開発者ガイド

Borland VisiBroker® for .NET™ 7.0

Borland®
Excellence Endures™

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

2006 年 5 月 11 日初版発行
著者：Borland Software Corporation
発行：ボーランド株式会社
PDF

目次

第 1 章

VisiBroker for .NET の概要

Borland VisiBroker for .NET 製品は，実行時環境と一連の開発者ツールを備え，Microsoft .NET ランタイムから J2EE および CORBA サーバーへの高パフォーマンスの接続性を実現します。この製品を使用して .NET Framework 用に開発したアプリケーションは，柔軟性，相互運用性および安全性に富むプロトコルである IIOP を通じて，サーバー側の異種コンポーネントにアクセスできます。

重要 VisiBroker for .NET は，以前のリリースでは *Janeva* と呼ばれていました。*Janeva* という名前は，例，コマンド，パラメータ，クラス名，プロパティ，および UI 要素の中に引き続き数多く現れます。この開発者ガイドでは，これらの要素を指す場合に *Janeva* という名前を使用しています。

VisiBroker のマニュアル

このマニュアルに加えて，VisiBroker には次のマニュアルがあります。

- *Borland VisiBroker 概要*— VisiBroker の機能の概要について説明します。
- *Borland VisiBroker インストールガイド*— VisiBroker をネットワークにインストールする方法について説明します。このマニュアルは，Windows または UNIX 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を対象としています。
- *Borland VisiBroker セキュリティガイド*— VisiSecure for VisiBroker for Java および VisiBroker for C++ など，VisiBroker のセキュリティを確保するための Borland のフレームワークについて説明しています。
- *Borland VisiBroker VisiTime ガイド*— Borland による OMG Time Service 仕様の実装について説明します。
- *Borland VisiBroker VisiNotify ガイド*— Borland による OMG 通知サービス仕様の実装について説明します。通知メッセージフレームワークの主な機能として，特に Quality of Service (QoS) のプロパティ，フィルタリング，および Publish/Subscribe Adapter（PSA）の使用方法が記載されています。
- *Borland VisiBroker VisiTransact ガイド*— Borland による OMG Object Transaction Service 仕様の実装および Borland Integrated Transaction Service コンポーネントについて説明します。

- *Borland VisiBroker VisiTelcoLog ガイド* — Borland による OMG Telecom Log Service 仕様の実装について説明します。
- *Borland VisiBroker GateKeeper ガイド* — Web ブラウザやファイアウォールによるセキュリティ制約の下で，VisiBroker GateKeeper を使用して，VisiBroker のクライアントがネットワークを介してサーバーとの通信を確立する方法について説明します。
- *Borland VisiBroker for C++ 開発者ガイド* — C++ による VisiBroker アプリケーションの開発方法について記載されています。Visibroker ORB の設定と管理，およびプログラミングツールの使用方法について説明します。また，IDL コンパイラ，Smart Agent，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，OAD，QoS，およびインターフェースリポジトリについても説明します。
- *Borland VisiBroker for C++ API リファレンス* — VisiBroker for C++ に付属するクラスとインターフェースについて説明します。
- *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記載されています。Visibroker ORB の設定と管理，およびプログラミングツールの使用方法について説明します。また，IDL コンパイラ，Smart Agent，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），Quality of Service（QoS），およびインターフェースリポジトリについても説明します。
- *Java API ドキュメント* — VisiBroker for Java で提供されるクラスおよびインターフェースについての説明が記載されています。
- *VisiBroker for .NET API ドキュメント* — VisiBroker for .NET に付属するクラスとインターフェースについて説明します。

通常，マニュアルにアクセスするには，VisiBroker とともにインストールされるヘルプビューアを使用します。ヘルプは，スタンドアロンのヘルプビューアからアクセスすることも，VisiBroker コンソールからアクセスすることもできます。どちらの場合も，ヘルプビューアを起動すると独立したウィンドウが表示されるため，このウィンドウからヘルプビューアのメインツールバーにアクセスしてナビゲーションや印刷を行ったり，ナビゲーションペインにアクセス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 VisiBroker ブックとリファレンス文書の目次，完全なインデックス，および包括的な検索を実行できるページがあります。

マニュアルの表記規則

このマニュアルでは，特別なテキストを示すために次の表の書体と記号を使用します。

表 1.1　マニュアルの表記規則

表記規則	用途
italic	新規の用語およびマニュアル名に使用されます。
`computer`	サンプルのコマンドラインおよびコード。
bold computer	テキスト中の太字は，ユーザーが入力する情報を示します。コードサンプル中の太字は，重要な文を強調表示します。
< >	ユーザーまたはアプリケーションが指定する情報（変数など）。
[ ]	省略可能な項目。
...	繰り返しが可能な直前の引数。
|	二者択一の選択。
〔〕	キーボード上のキー。例：〔*Esc*〕を押して終了します。

Borland サポートへの連絡

ボーランド社は各種のサポートオプションを用意しています。それらにはインターネット上の無償サービスが含まれており，大規模な情報ベースを検索したり，他の Borland 製品ユーザーからの情報を得ることができます。さらに Borland 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。

ボーランド社のサポートサービスの詳細は，次の弊社 Web サイトにアクセスしてください。

http://www.borland.com/devsupport

お住まいの地域を選択できるようになっています。

世界共通の Borland サポートに関しては，次のサイトにアクセスしてください。

http://www.borland.com/devsupport/contacts

ボーランド社のサポートへの連絡にあたっては，次の情報を用意してください。

- 名前
- 会社名およびサイト ID
- 電話番号
- ユーザー ID 番号（米国のみ）
- オペレーティングシステムとそのバージョン（たとえば，Windows 2000 Server）
- Borland 製品名とそのバージョン（たとえば，VisiBroker 7.0）
- 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

次のオンラインソースから自由に情報を入手できます。

ワールドワイドウェブ http://www.borland.com/jp/

オンラインサポート http://support.borland.com（ユーザー ID が必要）

Listserv 電子ニュースレター（英文）を購読する場合は，次のサイトに用意されているオンライン書式を使用してください。

http://www.borland.com/contact/listserv.html

次の URL にあるボーランドのインターナショナルリストサーバーもご利用になれます。

http://www.borland.com/contact/intlist.html

Web サイト

定期的に www.borland.com/jp/ をチェックしてください。VisiBroker 製品チームによるホワイトペーパー，競合製品の分析，FAQ の回答，サンプルアプリケーション，最新ソフトウェア，最新のマニュアル，および新旧製品に関する情報が掲載されます。

特に，次の URL をチェックすることをお勧めします。

- http://www.borland.com/downloads/index.html（更新されたソフトウェアおよびその他のファイル）
- http://www.borland.com/techpubs（マニュアル）
- http://community.borland.com（英語，開発者向けの弊社 Web ベースニュースマガジンがあります）

Borland ニュースグループ

VisiBroker を対象とした数多くのニュースグループに参加できます。

次のサイトには，VisiBroker と Borland 製品を対象としたユーザーニュースグループがあります。

http://www.borland.com/newsgroups

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

VisiBroker for .NET モデルの概要

この章では，VisiBroker for .NET コンポーネントを紹介し，VisiBroker for .NET を利用してアプリケーションの相互運用を可能にする技術について説明します。

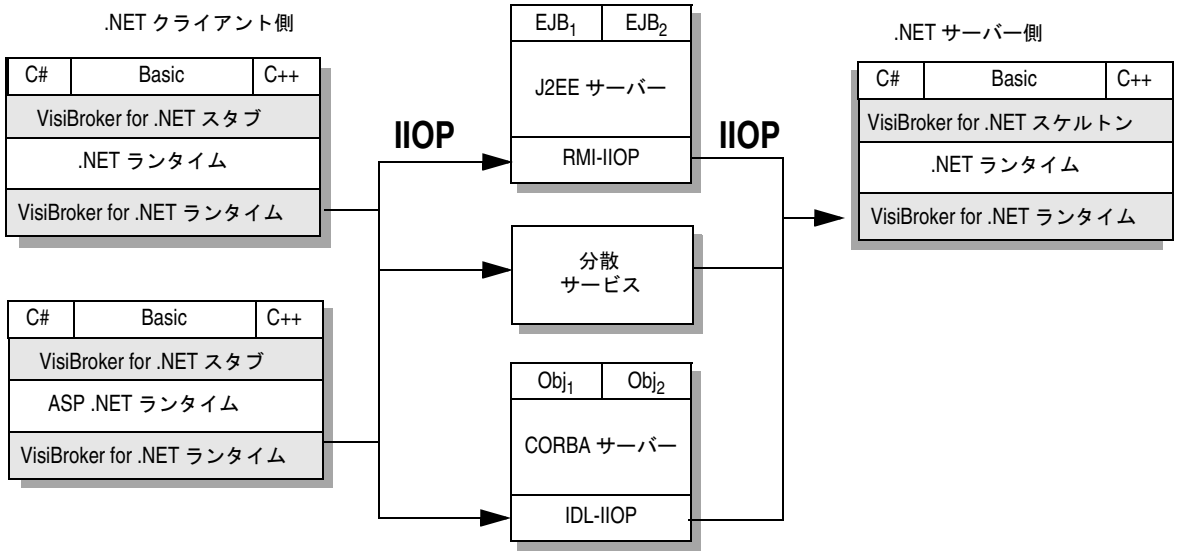
VisiBroker for .NET の概要

VisiBroker for .NET 製品は，Microsoft .NET ランタイムと J2EE および CORBA コンポーネントとの高性能な接続を実現します。この製品を使用すると，拡張性に優れたセキュアで相互運用可能な通信プロトコルである IIOP を介して異種サーバー側コンポーネントへのアクセスが可能な，.NET Framework 向けのマネージクライアント側アプリケーションとサーバー側アプリケーション（および ASP.NET アプリケーション）を構築できます。

図 2.1 は，VisiBroker for .NET を使用したアプリケーションの配布を示しています。図の左側と右側に 2 つの .NET アプリケーション環境が示されています。上側はスタンドアロン .NET クライアントアプリケーションの実行を，下側は ASP.NET ホストアプリケーションの実行を示します。図の中央部分は，J2EE および CORBA サーバー環境を示します。

VisiBroker for .NET が提供する機能は，クライアントスタブ，サーバースケルトン，および VisiBroker for .NET ランタイムとして，影付きの領域に示されています。J2EE および CORBA サーバー環境の領域には影が付いていないことに注意してください。これは，.NET 環境と相互運用する場合でも，VisiBroker for .NET をサーバー環境に配布する必要がないことを示しています。

図 2.1 VisiBroker for .NET のクライアント側配布図



VisiBroker for .NET 開発者ツール

VisiBroker for .NET を使用したアプリケーションが J2EE オブジェクトや CORBA オブジェクトのメソッドを呼び出すには，「スタブ」と「スケルトン」が必要です。スタブとスケルトンは，インターフェース固有のオブジェクトであり，異なる実行環境で実行されているオブジェクト上でアプリケーションがメソッドを呼び出せるように，パラメータマーシャリングおよび通信を提供します。VisiBroker for .NET 開発者ツールは，CORBA および J2EE サーバーオブジェクトとの通信に必要なスタブとスケルトンを生成するコンパイラを提供します。

J2EE ベースコンパイラは，Java Remote Method Invocation（RMI）ファイルに指定されたインターフェースを読み取ります。 CORBA ベースコンパイラは，Interface Definition Language（IDL）ファイルを読み取ります。生成されたスタブとスケルトンは，.NET Common Type System（CTS），つまり Microsoft の言語に依存しない型システムをターゲットとします。コンパイラは C# プログラミング言語でスタブとスケルトンを生成します。C# スタブやスケルトンは，いったん C# コンパイラによって Microsoft Common Intermediate Language（CIL）にコンパイルされると，どの .NET 互換言語からでも使用できるようになります。

VisiBroker for .NET ランタイム

VisiBroker for .NET ランタイムは，ライブラリとネットワークリソースの集合であり，エンドユーザーアプリケーションに統合されるため，アプリケーションでオブジェクトを検索して使用することができます。ランタイムは，リモートオブジェクトを使用する際に必要な CORBA および J2EE の基本 API を公開します。これらの API は，Microsoft Common Language System に準拠しているため，あらゆる .NET プログラミング言語からアクセスできます。

VisiBroker for .NET ランタイムは次の機能を備えています。

- **マーシャリング**：IIOP パケットの読み取りおよび書き込みのための，高性能で拡張性に優れたエンジン。
- **接続管理**：TCP 接続およびその他の通信リソースの割り当てを制御します。

- **セキュリティ**：広く採用されている標準の SSL，TLS，X.509 などに基づいて，メッセージを暗号化および認証します。このため，任意の J2EE 1.3 互換製品と安全に接続できます。
- **Objects-by-value**：さまざまな複合データ型をクライアントとサーバーの境界を超えて渡すことができます（J2EE 1.3 製品の場合）。
- **呼び出しコンテキストプロパゲーション**：IIOP パケットをシステムレベルデータで補完できます。
- **ポータブルインターセプタ**：ユーザーレベルデータまたはシステムレベルデータで IIOP パケットを拡張するための機能。これは，OTS 仕様と XA 仕様に基づいて分散トランザクションをサポートする製品では特に重要な機能です。相互運用可能なトランザクションのサポートは J2EE 1.3 のオプションです。そのため，このサポートは一部の J2EE ベンダーによってのみ提供されます。

VisiBroker for .NET の機能

VisiBroker for .NET 製品は次の機能を備えています。

- **ハイパフォーマンス**：VisiBroker for .NET は，クライアント／サーバーネットワークに IIOP を使用することで，バイナリデータフォーマット処理を提供します。
- **ステートフルサービス**：VisiBroker for .NET は，完全な分散オブジェクトモデルを提供します。これは，任意のサーバー側コンポーネントおよび任意のライフサイクル要件をサポートします。
- **高度なセキュリティ**：VisiBroker for .NET では，最新のセキュリティ標準に基づく暗号化，認証，および権限がすべてサポートされています。
- **複合データ型のサポート**：VisiBroker for .NET を使用すると，データ変換は自動的に処理されます。これにより，効率がよくなると同時に，エラーが少なくなります。
- **エンタープライズ QoS**：VisiBroker for .NET は，高度な QoS（Quality of Service）を標準で提供します。たとえば，次のものがあります。
  - **負荷分散**：要求を複数のサービスプロバイダに分散する。
  - **フォールトトレランス**：障害の発生したサーバーから代替プロバイダに要求をリダイレクトする。
  - **トランザクション**：アプリケーション境界を超えて 2 フェーズコミットトランザクションコンテキストを伝搬し，クライアント側でトランザクションを開始する。
  - **スケーラビリティ**：リソース使用率を最適化するために，接続の有効期間，接続の多重化などを制御する。

.NET とは

Microsoft .NET は，デスクトップアプリケーションと Web ベースアプリケーションの両方を構築する単一の手法を開発者に提供します。また，開発者はさまざまなプログラミング言語を使用しながら，同じツールやスキルを使用して多様なシステムに合わせたソフトウェアを開発できるため，互換性がないソフトウェアコンポーネントの共存を助け，アプリケーション間の競合を最小限に抑えることができます。

.NET Framework は，.NET アプリケーションを構築するための .NET Framework クラスライブラリ（FCL）と，アプリケーションを実行するための「共通言語ランタイム（CLR）」で構成されます。

共通言語ランタイム

共通言語ランタイム（CLR）は，アプリケーションを実行するための Microsoft .NET Framework 内のランタイムエンジンです。CLR は，「マネージ」アプリケーションに，言語間の統合，コードアクセスセキュリティ，オブジェクト存続期間管理，デバッグおよびプロファイリングサポートなどのサービスも提供します。

C#，C++，Microsoft Visual Basic，JScript など，ほとんどすべての言語で CLR 用のプログラムを作成できます。ランタイムは，コード作成の日常的な作業の多くを支援することにより，プログラミングを簡素化します。これらのタスクには，バグの原因となる可能性の高いメモリー管理，セキュリティ管理，エラー処理などが含まれます。

選択したプログラミング言語で書かれたコードは，.NET 言語でコンパイラを使用してコンパイルされるときに，「共通中間言語（CIL）」と呼ばれる，アセンブリに似た言語にコンパイルされます。CIL は，実行時に共通言語ランタイムによって実行可能コードにコンパイルされます。

.NET Framework クラスライブラリ

Windows アプリケーションを作成するプログラマは，Windows API，標準クラスライブラリ，およびそれぞれ独自の関数またはクラスに精通しています。.NET Framework クラスライブラリ（FCL）には機能のプリパッケージセットが組み込まれており，開発者はこれを使用して，共通言語ランタイムをターゲットとする型，メソッド，およびプロパティを使用するアプリケーションを構築できます。FCL で提供される型を使用してコードを作成することが，完全に相互運用可能な .NET アプリケーションを構築する最も確実な方法です。

FCL に組み込まれている機能には，次のものがあります。

- ASP.NET。Web アプリケーションおよび Web サービスの構築に使用します。
- Windows Forms。ユーザーインターフェース開発用です。
- ADO.NET。アプリケーションをデータベースに接続するのに使用します。

.NET リモート処理

伝統的に分散アプリケーションは，ネットワーク帯域幅を効率よく利用するバイナリプロトコル（たとえば IIOP）を使用した DCOM，CORBA，および Java RMI リモート処理技術をベースにしています。それとは対照的に，.NET の相互運用性はほとんどの場合，XML と SOAP に重点を置いています。

VisiBroker for .NET ランタイムは，IIOP の「マネージ」コードを実装し，.NET Framework に対応しています。VisiBroker for .NET を使用すると，開発者は .NET リモート処理スタイルの呼び出しを使用してリモートオブジェクト上のメソッドを検索して呼び出すことができるため，CORBA または Java RMI スタイルの呼び出しの作成方法を学習する必要はありません。アプリケーションの開発例については，第 3 章「VisiBroker for .NET クライアントアプリケーションの開発」と第 6 章「VisiBroker for .NET リモート処理サーバーの開発」を参照してください。

マネージアプリケーションとアンマネージアプリケーション

.NET Framework は，いわゆる「マネージ」アプリケーションと「アンマネージ」アプリケーションをサポートします。マネージアプリケーションとは，サポートされている .NET 言語（たとえば C#）を使用して作成したプログラムであり，Framework が規定する各種規則にしたがうものです。VisiBroker for .NET コードはすべてマネージコードです。

アンマネージアプリケーションとは，サポートされていない言語で作成されたプログラムで，.NET Framework 規則に完全にしたがうものではありません。このアプリケーション

は，ほとんどがレガシーアプリケーションですが，.NET Framework が提供するラッパープロセス内ではまだ実行することができます。

J2EE とは

Java 2 Platform，Enterprise Edition（J2EE）技術およびそのコンポーネントに基づくモデルでは，エンタープライズでの開発および配布が簡素化されます。J2EE プラットフォームは，インフラストラクチャを管理し，セキュアかつ強固で相互運用可能なビジネスアプリケーションの開発を可能にする Web サービスをサポートします。J2EE は，Enterprise JavaBeans，Java Servlets，Java Server Pages，およびデータベースアクセス用の JDBC などを実装するいくつかの API で構成されています。

J2EE では，標準化されたモジュラーコンポーネントを基に，それらのコンポーネントにサービスの完全セットを提供してアプリケーションの振る舞いのほとんどの詳細を扱うことにより，エンタープライズアプリケーションを簡素化します。J2EE では，既存のエンタープライズリソースと対話するために CORBA 技術を活用しています。

Enterprise JavaBeans

Enterprise JavaBeans（EJB）技術により，開発者はセッション Bean とエンティティ Bean という 2 つの異なるタイプの EJB コンポーネントを定義することで，企業内で有用な各種オブジェクトをモデル化できます。セッション Bean は，クライアントセッションに関連付けられた振る舞いを表します。エンティティ Bean は，データの集合（たとえば，データベース内のレコード）を表し，そのデータに対するオペレーションをカプセル化します。エンティティ Bean は永続的であるように意図されており，関連付けられたデータがある限り存続します。

クライアントアプリケーションは，厳密に標準化された EJBHome インターフェースと EJBObject インターフェースを使用して，リモートオブジェクト上でメソッドを検索しインスタンス化して呼び出すことにより，EJB と通信します。VisiBroker for .NET 開発者ツールを使用すると，Java RMI ソースから .NET 互換 C# 言語に至るまで，EJB との通信に必要なすべてのコードを生成できます。

Java RMI

Java Remote Method Invocation（RMI）技術により，開発者は完全に Java プログラミング言語環境で作業を行い，Java テクノロジーベースの分散アプリケーションを作成することができます。別の Interface Definition Language（IDL）やマッピングを覚える必要はありません。Internet Inter-Orb Protocol（RMI-IIOP）上で実行される Java RMI 技術により，J2EE プラットフォームに CORBA 分散コンピュータ機能が提供されます。

CORBA と同様に，RMI-IIOP は，何百ものベンダーやユーザーが OMG に参加して策定されたオープンスタンダードに基づいています。RMI-IIOP は，CORBA と同様に，通信プロトコルとして IIOP を使用します。IIOP は，C++，C，COBOL，および CORBA がサポートするその他の言語で作成されたアプリケーションコンポーネントが，Java プラットフォーム上で実行されるコンポーネントと通信できるようにすることにより，レガシーアプリケーションとプラットフォームの統合を容易にします。

CORBA とは

Common Object Request Broker Architecture（CORBA）は，アーキテクチャの仕様の 1 つで，分散アプリケーションが相互の詳細な通信要件を理解していなくても相互運用が可能です。CORBA は，何百ものベンダーやユーザーが Object Management Group に参加して策定されたオープンスタンダードに基づいています。

CORBA アプリケーションの一般的なモデルは，「ミドルウェア」，具体的には Object Request Broker（ORB）と呼ばれる中間レイヤを使用することを除いては，標準的なクライアント／サーバーモデルです。ORB とは，分散アプリケーション間の対話を管理するサービスの集合です。

Interface Definition Language

Interface Definition Language（IDL）は，リモートオブジェクトに対して CORBA インターフェースを記述するのに使用する記述言語です。実装言語（C++，Java，C#，またはその他の高水準言語）でクライアントスタブファイルとサーバースケルトンファイルを生成するには，IDL コンパイラを使用します。Object Management Group（OMG）は，各種プログラミング言語への言語マッピングの仕様を規定しています。VisiBroker for .NET は，C# で IDL の言語マッピングを提供します。詳細は，付録 B「IDL から C# へのマッピング」を参照してください。

IDL コードはどの IDE でも作成できますが，.NET 互換スタブとスケルトンを生成するには，IDL コンパイラが必要です。VisiBroker for .NET 開発者ツールを使用すると，組み込まれている IDL コンパイラの 1 つを使用して，IDL ファイルから C# クライアントスタブを生成できます。IDL コンパイラは，IDL ファイルを読み取り，スタブが含まれているクラスまたはその他のアドレス指定可能なオブジェクトを生成します。このスタブは，単純なメッセージ要求をアプリケーションから受け取る一般的なメソッドです。スタブはこの要求を，たとえばサーバー上のオブジェクト実装に渡し，応答を受信すると応答をデコードし，結果を呼び出し元のアプリケーション，つまりクライアントに戻します。

VisiBroker for .NET 機能は，Object Management Group（OMG）の CORBA 仕様（バージョン 2.4）に準拠しており，Borland AppServer との相互運用が可能です。

CORBA および .NET リモート処理

.NET の相互運用性はほとんどの場合，XML と SOAP に重点を置いています。これらの技術は，基本的にコネクションレスプロトコル（たとえば，HTTP）を使用できるなど，それぞれ利点はありますが，同期通信に関してはかなりの欠点があります。

このような場合は，IIOP などのピアツーピアプロトコルを使用した方が効率がよく安全です。さらに，同期クライアント／サーバー通信を使用すると，密に結合されたシステム間でバイナリデータを受け渡すことができるため，データセキュリティやリカバリ能力も向上します。

VisiBroker for .NET を使用すると，CORBA ミドルウェアにブートストラップし，クライアントコード内で CORBA スタイルの呼び出しまたは .NET リモート処理呼び出しのいずれかを使用して，オブジェクトを検索することができます。アプリケーションの開発例については，第 3 章「VisiBroker for .NET クライアントアプリケーションの開発」と第 6 章「VisiBroker for .NET リモート処理サーバーの開発」を参照してください。

Microsoft Visual Studio .NET のオプション

VisiBroker for .NET のインストール時に Microsoft Visual Studio .NET コンポーネントを選択した場合，VisiBroker for .NET アプリケーション開発を円滑に進められるように，Visual Studio 環境に特別な要素がいくつか追加されます。

Visual Studio で VisiBroker for .NET オプションを設定する手順は，次のとおりです。

1 ［Tools］メニューを選択して，［Options］をクリックします。

2 ［Borland VisiBroker for .NET］オプショングループを選択します。

次の設定オプションがあります。

- インストールディレクトリ：VisiBroker for .NET コンポーネントがインストールされるディレクトリ。
- JRE ディレクトリ : Java Runtime Environment がインストールされるディレクトリ。
- サポートされているファイル拡張子：サポートされている各ファイル拡張子ごとの VisiBroker for .NET コンパイラを表示します。
- デフォルト : 各 VisiBroker for .NET コンパイラのデフォルトのコマンドライン引数を設定。コマンドライン引数の説明については，付録 A「コンパイラオプション」を参照してください。

第 3 章

VisiBroker for .NET クライアントアプリケーションの開発

この章では，VisiBroker for .NET ランタイムを使用して J2EE および CORBA サーバーオブジェクトにアクセスできる，.NET クライアントアプリケーションを作成するための開発プロセスを紹介します。リモートオブジェクト上で呼び出しを行うための 3 つの異なる方法を，単純な例を使用して説明します。

VisiBroker for .NET では，分散オブジェクトと通信するクライアントアプリケーションを開発する方法として，.NET リモート処理，CORBA，J2EE の 3 つを用意しています。この 3 つの技術はそれぞれ，基本的に同じ手順を行う標準的な方法を定義しています。ミドルウェアのブートストラップ，リモートオブジェクトの検索とインスタンス化，およびオブジェクト上のメソッドの呼び出しです。

3 つの技術はそれぞれ，構文，API，およびプログラミングモデルが少し異なりますが，以下の例では，どの方法で作成する場合でも，それぞれについて同じ結果が得られるようになっています。

参照箇所

Microsoft 製品での開発経験があり，.NET リモート処理はすでに経験しているが分散技術は初めてという方の場合は，13 ページの「簡単な .NET リモート処理の例」から始めてください。J2EE に精通している開発者は 14 ページの「簡単な J2EE の例」から始め，CORBA に精通している開発者は 15 ページの「簡単な CORBA の例」から始めてください。

簡単な例

以下のセクションでは，ミドルウェアのブートストラップ，リモートオブジェクトの検索とインスタンス化，およびオブジェクト上でのメソッド呼び出しの方法を，3 つの簡単な例で示します。

簡単な .NET リモート処理の例

Microsoft 製品での開発経験があり，.NET リモート処理はすでに経験しているが分散技術は初めてという方には，喜んでいただけるでしょう。なぜなら，.NET リモート処理プ

ログラミングモデルを使用して，J2EE サーバーおよび CORBA サーバー上のオブジェクトと相互運用可能な .NET アプリケーションを開発することができるからです。

以下の 3 行のコードで，リモートオブジェクト MyServer のインスタンス化と，そのオブジェクトでの Method() の呼び出しがどれほど容易であるかがわかります。

```
static void Main(string[] args) {
   RemotingConfiguration.Configure ("MyApplication.exe.config");
   MyServerHome myServerHome = new MyServerHomeRemotingProxy();
   MyServer myServer = myServerHome.Create();
   myServer.Method();
}
```

サーバーとの接続の確立およびリモートオブジェクトの検索の情報は，16 ページの「.NET リモート処理の設定」に示されているとおり，XML 設定ファイルに入っています。

例を 1 行ずつ見ていきます。

最初の行では，.NET リモート処理が設定されている設定ファイルを指定します。

```
RemotingConfiguration.Configure ("MyApplication.exe.config");
```

コードの次の行では，「ファクトリ」オブジェクト MyServerHome をインスタンス化します。

```
MyServerHome myServerHome = new MyServerHomeRemotingProxy();
```

ファクトリオブジェクトは，リモートオブジェクトを検索して作成するためのルックアップメカニズムの 1 つです。メソッドを呼び出す実際のオブジェクトのインスタンスを検索して作成するために，最初にこのオブジェクトをルックアップします。

.NET にはオブジェクトの型を「narrow」（キャスト）するという概念はありません。オブジェクトの検索とその特定の型へのキャストを，一度に行います。

その次の行では，myServer というインスタンスを作成します。

```
MyServer myServer = myServerHome.Create();
```

これで，myServer のインスタンスでメソッドを呼び出すことができます。

```
myServer.Method();
```

実に簡単です。VisiBroker for .NET プロトコルを使用して .NET リモート処理を設定する方法については，16 ページの「.NET リモート処理の設定」を参照してください。

簡単な J2EE の例

VisiBroker for .NET は，EJB への呼び出しの作成に精通している開発者が，.NET アプリケーションで EJB への呼び出しを行えるようにするメソッドを提供します。

次の例について考えてみます。

```
static void Main(string[] args) {
   J2EE.Naming.Context root = new J2EE.Naming.InitialContext(args);
   string serverName = "location/of/my/server";
   object myServerHomeObject = root.Lookup(serverName);
   MyServerHome myServerHome = (MyServerHome)
      J2EE.Rmi.PortableRemoteObject.Narrow(myServerHomeObject,
         typeof(MyServerHome));
   MyServer myServer = myServerHome.Create();
   myServer.Method();
}
```

見てわかるとおり，この例は .NET リモート処理の例より少し複雑です。オブジェクトを検索するために必要な詳細を隠蔽する設定ファイルはありません。

例を 1 行ずつ見ていきます。

最初の行で，J2EE ネーミングサービスのルートコンテキストを設定します。

```
J2EE.Naming.Context root = new J2EE.Naming.InitialContext(args);
```

その次の 2 行で，サーバー上の EJBHome オブジェクト（myServerHomeObject）の位置を入れる変数を宣言し，オブジェクトをルックアップします。

```
string serverName = "location/of/my/server";
object myServerHomeObject = root.Lookup(serverName);
```

次の行では，myServerHomeObject をその型 MyServerHome に narrow（キャスト）します。

```
MyServerHome myServerHome = (MyServerHome)
   J2EE.Rmi.PortableRemoteObject.Narrow(myServerHomeObject, typeof(MyServerHome));
```

その次の行では，myServer というインスタンスを作成します。

```
MyServer myServer = myServerHome.Create();
```

最終的に，MyServer 上でメソッドを呼び出すことができます。

```
myServer.Method();
```

簡単な CORBA の例

VisiBroker for .NET は，CORBA への呼び出しの作成に精通している開発者が，.NET アプリケーションで CORBA オブジェクトへの呼び出しを行えるようにするメソッドを提供します。

以下の例は，その呼び出しを示しています。

```
static void Main(string[] args) {
   CORBA.ORB orb = CORBA.ORB.Init(args);
   CORBA.Object rootObject = orb.ResolveInitialReferences("NameService");
   CosNaming.NamingContextExt root =
      CosNaming.NamingContextExtHelper.Narrow(rootObject);
   string serverName = "location/of/my/server";
   CORBA.Object myServerHomeObject = root.ResolveStr(serverName);
   MyServerHome myServerHome = MyServerHomeHelper.Narrow(myServerHomeObject);
   MyServer myServer = myServerHome.Create();
   myServer.Method();
}
```

見てわかるとおり，この例は .NET リモート処理の例より少し複雑です。オブジェクトを検索するために必要な詳細を隠蔽する設定ファイルはありません。

例を 1 行ずつ見ていきます。

最初の行で，ORB を初期化します。

```
CORBA.ORB orb = CORBA.ORB.Init(args);
```

次の 2 行で，CORBA ネーミングサービスのルートコンテキストを取得します。

```
CORBA.Object rootObject = orb.ResolveInitialReferences("NameService");
CosNaming.NamingContextExt root = CosNaming.NamingContextExtHelper.Narrow(rootObject);
```

その次の 2 行で，サーバー上のファクトリオブジェクト（myServerHomeObject）の位置を入れる変数を宣言し，オブジェクトをルックアップします。

```
string serverName = "location/of/my/server";
CORBA.Object myServerHomeObject = root.ResolveStr(serverName);
```

次の行では，myServerHomeObject をその型 MyServerHome に narrow（キャスト）します。

```
MyServerHome myServerHome = MyServerHomeHelper.Narrow(myServerHomeObject);
```

その次の行では，myServer というインスタンスを作成します。

```
MyServer myServer = myServerHome.Create();
```

最終的に，MyServer 上でメソッドを呼び出すことができます。

```
myServer.Method();
```

.NET リモート処理の設定

このセクションでは，13 ページの「簡単な .NET リモート処理の例」の .NET の例で言及された設定ファイルの詳細を説明します。

.NET リモート処理の例を思い出してみましょう。

```
static void Main(string[] args) {
   RemotingConfiguration.Configure ("MyApplication.exe.config");
   MyServerHome myServerHome = new MyServerHomeRemotingProxy();
   MyServer myServer = myServerHome.Create();
   myServer.Method();
}
```

サーバーとの接続の確立およびリモートオブジェクトの検索の情報は，XML 設定ファイルに隠蔽されています。このテクニックは，.NET では宣言アクティベーションと呼ばれます。

この例の設定ファイルは，次のようになります。

```
<configuration>
   <system.runtime.remoting>
      <application name="MyApplication">
         <client>
            <wellknown type="MyServerHomeRemotingProxy, MyApplicationAssembly"
               url="janeva:corbaname:rir:#location/of/my/server/object"/>
         </client>
         <channels>
            <channel type="Janeva.Remoting.IiopChannel,
               Borland.Janeva.Runtime"/>
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

オブジェクト位置の指定

例の最初の行で MyServerHome をインスタンス化するときに，MyServerHomeRemotingProxy() で new 演算子を使用しました。呼び出しを行うオブジェクトを検索するために，例の設定ファイルでは wellknown 要素を使用しています。

```
<wellknown type="MyServerHomeRemotingProxy, MyApplication"
   url="janeva:corbaname:rir:#location/of/my/server/object"/>
```

MyServerHomeRemotingProxy は型名で，MyApplication はその型が定義されているアセンブリの名前です。

メモ MyServerHome は wellknown オブジェクトとして表されます。これは，SAO（Server Activated Object：サーバーアクティブ化オブジェクト）とも呼ばれます。CORBA サーバーオブジェクトまたは EJB サーバーオブジェクトは SAO として表すことができます。また，EJB は CAO（Client Activated Object：クライアントアクティブ化オブジェクト）として表すこともできます。詳細は，18 ページの「クライアントアクティブ化オブジェクトとサーバーアクティブ化オブジェクト」を参照してください。

.NET プログラミングモデルでは，URL を使用してリモートオブジェクトを検索する必要があります。URL は，次の 2 つの部分で構成されています。

- janeva: プロトコルプレフィクスは，設定ファイルの <channel> 要素で指定された IIOP チャネル（Janeva.Remoting.IiopChannel）を使用するようにアプリケーションに指示します。

  一般に，.NET では，URL の最初の部分に通信プロトコルが含まれています。VisiBroker for .NET は，新しいプロトコル CORBA IIOP を使用して .NET リモート処理を拡張します。

- corbaname:rir:#location/of/my/server/object は，いくつかの CORBA ORB string_to_object() 互換 URL スキームの 1 つです。URL スキームの他の例および説明については，表 3.1 を参照してください。

URL スキーム

ブートストラップの問題に対処し，人間が解読可能なオブジェクトリファレンスをさらに便利に交換できるようにするために，VisiBroker for .NET では，表 3.1 にリストされている形式の URL をオブジェクトリファレンスに変換できます。

表 3.1　.NET リモート処理 URL スキーム

URL スキーム	例	説明
corbaname:	janeva:corbaname:rir:#location/of/my/server/object または janeva:corbaname:rir:<NS_host>:<NS_port>#location/of/my/server/object	corbaname URL スキームは，EJB を解決する際に最もよく使用されます。 このスキームでは，URL でネーミングサービスの項目を表すことができます。ホストアドレスはネーミングサービスの場所と監視ポートであり，<NS_host_name>:<NS_port> または <NS_ip_address>:<NS_port> の形式で表すことができます。corbaname URL スキームの詳細は，OMG CORBA 仕様を参照してください。
corbaloc:	janeva:corbaloc:rir:<host>:<port>/object_key	corbaloc URL スキームでは，場所とオブジェクトキーに基づいてサーバーオブジェクトに直接アクセスできます。処理能力の制限により，このスキームはあまり使用されません。corbaloc URL スキームの詳細は，OMG CORBA 仕様を参照してください。
osagent:	janeva:osagent:poa:<poa_name>:<object_id>[:<server_host_name>] または janeva:osagent:repid:<interface_repository_id>[:<object_name>][:<server_host_name>]	osagent スキームは，Borland VisiBroker CORBA サーバーオブジェクトで使用される専用の機能です。 あいまいさを避けるため，<interface_repository_id> 内のすべてのコロン（:）の前にバックスラッシュ（\）を付加します。たとえば，janeva:osagent:repid:IDL\:com/semagroup/targys/servicelayer/corba/ServiceRootI\:1.0:SL_demo_server のようになります。
IOR:	janeva:IOR:<stringified_object_reference>	IOR URL スキームを使用すると，文字列化されたオブジェクトリファレンス（IOR）でオブジェクトをルックアップできます。

表 3.1　.NET リモート処理 URL スキーム （続き）

URL スキーム	例	説明
http:	janeva:http:<host_address>/location/of/my/ior/file	HTTP URL スキームは，文字列化されたオブジェクトリファレンスを含むテキストファイルをポイントします。
file:	janeva:file:<host_address>/location/of/my/ior/file	file URL スキームは，文字列化されたオブジェクトリファレンスを含むテキストファイルをポイントします。

リモート処理チャネルの指定

リモートオブジェクトと通信するために，.NET クライアントアプリケーションは，リモート処理「チャネル」を作成して登録する必要があります。チャネルは，クライアントとリモートオブジェクトとの間の通信用に，コンジットを提供します。

VisiBroker for .NET は，.NET Framework Channels 型を使用せずに，IIOP 上にチャネルを作成するための Janeva.Remoting.IiopChannel 型を提供します。

```
<channel type="Janeva.Remoting.IiopChannel, Borland.Janeva.Runtime"/>
```

2 番目の引数は，VisiBroker for .NET ランタイムアセンブリ名です。

クライアントアクティブ化オブジェクトとサーバーアクティブ化オブジェクト

VisiBroker for .NET は，リモート処理可能なオブジェクトに対して両方のタイプのアクティブ化をサポートしています。

- **サーバーでのアクティブ化**。サーバーアクティブ化オブジェクト（SAO）は，必要なときにのみサーバーによって作成されます。これらは，new または Activator.GetObject を呼び出してクライアントプロキシを作成しても作成されず，クライアントが最初にプロキシのメソッドを呼び出すときに作成されます。この章の前のセクションに，このオブジェクトアクティブ化メソッドの例が示されています。
- **クライアントでのアクティブ化**。アプリケーションがメソッド呼び出しとメソッド呼び出しの間で状態を保持する必要があり，各クライアントと固有のオブジェクトインスタンスを対にする必要がある場合は，クライアントアクティブ化オブジェクトを使用します。クライアントアクティブ化オブジェクト（CAO）は，クライアントが new または Activator.CreateInstance を呼び出すと際に，サーバーで作成されます。

VisiBroker for .NET でサポートされているリモートオブジェクトは，クライアント側で SAO として使用できます。また，J2EE サーバーオブジェクトを SAO として表すこともできます。

VisiBroker for .NET でのクライアントのアクティブ化は，多くの J2EE コンポーネントがファクトリ設計パターンにしたがっているという事実に基づいて行われます。つまり，リモートでアクセス可能な EJB（ステートフルセッション Bean，ステートレスセッション Bean，またはエンティティ Bean）は，Bean インスタンスを作成または解決する際に使用されるホームインターフェースを公開します。CAO として設定された EJB の場合は，ホームインターフェースの解決をスキップし，Bean のプロキシクラスのインスタンスを作成するだけで Bean インスタンスを作成または解決できます。

たとえば，次のような単純な EJB インターフェース SimpleSession とそのホームインターフェース SimpleSessionHome を考えます。

```
public interface SimpleSession extends javax.ejb.EJBObject {
   public void ping() throws java.rmi.RemoteException;
}
```

```
public interface SimpleSessionHome extends javax.ejb.EJBHome {
   public SimpleSession create(String name);
}
```

SAO として設定された SimpleSession インターフェースには，次のような C# コードでクライアント側からアクセスできます。

```
SimpleSessionHome home = new SimpleSessionHomeRemotingProxy();
SimpleSession session = home.Create("my name");
session.Method();
```

SimpleSession インターフェースが CAO として表されている場合，クライアントコードは次のようになります。

```
SimpleSession session = new SimpleSessionRemotingProxy("my name");
session.Method();
```

次に，VisiBroker for .NET が J2EE コンポーネントのクライアントアクティブ化モデルをどのようにサポートするかについて詳しく説明します。

最初に，java2cs コンパイラが EJB ホームインターフェースに関する情報を拡張します。コンパイラは，生成された C# コードで，EJB ホームインターフェースで定義されたメソッドのいくつかを Bean のリモートプロキシクラスのコンストラクタにマップします。セッション EJB ホーム（ステートフルまたはステートレス）の場合は，create() メソッドをマップします。エンティティ EJB ホームの場合は，findByPrimaryKey() メソッドをマップします。また，java2cs コンパイラは，元の home メソッドのパラメータを生成されたプロキシコンストラクタでも維持します。たとえば，SimpleSessionHome.create(String name) メソッドは，生成された C# コードで SimpleSessionRemotingProxy(string name) コンストラクタにマップされます。

CAO リモートプロキシの新しいインスタンスが作成される際に，VisiBroker for .NET ランタイムはバックグラウンドでいくつかの処理を実行します。最初に，リモート処理オブジェクト設定で指定された VisiBroker for .NET URL に基づいて，Bean のホームインターフェースを解決します。次に，ランタイムは，セッション Bean の場合は対応するセッションの create() メソッドを，エンティティ Bean の場合はエンティティの findByPrimaryKey() をリモートに呼び出します。最後に，この呼び出しの結果になる EJB インスタンスのリモートプロキシが，new 文から返されるオブジェクトになります。

VisiBroker for .NET CAO モデルは元の .NET リモート処理 CAO モデルにたいへんよく似ていますが，次のような特徴があります。

1. CAO として設定された EJB リモートプロキシを作成しても，サーバー側（EJB コンテナ）で新しい EJB インスタンスが作成されるとは限りません。セッション Bean では EJB インスタンスが作成されますが，エンティティ Bean では動作が異なります。エンティティ Bean では，CAO コンストラクタの呼び出しは findByPrimaryKey() の呼び出しに変換されます。そのため，対応する主キーを持つ既存のインスタンスが必要です。そのようなインスタンスがない場合は，例外がスローされます。CAO として表されたエンティティ Bean は，Bean インスタンスの作成ではなく，解決にのみ使用されます。新しいエンティティインスタンスを作成するには，SAO モデルを使用します。
2. VisiBroker for .NET CAO は，ライフタイムリースモデルをサポートしません。これは，EJB がこの概念をモデル化しているためです。また，EJB ライフサイクルは，EJB のタイプによって異なります。クライアント側の開発者はこれらの違いについて理解し，EJB インスタンスが必要なくなったときに，EJB インターフェースまたはホームの Remove() メソッドを明示的に呼び出す必要があります。

CAO の例の設定ファイルは，次のようになります。

```
<configuration>
   <system.runtime.remoting>
      <application name="MyApplication">
         <client url="janeva:corbaname:rir:#location/of/my/server/object">
            <activated type="SimpleSessionRemotingProxy,
               MyApplicationAssembly"/>
         </client>
         <channels>
            <channel type="Janeva.Remoting.IiopChannel,
               Borland.Janeva.Runtime"/>
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

プログラマティックアクティベーション

設定ファイルにサーバー側オブジェクトへの .NET 呼び出しを記述する方法とは別に，リモート処理チャネルをアクティブ化し，リモートオブジェクトの位置をコードに直接指定する方法があります。次のサンプルは，SAO の場合のコードを示しています。

```
static void Main(string[] args) {
   Janeva.Remoting.IiopChannel channel = new Janeva.Remoting.IiopChannel(args);
   System.Runtime.Remoting.Channels.ChannelServices.RegisterChannel(channel);
   string objectUrl = "janeva:corbaname:rir:#" +
      "location/of/my/server/object";
   MyServerHome myServerHome = (MyServerHome)
      System.Activator.GetObject(typeof(MyServerHome), objectUrl);
   MyServer myServer = myServerHome.Create();
   myServer.Method();
}
```

次のサンプルは，CAO の場合のコードを示しています。

```
static void Main(string[] args) {
   Janeva.Remoting.IiopChannel channel = new Janeva.Remoting.IiopChannel(args);
   System.Runtime.Remoting.Channels.ChannelServices.RegisterChannel(channel);
   string objectUrl = "janeva:corbaname:rir:#" +
      "location/of/my/server/object";
   MyServer myServer = (MyServer) System.Activator.CreateInstance(
      typeof(MyServerRemotingProxy), new object[] {"my name"});
   myServer.Method();
}
```

