HITACHI
Inspire the Next

VisiBroker Version 5

Borland(R) Enterprise Server VisiBroker(R)　デベロッパーズガイド

解説・手引・文法・操作書

3020-3-Y30-30

■ 対象製品

●適用 OS：Windows Server 2008 x86，Windows Server 2008 x64，Windows Server 2008 R2，Windows Server 2012，Windows Server 2012 R2，Windows 7 x86，Windows 7 x64，Windows 8 x86，Windows 8 x64，Windows 8.1 x86，Windows 8.1 x64

P-2464-AF64 Cosminexus TPBroker 05-22

●適用 OS：Windows Server 2008 x64，Windows Server 2008 R2，Windows Server 2012，Windows Server 2012 R2，Windows 7 x64，Windows 8 x64，Windows 8.1 x64

P-2964-AF64 Cosminexus TPBroker 05-22

●適用 OS：AIX V6.1，AIX V7.1

P-1M64-CF61 Cosminexus TPBroker 05-23

●適用 OS：Red Hat Enterprise Linux 5 Advanced Platform（AMD/Intel 64），Red Hat Enterprise Linux 5（AMD/Intel 64），Red Hat Enterprise Linux Server 6（64-bit x86_64）

P-9S64-AF61 Cosminexus TPBroker 05-23

これらのプログラムプロダクトのほかにも，このマニュアルをご利用になれる場合があります。詳細は「リリースノート」でご確認ください。

■ 輸出時の注意

本製品を輸出される場合には、外国為替及び外国貿易法の規制並びに米国輸出管理規則など外国の輸出関連法規をご確認の上、必要な手続きをお取りください。

なお、不明な場合は、弊社担当営業にお問い合わせください。

■ 商標類

■ 発行

2015 年 4 月　3020-3-Y30-30

■ 著作権

変更内容

変更内容(3020-3-Y30-30)　Cosminexus TPBroker 05-23，Cosminexus TPBroker 05-22

追加・変更内容	変更個所
Windows Vista，XP，HP-UX を削除した。	－

単なる誤字・脱字などはお断りなく訂正しました。

はじめに

このマニュアルは，Borland Enterprise Server VisiBroker の基本的な使用方法および高度な機能の取り扱い方法について説明したものです。Borland Enterprise Server VisiBroker は，分散オブジェクトベースのアプリケーションを，Common Object Request Broker Architecture（CORBA）仕様に従って開発し運用できるようにします。

■ 対象読者

Borland Enterprise Server VisiBroker を用いて，CORBA の仕様に基づく分散アプリケーションを開発する方を対象としています。また，オブジェクト指向の開発に精通した C++または Java プログラマの方を対象としています。

■ このマニュアルで使用している記号

このマニュアルでは，次に示す表記を使用しています。

表記	意味
＜　＞＋＜　＞	＋の前のキーを押したまま，あとのキーを押すことを意味します。

■ 文法の記号

このマニュアルで使用する文法記述記号を説明します。文法記述記号は文法の記述形式について説明する記号です。

文法記述記号	意味
ボールド体 （boldface）	ボールド体は，記述どおりに構文をタイプすることを示します。また，コードサンプル部分を強調表示する場合にも使用されます。UNIX の場合は，データベース名，ファイル名，および同義語を示すのに使用されます。
|	横に並べられた複数の項目に対し，項目間の区切りを示し，「または」の意味を示します。 （例） A | B | C は「A，B，または C」を意味します。
[ ]	この記号で囲まれている項目は省略してもよいことを意味します。複数の項目が横に並べて記述されている場合には，すべてを省略するか，どれか一つを選択します。 （例） [A | B] では「何も指定しない」か，「A または B と指定する」ことを意味します。
{ }	この記号で囲まれている項目は，一つの構文の要素として扱うことを意味します。
＜ ＞	この記号で囲まれている項目は，該当する要素を指定することを意味します。
…	記述が省略されていることを示します。

目次

第 1 編　基本概念

第 4 編 ツールとサービス

1 CORBA モデルの解説

この章では, CORBA 2.5 の仕様に完全に準拠したインプリメンテーションである Borland Enterprise Server VisiBroker について説明します。また, Borland Enterprise Server VisiBroker の機能とコンポーネントについても説明します。

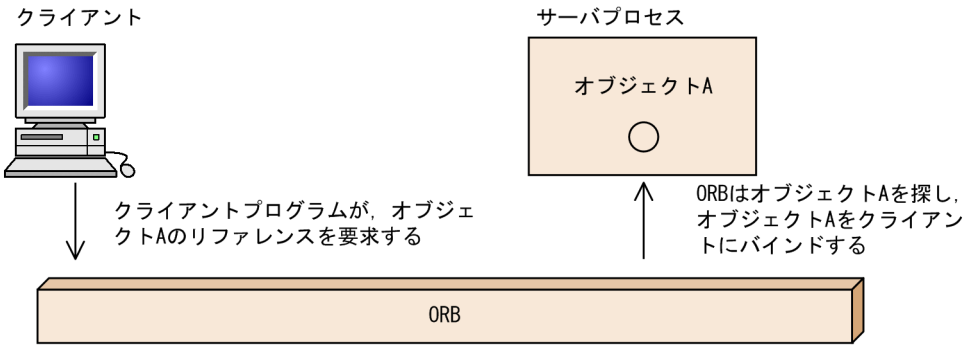
1.1　CORBA とは

CORBA（Common Object Request Broker Architecture）は，アプリケーションが，書いた言語やアプリケーションの場所に関係なく，分散アプリケーション間のインターオペラビリティ（アプリケーション間の通信）を実現します。

CORBA の仕様は，分散オブジェクトアプリケーションの開発を単純化し，コストを削減するために OMG（オブジェクトマネージメントグループ）が採用したものです。CORBA では，オブジェクト指向技術を使用し，アプリケーション間での再利用と共有ができるソフトウェアコンポーネントを作成します。各オブジェクトは，細かな内部処理をカプセル化し，アプリケーションの複雑さを低減する優れたインタフェースを提供します。一度インプリメントしテストしたオブジェクトは，繰り返し使用できるためアプリケーションの開発コストを削減できます。

クライアントプログラムがオブジェクトを処理する流れを図 1-1 に示します。図 1-1 の ORB（Object Request Broker）は，使用したいオブジェクトにクライアントアプリケーションを接続します。クライアントプログラムは，通信相手のオブジェクトインプリメンテーションが同じコンピュータにあるのか，またはネットワークのどこかにあるリモートコンピュータにあるのかを意識する必要はありません。クライアントプログラムは，オブジェクト名とオブジェクトインタフェースの使用方法だけを知っていればよいのです。オブジェクトの探索，リクエストのルーティング，および結果の応答は，ORB が担当します。

図 1-1　オブジェクトを処理するクライアントプログラム



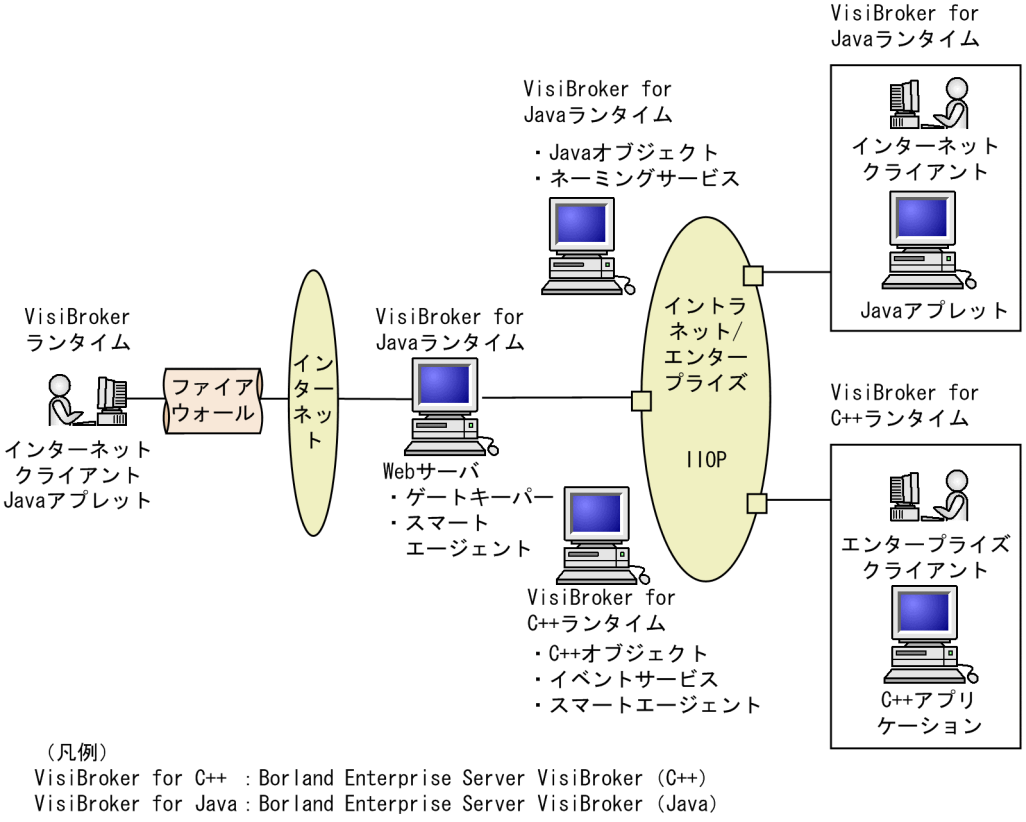
注

ORB は独立したプロセスではありません。ORB はエンドユーザアプリケーション内で統合されるライブラリとネットワークリソースの集まりで，クライアントアプリケーションがオブジェクトを探して使用できるようにします。

1.2　Borland Enterprise Server VisiBroker とは

Borland Enterprise Server VisiBroker は完全な CORBA 2.5 の仕様の ORB ランタイムを提供し，オープンで柔軟かつインターオペラビリティを持った C++および Java の双方で分散オブジェクトを構築し，配置し，管理するための開発環境をサポートします。Borland Enterprise Server VisiBroker で構築したオブジェクトは，インターネットまたはイントラネットを介した分散オブジェクト間通信用の OMG の IIOP（Internet Inter-ORB Protocol）を使用して通信する Web ベースのアプリケーションから容易にアクセスできます。Borland Enterprise Server VisiBroker は IIOP の組み込みインプリメンテーションを備えており，高性能とインターオペラビリティを約束します。Borland Enterprise Server VisiBroker のアーキテクチャを図 1-2 に示します。

図 1-2　Borland Enterprise Server VisiBroker アーキテクチャ



1.3　Borland Enterprise Server VisiBroker の機能

Borland Enterprise Server VisiBroker には，次に示す機能があります。

1.3.1　Borland Enterprise Server VisiBroker のスマートエージェントアーキテクチャ

Borland Enterprise Server VisiBroker のスマートエージェント（osagent）は，クライアントアプリケーションとオブジェクトインプリメンテーションの両方に機能を提供する動的な分散ディレクトリサービスです。ネットワーク上の複数のスマートエージェントは，協調動作することで負荷を分散し，クライアントからサーバオブジェクトにアクセスしやすくします。スマートエージェントはネットワークで使用できるオブジェクトを管理し，起動時にクライアントアプリケーションが要求するオブジェクトを探します。Borland Enterprise Server VisiBroker のスマートエージェントアーキテクチャは，サーバクラッシュやネットワーク障害などのエラーによってクライアントアプリケーションとサーバオブジェクトとのコネクションが失われたかどうかを確認できます。障害を検出すると，（そのような構成であれば）クライアントと別のホスト上のほかのサーバとのコネクションの確立が自動的に試行されます。スマートエージェントの詳細については，「12.　スマートエージェントの使用」および「10.5　Quality of Service の使用」を参照してください。

1.3.2　ロケーションサービスを使用した高度なオブジェクト探索

Borland Enterprise Server VisiBroker は CORBA の仕様の拡張機能である強力なロケーションサービスを提供し，複数のスマートエージェントから情報にアクセスできるようにします。ネットワーク上でスマートエージェントと一緒に使用すると，ロケーションサービスはクライアントがバインドできるオブジェクトの使用可能なインスタンスをすべて参照できます。コールバック機構であるトリガーを使用すれば，クライアントアプリケーションはオブジェクトの可用性に関する変更をすぐに知ることができます。インタセプタと一緒に使用すると，ロケーションサービスは，サーバオブジェクトへのクライアントリクエストの高度な負荷分散の開発に役立ちます。詳細については，「13.　ロケーションサービスの使用」を参照してください。

1.3.3　インプリメンテーションとオブジェクト活性化のサポート

Borland Enterprise Server VisiBroker の OAD（オブジェクト活性化デーモン）を使用すれば，クライアントが使用したい場合にオブジェクトインプリメンテーションを自動的に起動できます。また，Borland Enterprise Server VisiBroker は，クライアントリクエストを受信するまでオブジェクトの活性化を遅延できる機能を提供します。活性化の遅延対象は，特定オブジェクトまたはサーバ上のオブジェクトのクラス全体です。サーバントマネージャの詳細については，「7.　POA の使用」を参照してください。

1.3.4　スレッドとコネクションの強力な管理

Borland Enterprise Server VisiBroker のスレッドパーセッションモデルを使用すると，複数のリクエストにサービスを提供するために，スレッドはクライアントごとにコネクションが自動的に割り当てられ，コネクションの終了とともにスレッドも終了します。スレッドプーリングモデルを使用すると，サーバオブジェクトへのリクエストのトラフィック量に基づいてスレッドが割り当てられます。つまり，非常にアクティブなクライアントは，複数のスレッドからサービスされ，リクエストが迅速に実行されるようにするのに対し，それよりもアクティブではないクライアントは一つのスレッドを共用しながらも，リクエストがすぐにサービスされるようにします。

Borland Enterprise Server VisiBroker のコネクション管理は，サーバとのクライアントコネクションの数を最小限にします。同じサーバにあるオブジェクトに対するすべてのクライアントリクエストは，別々のスレッドから発行されていても，同じコネクションを通じて多重化されます。また，解放されたクライアントコネクションは，あとで同じサーバに再接続するために再使用され，クライアントが同じサーバへ新たにコネクションを確立するときのオーバヘッドを発生させないようにします。

スレッドとコネクションの動作はすべて設定できます。Borland Enterprise Server VisiBroker によるスレッドとコネクションの管理方法の詳細については，「8.　スレッドとコネクションの管理」を参照してください。

1.3.5　IDL コンパイラ

Borland Enterprise Server VisiBroker には，オブジェクト開発を容易にする二つの IDL コンパイラが提供されています。

idl2cpp（C++の場合）

idl2cpp コンパイラは IDL ファイルを入力として受け取り，必要なクライアントスタブとサーバスケルトンを C++で生成します。

idl2java（Java の場合）

idl2java コンパイラは IDL ファイルを入力として受け取り，必要なクライアントスタブとサーバスケルトンを Java で生成します。

idl2ir（C++および Java の場合）

idl2ir コンパイラは IDL ファイルを受け取り，IR（インタフェースリポジトリ）にその内容を格納します。

これらのコンパイラの詳細については，「11.　IDL の使用」および「16.　インタフェースリポジトリの使用」を参照してください。

1.3.6　DII と DSI を使用した動的起動

動的起動では，Borland Enterprise Server VisiBroker は DII（動的起動インタフェース）と DSI（動的スケルトンインタフェース）のインプリメンテーションを提供します。DII は，コンパイル時に定義されていないオブジェクトへの要求を，クライアントアプリケーションが動的に生成できるようにします。DII については，「17.　動的起動インタフェースの使用」を参照してください。DSI は，コンパイル時に定義されていないオブジェクトへのクライアントオペレーション要求をサーバがディスパッチできるようにします。DSI については，「18.　動的スケルトンインタフェースの使用」を参照してください。

1.3.7　インタフェースリポジトリとインプリメンテーションリポジトリ

インタフェースリポジトリ（IR）は，VisiBroker ORB オブジェクトのメタ情報のオンラインデータベースです。オブジェクトに関するメタ情報には，モジュール，インタフェース，オペレーション，属性，および例外についての情報があります。「16.　インタフェースリポジトリの使用」では，IR のインスタンスを起動する方法，IDL ファイルから IR に情報を追加する方法，および IR から情報を取り出す方法を説明します。

インプリメンテーションリポジトリは，VisiBroker ORB オブジェクトのインプリメンテーションについてのメタ情報のオンラインデータベースです。OAD は，クライアントがオブジェクトを参照するときにインプリメンテーションを自動的に活性化するための，インプリメンテーションリポジトリとの Borland Enterprise Server VisiBroker のインタフェースです。「15.　オブジェクト活性化デーモンの使用」を参照してください。

1.3.8 サーバ側のポータビリティ

Borland Enterprise Server VisiBroker は，BOA（基本オブジェクトアダプタ）の代替機能である CORBA の仕様に準拠した POA（ポータブルオブジェクトアダプタ）をサポートします。POA は，オブジェクトの活性化，トランジェントまたはパーシステントオブジェクトのサポートなどの幾つかの BOA 機能を共用します。また POA は，オブジェクトのインスタンスの作成と管理をする POA マネージャとサーバントマネージャなどの新機能も持っています。詳細については，「7. POA の使用」を参照してください。

1.3.9 インタセプタとオブジェクトラッパーを使用した VisiBroker ORB のカスタマイズ

VisiBroker 4.x インタセプタは，クライアントとサーバの間の隠蔽された通信を開発者に見えるようにします。VisiBroker 4.x インタセプタは，Borland が独自に開発したインタセプタです。このインタセプタを使用すれば，分散アプリケーションの特殊なニーズを満たす負荷分散，監視，およびセキュリティを可能にするカスタマイズされたクライアントコードとサーバコードを使って，VisiBroker ORB を拡張できます。詳細については，「19. ポータブルインタセプタの使用」を参照してください。

Borland Enterprise Server VisiBroker には OMG 標準化機能に基づいたポータブルインタセプタもあります。このインタセプタによって，インタセプタに対するポータブルコードの書き込みが可能となり，この機能を各種ベンダ ORB で使用できます。詳細については，CORBA 2.5 の仕様のインタセプタについての記述を参照してください。

Borland Enterprise Server VisiBroker のオブジェクトラッパーは，クライアントアプリケーションがバインドしたオブジェクトのメソッドを呼び出すとき，またはサーバアプリケーションがオペレーション要求を受信するときに呼び出されるメソッドを定義できるようにします。詳細については，「21. オブジェクトラッパーの使用」を参照してください。

1.3.10 イベントキュー

サーバ側の唯一の機能としてイベントキューがあります。サーバは，サーバが対象とするイベントのタイプに基づいてイベントキューにリスナーを登録できます。サーバは，必要なときにこのイベントを処理します。

イベントキューの詳細については，「22. イベントキュー」を参照してください。

1.3.11 ネーミングサービスのバッキングストア（外部記憶装置）

インターオペラビリティがある新しいネーミングサービスは，プラガブルバッキングストア（外部記憶装置）と統合して，ネーミングサービスの状態をパーシステンスにできます。これによって，ネーミングサービスでのフォルトトレランスとフェールオーバー機能を容易にします。詳細については，「14.10 プラガブルバッキングストア」を参照してください。

1.3.12 Web ネーミング（Java）

Web ネーミング機能によって，URL（ユニフォームリソースロケータ）とオブジェクトを対応させることができます。これによって，URL を指定してそのオブジェクトのリファレンスを取得できます。詳細については，「26. URL ネーミングの使用」を参照してください。

1.3.13　IDL を使用しないインタフェースの定義（Java）

Borland Enterprise Server VisiBroker の java2iiop コンパイラによって，IDL（インタフェース定義言語）の代わりに Java 言語を使用してインタフェースを定義できます。CORBA 分散オブジェクトとインターオペラビリティを持たせたい既存の Java コードがある場合，または IDL を利用したくない場合に java2iiop コンパイラを使用できます。詳細については，「23.　RMI-IIOP の使用」を参照してください。

1.3.14　ゲートキーパー

Borland Enterprise Server VisiBroker ゲートキーパーによって，Web サーバ上にあるオブジェクトに対してクライアントプログラムがオペレーション要求を発行し，Web ブラウザが強要するセキュリティ制限に準拠している限り，そのオブジェクトからのコールバックを受信できます。また，ゲートキーパーはファイアウォールを介した通信を処理し，HTTP デーモンとして使用できます。ゲートキーパーは OMG CORBA ファイアウォール仕様に完全に準拠しています。詳細については，マニュアル「Borland Enterprise Server VisiBroker　ゲートキーパーガイド」を参照してください。

1.4　CORBA に対する Borland Enterprise Server VisiBroker の準拠

Borland Enterprise Server VisiBroker は，OMG の CORBA 2.5 の仕様に完全に準拠しています。詳細については，CORBA の仕様を参照してください。

1.5　Borland Enterprise Server VisiBroker の開発環境

Borland Enterprise Server VisiBroker は開発フェーズと配置フェーズの両方で使用します。Borland Enterprise Server VisiBroker の開発環境には，次のコンポーネントが含まれます。

- アドミニストレーションツールとプログラミングツール
- VisiBroker ORB

1.5.1　プログラミングツール

開発フェーズで使用するツールを表 1-1 に示します。

表 1-1　プログラミングツール一覧

ツール	目的
idl2ir	Borland Enterprise Server VisiBroker の IDL ファイルで定義されたインタフェースを IR に格納します。
idl2java	Java のスタブとスケルトンを IDL ファイルから生成します。
idl2cpp	C++のスタブとスケルトンを IDL ファイルから生成します。
java2iiop	Java のスタブとスケルトンを Java ファイルから生成します。このツールによって，使用するインタフェースを IDL ではなく Java で定義できます。
java2idl	Java バイトコードを格納したファイルから IDL ファイルを生成します。

1.5.2　CORBA サービスツール

開発時に VisiBroker ORB を管理するために使用するツールを表 1-2 に示します。

表 1-2　CORBA サービスツール一覧

ツール	目的
irep	インタフェースリポジトリを管理するのに使用します。「16.　インタフェースリポジトリの使用」を参照してください。
oad	OAD を管理するのに使用します。「15.　オブジェクト活性化デーモンの使用」を参照してください。
nameserv	ネーミングサービスのインスタンスを起動するのに使用します。「14.　ネーミングサービスの使用」を参照してください。

1.5.3　アドミニストレーションツール

開発時に VisiBroker ORB を管理するために使用するツールを表 1-3 に示します。

表 1-3　アドミニストレーションツール一覧

ツール	目的
oadutil list	OAD に登録されている VisiBroker ORB オブジェクトインプリメンテーションをリスト出力します。

ツール	目的
oadutil reg	OAD に VisiBroker ORB オブジェクトインプリメンテーションを登録します。
oadutil unreg	OAD から VisiBroker ORB オブジェクトインプリメンテーションを登録解除します。
osagent	スマートエージェントを管理します。「12.　スマートエージェントの使用」を参照してください。
osfind	特定のネットワークで動作しているオブジェクトについて報告します。

1.6　Java 開発環境

Borland Enterprise Server VisiBroker は Java ランタイム環境で次のコンポーネントを使用します。

- Java 2 標準版
- Java ランタイム環境
- Borland Enterprise Server VisiBroker での必要事項

1.6.1　Java 2 標準版

VisiBroker ORB を使用するアプレットやアプリケーションを開発するには，Inprise JBuilder のような Java 開発環境が必要です。JavaSoft の JDK（Java Developer's Kit）もまた Java ランタイム環境を含んでいます。

Sun Microsystems は，Java ランタイム環境を組み込んだ JavaSoft の JDK を Solaris，Windows プラットフォーム用に用意しました。この JDK は Sun Microsystems の Web サイトからダウンロードできます。

JDK は IBM AIX，OS/2，SGI IRIX，および HP-UX にも格納されています。このバージョンの JDK は，それぞれのハードウェアベンダの Web サイトからダウンロードできます。各プラットフォームが使用できるものを確認するには，Sun Microsystems の Java Soft Web サイトにアクセスしてください。

1.6.2　Java ランタイム環境

Borland Enterprise Server VisiBroker の開発環境のサービスおよびツールを実行するすべてのエンドユーザには，Java ランタイム環境が必要です。Java ランタイム環境とは，Java アプリケーションを解釈して実行するエンジンのことです。一般に，Java ランタイム環境は Java 開発環境にバンドルされています。

1.6.3　CORBA に対する Borland Enterprise Server VisiBroker の準拠での必要事項

ゲートキーパーには，JavaServer Web Development Kit 1.0.1 で取得する Servlet API 2.1 を使用する必要があります。

1.6.4　Java 対応 Web ブラウザ

アプレットは，Netscape Communicator，Netscape Navigator，Microsoft Internet Explorer などの Java 対応の Web ブラウザで実行できます。

1.7　C++または Java の Borland Enterprise Server VisiBroker でのインターオペラビリティ

Borland Enterprise Server VisiBroker（Java）で開発したアプリケーションは，Borland Enterprise Server VisiBroker（C++）で開発したオブジェクトインプリメンテーションにリクエストできます。同じように，Borland Enterprise Server VisiBroker（C++）で作成したアプリケーションも Borland Enterprise Server VisiBroker（Java）で開発したオブジェクトインプリメンテーションにリクエストできます。例えば，Borland Enterprise Server VisiBroker（C++）で Java アプリケーションを使用する場合，単に Java アプリケーションを開発するために使用した同じ IDL を，Borland Enterprise Server VisiBroker（C++）で提供されている IDL コンパイラへの入力として使用します。これで，結果として生成された C++スケルトンを使用してオブジェクトインプリメンテーションを開発できます。Borland Enterprise Server VisiBroker（Java）で C++アプリケーションを使用するには，同じ処理を行います。ただし，Borland Enterprise Server VisiBroker（Java）で提供されている IDL コンパイラを使用します。

また，Borland Enterprise Server VisiBroker（Java）で書かれたオブジェクトインプリメンテーションは，Borland Enterprise Server VisiBroker（C++）で作成したクライアントと一緒に動作します。すなわち，Borland Enterprise Server VisiBroker（Java）で作成したサーバは，どの CORBA 準拠クライアントとも一緒に動作し，Borland Enterprise Server VisiBroker（Java）で書かれたクライアントは，どの CORBA 準拠サーバとも一緒に動作します。また，これは Borland Enterprise Server VisiBroker（C++）のオブジェクトインプリメンテーションにも適用されます。

1.8　ほかの ORB 製品とのインターオペラビリティ

CORBA 準拠のソフトウェアオブジェクトは IIOP を使用して通信し，互いのインプリメンテーションについて知識を持たない異なるベンダが開発した場合でも，完全なインターオペラビリティを実現します。Borland Enterprise Server VisiBroker が IIOP を使用すれば，Borland Enterprise Server VisiBroker を使用して開発したクライアントアプリケーションとサーバアプリケーションは，さまざまなベンダの ORB 製品とのインターオペラビリティを実現できます。

1.9　IDL から C++へのマッピング（C++）

Borland Enterprise Server VisiBroker は，OMG IDL/C++言語マッピング仕様に従っています。idl2cpp コンパイラがインプリメントしている，Borland Enterprise Server VisiBroker の現在の IDL の C++言語へのマッピングの概要については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「IDL から C++言語へのマッピング」の記述を参照してください。

マッピング仕様の詳細については，OMG IDL/C++言語マッピング仕様を参照してください。

1.10　IDL から Java へのマッピング（Java）

Borland Enterprise Server VisiBroker は OMG IDL/Java 言語マッピング仕様に従っています。idl2java コンパイラがインプリメントしている Borland Enterprise Server VisiBroker の現在の IDL の Java 言語へのマッピングの概要については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「IDL から Java へのマッピング」の記述を参照してください。

マッピング仕様の詳細については，OMG IDL/Java 言語マッピング仕様を参照してください。

2 環境設定

この章では，Borland Enterprise Server VisiBroker を使用する前に設定する環境変数について説明します。

2.1　PATH 環境変数の設定

ここでは，PATH 環境変数の設定方法を説明します。

注

- PATH 環境変数は，Borland Enterprise Server VisiBroker の bin ディレクトリを含めるようにインストール中に自動的に設定されます。
- 環境変数 PATH に JavaVM のディレクトリをダブルクォーテーション(")で囲って設定しないでください。

PATH 環境変数の明示的な設定を選択する場合の手順を次に説明します。

2.1.1　Windows の DOS コマンドによる PATH 環境変数の設定

Borland Enterprise Server VisiBroker が c:¥inprise¥vbroker にインストールされている場合, 次の DOS コマンドで PATH 環境変数を設定できます。

```
prompt> set PATH=c:¥inprise¥vbroker¥bin;%PATH%
```

Borland Enterprise Server VisiBroker 配布内容が，ドライブ C のデフォルトのディレクトリ BES にインストールされている場合，PATH 環境変数を次のように設定できます。

```
prompt> set PATH=c:¥BES¥vbroker¥bin;%PATH%
```

2.1.2　Windows のシステムコントロールパネルによる PATH 環境変数の設定

DOS の set コマンドを使用して Windows の環境変数を設定できますが，システムコントロールパネルを使用して自動的に PATH 環境変数を設定する方が簡単です。Borland Enterprise Server VisiBroker が c:¥inprise¥vbroker にインストールされている場合，次に示す手順でシステムコントロールパネルの PATH 環境変数を編集します。

1. システムコントロールパネルを開きます。
2. システムプロパティウィンドウで環境変数ボタンを選択します。
3. 変数に「PATH」を選択して編集します。
4. 編集ボタンをクリックして変数の値を編集します。
5. 次のパス名を PATH に追加します。

   ```
   c:¥inprise¥vbroker¥bin
   ```

   Borland Enterprise Server VisiBroker がデフォルトのディレクトリにインストールされていれば，PATH 環境変数は次のようになります。

   ```
   c:¥BES¥vbroker¥bin
   ```

注

システムコントロールパネルで環境変数を変更しても，現在実行中のアプリケーションには反映されませんが，そのあと起動されたアプリケーションとコマンドプロンプトには新しい設定が反映されます。

2.1.3　UNIX での PATH 環境変数の設定

csh を使用していて，かつ Borland Enterprise Server VisiBroker を/usr/local/vbroker にインストールしている場合，次に示すコマンドを使用して PATH 環境変数を更新できます。

```
prompt> setenv PATH /usr/local/vbroker/bin:$PATH
```

Bourne シェルを使用していて，かつ Borland Enterprise Server VisiBroker を/usr/local/vbroker にインストールしている場合，次に示すコマンドを使用して PATH 環境変数を更新できます。

```
prompt> PATH=$PATH:/usr/local/vbroker/bin
prompt> export PATH
```

2.2　CLASSPATH 環境変数の設定（Java）

CLASSPATH 環境変数には，システムで使用している各種 Java パッケージの位置を定義します。
Borland Enterprise Server VisiBroker をインストールしたり構成したりする場合は，CLASSPATH 環境変数を設定する必要はありません。

2.3　VBROKER_ADM 環境変数の設定

VBROKER_ADM 環境変数には，Borland Enterprise Server VisiBroker の OAD（オブジェクト活性化デーモン）およびスマートエージェントの構成情報が格納されている管理ディレクトリを定義します。

VBROKER_ADM 環境変数は，必ず設定してください。

2.3.1　Windows での VBROKER_ADM 環境変数の設定

自分のディレクトリ C:¥my¥adm を使用したい場合は, 次のように VBROKER_ADM 環境変数を設定します。

```
prompt> set VBROKER_ADM=c:¥my¥adm
```

VBROKER_ADM 環境変数は vregedit ツールを使用してレジストリに設定することもできます。ただし，レジストリと環境変数の両方が設定されている場合，環境変数の設定が有効になります。

2.3.2　UNIX での VBROKER_ADM 環境変数の設定

csh を使用していて，かつ Borland Enterprise Server VisiBroker を/usr/local にインストールしている場合，次のように VBROKER_ADM 環境変数を設定します。

```
prompt> setenv VBROKER_ADM /usr/local/vbroker/adm
```

Bourne シェルを使用していて，かつ Borland Enterprise Server VisiBroker を/usr/local にインストールしている場合，次のように VBROKER_ADM 環境変数を設定します。

```
prompt> VBROKER_ADM=/usr/local/vbroker/adm
prompt> export VBROKER_ADM
```

2.4　OSAGENT_PORT 環境変数の設定

OSAGENT_PORT 環境変数には，スマートエージェントが監視するポート番号を定義します。ポート番号は 5001 から 65535 の範囲で任意の値を設定できますが，デフォルトではスマートエージェントはポート番号 14000 で監視します。

2.4.1　Windows での OSAGENT_PORT 環境変数の設定

スマートエージェントにポート番号 10000 で監視させたい場合は，次のように OSAGENT_PORT 環境変数を設定します。

```
prompt> set OSAGENT_PORT=10000
```

OSAGENT_PORT 環境変数は vregedit ツールを使用してレジストリに設定することもできます。ただし，レジストリと環境変数の両方が設定されている場合，環境変数の設定が有効になります。

2.4.2　UNIX での OSAGENT_PORT 環境変数の設定

csh を使用していて，スマートエージェントにポート番号 10000 で監視させたい場合，次のように OSAGENT_PORT 環境変数を設定します。

```
prompt> setenv OSAGENT_PORT 10000
```

Bourne シェルを使用していて，スマートエージェントにポート番号 10000 で監視させたい場合，次のように OSAGENT_PORT 環境変数を設定します。

```
prompt> OSAGENT_PORT=10000
prompt> export OSAGENT_PORT
```

2.5　ロギング出力

Borland Enterprise Server VisiBroker のツールの多くは，実行中のツールに関する情報を表示するバーボースモードを提供しています。また Borland Enterprise Server VisiBroker ライブラリとリンクしたアプリケーションは，出力を生成することもできます。UNIX のシステムでは，この出力はコンソールに書き込まれます。Windows のシステムでは，この出力は対応するログファイルに書き込まれます。

表 2-1 は Windows で生成できるログファイル名をまとめたものです。

表 2-1　Windows で生成されるログファイル名のまとめ

ファイル名	説明
osagent.log	スマートエージェントによって生成されます。 osagent に-v オプションを指定して起動した場合，このファイルは単調増加します。
viserr.log	スマートエージェントおよび，C++ORB 機能を使用したアプリケーションによって生成されます。
vislog.log	
visout.log	

osagent.log ファイルの出力先は，次の規則に従って決定されます。

1. OSAGENT_LOG_DIR 環境変数が指すディレクトリ
2. <osagent を起動したドライブ>¥vbroker¥log¥
   ディレクトリが存在しない場合は，作成を試みます。
3. カレントディレクトリ

注

osagent.log ファイルの出力先に書き込み権限を与えてください。書き込み権限がないと，osagent は起動できません。

viserr.log，vislog.log，visout.log ファイルの位置は，次の規則に従って決定されます。

1. VBROKER_ADM 環境変数が指すディレクトリ内の log ディレクトリ
   ディレクトリが存在しない場合は，作成を試みます。
2. カレントディレクトリ

3 プロパティの設定

この章では，プロパティテキストファイルやコマンドライン引数で VisiBroker ORB プロパティを設定する方法を説明します。

3.1　概要

VisiBroker ORB は，その特徴を定義する特定のプロパティのセットを持ちます。例えば，vbroker.agent.debug は，VisiBroker アプリケーションとスマートエージェントの間の通信についてのデバッグ情報を出力するよう VisiBroker ORB に指示します。VisiBroker ORB の各プロパティは string，unsigned long，boolean などのあらかじめ定められたデータ型と値を持ちます。例えば，vbroker.agent.enableLocator=false は VisiBroker アプリケーションに対してスマートエージェントとの通信を行わないように指示します。

VisiBroker ORB が初期化処理を開始すると，これらのプロパティの多くが読み込まれます。

VisiBroker ORB プロパティは，アプリケーション起動時にプロパティファイルまたはコマンドライン引数に指定できます。プロパティファイルは次のように記述されています。

コードサンプル 3-1　プロパティファイルの抜粋

```
vbroker.agent.enableLocator=false
```

コマンドライン引数にプロパティを指定すると次のようになります。

コードサンプル 3-2　コマンドライン引数によるプロパティ設定例

C++の場合

```
Server -Dvbroker.agent.port=14999
```

Java の場合

```
vbj Server -vbroker.agent.port 5024
```

プロパティの優先順位については，「3.3　Windows および UNIX プラットフォームでのプロパティの優先順位」を参照してください。Java アプレットに関しては，「3.4　アプレットのプロパティの優先順位」を参照してください。

プロパティは，CORBA::ORB_init()（C++の場合）または ORB.init()（Java の場合）が呼び出されたときに渡された引数から読み込まれます。プロパティマネージャ内のメモリにプロパティが格納されると，ファイルまたはコマンドライン引数は参照されません。

作成したクライアントアプリケーションを java コマンドで起動する場合には，次のプロパティ一覧で示したプロパティ情報を，JavaVM に渡して起動する必要があります。各プロパティ値に指定する値はアプリケーション起動時に-VBJdebug オプションを指定して出力されるデバッグ情報を参考にしてください。

表 3-1　プロパティ一覧

#	プロパティ名
1	javax.rmi.CORBA.StubClass
2	javax.rmi.CORBA.UtilClass
3	javax.rmi.CORBA.PortableRemoteObjectClass
4	org.omg.CORBA.ORBClass
5	org.omg.CORBA.ORBSingletonClass
6	vbroker.agent.port
7	vbroker.orb.admDir

#	プロパティ名
8	java.endorsed.dirs
9	application.home
10	java.class.path

3.2　Borland Enterprise Server VisiBroker のプロパティの設定

Borland Enterprise Server VisiBroker のプロパティは，次の方法で設定できます。

- シェル／コンソールの環境変数
- Windows レジストリ
- コマンドライン引数
- プロパティファイル（ORBpropStorage オプションを使用）
- アプレットのパラメタ（ORB.init の第 1 パラメタ）（Java の場合）
- システムプロパティ（Java の場合）
- プロパティ（Java の場合）

3.2.1　シェル／コンソールの環境変数

環境変数は，プログラムが起動するときに自動的に読み込まれます。環境変数は，次の表に示すとおりにプロパティに変換されます。

表 3-2　シェル／コンソール環境から設定できるプロパティ

環境変数	プロパティ名
OSAGENT_ADDR	vbroker.agent.addr
OSAGENT_ADDR_FILE	vbroker.agent.addrFile
OSAGENT_PORT	vbroker.agent.port
VBROKER_ADM	vbroker.orb.admDir

注

Java の場合，VisiBroker ORB が提供する vbj コマンドやネーミングサービス（nameserv プロセス）などを使用しないで，Java コマンドによって Java アプリケーションを起動したときは，環境変数からプロパティへの変換は行われません。

次に示すのは，環境変数の設定例です。

コードサンプル 3-3　環境変数の設定

UNIX（csh の場合）

```
setenv OSAGENT_PORT 10000
setenv VBROKER_ADM /usr/local/vbroker/adm
```

Windows

```
set OSAGENT_PORT=10000
set VBROKER_ADM=c:¥Borland¥VBroker¥adm
```

注

Borland Enterprise Server VisiBroker の環境変数の設定の詳細については，「2.　環境設定」を参照してください。

3.2.2　Windows レジストリ

環境変数は Windows レジストリに設定できます。レジストリの設定は vregedit ツールを使用して容易に変更できます。

Windows レジストリに設定した環境変数は，通常の環境変数と同じようにシステムプロパティに変換されます（Java の場合）。ただし，環境変数の設定がレジストリの設定よりも優先されます。

3.2.3　コマンドライン引数

プロパティファイルに記載される任意のプロパティは，コマンドライン引数によっても設定できます。CORBA::ORB_init()（C++の場合）または ORB.init()（Java の場合）へコマンドライン引数を渡してください。

コードサンプル 3-4　コマンドラインからのプロパティ設定

C++の場合

```
Server -Dvbroker.agent.port=1024
```

注

ORB で始まる VisiBroker 3.x 形式のプロパティには，-D を指定する必要はありません。例を次に示します。

```
Server -ORBxxxx yyyy
```

Java の場合

```
vbj Server -vbroker.agent.port 1024
```

注

コマンドラインで設定したプロパティは，デフォルトプロパティをオーバーライドします。

コマンドラインの引数は，パラメタとしてアプリケーションクラスに渡されます。例えば，次のコマンドを実行すると，プロパティ vbroker.agent.port に値 15000 が設定され，Server というアプリケーションに渡されます。

```
vbj Server -vbroker.agent.port 15000
```

パラメタリストには二つ以上のプロパティを指定できます。

vbj Server -ORBagent.Port 15000 などのように，ORB または vbroker.で始まるプロパティだけがこの方法で指定できます。また，ORB で始まるプロパティは適切な vbroker.プロパティに変換します（ORBpropStorage は例外です）。

これらの設定は ORB.init の第 1 パラメタとして渡されることで有効になります。

3.2.4　プロパティファイル（ORBpropStorage オプションを使用）

プロパティ名 ORBpropStorage を使用して，コマンドライン引数でシステムプロパティとして指定します。

なお，Java の場合は，アプレットのパラメタで指定する方法もあります。詳細については，「3.2.5　アプレットのパラメタ（ORB.init の第 1 パラメタ）(Java の場合)」を参照してください。

プロパティファイルはテキストファイルで，次の形式でプロパティを 1 行ずつ記述します。

```
<property name>=<property value>
```

VisiBroker ORB はあらかじめ定義されたプロパティ名のセットを持ちます。プロパティ名は大文字と小文字が区別されるため，リスト表示された名前をそのまま正確に入力するよう注意してください。正しい形

式であれば，各プロパティをどの順番で入力してもかまいません。ただし，プロパティを論理グループに分割した方がファイルは読みやすくなります。各グループはコメント行（「#」で始まる行）でラベルづけできます。空行とコメント行は無視されます。

コードサンプル 3-5　プロパティのグループ化

```
#OSAgent properties
vbroker.agent.debug=false
vbroker.agent.addr=null
vbroker.agent.port=14000
vbroker.agent.addrFile=null
vbroker.agent.enableLocator=true
```

プロパティのデータ型には，次の三つがあります。

- string
- unsigned long
- boolean

文字列の値が null であれば，プロパティの値に null と入力できます。

コードサンプル 3-6　null 値の設定

```
vbroker.repository.name=null
```

値が boolean 値の場合は，true または false を入力します。

コードサンプル 3-7　boolean 値の設定

```
vbroker.agent.enableLocator=true
```

プロパティを使用するには，プロパティをファイルに格納し，次のコマンドライン引数によって参照します。

C++の場合

```
-ORBpropStorage filename
```

Java の場合

```
-DORBpropStorage=filename
```

filename は，相対パスまたは絶対パスのどちらでも指定できます。

コードサンプル 4-8　プロパティファイルの指定

C++の場合

```
Server -ORBpropStorage myprops
```

Java の場合

```
vbj -DORBpropStorage=myprops Server
```

注

Java の場合，-D は必須であり，-D に続く文字列がプロパティであることを示します。

3.2.5　アプレットのパラメタ（ORB.init の第 1 パラメタ）（Java の場合）

アプレットのパラメタは，HTML の<param>タグを使って指定します。

```
<applet ...>
   <param name="vbroker.agent.port" value="15000">
</applet>
```

一つの param タグで一つのプロパティを定義します。ここでは，ORB で始まるプロパティから適切な vbroker.プロパティへの変換は行いません。これらの設定は ORB.init の第 1 パラメタとして渡されます。

コードサンプル 3-9　HTML からのプロパティの設定

```
...
<applet class="MyCORBAApplet.class"
      codebase="."
      archive="my Lib.jar">
   <param name="vbroker.orb.alwaysTunnel" value="true">
   <param name="vbroker.orb.gatekeeper.ior" value="">
</applet>
...
```

3.2.6　システムプロパティ（Java の場合）

-D を使って定義したプロパティは，JavaVM によってシステムプロパティとして設定されます。

```
vbj -Dvbroker.agent.port=15000 Server
```

-D の代わりに，次の形式でも指定できます。

```
-J-D<name>=<value>
-VBJprop <name>=<value>
```

コマンドラインの引数と異なる形式を使用できるのは，VisiBroker ではなく JavaVM によって構文解析されるためです。環境変数とレジストリの設定はシステムプロパティに変換され，JavaVM に渡されます。

注

VisiBroker ORB が提供する vbj コマンドやネーミングサービス（nameserv プロセス）などを使用しないで，Java コマンドによって Java アプリケーションを起動した場合は，環境変数とレジストリの設定からシステムプロパティへの変換は行われません。

3.2.7　プロパティ（Java の場合）

ORB.init は java.util.Properties 型のパラメタを受け取ります。このパラメタを使用して，プロパティのセットを渡すことができます。

3.3　Windows および UNIX プラットフォームでのプロパティの優先順位

Windows および UNIX プラットフォームでの，C++および Java のプロパティは次の優先順位で設定されます。

1. コマンドライン引数（C++および Java の場合）
2. システムプロパティ（Java の場合）
3. プロパティファイル（ORBpropStorage オプションを使用）（C++および Java の場合）
4. プログラムで ORB.init の第 2 パラメタに渡されるプロパティ（Java の場合）

3.4　アプレットのプロパティの優先順位

アプレットのプロパティは次の優先順位で設定されます。

1. アプレットのパラメタ（ORB.init の第 1 パラメタ）
2. プロパティファイル（ORBpropStorage オプションを使用）
   URL で指定する場合もあります。
3. プログラムで ORB.init の第 2 パラメタに渡されるプロパティ

3.5　Borland Enterprise Server VisiBroker プロパティ

Borland Enterprise Server VisiBroker で使用できるプロパティのリストについては，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「Borland Enterprise Server VisiBroker プロパティ」を参照してください。

サーバエンジンプロパティについては，「7.7.1　サーバエンジンプロパティの設定」を参照してください。

4 Borland Enterprise Server VisiBroker によるサンプルアプリケーションの開発

この章では, C++および Java 用のオブジェクトベースの分散アプリケーションの開発手順を，サンプルアプリケーションを使用して説明します。

4.1　開発手順

この節では，サンプルアプリケーションのコードの位置と開発手順の概要について説明します。

4.1.1　サンプルアプリケーションのパッケージの位置

サンプルアプリケーションの C++および Java のコードは，Borland Enterprise Server VisiBroker のパッケージがインストールされている examples/vbe/basic/bank_agent ディレクトリ（Windows の場合は，"/"を"¥"に読み替えてください）下の bank_agent.html ファイルにあります。各パッケージの位置がわからない場合は，システム管理者に問い合わせてください。

4.1.2　開発手順の概要

Borland Enterprise Server VisiBroker で分散アプリケーションを開発する場合，まずアプリケーションに必要なオブジェクトを識別する必要があります。図 4-1 に，サンプルのバンクアプリケーションを開発する手順を示します。バンクサンプルを開発する手順の概要を次に示します。

1. IDL（インタフェース定義言語）を使用して各オブジェクトの仕様を記述します。

   IDL は，オブジェクトが提供するオペレーションとその起動方法を指定するために実装者が使用する言語です。この例では，balance()メソッドを使用して Account インタフェースを，open()メソッドを使用して AccountManager インタフェースをそれぞれ IDL で定義します。

2. IDL コンパイラを使用してクライアントのスタブコードとサーバ POA のサーバントコードを生成します。

   手順 1.のインタフェース仕様で，idl2java（Java）コンパイラまたは idl2cpp（C++）コンパイラを使用して，クライアント側のスタブ（Account オブジェクトメソッドと AccountManager オブジェクトメソッドとのインタフェースを提供する）とサーバ側のクラス（リモートオブジェクトのインプリメンテーションのクラスを提供する）を作成します。

3. クライアントプログラムコードを記述します。

   クライアントプログラムのインプリメンテーションを完成させるには，VisiBroker ORB を初期化し，Account オブジェクトと AccountManager オブジェクトにバインドし，これらのオブジェクトのメソッドを呼び出し，残高を表示します。

4. サーバオブジェクトコードを記述します。

   サーバオブジェクトコードのインプリメンテーションを完成させるには，AccountPOA クラスと AccountManagerPOA クラスから派生を行い，interface のメソッドのインプリメンテーションを提供し，サーバの main ルーチンをインプリメントする必要があります。

5. クライアントとサーバコードをコンパイルします。

   C++の場合

   クライアントプログラムを生成するには，クライアントプログラムコードをクライアントスタブとコンパイルしてリンクしてください。

   Account サーバを生成するには，サーバオブジェクトコードをサーバスケルトンとコンパイルしてリンクしてください。

   Java の場合

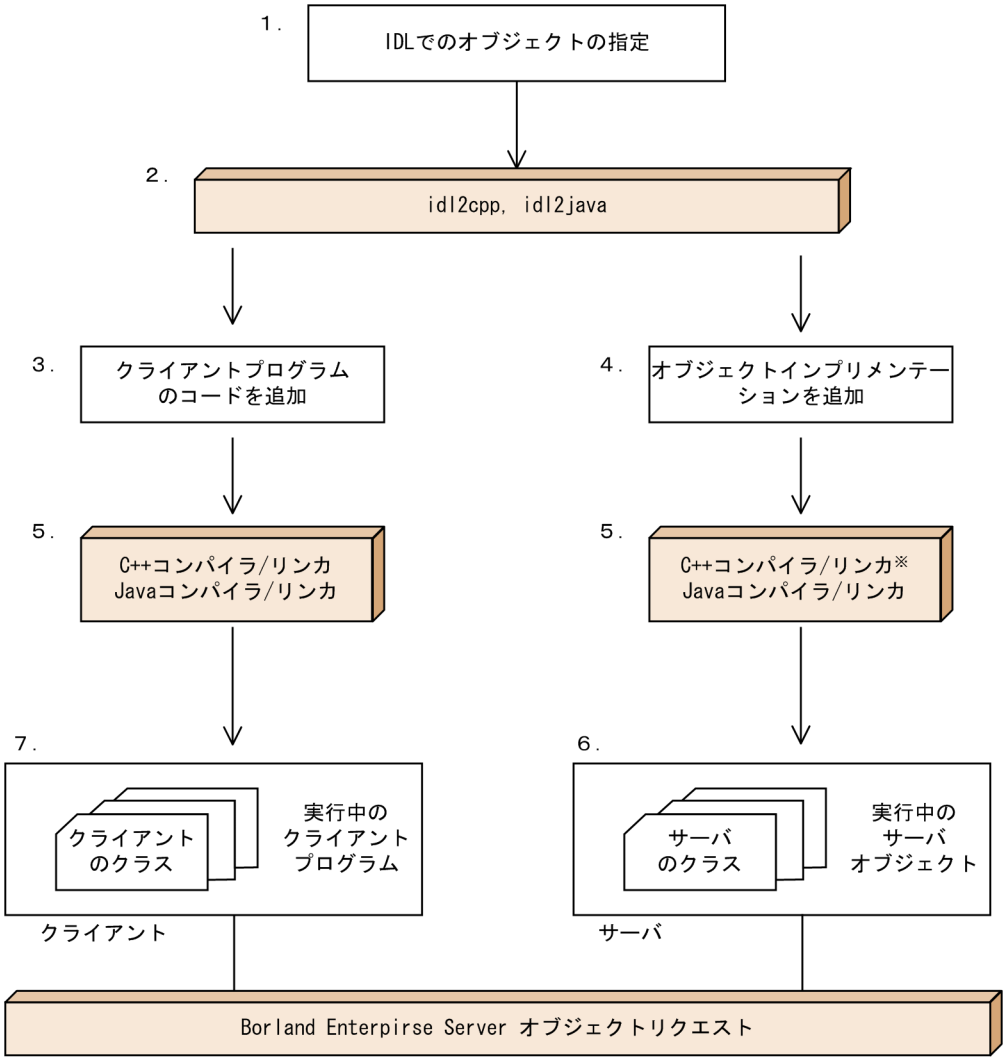
   クライアントプログラムを生成するには，クライアントプログラムコードをクライアントスタブとコンパイルしてください。

   Account サーバを生成するには，サーバオブジェクトコードをサーバスケルトンとコンパイルしてください。

6. サーバを起動します。

7. クライアントプログラムを実行します。

図 4-1　サンプルのバンクアプリケーションの開発



注※：C++でアプリケーションを作成している場合，サーバオブジェクト
のコードをコンパイルしてリンクする必要があります。

ORB は hv または HV で始まる名称の関数，クラス，グローバル変数，環境変数を使用します。この名称で始まる関数，クラス，グローバル変数，環境変数などを使用した開発はしないでください。

4.2　手順 1：オブジェクトインタフェースの定義

Borland Enterprise Server VisiBroker によるアプリケーション生成の最初の手順は，OMG の IDL（インタフェース定義言語）を使用して，使用するすべてのオブジェクトとそのインタフェースを指定することです。IDL はさまざまなプログラミング言語にマッピングできます。C++および Java のそれぞれのマッピングについては，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「IDL から C++言語へのマッピング」および「IDL から Java へのマッピング」の記述を参照してください。

次に idl2cpp（C++）または idl2java（Java）コンパイラを使用して IDL 指定からスタブルーチンとサーバントコードを生成します。クライアントプログラムはスタブルーチンを使用してオブジェクトのオペレーションを呼び出します。自分で記述したコードとともにサーバントコードを使用して，オブジェクトをインプリメントするサーバを作成してください。クライアントとオブジェクト用のコードは一度完成すれば，クライアント Java アプレットまたはアプリケーションとオブジェクトサーバを作成するために C++コンパイラまたは Java コンパイラへの入力として使用されます。

4.2.1　IDL での Account インタフェースの記述

IDL は C++と似た構文を持ち，モジュール，インタフェース，データ構造などの定義に使用できます。

IDL サンプル 4-1 は，サンプル bank_agent の Bank.idl ファイルの内容です。Account インタフェースは，現在の残高を取得するために C++で一つのメソッド，Java で一つのメソッドを提供します。AccountManager インタフェースは，ユーザのアカウントが存在しなければそれを生成します。

IDL サンプル 4-1　Bank.idl ファイルは Account インタフェース定義を提供する

```
module Bank{
   interface Account {
      float balance();
   };
   interface AccountManager {
      Account open(in string name);
   };
};
```

4.3　手順 2：クライアントスタブとサーバサーバントの生成

IDL で作成するインタフェース定義は，Borland Enterprise Server VisiBroker の idl2cpp コンパイラがクライアントプログラム用の C++スタブルーチンとオブジェクトインプリメンテーション用のスケルトンコードを生成するために，または idl2java コンパイラがクライアントプログラム用の Java クラスとオブジェクトインプリメンテーション用のスケルトンコードを生成するために使用します。

- C++の場合
  スタブルーチンは，すべてのメンバ関数呼び出しでクライアントプログラムが使用します。
- Java の場合
  Java クラスは，すべてのメソッド呼び出しでクライアントプログラムが使用します。

自分で記述したコードとともにスケルトンコードを使用して，C++および Java 用のオブジェクトをインプリメントするサーバを生成してください。

- C++の場合
  クライアントプログラムとサーバオブジェクト用のコードが完成すると，このコードを C++コンパイラとリンカへの入力として使用してクライアントとサーバを作成します。
- Java の場合
  クライアントプログラムとサーバオブジェクト用のコードが完成すると，このコードを Java コンパイラへの入力として使用してクライアントとサーバの実行可能クラスを作成します。

クライアントスタブとサーバサーバントを生成する手順については，「4.1　開発手順」を参照してください。

Bank.idl ファイルは特別な処理を必要としないので，次のコマンドでファイルをコンパイルできます。

- C++の場合
  ```
  prompt> idl2cpp Bank.idl
  ```
- Java の場合
  ```
  prompt> idl2java Bank.idl
  ```

idl2cpp および idl2java コンパイラに関するコマンドラインオプションの詳細については，「11.　IDL の使用」を参照してください。

4.3.1　IDL コンパイラが作成するファイル

(1)　C++の場合

idl2cpp コンパイラは Bank.idl ファイルから次の四つのファイルを生成します。

Bank_c.hh
Account および AccountManager クラスの定義を含みます。

Bank_c.cpp
クライアントが使用する内部スタブルーチンを含みます。

Bank_s.hh
AccountPOA および AccountManagerPOA サーバントクラスの定義を含みます。

Bank_s.cpp
サーバが使用する内部ルーチンを含みます。

ユーザは Bank_c.hh ファイルと Bank_c.cpp ファイルを使用してクライアントアプリケーションを構築します。Bank_s.hh ファイルと Bank_s.cpp ファイルはサーバオブジェクトを構築するために使用します。生成されたファイルとソースファイルとを区別するために，生成されたファイルにはすべて.cpp か.hh という拡張子が付けられます。

Windows
idl2cpp コンパイラから生成されたファイルのデフォルトの拡張子は.cpp ですが，Borland Enterprise Server VisiBroker の例に対応する Makefile は，-src_suffix を使用して，出力を指定の拡張子に変更します。

(2) Java の場合

Java では，ファイルごとに一つのパブリックインタフェースまたはクラスだけ使用できるので，IDL ファイルをコンパイルすると複数の.java ファイルを生成します。このようなファイルは生成された Bank というサブディレクトリに格納されます。Bank ディレクトリは IDL で指定されたモジュール名であり，生成済みのファイルが属するパッケージです。生成される.java ファイルのリストを次に示します。

_AccountManagerStub.java
クライアント側の AccountManager オブジェクトのスタブコードです。

_AccountStub.java
クライアント側の Account オブジェクトのスタブコードです。

Account.java
Account インタフェース宣言です。

AccountHelper.java
ユーティリティメソッドを定義する AccountHelper クラスを宣言します。

AccountHolder.java
Account オブジェクトを渡すためのホルダを提供する AccountHolder クラスを宣言します。

AccountManager.java
AccountManager インタフェース宣言です。

AccountManagerHelper.java
ユーティリティメソッドを定義する AccountManagerHelper クラスを宣言します。

AccountManagerHolder.java
AccountManager オブジェクトを渡すためのホルダを提供する AccountManagerHolder クラスを宣言します。

AccountManagerOperation.java
このインタフェースは，Bank.idl ファイルで AccountManager インタフェースに定義されたメソッドシグニチャを宣言します。

AccountManagerPOA.java
サーバ側の AccountManager オブジェクトインプリメンテーション用の POA サーバントコード（インプリメンテーションベースコード）です。

AccountManagerPOATie.java

サーバ側の AccountManager オブジェクトを tie 機能を使用してインプリメントするためのクラスです。tie 機能の詳細については，「9.　tie 機能の使用」を参照してください。

AccountOperations.java

このインタフェースは，Bank.idl ファイルで Account インタフェースに定義されたメソッドシグニチャを宣言します。

AccountPOA.java

サーバ側の Account オブジェクトインプリメンテーション用の POA サーバントコード（インプリメンテーションベースコード）です。

AccountPOATie.java

サーバ側の Account オブジェクトを tie 機能を使用してインプリメントするためのクラスです。tie 機能の詳細については，「9.　tie 機能の使用」を参照してください。

Helper クラス，Holder クラス，および Operations クラスの詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「生成されるインタフェースとクラス（Java）」の記述を参照してください。

4.4　手順 3：クライアントのインプリメント

bank クライアントをインプリメントするために使用するクラスの多くは，前述の例で説明したように，idl2cpp（C++）が生成した Bank コード，または idl2java（Java）コンパイラが生成した Bank パッケージに入っています。

bank_agent ディレクトリに格納されている Client.C ファイル（C++）および Client.java ファイル（Java）はこの例をわかりやすく説明します。通常は，このファイルは自分で作成してください。

4.4.1　Client.C

Client プログラムは，bank のアカウントの現在の残高を取得するクライアントアプリケーションをインプリメントします。bank クライアントプログラムは次の手順を実行します。

1. VisiBroker ORB を初期化します。
2. AccountManager オブジェクトにバインドします。
3. bind()メソッドが返すオブジェクトリファレンスを使用して，Account の残高を取得します。
4. Account オブジェクトで balance を呼び出して，残高を取得します。

コードサンプル 4-1　クライアント側のプログラム（C++）

```
#include "Bank_c.hh"
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      // Get the manager Id
      PortableServer::ObjectId_var managerId =
            PortableServer::string_to_ObjectId("BankManager");
      // Locate an account manager. Give the full POA name and
      // the servant ID.
      Bank::AccountManager_ptr manager =
      Bank::AccountManager::_bind(
            "/bank_agent_poa", managerId);
      // use argv[1] as the account name, or a default.
      const char* name = argc > 1 ? argv[1] : "Jack B. Quick";
      // Request the account manager to open a named account.
      Bank::Account_ptr account = manager->open(name);
      // Get the balance of the account.
      float balance = account->balance();
      // Print out the balance.
      cout << "The balance in "<< name << "'s account is $"
            << balance << endl;
   } catch(const CORBA::Exception& e) {
     cerr << e << endl;
   }
}
```

4.4.2　Client.java

Client クラスは，bank のアカウントの現在の残高を取得するクライアントアプリケーションをインプリメントします。bank クライアントプログラムは次の手順を実行します。

1. VisiBroker ORB を初期化します。
2. AccountManager オブジェクトにバインドします。
3. AccountManager オブジェクトで open を呼び出して，Account オブジェクトを取得します。
4. Account オブジェクトで balance を呼び出して，残高を取得します。

コードサンプル 4-2　クライアント側のプログラム（Java）

```
public class Client {
   public static void main(String[ ] args){
      // Initialize the ORB.
      org.omg.CORBA.ORB orb =
                        org.omg.CORBA.ORB.init(args, null);
      // Get the manager Id
      byte[ ] managerId = "BankManager".getBytes();
      // Locate an account manager.
      // Give the full POA name and the servant ID.
      Bank.AccountManager manager =
            Bank.AccountManagerHelper.bind(orb,
                                           "/bank_agent_poa",
                                            managerId);
      // use args[0] as the account name, or a default.
      String name = args.length > 0 ? args[0] : "Jack B. Quick";
      // Request the account manager to open a named account.
      Bank.Account account = manager.open(name);
      // Get the balance of the account.
      float balance = account.balance();
      // Print out the balance.
      System.out.println("The balance in " + name +
                         "'s account is $" + balance);
   }
}
```

4.4.3　AccountManager オブジェクトへのバインド

クライアントプログラムは，open（String name）メソッドを呼び出す前に，まず bind()メソッドを使用して AccountManager オブジェクトをインプリメントするサーバとのコネクションを確立する必要があります。

C++の場合

bind()メソッドのインプリメンテーションは，idl2cpp コンパイラが自動的にインプリメントします。bind()メソッドは，サーバを探してコネクションを確立するよう VisiBroker ORB にリクエストします。

Java の場合

bind()メソッドのインプリメンテーションは，idl2java コンパイラが自動的に生成します。bind()メソッドは，サーバを探してコネクションを確立するよう VisiBroker ORB にリクエストします。

サーバの探索に成功し，コネクションが確立されると，サーバの AccountManagerPOA オブジェクトに対応するプロキシオブジェクトが作成されます。AccountManager オブジェクトのオブジェクトリファレンスはクライアントプログラムに返されます。

4.4.4　Account オブジェクトの取得

次に，クライアントクラスは，指定された顧客名に対する Account オブジェクトのオブジェクトリファレンスを取得するために AccountManager オブジェクトの open()メソッドを呼び出す必要があります。

4.4.5　残高の取得

クライアントプログラムが Account オブジェクトとのコネクションを確立すると，balance()メソッドを使用して残高を取得できます。クライアント側の balance()メソッドは実際には idl2cpp または idl2java コンパイラが生成したスタブであり，リクエストが必要とするデータをすべて集め，それをサーバオブジェクトに送ります。

4.4.6　AccountManagerHelper.java（Java）

このファイルは Bank パッケージに入っています。このファイルは AccountManagerHelper オブジェクトを格納し，このオブジェクトをインプリメントするサーバにバインドするための複数のメソッドを定義します。bind()クラスメソッドは，指定された POA マネージャとコンタクトしてオブジェクトを解決します。サンプルアプリケーションでは，オブジェクト名を受け付ける bind()メソッドを使用していますが，クライアントはオプションで特定のホストおよび特別なバインドオプションを指定できます。Helper クラスの詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「Helper クラス」の記述を参照してください。

コードサンプル 4-3　AccountManagerHelper.java ファイルの一部

```
package Bank;
public final class AccountManagerHelper {
  . . .
  public static Bank.AccountManager bind(
                                org.omg.CORBA.ORB orb){
    return bind(orb, null, null, null);
    . . .
  }
}
```

4.4.7　そのほかのメソッド

これまで説明してきたメソッド以外にも，クライアントプログラムが AccountManager オブジェクトリファレンスを処理できるようにするメソッドが提供されています。

