるように設定できます。これは、管理コンソールから［Allow unrecoverable completion］オプションを設定するか（トランザクションマネージャを右クリックし、［Properties］を選択）、パーティションのシステムプロパティ `EJBAllowUnrecoverableCompletion` を設定して、トランザクションマネージャで有効にできます。［Allow unrecoverable completion］を有効にすると、トランザクションコミットプロセス時に、コンテナは、関与するすべてのリソースに対して 1 フェーズコミット呼び出しを実行します。［Allow unrecoverable completion］を有効にすると、トランザクションが完了する前にエラーが発生しても回復することができず、関与するリソースに不整合が発生する可能性があるため、慎重に使用してください。

異種 2 フェーズコミットトランザクションをサポートするには、基底のリソースの XA サポートに OTS エンジンを統合する必要があります。DBMS ベンダーによる XA 対応の JDBC ドライバ、およびメッセージサービスプロバイダが提供する JMS サポートにより、EJB コンテナと OTS エンジンは、単一トランザクションに複数のリソースを関与させることができます。

同種データベースに対して 2 フェーズコミットを行う場合は、DBMS サーバーを設定する必要があります。この場合、最初のデータベースへのコミットを制御するのはコンテナですが、それ以降のデータベースへのコミットは、DBMS に組み込まれているトランザクションコーディネータを使用して、DBMS サーバーが制御します。詳細については、各ベンダーによって提供される DBMS サーバーのマニュアルを参照してください。

2 フェーズコミットトランザクションを使用する場合

パフォーマンスが高い分散アプリケーションを構築するための基本的な 1 つの方法は、リモートプロシージャコール（RPC）の数を制限することです。ここでは、2 フェーズコミットトランザクションを使用する場合としない場合の一般的な状況について説明します。必要がない場合は 2 フェーズコミットトランザクションの使用を回避すると、XAResource オブジェクトおよび OTS エンジンに関連する不要な RPC が使用されなくなるため、アプリケーションのパフォーマンスが大幅に向上します。

同じトランザクション内で複数の JDBC 接続を使って 1 つのベンダーの複数のデータベースリソースにアクセスする場合

1 つのベンダーによる複数のデータベースにアクセスするシナリオでは、多くの場合、2 フェーズコミットの使用を回避できます。1 つのデータベースにアクセスし、最初のデータベースへの接続を介してトンネリングアクセスすることで、2 つ目のデータベースにアクセスできます。Oracle などの DBMS にはこの機能が備わっています。この場合、AppServer パーティションは、「前面」のデータベースへの 1 つの JDBC 接続だけで設定できます。「背後」のデータベースへのアクセスは、最初の JDBC 接続をトンネリングします。

同じトランザクション内で同じデータベースリソースへの複数の JDBC 接続を使用する場合

単一トランザクション内で、1 つのデータベースに対して複数のリソースによって複数の JDBC 接続が取得および使用される場合は、2 フェーズコミットの使用を回避できます。JDBC 接続は、XA データソースから取得する必要があります。ただし、関係するリソースが 1 つだけなので、2 フェーズコミットではなく、1 フェーズコミットを使ってトランザクションを実行できます。これは、OTS エンジンではなく、トランザクションマネージャを使用することで実現できます。もう 1 つの方法としては、トランザクションに関連するすべての EJB を分散したパーティションに配置するのではなく、まとめて配置します。この場合は、使用されるのは XA 以外のデータソースであり、2 フェーズコミットは必要ありません。

単一トランザクション内で複数の異種リソースを使用する場合

この場合は、2 フェーズコミットトランザクションを使用する必要があります。このような状況は、1 つのトランザクションで Oracle と Sybase の両方を処理している場合や、Oracle データベースと MQSeries などの JMS プロバイダへのアクセスを含むトランザクションの場合などです。後者の場合、トランザクションは JTA XAResource オブジェクトを使って調整され、Oracle の場合は JDBC を介して、MQSeries の場合は JMS を介して取得されて、両方のリソースが 2 フェーズコミットトランザクションの実行に使用されます。OTS エンジンが提供する 2 フェーズコミット機能は、単一トランザクションで複数の互換性のないリソースにアクセスする場合にだけ必要です。

メモ デフォルトのトランザクションサービスとして OTS エンジンを使用するには、最初にトランザクションマネージャを停止する必要があります。

EJB と 2PC トランザクション

J2EE プラットフォームへのメッセージングの導入によって、単一トランザクション内で EJB から複数のリソースにアクセスする多くの一般的なシナリオが用意されました。前に説明したように、トランザクションに複数のリソースが含まれる場合、2 フェーズコミットプロトコルを使ってトランザクションを確実に実行するには OTS エンジンが必要です。次のようなサンプルシナリオがあります。

- 1 つのトランザクション内で、それぞれ異なるデータベースに永続化されている 2 種類のエンティティ Bean にセッション Bean がアクセスする。
- セッション Bean がエンティティ Bean にアクセスし、同じトランザクション内で、メッセージを JMS キューに送信するなどのメッセージング作業をいくつか実行する。
- メッセージ駆動型 Bean の `onMessage` メソッドで、メッセージ配信時にエンティティ Bean にアクセスする。

上記の各サンプルでは、単一トランザクションの一部として、セッション Bean 内またはメッセージ駆動型 Bean 内から 2 種類のリソースにアクセスする必要があります。これらの EJB は、`REQUIRED` トランザクション属性が定義されてから、OTS エンジンにアクセスする必要があります。ただし、OTS エンジンが実行されている場合は、そのパーティションにデプロイメントされたすべてのモジュールは OTS エンジンを認識でき、利用を試みることができます。OTS エンジンは、トランザクションに登録されているリソースが 1 つの場合にだけ 1 フェーズコミットを実行しますが、これは外部プロセスなので、追加の RMI 負荷が発生します。可能であれば、2 フェーズコミットトランザクションに関連しない EJB に対しては、インプロセスのトランザクションマネージャを使用します。AppServer でトランザクションサービスをより有効に活用するには、2PC トランザクションの実行が必要な EJB に対して、Bean レベルプロパティ `ejb.transactionManagerInstanceName` を指定します。このプロパティにより、関連する Bean の任意のメソッドでトランザクションを確立する EJB コンテナが使用する OTS エンジンの名前が決まります。すべての EJB でトランザクションマネージャと OTS エンジンの両方を使用できますが、`ejb.transactionManagerInstanceName` が指定されていない EJB だけがトランザクションマネージャを検索します。

このプロパティは、セッション Bean またはメッセージ駆動型 Bean に対して一般に使用できます。これは、トランザクションが通常、セッション Bean の前面またはメッセージ駆動型 Bean の `onMessage` メソッドで確立されるためです。

`ejb.transactionManagerInstanceName` プロパティを設定するには、管理コンソールを使用します。デプロイメントされた EJB モジュールに移動し、右クリックして、［DDEditor］を選択します。DDEditor のナビゲーションペインで、必要な Bean を選択します。［Properties］タブを選択し、`ejb.transactionManagerInstanceName` プロパティを追加します。このプロパティを `String` として定義し、「MyTwoPhaseEngine」など、一意の名前を指定します。

次に、OTS エンジンのファクトリ名を `ejb.transactionManagerInstanceName` の値に変更します。管理コンソールで、「corbaSample」設定から「OTS エンジン」管理オブジェクトタイプとして識別された OTS エンジンを選択します。右クリックし、ドロップダウンメニューから［Properties］を選択します。［Properties］ダイアログボックスで、［Settings］タブを選択し、［Factory Name］の値を変更します。［OK］をクリックし、サービスを再起動します。OTS エンジンは、AppServer サーバーと関係なくコマンドラインから起動することもできます。ファクトリ名は、次のように `vbroker.ots.name` プロパティを使って指定することもできます。

```
prompt> ots -Dvbroker.ots.name=<MyTwoPhaseEngine>
```

これで、EJB は「MyTwoPhaseEngine」という名前の OTS エンジンを使用するようになります。前に説明したように、パーティションが複数の J2EE モジュールをホストしている場合がありますが、`ejb.transactionManagerInstanceName` が設定されている Bean だけが OTS エンジン（デフォルト以外の）に割り当てられます。トランザクション内でメソッド呼び出しが必要であっても、2PC が必要でないパーティション内のその他の Bean は、ローカルサービスアフィニティにより、常にトランザクションマネージャを検索します。

次に、デプロイメント設定の使用方法のサンプルを示します。下に表示されているコードは、デプロイメントされた EJB モジュールにパッケージされているデプロイメントデスクリプタ `ejb-borland.xml` からの抜粋です。これらは、DDEditor で表示できます。`ejb.transactionManagerInstanceName` プロパティは、セッション Bean「`OrderSesEJB`」に対して設定します。`OrderSesEJB` は、顧客から注文を受け、データベース内に注文を作成し、部品を製造するように製造元にメッセージを送信します。

```
<ejb-jar>
    <enterprise-beans>
        <session>
            <ejb-name>OrderSesEJB</ejb-name>
            <bean-home-name>OrderSes</bean-home-name>
            <bean-local-home-name />
            <ejb-local-ref>
                <ejb-ref-name>ejb/OrderEntLocal</ejb-ref-name>
                <jndi-name>OrderEntLocal</jndi-name>
            </ejb-local-ref>
            <ejb-local-ref>
                <ejb-ref-name>ejb/ItemEntLocal</ejb-ref-name>
            </ejb-local-ref>
            <resource-ref>
                <res-ref-name>jms/QueueConnectionFactory</res-ref-name>
                <jndi-name>QueueConnectionFactory</jndi-name>
            </resource-ref>
            <resource-env-ref>
                <resource-env-ref-name>jms/OrderQueue</resource-env-ref-name>
                <jndi-name>OrderQueue</jndi-name>
            </resource-env-ref>
            <property>
                <prop-name>ejb.transactionManagerInstanceName</prop-name>
                <prop-type>String</prop-type>
                <prop-value>TwoPhaseEngine</prop-value>
            </property>
        </session>

<ejb-jar>
```

ランタイムシナリオのサンプル

下の図は、標準のトランザクションマネージャと OTS エンジンが共存している設定を表しています。このデプロイメント 設定では、2PCトランザクションに関与する Bean は、「 TwoPhaseEngine」という 名前の OTS エンジンによってトランザクションが管理され、2PCトランザクションの必要がない Bean は、デフォルト のインプロセスのトランザクションマネージャを使用します。

使用するサンプルアーカイブは、AppServer パーティション内の `complex.ear` です。次の 3 つの Bean があります。

- **`OrderSesEJB`：**顧客から注文を受け、データベース内に注文を作成し、部品を製造するように製造元にメッセージを送信します。
- **`UserSesEJB`：**企業データベース内に新規ユーザーを作成します。アクセスするデータベースは 1 つだけなので、1PC エンジン（トランザクションマネージャ）にだけアクセスする必要があります。
- **`OrderCompletionMDB`：**部品の発送に関して製造元から通知を受け取ります。また、エンティティ Bean を使ってデータベースを更新します。

次の手順で、このデプロイメントシナリオのサンプルを設定します。

1. DDEditor を使用して、`OrderSesEJB` Bean と `OrderCompletionMDB` Bean に `ejb.transactionManagerInstance` プロパティを追加します。このサンプル用の上記の XML 抽出部分を参照してください。
2. 次に、管理コンソールを使用して、OTS エンジンのファクトリ名を「TwoPhaseEngine」に設定して起動します。
3. ローカルのトランザクションマネージャは有効にしておきます。

次の図は、クライアントと AppServer パーティション間の関係と、上記の設定に基づいて AppServer パーティションが正しいトランザクションサービスを検索する方法を示します。すべての Bean は、トランザクションがコンテナ管理されることを前提としています。

1PC の使い方のサンプル

1 クライアントは、`UserSesEJB` のメソッドを呼び出します。これは、データベース内にユーザーを作成するメソッドのインプリメンテーションです。

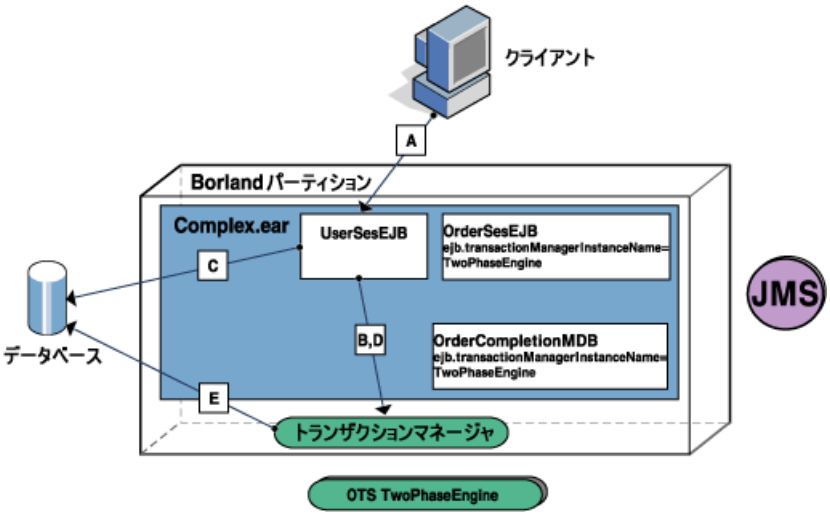
2 その呼び出しが実際に呼び出される前に、パーティションは、次に示すように、インプロセスのトランザクションマネージャを使ってトランザクションを開始します。

3 セッション Bean は、いくつかのデータベース作業を実行します。

4 呼び出しが終了すると、パーティションは commit を発行します。

5 トランザクションマネージャは、データベースリソースに対して `commit_one_phase()` を呼び出します。

図 18.1 1PC の使い方のサンプル



2PC の使い方のサンプル

1 クライアントは、`OrderSesEJB.create()` メソッドを呼び出して、新しい注文を作成します。

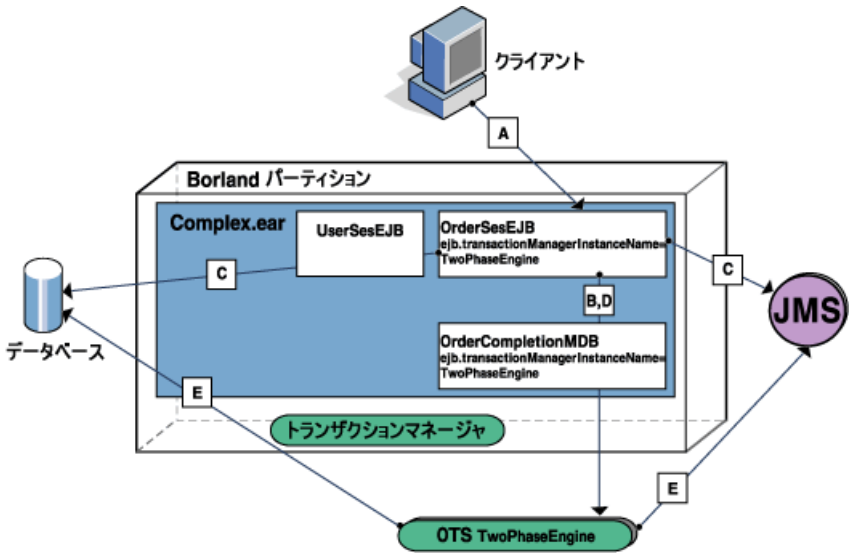
2 この Bean は、**TwoPhaseEngine** という名前の OTS エンジンを使用するように設定されているため、コンテナは、「TwoPhaseEngine」という名前の正しいトランザクションサービスを検索し、それを使ってトランザクションを開始します。

3 セッション Bean は、いくつかのデータベース作業を実行し、JMS キューにメッセージを送信します。

4 呼び出しが終了すると、パーティションは commit を発行します。

5 OTS エンジンは、データベースと JMS リソースを使用して、トランザクションの実行を調整します。

図 18.2　2PC の使い方のサンプル



MDB を使用した 2PC の使い方のサンプル

ある時点で、`REQUIRED` トランザクション属性を持つ `onMessage()` メソッドを呼び出すことにより、非同期メッセージが `OrderCompletionMDB` に配信されます。コンテナは、ITS を使ってトランザクションを開始し、次に `onMessage()` メソッドを呼び出します。メソッドの本体で、Bean がデータベースを更新して注文配送を通知します。2 つのリソースが関連していることに注意してください。最初のリソースは JMS リソースです。これは、メッセージを取得した MDB インスタンスに関連付けられています。2 つ目のリソースは、MDB インスタンスが更新したデータベースです。このシナリオは、ほぼ上のサンプル図と同じです。

メモ　MDB では、`ejb.transactionManagerInstanceName` もサポートされています。詳細については、177 ページの「MDB とトランザクション」を参照してください。

Enterprise JavaBeans の宣言的なトランザクション管理

Enterprise JavaBeans（EJB）のトランザクション管理は、EJB コンテナと EJB によって処理されます。Enterprise JavaBeans を使用すると、アプリケーションは、単一のトランザクション内で複数のデータベースのデータを更新できます。

EJB では、従来のトランザクション管理の形式とは違う宣言形式でトランザクションを管理します。宣言的な管理では、EJB がデプロイメント時にトランザクション属性を宣言します。このトランザクション属性には、EJB コンテナとエンタープライズ Bean 自身のどちらかでトランザクションを管理するかどうか、およびその場合はどの範囲までトランザクションを管理するかを指示します。

従来はアプリケーションがトランザクションのすべての面を管理していました。そのため、次のようなオペレーションを必要としました。

- トランザクションオブジェクトを作成する
- トランザクションを明示的に開始する
- トランザクションに関連するリソースを登録する
- トランザクションコンテキストを監視する
- すべての更新が完了したら、トランザクションをコミットする

トランザクションを最初から最後まで管理するアプリケーションを作成するには、開発者が広範囲なトランザクション処理に関する専門知識を持つ必要があります。そのようなアプリケーションのコードは複雑で難しく、エラーがよく発生します。

プログラマのかわりに、EJB コンテナが宣言によるトランザクション管理を使用して、トランザクションのほとんどの面を管理します。EJB コンテナは、トランザクションの開始と終了を処理するとともに、トランザクションオブジェクトの存続期間の最初から最後まで、トランザクションコンテキストを保持します。特に分散環境におけるトランザクションでは、これによってアプリケーション開発者の責任と作業量が大きく軽減されます。

Bean 管理のトランザクションとコンテナ管理のトランザクション

EJB がビジネスメソッドのコードで独自のトランザクションを確立した場合、その Bean は Bean 管理のトランザクションを使用することになります。一方、エンタープライズ Bean がすべてのトランザクションの確立を EJB コンテナに依頼し、コンテナがアプリケーションアセンブラのデプロイメント指示に基づいてトランザクションを確立した場合、そのエンタープライズ Bean はコンテナ管理のトランザクションを使用することになります。

ステートフルまたはステートレスの EJB セッション Bean では、Bean 管理のトランザクションとコンテナ管理のトランザクションを使用できます。ただし、1 つの Bean で両方を同時に使用できません。EJB エンティティ Bean では、コンテナ管理のトランザクションしか使用できません。EJB が使用するトランザクションの種類は、Bean プロバイダが決定します。

トランザクションを Bean のあるオペレーションで開始し、別のオペレーションで終了する場合は、EJB が独自のトランザクションを管理できます。ただし、そのように設計すると、最初のオペレーションでトランザクション開始メソッドを呼び出した後で、確実にトランザクション終了メソッドが呼び出されることを保証しにくくなります。

エンタープライズ Bean では、Bean 管理のトランザクションではなく、できるだけコンテナ管理のトランザクションを使用してください。コンテナ管理のトランザクションを使用すれば、プログラミングが簡単になり、プログラムエラーも少なくなります。また、コンテナ管理のトランザクション Bean の方がカスタマイズしやすく、ほかの Bean と容易に組み合わせることができます。

ローカルトランザクションとグローバルトランザクション

トランザクションには、1 つ以上のリソースマネージャが維持するデータに実行された作業の原子単位が含まれます。リソースマネージャの例としては、データベース管理システムと JMS メッセージプロバイダがあります。ローカルトランザクションには、外部トランザクションマネージャから独立した 1 つのリソースマネージャに実行された作業が含まれます。たとえば、データベースから取得する JDBC 接続は、接続の autoCommit モードがオフの場合、データベースを更新するために SQL オペレーションを実行してから、ローカルトランザクション内で `commit()` オペレーションを使って作業をコミットできます。autoCommit モードがオフでない場合、各オペレーションはローカルトランザクション内で実行されます。グローバルトランザクションは、パーティショントランザクションマネージャ、OTS エンジンなどのトランザクションマネージャによって調整され、1 つ以上の分散リソースマネージャに実行された作業を含めることもできます。コンテナ管理および Bean 管理の EJB トランザクション管理は、グローバルトランザクションの使用を意味します。1 つのリソースマネージャがグローバルトランザクションに参加する場合、すべての作業はグローバルトランザクションのかわりにローカルトランザクション内で実行されます。詳細については、EJB 仕様バージョン 2.0 のセクション 17.6.4 の「Local transaction optimization」を参照してください。

Bean 管理のトランザクションによって定義された EJB のメソッドは、JTA インターフェース `javax.transaction.UserTransaction` のインプリメンテーションハンドルを取得して、明示的にグローバルトランザクションに参加するためにオペレーションを呼び出す必要があります。

コンテナ管理のトランザクションでは、EJB コンテナは各 EJB のメソッド呼び出しに介入し、一定の規則にしたがって作業をグローバルトランザクションの一部として処理する必要があるかどうかを決定します。コンテナの決定は、コンポーネントデプロイメントデスクリプタの中でアプリケーションアセンブラによって設定されたメソッドのトランザクション属性値に依存し、またグローバルトランザクションコンテキストがメソッドの呼び出し時に存在するかどうかに依存します。EJB 仕様バージョン 2.0 のセクション 17.6.2.7 の「Transaction attribute summary」の表 14 を参照してください。グローバルトランザクションコンテキストなしにメソッドが処理される場合、メソッド内から外部リソースマネージャに対して実行される作業はローカルトランザクションを使って行われます。次に、コンテナ管理のトランザクション境界を持つ EJB の EJB メソッドに対してローカルトランザクションを使用する場合のサンプルを示します。

- トランザクション属性が `NotSupported` に設定され、リソースへのアクセスが検出された場合
- トランザクション属性が `Supports` に設定されており、a）メソッドがグローバルトランザクションから呼び出されていない、および b）リソースがアクセスされた場合
- トランザクション属性が `Never` に設定され、リソースへのアクセスが検出された場合

トランザクションの属性

Bean 管理のトランザクションを使用する EJB では、各メソッドにトランザクション属性が関連付けられます。これらの属性の値は、Bean が関与するトランザクションの管理方法をコンテナに指示します。Bean のメソッドには、6 種類のトランザクション属性を関連付けることができます。この関連付けは、アプリケーションアセンブラまたはデプロイヤがデプロイメント時に行います。

次のような属性があります。

- `Required` - この属性を使用する場合、グローバルトランザクションコンテキスト内で、関連付けられたメソッドによって作業が実行されます。呼び出し元がすでにトランザクションコンテキストを持っている場合、コンテナはそのコンテキストを使用します。呼び出し元がトランザクションコンテキストを持っていない場合、コンテナは新しいトランザクションを自動的に開始します。この属性を使用すると、同じグローバルトランザクションを使って複数の Bean を簡単にまとめることができ、各 Bean の作業を調整できます。
- `RequiresNew` - この属性は、メソッドに既存のトランザクションを関連付けない場合に使用します。この属性を使用すると、コンテナは常に新しいトランザクションを開始します。
- `Supports` - この属性の場合、メソッドはグローバルトランザクションを使用しません。この属性は、Bean メソッドが 1 つのトランザクションリソースにアクセスする場合、またはトランザクションリソースにアクセスしない場合で、かつ別のエンタープライズ Bean を呼び出さない場合に使用してください。この属性の目的は、グローバルトランザクションのコストを省いて最適化することです。この属性が設定され、グローバルトランザクションがすでに存在する場合、EJB コンテナは、呼び出すメソッドを既存のグローバルトランザクションに含めます。この属性が設定され、既存のグローバルトランザクションが存在しない場合、コンテナは、呼び出すメソッドのためにローカルトランザクションを開始します。このローカルトランザクションは、メソッドの終了とともに完了します。
- `NotSupported` - この属性の場合、Bean はグローバルトランザクションを使用しません。この属性を設定する場合、メソッドはグローバルトランザクションに関与させないでください。EJB コンテナは、既存のすべてのグローバルトランザクションを中断し、そのメソッドのためのローカルトランザクションを開始します。このローカルトランザクションは、メソッドの終了とともに完了します。
- `Mandatory` - この属性は使用しないことをお勧めします。この属性は Requires に似ていますが、呼び出し元があらかじめ関連付けられたトランザクションを持っている必要があります。持っていない場合、コンテナは `javax.transaction.TransactionRequiredException` を発生させます。この属性を使用すると、呼び出し元のトランザクションに関して仮定が行われるため、Bean の組み合わせの柔軟性が損なわれます。
- `Never` - この属性は使用しないことをお勧めします。この属性を使用すると、EJB コンテナは、メソッドのローカルトランザクションを開始します。このローカルトランザクションは、そのメソッドの終了とともに完了します。

通常は、`Required` と `RequiresNew` の 2 つの属性だけを使用してください。`Supports` 属性と `NotSupported` 属性は最適化のために使用します。`Never` と `Mandatory` は、Bean の組み合わせの柔軟性を損なうため、お勧めできません。また、トランザクションの同期化を考慮した Bean で `javax.ejb.SessionSynchronization` インターフェースを実装している場合、アセンブラまたはデプロイヤで指定できる属性は、`Required`、`RequiresNew`、または `Mandatory` のものだけです。これらの属性を使用すると、コンテナは必ずグローバルトランザクション内で Bean を呼び出します。トランザクションの同期は、グローバルトランザクション内でしか行うことができません。

メモ クライアントが呼び出した EJB が他の EJB を呼び出し、両方の EJB が同じデータベースにアクセスする場合、呼びされるメソッドのトランザクション属性が必須に設定されていないと、1 つの JDBC 接続だけが使用されます。これは、各 Bean で行われる処理が 1 つのトランザクションの一部になるためです。

JTA API を使用したプログラムによるトランザクション管理

すべてのトランザクションは、Java Transaction API（JTA）を使用します。コンテナ管理のトランザクションでは、プラットフォームがトランザクション境界を指定し、コンテナは JTA API を使用します。開発者がこの API を Bean のコードで使用する必要はありません。

ただし、自身のトランザクション（Bean 管理のトランザクション）を管理する Bean は、JTA の `javax.transaction.UserTransaction` インターフェースを使用する必要があります。このインターフェースにより、クライアントまたはコンポーネントがトランザクション境界を指定できます。Bean 管理のトランザクションを利用する Enterprise JavaBeans は、`EJBContext.getUserTransaction()` メソッドを使用します。

また、すべてのトランザクション対応クライアントは、JNDI を使って `UserTransaction` インターフェースを検索します。この場合には、次のコード行に示すように、JNDI ネーミングサービスによる JNDI の `InitialContext` を作成します。

```
javax.naming.Context context = new javax.naming.InitialContext();
```

次のコードでは、Bean が `InitialContext` オブジェクトを取得したら、JNDI `lookup()` 操作を使って `UserTransaction` インターフェースを取得します。

```
javax.transaction.UserTransaction utx = (javax.transaction.UserTransaction)
      context.lookup("java:comp/UserTransaction");
```

EJB は、`EJBContext` オブジェクトから `UserTransaction` インターフェースへのリファレンスを取得できます。エンタープライズ Bean は、デフォルトで `EJBContext` オブジェクトへのリファレンスを継承するからです。したがって、Bean は、`InitialContext` オブジェクトを取得してから JNDI `lookup()` メソッドを使用するかわりに、`EJBContext.getUserTransaction()` メソッドを使用します。エンタープライズ Bean 以外のトランザクション対応クライアントでは、JNDI による検索を行う必要があります。

`UserTransaction` インターフェースへのリファレンスを持つ Bean またはクライアントは、自身のトランザクションを開始して管理します。つまり、`UserTransaction` インターフェースのメソッドを使用すると、トランザクションの開始、コミット、またはロールバックができます。`begin()` メソッドを使ってトランザクションを開始し、次に `commit()` メソッドを使ってデータベースの変更をコミットします。または、`rollback()` メソッドを使用して、トランザクションのすべての変更を破棄し、データベースをトランザクション開始前の状態に戻します。`begin()` メソッドから `commit()` メソッドまでの間には、トランザクションの作業を実行するコードを記述します。

JDBC API の変更

AppServer では、標準の Java Database Connectivity（JDBC）API を使用して、ベンダーが提供するドライバによって JDBC をサポートするデータベースにアクセスします。データベースへのアクセス要求は、AppServer JDBC 接続プールを介して一元管理されます。ここでは、トランザクションの JDBC 動作に対して AppServer JDBC プールが行う変更について説明します。

JDBC プールは、トランザクション型アプリケーションがデータベースへの JDBC 接続を取得できる擬似 JDBC ドライバです。JDBC プールは、JDBC 接続をトランザクションマネージャのトランザクションに関連付け、JDBC 接続を作成する JDBC ドライバに接続要求をデリゲートします。JDBC プールを使って接続が取得されると、トランザクションサービスによってトランザクションが自動的に調整されます。

JDBC プールとそれに関連付けられているリソースは、DBMS への完全なトランザクションアクセスを提供します。JDBC プールは、リソースをトランザクションコーディネータに透過的に登録します。JDBC API のバージョン 1.x の制約により、JDBC プールでは 1 フェーズコミットだけを使用できます。JDBC API のバージョン 2.0 は、完全な 2 フェーズコミットをサポートします。

JDBC API の動作の変更

Java で記述されているトランザクション型アプリケーションに対して JDBC アクセスを有効にするには、JDBC API を使用します。JDBC API については、次の Web サイトを参照してください。

http://www.javasoft.com/products/jdk/1.2/docs/guide/jdbc/spec/jdbc-spec.frame.html

ただし、一部の JDBC メソッドの動作は、パーティションによって管理されるトランザクションのコンテキスト内で呼び出された場合、パーティションのトランザクションサービスによって上書きされます。次のメソッドが影響を受けます。

- `Java.sql.Connection.commit()`
- `Java.sql.Connection.rollback()`
- `Java.sql.Connection.close()`
- `Java.sql.setAutoCommit(boolean)`

ここでは、この後、パーティション管理のトランザクション用に合わせたこれらのメソッドのセマンティクスの変更について説明します。

メモ スレッドがトランザクションに関連付けられていない場合は、これらのすべてのメソッドが標準の JDBC トランザクションセマンティクスを使用します。

上書きされた JDBC メソッド

Java.sql.Connection.commit()

JDBC API で定義されているように、このメソッドは、前の `commit()` または `rollback()` 以降に JDBC 接続で実行されたすべての作業をコミットし、すべてのデータベースのロックを解放します。

グローバルトランザクションが現在の実行スレッドに関連付けられている場合は、このメソッドを使用しないでください。グローバルトランザクションがコンテナ管理のトランザクションでなく（アプリケーションが独自のトランザクションを管理する）、コミットが必要な場合は、JDBC 接続で直接 `commit()` を呼び出すのではなく、JTA API を使ってコミットを実行してください。

Java.sql.Connection.rollback()

JDBC API で定義されているように、このメソッドは、前の `commit()` または `rollback()` 以降に JDBC 接続で実行されたすべての作業をロールバックし、すべてのデータベースのロックを解放します。

グローバルトランザクションが現在の実行スレッドに関連付けられている場合は、このメソッドを使用しないでください。グローバルトランザクションがコンテナ管理のトランザクションでなく（アプリケーションが独自のトランザクションを管理する）、ロールバックが必要な場合は、JDBC 接続で直接 `rollback()` を呼び出すのではなく、JTA API を使ってロールバックを実行してください。

Java.sql.Connection.close()

JDBC API で定義されているように、このメソッドは、データベース接続とその接続に関連付けられているすべての JDBC リソースを閉じます。

スレッドがトランザクションに関連付けられている場合、この呼び出しでは、接続に関する処理が完了したことが JDBC プールに通知されるだけです。JDBC プールは、トランザクションが完了すると、その接続を接続プールに戻します。JDBC プールによって開かれた JDBC 接続は、アプリケーションが明示的に閉じることはできません。

Java.sql.Connection.setAutoCommit(boolean)

JDBC API で定義されているように、このメソッドは、トランザクションの自動コミットモードを設定するために使用します。`setAutoCommit()` メソッドを使用して、Java アプリケーションで次のいずれかを実行できます。

- すべての SQL 文を個別のトランザクションとして実行しコミットする（true に設定した場合）。これはデフォルトのモードです。
- 接続で `commit()` または `rollback()` を明示的に呼び出す（false に設定した場合）。

スレッドがトランザクションに関連付けられている場合、JDBC プールは、パーティションのトランザクションサービストランザクションのスコープ内で作成されたすべての接続に対して、自動コミットモードをオフにします。これは、トランザクションサービスがトランザクションの完了までを制御する必要があるためです。アプリケーションがトランザクションに関連付けられている場合、自動コミットモードを true に設定しようとすると、`java.sql.SQLException()` が生成されます。

EJB 例外の処理

Enterprise JavaBeans は、トランザクションの処理中にエラーが発生すると、アプリケーションレベルまたはシステムレベルの例外を発生させます。アプリケーションレベルの例外は、ビジネスロジックのエラーに関係し、呼び出し元のアプリケーションによって処理されることが想定されています。一方、実行時エラーなどのシステムレベルの例外は、アプリケーションの範囲を超えており、アプリケーション、Bean、または Bean コンテナが処理します。

EJB では、`Home` インターフェースと `Remote` インターフェースの `throws` 節で、アプリケーションレベルの例外とシステムレベルの例外を宣言します。Bean のメソッドの呼び出し時に、プログラムの `try/catch` ブロックにチェック例外があるかどうかを確認する必要があります。

システムレベルの例外

EJB は、`java.ejb.EJBException`（`java.rmi.RemoteException` の場合もあり）というシステムレベルの例外を発生させて、システムレベルの予期しない障害を知らせます。たとえば、データベース接続を開くことができないと、この例外が発生します。`java.ejb.EJBException` は実行時例外なので、エンタープライズ Bean のビジネスメソッドの `throws` 節の中に記述する必要はありません。

システムレベルの例外が発生したら、通常、トランザクションをロールバックする必要があります。多くの場合は、Bean を管理するコンテナがロールバックを実行します。特に Bean 管理のトランザクションの場合は、クライアントがトランザクションをロールバックする必要があります。

アプリケーションレベルの例外

EJB はアプリケーションレベルの例外を発生させて、システム関連の問題ではなくビジネスロジックのエラーであるアプリケーション固有のエラー状況を知らせます。アプリケーションレベルの例外とは、`java.ejb.EJBException` 以外の例外のことです。アプリケーションレベルの例外はチェック例外なので、チェック例外の発生の可能性があるメソッドを呼び出すときは、チェック例外があるかどうかを確認する必要があります。

EJB のビジネスメソッドは、アプリケーション例外を使用して、無効な入力値や受け入れ限度の超過などアプリケーションの異常状態を知らせます。たとえば、口座引き落とし処理を行う Bean のメソッドは、アプリケーション例外を発生させて、残高不足のため引き落とし操作ができないことを知らせます。多くの場合、クライアントは、トランザクション全体をロールバックしなくても、アプリケーションレベルのエラーから回復できます。

アプリケーションまたは呼び出し元のプログラムは、生成された例外をそのまま受け取るため、問題を正確に知ることができます。アプリケーションレベルの例外が発生しても、EJB のインスタンスは、クライアントのトランザクションを自動的にはロールバックしません。したがって、クライアントには、エラーメッセージを評価し、必要に応じて状況を修正して、トランザクションを回復する機会があります。クライアントは、そのトランザクションを破棄することもできます。

アプリケーション例外の処理

アプリケーションレベルの例外はビジネスロジックのエラーを知らせるため、こうした例外はクライアントで処理するようにします。トランザクションをロールバックする必要がある場合、ロールバックの対象にするトランザクションに自動的にマークが付けられるわけではありません。トランザクションを破棄してからロールバックしなければならないこともありますが、トランザクションの再試行で対応できる場合がほとんどです。

Bean プロバイダは、クライアントがトランザクションを継続した場合でも、Bean のデータの整合性が失われないように保証する責任があります。プロバイダが整合性を保証できない場合、Bean は、トランザクションにロールバックのマークを付加します。

トランザクションのロールバック

クライアントプログラムでアプリケーション例外を受け取ったら、現在のトランザクションに「ロールバック」のマークが付いていないかどうかを最初に確認する必要があります。たとえば、クライアントが `javax.transaction.TransactionRolledbackException` を受け取ったような場合です。この例外は、ヘルパーエンタープライズ Bean が失敗したため、トランザクションが破棄されたか、「ロールバックのみ」とマークされていることを知らせます。通常、クライアントは、呼び出されたエンタープライズ Bean が実行されるトランザクションコンテキストを知りません。呼び出された Bean は、呼び出し元プログラムのトランザクションコンテキストとは異なる独自のトランザクションコンテキスト内で実行されることも、呼び出し元プログラムのコンテキスト内で実行されることもあります。

EJB が呼び出し元プログラムと同じトランザクションコンテキスト内で実行された場合は、その Bean（またはコンテナ）が、トランザクションにロールバックのマークを付けた可能性があります。EJB コンテナがトランザクションにロールバックのマークを付けた場合、クライアントは、そのトランザクションに含まれるすべての作業を停止する必要があります。通常、宣言的なトランザクションを使用するクライアントは、`javax.transaction.TransactionRolledbackException` などの適切な例外を受け取ります。宣言的なトランザクションとは、トランザクションの細部がコンテナから管理されているトランザクションのことです。

自身が EJB であるクライアントは、`javax.ejb.EJBContext.getRollbackOnly` メソッドを呼び出して、自分のトランザクションにロールバックのマークが付いているかどうかを調べます。

Bean 管理のトランザクション、つまりクライアントが明示的に管理するトランザクションでは、クライアントが `java.transaction.UserTransaction` インターフェースの `rollback` メソッドを呼び出して、トランザクションをロールバックする必要があります。

トランザクションを継続するときの選択肢

トランザクションにロールバックのマークが付いていない場合、クライアントには次の 3 つの選択肢があります。

- トランザクションをロールバックする。
- チェック例外を発生させるか元の例外を再発生させて、責任を回避する。
- トランザクションを再試行して継続する。トランザクションの一部だけを再試行する場合もあります。

クライアントで、ロールバックのマークが付いていないトランザクションについてチェック例外を受け取った場合、最も安全な対応方法は、トランザクションをロールバックすることです。クライアントは、トランザクションをロールバックする方法をトランザクションに「ロールバックのみ」のマークを付けることによって実行します。すでにクライアントがトランザクションを開始している場合は、`rollback` メソッドを呼び出し、トランザクションを実際にロールバックします。

クライアントでは、独自のチェック例外を発生させたり、元の例外を再発生させることもできます。例外を発生させることにより、トランザクションチェイン中の別のプログラムにトランザクションを破棄するかどうかの判断をさせます。ただし、多くの場合、トランザクションを継続するかどうかを最も適切に判断できるのは、問題の発生箇所に最も近いコードです。

クライアントでは、トランザクションを継続できます。例外のメッセージを調べると、メソッドを別のパラメータで再度呼び出せばうまく処理できるかどうかがわかります。ただし、トランザクションを再試行すると危険な場合があります。エンタープライズ Bean がその状態の終了処理を正しくできるかどうかはわからず、その保証もありません。

ただし、ステートレスセッション Bean を呼び出すクライアントは、発生した例外から問題を判断できる場合、処理が成功すると見込んでトランザクションを再試行することができます。この場合はステートレスな Bean を呼び出すので、Bean がトランザクションを放置している状態がクライアントで認識されないという問題は発生しません。

第 19 章

メッセージ駆動型 Bean と JMS

JMS と EJB

仕様では、JMS メッセージプロデューサや同期コンシューマとして機能する Bean に制限を設けていません。標準 JMS API により、キューに対するメッセージ送信や、トピックの公開が可能です。メッセージの同期スタイル消費を実行する限り (`javax.jms.MessageListener` に基づかない)、コンシューマ側に問題は発生しません。状況を複雑にする要素は、アプリケーションの他の作業によって共有されるトランザクションコンテキストに JMS メッセージの送信要求または受信要求が加わる必要があるという点です。しかし、この問題については、非 EJB アプリケーションで JMS や JTA を使って解決できる見通しです。EJB に特別な措置は必要ありません。

EJB メソッド呼び出しは同期的であり、呼び出しの一部は Bean による処理が終了するまで待機する必要があります。このことは、他の Bean、データベースなどの呼び出しにも適用される場合があります。RMI のこの振る舞いは、一般には望ましいものではありません。たとえば、メソッドを呼び出したとき、重い処理を実行する前にそれを返して、処理の間は、呼び出し元を別のタスクに振り向けたい場合があります。これに対しては、クライアント側でスレッド化するのが普通ですが、問題が 2 つあります。

- つまり、クライアントのプログラミングモデルが真の非同期方式になっていないことと、
- クライアントが EJB の場合、スレッド化はメソッドインプリメンテーションで禁止されていることです。

最も望ましいのは、AppClient、サーブレット、EJB、その他コンポーネントに、JMS API を使用してメッセージを起動する機能を与え、そのメッセージで EJB を非同期に駆動する方法です。こうすれば、EJB はメッセージを別の EJB に送信することができるほか、直接データアクセスやその他のビジネスロジックの処理が可能です。メッセージがキューに入るまで、呼び出し元は待機します。一方、EJB は最適な方法でメッセージを処理できます。この EJB の処理には、通常は、次の 3 つの操作からなる 1 つの作業単位が含まれます。

1. メッセージのデキュー
2. インスタンスの起動と、ビジネスロジックの要求する作業の実行
3. オプションで、応答メッセージのキューバック

この作業単位を機能させるには、Enterprise システム側で、トランザクション保証とコンテナ管理保証に対応できることが必要です。

EJB 2.0 メッセージ駆動型 Bean（MDB）

EJB 2.0 仕様では、JMS と、エンタープライズ Bean の非同期呼び出し間の統合を EJB コンテナで対応させて形式化しています。これにより、開発者の負担が軽減され、JMS リスナーであり、また EJB でもあるクラスを提供するだけで済むようになりました。それには、`javax.jms.MessageListener` と `javax.ejb.MessageDrivenBean` をこのクラスで実装する必要があります。アプリケーションプログラマの仕事は、このクラスと、すべてのデプロイメント設定を収めた XML デスクリプタを提供するだけです。

クライアントから見ると、この EJB は存在しません。クライアントはキューやトピックにメッセージを公開するだけです。EJB コンテナは、公開したキューやトピックに MDB を関連付け、ライフサイクル、プーリング、同時性、リエントラント、セキュリティ、トランザクション、メッセージ、ハンドリング、例外処理などを操作します。

EJB 2.1 MDB

EJB 2.1 に J2EE Connector Architecture 1.5 (JCA 1.5) を統合することによち、MDB は JMS ベースのプロバイダに加えて非 JMS メッセージングサーバーからのメッセージも処理できるようになりました。JCA 1.5 準拠のリソースアダプタ実装は、すべてのタイプのメッセージングサーバーにデプロイメントでき、また、アプリケーションサーバーにもデプロイメントできます。メッセージングサーバーからの着信メッセージをアプリケーションサーバーに渡すように設定すると、2.1 MDB をドライブするメッセージのソースとしてリソースアダプタを選択できます。

JCA 1.5 は、EJB コンテナと非同期コネクタ間のメッセージングコントラクトである Message Inflow コントラクトを定義することにより、EIS または他のタイプのメッセージングプロバイダからの着信メッセージを自動的に処理できるようにします。EJB 2.1 MDB は標準の javax.ejb.MessageDrivenBean インターフェース、およびコネクタが定義する特定のメッセージングインターフェースを実装する必要があります。コネクタが JMS ベースのプロバイダの場合、MDB は javax.jms.MessageListener を実装する必要がありますが、非 JMS プロバイダの場合、プロバイダに固有の他のタイプのインターフェースを実装する必要があります。

Borland Application Server 6.7 は、EJB 2.1 MDB が JMS プロバイダからのメッセージを処理する方法として、JCA リソースアダプタを経由する間接的な場合と、事前にデプロイメントされている JCA リソースアダプタを必要としない直接的な場合の両方に対応しています。

MDB のクライアントビュー

セッション Bean やエンティティ Bean の場合とは異なり、クライアントは MDB にバインドしません。クライアントに必要なことは、MDB が監視対象として設定された送信先にメッセージを送信することだけです。通常、クライアントはデプロイメントデスクリプタの JMS 送信先仕様に `<resource-ref>` と `<resource-env-ref>`（EJB 2.0 の場合）、または `<message-destination-ref>`（EJB 2.1 の場合）も使用し、MDB デプロイメントデスクリプタでの設定と同じ JNDI 名をポイントします。クライアントデプロイメントデスクリプタを JMS プロバイダと通信するように設定する方法については、「*JMS の使い方*」の章の 205 ページの「J2EE アプリケーションコンポーネントにおける JMS 接続ファクトリと送信先の取得」のセクションを参照してください。

たとえば、クライアントが認識する必要がある EJB メタデータやハンドルはありません。これは、メッセージ駆動型 Bean の RMI クライアントビューがないためです。

MDB 設定

MDB は EJB インターフェースを公開しないので、EJBHome オブジェクトが持つような意味での JNDI 名はありません。デプロイメントされた MDB は、着信メッセージを処理する準備の過程で、メッセージプロバイダと通信します。

EJB 2.0 MDB は、MDB のデプロイメント前に JNDI にあらかじめ存在する必要がある 2 つの JMS リソースオブジェクトと関連付けられます。具体的には、次の 2 つです。

- JMS プロバイダの接続に使用する JMS 接続ファクトリ
- 入力メッセージを監視するためのプロバイダ上の JMS キューやトピック

これらのオブジェクトを MDB の ejb-borland.xml デプロイメントデスクリプタで指定する元の JNDI 名。<connection-factory-name> は、JMS サービスプロバイダとの接続に使用するリソース接続ファクトリを取得します。<message-driven-destination-name> 要素は、MDB の監視場所である実トピックとキューを取得します。以上の要素を指定すると、MDB にとって JMS サービスプロバイダとの接続、メッセージの受信、応答の送信に必要な情報がすべて揃います。

EJB 2.1 MDB は、次の 2 つの方法のいずれかによって設定できます。EJB 2.1 MDB が javax.jms.MessageListener を実装して JMS ベースの MDB であることを示すと、JCA 1.5 コネクタを使用しないで JMS プロバイダと直接通信するように設定できます。この場合は、MDB の ejb-borland.xml デプロイメントデスクリプタで、<jms-provider-ref> 要素の下に JMS リソースオブジェクトの JNDI 名を指定できます。または、Borland 固有のデプロイメントデスクリプタファイル ejb-borland.xml の <resource-adapter-ref> 要素を使用して、JCA 1.5 コネクタからメッセージを受信するように EJB 2.1 MDB を設定できます。

EJB 2.0 MDB から JMS サーバーへの接続

EJB 2.0 MDB は、着信メッセージのソースである JMS サーバーに接続するための特別な方法です。標準デプロイメントデスクリプタファイル ejb-jar.xml で、MDB の宣言内の <message-driven-destination> 要素を使用して、受信する着信メッセージの送信元である JMS 送信先のタイプを定義します。たとえば、次のようになります。

```
<message-driven>
 <ejb-name>MyMDBTopic</ejb-name>
 ...
 <message-driven-destination>
  <destination-type>javax.jms.Topic</destination-type>
  <subscription-durability>Durable</subscription-durability>
 </message-driven-destination>
 ...
</message-driven>
```

この要素の使い方については、J2EE 1.3 仕様を参照してください。Borland 固有の XML ファイル ejb-borland.xml では、同等の要素 <message-driven-destination> を使用して、JMS 送信先の論理名と JNDI 名をバインドします。JMS サーバーとの接続に必要な JMS 接続ファクトリの JNDI 名も、<connection-factory-name> を使用して定義する必要があります。たとえば、次のようにします。

```
<message-driven>
 <ejb-name>MyMDBTopic</ejb-name>
 ...
 <message-driven-destination>jms/resources/Topic</message-driven-destination>
 <connection-factory-name>jms/resources/tcf</connection-factory-name>
 ...
</message-driven>
```

JNDI の下でバインドされるこれらの JMS リソースオブジェクト設定の詳細は、「*JMS の使い方*」の章の 201 ページの「JMS 接続ファクトリと宛先の設定」のセクションを参照してください。

メモ MDB を REQUIRED トランザクション属性と一緒にデプロイメントするときは、XA 接続ファクトリが必要です。このデプロイメントの全体的な考え方は、MDB を駆動するメッセージの消費において、MDB.onMessage() メソッドで実行される他の作業と同じトランザクションを共有することです。そのために、このコンテナでは、JMS サービスプロバイダやトランザクションでリストされた他のリソースと XA 調整を行います。

EJB 2.1 MDB からメッセージソースへの接続

EJB 2.1 および JCA 1.5 に更新された結果、標準デプロイメントデスクリプタ ejb-jar.xml、および J2EE 1.4 用の Borland 独自のデプロイメントデスクリプタ ejb-borland.xml の両方が変更されています。

ejb-jar.xml の変更

各 EJB 2.1 MDB は、デプロイメントデスクリプタの情報に基づいて、そのメッセージソースに接続されます。EJB 2.1 の標準デプロイメントデスクリプタ ejb-jar.xml は、コネクタベースの MDB に対応するように変更されました。

EJB 2.1 では、新しい要素 <messaging-type>、<message-destination-type>、および <activation-config> が ejb-jar.xml ファイルに追加されます。

<messaging-type> 要素は、MDB が実装する完全修飾インターフェース名を提示することにより、使用されるメッセージを示します。インターフェース名が提示されない場合、コンテナはデフォルトの JMS メッセージタイプ javax.jms.MessageListener になります。

オプションの <message-destination-type> 要素は、Bean のメッセージ取得先のタイプを表す完全修飾インターフェース名を示します。JMS メッセージタイプ javax.jms.MessageListener を表す MDBS に対して指定できる値は、javax.jms.Topic または javax.jms.Queue です。

コネクタベースの MDB が JMS に排他的に依存しなくなったため、EJB 2.0 の <message-driven-destination><message-selector> および <acknowledge-mode> 要素は、EJB 2.1 では削除されました。EJB 2.1 MDB のアクティブ化に必要な設定プロパティは、<activation-config> の下で、名前と値のペアの汎用的な組み合わせとして定義できます。メッセージサービスを記述するプロパティ名と値は、使用するサービスのタイプによって異なります。これらの <activation-config> プロパティは、メッセージ駆動型 Bean がデプロイメントされたときに検査されます。EJB 2.0 から削除された JMS 関連の各要素は、<messaging-type> 要素で JMS メッセージングタイプ（javax.jms.MessageListener）を指定している場合、<activation-config-property> 要素によって表すことができます。

JMS ベースの MDB を EJB 2.1 ejb-jar.xml ファイルで定義する例を以下に示します。

```
<enterprise-beans>
 <message-driven>
   <ejb-name>EJB_SEC_MDB_TOPIC_CMT</ejb-name>
   <ejb-class>com.sun.ts.tests.ejb.ee.sec.mdb.MsgBean</ejb-class>
   <messaging-type>javax.jms.MessageListener</messaging-type>
   <transaction-type>Container<transaction-type>
   <message-destination-type>javax.jms.Topic</message-destination-type>
   <message-destination-link>StockTopic</message-destination-link>
   <activation-config>
    <activation-config-property>
     <activation-config-property-name>acknowledgeMode
           </activation-config-property-name>
     <activation-config-property-value>Auto-acknowledge
           <activation-config-property-value>
    </activation-config-property>
    <activation-config-property>
     <activation-config-property-name>destinationType
           </activation-config-property-name>
     <activation-config-property-value>javax.jms.Topic
           <activation-config-property-value>
    </activation-config-property>
    <activation-config-property>
     <activation-config-property-name>subscriptionDurability
           </activation-config-property-name>
     <activation-config-property-value>DURABLE
           <activation-config-property-value>
    </activation-config-property>
   <activation-config>
 </message-driven>
 ...
</enterprise-beans>
```

メッセージングサービスを記述するために <activation-config> で使用されるプロパティの名前と値は、使用するメッセージサービスのタイプによって異なりますが、EJB 2.1 では JMS ベースの MDB について、常に次の 4 つのプロパティを使用するように定義しています。

<activation-config-property-name>	説明	<activation-config-property-value>
acknowledgeMode	MDB がメッセージを受信したことを MDB コンテナが JMS プロバイダに通知できます。	Auto-acknowledge（デフォルト）または Dups-ok-acknowledge
messageSelector	MDB が受信するメッセージを選択できます。受信するメッセージに基づいて MDB がプロパティを設定できます。設定される各プロパティは、式またはブール値です。	文字列セレクタ
destinationType	MDB が受信するメッセージの送信元のタイプを示します。	javax.jms.Queue または javax.jms.Topic
subscriptionDurablity	MDB コンテナがプロバイダからの接続を解除されたときに、MDB が受信したメッセージをすべて JMS プロバイダが保存する必要があるかどうかを決定します。	NonDurable（デフォルト）または Durable

JCA 1.5 仕様の Message Inflow コントラクトは、メッセージングサービスプロバイダとアプリケーションサーバーの間のコントラクトで、MDB へのメッセージ配信に関するものです。このコントラクトの一部として、メッセージングプロバイダは ActivationSpec という JavaBean を実装します。ActivationSpec は、メッセージングプロバイダがメッセージをデプロイメントするために必要なプロパティを定義します。管理者はこれらのプロパティのデフォルト値を定義できますが、MDB を含むアプリケーションをデプロイメントすると、MDB のデプロイメントデスクリプタに定義された `<activation-config-property>` 要素によって上書きされます。JMS プロバイダは Sun 仕様に準拠しているため、上に示したプロパティをその ActivationSpec に定義しています。プロパティを MDB のデプロイメントデスクリプタに含めず、かわりに管理者が定義することができます。反対に、プロバイダ固有のプロパティの場合は、これまで管理者が定義する必要があったプロパティを MDB のデプロイメントデスクリプタに含めることも考えられます。

標準デスクリプタ要素 `<message-destination-link>` は、メッセージ送信先の論理名の定義に使用されます。この要素は `<message-destination>` 要素と併せて使用され、アプリケーション内のメッセージフローを示します。JMS プロバイダメッセージソースを指定する MDB の場合、JMS 送信先オブジェクトは、`<message-destination-link>` の目的の `<message-destination>` が MDB のデプロイメントデスクリプタにあれば、それを使用して解決されます。

EJB 2.1 MDB では、MDB のアプリケーションロジック内で使用される JMS 送信先の定義に、`<resource-env-ref>` のかわりに標準デプロイメントデスクリプタ要素 `<message-destination-ref>` を使用できます。

ejb-borland.xml の変更

Borland 独自のデプロイメントデスクリプタは変更され、新しい接続ベースの MDB を含めることができます。これには、新しい要素 <message-source> が含まれます。この要素によってアプリケーションアセンブラは、JCA 1.5 リソースアダプタを使用して、または JMS メッセージングタイプ MDB の場合は直接 JMS プロバイダに対して、MDB のアクティブ化を指定できます。JMS プロバイダを使用している場合は、<jms-provider-ref> 要素を次のように使用する必要があります。

```
<enterprise-beans>
   <message-driven>
      <ejb-name>EJB_SEC_MDB_TOPIC_CMT</ejb-name>
      <message-source>
         <jms-provider-ref>
            <message-driven-destination-name>
              Jms/MyTopic
            </message-driven-destination-name>
            <connection-factory-name>jms/myTCF</connection-factory-name>
            <pool>
               <max-size>120</max-size>
               <init-size>100</init-size>
               <wait-timeout>600</wait-timeout>
            </pool>
         </jms-provider-ref>
      </message-source>
          ....
   </message-driven>
</enterprise-beans>
```

コネクタベースの非 JMS メッセージングプロバイダを使用している場合は、次の <message-source> を使用します。

```
<enterprise-beans>
   <message-driven>
      <ejb-name>EJB_SEC_MDB_TOPIC_CMT</ejb-name>
      <message-source>
         <resource-adapter-ref>
            <instance-name>
              MyResourceApadter
            </instance-name>
         </resource-adapter-ref>
      </message-source>
   </message-driven>
</enterprise-beans>
```

リソースアダプタには、さまざまな管理オブジェクトを表す JavaBean クラスのオプションのセットである Java クラス名とインターフェース型があります。管理オブジェクトは、メッセージングスタイルまたはメッセージプロバイダに固有で、MDB のアプリケーションロジックから <resource-env-ref> を使用して参照できます。たとえば、一部のメッセージングスタイルでは、アプリケーションが特定の管理オブジェクトを使用し、メッセージングスタイル固有の API を使用して、接続オブジェクト経由でメッセージを送信および同期受信することが必要です。Borland デプロイメントデスクリプタ要素 <resource-env-ref> は拡張され、管理オブジェクトのプロパティ値を上書きします。たとえば、次のようになります。

```
...
  <message-driven>
     <message-source>
        <resource-adapter-ref>
           <instance-name>ResourceAdapter1</instance-name>
        </resource-adapter-ref>
     </message-source>
     ...
     <resource-env-ref>
        <resource-env-ref-name>mdbRequiredConnFactory</resource-env-ref-name>
        <admin-object>
           <property>
               <prop-name>serverUrl</prop-name>
               <prop-type>String<prop-type>
               <prop-value>localhost:7222</prop-value>
           </property>
        </admin-object>
     </resource-env-ref>
```

```
    ...
</message-driven>
...
```

MDB のクラスタリング

MDB のクラスタリングは、他のエンタープライズ Bean のクラスタリングとは異なります。MDB の場合、プロデューサが宛先にメッセージを転送します。メッセージはコンシューマがメッセージを宛先から取り出すまで（メッセージに永続性がない場合は、ホストサーバーがクラッシュするまで）宛先に残ります。これは、*pull* モデルです。コンシューマが要求するまでメッセージが宛先に残るからです。コンテナは、宛先で次に利用可能なメッセージを求めて競合します。MDB は、理想的な負荷分散パラダイムを備えており、他のエンタープライズ Bean インプリメンテーションの場合よりスムーズに負荷を分散できます。最も負荷が小さなサーバーがメッセージを要求し、取得できます。この最適負荷分散の欠点は、プロデューサとコンシューマ間の宛先の位置により、メッセージングでコンテナに余分な負担がかかることです。

ただし、VisiBroker にあるようなメッセージングサービスに関するフェイルオーバーと同じ概念はありません。コンシューマがいなくなれば、キューにはメッセージが代入されます。コンシューマがオンラインに戻ると、メッセージの消費は再開されます。もちろん、JMS サーバー自体はフォールトトレラントでなければなりません。このようなメッセージが期待される状況では、応答遅延を除き、クライアント側に「障害」が認識されるのは避けなくてはなりません。この種のフォールトトレラントで必要なことは、障害コンシューマの検出方法と、障害後に起動する方法だけです。

つまり、メッセージングサーバーで 1 つ以上のパーティションに MDB を配置しておけば、メッセージの転送先 1 か所でも、いざ障害が発生すると別のパーティションに切り替えることができるということです。ほとんどの JMS 製品では、負荷分散モードまたはフォールトトレラントモードでキューを操作できます。つまり、MDB 複製を同じキューに登録しておくと、メッセージが負荷分散アルゴリズムにしたがってデプロイメントされます。あるいは、障害が発生するまですべてのメッセージの宛先を 1 つのコンシューマとし、障害が発生したら別のコンシューマに切り替える方法もあります。MDB から JMS サービスプロバイダに確立される接続では、負荷分散ノードとフォールトトレラントノードの両方またはどちらかを提供できます。JMS サービスプロバイダには、フォールトトレランス機能があります。クラスタリングとフォールトトレランス機能の詳細は、221 ページの「JMS プロバイダの接続性」を参照してください。

ちなみに、どのメッセージもそれを消費するのは、トピックをサブスクライブしているコンテナにある MDB インスタンスだけです。つまり、MDB の並列インスタンスでメッセージを同時処理するとき、メッセージを受け取るのはインスタンスのいずれか 1 つだけだということです。これにより、他のインスタンスは、トピックに転送された他のメッセージを処理できます。なお、特定のトピックにバイドされたコンテナは、そのトピックに転送されたメッセージを消費します。JMS サブシステムは各メッセージ駆動型 Bean を、メッセージに対する独立したサブスクライバとして別々のコンテナで処理します。つまり、クラスタ内の複数のコンテナに同じ MDB をデプロイメントしておくと、各 Bean のデプロイメントは、サブスクライブするトピックからメッセージを消費します。このような振る舞いが必要でなく、メッセージの消費は 1 か所だけでよい場合、トピックではなくキューのデプロイメントを考えてください。

エラーからの回復

次のセクションでは、JMS サーバーの接続エラーと、接続のリバインド試行に関するプロパティの設定について説明します。また、MDB がメッセージの受信に失敗した場合のメッセージの再配信についても説明します。

JMS プロバイダメッセージソースによって設定された EJB 2.0 および EJB 2.1 MDB のリバインド

接続エラーは通常 Bean のデプロイメント後に発生し、リバインドの試行が必要になります。Bean をデプロイメントしようとして JMS サーバーの接続が確立されていなかった場合にもエラーが発生します。デプロイメント後にエラーが発生する場合でも、デプロイメント時に接続されていなかった場合でも、リバインドの試行に関するプロパティを設定しておくと、コンテナは透過的に JMS サービスプロバイダの接続をリバインドしようとします。これにより、MDB インスタンスのフォールトトレランスを強化できます。

実行されるリバインド試行回数と試行間の時間間隔を制御する Bean レベルプロパティは次の 2 つです。

- `ejb.mdb.rebindAttemptCount`：現在の MDB において、失敗した JMS 接続を EJB コンテナが再試行する回数。デフォルト値は `5` です。

  コンテナによる試行回数に上限を設定しない場合は、`ejb.mdb.rebindAttemptCount=0` を明示的に指定する必要があります。
- `ejb.mdb.rebindAttemptInterval`：連続する 2 つの再試行の間の時間間隔を秒数で表したもの。デフォルト値は、`60` です。

JMS プロバイダメッセージソースによって設定された EJB 2.0 および EJB 2.1 MDB に対して再配信されたメッセージ

MDB がなんらかの理由でメッセージの受信に失敗した場合、メッセージは JMS サービスによって再配信されます。メッセージは 5 回まで再配信されます。5 回の試行の後、メッセージはデッドキューに配信されます（設定されている場合）。再配信の試行回数を制御する Bean レベルのプロパティは次の 1 つです。

- `ejb.mdb.maxRedeliverAttemptCount`：MDB がメッセージを受信できない場合に JMS サービスプロバイダによって再配信されるメッセージの最大数。デフォルト値は、`5` です。

メッセージをデッドキューに配信するための Bean レベルのプロパティは、次の 2 つです。

- `ejb.mdb.unDeliverableQueueConnectionFactory`：JMS サービスの接続を作成するために接続ファクトリの JNDI 名を検索します。
- `ejb.mdb.unDeliverableQueue`：キューの JNDI 名を検索します。

  `unDeliverableQueueConnectionFactory` と `unDeliverableQueue` の XML サンプルは次のとおりです。

```
<ejb-jar>
   <enterprise-beans>
       <message-driven>
           <ejb-name>MyMDB</ejb-name>
          <message-driven-destination-name>serial://jms/q
            </message-driven-destination-name>
           <connection-factory-name>serial://jms/xaqcf
            </connection-factory-name>
           <pool>
               <max-size>20</max-size>
               <init-size>0</init-size>
           </pool>
           <resource-ref>
               <res-ref-name>jms/QueueConnectionFactory</res-ref-name>
               <jndi-name>jms/xaqcf</jndi-name>
           </resource-ref>
```

```
                <property>
                    <prop-name>ejb.mdb.maxRedeliverAttemptCount</prop-name>
                    <prop-type>String</prop-type>
                    <prop-value>3</prop-value>
                </property>
                <property>
                    <prop-name>ejb.mdb.unDeliverableQueueConnectionFactory
                       </prop-name>
                    <prop-type>String</prop-type>
                    <prop-value>serial://jms/qcf</prop-value>
                </property>
                <property>
                    <prop-name>ejb.mdb.unDeliverableQueue</prop-name>
                    <prop-type>String</prop-type>
                    <prop-value>serial://jms/q2</prop-value>
                </property>
                <property>
                    <prop-name>ejb-designer-id</prop-name>
                    <prop-type>String</prop-type>
                    <prop-value>MyMDB</prop-value>
                </property>
            </message-driven>
        </enterprise-beans>
        <assembly-descriptor />
    </ejb-jar>
```

DDEditor には、次のようなプロパティを設定できます。コンソールから、左側のツリーに移動し、自分の MDB があるモジュールを探します。モジュールを右クリックし、［DDEditor］を選択します。DDEditor が表示されたら、ナビゲーションペインで Bean ノードを選択し、Bean に対応するエディタのパネルを開きます。内容ペインの［Properties］タブを選択し、プロパティを追加します。

MDB とトランザクション

トランザクションにおける JMS の使い方の詳細は、209 ページの「JMS とトランザクション」を参照してください。このセクションでは、トランザクションにおける MDB についてのみ説明します。

MDB を使用する一般的な状況としては、2 フェーズコミット（2PC）を必要とするトランザクションがあります。そのような MDB には、`REQUIRED` トランザクション属性が割り当てられます。MDB アプリケーションメソッドは、外部リソースにアクセスして更新するために記述される場合があります。MDB メソッドに対応するコンテナ管理トランザクションを完了するには、メソッドをトリガーしたメッセージを受け取り、外部リソースに対するすべての作業をメソッドから実行する必要があります。そのために、OTS エンジンなどの 2PC トランザクションサービスでトランザクションを調整する必要があります。MDB で OTS エンジンを最適に使用する方法の詳細ついては、「トランザクション管理」「157 ページの「EJB と 2PC トランザクション」を参照してください。

第 20 章

Java Bean 型のオブジェクトを JNDI に登録する

この章では、Java Bean 型のオブジェクトを JNDI に登録し、JNDI から検索する方法を説明します。

Java Bean 型のオブジェクトを JNDI に追加する

Java Bean 型のオブジェクトを JNDI に追加するには：

1. `jndi-object` という新しいエントリを `jndi-definitions.xml` ファイルに追加します。
2. このオブジェクトの名前を `jndi-name` に設定します。この名前はオブジェクトを JNDI にバインドするためやクライアントでオブジェクトを検索するために使用されます。
3. このオブジェクトのクラスを `class-name` に設定します。ライブラリとしてデプロイメントされるので、パーティションのクラスパスにクラスを配置する必要があります。
4. プロパティのリストを設定します。各プロパティ エントリは、プロパティ名、プロパティ型、プロパティ値で構成されます。

例

```
<jndi-object>
   <jndi-name>TestObject</jndi-name>
   <class-name>examples.j2ee.jndi.Foo</class-name>
   <property>
       <prop-name>street</prop-name>
       <prop-type>String</prop-type>
       <prop-value>Park west</prop-value>
   </property>
   <property>
       <prop-name>postal</prop-name>
       <prop-type>String</prop-type>
       <prop-value>1262445</prop-value>
   </property>
</jndi-object>
```

第 21 章

Borland AppServer を使用したリソースへの接続：定義アーカイブ（DAR）の使い方

J2EE は、Java 標準インターフェースを使用するリソースとの接続を確立するための統一メカニズムを指定します。リソースマネージャの場所の詳細と接続属性を含むリソース関連オブジェクトは、JNDI サービスプロバイダの下でバインドされており、アプリケーション JNDI 検索のリソース接続ファクトリとして取得できます。サンプルのリソース接続ファクトリとしては、JDBC データソースと JMS 接続ファクトリがあります。JNDI からリソース接続ファクトリを取得すると、目的のリソースマネージャへの接続を確立できます。リレーショナルデータベースへの接続は JDBC データソースを介して取得し、メッセージブローカーへの接続は JMS 接続ファクトリを介して取得し、一般企業情報システム（EIS）接続は JCA リソースアダプタを介して取得します。

リソース接続ファクトリおよび JMS の送信先などのその他のリソース関連 JNDI オブジェクトを作成、編集、およびデプロイメントするには、Borland 管理コンソール と Borland デプロイメントデスクリプタエディタ（DDEditor）を使用します。一般に JNDI 定義モジュールと呼ばれる XML デスクリプタファイル（`jndi-definitions.xml`）は、リソース関連オブジェクトを表すプロパティを取得します。このファイルは Data ARchive（DAR）モジュールにパッケージされています。

Borland AppServer （AppServer）でパーティションがホストするネーミングサービスは、CosNaming サービスプロバイダのインプリメンテーションであるデフォルトの JNDI サービスプロバイダを表します。リソース関連オブジェクトは、標準 AppServer デプロイメント手順を使用する DAR モジュールまたは RAR モジュールのデプロイメントを介して、AppServer パーティションのネーミングサービスでバインドされます。その場合、リソース接続ファクトリのインスタンスまたは JMS 送信先を作成するために必要なプロパティだけが JNDI にバインドされたオブジェクトに保存されます。リソース関連オブジェクトの JNDI 検索中に、目的のリソースオブジェクトのインスタンスは、取得されたオブジェクトから保存されたプロパティ値を使って作成されます。新しく作成されたインスタンスは、JNDI `lookup()` メソッドの呼び出し元に返されます。このようにして、DAR はベンダー固有のリソースオブジェクトのクラスをロードせずに AppServer パーティションに正しくデプロイメントできます。リソースベンダーのクラスライブラリだけは、リソース関連オブジェクトの JNDI 検索を実際に実行するアプリケーションプロセスに必要です。

メモ AppServer の古いバージョンでは、シリアルプロバイダと呼ばれるファイルシステムサービスプロバイダが DAR モジュールと JNDI 定義モジュールをデプロイメントするデフォルトの JNDI サービスプロバイダでした。このプロバイダにバインドされたリソース関連オブジェクトはデプロイメント時のリソースオブジェクトの作成に関与しているので、事前にベンダークラスライブラリをデプロイメントする必要がありました。さらに、リソース関連オブジェクトの JNDI 名には "serial://" というシリアル URL プレフィクスが必要でした。ネーミングサービスをデフォルトサービスプロバイダにすれば、JNDI の名前仕様にこのプレフィクスは必要なくなります。このプレフィクスが付いている JNDI 名を持つ既存の DAR/JNDI 定義モジュールのデプロイメントは、自動的にネーミングサービスにバインドされます。

J2EE のリソース関連オブジェクトは、リソースリファレンスを介して取得されます。コンポーネントのデプロイメントデスクリプタ内のリソースリファレンス要素を使用して、EJB、サーブレットおよびその他の J2EE アプリケーションコンポーネントからリソース接続ファクトリまたは JMS 送信先を参照できます。JDBC データソースのリソースリファレンスの定義の詳細については 185 ページの「JDBC の使い方」のセクションを参照し、JMS 接続ファクトリと送信先のリソースリファレンス定義のサンプルについては 199 ページの「JMS の使い方」のセクションを参照してください。

各 AppServer パーティションには `default-resources.dar` という名前のデプロイメント済みの DAR モジュールがあり、それには JDBC データソース、JMS 接続ファクトリ、および JMS 送信先の定義サンプルがあります。このモジュールは、次の手順を使って検査、更新、および再デプロイメントできます。

1. Borland Management コンソールの左側ペインで、パーティションのデプロイメントモジュールノードの **default-resources.dar** に移動します。
2. **default-resources.dar** を右クリックし、コンテキストメニューから［**Edit deployment descriptor**］を選択します。Borland デプロイメントデスクリプタエディタ（DDEditor）ウィンドウが表示されます。利用可能なデータソースと接続ファクトリが左側ペインに表示されます。
3. Borland デプロイメントデスクリプタエディタのナビゲーションペインのルートノードを右クリックし、適切なオプションを選択して追加するオブジェクトを新規作成します。

J2EE コンポーネントがリソースリファレンスの JNDI 検索を実行する場合、実行時環境でリソースオブジェクトに関連付けられたベンダークラスが使用可能になっている必要があります。J2EE コンポーネントを AppServer パーティションにデプロイメントする場合、ベンダークラスライブラリをライブラリアーカイブとして AppServer パーティションにデプロイメントする必要があります。この規則の例外としては、依存するクラスライブラリが AppServer にバンドルされているリソースオブジェクトの JNDI 検索があります。この例としては、JDataStore データソースまたは AppServer とともにインストールされる JMS メッセージサーバーの任意の JMS リソースオブジェクトがあります。

JNDI 定義モジュール

リソース関連オブジェクトは、JNDI 定義モジュールを含む DAR ファイルのデプロイメントを介してネーミングサービスにバインドされます。DAR ファイルには特別な .dar ファイル拡張子が付けられます。DAR ファイルは、個別またはほかの J2EE モジュールとともに EAR ファイルにパッケージして AppServer パーティションにデプロイメントする必要があります。

メモ DAR は、J2EE 仕様の一部では*ありません*。これは、Borland 固有のインプリメンテーションであり、リソース接続ファクトリと JMS 送信先を簡単にデプロイメントしたり管理することを目的としています。接続ファクトリクラスまたは JMS 送信先ベンダークラスは、このアーカイブタイプにパッケージ*しません*。これらのクラスは、ライブラリとして個別のパーティションにデプロイメントしてください。

必要になる DAR の唯一のコンテンツは、`jndi-definitions.xml` という XML デスクリプタファイルです。このファイルにはリソース関連オブジェクトの定義が含まれます。各オブジェクトには、JNDI 内の場所を特定する JNDI 名が指定されています。ほかのデスクリプタと同様に、DAR の `META-INF` ディレクトリに配置されます。したがって DAR の内容は次のとおりです。

```
META-INF/jndi-definitions.xml
```

デスクリプタファイルを含む DAR は、コンソールまたはコマンドラインユーティリティを使用したり、あるいは EAR の一部としてほかの J2EE モジュールをデプロイメントするようにデプロイメントします。名前付き DAR は、同じパーティションまたは AppServer クラスタにいくつでもデプロイメントできます。2 つ以上のデプロイメントされた DAR に同じ JNDI 名のリソースオブジェクトの定義がある場合は、後でデプロイメントしたモジュールが同じノードにバインドされている既存オブジェクトを上書きします。

DAR で定義され、デプロイメントされたリソースオブジェクトは、『*管理コンソールユーザーズガイド*』の「JNDI ブラウザ」を使ってネーミングサービスの名前空間で検査できます。

Borland AppServer の前バージョンから DAR に移行

IAS 4.1 や BAS 4.5 など前バージョンの製品には、`jndi-definitions.xml` デスクリプタを収める DAR モジュールがありません。カスタマイズした `jndi-definitions.xml` ファイルを AppServer に変換する場合は、次の手順にしたがいます。

1 デフォルトリソースの内容をすべて上書きする場合、`META-INF` という名前で一時ディレクトリを作成し、既存の `jndi-definitions.xml` ファイルとともに保存します。

2 コマンドウィンドウを開き、次の `jar` コマンドを実行します。

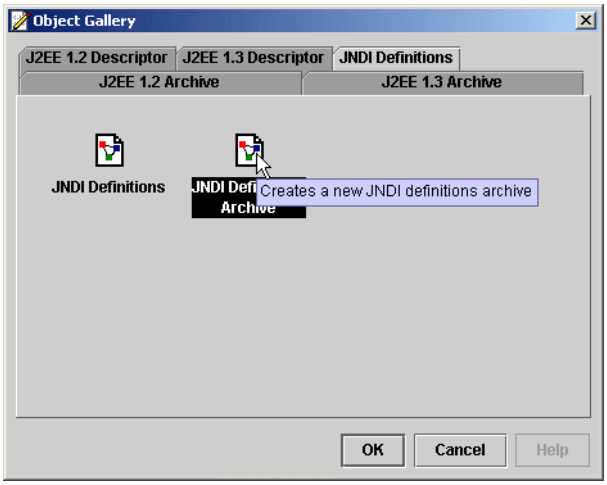
```
prompt>jar uvMf default-resources.dar META-INF/jndi-
definitions.xml
```

3 通常の手順で、このモジュールをデプロイメントします。

古い `jndi-definitions.xml` ファイルをほとんど変更していない場合、古いファイルからデプロイメント済み DAR にあるファイルに、該当する XML 行を移動するのは簡単です。

DAR の作成とデプロイメント

JNDI 定義モジュールを新規作成するには、DDEditor の指示にしたがって操作します。DDEditor を開き、［File | New...］を選択します。［Object Gallery］ウィンドウが開きます。



［JNDI Definitions］タブを選択し、［JNDI Definitions Archive］を選択して新しい DAR を作成します。［OK］をクリックします。これで、JDBC データソースまたは JMS リソースの追加が終了しました。あるいは、後から実行することもできます。操作が終了したら、［File ｜ Save As］を選択してモジュールを保存します。

アーカイブを保存したら、J2EE デプロイメントウィザードでモジュールをデプロイメントします。ウィザードは DAR からリソース定義を読み取ってターゲットパーティションのネーミングサービスにバインドします。ウィザードを開始するには、コンソールを開いて［Wizards | Deployment Wizard］を選択します。画面に表示される指示にしたがいます。

デプロイメントされた DAR の有効化と無効化

DAR モジュールがパーティションにデプロイメントされると有効になります。つまり、ネーミングサービスがアクティブになっている間、リソースオブジェクトの定義がパーティションのネーミングサービスにバインドされます。DAR モジュールを有効にすると、リソースオブジェクトの定義がネーミングサービスにリバインドされ、プロセスでは、指定されている JNDI 名の既存の内容が上書きされます。DAR モジュールを無効にしても、アクティブなネーミングサービスの内容に直接的な影響はありません。その後にパーティションを再起動すると、無効になった DAR はパーティションにデプロイメントされないので、リソースオブジェクトの定義はネーミングサービスにバインドされません。デフォルトでは、ネーミングサービスはオブジェクトバインディングをメモリに保存します。ホストパーティションが再起動されるたびに、それまでデプロイメントされていた DAR のリソースオブジェクトのバインディングは破棄されます。ネーミングサービスが JDBC バッキングストアで設定されている場合、デプロイメントされた後に無効になったバインディングも含めて、すべての DAR のリソースオブジェクトのバインディングが維持されます。このようなバインディングは、JNDI ブラウザを使って検索して完全に削除します。

デプロイメントされた DAR モジュールを操作するには、コンソールを使ってパーティションにデプロイメントされたモジュールのセットから選択し、右クリックして適切なアクションを選択します。

アプリケーション EAR の DAR モジュールのパッケージ

完全なアプリケーションを構成するアーカイブをすべて 1 つのデプロイメントユニットにパッケージすると便利な場合があります。たとえば、EJB アーカイブに EJB があり、Web アーカイブにサーブレットと JSP があり、DAR で定義したデータソースまたは JMS 管理オブジェクトに依存しているとします。コンソールのアーカイブツールを使用すれば、アーカイブを 1 つの EAR モジュールに簡単にパッケージできます。

メモ DAR は J2EE 仕様を構成しないので、DAR とともに少なくとも有効な J2EE モジュールをもう 1 つ EAR にインクルードする必要があります。DAR ファイルを格納する EAR は、有効な J2EE アーカイブの一部ではありません。

第 22 章

JDBC の使い方

JDBC データソースなどのリソース関連オブジェクトは、移植可能な J2EE の規定の方法で JNDI を介して取得できます。JDBC データソースは、アプリケーションコンポーネントのデプロイメントデスクリプタで定義された J2EE リソースリファレンスの JNDI 検索を実行することによって解決されます。リソースリファレンス定義には、標準 J2EE デプロイメントデスクリプタと Borland 独自のデプロイメントデスクリプタの両方を使用します。標準デプロイメントデスクリプタでは、リソースリファレンスはアプリケーションの JNDI 環境ネーミングコンテキストである `java:comp/env/` に基づいて論理名を指定します。Borland のデプロイメントデスクリプタは、リソースリファレンスの論理名を JDBC リソース定義の実際の JNDI ロケーションに関連付けることによって標準デスクリプタを補足します。たとえば EJB JAR コンポーネントでは、標準 J2EE デプロイメントデスクリプタ `ejb-jar.xml` は JDBC データソースの <resource-ref> 要素を使って EJB のリソースリファレンスを指定します。Borland AppServer（AppServer）におけるリソースリファレンスの JNDI 検索では、JDBC データソース定義を取得し、その定義から目的のデータソースオブジェクトを作成して検索の呼び出し元に返します。JDBC データソース定義にあるプロパティ値が作成されるデータソースオブジェクトのタイプと特性を決定します。

リソースリファレンスの検索を実行する前に、まず必要なデータソース定義を物理的な JNDI ロケーションにバインドする必要があります。AppServer では JDBC データソース定義は、定義アーカイブ（DAR）モジュールのデプロイメント中に JNDI サービスプロバイダにバインドされます。デフォルトでは、このオブジェクトは BES の JNDI CosNaming サービスプロバイダであるパーティションネーミングサービスにバインドされます。この章では、DAR モジュールの JDBC データソースを定義する方法、および各自の J2EE アプリケーションから JDBC データソースのリファレンスを取得する方法について説明します。

JDBC データソースの設定

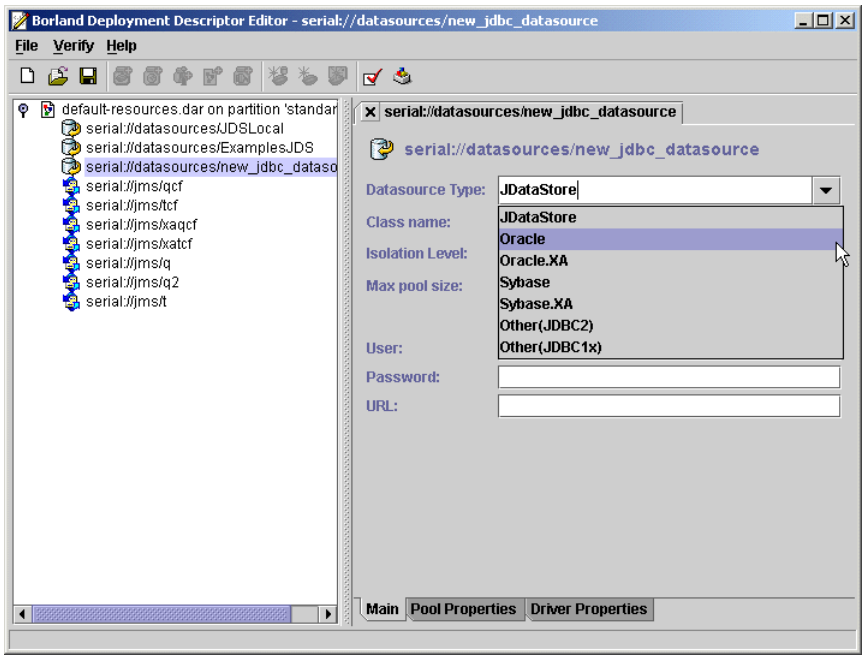
コンソールを利用して、データソースを設定するパーティションの［Deployed Modules］リストに移動します。デフォルトでは、すべてのパーティションに、`default-resources.dar` というデプロイメント済みの JNDI 定義モジュール（DAR）があります。そのモジュールを右クリックし、コンテキストメニューから［Edit deployment descriptor］を選択します。デプロイメントデスクリプタエディタ（DDEditor）が表示されます。

DDEditor のナビゲーションペインには、製品で設定済みのデータソースがリストされます。これらは、必要に応じてユーザーの要件に合わせて個別に編集できます。

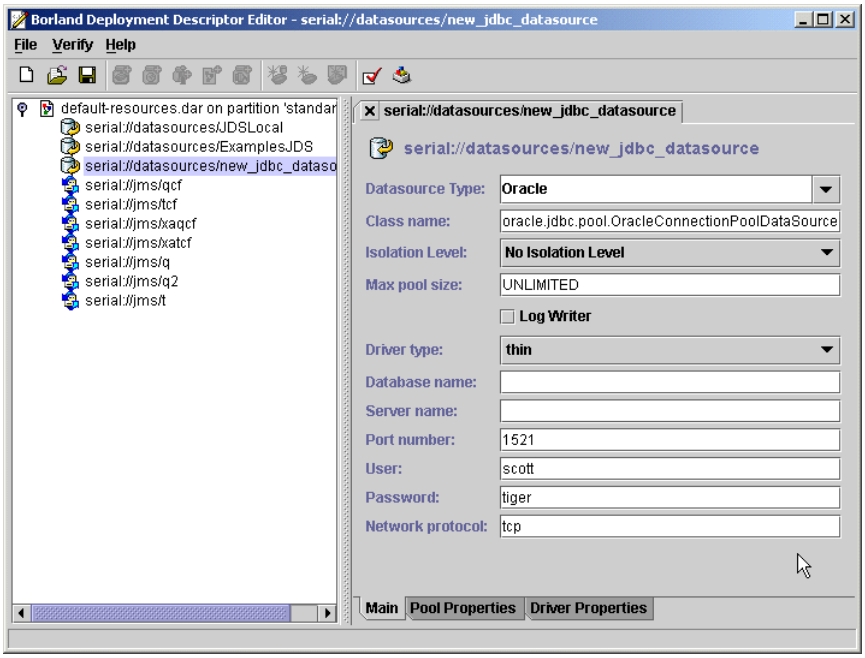
新しい JDBC データソースを作成するには、ナビゲーションペインのツリー最上部のノードを右クリックし、コンテキストメニューの［New Jdbc Datasource］を選択します。

新しく作成したデータソースの JNDI 名を要求するダイアログボックスが表示されます。JNDI 名を指定すると、このデータソースの表示がナビゲーションペインのツリーに表示されます。表示をクリックして設定パネルを開きます。

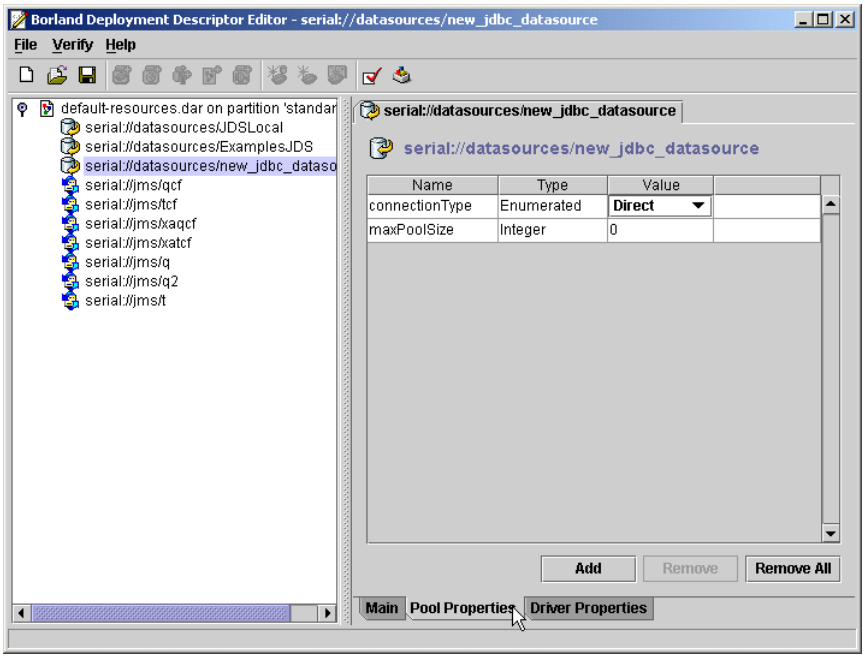
DDEditor には、使用頻度が高い JDBC ドライバに関する情報が組み込まれており、該当する JDBC データソースのクラス名や基本プロパティには値が自動的に入力されます。目的の JDBC データソースが［Datasource Type］リストに表示されたら、選択します。表示されない場合は、［Other(JDBC2)］を選択します。



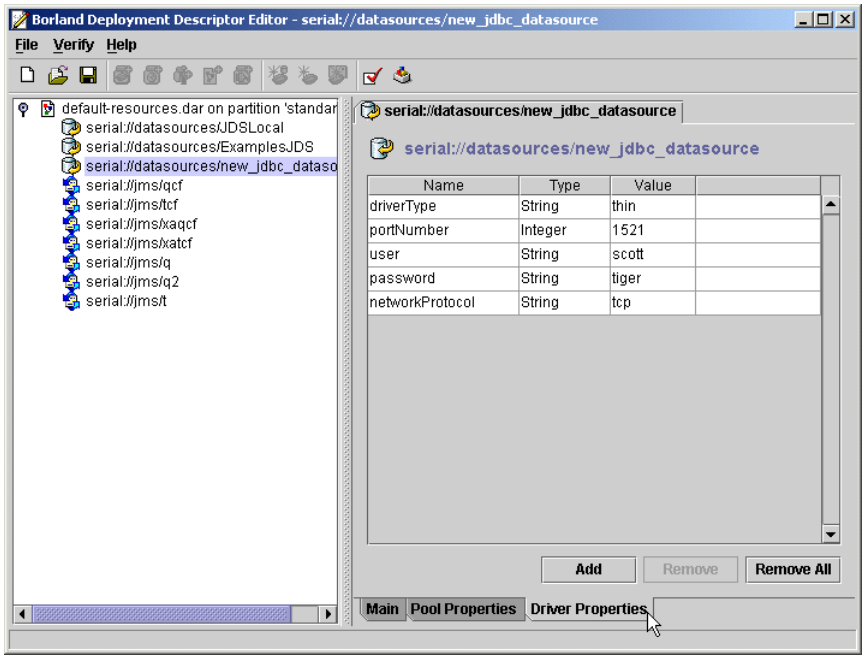
データソースの選択に必要な基本プロパティが内容ペインの［Main］タブに表示されます。データソースが DDEditor に登録済みであれば、以上のプロパティには自動的に値が入力されます。



［Driver Properties］タブや［Pool Properties］タブに、［Main］タブの情報の一部が表示されますが、［Main］タブに表示されない共通性の少ないプロパティもここで設定できます。



プールのプロパティを追加するには、［Add］ボタンをクリックし、［Name］の下のドロップダウンリストから追加するプロパティを選択します。プールのプロパティについては、189 ページの「JDBC データソースの接続プールプロパティの定義」参照してください。ドライバのプロパティを追加する手順も同じです。



具体的に定義するプロパティについては、データベースのマニュアルを参照してください。

操作が終了したら、モジュールを保存し、最後のモーダルウィンドウを閉じます。JNDI 定義モジュールはパーティションに自動的に再デプロイメントされます。

ドライバライブラリのデプロイメント

デプロイメントされたアプリケーションコンポーネントにサードパーティ製の JDBC データソースの JNDI 検索が含まれている場合はベンダーライブラリが必要で、検索を実行する前にライブラリアーカイブとして目的のパーティションにデプロイメントする必要があります。ネイティブの完全 Java データベースである JDataStore を使用している場合、この手順は必要ありません。Oracle や Sybase など、別のデータベースに接続するには、個々の JDBC ドライバを、まずターゲットパーティションにデプロイメントします。ライブラリを複数のパーティションにデプロイメントするには、次の作業を実行します。

1 コンソールの［Wizard］メニューで［Deployment Wizard］を選択します。デプロイメントウィザードが開きます。

2 ［Add］ボタンをクリックして表示されるウィンドウでライブラリファイルに移動し、［OK］をクリックします。ライブラリ名がデプロイメントウィザードの選択ボックスに表示されます。

3 ［Next］ボタンをクリックします。パーティションの名前がデプロイメントウィザードウィンドウに表示されます。

4 ライブラリをデプロイメントするパーティションを選択して［Finish］ボタンをクリックします。デプロイメントステータスが別のウィンドウに表示されます。

5 ［Close］ボタンをクリックしてこのウィンドウを閉じます。管理 コンソールのナビゲーションペインのパーティションの［Deployed Modules］ノードをチェックすれば、ライブラリが正常にデプロイメントされたことを検証できます。ライブラリの名前が［Deployed Modules］ノードに表示されます。

6 デプロイメントを有効にするために、パーティションを停止してから再起動します。

次の手順で 1 つのパーティションにライブラリをデプロイメントします。

1. BES コンソールのナビゲーションペインでパーティションの名前を右クリックし、コンテキストメニューから［Deployed Modules］を選択します。デプロイメントウィザードが開きます。
2. ［Add］ボタンをクリックして表示されるウィンドウでライブラリファイルに移動し、［OK］をクリックします。ライブラリ名がデプロイメントウィザードの選択ボックスに表示されます。
3. ［Next］ボタンをクリックします。パーティション名がデプロイメントウィザードウィンドウに表示されます。
4. ライブラリをデプロイメントするパーティションを選択して［Finish］ボタンをクリックします。デプロイメントステータスが別のウィンドウに表示されます。
5. ［Close］ボタンをクリックしてこのウィンドウを閉じます。管理 コンソールのナビゲーションペインのパーティションの［Deployed Modules］ノードをチェックすれば、ライブラリが正常にデプロイメントされたことを検証できます。ライブラリの名前が［Deployed Modules］ノードに表示されます。
6. デプロイメントを有効にするために、パーティションを停止してから再起動します。

JDBC データソースの接続プールプロパティの定義

実行時には、各 JDBC データソースは接続プールのインスタンスに関連付けられます。接続プールは接続の再利用をサポートし、データベース接続を最適化します。データソースによっては、接続プールとしてほかのデータソースとは異なる措置が必要なものがあります。そのような接続プールには数多くの設定オプションが用意されています。プールサイズの制御、文の実行時の振る舞い、トランザクションパラメータは、DAR デスクリプタファイルの `<visitransact-datasource>` 要素でプロパティとして指定します。プロパティは、`<property>` 要素で指定します。そのような要素には `<prop-name>` 要素、`<prop-type>` 要素、`<prop-value>` 要素があります。次の表に、すべてのプロパティ、指定できる値、デフォルト、説明を一覧表示します。

名前	指定できる値	説明	デフォルト値
FinalizeNoTxBusyConnections	このプロパティは、他の接続プールプロパティとは異なり、partition_server.config を編集し、次の行を追加して BAS パーティションで設定する必要があります。vmpram -DFinalizeNoTxBusyConnections この設定は、パーティション内のアクティブなすべての BAS JDBC 接続プールに反映されます。	JDBC 接続が NoTxBusy の状態になったら、アプリケーションは接続を閉じる必要があります。閉じないと、JDBC 接続プールはいつまでも参照を続け、基底のデータベース接続は解放されません。このプロパティを設定すると、BES 接続プールは JDBC 接続の弱参照を維持するように設定され、スコープ外の NoTxBusy 接続はアプリケーションによって閉じられていない場合、JVM のガベージコレクションが発生すると解放されます。	
connectionType	Enumerated: ■ Direct ■ XA	接続プールから取り出すすべての接続のトランザクションの関連。「Direct」か「XA」。	適用なし。プロパティの指定は必須です。

名前	指定できる値	説明	デフォルト値
optimizeXA	Boolean	XAResource API 呼び出しは、AppServer JDBC 接続プールのパフォーマンスの最適化のために、デフォルトで最小限に保たれます。optimizeXA の値を false に設定すると、この最適化が無効になります。一定の条件下では、データソースの optimizeXA プロパティを false に設定する必要があります。たとえば、2 フェーズコミット時に、AppServer JDBC 接続プールのデフォルトの XAResource 最適化と、Oracle など特定のベンダーリソースマネージャ間に競合が発生する場合があります。optimizeXA を false に設定する場合、分散トランザクションのスコープで JDBC 接続を使用するアプリケーションは、トランザクションの完了前に接続の close() を発行する必要があります。そうしなければ、予期しないトランザクション完了条件が発生します。	True
maxPoolSize	Integer	データソース接続プールから取得できるデータベース接続の最大数を指定します。	0 は無制限サイズを指定します。
waitTimeout	Integer	maxPoolSize 接続が開かれているときに、接続が解放されるまで待つ時間を秒単位で指定します。maxPoolSize プロパティを使用しており、プールがいっぱいで、これ以上接続を使用できない場合は、JDBC 接続を検索するスレッドは、待ち時間が無制限に設定されている（0 秒に設定）と、その接続が使用できるようになるまで待機します。必要に応じて、waitTimeout 時間を設定できます。	30
busyTimeout	Integer	ビジー接続が解放されるまで待つ時間を秒単位で指定します。	600（10 分）
idleTimeout	Integer	このタイムアウトを超えてアイドル状態が続いたプールされた接続は閉じられます。アイドル接続に対して、60 秒ごとに idleTimeout の期限切れが確認されます。idleTimeout の値の単位は秒数です。	600（10 分）
queryTimeout	Integer	このデータソースでデータベースクエリーを実行するときの制限時間を秒単位で指定します。	0 は無限時間を指定します。

名前	指定できる値	説明	デフォルト値
dialect	Enumerated: ■ oracle ■ sybase ■ interbase ■ jdatastore	コンテナ管理の永続性の間に実行する自動テーブル作成のヒントとしてデータベースベンダーを指定します。	このプロパティはオプションです。デフォルト値はありません。
isolationLevel	Enumerated: ■ TRANSACTION_NONE ■ TRANSACTION_READ_COMMITTED ■ TRANSACTION_READ_UNCOMMITTED ■ TRANSACTION_REPEATABLE_READ ■ TRANSACTION_SERIALIZEABLE	現在のデータソースの接続プールで開かれたすべての接続に関連付けられたデータソースの分離レベルを示します。以上の分離レベルの詳細については、J2EE 1.3 仕様を参照してください。	デフォルトレベルは、JDBC ドライバベンダーが提供します。
reuseStatements	Boolean	再利用のためにキャッシュする準備 SQL 文を要求する最適化指示文。接続プールから取得するすべての接続に適用します。	True
initSQL	String	新たなトランザクション用に接続を取得するたびに実行する「;」区切りの SQL 文のリストを指定します。SQL は、接続でアプリケーション作業が実行される前に実行されます。	このプロパティはオプションです。デフォルト値はありません。
refreshFrequency	Integer	dbPingSQL を使用する場合、このプロパティは、アイドル状態の接続を閉じるまでの時間を秒単位で指定します。タイムアウトが経過すると、接続が有効な接続かどうかを確認されます。アイドル接続に対しては、60 秒ごとに refreshFrequency のタイムアウトが確認されます。	300（5 分）
dbPingSQL	String	refreshFrequency の間、接続プールにある開いている接続を検査し、接続をリフレッシュするための SQL 文を指定します。	定義されていません。SQL を指定しない場合、コンテナは、java.sql.ConnectionisClosed() メソッドを使って接続を検査します。
resSharingScope	Enumerated: ■ Shareable ■ Unshareable	接続文と結果セットを再利用のためにキャッシュするかどうかを示します。Shareable に設定すると、接続文と結果セットはキャッシュされ、接続スループットが最適化されます。Unshareable の場合、アプリケーションが接続を閉じるたびに接続が閉じられます。	Shareable