各例の最初の 2 行は，IIOP 上の VisiBroker for .NET リモート処理チャネルを設定します。

```
Janeva.Remoting.IiopChannel channel = new Janeva.Remoting.IiopChannel(args);
System.Runtime.Remoting.Channels.ChannelServices.RegisterChannel(channel);
```

3 番目の行では，サーバー上のファクトリオブジェクト（myServerHomeObject）の位置を入れる変数を宣言し，前のセクションに記載されている J2EE および CORBA の例で行ったのと同様の方法で検索を行います。ただし，.NET ではオブジェクトの型を narrow（キャスト）できません。

第 4 章

プロパティの設定

VisiBroker for .NET のプロパティを設定するには, 3 とおりの方法があります。これらの方法を優先順位が高い順に示します。

1. コマンドライン引数を使用する
2. プログラムでプロパティを設定する
3. 設定ファイルを使用する

メモ　優先順位が高い設定は，優先順位が低い設定を上書きします。たとえば，コマンドラインで設定されたプロパティは，プログラムで定義されたプロパティを上書きします。

コマンドラインでプロパティを設定する

VisiBroker for .NET アプリケーションをコマンドプロンプトから実行する場合は，VisiBroker for .NET プロパティをスペース区切りのキー／値ペアとして指定できます。キーの前には，ハイフン（-）をつけます。たとえば，次のようになります。

```
Client -ORBInitRef NameService=corbaloc:iiop:1.2@host1:3075/NameService
```

VisiBroker for .NET スタイルの API を使用している場合は, アプリケーションコードで, 対応するバージョンの Janeva.Remoting.IiopChannel() コンストラクタにコマンドライン引数を渡すことができます。たとえば，次のようになります。

```
static void Main(string[] args) {
  Janeva.Remoting.IiopChannel channel = new Janeva.Remoting.IiopChannel(args);
  …
}
```

CORBA スタイルの API を使用している場合は, 引数を静的な ORB.Init() コンストラクタに渡します。

```
static void Main(string[] args) {
  CORBA.ORB orb = CORBA.ORB.Init(args);
  ...
}
```

J2EE 開発者の場合は，J2EE.Naming.InitialContext() を使用する同等の ORB 初期化 API を使用できます。たとえば，ポート 2809 を監視するネーミングサービスを使用するローカルホストで J2EE サーバーが稼動しているとします。クライアントは -ORBInitRef スタイルの初期化を使用して，ネーミングサービスをポイントできます。

```
Client -ORBInitRef NameService=corbaname:iiop:localhost:2809/NameService
```

アプリケーションコードでは，これらの引数を静的な J2EE.Naming.InitialContext コンストラクタに渡します。

```
static void Main(string[] args) {
  J2EE.Naming.Context context =
    J2EE.Naming.InitialContext(args);
    …
}
```

プログラムでプロパティを設定する

VisiBroker for .NET プロパティは System.Collections.Hashtable オブジェクトに格納できます。また，CORBA.ORB.Init()，J2EE.Naming.InitialContext()，または Janeva.Remoting.IiopChannel() に渡すことができます。こうすると，コマンドラインを使用するより簡単に VisiBroker for .NET プロパティを設定できます。また，コマンドラインを使用できない場合にも便利です。

.NET リモート処理の開発者であれば，Hashtable 設定を適切なバージョンの Janeva.Remoting.IiopChannel コンストラクタに渡すことができます。

```
static void Main(string[] args) {
   System.Collections.Hashtable props = new System.Collections.Hashtable();
   props.Add("ORBInitRef",
      "NameService=corbaloc:iiop:1.2@host1:3075/NameService");
   props.Add("janeva.transaction", true);
   Janeva.Remoting.IiopChannel channel =
      new Janeva.Remoting.IiopChannel(args. props);

// ここに他のコードが入ります
   …
}
```

次の CORBA の例は，Hashtable オブジェクトを作成し，3 つのプロパティを設定します。

```
static void Main(string[] args) {
   System.Collections.Hashtable props = new System.Collections.Hashtable();
   props.Add("ORBInitRef",
      "NameService=corbaloc:iiop:1.2@host1:3075/NameService");
   props.Add("janeva.transactions", true);
   CORBA.ORB orb = CORBA.ORB.Init(args, props);

// ここに他のコードが入ります
   …
}
```

J2EE 開発者の場合は，Hashtable を使用してアプリケーションを初期化できます。

```
static void Main(string[] args) {
   System.Collections.Hashtable props = new System.Collections.Hashtable();
   props.Add("ORBInitRef",
      "NameService=corbaloc:iiop:1.2@host1:3075/NameService");
   props.Add("janeva.transactions", true);
   J2EE.Naming.InitialContext context = new J2EE.Naming.InitialContext(props);

// ここに他のコードが入ります
   …
}
```

設定ファイルでプロパティを設定する

設定ファイルを使用することで，VisiBroker for .NET プロパティを設定できます。

重要 VisiBroker for .NET 7.0 では, 設定ファイルの <janeva> セクションの名前が <visinet> に変更されました。ただし，旧バージョンとの後方互換性のために，セクション名 <janeva> も引き続きサポートされます。

適切な名前を指定することで, 設定ファイルが自動的にロードされます。ASP.NET アプリケーションの場合は, Web.config ファイルがロードされます。他のアプリケーションの場合は，<app_assembly_name>.exe と同じディレクトリに置かれた <app_assembly_name>.exe.config ファイルがロードされます。

メモ Microsoft Visual Studio .NET では, app.config ファイルをプロジェクトに追加して, ビルドに含まれている適切な名前の XML 設定ファイルを取得する必要があります。

次の例は，設定ファイルのサンプルを示します。

```
<configuration>
   <configSections>
      <section name="visinet" type="Janeva.Settings, Borland.Janeva.Runtime"/>
   </configSections>
   <visinet>
      <transactions enabled="true"/>
      <server defaultPort="10000">
         <remoting enabled="true"/>
      </server>
   </visinet>
</configuration>
```

設定ファイルでは, すべての VisiBroker for .NET 設定が <visinet> セクションにまとめられていることに注意してください。VisiBroker for .NET 設定は標準の .NET 設定 XML には含まれていないため, .NET ランタイムに <visinet> XML を読み取るように指示する必要があります。それには，上の例で示したように，<configSections> セクションを追加します。

VisiBroker for .NET プロパティの説明

設定ファイルには, 各 VisiBroker for .NET プロパティに対応する設定が含まれています。以下のセクションでは, 各 VisiBroker for .NET プロパティとそれに対応する設定ファイルの設定について詳しく説明します。

ネーミングサービスの解決

次のプロパティは，ネーミングサービスを解決するために使用されます。

ORBInitRef

型 : string

デフォルト値 : なし

XML :

```
<naming url="NameService=URL" />
```

次の表に示すように，各アプリケーションサーバーには独自の URL 構文があります。

アプリケーションサーバー	ネーミングサービス URL の形式
WebLogic 7 または 8	corbaloc::localhost:7001/NameService
IBM WebSphere 5	corbaname:iiop:localhost:2809/NameServiceServerRoot
Oracle OC4J	corbaloc:iiop:1.2@localhost:5555/NameService
Sybase	corbaloc:iiop:1.2@localhost:9000/NameService

デフォルトのポート番号は配布ごとに異なります。

メモ Borland AppServer のネーミングサービスは, OSAgent に基づいて自動的に解決されます。そのため, Borland AppServer ではこの設定はオプションです。他のアプリケーションサーバーでは，この設定が必要です。

例

コマンドラインを使用してネーミングサービスを解決するには，引数を次の形式で指定する必要があります。

```
> client -ORBInitRef NameService=corbaloc::localhost:7001/NameService
```

設定ファイルのプロパティ設定は，次の例のようになります。

```
<visinet>
   <naming url="corbaloc::localhost:7001/NameService" />
</visinet>
```

ライセンスのプロパティ

このプロパティを設定して, 必要であれば VisiBroker for .NET ランタイムがライセンスを検出できるようにします。

janeva.license.dir

VisiBroker for .NET ライセンスファイルがあるディレクトリのパスを設定します。絶対パスまたは現在のディレクトリからの相対パスを指定できます。

型 : string

デフォルト値 : なし

XML :

```
<license dir="path" />
```

例

次の例は，設定ファイルで janeva.license.dir プロパティを設定しています。

```
<visinet>
   <license dir="C:¥Program Files¥Borland¥Janeva" />
</visinet>
```

トランザクションのプロパティ

これらのプロパティを設定して, VisiBroker for .NET トランザクションのサポートを有効にします。

janeva.transactions

クライアント境界トランザクションをサポートするには, このプロパティを true に設定します。この機能を有効にしないと, 新しいトランザクションを開始できないことに注意してください。 つまり，トランザクションが無効な場合は, orb.ResolveInitialReferences("TransactionCurrent") の呼び出しが失敗します。

デフォルトでは，この機能は無効になっています。これを有効にすると，リモート起動中に余分なパフォーマンスオーバーヘッドが発生します。

型 : 論理値 [true | false]

デフォルト値 : false

XML :

```
<transactions enabled="value" />
```

メモ　設定ファイルに <transactions> セクションがあり，enabled 属性がない場合は，トランザクションがデフォルトで有効になります。

例

次の設定例は，janeva.transactions プロパティを true に設定しています。

```
<visinet>
   <transactions enabled="true" />
</visinet>
```

```
<visinet>
   <transactions />
</visinet>
```

janeva.transactions.factory.url

この URL は，トランザクションサービスの現在のファクトリを示します。

型 : string

デフォルト値 : なし

XML :

```
<transactions>
   <factory url="corbaloc::URL" />
</transactions>
```

例

次の設定例は，janeva.transactions.factory.url プロパティを設定しています。

```
<visinet>
   <transactions enabled="true">
      <factory url="corbaloc::localhost:6666/TransactionFactory" />
   </transactions>
</visinet>
```

サーバー側のプロパティ

これらのプロパティを使用して，VisiBroker for .NET のサーバー側サポートを設定します。

janeva.server.defaultPort

このプロパティは, VisiBroker for .NET サーバーが IIOP 要求を監視するポートを設定します。値 0 を指定すると，システムがポート番号をランダムに選択します。

型 : 整数

デフォルト値 : 0

XML :

```
<server defaultPort="value">
```

janeva.server.remoting

このプロパティは，リモート処理スタイルのコールバックとリモート処理スタイルの VisiBroker for .NET サーバーを使用する場合に設定します。true に設定すると, リモート処理スタイルのコールバックとリモート処理スタイルの VisiBroker for .NET サーバーが有効になります。

この機能は，デフォルトでは無効になっています。有効にすると，リモート起動中に余分なパフォーマンスオーバーヘッドが発生します。

型 : 論理値 [true | false]

デフォルト値：false

XML：

```
<server><remoting enabled="value" /></server>
```

例

次の例は，設定ファイルで janeva.server.port プロパティと janeva.server.remoting プロパティを設定しています。

```
<visinet>
   <server defaultPort="2809">
      <remoting enabled="true" />
   </server>
</visinet>
```

相互運用のプロパティ

このプロパティを使用して，VisiBroker for .NET の相互運用性に関するさまざまな設定を行います。

janeva.interop.jvmType

このプロパティは，VisiBroker for .NET がネットワーク経由で特定のデータ型を書き込む方法を制御します。これは，発信接続の受信側で JVM を指定します。この設定は，Java プログラムとして稼動しているサーバーと通信する場合に適しています。.NET クライアントと .NET サーバーの間で通信する場合は，両方でこのプロパティを同じ値に設定する必要があります。

型：整数 [1|2|3]

デフォルト値：1

XML：

```
<interop jvmType="value" />
```

JDK の変更に伴い，さまざまなデータ型のマーシャリング形式も変化してきました。このように変化するデータ型を記述するために，このフラグを使用して，相互運用する VM のタイプを指定します。

現在，このフラグには 3 つの有効な設定があります。

1 値 1 は，バージョン 1.1，1.2，または 1.3 の JVM を使用することを示します。

2 値 2 は，バージョン 1.4.0 または 1.4.1 の JVM を使用することを示します。

3 値 3 は，バージョン 1.4.2 以降の JVM を使用することを示します。

JVM タイプ 1 と 2 の主な違いは，次のプロパティの記述形式です。

```
java.lang.Random
java.math.BigDecimal
java.math.BigInteger
```

JDK バージョン 1.4.0 ではこの形式が変更されました。このデータを VisiBroker for .NET プロセス間で送信する必要がある場合は，このフラグを適切に設定する必要があります。

JVM タイプ 2 と 3 の主な違いは，次のプロパティの記述形式です。

```
java.util.Vector
java.util.Stack
```

JDK バージョン 1.4.2 ではこの形式が変更されました。このデータを VisiBroker for .NET プロセス間で送信する必要がある場合は，このフラグを適切に設定する必要があります。

JVM の相互運用では，次の点に注意が必要です。

- janeva.interop.jvmType プロパティは，VisiBroker for .NET の「書き込み側」にのみ影響します。
- VisiBroker for .NET の「*読み取り側*」はすべての JVM をサポートします。このため，jvmType フラグの設定に関係なく，任意の JVM で稼動している J2EE アプリケーションから Random，Vector，Stack の各インスタンスを受け取ることができます。このようなオブジェクトを VisiBroker for .NET プロセスから J2EE アプリケーションに送信する必要がある場合にのみ，jvmType を指定します。

例

次の例は，設定ファイルで janeva.interop.jvmType プロパティを設定しています。

```
<visinet>
   <interop jvmType="2"/>
</visinet>
```

セキュリティのプロパティ

これらのプロパティを使用して，VisiBroker for .NET のセキュリティサポートを設定します。

janeva.security

VisiBroker for .NET のセキュリティサポートを有効にするには，このプロパティを true に設定します。

この機能は，デフォルトでは無効になっています。有効にすると，リモート起動中に余分なパフォーマンスオーバーヘッドが発生します。

型：論理値 [true | false]

デフォルト値：false

XML：

```
<security enabled="value"/>
```

メモ　設定ファイルに <security> セクションがあり，enabled 属性がない場合は，セキュリティがデフォルトで有効になります。

janeva.security.username

このプロパティは，認証のためにサーバー側に渡されるセキュリティ ID のユーザー名を設定します。このプロパティは，janeva.security.password プロパティとともに使用されます。

型：string

デフォルト値：なし

XML：

```
<security><identity><username>value</username></identity></security>
```

janeva.security.password

クリアテキスト形式でパスワードを指定します。

型：string

デフォルト値：なし

XML :

```
<security>
   <identity>
      <password>value</password>
   </identity>
</security>
```

janeva.security.realm

これは, セキュリティ ID 設定でユーザー名やパスワードと組み合わせて使用される認証領域です。デフォルトでは, ユーザーは default という名前のセキュリティ領域に属しています。これ以外の認証領域を使用する場合は, このプロパティを設定します。

型 : string

デフォルト値 : default

XML :

```
<security>
   <identity>
      <realm>value</realm>
   </identity>
</security>
```

janeva.security.certificate

このプロパティは, 認証に使用される証明書を設定します。Windows 証明書ストアにある証明書のフレンドリ名を表す文字列を指定します。

型 : string

デフォルト値 : なし

XML :

```
<security><identity><certificate>value</certificate></identity></security>
```

例

次の例は, 設定ファイルでセキュリティ ID の janeva.security.username, janeva.security.password, janeva.security.realm の各プロパティを指定しています。

```
<visinet>
   <security enabled="true">
      <identity>
         <username>admin</username>
         <password>foobar</password>
         <realm>MyRealm</realm>
      </identity>
   </security>
</visinet>
```

次の例では, 設定ファイルでセキュリティ ID の janeva.security.certificate プロパティを設定しています。

```
<visinet>
   <security enabled="true">
      <identity>
         <certificate>joeshopper</certificate>
      </identity>
   </security>
</visinet>
```

サーバー側セキュリティのプロパティ

これらのプロパティを使用して，VisiBroker for .NET のサーバー側サポートを設定します。

janeva.security.server

VisiBroker for .NET のサーバー側セキュリティサポートを有効にするには，このプロパティを true に設定します。

デフォルトでは，この機能は無効になっています。これを有効にすると，リモート起動中に余分なパフォーマンスオーバーヘッドが発生します。

型：論理値 [true | false]

デフォルト値：false

XML：

```
<security>
   <server enabled="value"/>
</security>
```

メモ　設定ファイルに <security><server> セクションがあり，enabled 属性がない場合は，サーバー側セキュリティがデフォルトで有効になります。

Åüjaneva.security.server.defaultPort

SSL over IIOP で使用されるポートを設定します。

型：整数

デフォルト値：なし

XML：

```
<security>
   <server defaultPort="value"/>
</security>
```

janeva.security.server.certificate

このプロパティは，証明書のフレンドリ名を指定します。このセクションで指定された証明書は，SSL 接続のサーバーピアを特定する際に使用されます。この設定で値が指定されなかった場合，VisiBroker for .NET ランタイムは，janeva.security.certificate 設定で指定された証明書を使用しようとします。これら両方の設定に値が指定されていない場合，VisiBroker for .NET ランタイムは，設定が正しくないと判断し，初期化を実行しません。

型：string

デフォルト値：なし

XML：

```
<security>
   <server>
      <certificate>value</certificate>
   </server>
</security>
```

例

次の例は，設定ファイルで server-side セキュリティプロパティを設定しています。

```
<visinet>
   <security>
      <server enabled="true" defaultPort="15000">
         <certificate>Book Store</certificate>
      </server>
   </security>
</visinet>
```

ファイアウォールのプロパティ

このプロパティを使用して, VisiBroker for .NET ファイアウォールサポートを有効にします。

janeva.firewall

Borland Gatekeeper などの高レベルなファイアウォールゲートウェイをサポートします。

デフォルトでは，この機能は無効になっています。これを有効にすると，リモート起動中に余分なパフォーマンスオーバーヘッドが発生します。

型：論理値 [true | false]

デフォルト値：false

XML：

```
<firewall enabled="value"/>
```

メモ　設定ファイルに <firewall> セクションがあり，enabled 属性がない場合は，ファイアウォールがデフォルトで有効になります。

例

次の例は，設定ファイルで janeva.firewall プロパティを設定しています。

```
<visinet>
   <firewall enabled="true"/>
</visinet>
```

ポータブルインターセプタのプロパティ

このプロパティを使用して，ポータブルインターセプタを設定します。

janeva.orb.init

ORB によってロードされるポータブルインターセプタを指定します。メインクラスを格納するアセンブリにポータブルインターセプタが含まれている場合は，クラス名を指定するだけで十分です。メインクラスを格納するアセンブリ以外のアセンブリにポータブルインターセプタが含まれている場合は, 厳密なアセンブリ名を指定する必要があります。必要な数だけのポータブルインターセプタを指定できます。

型：string

デフォルト値：なし

XML：

```
<orb>
   <init type="value"/>
</orb>
```

例

次の例は，設定ファイルで janeva.orb.init プロパティを設定しています。

```
<visinet>
   <orb>
      <init type="MyInterceptor, MyInterceptorAssembly, version=1.2.3.4,
         culture=neutral, publicKeyToken=xxxx"/>
      <init type="MyInterceptor2″/>
   </orb>
</visinet>
```

VisiBroker Smart Agent のプロパティ

これらのプロパティは，オブジェクトの登録とルックアップに Smart Agent（osagent）を使用する場合に設定します。

janeva.agent

このプロパティを使用して，Smart Agent を無効にできます。

型：論理値 [true | false]

デフォルト値：false

XML：

```
<agent enabled=″value″/>
```

janeva.agent.port

このプロパティは，Smart Agent によって使用されるポートを設定します。

型：整数

デフォルト値：14000

XML：

```
<agent port=″value″/>
```

janeva.agent.addr

このプロパティは，Smart Agent の物理的な場所を IP アドレスまたはホスト名で指定します。指定しないと，VisiBroker for .NET は，ping を実行して，正しいポートを持つ Smart Agent をネットワークで探します。ホストアドレスを指定すると，ネットワークトラフィックが減ります。これは，VisiBroker for .NET が指定されたホストアドレスとポートで Smart Agent を ping するためです。

型：string

デフォルト値：なし

XML：

```
<agent addr=″value″/>
```

例

次の例は，設定ファイルで janeva.agent janeva.agent.port プロパティと janeva.agent.addr プロパティを設定しています。

```
<visinet>
   <agent enabled=″true″ port="14001" addr="localhost.localdomain.com"/>
</visinet>
```

VisiBroker のプロパティの設定

VisiBroker for .NET は, 一連の Borland VisiBroker 製品に最初に導入されたプロパティをすべてサポートしています。これらのプロパティには, ファイアウォールサポートの微調整に使用される設定が含まれています。設定ファイルの <vbroker> セクションで, VisiBroker のプロパティをキー／値の形式の属性として設定できます。

次の例は, 設定ファイルで VisiBroker GateKeeper プロパティのいくつかを設定しています。

```
<visinet>
   <firewall enabled="true"/>
   <vbroker
      vbroker.orb.alwaysProxy="true"
      vbroker.orb.gatekeeper.ior="ior:…"
   />
</visinet>
```

第 5 章

VisiBroker for .NET アプリケーションの構築と配布

この章では，VisiBroker for .NET を使用した .NET アプリケーションを構築および配布するためのプロセスを説明します。この章には，次のトピックが記載されています。

VisiBroker for .NET スタブとスケルトンの生成

J2EE および CORBA 技術によってオブジェクトレベルのインターフェースが定義されるため，.NET アプリケーションとサーバーオブジェクト間の通信は，これらのインターフェースを通じて排他的に行われます。CORBA では，インターフェースは IDL で定義され，J2EE では Java RMI で定義されます。

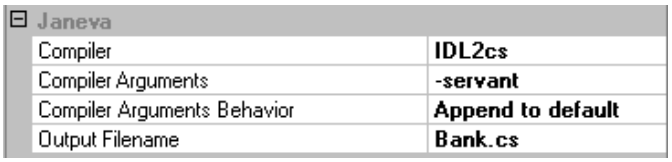
VisiBroker for .NET `java2cs` および `idl2cs` ツールは，Java RMI または IDL から C# へインターフェースを変換します。VisiBroker for .NET は，IDE プロジェクトでこれらのツールを設定して使用できるように, Microsoft Visual Studio .NET に機能を追加します。コマンドラインを使用して，インターフェースをコンパイルすることもできます。

Visual Studio

Visual Studio .NET で VisiBroker for .NET のスタブとスケルトンを生成する手順は，次のとおりです。

1. Visual Studio プロジェクトに IDL，JAR，または EAR を追加します。
2. 目的のファイルを選択して，図 5.1 に示されている VisiBroker for .NET プロパティを確認します。

図 5.1 Microsoft Visual Studio .NET VisiBroker for .NET のプロパティ



IDL ファイルの場合は，コンパイラを IDL2cs に設定してください。JAR または EAR ファイルは Java2cs コンパイラを使用する必要があります。コンパイラ引数を追加し，プロパティダイアログで出力ファイルの名前を変更できます。

重要： サーバースケルトンコードを生成する場合は，コンパイラ引数に `-servant` コンパイラフラグを追加する必要があります。

3 インターフェースファイルだけをコンパイルするには，ソリューションエクスプローラで目的のファイルを右クリックして，［Build and Browse］を選択します。

コンパイルが正常に行われると，C# ファイルが生成され，プロジェクトに追加されます。

コマンドライン コンパイラをコマンドラインで使用するには，コマンドプロンプトから実行できるように，ツールがパスに入っていることを確認してください。コンパイラは，VisiBroker for .NET インストールディレクトリの bin ディレクトリに入っています。コンパイラがパスに入っているかどうかを調べるには，コマンドプロンプトを開いて，`idl2cs` と入力します。コンパイラスイッチのリストが表示されるはずです。

インストールプロセス中にコンパイラをパスに追加しなかった場合は，以下のように入力して，コマンドラインから `idl2cs` をパスに追加できます。

```
prompt> set PATH=<VisiBroker Home>¥VisBroker.NET¥bin;%PATH%
```

コンパイラがパスに入っていることが確認できれば，使用できます。

```
prompt> idl2cs Example.idl
```

コンパイルが正常に行われると，C# ファイルが生成されます。

VisiBroker for .NET ランタイムライブラリへの参照の追加

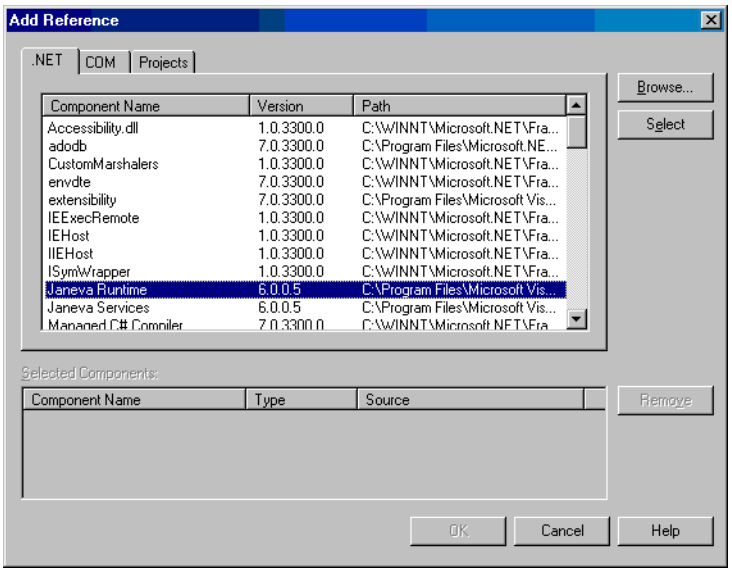
VisiBroker for .NET ランタイムを利用するには，アプリケーションから VisiBroker for .NET の DLL を参照する必要があります。以下のセクションで，VisiBroker for .NET ランタイムライブラリのリファレンスをアプリケーションに追加する方法について説明します。

Visual Studio Visual Studio .NET で VisiBroker for .NET ランタイムライブラリへの参照を追加する手順は，次のとおりです。

1 ソリューションエクスプローラでアプリケーションの［References node］を右クリックします。

2 ［Add Reference］を選択します。

図 5.2 Microsoft Visual Studio .NET Add Reference ダイアログ



3 .NET タブで適切な VisiBroker for .NET リファレンスを選択し，[Select] をクリックします。

クライアントアプリケーションを構築する場合は，VisiBroker for .NET ランタイムリファレンスだけを選択します。　サーバーアプリケーションを構築する場合は，VisiBroker for .NET ランタイムリファレンスと VisiBroker for .NET サービスリファレンスの両方を選択します。

4 [OK] をクリックします。

VisiBroker for .NET ランタイムリファレンスを選択した場合は，Borland.Janeva.Runtime が参照リストに表示されます。VisiBroker for .NET サービスリファレンスを選択した場合は，Borland.Janeva.Services が参照リストに表示されます。

コマンドライン　コンパイル時に VisiBroker for .NET ランタイムライブラリへの参照を追加するには，C# コマンドラインコンパイラを呼び出して　C#　ソースコードを指定する際に，Borland.Janeva.Runtime.dll または Borland.Janeva.Services.dll をリファレンスとしてインクルードします。

```
prompt> csc /r:Borland.Janeva.Runtime.dll Client.cs
```

VisiBroker for .NET アプリケーションの配布

VisiBroker for .NET 技術を使用したアプリケーションを配布するには，次の項目が必要です。

- Microsoft .NET Framework 再配布パッケージ
- VisiBroker for .NET ランタイムライブラリ
- VisiBroker for .NET 配布ライセンスキー

Microsoft .NET Framework 再配布パッケージ

VisiBroker for .NET は .NET 製品です。したがって，実行には .NET Framework 再配布パッケージが必要です。これは，Microsoft の Web サイトから無料でダウンロードできます。IDE またはオペレーティングシステムに組み込まれている場合もあります。

VisiBroker for .NET ランタイムライブラリ

配布については，VisiBroker for .NET はフロントエンドのクライアントアプリケーションまたは ASP.NET サーバーアプリケーションをサポートしています。VisiBroker for .NET で作成されたアプリケーションを実行するマシンごとに，以下の VisiBroker for .NET ランタイムライブラリをインストールする必要があります。

- Borland.Janeva.Runtime.dll
- Borland.Janeva.Runtime.Private.dll
- Borland.Janeva.NTD.dll
- Borland.BC.Bootstrap.dll
- Borland.BC.Rt.Core.dll
- Borland.BC.Jre.dll

セキュリティ，ファイアウォール，トランザクションなどのサービスを使用する場合に限り，次の 2 つをインストールする必要があります。

- Borland.Janeva.Services.dll
- Borland.Janeva.Services.Private.dll

使用するアプリケーションサーバーに応じて，次の 1 つ以上をインストールする必要があります。

- Borland.Janeva.[BES | Oracle | WebLogic | WebSphere].dll

このインストールには，次の 2 つの方法があります。

- VisiBroker for .NET CD を使用してターゲットマシン上に VisiBroker for .NET ランタイムライブラリをインストールする方法
- インストールした VisiBroker for .NET 開発環境のランタイムライブラリを，アプリケーション設定プログラムにパッケージする方法

VisiBroker for .NET ランタイムライブラリを GAC にインストールした場合，VisiBroker for .NET ランタイムを使用する同じホスト上のクライアントでランタイムライブラリを共有できます。

VisiBroker for .NET 配布ライセンスキー

VisiBroker for .NET 配布ライセンスキーは, VisiBroker 配布 CD-ROM の License ディレクトリにあります。ライセンスの使用方法は，以下の 3 とおりです。

- ライセンスを埋め込みリソースとしてインクルードする
- ライセンスをアプリケーションの仮想ルートにコピーする（ASP.NET 配布の場合）
- アプリケーション設定ファイルでライセンスファイルの場所を指定する

重要 ライセンス契約を参照して，配布ライセンスを使用できるマシンの数およびタイプに関する制限を確認してください。

ライセンスを埋め込みリソースとしてインクルードする

以下の手順は，Microsoft Visual Studio .NET を使用して，VisiBroker for .NET 配布ライセンスをアプリケーションに埋め込みリソースとしてインクルードするためのステップを説明しています。

Visual Studio .NET を使用してリソースを埋め込む手順

1 ライセンスファイル（client.slip または server.slip）を VisiBroker 配布 CD-ROM の License ディレクトリからプロジェクトディレクトリにコピーします。

2 SLIP ファイルの名前を borland.slip に変更します。

3 ソリューションエクスプローラで［Show All Files］をクリックします。

4 目的のライセンスファイルを右クリックして，［Include In Project］を選択します。

5 目的のライセンスファイルを右クリックして，［Properties］を選択します。

6 ［Build Action］プロパティを「Embedded Resource」に変更します。

アプリケーション仮想ルートへのライセンスのコピー

VisiBroker for .NET 配布ライセンスを ASP.NET サーバーアプリケーションのアプリケーションルートにコピーする手順は，次のとおりです。

1 ライセンスファイル（client.slip または server.slip）を VisiBroker 配布 CD-ROM の License ディレクトリからアプリケーションのルートインストールディレクトリにコピーします。

2 SLIP ファイルの名前を borland.slip に変更します。

アプリケーション設定ファイルの変更

VisiBroker for .NET 配布ライセンスの位置をアプリケーション設定ファイルに含める手順は，次のとおりです。

1 ライセンスファイル（client.slip または server.slip）を VisiBroker 配布 CD-ROM の License ディレクトリからネットワーク上のディレクトリにコピーします。

2 XML を次の例に示されているように変更して，<license> 要素を追加します。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <configSections>
      <section name="visinet" type="Janeva.Settings, Borland.Janeva.Runtime"/>
   </configSections
   <visinet>
      <license dir="C:¥Program Files¥Borland¥Janeva"/>
   </visinet>
</configuration>
```

<license> dir の値は，Borland ライセンスキーが含まれているファイルの絶対パスまたは相対パスにする必要があります。

第 6 章

VisiBroker for .NET リモート処理サーバーの開発

この章では，VisiBroker for .NET リモート処理サーバーの開発方法について説明します。特に VisiBroker for .NET に MarshalByRefObject オブジェクトを実装する方法について詳しく説明します。

はじめに

この章では，.NET リモート処理サーバーと VisiBroker for .NET サーバーの概念を紹介します。

.NET リモート処理について

MarshalByRefObject オブジェクトは，サーバーで実行され，クライアントからのメソッド呼び出しを受け入れるリモートオブジェクトです。 .NET リモート処理の MarshalByRefObjects は，次の 2 つのグループに分類できます。

- サーバーアクティブ化オブジェクト（SAO）
- クライアントアクティブ化オブジェクト（CAO）

SAO は Singleton または SingleCall としてマークされます。Singleton の場合は，1 つのインスタンスがすべてのクライアントからの要求をマルチスレッドで処理します。SingleCall モードで SAO を使用すると，各要求に対して新しいオブジェクトが作成され，後で破棄されます。VisiBroker for .NET では，Singleton と SingleCall の両方の SAO モードがサポートされています。ほかに，VisiBroker for .NET では，サーバーで実行されるか，サーバーコールバックのためにクライアントで実行される一時的オブジェクト MarshalByRefObject がサポートされています。

VisiBroker for .NET サーバーについて

VisiBroker for .NET サーバーの開発は，常に IDL インターフェースの定義から開始されます。IDL インターフェースでビジネスロジックを定義し，クライアントとサーバーの両

方がこれにしたがいます。たとえば，次の例の IDL ファイルは 3 つのインターフェースを定義しています。

- AccountManager インターフェースは，ファクトリ設計パターンにしたがい，新しい銀行口座を開くための open メソッドを持ちます。
- Account インターフェースは，残高の照会と入金／出金の操作を行います。
- Callback インターフェースは，銀行取引のイベントを通知します。

```
// Bank.idl
module Bank {
   interface Callback {
      void notify(in string message);
   };

   interface Account {
      float balance();
      void credit(in float amount);
      void debit(in float amount);
   };

   interface AccountManager {
      Account open(in float balance, in Callback callback);
   };
};
```

サーバーは AccountManager インターフェースと Account インターフェースの両方を実装します。クライアントは Callback インターフェースの実装を提供します。これにより，Bank サーバーは，クライアントに銀行取引のすべてのイベントを通知するためのコールバックを実行できます。

以下の 2 つのセクションでは，.NET リモート処理スタイルで Bank サーバーを記述する方法と，.NET リモート処理スタイルのクライアントにコールバックの実装を追加する方法について簡単に説明します。

.NET リモート処理スタイルのサーバーの開発

サーバーにはビジネスロジックを実装する必要があります。Bank サンプルでは，Bank サーバーが AccountManager インターフェースと Account インターフェースの両方の実装を提供する必要があります。以下のコードは，サーバー側での AccountManager インターフェースと Account インターフェースの実装を示しています。

```
namespace Server {

   public class AccountImpl : MarshalByRefObject, Bank.Account {

      private float _balance;
      private Callback _callback;
      internal AccountImpl(float balance, Callback callback) {
         _balance = balance;
         _callback = callback;
         _callback.Notify("Created account with $" + _balance);
      }

      public float Balance() {
         _callback.Notify("Current balance is $" + _balance);
         return _balance;
      }

      public void Credit(float amount) {
         _callback.Notify("Crediting account with $" + amount);
         _balance += amount;
      }
```

```
        public void Debit(float amount) {
            if(amount <= _balance) {
                _callback.Notify("Debiting account by $" + amount);
                _balance -= amount;
            }
            else {
                _callback.Notify("Insufficient funds to debit $" + amount);
            }
        }
    }

    public class AccountManagerImpl : MarshalByRefObject, Bank.AccountManager {

        public AccountManagerImpl() {
            Console.WriteLine("AccountManager created on : " +
                System.DateTime.UtcNow.ToLongTimeString());
        }

        public Account Open(float balance, Callback callback) {
            Console.WriteLine("Opening a new account with balance = $" + balance);
            return new AccountImpl(balance, callback);
        }
    }
}
```

AccountManagerImpl クラスの Open() メソッドは，初期残高（balance）およびクライアントから渡される Callback オブジェクトリファレンスを受け取り，AccountImpl クラスの新しいインスタンスを作成します。

AccountImpl クラスの Balance() メソッドは，単にクライアントに残高（balance）を返します。Credit() メソッドは，渡された金額（amount）を口座残高（account balance）に入金し，Debit() メソッドは，要求された金額を口座残高から出金します。この 3 つの口座処理イベントは，Callback オブジェクトを介してクライアントに通知されます。

インターフェースの実装が完了したら，次にサーバーでは，AccountManagerImpl オブジェクトを既知の SingleCall サービスオブジェクトまたは既知の Singleton オブジェクトとして，.NET リモート処理システムに登録します。AccountImpl は，作成元のプロセスより存続期間が短い一時的なオブジェクトです。

Singleton オブジェクトの設定

サーバー実装オブジェクトを既知のサービスタイプ Singleton として設定すると，サーバー実装オブジェクトのインスタンスが 1 つだけ作成されます。これが，すべてのクライアントからの要求を処理する Singleton インスタンスです。 この設定は，.NET RemotingConfiguration API を使用して明示的に行うことも，.NET リモート処理設定ファイルを使用して暗黙的に行うこともできます。

明示的な登録

Singleton サーバー実装オブジェクトは，次の文を使用して，サーバー側のリモート処理システムに明示的に登録されます。

```
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(<TheServerImplClass>),
        "<objectURI>", WellKnownObjectMode.Singleton);
```

Bank サンプルでは，次のコードにより，AccountManagerImpl クラスのインスタンスが，AccountManager.iiop をエンドポイントの URL とする既知のサービスタイプ Singleton として明示的に登録されます。

```
RemotingConfiguration.RegisterWellKnownServiceType(typeof(
    Server.AccountManagerImpl),"AccountManager.iiop",
    WellKnownObjectMode.Singleton);
```

暗黙的な登録

サーバー実装オブジェクトを既知のサービスタイプ Singleton として暗黙的に登録するには，次に例に示すように，.NET リモート処理設定ファイルで <service> プロパティを指定します。

```
<configuration>
   <system.runtime.remoting>
      <application>
         <service>
            <wellknown mode="Singleton"
                  type="<namespace>.<implclassname>, <assembly>"
                  objectUri="<objectURI>"/>
         </service>
      </application>
   </system.runtime.remoting>
</configuration>
```

また，.NET RemotingConfiguration を呼び出して設定ファイルをロードします。

```
RemotingConfiguration.Configure("<configfile>");
```

Bank サンプルでは，サーバーの設定ファイルは最終的に次のようになります。

```
<configuration>
   <configSections>
      <section name="visinet" type="Janeva.Settings, Borland.Janeva.Runtime"/>
   </configSections>
   <visinet>
      <agent port="24300" addr="localhost"/>
      <server defaultPort="10000">
         <remoting enabled="true"/>
      </server>
   </visinet>
   <system.runtime.remoting>
      <application name="Server">
         <channels>
            <channel type="Janeva.Remoting.IiopChannel,
               Borland.Janeva.Runtime"/>
         </channels>
         <service>
            <wellknown mode="Singleton"
                  type="Server.AccountManagerImpl, Server"
                  objectUri="AccountManager.iiop"/>
         </service>
      </application>
   </system.runtime.remoting>
   <janeva.runtime.remoting>
      <wellknown objectUri="AccountManager.iiop" jndi="a/b/c"/>
   </janeva.runtime.remoting>
</configuration>
```

Janeva.Remoting.IiopChannel 型とそのプロパティの詳細は，18 ページの「リモート処理チャネルの指定」を参照してください。

SingleCall オブジェクトの設定

サーバーオブジェクトを既知の SingleCall オブジェクトとして設定した場合，サーバーは，クライアントがメソッドを呼び出すたびに 1 つのインスタンスを作成し，メソッドを実行して，オブジェクトを破棄します。Singleton モードと同様に，この設定は，.NET RemotingConfiguration API を使用して明示的に行うことも，.NET リモート処理設定ファイルを使用して暗黙的に行うこともできます。

明示的な登録

SingleCall サーバー実装オブジェクトを明示的に登録するには，次のコードを使用します。

```
RemotingConfiguration.RegisterWellKnownServiceType(typeof(<TheServerImplClass>),
   “<objectURI>”, WellKnownObjectMode.SingleCall);
```

暗黙的な登録

SingleCall サーバー実装オブジェクトを暗黙的に登録するには，.NET リモート処理の設定ファイルで，<wellknown> プロパティのモード属性を SingleCall に変更します。

```
<configuration>
    <configSections>
      <section name="visinet" type="Janeva.Settings, Borland.Janeva.Runtime"/>
   </configSections>
   <visinet>
      <agent port="24300" addr="localhost"/>
      <server defaultPort=”10000”>
         <remoting enabled="true"/>
      </server>
   </visinet>
   <system.runtime.remoting>
      <application>
         <service>
            <wellknown mode="SingleCall"
                  type="<namespace>.<implclassname>, <assembly>"
                  objectUri="<objectURI>"/>
         </service>
      </application>
   </system.runtime.remoting>
</configuration>
```

Bank サンプルサーバーで Singleton モードと SingleCall モードの出力を比較すると，Singleton モードでは，クライアントが open メソッドを呼び出そうとする回数に関係なく，AccountManagerImpl クラスコンストラクタが一度だけ呼び出されることがわかります。SingleCall モードでは，クライアントが open メソッドを呼び出すたびに，コンストラクタが呼び出されます。

VisiBroker for .NET リモート処理クライアントへのコールバックの追加

コールバックオブジェクトを VisiBroker for .NET リモート処理クライアントに追加するのは簡単です。それには，IDL ファイルで定義されたコールバックインターフェースを実装し，コールバックオブジェクトのインスタンスを作成して，それをオブジェクトリファレンスとしてサーバー起動メソッドに渡します。VisiBroker for .NET では，コールバックオブジェクトは一時的なオブジェクトです。

次のコードは，Bank サンプルの最終的なクライアント実装を示しています。

```
using System;
using System.Runtime.Remoting;
using Bank;

namespace Client {

   public class CallbackImpl : MarshalByRefObject, Callback {

      public void Notify(string message) {
         Console.WriteLine("  Callback: " + message);
      }
   }

   public class Client {
```

```
        static void Main(string[] args) {
           try {
              RemotingConfiguration.Configure("Client.config");
              AccountManager manager = new AccountManagerRemotingProxy();
              Callback callback = new CallbackImpl();
              Account account = manager.Open(1000, callback);
              Console.WriteLine("Balance = $" + account.Balance());
              Console.WriteLine("Withdrawing $500");
              account.Debit(500);
              Console.WriteLine("balance = $" + account.Balance());
              Console.WriteLine("Depositing $100");
              account.Credit(100);
              Console.WriteLine("Balance = $" + account.Balance());
              Console.WriteLine("Withdrawing $700");
              account.Debit(700);
              Console.WriteLine("Balance = $" + account.Balance());
           }
           catch(Exception e) {
              Console.WriteLine(e);
           }
           Console.WriteLine("Press enter key to stop the client...");
           Console.ReadLine();
        }
     }
  }
```

次に，Bank クライアントによって使用される .NET リモート処理設定ファイル Client.config を示します。

```
<configuration>
   <system.runtime.remoting>
      <application name="Client">
         <channels>
            <channel type="Janeva.Remoting.IiopChannel,
               Borland.Janeva.Runtime"/>
         </channels>
         <client>
            <wellknown type="Bank.AccountManagerRemotingProxy, Client"
                  url="janeva:corbaloc::localhost:10000/AccountManager.iiop"/>
         </client>
      </application>
   </system.runtime.remoting>
</configuration>
```

VisiBroker for .NET リモート処理設定ファイルのリモート処理セクションの記述方法については，16 ページの「.NET リモート処理の設定」を参照してください。設定ファイルで VisiBroker for .NET プロパティを設定する方法については，第 4 章「プロパティの設定」を参照してください。

プロパティ

デフォルトでは，VisiBroker for .NET リモート処理サーバーおよびコールバック機能はオフになっています。VisiBroker for .NET リモート処理サーバーを開発したり，リモート処理クライアントにコールバックオブジェクトを追加するには，この機能を明示的に有効にする必要があります。それには，janeva.server.remoting プロパティを true に設定します。設定ファイルで VisiBroker for .NET プロパティを設定する方法については，第 4 章「プロパティの設定」を参照してください。

第 7 章

ヒントとカスタムマーシャリングの使用

この章では，ヒントを使用して，VisiBroker for .NET の値型に対する java2cs のコード生成を制御する方法について説明します。

VisiBroker for .NET は，Java の値型に対するコード生成をカスタマイズするための強力なメカニズムを備えています。値型は，Java で実装されている値クラスです（通常は，java.io.Serializable を直接的または間接的に拡張する）。これらのクラスは状態を持ち，ネットワークを介して状態としてマーシャリングされることを意図されています。

VisiBroker for .NET コード生成の例

ヒントの使用方法と，ヒントが java2cs のコード生成に与える影響を完全に理解できるように，次に「User」という名前の単純な Java 型の例を示します。

```
public class User implements java.io.Serializable {
   public String name;
   private String password;
   public User (String name, String password) {
      this.name = name;
      this.password = password;
   }
}
```

このクラスは，初期化できず，オブジェクトのプライベートな状態を使用する方法もないため，現実的でないことは明らかです。ただし，簡単にするため，このオブジェクトの具体的な実装（適切なコンストラクタとメソッド）は省略します。ここでの説明では，Java クラス内のメソッドは関係ありません。

メモ ここでは，この Java クラスに対応する C# クラスを生成します。メソッドの*移植*には，基本的に Java クラスのリバースエンジニアリングが必要であり，したがってメソッドの移植はサポートされていないため，Java クラス内のメソッドは無関係になります。Java バージョンの値型に対応する C# クラスに同じまたは似たメソッドを用意する必要がある場合は，対応する C# のメソッドを独自に実装する必要があります。その方法については，後半のセクションで説明します。

このサンプル Java クラス User から生成される C# コードの重要な部分を次に示します。