サンプルのクライアントアプリケーションではこれらのメソッドはほとんど使用していません。これらのメソッドの詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」を参照してください。

4.5　手順 4：サーバのインプリメント

クライアントの場合と同様に，bank サーバをインプリメントするために使用するクラスの多くは，idl2cpp コンパイラが生成するヘッダファイル（C++）または idl2java コンパイラが生成する Bank パッケージ（Java）に入っています。Server.C および Server.java ファイルは，この例をわかりやすく説明するために用意されたサーバインプリメンテーションです。通常は，プログラマがこのファイルを作成します。

4.5.1　サーバプログラム

このファイルは，サンプルの bank でサーバ側の Server クラスをインプリメントします。コードサンプル 4-4 は C++のサーバ側プログラムの例です。コードサンプル 4-5 は Java のサーバ側プログラムの例です。サーバプログラムは次のように動作します。

- ORB を初期化します。
- 必要なポリシーでポータブルオブジェクトアダプタを作成します。
- アカウントマネージャのサーバントオブジェクトを作成します。
- サーバントオブジェクトを活性化します。
- POA マネージャ（と POA）を活性化します。
- 入力リクエストを待ちます。

コードサンプル 4-4　Server.C プログラム（C++）

```
#include ”BankImpl.h”
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

      // get a reference to the rootPOA
      PortableServer::POA_var rootPOA =
            PortableServer::POA::_narrow(
            orb->resolve_initial_references(”RootPOA”));

      CORBA::PolicyList policies;
      policies.length(1);
      policies[(CORBA::ULong)0] =
            rootPOA->create_lifespan_policy(
                  PortableServer::PERSISTENT);

      // get the POA Manager
      PortableServer::POAManager_var poa_manager =
            rootPOA->the_POAManager();

      // Create myPOA with the right policies
      PortableServer::POA_var myPOA = rootPOA->create_POA(
            ”bank_agent_poa”,
            poa_manager, policies);

      // Create the servant
      AccountManagerImpl managerServant;

      // Decide on the ID for the servant
      PortableServer::ObjectId_var managerId =
            PortableServer::string_to_ObjectId(”BankManager”);

      // Activate the servant with the ID on myPOA
      myPOA->activate_object_with_id(
            managerId, &managerServant);

      // Activate the POA Manager
      poa_manager->activate();
```

```
      cout << myPOA->servant_to_reference(&managerServant) <<
            "is ready" << endl;

      // Wait for incoming requests
      orb->run();
   } catch(const CORBA::Exception& e) {
      cerr << e << endl;
      return 1;
   }
   return 0;
}
```

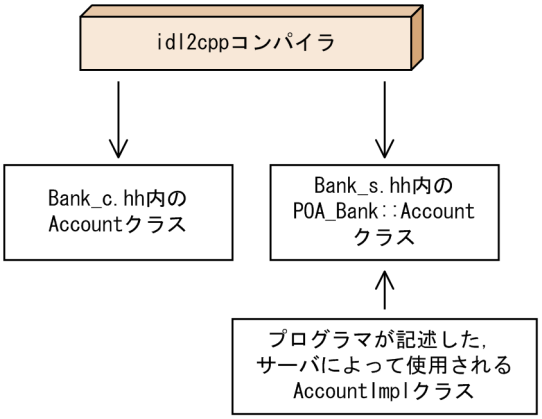
コードサンプル 4-5　Server.java プログラム（Java）

```
public class Server {
   public static void main(String[ ] args) {
      try {
         // Initialize the ORB.
         org.omg.CORBA.ORB orb =
                        org.omg.CORBA.ORB.init(args,null);
         // get a reference to the rootPOA
         POA rootPOA = POAHelper.narrow(
                orb.resolve_initial_references("RootPOA"));
         // Create policies for our persistent POA
            org.omg.CORBA.Policy[ ] policies = {
            rootPOA.create_lifespan_policy(
                              LifespanPolicyValue.PERSISTENT)
         };
         // Create myPOA with the right policies
         POA myPOA = rootPOA.create_POA("bank_agent_poa",
                                        rootPOA.the_POAManager(),
                                        policies );
         // Create the servant
         AccountManagerImpl managerServant =
                                    new AccountManagerImpl();
         // Decide on the ID for the servant
         byte[ ] managerId = "BankManager".getBytes();
         // Activate the servant with the ID on myPOA
         myPOA.activate_object_with_id(
                                   managerId,managerServant);
         // Activate the POA manager
         rootPOA.the_POAManager().activate();
         System.out.println(
                myPOA.servant_to_reference(managerServant) +
                                                "is ready.");
         // Wait for incoming requests
         orb.run();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

4.5.2　Account クラス階層について（C++）

ユーザがインプリメントした Account クラスは，POA_Bank::Account クラスから派生します。POA_Bank::Account クラスは idl2cpp コンパイラが生成したものです。Bank_c.hh ファイルに定義してある POA_Bank::Account クラスの定義を見ると，Account クラスから派生していることがわかります。クラス階層を図 4-2 に示します。

図 4-2　AccountImpl インタフェースのクラス階層



4.6　手順５：サンプルプログラムのビルド

C++の場合

ユーザが作成した Client.C と生成された Bank_c.cpp ファイルがコンパイルされ，リンクされるとクライアントプログラムが作成されます。ユーザが作成した Server.C ファイルは，生成された Bank_s.cpp ファイルおよび Bank_c.cpp ファイルと一緒にコンパイルされ，リンクされて Bank アカウントサーバが作成されます。クライアントプログラムとサーバは，両方とも VisiBroker ORB ライブラリとリンクしなければなりません。

Borland Enterprise Server VisiBroker をインストールしたディレクトリの examples ディレクトリには，このサンプルとほかの Borland Enterprise Server VisiBroker のサンプルである Makefile.cpp（C++）または vbmake.bat（Java）が入っています。

C++の場合

サンプルのディレクトリには stdmk（UNIX 用）か stdmk_nt（Windows 用）という名前のファイルがあり，ファイルの位置と Makefile が使用する変数設定を定義しています。

注（C++の場合）

コンパイラが指定のフラグをサポートしていなければ，stdmk ファイルか stdmk_nt ファイルをカスタマイズする必要があります。

4.6.1　サンプルのコンパイル

(1)　Windows

Borland Enterprise Server VisiBroker が C:¥vbroker にインストールされている場合，サンプルをコンパイルするには次のコマンドを入力します。

C++の場合

```
prompt> C:
prompt> cd ¥vbroker¥examples¥vbe¥basic¥bank_agent
prompt> nmake -f Makefile.cpp
```

Visual C++のコマンド nmake は，idl2cpp コンパイラを実行してから各ファイルをコンパイルします。

Java の場合

```
prompt> C:
prompt> cd ¥vbroker¥examples¥vbe¥basic¥bank_agent
prompt> vbmake
```

vbmake コマンドは，idl2java コンパイラを実行してから各ファイルをコンパイルするバッチファイルです。

nmake（C++）または vbmake（Java）の実行中に何らかの問題が生じた場合は，PATH 環境変数が Borland Enterprise Server VisiBroker をインストールした bin ディレクトリをポイントしていることを確認してください。また C++の場合，Borland Enterprise Server VisiBroker をインストールしたディレクトリに VBROKERDIR 環境変数を設定してみてください。

(2)　UNIX

Borland Enterprise Server VisiBroker が/opt/vbroker にインストールされている場合，サンプルをコンパイルするには次のコマンドを入力します。

```
prompt> cd /opt/vbroker/examples/vbe/basic/bank_agent
```

C++の場合

```
prompt> make cpp
```

Java の場合

```
prompt> make java
```

このサンプルの make は標準の UNIX の機能です。PATH に make がなければ，システム管理者に問い合わせてください。

4.7　手順 6：サーバの起動とサンプルの実行

これでクライアントプログラムとサーバインプリメンテーションがコンパイルできたので，Borland Enterprise Server VisiBroker アプリケーションを実行するための準備ができました。

4.7.1　スマートエージェントの起動

Borland Enterprise Server VisiBroker クライアントプログラムやサーバインプリメンテーションを実行する前に, まずローカルネットワーク上の一つ以上のホストでスマートエージェントを起動する必要があります。

スマートエージェントを起動するための基本コマンドは次のとおりです。

```
prompt> osagent
```

Windows を実行中で，スマートエージェントを NT サービスとして起動したければ，インストール時に ORB サービスを NT サービスとして登録する必要があります。サービスを登録したら，サービスコントロールパネルを介してスマートエージェントを NT サービスとして起動できます。

スマートエージェントについては，「12.　スマートエージェントの使用」を参照してください。

4.7.2　サーバの起動

(1)　Windows

コマンドプロンプトウィンドウを開き，次の DOS コマンドを使用してサーバを起動します。

C++の場合

```
prompt> Server
```

Java の場合

```
prompt> vbj Server
```

(2)　UNIX

次のように入力して Account サーバを起動します。

C++の場合

```
prompt> Server&
```

Java の場合

```
prompt> vbj Server&
```

4.7.3　クライアントの実行

(1)　Windows

別のコマンドプロンプトウィンドウを開き，次の DOS コマンドを使用してクライアントを起動します。

C++の場合

```
prompt> Client
```

Java の場合

```
prompt> vbj Client
```

次に示すような出力がされたかどうかを確認してください(アカウントの残高はランダムに計算されます)。

```
The balance in the account in $168.38.
```

(2)　UNIX

次のように入力してクライアントプログラムを起動します。

C++の場合

```
prompt> Client
```

Java の場合

```
prompt> vbj Client
```

次に示すような出力がされたかどうかを確認してください(アカウントの残高はランダムに計算されます)。

```
The balance in the account in $168.38.
```

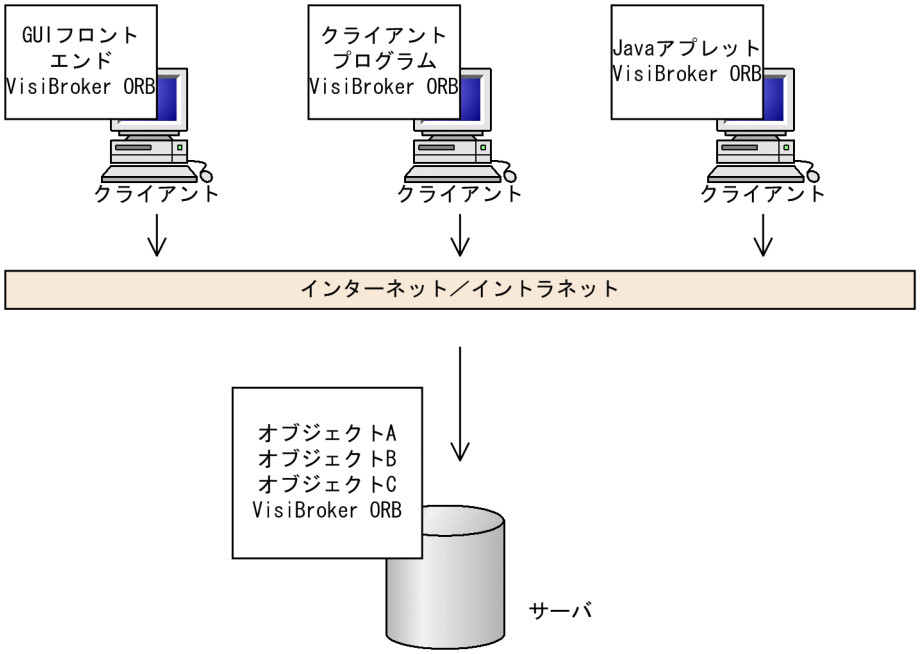
4.8　Borland Enterprise Server VisiBroker を使用したアプリケーションの配置

Borland Enterprise Server VisiBroker は配置フェーズでも使用します。このフェーズは，テストしてリリース準備ができているクライアントプログラムまたはサーバアプリケーションを，開発者が作成した場合に発生します。この時点で，システム管理者はクライアントプログラムをエンドユーザのデスクトップに配置したり，サーバアプリケーションをサーバクラスマシンに配置したりする準備ができています。

配置については，VisiBroker ORB はフロントエンドでクライアントプログラムをサポートします。クライアントプログラムを実行する各マシンに VisiBroker ORB をインストールしなければなりません。同じサーバマシン上の（VisiBroker ORB を使用する）複数のクライアントが VisiBroker ORB を共有します。また，VisiBroker ORB は，ミドルティア上のサーバアプリケーションもサポートします。サーバアプリケーションを実行する各マシンに ORB をインストールしなければなりません。同じサーバマシン上の（VisiBroker ORB を使用する）サーバアプリケーションまたはオブジェクトで VisiBroker ORB を共有します。クライアントには GUI フロントエンド，アプレット，およびクライアントプログラムが使用できます。サーバインプリメンテーションにはミドルティア上のビジネスロジックが含まれます。

図 4-3 に，VisiBroker ORB と一緒に配置されたクライアントプログラムとサーバプログラムを示します。

図 4-3　VisiBroker ORB と一緒に配置されたクライアントプログラムとサーバプログラム



4.8.1　Borland Enterprise Server VisiBroker アプリケーション

(1)　アプリケーションの配置

Borland Enterprise Server VisiBroker を使用して開発したアプリケーションを配置するには，まずアプリケーションを実行するホストでランタイム環境を設定して，必要なサポートサービスがローカルネットワークで使用できる状態であることを確認してください。

C++の場合

Borland Enterprise Server VisiBroker（C++）を使用して開発したアプリケーションに必要なランタイム環境には次のようなコンポーネントが含まれます。

- Borland Enterprise Server VisiBroker をインストールした bin サブディレクトリにある Borland Enterprise Server VisiBroker C++ライブラリ
- アプリケーションに必要なサポートサービスの可用性

VisiBroker ORB ライブラリは，配置されたアプリケーションを実行するホストにインストールしなければなりません。このライブラリは，アプリケーション環境用の PATH にインストールしなければなりません。

Java の場合

Borland Enterprise Server VisiBroker（Java）を使用して開発したアプリケーションに必要なランタイム環境には，次のようなコンポーネントが含まれます。

- Java ランタイム環境
- Borland Enterprise Server VisiBroker をインストールした lib サブディレクトリの vbjorb.jar ファイルにあるアーカイブした Borland Enterprise Server VisiBroker の Java パッケージ
- アプリケーションに必要なサポートサービスの可用性

Java ランタイム環境は，配置されたアプリケーションを実行するホストにインストールしなければなりません。Borland Enterprise Server VisiBroker の Java パッケージは，配置されたアプリケーションを実行するホストにインストールしなければなりません。

(a)　環境変数

配置されたアプリケーションが特定のホストのスマートエージェント（osagent）を使用する場合，アプリケーションを実行する前に OSAGENT_ADDR 環境変数を設定しなければなりません。vbj 実行形式ファイルを使用する場合は，環境変数が自動的に設定されます。ただし，次のコマンドライン引数を使用する場合，コマンドライン引数の指定が環境変数の指定より優先されます。

C++の場合

-ORBagentAddr コマンドライン引数または vbroker.agent.addr コマンドライン引数を使用してホスト名または IP アドレスを指定できます。

Java の場合

vbroker.agent.addr（Java）コマンドライン引数を使用してホスト名または IP アドレスを指定できます。

配置されたアプリケーションがスマートエージェント（osagent）との通信時に特定の UDP ポートを使用する場合，アプリケーションを実行する前に OSAGENT_PORT 環境変数を設定しなければなりません。次のコマンドライン引数を使用する場合，コマンドライン引数の指定が環境変数の指定より優先されます。

C++の場合

-ORBagentPort コマンドライン引数または vbroker.agent.port コマンドライン引数を使用して IP ポート番号を指定できます。

Java の場合

vbroker.agent.port コマンドライン引数を使用して UDP ポート番号を指定できます。

表 4-2 に C++アプリケーションのコマンドライン引数を示します。また，表 4-3 に Java アプリケーションのコマンドライン引数を示します。

環境変数の詳細については，「2.　環境設定」を参照してください。

(b)　サポートされているサービスを使用するには

スマートエージェント（osagent）は，配置されたアプリケーションを実行するネットワークのどこかで実行しなければなりません。このように，実行するアプリケーションの要件によっては，ほかの Borland Enterprise Server VisiBroker のプログラムを起動する必要があります。これらのサービスには，表 4-1 のものが含まれます。

表 4-1　実行するアプリケーション要件によって確認する必要があるケース

サービス	必要なケース
オブジェクト活性化デーモン（oad）	アプリケーションが，オンデマンドで起動しなければならないオブジェクトをインプリメントするサーバである場合。
インタフェースリポジトリ（irep）	アプリケーションが，動的スケルトンインタフェースまたは動的インプリメンテーションインタフェースを使用する場合。これらのインタフェースの説明については，「16.　インタフェースリポジトリの使用」を参照してください。
ゲートキーパー	アプリケーションを，ネットワークセキュリティのファイアウォールを使用した環境で実行する必要がある場合。

(2)　vbj の使用（Java）

vbj コマンドを使用してアプリケーションを起動し，アプリケーションの動作を制御するコマンドライン引数を入力できます。

```
vbj -Dvbroker.agent.port=10000 <class>
```

(3)　アプリケーションの実行（C++）

ここで，クライアントプログラムとサーバインプリメンテーションをコンパイルしたので，最初の Borland Enterprise Server VisiBroker アプリケーションを実行する準備ができました。

Borland Enterprise Server VisiBroker クライアントプログラムやサーバインプリメンテーションを実行する前に，まず，ローカルネットワーク上の最低一つのホストでスマートエージェントを起動する必要があります。スマートエージェントについては，「4.7.1　スマートエージェントの起動」を参照してください。

(4)　クライアントアプリケーションの実行

クライアントアプリケーションは VisiBroker ORB オブジェクトを使用するアプリケーションですが，自身の VisiBroker ORB オブジェクトをほかのクライアントアプリケーションには提供しません。

C++の場合

クライアントアプリケーションに指定できるコマンドライン引数を表 4-2 に示します。この引数はサーバにも適用できます。

表 4-2　クライアントアプリケーションのコマンドライン引数（C++）

オプション	説明
-ORBagent <0|1>	VisiBroker の_bind()時に，スマートエージェントがサーバを探索するかどうかを指定します。 0 を設定した場合，探索しません。

オプション	説明
-ORBagent <0|1>	1 を設定した場合，探索します。 デフォルトは 1 です。
-ORBagentAddr <hostname|ip_address>	このクライアントが使用するスマートエージェントを実行しているホストのホスト名または IP アドレスを指定します。 hostname を設定した場合，ホスト名を指定します。 ip_address を設定した場合，IP アドレスを指定します。 このオプションの指定を省略した場合や，自ホストまたは agentaddr ファイルで指定したホストからスマートエージェントを探索できない場合は，ブロードキャストメッセージでスマートエージェントを見つけます。
-ORBagentPort <port_number>	スマートエージェントのポート番号を指定します。複数の VisiBroker ORB ドメインが必要な場合に有用です。 指定を省略した場合，14000 番で動作します。
-ORBconnectionMax <#>	コネクションの最大数を指定します。 指定を省略した場合，無制限にコネクションを許可します。
-ORBconnectionMaxIdle <#>	コネクションが非アクティブな状態の最大監視時間を秒単位で指定します。最大監視時間を経過してもコネクションが非アクティブなままの場合，VisiBroker がコネクションを終了します。 このオプションはインターネットアプリケーションで設定します。 0 を設定した場合，監視しません。 デフォルトは 0 です。
-ORBDefaultInitRef	デフォルトの初期リファレンスを指定します。
-ORBInitRef	初期リファレンスを指定します。
-ORBnullstring <0|1>	1 を設定した場合，VisiBroker ORB が C++ NULL 文字をストリームします。NULL 文字列は長さが 0 の文字列としてマーシャルされます。これは，長さが 1 の文字列としてマーシャルされる空の文字列（""）とは異なり，単独の文字（"¥0"）です。 0 を設定した場合，NULL 文字列をマーシャルしたときは，CORBA::BAD_PARAM となります。NULL 文字列をアンマーシャルしたときは，CORBA::MARSHAL となります。 デフォルトは 0 です。 注 サーバとクライアントが同一プロセス内に存在し null 文字列を送信するようなエラーケースの場合，BAD_PARAM 例外は発生しません。サーバとクライアントが別プロセスの場合は CORBA::BAD_PARAM 例外が発生します。
-ORBrcvbufsize <buffer_size>	応答を受信するために使用する TCP バッファのサイズをバイト単位で指定します。指定値は，性能やベンチマークの結果に著しく影響します。 指定を省略した場合，デフォルト値が設定されます。 デフォルトは OS よって異なるため，各 OS のマニュアルを参照してください。

オプション	説明
-ORBsendbufsize <buffer_size>	クライアント要求を送信するために使用する TCP バッファのサイズをバイト単位で指定します。指定値は，性能やベンチマークの結果に著しく影響します。 指定を省略した場合，デフォルト値が設定されます。 デフォルトは OS よって異なるため，各 OS のマニュアルを参照してください。
-ORBtcpNoDelay <0|1>	ソケットが要求を送信する契機を指定します。指定値は，性能やベンチマークの結果に著しく影響します。 1 を設定した場合，すべてのソケットが即座に要求を送信します。 0 を設定した場合，ソケットはバッファが満杯になった時点で要求を一括して送信します。 デフォルトは 0 です。

Java の場合

クライアントは，vbj コマンドまたは Java 対応の Web ブラウザから起動します。

クライアントアプリケーションに指定できるコマンドライン引数を表 4-3 に示します。

表 4-3　クライアントアプリケーションのコマンドライン引数（Java）

オプション	説明
-DORBagentAddr=<hostname|ip_address>	このクライアントが使用するスマートエージェントを実行しているホストのホスト名または IP アドレスを指定します。 hostname を設定した場合，ホスト名を指定します。 ip_address を設定した場合，IP アドレスを指定します。 このオプションの指定を省略した場合や，自ホストまたは agentaddr ファイルで指定したホストからスマートエージェントを探索できない場合は，ブロードキャストメッセージでスマートエージェントを見つけます。
-DORBagentAddrFile=<file_name>	デフォルトファイルの agentaddr の代わりに使用するファイルを指定します。
-DORBagentNoFailOver=<false|true>	VisiBroker アプリケーションが通信している osagent が終了した場合，ほかの osagent と通信するかどうかを指定します。 true を設定した場合，VisiBroker アプリケーションは，先に通信していた osagent にだけ再度通信を試みます。 false を設定した場合，VisiBroker アプリケーションは，先に通信していた osagent だけでなくほかの osagent にも再度通信を試みます。 不正な値を指定した場合，true を設定したときと同じ動作をします。 デフォルトは false です。
-DORBagentPort=<port_number>	スマートエージェントのポート番号を指定します。複数の VisiBroker ORB ドメインが必要な場合に有用です。 デフォルトは 14000 です。指定を省略した場合，デフォルト値が設定されます。 アプリケーション内で ORB.init()メソッドの第 2 引数に ORBagentPort または ORBdisableLocator のプロパティを設定した場合，その効果はありません。ORB オプションの値を有効にする場合は，コマンドライン上で-D オプションで指定してください。

オプション	説明
-DORBalwaysProxy=<false|true>	クライアントが常にゲートキーパーを使用して接続する必要があるかどうかを指定します。 false を設定した場合，ゲートキーパーを使用しないで接続できます。 true を設定した場合，常にゲートキーパーを使用して接続します。また，true を設定した場合は，必ず-DORBgatekeeperIOR オプションも設定してください。 不正な値を指定した場合，true を設定したときと同じ動作をします。 デフォルトは false です。
-DORBalwaysTunnel=<false|true>	クライアントが常に HTTP を使用してゲートキーパーに接続する必要があるかどうかを指定します。 false を設定した場合，HTTP を使用しないで接続できます。 true を設定した場合，常に HTTP を使用して接続します。また，true を設定した場合は，必ず-DORBgatekeeperIOR オプションも設定してください。 不正な値を指定した場合，true を設定したときと同じ動作をします。 デフォルトは false です。
-DORBbackCompat=<false|true>	旧バージョンの VisiBroker クライアントと VisiBroker サーバとの互換性を確保するかどうかを指定します。指定値はランタイムに通知されます。 false を設定した場合，互換性を確保しません。 true を設定した場合，互換性を確保します。 旧バージョンの VisiBroker に基づいてサーバやクライアントの環境を構成する場合は，true を指定してください。 デフォルトは false です。
-DORBconnectionMax=<#>	コネクションの最大数を指定します。 指定を省略した場合，無制限にコネクションを許可します。
-DORBconnectionMaxIdle=<#>	コネクションが非アクティブな状態の最大監視時間を秒単位で指定します。最大監視時間を経過してもコネクションが非アクティブなままの場合，VisiBroker がコネクションを終了します。 このオプションはインターネットアプリケーションで設定します。 0 を設定した場合，監視しません。 最大アイドル時間の監視は，プロパティ vbroker.orb.gcTimeout の設定値（デフォルト 30 秒）の範囲の誤差があります。 デフォルトは 0 です。
-DORBdebug=<false|true>	デバッグ機能を使用するかどうかを指定します。 false を設定した場合，デバッグ機能を使用しません。 true を設定した場合，デバッグ機能を使用します。 デフォルトは false です。
-DORBdebugDir=<directory>	スレッドのデバッグ情報を書き込むディレクトリを指定します。 デフォルトでは，カレントワーキングディレクトリを使用します。
-DORBdebugThreads=<false|true>	スレッドのデバッグ機能を使用するかどうかを指定します。 false を設定した場合，デバッグ機能を使用しません。 true を設定した場合，デバッグ機能を使用します。 デフォルトは false です。

オプション	説明
-DORBDefaultInitRef	デフォルトの初期リファレンスを指定します。 nameserv には，ORBDefaultInitRef を使用しないでください。ORBInitRef を使用してください。
-DORBdisableAgentCache=<false|true>	スマートエージェントのキャッシュを有効にするかどうかを指定します。 false を設定した場合，有効にします。 true を設定した場合，無効にします。 不正な値を指定した場合，true を設定したときと同じ動作をします。 デフォルトは false です。
-DORBdisableGatekeeperCallbacks=<false|true>	ゲートキーパーのコールバックを有効にするかどうかを指定します。 false を設定した場合，有効にします。 true を設定した場合，無効にします。また，true を設定した場合は，-DORBgatekeeperIOR オプションも設定してください。 デフォルトは false です。
-DORBdisableLocator=<false|true>	スマートエージェントとゲートキーパーを有効にするかどうかを指定します。 false を設定した場合，有効にします。 true を設定した場合，無効にします。 不正な値を指定した場合，true を設定したときと同じ動作をします。 アプリケーション内で ORB.init()メソッドの第 2 引数に ORBagentPort または ORBdisableLocator のプロパティを設定した場合，その効果はありません。ORB オプションの値を有効にする場合は，コマンドライン上で-D オプションで指定してください。 デフォルトは false です。
-DORBgatekeeperIOR=<URL>	IOR に対応する URL を指定します。
-DORBgcTimeout=<#>	ORB のガーベッジコレクションを実行する周期を秒単位で指定します。 デフォルトは 30 です。
-DORBInitRef	初期リファレンスを指定します。
-DORBmbufSize=<buffer_size>	Borland Enterprise Server VisiBroker がオペレーション要求処理に使用する中間バッファのサイズを指定します。 性能向上のため，ORB は前のバージョンの Borland Enterprise Server VisiBroker よりも複雑なバッファ管理を行います。送信または受信したデータがデフォルトより大きければ，リクエストごと，または応答ごとに新しいバッファが割り当てられます。 4 キロバイトより大きいデータをアプリケーションが頻繁に送信する場合に，バッファ管理の利点を生かしたいとき，このシステムプロパティを使用して，デフォルトのバッファサイズよりも大きいバイト数を指定できます。 送信バッファと受信バッファのデフォルトサイズは，それぞれ 4 キロバイトです。
-DORBservices=<service>	インストールする ORB の特殊サービスを指定します。 インストールできるサービスは，ユーザが作成したサービス，Borland Enterprise Server VisiBroker が提供する ORBManager サービス，

オプション	説明
-DORBservices=<service>	および別製品の Borland Enterprise Server VisiBroker サービス（ネーミングサービスなど）です。
-DORBtcpNoDelay=<false|true>	ネットワークコネクションがデータを送信する契機を指定します。 false を設定した場合，ネットワークコネクションは，バッファが満杯になった時点でデータを一括して送信します。 true を設定した場合，すべてのネットワークコネクションが即座にデータを送信します。 デフォルトは false です。
-DORBwarn=<0|1|2>	警告メッセージの出力レベルを指定します。 0 を設定した場合，警告メッセージを出力しません。 1 を設定した場合，次の警告メッセージを出力します。 • ユーザ作成コードからの CORBA 以外の例外 • ユーザ作成コードからの CORBA 以外の例外のスタックトレース 2 を設定した場合，次の警告メッセージを出力します。 • ユーザ作成コードからの CORBA 以外の例外 • ユーザ作成コードからの CORBA 以外の例外のスタックトレース • CORBA の例外 • CORBA の例外のスタックトレース デフォルトは 0 です。

5 例外の処理

この章では，システム例外およびユーザ例外について説明します。

5.1　CORBA モデルでの例外

CORBA モデルでの例外にはシステム例外とユーザ例外の両方が含まれます。CORBA の仕様では，クライアントリクエストの処理でエラーが発生した場合に発生する可能性のある一組のシステム例外を定義します。システム例外は通信障害の場合にも発生します。システム例外はいつでも発生する可能性があり，インタフェース内で宣言する必要はありません。ユーザ例外は，生成するオブジェクトについて IDL で定義でき，このような例外が起こる環境を指定できます。これらはメソッドシグニチャに含まれます。クライアントリクエストの処理中にオブジェクトが例外を発生させると，ORB はこの情報をクライアントに反映する責任を負います。

5.2　システム例外

オブジェクトインプリメンテーションは，「20.　VisiBroker 4.x インタセプタの使用」で説明しているインタセプタを介してシステム例外を発生させることができますが，通常は VisiBroker ORB がシステム例外を発生させます。

主要な CORBA 例外の一覧と VisiBroker ORB がその例外を発生させた理由を表 5-1 に示します。また，CORBA 例外のマイナーコードを表 5-2 に示します。

表 5-1　主要な CORBA 例外，および考えられる原因

例外	例外の内容	考えられる原因
CORBA::BAD_CONTEXT	サーバに無効コンテキストが渡されました。	クライアントがオペレーションを呼び出したが，渡されたコンテキストにオペレーションで必要なコンテキスト値がない場合に，オペレーションはこの例外を発生します。
CORBA::BAD_INV_ORDER	オペレーション要求の前に，必要な前提条件オペレーションが呼び出されていません。	• リクエストを実際に送信する前に，CORBA::Request::get_response()メソッドかCORBA::Request::poll_response()メソッドを呼び出そうとした可能性があります。 • C++ ORB では，リモートメソッド呼び出しのインプリメンテーション外で，exception::get_client_info()メソッドを呼び出そうとした可能性があります。この関数が有効なのは，リモート呼び出しのインプリメンテーション内だけです。 • すでにシャットダウンされた VisiBroker ORB でオペレーションが呼び出されました。
CORBA::BAD_OPERATION	無効なオペレーションが実行されました。	• サーバは，IDL で定義されていないオペレーションを受信すると，この例外を発生させます。クライアントとサーバが，同じ IDL からコンパイルされているかどうかを確認してください。 • リクエストが，リターン値を使用するように設定されていないと，CORBA::Request::return_value()メソッドはこの例外を発生させます。DII 呼び出しをするときにリターン値が期待される場合には，必ず，CORBA::Request::set_return_type()メソッドを呼び出してリターン値型を設定してください。
CORBA::BAD_PARAM	無効なパラメタが引き渡されました。	• sequence 型の無効インデックスにアクセスしようとすると発生します。必ず，length()メソッドを使用してシーケンスの長さを設定してから，そのシーケンスの要素を格納または検索してください。 • Java ORB では，列挙体のデータの範囲外の値を送信しようとした可能性があります。 • 無効な TCKind を指定して TypeCode を構築しようとした可能性があります。 • nil または NULL オブジェクトリファレンスを Any に挿入しようとした可能性があります。 • DII と oneway メソッド呼び出しを使用して out 引数が指定された可能性があります。

例外	例外の内容	考えられる原因
CORBA::BAD_PARAM	無効なパラメタが引き渡されました。	• インタフェースリポジトリに登録するインプリメンテーションオブジェクトの登録情報（例えば登録名）がすでに存在している場合，インタフェースリポジトリはこの例外を発生させます。 • C++ ORB では，無効な CORBA::Object_ptr が in 引数として渡されると（例えば，nil リファレンスが渡されると），この例外を発生させます。 • C++ ORB では，NULL ポインタの送信が試行されるとこの例外を発生させます。例えば，リターン値として NULL を返そうとしたり，sequence を返さなければならないメソッドから out パラメタを返そうとすると，この例外が発生します。この場合，新しい sequence（長さは 0 の可能性がある）が代わりに返されなければなりません。NULL 値で送信できないタイプには，Any，Context，struct，または sequence が含まれます。 • Java ORB では，null リファレンスが渡されると，この例外を発生させます。
CORBA::BAD_TYPECODE	ORB が不正な TypeCode を検出しました。	−
CORBA::CODESET_INCOMPATIBLE	クライアントとサーバのコードセットに互換がないため，通信に失敗しました。	クライアントとサーバが使用するコードセットが，一致していません。例えば，クライアントは ISO 8859-1 を使用し，サーバは日本語コードセットを使用しています。
CORBA::COMM_FAILURE	通信障害が発生しました。	C++ORB では，一端のコネクション障害によって，既存のコネクションがクローズすることがあります。また，クライアントマシンまたはサーバマシンのリソース制限によって（最大コネクション数に到達し）通信が失敗した場合，この例外が発生します。 Java ORB では，オペレーションの進行中（クライアントがリクエストを送信後で，サーバからクライアントに応答が返される前）に通信が損失すると，この例外が発生します。
CORBA::DATA_CONVERSION	データ変換エラーが発生しました。	マーシャリングで不正な文字コードを検出した場合，この例外が発生します。
CORBA::IMP_LIMIT	インプリメンテーションの上限に違反しました。	−
CORBA::INITIALIZE	必要な初期化が実行されませんでした。	ORB_init()メソッドまたは ORB.init()メソッドが呼び出されなかった可能性があります。VisiBroker ORB 対応のオペレーションを実行する前には，クライアントは，C++アプリケーションの場合は ORB_init()メソッドを，Java アプリケーションの場合は ORB.init()メソッドを呼び出す必要があります。この呼び出しは通常，main ルーチンの最上部でプログラムが起動されるとすぐに実行されます。 Java ORB では，起動時に指定するオプションに不正値を設定したおそれがあります。例えば，vbroker.orb.gcTimeout にマイナスの値を設定した場合が該当します。
CORBA::INTERNAL	内部エラーが発生しました。	VisiBroker ORB の内部データ構造が破壊されるなど，ORB 内部で論理矛盾を検出した場合に発生します。

例外	例外の内容	考えられる原因
CORBA::INTF_REPOS	インタフェースリポジトリへのアクセスエラーが発生しました。	get_interface()メソッドの呼び出し時に，オブジェクトインプリメンテーションが IR を見つけられないと，クライアントでこの例外が発生します。IR が実行されていることを確認してください。また，リクエストされたオブジェクトのインタフェース定義が，IR にロードされていることを確認してください。
CORBA::INV_IDENT	識別子の構文が無効です。	IR に渡された識別子の書式が正しくありません。 不当なオペレーション名が動的起動インタフェースで使用されています。
CORBA::INV_OBJREF	無効なオブジェクトリファレンスが検出されました。	使用できるプロファイルが指定されていないオブジェクトリファレンスが取得されると，VisiBroker ORB はこの例外を発生させます。 文字列型オブジェクトリファレンスの先頭文字が「IOR:」でないと，CORBA::ORB::string_to_object()メソッドは，この例外を発生させます。 IOR ファイルを指定する場合，ファイル中の文字列は object_to_string の結果と同じである必要があります。ファイル中には，空白，タブ，改行コードを含めて不正な文字が存在してはいけません。これに該当していない場合は INV_OBJREF 例外が発生します。
CORBA::INV_POLICY	無効なポリシーの変更が検出されました。	この例外は，どの呼び出しからも発生する可能性があります。特定の呼び出しに適用されるポリシーの変更同士で互換性がないために呼び出しができない場合に発生します。
CORBA::INVALID_TRANSACTION	トランザクションコンテキストが不正です。	この例外の詳細については，トランザクションサービスのドキュメントを参照してください。
CORBA::MARSHAL	マーシャルパラメタまたは結果が不当です。	ネットワークからの要求または応答が構造的に不当です。このエラーは，通常，クライアント側またはサーバ側のランタイムのどちらかのバグを示します。例えば，サーバからの応答で，メッセージが 1000 バイトであるが，実際のメッセージが 1000 バイトより短いか長いことを示す場合，VisiBroker ORB はこの例外を発生させます。MARSHAL 例外は DII や DSI を不当に使用しても発生します。例えば，送信された実際のパラメタの型がオペレーションの IDL シグニチャと一致しない場合です。
CORBA::NO_IMPLEMENT	リクエストオブジェクトが見つかりませんでした。	呼び出されたオペレーションがある（IDL 定義がある）のに，そのオペレーションのインプリメンテーションがないことを示します。
CORBA::NO_PERMISSION	許可されていないオペレーションを実行しようとしました。	－
CORBA::NO_RESOURCES	必要な資源を取得できませんでした。	• 新しいスレッドが生成できない場合に，この例外が発生します。 • リモートクライアントがコネクションを確立しようとしたときに，サーバがファイルディスクリプタを使い果たすなどしてソケットを生成できないと，サーバはこの例外を発生させます。

例外	例外の内容	考えられる原因
CORBA::NO_RESOURCES	必要な資源を取得できませんでした。	• ファイルディスクリプタを使い果たしたなどの理由でコネクションの確立が失敗すると，クライアントも同じようにこの例外を発生させます。 • C++ORB では，メモリを使い切った場合にもこの例外が発生します。
CORBA::NO_RESPONSE	クライアントが送信したリクエストの応答がまだありません。	OAD に登録したオブジェクトインプリメンテーションの活性化が，OAD に指定した時間内に行われなかった場合に，この例外が発生します。
CORBA::OBJ_ADAPTER	オブジェクトアダプタが障害を検出しました。	アプリケーションのサーバントマネージャの問題を検出した場合などに，POA はこの例外を発生させます。
CORBA::OBJECT_NOT_EXIST	リクエストされたオブジェクトが存在していません。	• 該当するサーバ内に存在しないインプリメンテーションでオペレーションを実行しようとすると，サーバは，この例外を発生させます。 • 非活性化したインプリメンテーションでオペレーションを起動しようとすると，この例外がクライアントによって表示されます。 例えば，オブジェクトへのバインドが失敗，または自動リバインドが失敗すると，OBJECT_NOT_EXIST 例外が発生します。
CORBA::REBIND	クライアントが，QoS ポリシーに矛盾する IOR を受信しました。	設定された QoS ポリシーに矛盾する IOR をクライアントが取得するとこの例外が発生します。RebindPolicy に NO_REBIND，NO_RECONNECT，または VB_NOTIFY_REBIND の値があると，バインドされたオブジェクトリファレンスの呼び出しの結果，オブジェクト転送メッセージやロケーション転送メッセージが生成されます。
CORBA::TIMEOUT	オペレーションがタイムアウトしました。	コネクションを設定しようとしているとき，またはリクエストの送受信の完了を待っているときに，指定時間前にオペレーションが完了しないと，TIMEOUT 例外が発生します。
CORBA::TRANSACTION_REQUIRED	リクエスト時に無効なトランザクションコンテキストがトランザクションサービスに渡されましたが，アクティブなトランザクションが必要です。	この例外の詳細については，トランザクションサービスのドキュメントを参照してください。
CORBA::TRANSACTION_ROLLEDBACK	リクエストに対応するトランザクションがすでにロールバックされているか，またはロールバック用にマーキングされています。	この例外の詳細については，トランザクションサービスのドキュメントを参照してください。
CORBA::TRANSIENT	通信エラーが検出されましたが，再接続できる場合があります。	通信障害が発生したおそれがありますが，VisiBroker ORB は通信が失敗したサーバとリバインドする必要があるとシグナル通知しています。RebindPolicy の設定によっては，この例外は発生しません。
CORBA::UNKNOWN	未知の例外です。	• サーバが発生させたのは，Java ランタイム例外などの適切な例外ではありません。

例外	例外の内容	考えられる原因
CORBA::UNKNOWN	未知の例外です。	• サーバとクライアント間に IDL 不一致があり，クライアントプログラムで，この例外は定義されていません。 • DII では，サーバがコンパイル時にクライアントに未知の例外を発生させ，クライアントが CORBA::Request の例外リストを指定しなかった場合です。 サーバが Java アプリケーションの場合は，vbroker.orb.warn=2 プロパティを設定して，どのランタイム例外が問題の原因かを調べます。

（凡例）－：該当しない

表 5-2　CORBA 例外のマイナーコード

システム例外	マイナーコード	説明
BAD_PARAM	1	値ファクトリの登録，登録解除，または探索に失敗しました。
	2	RID がすでに IR に定義されています。
	3	名前がすでに IR のコンテキストで使用されています。
	4	ターゲットが有効なコンテナではありません。
	5	継承されたコンテキストで名前がクラッシュしました。
	6	abstract インタフェースのタイプが正しくありません。
MARSHAL	1	値ファクトリを探索できません。
NO_IMPLEMENT	1	ローカル値インプリメンテーションがありません。
	2	値のインプリメンテーションバージョンの互換性がありません。
BAD_INV_ORDER	1	IR に依存性が存在し，オブジェクトがデストラクトできません。
	2	IR 内のデストラクトできないオブジェクトをデストラクトしようとしました。
	3	オペレーションがデッドロックになりました。
	4	VisiBroker ORB がシャットダウンしました。
OBJECT_NOT_EXIST	1	活性化されていない（登録解除されている）値をオブジェクトリファレンスとして渡そうとしました。

コードサンプル 5-1　SystemException クラス（C++）

```
class SystemException : public CORBA::Exception {
   public:
      static const char*_id;
      virtual ~SystemException();
      CORBA::ULong minor() const;
      void minor(CORBA::ULong val);
      CORBA::CompletionStatus completed() const;
      void completed(CORBA::CompletionStatus status);
      . . .
      static SystemException *_downcast(Exception *exc);
      . . .
};
```

コードサンプル 5-2 SystemException クラス（Java）

```
public abstract class org.omg.CORBA.SystemException
                        extends java.lang.RuntimeException {
   protected SystemException(java.lang.String reason,
      int minor,CompletionStatus completed) {...}
   public String toString() {...}
   public CompletionStatus completed;
   public int minor;
}
```

5.2.1 完了状態の取得

システム例外には，例外発生時に，オペレーションが完了したかどうかを伝える完了状態があります。CompletionStatus の列挙値を次に示します。オペレーションの完了状態が判定できない場合には，COMPLETED_MAYBE が返されます。

IDL サンプル 5-1 CompletionStatus 値

```
enum CompletionStatus {
   COMPLETED_YES = 0;
   COMPLETED_NO = 1;
   COMPLETED_MAYBE = 2;
};
```

C++の場合，SystemException メソッドを使用して完了状態を検索できます。

コードサンプル 5-3 完了状態の検索

```
CompletionStatus completed();
```

5.2.2 マイナーコードの取得と設定（C++）

SystemException メソッドを使用してマイナーコードを検索，設定できます。マイナーコードからエラーのタイプの詳細情報がわかります。

コードサンプル 5-4 マイナーコードの検索と設定

```
ULong minor() const;
void minor(ULong val);
```

5.2.3 システム例外のタイプの判定（C++）

Borland Enterprise Server VisiBroker の例外クラスの設計によって，ユーザのプログラムはどのようなタイプの例外でもキャッチし，_downcast()メソッドによってそのタイプを判定できます。静的メソッドである_downcast()メソッドは，Exception オブジェクトへのポインタを受け付けます。CORBA::Object で定義した_downcast()メソッドによって，ポインタが SystemException タイプの場合，_downcast()メソッドはポインタを返します。ポインタが SystemException タイプでない場合，_downcast()メソッドは NULL ポインタを返します。詳細については，「23. RMI-IIOP の使用」を参照してください。

5.2.4 システム例外のキャッチ

アプリケーションは，VisiBroker ORB とリモートコールをトライキャッチブロックで囲むのがよいでしょう。「4. Borland Enterprise Server VisiBroker によるサンプルアプリケーションの開発」で説明したアカウントのクライアントプログラムが C++の例外をどのように出力するのかを，コードサンプル 5-5 でわかりやすく説明します。コードサンプル 5-6 は，アカウントクライアントプログラムが Java の例外をどのように出力するかを示します。

コードサンプル 5-5　例外の出力（C++）

```
#include "Bank_c.hh"
int main(int argc, char* const* argv) {
   try {
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      PortableServer::ObjectId_var managerId =
            PortableServer::string_to_ObjectId("BankManager");
      Bank::AccountManager_var manager =
            Bank::AccountManager::_bind("/bank_agent_poa",
                  managerId);
      const char* name = argc >1 ? argv [1] : "Jack B. Quick";
      Bank::Account_var account = manager->open(name);
      CORBA::Float balance = account->balance();
      cout << "The balance in " << name << "'s account is $"
           << balance << endl;
   } catch(const CORBA::Exception& e) {
     cerr << e << endl;
     return 1;
   }
   return 0;
}
```

コードサンプル 5-6　例外の出力（Java）

```
public class Client {
   public static void main(String[ ] args) {
      try {
         org.omg.CORBA.ORB orb =
                        org.omg.CORBA.ORB.init(args, null);
         byte[ ] managerId = "BankManager".getBytes();
         Bank.AccountManager manager =
               Bank.AccountManagerHelper.bind(orb,
                                        "/bank_agent_poa",
                                        managerId);
         String name =
                 args.length > 0 ? args[0] : "Jack B. Quick";
         Bank.Account account = manager.open(name);
         float balance = account.balance();
         System.out.println("The balance in "
                        + name + "'s account is $" + balance);
     } catch (Exception e) {
         System.err.println(e);
     }
   }
}
```

上記のような変更を加えたクライアントプログラムをサーバが存在しない場合に実行するときは，次のような出力が行われ，オペレーションが未完であることと，例外の理由を示します。

C++の場合

```
prompt> Client
Exception: CORBA::OBJECT_NOT_EXIST
         Minor: 0
         Completion Status: NO
```

Java の場合

```
prompt>vbj Client
org.omg.CORBA.OBJECT_NOT_EXIST:
      Could not locate the following POA:
      poa name : /bank_agent_poa
      minor code:0 completed: No
```

5.2.5　システム例外への例外のダウンキャスト

キャッチした例外を SystemException へダウンキャストするために，アカウントのクライアントプログラムを修正できます。コードサンプル 5-7 および 5-8 に，クライアントプログラムの修正方法を示します。コードサンプル 5-9 および 5-10 に，システム例外が発生した場合に表示される出力を示します。

コードサンプル 5-7 システム例外への例外のダウンキャスト（C++）

```
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

      // Bind to an account.
      Account_var account = Account::_bind();

      // Get the balance of the account.
      CORBA::Float acct_balance = account->balance();
      // Print out the balance.
      cout << "The balance in the account is $"
           << acct_balance << endl;
      } catch(const CORBA::Exception& e) {
         CORBA::SystemException* sys_excep;
         sys_excep = CORBA::SystemException::_downcast
                                  ((CORBA::Exception)(&e));
         if(sys_excep != NULL) {
            cerr << "System Exception occurred:" << endl;
            cerr << "exception name: " <<
            sys_excep->_name() << endl;
            cerr << "minor code: " << sys_excep->minor()
                 << endl;
            cerr << "completion code: "
                 << sys _excep->completed()
                 << endl;
      } else {
         cerr << "Not a system exception" << endl;
         cerr << e << endl;
      }
   }
}
```

コードサンプル 5-8 システム例外への例外のダウンキャスト（Java）

```
public class Client {
   public static void main(String[ ] args){
     try {
        // Initialize the ORB
        org.omg.CORBA.ORB orb =
                          org.omg.CORBA.ORB.init(args,null);
        // Bind to an account
        Account account = AccountHelper.bind(orb, "/bank_poa",
                     "BankAccount".getBytes());
        // Get the balance of the account
        float balance = account.balance();
        // Print the account balance
        System.out.println("The account balance is $" + balance);
     }catch(Exception e){
        if (e instanceof org.omg.CORBA.SystemException){
          System.err.println("System Exception occurred:");
        }else {
          System.err.println("Not a system exception");
        }
        System.err.println(e);
     }
   }
}
```

コードサンプル 5-9 システム例外からの出力（C++）

```
System Exception occurred:
      exception name: CORBA::NO_IMPLEMENT
      minor code: 0
      completion code: 1
```

コードサンプル 5-10 システム例外からの出力（Java）

```
System Exception occurred:
in thread "main"org.omg.CORBA.OBJECT_NOT_EXIST
                                  minor code: 0 completed: No
```

(1)　特定の型のシステム例外のキャッチ

すべての型の例外をキャッチするのではなく，主要な各型の例外を明確にキャッチするように選択できます。コードサンプル 5-11 に C++，およびコードサンプル 5-12 に Java のそれぞれの方法を示します。

コードサンプル 5-11　特定の型の例外のキャッチ（C++）

```
. . .
int main(int argc,char* const* argv){
   try {
       // Initialize the ORB.
       CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

       // Bind to an account.
       Account_var account = Account::_bind();

       // Get account balance.
       CORBA::Float acct_balance = account->balance();
       // Print out the balance.
       cout << "The balance in the account is $"
            << acct_balance << endl;
   }
   // Check for system errors
   catch(const CORBA::SystemException& sys_excep) {
      cout << "System Exception occurred:" << endl;
      cout << "exception name: " << sys_excep->_name() << endl;
      cout << "minor code: " << sys_excep->minor() << endl;
      cout << "completion code: " << sys_excep->completed()
           << endl;
   }
}
. . .
```

コードサンプル 5-12　特定の型の例外のキャッチ（Java）

```
public class Client {
   public static void main(String[ ] args) {
     try {
        org.omg.CORBA.ORB orb =
                        org.omg.CORBA.ORB.init(args, null);
        byte[ ] managerId = "BankManager".getBytes();
        Bank.AccountManager manager =
             Bank.AccountManagerHelper.bind(orb,
                                        "/bank_agent_poa",
                                        managerId);
        String name =
                args.length > 0 ? args [0] : "Jack B. Quick";
        Bank.Account account = manager.open(name);
        float balance = account.balance();
        System.out.println("The balance in " + name +
                           "'s account is $" + balance);
     } catch(org.omg.CORBA.SystemException e) {
        System.err.println("System Exception occurred:");
        System.err.println(e);
     }
   }
}
```

5.3　ユーザ例外

オブジェクトのインタフェースを IDL で定義する場合には，オブジェクトが発生させる可能性のあるユーザ例外を指定できます。コードサンプル 5-13 および 5-14 に，オブジェクトについて指定するユーザ例外を，idl2cpp コンパイラ（C++）または idl2java コンパイラ（Java）が UserException コードから派生させる場合について示します。

コードサンプル 5-13　UserException クラス（C++）

```
class UserException: public Exception {
   public:
      . . .
      static const char*_id;
      virtual ~UserException();
      static UserException *_downcast(Exception *);
   };
```

コードサンプル 5-14　UserException クラス（Java）

```
public abstract class UserException extends
                                          java.lang.Exception {
   protected UserException();
   protected UserException(String reason);
}
```

5.3.1　ユーザ例外の定義

「4.　Borland Enterprise Server VisiBroker によるサンプルアプリケーションの開発」で説明したアカウントアプリケーションを拡張して，account オブジェクトが例外を発生させるようにしたい場合，account オブジェクトの資金が不十分なら，AccountFrozen という名前のユーザ例外を発生させる必要があります。Acoount インタフェースの IDL 指定にユーザ例外を追加するために必要な追加コードを，ボールド体で示します。

IDL サンプル 5-2　ユーザ例外の定義

```
// Bank.idl
module Bank {
   interface Account {
      exception AccountFrozen {
      };
      float balance() raises(AccountFrozen);
   };
};
```

idl2cpp コンパイラ（C++）または idl2java（Java）コンパイラは，AccountFrozen 例外クラスに対して次に示すコードを生成します。

コードサンプル 5-15　idl2cpp コンパイラが生成する AccountFrozen クラス（C++）

```
class Account : public virtual CORBA::Object {
   . . .
   class AccountFrozen: public CORBA_UserException {
      public:
         static const CORBA_Exception::Description description;

        AccountFrozen() {}
        static CORBA::Exception *_factory() {
           return new AccountFrozen();
        }
        ~AccountFrozen() {}
        virtual const CORBA_Exception::Description& _desc()
              const;
        static AccountFrozen *_downcast(CORBA::Exception *exc);
        CORBA::Exception *_deep_copy() const {
           return new AccountFrozen(*this);
        }
```

```
        void _raise() const {
           raise *this;
        }
     }
   . . .
}
```

コードサンプル 5-16　idl2java コンパイラが生成する AccountFrozen クラス（Java）

```
package Bank;
public interface Account extends
                          com.inprise.vbroker.CORBA.Object,
     Bank.AccountOperations, org.omg.CORBA.portable.IDLEntity {
}

package Bank;
public interface AccountOperations {
   public float balance () throws
                         Bank.AccountPackage.AccountFrozen;
}

package Bank.AccountPackage;
public final class AccountFrozen extends
                               org.omg.CORBA.UserException {
   public AccountFrozen (){...}
   public AccountFrozen (java.lang.String _reason){...}
   public synchronized java.lang.String toString(){...}
}
```

(1)　例外を発生させるためのオブジェクトの修正

適切なエラー条件下で例外を発生させることによって例外を使用するように，AccountImpl オブジェクトを修正する必要があります。

コードサンプル 5-17　例外を発生させるためのオブジェクトインプリメンテーションの修正（C++）

```
CORBA::Float AccountImpl::balance()
{
      if( _balance < 50 ) {
            throw Account::AccountFrozen();
      } else {
           return _balance;
      }
}
```

コードサンプル 5-18　例外を発生させるためのオブジェクトインプリメンテーションの修正（Java）

```
public class AccountImpl extends Bank.AccountPOA {
   public AccountImpl(float balance) {
       _balance = balance;
   }
   public float balance()throw new AccountFrozen {
      if (_balance < 50) {
         throw new AccountFrozen();
      }else {
        return _balance;
      }
  }
   private float _balance;
}
```

(2)　ユーザ例外のキャッチ

オブジェクトインプリメンテーションが例外を発生させる場合，ORB は例外をクライアントプログラムに反映させる責任を負います。UserException についてのチェックは SystemException についてのチェックと同様です。AccountFrozen 例外をキャッチするようにアカウントのクライアントプログラムを修正するには，コードサンプル 5-19 および 5-20 で示すようにコードの修正をしてください。

コードサンプル 5-19 UserException のキャッチ（C++）

```
. . .
   try {
      // Initialize the ORB.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

      // Bind to an account.
      Account_var account = Account::_bind();

      // Get the balance of the account.
      CORBA::Float acct_balance = account->balance();
   }
   catch(const Account::AccountFrozen& e) {
      cerr << "AccountFrozen returned:" << endl;
      cerr << e << endl;
      return(0);
   }
   // Check for system errors
   catch(const CORBA::SystemException& sys_excep) {
   }
. . .
```

コードサンプル 5-20 UserException のキャッチ（Java）

```
public class Client {
   public static void main(String[ ] args) {
     try {
        // Initialize the ORB
        org.omg.CORBA.ORB orb =
                        org.omg.CORBA.ORB.init(args, null);
        // Bind to an account
        Account account = AccountHelper.bind(orb, "/bank_poa",
                             "BankAccount".getBytes());

        // Get the balance of the account
        float balance = account.balance();

        // Print the account balance
        System.out.println("The account balance is $" +
                           balance);
     }
     // Check for AccountFrozen exception
     catch(Account.AccountFrozen e) {
        System.err.println("AccountFrozen returned:");
        System.err.println(e);
     }
     // Check for system error
     catch(org.omg.CORBA.SystemException sys_excep) {
     . . .
     }
   }
}
```

(3) ユーザ例外へのフィールドの追加

ユーザ例外に特定の値を対応させることができます。理由コードを AccountFrozen ユーザ例外に追加するように IDL インタフェース定義を修正する方法をコードサンプル 5-21 に示します。例外を発生させるオブジェクトインプリメンテーションは，理由コードの設定に責任を負います。理由コードは，例外が出力ストリームにあると自動的に出力されます。

コードサンプル 5-21 AccountFrozen 例外への理由コードの追加

```
// Bank.idl
module Bank {
   interface Account {
      exception AccountFrozen {
         int reason;
      };
      float balance() raises(AccountFrozen);
   };
};
```

6 サーバの基本事項

この章では, クライアントリクエストを受信するサーバの設定に必要な手順の概要について説明します。

6.1　概要

サーバ設定の基本手順は，次のとおりです。

- VisiBroker ORB の初期化
- POA の作成と設定
- POA マネージャの活性化
- オブジェクトの活性化
- クライアントリクエストを待つ

この章では，ポイントを明確にするために，各手順の概要を説明します。各手順の詳細は，個々の要件によって異なります。

6.2　VisiBroker ORB の初期化

前章までで説明したように, VisiBroker ORB はクライアントリクエストとオブジェクトインプリメンテーション間の通信リンクを提供します。各アプリケーションは，VisiBroker ORB と通信を行う前に VisiBroker ORB を初期化しなければなりません。

コードサンプル 6-1　VisiBroker ORB の初期化（C++）

```
// Initialize the ORB.
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
```

コードサンプル 6-2　VisiBroker ORB の初期化（Java）

```
// Initialize the ORB.
org.omg.CORBA.ORB orb=org.omg.CORBA.ORB.init(args,null);
```

6.3　POA の作成

旧バージョンの CORBA オブジェクトアダプタ（基本オブジェクトアダプタ：BOA）ではポータブルなオブジェクトサーバコードを使用できませんでした。新しい仕様はこの問題を解決するために OMG によって開発されたもので，POA（ポータブルオブジェクトアダプタ）といいます。

注

この節では，POA の基本機能の幾つかを紹介します。詳細については，「7.　POA の使用」および OMG 仕様を参照してください。

基本的には，POA（およびそのコンポーネント）はクライアントリクエスト受信時にどのサーバントを起動するかを決定してから，そのサーバントを起動します。サーバントとは abstract オブジェクトのインプリメンテーションを提供するプログラミングオブジェクトです。サーバントは CORBA オブジェクトではありません。

各 VisiBroker ORB は一つの POA（rootPOA といいます）を提供します。追加の POA を作成して，それぞれの POA を異なった動作で構成できます。また，POA が制御するオブジェクトの特徴を定義することもできます。

サーバントを使った POA の設定手順は次のとおりです。

- rootPOA のリファレンスの取得
- POA ポリシーの定義
- rootPOA の子として POA を作成
- サーバントの作成と活性化
- POA のマネージャを介した POA の活性化

上記の手順には，アプリケーションによって異なるものがあります。

6.3.1　rootPOA のリファレンスの取得

オブジェクトを管理したり新たに POA を作成したりするために，すべてのサーバアプリケーションは rootPOA のリファレンスを取得する必要があります。

コードサンプル 6-3　rootPOA のリファレンスの取得（C++）

```
// get a reference to the rootPOA
CORBA::Object_var obj =
      orb->resolve_initial_references("RootPOA");
// narrow the object reference to a POA reference
PortableServer::POA_var rootPOA =
      PortableServer::POA::_narrow(obj);
```

コードサンプル 6-4　rootPOA のリファレンスの取得（Java）

```
//2.Get a reference to the rootPOA
org.omg.CORBA.Object obj =
                  orb.resolve_initial_references("RootPOA");
// Narrow the object reference to a POA reference
POA rootPoa =  org.omg.PortableServer.POAHelper.narrow(obj);
```

rootPOA のリファレンスは，resolve_initial_references を使用して取得できます。
resolve_initial_references は，CORBA::Object 型の値を返します。返されたオブジェクトリファレンスを希望の型（上記のサンプルでは PortableServer::POA）にナロウするのはプログラマの責任です。

これで，必要ならこのリファレンスを使用してほかの POA を作成できるようになります。

6.3.2　子 POA の作成

rootPOA にはあらかじめ定義されたポリシーの集合があり，これらは変更できません。ポリシーとは，POA の動作と POA が管理するオブジェクトを制御するオブジェクトのことです。別のライフスパンポリシーのような別の動作が必要なら，新しい POA を作成できます。

POA は，create_POA を使用して既存 POA の子として作成します。必要なだけの数の POA を作成できます。

注

子 POA は親 POA のポリシーを継承しません。

次に示すサンプルでは，子 POA は rootPOA から作成され，パーシステントなライフスパンポリシーを持ちます。この子 POA の状態管理には rootPOA の POA マネージャを使用します。POA マネージャの詳細については，「6.3.4　POA の活性化」で説明します。

コードサンプル 6-5　ポリシーと子 POA の作成（C++）

```
CORBA::PolicyList policies;
policies.length(1);
policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
      PortableServer::PERSISTENT);
// Create myPOA with the right policies
PortableServer::POAManager_var rootManager =
      rootPOA->the_POAManager();
PortableServer::POA_var myPOA = rootPOA->create_POA(
      "bank_agent_poa", rootManager,policies );
```

コードサンプル 6-6　ポリシーと子 POA の作成（Java）

```
// Create policies for our persistent POA
org.omg.CORBA.Policy[ ] policies = {
   rootPOA.create_lifespan_policy(
                              LifespanPolicyValue.PERSISTENT)
};
// Create myPOA with the right policies
POA myPOA =  rootPOA.create_POA( "bank_agent_poa",
                                 rootPOA.the_POAManager(),
                                 policies );
```

6.3.3　サーバントメソッドのインプリメント

IDL は C++と似た構文を持ち，モジュール，インタフェース，データ構造などの定義に使用できます。インタフェースを含む IDL をコンパイルするとき，サーバントのベースクラスとして動作するクラスが生成されます。例えば，Bank.idl ファイルには，AccountManager インタフェースが記述されます。

コードサンプル 6-7　Bank.idl に記述されるインタフェース

```
module Bank{
   interface Account {
         float balance();
   };
   interface AccountManager {
         Account open (in string name);
   };
};
```

コードサンプル 6-8 にサーバ側の AccountManager インプリメンテーションを示します。

コードサンプル 6-8　AccountManagerImpl コード（C++）

```
class AccountManagerImpl : public POA_Bank::AccountManager {
   private:
      Dictionary _accounts;
   public:
```

```
        virtual Bank::Account_ptr open(const char* name) {
           // Lookup the account in the account dictionary.
           Bank::Account_ptr account =
                 (Bank::Account_ptr) _accounts.get(name);
           if(account == Bank::Account::_nil()) {
              // Make up the account's balance, between 0 and
              // 1000 dollars.
           float balance = abs(rand()) % 100000 / 100.0;
           // Create the account implementation, given
           // the balance.
           AccountImpl *accountServant =
                 new AccountImpl(balance);
           try {
               // Activate it on the default POA which is root
               // POA for this servant
               PortableServer::POA_var rootPOA = _default_POA();
               CORBA::Object_var obj =
                   rootPOA->servant_to_reference(accountServant);
               account = Bank::Account::_narrow(obj);
           } catch(const CORBA::Exception& e) {
               cerr << "_narrow caught exception: " << e << endl;
           }
           // Print out the new account.
           cout << "Created " << name << "'s account: " <<
                 account << endl;
           // Save the account in the account dictionary.
           _accounts.put(name, account);
        }

        // Return the account.
        return Bank::Account::_duplicate(account);
     }
  };
```

Java の場合，コードサンプル 6-9 に示すように，AccountManagerPOA.java が作成され，サーバ側の AccountManager オブジェクトインプリメンテーションのスケルトンコード（インプリメンテーションベースコード）として動作します。

コードサンプル 6-9　AccountManagerImpl コード（Java）