名前	指定できる値	説明	デフォルト値
maxPreparedStatementCacheSize	Integer	AppServer JDBC プール内の各接続は、再利用のために java.sql.PreparedStatement オブジェクトをキャッシュします。 各 PreparedStatement キャッシュは、SQL 文による一意の要求を表す SQL リテラル文字列によって整理されます。 このプロパティは、プールされる JDBC 接続ごとに、キャッシュされる PreparedStatements の数を制限します。このプロパティは、キャッシュの最大サイズを指定します。キャッシュが上限に達すると、その後の javax.sql.Connection.prepareStatement() 呼び出しでは、作成される PreparedStatement オブジェクトのインスタンスはキャッシュされずに、呼び出し元に戻されます。キャッシュの存続期間は、JDBC 接続の存続期間と同じです。たとえば、アイドル接続がタイムアウトになると、接続と PreparedStatement キャッシュは両方とも破棄されます。未解決のパラメータ化された SQL 文がキャッシュされます。たとえば、SELECT NAME FROM CUSTOMER WHERE AGE=20 という文は、SELECT NAME FROM CUSTOMER WHERE :age='?' としてキャッシュされます。このプロパティは、データソースの reuseStatements プロパティが true（デフォルト）に設定されているときだけ有効です。デフォルト値は 40 ですが、一般のアプリケーションではこれで十分です。	40

名前	指定できる値	説明	デフォルト値
maxPreparedStatementsPerQuery	Integer	並行性が高い場合や、CPM 2.0 エンティティ Bean を処理する場合などの特定の条件下では、同じプールされた接続上の同じ SQL クエリーで、複数の PreparedStatement を同時に処理できます。たとえば、SELECT name FROM table1 WHERE id=? という SQL クエリーは、? に異なる値が使用されると、異なる結果セットを返します。PreparedStatement キャッシュには SQL クエリーごとに 1 つのエントリがありますが、クエリー用のキャッシュでは、2 つ以上の PreparedStatements が存在することができます。 このプロパティは、1 つのクエリーでキャッシュされる PreparedStatements の最大数を指定します。特定のクエリーでこの制限を超えた場合、その後の javax.sql.Connection.prepareStatement() 呼び出しでは、作成された PreparedStatement オブジェクトのインスタンスはキャッシュされずに、呼び出し元に戻されます。このプロパティは maxPreparedStatementCacheSize と同様に、データソースの reuseStatements プロパティが true（デフォルト）に設定されているときだけ有効です。	20

デバッグの出力

データソースでは、アプリケーションの処理時に、数多くのシステムプロパティをログアクティビティ、接続プール、接続レベル、文レベルに設定できます。以上のプロパティは通常のアプリケーションの実行時に設定することはありませんが、制御の JDBC フローの詳細が必要な場合、これらのオプションが役立ちます。JDBC データソースや接続で問題が発生した場合、Borland テクニカルサポートに以上のプロパティセットで生成される実行時出力を提供すると原因の究明に有効な場合があります。以上のプロパティをパーティションに設定すると、JDBC アクティビティの間にログメッセージが生成されます。メッセージを実際にパーティションログに書き込むには、`log4j` に設定を追加する必要があります。`logConfiguration.xml` という名前のパーティションの `log4j` 設定ファイルを探して次の <logger> 要素を追加します。

```
...
<log4j:configuration>
 ...
    <logger name="com.inprise.visitransact.jdbc2" additivity="true">
       <level value="DEBUG" />
    </logger>
    ...
</log4j:configuration>
```

メモ BAS ログは、Log4j インフラストラクチャに基づきます。Log4j を使用する一部のユーザーアプリケーションにより、パーティションが停止することがあります。ユーザーアプリケーションでは、アーカイブで設定ファイルをデプロイメントするのではなく、パーティション単位で log4j 設定ファイルを使用する必要があります。デフォルトでは、このファイルは、パーティションの管理オブジェクトのフットプリントである <install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/mos/<partition_name>/adm/properties/logConfiguration.xml です。または、<install_dir>/bin/partition.config ファイルの次の行のコメントを解除します。

vmprop borland.enterprise.server.partition.disableSystemRedirect=true

システムプロパティ名	型	説明	デフォルト値
DataSourceDebug	Boolean	すべてのデータソースに対して、データソースレベルでアクティビティをレポートします。	False
ConnectionPoolDebug	Boolean	すべてのデータソースに対して、接続プールレベルでアクティビティをレポートします。	False
ConnetionPoolStateDebug	Boolean	接続プールの接続の推移をレポートします。	False
JDBCProxyDebug	Boolean	すべての接続に対して、 接続レベルでアクティビティをレポートします。	False
PreparedStatementCacheDebug	Boolean	すべての文に対して、 準備文レベルでアクティビティをレポートします。	False

Borland AppServer のプールされた接続の状態について

EJB コンテナの静的収集オプションを有効にすると、パーティションイベントログには JDBC 接続プールに関する重要な統計値が収集されます。このログには、プール JDBC2 接続の各種ライフサイクル状態の接続数がリストで記録されます。次に各状態の説明を示します。

- **Free：**アプリケーションで利用できるキャッシュ／プールされた接続
- **TxBusy：**トランザクションで使用中のキャッシュ済み接続
- **NoTxBusy：**トランザクションコンテキストがないアプリケーションで使用中のキャッシュ済み接続
- **Committed：**トランザクションサービスからの `commit()` 呼び出しを受け取ったトランザクションに関連付けられた接続
- **RolledBack：**トランザクションサービスからの `rollback()` 呼び出しを受け取ったトランザクションに関連付けられた接続
- **Prepared：**トランザクションサービスからの `prepare()` 呼び出しを受け取ったトランザクションに関連付けられた接続
- **Forgot：**トランザクションサービスからの `forget()` 呼び出しを受け取ったトランザクションに関連付けられた接続
- **TxBusyXaStart：**トランザクションブランチに関連付けられたプールされた接続
- **TxBusyXaEnd：**トランザクションブランチとの関連付けが解消されたプールされた接続
- **BusyTimedOut：**`busyTimeout` プールプロパティの指定時間を過ぎてトランザクションに滞留したためプールから削除されたキャッシュ済み接続
- **IdleTimedOut：**プールの `idleTimeout` プロパティを過ぎてアイドルであったためプールから削除された接続
- **JdbcHalfCompleted：**接続がプール管理（更新など）に関係のあるバックエンドハウスキーピングアクティビティにかかわっているため、アクティビティが終了するまで利用できない遷移状態。
- **Closed：**基底の JDBC 接続が閉じられた
- **Discarded：**（タイムアウトエラーなどのため）削除されたキャッシュ済み接続
- **JdbcFinalized：**リファレンス先のない接続をゴミ箱に収集した

従来の JDBC 1.x ドライバのサポート

JDBC 1x ドライバはデータソースオブジェクトを提供しません。ただし、J2EE 仕様では、データベース接続は、常に `javax.sql.DataSource` インターフェースで取得されます。ユーザーが引き続き JDBC 1x ドライバを使用できるように、AppServer では、移植可能な J2EE コードを利用できる JDBC 1x データソースのインプリメンテーションを提供します。このインプリメンテーションは、JDBC 1x 仕様の DriverManager 接続メカニズムの前面で提供しています。

DDEditor で、このようなドライバの最上部にデータソースを定義する場合、［Datasource Type］フィールドで［Other(JDBC1x)］を選択します。［Main］パネルで、ドライバマネージャクラス名と、特定のデータベースとドライバの接続 URL を指定します。

クラス名 `com.inprise.visitransact.jdbc1w2.InpriseConnectionPoolDataSource` は、JDBC ドライバの `DriverManager` クラスではありません。これは、ラッパークラスです。ベンダーのクラスは、エディタパネルの［Driver class name］テキストボックスで指定します。

JDBC データソースの定義の応用

サーバーのグラフィカルツールを使用するしないにかかわらず、データソースの定義では、XML 形式でコンテナに何らかの情報を提供します。JDBC データソースの定義と、その定義を JNDI にバインドするときに何が行われるか調べてみましょう。まず、`jndi-definitions.xml` ファイルの DTD から調べることにします。太字で印刷されている要素は JDBC データソースのメイン要素です。

```
<!ELEMENT jndi-definitions (visitransact-datasource*, driver-datasource*, jndi-
object*)>
<!ELEMENT visitransact-datasource (jndi-name, driver-datasource-jndiname,
property*)>
<!ELEMENT driver-datasource (jndi-name, datasource-class-name,
log-writer?, property* )>
<!ELEMENT jndi-object (jndi-name, class-name, property* )>
<!ELEMENT property (prop-name, prop-type, prop-value)>
  <!ELEMENT prop-name (#PCDATA)>
  <!ELEMENT prop-type (#PCDATA)>
  <!ELEMENT prop-value (#PCDATA)>
  <!ELEMENT jndi-name (#PCDATA)>
  <!ELEMENT driver-datasource-jndiname (#PCDATA)>
  <!ELEMENT datasource-class-name (#PCDATA)>
  <!ELEMENT log-writer (#PCDATA)>
  <!ELEMENT class-name (#PCDATA)>
```

JDBC データソースの定義には、2 つの XML 要素が関係します。最初の要素が `<visitransact-datasource>` 要素です。ここには、アプリケーションコードが検索するデータソースを定義します。次のような情報を組み込みます。

- **jndi-name :** JNDI がデータソースの名前として参照します。エンタープライズ Bean のリソースリファレンスにもある名前です。
- **driver-datasource-jndiname :** ライブラリとしてパーティションにデプロイメントするデータベースや JMS ベンダーが提供するドライバクラスの JNDI 名です。次に紹介する `<driver-datasource>` 要素が参照する名前でもあります。
- **properties :** これらは接続プールのデータソースのロールのプロパティです。以上のプロパティについては、189 ページの「JDBC データソースの接続プールプロパティの定義」でさらに詳しく説明します。

ここでは、XML によるデータソース定義のこの部分のサンプルで検討しましょう。次のサンプルでは、Oracle を使用する例を紹介します。

```
<jndi-definitions>
        <visitransact-datasource>
            <jndi-name>datasources/Oracle</jndi-name>
            <driver-datasource-jndiname>datasources/OracleDriver
              </driver-datasource-jndiname>
            <property>
               <prop-name>connectionType</prop-name>
               <prop-type>Enumerated</prop-type>
               <prop-value>Direct</prop-value>
            </property>
            ...
            <!-- other properties as needed -->
            ...
        </visitransact-datasource>
        ...
    </jndi-definitions>
```

まだ完成ではありません。ドライバの情報をさらに指定する必要があります。それでデータソース定義の残り半分が完成です。それには、`<driver-datasource>` 要素に次の情報を指定します。

- **jndi-name :** ドライバクラスの JNDI 名と、その値は、`<visitransact-datasource>` 要素の `<driver-datasource-jndiname>` 値と同じ *であるものとします。*
- **datasource-class-name :** これは、リソースベンダーが提供する接続ファクトリクラスの名前です。パーティションにライブラリとしてデプロイメントするクラスと同じです。
- **log-writer :** 一部のベンダー接続ファクトリクラス向けの詳細モードを起動するための論理要素です。このプロパティの使用方法については、リソースのマニュアルを参照してください。

- **properties :** ユーザー名、パスワードなど、JDBC リソース固有のプロパティがあります。これらのプロパティは、ドライバクラスに渡されて処理されます。プロパティについては、JDBC リソースのマニュアルを参照してください。XML でプロパティを指定する方法については、次に説明します。

以上の説明を基に、先の Oracle データソースの定義を完成させましょう。完全を期するため、まず前述の XML を再び掲載します。

```
<jndi-definitions>
 <visitransact-datasource>
       <jndi-name>datasources/Oracle</jndi-name>
       <driver-datasource-jndiname>datasources/OracleDriver
          </driver-datasource-jndiname>
     <log-writer>False</log-writer>
     <property>
          <prop-name>connectionType</prop-name>
          <prop-type>Enumerated</prop-type>
          <prop-value>Direct</prop-value>
     </property>
 </visitransact-datasource>
    ...
```

ドライバデータソース JNDI 名が太字であることに注意してください。さらに次の要素を追加します。

```
   <driver-datasource>
 <jndi-name>datasources/OracleDriver</jndi-name>
 <datasource-class-name>oracle.jdbc.pool.OracleConnectionPoolDataSource</
datasource-class-name>
 <property>
   <prop-name>user</prop-name>
   <prop-type>String</prop-type>
   <prop-value>MisterKittles</prop-value>
 </property>
 <property>
   <prop-name>password</prop-name>
   <prop-type>String</prop-type>
   <prop-value>Mittens</prop-value>
 </property>

  ...

  // 必要に応じてほかのプロパティを追加
  ...

   </driver-datasource>
</jndi-definitions>
```

以上で JDBC データソースの定義が完成しました。DAR としてパッケージした XML ファイルは、パーティションにデプロイメントできます。これにより、データソースがネーミングサービスに登録され、検索の対象になります。

J2EE アプリケーションコンポーネントから JDBC リソースに接続

EJB コンポーネントの ejb-borland.xml などの Borland 独自のデプロイメントデスクリプタでは、データソースの論理名を JDBC データソース定義の実際の JNDI ロケーションにマッピングするために <resource-ref> 要素を使用します。論理名のロケーションへのマッピングは、アプリケーションコンポーネントで目的のデータソースに対する JNDI 検索が実行されるときに行われます。要素は、各ユーザーのコンポーネント定義内で使用します。たとえば、エンティティ Bean の <resource-ref> は、<entity> タグ内に存在する必要があります。Borland デプロイメントデスクリプタの <resource-ref> 要素の DTD 表現を調べてみます。

```
<!ELEMENT resource-ref (res-ref-name, jndi-name, cmp-resource?)>
```

この要素では、次のように指定します。

- **res-ref-name :** これは、リソースの論理名であり、標準 ejb-jar.xml デスクリプタファイルの <resource-ref> 要素で使用するものと同じ論理名です。これは、アプリケーションコードがデータソースの検索に使用する名前です。
- **jndi-name :** 論理名にバインドされるデータソースの JNDI 名です。DAR でデプロイメントする <visitransact-datasource> 要素の対応する <jndi-name> 要素の値と同じ値とします。
- **cmp-resource :** エンティティ Bean だけに関連するオプションの論理要素です。True に設定すると、コンテナの CMP エンジンがこのデータソースを監視します。

先に定義した Oracle データソースを使用するエンティティ Bean のサンプルで調べてみましょう。

```
<entity>
 <ejb-name>entity_bean</ejb-name>
 ...
    <resource-ref>
  <res-ref-name>jdbc/MyDataSource</res-ref-name>
  <jndi-name>datasources/Oracle</jndi-name>
  <cmp-resource>True</cmp-resource>
    </resource-ref>
 ...
</entity>
```

ご覧のように、データソース定義の <visitransact-datasource> 要素と同じ JNDI 名を使用しました。次に、データソースオブジェクトリファレンスの取得方法について調べてみましょう。このために、アプリケーションはデプロイメント済みコンポーネントの <res-ref-name> 値を検索します。オブジェクトリファレンスは、リモートの CosNaming プロバイダから取り出します。次に例を示します。

```
javax.sql.DataSource ds1;

try {
 javax.naming.Context ctx = (javax.naming.Context)
    new javax.naming.InitialContext();
 ds1 = (DataSource)ctx.lookup("java:comp/env/jdbc/MyDataSource");
}
catch (javax.naming.NamingException exp) {
 exp.printStackTrace();
}
```

以上で、データベース java.sql.Connection は、ds1 から取得できます。

第 23 章

JMS の使い方

JMS 接続ファクトリ、JMS キュー／トピックの送信先などのリソース関連オブジェクトは、移植可能な J2EE の規定の方法で JNDI を介して取得できます。JMS リソースオブジェクトは、アプリケーションコンポーネントのデプロイメントデスクリプタで定義された J2EE リソースリファレンスの JNDI 検索を実行することによって解決されます。リソースリファレンス定義には、標準 J2EE デプロイメントデスクリプタと Borland 独自のデプロイメントデスクリプタの両方を使用します。標準デプロイメントデスクリプタでは、リソースリファレンスはアプリケーションの JNDI 環境ネーミングコンテキストである `java:comp/env/` に基づいて論理名を指定します。Borland のデプロイメントデスクリプタは、リソースリファレンスの論理名を JMS リソース定義の実際の JNDI ロケーションに関連付けることによって標準デスクリプタを補足します。たとえば EJB JAR コンポーネントでは、標準 J2EE デプロイメントデスクリプタ `ejb-jar.xml` は JMS 接続ファクトリの `<resource-ref>` 要素と JMS トピック／キューの `<resource-env-ref>` 要素を使って EJB のリソースリファレンスを指定します。Borland AppServer（AppServer）におけるリソースリファレンスの JNDI 検索では、JMS リソース定義を取得し、その定義から目的の JMS オブジェクトを作成して検索の呼び出し元に返します。JMS リソース定義にあるプロパティ値が作成されるリソースオブジェクトのタイプと特性を決定します。

リソースリファレンスの検索を実行する前に、まず必要なリソース定義を物理的な JNDI ロケーションにバインドする必要があります。AppServer では、JMS リソース定義は定義アーカイブ（DAR）モジュールのデプロイメント時に JNDI サービスプロバイダにバインドされます。デフォルトでは、このオブジェクトは AppServer の JNDI CosNaming サービスプロバイダであるパーティションのネーミングサービスにバインドされます。この章では、DAR モジュールで JMS リソースオブジェクトを定義する方法について説明し、J2EE アプリケーションから JMS リソースオブジェクトのハンドルを取得する方法の詳細について説明します。JMS のアクティビティ、およびそれがトランザクションを関連付ける方法についても説明します。

DAR の中には、JNDI プロバイダ（ネーミングサービス）にバインドする各リソース関連オブジェクトのプロパティを含む JNDI 定義モジュール（`jndi-definitions.xml` ファイル）があります。アプリケーション EAR がデプロイメントされると、DAR ファイルの内容がパーティションのネーミングサービスにデプロイメントされます。`jndi-definitions.xml` ファイルに定義されたリソース関連オブジェクトのプロパティは、パーティションがホストするネーミングサービスで JNDI にバインドされたオブジェクトに格納されます。

アプリケーションクライアントまたは EJB コンポーネントは、リソース関連オブジェクトの JNDI 検索を行うとき、JNDI プロバイダと通信する `lookup()` メソッドを次のようにして呼び出します。

1 アプリケーションクライアントは、標準デプロイメントデスクリプタ（EJB の場合は `ejb-jar.xml`）内の `<resource-ref>` 要素を参照して、リソースの論理名を取得します（この検索はコンポーネントのローカル名前空間 `java:comp/env` 内で行われ、オブジェクトの論理名を取得します）。この論理名は、`<resource-ref-name>` サブ要素で指定されています。たとえば、`ejb-jar.xml` の場合は、次のようになります。

```
...
 <description> この例では、2 フェーズコミットトランザクションでの JMS XA と
JDBC XA を示します。</description>
 <enterprise-beans>
  <session>
   ...
     <resource-ref>
        <description />
        <res-ref-name>jms/insurance/ConnectionFactory</res-ref-name>
        <res-type>javax.jms.ConnectionFactory</res-type>
        <res-auth>Container</res-auth>
     </resource-ref>
     ...
  </session>
```

2 コンテナはこの論理名を使用して、Borland 独自のデプロイメントデスクリプタ `ejb-borland.xml` から、JMS リソース定義（JNDI にバインドされたオブジェクト）の実際の JNDI ロケーションを次のようにして取得します。

```
...
 <enterprise-beans>
  <session>
  ...
     <resource-ref>
        <res-ref-name>jms/insurance/ConnectionFactory</res-ref-name>
        <jndi-name>jms/xacf</jndi-name>
     </resource-ref>
  </session>
```

3 次にコンテナは、バインドされたオブジェクトに格納されているプロパティ値を使用して、リソースオブジェクトのインスタンスを作成します。関連する DAR がデプロイメントされたとき、ConnectionFactory オブジェクトには以下のプロパティが `jndi-definitions.xml` ファイルに基づいて格納されています。

```
...
<jndi-definitions>
  <jndi-object>
     <jndi-name>jms/xacf</jndi-name>
     <classname>com.tibco.tibjms.TibjmsXAConnectionFactory</class-name>
     <property>
        <prop-name>serverUrl</prop-name>
        <prop-type>String</prop-type>
        <prop-value>localhost:7222</prop-value>
     </property>
  </jndi-object>
```

4 このインスタンスは、コンテナによって Borland 独自の API（JMS プロキシレイヤ内）にラップされ、`lookup()` の呼び出し元（アプリケーションクライアントまたは他の J2EE コンポーネント）に戻されます。

JMS 1.1 – 共通 API

JMS 1.1 では、ドメインに依存しない統一的な API を JMS クライアントアプリケーションで使用することができます。クライアントは、汎用 JMS ConnectionFactory のハンドルを取得し、そこから取得した汎用セッションオブジェクトをキューやトピックで使用して、メッセージを処理できます。共通 API と、そのドメイン固有 API を次の表に示します。

JMS 共通 API（JMS 1.1）	ポイントツーポイントドメイン API	Publish/Subscribe ドメイン
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber
XAConnectionFactory	XAQueueConnectionFactory	XATopicConnectionFactory
XAConnection	XAQueueConnection	XATopicConnection
XASession	XAQueueSession	XATopicSession

共通インターフェースの使用方法における主な変更は、1 つ以上のキューやトピックの宛先に対して、同じセッションの同じトランザクションからすべて同時にアクセスできるようになったことです。この変更により、1 つのトランザクション内のすべてのメッセージ（キューおよびトピックとやり取りするメッセージ）が送信されてトランザクションが成功したとみなされるか、トランザクション全体が失敗して、どのメッセージもデプロイメントされないかのいずれかになります。

Borland AppServer は、ドメインに依存しない JMS 1.1 API をサポートし、また、すべての JMS 1.1 API を使用する際の J2EE 1.4 による制約にしたがいます。

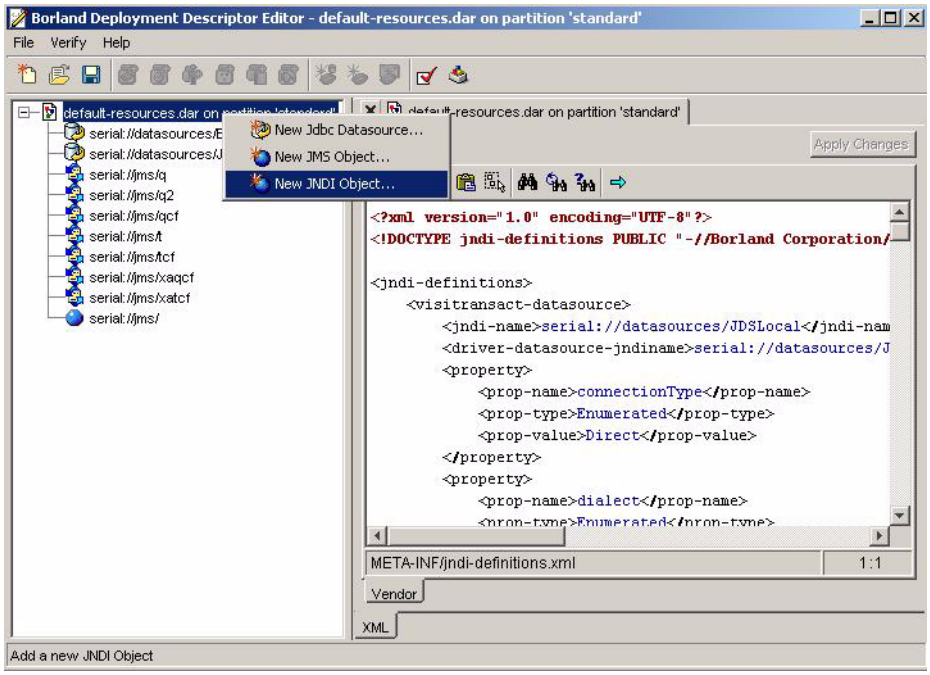
JMS 接続ファクトリと宛先の設定

管理コンソールを使用して、JMS リソースオブジェクトを設定するパーティションの［Deployed Modules］リストに移動します。デフォルトでは、すべてのパーティションに、`default-resources.dar` というデプロイメント済みの JNDI 定義モジュール（DAR）があります。そのモジュールを右クリックし、コンテキストメニューから［Edit deployment descriptor］を選択します。デプロイメントデスクリプタエディタ（DDEditor）が表示されます。

DDEditor のナビゲーションペインには、製品で設定済みの JMS 接続ファクトリとキュー／トピックのリストが表示されます。接続ファクトリ名をクリックします。右側のペインにプロパティが表示されます。接続ファクトリごとに、［Tibco］、［Sonic］、［WMQ］、または別の JMS プロバイダ（［Other］）を選択できます。DDEditor には、Tibco に関する情報が組み込まれており、各クラスのクラス名が自動的に代入されます。また、［JMS Object type］ドロップダウンリストからオブジェクトリソースタイプを選択することもできます。OpenJMS では `openjms.xml` ファイルを編集して、接続ファクトリと送信先を設定する必要があります。このファイルへのアクセス方法の詳細は、227 ページの「OpenJMS の JNDI オブジェクトの設定」を参照してください。

［JMS provider］リストで［Other］を選択した場合は、JMS ベンダーのマニュアルで、接続ファクトリ、トピック、キューの実装クラスの正しい名前を確認してください。また、［Main］パネルには、代入するプロパティが提示されないので、［Properties］タブを使って適切なプロパティを設定する必要があります。

新しい JMS オブジェクトを作成するには、ナビゲーションペインでルートノードを右クリックし、コンテキストメニューから［New JMS Object］を選択します。



作成する JMS オブジェクトの JNDI 名を指定するダイアログボックスが表示されます。デフォルトでは、指定する名前はネーミングサービス内の場所に対応します。JNDI 名に "serial//" プレフィクスを付けて指定する場合は、プレフィクスの後の残りの名前がネーミングサービス内の場所に対応します。JNDI 名を指定すると、この JMS オブジェクトの表示がナビゲーションペインのツリーに表示されます。表示をクリックして設定パネルを開きます。

DDEditor には、Tibco、Sonic、WMQ に関する情報が組み込まれており、クラス名が自動的に代入されます。

メモ ［Main］パネルには、Tibco 、Sonic、および WMQ 以外の JMS オブジェクトは示されません。適切なプロパティを設定するには、［Properties］タブを使用する必要があります。

操作が終了したら、［File ｜ Save As］を選択します。モジュールは元のパーティションに保存され、再デプロイメントされます。

JMS 接続ファクトリの接続プールプロパティの定義

AppServer に定義された各 JMS 接続ファクトリには、接続プールが関連付けられています。DAR モジュールの `jndi-definitions.xml` ファイルで定義する各 JMS 接続ファクトリには、接続プールプロパティを指定できます。AppServer パーティションシステムのプロパティは、パーティションの Java 仮想マシンで確立されるすべての JMS 接続プールのデフォルト動作を指定できます。ただし、`jndi-definitions.xml` ファイルの個々の JMS 接続ファクトリに対して定義されるプロパティは、システムプロパティ値を上書きします。

次の表に、JMS 接続プールのデフォルト設定として使用する AppServer パーティションシステムのプロパティをリストします。

プロパティ名	説明	型	このプロパティの有効値
JMSConnectionMaxPoolSize	JMS 接続プールで利用できる JMS 接続の最大数	Integer	0<n。ここで n は JMS 接続プールで利用できる接続の最大数。 デフォルトは 0 で、 接続数は無制限
JMSConnectionWaitTimeout	JMS 接続プールで接続が解放されるまで待つ時間	Integer	デフォルトは 30 秒
JMSConnectionIdleTimeout	JMS 接続プールで接続を破棄する前にアイドル状態で待つ時間	Integer	デフォルトは 60 秒
JMSConnectionPoolDebug	AppServer JMS 接続プーリングに関連するデバッグメッセージの表示をオンにする	boolean	デフォルト値は true true - デバッグをオン false - デバッグをオフ
JMSConnectionPoolDisable	JMS 接続ファクトリの AppServer 接続プーリングの使用を制御する	boolean	デフォルト値は false false - JMS 接続をオン true - JMS 接続をオフ
JMSConnectionPoolMonitorLevel	AppServer JMS 接続とセッションプール用の JMS プールの監視をオンにする。 **重要：** それぞれの JMS 接続には、JMS セッションプールで管理される JMS セッションのセットを作成できます。このプロパティ値を設定する場合は、 接続のパフォーマンスに対する影響を考慮してください。各レベルとともに、カウンタと状態の値を保守し収集する作業は増加するためです。 " 最大 " を選択すると、接続のパフォーマンスに対する影響は最大になります。	String	■ "none"（デフォルト） ■ "minimum" ■ "medium" ■ "maximum"
JMSSessionMaxPoolSize	接続ファクトリの各 JMS セッションプールの JMS セッションの最大数	Integer	0<n。ここで n は、 JMS 接続に関連する JMS セッションプールで利用できる JMS セッションの最大数。デフォルトは 0 で、 接続数は無制限
JMSSessionWaitTimeout	JMS セッションプールでセッションが解放されるまで待つ時間	Integer	デフォルトは 30 秒
JMSSessionPoolDisable	JMS 接続ごとの AppServer セッションプーリングの使用を制御する。このプロパティの値を true にして JNDI で JMS 接続ファクトリを検索すると、 ベンダー JMS 接続ファクトリが返される。AppServer プロキシクラスではラップされない。	boolean	デフォルト値は false false - JMS セッションをオン true - JMS セッションをオフ
JMSSessionPoolDebug	AppServer JMS セッションプーリングに関連するデバッグメッセージの表示をオンにする	boolean	デフォルト値は false false - デバッグをオフ true - デバッグをオン

個々の JMS 接続ファクトリのプロパティの定義

JMS プールプロパティは、jndi-definitions.xml ファイルの個々の接続ファクトリに対して定義できます。このプロパティはパーティションのシステムプロパティを上書きします。<property> 要素を使用して、プールプロパティを追加します。たとえば、次のようになります。

```
<jndi-definitions>
   <!-- ************************** -->
   <!-- * JMS Connection Factories * -->
   <!-- ************************** -->
       <jndi-object>
           <jndi-name>jms/cf</jndi-name>
           <class-name>com.tibco.tibjms.TibjmsConnectionFactory</class-name>
           <property>
               <prop-name>serverUrl</prop-name>
               <prop-type>String</prop-type>
               <prop-value>localhost:7222</prop-value>
           </property>
           <property>
               <prop-name>besConnectionPoolMaxPoolSize</prop-name>
               <prop-type>Integer</prop-type>
               <prop-value>11</prop-value>
           </property>
           <property>
               <prop-name>besConnectionPoolDebug</prop-name>
               <prop-type>Boolean</prop-type>
               <prop-value>true</prop-value>
           </property>
           <property>
               <prop-name>besSessionPoolDisable</prop-name>
               <prop-type>Boolean</prop-type>
               <prop-value>true</prop-value>
           </property>
       </jndi-object>
       ...
   </jndi-definitions>
```

次に、JMS 接続ファクトリプールのプロパティおよび上書きされる対応するシステムプロパティの完全なリストを示します。

個々のプールプロパティ	対応するシステムプロパティ
besConnectionPoolMaxPoolSize	JMSConnectionMaxPoolSize
besConnectionPoolWaitTimeout	JMSConnectionWaitTimeout
besConnectionPoolIdleTimeout	JMSConnectionIdleTimeout
besConnectionPoolMonitorLevel	JMSConnectionPoolMonitorLevel
besConnectionPoolDisable	JMSConnectionPoolDisable
besConnectionPoolDebug	JMSConnectionPoolDebug
besSessionPoolMaxPoolSize	JMSSessionMaxPoolSize
besSessionPoolWaitTimeout	JMSSessionWaitTimeout
besSessionPoolDisable	JMSSessionPoolDisable
besSessionPoolDebug	JMSSessionPoolDebug

J2EE アプリケーションコンポーネントにおける JMS 接続ファクトリと送信先の取得

JMS 接続ファクトリオブジェクトは、JDBC データソースオブジェクトと同じ方法で取得できます。ファクトリオブジェクトは、標準 J2EE と Borland 固有のデプロイメントデスクリプタの両方の <resource-ref> 要素の中で宣言されます。ただし、アプリケーションが JMS プロバイダの送信先と対話する必要がある場合は特別な設定が必要です。<resource-env-ref> 要素は、両方のデスクリプタで指定し、JMS 送信先を少なくとも 1 つ定義する必要があります。JMS 送信先は、メッセージを生成／使用するためのキューまたはトピックです。標準 J2EE デプロイメントデスクリプタは JMS 接続ファクトリと送信先の論理名とタイプを提供しますが、Borland 固有のデプロイメントデスクリプタは論理名を JNDI 検索を介して解決される実際のターゲットオブジェクトのリファレンスにマップします。

J2EE 1.2 および J2EE 1.3

標準 J2EE デプロイメントデスクリプタの <resource-ref> 要素と <resource-env-ref> 要素の詳細は、J2EE 1.3 仕様で説明されています。この 2 つの要素は、EJB、サーブレット、アプリケーションクライアントなどの JMS API を使用するすべてのアプリケーションコンポーネントに適用されます。同様に、対応する <resource-ref> 要素と <resource-env-ref> 要素は、対応する Borland 固有のデプロイメントデスクリプタに存在します。JMS を使用する EJB セッション Bean のデプロイメントデスクリプタを調べてみます。最初に、標準 EJB デスクリプタ ejb-jar.xml を次に示します。

```
...
<session>
 <ejb-name>session_bean</ejb-name>
 ...
 <resource-ref>
     <res-ref-name>jms/MyJMSQueueConnectionFactory</res-ref-name>
  <res-type>javax.jms.QueueConnectionFactory</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
 </resource-ref>
 <resource-env-ref>
  <res-env-ref-name>jms/MyJMSQueue</res-env-ref-name>
  <res-env-ref-type>javax.jms.Queue</res-env-ref-type>
 </resource-env-ref>
 ...
</session>
```

上に示した移植可能なデスクリプタは、それぞれ <resource-ref> と <resource-env-ref> を介して、JMS 接続ファクトリと JMS キューの論理名を定義します。EJB コンポーネントの ejb-borland.xml などの Borland 固有のデプロイメントデスクリプタでは、アプリケーションコンポーネントで目的の接続ファクトリに JNDI 検索を実行するときに、論理名を JMS 接続ファクトリ定義の実際の JNDI ロケーションに解決するために <resource-ref> 要素を使用します。この要素は、個々のコンポーネントのデスクリプタ定義の内部で使用されます。たとえば、エンティティ Bean の <resource-ref> は、<entity> タグ内に存在する必要があります。J2EE 1.2 および 1.3 Borland デプロイメントデスクリプタの <resource-ref> 要素の DTD 表現を調べてみます。

```
<!ELEMENT resource-ref (res-ref-name, jndi-name, cmp-resource?)>
```

この要素では、次のように指定します。

- **res-ref-name**：これは、リソースオブジェクトの論理名であり、標準 ejb-jar.xml デスクリプタファイルの <resource-ref> 要素で使用するものと同じ論理名です。これは、アプリケーションコンポーネントが JMS 接続ファクトリの検索に使用する名前です。
- **jndi-name**：論理名にバインドされる接続ファクトリの JNDI 名です。接続ファクトリが定義されているデプロイメント DAR の対応する <jndi-object> 要素の <jndi-name> 要素の値に一致する必要があります。

<resource-ref> 要素が JMS 接続ファクトリの論理名を目的の接続ファクトリ定義の実際の JNDI ロケーションにマップするために使用されるように、<resource-env-ref> 要素はキュー、トピックなどの JMS 送信先の論理名を送信先の定義の実際の JNDI ロケーションにマップします。Borland デプロイメントデスクリプタにおけるこの要素の DTD 表現は次のとおりです。

```
<!ELEMENT resource-env-ref (resource-env-ref-name, jndi-name)>
```

次の 2 つの要素を指定します。

- **resource-env-ref-name**：トピックまたはキューの論理名であり、その値は J2EE 標準デスクリプタの <res-env-ref-name> の値と同じものです。
- **jndi-name**：論理名を解決するトピックまたはキューの JNDI 名です。

上で定義されている ejb-jar.xml に対応する Borland デスクリプタ ejb-borland.xml の最終的な内容は次のとおりです。

```
<session>
 <ejb-name>session_bean</ejb-name>
 ...
 <resource-ref>
  <res-ref-name>jms/MyJMSQueueConnectionFactory</res-ref-name>
  <jndi-name>resources/qcf</jndi-name>
 </resource-ref>
 <resource-env-ref>
  <resource-env-ref-name>jms/MyJMSQueue</resource-env-ref-name>
  <jndi-name>resources/q</jndi-name>
 </resource-env-ref>
 ...
</session>
```

<resource-ref> 要素と <resource-env-ref> 要素は、JMS 関連リソースオブジェクトを必要とするすべての J2EE コンポーネントに使用できます。たとえば、JMS API を使用するアプリケーションクライアントは、クライアントデプロイメントデスクリプタの <resource-ref> 要素と <resource-env-ref> 要素のアプリケーションコードと仕様において、JNDI 検索またはリソースリファレンスを介して、EJB と同じ方法で接続ファクトリと送信先を取得する必要があります。たとえば、J2EE 標準デスクリプタの application-client.xml は次のとおりです。

```
<application-client>
...
<resource-ref>
 <res-ref-name>jms/MyJMSTopicConnectionFactory</res-ref-name>
 <res-type>javax.jms.TopicConnectionFactory</res-type>
 <res-auth>Application</res-auth>
</resource-ref>
<resource-env-ref>
 <res-env-ref-name>jms/MyJMSTopic</res-env-ref-name>
 <res-env-ref-type>javax.jms.Topic</res-env-ref-type>
</resource-env-ref>
...
 </application-client>
```

対応する Borland デスクリプタ application-client-borland.xml は次のとおりです。

```
<application-client>
...
<resource-ref>
 <res-ref-name>jms/MyJMSTopicConnectionFactory</res-ref-name>
 <jndi-name>resources/tcf</jndi-name>
</resource-ref>
<resource-env-ref>
 <resource-env-ref-name>jms/MyJMSTopic</resource-env-ref-name>
 <jndi-name>resources/t</jndi-name>
</resource-env-ref>
...
 </application-client>
```

次に、アプリケーションロジックで JMS 接続ファクトリと送信先へのオブジェクトリファレンスを取得する方法を示します。接続ファクトリの取得では、アプリケーションは J2EE デプロイメントデスクリプタの <resource-ref> 要素から <res-ref-name> 値の JNDI 検索を実行します。送信先オブジェクトを取得するには、J2EE デプロイメントデスクリプタの <resource-env-ref> 要素の <res-env-ref-name> 値に対して JNDI 検索を実行します。<jndi-name> に指定された名前は、デプロイメント DAR モジュールの JMS リソース定義の <jndi-object> 要素の JNDI 名と同じです。検索が成功すると JMS リソースオブジェクトが取得されます。つまり、論理名 jms/MyJMSTopicConnectionFactory によって識別される JMS 接続ファクトリに対して、resources/tcf のネーミングサービスからデプロイメントされた JMS 定義オブジェクトが取得され、それによって接続ファクトリオブジェクトが作成されてアプリケーションに返されます。

たとえば、前述のクライアントデスクリプタに対応するアプリケーションクライアントコードは、次のようにして JMS リソースオブジェクトを解決します。

```
javax.jms.TopicConnectionFactory myTCF;
javax.jms.Topic myTopic;
try {
 javax.naming.Context ctx = (javax.naming.Context) new
javax.naming.InitialContext();
  myTCF = (TopicConnectionFactory) ctx.lookup("java:comp/env/jms/
MyJMSTopicConnectionFactory");
  // これで myTCF から接続を取得できる
  myTopic = (Topic) ctx.lookup("java:comp/env/jms/MyJMSTopic");
  ...
}
catch (javax.naming.NamingException exp) {
 exp.printStackTrace();
}
```

J2EE 1.4

J2EE の以前のバージョンでは、各アプリケーションコンポーネントが標準デプロイメントデスクリプタ内で <resource-env-ref> を宣言して、自身のローカル名前空間から JMS 宛先を検索する必要がありました。別のアプリケーションコンポーネントが同じ宛先を参照している場合、それらの <resource-env-ref> を同じ宛先にバインドする必要があることを開発時に認識する手段はありませんでした。

次に、<resource-env-ref> を使用して 2 つの異なるアプリケーションコンポーネント（この場合はセッション Bean）で同じ JMS 宛先を定義する例を示します。

```
...
<ejb-jar ... >
   <enterprise-beans>
         <session>
            <ejb-name>SenderEJB</ejb-name>
            ...
            <resource-ref>
               <res-ref-name>jms/ConnectionFactory</res-ref-name>
               <res-type>javax.jms.ConnectionFactory</res-type>
               <res-auth>Container</res-auth>
            </resource-ref>
            <resource-env-ref>
               <resource-env-ref-name>jms/LogicalNameA</resource-env-ref-name>
               <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
            </resource-env-ref>
         </session>
         <session>
            <ejb-name>ReceiverEJB</ejb-name>
            ...
            <resource-ref>
               <res-ref-name>jms/ConnectionFactory</res-ref-name>
               <res-type>javax.jms.ConnectionFactory</res-type>
               <res-auth>Container</res-auth>
            </resource-ref>
            <resource-env-ref>
               <resource-env-ref-name>jms/LogicalNameB</resource-env-ref-name>
               <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
            </resource-env-ref>
         </session>
```

J2EE 1.4 では、<resource-env-ref> も引き続き使用できますが、JMS 宛先を指定するために新しい要素 <message-destination-ref> が導入されています。<resource-env-ref> 要素のかわりに <message-destination-ref> 要素を使用して、上記と同じ例を標準デプロイメントデスクリプタ ejb-jar.xml 内で作成し直した例を次に示します。

```
...
<ejb-jar ... >
   <enterprise-beans>
      <session>
         <ejb-name>SenderEJB</ejb-name>
         ...
         <resource-ref>
            <res-ref-name>jms/ConnectionFactory</res-ref-name>
            <res-type>javax.jms.ConnectionFactory</res-type>
            <res-auth>Container</res-auth>
         </resource-ref>
         <message-destination-ref>
            <message-destination-ref-name>jms/LogicalNameA
              </message-destination-ref-name>
           <message-destination-type>javax.jms.Queue
             </message-destination-type>
          <message-destination-usage>Produces</message-destination-usage>
           <message-destination-link>MsgQueue1</message-destination-link>
         </message-destination-ref>
      </session>
      <session>
         <ejb-name>ReceiverEJB</ejb-name>
         ...
         <resource-ref>
            <res-ref-name>jms/ConnectionFactory</res-ref-name>
            <res-type>javax.jms.ConnectionFactory</res-type>
            <res-auth>Container</res-auth>
         </resource-ref>
         <message-destination-ref>
            <message-destination-ref-name>jms/LogicalNameB
              </message-destination-ref-name>
           <message-destination-type>javax.jms.Queue
             </message-destination-type>
            <message-destination-usage>Consumes</message-destination-usage>
           <message-destination-link>MsgQueue1</message-destination-link>
         </message-destination-ref>
      </session>
   </enterprise-beans>
   <assembly-descriptor>
   <message-destination>
         <message-destination-name>MsgQueue1</message-destination-name>
      </message-destination>
   </assembly-descriptor>
</ejb-jar>
```

この例では、2 つの <message-destination-ref> 要素が Queue MsgQueue1 という同じ宛先を参照しています。これらの <message-destination-ref> には、どちらも値 MsgQueue1 を持つ <message-destination-link> 要素があります。このリンク要素により、<message-destination-ref> が <assembly-descriptor> 内の <message-destination> 要素にマップされ、2 つの <message-destination-ref> が同じキューに解決されます。ejb-borland には、<assembly-descriptor> 要素内に、対応する <message-destination> 要素があります。実行時には、ejb-jar.xml 内の <message-destination> 要素は、ejb-borland.xml デスクリプタで指定された <message-destination> 要素の <jndi-name> に、次のように解決されます。

```
...
<ejb-jar ... >
   <enterprise-beans>
      ...
   </enterprise-beans>
   <assembly-descriptor>
   <message-destination>
         <message-destination-name>MsgQueue1</message-destination-name>
         <jndi-name>jms/TibcoQueue1</jndi-name>
      </message-destination>
   </assembly-descriptor>
</ejb-jar>
```

複数の <message-destination-ref> が同じ基底宛先に解決されるアプリケーションで JMS メッセージフローを示すには、それぞれが <assembly-descriptor> 内の <message-destination> 要素に対応する値で <message-destination-link> を宣言する必要があります。<message-destination-link> 内の値は、<message-destination> 要素の <message-destination-name> の値と一致する必要があります。各 <message-destination-ref> は、実行時に同じ宛先に解決されます。

同じアプリケーション内の別の J2EE モジュールに定義されている <message-destination> にもリンクできます。たとえば、<message-destination-link> ../other/other.jar#destination </message-destination-link> は、相対パス ../other/other.jar にある JAR ファイル内の destination という名前の <message-destination> にリンクします。

また、使用するすべての宛先に関して、ejb-jar.xml に <message-destination> 要素を指定する必要があります。

重要 Borland デプロイメントデスクリプタにある <message-destination> の JNDI 名は、<message-destination> 要素にリンクされている <message-destination-ref> の指定された JNDI 名よりも優先されます。

JMS とトランザクション

トランザクションを含む EJB Bean コードで JMS API を使用するための規則については、EJB 2.0 仕様、セクション 17.3.5 で説明されています。

以下にその摘要を示します。

> 17.3.5 トランザクションにおける JMS API の使用
>
> Bean プロバイダは、単一のトランザクション内で、JMS 要求／応答パラダイム（JMS メッセージの送信に続けて、そのメッセージに対する同期応答を受信すること）を使用してはならない。
>
> JMS メッセージはトランザクションがコミットされるまで最終送信先に配信されないので、同じトランザクションで応答を受信することはありません。コンテナが Bean のかわりに JMS セッションのトランザクションエンリストを管理するので、createSession(boolean transacted,int acknowledgeMode),createQueueSession(boolean transacted,int acknowledgeMode) メソッドと createTopicSession(boolean transacted, int acknowledgeMode) メソッドのパラメータは無視されます。セッションが処理されたことは Bean プロバイダが指定しますが、肯定応答モードの値としては 0 を返すことをお勧めします。
>
> Bean プロバイダは、トランザクション内または未指定のトランザクションコンテキスト内のいずれでも JMS acknowledge() メソッドを使用しないようにします。未指定トランザクションコンテキスト内のメッセージ肯定応答は、コンテナで処理される。セクション 17.6.5 には、未指定トランザクションコンテキストによるメソッド呼び出しのインプリメンテーションにコンテナを使用できる技法に関する説明があります。

JMS 要求／応答パラダイムと JMS acknowledge() メソッドを使用しないことは、EJB Bean コードと同様にアプリケーションクライアントなどのその他の J2EE コンポーネントでも同じです。上に示した規則に加えて、アプリケーションコードでは JMS XA API は使用しないようにします。プログラムは、非トランザクション JMS プログラムのコードとまったく同様に記述する必要があります。グローバルトランザクションがアクティブの場合、必要な XA ハンドシェークはコンテナが処理します。唯一必要な設定は、JMS 接続ファクトリの JNDI オブジェクトを参照するデプロイメントデスクリプタ要素 <resource-ref> が XA バリアントを使用するように設定することです。宛先が XA でない場合は、プログラムは実行されますが、原子性は保証されません。つまり、ローカルトランザクションになります。また、AppServer でトランザクションハンドシェークを自動的に処理するには、アプリケーションの実行場所をコンテナ、EJB、Web、または AppClient にする必要があります。たとえば、JMS XA API 呼び出しを持たない Java クライアントは、JMS アクティビティをグローバルトランザクションに参加させないので、J2EE アプリケーションクライアントとして記述する必要があります。また、すべての接続ファクトリはデプロイメントデスクリプタ要素 <resource-ref> を介して検索する必要があります。これにより、コンテナで JMS API 呼び出しをトラップし、適切なフックを挿入することができます。

EJB 2.1 仕様から抜粋された次のような文を詳しく調べてみます。

> コンテナが Bean のかわりに JMS セッションのトランザクションエンリストを管理するので、createSession(boolean transacted,int acknowledgeMode),createQueueSession(boolean transacted,int acknowledgeMode) メソッドと createTopicSession(boolean transacted, int acknowledgeMode) メソッドのパラメータは無視されます。セッションが処理されたことは Bean プロバイダが指定しますが、肯定応答モードの値としては 0 を返すことをお勧めします。

ここでは、グローバルトランザクションがアクティブの場合、JMS セッションが生成／使用するメッセージは、グローバルトランザクションが管理する作業単位に含めることを前提にします。トランザクションエンリストを生成するために、createSession()、createQueueSession() または createTopicSession() を呼び出す接続の親接続ファクトリは、それぞれ javax.jms.XAConnectionFactory、javax.jms.XAQueueConnectionFactory または javax.jms.XATopicConnectionFactory として定義する必要があります。つまり、J2EE コンポーネントのために使用する JMS 接続ファクトリの定義を含む J2EE デプロイメントデスクリプタ要素 <resource-ref> の <res-type> の値は、javax.jms.XAConnectionFactory、javax.jms.XAQueueConnectionFactory、または javax.jms.XATopicConnectionFactory である必要があります。接続ファクトリに非 XA 接続ファクトリ <res-type> がある場合にもプログラムは動作しますが、JMS セッションで実行される作業はグローバルトランザクションに含まれません。その場合、transacted パラメータと acknowledgeMode パラメータはメッセージの生成／使用の動作に影響します。次にこの例を示します。

```
import javax.jms.*;

QueueConnectionFactory nonXAQCF;
Queue                  myQueue;

try {
 javax.naming.Context ctx = (javax.naming.Context) new
javax.naming.InitialContext();
  nonXAQCF = (QueueConnectionFactory) ctx.lookup("java:comp/env/jms/
MyJMSQueueConnectionFactory");
  myQueue = (Queue) ctx.lookup("java:comp/env/jms/MyJMSQueue");
 }
 catch (javax.naming.NamingException exp) {
  exp.printStackTrace();
 }

 // メモ：現在、グローバルトランザクションコンテキストは、セッションの作成時にアクティブになります。

 QueueSession qSession = conn.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
 QueueSender = qSession.createSender(myQueue);
 TextMessage msg = qSession.createTextMessage("A Message ");
 sender.send(msg);
```

ここで、TextMessage msg は、アクティブなグローバルトランザクションの結果にかかわらず、キューに入れられます。これは J2EE Compatibility Test Suite のテスト事例と同様です。この機能は、グローバルトランザクションで便利です。これにより、含めるグローバルトランザクションの完了結果にかかわらず、ログメッセージを送信する必要があります。

AppServer では、1 つのグローバルトランザクション内の複数のリソースアクセスがサポートされます。これにより、他の種類のリソースマネージャアクセスとともに JMS メッセージを送受信する作業単位を実行できます。つまり、たとえば EJB では、データベースなどの非 JMS リソースに対して作業を行うコードを記述したり、実行されるすべての作業のトランザクションを完了するためのコンテナを含むメッセージをキューに送信することが望まれます。トランザクションの完了時には、データベースに実行された作業がコミットされ、メッセージがキューに入れられるか、またはトランザクションの処理中に失敗が発生した場合は、データベース作業がロールバックされ、メッセージはキューに配信されません。

アプリケーションコードでは、次の `doSomeWork()` などの EJB メソッドは AppServer でサポートされます。

```
// セッション Bean のビジネスメソッド、EJB コンテナがトランザクションをマーク
     void doSomeWork()
     {
       // データベース接続の確立
          java.sql.Connection dbConn = datasource.getConnection();

        // SQL の実行
          ...

       // 同じトランザクション内でリモート EJB を呼び出す
          ejbRemote.doWork();

       // JMS メッセージをキューに送信
          jmsSender.send(msg);

      }
```

JMS サービスのセキュリティの有効化

セキュリティなどの JMS サービスに関するベンダー固有の情報については、221 ページの「JMS プロバイダの接続性」を参照してください。

JMS 接続ファクトリと送信先を設定するための高度な概念

JDBC データソースリソースオブジェクトと、JMS プロバイダである Tibco、SonicMQ、および WMQ 用の JMS 接続ファクトリおよび接続先は、`jndi-definitions.xml` デスクリプタを使用して DAR モジュールで定義されます。サンプル BAS 構成 j2eeSample の Welcome パーティションにデプロイメントされるモジュール `tibco-resources.dar` には、Tibco JMS サーバーオプションにより、AppServer インストール用に定義されたデフォルトの Tibco 接続ファクトリ、トピック、およびキューが含まれます。DDEditor を使用すれば、これら既存の定義を環境に応じて編集し、新しい定義を作成できます。JDBC データソースと同様に、JMS 接続ファクトリは JMS ベンダーが提供する接続ファクトリクラスをラップするクラスです。バンドルされていない JMS ベンダー、または AppServer で機能することが確認されていない JMS ベンダーを使用する場合は、そのベンダーの接続ファクトリクラスをパーティションにデプロイメントする必要があります。

JMS キューに関するベンダー固有の情報については、221 ページの「JMS プロバイダの接続性」を参照してください。

第 24 章

Hibernate を使用する

このセクションには、Hibernate と Borland AppServer の統合に関する情報、Hibernate アプリケーションの作成方法、Hibernate アプリケーションのパッケージ方法、Hibernate サービスの有効化、無効化方法、Hibernate 統計ログの有効化、無効化方法があります。

概要

Hibernate は、強力で、高性能の、オブジェクト / リレーショナル永続性およびクエリー サービスです。hibernate アプリケーションは、標準 J2EE アプリケーションで、Object Relational Mapping（ORM）と永続性用の Hibernate エントリを使用します。Hibernate の詳細については、www.hibernate.org を参照してください。

Borland AppServer は、Hibernate と緊密に連携しているので、hibernate アプリケーションの開発とデプロイメントが簡単にできます。

Borland AppServer の Hibernate サポートには次の機能があります。

- Hibernate と Borland AppServer を統合すると、Borland AppServer の基盤機能（トランザクション、ネーミング サービス、接続プール、コンテナ管理データ ソースなど）を活用する Hibernate エンティティを容易に使用できます。
- Hibernate アプリケーションに Hibernate ライブラリをパッケージする必要はもうありません。Hibernate 3.1.3 ライブラリは、パーティションのクラス パスで利用できます。
- デプロイメント中に、Borland AppServer は hibernate アプリケーションを検出し、自動的に hibernate の成果物（セッション ファクトリ）を作成し、JNDI に公開します。
- Borland AppServer により Hibernate アプリケーションを JTA と CMT を使用して AppServer データ ソースとトランザクション マネージャ（JTS と OTS）とともに動作するようにできます。
- Borland AppServer でデプロイメントされたアプリケーション内のすべての機能と Hibernate の ORM 機能を使用できます。
- Borland AppServer では柔軟に既存の CMP 2.0 エンティティ BEAN を Hibernate エンティティとともに使用できます。
- ユーザーのモデル（永続化層）の開発がさらに簡単になります。永続性エンティティが POJO で、エンティティ Bean のようなインターフェースを実装する必要がないからです。

Hibernate アプリケーションを作成する

データ ソース接続を作成する

Hibernate のセッション ファクトリでは、接続データ ソースを Hibernate 設定ファイルに指定することが必要です。DAR（JNDI 定義モジュール）や `jndi-definitions.xml` ファイルを使用して Borland AppServer のデータ ソースを作成できます。`default-resources.dar` ファイルには複数のデフォルト データ ソースが含まれます。

特定のデータベースを使用する場合は、対象データベース用のデータ ソースが作成されていることを確認し、hibernate のセッション ファクトリ設定ファイル（`hibernate.cfg.xml`）の `connection.datasource` プロパティにデータ ソースの `jndi-name` を使用する必要があります。

例

```
<visitransact-datasource>
  <jndi-name>datasources/JDSLocal</jndi-name> …
```

`connection.datasource` プロパティには `hibernate.cfg.xml` の `datasource/JDSLocal` を使用します。

設定ファイルとセッション ファクトリの使用

Borland AppServer は hibernate アプリケーションを検出し、自動的にセッション ファクトリを作成し、JNDI に公開します。

Hibernate 設定ファイルの使用

メモ このセクションでは、Borland AppServer で使用するために Hibernate のセッション ファクトリを設定する方法を説明します。Hibernate のセッション ファクトリの設定の詳細については、www.hibernate.org にある Hibernate のドキュメントを参照してください。

hibernate の設定ファイル（`hibernate.cfg.xml`）を使用してセッション ファクトリを設定できます。hibernate 設定ファイルで `session-factory` 要素の `name` 属性に JNDI 名を指定する必要があります。

```
<session-factory name="hibernate/SF">
...
```

hibernate 設定ファイルを設定し、パッケージすると、Borland AppServer は自動的にセッション ファクトリを作成し、JNDI に公開します。

Hibernate アプリケーションで、次のコードを使用して JNDI からセッション ファクトリを取得します。

```
Context ctx = new InitialContext();
ctx.lookup( “java:comp/env/hibernate/SF” );
```

メモ このコードを使用すると、セッション ファクトリを手動で作成する必要がありません。

データ ソースを設定する

「データ ソース接続を作成する」セクションで作成した同じ接続のデータ ソースを使用する必要があります。データ ソースを指定するために、次のプロパティを使用します。

```
<property name="connection.datasource">datasources/JDSLocal</property>
```

トランザクションを設定する

次のプロパティで、アプリケーション サーバーのトランザクション サポートを設定できます。

- `transaction.factory_class`

  このクラスを設定するには、次のコードを使用します。

```
<property name="transaction.factory_class">
  org.hibernate.transaction.JTATransactionFactory
</property>
```

メモ　JTA を直接使用する（BMT の）場合、`org.hibernate.transaction.JTATransactionFactory` を使用する必要があります。CMT セッション Bean では、`org.hibernate.transaction.CMTTransactionFactory` を使用する必要があります。

- `transaction.manager_lookup_class`

  このクラスを設定するには、次のコードを使用します。

```
<property name="transaction.manager_lookup_class">
   org.hibernate.transaction.BESTransactionManagerLookup
</property>
```

自動セッション フラッシュのプロパティを設定する

セッションに対して JTA トランザクション境界（BMT または CMT）を使用している場合は、自動セッション フラッシュ プロパティを設定して、トランザクションが完了する前にセッションをフラッシュする必要があります。

```
<property name="hibernate.transaction.flush_before_completion">true</property>
```

自動セッション クローズのプロパティを設定する

コンテナ管理トランザクションを使用している場合、トランザクションが完了した後、セッションをクローズするとき、次のプロパティを設定する必要があります。

```
<property name="hibernate.transaction.auto_close_session">true</property>
```

統計情報の収集用プロパティを設定する

Hibernate サービスは、各セッション ファクトリの統計情報を表示します。Hibernate で統計情報の収集を有効にするには、セッション ファクトリで次のプロパティを設定する必要があります。

```
<property name="hibernate.generate_statistics">true</property>
```

Hibernate の統計情報の詳細については、「Hibernate 統計を使用する」を参照してください。

エンティティ マッピングを設定する

セッション ファクトリのすべてのエンティティ マッピング ファイルを含めることができます。それには、次のプロパティを設定する必要があります。

```
<mapping resource="com/borland/examples/ejb/insurance/Claim.hbm.xml"/>
```

メモ　XML マッピング ファイルのパスは、Hibernate 設定ファイルへの相対パスです。

ダイアレクトを設定する

dialect プロパティを使用して、ダイアレクトを指定できます。たとえば、JdataStore データ ソースを使用する場合は、次のようにダイアレクトを指定できます。

```
<property name="dialect">
  org.hibernate.dialect.JdataStoreDialect
</property>
```

サポートしているほかのダイアレクトについては、www.hibernate.org の Hibernate ドキュメントを参照してください。

データベース スキーマを Drop と Recreate するためのプロパティを設定する

起動時にデータベース スキーマを drop および recreate するためには、hbm2ddl.auto プロパティを create に設定します。

```
<property name="hbm2ddl.auto">create</property>
```

実行済み SQL のログのプロパティを設定する

すべての実行済み SQL を標準出力（stdout）に表示するには、show_sql プロパティを true に設定する必要があります。

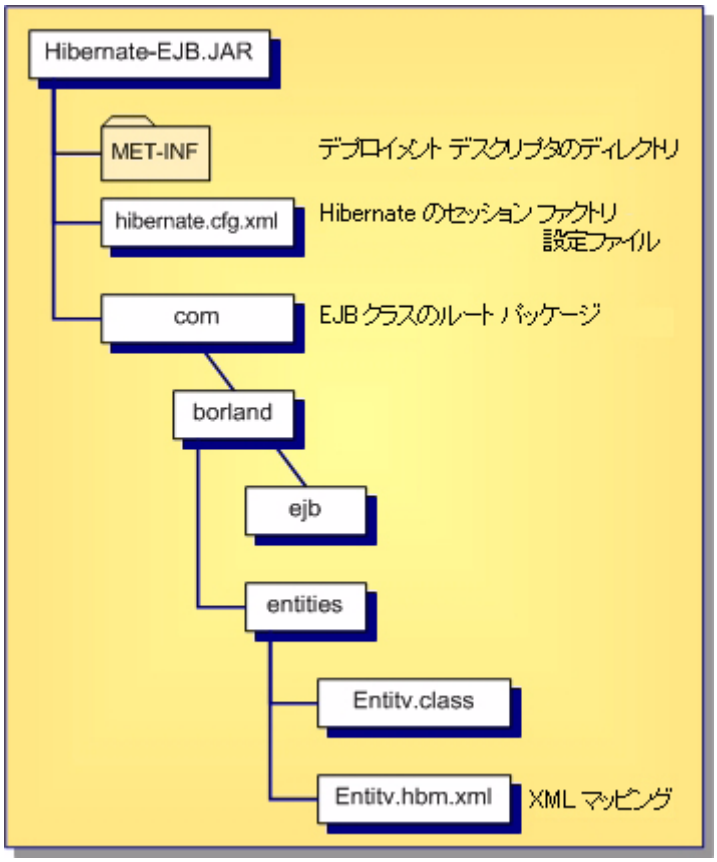
```
<property name="show_sql">true</property>
```

パッケージング

EJB モジュール：EJB-JAR

EJB-JAR の場合、`Archive` ルートに `hibernate.cfg.xml` ファイルを含める必要があります。

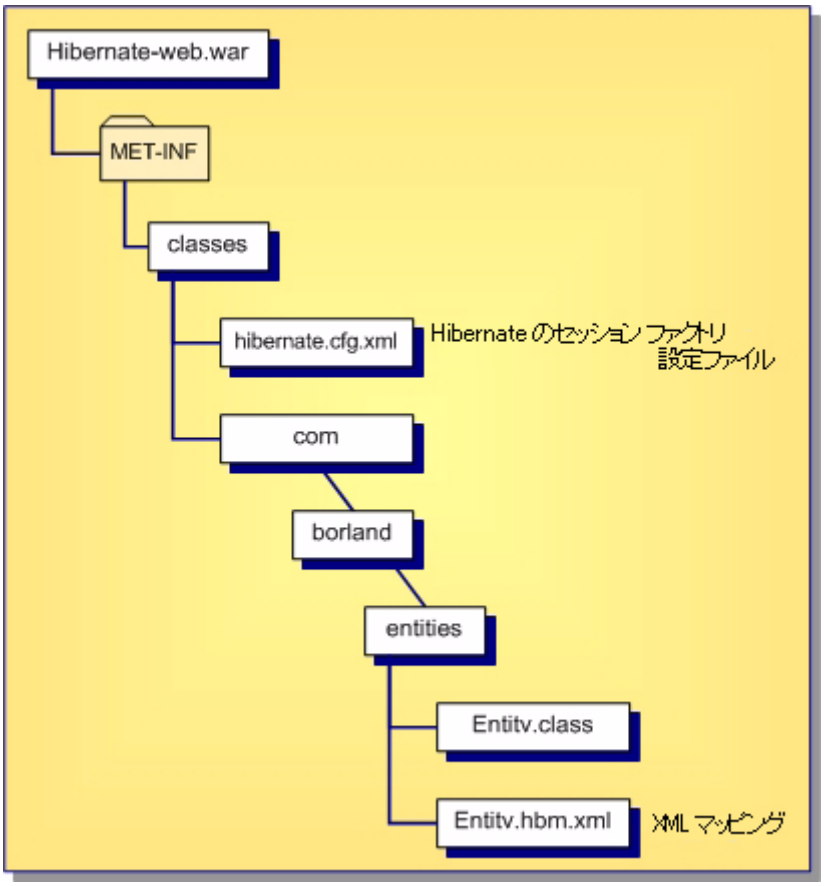
たとえば、EJB-JAR ファイルが `Hibernate-EJB.jar` の場合は、パッケージは次の図のようになります。



WEB モジュール：WAR

WAR の場合、`WEB-INF/classes` フォルダに `hibernate.cfg.xml` ファイルを含める必要があります。

たとえば、WAR ファイルが `Hibernate-web.war` の場合は、パッケージは次の図のようになります。



XML マッピングのパッケージング

XML マッピング ファイルとエンティティを同じ場所に配置することをお勧めします。さらに XML マッピング ファイルに `<Entity-Class-Name>.hbm.xml` という名前を付けることもお勧めします。「EJB モジュール：EJB-JAR」と 「WEB モジュール：WAR」セクションの図は、XML マッピング ファイルの配置場所を示します。`hibernate.cfg.xml` で設定ファイルの場所の相対パスから、これらの XML マッピング ファイルを参照できます。

複数セッション ファクトリの設定ファイルのパッケージング

アプリケーションで複数のデータベース（データ ソース）を使用する必要がある場合、複数の hibernate セッション ファクトリを設定する必要があります。デフォルトでは、Borland AppServer のデプロイメントは、hibernate.cfg.xml のみ検索します。複数のセッション ファクトリを設定するには：

1. hibernate 設定ファイル（hibernate.cfg.xml）と同じ場所に hibernate-borland.properties ファイルを配置します。
2. hibernate-borland.properties ファイルに hibernate のセッション ファクトリ設定ファイルのリストを指定します。各ファイルを必ずセミコロン（;）で区切ります。

例 hibernate.cfg.files=hibernate.cfg.xml;hibernate.cfg1.xml …

メモ プロパティ ファイルにファイルのリストを指定すると、Hibernate デプロイメントは、これらのセッション ファクトリのみ設定します。

EAR モジュールを設定する使用

EJB-JAR または WAR 内で hibernate 設定ファイルとエンティティを含めることが必要です。EJB-JAR は Hibernate エンティティとセッション ファクトリを使用する場合、すべての設定ファイル、XML マッピング、エンティティ クラス、モジュール自体（EJB-JAR または WAR）内で、および適切な場所で、EJB-JAR が必要とするファイルを含める必要があります。EJB-JAR と WAR は、EAR モジュール内でパッケージできます。

メモ EAR 直下にライブラリとして hibernate エンティティと設定ファイルを配置できません。

Hibernate サービスを有効および無効にする

Borland 管理コンソールでは、ユーザーは Hibernate サービスを有効と無効にできます。

Hibernate サービスを有効または無効にするには：

1. Borland 管理コンソールを開きます。
2. [Configurations] を開きます。
3. Hibernate サービスを有効にするには、Hibernate Service を右クリックし、[Enable] を選択します。

   Hibernate サービスを無効にするには、Hibernate Service を右クリックし、[Disable] を選択します。

Hibernate 統計を使用する

Borland 管理コンソールでは、ユーザーは Hibernate 統計ログを有効と無効にできます。

Hibernate 統計ログを有効または無効にするには：

1. Borland 管理コンソールを開きます。
2. [Configurations] を開きます。
3. Hibernate Service サービスを右クリックして、［Properties］を選択します。
4. Hibernate 統計ログを有効にするには、[hibernate.statistic.enable] チェック ボックスをオンにします。

   Hibernate 統計ログを無効にするには、[hibernate.statistic.enable] チェック ボックスをオフにします。
5. [hibernate.statistic.period] フィールドで、Hibernate 統計をロギングする時間間隔（ミリ秒）を入力します。

第 25 章

JMS プロバイダの接続性

Borland AppServer (AppServer) は、一定の必要条件を満たすすべての JMS プロバイダをサポートします。JMS の接続性には 3 つの観点があります。具体的には、実行時の接続性、JMS 管理オブジェクトの設定（接続ファクトリとキュー／トピック）、およびサービス管理です。3 つのレベルすべてが満たされれば最良の結果が得られますが、多くの場合は実行時レベルの接続性だけで、またはベンダー固有の方法で十分です。

Borland AppServer 6.7 には、Tibco EMS 4.2.0 V12 と OpenJMS 0.7.6.1 JMS プロバイダがバンドルされています。OpenJMS は、パーティションレベルのサービスとしてバンドルされています。

実行時の接続性

実行時の接続性は、J2EE 仕様への準拠によって定義されます。CTS に準拠する JMS 製品が JMS 仕様の API もオプションで実装している場合は、AppServer ランタイムにシームレスに組み込むことができます。トランザクションや MDB サポートなどのすべての機能が保持されます。

JMS 製品には、MDB および J2EE コンテナにインターセプトされるメッセージングをサポートするために、トランザクションメッセージング機能が必要です。つまり、JMS のキューやトピックはトランザクションリソースであることが求められます。AppServer では、JMS 製品が JTA XAResource インターフェースを実装し、JMS XA API をサポートしている必要があります。

また、JMS 製品が `javax.jms.ConnectionConsumer` インターフェースをサポートしていることも必要です。後者は、MDB の主たる目的がメッセージの同時消費にあるため特に重要です。これは、ConnectionConsumer インターフェースで実現しています。このメカニズムは、`javax.jms.Session` オブジェクトの一部の最適化メソッドである `Session.run()` と `Session.setMessageListener()` とも協調して機能します。

さらに、BES 6.5 では、両方の JMS プロバイダで CTS 1.3.1 を渡します。

JMS管理オブジェクト（接続ファクトリ、キュー、およびトピック）の設定

JMS プロバイダの管理オブジェクト（接続ファクトリや接続先など）が JavaBeans 仕様（JMS 仕様で使用）に準拠している場合、Borland デプロイメントデスクリプタエディタ（DDEditor）ツールは JMS 製品固有のメカニズムを使用せずにこれらのオブジェクトを定義、編集して、AppServer JNDI ツリーにデプロイメントすることができます。

その他の JMS サービスプロバイダを使用する場合の AppServer と管理オブジェクトの必要条件（キュー、トピック、および接続ファクトリ）の詳細については、後述の 222 ページの「Borland デプロイメントデスクリプタを使用した管理オブジェクトの設定」を参照してください。

Borland デプロイメントデスクリプタを使用した管理オブジェクトの設定

Borland デプロイメントデスクリプタエディタから Tibco と Sonic の管理オブジェクトのプロパティを設定できます。それには、次の手順にしたがいます。

1 管理コンソールから、または［スタート］メニューから単独で Borland デプロイメントデスクリプタエディタを起動します。

2 ［File | New］を選択し、［JNDI Definitions］タブをクリックして前面に表示します。

3 ［JNDI Definitions Archive］を選択し、［OK］をクリックし、新しい JMS オブジェクトを作成します。

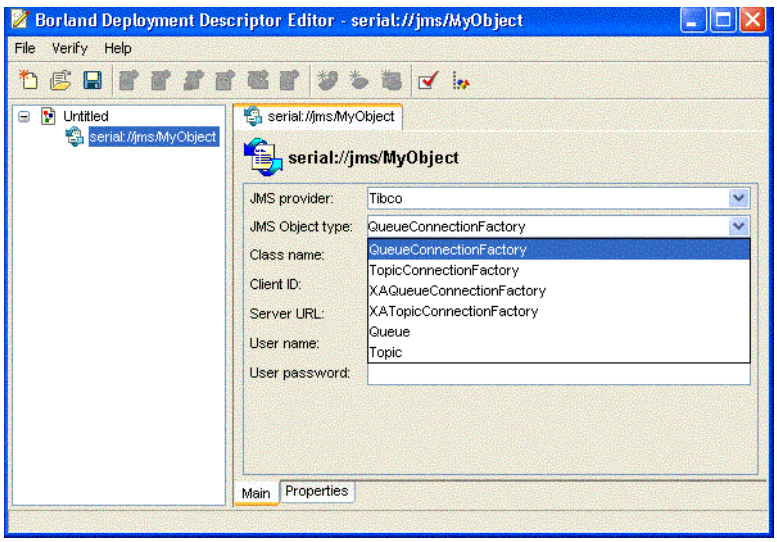
4 左側ペインで［Untitled］を右クリックし、［New JMS Object］を選択します。

5 ［New JMS Object］ダイアログで JMS オブジェクトの名前を指定し、［OK］をクリックします。JMS オブジェクトがアーカイブに表示されます。

6 JMS オブジェクトをクリックし、［Main］タブを選択します。

7 ドロップダウンメニューから各フィールドを選択し、プロパティのフィールドに情報を入力してオブジェクトを設定します。

8 JMS オブジェクトにプロパティを追加するには、［Properties］タブを選択し、［Add］をクリックしてプロパティ（名前、タイプ、値）を追加します。



JMS プロバイダのサービス管理

AppServer サービスコントロールインフラストラクチャは、JMS サービスプロセス（ JMS プロバイダ内での JVM プロセスかネイティブプロセスのいずれかの形式）を最初のクラス管理オブジェクトとして管理できます。サポートされるプロバイダ（ Tibco、または OpenJMS）に、開始、停止、サーバー設定などの操作が提供されます。

Tibco EMS 4.2

Tibco は、J2EE 仕様に準拠する実行時レベルの接続性を実現します。Tibco 4.2 は JMS 1.1 と互換性があり、統一的な JMS API をサポートします。

Tibco の付加価値

Tibco の付加価値は次のとおりです。

- 透過的なインストール
- Tibco Management Console を AppServer 管理コンソールの *［Tools］* メニューから使用できる

Tibco の管理オブジェクトの設定

Tibco の管理オブジェクトのプロパティは AppServer で定義され、Borland デプロイメントデスクリプタエディタを使ってグラフィカルに設定できます。

222 ページの「Borland デプロイメントデスクリプタを使用した管理オブジェクトの設定」を参照してください。

Tibco の自動キュー作成機能

Tibco には、指定されたキューがサーバーに存在しない場合は、Tibco サーバーが必要に応じてキューを作成する自動キュー作成機能があります。

Tibco Management Console

メモ Windows プラットフォームでは、AppServer から Tibco Management Console を起動できます。Windows 以外のすべてのプラットフォームでコンソールを起動するには、`<tibco_home>` ディレクトリから実行可能ファイルを起動します。

BES では、Tibco Management Console を使って細かい設定を実行できます。Tibco Management Console を起動するには、AppServer 管理コンソールの［Tools］メニューから［Tibco Admin Console］を選択します。

フォールトトレラントの Tibco 接続のためのクライアントの設定

プライマリサーバーの障害時にバックアップサーバーに接続するには、クライアントアプリケーションは次のように、接続ファクトリの `jndi-object` XML に複数のサーバー URL を指定する必要があります。

```
<jndi-object>
      <jndi-name>jms/XAConnectionFactory</jndi-name>
      <class-name>com.tibco.tibjms.TibjmsXAConnectionFactory</class-name>
      <property>
         <prop-name>serverUrl</prop-name>
         <prop-type>String</prop-type>
         <prop-value>localhost:7222,anotherhost:7222</prop-value>
      </property>
   </jndi-object>
   <jndi-object>
      <jndi-name>jms/ConnectionFactory</jndi-name>
      <class-name>com.tibco.tibjms.TibjmsConnectionFactory</class-name>
      <property>
         <prop-name>serverUrl</prop-name>
         <prop-type>String</prop-type>
         <prop-value>localhost:7222,anotherhost:7222</prop-value>
      </property>
   </jndi-object>
   <jndi-object>
    <jndi-name>jms/XAQueueConnectionFactory</jndi-name>
    <class-name>com.tibco.tibjms.TibjmsXAQueueConnectionFactory</class-name>
    <property>
     <prop-name>serverUrl</prop-name>
     <prop-type>String</prop-type>
     <prop-value>localhost:7222,anotherhost:7222</prop-value>
    </property>
   </jndi-object>
   <jndi-object>
    <jndi-name>jms/QueueConnectionFactory</jndi-name>
    <class-name>com.tibco.tibjms.TibjmsQueueConnectionFactory</class-name>
    <property>
     <prop-name>serverUrl</prop-name>
     <prop-type>String</prop-type>
     <prop-value>localhost:7222,anotherhost:7222</prop-value>
    </property>
   </jndi-object>
   <jndi-object>
    <jndi-name>jms/XATopicConnectionFactory</jndi-name>
    <class-name>com.tibco.tibjms.TibjmsXATopicConnectionFactory</class-name>
    <property>
     <prop-name>serverUrl</prop-name>
     <prop-type>String</prop-type>
     <prop-value>localhost:7222,anotherhost:7222</prop-value>
    </property>
   </jndi-object>
    <jndi-name>jms/TopicConnectionFactory</jndi-name>
    <class-name>com.tibco.tibjms.TibjmsTopicConnectionFactory</class-name>
    <property>
     <prop-name>serverUrl</prop-name>
     <prop-type>String</prop-type>
     <prop-value>localhost:7222,anotherhost:7222</prop-value>
    </property>
   </jndi-object>
```

Tibco のセキュリティの有効化

メモ SSL の情報については、Tibco のマニュアルを参照してください。Tibco のマニュアルは、`<install_dir>¥jms¥tibco¥doc¥html` にあります。

Tibco のセキュリティを有効にするには、`/<install_dir>/jms/tibco/bin` にある `tibemsd.conf` ファイルを変更するか、Tibco 管理ツールを使って `tibemsd.conf` ファイルを設定します。

メモ 次の手順を実行する前に、Tibco サービスをアクティブ化してください。

1. Borland 管理コンソールの［Tools］メニューから、［Tibco Admin Console］を選択し、Tibco Management Console を起動します。
2. 「`connect`」と入力します。
3. ログイン名とパスワードを入力します。
4. 接続したら、「`set server authorization=enabled`」と入力します。
5. これでセキュリティが有効になります。クライアントの認証では、ユーザーを作成して承認グループに追加する必要があります。たとえば、`create user <name> [<description>] [password=<password>]` というコマンドを使ってユーザーを作成します。
6. 「`add member <group-name> <user-name> [,<user-name2>,...]`」と入力して、メンバーを追加します。

Tibco のセキュリティの無効化

上に示した Tibco のセキュリティを有効にする手順にしたがってください。ただしステップ 4 では、セキュリティを有効にするのではなく、サーバー承認を次のように無効にします。

```
set server authorization=disabled.
```

OpenJMS

OpenJMS は、AppServer パーティションの存続期間に拘束されます。AppServer には、OpenJMS の完全なフットプリントが含まれます。

メモ OpenJMS 7.6 は、JMS 1.0 と互換性があり、統一的な API をサポートしません。

OpenJMS の付加価値は次のとおりです。

- 透過的なインストール
- 自動的なテーブルの作成をサポート
- AppServer のネーミングサービス（JNDI）、トランザクションサービス、およびデータソースと初期状態で統合
- パーティションレベルでのサービス管理
- VisiBroker を使用した RMI コネクタのサポート
- Borland 管理コンソール 使用した存続期間管理

Borland AppServer バージョン 6.7 に AppServer と OpenJMS をインストールする場合、OpenJMS はパーティションレベルのテンプレートとしてパッケージされています。つまり、このテンプレートから作成されたパーティションは、OpenJMS をインプロセスサービスとして取得します。

パーティションテンプレートの次のプロパティは、AppServer と OpenJMS をインストールするときにデフォルトで `true` に設定されます。

- `ejb.mdb.use_jms_threads=true`

  このプロパティは、OpenJMS で開始されるトランザクションを AppServer に伝達するために必要です。

- `ejb.mdb.local_transaction_optimization=true`

  このプロパティは、XA 以外の JMS 接続ファクトリをトランザクションで使用するために必要です。このプロパティを設定しなければ、トランザクション `onMessage` メソッドを持つ MDB がデプロイメントできなくなります。

- `jts.allow_unrecoverable_completion=true`

  デフォルトでは、AppServer はメッセージの永続化に JDataStore データベースを使用します。アプリケーションデータベースがメッセージの永続化に使用するデータベースとは異なる場合に 2 フェーズコミットを実現するには、このプロパティを true に設定します。

各プロパティの詳細については、353 ページの「EJB、JSS、および JTS のプロパティ」を参照してください。

パーティションで OpenJMS サービスだけを実行するスタンドアロンモードで OpenJMS を使用することもできますが、OpenJMS をインプロセスサービスとして使用する場合は、次のような利点があります。

- 2 フェーズコミット（2PC）の使用を回避し、その結果関連するパフォーマンスコストとデプロイメントの複雑さを軽減できます。これには、データベースに接続するパーティションの異なるコンポーネント間で共有される JDBC 接続が含まれます。それには、アプリケーションデータが保存されているデータベースと同じデータベースで OpenJMS によるメッセージの永続化を行います。この 2PC の使用を回避できるのは、AppServer パーティションに埋め込まれたコネクタまたは RMI コネクタによって OpenJMS にアクセスする場合だけです。詳細については、229 ページの「データソースを設定して 2PC を最適化する」を参照してください。
- すべてのコンポーネントは 1 つの仮想マシンで一元管理されるので、TCP/IP のコストを回避できます。JMS クライアントライブラリは、通常の Java 呼び出しを使ってインプロセスの JMS サービスを呼び出します。逆も同じです。
- VBJ はローカルの呼び出しを最適化するので、アプリケーションで 2 種類の接続を用意する必要がありません。JMS サーバーに対するクライアントの位置に無関係な RMI コネクタだけを使用できます。

OpenJMS 製品マニュアルは、`<appserver_install>/jms/openjms/docs` ディレクトリにあります。

OpenJMS の JNDI オブジェクトの設定

AppServer の各パーティションで OpenJMS のインスタンスをホストできるので、パーティションごとに専用の設定ファイル `openjms.xml` があります。`openjms.xml` ファイルには、インスタンスがホストするさまざまな OpenJMS コネクタや JNDI オブジェクトの情報が含まれます。

メモ AppServer は、OpenJMS の管理 GUI をサポートしていません。OpenJMS のキューを作成および削除するには、`openjms.xml` ファイルを編集します。

新しいキュー、トピック、接続ファクトリを追加するには、設定ファイル `openjms.xml` を変更する必要があります。このファイルにアクセスするには、次の手順にしたがいます。

1. Borland 管理コンソールの左側ペインで対応するパーティションを選択します。
2. 左側ペインで OpenJMS サービスをクリックします。
3. ドロップダウンメニューから［**Properties**］を選択します。
4. プロパティペインで［openjms.xml］タブをクリックして前面に表示します。
5. ファイルを編集して JNDI オブジェクトを追加します。

   このファイルの詳細については、`<appserver_install>/jms/openjms/docs` にある OpenJMS マニュアルを参照してください。

次のサンプルコードに、キューとトピックの接続ファクトリを追加する方法を示します。

- アプリケーションで必要な数だけファクトリを追加できます。
- 追加する各オブジェクトに一意の名前を付けて、JNDI で上書きされないようにします。名前は複数のインスタンス間で一意にする必要があります。
- 埋め込みスキームには、それぞれ 1 つ以上の TopicConnectionFactory と QueueConnectionFactory が必要です。
- コネクタにポートを指定しない場合は、デフォルトのポートが使用されます。詳細については、`<appserver_install>/jms/openjms/docs` にある OpenJMS マニュアルを参照してください。

```
<Configuration>

  <ServerConfiguration host="localhost" embeddedJNDI="false" />

  <JndiConfiguration>
    <property name="java.naming.factory.initial"
              value="com.inprise.j2ee.jndi.CtxFactory" />
    <property name="java.naming.provider.url"
              value="serial://" />
  </JndiConfiguration>

  <Connectors>
    <Connector scheme="embedded">
      <ConnectionFactories>
        <QueueConnectionFactory name="jms/EmbeddedQueueConnectionFactory" />
        <TopicConnectionFactory name="jms/EmbeddedTopicConnectionFactory" />
      </ConnectionFactories>
    </Connector>

   <Connector scheme="tcp">
      <ConnectionFactories>
        <QueueConnectionFactory name="jms/TcpQueueConnectionFactory" />
        <TopicConnectionFactory name="jms/TcpTopicConnectionFactory" />
        <QueueConnectionFactory name="jms/qcf" />
        <QueueConnectionFactory name="jms/QueueConnectionFactory" />
        <QueueConnectionFactory name="jms/xaqcf" />
        <TopicConnectionFactory name="jms/tcf" />
        <TopicConnectionFactory name="jms/TopicConnectionFactory" />
        <TopicConnectionFactory name="jms/xatcf" />
      </ConnectionFactories>
    </Connector>

    <Connector scheme="rmi">
      <ConnectionFactories>
```

```
            <QueueConnectionFactory name="jms/qcf" />
            <QueueConnectionFactory name="jms/QueueConnectionFactory" />
            <QueueConnectionFactory name="jms/xaqcf" />
            <TopicConnectionFactory name="jms/tcf" />
            <TopicConnectionFactory name="jms/TopicConnectionFactory" />
            <TopicConnectionFactory name="jms/xatcf" />
          </ConnectionFactories>
        </Connector>

    </Connectors>
```

メモ アプリケーションで JNDI 検索を実行する準備として Borland デプロイメントデスクリプタに JMS リソースオブジェクトを設定する場合は、jndi-name 要素の値に serial:// をプレフィクスとして追加します。たとえば、serial://jms/q などとします。OpenJMS リソースオブジェクトは、DAR ファイルとは独立してデプロイメントされます。BAS パーティションの起動時には、serial:// プレフィクス名によって直接 JNDI でバインドされます。OpenJMS リソースオブジェクトの JNDI 検索を実行するアプリケーションは、オブジェクトを解決するために serial:// プレフィクスを使用する必要があります。

OpenJMS の接続モード

OpenJMS では、クライアントは埋め込みコネクタ、TCP コネクタ、または RMI コネクタを使ってアクセスできます。

OpenJMS がインプロセスとしてインストールされている場合は、埋め込みコネクタを使用します。`openjms.xml` ファイルの embedded connector（埋め込みコネクタ）セクションに、ローカルで必要なすべての接続ファクトリを指定します。埋め込みコネクタまたは RMI コネクタを使って OpenJMS をパーティションレベル（インプロセス）サービスとして使用する場合にだけ 2PC を最適化できます。埋め込みモードでは、JMS クライアントはローカルの Java 呼び出しを使って JMS サーバーにアクセスし、埋め込みのキュー／接続ファクトリを使用します。この接続ファクトリを使用すれば、最適な方法で TCP/IP のコストを回避できます。

OpenJMS をアウトプロセスサービスで使用する場合は、RMI コネクタまたは TCP コネクタを使用する必要があります。AppServer の RMI コネクタは RMI-over-IIOP を使用するように設定されるので、クライアントから JMS サーバーへのトランザクションコンテキストを実行できます。クライアントが OpenJMS サーバーと同じ場所にある場合は、ローカルの呼び出しを最適化できるので、さらに効率的になります。TCP コネクタはカスタムプロトコルに基づいているので、トランザクションコンテキストを保持していません。

重要 TCP コネクタまたは RMI コネクタを使用しないときは無効にできます。埋め込みコネクタは、使用しない場合でも無効にしないでください。埋め込みコネクタは、AppServer のパーティションレベルサービスの一部として内部的に OpenJMS のサービス管理（起動、停止など）のために使用します。

OpenJMS のデータソースの変更

デフォルトでは、OpenJMS サービスが起動すると、`partition.xml` ファイルを確認して OpenJMS メッセージが永続化されるデータソースの場所を特定します。このデータソースエントリは、DAR ファイルに存在する必要があります。DAR ファイルにエントリが見つからない場合、OpenJMS サービスは `openjms.xml` ファイルに指定されているデータソースをデフォルトで使用します。OpenJMS で `partition.xml` ファイルで設定されたデータソースだけを使用するには、`openjms.xml` ファイルで <DatabaseConfiguration> エントリをコメントアウトします。この場合、データソースが見つからない場合はエラーメッセージが表示されます。データソースを変更し、J2EE アプリケーションが使用するデータソースと同じデータソースをポイントするようにます。データソースを変更するには、次の手順にしたがいます。

1 Borland 管理コンソールの左側ペインで OpenJMS サービスをクリックします。

2 ドロップダウンメニューから［**Properties**］を選択します。

3 ［**General**］タブの［**Name**］テキストボックスにデータソースのパスを入力します。

4 パーティションを再起動するときに以前に保存されている（未配信の）メッセージを削除しない場合は、［Clean messages on startup］チェックボックスのチェックをはずします（オプション）。配信されたメッセージは、自動的にデータベースから削除されます。何らかの原因で配信できなかったメッセージはデータベースに残ります。このボックスをチェックすると、このメッセージがクリーンアップされます。デフォルトでは、このチェックボックスはチェックされます。

`openjms.xml` 設定ファイルで正しいデータベースドライバを指定することもできます。このファイルにアクセスするには、OpenJMS サービスを右クリックして表示されるメニューの［Properties］を選択します。プロパティペインで、［`openjms.xml`］タブをクリックします。

OpenJMS のテーブルの作成

JDataStore 以外のデータベースを選択する場合は、データベースを使用する前に適切なテーブルを作成する必要があります。JDataStore では、テーブルはあらかじめ作成されます。その他のデータベースでテーブルを作成するには、OpenJMS が提供するスクリプトを使用します。スクリプトは、`<bas_install>¥jms¥openjms¥config¥db` ディレクトリにあります。詳細については、『OpenJMS User 開発者ガイド』を参照してください。このガイドは、`<appserver_install>/jms/openjms/docs` ディレクトリにあります。

データソースを設定して 2PC を最適化する

OpenJMS をパーティションレベルのサービスとして使用すれば、2 フェーズコミットを最適化できます。OpenJMS は、あらゆるリレーショナルデータベースでデータを永続化するように設定できます。デフォルトでは、AppServer はパーティションの JDataStore データベースを使ってメッセージを永続化します。デフォルトのデータソースを変更し、J2EE アプリケーションが使用するデータソースと同じデータソースをポイントするようにます。データソースの変更方法の詳細については、229 ページの「OpenJMS のデータソースの変更」を参照してください。このようにして OpenJMS とアプリケーションで単一のトランザクションリソースを使用すれば、2 フェーズコミットを回避できます。

重要 パーティションに複数のメッセージアプリケーションがあり、それぞれが個別のアプリケーションデータのデータソースを使用する場合、各アプリケーションに対して 2 フェーズコミットを最適化することはできません。2 フェーズコミットの最適化は、OpenJMS と同じデータソースを使用するアプリケーションだけで有効です。OpenJMS は、パーティションのアプリケーションの数にかかわらず、パーティションのすべてのアプリケーションの単一のデータソースのデータだけを永続化できます。したがって、パーティションに複数のアプリケーションがあり、各アプリケーションが個別のデータベースにデータを保存する場合、OpenJMS データソースがポイントできるデータベースは 1 つだけです。2PC を最適化できるは、このデータベースにデータを保存するアプリケーションだけです。

OpenJMS のセキュリティ設定

OpenJMS バージョン 0.7.6 の認証とセキュリティ機能は、次の AppServer 設定で使用できます。

1 TCP コネクタによる OpenJMS 認証

2 VBJ ベースの RMI コネクタによる OpenJMS 認証

メモ HTTPS 接続と TCPS 接続は AppServer 6.7 ではサポートされていません。

`openjms.xml` ファイルの次の XML コードに、上記の 1 と 2 の設定だけでセキュリティを有効にする方法の例を示し、認証されたユーザーのリストを提供します。

```
<SecurityConfiguration securityEnabled="true"/>
  <Users>
    <User name="admin" password="admin"/>
    <User name="j2ee" password="j2ee"/>
  </Users>
```

3 AppServer セキュリティを使用する VBJ ベースの RMI コネクタでの OpenJMS 認証

この設定でセキュリティを設定する方法については、`<appserver_install>/examples/security/Readme.html` のドキュメントを参照してください。

OpenJMS のセキュリティの使い方の詳細については、`<appserver_install>/jms/openjms/docs` ディレクトリにある OpenJMS マニュアルを参照してください。

OpenJMS のパーティションレベルのプロパティの指定

OpenJMS は、パーティションレベルのサービスとして AppServer に統合するために、新しいサービスとしてパーティションの設定に導入されます。次のプロパティは、`partition.xml` ファイルの OpenJMS のプロパティです。この設定ファイルは、`<appserver_install>/var/domains/base/configurations/<my_config>/mos/<openjms_partition>/adm/properties` ディレクトリにあります。

`partition.xml` ファイルの次のコードは、OpenJMS をパーティションレベルのサービスとして作成します。

```
- <service name="jms"
        runas.propstorage="management_runas.properties"
        version="6.7" description="JMS Service based on OpenJMS(tm)
           version 0.7.6.1"
        vendor="Borland Software Corporation"
        class="com.borland.enterprise.server.services.PartitionService"
        startup.synchronization="service_ready"
        startup.service_ready.max_wait="0"
        shutdown.synchronization=""
        shutdown.phase="1">
  <properties lifecycle.class="com.borland.jms.JmsPartitionService"
        openjms.configfile="adm/openjms/conf/openjms.xml"
        openjms.home="../../../../../../../jms/openjms"
        openjms.clean_messages_on_startup="true"
        openjms.datasource="serial://datasources/OpenJmsDataSource"
        openjms.sql_file="adm/openjms/conf/openjms.sql"
        openjms.datasource_lookup_interval="1"
        openjms.max_datasource_lookup_retries="1" />
  </service>
```

次の表にプロパティを示します。

プロパティ名	説明	デフォルト値
lifecycle.class	インプロセスの JMS サービスの追加に使用します。 OpenJMS(tm) では、このプロパティの値を com.borland.jms.JmsPartitionService にする必要があります。 このプロパティに指定されているクラスが Java CLASSPATH にある場合、パーティション起動プログラムのリフレクションベースのコードが動的にそれを検出します。 検出されると、起動プログラムはサービスをロードして開始します。	com.borland.jms.JmsPartitionService
openjms.configfile	このプロパティは、 設定ファイルの場所を指定します。 場所はパーティションの現在の作業ディレクトリに対して相対的になります。 このファイルは、 OpenJMS(tm) の設定が保存される中央の場所になります。埋め込みの OpenJMS(tm) サービスが AppServer パーティションと動作するには、このファイルが必要です。 このファイルには、OpenJMS(tm) サービスの開始時に作成する必要がある JNDI オブジェクト（キューとトピック）の一覧も含まれます。	adm/openjms/conf/openjms.xml
openjms.home	このプロパティは、 OpenJMS(tm) をインストールする場所を指定します。 OpenJMS は、ここで指定される値を使用して、さまざまなリソースの場所を特定します。	<AppServerInstallRoot>/jms/openjms
openjms.clean_messages_on_startup	このプロパティは、パーティションの再起動時に JMS メッセージを保存しているデータベースをクリーンアップするかどうかを指定します。このプロパティは、現在 JDataStore だけで使用できます。 その他のデータベースでは、メッセージを手動で削除する必要があります。	true

プロパティ名	説明	デフォルト値
openjms.datasource	このプロパティは、 OpenJMS(tm) でメッセージを永続化するために使用するデータソースの JNDI 名を指定します。 このデータソースがアプリケーションが使用しているデータソースと同じ場合、 JDBC 接続プールは両者で共有され、メッセージの永続化とアプリケーションへのデータアクセスの提供に対して単一の JDBC 接続が使用されるので、 2PC の必要はありません。 指定されたデータソースが起動時に JNDI 名前空間で使用できない場合、 起動コードは以下の openjms.datasource_lookup_interval プロパティと openjms.max_datasource_lookup_retries プロパティを使ってターゲットデータソースへの接続を複数回試みます。 それでも検索に失敗すると、初期化コードが設定ファイル（openjms.xml）に指定されている情報から内部的にデータソースを構築します。 **メモ：** 起動コードは、ユーザーが指定したデータソースを使用できない場合にだけ openjms.xml ファイルの情報を使用します。 データソースが JNDI でデプロイメント済みまたは使用できる場合は、 openjms.xml ファイルの RDBMS 設定は無視されます。	serial://datasources/JDSLocal
openjms.sql_file	このプロパティは、データベーステーブルを削除および作成するための SQL 文を含むファイルの指定に使用します。このテーブルは、 OpenJMS(tm) がメッセージの永続化に使用します。	adm/openjms/conf/openjms.sql
openjms.datasource_lookup_interval	このプロパティは、 連続してデータソース検索を試みる間隔を指定します。 上記の openjms.datasource property プロパティを参照してください。	1 秒
openjms.max_datasource_lookup_retries	このプロパティは、デフォルトのデータソースを使用する前にデータソースを検索する回数を指定します。上記の openjms.datasource プロパティを参照してください。	5 秒
openjms.recreate_database_on_startup	このプロパティを設定すると、 サービスを起動するたびにデータベースが再作成されます。これは前のメッセージがその後の実行に必要ない場合（テスト中など）に便利です。	false
openjms.database.softcommit	このプロパティは、 JDataStore を使って JMS メッセージを永続化する場合にだけ適用されます。 このプロパティによってコミットプロセスのパフォーマンスは向上しますが、 ごく一部の失敗の際の回復能力が犠牲になります。 詳細については、 JDataStore ドキュメントを参照してください。	true