```
[System.Serializable] public class User {

   private string _Name;

   public virtual string Name {
      get { return this._Name; }
      set { this._Name = value; }
   }

   private string _Password;

   public virtual string Password {
      get { return this._Password; }
      set { this._Password = value; }
   }

   // 他の一般的なオブジェクトメソッドは省略
}
```

C# 型 User は，Java クラス User を表します。このコードは，いくつかの点で明らかに正しくありません。

- プライベートフィールド password（C# では _Password）にパブリックなアクセッサを提供しています。これは，Java 型が同じアクセッサを提供するかどうかに関係なく発生します。前述のとおり，コンパイラは Java メソッドを参照しません。
- name フィールド（C# の _Name）のアクセス修飾子をパブリックからプライベートに降格していますが，アクセス用にはパブリックプロパティを提供しています。
- C# オブジェクトには，Java 型から生成されたコンストラクタまたはメソッドがありません。

つまり，このクラスはほとんど使用できません。ただし，実際の値型を構築するための出発点として使用できます。生成されたコードからこのコードを切り取り，それをソースに追加し，有効なコンストラクタやメソッドを追加できます。このクラスが再度生成されないようにし，かわりに独自のバージョンを使用する方法については，後で説明します。

ValueFactory クラス

次に，User に対して生成された ValueFactory クラスを見てみます。このクラスは，ネットワークから Java クラス User を読み取る際に，C# 型 User のインスタンスを作成および初期化します。また，C# クラス User のインスタンスをリモートサーバーに渡す際に，ネットワークに正しいデータを書き込みます。ValueFactory が対応する Java 型に関連付けられていることは重要です。つまり，各 Java 型に対して個別のファクトリがあります。これにより，複数の Java 型を特定の C# 型にマップできます。

ValueFactory のメソッド

ValueFactory クラスには多くのメソッドがありますが，次の例では，理解しておく必要がある重要なメソッドを中心に説明します。

```
public class UserValueFactory : CORBA.ValueFactory {

   public virtual CORBA.TypeCode GetTypeCode() {
      return UserHelper.GetTypeCode();
   }

   public virtual System.Type GetValueType() {
      return typeof(User);
   }

   public virtual User CreateObject() {
      return new User();
   }

   public virtual void InitObject(UserValueData src_data, User dst_object) {
      dst_object.Name = src_data.Name;
```

```
        dst_object.Password = src_data.Password;
    }

    public virtual void InitData(User src_object, UserValueData dst_data) {
        dst_data.Name = src_object.Name;
        dst_data.Password = src_object.Password;
    }
}
```

次の例に示すように，UserValueData は，User クラスのすべてのインスタンスメンバーをパブリックデータメンバーとして含むクラスです。

```
public class UserValueData {
    public string Name;
    public string Password;
}
```

次の表で，ValueFactory のメソッドについて説明します。

表 7.1 ValueFactory のメソッド

メソッド名	説明
GetValueType	Java 型 MyValue にマップされるクラスの型を返します。
CreateObject	Java 型 MyValue に対応する C# 型の新しいインスタンスを返します。
InitObject	Java MyValue を読み取る際に使用されます。CreateObject によって作成された C# 型は，ValueData クラスとともに，このメソッドに渡されます。InitObject が呼び出されるとき，MyValue のデータは ValueData クラスにすでにアンマーシャリングされています。InitObject は，ValueData クラスのフィールドを C# MyValue クラスに割り当てるだけです。このパターンの実用性については，後で説明します。
InitData	C# MyValue をストリームに書き込む際に使用されます。このメソッドは，C# MyValue のメンバーの状態を ValueData クラスに転送するだけです。次に，インフラストラクチャは，ValueData クラスの状態をマーシャリングします。

上の表によると，ValueData クラスは，C# 型でデータが格納または維持される方法に関係なく，ネットワークでマーシャリングされたデータを表すことがわかります。

ValueFactory は，1 つの手順（CreateObject）でオブジェクトを作成し，別の手順（InitObject）でデータを読み取ることに注意してください。これには正当な理由があります。他のステートフルな型から継承された型をアンマーシャリングまたはマーシャリングする場合，各型のファクトリは，その階層レベルの状態だけをマーシャリングおよびアンマーシャリングする役割を持つのが普通です。そのため，インフラストラクチャは，最初にアンマーシャリングされる型のインスタンスを初期化しますが，ベースクラスから順に階層内の各型に対応するファクトリにインスタンスを渡し，関連する状態をアンマーシャリングしていきます。書き込み時には，今度は InitData メソッドを使用して，これと同じプロセスが繰り返されます。

ヒントの概要

ヒントファイルは，ユーザーが生成コードをカスタマイズできるように，java2cs コンパイラにヒントを提供するための XML ファイルです。

次は，簡単な hints.xml ファイルの例です。

```
<?xml version="1.0" ?>
<hints>
   <hint>
      <java-class>User</java-class>
      <cs-impl-type>UserData</cs-impl-type>
   </hint>
</hints>
```

上のヒントファイルを使用して java2cs コンパイラを実行するには，コマンドラインに次のように入力します。

```
java2cs -hint_file hints.xml -o User.cs User
```

値型の実装を提供する

次のヒントを使用してコンパイラを実行すると，コンパイラは，User クラスの生成を中止します。

```
<?xml version="1.0" ?>
<hints>
   <hint>
      <java-class>User</java-class>
      <cs-impl-type>User</cs-impl-type>
   </hint>
</hints>
```

これで，目的の User クラスの実装を記述し，生成されたコードとともにコンパイルできるようになります。

デフォルトの実装を別の名前のカスタム実装で置き換える

次のヒントを使用してコンパイラを実行すると，C# 型の名前が User から UserData に変わります。

```
<?xml version="1.0" ?>
<hints>
   <hint>
      <java-class>User</java-class>
      <cs-impl-type>UserData</cs-impl-type>
   </hint>
</hints>
```

上のヒントを使用すると，コンパイラは，User クラスも UserData クラスも生成しなくなります。ただし，その他のクラスは，UserData クラスが実装されると仮定して生成されます。

このサンプルヒントファイルを使用してコードを生成すると，ValueFactory は User クラスを参照しなくなります。かわりに，UserData クラスを参照します。

```
public virtual System.Type GetValueType() {
   return typeof(UserData);
}

public virtual UserData CreateObject() {
   return new UserData();
}
```

```
public virtual void InitObject(UserValueData src_data,
         UserData dst_object) {
   dst_object.Name = src_data.Name;
   dst_object.Password = src_data.Password;
}

public virtual void InitData(UserData src_object,
            UserValueData dst_data) {
   dst_data.Name = src_object.Name;
   dst_data.Password = src_object.Password;
}
```

これで，UserData クラスを記述し（次の例を参照），生成されたコードとともに使用できるようになります。

```
[System.Serializable] public class UserData {

   private string _Name;
   private string _Password;

   public UserData() {
   }

   public UserData(string name, string password) {
      _Name = name;
      _Password = password;
   }

   internal void Init(string name, string password) {
      _Name = name;
      _Password = password;
   }

   public string Name {
      get {
         return _Name;
      }
   }

   public string Password {
      get{
         return _Password;
      }
   }
}
```

このクラスは，そのままでは使用できません。このクラスをコンパイルするには，可視のプロパティ（フィールド）を Name および Password という名前で InitObject と InitData に公開する必要があります。生成された ValueFactory クラスの InitObject と InitData のコードを参照してください。

このように修正するには，可視のプロパティを追加するか，フィールド名を Name および Password に変更し，生成されたコードから可視になるようにします。

これは簡単ですが，クラスフィールドを公開しないで，上記のクラスをそのまま維持することもできます。つまり，InitObject と InitData をそのまま受け取り，ヒントファイルを記述し直す必要があります。

```
<?xml version="1.0" ?>
<hints>
   <hint>
      <java-class>User</java-class>
      <cs-impl-type>UserData</cs-impl-type>
      <mode>custom</mode>
   </hint>
</hints>
```

このヒントファイルと前のヒントファイルの違いは，モードが custom に設定されていることだけです。生成されるコードは，ほとんど変わりません。実際，InitObject メソッドと InitData メソッドだけが異なります。次のように生成されます。

```
public abstract class UserValueFactory : CORBA.ValueFactory {

   public abstract void InitObject(UserValueData src_data,
                  UserData dst_object);

   public abstract void InitData(UserData src_object,
                  UserValueData dst_data);
}
```

クラスとこれらのメソッドが具象でなくなることに注目してください。User 型のファクトリを提供する必要がありますが，これは簡単な実装です。

```
public class UserFactory: UserValueFactory {

   public override void InitObject(UserValueData src_data,
                  UserData dst_object) {
      dst_object.Init(src_data.Name, src_data.Password);
   }

   public override void InitData(UserData src_object,
                  UserValueData dst_data) {
      dst_data.Name = src_object.Name;
      dst_data.Password = src_object.Password;
   }
}
```

生成されたコードの Helper クラスの 1 つが使用されている限り，この ValueFactory は，User Java クラスの ValueFactory として自動的に登録されます。ValueFactory を明示的に登録するには，ORB.RegisterValueFactory() を呼び出します。または，ORB.RegisterAssembly() を呼び出して，提供されているアセンブリ内のすべてのファクトリを登録します。

メソッドを含むインターフェースのマップ

次の Java インターフェースがあるとします。

```
public interface Principal {
   public String getUserName();
}
```

また，次の Java クラスがあります。

```
public class User implements Principal, java.io.Serializable {
   private String name;
   private String password;

   public User(String name, String password) {
      this.name = name;
      this.password = password;
   }

   public String getUserName() {
      return name;
   }
}
```

インターフェースとクラスのどちらにもヒントを使用しないで，これらのインターフェースとクラスに対してコンパイラを実行すると，次の警告が生成されます。

```
java2cs：(警告) 型 Principal が完全に有効になるには，マッピングのヒントが必要です（メソッドシグニチャは無視される)。

java2cs：(警告) 型 User が完全に有効になるには，マッピングのヒントが必要です（メソッドシグニチャは無視される)。
```

この警告は，java2cs コンパイラによって無視されるメソッドがインターフェース (リモートインターフェースではない) にあることを示します。コンパイラは，リモートで起動されるように設計されていないメソッドをマップできないため，これらのメソッドは無視されます。これは，このようなメソッドが受け取るパラメータがローカルのコンテキストでのみ有効になるためです。生成されたコードを確認すると，コンパイラは，Principal に対して次のコードを生成します。

```
public interface Principal {
}
```

さらに，User に対して次のコードを生成します。

```
[System.Serializable] public class User : Principal {
    ...
}
```

コンパイラは，getUserName メソッドに対するコードの生成を無視しました。コンパイラの警告によると，これはほとんど予期しないことなので，ヒントを使用して，これを適切な .NET インターフェースにマップする必要があります。

たとえば，次のヒントファイルを使用するとします。User のヒントは提供しないことに注意してください。

```
<hints>
   <hint>
      <java-class>Principal</java-class>
      <cs-sig-type>IPrincipal</cs-sig-type>
   </hint>
</hints>
```

これは，Principle インターフェースを C# の IPrinciple インターフェース（コンパイラによって生成されない) にマップします。また，次のように IAuthenticatable を .NET コードに追加するとします。System.Security.Principals.IPrincipal などの既存のインターフェースを使用できることに注意してください。

```
public interface IPrincipal {
   string GetName();
}
```

これで，多少よく機能するようになります。生成された User は IPrincipal を拡張します。

```
[System.Serializable] public class User : IPrincipal {
   ...
}
```

コンパイラは，まだ次の警告を生成します。

```
java2cs：(警告) 型 User が完全に有効になるには，マッピングのヒントが必要です（メソッドシグニチャは無視される)。
```

今回，この警告が生成される理由は明らかです。生成される User クラスは，IPrincipal に実装する必要がある GetName という名前のメソッドがあることを認識できません。認識したとしても，メソッドの実装方法はわかりません。

したがって，コンパイラは，実装する必要があるメソッドを含む値クラスを生成するたびに警告を生成しますが，これは正しい動作です。

シグニチャタイプによる実装の詳細の隠蔽

上の例の User 型は，インターフェースを実装しました。インターフェースを実装するクラスを開発することはよくありますが，ここで使用するクラスは，ユーザーに公開されないプライベートな実装です。たとえば，コレクションの Iterator を考えます。Iterator インターフェースはパブリックですが，どの実装も通常は隠蔽され，ユーザーに公開されることはありません。

たとえば，User がそのような型だとすると，ValueFactory はパブリッククラスなので，ValueFactory がシグニチャで実際にその型を公開すると不都合です。これを回避するには，ヒントでシグニチャタイプを使用して，ValueFactory によって公開される要素を制御します。

次のヒントを使用するとします。

```
<hints>
   <hint>
      <java-class>Principal</java-class>
      <cs-sig-type>IPrincipal</cs-sig-type>
   </hint>
   <hint>
      <java-class>User</java-class>
      <cs-sig-type>IPrincipal</cs-sig-type>
      <cs-impl-type>UserData</cs-impl-type>
      <mode>custom</mode>
   </hint>
</hints>
```

次の ValueFactory が生成されます。

```
public abstract class UserValueFactory : CORBA.ValueFactory {

   public virtual System.Type GetValueType() {
      return typeof(UserData);
   }

   public virtual IPrincipal CreateObject() {
      return new UserData();
   }

   public abstract void InitObject(UserValueData src_data,
                  IPrincipal dst_object);

   public abstract void InitData(IPrincipal src_object,
                  UserValueData dst_data);
}
```

ファクトリが使用する実装は UserData ですが，すべてのシグニチャが IPrincipal を使用していることに注意してください。

明示的なファクトリコード

すべてのファクトリコードを自分で記述する方が便利な場合があります。それには, 次のヒントを使用します。

```
<hints>
   <hint>
      <java-class>Principal</java-class>
      <cs-sig-type>IPrincipal</cs-sig-type>
   </hint>
   <hint>
      <java-class>User</java-class>
      <cs-sig-type>UserData</cs-sig-type>
      <mode>custom</mode>
   </hint>
</hints>
```

前に生成されたコードとの違いは，GetValueType メソッドと CreateObject メソッドだけで，これらは抽象メソッドになります。

```
public abstract System.Type GetValueType();

public abstract UserData CreateObject();
```

ここで重要なことは，ヒントで cs-impl-type ではなく cs-sig-type 要素が使用されることです。これは, コンパイラに実装クラスのすべてのリファレンスを除外するように指示します。

ヒントの他の要素を調整することで，コード生成の他の部分を変更できます。たとえば，次のヒントを使用するとします。

```
<hints>
   <hint>
      <java-class>Principal</java-class>
      <cs-sig-type>IPrincipal</cs-sig-type>
   </hint>
   <hint>
      <java-class>User</java-class>
      <cs-sig-type>UserData</cs-sig-type>
   </hint>
</hints>
```

これにより，引き続き次のような InitObject メソッドと InitData メソッドが生成されます。

```
public virtual void InitObject(UserValueData src_data,
            UserData dst_object) {
   dst_object.Name = src_data.Name;
   dst_object.Password = src_data.Password;
}

public virtual void InitData(UserData src_object,
            UserValueData dst_data) {
   dst_data.Name = src_object.Name;
   dst_data.Password = src_object.Password;
}
```

Immutable

前のサンプル UserData クラスを少し変更することにします。次の例では，init メソッドとデフォルトの void コンストラクタを削除しました。

```
[System.Serializable] public class UserData {

   private string _Name;
   private string _Password;

   public UserData(String name, string password) {
      _Name = name;
      _Password = password;
   }

   public string Name {
      get {
         return _Name;
      }
   }

   public string Password {
      get{
         return _Password;
      }
   }
}
```

これは，フィールドを初期化しないと作成できないクラスの例です。また，いったん作成されると，フィールドを初期化する方法はありません。Name フィールドと Password フィールドを設定するメソッドはありませんが，ここでは，ネットワークからオブジェクトの状態を読み取り，その値をオブジェクトの状態として正確に設定する必要があります。

ただし，現在の ValueFactory は，CreateObject メソッドでオブジェクトを作成し，別の手順（InitObject）でオブジェクトを初期化します。ここでは，これが機能しないことは明らかです。このような場合は，ヒントで immutable モードを設定します。

次のヒントを使用します。

```
<hints>
   <hint>
      <java-class>Principal</java-class>
      <cs-sig-type>IPrincipal</cs-sig-type>
   </hint>
   <hint>
      <java-class>User</java-class>
      <cs-sig-type>IPrincipal</cs-sig-type>
      <cs-impl-type>UserData</cs-impl-type>
      <mode>immutable</mode>
   </hint>
</hints>
```

InitObject には次のシグニチャが生成されます。

```
public abstract IPrincipal InitObject(UserValueData src_data);
```

また，CreateObject の呼び出しは生成されなくなります（抽象でもそれ以外でも）。

ここで，InitObject が引数として IPrincipal を受け取ることより，これを返す方法に注目してください。これにより，すでにアンマーシャリングされている値データを使用して UserData を作成し，それを返す ValueFactory を記述できます。

そのような ValueFactory は，次のようになります。

```
public class UserFactory: UserValueFactory {

   public override IPrincipal InitObject(UserValueData src_data);
      return new UserData(src_data.Name, src_data.Password);
   }

   public override void InitData(UserData src_object,
               UserValueData dst_data) {
      dst_data.Name = src_object.Name;
      dst_data.Password = src_object.Password;
   }
}
```

immutable モードを使用する場合は，データオブジェクト（すべてのベースクラスのすべてのデータも含む）のすべての状態を使用して，自由に変換できないオブジェクトを適切に初期化する必要があります。

カスタムマーシャリング

ネットワークにパスワードを書き込む場合は，パスワードを暗号化して，そのまま送信されないようにできます。それには，Java クラスでカスタムマーシャリングを使用する必要があります。

次の Java User クラスを考えます。

```
public class User implements Principal, java.io.Serializable {
   private String name;
   transient private String password;

   public User(String name, String password) {
      this.name = name;
      this.password = password;
   }

   public String getUserName() {
      return name;
   }

   private void writeObject(java.io.ObjectOutputStream s)
      throws java.io.IOException {
      s.defaultWriteObject();
      s.writeObject(encrypt(password));
   }

   private void readObject(java.io.ObjectInputStream s)
      throws java.io.IOException, ClassNotFoundException {
      s.defaultReadObject();
      password = encrypt((String) s.readObject());
   }

   private String encrypt(String val) {
      char[] result = new char[val.length()];
      for (int 1 = 0; i < val.length(); i++) {
         result[i] = (char) (((byte) val.charAt(i)) ^ 0x77);
      }
      return new String(result);
   }
}
```

これは，カスタムマーシャリングされる Java Serializable クラスです。このクラスに対するデフォルトのコード生成（ヒントなし）では，いくつかの変更点があります。値クラスは生成されなくなります。クラスがカスタムマーシャリングされることをコンパイラが認識し，クラス内に適切なフィールドを生成できなくなるためです。ただし，ValueData クラスを生成することは認識しています。このクラスは，デフォルトのマーシャリングを使用する場合にマーシャリングされるフィールド（一時的でないフィールド）を表すためで

す。上のサンプルコードで示されているように，このクラスは，いくつかの追加データもマーシャリングします。

ValueData は，次のように生成されます。

```
public class UserValueData {
   public string Name;
}
```

ValueFactory は，次のように生成されます。

```
public abstract System.Type GetValueType();

public abstract User CreateObject();

public abstract void ReadObject(UserValueData data,
               CORBA.ObjectInputStream input,
               User obj);

public abstract void WriteObject(User obj,
               UserValueData data,
               CORBA.ObjectOutputStream output);

public static void DefaultReadValueData(UserValueData data,
               CORBA.ObjectInputStream input) {
   ...
}

public static void WriteValueData(UserValueData data,
               CORBA.ObjectOutputStream output) {
   ...
}
```

GetValueType メソッドと CreateObject メソッドが抽象メソッドになっていることに注目してください。コンパイラは C# クラスの名前を認識できないため，これらの実装を提供するように要求されます。さらに，InitObject メソッドと InitData メソッドがないことにも注目してください。かわりに，新しく ReadObject メソッドと WriteObject メソッドがあります。適切なカスタムマーシャリングロジックを提供するには，この 2 つのメソッドを実装する必要があります。例を見てわかるように，ValueData オブジェクトと値クラスはそのままメソッドに渡されますが，ほかに Stream も渡されます。これで，カスタムマーシャリングロジックを記述できます。最後に，ユーザーがデフォルトでマーシャリングされたデータを読み書きできるように，いくつかの追加メソッド（DefaultReadValueData と WriteValueData）が生成されます。

Java でカスタムマーシャリングを使用する際の一般的な用途は，マーシャリング時にシリアル化可能フィールドをゆっくりと計算し，アンマーシャリング時に一時的フィールドをゆっくりと初期化することです。実際のマーシャリングは同じままです。カスタムマーシャリングでデフォルトのフィールドを読み書きし，ストリームの最後にいくつかのデータを追加することもあります。

次の UserData の実装を使用する上の Java クラスのサンプル ValueFactory を示します。

```
[System.Serializable] public class UserData {

   private string _Name;
   private string _Password;

   public UserData() {
   }

   public UserData(string name, string password) {
      _Name = name;
      _Password = password;
   }

   internal Init(string name, string password) {
      _Name = name;
      _Password = password;
   }

   public string Name {
      get {
```

```
            return _Name;
         }
      }

      public string Password {
         get{
            return _Password;
         }
      }
   }
```

ValueFactory は次のとおりです。

```
   public class UserFactory : UserValueFactory {

      public override System.Type GetValueType() {
         return typeof(UserData);
      }

      public override UserData CreateObject() {
         return new UserData();
      }

      public string Encrypt(string val) {
         char[] ersult = new char[val.Length];
         for(int i = 0; i < val.Length; i++) {
            result[i] = (char) (((byte) val[i] ^ 0x77);
         }
         return new string(result);
      }

      public override void ReadObject(UserValueData data,
                     CORBA.ObjectInputStream input,
                     User obj) {
         DefaultReadValueData(data, input);
         obj.Init(data.Name, Encrypt(input.ReadString()));
      }

      public override void WriteObject(User obj,
                     UserValueData data,
                     CORBA.ObjectOutputStream output) {
         data.Name = obj.Name;
         DefaultWriteValueData(data, output);
         output.WriteObject(Encrypt(obj.Password));
      }
   }
```

前に示したように，他のヒント技術を使用して，値オブジェクトの名前を変更したり，公開されるシグニチャを変更することができます。また，DefaultWriteValueData の後に追加データを記述したり，DefaultReadValueData の後で同じ追加データを読み取ることもできます。さらに，DefaultWrite/ReadValueData は呼び出す必要はありません。

ヒントファイルのスキーマ

次は，ヒントファイルのスキーマです。

```
<?xml version="1.0" ?>
<xsd:schema
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="hints">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="hint" type="hintType" minOccurs="1"
               maxOccurs="unbounded" />
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:complexType name="hintType">
      <xsd:sequence>
         <xsd:element name="java-class" type="xsd:string"/>
         <xsd:element name="cs-sig-type" type="xsd:string" minOccurs="0"/>
         <xsd:element name="cs-impl-type" type="xsd:string" minOccurs="0"/>
         <xsd:element name="mode" type="modeType" minOccurs="0"/>
      </xsd:sequence>
   </xsd:complexType>
   <xsd:simpleType name="modeType">
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="automatic"/>
         <xsd:enumeration value="custom"/>
         <xsd:enumeration value="immutable"/>
      </xsd:restriction>
   </xsd:simpleType>
</xsd:schema>
```

1 対多マーシャリングの優先順位

VisiBroker for .NET には一連の組み込み value ファクトリがあり，これらの優先順位は事前に決定されています。特定の型のマーシャリング方法があいまいである場合，デフォルトの動作は次のようになります。

```
// ユーザーが型のデフォルトのマーシャリング方法を
// 認識できるように，value ファクトリには決定的な
// 順序が必要です。マーシャリングの優先順位は，
// 登録順序に基づいており，最後に登録された
// ファクトリの優先順位が最も高くなります。

CORBA.ValueFactory[] factories = {

   // 最も優先順位が低いのは JDK 1.4 型です。これらの型は
   // 古い JDK では意味がないためです。
   new J2EE.Factories.LinkedHashMapValueFactory(),
   new J2EE.Factories.LinkedHashSetValueFactory(),

   // 次に優先順位が高いのは，JDK 1.0 型と 1.1 型です。
   // これらはほとんど使用されなくなりました。
   new J2EE.Factories.HashtableValueFactory(),
   new J2EE.Factories.PropertiesValueFactory(),
   new J2EE.Factories.StackValueFactory(),
   new J2EE.Factories.VectorValueFactory(),
```

```
    // 最後は JDK 1.2 型です（関連する JDK 1.3 型は
    // 存在しないことに注意)。
    // 最初は,「最も一般的でない」型です。
    new J2EE.Factories.LinkedListValueFactory(),
    new J2EE.Factories.TreeMapValueFactory(),
    new J2EE.Factories.TreeSetValueFactory(),

    // 最後は,「最も一般的な」型です。
    new J2EE.Factories.HashMapValueFactory(),
    new J2EE.Factories.HashSetValueFactory(),

    // 最後に, ArrayList がこの人気コンテストに優勝します！
    new J2EE.Factories.ArrayListValueFactory(),
};

foreach(CORBA.ValueFactory factory in factories) {
    orb.RegisterValueFactory(factory);
}
```

配列の下の方のアイテムが上の方のアイテムより優先します。もちろん, これでは不都合な場合もあります。別の優先順位が必要な場合は, デフォルトの動作を手作業で上書きする必要があります。最も簡単な方法は，優先する ValueFactory をメインプログラムで明示的に登録します。競合する型（java.util.HashMap など）より java.util.Hashtable を優先させる場合は，メインプログラムに次のコードを入れます。

```
CORBA.ORB orb = CORBA.ORB.Init();
orb.RegisterValueFactory(J2EE.Util.HashtableValueFactory.GetSingleton());
```

ORB.Init は，上で示した ValueFactory の登録などのすべてのデフォルトの ORB 動作を設定します。このデフォルトでは, HashMap ValueFactory が Hashtable ValueFactory より優先します。ただし，ORB を初期化した後で，Hashtable ValueFactory を明示的に登録します。これで，前に登録されたどの ValueFactory より Hashtable ValueFactory が優先されます。

第 8 章

QoS の使用

Quality of Service（QoS）は，ポリシーを使用して，クライアントアプリケーションとその接続先のサーバーとの間の接続を定義および管理します。

QoS の概要

QoS ポリシー管理は，次のような状況で利用できる操作を通して実行されます。

- ORB レベルのポリシーは，局所性制約付きの PolicyManager によって処理されます。この PolicyManager を使用すると，ポリシーを設定したり，現在の Policy オーバーライドを表示することができます。ORB レベルで設定されたポリシーは，システムデフォルトをオーバーライドします。
- スレッドレベルのポリシーは，PolicyCurrent を介して設定されます。PolicyCurrent には，スレッドレベルの Policy オーバーライドを表示および設定するためのオペレーションがあります。スレッドレベルで設定されたポリシーは，システムデフォルト，および ORB レベルで設定された値をオーバーライドします。
- オブジェクトレベルのポリシーを適用するには，ベースオブジェクトのインターフェースの QoS オペレーションを利用します。オブジェクトレベルで適用されたポリシーは，システムデフォルト，および ORB またはスレッドレベルで設定された値をオーバーライドします。

CORBA オブジェクト単位のポリシー設定

CORBA オブジェクト単位で QoS ポリシーを設定するには，CORBA.ObjectOperations メソッドを使用します。CORBA オブジェクト単位で QoS ポリシーを設定するには，次の例のように CORBA オブジェクトを CORBA.ObjectOperations にキャストし，メソッド SetPolicyOverrides_() を呼び出す必要があります。

```
// 排他的接続ポリシーを設定します。
bool deferBind = true;
Any policyValue = orb.CreateAny();
policyValue.InsertBoolean(deferBind);
Policy policies =
   orb.CreatePolicy(EXCLUSIVE_CONNECTION_POLICY_TYPE.Value, policyValue);
```

```
Calc.VisiCalc calc = Calc.VisiCalcHelper.Narrow(
   ((CORBA.ObjectOperations)objRef).SetPolicyOverrides_(
      new Policy [] {orb.CreatePolicy(
         QoSExt.EXCLUSIVE_CONNECTION_POLICY_TYPE.Value, policyValue)},
         SetOverrideType.SET_OVERRIDE));
```

ポリシーオーバーライドと有効なポリシー

有効なポリシーとは，適用可能なポリシーオーバーライドがすべて適用された後で，要求に適用されるポリシーです。有効なポリシーは，IOR によって指定されたポリシーを有効なオーバーライドと比較することによって判定されます。有効なポリシーは，有効なオーバーライドおよび IOR で指定された Policy によって許可された値の共通部分です。共通部分が空の場合は，例外 CORBA.INV_POLICY が生成されます。

QoS のインターフェース

QoS ポリシーの取得および設定には，次のインターフェースが使用されます。

Object

VisiBroker for .NET では，CORBA.Object を拡張して OMG メッセージング仕様で定義されている QoS を追加サポートしています。したがって，公開される Object インターフェースは 2 つあります。

Object のメソッド

CORBA.Object **インターフェース**にある次のメソッドを使用して，有効なポリシーを取得したり，ポリシーオーバーライドを取得または設定します。

GetClientPolicy_

```
CORBA.Policy GetClientPolicy_(int type)
```

オブジェクトリファレンス用に有効なオーバーライド Policy を戻します。有効なオーバーライドを取得するには，指定された PolicyType のオーバーライドをまず Object スコープでチェックし，次に Current スコープ，最後に ORB スコープでチェックします。要求された PolicyType のオーバーライドが存在しない場合は，その PolicyType のシステム依存のデフォルト値が使用されます。デフォルトのポリシー値が指定されないため，可搬性のあるアプリケーションを作成するには，ORB のスコープで必要なデフォルト値を設定する必要があります。

有効な Policy とは要求が実行された場合に使用される Policy です。この Policy は，GetClientPolicy_ によって戻された際に，まず PolicyType の有効なオーバーライドを取得することによって判別されます。

有効なオーバーライドは次に IOR で指定された Policy と比較されます。有効な Policy は，有効なオーバーライドおよび IOR で指定された Policy によって許可された値の共通部分です。共通部分が空の場合は，システム例外 INV_POLICY が発生します。それ以外の場合は，共通部分内で正しい値を指定した Policy が有効な Policy として返されます。IOR 内に Policy 値がない場合は，任意の正しい値が使用できることを意味します。GetPolicy_ の前にオブジェクトリファレンスの NonExistent_ または ValidateConnection_ を呼び出すことで，戻される有効なポリシーの正確性が保証されます。

オブジェクトリファレンスがバインドされる前に GetPolicy_ が呼び出された場合，戻される有効な Policy は実装に依存します。そのような場合，通常の実装は次のいずれかの処理を行います。つまり，例外 CORBA.BAD_INV_ORDER を生成するか，バインディングが実行されると変化する可能性がある PolicyType に対するなんらかの値を返すか，バインディングを試みてから有効なポリシーを返します。

RebindPolicy の値が TRANSPARENT，VB_TRANSPARENT，または VB_NOTIFY_REBIND である場合は，透過的な再バインディングにより，有効なポリシーが呼び出しのたびに変わる可能性があることに注意してください。

パラメータ	説明
type	要求されるポリシーの型

GetPolicy_

```
CORBA.Policy GetPolicy_(int policy_type)
```

オブジェクトリファレンスの有効なポリシー，つまり policy_type パラメータで指定されるタイプの Policy オブジェクトを返します。

パラメータ	説明
policy_type	取得するポリシーのタイプ。

GetPolicyOverrides_

```
CORBA.Policy[] GetPolicyOverrides_(int[] types)
```

Object スコープで設定された（指定されたポリシー型の）Policy オーバーライドのリストを戻します。指定されたシーケンスが空の場合，このスコープでの Policy オーバーライドがすべて戻されます。要求された PolicyType のうち Object スコープでオーバーライドされるものがない場合は，空のシーケンスが返されます。

パラメータ	説明
types	クエリ対象のポリシー型

SetPolicyOverrides_

```
CORBA.Object SetPolicyOverrides_(CORBA.Policy[] policies, CORBA.SetOverrideType set_add)
```

指定された SetOverrideType オブジェクトの値に基づいて，この Object 内の既存のポリシーを指定されたポリシーで置き換えるか，指定されたポリシーを既存のポリシーに追加することにより，新しい Object を返します。

このメソッドは，CORBA.PolicyManager の同じ名前のメソッドと同様の動作を行います。ただし，このメソッドは，要求されたポリシーオーバーライドのリストを使用して，オブジェクト，スレッド，または ORB の現在のポリシーセットを更新します。また，このメソッドは CORBA.Object を返しますが，同じ名前の他のメソッドは void を返します。

パラメータ	説明
policies	追加または置換に使用されるポリシーを含む Policy オブジェクトの配列。
set_add	指定されたポリシーで既存のポリシーを置き換えることを示す SetOverrideType.SET_OVERRIDE，または指定されたポリシーを既存のポリシーに追加することを示す SetOverrideType.ADD_OVERRIDE のいずれか。

ValidateConnection_

```
bool ValidateConnection_(out CORBA.Policy[] inconsistent_policies)
```

オブジェクトの現在の有効なポリシーが呼び出しの実行を許可するかどうかを表すブール値を返します。オブジェクトの現在の有効なポリシーが呼び出しの実行を許可する場合は，TRUE を返します。オブジェクトリファレンスがまだバインドされていない場合は，この操作の一環としてバインディングが行われます。オブジェクトリファレンスがすでにバインドされているが，現在のポリシーオーバーライドが変更されたか，他のなんらかの理由でバインディングが無効になった場合，RebindPolicy オーバーライドの設定とは関係なく，再バインドが試行されます。

ValidateConnection_ オペレーションは，現在の有効な RebindPolicy によって暗黙的な再バインドが使用禁止になった場合，そのような再バインドを強制する唯一の方法です。バインドまたは再バインドの試行は ORB による GIOP LocateRequests の処理を含む場合があります。現在の有効なポリシーがシステム例外 INV_POLICY を発生させる呼び出しを引き起こす場合は，FALSE を返します。

現在の有効なポリシーに互換性がない場合，アウトパラメータ inconsistent_policies には非互換性を引き起こすこれらのポリシーが含まれています。この戻されたポリシーのリストは完全なものであるという保証はありません。ポリシーオーバーライドに無関係なんらかの理由によってバインディングが失敗した場合は，適切なシステム例外が発生します。

パラメータ	説明
inconsistent_policies	呼び出しを実行できない原因となる矛盾ポリシーのリストを戻す出力パラメータ

PolicyManager

CORBA.PolicyManager インターフェースは，ORB レベルで Policy オーバーライドを取得および設定するためのメソッドを提供します。

PolicyManager のメソッド

GetPolicyOverrides

```
CORBA.Policy[] GetPolicyOverrides(int[] ts)
```

このメソッドは，要求された PolicyTypes に対して，すべてのオーバーライドされるポリシーを示す PolicyList シーケンスを返します。指定されたシーケンスが空（リストの長さが 0）の場合，現在のコンテキストレベルのすべての Policy オーバーライドが返されます。要求された PolicyType のうち対象となる PolicyManager でオーバーライドされるものがない場合は，空のシーケンスが返されます。

SetPolicyOverrides

```
void SetPolicyOverrides(CORBA.Policy[] policies, CORBA.SetOverrideType set_add)
```

このメソッドは，要求された Policy オーバーライドのリストを使用して現在のポリシーオーバーライドセットを変更します。空のポリシーシーケンスと SET_OVERRIDE モードを使用して SetPolicyOverrides を呼び出すと，PolicyManager からすべてのオーバーライドが削除されます。

最初の入力パラメータ policies は Policy オブジェクトへの参照のシーケンスです。2 番目のパラメータ set_add は，CORBA.SetOverrideType 型です。このパラメータに ADD_OVERRIDE を使用すると，PolicyManager にすでに存在する他のオーバーライドに指定したポリシーを追加するように指定できます。または，SET_OVERRIDES を使用すると，オーバーライドを含まない PolicyManager に指定したポリシーを追加するように指定できます。

クライアント側でのオペレーションの呼び出しに関する特定のポリシーだけが，このオペレーションを使用してオーバーライドできます。クライアントに適用されないポリシーをオーバーライドしようとすると，例外 CORBA.NO_PERMISSION が生成されます。この要求により，指定された PolicyManager の状態が矛盾する場合，ポリシーは変更または追加され

ず，InvalidPolicies 例外が生成されます。他の PolicyManager で設定されたポリシーとの互換性は評価されません。

パラメータ	説明
policies	Policy オブジェクトへの参照のシーケンス。
set_add	CORBA.SetOverrideType 型のパラメータです。指定したポリシーを PolicyManager にすでに存在する他のオーバーライドに追加するか (ADD_OVERRIDE)，ほかにオーバーライドがない空の PolicyManager に追加するか (SET_OVERRIDE) を指定します。この要求により，指定された PolicyManager の状態が矛盾する場合，ポリシーは変更または追加されず，InvalidPolicies 例外が生成されます。

PolicyCurrent

CORBA.PolicyCurrent インターフェースは，新しいメソッドを追加しないで PolicyManager と Current から派生されます。したがって，PolicyManager インターフェースのオペレーションを PolicyCurrent でも使用できます。これらのメソッドの詳細は，64 ページの「PolicyManager」を参照してください。

PolicyCurrent は，スレッドレベルでオーバーライドされたポリシーへのアクセスを提供します。PolicyCurrent の識別子を指定して ResolveInitialReferences を呼び出すと，スレッドの PolicyCurrent への参照を取得できます。

DeferBindPolicy

QoSExt.DeferBindPolicy は，リモートオブジェクトが初めて作成されたときに ORB がリモートオブジェクトに通信するのか，リモートオブジェクトが初めて呼び出されるまで通信を遅らせるのかを決定します。 デフォルトでは，ORB は，Bind または StringToObject の呼び出し時に（リモート）オブジェクトに接続します。

DeferBindPolicy の有効な値は，TRUE および FALSE です。DeferBindPolicy を TRUE に設定すると，バインドするインスタンスが初めて呼び出されるまで，すべてのバインドが遅延されます。デフォルト値は FALSE です。

クライアントオブジェクトを作成し，DeferBindPolicy を true に設定した場合は，サーバーの起動を最初の呼び出しまで遅延できます。このオプションは，生成されるヘルパークラスの Bind メソッドのオプションとして，以前からありました。

DeferBindPolicy のプロパティ

Value

```
bool Value
```

DeferBindPolicy の現在の設定を返します。

例

次のサンプルコードは，DeferBindPolicy を作成し，ORB でポリシーを設定する例を示しています。

```
public class DeferBindClient {
   static void Main(string[] args) {
      try {
         CORBA.ORB orb = CORBA.ORB.Init(args);

         // フラグとリファレンスを初期化します。
         bool deferMode = true;
         Any policyValue = orb.CreateAny();
         policyValue.InsertBoolean(deferMode);
```

```
            Policy policies =
               orb.CreatePolicy(DEFER_BIND_POLICY_TYPE.Value, policyValue);

            // スレッドマネージャへのリファレンスを取得します。
            PolicyManager orbManager =
               PolicyManagerHelper.Narrow(
                  orb.ResolveInitialReferences("ORBPolicyManager"));

            // ORB レベルでポリシーを設定します。
            orbManager.SetPolicyOverrides(new Policy[] {policies},
               SetOverrideType.SET_OVERRIDE);

            // バインドメソッドを取得します。
            byte[] managerId = orb.StringToObjectId("BankManager");

            Bank.AccountManager manager =
               Bank.AccountManagerHelper.Bind("/qos_poa", managerId);

            // 口座名として Jack B. Quick を使用します。
            string name = "Jack B. Quick";

            // アカウントマネージャに指定した口座を開くように要求します。
            Bank.Account account = manager.Open(name);

            // 口座の残高を取得します。
            float balance = account.Balance();

            // 残高を印刷します。
            Console.WriteLine(
               "¥n The balance in " + name + "'s account is $" + balance);
         }
         catch (Exception e) {
            Console.WriteLine(e);
         }
      }
   }
```

ExclusiveConnectionPolicy

QoSExt.ExclusiveConnectionPolicy は VisiBroker for .NET 固有のポリシーです。これを使用して，指定したサーバーオブジェクトへの排他的（非共有）接続を確立できます。このポリシーには，TRUE または FALSE のブール値を指定できます。このポリシーを TRUE に設定すると，サーバーオブジェクトへの接続が排他的になります。ポリシーを FALSE に設定すると，既存の接続を再利用できる場合は再利用され，再利用できない場合にのみ新しい接続が確立されます。デフォルト値は FALSE です。

ExclusiveConnectionPolicy のプロパティ

Value

```
bool Value
```

ExclusiveConnectionPolicy の現在の設定を返します。

RelativeConnectionTimeoutPolicy

QoSExt.RelativeConnectionTimeoutPolicy は，有効なエンドポイントの 1 つを使用してオブジェクトへの接続を試みる場合に，その試行を中止するまでのタイムアウト値を指定できます。タイムアウトは，オブジェクトへの接続方法が HTTP トンネリングしかないようにファイアウォールで保護されているオブジェクトで発生する可能性があります。

メモ このポリシーは，インプロセス通信には適用されません。

このポリシー値は unsigned long long 型で，タイムアウトを 100 ナノ秒単位で指定します。この値は，ORB が接続を試みるすべてのエンドポイントに適用されます。したがって，

複数の接続が試みられた場合，経過時間は，設定されたタイムアウトの倍数になります。**精度は，Java 仮想マシンの実装によっても制限されます。**

RelativeConnectionTimeoutPolicy のメソッド

RelativeExpiry

```
long RelativeExpiry()
```

タイムアウトを 100 ナノ秒の倍数単位で取得します。

例

次のコード例は，RelativeConnectionTimeoutPolicy の作成方法を示しています。

```
public class ConnClient {
   static void Main(string [] args) {
      try {
         // ORB を初期化します。
         ORB orb = ORB.Init(args);

         // マネージャの ID を取得します。
         byte[] managerId = orb.StringToObjectId("BankManager");

         string name = "Jack B. Quick";

         // タイムアウトを 100 ナノ秒単位で指定します。
         // 20 秒のタイムアウトを設定するには，20 * 10^7 ナノ秒を設定します
         int connTimeout = 20;
         Any ctopolicyValue =  orb.CreateAny();

         ctopolicyValue.InsertUlonglong(connTimeout * 10000000);

         Policy ctoPolicy = orb.CreatePolicy(
            RELATIVE_CONN_TIMEOUT_POLICY_TYPE.Value, ctopolicyValue);

         PolicyManager orbManager = PolicyManagerHelper.Narrow(
            orb.ResolveInitialReferences("ORBPolicyManager"));

         orbManager.SetPolicyOverrides(new Policy [] {ctoPolicy},
            SetOverrideType.SET_OVERRIDE);

         // AccountManager を検索します
         // 完全な POA 名とサーバント ID を指定します
         AccountManager source =
            AccountManagerHelper.Bind("/qos_poa", managerId);