```
import org.omg.PortableServer.*;
import java.util.*;
public class AccountManagerImpl extends
                                  Bank.AccountManagerPOA {
   public synchronized Bank.Account open(String name){
     // Lookup the account in the account dictionary.
     Bank.Account account =
                         (Bank.Account) _accounts.get(name);
     // If there was no account in the dictionary, create one.
     if(account == null){
        // Make up the account's balance,
        // between 0 and 1000 dollars.
        float balance =
               Math.abs(_random.nextInt()) % 100000 / 100f;
        // Create the account implementation, given
        // the balance.
        AccountImpl accountServant =
                                  new AccountImpl(balance);
        try {
          // Activate it on the default POA
          // which is rootPOA for this servant
          account =
             Bank.AccountHelper.narrow(_default_POA().
             servant_to_reference(accountServant));
        }catch (Exception e){
          e.printStackTrace();
        }
        // Print out the new account.
        System.out.println(
              "Created " + name + "'s account: " + account);
        // Save the account in the account dictionary.
        _accounts.put(name, account);
     }
```

```
        // Return the account.
        return account;
    }
    private Dictionary _accounts =  new Hashtable();
    private Random _random =  new Random();
}
```

AccountManager インプリメンテーションは，サーバコードで作成し活性化しなければなりません。このサンプルでは，AccountManager は，記録先であるアクティブオブジェクトマップにオブジェクト ID を渡す activate_object_with_id を使用して活性化されます。アクティブオブジェクトマップは，オブジェクト ID をサーバントにマッピングする単なるテーブルです。この手法はオブジェクトの明示的な活性化と呼ばれ，POA がアクティブなときは常にこのオブジェクトが使用できるようにします。

コードサンプル 6-10　サーバントの作成と活性化（C++）

```
// Create the servant
AccountManagerImpl managerServant;
// Decide on the ID for the servant
PortableServer::ObjectId_var managerId =
      PortableServer::string_to_ObjectId("BankManager");
// Activate the servant with the ID on myPOA
myPOA->activate_object_with_id(managerId,&managerServant);
```

コードサンプル 6-11　サーバントの作成と活性化（Java）

```
// Create the servant
AccountManagerImpl managerServant =
                                    new AccountManagerImpl();
// Decide on the ID for the servant
byte[ ] managerId = "BankManager".getBytes();
// Activate the servant with the ID on myPOA
myPOA.activate_object_with_id(managerId, managerServant);
```

6.3.4　POA の活性化

最後の手順は，使用する POA に対応する POA マネージャの活性化です。デフォルトでは，POA マネージャは待機状態で作成されます。この状態では，すべてのリクエストは保留待ち行列に転送され，処理されません。リクエストをディスパッチできるようにするには，POA に対応する POA マネージャを待機状態からアクティブな状態に変えなければなりません。POA マネージャとは，POA の状態（リクエストを待ち行列に入れるか，処理するか，または破棄するか）を制御するオブジェクトに過ぎません。POA 生成時に POA マネージャは POA に対応づけられます。使用する POA マネージャを指定でき，システムに新しいものを作成させることもできます（POA マネージャ名として create_POA()に，C++では NULL，Java では null を入力してください）。

コードサンプル 6-12　POA マネージャの活性化（C++）

```
// Activate the POA manager
PortableServer::POAManager_var mgr=rootPoa ->the_POAManager();
mgr->activate();
```

コードサンプル 6-13　POA マネージャの活性化（Java）

```
// Activate the POA manager
rootPOA.the_POAManager().activate();
```

6.4　オブジェクトの活性化

前節では, オブジェクトの明示的な活性化について簡単に説明しました。オブジェクトを活性化するには幾つかの方法があります。

明示的な活性化

POA の呼び出しによるサーバの起動時にすべてのオブジェクトが活性化されます。

オンデマンドによる活性化

まだオブジェクト ID に対応していないサーバントに対するリクエストをサーバントマネージャが受信すると，サーバントマネージャはオブジェクトを活性化します。

暗黙的な活性化

クライアントリクエストではなく，POA によるオペレーションを契機としてサーバが暗黙的にオブジェクトを活性化します。

デフォルトサーバントによる活性化

POA はデフォルトサーバントを使用してクライアントリクエストを処理します。

オブジェクトの活性化については,「7.　POA の使用」を参照してください。ここでは，オブジェクトを活性化するには幾つかの方法があるということだけを意識してください。

6.5　クライアントリクエストを待つ

POA の設定が完了したら，orb.run()を使用してクライアントリクエストを待つことができます。このプロセスはサーバが終了するまで動作します。

コードサンプル 6-14　入力リクエストを待つ（C++）

```
// Wait for incoming requests.
orb->run();
```

コードサンプル 6-15　入力リクエストを待つ（Java）

```
// Wait for incoming requests
orb.run();
```

6.6　コードサンプルのまとめ

ここでは，この章で説明したコード全体を示します。コードサンプル 6-16 は，この章で説明した C++のコード全体を示します。コードサンプル 6-17 は，この章で説明した Java のコード全体を示します。

コードサンプル 6-16　サーバ側コード全体（C++）

```
// Server.C
#include "Bank_s.hh"
#include <math.h>

class Dictionary {
   private:
      struct Data {
         const char* name;
         void* value;
      };

      unsigned _count;
      Data* _data;
   public:
      Dictionary() {
      _count = 0;
   }
   void put(const char* name, void* value) {
      Data* oldData = _data;
      _data = new Data[_count +1];
      for(unsigned i = 0; i < _count; i++) {
         _data[i] = oldData[i];
      }
      _data[_count].name = strdup(name);
      _data[_count].value = value;
      _count++;
   }
   void* get(const char* name) {
      for(unsigned i =0; i < _count; i++) {
         if(!strcmp(name, _data[i].name)) {
            return _data[i].value;
         }
      }
      return 0;
   }
};

class AccountImpl : public POA_Bank::Account {
   private:
      float _balance;
   public:
      AccountImpl(float balance) {
         _balance = balance;
      }
      virtual float balance() {
         return _balance;
   }
};

class AccountManagerImpl : public POA_Bank::AccountManager {
   private:
      Dictionary _accounts;
   public:
      virtual Bank::Account_ptr open(const char* name) {
         // Lookup the account in the account dictionary.
         Bank::Account_ptr account =
               (Bank::Account_ptr) _accounts.get(name);

         if(account == Bank::Account::_nil()) {
            // Make up the account's balance, between
            // 0 and 1000 dollars.
            float balance = abs(rand()) % 100000 / 100.0;
            // Create the account implementation, given
            // the balance.
            AccountImpl *accountServant =
```

```
                    new AccountImpl(balance);
                try {
                   // Activate it on the default POA which is
                   // rootPOA for this servant
                   PortableServer::POA_var rootPOA = _default_POA();
                   CORBA::Object_var obj =
                      rootPOA->servant_to_reference(accountServant);
                   account = Bank::Account::_narrow(obj);
                   } catch(const CORBA::Exception& e) {
                      cerr << "_narrow caught exception: " << e
                           << endl;
                   }
                   //Print out the new account.
                   cout << "Created " << name << "'s account: "
                        << account << endl;
                   // Save the account in the account dictionary.
                   _accounts.put(name, account);
                }
                // Return the account.
                return Bank::Account::_duplicate(account);
          }
    };
    int main(int argc, char* const* argv) {
       try {
          // Initialize the ORB.
          CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

          // get a reference to the rootPOA
          CORBA::Object_var obj =
                orb->resolve_initial_references("RootPOA");
          // narrow the object reference to a POA reference
          PortableServer::POA_var rootPOA =
                PortableServer::POA::_narrow(obj);

          CORBA::PolicyList policies;
          policies.length(1);

          policies[(CORBA::ULong)0] =
                rootPOA->create_lifespan_policy(
                PortableServer::PERSISTENT
          );
          // Create myPOA with the right policies
          PortableServer::POAManager_var rootManager =
                rootPOA->the_POAManager();
          PortableServer::POA_var myPOA =
                rootPOA->create_POA( "bank_agent_poa",
                rootManager,policies );

          // Create the servant
          AccountManagerImpl managerServant;

          // Decide on the ID for the servant
          PortableServer::ObjectId_var managerId =
                PortableServer::string_to_ObjectId("BankManager");

          // Activate the servant with the ID on myPOA
          myPOA->activate_object_with_id(managerId,&managerServant);

          // Activate the POA Manager
          rootPOA->the_POAManager()->activate();

          cout << myPOA->servant_to_reference(&managerServant)
                << " is ready" << endl;

          // Wait for incoming requests
          orb->run();
       } catch(const CORBA::Exception& e) {
          cerr << e << endl;
       }
    }
```

コードサンプル 6-17　サーバ側コード全体（Java）

```
// Server.java
import org.omg.PortableServer.*;
```

```
public class Server {
  public static void main(String[ ] args){
   try {
     // Initialize the ORB.
     org.omg.CORBA.ORB orb =
                          org.omg.CORBA.ORB.init(args, null);
     // get a reference to the rootPOA
     POA rootPOA = POAHelper.narrow(
       orb.resolve_initial_references("RootPOA"));

     // Create policies for our persistent POA
     org.omg.CORBA.Policy[ ] policies = {
        rootPOA.create_lifespan_policy(
                             LifespanPolicyValue.PERSISTENT)
     };
     // Create myPOA with the right policies
     POA myPOA = rootPOA.create_POA( "bank_agent_poa",
                                     rootPOA.the_POAManager(),
                                     policies );

     // Create the servant
     AccountManagerImpl managerServant =
                                    new AccountManagerImpl();
     // Decide on the ID for the servant
     byte[ ] managerId = "BankManager".getBytes();
     // Activate the servant with the ID on myPOA
     myPOA.activate_object_with_id(managerId,
                                    managerServant);
     // Activate the POA manager
     rootPOA.the_POAManager().activate();
     System.out.println(
       myPOA.servant_to_reference(managerServant) +
       "is ready.");
     // Wait for incoming requests
     orb.run();
   }catch (Exception e) {
     e.printStackTrace();
   }
  }
}
```

7 POA の使用

この章では，POA の使用について説明します。

7.1　ポータブルオブジェクトアダプタとは

POA（ポータブルオブジェクトアダプタ）は，BOA（基本オブジェクトアダプタ）に代わり，サーバ側のポータビリティを提供します。

POA は，オブジェクトのインプリメンテーションと VisiBroker ORB 間の中間アダプタです。中間アダプタの役割として，POA はリクエストをサーバントに転送し，その結果サーバントが動作して，必要に応じて子 POA が生成されます。

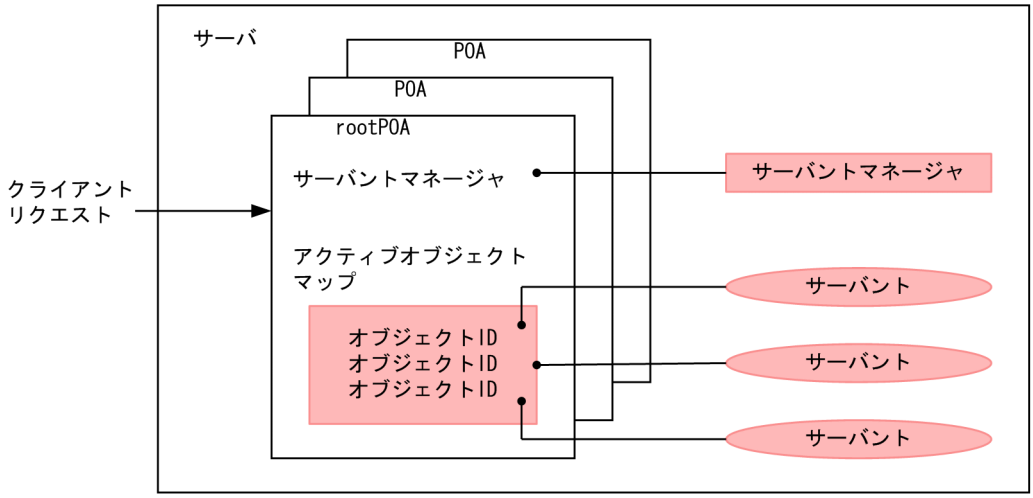
各サーバは，複数の POA をサポートできます。少なくとも一つの POA（rootPOA）がなければなりません。rootPOA は自動的に生成されます。POA のセットは階層型になっており，すべての POA にはその親として rootPOA があります。

サーバントマネージャは，サーバントを探して POA のオブジェクトに対応させます。abstract オブジェクトがサーバントに割り当てられると，そのオブジェクトはアクティブなオブジェクトと呼ばれ，そのサーバントはアクティブなオブジェクトをインカネートすると言われます。それぞれの POA にはアクティブオブジェクトマップが一つずつあり，このマップによって，アクティブなオブジェクトのオブジェクト ID とアクティブなサーバントが対応づけられます。POA の概要を図 7-1 に示します。

注

ここでは，POA の主要なテーマだけに焦点を当てています。詳細な説明については，OMG 仕様を参照してください。

図 7-1　POA の概要



7.1.1　POA 用語

POA について説明する前に，幾つかの用語を表 7-1 に定義しておきます。

表 7-1　ポータブルオブジェクトアダプタ用語

用語	説明
アクティブオブジェクトマップ	アクティブな VisiBroker CORBA オブジェクトを（そのオブジェクト ID によって）サーバントにマッピングするテーブルです。それぞれの POA にはアクティブオブジェクトマップが一つずつあります。

用語	説明
アダプタアクティベータ	存在しない子 POA に対するリクエストを受信したときにオンデマンドで POA を生成できるオブジェクトです。
エーテライズ	サーバントと抽象化された CORBA オブジェクト間の対応を削除することです。
インカネート	サーバントを抽象化された CORBA オブジェクトに対応づけることです。
オブジェクト ID	オブジェクトアダプタ内の CORBA オブジェクト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ることができ，オブジェクト ID の作成元であるオブジェクトアダプタ内だけで一意です。サーバントはオブジェクト ID によって abstract オブジェクトに対応づけられます。
パーシステントオブジェクト	作成元であるサーバプロセスが消滅しても存在する CORBA オブジェクトです。
POA マネージャ	POA が入力リクエストを受信するか破棄するかなどの POA の状態を管理するオブジェクトです。
ポリシー	対応する POA の動作および POA が管理するオブジェクトを制御するオブジェクトです。
rootPOA	各 VisiBroker ORB は rootPOA という一つの POA で生成されます。必要なら rootPOA から追加の POA を生成できます。
サーバント	CORBA オブジェクトのメソッドをインプリメントするが，CORBA オブジェクト自身はインプリメントしないコードです。
サーバントマネージャ	オブジェクトとサーバントとの対応の管理と，オブジェクトが存在するかどうかの決定の責任を負うオブジェクトです。一つ以上のサーバントマネージャを存在させることができます。
トランジェントオブジェクト	作成元であるプロセス内でしか存在できない CORBA オブジェクトです。

7.1.2　POA の作成および使用手順

正確にはさまざまな手順がありますが，POA のライフサイクル中に行われる基本手順は次のとおりです。

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

稼働条件によっては，これらの手順の幾つかが任意になる場合があります。例えば，リクエストを処理するために POA が必要な場合，その POA だけを活性化すればよいのです。

7.2 POA ポリシー

各 POA は，その特徴を定義するポリシーの集合から成ります。新しい POA を作成する場合は，各ポリシーのデフォルトを使用するか，または要件に合わせて異なる値を使用できます。ポリシーは POA の作成時だけ設定でき，既存 POA のポリシーは変更できません。POA は親 POA のポリシーは継承しません。

次に，POA ポリシーとその値，およびデフォルト値（rootPOA が使用）を説明します。

(1) スレッドポリシー

スレッドポリシーでは，POA が使用するスレッドモデルを指定します。ThreadPolicy に使用できる値は次のとおりです。

ORB_CTRL_MODEL（デフォルト）

POA はリクエストをスレッドに割り当てる責任を負います。マルチスレッド環境では，同時に発生した複数のリクエストは，複数のスレッドで処理されます。Borland Enterprise Server VisiBroker はマルチスレッドモデルを使用することに注意してください。

SINGLE_THREAD_MODEL

POA プロセスはリクエストを順次処理します。マルチスレッド環境では，POA がサーバントおよびサーバントマネージャに行う呼び出しはすべてスレッドセーフです。

MAIN_THREAD_MODEL

呼び出しは識別されたメインスレッドで処理されます。このポリシーを使用する POA に対するすべてのリクエストは，順次処理されます。マルチスレッド環境では，このポリシーを使用する POA によって呼び出されるサーバントのすべての処理は，スレッドセーフでなければなりません。

アプリケーションプログラマは，ORB::run()メソッドまたは ORB::perform_work()メソッドを呼び出してメインスレッドを指定します。このメソッドの詳細については，「7.4　オブジェクトの活性化」を参照してください。

(2) ライフスパンポリシー

ライフスパンポリシーでは，POA でインプリメントされたオブジェクトのライフスパンを指定します。LifespanPolicy に使用できる値は次のとおりです。

TRANSIENT（デフォルト）

POA が活性化したトランジェントオブジェクトは，そのオブジェクトが生成された POA 内でしか存在できません。POA が非活性化されてから，POA が生成したオブジェクトリファレンスを使おうとすると，OBJECT_NOT_EXIST 例外が発生します。

PERSISTENT

POA が活性化したパーシステントオブジェクトは，そのオブジェクトが最初に生成されたプロセスが消滅しても存在できます。パーシステントオブジェクトで呼び出されたリクエストは，プロセス，POA，およびオブジェクトをインプリメントしたサーバントを暗黙的に活性化します。

(3) ID Uniqueness ポリシー

ID Uniqueness（オブジェクト ID の一意性）ポリシーでは，多くの abstract オブジェクトが一つのサーバントを共有できるようにします。IdUniquenessPolicy に使用できる値は次のとおりです。

UNIQUE_ID（デフォルト）

活性化されたサーバントはオブジェクト ID を一つだけサポートします。

MULTIPLE_ID

活性化されたサーバントは一つ以上のオブジェクト ID を持てます。オブジェクト ID はランタイムに呼び出されているメソッド内で決定しなければなりません。

(4)　ID Assignment ポリシー

ID Assignment（ID 割り当て）ポリシーでは，サーバアプリケーションと POA のどちらがオブジェクト ID を生成するかを指定します。IdAssignmentPolicy に使用できる値は次のとおりです。

USER_ID

オブジェクトはアプリケーションによってオブジェクト ID を割り当てられます。

SYSTEM_ID（デフォルト）

オブジェクトは POA によってオブジェクト ID を割り当てられます。PERSISTENT ポリシーも設定している場合，オブジェクト ID は同じ POA のすべての実体化にわたって一意でなければなりません。

通常，USER_ID はパーシステントオブジェクト用であり，SYSTEM_ID はトランジェントオブジェクト用です。

(5)　Servant Retention ポリシー

Servant Retention（サーバント保持）ポリシーでは，アクティブオブジェクトマップ内のアクティブなサーバントを POA が保持するかどうかを指定します。ServantRetentionPolicy に指定できる値は次のとおりです。

RETAIN（デフォルト）

POA はアクティブオブジェクトマップにアクティブなサーバントを保持します。そのため，POA にオブジェクトの探索を要求すると，アクティブオブジェクトマップからの探索が行われます。RETAIN は通常，ServantActivator または POA の明示的な活性化メソッドと一緒に使用します。

NON_RETAIN

POA はアクティブオブジェクトマップにアクティブなサーバントを保持しません。そのため，POA にオブジェクトの探索を要求しても，アクティブオブジェクトマップからの探索は行われません。

NON_RETAIN は通常，ServantLocator と一緒に使用しなければなりません。

ServantActivator と ServantLocator は，サーバントマネージャのタイプです。サーバントマネージャの詳細については，「7.5　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。

(6)　Request Processing ポリシー

Request Processing（リクエスト処理）ポリシーでは，POA のリクエストの処理方法を指定します。RequestProcessingPolicy に指定できる値は次のとおりです。

USE_ACTIVE_OBJECT_MAP_ONLY（デフォルト）

アクティブオブジェクトマップにオブジェクト ID が存在しなければ，OBJECT_NOT_EXIST 例外が返されます。POA にこの値を指定する場合は，必ず RETAIN ポリシーも指定してください。

USE_DEFAULT_SERVANT

アクティブオブジェクトマップにオブジェクト ID が存在していないか，または NON_RETAIN ポリシーが設定されていれば，リクエストはデフォルトサーバントにディスパッチされます。デフォルトサーバントが登録されていなければ，OBJ_ADAPTER 例外が返されます。POA にこの値を指定する場合は，必ず MULTIPLE_ID ポリシーも指定してください。

USE_SERVANT_MANAGER

アクティブオブジェクトマップにオブジェクト ID が存在していないか，または NON_RETAIN ポリシーが設定されていれば，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。

(7) Implicit Activation ポリシー

Implicit Activation（暗黙的な活性化）ポリシーでは，POA がサーバントの暗黙的な活性化をサポートするかどうかを指定します。ImplicitActivationPolicy に指定できる値は次のとおりです。

IMPLICIT_ ACTIVATION

POA はサーバントの暗黙的な活性化をサポートします。サーバントを活性化するには，次の二つの方法があります。

- POA::servant_to_reference()メソッド（C++），または org.omg.PortableServer.POA.servant_to_reference()メソッド（Java）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。
- サーバントで_this()メソッドを呼び出す。

POA にこの値を指定する場合は，必ず SYSTEM_ID ポリシーと RETAIN ポリシーも指定してください。

NO_IMPLICIT_ACTIVATION（デフォルト）

POA はサーバントの暗黙的な活性化をサポートしません。

(8) Bind Support ポリシー

Bind Support（バインドサポート）ポリシー（Borland Enterprise Server VisiBroker 固有のポリシー）は POA とアクティブなオブジェクトの Borland Enterprise Server VisiBroker osagent への登録を制御します。数千ものオブジェクトがある場合，そのすべてを osagent に登録するのは好ましくありません。その代わりに，POA を osagent に登録できます。クライアントがリクエストすると，osagent がリクエストを正しく転送できるように，POA 名とオブジェクト ID がバインドリクエストに含まれます。

BindSupportPolicy に指定できる値は次のとおりです。

BY_INSTANCE

すべてのアクティブなオブジェクトを osagent に登録します。POA にこの値を指定する場合は，必ず PERSISTENT ポリシーと RETAIN ポリシーも指定してください。

BY_POA（デフォルト）

POA だけを osagent に登録します。POA にこの値を指定する場合は，必ず PERSISTENT ポリシーも指定してください。

NONE

POA もアクティブなオブジェクトも osagent に登録しません。

7.3　POA の作成

POA を使用してオブジェクトをインプリメントするには，最低一つの POA オブジェクトがサーバ上に存在しなければなりません。POA が確実に存在するように，VisiBroker ORB の初期化中に rootPOA が提供されます。この POA は「7.2　POA ポリシー」で説明したデフォルトの POA ポリシーになります。

rootPOA を取得すれば，特定のサーバ側ポリシーのセットをインプリメントする子 POA を作成できます。

7.3.1　POA ネーミング規則

各 POA は自身の名前と完全な POA 名称（完全階層パス名）を常に把握しています。階層はスラント（/）で示されます。例えば，/A/B/C は，POA C は POA B の子であり，POA B は POA A の子であるという意味です。最初のスラントは rootPOA を示します。Bind Support:BY_POA ポリシーを POA C に設定すると，/A/B/C は osagent に登録され，クライアントは/A/B/C にバインドできるようになります。

使用する POA 名にエスケープ文字またはそれ以外の区切り文字が含まれている場合，名前を内部で記録するときに，Borland Enterprise Server VisiBroker はこれらの文字の前に二重¥マーク（¥¥）を付加します。例えば，次のような階層に二つの POA があるとします。

C++の場合

```
PortableServer::POA_var myPOA1 =
                          rootPOA->create_POA("A/B",
                          poa_manager,
                          policies);
PortableServer::POA_var myPOA2 =
                          myPOA1->create_POA("¥t",
                          poa_manager,
                          policies);
```

クライアントがバインドするときには，次のように使用します。

```
Bank::AccountManager_var manager = Bank::AccountManager::_bind
                        ("/A¥¥/B/¥t", managerId);
```

Java の場合

```
org.omg.PortableServer.POA myPOA1 =
                          rootPOA.create_POA("A/B",
                                             poaManager,
                                             policies);
org.omg.PortableServer.POA myPOA2 =
                          myPOA1.create_POA("¥t",
                                             poaManager,
                                             policies);
```

クライアントがバインドするときには，次のように使用します。

```
org.omg.CORBA.Object manager =
        ((com.inprise.vbroker.orb.ORB) orb).bind("/A¥¥/B/¥t",
          managerId,
          null,
          null);
```

7.3.2　rootPOA の取得

コードサンプル 7-1 および 7-2 に，サーバアプリケーションがどのように rootPOA を取得できるかを示します。

コードサンプル 7-1　rootPOA の取得（C++）

```
// Initialize the ORB.
CORBA::Object_var obj =
      orb->resolve_initial_references("RootPOA");
// get a reference to the rootPOA
PortableServer::POA_var rootPOA =
      PortableServer::POA::_narrow(obj);
```

コードサンプル 7-2　rootPOA の取得（Java）

```
// Initialize the ORB.
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
// get a reference to the rootPOA
org.omg.PortableServer.POA rootPOA =
      POAHelper.narrow(
                  orb.resolve_initial_references("RootPOA"));
```

注

resolve_initial_references メソッドは，CORBA::Object 型の値（C++）または org.omg.CORBA.Object 型の値（Java）を返します。返されたオブジェクトリファレンスを希望の型（上記のサンプルでは PortableServer::POA（C++），または org.omg.PortableServer.POA（Java））にナロウするのはプログラマの責任です。

7.3.3　POA プロパティの設定

親 POA のポリシーは継承されません。POA に特定の特徴を持たせたければ，デフォルト値と異なるすべてのポリシーを特定する必要があります。POA ポリシーの詳細については，「7.2　POA ポリシー」を参照してください。

コードサンプル 7-3　POA のポリシーの作成例（C++）

```
CORBA::PolicyList policies;
policies.length(1);
policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy
      (PortableServer::PERSISTENT);
```

コードサンプル 7-4　POA のポリシーの作成例（Java）

```
org.omg.CORBA.Policy[ ] policies = {
   rootPOA.create_lifespan_policy(LifespanPolicyValue.
                                  PERSISTENT)
};
```

7.3.4　POA の作成と活性化

POA は，その親 POA の create_POA を使用して生成されます。POA には任意の名前を付けられます。ただし，同じ親を持つ POA 同士の名前はすべて一意でなければなりません。二つの 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 を生成するには，次のように create_POA を使用してください。

```
POA create_POA(POA_Name, POAManager, PolicyList);
```

POA マネージャは POA の状態（リクエストを処理するかどうかなど）を制御します。POA マネージャ名として null が create_POA に渡されると，新しい POA マネージャオブジェクトが生成されて POA に対応づけられます。通常は，すべての POA に対して同じ POA マネージャを持ちたいでしょう。POA マネージャの詳細については，「7.6　POA マネージャによる POA 管理」を参照してください。

POA マネージャ（および POA）は，生成後に自動的に活性化されるわけではありません。使用している POA に対応する POA マネージャを活性化するには，activate()メソッドを使用してください。

コードサンプル 7-5　POA の作成例（C++）

```
CORBA::PolicyList policies;
policies.length(1);
policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy
      (PortableServer::PERSISTENT);
// Create myPOA with the right policies
PortableServer::POAManager_var rootManager =
      rootPOA->the_POAManager();
PortableServer::POA_var myPOA =
      rootPOA->create_POA("bank_agent_poa",
      rootManager, policies);
```

コードサンプル 7-6　POA の作成例（Java）

```
// Create policies for our persistent POA
org.omg.CORBA.Policy[ ] policies = {
      rootPOA.create_lifespan_policy(
                        LifespanPolicyValue.PERSISTENT)};
// Create myPOA with the right policies
org.omg.PortableServer.POA myPOA =
      rootPOA.create_POA( "bank_agent_poa",
                        rootPOA.the_POAManager(),
                        policies );
```

7.4　オブジェクトの活性化

CORBA オブジェクトがアクティブなサーバントと対応づけられると，POA の Servant Retention（サーバント保持）ポリシーが RETAIN である場合，対応するオブジェクト ID がアクティブオブジェクトマップに記録され，そのオブジェクトが活性化されます。活性化するには幾つかの方法があります。

明示的な活性化

サーバアプリケーション自身が activate_object または activate_object_with_id を呼び出すことによって明示的にオブジェクトを活性化します。

オンデマンドによる活性化

サーバアプリケーションは，ユーザ提供のサーバントマネージャを介して POA にオブジェクトを活性化するよう指示します。set_servant_manager を使用して，まず POA にサーバントマネージャを登録しなければなりません。

暗黙的な活性化

幾つかのオペレーションに対する応答として，サーバが単独でオブジェクト活性化を行います。サーバントがアクティブでなければ，クライアントがサーバントをアクティブにできる方法はありません（例えば，非アクティブなオブジェクトをリクエストしてもオブジェクトをアクティブにはできません）。

デフォルトサーバントによる活性化

POA はサーバントを一つだけ使用して，すべてのオブジェクトをインプリメントします。

7.4.1　オブジェクトの明示的な活性化

IdAssignmentPolicy::SYSTEM_ID を POA に設定すれば，オブジェクト ID の指定をしないでオブジェクトを明示的に活性化できます。サーバはオブジェクトのオブジェクト ID の活性化，割り当て，リターンを行う POA の activate_object を呼び出します。このタイプの活性化はトランジェントオブジェクトでは最もよく使用されます。オブジェクトもサーバントもあまり長期間は必要ではないので，サーバントマネージャは不要です。

オブジェクト ID を使用してオブジェクトを明示的に活性化することもできます。一般的なシナリオは，サーバが管理するすべてのオブジェクトを活性化するためにユーザが activate_object_with_id を呼び出すサーバ初期化中です。すべてのオブジェクトは活性化済みなので，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リクエストを受信すると，OBJECT_NOT_EXIST 例外が発生します。サーバが多数のオブジェクトを管理している場合，お勧めできません。

コードサンプル 7-7　activate_object_with_id を使用した明示的な活性化の例（C++）

```
// Create the servant
AccountManagerImpl managerServant;
// Decide on the ID for the servant
PortableServer::ObjectId_var managerId =
      PortableServer::string_to_ObjectId("BankManager");
// Activate the servant with the ID on myPOA
myPOA->activate_object_with_id(managerId,&managerServant);
// Activate the POA Manager
PortableServer::POAManager_var rootManager =
      rootPOA->the_POAManager();
rootManager->activate();
```

コードサンプル 7-8　activate_object_with_id を使用した明示的な活性化の例（Java）

```
// Create the account manager servant.
Servant managerServant = new AccountManagerImpl(rootPoa);
// Activate the newly created servant.
testPoa.activate_object_with_id(
                  "BankManager".getBytes(), managerServant);
```

```
// Activate the POAs
testPoa.the_POAManager().activate();
```

7.4.2　オブジェクトのオンデマンドによる活性化

対応するサーバントを持たないオブジェクトをクライアントがリクエストすると，オンデマンドによる活性化が発生します。このリクエストの受信後，POA はオブジェクト ID に対応するアクティブなサーバントを求めてアクティブオブジェクトマップを検索します。見つからなければ，POA はオブジェクト ID 値をサーバントマネージャに渡すサーバントマネージャの incarnate を呼び出します。サーバントマネージャができることは次の三つがあり，どれか一つを行います。

- 適切なサーバントを見つけます。このサーバントはリクエストに対する適切なオペレーションを実行します。
- クライアントに返される OBJECT_NOT_EXIST 例外を発生させます。
- リクエストをほかのオブジェクトに転送します。

POA ポリシーは，追加する処理を決定します。次に C++および Java について説明します。

C++の場合

RequestProcessingPolicy::USE_SERVANT_MANAGER と ServantRetentionPolicy::RETAIN が有効なら，アクティブオブジェクトマップはサーバントとオブジェクト ID の対応を更新します。

Java の場合

RequestProcessingPolicy.USE_SERVANT_MANAGER と ServantRetentionPolicy.RETAIN が有効なら，アクティブオブジェクトマップはサーバントとオブジェクト ID の対応を更新します。

オンデマンドによる活性化の例については，コードサンプル 7-13 を参照してください。

7.4.3　オブジェクトの暗黙的な活性化

(1)　C++の場合

ImplicitActivationPolicy::IMPLICIT_ACTIVATION，IdAssignmentPolicy::SYSTEM_ID，および ServantRetentionPolicy::RETAIN を設定して POA を活性化した場合，あるオペレーションでサーバントを暗黙的に活性化できます。暗黙的な活性化は次のメソッドで行えます。

- POA::servant_to_reference メソッド
- POA::servant_to_id メソッド
- _this()サーバントメソッド

POA に ObjectIdUniquenessPolicy::UNIQUE_ID を設定すると，非アクティブなサーバントに対して上記のオペレーションのどれかを行った場合に暗黙的な活性化ができます。

POA に ObjectIdUniquenessPolicy::MULTIPLE_ID を設定すると，たとえサーバントがすでにアクティブでも，servant_to_reference オペレーションと servant_to_id オペレーションは常に暗黙的な活性化を行います。

(2)　Java の場合

ImplicitActivationPolicy.IMPLICIT_ACTIVATION，IdAssignmentPolicy.SYSTEM_ID，および ServantRetentionPolicy.RETAIN を設定して POA を活性化した場合，あるオペレーションでサーバントを暗黙的に活性化できます。暗黙的な活性化は次のメソッドで行えます。

- POA.servant_to_reference メソッド
- POA.servant_to_id メソッド
- _this()サーバントメソッド

POA に ObjectIdUniquenessPolicy.UNIQUE_ID を設定すると，非アクティブなサーバントに対して上記のオペレーションのどれかを行った場合に暗黙的な活性化ができます。

POA に ObjectIdUniquenessPolicy.MULTIPLE_ID を設定すると，たとえサーバントがすでにアクティブでも，servant_to_reference オペレーションと servant_to_id オペレーションは常に暗黙的な活性化を行います。

7.4.4　デフォルトサーバントによる活性化

オブジェクト ID が何であろうと POA に同じサーバントを起動させるには，RequestProcessing::USE_DEFAULT_SERVANT ポリシー（C++），または RequestProcessing.USE_DEFAULT_SERVANT ポリシー（Java）を使用してください。これは各オブジェクトに少量のデータしか持たせていない場合に便利です。

コードサンプル 7-9　同じサーバントによるすべてのオブジェクトの活性化例（C++）

```
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      PortableServer::Current_var cur =
            PortableServer::Current::_instance();
      DataStore::_create();

      // get a reference to the rootPOA
      CORBA::Object_var obj =
            orb->resolve_initial_references("RootPOA");
      PortableServer::POA_var rootPOA =
            PortableServer::POA::_narrow(obj);

      CORBA::PolicyList policies;
      policies.length(3);

      // Create policies for our persistent POA
      policies[(CORBA::ULong)0] =
            rootPOA->create_lifespan_policy
            (PortableServer::PERSISTENT);
      policies[(CORBA::ULong)1] =
            rootPOA->create_request_processing_policy
            (PortableServer::USE_DEFAULT_SERVANT);
      policies[(CORBA::ULong)2] =
            rootPOA->create_id_uniqueness_policy
            (PortableServer::MULTIPLE_ID);

      // Create myPOA with the right policies
      PortableServer::POAManager_var rootManager =
            rootPOA->the_POAManager();
      PortableServer::POA_var myPOA =
            rootPOA->create_POA
            ("bank_default_servant_poa",
            rootManager, policies);
      // Set the default servant
      AccountManagerImpl * managerServant =
            new AccountManagerImpl(cur);
      myPOA->set_servant( managerServant );

      // Activate the POA Manager
      rootManager->activate();

      // Generate two references - one for checking
      // and another for savings.
      // Note that we are not creating any
```

```
        // servants here and just manufacturing a reference
        // which is not yet backed by a servant
        PortableServer::ObjectId_var an_oid =
              PortableServer::string_to_ObjectId
              ("CheckingAccountManager");
        CORBA::Object_var cref =
              myPOA->create_reference_with_id(an_oid.in(),
              "IDL:Bank/AccountManager:1.0");

        an_oid = PortableServer::string_to_ObjectId
              ("SavingsAccountManager");
        CORBA::Object_var sref = myPOA->create_reference_with_id
              (an_oid.in(), "IDL:Bank/AccountManager:1.0");

        // Write out Checking reference
        CORBA::String_var string_ref =
              orb->object_to_string(cref.in());
        ofstream crefFile("cref.dat");
        crefFile << string_ref << endl;
        crefFile.close();
        // Now write out the Savings reference
        string_ref = orb->object_to_string(sref.in());
        ofstream srefFile("sref.dat");
        srefFile << string_ref << endl;
        srefFile.close();
        cout << "Bank Manager is ready" << endl;
        // Wait for incoming requests
        orb->run();
        DataStore::_destroy();
     }
     catch(const CORBA::Exception& e) {
        cerr << e << endl;
     }
     return 1;
  }
```

コードサンプル 7-10　同じサーバントによるすべてのオブジェクトの活性化例（Java）

```
  import org.omg.PortableServer.*;
  public class Server {
     public static void main(String[ ] args) {
        try {
           // Initialize the ORB.
           org.omg.CORBA.ORB orb
                        = org.omg.CORBA.ORB.init(args,null);
           // get a reference to the rootPOA
           POA rootPOA = POAHelper.narrow(
             orb.resolve_initial_references("RootPOA"));
           // Create policies for our persistent POA
           org.omg.CORBA.Policy[ ] policies = {
             rootPOA.create_lifespan_policy(
                   LifespanPolicyValue.PERSISTENT),
             rootPOA.create_request_processing_policy(
                   RequestProcessingPolicyValue.USE_DEFAULT_SERVANT
             )
           };
           // Create myPOA with the right policies
           POA myPOA = rootPOA.create_POA(
                 "bank_default_servant_poa",
                 rootPOA.the_POAManager(),
                 policies );
           // Create the servant
           AccountManagerImpl managerServant =
                                      new AccountManagerImpl();
           // Set the default servant on our POA
           myPOA.set_servant(managerServant);
           org.omg.CORBA.Object ref;
           // Activate the POA manager
           rootPOA.the_POAManager().activate();
           // Generate the reference and write it out. One
           // for each Checking and Savings account types.
           // Note that we are not creating any servants
           // here and just manufacturing a reference
           // which is not yet backed by a servant.
           try {
```

```
            ref = myPOA.create_reference_with_id(
                                "CheckingAccountManager".getBytes(),
                                "IDL:Bank/AccountManager:1.0");
            // Write out checking object ID
            java.io.PrintWriter pw = new java.io.PrintWriter(
                  new java.io.FileWriter("cref.dat"));
            pw.println(orb.object_to_string(ref));
            pw.close();
            ref = myPOA.create_reference_with_id(
                              "SavingsAccountManager".getBytes(),
                              "IDL:Bank/AccountManager:1.0");
            // Write out savings object ID
            pw = new java.io.PrintWriter(
                   new java.io.FileWriter("sref.dat"));
            System.gc();
            pw.println(orb.object_to_string(ref));
            pw.close();
         } catch ( java.io.IOException e ){
            System.out.println("Error writing the IOR to file ");
            return;
         }
         System.out.println("Bank Manager is ready.");
         // Wait for incoming requests
         orb.run();
       }catch (Exception e){
         e.printStackTrace();
       }
     }
   }
```

7.4.5　オブジェクトの非活性化

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。例えば，これはガーベッジコレクション手法の一形態として行います。サーバントをマップから削除すると,そのサーバントは非活性化されます。deactivate_object()メソッドを使用してオブジェクトを非活性化できます。オブジェクトを非活性化すると，このオブジェクトが永遠に失われるということではありません。あとで再度活性化できます。

コードサンプル 7-11　オブジェクトの非活性化例（C++）

```
// DeActivatorThread
class DeActivatorThread: public VISThread {
   private :
      PortableServer::ObjectId _oid;
      PortableServer::POA_ptr _poa;

   public :
      virtual ~DeActivatorThread(){}
      // Constructor
      DeActivatorThread(const PortableServer::ObjectId& oid,
            PortableServer::POA_ptr poa ): _oid(oid),
            _poa(poa) {
         // start the thread
         run();
      }

      // implement begin() callback
      void begin() {
         // Sleep for 15 seconds
         VISPortable::vsleep(15);

         CORBA::String_var s =
            PortableServer::ObjectId_to_string (_oid);
         // Deactivate Object
         cout << "¥nDeactivating the object with ID =
            " << s << endl;
         if ( _poa )
            _poa->deactivate_object( _oid );
      }
};

// Servant Activator
```

```
class AccountManagerActivator :
      public PortableServer::ServantActivator {
   public:
      virtual PortableServer::Servant incarnate
            (const PortableServer::ObjectId& oid,
            PortableServer::POA_ptr poa) {
         CORBA::String_var s =
            PortableServer::ObjectId_to_string (oid);
         cout << "¥nAccountManagerActivator.incarnate called
            with ID = " << s << endl;
         PortableServer::Servant servant;

         if ( VISPortable::vstricmp( (char *)s,
               "SavingsAccountManager" ) == 0 )
            // Create CheckingAccountManager Servant
            servant = new SavingsAccountManagerImpl;
         else if ( VISPortable::vstricmp( (char *)s,
               "CheckingAccountManager" ) == 0 )
            // Create CheckingAccountManager Servant
            servant = new CheckingAccountManagerImpl;
         else
            throw CORBA::OBJECT_NOT_EXIST();
         // Create a deactivator thread
         new DeActivatorThread( oid, poa );
         // return the servant
         servant->_add_ref();
         return servant;
      }

      virtual void etherealize (
               const PortableServer::ObjectId& oid,
            PortableServer::POA_ptr adapter,
            PortableServer::Servant servant,
            CORBA::Boolean cleanup_in_progress,
            CORBA::Boolean remaining_activations) {
      // If there are no remaining activations i.e ObjectIds
      // associated with the servant delete it.

      CORBA::String_var s =
            PortableServer::ObjectId_to_string (oid);
      cout << "¥nAccountManagerActivator.etherealize called
            with ID = " << s << endl;
      if (!remaining_activations)
         delete servant;
   }
};
```

コードサンプル 7-12　オブジェクトの非活性化例（Java）

```
import org.omg.PortableServer.*;
public class AccountManagerActivator extends
                                       ServantActivatorPOA {
   public Servant incarnate (
            byte[ ] oid, POA adapter) throws ForwardRequest {
      Servant servant;
      String accountType = new String(oid);
      System.out.println(
        "¥nAccountManagerActivator. incarnate
         called with ID = " + accountType + "¥n");
      // Create Savings or Checking Servant based on
      // AccountType
      if (accountType.equalsIgnoreCase(
                                    "SavingsAccountManager"))
         servant = (Servant )new SavingsAccountManagerImpl();
      else
         servant = (Servant)new CheckingAccountManagerImpl();
         new DeactivateThread(oid, adapter).start();
         return servant;
      }
      public void etherealize (byte[ ] oid,
            POA adapter,
            Servant serv,
            boolean cleanup_in_progress,
            boolean remaining_activations) {
         System.out.println(
```

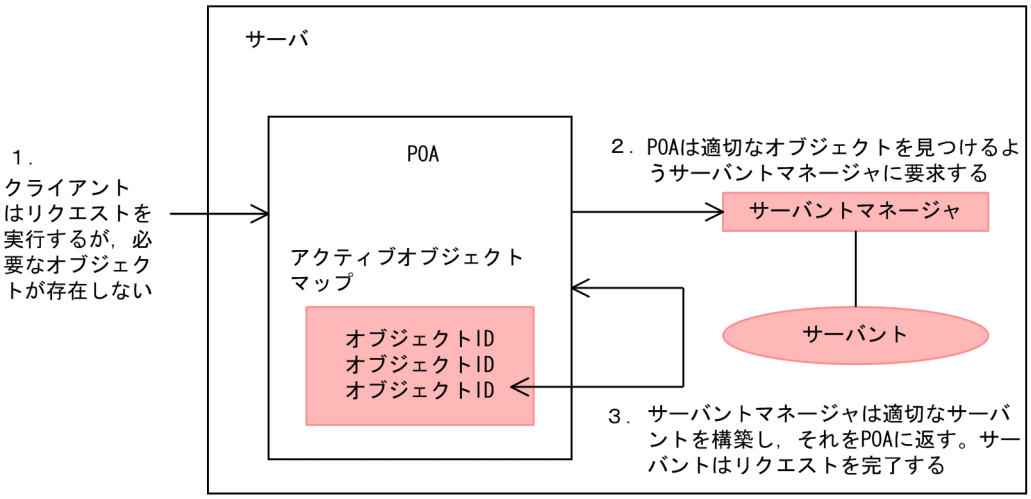
```
            "¥nAccountManagerActivator.etherealize
             called with ID = " + new String(oid) + "¥n");
          System.gc();
       }
}
class DeactivateThread extends Thread {
   byte[ ] _oid;
   POA _adapter;
   public DeactivateThread(byte[ ] oid, POA adapter) {
      _oid = oid;
      _adapter = adapter;
   }
   public void run() {
      try {
         Thread.currentThread().sleep(15000);
         System.out.println(
                    ¥nDeactivating the object with ID = " +
                    new String(_oid) + "¥n");
         _adapter.deactivate_object(_oid);
   } catch (Exception e) {
     e.printStackTrace();
   }
}
}
```

7.5　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，サーバントを見つけて返す，サーバントを非活性化する，という 2 種類のオペレーションを行います。サーバントマネージャは，非アクティブなオブジェクトに対するリクエストが受信されたときに POA がオブジェクトを活性化できるようにします。サーバントマネージャの使用は任意です。例えば，起動時にサーバがすべてのオブジェクトをロードする場合は，サーバントマネージャは不要です。サーバントマネージャは，ForwardRequest 例外を使用してほかのオブジェクトにリクエストを転送するようにクライアントに知らせることもできます。

サーバントは，インプリメンテーション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サーバントのオブジェクト ID のマップを維持します。クライアントリクエストを受信すると，POA はまずマップをチェックして（クライアントリクエストに埋め込まれている）オブジェクト ID が記録されているかどうかを調べます。このオブジェクト ID が見つかれば，POA はリクエストをサーバントに転送します。オブジェクト ID がマップになければ，サーバントマネージャは適切なサーバントの探索と活性化を要求されます。これは，あくまでもシナリオ例です。正確なシナリオは実際に使用する POA ポリシーによって異なります。サーバントマネージャ機能の例を図 7-2 に示します。

図 7-2　サーバントマネージャ機能の例



サーバントマネージャには ServantActivator と ServantLocator という二つのタイプがあります。どのコールバックを使用するかは，設定済みのポリシーのタイプによって決定します。POA ポリシーの詳細については，「7.2　POA ポリシー」を参照してください。一般には，ServantActivator がパーシステントオブジェクトを活性化し，ServantLocator がトランジェントオブジェクトを活性化します。

C++の場合

サーバントマネージャを使用するには，RequestProcessingPolicy::USE_SERVANT_MANAGER と，サーバントマネージャのタイプを定義するポリシー（ServantActivator の場合は ServantRetentionPolicy::RETAIN，ServantLocator の場合は ServantRetentionPolicy::NON_RETAIN）を指定しなければなりません。

Java の場合

サーバントマネージャを使用するには，RequestProcessingPolicy.USE_SERVANT_MANAGER と，サーバントマネージャのタイプを定義するポリシー（ServantActivator の場合は ServantRetentionPolicy.RETAIN，ServantLocator の場合は ServantRetentionPolicy.NON_RETAIN）を指定しなければなりません。

7.5.1　ServantActivator

C++の場合

ServantActivator は，ServantRetentionPolicy::RETAIN と RequestProcessingPolicy::USE_SERVANT_MANAGER が設定された場合に使用します。

Java の場合

ServantActivator は，ServantRetentionPolicy.RETAIN と RequestProcessingPolicy.USE_SERVANT_MANAGER が設定された場合に使用します。

このタイプのサーバントマネージャによって活性化されたサーバントは，アクティブオブジェクトマップで管理されます。

サーバントアクティベータを使用したリクエストの処理中には，次のようなイベントが発生します。

1. クライアントリクエストを受信します（クライアントリクエストには POA 名，オブジェクト ID などの情報が含まれます）。
2. POA はまず，アクティブオブジェクトマップをチェックします。オブジェクト ID がそこで見つかれば，オペレーションは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 が見つからなければ，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 と，オブジェクトを活性化している POA を渡します。
4. サーバントマネージャは適切なサーバントを探します。
5. サーバント ID がアクティブオブジェクトマップに入力され，クライアントに応答が返されます。

注

etherealize および incarnate メソッドインプリメンテーションはユーザが指定するコードです。

そのあとで，サーバントを非活性化できます。これは，deactivate_object オペレーション，該当する POA と対応する POA マネージャの非活性化など，幾つかのソースから行えます。オブジェクトの非活性化の詳細については，「7.4.5　オブジェクトの非活性化」を参照してください。

コードサンプル 7-13　サーバントアクティベータタイプのサーバントマネージャを示すサーバコードサンプル（C++）

```
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

      DataStore::_create();

      // get a reference to the rootPOA
      CORBA::Object_var obj =
            orb->resolve_initial_references("RootPOA");
      PortableServer::POA_var rootPOA =
            PortableServer::POA::_narrow(obj);

      CORBA::PolicyList policies;
      policies.length(2);

      policies[(CORBA::ULong)0] =
            rootPOA->create_lifespan_policy
                  (PortableServer::PERSISTENT);
      policies[(CORBA::ULong)1] =
            rootPOA->create_request_processing_policy(
                  PortableServer::USE_SERVANT_MANAGER);

      // Create myPOA with the right policies
```

```
        PortableServer::POAManager_var rootManager =
              rootPOA->the_POAManager();
        PortableServer::POA_var myPOA =
              rootPOA->create_POA("bank_servant_activator_poa",
              rootManager, policies);
        // Create a Servant activator
        AccountManagerActivator servant_activator_impl;

        //Set the servant activator
        myPOA->set_servant_manager(&servant_activator_impl);

        // Generate two references - one for checking and another
        //for savings.Note that we are not creating any
        //servants here and just manufacturing a reference which
        //is not yet backed by a servant
        PortableServer::ObjectId_var an_oid =
              PortableServer::string_to_ObjectId
                    ("CheckingAccountManager");
        CORBA::Object_var cref = myPOA->create_reference_with_id
              (an_oid.in(), IDL:Bank/AccountManager:1.0");

        an_oid = PortableServer::string_to_ObjectId
              ("SavingsAccountManager");
        CORBA::Object_var sref = myPOA->create_reference_with_id
              (an_oid.in(), "IDL:Bank/AccountManager:1.0");

        //Activate the POA Manager
        rootManager->activate();

        // Write out Checking reference
        CORBA::String_var string_ref = orb->object_to_string
              (cref.in());
        ofstream crefFile("cref.dat");
        crefFile << string_ref << endl;
        crefFile.close();
        // Now write out the Savings reference
        string_ref = orb->object_to_string(sref.in());
        ofstream srefFile("sref.dat");
        srefFile << string_ref << endl;
        srefFile.close();

        // Waiting for incoming requests
        cout << "BankManager Server is ready" << endl;
        orb->run();

        DataStore::_destroy();
     }
     catch(const CORBA::Exception& e) {
        cerr << e << endl;
     }
     return 1;
  }
```

コードサンプル 7-14　サーバントアクティベータタイプのサーバントマネージャを示すサーバコードサンプル（Java）

```
  import org.omg.PortableServer.*;
  public class Server {
     public static void main(String[ ] args) {
        try {
           // Initialize the ORB.
           org.omg.CORBA.ORB orb =
                        org.omg.CORBA.FORB.init(args, null);
           // get a reference to the rootPOA
           POA rootPOA = POAHelper.narrow(
             orb.resolve_initial_references("RootPOA"));
           // Create policies for our POA.
           // We need persistence life span and
           // use servant manager request processing policies
           org.omg.CORBA.Policy[ ] policies = {
              rootPOA.create_lifespan_policy(
                 LifespanPolicyValue.PERSISTENT),
              rootPOA.create_request_processing_policy(
                 RequestProcessingPolicyValue.
```

```
                  USE_SERVANT_MANAGER)
               };

               // Create myPOA with the right policies
               POA myPOA = rootPOA.create_POA(
                     "bank_servant_activator_poa",
                     rootPOA.the_POAManager(),
                     policies );
               // Create the servant activator servant
               //  and get its reference
               ServantActivator sa =
                     new AccountManagerActivator()._this(orb);
               // Set the servant activator on our POA
               myPOA.set_servant_manager(sa);
               org.omg.CORBA.Object ref;
               // Activate the POA manager
               rootPOA.the_POAManager().activate();
               // Generate the reference and write it out.
               // One for each Checking and Savings
               // account types .Note that we are not creating
               // any servants here and just manufacturing a
               // reference which is not yet backed by a servant.
               try {
                 ref = myPOA.create_reference_with_id(
                      "CheckingAccountManager".getBytes(),
                      "IDL:Bank/AccountManager:1.0");
                 // Write out checking object ID
                 java.io.PrintWriter pw =
                 new java.io.PrintWriter(
                           new java.io.FileWriter("cref.dat"));
                 pw.println(orb.object_to_string(ref));
                 pw.close();
                 ref = myPOA.create_reference_with_id(
                       "SavingsAccountManager".getBytes(),
                       "IDL:Bank/AccountManager:1.0");
                 // Write out savings object ID
                 pw = new java.io.PrintWriter(
                      new java.io.FileWriter("sref.dat"));
                 System.gc();
                 pw.println(orb.object_to_string(ref));
                 pw.close();
               } catch ( java.io.IOException e ) {
                  System.out.println(
                              "Error writing the IOR to file ");
                  return;
               }
               System.out.println("Bank Manager is ready.");
               // Wait for incoming requests
               orb.run();
          } catch (Exception e) {
             e.printStackTrace();
          }
       }
    }
```

このサンプルのサーバントマネージャを，次に示します。

コードサンプル 7-15　サーバントアクティベータのサンプルのサーバントマネージャ（C++）

```
// Servant Activator
class AccountManagerActivator :
      public PortableServer::ServantActivator {
   public:
      virtual PortableServer::Servant incarnate
            (const PortableServer::ObjectId& oid,
            PortableServer::POA_ptr poa) {
         CORBA::String_var s =
               PortableServer::ObjectId_to_string (oid);
         cout << "¥nAccountManagerActivator.incarnate called
               with ID = " << s << endl;
         PortableServer::Servant servant;

      if ( VISPortable::vstricmp( (char *)s,
            "SavingsAccountManager" ) == 0 )
```

```
         //Create CheckingAccountManager Servant
         servant = new SavingsAccountManagerImpl;
      else if ( VISPortable::vstricmp( (char *)s,
            "CheckingAccountManager" ) == 0 )
         //Create CheckingAccountManager Servant
         servant = new CheckingAccountManagerImpl;
      else
         throw CORBA::OBJECT_NOT_EXIST();
      // Create a deactivator thread
      new DeActivatorThread( oid, poa );
      // return the servant
      return servant;
   }

   virtual void etherealize (
         const PortableServer::ObjectId& oid,
         PortableServer::POA_ptr adapter,
         PortableServer::Servant servant,
         CORBA::Boolean cleanup_in_progress,
         CORBA::Boolean remaining_activations) {
      // If there are no remaining activations i.e ObjectIds
      // associated with the servant delete it.
      CORBA::String_var s =
            PortableServer::ObjectId_to_string (oid);
      cout << "¥nAccountManagerActivator.etherealize called
         with ID = " << s << endl;
      if (!remaining_activations)
         delete servant;
   }
};
```

コードサンプル 7-16　サーバントアクティベータのサンプルのサーバントマネージャ（Java）

```
import org.omg.PortableServer.*;
public class AccountManagerActivator extends
                                      ServantActivatorPOA {
   public Servant incarnate (
       byte[ ] oid, POA adapter) throws ForwardRequest {
         Servant servant;
         String accountType = new String(oid);
         System.out.println(
           "¥nAccountManagerActivator.incarnate
             called with ID = " + accountType + "¥n");
      // Create Savings or Checking Servant based on
      // AccountType
      if ( accountType.equalsIgnoreCase(
                                    "SavingsAccountManager"))
         servant = (Servant )new SavingsAccountManagerImpl();
      else
         servant = (Servant)new CheckingAccountManagerImpl();
      new DeactivateThread(oid, adapter).start();
      return servant;
   }
   public void etherealize (byte[ ] oid,
         POA adapter,
         Servant serv,
         boolean cleanup_in_progress,
         boolean remaining_activations){
     System.out.println(
        "¥nAccountManagerActivator.
         etherealize called with ID = " +
         new String(oid) + "¥n");
     System.gc();
   }
}
class DeactivateThread extends Thread {
   byte[ ] _oid;
   POA _adapter;
   public DeactivateThread(byte[ ] oid, POA adapter) {
     _oid = oid;
     _adapter = adapter;
   }

   public void run(){
     try {
```

```
            Thread.currentThread().sleep(15000);
            System.out.println(
              "¥nDeactivating the object with ID = " +
               new String(_oid) + "¥n");
            _adapter.deactivate_object(_oid);
          }catch (Exception e) {
            e.printStackTrace();
          }
        }
    }
```

7.5.2　ServantLocator

POA のアクティブオブジェクトマップがきわめて大きくなりメモリを消費するような状況はよくあります。メモリ消費量を減らすために，サーバントとオブジェクトの対応をアクティブオブジェクトマップに格納しないという意味の RequestProcessingPolicy::USE_SERVANT_MANAGER と ServantRetentionPolicy::NON_RETAIN（C++）設定，または RequestProcessingPolicy.USE_SERVANT_MANAGER と ServantRetentionPolicy.NON_RETAIN（Java）設定で POA を作成できます。対応が格納されないので，ServantLocator サーバントマネージャをリクエストごとに起動できます。

サーバントロケータを使用したリクエストの処理中には，次のようなイベントが発生します。

1. クライアントリクエストを受信します（クライアントリクエストには POA 名およびオブジェクト ID が含まれます）。
2. ServantRetentionPolicy::NON_RETAIN（C++），または ServantRetentionPolicy.NON_RETAIN（Java）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のオブジェクト ID を探しません。
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，オブジェクトを活性化している POA などのパラメタを渡します。
4. サーバントロケータは適切なサーバントを探します。
5. サーバントに対するオペレーションが実行され，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。

注

preinvoke メソッドと postinvoke メソッドはユーザが指定するコードです。

コードサンプル 7-17　サーバントロケータタイプのサーバントマネージャを示すサーバコードサンプル (C++)

```
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

      // And the Data source
      DataStore::_create();

      // get a reference to the rootPOA
      CORBA::Object_var obj =
            orb->resolve_initial_references("RootPOA");
      PortableServer::POA_var rootPOA =
            PortableServer::POA::_narrow(obj);

      CORBA::PolicyList policies;
      policies.length(3);

      // Create a child POA with Persistence life span policy
      // that uses servant manager with non-retain retention
      // policy ( no Active Object Map ) causing the POA to use
      // the servant locator.
      policies[(CORBA::ULong)0] =
```

```
            rootPOA->create_lifespan_policy
            (PortableServer::PERSISTENT);
        policies[(CORBA::ULong)1] =
            rootPOA->create_servant_retention_policy
            (PortableServer::NON_RETAIN);
        policies[(CORBA::ULong)2] =
            rootPOA->create_request_processing_policy
            (PortableServer::USE_SERVANT_MANAGER);
        PortableServer::POAManager_var rootManager =
            rootPOA->the_POAManager();
        PortableServer::POA_var myPOA =
            rootPOA->create_POA("bank_servant_locator_poa",
            rootManager, policies);
        // Create the servant locator
        AccountManagerLocator servant_locator_impl;
        myPOA->set_servant_manager(&servant_locator_impl);

        // Generate two references - one for checking and another
        // for savings.Note that we are not creating any
        // servants here and just manufacturing a reference which
        // is not yet backed by a servant
        PortableServer::ObjectId_var an_oid =
            PortableServer::string_to_ObjectId
            ("CheckingAccountManager");
        CORBA::Object_var cref = myPOA->create_reference_with_id
            (an_oid.in(), "IDL:Bank/AccountManager:1.0");

        an_oid = PortableServer::string_to_ObjectId
            ("SavingsAccountManager");
        CORBA::Object_var sref = myPOA->create_reference_with_id
            (an_oid.in(), "IDL:Bank/AccountManager:1.0");

        // Activate the POA Manager
        rootManager->activate();

        // Write out Checking reference
        CORBA::String_var string_ref =
            orb->object_to_string(cref.in());
        ofstream crefFile("cref.dat");
        crefFile << string_ref << endl;
        crefFile.close();
        // Now write out the Savings reference
        string_ref = orb->object_to_string(sref.in());
        ofstream srefFile("sref.dat");
        srefFile << string_ref << endl;
        srefFile.close();

        // Wait for incoming requests
        cout << "Bank Manager is ready" << endl;
        orb->run();
        // Destroy the accounts database
        DataStore::_destroy();
    }
    catch(const CORBA::Exception& e) {
        cerr << e << endl;
    }
    return 1;
}
```

コードサンプル 7-18　サーバントロケータタイプのサーバントマネージャを示すサーバコードサンプル (Java)

```
import org.omg.PortableServer.*;
   public class Server {
      public static void main(String[ ] args) {
         try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb =
                        org.omg.CORBA.ORB.init(args,null);
            // get a reference to the rootPOA
            POA rootPOA = POAHelper.narrow(
               orb.resolve_initial_references("RootPOA"));
            // Create policies for our POA.
            // We need persistence life span,
```

```
                // use servant manager request processing
                // policies and non retain retention policy.
                // This non retain policy will let us use the

                // servant locator instead of servant activator
                org.omg.CORBA.Policy[ ] policies = {
                   rootPOA.create_lifespan_policy(
                        LifespanPolicyValue.PERSISTENT),
                   rootPOA.create_servant_retention_policy(
                        ServantRetentionPolicyValue.NON_RETAIN),
                   rootPOA.create_request_processing_policy(
                        RequestProcessingPolicyValue.
                        USE_SERVANT_MANAGER)
                };
                // Create myPOA with the right policies
                POA myPOA = rootPOA.create_POA(
                      "bank_servant_locator_poa",
                      rootPOA.the_POAManager(),
                      policies );
                // Create the servant locator servant
                // and get its reference
                ServantLocator sl =
                         new AccountManagerLocator()._this(orb);
                // Set the servant locator on our POA
                myPOA.set_servant_manager(sl);
                org.omg.CORBA.Object ref ;
                // Activate the POA manager
                rootPOA.the_POAManager().activate();
                // Generate the reference and write it out.
                // One for each Checking and Savings
                // account types .Note that we are not creating
                // any servants here and just manufacturing a
                // reference which is not yet backed by a servant.
                try {
                  ref = myPOA.create_reference_with_id(
                          "CheckingAccountManager".getBytes(),
                          "IDL:Bank/AccountManager:1.0");
                  // Write out checking object ID
                  java.io.PrintWriter pw =
                  new java.io.PrintWriter(
                            new java.io.FileWriter("cref.dat"));
                  pw.println(orb.object_to_string(ref));
                  pw.close();
                  ref = myPOA.create_reference_with_id(
                          "SavingsAccountManager".getBytes(),
                          "IDL:Bank/AccountManager:1.0");
                  // Write out savings object ID
                  pw = new java.io.PrintWriter(
                      new java.io.FileWriter("sref.dat"));
                  System.gc();
                  pw.println(orb.object_to_string(ref));
                  pw.close();
                } catch ( java.io.IOException e ){
                   System.out.println(
                               "Error writing the IOR to file");
                   return;
                }
                System.out.println("BankManager is ready.");
                // Wait for incoming requests
                orb.run();
             } catch (Exception e) {
                e.printStackTrace();
             }
          }
       }
```

このサンプルのサーバントマネージャは，次のとおりです。

コードサンプル 7-19　サーバントロケータのサンプルのサーバントマネージャ（C++）

```
   // Servant Locator
   class AccountManagerLocator :
         public PortableServer::ServantLocator {
      public:
```

```
      AccountManagerLocator (){}

      // preinvoke is very similar to ServantActivator 's
      // incarnate method but gets called every time a
      // request comes in unlike incarnate() which gets called
      // every time the POA does not find a servant in the
      // active object map
      virtual PortableServer::Servant preinvoke
            (const PortableServer::ObjectId& oid,
            PortableServer::POA_ptr adapter,
            const char* operation,
            PortableServer::ServantLocator::
                  Cookie& the_cookie) {
         CORBA::String_var s =
               PortableServer::ObjectId_to_string (oid);
         cout << "¥nAccountManagerLocator.preinvoke called
               with ID = " << s << endl;
         PortableServer::Servant servant;

         if ( VISPortable::vstricmp( (char *)s,
               "SavingsAccountManager" ) == 0 )
            //Create CheckingAccountManager Servant
            servant = new SavingsAccountManagerImpl;
         else if ( VISPortable::vstricmp( (char *)s,
               "CheckingAccountManager" ) == 0 )
         // Create CheckingAccountManager Servant
            servant = new CheckingAccountManagerImpl;
         else
            throw CORBA::OBJECT_NOT_EXIST();

         // Note also that we do not spawn of a thread to
         // explicitly deactivate an object unlike a servant
         // activator , this is because the POA itself calls
         // post invoke after the request is complete.In the
         // case of a servant activator the POA calls
         // etherealize() only if the object is deactivated
         // by calling poa->de_activate object or the POA
         // itself is destroyed.