プロパティ名	説明	デフォルト値
openjms.database	このプロパティは JDS だけに適用され、JDS データベースの名前を指定するために使用します。	openjms.jds
openjms.use_bes_transactions	OpenJMS は、メッセージをディスパッチする前にトランザクションを開始します。OpenJMS を含むパーティションのトランザクションサービスを使用します。パーティションで使用できるトランザクションサービスがない場合は、スマートエージェントドメインから選択されます。トランザクションで OpenJMS を使用する場合はこのプロパティを使用します。このプロパティは、トランザクションを実行しないメッセージアプリケーションでは無効です。ただし、余分なトランザクションの開始と伝達による負荷を抑制するには、このプロパティをオフにします。	true

OpenJMS トポロジ

重要 TCP コネクタを使用する OpenJMS サービスが 2 つある場合は、openjms.xml ファイルで必ず別のポート番号を指定してください。このファイルを開くには、Borland 管理コンソール の OpenJMS サービスを右クリックしてドロップダウンメニューから［Properties］を選択します。プロパティペインで、［openjms.xml］タブをクリックします。

OpenJMS は、次の 2 つのトポロジで実行するように設定できます。

- **サーバー共有モード** - OpenJMS サービスは専用のパーティションでホストされ、パーティションのほかのサービスは無効になります。OpenJMS サービスは、OpenJMS パーティションがある設定と同じ osagent ドメインのすべてのパーティションに対する共有サービスとして使用できます。リモートクライアントは、RMI コネクタまたは TCP コネクタを使って OpenJMS にアクセスできます。OpenJMS は、この設定ファイルに指定された JNDI オブジェクトをバインドするためのネーミングサービスを必要とするので、トランザクションサービスとネーミングサービスは OpenJMS をホストするパーティションで有効にするか、またはスマートエージェントドメインで使用可能にする必要があります。
- **埋め込みサービスモード** - OpenJMS は設定されている各パーティションの埋め込みサービスとして実行されます。各パーティションは、TCP コネクタまたは RMI コネクタではなく仮想マシンの埋め込み OpenJMS のコネクタを使用します。JMS クライアントは、埋め込みのキュー／トピック接続ファクトリを使用します。この接続ファクトリを使用すれば、最適な方法で TCP/IP のコストを回避できます。JMS クライアントはこのモードで RMI コネクタを使用できますが、パフォーマンスを最大にするためには、ローカル（埋め込み）のコネクタを使用する必要があります。

メモ 複数の OpenJMS サービスインスタンスが AppServer の 1 つのスマートエージェントドメインで実行されている場合、データベースの共有性や実行中のインスタンスに対する自動フェイルオーバー機能はありません。これは、OpenJMS に対するクラスタリングがサポートされていないためです。

OpenJMS でのメッセージ駆動型 Bean（MDB）の使用

MDB をサポートする AppServer パーティションでは、MDB は JMS サーバーにアクセスできる必要があります。MDB が OpenJMS サーバーにアクセスできるようにするために、次のことを確認してください。

1. OpenJMS がインプロセスサービスとしてパーティションにインストールされ有効になっているか、またはドメインで使用できる。OpenJMS サービスを右クリックし、メニューから［Start］を選択してサービスを有効にします。
2. リソースリファレンスが正しいタイプの接続ファクトリをポイントするように `ejb-jar.xml` ファイルで設定されている。

重要 MDB にトランザクションアクセスが必要な場合は、埋め込み接続ファクトリまたは RMI 接続ファクトリを MDB で使ってトランザクションの伝達をサポートする必要があります。

その他の JMS プロバイダ

Borland AppServer は、SonicMQ 6.0/6.1 および WebSphereMQ 5.3/6.0 JMS プロバイダをサポートします。SonicMQ を AppServer に統合する方法については、235 ページの「SonicMQ の Borland AppServer との統合」のセクションを参照してください。WebSphereMQ を AppServer に統合する方法については、239 ページの「WebSphereMQ の Borland AppServer（BAS）との統合」のセクションを参照してください。

第 26 章

SonicMQ の Borland AppServer との統合

このドキュメントでは、単独でインストールされている SonicMQ 6.0/6.1 JMS プロバイダと協調して動作するように、Borland AppServer（AppServer）を設定する手順を示します。Both SonicMQ バージョン 6.0 と 6.1 は、どちらも JMS 1.1 に準拠しています。

メモ SonicMQ を別途購入する必要があります。このプロダクトは AppServer 6.7 にはバンドルされていません。

SonicMQ のインストール

SonicMQ は、AppServer のインストールに依存しない場所にインストールします。SonicMQ の機能を Sonic 管理コンソールから管理するために、必ず管理機能をインストールしてください。

AppServer での SonicMQ 管理オブジェクトの設定

Borland 独自の DAR モジュールに JNDI からアクセスする JMS 管理オブジェクトを定義する必要があります。AppServer の Borland デプロイメントデスクリプタエディタ（DDEditor）ツールを使用すると、DAR モジュール内に管理オブジェクトを作成できます。222 ページの「Borland デプロイメントデスクリプタを使用した管理オブジェクトの設定」を参照してください。

Sonic JMS Administered Objects ツールを使用して管理オブジェクトに関して設定できるすべてのプロパティについては、『*SonicMQ V6.1 Configuration and Management Guide*』を参照してください。

DAR モジュールの JMS 接続ファクトリオブジェクトの定義に適用できる AppServer 関連プロパティの説明は、『*Borland AppServer 開発者ガイド*』の 199 ページの「JMS の使い方」を参照してください。

AppServer 環境での SonicMQ ライブラリモジュールの解決

SonicMQ サーバーにアクセスする J2EE アプリケーションをデプロイメントする場合は、SonicMQ 6.0/6.1 クライアントライブラリ `sonic_Client.jar` と `sonic_XA.jar`、およびそれらに依存するライブラリを AppServer によってロードする必要があります。

AppServer 内の SonicMQ クライアントライブラリを有効にするには、`<AppServer>/bin` に置かれている JMS 関連の設定ファイルに、次のようにして更新を適用する方法をお勧めします。

- `sonic.config` ファイルを編集して、`jms.home` の値を外部の SonicMQ インストールのルートディレクトリに設定します。たとえば、次のようになります。

  ```
  set jms.home=C:/SonicMQ/V61
  ```

- `jms.config` ファイルを編集します。ステートメントのコメントを解除して、`sonic.config` をインクルードします。他の JMS プロバイダの include ステートメントがコメントになっていることを確認してください。

  ```
  #include $var(installRoot)/bin/tibco.config
  #include $var(installRoot)/bin/openjms.config
  include $var(installRoot)/bin/sonic.config
  ```

これにより、SonicMQ クライアントライブラリは、すべての AppServer パーティション、および AppServer appclient ツールが実行する J2EE クライアントアプリケーションによって解決できるようになります。

AppServer にデプロイメントされた SonicMQ キューでの自動キュー作成の設定

SonicMQ JMS キューの定義を含む DAR モジュールをパーティションにデプロイメントするとき、目的の SonicMQ サーバーに JMS キューを自動的に作成するように AppServer を設定できます。JMS キューを自動的に作成するためには、特定の SonicMQ 管理ライブラリを AppServer から使用できる必要があります。これらのライブラリは、パーティションのクラスパスからロードする必要があります。これは、AppServer の設定ファイル `sonic.config` と `jms.config` を上に示したように更新することによって可能になります。また、次の手順を実行する必要があります。

- パーティションの設定ファイル `partition.xml` 内のネーミングサービス定義で、`jns.auto-create-queues` プロパティが次のように true に設定されていることを確認してください。

  ```
  <service name="visinaming"
           runas.propstorage="management_runas.properties" version="6.7"
          description="Naming Service" vendor="Borland Software Corporation"
          class="com.borland.enterprise.server.services.naming.NamingService"
           startup.synchronization="service_ready"
  startup.service_ready.max_wait="0"
           shutdown.synchronization="" shutdown.phase="1">
           <properties jns.name="namingservice"
           jns.auto-create-queues="true">
           </properties>
  </service>
  ```

- パーティションの `partition-server.config` ファイルを更新し、次のようにして目的の SonicMQ サーバーを検索できるようにします。
  - **a** 管理コンソールを開きます。
  - **b** コンソールの左端の［Installation］アイコンをクリックして、［Installation］表示に切り替えます。
  - **c** 左側のペインで、変更するパーティションに移動します。右側のペインにパーティションの［General Properties］ページが開きます。
  - **d** 右側のペインの下部にある［Files］タブをクリックします。
  - **e** 左下のペインで、partition-server.config を選択します。
  - **f** ファイルの最後までスクロールして、変更するプロパティだけに対して次のように入力します。

    ```
    vmprop <property_name>=<value>
    ```

    この操作は、次の 5 つのプロパティのどれについてもできます。

プロパティ	デフォルト値
sonicmq.domainName	domain1
sonicmq.brokerURL	tcp://localhost:2506
sonicmq.user	Administrator
sonicmq.pwd	Administrator
sonicmq.brokerName	/Brokers/Broker1

  - **g** 編集の結果を保存してパーティションを再起動します。

メモ キューが自動的に作成されるようにするには、SonicMQ JMS キューを使用する DAR モジュールをデプロイメントする前に、SonicMQ Server をアクティブにしておく必要があります。

第 27 章

WebSphereMQ の Borland AppServer（BAS）との統合

このドキュメントでは、単独でインストールされている WebSphereMQ 5.3/6.0 JMS プロバイダと協調して動作するように、Borland AppServer（AppServer）を設定する手順を示します。

メモ WebSphereMQ を別途購入する必要があります。このプロダクトは AppServer 6.7 にはバンドルされていません。

サポートするバージョン

WebSphereMQ 5.3 と 6.0 は、いずれもこのプロダクトとの動作が保証されています。

WebSphereMQ の設定

WebSphereMQ を設定するには、次の手順にしたがってください。

WebSphereMQ 5.3

WMQ 5.3 をインストールした直後の状態では、JMS 1.1 API はサポートされません。JMS1.1 の機能を利用するには、修正パック 06（CSD06）以上を WMQ 5.3 に追加インストールする必要があります。

「標準の」WebSphereMQ Client は、クライアントアプリケーションが接続されているキューマネージャによって管理される、ローカル（1 フェーズコミット）トランザクションだけをサポートします。分散トランザクション（2PC）をサポートするには、WebSphereMQ Extended Transactional Client をインストールする必要があります。

WebSphereMQ Extended Transactional Client は、WebSphereMQ バージョン 5.3 の有償の機能です。これによって WebSphereMQ の機能が拡張され、WebSphereMQ クライアントアプリケーションは、グローバルに調整されるトランザクションに参加できます。つまり、WebSphereMQ クライアントアプリケーションは 2 フェーズコミット（XA に適合）を利用できるようになり、外部トランザクションマネージャが管理するグローバルトランザクションに加わることができます。

WebSphereMQ 6.0

WebSphereMQ 6.0 のデフォルトのインストールは、JMS 1.1 API をサポートします。

WebSphereMQ 6.0 は、分散トランザクション（2PC）のサポートが組み込まれているため、MQ Extended Transactional Client をインストールする必要はありません。

WebSphereMQ による管理オブジェクトの設定

WebSphereMQ の管理オブジェクトのプロパティは BES で定義され、Borland デプロイメントデスクリプタエディタを使ってグラフィカルに設定できます。222 ページの「Borland デプロイメントデスクリプタを使用した管理オブジェクトの設定」を参照してください。

WebSphereMQ 5.3 で使用できる JNDI プロパティと他の設定オプションの詳細なリストは、http://publibfp.boulder.ibm.com/epubs/pdf/csqzaw12.pdf に公開されているドキュメント「*WebSphereMQ Using Java*」を参照してください。

WebSphereMQ 6.0 で使用できる JNDI プロパティと他の設定オプションの詳細なリストは、http://publibfp.boulder.ibm.com/epubs/pdf/csqzaw13.pdf に公開されているドキュメント「*WebSphereMQ Using Java*」を参照してください。

実行時の WebSphereMQ ライブラリモジュールの検索

WMQ5.3 サーバーにアクセスする J2EE アプリケーションをデプロイメントする場合は、WMQ 5.3 Client のライブラリを BAS パーティションにロードする必要があります。BAS パーティションで必要とされるライブラリの全セットを以下に示します。

- `com.ibm.mq.jar`
- `com.ibm.mqjms.jar`
- `com.ibm.mqbind.jar`
- `com.ibm.mqetclient.jar`（この jar は WMQ Extended Transactional Client のインストールに含まれています）

これらのライブラリを BAS で使用できるようにする方法の 1 つは、J2EE アプリケーションをホストする BAS パーティションにライブラリをデプロイメントすることです。ただし、`<BAS_install>/bin` にある JMS 関連設定ファイルを次のように更新する方が優れた方法です。

- `wmq53.config` を編集して、`jms.home` の値を外部の WMQ5.3 インストールのルートディレクトリに設定します。
- `jms.config` ファイルを編集します。include ステートメントのコメントを解除して、`wmq53.config` をインクルードします。他の JMS プロバイダの include ステートメントがコメントになっていることを確認してください。

```
#include $var(installRoot)/bin/tibco.config
#include $var(installRoot)/bin/openjms.config
#include $var(installRoot)/bin/sonic.config
include $var(installRoot)/bin/wmq53.config
```

これにより、WMQ5.3 クライアントライブラリは、すべての BAS パーティション、および BAS ツール appclient が実行する J2EE クライアントアプリケーションによって解決できるようになります。

WebSphereMQ 6.0

WebSphereMQ 6.0 サーバーにアクセスする J2EE アプリケーションをデプロイメントする場合は、WebSphereMQ 6.0 Client ライブラリを BAS パーティションにロードする必要があります。BAS パーティションで必要とされるライブラリの全セットを以下に示します。

- `com.ibm.mq.jar`
- `com.ibm.mqjms.jar`
- `dhbcore.jar`
- `com.ibm.mqetclient.jar` （拡張トランザクションクライアント）

これらのライブラリを BAS で使用できるようにする方法の 1 つは、J2EE アプリケーションをホストする BAS パーティションにライブラリをデプロイメントすることです。ただし、`<BAS_install>/bin` にある JMS 関連設定ファイルを次のように更新する方が優れた方法です。

- `wmq60.config` ファイルを編集して、`jms.home` の値を外部の WebSphereMQ 6.0 インストールのルートディレクトリに設定します。
- `jms.config` ファイルを編集します。include ステートメントのコメントを解除して、`wmq53.config` をインクルードします。他の JMS プロバイダの include ステートメントがコメントになっていることを確認してください。

```
#include $var(installRoot)/bin/tibco.config
#include $var(installRoot)/bin/openjms.config
#include $var(installRoot)/bin/sonic.config
include $var(installRoot)/bin/wmq60.config
```

これにより、WebSphereMQ 6.0 クライアントライブラリは、すべての BAS パーティション、および BAS ツール appclient が実行する J2EE クライアントアプリケーションによって解決できるようになります。

第 28 章

JACC の使い方

Java Authorization Contract for Containers（JACC）仕様は、J2EE アプリケーションサーバーと承認ポリシープロバイダとの間のサブコントラクトを定義しています。すべての J2EE アプリケーションコンテナ、Web コンテナ、およびエンタープライズ Bean コンテナは、このコントラクトをサポートしている必要があります。この仕様によって定義されるコントラクトは、3 つのサブコントラクトに分けられます。これらのサブコントラクト全体により、承認プロバイダのインストールおよび設定が説明されます。この承認プロバイダは、コンテナがアクセスの決定を実行する際に使用されます。

JACC コントラクト

3 つのサブコントラクトとは、Provider Configuration サブコントラクト、Policy Configuration サブコントラクト、および Policy Decision and Enforcement サブコントラクトです。

Provider Configuration サブコントラクト

Provider Configuration サブコントラクトは、ポリシープロバイダをコンテナと統合するためにプロバイダとコンテナが満たす必要がある要件を定義しています。

Policy Configuration サブコントラクト

Policy Configuration サブコントラクトは、宣言的 J2EE 認証ポリシーから J2SE ポリシープロバイダ内のポリシーステートメントへの変換をサポートするために、コンテナデプロイメントツールとプロバイダの間のやり取りを定義します。

Policy Decision and Enforcement サブコントラクト

Policy Decision and Enforcement サブコントラクトは、コンテナのポリシー適用ポイントと J2EE コンテナが必要とするポリシー決定との間のやり取りを定義します。

JACC ベースの承認の動作

JACC により、アプリケーションサーバー内の EJB と Web コンテナはサードパーティの承認プロバイダとやり取りを行い、J2EE リソースへのアクセスが行われると承認を判断します。J2EE アプリケーションサーバー内の Web および EJB コンテナは、JACC 互換の承認プロバイダを使用して、リソースやサービスへのクライアントアクセスを制限します。プロバイダは、アプリケーションのデプロイメント時にデプロイメントツールによって伝達されたポリシー情報に基づいて、この制限を実行します。プロバイダは、この情報をリポジトリに格納して、承認を判断するときに使用します。承認の判断は、プリンシパル（ユーザー）が特定のリソースにアクセスするために必要な特権を持つロールに所属しているかどうかに基づいて、プロバイダによって行われます。

アプリケーションのデプロイメント時に、AppServer は次を実行します。

1. デプロイメントされるモジュールを一意に識別する固有の contextID を作成します。
2. モジュールの各リソースにアクセスするために必要な許可のセットによる PolicyConfiguration を構築します。
3. JACC API を通じて、セキュリティポリシー情報をプロバイダに伝達します。

クライアントまたはユーザーが EJB メソッド、サーブレット、または URL へのアクセス要求を行うと、次が実行されます。

1. EJB コンテナまたは Web コンテナは、適切な許可オブジェクトと、呼び出し元のプリンシパルを含む ProtectionDomain オブジェクトを作成します。
2. 次にコンテナは、プロバイダによって実装された java.security.Policy オブジェクトの Policy.implies メソッドを呼び出し、この 2 つのオブジェクトをプロバイダに渡します。
3. プロバイダは、保存してあるポリシー情報に基づいて（プリンシパルとロールの対応を使用して）判断を行い、コンテナにブール値を返します。
4. プリンシパルが所属しているロールにリソースへのアクセス許可がある場合、implies メソッドは true を返し、このユーザーはコンテナによってリソースへのアクセスを許可されます。そうでない場合は false が返され、このユーザーはリソースへのアクセスを拒否されます。

Borland AppServer での JACC プロバイダの設定

AppServer 内の JACC プロバイダは、Provider Configuration Subcontract セクションで指定された標準の java.security.Policy オブジェクトを実装します。これは、アクセス決定を行うために使用されます。また JACC プロバイダは PolicyConfigurationFactory クラスと PolicyConfiguration インターフェースも実装しているため、デプロイメントツールはアプリケーションのデプロイメント時にすべてのセキュリティ要素をプロバイダに伝達できます。

次のプロパティは、AppServer JACC プロバイダのインストールを制御します。

プロパティ名	説明	デフォルト値
javax.security.jacc.policy.provider	アプリケーションサーバーによってポリシーの置換に使用されるポリシー実装クラスを指定します。	com.borland.security.jacc.provider.BESJACCPolicy
javax.security.jacc.PolicyConfigurationFactory.provider	プロバイダの PolicyConfigurationFactory 実装クラスを指定します。	com.borland.security.jacc.provider.BESPolicyConfigurationFactory

AppServer 管理コンソールを使用した JACC プロバイダの設定

JACC プロバイダは、AppServer 管理コンソールを使用して設定できます。または、JACC プロバイダのプロパティを `partition_server.config` ファイルで設定できます。

AppServer 管理コンソールを使用してプロパティを設定するには、次の手順にしたがいます。

1 コンソールの左ペインで、パーティション名を選択します。

2 パーティション名を右クリックして、表示されるメニューから［プロパティ］を選択します。

3 ［Partition Properties］ページが表示されます。

4 ［セキュリティ］タブをクリックします。

5 ［JACC Properties］ボックスで、2 つのプロパティを設定します。

設定ファイルによる JACC プロバイダの設定

`partition_server.config` ファイルで JACC プロバイダのプロパティを設定するには、次の手順にしたがいます。

1 次のディレクトリに移動します。

```
<install_dir>\var\domains\base\configurations\j2eeSample\mos\\adm\properties
```

2 `partition_server.config` ファイルを開きます。

3 次の行を見つけます。

```
#JACC provider configuration
vmprop
javax.security.jacc.policy.provider=com.borland.security.jacc.provider.BESJACCPolicy
vmprop
javax.security.jacc.PolicyConfigurationFactory.provider=com.borland.security.jacc.provider.BESPolicyConfigurationFactory
```

4 必要に応じて、プロパティを設定します。

メモ このプロパティを空白のままにしておくと、JACC プロバイダが有効にならないため、システムは以前の AppServer のリリースと同じセキュリティフレームワークにフォールバックします。

JACC プロバイダの有効化と無効化

次のいずれかを使用することができます。

- AppServer セキュリティを JACC プロバイダとして設定する（デフォルトの設定）
- AppServer セキュリティで JACC を無効にする - 基盤となるセキュリティメカニズムは、以前の AppServer のリリースと同じ
- 外部 JACC プロバイダを使用するように AppServer を設定する

デフォルトでは、AppServer をインストールすると Borland VisiSecure が JACC プロバイダとしてインストールされます。AppServer に同梱されている JACC プロバイダは、すべての JACC API と互換性があり、JACC 仕様で指定されている Provider Configuration サブコントラクトを実装します。

AppServer の管理コンソールのセキュリティプロパティは、AppServer セキュリティを JACC API で使用できるようにデフォルトで設定されています。AppServer セキュリティプロバイダで JACC を使用しない場合は、管理コンソールのセキュリティプロパティをオフにしておく必要があります。

サードパーティの JACC ベースのセキュリティプロバイダを AppServer にプラグインすることで、セキュリティ基盤を拡張することもできます。外部のプロバイダを使用する場合は、［Partition Properties］ダイアログボックスの［JACC Properties］ボックスに、適切なプロパティの値を入力する必要があります。また、外部 JACC プロバイダ関連の jar ファイルをライブラリモジュールとしてパーティションにデプロイメントしておく必要があります。

外部 JACC プロバイダの設定

JACC と互換性がある任意の外部プロバイダを AppServer にプラグインできます。このプロバイダの実装と設定は、次に示すガイドラインにしたがう必要があります。

- このプロバイダは、`java.security.Policy` の実装を提供する必要があります。また、前のセクションで説明したように、管理コンソールまたは設定ファイルによって正しく設定する必要があります。
- このプロバイダは、`PolicyConfigurationFactory` の実装を提供する必要があります。また、管理コンソールまたは設定ファイルによって正しく設定する必要があります。
- プロバイダに依存するすべての jar ファイルは、ライブラリモジュールとしてパーティションにデプロイメントする必要があります。

本製品には、プロバイダの正しい実装方法と BES を使用した設定を示す例が同梱されています。詳細は、`<install dir>/examples/security/jacc` を参照してください。

外部の JACC プロバイダは、Borland 管理コンソールを使用して設定できます。または、セキュリティのプロパティを `partition_server.config` ファイルで設定できます。

第 29 章

BAS での ADLoginModule の使用

Active Directory は、Windows プラットフォーム用の Microsoft のディレクトリサービスの実装です。これにより、ネットワーク環境を構成するディレクトリの ID、リソース、関係を管理することができます。ADLoginModule は、BAS にバンドルされている新しい LoginModule で、LDAPLoginModule の後継です。Active Directory 専用のバックエンドのユーザーストアとして動作します。

ADLoginModule のしくみ

ユーザープリンシパル名

LDAPLoginModule とは異なり、ADLoginModule は、デフォルトではユーザープリンシパル名（UPN）を使用して Active Directory サーバーにバインドし、認証を行います。UPN は、オブジェクト名と完全修飾ドメイン名（FQDN）を組み合わせて形成され、*objectname@FQDN* となります。たとえば、ドメイン `abc.def.net` のユーザー `user1` の場合は、ユーザープリンシパル名 `user1@abc.def.net` がセキュリティプリンシパルとして使用されます（LDAPLoginModule では DN）。

認証

認証処理には、2 つのステップがあります。

1 ユーザー名とパスワードのペアをユーザーのバックエンドストアに基づいて検証する

2 後のステップで認証に使用されるユーザーの属性を生成する

最初のステップで、ADLoginModule は、渡されたユーザー名とドメイン名からユーザープリンシパル名を形成します。ユーザーが指定したパスワードに基づいて、ADLoginModule は Active Directory にバインドされます。バインド操作の成功は、そのユーザーが Active Directory サーバーに認証されたことを意味します。

認証が成功すると、ADLoginModule は Active Directory からユーザーエントリの識別名（DN）を取得し、JAAS 設定で指定されたオプションから指定された属性のセットを生成します。そのために、ADLoginModule は SEARCHBASE コンテキストを検索し、フィルタ「userPrincipalName=*UPN*」を満たすエントリを探します。

入手した DN 情報を使用して、ADLoginModule は JAAS 設定で指定されたオプションに基づいてエントリの必要な属性を生成します。

ADLoginModule の設定

新しいオプション DOMAINNAME が ADLoginModule に追加されました。このオプションは、エントリが認証されるドメインを示します。サンプルの設定は、次のとおりです。

```
adrealm {
    com.borland.security.provider.authn.ADLoginModule required
    INITIALCONTEXTFACTORY=com.sun.jndi.ldap.LdapCtxFactory
    PROVIDERURL="ldap://testing.net"
    DOMAINNAME=abc.def.net
    SEARCHBASE="cn=users,dc=abc,dc=def,dc=net"
  };
```

この設定では、ユーザーの認証はホスト `testing.net` の Active Directory Server に基づいて、ドメイン `adc.def.net` に対して行われます。ユーザーエントリは、SEARCHBASE “cn=users,dc=abc,dc=def,dc=net” から検索されます。

詳細な設定オプション

LDAPLoginModule と同様に、ADLoginModule は、JAAS 設定ファイル内の次のエントリで設定できます。

```
<realm-name> {
     com.borland.security.provider.authn.ADLoginModule
        authentication-requirements-flag
     INITIALCONTEXTFACTORY=connection-factory-name
     PROVIDERURL=backend-url
     DOMAINNAME=[domain name as in DNS-mapped format, for example, abc.def.net]
     SEARCHBASE=search-start-point
     USERATTRIBUTES=attribute1, attribute2, ...
     USERNAMEATTRIBUTE=attribute
     QUERY=dynamic-query
  };
```

オプションの詳細について説明します。

プロパティ名	説明
INITIALCONTEXTFACTORY	JNDI が LDAP にバインドするために使用する InitialContextFactory クラスです。
PROVIDERURL	ディレクトリサーバーの URL。 形式は、ldap://<servername>:<port> です。 この属性は必須です。
DOMAINNAME	Active Directory の新しい属性です。ユーザーのドメイン名を示します。 必須ではありませんが、AD でログインを実行する場合には設定することをお勧めします。 DN を使用するログインでは、USERNAMEATTRIBUTE を「DN」に設定する必要があります。
SEARCHBASE	ディレクトリのルックアップのための検索ベースを明示的に設定します。 この属性は省略可能です。 指定しないと、ドメインのルートコンテキストから検索が実行されます。
USERATTRIBUTES	これは、 認証されたユーザーに対して取得および保存される属性のカンマで区切られたリストです。 この属性はオプションです。 指定しないと、 エントリのすべての属性が生成されます。 詳細は、『Security User Guide』の LDAPLoginModule を参照してください。
USERNAMEATTRIBUTE	CallbackHandler または IdentityWallet からシステムでユーザーを認証するときは、 名前とパスワードのペアが必要です。 この属性は、ドメインまたは DN 内のユーザー名の「名前」の意味を定義します。この属性はオプションです。指定しない場合（デフォルト）は、DOMAINNAME オプションを指定する必要があり、ユーザーの入力は、ドメイン内のユーザー名として扱われます。 UPN の形式は、 <username>@<domainname> です。 これに対して、 ログインで DN を使用する場合は、 このオプションを「DN」に設定する必要があります。 この場合、 ユーザーからの入力は、 直接 DN として処理されます。
QUERY	ディレクトリサーバーに対して動的にクエリを実行して他の情報を取得し、 結果を属性として表すためのメカニズムを提供します。 詳細は、『Security User Guide』の LDAPLoginModule を参照してください。 この属性は、 省略可能です。

第 30 章

JAXR の使い方

このドキュメントでは、Java API for XML Registries（JAXR）について説明します。JAXR は、J2EE 1.4 仕様の一部です。J2EE の開発者は、主に Web サービスで使用される各種の XML レジストリにアクセスするための共通の標準 API として、これを使用できます。Sun による JAXR 仕様は、http://java.sun.com/xml/jaxr/index.jsp にあります。

Borland AppServer（BAS）は、Apache jUDDI と Apache scout を統合して、UDDI レジストリと JAXR との互換性を提供します。Apache jUDDI は、Web サービス用の Universal Description, Discovery, and Integration（UDDI）仕様に基づくオープンソースの Java 実装です。

JAXR 仕様では、異なる機能レベルを持つ 2 種類のプロバイダが定義されています。各プロバイダは、2 つの一般的なレジストリ仕様である UDDI と ebXML とのやり取りを行うために、異なるレベルのサポートを提供しています。タイプ 0 のプロバイダは、UDDI レジストリへのアクセスをサポートし、タイプ 1 のプロバイダは UDDI と ebXML レジストリ両方へのアクセスをサポートします。

Apache scout は BAS と統合されており、タイプ 0 の jUDDI JAXR プロバイダです。これにより、jUDDI クライアントが標準 JAXR API に適応するようになります。

BAS での JAXR の使用

JAXR API を使用する前に、JVM を実行するためにクラスパスを設定し、システムのプロパティを設定する必要があります。`juddi.ear` を BAS パーティションにデプロイメントする必要があります。`juddi.ear` ファイルは、BAS リポジトリ `<BAS_home>/var/repository/archives/ears` にあります。

BAS パーティションが `juddi.ear` をホストするために必要な次のライブラリを含める必要があります。

- `<BAS_home>/lib/scout.jar`
- `<BAS_home>/lib/juddi.jar`
- `<BAS_home>/lib/axis/axis.jar`
- `<BAS_home>/lib/axis/commons-discovery-0.2.jar`

jar ファイルはライブラリとして J2EE アプリケーションに含めることができます（ear、jar、または war ファイル）。または、jar ファイルを静的ライブラリとして BAS パーティションにデプロイメントできます。

JAXR を Java クライアントアプリケーションで実行している場合、上に示したすべてのライブラリと、下に示すライブラリをクラスパスに含める必要があります。

- `<BAS_home>/lib/axis/commons-logging.jar`

- <BAS_home>/lib/axis/asrt.jar

システムプロパティ

JAXR プロバイダを UDDI で使用するには、最初に ConnectionFactory 実装クラスの名前を指定する必要があります。これには、システムプロパティ `javax.xml.registry.ConnectionFactoryClass` を `org.apache.ws.scout.registry.ConnectionFactoryImpl` に設定します。デフォルトでは、BAS パーティションはその JVM として、このプロパティを自動的に設定します。アプリケーションユーザーは、このプロパティを設定する必要はありません。JAXR をスタンドアロン java アプリケーションで実行している場合、このシステムプロパティは、JVM をポイントするように設定する必要があります。そのように指定しないと、デフォルト値の `com.sun.xml.registry.common.ConnectionFactoryImpl` が使用されますが、これは見つかりません。これにより、`ConnectionFactory.newInstance()` メソッドが呼び出されると `JAXRException` が発生します。UDDI の BAS JAXR プロバイダは、JNDI 経由での ConnectionFactory のルックアップをサポートしていません。

JAXR 接続プロパティ

接続固有のプロパティは、ファクトリから接続を取得する前に、ConnectionFactory に設定する必要があります。詳細なプロパティの一覧およびその説明については、JAXR 仕様を参照してください。次に、接続を取得するために必要なプロパティのサブセットを示します。

プロパティ	説明
javax.xml.registry.queryManagerURL	jUDDI レジストリの UDDI に対する照会 API の URL です。この URL の形式は、次のとおりです。http://<hostname>:<port>/juddi/inquiry. このプロパティは必須です。
javax.xml.registry.lifeCycleManagerURL	UDDI レジストリの UDDI に対する公開 API の URL です。この URL の形式は、次のとおりです。http://<hostname>:<port>/juddi/publish.
javax.xml.registry.authenticationMethod	レジストリを使用して認証を行うときの認証方法。この値は、`UDDI_GET_AUTHTOKEN` または `HTTP_BASIC` のいずれかです。何も指定しない場合、デフォルト値の `UDDI_GET_AUTHTOKEN` が使用されます。

BAS JAXR サンプルコード

次の例では、JAXR API を使用して接続を作成する方法を示します。

```
import javax.xml.registry.Connection;
import javax.xml.registry.ConnectionFactory;
import java.util.Properties;

public class TestConnection
{
   public static void main(String[] args)
   {
      Properties prop = new Properties();
      try
      {
         String queryurl = "http://localhost:8080/juddi/inquiry";
         prop.setProperty("javax.xml.registry.queryManagerURL", queryurl);
         prop.setProperty("javax.xml.registry.lifeCycleManagerURL", queryurl);
         ConnectionFactory factory = ConnectionFactory.newInstance();
         factory.setProperties(prop);
         Connection con = factory.createConnection();
         if(con == null)
            System.out.println("No Connection");
      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
   }
}
```

第 31 章

スケジューラサービスの使用

Borland AppServer 6.7（AppServer）は、J2EE 1.4 準拠の EJB タイマーサービスをサポートしています。AppServer では、このサービスをスケジューラサービスと言います。AppServer のスケジューラサービスは、Quartz に基づいています。EJB タイマーサービスの一般的な情報については、EJB 2.1 仕様を参照してください。Quartz 関連のドキュメントを入手するには、`http://www.opensymphony.com/quartz/documentation.action` を参照してください。

スケジューラサービスは、パーティションレベルのサービスです。つまり、パーティションを作成するたびに、それが自動的にパーティションサービスの 1 つとして含まれます。スケジューラサービスは、EJB コンテナがダウンしているときでも使用できます。

スケジューラサービスの設定

よく使用されるスケジューラサービスのプロパティの一部は、AppServer 管理コンソールで設定できます。それには、次の手順にしたがいます。

1. AppServer 管理コンソールを開きます。
2. 設定するスケジューラサービスがあるパーティションの名前をダブルクリックして、ノードを展開します。
3. パーティションの下の［Scheduler Service］ノードを右クリックします。
4. 表示されるメニューから、［Properties］を選択します。［Properties］ダイアログボックスが表示されます。
5. ［General］タブで、次のスケジューラサービスを設定します。

   ［Transaction Timeout］ - トランザクションは、ここで指定した時間内に成功する必要があります。このフィールドに設定された時間内にトランザクションが完了しなかった場合、そのトランザクションにはロールバックのマークが付けられます。

   ［Max Redelivery Count］ - スケジューライベントが含まれるトランザクションがロールバックされたアプリケーションに対して、スケジューラサービスがメッセージの再送信を試行する回数を指定します。

   ［Clean events on startup］ - このチェックボックスがオンになっていると、このパーティションが再起動されたときに、すべてのジョブおよびトリガーがデータベースから削除されます。これは、スケジューライベントを存続させるように JobStoreCMT を設定している場合にのみ適用されます。現在、このオプションは JDataStore でのみサポートされています。

   ［Soft Commit］ - ソフトコミットを有効にする場合は、このチェックボックスをオンにします。ソフトコミットを有効にすると、オペレーティングシステムのキャッシュは、

コミットされたトランザクションからファイルへの書き込みをバッファリングできます。ソフトコミットによってパフォーマンスは向上しますが、最後にコミットされたトランザクションの耐久性は保証されなくなります。

6 ［Quartz］タブをクリックして前面に表示します。

7 以下のプロパティを設定します。

［**Maximum number of threads**］- スレッドプール内のスレッドの最大数を指定します。

［**Job Store Type**］- ドロップダウンメニューのデフォルトの選択肢は、［Memory］です。この場合、スケジューライベントはメモリ内に格納されます。イベントをデータベースに永続化する場合は、メニューから［JDBC(CMT)］を選択します。

［Job Store Type］として［JDBC(CMT)］を選択した場合は、［Job Store］ボックスで次を設定する必要があります。

［**Database**］- ドロップダウンメニューからデータベースを選択します

［**Container Managed DataSource**］- コンテナ管理データソースの URL を指定します。［Container Managed DataSource］の詳細は、Quarts のドキュメントを参照してください。

［**Non Container Managed DataSource**］- コンテナ以外の管理データソースの URL を指定します。

8 詳細なプロパティを設定するには、［Advanced］ボタンをクリックします。［Scheduler (Quartz) Properties］ページが表示されます。ここでは、詳細なプロパティを設定できます。

JDataStore を使用したスケジューライベントの永続化

AppServer Scheduler Service は、あらゆるリレーショナルデータベースでデータを永続化するように設定できます。デフォルトでは、AppServer は JDataStore を使用して永続化を実現します。スケジューライベントを保存するデータベースを指定していない場合、AppServer はデフォルトでこれらのイベントを JDataStore データベースに保存します。

スケジューライベントを永続化するための他のデータベースの設定

デフォルトでは、パーティションの JDataStore データベースがスケジューラデータの永続化に使用されます。ただし、アプリケーションデータの永続化に使用しているデータベースをスケジューラデータの永続化に使用するために、別のデータベースを設定することもできます。JDataStore 以外のデータベースを使用する場合は、次の手順にしたがいます。

- そのデータベース用に Quartz が提供しているスクリプトを使用して、データベース内に適切なテーブルを作成します。これらのスクリプトは、Quartz のフットプリント内にあります。
- `<partition_working_directory>/adm/scheduler/bes.properties` にある Quartz の設定ファイルから、正しいデータベースドライバを選択します。

2PC 最適化のための設定

アプリケーション内でトランザクションにタイマーが結び付けられている場合、なんらかの理由でトランザクションがロールバックされると、タイマーの作成または削除もそのトランザクションとともにロールバックされます。同様に、EJB に送信されたスケジューライベントを含むトランザクションがその後ロールバックされると、スケジューラサービスはイベントの再送信を試みます。EJB 2.1 仕様では、少なくとも 1 回の再送信が試行されることになっています。スケジューラサービスが実行する再送信の試行回数は、設定可能です。デフォルトは 1 です。つまり、トランザクションがロールバックされると、AppServer 内のスケジューラサービスもメッセージを 1 回だけ再送信しようとします。再送信回数の上限を設定する方法については、255 ページの「スケジューラサービスの設定」のセクションを参照してください。

2PC の最適化を実現するには、共通のデータソースを使用してスケジューライベントを永続化し、J2EE アプリケーションが使用するアプリケーションデータを保存する必要があります。パーティション内に複数のアプリケーションがあり、それぞれ別のデータソースを使用している場合、各アプリケーションで 2PC の最適化を行うことはできません。同じデータソースをスケジューラサービスとして使用しているアプリケーションでのみ最適化できます。

一部のデプロイメントでは、2PC 対応の（XA）データソースを使用する必要があります。つまり、トランザクションが使用するデータソースとして `bes.properties` ファイル内で指定する JNDI 名は、DAR ファイル内で XA データソースをポイントしている必要があります。

メモ ロールバック操作などのトランザクション動作は、CMT に永続的ストレージを設定した場合にのみ使用できます。

スケジューラサービス用のパーティションサービスのプロパティ

Quartz は、パーティションの設定ファイル `partition.xml` の新しいサービスとして導入されました。次の表に、Quartz と統合した場合のパーティションサービスのプロパティをリストします。

プロパティ名	説明	デフォルト値
`lifecycle.class`	BES パーティションにより、動的に新しいサービスを追加できます。 追加されたサービスは、パーティションプロセスのライフサイクルにしたがいます。	`com.borland.jms.SchedulerPartitionService`
`properties.location`	設定ファイルの場所を指定します。	`<appserverInstallRoot>\var\domains\base\configurations\<configName>\mos\<partitionName>\adm\scheduler\bes.properties`
`sql.location`	データベース内にテーブルを作成するために使用する sql スクリプトの場所を指定します。	`<partition_dir>\adm\scheduler\tables_jdatastore.sql`
`scheduler.clean_persistent_data_on_startup`	パーティションの再起動時に、スケジューリングデータを保存しているデータベースをクリーンアップするかどうかを指定します。	`false`

プロパティ名	説明	デフォルト値
scheduler.database_softcommit	このプロパティは、 永続性を得るためのバッキングストアとして JDataStore を使用している場合にのみ意味があります。 このプロパティによってコミットプロセスのパフォーマンスは向上しますが、ごく一部の失敗の際の回復能力が犠牲になります。 詳細は、 JDataStore ドキュメントを参照してください。 このプロパティは、 AppServer JSS でも使用されます。	true
scheduler.transaction_timeout	トランザクションタイムアウト	タイムアウトはありません。タイムアウトになるまでの時間を秒数で指定して、デフォルト値を上書きできます。
scheduler.auto_create_tables	存在していない場合は、 自動的に Quartz テーブルを作成します。	true
scheduler.max_redelivery_count	トランザクションがロールバックした場合に、 スケジューラサービスがイベントを再送信する回数です。	1
scheduler.use_default_datasource	デフォルトのデータソースを使用するかどうかを指定します。 デフォルトのデータソースは、 JNDI URL が jdbc/quartz で、adm/scheduler/database/scheduler.jds にある JDataStore データベースをポイントします。	あり

AppServer で使用される Quartz のプロパティ

次の表に、AppServer のスケジューラサービスで使用される Quartz のプロパティを示します。これらのプロパティは、`<appserver-install>\var\domains\base\configurations\<configuration_name>\mos\<partition_name>\adm\scheduler\bes.properties` ファイルにリストされています。これらのプロパティの詳細な説明については、Quartz のドキュメントを参照してください。

プロパティ名	説明	デフォルト値
org.quartz.scheduler.instanceName	スケジューラの名前を指定します。	TestScheduler
org.quartz.scheduler.instanceId	スケジューラの ID を指定します。	AUTO
org.quartz.scheduler.wrapJobExecutionInUserTransaction	このプロパティを true に設定すると、 ジョブの実行を呼び出す前に UserTransaction が起動されます。 このトランザクションは、ジョブの実行メソッドが完了し、 JobDataMap が更新されてからコミットされます。	True
org.quartz.scheduler.userTransactionURL	アプリケーションサーバーの UserTransaction マネージャの JNDI URL を指定します。 これは、 JobStoreCMT とのみ併用されます。	java:comp/UserTransaction
org.quartz.threadPool.class	threadpool クラスを指定します。	org.quartz.simpl.SimpleThreadPool

プロパティ名	説明	デフォルト値
org.quartz.threadPool.threadCount	ジョブを同時実行できるスレッドの数を指定します。 適切な値は、1 ～ 100 です。	30
org.quartz.threadPool.threadPriority	スレッドの優先順位を指定します。 Thread.MIN_PRIORITY (1) と Thread.MAX_PRIORITY(10) との間の値を指定できます。	5
org.quartz.threadPool.makeThreadsDaemons	このプロパティを true に設定すると、プール内のスレッドはデーモンスレッドとして作成されます。	True
org.quartz.jobStore.class	JobStore クラスを指定します。 非永続的なジョブおよびトリガーの場合は、このプロパティを RAMJobStore に設定し、永続的なジョブおよびトリガーの場合は、JobStoreTx または JobStoreCMT に設定します。 JobStoreTx は、スタンドアロンのスケジューラサービス用です。AppServer でデータソースを管理する場合は、JobStoreCMT を使用します。	RAMJobStore（メモリ）。 現在、AppServer スケジューラサービスは、RAMJobStore と JobStoreCMT だけをサポートしています。
org.quartz.jobStore.driverDelegateClass	Oracle データベースの場合は org.quartz.impl.jdbcjobstore.oracle.OracleDelegate、JDataStore の場合は org.quartz.impl.jdbcjobstore.HSQLDBDelegate。	org.quartz.impl.jdbcjobstore.HSQLDBDelegate. 現在、AppServer のスケジューラサービスは、JdataStore と Oracle のデータベースだけをサポートしています
org.quartz.jobStore.dataSource	コンテナ管理トランザクション（CMT）のデータソースの名前を指定します。JobStoreCMT には、1 つの CMT データソースと 1 つの CMT 以外のデータソースが必要です。	myDS
org.quartz.jobStore.nonManagedTXDataSource	コンテナ以外が管理するトランザクションのデータソースの名前を指定します。	myDSNoTx
org.quartz.dataSource.NAME_CMT.jndiURL	CMT データソースの JNDI URL を指定します。 NAME_CMT は、CMT データソースの名前です。	jdbc/Quartz
org.quartz.dataSource.NAME_NOT_CMT.jndiURL	CMT 以外のデータソースの JNDI URL を指定します。NAME_NOT_CMT は、CMT 以外のデータソースの名前です。	jdbc/Quartz

クラスタリングのサポート

Borland AppServer は、スケジューラサービスのクラスタリングをサポートします。たとえば、条件が同じ 2 つのパーティションの両方でスケジューラサービスが有効になっているとします。これらに同じアプリケーションをデプロイメントし、片方のアプリケーションにタイマーを登録した場合、そのパーティションがダウンすると、両方のアプリケーションが同一のデータベースをポイントしていれば、複製がタイマーイベントを取得できます。AppServer のスケジューラサービスは、フェイルオーバーをサポートしています。

第 32 章

VisiConnect の概要

J2EE コネクタアーキテクチャ

情報技術環境では、エンタープライズアプリケーションは一般に企業情報システム（EIS）に関連する機能やデータを利用します。従来は、標準規格でない各ベンダー独自のアーキテクチャが使用されてきました。このため、複数のベンダーがかかわるとアーキテクチャの数も増え、エンタープライズアプリケーション環境が非常に複雑になりました。Java 2 Enterprise Edition（J2EE）1.4 プラットフォームや J2EE コネクタアーキテクチャ（コネクタ）1.5 規格の導入によって、この作業が大幅に簡素化されました。

VisiConnect は Borland によるコネクタ 1.5 規格のインプリメンテーションで、さまざまな EIS を Borland AppServer（AppServer）に統合するための簡潔な環境です。コネクタは、J2EE プラットフォームのアプリケーションサーバーと EIS を統合するためのソリューションを提供することにより、J2EE プラットフォームの利点である接続、トランザクション、およびセキュリティ基盤を活用できるようにして、EIS の統合という課題に対応しています。コネクタによって、EIS ベンダーはアプリケーションサーバーごとに自社のプラットフォームへ独自に統合する必要がなくなります。VisiConnect はコネクタに完全に適合しているため、EIS との統合のために AppServer 自体をカスタマイズする必要はありません。

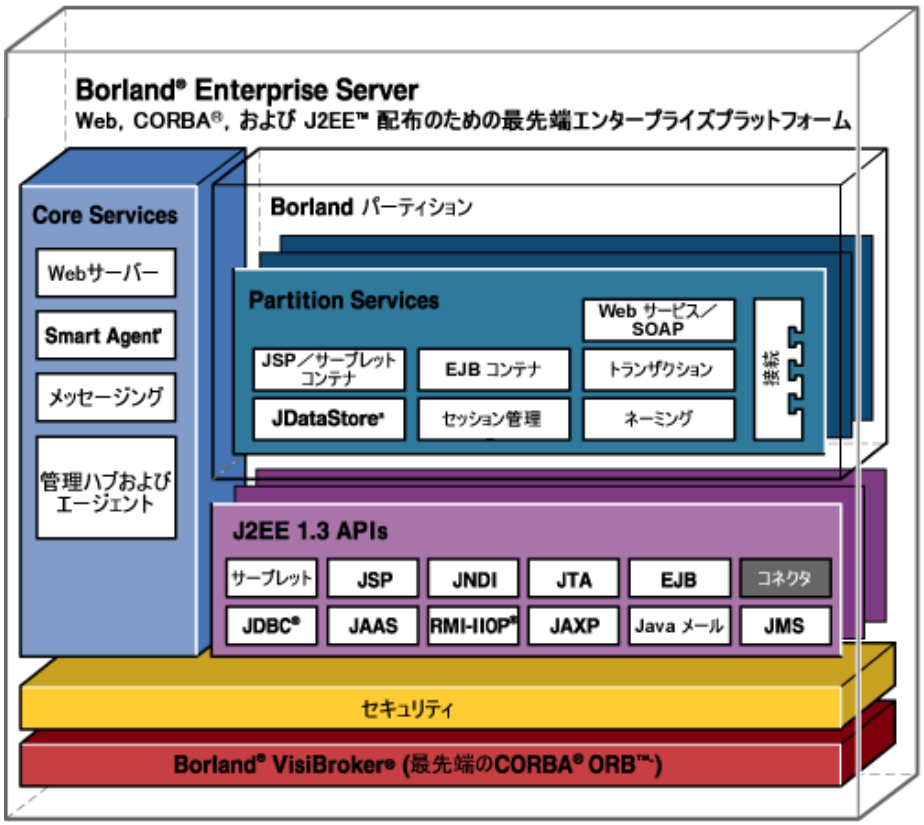
コネクタによって、EIS ベンダーは EIS 用として標準のリソースアダプタを提供すればよくなります。AppServer にデプロイメントしたリソースアダプタは、それぞれが EIS と AppServer との統合のインプリメンテーションとなります。VisiConnect によって、Borland Enterprise Server では異種 EIS へのアクセスが可能になります。この結果、EIS ベンダーはコネクタに準拠した標準のリソースアダプタを 1 つ提供するだけで済みます。こうしたリソースアダプタは、デフォルトで AppServer にデプロイメントされるようになっています。

コンポーネント

コネクタ環境は、アプリケーションサーバーでのコネクタのインプリメンテーションと EIS 固有のリソースアダプタという 2 つの主要コンポーネントで構成されています。

J2EE 1.4 アーキテクチャでは、コネクタは J2EE コンテナを拡張したもので、アプリケーションサーバーとも呼ばれます。J2EE 1.4 プラットフォームとコネクタ 1.5 仕様に適合している VisiConnect は、AppServer の拡張機能であり、サービスではありません。次の図は、AppServer アーキテクチャでの VisiConnect を示しています。

図 32.1　AppServer での VisiConnect



上の図で、VisiConnect は、「コネクタ」というモジュールで表されています。

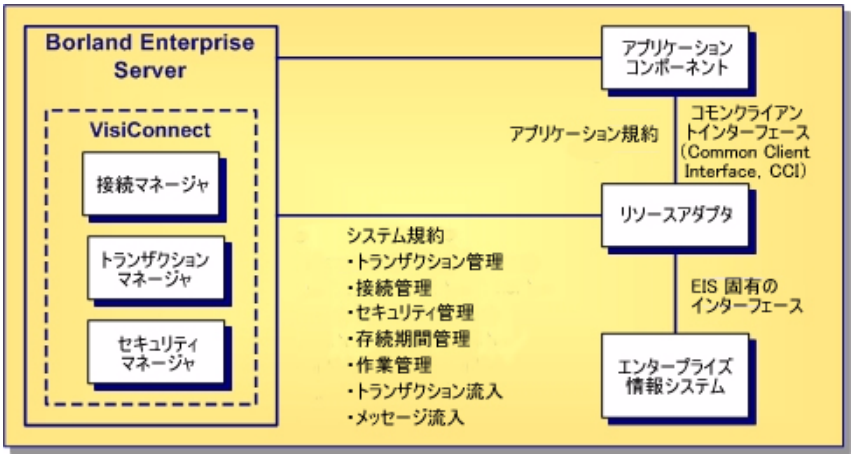
リソースアダプタは、EIS 固有のシステムレベルのドライバで、EIS へのアクセスを可能にします。簡単に言えば、リソースアダプタとは JDBC ドライバのようなものです。リソースアダプタと EIS との間のインターフェースは、EIS によってさまざまです。Java インターフェースの場合もあれば、ネイティブのインターフェースの場合もあります。

コネクタは、次の 3 つの主要コンポーネントで構成されています。

- **システム協定**。リソースアダプタとアプリケーションサーバー（AppServer）を統合します。
- **コモンクライアントインターフェース**。Java アプリケーション、フレームワーク、および開発ツールがリソースアダプタとやり取りできるようにするための標準クライアント API です。
- **パッケージングとデプロイメント**。各種のリソースアダプタをモジュール形式で J2EE アプリケーションへ組み込めるようにします。

次の図はコネクタアーキテクチャのしくみを示しています。

図 32.2 コネクタアーキテクチャ



リソースアダプタとその付属アイテムがコネクタとして機能します。VisiConnect は、EIS ベンダーやサードパーティアプリケーション開発会社がコネクタ 1.5 規格にしたがって作成したリソースアダプタをサポートしています。リソースアダプタには、特定の EIS と対話するために必要なコンポーネント（Java コードと、必要であればネイティブのコード）を含みます。

システム規約

コネクタ仕様では、アプリケーションサーバーと EIS 固有のリソースアダプタとの間の一連のシステムレベル協定を規定しています。この協定によって、システムレベルのしくみのすべてがアプリケーションコンポーネントから透過になります。したがって、アプリケーションコンポーネントプロバイダは、ビジネスとプレゼンテーションのロジックの開発に専念でき、EIS に関連したシステムレベルの問題を意識する必要がなくなります。これにより、アプリケーションコンポーネントの開発が容易になり、保守もしやすくなります。

コネクタ仕様に対応して、VisiConnect は次の機能について定義されている協定の標準セットを実装しています。

- **接続管理**。基底の EIS への接続をアプリケーションサーバーがプールすることにより、アプリケーションコンポーネントに EIS への接続サービスを提供します。これにより、高度に拡張可能なアプリケーション環境で、異種 EIS へのアクセスを必要とする多くのクライアントをサポートすることができます。
- **トランザクション管理**。アプリケーションサーバーのトランザクションマネージャと、EIS リソースマネージャへのトランザクションアクセスをサポートする EIS との協定によって、アプリケーションサーバーが複数のリソースマネージャのトランザクションを管理できます。
- **セキュリティ管理**。基底の EIS に安全にアクセスできます。安全なアプリケーション環境を実現して、EIS へのセキュリティ上の脅威を減らし、EIS が管理する貴重な情報リソースを守ります。
- **存続期間管理** により、アプリケーションサーバーはリソースアダプタの存続期間を管理できます。このコントラクトによってアプリケーションサーバーは、デプロイメント時またはアプリケーションサーバーの起動時に、リソースアダプタインスタンスをブートストラップできます。また、デプロイメント解除時またはアプリケーションサーバーの正常なシャットダウン時には、リソースアダプタインスタンスに通知できます。

- **作業管理** により、リソースアダプタは、実行する `Work` インスタンスをアプリケーションサーバーに送信することにより、作業（ネットワークエンドポイントの監視、アプリケーションコンポーネントの呼び出しなど）を実行できます。アプリケーションサーバーはスレッドをディスパッチして、送信された `Work` インスタンスを実行します。これにより、リソースアダプタがスレッドを直接作成または管理することを防止でき、アプリケーションサーバーがプールスレッドを効率よく管理でき、また、実行時環境を細かく制御できます。リソースアダプタは、`Work` インスタンスが実行されるセキュリティコンテキストとトランザクションコンテキストを管理できます。
- **トランザクション流入** により、リソースアダプタは、インポートされたトランザクションをアプリケーションサーバーに伝達できます。このコントラクトにより、リソースアダプタはトランザクションの完了を転送でき、EIS によるリカバリ呼び出しをクラッシュし、インポートされたトランザクションの ACID プロパティを保存できます。
- **メッセージ流入** により、リソースアダプタは、メッセージの配信に使用される特定のメッセージングスタイル、メッセージングセマンティクス、メッセージングインフラストラクチャに関係なく、アプリケーションサーバー内に存在するメッセージのエンドポイントにメッセージを非同期で配信できます。このコントラクトは、標準のメッセージプロバイダ接続性コントラクトとしても使用されます。リソースアダプタを介して、Java メッセージサービス（JMS）、Java API for XML Messaging（JAXM）などの幅広いメッセージプロバイダを J2EE と互換性があるアプリケーションサーバーに組み込めます。

接続管理

EIS への接続は、作成にも破棄にもコストがかかるリソースです。アプリケーションの拡張可能性を確保するためには、Borland Enterprise Server は基底の EIS への接続をプールできる必要があります。アプリケーションコンポーネントの開発を簡略にするには、基底の EIS にアクセスするコンポーネントから見てこの接続プール機構が透過である必要があります。

コネクタ仕様では接続のプールや管理を定めて、アプリケーションコンポーネントの性能と拡張性を最適にしています。Borland Enterprise Server とリソースアダプタとを定義する接続管理協定は、次のものを提供します。

- 管理（n 層）アプリケーションと非管理（2 層）アプリケーションの両方の接続を取得するための一貫性のあるアプリケーション開発モデル。
- 基底の EIS のインプリメンテーションに対して不透過な、コモンクライアントインターフェース（CCI）に基づく標準接続ファクトリと接続インターフェースを提供するリソースアダプタへのフレームワーク。
- 設定された一連のリソースアダプタに、高度な接続プール、トランザクション管理、セキュリティ管理、エラー検索、ログなどのさまざまな QoS（Quality of Service）を提供するための共通のメカニズム。
- アプリケーションサーバーでの接続プール機能の実装。

VisiConnect は、次の目的のために接続管理を使用します。

- EIS に新しい接続を作成する。
- JNDI（Java Naming and Directory Interface）の名前空間に接続ファクトリを設定する。
- EIS への正しい接続をプールされた既存の接続セットから探し、それを再利用する。
- AppServer のトランザクションサービスとセキュリティサービスに連結する。

VisiConnect を使用して、AppServer は EIS への接続の確立、設定、キャッシュ、再利用を自動的に行います。

アプリケーションコンポーネントは、基底の EIS への接続を取得するために、接続ファクトリを使用して、JNDI 名前空間でリソースアダプタ接続ファクトリを探します。接続ファクトリは、VisiConnect の接続マネージャインスタンスに接続作成要求を委任します。この要求を受け取ると、接続マネージャは接続プールで検索を実行します。接続要求の条件に合う接続がプールにない場合、VisiConnect は、基底の EIS への新しい物理接続を作成するためにリソースアダプタで実装されている `ManagedConnectionFactory` を使用します。適合する接続がプールにある場合、VisiConnect は適合する `ManagedConnection` インスタンスを使用して、接続要求に対応します。新規の `ManagedConnection` インスタンスが作成されると、サーバーはこの `ManagedConnection` インスタンスを接続プールに追加します。

VisiConnect は、`ConnectionEventListener` に `ManagedConnection` インスタンスを登録します。このリスナーによって、VisiConnect は ManagedConnection インスタンスの状態に関連するイベント通知を受けることができます。VisiConnect はこれらの通知を使用して、接続のプール、トランザクション、および接続の終了処理を管理したり、エラー状況に対応します。

VisiConnect は `ManagedConnection` インスタンスを使用して、アプリケーションレベルで基底の物理接続のハンドルの役割を果たす Connection インスタンスをアプリケーションコンポーネントに提供します。この結果、コンポーネントは、基底の物理接続を直接使用するのではなく、このハンドルを使って EIS リソースにアクセスします。

トランザクション管理

複数の EIS へのトランザクションアクセスは、エンタープライズアプリケーションにとって大切で、場合によっては不可欠な要件です。コネクタは、複数の異種 EIS へのトランザクションアクセスをサポートします。データの一貫性と完全性を維持するために、多くの対話をまとめてコミットするか、まったくしないのどちらかにする必要があります。

VisiConnect は AppServer のトランザクションマネージャを使ってリソースアダプタをサポートし、次のトランザクションサポートレベルに対応しています。

- **トランザクションサポートなし**：リソースアダプタがローカルトランザクションも XA トランザクションもサポートしていない場合は、トランザクションに対応できません。アプリケーションコンポーネントがトランザクション非対応のリソースアダプタを使用している場合、そのアプリケーションコンポーネントは、トランザクションにおいて EIS とのどのような接続も利用してはなりません。アプリケーションコンポーネントがトランザクションにおいて EIS 接続を必要とする場合、アプリケーションコンポーネントは、ローカルトランザクションまたは XA トランザクションをサポートするリソースアダプタを使用する必要があります。
- **ローカルトランザクションサポート**：アプリケーションサーバーは、リソースアダプタにとってローカルとなっているリソースを直接管理します。XA トランザクションとは異なり、ローカルトランザクションは、2 フェーズコミット（2PC）プロトコルに関与することも、分散トランザクション（トランザクションコンテキストを単に伝達する）としても関与することもできません。ローカルトランザクションは 1 フェーズコミット（1PC）最適化だけを対象とします。リソースアダプタは、自身の Sun 標準のデプロイメントデスクリプタの中で、トランザクションサポートの種類を定義します。アプリケーションコンポーネントがトランザクションの一部として EIS 接続を要求する場合、AppServer は、現在のトランザクションコンテキストに基づいてローカルトランザクションを開始します。アプリケーションが接続を閉じると、AppServer はローカルトランザクションをコミットし、トランザクションが完了したら EIS 接続を除去します。
- **XA トランザクションサポート**：トランザクションは、リソースアダプタと EIS の外部にあるトランザクションマネージャによって管理されます。Sun 標準のデプロイメントデスクリプタの中で、リソースアダプタによるトランザクションサポートの種類を指定します。アプリケーションコンポーネントがトランザクションの一部として EIS 接続要求を切り分けるとき、AppServer はトランザクションマネージャに XA リソースを登録します。アプリケーションコンポーネントが接続を閉じるときに、AppServer がトランザクションマネージャから XA リソースの登録を解除し、トランザクションが完了したら EIS 接続の終了処理を行います。

  コネクタ 1.5 仕様に準拠しているため、VisiConnect は、上記の 3 つのトランザクションレベルのいずれにも完全に対応しています。

1 フェーズコミットの最適化

多くの場合、1 つのトランザクションはその適用範囲が 1 つの EIS に限定されており、EIS リソースマネージャは独自のトランザクション管理を行います。これがローカルトランザクションです。XA トランザクションは、複数のリソースマネージャにわたることが可能です。このため、外部トランザクションマネージャ（通常 Borland Enterprise Server にパッケージされたトランザクションマネージャ）がトランザクションの調整を実行する必要があります。この外部トランザクションマネージャは、複数の EIS にまたがるトランザクションを管理するために、2 フェーズコミット（2PC）プロトコルを使用したり、トランザクションコンテキストを分散トランザクションとして伝達することができます。XA トランザクションに 1 つのリソースマネージャだけが関与している場合は、1 フェーズコミット（1PC）プロトコルを使用します。単体のリソースマネージャが自身のトランザクション管理を扱っている環境では、1PC XA トランザクションと比較してコストが小さいリソースを扱うため、1PC 最適化が実行可能です。

セキュリティ管理

コネクタ 1.5 仕様への準拠の中で、VisiConnect はコンテナ管理のサインオンとコンポーネント管理のサインオンの両方をサポートします。実行時に、VisiConnect は起動コンポーネントのデプロイメントデスクリプタで指定された情報を基に選択されたサインオン機構を判別します。コンポーネントによって要求されたサインオンメカニズムを VisiConnect が判別できない場合（通常、リソースアダプタの接続ファクトリの不適正な JNDI 検索の実行によって）、VisiConnect はコンテナ管理のサインオンを試みます。コンポーネントが明示的なセキュリティ情報を指定した場合、コンテナ管理のサインオンの場合であっても、この情報は接続を取得するための呼び出し時に提示されます。

コンポーネント管理のサインオン

コンポーネント管理のサインオンを使用する場合、コンポーネントは EIS への接続の取得を要求をするときに、必要なすべてのセキュリティ情報（通常、ユーザー名とパスワード）を提供します。Borland Enterprise Server は、接続の要求とともにセキュリティ情報を転送すること以外に追加のセキュリティ処理は行いません。リソースアダプタは、インプリメンテーション固有の方法で EIS サインオンを実行するために、コンポーネントが提供するセキュリティ情報を使用します。

コンテナ管理のサインオン

コンテナ管理のサインオンを使用する場合、コンポーネントはどのようなセキュリティ情報も提示しないため、コンテナは接続を取得する要求において、必要なサインオン情報を判別し、この情報をリソースアダプタに提供する必要があります。コンテナは適切なリソース方針を判別し、リソース方針情報を JAAS（Java Authentication and Authorization Service）の Sbject オブジェクトの形式でリソースアダプタに提供する必要があります。

EIS 管理のサインオン

EIS 管理のサインオンを使用する場合、リソースアダプタは設定済みの固定された 1 組のセキュリティ情報によって、すべての EIS 接続を内部で取得します。この場合、起動コンポーネントによる新規の接続の要求において、リソースアダプタは自身に渡されたセキュリティ情報に依存しません。

認証メカニズム

AppServer のユーザーは、保護されている AppServer リソースにアクセスする際は必ず、認証を受ける必要があります。このため、各ユーザーは、認証情報（ユーザー名とパスワードのペア、またはデジタル証明書）を AppServer に提供する必要があります。AppServer では、次の種類の認証メカニズムがサポートされています。

- パスワード認証。ユーザー ID とパスワードが要求され、クリアテキスト形式で App に送信されます。App は情報をチェックし、信頼できる場合は、保護されているリソースへのアクセスを許可します。
- SSL（または HTTPS）プロトコルを使用すると、パスワード認証より高いセキュリティレベルを提供できます。SSL プロトコルはクライアントと AppServer 間で転送されるデータを暗号化するため、ユーザーのユーザー ID とパスワードがクリアテキスト形式で送信されることはありません。したがって、AppServer は、ユーザーの ID とパスワードの機密性を損なわずにユーザーを認証できます。
- 証明書認証。SSL または HTTPS クライアント要求が開始されると、AppServer は、クライアントにデジタル証明書を提示して応答します。クライアントはデジタル証明書を検証し、SSL 接続が確立されます。CertAuthenticator クラスは、クライアントのデジタル証明書からデータを抽出してその証明書を所有している AppServer ユーザーを特定し、最後に AppServer セキュリティ領域から認証されたユーザーを取得します。
- また、相互認証も使用できます。この場合、AppServer は自身を認証するだけでなく、要求側のクライアントの認証も要求します。クライアントは、信頼できる証明機関によって発行されたデジタル証明書を送信するように求められます。相互認証は、信頼できるクライアントだけにアクセスを制限する必要がある場合に便利です。たとえば、提供したデジタル証明書を持つクライアントだけを受け入れることにより、アクセスを制限できます。

詳細については、開発者ガイドの「セキュリティの概要」を参照してください。

セキュリティマップ

コネクタ 1.5 仕様の 8.5 節では、サインオンの実行を委任するリソースプリンシパルを定義するためのさまざまなオプションが規定されています。VisiConnect は、仕様で規定されているプリンシパルのマッピングオプションを実装しています。

このオプションでは、リソース方針は、起動コンポーネントの開始呼び出し方針の ID からのマッピングによって判別されます。判別されたリソース方針は、マッピング元のプリンシパルのセキュリティ属性の ID を継承しません。かわりに、リソースプリンシパルは定義されたマッピングを基に ID とセキュリティ属性を取得します。したがって、コンテナ管理のサインオンを有効にして使用するために、VisiConnect では、リソースプリンシパルと開始プリンシパルの関連付けを指定するためのセキュリティマップが提供されています。また、このモデルを拡張して、VisiConnect では、開始呼び出しロールをリソースロールにマッピングするためのメカニズムが提供されています。

コンポーネントがコンテナ管理のサインオンを要求したときに、デプロイメントリソースアダプタにセキュリティマップが設定されていなかった場合は、null JAAS Subject オブジェクトを使って接続の取得が試みられます。これは、リソースアダプタのインプリメンテーションを基にサポートされます。

定義済みの接続管理システム協調で AppServer とリソースアダプタの間でどのようにセキュリティ情報を交換するかが定義されている一方で、コンテナ管理のサインオンとコンポーネント管理のサインオンのどちらを使用するかは、接続を要求するコンポーネント用に定義されたデプロイメント情報を基に判定されます。

セキュリティマップは、ra-borland.xml デプロイメントデスクリプタの security-map 要素で指定します。この要素には、開始ロールとリソースロールとの関連付けを定義します。各 security-map 要素は、リソースアダプタと EIS サインオンの処理のために、適切なリソースロール値を定義するしくみを提供します。security-map 要素は、管理された接続や接続ハンドルを割り当てる際に使用される定義済みの開始ロールセットとそれに対応するリソースロールを指定する手段を提供します。

デフォルトのリソースロールは、security-map 要素で接続ファクトリに定義できます。それには、user-role の値に「*」を指定し、それに対応する resource-role 値を指定します。セキュリティマップ内で現在の ID と一致するものがない場合、定義した resource-role が常に利用されます。

これは省略可能な要素です。ただし、コンテナ管理のサインオンがリソースアダプタによってサポートされており、いずれかのコンポーネントがそれを使用する場合は、何らかの形式で指定する必要があります。また、デプロイメント時に接続プールに取り込む試みは、定義済みのデフォルトのリソースロールが指定されている場合はこれを使って行われます。

セキュリティポリシー処理

コネクタ 1.5 仕様では、アプリケーションサーバーで実行するリソースアダプタのデフォルトのセキュリティポリシーが定義されています。また、デフォルトのセキュリティポリシーを上書きする独自のセキュリティポリシーをリソースアダプタが提供するための方法も定義されています。

この仕様に対応して、AppServer は、リソースアダプタの実行時環境を動的に変更します。リソースアダプタで特定のセキュリティポリシーが定義されていない場合、Borland Enterprise Server は、リソースアダプタの実行時環境をコネクタ 1.5 仕様で指定されているデフォルトのセキュリティポリシーで上書きします。リソースアダプタで特定のセキュリティポリシーが定義されている場合、AppServer はまず、リソースアダプタのデフォルトのセキュリティポリシーとリソースアダプタで定義されている特定のポリシーの組合せでリソースアダプタの実行時環境を上書きします。リソースアダプタは、ra.xml デプロイメントデスクリプタファイルの security-permission-spec 要素を使って特定のセキュリティポリシーを定義します。

セキュリティポリシー処理の要件については、コネクタ 1.5 仕様（http://java.sun.com/j2ee/download.html#connectorspec）のセクション 18.2「Security Permissions」を参照してください。

コモンクライアントインターフェース（Common Client Interface、CCI）

CCI は、アプリケーションコンポーネント用の標準クライアント API を定義しています。CCI を使用すると、アプリケーションコンポーネント、エンタープライズアプリケーション統合（EAI）フレームワーク、および開発ツールが共通クライアント API を利用して異種 EIS でやり取りできます。

CCI は、EAI およびエンタープライズツールベンダーによる使用を目的としています。コネクタ 1.5 仕様は、CCI をほとんどのアプリケーション開発者が使用するアプリケーションレベルのプログラミングインターフェースとしてではなく、ツールベンダーが提供するより豊富な機能を実現するための基盤とすることを推奨しています。アプリケーションコンポーネント自体が API に書き込むことも可能です。CCI は低レベルのインターフェースなので、通常、既存のモジュールを J2EE 1.4 プラットフォームに移行するために使用されます。CCI を使用すると、従来の EIS クライアントを AppServer に直接統合できるため、コストをかけずにスムーズに J2EE 1.4 に移行できます。

CCI は、EIS に対する関数の実行と結果取得に焦点を当てたリモート関数呼び出しインターフェースを定義します。CCI は、特定の EIS に依存しません。つまり、特定の EIS のデータ型、呼び出しのフック、署名にバインドされていません。CCI は、リポジトリの EIS 固有のメタデータで駆動できます。

CCI によって、AppServer は EIS への接続の作成と管理や対話を実行したり、入力、出力、または戻り値のデータレコードを管理することができます。CCI は、Java Bean アーキテクチャや Java Collection フレームワークを活用するために設計されています。

コネクタ 1.5 仕様は、リソースアダプタが CCI をクライアント API としてサポートすることを推奨している一方で、リソースアダプタにシステム協定を実装することを義務付けています。開発者は、次のような CCI 以外のクライアント API を提供するリソースアダプタを開発することもできます。

- Java Database Connectivity（JDBC）API（一般的な EIS 型のインターフェースの例）。
- IBM CICS Java Gateway に基づくクライアント API（EIS 固有のインターフェースの例）。

アプリケーション規約を形成する CCI は、以下で構成されます。

- **ConnectionFactory**。ConnectionFactory インプリメンテーションは、EIS とやり取りするための手段として Connection オブジェクトと Interaction オブジェクトを作成します。ConnectionFactory の getConnection メソッドが EIS インスタンスへの接続を取得します。

- **Connection**。Connection インプリメンテーションは、EIS インスタンスへのアプリケーションレベルのハンドルを表します。実際の接続は、ManagedConnection によって表されます。アプリケーションは、ConnectionFactory オブジェクトの getConnection メソッドを使って Connection オブジェクトを取得します。
- **Interaction**。Interaction インプリメンテーションは、特定の対話を実行します。ConnectionFactory を使って作成されます。Interaction インプリメンテーションを通して対話を実行するには、具体的な対話の特性を特定する InteractionSpec、およびやり取りされるデータを運ぶ Input と Output の 3 つの引数が必要です。
- **InteractionSpec**。InteractionSpec インプリメンテーションは、コネクタの対話関連のプロパティ（呼び出すプログラム名、対話モードなど）をすべて定義します。InteractionSpec は、特定の対話が実行されるときに Interaction インプリメンテーションに引数として渡されます。
- **Input と Output**。Input と Output はレコードです。

レコードは、実際のレコードバイトを型と組み合わせたアプリケーションのデータ要素の論理的集合です。例として、COBOL や C データ構造があります。CCI では、Record インプリメンテーションはストリームを使用します。javax.resource.cci.Streamable インターフェースでは、ストリームの読み書きが read および write メソッドによって処理されます。javax.resource.cci.Record インターフェースでは、getRecordName() と getRecordShortDescription()、および setRecordName() と setRecordShortDescription() が、それぞれレコードデータの取得および設定を行います。

再利用する EIS 機能によって外部化されるすべてのデータ構造についてレコードを作成する必要があります。次に、リソースアダプタを通して EIS とデータをやり取りする input および output オブジェクトとしてレコードを使用します。レコードを作成する際は、次のオプションを利用できます。

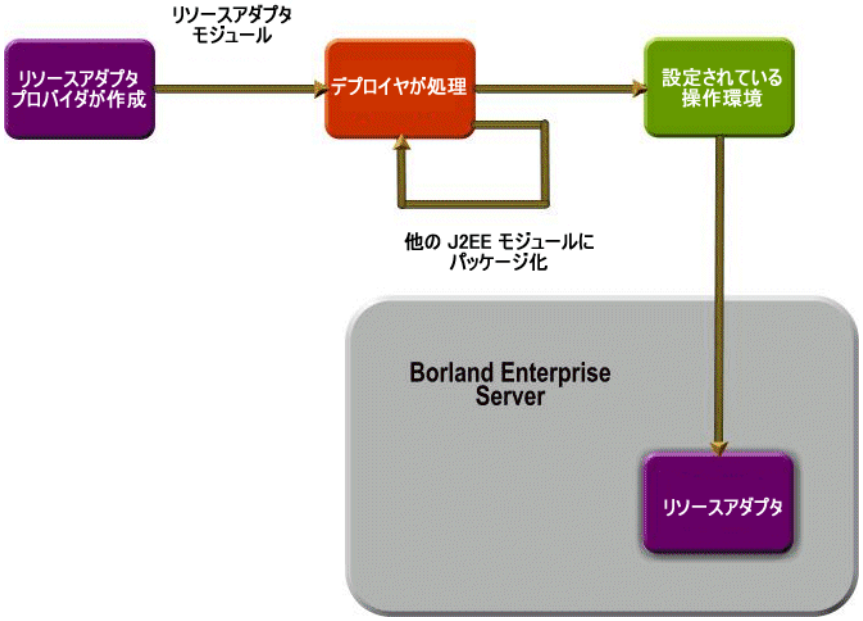
- **ネストしたレコード、または階層構造のレコードへの直接アクセス**。ユーザーによっては、直接的なまたは「フラットな」アクセッサメソッドの方が便利、あるいは自然に感じる場合があります。たとえば、COBOL を熟知しているプログラマは、フィールド名がレコード内で一意である場合、サブレコードのフィールドを直接参照したい場合があります。これは、COBOL のフィールド名がスコープされる方法に似ています。フィールド名が一意の場合、フィールド名を限定する必要はありません。
- **カスタムレコードと動的レコード**。通常、カスタムレコードと動的レコードの 2 種類のレコードを作成できます。これらのレコードの主な相違点は、フィールドへのアクセス方法です。動的レコードでは、フィールド名を指定してフィールドを見つけ、情報のオフセットとマーシャリングを検索した後、フィールドにアクセスします。カスタムレコードでは、情報のオフセットとマーシャリングがコード内にあるため、より迅速にアクセスできます。カスタムレコードを生成すると、コードの効率がよくなりますが、使用が制限されます。
- **通知ありまたは通知なしのレコード**。レコードを通知付きで作成すると、レコードのプロパティはバインドされます。

メモ プロパティをバインドする必要がない場合は、通知なしでレコードを作成すると、効率がよくなります。

パッケージとデプロイメント

さまざまなリソースアダプタを AppServer などの J2EE 1.4 プラットフォーム対応のアプリケーションサーバーにデプロイメントできるように、コネクタはパッケージングとデプロイメントのインターフェースを提供します。

図 32.3 AppServer と VisiConnect でのパッケージとデプロイメント



リソースアダプタは Java のインターフェースとクラスのセットをパッケージし、これによってコネクタ指定のシステム協定とリソースアダプタが提供する EIS 固有の機能が実装されます。リソースアダプタでは、基底の EIS に固有のネイティブライブラリや次のような付属アイテムの使用を必要条件にすることも可能です。

- マニュアル
- ヘルプファイル
- EJB のコードジェネレータ
- EIS を直接設定できるように設定ユーティリティを直接提供するツール
- リモートのリソースアダプタコンポーネントに追加のデプロイメント機能を提供するツール
- たとえば、IBM CICS で、メインフレームで実行するために必要になる場合がある JCL スクリプトのセット

Java のインターフェースとクラスは、リソースアダプタモジュールを作成するために必要な付属アイテムとデプロイメントデスクリプタでパッケージされます。デプロイメントデスクリプタは、リソースアダプタと AppServer との間のデプロイメント協定を定義します。

リソースアダプタは、共有のスタンドアロンモジュールまたは J2EE アプリケーションの一部としてパッケージしてデプロイメント可能です。デプロイメントの際、リソースアダプタモジュールは、AppServer にインストールされ、インストール先の操作環境に合わせて設定されます。リソースアダプタの設定は、デプロイメントデスクリプタに定義されたプロパティに基づいて行われます。

VisiConnect の機能

VisiConnect では、コネクタ規格の拡張機能として、次のような付加価値が高い機能が提供されています。

- VisiConnect パーティションサービス
- クラスローディングの追加サポート
- セキュリティで保護されたパスワード認証情報ストレージ
- 接続リークの検出
- ra.xml 仕様のセキュリティポリシー処理

VisiConnect パーティションサービス

VisiConnect サービス対応の Borland パーティションは、リソースアダプタをバンドルする J2EE アプリケーション、またはスタンドアロンのリソースアダプタコンポーネントの開発とデプロイメントをサポートように設計されています。AppServer パーティションは、統合された VisiConnect サービスを提供します。ツールには、デプロイメントデスクリプタエディタ（DDE）と、リソースアダプタとその関連デスクリプタファイルをパッケージングおよびデプロイメントするタスクウィザードが含まれています。

これにより、VisiConnect を実行するための高度なモジュール環境が提供されます。AppServer は、デプロイメント用のパーティションにデフォルトの VisiConnect サービスを提供します。

クラスローディングの追加サポート

VisiConnect は、リソースアダプタの Manifest.mf ファイルの ClassPath エントリで指定されているプロパティまたはクラスのロードをサポートします。リソースアダプタ内にあり、リソースアダプタによって使用されるプロパティとクラスを設定する方法を次に説明します。

リソースアダプタ（RAR）アーカイブファイルとそれを使用するアプリケーションコンポーネント（たとえば、EJB JAR）は、エンタープライズアプリケーション（EAR）アーカイブに含まれます。RAR には、JAR ファイルに格納されている Java プロパティのようなリソースが必要ですが、その JAR ファイルは、RAR 自体ではなく、EAR ファイルに含まれます。

RAR Java クラスへのリファレンスを指定するには、RAR Manifest.mf ファイルに ClassPath= エントリを追加します。また、EJB Java クラスを EAR 内にある同じ JAR ファイルに格納することもできます。このようにすると、Java クラスが必要な EAR 内のコンポーネントのために、Java クラスを含む「サポート」JAR ファイルを提供できます。

セキュリティで保護されたパスワード認証情報ストレージ

VisiConnect では、リソースアダプタデプロイヤがセキュリティで保護されたパスワード認証情報ストレージを利用して、特定の承認／認証メカニズムを組み込むための標準メソッドが提供されています。

このストレージメカニズムを使用して、ユーザーロール（AppServer のロール。AppServer のユーザー名とパスワードの組合せまたは認証情報に関連付けられる）をリソースロール（EIS のロール。EIS のユーザー名とパスワードの組合せまたは認証情報に関連付けられる）にマッピングします。

接続リークの検出

VisiConnect では、接続リークを回避するために 2 つのメカニズムが提供されています。

- ガベージコレクタの活用
- 接続オブジェクトの使用を追跡するためのアイドルタイマーの提供

ra.xml 仕様のセキュリティポリシー処理

VisiConnect では、管理される実行時環境でリソースアダプタを実行するための一連のセキュリティ権限が提供されています。また、AppServer は、システムリソースにアクセスするための明示的権限をリソースアダプタに与えます。

リソースアダプタ

VisiConnect では、サンプルとしていくつかのリソースアダプタのソースコードが提供されています。これらのリソースアダプタは、JDBC 2.0 呼び出しのためのラッパーで、CCI を使用するものと使用しないものがあります。各リソースアダプタには、3 つのトランザクションレベルをサポートするデプロイメントデスクリプタが提供されています。

VisiConnect には、これらの JDBC リソースアダプタのための簡単なサンプルアプリケーションが用意されています。EJB は EIS のデータをモデル化するために使用され、J2EE クライアントおよびサーブレットはリソースアダプタにクエリーを送り、出力を表示するために使用されます。サンプルでは、JDBC 2.0 準拠のドライバがサポートする RDBMS を使用します。デフォルトでは、サンプルは、EIS として JDataStore を使用するように設定されていますが、任意の JDBC 2.0 RDBMS を使用するように簡単に設定できます。コンポーネントは、J2EE アプリケーションとしてパッケージされています。詳細については、AppServer に添付されている VisiConnect サンプルの `README` を参照してください。

製品に付属するその他のリソースアダプタのサンプルには、Tibco や OpenJMS などの JMS プロバイダと統合するための手順を含むオープンソースの汎用的な JMS リソースアダプタ、電子メールサーバーを EIS として使用できる Mail リソースアダプタがあります。これらのサンプルは、メッセージインフローの使い方を説明し、EIS からアプリケーションサーバーへの受信通信と送信通信機能を可能にします。

第 33 章

パーティションインターセプタの実装

パーティションインターセプタを実装するには、次の手順にしたがう必要があります。

1. `module-borland.xml` デスクリプタファイルを使ってインターセプタを定義します。
2. インターセプタクラスを作成します。
3. クラスとデスクリプタファイルを JARing します。
4. JAR を目的のパーティションにデプロイメントします。

インターセプタの定義

`module-borland.xml` ファイルを作成してインターセプタを定義します。このファイルでは、次の DTD が使用されます。

```
<!ELEMENT module (Partition-interceptor?)>
<!ELEMENT Partition-interceptor (class-name, argument?, priority?)>
<!ELEMENT class-name (#PCDATA)>
<!ELEMENT argument (key, value)>
<!ELEMENT key (#PCDATA)>
<!ELEMENT value (#PCDATA)>
<!ELEMENT priority (#PCDATA)>
```

`<class-name>` 要素は、JAR 内に含まれるインプリメンテーションのフルパスのクラス名を含む必要があります。

`<priority>` 要素は、特定のパーティションのインターセプタのセットの起動順序を制御するオプションのフィールドです。この値は、0 ～ 9 までの数値で指定する必要があります。優先順位 0 は優先順位 9 より上にランクされます。インターセプタは、ロード時には正順に起動され、シャットダウン時には逆順に起動されます。2 つ以上のインターセプタが同じ優先順位の場合、ほかと比較してどのインターセプタを起動するかを決定する方法はありません。

`<argument>` は、`<key>` 要素と `<value>` 要素のペアを含むオプションの要素です。これらは、クラスインプリメンテーションに `java.util.HashMap` として渡されます。コードで、該当する値をこのタイプから抽出する必要があります。引数に対しては、JVM インプリメンテーションによって制限が適用されます。

たとえば、次の XML は InterceptorImpl と呼ばれるインターセプタを定義します。

```
<module>
<Partition-interceptor>
    <class-name>com.borland.enterprise.examples.InterceptorImpl</class-name>
    <argument>
        <key>key1</key>
        <value>value1</value>
    </argument>
    <argument>
        <key>key2</key>
        <value>value2</value>
    </argument>
    <argument>
        <key>key3</key>
        <value>value3</value>
    </argument>
    <priority>1</priority>
</Partition-interceptor>
</module>
```

インターセプタクラスの作成

クラスは、以下を実装する必要があります。

```
com.borland.enterprise.server.Partition.service.PartitionInterceptor
```

次のメソッドを使用できます。

```
public void initialize(java.util.HashMap args);
```

このメソッドは、Tomcat コンテナなどのパーティションサービスが作成および初期化される前に呼び出されます。このメソッドは、各インターセプタのロード時に呼び出されるため、<priority> パラメータの影響は受けません。

```
public void startupPreLoad();
```

このメソッドは、パーティションサービスが開始された後のパーティションサービスがモジュールをロードする前に呼び出されます。

```
public void startupPostLoad();
```

このメソッドは、すべてのパーティションサービスが個々のモジュールをロードした後に呼び出されます。

```
public void shutdownPreUnload();
```

このメソッドは、パーティションサービスが個々のモジュールをアンロードする前に呼び出されます。<priority> パラメータは、優先順位を逆転します。最初に優先順位 9 のインターセプタ、次に優先順位 8 という順序で呼び出されます。

```
public void shutdownPostUnload();
```

このメソッドは、サービスがモジュールをアンロードした後に呼び出されます。

```
public void PartitionTerminating();
```

このメソッドは、サービスがシャットダウンされた後のパーティションがシャットダウンする直前に呼び出されます。

次のサンプルコードは、上記の module-borland.xml デスクリプタで定義されたクラス InterceptorImpl を示します。

```
package com.borland.enterprise.examples;

// このインターフェースは xmlrt.jar に含まれます。
import com.borland.enterprise.server.Partition.service.PartitionInterceptor;

public class InterceptorImpl implements PartitionInterceptor {
    static final String _className = "InterceptorImpl";

    public void initialize(java.util.HashMap args) {
    // System.out と System.err に書き込むと
    // 出力が記録されます。
    // 記録するために必要な条件はありません。
     System.out.println(_className + ": initialize");
     System.out.println("key1 has value " + args.get("key1").toString());
     System.out.println("key2 has value " + args.get("key2").toString());
     System.out.println("key3 has value " + args.get("key2").toString());
    }
    public void startupPreLoad() {
    // System.out と System.err に書き込むと
    // 出力が記録されます。
    // 記録するために必要な条件はありません。
    System.out.println(_className + ": startupPreLoad");
    }
    public void startupPostLoad() {
    // System.out と System.err に書き込むと
    // 出力が記録されます。
    // 記録するために必要な条件はありません。
      System.out.println(_className + ": startupPostLoad");
    }
    public void shutdownPreUnload() {
    // System.out と System.err に書き込むと
    // 出力が記録されます。
    // 記録するために必要な条件はありません。
     System.out.println(_className + ": shutdownPreUnload");
    }
    public void shutdownPostUnload() {
    // System.out と System.err に書き込むと
    // 出力が記録されます。
    // 記録するために必要な条件はありません。
      System.out.println(_className + ": shutdownPostUnload");
    }
    public void PartitionTerminating() {
    // System.out と System.err に書き込むと
    // 出力が記録されます。
    // 記録するために必要な条件はありません。
    System.out.println(_className + ": PartitionTerminating");
    }
}
```

JAR ファイルの作成

Java の JAR ユーティリティを使用して、クラスとそのデスクリプタファイルの JAR ファイルを作成します。

インターセプタのデプロイメント

デプロイメントウィザードを使用して、インターセプタをパーティションにデプロイメントします。［Verify deployment descriptors］チェックボックスと［Generate stubs］チェックボックスは、どちらもチェック**しない**でください。

重要 インターセプタをデプロイメントした後で、パーティションを再起動する必要があります。

JAR ファイルを次の 2 つのディレクトリのいずれかにコピーすることもできます。その後、必ずパーティションを手動で再起動してください。

- `<install_dir>/var/servers/<server_name>/Partitions/<Partition_name>/lib`
- `<install_dir>/var/servers/<server_name>/Partitions/<Partition_name>/lib/system`

第 34 章

VisiConnect の使い方

Java 2 Enterprise Edition（J2EE）コネクタアーキテクチャを使用すると、EIS ベンダーやサードパーティのアプリケーション開発者は、リソースアダプタを開発して J2EE 1.4 プラットフォーム仕様をサポートするアプリケーションサーバーにデプロイメントできます。リソースアダプタは、J2EE コンポーネントと EIS をプラットフォーム固有の方法で統合します。リソースアダプタが Borland AppServer（AppServer）にデプロイメントされると、さまざまな異種 EIS にアクセスできる堅固な J2EE アプリケーションを開発できます。リソースアダプタは、Java コンポーネントのほかに、必要に応じて EIS との対話に必要なネイティブなコンポーネントをカプセル化します。

VisiConnect を使用する前に、コネクタ 1.5 仕様をよくお読みください。

VisiConnect サービス

リソースアダプタは、VisiConnect パーティションサービスが有効になっているパーティションでホストされます。同じパーティションに複数のリソースアダプタをデプロイメントできます。VisiConnect は、デプロイメントリソースアダプタの接続ファクトリを JNDI を通してクライアントが使用できるようにします。これにより、クライアントは、JNDI を使用して特定のリソースアダプタの接続ファクトリを検索できます。

サービスの概要

VisiConnect サービスは、オプション機能をすべて備えたコネクタ 1.5 仕様の完全なインプリメンテーションです。

デプロイメントされたコネクタ内のリソースアダプタオブジェクトは、すべてリソースアダプタオブジェクトであると同時に CORBA オブジェクトでもあります。

他のコネクタのインプリメンテーションとは異なり、VisiConnect では、分割方法に制約がありません。任意の数のマシンで実行されている任意の数のパーティションには、任意の数のリソースアダプタを格納できます。さらに、分散トランザクションプロトコルのサポートにより、リソースアダプタを任意に分割できます。この分割により、デプロイメント時にアプリケーションを設定して、アプリケーション全体のパフォーマンスを最適化できます。

接続管理

ra.xml デプロイメントデスクリプタファイルには、ManagedConnectionFactory インスタンスの単一の設定を宣言するための config-property 要素が含まれています。通常、この設定プロパティは、リソースアダプタプロバイダが設定します。しかし、設定プロパティが設定されていない場合は、リソースアダプタデプロイヤがプロパティの値を提供する必要があります。

Borland は、コネクタおよびその接続ファクトリプロパティを定義するための独自のデプロイメントデスクリプタ、ra-borland.xml を提供します。ra-borland.xml デスクリプタの使い方の詳細は、「Borland DTD」を参照してください。

接続プロパティの設定

以下の接続プールプロパティを設定できます。

プロパティ	値型	説明	デフォルト値
wait-timeout	Integer	maximum-capacity 接続が開かれているときに、接続が解放されるまで待つ時間を秒単位で指定します。maximum-capacity プロパティを使用しており、プールがいっぱいで、これ以上接続を使用できない場合は、接続を検索するスレッドは、待ち時間が無制限に設定されている（0 秒に設定）と、その接続が使用できるようになるまで待機します。必要に応じて、wait-timeout 時間を設定できます。	30
busy-timeout	Integer	ビジー接続が解放されるまで待つ時間を秒単位で指定します。接続が長時間ビジーの場合は、それを使用するアプリケーションがハングして接続を解放できなくなります。このタイムアウト機能により、必要以上に長くビジー状態が続いた場合に接続を確実にタイムアウトにできます。	600（10 分）
idle-timeout	Integer	このタイムアウトを超えてアイドル状態が続いたプールされた接続は、リソースを節約するために閉じられます。アイドル接続に対して、60 秒ごとに idle-timeout の期限切れが確認されます。idle-timeout の値の単位は秒数です。0（ゼロ）値は、接続クリーンアップが無効であることを示します。	600（10 分）
maximum-capacity	Integer	VisiConnect によって許可される、管理接続の最大数を指定します。新しく割り当てられた管理接続に対するリクエストがこの上限を超えると、ResourceAllocationException が生成されます。	10

以下のプロパティは使用されなくなりました。VisiConnect はこれらを無視します。上の表に記載されているプールのプロパティ busy-timeout、idle-timeout、および wait-timeout に置き換えられました。ra-borland.xml から古い形式のプロパティを削除することはできません。

使用されなくなったプールのプロパティ

プロパティ	デフォルト値	説明
initial-capacity	1	デプロイメント時に VisiConnect が取得を試みる管理接続の初期数を指定します。
capacity-delta	1	保持される接続プールのサイズ変更時に VisiConnect が取得を試みる追加の管理接続の数を指定します。
cleanup-enabled	true	システムリソースを制御するために、接続プールで未使用の管理された接続を再利用するかどうかを示します。
cleanup-delta	1	接続プール管理が未使用の管理接続に対して行う再要求の間隔を指定します。

セキュリティマップを使用したセキュリティ管理

セキュリティマップを使用すると、次のようなユーザーロールを定義できます。

1 コンテナ管理のサインオンで EIS が直接使用するユーザーロール（use-caller-identity）

2 コンテナ管理のサインオンで適切なリソースロールにマップされるユーザーロール（run-as）

最初のユーザーロールの場合、実行時に特定されたユーザーロールがマッピングで見つかると、ユーザーロール自身を使用して EIS と通信するためのセキュリティ情報が提供されます。2 番めのユーザーロールの場合、実行時に特定されたユーザーロールがマッピングで見つかると、関連付けられているリソースロールを使用して EIS と通信するためのセキュリティ情報が提供されます。

use-caller-identity オプションは、実行時に特定されたユーザーロールのユーザー ID が EIS でも使用できる場合に使用されます。たとえば、AppServer では、ロール「Borland」に属するユーザー ID “borland”/”borland” を使用でき、使用可能な EIS である JDataStore データベースでも “borland”/”borland” という ID を使用できるとします。JDataStore を処理するリソースアダプタが、次のように設定されたセキュリティマップを使用してデプロイメントされる場合を考えます。

```
<security-map>
 <user-role>Borland</user-role>
   <use-caller-identity></use-caller-identity>
</security-map>
```

この場合、このサーバーインスタンス上にあるこの JDataStore データベースを使用するアプリケーションは、use-caller-identity を使用して JDataStore データベースにアクセスできます。

メモ 現在の VisiSecure の制限により、リソースボールトとユーザーボールトの両方で caller id を定義する必要があります。

run-as オプションは、実行時に特定されたユーザーロールのユーザー ID を EIS の ID にマップすることが有効である場合に使用されます。たとえば、AppServer では「Demo」ロールに属する “demo”/”demo” というユーザー ID を使用でき、使用可能な EIS である Oracle データベースには Demo ユーザーにとって最適な “scott”/”tiger” という ID があるとします。Oracle を処理するリソースアダプタが、次のように設定されたセキュリティマップを使用してデプロイメントされる場合を考えます。

```
<security-map>
 <user-role>Demo</user-role>
 <run-as>
 <role-name>oracle_demo</role-name>
  <role-description>Oracle demo role</role-description>
 </run-as>
</security-map>
```

この場合、リソースボールト（下記を参照）で `oracle_demo` ロールが定義されていると、このサーバーインスタンス上にあるこの Oracle データベースを使用するアプリケーションは、run-as を使用して Oracle データベースにアクセスできます。

run-as を使用する場合は、VisiConnect がリソースロールのセキュリティ情報の抽出に使用するボールトを指定する必要があります。このボールトには、リソースロール名と一連の認証情報が書き込まれます。run-as を使用してセキュリティマップが定義されているリソースアダプタを読み込む場合、VisiConnect は、定義済みロール名の認証情報をボールトから呼び出します。

承認ドメイン

ra-borland.xml デスクリプタファイルの <authorization-domain> 要素は、特定のユーザーロールに関連付けられた承認ドメインを指定します。<security-map> が設定されている場合、関連付けられているドメインを <authorization-domain> に設定する必要があります。<authorization-domain> が設定されていない場合、VisiConnect は、**デフォルトの**承認ドメインを使用することを前提としています。承認ドメインの使用の詳細は、『*Security Guide*』の「“ セキュリティの概要」を参照してください。

デフォルトのロール

また、<security-map> 要素を使用すると、デフォルトのユーザーロールを定義して、適切なリソースロールと関連付けることができます。実行時に特定されたユーザーロールがマッピングにない場合は、このデフォルトのロールが優先されます。<security-map> 要素でデフォルトのユーザーロールを定義するには、<user-role> 要素に「*」という値を指定します。たとえば、次のようになります。

```
<user-role>*</user-role>
```

対応する <role-name> エントリが <security-map> 要素で指定されている必要があります。AppServer のユーザーロールとリソースロールの関連付けの例を次に示します。

```
<security-map>
    <user-role>*</user-role>
    <run-as>
        <role-name>SHME_OPR</role-name>
    </run-as>
</security-map>
```

接続プールパラメータに AppServer が接続を初期化するように指定していても、デプロイメント時にはデフォルトのユーザーロールが使用されます。デフォルトのユーザーロールのエントリも <security-map> 要素もないと、サーバーはコンテナ管理セキュリティを利用した接続を確立できない場合があります。

リソースボールトの生成

前述のように、run-as セキュリティマッピングを使用するには、AppServer に提供されるボールトでリソースロールが定義されている必要があります。このようなボールトを「リソースボールト」と言います。

VisiConnect で提供されているツール ResourceVaultGen を使用すると、リソースボールトを作成し、作成したボールトでロールオブジェクトをインスタンス化できます。ロール名と関連するセキュリティ認証が、ResourceVaultGen によってリソースボールトに書き込まれます。この時点でリソースボールトに書き込むことができるのはパスワード認証タイプの認証だけです。ResourceVaultGen の使い方を次に示します。

```
java -Dborland.enterprise.licenseDir=<install_dir/var/domains/base/
configurations/<configuration_name>/mos/<partition_name>/adm> -
Dserver.instance.root=<install_dir/var/domains/base/configurations/
<configuration_name>/mos/<partition_name>/adm/properties/
management_vbroker.properties>
com.borland.enterprise.visiconnect.tools.ResourceVaultGen -rolename <role_name>
-username <user_name> -password <password> -vaultfile <full path to vault file>
-vpwd <vault_password>
```

オプションの意味は次のとおりです。

`-rolename`	リソースボールトに格納するリソースロール名。
`-username`	リソースロールに関連付けるリソースユーザー名。
`-password`	リソースロールに関連付けるリソースパスワード。
`-vaultfile`（オプション）	リソースロールを書き込むボールトファイルのパス。指定がない場合、ResourceVaultGen はデフォルトのリソースボールトファイル <*install_dir*/var/domains/base/configurations/<configuration_name>/mos/<partition_name>/adm/properties/management_vbroker.properties> に書き込みます。ボールトファイルがない場合は、指定されたロケーションに新しいボールトファイルが書き込まれます。
`-vpwd`（オプション）	アクセス承認のためのボールトに割り当てるパスワード。指定がない場合、ボールトはパスワードなしで作成されます。

ResourceVaultGen を使用する場合は、次の Jar が CLASSPATH にあることを確認してください。

- lm.jar
- visiconnect.jar
- vbsec.jar
- jsse.jar
- jaas.jar
- jce1_2_1.jar
- sunjce_provider.jar
- local_policy.jar
- US_export_policy.jar

メモ ボールトを生成しようとするときに CLASSPATH にこれらの Jar が存在しないと、無効なボールトファイルが生成されます。無効なボールトファイルを再利用しようとすると、EOFException が発生します。これを解決するには、無効なボールトファイルを削除し、CLASSPATH 内に適切な Jar があることを確認した上で、ResourceVaultGen を使用してボールトファイルを再生成します。

リソースアダプタのデプロイメント時にセキュリティマップ情報が指定されると、VisiConnect はボールトを使用します。リソースボールトがパスワード保護されている場合、VisiConnect に次のプロパティを渡す必要があります。

```
-Dvisiconnect.resource.security.vaultpwd=<vault_password>
```

リソースボールトがユーザーが指定した場所にある場合（-vaultfile ...）、VisiConnect に次のプロパティを渡す必要があります。

```
-Dvisiconnect.resource.security.login=<path of specified vault file>
```

以下の例は、ResourceVaultGen の使い方を示しています。

例 1 ：

```
java -Dborland.enterprise.licenseDir=/opt/BES/var<install_dir/var/domains/base/
configurations/<configuration_name>/mos/<partition_name>/adm/properties/
management_vbroker.properties>
 -Dserver.instance.root=/opt/BES/var/servers/servername -
Dpartition.name=standard
 com.borland.enterprise.visiconnect.tools.ResourceVaultGen -rolename
administrator
 -username red -password balloon -vaultfile
 /opt/BES/var/servers/servername/adm/properties/partitions/standard/
resourcevault -vpwd
 lock
```

この使用例では、ユーザー名 `red`、パスワード `balloon` というパスワード認証に関連付けられた `administrator` ロールが格納されている `resourcevault` というリソースボールトが /opt/BES/var/servers/servername/adm/properties/partitions/standard に生成されます。ボールトファイル自体は、パスワード `lock` を使用してパスワード保護されています。VisiConnect がこのボールトを使用するには、次のプロパティが設定されている必要があります。

```
-Dvisiconnect.resource.security.vaultpwd=lock
-Dvisiconnect.resource.security.login=resourcevault
```

例 2：

```
java -Dborland.enterprise.licenseDir=/opt/BES/var/domains/base/configurations/
<configuration_name>/mos/<partition_name>/adm/properties/
management_vbroker.properties>
-Dserver.instance.root=/opt/BES/var/domains/base/configurations/
<configuration_name>/mos/<partition_name>/adm/properties/
management_vbroker.properties>
-Dpartition.name=petstore
com.borland.enterprise.visiconnect.tools.ResourceVaultGen
-rolename manager accounts -username mickey daffy
-password mouse duck -vpwd goofy
```

この使用例では、ユーザー名 `mickey`、パスワード `mouse` というパスワード認証に関連付けられた `manager` ロールと、ユーザー名 `daffy`、パスワード `duck` というパスワード認証に関連付けられた `accounts` ロールが格納されている `resource_vault` というデフォルトのリソースボールトが /opt/BES/var/servers/servername/adm/properties/partitions/petstore に生成されます。ボールトファイル自体は、パスワード `goofy` を使用してパスワード保護されています。VisiConnect がこのボールトを使用するには、次のプロパティが設定されている必要があります。

```
-Dvisiconnect.resource.security.vaultpwd=goofy
```

例 3 ：

```
java -Dborland.enterprise.licenseDir=/opt/BES/var/servers/servername/adm -
Dserver.instance.root=/opt/BES/var/servers/servername
-Dpartition.name=standard
com.borland.enterprise.visiconnect.tools.ResourceVaultGen
-rolename OClone ENolco -username darkstar geraldo -password meteor rivera
```

この使用例では、ユーザー名 `darkstar`、パスワード `meteor` という パスワード 認証に関連付けられた `developer` ロールと 、ユーザー名 `geraldo`、パスワード `rivera` という パスワード 認証に関連付けられた `host` ロールが格納されている `resource_vault` という デフォルト のリソースボールト が /opt/BES/var/domains/base/configurations/<configuration_name>/mos/<partition_name>/adm/properties/management_vbroker.properties> に生成されます。ボールト ファイル自体はパスワード 保護されていません。VisiConnect がこのボールト を使用するための追加パラメータは不要です。

メモ `ResourceVaultGen` を使用して、無効な文字を含む既存のファイルにボールト情報を書き込むことはできません。たとえば、「touch」によって生成されたファイル、StarOffice 文書または Word 文書などには書き込むことができません。`ResourceVaultGen` がボールト情報を書き込むことができるのは、`ResourceVaultGen` 自身が生成した新しいファイルまたは有効な既存のボールトファイルだけです。

リソースアダプタの概要

コネクタ 1.5 仕様によると、ユーザーは、EAR（Enterprise Archive）の一部として RAR（Resource Archive）をデプロイメントできます。AppServer と VisiConnect を使用して、スタンドアロン RAR をデプロイメントすることもできます。RAR をデプロイメントしたら、次の操作を実行します。

- 接続を取得するためにコードを記述する。
- Interaction オブジェクトを作成する。
- Interaction Spec を作成する。
- レコードセットや結果セットのインスタンスを作成する。
- レコードオブジェクトにデータが入力されるように実行コマンドを実行する。

この章では、概念的な情報のほか、必要なコードの理解のためにその記述手順についても説明します。

企業情報システム（EIS）ベンダーとサードパーティのアプリケーション開発者は、J2EE コネクタアーキテクチャを使用してリソースアダプタを開発して J2EE 1.4 準拠の任意の Borland Enterprise Server にデプロイメントできます。リソースアダプタは、J2EE コネクタアーキテクチャ（コネクタ）の主要コンポーネントで、J2EE アプリケーションコンポーネントと EIS をプラットフォーム固有の方法で統合します。リソースアダプタが AppServer にデプロイメントされると、さまざまな異種 EIS にアクセスできる堅固な J2EE アプリケーションを開発できます。リソースアダプタは、Java コンポーネントのほかに、必要に応じて EIS との対話に必要なネイティブなコンポーネントをカプセル化します。

開発の概要

詳細は、289 ページの「リソースアダプタの開発」を参照してください。

リソースアダプタを最初から開発するには必要なインターフェースとデプロイメントデスクリプタを実装し、RAR（Resource Adapter Archive）にパッケージしてからその RAR を AppServer にデプロイメントする必要があります。リソースアダプタを開発する手順は、次のとおりです。

1 リソースアダプタが必要とする各種インターフェースとクラス用の Java コードを、コネクタ 1.5 仕様の範囲内で記述します。

2 これらのクラスを ra.xml 標準デプロイメントデスクリプタファイルで指定します。

3 インターフェースとインプリメンテーションのための Java コードをコンパイルしてクラスファイルに出力します。

4 Java クラスを JAR（Java Archive）ファイルにパッケージします。

5 次に説明するようにして、リソースアダプタ固有のデプロイメントデスクリプタを作成します。

- `ra.xml`: Sun の標準 DTD を使用してリソースアダプタに関連する属性とデプロイメントプロパティを記述します。
- `ra-borland.xml` : AppServer 固有の追加のデプロイメント情報を追加します。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が記述されています。

6 RAR（Resource Adapter Archive）ファイルを作成します（リソースアダプタをパッケージします）。

7 RAR を AppServer にデプロイメントするか、EAR（Enterprise Application Archive）ファイルに含めて、J2EE アプリケーションの一部としてデプロイメントします。

既存のリソースアダプタの編集

AppServer にデプロイメントする既存のリソースアダプタがある場合は、前述の Borland 固有のデプロイメントデスクリプタを編集してからアダプタをパッケージし直すだけで済みます。次に、その操作方法とサンプルを示します。

1 RAR 展開用の空のディレクトリを作成します。

```
mkdir c:/temp/staging
```

2 デプロイメントするリソースアダプタをそのディレクトリにコピーします。

```
cp shmeAdapter.rar c:/temp/staging
```

3 リソースアダプタアーカイブの内容を展開します。

```
jar xvf shmeAdapter.rar
```

展開用ディレクトリの内容は次のようになります。

- リソースアダプタをインプリメンテーションする Java クラスを含む JAR
- ファイル `Manifest.mf` および `ra.xml` を含む `META-INF` ディレクトリ

1 Borland デプロイメントデスクリプタエディタ（DDEditor）を使用して `ra-borland.xml` ファイルを作成し、展開用の `META-INF` ディレクトリに保存します。DDEditor の使い方については、『*管理コンソールユーザーズガイド*』の「デプロイメントデスクリプタエディタの使い方」を参照してください。

2 リソースアダプタアーカイブを新規作成します。

```
jar cvf shmeAdapter.rar -C c:/temp/staging
```

3 これで、リソースアダプタを AppServer にデプロイメントできるようになります。

リソースアダプタのパッケージ

リソースアダプタは、RAR に含まれている J2EE コンポーネントです。リソースアダプタでは一般的なディレクトリ形式を使用します。リソースアダプタのディレクトリ構造のサンプルを次に示します。

コードのサンプル 53.1 リソースアダプタのディレクトリ構造：

```
.META-INF/ra.xml
.META-INF/ra-borland.xml
./images/shmeAdapter.jpg
./readme.html
./shmeAdapter.jar
./shmeUtilities.jar
./shmeEisSdkWin32.dll
./shmeEisSdkUnix.so
```

前述の構造では、リソースアダプタが直接使用することはない画像や Readme ファイルなどの文書や関連ファイルを入れることができます。リソースアダプタをパッケージすることは、これらのファイルも同様にパッケージすることを意味します。

リソースアダプタのパッケージは、次の手順で行います。

1. 一時的な展開用ディレクトリを作成します。
2. リソースアダプタの Java クラスをコンパイルして展開用ディレクトリに出力します。または、前述のように、事前にコンパイルしたクラスを展開用ディレクトリにコピーします。
3. リソースアダプタの Java クラスを保存する JAR ファイルを作成します。この JAR を展開用ディレクトリの最上位に追加します。
4. 展開用ディレクトリの中に、`META-INF` サブディレクトリを作成します。
5. このサブディレクトリに ra.xml デプロイメントデスクリプタを作成し、リソースアダプタのエントリを追加します。ra.xml の文書型定義については、次のサイトにある Sun Microsystems の文書を参照してください。`http://java.sun.com/dtd/connector_1_0.dtd`
6. 同じ META-INF サブディレクトリに ra-borland.xml デプロイメントデスクリプタを作成し、リソースアダプタのエントリを追加します。必要なエントリの詳細は、このマニュアルの最後にある DTD を参照してください。
7. リソースアダプタアーカイブを作成します。

   ```
   jar cvf resource-adapter-archive.rar -C staging-directory
   ```

   このコマンドは、サーバーにデプロイメントする RAR ファイルを作成します。JAR コマンドで `-C staging-directory` オプションを指定すると、`staging-directory` に移動します。これにより、RAR ファイルに記録されているディレクトリパスは、リソースアダプタが展開されたディレクトリを基準にした相対パスになります。
   1 つまたは複数のリソースアダプタは、ディレクトリに展開して JAR ファイルにパッケージすることができます。

リソースアダプタのデプロイメントデスクリプタ

AppServer は、2 つの XML ファイルを使用してデプロイメント情報を指定します。1 つのファイルは、Sun Microsystems のリソースアダプタ用 DTD をベースにした `ra.xml` です。もう 1 つのファイルは、AppServer に必要な追加のデプロイメント情報が記述された Borland 独自の `ra-borland.xml` です。

ra.xml の設定

リソースアダプタに関連付けられた `ra.xml` ファイルがまだない場合は、自分で新しく作成するか、既存のファイルを編集する必要があります。こうしたプロパティは、テキストエディタや Borland DDEditor で編集できます。`ra.xml` ファイルの作成方法の最新情報については、次の場所にあるコネクタ仕様を参照してください。
`http://java.sun.com/j2ee/connector`.