         Account account = source.Open(name);
         float balance = account.Balance();
         Console.WriteLine("The balance in {0}'s account is {1}$", name, balance);
      }
      catch (Exception e) {
         Console.WriteLine(e);
      }
   }
}
```

RebindPolicy

Messaging.RebindPolicy は，クライアント側の ORB が閉じた接続，GIOP ロケーション転送メッセージ，およびオブジェクト障害を処理する方法を決定します。ORB は，CORBA.Object インスタンスの有効なポリシーを調べることで，フェイルオーバー，再バインド，および再接続を処理します。

OMG によって定義されたポリシー値は，ORB がターゲットサーバーに正しくバインドされた後で，透過的に再バインドするかどうかを決定します。拡張されたポリシー値は，ORB

がターゲットオブジェクトに正しくバインドされた後で，透過的にフェイルオーバーするかどうかを決定します。

RebindPolicy はクライアント側のみのポリシーです。

メモ RebindPolicy は，オブジェクトに正しくバインドされた後にのみ適用されます。GIOP ベースのプロトコルの場合，LocateRequest メッセージの結果として OBJECT_HERE というステータスの LocateReply メッセージが戻される状態になると，オブジェクトリファレンスはバインドされているとみなされます。

RebindPolicy は，クライアント側でのみ設定されます。RebindPolicy は，6 つの値のうちの 1 つをとります。これらの値により，接続の切断，オブジェクト転送要求，またはオブジェクト障害の場合の動作が決まります。RebindPolicy は，再バインドの際のクライアントの動作を定義するために，以下の定数を受け入れます。

現在サポートされている値は次のとおりです。

- Messaging.TRANSPARENT は，リモート要求の実行時に ORB がオブジェクト転送および必要な再接続をサイレントに処理できるようにします。
- Messaging.NO_REBIND は，リモート要求を行う際に ORB が閉じている接続を再度開く動作をサイレントに処理できるようにしますが，クライアントから見える有効な QoS ポリシーの変更をもたらすような透過的オブジェクト転送は防止されます。RebindMode を NO_REBIND に設定すると，明示的な再バインドだけが許可されます。
- Messaging.NO_RECONNECT は，オブジェクト転送のサイレント処理または閉じた接続を再度開く処理を ORB に禁じます。RebindMode を NO_RECONNECT に設定した場合は，再バインドや再接続を明示的に行う必要があります。
- QoSExt.VB_TRANSPARENT はデフォルトのポリシーです。TRANSPARENT の機能を拡張して，暗黙的と明示的のどちらのバインディングの場合でも，透過的な再バインドを許可します。
- QoSExt.VB_NOTIFY_REBIND は，再バインドが必要な場合に例外を生成します。クライアントは，この例外を受け取ると，2 度目の呼び出しでバインドを行います。
- QoSExt.VB_NO_REBIND はフェイルオーバーを有効にしません。クライアントの ORB は，閉じた接続を同じサーバーに向けて再オープンできるだけで，どのようなオブジェクト転送も許可されません。

メモ クライアントの有効なポリシーが VB_TRANSPARENT であり，クライアントのサーバーが状態データを保持している場合は，次の点に注意が必要です。VB_TRANSPARENT を使用すると，クライアントは，サーバーの変更に気付かないまま新しいサーバーに接続されます。また，元のサーバーが保持していた状態データは失われます。

次の表は，RebindMode 型とその動作の一覧です。

表 8.1 RebindMode ポリシー

RebindMode 型	同じオブジェクトに対する閉じた接続の再確立	オブジェクトの転送	オブジェクトのフェイルオーバー
NO_RECONNECT	いいえ。REBIND 例外が生成されます。	いいえ。REBIND 例外が生成されます。	いいえ
NO_REBIND	はい	はい（ポリシーと一致する場合）いいえ。REBIND 例外が生成されます。	いいえ
TRANSPARENT	はい	はい	いいえ
VB_NOTIFY_REBIND	はい	はい	はい。VB_NOTIFY_REBIND は，障害の検出後に例外を生成し，後続の要求でフェイルオーバーを試みます。
VB_TRANSPARENT	はい	はい	はい。透過的に行われます。

[1] 通信で障害が発生したり，オブジェクトでエラーが発生した場合は，該当する CORBA 例外が生成されます。

例

次のサンプルコードは，タイプが TRANSPARENT の RebindPolicy を作成し，ORB レベル，スレッドレベル，およびオブジェクトレベルでそのポリシーを設定します。

```
using System;
using System.IO;
using CORBA;
using QoSExt;
using Messaging;
using Bank;

public class TransparentClient {
   static void Main(string[] args) {
      try {
         short rebindMode = Messaging.TRANSPARENT.Value;

         // 初期化，ORB の
         CORBA.ORB orb = CORBA.ORB.Init(args);

         // オブジェクト ID を取得します
         byte[] managerId = orb.StringToObjectId("BankManager");

         // AccountManager を検索します。完全な POA 名とオブジェクト ID を指定します
         Bank.AccountManager manager =
            Bank.AccountManagerHelper.Bind("/qos_poa", managerId);
         string s = orb.ObjectToString(manager);
         CORBA.Object obj = orb.StringToObject(s);

         // クライアント側のポリシーを作成し，サーバー側 ORB によって
         // 生成される TRANSIENT 例外を受け取ることができるようにします
         Any policyValue = orb.CreateAny();
         RebindModeHelper.Insert(policyValue, rebindMode);

         Policy myRebindPolicy =
            orb.CreatePolicy(REBIND_POLICY_TYPE.Value, policyValue);

         // AccountManager オブジェクトでポリシーを設定します。
         Bank.AccountManager manager = Bank.AccountManager.Narrow(
            ((CORBA.ObjectOperations)obj.SetPolicyOverrides_(
               new Policy [] {orb.CreatePolicy(
                  QoSExt.EXCLUSIVE_CONNECTION_POLICY_TYPE.Value, policyValue)},
                  SetOverrideType.SET_OVERRIDE));
```

```
            //ORB ポリシーマネージャへのリファレンスを取得します。
            PolicyManager orbManager = null;

            try {

               orbManager =
                  PolicyManagerHelper.Narrow(orb.ResolveInitialReferences(
                     "ORBPolicyManager"));
            }
            catch (CORBA.ORBNS.InvalidName e) {
            }

            // スレッドごとのマネージャへのリファレンスを取得します。
            CORBA.PolicyManager current = null;
            try {
               current =
                  PolicyManagerHelper.Narrow(orb.ResolveInitialReferences(
                     "PolicyCurrent"));
            }
            catch (CORBA.ORBNS.InvalidName e) {
            }

            //ORB レベルでポリシーを設定します。
            try {
               orbManager.SetPolicyOverrides(new Policy[] {myRebindPolicy},
                  SetOverrideType.SET_OVERRIDE);
            }
            catch (CORBA.InvalidPolicies e) {
            }

            // スレッドレベルでポリシーを設定します。
            try {
               current.SetPolicyOverrides(new Policy[] {myRebindPolicy},
                  SetOverrideType.SET_OVERRIDE);
            }
            catch (CORBA.InvalidPolicies e) {
            }

            CORBA.Object oldObjectReference =
               Bank.AccountManagerHelper.Bind("/qos_poa", managerId);

            CORBA.Object newObjectReference =
               ((CORBA.ObjectOperations)oldObjectReference).SetPolicyOverrides_(
                  new Policy [] {myRebindPolicy}, SetOverrideType.SET_OVERRIDE);
         }
         catch (Exception e) {
            Console.WriteLine(e);
         }
      }
   }
```

RebindForwardPolicy

QoSExt.RebindForwardPolicy は, LOCATION_FORWARD の間に接続に障害が発生した場合，クライアント ORB が再バインドを試みるかどうかを決定します。クライアントが新しいオブジェクトに転送される場合は, 新しい送信先オブジェクトへの接続が試行されます。これに失敗すると，次の場合，ORB は透過的に元のオブジェクト（転送元）に接続し直そうとします。

- この時点で，転送数がこのポリシーの forward_count に指定された値を超えていない場合。
- 同じ送信先オブジェクトへの接続を連続して 2 回試みていない場合。

↓üvbroker.orb.rebindForward プロパティが forward_count at the ORB レベルの値に設定されている場合。QoS ポリシーの場合と同様に，forward_count 値は，プログラムによって ORB, スレッド, またはオブジェクトのレベルで上書きできます。このプロパティのデフォルト値 0 は，制限値が指定されていないことを示しています。

RebindForwardPolicy のメソッド

ForwardCount

```
short ForwardCount()
```

RebindForward ポリシーの forward_count の現在の設定を戻します。

RelativeRequestTimeoutPolicy

Messaging.RelativeRequestTimeoutPolicy は，要求またはその応答を配信するための相対的な時間を示します。この時間を超えると，要求はキャンセルされます。このポリシーは，同期および非同期の両方の起動に適用されます。指定したタイムアウト内に要求が完了したとすると，応答がタイムアウトによって破棄されることはありません。タイムアウト値は，100 ナノ秒単位で指定します。

例

次のコードは，RelativeRequestTimeoutPolicy の作成方法を示しています。

```
public class RequestTimeoutClient {
   static void Main(string[] args) {
      try {

         CORBA.ORB orb = CORBA.ORB.Init(args);

         // オブジェクト ID を取得します
         byte[] managerId = orb.StringToObjectId("BankManager");

         // AccountManager を検索します。完全な POA 名とオブジェクト ID を指定します
         Bank.AccountManager manager =
            Bank.AccountManagerHelper.Bind("/qos_poa", managerId);

         string s = orb.ObjectToString(manager);

         // タイムアウトを 100 ナノ秒単位で指定します。
         // 50 秒のタイムアウトを設定するには，50 * 10^7 ナノ秒を設定します
         int reqTimeout = 20;
         CORBA.Any policyValue =  orb.CreateAny();
         policyValue.InsertUlonglong(reqTimeout * 10000000);

         //RelativeRequestTimeoutPolicy を設定します
         CORBA.Policy reqPolicy = orb.CreatePolicy(
            RELATIVE_REQ_TIMEOUT_POLICY_TYPE.Value, policyValue);

         // スレッドマネージャへのリファレンスを取得します。
         PolicyManager orbManager = PolicyManagerHelper.Narrow(
            orb.ResolveInitialReferences("ORBPolicyManager"));

         //ORB レベルでポリシーを設定します。
         orbManager.SetPolicyOverrides(new Policy[] {reqPolicy},
            SetOverrideType.SET_OVERRIDE);
      }
      catch (Exception e) {
         Console.WriteLine(e);
      }
   }
}
```

RelativeRoundTripTimeoutPolicy

Messaging.RelativeRoundtripTimeoutPolicy は，要求またはその応答を配信するための相対的な時間を指定します。この時間が経過しても応答が送信されない場合，要求はキャンセルされます。また，要求がすでに送信済みで応答が送信先から返されている場合，この時間が経過すると応答は破棄されます。このポリシーは，同期および非同期の両方の起動に

適用されます。指定したタイムアウト内に要求が完了したとすると,応答がタイムアウトによって破棄されることはありません。タイムアウト値は，100 ナノ秒単位で指定します。

例

次のコードは，RelativeRoundTripTimeoutPolicy の作成方法を示しています。

```
public class RoundtripTimeoutClient {
   static void Main(string[] args) {
      try {

         CORBA.ORB orb = CORBA.ORB.Init(args);

         // オブジェクト ID を取得します
         byte[] managerId = orb.StringToObjectId("BankManager");

         // AccountManager を検索します。完全な POA 名とオブジェクト ID を指定します
         Bank.AccountManager manager =
            Bank.AccountManagerHelper.Bind("/qos_poa", managerId);

         string s = orb.ObjectToString(manager);

         // タイムアウトを 100 ナノ秒単位で指定します。
         // 20 秒のタイムアウトを設定するには，20 * 10^7 ナノ秒を設定します
         int rttTimeout = 50;
         Any policyValue = orb.CreateAny();
         policyValue.InsertUlonglong(rttTimeout * 10000000);

         // ポリシーを作成します。
         CORBA.Policy rttPolicy =
            orb.CreatePolicy(RELATIVE_RT_TIMEOUT_POLICY_TYPE.Value,
               policyValue);

         // スレッドマネージャへのリファレンスを取得します。
         PolicyManager orbManager =
            PolicyManagerHelper.Narrow(
               orb.ResolveInitialReferences("ORBPolicyManager"));

         // ORB レベルでポリシーを設定します。
         orbManager.SetPolicyOverrides(new Policy[] {rttPolicy},
            SetOverrideType.SET_OVERRIDE);
      }
      catch (Exception e) {
         Console.WriteLine(e);
      }
   }
}
```

SyncScopePolicy

Messaging.SyncScopePolicy は，要求の対象に関する要求の同期レベルを定義します。このインターフェースは，CORBA.Policy から派生されるローカルオブジェクトです。

SyncScope 型の値を SyncScopePolicy と組み合わせて使用して，一方向オペレーションの動作を制御します。これは,オペレーション要求の宛先に関わる同期化スコープを指定する一方向オペレーションに適用されます。一方向ではないオペレーションが呼び出された場合は無視されます。

DII の実装では，オペレーションが一方向として宣言されているかどうかを確認するためにインターフェース定義を調べる必要はないため，このポリシーは DII で INV_NO_RESPONSE フラグを使用した場合にも適用されます。

デフォルトの SyncScopePolicy は，SYNC_WITH_TRANSPORT です。

ORB 実装間の可搬性を保証するために，ORB レベルの SyncScopePolicy をアプリケーションで明示的に設定する必要があります。SyncScopePolicy のインスタンスを作成すると，Messaging.SyncScope 型の値が CORBA.ORB.CreatePolicy に渡されます。このポリシーは，クライアント側のオーバーライドとしてのみ適用可能です。

次の表は，SyncScope の値とその動作の一覧です。

表 8.2　SyncScope の有効な値

SyncScope のタイプ	説明
SYNC_WITH_TRANSPORT	デフォルト。ORB は，トランスポートが要求メッセージを受け入れた後でのみクライアントに制御を戻します。要求が配信される保証はありませんが，トランスポートの特性に関する知識と組み合わせて，実用的なレベルの保証を得ることができます。サーバーからの応答はないため，このレベルの同期化での位置転送はできません。
SYNC_NONE	ORB は，クライアントに制御を戻し（たとえば，メソッドの呼び出しから戻り）てから要求メッセージをトランスポートプロトコルに渡します。クライアントはブロックされないことが保証されます。サーバーからの応答はないため，このレベルの同期化での位置転送はできません。
SYNC_WITH_SERVER	サーバー側の ORB は宛先実装を呼び出す前に応答を送信します。NO_EXCEPTION という応答が送信された場合，必要な位置転送はいずれも実行済みです。この応答を受け取ると，クライアント側の ORB はクライアントアプリケーションに制御を戻します。クライアントはすべての位置転送が完了するまでブロックされます。 POA を使用するサーバーの場合，応答は ServantManager を呼び出してから宛先 Servant に要求が配信されるまでの間に送信されます。
SYNC_WITH_TARGET	CORBA 2.2 の同期的非一方向オペレーションと同等です。呼び出された操作を宛先が完了すると，サーバー側の ORB は応答メッセージだけを送信します。LOCATION_FORWARD 応答は宛先を呼び出す前にすでに送信されています。SYSTEM_EXCEPTION 応答は（例外の意味に応じて）随時送信可能です。この操作は一方向で宣言されていますが，実際のオペレーションは同期操作として動作します。同期化がこの形式になっていると，宛先がすでに要求を確認して処理していることをクライアントが確実に知ることができます。CORBA 2.2 では，この最高レベルの同期化でのみ OTS を使用できます。より低いレベルの同期化で呼び出された操作の場合，その操作先はクライアントの現在のトランザクションに参加できません。

QoS の例外

- CORBA.INV_POLICY は，Policy オーバーライドどうしに互換性がない場合に生成されます。
- CORBA.REBIND は，RebindPolicy の値が NO_REBIND，NO_RECONNECT，または VB_NOTIFY_REBIND であり，バインドされたオブジェクトリファレンスによる呼び出しの結果，オブジェクト転送メッセージまたはロケーション転送メッセージが生成された場合に生成されます。
- CORBA.PolicyError は，要求された Policy がサポートされていない場合に生成されます。
- CORBA.InvalidPolicies は，オペレーションに PolicyList シーケンスが渡された場合に生成されることがあります。例外本体には，そのシーケンスの有効でないポリシーが保持されます。ポリシーが有効でない理由は，現在のスコープ内ですでにオーバーライドされているか，要求された他のポリシーと両立しないかのどちらかです。

第 9 章

動的に管理される型の使い方

DynAny インターフェースは，実行時に基本データ型と構造データ型を動的に作成する方法を提供します。また，コンパイル時には Any オブジェクトが保持する型をサーバーがわからない場合でも，Any オブジェクトの情報を解釈したり，抽出することができます。DynAny インターフェースを使用すると，実行時にデータ型を作成および解釈できる効果的なクライアント／サーバーアプリケーションを構築できます。

DynAny 型

DynAny オブジェクトには，基本データ型（bool，int，float など）または構造データ型のいずれかの値が関連付けられます。DynAny インターフェースは，含まれているデータ型を判定するメソッドと，プリミティブデータ型の値を設定および抽出するメソッドを提供します。

構造データ型は，次のインターフェースによって表されますが，これらはすべて DynAny から派生されます。これらのインターフェースは，それぞれが保持する値を設定および抽出するために適した一連のメソッドを個別に提供します。

表 9.1　構造データ型を表す DynAny から派生するインターフェース

インターフェース	TypeCode	説明
DynArray	_tk_array	同じデータ型の値の配列。要素数は固定。
DynEnum	_tk_enum	単一の列挙の値。
DynFixed	_tk_fixed	サポートされていません。
DynSequence	_tk_sequence	同じデータ型の値のシーケンス。要素数は増減できます。
DynStruct	_tk_struct	構造体。
DynUnion	_tk_union	共用体。
DynValue	_tk_value	サポートされていません。

使用上の制限

DynAny オブジェクトは，オブジェクトの作成元のプロセスによってローカルでのみ使用できます。DynAny オブジェクトをバインドされたオブジェクトに対するオペレーション要求

のパラメータとして使用したり，ObjectToString メソッドを使用して DynAny オブジェクトを外部化すると，MARSHAL 例外が生成されます。

さらに，DynAny オブジェクトをパラメータとして DII 要求で使用すると，NO_IMPLEMENT 例外が生成されます。

DynAny の作成

DynAny オブジェクトを作成するには，DynAnyFactory オブジェクトのオペレーションを呼び出します。まず DynAnyFactory オブジェクトへの参照を取得し，次にそのオブジェクトを使用して新しい DynAny オブジェクトを作成します。

DynAny 内の値の初期化と使用

VisiBroker for .NET の DynAny.Insert<Type> メソッドを使用すると，DynAny オブジェクトをさまざまな基本データ型で初期化できます。ここで，<Type> は，bool，octet，char などのデータ型です。DynAny に定義されている TypeCode に一致しない型を挿入しようとすると，TypeMismatch 例外が生成されます。

VisiBroker for .NET の DynAny.Get<Type> メソッドを使用すると，DynAny オブジェクトに含まれる値にアクセスできます。ここで，<Type> は，bool，octet，char などのデータ型です。DynAny に定義されている TypeCode に一致しない DynAny コンポーネントの値にアクセスしようとすると，TypeMismatch 例外が生成されます。

DynAny インターフェースは，Any オブジェクトをコピー，代入，変換するためのメソッドも提供します。

構造データ型

以下の型は，DynAny インターフェースから派生され，構造データ型を表すために使用されます。

構造データ型内の複数の要素間の移動

DynAny から派生されるインターフェースは，実際に複数のコンポーネントを持つ場合があります。DynAny インターフェースには，それらのコンポーネントを反復処理するためのメソッドが用意されています。複数のコンポーネントを持つ DynAny 派生オブジェクトは，現在のコンポーネントを示すポインタを保持しています。

DynAny のメソッド	説明
Rewind	現在のコンポーネントポインタを最初のコンポーネントにリセットします。オブジェクトにコンポーネントが 1 つしかない場合は，何も効果がありません。
Next	ポインタを次のコンポーネントに進めます。次のコンポーネントがないか，オブジェクトにコンポーネントが 1 つしかない場合は，FALSE が返されます。
CurrentComponent	DynAny オブジェクトを返します。コンポーネントの TypeCode に基づいて，このオブジェクトを適切な型に narrow（キャスト）できます。
Seek	現在のコンポーネントポインタを 0 から始まるインデックスで指定される特定のコンポーネントに設定します。指定されたインデックスにコンポーネントがない場合は，FALSE が返されます。負のインデックスを指定した場合，現在のコンポーネントポインタは -1（コンポーネントなし）に設定されます。

DynEnum

このインターフェースは，単一の列挙型定数を表します。この値を文字列または整数値として設定および取得するためのメソッドが提供されます。

DynStruct

このインターフェースは，動的に構築される struct 型を表します。この構造体のメンバーは，NameValuePair オブジェクトのシーケンスを使用して取得または設定できます。各 NameValuePair オブジェクトには，メンバーの名前，およびメンバーの型と値を含む Any が含まれます。

Rewind，Next，CurrentComponent，Seek の各メソッドを使用して，構造体内のメンバー間を移動できます。構造体のメンバーを設定および取得するためのメソッドが提供されています。

DynUnion

このインターフェースは union を表し，2 つのコンポーネントを含みます。最初のコンポーネントはディスクリミネータを表し，2 番目のコンポーネントはメンバー値を表します。

Rewind，Next，CurrentComponent，Seek の各メソッドを使用して，コンポーネント間を移動できます。共用体のディスクリミネータとメンバー値を設定および取得するためのメソッドが提供されています。

DynSequence と DynArray

DynSequence または DynArray は，どちらも基本データ型または構造データ型のシーケンスを表します。このとき，そのシーケンスや配列内のコンポーネントごとに個別の DynAny オブジェクトを生成する必要はありません。DynSequence 内のコンポーネントの数は変更できますが，DynArray 内のコンポーネントの数は固定です。

Rewind，Next，CurrentComponent，Seek の各メソッドを使用して，DynArray または DynSequence のメンバー間を移動できます。

第 10 章

ポータブルインターセプタの使い方

このセクションでは，OMG 仕様で定義されている新しいポータブルインターセプタの概要について説明します。ポータブルインターセプタのサンプルコードは，VisiBroker for .NET インストールに収録されています。

ポータブルインターセプタの概要

VisiBroker for .NET には，*インターセプタ*と呼ばれる一連のインターフェースが用意されています。インターセプタは，セキュリティ，トランザクション，ログなどの ORB の追加機能を組み込むためのフレームワークです。これらのインターセプタインターフェースは，*コールバックメカニズム*に基づいています。たとえば，インターセプタを使用すると，クライアントとサーバーの間の通信を検知し，必要であればこれらの通信を変更することで，実質的に ORB の動作を変更できます。

最も簡単な使用方法として，インターセプタはコードのトレースに役立ちます。クライアントとサーバーの間で送信されるメッセージを監視できるため，ORB がどのように要求を処理しているかを正確に特定できます。

監視ツールやセキュリティ層などさらに高度なアプリケーションを構築する場合，インターセプタは，このようなより低いレベルのアプリケーションの動作に必要な情報と制御を提供します。たとえば，さまざまなサーバーのアクティビティを監視し，負荷分散を実行するアプリケーションを開発できます。

ポータブルインターセプタの種類

OMG 仕様で定義されているポータブルインターセプタには，次の 2 種類があります。

- **リクエストインターセプタ**は，ORB サービスがクライアントとサーバーの間でコンテキスト情報を転送できるようにします。リクエストインターセプタには，*クライアントリクエストインターセプタ*と*サーバーリクエストインターセプタ*があります。
- **IOR インターセプタ**は，サーバーまたはオブジェクトの ORB サービス関連の機能を記述する情報を ORB サービスが IOR に追加できるようにします。たとえば，SSL などのセキュリティサービスがタグ付きコンポーネントを IOR に追加することで，そのコン

ポーネントを認識するクライアントは，コンポーネント内の情報に基づいてサーバーとの接続を確立できます。

ポータブルインターセプタのクラスとインターフェース

すべてのポータブルインターセプタは，次のベースインターセプタ API クラスの 1 つを実装します。これらの API クラスは，VisiBroker for .NET によって定義および実装されています。

- `ClientRequestInterceptor`
- `ServerRequestInterceptor`
- `IORInterceptor`

Interceptor クラス

上のインターセプタクラスはすべて，共通のクラス `Interceptor` から派生されています。この `Interceptor` クラスには，継承先のクラスで使用できる一般的なメソッドが定義されています。

リクエストインターセプタ

サービスがクライアントとサーバーの間でコンテキスト情報を転送できるように，*リクエストインターセプタ*を使用して，特定のインターセプトポイントで要求／応答シーケンスの流れをインターセプトします。各インターセプトポイントで ORB は，インターセプタが要求情報にアクセスするためのオブジェクトを提供します。リクエストインターセプタには次の 2 種類があり，それぞれに対応する要求情報インターフェースがあります。

- `ClientRequestInterceptor` と `ClientRequestInfo`
- `ServerRequestInterceptor` と `ServerRequestInfo`

ClientRequestInterceptor

`ClientRequestInterceptor` のインターセプトポイントは，クライアント側で実装されます。次の表に示すように，OMG によって `ClientRequestInterceptor` で定義されているインターセプトポイントは 5 つあります。

表 10.1 ClientRequestInterceptor のインターセプトポイント

インターセプトポイント	説明
`SendRequest`	要求がサーバーに送信される前に，クライアント側インターセプタが要求を照会し，サービスコンテキストを変更できるようにします。
`SendPoll`	TII（Time-Independent Invocation）[1] ポーリングが応答シーケンスを取得する間に，クライアント側インターセプタが要求を照会できるようにします。
`ReceiveReply`	応答情報がサーバーから戻され，クライアントが制御を獲得する前に，クライアント側インターセプタが応答情報を照会できるようにします。
`ReceiveException`	例外が発生してクライアントに送信される前に，クライアント側インターセプタが例外の情報を照会できるようにします。
`ReceiveOther`	通常の応答以外の要求結果や例外を受け取った場合に，クライアント側インターセプタが使用可能な情報を照会できるようにします。

[1]TII は，VisiBroker for .NET では実装されません。したがって，`SendPoll( )` インターセプトポイントは呼び出されません。

クライアント側の規則

次にクライアント側の規則を示します。

- 開始インターセプトポイントは，SendRequest と SendPoll です。特定の要求／応答シーケンスにおいて，このどちらか一方のインターセプトポイントだけが呼び出されます。
- 終了インターセプトポイントは，ReceiveReply，ReceiveException，および ReceiveOther です。
- 中間インターセプトポイントはありません。
- 終了インターセプトポイントは，SendRequest または SendPoll の実行が成功した場合にのみ呼び出されます。
- ReceiveException は，ORB のシャットダウンが原因で要求がキャンセルされた場合に，マイナーコード 4（ORB のシャットダウン）のシステム例外 BAD_INV_ORDER とともに呼び出されます。
- ReceiveException は，要求がなんらかの理由でキャンセルされた場合に，マイナーコード 3 のシステム例外 TRANSIENT とともに呼び出されます。

正常な呼び出し	SendRequest の後に ReceiveReply が続きます。開始ポイントの後に終了ポイントが続きます。
再試行	SendRequest の後に ReceiveOther が続きます。開始ポイントの後に終了ポイントが続きます。

ServerRequestInterceptor

ServerRequestInterceptor のインターセプトポイントは，サーバー側で実装されます。ServerRequestInterceptor で定義されているインターセプトポイントは 5 つあります。次の表に，ServerRequestInterceptor のインターセプトポイントを示します。

表 10.2 ServerRequestInterceptor のインターセプトポイント

インターセプトポイント	説明
ReceiveRequestServiceContexts	サーバー側インターセプタが着信要求からサービスのコンテキスト情報を取得し，PortableInterceptor.Current のスロットに転送できるようにします。
ReceiveRequest	オペレーションパラメータを含むすべての情報が使用可能になったら，サーバー側インターセプタが要求情報を照会できるようにします。
SendReply	対象のオペレーションが呼び出されて応答がサーバーに戻される前に，サーバー側インターセプタが応答情報を照会し，応答のサービスコンテキストを変更できるようにします。
SendException	例外が発生してクライアントに送信される前に，サーバー側インターセプタが例外の情報を照会し，応答のサービスコンテキストを変更できるようにします。
SendOther	通常の応答以外の要求結果や例外を受け取った場合に，サーバー側インターセプタが使用可能な情報を照会できるようにします。

サーバー側の規則

次にサーバー側の規則を示します。

- 開始インターセプトポイントは，ReceiveRequestServiceContexts です。このインターセプトポイントは，すべての要求／応答シーケンスで呼び出されます。
- 終了インターセプトポイントは，SendReply，SendException，および SendOther です。特定の要求／応答シーケンスにおいて，このどちらか一方のインターセプトポイントだけが呼び出されます。

- 中間インターセプトポイントは，ReceiveRequest です。これは，ReceiveRequestServiceContexts の後で，終了インターセプトポイントの前に呼び出されます。
- 例外では，ReceiveRequest は呼び出されません。
- 終了インターセプトポイントは，ReceiveRequestServiceContext の実行が成功した場合にのみ呼び出されます。
- SendException は，ORB のシャットダウンが原因で要求がキャンセルされた場合に，マイナーコード 4（ORB のシャットダウン）のシステム例外 BAD_INV_ORDER とともに呼び出されます。
- SendException は，要求がなんらかの理由でキャンセルされた場合に，マイナーコード 3 のシステム例外 TRANSIENT とともに呼び出されます。

正常な呼び出し　インターセプトポイントの順序は，ReceiveRequestServiceContexts，ReceiveRequest，SendReply です。つまり，開始ポイント，中間ポイント，終了ポイントの順に続きます。

IORInterceptor

アプリケーションで IORInterceptor を使用すると，クライアントの ORB サービス実装が正しく機能するように，サーバーまたはオブジェクトの ORB サービス関連機能に関する情報をオブジェクトリファレンスに追加できます。それには，インターセプトポイント EstablishComponents を呼び出します。このインターセプトポイントには IORInfo のインスタンスが渡されます。

PortableInterceptor（PI）Current

PortableInterceptor.Current オブジェクト（以下 PICurrent）は，現在アクティブな要求コンテキストにスレッド固有の情報を関連付けるためにポータブルインターセプタの実装によって使用されるスロットのテーブルです。PICurrent の使用は任意ですが，通常，インターセプタ内でクライアントのスレッド固有の情報が必要な場合に使用されます。

PICurrent は，次の呼び出しを介して取得します。

```
ORB.ResolveInitialReferences("PICurrent");
```

Codec

Codec は，インターセプタが IDL データ型形式と CDR カプセル化形式の間でコンポーネントを転送するためのメカニズムです。

CodecFactory

このクラスを使用し，エンコーディング形式のメジャーバージョンとマイナーバージョンを指定して，Codec オブジェクトを作成します。CodecFactory は，次の呼び出しを介して取得します。

```
ORB.ResolveInitialReferences("CodecFactory");
```

ポータブルインターセプタの作成

ポータブルインターセプタを作成する一般的な方法は次のとおりです。

1 インターセプタは，次のインターセプタインターフェースのいずれか 1 つから継承される必要があります。
   - ClientRequestInterceptor
   - ServerRequestInterceptor
   - IORInterceptor
2 インターセプタは，インターセプタで使用できる 1 つまたは複数のインターセプトポイントを実装します。
3 インターセプタは，名前を付けることも匿名にすることもできます。同じ型のインターセプタに同じ名前を付けることはできません。ただし，匿名のインターセプタは ORB にいくつでも登録できます。

ポータブルインターセプタの登録

ポータブルインターセプタを使用するには，まず ORB に登録する必要があります。ポータブルインターセプタを登録するために，janeva.orb.init プロパティが提供されています。

```
-janeva.orb.init pi_class_name[,assembly_name]
```

次に示すように，janeva.pi.init 設定のリストを指定して，複数のポータブルインターセプタを構成できます。

```
-janeva.orb.init pi_1 -janeva.orb.init pi_2 -janeva.orb.init pi_n
```

各 janeva.orb.init インスタンスは，前のインスタンスを上書きするのではなく，ポータブルインターセプタリストにインスタンスを追加します。

VisiBroker for .NET によるポータブルインターセプタの拡張機能

POA スコープ付きサーバーリクエストインターセプタ

OMG によって指定されるポータブルインターセプタは，グローバルなスコープを持ちます。VisiBroker for .NET では，PortableInterceptorExt という新しいモジュールを追加することで，ポータブルインターセプタのパブリックな拡張機能として「POA スコープ付きサーバーリクエストインターセプタ」を定義しています。この新しいモジュールには PortableInterceptor.IORInfo から継承されたローカルインターフェース IORInfoExt があり，POA スコープ付きサーバーリクエストインターセプタをインストールするためのメソッドが追加されています。

IORInfoExt インターフェース

```
using PortableInterceptor;

namespace PortableInterceptorExt {
   public interface IORInfoExt : IORInfo {
      void AddServerRequestInterceptor(
         ServerRequestInterceptor interceptor);
      string FullPoaName();
   }
}
```

ポータブルインターセプタの実装の制限

次に，ポータブルインターセプタの実装の制限を示します。

ClientRequestInfo:

- Arguments, Result, Exceptions, Contexts, および OperationContexts は，DII 呼び出しでのみ使用できます。
- ReceivedException と ReceivedExceptionId は，アプリケーションがユーザー例外を生成すると，常に CORBA.UNKNOWN 例外とそれぞれのリポジトリ ID を返します。

ServerRequestInfo:

- Exceptions は値を返さず，動的呼び出しおよび静的スタブ呼び出しの両方で CORBA.NO_RESOURCES 例外を発生させます。
- Contexts は，呼び出し時に利用できるコンテキストのリストを返します。
- SendingException が適切なユーザー例外を返すのは，動的呼び出しの場合（ユーザー例外を Any に挿入できるか，その TypeCode 情報を利用できる場合）だけです。
- Arguments, Result, Contexts, および OperationContexts は，DSI 呼び出しでのみ使用できます。

第 11 章

ポータブルオブジェクトアダプタの使い方

ポータブルオブジェクトアダプタの概要

ポータブルオブジェクトアダプタ（POA）は，クライアントからの着信要求を受け取り，それらの要求を適切なオブジェクト実装にマップするために使用されるサービスです。J2EE 開発者であれば，呼び出しを論理的に自分自身に属する一連のオブジェクトにマップする役割を担うという点で，POA を EJB コンテナに似た機能と考えることができます。

他のコンテナと同様に，POA にも，外部的な側面と内部的な側面があると考えられます。POA の内部モデルは「サーバントオブジェクト」です。これは，ユーザーのビジネスロジックを実装するオブジェクトです。POA の外部モデルは「オブジェクトリファレンス」です。これは，分散システム呼び出しで使用できる参照です。たとえば，これらのオブジェクトリファレンスは，RMI/J2EE での `java.rmi.Remote` のインスタンス，CORBA での CORBA オブジェクトリファレンス，または .NET リモート処理での `MarshalByRefObject` のインスタンスに類似しています。POA の役割は，外部オブジェクトリファレンスと内部サーバントオブジェクトの間のマッピングです。

POA は，オブジェクトの実装と ORB を仲介します。POA は，仲介役として，要求を「サーバント」に送信します。その結果，サーバントが実行され，必要であれば子 POA が作成されます。

サーバーは複数の POA をサポートできます。少なくとも，「ルート POA」と呼ばれる POA が 1 つ存在する必要があります。ルート POA は自動的に作成されます。複数の POA は階層構造を構成し，すべての POA の上位にルート POA があります。

「サーバントマネージャ」はサーバントを探し，それを POA に対応するオブジェクトに割り当てます。サーバントに割り当てられたオブジェクトリファレンスは「アクティブオブジェクト」と呼ばれます。また，そのサーバントは，アクティブオブジェクトを「具現化する」と言います。各 POA には「アクティブオブジェクトマップ」が 1 つあり，そこで，アクティブオブジェクトのオブジェクト ID と，それに関連付けられているアクティブなサーバントを管理します。

POA の用語

次の表に，この章を読み進める上で必要な用語の定義を示します。

表 11.1 ポータブルオブジェクトアダプタ（POA）の用語

用語	説明
アクティブオブジェクトマップ	オブジェクト ID を介してアクティブオブジェクトリファレンスをサーバントにマップするテーブルです。アクティブオブジェクトマップは，各 POA に 1 つあります。
アダプタアクティベータ	存在しない子 POA への要求を受け取ったとき，オンデマンドでオブジェクトを作成できるオブジェクトです。
霊化	サーバントとオブジェクトリファレンスの関連付けを削除することです。
具現化	サーバントをオブジェクトリファレンスに関連付けることです。
ObjectID	オブジェクトアダプタ内でオブジェクトリファレンスを識別する手段です。オブジェクト ID は，オブジェクトアダプタまたはアプリケーションによって割り当てられ，作成されたオブジェクトアダプタ内でのみ一意です。サーバントは, オブジェクト ID を介してオブジェクトリファレンスに関連付けられます。
永続的オブジェクト	作成元のサーバープロセスの外部でも有効なオブジェクトリファレンスです。
POA マネージャ	POA の状態を制御するオブジェクトです。たとえば，POA が，着信した要求を受信するか，それとも破棄するかを制御します。
ポリシー	関連する POA，およびその POA が管理するオブジェクトの動作を制御するオブジェクトです。
ルート POA	各 ORB は，ルート POA と呼ばれる POA を使用して作成されます。必要であれば，ルート POA から追加の POA を作成できます。
サーバント	オブジェクトリファレンスのメソッドを実装するコードです。オブジェクトリファレンス自体ではありません。
サーバントマネージャ	サーバントとオブジェクトの関連付けを管理したり，オブジェクトの存在を確認する役割を持つオブジェクトです。複数のサーバントマネージャが存在できます。
一時的オブジェクト	作成元のプロセスの内部でのみ有効なオブジェクトリファレンスです。

POA の作成と使用の手順

手順の細部は異なりますが，次に，POA の存続期間中に行う基本的な手順を示します。

1. POA ポリシーを定義します。
2. POA を作成します。
3. POA マネージャを使用して POA をアクティブ化します。
4. サーバントを作成し，アクティブ化します。
5. サーバントマネージャを作成し，使用します。
6. アダプタアクティベータを使用します。

必要であれば，これらの手順の一部は省略できます。たとえば，POA で要求を処理する場合は，POA をアクティブ化するだけで済みます。

POA ポリシー

各 POA は，一連のポリシーによって特性が定義されます。新しい POA を作成する際は，デフォルトのポリシーセットを使用することも，必要であれば別の値を使用することもできます。ポリシーは，POA の作成時にのみ設定できます。既存の POA のポリシーは変更できません。POA は，親 POA のポリシーを継承しません。

以下に，POA ポリシー，その値，およびルート POA が使用するデフォルト値を示します。

スレッドポリシー

スレッドポリシーは，POA によって使用されるスレッドモデルを指定します。次の表で，スレッドポリシーの有効な値について説明します。

Value	説明
ORB_CTRL_MODEL	デフォルトです。デフォルトの POA スレッドモデルはマルチスレッドです。つまり，同時に複数の呼び出しが複数のスレッドにディスパッチされます。したがって，サーバント実装はスレッドセーフである必要があります。サーバントがスレッドセーフでない場合は，適切なロック機能を使用してサーバントをスレッドセーフにするか，別のスレッドポリシーを使用する必要があります。
SINGLE_THREAD_MODEL	POA は要求を順番に処理します。マルチスレッド環境では，POA からサーバントやサーバントマネージャへの呼び出しは，すべてスレッドセーフです。
MAIN_THREAD_MODEL	呼び出しは，特定の「メイン」スレッドで処理されます。すべてのメインスレッド POA に対する要求は，順番に処理されます。マルチスレッド環境では，このポリシーを持つ POA によって処理される呼び出しは，すべてスレッドセーフです。アプリケーションプログラマは，ORB.Run() または ORB.PerformWork() を呼び出して，メインスレッドを指定します。これらのメソッドについては，91 ページの「オブジェクトのアクティブ化」を参照してください。

存続期間ポリシー

存続期間ポリシーは，POA に実装されるオブジェクトの存続期間を指定します。次の表に，存続期間ポリシーの有効な値をリストします。

Value	説明
TRANSIENT	デフォルトです。POA によってアクティブ化された一時的オブジェクトは，作成元の POA より長く存続することはできません。POA が非アクティブ化された後で，その POA によって生成されたオブジェクトリファレンスを使用しようとすると，OBJECT_NOT_EXIST 例外が発生します。
PERSISTENT	POA によってアクティブ化された永続的オブジェクトは，最初に作成されたプロセスより長く存続できます。永続的オブジェクトに対して要求が呼び出されると，プロセス，POA，およびそのオブジェクトを実装するサーバントが暗黙的にアクティブ化されます。

オブジェクト ID の一意性ポリシー

オブジェクト ID の一意性ポリシーを使用すると，複数のオブジェクトリファレンスが同じサーバントを共有できます。次の表に，オブジェクト ID の一意性ポリシーの有効な値をリストします。

Value	説明
UNIQUE_ID	デフォルトです。アクティブ化されたサーバントは，1 つのオブジェクト ID だけをサポートします。
MULTIPLE_ID	アクティブ化されたサーバントは，1 つ以上のオブジェクト ID を持つことができます。オブジェクト ID は，実行時に呼び出されるメソッド内で決定する必要があります。

ID の割り当てポリシー

ID の割り当てポリシーは，オブジェクト ID をサーバーアプリケーションと POA のどちらで生成するかを指定します。次の表に，ID の割り当てポリシーの有効な値をリストします。

Value	説明
USER_ID	オブジェクトは，アプリケーションからオブジェクト ID を割り当てられます。
SYSTEM_ID	デフォルトです。オブジェクトは，POA からオブジェクト ID を割り当てられます。同時に PERSISTENT ポリシーも設定されている場合，オブジェクト ID は，同じ POA のすべてのインスタンス化にわたって一意である必要があります。

通常，USER_ID は永続的オブジェクトで使用され，SYSTEM_ID は一時的オブジェクトで使用されます。永続的オブジェクトで SYSTEM_ID を使用する場合は，サーバントまたはオブジェクトリファレンスからオブジェクトを抽出できます。

サーバント管理ポリシー

サーバント管理ポリシーは，POA がアクティブなサーバントをアクティブオブジェクトマップで管理するかどうかを指定します。次の表に，サーバント管理ポリシーの有効な値をリストします。

Value	説明
RETAIN	デフォルトです。POA は，オブジェクトリファレンスのアクティブ化をアクティブオブジェクトマップで追跡します。RETAIN は，通常，サーバントアクティベータまたは POA の明示的なアクティブ化メソッドとともに使用されます。
NON_RETAIN	POA は，アクティブなサーバントをアクティブオブジェクトマップで管理しません。NON_RETAIN は，サーバントロケータとともに使用する必要があります。

サーバントアクティベータとサーバントロケータは，サーバントマネージャの一種です。サーバントマネージャの詳細は，94 ページの「サーバントとサーバントマネージャの使用」を参照してください。

要求処理ポリシー

要求処理ポリシーは，POA が要求を処理する方法を指定します。次の表に，要求処理ポリシーの有効な値をリストします。

Value	説明
USE_ACTIVE_OBJECT_MAP_ONLY	デフォルトです。アクティブオブジェクトマップのリストにオブジェクト ID がない場合は，OBJECT_NOT_EXIST 例外が返されます。この値は，RETAIN ポリシーと組み合わせて使用する必要があります。
USE_DEFAULT_SERVANT	アクティブオブジェクトマップのリストにオブジェクト ID がない場合，または NON_RETAIN ポリシーが設定されている場合は，デフォルトのサーバントに要求がディスパッチされます。デフォルトのサーバントが登録されていない場合は，OBJ_ADAPTER 例外が返されます。この値は，MULTIPLE_ID ポリシーと組み合わせて使用する必要があります。
USE_SERVANT_MANAGER	アクティブオブジェクトマップのリストにオブジェクト ID がない場合，または NON_RETAIN ポリシーが設定されている場合は，サーバントマネージャを使用してサーバントを取得します。

暗黙的アクティブ化ポリシー

暗黙的アクティブ化ポリシーは, POA がサーバントの暗黙的アクティブ化をサポートするかどうかを指定します。次の表に，暗黙的アクティブ化ポリシーの有効な値をリストします。

Value	説明
IMPLICIT_ACTIVATION	POA は, サーバントの暗黙的なアクティブ化をサポートします。サーバントをアクティブ化するには，次の 2 とおりの方法があります。 • PortableServer.POA.ServantToReference() を使用して，サーバントをオブジェクトリファレンスに変換する。 • サーバントの This_() を呼び出す。 この値は, SYSTEM_ID ポリシーおよび RETAIN ポリシーと組み合わせて使用する必要があります。
NO_IMPLICIT_ACTIVATION	デフォルトです。POA は, サーバントの暗黙的なアクティブ化をサポートしません。

バインドサポートポリシー

バインドサポートポリシー（VisiBroker 固有のポリシー）は，VisiBroker Smart Agent（osagent）への POA とアクティブオブジェクトの登録を制御します。数千ものオブジェクトがある場合, それらをすべて osagent に登録することは困難です。このような場合は, かわりに POA を osagent に登録できます。クライアントが要求を行う際，osagent が要求を正しく転送できるように，POA の名前とオブジェクト ID がバインド要求に入れられます。次の表に，バインドサポートポリシーの有効な値をリストします。

Value	説明
BY_INSTANCE	すべてのアクティブオブジェクトが osagent に登録されます。この値は, PERSISTENT ポリシーおよび RETAIN ポリシーと組み合わせて使用する必要があります。
BY_POA	デフォルトです。POA だけが osagent に登録されます。この値は, PERSISTENT ポリシーと組み合わせて使用する必要があります。
NONE	POA とアクティブオブジェクトのどちらも osagent に登録されません。

メモ ルート POA は，NONE アクティブ化ポリシーを使用して作成されます。

POA の作成

POA を使用してオブジェクトを実装するには, サーバー上に POA オブジェクトが少なくとも 1 つ存在する必要があります。POA が確実に存在するように，ORB の初期化中にルート POA が提供されます。この POA は, 前に説明したデフォルトの POA ポリシーを使用します。

ルート POA を取得したら，サーバー側の特定のポリシーセットを実装した子 POA を作成できます。

POA の命名規則

各 POA は，その名前と完全な POA 名（すべての階層を明示したパス名）を認識しています。階層は，スラッシュ（/）で表されます。たとえば，/A/B/C は，POA C が POA B の子 POA であり，POA B が POA A の子 POA であることを意味します。最初のスラッシュは，ルート POA を表します。POA C にバインドサポートの BY_POA ポリシーが設定されている場合は，/A/B/C が登録され，クライアントは /A/B/C にバインドします。

POA 名にエスケープ文字やデリミタが含まれている場合，これらの文字を内部的に記録するとき，VisiBroker for .NET はその直前に 2 つのバックスラッシュ（\\）を付けます。

ルート POA の取得

次のサンプルコードに，サーバーアプリケーションがルート POA を取得する手続きを示します。

```
// ORB を初期化します。
CORBA.ORB orb = CORBA.ORB.Init(args);