         //return the servant
         return servant;
      }

   virtual void postinvoke (const PortableServer::ObjectId& oid,
         PortableServer::POA_ptr adapter,
         const char* operation,
         PortableServer::ServantLocator::Cookie the_cookie,
         PortableServer::Servant the_servant) {
      CORBA::String_var s =
            PortableServer::ObjectId_to_string (oid);
      cout << "¥nAccountManagerLocator.postinvoke called
            with ID = " << s << endl;
      delete the_servant;
   }
};
```

コードサンプル 7-20　サーバントロケータのサンプルのサーバントマネージャ（Java)

```
import org.omg.PortableServer.*;
import org.omg.PortableServer.
                        ServantLocatorPackage.CookieHolder;
public class AccountManagerLocator extends
                                    ServantLocatorPOA {
   public Servant preinvoke (byte[ ] oid,POA adapter,
         java.lang.String operation,
         CookieHolder the_cookie) throws ForwardRequest {
      String accountType = new String(oid);
      System.out.println(
            "¥nAccountManagerLocator.
             preinvoke called with ID = " +
            accountType + "¥n");
      if ( accountType.equalsIgnoreCase
                              ("SavingsAccountManager"))
         return new SavingsAccountManagerImpl();
      return new CheckingAccountManagerImpl();
```

```
    }
    public void postinvoke (byte[ ] oid,
          POA adapter,
          java.lang.String operation,
          java.lang.Object the_cookie,
          Servant the_servant) {
       System.out.println(
            "¥nAccountManagerLocator.postinvoke called
             with ID = " +
            new String(oid) + "¥n");
    }
}
```

7.6　POA マネージャによる POA 管理

POA マネージャは，POA の状態（リクエストを待ち行列に入れるか破棄するか）を管理し，POA を非活性化できます。各 POA は POA マネージャオブジェクトに対応します。POA マネージャは一つ以上の POA を制御できます。

POA マネージャは POA が生成されたときにその POA に対応づけられます。使用する POA マネージャを指定することも，null を指定して新しい POA マネージャを生成することもできます。

コードサンプル 7-21　POA およびその POA マネージャに名前を付ける（C++）

```
PortableServer::POAManager_var rootManager =
      rootPOA->the_POAManager();
PortableServer::POA_var myPOA =
      rootPOA->create_POA(
            "bank_servant_locator_poa", rootManager, policies);

PortableServer::POA_var myPOA = rootPOA->create_POA(
      "bank_servant_locator_poa",
      null,
      policies );
```

コードサンプル 7-22　POA およびその POA マネージャに名前を付ける（Java）

```
POA myPOA = rootPOA.create_POA( "bank_agent_poa",
      rootPOA.the_POAManager(),
      policies );
POA myPOA = rootPOA.create_POA( "bank_agent_poa",
      null,
      policies );
```

POA マネージャは，対応する POA がすべてデストラクトされると，自身もデストラクトされます。

POA マネージャには次に示す四つの状態があります。

- 待機
- アクティブ
- 破棄
- 非アクティブ

これらの状態は，さらに POA の状態を決定します。これらの状態について詳しく説明します。

7.6.1　カレントの状態の取得

POA マネージャのカレントの状態を取得するには，次のように使用します。

```
enum State{HOLDING, ACTIVE, DISCARDING, INACTIVE};
State get_state();
```

7.6.2　待機状態

デフォルトでは，POA マネージャは待機状態で生成されます。POA マネージャが待機状態のときは，POA はすべての入力リクエストを待ち行列に入れます。

POA マネージャが待機状態のときは，アダプタアクティベータを必要とするリクエストも待ち行列に入れられます。

POA マネージャの状態を待機状態に変えるには，次のように使用します。

```
void hold_requests(in boolean wait_for_completion)
   raises (AdapterInactive);
```

wait_for_completion は Boolean です。FALSE なら，このオペレーションは状態を待機状態に変更後すぐにリターンします。TRUE なら，このオペレーションは，状態変更より前に開始されたリクエストがすべて完了した場合か，POA マネージャが待機以外の状態に変更された場合だけリターンします。AdapterInactive は，このオペレーションを呼び出す前に POA マネージャが非アクティブな状態だった場合に発生する例外です。

注

現在非アクティブな状態の POA マネージャは，待機状態に変更できません。

待ち行列に入れられたが開始されていないリクエストは，待機状態中は引き続き待ち行列に入れられます。

7.6.3　アクティブな状態

POA マネージャがアクティブな状態のときは，対応する POA はリクエストを処理します。

POA マネージャをアクティブな状態に変更するには，次のように使用します。

```
void activate()
   raises (AdapterInactive);
```

AdapterInactive は，このオペレーションを呼び出す前に POA マネージャが非アクティブな状態だった場合に発生する例外です。

注

現在非アクティブな状態の POA マネージャは，アクティブな状態に変更できません。

7.6.4　破棄状態

POA マネージャが破棄状態のときは，対応する POA は，開始していないリクエストをすべて破棄します。さらに，対応する POA に登録されたアダプタアクティベータは呼び出されません。この状態は，POA が受信するリクエストが多過ぎる場合に便利です。プログラマは，リクエストが破棄されたことと，リクエストを再送するようにクライアントに通知する必要があります。POA が受信するリクエストが多過ぎる理由とその時期を決定する固有の動作はありません。決定は，設定済みのスレッド監視に左右されます。

POA マネージャを破棄状態に変更するには，次のように使用します。

```
void discard_requests(in boolean wait_for_completion)
   raises (AdapterInactive);
```

wait_for_completion オプションは Boolean です。FALSE なら，このオペレーションは状態を破棄状態に変更後すぐにリターンします。TRUE なら，このオペレーションは，状態変更より前に開始されたリクエストがすべて完了した場合か，POA マネージャが破棄以外の状態に変更された場合だけリターンします。AdapterInactive は，このオペレーションを呼び出す前に POA マネージャが非アクティブな状態だった場合に発生する例外です。

注

現在非アクティブな状態の POA マネージャは，破棄状態に変更できません。

7.6.5　非アクティブな状態

POA マネージャが非アクティブな状態のときは，対応する POA は入力リクエストをすべて拒否します。この状態は，対応する POA を終了しようとするときに使用します。

POA の終了後にリクエストを送信すると，クライアント側で PortableServer::POA::AdapterNonExistent 例外または CORBA::OBJECT_NOT_EXIST 例外が発生する場合があります。

注

非アクティブな状態の POA マネージャは，ほかの状態には変更できません。

POA マネージャを非アクティブな状態に変更するには，次のように使用します。

```
void deactivate(in boolean etherealize_objects, in boolean
   wait_for_completion) raises (AdapterInactive);
```

C++の場合

状態変更後，etherealize_objects が TRUE なら，ServantRetentionPolicy::RETAIN と RequestProcessingPolicy::USE_SERVANT_MANAGER を設定した対応するすべての POA は，すべてのアクティブな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。etherealize_objects が FALSE なら，etherealize は呼び出されません。wait_for_completion オプションは Boolean です。FALSE なら，このオペレーションは状態を非アクティブな状態に変更後すぐにリターンします。TRUE なら，このオペレーションは，状態変更より前に開始されたリクエストがすべて完了した場合か，etherealize が対応するすべての POA（ServantRetentionPolicy::RETAIN と Request ProcessingPolicy::USE_SERVANT_MANAGER を設定した POA）で呼び出された場合だけリターンします。AdapterInactive は，このオペレーションを呼び出す前に POA マネージャが非アクティブな状態だった場合に発生する例外です。

Java の場合

状態変更後，etherealize_objects が TRUE なら，ServantRetentionPolicy.RETAIN と RequestProcessingPolicy.USE_SERVANT_MANAGER を設定した対応するすべての POA は，すべてのアクティブな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。etherealize_objects が FALSE なら，etherealize は呼び出されません。wait_for_completion オプションは Boolean です。FALSE なら，このオペレーションは状態を非アクティブな状態に変更後すぐにリターンします。TRUE なら，このオペレーションは，状態変更より前に開始されたリクエストがすべて完了した場合か，etherealize が対応するすべての POA（ServantRetentionPolicy.RETAIN と Request ProcessingPolicy.USE_SERVANT_MANAGER を設定した POA）で呼び出された場合だけリターンします。AdapterInactive は，このオペレーションを呼び出す前に POA マネージャが非アクティブな状態だった場合に発生する例外です。

7.7　監視プロパティとディスパッチプロパティの設定

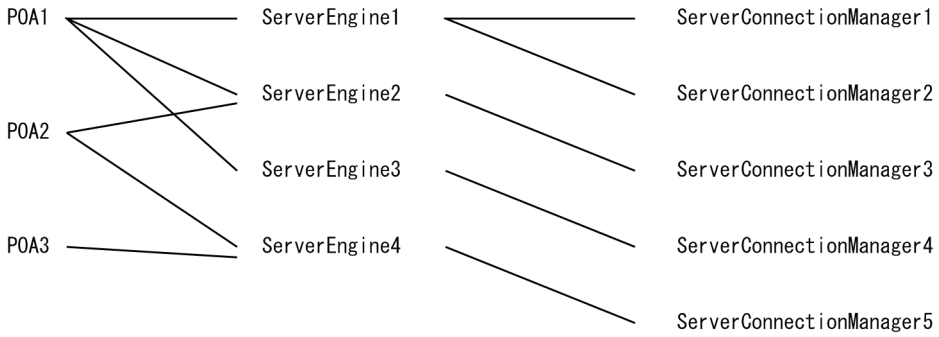
POA は，以前 BOA がサポートしていたリスナー機能とディスパッチャ機能をカバーするポリシーを持っていません。これらの機能を提供するために，VisiBroker 固有ポリシー（ServerEnginePolicy）を使用できます。

サーバエンジンの内容は，次のとおりです。

- ホスト名
- プロキシホスト名
- サーバコネクションマネージャまたはサーバコネクションマネージャのリスト

これらが互いにどのように適合するかを図 7-3 に示します。

図 7-3　サーバエンジンの概要



最も単純なケースでは，各 POA が独自のサーバエンジンを一つずつ持ちます。この場合，別々の POA に対するリクエストは別々のポートに到着します。一つの POA に複数のサーバエンジンを持たせることもできます。このシナリオでは，一つの POA は複数の入力ポートからのリクエストをサポートします。

POA 間でサーバエンジンを共用できることに注意してください。サーバエンジンを共用する場合は，各 POA は同じポートを監視します。（複数の）POA に対するリクエストが同じポートに到着しても，リクエストに埋め込まれた POA 名のおかげで，これらのリクエストは正確にディスパッチされます。このシナリオは，デフォルトサーバエンジンを使用して（POA 生成時に新しいサーバエンジンを指定しないで）複数の POA を生成するような場合に起こり得ます。

7.7.1　サーバエンジンプロパティの設定

次に示すプロパティは，デフォルトで使用するサーバエンジンを決定します。

```
vbroker.se.<server_engine_name>.host
vbroker.se.<server_engine_name>.proxyHost
vbroker.se.<server_engine_name>.scms
```

サーバエンジンポリシーを指定しないと，POA はサーバエンジン名として，iiop_tp を仮定し，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
```

デフォルトのサーバ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その名前を入力し，新しいサーバエンジンのすべてのコンポーネントの値を定義します。次にサンプルを示します。

```
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
```

7.7.2　サーバコネクションマネージャプロパティの設定

サーバコネクションマネージャは，マネージャ，リスナー，ディスパッチャという三つのプロパティグループで構成されます。

(1)　マネージャプロパティ

次に示すマネージャプロパティを設定できます。

vbroker.se.<server_engine>.scm.<server_connection_mgr>.manager.type

コネクションマネージャタイプを識別します。C++でサポートされているタイプは Socket および Local です。Java では，Socket だけサポートされています。

vbroker.se.<server_engine>.scm.<server_connection_mgr>.manager.connectionMax

同時に確立される入力コネクションの最大許容数を定義します。デフォルト値は 0 で，これはコネクション数を制限しないという意味です。

vbroker.se.<server_engine>.scm.<server_connection_mgr>.manager.connectionMaxIdle

コネクションがシャットダウンされるまでのアイドル状態の最大秒数を定義します。デフォルト値は 0 で，これはタイムアウトがないという意味です。

最大アイドル時間の監視は，プロパティ vbroker.orb.gcTimeout の設定値（デフォルト 30 秒）の範囲の誤差があります。

(2)　リスナープロパティ

次に示すリスナープロパティを設定できます。

vbroker.se.<server_engine>.scm.<server_connection_mgr>.listener.type

リスナータイプを識別します。サポートされているのは IIOP だけです。

vbroker.se.<server_engine>.scm.<server_connection_mgr>.listener.port

コネクションマネージャが使用するこのサーバと対応する POA の監視ポートを定義します。デフォルト値は 0 で，これはシステム（OS）によって自動的に割り当てられたポート番号を使用するという意味です。

vbroker.se.<server_engine>.scm.<server_connection_mgr>.listener.proxyPort

プロキシホスト名プロパティとともに使用するプロキシポート番号を指定します。デフォルト値は 0 で，これはシステム（OS）によって自動的に割り当てられたポート番号を使用するという意味です。

(3)　ディスパッチャプロパティ

次に示すディスパッチャプロパティを設定できます。

vbroker.se.<server_engine>.scm.<server_connection_mgr>.dispatcher.type

ディスパッチャタイプを識別します。現在サポートされているタイプは ThreadPool と ThreadSession だけです。

vbroker.se.<server_engine>.scm.<server_connection_mgr>.dispatcher.threadMax

タイプが ThreadPool に設定されている場合だけ使用します。

vbroker.se.<server_engine>.scm.<server_connection_mgr>.dispatcher.threadMaxIdle

タイプが ThreadPool に設定されている場合だけ使用します。

vbroker.se.<server_engine>.scm.<server_connection_mgr>.dispatcher.threadMin

タイプが ThreadPool に設定されている場合だけ使用します。

7.7.3　これらのプロパティはいつ使用するか

幾つかのサーバエンジンプロパティの変更が必要な場合はよくあります。これらのプロパティを変更する方法は必要事項によって異なります。例えば，ポート番号を変更したい場合は，次のようにします。

- デフォルトの listener.port プロパティを変更
- 新しいサーバエンジンを生成

デフォルトの listener.port プロパティを変更することは最も単純な方法ですが，デフォルトのサーバエンジンを使用するすべての POA に影響を与えます。

特定の POA のポート番号を変更したい場合は，新しいサーバエンジンを生成し，この新サーバエンジンのプロパティを定義し，さらに POA の生成時に新サーバエンジンを参照する必要があります。「7.7.1　サーバエンジンプロパティの設定」ではサーバエンジンプロパティの更新方法を説明しました。次に示すコードの一部では，サーバエンジンのプロパティの定義方法とユーザ定義のサーバエンジンポリシーによる POA の生成方法を示しています。

コードサンプル 7-23　特定サーバエンジンによる POA の生成（C++）

```
// static initialization
AccountRegistry AccountManagerImpl::_accounts;
int main(int argc, char* const* argv)
{
   try {
   // Initialize the orb
   CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
   // Get the property manager; notice the value returned
   // is not placed into a 'var' type.
   VISPropertyManager_ptr pm = orb->getPropertyManager();
   pm->addProperty("vbroker.se.mySe.host", "");
   pm->addProperty("vbroker.se.mySe.proxyHost", "");
   pm->addProperty("vbroker.se.mySe.scms", "scmlist");
   pm->addProperty("vbroker.se.mySe.scm.scmlist.manager
       .type", "Socket");
   pm->addProperty("vbroker.se.mySe.scm.scmlist.manager
       .connectionMax", 100UL);
   pm->addProperty("vbroker.se.mySe.scm.scmlist.manager
       .connectionMaxIdle", 300UL);
   pm->addProperty("vbroker.se.mySe.scm.scmlist.listener
       .type", "IIOP");
   pm->addProperty("vbroker.se.mySe.scm.scmlist.listener
       .port", 55000UL);
   pm->addProperty("vbroker.se.mySe.scm.scmlist.listener
       .proxyPort", 0UL);
   pm->addProperty("vbroker.se.mySe.scm.scmlist.dispatcher
       .type", "ThreadPool");
   pm->addProperty("vbroker.se.mySe.scm.scmlist.dispatcher
       .threadMax", 100UL);
   pm->addProperty("vbroker.se.mySe.scm.scmlist.dispatcher
       .threadMin", 5UL);
   pm->addProperty("vbroker.se.mySe.scm.scmlist.dispatcher
       .threadMaxIdle", 300UL);
   // Get a reference to the rootPOA
   CORBA::Object_var obj =
       orb->resolve_initial_references("RootPOA");
```

```
   PortableServer::POA_var rootPOA =
       PortableServer::POA::_narrow(obj);
   // Create the policies
   CORBA::Any_var seAny(new CORBA::Any);
   // The SERVER_ENGINE_POLICY_TYPE requires a sequence,
   // even if only one engine is being specified.
   CORBA::StringSequence_var engines =
      new CORBA::StringSequence(1UL);
   engines->length(1UL);
   engines[0UL] = CORBA::string_dup("mySe");
   seAny <<= engines;
   CORBA::PolicyList_var policies = new CORBA::PolicyList(2UL);
   policies->length(2UL);
   policies[0UL] = orb->create_policy(
      PortableServerExt::SERVER_ENGINE_POLICY_TYPE, seAny);
      policies[1Ul] = rootPOA->create_lifespan_policy
         (PortableServer::PERSISTENT);
   // Create our POA with our policies
   PortableServer::POAManager_var manager =
      rootPOA->the_POAManager();
   PortableServer::POA_var myPOA = rootPOA->create_POA(
         "bank_se_policy_poa", manager, policies);

   // Create the servant
   AccountManagerImpl* managerServant =
         new AccountManagerImpl();
   // Activate the servant
   PortableServer::ObjectId_var oid =
         PortableServer::string_to_ObjectId("BankManager");
   myPOA->activate_object_with_id(oid ,managerServant);
   // Obtain the reference
   CORBA::Object_var ref =
            myPOA->servant_to_reference(managerServant);
   CORBA::String_var string_ref =
            orb->object_to_string(ref.in());
   ofstream refFile("ref.dat");
   refFile << string_ref << endl;
   refFile.close();
   // Activate the POA manager
   manager->activate();
   // Wait for Incoming Requests
   cout << "AccountManager Server ready" << endl;
   orb->run();
   }
   catch(const CORBA::Exception& e) {
   cerr << e << endl;
   return (1);
   }
return (0);
}
```

コードサンプル 7-24　特定サーバエンジンによる POA の生成（Java）

```
// Server.java
import org.omg.PortableServer.*;
public class Server {
   public static void main(String[ ] args) {
      try {
      // Initialize the ORB.
      org.omg.CORBA.ORB orb =
                           org.omg.CORBA.ORB.init(args,null);
      // Get property manager
      com.inprise.vbroker.properties.PropertyManager pm =
           ((com.inprise.vbroker.orb.ORB)orb).
                                        getPropertyManager();
      pm.addProperty("vbroker.se.mySe.host", "");
      pm.addProperty("vbroker.se.mySe.proxyHost", "");
      pm.addProperty("vbroker.se.mySe.scms", "scmlist");
      pm.addProperty(
         "vbroker.se.mySe.scm.scmlist.manager.type",
         "Socket");
      pm.addProperty(
         "vbroker.se.mySe.scm.scmlist.manager.connectionMax",
          100);
      pm.addProperty(
```

```
            "vbroker.se.mySe.scm.scmlist.
             manager.connectionMaxIdle", 300);
          pm.addProperty(
            "vbroker.se.mySe.scm.scmlist.listener.giopVersion", "1.2");
          pm.addProperty(
            "vbroker.se.mySe.scm.scmlist.listener.type", "IIOP");
          pm.addProperty(
            "vbroker.se.mySe.scm.scmlist.listener.port", 55000);
          pm.addProperty(
            "vbroker.se.mySe.scm.scmlist.listener.proxyPort", 0);
          pm.addProperty(
             "vbroker.se.mySe.scm.scmlist.dispatcher.type",
             "ThreadPool");
          pm.addProperty(
             "vbroker.se.mySe.scm.scmlist.dispatcher.threadMax",
              100);
          pm.addProperty(
             "vbroker.se.mySe.scm.scmlist.dispatcher.threadMin",
              5);
          pm.addProperty(
             "vbroker.se.mySe.scm.scmlist.
              dispatcher.threadMaxIdle",300);
          // get a reference to the rootPOA
          POA rootPOA = POAHelper.narrow(
             orb.resolve_initial_references("RootPOA"));
          // Create our server engine policy
          org.omg.CORBA.Any seAny = orb.create_any();
          org.omg.CORBA.StringSequenceHelper.insert(
             seAny, new String[ ]{"mySe"});
          org.omg.CORBA.Policy sePolicy =
          orb.create_policy(
                com.inprise.vbroker.PortableServerExt.
                SERVER_ENGINE_POLICY_TYPE.value, seAny);
             // Create policies for our persistent POA
             org.omg.CORBA.Policy[ ] policies = {
             rootPOA.create_lifespan_policy(
                 LifespanPolicyValue.PERSISTENT),sePolicy
          };
          // Create myPOA with the right policies
          POA myPOA = rootPOA.create_POA("bank_se_policy_poa",
                rootPOA.the_POAManager(),
                policies );
          // Create the servant
          AccountManagerImpl managerServant =
                                      new AccountManagerImpl();
          // Activate the servant
          myPOA.activate_object_with_id(
             "BankManager".getBytes(), managerServant);
          // Obtaining the reference
          org.omg.CORBA.Object ref = myPOA.servant_to_reference(
             managerServant);
          // Now write out the IOR
          try {
             java.io.PrintWriter pw =
                   new java.io.PrintWriter(
                             new java.io.FileWriter("ior.dat"));
             pw.println(orb.object_to_string(ref));
             pw.close();
          } catch (java.io.IOException e ) {
             System.out.println(
                       "Error writing the IOR to file ior.dat");
             return;
          }
          // Activate the POA manager
          rootPOA.the_POAManager().activate();
          System.out.println(ref + "is ready.");
          // Wait for incoming requests
          orb.run();
          } catch (Exception e){
             e.printStackTrace();
          }
       }
    }
```

7.8　アダプタアクティベータ

アダプタアクティベータは POA と対応し，オンデマンドで子 POA を生成する機能を提供します。これが実行できるのは，find_POA オペレーション中か，または特定の子 POA の名前を指定したリクエストを受信したときです。

アダプタアクティベータは，子 POA（またはその子の一つ）の名前を指定したリクエストを受信したり，活性化パラメタの値を TRUE に設定して find_POA を呼び出したりしたときに，オンデマンドで子 POA を生成する機能を POA に提供します。実行開始時に必要なすべての POA を生成するアプリケーションサーバには，アダプタアクティベータを使用したり提供したりする必要はありません。アダプタアクティベータはリクエスト処理中に POA を作成する場合だけ必要です。

POA からアダプタアクティベータへのリクエストの処理中は，新 POA（または任意の子孫 POA）が管理するオブジェクトへのリクエストはすべて待ち行列に入れられます。このシリアリゼーションによって，アダプタアクティベータは新 POA にリクエストが配達される前に，その POA の初期化を完了できます。

アダプタアクティベータの使用例については，POA のサンプルプログラム adaptor_activator を参照してください。

7.9　リクエストの処理

リクエストには，ターゲットのオジェクトのオブジェクト ID と，ターゲットのオブジェクトリファレンスを作成した POA が含まれます。クライアントがリクエストを送信すると，VisiBroker ORB はまず適切なサーバを探すか，必要ならサーバを起動します。それから VisiBroker ORB はそのサーバ内の適切な POA を探します。

VisiBroker ORB は適切な POA を見つけると，リクエストをその POA に渡します。その時点でリクエスト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活性化状態によって異なります。オブジェクト活性化状態については，「7.4　オブジェクトの活性化」を参照してください。

- C++で POA に ServantRetentionPolicy::RETAIN が設定してあれば，または Java で POA に ServantRetentionPolicy.RETAIN が設定してあれば，POA はアクティブオブジェクトマップを見て，リクエストに指定されたオブジェクト ID に対応するサーバントを探します。サーバントが存在するなら，POA はサーバントの適切なメソッドを呼び出します。
- C++で POA に ServantRetentionPolicy::NON_RETAIN または ServantRetentionPolicy::RETAIN が設定してあるか，Java で POA に ServantRetentionPolicy.NON_RETAIN または ServantRetentionPolicy.RETAIN が設定してある状態で，適切なサーバントが見つからない場合は，次のような結果になります。
  - POA に RequestProcessingPolicy::USE_DEFAULT_SERVANT（C++），または RequestProcessingPolicy.USE_DEFAULT_SERVANT（Java）が設定してあれば，POA 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 に RequestProcessingPolicy::USE_SERVANT_MANAGER（C++），または RequestProcessingPolicy.USE_SERVANT_MANAGER（Java）が設定してあれば，POA 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 に RequestProcessingPolicy::USE_OBJECT_MAP_ONLY（C++），または RequestProcessingPolicy.USE_OBJECT_MAP_ONLY（Java）を設定してあれば，例外が発生します。

サーバントマネージャが呼び出されたが，オブジェクトをインカネートできない場合には，サーバントマネージャが ForwardRequest 例外を発生させることができます。

8 スレッドとコネクションの管理

この章では, クライアントプログラムとオブジェクトインプリメンテーションでのマルチスレッドの使用について説明します。この章を読むと，Borland Enterprise Server VisiBroker が使用するスレッドおよびコネクションモデルについて理解できます。

8.1　Borland Enterprise Server VisiBroker でのスレッドの使用

スレッドとは，プロセス中の一つの連続した制御の流れのことです。スレッドは，基本的な部分をほかのスレッドと共用することによってオーバヘッドを減らせることからライトウェイトプロセスともいいます。スレッドは軽いので，一つのプロセス内部に多くのスレッドが存在できます。

マルチスレッドを使用すると，アプリケーション内に並行性がもたらされるので，性能が向上します。独立した幾つもの計算を同時に処理するスレッドを利用して，アプリケーションを効率的に構築できます。例えば，データベースシステムは，同時に複数のファイルオペレーションとネットワークオペレーションを実行しながら，多くのユーザの処理を進めることができます。複数のリクエストを非同期に処理させる一つの制御用のスレッドでソフトウェアを作成できますが，各リクエストを独立したシーケンスとして作成し，システムに異なるオペレーション間の同期処理をさせることによって，コードを単純化できます。

マルチスレッドは次の場合に役立ちます。

- ほかの処理（ウィンドウ描画，文書の出力，マウスクリックへの応答，スプレッドシートの列計算，シグナル処理など）にお互いの処理が依存しないオペレーションのグループがある場合
- データの排他処理がほとんど発生しそうにない場合（共用データの量は識別可能で少量である場合）
- タスクをさまざまな処理に分けることができる場合。例えば，あるスレッドはシグナルを処理し，別のスレッドはユーザインタフェースを処理できる

8.2　Borland Enterprise Server VisiBroker が提供するスレッドポリシー

Borland Enterprise Server VisiBroker はスレッドプーリングとスレッドパーセッションという二つのスレッドポリシーを提供します。両モデルは基本的に次の点で異なります。

- 生成される状況
- 同じクライアントからの同時に発生したリクエストを処理する方法
- スレッドを解放する時期とその方法

デフォルトのスレッドポリシーはスレッドプーリングポリシーです。スレッドパーセッションポリシーの設定およびスレッドプーリングのプロパティの変更については,「8.6　ディスパッチポリシーとプロパティの設定」を参照してください。

8.3　スレッドプーリングポリシー

サーバがスレッドプーリングポリシーを使用する場合, サーバはクライアントリクエストの処理用に割り当て可能なスレッドの最大数を定義します。ワーカスレッドがクライアントリクエストごとに割り当てられますが，それはその特定のリクエストの期間だけです。リクエストが完了すると，そのリクエストに割り当てられたワーカスレッドは, クライアントから続いて要求されるリクエストを処理するために再度割り当てができるように，利用できるスレッドのプールに入れられます。

このモデルを使用すると, スレッドはサーバオブジェクトに対するリクエストトラフィックの量に基づいて割り当てられます。つまり, サーバに対し同時に多くのリクエストをする非常にアクティブなクライアントはマルチスレッドによってサービスされ，各リクエストの迅速な実行が保証されるのに対し，それほどアクティブでないクライアントは一つのスレッドを共用でき，そのリクエストは即時にサービスを受けられます。その上，スレッドはデストラクトされるのではなく再利用され，複数のコネクションに割り当てることができるので，ワーカスレッドの生成およびデストラクトに対応するオーバヘッドは減少します。

Borland Enterprise Server VisiBroker は，同時のクライアントリクエストの数に基づいてスレッドプール中のスレッド数を動的に割り当てることでシステム資源を節約します。クライアントが非常にアクティブになると，スレッドはそのニーズに合わせて割り当てられます。スレッドがアイドルのままなら，Borland Enterprise Server VisiBroker は現在のクライアントリクエストを満たすだけのスレッドを残してスレッドを解放します。これによって，サーバのアクティブなスレッド数は常に最適な数に保たれます。

スレッドプールのサイズはサーバアクティビティに基づいて大きくなり, 個々の分散システムのニーズに合わせて実行前または実行中に設定できます。スレッドプーリングで設定できる内容は，次のとおりです。

- 最大スレッド数および最小スレッド数
- 最大アイドル時間

クライアントリクエストを受信するたびに, そのリクエストを処理するためにスレッドプールからスレッドを割り当てようとします。これが最初のクライアントリクエストで，プールが空なら，スレッドが生成されます。同様に，すべてのスレッドがビジーなら，新しいスレッドが生成されてそのリクエストが処理されます。

サーバはクライアントリクエストを処理するために割り当て可能なスレッドの最大数を定義できます。利用できるスレッドがプール中になく，最大数のスレッドがすでに生成されている場合は，現在使用中のスレッドが解放されてプール中に戻されるまで，そのリクエストは待たされます。

スレッドプーリングはデフォルトのスレッドポリシーです。特に環境を設定する必要はありません。スレッドプーリングのプロパティを設定したい場合は,「8.6　ディスパッチポリシーとプロパティの設定」を参照してください。

図 8-1　スレッドのプールが利用できる

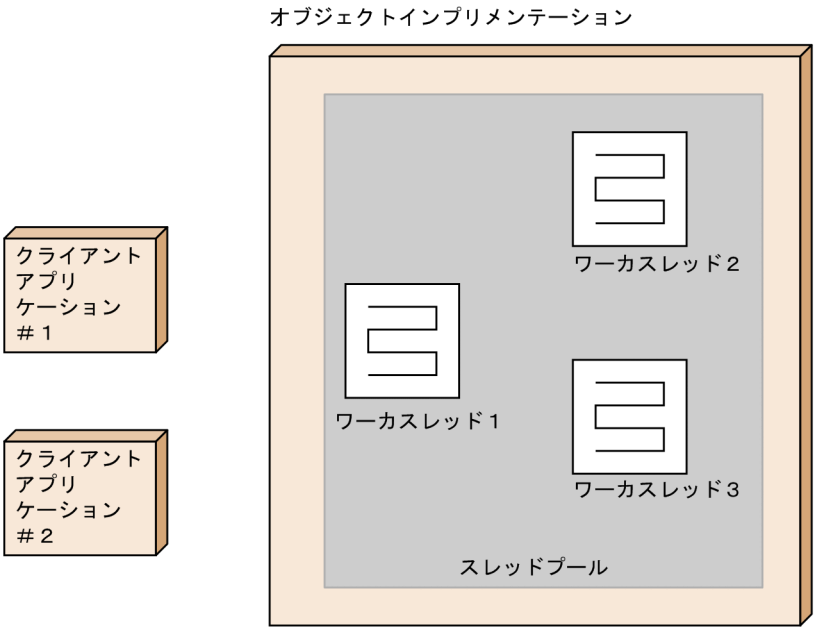


図 8-1 は，スレッドプーリングポリシーを使用したオブジェクトインプリメンテーションを示しています。その名が示すとおり，このポリシーではワーカスレッドをプールできます。

図 8-2　クライアントアプリケーション#1 がリクエストを送信

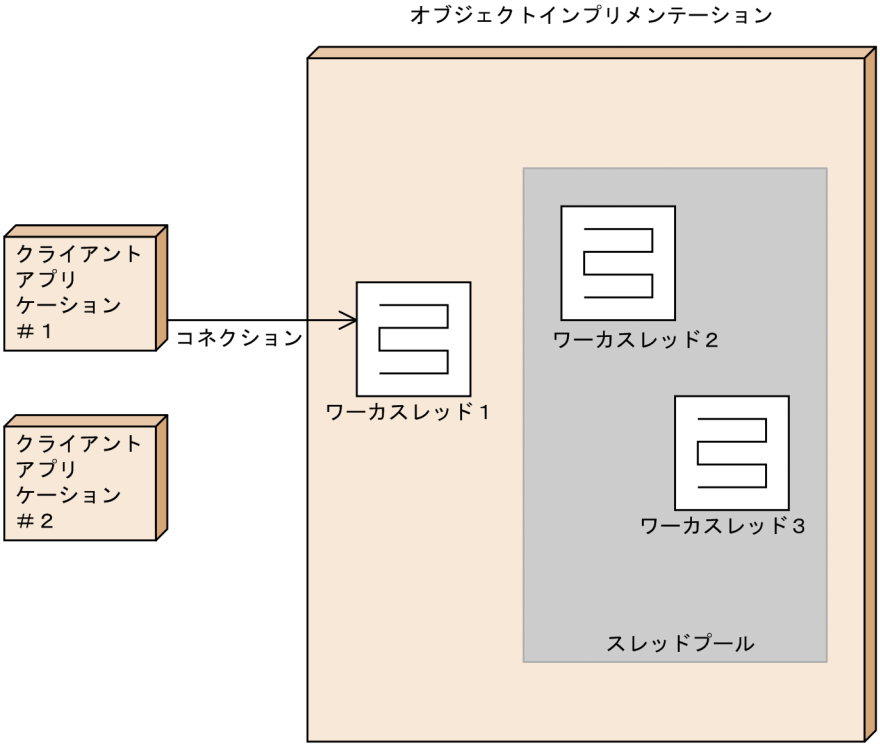


図 8-2 では，クライアントアプリケーション#1 がオブジェクトインプリメンテーションとのコネクションを確立して，リクエストの処理のためにスレッドが生成されます。スレッドプーリングでは，クライアントごとに一つのコネクションがあり，コネクションごとに一つのスレッドがあります。リクエストが入ってくると，ワーカスレッドはリクエストを受信します。そのワーカスレッドはもうプール中にはありません。

ワーカスレッドはスレッドプールから削除され，リクエストがあるかどうかを常に監視します。リクエストが入ってくると，そのワーカスレッドはリクエストを読み込んで，そのリクエストを適切なオブジェクトインプリメンテーションにディスパッチします。リクエストをディスパッチする前に，ワーカスレッドは次のリクエストの有無を監視するほかのワーカスレッドを一つ割り当てます。

図 8-3　クライアントアプリケーション#2 がリクエストを送信

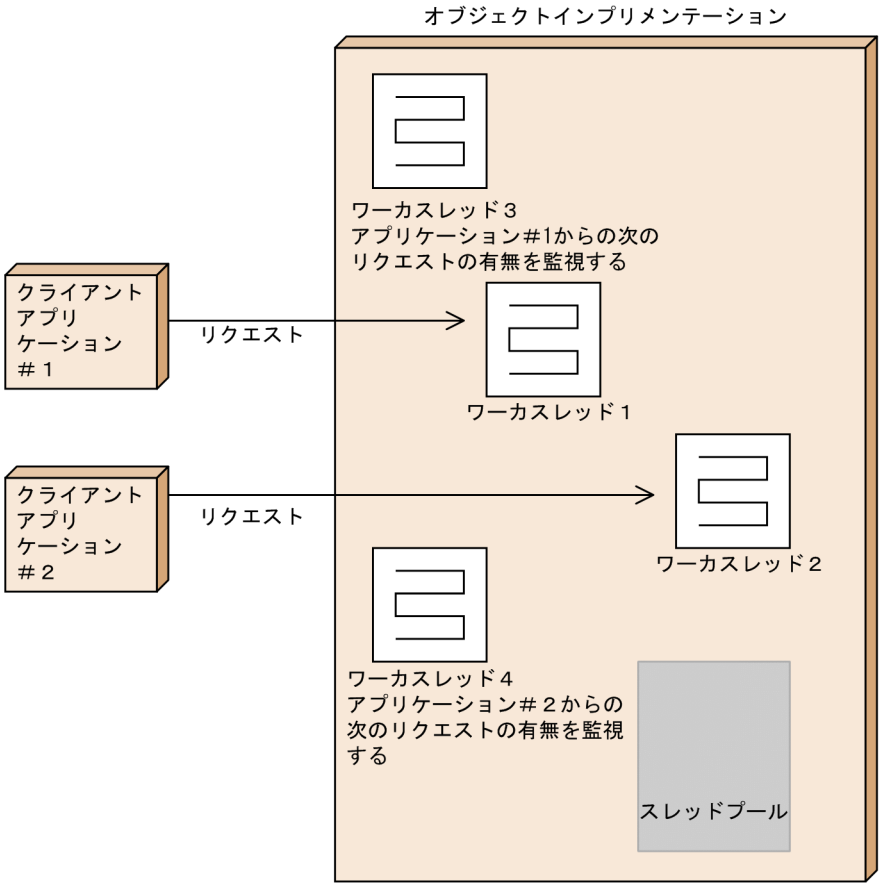


図 8-3 で示すように，クライアントアプリケーション#2 が自身のコネクションを確立してリクエストを送信すると，第 2 のワーカスレッドが生成されます。現在はワーカスレッド 3 が入力リクエストの有無を監視しています。

図 8-4　クライアントアプリケーション#1 が 2 番目のリクエストを送信

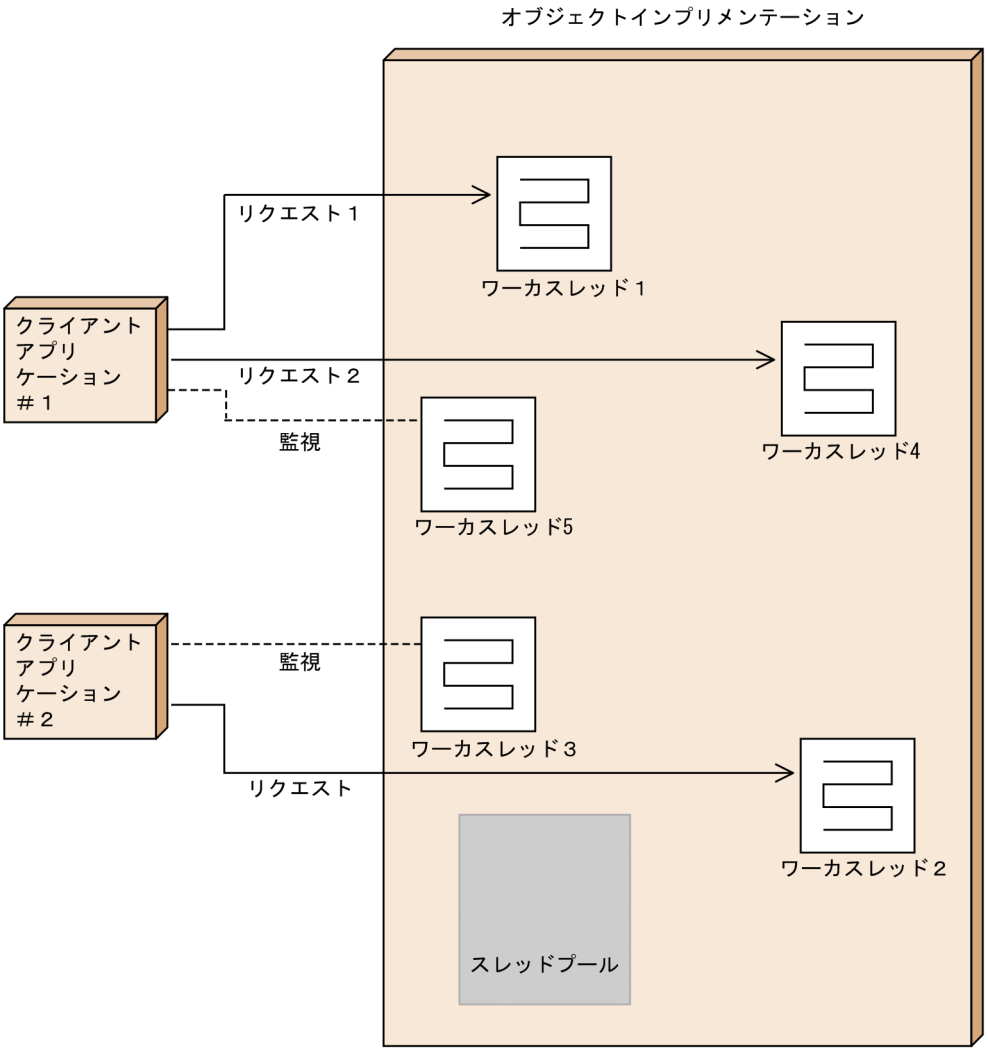


図 8-4 は，クライアントアプリケーション#1 から 2 番目のリクエストが入ってくると，ワーカスレッド 4 を使用することを示しています。新しいリクエストの有無を監視するためにワーカスレッド 5 が生成されます。クライアントアプリケーション#1 からさらにリクエストが入ってきたら，各リクエストを処理するためにワーカスレッドが次々に割り当てられます（各ワーカスレッドは，監視スレッドがリクエストを受信したあとに生成されます）。ワーカスレッドはそのタスクを完了すると，スレッドプールに戻され，クライアントからのリクエストを処理するために利用できる状態になります。

8.4　スレッドパーセッションポリシー

スレッドパーセッションポリシーでは，スレッドはクライアントおよびサーバのプロセス間のコネクションによって割り当てられます。サーバがスレッドパーセッションポリシーを選択すると，新しいクライアントがサーバに接続するたびに新しいスレッドが割り当てられます。個々のクライアントから受信したすべてのリクエストを処理するために，一つのスレッドが割り当てられます。このため，スレッドパーセッションはスレッドパーコネクションとも呼ばれます。クライアントがサーバとのコネクションを切断すると，スレッドはデストラクトされます。クライアントコネクションごとに割り当て可能なスレッドの最大数は，vbroker.se.iiop_ts.scm.iiop_ts.manager.connectionMax プロパティの設定で制限できます。

図 8-5　スレッドパーセッションポリシーを使用したオブジェクトインプリメンテーション

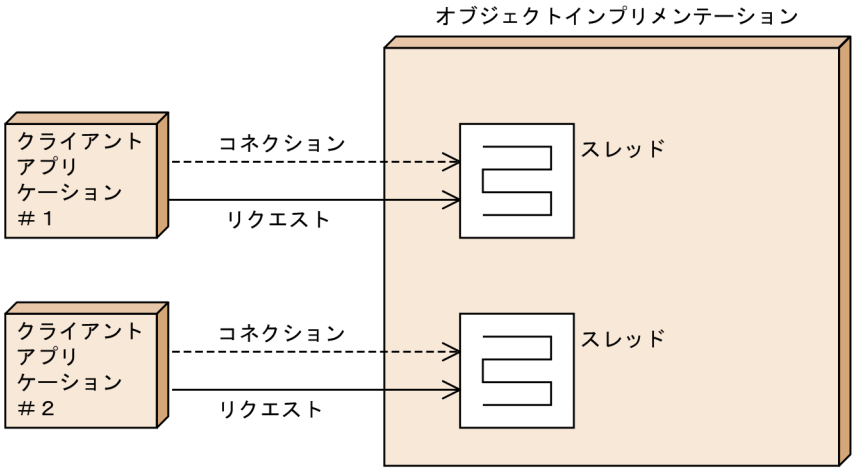


図 8-5 は，スレッドパーセッションポリシーの使用を示しています。クライアントアプリケーション#1 がオブジェクトインプリメンテーションとのコネクションを確立します。クライアントアプリケーション#2 とオブジェクトインプリメンテーションの間には別のコネクションが存在します。クライアントアプリケーション#1 からオブジェクトインプリメンテーションにリクエストが入ってくると，ワーカスレッドがそのリクエストを処理します。クライアントアプリケーション#2 からリクエストが入ってくると，そのリクエストを処理するために別のワーカスレッドが割り当てられます。

図 8-6　同じクライアントから 2 番目のリクエストが入ってくる

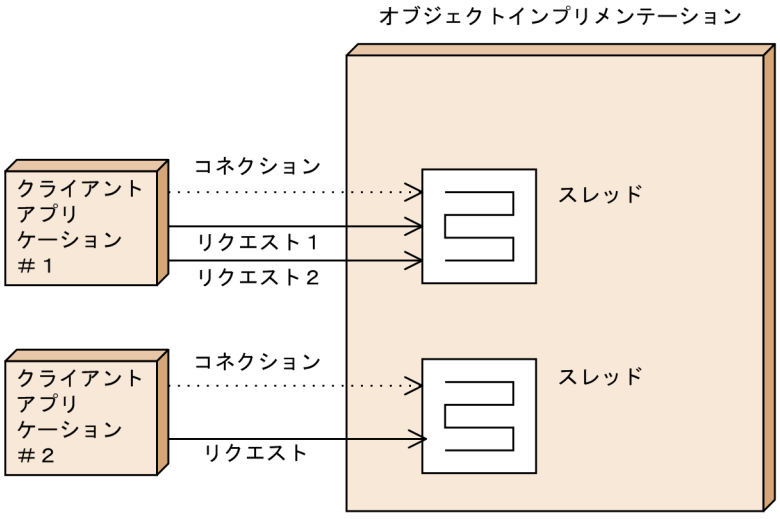


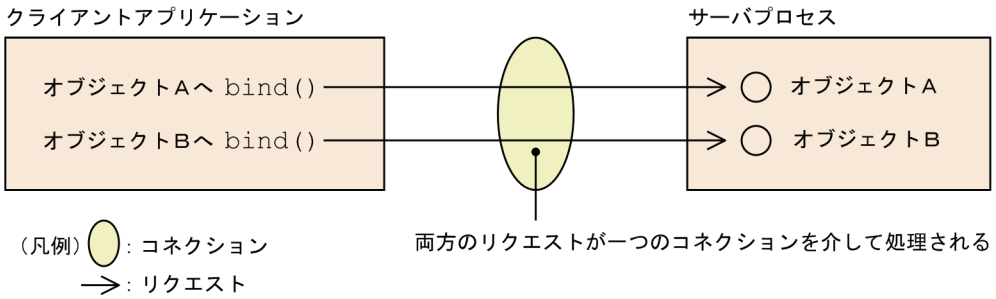
図 8-6 は，クライアントアプリケーション#1 から 2 番目のリクエストがオブジェクトインプリメンテーションに入ってきたところを示しています。リクエスト 1 を処理するのと同じスレッドがリクエスト 2 を処理します。このスレッドは，リクエスト 1 の処理を完了するまでリクエスト 2 を待たせます（スレッドパーセッションでは，同じクライアントからの複数のリクエストは並列処理しません）。リクエスト 1 が完了すると，スレッドはクライアントアプリケーション#1 からのリクエスト 2 を処理できます。クライアントアプリケーション#1 から複数のリクエストが入ってくることがありますが，リクエストは入ってきた順に処理され，追加スレッドがクライアントアプリケーション#1 に割り当てられることはありません。

8.5　Borland Enterprise Server VisiBroker が提供するコネクション管理

基本的に，Borland Enterprise Server VisiBroker が提供するコネクション管理はサーバとのクライアントコネクションの数を最小にします。つまり，サーバプロセスごとにコネクションは一つだけあり，これが共有されます。すべてのクライアントリクエストは，たとえ別々のスレッドから発行されても，同じコネクションで多重化されます。さらに，解放されたクライアントコネクションは同じサーバとの以降の再コネクションのために再利用されるので，クライアントはサーバとの新しいコネクションのオーバヘッドを発生させないで済みます。

図 8-7 のシナリオでは，クライアントアプリケーションがサーバプロセスの二つのオブジェクトにバインドされています。それぞれの bind()メソッドは，サーバプロセス中の別々のオブジェクト用 bind()メソッドであっても，サーバプロセスとの共通コネクションを共有します。

図 8-7　同じサーバプロセス中の二つのオブジェクトにバインド



サーバ上の一つのオブジェクトにバインドされたマルチスレッドを使用したクライアントのコネクションを図 8-8 に示します。

図 8-8　サーバプロセス中の一つのオブジェクトにバインド

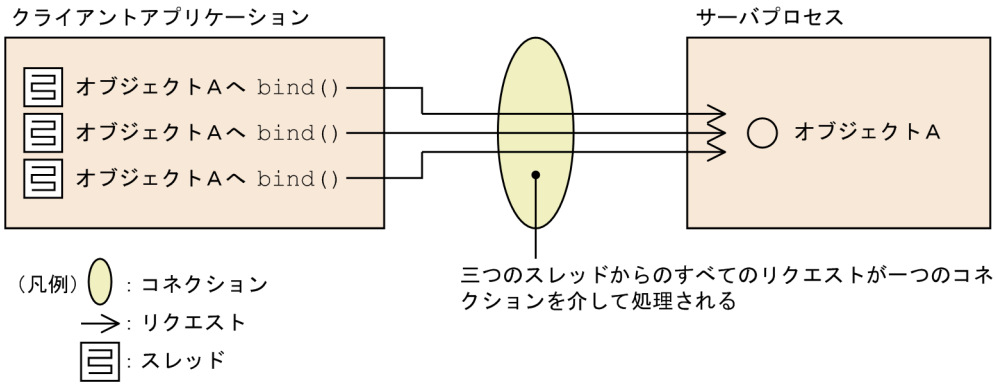


図 8-8 で示すように，すべてのスレッドからのすべての呼び出しは，同じコネクションによってサービスを受けます。図 8-8 で示したシナリオの場合，使用する最も効率的なマルチスレッドモデルはスレッドプーリングモデル（デフォルト）です。このシナリオでスレッドパーセッションモデルを使用すると，クライアントアプリケーション中のすべてのスレッドからのすべてのリクエストを処理するためにサーバ上のスレッドが一つだけ割り当てられ，その結果，性能が低下してしまいます。

サーバへの，またはクライアントからのコネクションの最大数を設定できます。アイドルなコネクションは最大数に達したときに再利用されるので，資源を確実に節約できます。

8.6　ディスパッチポリシーとプロパティの設定

マルチスレッドのオブジェクトサーバ内の各 POA は，スレッドパーセッションとスレッドプーリングという二つのディスパッチモデルから選択できます。ディスパッチポリシーを選択するには，ServerEngine の dispatcher.type プロパティを設定してください。

```
vbroker.se.<srvr_eng_name>.scm.
    <srvr_connection_mngr_name>.dispatcher.type="ThreadPool"
vbroker.se.<srvr_eng_name>.scm.
 <srvr_connection_mngr_name>.dispatcher.type="ThreadSession"
```

これらのプロパティの詳細については，「7.7　監視プロパティとディスパッチプロパティの設定」，またはマニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「サーバ側スレッドセッションコネクションのプロパティ」および「サーバ側スレッドプールコネクションのプロパティ」の記述を参照してください。

8.6.1　スレッドプーリング

ThreadPool（スレッドプーリング）は，ServerEnginePolicy を指定しないで POA を生成する場合のデフォルトのディスパッチポリシーです。

ThreadPool に指定できるプロパティについては，マニュアル「Borland Enterprise Server VisiBroker プログラマーズリファレンス」の「サーバ側スレッドプールコネクションのプロパティ」および「サーバ側ローカルスレッドプールコネクションのプロパティ」を参照してください。

8.6.2　スレッドパーセッション

ディスパッチャタイプとして ThreadSession を使用する場合は，se.default プロパティを iiop_ts に設定してください。

```
vbroker.se.default=iiop_ts
```

8.6.3　コーディングの考慮事項

VisiBroker ORB オブジェクトをインプリメントするサーバ内のコードは，すべてがスレッドセーフである必要があります。オブジェクトインプリメンテーション内のシステム全体にわたる資源にアクセスする場合は特に注意が必要です。例えば，スレッドセーフではないデータベースアクセスメソッドが多くあるとします。オブジェクトインプリメンテーションはこのような資源にアクセスを試みる前に，まず排他制御してその資源に対するアクセスをロックする必要があります。

オブジェクトへのシリアライズなアクセスが必要な場合は，ThreadPolicy の値に SINGLE_THREAD_MODEL を設定してこのオブジェクトを活性化する POA を生成する必要があります。

9 tie 機能の使用

この章では，tie 機能を使用して既存の C++および Java コードを分散オブジェクトシステムに組み込む方法について説明します。この章を読むと，デリゲータインプリメンテーションを生成したり，インプリメンテーション継承を提供したりできます。

9.1 tie 機能の働き

通常，オブジェクトインプリメンテーションクラスは，idl2cpp または idl2java コンパイラによって生成されたサーバントクラスを継承します。また，サーバントクラスは，PortableServer.Servant::Servant（C++），org.omg.PortableServer.Servant（Java）を継承します。既存のクラスを変更して Borland Enterprise Server VisiBroker サーバントクラスを継承するのが不便または不可能な場合は，tie 機能が代替手段となります。

tie 機能はオブジェクトサーバに，PortableServer::Servant（C++）または org.omg.PortableServer.Servant（Java）を継承するデリゲータインプリメンテーションクラスを提供します。デリゲータインプリメンテーションは独自のセマンティクスを持たないで，ただ受信したすべてのリクエストを，個別にインプリメントできる実インプリメンテーションクラスにデリゲート（委任）するだけです。実インプリメンテーションクラスは，PortableServer::Servant（C++），または org.omg.PortableServer.Servant（Java）を継承する必要はありません。

tie 機能を使用することで，二つのファイルが IDL コンパイラから生成されます。

- <interface_name>POATie は，すべての IDL 定義メソッドのインプリメンテーションをデリゲートまで遅延させます。デリゲートは<interface_name>Operations インタフェースをインプリメントします。レガシーインプリメンテーションは，オペレーションインタフェースをインプリメントし，またリアルインプリメンテーションをデリゲートするために少しだけ継承できます。
- <interface_name>Operations は，オブジェクトインプリメンテーションがインプリメントしなければならないすべてのメソッドを定義します。このインタフェースは，tie 機能を使用する場合に，対応する<interface_name>POATie クラスのデリゲートオブジェクトとして動作します。

9.2　サンプルプログラム

9.2.1　tie 機能を使用したサンプルプログラムの格納場所

tie 機能を使用したサンプルの Bank は，Borland Enterprise Server VisiBroker をインストールしたディレクトリの examples/vbe/basic/bank_tie 下に入っています。

9.2.2　tie テンプレートの考察（C++）

idl2cpp コンパイラは，コードサンプル 9-1 に示すように_tie_Account テンプレートクラスを自動的に生成します。POA_Bank_Account_tie クラスは，オブジェクトサーバが実体化して，AccountImpl のインスタンスで初期化します。POA_Bank_Account_tie クラスは，受信するオペレーション要求すべてを実際のインプリメンテーションである AccountImpl に任せます。このサンプルでは，AccountImpl クラスは POA_Bank::Account クラスを継承しません。

コードサンプル 9-1　POA_Bank_Account_tie テンプレートの考察

```
. . .
template <class T> class POA_Bank_Account_tie :
      public POA_Bank::Account {
   private:
      CORBA::Boolean _rel;
      PortableServer::POA_ptr _poa;
      T *_ptr;
      POA_Bank_Account_tie(const POA_Bank_Account_tie&) {}
      void operator=(const POA_Bank_Account_tie&) {}

   public:
      POA_Bank_Account_tie (T& t): _ptr(&t), _poa(NULL),
            _rel((CORBA::Boolean)0) {}

      POA_Bank_Account_tie (T& t, PortableServer::POA_ptr poa):
            _ptr(&t),
            _poa(PortableServer::_duplicate(poa)),
            _rel((CORBA::Boolean)0) {}

      POA_Bank_Account_tie (T *p, CORBA::Boolean release = 1) :
            _ptr(p), _poa(NULL),_rel(release) {}

      POA_Bank_Account_tie (T *p, PortableServer::POA_ptr poa,
            CORBA::Boolean release = 1)
            : _ptr(p), _poa(PortableServer::_duplicate(poa)),
            _rel(release) {}

      virtual ~POA_Bank_Account_tie() {
         CORBA::release(_poa);
         if (_rel) {
            delete _ptr;
         }
      }

      T* _tied_object() { return _ptr; }

      void _tied_object(T& t) {
         if (_rel) {
         delete _ptr;
      }
      _ptr = &t;
      _rel = 0;
   }

   void _tied_object(T *p, CORBA::Boolean release=1) {
      if (_rel) {
         delete _ptr;
      }
      _ptr = p;
```

```
        _rel = release;
    }

    CORBA::Boolean _is_owner() { return _rel; }

    void _is_owner(CORBA::Boolean b) { _rel = b; }

    CORBA::Float balance() {
        return _ptr->balance();
    }

    PortableServer::POA_ptr _default_POA() {
        if ( !CORBA::is_nil(_poa) ) {
            return _poa;
        } else {
            return PortableServer_ServantBase::_default_POA();
        }
    }
};
```

9.2.3　_tie_Account クラスを使用するためのサーバの変更（C++）

コードサンプル 9-2 に_tie_Account クラスを使用する際に必要な Server.C ファイルの変更内容を示します。

コードサンプル 9-2　_tie クラスを使用したサーバの例

```
#include "Bank_s.hh"
#include <math.h>
. . .
int main(int argc, char* const* argv) {
    try {
        // Initialize the ORB.
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // get a reference to the rootPOA
        PortableServer::POA_var rootPOA =
              PortableServer::POA::_narrow(
              orb->resolve_initial_references("RootPOA"));

        CORBA::PolicyList policies;
        policies.length(1);
        policies[(CORBA::ULong)0] =
              rootPOA->create_lifespan_policy(
              PortableServer::PERSISTENT);

        // get the POA Manager
        PortableServer::POAManager_var poa_manager =
              rootPOA->the_POAManager();

        // Create myPOA with the right policies
        PortableServer::POA_var myPOA =
              rootPOA->create_POA("bank_agent_poa",
              poa_manager, policies);
        // Create the servant
        AccountManagerImpl managerServant(rootPOA);

        // Create the delegator
        POA_Bank_AccountManager_tie<AccountManagerImpl>
              tieServer(managerServant);

        // Decide on the ID for the servant
        PortableServer::ObjectId_var managerId =
              PortableServer::string_to_ObjectId("BankManager");

        // Activate the servant with the ID on myPOA
        myPOA->activate_object_with_id(managerId, &tieServer);

        // Activate the POA Manager
        poa_manager->activate();

        cout << myPOA->servant_to_reference(&tieServer) <<
```

```
            "is ready" << endl;

        // Wait for incoming requests
        orb->run();
    } catch(const CORBA::Exception& e) {
        cerr << e << endl;
        return 1;
    }
    return 0;
}
```

9.2.4　Server クラスの変更（Java）

次のコードサンプルは，Server クラスに加える変更を示します。AccountManagerPOATie のインスタンスを生成するための追加手順に注意してください。

コードサンプル 9-3　bank_tie ディレクトリの Server.java ファイル

```
import org.omg.PortableServer.*;
public class Server {
   public static void main(String[ ] args) {
      try {
         // Initialize the ORB.
         org.omg.CORBA.ORB orb =
                          org.omg.CORBA.ORB.init(args,null);
         // get a reference to the rootPOA
         POA rootPOA = POAHelper.narrow(
            orb.resolve_initial_references("RootPOA"));
         // Create policies for our persistent POA
         org.omg.CORBA.Policy[ ] policies = {
              rootPOA.create_lifespan_policy(
                LifespanPolicyValue.PERSISTENT)
         };
         // Create myPOA with the right policies
         POA myPOA = rootPOA.create_POA("bank_agent_poa",
              rootPOA.the_POAManager(), policies);
         // Create the tie which delegates
         //      to an instance of AccountManagerImpl
         Bank.AccountManagerPOATie tie =
              new Bank.AccountManagerPOATie(
                 new AccountManagerImpl(rootPOA));
         // Decide on the ID for the servant
         byte[ ] managerId = "BankManager".getBytes();
         // Activate the servant with the ID on myPOA
         myPOA.activate_object_with_id(managerId,tie);
         // Activate the POA manager
         rootPOA.the_POAManager().activate();
         System.out.println("Server is ready.");
         // Wait for incoming requests
         orb.run();
      } catch (Exception e){
         e.printStackTrace();
      }
   }
}
```

9.2.5　AccountManager の変更（Java）

AccountManager クラスに加える変更（サンプル bank_agent と比較して）は次のとおりです。

- AccountManagerImpl は，Bank.AccountManagerPOA をもう継承しません。
- 新しい Account を生成する場合には，AccountPOATie も生成し，初期化します。

コードサンプル 9-4　AccountManagerImpl クラス

```
import org.omg.PortableServer.*;
import java.util.*;
```

```
public class AccountManagerImpl implements
                          Bank.AccountManagerOperations {
   public AccountManagerImpl(POA poa) {
      _accountPOA = poa;
   }
   public synchronized Bank.Account open(String name) {
      // Lookup the account in the account dictionary.
      Bank.Account account =
                        (Bank.Account)_accounts.get(name);
      // If there was no account in the dictionary,
      // create one.
      if (account == null) {
         // Make up the account's balance, between 0 and
         // 1000 dollars.
         float balance =
               Math.abs(_random.nextInt()) % 100000 / 100f;
         // Create an account tie
         // which delegate to an instance of AccountImpl
         Bank.AccountPOATie tie =
           new Bank.AccountPOATie(new AccountImpl(balance));
         try {
            // Activate it on the default POA
            // which is rootPOA for this servant
            account = Bank.AccountHelper.narrow(
                     _accountPOA.servant_to_reference(tie));
         }catch (Exception e){
            e.printStackTrace();
         }
         // Print out the new account.
         System.out.println(
              "Created " + name + "'s account: " + account);
         // Save the account in the account dictionary.
         _accounts.put(name, account);
      }
      // Return the account.
      return account;
   }
   private Dictionary _accounts = new Hashtable();
   private Random _random = new Random();
   private POA _accountPOA = null;
}
```

9.2.6　Account クラスの変更（Java）

Account クラスに加える変更（サンプル bank_agent と比較して）は，もう Bank.AccountPOA を継承しなくするということだけです。

コードサンプル 9-5　AccountImpl クラス

```
// Server.java
public class AccountImpl implements Bank.AccountOperations {
   public AccountImpl(float balance) {
      _balance = balance;
   }
   public float balance() {
      return _balance;
   }
   private float _balance;
}
```

9.2.7　tie のサンプルプログラムの構築

「4.　Borland Enterprise Server VisiBroker によるサンプルアプリケーションの開発」で説明した内容も，tie 機能を実装するサンプルに流用できます。

10 クライアントの基本事項

この章では, クライアントプログラムがどのように分散オブジェクトにアクセスして使用するのかについて説明します。

10.1　VisiBroker ORB の初期化

ORB（Object Request Broker）はクライアントとサーバ間の通信リンクを提供します。クライアントがリクエストすると，VisiBroker ORB はオブジェクトインプリメンテーションを探して，リクエストをそのオブジェクトに渡し（必要ならオブジェクトを活性化して），クライアントに応答を返します。クライアントは，オブジェクトが同じマシンにあるのかネットワークのどこかにあるのかを意識しません。

注（Java の場合）

VisiBroker ORB は，システムリソースを集中的に使用するため，一つのプロセスにつき一つの VisiBroker ORB のインスタンスだけを作成することをお勧めします。