トランザクションレベルタイプの設定

リソースアダプタで対応できるトランザクションレベルタイプを `ra.xml` デプロイメントデスクリプタに指定することは非常に重要です。次の表に、対応できるトランザクションレベルと XML での表記規則を示します。

対応するトランザクションのタイプ	XML 表記の DTD
なし	<transaction-support>**NoTransaction**</transaction-support>
Local	<transaction-support>**LocalTransaction**</transaction-support>
XA	<transaction-support>**XA**</transaction-support>

ra-borland.xml の設定

`ra-borland.xml` ファイルには、リソースアダプタを AppServer にデプロイメントするために必要な情報が記述されています。RAR ファイルをデプロイメントするには、このファイルに特定の属性を指定する必要があります。この機能は、AppServer の EJB、EAR、WAR、およびクライアントコンポーネントに対応する `.xml` 拡張機能と整合性があります。

`ra-borland.xml` ファイルに Borland 固有のデプロイメントプロパティを指定するまで、RAR をサーバーにデプロイメントすることはできません。RAR をデプロイメント可能にするためには、ra-borland.xml に次の属性が必要です。

- リソースアダプタのインスタンス名。この名前は、このパーティションにデプロイメントされたすべての RAR の間で一意である必要があります。これは、デプロイメントされたリソースアダプタを VisiConnect サービスが一意に識別するために使用されます。これは、リソースアダプタが着信をサポートしている場合に、受信メッセージを受け取る予定のリソースアダプタを識別するために、エンドポイント MDB が `ejb-borland.xml` デスクリプタ内で使用する名前です。
- `ra.xml` ファイルの各接続定義
  - 接続ファクトリのインターフェースクラス名。これは、リソースアダプタ内のすべての接続定義で一意である必要があります。このクラス名は、`ra-borland.xml` 内の特定の接続ファクトリの仕様を `ra.xml` ファイル内の対応するファクトリの仕様に関連付けます。
  - 接続ファクトリ名。これは、このパーティションにデプロイメントされたすべてのリソースアダプタの間で一意である必要があります。
  - JNDI 接続ファクトリ名。これは、このパーティションにデプロイメントされたすべてのリソースアダプタの間で一意である必要があります。

次の省略可能な属性も `ra-borland.xml` ファイルで指定できます。

- 現在のリソースアダプタと共有されるリソースアダプタコンポーネントを含む個別デプロイメント接続ファクトリへのリファレンス。
- すべての共有ライブラリのコピー先となるディレクトリ。

- リソースアダプタ／EIS サインオン処理に対するセキュリティプリンシパルのマッピング。このマッピングにより、コンテナ管理セキュリティを利用するアプリケーションに対する EIS 接続を要求するときのリソースプリンシパルが特定されます。また、このリソースプリンシパルは、初期デプロイメント中に EIS 接続を要求するときに使用されます。
- `ra.xml` ファイルの各接続定義
  - 接続ファクトリの説明
  - Logging-required フラグ。`ManagedConnectionFactory` クラスおよび `ManagedConnection` クラスでログを記録する必要があるどうかを示します。
  - ログファイルの場所
  - 接続プールのプロパティ
    - `busy-timeout`：ビジー接続が解放されるまで待つ時間を秒単位で指定します。デフォルトは 600 秒です
    - `idle-timeout`：プールされた接続は、このタイムアウトを超えてアイドル状態が続くと閉じられます。アイドル接続に対して、60 秒ごとに期限切れが確認されます。`idle-timeout` の値の単位は秒数です。デフォルトは 600 秒です。
    - `wait-timeout`:：接続が解放されるまで待機する秒数。デフォルトは 30 秒です。

コネクタ 1.5 のデプロイメントデスクリプタへの変更

BAS は、コネクタ 1.0 とコネクタ 1.5 の両方をサポートします。コネクタ 1.5 のデプロイメントデスクリプタにおける重要な変更を次に示します。

- 新しいリソースアダプタ実装クラスを ra.xml で指定する必要があります。

```
<resourceadapter>
   <resourceadapter-class>
      .ResourceAdapter.Implementation.Class
   </resourceadapter-class>
⋮
```

- 複数の接続定義を同じ RAR 内の <outbound-resourceadapter> に指定できます。
- 複数のメッセージアダプタを <inbound-resourceadapter> に指定できます。
- 設定プロパティは、`config-property` 要素を使用してリソースアダプタレベルで指定できます。config-property の型はオブジェクトでなくてはならず、プリミティブであってはなりません。たとえば、`int` でなく、`java.lang.Integer` を使用する必要があります。また、リソースアダプタインプリメンテーションの set メソッドで同じ型を使用する必要があります。たとえば、この例では、`setCount(java.lang.Integer value)` です。

```
<config-property>
   <description>Open User Name</description>
   <config-property-name>Count</config-property-name>
   <config-property-type>java.lang.Integer</config-property-type>
   <config-property-value>100</config-property-value>
</config-property>
```

  config-property は、各接続定義に固有で設定でき、接続定義ごとに設定することもできます。

メモ 受信リソースアダプタからメッセージを受信するエンドポイントとして設定されているメッセージ駆動型 Bean は、`ra.xml` で指定されている `messagelistener-type` インターフェースを実装する必要があります。

```
<inbound-resourceadapter>
   <messageadapter>
      <messagelistener>
         <messagelistener-type> </messagelistener-type>
```

EJB には、エンドポイントをアクティブ化するために必要なアクティブ化設定が必要です。

リソースアダプタのクラスローダーについて

Borland AppServer には、モジュール単位のクラスローダーポリシーを提供します。.ear、.war、.rar、または ejb.jar のデプロイメントモジュールには固有のクラスローダーがあり、そのクラスパスには、モジュールに組み込まれているすべてのクラスが含まれます。これにより、各モジュールはアクセス先のクラスを完全に制御できます。複数のモジュールがライブラリのバージョンの異なる独自のコピーを持ったり、他のモジュールで使用されるパッケージと競合せずに同じパッケージ名を使用できます。これは強力な機能ですが、複数のモジュールが協調して動作し、同じクラスを共有する場合には、状況が複雑になる可能性があります。

各モジュールには独自のクラスローダーがありますが、VisiConnect を実行するパーティションには、いくつかの独自のクラスローダーがあります。このパーティション単位のクラスローダーのクラスは、パーティションで実行されるすべてのモジュールで使用できます。一般に、モジュール単位のクラスローダーがパーティションレベルのクラスローダーよりも優先されるので、1 つのクラスが両者で見つかった場合は、モジュール単位の方が使用されます。

クラスローダーに関して、VisiConnect ユーザーがアプリケーションのパッケージングで注意を払う必要がある場面が 2 種類あります。

- 送信通信の接続ファクトリと接続
- 受信通信のメッセージリスナー

接続ファクトリと接続

送信通信の接続ファクトリと接続インターフェースおよび実装クラスは、リソースアダプタとともに提供されることがあります。インターフェースクラスは標準ベースで JDK または拡張機能の一部として提供され、実装クラスはリソースアダプタに固有で独自の規格に基づいていることがあります。たとえば、JMS リソースアダプタは、標準の `javax.jms.QueueConnectionFactory` クラスと、`javax.jms.QueueConnection` クラスの独自の実装を提供します。

`javax.jms` パッケージのクラスは Borland AppServer によって提供され、パーティションのクラスローダーに存在し、そのクラス定義は、すべてのモジュールで共有されます。ただし、リソースアダプタではこれらの `javax.jms` インターフェースを実装する独自のクラスが提供され、各モジュールは、この独自のクラス定義のコピーを持つことがあります。

リソースアダプタの接続を取得するために、クライアントプログラムはリソースアダプタの接続ファクトリの 1 つに対して JNDI 検索を実行します。この JNDI 検索で返されるオブジェクトは、クライアントが存在する `.ear`、`.war`、または `.jar` モジュールで使用できるクラス定義を使用して作成される必要があります。そうでない場合は、クライアントコードで接続ファクトリオブジェクトを使用しようとすると、`ClassCastException` が返されます。クライアントは、次に接続ファクトリを使用して Connection インスタンスを作成します。このオブジェクトも、クライアントコードがオブジェクトを操作できるように、クライアントモジュールのクラスローダーを使用して作成される必要があります。

また、AppServer 内で実行される VisiConnect サービスは、リソースアダプタによって提供されるクラスの一部を使用する必要があります。たとえば、1.5 レベルのリソースアダプタには、`javax.resource.spi.ResourceAdapter` のインプリメンテーションが含まれます。これは、VisiConnect によりリソースアダプタの開始と停止に使用されます。VisiConnect がリソースアダプタクラスを使用する場合は、常にリソースアダプタのクラスローダーが使用されます。スタンドアロンの `.rar`（クライアントで `.ear` に埋め込まれるのではなく）としてデプロイメントされたリソースアダプタは、独自のクラスローダーを持ち、したがって、リソースアダプタクラスのクラス定義の独自のコピーを持ちます。この場合、`ClassCastExceptions` が発生する可能性があります。

この問題は、たとえばメソッド `ManagedConnectionFactory.setResourceAdapter(javax.resource.spi.ResourceAdapter)` で発生します。`ManagedConnectionFactory` インスタンスはクライアントのクラスローダーを使用して作成され、ResourceAdapter インスタンスは `.rar` クラスローダーを使用して作成されます。このメソッドのインプリメンテーションで、ResourceAdapter インスタンスを固有の実装クラスにキャストすると、`ClassCastException` が発生します。

メッセージリスナー

受信リソースアダプタでは、メッセージリスナーとなるクラスを指定する必要があります。このクラスは、このリソースアダプタからの受信通信のエンドポイントとして動作する MDB に実装されます。リソースアダプタが MDB に渡すメッセージを持っている場合は、メッセージリスナークラスのメソッドが呼び出されます。たとえば、多くの JMS リソースアダプタは、`javax.jms.MessageListener` をメッセージリスナークラスとして使用し、このクラスの `onMessage(javax.jms.Message)` メソッドで実際に着信メッセージを受信します。これらのクラスは `javax.jms` パッケージで提供され、パーティションのクラスローダー内に存在するので、リソースアダプタと MDB クライアントの両方に共有され、この場合は `ClassCastExceptions` が発生することはありません。

ただし、リソースアダプタは固有のメッセージリスナークラスを提供でき、そのクラスは実際にメッセージを配信するメソッドをいくつでも持つことができます。このすべてのメソッドで、固有のオブジェクトを引数として使用できます。これが `ClassCastExceptions` の原因になる場合があります。

VisiConnect では、メッセージリスナークラスが独自の場合でも、MDB 内のメッセージ配信メソッドが MDB 自身のクラスローダーにあるメッセージリスナーの定義を使用して呼び出されるように、MDB の呼び出しが適切に処理されます。ただし、メソッドが固有のオブジェクトである引数を受け取る場合、VisiConnect は、オブジェクトのリソースアダプタのクラスから MDB のクラス定義にマップできません。これにより、`ClassCastExceptions` が発生する可能性があります。

たとえば、製品にサンプルとして付属する Mail リソースアダプタは、`com.borland.enterprise.ra.mail.api.MailListener` というメッセージリスナークラスを提供します。このクラスには、`onMessage(javax.mail.Message)` というメッセージ配信メソッドが含まれます。メッセージリスナークラスは固有ですが、`onMessage()` メソッドは、引数に固有でないオブジェクトを受け取ります。この場合、`ClassCastExceptions` は発生しません。メッセージリスナークラス自身が固有である場合もあります。メッセージ配信メソッドに固有のオブジェクトの引数がある場合にのみ、クラスローダーの問題が発生します。

ClassCastExceptions の修正

前述の問題のいずれかが発生した場合、基本的な解決方法は次の 2 つです。

- リソースアダプタ `.rar` とそのクライアントを含む `.ear` を作成します。`.ear` 全体で 1 つのクラスローダーを共有するので、リソースアダプタとクライアント間でオブジェクトを移動するときの `ClassCastExceptions` が発生しません。
- `.rar` とクライアントモジュールの両方からリソースアダプタクラスを削除し、これらのクラスをライブラリ `.jar` としてパーティションにデプロイメントします。ライブラリ `.jar` は、パーティションのクラスローダーに置かれ、すべてのデプロイメントモジュールに共有されます。そのため、すべてのモジュールがリソースアダプタクラスの同じクラス定義を使用するので、`ClassCastExceptions` が発生しません。

リソースアダプタの開発

ここでは、コネクタ 1.5 準拠のリソースアダプタを開発する方法について説明します。リソースアダプタは、次のようなシステム協定の必要条件を実装する必要があります。その詳細を以下に示します。

- 接続管理
- セキュリティ管理
- トランザクション管理
- パッケージングとデプロイメント

接続管理

リソースアダプタの接続管理協定には、システム協定に必要なクラスとインターフェースの数を指定します。リソースアダプタは、次のインターフェースを実装する必要があります。

- `javax.resource.spi.ManagedConnection`
- `javax.resource.spi.ManagedConnectionFactory`
- `javax.resource.spi.ManagedConnectionMetaData`

リソースアダプタが提供する ManagedConnection インプリメンテーションは、アプリケーションサーバーをサポートするために、次のインターフェースとクラスのインプリメンテーションを提供する必要があります。接続とそれに関連するトランザクションを最終的に管理するのは、Borland Enterprise Server です。

メモ 管理されていない（アプリケーションサーバーで管理されていない）環境では、こうしたインターフェースやクラスを使用する必要はありません。

- `javax.resource.spi.ConnectionEvent`
- `javax.resource.spi.ConnectionEventListener`

また、リソースアダプタで次のメソッドを実装して、エラー記録とトレースの機能を備えてください。

- `ManagedConnectionFactory.setLogWriter()`
- `ManagedConnectionFactory.getLogWriter()`
- `ManagedConnection.setLogWriter()`
- `ManagedConnection.getLogWriter()`

管理されていない 2 階層アプリケーションでリソースアダプタを使用する場合は、`javax.resource.spi.ConnectionManager` *インターフェースのデフォルトのインプリメンテーション*をリソースアダプタに備えてください。`ConnectionManager` のデフォルトのインプリメンテーションにより、リソースアダプタは独自のサービスを提供できます。このようなサービスには、接続プール、エラー記録、トレース、およびセキュリティ管理があります。デフォルトの `ConnectionManager` は、基底の EIS への物理接続の作成を `ManagedConnectionFactory` に委任します。

アプリケーションサーバー管理の環境では、リソースアダプタで `ConnectionManager` 実装クラスを使用しないでください。管理環境では、リソースアダプタはそれ自体の接続プールをサポートできません。この場合は、Borland Enterprise Server が接続プールを管理します。ただし、リソースアダプタは、1 つの物理接続ごとに、アプリケーションサーバーとそのコンポーネントには透過的な `ConnectionManager` インスタンスを複数持つことができます。

トランザクション管理

リソースアダプタは、提供するトランザクション対応のレベルに基づいて簡単に分類できます。これらのレベルを次に示します。

- **NoTransaction**：リソースアダプタは、ローカルトランザクションと JTA トランザクションのどちらもサポートせず、トランザクションインターフェースを実装しません。
- **LocalTransaction**：リソースアダプタは、`LocalTransaction` インターフェースを実装することによってリソースマネージャのローカルトランザクションに対応しています。ローカルトランザクション管理協定は、Sun Microsystems のコネクタ 1.5 仕様のセクション 6.7 で規定されています。
- **XATransaction**：リソースアダプタは、LocalTransaction インターフェースを実装してリソースマネージャのローカルトランザクションに対応し、XAResource インターフェースを実装してリソースマネージャの JTA/XA トランザクションに対応します。XA リソースベースの協定は、Sun Microsystems のコネクタ 1.5 仕様のセクション 6.7 で規定されています。

前述のトランザクション対応レベルは、サーバー管理トランザクションの調整を可能にするためにリソースアダプタが実装する必要があるトランザクションサポートの主要な手順を反映しています。トランザクションの機能と、基底の EIS の必要条件に応じて、リソースアダプタは前述のレベルのいずれかを選択してサポートすることができます。

セキュリティ管理

リソースアダプタのセキュリティ管理協定の必要条件を次に示します。

- リソースアダプタは、`ManagedConnectionFactory.createManagedConnection()` メソッドを実装してセキュリティ協定をサポートする必要があります。
- リソースアダプタは、`ManagedConnection.getConnection()` メソッドのインプリメンテーションの一部として再認証をサポートする必要はありません。
- リソースアダプタは、セキュリティ協定のサポートをそのデプロイメントデスクリプタの一部として指定する必要があります。関係するデプロイメントデスクリプタの要素を次に示します。
  - `<authentication-mechanism></authentication-mechanism>`
  - `<authentication-mechanism-type></authentication-mechanism-type>`
  - `<reauthentication-support></reauthentication-support>`
  - `<credential-interface></credential-interface>`

これらのデスクリプタの要素の詳細は、コネクタ 1.5 仕様のセクション 10.3.1 を参照してください。

パッケージングとデプロイメント

パッケージされたリソースアダプタモジュールのファイル形式は、リソースアダプタプロバイダとリソースアダプタデプロイヤの間の協定を定義します。パッケージされたリソースアダプタは、次の要素を含みます。

- コネクタシステムレベルの協定とリソースアダプタの機能の両方を実装するために必要な Java クラスとインターフェース
- リソースアダプタの Java ユーティリティクラス
- リソースアダプタに必要なプラットフォーム依存のネイティブライブラリ
- ヘルプファイルとマニュアル
- 前述の要素を結び付ける説明が記載されたメタ情報

パッケージの必要条件の詳細は、コネクタ 1.5 仕様のセクション 10.3 と 10.5 を参照してください。デプロイメントの必要条件と、サポートする JNDI の設定および検索について、個々に説明してあります。

リソースアダプタのデプロイメント

リソースアダプタのデプロイメントは、EJB、エンタープライズアプリケーション、および Web アプリケーションのデプロイメントに似ています。これらのモジュールと同様に、リソースアダプタはアーカイブファイルまたは展開されたディレクトリとしてデプロイメントできます。リソースアダプタは、AppServer コンソールまたは `iastool` ユーティリティを使用して動的にデプロイメントすることも、EAR に含めてデプロイメントすることもできます。デプロイメントの詳細は、AppServer の『*ユーザーズガイド*』を参照してください。

リソースアダプタをデプロイメントするときは、モジュールに名前を指定する必要があります。この名前は、リソースアダプタのデプロイメントの論理リファレンスで、特に、リソースアダプタを更新したりデプロイメントを解除するために使用できます。AppServer では、RAR ファイル名、またはリソースアダプタがあるデプロイメントディレクトリのファイル名と一致するデプロイメント名が暗黙的に割り当てられます。サーバーの起動後は、この論理名を使用してリソースアダプタを管理できます。リソースアダプタのデプロイメント名は、モジュールのデプロイメントが解除されるまで AppServer 内でアクティブなままになります。

アプリケーション開発の概要

アプリケーションコンポーネントの開発

コモンクライアントインターフェース（Common Client Interface、CCI）

EIS にアクセスするためにアプリケーションコンポーネントが使用するクライアント API は、次のように分類できます。

- コネクタ 1.5 仕様のセクション 9 で定義されている標準のコモンクライアントインターフェース（CCI）。
- リソースアダプタの種類および基底の EIS に固有の一般的なクライアントインターフェース。このようなインターフェースの例として、RDBMS 向けの JDBC があります。
- 特定のリソースアダプタおよび基底の EIS に固有の独自クライアントインターフェース。このようなインターフェースの例として、IBM CICS トランザクションプロセッサ向けの CICS Java Gateway や、SAP R/3 エンタープライズリソースプランニングシステム向けの JFC があります。

コネクタ 1.5 仕様では、EIS にアクセスするための CCI が定義されています。CCI はアプリケーションコンポーネント用の標準のクライアント API です。これを使用すると、アプリケーションコンポーネントと EAI フレームワークが異種 EIS 間でやり取りできます。CCI は、エンタープライズアプリケーション統合（EAI）、サードパーティのエンタープライズツールベンダー、および既存モジュールの J2EE プラットフォームへの移行での使用を主な目的としています。

CCI において、接続ファクトリは、EIS インスタンスへの接続を可能にするパブリックインターフェースです。このサービスを提供するために、ConnectionFactory インターフェースがリソースアダプタで実装されています。アプリケーションは、JNDI 名前空間内で ConnectionFactory インスタンスを検索し、それを使用して EIS 接続の取得を要求します。

次に、アプリケーションは、返された Connection インターフェースを使用して EIS にアクセスします。CCI と EIS 固有の API の両方で一貫したアプリケーションプログラミングモデルを提供するため、ConnectionFactory および Connection インターフェースは、インターフェーステンプレート設計パターンに準拠します。この設計パターンでは、接続を作成および閉じるスケルトンを定義し、適切なステップをサブクラスに任せます。これにより、これらのインターフェースを簡単に拡張して、接続を作成したり閉じたりする特定のステップを、それらの操作の構造を変更せずに再定義できるようにできます。これらのインターフェースへのインターフェーステンプレート設計パターンの適用については、コネクタ 1.5 仕様のセクション 5.5.1 を参照してください。
(`http://java.sun.com/j2ee/connector`)

管理対象アプリケーションでの接続の取得

*管理対象アプリケーション*が `res-type` 変数で指定されているように接続ファクトリから EIS インスタンスへの接続を取得する場合、次の手順を実行します。

1 アプリケーションアセンブラまたはコンポーネントプロバイダが、デプロイメントデスクリプタを使用して、アプリケーションコンポーネントの接続ファクトリ要件を指定します。

```
res-ref-name: shme/shmeAdapter
res-type:javax.resource.cci.ConnectionFactory
res-auth: Application|Container
```

2 リソースアダプタデプロイヤが、リソースアダプタの設定情報を設定します。

3 VisiConnect は、設定済みのリソースアダプタを使用して、基底の EIS への物理接続を作成します。

4 アプリケーションコンポーネントは、コンポーネントの環境内で接続ファクトリインスタンスの JNDI 検索を実行します。

```
// 初期 JNDI ネーミングコンテキストを取得します。
 javax.naming.Context ctx = new javax.naming.InitialContext();
 // JNDI 検索を実行して接続ファクトリを取得します。
 javax.resource.cci.ConnectionFactory cxFactory =
        (javax.resource.cci.ConnectionFactory)ctx.lookup(
            "java:comp/env/shme/shmeAdapterConnectionFactory");
```

5 コンテキスト検索で渡される JNDI 名は、コンポーネントのデプロイメントデスクリプタの `res-ref-element` で指定されているものと同じです。JNDI 検索は、`res-type` 要素で指定されている `java.resource.cci.ConnectionFactory` 型の接続ファクトリインスタンスを返します。

6 アプリケーションコンポーネントは、接続ファクトリで `getConnection()` メソッドを呼び出して、EIS 接続の取得を要求します。返された接続インスタンスは、基底の物理接続へのアプリケーションレベルのハンドルを表します。アプリケーションコンポーネントは、接続ファクトリで `getConnection()` メソッドを複数回呼び出すことにより、複数の接続を要求します。

```
javax.resource.cci.Connection cx = cxFactory.getConnection();
```

7 アプリケーションコンポーネントは、返された接続を使用して基底の EIS にアクセスします。これは、リソースアダプタ固有の機能です。

8 接続を終了すると、コンポーネントは、接続インターフェースで `close()` メソッドを使用して接続を閉じます。

```
cx.close();
```

9 アプリケーションコンポーネントが割り当てられた接続を使用後に閉じるのに失敗した場合、その接続は未使用の接続とみなされます。AppServer は、未使用の接続のクリーンアップを管理します。コンテナがコンポーネントインスタンスを終了すると、コンテナは、そのコンポーネントインスタンスが使用したすべての接続をクリーンアップします。

非管理対象アプリケーションでの接続の取得

非管理対象アプリケーションの場合、アプリケーションコンポーネントで同様のプログラミングモデルにしたがう必要があります。非管理対象アプリケーションは、接続ファクトリインスタンスを検索し、EIS 接続の取得を要求し、接続を使用して EIS とやり取りし、完了したら接続を閉じます。

*非管理対象アプリケーション*コンポーネントが接続ファクトリから EIS インスタンスへの接続を取得する場合、次の手順を実行します。

1. アプリケーションコンポーネントは、`javax.resource.cci.ConnectionFactory` インスタンスで `getConnection()` メソッドを呼び出し、基底の EIS インスタンスへの接続を取得します。
2. 接続ファクトリインスタンスは、デフォルトの接続マネージャインスタンスに接続要求を委任します。リソースアダプタは、デフォルトの接続マネージャのインプリメンテーションを提供します。
3. 接続マネージャインスタンスは、`ManagedConnectionFactory.createManagedConnection()` メソッドを呼び出すことにより、基底の EIS インスタンスへの新しい物理接続を作成します。
4. `ManagedConnectionFactory.createManagedConnection()` を呼び出すと、基底の EIS への新しい物理接続が作成されます。この EIS は、`ManagedConnectionFactory.createManagedConnection()` メソッドが返す `ManagedConnection` インスタンスによって表されます。`ManagedConnectionFactory` は、JAAS `Subject` オブジェクトからのセキュリティ情報、`ConnectionRequestInfo`、設定されているプロパティ（ポート番号、サーバー名など）を使用して、新しい `ManagedConnection` インスタンスを作成します。
5. 接続マネージャインスタンスは、`ManagedConnection.getConnection()` メソッドを呼び出してアプリケーションレベルの接続ハンドルを取得します。このメソッドを呼び出しても、必ずしも EIS インスタンスへの新しい物理接続が作成されるわけではありません。このメソッドは、`ManagedConnection` インスタンスで表される基底の物理接続にアクセスするためにアプリケーションによって使用される一時ハンドルを作成します。
6. 接続マネージャインスタンスは、接続ファクトリインスタンスへの接続ハンドルを返します。この接続ファクトリが、接続を要求しているアプリケーションコンポーネントに接続を返します。

サンプルコード —CCI に基づくプログラミング

次のコードの抜粋は、CCI に基づくアプリケーションプログラミングモデルの例を示しています。接続の取得を要求し、接続ファクトリを取得後、Interaction と InteractionSpec を作成します。レコードファクトリとレコードを取得してレコードとやり取りし、結果セットとカスタムレコードを使用して同じことを実行します。

```
// JNDI 名前空間から接続ファクトリインスタンスを検索後、
// EIS インスタンスへの接続を取得します。この例では、コンポーネントは、
// コンテナが EIS サインオンを管理することを許可します。
javax.naming.Context ctx = new javax.naming.InitialContext();
javax.resource.cci.ConnectionFactory cxFactory =
(javax.resource.cci.ConnectionFactory)ctx.lookup(
  "java:comp/env/shme/shmeAdapter" );
javax.resource.cci.Connection cx = cxFactory.getConnection();

// Interaction インスタンスを作成します。
javax.resource.cci.Interaction ix = ct.createInteraction();

// 個々の InteractionSpec の新しいインスタンスを作成します。
com.shme.shmeAdapter.InteractionSpecImpl ixSpec = new
com.shme.shmeAdapter.InteractionSpecImpl();
ixSpec.setFunctionName( "S_EXEC" );
ixSpec.setInteractionVerb( javax.resource.cci.InteractionSpec.SYNC_SEND_RECEIVE
);
// ...
// RecordFactory インスタンスを取得します。
javax.resource.cci.RecordFactory recFactory = // ... RecordFactory を取得します。

// RecordFactory インスタンスを使用して汎用の MappedRecord を作成します。
// このレコードインスタンスは、Interaction の実行のための入力として機能します。
// レコードの名前は、以下のレコード型のメタデータのポインタとして機能します。
javax.resource.cci.MappedRecord input = recFactory.createMappedRecord(
"ShmeExecRecord" );

// 汎用 MappedRecord インスタンスに入力値を格納します。コンポーネント
// コードは、メタデータリポジトリからアクセスしたメタデータに基づいて
// 値を追加します
input.put( "<key: element0>", new String( "S_APP01"     );
input.put( "<key: element1>", // ... );
// ...

// Interaction の実行によって設定される出力値を保持するための汎用
// IndexedRecord を作成します。
javax.resource.cci.IndexedRecord output =
    recFactory.createIndexedRecord( "ShmeExecRecord" );

// Interaction を実行します。
boolean response = ix.execute( ixSpec, input, output );

// 出力 IndexedRecord からデータを抽出します。汎用 IndexedRecord で
// メタデータリポジトリ内の型マッピング情報を使用して型マッピングが
// 実行されます。コンポーネントは IndexedRecord で汎用メソッドを
// 使用するため、コンポーネントコードは必要な型キャストを実行します
java.util.Iterator iter = output.iterator();

while ( iter != null && iter.hasNext() )
{
  // レコード要素を取得し、値を抽出します ...
}

// Interaction の実行によって返された ResultSet の必要条件を
// 設定します。このステップは任意です。必要条件が明示的に設定されていない場合
// はデフォルト値が使用されます。
com.shme.shmeAdapter.InteractionSpecImpl rsIxSpec =
    new com.shme.shmeAdapter.InteractionSpecImpl();
rsIxSpec.setFetchSize( 20 );
rsIxSpec,setResultSetType( javax.resource.cci.ResultSet.TYPE_SCROLL_INSENSITIVE
);

// ResultSet を返す Interaction を実行します。
```

```
javax.resource.cci.ResultSet rSet =
    (javax.resource.cci.ResultSet)ix.execute( rsIxSpec, input );

// ResultSet を繰り返します。この例では、最初の行にカーソルを置き、
// 次に ResultSet の内容を末尾に向かって繰り返し処理します。
// 次に、適切な get メソッドを使用して列の値を取得します。
rSet.beforeFirst();


while ( rSet != null && rSet.next() )
{
// 適切な get メソッドを使用して現在の行の列の値を取得します。
}

// 次のコードは、ResultSet を使用した逆繰り返し処理の例です
rSet.afterLast();

while ( rSet.previous() )
{
// 適切な get メソッドを使用して現在の行の列の値を取得します。
}

// Record インターフェースを拡張して EIS 固有のカスタム Record を表します。
// CustomerRecord インターフェースは、フィールドの値に対して単純なアクセッサ /
// ミューテータをサポートします。開発ツールは、CustomerRecord の
// 実装クラスを生成します。
public interface CustomerRecord extends javax.resource.cci.Record,
    javax.resource.cci.Streamable
{
  public void setName( String name );
  public void setId( String custId );
  public void setAddress( String address );

  public String getName();
  public String getId();
  public String getAddress();
}

// 空の CustomerRecord インスタンスを作成し、Interaction を実行して
// 得られた出力を格納します。
CustomerRecord customer = // ... インスタンスを作成します。

// Interaction への入力として PurchaseOrderRecord インスタンスを作成し、
// このインスタンスにプロパティを設定します。このほかに、カスタムレコードの例
// として、PurchaseOrderRecord があります。
PurchaseOrderRecord purchaseOrder = // ... インスタンスを作成します。
purchaseOrder.setProductName( "..." );
purchaseOrder.setQuantity( "..." );
// ...

// 出力 CustomerRecord インスタンスを格納する Interaction を実行します。
boolean crResponse = ix.execute( rsIxSpec, purchaseOrder, customer  );

// CustomerRecord をチェックします。
System.out.println( "Customer Name = [" + customer.getName() + "],
                     Customer ID = [" + customer.getId() + "],
                     Customer Address = [" + customer.getAddress() + "]" );
```

アプリケーションコンポーネントのデプロイメントデスクリプタ

アプリケーションコンポーネントデプロイメントデスクリプタでは、コンポーネントが使用するリソースアダプタの接続ファクトリ情報を指定する必要があります。次のデプロイメントデスクリプタに適切なエントリが必要です。

1. コンポーネントの Sun 標準デプロイメントデスクリプタ。たとえば、ejb-jar.xml では、次のエントリが必要です。
   - `res-ref-name: shme/shmeAdapter`
   - `res-type: javax.resource.cci.ConnectionFactory`
   - `res-auth: Application|Container`
2. また、バージョン固有のエントリを含めることもできます。たとえば、EJB 2.0 の res-sharing-scope には、次のエントリを入れることができます。
   - `res-sharing-scope: Shareable|Unshareable`
3. コンポーネントの Borland 固有デプロイメントデスクリプタ。たとえば、ejb-borland.xml では、次のエントリが必要です。
   - `res-ref-name: shme/shmeAdapter`
   - `res-type: javax.resource.cci.ConnectionFactory`
4. また、バージョン固有のエントリを含めることもできます。たとえば、EJB 1.1 の cmp-resource には、次のエントリを入れることができます。
   - `cmp-resource: True|False`

次に、2 つの EJB 用のデプロイメントデスクリプタの詳細なサンプルを示します。最初のサンプルは、EJB 2.0 仕様に、2 番めのサンプルは EJB 1.1 仕様に準拠して記述されています。標準的なデプロイメントデスクリプタと Borland 固有のデプロイメントデスクリプタの両方のサンプルを示します。これらのサンプルでは、架空のリソースアダプタを参照しています。

EJB 2.x 用サンプル

ejb-jar.xml デプロイメントデスクリプタ

このサンプルでは、コンテナ管理の永続性を使用します。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
    <display-name>SHME Integration Jar</display-name>
    <enterprise-beans>
        <session>
            <description>Interface EJB for shmeAdapter Class /shme/test/
              shmeAdapter/schema/Customer</description>
            <display-name>customer_bean</display-name>
            <ejb-name>shme/customer_bean</ejb-name>
            <home>com.shme.test.shmeAdapter.schema.CustomerHome</home>
            <remote>com.shme.test.shmeAdapter.schema.CustomerRemote</remote>
            <ejb-class>com.shme.test.shmeAdapter.schema.CustomerBean
              </ejb-class>
            <session-type>Stateful</session-type>
            <transaction-type>Container</transaction-type>
            <env-entry>
                <description>コネクタ設定用の SHME リポジトリ URL
                   </description>
                <env-entry-name>repositoryUrl</env-entry-name>
                <env-entry-type>java.lang.String</env-entry-type>
                <env-entry-value>s_repository://S_APP01</env-entry-value>
            </env-entry>
            <env-entry>
                <description>Location of Resource Adapter Configuration within
                   the SHME Repository</description>
```

```
                <env-entry-name>configurationUrl</env-entry-name>
                <env-entry-type>java.lang.String</env-entry-type>
                <env-entry-value>/shme/client</env-entry-value>
            </env-entry>
            <resource-ref>
                <description>Reference to SHME Resource Adapter</description>
                <res-ref-name>shme/shmeAdapter</res-ref-name>
                <res-type>com.shme.shmeAdapter.ConnectionFactory</res-type>
                <res-auth>Container</res-auth>
                <res-sharing-scope>Shareable</res-sharing-scope>
            </resource-ref>
    </session>
    </enterprise-beans>
    <assembly-descriptor>
        <container-transaction>
            < メソッド >
                <ejb-name>customer_bean</ejb-name>
                <method-intf>Remote</method-intf>
                <method-name>s_exec_customer_query</method-name>
                <method-params/>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

次のサンプルは、前述の ejb-jar.xml に対応します。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Borland Software Corporation//DTD Enterprise
   JavaBeans 2.0//EN" "http://www.borland.com/devsupport/appserver/dtds/
   ejb-jar_2_0-borland.dtd">
<ejb-jar>
    <enterprise-beans>
        <session>
            <ejb-name>shme/customer_bean</ejb-name>
            <bean-home-name>shme/customer_bean</bean-home-name>
            <resource-ref>
                <res-ref-name>shme/shmeAdapter</res-ref-name>
                <jndi-name>eis/shmeAdapter</jndi-name>
            </resource-ref>
        </session>
    </enterprise-beans>
</ejb-jar>
```

EJB 1.1 用サンプル

ejb-jar.xml デプロイメントデスクリプタ

このサンプルでは、Bean 管理の永続性を使用します。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
   1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>
<ejb-jar>
 <description />
     <display-name>ShmeAdapter Interface Jar</display-name>
     <small-icon />
  <large-icon />
     <enterprise-beans>
      <session>
  <description>SHME クラス /shme/test/shmeAdapter/schema/ のインターフェース
EJB
     Customer</description>
   <display-name>customer_bean</display-name>
  <ejb-name>shme/customer_bean</ejb-name>
 <home>com.shme.test.shmeAdapter.schema.CustomerHome</home>
  <remote>com.shme.test.shmeAdapter.schema.CustomerRemote</remote>
   <ejb-class>com.shme.test.shmeAdapter.schema.CustomerBean</ejb-class>
   <session-type>Stateless</session-type>
   <transaction-type>Bean</transaction-type>
   <env-entry>
        <description> コネクタ設定用の SHME リポジトリ URL
```

```
          </description>
       <env-entry-name>repositoryUrl</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
   <env-entry-value>s_repository://S_APP01</env-entry-value>
    </env-entry>
  <env-entry>
   <description>SHME リポジトリ内のリソースアダプタ設定の場所 </description>
      <env-entry-name>configurationUrl</env-entry-name>
   <env-entry-type>java.lang.String</env-entry-type>
   <env-entry-value>/shme/client</env-entry-value>
   </env-entry>
  <resource-ref>
   <description>Reference to SHME Resource Adapter</description>
  <res-ref-name>shme/shmeAdapter</res-ref-name>
  <res-type>com.shme.shmeAdapter.ConnectionFactory</res-type>
  <res-auth>Container</res-auth>
  </resource-ref>
    </session>
 </enterprise-beans>
 <ejb-client-jar />
 </ejb-jar>
```

ejb-inprise.xml デプロイメントデスクリプタ

次のサンプルは、前述の ejb-jar.xml に対応します。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE inprise-specific PUBLIC '-//Inprise Corporation//DTD Enterprise
   JavaBeans 1.1//EN' 'http://www.borland.com/devsupport/appserver/dtds/
    ejb-inprise.dtd'>
<inprise-specific>
  <enterprise-beans>
    <session>
      <ejb-name>shme/customer_bean</ejb-name>
      <bean-home-name>shme/customer_bean</bean-home-name>
      <timeout>0</timeout>
      <resource-ref>
        <res-ref-name>shme/shmeAdapter</res-ref-name>
        <jndi-name>eis/shmeAdapter</jndi-name>
        <cmp-resource>False</cmp-resource>
      </resource-ref>
    </session>
  </enterprise-beans>
</inprise-specific>
```

その他の考慮事項

インプリメンテーションが貧弱なリソースアダプタでの作業

市販されているリソースアダプタの中には、インプリメンテーションが貧弱なものがあります。J2EE Compatibility Test Suite（CTS）は、コネクタのインプリメンテーションが仕様に準拠しているかどうかをテストしますが、リソースアダプタがコネクタ仕様に準拠しているかどうかをテストするためのメカニズムはまだ存在していないため、現在、それを判断するのは、簡単ではありません。しかし、次に示すようないくつかの症状から、リソースアダプタがコネクタ仕様に準拠していないと判断できます。

1. リソースアダプタがデプロイメント時に不明なエラーを生成する
2. 接続ファクトリでのメソッド呼び出し中にリソースアダプタが不明なエラーを生成する。

VisiConnect は、J2EE 1.4 とコネクタ 1.5 要件を厳密に実装するため、インプリメンテーションが貧弱なリソースアダプタを検出して問題を無視しないのは、コネクタコンテナだけである場合があります。

インプリメンテーションが貧弱なリソースアダプタの例

一般に、インプリメンテーションが貧弱なリソースアダプタは、コネクタ 1.5 仕様に準拠していません。このようなリソースアダプタの例を次に示します。

- `java.io.Serializable` だけを実装し、コネクタ仕様（セクション 10.5「JNDI Configuration and Lookup」）で定められている `java.io.Serializable` と `javax.resource.Referenceable` の両方の実装は行っていない接続ファクトリを持つリソースアダプタ。AppServer などのアプリケーションサーバーのローカル JNDI コンテキスト ハンドラは、これらのインターフェースを両方とも実装する場合にのみ、オブジェクトを登録できます。リソースアダプタが接続ファクトリを `Serializable` として実装し、`Referenceable` を実装しない場合、アプリケーションサーバーが接続ファクトリを JNDI にデプロイメントしようとすると、例外が発生します。
- `javax.resource.Referenceable`（`javax.naming.Referenceable` から `getReference()` を継承する）を正しく実装していない接続ファクトリを持つリソースアダプタ。J2SE 1.3.x および 1.4.x 仕様では、`javax.naming.Referenceable` に対して `getReference()` が次のいずれかを実行することが指定されています。

  a. `Referenceable` オブジェクトの有効な null 以外のリファレンスを返す

  b. 例外（`javax.naming.NamingException`）を生成する

  `Referenceable` が `null` を返すような `Referenceable` をリソースアダプタが実装している場合、クライアントが `getConnection()` のような接続ファクトリメソッドを呼び出そうとすると、例外が発生します。
- `Referenceable` を正しく実装しているが、`javax.naming.spi.ObjectFactory` のインプリメンテーションを提供しない接続ファクトリを持つリソースアダプタ。`javax.naming.spi.ObjectFactory` のインプリメンテーションは、コネクタ仕様（セクション 10.5「JNDI Configuration and Lookup」）で要求されています。このようなリソースアダプタは、アプリケーションサーバーに問題なくデプロイメントできますが、アプリケーションサーバーの管理外にある JNDI に非管理対象のコネクタとしてデプロイメントすることはできません。また、JNDI の `Reference` ベースの接続ファクトリ検索をバックアップするメカニズムを持つ `javax.naming.spi.ObjectFactory` インプリメンテーションソースアダプタも含まれます。

- 接続ファクトリまたは接続インターフェースを指定しているが、接続ファクトリまたは接続クラス内でそのインターフェースを実装していないリソースアダプタ。関連する要件については、コネクタ仕様のセクション 10.6「Resource Adapter XML DTD」を参照してください。たとえば、特定のリソースアダプタの `ra.xml` に次の要素があるとします。

```
//...
<connection-interface>java.sql.Connection</connection-interface>
<connection-impl-class>com.shme.shmeAdapter.ShmeConnection</connection-impl-
class>
//...
```

  しかし、`ShmeConnection` のインプリメンテーションは次のようになっています。

```
package shme;
public class ShmeConnection
{
private ShmeManagedConnection mc;
  public ShmeConnection( ShmeManagedConnection mc )
  {
   System.out.println( "In ShmeConnection" );
   this.mc = mc;
  }
}
```

  このリソースアダプタの接続ファクトリで `getConnection()` を呼び出そうとすると、`java.lang.ClassCastException` が発生します。これは、AppServer に対して、`ra.xml` で、リソースアダプタから返される接続オブジェクトが `java.sql.Connection` にキャストされることを指定しているためです。

貧弱なリソースアダプタインプリメンテーションでの作業

貧弱なリソースアダプタインプリメンテーションで作業する場合は、次のような処理を実行します。

コネクタの接続ファクトリまたは接続クラスを拡張し、インプリメンテーションが貧弱なコードを正しく実装させます。たとえば、`Serializable` は実装しているが `Referenceable` は実装していない接続ファクトリを取り扱う場合は、元の接続ファクトリを拡張して `Referenceable` を実装させます。つまり、`getReference()` と `setReference()` の両方を実装させます。

例として、接続ファクトリが `com.shme.BadConnectionFactory` である場合に、接続ファクトリを `com.shme.GoodConnectionFactory` として拡張し、`Referenceable` を実装するサンプルを次に示します。

```
package com.shme.shmeAdapter;

public class GoodConnectionFactory
{
 private javax.naming.Reference ref;
// ...
 public javax.naming.Reference getReference()
{
// getReference() が null を返さないように実装します。
// ...
  return ref;
    }
public javax.naming.Reference setReference( javax.naming.Reference ref )
 //  this.ref = ref;
  }
 //
```

また、機能が貧弱な `getReference()` に対処する手段はさまざまありますが、それらの主な目的は、`null` を絶対に返さないような `getReference()` を実装することです。最もよい方法は、次のインプリメンテーションを行うことです。

- `getReference()` でフォールバックメカニズムを実装します。これは、接続ファクトリのリファレンス属性が `null` である場合、リファレンスを設定して正しく返されるようにします。つまり、登録可能な javax.naming.Reference オブジェクトを返すようにします。

- `javax.naming.spi.ObjectFactory` を実装するヘルパークラスを実装します。これにより、フォールバックオブジェクトを提供して有効な Reference インスタンスから接続ファクトリオブジェクトを作成できます。

例として、接続ファクトリが `com.shme.BadConnectionFactory` である場合に、接続ファクトリを `com.shme.GoodConnectionFactory` として拡張し、`getReference()` をオーバーライドするサンプルを次に示します。

```
package com.shme.shmeAdapter;

public class GoodConnectionFactory
{
  // ...

  public javax.naming.Reference getReference()
  {
    if ( ref == null )
    {
      ref = new javax.naming.Reference( this.getClass().getName(),
        "com.shme.shmeAdapter.GoodCFObjectFactory"
          /* GoodConnectionFactory リファレンスのオブジェクトファクトリ */,
           null );
      String value;
      value = managedCxFactory.getClass().getName();

      if ( value != null )
      {
        ref.add( new javax.naming.StringRefAddr(
          "managedconnectionfactory-class", value ) );
      }

      value = cxManager.getClass().getName();

      if ( value != null )
      {
        ref.add( new javax.naming.StringRefAddr(
          "connectionmanager-class", value ) );
      }
    }

    return ref;
  }

  // ...
}
```

次に、関連するオブジェクトファクトリクラスを実装します。この例では、コードは次のようになります。

com.shme.shmeAdapter.GoodCFObjectFactory

```
package com.shme.shmeAdapter;

import javax.naming.spi.*;
import javax.resource.spi.*;

public class GoodCFObjectFactory implements ObjectFactory {
  public GoodCFObjectFactory() {};

  public Object getObjectInstance( Object obj,
                                   javax.naming.Name name,
                                   javax.naming.Context context,
                                   java.util.Hashtable env )
        throws Exception
  {
    if ( !( obj instanceof javinstance ofReference ) )
    {
      return null;
    }

    javax.naming.Reference ref = (javax.naming.Reference)obj;

    if ( ref.getClassName().equals(
"com.shme.shmeAdapter.GoodConnectionFactory" ) )
    {
      ManagedConnectionFactory refMcf = null;
      ConnectionManager refCm = null;

if ( ref.get( "managedconnectionfactory-class" ) != null )
      {
        String managedCxFactoryStr =
         (String)ref.get( "managedconnectionfactory-class" ).getContent();
        Class mcfClass = Class.forName( managedCxFactoryStr );
        refMcf = (ManagedConnectionFactory)mcfClass.newInstance();
      }

if ( ref.get( "connectionmanager-class" ) != null )
      {
        String cxManagerStr = (String)ref.get( "connectionmanager-class"
).getContent();
        Class cxmClass = Class.forName( cxManagerStr  );
        java.lang.ClassLoader cloader = cxmClass.getClassLoader();
        refCm = (ConnectionManager)cxmClass.newInstance();
      }

      GoodConnectionFactory cf = null;

      if ( refCm != null )
      {
        cf = new GoodConnectionFactory( refMcf, refCm  );
      }
      else
      {
        cf = new GoodConnectionFactory( refMcf );
      }

    return cf;
    }

    return null;
  }
}
```

ra.xml 標準デプロイメントデスクリプタファイルでクラスを更新します。たとえば、インプリメンテーション拡張前は、`ra.xml` は次のようになっています。

```
<managedconnectionfactory-class>com.shme.shmeAdapter.
LocalTxManagedConnectionFactory</managedconnectionfactory-class>
<connectionfactory-interface>javax.sql.DataSource</connectionfactory-interface>
<connectionfactory-impl-class>com.shme.shmeAdapter.BadConnnectionFactory</
connectionfactory-impl-class>
<connection-interface>java.sql.Connection</connection-interface>
<connection-impl-class>com.shme.Connection</connection-impl-class>
```

インターフェース拡張後は、ra.xml は次のようになります。

```
<managedconnectionfactory-class>com.shme.shmeAdapter.
LocalTxManagedConnectionFactory </managedconnectionfactory-class>
<connectionfactory-interface>javax.sql.DataSource</connectionfactory-interface>
<connectionfactory-impl-class>com.shme.shmeAdapter.GoodConnectionFactory</
connectionfactory-impl-class>
<connection-interface>java.sql.Connection</connection-interface>
<connection-impl-class>com.shme.shmeAdapter.Connection</connection-impl-class>
```

このサンプルが示すように、この変換は接続ファクトリだけに影響します。他のリソースアダプタクラスは、この変換による影響を受けません。

拡張されたインプリメンテーション（およびすべてのヘルパークラス）の Java コードをクラスファイルにコンパイルします。

それらをリソースアダプタの Java アーカイブファイル（.jar）にパッケージします。

この拡張された .jar でリソースアダプタアーカイブファイル（.rar）を更新します。

スタンドアロンで、または AppServer 内のパーティションサービスとして実行している VisiConnect に、リソースアダプタアーカイブをデプロイメントします。または、リソースアダプタアーカイブをエンタープライズアプリケーションアーカイブファイル（.ear）に含めて J2EE アプリケーションの一部としてデプロイメントします。

これで、動作が不適切なリソースアダプタを適切に動作するリソースアダプタに変換できました。

リソースアダプタの設計によっては、既存の API インプリメンテーションを拡張できない場合もあります。そのような場合は、問題のあるクラスを再実装し、`ra.xml` で再実装したクラスを参照するように要素を設定する必要があります。あるいは、いっそのこと、コネクタ仕様に準拠している別のリソースアダプタを選択します。

第 35 章

Borland AppServer Ant タスクと AppServer サンプルの実行

多くの Borland AppServer（AppServer）サンプルで、Ant ビルドスクリプトシステムが使用されるようになりました。Ant の Borland AppServer バージョンには、AppServer の一部のコマンドラインツール用として、Ant のコア機能以外に次のコマンドラインを含むいくつかのカスタマイズされたタスクが含まれます。

- `appclient`
- `iastool`
- `idl2java`
- `java2iiop`

これらのカスタマイズされた Ant タスクには、`exec` または `apply` 指示文を使用する場合に比べて、次の利点があります。

- カスタマイズされた Ant タスクは、Ant スクリプトを実行するために使用される VM で実行されるため、`exec`/`apply` コマンドを使って新しい JVM を生成する場合に比べて、実行速度が速く、メモリの消費量も少なくなります。
- カスタマイズされたタスクのコマンド構文は、exec/apply バージョンのコマンド構文より大幅に単純化されています。
- Ant が備えている filesets や patternsets などの機能は、より自然な形式で使用できます。

一般構文と使い方

次の表は、現在定義されている Ant タスクと対応するコマンドラインツールとの関係を示します。

Ant タスク名	対応するコマンドライン	機能
appclient	appclient	クライアントアプリケーションを実行します。
idl2java	idl2java	IDL を Java クラスに変換します。
java2iiop	java2iiop	java2iiop コマンドを実行します。
iastool	iastool	iastool を実行します。

一般に AppServer Ant タスクは、対応するコマンドラインツールと同じパターンを使用します。

名前／値ペアの変換

すべての名前／値ペアのコマンドライン引数は、Ant タスク属性に変換できます。名前／値ペアのコマンドライン引数は、対応する XML 属性に変換する必要があります。たとえば、次のコマンドラインの場合：

```
iastool -verify -src cart_beans_client.jar -role DEVELOPER
```

次の Ant タスクに変換されます。

```
<iastool option="verify" src="cart_beans_client.jar" role="DEVELOPER" />
```

名前のみの引数の変換

名前のみのコマンドライン引数はすべて、boolean 型の Ant タスク属性に変換できます。-nowarn などの boolean 型属性については、各コマンドラインツールの使い方の説明にしたがって属性のデフォルトの使用方法を使用します。iastool コマンドライン属性の詳細については、315 ページの「iastool コマンドライン ユーティリティ」を参照してください。

たとえば、次のコマンドは、warn 属性を false に設定します。

```
iastool -verify -src cart_beans_client.jar -role DEVELOPER -nowarn -nostrict
```

対応する Ant タスクは、次のとおりです。

```
<iasverify src="cart_beans_client.jar" role="DEPLOYER" nowarn="true"
strict="false" />
```

メモ Ant タスクで "warn" を属性として使用することは**できません**。たとえば、次のラインでは構文エラーが生成されます。

```
****** INCORRECT SYNTAX!!! ******
<iasverify src="cart_beans_client.jar" role="DEPLOYER" warn="false"
strict="false" />
```

複数ファイルの引数

多くのコマンドは、複数のファイルを処理できるか、複数のファイルを指定できるオプションがあります。対応する Ant タスクでこの機能を実現する方法には、いくつかの種類があります。たとえば、次の `iastool -merge` コマンドの場合：

```
iastool -merge -target build¥client.jar -type lib client¥build¥local_client.jar
build¥local_stubs.jar
```

このコマンドに対応する Ant タスクは、次のとおりです。

```
<iastool option="merge" target="${build.dir}/client.jar" type="lib"
 jars="client/build/local_client.jar ; build/local_stubs.jar" />
```

メモ `jars` 属性では、複数のファイルはセミコロン（;）またはコロン（:）で区切る必要があります。スペースとカンマは区切りとしては**無効**です。

Ant には、複数のファイルを指定するための便利な `<fileset>` タスクがあります。

```
<iastool option="merge" target="build/client.jar" type="lib" >
    <fileset dir="client/build" includes="local_client.jar" />
    <fileset dir="build" includes="local_stubs.jar" />
</iastool>
```

また、Ant の patternset 機能も便利です。次のように変更すると、build ディレクトリとそのサブディレクトリ内にあるすべての jar ファイルが対象になります。

```
<iastool option="merge" target="${build.dir}/client.jar" type="lib" >
    <fileset dir="${build.dir}" includes="**/*.jar" />
</iastool>
```

クラスパス属性には、複数のパスをセミコロンで区切って指定できます。

```
<iastool option="verify" src="cart_beans_client.jar" role="DEPLOYER"
classpath="alib.jar;blib.jar" />
```

または、次のように `<classpath>` 要素を使用します。

```
<iastool option="verify" src="cart_beans_client.jar" role="DEPLOYER" >
   <classpath>
       <pathelement location="alib.jar" />
```

```
        <pathelement location="blib.jar" />
    </classpath>
</iastool>
```

iastool の構文と使い方

iastool Ant タスクは、次の 2 つのスタイルを使用できます。

1 <iastool option="*myoption*" />

2 <ias*myoption* />

たとえば、次のコマンドラインの場合：

```
iastool -verify -src cart_beans_client.jar
```

次の Ant タスクに変換されます。

```
<iastool option="verify" src="cart_beans_client.jar" />
```

または、下位互換の Ant タスク用に古い Ant スタイルを使用できます。

```
<iasverify src="cart_beans_client.jar" />
```

次の表に、各 iastool オプションの Ant タスクスタイルを示します。

iastool Ant タスクスタイル 1	iastool Ant タスクスタイル 2	対応するコマンドライン	機能	Fileset 属性
<iastool option="compilejsp" />	<iascompilejsp />	iastool -compilejsp	JSP をプリコンパイルします。	
<iastool option="compress" />	<iascompress />	iastool -compress	JAR ファイルを圧縮します。	
<iastool option="deploy" />	<iasdeploy />	iastool -deploy	J2EE モジュールをデプロイメントします。	JAR
<iastool option="dumpstack" />	<iasdumpstack />	iastool -dumpstack	パーティションプロセスのスタックトレースを次の場所の stdout.log にダンプします。 <install_dir>/var/domains/<domain-name>/configurations/<configuration-name>/mos/<partition-name>/adm/logs/<partition_name>.stdout.log	
<iastool option="genclient" />	<iasgenclient />	iastool -genclient	クライアントライブラリを生成します。	JAR
<iastool option="gendeployable" />	<iasgendeployable />	iastool -gendeployable	手動でデプロイメント可能なモジュールを生成します。	
<iastool option="genstubs" />	<iasgenstubs />	iastool -genstubs	スタブライブラリを生成します。	
<iastool option="info" />	<iasinfo />	iastool -info	システム設定情報を表示します。	
<iastool option="kill" />	<iaskill />	iastool -kill	管理オブジェクトを強制終了します。	
<iastool option="listhubs" />	<iaslisthubs />	iastool -listhubs	管理ポートで使用できるハブをリストします。	
<iastool option="listpartitions" />	<iaslistpartitions />	iastool -listpartitions	ハブで実行されているパーティションをリストします。	

iastool Ant タスクスタイル 1	iastool Ant タスクスタイル 2	対応するコマンドライン	機能	Fileset 属性
<iastool option="listservices" />	<iaslistservices />	iastool -listservices	ハブで実行されているサービスをリストします。	
<iastool option="manage" />	<iasmanage />	iastool -manage	管理オブジェクトをアクティブに管理します。	
<iastool option="merge" />	<iasmerge />	iastool -merge	JAR ファイルのセットを 1 つの JAR ファイルにマージします。	JAR
<iastool option="migrate" />	<iasmigrate />	iastool -migrate	モジュールを J2EE 1.2 から J2EE 1.3 に移行します。	
<iastool option="newconfig" />	<iasnewconfig/>	iastool -newconfig	設定を新規作成します。	
<iastool option="patch" />	<iaspatch />	iastool -patch	JAR ファイルにパッチを適用します。	
<iastool option="ping" />	<iasping />	iastool -ping	管理オブジェクトまたはハブに対して ping を実行して、 現在の状態を取得します。	
<iastool option="pservice" />	<iaspservice />	iastool -pservice	パーティションサービスを有効／無効にするか、 状態を取得します。	
<iastool option="removestubs" />	<iasremovestubs />	iastool -removestubs	JAR からスタブを除去します。	
<iastool option="restart" />	<iasrestart />	iastool -restart	管理オブジェクトを再起動します。	
<iastool option="setmain" />	<iassetmain />	iastool -setmain	EAR のクライアント JAR または JAR のメインクラスを設定します。	
<iastool option="stop" />	<iasstop />	iastool -stop	管理オブジェクトを停止します。	
<iastool option="uncompress" />	<iasuncompress />	iastool -uncompress	JAR ファイルを解凍します。	
<iastool option="undeploy" />	<iasundeploy />	iastool -undeploy	管理オブジェクトのデプロイメントを解除します。	
<iastool option="unmanage" />	<iasunmanage />	iastool -unmanage	アクティブ管理から管理オブジェクトを削除します。	
<iastool option="verify" />	<iasverify />	iastool -verify	J2EE モジュールを検証します。	

メモ 「Fileset 属性」列には、複数のファイル名を適用できる属性が示されています。このような属性では、Ant `<fileset>` 要素を使って複数のファイルを指定できます。複数のファイルを指定する方法については、306 ページの「複数ファイルの引数」を参照してください。

属性の省略

Ant タスク呼び出しで属性を省略することは、コマンドラインツールでオプションを省略することと同等です。いくつかの属性はデフォルトが `true` なので、属性を省略しても `false` に設定されるとは**限りません**。

これらのオプションのデフォルト値の詳細については、315 ページの「iastool コマンドライン ユーティリティ」を参照してください。

iastool Ant タスクのサンプル

次に、iastool Ant タスクの使い方を詳しく説明するサンプルを示します。それぞれの iastool オプションと属性の機能の詳細については、315 ページの「iastool コマンドライン ユーティリティ」を参照してください。

deploy

```
<target name="deploy" description=" サンプルをサーバーにデプロイメントします ">
<iastool option="deploy" hub="${hub.name}" cfg="${cfg.name}"
 partition="${partition.name}" mgmtport="${default.mgmtport}"
 jars="${build.dir}/hello.ear;${bes.lib.dir}/../var/repository/archives/wars/
   bank_form.war"
 realm="${realm.name}" user="${server.user.name}" pwd="${server.user.pwd}" />
</target>
```

merge

```
<iastool option="merge" target="${build.dir}/helloclient.jar" type="lib">
    <fileset dir="${build.dir}" includes="hello_stubs.jar" />
</iastool>
```

ping

```
<target name="ping">
<iastool option="ping" hub="${hub.name}" cfg="${cfg.name}"
 partition="${partition.name}" mgmtport="${default.mgmtport}"
 realm="${realm.name}" user="${server.user.name}" pwd="${server.user.pwd}" />
</target>
```

restart

```
<target name="iastoolrestart">
<iastool option="-restart" hub="${hub.name}" cfg="${cfg.name}"
 partition="${partition.name}" mgmtport="${default.mgmtport}"
 realm="${realm.name}" user="${server.user.name}" pwd="${server.user.pwd}" />
</target>
```

java2iiop の構文と使い方

java2iiop Ant タスクは対応するコマンドラインツールと大きく異なります。これは、Borland Ant タスクの使用パターンの例外です。Ant タスク java2iiop は、個々のファイルのかわりにディレクトリのクラスをとります。クラスパス属性は、java2iiop でコンパイルするクラスが存在するディレクトリをポイントします。クラスパスは Ant のパスに似た構造で、使い方は非常に柔軟ですが、java2iiop タスクでクラスパスを使用する場合は、次のいずれかのスタイルだけを使用できます。

1 属性として使用する場合、値はコロンまたはセミコロンで区切られたロケーションのリストだけを受け入れます。

```
<java2iiop classpath="${path1}:${path2}"/>
```

2 ネストされたクラスパス要素として使用する場合：
次のような一般的な形式になります。

```
<java2iiop>
   <classpath>
      <pathelement path="${path1}"/>
      <pathelement location="lib/helper.jar"/>
   </classpath>
</java2iiop>
```

ロケーション属性はプロジェクトのベースディレクトリに基づいて 1 つのファイルまたはディレクトリ（または絶対ファイル名）を指定しますが、パス属性はコロンまたはセミコロンで区切られたロケーションのリストを受け入れます。パス属性は定義済みのパスとともに使用することを目的としています。その他の場合は、ロケーション属性を持つ複数の要素を使用するようにしてください。

java2iiop の対応するコマンドライン引数の詳細については、『VisiBroker® for Java 開発者ガイド』を参照してください。

java2iiop Ant タスクのサンプル

```
<target name="create_ejb_stubs" depends="home">
    <java2iiop root_dir="${stubsPath}" list_files="true"
classpath="${outputPath}" />
</target>
```

idl2java の構文と使い方

idl2java Ant タスクは対応するコマンドラインツールに似ています。このタスクは、ネストされたパスのような構造のファイルセットを受け取ります。このファイルセットは、コマンドラインのファイル入力に対応します。

```
<idl2java>
    <fileset dir="server" includes="*.idl" />
</idl2java>
```

idl2java の対応するコマンドライン引数の詳細については、『VisiBroker® for Java 開発者ガイド』を参照してください。

属性	型	必須
classpath	Path	はい
back_Compat_Mapping	boolean	いいえ
bind	boolean	いいえ
boa	boolean	いいえ
comments	boolean	いいえ
compile	boolean	いいえ
compiler	String	いいえ
destDir	Path	いいえ
dynamic_Marshal	boolean	いいえ
examples	boolean	いいえ
export_All	boolean	いいえ
exported	String	いいえ
gen_Included_Files	boolean	いいえ
idl2package	String	いいえ
idl_Strict	boolean	いいえ
import	String	いいえ
imported	String	いいえ
include	File	いいえ
invoke_Handler	boolean	いいえ
line_Directives	boolean	いいえ
list_Files	boolean	いいえ
list_Includes	boolean	いいえ
map_Keyword	String	いいえ
narrow_Compliance	boolean	いいえ
obj_Wrapper	boolean	いいえ
object_Method	boolean	いいえ
package	String	いいえ
retain_Comments	boolean	いいえ
root_Dir	File	いいえ
sealed	String	いいえ
servant	boolean	いいえ
srcDir	Path	いいえ
srcFile	String	いいえ
stream_Marshal	boolean	いいえ
strict	boolean	いいえ
tie	boolean	いいえ
undefine	String	いいえ
VBJclassPath	Path	いいえ

属性	型	必須
VBJdebug	String	いいえ
VBJjavaVM	File	いいえ
VBJprop	String	いいえ
VBJquoteSpaces	String	いいえ
VBJtag	String	いいえ
version	String	いいえ
warn_Missing_Define	String	いいえ
warn_Unrecognized_Pragmas	boolean	いいえ

idl2java Ant タスクのサンプル

```
<target name="idl2java" depends="init">
    <idl2java package="com.borland.examples.webservices.visibroker"
root_dir="${server-skel-src}">
        <fileset dir="server" includes="*.idl" />
    </idl2java>
    <javac srcdir="${server-skel-src}" destdir="${server-classes}"
classpathref="classpath"/>
</target>
```

appclient の構文と使い方

```
<!-- サンプルを実行します。-->
<target name="execute" description="「Hello World」サンプルを実行します ">
    <appclient jar="${build.dir}/hello.ear" uri="helloclient.jar" args="World"/
>
</target>
```

Borland AppServer サンプルのビルドと実行

メモ 多くの AppServer サンプルでは、次の場所に独自の `readme.html` ファイルが置かれています。

```
<install_dir>/examples
```

次の手順で AppServer サンプルをビルドします。

1. コマンドラインウィンドウを開きます。
2. 現在のディレクトリをサンプルディレクトリに設定します。`<install_dir>/examples/j2ee/hello` に置かれている「Hello World」サンプルから開始することをお勧めします。
3. コマンドラインで、ant と入力します。

   サンプルが自動的にビルドされます。

メモ サンプルをビルドするために、サーバーが実行中である必要はありません。ただし、デプロイメントとデプロイメント解除には、サーバーが実行中である必要があります。サンプルを実行するには、パーティションが実行中である必要があります。

サンプルのデプロイメント

1 サーバーが実行されていることを確認します。

2 コマンドラインで、`ant deploy` と入力します。

これで、<install_dir>¥examples¥deploy.properties ファイルで設定されているハブ、設定、およびパーティションにサンプルがデプロイメントされます。

複数のハブ／設定／パーティションの組み合わせにデプロイメントする場合は、`deploy.properties` ファイルを編集して設定を変更するか、コマンドラインで `-D` オプションを使って `deploy.properties` 設定を上書きします。

たとえば、「myhub」という名前のハブを使用するには、次のコマンドを使用します。

```
ant -Dhub.name=myhub deploy
```

これで、`deploy.properties` というデフォルトのハブ名が `myhub` 値で上書きされます。

サンプルの実行

1 パーティションが実行されていることを確認します。

2 コマンドラインで、`ant execute` と入力します。

応答の細部は、サンプルごとに異なります。

サンプルのデプロイメント解除

1 サーバーが実行されていることを確認します。

2 コマンドラインで、`undeploy` と入力します。

トラブルシューティング

1 `<appserver_install_dir>/bin` ディレクトリがパスに含まれており、ほかのすべての Ant インストールより優先するように指定されていることを確認します。

2 `ant execute` コマンドを呼び出す前に、サーバーとパーティションが実行されていることを確認します。

3 `<appserver_install_dir>¥examples¥deploy.properties` には、ハブ、設定、パーティション、および管理ポートのデフォルト設定が含まれます。これらのデフォルトプロパティには、次のようなものがあります。

- `hub.name=`*`your_machine_name`*
- `cfg.name=j2ee`
- `partition.name=standard`
- `realm.name=ServerRealm`
- `server.user.name=admin`
- `server.user.pwd=admin`

your_machine_name は、インストール時に指定されたマシン名です。これらの値は、必要に応じてリセットしたり、Ant コマンドラインで `-D` オプションを使って指定することができます。

第 36 章

iastool コマンドライン ユーティリティ

ここでは、管理オブジェクトを管理するために使用できる `iastool` コマンドライン ユーティリティについて説明します。

iastool コマンドライン ツールの使い方

`iastool` ユーティリティは、管理オブジェクトを操作するためのコマンドライン ユーティリティの総称です。`iastool` では、次のコマンドライン ユーティリティを使用できます。

使用 ...	目的 ...
-clonepartition	既存のパーティションのクローンを作成し、新しいパーティションにします。詳細は、「clonepartition」を参照してください。
-compilejsp	スタンドアロンの WAR または EAR 内のすべての WAR にある JSP をプリコンパイルします。詳細は、「compilejsp」を参照してください。
-compress	JAR ファイルを圧縮します。詳細は、「compress」を参照してください。
-deploy	J2EE モジュールを指定したパーティションにデプロイします。詳細は、「deploy」を参照してください。
-dumpstack	パーティション プロセスのスタック トレースをパーティションの `stdout.log` ファイルにダンプします。詳細は、「dumpstack」を参照してください。
-genclient	クライアント スタブ、EJB インターフェイス、および依存クラスを含むライブラリを生成します。詳細は、「genclient」を参照してください。
-gendeployable	手動でデプロイメント可能なモジュールを生成します。詳細は、「gendeployable」を参照してください。
-gendeployable verify	手動でデプロイメント可能なモジュールを生成し、検証します。詳細は、「gendeployableverify」を参照してください。
-genstubs	クライアントまたはサーバー スタブだけを含むライブラリを生成します。詳細は、「genstubs」を参照してください。
-info	システム設定情報を表示します。詳細は、「info」を参照してください。
-jndinamespace	コンテキストにバインドされたオブジェクトのリストを表示します。詳細は、「jndinamespace」を参照してください。
-kill	管理オブジェクトを強制終了します。詳細は、「kill」を参照してください。
-listpartitions	ハブ上のパーティションを一覧表示します。詳細は、「listpartitions」を参照してください。

使用 ...	目的 ...
-listhubs	管理ポートで使用できるハブを一覧表示します。 詳細は、「listhubs」を参照してください。
-listservices	ハブ上のサービスを一覧表示します。 詳細は、「listservices」を参照してください。
-manage	管理オブジェクトをアクティブに管理します。 詳細は、「manage」を参照してください。
-merge	JAR ファイルセットを 1 つの JAR ファイルにマージします。 詳細は、「merge」を参照してください。
-migrate	J2EE 1.2、1.3、1.4 から別のバージョンのターゲット J2EE に移行します。詳細は、「migrate」を参照してください。
-newconfig	設定テンプレートから新規設定を作成します。 詳細は、「newconfig」を参照してください。
-patch	JAR ファイルに 1 つまたは複数のパッチを適用します。詳細は、「patch」を参照してください。
-ping	管理オブジェクトまたはハブに対して ping を実行して、 現在の状態を取得します。 詳細は、「ping」を参照してください。
-pservice	パーティション サービスを有効、無効にするか、状態を取得します。 詳細は、「pservice」を参照してください。
-removestubs	JAR ファイルからすべてのスタブ ファイルを削除します。 詳細は、「removestubs」を参照してください。
-restart	ハブまたは管理オブジェクトを再起動します。 詳細は、「restart」を参照してください。
-setmain	スタンドアロンのクライアント JAR、または EAR 内のクライアント JAR のメイン クラスを設定します。 詳細は、「setmain」を参照してください。
-start	管理オブジェクトを起動します。 詳細は、「start」を参照してください。
-stop	ハブまたは管理オブジェクトを停止します。 詳細は、「stop」を参照してください。
-uncompress	JAR ファイルを解凍します。詳細は、「uncompress」を参照してください。
-undeploy	パーティションから J2EE モジュールを削除します。 詳細は、「undeploy」を参照してください。
-unmanage	アクティブ管理から管理オブジェクトを削除します。詳細は、「unmanage」を参照してください。
-usage	コマンドライン オプションの使い方を表示します。詳細は、「usage」を参照してください。
-verify	J2EE モジュールを検証します。 詳細は、「verify」を参照してください。

clonepartition

既存のパーティションのクローンを作成し、新しいパーティションにします。このユーティリティは、Borland 管理コンソールでのクローン機能と同等です。新しいパーティションの名前は重複しないようにする必要があります。

構文

```
-clonepartition [-hub <hub name>] [-cfg <configuration name>] [-mgmtport
<99999>]\n  [-bare] [-realm <realm>] [-user <username>] [-pwd <password>] [-
file <login_file>]\n  -osagent <osagent port number> [-partition<name partition
to be cloned>]   [-newpartition<name of new partition>]\n [-newdesc
<description of new partition>] -newdisplayname<display name of new partition>
-newgroup<group name for creating the new partition like {hubname}/
{configurationname}> -moagent<agent name> [-targetagent <target agent name>] -
jmxport<jmx port number default:8082> -tomcatport<Tomcat port number
default:8080>
```

デフォルト出力
デフォルト値でパーティションを作成します。

オプション
compilejsp ツールで使用できるオプションを次の表に示します。

オプション	説明
-hub <hub>	実行中のパーティションを一覧表示するハブの名前を指定します。
-cfg <configname>	パーティションを一覧表示する設定の名前を指定します。
-mgmtport <nnnnn>	指定したハブで使用する管理ポート番号を指定します。 デフォルト値は、 42424 です。
-bare	実行中のパーティション名以外の出力情報をオフにします。
-realm <realm>	ユーザーとパスワードのオプションを指定したら、ユーザー認証に使用する領域を指定します。
-user <username>	指定した領域に対して認証するユーザーを指定します。
-pwd <password>	指定した領域に対して認証するユーザーのパスワードを指定します。
-file <login_file>	ユーザーを認証するために使用する領域、 ユーザー名、 およびパスワードを含むログイン スクリプトファイルを指定します。 このファイルがある場所のフル パスか相対パスが必要です。
-osagent	osagent ポートを指定します。
-partition	クローンを作成するパーティション名を指定します。
-newpartition	新しいパーティションの名前を指定します。
-newdesc	新しいパーティションの説明。
-newdisplayname	name パーティションの名前を表示します。
-newgroup	パーティションが作成されるグループ名（ハブ / 設定）を指定します。
-moagent	管理オブジェクトが関連付けられているエージェントを指定します。
-targetagent	作成が必要なパーティションがある対象のエージェントを指定します。
-jmxport	Java 管理コンソールのポート番号を指定します。デフォルト値は、 8082 です。
-tomcatport	Tomcat サービスのポート番号を指定します。 デフォルト値は、 8080 です。

例
この例では、クローン パーティションのオプションについて使用方法を示します。サンプルのクローンは "MyPartition" で、指定したポート番号、設定、エージェントで "VJ_partition" を作成します。

```
-clonepartition -hub gvijay -cfg j2eeSample -mgmtport 20001 -realm ServerRealm
-user admin -pwd admin -osagent 19999 -partition MyPartition -newpartition
VJ_partition  -newgroup gvijay/j2eeSample -moagent gvijay -jmxport 8084 -
tomcatport 8086
```

compilejsp

このツールを使用して、スタンドアロンの WAR または EAR 内のすべての WAR にある JSP ページをプリコンパイルします。JSP ページは、Java サーブレット クラスにコンパイルされ、WAR ファイルに保存されます。この操作により、JSP ページを最初のアクセス時より高速に処理できます。

メモ iastool を使用して JSP をコンパイルすると、メモリ不足エラーが発生することがあります。この問題を解決するには、システムの仮想メモリのサイズを大きくします。

構文

```
-compilejsp -src <war_or_ear> -target <target_file> [-overwrite]
[-package <package_root>] [-excludefile <exclude_file>] [-loglevel <0-4>]
[-classpath <classpath>]
```

デフォルト出力

デフォルトでは、`compilejsp` は操作が成功したかどうかを報告します。

オプション

`compilejsp` ツールで使用できるオプションを次の表に示します。

オプション	説明
`-src <war_or_ear>`	コンパイルする WAR または EAR ファイルを指定します。ファイルがある場所のフル パスか相対パスが必要です。デフォルト値はありません。
`-target <target_file>`	生成されるターゲットの WAR/EAR アーカイブ ファイルの名前を指定します。 指定したファイル名がすでに存在する場合は、`-overwrite` オプションを使って既存のターゲットを上書きします。 ファイルがある場所のフル パスか相対パスが必要です。 デフォルト値はありません。
`-overwrite`	<target_file> がすでに存在する場合は上書きすることを指定します。 <target_file> が存在し、 `-overwrite` が使用されていない場合、 ターゲットの JAR がソースの JAR と異なる必要があるというエラー メッセージが表示されます。
`-package <package_root>`	プリコンパイルした JSP サーブレット クラスの基本パッケージ名を指定します。 デフォルトは `com.bes.compiledjsp` です。
`-excludefile <exclude_file>`	コンパイル操作から除外する JSP ファイルのリストを含むテキスト ファイルを指定します。 詳細については、「excludefile オプションの使い方」を参照してください。
`-loglevel <0-4>`	生成される出力診断メッセージの量を指定します。2 より大きい値を指定すると、 検査用の一時サーブレット Java ファイルが残されます。 デフォルトは 2 です。
`-classpath <classpath>`	JSP ページをコンパイルするために必要な追加ライブラリを指定します。 デフォルト値はありません。

例

現在のディレクトリにある `proj1.war` という WAR ファイルに含まれる JSP ページを同じ場所の `proj1compiled.war` という WAR ファイルにプリコンパイルするには、次のコマンドを実行します。

```
iastool -compilejsp -src proj1.war -target proj1compiled.war
```

`c:\myprojects\` ディレクトリ内にある `proj1.ear` という EAR ファイルに含まれる JSP ページを同じ場所の `proj1compiled.ear` という EAR ファイルにプリコンパイルし、最大の量の診断メッセージを生成するには、次のコマンドを実行します。

```
iastool -compilejsp -src c:\myprojects\proj1.ear -target
c:\myprojects\proj1compiled.ear -loglevel 4
```

excludefile オプションの使い方

`compilejsp excludefile` オプションによって、コンパイル操作から除外する JSP のリストがあるテキスト ファイルを指定できます。以下に、使い方の規則の詳細を示します。

- '#' で始まるコメント行と空白行は無視されます。
- 各行の先頭および末尾の空白スペースは削除されます。
- 除外ファイルの各行は、それぞれの除外パターン エントリを表します。エントリは完全一致の文字列または Java パターンの正規表現です。
- 各 JSP 除外エントリは、まず JSP URL との完全一致のために使用されます。一致候補がない場合、JSP 除外エントリは JSP URL と正規表現で照合するために使用されます。

- JSP URL は、上記のアルゴリズムを使って各 JSP 除外エントリと比較されます。一致した JSP はコンパイルから除外されます。JSP URL がどの JSP 除外エントリとも一致しなければ、JSP はコンパイルされます。
- パターン エントリが有効な Java 正規表現ではない場合は警告が表示されます。その場合にも、JSP URL との完全一致の比較は実行されます。
- `iastool` の `-compilejsp -loglevel` オプションが 3 以上に設定されている場合は、除外パターン エントリ、除外される JSP ページ数、および除外される JSP URL が表示されます。
- アーカイブの JSP ファイルがすべて除外されると、`-compilejsp` コマンドは失敗します。

次に、除外パターンのサンプルを示します。

```
# このパターンは /jsp/test/test.jsp という JSP を除外します。
/jsp/test/test[.]jsp

# このパターンは JSP の /jsp/test/test.jsp、/jsp/test/test2jsp などを除外します。
# 正規表現 "." が任意の 1 文字を表すためです。
/jsp/test/test.jsp

# このパターンは /include URL パスにあるすべてのファイルを除外します。
/include/.*

# このパターンはすべての include.jsp ファイルを除外します。
.*/include[.]jsp

# このパターンは "tmp_" で始まる /jsp URL パスのすべての JSP ファイルと、
# /jsp 以下の "tmp_" で始まる URL パスのすべての JSP ファイルを除外します。
/jsp/tmp_.*[.]jsp

# このパターンは "tmp_" で始まる /jsp 直下（子孫のディレクトリを除く）の URL
パスのすべての JSP ファイルを除外します。
/jsp/tmp_[^/]*[.]jsp
```

compress

このツールを使用して、JAR ファイルを圧縮します。

構文

```
-compress -src <srcjar> -target <targetjar>
```

デフォルト出力

デフォルトでは、`compress` は操作が成功したかどうかを報告します。

オプション

`compress` ツールで使用できるオプションを次の表に示します。

オプション	説明
`-src <srcjar>`	圧縮する JAR ファイルを指定します。ファイルがある場所のフル パスか相対パスが必要です。 デフォルト値はありません。
`-target <targetjar>`	生成される圧縮 JAR ファイルの名前を指定します。ファイルがある場所のフル パスか相対パスが必要です。 デフォルト値はありません。

例

現在のディレクトリにある `proj1.jar` という JAR ファイルを同じディレクトリの `proj1compress.jar` ファイルに圧縮するには、次のコマンドを実行します。

```
iastool -compress -src proj1.jar -target proj1compress.jar
```

`c:\myprojects\` ディレクトリにある `proj1.jar` という JAR ファイルを同じディレクトリの `proj1compress.jar` ファイルに圧縮するには、次のコマンドを実行します。

```
iastool -compress -src c:\myprojects\proj1.jar
-target c:\myprojects\proj1compress.jar
```

deploy

このツールを使用して、指定したハブと設定内の指定したパーティションに J2EE モジュールをデプロイします。

構文

```
-deploy -jars <jar1,jar2,...> <-hub <hub> | -host <host>:listener_port>>
-cfg <configname> -partition <partitionname> [-force_restart] [-cp <classpath>]
[-args <args>] [-javac_args <args>] [-noverify] [-nostubs] [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

デフォルト出力

デフォルトでは、`deploy` は操作が成功したかどうかを報告します。

オプション

`deploy` ツールで使用できるオプションを次の表に示します。

オプション	説明
`-jars <jar1,jar2...>`	デプロイする 1 つまたは複数の JAR ファイルの名前を指定します。複数の JAR ファイルを指定する場合は、ファイル間をカンマ（,）で区切ります（スペースなし）。ファイルがある場所のフル パスか相対パスが必要です。デフォルト値はありません。
`-hub <hub>`	JAR ファイルをデプロイするハブの名前を指定します。
`-host<host>: <listener_port>`	目的のパーティションが稼動しているマシンのホスト名およびリスナー ポートを指定します。このオプションにより、`iastool` ユーティリティは、`iastool` が稼動しているマシンとは異なるサブネット上のパーティションを検索できます。
`-cfg <configname>`	JAR ファイルをロードするパーティションを含む設定の名前を指定します。
`partition <partitionname>`	JAR ファイルをロードするパーティションの名前を指定します。
`-force_restart`	モジュールをデプロイしたら、指定したパーティションを再起動します。このオプションを指定しない場合、パーティションを手動で再起動しないとモジュールを初期化できません。
`-cp <classpath>`	デプロイする JAR ファイルの依存関係を含むクラスパスを指定します。
`-args <args>`	JAR ファイルで必要な引数を指定します。詳細については、**『Borland VisiBroker for Java 開発者ガイド』**の「Java 対応プログラマ ツール」を参照してください。
`-javac_args <args>`	JAR ファイルで必要な Java コンパイラ引数を指定します。
`-noverify`	指定した管理ポートのパーティションへのアクティブな接続の検証を無効にします。
`-nostubs`	デプロイメント モジュールに対して、クライアント側またはサーバー側スタブ ファイルを作成しません。
`-mgmtport <nnnnn>`	指定したハブで使用する管理ポート番号を指定します。デフォルトは 42424 です。
`-realm <realm>`	ユーザーとパスワードのオプションを指定したら、ユーザー認証に使用する領域を指定します。
`-user <username>`	指定した領域に対して認証するユーザーを指定します。
`-pwd <password>`	指定した領域に対して認証するユーザーのパスワードを指定します。
`-file <login_file>`	ユーザーを認証するために使用する領域、ユーザー名、およびパスワードを含むログイン スクリプト ファイルを指定します。このファイルがある場所のフル パスか相対パスが必要です。詳細については、「スクリプト ファイルから iastool コマンドライン ツールを実行する」を参照してください。

dumpstack

このツールを使用して、パーティション内で実行されているスレッドに関する診断情報を取得します。このツールにより、パーティションですべてのスレッドのスタック トレースが生成され、その出力は次の場所にあるパーティションの `stdout.log` に保存されます。

```
<install_dir>/var/domains/<domain-name>/configurations/<configuration-name>/
    mos/<partition-name>/adm/logs/<partition_name>.stdout.log
```