// ルート POA へのリファレンスを取得します。
PortableServer.POA rootPOA =
      POAHelper.Narrow(orb.ResolveInitialReferences("RootPOA"));
```

メモ ResolveInitialReferences メソッドは，CORBA.Object 型の値を返します。返されるオブジェクトリファレンスは，目的の型にナローイングする必要があります。この例では，PortableServer.POA に narrow（キャスト）しています。

POA ポリシーの設定

ポリシーは，親 POA から継承されません。POA に固有の特性が必要な場合は，デフォルト値とは異なるポリシーをすべて指定する必要があります。POA ポリシーの詳細は，86 ページの「POA ポリシー」を参照してください。

```
CORBA.Policy[] policies = {
   rootPOA.CreateLifespanPolicy(LifespanPolicyValue.PERSISTENT),
   rootPOA.CreateRequestProcessingPolicy(
      RequestProcessingPolicyValue.USE_DEFAULT_SERVANT),
   rootPOA.CreateIdUniquenessPolicy(IdUniquenessPolicyValue.MULTIPLE_ID)
};
```

POA の作成およびアクティブ化

POA は，親 POA の CreatePOA を使用して作成されます。POA には任意の名前を付けることができますが，同じ親を持つ POA に同じ名前を付けることはできません。2 つの POA に同じ名前を付けようとすると，CORBA 例外（AdapterAlreadyExists）が発生します。

新しい POA を作成するには，次のように CreatePOA を使用します。

```
CreatePOA("ThePOAName", thePOAManager, thePolicyList);
```

POA マネージャ（<POAManager>）は，POA の状態（要求の処理中であるかどうかなど）を制御します。POA マネージャの名前として null が CreatePOA に渡されると，新しい POA マネージャオブジェクトが作成され，POA に関連付けられます。通常は，すべての POA に同じ POA マネージャを関連付けます。POA マネージャの詳細は，98 ページの「POA マネージャを使った POA の管理」を参照してください。

POA マネージャと POA は，作成されても自動的にはアクティブ化されません。POA に関連付けられた POA マネージャをアクティブ化するには，Activate() を使用します。次のサンプルコードは，POA を作成し，POA マネージャをアクティブ化する例です。

```
// 永続的 POA のポリシーを作成します。
CORBA.Policy[] policies = {
   rootPOA.CreateLifespanPolicy(LifespanPolicyValue.PERSISTENT)
};

// 適切なポリシーで myPOA を作成します。
PortableServer.POA myPOA =
   rootPOA.CreatePOA("bank_agent_poa",
      rootPOA.ThePOAManager,
      policies);
```

```
// POA マネージャをアクティブ化します。
rootPOA.ThePOAManager.Activate();
```

オブジェクトのアクティブ化

オブジェクトリファレンスがアクティブなサーバントに関連付けられ, POA のサーバント管理ポリシーが RETAIN である場合は, 関連付けられているオブジェクト ID がアクティブオブジェクトマップに記録され, オブジェクトがアクティブ化されます。アクティブ化は, 次のいずれかの方法で行われます。

- **明示的なアクティブ化**。ActivateObject または ActivateObjectWithId を呼び出すことで, サーバーアプリケーション自身が明示的にオブジェクトをアクティブ化します。
- **オンデマンドのアクティブ化**。ユーザーが提供するサーバーマネージャを介してオブジェクトをアクティブ化するように, サーバーアプリケーションが POA に指示します。最初に, SetServantManager で POA にサーバントマネージャを登録する必要があります。
- **暗黙的なアクティブ化**。サーバーは, なんらかの操作への応答としてのみ, オブジェクトをアクティブ化します。サーバントがアクティブでない場合, クライアントがこれをアクティブ化する手段はありません。たとえば, アクティブでないオブジェクトを要求しても, それをアクティブ化することはできません。
- **デフォルトサーバント**。POA が単一のサーバントを使用してすべてのオブジェクトを実装します。

オブジェクトの明示的なアクティブ化

POA で IdAssignmentPolicy.SYSTEM_ID を設定すると, オブジェクト ID を指定しなくても, オブジェクトを明示的にアクティブ化できます。サーバーが POA の ActivateObject を呼び出すと, オブジェクトがアクティブ化され, オブジェクト ID が割り当てられて返されます。このタイプのアクティブ化は, 一時的オブジェクトで最もよく使用されます。オブジェクトとサーバントのどちらも長期間必要になることがないため, サーバントマネージャは不要です。

オブジェクトは, オブジェクト ID を使用して明示的にアクティブ化することもできます。一般的な例として, サーバーの初期化中に ActivateObjectWithId を呼び出して, そのサーバーによって管理されるすべてのオブジェクトをアクティブ化する場合があります。すべてのオブジェクトがアクティブ化されるため, サーバントマネージャは不要です。存在しないオブジェクトに対する要求を受け取ると, OBJECT_NOT_EXIST 例外が発生します。サーバーが大量のオブジェクトを管理している場合, この例外は明らかに悪影響を及ぼします。

次のサンプルコードは, ActivateObjectWithId を使用した明示的アクティブ化の例です。

```
// 口座マネージャサーバントを作成します。
Servant managerServant = new AccountManagerImpl(rootPoa);

// 新しく作成したサーバントをアクティブ化します。
byte[] managerId = orb.StringToObjectId("BankManager");
testPoa.ActivateObjectWithId(managerId, managerServant);

// POA をアクティブ化します。
testPoa.ThePOAManager.Activate();
```

オブジェクトのオンデマンドのアクティブ化

オンデマンドのアクティブ化は, サーバントが関連付けられていないオブジェクトをクライアントが要求した場合に行われます。POA は, 要求を受け取ると, そのオブジェクト ID に関連付けられているアクティブなサーバントをアクティブオブジェクトマップから探します。該当するサーバントが見つからない場合, POA は, サーバントマネージャの

Incarnate を呼び出して，サーバントマネージャにオブジェクト ID 値を渡します。サーバントマネージャは，次の 3 つのいずれかの処理を行います。

- 要求に対して適切な操作を実行するサーバントを探します。
- OBJECT_NOT_EXIST 例外を生成します。これはクライアントに返されます。
- 要求を別のオブジェクトに転送します。

POA ポリシーによっては，その他の処理も行われます。 たとえば，RequestProcessingPolicy.USE_SERVANT_MANAGER と ServantRetentionPolicy.RETAIN が有効な場合は，サーバントとオブジェクト ID の関連付けによってアクティブオブジェクトマップが更新されます。

オブジェクトの暗黙的なアクティブ化

POA が ImplicitActivationPolicy.IMPLICIT_ACTIVATION，IdAssignmentPolicy.SYSTEM_ID，および ServantRetentionPolicy.RETAIN を使用して作成されている場合は，特定の操作によってサーバントを暗黙的にアクティブ化できます。暗黙的なアクティブ化は，次のメソッドで行われます。

- POA.ServantToReference メソッド
- POA.ServantToId メソッド
- This_() サーバントメソッド

POA に IdUniquenessPolicy.UNIQUE_ID が設定されている場合は，アクティブでないサーバントで上記のいずれかのオペレーションが実行されると，暗黙的なアクティブ化が行われます。

POA に IdUniquenessPolicy.MULTIPLE_ID が設定されている場合は，サーバントがアクティブになっていても，ServantToReference と ServantToId オペレーションは常に暗黙的アクティブ化を実行します。

デフォルトサーバントによるアクティブ化

RequestProcessing.USE_DEFAULT_SERVANT ポリシーを使用すると，オブジェクト ID に関係なく，POA は常に同じサーバントを呼び出すようになります。これは，各オブジェクトに関連付けられているデータがほとんどない場合に便利です。

次に，同じサーバントですべてのオブジェクトをアクティブ化する例を示します。

```
using System;
using System.IO;
using PortableServer;
using CORBA;

public class Server {
   static void Main(string [] args) {
      try {
         // 初期化，ORB の
         ORB orb = ORB.Init(args);

         // ルート POA へのリファレンスを取得します。
         POA rootPOA =
            POAHelper.Narrow(orb.ResolveInitialReferences("RootPOA"));

         // 永続的 POA のポリシーを作成します。
         Policy[] policies = {
            rootPOA.CreateLifespanPolicy(
               LifespanPolicyValue.PERSISTENT),
            rootPOA.CreateRequestProcessingPolicy(
               RequestProcessingPolicyValue.USE_DEFAULT_SERVANT),
            rootPOA.CreateIdUniquenessPolicy(
               IdUniquenessPolicyValue.MULTIPLE_ID)
```

```
            };

            // 適切なポリシーで myPOA を作成します。
            POA myPOA = rootPOA.CreatePOA("bank_default_servant_poa",
               rootPOA.ThePOAManager,
               policies );

            // サーバントを作成します。
            AccountManagerImpl managerServant = new AccountManagerImpl();
            myPOA.SetServant(managerServant);

            // POA マネージャをアクティブ化します。
            rootPOA.ThePOAManager.Activate();

            // 参照を生成し，それを書き出します。参照は，当座預金口座と
            // 普通預金口座のそれぞれに 1 つずつあります。ここでは，
            // サーバントを作成しておらず，サーバントの裏付けのない
            // 参照を生成しているだけであることに注意してください。

            // 当座のオブジェクト ID を書き出します。
            try {
               CORBA.Object objref = myPOA.CreateReferenceWithId(
                  orb.StringToObjectId("CheckingAccountManager"),
                  "IDL:Bank/AccountManager:1.0");

               StreamWriter writer = new StreamWriter("cref.dat");
               writer.WriteLine(orb.ObjectToString(objref));
               writer.Close();
            }
            catch (Exception e) {
               Console.WriteLine("Error writing the IOR for
                  CheckingAccountManager to file");
               Console.WriteLine(e);
            }

            try {
               // 普通のオブジェクト ID を書き出します。
               CORBA.Object objref = myPOA.CreateReferenceWithId(
                  orb.StringToObjectId("SavingsAccountManager"),
                  "IDL:Bank/AccountManager:1.0");
               StreamWriter writer = new StreamWriter("sref.dat");
               writer.WriteLine(orb.ObjectToString(objref));
               writer.Close();

            }
            catch (Exception e) {
               Console.WriteLine("Error writing the IOR for
                  SavingsAccountManager to file");
               Console.WriteLine(e);
            }

            Console.WriteLine("DefaultServantServer is ready.");
            // 着信要求を待機します。
            orb.Run();
         }
         catch(Exception e) {
            Console.WriteLine(e);
         }
      }
   }
```

オブジェクトの非アクティブ化

POA は，アクティブオブジェクトマップからサーバントを削除できます。この処理は，たとえば，一種のガベージコレクションの過程で行われます。マップから削除されたサーバントは，非アクティブになります。オブジェクトは，DeactivateObject() を使用して非アクティブ化できます。オブジェクトを非アクティブ化しても，そのオブジェクトが永久に失われるわけではありません。後でいつでも再アクティブ化できます。

サーバントとサーバントマネージャの使用

サーバントマネージャは，サーバントを検索して返す操作と，サーバントを非アクティブ化する操作の 2 種類の操作を実行します。POA は，サーバントマネージャを使用して，アクティブでないオブジェクトに対する要求を受信したときにオブジェクトをアクティブ化できます。サーバントマネージャの使用は任意です。たとえば，起動時にサーバーがすべてのオブジェクトをロードする場合，サーバントマネージャは不要です。また，サーバントマネージャは，ForwardRequest 例外を使用して，別のオブジェクトに要求を転送するようにクライアントに指示することもできます。

サーバントは，ある実装のアクティブなインスタンスです。POA は，アクティブなサーバントとそれらのサーバントのオブジェクト ID のマップを管理します。POA は，クライアント要求を受け取ると，まずこのマップをチェックし，オブジェクト ID（クライアント要求に埋め込まれている）が記録されているかどうかを確認します。オブジェクト ID が見つかった場合，POA は，要求をサーバントに転送します。オブジェクト ID がマップに見つからなかった場合は，適切なサーバントを見つけてアクティブ化するように，サーバントマネージャに要求します。これは，あくまでも 1 つの例です。実際の処理の流れは，使用している POA ポリシーによって異なります。

サーバントマネージャには，*サーバントアクティベータ*と*サーバントロケータ*の 2 種類があります。どちらのサーバントマネージャが使用されるかは，現在設定されているポリシーによって決まります。POA ポリシーの詳細は，86 ページの「POA ポリシー」を参照してください。通常，サーバントアクティベータは永続的オブジェクトをアクティブ化し，サーバントロケータは一時的オブジェクトをアクティブ化します。

サーバントマネージャを使用するには，サーバントマネージャの種類を定義するポリシー（サーバントアクティベータの場合は ServantRetentionPolicy.RETAIN，サーバントロケータの場合は ServantRetentionPolicy.NON_RETAIN）とともに，RequestProcessingPolicy.USE_SERVANT_MANAGER を設定する必要があります。

ServantActivators

サーバントアクティベータは，ServantRetentionPolicy.RETAIN と RequestProcessingPolicy.USE_SERVANT_MANAGER の設定時に使用します。

この種類のサーバントマネージャによってアクティブ化されたサーバントは，アクティブオブジェクトマップに記録されます。

サーバントアクティベータを使用して要求を処理している間に，次の手順が実行されます。

1. クライアント要求が受信されます。クライアント要求は，POA 名やオブジェクト ID などを保持しています。
2. まず，POA はアクティブオブジェクトマップをチェックします。ここでオブジェクト ID が見つかった場合は，処理がサーバントに渡され，クライアントに応答が返されます。
3. アクティブオブジェクトマップにオブジェクト ID が見つからなかった場合，POA はサーバントマネージャの Incarnate を呼び出します。Incarnate は，オブジェクト ID，およびオブジェクトがアクティブ化される POA を渡します。
4. サーバントマネージャが適切なサーバントを探します。
5. アクティブオブジェクトマップにサーバント ID が入力され，クライアントに応答が返されます。

メモ Etherealize メソッドと Incarnate メソッドの実装コードは，ユーザーが提供します。

サーバントは，後で非アクティブ化される場合があります。これには，DeactivateObject オペレーション，POA に関連付けられている POA マネージャの非アクティブ化など，いくつかの場合が考えられます。オブジェクトの非アクティブ化の詳細は，93 ページの「オブジェクトの非アクティブ化」を参照してください。

次に，サーバントアクティベータの実装例を示します。

```
using System;
using System.Threading;
using System.Collections;

public class
   AccountManagerActivator : PortableServer.ServantActivator {

   private Hashtable _objectMap = new Hashtable();

   public AccountManagerActivator() {
      Console.WriteLine("AccountManagerActivator() called.");
      // オブジェクトマップを追加します。
      _objectMap.Add("SavingsAccountManager",
         new SavingsAccountManagerImpl());
      _objectMap.Add("CheckingAccountManager",
         new CheckingAccountManagerImpl());
   }

   public PortableServer.Servant Incarnate(byte[] oid,
      PortableServer.POA adapter) {

      try {
         Console.WriteLine(
            "AccountManagerActivator.Incarnate() called.");

         string accountType = CORBA.ORB.Init().ObjectIdToString(oid);
         Console.WriteLine("¥nAccountManagerActivator.Incarnate()
            called with ID = " + accountType);

         new ObjectDeactivator(adapter, oid);
         return (PortableServer.Servant) _objectMap[accountType];
      }
      catch (Exception e) {
         Console.WriteLine(e);
      }
      return null;
   }

   public void Etherealize(byte[] oid,
      PortableServer.POA adapter,
      PortableServer.Servant serv,
      bool cleanupInProgress,
      bool remainingActivations) {

      Console.WriteLine("Etheralize() called.");

      try {
         string accountType = CORBA.ORB.Init().ObjectIdToString(oid);
         Console.WriteLine("¥nAccountManagerActivator.Etherealize()
            called with ID = " + accountType);
      }
      catch (Exception e) {
         Console.WriteLine(e);
      }
   }

   private const int ONE_SECOND = 1000;

   private class ObjectDeactivator {
      private PortableServer.POA _adapter;
      private byte[] _oid;

      public ObjectDeactivator(PortableServer.POA adapter, byte[] oid) {
         _adapter = adapter;
         _oid = oid;
         new Thread(new ThreadStart(Deactivate)).Start();
      }

      public void Deactivate() {
         Console.WriteLine("Deactivate() called.");
         try {
            Thread.Sleep(ONE_SECOND * 15);
            Console.WriteLine("¥nDeactivating the object with ID = " +
```

```
            CORBA.ORB.Init().ObjectIdToString(_oid));
            _adapter.DeactivateObject(_oid);
         }
         catch (Exception e) {
            Console.WriteLine(e);
         }
      }
   }
}
```

次に，92 ページの「デフォルトサーバントによるアクティブ化」のサンプルコードに対応するサーバー実装を示します。この例は，サーバントアクティベータを使用してサーバントをアクティブ化する場合の相違点を示しています。

```
// 永続的 POA のポリシーを作成します。
CORBA.Policy[] policies = {
   rootPOA.CreateLifespanPolicy(
      LifespanPolicyValue.PERSISTENT),
   rootPOA.CreateRequestProcessingPolicy(
      RequestProcessingPolicyValue.USE_SERVANT_MANAGER)
};

// 適切なポリシーで myPOA を作成します。
POA myPOA =
   rootPOA.CreatePOA("bank_servant_activator_poa",
      rootPOA.ThePOAManager,
      policies );

// サーバントアクティベータサーバントを作成し，そのリファレンスを取得します。
ServantActivator sa = new AccountManagerActivator(orb);

// 独自の POA にサーバントアクティベータを設定します。
myPOA.SetServantManager(sa);

// POA マネージャをアクティブ化します。
rootPOA.ThePOAManager.Activate();
```

ServantLocators

一般に，POA のアクティブオブジェクトマップのサイズはかなり大きくなり，メモリーを消費します。メモリーの消費量を削減するには，POA の作成時に RequestProcessingPolicy.USE_SERVANT_MANAGER と ServantRetentionPolicy.NON_RETAIN を使用します。この場合，サーバントとオブジェクトの関連付けは，アクティブオブジェクトマップに保存されません。関連付けが保存されていないため，要求があるたびにサーバントロケータが呼び出されます。

サーバントロケータを使用して要求を処理している間に，次の手順が実行されます。

1. クライアント要求が受信されます。クライアント要求は，POA 名とオブジェクト ID を保持しています。
2. ServantRetentionPolicy..NON_RETAIN を使用しているので，POA は，アクティブオブジェクトマップでオブジェクト ID を検索しません。
3. POA は，サーバントマネージャの Preinvoke を呼び出します。Preinvoke は，オブジェクト ID，オブジェクトがアクティブ化される POA などのパラメータを受け渡します。
4. サーバントロケータが適切なサーバントを探します。
5. サーバントで処理が行われ，クライアントに応答が返されます。
6. POA は，サーバントマネージャの Postinvoke を呼び出します。

メモ Preinvoke メソッドと Postinvoke メソッドの実装コードは，ユーザーが提供します。

次に，サーバントロケータの実装例を示します。

```
using System;
using CORBA;
using PortableServer;
using PortableServer.ServantLocatorNS;

public class AccountManagerLocator : ServantLocator {
   private ORB _orb;

   public AccountManagerLocator(ORB orb) {
      _orb = orb;
   }

   public Servant Preinvoke (byte[] oid, POA adapter,
      string operation, out object theCookie) {

      string accountType = _orb.ObjectIdToString(oid);
      theCookie = null;

      Console.WriteLine("¥nAccountManagerLocator.preinvoke
         called with ID = {0}¥n", accountType);
      if (accountType.Equals("SavingsAccountManager")) {
         return new SavingsAccountManagerImpl();
      }

      return new CheckingAccountManagerImpl();
   }

   public void Postinvoke (byte[] oid,
      POA adapter,
      string operation,
      object theCookie,
      Servant theServant) {
      string id = _orb.ObjectIdToString(oid);
      Console.WriteLine("¥nAccountManagerLocator.postinvoke
         called with ID = {0}¥n", id);
   }
}
```

次に，92 ページの「デフォルトサーバントによるアクティブ化」のサンプルコードに対応するサーバー実装を示します。この例は，サーバントロケータを使用してサーバントをアクティブ化する場合の相違点を示しています。

```
// 独自の POA のポリシーを作成します。永続的存続期間のポリシー，
// サーバントマネージャを使用する要求処理ポリシー，および
// NON_RETAIN サーバント管理ポリシーが必要です。この NON_RETAIN ポリシーにより，
// サーバントアクティベータではなくサーバントロケータを使用できます
CORBA.Policy[] policies = {
   rootPOA.CreateLifespanPolicy(
      LifespanPolicyValue.PERSISTENT),
   rootPOA.CreateServantRetentionPolicy(
      ServantRetentionPolicyValue.NON_RETAIN),
   rootPOA.CreateRequestProcessingPolicy(
      RequestProcessingPolicyValue.USE_SERVANT_MANAGER)
};

// 適切なポリシーで myPOA を作成します。
POA myPOA = rootPOA.CreatePOA("bank_servant_locator_poa",
   rootPOA.ThePOAManager, policies);

// サーバントロケータサーバントを作成し，そのリファレンスを取得します。
ServantLocator sl = new AccountManagerLocator(orb);

// 独自の POA にサーバントアクティベータを設定します。
myPOA.SetServantManager(sl);

// POA マネージャをアクティブ化します。
rootPOA.ThePOAManager.Activate();
```

POA マネージャを使った POA の管理

POA マネージャは，POA の状態（要求をキューに入れるか，破棄するか）を制御し，POA を非アクティブ化できます。POA は，それぞれ 1 つの POA マネージャオブジェクトに関連付けられています。POA マネージャは，1 つ以上の POA を制御できます。

POA マネージャは，POA の作成時に POA に関連付けられます。使用する POA マネージャを指定するか，null を指定して新しい POA マネージャを作成できます。

次に，POA とその POA マネージャを指定する例を示します。

```
POA myPOA = rootPOA.CreatePOA("MyPOA",
   rootPOA.ThePOAManager, policies);

POA myPOA = rootPOA.CreatePOA("MyPOA", null, policies);
```

関連付けられている POA がすべて破棄されると，POA マネージャも「破棄」されます。

POA マネージャには，次の 4 種類の状態があります。

- Holding
- Active
- Discarding
- Inactive

これらの状態によって POA の状態が決まります。以下のセクションでは，これらの状態について詳しく説明します。

現在の状態の取得

POA マネージャの現在の状態を取得するには，次の構文を使用してください。

```
State state = manager.GetState();
```

停止状態

POA マネージャは，作成時にデフォルトで停止状態になります。POA マネージャが停止状態の場合，POA は着信した要求をすべてキューに入れます。

POA マネージャが停止状態の場合は，アダプタアクティベータを必要とする要求もキューに入ります。

POA マネージャを停止状態にするには，次の構文を使用します。

```
manager.HoldRequests(waitForCompletion);
```

waitForCompletion は Boolean です。FALSE の場合，このオペレーションは，状態を停止に変更し，ただちに戻ります。TRUE の場合，このオペレーションは，状態の変更前に開始したすべての要求が完了するか，POA マネージャが停止以外の状態に変化しないと戻りません。AdapterInactive は，このオペレーションを呼び出した時点で POA マネージャがすでに非アクティブ状態の場合に生成される例外です。

メモ 非アクティブ状態の POA マネージャは，停止状態に変更できません。

キューに入っており，まだ起動されていない要求は，停止状態の間，そのままキューの中に保持されます。

アクティブ状態

POA マネージャがアクティブ状態の場合，関連付けられている POA は，要求を処理します。

POA マネージャをアクティブ状態にするには，次の構文を使用します。

```
manager.Activate();
```

AdapterInactive は，このオペレーションを呼び出した時点で POA マネージャがすでに非アクティブ状態の場合に生成される例外です。

メモ　非アクティブ状態の POA マネージャは，アクティブ状態に変更できません。

破棄状態

POA マネージャが破棄状態の場合，関連付けられている POA は，まだ開始されていない要求をすべて破棄します。また，その POA に登録されているアダプタアクティベータは呼び出されません。POA が受け取る要求の数が多すぎる場合は，この状態が便利です。この場合，破棄された要求を再送信するようにクライアントに通知する必要があります。POA が受け取る要求数が多すぎるかどうかを判定するための機能は用意されていません。必要であれば，ユーザー自身でスレッドの監視機能を設定してください。

POA マネージャを破棄状態にするには，次の構文を使用します。

```
manager.DiscardRequests(waitForCompletion);
```

waitForCompletion オプションは Boolean です。FALSE の場合，このオペレーションは，状態を停止に変更し，ただちに戻ります。TRUE の場合，このオペレーションは，状態の変更前に開始したすべての要求が完了するか，POA マネージャが破棄以外の状態に変化しないと戻りません。AdapterInactive は，このオペレーションを呼び出した時点で POA マネージャがすでに非アクティブ状態の場合に生成される例外です。

メモ　非アクティブ状態の POA マネージャは，破棄状態に変更できません。

非アクティブ状態

POA マネージャが非アクティブ状態の場合，関連付けられている POA は，着信した要求を受け付けません。この状態は，関連する POA をシャットダウンするときに使用します。

メモ　非アクティブ状態の POA マネージャは，停止状態に変更できません。

POA マネージャを非アクティブ状態にするには，次の構文を使用します。

```
manager.Deactivate(etherealizeObjects, waitForCompletion);
```

状態の変化後，etherealizeObjects が TRUE の場合，ServantRetentionPolicy.RETAIN と RequestProcessingPolicy.USE_SERVANT_MANAGER が設定されているすべての関連する POA は，すべてのアクティブオブジェクトに対してサーバントマネージャの Etherealize を呼び出します。etherealizeObjects が FALSE の場合，Etherealize は呼び出されません。waitForCompletion オプションは Boolean です。FALSE の場合，このオペレーションは，状態を非アクティブに変更し，ただちに戻ります。TRUE の場合，このオペレーションは，状態の変更前に開始されたすべての要求が完了するか，関連するすべての POA（ServantRetentionPolicy.RETAIN と RequestProcessingPolicy.USE_SERVANT_MANAGER が設定されている）で Etherealize が呼び出されるまで戻りません。AdapterInactive は，このオペレーションを呼び出した時点で POA マネージャがすでに非アクティブ状態の場合に生成される例外です。

リスナーとディスパッチャ：サーバーエンジン，サーバー接続マネージャ，およびそれらのプロパティ

メモ　リスナーとディスパッチャの機能に関するポリシーは，POA ではサポートされていません。 これらの機能を提供するには，VisiBroker for .NET 固有のポリシー（ServerEnginePolicy）を使用します。

VisiBroker for .NET では，VisiBroker for .NET サーバーのエンドポイントを定義および調整するために，たいへん柔軟性のあるメカニズムが提供されています。この場合のエン

ドポイントとは，クライアントがサーバーと通信するための通信チャネルの接続先です。*サーバーエンジン*は，設定可能なプロパティのセットとして提供される接続エンドポイントのための仮想抽象コンポーネントです。

抽象サーバーエンジンは，次の項目を制御できます。

- 接続リソースの種類
- 接続管理
- スレッドモデルと要求のディスパッチ

サーバーエンジンと POA

VisiBroker for .NET の POA は，サーバーエンジンと多対多の関係を持つことができます。1 つの POA を複数のサーバーエンジンに，また 1 つのサーバーエンジンを複数の POA に関連付けることができます。そのため，POA（および POA のオブジェクトリファレンス）は，複数の通信チャネルをサポートできます。

最も単純な例は，POA がそれぞれ固有のサーバーエンジンを 1 つだけ持つ場合です。その場合，各 POA への要求は，それぞれ異なるポートで受信されます。また，1 つの POA が複数のサーバーエンジンを持つこともできます。その場合は，1 つの POA が複数の入力ポートから着信する要求をサポートします。

POA はサーバーエンジンを共有することもできます。サーバーエンジンが共有される場合は，複数の POA が同じポートを監視します。複数の POA に対する要求が同じポートに着信しても，要求に POA 名が埋め込まれているため，要求は正しくディスパッチされます。このような状況は，デフォルトのサーバーエンジンを使用し，新しいサーバーエンジンを指定しないで複数の POA を作成する場合などに発生します。

サーバーエンジンは名前によって識別され，その名前が最初に組み込まれるときに定義されます。VisiBroker for .NET では，デフォルトで次の 3 つのサーバーエンジン名が定義されています。

- `iiop_tp`：スレッドプールディスパッチャを使用した TCP トランスポート
- `iiop_ts`：セッションごとスレッドディスパッチャを使用した TCP トランスポート
- `iiop_tm`：メインスレッドディスパッチャを使用した TCP トランスポート

POA とサーバーエンジンの関連付け

POA に関連付けられているデフォルトのサーバーエンジンを変更するには，プロパティ `vbroker.se.default` を使用します。たとえば，次のように設定します。

```
vbroker.se.default=MySE
```

これは，MySE という名前の新しいサーバーエンジンを定義しています。ルート POA と，作成されたすべての子 POA は，デフォルトでこのサーバーエンジンに関連付けられます。

また，SERVER_ENGINE_POLICY_TYPE POA ポリシーを使用すると，POA を特定のサーバーエンジンに明示的に関連付けることができます。たとえば，次のようになります。

```
// ServerEngine ポリシー値を作成します。
Any seAny = orb.CreateAny();
StringSequenceHelper.Insert(seAny, new String [] {"mySE"});
Policy sePolicy = orb.CreatePolicy(
   PortableServerExt.SERVER_ENGINE_POLICY_TYPE.Value, seAny);

// POA のポリシーを作成します。
Policy [] policies = {
   rootPOA.CreateLifespanPolicy(LifespanPolicyValue.PERSISTENT),
   sePolicy
};

// ポリシー付きで POA を作成します。
POA myPOA = rootPOA.CreatePOA("bank_se_policy_poa",
   rootPOA.ThePOAManager,
   policies);
```

POA は IOR テンプレートを持ち，そのプロファイルは，POA に関連付けられているサーバーエンジンから取得されます。

サーバーエンジンポリシーを指定しないと，POA は，サーバーエンジン名が iiop_tp であるとみなして，次のデフォルト値を使用します。

```
vbroker.se.iiop_tp.host=null
vbroker.se.iiop_tp.proxyHost=null
vbroker.se.iiop_tp.scms=iiop_tp
```

デフォルトのサーバーエンジンポリシーを変更するには，vbroker.se.default プロパティを使用して新しいサーバーエンジンポリシー名を入力し，新しいサーバーエンジンのすべての要素に値を定義してください。たとえば，次のようになります。

```
vbroker.se.default=abc,def
vbroker.se.abc.host=cob
vbroker.se.abc.proxyHost=null
vbroker.se.abc.scms=cobscm1,cobscm2
vbroker.se.def.host=gob
vbroker.se.def.proxyHost=null
vbroker.se.def.scms=gobscm1
```

サーバーエンジンのエンドポイントのホストの定義

サーバーエンジンは接続のエンドポイントの定義に使用されるため，エンドポイントのホストを指定するために次のプロパティが提供されています。

- vbroker.se.<se-name>.host=<host-URL>（例：vbroker.se.mySE.host=host.borland.com）)
- vbroker.se.<se-name>.proxyHost=<proxy-host-URL-or-IP-address>（例：vbroker.se.mySE.proxyHost=proxy.borland.com）

proxyHost プロパティの値には，IP アドレスを指定することもできます。その場合は，IOR 内のデフォルトホスト名がその IP アドレスに置き換えられます。

サーバーエンジンの抽象エンドポイントは，サーバー接続マネージャ（SCM）と呼ばれる設定可能な一連のエンティティによってさらに詳細に設定できます。サーバーエンジンは，複数の SCM を持つことができます。SCM は，複数のサーバーエンジンで共有できません。SCM も名前によって識別され，サーバーエンジンに対して次のように定義されます。

```
vbroker.se.<se-name>.scms=<SCM-name>[,<SCM-name>,...]
```

サーバー接続マネージャ

サーバー接続マネージャ（SCM）は，エンドポイントの設定可能なコンポーネントを定義します。SCM は，接続リソースを管理し，要求を監視し，関連付けられている POA に要

求をディスパッチします。これらの機能を実行するため，プロパティグループを介して定義される次の 3 つの論理エンティティが SCM によって提供されます。

- マネージャ
- リスナー
- ディスパッチャ

各 SCM は，マネージャ，リスナー，ディスパッチャを 1 つずつ持ちます。この 3 つがすべて定義されている場合に，単一のエンドポイント定義が形成され，クライアントはサーバーと通信できるようになります。

マネージャ

マネージャは，接続リソースの設定可能部分を定義する一連のプロパティです。VisiBroker for .NET は，Socket 型のマネージャを提供します。

```
vbroker.se.<se-name>.scm.<scm-name>.manager.type=Local|Socket
```

サーバーのエンドポイントで受け入れることができる最大同時接続数を指定するには，connectionMax プロパティを使用します。

```
vbroker.se.<se-name>.scm.<scm-name>.manager.connectionMax=<integer>
```

connectionMax を 0 に設定すると，接続数に制限がないことを示します。これはデフォルトの設定です。

最大アイドル時間を指定するには，connectionMaxIdle プロパティを使用します。

```
vbroker.se.<se-name>.scm.<scm-name>.manager.connectionMaxIdle=<seconds>
```

connectionMaxIdle を 0 に設定すると，タイムアウトがないことを示します。これはデフォルトの設定です。

マネージャがアイドル状態の接続を回収するためのガベージコレクション時間を指定することもできます。connectionMaxIdle 時間の経過後も，ガベージコレクションによって回収されるまでの間，接続はアイドル状態のままです。garbageCollectTimer プロパティを使用して，ガベージコレクションの周期を秒単位で指定できます。

```
vbroker.se.<se-name>.scm.<scm-name>.manager.garbageCollectTimer=<seconds>
```

ガベージコレクション時間は，次のプロパティを使用して指定します。

```
vbroker.orb.gcTimeout=<seconds>
```

0 を指定すると，接続はガベージコレクションによって回収されません。

リスナー

リスナーは，SCM がメッセージを監視する方法を決定する SCM コンポーネントです。マネージャと同様に，リスナーも一連のプロパティで構成されます。VisiBroker for .NET では，TCP 接続に対して IIOP リスナーが定義されています。

リスナーは，実際の基礎トランスポートメカニズムに密接しているため，異なるリスナータイプ間ではリスナーのプロパティに可搬性がありません。次に定義されるように，各リスナータイプが独自のプロパティセットを持ちます。

IIOP リスナーのプロパティ

IIOP リスナーでは，ホストと組み合わせて，ポートと（必要であれば）プロキシポートを定義する必要があります。これらは，port プロパティと proxyPort プロパティを使用して，次のように設定されます。

```
vbroker.se.<se-name>.scm.<scm-name>.listener.port=<port>
vbroker.se.<se-name>.scm.<scm-name>.listener.proxyPort=<proxy-port>
```

メモ port プロパティを設定しない場合，または 0 に設定した場合は，ポートが無作為に選択されます。proxyPort プロパティの値を 0 にすると，listener.port プロパティによって定義されるか，システムによって無作為に選択された実際のポートが IOR に含まれます。実際のポートを宣言する必要がない場合は，プロキシポートを正数（0 以外）に設定してください。

送信／受信のバッファサイズ，ソケット遅延時間，アクティブでないソケットを存続させるかどうかなど，標準の TCP ソケットオプションを定義するためのプロパティ設定もサポートされています。これらのメカニズムのために，次のプロパティが提供されています。

```
vbroker.se.<se-name>.scm.<scm-name>.listener.rcvBuffSize=<bytes>
vbroker.se.<se-name>.scm.<scm-name>.listener.sendBuffSize=<bytes>
vbroker.se.<se-name>.scm.<scm-name>.listener.socketLinger=<seconds>
vbroker.se.<se-name>.scm.<scm-name>.listener.keepAlive=true|false
```

なんらかの理由で，TCP ソケットのプロパティとして単にシステムのデフォルト値を使用する場合は，該当するプロパティの値を 0 に設定します。

また，VisiBroker for .NET では，GIOP のバージョンを指定するためのプロパティもサポートされています。

```
vbroker.se.<se-name>.scm.<scm-name>.listener.giopVersion=<version>
```

ディスパッチャ

ディスパッチャは，SCM がスレッドに要求をディスパッチする方法を決定する一連のプロパティを定義します。ThreadPool，ThreadSession，MainThread という 3 つのタイプのディスパッチャが提供されています。ディスパッチャタイプは，次のように type プロパティを使用して設定します。

```
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.type=ThreadPool|ThreadSession|MainThread
```

ディスパッチャタイプが ThreadPool の場合は，SCM を介してさらに詳細な制御が提供されます。ThreadPool は，スレッドプール内に作成できる最小スレッド数と最大スレッド数，およびアイドル状態のスレッドが破棄されるまでの最大時間（秒）を定義します。これらの値は，次のプロパティで制御されます。

```
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.threadMin=<integer>
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.threadMax=<integer>
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.threadMaxIdle=<seconds>
```

プロパティを使用するタイミング

サーバーエンジンのプロパティの一部を変更する必要があることはよくあります。これらのプロパティを変更する方法は，目的に応じて異なります。たとえば，ポート番号を変更する場合は，次の方法があります。

- デフォルトの listener.port プロパティを変更する。
- 新しいサーバーエンジンを作成する。

デフォルトの listener.port プロパティの変更が最も簡単ですが，デフォルトサーバーエンジンを使用するすべての POA に影響が出ます。そして，それで問題がない場合と，問題になる場合があります。

特定の POA でポート番号を変更する場合は，新しいサーバーエンジンを作成し，そのプロパティを定義した後で，POA 作成時にそのサーバーエンジンを参照する必要があります。

前のセクションでは，サーバーエンジンのプロパティを更新する方法を示しました。次のコードでは，ユーザー定義のサーバーエンジンポリシーを使用して POA を作成する方法を示します。

```
using System;
using System.IO;
using PortableServer;
using CORBA;

public class Server {
   static void Main(string [] args) {
      try {
         // 初期化，ORB の
         ORB orb = ORB.Init(args);

         // ルート POA へのリファレンスを取得します。
         POA rootPOA =
            POAHelper.Narrow(orb.ResolveInitialReferences("RootPOA"));

         // 独自のサーバーエンジンポリシーを作成します。
         Any seAny = orb.CreateAny();
         StringSequenceHelper.Insert(seAny, new String [] {"mySe"});
         Policy sePolicy = orb.CreatePolicy(
            PortableServerExt.SERVER_ENGINE_POLICY_TYPE.Value, seAny);

         // 永続的 POA のポリシーを作成します。
         Policy [] policies = {
            rootPOA.CreateLifespanPolicy(
               LifespanPolicyValue.PERSISTENT), sePolicy
         };

         // 適切なポリシーで myPOA を作成します。
         POA myPOA = rootPOA.CreatePOA("bank_se_policy_poa",
            rootPOA.ThePOAManager, policies);

         // サーバントを作成します。
         AccountManagerImpl managerServant = new AccountManagerImpl();

         // サーバントの ID を決定します。
         byte [] managerId = orb.StringToObjectId("BankManager");

         // サーバントをアクティブ化します。
         myPOA.ActivateObjectWithId(managerId, managerServant);

         // リファレンスを取得します。
         CORBA.Object objRef = myPOA.ServantToReference(managerServant);

         // IOR を書き出します。
         try {
            StreamWriter writer = new StreamWriter("ior.dat");
            writer.WriteLine(orb.ObjectToString(objRef));
            writer.Close();
         }
         catch (Exception e) {
            Console.WriteLine("Error writing the IOR to file ior.dat");
            Console.WriteLine(e);
         }

         // POA マネージャをアクティブ化します。
         rootPOA.ThePOAManager.Activate();

         Console.WriteLine("{0} is ready.", objRef);

         // 着信要求を待機します。
         orb.Run();
      }
      catch(Exception e) {
         Console.WriteLine(e);
```

```
        }
        Console.ReadLine();
    }
}
```

アダプタアクティベータ

アダプタアクティベータは，POA に関連付けられており，オンデマンドで子 POA を作成する機能を提供します。これは，FindPOA オペレーションの間か，特定の子 POA を指定する要求が受信されたときに行われます。

アダプタアクティベータは，子 POA（またはその 1 つ）を指定する要求を受信したときの処理過程で，またはアクティブ化パラメータ値が TRUE の状態で FindPOA が呼び出されたときに，子 POA をオンデマンドで作成する機能を POA に提供します。実行の開始時に，必要な POA をすべて作成するサーバーでは，アダプタアクティベータの使用や提供は不要です。アダプタアクティベータは，要求処理中に POA を作成する必要がある場合にのみ必要です。

POA からアダプタアクティベータへの要求が処理されている間，新しい POA（または子以下の POA）によって管理されるオブジェクトに対する要求は，すべてキューに入れられます。このシリアル化により，新しい POA に要求が配信される前に，アダプタアクティベータは POA の初期化を完了できます。

要求の処理

要求は，ターゲットオブジェクトのオブジェクト ID，およびそのオブジェクトリファレンスを作成した POA を保持します。クライアントが要求を送信すると，まず ORB が適切なサーバーを探すか，必要であればサーバーを起動します。次に，ORB は，そのサーバー上で適切な POA を探します。

適切な POA が見つかると，ORB はその POA に要求を配信します。その時点で要求がどのように処理されるかは，POA のポリシーとオブジェクトのアクティブ化状態によって決まります。オブジェクトのアクティブ化状態の詳細は，91 ページの「オブジェクトのアクティブ化」を参照してください。

- POA が ServantRetentionPolicy.RETAIN ポリシーを持つ場合，POA はアクティブオブジェクトマップを参照して，要求にあるオブジェクト ID に関連付けられたサーバントを探します。サーバントが見つかった場合は，そのサーバントの適切なメソッドを呼び出します。
- POA が ServantRetentionPolicy..NON_RETAIN または ServantRetentionPolicy..RETAIN を持ち，適切なサーバントが見つからなかった場合は，次の処理に続きます。
- POA が RequestProcessingPolicy..USE_DEFAULT_SERVANT を持つ場合は，デフォルトサーバントの適切なメソッドを呼び出します。
- POA が RequestProcessingPolicy.USE_SERVANT_MANAGER を持つ場合は，サーバントマネージャの Incarnate または Preinvoke を呼び出します。
- POA が RequestProcessingPolicy.USE_OBJECT_MAP_ONLY を持つ場合は，例外が生成されます。

サーバントマネージャを起動しても，オブジェクトを具現化できない場合は，サーバントマネージャが ForwardRequest 例外を生成することがあります。

第 12 章

トランザクションサービスの使い方

この章では，VisiBroker for .NET でトランザクションを使用する方法について説明します。各 API の詳細は，Borland VisiBroker for .NET API ドキュメントを参照してください。

トランザクション対応の VisiBroker for .NET の設定

トランザクションを使用して実行するには，次の手順にしたがいます。

1. サービス DLL への参照をアプリケーションに追加します。これは，この DLL で定義されている CosTransations 名前空間にアクセスするためにも必要です。
2. アプリケーションを実行する際に，janeva.transactions プロパティを true に設定します。

VisiBroker for .NET 管理のトランザクションの作成

VisiBroker for .NET 管理のトランザクションでは，Current インターフェースを使用してすべてのトランザクションを管理します。Current を使用してトランザクションを開始したり，Current を使用してトランザクションを暗黙的に伝搬します。つまり，トランザクションを開始する際は，常に Current.Begin() を使用します。

Current は，プロセス全体に対して有効なオブジェクトとして，各スレッドのトランザクションコンテキストの関連付けを管理します。各スレッドは，1 つのトランザクションコンテキストと個別に関連付けられています。

VisiBroker for .NET 管理のトランザクションでは，トランザクションサービスがトランザクションコンテキストを各参加者に透過的に転送するため，トランザクションの参加者が同じトランザクションコンテキストを共有します。つまり，オリジネータが他のオブジェクトにアクションの実行を要求し，結果として他のオブジェクトが呼び出されても，トランザクションの状態は維持されます。

Current オブジェクトリファレンスの取得

VisiBroker for .NET 管理のトランザクションにアクセスするには,Current へのオブジェクトリファレンスを取得する必要があります。Current オブジェクトリファレンスは,プロセスを通じて有効です。次に，Current オブジェクトへの参照を取得するための一般的な手順とサンプルコードを示します。

1. orb.ResolveInitialReferences() メソッドを呼び出します。このメソッドは,Current オブジェクトへの参照を取得します。
2. 返されたオブジェクトを CosTransactions.Current オブジェクトに narrow（キャスト）します。

たとえば，次のようになります。