VisiBroker ORB が実行する作業の大部分はプログラマに意識されませんが，クライアントプログラムは明示的に VisiBroker ORB を初期化しなければなりません。

マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「コマンドラインオプション」または「Borland Enterprise Server VisiBroker プロパティ」の記述で説明している VisiBroker ORB オプションをコマンドライン引数として指定できます。したがって，これらのオプションが確実に有効になるように，ORB_init に argc と argv（C++），または ORB.init に args（Java）を渡す必要があります。コードサンプル 10-1 に VisiBroker ORB の初期化（C++），コードサンプル 10-2 に VisiBroker ORB の初期化（Java）を示します。

コードサンプル 10-1　VisiBroker ORB の初期化（C++）

```
#include <fstream.h>
#include "Bank_c.hh"

int main(int argc, char* const* argv) {
   CORBA::ORB_var orb;
   CORBA::Float balance;
   try {
      // Initialize the ORB.
      orb = CORBA::ORB_init(argc, argv);
      . . .
    } catch {
      . . .
    }
}
```

コードサンプル 10-2　VisiBroker ORB の初期化（Java）

```
public class Client {
   public static void main (String[ ] args) {
      org.omg.CORBA.ORB orb =
                        org.omg.CORBA.ORB.init(args, null);
      . . .
    }
}
```

10.2　オブジェクトへのバインド

クライアントプログラムは，オブジェクトのリファレンスを取得することによってリモートオブジェクトを使用します。オブジェクトリファレンスは，通常は静的<interface_name>::_bind()メソッド（C++），または<interface_name>Helper の bind()メソッド（Java）を使用して取得されます。VisiBroker ORB はオブジェクトをインプリメントするサーバを探してそのサーバとのコネクションを確立する，というようなオブジェクトリファレンス取得についての手順の大部分を隠します。

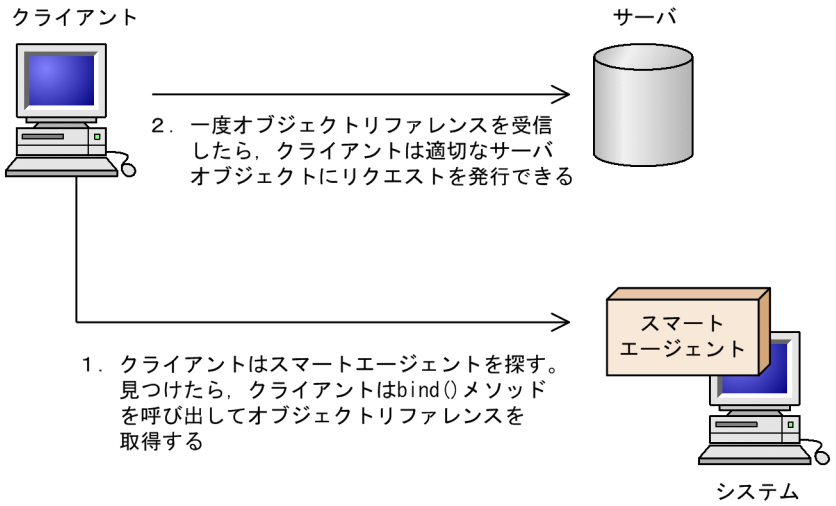
10.2.1　バインドプロセス中に行われる動作

サーバプログラムが create_POA()を実行すると，生成した POA がスマートエージェントに登録されます。

この状態で，クライアントプログラムが静的 bind()メソッドを呼び出した場合，VisiBroker ORB はプログラムのために幾つかの機能を実行し，スマートエージェントからオブジェクトインプリメンテーションの位置情報を得られます。クライアントとスマートエージェントの相互動作を図 10-1 に示します。

- VisiBroker ORB は，リクエストされたインタフェースを提供するオブジェクトインプリメンテーションを探すためにスマートエージェントにコンタクトします。_bind()メソッド（C++）または bind()メソッド（Java）の呼び出し時にオブジェクト名が指定されていたら，その名前はディレクトリサービス検索で使用します。「15.　オブジェクト活性化デーモンの使用」で説明する OAD（オブジェクト活性化デーモン）にサーバオブジェクトが登録されている場合は，OAD を起動しておく必要があります。
- オブジェクトインプリメンテーションが見つかると，VisiBroker ORB は見つけたオブジェクトインプリメンテーションとクライアントプログラム間のコネクション確立を試みます。
- コネクションの確立が成功すると，VisiBroker ORB はプロキシオブジェクトを生成して，そのオブジェクトのリファレンスを返します。クライアントはサーバオブジェクトと相互動作するプロキシオブジェクトのメソッドを呼び出します。

図 10–1　クライアントとスマートエージェントの相互動作



注

クライアントプログラムがサーバクラスのコンストラクタを呼び出すことはありません。その代わり，静的_bind()メソッド（C++），または静的 bind()メソッド（Java）呼び出しによってオブジェクトリファレンスを取得できます。

コードサンプル 10-3　バインド呼び出しの例（C++）

```
. . .
PortableServer::ObjectId_var manager_id =
   PortableServer::string_to_ObjectId("BankManager");
Bank::AccountManager_var =
   Bank::AccountManager::_bind("/bank_agent_poa", manager_id);
. . .
```

コードサンプル 10-4　バインド呼び出しの例（Java）

```
. . .
Bank.AccountManager manager =
   Bank.AccountManagerHelper.bind(orb,
               "/bank_agent_poa", "BankManager".getBytes());
. . .
```

10.3　オブジェクトのオペレーションの呼び出し

クライアントプログラムは,オブジェクトリファレンスを使用してオブジェクトのオペレーションを呼び出したりオブジェクトに含まれたデータを参照したりします。オブジェクトリファレンスのさまざまな操作方法は,「10.4　オブジェクトリファレンスの操作」で説明します。

コードサンプル 10-5　オブジェクトリファレンスを使用したオペレーションの呼び出し（C++）

```
. . .
//  Invoke the balance operation.
   balance = account->balance();
   cout <<"Balance is $" << balance << endl;
. . .
```

コードサンプル 10-6　オブジェクトリファレンスを使用したオペレーションの呼び出し（Java）

```
. . .
// Invoke the balance operation.
  System.out.println(
          "The balance in Account1: $" + account1.balance());
. . .
```

10.4　オブジェクトリファレンスの操作

静的_bind()メソッド（C++），または bind()メソッド（Java）は，CORBA オブジェクトのリファレンスをクライアントプログラムに返します。クライアントプログラムは，オブジェクトリファレンスを使用することで，IDL インタフェース定義で定義されたオブジェクトのオペレーションを呼び出せます。さらに，オブジェクトの操作に使用できる CORBA::Object クラス（C++），または org.omg.CORBA.Object クラス（Java）からすべての VisiBroker ORB オブジェクトを継承するメソッドもあります。

10.4.1　nil リファレンスのチェック（C++）

CORBA クラスのメソッドである次の is_nil()メソッドを使用して，オブジェクトリファレンスが nil であるかどうかを調べることができます。渡されたオブジェクトリファレンスが nil の場合，このメソッドは 1 を返します。オブジェクトリファレンスが nil でない場合，0 を返します。

コードサンプル 10-7　nil オブジェクトリファレンスをチェックするメソッド

```
class CORBA {
   . . .
   static Boolean is_nil(CORBA::Object_ptr obj);
   . . .
};
```

10.4.2　nil リファレンスの取得（C++）

CORBA::Object クラスの_nil()メソッドを使用して nil オブジェクトリファレンスを取得できます。この関数は，Object_ptr にキャストする NULL 値を返します。

コードサンプル 10-8　nil リファレンスを取得するメソッド

```
class Object {
   . . .
   static CORBA::Object_ptr _nil();
   . . .
};
```

10.4.3　オブジェクトリファレンスの複製（C++）

ユーザのクライアントプログラムが_duplicate()メソッドを起動すると，オブジェクトリファレンスのリファレンスカウントが一つずつ増え，同じオブジェクトリファレンスが返されます。クライアントプログラムは_duplicate()メソッドを使用して，オブジェクトリファレンスのリファレンスカウントを増やせるので，リファレンスはデータ構造に格納されるか，またはパラメタとして渡されます。リファレンスカウントが増えると，オブジェクトリファレンスに対応するメモリはリファレンスカウントがゼロになるまで解放されません。

IDL コンパイラは，指定の各オブジェクトインタフェースに対して_duplicate()メソッドを生成します。_duplicate()メソッドは，汎用 Object_ptr を受け付けて返します。

コードサンプル 10-9　オブジェクトリファレンスを二重化するメソッド

```
class Object {
   . . .
   static CORBA::Object_ptr _duplicate(CORBA::Object_ptr obj);
   . . .
};
```

注

POA や ORB のオブジェクトは，リファレンスカウントをサポートしていないため，_duplicate()メソッドを使用しても無効です。

10.4.4　オブジェクトリファレンスの解放（C++）

オブジェクトリファレンスが必要でなくなったら，オブジェクトリファレンスを解放する必要があります。オブジェクトリファレンスを解放する方法の一つとして，CORBA::Object クラスの_release()メソッドを起動する方法があります。

注

必ず_release()メソッドを使用してください。オブジェクトリファレンスに対して operator delete を使用しないでください。

コードサンプル 10-10　オブジェクトリファレンスの解放

```
class CORBA {
   class Object {
      . . .
      void _release();
      . . .
   };
};
```

また，CORBA クラスの release()メソッドを使用することもできます。この関数は CORBA との互換性を提供します。

コードサンプル 10-11　オブジェクトリファレンスを解放する CORBA メソッド

```
class CORBA {
   . . .
   static void release(Object_ptr);
   . . .
};
```

10.4.5　リファレンスカウントの取得（C++）

各オブジェクトリファレンスにはリファレンスカウントがあり，これを使用してリファレンスが何回複製されたかを調べることができます。_bind()メソッドを起動してオブジェクトリファレンスを最初に取得する場合，リファレンスカウントは 1 に設定されます。オブジェクトリファレンスを_release()メソッドで解放するたびに，リファレンスカウントを 1 ずつ減らします。リファレンスカウントが 0 になったら，Borland Enterprise Server VisiBroker はオブジェクトリファレンスを自動的に削除します。コードサンプル 10-12 にリファレンスカウントを返す_ref_count()メソッドを示します。

注

リモートクライアントがオブジェクトリファレンスを二重化したり解放したりする際，サーバのオブジェクトリファレンスカウントは影響を受けません。

コードサンプル 10-12　リファレンスカウントを取得するメソッド

```
class Object {
   . . .
   CORBA::Long _ref_count() const;
   . . .
};
```

10.4.6　リファレンスの文字列への変換

Borland Enterprise Server VisiBroker では，オブジェクトリファレンスを文字列に変換したり，文字列をオブジェクトリファレンスに戻したりするメソッドを VisiBroker ORB クラスで提供しています。CORBA の仕様では，オブジェクトリファレンスを文字列に変換するプロセスを文字列化，文字列をオブジェクトリファレンスに戻すプロセスを非文字列化と呼びます。

文字列化と非文字列化のメソッドを表 10-1 に示します。

表 10-1　文字列化と非文字列化のメソッド

メソッド	説明
object_to_string	オブジェクトリファレンスを文字列に変換します。
string_to_object	文字列をオブジェクトリファレンスに戻します。

クライアントプログラムは，object_to_string メソッドを使用してオブジェクトリファレンスを文字列に変換し，それをほかのクライアントプログラムに渡します。すると 2 番目のクライアントは，string_to_object メソッドを使用してオブジェクトリファレンスを非文字列化し，オブジェクトへの明示的なバインドを必要としないでオブジェクトリファレンスを使用できます。

注 1（C++の場合）

object_to_string の呼び出し元は，返された文字列を，CORBA::string_free()メソッドで解放する必要があります。

注 2

VisiBroker ORB または POA のようなローカルスコープのオブジェクトリファレンスは文字列化できません。文字列化しようとすると，マイナーコード 4 の MARSHAL 例外が発生します。

10.4.7　オブジェクト名とインタフェース名の取得

オブジェクト名とインタフェース名，およびオブジェクトリファレンスに対応するリポジトリ ID の取得に使用できる Object クラスが提供するメソッドを，表 10-2 に示します。インタフェースリポジトリについては，「16.　インタフェースリポジトリの使用」を参照してください。

注

静的_bind()メソッドの呼び出し時にオブジェクト名を指定しなかった場合，結果として生じたオブジェクトリファレンスを指定して_object_name()メソッドを呼び出すと NULL（C++）または null（Java）が返されます。

表 10-2　インタフェース名とオブジェクト名を取得するメソッド

メソッド	説明
_interface_name（C++の場合）	このオブジェクトのインタフェース名を返します。
_object_name	このオブジェクトの名前を返します。
_repository_id	リポジトリのタイプ識別子を返します。

10.4.8　オブジェクトリファレンスのタイプの判定

C++の場合

ユーザは_hash()メソッドを使用して，オブジェクトリファレンスのハッシュ値を取得できます。この値は一意であるとは限りませんが，オブジェクトリファレンスのライフスパンを通して一定の値を保ちます。そして，この値はハッシュテーブルに格納されます。

オブジェクトリファレンスが特定のタイプかどうかは，_is_a()メソッド（C++および Java）を使用してチェックできます。最初に，_repository_id()メソッド（C++および Java）を使用して，チェックしたいタイプのリポジトリ ID を取得する必要があります。このメソッドは，オブジェクトが_repository_id()メソッドで表されるタイプのインスタンスか，サブタイプであれば，1（C++）または true（Java）を返し

ます。オブジェクトが指定のタイプでなければ，0（C++）または false（Java）を返します。このとき，タイプを判定するためにリモート呼び出しが必要な場合があるので注意してください。

注（Java の場合）

instanceof キーワードはランタイムタイプの判定には使用できません。

二つのオブジェクトリファレンスが同じオブジェクトインプリメンテーションを参照するかどうかのチェックには，_is_equivalent()メソッド（C++および Java）を使用します。このメソッドは，これらのオブジェクトリファレンスが等しければ 1（C++）または true（Java）を返します。オブジェクトリファレンスが異なるなら，このメソッドは 0（C++）または false（Java）を返しますが，オブジェクトリファレンスが二つの異なったオブジェクトであるということを示すとは限りません。これはライトウェイトメソッドであり，サーバオブジェクトとの実際の通信は必要としません。オブジェクトリファレンスのタイプを判定するメソッドを表 10-3 に示します。

表 10–3　オブジェクトリファレンスのタイプを判定するメソッド

メソッド	説明
_hash（C++の場合）	オブジェクトリファレンスのハッシュ値を返します。
_is_a	指定されたインタフェースをオブジェクトがインプリメントするかどうかを判定します。
_is_equivalent	二つのオブジェクトが同じインタフェースインプリメンテーションを参照するなら，1（C++）または true（Java）を返します。

10.4.9　バインドされたオブジェクトの位置と状態の判定

オブジェクトリファレンスが有効であれば，クライアントプログラムは，_is_bound()メソッド（C++および Java）を使用してオブジェクトがバインドされているかどうかを判定できます。このメソッドは，オブジェクトがバインドされていれば 1（C++）または true（Java）を，バインドされていなければ 0（C++）または false（Java）を返します。

_is_local()メソッドは，クライアントプログラムとオブジェクトインプリメンテーションが同じプロセスまたはアドレス空間に常駐する場合に 1（C++）または true（Java）を返します。

_is_remote()メソッドは，クライアントプログラムとオブジェクトインプリメンテーションが同じホストにあるかどうかに関係なく，異なるプロセスに存在する場合に 1（C++）または true（Java）を返します。

オブジェクトリファレンスの位置と状態を判定するメソッドを表 10-4 に示します。

表 10–4　オブジェクトリファレンスの位置と状態を判定するメソッド

メソッド	説明
_is_bound	このオブジェクトに対してコネクションが現在アクティブなら，1（C++）または true（Java）を返します。
_is_local	このオブジェクトがローカルアドレス空間でインプリメントされたなら，1（C++）または true（Java）を返します。
_is_remote	このオブジェクトのインプリメンテーションがローカルアドレス空間になければ，1（C++）または true（Java）を返します。

注

メソッドが呼び出されたプロセスと同じプロセスにオブジェクトがある場合，_is_local()は 1（C++）または true（Java）を返します。

10.4.10　non_existent オブジェクトのチェック（C++）

_non_existent()メソッドを使用して，オブジェクトリファレンスに対応するオブジェクトインプリメンテーションがまだ存在するかどうかを判定できます。実際には，このメソッドはオブジェクトを ping して，オブジェクトがまだ存在するかどうかを判定し，存在すれば FALSE を返します。

10.4.11　オブジェクトリファレンスのナロウイング

オブジェクトリファレンスのタイプを一般的なスーパータイプから特定のサブタイプに変更するプロセスをナロウイングといいます。

注 1（C++の場合）

_narrow()メソッドは，新しい C++オブジェクトを構築し，そのオブジェクトのポインタを返します。オブジェクトがもう必要なければ，_narrow()メソッドで返されたオブジェクトリファレンスを解放する必要があります。

注 2（Java の場合）

ナロウイングに Java キャスティング機能は使用できません。

Borland Enterprise Server VisiBroker では，オブジェクトの narrow()メソッドを使用してナロウイングができるように，それぞれのオブジェクトインタフェースのタイプグラフを保持しています。

C++の場合

narrow メソッドが要求したタイプにオブジェクトをナロウできないと判定した場合，NULL を返します。

Java の場合

ナロウイングが失敗すると，IDL 例外である CORBA::BAD_PARAM 例外が返されます。それはオブジェクトリファレンスがリクエストされたタイプをサポートしていないためです。

コードサンプル 10-13　AccountManager 用に生成された narrow メソッド（C++）

```
Bank::AccountManager_ptr
Bank::AccountManager::_narrow(CORBA::Object * _obj) {
                    :
}
```

コードサンプル 10-14　AccountManager 用に生成された narrow メソッド（Java）

```
public abstract class AccountManagerHelper {
   . . .
   public static Bank.AccountManager narrow (
                               org.omg.CORBA.Object object){
      . . .
   }
   . . .
}
```

10.4.12　オブジェクトリファレンスのワイドニング

オブジェクトリファレンスのタイプをスーパータイプに変換することをワイドニングといいます。コードサンプル 10-15 および 10-16 では，C++および Java のそれぞれの Account ポインタの Object ポインタへのワイドニング例を示します。Account クラスは Object クラスを継承するので，ポインタ acct を Object ポインタとしてキャストできます。

コードサンプル 10-15　オブジェクトリファレンスのワイドニング（C++）

```
. . .
Account *acct;
CORBA::Object *obj;
acct = Account::_bind();
obj = (CORBA::Object *)acct; . . .
```

コードサンプル 10-16　オブジェクトリファレンスのワイドニング（Java）

```
. . .
Account account;
org.omg.CORBA.Object obj;
account = AccountHelper.bind();
obj = (org.omg.CORBA.Object)account;
. . .
```

10.5　Quality of Service の使用

QoS（Quality of Service）は，各ポリシーを利用してクライアントアプリケーションとそれに接続されているサーバとのコネクションの定義と管理を行います。

10.5.1　QoS の概要

QoS ポリシー管理は，次のオペレーションによって行われます。

- VisiBroker ORB レベルポリシーは局所に限定された PolicyManager によって処理され，この PolicyManager を介して，ポリシーを設定したり現在の Policy を見たりできます。VisiBroker ORB レベルで設定されたポリシーはシステムのデフォルトを変更します。
- スレッドレベルポリシーは PolicyCurrent を介して設定され，PolicyCurrent にはスレッドレベルでの Policy の変更の表示と設定を行うオペレーションが含まれます。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と VisiBroker ORB レベルで設定された値を変更します。
- ベースオブジェクトインタフェースの QoS オペレーションにアクセスすることによって，オブジェクトレベルポリシーが適用できます。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と VisiBroker ORB レベルまたはスレッドレベルで設定された値を変更します。

(1)　ポリシーの変更および有効ポリシー

有効ポリシーとは，適用できるすべてのポリシーの変更が完了し，最終的にリクエストに適用されるポリシーのことです。有効ポリシーは，有効な変更の内容と IOR に指定されたポリシーを比較することで決定します。有効ポリシーは，有効な変更の内容と IOR に指定された Policy との共通部分です。共通部分がない場合，org.omg.CORBA.INV_POLICY 例外が発生します。

10.5.2　QoS インタフェース

QoS ポリシーの取得と設定には，次のインタフェースを使用します。

(1)　CORBA::Object または org.omg.CORBA.Object

CORBA::Object（C++）および org.omg.CORBA.Object（Java）に含まれる次のメソッドは，有効ポリシーの取得やポリシーの変更の取得または設定に使用します。

- _get_policy は，オブジェクトリファレンスの有効ポリシーを返します。
- _set_policy_override は，オブジェクトレベルの，リクエストされたポリシーの変更のリストで新しいオブジェクトリファレンスを返します。

(2)　CORBA::Object または com.inprise.vbroker.CORBA.Object（Borland）

Java の場合

このインタフェースを使用するには，org.omg.CORBA.Object を com.inprise.vbroker.CORBA.Object にキャストする必要があります。このインタフェースは org.omg.CORBA.Object から派生するので，org.omg.CORBA.Object で定義されたメソッドに加えて次のメソッドが使用できます。

C++の場合

次のメソッドは，メソッド名の先頭の"_"がない名称で提供されています。

- _get_client_policy は，サーバ側ポリシーとの共通部分以外のオブジェクトリファレンスの有効 Policy を返します。有効な変更は，指定された変更をオブジェクトレベル，スレッドレベル，VisiBroker ORB レベルの順でチェックすることで取得されます。リクエストされた PolicyType の変更を指定していない場合，PolicyType のシステムデフォルト値が使用されます。
- _get_policy_overrides は，オブジェクトレベルで設定された指定ポリシータイプの Policy を変更するリストを返します。指定されたシーケンスが空の場合，オブジェクトレベルのすべての変更が返されます。オブジェクトレベルで変更された PolicyType がなければ，空シーケンスが返されます。
- _validate_connection は，オブジェクトに対して現在有効なポリシーが呼び出しを許可するかどうかに基づいて，boolean 値を返します。オブジェクトリファレンスがバインドされていなければ，バインディングが発生します。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，現在のポリシーの変更作業が変更された場合，またはバインディングがもう有効でない場合は，RebindPolicy の変更の設定には関係なくリバインドが試みられます。現在の有効ポリシーが INV_POLICY 例外を発生させる場合は，リターン値は false です。現在の有効ポリシーに不具合があれば，不具合なポリシーを記載した PolicyList タイプのシーケンスが返されます。

(3)　CORBA::PolicyManager または org.omg.CORBA.PolicyManager

PolicyManager は，VisiBroker ORB レベルの Policy の変更の取得と設定を行うメソッドを提供するインタフェースです。

- get_policy_overrides は，リクエストされた PolicyTypes の変更されたすべてのポリシーの PolicyList シーケンスを返します。指定されたシーケンスが空の場合，カレントコンテキストレベルのすべてについて Policy の変更が返されます。リクエストされた PolicyTypes が一つもターゲットの PolicyManager で変更されていなければ，空のシーケンスが返されます。
- set_policy_overrides は，リクエストされた Policy の変更のリストでカレントの変更作業を変更します。第 1 入力パラメタの policies は，Policy オブジェクトのリファレンスのシーケンスです。SetOverrideType 型（C++）または org.omg.CORBA.SetOverrideType 型（Java）の第 2 パラメタである set_add は，ADD_OVERRIDE を使用してこれらのポリシーを PolicyManager にすでに存在するほかの変更に追加するか，または SET_OVERRIDES を使用して，変更を含まない PolicyManager にこれらのポリシーを追加するかを示します。ポリシーの空シーケンスと SET_OVERRIDES モードを指定して set_policy_overrides を呼び出すと，すべての変更を PolicyManager から削除します。クライアントに適用できないポリシーを変更しようとすると，NO_PERMISSION 例外（C++）または org.omg.CORBA.NO_PERMISSION 例外（Java）が発生します。指定された PolicyManager が，リクエストが原因で不一致な状態になった場合には，ポリシーの変更や追加は行われないで，InvalidPolicies 例外が発生します。

(4)　CORBA::PolicyCurrent または org.omg.CORBA.PolicyCurrent

PolicyCurrent インタフェースは，新たなメソッドを追加しないで PolicyManager から派生します。このインタフェースは，スレッドレベルで変更されたポリシーへのアクセスを提供します。resolve_initial_references（C++）または org.omg.CORBA.ORB.resolve_initial_references（Java）で"PolicyCurrent"という識別子を指定して実行することでスレッドの PolicyCurrent のリファレンスを取得できます。

(5)　QoSExt::DeferBindPolicy または com.inprise.vbroker.QoSExt.DeferBindPolicy

DeferBindPolicy は，リモートオブジェクトが最初に作成されたときに VisiBroker ORB がそのオブジェクトとのコンタクトを試みるか，最初の呼び出しが行われるまでこのコンタクトを遅延させるかを決定します。DeferBindPolicy の値は true と false です。DeferBindPolicy を true に設定すると，バインディングインスタンスの最初の呼び出し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生成し，DeferBindPolicy を true に設定すると，最初の呼び出しまでサーバ起動を延期できます。以前このオプションは,生成されたヘルパークラスのバインドメソッドのオプションとして提供されていました。

コードサンプル 10-17 は，DeferBindPolicy（C++）を作成して，VisiBroker ORB 上にそのポリシーを設定する例を示します。コードサンプル 10-18 は，DeferBindPolicy（Java）を作成して，VisiBroker ORB 上にそのポリシー，スレッド，およびオブジェクトレベルを設定するサンプルを示します。

コードサンプル 10-17　DeferBindPolicy の作成（C++）

```
//Initialize the flag and references
 CORBA::Boolean deferMode = (CORBA::Boolean)1;
 CORBA::Any policy_value;
 policy_value << = CORBA::Any::from_boolean(deferMode);

 CORBA::Policy_var policy =
    orb->create_policy(QoSExt::DEFER_BIND_POLICY_TYPE,policy_value);

 CORBA::PolicyList policies;
 policies.length(1);
 policies [0] = CORBA::Policy::_duplicate(policy);

//Get a reference to the thread manager
 CORBA::Object_var obj =
   orb->resolve_initial_references("ORBPolicyManager");
 CORBA::PolicyManager_var orb_mgr =
   CORBA::PolicyManager::_narrow(obj);

//Set the policy on the ORB level
 orb_mgr->set_policy_overrides(policies,CORBA::SET_OVERRIDE);
```

コードサンプル 10-18　DeferBindPolicy の作成（Java）

次のサンプルでは DeferBindPolicy を作成して，VisiBroker ORB 上にそのポリシー，スレッド，およびオブジェクトレベルを設定します。

```
//Initialize the flag and the references
 boolean deferMode = true;
 Any policyValue = orb.create_any();
 policyValue.insert_boolean(deferMode);

 Policy policies =
 orb.create_policy(DEFER_BIND_POLICY_TYPE.value,policyValue);

//Get a reference to the thread manager
   PolicyManager orbManager =
     PolicyManagerHelper.narrow(
       orb.resolve_initial_references("ORBPolicyManager"));

//Set the policy on the ORB level
 orbManager.set_policy_overrides(new Policy[ ] {policies},
 SetOverrideType.SET_OVERRIDE);

//Get the binding method
  byte[ ] managerId ="BankManager".getBytes();
  Bank.AccountManager manager =
    Bank.AccountManagerHelper.bind(orb,"/qos_poa",managerId);
```

(6)　QoSExt::ExclusiveConnectionPolicy または com.inprise.vbroker.QoSExt.ExclusiveConnectionPolicy

ExclusiveConnectionPolicy は Borland Enterprise Server VisiBroker 固有のポリシーであり，指定のサーバオブジェクトへの排他（非共用）コネクションを設定できます。このポリシーに boolean 値（true または false）を割り当てます。ポリシーが false の場合，既存のコネクションが存在すれば，既存のコネクションを使用（共用）します。既存のコネクションが存在しない場合は，新しいコネクションを作成します。デフォルト値は false です。

このポリシーは VisiBroker 3.x の Object._clone()と同じ機能を提供します。

examples/vbe/QoS_policies/qos/CloneClient.java サンプルで，排他および非排他コネクションの設定方法の例を示します。

(7)　QoSExt::RelativeConnectionTimeoutPolicy または com.inprise.vbroker.QoSExt.RelativeConnectionTimeoutPolicy

RelativeConnectionTimeoutPolicy は，使用できる終端のどれかを使用してオブジェクトへの接続をリトライする場合のタイムアウトを指定します。タイムアウトの状態は，ファイアウォール（オブジェクトに接続する唯一の方法が HTTP トンネルである）で保護されたオブジェクトで発生しやすくなります。

コードサンプル 10-19　RelativeConnectionTimeoutPolicy の作成（Java）

次のコードは RelativeConnectionTimeoutPolicy の作成方法を示します。

```
Any connTimeoutPolicyValue =orb.create_any();

// Input is in 100s of Nanoseconds.
// To specify a value of 20 seconds,
// use 20 *10 ^7 nanoseconds as input

int connTimeout =20;

connTimeoutPolicyValue.insert_ulonglong(connTimeout *10000000);
org.omg.CORBA.Policy ctoPolicy =
      orb.create_policy(RELATIVE_CONN_TIMEOUT_POLICY_TYPE.value,
            connTimeoutPolicyValue);
PolicyManager orbManager =PolicyManagerHelper.narrow (
            orb.resolve_initial_references("ORBPolicyManager"));

orbManager.set_policy_overrides(new Policy[ ] ¥{ctoPolicy ¥},
       SetOverrideType.SET_OVERRIDE);
```

(8)　Messaging::RebindPolicy または org.omg.Messaging.RebindPolicy

RebindPolicy は，リバインディング時にクライアントの動作を定義するために Messaging::RebindMode 型の値（C++）または org.omg.Messaging.RebindMode 型の値（Java）を読み込みます。RebindPolicy はクライアント側だけに設定されます。RebindPolicy には，コネクション切断，オブジェクト転送リクエスト，オブジェクト障害などの場合に動作を決定する六つの値のうち一つを指定できます。現在サポートされている値は，次のとおりです。

- Messaging::TRANSPARENT（C++）または org.omg.Messaging.TRANSPARENT（Java）は，リモートリクエスト時に，VisiBroker ORB がオブジェクト転送および必要なリコネクションを透過的に処理します。コードサンプル 10-20 では，TRANSPARENT 型の RebindPolicy（Java）を作成して，VisiBroker ORB でそのポリシー，スレッド，およびオブジェクトレベルを設定します。
- Messaging::NO_REBIND（C++）または org.omg.Messaging.NO_REBIND（Java）は，リモートリクエスト時に，クローズしたコネクションの再オープンを VisiBroker ORB が透過的に処理できるようにしますが，クライアントが想定している有効 QoS ポリシーの変更の原因となる透過的なオブジェクト転送はしません。RebindMode を NO_REBIND に設定すると，明示的なリバインドだけが許可されます。
- Messaging::NO_RECONNECT（C++）または org.omg.Messaging.NO_RECONNECT（Java）は，VisiBroker ORB がオブジェクト転送やクローズしたコネクションの再オープンを透過的に処理しないようにします。RebindMode を NO_RECONNECT に設定した場合は，明示的にリバインドとリコネクトをする必要があります。
- QoSExt::VB_TRANSPARENT（C++）または com.inprise.vbroker.QoSExt.VB_TRANSPARENT はデフォルトポリシーです。これは，明示的なバインディングと暗黙的なバインディングの双方による透過的バインディングを許可することで，TRANSPARENT の機能を継承します。

VB_TRANSPARENT は，VisiBroker 3.x のオブジェクトフェールオーバーインプリメンテーションとの互換性を持つように設計されています。

- QoSExt::VB_NOTIFY_REBIND（C++）または com.inprise.vbroker.QoSExt.VB_NOTIFY_REBIND（Java）は，リバインドが必要な場合に例外を発生させます。クライアントはこの例外をキャッチして，2 回目の呼び出しでバインドします。
- QoSExt::VB_NO_REBIND（C++）または com.inprise.vbroker.QoSExt.VB_NO_REBIND（Java）は，オブジェクト障害後 osagent を使用してほかのオブジェクト呼び出しを有効にしません。これは，クライアント VisiBroker ORB が同じサーバへのクローズしたコネクションを再びオープンすることだけができるようになります。オブジェクトフォワーディングはできません。

注 1

クライアントの有効ポリシーが VB_TRANSPARENT で，かつクライアントが状態データを保持しているサーバとともに動作中の場合，VB_TRANSPARENT は，クライアントにサーバの変更を気づかせることなくクライアントを新しいサーバに接続できます。元のサーバが保持していた状態データは失われます。

注 2（C++の場合）

NO_REBIND または NO_RECONNECT の場合，CORBA::Object クラスの_validate_connection を呼び出すことによって，クローズしたコネクションの再オープンや転送が明示的に許可されることがあります。

表 10-5 に，異なる RebindMode 型の動作を示します。

表 10–5　RebindMode ポリシー

RebindMode 型	同じオブジェクトへのクローズしたコネクションの再確立	オブジェクト転送の可否	オブジェクトフェールオーバーの可否※
NO_RECONNECT	行わないで，REBIND 例外を発生させます。	不可。REBIND 例外を発生させます。	不可。
NO_REBIND	行います。	ポリシーが一致すれば可。 それ以外は，不可。REBIND 例外を発生させます。	不可。
TRANSPARENT	行います。	可。	不可。
VB_NO_REBIND	行います。	不可。行わないで，例外を発生させます。	不可。行わないで，例外を発生させます。
VB_NOTIFY_REBIND	行います。	可。	可。障害検出後，VB_NOTIFY_REBIND は例外を発生させてから，以降のリクエストで osagent を使用してほかのオブジェクト呼び出しを試みます。

RebindMode 型	同じオブジェクトへのクローズしたコネクションの再確立	オブジェクト転送の可否	オブジェクトフェールオーバーの可否※
VB_TRANSPARENT	行います。	可。	透過的に可。

注※　通信障害またはオブジェクト障害でフェールオーバーできない場合は，適切な CORBA 例外が発生します。

コードサンプル 10-20　TRANSPARENT 型の RebindPolicy の作成

次のサンプルでは，TRANSPARENT 型の RebindPolicy を作成して，VisiBroker ORB でそのポリシー，スレッド，およびオブジェクトレベルを設定します。

```
Any policyValue=orb.create_any();
RebindModeHelper.insert(policyValue,
             org.omg.Messaging.TRANSPARENT.value);
Policy myRebindPolicy =orb.create_policy(
                REBIND_POLICY_TYPE.value,policyValue);

//get a reference to the ORB policy manager
org.omg.CORBA.PolicyManager manager;

try {
  manager =
      PolicyManagerHelper.narrow(orb.resolve_initial_references
                        ("ORBPolicyManager"));
}
catch(org.omg.CORBA.ORBPackage.InvalidName e){}

//get a reference to the per-thread manager
org.omg.CORBA.PolicyManager current;
try {
   current=PolicyManagerHelper.narrow(orb.resolve_initial_references
                        ("PolicyCurrent"));
}
catch(org.omg.CORBA.ORBPackage.InvalidName e){}

//set the policy on the orb level
try{
   manager.set_policy_overrides(myRebindPolicy,
                                    SetOverrideType.SET_OVERRIDE);
}
catch (InvalidPolicies e){}

//set the policy on the Thread level
try{
   current.set_policy_overrides(myRebindPolicy,
                                    SetOverrideType.SET_OVERRIDE);
}
catch (InvalidPolicies e){}

//set the policy on the object level:
org.omg.CORBA.Object oldObjectReference=bind(...);
org.omg.CORBA.Object newObjectReference=oldObjectReference._set_policy_override
         (myRebindPolicy,SetOverrideType.SET_OVERRIDE);
```

QoS ポリシーと型の詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「QoS インタフェースとクラス」の記述，および CORBA 2.5 仕様のメッセージングについての記述を参照してください。

(9)　Messaging::RelativeRequestTimeoutPolicy または org.omg.CORBA.Messaging.RelativeRequestTimeoutPolicy

RelativeRequestTimeoutPolicy は，Request またはその応答 Reply が渡される相対的な時間を示します。この時間が過ぎると，Request 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呼

び出しに適用されます。リクエストは指定のタイムアウト時間内に完了するものと仮定するので，Reply 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 ナノ秒単位で指定されます。

コードサンプル 10-21　RelativeRequestTimeoutPolicy の作成

次のサンプルでは，RelativeRequestTimeoutPolicy の作成方法を示します。

```
//Specify the timeout in 100s of Nanoseconds
//To set a timeout of 20 secs,set 20 *10 ^7 nanoseconds

int reqTimeout =20;
RelativeRequestTimeoutPolicyImpl relReq =
       new RelativeRequestTimeoutPolicyImpl(reqTimeout *10000000);
 Any policyValue =orb.create_any();¥par
 RelativeRequestTimeoutPolicyHelper.insert(policyValue,
                  (RelativeRequestTimeoutPolicy)relReq);

//set the RelativeRequestTimeoutPolicy
org.omg.CORBA.Policy reqPolicy =orb.create_policy(
        RELATIVE_REQ_TIMEOUT_POLICY_TYPE.value,policyValue);
 PolicyManager orbManager =PolicyManagerHelper.narrow(
                  orb.resolve_initial_references("ORBPolicyManager"))
orbManager.set_policy_overrides(new Policy[ ]
              SetOverrideType.SET_OVERRIDE);
```

(10)　Messaging::RelativeRoundTripTimeoutPolicy または org.omg.CORBA.Messaging.RelativeRoundtripTimeoutPolicy

RelativeRoundtripTimeoutPolicy は，Request またはその該当する Reply が渡される相対的な時間を指定します。このタイムアウトが過ぎても応答が渡されなかった場合，Request はキャンセルされます。また，Request がすでに渡されており，Reply がターゲットから返されると，Reply はこの時間が過ぎたら破棄されます。このポリシーは同期および非同期の両方の呼び出しに適用されます。タイムアウト値は 100 ナノ秒単位で指定します。

コードサンプル 10-22　RelativeRoundTripTimeoutPolicy の作成

次のサンプルでは，RelativeRoundTripTimeoutPolicy の作成方法を示します。

```
//Specify the timeout in 100s of Nanoseconds
//To set a timeout of 90 secs,set 90 *10 ^7 nanosecs

int rttTimeout =20;
RelativeRoundtripTimeoutPolicyImpl relRtt =new
        RelativeRoundtripTimeoutPolicyImpl(rttTimeout *10000000);
Any policyValue =orb.create_any();
          RelativeRoundtripTimeoutPolicyHelper.insert(policyValue,
                   (RelativeRoundtripTimeoutPolicy)relRtt);

//set the RelativeRoundtripTimeoutPolicy

org.omg.CORBA.Policy rttPolicy =orb.create_policy(
          RELATIVE_RT_TIMEOUT_POLICY_TYPE.value,policyValue);
PolicyManager orbManager =PolicyManagerHelper.narrow(
          orb.resolve_initial_references("ORBPolicyManager"));
orbManager.set_policy_overrides(new Policy[ ]
            SetOverrideType.SET_OVERRIDE);
```

(11)　Messaging::SyncScopePolicy または org.omg.CORBA.Messaging.SyncScopePolicy

SyncScopePolicy は，ターゲットに関するリクエストの同期レベルを定義します。SyncScope 型の値は，一方向オペレーションの動作を制御するために，SyncScopePolicy とともに使用されます。

SyncScopePolicy のデフォルトは，SYNC_WITH_TRANSPORT です。

注

アプリケーションは，VisiBroker ORB インプリメンテーションのポータビリティを確保するために，明示的に VisiBroker ORB レベルの SyncScopePolicy を設定する必要があります。SyncScopePolicy のインスタンスが作成される場合，Messaging::SyncScope 型の値は CORBA::ORB::create_policy に渡されます。このポリシーは，クライアント側の変更だけ適用できます。

10.5.3　QoS 例外

- CORBA::INV_POLICY（C++）または org.omg.CORBA.INV_POLICY（Java）は，Policy の変更の間に不具合があると発生します。
- CORBA::REBIND（C++）または org.omg.CORBA.REBIND（Java）は，RebindPolicy の値が NO_REBIND，NO_RECONNECT，VB_NOTIFY_REBIND のどれかであり，バインドされたオブジェクトリファレンスの呼び出しの結果としてオブジェクトフォワードまたはロケーションフォワードメッセージが出力されると発生します。
- CORBA::PolicyError（C++）または org.omg.CORBA. PolicyError（Java）は，リクエストされた Policy がサポートされていない場合に発生します。
- CORBA::InvalidPolicies（C++）または org.omg.CORBA.InvalidPolicies（Java）は，オペレーションを PolicyList シーケンスに渡すと発生します。例外本体に含まれるのは，ポリシーがカレントのスコープ内ですでに変更されているためか，またはリクエストされたほかのポリシーと一緒であると有効にならないために有効ではないシーケンスのポリシーです。

11 IDL の使用

この章では，CORBA の IDL（インタフェース記述言語）の使用方法について説明します。

11.1 IDL とは

IDL はリモートオブジェクトがインプリメントしているインタフェースを記述する記述言語です（プログラム言語ではありません）。IDL 内では，インタフェースの名称やそれぞれの属性名とメソッド名などを定義できます。IDL ファイルを生成すれば，IDL コンパイラを使用してスタブファイルやサーバスケルトンファイルを C++または Java プログラム言語で生成できます。

OMG はこのような言語マッピングの仕様を規定しています。Borland Enterprise Server VisiBroker は OMG が提案した仕様に準拠するので，言語マッピングについての情報はこのマニュアルには記載していません。言語マッピングの詳細については，OMG Web サイトにアクセスしてください。

11.2　IDL コンパイラのコード生成方法

IDL を使用して，クライアントプログラムが使用するオブジェクトインタフェースを定義してください。idl2cpp および idl2java コンパイラは，このインタフェース定義を使用してコードを生成します。

idl2cpp および idl2java コンパイラの構文の詳しい使い方については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「idl2cpp」および「idl2java」の記述を参照してください。

11.2.1　IDL の指定例

インタフェース定義は，オブジェクトの名前と，オブジェクトが提供するすべてのメソッドの名前を定義します。各メソッドには，メソッドに渡すパラメタ，パラメタの型，およびパラメタが入力用，出力用，入出力用のどれなのかを指定します。IDL サンプル 11-1 は，example という名前のオブジェクトの IDL 仕様を示しています。example オブジェクトは，op1 という一つのメソッドだけを所有しています。

IDL サンプル 11-1　IDL 指定の例

```
// IDL specification for the example object
interface example {
   long op1 (in char x, out short y);
};
```

%TPDIR%¥idl ディレクトリの orb.idl や，CosTransactions.idl など orb.idl をインクルードする IDL ファイルを idl2java または idl2cpp でコンパイルした場合に次の警告メッセージが出力されます。この警告メッセージはトランザクション処理などの実行時には影響を与えません。

```
(warning)::CORBA::Environment declared (”orb.idl”, line 28) but never defined
(warning)(references to it will be permitted, but no code will be generated for this
definition)
```

11.3　生成されたコードの考察

IDL コンパイラは前述の IDL サンプルから幾つかのファイルを生成します。

- _exampleStub.java はクライアント側の example オブジェクトのスタブコードです。
- example.java は example インタフェース宣言です。
- exampleHelper.java は，example インタフェースのユーティリティ機能とサポート機能を定義する exampleHelper クラスを宣言します。
- exampleHolder.java は，out パラメタと inout パラメタを渡すためのホルダを提供する exampleHolder クラスを宣言します。
- exampleOperations.java は example インタフェースでメソッドを定義し，クライアント側とサーバ側の両方で使用されます。また，tie 機能を提供するために tie クラスと一緒に動作します。
- examplePOA.java は，サーバ側の example オブジェクトのスケルトンコード（インプリメンテーションベースコード）を格納します。
- examplePOATie.java は，tie 機能を使用してサーバ側の example オブジェクトをインプリメントする場合に使用するクラスを格納します。

11.3.1　_<interface_name>Stub.java

idl2java コンパイラによって，ユーザ定義型ごとにスタブクラスが生成されます。これは，<interface_name>インタフェースをインプリメントするクライアント側で実体化されるクラスです。

コードサンプル 11-1　スタブクラスコードの例（Java）

```
public class exampleStub extends
                com.inprise.vbroker.CORBA.portable.ObjectImpl
        implements example {
   final public static java.lang.Class _opsClass =
                                    exampleOperations.class;
   public java.lang.String[ ] ids (){
      . . .
   }
   public int op1 (char x, org.omg.CORBA.ShortHolder y){
      . . .
}
```

11.3.2　<interface_name>.java

<interface_name>.java ファイルは，各 IDL インタフェース用に生成された Java インタフェースです。これは IDL インタフェース定義を，適切な Java インタフェースにマッピングするためのものです。このインタフェースは，クライアントとサーバスケルトンの両方によってインプリメントされます。

コードサンプル 11-2　インタフェース宣言コードの例（Java）

```
public interface example extends
                          com.inprise.vbroker.CORBA.Object,
      exampleOperations,
      org.omg.CORBA.portable.IDLEntity {
}
```

11.3.3　<interface_name>Helper.java

idl2java によって，ユーザ定義型ごとにヘルパークラスが生成されます。ヘルパークラスは，生成された Java インタフェースのさまざまな静的メソッドを持つ final クラスです。

コードサンプル 11-3　ヘルパークラスコードの例（Java）

```
public final class exampleHelper {
   public static example narrow (
                            final org.omg.CORBA.Object obj){
      . . .
   }
   public static example unchecked_narrow (
                                  org.omg.CORBA.Object obj){
      . . .
   }
   public static example bind (org.omg.CORBA.ORB orb){
   . . .
   }
   public static example bind (
               org.omg.CORBA.ORB orb, java.lang.String name){
   . . .
   }
   public static example bind (
         org.omg.CORBA.ORB orb, java.lang.String name,
         java.lang.String host,
         com.inprise.vbroker.CORBA.BindOptions _options){
      . . .
   }
   public static example bind (
         org.omg.CORBA.ORB orb, java.lang.String fullPoaName,
         byte[ ] oid){
      . . .
   }
   public static example bind (org.omg.CORBA.ORB orb,
         java.lang.String fullPoaName, byte[ ] oid,
         java.lang.String host,
         com.inprise.vbroker.CORBA.BindOptions _options){
      . . .
   }
   public java.lang.Object read_Object (
          final org.omg.CORBA.portable.InputStream istream){
      . . .
   }
   public void write_Object (
          final org.omg.CORBA.portable.OutputStream ostream,
          final java.lang.Object obj){
      . . .
   }
   public java.lang.String get_id (){
      . . .
   }
   public org.omg.CORBA.TypeCode get_type (){
      . . .
   }
   public static example read (
           final org.omg.CORBA.portable.InputStream _input){
      . . .
   }
   public static void write (
          final org.omg.CORBA.portable.OutputStream _output,
   final example value){
      . . .
   }
   public static void insert (
           final org.omg.CORBA.Any any, final example value){
      . . .
   }
   public static example extract (final org.omg.CORBA.Any any){
      . . .
   }
   public static org.omg.CORBA.TypeCode type (){
      . . .
   }
   public static java.lang.String id (){
      . . .
   }
}
```

11.3.4　<interface_name>Holder.java

idl2java コンパイラによって，ユーザ定義型ごとにホルダークラスが生成されます。これは，out パラメタと inout パラメタとして渡される場合，<interface_name>インタフェースをサポートするオブジェクトをラッピングするオブジェクトのクラスを提供します。

コードサンプル 11-4　ホルダークラスの例（Java）

```
public final class exampleHolder implements
                        org.omg.CORBA.portable.Streamable {
   public foo.example value;
   public exampleHolder (){
   }
   public exampleHolder (final foo.example _vis_value){
      . . .
   }
   public void _read (
            final org.omg.CORBA.portable.InputStream input){
      . . .
   }
   public void _write (
          final org.omg.CORBA.portable.OutputStream output){
      . . .
   }
   public org.omg.CORBA.TypeCode _type (){
      . . .
   }
}
```

11.3.5　<interface_name>Operations.java

IDL 宣言で定義されたすべてのメソッドを含む idl2java コンパイラによって，ユーザ定義型ごとにオペレーションクラスが生成されます。

コードサンプル 11-5　オペレーションコードの例（Java）

```
public interface exampleOperations {
   public int op1 (char x, org.omg.CORBA.ShortHolder y);
}
```

11.3.6　<interface_name>POA.java

<interface_name>POA.java ファイルはインタフェースのサーバ側のスケルトンです。このファイルは in パラメタをアンマーシャルしてからオブジェクトインプリメンテーションに渡し，リターン値と（あれば）out パラメタをマーシャルし直します。

コードサンプル 11-6　ExamplePOA.java ファイル（Java）

```
public abstract class examplePOA extends
                    org.omg.PortableServer.Servant implements
    org.omg.CORBA.portable.InvokeHandler, exampleOperations {
   public example _this (){
   . . .
   }
   public example _this (org.omg.CORBA.ORB orb){
   . . .
   }
   public java.lang.String[ ] _all_interfaces (
                       final org.omg.PortableServer.POA poa,
      . . .
   }
   public org.omg.CORBA.portable.OutputStream _invoke (
        java.lang.String opName,
        org.omg.CORBA.portable.InputStream _input,
        org.omg.CORBA.portable.ResponseHandler handler){
      . . .
```

```
    }
    public static org.omg.CORBA.portable.OutputStream _invoke (
       exampleOperations _self,
       int _method_id, org.omg.CORBA.portable.InputStream _input,
       org.omg.CORBA.portable.ResponseHandler _handler){
       . . .
    }
}
```

11.3.7　<interface_name>POATie.java

<interface_name>POATie.java ファイルは，<interface_name>インタフェースのデリゲータインプリメンテーションです。すべてのオペレーションのデリゲート先である<interface_name>Operations クラスをインプリメントするインプリメンテーションクラスのインスタンスで，tie クラスの各インスタンスを初期化する必要があります。

コードサンプル 11-7　Example POATie ファイル（Java）

```
public class examplePOATie extends examplePOA {
   public examplePOATie (final exampleOperations _delegate){
      . . .
   }
   public examplePOATie (final exampleOperations _delegate,
   final org.omg.PortableServer.POA _poa){
      . . .
   }
   public exampleOperations _delegate (){
      . . .
   }
   public void _delegate (final exampleOperations delegate){
      . . .
   }
   public org.omg.PortableServer.POA _default_POA (){
      . . .
   }
   public int op1 (char x, org.omg.CORBA.ShortHolder y){
      . . .
   }
}
```

11.3.8　クライアント用に生成されたコードの考察（C++）

コードサンプル 11-8 に，IDL コンパイラ（IDL サンプル 11-1）が二つのクライアントファイル example_c.hh と example_c.cc をどのように生成するかを示します。この二つのファイルはクライアントが使用する example クラスを提供します。規則で IDL コンパイラが生成するファイルには必ず.cc または.hh という拡張子が付けられ，ユーザが自分で作成したファイルと区別できるようになっています。ファイルに別の拡張子を付けたければファイル生成の規則を変えることもできます。規則を変える方法については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「idl2cpp」の記述を参照してください。

注

IDL コンパイラが生成したファイルの内容は変更しないでください。

コードサンプル 11-8　example_c.hh 生成ファイルの生成クラス例（C++）

```
class example : public virtual CORBA_Object {
   protected:
      example() {}
      example(const example&) {}
public:
   virtual ~example() {}
   static  const CORBA::TypeInfo *_desc();
   virtual const CORBA::TypeInfo *_type_info() const;
   virtual void *_safe_narrow(const CORBA::TypeInfo& ) const;
   static CORBA::Object*_factory();
```

```
    example_ptr _this();
    static example_ptr _duplicate(example_ptr _obj)
                { /* . . . */ }
    static example_ptr _nil() { /* . . . */ }
    static example_ptr _narrow(CORBA::Object* _obj);
    static example_ptr _clone(example_ptr _obj)
                { /* . . . */ }
    static example_ptr _bind(
       const char *_object_name = NULL,
       const char *_host_name = NULL,
       const CORBA::BindOptions* _opt = NULL,
       CORBA::ORB_ptr _orb = NULL);
    static example_ptr _bind(
       const char *_poa_name,
       const CORBA::OctetSequence& _id,
       const char *_host_name = NULL,
       const CORBA::BindOptions* _opt = NULL,
       CORBA::ORB_ptr _orb = NULL);
    virtual CORBA::Long op1(CORBA::Char _x, CORBA::Short_out _y);
};
```

11.3.9　IDL コンパイラが生成するメソッド（スタブ）

コードサンプル 11-8 では IDL コンパイラが生成する op1 メソッドを，ほかの幾つかのメソッドと一緒に示しました。op1 メソッドはスタブとも呼ばれます。それはクライアントプログラムがこのメソッドを起動すると，インタフェースリクエストと引数を実際にメッセージにパッケージ化し，そのメッセージをオブジェクトインプリメンテーションに送信し，応答を待ち，その応答をデコードし，結果をユーザのプログラムに返すためです。

example クラスは，CORBA::Object クラスから派生するので使用できる継承されたメソッドが幾つかあります。

11.3.10　ポインタタイプ<interface_name>_ptr 定義

IDL コンパイラは，常にポインタタイプ定義を提供します。コードサンプル 11-9 に example クラスのタイプ定義を示します。

コードサンプル 11-9　example_c.hh 生成ファイルの_ptr タイプ（C++）

```
typedef example *example_ptr;
```

11.3.11　自動メモリ管理<interface_name>_var クラス

IDL コンパイラは, example_var という名前のクラスも生成します。このクラスは example_ptr の代わりに使用できます。example_var クラスは動的に割り当てられたオブジェクトリファレンスに対応するメモリを自動的に管理します。example_var オブジェクトが削除されると，example_ptr に対応するオブジェクトが解放されます。example_var オブジェクトが新しい値を割り当てられると，example_ptr によってポイントされた古いオブジェクトリファレンスは, 割り当て後に解放されます。キャスティング演算子も提供されており，これによって example_var をタイプ example_ptr に割り当てることができます。

コードサンプル 11-10　example_c.hh 生成ファイルの example_var クラス（C++）

```
class example_var : public CORBA::_var {
   . . .
   public:
      static example_ptr _duplicate(example_ptr);
      static void _release(example_ptr);
      example_var();
      example_var(example_ptr);
      example_var(const example_var &);
      ~example_var();
      example_var& operator=(example_ptr);
```

```
        example_var& operator=(const example_var& _var)
                                { /* . . . */ }
        operator example* () const { return _ptr; }
        . . .
};
```

表 11-1 に，_var クラスのメソッドを説明します。

表 11-1　_var クラスのメソッド

メソッド	説明
example_var()	_ptr を NULL に初期化するコンストラクタ。
example_var (example_ptr ptr)	渡された引数に_ptr を初期化したオブジェクトを作成するコンストラクタ。var はデストラクト時に_ptr で release()メソッドを起動します。_ptr のリファレンスカウントが 0 になると，そのオブジェクトは削除されます。
example_var(const example_var& var)	パラメタ var として渡されたオブジェクトのコピーを作成し，_ptr を新しくコピーされたオブジェクトにポイントするコンストラクタ。
~example()	_ptr がポイントするオブジェクトで_release()メソッドを一度起動するデストラクタ。
operator=(example_ptr p)	_ptr がポイントするオブジェクトで_release()メソッドを起動し，p を_ptr に格納する代入演算子。
operator=(const example_ptr p)	_ptr がポイントするオブジェクトで_release()メソッドを起動し，p の_duplicate()メソッドを_ptr に格納する代入演算子。
example_ptr operator->()	該当するオブジェクトに格納された_ptr を返します。この演算子は，該当するオブジェクトが正しく初期化されるまで呼び出してはいけません。

11.4　サーバ用に生成されたコードの考察（C++）

コードサンプル 11-11 に，IDL コンパイラが example_s.hh と example_s.cc という二つのサーバファイルを生成する方法を示します。この二つのファイルは POA_example クラスを提供します。
POA_example クラスは，インプリメンテーションクラスを派生するためにサーバが使用します。
POA_example クラスは，PortableServer_ServantBase クラスから派生します。

注

IDL コンパイラが生成したファイルの内容は変更しないでください。

コードサンプル 11-11　example_s.hh 生成ファイルのクラス例

```
class POA_example : public virtual PortableServer_ServantBase {
   protected:
      POA_example(){}
      virtual ~POA_example(){}
public:
   static const CORBA::TypeInfo _skel_info;
   virtual const CORBA::TypeInfo *_type_info() const;
   example_ptr _this();
   virtual void *_safe_narrow(const CORBA::TypeInfo& ) const;
   static POA_example * _narrow(
         PortableServer_ServantBase *_obj);
   // The following operations need to be implemented
   virtual CORBA::Long op1(
         CORBA::Char _x, CORBA::Short_out _y) = 0;
   // Skeleton Operations implemented automatically
   static void _op1(void *_obj, CORBA::MarshalInBuffer &_istrm,
   const char *_oper, VISReplyHandler& handler);
};
```

11.4.1　IDL コンパイラが生成するメソッド（スケルトン）

IDL サンプル 11-1 に示す IDL 仕様で宣言された op1 メソッドは，_op1 メソッドで生成されることに注意してください。POA_example クラスは，op1 という名前の純仮想メソッドを宣言します。
POA_example から派生したインプリメンテーションクラスは，このメソッドのインプリメンテーションを提供する必要があります。

POA_example クラスはスケルトンと呼ばれ，そのメソッド（_op1）はクライアントリクエストの受信時に POA が起動します。スケルトンの内部メソッドはリクエストのすべてのパラメタをマーシャルし，ユーザの op1 メソッドを起動してから，返されるパラメタまたは例外を応答メッセージにマーシャルします。ORB は，その応答をクライアントプログラムに送信します。

コンストラクタとデストラクタは，両方とも保護されており，継承されたメンバだけが起動できます。コンストラクタはオブジェクト名を受け付けるので，サーバは複数の異なるオブジェクトを実体化できます。

11.4.2　IDL コンパイラが生成するクラステンプレート

POA_example クラスに加えて，IDL コンパイラは_tie_example という名前のクラステンプレートを生成します。このテンプレートは，クラスを POA_example から派生させたくない場合に使用します。テンプレートは，新しいクラスを継承するために変更できない既存のアプリケーションのオブジェクトラッパークラスを提供する場合に役立ちます。コードサンプル 11-12 に，example クラス用に IDL コンパイラが生成したテンプレートクラスを示します。

コードサンプル 11-12　example クラス用に生成したテンプレートクラス（C++）

```
template <class T>
class POA_example_tie : public POA_example {
   public:
```

```
        POA_example_tie (T& t): _ptr(&t), _poa(NULL),
                                _rel((CORBA::Boolean)0) {}
        POA_example_tie (T& t, PortableServer::POA_ptr poa):
                                _ptr(&t),
           _poa(PortableServer::_duplicate(poa)),
                                _rel((CORBA::Boolean)0) {}
        POA_example_tie (T *p, CORBA::Boolean release= 1)
           : _ptr(p), _poa(NULL), _rel(release) {}
        POA_example_tie (T *p, PortableServer::POA_ptr poa,
                                CORBA::Boolean release =1)
           : _ptr(p), _poa(PortableServer::_duplicate(poa)),
                                _rel(release) {}
        virtual ~POA_example_tie() { /* . . . */ }
        T* _tied_object() { /* . . . */ }
        void _tied_object(T& t) { /* . . . */ }
        void _tied_object(T *p, CORBA::Boolean release=1)
                                { /* . . . */ }
        CORBA::Boolean _is_owner() { /* . . . */ }
        void _is_owner(CORBA::Boolean b) { /* . . . */ }
        CORBA::Long op1(CORBA::Char _x, CORBA::Short_out _y)
                                { /* . . . */ }
        PortableServer::POA_ptr _default_POA() { /* . . . */ }
};
```

_tie テンプレートクラスの使用については，「9.　tie 機能の使用」を参照してください。

オブジェクトデータベースとユーザのサーバを統合するために，_ptie テンプレートを生成することもできます。

11.5　IDL のインタフェース属性の定義

インタフェース定義では，オペレーションに加えて，インタフェースの一部分として属性を定義できます。デフォルトでは，すべての属性は read-write であり，IDL コンパイラは属性の値を設定するメソッドと属性の値を取得するメソッドの二つのメソッドを生成します。また，read-only 属性も指定できますが，この場合は読み込みメソッドだけが生成されます。

IDL サンプル 11-2 は，read-write 属性と read-only 属性という二つの属性を定義する IDL 指定を示しています。コードサンプル 11-13 は，IDL で宣言されたインタフェース用に生成されたオペレーションクラスを示しています。

IDL サンプル 11-2　read-write と read-only という二つの属性を持つ IDL 指定

```
interface Test {
   attribute long count;
   readonly attribute string name;
};
```

コードサンプル 11-13　Test インタフェース用に生成されたコード（C++）

```
class Test : public virtual CORBA::Object {
      . . .
      // Methods for read-write attribute
      virtual CORBA::Long count();
      virtual void count(CORBA::Long __count);

      // Method for read-only attribute.
      virtual char * name();
      . . .
};
```

コードサンプル 11-14　TestOperations インタフェース用に生成されたコード（Java）

```
public interface TestOperations {
   // Methods for read-write attribute
   public int count ();
   public void count (int count);

   // Method for read-only attribute.
   public java.lang.String name ();
}
```

11.6　リターン値を持たない oneway メソッドの指定

IDL では，リターン値を持たない oneway（一方向）メソッドと呼ばれるオペレーションを指定できます。このオペレーションには入力パラメタしかありません。oneway メソッドが呼び出されると，リクエストはサーバに送信されますが，このリクエストが実際に受信されたことを示すオブジェクトインプリメンテーションからの応答はありません。Borland Enterprise Server VisiBroker は TCP/IP を使用してクライアントをサーバに接続します。これによってすべてのパケットの配信が保証され，サーバが使用可能であるかぎり，クライアントはリクエストがサーバに届いたことを確信できます。それでもクライアントには，リクエストが実際にオブジェクトインプリメンテーション自体によって処理されたことを知る方法はありません。

注

一方向オペレーションは，例外またはリターン値を発生させることはできません。

IDL サンプル 11-3　一方向オペレーションの定義

```
interface oneway_example {
   oneway void set_value(in long val);
};
```

11.7　別のインタフェースを継承するインタフェースの IDL での指定

IDL では，別のインタフェースを継承するインタフェースを指定できます。IDL コンパイラによって生成されるクラスは，この継承関係を反映します。親インタフェースが宣言したすべてのメソッド，データ型定義，定数，および列挙体は，派生インタフェースからも参照できます。

IDL サンプル 11-4　インタフェース定義での継承の例

```
interface parent {
   void operation1();
};
interface child : parent {
   . . .
   long operation2(in short s);
};
```

コードサンプル 11-15，11-16 は IDL サンプル 11-4 に示すインタフェース定義から生成された C++コードおよび Java コードを示しています。

コードサンプル 11-15　IDL サンプル 11-4 から生成されたコード（C++）

```
. . .
class parent : public virtual CORBA::Object {
   . . .
   void operation1();
   . . .
};

class child : public virtual parent {
   . . .
   CORBA::Long operation2(CORBA::Short s);
   . . .
};
```

コードサンプル 11-16　IDL サンプル 11-4 から生成されたコード（Java）

```
public interface parentOperations {
   public void operation1 ();
}
public interface childOperations extends parentOperations {
   public int operation2 (short s);
}
public interface parent extends
          com.inprise.vbroker.CORBA.Object, parentOperations,
          org.omg.CORBA.portable.IDLEntity {
}
public interface child extends childOperations, Baz.parent,
          org.omg.CORBA.portable.IDLEntity {
}
```

12 スマートエージェントの使用

この章では, オブジェクトインプリメンテーションを見つけるためにクライアントプログラムが登録する，スマートエージェント（osagent）について説明します。また，自分の VisiBroker ORB ドメインの設定方法，異なるローカルネットワークのスマートエージェントの接続方法，およびホスト間のオブジェクトのマイグレート方法を説明します。

12.1　スマートエージェントとは

Borland Enterprise Server VisiBroker のスマートエージェント（osagent）は，クライアントプログラムとオブジェクトインプリメンテーションの両方が使用する機能を提供する，動的な分散ディレクトリサービスです。スマートエージェントは，ご使用のローカルネットワークの少なくとも一つのホストで起動する必要があります。クライアントプログラムが，あるオブジェクトの bind()メソッドを呼び出すと，自動的にスマートエージェントに問い合わせが行われます。スマートエージェントは，クライアントとインプリメンテーションの間にコネクションを確立できるように，指定されたインプリメンテーションを探します。スマートエージェントを使用した通信は，クライアントプログラムから見て透過的です。