スタック トレースは、パーティションの問題を診断する場合に役立つ可能性があります。ログ ファイルは、次のディレクトリにあります。

```
<install_dir>\var\domains\<domain_name>\configurations\<config_name>\
    <partition_name>\adm\logs\partition_log.xml
```

構文

```
-dumpstack <-hub <hub> | -host <host>:<listener_port>> -cfg <configname>
-partition <partitionname> [-mgmtport <nnnnn>] [-realm <realm>]
[-user <username>] [-pwd <password>] [-file <login_file>]
```

オプション

`dumpstack` ツールで使用できるオプションを次の表に示します。

オプション	説明
-hub <hub>|-host <hostname>:<listener_port>	目的のパーティション プロセスが稼動しているハブ名またはホスト名とマシンのリスナー ポートを指定します。 ハブ名またはホスト名のいずれかとリスナー ポートを指定する必要があります。 リスナー ポートを指定すると、`iastool` ユーティリティは、`iastool` が実行されているマシンとは異なるサブネット上のハブを検索できます。
-cfg <configname>	指定したパーティションを含む設定名を指定します。
-partition <partitionname>	診断するパーティション名を指定します。 有効なパーティション名を指定する必要があります。
-mgmtport <nnnnn>	指定したハブで使用する管理ポート番号を指定します。デフォルトは 42424 です。
-realm <realm>	ユーザーとパスワードのオプションを指定したら、 ユーザー認証に使用する領域を指定します。
-user <username>	指定した領域に対して認証するユーザーを指定します。
-pwd <password>	指定した領域に対して認証するユーザーのパスワードを指定します。
-file <login_file>	ユーザーを認証するために使用する領域、 ユーザー名、 およびパスワードを含むログイン スクリプト ファイルを指定します。このファイルがある場所のフル パスか相対パスが必要です。詳細については、「スクリプト ファイルから iastool コマンドライン ツールを実行する」を参照してください。

例

次のサンプルは、`BES1` ハブ上の `j2ee` 設定にある `standard` パーティションのスレッド ダンプを実行します。

```
iastool -dumpstack -hub BES1 -cfg j2ee -partition standard
```

次のサンプルは、特定のリスナー ポート上のコンピュータ ホストにある `standard` パーティションのスレッド ダンプを実行します。`-host` オプションは、`iastool` が実行されるマシンとパーティションが稼動しているマシンが同じホスト マシンであるかどうかに関係なく使用できます。

```
iastool -dumpstack -host mymachine:1234 -cfg j2ee -partition standard
```

genclient

このツールを使用して、1 つまたは複数の EJB JAR ファイルのクライアント スタブ ファイル、EJB インターフェイス、および依存クラス ファイルを含むライブラリを生成し、それらを 1 つまたは複数のクライアント JAR ファイルにパッケージします。クライアント JAR は EJB ではなく、EJB クライアントです。

genclient が引数リストにある EJB JAR のいずれかを正しく処理できなかった場合にエラーが表示されますが、genclient ツールは指定リストの残りの EJB JAR について引き続き処理を行い、クライアント JAR を生成します。

genclient ツールが 100% 成功したときは 0、1 つでも失敗があれば 1 を生成します。

構文

```
-genclient -jars <jar1,jar2,...> -target <client_jar> [-cp <classpath>]
[-args <java2iiop_args>] [-javac_args <args>]
```

デフォルト出力

デフォルト出力では標準出力（stdout）に何も返されません。

オプション

genclient ツールで使用できるオプションを次の表に示します。

オプション	説明
-jars <jar1,jar2,...>	1 つまたは複数のクライアント JAR ファイルを生成する対象になる 1 つまたは複数の JAR ファイルを指定します。複数の JAR ファイルを指定する場合は、ファイル間をカンマ（,）で区切ります（スペースなし）。JAR ファイルがある場所のフル パスか相対パスが必要です。デフォルト値はありません。
-target <client_jar>	ローカルホスト上に生成するクライアント JAR ファイルを指定します。JAR ファイルがある場所のフル パスか相対パスが必要です。デフォルト値はありません。
-cp <classpath>	クライアント JAR ファイルを生成する JAR ファイルのクラス依存関係を収めるクラスパスを指定します。 デフォルトは none です。
-args <java2iiop_args>	ファイルで必要な引数を指定します。 詳細については、**『Borland VisiBroker for Java 開発者ガイド』**の「Java 対応プログラマ ツール」を参照してください。
-javac_args <args>	JAR ファイルで必要な Java コンパイラ引数を指定します。

例

次のサンプルは、各 EJB JAR ファイルからデプロイメント可能なモジュールクライアント JAR ファイルを手動で生成します（proj1.jar、proj2.jar、および proj3.jar から EJB JAR myproj.jar）。

```
iastool -genclient -jars proj1.jar,proj2.jar,proj3.jar -target myproj.jar
```

gendeployable

このツールを使用して、手動でデプロイメント可能なサーバー側モジュールを生成します。デプロイメント可能なサーバー側 JAR ファイルとは、スタブですべての外部コード リファレンスを解決できるようにコンパイルされているデプロイメント可能なアーカイブ（EAR、WAR、または JAR Bean のみ）です。

たとえば、まず gendeployable を使ってデプロイメント可能なサーバー側 JAR ファイルをローカル マシンに作成し、次に deploy ツールを使用して、そのファイルをハブ上にコピーしてロードします。新しい JAR ファイルの存在がハブに伝えられ、自動的にロードされます。このコマンドライン ツールを利用することで、複数のサーバーでも作成とデプロイメントのスクリプトを簡単に、また短時間に作成できます。デプロイメント可能なサーバー側 JAR ファイルは、手動で各ハブの正しい場所にコピーすることもできますが、それを認識させてロードするにはハブごとに再起動する必要があります。

構文

```
-gendeployable -src <input_jar> -target <output_jar> [-cp <classpath>]
[-args <java2iiop_args>] [-javac_args <args>]
```

デフォルト出力

デフォルト出力では標準出力（stdout）に何も返されません。

オプション

gendeployable ツールで使用できるオプションを次の表に示します。

オプション	説明
-src <input_jar>	デプロイメント可能な JAR ファイルを新しく生成する JAR ファイル（または拡張 JAR のディレクトリ）を指定します。 JAR ファイル場所のフルパスか相対パスが必要です。 デフォルト値はありません。
-target <output_jar>	ローカルホスト上に生成するデプロイメント可能な JAR ファイルを指定します。 JAR ファイルがある場所のフル パスか相対パスが必要です。 デフォルト値はありません。
-cp <classpath>	クライアント JAR ファイルを生成する JAR ファイルのクラス依存関係を収めるクラスパスを指定します。 デフォルトは none です。
-args <java2iiop_args>	ファイルで必要な引数を指定します。 詳細については、**『Borland VisiBroker for Java 開発者ガイド』**の「Java 対応プログラマ ツール」を参照してください。
-javac_args <args>	JAR ファイルで必要な Java コンパイラ引数を指定します。

例

次のサンプルは、proj1.jar のサーバー側のデプロイメント可能なモジュール JAR ファイルを serverside.jar ファイル内に生成します。

```
iastool -gendeployable -src proj1.jar -target serverside.jar
```

gendeployableverify

このコマンドライン プロパティは、gendeployable と verify ユーティリティを結合します。この結合オプションは、全体のスタブ生成と検証プロセスを促進します。手動でデプロイメント可能なサーバー側モジュールが生成され、検証されます。

構文

```
-gendeployableverify -src <input_jar>  -target <output_jar> [-args
<java2iiop_args>] [-javac_args <args>] [-role <DEVELOPER|ASSEMBLER|DEPLOYER>]
[client] [-nowarn] [-strict] [-cp <classpath>]
```

デフォルト出力

デフォルト出力では標準出力（stdout）に何も返されません。

オプション

gendeployableverify ツールで使用できるオプションを次の表に示します。

オプション	説明
-src <input_jar>	デプロイメント可能な JAR ファイルを新しく生成する JAR ファイル（または拡張 JAR のディレクトリ）を指定します。 JAR ファイル場所のフルパスか相対パスが必要です。 デフォルト値はありません。
-target <output_jar>	ローカルホスト上に生成するデプロイメント可能な JAR ファイルを指定します。 JAR ファイルがある場所のフル パスか相対パスが必要です。 デフォルト値はありません。
-cp <classpath>	クライアント JAR ファイルを生成する JAR ファイルのクラス依存関係を収めるクラスパスを指定します。 デフォルトは none です。

オプション	説明
-args <java2iiop_args>	ファイルで必要な引数を指定します。 詳細については、**『Borland VisiBroker for Java 開発者ガイド』**の「Java 対応プログラマ ツール」を参照してください。
-javac_args <args>	JAR ファイルで必要な Java コンパイラ引数を指定します。
-role <DEVELOPER|ASSEMBLER|DEPLOYER>	エラー チェックのレベルを指定します。
-nowarn	ツールに対して、 デプロイメントを不可能にしているエラーだけを報告し、 警告は報告しないように指定します。
-strict	ツールに対して、最も詳しいレベルで矛盾点を報告するように指定します。矛盾点の多くは、アプリケーション全体の整合性に影響しません。
-classpath <classpath>	アプリケーション クラスとリソースの検索パスを指定します。 1 つまたは複数のディレクトリ、ZIP、または JAR ファイル エントリを入力する場合は、 各エントリをセミコロン（;）で区切ります。

例
次のサンプルは、`proj1.jar` のサーバー側のデプロイメント可能なモジュール JAR ファイルを `serverside.jar` ファイル内に生成し、検証します。

```
iastool -gendeployableverify  -src proj1.jar -target serverside.jar
```

genstubs

このツールを使用して、クライアント スタブやサーバー スタブを含むスタブ ライブラリ ファイルを作成します。

構文

```
-genstubs -src <input_jar> -target <output_jar> [-client] [-cp <classpath>]
[-args <java2iiop_args>] [-javac_args <args>]
```

デフォルト出力
デフォルト出力では標準出力（stdout）に何も返されません。

オプション
`genstubs` ツールで使用できるオプションを次の表に示します。

オプション	説明
-src <input_jar>	スタブ ライブラリを生成する JAR ファイル（または拡張 JAR のディレクトリ）を指定します。 JAR ファイル場所のフル パスか相対パスが必要です。 デフォルト値はありません。
-target <output_jar>	ローカルホストに生成される JAR ファイル名を指定します。 JAR ファイル場所のフル パスか相対パスが必要です。 デフォルト値はありません。
-client	クライアント側のスタブの生成を指定します。このオプションが指定されない場合、 `genstubs` ツールはサーバー側のスタブを生成します。
-cp <classpath>	クライアント JAR ファイルを生成する JAR ファイルのクラス依存関係を収めるクラスパスを指定します。 デフォルトは none です。
-args <java2iiop_args>	ファイルで必要な引数を指定します。 詳細については、**『Borland VisiBroker for Java 開発者ガイド』**の「Java 対応プログラマ ツール」を参照してください。
-javac_args <args>	JAR ファイルで必要な Java コンパイラ引数を指定します。

例
次のサンプルは、EJB JAR `proj1.jar` のサーバー側スタブを EJB JAR `serverside.jar` に生成します。

```
iastool -genstubs -src proj1.jar -target serverside.jar
```

次のサンプルは、EJB JAR `myproj.jar` のクライアント側スタブを EJB JAR `client-side.jar` に生成します。

```
iastool -genstubs -src c:\dev\proj1.jar -target
-client c:\builds\client-side.jar
```

info

このツールを使用して、`iastool` が稼動している JVM の Java システムプロパティを表示します。

構文

```
-info
```

デフォルト出力
デフォルト出力は、`iastool` が稼動している JVM の現在の Java システム プロパティです。たとえば、出力された最初の数行は次のリスト（部分）のようになります。

```
application.home              : C:\Program Files\AppServer
awt.toolkit                   : sun.awt.windows.WToolkit
file.encoding                 : Cp1252
file.encoding.pkg             : sun.io
file.separator                : \
java.awt.fonts                :
java.awt.graphicsenv          : sun.awt.Win32GraphicsEnvironment
java.awt.printerjob           : sun.awt.windows.WPrinterJob
java.class.path               : C:\Program Files\AppServer\jdk\lib\tools.jar

.
.
.
```

例
次のサンプルは、設定情報を表示します。

```
iastool -info|more
```

jndinamespace

このツールを使用して、コンテキストにバインドされたオブジェクトを表示します。このユーティリティは、コマンドラインの JNDI ブラウザと同じです。対話型ユーティリティで、ユーザーの入力によりバインドされたオブジェクトが表示されます。

構文

```
-jndinamespace -osagent <osagent port number>
```

デフォルト出力
指定コンテキストにバインドされたオブジェクトを表示します。

オプション
```
Osagent
```

このオプションは、osagent ポート番号を指定します。必須です。

例
次のサンプルは、コンテキストにバインドされたオブジェクトの表示方法を示します。

```
iastool -jndinamespace -osagent 19999
```

次の出力が表示されます。

```
Retrieving the jndi name space details
The list of nodes under  19999
[0]order
[1]serial_provider_defaultname
[2]datasources
Enter the context [0,1,...] or X to exit
```

2 を入力します。

次の出力が表示されます。

```
The list of nodes under  19999/datasources
[0]Jdatastore
[1]Oracle
Enter the context [0,1,...] or X to exit
```

X を入力して終了します。

kill

このツールを使用して、指定したハブおよび設定上の管理オブジェクトを強制終了します。

構文

```
-kill <-hub <hub> | -host <host>:listener_port>> -cfg <configname>
-mo <managedobjectname> -moagent <managedobjectagent> [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

デフォルト出力

デフォルトでは、`kill` ツールは、強制終了された管理オブジェクトを一覧表示します。

オプション

`kill` ツールで使用できるオプションを次の表に示します。

オプション	説明
-hub <hub>	強制終了する管理オブジェクトが存在するハブの名前を指定します。
-host <host>:<listener_port>	目的の管理オブジェクトが稼動しているマシンのホスト名およびリスナー ポートを指定します。このオプションにより、`iastool` ユーティリティは、`iastool` が実行されているマシンとは異なるサブネット上のハブを検索できます。
-cfg <configname>	指定した管理オブジェクトを含む設定名を指定します。
-mo <managedobjectname>	管理オブジェクトの名前を指定します。
-moagent <managedobjectagent>	管理オブジェクトのエージェント名を指定します。 指定したハブに複数のエージェントがある場合は、 このオプションを使用します。
-mgmtport <nnnnn>	指定したハブで使用する管理ポート番号を指定します。 デフォルトは 42424 です。
-realm <realm>	ユーザーとパスワードのオプションを指定したら、ユーザー認証に使用する領域を指定します。
-user <username>	指定した領域に対して認証するユーザーを指定します。

オプション	説明
-pwd <password>	指定した領域に対して認証するユーザーのパスワードを指定します。
-file <login_file>	ユーザーを認証するために使用する領域、ユーザー名、およびパスワードを含むログイン スクリプト ファイルを指定します。このファイルがある場所のフル パスか相対パスが必要です。詳細については、「スクリプト ファイルから iastool コマンドライン ツールを実行する」を参照してください。

例

次のサンプルは、デフォルトの管理ポートを使用する管理オブジェクト `j2ee-server` を強制終了します。

```
iastool -kill -hub AppServer1 -cfg j2ee -mo j2ee-server
```

listpartitions

このツールを使用して、特定のハブ上で稼動しているパーティションを一覧表示します。また、オプションで、特定の設定または管理ポート上で稼動しているハブを一覧表示します。

構文

```
-listpartitions <-hub <hub> | -host <host>:<listener_port>>
[-cfg <configname>] [-mgmtport <nnnnn>] [-bare] [-realm <realm>]
[-user <username>] [-pwd <password>] [-file <login_file>]
```

デフォルト出力

デフォルトでは、`listpartitions` ツールは、指定したハブ上で稼動しているパーティション、または、指定した設定または管理ポート上の指定したハブ上で実行中のパーティションを表示します。

オプション

`listpartitions` ツールで使用できるオプションを次の表に示します。

オプション	説明
-hub <hub>	実行中のパーティションを一覧表示するハブの名前を指定します。
-host <host>:<listener_port>	目的のパーティションが稼動しているマシンのホスト名およびリスナー ポートを指定します。このオプションにより、`iastool` ユーティリティは、`iastool` が実行されているマシンとは異なるサブネット上のハブを検索できます。
-cfg <configname>	パーティションを一覧表示する設定の名前を指定します。
-mgmtport <nnnnn>	指定したハブで使用する管理ポート番号を指定します。デフォルトは 42424 です。
-bare	実行中のパーティション名以外の出力情報をオフにします。
-realm <realm>	ユーザーとパスワードのオプションを指定したら、ユーザー認証に使用する領域を指定します。
-user <username>	指定した領域に対して認証するユーザーを指定します。
-pwd <password>	指定した領域に対して認証するユーザーのパスワードを指定します。
-file <login_file>	ユーザーを認証するために使用する領域、ユーザー名、およびパスワードを含むログイン スクリプト ファイルを指定します。このファイルがある場所のフル パスか相対パスが必要です。詳細については、「スクリプト ファイルから iastool コマンドライン ツールを実行する」を参照してください。

例
次のサンプルは、デフォルトの管理ポートを使用するハブ AppServer1 上で実行中のパーティションを一覧表示します。

```
iastool -listpartitions -hub AppServer1
```

次のサンプルは、デフォルトの管理ポートを使用するハブ AppServer1 上で実行中のパーティションを一覧表示します。

```
iastool -listpartitions -hub AppServer1 -mgmtport 24100
```

listhubs

このツールを使用して、同じ LAN 上にある特定の管理ポートで実行中のハブを一覧表示します。

構文

```
-listhubs [-mgmtport <nnnnn>] [-bare] [-realm <realm>] [-user <username>]
[-pwd <password>] [-file <login_file>]
```

デフォルト出力
デフォルトでは、listhubs ツールは、デフォルトの管理ポート、または指定した管理ポート上で実行中のハブを表示します。

メモ その時点で稼動していないハブは表示されません。

オプション
listhubs ツールで使用できるオプションを次の表に示します。

オプション	説明
-mgmtport <nnnnn>	一覧表示する実行中のハブの管理ポート番号を指定します。デフォルトは 42424 です。
-bare	実行中のハブ名以外の出力情報をオフにします。
-realm <realm>	ユーザーとパスワードのオプションを指定したら、ユーザー認証に使用する領域を指定します。
-user <username>	指定した領域に対して認証するユーザーを指定します。
-pwd <password>	指定した領域に対して認証するユーザーのパスワードを指定します。
-file <login_file>	ユーザーを認証するために使用する領域、ユーザー名、およびパスワードを含むログイン スクリプト ファイルを指定します。このファイルがある場所のフル パスか相対パスが必要です。詳細については、「スクリプト ファイルから iastool コマンドライン ツールを実行する」を参照してください。

例
次のサンプルは、デフォルトの管理ポートで実行中のハブを一覧表示します。

```
iastool -listhubs
```

次のサンプルは、管理ポート 24410 で実行中のハブを一覧表示します。

```
iastool -listhubs -mgmtport 24100
```

listservices

このツールを使用して、ハブ上で実行中の 1 つまたは複数のサービスを一覧表示します。

構文

```
-listservices  <-hub <hub> | -host <host>:<listener_port>> [-cfg <configname>]
[-mgmtport <nnnnn>] [-bare] [-realm <realm>] [-user <username>]
[-pwd <password>] [-file <login_file>]
```

デフォルト出力
デフォルトでは、`listservices` は、特定の管理ポートの指定されたハブに対して登録されているすべてのパーティション サービスを一覧表示します。

オプション
`listservices` ツールで使用できるオプションを次の表に示します。

オプション	説明
-hub <hub>	実行中のサービスを一覧表示するハブの名前を指定します。
-host <host>:<listener_port>	目的のサービスが稼動しているマシンのホスト名およびリスナー ポートを指定します。 このオプションにより、 `iastool` ユーティリティは、 `iastool` が実行されているマシンとは異なるサブネット上のハブを検索できます。
-cfg <configname>	サービスを一覧表示する設定の名前を指定します。
-mgmtport <nnnnn>	指定したハブで使用する管理ポート番号を指定します。デフォルトは 42424 です。
-bare	実行中のサービス以外の情報の出力をオフにします。
-realm <realm>	ユーザーとパスワードのオプションを指定したら、 ユーザー認証に使用する領域を指定します。
-user <username>	指定した領域に対して認証するユーザーを指定します。
-pwd <password>	指定した領域に対して認証するユーザーのパスワードを指定します。
-file <login_file>	ユーザーを認証するために使用する領域、ユーザー名、およびパスワードを含むログイン スクリプト ファイルを指定します。 このファイルがある場所のフル パスか相対パスが必要です。 詳細については、「スクリプト ファイルから iastool コマンドライン ツールを実行する」を参照してください。

例
次のサンプルは、`salsa` ハブで実行中のすべてのサービスを一覧表示します。

```
iastool -listservices -hub salsa
```

manage
このツールを使用して、設定内の管理オブジェクトをアクティブに管理します。

構文
```
-manage  (-hub <hub>|-host <host>:<listener_port>) [-cfg <configname>] -mo
<managedobjectname> [-moagent <managedobjectagent>] [-mgmtport <99999>] [-realm
<realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

デフォルト出力
デフォルト出力では標準出力（stdout）に何も返されません。

オプション
`manage` ツールで使用できるオプションを次の表に示します。

オプション	説明
-hub <hub>	管理オブジェクトが稼動しているハブの名前を指定します。
-host <host>:<listener_port>	管理オブジェクトが稼動しているマシンのホスト名およびリスナー ポートを指定します。 このオプションにより、 iastool ユーティリティは、 iastool が稼動しているマシンとは異なるサブネット上のハブを検索できます。

オプション	説明
-cfg <configname>	管理オブジェクトが関連付けられている設定の名前を指定します。
-mo <managedobjectname>	管理オブジェクトの名前を指定します。
-moagent <managedobjectagent>	管理オブジェクトが関連付けられているエージェントを指定します。
-mgmtport <99999>	管理オブジェクトのエージェントが関連付けられているポートを指定します。
-realm <realm>	ユーザーとパスワードのオプションを指定したら、ユーザー認証に使用する領域を指定します。
-user <username>	指定した領域に対して認証するユーザーを指定します。
-pwd <password>	指定した領域に対して認証するユーザーのパスワードを指定します。
-file <login_file>	ユーザーを認証するために使用する領域、ユーザー名、およびパスワードを含むログイン スクリプト ファイルを指定します。このファイルがある場所のフル パスか相対パスが必要です。詳細については、「スクリプト ファイルから iastool コマンドライン ツールを実行する」を参照してください。

例

次のサンプルは、デフォルトの管理ポートを使って管理オブジェクト j2ee-server をアクティブな管理モードにします。

```
iastool -manage -hub AppServer1 -cfg j2ee -mo j2ee-server
```

merge

このツールを使用して、指定した EJB-JAR のリストの内容を保持する単一の Java アーカイブ ファイル（EJB-JAR）を生成します。それらの JAR ファイルが複数の EJB 1.1 と EJB 2.0 デプロイメント デスクリプタを保持している場合は、それらのデプロイメント デスクリプタが統合されて 1 つのデプロイメント デスクリプタになります。引数リスト内にある EJB-JAR の 1 つに対するマージが失敗すると、エラーが表示され、merge コマンドが失敗の表示を終了します。

構文

```
-merge -jars <jar1,jar2,...> -target <new_jar> -type <valid_type>
```

デフォルト出力

デフォルト出力では標準出力（stdout）に何も返されません。

オプション

merge ツールで使用できるオプションを次の表に示します。

オプション	説明
-jars <jar1,jar2,..>	マージする JAR ファイルをカンマで区切って指定します（スペースなし）。JAR ファイルがある場所のフル パスか相対パスが必要です。 デフォルト値はありません。

オプション	説明
-target <new_jar>	作成する新しい JAR ファイルの名前を指定します。この JAR ファイルは、指定した JAR ファイルのリストのマージされた内容を保持します。新しい JAR ファイル場所のフル パスか相対パスが必要です。デフォルト値はありません。
-type *valid_type*	サポートされた次の形式の 1 つを使用して、新しいアーカイブ ファイルの種類を指定します。 ■ `ejb2.0` - バージョン 2.0 Enterprise Java Bean ■ `ejb1.1` - バージョン 1.1 Enterprise Java Bean ■ `ear1.3` - バージョン 1.3 Enterprise Application Resource ■ `ear1.2` - バージョン 1.2 Enterprise Application Resource ■ `lib` - Library ファイル ■ `war2.3` - バージョン 2.3 Web アプリケーション アーカイブ ■ `war2.2` - バージョン 2.2 Web アプリケーション アーカイブ ■ `rar1.0` - バージョン 1.0 リソース アダプタ アーカイブ ■ `client1.2` - バージョン 1.2 クライアント JAR ■ `client1.3` - バージョン 1.3 クライアント JAR ■ `jndi1.2` - バージョン 1.2 Java ネーミングとディレクトリ インターフェイス

例

次のサンプルは、EJB-JAR ファイル `proj1.jar`、`proj2.jar`、および `proj3.jar` を新しいバージョン 2.0 EJB-JAR ファイル `combined.jar` にマージします。

```
iastool -merge -jars proj1.jar,proj2.jar,proj2.jar
-target combined.jar -type ejb2.0
```

migrate

このツールを使用して、J2EE バージョン 1.2 から J2EE バージョン 1.3 または J2EE 1.4 J2EE など、あるバージョンから別のバージョンに JAR ファイルまたは XML ファイルを変換します。

メモ `migrate` コマンドは、EJB のデプロイメント デスクリプタのみを変換します。したがって、コードを変更するには、デプロイメントで適切に変換を実装する必要があります。

変換が失敗すると、エラーが表示されます。

構文

```
-migrate [-to[1.2|1.3|1.4]] -src <src-archive> -target <target-archive>
```

デフォルト出力

デフォルトでは標準出力（stdout）に何も返されません。

オプション

`migrate` ツールで使用できるオプションを次の表に示します。

オプション	説明
-to[1.2|1.3|1.4]	ソースの J2EE モジュールを移行するターゲットのバージョンを指定します。 たとえば、 J2EE 1.3 ソース モジュールは、 J2EE 1.2 または J2EE 1.4 に移行でき、 J2EE 1.2 モジュールはターゲットの J2EE 1.3 または J2EE 1.4 モジュールに移行できます。このオプションを使用しないと、 デフォルトの J2EE 1.4 になります。
-src <src-archive>	変換する J2EE モジュールを指定します。アーカイブ ファイルのフル パスまたは相対パス（または拡張 JAR のディレクトリ）を指定する必要があります。 デフォルト値はありません。
-target <target-archive>	作成される J2EE モジュールの名前を指定します。 アーカイブファイル場所のフル パスか相対パスが必要です。 デフォルト値はありません。

例

次のサンプルは、J2EE バージョン 1.2 から J2EE バージョン 1.3 へ、`myj1_2.jar` から `myj1_3.jar` と呼ばれる新しいファイルに移行します。

```
iastool -migrate -src myj1_2.jar -target myj1_4.jar //1.4 モジュールに変換されます。
iastool -migrate -to 1.3 -src myj1_2.jar -target myj1_3.jar // 1.2 モジュールが 1.3 に変換されます。
```

newconfig

このツールを使用して、設定テンプレートから設定を新規作成します。このコマンドは、新しい設定の名前、インストールの設定テンプレート ディレクトリに対するテンプレート ファイルの相対パスとファイル名、および設定の新規作成に使用するテンプレートのプロパティを上書きするためのプロパティ ファイル（オプション）を受け取ります。

構文

```
-newconfig (-hub <hub> | -host <host>:<listener_port>) -cfg <configname>
   -template <template_path> [-property <property_path>] [-mgmtport <99999>]
   [-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

デフォルト出力

デフォルト出力では標準出力（stdout）に何も返されません。

オプション

オプション	説明
`-hub <hub>`	設定を新規作成するハブの名前を指定します。
`-host <host>:<listener_port>`	設定を新規作成するために、ハブが稼動しているマシンのホスト名とリスナー ポートを指定します。 このオプションにより、 iastool ユーティリティは、 iastool が稼動しているマシンとは異なるサブネット上のハブを検索できます。
`-cfg <configname>`	新しい設定の名前を指定します。
`-template <template_path>`	設定テンプレートが保存されているパスを指定します。 `template_path` には、設定テンプレートの XML ファイルのフル パス、または設定テンプレート ディレクトリ（`<install_dir>/var/templates/configurations/`）に対する相対パスを指定します。
`-property <property_path>`	テンプレートで設定を新規作成するために使用するプロパティを上書きするプロパティ ファイル（オプション）のパスを指定します。
`-mgmtport <99999>`	指定したハブで使用する管理ポート番号を指定します。デフォルトは 42424 です。
`-realm <realm>`	ユーザーとパスワードのオプションを指定したら、 ユーザー認証に使用する領域を指定します。
`-user <username>`	指定した領域に対して認証するユーザーを指定します。
`-pwd <password>`	指定した領域に対して認証するユーザーのパスワードを指定します。
`-file <login_file>`	ユーザーを認証するために使用する領域、ユーザー名、およびパスワードを含むログイン スクリプト ファイルを指定します。 このファイルがある場所のフル パスか相対パスが必要です。 詳細については、「スクリプト ファイルから iastool コマンドライン ツールを実行する」を参照してください。

例

```
iastool -newconfig -hub myhub -cfg SimpleProcessConfig -template native.xml
   -property c:\simple.properties
```

patch

このツールを使用して、JAR ファイルに 1 つまたは複数のパッチを適用します。適用されたパッチを使って新しい JAR ファイルを生成します。

構文

```
-patch -src <original_jar> -patches <patch1_jar,...> -target <new_jar>
```

デフォルト出力

デフォルト出力には、適用されたパッチが表示されます。

オプション

patch ツールで使用できるオプションを次の表に示します。

オプション	説明
-src <original_jar>	1 つまたは複数のパッチを適用する JAR ファイルを指定します。JAR ファイル場所のフル パスか相対パスが必要です。デフォルト値はありません。
-patches <patch1_jar,...>	適用するパッチを含む 1 つまたは複数の JAR ファイルを指定します。 複数のファイルを指定する場合は、ファイル間をカンマ（,）で区切ります（スペースなし）。ファイルがある場所のフル パスか相対パスが必要です。 デフォルト値はありません。
-target <new_jar>	作成される新しい JAR ファイルの名前を指定します。 JAR ファイル場所のフル パスか相対パスが必要です。 デフォルト値はありません。

例

次のサンプルは、mypatch1.jar および mypatch2.jar ファイルに含まれるパッチを myold.jar ファイルに適用します。これらのファイルはすべて現在のディレクトリにあり、同じ場所に mynew.jar と呼ばれる新しいファイルを作成します。

```
iastool -patch -src myold.jar -patches mypatch1.jar,mypatch2.jar
-target mynew.jar
```

ping

このツールを使用して、ハブまたは管理オブジェクトの現在の状態を検証します。ping コマンドは、実行されていないハブに関しては何も返しません。

構文

```
-ping <-hub <hub> | -host <host>:<listener_port>> [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

または

```
-ping <-hub <hub> | -host <host>:<listener_port>> -cfg <configname>
-mo <managedobjectname> -moagent <managedobjectagent> [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

デフォルト出力

デフォルト出力には、プロセスが ping されたり実行中の場合は、ハブの名前と状態、オプションでサービスまたはパーティションが表示されます。たとえば、次のようになります。

```
Pinging Hub xyz_corp1: Running（実行中）
```

ping ツールは、次の状態の 1 つを返します。

- Running（実行中）
- Starting（起動中）
- Stopping（停止中）
- Not Running（実行中ではない）
- Restarting（再起動中）
- Cannot Load（ロードできない）
- Cannot Start（起動できない）
- Terminated（終了）
- Unknown（不明）

オプション

ping ツールで使用できるオプションを次の表に示します。

オプション	説明
-hub <hub>	ping するハブ、またはサービスを ping するサーバーを指定します。 デフォルト値はありません。
-host <host>:<listener_port>	目的のハブまたは管理オブジェクトが稼動しているマシンのホスト名およびリスナー ポートを指定します。 このオプションにより、iastool ユーティリティは、 iastool が実行されているマシンとは異なるサブネット上の管理オブジェクトを検索できます。
-cfg <configname>	管理オブジェクトに対して ping を実行する設定の名前を指定します。
-mo <managedobjectname>	管理オブジェクトの名前を指定します。
-moagent <managedobjectagent>	管理オブジェクトのエージェント名を指定します。 指定したハブに複数のエージェントがある場合は、 このオプションを使用します。
-mgmtport <nnnnn>	指定したハブで使用する管理ポート番号を指定します。 デフォルトは 42424 です。
-realm <realm>	ユーザーとパスワードのオプションを指定したら、ユーザー認証に使用する領域を指定します。
-user <username>	指定した領域に対して認証するユーザーを指定します。
-pwd <password>	指定した領域に対して認証するユーザーのパスワードを指定します。
-file <login_file>	ユーザーを認証するために使用する領域、 ユーザー名、 およびパスワードを含むログイン スクリプト ファイルを指定します。このファイルがある場所のフル パスか相対パスが必要です。詳細については、「スクリプト ファイルから iastool コマンドライン ツールを実行する」を参照してください。

例

次のサンプルは、デフォルトの管理ポートでハブ AppServer1 に対して ping を実行します。

```
iastool -ping -hub AppServer1
```

次のサンプルは、管理ポート 24410 の AppServer1 で実行中のパーティション ネーミング サービスに対して ping を実行します。

```
iastool -ping -hub AppServer1 -cfg j2ee -mo standard_visinaming -mgmtport 24410
```

pservice

このツールを使用して、パーティション サービスを有効、無効、またはパーティション サービスの状態を取得します。

構文

```
-pservice <hub <hub> | -host <host>:<listener_port>> -cfg <configname>
-partition <partitionname> -moagent <managedobjectagent>
-service <servicename> <-enable|-disable|-status> [-force_restart]
[-mgmtport <nnnnn>] [-realm <realm>] [-user <username>] [-pwd <password>]
[-file <login_file>]
```

デフォルト出力

デフォルト出力では標準出力（stdout）に何も返されません。

オプション

`pservice` ツールで使用できるオプションを次の表に示します。

オプション	説明
-hub <hub>	目的のパーティション サービスが存在するハブを指定します。デフォルト値はありません。
-host <host>:<listener_port>	目的のパーティション サービスが稼動しているマシンのホスト名およびリスナー ポートを指定します。 このオプションにより、`iastool` ユーティリティは、`iastool` が実行されているマシンとは異なるサブネット上のパーティション サービスを検索できます。
-partition <partitionname>	パーティションの名前を指定します。
-moagent <managedobjectagent>	管理オブジェクトのエージェント名を指定します。 指定したハブに複数のエージェントがある場合は、 このオプションを使用します。
-service <servicename>	サービスの名前を指定します。
-enable | -disable | -status	パーティション サービスに対して実行する操作を指定します。
-force_restart	有効化、無効化、または状態の取得操作を完了した後、指定したパーティションを再起動します。このオプションを指定しない場合、パーティションを手動で再起動しないとモジュールを初期化できません。
-mgmtport <nnnnn>	指定したハブで使用する管理ポート番号を指定します。 デフォルトは 42424 です。
-realm <realm>	ユーザーとパスワードのオプションを指定したら、ユーザー認証に使用する領域を指定します。
-user <username>	指定した領域に対して認証するユーザーを指定します。
-pwd <password>	指定した領域に対して認証するユーザーのパスワードを指定します。
-file <login_file>	ユーザーを認証するために使用する領域、 ユーザー名、 およびパスワードを含むログイン スクリプト ファイルを指定します。このファイルがある場所のフル パスか相対パスが必要です。詳細については、「スクリプト ファイルから iastool コマンドライン ツールを実行する」を参照してください。

例

次のサンプルは、標準のパーティションでパーティション ネーミング サービスを有効にします。

```
iastool -pservice -hub AppServer1 -cfg j2ee -partition standard
-service standard_visinaming -enable -force_restart -mgmtport 24431
```

removestubs

このツールを使用して、JAR ファイルからすべてのスタブ ファイルを削除します。

構文

```
-removestubs -jars <jar1,jar2,...> [-targetdir <dir>]
```

デフォルト出力

デフォルト出力では標準出力（stdout）に何も返されません。

オプション

`removestubs` ツールで使用できるオプションを次の表に示します。

オプション	説明
`-jars <jar1,jar2...>`	1 つまたは複数のスタブ ファイルを削除する JAR ファイルを指定します。複数の JAR ファイルを指定する場合は、JAR ファイル間をカンマ（,）で区切ります（スペースなし）。JAR ファイル場所のフル パスか相対パスが必要です。 デフォルト値はありません。
`-targetdir <dir>`	削除したスタブ ファイルを保存するディレクトリを指定します。 このオプションが指定されている場合は、 フル パスまたは相対パスを指定する必要があります。デフォルト値はありません。 保存先ディレクトリが指定されていないと、 スタブ ファイルは削除されても、 保存されません。

例

次のサンプルは、現在のディレクトリにある EJB JAR ファイル `proj1.jar`、`proj2.jar`、および `proj3.jar` からスタブ ファイルを削除し、`c:\examples\proto` にコピーします。

```
iastool -removestubs -jars proj1.jar,proj2.jar,proj3.jar
-targetdir c:\examples\proto
```

restart

このツールを使用して、ハブまたは管理オブジェクトを再起動します。`restart` ツールをハブに対して実行するには、そのハブがすでに実行されている必要があります。

構文

```
-restart <-hub <hub> | -host <host>:<listener_port>> [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

または

```
-restart <-hub <hub> | -host <host>:<listener_port>> [-cfg <configname>]
-mo <managedobjectname> -moagent <managedobjectagent> [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

デフォルト出力

デフォルト出力には、再起動されたハブまたは管理オブジェクトが表示されます。

管理オブジェクトがシャットダウンしたり再起動できない場合など、`restart` ツールが失敗すると、状態コードとともにエラーが表示されます。状態コードは、標準エラー出力（stderr）に返されます。

オプション

`restart` ツールで使用できるオプションを次の表に示します。

オプション	説明
-hub <hub>	再起動するハブの名前を指定します。 特定のハブ上の管理オブジェクトを検索するためにも使用できます。
-host <host>:<listener_port>	目的の管理オブジェクトが稼動しているマシンのホスト名およびリスナー ポートを指定します。このオプションにより、`iastool` ユーティリティは、 `iastool` が実行されているマシンとは異なるサブネット上の管理オブジェクトを検索できます。
-cfg <configname>	管理オブジェクトを検索する設定の名前を指定します。
-mo <managedobjectname>	管理オブジェクトの名前を指定します。
-moagent <managedobjectagent>	管理オブジェクトのエージェント名を指定します。 指定したハブに複数のエージェントがある場合は、 このオプションを使用します。
-mgmtport *nnnnn*	指定したハブで使用する管理ポート番号を指定します。 デフォルトは 42424 です。
-realm *realm*	ユーザーとパスワードのオプションを指定したら、ユーザー認証に使用する領域を指定します。
-user *username*	指定した領域に対して認証するユーザーを指定します。
-pwd *password*	指定した領域に対して認証するユーザーのパスワードを指定します。
-file <login_file>	ユーザーを認証するために使用する領域、 ユーザー名、 およびパスワードを含むログイン スクリプト ファイルを指定します。このファイルがある場所のフル パスか相対パスが必要です。詳細については、「スクリプト ファイルから iastool コマンドライン ツールを実行する」を参照してください。

例

次のサンプルは、デフォルトの管理ポートでハブ `AppServer1` を再起動します。

```
iastool -restart -hub AppServer1
```

次のサンプルは、管理ポート 24410 を使用するハブ `AppServer1` で稼動しているパーティション ネーミング サービスを再起動します。

```
iastool -restart -hub AppServer1 -cfg j2ee -mo standard_visinaming  -mgmtport
24410
```

setmain

このツールを使用して、スタンドアロンのクライアント JAR、または EAR ファイル内のクライアント JAR のメイン クラスを設定します。メイン クラスが設定されると、`java -jar jarfile` コマンドは、JAR ファイルに対して設定されているメイン クラスを自動的に呼び出します。

構文

```
-setmain -jar <jar_or_ear> [-uri <client_jar_in_ear>] -class <main_classname>
```

デフォルト出力

デフォルト出力には、指定した JAR ファイルに対して設定されているメイン クラスが表示されます。

オプション
setmain ツールで使用できるオプションを次の表に示します。

オプション	説明
-jar <jar_or_ear>	メイン クラスを設定する JAR または EAR ファイルの名前を指定します。
-uri <client_jar_in_ear>	EAR ファイルにメイン クラスを設定する場合は、 EAR で -uri オプションを使用して、クライアント JAR の URI（Uniform Resource Identifier）パスを特定する必要があります。
-class <main_classname>	指定したクライアント JAR 内でメイン クラスとして設定されるクラス名を指定します。 このクラスは、クライアント JAR ファイル内に存在し、main() メソッドを含む必要があります。

例
次のサンプルは、スタンドアロンのクライアント JAR のメイン クラスを設定します。

```
iastool -setmain -jar myclient.jar -class com.bes.myjclass
```

次のサンプルは、EAR ファイル内に含まれるクライアント JAR のメイン クラスを設定します。

```
iastool -setmain -jar myapp.ear -uri base/myapps/myclient.jar
-class com.bes.myjclass
```

start

このツールを使用して、指定したハブおよび設定上の管理オブジェクトを起動します。

構文

```
-start <-hub <hub> | -host <host>:<listener_port>> -cfg <configname>
-mo <managedobjectname> -moagent <managedobjectagent> [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

デフォルト出力
デフォルト出力には、起動された管理オブジェクトが表示されます。

オプション
start ツールで使用できるオプションを次の表に示します。

オプション	説明
-hub <hub>	起動する管理オブジェクトが存在するハブ名を指定します。
-host <host>:<listener_port>	目的の管理オブジェクトが稼動しているマシンのホスト名およびリスナー ポートを指定します。 このオプションにより、iastool ユーティリティは、 iastool が実行されているマシンとは異なるサブネット上のハブを検索できます。
-cfg <configname>	目的の管理オブジェクトを含む設定の名前を指定します。
-mo <managedobjectname>	目的の管理オブジェクトの名前を指定します。
-moagent <managedobjectagent>	管理オブジェクトのエージェント名を指定します。 指定したハブに複数のエージェントがある場合は、このオプションを使用します。
-mgmtport <nnnnn>	指定したハブで使用する管理ポート番号を指定します。 指定しない場合、 デフォルトは 42424 になります。
-realm <realm>	ユーザーとパスワードのオプションを指定したら、ユーザー認証に使用する領域を指定します。
-user <username>	指定した領域に対して認証するユーザーを指定します。

オプション	説明
-pwd <password>	指定した領域に対して認証するユーザーのパスワードを指定します。
-file <login_file>	ユーザーを認証するために使用する領域、ユーザー名、およびパスワードを含むログイン スクリプト ファイルを指定します。このファイルがある場所のフル パスか相対パスが必要です。詳細については、「スクリプト ファイルから iastool コマンドライン ツールを実行する」を参照してください。

例

次のサンプルは、管理ポート 24410 の j2ee 設定内の AppServer1 で実行中のパーティション ネーミング サービスを起動します。

```
iastool -start -hub AppServer1 -cfg j2ee -mo standard_visinaming  -mgmtport
24410
```

stop

このツールを使用して、ハブまたは管理オブジェクトをシャットダウンします。

構文

```
-stop <-hub <hub> | -host <host>:<listener_port>> [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

または

```
-stop <-hub <hub> | -host <host>:<listener_port>> [-mgmtport <nnnnn>]
-cfg <configname> -mo <managedobjectname> -moagent <managedobjectagent>
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

デフォルト出力

デフォルト出力には、シャットダウンされたプロセスが表示されます。

管理オブジェクトがシャットダウンできない場合など、stop ツールが失敗すると、状態コードとともにエラーが表示されます。状態コードは、標準エラー出力（stderr）に返されます。

オプション

stop ツールで使用できるオプションを次の表に示します。

オプション	説明
-hub <hub>	シャットダウンするハブ、またはシャットダウンする管理オブジェクトが存在するハブの名前を指定します。
-host <host>:<listener_port>	目的のハブまたは管理オブジェクトが稼動しているマシンのホスト名およびリスナー ポートを指定します。このオプションにより、iastool ユーティリティは、iastool が実行されているマシンとは異なるサブネット上のハブを検索できます。
-cfg <configname>	目的の管理オブジェクトを含む設定の名前を指定します。
-mo <managedobjectname>	目的の管理オブジェクトの名前を指定します。
-moagent <managedobjectagent>	管理オブジェクトのエージェント名を指定します。指定したハブに複数のエージェントがある場合は、このオプションを使用します。
-mgmtport <nnnnn>	指定したハブで使用する管理ポート番号を指定します。デフォルトは 42424 です。
-realm <realm>	ユーザーとパスワードのオプションを指定したら、ユーザー認証に使用する領域を指定します。
-user <username>	指定した領域に対して認証するユーザーを指定します。

オプション	説明
-pwd <password>	指定した領域に対して認証するユーザーのパスワードを指定します。
-file <login_file>	ユーザーを認証するために使用する領域、ユーザー名、およびパスワードを含むログイン スクリプト ファイルを指定します。このファイルがある場所のフル パスか相対パスが必要です。詳細については、「スクリプト ファイルから iastool コマンドライン ツールを実行する」を参照してください。

例

次のサンプルは、管理ポート 24410 の j2ee 設定内の AppServer1 で実行中のパーティション ネーミング サービスを停止します。

```
iastool -stop -hub AppServer1 -cfg j2ee -mo standard_visinaming -mgmtport 24410
```

uncompress

このツールを使用して、JAR ファイルを解凍します。

構文

```
-uncompress -src <srcjar> -target <targetjar>
```

デフォルト出力

デフォルトでは、uncompress は操作が成功したかどうかを報告します。

オプション

uncompress ツールで使用できるオプションを次の表に示します。

オプション	説明
-src <srcjar>	解凍する JAR ファイルを指定します。ファイルがある場所のフル パスか相対パスが必要です。 デフォルト値はありません。
-target <targetjar>	生成される解凍 JAR ファイルの名前を指定します。ファイルがある場所のフル パスか相対パスが必要です。 デフォルト値はありません。

例

次に、現在のディレクトリにある圧縮 JAR ファイル small.jar を同じディレクトリの解凍ファイル big.jar に変換するサンプルを示します。

```
iastool -uncompress -src small.jar -target big.jar
```

次に、ディレクトリ c:\myprojects\ にある JAR ファイル small.jar を同じディレクトリのファイル big.jar に解凍するサンプルを示します。

```
iastool -uncompress -src c:\myprojects\small.jar -target c:\myprojects\big.jar
```

undeploy

このツールを使用して、指定したハブと設定内の指定したパーティションから J2EE モジュールをデプロイメント解除します。

構文

```
-undeploy -jar <jar> <-hub <hub> | -host <host>:<listener_port>>
-cfg <config_name> -partition <partitionname> [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

デフォルト出力

デフォルトでは、undeploy ツールは操作が成功したかどうかを報告します。

オプション
undeploy ツールで使用できるオプションを次の表に示します。

オプション	説明
-jar <jar>	デプロイメント解除する JAR ファイルの名前を指定します。ファイルがある場所のフル パスか相対パスが必要です。 デフォルト値はありません。
-hub <hub>	JAR ファイルのデプロイメント解除元のハブの名前を指定します。
-host <host>:<listener_port>	目的のデプロイされたモジュールが存在するマシンのホスト名およびリスナー　ポートを指定します。 このオプションにより、iastool ユーティリティは、iastool が実行されているマシンとは異なるサブネット上のモジュールを検索できます。
-cfg <configname>	パーティションの設定名を指定します。
-partition <partitionname>	JAR ファイルを含むパーティション名を指定します。
-mgmtport <nnnnn>	指定したハブで使用する管理ポート番号を指定します。 指定しない場合、 デフォルトは 42424 になります。
-realm <realm>	ユーザーとパスワードのオプションを指定したら、 ユーザー認証に使用する領域を指定します。
-user <username>	指定した領域に対して認証するユーザーを指定します。
-pwd <password>	指定した領域に対して認証するユーザーのパスワードを指定します。
-file <login_file>	ユーザーを認証するために使用する領域、ユーザー名、およびパスワードを含むログイン スクリプト ファイルを指定します。 このファイルがある場所のフル パスか相対パスが必要です。 詳細については、「スクリプト ファイルから iastool コマンドライン ツールを実行する」を参照してください。

unmanage

このツールを使用して、アクティブな管理モードから管理オブジェクトを削除します。

構文

```
-unmanage  (-hub <hub>|-host <host>:<listener_port>) [-cfg <configname>] -mo
<managedobjectname> [-moagent <managedobjectagent>] [-mgmtport <99999>] [-realm
<realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

デフォルト出力
デフォルト出力では標準出力（stdout）に何も返されません。

オプション
unmanage ツールで使用できるオプションを次の表に示します。

オプション	説明
-hub <hub>	管理オブジェクトが稼動しているハブの名前を指定します。
-host <host>:<listener_port>	管理オブジェクトが稼動しているマシンのホスト名およびリスナー ポートを指定します。 このオプションにより、 iastool ユーティリティは、iastool が稼動しているマシンとは異なるサブネット上のハブを検索できます。
-cfg <configname>	管理オブジェクトが関連付けられている設定の名前を指定します。
-mo <managedobjectname>	管理オブジェクトの名前を指定します。

オプション	説明
-moagent <managedobjectagent>	管理オブジェクトが関連付けられているエージェントを指定します。
-mgmtport <99999>	管理オブジェクトのエージェントが関連付けられているポートを指定します。
-realm <realm>	ユーザーとパスワードのオプションを指定したら、ユーザー認証に使用する領域を指定します。
-user <username>	指定した領域に対して認証するユーザーを指定します。
-pwd <password>	指定した領域に対して認証するユーザーのパスワードを指定します。
-file <login_file>	ユーザーを認証するために使用する領域、ユーザー名、およびパスワードを含むログイン スクリプト ファイルを指定します。このファイルがある場所のフル パスか相対パスが必要です。詳細については、「スクリプト ファイルから iastool コマンドライン ツールを実行する」を参照してください。

例

次のサンプルは、デフォルトの管理ポートを使って管理オブジェクト j2ee-server をアクティブな管理モードから削除します。

```
iastool -unmanage -hub AppServer1 -cfg j2ee -mo j2ee-server
```

usage

引数なしで呼び出された場合、usage は、認識されたコマンドライン オプションとそれぞれの簡潔な説明の一覧を表示します。複数の引数を持つ usage を呼び出すと、特定のコマンドとその引数の詳細な説明が表示されます。

構文

```
-usage
-usage <tool>
-usage <tool1 tool2 tool3>
```

メモ usage コマンドへの引数には、行間隔を確保するハイフンは不要です。

デフォルト出力

デフォルトでは、usage ツールは各コマンドライン ツールの一覧と簡単な説明を表示します。

例

次のサンプルは、各コマンドライン ツールの一覧と簡単な説明を表示します。

```
iastool -usage
```

次のサンプルは、compress ツールの詳細な説明のサンプルを示します。

```
iastool -usage compress
```

次のサンプルは、-start、-stop、および -restart ツールの詳細な説明のサンプルを示します。

```
iastool -usage start stop restart
```

verify

このツールは、アーカイブ ファイルの正当性と整合性を確認し、アプリケーションのデプロイメントに必要な要素がすべて所定の位置にあるかどうかを確認します。

次に、検証プロセスがサポートする、アプリケーションの存続期間のフェーズおよび適切な検証レベルに対応する役割（J2EE 役割定義と同様）を示します。

- **DEVELOPER：**一番低い確認レベルです。すべての XML 構文と、現在のアーカイブの種類に関連した標準または独自のキーワードがチェックされます。アーカイブ ファイルの整合性はチェックされますが、このレベルでは外部リソースは確認されません。
- **ASSEMBLER：**アーカイブを個別に確認してエラーがないことを確認した後で、アプリケーションに組み込まれたほかのリソースを確認します。たとえば、このレベルは URI（Uniform Resource Identifiers）の存在と正当性は検証しますが、EJB リンクや JNDI リンクは検証しません。
- **DEPLOYER：**（デフォルト）すべてのチェックがオンになっています。このレベルでは、アプリケーションがデプロイメントされる動作環境だけでなく、EJB リンクや JNDI リンクもチェックされます。

サポートされているアーカイブの種類は、EAR、EJB、WAR、JNDI およびクライアント JAR です。アーカイブの確認プロセスでは、一般に次のようなチェックが行われます。

- XML 構文に対してコードが正しいかどうかをチェックする XML コードのパスオーバー
- 標準または独自の XML デスクリプタの意味と、サポートされている各アーカイブの種類に対して必要なデスクリプタの準拠性の確認。

確認は、常に最上位のモジュールからその下位モジュールへと順に階層的に行われ、最後にアーカイブ間のリンクがチェックされます。

構文

```
-verify -src <srcjar> [-role <DEVELOPER|ASSEMBLER|DEPLOYER>] [-nowarn]
[-strict] [-classpath <classpath>]
```

デフォルト出力

デフォルトでは、指定したモジュールでエラーが見つからなかった場合など、`verify` は何も報告しません。

オプション

`verify` ツールで使用できるオプションを次の表に示します。

オプション	説明
`-src <srcjar>`	検証する JAR ファイル（または拡張 JAR のディレクトリ）を指定します。 ファイルがある場所のフル パスか相対パスが必要です。 デフォルト値はありません。
`-role <DEVELOPER|ASSEMBLER|DEPLOYER>`	実行するエラーチェックのレベルを指定します。 ■ `DEVELOPER` ■ `ASSEMBLER` ■ `DEPLOYER`（デフォルト） 詳細については、 上述の役割の説明を参照してください。
`-nowarn`	ツールに対して、 デプロイメントを不可能にしているエラーだけを報告し、警告は報告しないように指定します。
`-strict`	ツールに対して、最も詳しいレベルの矛盾点を報告するように指定します。 矛盾点の多くは、アプリケーション全体の整合性に影響しません。
`-classpath <classpath>`	アプリケーション クラスとリソースの検索パスを指定します。 1 つまたは複数のディレクトリ、ZIP、または JAR ファイル エントリを入力する場合は、 各エントリをセミコロン（;）で区切ります。

例

次のサンプルは、`c:\examples\soap` ディレクトリ内にある JAR ファイル `soap-client.jar` の DEVELOPER（**開発者**）レベルの検証を実行します。

```
-verify -src c:\examples\soap\soap-client.jar -role DEVELOPER
```

スクリプト ファイルから iastool コマンドライン ツールを実行する

iastool ユーティリティ ツールを使用するには、ログイン情報（領域、ユーザー名、およびパスワード）の入力が必要な場合があります。スクリプト ファイルから iastool コマンドを実行する場合にログイン情報を入力すると、スクリプト ファイルにアクセスできるユーザーすべてに、領域、ユーザー名、パスワード情報が露出してしまいます。この情報を保護するには、次の 2 つの方法があります。

- 領域、ユーザー名、およびパスワード情報をファイルに入力し、そのファイルをコマンドにパイプする。
- 領域、ユーザー名、パスワード情報をファイルに入力し、-file オプションが設定されたコマンドにファイルを渡す。

ファイルを iastool ユーティリティにパイプする

次のサンプルは、デフォルトの Borland デプロイメント プラットフォームのインストール ディレクトリにあるファイル mylogin.txt を iastool ユーティリティにパイプして、east1 というハブを ping します。

```
iastool -ping -hub east1 < c:\AppServer\mylogin.txt
```

ここで、ファイル mylogin.txt 内の 3 行は、入力した領域、ユーザー名、およびパスワードと一致します。

```
2
username
password
```

メモ ファイルの内容は、コマンドラインで入力した内容と完全に一致します。ファイルの最初のエントリは、領域名ではなく realm オプションです。ただし、realm オプションを指定しないで ping ツールを実行すると、番号の一覧が表示され、選択することができます。2 行目は username、3 行目は password です。このファイルは、iastool ユーティリティで読み込め、許可されていないユーザーには読み込めないように、セキュリティで保護されます。

ファイルを iastool ユーティリティに渡す

次のサンプルは、-file オプションを使用して、iastool ユーティリティにファイルを渡して、east1 というハブを ping します。

```
iastool -ping -hub east1 -file c:\AppServer\mylogin.txt
```

ここで、mylogin.txt には、次の形式があります。

```
Default Login
Smart Agent port number
username
password
false
ServerRealm
```

-file オプションでは、完全なファイル名（ファイル名と相対パスまたは絶対パス）を提供する必要があります。ファイルを iastool ユーティリティに渡すと、3 番目（username）、4 番目（password）、および 6 番目（realm name）の行だけが使用されます。その他の行は省略できませんが、iastool ユーティリティはそこに含まれる情報を無視します。たとえば、次のようになります。

```
Default Login
12448
myusername
mypassword
false
ServerRealm
```

第 37 章

パーティション XML リファレンス

ここでは、パーティションの partition.xml 設定ファイルの XML 定義について説明します。このファイルには、パーティションの設定の中心になるメタデータが含まれます。

<partition> 要素

partition 要素は、Borland AppServer（AppServer）パーティションの設定を制御する設定が定義された属性や下位要素を含むスキーマのルート ノードです。

属性	説明
version	パーティションの製品バージョン
name	パーティションの名前。
description	パーティションの説明

構文

```
<partition version="version number" name="partition name"
description="description">
    .
    .
    .
</partition>
```

partition 要素は、次の下位要素を含みます。

- <jmx>
- <statistics.agent>
- <security>
- <container>
- <user.orb>
- <management.orb>
- <shutdown>
- <services>
- <archives>

<jmx> 要素

jmx 要素には、JMX エージェントを設定するための下位要素があります。

jmx 要素は、次の下位要素を含みます。

- <mbean.server>
- <mlet.service>
- <http.adaptor>
- <rmi-iiop.adaptor>

<mbean.server> 要素

mbean.server 要素は、JMX エージェントの MBean サーバーを有効または無効にするために使用します。MBean サーバーは、JMX のエージェント仕様レベルで定義されるインターフェースとファクトリオブジェクトです。

属性	説明
enable	MBean サーバーを有効または無効にします。 有効な値は、true（デフォルト）または false です。

<mlet.service> 要素

mlet.service 要素は、JMX エージェントの MLet サービスを設定します。MLet サービスによって、MBean サーバーの JVM 内の MBean クラスとリソースを 1 つの操作で簡単にリモートホストからロードして登録できます。

属性	説明
enable	MLet サービスを有効または無効にします。 有効な値は、true または false（デフォルト）です。

<http.adaptor> 要素

http.adaptor 要素は、JMX エージェントの HTTP アダプタを設定します。HTTP アダプタは、HTML 3.2 準拠のブラウザまたはアプリケーションを使ってパーティションを管理するための HTTP プロトコルのアダプタです。

属性	説明
enable	HTTP アダプタを有効または無効にします。 有効な値は、true または false（デフォルト）です。
port	HTTP アダプタが監視するポートです。デフォルトのポート番号は 8082 です。
host	サーバーが監視するホスト名を定義します。デフォルト設定（localhost）のままにすると、ほかのコンピュータからサーバーにアクセスできません。この設定はセキュリティ上の理由から適切で、サーバーを外部に開く場合は明示的に開く必要があります。すべてのローカルインターフェースに対してサーバーを開く 0.0.0.0 も使用できます。デフォルトは localhost です。
authentication.method	認証メソッドを設定します。有効な値は、none、basic、digest です。デフォルトは none です。
socket.factory.name	ObjectName を使ってデフォルトのソケットファクトリを別のソケットファクトリに置き換えます。指定された MBean には public ServerSocket createServerSocket(int port, int backlog, String host) throws IOException メソッドが必要です。
processor.name	XML プロセッサとして使用する MBean の ObjectName を設定します。MBean は、mx4j.tools.adaptor.http.ProcessorMBean インターフェースを実装する必要があります。

http.adaptor 要素は、次の下位要素を含みます。

- <xslt.processor>

<xslt.processor> 要素

xslt.processor 要素は、HTTP アダプタの XSLT プロセッサを設定します。XSLT プロセッサは、未処理の XML を Web ブラウザで表示可能な XML に変換します。このプロパティが有効でない場合に MX4J Web コンソールを使用すると、Web ブラウザに未処理の XML が表示されます。

属性	説明
enable	XSLT プロセッサを有効または無効にします。 有効な値は、true（デフォルト）または false です。
File	XSL ファイルを探す場所を指定します。 ターゲットファイルがディレクトリの場合は、 XSL ファイルがそのディレクトリにあるとみなされます。 それ以外の場合で、 JAR ファイルまたは ZIP ファイルをポイントする場合、ファイルはその中にあるとみなされます。ファイルシステムをポイントする方法は、 テストに使用する場合に便利です。
PathInJar	XSL ファイルがある JAR ファイル内のディレクトリを設定します。
LocaleString	文字列を使ってロケールを設定します。 デフォルトは en です。
UseCache	変換オブジェクトをキャッシュするかどうかを指定します。これにより、変換処理が速くなります。 通常は true に設定しますが、テストを簡単にする場合は false に設定します。デフォルトは true です。

<rmi-iiop.adaptor> 要素

rmi-iiop.adaptor 要素は、JMX エージェントの RMI-IIOP アダプタを設定します。RMI-IIOP アダプタは、クライアントフレームワークに基づくので、マネージャまたは管理アプリケーションが RMI を使って MBean サーバーと通信する場合に役立ちます。

属性	説明
enable	RMI-IIOP アダプタを有効または無効にします。 有効な値は、 true（デフォルト）または false です。
port	RMI-IIOP アダプタポートに割り当てられるポート番号。 ポート番号を指定しないか、 ポート番号に 0 を指定すると、 ランダムなポート番号が割り当てられます。

<statistics.agent> 要素

statistics.agent 要素は、パーティションの統計情報エージェントを設定します。パーティションの統計情報エージェントは、次の 2 つのコンポーネントで構成されます。

- パーティションの統計情報データを定期的に収集し、そのデータをディスクに保存する統計情報コレクタ。これらの定期的なデータサンプルはディスクに保存され、製品ツールがパーティションに関する現在および時系列の統計データを提供する基礎になります。
- 履歴データをディスクから定期的に*削除*（クリーンアップ）する統計情報の解放機能。

パーティションの統計情報エージェントは、短期間の統計データを収集することが目的です。ただし、使用できるディスク領域は、物理的な容量にだけ制限されます。

属性	説明
enable	統計情報エージェントを有効または無効にします。無効な統計情報エージェントは、 統計情報データの収集や削除を行いません。 有効な値は、 true（デフォルト） または false です。
level	パーティションから収集する統計情報の詳細レベルを設定します。有効な値は、none、minimum（デフォルト）、および maximum です。
snapshot.period_secs	パーティションの統計情報を収集してディスクに書き込む頻度（秒単位）を指定します。 デフォルトは 10 秒です。
reap.enable	ディスク上のパーティション統計情報データの解放（クリーンアップ）を有効または無効にします。 有効な値は、true（デフォルト） または false です。
reap.older_than_secs	reap_enable が true の場合、 統計情報データが削除されるまでディスク上に存続できる時間のしきい値（秒単位）を設定します。デフォルトは 600 秒（10 分）です。
reap.period_secs	reap_enable が true の場合、reap.older_than_secs より古い統計情報データをディスクからクリーンアップするための消去間隔（秒単位）を設定します。 デフォルトは 60 秒（1 分）です。

<security> 要素

security 要素では、指定されたパーティションのセキュリティを設定できます。この空の要素には、次の表で説明する属性が含まれます。

属性	説明
enable	パーティションのセキュリティを有効または無効にします。 有効な値は、true（デフォルト） または false です。
manager	パーティションが使用するセキュリティマネージャの名前を指定します。 有効な値は、 利用可能なセキュリティプロバイダの名前です。その例を次に示します。 com.borland.security.provider.CertificateWallet.
policy	パーティションのセキュリティ規則を定義したセキュリティポリシーファイルの名前を指定します。有効な値は、完全修飾されたセキュリティポリシーファイルの名前です。 その例を次に示します。 <install_dir>/va/¥security/profile/¥management/java_security.policy

<container> 要素

container 要素では、パーティションでのクラスロードの使い方を指定します。

属性	説明
system.classload.prefixes	これは、 カンマ区切りのリソースプレフィクスのリストです。 カスタムクラスローダーは、 それ自体のロードを試行する前に、 システムクラスローダーにこのリソースプレフィクスを委任します。
verify.on.load	true が指定されている場合、ロードされる JAR に対して verify を実行します。デフォルトは true です。

属性	説明
classloader.policy	パーティションが使用するクラスローダーのタイプを決定します。 有効な値は、 per_module または container です。 per_module クラスローダーのポリシーは、 各デプロイメントモジュールに対して個別のアプリケーションクラスローダーを作成します。このポリシーは、動的デプロイメントを有効にする場合に必要です。 container ポリシーは、すべてのデプロイメントモジュールを共有クラスローダーにロードします。 このポリシーが選択されている場合は、 動的デプロイメントは実行できません。
classloader.classpath	アプリケーションクラスローダーの各インスタンスがロードする JAR ファイルをセミコロン（;）で区切ったリストを含みます。 これは、これらの JAR をすべてのモジュールにバンドルするのと同じ意味を持ちます。

<user.orb> 要素

user.orb 要素は、パーティションのユーザードメイン ORB に使用される VisiBroker 設定を制御します。

属性	説明
orb.propstorage	パーティションのユーザー ORB プロパティファイルのパス。相対パスは、パーティションのプロパティディレクトリ（partition.xml が存在するディレクトリ）からの相対パスです。
use.default.smartagent.port	このプロパティは、 パーティションがスマートエージェントのポート値を識別するときに、 SCU スマートエージェントの設定を使用かどうかを定義します。
use.default.smartagent.addr	このプロパティは、 パーティションがスマートエージェントのホストアドレス値を識別するときに、SCU スマートエージェントの設定を使用かどうかを定義します。

<management.orb> 要素

management.orb 要素は、パーティションの管理ドメイン ORB の VisiBroker 設定を制御します。

属性	説明
orb.propstorage	パーティションの管理ドメイン ORB プロパティファイルのパス。
required_roles.propstorage	パーティションの管理ドメイン ORB の必須ロール設定ファイルのパス
runas.propstorage	パーティションの管理ドメイン ORB の runas 設定ファイルのパス

すべてのパスは、パーティションのプロパティディレクトリ（partition.xml が存在するディレクトリ）からの相対パスです。

<shutdown> 要素

shutdown 要素は、パーティションの停止時に実行される処理を決定します。この空の要素に属性はありません。

属性	説明
dump_threads	パーティションのシャットダウン時に実行しているスレッドの診断情報をダンプするかどうかを示します。
dump_threads.count	この値を指定すると、シャットダウン時にスレッドの状態がその回数だけダンプされます。これにより、一部のスレッドが単に終了までに時間がかかっている場合に、最終的に終了することを確認できます。
delay.1	サポート用に使用する予定です。
garbage_collection.1	サポート用に使用する予定です。
delay.2	サポート用に使用する予定です。
runfinalizersonexit	サポート用に使用する予定です。
delay.3	サポート用に使用する予定です。
garbage_collection.2	サポート用に使用する予定です。
delay.4	サポート用に使用する予定です。
runfinalization	サポート用に使用する予定です。

<services> 要素

services 要素では、パーティションのサービスを設定できます。各パーティションサービスには、固有の設定を持つ service 下位要素があります。services 要素自体には、次の属性があります。

属性	説明
autostart	パーティションの起動時に開始されるパーティションサービスのリスト。 値は、スペース区切りのパーティションサービス名のリストです。
startorder	autostart 属性によって起動するように設定されたパーティションサービスに適用される開始順序。指定されていないパーティションサービスは、指定されたパーティションサービスの後に開始されます。 有効な値は、開始順序（左から右の順序）を表すスペース区切りのパーティションサービス名のリストです。
shutdownorder	パーティションのシャットダウン時に実行しているパーティションサービスに適用されるシャットダウン順序。指定されていないパーティションサービスは、指定されたパーティションサービスの前に停止されます。 有効な値は、シャットダウン順序（左から右の順序）を表すスペース区切りのパーティションサービス名のリストです。
administer	ユーザーに表示されるパーティションサービスのリスト。パーティションサービスを一覧表示すると、ツール内に表示されます。

<services> 要素は、次の下位要素を含みます。

- service

<service> 要素

<service> 要素により、パーティションサービスを設定できます。ここに含まれる属性は、パーティションによるサービスの管理、およびサービスの設定メタデータを含む properties 下位要素を制御します。

属性	説明
name	パーティションサービスの名前
version	パーティションサービスのバージョン
description	パーティションサービスの説明
vendor	パーティションサービスのベンダーの説明

属性	説明
class	パーティションのサービスプラグインアーキテクチャを実装し、サービスに管理インターフェースと制御インターフェースを提供するクラスです。
in.management.domain	サービスがパーティションの管理ドメインで実行するか、パーティションのユーザードメインで実行するかを示すフラグです。
startup.synchronization	サービスの開始時に実行される同期化のタイプ。有効な値は次のとおりです。 ■ service_ready - サービスの準備完了を startup.service_ready.max_wait ミリ秒まで待機します。 ■ delay - 常に startup.delay ミリ秒間待機します。準備完了までサービスを監視しません。 デフォルトは非同期です。
startup.service_ready.max_wait	startup.synchronization 値が service_ready の場合に、パーティションがサービスの開始を待機する最大時間（ミリ秒単位）を制限します。0 の値は、時間が制限されていないことを表します。デフォルト値は 0 です。
startup.delay	startup.synchronization 値が delay の場合に、パーティションがサービスに開始の機会を与えるために待機する時間（ミリ秒単位）を定義します。0 は、無限に待機することを表します。デフォルトは 0 です。
shutdown.synchronization	サービスのシャットダウン時に実行される同期化のタイプ。有効な値は次のとおりです。 ■ service_shutdown - サービスの停止を shutdown.service_shutdown.max_wait ミリ秒まで待機します。 ■ delay - 常に shutdown.delay ミリ秒間待機します。停止までサービスを監視しません。 デフォルトは非同期です。
shutdown.service_shutdown.max_wait	shutdown.synchronization 値が service_shutdown の場合に、パーティションがサービスの停止を待機する最大時間（ミリ秒単位）を制限します。0 の値は、時間が制限されないことを表します。デフォルト値は 0 です。
shutdown.delay	shutdown.synchronization 値が delay の場合に、パーティションがサービスに停止の機会を与えるために待機する時間（ミリ秒単位）を定義します。0 は、無限に待機することを表します。デフォルトは 0 です。
shutdown.phase	このプロパティは、サービスがシャットダウンするパーティションのシャットダウンフェーズを制御します。パーティションは 2 つのフェーズでシャットダウンします。最初のフェーズでは、ユーザー機能を提供するすべてのサービスとコンポーネントがシャットダウンし、2 番めのフェーズでは、パーティションの独自のインフラストラクチャがシャットダウンします。有効な値は、1（デフォルト）と 2 です。 一般にパーティションサービスは、フェーズ 2 ではシャットダウンしません。

<properties> 要素

properties 要素は、特定のサービスの設定メタデータを提供します。

<archives> 要素

archives 要素は、パーティションがホストできるアーカイブの設定メタデータを含みます。特定のアーカイブは、そのアーカイブに固有の属性を含む archive 下位要素を持つことができます。アーカイブは、archive 下位要素を持つ必要はありません。

属性	説明
ear.repository.path	パーティションの EAR ディレクトリのパス。 このディレクトリに存在するすべての EAR は、archive 要素で無効にされていない限り、起動時にパーティションによってロードされます。
war.repository.path	パーティションの WAR ディレクトリのパス。 このディレクトリに存在するすべての WAR は、 archive 要素で無効にされていない限り、 起動時にパーティションによってロードされます。
ejbjar.repository.path	パーティションの EJB jars ディレクトリのパス。 このディレクトリに存在するすべての EJB jars は、 archive 要素で無効にされていない限り、 起動時にパーティションによってロードされます。
rar.repository.path	パーティションの RAR ディレクトリのパス。 このディレクトリに存在するすべての RAR は、archive 要素で無効にされていない限り、起動時にパーティションによってロードされます。
dar.repository.path	パーティションの DAR ディレクトリのパス。 このディレクトリに存在するすべての DAR は、 archive 要素で無効にされていない限り、 起動時にパーティションによってロードされます。
lib.repository.path	パーティションの lib ディレクトリのパス。 このディレクトリに存在するすべての JAR ファイルは、 パーティションのシステムクラスパスに置かれます。
classes.repository.path	パーティションの classes ディレクトリのパス。このディレクトリに存在するすべてのクラスは、パーティションのシステムクラスパスに置かれます。

すべてのパスは、パーティションのルートディレクトリからの相対パスです。

<archive> 要素

archive 要素は、アーカイブに固有の設定メタデータを含みます。パーティションのアーカイブリポジトリディレクトリにあるアーカイブは、デフォルト以外の設定を適用する必要がある場合を除いて、archive 要素を必要としません。

属性	説明
name	この要素が関係するアーカイブの名前。 アーカイブのファイル名です。
disable	起動時にパーティションの該当するアーカイブのホストを無効にするためのフラグ。 有効な値は、 true または false（デフォルト）です。
path	パーティションリポジトリの外部に存在するアーカイブのパス。 指定されたパスから、 アーカイブをホストするパーティションを取得するために使用します。

すべてのパスは、パーティションのルートディレクトリからの相対パスです。

第 38 章

EJB、JSS、および JTS のプロパティ

EJB コンテナレベルのプロパティ

EJB コンテナのプロパティを partition.xml ファイルに設定します（各パーティションには独自のプロパティファイルがあります）。このファイルは、次のディレクトリにあります。

```
<install_dir>/var/domains/base/configurations/configuration_name/mos/
partition_name/adm/properties
```

プロパティ	説明	デフォルト値
ejb.copy_arguments=true|false	このフラグを指定すると、引数が Bean 内のインプロセス呼び出しにコピーされます。 デフォルトでは、 Bean 内での呼び出しは参照渡しのセマンティクスを使用します。 このフラグを有効にすると、 Bean 内での呼び出しで値渡しのセマンティクスが使用されます。 **メモ：**値渡しのセマンティクスを使用すると、 多くの EJB の実行速度がかなり遅くなります。	false
ejb.use_java_serialization=true|false	設定すると、セッション永続性などについて IIOP シリアライゼーションの使用が Java シリアライゼーションで上書きされます。	false

プロパティ	説明	デフォルト値
ejb.useDynamicStubs=true|false	このプロパティは、 ローカルインターフェースを持つ CMP 2.0 エンティティ Bean にだけ関連します。 プロパティが設定されている場合、 コンテナは動的なプロキシベースの方法で呼び出しをディスパッチします（軽量で非 CORBA のカスタムリファレンスの作成）。 設定されていない場合、 コンテナは CORBA を使って呼び出しをディスパッチします。 これらのローカルな動的スタブには、 呼び出し元と呼び出される側が同じ VM に存在するために多くの最適化が用意されています。これにより、Bean に対して CORBA 層を介さずに直接的なディスパッチを実行できます。 また、 動的スタブは EJB コンテナのデータ構造に対応しているため、ターゲットの Bean に高速でアクセスできます。 現時点では、 スタブジェネレータ java2iiop（iastool から呼び出されるか、 直接呼び出される）も、 アーカイブ内のすべてのインターフェース用のスタブを生成します。 `ejb.useDynamicStubs` がアクティブな場合は、 選択した CMP 2.0 Bean に対応するスタブのサブセットは無視されます。 この機能を使用すると、 ディスパッチメカニズム全体が動的になり、サーバー側に動的スケルトンが提供されるだけでなく、 クライアント側にも動的スタブが提供されます。 静的に生成されたアーカイブ内のすべてのスタブクラスとスケルトンクラスは無視されます。 プロパティは Bean 内で設定します。 ただし、 すべてのエンティティ Bean でプロパティを使用しても問題がなければ、デプロイメントデスクリプタの EAR レベルでプロパティを設定するのが最も簡単な方法です。 **重要：**このプロパティは、 `ejb.usePKHashCodeAndEquals` と連係して使用する必要があります。	true
ejb.usePKHashCodeAndEquals=true|false	アクティブキャッシュ（ TxReady キャッシュ）と関連キャッシュ（ Ready Bean キャッシュ）をサポートするデータ構造は、 `java.util.Hashtable` と `java.util.HashMap` を使用します。これらのデータ構造にプールされた値（ エンティティ Bean のインスタンス）は、キャッシュされるエンティティ Bean の主キー値に関連付けられています。Hashtable のインプリメンテーションは、値の配置や検索に使用するキーの計算用 hashCode() メソッドと呼び出し用 equals() メソッドに依存します。これらのデータ構造はクリティカルなコードパスにあり、エンティティ Bean のメソッドに呼び出しをディスパッチしている間、コンテナによって頻繁にアクセスされます。Borland AppServer でのデフォルトは、reflection ベースの計算です。このプロパティが設定されている場合、コンテナはユーザーが提供する equals() メソッドと hashCode() メソッドのインプリメンテーションを使用します。	true
ejb.no_sleep=true|false	通常は、コンテナを埋め込むメインプログラムから設定します。 このプロパティを設定すると、EJB コンテナは現在のスレッドをブロックせず、 制御がユーザーコードに戻されます。	false
ejb.trace_container=true|false	コンテナが実行中の処理をユーザーに通知する便利なデバッグ情報を有効にします。デバッグメッセージインターセプタをインストールします。	false
ejb.xml_validation=true|false	設定すると、 デプロイメント時の DTD に対して XML デスクリプタが有効になります。	true
ejb.xml_verification=true|false	設定すると、 J2EE アーカイブがデプロイメント時に検証されます。	false
ejb.classload_policy=per_module|container|none	スタンドアロン EJB コンテナのクラスローディングの動作を定義します。 パーティションには適用されません。 `per_module` が設定されている場合、 コンテナは、 デプロイメントされた各 J2EE アーカイブごとにカスタムクラスローダーの新しいインスタンスを使用します。 `none` が設定されている場合、コンテナはシステムクラスローダーを使用します。 動的デプロイメントと EAR のデプロイメントは、 このモードでは動作しません。 `container` が設定されている場合、コンテナは単一のカスタムクラスローダーを使用します。 これにより、EAR のデプロイメントが有効になりますが、 動的デプロイメント機能は無効になります。	per_module

プロパティ	説明	デフォルト値
ejb.module_preload=true|false	デプロイメント時に J2EE アーカイブ全体をメモリにロードします。これで、アーカイブを上書きしたり、再ビルドできます。このオプションは、スタンドアロンの EJB コンテナを実行している JBuilder では必須です。	false
ejb.system_classpath_first=true|false	true に設定すると、カスタムクラスローダーは、最初にシステムのクラスパスを参照します。	false
ejb.sfsb.keep_alive_timeout=<num>	ejb-borland.xml デスクリプタで使用される <timeout> 要素のデフォルト値を定義します。このプロパティは、<timeout> 要素がスキップされるか 0 に設定される EJB に影響を及ぼします。このプロパティは、ステートフルセッション Bean が非アクティブ化された後に、永続的ストレージ（JSS）でその非アクティブなステートフルセッション Bean が保持される時間を秒単位で定義します。この時間が経過すると、JSS は永続的ストレージからそのセッションの状態を削除します。削除されると、後からアクティブ化することはできません。	86400 (=24 hours)
ejb.cacheTimeout=<integer>	このプロパティは、指定されたタイムアウト期間が過ぎたらエンティティ Bean のデータフィールドを無効にするようにコンテナに指示します。プロパティは間隔を指定して使用します。コンテナは、この間隔が過ぎるまで、データベースから Bean の状態をロードせずにキャッシュされた状態を使用します。指定された期間の終了時に、コンテナは Bean をダーティとしてマークを付けます（ただし、主キーとの関連付けは維持されます）。これにより、インスタンスは、Bean が新しいトランザクションで使用される前に、キャッシュではなくデータベースから Bean の状態をロードします。このプロパティは、頻繁に変更されないエンティティ Bean で使用します。 プロパティは、秒単位のキャッシュ間隔を表す正の整数です。 これは、コミットモード A でのみ有効です。それ以外のコミットモードで指定された場合は無視されます。	0（タイムアウトなし）
ejb.sfsb.aggressive_passivation=true|false	true に設定すると、ステートフルセッション Bean は、最後に使用されてからの時間に関係なく、非アクティブ化されます。これによってフェイルオーバーサポートが有効になるため、EJB コンテナが失敗した場合は、クラスタ内の EJB コンテナの 1 つによって、最後に保存された状態からセッションを回復できます。false に設定すると、最後の非アクティブ化が試行されてから使用されていない Bean だけが JSS に対して非アクティブ化されます。これにより、フェイルオーバーのサポートは確実性が低下しますが、速度は向上します。パフォーマンスより可用性の高さを重視する場合は、この設定を使用してください。	true
ejb.sfsb.factory_name=<string>	設定すると、ステートフルセッション Bean は同じ EJB コンテナまたはパーティション内で実行されている JSS とは異なる JSS を使用します。使用する JSS のファクトリ名を指定します。これは、スマートエージェント（osagent）に登録されている JSS の名前です。	なし
ejb.logging.verbose=true|false	true に設定すると、EJB コンテナは予期しない状況に関するメッセージを記録します。ユーザーは、このメッセージに注意する必要があります。メッセージは、>>>> EJB LOG <<<< ヘッダーでマークされます。false に設定すると、これらのメッセージは記録されません。	true
ejb.logging.doFullExceptionLogging=true|false	設定すると、コンテナは、EJB インプリメンテーションで生成された予期しないすべての例外を記録します。	false
ejb.jss.pstore_location=<path>	JSS バックエンドストレージとして使用されるファイルのデフォルトの名前と場所を上書きします。スタンドアロンの EJB コンテナだけに適用されます。このオプションは使用されなくなりました。かわりに、jss.pstore と jss.workingDir を使用します。	なし
ejb.jdb.pstore_location=<path>	データベースサービスによって使用されるファイルのデフォルトの名前と場所を上書きします。スタンドアロンの EJB コンテナだけに適用されます。	なし

プロパティ	説明	デフォルト値
ejb.interop.marshal_handle_as_ior=true|false	true に設定すると、 javax.ejb.Handle の各インスタンスが CORBA IOR としてマーシャリングされます。 そうでない場合は、 CORBA 抽象インターフェースとしてマーシャリングされます。詳細については、 CORBA IIOP の仕様を参照してください。	false
ejb.finder.no_custom_marshal=true|false	マルチオブジェクトファインダがオブジェクトのコレクションを戻す場合、 EJB コンテナはデフォルトで次を実行します。 ■ カスタム Vector インプリメンテーションを作成し、呼び出し元に返します。 ■ 返された Vector に対してファインダの呼び出し元が参照／繰り返しを実行するのに合わせて、必要になった時点で IOR を（主キーから）作成します。 ■ Vector 全体の IOR を計算します。結果は、IOR が作成された JVM に残されます。 このプロパティを true に設定した場合、 EJB コンテナは上記のいずれも実行しません。	false
ejb.collect.stats_gather_frequency=<num>	コンテナ統計情報のプリントアウトの秒単位の時間間隔。 ゼロが設定されている場合は、 統計収集を無効にします。 その場合、 統計収集が実行されないため、 統計情報は表示されません。 ゼロを設定すると、 ejb.collect.display_statistics、 ejb.collect.statistics、 ejb.collect.display_detail_statistics の各プロパティは無視されます。	5
ejb.collect.display_statistics=true|false	このフラグは、タイマー診断を有効にします。これにより、ユーザーはコンテナによる CPU の使用状況を確認できます。	false
ejb.collect.statistics=true|false	このプロパティは、 タイマー値をログに書き込まないことを除いて ejb.collect.display_statistics プロパティと同じです。	false
ejb.collect.display_detail_statistics=true|false	このフラグは、ejb.collect.display_statistics オプションと同じようにタイマー診断を有効にします。さらに、メソッドレベルのタイミング情報が出力されます。これにより、開発者は Bean のさまざまなメソッドによる CPU の使用状況を確認できます。このフラグをコンソールに出力する場合は、端末の画面を拡大して、長い行が折り返されないようにしてください。	false
ejb.mdb.threadMaxIdle=<num>	メッセージ駆動型 Bean を実行するために VM 全体で使用されるスレッドプールがあり、 EJB コンテナによって管理されます。 このプールは、 RMI 呼び出しを処理する ORB ディスパッチャプールと同じように設定できます。 この特別なプロパティは、 スレッドが解放されるまでアイドル状態にしておくことのできる最長時間を秒単位で制御します。	300
ejb.mdb.threadMax=<num>	MDB スレッドプールで許可される最大スレッド数。	0（制限なし）
ejb.mdb.threadMin=<num>	MDB スレッドプールで許可される最大スレッド数。	0
ejb.allowNullsInFinders=true|false	このプロパティは、 CMP 2.x にのみ適用できます。 このプロパティを true に設定すると、検索の戻り値または検索のコレクションの一部で NULL が許可されます。 デフォルトでは、このプロパティは False に設定されています。	False

EJB のカスタマイズプロパティ：デプロイメントデスクリプタレベル

これらのプロパティは、特定の EJB の動作をカスタマイズします。一部のプロパティは、特定タイプの EJB（セッションやエンティティなど）にだけ適用され、それ以外のプロパティはすべての種類の Bean に適用されます。これらのプロパティは、複数の場所で設定できます。設定できる場所を優先順位が高い方から示します。

1 JAR ファイルの ejb-borland.xml デプロイメントデスクリプタ内の EJB レベルで定義されたプロパティ要素。この設定は、この特定の EJB にだけ影響します。たとえば、次の XML は、`data` という EJB に対して `ejb.maxBeansInPool` プロパティを 99 に設定します。

```
<ejb-jar>
   . . .
   <enterprise-beans>
      <entity>
         <ejb-name>data</ejb-name>
         <bean-home-name>data</bean-home-name>
         <property>
            <prop-name>ejb.maxBeansInPool</prop-name>
            <prop-type>Integer</prop-type>
            <prop-value>99</prop-value>
         </property>
      </entity>
   </enterprise-beans>
   . . .
</ejb-jar>
```

2 JAR ファイルの ejb-borland.xml デプロイメントデスクリプタ内の <ejb-jar> レベルで定義されたプロパティ要素。この設定は、この JAR 内で定義されたすべての EJB に影響します。たとえば、次の XML は、特定の JAR ファイルのすべての EJB に対して `ejb.maxBeansInPool` プロパティを 99 に設定します。

```
<ejb-jar>
   . . .
   <property>
      <prop-name>ejb.maxBeansInPool</prop-name>
      <prop-type>Integer</prop-type>
      <prop-value>99</prop-value>
   </property>
   . . .
</ejb-jar>
```

3 EAR ファイルの `application-borland.xml` デプロイメントデスクリプタ内の `<application>` レベルで定義されたプロパティ要素。この設定は、この EAR ファイルに配置されたすべての JAR で定義されたすべての EJB に影響します。たとえば、次の XML は、EAR レベルで `ejb.maxBeansInPool` プロパティを 99 に設定します。

```
<application>
   . . .
   <property>
      <prop-name>ejb.maxBeansInPool</prop-name>
      <prop-type>Integer</prop-type>
      <prop-value>99</prop-value>
   </property>
   . . .
</application>
```

4 EJB コンテナレベルのプロパティとして定義された EJB プロパティ。これは、この EJB コンテナにデプロイメントされたすべての EJB に影響します。たとえば、次のコマンドは、スタンドアロンで起動された EJB コンテナにデプロイメントされたすべての Bean に対して、`ejb.maxBeansInPool` プロパティを 99 に設定します。

```
vbj -Dejb.maxBeansInPool=99 com.inprsie.ejb.Container ejbcontainer hello.ear
-jns -jss -jts
```

EJB プロパティの完全なインデックス

すべての種類の EJB に共通するプロパティ

プロパティ	型	説明	デフォルト値
ejb.default_transaction_attribute	Enumeration (NotSupported,Supports, Required,RequiresNew,Mandatory, Never)	このプロパティは、標準デプロイメントデスクリプタで trans-attribute が定義されていないメソッドのトランザクション属性値を指定します。このプロパティが指定されていない場合、EJB コンテナでデフォルトのトランザクション属性は設定されません。したがって、このプロパティを指定すると、デフォルトのトランザクション属性が設定されているほかの application server で作成された J2EE アプリケーションを簡単に移植できます。	なし

エンティティ Bean プロパティ（すべてのタイプのエンティティ－BMP、CMP 1.1、CMP 2－に適用）

プロパティ	型	説明	デフォルト値
ejb.maxBeansInPool	Integer	このオプションでは、準備完了プール内の最大の Bean 数を指定します。準備完了プールがこの制限を超えた場合は、unsetEntityContext が呼び出されて、エンティティがコンテナから削除されます。	1000
ejb.maxBeansInCache	Integer	このオプションは、トランザクションではなく、主キーに関連付けられた Bean を保持するキャッシュ内の Bean の最大数を指定します。これは、オプション「A」と「B」に関係します（下記の ejb.transactionCommitMode を参照）。キャッシュがこの制限を超えた場合は、ejbPassivate が呼び出されて、エンティティが準備完了プールに移されます。	1000
ejb.maxBeansInTransactions	Integer	1 つのトランザクションから、任意の数の多くのエンティティにアクセスできます。このプロパティにより、EJB コンテナが作成する物理的な Bean インスタンス数の上限を設定します。アクセスされるデータベースエンティティ／データベース行の数に関係なく、コンテナは、制限された数のエンティティオブジェクト（ディスパッチャ）でトランザクションを完了します。このデフォルトは、ejb.maxBeansInCache/2 という計算で求められます。ejb.maxBeansInCache プロパティが設定されていない場合は、500 になります。	Calculated

プロパティ	型	説明	デフォルト値
ejb.transactionCommitMode	Enumeration (A|Exclusive, B|Shared, C|None)	このフラグは、トランザクションの面から見たエンティティ Bean の特性を指定します。 次の値を指定できます。 A または Exclusive：このエンティティは、データベース内の特定のテーブルに排他的にアクセスします。 したがって、 最後にコミットされたトランザクションの Bean の状態を次のトランザクションの最初の Bean の状態とみなすことができます。 たとえば、トランザクションをまたがって Bean をキャッシュする場合です。 B または Shared：このエンティティは、 データベース内の特定のテーブルへのアクセスを共有します。ただし、パフォーマンス上の理由から、 ejbActivate と ejbPassivate をトランザクション間で無駄に呼び出すことのないように、特定の Bean はトランザクション間で特定の主キーに関連付けられたままになります。 これらの Bean はアクティブプールに残ります。この設定はデフォルトです。 C または None：このエンティティは、 データベース内の特定のテーブルまでのアクセスを共有します。トランザクション間で特定の主キーとの関連付けが解除され、トランザクションごとに準備完了プールに戻される Bean があります。これは一般に有効な設定ではありません。	Shared
ejb.transactionManagerInstanceName	String	このプロパティは、メソッドの呼び出しで起動されるトランザクションを制御するために、特定のトランザクションマネージャを名前で指定します。 このオプションを使用すると、 特定のトランザクションを 2PC で完了する必要があるときに、エンティティ Bean などのシステムでほかのすべてのトランザクションに 2PC トランザクションマネージャを使用する RPC のオーバーヘッドを避けることができます。 これは、 MDB でもサポートされています。	なし

プロパティ	型	説明	デフォルト値
ejb.findByPrimaryKeyBehavior	Enumeration (Verify, Load, None)	このフラグは、 findByPrimaryKey メソッドに必要な動作を示します。次の値を指定できます。 Verify : findByPrimaryKey の標準の動作です。 指定された主キーがデータベース内に存在するかどうかを簡単に検証します。 Load : この動作は、finder 呼び出しがアクティブなトランザクションで実行されているときに findByPrimaryKey が起動されると、Bean の状態をコンテナにロードします。 検出されるオブジェクトが一般的な形式で使用されていることと、 見つかったオブジェクトの状態をそのままロードしても問題がないことが前提です。 この設定はデフォルトです。 None : この動作は、findByPrimaryKey が no-op である必要があることを示します。基本的には、 これにより、 オブジェクトが実際に使用されるまで Bean の検証が遅延されます。 オブジェクトは常に、 find の呼び出しからオブジェクトが実際に使用されるまでの間に削除可能なので、 ほとんどのプログラムでは、この最適化によってクライアントのロジックを変更する必要はありません。	
ejb.checkExistenceBeforeCreate	Boolean	エンティティ Bean がマッピングされるほとんどのテーブルには、 主キー制約があります。 すでに存在する Bean を CMP エンジンが作成しようとすると、 この制約は無視され、 DuplicateKeyException が生成されます。 ただし、 一部のテーブルは主キー制約を定義しません。 このような場合、 checkExistanceBeforeCreate プロパティを使ってエンティティの重複を避けることができます。 True が設定されている場合、 CMP エンジンはデータベースをチェックし、 挿入操作を行う前にエンティティが存在するかどうかを確認します。 エンティティが存在する場合は、 DuplicateKeyException が生成されます。	False

メッセージ駆動型 Bean プロパティ

プロパティ	型	説明	デフォルト値
ejb.mdb.use_jms_threads	Boolean	onMessage() メソッドを実行するために、コンテナ管理スレッドのかわりに JMS プロバイダのディスパッチャスレッドを使用するように切り替えるオプション。 OpenJMS では、メッセージが JMS プロバイダのディスパッチャスレッドに配信されるので、この値は true になります。	false **メモ：** この値は、OpenJMS ではデフォルトで true です。
ejb.mdb.local_transaction_optimization	Boolean	このプロパティは現在 OpenJMS だけで使用し、XAConnectionFactory を使用せずに原子性を実現するために使用します。 メッセージの永続化とアプリケーションデータに対して、 同じデータベースを使用します。	true
ejb.mdb.maxMessagesPerServerSession	Integer	複数のメッセージを含む ServerSession を一括ロードするオプションをサポートする JMS プロバイダの場合、このプロパティを使ってパフォーマンスを調整します。	5
ejb.mdb.max-size	Integer	これは、 プール内の最大の接続数です。	なし
ejb.mdb.init-size	Integer	プールが最初に作成されたときに、AppServer がプールを満たすために使用する接続の数です。	なし
ejb.mdb.wait_timeout	Integer	maxPoolSize 接続が開かれているときに、接続が解放されるまで待つ時間を秒単位で指定します。 maxPoolSize プロパティを使用しており、プールがいっぱいで、これ以上接続を使用できない場合は、 JDBC 接続を検索するスレッドは、 待ち時間が無制限に設定されている（0 秒に設定）と、その接続が使用できるようになるまで待機します。 必要に応じて、waitTimeout 時間を設定できます。	30
ejb.mdb.rebindAttemptCount	Integer	これは、失敗した JMS 接続や MDB に対して確立できなかった接続について、 EJB コンテナが再確立を試行する回数です。 コンテナによる試行回数に上限を設定しない場合は、 `ejb.mdb.rebindAttemptCount=0` を明示的に指定する必要があります。	5
ejb.mdb.rebindAttemptInterval	Integer	失敗した JMS 接続や確立されなかった接続に関して再試行を実行するときの時間間隔の秒数(上記のプロパティを参照)	60
ejb.mdb.maxRedeliverAttemptCount	Integer	これは、 MDB が何らかの理由でメッセージの受信に失敗した場合に、JMS サービスプロバイダによって実行されるメッセージの再配信回数です。 メッセージは 5 回まで再配信されます。 5 回の試行の後、 メッセージはデッドキューに配信されます（設定されている場合)。	5

プロパティ	型	説明	デフォルト値
ejb.mdb.unDeliverableQueueConnectionFactory	String	MDB が何らかの理由でメッセージの受信に失敗した場合、メッセージは JMS サービスによって再配信されます。メッセージは 5 回まで再配信されます。5 回の試行の後、メッセージはデッドキューに配信されます(設定されている場合)。このプロパティは、JMS サービスの接続を作成するために接続ファクトリの JNDI 名を検索します。このプロパティは、ejb.mdb.unDeliverableQueue プロパティと連係して使用します。	なし
ejb.mdb.unDeliverableQueue	String	MDB が何らかの理由でメッセージの受信に失敗した場合、メッセージは JMS サービスによって再配信されます。メッセージは 5 回まで再配信されます。5 回の試行の後、メッセージはデッドキューに配信されます(設定されている場合)。このプロパティは、キューの JNDI 名を検索します。このプロパティは、ejb.mdb.unDeliverableQueueConnectionFactory プロパティと連係して使用します。	なし
ejb.transactionManagerInstanceName	String	このプロパティは、「必須」トランザクション属性を持つ MDB だけをサポートします。このプロパティは、onMessage() 呼び出しで起動されるトランザクションを制御するために、特定のトランザクションマネージャを名前で指定します。 このオプションを使用すると、この特定のトランザクションを 2PC で完了する必要があるときに、エンティティ Bean などのシステムでほかのすべてのトランザクションに 2PC トランザクションマネージャを使用する RPC のオーバーヘッドを避けることができます。 詳細については、 MDB の章を参照してください。	なし