```
CORBA.ORB orb = ...;
CosTransactions.Current current = CosTransactions.CurrentHelper.Narrow(
   orb.ResolveInitialReferences("TransactionCurrent"));
```

CosTransactions.Current を narrow(キャスト)する場合は,CosTransactions モジュールから提供される元の一連のメソッドを使用するように指定します。

CosTransactions モジュールの概要

CosTransactions モジュールは,最終的な OMG Transaction Service ドキュメントに準拠するトランザクションサービス IDL です。このモジュールにより，CORBA 準拠のメソッドを使用するように厳しく制限されます。 このモジュールの IDL は，ファイル CosTransactions.idl に含まれています。

トランザクションサービスのクラスとインターフェース

Current インターフェース

Current インターフェースは，次のためのメソッドを定義します。

- プログラムでトランザクションを管理する
- 暗黙的にトランザクションを伝搬する
- 現在のトランザクションに関する情報を取得する
- リソースと同期オブジェクトを登録する

Current のメソッド

以下のセクションでは，Current の重要なメソッドについて説明します。詳細は，Borland VisiBroker for .NET API ドキュメントを参照してください。

Begin

このメソッドは,新しいトランザクションを作成します。ネストしたトランザクションはサポートされていないため，これが常に最上位のトランザクションになります。

クライアントスレッドが新しいトランザクションに関連付けられるように，スレッドのトランザクションコンテキストが変更されます。クライアントスレッドがすでにトランザクションに関連付けられている場合は,
例外 CosTransactions.SubtransactionsUnavailable が発行されます。

Commit

このメソッドは，クライアントスレッドに関連付けられているトランザクションをコミットします。このメソッドの効果は，対応する Terminator オブジェクトの Commit メソッドを呼び出した場合と同じです。

このトランザクションにロールバックのマークが付けられているか，ロールバックを提案する Resource がある場合は，このメソッドを呼び出すと，CORBA.TRANSACTION_ROLLEDBACK が生成されます。
現在のトランザクションがない場合は，例外 CosTransactions.NoTransaction が生成されます。 呼び出し元がトランザクションオリジネータでない場合，Commit は例外 CORBA.NO_PERMISSION を生成します。

checked behavior のためのチェックが行われます。

このメソッドから戻った時点で，クライアントスレッドとトランザクションの関連付けが解除されます。トランザクションが存在するかのように Current を使用しようとすると，NoTransaction または CORBA.TRANSACTION_REQUIRED などの例外が生成されるか，null オブジェクトリファレンスが返されます。

トランザクションが完了し，関連するすべての同期オブジェクトに通知されるまで，このメソッドは戻りません。

GetControl

このメソッドは Control オブジェクトリファレンスを返します。これは，現在クライアントスレッドに関連付けられているトランザクションコンテキストを表します。

クライアント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，null オブジェクトリファレンスが返されます。

GetStatus

このメソッドは列挙値（enum Status）を返します。これは，クライアントスレッドに関連付けられているトランザクションの状態を表します。

このメソッドを呼び出すことは，対応する Coordinator オブジェクトの GetStatus メソッドを呼び出すことと同じです。現在のスレッドに関連付けられているトランザクションがない場合，このメソッドは CosTransactions.StatusNoTransaction を返します。

戻り値は次のとおりです。

戻り値	説明
StatusActive	ターゲットオブジェクトに関連付けられているトランザクションの状態がアクティブです。トランザクションにロールバックまたはタイムアウトのマークが付けられていない限り，トランザクションが開始された後で，Coordinator が Prepare 文が発行する前に，トランザクションサービスはこの状態を返します。
StatusMarkedRollback	トランザクションはターゲットオブジェクトに関連付けられており，ロールバックのマークが付けられています。これは，通常，RollbackOnly メソッドの結果です。
StatusPrepared	トランザクションはターゲットオブジェクトに関連付けられており，準備が完了しています。
StatusCommitted	トランザクションはターゲットオブジェクトに関連付けられており，コミットされています。通常は，経験則が存在します。そうでない場合は，トランザクションがすぐに破棄され，StatusNoTransaction が返されています。
StatusRolledBack	トランザクションはターゲットオブジェクトに関連付けられており，その結果がロールバックと決定されています。通常は，経験則が存在します。そうでない場合は，トランザクションがすぐに破棄され，StatusNoTransaction が返されています。
StatusUnknown	トランザクションはターゲットオブジェクトに関連付けられていますが，トランザクションサービスは現在の状態を特定できません。これは一時的な状態です。次の呼び出しでは，異なる状態が返されます。

戻り値	説明
StatusNoTransaction	現在，ターゲットオブジェクトに関連付けられたトランザクションはありません。トランザクションが完了すると，この値が返されます。
StatusPreparing	ターゲットオブジェクトに関連付けられているトランザクションが準備中です。トランザクションの準備が開始され，そのプロセスが完了していない場合，トランザクションサービスはこの状態を返します。通常は，1 つ以上の Resources から準備のための応答を待機している状態です。
StatusCommitting	ターゲットオブジェクトに関連付けられているトランザクションがコミット中です。トランザクションのコミットが開始され，そのプロセスが完了していない場合，トランザクションサービスはこの状態を返します。通常は，1 つ以上の Resources からの応答を待機している状態です。
StatusRollingBack	ターゲットオブジェクトに関連付けられているトランザクションがロールバック中です。トランザクションのロールバックが開始され，そのプロセスが完了していない場合，トランザクションサービスはこの状態を返します。通常は，1 つ以上の Resources からの応答を待機している状態です。

GetTransactionName

このメソッドは，トランザクションの名前をわかりやすく表示する文字列を返します。このメソッドは，診断とデバッグのために使用できます。

このメソッドの効果は，対応する Coordinator オブジェクトの GetTransactionName メソッドを呼び出した場合と同じです。クライアントスレッドに関連付けられているトランザクションがない場合は，空の文字列が返されます。

Resume

クライアントスレッドを特定のトランザクションに関連付けます。通常は，次のどちらかのために使用されます。

- 暗黙的なトランザクションの伝搬に使用するために，トランザクションコンテキストをスレッドに関連付ける
- Suspend メソッドによって一時停止されたトランザクションを再開する

クライアントスレッドは，特定のトランザクションに関連付けられます。クライアントスレッドがすでに他のトランザクションに関連付けられている場合は，以前のトランザクションコンテキストが破棄されます。NULL コントロールを使用して Resume を呼び出すと，現在のスレッドに関連付けられたトランザクションはなくなり，トランザクションコンテキストは破棄されます。

注意 Resume を介して設定したトランザクションコンテキストがあれば，呼び出し元のオブジェクトに伝搬されます。

Rollback

クライアントスレッドに関連付けられているトランザクションをロールバックします。これは，対応する Terminator オブジェクトの Rollback メソッドを呼び出すことと同じです。トランザクションが完了し，関連するすべての同期オブジェクトに通知されるまで，このメソッドは戻りません。このメソッドから戻った時点で，クライアントスレッドとトランザクションの関連付けが解除されます。トランザクションが存在するかのように Current を使用しようとすると，CosTransactions.NoTransaction または CORBA.TRANSACTION_REQUIRED などの例外が生成されるか，null オブジェクトリファレンスが返されます。経験則が発生すると，このメソッドは，経験則関連の例外を生成します。

呼び出し元がトランザクションオリジネータでない場合，Rollback は例外 CORBA.NO_PERMISSION を生成します。

RollbackOnly

このメソッドは，ロールバックだけがトランザクションの結果になるように，クライアントスレッドに関連付けられているトランザクションを変更します。この要求の効果は，対応

する Coordinator オブジェクトの RollbackOnly メソッドを呼び出した場合と同じです。Rollback オペレーションの実行が制限されているクライアントでも，RollbackOnly を呼び出すことができます。

SetTimeout

このメソッドは，この後このプログラム内のすべてのスレッドで Current.Begin メソッドを呼び出すことによって開始されるトランザクションに対して，新しいタイムアウトを確立します。

新しいタイムアウトを確立するには，次の seconds パラメータの値を使用します。

Value	効果
= 0	この後で開始されるトランザクションのタイムアウトを，そのトランザクションが使用するトランザクションサービスインスタンスのデフォルトタイムアウトに設定します。
> 0	新しいタイムアウトを指定した秒数に設定します。seconds パラメータが，使用されるトランザクションサービスインスタンスの最大タイムアウト値を超える場合，新しいタイムアウトはその最大値に設定され，範囲内に収められます。

メモ この後でプロセス内の任意のスレッドで Begin を呼び出すことによって作成されたトランザクションが，確立されたタイムアウトを過ぎてもトランザクションの完了を開始できない場合，トランザクションはロールバックされます。トランザクションが完了段階に入る前に（2 フェーズ処理または 1 フェーズ処理を開始する前に）タイムアウトが発生した場合，トランザクションはロールバックされます。それ以外の場合，タイムアウトは無視されます。

Suspend

このメソッドは，クライアントスレッドに現在関連付けられているトランザクションを一時停止し，トランザクションの Control オブジェクトを返します。クライアント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，null オブジェクトリファレンスが返されます。

この Control オブジェクトを Resume メソッドに渡して，このコンテキストを同じスレッドまたは別のスレッドで再確立できます。

Suspend を呼び出すと，クライアントスレッドに関連付けられたトランザクションはなくなります。トランザクションが存在するかのように Current を使用しようとすると，CosTransactions.NoTransaction または CORBA.TRANSACTION_REQUIRED などの例外が生成されるか，null オブジェクトリファレンスが返されます。

TransactionFactory インターフェース

TransactionFactory インターフェースは，プログラムで VisiBroker for .NET 管理以外のトランザクションを開始するためのメソッドを定義します。TransactionFactory インターフェースを使用すると，トランザクションコンテキストの伝搬を直接制御できます。

TransactionFactory オブジェクトは，バインディングなどの CORBA オブジェクトと同じ方法で取得します。

TransactionFactory のメソッド

以下のセクションでは，TransactionFactory の重要なメソッドについて説明します。詳細は，Borland VisiBroker for .NET API ドキュメントを参照してください。

Create

このメソッドは，タイムアウトパラメータ（time_out）を受け取り，新しいトランザクションを作成します。これは，Control オブジェクトを返します。この Control オブジェクトを使用して，新しいトランザクションへの参加を管理または制御できます。Control オブ

ジェクトは任意のスレッドで使用できます。また，他の CORBA オブジェクトと同様に，明示的に受け渡しできます。

メモ　このメソッドを使用するトランザクションには，checked behavior を提供できません。

新しいタイムアウトを確立するには，次の time_out パラメータの値を使用します。

Value	効果
= 0	この後で開始されるトランザクションのタイムアウトを，そのトランザクションが使用するトランザクションサービスインスタンスのデフォルトタイムアウトに設定します。
> 0	新しいタイムアウトを指定した秒数に設定します。seconds パラメータが，使用されるトランザクションサービスインスタンスの最大タイムアウト値を超える場合，新しいタイムアウトはその最大値に設定され，範囲内に収められます。

新しいタイムアウトは，この呼び出しで作成されたトランザクションにのみ適用されます。タイムアウトが発生する前に，トランザクションが完了を開始しなかった場合（2 フェーズ処理または 1 フェーズ処理が開始されなかった場合）は，トランザクションがロールバックされます。

Recreate

このメソッドは，PropagationContext パラメータを使用して新しい Control オブジェクトを作成します。この Control オブジェクトを使用して，トランザクションへの参加を管理または制御できます。通常，アプリケーションではこのメソッドを呼び出しません。

トランザクションの　PropagationContext　を取得するには，トランザクションの Coordinator オブジェクトの CosTransactions.CoordinatorOperations.GetTxcontext メソッドを呼び出します。

Control インターフェース

Control インターフェースを使用すると，プログラムでトランザクションコンテキストを明示的に管理または伝搬できます。Control オブジェクトは，1 つの特定のトランザクションに暗黙的に関連付けられます。

Control インターフェースは，GetCoordinator と GetTerminator の 2 つのメソッドを定義します。GetCoordinator メソッドは Coordinator オブジェクトを返します。このオブジェクトは，トランザクションの参加者によって使用されるメソッドをサポートします。GetTerminator メソッドは Terminator オブジェクトを返します。このオブジェクトは，トランザクションを完了するためのメソッドをサポートします。Terminator オブジェクトと Coordinator オブジェクトがサポートするメソッドは，通常，複数の当事者によって実行されます。この 2 つのオブジェクトにより，これらのメソッドを必要とする当事者にのみメソッドを提供できます。

Control オブジェクトを取得するには，TransactionFactory のメソッドを使用します（111 ページの「TransactionFactory インターフェース」を参照）。また，Current オブジェクトのメソッドを使用して，スレッドに関連付けられている現在のトランザクションに対する Control オブジェクトを取得することもできます。GetControl メソッドまたは Suspend メソッドの詳細は，108 ページの「Current インターフェース」を参照してください。

Control のメソッド

以下のセクションでは，Control の重要なメソッドについて説明します。詳細は，Borland VisiBroker for .NET API ドキュメントを参照してください。

GetCoordinator

このメソッドは, Coordinator オブジェクトを返します。Coordinator は, トランザクションの参加者によって呼び出されるメソッドを提供します。通常，これらの参加者は，回復可能なオブジェクトまたは回復可能なオブジェクトのエージェントです。

GetTerminator

このメソッドは, Terminator オブジェクトを返します。Terminator を使用して, Control に関連付けられたトランザクションをロールバックまたはコミットできます。Terminator オブジェクトを送信できないか, 他の実行環境で使用できないために, Control が要求されたオブジェクトを提供できない場合は，例外 CosTransactions.Unavailable が生成されます。

Terminator インターフェース

Terminator インターフェースは, トランザクションをコミットまたはロールバックするためのメソッドをサポートします。通常，これらのメソッドは，トランザクションオリジネータによって使用されます。ただし, トランザクションの Terminator オブジェクトにアクセスできる場合は，任意のプログラムがトランザクションをコミットまたはロールバックできます。

Terminator のメソッド

以下のセクションでは，Terminator の重要なメソッドについて説明します。詳細は，Borland VisiBroker for .NET API ドキュメントを参照してください。

Commit

トランザクションをコミットする前に, このメソッドはいくつかのチェックを行います。トランザクションにロールバックのみのマークが付けられておらず，トランザクションのすべての参加者がコミットに同意する場合，トランザクションはコミットされ，オペレーションは正常に終了します。それ以外の場合は，トランザクションがロールバックされ，標準の例外 CORBA.TRANSACTION_ROLLEDBACK が生成されます。

report_heuristics パラメータが true の場合，トランザクションサービスは，必要であれば例外 CosTransactions.HeuristicMixed と CosTransactions.HeuristicHazard を使用して, 矛盾があることまたは矛盾した結果が生成されることを報告します。経験則出力に関連するリソースについての情報が，トランザクションサービスのインスタンスに対応する経験則ログファイルに書き込まれます。

トランザクションがコミットされると，このトランザクションの間に回復可能なオブジェクトに加えられた変更がすべて確定され，他のトランザクションやクライアントから認識できるようになります。

Rollback

このメソッドは, トランザクションをロールバックします。トランザクションがロールバックされると，このトランザクションの間に回復可能なオブジェクトに加えられた変更がすべてロールバックされます。リソースによって適用されるアイソレーションの程度に応じて，トランザクションによってロックされたすべてのリソースが他のトランザクションでも使用できるようになります。

トランザクションが完了し，関連するすべての同期オブジェクトに通知されるまで，このメソッドは戻りません。

Coordinator インターフェース

Coordinator インターフェースは, トランザクションの参加者によって使用されるメソッドを提供します。通常，これらの参加者は，回復可能なオブジェクトまたは回復可能なオブジェクトのエージェントです。各 Coordinator は, 1 つのトランザクションに暗黙的に関連付けられます。

Coordinator の次のメソッドは同等です。つまり, これらのメソッドは同じ結果を返します。

- `GetStatus`
- `GetTopLevelStatus`
- `GetParentStatus`

同様に, いくつかのメソッドは, ターゲットオブジェクトとパラメータが同じ Coordinator オブジェクトを参照する場合にのみ `TRUE` を返します。したがって，次のメソッドも同等です。

- `IsSameTransaction`
- `IsRelatedTransaction`
- `IsAncestorTransaction`
- `IsDescendantTransaction`

また，次のメソッドは同等です。

- `HashTransaction`
- `HashTopLevelTran`

Coordinator のメソッド

以下のセクションでは，Coordinator の重要なメソッドについて説明します。詳細は, Borland VisiBroker for .NET API ドキュメントを参照してください。

GetStatus

このメソッドは，ターゲットオブジェクトに関連付けられているトランザクションの状態を列挙値（`enum Status`）で返します。ターゲットオブジェクトに関連付けられているトランザクションがない場合，このメソッドは値 `StatusNoTransaction` を返します。

VisiBroker for .NET ではネストしたトランザクションはサポートされていないため, `GetStatus`, `GetTopLevelStatus`, `GetParentStatus` の各メソッドは同じ結果を返します。

戻り値は次のとおりです。これらは CosTransactions.idl で定義されます。

StatusActive	StatusUnknown
StatusMarkedRollback	StatusNoTransaction
StatusPrepared	StatusPreparing
StatusCommitted	StatusCommitting
StatusRolledBack	StatusRollingBack

各ステータス値の詳細は，109 ページの「GetStatus」を参照してください。

GetTransactionName

このメソッドは, トランザクションの名前をわかりやすく表示する文字列を返します。このメソッドは, 診断とデバッグのために使用できます。クライアントスレッドに関連付けられているトランザクションがない場合は，空の文字列が返されます。

GetTxcontext

GetTxcontext メソッドは PropagationContext を返します。この PropagationContext をトランザクションサービスドメインで使用して，トランザクションを別のトランザクションサービスドメインにエキスポートできます。

HashTransaction

このメソッドは，ターゲットオブジェクトに関連付けられているトランザクションのハッシュコードを返します。各トランザクションは 1 つのハッシュコードを持ちます。このハッシュコードを他のトランザクションのハッシュコードと照らし合わせることで，Coordinator が同じかどうかを効率よく確認できます。2 つの Coordinator のハッシュコードが異なる場合，これらは別のトランザクションを表しています。2 つのハッシュコードが等しい場合は，IsSameTransaction を使用して，それらが等しいかどうかを確認する必要があります。これは，2 つの Coordinator のハッシュコードが等しくても，実際には異なるトランザクションを表すことがあるためです。

IsSameTransaction

ターゲットオブジェクトとパラメータオブジェクトの両方が同じトランザクションを参照する場合にのみ，このメソッドは true を返します。

RegisterResource

このメソッドは，指定されたリソースをターゲットオブジェクトに関連付けられたトランザクションの参加者として登録します。トランザクションが終了すると，リソースは，トランザクションの間に実行された更新を準備，コミット，またはロールバックする要求を受け取ります。Resource のメソッドの詳細は，116 ページの「Resource インターフェース」を参照してください。

このメソッドは，このリソースによってリカバリ中に使用される RecoveryCoordinator を返します。

RegisterSynchronization

このメソッドは，トランザクションが完了する前後に，指定された Synchronization オブジェクトが必要な処理を実行するための通知を受けるように，このオブジェクトを登録します。これらのメソッドについては，Synchronization インターフェースの解説の中で説明されています。117 ページの「Synchronization インターフェース」を参照してください。

RegisterSubtranAware

VisiBroker for .NET ではネストしたトランザクションはサポートされていないため，このメソッドは，常に例外 CosTransactions.SubtransactionsUnavailable を生成します。

RollbackOnly

このメソッドは，ロールバックだけがトランザクションの結果になるように，Coordinator に関連付けられているトランザクションを変更します。

RecoveryCoordinator インターフェース

Coordinator にリソースを登録すると，RecoveryCoordinator が返されます。RecoveryCoordinator は，暗黙的に 1 つのリソース登録要求に関連付けられ，そのリソースによってのみ使用されます。リカバリが必要な場合，リソースはリカバリ処理で RecoveryCoordinator を使用できます。

また，トランザクションの現在の状態を取得する必要がある場合にも，RecoveryCoordinator を使用できます。たとえば，リソースは独自のタイムアウトを設定で

きるので，タイムアウトまでにコミットやロールバックが行われなかった場合は，ReplayCompletion を呼び出してトランザクションの状態を特定できます。

RecoveryCoordinator のメソッド

次のセクションでは，RecoveryCoordinator のメソッドについて説明します。詳細は，Borland VisiBroker for .NET API ドキュメントを参照してください。

ReplayCompletion

このメソッドは，リソースが使用可能であることをトランザクションサービスに通知します。通常，このメソッドはリカバリ中に使用されます。また，リソースは，このメソッドを使用してトランザクションの状態を特定できます。

メモ このメソッドは，完了を開始しません。

Resource インターフェース

VisiBroker for .NET では，2 フェーズコミットプロトコルを使用して，各リソースが登録されている最上位のトランザクションを完了します。つまり，この各リソースは，トランザクションの間に変更される可能性があります。Resource インターフェースは，各リソースでトランザクションサービスによって呼び出されるメソッドを定義します。Resource インターフェースをサポートする各オブジェクトは，1 つの最上位トランザクションに暗黙的に関連付けられます。

VisiBroker for .NET の Resource インターフェースは CosTransactions.idl ファイルに含まれていますが，使用する Resource には実装を提供する必要があります。通常のアプリケーションは，Resource を実装しません。

Resource のメソッド

以下のセクションでは，Resource の重要なメソッドについて説明します。詳細は，Borland VisiBroker for .NET API ドキュメントを参照してください。

Commit

このメソッドは，リソースに関連付けられているすべての変更をコミットしようとします。経験則出力の例外が生成された場合，リソースは，Forget メソッドが実行されるまで，経験則による決定を永続的ストレージに保存する必要があります。これにより，リカバリ中に Commit が再度呼び出されても，同じ結果を返すことができます。それ以外の場合，リソースは，トランザクションに関するすべての情報をすぐに破棄できます。

CommitOnePhase

このメソッドは，トランザクションの間に加えられたすべての変更をコミットするようにリソースに要求します。このメソッドは，トランザクションに参加するリソースが 1 つしかない場合に使用するように最適化されています。リソースでは，最初に Prepare を呼び出し，次に Commit または Rollback を呼び出すかわりに，このメソッドを呼び出すことができます。

経験則出力の例外が生成された場合，リソースは，Forget メソッドが実行されるまで，経験則による決定を永続的ストレージに保存する必要があります。これにより，リカバリ中に CommitOnePhase が再度呼び出されても，同じ結果を返すことができます。それ以外の場合，リソースは，トランザクションに関するすべての情報をすぐに破棄できます。

CommitOnePhase の処理中にエラーが発生した場合は，エラーが修復されると，同じメソッドが再度呼び出されます。リソースは 1 つしかないため，例外 HeuristicHazard を使用して，そのリソースに関連する経験則による決定が報告されます。

Forget

VisiBroker for .NET は，ヒューリスティックな例外を受け取ると，その例外を記録します。トランザクションサービスは，最終的にリソースの Forget を呼び出します。つまり，リソースは，経験則の例外を生成したトランザクションに関するすべての情報を破棄できます。このメソッドは，経験則の例外が Rollback，Commit，または CommitOnePhase から生成された場合にのみ呼び出されます。

Prepare

このメソッドは準備処理を実行します。これは，Resource オブジェクトの 2 フェーズコミットプロトコルにおける最初の手順です。処理が完了すると，このメソッドは次の Vote 値のいずれかを返します。

- VoteReadOnly— リソースに関連付けられた永続的データは，トランザクションによって変更されていません。
- VoteCommit— 次のデータが永続的ストレージに保存されています。

  トランザクションの間に変更されたすべてのデータ

  RecoveryCoordinator オブジェクトへの参照

  リソースの準備が整ったことを示すデータ

- VoteRollback— 関連データを保存できない，結果に一貫性がない，トランザクションに関する情報がない（たとえば，クラッシュ後の状態）など，リソースがロールバックを要求する状況が発生しました。

VoteReadOnly または VoteRollback が返された場合，リソースは，トランザクションに関するすべての情報を破棄できます。

経験則出力の例外が生成された場合，リソースは，Forget メソッドが呼び出されるまで，経験則による決定を永続的ストレージに保存する必要があります。これにより，Prepare が再度呼び出されても，同じ結果を返すことができます。

Rollback

このメソッドは，リソースオブジェクトに関連付けられているすべての更新をロールバックします。

経験則出力の例外が生成された場合，リソースは，Forget メソッドが呼び出されるまで，経験則による決定を永続的ストレージに保存する必要があります。これにより，リカバリ中に Rollback が再度呼び出されても，同じ結果を返すことができます。それ以外の場合，リソースは，トランザクションに関するすべての情報をすぐに破棄できます。

Synchronization インターフェース

Synchronization インターフェースは，2 フェーズコミットプロトコルまたは 1 フェーズコミットプロトコルを開始する前と，それが完了した後に，トランザクションオブジェクトが通知を受けるためのメソッドを定義します。 CosTransactions モジュールの Synchronization インターフェースには，次の 2 つのメソッドが用意されています。

- BeforeCompletion— トランザクションのコミットを開始する前に，BeforeCompletion が呼び出されます。
- AfterCompletion— トランザクションが完了した後で，トランザクションオブジェクトが通知を受けます。これは，トランザクションがコミットされてもロールバックされても，すべてのトランザクションに適用されます。

次の 2 つの制限に注意する必要があります。

- BeforeCompletion を呼び出す際に，トランザクションサービスが Synchronization オブジェクトにアクセスできない場合，トランザクションはロールバックされます。完了

後に Synchronization オブジェクトを使用できない場合, そのオブジェクトは無視されます。

- トランザクションサービスインスタンスのエラーが修復されても, Synchronization オブジェクトは失われたままです。完了は再開されますが, Synchronization オブジェクトは元に戻りません。エラーが発生した場合, VisiTransact 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完了した状況は, Synchronization オブジェクトに通知されません。

メモ BeforeCompletion が呼び出されなかった場合に, AfterCompletion が呼び出されることがあります。完了処理の開始時にトランザクションがまだコミットを継続している場合にのみ, BeforeCompletion が呼び出されます。AfterCompletion は常に呼び出されます。ただし, トランザクションが完了する前に, トランザクションサービスがクラッシュした場合を除きます。

Synchronization オブジェクトを回復することはできません。トランザクションサービスのインスタンスにエラーが発生した場合, 完了したトランザクションに Synchronization オブジェクトは含まれません。

メモ これらのメソッドのシグニチャは Synchronization インターフェースによって固定されていますが, 実装はユーザーによって定義されます。これにより, アプリケーションでは, トランザクションが完了する前および後のトランザクションのキーポイントで処理をカスタマイズできます。

Synchronization のメソッド

以下のセクションでは, Synchronization の重要なメソッドについて説明します。詳細は, Borland VisiBroker for .NET API ドキュメントを参照してください。

AfterCompletion

これは, トランザクションの完了後にカスタマイズされた処理を実行するためのメソッドとして, ユーザーによって記述されます。これは, 本質的にコールバックです。

メモ AfterCompletion メソッドは, 通常の処理中に常に呼び出されます。

Synchronization インターフェースの IDL は, TransactionalObject インターフェースを継承します。プログラマは, この IDL に準拠する AfterCompletion メソッドの実装を記述する必要があります。

特定のトランザクションを処理する際に AfterCompletion を呼び出す場合は, 次の操作を行う必要があります。

1. トランザクションオリジネータまたは他のトランザクション参加者によって Synchronization オブジェクトを作成する必要があります。
2. トランザクションの Coordinator を取得し, Coordinator と Current で RegisterSynchronization メソッドを呼び出して, Synchronization オブジェクトを登録する必要があります。RegisterSynchronization メソッドの詳細は, 114 ページの「Coordinator インターフェース」を参照してください。登録は, トランザクションが作成された後で, 2 フェーズコミットプロトコルが開始される前に, 行う必要があります。

複数の Synchronization オブジェクトを作成し, それらを 1 つのトランザクションに登録することができます。

トランザクションサービスは, 2 フェーズコミットプロトコルの完了後に, このメソッドを呼び出します。たとえば, トランザクションオブジェクトで AfterCompletion を使用して, トランザクションの結果を検索できます。これは, 回復可能なオブジェクトではなく, 結果が自動的に通知されないトランザクションオブジェクトの場合に特に便利です。

GetStatus を呼び出して, トランザクションにロールバックのマークが付けられているかどうかを確認することもできます。

Synchronization は TransactionalObject を継承するため, Current オブジェクトを介してトランザクションコンテキストを使用できます。

例外はすべて無視されます。

BeforeCompletion

これは，トランザクションの完了が開始されたときにカスタマイズされた処理を実行するためのメソッドとして，ユーザーによって記述されます。このメソッドは，トランザクションが正常に完了処理を継続している場合にのみ呼び出されます。これは，本質的にコールバックです。

メモ：アプリケーションがコミットを開始すると，トランザクションサービスがトランザクションの完了を開始する前に，BeforeCompletion メソッドが呼び出されます。ロールバック要求の場合，BeforeCompletion メソッドは呼び出されません。

Synchronization インターフェースの IDL は, TransactionalObject インターフェースを継承します。プログラマは，この IDL に準拠する BeforeCompletion メソッドの実装を記述する必要があります。

特定のトランザクションを処理する際に BeforeCompletion を呼び出す場合は，Coordinator インターフェースの RegisterSynchronization メソッドを使用して，Synchronization オブジェクトを登録する必要があります。Synchronization オブジェクトの登録は，トランザクションオブジェクトまたは回復可能なサーバーから行います。RegisterSynchronization メソッドの詳細は，114 ページの「Coordinator インターフェース」を参照してください。登録は，トランザクションが作成された後で，2 フェーズコミットプロトコルが開始される前に，行う必要があります。

複数の Synchronization オブジェクトを作成し, それらを 1 つのトランザクションに登録することができます。

トランザクションサービスは，トランザクション作業が完了した後で, 2 フェーズコミットプロトコルが開始される前に（参加しているリソースの Prepare が呼び出される前に），このメソッドを呼び出します。完了処理の開始時にトランザクションがまだコミットを継続している場合にのみ，トランザクションサービスは BeforeCompletion を呼び出します。つまり，Terminator.Commit が呼び出され，トランザクションにロールバックのマークが付けられていない場合です。Terminator.Rollback が呼び出された場合，つまり最初の Synchronization オブジェクトによってトランザクションにロールバックのマークが付けられた場合，またはトランザクションにロールバックのマークがすでに付けられている場合，このトランザクションでは BeforeCompletion は呼び出されません。

トランザクションを確実にロールバックするには，このメソッド内で RollbackOnly メソッドを呼び出します。GetStatus を呼び出して, トランザクションにロールバックのマークが付けられているかどうかを確認することもできます。ただし, メソッドが呼び出された時点では，その状態に基づいてトランザクションが実際にコミットされるかどうかを示すことはできません。

Synchronization インターフェースは TransactionalObject を継承するため, Current オブジェクトを介してトランザクションコンテキストを使用できます。 つまり，BeforeCompletion は，GetStatus や GetControl など，Current オブジェクトのすべてのメソッドを使用できます。

Synchronization オブジェクトによって CORBA の例外が生成されると, トランザクションがロールバックされます。

TransactionalObject インターフェース

TransactionalObject は，トランザクションオブジェクトのメソッドを呼び出す際に，トランザクションコンテキストを自動的に伝搬するためのインターフェースです。TransactionalObject インターフェースはメソッドを定義しません。

トランザクションに対して動作するメソッドには，トランザクションコンテキストに対するアクセス権が必要です。これらのメソッドでトランザクションコンテキストを使用できるようにするには，次の 2 つの方法があります。

- 明示的な伝播 — メソッドは，トランザクションコンテキストを Terminator, Control, Coordinator，または PropagationContext 構造体として受け渡します。
- 暗黙的な伝搬 — トランザクションコンテキストは，メソッドの呼び出し時に自動的（および暗黙的）に渡されます。

暗黙的な伝搬の方が簡単なので，通常はこの方法が使用されます。　　これは，TransactionalObject インターフェースによってトランザクションオブジェクトに提供される機能です。TransactionalObject のインスタンスは，暗黙的な伝搬に参加できます。暗黙的な伝搬では，クライアントスレッドに関連付けられているトランザクションコンテキストが，メソッドの呼び出しを介して TransactionalObject インターフェースに自動的に伝搬されます。

VisiBroker for .NET 管理のトランザクションを使用するには，すべてのトランザクションオブジェクトが TransactionalObject を継承する必要があります。VisiBroker for .NET 管理のトランザクションを使用することで，checked behavior の機能を活用できます。

トランザクションコンテキストは，`CosTransactions.TransactionalObject` を継承するオブジェクトに常に暗黙的に渡されます。また，プログラムに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こともできます。

第 13 章

セキュリティサービスの使い方

今日，インターネットを使った分散アプリケーションで事業を運営する企業はますます増え，クオリティの高いアプリケーションセキュリティに対するニーズがさらに高まっています。

クレジットカードの番号や預金の残高などの機密情報は，インターネット接続を利用して Web ブラウザと商用 Web サーバーとの間を日常的にやり取りされます。たとえば，インターネットを利用して銀行と取引するユーザーにとっては，次の点が確実である必要があります。

- 銀行を装った違法サイトではなく，自分の口座がある銀行のサーバーと実際に通信をしていること。
- 銀行と交換するデータが，ネットワークに忍び込んで不正に情報を得ようとする不正ユーザーにはわからない形式になっていること。
- 銀行のソフトウェアとやり取りするデータが，不正な修正が加えられたりせずに元の状態のまま届くこと。たとえば，50,000 円の手形の支払い指図が誤ってまたは不正に 5,000,000 円に変えられることがあってはなりません。

VisiBroker for .NET Security は，クライアントが銀行のサーバーを認証できるようにするサービスです。また，銀行側のサーバーでも，セキュリティで保護された接続を利用してクライアントを認証できます。従来のアプリケーションでは，セキュリティで保護された接続が確立されると，クライアントは認証のためにユーザー名とパスワードを送信します。この方法は引き続き利用でき，VisiBroker for .NET Security の接続が確立されたら，ユーザー名とパスワードを暗号化して交換できます。

VisiBroker for .NET Security の概要

VisiBroker for .NET Security を使用して，クライアントとサーバーの間にセキュリティで保護された接続を確立できます。また，VisiBroker for .NET Security は，セキュリティで保護された通信のためのフレームワークを提供します。VisiBroker for .NET Security では，SSL と TLS（Transport Layer Security）の通信に Microsoft Windows Secure Channel（Schannel）のライブラリが使用され，暗号化操作に Microsoft CryptoAPI が使用されます。

VisiBroker for .NET Security には次の機能があります。

- J2EE サーバーと CORBA サーバーの相互運用：VisiBroker for .NET Security は，CORBA Common Secure Interoperability 仕様（CSIv2）を介して，EJB セキュリティとシームレスに相互運用されます。
- Microsoft Windows 証明書ストアとの統合：VisiBroker for .NET Security では，公開キーと秘密キーの管理に Microsoft Windows 証明書ストアが使用されます。
- ASP.NET との統合：VisiBroker for .NET Security では，ASP.NET アプリケーションによって認証されたセキュリティ ID が J2EE サーバーや CORBA サーバーに伝搬されます。
- Secure Transport Layer：VisiBroker for .NET Security では，セキュリティで保護されているトランスポート層として SSL プロトコルと TLS プロトコルを利用しています。これらのプロトコルは，信頼モデルを介して，メッセージの機密性，メッセージの完全性，および証明書ベースの認証をサポートします。
- Borland GateKeeper との統合：VisiBroker for .NET Security は，GateKeeper を介したセキュリティで保護された接続をサポートします。詳細は，第 16 章「VisiBroker for .NET と Borland GateKeeper の使用」および『VisiBroker GateKeeper ガイド』を参照してください。

VisiBroker for .NET Security の有効化

デフォルトでは，VisiBroker for .NET Security は無効になっています。VisiBroker for .NET Security を有効にするには，次のように，設定ファイルに <security> セクションを追加します。

```
<visinet>
   <security enabled="true">
   </security>
</visinet>
```

または，janeva.security プロパティを true に設定することで，セキュリティを有効化することもできます。手順については，プロパティに関する章を参照してください。

J2EE サーバーと CORBA サーバーの相互運用

VisiBroker for .NET Security では，次の 2 種類のユーザー認証がサポートされています。

- 「ユーザー名／パスワード認証」
- 「証明書ベースの認証」

VisiBroker for .NET Security では，.NET リモート処理 API，CORBA ベースの API，および設定ファイルを使用して，セキュリティ ID を設定できます。これらの方法について，以下のセクションで説明します。

ユーザー名／パスワード認証

VisiBroker for .NET Security では，J2EE サーバーまたは CORBA サーバーでユーザーを認証する必要がある場合に，複数の方法でユーザーの認証情報を設定し，サーバー側に渡すことができます。ユーザー名／パスワード認証では，VisiBroker for .NET クライアントがユーザー名とパスワードをサーバーに渡すことでユーザーを認証できます。ユーザー名／パスワード認証を実装するには，.NET リモート処理 API，CORBA ベースの API，またはアプリケーション設定ファイルのいずれかを使用します。

.NET リモート処理 API を使用したユーザー名／パスワード認証

以下の例は，.NET リモート処理 API を使用して，ユーザー名／パスワード認証を行う方法を示しています。

最初に，リモートプロキシリファレンスを解決します。

```
// 設定ファイルで既知のリモートオブジェクトとして設定された
// CartHomeRemotingProxy を作成します
CartHome home = new CartHomeRemotingProxy();
```

次に，このリモートプロキシオブジェクトの Sink プロパティを解決します。

```
// セキュリティ認証情報を設定します
IDictionary props =
   System.Runtime.Remoting.Channels.ChannelServices.GetChannelSinkProperties(
      home);
```

次に，ユーザー名とパスワードのプロパティを設定します。

```
props["username"] = "joeshopper";
props["password"] = "joepass";
```

また，領域を設定することもできます。

```
props["realm"] = "myuprealm";
```

realm プロパティを指定しないと，領域のデフォルトは default になります。

メモ アプリケーションサーバーによってデフォルトの領域名が異なる場合があります。デフォルトの領域名は設定ファイルで設定できます。設定ファイルで設定されたデフォルトの領域を上書きするには，上のように，コマンドラインまたはプログラムを使用してプロパティを設定します。詳細は，第 4 章「プロパティの設定」を参照してください。

プロパティを設定したら，リモートプロキシのメソッドを呼び出すことができます。ユーザー名，パスワード，および領域は，呼び出しコンテキストの一部として，サーバー側に透過的に渡されます。

```
// Cart セッションの新規インスタンスを作成します
Cart cart = home.Create(...);
```

リモートプロキシに対して最初の呼び出しを行う前に，認証情報を設定する必要があります。認証情報を設定しないとサーバーに渡されません。

同じサーバー上のすべてのオブジェクトが，セキュリティで保護された同じ接続を使用することに注意してください。最初の呼び出しが完了した後で，引き続き同じオブジェクトに対する呼び出しまたは同じサーバー上の別のオブジェクトに対する呼び出しを行うと，最初の呼び出しで確立された認証情報が使用されます。認証情報を変更するには，Sink プロパティを解決し，username プロパティと password プロパティを再度設定します。

次の例で，カートオブジェクトの認証情報を再度設定する必要はありません。このカートは，セキュリティで保護されたホームオブジェクトへの接続によって確立された認証情報と同じ認証情報を使用します。

```
// 新しい本をカートに追加します
Item book = new Book();
book.Title = "War and Peace";
book.Price = 20.99f;
cart.AddItem(book);
```

他のサンプルコードは，
<janeva_install_dir>¥examples¥Advanced¥Security¥RemotingUsernameClient
ディレクトリにあります。

CORBA ベースの API を使用したユーザー名／パスワード認証

以下の例は, CORBA ベースの API を使用してユーザー認証情報を確立する方法を示しています。

最初に，orb インスタンスで VisiBroker for .NET セキュリティコンテキストを解決します。Janeva.Security.Context は, ユーザーの ID を操作するための API を公開するオブジェクトです。

```
// 初期化，ORB の
CORBA.ORB orb = CORBA.ORB.Init(args);

// セキュリティコンテキストを解決します
Janeva.Security.Context context =
   (Janeva.Security.Context) orb.ResolveInitialReferences("SecurityContext");
```

次に, Janeva.Security.IdentityWallet クラスを使用して, ユーザー名, パスワード, および領域を設定します。

```
// 認証情報を使用してウォレットを作成します
Janeva.Security.IdentityWallet wallet = new Janeva.Security.IdentityWallet(
   "joeshopper", "joepass".ToCharArray(), "myuprealm");
```

realm プロパティを指定しないと，領域のデフォルトは default になります。

メモ アプリケーションサーバーによってデフォルトの領域名が異なる場合があります。デフォルトの領域名は設定ファイルで設定できます。設定ファイルで設定されたデフォルトの領域を上書きするには, 上のように, コマンドラインまたはプログラムを使用してプロパティを設定します。詳細は，第 4 章「プロパティの設定」を参照してください。

最後に，ウォレットを使用してセキュリティコンテキストの Login メソッドを呼び出します。

```
// ウォレットを使用してセキュリティコンテキストにログインします。
context.Login(wallet);
```

Janeva.Security.Context オブジェクトには複数の Login メソッドがあります。詳細は, VisiBroker for .NET API リファレンスを参照してください。

同じサーバー上のすべてのオブジェクトが，セキュリティで保護された同じ接続を使用することに注意してください。最初の呼び出しが完了した後で, 引き続き同じオブジェクトに対する呼び出しまたは同じサーバー上の別のオブジェクトに対する呼び出しを行うと，最初の呼び出しで確立された認証情報が使用されます。認証情報を変更するには，Logout を呼び出した後で，Login を再度呼び出します。

Login メソッドを使用して認証情報を設定したら，サーバーのメソッドを呼び出すことができます。

```
// Cart セッションの新規インスタンスを作成します
Cart cart = home.Create(...);
```

Janeva.Security.Context にログインしたら, その後のリモート呼び出しでも同じ認証情報が使用されます。

```
// 新しい本をカートに追加します
Item book = new Book();
book.Title = "War and Peace";
book.Price = 20.99f;
cart.AddItem(book);
```

設定ファイルを使用したユーザー名／パスワード認証

以下の例は，設定ファイルを使用してセキュリティ認証情報を設定する方法を示しています。

```
<configuration>
   <visinet>
      <security enabled="true">
         <identity>
            <username>joeshopper</username>
            <password>joepass</password>
            <realm>myuprealm</realm>
         </identity>
      </security>
   </visinet>
</configuration>
```

設定ファイルで設定された ID は，アプリケーション全体に影響を及ぼします。各リモート呼び出しで同じ ID が使用されます。

証明書ベースの認証

VisiBroker for .NET の証明書は，Microsoft Windows 証明書ストアに基づいてサポートされます。VisiBroker for .NET アプリケーションで証明書を使用するには，証明書を証明書ストアにインポートする必要があります。証明書を発行および管理する方法については，Microsoft のドキュメントを参照してください。

任意で設定できる証明書の属性に「フレンドリ名」があります。VisiBroker for .NET では，特定の証明書を処理するための ID として証明書のフレンドリ名が使用されます。証明書にフレンドリ名がない場合は，Microsoft Windows の［インターネットオプション］画面で設定できます。

メモ　証明書を使用してクライアントを認証する場合，その証明書には公開キーと秘密キーの両方が必要です。これは，SSL ／ TLS プロトコルの要件です。

.NET リモート処理 API を使用した証明書ベースの認証

以下の例は，.NET リモート処理 API を使用して，証明書ベースの認証を行う方法を示しています。

最初に，リモートプロキシリファレンスを解決します。

```
// 設定ファイルで既知のリモートオブジェクトとして設定された
// CartHomeRemotingProxy を作成します
CartHome home = new CartHomeRemotingProxy();
```

次に，このリモートプロキシオブジェクトの Sink プロパティを解決します。

```
// セキュリティ認証情報を設定します
IDictionary props =
   System.Runtime.Remoting.Channels.ChannelServices.GetChannelSinkProperties(
      home);
```

次に，証明書のフレンドリ名を設定します。

```
props["certificate"] = "joeshopper";
```

メモ　フレンドリ名を使用するかわりにアスタリスク（*）を指定して，使用する証明書の指定を VisiBroker for .NET に任せることもできます。

プロパティを設定したら，リモートプロキシのメソッドを呼び出すことができます。証明書は，呼び出しコンテキストの一部として，サーバー側に透過的に渡されます。

```
// Cart セッションの新規インスタンスを作成します
Cart cart = home.Create(...);
```

リモートプロキシに対して最初の呼び出しを行う前に，認証情報を設定する必要があります。認証情報を設定しないとサーバーに渡されません。

同じサーバー上のすべてのオブジェクトが，セキュリティで保護された同じ接続を使用することに注意してください。最初の呼び出しが完了した後で，引き続き同じオブジェクトに対する呼び出しまたは同じサーバー上の別のオブジェクトに対する呼び出しを行うと，最初の呼び出しで確立された認証情報が使用されます。認証情報を変更するには，Sink プロパティを解決し，username プロパティと password プロパティを再度設定します。

次の例で，カートオブジェクトの認証情報を再度設定する必要はありません。このカートは，セキュリティで保護されたホームオブジェクトへの接続によって確立された認証情報と同じ認証情報を使用します。

```
// 新しい本をカートに追加します
Item book = new Book();
book.Title = "War and Peace";
book.Price = 20.99f;
cart.AddItem(book);
```

他のサンプルコードは，
<janeva_install_dir>¥examples¥Advanced¥Security¥RemotingCertificateClient
ディレクトリにあります。

CORBA ベースの API を使用した証明書ベースの認証

以下の例は，CORBA ベースの API を使用して証明書ベースの認証を行う方法を示しています。

最初に，orb インスタンスで VisiBroker for .NET セキュリティコンテキストを解決します。Janeva.Security.Context は，ユーザーの ID を操作するための API を公開するオブジェクトです。

```
// 初期化，ORB の
CORBA.ORB orb = CORBA.ORB.Init(args);

// セキュリティコンテキストを解決します
Janeva.Security.Context context =
   (Janeva.Security.Context) orb.ResolveInitialReferences("SecurityContext");
```

証明書のフレンドリ名を設定するには，Janeva.Security.CertificateWallet クラスを使用する必要があります。

```
// 認証情報を使用してウォレットを作成します
Janeva.Security.CertificateWallet wallet = new
   Janeva.Security.CertificateWallet("joeshopper",
      CertificateWallet.CLIENT_AUTHENTICATION);
```

2 番目のパラメータは，証明書の使用方法を定義します。このパラメータをセキュリティで保護されたサーバーで使用する場合は，設定が異なります（128 ページの「.NET サーバーのセキュリティの有効化」を参照）。このパラメータの他の値については，VisiBroker for .NET API リファレンスを参照してください。

メモ　フレンドリ名を使用するかわりにアスタリスク（*）を指定して，使用する証明書の指定を VisiBroker for .NET に任せることもできます。

最後に，ウォレットを使用してセキュリティコンテキストの Login メソッドを呼び出します。