POA に PERSISTENT ポリシーを設定し，activate_object_with_id を使用すると，スマートエージェントはオブジェクトまたはインプリメンテーションを登録して，クライアントプログラムから使用できるようにします。オブジェクトまたはインプリメンテーションが非活性化されると，スマートエージェントはそれを使用可能オブジェクトのリストから削除します。クライアントプログラムの場合と同様に，スマートエージェントを使用した通信は，オブジェクトインプリメンテーションから見て透過的です。

12.1.1　スマートエージェントの探索

Borland Enterprise Server VisiBroker はブロードキャストメッセージを使用して，クライアントプログラムまたはオブジェクトインプリメンテーションで使用するスマートエージェントを探します。最初に応答したスマートエージェントが使用されます。スマートエージェントが見つかったあと，スマートエージェントへの登録リクエストや検索リクエストの送信には，ポイントツーポイント UDP コネクションが使用されます。UDP プロトコルを使用するのは，TCP コネクションよりもネットワーク資源の消費が少ないためです。すべての登録リクエストおよび探索リクエストは動的なので，必要な構成ファイルまたはマッピングは存在しません。

注

ブロードキャストメッセージは，スマートエージェントを探すためだけに使用されます。スマートエージェントとのほかのすべての通信には，ポイントツーポイント通信が使用されます。ブロードキャストメッセージの使用を抑止する方法の詳細については，「12.5　ポイントツーポイント通信の使用」を参照してください。

12.1.2　エージェント間の協力によるオブジェクトの探索

スマートエージェントがローカルネットワークの複数のホストで起動されると，各スマートエージェントは使用可能なオブジェクトのサブセットを認識し，見つからないオブジェクトはほかのスマートエージェントと通信して探します。スマートエージェントプロセスの一つが不測の事態で終了した場合，そのスマートエージェントに登録されたすべてのインプリメンテーションがこのイベントを発見し，これらのインプリメンテーションは自動的に別の使用可能なスマートエージェントに再登録します。

12.1.3　OAD との協力によるオブジェクトへの接続

オブジェクトインプリメンテーションは，オンデマンドで開始できるように OAD（オブジェクト活性化デーモン）に登録できます。このようなオブジェクトは，実際にアクティブな状態で OAD 内に存在するかのようにスマートエージェントに登録されます。クライアントがこれらのオブジェクトの一つにリクエストすると，そのリクエストは OAD で受け付けられます。そのあと，OAD はそのリクエストを実サーバに転送します。このとき，実サーバが起動されていない場合は OAD が起動してリクエストを転送します。スマートエージェントは，実オブジェクトインプリメンテーションが OAD によって実際に起動されているかどうかは知りません。

12.1.4　スマートエージェント（osagent）の起動

スマートエージェントの少なくとも一つのインスタンスが，ローカルネットワークのホストで実行中でなければなりません。ローカルネットワークとは，内部でブロードキャストメッセージを送信できるサブネットワークを指します。

Windows

Windows のシステムでスマートエージェントを起動するには，コマンドプロンプトで次のコマンドを入力してください。

```
prompt> osagent [options]
```

UNIX

UNIX のシステムでスマートエージェントを起動するには，次のコマンドを入力してください。

```
prompt> osagent [options] &
```

osagent コマンドには，表 12-1 のコマンドライン引数を指定できます。

表 12-1　osagent コマンドのオプション

オプション	説明
-p UDP_port	環境変数の値（UNIX の場合），または環境変数およびレジストリの値（Windows の場合）より優先して使用される osagent の UDP ポートを指定します。 注 ポート番号の有効範囲は，5001～65535 の範囲の整数値です。それ以外の値を指定した場合，動作の保証はできません。
-v	実行時に情報および診断メッセージを提供するバーボースモードをオンにします。
-help，-?	ヘルプメッセージを出力します。
-n，-N	Windows でシステムトレーアイコンを使用禁止にします。
-a ip_address	osagent がデフォルトで使用する IP アドレスを指定します。 注 マルチホームホスト上で，osagent に-a オプションを指定して起動する場合は，localaddr ファイルに-a オプションで指定したネットワークだけを記述してください。記述が正しくない場合，次のような動作になります。 • -a オプションで指定していないネットワークへ電文を送信してしまう場合があります。 • -v オプションを指定している場合，バーボースログに出力される osagent が使用するインタフェースのリストに，-a オプションで指定していないインタフェースが表示されます。
-g	UNIX でバーボースモード時に出力される情報および診断メッセージをログファイルに出力します。

次に示す osagent コマンド例は，特定の UDP ポートを指定しています。

例

```
osagent -p 17000
```

(1)　バーボース出力

osagent に-v オプションを指定した場合にバーボース出力が行われます。

UNIX

バーボース出力は stdout に出力されます。

Windows

バーボース出力は「2.5　ロギング出力」を参照してください。

```
prompt> osagent
```

(2)　エージェントを使用禁止にする

ランタイムに ORB のプロパティを渡すことによって，スマートエージェントとの通信を禁止できます。

C++の場合

```
prompt> Server -Dvbroker.agent.enableLocator=false
```

Java の場合

```
prompt> vbj -Dvbroker.agent.enableLocator=false Server
```

文字列から変換したオブジェクトリファレンス，ネーミングサービス，または URL リファレンスを利用する場合は，スマートエージェントは不要のため使用禁止にできます。オブジェクト名に bind メソッドを利用する場合は，必ずスマートエージェントを使用してください。

(3)　スマートエージェント起動時の注意事項

osagent 起動時に，socket 関数で次のエラーが発生した場合，最大で 5 回リトライ処理をします。5 回リトライ処理をしても，次のエラーが発生し続けると，osagent からアプリケーションへの送信メッセージが遅延，または osagent がすぐに終了します。

- Unix の場合：EAGAIN または EWOULDBLOCK
- Windows の場合：WSAEWOULDBLOCK

12.1.5　エージェントの可用性の確保

ローカルネットワークの複数のホストでスマートエージェントを起動すると，クライアントは，スマートエージェントの一つが不測の事態で終了した場合でも，オブジェクトへのバインドを続行できます。スマートエージェントが使用不能になると，そのスマートエージェントに登録されたすべてのオブジェクトインプリメンテーションは，別のスマートエージェントに自動的に再登録されます。ローカルネットワークで動作中のスマートエージェントがなければ，オブジェクトインプリメンテーションは新しいスマートエージェントにコンタクトできるまでリトライを続けます。

その際，C++アプリケーションは stdout に「VisiBroker: Unable to locate agent. Will try every 15 seconds to locate agent.」というメッセージを出力します。ローカルネットワーク上でスマートエージェントを起動するか，agentaddr ファイルにスマートエージェントが起動しているホストを指定してください。詳細については「12.5　ポイントツーポイント通信の使用」を参照してください。

スマートエージェントが終了しても，スマートエージェントが終了する前にクライアントとオブジェクトインプリメンテーションの間に確立されたコネクションは中断しないで継続します。ただし，クライアントが新たに_bind()リクエストを発行すると，新しいスマートエージェントにコンタクトされます。

これらのフォルトトレラントな機能を利用するために，特別なコーディングは必要ありません。ローカルネットワークの一つ以上のホストでスマートエージェントが起動されていることを確認すればよいだけです。

(1)　クライアントの存在の確認

スマートエージェントは，クライアントがまだ接続されているかどうかを確認するために，各クライアントとの通信状態を一定時間ごとにチェックします。

チェック時に一定時間の間スマートエージェントとの通信が行われていないクライアントに対して，「Are You Alive」メッセージ（ハートビートメッセージ）を送信します。クライアントが応答しなければ，スマートエージェントはクライアントがコネクションを終了したものとみなします。

クライアントへのポーリング間隔は変更できません。

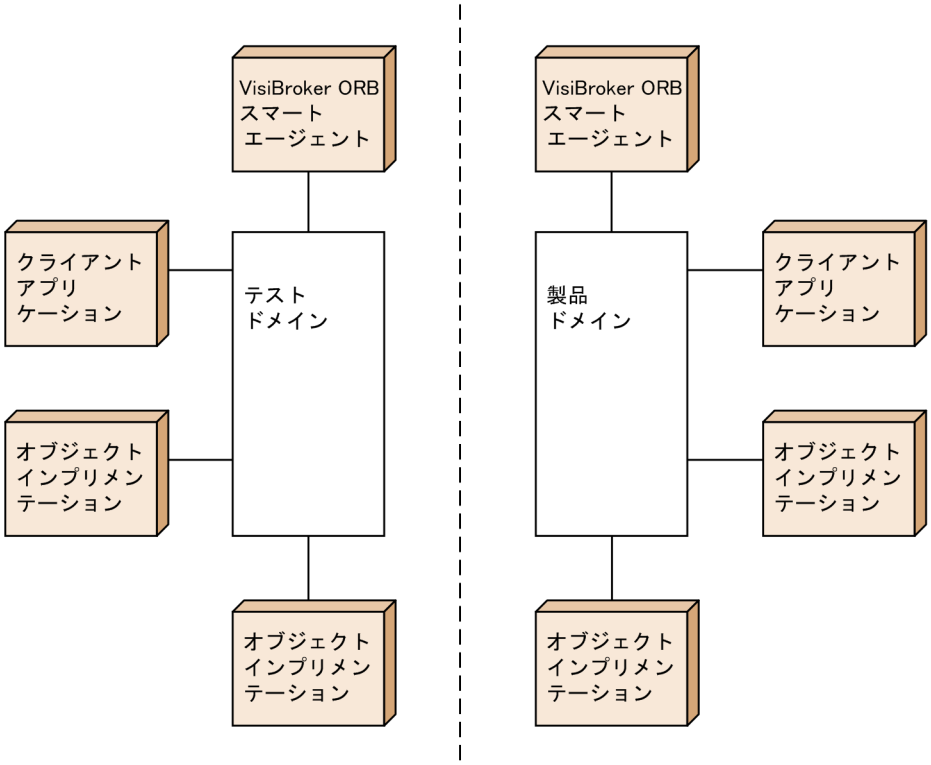
注

「クライアント」という用語の使用がオブジェクトまたはプロセスの機能を説明するとは限りません。オブジェクトリファレンスのためにスマートエージェントに接続するプログラムはどれもクライアントです。

12.2　VisiBroker ORB ドメイン内の作業

図 12-1 のように，同時に複数の VisiBroker ORB ドメインを実行することが望ましい場合がよくあります。一つのドメインを製品用のクライアントプログラムとオブジェクトインプリメンテーションで構成し，もう一つのドメインを同一クライアントとオブジェクトの，まだ一般向けにリリースされていないテストバージョンで構成できます。複数の開発者が同じローカルネットワークで作業している場合，それぞれの開発者は，テスト作業が相互に干渉し合うことがないように，自身の VisiBroker ORB ドメインを確立することを望む可能性があります。

図 12–1　別々の ORB ドメインの同時実行



Borland Enterprise Server VisiBroker では，各ドメインのスマートエージェントに対する一意の UDP ポート番号を使用することによって，同じネットワーク上の複数の VisiBroker ORB ドメインを相互に区別できます。このポート番号は，OSAGENT_PORT 環境変数で指定できます。OSAGENT_PORT 環境変数のデフォルト値は，14000 です。異なるポート番号を使用したい場合は，システム管理者に問い合わせて，使用できるポート番号を確認してください。デフォルト設定を変更するには，スマートエージェント，OAD，オブジェクトインプリメンテーション，またはその ORB ドメインに割り当てられたクライアントプログラムを実行する前に，OSAGENT_PORT 環境変数を設定する必要があります。

Windows では，OSAGENT_PORT 環境変数を設定する代わりに，Windows のレジストリ設定，または vregedit.exe ユーティリティプログラム（Borland Enterprise Server VisiBroker をインストールしたディレクトリの bin ディレクトリ内にある）を使用して設定することもできます。

コードサンプル 12-1　csh を実行する UNIX のシステムでの OSAGENT_PORT 環境変数の設定

```
prompt> setenv OSAGENT_PORT 5678
prompt> osagent &
prompt> oad &
```

また，スマートエージェントは，自身の ORB ドメインに属するアプリケーション（osfind，nameserv および oad も含む）と通信するために，OSAGENT_PORT 環境変数で設定するポート番号とは別に，OSAGENT_CLIENT_HANDLER_PORT 環境変数で設定するポート番号も使用します。
OSAGENT_CLIENT_HANDLER_PORT 環境変数で設定するポート番号は，TCP と UDP の両方のプロトコルで使用され，両方とも同じ番号です。

OSAGENT_CLIENT_HANDLER_PORT 環境変数のデフォルト値はありません。
OSAGENT_CLIENT_HANDLER_PORT 環境変数が設定されていない場合は，OS によって自動的に割り当てられたポート番号を使用して通信を行います。

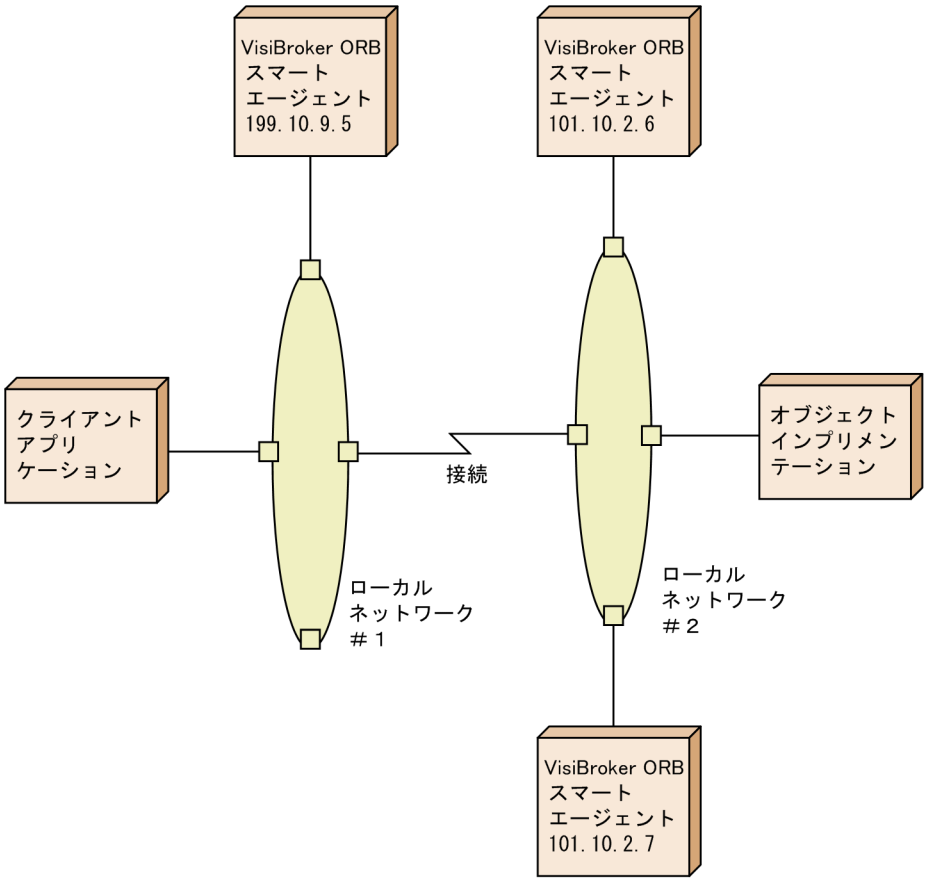
注

OSAGENT_CLIENT_HANDLER_PORT 環境変数に 0 以外を指定する場合，OSAGENT_PORT 環境変数と同様にポート番号の有効範囲は，5001～65535 の範囲の整数値です。それ以外の値を指定した場合，動作の保証はできません。

12.3　異なるローカルネットワーク上のスマートエージェントの接続

ローカルネットワーク上で複数のスマートエージェントを起動する場合，スマートエージェントは UDP ブロードキャストメッセージを使用して互いに探し合います。ネットワーク管理者は，IP サブネットマスクを使用してブロードキャストメッセージのスコープを指定することによって，ローカルネットワークを構成します。図 12-2 に，ネットワークリンクによって接続された二つのローカルネットワークを示します。

図 12-2　別々のローカルネットワークに存在する二つのスマートエージェント



ある一つのネットワーク上のスマートエージェントが別のローカルネットワーク上のスマートエージェントとコンタクトできるようにするには，リモートスマートエージェントの IP アドレスを，agentaddr という名前のファイルで使用可能にする必要があります。これは，二つのネットワーク上のスマートエージェントが UDP ブロードキャストを介して互いを検知できない場合だけ必要です。ローカルネットワーク#1 のスマートエージェントからほかのネットワークのスマートエージェントへ接続できるようにするためのファイルの指定内容を，コードサンプル 12-2 に示します。このファイルへのパスは，スマートエージェントプロセス用に設定された VBROKER_ADM 環境変数で指定されます。OSAGENT_ADDR_FILE 環境変数を設定することによって，このファイル名を変更できます。このファイルには，1 行につき一つの IP アドレスを記述できます。また「#」で始まる行は，コメントとみなされて無視されます。

コードサンプル 12-2　ネットワーク#1 のスマートエージェント用の agentaddr ファイルの内容

```
# list all OSAgent addresses.
101.10.2.6
101.10.2.7
```

適切な agentaddr ファイルによって，ローカルネットワーク#1 のクライアントプログラムは，ローカルネットワーク#2 のオブジェクトインプリメンテーションを探して使用できます。環境変数の詳細については，「2.　環境設定」を参照してください。

注

リモートネットワークで複数のスマートエージェントを実行している場合，リモートネットワーク上のすべてのスマートエージェントの IP アドレスを記述する必要があります。

12.3.1　スマートエージェントの互いの検知方法

エージェント 1 とエージェント 2 という二つのエージェントが，同じサブネット上の異なる二つのマシンから同じ UDP ポートを監視し，エージェント 1 がエージェント 2 より前に起動する場合，次のようなことが起こります。

- エージェント 2 を起動すると，自分の存在を UDP ブロードキャストして，ほかのスマートエージェントを探すためにリクエストメッセージを送信します。
- エージェント 1 はエージェント 2 がネットワーク上で使用可能であることに気づき，リクエストメッセージに応答します。
- エージェント 2 は，ほかのエージェント（エージェント 1）がネットワーク上で使用可能であることに気づきます。

エージェント 2 が正常に終了すると（例えば，＜ Ctrl ＞＋＜ C ＞を使用して終了させる），エージェント 1 はエージェント 2 がもう使用できないことを通知されます。

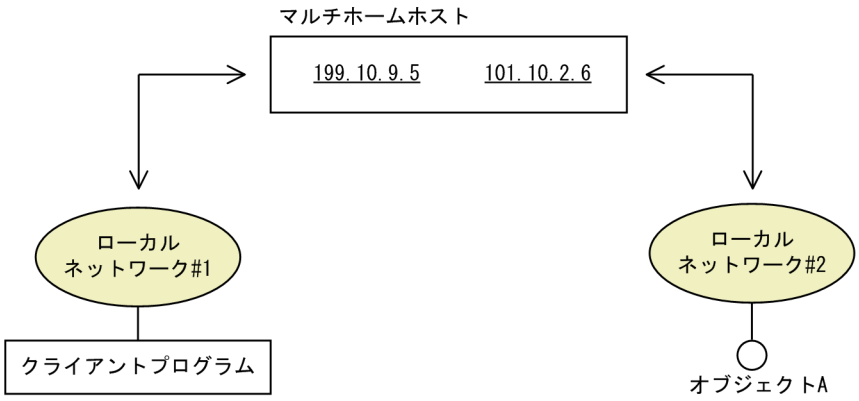
エージェント 2 が異常終了すると（例えば，タスクマネージャを使用してエージェント 2 を終了させる），エージェント 1 はエージェント 2 が使用できないことを通知されません。このため，エージェント 1 のディクショナリに存在しないオブジェクトリファレンスをクライアントが要求すると，エージェント 1 はエージェント 2 にリクエストを転送します。その結果，エージェント 1 はエージェント 2 からの応答を得られないため，エージェント 2 が使用できないことを検出し，エージェント 1 はクリーンアップ（以後，エージェント 2 を使用できないとみなします）を行います。または，エージェント 1 が行うハートビートメッセージにエージェント 2 が返信しない場合，エージェント 1 はエージェント 2 が使用できないとみなし，クリーンアップを行います。

エージェント 1 がクリーンアップを行うまで, osfind を実行しても二つのエージェントを表示し, エージェント 2 については ObjLocation::Fail 例外を表示します。

12.4　マルチホームホストを使用した作業

複数の実 IP アドレスを持つホスト（マルチホームホストと呼ばれます）上でスマートエージェントを起動すると，スマートエージェントは別々のローカルネットワークに存在するオブジェクトをブリッジするための強力なメカニズムを提供できます。そのホストが接続されているすべてのローカルネットワークは，ただ一つのスマートエージェントとだけ通信でき，ローカルネットワークのブリッジを効果的に行います。マルチホームホスト環境のスマートエージェントを図 12-3 に示します。

図 12-3　マルチホームホストのスマートエージェント



UNIX

マルチホーム UNIX ホストのスマートエージェントは，ポイントツーポイントコネクションまたはブロードキャストコネクションをサポートするすべてのホストインタフェースの監視とブロードキャストを行うように自身を動的に構成します。「12.4.1　スマートエージェント用インタフェースの指定」で説明するように，localaddr ファイルを使用してインタフェース設定値を明示的に指定できます。

Windows

マルチホーム Windows ホストのスマートエージェントは，正しいサブネットマスク値とブロードキャストアドレス値を動的に決定できません。この制限を克服するには，スマートエージェントに使用させたいインタフェース設定値を，localaddr ファイルで明示的に指定する必要があります。

-v（バーボース）オプションを使用してスマートエージェントを起動すると，作成されたメッセージの先頭に，スマートエージェントが使用する各インタフェースがリスト表示されます。コードサンプル 12-3 は，マルチホームホストでバーボースオプションを使用して起動されたスマートエージェントからの出力例を示しています。

コードサンプル 12-3　マルチホームホストで起動されたスマートエージェントからのバーボース出力

```
Bound to the following interfaces:
Address: 199.10.9.5 Subnet: 255.255.255.0 Broadcast:199.10.9.255
Address: 101.10.2.6 Subnet: 255.255.255.0 Broadcast:101.10.2.255
. . .
```

コードサンプル 12-3 に示すように，出力には，マシンの各インタフェースのアドレス，サブネットマスク，およびブロードキャストアドレスが表示されます。UNIX の場合，この出力は，UNIX コマンド ifconfig -a の結果と一致するはずです。

これらの設定値を変更したい場合は，このインタフェース情報を localaddr ファイルに指定できます。詳細については，「12.4.1　スマートエージェント用インタフェースの指定」を参照してください。

12.4.1　スマートエージェント用インタフェースの指定

注

シングルホームホストでインタフェース情報を指定する必要はありません。

マルチホームホストでスマートエージェントに使用させたい各インタフェースのインタフェース情報を，localaddr ファイルに指定できます。localaddr ファイルの別々の行に，各インタフェースの情報（ホストの IP アドレス，サブネットマスク，およびブロードキャストアドレス）を指定します。デフォルトでは，Borland Enterprise Server VisiBroker は VBROKER_ADM ディレクトリの localaddr ファイルを探索します。このファイルをポイントするように OSAGENT_LOCAL_FILE 環境変数を設定することによって，この位置を変更できます。このファイルの「#」で始まる行は，コメントとみなされて無視されます。コードサンプル 12-4 に，前述のマルチホームホストの localaddr ファイルの内容を示します。

コードサンプル 12-4　サンプルの localaddr ファイルの内容

```
#entries of format <address> <subnet_mask> <broadcast address>
199.10.9.5 255.255.255.0 199.10.9.255
101.10.2.6 255.255.255.0 101.10.2.255
```

(1)　UNIX

スマートエージェントは，UNIX を実行するマルチホームホストで自動的に自身を構成できますが，localaddr ファイルを使用して，ホストに含まれているインタフェースを明示的に指定できます。次のコマンドを使用して，UNIX ホストの使用できるすべてのインタフェース値を表示できます。コマンドの詳細は各 OS のマニュアルを参照してください。

Solaris または AIX

```
prompt> ifconfig -a
```

このコマンドによる出力は，次のようになります。

```
lo0: flags=849<UP,LOOPBACK,RUNNING,MULTICAST> mtu 8232
      inet 127.0.0.1 netmask ff000000
le0: flags=863
         <UP,BROADCAST,NOTRAILERS,RUNNING,MULTICAST> mtu 1500
      inet 199.10.9.5 netmask ffffff00 broadcast 199.10.9.255
le1: flags=863
         <UP,BROADCAST,NOTRAILERS,RUNNING,MULTICAST> mtu 1500
      inet 101.10.2.6 netmask ffffff00 broadcast 101.10.2.255
```

HP-UX

lanscan コマンドで出力されるネットワークインタフェース名を指定して，ifconfig コマンドを実行します。

(2)　Windows

Windows を実行するホストの場合，スマートエージェントが自動的に自身を構成できないため，マルチホームホストでは localaddr ファイルを使用する必要があります。ネットワーク制御パネルで TCP/IP プロトコルプロパティにアクセスすることによって，このファイルの適切な値を取得できます。ホストが Windows を実行している場合，ipconfig コマンドが必要な値を提供します。このコマンドは次のように実行してください。

```
prompt> ipconfig
```

このコマンドによる出力は，次のようになります。

```
Ethernet adapter El59x1:
      IP Address.........:199.10.9.5
      Subnet Mask ........:255.255.255.0
      Default Gateway ....:199.10.9.1
```

```
Ethernet adapter Elnk32:
      IP Address.........:101.10.2.6
      Subnet Mask ........:255.255.255.0
      Default Gateway ...:101.10.2.1
```

12.5　ポイントツーポイント通信の使用

Borland Enterprise Server VisiBroker は，UDP ブロードキャストメッセージを使用しないでスマートエージェントプロセスを探す三つの異なる機能を提供します。

これらの三つの機能を使用すると，Borland Enterprise Server VisiBroker は，次の順序でスマートエージェントの探索を行います。

C++の場合

1. 実行時パラメタとして指定されたホスト
   詳細については，「12.5.1　実行時パラメタとしてのホストの指定」，および「12.5.2　環境変数による IP アドレスの指定」を参照してください。
2. 自ホスト
3. agentaddr ファイルによって指定されたホスト
   詳細については，「12.5.3　agentaddr ファイルによるホストの指定」を参照してください。
4. 1.と同じホスト

Java の場合

1. 実行時パラメタとして指定されたホスト
   詳細については，「12.5.1　実行時パラメタとしてのホストの指定」，および「12.5.2　環境変数による IP アドレスの指定」を参照してください。
2. agentaddr ファイルによって指定されたホスト
   詳細については，「12.5.3　agentaddr ファイルによるホストの指定」を参照してください。
   agentaddr ファイルが指定されない場合で，かつ$VBROKER_ADM/agentaddr ファイルが存在する場合に，Java では，次のとおり実行時パラメタとして指定されたものとみなします。
   vbroker.agent.addrFile=$VBROKER_ADM/agentaddr
3. 自ホスト

これらの代替アプローチのどれかを使用してスマートエージェントが見つかると，それ以降のすべての受け渡しにはスマートエージェントが使用されます。

注

これらの代替アプローチを使用してもスマートエージェントを探し出せなかった場合は，UDP ブロードキャストメッセージを使用した探索を行います。

12.5.1　実行時パラメタとしてのホストの指定

コードサンプル 12-5 に，クライアントプログラムまたはオブジェクトインプリメンテーションのランタイムパラメタとして，スマートエージェントが実行されている IP アドレスを指定する方法を示します。IP アドレスを指定するとポイントツーポイントコネクションが確立されるので，ローカルネットワークの外部にあるホストの IP アドレスさえ指定できます。この機能は，ほかのすべてのホスト指定に優先します。

コードサンプル 12-5　ランタイムパラメタとしてのスマートエージェントの IP アドレスの指定

C++の場合

```
prompt> Server -Dvbroker.agent.addr=<ip_address>
```

Java の場合

```
prompt> vbj -Dvbroker.agent.addr=<ip_address> Server
```

また，プロパティファイルを介して IP アドレスを指定することもできます。vbroker.agent.addr の説明も参照してください。

コードサンプル 12-6　プロパティファイルへのスマートエージェントの IP アドレスの指定

```
vbroker.agent.addr=<ip_address>
```

デフォルトでは，プロパティファイルの vbroker.agent.addr は NULL に設定されています。

エージェントが存在している可能性があるホスト名をリスト出力してから，プロパティファイルの vbroker.agent.addrFile オプションを使用してそのファイルを指定できます。

12.5.2　環境変数による IP アドレスの指定

クライアントプログラムまたはオブジェクトインプリメンテーションを開始する前に OSAGENT_ADDR 環境変数を設定することによって，スマートエージェントの IP アドレスを指定できます。この環境変数は，osagent の動作しているホストがランタイムパラメタとして指定されていない場合に優先されます。

(1)　UNIX

```
prompt> setenv OSAGENT_ADDR 199.10.9.5
prompt> client
```

(2)　Windows

Windows で OSAGENT_ADDR 環境変数を設定するには，システムコントロールパネルを使用して環境変数を編集できます。Windows の場合の編集方法を次に示します。

1.「システムプロパティ」の「環境」を選択し，現在の変数を選択します。

2.変数ボックスに OSAGENT_ADDR とタイプ入力します。

3.値ボックスに，199.10.9.5 のように IP アドレスをタイプ入力します。

12.5.3　agentaddr ファイルによるホストの指定

「12.3　異なるローカルネットワーク上のスマートエージェントの接続」で述べたように，クライアントプログラムまたはオブジェクトインプリメンテーションは，UDP ブロードキャストメッセージを使用しないでスマートエージェントを探すために，agentaddr ファイルを使用できます。スマートエージェントを実行している各ホストの IP アドレスか完全に修飾されたホスト名を収納したファイルを作成したあと，このファイルのパスをポイントするように OSAGENT_ADDR_FILE 環境変数を設定してください。クライアントプログラムまたはオブジェクトインプリメンテーションにこの環境変数が設定されていれば，VisiBroker ORB はスマートエージェントが見つかるまで，このファイル内の各アドレスをトライします。これは，ホストを指定する機能の中で最も優先度が低い機能です。このファイルが指定されないと，$VBROKER_ADM/agentaddr ファイルが使用されます。

注

agentaddr ファイルに日本語や 2 バイトコード文字を含むホスト名を指定しないでください。指定した場合，日本語や 2 バイトコード文字を含むホスト名の osagent には接続されません。

12.6　オブジェクト可用性の確保

複数のホストでオブジェクトのインスタンスを開始することによって, これらのオブジェクトのフォルトトレランスを提供できます。インプリメンテーションが使用不能になると, VisiBroker ORB はクライアントプログラムとオブジェクトインプリメンテーションの間のコネクションの切断を検出し, クライアントが設定した有効なリバインドポリシーによってオブジェクトインプリメンテーションの別のインスタンスとコネクションを確立するために, 自動的にスマートエージェントにコンタクトします。クライアントポリシーの設定の詳細については,「10.5　Quality of Service の使用」を参照してください。

注

VisiBroker ORB がクライアントを別のオブジェクトインプリメンテーションに再接続させようとする場合，リバインドオプションを使用可能にする必要があります。これはデフォルトの動作です。

12.6.1　状態を維持しないオブジェクトのメソッドの呼び出し

クライアントプログラムは, オブジェクトの新しいインスタンスが使用中であるかどうかを関知することなく，状態を維持しないオブジェクトインプリメンテーションのメソッドを呼び出せます。

12.6.2　状態を維持するオブジェクトのフォルトトレランスの実現

状態を維持するオブジェクトインプリメンテーションに対してもフォルトトレランスを実現できますが, これはクライアントプログラムから見て透過的ではありません。この場合，クライアントプログラムは QoS（Quality of Service）ポリシー VB_NOTIFY_REBIND を使用するか，または，VisiBroker ORB オブジェクトのインタセプタを登録して状態の変更時に ORB によって呼び出されるよう実装しておきます。QoS の使用に関する詳細については,「10.5　Quality of Service の使用」を参照してください。

オブジェクトインプリメンテーションとのコネクションが失敗し, VisiBroker ORB がクライアントを別のオブジェクトインプリメンテーションに再接続すると，bind インタセプタの bind メソッドが VisiBroker ORB によって呼び出されます。クライアントは，このバインドメソッドを実装することによって，別のオブジェクトインプリメンテーションに再接続されたことを知ることができます。インタセプタの詳細については,「20.　VisiBroker 4.x インタセプタの使用」を参照してください。

12.6.3　OAD に登録されたオブジェクトの複製

オブジェクトがダウンしても OAD によって再起動されるので, OAD はより高いオブジェクト可用性を保証します。ホストが使用不能になった時のためにフォルトトレランスを望む場合は，OAD を複数のホストで起動し，オブジェクトを各 OAD インスタンスに登録する必要があります。

注

Borland Enterprise Server VisiBroker が提供するタイプのオブジェクト複製では，マルチキャストやミラーリング機能を提供しません。任意の時点で，クライアントプログラムと個々のオブジェクトインプリメンテーションの間には常に一対一の対応関係があります。

12.7　ホスト間のオブジェクトのマイグレート

オブジェクトマイグレーションとは，一つのホストでオブジェクトインプリメンテーションを終了し，別のホストでそれを開始する方法です。オブジェクトマイグレーションを使用して，過負荷のホストから，より多くの資源または処理能力を持つホストにオブジェクトを移動することによって負荷分散ができます（異なる osagent に登録されたサーバ間の負荷分散はありません）。ハードウェアまたはソフトウェアの保守のためにホストをシャットダウンしなければならない場合，オブジェクトを使用できる状態に保つためにもオブジェクトマイグレーションを使用できます。

注

状態を維持しないオブジェクトのマイグレーションは，クライアントプログラムから見て透過的です。マイグレートされたオブジェクトインプリメンテーションにクライアントが接続されると，スマートエージェントはコネクションの切断を検出し，クライアントを新しいホストの新しいオブジェクトに透過的に再接続します。

12.7.1　状態を維持するオブジェクトのマイグレート

状態を維持するオブジェクトのマイグレーションもできますが，マイグレーションプロセスが始まる前に接続したクライアントプログラムから見て，このマイグレーションは透過的ではありません。この場合，クライアントプログラムはオブジェクトのインタセプタを登録する必要があります。

(1)　C++

元のオブジェクトとのコネクションが失われ，VisiBroker ORB がクライアントをオブジェクトに再接続すると，インタセプタの bind_succeeded()メソッドが VisiBroker ORB によって呼び出されます。クライアントは，オブジェクトの状態を最新のものにするためにこのメソッドをインプリメントできます。

(2)　Java

元のオブジェクトとのコネクションが失われ，VisiBroker ORB がクライアントをオブジェクトに再接続すると，インタセプタの bind_succeeded()メソッドが VisiBroker ORB によって呼び出されます。クライアントは，オブジェクトの状態を最新のものにするためにこのメソッドをインプリメントできます。

インタセプタについては，「20.　VisiBroker 4.x インタセプタの使用」を参照してください。

12.7.2　実体化されたオブジェクトのマイグレート

マイグレートしたい VisiBroker ORB オブジェクトが，インプリメンテーションのクラスを実体化するサーバプロセスによって生成されていた場合，ユーザは新しいホストでオブジェクトを開始し，サーバプロセスを終了するだけです。元のインスタンスが終了すると，それはスマートエージェントから登録解除されます。新しいホストで新しいインスタンスを開始すると，それはスマートエージェントに登録されます。この時点から，クライアントの呼び出しは，新しいホストのオブジェクトインプリメンテーションに転送されます。

12.7.3　OAD に登録されたオブジェクトのマイグレート

マイグレートしたい VisiBroker ORB オブジェクトが OAD に登録されている場合，旧ホストの OAD から VisiBroker ORB オブジェクトを登録解除する必要があります。そのあと，VisiBroker ORB オブジェクトを新しいホストの OAD に登録してください。この手順を次に示します。

1. 旧ホストの OAD からオブジェクトインプリメンテーションを登録解除します。

2. 新ホストの OAD にオブジェクトインプリメンテーションを登録します。
3. 旧ホストのオブジェクトインプリメンテーションを終了します。

オブジェクトインプリメンテーションの登録および登録解除の詳細については,「15.　オブジェクト活性化デーモンの使用」を参照してください。

12.8　すべてのオブジェクトとサービスの報告

スマートファインダ（osfind）コマンドは，スマートエージェント（osagent）と連携して，現在ネットワーク上で使用可能な次の情報を報告します。

- スマートエージェントが動作するホストおよび数
- Borland Enterprise Server VisiBroker オブジェクト
- オブジェクト活性化デーモン（OAD）が動作するホストおよび数
- オブジェクト活性化デーモンに登録されたオブジェクト

接続したスマートエージェントが保持する情報が使用できないスマートエージェントの場合には，ObjLocation::Fail 例外を表示します。

osfind コマンドの構文は次のとおりです。

構文

```
osfind [options]
```

osfind コマンドでは表 12-2 に示すオプションが有効です。オプションを指定しないと，osfind はドメイン内のすべてのエージェント，OAD，およびインプリメンテーションをリスト表示します。

表 12-2　osfind コマンドのオプション

オプション	説明
-a	ドメイン内のすべてのスマートエージェントをリスト表示します。
-b	VisiBroker 2.0 と下位互換性のある osfind 機能を使用します。
-d	ドットを含む数値アドレスでホスト名を出力します。
-f <agent address filename>	ファイルに指定されたホストで実行されているスマートエージェントを問い合わせます。このファイルには，1 行につき一つの IP アドレス，または完全に修飾されたホスト名があります。すべてのスマートエージェントを報告する場合にこのファイルは使用されないことに注意してください。このファイルはオブジェクトインプリメンテーションとサービスの報告時だけに使用されます。
-g	オブジェクトがあるかどうかを確認します。このオプションを指定してコマンドを実行するとシステムのロードが大きく遅れる原因となる場合があります。BY_INSTANCE を登録したオブジェクトだけについて，その有無が確認されます。 OAD に登録されたオブジェクト，または BY_POA ポリシーを登録したオブジェクトについては，その有無の確認は行われません。
-h,-help,-usage,-?	このオプションのヘルプ情報を出力します。
-o	ドメイン内のすべての OAD をリスト表示します。
-C	終了する前にポーズします。コマンドの実行完了と同時に終了してしまうような Windows のコンソールで使用すると，コンソールがポーズするため終了しなくなります。

コードサンプル 12-7 は，osfind コマンドの出力例を示しています。

コードサンプル 12-7　osfind コマンドの出力例

```
prompt>osfind
osfind: Found one agent at port 14000
        HOST: HostA
osfind: Found 1 OADs in your domain
        HOST: HostA
```

```
osfind: Following are the list of Implementations registered with OADs.
        HOST: HostA
                REPOSITORY ID: IDL:Bank/Account:1.0
                        OBJECT NAME: Jack B. Quick
osfind: Following are the list of Implementations started manually.
        HOST: HostA
                REPOSITORY ID: IDL:visigenic.com/Activation/OAD:1.0
                        OBJECT NAME: 172.17.113.31
```

12.9　オブジェクトへのバインド

クライアントアプリケーションがインタフェース上のメソッドを呼び出すには，まず bind メソッドでオブジェクトリファレンスを取得しなければなりません。

クライアントアプリケーションが bind メソッドを呼び出すと，VisiBroker ORB はアプリケーションの代わりに幾つかの機能を実行します。

- VisiBroker ORB は osagent にコンタクトして，要求されたインタフェースを提供しているオブジェクトサーバを探索します。オブジェクト名とホスト名（または IP アドレス）が指定されていれば，これらはディレクトリサービス検索をさらに限定するために使用されます。
- オブジェクトインプリメンテーションが探索されると，VisiBroker ORB は探索されたオブジェクトインプリメンテーションとクライアントアプリケーション間のコネクションを設定しようとします。
- コネクションの設定に成功すると，VisiBroker ORB は必要に応じてプロキシオブジェクトを生成し，そのオブジェクトにリファレンスを返します。

注

VisiBroker ORB は個別のプロセスではありません。VisiBroker ORB はクラスとほかのリソースの集まりであり，これによってクライアントとサーバ間で通信できるようになります。

13 ロケーションサービスの使用

この章では，Borland Enterprise Server VisiBroker ロケーションサービスの使用方法について説明します。Borland Enterprise Server VisiBroker ロケーションサービスは, 特定の属性に基づいてオブジェクトインスタンスを見つけられる高度なオブジェクトディスカバリを提供します。ロケーションサービスは Borland Enterprise Server VisiBroker スマートエージェントを使って，ネットワーク上でどのオブジェクトが現在アクセスできるか，それらがどこに存在するかを通知します。ロケーションサービスは CORBA の仕様に対する Borland Enterprise Server VisiBroker の拡張機能で，Borland Enterprise Server VisiBroker でインプリメントされたオブジェクトを見つけるためだけに有用です。

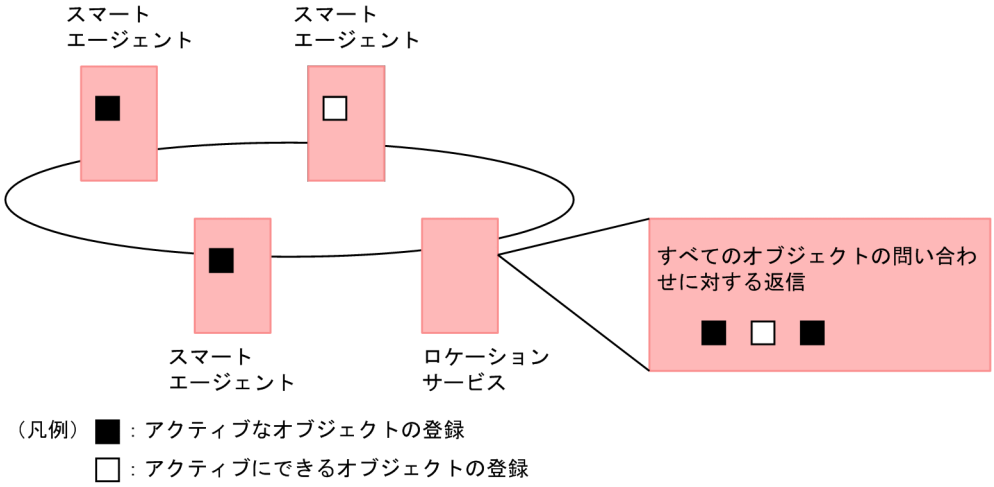
13.1　ロケーションサービスとは

ロケーションサービスは CORBA の仕様に対する拡張機能で，オブジェクトインスタンスを見つける汎用機能を提供します。ロケーションサービスは，自身が知っているインスタンスのリストを格納しているカタログを保守するスマートエージェントと直接通信します。ロケーションサービスから問い合わせがあると，スマートエージェントはその問い合わせをほかのスマートエージェントに転送し，その応答を集めてロケーションサービスに返します。

ロケーションサービスは，BY_INSTANCE ポリシーを指定した POA に登録されたすべてのオブジェクトインスタンスについて知っています。これらのオブジェクトを含むサーバは，手動起動または OAD による自動起動ができます。

図 13-1 はこの概念を表したものです。

図 13-1　スマートエージェントを使用した，オブジェクトのインスタンスの検索



注

サーバは，インスタンスを生成するときにインスタンスのスコープを指定します。グローバルにスコープされたインスタンスだけがスマートエージェントに登録されます。

ロケーションサービスは，スマートエージェントが各オブジェクトインスタンスについて保持している情報を使用できます。各オブジェクトインスタンスに対して，ロケーションサービスは IDL サンプル 13-1 のような ObjLocation::Desc 構造体の中にカプセル化された情報を保持しています。

IDL サンプル 13-1　Desc 構造体の IDL

```
struct Desc {
     Object ref;
     ::IIOP::ProfileBodyValue iiop_locator;
     string repository_id;
     string instance_name;
     boolean activable;
     string agent_hostname;
};
typedef sequence<Desc> DescSeq;
```

Desc 構造体の IDL には次のような情報が入っています。

- オブジェクトリファレンスである ref は，オブジェクトを呼び出すハンドルです。

- iiop_locator インタフェースは，インスタンスのサーバのホスト名およびポートへのアクセスをできるようにします。この情報は，唯一サポートされているプロトコルである IIOP にオブジェクトが接続されている場合だけ意味があります。ホスト名はインスタンス記述内の文字列として返されます。
- repository_id は，IR（インタフェースリポジトリ）とインプリメンテーションリポジトリの中で検索できるオブジェクトインスタンスのインタフェース指定です。一つのインスタンスが複数のインタフェースに応じる場合は，インタフェースごとに一つのインスタンスがあるかのように，カタログにはインタフェースごとにエントリが含まれます。
- instance_name は，そのサーバがオブジェクトに与えた名前です。
- activable フラグは，OAD が活性化できるインスタンスと，手動操作で起動されるインスタンスを識別します。
- agent_hostname は，インスタンスが登録されているスマートエージェント名です。

ロケーションサービスは負荷分散や監視などの目的に役立ちます。あるオブジェクトの複製が幾つかのホスト上にある場合, 複製を提供するホスト名と各ホストの最近の負荷平均のキャッシュを維持するバインドインタセプタを配置できます。インタセプタは, オブジェクトのインスタンスを現在提供しているホストについてロケーションサービスに尋ねることでキャッシュを更新してから, ホストに問い合わせて負荷平均を取得します。そのあと, インタセプタは最も負荷が軽いホスト上の複製に対するオブジェクトリファレンスを返します。インタセプタの記述の詳細については,「19.　ポータブルインタセプタの使用」を参照してください。

13.2　ロケーションサービスコンポーネント

ロケーションサービスは Agent インタフェースを介してアクセスできます。Agent インタフェースのメソッドは, 二つのグループに分類できます。インスタンスを記述するデータをスマートエージェントに問い合わせるグループと，トリガーの登録や登録解除を行うグループです。トリガーは，ロケーションサービスのクライアントにインスタンスの可用性の変更を通知する通知機能を提供します。

13.2.1　ロケーションサービスエージェントとは

ロケーションサービスエージェントは, スマートエージェントのネットワークに接続されたオブジェクトを探索するメソッドの集まりです。IR の ID に基づいて，または IR の ID とインスタンス名の組み合わせに基づいて問い合わせができます。問い合わせ結果はオブジェクトリファレンスとして, またはより詳細なインスタンスの情報として返されます。オブジェクトリファレンスは, スマートエージェントが見つけたオブジェクトの特定インスタンスのハンドルに過ぎません。インスタンス記述に含まれるのはオブジェクトリファレンス，インスタンスのインタフェース名，インスタンス名，ホスト名とポート番号，および状態に関する情報（実行中か活性化可能かなど）です。

注

該当するサービスはコア VisiBroker ORB に統合されたため，旧バージョンのように locserv 実行形式ファイルはもう存在しません。

次の IDL サンプルが与える IR の ID とインスタンス名の使用方法を図 13-2 に示します。

```
module Automobile {
         interface Car{...};
         interface Sedan:Car {...};
}
```

図 13-2　IR の ID とインスタンス名の使用

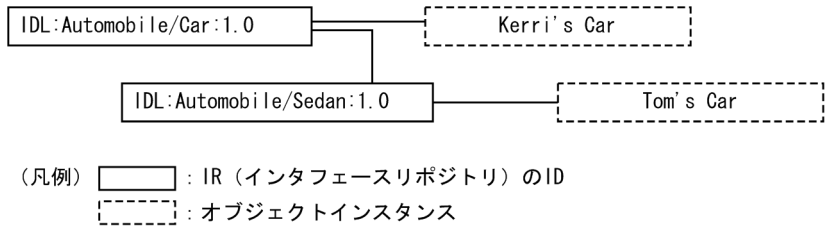
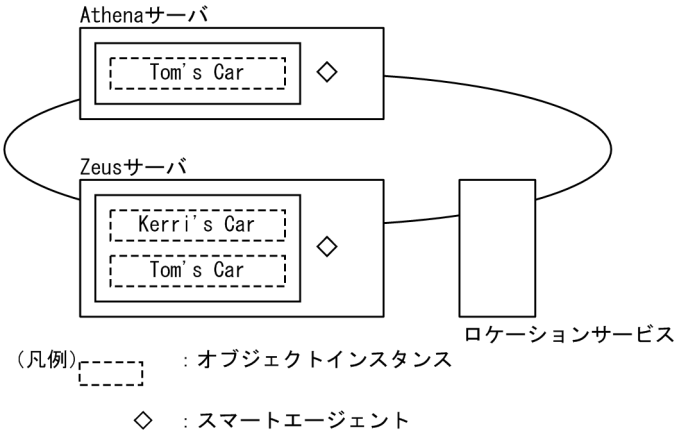


図 13-3 は，図 13-2 の例を使って，ネットワーク上のスマートエージェントと Car のインスタンスのリファレンスをわかりやすく示したものです。この例では, Kerri's Car のインスタンスと Tom's Car の二つの複製という三つのインスタンスがあります。

図 13-3　一つのインタフェースのインスタンスを持つネットワーク上のスマートエージェント



以降，Agent クラスが提供したメソッドを使用して Borland Enterprise Server VisiBroker スマートエージェントの情報を問い合わせる方法を説明します。それぞれの照会メソッドは，失敗の理由を示す Fail 例外を発生させることができます。

(1)　スマートエージェントを実行するすべてのホスト名の取得

all_agent_locations()メソッドの HostnameSeq（C++）または String[ ]（Java）を使用して，Borland Enterprise Server VisiBroker スマートエージェントのホストとして動作しているサーバを見つけられます。図 13-3 の例では，このメソッドは Athena と Zeus という二つのサーバの名前を返します。

(2)　アクセス可能なすべてのインタフェースの検索

アクセス可能なインタフェースをすべて見つけるために，ネットワーク上の Borland Enterprise Server VisiBroker スマートエージェントに問い合わせることができます。そのためには all_repository_ids()メソッドの RepositoryIDSeq（C++），または String[ ]（Java）を使用できます。図 13-3 の例では，このメソッドは Car と Sedan という二つのインタフェースのリポジトリ ID を返します。

注

旧バージョンの VisiBroker ORB では，IDL インタフェース名を使用してインタフェースを識別していましたが，ロケーションサービスは，その代わりにリポジトリ ID を使用します。この違いを説明すると，インタフェース名が::module1::module2::interface の場合，これに等しいリポジトリ ID は IDL:module1/module2/interface:1.0 です。図 13-2 の例では，Car のリポジトリ ID は IDL:Automobile/Car:1.0，Sedan のリポジトリ ID は IDL:Automobile/Sedan:1.0 になります。

(3)　あるインタフェースのインスタンスのリファレンスの取得

個々のインタフェースのすべての使用可能インスタンスを見つけるために，ネットワーク上の Borland Enterprise Server VisiBroker スマートエージェントに問い合わせることができます。問い合わせる場合，表 13-1 または表 13-2 のメソッドのどれかを使用できます。

表 13-1　任意のインタフェースをインプリメントするオブジェクトのリファレンスの取得（C++）

メソッド	説明
CORBA::ObjectSeq* all_instances (const char*_repository_id)	このメソッドを使用すると，インタフェースのインスタンスのオブジェクトリファレンスが返されます。

メソッド	説明
DescSeq* all_instance_descs (const char*_repository_id)	このメソッドを使用すると，インタフェースのインスタンスのインスタンス記述が返されます。

表 13-2　任意のインタフェースをインプリメントするオブジェクトのリファレンスの取得（Java）

メソッド	説明
org.omg.CORBA.Object[ ] all_instances (String repository_id)	このメソッドを使用すると，インタフェースのインスタンスのオブジェクトリファレンスが返されます。
Desc[ ] all_instance_descs (String repository_id)	このメソッドを使用すると，インタフェースのインスタンスのインスタンス記述が返されます。

図 13-3 の例では，IDL:Automobile/Car:1.0 というリクエストでどれかのメソッドを呼び出すと，Tom's Car on Athena，Tom's Car on Zeus および Kerri's Car という Car インタフェースの三つのインスタンスが返されます。Tom's Car インスタンスは，二つの異なるスマートエージェントによって検索されるため，2 度返されます。

(4)　あるインタフェースの同名インスタンスに対するリファレンスの取得

表 13-3 または表 13-4 のメソッドの一つを使用して，特定のインスタンス名があればそれをすべて返すようにネットワーク上の Borland Enterprise Server VisiBroker スマートエージェントに問い合わせることができます。

表 13-3　あるインタフェースの同名インスタンスに対するリファレンス（C++）

メソッド	説明
CORBA::ObjectSeq* all_replica (const char*_repository_id, const char*_instance_name)	このメソッドを使用すると，インタフェースの同名インスタンスのオブジェクトリファレンスが返されます。
DescSeq all_replica_descs (const char*_repository_id, const char*_instance_name)	このメソッドを使用すると，インタフェースの同名インスタンスのインスタンス記述が返されます。

表 13-4　あるインタフェースの同名インスタンスに対するリファレンス（Java）

メソッド	説明
org.omg.CORBA Object[ ] all_replica (String repository_id, String instance_name)	このメソッドを使用すると，インタフェースの同名インスタンスのオブジェクトリファレンスが返されます。
Desc[ ] all_replica_descs (String repository_id, String instance_name)	このメソッドを使用すると，インタフェースの同名インスタンスのインスタンス記述が返されます。

図 13-3 の例では，リポジトリ ID に IDL:Automobile/Sedan:1.0 を，インスタンス名に Tom's Car を指定してどれかのメソッドを呼び出すと，二つのインスタンスが返されます。これは，二つの異なるスマートエージェントによってインスタンスが発生するためです。

13.2.2　トリガーとは何か

トリガーとは本来, 指定されたインスタンスの使用の可否をユーザに判定させるコールバック機能です。トリガーは Agent のポーリングの非同期な代替手段で，通常はオブジェクトとのコネクションが絶たれたあとの回復に使用されます。問い合わせはさまざまな方法で行われますが，トリガーは特殊な用途です。

(1)　トリガーメソッドの考察

Agent クラスのトリガーメソッドについて表 13-5 および表 13-6 に示します。

表 13-5　トリガーメソッド（C++）

メソッド	説明
void reg_trigger (const TriggerDesc& _desc, TriggerHandler_ptr _handler);	このメソッドはトリガーハンドラを登録するために使用します。
void unreg_trigger (const TriggerDesc& _desc, TriggerHandler_ptr _handler)	このメソッドはトリガーハンドラの登録を解除するために使用します。

表 13-6　トリガーメソッド（Java）

メソッド	説明
void reg_trigger (com.inprise.vbroker.ObjLocation.TriggerDesc desc,com.inprise.vbroker.ObjLocation.TriggerHandler handler)	このメソッドはトリガーハンドラを登録するために使用します。
void unreg_trigger (com.inprise.vbroker.ObjLocation.TriggerDesc desc,com.inprise.vbroker.ObjLocation.TriggerHandler handler)	このメソッドはトリガーハンドラの登録を解除するために使用します。

どちらの Agent トリガーメソッドも，失敗の理由を示す Fail 例外を発生させることができます。

TriggerHandler インタフェースは，表 13-7 および表 13-8 で説明するメソッドで構成されます。

表 13-7　TriggerHandler インタフェースメソッド（C++）

メソッド	説明
void impl_is_ready(const Desc& _desc);	このメソッドは，desc に一致するインスタンスがアクセス可能になると，ロケーションサービスによって呼び出されます。
void impl_is_down(const Desc& _desc)	このメソッドは，インスタンスが使用不能になると，ロケーションサービスによって呼び出されます。

表 13-8　TriggerHandler インタフェースメソッド（Java）

メソッド	説明
void impl_is_ready (com.inprise.vbroker.ObjLocation.TriggerDesc desc)	このメソッドは，desc に一致するインスタンスがアクセス可能になると，ロケーションサービスによって呼び出されます。
void impl_is_down (com.inprise.vbroker.ObjLocation.TriggerDesc desc)	このメソッドは，インスタンスが使用不能になると，ロケーションサービスによって呼び出されます。

(2)　トリガーの生成

TriggerHandler はコールバックオブジェクトです。TriggerHandler をインプリメントするには，TriggerHandlerPOA クラスから派生させ，その impl_is_ready()メソッドと impl_is_down()メソッドをインプリメントします。ロケーションサービスにトリガーを登録するには，Agent インタフェースの reg_trigger()メソッドを使用します。このメソッドには，監視したいインスタンスを記述し，インスタンスの可用性が変化した場合に呼び出したい TriggerHandler オブジェクトを指定する必要があります。インスタンス記述（TriggerDesc）には，リポジトリ ID，インスタンス名，ホスト名などのインスタンス情報の組み合わせを入れることができます。提供するインスタンス情報が詳細であればあるほど，インスタンスの仕様も特定化できます。

IDL サンプル 13-2　TriggerDesc の IDL

```
struct TriggerDesc {
    string repository_id;
    string instance_name;
    string host_name;
};
```

注

TriggerDesc のフィールドに空の文字列 ("") を設定すると，このフィールドは無視されます。各フィールドのデフォルト値は空の文字列です。

例えば，リポジトリ ID しか持たない TriggerDesc は，そのインタフェースのどのインスタンスにも一致します。図 13-3 の例に戻ると，IDL:Automobile/Car:1.0 のどのインスタンスのトリガーも，Tom's Car on Athena，Tom's Car on Zeus，Kerri's Car というインスタンスのうちの一つが使用可能または使用不能になるとして発生します。ただし，TriggerDesc に「Tom's Car」というインスタンス名を追加すると，二つの「Tom's Car」というインスタンスのどちらかの可用性が変化する場合だけトリガーが発生するように指定を厳しくします。最後に Athena というホスト名を追加すると，Athena サーバの Tom's Car インスタンスが使用可能または使用不能になる場合に限りトリガーが発生するように，トリガーをさらに改善できます。

(3)　トリガーが検出した最初のインスタンスだけを確認

トリガーは何度も呼び出されるものです。TriggerHandler は，トリガー記述に合致するオブジェクトがアクセス可能になるたびに呼び出されます。最初のインスタンスがアクセス可能になる場合だけを知りたいときがあるでしょう。この場合には，このようなインスタンスの存在が最初に見つかったあと，Agent の unreg_trigger()メソッドを呼び出してトリガーを登録解除します。

13.3　Agent の問い合わせ

この節では, ロケーションサービスを使用してインタフェースのインスタンスを見つける二つのサンプルを示します。最初のサンプルでは，次の IDL の抜粋部分のような Account インタフェースを使用します。

IDL サンプル 13-3　Account インタフェース定義の例

```
//Bank.idl
module Bank {
   interface Account {
      float balance();
   };
   interface AccountManager {
      Account open (in string name);
   };
};
```

13.3.1　あるインタフェースのすべてのインスタンスの検索

コードサンプル 13-1 および 13-2 は, all_instances()メソッドを使用して Account インタフェースのすべてのインスタンスを探します。スマートエージェントへの問い合わせは, ORB::resolve_initial_references()メソッド（C++）または ORB.resolve_initial_references()メソッド（Java）に「LocationService」を渡し，そのメソッドが返したオブジェクトを ObjLocation::Agent（C++）または ObjLocation.Agent（Java）にナロウすることによって行われることに注意してください。また Account リポジトリ ID のフォーマットは IDL:Bank/Account:1.0 であることにも注意してください。

コードサンプル 13-1　AccountManager インタフェースの要求を満たすすべてのインスタンスの検索 (C++)

```
#include ”corba.h”
#include ”locate_c.hh”

// USE_STD_NS is a define setup by VisiBroker to use the std
// namespace USE_STD_NS

int main(int argc, char** argv)
{
   try {
      // ORB initialization
      CORBA::ORB_var the_orb = CORBA::ORB_init(argc, argv);
      // Obtain a reference to the Location Service
      CORBA::Object_var obj = the_orb->
         resolve_initial_references(”LocationService”);
      if ( CORBA::is_nil(obj) ) {
         cout << ”Unable to locate initial LocationService”
              << endl;
         return 0;
      }
      ObjLocation::Agent_var the_agent =
           ObjLocation::Agent::_narrow(obj);
      // Query the Location Service for all implementations of
      // the Account interface
      ObjLocation::ObjSeq_var accountRefs =
         the_agent->all_instances(
         ”IDL:Bank/AccountManager:1.0”);
      cout << ”Obtained ” << accountRefs->length()
           << ”Account objects” << endl;
      for (CORBA::ULong i=0; i < accountRefs->length(); i++) {
         cout << ”Stringified IOR for account #”<< i << ”:”
              << endl;
         CORBA::String_var stringified_ior(
              the_orb->object_to_string(accountRefs[i]));
         cout << stringified_ior << endl;
         cout << endl;
      }
   } catch (const CORBA::Exception& e) {
      cout << ”Caught exception: ” << e << endl;
```

```
        return 0;
    }
    return 1;
}
```

コードサンプル 13-2　AccountManager インタフェースの要求を満たすすべてのインスタンスの検索 (Java)

```
// AccountFinder.java
public class AccountFinder {
   public static void main(String[ ] args){
   try {
      // Initialize the ORB.
      org.omg.CORBA.ORB orb =
                          org.omg.CORBA.ORB.init(args,null);
      com.inprise.vbroker.ObjLocation.Agent the_agent = null;
      try {
         the_agent =
             com.inprise.vbroker.ObjLocation.AgentHelper.narrow(
             orb.resolve_initial_references("LocationService"));
      } catch (org.omg.CORBA.ORBPackage.InvalidName e){
         System.out.println(
            "Not able to resolve references " +
            "for LocationService");
         System.exit(1);
      } catch (Exception e){
         System.out.println(
            "Unable to locate LocationService!");
         System.out.println("Caught exception: " + e);
         System.exit(1);
      }
      org.omg.CORBA.Object[ ] accountRefs =
         the_agent.all_instances(
                      "IDL:Bank/AccountManager:1.0");
      System.out.println("Agent returned " +
         accountRefs.length  +
         "object references");
      for (int i=0; i <accountRefs.length; i++){
         System.out.println(
             "Stringified IOR for account #" + (i+1) + ":");
         System.out.println(orb.object_to_string(
                                        accountRefs[i]));
         System.out.println();
      }
   }catch (Exception e){
      System.out.println("Caught exception: " + e);
      System.exit(1);
   }
}
}
```

13.3.2　スマートエージェントが認識するものをすべて検索

コードサンプル 13-3 および 13-4 は，スマートエージェントが認識するものをすべて検索する方法を示します。これにはまず，all_repository_ids()メソッドを呼び出して，既知のインタフェースをすべて取得します。次に，all_instances_descs()メソッドをインタフェースごとに呼び出して，インスタンス記述を取得します。