ステートフルセッション Bean プロパティ

プロパティ	型	説明	デフォルト値
ejb.sfsb.passivation_timeout	Integer	非アクティブなステートフルセッション Bean を永続的ストレージ（JSS）に保存する時間間隔を秒単位で定義します。	5
ejb.sfsb.instance_max	Integer	EJB コンテナのメモリに同時に存在できる特定のステートフルセッション Bean の最大数を定義します。値が最大値に達した後にステートフルセッションの新しいインスタンスを割り振らなければならない状況になると、EJB コンテナからリソース不足を知らせる例外が生成されます。0 は特別な値です。これは最大値が設定されていないことを表します。このプロパティは、ejb.sfsb.passivation_timeout プロパティがゼロ以外の値に設定されている場合にだけ適用されます。	0
ejb.sfsb.instance_max_timeout	Integer	ejb.sfsb.instance_max プロパティで定義したステートフルセッションの最大値に達すると、EJB コンテナは、リソース不足を知らせる例外を生成する前に、新しい Bean の割り振りに対して、このプロパティで定義した時間だけ要求をブロックして、ステートフルセッションの値が減るのを待ちます。このプロパティは、ms（1/1000 秒）単位で設定します。0 は特別な値です。0 に設定すると待機時間が 0 となり、ただちにリソース不足を知らせる例外が生成されます。	0
ejb.jsec.doInstanceBasedAC	Boolean	true が設定されている場合、EJB コンテナは、EJB のメソッドを呼び出すプリンシパルがこの Bean を生成したプリンシパルと同じであるかどうかを確認します。この確認が失敗すると、メソッドは java.rmi.AccessException（または javax.ejb.AccessLocalException）例外を生成します。これはステートフルセッション Bean にだけ適用されます。	True

EJB セキュリティのプロパティ

プロパティ	型	説明	デフォルト値
ejb.security.transportType	Enumeration (CLEAR_ONLY, SECURE_ONLY, ALL)	このプロパティは、 特定の EJB の保護品質を設定します。 CLEAR_ONLY に設定すると、クライアントは、この EJB に対してセキュリティで保護されていない接続だけを受け付けます。 EJB にメソッド許可が割り当てられていない場合は、これがデフォルト設定になります。 SECURE_ONLY に設定すると、クライアントは、この EJB に対してセキュリティで保護された接続だけを受け付けます。 EJB に少なくとも 1 つのメソッド許可がある場合は、これがデフォルト設定になります。 ALL に設定すると、 クライアントは、 セキュリティで保護された接続とセキュリティで保護されていない接続の両方を受け付けます。 このプロパティの設定により、ServerQoPConfig ポリシーの転送値が制御されます。	なし
ejb.security.trustInClient	Boolean	このプロパティは、 特定の EJB の保護品質を設定します。 true に設定すると、EJB コンテナは、 クライアントに認証 ID を提供するように要求します。 メソッド許可が設定されていないメソッドが少なくとも 1 つ存在する場合は、このプロパティはデフォルトで false に設定されます。そうでない場合は、true に設定されます。このプロパティの設定により、ServerQoPConfig ポリシーの転送値が制御されます。	

Java セッションサービス（JSS）のプロパティ

JSS は、スタンドアロン EJB コンテナ（-jss オプション）の一部やパーティションの一部として実行できます。

JSS 設定情報は、「パーティションサービス」として、各パーティションのデータディレクトリの partition.xml ファイルにあります。デフォルトでは、このファイルは次のディレクトリにあります。

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>
/mos/<partition_name>/adm/properties/
```

たとえば、「standard」というパーティションの場合、JSS 設定情報はデフォルトで次の場所にあります。

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>
/mos/standard/adm/properties/partition.xml
```

詳細については、『*partition.xml リファレンス*』の 350 ページの「<services> 要素」を参照してください。

また、パーティションのデータディレクトリの場所については、次の場所にある configuration.xml ファイルを参照してください。

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/
```

そして、パーティション管理オブジェクトのディレクトリ属性を検索します。

```
<partition-process directory=
```

JSS は 2 種類のバックエンドストレージ（JDataStore または JDBC データソース）をサポートします。詳細については、51 ページの「Java セッションサービス（JSS）の設定」の「設定」を参照してください。

プロパティ	コンソールプロパティ名	説明	デフォルト値
jss.workingDir=<path>	Working directory	バックエンドデータベース（JDataStore）ファイルがあるディレクトリ。 **メモ：**このプロパティは、 jss.pstore プロパティで、 JDataStore ファイルをバックエンドストレージに使用する設定になっている場合にだけ適用されます。	値を設定しなかった場合、 JSS はパーティションで実行されます。 パーティションの作業ディレクトリ <install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/mos/<partition_name> が適用されます。 値を設定しなかった場合、 JSS はスタンドアロン EJB コンテナの一部として実行されます。 コンテナが開始した現在のディレクトリが適用されます。
jss.factoryName=<char_string>	Factory name	このサービスで作成した JSS ファクトリの名前です。 サービスはこの名前でスマートエージェント（osagent）に登録されます。	値を設定しなかった場合、 JSS はパーティションで実行されます。 デフォルト値は、 <server_name>:file:<install_dir>/ var/domains/<domain_name>/configurations/<configuration_name>/mos/<partition_name>/ です。 値を設定しなかった場合、 JSS はスタンドアロン EJB コンテナで実行されます。 デフォルト値は EJB/JSS[<container_name>] です。
jss.softCommit=true|false	Soft commit	true の場合、 JSS は、 ソフトコミットモードを有効にして JDataStore バックエンドデータベースを使用します。 このプロパティを設定すると、 セッションサービスのパフォーマンスが向上しますが、 最近コミットしたトランザクションはシステムクラッシュ後にロールバックされるおそれがあります。 **メモ：**このプロパティは、 jss.pstore プロパティで、 JDataStore ファイルをバックエンドストレージに使用する設定になっている場合にだけ適用されます。 詳細については、 http://info.borland.com/techpubs/jdatastore/ にある JDataStore ドキュメントを参照してください。	true
jss.maxIdle=<numeric value>	Max idle	JSS ガベージコレクションジョブの実行と実行の間の時間（秒）。 JSS ガベージコレクションジョブでは、 バックエンドデータベースから期限切れセッションの状態を削除します。 0 に設定すると、 ガベージコレクションジョブは開始しません。	1800 (=30min)

プロパティ	コンソールプロパティ名	説明	デフォルト値
jss.debug=true|false		デバッグ情報を出力します。 true に設定した場合、 JSS はデバッグトレースを出力します。	false
jss.pstore=<char_string>	Persistent store	バックエンドストレージに使用する JDatastore ファイルを指定します。 ファイルがない場合、 JSS は .jds 拡張子でファイルを作成します。 例、 jss_factory.jds。 JDBC をサポートする互換データベースでは、 JNDI 名に serial: プレフィクスを付けて指定します。 たとえば、 バックエンドストレージに使用する場合は、 serial://datasources/OracleDB と指定します。 この場合、JSS は、ネーミングサービスで、指定 JNDI 名でデプロイメントされたデータソースを使用します。	JSS がパーティションで実行する場合、 jss_factory.jds など、指定した JDataStore ファイルが使用されます。 JSS がスタンドアロン EJB コンテナで実行する場合、 <container_name>_jss.jds が使用されます。
jss.backingStoreType=<Dx|JDBC>		使用する永続性バックエンドストレージのタイプを指定します。 設定できる値は、 Dx（ローカルの JDatastore データベース）または JDBC（jss.pstore プロパティの値を使って提供される JNDI 名から解決される JDBC データソース名）です。	パーティションで実行される JSS のデフォルトは Dx（ローカルの JDataStore）です。 JSS がスタンドアロン EJB コンテナで実行する場合、 デフォルトは Dx です。
jss.userName=<char_string>	User name	JDataStore バックエンドデータベースとの接続時を JSS が開くときに使用するユーザー名。 **メモ：**このプロパティは、 jss.pstore プロパティで、 JDataStore ファイルをバックエンドストレージに使用する設定になっている場合にだけ適用されます。	<default-user-name>
jss.passWord=<char_string>		JSS 永続性ストレージが jss.backingStoretype=DX によって定義されている場合は、このプロパティを使ってローカルの JDataStore データベースへの jss.userName アクセスに必要なパスワードを設定します。 **メモ：**このプロパティは、 jss.backingStoretype=Dx（永続性バックエンドストレージに JDataStore を使用する設定）の場合にだけ有効です。	masterkey

パーティショントランザクションサービス（トランザクションマネージャ）

次に示すプロパティは、パーティショントランザクションサービス（トランザクションマネージャ）の動作に影響します。これらのプロパティは、スタンドアロン EJB コンテナとパーティションのいずれかがホストしていれば指定できます。

パーティションにパーティショントランザクションサービスを設定する場合、`<install_dir>/var/domains/base/configurations/<configuration_name>/mos/<partition_name>/adm/properties` に配置された `partition.xml` ファイルにプロパティを設定します。

EJB コンテナをスタンドアロンで実行している場合は、後述の「JTS システムプロパティ」で説明されたシステムプロパティ名を使って指定する必要があります。たとえば、JTS がスタンドアロン EJB コンテナによってホストされる場合、次のようにプロパティ `jts.allow_unrecoverable_completion` に相当するシステムプロパティを使用して、このプロパティを指定する必要があります。

```
prompt% vbj -DEJBAllowUnrecoverableCompletion com.inprise.ejb.Container
ejbcontainer beans.jar -jns -jts
```

プロパティ	説明	デフォルト値
jts.allow_unrecoverable_completion=true|false	`true` に設定すると、複数のリソースが登録されている場合に、回復不可能な（2 フェーズでない）操作を実行するようにコンテナの組み込み JTS インプリメンテーションに指示します。これは、 各自の責任で使用してください。開発者用の機能としてのみ提供されています。 OpenJMS では、このプロパティはデフォルトで true に設定されます。	False
jts.no_global_tids=true|false	デフォルトでは、 JTS は X/Open XA 互換のトランザクション ID を生成します。このプロパティを `true` に設定すると、トランザクションのキー生成動作が変更されて、 XA 準拠でないトランザクション ID を生成します。 EJB コンテナは、 デフォルトで XA 準拠のプロパティを生成することで、 JDBC2/XA ドライバとシームレスに機能します。	False
jts.no_local_tids=true|false	最適化の 1 つに、同じ VM 内に存在するトランザクションサービス内でトランザクションが開始されたことを EJB コンテナが検出し、そのトランザクションの比較を高速化するという方法があります。 このプロパティを `true` に設定すると、この機能は無効になります。 このローカルトランザクション ID（ローカルトランザクション ID）は、 グローバルトランザクション ID のサブセットなので、トランザクションの比較が高速化されます。	False
jts.timeout_enable=true|false	デフォルトでは、 JTS のトランザクションタイムアウト機能は無効です。 有効にすると、 JTS によって作成された新しい各トランザクションは、 JTS タイムアウトマネージャでタイムアウトになったものとして登録されます。トランザクションが完了する前にタイムアウトになると、 JTS は自動的にそのトランザクションをロールバックします。	False
jts.timeout_interval=<num>	JTS タイムアウトマネージャは、このプロパティ値で指定されている秒単位間隔で、タイムアウト期間が過ぎて登録されているトランザクションを調べます。`0` の値を設定すると、 9999 秒間隔になります。	5
jts.default_timeout=<num>	Bean 管理のトランザクションのタイムアウト時間は、 JTA `UserTransaction setTransactionTimeout()` メソッドを使って設定できます。 このメソッドを使用しない場合や、コンテナ管理のトランザクションでない場合は、 デフォルトのトランザクションのタイムアウト値が適用されます。このデフォルト値は、`jts.default_timeout` プロパティ値を使用して、 JTS の起動時に設定できます。 このプロパティの設定単位は `1` 秒です。	600
jts.default_max_timeout=<num>	`jts.default_timeout` プロパティのタイムアウト値が長くなりすぎるのを避けるために、 `jts.default_max_timeout` プロパティは、トランザクションがタイムアウトせずにアクティブな状態を保つことができる時間の上限を制御します。 このプロパティの設定単位は 1 秒です。	3600
jts.trace=true|false	このプロパティを設定すると、 JTS デバッグメッセージが生成されます。	False
jts.transaction_debug_timeout=<num>	設定すると、このプロパティは、 JTS によって管理されているアクティブなトランザクションのリストを表示します。 この値は、トランザクションが表示される間隔を秒単位で示します。	None

第 39 章

AppServer 6.7 での LifeRay Portal の使用

ここでは、Liferay EAR のデプロイメントの準備、Borland AppServer（AppServer）コンソールを使用した Liferay 設定の作成、LifeRay ポータルのデプロイメント、およびカスタムポートレットのデプロイメントの手順について説明します。

Liferay は、ポートレットをデプロイメントするために作成されたオープンソースのポータルです。このポータルでは、パーソナル化、ユーザー／グループ管理、Web メール、掲示板、およびコンテンツ管理のすべてが 1 つにパッケージされて提供されています。これは、Java Portlet Specification JSR-168 に準拠した数多くのポートレットアプリケーションにバンドルされてデプロイメントされています。

LifeRay Portal を AppServer で使用するには、次の手順にしたがいます。

1. http://www.liferay.com から、LifeRay 4.0 EAR ファイルをダウンロードします。
2. Borland 管理コンソールを開いてログインします。
3. LifeRay Portal の設定を作成します。管理コンソールを使用して設定を作成すると、LifeRay パーティション、JDataStore、JMS があらかじめ設定されています。
   a. 左側のペインで［Configurations］ノードを右クリックし、メニューから［Add Configuration...］を選択します。
   b. ［Template Gallery］の［Portals］をクリックします。
   c. ［Template Gallery］の右側のペインから［LifeRay Portal Configuration version 4.0］を選択し、［Select］ボタンをクリックします。［Create New Configuration］ダイアログボックスが表示されます。
   d. ［Name］フィールドに、新しい LifeRay 設定の名前を入力します。
   e. ［Value］列の値をダブルクリックして、［Configuration Properties］ボックスのスマートエージェントポートを変更します。
   f. ［OK］をクリックします。
4. 設定名を右クリックし、表示されるメニューから［Start］を選択して、設定を実行します。

5 LifeRay EAR ファイルをホストする LifeRay サーバーを作成します。EAR ファイルをサーバーにデプロイメントします。

LifeRay サーバーを作成するには

a Borland 管理コンソールの左側のペインにある LifeRay パーティションの下の［Hosted Modules］を右クリックします。

b メニューから［Host LifeRay module...］を選択します。［Host LifeRay Portal］ダイアログが開きます。

c ［Liferay Ear Path］ボックスに、LifeRay EAR ファイルのパスを入力します。

d ［Host Target Directory］フィールドに、LifeRay モジュールをホストするディレクトリのパスを入力します。このディレクトリは、エージェントと同じマシン上に存在する必要があります。

e ［Generate Stub］チェックボックスが選択されていることを確認してください。

f デフォルトのモジュール名を変更する場合は、［Module Name］テキストボックスに名前を入力します。デフォルトでは、モジュール名は LifeRay モジュールをホストするディレクトリと同じ名前になります。

g ［OK］をクリックします。状態ボックスが表示されます。AppServer は、まずスタブを生成してから、ステップ 3 で入力したディレクトリに EAR ファイルの内容を抽出します。

6 ブラウザで `http://localhost:8080` を開き、LifeRay モジュールが正常にデプロイメントされていることを確認します。ブラウザに LifeRay ポータルが表示されるはずです。LifeRay Portal のデフォルトのログイン名は「test@liferay.com」、パスワードは「test」です。

メモ LifeRay パーティションのプロパティを変更するには、管理コンソールの左側のペインで LifeRay パーティション名を右クリックし、メニューから［Properties］を選択します。必要なタブをクリックして前面に表示し、そのタブに関連付けられているプロパティを変更します。

他のデータベースの使用

デフォルトでは、LifeRay は JDataStore データベースを使用してデータを保存します。JDataStore 以外のデータベースも使用できます。サポートされているデータベースについては、Web サイト http://www.liferay.com/web/guest/documentation/development/databases を参照してください。JDataStore 以外のデータベースを使用する場合は、次の手順にしたがいます。

1 `jndi-definitions.xml` ファイルで使用するデータベースの JNDI 情報に基づいて、新しい `liferay.dar` ファイルを作成します。

使用するデータベースに合わせて `jndi-definitions.xml` ファイルを編集する方法については、Web サイト http://www.liferay.com/web/guest/documentation/development/databases を参照してください。

DAR ファイルの作成方法については、『*Borland AppServer ユーザーズガイド*』の「JNDI 定義デスクリプタアーカイブ（DAR）の作成」を参照してください。

2 作成した新しいファイルで、LifeRay Portal の設定に含まれるデフォルトの `liferay.dar` を置き換えます。

ポートレットまたは J2EE モジュールの LifeRay モジュールへのデプロイメント

ホストされている Liferay モジュールに、EJB JAR、WAR、RAR、またはライブラリ JAR を追加できます。これらのいずれかを LifeRay モジュールにデプロイメントするには

1. Borland 管理コンソールを開きます。
2. 左側のペインの LifeRay のパーティションノードを展開します。
3. ［Hosted Modules］の下の LifeRay がホストするモジュールを右クリックし、メニューから［Deploy Portlet］を選択します。LifeRay Portal デプロイメントウィザードが開きます。

   Liferay 3.6 をお使いの場合は、手順 4 に進んでください。
4. ［Add］ボタンをクリックして、デプロイメントするポートレット（WAR ファイル）をウィザードがポイントするようにします。
5. ［Finish］ボタンをクリックします。

ポートレットが正常にデプロイメントされたかどうかを確認するには

1. Web ブラウザを開きます。
2. Web ブラウザに「http://localhost:8080」と入力して LifeRay のポータルを開きます。
3. ポータルに、デフォルトログインの「test@liferay.com」とパスワードの「test」を使用してログインします。
4. Liferay 4.0 をお使いの場合、[Add Content] をクリックしてください。

   Liferay 3.6 をお使いの場合は、スクロールダウンして［Add Portlet to Wide Column］フィールドを表示します。このフィールドのドロップダウンメニューには、新たに追加されたポートレットが表示されています。
5. そのポートレットを選択し、［Add］ボタンをクリックしてポータルにポートレットを追加します。

第 40 章

Borland AppServer 6.7 の JBuilder 2006 との統合

この章では、JBuilder 2006 用 Borland AppServer 6.7 プラグインのインストール方法、プラグインの設定方法、および JBuilder 2006 に組み込んだ Borland AppServer 6.7 の使用方法を説明します。

Borland AppServer 6.7 用 JBuilder 2006 の設定

プラグインをインストールし、JBuilder を再起動したら、プラグインを使用できるように JBuilder を設定する必要があります。

Borland AppServer 6.7 用に JBuilder を設定するには

1. ［Enterprise|Configure Servers］を選択して［Configure Servers］ダイアログボックスを表示します。
   ダイアログボックスの右側にサーバーのデフォルト設定が表示されます。［General］ページには共通のフィールドが表示され、［Custom］ページにはサーバー固有のフィールドが表示されます。［Custom］設定を変更すると［General］ページの設定が更新される場合があります。
2. 左側のペインの User Home フォルダから Borland Enterprise Server AppServer Edition 6.x を選択します。

メモ　6.x は Borland Enterprise Server AppServer Edition 6.0RP1、Borland Enterprise Server AppServer Edition 6.5（パッチ 11）、および Borland AppServer 6.7 の AppServer バージョンを示します。

3. ダイアログボックス最上部の［Enable Server］オプションを選択します。
   このオプションを選択すると、Borland AppServer 6.7 用のフィールドが有効になります。このオプションを選択するまで、どのフィールドも編集できません。［Enable Server］チェックボックスにより、［Project ｜ Project Properties ｜ Server］を使用してプロジェクトのサーバーを選択したときに、このサーバーをサーバーのリストに表示するかどうかも決定されます。
4. ［General］タブの以下のフィールドを表示し、必要に応じて変更します。
   - ［Home Directory］: Borland AppServer 6.7 がインストールされるディレクトリ。デフォルトは `Borland/AppServer` です。デフォルトディレクトリが正しくない場合は、省略符［...］ボタンを使用して正しいディレクトリを参照します。
   - ［Native Executable Launcher］: このサーバーを実行するネイティブな実行可能ファイル。デフォルトでは、`<APPSERVER_HOME>/bin` フォルダの partition.exe です。

JBuilder がネイティブな実行可能ファイルを検出すると、このフィールドは自動的に入力されます。

- ［VM Parameters］：仮想マシンに渡すパラメータ。
- ［Server Parameters］：サーバーに渡すパラメータ。
- ［Working Directory］：作業ディレクトリの名前と場所。

5 ［Custom］タブをクリックしてサーバー固有のフィールドを表示し、必要に応じて変更します。次の各フィールドを変更または入力します。

- ［JDK Installation Directory］：JDK v 1.5.0 のあるディレクトリ。Borland AppServer 6.7 では、このフィールドは自動的に `<APPSERVER_HOME>/jdk/jdk1.5.0` フォルダに設定されます。プロジェクトでは、この JDK を使って AppServer パーティションが実行されます。
- ［Server Name］：Borland AppServer 6.7 のハブ名。
- ［Configuration Name］：パーティションを管理する設定の名前。デフォルトでは jbuilder になっています。
- ［Partition Name］：モジュールを実行するパーティションの名前。デフォルトでは jbpartition になっています。
- ［Add A Management Agent Item To The Enterprise Menu］：［Management Agent］項目を JBuilder Enterprise のメニューに追加して、Management Agent を素早く JBuilder IDE から起動できるようにします。
- ［Server Realm］：サーバー領域の名前。詳細は、『*Borland 管理コンソールユーザーズガイド*』を参照してください。デフォルトの設定値は ServerRealm です。
- ［User Name］：サーバーがユーザーを識別するために使用する名前。デフォルト値は、admin です。
- ［User Password］：サーバーがユーザーを識別するために使用するパスワード。デフォルト値は、admin です。
- ［Advanced Settings］：このボタンをクリックすると［Advanced Settings］ダイアログボックスが表示されます。このダイアログボックスを使用して、Management Agent が使用するポート番号の変更や［Use Security］オプションの選択を行います。管理ポートは、JBuilder で起動時やデプロイメント時にサーバーを検出するために使用されます。管理ポートは、デフォルトと異なる管理ポートを使用してリモートサーバーにデプロイメントする場合にのみ変更します。ポート番号を変更する場合は、必ずサーバーと同じポート番号を入力してください。ポート番号が正しくないとサーバーは起動しません。選択するポートとセキュリティは、サーバーの設定と一致する必要があります。値は Borland AppServer のプロパティファイルから読み取られますが、Home Directory の設定を変更すると、これらの値は自動的に変更されます。ただし、Management Agent の動作中に JBuilder でポート番号を変更すると、Management Agent は自動的にシャットダウンされます。

6 ［OK］をクリックしてダイアログボックスを閉じ、設定を保存します。
Borland Enterprise Server AppServer Edition 6.x を対象にしていたすべてのプロジェクトは、新しい Borland AppServer ホームを対象とするように自動的に更新されます。

JBuilder での Borland 管理コンソールの表示

Borland AppServer 6.7 用 JBuilder 2006 をインストールして設定すると、Borland AppServer 管理コンソールが JBuilder メッセージペインに表示されます。それには、`jbuilder.config` 設定ファイルを編集する必要があります。

1 プロジェクトを保存し、JBuilder を終了します。

2 テキストエディタで `jbuilder.config` を開きます。このファイルは、`<JBUILDER_HOME>/bin` フォルダにあります。

3 この設定ファイルに、次の VM パラメータを追加します。
```
vmparam -Djava.endorsed.dirs=<APPSERVER_HOME>/lib/endorsed
```

4 JBuilder を再起動します。

5 ［View|Panes|BAS Console 6.7］を選択します。
Borland 管理コンソールがメッセージペインに表示されます。

JBuilder を使った VisiBroker 開発

［Enterprise Setup］ダイアログボックス（［Enterprise|Enterprise Setup］）の CORBA ノードを使用して、Borland AppServer で VisiBroker 7.0 を使用するようにセットアップします。

JBuilder で ORB を使用できるようにするには

1 ［Enterprise|Enterprise Setup］を選択して［Enterprise Setup］ダイアログボックスを表示します。CORBA ページを選択します。このダイアログボックスのパラメータを使用すると、JBuilder で CORBA アプリケーションを開発できます。

2 ［ Configuration］ドロップダウンリストから、［ VisiBroker (Borland Enterprise Server AppServer Edition 6.x)］オプションを選択します。このオプションは、`<APPSERVER_HOME>/bin` フォルダを指すように自動的に更新されます。

3 ［Tools］メニューからスマートエージェントを起動するには、［Add The VisiBroker Smart Agent Item To The Tools Menu］オプションを選択します。

4 ESmartAgent ポートの番号を［SmartAgent Port］フィールドに入力します。

5 ［OK］をクリックして設定を保存します。

重要 CORBA アプリケーションの実行時設定では、［Edit Runtime Configuration］ダイアログボックス（［Run|Configurations|Edit］）の［VM Parameters］フィールドに次のパラメータを追加する必要があります。

```
-Dvbroker.agent.port=<your_osagent_port>
-Dborland.enterprise.licenseDir=<APPSERVER_HOME>/var
-Dborland.enterprise.licenseDefaultDir=<APPSERVER_HOME>/license
```

システムのセットアップが完了し、Borland AppServer 6.7 とともにインストールした VisiBroker 7.0 を使用できるようになりました。アプリケーションの実行前に、［Tools| VisiBroker Smart Agent］を選択してスマートエージェントを起動します。

JBuilder デプロイメントデスクリプタエディタを使用した J2EE 1.4 アプリケーションの開発

Borland AppServer 6.7 は、J2EE 1.4 アプリケーションの開発に対応しています。JBuilder デプロイメントデスクリプタエディタには、Borland AppServer 6.7 を対象にした J2EE 1.4 アプリケーション用に、新しいページが用意されています。

J2EE 1.4 では、各デプロイメントデスクリプタが DTD ではなく XML スキームに対して有効であることが必要です。JBuilder 2006 用 Borland AppServer 6.7 プラグインは、この変更に対応しています。プロジェクトのペインで J2EE デプロイメントデスクリプタを右クリックし、［Validate］を選択すると、そのデスクリプタは XML スキームファイルに対して有効になります。Borland AppServer デプロイメント用の J2EE モジュールは、次のスキームファイルに対して有効になります。

- アプリケーションモジュール：`application_1_4-borland.xsd`
- アプリケーションクライアントモジュール：`application-client_1_4-borland.xsd`
- コネクタモジュール：`connector_1_5.xsd`
- EJB モジュール：`ejb-jar_2_1-borland.xsd`
- Web モジュール：`web-app_2_4-borland.xsd`

標準 Java コメント付きの J2EE 1.4 XML スキームは、Java 2 Platform、Enterprise Edition (J2EE) : XML Schemas for J2EE Deployment Descriptors（`http://java.sun.com/xml/ns/j2ee/`）で参照できます。

Borland AppServer 6.7 の更新に対応して、JBuilder デプロイメントデスクリプタエディタには次のエンティティのページが更新または新規に作成されています。

- ［Message Destinations］ページ：Web モジュール、EJB モジュール、アプリケーションクライアントモジュール
- ［Message Destination Reference］ページ：Web モジュール、アプリケーションクライアントモジュール、セッション Bean、エンティティ Bean、メッセージ駆動型 Bean
- ［Message-Driven Bean］ページ：メッセージ駆動型 Bean
- ［Resource Environment References］ページ：メッセージ駆動型 Bean
- ［Admin Object and Admin Object Properties］ページ：メッセージ駆動型 Bean
- ［Resource Adapter］ページ：コネクタモジュール
- ［BES Connection Definition］ページ：コネクタモジュール

メモ JBuilder には、標準およびサーバー固有の J2EE 1.3 モジュールとデプロイメントデスクリプタ用に、デプロイメントデスクリプタエディタが用意されています。このエディタについては、JBuilder オンラインヘルプの *「Developing Applications with Enterprise JavaBeans」* にある「Editing EJB deployment descriptors」を参照してください。

［Message Destinations］ページ

［Messages Destinations］ページは、Web モジュール、EJB モジュール、およびアプリケーションクライアントモジュール用の新しい DD Editor ページです。

［Message Destinations］ページでは、Web モジュール、EJB モジュール、またはアプリケーションクライアントモジュール用の新しい J2EE 1.4 デプロイメントデスクリプタ要素 `<message-destination-name>` を設定します。この要素は、メッセージ送信先リファレンスの名前を指定します。Borland AppServer 6.7 の値は、Web モジュール、EJB モジュール、またはアプリケーションクライアントモジュールで使用されるメッセージ送信先リファレンスの名前を表す JNDI 名です。

［Message Destinations］ページを表示して標準および Borland AppServer 固有のデプロイメントデスクリプタ要素を設定するには

1. プロジェクトペインで Web モジュール、EJB モジュール、またはアプリケーションクライアントモジュールを選択します。
   内容ペインの下部にある［Select the DD Editor］タブを選択します。
2. 構造ペインを開きます。
3. モジュールノードを展開して、［Message Destinations］ノードを選択します。
4. メッセージ送信先を追加するには、ノードを右クリックし、［Add］を選択します。
5. DD Editor の［Standard］タブをクリックします。
   a. ［Name］フィールドに、メッセージ送信先の名前を入力します。デプロイメントファイル内のメッセージ送信先名の名前は、互いに重複しないようにする必要があります。
   b. ［Language］フィールドに、［Display Name］、［Description］、およびアイコンに関連付ける言語を入力します。言語ごとに 1 つの表示名、説明、および大小のアイコンを指定できます。［Add］と［Remove］ボタンを使用して言語を追加および削除します。
   c. ［Display Name］フィールドに、表示する名前を入力します。
   d. ［Description］フィールドに、説明を入力します。
   e. ［Large Icon］フィールドに、大きいアイコン（32 x 32 ピクセル）の場所を入力します。アイコンは、モジュールツリー内に存在する必要があります。
   f. ［Small Icon］フィールドに、小さいアイコン（16 x 16 ピクセル）の場所を入力します。アイコンは、モジュールツリー内に存在する必要があります。
6. DD Editor の［BES］タブをクリックします。［JNDI Name］フィールドに、メッセージ送信先の JNDI 名を入力します。詳細は、『*Borland AppServer 開発者ガイド*』の 199 ページの「JMS の使い方」を参照してください。

［Message Destination Reference］ページ

［Message Destinations Reference］ページは、Web モジュールとアプリケーションクライアントモジュール、またエンティティ Bean、セッション Bean、およびメッセージ駆動型 Bean 用の新しい DD Editor ページです。

［Message Destination Reference］ページでは、Web モジュールとアプリケーションクライアントモジュール、およびエンティティ Bean、セッション Bean、メッセージ駆動型 Bean 用の新しい J2EE 1.4 デプロイメントデスクリプタ要素 `<message-destination-ref>` を設定します。メッセージ送信先リファレンスでは、リソースに関連付けられたリファレンスが宣言されます。

［Message Destination Reference］ページを表示して Borland AppServer 6.7 固有のデプロイメントデスクリプタ要素を設定するには

1 プロジェクトペインで Web モジュール、EJB モジュール、またはアプリケーションクライアントモジュールを選択します。
内容ペインの下部にある［Select the DD Editor］タブを選択します。

2 構造ペインを開きます。

3 モジュールまたは Bean ノードを展開して、［Message Destination References］ノードを選択します。

4 メッセージ送信先リファレンスを追加するには、ノードを右クリックし、［Add］を選択します。

5 DD Editor の［Standard］タブをクリックします。以下の属性を設定します。

   a ［Name］フィールドに、メッセージ送信先リファレンスの名前を入力します。デプロイメントコンポーネントコードでは、この名前が使用されます。

   b ［Type］フィールドに、メッセージ送信先の Java のタイプを指定します。このタイプにより、送信先で実装される Java インターフェースが指定されます。

   c ［Usage］フィールドで、メッセージ送信先の使用方法を選択します。メッセージがメッセージ送信先で使用される場合は［Consumes］、メッセージが送信先用に生成される場合は［Produces］、メッセージが使用も生成もされる場合は［ConsumeProduces］を選択します。アセンブラは、この情報を使用して送信先のプロデューサとコンシューマをリンクします。

   d ［Link］フィールドに、メッセージ送信先リンクを指定します。この要素は、メッセージ送信先リファレンスまたはメッセージ駆動型 Bean をメッセージ送信先にリンクします。アセンブラは、アプリケーションでのプロデューサとコンシューマ間のメッセージフローを反映するように値を設定します。この値は、同じデプロイメントファイル、または同じ J2EE アプリケーションユニット内の別のデプロイメントファイルのメッセージ送信先の名前である必要があります。または、参照するメッセージ送信先を含むデプロイメントファイルのパス名に、送信先の名前を # でパス名と区切って追加して、値を組み立てます。このパス名は、メッセージ送信先を参照するデプロイメントコンポーネントを含むデプロイメントファイルを基準とする相対名です。これにより、複数のメッセージ送信先の名前が同じ場合でも、それぞれを区別できます。

   e ［Description Language］フィールドに、説明に使用する言語を入力します。［Add］と［Remove］ボタンを使用して言語を追加および削除します。言語別に 1 つの説明を加えることができます。

   f ［Description］フィールドに、メッセージ送信先リファレンスの説明を入力します。

6 DD Editor の［BES］タブをクリックします。

7 メッセージ送信先の JNDI 名を入力します。詳細は、『*Borland AppServer 開発者ガイド*』の 199 ページの「JMS の使い方」を参照してください。

[Message-Driven Bean] ページ

Borland 固有の［Message-Driven Bean］ページは、メッセージ駆動型 Bean に対応して更新されました。

DD Editor の［Message-Driven Bean］ページでは、メッセージ駆動型 Bean のデプロイメントデスクリプタを設定します。標準と Borland AppServer 6.7 固有のどちらのページも、J2EE 1.4 実装に対応して更新されています。

［Message-Driven Bean］ページを表示して標準および BAS 固有のデプロイメントデスクリプタ要素を設定するには

1 プロジェクトペインで EJB モジュールを選択します。
DD Editor がコンテンツペインに表示されます。

2 構造ペインを開きます。

3 EJB モジュールを展開し、［Message-Driven Bean］ノードを展開します。メッセージ駆動型 Bean を選択します。
DD Editor に、［Message-Driven Bean］ページが表示されます。

4 DD Editor の［Standard］タブをクリックします。以下の属性を設定します。

   a ［Name］フィールドに、メッセージ駆動型 Bean の名前を入力します。

   b ［EJB Class］フィールドに、Bean のビジネスメソッドを実装する Java クラスの完全な名前を入力します。この情報は必須です。

   c ［Messaging Type］フィールドで、Bean のメッセージングタイプを選択します。

   d ［Transaction Type］ドロップダウンリストから、Bean のトランザクションを管理する方法を選択します。トランザクションは、Bean 自体またはコンテナによって管理できます。

   e ［Message Destination Type］ドロップダウンリストから、メッセージ送信先のタイプを選択します。これは、実際のトピックか、メッセージ駆動型 Bean が監視するキューです。

   f ［Message Destination Link］フィールドに、Bean のメッセージ送信先を入力します。

   g ［Description Language］フィールドに、アクティベーション設定の説明に使用する言語を入力します。［Add］と［Remove］ボタンを使用して言語を追加および削除します。言語別に 1 つの説明を加えることができます。

   h ［Description］フィールドに、Bean の説明を入力します。

   i ［Language］フィールドに、表示情報の言語を入力します。［Add］と［Remove］ボタンを使用して言語を追加および削除します。言語別に 1 つの表示説明を加えることができます。

   j ［Display Name］フィールドに、表示目的の Bean を識別するために使用する名前を入力します。

   k ［Description］フィールドに、表示目的の説明を入力します。

   l ［Large Icon］フィールドに、Bean に関連付ける大きいアイコン（32 x 32 ピクセル）の名前を入力します。

   m ［Small Icon］フィールドに、Bean に関連付ける小さいアイコン（16 x 16 ピクセル）の名前を入力します。

5 DD Editor の［BES］タブをクリックします。このページでは、Borland AppServer `<message-source>` 要素を設定します。

   a ［Message Source Type］ドロップダウンリストから、メッセージソースのタイプを選択します。jms-provider-ref を選択して、EJB 2.0 実装を使用してメッセージソースを JMS プロバイダ経由でアクティブ化します。adapter-ref を選択して、メッセージソースを JCA 1.5 リソースアダプタ経由でアクティブ化します。

   b ［Destination Name］フィールドに、メッセージ駆動型 Bean の送信先を入力します。これは、実際のトピックか、メッセージ駆動型 Bean が監視するキューです。このフィールドは、メッセージソースタイプとして jms_provider_ref が選択されている場合に使用できます。

c ［Connection Factory Name］フィールドに、JMS ブローカーに接続するために使用するリソース接続ファクトリを入力します。このフィールドは、メッセージソースタイプとして jms_provider_ref が選択されている場合に使用できます。

d ［Initial Pool Size］フィールドに、初期接続数を入力します。このフィールドは、メッセージソースタイプとして jms_provider_ref が選択されている場合に使用できます。

e ［Maximum Pool Size］フィールドに、最大接続数を入力します。このフィールドは、メッセージソースタイプとして jms_provider_ref が選択されている場合に使用できます。

f ［Wait Timeout］フィールドに、接続するまでの待機時間を入力します（秒単位）。このフィールドは、メッセージソースタイプとして jms_provider_ref が選択されている場合に使用できます。

g ［Instance Name］フィールドに、J2EE リソースに接続するリソースアダプタインスタンスの名前を入力します。このフィールドは、メッセージソースタイプとして resource_adapter_ref が選択されている場合に使用できます。

［Resource Environment References］ページ

Borland 固有の［Resource Environment References］ページは、メッセージ駆動型 Bean に対応して更新されました。

［Resource Environment References］ページでは、メッセージ駆動型 Bean 用の `<resource-environment-ref>` 要素を設定します。リソース環境リファレンスは、JNDI 名または管理オブジェクトのいずれかに設定できます。リソース環境リファレンスは、クライアントアプリケーションが使用する論理名をオブジェクトの物理的な名前にマップします。

［Resource Environment Reference］ページを表示して Borland AppServer 6.7 固有のデプロイメントデスクリプタ要素を設定するには

1 プロジェクトペインで EJB モジュールを選択します。
DD Editor がコンテンツペインに表示されます。

2 構造ペインを開きます。

3 EJB モジュールを展開し、［Message-Driven Bean］ノードを選択します。メッセージ駆動型 Bean を選択します。

4 ［Resource Environment References］ページを右クリックし、［Add］を選択します。

5 DD Editor の［BES］タブをクリックします。

a ［Resource Environment References Type］ドロップダウンリストから、リファレンスのタイプを選択します。JNDI リファレンスを選択する場合は、［JNDI name］を選択します。管理オブジェクトを選択する場合は、［Admin Object］を選択します。管理オブジェクトを選択した場合は、このオブジェクトのプロパティを設定する必要があります。詳細は、381 ページの「［Admin Object and Admin Object Properties］ページ」を参照してください。

b ［JNDI Name］フィールドに、論理名をオブジェクト名にマップする JNDI Bean の名前を入力します。このフィールドは、リソース環境タイプとして JNDI 名が選択されている場合にのみ使用できます。詳細は、『*Borland AppServer 開発者ガイド*』の 199 ページの「JMS の使い方」を参照してください。

［Admin Object and Admin Object Properties］ページ

［Admin Object］ページと［Admin Object Properties］ページは、メッセージ駆動型 Bean のリソース環境用の新しいページです。

［Admin Object］ページでは、リソース環境リファレンスの管理オブジェクトを追加します。［Admin Object Properties］ページでは、オブジェクトのプロパティを設定します。このページは、［Resource Environment References Type］で［Admin Object］を選択した場合にのみ使用できます。管理オブジェクトは、メッセージングスタイルまたはメッセージングプロバイダだけに存在します。

［Admin Object］ページを表示して Borland AppServer 6.7 固有のデプロイメントデスクリプタ要素を設定するには

1. プロジェクトペインで EJB モジュールを選択します。
   DD Editor がコンテンツペインに表示されます。
2. 構造ペインを開きます。
3. EJB モジュールを展開し、［Message-Driven Bean］ノードを選択します。メッセージ駆動型 Bean を選択します。
4. ［Resource Environment References］ページを右クリックし、［Add］を選択します。
5. DD Editor の［BES］タブをクリックします。
6. ［Resource Environment Reference Type］ドロップダウンリストから、［Admin Object］を選択します。
7. 追加したエントリが表示されるまで、構造ペインの［ Resource Environment References］ノードを展開します。
8. ノードを展開し、［Admin Object Properties］ノードを選択します。ノードを右クリックし、［Add］を選択します。
   DD Editor に、［BES Admin Object Properties］ページが表示されます。
9. 次のようにプロパティを入力します。
   a. ［Name］フィールドに、プロパティ名を入力します。
   b. ［Type］ドロップダウンリストから、プロパティのタイプを選択します。［java.lang.String］、［java.lang.Boolean］、［Integer］のいずれかを選択します。［<Unspecified>］を選択することもできます。
   c. ［Value］フィールドに、プロパティの値を入力します。値は、プロパティのタイプに適合している必要があります。

[Resource Adapter] ページ

Borland 固有の［Resource Adapter］ページは、コネクタモジュール用の新しいページです。

［Resource Adapter］ページでは、コネクタモジュールに対して、Borland 固有 JCA 1.5 デプロイメントデスクリプタ `<resourceadapter>` 要素を設定します。この要素は、コネクタのリソースアダプタを記述します。

［Resource Adapter］ページを表示して Borland AppServer 6.7 固有のデプロイメントデスクリプタ要素を設定するには

1 プロジェクトペインでコネクタモジュールを選択します。
DD Editor がコンテンツペインに表示されます。

2 構造ペインを開きます。

3 コネクタモジュールを展開し、［Resource Adapter］ノードを選択します。

4 DD Editor の［BES］タブをクリックします。

   a ［Instance Name］フィールドに、接続ファクトリの名前を入力します。

   b ［Resource Adapter Link Reference］フィールドに、リソースアダプタリンクのリファレンスを入力します。これにより、複数のデプロイメントリソースアダプタを 1 つのデプロイメントリソースアダプタを関連付けることができます。このリンクにより、基本のリソースアダプタですでに設定されているリソースを別のリソースアダプタにリンクして再利用したり、属性の一部だけを変更することができます。このフィールドを使用すると、可能であればリソースの重複を避けることができます。基本のリソースアダプタのデプロイメントに定義された値は、他の値が指定されない限り、すべてリンク先のリソースアダプタに継承されます。

   c ［Resource Adapter Library Directory］フィールドに、すべての共有ライブラリのコピー先のディレクトリを入力します。

   d ［Authorization Domain］フィールドに、接続の承認ドメインを入力します。

[BES Connection Definition] ページ

BES 接続定義のページは、コネクションモジュール用の新しいページです。

［BES Connection Definition］ページでは、Borland コネクタモジュールに対して、Borland 固有 JCA 1.5 デプロイメントデスクリプタ `<outbound-resourceadapter>` 要素を設定します。設定する情報には、コネクタアーキテクチャの一部として必要なクラスとインターフェースの完全修飾名、管理接続の数、接続の時間間隔が含まれます。

［BES Connection Definition］ページを表示して Borland AppServer 6.7 固有のデプロイメントデスクリプタ要素を設定するには

1 プロジェクトペインでコネクタモジュールを選択します。
DD Editor がコンテンツペインに表示されます。

2 構造ペインを開きます。

3 コネクタモジュールと［Resource Adapter］ノードを展開します。

4 ［BES Connection Definitions］ノードを右クリックし、［Add］を選択します。

5 接続定義の属性を次のように設定します。

   a ［Factory Interface］フィールドに、ファクトリインターフェースの名前を入力します。

   b ［Factory Name］フィールドに、JMS ブローカーに接続するために使用するファクトリクラスの名前を入力します。

   c ［Description］フィールドに、接続の説明を入力します。

   d ［JNDI 名］フィールドに、接続ファクトリへの JNDI クラスの名前を入力します。詳細は、『*Borland AppServer 開発者ガイド*』の 199 ページの「JMS の使い方」を参照してください。

   e ManagedConnectionFactory または ManagedConnection クラスのログを記録するために、［Enable Logging］オプションをチェックします。

f ［Log File Name］フィールドに、ログの結果を書き込むファイルの名前と場所を入力します。

g ［Initial Capacity］フィールドに、デプロイメント時にサーバーが割り当てを試みる管理接続数の初期値を入力します。

h ［Maximum Capacity］フィールドに、サーバーが同時に割り当てを認める最大管理接続数を入力します。

i ［Busy Timeout］フィールドに、接続がビジーの場合に待機する時間を秒単位で入力します。

j ［Idle Timeout］フィールドに、接続がタイムアウトするまでの待機時間を秒単位で入力します。

k ［Wait Timeout］フィールドに、接続までの待機時間を秒単位で入力します。

l ［Capacity Delta］フィールドに、新しい接続を求める要求に応答するときにサーバーが割り当てを試みる管理接続数を入力します。

m システムリソースを節約するために、［Enable Cleanup］オプションを選択して、使用されていない管理接続の回収をサーバーが試みるようにします。

n ［Cleanup Interval］フィールドに、使用されている管理接続の回収を試みる時間間隔を秒単位で入力します。

6 接続定義プロパティを追加するには、追加した定義に対するノードを展開し、［Properties］ノードを右クリックして、［Add］を選択します。次のようにプロパティを入力します。

a ［Name］フィールドに、プロパティ名を入力します。

b ［Type］ドロップダウンリストから、プロパティのタイプを選択します。［java.lang.String］、［java.lang.Boolean］、［Integer］のいずれかを選択します。［<Unspecified>］を選択することもできます。

c ［Value］フィールドに、プロパティの値を入力します。値は、プロパティのタイプに適合している必要があります。

Borland AppServer 6.7 を対象にしたプロジェクトの実行設定の作成

JBuilder は、Borland AppServer 6.7 をターゲットにしたデプロイメント用に、デフォルト設定 jbuilder およびデフォルトパーティション jbpartition を使用します。この設定やパーティションがない場合は、プラグインを設定したときに自動的に作成されます。デフォルトパーティションは、Tomcat と JDataStore サービスと同じポート番号を共有します。パーティションは、デプロイメントされるサービスのタイプごとに自動的に作成されます。サーバー名は、マシンの ID と同じです。

複数の JBuilder 実行設定を使用して、複数のパーティションを起動できます。複数の実行設定を作成するには、次の手順にしたがいます。

1 ［Run|Configurations］を選択し、［New］をクリックします。

2 ［Run Type］を［Server］に変更します。表示されるサーバーは、［Project Properties|Server］を使用してプロジェクトに選択されたサーバーです。

3 ［Category］リストから［Server|Command Line］を選択します。

4 ［Partition］および［Configuration］フィールドで、使用する値に変更します。

5 ［OK］をクリックして［New Runtime Configuration］ダイアログボックスを閉じ、設定を保存します。

6 上に示した手順を繰り返して、他のパーティションを実行するための追加の実行設定を作成します。

7 複数のパーティションを実行するには、Management Agent（［Enterprise|Borland Enterprise Server Management Agent］）を使用します。

ネーミングサービスの競合を回避するために、ネーミングサービスが 1 つのパーティションでのみ有効になっていることを確認してください。あるパーティションでネーミングサービスを無効にするには、そのパーティションの実行設定を編集して（［Edit Runtime Configuration］ダイアログボックス）、［Category］リストの［Naming/Directory］サービスの選択を解除します。

パーティションを起動する前に、Tomcat および JDataStore サービスに対して固有のポート番号が設定されていることを確認してください。

重要 Tomcat のポート設定は、JBuilder の実行設定からは変更できません。サーバーを起動し、Borland AppServer 管理コンソールを開き、サーバー側でポートを設定する必要があります。それには、次の手順にしたがいます。

1 Borland Management Agent をコマンドライン `<APPSERVER_HOME>/bin/scu.exe` で起動します。

2 Borland AppServer 管理コンソールを `<APPSERVER_HOME>/bin/console.exe` で開きます。

3 コンソールにログインします。

4 ［Management Hubs］ノードを選択します。

5 実行パーティションが表示されるまで、［Management Hubs］ノードを展開します。パーティションノードを展開します。

6 ［Web Container］ノードを右クリックし、［Properties］を選択します。［Configure Web Container］ダイアログボックスが表示されます。

7 ［Service: HTTP］ノードを展開します。コネクタを選択します。

8 ［Connector］ページをスクロールして、［Port Number］フィールドを表示します。

9 ポート番号を使用する番号に変更します。

10 ［File ｜ Save］を選択します。

管理ポートの変更

Management Agent は、すべてのパーティションを管理します。［Advanced Settings］ダイアログボックス（［Tools|Configure Servers|Advanced Settings］）に設定されているデフォルトの管理ポートは 42424 です。JBuilder またはサーバーが使用する管理ポートは、変更できます。

JBuilder が使用するポートを変更するには

1 ［Enterprise|Configure Servers］を選択し、左側の［User Home Folder］から［Borland Enterprise Server AppServer Edition 6.x］を選択します。

2 ［Custom］タブをクリックし、［Advanced Settings］ボタンをクリックします。

3 ［Management Port］フィールドで、ポートを変更します（デフォルトは 42424 です）。

4 ［OK］を 2 回クリックします。

サーバーが使用する管理ポートを変更するには

1 JBuilder 2006 に組み込まれている Borland 管理コンソールを開きます（［View|Panes|BAS 6.7 Console］）。

メモ　まず、組み込まれているコンソールを有効にする必要があります。375 ページの「JBuilder での Borland 管理コンソールの表示」を参照してください。

2 ［Installations］をダブルクリックします。

3 サーバーの場所のノードを展開し、サーバーノードを選択します。

メモ　サーバーは、マシン ID でもあります。

4 ［Agents］ノードのサブノードにサーバーが表示されるまで、サーバーノードを展開します。

5 サーバーノードを右クリックし、［Properties］を選択します。

6 ［Management Port］フィールドで、ポート番号を変更します

7 ［OK］をクリックして設定を保存します。

管理ポートを変更すると、JBuilder で起動されていた管理エージェントはシャットダウンします。

JBuilder 2006 でのパーティションの起動

JBuilder でパーティションを起動すると、デフォルトで Management Agent が起動し、サーバーが起動されます。パーティションが起動すると、すべてのデプロイメント可能なアーカイブが自動的にデプロイメントされます。起動時の出力がメッセージペインに表示されます。

複数のパーティションを起動する場合や、パーティションを短時間で再デプロイメントする場合は、Management Agent を起動できます（[Enterprise|Borland Enterprise Server Management Agent]）。Management Agent は、VisiBroker Smart Agent を起動します。スマートエージェントは、JBuilder で ORB を使用できるようにし、初期ブートストラップに関する事項を処理します。

サーバーのパーティションおよび設定を起動するには、実行するプロジェクトペイン内のモジュールを右クリックします。[Run Using <Configuration_Name>] を選択します。通常、この名前はサーバーの実行時設定の名前です。

JBuilder でパーティションを実行すると、次のようになります。

- パーティションおよび設定が存在しない場合は、作成されます。パーティションおよび設定の名前は、サーバーの起動に使用される実行設定に基づいて決定されます。デフォルト設定を使用して設定またはサーバーを起動すると、アプリケーションサーバーのプロパティに設定されているパーティション名が使用されます。
- `jndi-definitions.xml` で定義されているリソースがある場合、そのリソースをパーティションディレクトリのルート（`<APPSERVER_HOME>\var\domains\base\configurations\<CONFIGURATION_NAME>\mos\<PARTITION_NAME>\dars\jbuilder.dar`）にデプロイメントします。`jndi-definitions.xml` ファイルは、この `.dar` ファイルにパッケージされてデプロイメントされます。

メモ `jndi-definitions.xml` ファイルは、データソース／メッセージングリソースが定義されている EJB 2.0 モジュールをプロジェクトが含んでいる場合に作成されます。このアクションは、[Project Properties] ダイアログボックスのサーバーノードの [Deployment|EJBs Service Properties] ページにある [Deploy jndi-definitions.xml] オプション（[Project Properties|Server|Services|Deployment|EJBs]）の選択を解除することによってオフにできます。

- [Remove Archives Already Deployed To Server] オプションを選択すると、パーティションにデプロイメントされているすべてのアーカイブが削除されます。サーバーの実行設定として、[Edit Runtime Configuration] ダイアログボックスの [Server|Archives] カテゴリで、このオプションを設定できます（[Run|Configurations|<Server_Config_Name>|Edit|Run|Category]）。
- 選択されたアーカイブをデプロイメントします。デフォルトでは、プロジェクト内のデプロイメント可能なすべてのアーカイブが選択されます。サーバーの実行設定として、[Edit Runtime Configuration] ダイアログボックスの [Server|Archives] カテゴリで、デプロイメントするアーカイブを選択できます（[Run|Configurations|<Server_Config_Name>|Edit|Run|Category]）。
- パーティションを起動します。パーティションの起動が完了すると、Borland AppServer 6.7 のメッセージペインにパーティションがリストされます。起動時にデプロイメントされるアーカイブがロードされ、アクセス可能になります。

メモ デフォルトでは、パーティションに関連付けられたすべてのサービスが起動します。

デプロイメント

EJB、WAR、および EAR モジュールを Borland AppServer 6.7 にデプロイメントするには、次の手順にしたがいます。

1 ［Enterprise|Configure Servers］を選択します。

2 ダイアログボックスの左側で、［Borland Enterprise Server AppServer Edition 6.x］を選択します。

3 ［Custom］タブをクリックし、サーバー、設定、およびパーティションの名前をサーバーと一致するように設定します（サーバーはリモートまたはローカルマシン上で動作します）。

4 ［Advanced Settings］ボタンをクリックして、Management Port の設定がサーバーのポート設定と一致していることを確認します。

5 ［OK］を 2 回クリックします。

これで、デプロイメントする準備ができました。JBuilder を使用して EJB、WAR、EAR のモジュールをデプロイメントするには、2 つの方法があります。［Deployment］ウィザードを使用する方法と、コンテキストメニューを使用する方法です。

［Deployment］ウィザードを使用してデプロイメントするには

1 プロジェクトをビルドします（［Project|Make］）。

2 Management Agent を起動します（［Enterprise|Borland Enterprise Server Management Agent］）。

3 ［Server Deployment］ウィザードを開きます（［Enterprise|Server Deployment］）。

4 ウィザードの最初のページで、デプロイメントするモジュールを選択します。パーティションがすでに起動している場合は、［Restart Partitions On Deploy (Cold Deploy)］オプションを設定します。［Next］をクリックします。

5 2 ページでは、モジュールをデプロイメントするパーティションをリストから選択します。

重要 選択したパーティションがすでに起動している場合は、パーティションを再起動してデプロイメントモジュールにアクセスします。

6 ［Finish］をクリックしてデプロイメントします。

コンテキストメニューからデプロイメントするには

1 プロジェクトをビルドします（［Project|Make］）。

2 Management Agent を起動します（［Enterprise|Borland Enterprise Server Management Agent］）。

3 プロジェクトペインで、デプロイメント可能なノードを右クリックします。

4 ［Deploy Options|Deploy］を選択します。

メモ 複数のモジュールをデプロイメントする場合は、プロジェクトペインで複数のデプロイメント可能ノードを選択して右クリックし、［Deploy Options］コンテキストメニューを使用できます。

リモートデバッグ

アプリケーションをリモートでデバッグするためには、パーティションを設定する必要があります。詳細は、次のトピックのいずれかを選択してください。

- JBuilder で管理しないパーティションのリモートデバッグの準備
- JBuilder で管理するパーティションのリモートデバッグの準備

設定、パーティション、およびサーバーが起動したら、388 ページの「JBuilder からのリモートデバッグ」のセクションの手順にしたがいます。詳細なデバッグのチュートリアルは、JBuilder オンラインヘルプの *「Developing Enterprise JavaBeans」*にある「Tutorial: Remote debugging with the Borland Enterprise Server AppServer Edition 6.0」を参照してください。手順は、Borland アプリケーションサーバーの 6.x と 6.7 バージョンに共通です。

JBuilder で管理しないパーティションのリモートデバッグの準備

JBuilder で管理しないパーティションのリモートデバッグを準備するには

1 Borland Management Agent をコマンドライン <APPSERVER_HOME>/bin/scu.exe で起動します。

2 Borland AppServer 管理コンソールを <APPSERVER_HOME>/bin/console.exe で開きます。
コンソールにログインします。

3 ［Management Hubs］ノードを選択します。デバッグするパーティションが表示されるまで、ノードを展開します。

4 パーティション名を右クリックし、［Properties］を選択します。
［Partition Properties］ダイアログボックスが表示されます。

5 ［Partition Process Settings］タブを選択します。

6 ［Enable JPDA Remote Debugging］オプションを選択します。

7 ［JPDA Debugging Transport Address］を 3999 に設定します。

8 ［Suspend Partition Until Debugger Attaches］オプションの選択を解除します。

9 ［OK］をクリックします。

JBuilder で管理するパーティションのリモートデバッグの準備

JBuilder で管理されるパーティションのリモートデバッグを準備するには、次の 2 つの方法があります。

1 サーバーをシャットダウンします。

2 ファイル：<APPSERVER_HOME>/var/domains/base/configurations/<CONFIGURATION_NAME>/configuration.xml を開きます。

3 JPDA 要素を見つけ、属性値を次のように編集します。

```
enable-jpda-debug="true"
jpda-transport-address="3999"
jpda-suspend="false"
```

JBuilder からのリモートデバッグ

サーバー、パーティション、および Management Agent が起動したら、JBuilder IDE から次の手順にしたがいます。

1 リモートデバッグセッションを起動するプロジェクトで、［Run|Configurations］を選択します。

2 サーバーの実行設定を選択し、［Edit］を選択します。

3 ［Debug|Connection］ノードを選択します。

4 ［Remote Attach］オプションを選択します。

5 ［Transport Type］を dt_socket に設定して、ローカルホストの値を 3999 に設定します。

6 ［OK］を 2 回クリックして、［Run Configuration］ダイアログボックスを閉じます。

7 デバッグするプロセスにブレークポイントを設定します。

8 ツールバー上の［Debug Project］ボタンの隣の下向き矢印をクリックして、作成または編集したサーバー設定を選択します。デバッガが起動し、リモートで実行中のパーティションにアタッチされて、ブレークポイントで停止します。

第 41 章

Borland AppServer の管理

ここでは、Borland AppServer（BAS）管理アーキテクチャの概要について説明します。また、BAS を適切に管理するために理解する必要がある重要な概念と用語についても説明します。

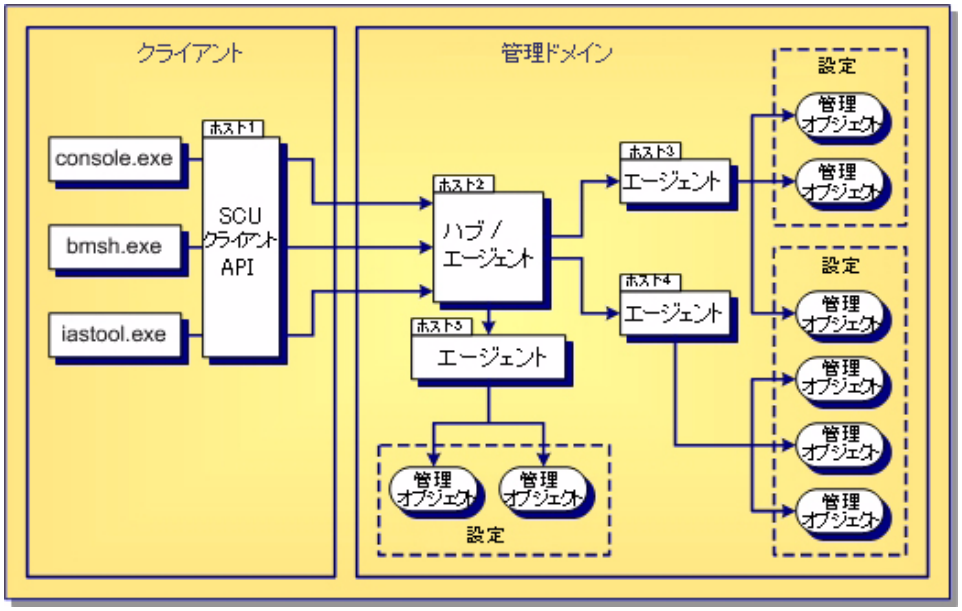
SCU プロセスの概要

AppServer の中心は SCU と呼ばれるプロセスです。SCU にはハブ ロジックとエージェント ロジックの両方が含まれており、1 つのプロセスで 2 つの機能が実現されます。これにより、SCU は次のモードをサポートしています。

- ハブ / ローカル エージェントでは、ハブとエージェントの両方が同一のホストに置かれます。
- ハブ / エージェントでは、ハブとエージェントは別のホストに置かれます。

デフォルトでは、AppServer のインストール時にハブがインストールされます。ただし、SCU にはハブ ロジックとエージェント ロジックの両方が含まれているので、インストール後に管理コンソールを使用して、ハブ / エージェントのモードを変更できます。

次の図では、各種エンティティの関係を説明します。



SCU プロセス（ハブとエージェント）の起動については、391 ページの「ハブとエージェントの起動」を参照してください。

ハブ

ハブは、AppServer 管理システムの管理用コントロール センターです。ハブとエージェントは同一の SCU プロセスの一部であるため、ハブはエージェントとしても動作します。

ハブには、別の名前を指定しない限り、インストール先のホストの名前がデフォルトで割り当てられます。

ハブの起動については、391 ページの「ハブとエージェントの起動」を参照してください。

ハブは、エージェントを使用して、分散リソースだけでなくローカル リソースも管理できます。デフォルトでは、BAS のインストール時にハブもインストールされます。BAS のインストールが完了した後で、管理コンソールを使用してハブをエージェントに変更できます。

エージェント

管理システムの管理者として、ハブは処理をエージェントに委任します。委任を受けたエージェントは、その処理を実装します。

ハブとエージェントは同じ SCU プロセスの一部なので、ハブには少なくとも 1 つのエージェント（ローカル エージェント）があります。ローカル エージェントは、ハブと同じホストに置かれます。

エージェントを、ハブをインストールしたホストとは別のホストにインストールできます。インストールするエージェントには、インストール先のホストの名前がデフォルトで割り当てられます。

デフォルトでは、AppServer のインストール時にハブがインストールされます。ただし、SCU にはハブ ロジックとエージェント ロジックの両方が含まれているので、インストール後に管理コンソールを使用して、ハブ / エージェントのモードを変更できます。

エージェントをハブに変更するには：

1. 管理コンソールのツリー構造で、エージェント ノードを展開します。
2. プロパティを変更するエージェントを右クリックします。
3. 表示されたメニューの ［Properties］を選択します。
4. ［Properties］ダイアログに ［General］タブが表示されていることを確認します。
5. ［Agent type］ドロップダウン メニューから、［Agent］を選択します。
6. ［OK］をクリックします。

重要 管理コンソールからエージェントを再起動するようにプロンプトが表示されます。変更した内容を有効にするには、エージェントを再起動する必要があります。

ハブとエージェントの起動

SCU 実行可能ファイル（`scu.exe`）を起動すると、ハブとエージェントが起動されます。

ハブとそのローカル エージェントの起動

ハブとそのローカル エージェントを起動するには：

1. ハブ / ローカル エージェントのインストール先であるマシン上で、コマンド プロンプト ウィンドウを開きます。
2. `<install_dir>/bin` に移動します。
3. 次のコマンドを入力し、Enter キーを押します。

   ```
   scu
   ```

   または

   ハブ / ローカル エージェントをインストールした Windows マシンで、［スタート］メニューから次のように選択します。

   [スタート｜すべてのプログラム｜ Borland AppServer ｜ Borland Management Agent]

エージェントの起動

エージェントを起動するには：

1 エージェントのインストール先であるマシン上で、コマンド プロンプト ウィンドウを開きます。

2 <install_dir>/bin に移動します。

3 次のコマンドを入力し、Enter キーを押します。

scu

または

エージェントをインストールした Windows マシンで、［スタート］メニューから次のように選択します。

[スタート｜すべてのプログラム｜ Borland AppServer ｜ Borland Management Agent]

メモ NFS ファイルシステムに BAS がインストールされており、これが NFS 全体で実行されている場合、SCU プロセスを起動しようとしたときに "There may be an agent running in this footprint" というメッセージが表示されることがあります。この問題は、ベンダーが異なる UNIX 実装で、クライアントとサーバーの特定の組み合わせの場合にのみ発生します。この問題を回避するために、BAS を NFS にインストールしないことをお勧めします。

管理ポート

ハブとそのエージェントは、互いを認識できるように、同じ管理ポートを設定する必要があります。ハブまたはエージェントをインストールすると、管理ポートにはデフォルトで 42424 が設定されます。ただし、ハブまたはエージェントの管理ポートは、後で管理コンソールを使って変更できます。

管理ポート番号を変更するには：

1 Borland 管理コンソールを開きます。

2 ツリー構造で、ポートを変更するエージェント名を右クリックし、［Properties］を選択します。

3 ［Properties］ダイアログに ［General］タブが表示されていることを確認します。

4 ［Management Port］フィールドにポート番号を入力します。

5 ［OK］をクリックします。

重要 管理コンソールからエージェントを再起動するようにプロンプトが表示されます。変更した内容を有効にするには、エージェントを再起動する必要があります。

管理ドメイン

管理ドメインは、単一の管理ポートを共有するハブとエージェントのセットで定義されます。複数のハブとエージェントで同じ管理ポートを使用でき、また、これらを同一の管理ドメイン内に配置できます。

設定

設定は、管理ドメインのエンティティを管理するためにハブが参照する情報の中心的なコンテナです。各ハブに対して、複数の設定を作成できます。単一の設定を複数のエージェントに適用できます。単一のエンティティとして監視および制御するリソースのコレクションは、設定内で定義します。

設定は、エージェントではなくハブに対して作成します。ハブだけが設定内の情報を理解および解釈することができます。作成された設定に基づいて、ハブは処理をエージェントに委任します。各エージェントは、そのホスト上に処理を実装します。

設定の詳細については、395 ページの「設定」を参照してください。

管理リソース

管理リソースは、管理するシステムの実際のエンティティです。次に例を示します。

- メモ帳などのテキスト エディタ
- Apache などの Web サーバー
- JDataStore などのデータベース
- Web ページ
- ping

管理オブジェクト

BAS では、設定、制御、監視などを行う各管理リソースを表すために管理オブジェクトを使用します。ハブやそのエージェントが管理する各管理リソースごとに、設定内に管理オブジェクトが存在する必要があります。

管理オブジェクトを使用して、管理リソースを次の 2 つのいずれかの方法でモデル化します。

- 管理リソースのライフサイクルの制御
- 状態の確認のみ

管理オブジェクトの BAS 管理は、ハブからの指示に応じてエージェントによって実装されます。各管理オブジェクトは、管理ドメイン内の同じエージェントに割り当てる必要があります。ただし、各エージェントは複数の管理オブジェクトを管理できます。

設定テンプレートを使って設定を作成する場合、テンプレート内の管理オブジェクトごとにエージェントを 1 つずつ指定します。エージェントに割り当てられた各管理オブジェクトでは、次の場合に、ハブがエージェントに管理オブジェクトの設定情報を送信します。

- 管理オブジェクトが追加されたとき
- 設定内の管理オブジェクト情報を更新したとき

管理オブジェクトの詳細については、399 ページの「管理オブジェクト」を参照してください。

設定テンプレートの詳細については、395 ページの「設定テンプレート」を参照してください。

Borland 管理コンソール

Borland 管理コンソールは、グラフィカル ユーザー インターフェイスです。Borland 管理コンソールでは、論理的な設定と物理的な設定の両方をグラフィクで表示できるほか、各設定の `configuration.xml` ファイルにアクセスできます。BAS 管理のすべての段階（設定の作成および変更から設定状態の監視および診断まで）で、管理コンソールを使用する必要があります。

第 42 章

設定

Borland AppServer（BAS）では、設定を使用して、分散リソースを機能の依存関係に基づいてグループ化、管理ができます。さらに、これらの定義されたリソースを単一のエンティティとして制御、監視、および管理することもできます。ここでは、システム リソースを管理する際の設定の重要な役割と、設定ライフサイクルの各段階について説明します。

設定の作成

新しい設定を作成するには、管理コンソールを使用する必要があります。設定は常にハブのコンテキストで作成し、設定内で定義した各管理オブジェクトは必ずエージェントに割り当てます。

設定の作成では、次の一部または全部で構成される設定テンプレートを使用します。

- 設定全体のプロパティ、要素、および属性
- その設定に適した管理オブジェクト
- 管理オブジェクトに固有のプロパティ、要素、および属性

BAS には、設定の作成後に管理オブジェクトやプロパティを追加、削除、および変更するための機能があります。ただし、最初に設定を作成した後で、管理オブジェクトとエージェント間の割り当てを変更しないことをお勧めします。

詳細については、399 ページの「管理オブジェクト」を参照してください。

設定テンプレート

設定テンプレートはモデルであり、設定は、このモデルを使って作成した実際のインスタンスです。BAS には、一般的な設定を短時間で正確に作成できるさまざまな設定テンプレートが用意されています。独自の設定を構築するための土台として便利な設定テンプレートもあれば、そのまま使用したり、独自の分散環境に合わせて変更して使用することができる一般的な設定のサンプルもあります。設定にアクセスしたり、設定を作成するには、管理コンソールを使用します。

設定を作成するには：

1 管理コンソールのツリー構造で、設定を作成するハブに移動し、ツリーを展開します。

2 設定を右クリックし、［Add Configuration］を選択します。

設定テンプレートの［Template Gallery］ダイアログが表示されます。このダイアログを使用して、目的に合ったテンプレートのカテゴリに移動し、その設定テンプレートを選択します。

メモ デフォルトで表示される設定テンプレートのカテゴリは、インストールされている Borland 製品によって異なります。すべてのカテゴリを表示し、すべての利用可能な設定テンプレートにアクセスするには [Favorite Categories] ドロップダウン リストをクリックして、[All categories] を選択します。

configuration.xml ファイル

設定情報は `configuration.xml` ファイルに格納されます。ハブは、`configuration.xml` という名前のファイルを探すようにコーディングされています。したがって、すべての `configuration.xml` ファイルは名前を変更**しない**でください。

設定の実装

設定の作成と変更以外に、管理コンソールでは設定を開始、監視、および停止することも実行できます。設定を実行すると、エージェントは、管理オブジェクトに対して次の*処理*の一部または全部を実装します。エージェントが実装できる基本処理は次のとおりです。

- 開始
- 停止
- Ping
- 強制終了

設定内の各管理オブジェクトには、各基本処理のデフォルトが設定されています。ただし、この設定は管理コンソールから上書きできます。詳細については、「管理オブジェクト」の 405 ページの「管理オブジェクトのアクション」を参照してください。

設定の開始

設定を開始するには、管理コンソールを使用します。設定を開始すると、ハブが各エージェントに設定の開始を通知します。各エージェントは、設定内の各管理オブジェクトに定義された開始順序に基づいて、管理リソースを開始します。

設定の実行

管理リソースが実行されると、各エージェントは、設定の管理オブジェクトに定義されたプロパティに基づいて管理オブジェクトの ping 処理を実行して、状態を定期的に確認します。ping から状態の変化が返されると、エージェントはそれをハブに通知します。

管理リソースが失敗していることが ping によって確認された場合、デフォルトでは、設定に定義されている管理オブジェクトの開始ストラテジをエージェント自体がただちに実装することにより、個別に BAS の自動回復機能を呼び出します。管理リソースが実行を再開したら、エージェントは状態の変化をハブに通知します。

設定の停止

設定は停止されるまで実行します。設定を停止するには、管理コンソールを使用します。設定を停止すると、ハブはエージェントに設定の停止を通知します。各エージェントは、設定の管理オブジェクトに定義された停止順序に基づいて、管理リソースを停止します。

エージェントが管理リソースを停止できない場合は、管理コンソールから手動で設定の管理オブジェクトを強制終了できます。

設定タスクのスケジュール

スケジュールされたタスクを設定に追加すると、設定内の MO に依存しないプロセスを開始できます。スケジュールされたタスクは、手動で開始や中止ができます。また、指定した（複数の）時間ルールに基づいて自動的にタスクを開始することもできます。スケジュールされたタスクを設定に追加する方法の詳細については、405 ページの「設定のタスクのスケジュール」を参照してください。

BAS インフラストラクチャの停止

ハブやスレーブ エージェントの SCU プロセスを停止しても、またはそのいずれかが突然ダウンしても、管理リソースの状態には反映されないことに注意してください。ハブまたはスレーブ エージェントのホストで SCU プロセスを停止しても、管理されているリソースは停止しません。

重要 ハブまたはスレーブ エージェントの SCU プロセスが実行されていない場合、管理リソースはアクティブに管理されません。

たとえば、次のようになります。

1. 設定 A を実行します。
2. 設定 A により、ハブは管理リソース 1 を開始するようにスレーブ エージェント 1 に指示します。
3. 開始されたスレーブ エージェント 1 は、管理オブジェクトの開始ストラテジと ping ストラテジに基づいて、管理リソース 1 に ping します。
4. 管理リソース 1 の実行中に、スレーブ エージェント 1 の SCU プロセスがダウンします。
5. 管理リソース 1 の状態は、対応するエージェントがダウンしても影響を受けないため、処理は続行されます。
6. ただし、スレーブ エージェント 1 がダウンした場合、管理リソース 1 は管理されなくなるため、自動回復機能は無効になります。たとえば、スレーブ エージェント 1 がダウンしているときに、管理リソース 1 が実行を停止した場合は、次のようになります。
   - 状態の変化はハブには通知されません。
   - 管理オブジェクトの開始ストラテジは実装されません。

ローカル ハブ / エージェントの停止

デフォルトでは、ハブ / エージェントの SCU プロセスを停止すると、管理リソースの状態が影響を受け、SCU プロセスがシャットダウンする前に、すべての管理リソースが停止します。ただし、SCU プロセスが突然ダウンした場合、管理リソースの状態は影響を受けません。

重要 ハブ / エージェントが実行されてない場合、管理リソースはアクティブに管理されません。

`agent.shutdown.policy` プロパティは、ハブ / エージェントのこのような動作を制御します。このプロパティは、エージェントのプロパティ ファイルで設定されます。このプロパティ ファイルは、デフォルトで `<install_dir>\var\domains\<domain_name>\adm\properties` に置かれています。

このプロパティの有効な値とその説明を次に示します。

プロパティ	値	説明
agent.shutdown.policy	■ stop - （ハブのみに適用）同じエージェントにすべての MO があるすべてのローカル設定を停止します。 ■ stop-and-escalate - （ハブのみに適用）すべてのローカル設定を停止します。停止しない場合は、強制終了に移行します。 ■ leave（デフォルト）- 終了時にすべての設定を現在の状態に保ちます。 ■ local-stop - 設定 / グループ ルールに関係なく、またグループの順序にも関係なく、エージェントがシャットダウンしたときに、エージェント上のすべての管理オブジェクトを停止しようとします。 ■ local-stop-and-escalate - SCU がある場合、このエージェント上のすべての MO をグループの順序などに関係なく停止します。停止しない場合は、強制終了に移行します。	ローカル ハブ / エージェントの SCU プロセスが突然停止した場合に、実行されていた設定の管理リソースを制御します。

ハブ / エージェントの再起動

ハブ / エージェントの SCU プロセスを停止してから再起動すると、デフォルトでは、SCU プロセスの停止時に実行されていたすべての設定が自動的に再起動されます。この動作は、agent.shutdown.policy プロパティが stop-and-escalate または stop に設定されていたハブ / エージェントに適用されます。

agent.shutdown.policy プロパティが leave （デフォルト値）に設定されていたハブ / エージェントを停止してから再起動すると、ハブ / エージェントは、シャットダウン時に実行されていた設定の管理を再開します。ハブ / エージェントは各管理リソースの状態を識別し、設定情報に基づいて必要な処理を行います。

第 43 章

管理オブジェクト

設定、制御、または監視だけを行う管理リソースやそのグループを表すために**管理オブジェクト**を使用します。

管理オブジェクトの作成と追加

新しい管理オブジェクトを作成して設定に追加するには、管理コンソールを使用します。設定を作成する場合、選択する設定テンプレートには、その設定に関係する管理オブジェクトが含まれています。

管理オブジェクトを既存の設定に追加する場合も、使用する管理オブジェクトのテンプレートを管理コンソールで選択します。管理オブジェクトの各テンプレートには、テンプレートが作成して設定に追加する管理オブジェクト タイプに固有の情報があらかじめ定義されています。

BAS には、設定に追加した管理オブジェクトを追加、削除、および変更するための機能があります。ただし、最初に設定を作成した後で、管理オブジェクトとエージェント間の割り当てを変更しないことをお勧めします。

詳細については、399 ページの「管理オブジェクト」を参照してください。

管理オブジェクト テンプレート

管理オブジェクト テンプレートはモデルであり、このテンプレートを使って作成する管理オブジェクトが実際のインスタンスです。BAS には、設定で使用する管理オブジェクトを短時間で正確に作成できる管理オブジェクト テンプレートの一式が組み込みで用意されています。

一部の設定テンプレートとは異なり、管理オブジェクト テンプレートはどれもサンプルではありません。すべてのテンプレートは、有効な管理オブジェクト タイプのモデルとして、管理リソースを表すために使用できます。管理オブジェクトにアクセスしたり、管理オブジェクトを作成するには、管理コンソールを使用します。

管理オブジェクトを作成して設定に追加するには：

1. 管理コンソールのツリー構造で、管理オブジェクトを作成して追加する設定に移動し、ツリーを展開します。
2. 設定の名前を右クリックし、［Add Managed Object］を選択します。

   管理オブジェクト テンプレートの［Template Gallery］ダイアログが表示されます。このダイアログを使用して、目的に合った管理オブジェクトのカテゴリに移動し、その管理オブジェクト テンプレートを選択します。

メモ 利用できるすべての管理オブジェクト テンプレートにアクセスするには、[Favorite Categories] ドロップダウン リストをクリックし、[All categories] を選択します。

メモ Apache Web サーバーの管理オブジェクトを作成する場合、管理オブジェクト名にスペースを入れることはできません。"Apache Managed Object" など、管理オブジェクト名にスペースが含まれると、Apache Web サーバーは起動に失敗します。

メモ UNIX のテキスト ファイルを管理オブジェクトのテンプレートの一部としてダウンロードする場合は、改行コード（Ctrl-M）に関する UNIX プラットフォームのルールに従う必要があります。このファイルを Windows のテキスト エディタを使用して編集した場合は、余分な Ctrl-M が含まれていないことを確認してください。カスタム実行可能ファイル管理オブジェクト テンプレートの一部として .sh ファイルが含まれている場合は、この点に注意してください。

管理オブジェクト タイプ

BAS は、必要な管理オブジェクトの `type` 属性を使用して、管理オブジェクトの制御方法および設定内で探す管理オブジェクト固有の情報を決定します。BAS は、設定内に各管理オブジェクトのタイプ属性値を書き込みます。これは、管理オブジェクトを含む設定を追加したとき、または既存の設定に管理オブジェクトを追加したときに行われます。

管理オブジェクト タイプ	管理オブジェクト サブタイプ	適用対象 ...
順序付きグループ 冗長グループ 状態プロキシ		純粋管理グループ（管理リソースを表さない）
プロセス		単純プロセス管理リソース
Java プロセス	VBJ プロセス	Java プロセス管理リソース
カスタム JavaScript カスタム実行可能ファイル		拡張性管理リソース
osagent VisiNaming プロセス Apache Web サーバー プロセス OTS Tibco パーティション パーティション サービス		BAS 固有管理リソース

メモ 管理エージェントで IIS サーバーを管理できるようにするには、Windows の管理コンソールを使用して、IISAdmin と WWW サービスの [スタートアップの種類] を [手動] に設定する必要があります。また、[回復] の各アクションは [何もしない] に設定してください。

純粋管理の管理オブジェクト タイプ

設定内で管理リソースの純粋管理に使用する管理オブジェクト タイプは次のとおりです。

- 順序付きグループ
- 冗長グループ
- 状態プロキシ

これらのタイプの管理オブジェクトは、単にほかの管理オブジェクトのコンテナです。設定内でこれらのタイプの管理オブジェクトを使用して、機能的に従属関係を持つ管理リソースのグループを表します。さらに、これらの管理オブジェクト グループをネストすることもできます。

順序付きグループ

順序付きグループ管理オブジェクトを使用して、設定内で管理オブジェクトのコレクションをグループ化することには、次の目的があります。

- グループのコンテキスト内で管理オブジェクトを開始および停止するときに BAS が使用する順序を定義する。
- グループ内の管理オブジェクトが表す管理リソースの状態を集計する。

たとえば、Web 要求の処理にデータベースへのアクセスを利用している Web サーバーを管理するように BAS を設定するとします。この場合、Web サーバーを起動する前にデータベースを利用できる状態にしておく必要があります。

1. 順序付きグループ管理オブジェクトを作成して、設定に追加します。
2. 次に示す管理オブジェクトを作成して、順序付きグループに追加します。
   - Web サーバー管理オブジェクト
   - データベース管理オブジェクト
3. BAS が最初にデータベースを開始し、次に Web サーバーを開始するように、開始順序を設定します。

   このように開始順序を定義することで、BAS はデータベースが正常に起動するまで Web サーバーが開始されないように制御できます。

この設定を実行した場合の実行順序は次のとおりです。

1. 設定が順序付きグループ管理オブジェクトを開始すると、データベースが起動されます。
2. データベースが正常に起動している場合にだけ、BAS は Web サーバーの起動を試みます。

この設定を停止した場合の実行順序は次のとおりです。

- 設定が順序付きグループ管理オブジェクトを停止すると、Web サーバーが最初に停止し、次にデータベースが停止します。

管理リソースを順番に開始および停止するように BAS を設定できるだけでなく、同じ開始順序または停止順序を複数の管理オブジェクトに割り当てることにより、BAS が管理オブジェクトを同時に開始または停止するようにできます。

メモ 各設定自体も順序付きグループです。設定には、設定のコンテキスト内で開始順序と停止順序を定義した管理オブジェクトのコレクションが含まれます。管理オブジェクトの状態は集計され、最終的に設定の状態として表されます。

メモ `<install_dir>/var/templates/configurations/examples` ディレクトリの `start_dependency` フォルダと `fail_dependency` フォルダにある各 `readme` ファイルを参照してください。このファイルでは、開始とエラーの依存関係が順序付きグループ MO に対してどのように機能するかを Op-Center の設定例を使って示しています。

設定の作成と順序付きグループ管理オブジェクトの追加の詳細については、395 ページの「設定の作成」と 399 ページの「管理オブジェクトの作成と追加」を参照してください。

冗長グループ

冗長グループ管理オブジェクトを使用して、同種の管理リソースのコレクションをグループ化できます。冗長管理オブジェクト タイプを使用して、グループのメンバー間にフェイルオーバーを設定できます。

デフォルトでは、設定を実行して冗長グループを開始すると、すべてのメンバーが開始されます。ただし、管理コンソールを使用して、冗長グループを次の方法でカスタマイズすることができます。

- BAS が開始および実行するメンバーの数を指定する。
- BAS が開始および実行するメンバーの最小数を指定する。
- BAS が開始および実行するメンバーの最大数を指定する。

これらの値にはそれぞれ、グループ内のメンバー総数または全体のメンバー総数（デフォルト）を上限として指定できます。

冗長グループ管理オブジェクトを停止するために設定を停止すると、実行中のすべてのメンバーも同時に停止します。

たとえば、3 台の Web サーバーが同じ機能を実行し、フェイルオーバーを使用するように設定されている場合、これらのサーバーを BAS で管理するには、次の処理を実行します。

1 冗長グループ管理オブジェクトを作成して、設定に追加します。

2 次に示すメンバーを作成して、グループに追加します。

- Web サーバー A の管理オブジェクト
- Web サーバー B の管理オブジェクト
- Web サーバー C の管理オブジェクト

3 冗長グループに次の値を定義します。

- 1 台は必須
- 指定最小数は 2 台
- 指定最大数は 2 台

この設定を実行した場合の実行順序は次のとおりです。

1 設定が冗長グループ管理オブジェクトを開始すると、BAS は Web サーバー A と Web サーバー B を同時に起動して実行しようとします。

メモ BAS がメンバーを開始する優先順位は、設定でのメンバーの順番（上から下）です。

2 この例では、これらの Web サーバーの 1 台を実行すると、グループに関する値の必要条件は満たされ、実行状態が冗長グループ管理オブジェクトに集計されます。

3 ただし、BAS は指定最小数の 2 台が実行されるまでメンバーの開始を続行します。たとえば、Web サーバー B が起動し、Web サーバー A が起動していない場合、BAS は Web サーバー C を起動しようとします。

4 設定が正常に実行され、メンバーを利用できる限り、BAS は指定最小数である 2 つのメンバーの実行を継続します。

この設定を停止した場合の実行順序は次のとおりです。

- 冗長グループ管理オブジェクトを停止する前に、実行中のすべての Web サーバー メンバーを停止する必要があります。BAS は、実行中のすべてのメンバーを同時に停止しようとします。

メモ `<install_dir>/var/templates/configurations/examples/fault_tolerance/datadir` ディレクトリの `readme` ファイルを参照してください。このファイルでは、Op-Center の設定例のフォールト トレランスの例を使って冗長グループをフェイルオーバーとフォールト トレランスに使用する方法を示しています。

設定の作成と冗長グループ管理オブジェクトの追加の詳細については、395 ページの「設定の作成」と 399 ページの「管理オブジェクトの作成と追加」を参照してください。

状態プロキシ

状態プロキシ管理オブジェクトを使用して、設定内のほかの場所で定義された別の MO の状態を表します。状態プロキシは次の特性を持ちます。

- プロキシの委任元 MO と同じ設定内の任意の場所で使用できます。
- ほかの MO と同様に所属先グループの状態と動作に影響を与えます。

状態プロキシは、それが表す MO の現在の「実行」状態（「実行中」、「停止」など）を反映しますが、実行中の動作（「開始中」、「停止中」など）は反映しません。状態プロキシは、それが表す MO のライフサイクルを制御しません。

状態プロキシは、複数の順序付けグループが同じ MO に対する依存関係を共有する場合に役立ちます。プロキシの委任元 MO がダウンすると、状態プロキシも同様にダウンします。このような状態の変化に適切に対応するように順序付けグループ MO をそれぞれ設定できます。

たとえば、2 つの順序付けグループ MO を個別に作成し、それらがネーミング サービス MO の同じインスタンスに依存するように設定するには、次の手順に従います。

1. 最初の順序付けグループ GroupA を作成します。
2. ネーミング サービス MO を GroupA に追加します。
3. 2 番めの順序付けグループ GroupB を作成します。
4. 状態プロキシ MO を GroupB に追加します。
5. 状態プロキシ MO の論理名には、GroupA で作成されたネーミング サービス MO の名前を設定します。構文は `${agent.name}/mo-name` を使用し、たとえば、`${agent1.name}/nsdb_jds` とします。

   GroupB の状態プロキシ MO が開始すると、GroupA のネーミング サービス MO を ping します。ネーミング サービス MO が停止された場合は状態プロキシ MO も停止し、両方のグループの状態は、この状態の変化を反映します。子が失敗したときにシャットダウンするように GroupB を設定した場合、GroupB はシャットダウン シーケンスを開始します。GroupB の動作は、ネーミング サービス MO 自体や GroupA には影響しません。

メモ `<install_dir>/var/templates/configurations/examples/state_proxy/datadir` ディレクトリの `readme` ファイルを参照してください。このファイルでは、Op-Center の設定例の状態プロキシ MO の例を使って状態プロキシの動作を示しています。

プロセス管理オブジェクト タイプ

プロセス管理オブジェクトは、管理リソースに直接関連付けられます。プロセス管理オブジェクトは、シングル プロセス（VisiBroker for Java プロセスなど）である管理リソースを表すために使用します。管理リソースは、オペレーティング システムに対してコマンドを発行することによって開始され、オペレーティング システムから ping と停止を受け付けます。

設定の作成とプロセス グループ管理オブジェクトの追加の詳細については、395 ページの「設定の作成」と 399 ページの「管理オブジェクトの作成と追加」を参照してください。

UNIX：プロセス管理オブジェクトの所有権の管理

UNIX ホストの特権ポートにアクセスするプロセスは、適切なアクセス許可を持つ必要があります。たとえば、ユーザー `root` のアクセス許可付きでプロセスを開始する必要があります。通常、`root` のアクセス許可付きで開始する必要があるプロセスは、設定内の一部のプロセスだけです。`setuser` スクリプトを使用すると、プロセス管理オブジェクトが `root` として、または `root` のアクセス許可付きで開始できるように BAS を設定できます。`setuser` ツールの使い方とマルチユーザー モードについては、**『インストール ガイド』**の「setuser ツールによる所有権の管理」を参照してください。

Java プロセス管理オブジェクト タイプ

Java プロセス管理オブジェクトは、管理リソースに直接関連付けられます。Java プロセス管理オブジェクトは、シングル Java プロセスである管理リソースを表すために使用します。管理リソースは、オペレーティング システムに対してコマンドを発行することによって開始され、オペレーティング システムから ping と停止を受け付けます。

設定の作成と Java プロセスまたは VBJ プロセス管理オブジェクトの追加の詳細については、395 ページの「設定の作成」と 399 ページの「管理オブジェクトの作成と追加」を参照してください。

拡張性管理オブジェクト タイプ

これらの管理オブジェクト タイプは、管理リソースに直接関連付けられます。管理オブジェクトの処理ストラテジのコードをカスタマイズする機能があるため、BAS を拡張することができます。これらのタイプにより、さまざまなリソースを管理することができます。

カスタム JavaScript 管理オブジェクト タイプ

カスタム JavaScript 管理オブジェクト タイプは、JavaScript 内で開始、ping、停止、および強制終了の処理ストラテジを実装する管理リソースを表すために使用します。BAS は JavaScript インタープリタを使用して、これらのストラテジを実行します。

重要 （UNIX のみ）管理オブジェクトがマルチユーザー モード（MUM）のエージェントによって開始される場合は、通常、JavaScript を実行できません。マルチユーザー モード（MUM）で実行される管理オブジェクトのいずれかに JavaScript が含まれる場合は、`agent.config` を変更して、JavaScript の実行を許可する必要があります。

カスタム実行可能ファイル管理オブジェクト タイプ

カスタム実行可能ファイル管理オブジェクト タイプは、各処理ストラテジのプロセスを実行する管理リソースを表すために使用します。

AppServer 固有管理オブジェクト タイプ

これらの管理オブジェクト タイプは、設定で AppServer 管理リソースに直接関連付けられています。

AppServer 設定と AppServer 管理オブジェクトの追加の詳細については、395 ページの「設定の作成」と 399 ページの「管理オブジェクトの作成と追加」を参照してください。

AppServer 管理オブジェクト タイプ	次の単一インスタンスを表すために使用
osagent	VisiBroker スマート エージェント - CORBA オブジェクト間の通信と RPC に使用する簡単なディレクトリ サービス
ネーミング サービスのプロセス	VisiBroker ネーミング サービス - 分散ディレクトリ サービス
Apache Web サーバー プロセス	Apache Web サーバー
OTS	VisiTransact - 2 フェーズ コミット トランザクション サービス
Tibco	Tibco Java メッセージ サービス（JMS）
パーティション	AppServer パーティション - J2EE コンポーネントと CORBA コンポーネントにサービスを提供するオブジェクト ホストの基本単位
パーティション サービス	次の AppServer パーティション サービスのいずれかを示します。 ■ EJB コンテナ ■ JDataStore データベース ■ Tomcat Web コンテナ ■ Java セッション サービス ■ VisiBroker インターセプタ ■ VisiConnect ■ OpenJMS

管理オブジェクトのアクション

管理リソースのライフサイクルを制御する各管理オブジェクトに対して、エージェントは次の基本処理のすべてを実行できます。

- 開始
- 停止
- Ping
- 強制終了

管理リソースの状態を確認する目的で設定された各管理オブジェクトに対して（pinger）、エージェントは ping 処理だけを実行します。

メモ **UNIX のみ**：管理エージェントは、JDataStore サーバーが `-ui none` オプションを使用して起動されていない限り、管理オブジェクト プロセスとして停止することはできません。例として、Pet Store クラスタを参照してください。

ストラテジ

エージェントが管理オブジェクト処理を実行する方法は、" ストラテジ " によって決定されます。各処理で使用できるストラテジは、管理オブジェクトの `type` によって決定されます。設定内の各管理オブジェクトに対して、各処理のデフォルトのストラテジが設定されます。ただし、管理コンソールから別のストラテジを指定したり、`configuration.xml` を直接編集することで、デフォルトのストラテジを上書きできます。`control-overrides` 要素内で、すべての処理パラメータを指定し、デフォルトのアクション ストラテジを上書きします。

設定のタスクのスケジュール

スケジュールされたタスクを設定に追加すると、設定内の MO に依存しないプロセスを開始できます。たとえば、毎週最初の営業日と最後の営業日のピーク以外の処理時間にバックアップ プロセスを開始するようにスケジュールされたタスクを追加できます。

Op-Center は、スケジュールされたタスクを設定された時間に開始します。このタスクは、設定が実行されていなくても開始されます。スケジュールされたタスクは設定の状態に影響しません。また、設定の状態もスケジュールされたタスクの動作に影響しません。スケジュールされたタスクが 1 つ以上含まれる設定において、タスクは、設定のほかの部分とは論理的に独立しています。

スケジュールされたタスクは次の特性を持ちます。

- 名前があります。
- 実行場所となるエージェントがあります。
- 1 つのプロセスを実行するように設定されます。
- タスクのスケジュールに 1 つ以上のルールを関連付けることができます。

メモ スケジュールされたタスクは、関連付けられたプロセスを実行するように設定できるだけです。プロセスを停止するようには設定できません。

スケジュールされたタスクを設定するには：

1. 設定を右クリックし、［Add Scheduled Task］を選択します。
2. オブジェクトの情報を入力し、［Next］をクリックします。
3. 実行するプロセスの処理設定を入力し、詳細処理設定を行ってから、［Next］をクリックします。
4. タスク スケジュールに 1 つ以上のルールを追加するには、［New］をクリックします。
5. 詳細オプションを設定するには、［Next］をクリックします。
6. ［Task Schedule］タブや［Advanced Options］タブでエントリを作成したら、［Finish］をクリックします。

管理オブジェクトの可用性スケジュールの作成

メモ AppServer パーティション サービスには、可用性スケジュールを設定できません。ただし、AppServer パーティション サービスを含む設定には、スケジュールされたタスクを設定できます。パーティション サービスの一覧については、404 ページの「パーティション サービス」を参照してください。設定のタスクをスケジュールするには、405 ページの「設定のタスクのスケジュール」を参照してください。

各管理オブジェクトは、MO の状態を「実行中」または「停止」に変更するタイミングを示す一連の時間依存ルールをオプションで持つことができます。このルールは、ルールを適用する開始時間を示す `cron` 形式の時間指定方法で表されます。MO のルールの場合は、適用された後のルールの継続時間も示されます。同じ時間帯に複数のルールを設定した場合は、最後のルールが使用されます。

MO のプロパティ エディタにある［Availability Schedule］タブを使用して、MO の実行状態を制御するルールを追加できます。MO の［Availability Schedule］タブで追加したルールを実装するには、設定が実行されている必要があります。スケジュールされたタスクに対して入力されるルールとは異なり、特定の MO に設定されるルールでは、MO の状態を「停止」または「実行中」のいずれかに指定できます。また、設定された状態が有効な継続時間も設定できます。たとえば、定期的なバックアップやメンテナンスの間、または休日のシャットダウンの間に、設定全体をシャットダウンしないで特定の MO だけを停止するルールを設定できます。設定した期間が過ぎると、MO はデフォルトの状態に戻ります。

MO の設定では MO のデフォルトの状態が「実行中」であると指示され、MO に設定されたルールでは MO の状態を「停止」に変更するように指示された場合は、ルールが優先されて、MO は停止されます。ただし、MO の設定では MO のデフォルトの状態が「停止」であると指示され、MO に設定されたルールでは MO の状態を「実行中」に変更するように指示された場合は、MO の設定のデフォルトの状態が優先されて、MO は停止状態のままになります。

可用性スケジュールの各ルールには、設定されたルールから生成された説明（" 毎日、夜の 12 時に 1 分間停止する " など）か、指定したカスタム名（" 休日のシャットダウン " など）のどちらかが付きます。ルールは標準的な `cron` 形式の構文で定義されます。それには、［Availability Schedule］タブの［Time Rule］ページにある入力メカニズムを使って MO のプロパティを編集するか、［Time Rule］ページの指示された場所で `cron` 構文自体を指定します。ルールは入力順に適用されます。ただし、ルールを設定した後で、いつでも並べ替えることができます。

メモ ルート管理オブジェクトに実行ルールを設定しても、設定は開始されません。設定は、管理者が開始する必要があります。

第 44 章

Borland 管理シェルの概要

Borland 管理シェル（BMSH）はスクリプト環境であり、ローカルとリモートのどちらでも AppServer を設定および実行するためのスクリプトを記述できます。BMSH でスクリプトを記述することにより、手作業で実行するとエラーが発生しやすい Borland 管理コンソール、テキスト エディタ、ほかの Borland AppServer ツールなどを組み合わせて実行する必要がある操作を自動化できます。

BMSH スクリプトは、テキスト エディタを使って記述でき、コンパイルや前処理を必要としないテキスト ファイルです。BMSH は、オープンソースの Mozilla Rhino プロジェクトを使って構築されています。BMSH で使用するキーワード、文、式、および演算子は、Web ページで使用する JavaScript と同じです。BMSH では、AppServer インストールに対応したスクリプトを記述するために、Mozilla Rhino の組み込み標準コマンドを拡張しています。また、Borland AppServer との間のスクリプト可能な API を提供するオブジェクトが用意されています。JavaScript の詳細については、Mozilla Rhino のドキュメントを参照してください。

BMSH の使い方

BMSH の使用例は次のとおりです。

- 通常のファイル システム操作の実行
- AppServer の起動と停止
- AppServer インスタンス内のパーティションの管理と設定
- AppServer サービスの設定と実行

BMSH の実行

BMSH スクリプトを実行するには次の 2 つの方法があります。

- `bmsh>` プロンプトでコマンドを対話形式で入力する。
- コマンド プロンプト（Windows）またはシェル プロンプト（UNIX）からスクリプト ファイルを実行する。パスが更新されていない場合は、インストールの `bin` ディレクトリに移動する必要があります。

メモ BMSH スクリプトは、すべてのプラットフォームで実行できます。

対話型の BMSH の使用（対話モード）

BMSH を対話モードで起動するには、コマンド プロンプトで次のように入力します。