```
// ウォレットを使用してセキュリティコンテキストにログインします。
context.Login(wallet);
```

Janeva.Security.Context オブジェクトには複数の Login メソッドがあります。詳細は，VisiBroker for .NET API リファレンスを参照してください。

同じサーバー上のすべてのオブジェクトが，セキュリティで保護された同じ接続を使用することに注意してください。最初の呼び出しが完了した後で，引き続き同じオブジェクトに対する呼び出しまたは同じサーバー上の別のオブジェクトに対する呼び出しを行うと，最

初の呼び出しで確立された認証情報が使用されます。認証情報を変更するには，Logout を呼び出した後で，Login を再度呼び出します。

Login メソッドを使用して認証情報を設定したら，オブジェクトのメソッドを呼び出すことができます。

```
// Cart セッションの新規インスタンスを作成します
Cart cart = home.Create(...);
```

Janeva.Security.Context にログインしたら，その後のリモート呼び出しでも同じ認証情報が使用されます。

```
// 新しい本をカートに追加します
Item book = new Book();
book.Title = "War and Peace";
book.Price = 20.99f;
cart.AddItem(book);
```

設定ファイルを使用した証明書ベースの認証

以下の例は，設定ファイルで証明書を指定する方法を示しています。

```
<configuration>
   <visinet>
      <security enabled="true">
         <identity>
            <certificate>joeshopper</certificate>
         </identity>
      </security>
   </visinet>
</configuration>
```

設定ファイルで設定された証明書は，アプリケーション全体に影響を及ぼします。証明書で示される ID が各リモート呼び出しで使用されます。

ASP.NET の統合

VisiBroker for .NET Security と ASP.NET との統合は，ID アサーションの概念に基づいています。ASP.NET 上の VisiBroker for .NET ランタイムは，外向き呼び出しを行うたびに，2 つの ID を呼び出し先のサーバーに渡します。Janiva ランタイムは，呼び出し先のサーバーに自分自身を通知し，呼び出し元の ID を受け入れるようにアサートします。

呼び出し元の ID は，ブラウザなどのクライアントが ASP.NET 層と通信する際に使用する ID です。VisiBroker for .NET がこの ID を呼び出し元の ID としてアサートする場合，実際には，ASP.NET 層が呼び出し元の認証を信頼して呼び出し元のかわりに要求を実行していることを呼び出し先のサーバーにアサートしています。

ASP.NET 層からのアサーションを受け入れるかどうかは，呼び出し先のサーバーが判断します。ASP.NET 層は自分自身を通知するため，呼び出し先のサーバーは，この層を認証し，この層にこのアサーションの権限があるかどうかを判断できます。

メモ　サーバーでは，どのピア ID（この場合は，ASP.NET 層の ID）からのアサーションを受け入れるかを定義する設定を明示的に行う必要があります。詳細は，サーバーのマニュアルを参照してください。

ASP.NET の設定

ASP.NET 環境では，ユーザーが認証されているかどうかを VisiBroker for .NET が暗黙的に検出し，ユーザーの ID を呼び出し元の ID としてサーバー側に渡します。122 ページの「J2EE サーバーと CORBA サーバーの相互運用」の例で示されているように，ピア ID を明示的に確立するには，VisiBroker for .NET Security API を使用します。

次の例は，設定ファイルを使用して，ASP.NET アプリケーションのピア ID を確立する方法を示しています。この例は，127 ページの「設定ファイルを使用した証明書ベースの認証」の例に似ていることに注意してください。

```
<configuration>
   <visinet>
      <security enabled="true">
         <identity>
            <username>peer</username>
            <password>pwd</password>
      ...
</configuration>
```

メモ　Janeva.Security.Context.ImportIdentity() API を使用して，呼び出し元の ID を明示的に設定することもできます。これにより，ASP.NET 環境以外でも，信頼モデルを使用できます。Janeva.Security.Context.ImportIdentity の詳細は，VisiBroker for .NET API リファレンスを参照してください。

他のサンプルコードは，
<janeva_install_dir>¥examples¥Advanced¥Security¥AspNetClient
ディレクトリにあります。

.NET サーバーのセキュリティの有効化

セキュリティで保護された .NET サーバーアプリケーションでは，janeva.security.server プロパティを true に設定することで，サーバー側で VisiBroker for .NET Security を有効にできます。次の例は，このプロパティをアプリケーション設定ファイルで設定する方法を示しています。

```
<configuration>
   <visinet>
      <security>
         <server enabled="true" defaultPort="15000">
            <certificate>cert_friendly_name</certificate>
         </server>
      ...
```

VisiBroker for .NET サーバーが SSL／TSL 通信に使用するポートをサーバー側で設定するには，janeva.security.server.defaultPort プロパティを設定します。このプロパティを設定ファイルで設定する方法については，前述の例を参照してください。

SSL ／ TSL プロトコルの要件により，サーバー側を証明書で識別する必要があります。それには，CORBA ベースの API または設定ファイルのプロパティを使用します。

次の例は，設定ファイルで証明書 ID を設定する方法を示しています。

```
<configuration>
   <visinet>
      <security>
         <server enabled="true" defaultPort="15000">
            <certificate>cert_friendly_name</certificate>
         </server>
      ...
```

次の例は，CORBA ベースの API を使用して証明書 ID を設定する方法を示しています。

最初に，orb インスタンスで VisiBroker for .NET セキュリティコンテキストを解決します。Janeva.Security.Context は，ID を操作するための API を公開するオブジェクトです。

```
Janeva.Security.Context context = (Janeva.Security.Context)
   CORBA.ORB.Init().ResolveInitialReferences("SecurityContext");
```

証明書を設定するには,Janeva.Security.CertificateWallet クラスを使用する必要があります。

```
Janeva.Security.CertificateWallet wallet = new Janeva.Security.CertificateWallet(
   "joeshopper", Janeva.Security.CertificateWallet.SERVER_AUTHENTICATION);
```

2 番目のパラメータは，サーバーの証明書の使用方法を定義します。このパラメータの他の値については，VisiBroker for .NET API リファレンスを参照してください。

最後に，ウォレットを使用してセキュリティコンテキストの Login メソッドを呼び出します。

```
context.Login(wallet);
```

Janeva.Security.Context オブジェクトには複数の Login メソッドがあります。詳細は，VisiBroker for .NET API リファレンスを参照してください。

メモ CORBA ベースの API による設定を使用する場合は，サーバーが着信要求の監視を開始する前に，つまりコードで CORBA.ORB.Run() メソッドを呼び出す前に，証明書を設定する必要があります。

他のサンプルコードは,<janeva_install_dir>¥examples¥Advanced¥Security¥SslServer ディレクトリにあります。

第 14 章

部分的に信頼されるアプリケーションとの VisiBroker for .NET の使用

コードアクセスセキュリティは，部分的に信頼されるコードをユーザーに問い合わせることなく実行するようにシステムを設定できる強力な機能です。これはデフォルトの設定になっています。さらに，部分的に信頼されるコードは，その信頼のレベルに応じた操作だけを実行できます。

コード本体に適用される信頼のレベルは，実行時にセキュリティポリシーエンジンに提供されるさまざまな証拠に依存します。証拠は，アセンブリ単位で提供されます。証拠には，さまざまな種類があります。アセンブリのソースとそのソースが所属する「ゾーン」（Internet Explorer と同じ）のようにホストの CLR 環境から提供される証拠もあれば，公開キートークンのようにアセンブリ自体から提供される証拠もあります。アセンブリに関連付けられた証拠に基づき，アセンブリは，ポリシーエンジンによってコードグループに割り当てられます。各コードグループは，帰属条件（「アセンブリはイントラネットゾーンに存在する必要がある」など）と，関連付けられた一連のアクセス許可を持つことができます。

部分的に信頼されるアプリケーションの概要については，次のマニュアルを参照してください。

- コードアクセスセキュリティの概要が記載された MSDN Web サイトの .NET マニュアル（http://msdn.microsoft.com/library/en-us/cpguide/html/cpconcodeaccesssecurity.asp）
- 部分的に信頼されるコードと部分的に信頼される環境に関する記事（http://msdn.microsoft.com/msdnmag/issues/02/06/Rich/default.aspx）

部分的に信頼される環境での VisiBroker for .NET の使用

部分的に信頼される環境で VisiBroker for .NET を使用するには，VisiBroker for .NET をローカルにインストールする必要があります。または，VisiBroker for .NET の DLL に完全な信頼を付与するなどのセキュリティポリシーを設定することもできます。これは，

VisiBroker for .NET が，部分的に信頼されるアセンブリ（PTA）からは直接使用できない他のアセンブリ（Visual J# など）を使用しているためです。

メモ　部分的に信頼されるコードからアセンブリを使用するには，アセンブリに AllowPartiallyTrustedCallersAttribute（APTCA）が適用されている必要があります。

VisiBroker for .NET のパブリックアセンブリは APTCA でマークされており，部分的に信頼される呼び出し元からロードして呼び出すことができます。ただし，VisiBroker for .NET の機能を実行するには，それ自体の完全な信頼が必要です。

VisiBroker for .NET がローカルにインストールされると，部分的に信頼されるアプリケーションが VisiBroker for .NET をロードし，そのメソッドを実行できるようになります。ただし，VisiBroker for .NET を使用してリモートサーバーと通信するには，ローカルセキュリティポリシーを変更して，部分的に信頼されるアセンブリに適切なアクセス許可を付与する必要があります。

たとえば，部分的に信頼されるアセンブリはネットワークにアクセスできません。特に，サーバーに自由にソケットを開くことはできません。例外は，「ローカルイントラネット」ゾーンからロードされたアセンブリだけです。このようなアセンブリは，自分のダウンロードに使用されたプロトコル（またはより安全なプロトコル）を使用して，ダウンロード元のサイトに逆に接続できます。通常，これは，http（最も一般的な例）を使用してダウンロードされたアセンブリがダウンロード元のホストに http（または https）でアクセスし直せることを意味します。

ただし，VisiBroker for .NET を使用する場合，アセンブリは IIOP（と TCP/UDP ソケット）を使用して通信し，デフォルトのセキュリティポリシーでは PAT にソケットを使用するアクセス許可が付与されません。このようなアクセス許可はセキュリティホールの原因になるため，VisiBroker for .NET はアサート使用されません。VisiBroker for .NET がこのようなアクセス許可をアサートすると，すべての部分的に信頼されるアプリケーションがダウンロード元のリモートホストにアクセスできるようになるため，セキュリティの面から推奨されません。

VisiBroker for .NET クライアントがリモートに通信できるようにするには，それらのアプリケーションがソケットアクセス許可を持つようにします。特定のアプリケーション（PTA）が特定のホストに接続するためのソケットアクセス許可を持つかどうかは，ユーザー側の必要に依存します。ユーザーは，アプリケーションに適切なアクセス許可が付与されるように，アプリケーションを実行するマシンのコードアクセスセキュリティポリシーを設定する必要があります。Microsoft のセキュリティフレームワークには，いくつかの選択肢が用意されています。さらに，カスタムコードグループを使用してセキュリティポリシーを調整することもできます。

VisiBroker for .NET に必要なアクセス許可

VisiBroker for .NET を正しく実行するには，次のアクセス許可を持つ部分的に信頼されるアプリケーションコードが必要です。アプリケーションの実際に動作内容によっては，他のアクセス許可が必要になる場合もあります。たとえば，ウィンドウを起動するには，UI アクセス許可が必要です。

`SecurityPermission.Execute`：このアクセス許可は，VisiBroker for .NET 自体には必要ありませんが，部分的に信頼されるアセンブリをロードして実行するために必要です。

`DnsPermission.Unrestricted`：DNS 名を解決するための機能です。VisiBroker for .NET は，このアクセス許可をアサートしません。かわりに，VisiBroker for .NET を使用して有効なホストを探します。PTA には DNS を解決する機能が必要です。

`SocketPermission`：VisiBroker for .NET を使用するコードは，常にリモートサーバーも呼び出しを行います。そのため，PTA には，適切なサーバーホストとポートにソケットを開く（コールバックの場合は接続する）ためのアクセス許可が必要です。VisiBroker for .NET は，同じホストであっても逆に接続するアクセス許可をアサートしません。そうすると，信頼されない任意のコードがホストに接続し直すことができるからです。アプリケーションにソケットアクセス許可を付与する場合は，VisiBroker for .NET コードがアクセ

スしているリモートサーバーにクライアントが接続できるようにする必要があります。このサーバーは，アプリケーションアセンブリを提供しているホストと同じでない場合があります。

セキュリティポリシーの設定方法の詳細は，セキュリティポリシー設定と caspol ツールのマニュアルを参照してください。

ノータッチデプロイメント環境の使用

VisiBroker for .NET. を使用するには，クライアントマシンに適切なセキュリティポリシーを設定して，部分的に信頼されるコード（厳密な名前，サイト，URL などの適切な証拠に基づく）に適切なアクセス許可を与える必要があります。VisiBroker for .NET がインストールされ，このようなセキュリティポリシーが適用されたら，VisiBroker for .NET を使用するアプリケーションコードを，ノータッチデプロイメント技術を使用して配布できます。VisiBroker for .NET をインストールしない場合，VisiBroker for .NET が正しく機能するには，VisiBroker for .NET の DLL に完全な信頼を付与する必要があります。他の帰属条件の方が現在の環境に適していると考えられない限り，VisiBroker for .NET の DLL に完全な信頼を付与する際は，厳密な名前帰属条件を使用することをお勧めします。

第 15 章

VisiBroker for .NET と COM の使用

この章では，VisiBroker for .NET を使用して，COM ベースのクライアントアプリケーションから RMI，EJB，または CORBA で開発されたサーバー側コンポーネントにアクセスできるようにする開発技術について説明します。

理論上は，共通言語ランタイム（CLR）向けに開発された任意のオブジェクトを COM ベースのクライアントに公開できますが，実際には，特定の開発技術と配布技術を使用することで，そのようなアクセスが簡単になり，柔軟性が高まり，問題も発生しなくなります。この章では，このようなさまざまな技術と，VisiBroker for .NET を使用して COM アクセスを機能させる方法について説明します。

管理オブジェクトを COM クライアントに公開する際に発生する最初の問題は，どのオブジェクトを公開するかを決定することです。一般には，ビジネスロジックへのアクセスを提供する型は公開し，ミドルウェアインフラストラクチャを提供するだけの型は隠蔽することが適切です。このガイドラインにしたがえば，コンポーネントインターフェースを COM から見えるようにし，マーシャリングスタブを見えないようにする必要があります。

-COM フラグを付けて VisiBroker for .NET コンパイラを実行すると，次のように，すべてのパブリック型の宣言に ComVisible 属性が追加されます。

```
[System.Runtime.InteropServices.ComVisible(value)]
```

value は，COM クライアントに公開される型では true，それ以外の型では false です。

次の生成される型は，COM クライアントから可視になります。

すべてのリモートインターフェース

- IDL で定義されるインターフェース
- RMI を使用して Java で定義されるインターフェース
- EJB を使用して Java で定義されるインターフェース

IDL で定義される次のデータ型

- struct
- union
- enum
- valuetype

Java で定義される次のデータ型

- パブリッククラス（java.lang.Throwable の直接的または間接的な拡張クラスを除く）
- パブリックインターフェース

次の生成される型は，COM クライアントから不可視になります。

IDL で定義される次のデータ型

- 例外
- 定数
- 値ボックス

Java で定義される次のデータ型

- java.lang.Throwable を直接的にまたは間接的に拡張するクラス

他のすべての生成されるクラスとインターフェース

- Helper クラス
- ValueFactory クラスと ValueData クラス
- Operations インターフェース
- MarshalingStub クラスと LocalStub クラス
- RemotingProxy クラス
- POA クラスと POATie クラス

COM 可視レベルの上書き

コンパイラによって設定されるデフォルトの COM 可視レベルは多くのアプリケーションに適していますが，型によっては COM 可視レベルの調整が必要になる場合があります。すべての ComVisible 宣言には，可視レベル値（true または false）の直前に完全なスコープ付きの型名が含まれます。各型の可視レベルの変更は，正規表現で簡潔に記述できます。VisiBroker for .NET コンパイラは，型データメンバーの COM 可視レベル属性を生成しません。プログラマがこれを調整して，クラスデータメンバーの可視レベルを制御する必要があります。型の static メンバーと const メンバーが COM 可視でないことにも注意してください。

ClassInterface の属性

デフォルトでは，次の型は ClassInterfaceType.AutoDual インターフェースクラスが必要であるとマークされます。

IDL で定義される次のデータ型

- struct
- union
- valuetype

Java で定義される次のデータ型

- パブリッククラス（java.lang.Throwable の直接的または間接的な拡張クラスを除く）

COM では，AutoDual インターフェースクラスが特定のクラスから提供されるすべてのパブリックメソッドおよびプロパティへの型アクセスをアーリーバインディング COM クライアントに提供します。ただし，AutoDual インターフェースクラスは，COM クライアントを脆弱にするおそれがあるため，注意して使用する必要があります。したがって，AutoDual であるクラスが変更された場合は，そのクラスを使用するすべてのアーリーバインディング COM クライアントを再コンパイル（Visual Basic などのインタープリタ言

語を使用する場合は再定義）する必要があります。そのため，基底の型が不変の場合にのみ AutoDual クラスインターフェースを使用することを強くお勧めします。新しいバージョンの型は，既存の COM クライアントを破壊しません。

ただし，COM で AutoDual を使用するには，上に示した IDL または Java で定義される型に対しても同様の不変の要件があります。つまり，struct，union，または valuetype の IDL 定義が変更されると（または，Java で定義されたクラスのシリアル化可能なデータフィールドを変更されると），IIOP マーシャリングエラーが発生します。

簡単に言えば，AutoDual 型のメソッドとフィールドレイアウトを変更不可にする必要があるのと同様に，IDL 型と Java 型のフィールドレイアウト（マーシャル後のフォーマットを決定する）も変更不可にする必要があります。そのため，VisiBroker for .NET コンパイラは，デフォルトの動作として，上に示した型を AutoDual としてマークします。

COM 可視性と同様に，特定の型（またはすべての型）の AutoDual のマークは，必要であれば調整できます。

メモ この動作は，将来変更される可能性があります。VisiBroker for .NET コンパイラは，生成されるインターフェースに `ClassInterface.None` と `ComInterfaceType.InterfaceIsDual` を追加する可能性があります。

カスタムインターフェースの定義

Microsoft は，実装クラスが後で変更される可能性がある場合，実装クラスで `ClassInterfaceType.AutoDual` を使用するかわりに，より堅実な方法としてユーザー定義インターフェースを使用することを推奨しています。また，実装クラスを `ClassInterfaceType.None` ClassInterface 属性でマークして，`_<impl class>` インターフェースが生成されないようにすることも推奨しています（そうでなければ，これが [default] インターフェースになる）。ユーザー定義インターフェースはインスペクト可能で，デュアルインターフェース COM サーバーを生成する必要がある場合は適切に `ComInterfaceType.InterfaceIsDual` InterfaceType 属性でマークできます。

現時点では，VisiBroker for .NET コンパイラは，明示的にはクラスを `ClassInterfaceType.None` でマークしません。また，インターフェースも `InterfaceIsDual` でマークされません。VisiBroker for .NET コンパイラは，Java ベースまたは生成されるパブリック実装クラスに対して AutoDual フラグを生成します。

さまざまな技術を使用できますが，次の Java クラス定義を使用して例を挙げます。

```
public class Quote implements java.io.Serializable {
  private String symbol;
  private float price;
  public Quote(String symbol, float price) {
    this.symbol = symbol;
    this.price = price;
  }
  public String getSymbol() {
    return symbol;
  }
  public float getPrice() {
    return price;
  }
}
```

この Java クラスは株価情報を表し，銘柄コード（symbol）と株価（price）に対応するデータフィールドを保持しています。このクラスでは，ユーザーが Quote の symbol と price にアクセスできますが，その値を変更することはできません。C# 型でも，symbol フィールドと price フィールドを同様に読み取り専用にすることが理想です。

デフォルトでは，このクラスに対して VisiBroker for .NET java2cs コンパイラを実行すると
(-COM フラグを有効にして)，次の C# クラスが生成されます。

```
[Serializable]
[ComVisible(true)]
[ClassInterface(ClassInterfaceType.AutoDual)]
public class Quote  {
  public Quote() {
  }
  public Quote(float Price, string Symbol) {
    this._Price = Price;
    this._Symbol = Symbol;
  }
  private float _Price;
  public virtual float Price {
    get { return this._Price; }
    set { this._Price = value; }
  }
  private string _Symbol;
  public virtual string Symbol {
    get { return this._Symbol; }
    set { this._Symbol = value; }
  }
}
```

このクラスには，次の短所があります。

1. AutoDual モードを使用しているため，大部分のアーリーバインディングクライアントの再コンパイルが必要になるという理由から，クラスが不変になります。
2. Symbol プロパティと Price プロパティは，パブリックの getter と setter を持ちます。これは，Symbol と Price を読み取り専用にするという設計方針に矛盾します。

そのため，この生成されるクラスを使用するかわりに，ユーザー定義インターフェースを使用します。次は，Quote を表す C# インターフェースです。

```
[System.Runtime.InteropServices.ComVisible(true)]
public interface Quote {
  string GetSymbol();
  float GetPrice();
}
```

このクラスは，COM 可視で，Symbol と Price の getter メソッドを持ちます。このインターフェースを適切な ComInterfaceType 属性でマークし，必要であればデュアルインターフェース，ディスパッチインターフェース，または IUnknown を指定します。

次の手順では，独自の実装を提供するため，Quote インターフェースを生成しないように VisiBroker for .NET コンパイラに指示します。それには，次のヒントを含むヒントファイルを使用します。

```
<?xml version="1.0"?>
<hints>
  <hint>
    <java-class>Quote</java-class>
    <cs-sig-type>Quote</cs-sig-type>
    <cs-impl-type>QuoteImpl</cs-impl-type>
    <mode>automatic</mode>
  </hint>
```

```
</hints>
```

このヒントは，Java 型 Quote がシグニチャ型 Quote と実装型 QuoteImpl の 2 つの C# 型にマップされることを示します。また，automatic コード生成モードを使用することを指定します。実際には，
automatic はデフォルトのコード生成モードなので，<mode/> 要素は省略できます。

XML 要素 <cs-sig-type/> は，クライアントが Quote とやり取りするときに使用される型名を示します。XML 要素 <cs-impl-type/> は，Quote を実装するために使用されます (QuoteImpl など)。

public の Quote 型と internal の QuoteImpl 型の両方の実装を提供する必要があります。Quote インターフェースは前に記載しました。次は QuoteImpl のコードです。

```
internal class QuoteImpl : Quote {
  internal string Symbol;
  internal float Price;
  public string GetSymbol() {
    return Symbol;
  }
  public float GetPrice() {
    return Price;
  }
}
```

この実装クラスについて，いくつか注意点があります。

- このクラスは internal なので，COM 可視性を示す必要はありません。COM 可視にできるのは public 型だけです。この実装クラスは，COM クライアントから不可視です（意図したとおり）。
- ヒントファイルで指定した automatic コード生成モードでは，このクラスが Java クラスのシリアル化可能なフィールドに対応するフィールドを持つ必要があります。Java クラスには 2 つのシリアル化可能なフィールド（symbol と price）があるため，C# の実装クラスにも対応する 2 つのフィールド（Symbol と Price）があります。もちろん，必要であれば，これらのフィールドをプロパティとして実装することもできます。
- QuoteImpl の 2 つのシリアル化可能なフィールド（Symbol と Price）は，QuoteImpl のマーシャリング時に，生成されるクラス QuoteValueFactory によって読み書きされるため，internal（または public）としてマークする必要があります。これらのフィールドを private または protected にすることはできません。

シリアル化可能なフィールドの重複を避けるようにする場合は，別の技術を使用して QuoteImpl クラスを実装できます。ここでは，生成されるクラス QuoteValueData を拡張して QuoteImpl を実装します。

```
internal class QuoteImpl : QuoteValueData, Quote {
  public string GetSymbol() {
    return Symbol;
  }
  public float GetPrice() {
    return Price;
  }
}
```

このクラスでは，Symbol フィールドと Price フィールドがベースクラス QuoteValueData から継承されるため，これらのフィールドを宣言しません。

パラメータと戻り値の配列のサポート

既知の問題として，管理コードで実装される型のメソッドを COM クライアントから呼び出す際に，メソッドのパラメータまたは戻り値に配列型が含まれる場合があります。

これらの問題に対応するために，-COM フラグが指定されると，VisiBroker for .NET コンパイラは，このような問題を含むメソッドに対してそれぞれ「オーバーロード」メソッドを生成します。

例として次のメソッドを考えます。

```
int[] GetLengths(string[] strings);
```

このメソッドは，string の配列をパラメータとして受け取り，integer の配列を返します。戻り値の各要素は，対応する入力文字列の長さを示します。このメソッドが次のように呼び出されたとします。

```
string[] strings = { "VisiBroker", "Rocks" };
int[] lengths = o.GetLengths(strings);
```

結果は，要素に 10 と 5 を含む配列になります。

ただし，この C# シグニチャを COM にエキスポートすると，一部の COM クライアントはメソッド GetLengths を呼び出せなくなります。たとえば，Excel スプレットシード内の次の Visual Basic コードを呼び出すとします。

```
Dim strings(1) As String
strings(0) = "VisiBroker"
strings(1) = "Rocks"
lengths = o.GetLengths(strings)
```

次のエラーが通知されます。

> コンパイルエラー : 関数またはインターフェースが予約されているか，または Visual Basic でサポートされていないオートメーションタイプが関数で使用されています。

この問題に対応するために，TVisiBroker for .NET コンパイラは，次のシグニチャを持つ「オーバーロード」メソッドを出力します。

```
object GetLengthsForCom(object strings);
```

このメソッドシグニチャは，すべての配列値のパラメータと戻り値を型 object に置き換えます。ただし，元のメソッド名にサフィックス ForCom が追加されているため，このメソッドは，技術的には元のメソッド GetLengths のオーバーロードではありません。C# では，戻り値の型に基づいてオーバーロードされたメソッドシグニチャを使用できないため，真の意味のオーバーロードは使用できません。

生成されたメソッドは，Visual Basic クライアントで次のように使用できます。

```
Dim strings(1) As String
strings(0) = "VisiBroker"
strings(1) = "Rocks"
lengths = o.GetLengthsForCom(strings)
```

これにより，32 ビット整数の配列である lengths 値が取得され，この配列要素には，期待される 10 と 5 の値が含まれます。

ProgId の競合の回避

Microsoft の COM 相互運用マニュアルには，非常に長い型名を COM クライアントにエキスポートしようとするとエラーが発生する可能性があることが記載されています。特に，C# の型名が 39 文字を超えると，COM クライアントはその型にほぼアクセスできなくなります。Microsoft は，長い型名があいまいになる場合は，注釈として `ProgId` を追加することを推奨しています。簡単な回避策は，正規表現に基づくツールを使用して，VisiBroker for .NET コンパイラによって生成されるコードを変更することです。

第 16 章

VisiBroker for .NET と Borland GateKeeper の使用

この章では, VisiBroker for .NET アプリケーションで VisiBroker GateKeeper サービスを使用するためのプロパティを設定する方法について説明します。GateKeeper の使用方法については,『VisiBroker GateKeeper ガイド』を参照してください。

Gatekeeper の概要

Borland VisiBroker GateKeeper は, CORBA General Inter-ORB Protocol (GIOP) 準拠の GIOP プロキシサーバーです。Borland VisiBroker GateKeeper は, CORBA クライアントと CORBA サーバーがインターネットブラウザ，ファイアウォール，および Java サンドボックスセキュリティに基づくセキュリティ制約の下でネットワークを介して通信できるようにします。つまり, Gatekeeper は, セキュリティ制約によってクライアントがサーバーと直接通信できない場合に，クライアントやサーバーのゲートウェイまたはプロキシとして機能します。

サーバーがクライアントに直接公開されていない場合，またはクライアントからサーバーへのアクセスが制約されている場合に，Gatekeeper はよく使用されます。後者の場合は，クライアントが署名なしのアプレットであるか，間にファイアウォールがあるかのどちらかです。

VisiBroker for .NET ファイアウォール機能の有効化

VisiBroker for .NET は, CORBA 2.6 準拠のファイアウォールをサポートしています。デフォルトでは，VisiBroker for .NET のファイアウォール機能は無効になっています。VisiBroker GateKeeper サービスを使用する VisiBroker for .NET アプリケーションを開発する場合は, `janeva.firewall` プロパティを使用して, ファイアウォール機能を明示的に有効にする必要があります。

VisiBroker for .NET サーバー側の設定

GateKeeper を介してクライアントとサーバーが通信できるようにするには，サーバーがクライアントにファイアウォールパスをエキスポートする必要があります。それには，いくつかのプロパティを設定します。次の表に，サーバー側の設定に固有のプロパティを示します。

表 16.1 サーバー側の GateKeeper プロパティ

プロパティ	有効な値	説明
vbroker.orb.exportFirewallPath	true false (default)	ファイアウォールパスをサーバーの IOR プロファイルコンポーネントに埋め込む場合は，このプロパティを true に設定します。 vbroker.orb exportFirewallPath=true
vbroker.se.iiop_tp.firewallPaths	< 空 > (デフォルト) <paths>	このプロパティを使用して，すべてのファイアウォールパスを宣言します。< パス > は，クライアントからサーバーへの通信パスに対するユーザー定義の名前です。コンマで区切って指定します。 vbroker.se.iiop_tp firewallPaths=x,y
vbroker.firewall-path.<pathname>	<empty> (default) < コンポーネント >	ファイアウォールパス <パス名> 内のコンポーネントを指定します。 vbroker.firewall-path.x=a,b vbroker.firewall-path.y=c
vbroker.firewall.< コンポーネント >.type	<empty> PROXY TCP	コンポーネントの型を指定します。 vbroker.firewall.a.type = PROXY vbroker.firewall.b.type = TCP
vbroker.firewall.<component>.ior	<empty> <ior_filename> <ior_URL> IOR:<stringified_ior>	コンポーネントの IOR を指定します。これは，vbroker.firewall.< コンポーネント >.type=PROXY とともに指定します。 file:C:/GateKeeper/GateKeeper.ior http://www.inprise.com/GK GateKeeper.ior IOR:239840284172907342349723423 4234
vbroker.firewall.< コンポーネント >.host	<empty> <fake host name>	コンポーネントサーバーの偽ホスト名を指定します。これは，vbroker.firewall.< コンポーネント >.type=TCP とともに指定し，コンポーネントは NAT 使用の TCP ファイアウォールです。
vbroker.firewall.< コンポーネント >.iiop_port	<empty> <fake IIOP Port>	コンポーネントサーバーの偽 IIOP ポートを指定します。これは，vbroker.firewall.< コンポーネント >.type=TCP とともに指定し，コンポーネントは NAT 使用の TCP ファイアウォールです。
vbroker.firewall.< コンポーネント >.ssl_port	<empty> (default) <fake SSL Port>	コンポーネントサーバーの偽 SSL ポートを指定します。これは，vbroker.firewall.< コンポーネント >.type=TCP とともに指定し，コンポーネントは NAT 使用の TCP ファイアウォールです。

表 16.1 サーバー側の GateKeeper プロパティ（続き）

プロパティ	有効な値	説明
vbroker.firewall.< コンポーネント >.hiop_port	<empty> (default) <fake HIOP Port>	コンポーネントサーバーの偽 HIOP ポートを指定します。これは，vbroker.firewall.< コンポーネント >.type=TCP とともに指定し，コンポーネントは NAT 使用の TCP ファイアウォールです。
vbroker.orb.enableBiDir	client server both none（デフォルト）	クライアントが vbroker.orb.enableBiDir=client と定義し，サーバーが vbroker.orb.enableBiDir=server と定義している場合は，Gatekeeper の vbroker.orb.enableBiDir の値によって接続状態が決定されます。 vbroker.se.exterior.scm.ex--iiop.manager.importBiDir プロパティを true に設定すると，Gatekeeper はクライアントからの双方向接続を受け付けます。 vbroker.se.exterior.scm.ex--iiop.manager.exportBiDir プロパティを true に設定すると，Gatekeeper はサーバーとの双方向接続を要求します。

VisiBroker for .NET クライアント側の設定

次の表に，クライアント側の設定に固有のプロパティを示します。

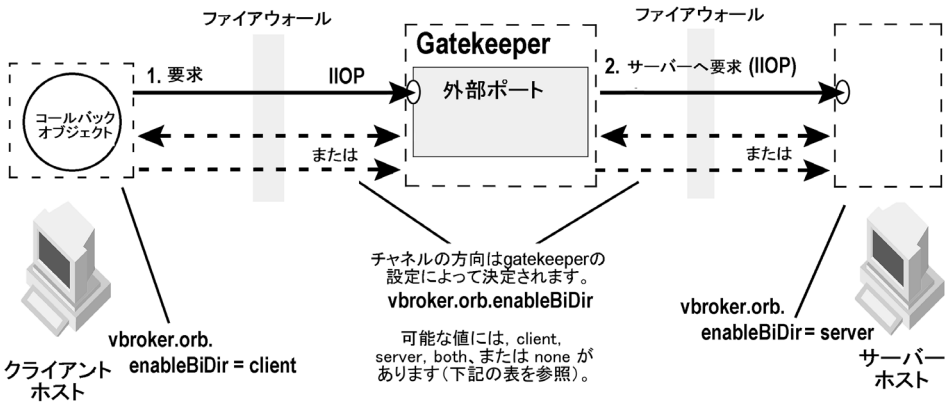
表 16.2 クライアント側の GateKeeper プロパティ

プロパティ	有効な値	説明
vbroker.orb.gatekeeper.ior	<empty> (default) <ior_filename>	Gatekeeper IOR ファイルの URL を指定します。
vbroker.orb.alwaysProxy	false (default) true	クライアントがサーバーに接続するときに，必ず別の Gatekeeper を経由するかどうかを指定します。
vbroker.locator.ior.ior	<empty> (default) <ior_filename>	Gatekeeper ロケータ IOR ファイルの URL を指定します。Gatekeeper が提供するロケーションサービスには制限があります。ロケーション要求を別の Gatekeeper に転送することはできません。
vbroker.orb.alwaysTunnel	false (default) true	クライアントがサーバーに接続するときに，必ず HTTP トンネル（IIOP ラッパー）を使用するかどうかを指定します。
vbroker.orb.enableBiDir	client server both none（デフォルト）	双方向接続を選択的に構築できます。クライアントが vbroker.orb.enableBiDir=client と定義し，サーバーが vbroker.orb.enableBiDir=server と定義している場合は，Gatekeeper の vbroker.orb.enableBiDir の値によって接続状態が決定されます。 たとえば，vbroker.se.exterior.scm.ex--iiop.manager.importBiDir プロパティを true に設定すると，Gatekeeper はクライアントからの双方向接続を受け付けます。vbroker.se.exterior.scm.ex--iiop.manager.exportBiDir プロパティを true に設定すると，Gatekeeper はサーバーとの双方向接続を要求します。

Gatekeeper の双方向サポートによるコールバック

双方向 IIOP では，サーバーは，クライアントが開始した接続を使用してクライアントに非同期情報を戻します。サーバーがクライアントとの接続を開始する必要はありません。

図 16.1 Gatekeeper の双方向サポートによるコールバック



上の図で，Gatekeeper はクライアントとサーバーの間に置かれ，クライアントに対してはサーバーとして，サーバーに対してはクライアントとして機能します。クライアント／Gatekeeper および Gatekeeper ／サーバーの通信チャネルは，一方向接続と双方向接続のどちらにも設定できます。

また，これらのチャネルを一方向または双方向に選択的に設定することもできます。次の表に，クライアントが vbroker.orb.enableBiDir=client と定義し，サーバーが vbroker.orb.enableBiDir=server と定義している場合に，GateKeeper の vbroker.orb.enableBiDir の値に応じてチャネルのタイプがどのように変わるかをまとめます。

表 16.3 一方向または双方向の通信

vbroker.orb.enableBiDir=	クライアントと Gatekeeper 間	Gatekeeper とサーバー間
client	一方向	双方向
server	双方向	一方向
both	双方向	双方向
なし	一方向	一方向

セキュリティに関する考慮事項

双方向 IIOP を使用すると，セキュリティに関する重大な問題が発生する可能性があります。セキュリティメカニズムが設定されていないと，悪意のあるクライアントがホストとポートを任意に選択して，双方向接続を要求する可能性があります。特に，自分のホスト側にないセキュリティ上重要なオブジェクトのホストとポートを指定することがあります。また，セキュリティメカニズムが設定されていないと，着信接続を受け付けたサーバーは，接続を開始したクライアントの ID を識別したり，クライアントの完全性を検査できません。さらに，サーバーが双方向接続を介して他のオブジェクトにアクセスできる可能性があります。クライアントの完全性に疑問がある場合は，双方向 IIOP を使用しないでください。セキュリティ上の理由から，VisiBroker for .NET を実行するサーバーは，双方向 IIOP を使用するように明示的に設定されていない限り，双方向 IIOP を使用しません。

例

次の例は，クライアント側の設定を示しています。クライアントは，常に GateKeeper をプロキシとして使用してサーバーと通信します。

```
<configuration>
   <configSections>
      <section name="visinet" type="Janeva.Settings, Borland.Janeva.Runtime"/>
   </configSections>
   <visinet>
      ...
      <firewall enabled="true"/>
      <vbroker vbroker.orb.alwaysProxy="true"/>
   </visinet>
</configuration>
```

次の例は，サーバー側の設定を示しています。これは，“proxy” という名前のノードを含む “internet” という名前のファイアウォールパスを定義します。このノードのタイプは PROXY です。

```
<configuration>
   <configSections>
      <section name="visinet" type="Janeva.Settings, Borland.Janeva.Runtime"/>
   </configSections>
   <visinet>
      ...
      <firewall enabled="true"/>
      <server defaultPort="10000">
         <remoting enabled="false"/>
      </server>
      <vbroker vbroker.orb.exportFirewallPath="true"
            vbroker.se.iiop_tp.firewallPaths=”internet”
            vbroker.firewall-path.internet=”proxy”
            vbroker.firewall.proxy.type=”PROXY”
            vbroker.firewall.proxy.ior=”http://localhost:9091/gatekeeper.ior”/>
   </visinet>
</configuration>
```

付録 A

コンパイラオプション

この章では VisiBroker for .NET コンパイラで使用できるオプションについて説明します。

オプションは左から右の順に処理され，最後の値が優先されます。デフォルトでは，リストのすべてのオプションが有効であり，オプションの前にはハイフン (-) が付きます。一部のオプションについては，-[no_] を使用するか，ハイフンを除去することで，デフォルト値を無効にできます。たとえば，#pragma が認識されない場合に警告を表示するためのデフォルト値は以下のとおりです。

```
warn_unrecognized_pragmas
```

デフォルト値をオフにするには，次のコマンドを入力します。

```
-no_warn_unrecognized_pragmas
```

idl2cs[j]

idl2cs ツールは IDL ソースファイルをコンパイルして，IDL 宣言用の C# マッピングを含むディレクトリ構造を作成します。idl2cs および idl2csj ツールは，次の点を除けば同一のツールです。つまり，idl2csj が Java 専用の環境で稼動するのに対して（UNIX およびより旧式の Windows マシンを含め，.NET のないプラットフォームでもコンパイラが稼動できます），idl2cs は C# 専用（.NET Framework）の環境で稼動します。

1 つの IDL ファイルが 1 つの C# ファイルにマップされます。Foo.idl 用のデフォルトの出力ファイル名は Foo.cs です。出力ファイルは，-o オプションを使用して指定できます。一般に IDL ファイル名は .idl 拡張子で終わる必要があります。

構文

```
idl2cs [options] {source_file}
```

例

```
idl2cs -no_Object_method Example.idl
```

オプション	説明
-D, -define foo[=bar]	プリプロセッサマクロ foo を定義します。オプションで bar の値を指定します。このオプションは，複数回使用できます。
-I, -include <dir>	# インクルードファイルのディレクトリへのフルパスまたは相対パスを指定します。インクルードファイルの検索に使用されます。このオプションは，複数回使用できます。

オプション	説明
-P, -no_line_directives	生成するコードでの行番号情報の生成を抑制します。デフォルトはオフです。
-H, -list_includes	標準エラー出力にインクルードファイルのフルパスを出力します。デフォルトはオフです。
-C, -retain_comments	プリプロセッサ出力で IDL ファイルのコメントを保持します。デフォルトはオフです。
-U, -undefine foo	プリプロセッサマクロ *foo* の定義を解除します。
-[no_]idl_strict	idl ソースの OMG 標準解釈に厳密にしたがうように指定します。デフォルトはオフです。
-[no_]builtin (TypeCode|Principal)	組み込み型 ::TypeCode または ::Principal を作成します。デフォルトはオンです。
-[no_]warn_unrecognized_pragmas	#pragma が認識されない場合に警告を表示します。デフォルトはオンです。
-[no_]back_compat_mapping	VisiBroker 3.x と互換性があるマッピングを使用します。デフォルトはオフです。
-[no_]preprocess	解析前に入力ファイルをプリプロセスします。デフォルトはオンです。
-[no_]preprocess_only	プリプロセス後に入力ファイルの解析を停止します。デフォルトはオフです。
-[no_]warn_all	すべての警告を同時にオンまたはオフにします。デフォルトはオフです。
-[no_]case_sensitive	識別子の大文字小文字を区別します。デフォルトはオンです。
-[no_]comments	コードのコメントの生成を抑制します。デフォルトはオンです。
-gen_included_files	# インクルードファイル用のコードを生成します。デフォルトはオフです。
-list_files	コード生成中に書き込まれたファイルをリストします。デフォルトはオフです。
-root_dir <path>	生成されるファイルが位置するディレクトリを指定します。
-[no_]servant	サーバント（サーバー側）コードを生成します。デフォルトはオンです。
-[no_]tie	タイクラスを生成します。デフォルトはオンです。
-[no_]warn_missing_define	前方参照宣言されたインターフェースが未定義の場合は警告を出します。デフォルトはオンです。
-[no_]strict_reverse_mapping	厳密な Java からの逆マッピングを使用します。デフォルトはオフです。
-o <file>	出力ファイルの名前を指定します。stdout の場合は，「-」を指定します。
-[no_]bind	bind() コードを生成します。デフォルトはオフです。
-idl2namespace <IDL name> <ns>	指定した IDL コンテナ型のデフォルト名前空間をオーバーライドします。
-[no_]Object_method	Object に関するメソッドをすべて生成します。デフォルトはオンです。
-namespace <ns>	生成されるコードのルート名前空間を指定します。
-map_keyword <kwd> <replacement>	回避するキーワードを指定し，その置換を指示します。
-[no_]mixed_caps	メソッドを MixedCaps に，メンバーを mixedCaps に変換します。デフォルトはオンです。
-[no_]examples	サンプル実装を生成します。デフォルトはオフです。
-hint_file <file_uri>	カスタム型マッピング用のヒントファイル URI を指定します。idl2csj の場合にのみ利用できます。
-[no_]remoting_proxy	.NET リモート処理とともに使用されるプロキシクラスを生成します。デフォルトはオンです。
-h, -help, -usage, -?	オプションのヘルプ情報を表示します。
-version	VisiBroker for .NET ソフトウェアのバージョン番号を表示します。
file1 [file2] ...	処理するファイルを 1 つ以上指定します。 または，標準入力の場合は「_」を指定します。

java2cs

このコマンドは Java クラスから C# コードを生成します。java2cs は Java RMI で定義されたリモートインターフェースを対応する C# に変換します。リモートインターフェース，EJB インターフェース，および値クラスが C# に変換されます。また，java2cs は入力型によって直接または間接的に参照される型も変換することに注意してください。

入力として複数の Java クラス名（Java バイトコード）を使用できます。複数のクラス名を入力する場合，クラス名の間に必ずブランクを配置してください。完全なスコープ付きクラス名を使用します。さらに入力として EJB JAR または EAR または任意のライブラリ JAR を指定できます。

メモ java2cs コンパイラは，CORBA インターフェースのオーバーロードされたメソッドをサポートしません。

メモ このコンパイラを使用するには，JDK 1.4 以降をサポートする Java 仮想マシンが必要です。

一部の Java リモートインターフェース定義で org.omg.CORBA.IDLEntity を拡張するクラスを使用する場合，次の項目が必要です。

- この型用の IDL 定義を含む IDL ファイル。これは，org.omg.CORBA.IDLEntity インターフェースが Java にマップされるすべての IDL データ型をマークするシグニチャインターフェースだからです。
- オブジェクト管理グループ（OMG）制定の CORBA 2.4 IDL2Java 仕様に準拠した，すべての関連（サポート）クラス。

一部の Java リモートインターフェース定義で org.omg.CORBA.IDLEntity を拡張するクラスを使用する場合，java2cs ツールのコマンドラインで -import <IDL ファイル > ディレクティブを使用します。

詳細は，http://www.omg.org にある CORBA 2.4 IDL2Java 仕様を参照してください。

構文