コードサンプル 13-3　スマートエージェントが認識するものをすべて検索（C++）

```
#include "corba.h"

#include "locate_c.hh"

// USE_STD_NS is a define setup by VisiBroker to use
// the std namespace if it exists
USE_STD_NS

int DisplaybyRepID(CORBA::ORB_ptr the_orb,
         ObjLocation::Agent_var the_agent,
         char * myRepId){
```

```
   ObjLocation::ObjSeq_var accountRefs;
   accountRefs = the_agent->all_instances(myRepId);
   cout << "Obtained " << accountRefs->length() <<
         "Account objects" << endl;
   for (CORBA::ULong i=0; i < accountRefs->length(); i++) {
      cout << "Stringified IOR for account #" << i << ":"
            << endl;
      CORBA::String_var stringified_ior(
            the_orb->object_to_string(accountRefs[i]));
      cout << stringified_ior << endl;
      cout << endl;
   }
   return(1);
}
void PrintUsage(char * name) {
   cout << "¥nUsage: ¥n" << endl;
   cout << "¥t" << name << " [Rep ID]" << endl;
   cout << "¥n¥tWith no argument,
         finds and prints all objects" << endl;
   cout << "¥tOptional rep_ID searches for
         specific rep ID¥n" << endl;
}
int main(int argc, char** argv) {
   char myRepId[255] = "";
   if (argc == 2) {
      if (!strcmp(argv[1], "-h") || !strcmp(argv[1], "/?") ||
            !strcmp(argv[1], "-?") ){
         PrintUsage(argv[0]);
         exit(0);
      } else {
         strcpy(myRepId, argv[1]);
      }
   }
   else if (argc > 2) {
      PrintUsage(argv[0]);
      exit(0);
   }
   try {
      CORBA::ORB_ptr the_orb = CORBA::ORB_init(argc, argv);
      CORBA::Object_ptr obj = the_orb->
      resolve_initial_references("LocationService");
      if ( CORBA::is_nil(obj) ) {
         cout << "Unable to locate initial LocationService"
               << endl;
         return 0;
      }
      ObjLocation::Agent_var the_agent =
            ObjLocation::Agent::_narrow(obj);
      ObjLocation::DescSeq_var descriptors;

      //Display stringified IOR for RepID requested and exit
      if (argc == 2) {
         DisplaybyRepID(the_orb, the_agent, myRepId);
         exit(0);
      }
      //Report all hosts running osagents
      ObjLocation::HostnameSeq_var HostsRunningAgents =
            the_agent->all_agent_locations();
      cout << "Located " << HostsRunningAgents->length() <<
            "Hosts running Agents" << endl;
      for (CORBA::ULong k=
            0; k<HostsRunningAgents->length(); k++){
         cout << "¥tHost #" << (k+1) << ": " <<
               (const char*) HostsRunningAgents[k] << endl;
      }
      cout << endl;

      // Find and display all Repository Ids
      ObjLocation::RepositoryIdSeq_var repIds =
            the_agent->all_repository_ids();
      cout << "Located " << repIds->length() <<
            " Repository Ids" << endl;
      for (CORBA::ULong j=0; j<repIds->length(); j++) {
         cout << "¥tRepository ID #" << (j+1) << ": " <<
```

```
            repIds[j] << endl;
      }
      // Find all Object Descriptors for each Repository Id
      for (CORBA::ULong i=0; i < repIds->length(); i++) {
         descriptors =
            the_agent->all_instances_descs(repIds[i]);
         cout << endl;
         cout << "Located " << descriptors->length()
            << " objects for " << (const char*) (repIds[i])
            << " (Repository Id #" << (i+1) << "):" <<endl;
         for (CORBA::ULong j=0; j < descriptors->length(); j++){
            cout << endl;
            cout << (const char*) repIds[i] << " #"
               << (j+1) << ":" << endl;
            cout << "¥tInstance Name ¥t="
               << descriptors[j].instance_name << endl;
            cout << "¥tHost ¥t="
               << descriptors[j].iiop_locator.host << endl;
            cout << "¥tPort ¥t="
               << descriptors[j].iiop_locator.port << endl;
            cout << "¥tAgent Host ¥t="
               << descriptors[j].agent_hostname << endl;
            cout << "¥tActivable ¥t="
               << (descriptors[j].activable?"YES":"NO") << endl;
         }
      }
   } catch (const CORBA::Exception& e) {
      cout << "CORBA Exception during execution of find_all: "
         << e << endl;
      return 0;
   }
   return 1;
}
```

コードサンプル 13-4　スマートエージェントが認識するものをすべて検索（Java）

```
// Find.java
public class Find {
public static void main(String[ ] args){
   try {
      // Initialize the ORB.
      org.omg.CORBA.ORB orb =
                     org.omg.CORBA.ORB.init(args,null);
      com.inprise.vbroker.ObjLocation.Agent agent = null;
      try {
         agent =
         com.inprise.vbroker.ObjLocation.AgentHelper.narrow(
         orb.resolve_initial_references("LocationService"));
      } catch (org.omg.CORBA.ORBPackage.InvalidName e){
         System.out.println(
            "Not able to resolve references " +
            "for LocationService");
      System.exit(1);
      } catch (Exception e){
         System.out.println(
            "Not able to resolve references " +
            "for LocationService");
         System.out.println("Caught exception: " + e);
         System.exit(1);
      }
      boolean done=false;
      java.io.BufferedReader in =
         new java.io.BufferedReader(
            new java.io.InputStreamReader(System.in));
      while (!done){
         System.out.print("-> ");
         System.out.flush();
         String line = in.readLine();
         if(line.startsWith("agents")){
            java.lang.String[ ] agentList =
                              agent.all_agent_locations();
            System.out.println(
                 "Located " + agentList.length  + "agents");
            for (int i=0; i < agentList.length; i++){
               System.out.println("¥t" + "Agent #" +
```

```
                (i+1) + ": " + agentList[i]);
            }
          }else if(line.startsWith("rep")){
            java.lang.String[ ] repIds =
                                  agent.all_repository_ids();
            System.out.println("Located " + repIds.length  +
                               "repository Ids");
            for (int i=0; i <repIds.length; i++){
               System.out.println("¥t" + "Repository Id #"
                                  + (i+1) + ": " + repIds[i]);
            }
          } else if(line.startsWith("objects ")){
             String names = line.substring(
                          "objects ".length(), line.length());
             PrintObjects(names,agent,orb);
          } else if(line.startsWith("quit")){
             done = true;
          } else {
             System.out.println("Commands: agents¥n" +
                "          repository_ids¥n" +
                "          objects <rep Id>¥n" +
                "          objects <rep Id><obj name>¥n" +
                "          quit¥n");
          }
       }
    } catch (com.inprise.vbroker.ObjLocation.Fail err){
       System.out.println(
           "Location call failed with reason " + err.reason);
    } catch (java.lang.Exception err){
       System.out.println("Caught error " + err);
       err.printStackTrace();
    }
}
public static void PrintObjects(String names,
               com.inprise.vbroker.ObjLocation.Agent agent,
               org.omg.CORBA.ORB orb)
       throws com.inprise.vbroker.ObjLocation.Fail {
    int space_pos = names.indexOf(' ');
    String repository_id;
    String object_name;
    if (space_pos == -1){
       repository_id = names;
       object_name = null;
    }else {
       repository_id = names.substring(0,names.indexOf(' '));
       object_name = names.substring(names.indexOf(' ') + 1);
    }
    org.omg.CORBA.Object[ ] objects;
    com.inprise.vbroker.ObjLocation.Desc[ ] descriptors;
    if (object_name == null){
       objects = agent.all_instances(repository_id);
       descriptors =
          agent.all_instances_descs(repository_id);
    }else {
       objects =
               agent.all_replica(repository_id,object_name);
               descriptors = agent.all_replica_descs(
                                  repository_id,object_name);
    }
    System.out.println(
                 "Returned " + objects.length  + " objects");
    for (int i=0; i<objects.length; i++){
       System.out.println("¥n¥nObject #" + (i+1) + ":");
       System.out.println("==================");
       System.out.println("¥tRep ID: " +
                          ((com.inprise.vbroker.CORBA.Object)
                               objects[i])._repository_id());
       System.out.println("¥tInstance:" +
                          ((com.inprise.vbroker.CORBA.Object)
                                 objects[i])._object_name());
       System.out.println(
                "¥tIOR: " + orb.object_to_string(objects[i]));
       System.out.println();
       System.out.println("Descriptor #" + (i+1));
```

```
            System.out.println(
                        "====================================");
            System.out.println(
                      "Host:" + descriptors[i].iiop_locator.host);
            System.out.println(
                      "Port:" + descriptors[i].iiop_locator.port);
            System.out.println(
                     "Agent Host:" + descriptors[i].agent_hostname);
            System.out.println(
                   "Repository Id:" + descriptors[i].repository_id);
            System.out.println(
                      "Instance:" + descriptors[i].instance_name);
            System.out.println(
                        "Activable:" + descriptors[i].activable);
        }
    }
    }
```

13.4　トリガーハンドラの記述と登録

ここでは，トリガーのインプリメントおよび登録の方法を示します。

13.4.1　トリガーハンドラのインプリメントと登録

コードサンプル 13-5 および 13-6 では，TriggerHandler のインプリメントと登録をします。TriggerHandlerImpl の impl_is_ready()メソッドと impl_is_down()メソッドは，トリガーを起動する原因となったインスタンスの記述を表示し，オプションとしてそれ自体の登録を解除します。

C++の場合

登録が解除されると，メソッドはプログラムを終了するために CORBA::ORB::shutdown()メソッドを呼び出します。このメソッドは，メインプログラムの impl_is_ready()メソッドを終了するために BOA に転送されます。

Java の場合

登録が解除されると，メソッドはプログラムを終了するために System.exit()メソッドを呼び出します。

TriggerHandlerImpl クラスは，このクラスの生成に使われた desc パラメタと Agent パラメタのコピーを保持していることに注意してください。unreg_trigger()メソッドには desc パラメタが必要です。Agent パラメタは，メインプログラムからのリファレンスが解放された場合に備えて複製されています。

コードサンプル 13-5　トリガーハンドラのインプリメント（C++）

```
// AccountTrigger.c
#include ”locate_s.hh”

// USE_STD_NS is a define set up by VisiBroker to use the
// std namespace USE_STD_NS Instances of this class
// will be called back by the Agent when the
// event for which it is registered happens.

class TriggerHandlerImpl :
      public _sk_ObjLocation::_sk_TriggerHandler
{
   public:
      TriggerHandlerImpl(
           ObjLocation::Agent_var agent,
           const ObjLocation::TriggerDesc& initial_desc)
           : _agent(ObjLocation::Agent::_duplicate(agent)),
           _initial_desc(initial_desc) {}

      void impl_is_ready(const ObjLocation::Desc& desc) {
         notification(desc, 1);
      }
      void impl_is_down(const ObjLocation::Desc& desc){
         notification(desc, 0);
      }

   private:
      void notification(const ObjLocation::Desc& desc,
                CORBA::Boolean isReady){
         if (isReady) {
            cout << ”Implementation is ready:” << endl;
         } else {
            cout << ”Implementation is down:” << endl;
         }
         cout << ”¥tRepository Id =
               ” << desc.repository_id << endl;
         cout << ”¥tInstance Name =
               ” << desc.instance_name << endl;
         cout << ”¥tHost Name =
               ” << desc.iiop_locator.host << endl;
         cout << ”¥tPort =
               ” << desc.iiop_locator.port << endl;
```

```
            cout << "¥tAgent Host =
                  " << desc.agent_hostname << endl;
            cout << "¥tActivable =
                  " << (desc.activable? "YES" : "NO") << endl;
            cout << endl;
            cout << "Unregister this handler and exit (yes/no)? "
                  << endl;
            char prompt [256];
            cin >> prompt;
            if ((prompt[0] == 'y') || (prompt[0] == 'Y')) {
               try {
                  _agent->unreg_trigger(_initial_desc, this);
               }
               catch (const ObjLocation::Fail& e) {
                  cout << "Failed to unregister trigger with
                     reason=[" << (int) e.reason << "]" << endl;
               }
               cout << "exiting..." << endl;
               CORBA::ORB::shutdown();
            }
         }

      private:
         ObjLocation::Agent_var _agent;
         ObjLocation::TriggerDesc _initial_desc;
   };

   int main(int argc,char* const *argv)
   {
      try {
         CORBA::ORB_var the_orb = CORBA::ORB_init(argc, argv);
         CORBA::BOA_var boa = the_orb->BOA_init(argc, argv);
         CORBA::Object_var obj = the_orb->
         resolve_initial_references("LocationService");
         if ( CORBA::is_nil(obj) ) {
            cout << "Unable to locate initial LocationService"
                  << endl;
            return 0;
         }
         ObjLocation::Agent_var the_agent =
               ObjLocation::Agent::_narrow(obj);

         // Create the trigger descriptor to notify us about
         // osagent changes with respect to Account objects
         ObjLocation::TriggerDesc desc;
         desc.repository_id = (const char*)
               "IDL:Bank/AccountManager:1.0";
         desc.instance_name = (const char*) "";
         desc.host_name = (const char*) "";

         ObjLocation::TriggerHandler_var trig =
               new TriggerHandlerImpl(the_agent, desc);
         boa->obj_is_ready(trig);
         the_agent->reg_trigger(desc,trig);
         boa->impl_is_ready();
      }
      catch (const CORBA::Exception& e) {
         cout << "account_trigger caught Exception: "
               << e << endl;
         return 0;
      }
      return 1;
   }
```

コードサンプル 13-6　トリガーハンドラのインプリメント（Java）

```
   // AccountTrigger.java

   import java.io.*;
   import org.omg.PortableServer.*;

   class TriggerHandlerImpl extends
         com.inprise.vbroker.ObjLocation.TriggerHandlerPOA {
      public TriggerHandlerImpl(
         com.inprise.vbroker.ObjLocation.Agent agent,
```

```
      com.inprise.vbroker.ObjLocation.TriggerDesc initial_desc){
         agent = agent;
         initial_desc = initial_desc;
   }
   public void impl_is_ready(
                com.inprise.vbroker.ObjLocation.Desc desc){
                                  notification(desc, true);
   }
   public void impl_is_down(
                 com.inprise.vbroker.ObjLocation.Desc desc){
                                   notification(desc,false);
   }

   private void notification(
     com.inprise.vbroker.ObjLocation.Desc desc,
     boolean isReady){
      if (isReady){
         System.out.println("Implementation is ready:");
      } else {
         System.out.println("Implementation is down:");
      }
      System.out.println(
       "¥tRepository Id = " + desc.repository_id + "¥n" +
       "¥tInstance Name = " + desc.instance_name + "¥n" +
       "¥tHost Name     = " + desc.iiop_locator.host + "¥n" +
       "¥tActivable     = " + desc.activable + "¥n" + "¥n");
      System.out.println(
              "Unregister this handler and exit (yes/no)?");
      try {
         BufferedReader in = new BufferedReader(
                          new InputStreamReader(System.in));
         String line = in.readLine();
         if(line.startsWith("y") || line.startsWith("Y")) {
           try {
              agent.unreg_trigger(_initial_desc, _this());
           } catch (com.inprise.vbroker.ObjLocation.Fail e){
              System.out.println(
                "Failed to unregister trigger with reason=[" +
                e.reason + "]");
           }
            System.out.println("exiting...");
            System.exit(0);
           }
      } catch (java.io.IOException e){
         System.out.println(
               "Unexpected exception caught: " + e);
         System.exit(1);
      }
   }

   private com.inprise.vbroker.ObjLocation.Agent _agent;
   private com.inprise.vbroker.ObjLocation.TriggerDesc
                                            _initial_desc;
}

public class AccountTrigger {

   public static void main(String args[ ]){
      try {
         // Initialize the ORB.
         org.omg.CORBA.ORB orb =
                          org.omg.CORBA.ORB.init(args,null);
         POA rootPoa = POAHelper.narrow(
                 orb.resolve_initial_references("RootPOA"));
         rootPoa.the_POAManager().activate();
         com.inprise.vbroker.ObjLocation.Agent the_agent =
         com.inprise.vbroker.ObjLocation.AgentHelper.narrow(
         orb.resolve_initial_references("LocationService"));
         // Create a trigger description and an appropriate
         // TriggerHandler. The TriggerHandler will be
         // invoked when the osagent become aware of any
         // new implementations of the interface
         // "Bank::AccountManager"
         com.inprise.vbroker.ObjLocation.TriggerDesc desc =
```

```
            new com.inprise.vbroker.ObjLocation.TriggerDesc(
                        "IDL:Bank/AccountManager:1.0", "", "");
         TriggerHandlerImpl trig =
                     new TriggerHandlerImpl(the_agent, desc);
         rootPoa.activate_object(trig);
         the_agent.reg_trigger(desc, trig._this());
         orb.run();
      }catch (Exception e){
         e.printStackTrace();
         System.exit(1);
      }
   }
}
```

14 ネーミングサービスの使用

この章では OMG の「インターオペラブルネーミング仕様」ドキュメント（formal/02-09-02）の完全なインプリメンテーションである Borland Enterprise Server VisiBroker ネーミングサービスの使用方法について説明します。

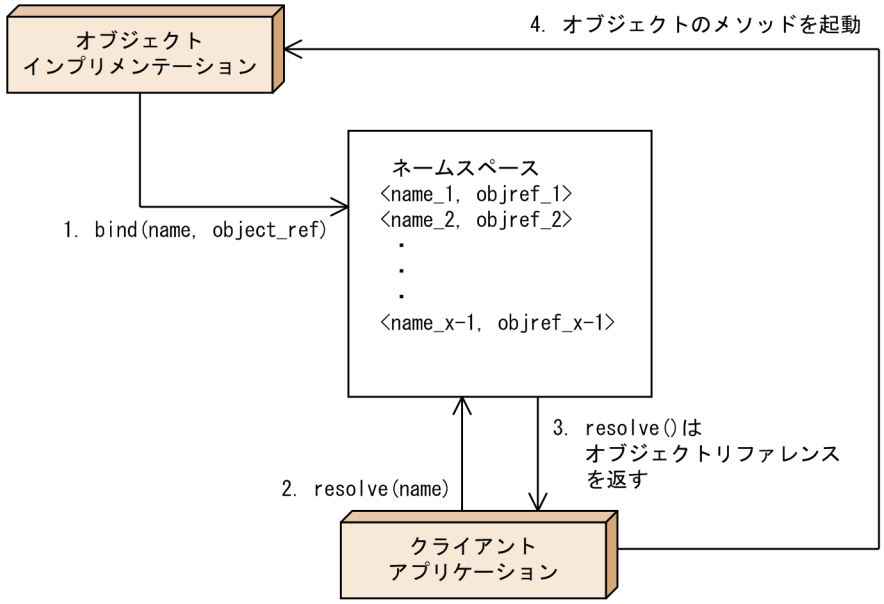
14.1　概要

ネーミングサービスによって，一つ以上の論理名称を一つのオブジェクトリファレンスに対応させ，これらの名前をネームスペースに格納できるようになります。また，クライアントアプリケーションはネーミングサービスを使用して，オブジェクトに割り当てられた論理名称を使用してオブジェクトリファレンスを取得できます。

図 14-1 に，次の内容のネーミングサービス簡略図を示します。

1. オブジェクトインプリメンテーションが，どのようにして名前をネームスペース内のオブジェクトの一つにバインドできるか。
2. クライアントアプリケーションが，ネーミングコンテキストまたはオブジェクトにオブジェクトリファレンスを返す名前を解決するために，どのように同じネームスペースを使用できるか。

図 14-1　ネームスペース内のネーミングコンテキストからのオブジェクト名のバインド，解決，使用



Borland Enterprise Server VisiBroker ネーミングサービスを使用してオブジェクトインプリメンテーションを探す場合は，スマートエージェントを使用した場合と比べて考慮しなくてはならない相違点が幾つかあります。

- スマートエージェントはフラットなネームスペースを使用しますが，ネーミングサービスは階層形式のものを使用します。
- C++の場合に，スマートエージェントを使用するとき，オブジェクトのインタフェース名は，クライアントアプリケーションとサーバアプリケーションのコンパイル時に定義されます。インタフェース名を変更するにはアプリケーションを再度コンパイルしなければなりません。これとは対照的に，ネーミングサービスでは，オブジェクトインプリメンテーションはランタイム時に論理名称とオブジェクトをバインドできます。
- C++の場合に，スマートエージェントを使用するとき，オブジェクトは一つのインタフェース名しか実装できませんが，ネーミングサービスによって，一つ以上の論理名称を一つのオブジェクトにバインドできます。
- Java の場合，オブジェクトのインタフェース名は，クライアントアプリケーションとサーバアプリケーションのコンパイル時に定義されます。インタフェース名を変更するにはアプリケーションを再度コ

ンパイルしなければなりません。これとは対照的に，ネーミングサービスでは，オブジェクトインプリメンテーションはランタイム時に論理名称とオブジェクトにバインドできます。

- Java の場合，オブジェクトは一つのインタフェース名しかインプリメントできませんが，ネーミングサービスによって，一つ以上の論理名称を一つのオブジェクトにバインドできます。

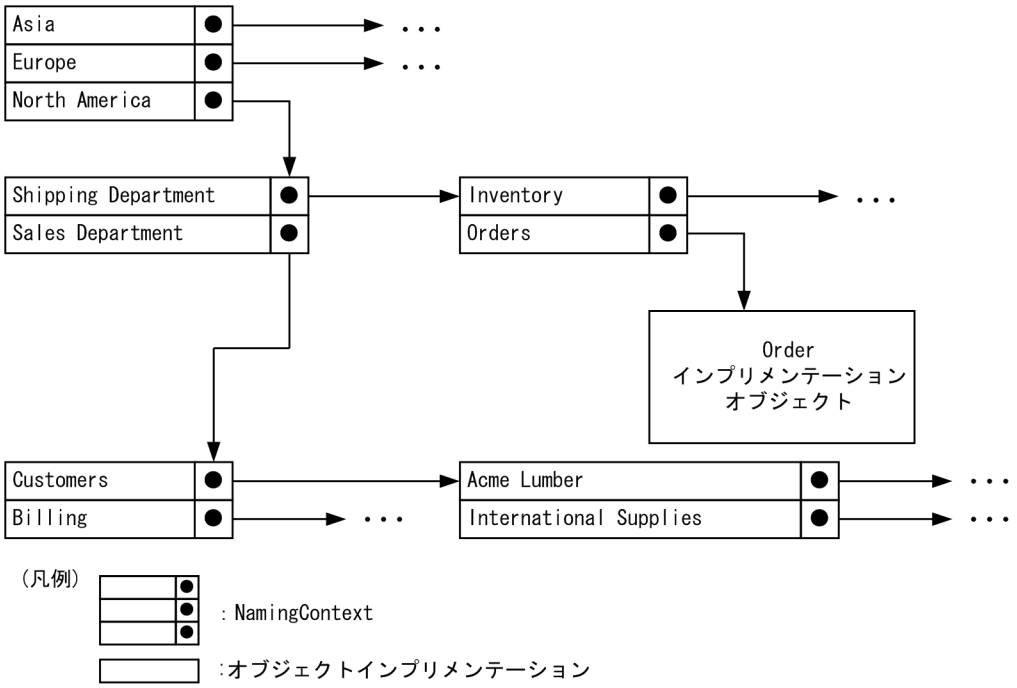
次の OS では，ネーミングサービスを使用する場合，実行方法は vbj を使用せず，nameserv プログラムを使用して起動させてください。

- Windows Vista
- Windows 7
- Windows Server 2008
- Windows Server 2008 R2

14.2　ネームスペースの解説

オーダーエントリシステムを構成する各オブジェクトに名前を付けるためにネーミングサービスをどのように使用できるかを，図 14-2 に示します。この階層オーダーエントリシステムのネームスペースは地域別，部署別などで構成されます。ネーミングサービスを使用すれば，特定名称を探す場合の検索対象にできる NamingContext オブジェクトの階層構造で，ネームスペースを構成できます。例えば，論理名称 NorthAmerica/ShippingDepartment/Orders を使用して Order オブジェクトを探せます。

図 14-2　オーダーエントリシステムのネーミング手法



14.2.1　ネーミングコンテキスト

Borland Enterprise Server VisiBroker ネーミングサービスを使用して，図 14-2 に示したネームスペースをインプリメントするには，それぞれを NamingContext オブジェクトでインプリメントする必要があります。NamingContext オブジェクトには，オブジェクトインプリメンテーションに，またはほかの NamingContext オブジェクトにバインドされた Name 構造体のリストが入っています。論理名称は NamingContext にバインドできますが，デフォルトでは NamingContext は対応する論理名称を持たず，またそのような名前も不要であるということを認識しておくことが大切です。

オブジェクトインプリメンテーションは NamingContext オブジェクトを使用して，提供するオブジェクトに名前をバインドします。クライアントアプリケーションは NamingContext を使用して，バインドされた名前をオブジェクトリファレンスへと解決します。

文字列化された名前を使用する場合に必要なメソッドを提供する NamingContextExt インタフェースも使用できます。

14.2.2　ネーミングコンテキストファクトリ

ネーミングコンテキストファクトリは，ネーミングサービスに接続するインタフェースを提供します。このファクトリはネーミングサービスを終了させたり，コンテキストがない場合に新しいコンテキストを生成す

るオペレーションを持ちます。各ファクトリは，ルートコンテキストを返す API も持っています。ルートコンテキストは情報を照会するために重要な役目を果たします。これは，参照可能なすべてのデータを格納するためのルートです。

Borland Enterprise Server VisiBroker ネーミングサービスには，デフォルトネーミングコンテキストファクトリと拡張ネーミングコンテキストファクトリという二つのクラスが提供され，これによってネームスペースが生成できるようになります。デフォルトネーミングコンテキストファクトリは，ルート NamingContext を持たない空のネームスペースを生成します。拡張ネーミングコンテキストファクトリはルート NamingContext 付きのネームスペースを生成するので，こちらのネーミングコンテキストファクトリを使用することをお勧めします。

オブジェクトインプリメンテーションが名前をオブジェクトにバインドしたり，クライアントアプリケーションが名前をオブジェクトリファレンスへと解決したりするには，これらの NamingContext オブジェクトを最低一つは取得しなければなりません。

図 14-2 に示した NamingContext オブジェクトは，五つすべてを一つのネームサービスプロセス内でインプリメントすることもできるし，最大で五つのネームサービスプロセス内で別々にインプリメントすることもできます。

14.2.3　Name と NameComponent

CosNaming::Name は，オブジェクトインプリメンテーションまたは CosNaming::NamingContext にバインドできる識別子を表します。Name はただの英数文字列ではなく，一つ以上の NameComponent 構造体のシーケンスです。

各 NameComponent には，id と kind という二つの属性文字列が含まれます。それぞれの id と kind が任意の NamingContext 内で一意であることを確認する場合以外は，ネーミングサービスはこれらの文字列の解釈や管理はしません。

id 属性と kind 属性は，名前がバインドされているオブジェクトを一意に識別するための文字列です。kind メンバは名前に詳細情報を付加します。例えば，「Inventory.RDBMS」という名前は「Inventory」という id メンバと「RDBMS」という kind メンバを持ちます。

IDL サンプル 14-1　NameComponent 構造体の IDL 指定

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

NameComponent の id 属性と kind 属性は，NULL 文字（0x00）とそのほかの印刷不能文字を除いた ISO 8859-1（Latin-1）文字セットの文字でなければなりません。NameComponent に指定する文字列は，どちらも 255 文字を超えてはいけません。さらに，ネーミングサービスはワイド文字を使用する NameComponent はサポートしません。

注

空文字列は Name の id 属性には指定できませんが，kind 属性には指定できます。

14.2.4　ネーム解決

クライアントアプリケーションは，NamingContext メソッド resolve を使用して，論理的な Name が与えられたオブジェクトリファレンスを取得します。Name は一つ以上の NameComponent オブジェクトで構成されているので，解決プロセスは，Name を構成するすべての NameComponent 構造体を調べなければなりません。

(1)　文字列化された名前

CosNaming::Name 識別子は可読性やデータ交換に向かない形式です。この問題を解決するために，文字列化された名前が定義されています。文字列化された名前とは，文字列と CosNaming::Name との一対一のマッピングです。二つの CosNaming::Name オブジェクトが等しければ，その文字列化された表記も等しく，これは逆の場合にも当てはまります。文字列化された名前では，スラント（/）はネームコンポーネントのセパレータとして，ピリオド（.）は id 属性と kind 属性のセパレータとして，¥記号はエスケープ文字としてそれぞれ使用されます。規定によって，空の kind 属性を持った NameComponent はピリオドを使用しません（Order など）。

コードサンプル 14-1　文字列化された名前の例

```
"Borland.Company/Engineering.Department/Printer.Resource"
```

注

以降のサンプルでは，NameComponent 構造体は文字列化された表記で示されます。

(2)　単純名と複合名

Billing のような単純名は NameComponent を一つしか持たないで，常にターゲットのネーミングコンテキスト上で解決されます。単純名はオブジェクトインプリメンテーションまたは NamingContext にバインドできます。

NorthAmerica/ShippingDepartment/Inventory のような複合名は三つの NameComponent 構造体のシーケンスで構成されます。n 個の NameComponent オブジェクトで構成される複合名をオブジェクトインプリメンテーションにバインドすると，シーケンス内の最初の（n-1）個の NameComponent オブジェクトはそれぞれ NamingContext へと解決され，最後の NameComponent オブジェクトはオブジェクトインプリメンテーションへと解決されなければなりません。

Name を NamingContext にバインドすると，シーケンス内の各 NameComponent 構造体は NamingContext を参照しなければなりません。

コードサンプル 14-2 は，三つのコンポーネントで構成され，CORBA オブジェクトにバインドされた複合名を示します。この名前に相当する文字列化された名前は NorthAmerica/SalesDepartment/Order です。いちばん上のネーミングコンテキスト内で解決されると，この複合名の最初の二つのコンポーネントは NamingContext オブジェクトへと解決され，最後のコンポーネントは論理名称「Order」を持ったオブジェクトインプリメンテーションへと解決されます。

コードサンプル 14-2　VisiBroker ORB オブジェクトにバインドされた複合名の例（C++）

```
. . .
// Name stringifies to "NorthAmerica/SalesDepartment/Order"
CosNaming::Name_var continentName =
      rootNamingContext->to_name("NorthAmerica");
CosNaming::NamingContext_var continentContext =
      rootNamingContext->bind_new_context(continentName);
CosNaming::Name_var departmentName =
      continentContext->to_name("SalesDepartment");
CosNaming::NamingContext_var departmentContext =
      continentContext->bind_new_context(departmentName);
CosNaming::Name_var objectName =
```

```
      departmentContext->to_name("Order");
departmentContext->rebind(objectName, myPOA->
      servant_to_reference(managerServant));
. . .
```

コードサンプル 14-3　VisiBroker ORB オブジェクトにバインドされた複合名の例（Java）

```
. . .
// Name stringifies to "NorthAmerica/SalesDepartment/Order "
NameComponent[ ] continentName =
                    { new NameComponent("NorthAmerica", "")};
NamingContext continentContext =
          rootNamingContext.bind_new_context(continentName);
NameComponent[ ] departmentName =
                  { new NameComponent("SalesDepartment", "")};
NamingContext departmentContext =
          continentContext.bind_new_context(departmentName);
NameComponent[ ] objectName = { new NameComponent("Order", "")};
departmentContext.rebind(
     objectName, myPOA.servant_to_reference(managerServant));
. . .
```

14.3　ネーミングサービスの実行

ネーミングサービスは次に示すコマンドで起動できます。

14.3.1　ネーミングサービスのインストール

Borland Enterprise Server VisiBroker をインストールすると，ネーミングサービスが自動的にインストールされます。ネーミングサービスは，Windows ではバイナリの実行形式ファイル，UNIX ではスクリプトである nameserv ファイルと，vbjorb.jar ファイルに格納されている Java クラスファイルで構成されています。

14.3.2　ネーミングサービスの設定

旧バージョンの VisiBroker Naming Service では，ネーミングサービスは変更のあったすべてのオペレーションをフラットファイルへロギングすることによってパーシステンスを維持しました。バージョン 4.0 以降は，ネーミングサービスはバッキングストアアダプタと連携動作します。すべてのバッキングストアアダプタがパーシステンスをサポートしているわけではないことに注意してください。デフォルトの InMemory アダプタは非パーシステントですが，ほかのアダプタはパーシステントです。アダプタの詳細については，「14.10　プラガブルバッキングストア」を参照してください。

注

ネーミングサーバは，起動時に自分自身をスマートエージェントに登録しなければなりません。したがって，ネーミングサービスを開始するためにスマートエージェントを先に実行しておく必要があります。これによって，クライアントは resolve_initial_references メソッドを呼び出すことでイニシャルルートコンテキストを検索できます。解決機能は，必要なリファレンスの検索のためにスマートエージェントを通して動作します。同様に，仕組みに加わっている各ネーミングサーバも，同じ機能を使用して仕組みをセットアップします。

14.3.3　ネーミングサービスの起動

bin ディレクトリにある nameserv プログラムを使用してネーミングサービスを起動できます。nameserv は，デフォルトでは com.inprise.vbroker.naming.ExtFactory ファクトリクラスを使用します。ネーミングサービスのオプションを表 14-1 に示します。

UNIX

```
nameserv [driver_options] [nameserv_options] <ns_name> &
```

Windows

```
start nameserv [driver_options] [nameserv_options] <ns_name>
```

表 14-1　ネーミングサービスのオプション

オプション		説明
driver_options （ファクトリ名の前に表示されなければなりません）	-J<Java option>	指定されたオプションを JavaVM に直接渡します。
	-VBJversion	Borland Enterprise Server VisiBroker のバージョン番号を出力します。
	-VBJdebug	Borland Enterprise Server VisiBroker のデバッグ情報を出力します。

オプション		説明
nameserv_options	-?，-h，-help，-usage	使用情報を出力します。
	-config <properties_file>	ネーミングサービス起動時に<properties_file>を構成ファイルとして使用します。このオプションで指定したファイル内では，「#」で始まる行はコメント行となり，無視されます。
<ns_name>		このネーミングサービスで使用する名前です。これは省略できます。デフォルト名はNameService です。

(1)　vbj によるネーミングサービスの起動（Java）

ネーミングサービスは vbj を使用しても起動できます。

```
prompt>vbj com.inprise.vbroker.naming.ExtFactory <ns_name>
```

14.4　コマンドラインからのネーミングサービスの呼び出し

ネーミングサービスユーティリティ（nsutil）は，コマンドラインにバインディングを格納したり，コマンドラインからバインディングを検索したりする機能を提供します。

14.4.1　nsutil の構成

nsutil を使用するには，まず次のどちらかを使用してネーミングサービスインスタンスを構成します。nsutil のオプションを表 14-2 に示します。

```
prompt>nameserve <factory_name>
```

または

```
prompt>nsutil -VBJprop <ns_config> <cmd>[args]
```

表 14-2　nsutil のオプション

オプション	説明
ns_config SVCnameroot=<factory_name> ORBInitRef=NameService=<url>	ファクトリ名を定義します。 注 SVCnameroot を使用するには，まず osagent を実行しなければなりません。 corbaloc:，corbaname:，file:，ftp:，http:，または ior:のような型を示すプリフィクスを付けたファイル名または URL。例えば，ローカルディレクトリにファイルを割り当てるには，ns_config 文字列は次のようになります。 -VBJprop ORBInitRef=NameService=<file:ns.ior>
cmd	ping と shutdown を加えた CosNaming オペレーションです。

14.4.2　nsutil の実行

ネーミングサービスユーティリティはすべての CosNaming オペレーションと二つの追加コマンドをサポートします。サポートしている CosNaming オペレーションを表 14-3 に示します。

表 14-3　nsutil でサポートしている CosNaming オペレーション

cmd	パラメタ
bind	name objRef
bind_context	name objRef
bind_new_context	name ctxRef
destroy	name
list	name*
new_context	なし
rebind	name objRef
rebind_context	name ctxRef

cmd	パラメタ
resolve	name
unbind	name

追加の nsutil コマンドを表 14-4 に示します。表 14-4 のコマンドを実行するにはスマートエージェントを起動する必要があります。スマートエージェントの起動については,「4.7.1　スマートエージェントの起動」を参照してください。

表 14-4　nsutil の追加コマンド

cmd	パラメタ	説明
ping	name	文字列化された name を解決し, オブジェクトが活性であるかどうかを調べるためにオブジェクトとコンタクトします。
shutdown	factory_name	コマンドラインから正しい手順でネーミングサービスをシャットダウンします。 factory_name はネーミングサービスの起動時に指定した名前です。 注 このコマンドの起動では初期コンテキストを設定しておく必要はありません。

nsutil コマンドからオペレーションを実行するには，オペレーション名とそのパラメタを<cmd>パラメタとして設定してください。例を次に示します。

```
prompt>nsutil -VBJprop ORBInitRef=NameService=file://ns.¥
       ior resolve myName <factory_name>
```

14.4.3　nsutil のクローズ

nsutil をクローズするには shutdown コマンドを使用してください。

```
prompt>nsutil -VBJprop ORBInitRef=NameService=file://ns.¥
       ior shutdown
```

14.5　ネーミングサービスへの接続

指定されたネーミングサービスからイニシャルオブジェクトリファレンスを取得できるようにクライアントアプリケーションを起動するには，3とおりの方法があります。ネーミングサービスを起動するときに，次の三つのコマンドラインオプションを使用できます。

- ORBInitRef
- ORBDefaultInitRef
- DSVCnameroot

14.5.1　resolve_initial_references の呼び出し

新規のネーミングサービスが提供する単純な機能を使用すれば，共通ネーミングコンテキストを返すようにresolve_initial_references メソッドを設定できます。クライアントプログラムが接続しようとするネーミングサーバのルートコンテキストを返す resolve_initial_references メソッドを使用してください。これらの三つのオプションの使い方を，三つの簡単な例で説明します。TestHost というホストで実行中の三つの Borland Enterprise Server VisiBroker ネーミングサービス（ns1，ns2，ns3）があり，三つのサーバアプリケーション（sr1，sr2，sr3）がそれぞれホスト TestHost の別々のポート（20001，20002，20003）で実行中の場合，サーバ sr1 は ns1 に，sr2 は ns2 に，sr3 は ns3 にバインドします。

コードサンプル 14-4　ルートネーミングコンテキストの取得方法を示すコードの抜粋部分（C++）

```
. . .
   CORBA::ORB_ptr orb = CORBA::ORB_init(argv, argc, NULL);
   CORBA::Object_var rootObj =
         orb->resolve_initial_references("NameService");
. . .
```

コードサンプル 14-5　ルートネーミングコンテキストの取得方法を示すコードの抜粋部分（Java）

```
. . .
   org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
   org.omg.CORBA.Object rootObj =
               orb.resolve_initial_references("NameService");
. . .
```

14.5.2　-DSVCnameroot の使用

-DSVCnameroot オプションを使用して，どの Borland Enterprise Server VisiBroker ネーミングサービスインスタンスへ接続したいかを指定できます(互いに無関係のネーミングサービスを複数実行している場合は特に重要です)。例えば，ns1 へ接続したい場合は，次のようにクライアントアプリケーションを起動してください。

C++の場合

```
<client_application> -DSVCnameroot=ns1
```

Java の場合

```
vbj -DSVCnameroot=ns1 <client_application>
```

これで，コードサンプル 14-4 および 14-5 で示したように，クライアントアプリケーション内部で ORB リファレンスの resolve_initial_references メソッドを呼び出すことによって，ns1 のルートコンテキストを取得できます。

注

このオプションを使用するには，osagent が実行中でなければなりません。

-DSVCnameroot オプションは osagent の機能に基づいており，ほかの CORBA インプリメンテーションとのインターオペラビリティがないことに注意してください。

14.5.3　-ORBInitRef（C++）および-DORBInitRef（Java）の使用

corbaloc URL または corbaname URL ネーミング機能を使用して，どの Borland Enterprise Server VisiBroker ネーミングサービスへ接続したいかを指定できます。

(1)　corbaloc URL の使用

ネーミングサービス ns2 を使用して接続したい場合は，次のようにクライアントアプリケーションを起動してください。

C++の場合

```
<client_application> -ORBInitRef NameService=
      corbaloc::TestHost:20002/NameService
```

Java の場合

```
vbj -DORBInitRef=NameService=
   corbaloc::TestHost:20002/NameService <client_application>
```

これで，コードサンプル 14-4 および 14-5 で示したように，クライアントアプリケーション内部で VisiBroker ORB リファレンスの resolve_initial_references メソッドを呼び出すことによって，ns2 のルートコンテキストを取得できます。

注

- このサンプルは，アクセスしたいネーミングサービスにバインドされたポート 20002 で実行中のサーバがある場合だけ有効です。
- ORBInitRef を使用し，corbaloc を指定した場合，URL 最後の NameService 指定がないと，ArrayIndexOutOfBoundsException 例外が発生します。

(2)　corbaname URL の使用

corbaname を使用して ns3 へ接続したい場合は，次のようにクライアントプログラムを起動してください。

C++の場合

```
<client_application> -ORBInitRef NameService=
      corbaname::TestHost:20003/
```

Java の場合

```
vbj -DORBInitRef=NameService=corbaname::TestHost:20003/ <client_application>
```

これで，コードサンプル 14-4 および 14-5 で示したように，クライアントアプリケーション内部で VisiBroker ORB リファレンスの resolve_initial_references メソッドを呼び出すことによって，ns3 のルートコンテキストを取得できます。

14.5.4　-ORBDefaultInitRef（C++）および-DORBDefaultInitRef（Java）

corbaloc URL または corbaname URL を使用して，どの Borland Enterprise Server VisiBroker ネーミングサービスへ接続したいかを指定できます。

(1)　-ORBDefaultInitRef（C++）または-DORBDefaultInitRef（Java）と corbaloc URL の使用

ns2 へ接続したい場合は，次のようにクライアントプログラムを起動してください。

C++の場合

```
<client_application> -ORBDefaultInitRef corbaloc::TestHost:20002
```

Java の場合

```
vbj -DORBDefaultInitRef=corbaloc::TestHost:20002 <client_application>
```

これで，コードサンプル 14-4 および 14-5 で示したように，クライアントアプリケーション内部で VisiBroker ORB リファレンスの resolve_initial_references メソッドを呼び出すことによって，ns2 のルートコンテキストを取得できます。

(2)　-ORBDefaultInitRef（C++）または-DORBDefaultInitRef（Java）と corbaname の使用

-ORBDefaultInitRef（C++）または-DORBDefaultInitRef（Java）と corbaname を組み合わせると，予期したものとは異なる動作をします。-ORBDefaultInitRef（C++）または-DORBDefaultInitRef（Java）を指定すると，スラントと文字列化された，オブジェクト key が常に corbaname に追加されます。

C++の場合

例えば，URL（corbaname::TestHost:20002）と指定してから-ORBDefaultInitRef と指定すると，resolve_initial_references（C++）の結果は新しい URL（corbaname::TestHost:20003/NameService）になります。

Java の場合

例えば，URL（corbaname::TestHost:20002）と指定してから-DORBDefaultInitRef と指定すると，resolve_initial_references（Java）の結果は新しい URL（corbaname::TestHost:20003/NameService）になります。

14.6　NamingContext

このオブジェクトは，VisiBroker ORB オブジェクトまたはほかの NamingContext オブジェクトにバインドされている名前のリストを保持し操作するために使用します。クライアントアプリケーションは，このインタフェースを使用して，そのコンテキスト内のすべての名前を解決またはリスト出力します。オブジェクトインプリメンテーションは，このオブジェクトを使用して複数の名前をオブジェクトインプリメンテーションにバインドしたり一つの名前を NamingContext オブジェクトにバインドしたりします。IDL サンプル 14-2 に，NamingContext の IDL 指定を示します。

IDL サンプル 14-2　NamingContext インタフェースの指定

```
module CosNaming {
   interface NamingContext {
      void bind(in Name n,in Object obj)
         raises(NotFound, CannotProceed,
                InvalidName, AlreadyBound);
      void rebind(in Name n, in Object obj)
         raises(NotFound, CannotProceed, InvalidName);
      void bind_context(in Name n, in NamingContext nc)
         raises(NotFound, CannotProceed,
                InvalidName, AlreadyBound);
      void rebind_context(in Name n,in NamingContext nc)
         raises(NotFound, CannotProceed, InvalidName);
      Object resolve(in Name n)
         raises(NotFound, CannotProceed, InvalidName);
      void unbind(in Name n)
         raises(NotFound, CannotProceed, InvalidName);
      NamingContext new_context();
      NamingContext bind_new_context(in Name n)
         raises(NotFound, CannotProceed, InvalidName,
                AlreadyBound);
      void destroy()
         raises(NotEmpty);
      void list(in unsigned long how_many,
            out BindingList bl,
            out BindingIterator bi);
   };
};
```

14.7　NamingContextExt

NamingContextExt インタフェースは NamingContext の拡張であり，文字列化された名前と URL を使用する場合に必要なオペレーションを提供します。

IDL サンプル 14-3　NamingContextExt インタフェースの指定

```
module CosNaming {
   interface NamingContextExt : NamingContext {
      typedef string StringName;
      typedef string Address;
      typedef string URLString;

      StringName to_string(in Name n)
         raises(InvalidName);
      Name to_name(in StringName sn)
         raises(InvalidName);

      exception InvalidAddress {};
      URLString to_url(in Address addr, in StringName sn)
         raises(InvalidAddress, InvalidName);
      Object resolve_str(in StringName n)
         raises(NotFound, CannotProceed, InvalidName);
   };
};
```

14.8　デフォルトネーミングコンテキスト

クライアントアプリケーションはデフォルトネーミングコンテキストを指定できますが, これはアプリケーションが自分のルートコンテキストとみなすネーミングコンテキストです。デフォルトネーミングコンテキストは，このクライアントアプリケーションに対してだけルートであり，実際にはほかのコンテキストに入っていてもかまいません。

14.8.1　デフォルトコンテキストの取得（C++）

VisiBroker ORB の resolve_initial_references メソッドを使用することによって，クライアントアプリケーションはデフォルトのネーミングコンテキストを取得できます。デフォルトのネーミングコンテキストは, クライアントアプリケーションの開始時に ORBInitRef コマンドライン引数を渡すことで指定しておく必要があります。コードサンプル 14-6 に，C++クライアントアプリケーションがどのようにこのメソッドを起動するかを示します。

コードサンプル 14-6　resolve_initial_references メソッドの起動

```
#include "CosNaming_c.hh"
. . .
int main(int argc, char* const* argv) {
   try {
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      . . .
      CORBA::Object_var ref =
            orb->resolve_initial_references("NameService");
      CosNaming::NamingContext_var rootContext =
         CosNaming::NamingContext::_narrow(ref);
      . . .
   } catch(const CORBA::Exception& e) {
      cout << "Failure: " << e << endl;
      exit(1);
   }
   exit(0);
}
```

14.8.2　デフォルトネーミングコンテキストの取得（Java）

ORB インタフェースの resolve_initial_references メソッドを使用することによって，Java クライアントアプリケーションをネーミングサービスに接続できます。この機能を使用するには, クライアントの起動時に SVCnameroot パラメタを指定しなければなりません。

例えば，ネーミングコンテキスト Inventory を自分のデフォルトネーミングコンテキストとして使用することになっている ClientApplication という Java アプリケーションを起動するには，次のコマンドを入力できます。

```
prompt> vbj -DSVCnameroot=¥
        NorthAmerica/ShippingDepartment/Inventory    ¥
        ClientApplication
```

このサンプルで，NorthAmerica はサーバ名，ShippingDepartment/Inventory はルートコンテキストから文字列化された名前です。

注

vbj コマンドを使用するときは，すべての-D プロパティを Java クラス名の前に指定しなければなりません。

14.9　ネーミングサービスプロパティ

ネーミングサービスプロパティを表 14-5 に示します。

表 14-5　ネーミングサービスプロパティ

プロパティ	デフォルト	説明
vbroker.naming.adminPwd	inprise	VisiBroker 管理ネーミングサービスオペレーションで必要なパスワードです。
vbroker.naming.enableClusterFailover	true	true に設定すると，ネーミングサービスから取得されたオブジェクトのフェールオーバーを処理するインタセプタをインストールすることを指定します。オブジェクト障害の場合，元のクラスタと同じクラスタから別のオブジェクトに透過的に再接続するよう試みます。
vbroker.naming.enableSlave	0	true の場合，マスタ／スレーブネーミングサービス構成を有効にします。マスタ／スレーブネーミングサービスの構成については，「14.12　フェールオーバー」を参照してください。
vbroker.naming.iorFile	ns.ior	ネーミングサービス IOR を格納する完全パス名を指定します。このプロパティを設定しないと，ネーミングサービスはカレントディレクトリの ns.ior という名前のファイルに IOR を出力しようとします。IOR を出力しようとすると，ネーミングサービスはファイルアクセス権限例外を透過的に無視します。
vbroker.naming.LogLevel	emerg	ネーミングサービスから出力されるログメッセージのレベルを指定します。
vbroker.naming.propBindOn	0	true の場合は，暗黙的なクラスタリング機能がオンになります。
vbroker.naming.smrr.pruneStaleRef	1	このプロパティは，ネームサービスクラスタがスマートラウンドロビン方法を使用する場合に関連してきます。このプロパティに 1 を設定した場合，以前スマートラウンドロビン方法でクラスタにバインドされた古いオブジェクトリファレンスをネームサービスが探索すると，それがバインディングから削除されます。このプロパティに 0 を設定した場合，クラスタ下の古いオブジェクトリファレンスバインディングは削除されません。ただし，スマートラウンドロビン方法を用いたクラスタは，常に，vbroker.naming.smrr.pruneStaleRef プロパティの値に関係なく，このようなオブジェクトバインディングがある場合に，resolve()メソッドまたは select()メソッド呼び出し時にアクティブなオブジェクトリファレンスを戻します。デフォルトでは，ネームサービス 4.5 の暗黙的なクラスタリングはプロパティ値を 1 に設定してスマートラウンドロビン方法を使用します。

14.10　プラガブルバッキングストア

旧バージョンのネーミングサービスでは，ネームスペース（つまりネーミングコンテキストとオブジェクト－名前のバインディングの集合）をメモリに保存し，変更できるすべてのオペレーションをネームスペースからロギングファイルへロギングしました。そして，前回のネームスペースを再生成するためにネーミングサービスを起動するときに，このフラットファイルを使用できました。

現在のネーミングサービスは，プラガブルバッキングストアを使用してネームスペースを維持します。ネームスペースがパーシステントかどうかは，どのようにバッキングストアを設定するか，つまり JDBC アダプタ，Java Naming and Directory Interface（LDAP 用に認証された JNDI），デフォルトのインメモリアダプタのどれを使用するかに左右されます。

14.10.1　バッキングストアのタイプ

サポートしているバッキングストアアダプタは 4 タイプあります。

- インメモリアダプタ
- リレーショナルデータベースの JDBC アダプタ
- DataExpress アダプタ
- JNDI（LDAP だけ）アダプタ

注

プラガブルバッキングストアアダプタの使用例については，Borland Enterprise Server VisiBroker をインストールしたディレクトリの examples/vbe/ins/pluggable_adaptors 内のコードを参照してください。

(1)　インメモリアダプタ

インメモリアダプタは，ネームスペース情報をメモリに保管し，パーシステントではありません。これはネーミングサービスがデフォルトで使用するアダプタです。

(2)　JDBC アダプタ

リレーショナルデータベースは JDBC を介してサポートされます。次に示すデータベースがネーミングサービス JDBC アダプタの処理用に認証されています。

- JDataStore
- Oracle
- Sybase
- Microsoft SQL Server
- DB2
- Interbase

(3)　DataExpress アダプタ

二つの JDBC アダプタに加えて，JDataStore データベースにネイティブにアクセスできるようにする DataExpress アダプタがあります。JDBC を介して JDataStore にアクセスするよりも速いのですが，DataExpress アダプタには幾つかの制限事項があります。このアダプタはネーミングサーバと同じマシン

で実行中のローカルデータベースだけをサポートします。リモート JDataStore データベースにアクセスするには，JDBC アダプタを使用する必要があります。

(4)　JNDI アダプタ

JNDI アダプタもサポートされています。Sun の JNDI は，企業全体の複数のネーミングおよびディレクトリサービスとの標準インタフェースを提供します。JNDI は，別々のネーミングベンダとサービスベンダが準拠しなければならない SPI（サービスプロバイダインタフェース）を持ちます。Netscape LDAP サーバ，Novell NDS，WebLogic Tengah などで別々の SPI モジュールを使用できます。JNDI をサポートすることによって，Borland Enterprise Server VisiBroker ネーミングサービスはこれらのネーミングおよびディレクトリサービス，ならびにそのほかの将来の SPI プロバイダへのポータブルアクセスをできるようにします。ただし，JNDI アダプタは Netscape LDAP サーバ 4.0 専用に認証されています。

14.10.2　構成と使用

バッキングストアアダプタを接続できます。つまり，使用するアダプタの型を，ネーミングサービスの起動時に使用する構成（プロパティ）ファイルにユーザ定義情報として格納して指定できます。インメモリアダプタを除くすべてのアダプタはパーシステントです。ネームスペース全体をメモリに保管するライトウェイトのネーミングサービスを使用したい場合には，インメモリアダプタを使用してください。

注

最新バージョンのネーミングサービスでは，ネーミングサービスの実行中は設定を変更できません。設定を変更するには，必ずサービスを停止し，構成ファイルを変更してからネーミングサービスを再起動してください。

(1)　プロパティファイル

そのほかの多くのネーミングサービスと同様，どのアダプタを使用するかや，その個々の構成などはネーミングサービスのプロパティファイルで処理されます。デフォルトのプロパティを表 14-6 に示します。

表 14-6　すべてのアダプタに共通のデフォルトプロパティ

プロパティ	デフォルト	説明
vbroker.naming.backingStoreType	InMemory	使用するネーミングサービスアダプタのタイプを指定します。 このプロパティは，ネーミングサービスを使用したいプラガブルバッキングストア（外部記憶装置）のタイプを指定します。有効なオプションは，InMemory，JDBC，Dx，および JNDI です。デフォルトは InMemory です。
vbroker.naming.cacheOn	0	ネーミングサービスキャッシュを使用するかどうかを指定します。
vbroker.naming.cacheSize	5	ネーミングサービスキャッシュがオンの場合にそのサイズを指定します。

(2)　JDBC アダプタプロパティ

vbroker.naming.backingStoreType

このプロパティは JDBC に設定しなければなりません。JDBC アダプタでは，poolSize，jdbcDriver，url，loginName，および loginPwd プロパティも設定する必要があります。

vbroker.naming.jdbcDriver

このプロパティは，バッキングストアとして使用するデータベースへのアクセスに必要な JDBC ドライバです。ネーミングサービスはこの設定に従って表 14-7 の中から適切な JDBC ドライバをロードします。デフォルトは Java DataStore JDBC ドライバです。

表 14-7　vbroker.naming.jdbcDriver プロパティを使用してデータベースへアクセスするための JDBC ドライバ

JDBC ドライバ値	説明
com.borland.datastore.jdbc.DataStoreDriver	JDataStore ドライバ
com.sybase.jdbc.SybDriver	Sybase ドライバ
oracle.jdbc.driver.OracleDriver	Oracle ドライバ
interbase.interclient.Driver	Interbase ドライバ
weblogic.jdbc.mssqlserver4.Driver	WebLogic MS SQLServer ドライバ
COM.ibm.db2.jdbc.app.DB2Driver	IBM DB2 ドライバ

vbroker.naming.loginName

このプロパティはデータベースに対応するログイン名です。デフォルトは VisiNaming です。

vbroker.naming.loginPwd

このプロパティはデータベースに対応するログインパスワードです。デフォルトは VisiNaming です。

vbroker.naming.poolSize

このプロパティは，バッキングストアとして JDBC アダプタを使用する場合にコネクションプールで使用するデータベースコネクションの数を指定します。デフォルト値は 5 ですが，データベースが扱える任意の値に増やせます。ネーミングサービスへのリクエスト数が多くなりそうな場合は，この値を大きくしてください。

vbroker.naming.url

このプロパティは，アクセスしたいデータベースの位置を指定します。この設定は使用するデータベースに依存します。デフォルトは JDataStore で，データベース位置は rootDB.jds というカレントディレクトリです。rootDB.jds に限らないで，任意の名前を付けてかまいません。これに従って構成ファイルを更新しなければなりません。データベースの URL 値を表 14-8 に示します。

表 14-8　vbroker.naming.url プロパティを使用してアクセスするデータベースの URL

URL 値	説明
jdbc:borland:dslocal:<db_name>	JDataStore URL
jdbc:sybase:Tds:<host>:<port>/<db_name>	Sybase URL
jdbc:oracle:thin:@<host>:<port>:<sid>	Oracle URL
jdbc:interbase://<server>/<full_db_path>	Interbase ※1 URL
jdbc:weblogic:mssqlserver4:<db_name>@<host>:<port>	WebLogic MS SQLServer URL
jdbc:db2:<db_name>	IBM DB2※2 URL
<full_path_JDataStore_db>	ネイティブドライバの DataExpress※3 URL

注※1

JDBC を介して InterBase にアクセスする前に，InterServer サーバを起動してください。InterBase サーバがローカルホストに存在するなら，localhost を<server>に指定します。これ以外の場合はホスト名を<server>に指定します。InterBase データベースが Windows に存在するなら，<full_db_path>に driver:¥¥dir1¥¥dir2¥¥db.gdb（最初の¥記号は 2 番目の¥記号をエスケープします）を指定します。InterBase データベースが UNIX に存在するなら，<full_db_path>に/dir1/dir2/db.gdb を指定します。

注※2

JDBC を介して DB2 にアクセスする前に，Client Configuration Assistant を使用してデータベースをエイリアス <db_name>で登録しなければなりません。データベースの登録後は，vbroker.naming.url プロパティに<host>や<port>を指定する必要はありません。

注※3

JDataStore データベースが Windows に存在するなら，<full path of the JDataStore database>に Driver:¥¥dir1¥¥dir2¥¥db.jds（最初の¥記号は 2 番目の¥記号をエスケープします）を指定します。JDataStore データベースが UNIX に存在するなら，<full path of the JDataStore database>に/dir1/dir2/db.jds を指定します。

(3)　DataExpress アダプタプロパティ

DataExpress アダプタプロパティについて説明します。

vbroker.naming.backingStoreType

このプロパティは Dx に設定する必要があります。

vbroker.naming.loginName

このプロパティはデータベースに対応するログイン名です。デフォルトは VisiNaming です。

vbroker.naming.loginPwd

このプロパティはデータベースに対応するログインパスワードです。デフォルト値は VisiNaming です。

vbroker.naming.url

このプロパティは，データベースの位置を指定します。

(4)　JNDI アダプタプロパティ

表 14-9 に，JNDI アダプタの構成ファイルに表示される設定例を示します。

表 14-9　JNDI アダプタ構成ファイルの例

設定	説明
vbroker.naming.backingStoreType=JNDI	プラガブルバッキングストア（外部記憶装置）のタイプを JNDI アダプタ用の JNDI に指定します。
vbroker.naming.loginName=<user name>	JNDI バッキングサーバ上のユーザログイン名です。
vbroker.naming.loginPwd=<password>	JNDI バッキングサーバのユーザパスワードです。
vbroker.naming.jndiInitialFactory=com.sun.jndi.ldap.LdapCtxFactory	JNDI 初期ファクトリを指定します。

設定	説明
vbroker.naming.jndiProviderURL=ldap://<hostname>:<ldap portNo>/<initial root context>	JNDI プロバイダ URL を指定します。
vbroker.naming.jndiAuthentication=simple	JNDI バッキングサーバがサポートする JNDI 認証タイプを指定します。

注

ユーザは，ディレクトリサーバにスキーマや属性を追加するために必要な権限を持つ必要があります。

(5)　キャッシング機能

キャッシング機能をオンにすることで，バッキングストアの性能を向上できます。例えば，JDBC アダプタの場合，解決またはバインドオペレーションがあるたびにデータベースに直接アクセスすると，かなりの時間が掛かりますが，結果をキャッシングすることでデータベースアクセスの回数を減らせます。キャッシング機能をオンにする前に知っておくべき留意点が幾つかあります。まず，基本となるデータにアクセスするネーミングサービスは，キャッシュを使用したネーミングサービスだけであることを確認してください。そうでなければ，古くなったデータがキャッシュに入っていることがあるため，ネーミングサービスを使用するクライアントが間違ったデータを取得することがあります。バッキングストアの性能向上が見られるのは，同じ一つのデータが複数回アクセスされた場合だけです。

注

使用環境での性能がキャッシング機能によって必ず向上するという確信がないかぎり，キャッシング機能はオンにしないでください。

このキャッシング機能は，コンテキストごとにキャッシュする実装になっています。この場合，各コンテキストにキャッシュがインストールされることになり，コンテキストとオブジェクトの両方をキャッシュするために使用されます。このキャッシュのサイズは調整できます。デフォルトでは，このキャッシュのサイズは 5 です。

キャッシング機能を使用するには，次のプロパティを構成ファイルに追加してください。

```
vbroker.naming.cacheOn=1
vbroker.naming.CacheSize=5
```

14.11　クラスタ

Borland Enterprise Server VisiBroker は，多数のオブジェクトバインディングを一つの名前に対応させることを可能とするクラスタ化機能をサポートします。さらに，ネーミングサービスはクラスタ内の別々のバインディング間で負荷分散を行えます。クラスタの生成時には負荷分散の方法を決定できます。それ以降にクラスタに対して名前とオブジェクトのバインディングを解決するクライアントの負荷が，異なるクラスタサーバメンバ間で分散されることになります。

クラスタは，Name をオブジェクトリファレンスのグループに対応づけるマルチバインド機能です。クラスタの生成は ClusterManager オブジェクトによって行われます。生成時に，ClusterManager の create_cluster メソッドは，使用する方法を指定する String パラメタを取り込みます。このメソッドはクラスタへのリファレンスを返し，これを使用してメンバの追加，削除，および検索ができます。クラスタの構成を決定したら，そのクラスタのリファレンスを特定の名前でネーミングサービス内の任意のコンテキストにバインドできます。こうすることで，Name に対する以降の resolve オペレーションはクラスタ内にバインドされたオブジェクトリファレンスを返します。

create_cluster メソッドの引数に""を指定した場合は，"RoundRobin"指定時と同じ動作をします。

create_cluster メソッドの引数に，"RoundRobin"，"SmartRoundRobin"，""以外の文字列を指定した場合，nameserv で NullPointerException が発生し，クライアントに不正なオブジェクトリファレンスが返ります。そのため，そのオブジェクトリファレンスを用いたリクエストで，UNKNOWN 例外が発生します。

14.11.1　クラスタ化方法

ネーミングサービスは，デフォルトではクラスタによる RoundRobin の方法を使用します。クラスタの生成後は，クラスタの方法を変更できません。ユーザが定義した方法はサポートしていませんが，今後 RoundRobin 以外の方法もできるように予定しています。デフォルトの RoundRobin の方法のほかに現在使用できる方法は，SmartRoundRobin の方法だけです。SmartRoundRobin と RoundRobin の違いは，SmartRoundRobin は，CORBA オブジェクトリファレンスがアクティブであること，つまりレディ状態である CORBA サーバをオブジェクトリファレンスが参照中であることを確認するために幾つかの検証をします。

注

アクティブであることが検証されたオブジェクトは，カレントインプリメンテーションが活性化するので，SmartRoundRobin の使用はお勧めできません。また，クラスタフェールオーバー機能は RoundRobin の方法だけ使用できます。

14.11.2　クラスタインタフェースと ClusterManager インタフェース

クラスタとネーミングコンテキストが酷似していても，コンテキストにはクラスタと無関係のメソッドがあります。例えば，ネーミングコンテキストをクラスタにバインドしても無意味です。クラスタにはネーミングコンテキストではなくオブジェクトリファレンスの集合が入るためです。ただし，クラスタインタフェースは，bind，rebind，resolve，unbind，list など NamingContext と同じメソッドを多く持っています。これらのオペレーションは，主にグループに対するオペレーションに関係します。クラスタ固有オペレーションは pick だけです。両者の重要な違いのもう一つは，クラスタは複合名をサポートしないということです。クラスタは階層ディレクトリ構造を持たないで，フラット構造でオブジェクトリファレンスを格納するので，コンポーネントを一つだけ持つ名前だけを使用できます。

(1)　クラスタインタフェースの IDL 指定

```
module CosNamingExt {
   typedef sequence<Cluster> ClusterList;
   enum ClusterNotFoundReason {
      missing_node,
      not_context,
      not_cluster_context
   };
   exception ClusterNotFound {
      ClusterNotFoundReason why;
      CosNaming::Name rest_of_name;
   };
   exception Empty {};
   interface Cluster {
      Object select() raises(Empty);
      void bind(in CosNaming::NameComponent n, in Object obj)
           raises(CosNaming::NamingContext::CannotProceed,
                  CosNaming::NamingContext::InvalidName,
                  CosNaming::NamingContext::AlreadyBound);
      void rebind(in CosNaming::NameComponent n, in Object obj)
           raises(CosNaming::NamingContext::CannotProceed,
                  CosNaming::NamingContext::InvalidName);
      Object resolve(in CosNaming::NameComponent n)
           raises(CosNaming::NamingContext::NotFound,
                  CosNaming::NamingContext::CannotProceed,
                  CosNaming::NamingContext::InvalidName);
      void unbind(in CosNaming::NameComponent n)
           raises(CosNaming::NamingContext::NotFound,
                  CosNaming::NamingContext::CannotProceed,
                  CosNaming::NamingContext::InvalidName);
      void destroy()
           raises(CosNaming::NamingContext::NotEmpty);
      void list(in unsigned long how_many,
            out CosNaming::BindingList bl,
            out CosNaming::BindingIterator bi);
   } ;
};
```