```
bmsh
```

bmsh> プロンプトが表示されます。BMSH では、AppServer インストールのパスが PATH 環境変数に含まれていると仮定されます。

対話型シェルを終了するには、bmsh> で次のように入力します。

```
quit()
```

重要 必ず括弧を付ける必要があります。

対話モードの例

ローカル インストールの管理ポートを表示するには、次のように入力します。

```
bmsh>bes.getManagementPort();
```

管理ポートが返されます。

たとえば、次のようになります。

```
42424
```

ローカル インストールの管理ポートを設定するには、次のように入力します。

```
bmsh>bes.setManagementPort(portnumber);
```

検索に使用する管理コンソールの管理ポートを表示するには、次のように入力します。

```
bmsh>bes.getConsolePort();
```

管理コンソールの管理ポートを設定するには、次のように入力します。

```
bmsh>bes.setConsolePort("portnumber");
```

シェルでは、プロンプトに対して入力することにより、すべての BMSH API を実行できます。使用するすべての変数の定義は、シェル セッションを終了するまで有効です。この方法は、スクリプトのコードを対話的にテストしたり、1 ～ 2 行の簡単な操作を実行する場合に便利です。BMSH の対話的な動作は、Mozilla-Rhino リリースと同じです。複雑な操作を繰り返して実行する場合は、次の節で説明するようなスクリプト ファイルを記述すると便利です。

BMSH スクリプト ファイルの使い方（バッチモード）

スクリプト ファイルを実行するには、BMSH バッチ モードを使用します。コマンド プロンプト（Windows）またはシェル プロンプト（UNIX）で、JavaScript コマンドとその引数を含むファイルを指定します。次に示すサンプルは、Borland AppServer インストールの <install_dir>/examples/bmsh/ にあります。

BMSH サンプル ディレクトリ（<install_dir>/examples/bmsh）に移動し、次のように入力します。

```
bmsh setManagementPort.js ?
```

このスクリプトにより、setManagementPort.js スクリプトの使い方が示されます。

次のスクリプトを実行すると、AppServer のローカル インストールのすべての管理ポートが設定されます。たとえば、コマンド プロンプトまたはシェル プロンプトで次のように入力します。

```
bmsh setManagementPort.js -p 33333
```

BMSH のファイルとフォルダ

ここでは、AppServer インストールの BMSH コンポーネントに関係するスクリプト ファイルが格納されているディレクトリについて説明します。ユーザーは独自の BMSH スクリプトを任意のディレクトリに置くことができます。

製品に付属する BMSH スクリプトは、次の場所にインストールされます。

```
<install_dir>/bin/bscript/autoload/
<install_dir>/bin/bscript/scripts/
<install_dir>/examples/bmsh/autoload/
<install_dir>/examples/bmsh/scripts
```

bin ディレクトリ内の BMSH のファイルとフォルダ

`<install_dir>/bin` ディレクトリには、AppServer インストールに対してグローバルに適用できるファイルとフォルダがあります。ここにある BMSH スクリプトは、製品に付属するほかのバイナリ ファイルと同様に扱う必要があります。ユーザーは変更しないでください。このディレクトリのファイルとフォルダは、インストール ディレクトリのデフォルト名や場所、およびファイルの内容によって異なる場合があります。ただし、このディレクトリのファイルとフォルダに、特定の BMS 設定やユーザー定義変数に関する固有情報を含めてはなりません。

/bin/bscript/autoload サブディレクトリ内の BMSH のファイルとフォルダ

`<install_dir>/bin/bms/autoload/` サブディレクトリには、BMSH の起動時にロードされるスクリプトが格納されています。BMSH の自動呼び出し機能の詳細については、410 ページの「自動呼び出し機能」を参照してください。

/bin/bscript/scripts 内のファイル

`<install_dir>/bin/bscript/scripts/` サブディレクトリには、Borland が記述しサポートするスクリプトが格納されています。このディレクトリ内のスクリプトをコマンドライン ツールとして使用して、BMS デバッグのオン / オフなど実行頻度が高い一般的な BMSH 機能を実行できます。スクリプトには、偶発的な変更や削除を防止するために読み取り専用の属性が設定されます。BMSH には、スクリプトの検索パス機能があります。スクリプトがコマンドラインに入力されると、このディレクトリでスクリプトが検索されます。

メモ ここに置かれているスクリプトは、BMSH の使い方を示すサンプルとして役立ちます。これらのスクリプトをコピーしてカスタマイズすることができます。ただし、カスタマイズした独自のスクリプトはここに保存しないでください。

examples ディレクトリ内の BMSH サブディレクトリ

`<install_dir>/examples/bmsh` サブディレクトリには、BMSH のサンプル スクリプトとサポート マニュアルが格納されています。これらのスクリプトは、独自のカスタマイズ スクリプトを作成する場合にテンプレートとして使用できます。

/examples/bmsh 内のファイル

`<install_dir>/examples/bmsh` サブディレクトリには、スクリプトとフォルダが格納されています。このサブディレクトリの目的は単に操作方法を示すことなので、Borland はこれらのスクリプトの機能性や操作性を保証していません。

/examples/bmsh/scripts 内のファイル

<install_dir>/examples/bmsh/scripts サブディレクトリには、<install_dir>/examples/bmsh/scripts サブディレクトリ内のサンプル スクリプトより使用頻度が低いサンプル スクリプトが格納されています。ここに置かれているスクリプトは、一般に特定の BMS 設定やアプリケーションに依存し、完成されていない場合や説明が省略されている場合があります。Borland は、これらのスクリプトの機能性や操作性を保証していません。

/examples/bmsh/autoload 内のファイル

<install_dir>/examples/bmsh/autoload サブディレクトリには、BMSH の自動呼び出し機能の使い方を示すサンプル スクリプトが格納されています。

BMSH の機能

BMSH には、Mozilla-Rhino 環境を拡張するいくつかの機能があります。

自動呼び出し機能

この BMSH の拡張メカニズムを使用して、BMSH のグローバルな変数、オブジェクト、および関数を定義することができます。この拡張メカニズムを使用して、インストールやデプロイメントの必要性に合わせて高度にカスタマイズした BMSH 環境を作成することもできます。BMSH が起動すると、自動呼び出しフォルダ内のスクリプトが実行されます。デフォルトの自動呼び出しフォルダは、<install_dir>/bin/bscript/autoload です。BMSH は自動呼び出しフォルダ内のスクリプトを実行してから、対話型コマンドやコマンドラインのスクリプト ファイルを実行します。BMSH は起動時にスクリプトをロードして実行するため、起動時間が長くなる可能性があります。

インストールが終了したら、組み込みの自動呼び出しファイルと 2 つのサンプルを使用できます。組み込みの自動呼び出しファイルは mgmt_utils.js という名前で、ポートを設定する管理ドメイン機能の簡単なラッパーが格納されています。インストールされた一部のコマンドライン スクリプトでは、このファイルがすでに自動的にロードされているとみなし、自動的にロードされる変数と関数の名前の先頭に下線を付けて表します。

BMSH examples フォルダには次の 2 つのサンプルがあります。これらのサンプルは、自動呼び出しフォルダにコピーするだけで試すことができます。次のとおりです。

- javaarray.js - JavaScript での Java 配列の作成を簡略化するユーティリティ ルーチン
- sendmail.js - LiveConnect の使い方を示すサンプル。このサンプルは、javax.mail インターフェイスだけを使用し、JavaScript オブジェクトは使用しません。このスクリプトでは、javaarray.js ファイルを使用する必要があります。

メモ AppServer オブジェクトと Hub クラスは、JavaScript 拡張メカニズム（bes_ext.js と hub_ext.js）を使用して拡張されます。これらの拡張メカニズムは次の場所にあります。

```
<install_dir>/bin/bscript/autoload
```

これらの拡張メカニズムには正式のマニュアルはありません。ただし、これらのファイルを開くことにより、API 拡張メカニズムを確認することができます。

検索パス機能

コマンドラインにスクリプトが入力されたとき、BMSH は現在のディレクトリ以外に特定のパスでスクリプトを検索します。BMSH の検索パスには、デフォルトで次のパスが設定されています。

```
<install_dir>/bin/bscript/scripts/
```

BMSH クラスパスへの JAR 機能の追加

JAR を BMSH クラスパスに追加するには、インストールの `bin` ディレクトリにある `bmsh.config` ファイルを編集します。先頭が次のコメントで始まるセクションを探します。

```
# Set up the list of jars in the product system directory
```

パスを JAR に追加します。パスには絶対パスを指定する必要があります。ただし、パスが製品インストールからの相対パスである場合は、可搬性のある `$var(installRoot)` 変数を使用できます。

BMSH オブジェクト

Rhino JavaScripting シェルには、ECMA JavaScript 標準で定義されているオブジェクト（`Date` や `RegExp` など）があります。BMSH では、AppServer 固有のオブジェクトを追加しています。BMSH オブジェクトは次のカテゴリに分類できます。

- 事前にインスタンス化されたオブジェクト - ファイル システムとの対話や、プロパティ ファイルの値の抽出と設定に使用するメソッドなどが格納されたオブジェクト
- ユーザーがインスタンス化するオブジェクト - Borland 管理ハブとの対話、プレースホルダーへのプロパティ値の代入、または HTTP Web ページとの対話に使用するメソッドなどが格納されたオブジェクト
- 静的オブジェクト - BMS エージェントの機能を使用してプロセスを起動、停止、ping するメソッドや、XML ファイルを解析および保存するメソッドなどが格納されたオブジェクト
- ファクトリ オブジェクト - 他のオブジェクト（通常は事前にインスタンス化されたオブジェクト）が生成したオブジェクトで、プロパティ ファイルでの値の抽出や設定またはテキスト ファイルの読み書きを行うメソッドなどが格納されたオブジェクト

事前にインスタンス化された BMSH オブジェクト

事前にインスタンス化されたオブジェクトでは、"new" を実行する必要はありません。BMSH が起動されるたびに、事前にインスタンス化されたオブジェクトが生成されます。事前にインスタンス化されたオブジェクトは、インスタンス化しないで使用できます。API の詳細については、API のマニュアルを参照してください。BMSH には、事前にインスタンス化された次の 2 つのオブジェクトがあります。

- `bes` - BMSH が実行されるローカル AppServer インストールに対して API を提供します。このオブジェクトは、前述の対話型サンプル内に示されています。
- `fso` - ローカル ファイル システムにアクセスするための汎用的な関数を提供します。`fso` には、AppServer ローカル プロパティ ファイルのプロパティを取得および設定するいくつかの Borland AppServer 固有のメソッドが含まれています。

ユーザーがインスタンス化する BMSH オブジェクト

BMSH は、ユーザーがインスタンス化する組み込み Mozilla Rhino オブジェクト（`Date`、`RegExp`）を拡張します。ユーザーがインスタンス化する BMSH オブジェクトでは、スクリプトの作成者が "new" を使ってインスタンスを生成することが必要です。このリリースには、ユーザーがインスタンス化する BMSH オブジェクトが 1 つだけあります（`XDOM`：XML ファイルにアクセスするためのスクリプト可能な API）。

BMSH ファクトリ オブジェクト

BMSH には、ほかのオブジェクトが生成したオブジェクト（ファクトリ オブジェクト）が含まれています。最も一般的なファクトリ オブジェクトは、ファイルを読み書きする `TextStream` オブジェクトです。`fso` オブジェクトは、これらのオブジェクトに対してファクトリとして動作します。"new" を使ってファクトリ オブジェクトを生成することはできません。

例

```
var inputStream = fso.openTextStream("a_filename",1) // 1 = 読み取り
```

`inputStream` は、BMSH の `TextStream` オブジェクトです。

Java LiveConnect

Rhino JavaScript インタープリタには、Java をその場でコンパイルする LiveConnect メカニズムがあります。このメカニズムにより、Java コードをスクリプト ファイル内に置くことができます。この機能は、BMSH が提供していない簡単な機能に適しています。たとえば、Java 実行時オブジェクトを使用して、別のプロセスを実行したり指定した時間だけ停止することができます。また、`joptionpane.js` ダイアログ ボックスなどの簡単なユーザー ダイアログ ボックスを表示することもできます。このメカニズムでは、AppServer JAR ファイルの機能にアクセスできません。

Java コードをスクリプト ファイルに組み込む方法については、`<install_dir>/examples/bmsh/autoload/sendmail.js` というサンプルを参照してください。

独自の Rhino スクリプト可能オブジェクトの記述

すべての BMSH オブジェクトは、Rhino のスクリプト可能オブジェクトから派生します。これは、オブジェクトをシェルに追加するための Rhino の標準メカニズムです。BMSH は Rhino に基づいているため、AppServer のユーザーもこのメカニズムを使用して、選択したオブジェクトを BMSH に追加することができます。このメカニズムでは、Java コンパイラを使用する必要があります。また、Java に関する高度な知識も必要です。

BMSH を拡張してオブジェクトを追加するには、`bmsh.config` ファイルを次の手順で編集する必要があります。

1. `<install_dir>/bin` ディレクトリ内の `bms.config` ファイルを開きます。
2. `# Set up the list of jars in the product system directory` というコメントで始まるセクションに移動します。
3. シェルに追加するオブジェクトが格納された JAR ファイルの絶対パスをポイントする次のような設定を追加します。

   ```
   addpath $var(installRoot)/lib/dom4j.jar
   addpath $var(installRoot)/lib/scu.jar
   addpath $var(installRoot)/lib/scu_client.jar
   ...
   ```

   たとえば、次のようになります。

   ```
   addpath $var(installRoot)/lib/my.jar
   ```

   または

   ```
   addpath c:/myfolder/my.jar
   ```

環境変数

Rhino 環境の JavaScript 配列を使用する BMSH では、環境変数を使用できます。たとえば、次のようになります。

```
for(i in environment)
  print(i +"=" + environment[i])
```

このスクリプト コードを切り取り、BMSH シェルに貼り付けます。次に、`bmsh` プロンプトで BMSH シェルを実行し、すべての BMSH 環境変数を表示します。

JavaScript 配列

JavaScript 配列は、連想配列（ハッシュテーブル）として実装されます。インデックスは文字列または数値です。たとえば、単一の環境変数値を取り出すには、`bmsh` プロンプトで次のスクリプト コードを実行します。

```
environment["user.dir"]
```

上記のサンプルでは、現在の作業ディレクトリが表示されます。これは次と同じです。

```
print( environment["user.dir"] )
```

または、次のように値を変数に代入することもできます。

```
var cwd = environment["user.dir"];
print( cwd );
```

Java 文字列配列

Java 文字列配列に対しては、JavaScript 配列とは異なる処理が行われます。たとえば、次のようになります。

```
var files = fso.findFiles(environment["user.dir"],"*",false);
for(var i=0; i < files.length; i++ )
   print("files["+ i + "]=" + files[i])
```

`findFiles()` ルーチンは、Java `String[]` を返します。文字列配列には、数値でのみアクセスできます。文字列配列はハッシュされていません。

JavaScript 組み込み関数

BMSH では、JavaScript 正規表現プロセッサを使用できます。`Date()` オブジェクトも使用できます。

BMSH の対話的な動作

BMSH の対話的な動作は、Mozilla-Rhino の操作と同じ規則に従います。

- `bmsh` プロンプトで行を入力し、**Enter** キーを押すと、スクリプトが実行されます。
- スクリプトで、行の最後の文字が左中括弧（`{`）の場合、右中括弧（`}`）が入力されるまでスクリプトは実行されません。この規則は、中括弧に関する C 言語の規則に準拠しています。
- スクリプトのステートメントが次の行に続く場合、セミコロン（`;`）を使用するかどうかはオプションです。

BMSH ヘルプ

グローバル関数 help() を使用すると、**『Borland 管理シェル API リファレンス』**の情報も BMSH シェルから使用できます。次の表では、help() コマンド オプションについて説明します。

オプション	説明
help();	表示内容 : Rhino ヘルプ。これは、Mozilla-Rhino リリースから変更されていません。 Rhino インタープリタのグローバルな組み込み関数を表示します。
help(*jsObject* | *jsClass*);	jsClass は、 BMSH JavaScript クラス名の関数とプロパティを表示します。 BMSH クラスだけが有効です。 たとえば、 help(Dom4j) です。 jsObject は、 BMSH JavaScript オブジェクトの関数とプロパティを表示します。 BMSH オブジェクトだけが有効です。 たとえば、 bmsh>help(appserver); です。
help(jsObject | *jsClass*, "String *function*" | "*property*");	jsObject | jsClass JavaScript は、 BMSH JavaScript オブジェクトまたは JavaScript クラスです。 function | property は、 関数パラメータの名前と型、および jsObject のプロパティの戻り型を表示します。 このパラメータは、 引用符で囲む必要があります。
Help(*jsObject* | *jsClass*, String *function* | *property*, boolean *verbose*);	jsObject | jsClass - BMSH JavaScript オブジェクト インスタンスまたは JavaScript クラスです。 function | property - 関数または jsObject のプロパティの名前です。このパラメータは、引用符で囲む必要があります。 たとえば、 help(appserver,"installRoot"); です。 verbose - 関数またはプロパティのパラメータや戻り値などの詳細な説明を表示します。 この情報は、 API のマニュアルに記述されている内容と同じです。 たとえば、次のようになります。 bmsh>help(appserver,"getManagementPort",true);

第 45 章

設定および configuration.xml ファイル

すべての設定情報は configuration.xml ファイルに格納されます。作成した設定ごとに 1 つの configuration.xml が生成され、ハブ ホストに格納されます。ハブ は、configuration.xml ファイルを探すようにコーディングされているため、このファイルの名前は変更しないでください。

設定要素

各 configuration.xml ファイル内のすべての設定情報は、configuration 要素内に記述されます。

```
<configuration ...attributes...>
        <configuration-id ...attributes.../>
        <main-root ...attributes.../>
        <managed-objects ...attributes...>
                ...
        </managed-objects>
        <properties>
                ...
        </properties>

</configuration>
```

configuration 要素の属性

次に configuration 要素の属性を示します。

属性	必須	デフォルト値	説明
revision	はい	該当なし	configuration.xml のバージョン。
display-name	はい	該当なし	設定の作成時に設定に付けた一意の名前。
description	いいえ	該当なし	設定に関する簡単な説明です。この説明は、管理コンソール内のいくつかのビューに表示されます。
small-icon	いいえ	該当なし	内部的に使用されます。

```
<configuration revision="revisionnumber"
 description="string" display-name="string">
       <configuration-id ...attributes.../>
       <main-root ...attributes.../>
       <managed-objects ...attributes...>
             ...
       </managed-objects>
       <properties>
             ...
       </properties>

</configuration>
```

configuration 要素のサブ要素

次に configuration 要素のサブ要素を示します。

要素	説明
configuration-id	設定の名前とバージョン情報を含みます。詳細については、「configuration-id 要素」を参照してください。
main-root	設定の開始アクションや停止アクションが実装される管理オブジェクトへのリファレンスを含みます。また、main-root として識別された管理オブジェクトによって設定の状態が表されます。詳細については、「main-root サブ要素」を参照してください。
managed-objects	設定に対して定義された管理オブジェクトおよび各管理オブジェクトに固有の設定情報を含みます。詳細については、「管理オブジェクトの要素および属性」を参照してください。
properties	定義された設定全体のプロパティを含みます。詳細については、「設定プロパティの使い方」を参照してください。

```
<configuration ...attributes...>
       <configuration-id ...attributes.../>
       <main-root ...attributes.../>
       <managed-objects ...attributes...>
             ...
       </managed-objects>
       <properties>
             ...
       </properties>

</configuration>
```

configuration-id 要素

各 configuration.xml では、configuration-id 要素内に次の設定識別情報が格納されます。

属性	必須	デフォルト値	説明
name	はい	該当なし	設定の名前。この名前を使用して、管理コンソールの表示で設定を識別します。設定を作成した後で、この名前を変更することはできません。
version	該当なし	1	設定のバージョン。設定の変更を保存するたびに、この値が 1 ずつ増加します。

```
<configuration ...attributes...>
       <configuration-id name="string" version="integer"/>
       <main-root ...attributes.../>
       <managed-objects ...attributes...>
              ...
       </managed-objects>
       <properties>
              ...
       </properties>

</configuration>
```

main-root サブ要素

main-root サブ要素は、設定のメインルートである 1 つの管理オブジェクトを識別します。

メインルート管理オブジェクトは

- その設定を開始したときに開始します。
- その設定を停止したときに停止します。
- その設定の状態を表します。

デフォルトでは、テンプレートを使って設定を作成すると、設定順序付きグループ管理オブジェクトが作成されます。デフォルトでは、設定内のほかのすべての管理オブジェクトは、この設定管理オブジェクトの子であり、これがメインルートになります。

main-root 要素には、次の属性があります。

属性	必須	デフォルト値	説明
mo-ref	はい	${HUB.name}/${CONFIG.name} - 設定順序付きグループ管理オブジェクト。${HUB.name}/${CONFIG.name} という置換文字列構文にしたがって、設定が属するハブの名前と設定の名前で構成される設定管理オブジェクトの論理名。	<Configuration_hub_name>/ <managed_object_name> という構文を使用して、設定内の 1 つのメインルートになる管理オブジェクトの論理名です。 **メモ：**設定を別のハブに移植するために、そのハブに対して置換文字列 ${HUB.name} を使用できます。

```
<configuration ...attributes...>
       <configuration-id ...attributes.../>
       <main-root mo-ref="managed_object_logical_name"/>
       <managed-objects ...attributes...>
              ...
       </managed-objects>
       <properties>
              ...
       </properties>
</configuration>
```

第 46 章

管理オブジェクトの要素および属性

ここでは、configuration.xml の managed-objects 要素、その属性、およびサブ要素をリストおよび説明します。また、各設定管理オブジェクト タイプで使用できるすべての要素、サブ要素、および属性について説明します。

一般的な XML 定義

管理オブジェクトは、configuration.xml ファイルの managed-objects 要素内にリストされます。各管理オブジェクトは、managed-object サブ要素内で定義されます。

```
<configuration>
    ...
    <managed-objects>
        <managed-object />
        ...
    </managed-objects>
    ...
</configuration>
```

managed-object 要素の属性

設定内で定義する各管理オブジェクトでは、次の属性の一部または全部を使用できます。これらの属性は、configuration.xml の managed-object 要素内に記述されます。

属性	必須	デフォルト値	説明
Agent	はい	設定の作成対象のハブと同じホストに存在するローカル エージェント。デフォルトでは、このハブ/ローカル エージェント名は、文字列変数 ${HUB.name} で表されます。	管理オブジェクトの割り当て先のエージェントの名前。指定できる値は次のとおりです。 ■ ${HUB.name}（デフォルト） ■ *hub_name*（デフォルトでは、ハブがインストールされるホストの名前） ■ *agent_name*（デフォルトでは、スレーブ エージェントがインストールされるホストの名前）
name	はい	一般的な管理リソース名または null。	管理オブジェクトに指定した論理名。管理オブジェクトを作成し、設定に追加した後は、この属性の値を変更できません。ただし、display-name 属性を使用することで、管理コンソールを介して表示される名前はいつでも変更できます。
display-name	いいえ	文字列変数 ${MO.name} によって表される管理オブジェクト name 属性値。	管理コンソールに表示する名前が、管理オブジェクト name 属性値のかわりに表示されます。
initial-manage	いいえ	true	AppServer でアクティブに管理する管理リソースに対しては、true（デフォルト）に設定します。true に設定すると、管理オブジェクトは「管理下」になり、設定を実行すると、AppServer はその管理リソースに対するアクションを実行してその状態を維持します。 AppServer でオブジェクトの状態の変更だけを監視し、アクションを実装しない場合は、false に設定します。
initial-monitor	いいえ	true	エージェントからの ping の間に AppServer で状態を監視する管理リソースに対しては、true（デフォルト）に設定します。 エージェントからの ping の間に AppServer で状態を監視しない管理リソースに対しては、false に設定します。
type	はい	設定を作成したり、管理オブジェクトを作成して設定に追加するために使用したテンプレートによって決定。	AppServer は、この属性を使用して、管理オブジェクトの制御方法および設定内で探す type 固有の情報を決定します。詳細については、「管理オブジェクト type 属性」を参照してください。

属性	必須	デフォルト値	説明
initial-desired-state	いいえ	Stopped	管理オブジェクトを最初に設定に追加するとき、管理オブジェクトの状態は、デフォルトで Stopped になります。管理オブジェクトを最初に作成して追加するときに、その管理オブジェクトを実行する場合は、Running に設定します。
data-directory	いいえ	${CONFIG.path}/mos/${MO.name}。ここで、置換文字列変数 ${CONFIG.path} は configuration.xml ディレクトリの絶対パス、${MO.name} は管理オブジェクト name 値を表します。	内部使用。
version	いいえ	該当なし	管理リソースのバージョンを指定します。
vendor	いいえ	該当なし	管理リソースのベンダーを指定します。
description	いいえ	該当なし	管理リソースを説明します。
deploy-data	いいえ	false	内部使用。

```
<configuration>
    ...
    <managed-objects>
          <managed-object agent="{$HUB.name}|hub_name|agent_name"
     name="managedobject_name"
     [display-name="${MO.name}|display_name"
     initial-desired-state="stopped|running"
     data-directory="${CONFIG.path}/mos/${MO.name}|fullyqualifiedpath"
     deploy-data="true|false" initial-manage="true|false"
     initial_monitor="true|false"]
     type="ordered-group|redundancy-group|state-proxy|process|java-process|
     userdefined-jscript|userdefined-executable|visientity|visiserver|osagent|
     apache-process|ots|tibco|partition|">
     [vendor="vendorname" version="version"
     description="description"]
               ...
          </managed-object>
          ...
    </managed-objects>
    ...
</configuration>
```

管理オブジェクト type 属性

次に、type 属性で使用できる値について説明します。

値	説明
ordered-group	1 体として管理できる関連する 1 組の管理オブジェクトを定義します。これにより、オブジェクトの階層を構築できます。ordered-group 自体は単なるプレースホルダー（論理構造）です。AppServer が制御する実際の管理オブジェクトは、グループの メンバーです。順序付きグループ管理オブジェクトについて詳細は、「順序付きグループ」を参照してください。ordered-group 管理オブジェクト type の詳細については、「ordered-group 管理オブジェクト タイプ」を参照してください。
redundancy-group	管理オブジェクトの集合を表すという点で ordered-group に似ています。redundancy-group 内の管理オブジェクトは、機能的に同等です。冗長グループ管理オブジェクトについて詳細は、「冗長グループ」を参照してください。redundancy-group 管理オブジェクト type の詳細については、「redundancy-group 管理オブジェクト タイプ」を参照してください。
state-proxy	プロキシ化された MO の存続期間に影響を与えることなく、設定の別の場所で定義されている別の MO の状態を表すために使用します。状態プロキシ管理オブジェクトについて詳細は、「状態プロキシ」を参照してください。state-proxy 管理オブジェクト type の詳細については、「state-proxy 管理オブジェクト タイプ」を参照してください。
process	シングル プロセスである管理リソースを表すために使用します。管理リソースは、オペレーティング システムに対してコマンドを発行することによって開始され、オペレーティング システムから ping と停止を受け付けます。プロセス管理オブジェクトの一般的な情報については、「プロセス管理オブジェクト タイプ」を参照してください。process 管理オブジェクト type の詳細については、「process 管理オブジェクト タイプ」を参照してください。
java-process	シングル Java プロセスである管理リソースを表すために使用します。管理リソースは、オペレーティング システムに対してコマンドを発行することによって開始され、オペレーティング システムから ping と停止を受け付けます。Java プロセス管理オブジェクトの一般的な情報については、「Java プロセス管理オブジェクト タイプ」を参照してください。java-process 管理オブジェクト type の詳細については、「java-process 管理オブジェクト タイプ」を参照してください。
custom-javascript	JavaScript で開始、ping、停止、および強制終了アクションを実装する管理リソースの 1 つのインスタンスを表すために使用します。カスタム JavaScript 管理オブジェクトの一般的な情報については、「拡張性管理オブジェクト タイプ」を参照してください。custom-javascript 管理オブジェクト type の詳細については、「custom-javascript 管理オブジェクト タイプ」を参照してください。
custom-executable	開始、ping、停止、および強制終了アクションの各プロセスを実行する管理リソースの 1 つのインスタンスを表すために使用します。カスタム実行可能管理オブジェクトの一般的な情報については、「拡張性管理オブジェクト タイプ」を参照してください。custom-executable 管理オブジェクト type の詳細については、「custom-executable 管理オブジェクト タイプ」を参照してください。
osagent	VisiBroker スマート エージェントの 1 つのインスタンスを表すために使用します。CORBA オブジェクト間の通信と RPC に使用する簡単なディレクトリ サービス osagent 管理オブジェクトの一般的な情報については、「AppServer 固有管理オブジェクト タイプ」を参照してください。osagent 管理オブジェクト type の詳細については、「osagent 管理オブジェクト タイプ」を参照してください。

値	説明
apache-process	Apache Web サーバーの 1 つのインスタンスを表すために使用します。Apache Web サーバー管理オブジェクトの一般的な情報については、「AppServer 固有管理オブジェクト タイプ」を参照してください。apache-process 管理オブジェクト type の詳細については、「apache-process 管理オブジェクト タイプ」を参照してください。
ots	VisiTransact（2 フェーズ コミット トランザクション サービス）の 1 つのインスタンスを表すために使用します。OTS 管理オブジェクトの一般的な情報については、「AppServer 固有管理オブジェクト タイプ」を参照してください。ots 管理オブジェクト type の詳細については、「ots 管理オブジェクト タイプ」を参照してください。
tibco	Tibco Java Messaging Service（JMS）の 1 つのインスタンスを表すために使用します。Tibco JMS 管理オブジェクトの一般的な情報については、「AppServer 固有管理オブジェクト タイプ」を参照してください。tibco 管理オブジェクト type の詳細については、「tibco 管理オブジェクト タイプ」を参照してください。
partition	AppServer パーティション（AppServer でオブジェクトをホストする際の基本単位。特に J2EE コンポーネントと CORBA コンポーネントにサービスを提供する）の 1 つのインスタンスを表すために使用します。Partition 管理オブジェクトの一般的な情報については、「AppServer 固有管理オブジェクト タイプ」を参照してください。 partition 管理オブジェクト type の詳細については、「partition 管理オブジェクト タイプ」を参照してください。

ordered-group 管理オブジェクト タイプ

ここでは、ordered-group 管理オブジェクト type のサブ要素とサブ要素の属性をリストおよび説明します。

メモ ordered-group 管理オブジェクト属性については、420 ページの「managed-object 要素の属性」を参照してください。

ordered-group のサブ要素

ordered-group のサブ要素は、次のとおりです。

ordered-group のサブ要素

サブ要素	説明
ordered-group	ordered-group 管理オブジェクト type 固有の設定情報を含みます。

```
<configuration>
    ...
   <managed-objects>
          <managed-object ...type="ordered-group"...>
               <ordered-group>
                     ...
               </ordered-group>
          </managed-object>
           ...
    </managed-objects>
    ...
</configuration>
```

さらに、managed-object のサブ要素 time-rules（タイプ固有ではない）を使用して、管理リソース プロセスを開始または停止するための cron 形式のルールを定義できます。

サブ要素	説明
time-rules	設定された時間ルールが実行される前の管理オブジェクトの基本状態を記述する属性を含みます。詳細については、「管理オブジェクト time-rules 要素」を参照してください。

```
...
   <managed-objects>
          <managed-object ...type="ordered-group"...>
               [<time-rules attributes >
                     ...
               </time-rules>]
               <ordered-group>
                     ...
               </ordered-group>
          </managed-object>
           ...
    </managed-objects>
...
```

managed-object のサブ要素 control-overrides （タイプ固有ではない）を使用して、デフォルトのアクション ストラテジやアクション パラメータを上書きできます。

サブ要素	説明
control-overrides	デフォルトを上書きするために指定されるアクション ストラテジやアクション パラメータを含みます。詳細については、「管理オブジェクト control-overrides 要素」を参照してください。

```
...
   <managed-objects>
          <managed-object ...type="ordered-group"...>
               [<control-overrides>
                     ...
               </control-overrides>]
               <ordered-group>
                     ...
               </ordered-group>
          </managed-object>
           ...
    </managed-objects>
...
```

ordered-group サブ要素の属性

次に ordered-group サブ要素の属性を示します。

属性	必須	デフォルト値	説明
fail-policy	いいえ	なし	ordered-group のメンバーが失敗したときのアクションを定義します。使用できる値は次のとおりです。 ■ none - 順序付きグループは、どのグループメンバーも停止しようとしません。 ■ ordered-stop - 順序付きグループは、失敗したメンバーより小さい stop シーケンス値を持つすべてのメンバーを member サブ要素の **stop** 属性で指定された順序で停止します。すべてのメンバーを停止した後で、順序付きグループは、メンバーの start 属性で指定された順序でメンバーを再開しようとします。 ■ stop-all - 順序付きグループは、すべてのメンバーを停止しようとします。すべてのメンバーが正常に停止すると、順序付きグループの状態は Failed に変更されます。通常、次に順序付きグループは、メンバーの start 属性で指定された順序でメンバーを再開しようとします。 ■ shutdown - 順序付きグループは、stop 属性で指定された順序ですべてのメンバーを停止しようとします。すべてのメンバーが停止すると、順序付きグループの状態は Cannot Start に設定されます。
cannot-start-policy	いいえ	なし	ordered-group のメンバーが開始に失敗したときのアクションを定義します。使用できる値は次のとおりです。 ■ none - 順序付きグループは、どのグループメンバーも停止しようとしません。 ■ ordered-stop - 順序付きグループは、開始に失敗したメンバーより小さい stop シーケンス値を持つすべてのメンバーを member サブ要素の **stop** 属性で指定された順序で停止します。すべてのメンバーを停止した後で、順序付きグループは、メンバーの start 属性で指定された順序でメンバーを再開しようとします。 ■ stop-all - 順序付きグループは、すべてのメンバーを停止しようとします。すべてのメンバーが正常に停止すると、順序付きグループの状態は **Failed** に変更されます。通常、次に順序付きグループは、メンバーの start 属性で指定された順序でメンバーを再開しようとします。 ■ shutdown - 順序付きグループは、stop 属性で指定された順序ですべてのメンバーを停止しようとします。すべてのメンバーが停止すると、順序付きグループの状態は Cannot Start に設定されます。

```
...
<managed-objects>
       <managed-object ...type="ordered-group"...>
            <ordered-group [fail-policy="null|ordered-stop|stop-all|shutdown"
            cannot-start-policy="null|ordered-stop|stop-all|shutdown"]>
                ...
            </ordered-group>
       </managed-object>
        ...
```

```
</managed-objects>
...
```

member サブ要素

サブ要素	必須	デフォルト値	説明
member	はい	該当なし	グループの各管理オブジェクト メンバーと、グループのコンテキスト内で各メンバーの開始順と停止順を指定します。

```
...
 <managed-objects>
        <managed-object ...type="ordered-group"...>
           <ordered-group>
                 [<member attributes/>]
                 [<member attributes/>]
                 [<member attributes/>]
                 [...]
           </ordered-group>
        </managed-object>
        ...
 </managed-objects>
...
```

member サブ要素の属性

次に member サブ要素の属性を示します。

属性	必須	デフォルト値	説明
mo-ref	はい	該当なし	ここでグループ化されるオブジェクトを表す管理オブジェクトの論理 AppServer 名。 *agent-name*/*mo-name* という構文に従います。
start	いいえ	1 - グループにメンバーがまだ存在しない場合。 **既存メンバーの最大開始値 + 1** - 既存のメンバーを含むグループにメンバーを追加する場合。	ordered-group のほかのメンバーに対してこの管理リソースが開始する順序。グループ内のメンバーの総数以下の正の整数である必要があります。start 順序は、"1" で始まって増加する数です。最初に開始するメンバーを設定するには、この属性を "1" に設定します。同時に複数のメンバーを開始するように設定するには、それらのメンバーに対してこの属性を同じ数に設定します。
stop	いいえ	start 順の逆。 1 - 開始順がない場合。	ordered-group のほかのメンバーに対してこのオブジェクトが停止する順序。グループ内のメンバーの総数以下の正の整数である必要があります。stop 順序は、"1" で始まって増加する数です。最初に停止するメンバーを設定するには、この属性を "1" に設定します。同時に複数のメンバーを停止するように設定するには、それらのメンバーに対してこの属性を同じ数に設定します。

属性	必須	デフォルト値	説明
stop-policy	いいえ	require-stop	指定できる値は次のとおりです。 ■ ignore：グループを停止するときにメンバーの状態を無視します。Ping のデフォルト動作です。 ■ require-stop：（デフォルト、Ping を除く）グループを停止とみなすには、メンバーが停止する必要があります。
group-policy-level	いいえ	all	このメンバーがグループ ポリシーに参加するレベルを指定します。 fail-policy と cannot-start-policy を参照してください。 指定できる値は次のとおりです。 ■ all：このメンバーは、グループのポリシーで指定されているように、停止とシャットダウンを順序どおりにトリガーします。 ■ start-only：このメンバーは、（グループがすでに実行された後の失敗時にではなく）通常の開始時にだけ、シャットダウンだけ（グループのポリシーで指定されている場合）をトリガーします。

```
...
<managed-objects>
     <managed-object ...type="ordered-group"...>
           <ordered-group>

                [<member mo-ref="${agent.name}/managedobject_name"
                  start="start_order" stop="stop_order"
                  stop-policy="require-stop|ignore"
                  group-policy-level="all|start-only"/>]
                [...]
           </ordered-group>
      </managed-object>
       ...
</managed-objects>
...
```

redundancy-group 管理オブジェクト タイプ

ここでは、redundancy-group 管理オブジェクト type のサブ要素とサブ要素の属性をリストおよび説明します。

メモ ordered-group 管理オブジェクト属性の詳細については、420 ページの「managed-object 要素の属性」を参照してください。

redundancy-group のサブ要素

次に redundancy-group タイプのサブ要素を示します。

サブ要素	説明
redundancy-group	redundancy-group 管理オブジェクト type 固有の設定情報を含みます。

```
...
<managed-objects>
       <managed-object ...type="redundancy-group"...>
            <redundancy-group>
                 ...
            </redundancy-group>
       </managed-object>
       ...
```

```
   </managed-objects>
   ...
```

さらに、managed-object のサブ要素 time-rules（タイプ固有ではない）を使用して、管理リソース プロセスを開始または停止するための cron 形式のルールを定義できます。

サブ要素	説明
time-rules	設定された時間ルールが実行される前の管理オブジェクトの基本状態を記述する属性を含みます。詳細については、「管理オブジェクト time-rules 要素」を参照してください。

```
...
   <managed-objects>
          <managed-object ...type="redundancy-group"...>
               [<time-rules ... >
                     ...
               </time-rules>]
               <redundancy-group>
                     ...
               </redundancy-group>
          </managed-object>
           ...
    </managed-objects>
...
```

managed-object のサブ要素 control-overrides（タイプ固有ではない）を使用して、デフォルトのアクション ストラテジやアクション パラメータを上書きできます。

サブ要素	説明
control-overrides	デフォルトを上書きするために指定されるアクション ストラテジやアクション パラメータを含みます。詳細については、「管理オブジェクト control-overrides 要素」を参照してください。

```
...
   <managed-objects>
          <managed-object ...type="redundancy-group"...>
               [<control-overrides>
                     ...
               </control-overrides>]
               <redundancy-group>
                     ...
               </redundancy-group>
          </managed-object>
           ...
    </managed-objects>
...
```

redundancy-group サブ要素の属性

次に redundancy-group サブ要素の属性を示します。

属性	必須	デフォルト値	説明
require	いいえ	1	グループを"起動"とみなすために実行されている必要があるメンバーの実際の数。グループ内のすべてのメンバーに設定するには、 -1 を入力します。
desired-range-min	いいえ	1	グループの実行時に起動して実行するように要求するメンバーの最小数。グループ内のすべてのメンバーに設定するには、 -1 を入力します。
desired-range-max	いいえ	1	起動して実行するように要求するメンバーの最大数。実行中のメンバーが多すぎる場合、 AppServer は、この値に合わせてメンバーをシャットダウンしようとします。この値は、グループ内のメンバーの総数を超えることはできません。すべてのメンバーに設定するには、 -1 を入力します。
count-member-unknown-as-stopped	いいえ	false	開始するメンバーの数を計算する際、 AppServer は、 desired-range-min 値から実行中のメンバーを引きます。デフォルトでは、未知の状態のメンバーが実行中のメンバーの数に含まれています。 未知の状態のメンバーを実行中のメンバーの数に入れない（停止とみなす）場合は、 count-member-unknown-as-stopped を true に設定します。
count-member-problem-as-stopped	いいえ	false	開始するメンバーの数を計算する際、 AppServer は、 desired-range-min 値から実行中のメンバーを引きます。デフォルトでは、状態が " 動作不良 " に等しいメンバーは実行中のメンバーの数に含まれません（停止とみなされる）。状態が"動作不良"に等しいメンバーを実行中のメンバーの数に入れる場合は、 count-member-problem-as-stopped を true に設定します。
excess-running-ok	いいえ	true	デフォルトでは、実行中のメンバーの数が desired-range-max 値を超える場合、グループは実行中とみなされます。実行中のメンバーの数が desired-range-max 値を超える場合、グループを動作不良とみなすには、 excess-running-ok を false に設定します。
clear-member-start-failures	いいえ	false	必要な数のメンバーが実行状態になると、その他のメンバーの失敗状態がクリアされます（たとえば、 Cannot Start が Stopped に変更される）。これにより、その他のメンバーの元の状態が Cannot Start であっても、フェイルオーバーに使用できるようになります。デフォルト値は false です。

```
...
<managed-objects>
       <managed-object ...type="redundancy-group"...>
            <redundancy-group [require="integer"
              desired-range-min="integer" desired-range-max="integer"
              count-member-unknown-as-stopped="true|false"
              count-member-problem-as-stopped="true|false"
              excess-running-ok="true|false"
              clear-member-start-failures="true|false"]>
                 ...
            </redundancy-group>
       </managed-object>
        ...
</managed-objects>
...
```

member サブ要素

サブ要素	必須	デフォルト値	説明
member	いいえ	該当なし	グループの各管理オブジェクト メンバーを指定します。

```
...
<managed-objects>
      <managed-object ...type="redundancy-group"...>
             <redundancy-group>
                   [<member attributes/>]
                   [<member attributes/>]
                   [<member attributes/>]
                   [...]
             </redundancy-group>
      </managed-object>
      ...
</managed-objects>
...
```

member サブ要素の属性

次に member サブ要素の属性を示します。

属性	必須	デフォルト値	説明
mo-ref	はい	該当なし	参照される管理オブジェクトの論理 AppServer 名。構文 ${*agent-name*}/*mo-name* に従います。
stop-policy	いいえ	require-stop	指定できる値は次のとおりです。 ■ ignore：グループを停止するときにメンバーの状態を無視します。これは Ping のデフォルト動作です。 ■ require-stop：（デフォルト、ping を除く）グループを停止とみなすには、メンバーが停止する必要があります。

```
...
<managed-objects>
      <managed-object ...type="redundancy-group"...>
            <redundancy-group>
                 [<member
                   mo-ref="${agent1_name}/managedobject1-name"
                   stop-policy="ignore|require-stop"/>]
                 [<member
                   mo-ref="${agent2_name}/managedobject2-name"
                   stop-policy="ignore|require-stop"/gt;]
                  [...]
            </redundancy-group>
      </managed-object>
       ...
```

```
</managed-objects>
...
```

state-proxy 管理オブジェクト タイプ

ここでは、state-proxy 管理オブジェクト type のサブ要素と属性をリストおよび説明します。

メモ state-proxy 管理オブジェクト属性の詳細については、420 ページの「managed-object 要素の属性」を参照してください。

state-proxy のサブ要素

次に state-proxy タイプのサブ要素を示します。

サブ要素	説明
state-proxy	state-proxy 管理オブジェクト type 固有の設定情報を含みます。

```
...
<managed-objects>
       <managed-object ...type="state-proxy"...>
             <state-proxy>
                   ...
             </state-proxy>
       </managed-object>
       ...
</managed-objects>
...
```

さらに、managed-object のサブ要素 time-rules（タイプ固有ではない）を使用して、管理リソース プロセスを開始または停止するための cron 形式のルールを定義できます。

サブ要素	説明
time-rules	設定された時間ルールが実行される前の管理オブジェクトの基本状態を記述する属性を含みます。詳細については、「管理オブジェクト time-rules 要素」を参照してください。

```
...
   <managed-objects>
          <managed-object ...type="state-proxy"...>
               [<time-rules ... >
                     ...
               </time-rules>]
               <state-proxy>
                     ...
               </state-proxy>
          </managed-object>
           ...
    </managed-objects>
...
```

managed-object のサブ要素 control-overrides （タイプ固有ではない）を使用して、デフォルトのアクション ストラテジやアクション パラメータを上書きできます。

サブ要素	説明
control-overrides	デフォルトを上書きするために指定されるアクション ストラテジやアクション パラメータを含みます。詳細については、「管理オブジェクト control-overrides 要素」を参照してください。

```
...
   <managed-objects>
          <managed-object ...type="state-proxy"...>
               [<control-overrides>
```

```
                    ...
                </control-overrides>]
                <state-proxy>
                    ...
                </state-proxy>
            </managed-object>
            ...
    </managed-objects>
...
```

state-proxy サブ要素の属性

次に state-proxy サブ要素の属性を示します。

属性	必須	デフォルト値	説明
mo-ref	はい	該当なし	参照される管理オブジェクトの論理 AppServer 名。構文 ${*agent-name*}/*mo-name* に従います。

```
...
<managed-objects>
      <managed-object ...type="state-proxy"...>
            [<state-proxy> mo-ref="${agent_name}/managedobject1-name"/>]
      </managed-object>
       ...
</managed-objects>
...
```

process 管理オブジェクト タイプ

ここでは、process 管理オブジェクト type のサブ要素と属性をリストおよび説明します。

メモ process 管理オブジェクト属性の詳細については、420 ページの「managed-object 要素の属性」を参照してください。

process のサブ要素

次に process 管理オブジェクト タイプのサブ要素を示します。

サブ要素	説明
process	process 管理オブジェクトタイプ固有の設定情報を含みます。

```
...
<managed-objects>
      <managed-object ...type="process"...>
            <process ...attributes...>
                ...
            </process>
             ...
      </managed-object>
</managed-objects>
...
```

さらに、managed-object のサブ要素 time-rules（タイプ固有ではない）を使用して、管理リソース プロセスを開始または停止するための cron 形式のルールを定義できます。

サブ要素	説明
time-rules	設定された時間ルールが実行される前の管理オブジェクトの基本状態を記述する属性を含みます。詳細については、「管理オブジェクト time-rules 要素」を参照してください。

```
...
   <managed-objects>
          <managed-object ...type="process"...>
               [<time-rules ... >
                     ...
               </time-rules>]
               <process>
                     ...
               <process>
          </managed-object>
           ...
    </managed-objects>
...
```

サブ要素	説明
control-overrides	デフォルトを上書きするために指定されるアクション ストラテジやアクション パラメータを含みます。詳細については、「管理オブジェクト control-overrides 要素」を参照してください。

```
...
   <managed-objects>
          <managed-object ...type="process"...>
                     ...
               [<control-overrides>
                     ...
               </control-overrides>]
          </managed-object>
           ...
    </managed-objects>
...
```

process サブ要素の属性

次に process サブ要素の属性を示します。

属性	必須	デフォルト値	説明
command	はい	該当なし	プログラム ファイルのフル パス。Windows ユーザーは拡張子を省略できます。その場合は、".exe" とみなされます。
directory	いいえ	${CONFIG.path}/mos/${MO.name}/ - ここで、置換文字列変数 ${CONFIG.path} は、管理オブジェクト configuration.xml ファイルのパスを表します。	プロセスの初期作業ディレクトリのフル パス。
data-id	いいえ	1	process 要素の一意の識別子。同じ管理オブジェクト定義に含まれる複数の process エントリを識別するために AppServer によって使用されます。指定できる値は正の整数です。

```
...
<managed-objects>
      <managed-object ...type="process"...>
           <process command="fullpath" directory="fullpathdirectory>
             data-id="integer">
            ...
           </process>
      </managed-object>
</managed-objects>
...
```

process のサブ要素

次に java-process サブ要素の要素を示します。

arguments サブ要素

サブ要素	必須	デフォルト値	説明
arguments	いいえ	該当なし	プロセスの開始時に参照されるコマンドライン引数を提供するために使用します。入力する各コマンドライン引数は、argument サブ要素内に記述されます。引数は、configuration.xml にリストされている順序でコマンドライン文字列に追加されます。 **メモ：**各 argument サブ要素に複数のコマンドライン引数を入れることはできません。たとえば、-user Joe は 2 つの引数なので、<argument>-user</argument> <argument>Joe</argument> のように argument 要素に入力する必要があります。

```
...
<managed-objects>
      <managed-object ...type="process"...>
           <process ... >
              [<arguments>
                  [<argument>argument1</argument>]
                  [<argument>argument2</argument>]
                  [...more arguments...]
              </arguments>]
           </process>
      </managed-object>
</managed-objects>
...
```

env-vars サブ要素

サブ要素	必須	デフォルト値	説明
env-vars	いいえ	path、library-path、env-vars の各属性が設定されていない場合は、エージェント SCU 環境を使用します。	プロセスの環境変数を設定するために使用します。設定する変数を name 属性と value 属性で定義する env-var 要素のシーケンスを入力します。 さらに、env-vars 要素の次の属性を使用して、標準の環境変数ソースを指定できます。これにより、複数の管理リソースで同じ環境変数を使用できます。 **メモ：**use-vbroker-env で同じ環境変数が設定されている場合は、env-vars 要素を使って設定された変数が優先します。

env-vars サブ要素の属性

属性	必須	デフォルト値	説明
use-current-env	いいえ	false	プロセスでエージェント SCU プロセス環境全体を使用する場合は、true に設定します。
use-default-env	いいえ	true	プロセスで、プロセスの実行に必要な共通ホストシステム環境変数を使用しない場合は、false に設定します。 **メモ：**use-user-login-env で同じ変数が設定されている場合は、use-default-env が優先します。
use-vbroker-env	いいえ	false	VisiBroker プロセスの場合は true に設定します。 **メモ:**use-default-env で同じ変数が設定されている場合は、use-vbroker-env 変数が優先します。
use-user-login-env	いいえ	false	UNIX のみ。ユーザーがデフォルトのシェルにログインしたとき、プロセスが管理オブジェクトプロセスの所有者の環境を使用するようにする場合は、true に設定します。 **メモ:**use-current-env で同じ変数が設定されている場合は、use-user-login-env が優先します。ログイン シェルは、シェルがログイン シェルとして呼び出された場合に、コマンドライン引数をシェル スクリプトとして実行する必要があります。また、コマンドラインの内容としてプログラム ファイルの絶対パスだけが指定されている場合、シェルのスクリプト言語は子プロセス（通常の KornShell、C Shell などのセマンティクス）を呼び出す必要があります。

```
...
<managed-objects>
      <managed-object ...type="process"...>
           <process ... >
              [<env-vars use-current-env="true|false" use-default-env="true|
false" use-vbroker-env="true|false" use-login-env="true|false">]
                   [<env-var name="variable1"
                   value="value1" />]
                   [<env-var name="variable2"
                   value="value2" />]
                   [...more env-vars...]
              </env-vars>]
           </process>
      </managed-object>
</managed-objects>
...
```

library-path サブ要素

サブ要素	必須	デフォルト値	説明
library-path	いいえ	なし	UNIX のみ。プロセスのライブラリ パスに項目を配置するために使用します。入力する各パスは、configuration.xml の library-path 要素の directory サブ要素内に記述されます。 **メモ：**env-vars 要素を使って同じ環境変数が設定されている場合は、library-path の設定が優先します。

```
...
<managed-objects>
      <managed-object ...type="process"...>
            <process ... >
                  [<library-path>
                   <directory>path1</directory>
                  [<directory>path2</directory>]
                  [...more paths...]
                </library-path>]
            </process>
      </managed-object>
</managed-objects>
...
```

path サブ要素

サブ要素	必須	デフォルト値	説明
path	いいえ	該当なし	非プラットフォーム固有。プロセス実行可能ファイルのパスに項目を配置するために使用します。入力する各パスは、configuration.xml の path 要素の directory サブ要素内に記述されます。 メモ：env-vars 要素を使って同じ環境変数が設定されている場合は、path の設定が優先します。

```
...
<managed-objects>
      <managed-object ...type="process"...>
            <process ... >
                  [<path>
                   <directory>path1</directory>
                  [<directory>path2</directory>]
                  [...more paths...]
                </path>]
            </process>
      </managed-object>
</managed-objects>
...
```

stdin|stdout|stderr のサブ要素

サブ要素	必須	デフォルト値	説明
stdin	いいえ	割り当てられたエージェントから継承。	プロセスの標準入力設定を指定するために使用します。
stdout	いいえ	割り当てられたエージェントから継承。	プロセスの標準出力設定を指定するために使用します。
stderr	いいえ	割り当てられたエージェントから継承。	プロセスの標準エラー設定を指定するために使用します。

stdin|stdout|stderr サブ要素の属性

属性	必須	デフォルト値	説明
path	いいえ	割り当てられたエージェントから継承。	標準入力、標準出力、または標準エラーのプロセス パス位置を設定するために使用します。
append	いいえ	false	stdout と stderr に適用されます。ファイルを追加モードで開く場合は、true に設定します。false に設定すると、ファイルが切り捨てられます。

```
...
<managed-objects>
      <managed-object ....type="process"...>
           <process ... >
               [<stdin path="path1/>]
               [<stdout path="path2"  append="true|false"/>]
               [<stderr path="path3" append="true|false"/>]
           </process>
      </managed-object>
</managed-objects>
...
```

プラットフォーム固有のサブ要素

サブ要素	必須	デフォルト値	説明
platform-specific (Windows)	いいえ	属性： show-gui="false" start-minimized="true" window-title：実行可能ファイルの名前。	GUI を備えたプロセスに Windows プラットフォーム固有の設定を指定するために使用します。この要素の Windows の属性は次のとおりです。 show-gui：プロセスの開始時にプロセス ウィンドウを起動する場合は、true に設定します。 window-title：プロセス ウィンドウのタイトルを指定します。 start-minimized：プロセスの開始後にウィンドウを最大化する場合は、false に設定します。
platform-specific（UNIX）	いいえ	属性： group-name="*agentgroupname*" user-name="*agentusername*" stdout-mode="644" stderr-mode="644" umask="*agentumask*" nice-value="*agentnicevalue*" root-directory="*agentrootdirectory*"	UNIX プラットフォーム固有の設定を指定するために使用します。この要素の UNIX の属性は次のとおりです。 group-name：プロセスのグループ名。設定方法については、次のとおりです。 user-name：プロセスのユーザー名。 stdout-protection-mode：stdoutput ファイルの 8 進法ファイル アクセス モード。 stderr-protection-mode：stderrput ファイルの 8 進法ファイル アクセス モード。 umask：プロセスの 8 進 umask。 nice-value：プロセスのスケジュール優先順位（正の整数）。 root-directory：プロセスのルート ディレクトリ。

重要 UNIX プラットフォーム固有の設定は、エージェント SCU プロセスがルート特権で開始する場合にだけ指定できます。

Windows

```
...
<managed-objects>
      <managed-object ...type="process"...>
            <process ... >
                  [<platform-specific show-gui="true|false"
                     window-title="string" start-minimized="true|false"/>
            </process>
      </managed-object>
</managed-objects>
...
```

UNIX

```
...
<managed-objects>
      <managed-object ...type="process"...>
            <process ... >
                  [<platform-specific group-name="groupname"
                     user-name="username" stdout-mode="octalfileaccessmode"
                     stderr-mode="octalfileaccessmode" umask="octalumask"
                     nice-value="integer"/>]
            </process>
      </managed-object>
</managed-objects>
...
```

java-process 管理オブジェクト タイプ

ここでは、java-process 管理オブジェクト type のサブ要素と属性をリストおよび説明します。

メモ　java-process 管理オブジェクト属性の詳細については、420 ページの「managed-object 要素の属性」を参照してください。

java-process のサブ要素

java-process 管理オブジェクト固有の設定情報は、次の java-process のサブ要素に含まれています。

- java-process
- VBJ-process

次に java-process タイプのサブ要素を示します。

サブ要素	説明
java-process	main() メソッドが呼び出される Java クラスに使用します。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
            <java-process ...attributes...>
             ...
            </java-process>
      </managed-object>
</managed-objects>
...
```

サブ要素	説明
VBJ-process	VisiBroker for Java プロセス（java ではなく vbj で開始されるプロセス）の定義に使用します。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
            <VBJ-process ...attributes...>
             ...
            </VBJ-process>
      </managed-object>
</managed-objects>
...
```

さらに、managed-object のサブ要素 time-rules（タイプ固有ではない）を使用して、管理リソース プロセスを開始または停止するための cron 形式のルールを定義できます。

サブ要素	説明
time-rules	設定された時間ルールが実行される前の管理オブジェクトの基本状態を記述する属性を含みます。詳細については、「管理オブジェクト time-rules 要素」を参照してください。

```
...
   <managed-objects>
         <managed-object ...type="java-process"...>
               [<time-rules ... >
                     ...
               </time-rules>]
               <java-process>
                     ...
               </java-process>
         </managed-object>
          ...
```

```
    </managed-objects>
...
```

managed-object のサブ要素 control-overrides（タイプ固有ではない）を使用して、デフォルトのアクション ストラテジやアクション パラメータを上書きできます。

サブ要素	説明
control-overrides	デフォルトを上書きするために指定されるアクション ストラテジやアクション パラメータを含みます。詳細については、「管理オブジェクト control-overrides 要素」を参照してください。

```
...
   <managed-objects>
         <managed-object ...type="java-process"...>
                ...
          [<control-overrides>
                ...
          </control-overrides>]
         </managed-object>
          ...
    </managed-objects>
...
```

java-process サブ要素の属性

次に java-process サブ要素の属性を示します。

属性	必須	デフォルト値	説明
command	はい	該当なし	Java プロセス プログラム ファイルのフルパス。Windows ユーザーは拡張子を省略できます。 その場合は、 ".exe" とみなされます。
main-class	はい	該当なし	*MyPackage.MyClass* の形式の Java プロセス管理リソースのメインクラス。
directory	いいえ	${AGENT.root}/var/domains/*domain-name*/。ここで、置換文字列変数 ${AGENT.root} は、この管理オブジェクトの割り当て先のエージェントのルートを表します。	プロセスの初期作業ディレクトリのフルパス。
vm-type	いいえ	該当なし	次の値を指定できます。 ■ classic ■ hotspot ■ client ■ server
data-id	いいえ	1	java-process 要素の一意の識別子。同じ管理オブジェクト定義に含まれる複数の java-process エントリを識別するために AppServer によって使用されます。指定できる値は正の整数です。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
           <process command="fullpath" main-class="MyPackage.MyClass"
             directory="fullpathdirectory"
             vm-type="server|client|classic"
             data-id="integer">
            ...
           </java-process>
```

```
      </managed-object>
</managed-objects>
...
```

java-process のサブ要素

次に java-process サブ要素の要素を示します。

サブ要素	必須	デフォルト値	説明
java-properties	いいえ	該当なし	管理オブジェクトの VM プロパティを提供するために使用します。設定するプロパティごとに name 属性と value 属性を含む java-property 要素のシーケンスを入力します (*name* に -D を入れないでください)。結果は java -D コマンドライン構文になります。 **メモ:** vbj-process タイプ管理オブジェクトの場合は、VBJ -VBJprop コマンドライン構文になります。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
           <java-process ... >
              [<java-properties>
                    [<java-property name="name1" value="value1" />]
                    [<java-property name="name2" value="value2"  />]
                    [...more_properties...]
               </java-properties>]
           </java-process>
      </managed-object>
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
classpath	いいえ	該当なし	java -classpath のクラスパス エントリを指定するために使用します。各クラスパス エントリは、classpath 要素の classpath-entry サブ要素内に記述されます。 **メモ:** vbj-process タイプ管理オブジェクトの場合は、かわりに -vbj classpath を使用することをお勧めします。詳細については、vbj-process のサブ要素の表を参照してください。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
           <java-process ...attributes...>
               [<classpath>
                   [<classpath-entry>classpath1</classpath-entry>[
                   [<classpath-entry>classpath2</classpath-entry>]
                   [...more classpaths...]
               </classpath>]
               [...]
           </java-process>
      </managed-object>
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
options	いいえ	該当なし	コマンドラインでメイン クラスより優先するオプションを指定するために使用します。各オプションは、options 要素の option サブ要素内に記述する必要があります。 **メモ :** 各 option サブ要素に複数のオプションを入れることはできません。たとえば、-Xms -Xmx は 2 つのオプションなので、<option>-Xms</option> <option>-Xmx</option> のように option 要素に入力する必要があります。

```
...
<managed-objects>
      <managed-object ...type="process"...>
           <java-process ...attributes...>
              [<options>
                  [<option>option1</option>]
                  [<option>option2</option>]
                  [...more options...]
              </options>]
           </java-process>
      </managed-object>
</managed-objects>
...
```

arguments サブ要素

このサブ要素については、434 ページの「arguments サブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
           <java-process ... >
              [<arguments>
                  [<argument>argument1</argument>]
                  [<argument>argument2</argument>]
                  [...more arguments...]
              </arguments>]
           </java-process>
      </managed-object>
</managed-objects>
...
```

env-vars サブ要素

このサブ要素については、434 ページの「env-vars サブ要素」を参照してください。

env-vars サブ要素の属性

このサブ要素については、435 ページの「env-vars サブ要素の属性」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
           <java-process ... >
              [<env-vars use-current-env="true|false" use-default-env="true|
                false" use-vbroker-env="true|false" use-login-env="true|false">
                 <env-var name="variable1"
                    value="value1" />
                 [<env-var name="variable2"
                    value="value2" />]
                 [...more env-vars...]
              </env-vars>]
           </java-process>
      </managed-object>
```

```
</managed-objects>
...
```

library-path サブ要素

このサブ要素については、436 ページの「library-path サブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
           <java-process ... >
                 [<library-path>
                  <directory>path1</directory>
                 [<directory>path2</directory>]
                 [...more paths...]
               </library-path>]
           </java-process>
      </managed-object>
</managed-objects>
...
```

path サブ要素

このサブ要素については、436 ページの「path サブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
           <java-process ... >
                 [<path>
                  <directory>path1</directory>
                 [<directory>path2</directory>]
                 [...more paths...]
               </path>]
           </java-process>
      </managed-object>
</managed-objects>
...
```

stdin|stdout|stderr のサブ要素

このサブ要素については、437 ページの「stdin|stdout|stderr のサブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ....type="java-process"...>
           <java-process ... >
               [<stdin path="path1"/>]
               [<stdout path="path2"
                append="true|false"/>]
               [<stderr path="path3"
                append="true|false"/>]
           </java-process>
      </managed-object>
</managed-objects>
...
```

platform-specific のサブ要素

このサブ要素については、438 ページの「プラットフォーム固有のサブ要素」を参照してください。

Windows

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
            <java-process ... >
                  [<platform-specific show-gui="true|false"
                     window-title="string" start-minimized="true|false"/>
            </java-process>
      </managed-object>
</managed-objects>
...
```

UNIX

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
            <java-process ... >
                  [<platform-specific group-name="groupname"
                     user-name="username" stdout-mode="octalfileaccessmode"
                     stderr-mode="octalfileaccessmode" umask="octalumask"
                     nice-value="integer"/>]
            </java-process>
      </managed-object>
</managed-objects>
...
```

VBJ-process サブ要素の属性

vbj-process 要素は、java-process 要素のすべての属性を継承します。詳細については、440 ページの「java-process サブ要素の属性」を参照してください。

VBJ-process 要素固有の追加属性はありません。

VBJ-process のサブ要素

次に VBJ-process 要素固有のサブ要素を示します。

サブ要素	必須	デフォルト値	説明
vbj-java-options	いいえ	該当なし	vbj -J コマンドライン構文のオプションを追加するために使用します。-J を入れないでください。各オプションは、vbj-java-options 要素の option サブ要素内に記述する必要があります。 **メモ**：各 option サブ要素に複数のオプションを入れることはできません。たとえば、-Xms -Xmx は 2 つのオプションなので、<option>-Xms</option> <option>-Xmx</option> のように option 要素に入力する必要があります。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
            <VBJ-process ... >
               [<vbj-java-options>]
                  [<option>option1</option>]
                  [<option>option2</option>]
                  [...more options...]
               </vbj-java-options>]
            </VBJ-process>
      </managed-object>
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
vbj-classpath	いいえ	該当なし	vbj -VBJClasspath クラスパスのクラスパス エントリを指定するために使用します。入力する各クラスパス エントリは、vbj-classpath 要素の classpath-entry サブ要素内に記述されます。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
           <VBJ-process ... >
              [<vbj-classpath>
                  [<classpath-entry>classpath1</classpath-entry>]
                  [<classpath-entry>classpath2</classpath-entry>]
                  [...more classpaths...]
              </vbj-classpath>]
           </VBJ-process>
      </managed-object>
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
vbj-add-jars	いいえ	該当なし	vbj -VBJaddJar コマンドライン オプションとともに使用する JAR のリストを提供するために使用します。コマンドラインに追加する各 JAR は、vbj-add-jars 要素の add-jars-entry サブ要素内に記述されます。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
           <VBJ-process ... >
              [<vbj-add-jars>
                  [<add-jars-entry>jar1</add-jars-entry>]
                  [<add-jars-entry>jar2</add-jars-entry>]
                  [...more JARs...]
              </vbj-add-jars>]
           </VBJ-process>
      </managed-object>
</managed-objects>
...
```

このサブ要素については、441 ページの「java-process のサブ要素」を参照してください。

メモ vbj-process 管理オブジェクトの場合は、VBJ -VBJprop コマンドライン構文になります。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
           <VBJ-process ... >
              [<java-properties>
                    [<java-property name="name1" value="value1" />]
                    [<java-property name="name2" value="value2"  />]
                    [...more_properties...]
              </java-properties>]
           </VBJ-process>
      </managed-object>
</managed-objects>
...
```

このサブ要素については、441 ページの「java-process のサブ要素」を参照してください。

メモ VBJ-process 管理オブジェクトの場合は、かわりに -vbj classpath を使用することをお勧めします。詳細については、vbj-process のサブ要素の表を参照してください。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
           <VBJ-process ...attributes...>
               [<classpath>
                   [<classpath-entry>classpath1</classpath-entry>[
                   [<classpath-entry>classpath2</classpath-entry>]
                   [...more classpaths...]
               </classpath>]
               [...]
           </VBJ-process>
      </managed-object>
</managed-objects>
...
```

options サブ要素

このサブ要素については、441 ページの「java-process のサブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="process"...>
           <VBJ-process ...attributes...>
              [<options>
                  [<option>option1</option>]
                  [<option>option2</option>]
                  [...more options...]
              </options>]
           </VBJ-process>
      </managed-object>
</managed-objects>
...
```

arguments サブ要素

このサブ要素については、434 ページの「arguments サブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
           <java-process ... >
              [<arguments>
                  [<argument>argument1</argument>]
                  [<argument>argument2</argument>]
                  [...more arguments...]
              </arguments>]
           </java-process>
      </managed-object>
</managed-objects>
...
```

env-vars サブ要素

このサブ要素については、434 ページの「env-vars サブ要素」を参照してください。

env-vars サブ要素の属性

このサブ要素については、435 ページの「env-vars サブ要素の属性」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
           <VBJ-process ... >
              [<env-vars use-current-env="true|false" use-default-env="true|
                false" use-vbroker-env="true|false" use-login-env="true|false">
                 <env-var name="variable1"
                    value="value1" />
                 [<env-var name="variable2"
                    value="value2" />]
                 [...more env-vars...]
              </env-vars>]
           </VBJ-process>
      </managed-object>
</managed-objects>
...
```

library-path サブ要素

このサブ要素については、436 ページの「library-path サブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
           <VBJ-process ... >
                 [<library-path>
                  <directory>path1</directory>
                 [<directory>path2</directory>]
                 [...more paths...]
              </library-path>]
           </VBJ-process>
      </managed-object>
</managed-objects>
...
```

path サブ要素

このサブ要素については、436 ページの「path サブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
           <VBJ-process ... >
                 [<path>
                  <directory>path1</directory>
                 [<directory>path2</directory>]
                 [...more paths...]
              </path>]
           </VBJ-process>
      </managed-object>
</managed-objects>
...
```

stdin|stdout|stderr のサブ要素

このサブ要素については、437 ページの「stdin|stdout|stderr のサブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ....type="java-process"...>
           <VBJ-process ... >
               [<stdin path="path1"/>]
               [<stdout path="path2"
                append="true|false"/>]
               [<stderr path="path3"
                append="true|false"/>]
           </VBJ-process>
      </managed-object>
</managed-objects>
...
```

platform-specific のサブ要素

このサブ要素については、438 ページの「プラットフォーム固有のサブ要素」を参照してください。

Windows

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
           <VBJ-process ... >
                [<platform-specific show-gui="true|false"
                   window-title="string" start-minimized="true|false"/>
           </VBJ-process>
      </managed-object>
</managed-objects>
...
```

UNIX

```
...
<managed-objects>
      <managed-object ...type="java-process"...>
           <VBJ-process ... >
                [<platform-specific group-name="groupname"
                   user-name="username" stdout-mode="octalfileaccessmode"
                   stderr-mode="octalfileaccessmode" umask="octalumask"
                   nice-value="integer"/>]
           </VBJ-process>
      </managed-object>
</managed-objects>
...
```

custom-javascript 管理オブジェクト タイプ

ここでは、`custom-javascript` 管理オブジェクト `type` のサブ要素と属性をリストおよび説明します。

メモ `custom-javascript` 管理オブジェクト属性の詳細については、420 ページの「managed-object 要素の属性」を参照してください。

重要 （UNIX のみ）管理オブジェクトがマルチユーザー モード（MUM）のエージェントによって開始される場合は、通常、JavaScript を実行できません。マルチユーザー モード（MUM）で実行される管理オブジェクトのいずれかに JavaScript が含まれる場合は、`agent.config` を変更して、JavaScript の実行を許可する必要があります。

custom-javascript のサブ要素

次に `custom-javascript` タイプのサブ要素を示します。

要素	説明
jscript	custom-javascript タイプ管理オブジェクトに固有の設定情報を含みます。

```
...
<managed-objects>
       <managed-object ...type="custom-javascript".../>
             ...
             <jscript ...attributes...>
                   ...
             </jscript>
       </managed-object>
       ...
</managed-objects>
...
```

さらに、`managed-object` のサブ要素 `time-rules`（タイプ固有ではない）を使用して、管理リソース プロセスを開始または停止するための `cron` 形式のルールを定義できます。

サブ要素	説明
time-rules	設定された時間ルールが実行される前の管理オブジェクトの基本状態を記述する属性を含みます。詳細については、「管理オブジェクト time-rules 要素」を参照してください。

```
...
   <managed-objects>
          <managed-object ...type="custom-javascript"...>
               [<time-rules ... >
                     ...
               </time-rules>]
               <jscript>
                     ...
               </jscript>
          </managed-object>
           ...
    </managed-objects>
...
```

要素	説明
control-overrides	管理オブジェクトのアクション ストラテジとアクション パラメータ情報を含む必須要素。詳細については、「管理オブジェクト control-overrides 要素」を参照してください。

```
...
<managed-objects>
       <managed-object ...type="custom-javascript".../>
             <control-overrides>
                   ...
```

```
            </control-overrides>
            ...
        </managed-object>
        ...
</managed-objects>
...
```

control-overrides 要素のサブ要素

次に control-overides 要素のサブ要素を示します。

サブ要素	必須	デフォルト値	説明
start	はい（pinger 管理オブジェクトの場合は不要）	該当なし	開始アクション ストラテジを指定し、開始アクション固有の情報を含む jscript サブ要素を参照するために使用します。

```
...
<managed-objects>
        <managed-object ...type="custom-javascript".../>
            <control-overrides>
                <start ...attributes.../>
                ...
            </control-overrrides>
            <jscript ...attributes...>
                ...
            </jscript>
        </managed-object>
        ...
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
ping	はい	該当なし	ping アクション ストラテジを指定し、ping アクション固有の情報を含む jscript サブ要素を参照するために使用します。

```
...
<managed-objects>
        <managed-object ...type="custom-javascript".../>
            <control-overrides>
                ...
                <ping ...attributes.../>
                ...
            </control-overrrides>
            <jscript ...attributes...>
                ...
            </jscript>
        </managed-object>
        ...
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
stop	はい（pinger 管理オブジェクトの場合は不要）	該当なし	停止アクション ストラテジを指定し、停止アクション固有の情報を含む jscript サブ要素を参照するために使用します。

```
...
<managed-objects>
       <managed-object ...type="custom-javascript".../>
              <control-overrides>
                     ...
                     <stop ...attributes.../>
                     ...
              </control-overrrides>
              <jscript ...attributes...>
                     ...
              </jscript>
       </managed-object>
       ...
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
kill	いいえ	該当なし	強制終了アクション ストラテジを指定し、 強制終了アクション固有の情報を含む jscript サブ要素を参照するために使用します。

```
...
<managed-objects>
       <managed-object ...type="custom-javascript".../>
              <control-overrides>
                     ...
                     <kill ...attributes.../>
                     ...
              </control-overrrides>
              <jscript ...attributes...>
                     ...
              </jscript>
       </managed-object>
       ...
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
parameters	いいえ	該当なし	アクション ストラテジの追加パラメータを指定します。 詳細については、「管理オブジェクト control-overrides 要素」、「parameters サブ要素の属性」を参照してください。

```
...
<managed-objects>
       <managed-object ...type="custom-javascript".../>
              <control-overrides>
                     <start ...attributes.../>
                     <ping ...attributes.../>
                     <stop ...attributes.../>
                     <kill ...attributes.../>
                     <parameters ...attributes.../>
              </control-overrrides>
              <jscript ...attributes...>
                     ...
              </jscript>
       </managed-object>
       ...
</managed-objects>
...
```

start サブ要素の属性

次に start サブ要素の属性を示します。

属性	必須	デフォルト値	説明
strategy	はい	jscript-control	管理リソースを開始する JavaScript を実行します。このアクションを無効にするには、null に設定します。
data-id-ref	いいえ	1	jscript 開始アクション ストラテジの一意の識別子。 開始アクションに対して実行する JavaScript への参照を含む jscript サブ要素を参照するために AppServer によって使用されます。指定できる値は正の整数です。

```
...
<managed-objects>
       <managed-object ...type="custom-javascript".../>
             <control-overrides>
                    <start strategy="jscript-control|null"
                        [data-id-ref="integer"]/>
                    <ping ...attributes.../>
                    <stop ...attributes.../>
                    <kill ...attributes.../>
                    <parameters ...attributes.../>
             </control-overrrides>
             <jscript ...attributes...>
                   ...
             </jscript>
       </managed-object>
       ...
</managed-objects>
...
```

ping サブ要素の属性

次に ping サブ要素の属性を示します。

属性	必須	デフォルト値	説明
strategy	はい	jscript-ping	管理リソースを ping する JavaScript を実行します。このストラテジを無効にするには、null に設定します。
data-id-ref	いいえ	2	jscript ping アクション ストラテジの一意の識別子。ping アクションに対して実行する JavaScript への参照を含む jscript サブ要素を参照するために AppServer によって使用されます。指定できる値は正の整数です。

```
...
<managed-objects>
       <managed-object ...type="custom-javascript".../>
             <control-overrides>
                <start strategy="jscript-control|null"[data-id-ref="integer"]/>
                <ping strategy="jscript-ping|null" [data-id-ref="integer"]/>
                <stop ...attributes.../>
                <kill ...attributes.../>
                <parameters ...attributes.../>
             </control-overrrides>
             <jscript ...attributes...>
                   ...
             </jscript>
       </managed-object>
       ...
</managed-objects>
...
```

stop サブ要素の属性

次に stop サブ要素の属性を示します。

属性	必須	デフォルト値	説明
strategy	はい	jscript-control	管理リソースを停止する JavaScript を実行します。このストラテジを無効にするには、null に設定します。
data-id-ref	いいえ	3	jscript 停止アクション ストラテジの一意の識別子。停止に対して実行する JavaScript への参照を含む jscript サブ要素を参照するために AppServer によって使用されます。指定できる値は正の整数です。

```
...
<managed-objects>
       <managed-object ...type="custom-javascript".../>
             <control-overrides>
                <start strategy="jscript-control|null"[data-id-ref="integer"]/>
                <ping strategy="jscript-ping|null" [data-id-ref="integer"]/>
                <stop strategy="jscript-control|null" [data-id-ref="integer"]/>
                <kill ...attributes.../>
                <parameters ...attributes.../>
             </control-overrrides>
             <jscript ...attributes...>
                  ...
             </jscript>
       </managed-object>
       ...
</managed-objects>
...
```

kill サブ要素の属性

次に kill サブ要素の属性を示します。

属性	必須	デフォルト値	説明
strategy	はい	jscript-control	管理リソースを強制終了する JavaScript を実行します。このストラテジを無効にするには、null に設定します。
data-id-ref	いいえ	4	jscript 強制終了アクション ストラテジの一意の識別子。 強制終了に対して実行する JavaScript への参照を含む jscript サブ要素を参照するために AppServer によって使用されます。指定できる値は正の整数です。

```
...
<managed-objects>
       <managed-object ...type="custom-javascript".../>
             <control-overrides>
                <start strategy="jscript-control|null"[data-id-ref="integer"]/>
                <ping strategy="jscript-ping|null" [data-id-ref="integer"]/>
                <stop strategy="jscript-control|null" [data-id-ref="integer"]/>
                <kill strategy="jscript-control|null" [data-id-ref="integer"]/>
                <parameters ...attributes.../>
             </control-overrrides>
             <jscript ...attributes...>
                  ...
             </jscript>
       </managed-object>
       ...
</managed-objects>
...
```

jscript サブ要素の属性

次に jscript サブ要素の属性を示します。

属性	必須	デフォルト値	説明
data-id	はい	1	jscript 要素の一意の識別子。同じ custom-javascript 管理オブジェクト定義に含まれる複数の jscript エントリを識別するために AppServer によって使用されます。指定できる値は正の整数です。

```
...
<managed-objects>
       <managed-object ...type="custom-javascript".../>
             <jscript data-id="integer1">
                   ...
             </jscript>
             [<jscript data-id="integer2">
                   ...
             </jscript>]
             [<jscript data-id="integer3">
                   ...
             </jscript>]
       </managed-object>
       ...
</managed-objects>
...
```

jscript のサブ要素

次に jscript のサブ要素を示します。

サブ要素	説明
run	実行する JavaScript へのリファレンスを含みます。これらの JavaScript は、configuration.xml にリストされる順序で実行されます。各 run に入れることができるスクリプトは 1 つだけです。複数の run 要素を定義できます。それぞれが独自の JavaScript を含み、管理オブジェクトに対する arguments が定義されます。

```
...
<managed-objects>
       <managed-object ...type="custom-javascript".../>
             <jscript data-id="integer1">
                   <run ...attributes...>
                         ...
                   </run>
             </jscript>
             [<jscript data-id="integer2">
                   <run ...attributes...>
                         ...
                   </run>
             </jscript>]
       </managed-object>
       ...
</managed-objects>
...
```

サブ要素	説明
classpath	このサブ要素については、「java-process のサブ要素」を参照してください。

```
...
<managed-objects>
       <managed-object ...type="custom-javascript".../>
             <jscript ...attributes... >
```

```
                [<classpath>
                    [<classpath-entry>classpath1</classpath-entry>[
                    [<classpath-entry>classpath2</classpath-entry>]
                    [...more classpaths...]
                </classpath>]
                  ...
              </jscript>
              [<jscript ...attributes... >
                <classpath>
                    [<classpath-entry>classpath1</classpath-entry>[
                    [<classpath-entry>classpath2</classpath-entry>]
                    [...more classpaths...]
                </classpath>]
                  ...
              </jscript>]
        </managed-object>
        ...
  </managed-objects>
  ...
```

run サブ要素の属性

次に run サブ要素で使用できる属性を示します。

属性	必須	デフォルト値	説明
script	はい	該当なし	JavaScript ファイルの絶対パス。JavaScript を埋め込む場合は、この属性を**使用しない**でください。
name	いいえ	script*n*	管理コンソールとログで表示の目的にだけ使用される JavaScript の名前。 参照には使用できません。

```
  ...
  <managed-objects>
        <managed-object ...type="custom-javascript".../>
              <jscript data-id="integer1">
                    <run script="fullpathfilename" [name="scriptn"|string1"]>
                          ...
                    </run>
                    ...
              </jscript>
              [<jscript data-id="integer2">
                    <run script="fullpathfilename" [name="scriptn+1"|string2"]>
                          ...
                    </run>
                    ...
              </jscript>]
        </managed-object>
        ...
  </managed-objects>
  ...
```

run サブ要素

次に run のサブ要素を示します。

サブ要素	必須	デフォルト値	説明
arguments	いいえ	該当なし	この要素の詳細については、「process のサブ要素」を参照してください。

```
  ...
  <managed-objects>
        <managed-object ...type="custom-javascript".../>
              <jscript data-id="integer1">
                    <run script="fullpath_filename" [name="scriptn"|string1"]>
                          [<arguments>
```

```
                        [<argument>argument1</argument>]
                        [<argument>argument2</argument>]
                      </arguments>]
                </run>
            </jscript>
            [<jscript data-id="integer2">
                <run script="fullpath_filename" [name="scriptn+1"|string2"]>
                      [<arguments>
                        [<argument>argument1</argument>]
                        [<argument>argument2</argument>]
                      </arguments>]
                </run>
            </jscript>]
        </managed-object>
        ...
</managed-objects>
...
```

![CDATA[セクション

JavaScript を XML に埋め込むには、`run` サブ要素の `<![CDATA[` セクションを使用します。

重要 AppServer は、埋め込まれた JavaScipt の次の項目を**サポートしません**。

- C++ 形式のコメント
- `try {`
  `} catch(Exception` *name*`) {`
- `return` ステートメント。かわりに `quit(`*returncode*`)` を使用することをお勧めします。

![CDATA[属性

`![CDATA[` セクションでは、`arguments` 要素を使ってコマンドライン引数を指定できます。この要素の詳細については、434 ページの「process のサブ要素」を参照してください。

```
...
<managed-objects>
        <managed-object ...type="custom-javascript".../>
            <jscript data-id="integer1">
                <run script="fullpathfilename" [name="scriptn"|string1"]>
                    [<arguments>
                        [<argument>argument1</argument>]
                        [<argument>argument2</argument>]
                    </arguments>]
                </run>
            </jscript>
            <jscript data-id="integer2">
                <run name="string" [name="scriptn+1"|string1"]>
                    <![CDATA[
                        an-embedded-javascript
                     [<arguments>
                        [<argument>argument1</argument>]
                        [<argument>argument2</argument>]
                        [...more arguments...]
                      </arguments>]
                    ]]>
                </run>
              ...
            </jscript>
        </managed-object>
        ...
</managed-objects>
...
```

custom-executable 管理オブジェクト タイプ

ここでは、custom-executable 管理オブジェクト type のサブ要素と属性をリストおよび説明します。

メモ　custom-executable 管理オブジェクト属性の詳細については、420 ページの「managed-object 要素の属性」を参照してください。

custom-executable のサブ要素

次に custom-executable タイプのサブ要素を示します。

要素	説明
process	管理リソースの開始、 停止、 ping、 および強制終了アクションに対して実行するプロセスの情報を含みます。通常、 custom-executable 管理オブジェクト定義には、アクションごとに process 要素があります。

```
...
<managed-objects>
       <managed-object ...type="custom-executable".../>
             ...
             <process ...attributes...>
                  ...
             </process>
             [<process ...attributes...>
                  ...
             </process>]
             [<process ...attributes...>
                  ...
             </process>]
             ...
       </managed-object>
       ...
</managed-objects>
...
```

さらに、managed-object のサブ要素 time-rules（タイプ固有ではない）を使用して、管理リソース プロセスを開始または停止するための cron 形式のルールを定義できます。

サブ要素	説明
time-rules	設定された時間ルールが実行される前の管理オブジェクトの基本状態を記述する属性を含みます。詳細については、「管理オブジェクト time-rules 要素」を参照してください。

```
...
   <managed-objects>
          <managed-object ...type="custom-javascript"...>
               [<time-rules ... >
                    ...
               </time-rules>]
               <process>
                    ...
               </process>
          </managed-object>
           ...
    </managed-objects>
...
```

要素	説明
control-overrides	管理オブジェクトのアクション ストラテジとパラメータ情報を含む必須要素。詳細については、「管理オブジェクト control-overrides 要素」を参照してください。

```
...
<managed-objects>
       <managed-object ...type="custom-executable".../>
             <control-overrides>
                   ...
             </control-overrides>
             ...
       </managed-object>
       ...
</managed-objects>
...
```

control-overrides のサブ要素

次に control-overides のサブ要素を示します。

サブ要素	必須	デフォルト値	説明
start	はい (pinger 管理オブジェクトの場合は不要)	該当なし	開始アクション ストラテジを指定し、開始アクション固有の情報を含む process 要素を参照するために使用します。

```
...
<managed-objects>
       <managed-object ...type="custom-executable".../>
             <control-overrides>
                   [<start ...attributes.../>]
                   ...
             </control-overrrides>
             <process ...attributes...>
                   ...
             </process>
             [<process ...attributes...>
                   ...
             </process>]
             [<process ...attributes...>
                   ...
             </process>]
             ...
       </managed-object>
       ...
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
ping	はい	該当なし	ping アクション ストラテジを指定し、ping アクション固有の情報を含む process 要素を参照するために使用します。

```
...
<managed-objects>
       <managed-object ...type="custom-executable".../>
             <control-overrides>
                   ...
                   <ping ...attributes.../>
                   ...
             </control-overrrides>
             <process ...attributes...>
                   ...
             </process>
             [<process ...attributes...>
                   ...
             </process>]
             [<process ...attributes...>
```

```
                ...
            </process>]
            ...
        </managed-object>
        ...
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
stop	はい (pinger 管理オブジェクトの場合は不要)	該当なし	停止アクション ストラテジを指定し、停止アクション固有の情報を含む process 要素を参照するために使用します。

```
...
<managed-objects>
        <managed-object ...type="custom-executable".../>
            <control-overrides>
                ...
                <stop ...attributes.../>
                ...
            </control-overrrides>
            <process ...attributes...>
                ...
            </process>
            [<process ...attributes...>
                ...
            </process>]
            [<process ...attributes...>
                ...
            </process>]
            ...
        </managed-object>
        ...
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
kill	いいえ	該当なし	強制終了アクション ストラテジを指定し、強制終了アクション固有の情報を含む process 要素を参照するために使用します。

```
...
<managed-objects>
        <managed-object ...type="custom-executable".../>
            <control-overrides>
                ...
                [<kill ...attributes.../>]
                ...
            </control-overrrides>
            <process ...attributes...>
                ...
            </process>
            [<process ...attributes...>
                ...
            </process>]
            [<process ...attributes...>
                ...
            </process>]
            ...
        </managed-object>
        ...
```

```
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
parameters	いいえ	該当なし	アクション ストラテジの追加パラメータを指定します。 詳細については、「管理オブジェクトの control-overrides 要素」、「parameters サブ要素の属性」を参照してください。

```
...
<managed-objects>
       <managed-object ...type="custom-executable".../>
             <control-overrides>
                    [<start ...attributes.../>]
                    <ping ...attributes.../>
                    [<stop ...attributes.../>]
                    [<kill ...attributes.../>]
                    <parameters ...attributes.../>
             </control-overrrides>
             <process ...attributes...>
                  ...
             </process>
             [<process ...attributes...>
                  ...
             </process>]
             [<process ...attributes...>
                  ...
             </process>]
             ...
       </managed-object>
       ...
</managed-objects>
...
```

start サブ要素の属性

次に start サブ要素の属性を示します。

属性	必須	デフォルト値	説明
strategy	はい	run-custom-executable	管理リソースを開始するプロセスを実行します。このアクションを無効にするには、null に設定します。
data-id-ref	はい	1	custom-executable 開始アクション ストラテジの一意の識別子。開始アクションに対して実行するプロセスに固有の設定情報を含む process サブ要素を参照するために AppServer によって使用されます。指定できる値は正の整数です。

```
...
<managed-objects>
       <managed-object ...type="custom-executable".../>
             <control-overrides>
                   <start strategy="run-custom-executable|null" data-id-
                        ref="integer"/>
                    <ping ...attributes.../>
                    <stop ...attributes.../>
                    <kill ...attributes.../>
                    <parameters ...attributes.../>
             </control-overrrides>
             <process ...attributes...>
                  ...
             </process>
             <process ...attributes...>
                  ...
             </process>
             <process ...attributes...>
```

```
                ...
            </process>
            ...
        </managed-object>
        ...
</managed-objects>
...
```

ping サブ要素の属性

次に ping サブ要素の属性を示します。

属性	必須	デフォルト値	説明
strategy	はい	ping-custom-executable	管理リソースを ping するプロセスを実行します。このアクションを無効にするには、 null に設定します。
data-id-ref	はい	2	custom-executable ping アクション ストラテジの一意の識別子。 ping アクションに対して実行するプロセスに固有の設定情報を含む process サブ要素を参照するために AppServer によって使用されます。指定できる値は正の整数です。

```
...
<managed-objects>
       <managed-object ...type="custom-executable".../>
             <control-overrides>
                    [<start strategy="run-custom-executable|null"
                      data-id-ref="integer"/>]
                    <ping strategy="ping-custom-executable|null"
                        data-id-ref="integer"/>
                    [<stop ...attributes.../>]
                    [<kill ...attributes.../>]
                    [<parameters ...attributes.../>]
             </control-overrrides>
             <process ...attributes...>
                  ...
             </process>
             [<process ...attributes...>
                  ...
             </process>]
             [<process ...attributes...>
                  ...
             </process>]
             ...
       </managed-object>
       ...
</managed-objects>
...
```

stop サブ要素の属性

次に stop サブ要素の属性を示します。

属性	必須	デフォルト値	説明
strategy	はい	run-custom-executable	管理リソースを停止するプロセスを実行します。このアクションを無効にするには、null に設定します。
data-id-ref	はい	3	custom-executable 停止アクション ストラテジの一意の識別子。 停止アクションに対して実行するプロセスに固有の設定情報を含む process サブ要素を参照するために AppServer によって使用されます。指定できる値は正の整数です。

```
...
<managed-objects>
       <managed-object ...type="custom-executable".../>
             <control-overrides>
                    [<start strategy="run-custom-executable|null"
                        data-id-ref="integer"/>]
                    <ping strategy="ping-custom-executable|null"
                        data-id-ref="integer"/>
                    [<stop strategy="run-custom-executable|null"
                        data-id-ref="integer"/>]
                    [<kill ...attributes.../>]
                    [<parameters ...attributes.../>]
             </control-overrrides>
             <process ...attributes...>
                  ...
             </process>
             [<process ...attributes...>
                  ...
             </process>]
             [<process ...attributes...>
                  ...
             </process>]
             ...
       </managed-object>
       ...
</managed-objects>
...
```

kill サブ要素の属性

次に kill サブ要素の属性を示します。

属性	必須	デフォルト値	説明
strategy	はい	run-custom-executable	管理リソースを強制終了するプロセスを実行します。このアクションを無効にするには、 null に設定します。
data-id-ref	はい	4	custom-executable 強制終了アクション ストラテジの一意の識別子。強制終了アクションに対して実行するプロセスに固有の設定情報を含む process サブ要素を参照するために AppServer によって使用されます。指定できる値は正の整数です。

```
...
<managed-objects>
       <managed-object ...type="custom-executable".../>
             <control-overrides>
                    [<start strategy="run-custom-executable|null"
                        data-id-ref="integer"/>]
```

```
                        <ping strategy="ping-custom-executable|null"
                            data-id-ref="integer"/>
                        [<stop strategy="run-custom-executable|null"
                            data-id-ref="integer"/>]
                        [<kill strategy="run-custom-executable|null"
                            data-id-ref="integer"/>]
                        [<parameters ...attributes.../>]
                  </control-overrrides>
                  <process ...attributes...>
                        ...
                  </process>
                  [<process ...attributes...>
                        ...
                  </process>]
                  [<process ...attributes...>
                        ...
                  </process>]
                  ...
            </managed-object>
            ...
      </managed-objects>
      ...
```

process サブ要素の属性

process 要素の属性とサブ要素のリストおよび説明については、433 ページの「process サブ要素の属性」を参照してください。

```
      ...
      <managed-objects>
            <managed-object ...type="custom-executable".../>
                  <control-overrides>
                        [<start strategy="run-custom-executable|null"
                            data-id-ref="integer"/>]
                        <ping strategy="ping-custom-executable|null"
                            data-id-ref="integer"/>
                        [<stop strategy="run-custom-executable|null"
                            data-id-ref="integer"/>]
                        [<kill strategy="run-custom-executable|null"
                            data-id-ref="integer"/>]
                        [<parameters ...attributes.../>]
                  </control-overrrides>
                  <process data-id="integer" command="fullpath"
                     directory="fullpathdirectory">
                        ...
                  </process>
                  [<process ...attributes...>
                        ...
                  </process>]
                  [<process ...attributes...>
                        ...
                  </process>]
                  ...
            </managed-object>
            ...
      </managed-objects>
      ...
```

osagent 管理オブジェクト タイプ

ここでは、osagent 管理オブジェクト type のサブ要素と属性をリストおよび説明します。

メモ osagent 管理オブジェクト属性の詳細については、420 ページの「managed-object 要素の属性」を参照してください。

osagent のサブ要素

osagent タイプ管理オブジェクトに固有の設定情報は、次のサブ要素に含まれています。

- process
- osagent

さらに、managed-object のサブ要素 time-rules（タイプ固有ではない）を使用して、管理リソース プロセスを開始または停止するための cron 形式のルールを定義できます。

サブ要素	説明
time-rules	設定された時間ルールが実行される前の管理オブジェクトの基本状態を記述する属性を含みます。詳細については、「管理オブジェクト time-rules 要素」を参照してください。

```
...
   <managed-objects>
         <managed-object ...type="osagent"...>
              [<time-rules ... >
                    <time-rule ... />
                    ...
              </time-rules>]
         </managed-object>
          ...
    </managed-objects>
...
```

managed-object のサブ要素 control-overrides（タイプ固有ではない）を使用して、デフォルトのアクション ストラテジやアクション パラメータを上書きできます。

サブ要素	説明
control-overrides	デフォルトを上書きするために指定されるアクション ストラテジやアクション パラメータを含みます。詳細については、「管理オブジェクト control-overrides 要素」を参照してください。

```
...
   <managed-objects>
         <managed-object ...type="osagent"...>
                    ...
              [<control-overrides>
                    ...
              </control-overrides>]
         </managed-object>
          ...
    </managed-objects>
...
```

サブ要素	説明
process	スマート エージェント (osagent) 管理オブジェクト プロセス設定情報を含みます。

```
...
<managed-objects>
      <managed-object ...type="osagent"...>
            <process ...attributes...>
                  ...
            </process>
            ...
```

```
      </managed-object>
</managed-objects>
...
```

サブ要素	説明
osagent	スマート エージェント（osagent）位置情報を含みます。

```
...
<managed-objects>
      <managed-object ...type="osagent"...>
            <process ...attributes...>
                ...
            </process>
            <osagent ...attributes.../>
      </managed-object>
</managed-objects>
...
```

process の属性とサブ要素

process 要素の属性とサブ要素のリストおよび説明については、以下のセクションを参照してください。

- process サブ要素の属性
- process のサブ要素

```
...
<managed-objects>
      <managed-object ...type="osagent"...>
            <process command="fullpath"  directory="fullpathdirectory">
                [<arguments>
                    [<argument>argument1<argument>
                    [<argument includePlatforms="Windows|
                      UNIX"argument3<argument>
                    [<argument>argument2<argument>]
                    [...more arguments...]
                </arguments>]
               [<library-path>
                    [<directory>path1</directory>]
                    [<directory>path2</directory>]
                    [...more librarypaths...]
                </library-path>]
                [<env-vars use-default-env="true|false"
                   use-current-env="true|false" use-vbroker-env="true|false">
                    [<env-var name="name" value="value"/>]
                    [...more env-vars...]
                <env-vars>]
            </process>
             ...
      </managed-object>
</managed-objects>
...
```

osagent のサブ要素

次に osagent のサブ要素を示します。

サブ要素	必須	デフォルト値	説明
port	はい	該当なし	スマート エージェント (osagent) ポート。
host	いいえ	該当なし	IP アドレスまたはホスト名による、このスマート エージェント（osagent）の物理位置。 AppServer は、ping アクションを実行する際に、このスマート エージェントを使用します。指定しないと、 AppServer は、ping を実行して、 正しいポートを持つ任意のスマート エージェント (osagent) をネットワーク上で探します。
logdir	いいえ	${CONFIG.path}/mos/${MO.name}	スマート エージェント（osagent）ログ ファイルが書き込まれるディレクトリ。