```
java2cs [options] {input_class_name}
```

例

```
java2cs -no_tie Account Client Server
```

分散オブジェクトを使用するために既存の Java バイトコードを利用する場合，または IDL を記述しない場合は，java2cs を使用します。java2cs を使用すると，Java バイトコードから必要なコンテナクラス，クライアントスタブ，およびサーバースケルトンを生成できます。

オプション	説明
-D, define foo[=bar]	プリプロセッサマクロ foo を定義します。オプションで bar の値を指定します。
-I, -include <dir>	# インクルードファイルのディレクトリへのフルパスまたは相対パスを指定します。インクルードファイルの検索に使用されます。
-P, -no_line_directives	生成するコードでの行番号情報の生成を抑制します。デフォルトはオフです。
-H, -list_includes	標準エラー出力にインクルードファイルのフルパスを出力します。デフォルトはオフです。
-C, -retain_comments	C# コードの生成時に Java ファイルのコメントを保持します。保持しない場合は，コメントは C# コードに表示されません。デフォルトはオフです。
-U, -undefine foo	プリプロセッサマクロ foo の定義を解除します。
-[no_]idl_strict	IDL ソースの OMG 標準解釈に厳密にしたがうように指定します。デフォルトはオフです。
-[no_]builtin (Typecode|Principal)	組み込み型 ::TypeCode または ::Principal を作成します。デフォルトはオンです。
-[no_]warn_unrecognized_pragmas	#pragma が認識されない場合に警告を表示します。デフォルトはオンです。

オプション	説明
-[no_]back_compat_mapping	VisiBroker 3.x と互換性があるマッピングを使用します。デフォルトはオフです。
-[no_]preprocess	解析前に入力ファイルをプリプロセスします。デフォルトはオンです。
-[no_]preprocess_only	プリプロセス後に入力ファイルの解析を停止します。デフォルトはオフです。
-[no_]warn_all	すべての警告を同時にオンまたはオフにします。デフォルトはオフです。
-[no_]idlentity_array_mapping	boxedRMI の boxedIDL に IDLEntity の配列をマップします。デフォルトはオフです。
-exported <pkg>	エキスポートされるパッケージを指定します。
-[no_]export_all	すべてのパッケージをエキスポートします。デフォルトはオフです。
-import <IDL file name>	追加の IDL 定義をロードします。
-imported <pkg> <IDL file name>	インポートされるパッケージの名前を指定します。
-gen_hints <file-name>	テンプレートヒントファイルを生成します。デフォルトはオフです。
-show_ignored	ロードできないクラスに関する警告をすべて出力します。デフォルトはオフです。
-list_classes	コンパイルされたクラスを出力します。デフォルトはオフです。
-[no_]ignore <class>|<package>	クラス（またはパッケージ内のすべてのクラス）とそれに依存するすべてのクラスを無視します。
-[no_]case_sensitive	識別子の大文字小文字を区別します。デフォルトはオンです。
-[no_]comments	生成されたコードにコメントを配置します。デフォルトはオンです。
-gen_included_files	# インクルードファイル用のコードを生成します。デフォルトはオフです。
-list_files	コード生成中に書き込まれたファイルをリストします。デフォルトはオフです。
-root_dir <path>	生成されるファイルが位置するディレクトリを指定します。
-[no_]servant	サーバント（サーバー側）コードを生成します。デフォルトはオンです。
-[no_]tie	tie クラスを生成します。デフォルトはオンです。
-[no_]warn_missing_define	前方参照宣言された名前が未定義の場合は警告を出します。デフォルトはオンです。
-[no_]strict_reverse_mapping	厳密な Java からの逆マッピングを使用します。デフォルトはオフです。
-o <file>	出力ファイルの名前を指定します。stdout の場合は,「-」を指定します。
-[no_]bind	bind() コードを生成します。デフォルトはオフです。
-idl2namespace <IDL name> <ns>	指定した IDL コンテナ型のデフォルト名前空間をオーバーライドします。
-[no_]Object_method	string，equals など，java.lang.Object メソッドで定義されるすべてのメソッドを生成します。デフォルトはオンです。
-namespace <ns>	生成されるコードのルート名前空間を指定します。
-map_keyword <kwd> <replacement>	回避するキーワードを指定し，その置換を指示します。
-[no_]mixed_caps	メソッドを MixedCaps に，メンバーを mixedCaps に変換します。デフォルトはオンです。
-[no_]examples	サンプル実装を生成します。デフォルトはオフです。
-hint_file <file-uri>	カスタム型マッピング用のヒントファイル URI を指定します。
-[no_]remoting_proxy	.NET リモート処理とともに使用されるプロキシクラスを生成します。デフォルトはオンです。
-h, -help, -usage, -?	オプションのヘルプ情報を表示します。
-version	VisiBroker for .NET ソフトウェアのバージョン番号を表示します。

オプション	説明
[file.jar] [file.ear] ...	処理する J2EE アーカイブ（JAR または EAR）のリストをオプションで指定します。
[class1] [class2] ...	処理する Java クラスのオプションリスト。ターゲット Java クラスを指定しない場合は，指定した J2EE アーカイブから自動的に決定されます。

付録 B

IDL から C# へのマッピング

この章では idl2cs コード生成ツールによって生成される，VisiBroker for .NET IDL から C# 言語へのマッピングについて説明します。

名前

デフォルトでは，IDL 名および識別子は大文字小文字混在で C# 名および識別子にマップされます。これはオプションマッピングであり，コンパイラ指令 -[no_]mixed_caps（デフォルトは大文字小文字混在）によって制御されます。

デフォルトでは，メソッド，属性，およびファクトリメソッドは，名前の先頭が大文字で始まり，名前の各論理「ワード」も先頭が大文字で始まるように指定されます。このコンテキストでは，識別子の一部の片側が下線（_）によって区切られている場合，その部分が論理ワードとみなされます。たとえば IDL メソッド名 foo_bar は，生成される C# コードで FooBar にマップされます。

enum，struct のメンバーフィールド，例外，および valuetype は，小文字で始まる名前にマップされますが，後続の各論理「ワード」は先頭が大文字になります。たとえば foo_bar は fooBar になります。

すべてが大文字の名前は，生成される C# コードでは大文字小文字混在の名前には変換されず，間の下線も削除されないという例外があります。たとえば FOO_BAR はそのままです。

先行または後続の下線は，すべての変換名で保持されます。たとえば _foo_bar_ は _FooBar_ になります。

マップされた C# コードで C# キーワードとの名前コリジョンが生成された場合，名前コリジョンはマップされた名前の先頭にアットマーク（@）を付加することによって解決されます。@ プレフィクスが付くのは C# の場合です。たとえば，C# キーワード string は @string にマップされますが，.NET で解釈する場合，シンボルの実際の名前は，引き続き string です。別の .NET 言語で string がキーワードではない場合（たとえば，J#），シンボルは string として認識されます。

さらに，C# 言語の性質により，単一の IDL 構造体が複数の（名前の違う）C# 構造体にマップされる場合があります。追加の名前は，説明的なサフィックスを追加することによって構築されます。たとえば，IDL インターフェース AccountManager は，C# インターフェース AccountManager と，追加の C# クラス AccountManagerOperations および AccountManagerHelper にマップされます。

追加の名前が他のマップされた IDL 名と競合するような例外的な場合には，前述した解決ルールが他のマップされた IDL 名に適用されます。つまり，必要とされる「追加」名のネーミングおよび使用が優先されます。

たとえば fooHelper という名前の IDL インターフェースは，foo という名前のインターフェースが存在するかどうかに関係なく，C# クラス _fooHelper にマップされます。C# クラス _fooHelper のヘルパークラスは _fooHelperHelper と命名されます。

IDL 名が変更されずに C# 識別子に正常にマップされ，それが C# 予約語と競合した場合には，コリジョンルールが適用されます。

予約済み生成サフィックス

マッピングでは，クラスサフィックスとして使用するために，複数の名前の使用が予約されています。ユーザー定義 IDL 型またはインターフェースにこれらの名前のうちのいずれかを使用すると（これも正しい IDL 名であると仮定），マップされた名前の先頭に下線（_）が付加されます。予約済みの生成サフィックス名は次のとおりです。

- Helper - C# クラス <type>Helper。ここで <type> は IDL ユーザー定義型の名前です。
- NS - ネスト化されたスコープ C# 名前空間名 <interface>NS。ここで <interface> は IDL インターフェースの名前です。
- オペレーション
- POATie
- POA
- RemotingProxy
- ValueFactory
- ValueData - C# クラス <valuetype>ValueData および <valuetype>ValueFactory。ここで <valuetype> は IDL valuetype 型の名前です。

予約語

マッピングでは独自の目的のために複数の語の使用を予約しています。ユーザー定義 IDL 型またはインターフェースにこれらの語のうちのいずれかを使用すると（これも正しい IDL 名であると仮定），マップされた語の先頭にアットマーク（@）が付加されます。C# 言語で予約済みのキーワードは次のとおりです。

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
dontinue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	false
finally	float	fixed	for
foreach	goto	if	implicit
in	int	interface	internal
is	lock	long	namespace
new	null	object	operator
out	override	params	private

protected	public	readonly	ref
return	sbyte	sealed	short
sizeof	stackalloc	static	string
struct	switch	this	throw
true	try	typeof	uint
ulong	unchecked	unsafe	ushort
使用	virtual	void	volatile
while			

基本型

次の表に，定義済みの IDL 型がどのように基本 C# 型にマップされるかを示します。

表 16.4　基本型マッピング

IDL 型	C# 型
boolean	bool
char	char
wchar	char
octet	byte
string	string
wstring	string
short	short
unsigned short	short
long	int
unsigned long	int
longlong	long
unsigned longlong	long
float	float
double	double

IDL 型とそのマップされた C# 型の間に潜在的なミスマッチが存在する場合，標準 CORBA 例外が発生する可能性があります。大半の部分では，例外は 2 つのカテゴリに分けられます。

- C# 型の範囲は IDL 型より大きくなっています。たとえば C# 文字は IDL 文字のスーパーセットです。
- .NET では符号なしの型に対して統一されたサポートがないので，符号なしの IDL 型は C# の符号付きの型にマップされます。開発者は大きな符号なしの IDL 型値が .NET では負の整数として正しく処理されることを確認する必要があります。

詳細は，以降のセクションを参照してください。

C# ヌル

C# null は，ヌル CORBA オブジェクトリファレンスおよび valuetype を表現する際にのみ使用できます（再帰的 valuetype を含む）。たとえば，空の文字列を表すには null ではなくて，長さゼロの文字列を使用する必要があります。これは，配列および構築されたあらゆる型（valuetype を除く）にも当てはまります。構造体として null を渡そうとすると，システム NullReferenceException が発生します。

boolean

IDL 型 boolean は C# 型 bool にマップされます。IDL 定数 TRUE および FALSE は，C# 定数 true および false にマップされます。

char

IDL 文字は文字セットのエレメントを表す 8 ビットの値であるのに対して，C# 文字は Unicode 文字を表す 16 ビットの符号なしの値です。タイプセーフを有効にするため，CORBA ランタイムはメソッド呼び出し中にパラメータがマーシャルされた際に，IDL chars からマップされたすべての C# chars の範囲の妥当性をアサートします。char が文字セットによって定義された範囲外の場合は，CORBA::DATA_CONVERSION 例外が発生します。

IDL wchar は C# char 型にマップされます。

String および WString

IDL 型 string は，バウンドおよびアンバウンドされたバリアントの両方とも，C# 型 string にマップされます。文字列の文字範囲チェックおよび境界チェックは，マーシャル時に実行されます。

IDL 型 wstring は，Unicode 文字列の表現に使用され，C# 型 string にマップされます。文字列の境界チェックは，マーシャル時に実行されます。

整数型

IDL short および unsigned short は C# 型 short にマップされます。IDL long および unsigned long は C# 型 int にマップされます。

メモ .NET では符号なしの型に対して統一されたサポートがないので，符号なしの IDL 型は C# の符号付きの型にマップされます。開発者は，.NET での負の整数が，大きな符号なしの値として正しく処理されることを確認する必要があります。

IDL 型拡張

ここでは，VisiBroker for .NET による IDL 型拡張のサポートについてまとめます。最初の表では概要を簡単に示し，次の表では新しい型のサポートについて概説しています。

表 16.5 サポートされる IDL 拡張の要約

型	VisiBroker for .NET でのサポート
longlong	あり
unsigned longlong	あり
long double	いいえ[1]
wchar	はい[2]

表 16.5 サポートされる IDL 拡張の要約 （続き）

型	VisiBroker for .NET でのサポート
wstring	はい[2]
fixed	いいえ[1]

[1] VisiBroker for .NET では，OMG 標準実装の将来のリリースでサポートされる可能性があります。

[2] ネットワークでは Unicode が使用されます。

表 16.6 新しい型用の IDL 拡張

新しい型	説明
longlong	64 ビット符号付き 2 の補数の整数
unsigned longlong	64 ビット符号なし 2 の補数の整数
long double	IEEE 標準 754-1985 拡張倍精度浮動小数点
wchar	ワイド文字
wstring	ワイド文字列
fixed	固定小数点 10 進数演算（有効桁数 31 桁）

定数

定数は同じ名前の公開抽象クラスに定数としてマップされ，Value という名前の public const int フィールドが含まれます。このフィールドは定数の値を保持します。

このサンプルコードは，モジュール内の IDL 定数から C# クラスへのマッピングを示しています。

```
/* Example.idl より */
module Example {
const long aLongerOne = -123;
};

// Example.cs
namespace Example {
public abstract class ALongerOne {
   public const int Value = (int) -123;
   }
}
```

メモ インターフェース内の定数または valuetype は，インターフェースまたは valuetype の名前に NS サフィックスが付加されて名前空間に置かれます。

構造型

IDL 構造型には enum，struct，union，sequence，および array が含まれます。型 sequence および array は両方とも C# array 型にマップされます。IDL 構造型 enum，struct，および union は，IDL 型のセマンティクスを実装する C# クラスにマップされます。生成される C# クラスには，元の IDL 型と同じ名前が付けられます。

enum

IDL enum は enum 値を宣言する enum 型と同じ名前で C# enum にマップされます。次のサンプルコードは C# enum にマップされた IDL enum の例です。

```
// Example.idl
module Example {
   enum EnumType (first, second, third};
};
```

```
// Example.cs
public enum EnumType {
   first
   second
   third
}
```

struct

IDL struct は，IDL メンバーフィールド用のインスタンス変数およびすべての値用のコンストラクタを備えた C# クラスに，同じ名前でマップされます。

このサンプルコードは，IDL struct の C# へのマッピングを示しています。

```
// Example.idl
module Example {
   struct StructType {
      long field1;
      string field2;
   };
};
```

```
// Example.cs
public sealed class StructType
   public int field1;
   public string field2;
   public StructType() {
      field2 = "";
   }
   public StructType (int field1, string field2) {
      this.field1 = field1;
      this.field2 = field2;
   }
   override public string ToString() {
      System.Text.StringBuilder _ret =
         new System.Text.StringBuilder("struct Example.StructType {");
      _ret.Append("¥n");
      _ret.Append("int field1=");
      _ret.Append(field1);
      _ret.Append(",¥n");
      _ret.Append("string field2=");
      _ret.Append("field2 != null?'¥"' + field2 + '¥"':null);
      _ret.Append("¥n");
      _ret.Append("}");
      return _ret.ToString();
   }
   override public int GetHashCode() {
      returns base.GetHashCode();
   }
   override public bool Equals(object o) {
      if(this == o) return true;
      if(o == null) return false;
      if(o is Example.StructType) {
         Example.StructType obj = (Example.StructType) o;
         bool res = true;
```

```
            do {
               res = this.field1 == obj.field1;
               if(!res) break;
               res = this.field2 == obj.field2 ||
                  (this.field2 != null && obj.field2 != null &&
                     this.field2Equals(obj.field2));
            } while(false);
            return res;
         }
         else {
            return false;
         }
      }
   }
```

union

IDL union は同じ名前のシールされた C# クラスにマップされます。このクラスは次の内容を備えています。

- デフォルトコンストラクタ
- 共用体の判別子用のアクセス用メソッド（名前は discriminator()）
- 各要素のアクセス用メソッド
- 各要素の変更用メソッド
- 複数の case ラベルを持つ各要素のための変更用メソッド
- デフォルトの変更用メソッド（必要な場合）

マップされた union 型の名前または任意のフィールド名で名前の競合が発生した場合，通常の名前の競合解決ルールが使用されます。つまり判別子の前に下線（_）が付加されます。

要素のアクセス用メソッドと変更用メソッドはオーバーロードされて，要素名に基づいて名前が付けられます。目的の要素が設定されていない場合は，アクセス用メソッドからシステム例外 CORBA::BAD_OPERATION が発生します。

要素に対応して複数の case ラベルがある場合，その要素に対する単純な変更用メソッドにより，最初の case ラベルの値に判別変数が設定されます。さらに，明示的な判別子パラメータを使用する特別な変更用メソッドが生成されます。

要素が default という case ラベルに対応している場合，変更用メソッドは，他のどの case ラベルとも一致しない値を判別変数に設定します。

case ラベルの集合が判別変数として可能性のある値を完全にカバーしている場合，union に default の case ラベルを指定することは誤りです。このような状況を検出し，不当なコードの生成を防ぐのは，C# コードジェネレータ（IDL コンパイラなどのツール）が備える機能です。

明示的な default という case ラベルが存在せず，case ラベルのセットが判別変数として考えられる値を完全にカバーしていない場合は，デフォルトメソッド _default() が作成されます。この場合，union の値は範囲外の値に設定されます。

このサンプルコードは IDL union の C# へのマッピングを示しています。

```
// Example.idl
module Example {
   enum EnumType { first, second, third, fourth, fifth, sixth };
   union UnionType switch (EnumType) {
      case first: long win;
      case second: short place;
      case third:
      case fourth: octet show;
      default: boolean other;
      };
};

// Example.cs
public sealed class UnionType {
   private object _object;
   private Example.EnumType _disc = Example.EnumType.fifth;
   internal bool _defaultState = false;

   // コンストラクタ
   public UnionType() {
   }

   // ディスクリミネータのアクセッサ
   public Example.EnumType discriminator() {
      return _disc;
   }

   // win
   public int Win() { ... }
   public void Win(int _vis_value) { ... }

   // place
   public short Place() { ... }
   public void Place(short _vis_value) { ... }

   // 表示
   public byte Show() { ... }
   public void Show(byte _vis_value) { ... }
   public void Show(Example.EnumType disc, byte _vis_value) { ... }

   // other
   public bool Other() {...}
   public void Other(bool _vis_value) { ... }
   public void Other(Example.EnumType disc, bool _vis_value) { ... )
   override public string ToString () { . . .}
   overrinde public int GetHashCode() { ... }
   public bool Equals(object o) { . . .}
}
```

sequence と array

IDL sequence は C# 配列にマップされます。マッピングでは，シーケンス型が必要な場所では，常にシーケンスエレメントのマップされた型の配列が使用されます。

IDL array は IDL のバウンドされたシーケンスと同じようにマップされます。マッピングでは，配列型が必要な場所では，常に配列エレメントのマップされた型の配列が使用されます。C# では，C# の添え字演算子がマップ済みの配列に適用されます。配列の長さを C# で利用可能にするには，配列を IDL 定数でバウンドします。この IDL 定数は，定数の規則にしたがってマップされることになります。

次のサンプルコードは配列のマッピングを示しています。

```
// Example.idl
const long ArrayBound = 42;
typedef long larray[ArrayBound];

// Example.cs
public abstract class ArrayBound {
   public const int Value = (int) 42;
}
```

モジュール

IDL モジュールは同じ名前で C# 名前空間にマップされます。モジュール内のすべての IDL 型宣言は，生成される名前空間内の対応する C# クラスまたはインターフェース宣言にマップされます。

モジュールで囲まれていない IDL 宣言は（名前の付いていない）C# グローバルスコープにマップされます。

次のサンプルコードは IDL モジュール用に生成された C# コードを示しています。

```
// Example.idl
module Example {
   ...
};

// Example.cs
namespace Example {
   ...
}
```

インターフェース

Foo という名前のユーザー定義型がある場合，idl2cs コンパイラは次のコードを生成します。

- public sealed class FooHelper
- public interface Foo : CORBA.Object, Example.FooOperations
- public class FooOperations
- public class _FooStub

特殊な「nil」オブジェクトリファレンスは存在しません。オブジェクトリファレンスが必要な場合には，いつでも自由に C# null を渡すことができます。

属性は C# アクセス用メソッドと変更用メソッドの対にマップされます。これらのメソッドは IDL の属性と同じ名前を持ち，オーバーロードされています。IDL の「readonly」属性に対しては，変更用メソッドは作成されません。

このサンプルコードは IDL インターフェースの C# へのマッピングを示しています。

```
// Example.idl
module Example {
   interface Foo {
      long method(in long arg) raises(AnException);
      attribute long assignable;
      readonly attribute long nonassignable;
   };
};

// Example.cs
namespace Example {
   public sealed class FooHelper { ... }
   public interface Foo : CORBA.Object, Example.FooOperations {
   }
   public interface FooOperations {
      int Method(in long arg) throws Example.AnException;
      int Assignable();
      void Assignable(int assignable);
      int Nonassignable ();
   }
   public class _FooStub : CORBA.ObjectIml, Example.Foo { ... }
}
```

シグニチャおよびオペレーションインターフェース

上の例では，C# にマップされると，Foo および FooOperations の 2 つのインターフェースが IDL インターフェースの完全なシグニチャを提供します。**シグニチャ**インターフェースは，IDL ファイルで宣言したインターフェースごとにシグニチャを定義し，オペレーションインターフェースは実装の詳細を提供します。

オペレーションインターフェースには IDL インターフェースで宣言されたオペレーションと属性だけが含まれています。C# Operations インターフェースには，マップされたオペレーションシグニチャが含まれています。このインターフェースへのオブジェクトリファレンスで，メソッドを呼び出すことができます。

ヘルパークラス

ヘルパークラスは CORBA 名前空間の大半のクラス用に用意されています。また，ユーザー定義型の場合は idl2cs コンパイラによって生成され，その型用に生成されたクラスの名前（Helper サフィックスが付加されたもの）が付けられます。ヘルパークラスを使用する理由は，クラスに用意されているメソッドが不要な場合に，ロードを回避するためです。型の操作に必要な複数のスタティックメソッドが提供されます。

- Any の型の挿入／抽出操作
- リポジトリ ID の取得
- タイプコードの取得
- ストリームとの間での型の読み取り／書き込み

ヘルパークラスは静的な narrow メソッドを宣言します。このメソッドを使用すれば，CORBA.Object のインスタンスをより具体的な型のオブジェクトリファレンスに narrow（キャスト）できます。オブジェクトリファレンスが要求される型をサポートしないために narrow（キャスト）が失敗した場合は，IDL 例外 CORBA::BAD_PARAM が発生します。別種のエラーの場合には，異なるシステム例外が発生します。ヌルに対する narrow（キャスト）は常に成功し，戻り値はヌルになります。

マップされた構造体，列挙，共用体，例外，valuetype，valuebox などのオブジェクトの場合，ヘルパークラスは，オブジェクトをストリームとの間で読み取り／書き込みしてオブジェクトのリポジトリ識別子を戻すメソッドを提供します。インターフェース用に生成されたヘルパークラスには，bind，narrow などの追加のメソッドが含まれています。

すべてのヘルパークラス用のメソッド

生成されるすべてのヘルパークラスには次のメソッドが存在します。

public static <interface_name> **Extract**(CORBA.Any **any**)

このメソッドは，指定された Any オブジェクトから型を抽出します。

パラメータ	説明
any	オブジェクトを含む Any オブジェクト。

public static void **Insert**(CORBA.Any **any**, <type_name> **_vis_value**)

このメソッドは，型を指定された Any オブジェクトに挿入します。

パラメータ	説明
any	型を含む Any オブジェクト。
_vis_value	挿入する型。

public static <type_name> **Read**(CORBA.InputStream **_input**)

このメソッドは，指定された入力ストリームから型を読み取ります。

パラメータ	説明
input	オブジェクトを読み取る入力ストリーム。

public static CORBA.TypeCode **GetTypeCode**()

このメソッドは，このオブジェクトに関連付けられた TypeCode を戻します。

public static void **Write**(CORBA.OutputStream **_output**, <type_name> _vis_value)

このメソッドは，指定された出力ストリームへ型を書き込みます。

パラメータ	説明
output	オブジェクトを書き込む出力ストリーム。
value	出力ストリームに書き込まれる型。

インターフェース用に生成されるメソッド

public static <*interface_name*> **Bind**()

このメソッドは，型 <*interface_name*> のオブジェクトの任意のインスタンスにバインドを試みます。

public static <*interface_name*> **Bind**(string **name**)

このメソッドは，指定されたインスタンス名を持つ型 <*interface_name*> のオブジェクトにバインドを試みます。

パラメータ	説明
name	対象とするオブジェクトのインスタンス名。

public static <*interface_name*> **Bind**(string **name**, string **host**,
CORBA.Visi.BindOptions **options**)

このメソッドは，指定された BindOptions を使用して，指定されたインスタンス名の指定されたホスト上に存在する型 <*interface_name*> のオブジェクトにバインドを試みます。

パラメータ	説明
name	対象とするオブジェクトのインスタンス名。
host	対象とするオブジェクトが配置されているオプションのホスト名。
オプション	このオブジェクトのバインドオプション。

public static <*interface_name*> **Narrow**(CORBA.Object **obj**)

このメソッドは型 <*interface_name*> のオブジェクトに CORBA.Object リファレンスを narrow（キャスト）しようとします。オブジェクトリファレンスを narrow（キャスト）できない場合は，null 値が戻されます。

パラメータ	説明
obj	型 <interface_name> に narrow（キャスト）されるオブジェクト。

生成されるスタブクラス

スタブクラスは idl2cs コンパイラによって生成され，クライアントが呼び出す <*interface_name*> 用のスタブ実装を提供します。このクラスは，オブジェクト実装に対して透過的に作用する実装を提供します。

抽象インターフェース

IDL 抽象インターフェースは，IDL インターフェースと同じ名前の単一の public C# インターフェースにマップされます。マッピング規則は，非抽象 IDL インターフェース用の C# オペレーションインターフェースを生成する規則と同様です。ただし，このインターフェースはシグニチャインターフェースとしても機能します。

マップされた C# インターフェースは IDL インターフェースと同じ名前を持ち，指定した型のインターフェースが別のインターフェースで使用された場合には，メソッド宣言でシグニチャとしても使用されます。ここにはマップされたオペレーションシグニチャであるメソッドも含まれています。

パラメータの引渡し

IDL パラメータは通常の C# の実際のパラメータにマップされます。IDL オペレーションの結果は，対応する C# メソッドの結果として戻されます。

このサンプルコードは IDL パラメータの C# へのマッピングを示しています。

```
// Example.idl
module Example {
   interface Modes {
      long operation(in long inArg, out long outArg, inout long inoutArg);
   };
};
```

```
// Example.cs
namespace Example;
public interface Modes : CORBA.Object, Example.ModesOperations {
}
public interface ModesOperations {
   int Operation(int inArg, out int outArg, ref int inoutArg);
}
```

インターフェーススコープ

OMG IDL から C# へのマッピング仕様では，宣言をインターフェーススコープ内でネストしたり，名前空間とインターフェースに同じ名前を指定することは許可されていません。したがって，インターフェーススコープは同じ名前に NS サフィックスを付けたパッケージにマップされます。

例外用のマッピング

IDL 例外は非常に類似した形で struct にマップされます。例外およびコンストラクタのフィールド用のインスタンス変数を備えた C# クラスにマップされます。

CORBA システム例外は チェックされない例外です。RuntimeException から（間接的に）継承されます。

ユーザー定義の例外は，チェック例外です。Exception から（間接的に）継承されます。

ユーザー定義の例外

ユーザー定義の例外は CORBA.UserException を拡張する C# クラスにマップされ，それ以外の場合は，ヘルパークラスの生成を含め，IDL struct 型と同様にマップされます。

例外がネストされた IDL スコープ内で定義された場合（原則的にインターフェースまたは valuetype 内），その C# クラス名は，含まれるインターフェースまたは valuetype の名前に NS サフィックスを追加した名前の，特別な名前空間に定義されます。それ以外の場合は，その C# クラス名は，例外の内部 IDL モジュールに対応する C# 名前空間内で定義されます。

このサンプルコードはユーザー定義の例外のマッピングを示しています。

```
// Example.idl
module Example {
   exception AnException {
      string reason;
   };
};

// Example.cs
namespace Example {
   public sealed class AnExceptionHelper : CORBA.Streamable { ... }

   public sealed class AnException : CORBA.UserException {
    public string reason;
    public AnException() : base(Example.AnExceptionHelper.GetRepID()) {
      }
    public AnException(string reason) : this() {
         this.reason = reason;
      }
    public AnException (string _reason, string reason) :
         base(Example.AnExceptionHelper.GetRepId() + ' ' + _reason) {
         this.reason = reason;
      }
      override public string ToString() { . . . }
      override public GetHashCode() { ... }
      override public bool Equals(object o) { ... }
   }
}
```

システム例外

標準 IDL システム例外は CORBA.SystemException を拡張する final C# クラスにマップされ，IDL メジャーおよびマイナー例外コード，および例外の理由を記述する文字列へのアクセスを提供します。CORBA.SystemException には public コンストラクタが存在しません。これを拡張するクラスのみインスタンス化することができます。

標準 IDL 例外ごとの C# クラス名は，その IDL 名と同じであり，CORBA 名前空間に位置するように宣言されます。デフォルトコンストラクタは，マイナーコードに 0，完了コードに COMPLETED_NO，理由文字列に空の文字列（“”）をそれぞれ指定します。これ以外にも，理由を設定して他のフィールドのデフォルトを使用するコンストラクタや，3 つのパラメータをすべて指定する必要があるコンストラクタもあります。

Any 型のマッピング

IDL 型 Any は C# クラス CORBA.Any にマップされます。このクラスには，事前定義済み型のインスタンスを挿入および抽出するために必要なメソッドがすべてあります。抽出オペレーションに不整合な型がある場合，CORBA::BAD_OPERATION 例外が発生します。

さらに，可搬性のあるスタブおよびスケルトンで使用するための高速インターフェースを提供するために，挿入および抽出メソッドが定義されています。各 IDL プリミティブ型に挿入と抽出のメソッドが定義されているほか，非プリミティブの IDL 型を処理するために，汎用のストリーム可能な型にも挿入と抽出のメソッドが定義されています。

挿入オペレーションは指定した値を設定し，必要であれば Any 型をリセットします。

アクセス用メソッド type() を使用してタイプコードを設定すると，値がクリアされます。値が設定される前に値を取得しようとすると，例外 CORBA::BAD_OPERATION が発生します。IDL の out パラメータに適切な値を設定できるように，このオペレーションは事前に提供されています。

特定のネストした型用のマッピング

IDL では，インターフェース内でネストした型宣言が可能です。C# ではクラスをインターフェース内でネストできません。したがって，C# クラスにマップされた IDL 型，およびインターフェースのスコープ内で宣言された IDL 型は，C# にマップされた際に特殊な「スコープ」名前空間に記述される必要があります。

これらの型宣言を含む IDL インターフェースは，マップされた C# クラス宣言を含めるために，スコープ名前空間を生成します。スコープ名前空間名は，IDL 型名に NS を追加して作成されます。

このサンプルコードは特定のネストされた型用のマッピングを示しています。

```
// Example.idl
module Example {
   interface Foo {
      exception e1 {};
   };
};

// Example.cs
namespace Example
   public sealed class FooHelper { ... }
   public interface Foo : CORBA.Object, Example.FooOperations {
   }
   public interface FooOperations {
   }
   namespace FooNS {
      public sealed class e1Helper : CORBA.Streamable { ... }
      public sealed class e1 : CORBA.UserException { ... }
```

```
    }
    public class _FooStub : CORBAObjectIpml, Example.Foo { ... }
}
```

TypeDef 用のマッピング

単純 C# 型にマップされた IDL 型は，C# 内でサブクラスにできません。したがって，単純型用の型宣言である typedef は，typedef 型が出現するオリジナル（マップ済みの型）の任意の場所にマップされます。単純型の場合，すべての typedef 用にヘルパークラスが生成されます。

配列でもシーケンスでもない typedef は，IDL の単純な型またはユーザー定義型（typedef 以外の種類）が得られるまで，元の型へと「分解」されます。

このサンプルコードは複雑な IDL typedef のマッピングを示しています。

```
// Example.idl
module Example {
    struct EmployeeName {
        string firstName;
        string lastName;
    };
    typedef EmployeeName EmployeeRecord;
};

// Example.cs
namespace Example {
    public sealed class EmployeeNameHelper : CORBA.Streamable { ... }
    public sealed class EmployeeName { ... }
    public sealed class EmployeeRecordHelper { ... }
}
```

付録 C

Java 組み込み型サポート

この章では，VisiBroker for .NET でサポートされている組み込み型の，Java から .NET へのマッピングについて説明します。

次の表に，デフォルトの Java パッケージから .NET 名前空間への VisiBroker for .NET でのマッピングを示します。

表 16.7　デフォルトパッケージから名前空間へのマッピング

Java パッケージ	.NET 名前空間	セクションの参照
java	J2EE	
java.lang	J2EE.Lang	171 ページの「java.lang」
java.io	J2EE.Io	173 ページの「java.io」
java.math	J2EE.Math	173 ページの「java.math」
java.net	J2EE.Net	173 ページの「java.net」
java.rmi	J2EE.Rmi	174 ページの「java.rmi」
java.sql	J2EE.Sql	174 ページの「java.sql」
javax	J2EE	
javax.ejb	J2EE.Ejb	174 ページの「javax.ejb」
javax.naming	J2EE.Naming	175 ページの「javax.naming」
javax.rmi	J2EE.Rmi	175 ページの「javax.rmi」
javax.transaction	J2EE.Transaction	176 ページの「javax.transaction」
java.util	J2EE.Util	176 ページの「java.util」

java.lang

表 16.8　java.lang のベース型

Java 型	.NET 型
java.lang.Error	J2EE.Lang.Error
java.lang.Exception	J2EE.Lang.Exception
java.lang.Object	System.Object
java.lang.RuntimeException	J2EE.Lang.RuntimeException

表 16.8 java.lang のベース型 （続き）

Java 型	.NET 型
java.lang.StringBuffer	System.Text.StringBuilder
java.lang.Throwable	J2EE.Lang.Throwable

表 16.9 java.lang のプリミティブラッパー型

Java 型	.NET 型
java.lang.Boolean	J2EE.Lang.Boolean
java.lang.Byte	J2EE.Lang.Byte
java.lang.Character	J2EE.Lang.Character
java.lang.Double	J2EE.Lang.Double
java.lang.Float	J2EE.Lang.Float
java.lang.Integer	J2EE.Lang.Integer
java.lang.Long	J2EE.Lang.Long
java.lang.Number	J2EE.Lang.Number
java.lang.Short	J2EE.Lang.Short

次の java.lang エラーおよび例外型は，同じ型名で J2EE.Lang 名前空間へマップされます。

表 16.10 java.lang のエラー型

AbstractMethodError	AssertionError
ClassCircularityError	ClassFormatError
Error	ExceptionInInitializerError
IllegalAccessError	IncompatibleClassChangeError
InstantiationError	InternalError
LinkageError	NoClassDefFoundError
NoSuchFieldError	NoSuchMethodError
OutOfMemoryError	StackOverflowError
ThreadDeath	UnknownError
UnsatisfiedLinkError	UnsupportedClassVersionError
VerifyError	VirtualMachineError

表 16.11 java.lang の RuntimeException 型

ArithmeticException	ArrayIndexOutOfBoundsException
ArrayStoreException	ClassCastException
IllegalArgumentException	IllegalMonitorStateException
IllegalStateException	IllegalThreadStateException
IndexOutOfBoundsException	NegativeArraySizeException
NullPointerException	NumberFormatException
RuntimeException	SecurityException
StringIndexOutOfBoundsException	UnsupportedOperationException

表 16.12 java.lang の例外型

ClassNotFoundException	CloneNotSupportedException
IllegalAccessException	InstantiationException
InterruptedException	NoSuchFieldException
NoSuchMethodException	

java.io

次の java.io 例外型は同じ型名で J2EE.Io 名前空間へマップされます。

表 16.13 java.io の例外型

CharConversionException	EOFException
FileNotFoundException	IOException
InterruptedIOException	InvalidClassException
InvalidObjectException	NotActiveException
NotSerializableException	ObjectStreamException
OptionalDataException	StreamCorruptedException
SyncFailedException	UTFDataFormatException
UnsupportedEncodingException	WriteAbortedException

java.math

表 16.14 java.math のベース型

Java 型	.NET 型
java.math.BigDecimal	J2EE.Math.BigDecimal
java.math.BigInteger	J2EE.Math.BigInteger

java.net

表 16.15 java.net のベース型

Java 型	.NET 型
java.net.URL	System.Uri
java.net.URI	System.Uri
java.net.InetAddress	System.Net.IPHostEntry
java.net.Inet4Address	System.Net.IPHostEntry
java.net.SocketAddress	System.Net.EndPoint
java.net.InetSocketAddress	System.Net.IPEndPoint

次の java.net 例外型は同じ型名で J2EE.Net 名前空間へマップされます。

表 16.16 java.net の例外型

BindException	ConnectException
MalformedURLException	NoRouteToHostException
PortUnreachableException	ProtocolException
SocketException	SocketTimeoutException
URISyntaxException	UnknownHostException
UnknownServiceException	

java.rmi

表 16.17 java.rmi のベース型

Java 型	.NET 型
java.rmi.Remote	CORBA.Object

次の java.rmi 例外型は同じ型名で J2EE.Rmi 名前空間へマップされます。

表 16.18 java.rmi の RuntimeException 型

RMISecurityException

表 16.19 java.rmi の例外型

AccessException	AlreadyBoundException
ConnectException	ConnectIOException
MarshalException	NoSuchObjectException
NotBoundException	RemoteException
ServerError	ServerException
ServerRuntimeException	StubNotFoundException
UnexpectedException	UnknownHostException
UnmarshalException	

java.sql

表 16.20 java.sql のベース型

Java 型	.NET 型
java.sql.Date	System.DateTime
java.sql.Time	System.DateTime
java.sql.Timestamp	System.DateTime

次の java.sql 例外型は同じ型名で J2EE.Sql 名前空間へマップされます。

表 16.21 java.sql の例外型

BatchUpdateException	DataTruncation
SQLException	SQLWarning

javax.ejb

表 16.22 javax.ejb のベース型

Java 型	.NET 型
javax.ejb.EJBHome	J2EE.Ejb.EJBHome
javax.ejb.EJBMetaData	J2EE.Ejb.EJBMetaData
javax.ejb.EJBObject	J2EE.Ejb.EJBObject
javax.ejb.Handle	J2EE.Ejb.Handle
javax.ejb.HomeHandle	J2EE.Ejb.HomeHandle

次の javax.ejb 例外型は同じ型名で J2EE.Ejb 名前空間へマップされます。

表 16.23 javax.ejb の例外型

AccessLocalException	CreateException
DuplicateKeyException	EJBException
FinderException	NoSuchEntityException
ObjectNotFoundException	RemoveException
TransactionRequiredLocalException	TransactionRolledbackLocalException

javax.naming

表 16.24 javax.naming のベース型

Java 型	.NET 型
javax.naming.Binding	J2EE.Naming.Binding
javax.naming.Context	J2EE.Naming.Context
javax.naming.InitialContext	J2EE.Naming.InitialContext
javax.naming.NameClassPair	J2EE.Naming.NameClassPair

次の javax.naming 例外型は同じ型名で J2EE.Naming 名前空間へマップされます。

表 16.25 javax.naming の例外型

AuthenticationException	AuthenticationNotSupportedException
CannotProceedException	CommunicationException
ConfigurationException	ContextNotEmptyException
InsufficientResourcesException	InterruptedNamingException
InvalidNameException	LimitExceededException
LinkException	LinkLoopException
MalformedLinkException	NameAlreadyBoundException
NameNotFoundException	NamingException
NamingSecurityException	NoInitialContextException
NoPermissionException	NotContextException
OperationNotSupportedException	PartialResultException
ReferralException	ServiceUnavailableException
SizeLimitExceededException	TimeLimitExceededException

javax.rmi

表 16.26 javax.rmi のベース型

Java 型	.NET 型
javax.rmi.PortableRemoteObject	J2EE.Rmi.PortableRemoteObject

javax.transaction

次の javax.transaction 例外型は同じ型名で J2EE.Transaction 名前空間へマップされます。

表 16.27 javax.transaction の例外型

HeuristicCommitException	HeuristicMixedException
HeuristicRollbackException	InvalidTransactionException
NotSupportedException	RollbackException
SystemException	TransactionRequiredException
TransactionRolledbackException	

java.util

表 16.28 java.util のベース型

Java 型	.NET 型
java.util.Calendar	J2EE.Util.Calendar
java.util.Date	System.DateTime
java.util.GregorianCalendar	J2EE.Util.GregorianCalendar
java.util.Locale	System.Globalization.CultureInfo
java.util.Random[1]	System.Random
java.util.TimeZone	System.TimeZone
java.util.SimpleTimeZone	System.TimeZone
sun.util.calendar.ZoneInfo[2]	System.TimeZone

1. この型の使用方法については，26 ページの「相互運用のプロパティ」を参照してください。
2. `sun.util.calendar.ZoneInfo` は，主に Sun の内部実装クラスになりますが，リモートプロシージャコールで使用されることもあります。

表 16.29 java.util のイテレーションインターフェース

Java 型	.NET 型
java.util.Comparator	System.Collections.IComparer
java.util.Iterator	System.Collections.IEnumerator
java.util.ListIterator	System.Collections.IEnumerator

表 16.30 java.util のコレクションインターフェース

Java 型	.NET 型
java.util.Collection	System.Collections.ICollection
java.util.List	System.Collections.IList
java.util.Map	System.Collections.IDictionary
java.util.Set	System.Collections.ICollection
java.util.SortedMap	System.Collections.IDictionary
java.util.SortedSet	System.Collections.ICollection

表 16.31 java.util の抽象コレクションクラス

Java 型	.NET 型
java.util.AbstractCollection	System.Collections.ICollection
java.util.AbstractList	System.Collections.IList
java.util.AbstractMap	System.Collections.IDictionary
java.util.AbstractSequentialList	System.Collections.IList
java.util.AbstractSet	System.Collections.ICollection
java.util.Dictionary	System.Collections.IDictionary

表 16.32 java.util の Public 具象コレクションクラス

Java 型	.NET 型
java.util.ArrayList	System.Collections.ArrayList
java.util.BitSet	System.Collections.BitArray
java.util.HashMap	System.Collections.Hashtable
java.util.HashSet	System.Collections.ArrayList
java.util.Hashtable	System.Collections.Hashtable
java.util.IdentityHashMap	System.Collections.Hashtable
java.util.LinkedHashMap	System.Collections.Hashtable
java.util.LinkedHashSet	System.Collections.ArrayList
java.util.LinkedList	System.Collections.ArrayList
java.util.Properties	System.Collections.Specialized.StringDictionary
java.util.Stack	System.Collections.Stack
java.util.TreeMap	System.Collections.Hashtable
java.util.TreeSet	System.Collections.ArrayList
java.util.Vector	System.Collections.ArrayList

表 16.33 java.util の内部具象コレクションクラス

Java 型	.NET 型
java.util.Arrays$ArrayList	System.Collections.ArrayList
java.util.Collections$CopiesList	System.Collections.ArrayList
java.util.Collections$SingletonList	System.Collections.ArrayList
java.util.Collections$SingletonMap	System.Collections.Hashtable
java.util.Collections$SingletonSet	System.Collections.ArrayList
java.util.Collections$SynchronizedCollection	System.Collections.ICollection
java.util.Collections$SynchronizedList	System.Collections.IList
java.util.Collections$SynchronizedMap	System.Collections.IDictionary
java.util.Collections$SynchronizedRandomAccessList	System.Collections.IList
java.util.Collections$SynchronizedSet	System.Collections.ICollection
java.util.Collections$SynchronizedSortedMap	System.Collections.IDictionary
java.util.Collections$SynchronizedSortedSet	System.Collections.ICollection
java.util.Collections$UnmodifiableCollection	System.Collections.ICollection
java.util.Collections$UnmodifiableList	System.Collections.IList
java.util.Collections$UnmodifiableMap	System.Collections.IDictionary
java.util.Collections$UnmodifiableSet	System.Collections.ICollection
java.util.Collections$UnmodifiableSortedMap	System.Collections.IDictionary
java.util.Collections$UnmodifiableSortedSet	System.Collections.ICollection
java.util.TreeMap$SubMap	System.Collections.Hashtable

次の java.util 例外型は同じ型名で J2EE.Util 名前空間へマップされます。

表 16.34 java.util の RuntimeException 型

ConcurrentModificationException	EmptyStackException
MissingResourceException	NoSuchElementException

表 16.35 java.util の例外型

TooManyListenersException

アプリケーションサーバーサポート

次の表に，VisiBroker for .NET に含まれるアプリケーションサーバー固有の型マッピングを示します。

表 16.36 アプリケーションサーバー型

アプリケーションサーバー	Java 型	.NET 型
Borland AppServer	com.inprise.ejb.iterator CustomVector	System.Collections.ArrayList

索引

記号

数値

A

B

C

D

E

F

G

H

I

J

M

N

O

P

Q

R

S

T

U

V

W

X

あ

い

う

お

か

き

く

こ

さ

し

す

せ

そ

ち

つ

て

と

な

に

ぬ

ね

は

ひ

ふ

へ

ほ

ま

め

も

ゆ

よ

ら

り

る

れ