```
...
<managed-objects>
      <managed-object ...type="osagent"...>
            <process の属性>
             ...
            </process>
            <osagent port="port_number"
               host="string" logdir="fullpathdirectory"/>
             ...
      </managed-object>
</managed-objects>
...
```

apache-process 管理オブジェクト タイプ

ここでは、apache-process 管理オブジェクト type のサブ要素と属性をリストおよび説明します。

メモ apache-process 管理オブジェクト属性の詳細については、420 ページの「managed-object 要素の属性」を参照してください。

メモ Apache Web サーバーの管理オブジェクトを作成する場合、管理オブジェクト名にスペースを入れることはできません。"Apache Web Server" など、管理オブジェクト名にスペースが含まれると、Apache Web サーバーは起動に失敗します。

apache-process のサブ要素

次に apache-process タイプのサブ要素を示します。

サブ要素	説明
apache-data	apache-process 管理オブジェクト タイプ固有の設定情報を含みます。

```
...
<managed-objects>
      <managed-object ...type="apache-process"...>
            <apache-data ...attributes...>
                ...
            </apache-data>
             ...
      </managed-object>
```

```
</managed-objects>
...
```

さらに、`managed-object` のサブ要素 `time-rules`（タイプ固有ではない）を使用して、管理リソース プロセスを開始または停止するための `cron` 形式のルールを定義できます。

サブ要素	説明
time-rules	設定された時間ルールが実行される前の管理オブジェクトの基本状態を記述する属性を含みます。詳細については、「管理オブジェクト time-rules 要素」を参照してください。

```
...
   <managed-objects>
          <managed-object ...type="apache-process"...>
               [<time-rules ... >
                     ...
               </time-rules>]
          </managed-object>
           ...
    </managed-objects>
...
```

サブ要素	説明
control-overrides	デフォルトを上書きするために指定されるアクション ストラテジやアクション パラメータを含みます。詳細については、「管理オブジェクト control-overrides 要素」を参照してください。

```
...
   <managed-objects>
          <managed-object ...type="apache-process"...>
               <apache-data>
                     ...
               </apache-data>
               [<control-overrides>
                     ...
               </control-overrides>]
          </managed-object>
           ...
    </managed-objects>
...
```

apache-data サブ要素の属性

次に apache-data サブ要素の属性を示します。

属性	必須	デフォルト値	説明
command	はい	${AGENT.root}/bin/<apache_managedobject_name>。ここで、置換文字列 ${AGENT.root} は、割り当てられたエージェントが存在するホストのルートを表します。	Apache Web サーバー管理リソースの開始に使用されるコマンドの絶対パス。
httpd-conf	はい	${CONFIG.path}/mos/${MO.name}/conf/httpd.conf - ここで、各置換文字列の意味は次のとおりです。 ■ ${CONFIG.path} - configuration.xml への絶対パス。 ■ ${MO.name} - 管理オブジェクト名（管理オブジェクト name 属性値による）を表します。	Apache httpd.conf ファイルの絶対パス。
nt-service	はい（Windows プラットフォームの場合）	false	この属性は、Op-Center が Microsoft NT サービスとして実行されている Apache 管理オブジェクトとやり取りする方法に影響します。この管理オブジェクトを Microsoft NT サービスとして実行する場合は true、そうではない場合は false に設定します。

```
...
<managed-objects>
      <managed-object ...type="apache-process"...>
            <apache-data command="fullpath" httpd-conf="fullpath"
               nt-service="true|false">
                ...
            </apache-data>
             ...
      </managed-object>
</managed-objects>
...
```

apache-data のサブ要素

次に apache-data のサブ要素を示します。

サブ要素	必須	デフォルト値	説明
arguments	いいえ	該当なし	この要素の詳細については、「process のサブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="apache-process"...>
            <apache-data attributes>
                [<arguments>
                     <argument>argument1</argument>
                          [<argument>argument2</argument>]
                    [...more arguments...]
                </arguments>]
                ...
            </apache-data>
             ...
      </managed-object>
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
env-vars	いいえ	該当なし	この要素の詳細については、「process のサブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="apache-process"...>
            <apache-data attributes>
                [<env-vars>
                    <env-var name="variable1"
                        value="value1"/>
                   [<env-var name="variable2"
                        value="value2"/>]
                   [...more env-vars...]
                </env-vars>]
                ...
            </apache-data>
             ...
      </managed-object>
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
library-path	いいえ	該当なし	UNIX のみ。この要素の詳細については、「process のサブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="apache-process"...>
            <apache-data attributes>
                [<library-path>
                    <directory>path1</directory>
                   [<directory>path2</directory>]
                   [...more paths...]
                </library-path>]
                ...
            </apache-data>
             ...
      </managed-object>
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
path	いいえ	該当なし	この要素の詳細については、「process のサブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="apache-process"...>
            <apache-data attributes>
                [<path>
                    <directory>path1</directory>
                  [<directory>path2</directory>]
                  [...more paths...]
                </path>]
                ...
            </apache-data>
             ...
      </managed-object>
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
stdin path	いいえ	該当なし	この要素の詳細については、「process のサブ要素」を参照してください。
stdout path	いいえ	該当なし	この要素の詳細については、「process のサブ要素」を参照してください。
stderr path	いいえ	該当なし	この要素の詳細については、「process のサブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="apache-process"...>
            <apache-data attributes>
                <stdin path="path1"/>
                <stdout path="path2"
                  append="true|false"/>
                <stderr path="path3"
                  append="true|false" />
                ...
            </apache-data>
             ...
      </managed-object>
</managed-objects>
...
```

ots 管理オブジェクト タイプ

ここでは、ots 管理オブジェクト type のサブ要素と属性をリストおよび説明します。

メモ ots 管理オブジェクト属性の詳細については、420 ページの「managed-object 要素の属性」を参照してください。

ots のサブ要素

次に ots タイプのサブ要素を示します。

サブ要素	説明
process	ots 管理オブジェクトに固有の設定情報は、process サブ要素に含まれます。process 要素の属性とサブ要素のリストおよび説明については、「process サブ要素の属性」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="ots"...>
            <process command="fullpath">
               [<arguments>
                   <argument>argument1</argument>
                   [...more arguments...]
               [<library-path>
                   <directory>path1</directory>
                   [...more paths...]
               </library-path>
               [<env-vars use-default-env="true|false"
                  use-current-env="true|false" use-vbroker-env="true|false"/>
                  [<env-var name="variable1" value="value1"/>]
                  [<env-var name="variable2" value="value2"="/>]
                  [...more env-vars...]
               </env-vars>]
               [<stdout path="path1"/>]
               [<stderr path="path2"/>]
            </process>
            ...
      </managed-object>
</managed-objects>
...
```

さらに、managed-object のサブ要素 time-rules（タイプ固有ではない）を使用して、管理リソース プロセスを開始または停止するための cron 形式のルールを定義できます。

サブ要素	説明
time-rules	設定された時間ルールが実行される前の管理オブジェクトの基本状態を記述する属性を含みます。詳細については、「管理オブジェクト time-rules 要素」を参照してください。

```
...
   <managed-objects>
         <managed-object ...type="ots"...>
              [<time-rules ... >
                    ...
              </time-rules>]
         </managed-object>
          ...
    </managed-objects>
...
```

サブ要素	説明
control-overrides	デフォルトを上書きするために指定されるアクション ストラテジやアクション パラメータを含みます。詳細については、「管理オブジェクト control-overrides 要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="ots"...>
                [<control-overrides>
                      ...
                </control-overrides>]
                <process command="fullpath">
                   [<arguments>
                       <argument>argument1</argument>
                       [...more arguments...]
                   [<library-path>
                       <directory>path1</directory>
                       [...more paths...]
                   </library-path>
                   [<env-vars use-default-env="true|false"
                         use-current-env="true|false"
                         use-vbroker-env="true|false"/>
                      [<env-var name="variable1" value="value1"/>]
                      [<env-var name="variable2" value="value2"="/>]
                      [...more env-vars...]
                   </env-vars>]
                   [<stdout path="path1"/>]
                   [<stderr path="path2"/>]
                </process>
              ...
      </managed-object>
</managed-objects>
...
```

tibco 管理オブジェクト タイプ

ここでは、tibco 管理オブジェクト type のサブ要素と属性をリストおよび説明します。

メモ tibco 管理オブジェクト属性の詳細については、420 ページの「managed-object 要素の属性」を参照してください。

tibco のサブ要素

tibco 管理オブジェクト固有の設定情報は、次のサブ要素に含まれています。

- tibco-data
- process

サブ要素	説明
tibco-data	tibco 管理オブジェクト タイプの Java メッセージ サービス固有の設定情報を含みます。

```
...
<managed-objects>
      <managed-object ...type="tibco"...>
            <tibco-data attributes/>
            <process attributes>
                  ...
            </process>
      </managed-object>
      ...
</managed-objects>
...
```

サブ要素	説明
process	Tibco JMS 管理オブジェクト プロセス設定情報を含みます。

```
...
<managed-objects>
      <managed-object ...type="tibco"...>
            <tibco-data attributes/>
            <process attributes>
                ...
            </process>
      </managed-object>
      ...
</managed-objects>
...
```

さらに、`managed-object` のサブ要素 `time-rules`（タイプ固有ではない）を使用して、管理リソース プロセスを開始または停止するための `cron` 形式のルールを定義できます。

サブ要素	説明
time-rules	設定された時間ルールが実行される前の管理オブジェクトの基本状態を記述する属性を含みます。詳細については、「管理オブジェクト time-rules 要素」を参照してください。

```
...
   <managed-objects>
         <managed-object ...type="tibco"...>
              [<time-rules ... >
                   ...
              </time-rules>]
         </managed-object>
          ...
    </managed-objects>
...
```

`managed-object` のサブ要素 `control-overrides`（タイプ固有ではない）を使用して、デフォルトのアクション ストラテジやアクション パラメータを上書きできます。

サブ要素	説明
control-overrides	デフォルトを上書きするために指定されるアクション ストラテジやアクション パラメータを含みます。詳細については、「管理オブジェクト control-overrides 要素」を参照してください。

```
...
   <managed-objects>
         <managed-object ...type="tibco"...>
                   ...
              [<control-overrides>
                   ...
              </control-overrides>]
         </managed-object>
          ...
    </managed-objects>
...
```

tibco-data サブ要素の属性

次に tibco-data サブ要素の属性を示します。

属性	必須	デフォルト値	説明
jms-home	はい	${AGENT.root}jms/tibco。ここで、置換文字列変数 ${AGENT.root} は、この tibco 管理オブジェクトの割り当て先のエージェントホストのルートを表します。	Tibco JMS プロバイダ ルートのフル パスおよびディレクトリ。
server-url	はい	tcp://*localhost*:7222	Tibco JMS プロバイダにプログラムによって接続するためのアドレス。 tcp://*hostname*:*portnumber* の形式で指定します。 **メモ：***portnumber* は、partner-url 属性の *portnumber* と異なっている必要があります。
server-name	はい	該当なし	Tibco JMS サーバーを識別する一意の名前。フォールトトレラント グループの場合は、グループ内の両方の Tibco JMS 管理オブジェクトが同じ server-name を持つ必要があります。
partner-url	はい（管理リソースがフォールトトレランス グループにある場合）	該当なし	同じフォールトトレラント グループ内の両方の Tibco JMS サーバーが相互に通信するために使用するアドレス。 tcp://*localhost*:*portnumber* の形式で指定します。 **メモ：***portnumber* は、server-url 属性の *portnumber* と異なっている必要があります。
shared-data-storage	はい	該当なし	メッセージの永続化に使用します。Tibco サーバーが完全な読み取り / 書き込みアクセス権を持つ絶対パスおよびディレクトリ。 **メモ :** フォールトトレラント グループの場合は、グループ内の両方の Tibco JMS 管理オブジェクトが同じ shared-data-storage 位置を使用する必要があります。

```
...
<managed-objects>
      <managed-object ...type="tibco"...>
            <tibco-data jms-home="fullpathdirectory"
                server-url="tcp://hostname:portnumber1" server-name="string"
                partner-url="tcp://hostname:portnumber2"
```

```
                shared-data-storage="fullpathdirectory"/>
            <process attributes>
                ...
            </process>
        </managed-object>
        ...
</managed-objects>
...
```

process サブ要素の属性

process 要素の属性とサブ要素のリストおよび説明については、433 ページの「process サブ要素の属性」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="tibco"...>
            <tibco-data attributes/>
            <process command="fullpath" directory="fullpathdirectory">
                  <stdout path="path1"/>
                  <stderr path="path2"/>
            </process>
      </managed-object>
      ...
</managed-objects>
...
```

partition 管理オブジェクト タイプ

ここでは、partition 管理オブジェクト type のサブ要素と属性をリストおよび説明します。

メモ partition 管理オブジェクト属性の詳細については、420 ページの「managed-object 要素の属性」を参照してください。

partition-process のサブ要素

partition タイプ管理オブジェクトに固有の設定情報は、partition の次のサブ要素に含まれています。

- partition-process
- partition-services
- jmx

サブ要素	説明
partition-process	AppServer パーティション管理オブジェクト プロセス設定情報を含みます。
partition-services	AppServer パーティション サービス管理オブジェクトへの管理オブジェクト リファレンスのリストを含みます。
jmx	JMX サーバーへの接続に必要な情報を含みます。

managed-object のサブ要素 control-overrides （タイプ固有ではない）を使用して、デフォルトのアクション ストラテジやアクション パラメータを上書きできます。

サブ要素	説明
control-overrides	デフォルトを上書きするために指定されるアクション ストラテジやアクション パラメータを含みます。詳細については、「管理オブジェクト control-overrides 要素」を参照してください。

```
...
   <managed-objects>
```

```
            <managed-object ...type="partition"...>
                        ...
                  [<control-overrides>
                        ...
                  </control-overrides>]
            </managed-object>
             ...
      </managed-objects>
...
```

partition-process サブ要素の属性

partition-process サブ要素は、process サブ要素のすべての属性を継承します。これらの属性の詳細については、433 ページの「process サブ要素の属性」を参照してください。

partition-process のサブ要素

partition-process サブ要素は、process のすべてのサブ要素を継承します。これらのサブ要素の詳細については、434 ページの「process のサブ要素」を参照してください。

次に partition-process 要素固有の追加のサブ要素を示します。

サブ要素	必須	デフォルト値	説明
jpda	いいえ	該当なし	パーティション管理リソースでの関数のデバッグ方法を決定します。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
            <partition-process command="fullpath">
                [<java-properties>
                   <java-property name="name1" value="value1"/>
                </java-properties>]
                [<jpda ...attributes.../>]
                [<env-vars use-default-env="true|false"
                      use-current-env="true|false"
                      use-vbroker-env="true|false"/>
                   [<env-var name="variable1" value="value1"/>]
                   [<env-var name="variable2" value="value2"="/>]
                   [...more env-vars...]
                </env-vars>]
                [<stdout path="path1"/>]
                [<stderr path="path2"/>]
            </partition-process>
            ...
      </managed-object>
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
optimizeit	いいえ	該当なし	Borland Optimizeit ServerTrace または Profiler で実行されている AppServer を管理するための設定情報を含みます。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
           <partition-process ...attributes...>
              [<optimizeit ...attributes...>]
                   ...
           </partition-process>
      </managed-object>
</managed-objects>
...
```

optimizeit サブ要素の属性

次に optimizeit サブ要素の属性を示します。

属性	必須	デフォルト値	説明
enable	はい	false	AppServer パーティション管理オブジェクトを Borland Optimizeit ServerTrace または Profiler とともに実行できるようにするには、 true に設定します。
mode	はい（enable が true に設定されている場合）	servertrace	enable 要素が true に設定されている場合に、有効にする Optimizeit 製品を指定します。AppServer パーティション管理オブジェクトを Optimizeit Profiler とともに実行する場合は、 profiler に設定します。
sthome	はい（enable が true に設定されている場合）	該当なし	ServerTrace または Profiler インストールのフル パスとディレクトリを指定します。
jdkpath	いいえ	テンプレート JDK のパス	JDK ホームのフル パスを指定するために使用されます。
xmlpath	いいえ	AppServer Template Optimizeit xml 設定ファイル。	Optimizeit xml 設定ファイルのフル パスを指定するために使用されます。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
           <partition-process ...attributes...>
              [<optimizeit enable="false|true"
                mode="servertrace|profiler" sthome="fullpath"
                jdkpath="fullpath" xmlpath="fullpath">]
                ...
           </partition-process>
      </managed-object>
</managed-objects>
...
```

optimizeit 要素のサブ要素

次に利用可能な optimizeit 要素のサブ要素を示します。

サブ要素	必須	デフォルト値	説明
strace	いいえ	該当なし	AppServer パーティションを Borland Optimizeit ServerTrace とともに実行するために使用されます。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
           <partition-process ...attributes...>
              <optimizeit ...attributes...>
                 [<strace ...attributes...>]
                 ...
           </partition-process>
      </managed-object>
</managed-objects>
...
```

サブ要素	必須	デフォルト値	説明
profiler	いいえ	該当なし	AppServer パーティションを Borland Optimizeit Profiler とともに実行するために使用されます。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
           <partition-process ...attributes...>
              <optimizeit ...attributes...>
                 [<profiler ...attributes...>]
                 ...
           </partition-process>
      </managed-object>
</managed-objects>
...
```

strace サブ要素の属性

次に strace サブ要素の属性を示します。

属性	必須	デフォルト値	説明
sthome	はい - optimizeit 属性が enable="true" で mode="servertrace" の場合。	該当なし	ServerTrace インストールのフル パスとディレクトリを指定します。
jdkpath	いいえ	テンプレート JDK のパス	JDK ホームのフル パスを指定するために使用されます。
xmlpath	いいえ	AppServer Template Optimizeit xml 設定ファイル。	Optimizeit xml 設定ファイルのフル パスを指定するために使用されます。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
           <partition-process ...attributes...>
              <optimizeit enable="true"
                mode="servertrace" sthome="fullpath"
                jdkpath="fullpath" xmlpath="fullpath">
                <strace sthome="fullpath"
                   jdkpath="fullpath" xmlpath="fullpath">
                 ...
               </strace>
           </partition-process>
      </managed-object>
</managed-objects>
...
```

profiler サブ要素の属性

次に profiler サブ要素の属性を示します。

属性	必須	デフォルト値	説明
sthome	はい - optimizeit 属性が enable="true" で mode="profiler" の場合。	該当なし	Profiler インストールのフル パスとディレクトリを指定します。

属性	必須	デフォルト値	説明
jdkpath	いいえ	テンプレート JDK のパス	JDK ホームのフル パスを指定するために使用されます。
xmlpath	いいえ	AppServer Template Optimizeit xml 設定ファイル。	Optimizeit xml 設定ファイルのフル パスを指定するために使用されます。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
           <partition-process ...attributes...>
              <optimizeit enable="true"
                mode="profiler" sthome="fullpath"
                jdkpath="fullpath" xmlpath="fullpath">
                 <profiler sthome="fullpath"
                jdkpath="fullpath" xmlpath="fullpath">
                 ...
                    ...
                 </profiler>
           </partition-process>
      </managed-object>
</managed-objects>
...
```

strace 要素のサブ要素

次に利用可能な strace 要素のサブ要素を示します。

strace 要素の classpath サブ要素

このサブ要素の詳細については、441 ページの「java-process のサブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
           <partition-process ...attributes...>
              <optimizeit enable="true"
                mode="servertrace" sthome="fullpath"
                jdkpath="fullpath" xmlpath="fullpath">
                 <strace ...attributes...>
                    [<classpath>
                       [<classpath-entry>classpath1</classpath-entry>]
                       [<classpath-entry>classpath2</classpath-entry>]
                       [...more classpaths...]
                    </classpath>]
                    ...
                 </strace>
                 ...
           </partition-process>
      </managed-object>
</managed-objects>
...
```

strace 要素の bootclasspath サブ要素

サブ要素	必須	デフォルト値	説明
bootclasspath	はい	該当なし	ServerTrace oibcp.jar のクラス パスを指定するために使用されます。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
           <partition-process ...attributes...>
              <optimizeit enable="true"
                mode="strace" sthome="fullpath"
                jdkpath="fullpath" xmlpath="fullpath">
                 <strace ...attributes...>
                    [<bootclasspath>
                      [<classpath-entry>classpath-to-oibcp.jar
                      </classpath-entry>]
                     </bootclasspath>]
                    ...
                  </strace>
                  ...
           </partition-process>
      </managed-object>
</managed-objects>
...
```

strace 要素の options サブ要素

このサブ要素の詳細については、441 ページの「java-process のサブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
           <java-process ...attributes...>
              <optimizeit enable="true"
                mode="servertrace" sthome="fullpath"
                jdkpath="fullpath" xmlpath="fullpath">
                 <strace ...attributes...>
                    [<options>
                      [<option>option1</option>]
                      [<option>option2</option>]
                      [...more options...]
                    </options>]
                    ...
                 </strace>
           </partition-process>
      </managed-object>
</managed-objects>
...
```

strace 要素の path サブ要素

このサブ要素については、436 ページの「path サブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
           <partition-process ...attributes...>
              <optimizeit enable="true"
                mode="servertrace" sthome="fullpath"
                jdkpath="fullpath" xmlpath="fullpath">
                 <strace ...attributes...>
                   [<path>
                      <directory>path1</directory>
                     [<directory>path2</directory>]
                     [...more paths...]
                    </path>]
                    ...
                 </strace>
           </partition-process>
      </managed-object>
</managed-objects>
...
```

profiler 要素のサブ要素

次に利用可能な `profiler` 要素のサブ要素を示します。

profiler 要素の classpath サブ要素

このサブ要素については、441 ページの「java-process のサブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
           <partition-process ...attributes...>
              <optimizeit enable="true"
                mode="profiler" sthome="fullpath"
                jdkpath="fullpath" xmlpath="fullpath">
                 <strace ...attributes...>
                    [<classpath>
                       [<classpath-entry>classpath1</classpath-entry>]
                       [<classpath-entry>classpath2</classpath-entry>]
                       [...more classpaths...]
                    </classpath>]
                    ...
                 </profiler>
                 ...
           </partition-process>
      </managed-object>
</managed-objects>
...
```

profiler 要素の bootclasspath サブ要素

サブ要素	必須	デフォルト値	説明
bootclasspath	はい	該当なし	Optimizeit Profiler oibcp.jar のクラス パスを指定するために使用されます。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
           <partition-process ...attributes...>
              <optimizeit enable="true"
                mode="profiler" sthome="fullpath"
                jdkpath="fullpath" xmlpath="fullpath">
                 <profiler ...attributes...>
                    [<bootclasspath>
                      [<classpath-entry>classpath-to-oibcp.jar
                      </classpath-entry>]
                    </bootclasspath>]
                    ...
                 </profiler>
                 ...
           </partition-process>
      </managed-object>
</managed-objects>
...
```

profiler 要素の options サブ要素

このサブ要素については、441 ページの「java-process のサブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
           <partition-process ...attributes...>
              <optimizeit enable="true"
                mode="profiler" sthome="fullpath"
                jdkpath="fullpath" xmlpath="fullpath">
                 <profiler ...attributes...>
                    [<options>
                      [<option>option1</option>]
                      [<option>option2</option>]
                      [...more options...]
                    </options>]
                    ...
                 </profiler>
           </partition-process>
      </managed-object>
</managed-objects>
...
```

profiler 要素の path サブ要素

このサブ要素については、436 ページの「path サブ要素」を参照してください。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
           <partition-process ...attributes...>
              <optimizeit enable="true"
                mode="profiler" sthome="fullpath"
                jdkpath="fullpath" xmlpath="fullpath">
                 <profiler ...attributes...>
                   [<path>
                      <directory>path1</directory>
                     [<directory>path2</directory>]
                     [...more paths...]
                    </path>]
                    ...
                 </profiler>
           </partition-process>
      </managed-object>
</managed-objects>
...
```

jpda サブ要素の属性

次に jpda 要素の属性を示します。

属性	必須	デフォルト値	説明
enable-jpda-debug	いいえ	false	パーティションで実行されるアプリケーションの JPDA デバッグを有効にするには、True に設定します。false（デフォルト）に設定すると、パーティションで実行されるアプリケーションの JPDA デバッグは無効になります。
jdpa-transport-address	いいえ	jdpa によって動的に割り当て。	このアプリケーションにアタッチするときに JPDA デバッガが使用するポートを設定します。host:port の形式を使用します。指定されない場合、JPDA は、ポートを動的に割り当て、ポート番号を出力します。ユーザーは、出力された番号を読み取り、デバッガをアタッチする際にそれを使用する必要があります。
jdpa-suspend	いいえ	true	デバッガがアタッチするまでパーティションの操作を一時停止しない場合は、false に設定します。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
            <partition-process command="fullpath">
                [<java-properties>
                   <java-property name="name1"
                     value="value1"/>
                </java-properties>]
                [<jpda enable-jpda-debug="true|false"
                   jpda-transport-address="host:port"
                   jpda-suspend="true|false"/>]
                [<env-vars use-default-env="true|false"
                   use-current-env="true|false" use-vbroker-env="true|false"/>
                   [<env-var name="variable1" value="value1"/>]
                   [<env-var name="variable2" value="value2"="/>]
                   [...more env-vars...]
                </env-vars>]
                [<stdout path="path1"/>]
                [<stderr path="path2"/>]
            </partition-process>
            ...
      </managed-object>
</managed-objects>
...
```

partition-services 要素のサブ要素

次に partition-services 要素のサブ要素を示します。

サブ要素	必須	デフォルト値	説明
member	はい	該当なし	パーティション管理オブジェクトの各パーティション サービスをリストおよび参照するために使用します。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
            <partition-process attributes>
                  ...
```

```
            </partition-process>
            <partition-services>
                  [<member attribute/>]
                  [...more members...]
                  ...
            </partition-services>
             ...
      </managed-object>
</managed-objects>
...
```

member サブ要素の属性

member サブ要素には次の属性だけがあります。

属性	必須	デフォルト値	説明
mo-ref	はい	該当なし	パーティション管理ソースの各パーティションサービスに対して、パーティション サービスの論理 AppServer 名は次の形式で表されます：*${agentname}/partition_managedobject_name_partitionservice_managedobjectname*。置換文字列 ${mo.agent} はパーティション管理オブジェクトの割り当て先のエージェントを、${MO.name} はパーティション管理オブジェクト name 属性値を表すために使用されます。たとえば、${MO.agent}/${MO.name}_jss です。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
            <partition-process attributes>
                  ...
            </partition-process>
            <partition-services>
                  <member
                    mo-ref="agentname/managedobjectname_partitionservicename/>
                  [...more members...]
            </partition-services>
             ...
      </managed-object>
</managed-objects>
...
```

重要 partition-services 要素で使用される mo-ref 命名規則は厳密です。これを変更することはできません。

```
<managed-object [...] type="partition-process">
    <partition-process [...] />
    <partition-services>
        <member mo-ref="${MO.agent}/${MO.name}__interceptor"/>
        <member mo-ref="${MO.agent}/${MO.name}_jdatastore"/>
        <member mo-ref="${MO.agent}/${MO.name}_jss"/>
        <member mo-ref="${MO.agent}/${MO.name}_jts"/>
        <member mo-ref="${MO.agent}/${MO.name}_tomcat4"/>
        <member mo-ref="${MO.agent}/${MO.name}_visiconnect"/>
        <member mo-ref="${MO.agent}/${MO.name}_ejbcontainer"/>
        [<member mo-ref="${MO.agent}/${MO.name}_jms"/>]
    </partition-services>
</managed-object>
```

jmx サブ要素の属性

次に jmx サブ要素の属性を示します。

属性	必須	デフォルト値	説明
jmxserver	はい	bes	JMX サーバー タイプを識別するために使用されます。AppServer 設定の場合、 有効な値は bes です。
classpath	はい	該当なし	JMX サーバーへの接続に使用されるクラスパスを識別するために使用されます。

```
...
<managed-objects>
      <managed-object ...type="partition"...>
            <partition-process attributes>
                  ...
            </partition-process>
            <partition-services>
                  [<member attribute/>]
                  [...more members...]
                  ...
            </partition-services>
            <jmx attributes />
             ...
      </managed-object>
</managed-objects>
...
```

パーティション サービス管理オブジェクト タイプ

partition タイプ管理オブジェクトでは、各パーティション サービスが configuration.xml 内で管理オブジェクトとして定義されます。次に AppServer の各 partition-services 管理オブジェクト タイプについて説明します。

パーティション サービス管理オブジェクト タイプ	説明
ps-interceptor	AppServer **パーティション存続期間インターセプタ** パーティション サービスを表します。パーティション存続期間インターセプタを使用して、パーティション イベントを監視し、必要に応じてそれらのイベントに応答できます。
ps-jdatastore	JDataStore（Borland の完全 Java リレーショナル データベース）パーティション サービスのインスタンスを表します。
ps-jss	Java セッション サービス（JSS）パーティション サービスのインスタンスを表します。
ps-jts	Java トランザクション サービス（JTS）パーティション サービスのインスタンスを表します。
ps-tomcat4	Tomcat Web コンテナ パーティション サービスのインスタンスを表します。
ps-ejbcontainer	EJB コンテナ パーティション サービスのインスタンスを表します。
ps-visiconnect	VisiConnect（Borland の Java コネクタ アーキテクチャ（JCA）インプリメンテーション）パーティション サービスのインスタンスを表します。

第 47 章

管理オブジェクト control-overrides 要素

control-overrides は managed-object 要素のサブ要素です。control-overrides 要素で、アクション パラメータやアクション ストラテジに対する管理オブジェクトのデフォルトの上書きを指定できます。

control-overrides のサブ要素

次に control-overrides のサブ要素を示します。

要素	説明
parameters	指定されたアクションパラメータ設定の上書きを含みます。

```
...
   <managed-objects>
          <managed-object ...attributes...>
                ...
                [<control-overrides>
                    [<parameters ...attributes.../>]
                    ...
                </control-overrides>]
                ...
          </managed-object>
            ...
     </managed-objects>
...
```

要素	説明
start	指定された開始アクション ストラテジを含みます。

```
...
   <managed-objects>
          <managed-object ...attributes...>
                ...
                [<control-overrides>
                    [<parameters ...attributes.../>]
                    [<start ...attributes.../>]
                    ...
                </control-overrides>]
```

```
                ...
            </managed-object>
             ...
     </managed-objects>
...
```

要素	説明
stop	指定された停止アクション ストラテジを含みます。

```
...
   <managed-objects>
          <managed-object ...attributes...>
                ...
                [<control-overrides>
                    [<parameters ...attributes.../>]
                    [<start ...attributes.../>]
                    [<stop ...attributes.../>]
                    ...
                </control-overrides>]
                ...
          </managed-object>
           ...
     </managed-objects>
...
```

要素	説明
kill	指定された強制終了アクション ストラテジを含みます。

```
...
   <managed-objects>
          <managed-object ...attributes...>
                ...
                [<control-overrides>
                    [<parameters ...attributes.../>]
                    [<start ...attributes.../>]
                    [<stop ...attributes.../>]
                    [<kill ...attributes.../>]
                </control-overrides>]
                ...
          </managed-object>
           ...
     </managed-objects>
...
```

要素	説明
ping	指定された ping アクション ストラテジを含みます。

```
...
   <managed-objects>
          <managed-object ...attributes...>
                ...
                [<control-overrides>
                    [<parameters ...attributes.../>]
                    [<start ...attributes.../>]
                    [<stop ...attributes.../>]
                    [<kill ...attributes.../>]
                    [<ping ...attributes.../>]
                </control-overrides>]
                ...
          </managed-object>
           ...
     </managed-objects>
...
```

parameters サブ要素の属性

次に parameters サブ要素の属性を示します。

属性	必須	デフォルト値	説明
local-restart	いいえ	false	ブール属性。true に設定すると、ハブを使用できない場合は、管理リソースをエージェント ベースで再起動できます。
escalate-stop	いいえ	false	ブール属性。true に設定すると、停止操作がタイムアウトになった場合は、停止操作が強制終了に変更されます。
ping-policy	いいえ	not-when-stopped	管理オブジェクトは、状態が停止の場合は ping されません。管理オブジェクトの状態にかかわらず ping するには、always に設定します。
ping-interval	いいえ	5（秒）	管理リソースの通常の監視におけるチェック間隔（秒）。0 以上の整数を指定できます。
start-timeout	いいえ	90（秒）	開始操作がタイムアウトになって開始の試行から戻るまでの時間（秒）。0 以上の整数を指定できます。
start-ping-interval	いいえ	管理オブジェクトの ping-interval 値	開始操作の進捗状況に対するチェック間隔（秒）。0 以上の整数を指定できます。
start-first-ping-delay	いいえ	管理オブジェクトの first-ping-delay 値	開始操作の進捗状況を BAS がチェックし始めるまでの時間（秒）。0 以上の整数を指定できます。
stop-timeout	いいえ	90（秒）	停止操作がタイムアウトになって停止の試行から戻るまでの時間（秒）。0 以上の整数を指定できます。
stop-retry-interval	いいえ	空白 - 再試行しない	前の停止操作がタイムアウトになってから操作を再試行するまでの時間（秒）。0 以上の整数を指定できます。
stop-ping-interval	いいえ	管理オブジェクトの ping-interval 値	停止操作の進捗状況に対するチェック間隔（秒）。0 以上の整数を指定できます。
kill-timeout	いいえ	90（秒）	強制終了操作がタイムアウトになって強制終了の試行から戻るまでの時間（秒）。0 以上の整数を指定できます。
kill-retry-interval	いいえ	空白 - 再試行しない	前の強制終了操作がタイムアウトになってから操作を再試行するまでの時間（秒）。0 以上の整数を指定できます。

属性	必須	デフォルト値	説明
kill-ping-interval	いいえ	管理オブジェクトの ping-interval 値	強制終了操作の進捗状況に対するチェック間隔（秒）。0 以上の整数を指定できます。
kill-first-ping-interval	いいえ	管理オブジェクトの first-ping-interval 値	停止操作の進捗状況を BAS がチェックし始めるまでの時間（秒）。0 以上の整数を指定できます。

```
...
   <managed-objects>
          <managed-object ...attributes...>
                ...
                [<control-overrides>
                    [<parameters local-restart="true|false"
               escalate-stop="true|false" ping-policy="not-when-stopped|always"
               ping-retry-interval="seconds" start-timeout="seconds"
               start-ping-interval="seconds" start-first-ping-interval=="seconds"
               stop-timeout="seconds"
               stop-retry-interval="seconds" stop-ping-interval="seconds"
               kill-timeout="seconds" kill-retry-interval="seconds"
               kill-ping-interval="seconds"
               kill-first-ping-interval="seconds"/>]
                     ...
                  </control-overrides>]
                  ...
          </managed-object>
           ...
    </managed-objects>
...
```

start サブ要素の属性

次に start サブ要素の属性を示します。

属性	必須	デフォルト値	説明
strategy	はい	デフォルトは、管理オブジェクトタイプによって異なります。詳細は、「start サブ要素のストラテジ属性」を参照してください。	管理オブジェクトの開始アクションで BAS が使用するストラテジを指定します。

```
...
   <managed-objects>
          <managed-object ...attributes...>
                ...
                [<control-overrides>
                    [<parameters ...attributes.../>]
                    [<start strategy="None|managedobjecttypestrategy"/>]
                    [<stop ...attributes.../>]
                    [<kill ...attributes.../>]
                    [<ping ...attributes.../>]
                </control-overrides>]
                ...
          </managed-object>
           ...
    </managed-objects>
...
```

stop サブ要素の属性

次に stop サブ要素の属性を示します。

属性	必須	デフォルト値	説明
strategy	はい	デフォルトは、管理オブジェクトタイプによって異なります。詳細は、「stop サブ要素のストラテジ属性」を参照してください。	管理オブジェクトの停止アクションで BAS が使用するストラテジを指定します。

```
...
   <managed-objects>
          <managed-object ...attributes...>
                ...
                [<control-overrides>
                    [<parameters ...attributes.../>]
                    [<start strategy="None|managedobjecttypestrategy"/>]
                    [<stop strategy="None|managedobjecttypestrategy"/>]
                    [<kill ...attributes.../>]
                    [<ping ...attributes.../>]
                </control-overrides>]
                ...
          </managed-object>
           ...
    </managed-objects>
...
```

kill サブ要素の属性

次に kill サブ要素の属性を示します。

属性	必須	デフォルト値	説明
strategy	はい	デフォルトは、管理オブジェクトタイプによって異なります。詳細は、「kill サブ要素のストラテジ属性」を参照してください。	管理オブジェクトの強制終了アクションで BAS が使用するストラテジを指定します。

```
...
   <managed-objects>
          <managed-object ...attributes...>
                ...
                [<control-overrides>
                    [<parameters ...attributes.../>]
                    [<start strategy="None|managedobjecttypestrategy"/>]
                  [<stop strategy="None|managedobjecttypestrategy"/>]
                    [<kill strategy="None|managedobjecttypestrategy"/>]
                    [<ping ...attributes.../>]
                </control-overrides>]
                ...
          </managed-object>
           ...
    </managed-objects>
...
```

ping サブ要素の属性

次に ping サブ要素の属性を示します。

属性	必須	デフォルト値	説明
strategy	はい	デフォルトは、管理オブジェクトタイプによって異なります。詳細は、「ping サブ要素のストラテジ属性」を参照してください。	管理オブジェクトの ping アクションで BAS が使用するストラテジを指定します。

```
...
   <managed-objects>
          <managed-object ...attributes...>
                ...
                [<control-overrides>
                    [<parameters ...attributes.../>]
                    [<start strategy="None|managedobjecttypestrategy"/>]
                  [<stop strategy="None|managedobjecttypestrategy"/>]
                    [<kill strategy="None|managedobjecttypestrategy"/>]
                    [<ping strategy="None|managedobjecttypestrategy"/>]
                </control-overrides>]
                ...
          </managed-object>
           ...
    </managed-objects>
...
```

start サブ要素のストラテジ属性

次に start サブ要素の strategy 属性として有効な値を管理オブジェクト タイプごとに示します。

管理オブジェクト タイプ	デフォルトの開始ストラテジ	説明
process	run-process	■ run-process - プロセスを開始するコマンドを実行します。 ■ null - 管理オブジェクトの開始アクションは実行されません。
custom-javascript	jscript-control	■ jscript-control - 開始アクションを実行する JavaScript を起動します。 ■ null - 管理オブジェクトの開始アクションは実行されません。
custom-executable	run-custom-executable	■ run-custom-executable - 開始アクションを実行するプロセスを実行します。 ■ null - 管理オブジェクトの開始アクションは実行されません。
ordered-group	該当なし	この管理オブジェクトのタイプは構成上のみ - アクション ストラテジは適用されません。
redundancy-group	該当なし	この管理オブジェクトのタイプは構成上のみ - アクション ストラテジは適用されません。
state-proxy	該当なし	この管理オブジェクトのタイプは構成上のみ - アクション ストラテジは適用されません。
osagent	run-process	■ run-process - プロセスを開始するコマンドを実行します。 ■ null - 管理オブジェクトの開始アクションは実行されません。
apache-process	run-apache-process	■ run-apache-process - Apache プロセスを開始するコマンドを実行します。 ■ null - 管理オブジェクトの開始アクションは実行されません。
ots	run-process	■ run-process - プロセスを開始するコマンドを実行します。 ■ null - 管理オブジェクトの開始アクションは実行されません。
tibco	run-process	■ run-process - プロセスを開始するコマンドを実行します。 ■ null - 管理オブジェクトの開始アクションは実行されません。
partition	start-partition	■ start-partition - パーティション管理リソースを開始します。 ■ always-success - 内部使用のみ。 ■ null - 管理オブジェクトの開始アクションは実行されません。

stop サブ要素のストラテジ属性

次に stop サブ要素の strategy 属性として有効な値を管理オブジェクト タイプごとに示します。

管理オブジェクト タイプ	デフォルトの停止ストラテジ	説明
ordered-group	該当なし	この管理オブジェクトのタイプは構成上のみ - アクション ストラテジは適用されません。
redundancy-group	該当なし	この管理オブジェクトのタイプは構成上のみ - アクション ストラテジは適用されません。
state-proxy	該当なし	この管理オブジェクトのタイプは構成上のみ - アクション ストラテジは適用されません。
process	stop-process-term-signal	UNIX：すべてのストラテジで、プロセスに sigterm 信号を送信します。 Windows の場合 ■ stop-process-term-signal - プロセスに Ctrl+c イベントを送信します。 ■ stop-gui-process - プロセスが所有するすべてのウィンドウに WM_CLOSE メッセージを送信します。 ■ stop-bes-launcher - 内部使用のみ。 ■ stop-process-windows-c-exit - プロセスが C ランタイムの exit 関数を呼び出します。 ■ stop-process-windows-exit-process - プロセスが Windows の ExitProcess 関数を呼び出します。 ■ kill-signal - プロセスに対して "kill -9" またはプラットフォーム固有の同等の機能を実行します。 ■ null - 管理オブジェクトの停止アクションは実行されません。
custom-javascript	jscript-control	■ jscript-control - 停止アクションを実行する JavaScript を起動します。 ■ null - 管理オブジェクトの停止アクションは実行されません。
custom-executable	run-custom-executable	■ run-custom-executable - 停止アクションを実行するプロセスを実行します。 ■ null - 管理オブジェクトの停止アクションは実行されません。
osagent	stop-process-windows-c-exit	■ stop-process-windows-c-exit - UNIX：プロセスに sigterm 信号を送信します。 Windows：プロセスが C ランタイムの exit 関数を呼び出します。 ■ kill-signal - プロセスに対して "kill -9" またはプラットフォーム固有の同等の機能を実行します。 ■ null - 管理オブジェクトの停止アクションは実行されません。

管理オブジェクトタイプ	デフォルトの停止ストラテジ	説明
apache-process	shutdown-apache-process	■ shutdown-apache-process - Apache プロセスをシャットダウンするコマンドを実行します。 ■ null - 管理オブジェクトの停止アクションは実行されません。
ots	stop-process-term-signal	■ stop-process-term-signal - UNIX：プロセスに sigterm 信号を送信します。 Windows：プロセスに Ctrl+c イベントを送信します。 ■ kill-signal - プロセスに対して "kill -9" またはプラットフォーム固有の同等の機能を実行します。 ■ null - 管理オブジェクトの停止アクションは実行されません。
tibco	stop-process-term-signal	■ stop-process-term-signal - UNIX：プロセスに sigterm 信号を送信します。 Windows：プロセスに Ctrl+c イベントを送信します。 ■ kill-signal - プロセスに対して "kill -9" またはプラットフォーム固有の同等の機能を実行します。 ■ null - 管理オブジェクトの停止アクションは実行されません。
partition	stop-partition	■ stop-partition - パーティション管理リソースを停止します。 ■ null - 管理オブジェクトの停止アクションは実行されません。

ping サブ要素のストラテジ属性

次に ping サブ要素の strategy 属性として有効な値を管理オブジェクト タイプごとに示します。

管理オブジェクト タイプ	デフォルトの ping ストラテジ	説明
process	check-pid	■ check-pid - プロセス ID がシステムプロセステーブルに存在するかどうかをチェックし、その開始時間を確認します。 ■ jscript-ping - ping アクションを実行する JavaScript を起動します。この JavaScript は、成功した場合は 0、失敗した場合は 1 を返す必要があります。 ■ ping-custom-executable - ping アクションを実行するプロセスを実行します。このプロセスは、成功した場合は 0、失敗した場合は 1 を返す必要があります。
ordered-group	該当なし	この管理オブジェクトのタイプは構成上のみ - アクション ストラテジは適用されません。
redundancy-group	該当なし	この管理オブジェクトのタイプは構成上のみ - アクション ストラテジは適用されません。
state-proxy	該当なし	この管理オブジェクトのタイプは構成上のみ - アクション ストラテジは適用されません。
custom-javascript	jscript-ping	ping アクションを実行する JavaScript を起動します。この JavaScript は、成功した場合は 0、失敗した場合は 1 を返す必要があります。
custom-executable	ping-custom-executable	ping アクションを実行するプロセスを実行します。 このプロセスは、 成功した場合は 0、 失敗した場合は 1 を返す必要があります。
osagent	check-osagent	osagent 管理オブジェクトの configuration.xml のプロパティエントリに基づいて、システムプロセステーブルにプロセス ID が存在するかどうかと、 osagent 管理リソースを使用できるかどうかをチェックします。
apache-process	ping-apache-process	Apache の httpd.pid ファイルにあるプロセス ID がプロセス テーブルにあるかどうかをチェックします。
ots	check-pid	プロセス ID がシステム プロセス テーブルに存在するかどうかをチェックし、 その開始時間を確認します。
tibco	ping-tibco-server	システム プロセス テーブルにプロセス ID が存在するかどうかをチェックし、 次に Tibco 管理リソースを使用できるかどうかをチェックします。
partition	ping-partition	システム プロセス テーブルにプロセス ID が存在するかどうかをチェックし、次に管理リソースにパーティション状態を要求します。

kill サブ要素のストラテジ属性

次に kill サブ要素の strategy 属性として有効な値を管理オブジェクト タイプごとに示します。

管理オブジェクト タイプ	デフォルトの強制終了ストラテジ	説明
ordered-group	該当なし	この管理オブジェクトのタイプは構成上のみ - アクション ストラテジは適用されません。
redundancy-group	該当なし	この管理オブジェクトのタイプは構成上のみ - アクション ストラテジは適用されません。
state-proxy	該当なし	この管理オブジェクトのタイプは構成上のみ - アクション ストラテジは適用されません。
process	kill-signal	■ kill-signal - プロセスに対して "kill -9" またはプラットフォーム固有の同等の機能を実行します。 ■ null - 管理オブジェクトの強制終了アクションは実行されません。
custom-javascript	null	■ jscript-control - 強制終了アクションを実行する JavaScript を起動します。 ■ null - 管理オブジェクトの強制終了アクションは実行されません。
custom-executable	null	■ run-custom-executable - 強制終了アクションを実行するプロセスを実行します。 ■ null - 管理オブジェクトの強制終了アクションは実行されません。
osagent	kill-signal	■ kill-signal - プロセスに対して "kill -9" またはプラットフォーム固有の同等の機能を実行します。 ■ null - 管理オブジェクトの強制終了アクションは実行されません。
apache-process	kill-signal	■ kill-signal - プロセスに対して "kill -9" またはプラットフォーム固有の同等の機能を実行します。 ■ null - 管理オブジェクトの強制終了アクションは実行されません。
ots	kill-signal	■ kill-signal - プロセスに対して "kill -9" またはプラットフォーム固有の同等の機能を実行します。 ■ null - 管理オブジェクトの強制終了アクションは実行されません。
tibco	kill-signal	■ kill-signal - プロセスに対して "kill -9" またはプラットフォーム固有の同等の機能を実行します。 ■ null - 管理オブジェクトの強制終了アクションは実行されません。
partition	kill-signal	■ kill-signal - プロセスに対して "kill -9" またはプラットフォーム固有の同等の機能を実行します。 ■ null - 管理オブジェクトの強制終了アクションは実行されません。

第 48 章

管理オブジェクト time-rules 要素

time-rules は managed-object 要素のサブ要素です。time-rules 要素内で、管理オブジェクトのプロセスの開始時間または停止時間を定義する cron 形式のルールをいくつかの time-rule サブ要素に設定して追加できます。

time-rules サブ要素

サブ要素	説明
time-rules	設定された時間ルールが実行される前の管理オブジェクトの基本状態を記述する属性を含みます。

```
...
   <managed-objects>
          <managed-object ...attributes...>
           ...
               [<time-rules attributes >
                     ...
               </time-rules>]
                     ...
          </managed-object>
           ...
    </managed-objects>
...
```

time-rules サブ要素の属性

次に time-rules サブ要素の属性を示します。

属性	必須	デフォルト値	説明
base-state	はい	Running （実行中）	管理オブジェクトの基本動作状態。 設定時間帯にだけ MO を実行するように制御する場合は、base-state を Stopped に設定し、 MO を実行する時間帯を指示する 1 つ以上の time-rule サブ要素を設定します。 逆に、 設定時間帯に MO を停止するように制御する場合は、 base-state を Running に設定し、 MO を停止する時間帯を指示する 1 つ以上の time-rule サブ要素を設定します。

```
...
<managed-objects>
      <managed-object ...attributes...>
            ...
           [<time-rules base-state="Running|Stopped">
                        ...
           </time-rules>]
            ...
      </managed-object>
       ...
</managed-objects>
...
```

time-rule サブ要素

デフォルトでは、設定の time-rules サブ要素は、作成された順序で、最初に入力された time-rule から評価されます。time-rule を評価する順序は、作成後にいつでも再調整できます。time-rule アクションは、ハブのホストのシステム クロックに基づいて実行されます。

time-rule の属性では、cron 形式を使用して、開始時間、停止時間、および指定した時間の後に time-rule の適用を継続する期間を指定します。有効な cron 構文の詳細については、使用している UNIX システムの crontab のマニュアル ページを参照してください。

メモ 監視されているが、管理されていない MO に対して指定した時間ルールは無視されます。

サブ要素	説明
time-rule	MO に対する特定のタイム イベントのルールを定義する属性を含みます。

```
...
   <managed-objects>
         <managed-object ...attributes...>
          ...
              <time-rules ... >
                   [<time-rule attributes>
                          ...
                   </time-rule>]
              </time-rules>
                   ...
         </managed-object>
          ...
    </managed-objects>
...
```

time-rule サブ要素の属性

次に time-rule サブ要素の属性を示します。

属性	必須	デフォルト値	説明
rule-name	いいえ	設定した時間ルールに基づいて、[Management Console Time Rules] タブに表示されるルール名（"Stop for 1 minute, at midnight, every day" など）を自動的に生成します。自動的に生成されたルール名は configuration.xml に表示されません。	[Management Console Time Rules] タブに表示される名前をカスタマイズしてルールに割り当てます。
type	はい	standard-time-rule	時間ルールのタイプ。現在サポートされているタイプは、デフォルトの standard-time-rule だけです。
state	はい	■ スケジュールされたタスクの時間ルールの場合：Running ■ その他の時間ルールの場合：Stopped	このルールに対して設定された時間に有効になる MO の実行状態。スケジュールされたタスクの時間ルールの場合、有効な state は Running だけです。設定階層内の MO に追加された時間ルールの場合、有効な値は Stopped または Running です。
hour	はい	0	cron 形式で指定した、開始時刻（時）です。時間は 0 ～ 23 の整数値で表されます。この値の範囲を表すエントリ、または特定の時間のカンマ区切りリストを表すエントリは、すべて有効です。たとえば、「12-14」は午後 0 時、午後 1 時、および午後 2 時を表し、「12,14,15」は午後 0 時、午後 2 時、および午後 3 時を表します。アスタリスク（*）は、任意の時間を表すワイルドカードとして有効です。hour のデフォルト値は 0（午前 0 時）です。
minute	はい	0	cron 形式で指定した、開始時刻（分）です。分は 0 ～ 59 の整数値で表されます。この値の範囲を表すエントリ、または特定の分のカンマ区切りリストを表すエントリは、すべて有効です。たとえば、「0-29」は、1 時間の最初の 30 分を表します。アスタリスク（*）は、1 分ごとを表すワイルドカードとして有効です。minute のデフォルト値は 0（1 時間の最初の分）です。

属性	必須	デフォルト値	説明
day	はい	*	day-of-week が指定されていない場合は、cron 形式で指定した、ルールを実行する日。日は 1 ～ 31 の整数値で表されます。値の範囲を表すエントリ、または値のカンマ区切りリストを表すエントリが有効です。たとえば、「1,15」というエントリは、月の 1 日と 15 日にだけルールが実行されることを意味します。デフォルトはアスタリスク（*）で、これは、ルールが毎日実行されることを意味します。
day-of-week	はい	*	cron 形式で指定した、ルールを実行する曜日。0 ～ 6 の整数値が有効です。値の範囲を表すエントリ、または値のカンマ区切りリストを表すエントリが有効です。たとえば、「1-5」は平日（月曜日～金曜日）にだけルールが実行されることを意味し、「0,6」は週末（土曜日と日曜日）にだけルールが実行されることを意味します。デフォルトはアスタリスク（*）で、これは、ルールが毎日実行されることを意味します。曜日の値は次のとおりです。 ■ 0=Sunday ■ 1=Monday ■ 2=Tuesday ■ 3=Wednesday ■ 4=Thursday ■ 5=Friday ■ 6=Saturday
month	はい	*	cron 形式で指定した、ルールを実行する月。値の範囲を表すエントリ、または値のカンマ区切りリストを表すエントリが有効です。たとえば、「1,3,5」という month エントリは、1 月、3 月、5 月の指定した時間にだけルールが実行されることを意味します。デフォルトはアスタリスク（*）で、これは、ルールが毎月実行されることを意味します。月の値は次のとおりです。 ■ 1=January ■ 2=February ■ 3=March ■ 4=April ■ 5=May ■ 6=June ■ 7=July ■ 8=August ■ 9=September ■ 10=October ■ 11=November ■ 12=December

属性	必須	デフォルト値	説明
duration	いいえ	1	このルールの duration-unit の繰り返し数を指定します。たとえば、duration-unit がデフォルトの minute である場合、1 というデフォルト値は、ルールがスケジュール時間に 1 分間適用されることを意味します。
duration-unit	いいえ	minute	ルールを適用する期間の単位。有効な値は minute、hour、day、または month です。

```
...
<managed-objects>
          <managed-object ...attributes...>
                ...
                 <time-rules ... >
                     [<time-rule rule-name="<name>"

                      type="standard-time-rule"
                      state="Running|Stopped"
                      hour="*|0-23"
                      minute="*|0-59"
                      day="*|1-31"
                      month="*|1-12"
                      day-of-week="*|0-6"
                      duration-unit="minute|hour|day|month">
                      </time-rule>
                          ...

                 </time-rules>
                ...
          </managed-object>
        ...
</managed-objects>
...
```

time-rule のサンプル

次に、time-rule サブ要素を使って MO に時間ルールを追加する例を示します。

- 次の time-rule は、毎日午後 10 時 30 分に 3 分間 MO を停止します。

  ```
  <time-rule state="Stopped" minute="30" hour="22" duration="3" duration-
  unit="minute"/>
  ```

- 次の time-rule は、常に MO を停止します。これは、複数のルールをセットで使用し、後の time-rule ルールで MO を実行する時間を指定する場合の最初のルールとして使用できます。

  ```
  <time-rule state="Stopped" minute="0" hour="0" duration="24" duration-
  unit="hour"/>
  ```

- 次の time-rule は、平日（月曜日～金曜日）の午前 9 時から午後 5 時まで MO を実行します。

  ```
  <time-rule state="Running" minute="0" hour="9" day-of-week="1-5" duration="8"
  duration-unit="hour"/>
  ```

- 次の time-rule は、1 月 1 日に MO を実行しません。

  ```
  <time-rule state="Stopped" minute="0" hour="0" day="1" month="1"
  duration="24" duration-unit="hour"/>
  ```

- 次の time-rule は、12 月 24 日から 31 日まで MO を実行しません。

  ```
  <time-rule state="Stopped" minute="0" hour="0" day="24-31" month="12"
  duration="24" duration-unit="24"/>
  ```

第 49 章

設定プロパティの使い方

Borland AppServer （BAS）には、インストール時に値が設定される組み込みの定義済みプロパティが付属します。さらに、configuration.xml ファイルで、設定全体のプロパティを定義できます。設定プロパティは、configuration.xml ファイル内で文字列置換メカニズムとして使用されます。

properties 要素

configuration 要素の properties サブ要素で設定プロパティを定義します。複数の properties サブ要素を同じ configuration.xml ファイルに記述できます。

```
<configuration>
    ...
    <properties>
        <property ...attributes.../>
        [<property ...attributes.../>]
        [...more properties...]
    </properties>
    <properties>
        <property ...attributes.../>
        [<property ...attributes.../>]
        [...more properties...]
    </properties>
    ...
</configuration>
```

プロパティの定義

各設定全体のプロパティは、properties 要素の property サブ要素内に記述する必要があります。これには次の構文が使用されます。

```
<property name="name" value="value"/>
```

ここで、*name* は定義するプロパティの名前、*value* はその値です。引用符（""）は必須です。

定義されたプロパティの参照

一度プロパティを定義すると、${*property-name*}（*property-name* は定義したプロパティの名前）などの文字列置換構文を使用して、設定内の任意の場所でそのプロパティを参照できます。ハブは、プロパティの名前を検索し、${*property-name*} などの置換文字列のかわりにプロパティの値を使用します。

たとえば、次のプロパティを定義します。

```
<properties>
    <property name="install-root" value="c:/apache" />
    <property name="apache-command" value="apachect1" />
</properties>
```

これらのプロパティを適切な管理オブジェクト ブロック内で参照できます。

```
<managed-object type="process" />
    <process command="${install-root}/bin/apache2/${apache-command}">
```

ハブは、<process> の command 属性を次のように解釈します。

```
command="c:/apache/bin/apache2/apachect1"
```

install-root プロパティと apache-command プロパティの値が適切な場所に代入されます。

ほかのプロパティに基づくプロパティの定義

ほかの定義済みプロパティに基づいて、設定全体のプロパティ値を定義することもできます。たとえば、次のようになります。

```
<properties>
    <property name="appHome" value="c:/windows" />
    <property name="appDir" value="system32" />
    <property name="theApp" value="${appHome}/${appDir}/notepad" />
</properties>
```

設定で theApp プロパティを使用すると、これをハブが c:/windows/system32/notepad として処理します。

メモ プロパティやマクロは、ネストできません。たとえば、${appHome${appDir}} は正しくありません。

ハブとエージェントのプロパティ

次のリストは、ハブとエージェントのプロパティの一部です。

プロパティ	説明
agent.heartbeat.startup.delay.max	管理オブジェクトの初期状態が、最初にハブをハートビートする前に計算されるのをエージェントが待機する最大時間を指定します。 デフォルトは 20（秒）です。
hub.heartbeat.misses.allowed	ハブで、 エージェントが停止していると判断するまでのハートビート間隔を指定します。 デフォルトは 2 です。 これは 2 回のハートビートの失敗は OK ですが、 3 回目のハートビートに失敗した場合、 ハブはエージェントが停止していると判断します。 デフォルトではハートビートの間隔は 30 秒に設定されています。
agent.to-hub.ior.file	設定されている場合、スレーブ エージェントは、IOR を使用します。 ハブに接続するためのこのプロパティ（"hub internal IOR"）で示されます。"hub internal IOR" は、スマート エージェントで、 rep-id IDL:borland.com/Enterprise/Management/Internal/HubInternal:1.0 と object name <hub-name>_InternalHub に登録されます。
agent.output.iors	このプロパティが true の場合、 SCU は管理 IOR を SCU の作業ディレクトリ（デフォルトでは、<install-dir>/var/domains/base）のファイル（<interface-name>.ior）に出力します。

組み込みの定義済みプロパティ

BAS には、インストール時に値が設定される組み込みのプロパティが付属します。これらのプロパティは、ユーザーが定義した設定全体のプロパティ内を含む、設定内で使用できます。

たとえば、次の記述があるとします。

```
<property name="theApp" value="c:/windows/system32/notepad" />
```

AGENT プロパティを使用すると、次のように記述できます。

```
<property name="theApp" value="${AGENT.env.windows.SystemRoot}/system32/
notepad" />
```

ハブとスレーブ エージェントが同じホストにない分散インストールでは、組み込みのプロパティを使用して、エージェントは認識しているがリモート ハブは認識していない情報（スレーブ エージェントがあるホスト名など）をやり取りできます。

メモ 組み込みの AGENT プロパティは、Java システムプロパティであり、<install_dir>/bin/scu.config、<install_dir>/var/domains/base/adm/properties/agent.config、agent.properties の各ファイルにあります。この Java システム プロパティはすべて設定プロパティの値として使用できます。

重要 6.5 より前のリリースの組み込みプロパティは引き続き有効ですが、このリリースから使用されなくなりました。組み込みプロパティの新しい形式と非推奨の形式を以下の表にリストします。

重要 組み込みの設定プロパティは、要素の値と属性の置き換えに使用できますが、AGENT、MO、%platform% の各プロパティは、name または agent の属性の値として使用できません。

メモ 以下の組み込みプロパティは変更できません。

プロパティ	6.5 で非推奨	説明
HUB.name	hub.name	" マスター ハブ " または " ローカル ハブ " の名前。デフォルトでは、ハブがインストールされているホストの名前です。

プロパティ	6.5 で非推奨	説明
AGENT.install.root	agent.root	BAS ハブ エージェント インストールのルート ディレクトリ。
AGENT.instance.root	agent.instance.root	エージェントのドメインのディレクトリ。たとえば、${AGENT.root}/var/domains/<domain-name> です。
AGENT.host	agent.host	エージェントがインストールされている物理リソースのホスト名。
AGENT.address	該当なし	エージェントのホストの IP アドレス。
AGENT.name	agent.name	エージェントの名前。デフォルトの AGENT.name 値はホスト名です。
AGENT.default.smartagent.port	agent.default.smartagent.port	エージェントの osagent がアクティブなローカル サブネット上のポート。
AGENT.default.smartagent.addr	agent.default.smartagent.addr	エージェントの osagent が動作するホストの IP アドレスまたはホスト名。
AGENT.env.windows.SystemRoot	agent.env.windows.SystemRoot	Windows SYSTEMROOT 環境変数の値。**Windows のみ。**
AGENT.env.windows.userprofile	agent.env.windows.userprofile	Windows USERPROFILE 環境変数の値。**Windows のみ。**
AGENT.env.unix.DISPLAY	agent.env.unix.DISPLAY	DISPLAY 環境変数の値。**UNIX のみ。**
AGENT.env.unix.TZ	agent.env.unix.TZ	TZ 環境変数の値。**UNIX のみ。**
AGENT.env.unix.LANG	agent.env.unix.LANG	LANG 環境変数の値。**UNIX のみ。**
AGENT.env.unix.LOCPATH	agent.env.unix.LOCPATH	LOCPATH 環境変数の値。**UNIX のみ。**
AGENT.env.unix.NLSPATH	agent.env.unix.NLSPATH	NLSPATH 環境変数の値。**UNIX のみ。**

プロパティ	6.5 で非推奨	説明
CONFIG.name	config.name	設定の名前。
CONFIG.path	config.path	configuration.xml ファイルへのパス（${AGENT.root}/var/domains/base/configurations/${CONFIG.name} など）。ハブとエージェントで有効です。

プロパティ	6.5 で非推奨	説明
MO.name	mo.name	管理オブジェクト name 属性の値。これは、各管理オブジェクトのコンテキスト内でのみ有効です。
MO.agent	mo.agent	管理オブジェクト agent 属性の値。これは、各管理オブジェクトのコンテキスト内でのみ有効です。

%platform% プロパティ

`%platform%` という特殊なプロパティは、プラットフォームのチェックに使用されます。ほかのプロパティはシステム上に任意のプロパティとして定義できますが、`%platform%` プロパティはネイティブに定義されます。このプロパティは次の値に解決されます。

- SunOS
- WinNT
- Win2000
- WinXP
- Win2003
- MacOS
- Linux
- AIX
- HPUX
- Windows
- UNIX

プロパティ式の使い方

プロパティの値が外部の状態に依存することがあります。たとえば、プロセスのコマンド属性を定義する場合、プロセスの開始に使用されるコマンドは、そのプロセスをホストするオペレーティング システムによって異なる可能性があります。BAS には、このようなチェックを実行する**プロパティ式**と呼ばれるメカニズムがあります。

プロパティ式は、`property` 要素の `value` 属性で使用します。プロパティ式を使用すると、定義した条件に基づいてプロパティの値を設定できます。これにより、設定の可搬性をさまざまな面で容易に向上させることができます。プロパティ式には次の 2 種類があります。

- `if` ステートメント
- `switch` ステートメント

if ステートメント

`if` ステートメントをプロパティの `value` 属性として使用できます。基本の構文は次のとおりです。

```
if(condition) {
    "result"
    }
    [else {
    "result"
    }]
```

ここで、***condition*** は、`true` か `false` かを評価する有効な演算子で結果が求められる比較、最初の ***result*** は、`condition` が `true` と評価された場合のプロパティの値です。条件が `false` と評価された場合は、`else` 演算子を使って 2 番目の結果を提供できます。

`if` ステートメントで使用される条件は、演算子を使ってプロパティの値を評価するためにだけ使用されます。有効な演算子は次のとおりです。

- `=`（等しい）
- `!`（等しくない）
- `#`（含む）

単項演算子 `?` もあります。この演算子は、プロパティ キーの存在を調べるだけで、比較する値を必要としません。

また、&（AND）演算子および |（OR）演算子を使用して、1 回のチェックで複数の条件を使用することもできます。

たとえば、次の条件があるとします。

```
${AGENT.default.smartagent.port} = "14000"
```

これは、エージェントのスマート エージェント ポートが `14000` の場合は `true`、14000 ではない場合は `false` になります。

メモ すべての値を引用符（""）で囲む必要があることに注意してください。

この条件を使用する `if` ステートメントの例は次のとおりです。

```
if(${AGENT.default.smartagent.port} = "14000") {
    "default_port"
    } else {
    "user_defined_port"
}
```

適切な XML 構文を使用して、この `if` ステートメントをプロパティの `value` 属性のかわりに使用できます。

```
<property name="isDefault" value=
    "if(${AGENT.default.smartagent.port} = &quot;14000&quot;) {
        &quot;default_port&quot;
        } else {
        &quot;user_defined_port&quot;
    }" />
```

警告 `if` ステートメントは XML ファイルにだけ配置されるため、正しく解析されるように、適切な XML 文法を使用する必要があります。したがって、引用符はネストできません。条件の値や結果を引用符で囲む場合は、`&quot;` を使用する必要があります。

基本的な if ステートメントの例

- プロパティ `agent.name` が存在するかどうかをチェックするステートメント

  ```
  if(${AGENT.name} ?) {
      "named_agent"
      } else {
      "no_name"
      }
  ```

  プロパティ値として使用する場合

  ```
  <property name="isNamed" value=
      "if(${AGENT.name} ?) {
          &quot;named_agent&quot;
          } else {
          &quot;no_name&quot;
          }" />
  ```

- `agent.host` プロパティに文字列 "`nett2`" が含まれるかどうかをチェックするステートメント

  ```
  if(${AGENT.name} # "nett2") {
      "bad_name"
      } else {
      "ok"
      }
  ```

  プロパティ値として使用する場合

  ```
  <property name="nameCheck" value=
      "if(${AGENT.name} # &quot;nett2&quot;) {
          &quot;bad_name&quot;
          } else {
          &quot;ok&quot;
          }" />
  ```

- &（AND）演算子と `%platform%` プロパティを使用して、プラットフォームが Windows 2003 であり、`AGENT.host` プロパティが存在するかどうかをチェックするステートメント

  ```
  if(${%platform%} = "Win2003" & ${AGENT.host} ?) {
      "ready"
      }
  ```

プロパティ値として使用する場合

```
<property name="sysCheck" value=
    "if(${%platform%} = &quot;Win2003&quot; & ${AGENT.host} ?) {
        &quot;ready&quot;
        }" />
```

ネストされた if ステートメント

デフォルトの構文の *result* を別の if ステートメントにすることで、if ステートメントをネストできます。最後に解決された if ステートメントの結果がプロパティの値になります。構文は次のとおりです。

```
if(condition) {
    if(condition) {
        "result"
        }
        [else {
        "result"
        }]
    [else {
    "result"
    }]
```

解決できる限り、ほかの *result* フィールドもネストされた if ステートメントで置換し続けることができます。

ネストされた if ステートメントの例

プラットフォームが UNIX かどうかをチェックしたうえ（UNIX ではない場合は "other_platform" を返す）、ユーザー名が "root" かどうかによって異なるコマンドを返すステートメント

```
if ($%platform% = "Unix") {
    if(${user.name} = "root" {
        "rm -rf *"
        } else {
        "rm -rf ." }
    else {
    "other_platform"
    }
```

プロパティ値として使用する場合

```
<property name="unixCommand" value=
    "if(${user.name} = &quot;root&quot; {
        &quot;rm -rf *&quot;
        } else {
        &quot;rm -rf .&quot; }
    else {
    &quot;other_platform&quot;
    }" />
```

switch ステートメント

また、switch ステートメントをプロパティの *value* 属性として使用できます。これを使用して、いくつかのケースを評価し、それぞれに異なる結果を提供したり、どのケースにも評価されない場合のデフォルト値を提供することができます。構文は次のとおりです。

```
switch(${property-name}) {
    case "value" : "result";
    [case "value" : "result";
    ...]
    [default : "default-result"]
}
```

ここで、*value* は、*property-name* という名前のプロパティがとり得る値、*result* は、switch ステートメントを使って定義されるプロパティの値です。

たとえば、次の switch ステートメントは、さまざまなオペレーティング システムをチェックして、コマンドの値を設定します。

```
switch (${%platform%}) {
    case "Win2003" : "dd";
    case "WinXP" : "dd -x";
    case "Win2000" : "";
    default : "dd1"
}
```

プロパティ値として使用する場合は、次のようになります。

```
<property name="command" value=
    "switch (${%platform%}) {
        case &quot;Win2003&quot; : &quot;dd&quot;;
        case &quot;WinXP&quot; : &quot;dd -x&quot;;
        case &quot;Win2000&quot; : &quot;&quot;;
        default : &quot;dd1&quot;
    }" />
```

第 50 章

Borland 管理シェル（BMSH）の API 仕様

このドキュメントは、Borland 管理シェルの API 仕様です。

BMSH オブジェクトには、次の 4 種類があります。

- 事前にインスタンス化されたオブジェクト
  - fso
  - bes
- ユーザーがインスタンス化するオブジェクト
  - Hub
  - PropertyContext
  - HttpPage
- 静的オブジェクト
  - BmsExec
  - Dom4j
  - BmsState
- ファクトリ オブジェクト - ほかのオブジェクト（通常は、事前にインスタンス化されたオブジェクト）によって生成されるオブジェクト
  - PropertyFileSO
  - TextStream

fso

fso オブジェクトは、ファイル システムとやり取りするためのメソッドとプロパティを持ちます。プロパティ ファイル（.properties）内の値を取得および設定するために便利なメソッドがあります。BMSH 内で変数 fso として事前にインスタンス化されています。

プロパティの概要

String　classpath [get]

説明：
クラスパスを戻します。

String jars [get]

説明：
クラスパスにある jar を戻します。

String pwd [get]

説明：
現在の作業ディレクトリを戻します。

メソッドの概要

void copyFile(String sourceFile, String destinationFile, boolean overwrite)

説明：
ファイルをコピー元からコピー先にコピーし、必要に応じてコピー先を上書きします。

パラメータ：
sourceFile - コピー元ファイル
destinationFile - コピー先ファイル
overwrite - true の場合は、コピー先ファイルが存在すれば上書きします。

戻り値：
なし

boolean createArchive(String arFile, String archiveAs, String fileToArchive)

説明：
arFile という名前でアーカイブを作成します。同じ名前のアーカイブ ファイルが存在すれば、置き換えます。

パラメータ：
arFile - 作成するアーカイブ ファイルの名前。このファイルには、archiveAs というエントリ名で fileToArchive が格納されます。マニフェストが作成されます。このルーチンは、アーカイブ ファイルを更新しません。ファイルを 1 つだけ含む .dar ファイルなどのアーカイブに使用が限定されます。
archiveAs - アーカイブされたファイルのパス。
fileToArchive - アーカイブするファイルのパス。

戻り値：
成功した場合は true、そうではない場合は false。

void createFolder(String folderName)

説明：
ディレクトリ パスを作成します。フル パスを作成します。java.io.File(folderName).mkdirs() を使用します。

パラメータ：
folderName - 作成するフォルダのパス。

戻り値：
なし

TextStreamcreateTextFile(String filename, boolean overwrite)

説明：
テキストファイルを作成し、ファイルを書き込むことができる TextStream オブジェクトを戻します。注意： メソッドが戻ったときにファイルは開いた状態です。

パラメータ：
filename - テキスト ファイルの名前。
overwrite - true の場合は、既存のテキスト ファイルを上書きします。

戻り値：
TextStream インターフェイス

boolean deleteFile(String filename)

説明：
ファイルを削除します。

パラメータ：
filename - 削除するファイル。

戻り値：
成功した場合は true。

boolean deleteFolder(String folderName)

説明：
ディレクトリを再帰的に削除します。

パラメータ：
folderName - （再帰的に）削除するフォルダのパス。

戻り値：
削除された場合は true、そうではない場合は false。

boolean exists(String filename)

説明：
ファイルが存在するかどうかを戻します。

パラメータ：
filename - 確認するファイル。

戻り値：
ファイルが存在する場合は true。

boolean extractFile(String jarFile, String jarEntry, String destDir, String usePath)

説明：
jar（または zip）ファイルからファイルを抽出します。

パラメータ：
jarFile - 抽出するファイルを含む jar ファイルの名前。
jarEntry - jarFile 内の抽出するファイルの名前。重要：jarEntry のファイル セパレータおよび大文字と小文字がアーカイブの内容と正確に一致していないと、エントリが検出されません。
destDir - jarFile の抽出先のディレクトリの名前。抽出時、この場所が jarEntry パスのルートになります。別の場所に抽出する場合は、usePath=="useDestDir" を設定します
usePath - ファイルの抽出先を選択します。usePath == "useDestDir" を設定すると、ファイルは destDir に抽出されます。usePath == "useZipPath" を設定するか、未定義の場合、ファイルのパスは、destDir をルートとする jarFile と同じになります。

戻り値：
成功した場合は true、そうではない場合は false。

int fileLength(String filename)

説明：
引数で指定されたファイルの長さを戻します。

パラメータ：
filename - 長さを検出するファイルの名前。

戻り値：
ファイルが見つからない場合は -1、そうではない場合はファイルの長さ（0 以上の整数）。

String[] files(String dir)

説明：
ディレクトリ内のファイルをリストします。リストはアルファベット順です。ファイルをフィルタリングしたり、再帰的に検索する場合は、fso.findFiles() を使用します。

パラメータ：
dir - リストするファイルを含むディレクトリ。

戻り値：
ファイルを格納する String[]。

String[] findFiles(String dir, String matchName, boolean recurse)

説明：
mathchName パラメータと一致するファイルとディレクトリを検索します。検索は dir パラメータから始まり、recurse パラメータが true の場合は再帰的に実行されます。大文字と小文字は区別されません。最適化は行われないため、数ギガバイトのディスクの検索には時間がかかります。リストはアルファベット順です。戻されるファイルにはフル パスが付きます。

パラメータ：
dir - リストするファイルを含むディレクトリ。再帰的なサブフォルダの検索はここから始まります。
matchName - 検索するファイル名の文字列。ワイルドカードを使用できます。
recurse - true の場合は、サブディレクトリを再帰的に検索します。

戻り値：
ファイルとディレクトリを格納する String[]。

String[] folders(String dir)

説明：
ディレクトリのサブディレクトリをリストします。リストはアルファベット順です。

パラメータ：
dir - リストする子フォルダを含むルート ディレクトリ。

戻り値：
子フォルダを格納する String[]。

String getBaseName(String path) (obsolete)

説明：
指定されたフル パスのベース ファイル名を戻します。
LiveConnect:java.io.File(path).getName(); を使用してください。

パラメータ：
path - ファイルのフル パス。

戻り値：
ベース ファイル名。

String getProperty(String propFile, String propName)

説明：
指定したプロパティを取得します。これは便利なルーチンです。同じファイルに何度もアクセスする場合は、まず fso.loadPropFile() でプロパティ ファイルを開き、PropertiesSO のメソッドを使ってプロパティにアクセスし、変更を加える方法が便利です。

パラメータ：
propFile - プロパティ ファイル。
propName - 取得するプロパティ。

戻り値：
プロパティ値または null。

boolean isDirectory(String filename)

説明：
ファイルがディレクトリかどうかを戻します。

パラメータ：
filename - 確認するファイル。

戻り値：
ファイルがディレクトリの場合は true。

Propertiesloadproperties(String fileName) (experimental)

説明：
java.util.Properties オブジェクトをプロパティ ファイルからロードします。

パラメータ：
fileName - プロパティ ファイル。

戻り値：
ファイルからロードされた java.util.Properties オブジェクト。

PropertyFiloadPropFile(String filename)
leSO

説明：
スクリプト可能な Java プロパティ ファイルをロードします。スクリプト可能なプロパティ ファイルには変更時のコメントが保持されます。このメソッドで開かれるプロパティ ファイルは、メソッドが戻ったときに開いた状態です。

パラメータ：
filename - ロードするプロパティ ファイル。

戻り値：
プロパティ ファイルに対するスクリプト可能なインターフェイス。

boolean moveFile(String currentName, String newName)

説明：
ファイルを移動します。ファイル名の変更に使用します。

パラメータ：
currentName - 現在のファイル名。
newName - 移動後のファイル名。

戻り値：
成功した場合は true。

TextStreamopenTextFile(String filename, int ioMode, boolean okToCreate)

説明：
ファイルを TextStream として開くためのファクトリ メソッド。例を次に示します。

```
var stream = fso.openTextFile( "fileFoo", 1, true)
while( !stream.atEndOfStream) {
   line = stream.readLine();
   print(line);
}
stream.close();
```

パラメータ：
filename - 開くファイルの名前。
ioMode - 読み取り専用の場合は 1、書き込み専用の場合は 2。
okToCreate - ファイルが存在しない場合に作成する場合は true。

戻り値：
TextStream オブジェクト。

boolean setProperty(String filename, String propname, String propvalue)

説明：
指定されたプロパティを設定します。これは便利なルーチンです。同じファイルに何度もアクセスする場合は、まず fso.loadPropFile() でプロパティ ファイルを開き、PropertiesSO のメソッドを使ってプロパティにアクセスし、変更を加える方法が便利です。このルーチンは、プロパティ ファイル内のコメントを維持します。ファイルにプロパティが存在しない場合は追加されます。例を次に示します。

```
fso.setProperty("vbroker.properties","vbroker.security.disabled",
"false");
```

パラメータ：
filename - プロパティ ファイル。
propname - 設定するプロパティ。
propvalue - プロパティの新しい値。プロパティが存在しない場合は追加されます。

戻り値：
成功した場合は true。

bes

bes オブジェクトは、**ローカルな** BES インストールに作用する機能へのスクリプト可能なインターフェイスです。これらの機能を使用するためにログインする必要はありません。このオブジェクトは、BMSH によって事前に bes としてインスタンス化されています。

プロパティの概要

String installRoot [get]

説明：
BES インストールのルート フォルダ プロパティ。これは、bmsh.exe と scu.exe がある BES /bin フォルダの親の名前になります。

String tempFolder [get]

説明：
一時ファイルのルート パス。

メソッドの概要

String getConsolePort()

説明：
コンソールの管理ポートを戻します。

戻り値：
コンソールのポート。

String getManagementPort(String domain)

説明：
ドメインの管理ポートを戻します。

戻り値：
ポート。

void setConsolePort(String port)

説明：
コンソールの管理検索ポートを設定します。

戻り値：
コンソールのポート。

void setManagementPort(String port, String domain)

説明：
指定されたドメインの管理ポートを設定します。

パラメータ：
port - ポートの新しい値。
domain - ドメイン名。

Hub

HubSO オブジェクトは、Borland 管理ハブとやり取りするためのコア メソッドとプロパティを持ちます。${installRoot}/bin/bscript/autoload/hub_ext.js には、設定と管理オブジェクトのための便利な API セットが含まれています。

メソッドの概要

void addArchivedConfiguration(String archiveFile)

説明：
ハブに設定を追加します。

void addConfiguration(String file)

説明：
ハブに設定を追加します。

パラメータ：
file - 設定に追加される configuration.xml ファイルのフル パス（スクリプトに対してローカル）。

String addConfigurationFromTemplate(String templatePath, String configName, Object templateProperties)

説明：
テンプレートに基づいて設定をハブに追加します。

パラメータ：
templatePath - テンプレート ファイルのパス。絶対パスではない場合は、installRoot/var/templates/configurations フォルダからの相対パスとみなされます。
configName - 新しい設定の名前。空の場合は、テンプレートのルート ファイル名が使用されます。
templateProperties - テンプレート プロパティの上書きを含む java.util.Properties オブジェクト。テンプレートをインスタンス化するときに使用されます。プロパティ名については、テンプレートの「プロパティ」の記述を参照してください。

戻り値：
成功した場合は「Success」、そうではない場合はエラー メッセージを含む文字列。

boolean connect(String osAgentPort, String hubName)

説明：
ハブに接続します。ハブに接続してから Hub API を呼び出す必要があります。これにより、VisiBroker ORB が作成されます。

パラメータ：
osAgentPort - osagent 管理ポート。
hubName - 管理ハブの名前。

戻り値：
接続が成功した場合は true、そうではない場合は false。

boolean corbalocConnect(String host, String port)

説明：
ハブに接続します。ハブに接続してから Hub API を呼び出す必要があります。これにより、VisiBroker ORB が作成されます。

パラメータ：
host - ハブが存在するホスト
port - 管理ハブの IIOP ポート

戻り値：
接続が成功した場合は true、そうではない場合は false。

void disconnect()

説明：
ハブとの接続を切断します。接続を切断する必要はありませんが、明示的に切断すると、接続のタイムアウトを待つより早くハブのリソースを解放できます。接続を切断すると、ハブは通知を送信しなくなります。TODO: 名前とパスワードでログイン。

String[] getAgents(String config)

説明：
設定に属するエージェントを取得します。

パラメータ：
config - 設定名。

戻り値：
設定に含まれる各エージェントを表す文字列からなる JavaScript 配列。

String[] getAgentsBelongingToHub()

説明：
ハブに属するエージェントの名前を取得します。

戻り値：
ハブに属する各エージェントを表す文字列からなる JavaScript 配列。

Object getConfiguration(String cfgName)

説明：
設定オブジェクトを取得します

戻り値：
Java Configuration オブジェクト。

ScriptablegetConfigurations()

説明：
ハブによって管理される設定を取得します。例を次に示します。

```
var h = new Hub();
h.connect(42424)
var cfgs = h.getConfigurations();
 for( c in cfgs )
  print(cfgs[c]);
```

戻り値：
ハブが管理している各設定を表す文字列からなる JavaScript 配列。

String getHost()

説明：
ハブのホスト名を取得します。

戻り値：
ハブのホスト名。

String getName()

説明：
ハブの名前を戻します。

戻り値：
ハブ名を表す文字列。

Object getNamedAgent(String name)

説明：
エージェントへのインターフェイスを取得します。

戻り値：
Java ManagementAgent オブジェクト

String getStateName(int state)

説明：
管理オブジェクトの状態を示す整数値を文字列に変換します。通知や API のほかのメソッドでは、状態が整数として戻されます。

パラメータ：
state - 通知によって戻される状態値。

戻り値：
状態を表す文字列。

boolean iorConnect(String ior)

説明：
ハブに接続します。ハブに接続してから Hub API を呼び出す必要があります。これにより、VisiBroker ORB が作成されます。

パラメータ：
ior - ハブを表す IOR 文字列

戻り値：
接続が成功した場合は true、そうではない場合は false。

void removeConfiguration(String configName, boolean stopFirst)

説明：
ハブから設定を削除します。

パラメータ：
configName - 設定名を表す文字列。
stopFirst - 削除する前に設定を停止する（推奨）必要があることを示す boolean 型フラグ。

void updateConfiguration(String file)

説明：
ハブの設定を更新します。

パラメータ：
file - 更新される configuration.xml ファイルのフル パス（スクリプトに対してローカル）。

Object waitForEvent()

説明：
ハブ イベントを待機します。この呼び出しは、ハブからイベントを受け取るまで待機してブロックします。イベントのフィルタリングは呼び出し元で行う必要があります。次の例は、すべてのイベントを監視して表示します。

```
var hubName = "myHub";
var hub = new Hub();
hub.connect(42424, hubName);

while(1) {
    print("Sleeping... Waiting for notifications...");
    var event = hub.waitForEvent();
    print(event.toString());
}
```

戻り値：
Event オブジェクト

Object waitForStartEvent(int timeoutInSec)

説明：
開始イベントを待機します。これは、イベントの状態が「RUNNING（実行中）」の場合にだけ戻ること以外は、waitForEvent() と同じです。

パラメータ：
timeoutInSec - 開始を待機する時間。

戻り値：
Event オブジェクト

Object waitForStopEvent(int timeout)

説明：
ハブの開始イベントを待機します。これは、イベントの状態が「STOPPED（停止）」の場合にだけ戻ること以外は、waitForEvent() と同じです。

戻り値：
Event オブジェクト

PropertyContext

PropertyContext クラスは、プロパティ名に対応するプレースホルダーを含む文字列にプロパティ値を設定するために使用されるスクリプト可能なインターフェイスを表します。PropertyContext は、プロパティとその値を表す要素を含む Dom4j によって初期化されます。静的な create メソッドは、ルート ノード内のプロパティ要素、それらの名前、およびそれらの値を検索するための xpath 文字列を制御します。

メソッドの概要

void addProperties(Object propertiesContainingNode)

説明：
子要素を含むラップされた XML ノードを指定して、それらのプロパティを既存のプロパティに追加します。

パラメータ：
propertiesContainingNode -

static PropertyContext create(org.w3c.dom.Node root, String propElementXPath, String nameAttr, String valueAttr)

説明：
ルート org.w3c.dom.Node によって初期化される PropertyContext オブジェクトを作成します。渡されるノードが org.w3c.dom.Document の場合は、getDocumentElement() の結果が実際のノードとして使用されます。オプションのパラメータにより、プロパティ要素、その要素内のプロパティ名属性、およびその要素内のプロパティ値属性を検索するための xpath 文字列を制御できます。

パラメータ：
root - プロパティの検索に使用するルートノード。
propElementXPath - プロパティ値を保持する要素の検索に使用する xpath。
nameAttr - プロパティ名を表す属性の検索に使用されます。
valueAttr - プロパティ値を表す属性の検索に使用されます。

戻り値：
オプションの xpath 文字列を使ってルート ノードによって初期化された PropertyContext インスタンス。

Object getProperties()

説明：
基底のプロパティ コンテナをネイティブ Java Object にラップして戻します。

戻り値：
プロパティを含む java.lang.Object。

String getProperty(String property)

説明：
個別のプロパティ値を取得します

パラメータ：
property - プロパティの名前。

戻り値：
プロパティ値。プロパティが未定義の場合は null。

void properties() (experimental)

説明：

String setProperty(String property, String value)

説明：
個別のプロパティ値を設定します

パラメータ：
property - プロパティの名前。
value - プロパティの新しい値。

戻り値：
プロパティの元の値。プロパティが未定義の場合は null。

String substituteProperties(String source)

説明：
source 文字列内にある既知のプロパティを保存されている値に置き換えます。

パラメータ：
source - "${property.name}" 形式のプロパティを含む文字列。

戻り値：
プロパティ値が置換された後の文字列。

HttpPage

HttpPage オブジェクトは、Web ページ（HTTP URL）とやり取りするためのメソッドとプロパティを持ちます。例を次に示します。

```
var p = new HttpPage("http://www.google.com");
var e = p.getPage();
if( e != null ) {
   var s = fso.openTextFile("google.html",2,true);
   s.write(e);
   s.close();
} else {
   print("Page:" + p.getUrl() + " is NOT available.");
}
```

メモ

1. リダイレクト（HTTP 300）に続けて接続（HTTP 要求）が行われます。
2. 発生する可能性が最も高い例外は、オブジェクトの作成時に発生する MalformedURLException です。
3. HTTP 応答を待機するときのタイムアウトは制御できません。
4. ページに含まれる JavaScript は実行されません。
5. ページに含まれる URL リファレンスはサポートされません。たとえば、ページ上で参照される .jpeg は取得されません。

メソッドの概要

int getHttpResponseCode()

説明：
URL の HTTP 応答コードを戻します。

戻り値：
HTTP ヘッダーの HTTP 応答コードを示す正の値。ページを受信しなかった場合は負の値。

String getHttpResponseMessage()

説明：
URL の HTTP 応答メッセージを戻します。

戻り値：
HTTP ヘッダーの HTTP 応答メッセージ。ページを受信しなかった場合は null。

String getPage()

説明：
Web ページのコンテンツを戻します。ページに含まれるハイパーリンクは取得されません。たとえば、ハイパーリンクが .jpg を参照していても、ファイルは取得されません。JavaScript を使用したリダイレクトは実行されません。JavaScript は実行されません。

戻り値：
Web ページを表す文字列。ページを受信しなかった場合は null。

int getPageLength()

説明：
Web ページのコンテンツの長さを戻します。

戻り値：
ページのコンテンツの長さを示す正の値。ページを受信しなかった場合は負の値。

static Object httpRequest(String urlStr)

説明：
指定された URL に HTTP 要求を発行します。Java HttpURLConnection オブジェクトを戻します。HttpURLConnection API は http://java.sun.com/j2se/1.4.1/docs/api/ で定義されています。

戻り値：
HttpURLConnection。接続が失敗した場合は null。

static int ping(String urlStr, int desiredRespCode)

説明：
URL を ping して、応答コードが引数に一致するかどうかを確認します。これは、HttpPage オブジェクトをインスタンス化しないで Web ページを ping できる便利なメソッドです。Web ページを BMS 管理オブジェクトとして ping するように設計されています。成功の内容をより詳細に把握する必要がある場合は、HttpPage オブジェクトをインスタンス化し、そのメソッドを使用する必要があります。

パラメータ：
urlStr - ping する HTTP URL。
desiredRespCode - 目的の HTTP 応答コード。

戻り値：
0 - httpResponseCode == httpRespCode の場合（ページが実行されている）。
1 - httpResponseCode != httpRespCode の場合（ページが実行されていない）。

BmsExec

BMS エージェントの機能を使ってプロセスの開始、停止、ping を行うメソッドがあります。

メソッドの概要

static boolean killProcess(String processToken)

説明：
pid/starttime 形式の引数文字列で指定したプロセスを終了します。この形式は、startProcess から戻される形式と同じです。プロセスの強制終了が成功した場合は true が戻され、そうではない場合は false が戻されます。プロセスを終了する方法としては最も直接的です。通常の手順にしたがってプロセスを停止するには、stopProcess() を使用します。

戻り値：
プロセスが強制終了された場合は true、そうではない場合は false。

static boolean pingProcess(String processToken)

説明：
pid/starttime 形式の引数文字列で指定したプロセスが実行中かどうかを戻します。この引数の形式は、startProcess から戻される形式と同じです。

戻り値：
プロセスが実行中の場合は true、そうではない場合は false。

static String startProcess(String cmdLine)

説明：
プロセスを開始し、pid/starttime 形式で PID と開始時刻を戻します。プロセスを開始できなかった場合は、null が戻されます。開始時刻を取得できなかった場合は、値として -1 が使用されます。

パラメータ：
cmdLine - 実行するコマンドライン。

戻り値：
PID と開始時刻を含む文字列。

static String startProcessX(Object processNode, Object propertyContext)

説明：
Dom4j processNode 引数で参照されるプロセスを開始します。

パラメータ：
processNode - 設定ファイルにある Dom4j <process> ノード。
propertyContext - <process> ノードにある ${} プロパティ置換値を含む PropertyContext ノード。

戻り値：
プロセスを開始し、pid/starttime 形式で PID と開始時刻を戻します。プロセスを開始できなかった場合は null。

static boolean stopProcess(String process, String strategy)

説明：
pid/starttime 形式の引数文字列で指定したプロセスの停止を試みます。この形式は、startProcess から戻される形式と同じです。プロセスの停止が成功した場合は true が戻され、そうではない場合は false が戻されます。これでプロセスを停止できない場合は、killProcess() を使用します。

パラメータ：
process - 停止するプロセスの PID と開始時刻。
strategy - 次のいずれかです。
Windows の場合：
"ctrl-c" - プロセスに ctrl-c を送信します。
"wm-close" - プロセスが所有するすべてのウィンドウに WM_CLOSE を送信します。
"c-exit" - C ランタイムの exit() を呼び出します。
"exit-process" - Windows ExitProcess() API を呼び出します。kill と同等です。
"bes-launcher-exit" - BES 起動プログラム内の API を呼び出します。これにより、定義されている Java クリーンアップ メソッドが呼び出され、終了します。
UNIX の場合：
どのストラテジを使用しても、プロセスに SIGTERM シグナルが送信されます。

戻り値：
プロセスが停止された場合は true、そうではない場合は false。

Dom4j

Dom4j オブジェクトには、XML ファイルの解析と保存に便利な静的メソッドがあります。Dom4j API については、http://www.dom4j.org/apidocs/index.html を参照してください。この API は、JavaScript の LiveConnect 機能を使って parseAXmlFile オブジェクトから戻されるオブジェクトとともに使用できます。例を次に示します。

```
var dom = Dom4j.parseXmlFile("logConfiguration.xml");

var rn = dom.selectSingleNode("/log4j:configuration/root");
var an = rn.selectSingleNode("appender-ref[@ref='SOCKET']");

 if( an == null ) {
       an = rn.addElement("appender-ref");
       an.addAttribute("ref","SOCKET");
}

Dom4j.save(dom,"logConfiguration.xml");
```

メソッドの概要

static String getAttribute(String xmlFile, String attrXPath)

説明：
XML ファイルから 1 つの属性値を取得します。1 つの属性値を取得する場合は便利なルーチンです。

パラメータ：
xmlFile - 使用するファイルの名前。
attrXPath - 設定する属性の xpath。

戻り値：
属性値。

static Object parseXmlFile(String xmlFile)

説明：
XML ファイルを解析し、Dom4j 文書を戻します。文書はネイティブな Dom4j 文書です。`normalized()` で標準化されています。

戻り値：
- org.dom4j.Document。

static boolean save(Object node, String xmlFile)

説明：
dom のノードをファイルに保存します。"pretty" XML 形式で保存されます。

パラメータ：
node - dom ノード。文書ノードを指定できます。
xmlFile - ノードの保存先ファイルの名前。

戻り値：
boolean - 保存された場合は `true`、そうではない場合は `false`。

static boolean setAttribute(String xmlFile, String attrXPath, String attrValue)

説明：
XML ファイルを開き、属性を設定して、ファイルを保存します。1 つの属性値を設定する場合は便利なルーチンです。

パラメータ：
xmlFile - 使用するファイルの名前。
attrXPath - 設定する属性の xpath。
attrValue - 属性に設定する値。

戻り値：
boolean - 保存された場合は `true`、そうではない場合は `false`。

BmsState

`state` オブジェクトは、管理オブジェクトのいくつかの状態を表すクラスです。

プロパティの概要

int STATE_DEPENDENCY_DOWN [get]

説明：
グループの依存関係が停止しています。

int STATE_ERROR_STARTING [get]

説明：
開始時にエラーがありました。

int STATE_ERROR_STOPPED [get]

説明：
エラーによってエンティティが停止しています。

int STATE_ERROR_STOPPING [get]

説明：
エラーによってサービスが停止しています。

int STATE_ERROR_WAIT_START [get]

説明：
開始の待機中にエラーが発生しました。

int STATE_ERROR_WAIT_STOP [get]

説明：
停止の待機中にエラーが発生しました。

int STATE_MALFUNCTIONING [get]

説明：
動作不良の状態です。

int STATE_RESTARTING [get]

説明：
（再開の要求によって）再開しようとしています。

int STATE_RUNNING [get]

説明：
起動して実行中です。

int STATE_STARTING [get]

説明：
開始しようとしています。

int STATE_STOPPED [get]

説明：
完全に停止しています。

int STATE_STOPPING [get]

説明：
停止しようとしています。

int STATE_WAITING_TO_RESTART [get]

説明：
再開を待機しています。

int STATE_WAITING_TO_START [get]

説明：
開始を待機しています。

int STATE_WAITING_TO_STOP [get]

説明：
停止を待機しています。

PropertyFileSO

PropertyFileSO オブジェクトには、.properties ファイルにアクセスするためのメソッドとプロパティがあります。PropertyFileSO は、.properties ファイル内のコメント（#）を維持する点で、java.util.Properties オブジェクトとは異なります。例を次に示します。

```
var pf = fso.loadPropFile("foo.properties");
var val = pf.getValue(propname);
pf.setValue(propname,"propValue");
pf.write()
pf.close()
```

1 つのプロパティだけを取得または設定する場合は、fso.get/setProperty() の使用をお勧めします。

メソッドの概要

void close()

説明：
プロパティ ファイルを閉じます。

String getValue(String propname)

説明：
プロパティ値を取得します。

パラメータ：
propname - 取得するプロパティの名前。

void setValue(String propname, String propvalue)

説明：
プロパティ値を設定します。

パラメータ：
propname - 取得するプロパティの名前。
propvalue - プロパティに設定する値。

void write()

説明：
プロパティ ファイルを書き込みます。行った変更を維持するには、プロパティ ファイルを書き込む必要があります。

TextStream

TextStream オブジェクトには、テキスト ファイルの読み書きに使用するメソッドとプロパティがあります。これは fso のファクトリ オブジェクトです。fso.openTextFile() または fso.createTextFile() によって生成されます。

プロパティの概要

boolean atEndOfStream [get]

説明：
ファイル ストリームの終端に到達したことを示します。例を次に示します。

```
while( !stream.atEndOfStream )
  line = readline();
```

メソッドの概要

void close()

説明：
ストリームを閉じます。

String readAll()

説明：
ファイルのすべてをバッファに読み取ります。

String readLine()

説明：
ファイルの次の行をバッファに読み取ります。

boolean reset()

説明：
ストリームを最初の文字位置にリセットします。

戻り値：
ストリームが最初の文字を指している場合は true。

void write(String text)

説明：
テキストをストリームの末尾に書き込みます。ストリームは書き込みモードで開かれている必要があります。fso.openTextStream() を参照してください。

パラメータ：
text - ストリームに書き込むテキスト。

void writeLine(String text)

説明：
テキストをストリームに書き込み、末尾に CR/LF を付加します。ストリームは書き込みモードで開かれている必要があります。

パラメータ：
text - ストリームに書き込むテキスト。

索引

記号

数字

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

あ

い

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

て

と

な

に

ね

は

ひ

ふ

へ
ヘルプ