(2)　ClusterManager インタフェースの IDL 指定

```
CosNamingExt module {
   interface ClusterManager {
      Cluster create_cluster(in string algo);
      Cluster find_cluster(
       in CosNaming::NamingContext ctx, in CosNaming::Name n)
         raises(ClusterNotFound,
             CosNaming::NamingContext::CannotProceed,
             CosNaming::NamingContext::InvalidName);
      Cluster find_cluster_str(
                 in CosNaming::NamingContext ctx, in string n)
         raises(ClusterNotFound,
             CosNaming::NamingContext::CannotProceed,
             CosNaming::NamingContext::InvalidName);
      ClusterList clusters();
   };
};
```

14.11.3　クラスタの生成

クラスタを生成するには，Cluster Manager インタフェースを使用します。ネーミングサーバの起動時に，Cluster Manager オブジェクトが一つだけ自動的に生成されます。Cluster Manager はネーミングサーバごとに一つだけあります。Cluster Manager の役割は，ネーミングサーバ内のクラスタの生成，検索，および状態の把握です。

1. クラスタオブジェクトの生成に使用したいネーミングサーバにバインドします。
2. ファクトリリファレンスの get_cluster_manager メソッド呼び出しによって，Cluster Manager へのリファレンスを取得します。

3.指定されたクラスタの方法でクラスタを生成します。

4.クラスタを使用して，オブジェクトを Name にバインドします。

5.Cluster オブジェクト自身を Name にバインドします。

6.指定されたクラスタの方法の Cluster リファレンスによって解決します。

コードサンプル 14-7　Cluster オブジェクトの生成と使用（C++）

```
. . .
ExtendedNamingContextFactory_var myFactory =
      ExtendedNamingContextFactory::_bind(orb, "NamingService");
ClusterManager_var clusterMgr =
      myFactory->get_cluster_manager();
Cluster_var clusterObj =
      clusterMgr->create_cluster("RoundRobin");
clusterObj->bind(new NameComponent(
      "member1", "aCluster"), obj1);
clusterObj->bind(new NameComponent(
      "member2", "aCluster"), obj2);
clusterObj->bind(new NameComponent(
      "member3", "aCluster"), obj3);
NameComponent_var myClusterName =
      new NameComponent("ClusterName", "");
root->bind(myClusterName, clusterObj);
root->resolve(myClusterName);
      // a member of the Cluster is returned
root->resolve(myClusterName);
      // the next member of the Cluster is returned
root->resolve(myClusterName);
      // the last member of the Cluster is returned
. . .
```

コードサンプル 14-8　Cluster オブジェクトの生成と使用（Java）

```
. . .
ExtendedNamingContextFactory myFactory =
    ExtendedNamingContextFactoryHelper.bind(orb,
                                        "NamingService");
ClusterManager clusterMgr = myFactory.get_cluster_manager();
Cluster clusterObj = clusterMgr.create_cluster("RoundRobin");
clusterObj.bind(new NameComponent("member1", "aCluster"),
                obj1);
clusterObj.bind(new NameComponent("member2", "aCluster"),
                obj2);
clusterObj.bind(new NameComponent("member3", "aCluster"),
                obj3);
NameComponent myClusterName =
                        new NameComponent("ClusterName", "");
root.bind(myClusterName, clusterObj);
root.resolve(myClusterName) //
                        a member of the Cluster is returned.
root.resolve(myClusterName) //
                 the next member of the Cluster is returned.
root.resolve(myClusterName) //
                 the last member of the Cluster is returned.
. . .
```

(1)　明示的なクラスタと暗黙的なクラスタ

クラスタ化機能はネーミングサービスに対して自動的にオンになります。注意点は，この機能がオンになると，オブジェクトをバインドするためにクラスタが透過的に生成されるということです。使用の方法はラウンドロビンに固定されます。この影響は，ネーミングサーバ内の同じ名前に幾つかのオブジェクトをバインドできるということです。反対に，この名前を解決すると，これらのオブジェクトの一つが返され，unbind オペレーションはその名前に対応するクラスタをデストラクトします。これは，ネーミングサービスが CORBA の仕様に準拠しなくなったという意味です。インターオペラブルネーミングサービスの仕様では，複数のオブジェクトを同じ名前でバインドする機能を明示的に禁止します。準拠しているネーミングサービスでは，クライアントが同じ名前を使用して異なるオブジェクトにバインドしようとすると，

AlreadyBound 例外が発生します。ユーザは最初から，個々のサーバにこの機能を使用するかどうかを決定する必要があり，その決定を守るべきです。

注

暗黙的なクラスタモードから明示的なクラスタモードへの切り替えはできません。バッキングストアを破壊するおそれがあるためです。

ネーミングサーバを暗黙的なクラスタ化機能と一緒に使用した場合，クラスタ化機能をオンにした状態でネーミングサーバを引き続き活性化してください。機能をオンにするには，構成ファイルに次のプロパティ値を定義してください。

```
vbroker.naming.propBindOn=1
```

注

明示的なクラスタ化および暗黙的なクラスタ化の両方のサンプルについては，次のインストールディレクトリ下の次のディレクトリのコードを参照してください。

```
examples/vbe/ins/implicit_clustering
examples/vbe/ins/explicit_clustering
```

14.11.4　負荷分散

Cluster Manager とスマートエージェントはどちらもラウンドロビン負荷分散機能を提供しますが，これらは異なる性質を持ちます。スマートエージェントの負荷分散は暗黙のうちに行われます。サーバの起動時，サーバは自動的に自分自身をスマートエージェントに登録し，これによって今度は Borland Enterprise Server VisiBroker が，容易だが独占的な方法でクライアントがサーバへのリファレンスを取得できるようにします。ただし，これらの自動化には相応の代償があります。グループを構成するものおよびグループのメンバの決定に際しては，プログラマが選択することはできません。スマートエージェントがすべてを決定します。ここで，代替手段を提供するクラスタが便利です。これはクラスタのプロパティをプログラムによって定義，生成する方法を提供します。プログラマはクラスタに適用する方法を定義でき，クラスタのメンバも自由に選択できます。方法は生成時に固定されますが，クライアントはクラスタの存続期間中にクラスタのメンバを追加したり削除したりできます。

14.12　フェールオーバー

ネーミングサービスはマスタ／スレーブモデルを使用したフェールオーバー機能をインプリメントします。アクティブモードのマスタとスタンドバイモードのスレーブという二つのネーミングサーバが同時実行中でなければなりません。マスタとスレーブの両ネーミングサーバは，同じ基本データをパーシステントなバッキングストアにサポートする必要があります。各サーバに強制的にバッキングストアと直接やり取りさせるために両方のサーバのキャッシング機能は必ずオフにして，データが確実に不変であるようにしてください。

両方のネーミングサーバがアクティブなら，ネーミングサービスを使用しているクライアントは常にマスタを優先します。マスタが不測の事態で終了した場合，スレーブネーミングサーバが引き継ぎます。このマスタからスレーブへの切り替えはシームレスであり，クライアントから見て透過的です。ただし，スレーブネーミングサーバはマスタサーバにはなりません。その代わり，マスタサーバが使用不能になった場合には一時的にバックアップします。この間に，ユーザはクラッシュしたマスタサーバの回復のための対策を行います。マスタが再び起動されたあとは，新しいクライアントからのリクエストだけがマスタサーバに送信されます。すでにスレーブネーミングサーバにバインドされているクライアントは，自動的にマスタにスイッチバックしません。

フェールオーバーが発生すると，クライアントから見て透過的ですが，わずかに遅延があります。これは，スレーブネーミングサーバのサーバオブジェクトを，入ってきたリクエストによってオンデマンドによる活性化をしなければならない場合があるためです。また，イテレータのような一時的なオブジェクトリファレンスは無効です。トランジェントなイテレータによるリファレンスを使用したクライアントは，これらのリファレンスの無効化に備えなければならないので，これは正常なことです。一般に，ネーミングサーバはあまり多くのイタレータオブジェクトを集中保管しないで，いつでもクライアントのイタレータによるリファレンスを無効にできます。これらはトランジェントリファレンスではなく，パーシステントリファレンスを使用したクライアントリクエストであればスレーブネーミングサーバに再転送されます。

注

すでにスレーブネーミングサーバにバインドされているクライアントは，自動的にマスタにスイッチバックしないで，一つのレベルのフェールオーバーしかサポートしません。したがって，スレーブネーミングサーバが使用不能になると，ネーミングサービスも使用不能になります。

14.12.1　フォルトトレランス用のネーミングサービスの設定

二つのネーミングサーバが実行中でなければなりません。その一つにはマスタ，もう一つにはスレーブと名づけてください。両方のサーバで同じプロパティファイルを使用できます。プロパティファイルでの適切な値については，コードサンプル 14-9 を参照してください。

コードサンプル 14-9　フォルトトレランスを使用するための設定

```
vbroker.naming.enableSlave=1
vbroker.naming.masterServer=<Master Naming Server Name>
vbroker.naming.masterHost=<host ip address for Master>
vbroker.naming.masterPort=<port number that Master is listening on>
vbroker.naming.slaveServer=<Slave Naming Server Name>
vbroker.naming.slaveHost=<host ip address for Slave>
vbroker.naming.slavePort=<port number that Slave is listening on>
```

特定ポートで強制的にネーミングサーバを起動するには，次に示すコマンドラインオプションでネーミングサーバを起動してください。

```
prompt> nameserv -J¥
        -Dvbroker.se.iiop_tp.scm.iiop_tp.listener.port=¥
        <port number> com.inprise.vbroker.naming.ExtFactory ¥
        <Naming_Server_Name>
```

注

　マスタサーバとスレーブサーバの起動順序に制限はありません。

14.13　プログラムのコンパイルとリンク（C++）

ネーミングサービスを使用する C++アプリケーションには，次に示す生成ファイルが必要です。

```
#include "CosNaming_c.hh"
#include "CosNamingExt_c.hh"
```

(1)　UNIX

UNIX アプリケーションは，次のライブラリのどれかとリンクする必要があります。

AIX

- libcosnm_r.a（マルチスレッド）

HP-UX

- libcosnm_r.sl（マルチスレッド）

Solaris

- libcosnm_r.so（マルチスレッド）

(2)　Windows

Windows アプリケーションは，次のライブラリとリンクする必要があります。

- cosnm_r.lib（マルチスレッド）

14.14　Java のインポート文

Java の場合，次に示すインポート文は，ネーミングサービスの Borland Enterprise Server VisiBroker 拡張機能を使用したい Java クラスに使用してください。

```
import com.inprise.vbroker.CosNamingExt.*;
. . .
```

OMG に準拠したネーミングサービスの機能へアクセスしたい場合は，次に示すパッケージが必要です。

```
import org.omg.CosNaming.*
import org.omg.CosNaming.NamingContextPackage.*
import org.omg.CosNaming.NamingContextExtPackage.*
```

14.15　サンプルプログラム

Borland Enterprise Server VisiBroker には，ネーミングサービスの使用方法を説明する幾つかのサンプルプログラムが提供されています。これらのサンプルは examples/vbe/ins ディレクトリに入っており，現在ネーミングサービスで使用できる新機能をすべて説明しています。さらに，examples/vbe/basic/bank_naming ディレクトリに入っている Bank ネーミングのサンプルは，ネーミングサービスの基本的な使い方を説明しています。

サンプルプログラムを実行する前に，まず「14.3　ネーミングサービスの実行」で説明したようにネーミングサービスを起動しなければなりません。さらに，最低一つのネーミングコンテキストが次に示すどれかの手段で生成されていることを必ず確認してください。

- 「14.3　ネーミングサービスの実行」で説明したようにネーミングサービスを起動します。これでイニシャルコンテキストが自動生成されます。
- VisiBroker コンソールを使用します。
- クライアントを NamingContextFactory にバインドさせ，create_context メソッドを使用します。
- クライアントに ExtendedNamingContextFactory を使用させます。

注

ネーミングコンテキストを生成していない場合，クライアントが bind を発行しようとしたときに CORBA.NO_IMPLEMENT 例外が発生します。

14.15.1　名前のバインド

Bank ネーミングのサンプルでは AccountManager インタフェースを使用して Account をオープンしたりアカウントの残高を問い合わせたりします。次に示す Server クラスは，名前をオブジェクトリファレンスにバインドするためのネーミングサービスの使い方を説明します。サーバは，ネーミングサーバのルートコンテキストに IOR をバインド登録し，これは次にクライアントが検索します。

このサンプルでは，次の方法を理解できるようになります。

1. ネーミングサービスのルートコンテキストへのリファレンスを取得するための，VisiBroker ORB インスタンスの resolve_initial_references メソッドの使用方法（サンプルでは，NameService のデフォルト名でネーミングサービスを起動しなければなりません）
2. NamingContextExtHelper クラスの narrow メソッド使用による，ルートコンテキストのリファレンスへのキャスト方法
3. AccountManagerImpl オブジェクトの POA およびサーバントの生成方法
4. 最後に NamingContext インタフェースの bind メソッドを使用して，「BankManager」という名前を AccountManagerImpl オブジェクトのオブジェクトリファレンスにバインドする方法

コードサンプル 14-10　Server.c（C++）

```
#include "CosNaming_c.hh"
#include "BankImpl.h"

// USE_STD_NS is a define setup by VisiBroker to use the std namespace
USE_STD_NS

int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
```

```
      // get a reference to the root POA
      PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(
                  orb->resolve_initial_references("RootPOA"));

      // get a reference to the Naming Service root_context
      CosNaming::NamingContext_var rootContext =
                  CosNaming::NamingContext::_narrow(
                  orb->resolve_initial_references("NameService"));

    CORBA::PolicyList policies;
    policies.length(1);

    policies[(CORBA::ULong)0] =
         rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);

      // get the POA Manager
      PortableServer::POAManager_var poa_manager =
                  rootPOA->the_POAManager();

      // Create myPOA with the right policies
      PortableServer::POA_var myPOA =
         rootPOA->create_POA("bank_agent_poa", poa_manager, policies);

      // Create the servant
      AccountManagerImpl managerServant;

      // Decide on the ID for the servant
      PortableServer::ObjectId_var managerId =
      PortableServer::string_to_ObjectId("BankManager");

      // Activate the servant with the ID on myPOA
      myPOA->activate_object_with_id(managerId, &managerServant);

      // Activate the POA Manager
      poa_manager->activate();

      CORBA::Object_var reference =
      myPOA->servant_to_reference(&managerServant);

      // Associate the bank manager with the name at the root context
      CosNaming::Name name;
      name.length(1);
      name[0].id = (const char *) "BankManager";
      name[0].kind = (const char *) "";
      rootContext->rebind(name, reference);

      cout << reference << " is ready" << endl;

      // Wait for incoming requests
      orb->run();
    }
      catch(const CORBA::Exception& e) {
      cerr << e << endl;
      return 1;
    }

    return 0;
  }
```

コードサンプル 14-11　Server.java（Java）

```
import org.omg.PortableServer.*;
import org.omg.CosNaming.*;

public class Server {
   public static void main(String[ ] args){
      try {
         // Initialize the ORB.
         org.omg.CORBA.ORB orb =
                        org.omg.CORBA.ORB.init(args,null);
         // get a reference to the rootPOA
         POA rootPOA = POAHelper.narrow(
                orb.resolve_initial_references("RootPOA"));
         // get a reference to the Naming Service root
         // context
```

```
            org.omg.CORBA.Object rootObj =
                    orb.resolve_initial_references("NameService");
            NamingContextExt root =
                        NamingContextExtHelper.narrow(rootObj);

            // Create policies for our persistent POA
            org.omg.CORBA.Policy[ ] policies = {
                rootPOA.create_lifespan_policy(
                                LifespanPolicyValue.PERSISTENT)
            };
            // Create myPOA with the right policies
            POA myPOA = rootPOA.create_POA(
                     "bank_agent_poa", rootPOA.the_POAManager(),
                     policies );
            // Create the servant
            AccountManagerImpl managerServant =
                                    new AccountManagerImpl();
            // Decide on the ID for the servant
            byte[ ] managerId = "BankManager".getBytes();
            // Activate the servant with the ID on myPOA
            myPOA.activate_object_with_id(managerId,
                                          managerServant);

            // Activate the POA manager
            rootPOA.the_POAManager().activate();

            // Associate the bank manager with the name at
            // the root context Note that casting is needed
            // as a workaround for a JDK 1.1.x bug.
            ((NamingContext)root).bind(
                  root.to_name("BankManager"),
                  myPOA.servant_to_reference(managerServant));

            System.out.println(
                     myPOA.servant_to_reference(managerServant)
                     + " is ready.");
            // Wait for incoming requests
            orb.run();
          }catch (Exception e){
            e.printStackTrace();
          }
        }
    }
```

15 オブジェクト活性化デーモンの使用

この章では，OAD（オブジェクト活性化デーモン）の使用方法について説明します。

15.1　オブジェクトとサーバの自動活性化

OAD（オブジェクト活性化デーモン）は，インプリメンテーションリポジトリを実装した Borland Enterprise Server VisiBroker のデーモンです。インプリメンテーションリポジトリは，サーバがサポートしているクラス，実体化されているオブジェクトとそれらに関する情報をランタイム時にリポジトリとして提供します。また，OAD はクライアントがオブジェクトを参照するときにインプリメンテーションを自動的に活性化する目的にも使用されます。OAD にオブジェクトインプリメンテーションを登録することで，利用したいオブジェクトは自動的に活性化されます。

オブジェクトインプリメンテーションは，コマンドラインインタフェース（oadutil）を使用して登録できます。「15.5　OAD との IDL インタフェース」で説明する，OAD との VisiBroker ORB インタフェースもあります。どちらの場合も，リポジトリ ID，オブジェクト名，活性化ポリシー，およびインプリメンテーションを表す実行可能プログラムを指定しなければなりません。

注

任意のバージョンの Borland Enterprise Server VisiBroker（Java）および VisiBroker for C++3.0 で生成されたサーバを実体化するには，Borland Enterprise Server VisiBroker OAD を使用できます。

OAD は，オブジェクトサーバがオンデマンドで活性化されるホスト上で起動しておけばよい，独立したプロセスです。

15.1.1　インプリメンテーションリポジトリデータの探索

OAD に登録されたすべてのオブジェクトインプリメンテーションについての活性化情報は，インプリメンテーションリポジトリに格納されます。デフォルトでは，インプリメンテーションリポジトリデータは impl_rep という名前のファイルに格納されます。このファイルのパス名は VBROKER_ADM 環境変数の値によって決まります。Borland Enterprise Server VisiBroker が/usr/local/vbroker ディレクトリにインストールされている場合，このファイルへのパスは/usr/local/vbroker/adm/impl_dir/impl_rep になります。これらのデフォルト値は「2.　環境設定」で説明した OAD 環境変数を使用して変更できます。

15.1.2　サーバの起動

OAD はクライアントリクエストに応じてサーバを起動します。次に示すタイプのクライアントが，OAD によってサーバを起動できます。

- VisiBroker for C++ 3.x クライアント
- VisiBroker 4.x クライアント（C++）
- VisiBroker for Java 3.x クライアント
- VisiBroker 4.x クライアント（Java）
- Borland Enterprise Server VisiBroker のクライアント
- Borland Enterprise Server VisiBroker 以外の IIOP プロトコルを使用するクライアント。IIOP に準拠するクライアントなら，サーバのリファレンス使用時に Borland Enterprise Server VisiBroker サーバを起動できます。サーバのエクスポートされたオブジェクトリファレンスは OAD をポイントし，生成されたサーバには IIOP 規則に従ってクライアントを転送できます。（ネームサービスなどを使用して）サーバのオブジェクトリファレンスの厳密な永続化を保証するために，OAD は常に同じポートで起動しなければなりません。次の例では，OAD はポート 16050 で起動されます。

```
prompt> oad -VBJprop vbroker.se.iiop_tp.scm.iiop_tp.¥
        listener.port=16050
```

注

ポート 16000 はデフォルトポートですが，これは listener.port プロパティを設定することによって変更できます。

15.2　OAD の起動

OAD は，クライアントのアクセス時に自動的に起動されるオブジェクトを登録できるようにするオプション機能です。まずスマートエージェントを起動したあとに，OAD を起動してください。

Windows の場合

コマンドプロンプトで次のコマンドを入力して OAD を起動してください。

```
prompt> oad
```

UNIX の場合

次のコマンドを入力して OAD を起動してください。

```
prompt> oad &
```

oad コマンドには，表 15-1 のコマンドライン引数を指定できます。

表 15-1　oad コマンドのオプション

オプション	説明
-verbose	バーボースモードをオンにして，メッセージが stdout に出力されるようにします。
-version	このツールのバージョンを出力します。
-path <path>	インプリメンテーションリポジトリ格納用のプラットフォーム固有ディレクトリを指定します。これは，環境変数の使用による設定をすべて変更します。
-filename <repository filename>	インプリメンテーションリポジトリの名前を指定します。このオプションを指定しないと，デフォルトは impl_rep です。これはユーザの環境変数設定をすべて変更します。
-timeout <# of seconds>	生成されたサーバプロセスが，リクエストされた VisiBroker ORB オブジェクトを活性化するのを，OAD が待つ時間を指定します。デフォルトのタイムアウトは 20 秒です。待ち時間を不定にしたい場合は，この値を 0 に設定してください。 生成されたサーバプロセスが，リクエストされた ORB オブジェクトをタイムアウト時間内に活性化しなければ，OAD は生成されたプロセスに対して Kill を実行し，クライアントには CORBA::NO_RESPONSE 例外が返されることになります。より詳細な情報を返したい場合は，バーボースオプションをオンにしてください。
-IOR <IOR filename>	OAD の文字列化された IOR を格納するファイル名を指定します。
-kill	オブジェクトが OAD から登録解除された場合に，その子プロセスに対して Kill を実行するよう規定します。
-no_verify	登録の有効性チェックをオフにします。
-?	コマンドの使い方を表示します。
-readonly	OAD を-readonly オプションで起動した場合，登録されたオブジェクトは何も変更されません。オブジェクトの登録や登録解除をしようとするとエラーを返します。-readonly オプションは，一般にインプリメンテーションリポジトリを変更し，追加変更を行わないように readonly モードで OAD をリスタートしたあとに使用します。

15.3　オブジェクト活性化デーモンユーティリティの使用

oadutil コマンドは，Borland Enterprise Server VisiBroker のシステムで使用できるオブジェクトインプリメンテーションの登録，登録解除，およびリスト出力を手動で行う手段を提供します。oadutil コマンドは Java でインプリメントされ，コマンドラインインタフェースを使用します。各コマンドは，oadutil コマンドを起動し，実行するオペレーションのタイプを第 1 引数として渡すことによってアクセスされます。

注

oadutil コマンドを使用するには，ネットワークの少なくとも一つのホストで OAD（オブジェクト活性化デーモンプロセス）を起動しなければなりません。

oadutil コマンドの構文は次のとおりです。

構文

```
oadutil {list|reg|unreg} [options]
```

このツールのオプションは，list，reg，unreg のどれを指定したかによって異なります。

インタフェース名からリポジトリ ID への変換

インタフェース名とリポジトリ ID は，活性化されたオブジェクトがインプリメントするべきインタフェースの型を表す方法です。IDL で定義されたすべてのインタフェースには一意のリポジトリ ID が割り当てられます。この文字列は，インタフェースリポジトリ（IR）や OAD，および VisiBroker ORB 自身への呼び出しで通信するときに型を識別するために使用します。

オブジェクトを OAD に登録または登録解除する場合，oadutil コマンドを使用すれば，オブジェクトの IDL インタフェース名か，そのリポジトリ ID を指定できます。

インタフェース名は，次のようにしてリポジトリ ID に変換できます。

1. インタフェース名の前に「IDL:」を付けます。
2. スコープ解決演算子の，先頭以外のすべてのインスタンス「::」をスラント「/」文字に置換します。
3. インタフェース名の後ろに「:1.0」を付けます。

   例えば，「::Module1::Module2::IntfName」という IDL インタフェース名を変換すると，次のようなリポジトリ ID になります。

   ```
   IDL:Module1/Module2/IntfName:1.0
   ```

#pragma id と#pragma プリフィクス機能を使用して，インタフェース名からのデフォルト生成のリポジトリ ID を変更できます。ユーザ定義の IDL ファイルで標準外のリポジトリ ID を指定するために #pragma id 機能を使用した場合，上記で説明した変換プロセスは効きません。この場合，-r リポジトリ ID 引数を使用してオブジェクトのリポジトリ ID を指定してください。

C++の場合

オブジェクトインプリメンテーションのインタフェースのリポジトリ ID を取得するには，すべての CORBA オブジェクトに対して定義された<interface_name>._repository_id()メソッドを使用してください。

Java の場合

オブジェクトインプリメンテーションのインタフェースのリポジトリ ID を取得するには，すべての CORBA オブジェクトに対して定義された java: <interface_name>Helper.id()メソッドを使用してください。

15.3.1　oadutil list によるオブジェクトのリスト出力

oadutil list コマンドは，OAD に登録されたすべての VisiBroker ORB オブジェクトインプリメンテーションを返します。各 OAD は，自分のインプリメンテーションリポジトリデータベースに登録情報を格納しています。

注

oadutil list コマンドを使用するには，ネットワークの少なくとも一つのホストで OAD（オブジェクト活性化デーモンプロセス）を起動しなければなりません。

oadutil list コマンドの構文は次のとおりです。

構文

```
oadutil list [options]
```

oadutil list コマンドには表 15-2 のコマンドライン引数を指定できます。

表 15-2　oadutil list コマンドのオプション

オプション	説明
-i <interface_name>	特定の IDL インタフェース名のオブジェクトについてのインプリメンテーション情報をリスト出力します。一度に指定できるオプションは，-i，-r，-s，-poa のうち一つだけです。 注 VisiBroker ORB とのすべての通信は，インタフェース名ではなくオブジェクトのリポジトリ ID を参照します。インタフェース名の指定時に行われる変換の詳細については，「15.3　オブジェクト活性化デーモンユーティリティの使用」の「インタフェース名からリポジトリ ID への変換」を参照してください。
-r <repository_id>	特定のリポジトリ ID のインプリメンテーション情報をリスト出力します。リポジトリ ID 指定の詳細については，「15.3　オブジェクト活性化デーモンユーティリティの使用」の「インタフェース名からリポジトリ ID への変換」を参照してください。一度に指定できるオプションは，-i，-r，-s，-poa のうち一つだけです。
-s <service name>	特定のサービス名のインプリメンテーション情報をリスト出力します。一度に指定できるオプションは，-i，-r，-s，-poa のうち一つだけです。
-poa <poa_name>	特定の POA 名に関するインプリメンテーション情報をリスト出力します。一度に指定できるオプションは，-i，-r，-s，-poa のうち一つだけです。
-o <object_name>	特定のオブジェクト名に関するインプリメンテーション情報をリスト出力します。これは，インタフェースまたはリポジトリ ID がコマンド文で指定された場合だけ使用できます。このオプションは，-s 引数または-poa 引数を使用した場合は適用できません。
-host <OAD host name>	特定のリモートホストで実行中の OAD に登録されたオブジェクトのインプリメンテーション情報をリスト出力します。
-verbose	バーボースモードをオンにして，メッセージが stdout に出力されるようにします。
-version	このツールのバージョンを出力します。
-full	OAD に登録されたすべてのインプリメンテーションの状態をリスト出力します。

コードサンプル 15-1 は，oadutil list コマンドの出力例を示しています。

コードサンプル 15-1　oadutil list コマンドの出力例

```
prompt>oadutil list
oadutil list:  located 1 record(s)
Implementation #1:
------------------
repository_id     =   IDL:Bank/Account:1.0
object_name       =   Jack B. Quick
reference data    =
path_name         =   vbj
activation_policy =   SHARED_SERVER
args              =   (length=1)[Server; ]
env               =   NONE
```

(1)　説明

oadutil list ユーティリティは，OAD に登録されたすべての VisiBroker ORB オブジェクトインプリメンテーションをリスト出力できるようにします。各オブジェクトの情報の内容は，次のとおりです。

- VisiBroker ORB オブジェクトのインタフェース名
- そのインプリメンテーションが提供したオブジェクトのインスタンス名
- サーバインプリメンテーションの実行可能ファイルの完全パス名
- VisiBroker ORB オブジェクトの活性化ポリシー（シェアードまたはアンシェアード）
- インプリメンテーションが OAD に登録されたときに指定されたリファレンスデータ
- 活性化時にサーバに渡す引数のリスト
- 活性化時にサーバに渡す環境変数のリスト

インタフェース名とオブジェクト名を指定したローカルリストリクエストの例を次に示します。

例

```
oadutil list -i Bank::AccountManager -o InpriseBank
```

ホスト IP アドレスを指定したリモートリストリクエストの例を次に示します。

例

```
oadutil list -host 206.64.15.198
```

15.3.2　oadutil の使用によるオブジェクトの登録

oadutil コマンドを使用すると，コマンドラインから，またはスクリプト内からオブジェクトインプリメンテーションを登録できます。パラメタは，インタフェース名とオブジェクト名，サービス名，または POA 名, これらに加えてインプリメンテーションを起動する実行可能ファイルへのパス名です。活性化ポリシーを指定しないと, デフォルトとしてシェアードサーバポリシーが使用されます。インプリメンテーションを記述しておき, 開発フェーズとテストフェーズでそのインプリメンテーションを手動で起動できます。インプリメンテーションを配置する準備が整ったら, oadutil を使用するだけで, OAD にそのインプリメンテーションを登録できます。

注

オブジェクトインプリメンテーションを登録する場合は，そのインプリメンテーションオブジェクトを構築したときに使用したのと同じオブジェクト名を使用してください。グローバルスコープを持った，名前付きオブジェクトだけを OAD に登録できます。

oadutil reg コマンドの構文は次のとおりです。

構文

```
oadutil reg [options]
```

注

oadutil reg コマンドを使用するには，ネットワークの少なくとも一つのホストで oad プロセスを起動しなければなりません。

oadutil reg コマンドのオプションには表 15-3 のコマンドライン引数を指定できます。

表 15-3　oadutil reg コマンドのオプション

オプション		説明
必須	-i <interface_name>	特定の IDL インタフェース名を指定します。一度に指定できるオプションは，-i，-r，-s，-poa のうち一つだけです。 リポジトリ ID 指定の詳細については,「15.3　オブジェクト活性化デーモンユーティリティの使用」の「インタフェース名からリポジトリ ID への変換」を参照してください。
	-r < repository_id>	特定のリポジトリ ID を指定します。一度に指定できるオプションは, -i, -r，-s，-poa のうち一つだけです。
	-s <service name>	特定のサービス名を指定します。一度に指定できるオプションは，-i，-r, -s，-poa のうち一つだけです。
	-poa <poa_name>	このオプションは，オブジェクトインプリメンテーションの代わりに POA を登録する場合に使用してください。一度に指定できるオプションは，-i，-r，-s，-poa のうち一つだけです。
	-o <object_name>	特定のオブジェクトを指定します。これは，インタフェース名またはリポジトリ ID がコマンド文で指定された場合だけ使用できます。このオプションは，-s 引数または-poa 引数を使用している場合には適用できません。
	-cpp <file name to execute>	-o/-r/-s/-poa 引数に一致するオブジェクトを生成し登録しなければならない実行可能ファイルの完全パス名を指定します。-cpp 引数で登録されたアプリケーションは，スタンドアロン実行可能ファイルでなければなりません。
	-java <full class name>	メインルーチンを含む Java クラスの完全名を指定します。このアプリケーションは，-o/-r/-s/-poa 引数に一致するオブジェクトを生成し登録しなければなりません。-java 引数で登録されたクラスは，コマンド vbj <full_classname>で実行されます。
任意指定	-host <OAD host name>	OAD を実行中である特定のリモートホストを指定します。
	-verbose	バーボースモードをオンにして，メッセージが stdout に出力されるようにします。
	-version	このツールのバージョンを出力します。
	-d <referenceData>	活性化時にサーバに渡すリファレンスデータを指定します。
	-a arg1 -a arg2	生成された実行可能ファイルにコマンドライン引数として渡す引数を指定します。複数の-a (arg)パラメタを使用して引数を渡せます。これらの引数は，生成された実行可能ファイルを生成するためにプロパゲートされます。

オプション		説明
任意指定	-e env1 -e env2	生成された実行可能ファイルに渡す環境変数を指定します。複数の-e (env)パラメタを使用して引数を渡せます。これらの引数は，生成された実行可能ファイルを生成するためにプロパゲートされます。
	-p {shared|unshared}	生成されたオブジェクトの活性化ポリシーを指定します。デフォルトポリシーは SHARED_SERVER です。 shared 指定時は，任意のオブジェクトの複数のクライアントが同じインプリメンテーションを共用します。OAD が一度に活性化するサーバは一つだけです。 unshared 指定時は，任意のインプリメンテーションの一つのクライアントだけが，活性化されたサーバにバインドされます。複数のクライアントが同じオブジェクトインプリメンテーションにバインドしたがると，クライアントアプリケーションごとに別々のサーバが活性化されます。クライアントアプリケーションが切断または終了すると，対応するサーバが終了します。

(1)　例 1：リポジトリ ID の指定

次に示すコマンドは，OAD に Borland Enterprise Server VisiBroker プログラム factory を登録します。これは，リポジトリ ID が IDL:ehTest/Factory:1.0（インタフェース名 ehTest::Factory に対応）のオブジェクトがリクエストされると活性化されます。活性化対象のオブジェクトのインスタンス名は ReentrantServer であり，その名前も，生成された実行可能ファイルにコマンドライン引数として渡されます。このサーバにはアンシェアードポリシーが指定されているので，リクエスト元のクライアントが生成されたサーバとのコネクションを切断すると，サーバが終了します。

C++の場合

```
prompt> oadutil reg -r IDL:ehTest/Factory:1.0   ¥
        -o ReentrantServer -cpp /home/developer¥
        /Project1/factory_r -a ReentrantServer  ¥
        -p unshared
```

Java の場合

```
prompt> oadutil reg -r IDL:ehTest/Factory:1.0  ¥
        -o ReentrantServer -java factory_r     ¥
        -a ReentrantServer -p unshared
```

注

上記の例では，指定された Java クラスが CLASSPATH になければなりません。

(2)　例 2：IDL インタフェース名の指定

次に示すコマンドは，OAD に Borland Enterprise Server VisiBroker クラス Server を登録します。この例では，指定されたクラスは，リポジトリ ID が IDL:Bank/AccountManager:1.0（インタフェース名 IDL 名 Bank::AccountManager に対応）で，インスタンス名が CreditUnion のオブジェクトを活性化しなければなりません。サーバはアンシェアードポリシーで起動されるので，リクエスト元のクライアントがコネクションを切断すると，サーバは確実に終了します。

サンプル（Java）

```
prompt> oadutil reg -i Bank::AccountManager -o CreditUnion ¥
        -java Server -a CreditUnion -p unshared -e DEBUG=1
```

注

上記の例では，指定された Java クラスが CLASSPATH になければなりません。

上記の登録は，リクエストされたサーバの生成時に次のコマンドを実行するよう OAD に指示します。

```
vbj -DDEBUG=1 Server CreditUnion
```

(3)　OAD へのリモート登録

リモートホストの OAD にインプリメンテーションを登録するには，oadutil reg に-host 引数を指定してください。

UNIX シェルから Windows の OAD へのリモート登録の実行例を次に示します。¥記号を oadutil に渡す前にシェルに解釈させないようにするには，二重¥記号（¥¥）が必要です。

例

```
prompt> oadutil reg -r IDL:Library:1.0 Harvard            ¥
        -cpp c:¥¥vbroker¥¥examples¥¥library¥¥libsrv.exe   ¥
        -p shared -host 100.64.15.198
```

15.3.3　オブジェクトの複数のインスタンスの区別

インプリメンテーションは ReferenceData を使用することで，同じオブジェクトの複数のインスタンスを区別できます。リファレンスデータの値はオブジェクト生成時にインプリメンテーションによって選択され，オブジェクトの存続期間中は一定に保たれます。ReferenceData typedef は複数のプラットフォームと VisiBroker ORB の間で移植できます。

注

Borland Enterprise Server VisiBroker は，生成しているオブジェクトのインタフェースを識別するために，CORBA が定義する inf_ptr を使用しません。Borland Enterprise Server VisiBroker で生成するアプリケーションは，常にこのパラメタで NULL 値を指定しなければなりません。

15.3.4　CreationImplDef クラスの使用による活性化プロパティの設定

CreationImplDef クラスには，OAD が VisiBroker ORB オブジェクトを活性化するために必要な path_name，activation_policy，args，および env というプロパティがあります。IDL サンプル 15-1 に，CreationImplDef struct を示します。

path_name プロパティには，オブジェクトをインプリメントする実行可能プログラムの正確なパス名を指定します。activation_policy プロパティは，IDL サンプル 15-4 に示すサーバの活性化ポリシーを表します。args プロパティと env プロパティはサーバのコマンドライン引数と環境設定を表します。

IDL サンプル 15-1　CreationImplDef IDL

```
module extension {
. . .

   enum Policy {
      SHARED_SERVER,
      UNSHARED_SERVER
   };

   struct CreationImplDef {
      CORBA::RepositoryId      repository_id;
      string                   object_name;
      CORBA::ReferenceData     id;
      string                   path_name;
      Policy                   activation_policy;
      CORBA::StringSequence    args;
      CORBA::StringSequence    env;
   };
. . .
};
```

15.3.5　VisiBroker ORB インプリメンテーションの動的変更

IDL サンプル 15-2 に，オブジェクトの登録を動的に変更するために使用できる change_implementation()メソッドを示します。このメソッドを使用して，オブジェクトの活性化ポリシー，パス名，引数，および環境変数を変更できます。

IDL サンプル 15-2　change_implementation

```
module Activation
{
. . .
   void change_implementation(
         in extension::CreationImplDef old_info,
         in extension::CreationImplDef new_info)
      raises ( NotRegistered, InvalidPath, IsActive );
. . .
};
```

注

オブジェクトのインプリメンテーション名とオブジェクト名は change_implementation()メソッドを使用して変更できますが，注意が必要です。変更すると，クライアントプログラムはそのオブジェクトを昔の名前で探せなくなります。

15.3.6　OAD::reg_implementation を使用した OAD の登録

oadutil reg コマンドを手動でまたはスクリプトで使用しなくても，Borland Enterprise Server VisiBroker では，クライアントアプリケーションが OAD::reg_implementation オペレーションを使って一つ以上のオブジェクトを活性化デーモンに登録できます。このオペレーションを使用すると，オブジェクトインプリメンテーションが OAD と osagent に登録されます。OAD は情報をインプリメンテーションリポジトリに格納し，クライアントがオブジェクトにバインドしようとしたときにオブジェクトインプリメンテーションを探し，活性化できるようにします。

IDL サンプル 15-3　OAD::reg_implementation オペレーション

```
module Activation {
. . .
   typedef sequence<ObjectStatus> ObjectStatus List;
. . .
   typedef sequence<InplementationStatus> ImplStatusList;
. . .
   interface OAD {
      // Register an implementation.
      Object reg_implementation(
            in extension::CreationImplDef impl)
         raises (DuplicateEntry, InvalidPath);
   }
}
```

CreationImplDef struct には OAD が必要とするプロパティがあります。プロパティには，repository_id，object_name，id，path_name，activation_policy，args，および env があります。これらの値を設定したり照会したりするオペレーションも用意されています。OAD はこれらの追加プロパティを使用して VisiBroker ORB オブジェクトを活性化します。

IDL サンプル 15-4　CreationImplDef インタフェース

```
struct CreationImplDef {
   CORBA::RepositoryId repository_id;
   string object_name;
   CORBA::ReferenceData id;
   string path_name;
   Policy activation_policy;
   CORBA::StringSequence args;
   CORBA::StringSequence env;
};
```

path_name プロパティには，オブジェクトをインプリメントする実行可能プログラムの正確なパス名を指定します。activation_policy プロパティはサーバの活性化ポリシーを表します。args プロパティと env プロパティはサーバに渡されるオプションの引数と環境設定を表します。

15.3.7　オブジェクトの生成と登録の例

コードサンプル 15-2 とコードサンプル 15-3 に，CreationImplDef クラスと OAD.reg_implementation()メソッドを使用してサーバを OAD に登録する方法を示します。この機能は独立した管理プロセスで使用されますが，オブジェクトインプリメンテーションそのもので使用する必要はありません。オブジェクトインプリメンテーションで使用する場合，これらのタスクをオブジェクトインプリメンテーションの活性化より前に実行しなければなりません。

コードサンプル 15-2　VisiBroker ORB オブジェクトの生成と OAD への登録（C++）

```
#include "oad_c.hh"
// USE_STD_NS is a define setup by VisiBroker to use the std namespace
USE_STD_NS

int main(int argc, char* const* argv)
{
   try {
      // Initialize the ORB.
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

      Activation::OAD_var anOAD = Activation::OAD::bind();

      // Create an ImplDef
      extension::CreationImplDef_var _implDef =
            anOAD->create_CreationImplDef();

      _implDef->repository_id = "IDL:Bank/AccountManager:1.0";
      _implDef->object_name = "BankManager";
      _implDef->path_name = "/user/TPBrokerV5/Server";
      _implDef->activation_policy = extension::SHARED_SERVER;
      try {
             anOAD->reg_implementation(
                    *((extension::CreationImplDef*)_implDef));

      } catch(const CORBA::Exception& e) {
            cerr << "reg_implementation Failed:" <<endl;
                  cerr << e << endl;
            return 1;
      }
   }
   catch(const CORBA::Exception& e) {
      cerr << e << endl;
      return 1;
   }

   return 0;
}
```

コードサンプル 15-3　VisiBroker ORB オブジェクトの生成と OAD への登録（Java）

```
// Register.java
import com.inprise.vbroker.Activation.*;
import com.inprise.vbroker.extension.*;

public class Register{

   public static void main(String[ ] args) {
      // Initialize the ORB.
      org.omg.CORBA.ORB orb =
                        org.omg.CORBA.ORB.init(args,null);
      // Locate an OAD

      try {
         OAD anOAD =
            OADHelper.bind(orb);
```

```
            // Create an ImplDef
            CreationImplDef _implDef = new com.inprise.vbroker.
               extension.CreationImplDef();
            _implDef.repository_id =
                                 "IDL:Bank/AccountManager:1.0";
            _implDef.object_name = "BankManager";
            _implDef.path_name = "vbj";
            _implDef.id = new byte[0];
            _implDef.activation_policy = com.inprise.vbroker.
               extension.Policy.SHARED_SERVER;
            _implDef.env = new String[0];

            String[ ] str = new String[1];
            str[0] = "Server";
            _implDef.args = str;
            try {
               anOAD.reg_implementation(_implDef);
            } catch (Exception e) {
               System.out.println("Caught " + e);
            }

         }
         catch (org.omg.CORBA.NO_IMPLEMENT e) {
         }
      }
   }
```

15.3.8　OAD が渡す引数

OAD はオブジェクトインプリメンテーションを開始するときに，そのインプリメンテーションが OAD に登録された時点で指定された引数をすべて渡します。

15.4　オブジェクトの登録解除

オブジェクトによって提供されたサービスがもう使用できなくなったか，一時的に停止されたら，そのオブジェクトを OAD から登録解除しなければなりません。VisiBroker ORB オブジェクトを登録解除すると，インプリメンテーションリポジトリからそのオブジェクトが削除されます。オブジェクトはスマートエージェントのディクショナリからも削除されます。オブジェクトが登録解除されると，クライアントプログラムはそのオブジェクトを探したり，使用したりできなくなります。また，OAD.change_implementation() メソッドを使用して，そのオブジェクトのインプリメンテーションを変更することもできなくなります。登録プロセスと同様，登録解除はコマンドラインまたはプログラムを使用して実行できます。ここで説明する OAD との VisiBroker ORB オブジェクトインタフェースもあります。

15.4.1　oadutil ツールの使用によるオブジェクトの登録解除

oadutil unreg コマンドを使用すると，OAD に登録された一つ以上のオブジェクトインプリメンテーションを登録解除できます。いったん登録解除されたオブジェクトは，クライアントがオブジェクトをリクエストしても，もう OAD によって自動的に活性化できません。oadutil unreg コマンドで登録解除できるのは，前回 oadutil reg コマンドによって登録されたオブジェクトだけです。

インタフェース名だけを指定すると，そのインタフェースに対応するすべての VisiBroker ORB オブジェクトが登録解除されます。または，インタフェース名とオブジェクト名によって VisiBroker ORB オブジェクトを特定できます。オブジェクトを登録解除すると，そのオブジェクトに対応するすべてのプロセスが終了します。

注

oadutil unreg コマンドを使用する前に，ネットワークの少なくとも一つのホストで oad プロセスを起動しなければなりません。

oadutil unreg コマンドの構文は次のとおりです。

構文

```
oadutil unreg [options]
```

oadutil unreg コマンドには表 15-4 のコマンドライン引数を指定できます。

表 15-4　oadutil unreg コマンドのコマンドライン引数

オプション		説明
必須	-i <interface_name>	特定の IDL インタフェース名を指定します。一度に指定できるオプションは，-i，-r，-s，-poa のうち一つだけです。 リポジトリ ID 指定の詳細については，「15.3　オブジェクト活性化デーモンユーティリティの使用」の「インタフェース名からリポジトリ ID への変換」を参照してください。
	-r <repository_id>	特定のリポジトリ ID を指定します。一度に指定できるオプションは，-i，-r，-s，-poa のうち一つだけです。
	-s <service name>	特定のサービス名を指定します。一度に指定できるオプションは，-i，-r，-s，-poa のうち一つだけです。
	-o <object_name>	特定のオブジェクト名を指定します。これは，インタフェース名またはリポジトリ ID がコマンド文で指定された場合だけ使用できます。このオプションは，-s 引数または-poa 引数を使用している場合には適用できません。

オプション		説明
必須	-poa <POA_name>	oadutil reg -poa <POA_name>を使用して登録された POA を，登録解除します。
任意指定	-host <host name>	OAD を実行中であるホスト名を指定します。
	-verbose	バーボースモードを有効にして，メッセージが stdout に出力されるようにします。
	-version	このツールのバージョンを出力します。

登録解除例

oadutil unreg ユーティリティは，次に示す三つの場所から一つ以上の VisiBroker ORB オブジェクトを登録解除します。

- OAD
- インプリメンテーションリポジトリ
- スマートエージェント

oadutil unreg コマンドの使用例を次に示します。このコマンドは，InpriseBank という名前を指定した Bank::AccountManager のインプリメンテーションをローカル OAD から登録解除します。

```
oadutil unreg -i Bank::AccountManager -o InpriseBank
```

15.4.2　OAD オペレーションを使用した登録解除

オブジェクトのインプリメンテーションは，OAD インタフェースのどのオペレーションまたは属性を使用しても VisiBroker ORB オブジェクトの登録を解除できます。

- unreg_implementation(in CORBA::RepositoryId repId, in string object_name)
- unreg_interface(in CORBA:: RepositoryId repId)
- unregister_all()
- attribute boolean destroy_on_unregister

unreg_implementation()

このオペレーションは，特定のリポジトリ ID とオブジェクト名を使用してインプリメンテーションを登録解除したい場合に使用します。このオペレーションは，指定されたリポジトリ ID とオブジェクト名を現在インプリメントしているすべてのプロセスを終了します。

unreg_interface()

このオペレーションは，特定のリポジトリ ID だけを使用してインプリメンテーションを登録解除したい場合に使用します。このオペレーションは，指定されたリポジトリ ID を現在インプリメントしているすべてのプロセスを終了します。

unregister_all()

このオペレーションは，すべてのインプリメンテーションを登録解除するために使用します。destroyActive が true に設定されていないかぎり，アクティブなインプリメンテーションはどれも実行を続けます。下位互換性を保つため，unregister_all()メソッドも使用できます。これは unregister_all_destroy(false)メソッドを呼び出すのと同じです。

destroy_on_unregister

この属性は，関連したインプリメンテーションの登録解除時に，発生したすべてのプロセスを解放する場合に使用します。デフォルト値は false です。

IDL サンプル 15-5　OAD 登録解除オペレーション

```
module Activation {
. . .
   interface OAD {
          . . .
      void unreg_implementation(
            in CORBA::RepositoryId repId,
            in string object_name)
         raises(NotRegistered);
          . . .
      }
}
```

15.4.3　インプリメンテーションリポジトリの内容表示

oadutil コマンドを使用すると，個々のインプリメンテーションリポジトリの内容をリスト出力できます。oadutil コマンドは，リポジトリ内のインプリメンテーションごとに，すべてのオブジェクトインスタンス名，実行可能プログラムのパス名，活性化モード，およびリファレンスデータをリスト表示します。実行可能プログラムに渡される引数または環境変数もすべてリスト表示されます。

15.5　OAD との IDL インタフェース

OAD は VisiBroker ORB オブジェクトとしてインプリメントされており，クライアントプログラムが OAD にバインドして，登録されたオブジェクトの状態を照会するためにインタフェースを使用できるようにしています。IDL サンプル 15-6 に，OAD の IDL インタフェース定義を示します。

IDL サンプル 15-6　OAD インタフェース定義

```
module Activation
{
   enum State {
      ACTIVE,
      INACTIVE,
      WAITING_FOR_ACTIVATION
   };
   struct ObjectStatus {
      long unique_id;
      State activation_state;
      Object objRef;
   };
   typedef sequence<ObjectStatus> ObjectStatusList;
   struct ImplementationStatus {
      extension::CreationImplDef impl;
      ObjectStatusList status;
   };
   typedef sequence<ImplementationStatus> ImplStatusList;

   exception DuplicateEntry {};
   exception InvalidPath {};
   exception NotRegistered {};
   exception FailedToExecute {};
   exception NotResponding {};
   exception IsActive {};
   exception Busy {};

   interface OAD {
      Object reg_implementation(
            in extension::CreationImplDef impl)
              raises (DuplicateEntry, InvalidPath);
      extension::CreationImplDef get_implementation(
            in CORBA::RepositoryId repId,
            in string object_name)
              raises ( NotRegistered);
      void change_implementation(
            in extension::CreationImplDef old_info,
            in extension::CreationImplDef new_info)
              raises (NotRegistered,InvalidPath,IsActive);
      attribute boolean destroy_on_unregister;
      void unreg_implementation(in CORBA::RepositoryId repId,
            in string object_name)
              raises ( NotRegistered );
      void unreg_interface(in CORBA::RepositoryId repId)
              raises ( NotRegistered );
      void unregister_all();
      ImplementationStatus get_status(
            in CORBA::RepositoryId repId,
            in string object_name)
              raises ( NotRegistered );
      ImplStatusList get_status_interface(
            in CORBA::RepositoryId repId)
              raises (NotRegistered);
      ImplStatusList get_status_all();
   };
};
```

16 インタフェースリポジトリの使用

この章では，IR（インタフェースリポジトリ）を生成し，Borland Enterprise Server VisiBroker ユーティリティまたは独自のコードを使用してそれにアクセスする方法について説明します。

IR には CORBA オブジェクトインタフェースの記述が入っています。IR 内のデータは IDL ファイル内のデータと同じもので，モジュール，インタフェース，オペレーション，およびパラメタの記述ですが，クライアントによるランタイムアクセス用に構成されています。クライアントは，IR（開発者用のオンライン参照ツールとしての機能を果たすこともある）を検索するか，参照対象の任意のオブジェクトのインタフェースを（動的起動インタフェースによってオブジェクトを起動するための準備として）検索できます。

16.1　インタフェースリポジトリとは

IR（インタフェースリポジトリ）は，CORBA オブジェクトのインタフェース情報のデータベースのようなもので，クライアントはこれを使用してランタイム時にインタフェースを調べたり更新したりできます。「13.　ロケーションサービスの使用」で説明したオブジェクトのインスタンスを記述するデータを保持する Borland Enterprise Server VisiBroker ロケーションサービスとは対照的に，IR のデータはインタフェース（型）を記述します。IR 内に格納されたインタフェースの要求を満たす使用可能なインスタンスが必ず存在するとは限りません。IR 内の情報は（一つまたは複数の）IDL ファイル（または複数ファイル）内の情報と同じものですが，クライアントがランタイムに使用しやすい方法で実装されています。

IR を使用するクライアントは，「17.　動的起動インタフェースの使用」で説明する DII（動的起動インタフェース）も使用できます。このようなクライアントは, IR を使用して未知のオブジェクトのインタフェースについて照会し，DII を使用してオブジェクトのメソッドを呼び出します。ただし，IR と DII の間にコネクションを確立する必要はありません。例えば，IR を使用して開発者向けの「IDL ブラウザ」ツールを作成できますが，このツールでは，メソッドの記述をブラウザからエディタにドラッグすると，テンプレートのメソッド呼び出しが開発者のソースコードに挿入されます。この場合，DII を使用しないで IR だけを使用します。

IR は，IR サーバ（インプリメンテーション）である Borland Enterprise Server VisiBroker の irep プログラムを使用して生成します。Borland Enterprise Server VisiBroker の idl2ir プログラムを使用して IR の更新やデータ入力ができるほか，IR の照会や更新，またはその両方を行う独自の IR クライアントを作成することもできます。

16.1.1　IR の内容

IR 内のオブジェクトは階層構造になっており，そのオブジェクトのメソッドがインタフェースに関する情報を明らかにします。通常，インタフェースはオブジェクトを記述するものと考えられていますが，CORBA 環境では，オブジェクトの集まりを使用してインタフェースを記述することに意味があります。理由は，この方法によってデータベースなどの新しいメカニズムが不要になるためです。

IR に入れることができるオブジェクトの種類の例として，IDL ファイルに IDL モジュール定義を，モジュールにインタフェース定義を，インタフェースにオペレーション（メソッド）定義をそれぞれ入れることができるということを考えてください。これに対応して，IR に ModuleDef オブジェクトを，ModuleDef オブジェクトに InterfaceDef オブジェクトを，InterfaceDef オブジェクトに OperationDef オブジェクトをそれぞれ入れることができます。このため，IR ModuleDef から，その中にどのような InterfaceDef が入っているがわかります。逆に, ある InterfaceDef がどの ModuleDef に入っているかがわかります。そのほかのすべての IDL 構成体，例えば例外，属性，valuetype も IR の中で表現できます。

IR にはタイプコードも入っています。タイプコードは IDL ファイル内で明示的に示されるのではなく，IDL ファイル内で定義または記述された型（long，string，struct など）から自動的に派生されます。タイプコードは，CORBA の any 型（任意の型を示し，動的起動インタフェースで使用される汎用型）のインスタンスのエンコードとデコードに使用されます。

16.1.2　使用できる IR の数

IR は，ほかのオブジェクトと同様に，幾つでも生成できます。IR の生成または使用について，Borland Enterprise Server VisiBroker が規定するポリシーはありません。IR をサイトにどのように配置し命名するかは，プログラマで決めてください。例えば，中央の IR にすべての「生成」オブジェクトのインタフェースを入れ，個々の開発者は各自のテスト用の IR を生成するという規則を採用してもかまいません。

注

IR は書き込み可能であり，アクセス制御によってプロテクトされていません。クライアントが誤って，または意図的に IR を破壊したり，IR から機密情報を取得したりするおそれがあります。

すべてのオブジェクト用に定義された_get_interface メソッド（C++）または_get_interface_def メソッド（Java）を使用したい場合は，VisiBroker ORB が IR 内のインタフェースを検索できるよう，最低一つの IR サーバを実行していなければなりません。使用可能な IR がないか, VisiBroker ORB のバインド先である IR がそのオブジェクト用のインタフェース定義と一緒にロードされていない場合，_get_interface メソッド（C++）または_get_interface_def メソッド（Java）は，NO_IMPLEMENT 例外を発生させます。

16.2　irep を使用した IR の生成と表示

Borland Enterprise Server VisiBroker の IR サーバは irep と呼ばれ，bin ディレクトリにあります。irep プログラムはデーモンとして実行します。irep は，あらゆるオブジェクトインプリメンテーションと同様に，OAD（オブジェクト活性化デーモン）に登録できます。oadutil ツールは，（CORBA::Repository などのインタフェース名ではなく）IDL:org.omg/CORBA/ Repository:1.0 のようなオブジェクト ID を必要とします。

16.2.1　irep を使用した IR の生成

IR を生成し，その内容を表示するには，irep プログラムを使用します。irep プログラムを使用するための構文は次のとおりです。

構文

```
irep <driverOptions> <otherOptions> IRepName [file.idl]
```

表 16-1 に，irep で IR を生成するための構文を示します。

表 16–1　irep で IR を生成するための構文

構文	説明
IRepName	IR のインスタンス名を指定します。クライアントは，この名前を指定することによって，この IR のインスタンスにバインドできます。
file.idl	irep が，自分の生成した IR にロードする IDL ファイルを指定します。irep は終了時に IR の内容をこのファイルに格納します。ファイルを指定しなければ，irep は空の IR を生成します。

表 16-2 に，irep のオプションの定義を示します。

表 16–2　irep のオプション

オプション		説明
ドライバオプション	-J<java option>	JavaVM にオプションを直接渡します。
	-VBJversion	VisiBroker のバージョンを出力します。
	-VBJdebug	VisiBroker のデバッグ情報を出力します。
	-VBJclasspath	クラスパスを指定します。これは CLASSPATH 環境変数の指定より優先されます。
	-VBJprop <name>[=<value>]	名前・値のペアを JavaVM に渡します。
	-VBJjavavm <jvmpath>	JavaVM パスを指定します。
	-VBJaddJar <jarfile>	JavaVM を実行する前に，CLASSPATH に jarfile を追加します。
そのほかのオプション	-D, -define foo[=bar]	プリプロセサマクロ（任意で値も指定できる）を定義します。
	-I, -include <dir>	#include サーチ用に追加ディレクトリを指定します。
	-P, -no_line_directives	プリプロセサから#line ディレクティブを発しません。デフォルトはオフです。

オプション		説明
そのほかのオプション	-H, -list_includes	見つかった#included ファイル名をそのまま表示します。デフォルトはオフです。
	-C, -retain_comments	前処理された出力にコメントを保持します。デフォルトはオフです。
	-U, -undefine foo	プリプロセサマクロの定義を削除します。
	-[no_]idl_strict	IDL ソースを厳密に OMG 標準解釈させます。デフォルトはオフです。
	-[no_]warn_unrecognized_pragmas	#pragma が認識されない場合に警告します。デフォルトはオンです。
	-[no_]back_compat_mapping	VisiBroker 3.x 対応のマッピングを使用します。
	-h, -help, -usage, -?	ヘルプを出力します。
	-version	ソフトウェアバージョン番号を表示します。
	-install <service name>	NT サービスとしてインストールします。
	-remove <service name>	この NT サービスを取り外します。

次の例は，TestIR という IR を Bank.idl というファイルから生成する方法を示しています。

例

```
irep TestIR Bank.idl
```

16.2.2　IR の内容表示

Borland Enterprise Server VisiBroker の ir2idl ユーティリティまたは Borland Enterprise Server VisiBroker コンソールアプリケーションを使用して IR の内容を表示できます。ir2idl ユーティリティの構文は次のとおりです。

構文

```
ir2idl -irep IRname
```

表 16-3 に，irep で IR の内容を表示するための構文を示します。

表 16-3　irep で IR の内容を表示するための構文

構文	説明
-irep IRname	IRname に指定した IR インスタンスにバインドするようプログラムに指示します。

ir2idl ユーティリティ引数の詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「ir2idl」の記述を参照してください。

16.3　idl2ir を使用した IR の更新

IR クライアントである Borland Enterprise Server VisiBroker idl2ir ユーティリティを使用して，IR を更新できます。idl2ir ユーティリティの構文は次のとおりです。

構文

```
idl2ir [arguments] idl_file_list
```

idl2ir ユーティリティ引数の詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「idl2ir」の記述を参照してください。

次の例は，TestIR という IR を Bank.idl ファイルからの定義によって更新する方法を示しています。

例

```
idl2ir -irep TestIR -replace Bank.idl
```

IR 内の項目は，idl2ir ユーティリティや irep ユーティリティを使用しても削除できません。項目の削除は次の手順で行います。

1. irep プログラムを終了するか中止します。
2. irep コマンドラインで指定した IDL ファイルを編集します。
3. 更新したファイルを使用して irep を再び起動します。

IR はシンプルなトランザクションサービスを持っています。指定された IDL ファイルがロードに失敗すると，IR はその内容を前の状態までロールバックします。IDL のロード後は，IR は以降のトランザクションで使用する状態にコミットします。どのリポジトリに対しても，ロールバック用にトランザクションの状態を格納した IRname.rollback ファイルがホームディレクトリにあります。

注

IR のすべての項目を削除したい場合，その内容を新しい空の IDL ファイルと置き換えることができます。例えば，Empty.idl という名前の IDL ファイルを使用すると，次のコマンドを実行できます。

```
idl2ir -irep TestIR -replace Empty.idl
```

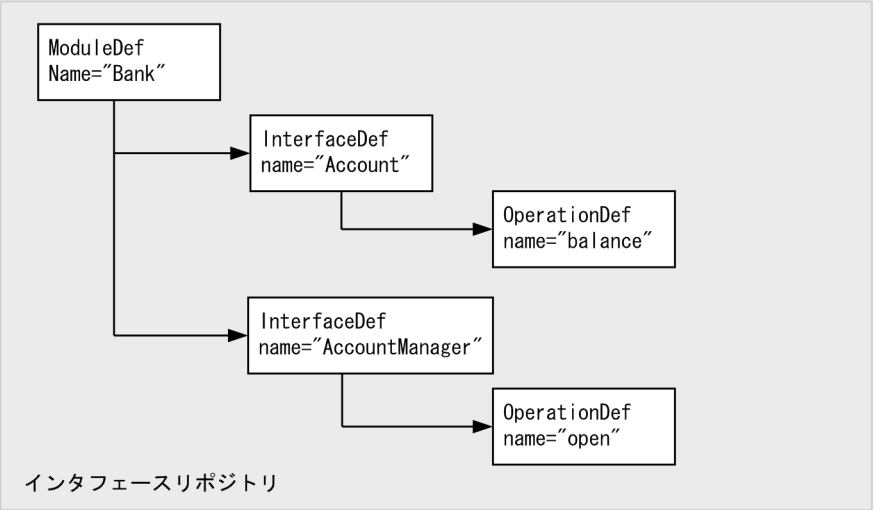
16.4　IR の構造の理解

IR 内のオブジェクトは階層構造になっており，その階層は，IDL 指定でインタフェースが定義されている方法に対応しています。IDL モジュール定義が幾つかのインタフェース定義を含むことがあるのと同様に，IR 内の一部のオブジェクトがほかのオブジェクトを含むことがあります。IDL サンプル 16-1 に示す IDL ファイルが，IR 内でどのようにしてオブジェクトの階層に変換されるのかを図 16-1 に示します。

IDL サンプル 16-1　Bank.idl ファイル

```
// Bank.idl
module Bank {
   interface Account {
      float balance();
   };
   interface AccountManager {
      Account open(in string name);
   };
};
```

図 16–1　Bank.idl の IR オブジェクト階層



OperationDef オブジェクトには，パラメタとリターンタイプを保持する追加データ構造体（インタフェースではない）のリファレンスが入っています。

16.4.1　IR 内のオブジェクトの識別

表 16-4 に，IR オブジェクトの識別と分類のために提供されるオブジェクトを示します。

表 16–4　IR オブジェクトの識別と分類に使用するオブジェクト

項目	説明
name	IDL 指定の中でモジュール，インタフェース，オペレーションなどに割り当てられた識別子に対応する文字列。識別子は一意でなくてもかまいません。
id	IRObject を一意に識別する文字列。RepositoryID には三つのコンポーネントがコロン（:）デリミタで区切られて入っています。最初のコンポーネントは「IDL:」で，最後は「:1.0」などのバージョン番号です。2 番目のコンポーネントはスラント（/）文字で区切られた一連の識別子です。通常，最初の識別子は一意のプリフィクスです。
def_kind	IR オブジェクトとして可能なすべての型を表す値を定義した列挙体です。

16.4.2　IR に格納できるオブジェクトの型

IR に入れることができるオブジェクトを表 16-5 に示します。これらのオブジェクトのほとんどは IDL 構文要素に対応しています。例えば，StructDef には IDL の struct 宣言と同じ情報が入っており，InterfaceDef には IDL のインタフェース宣言と同じ情報が入っています。IDL の基本型（boolean，long など）宣言と同じ情報が入っている PrimitiveDef に至るまで，すべてこのように対応しています。

表 16–5　IR に格納できるオブジェクト

オブジェクトタイプ	説明
Repository	ほかのすべてのオブジェクトが入っている最上位モジュールを示します。
ModuleDef	IDL モジュール宣言を示します。この中には，ModuleDef，InterfaceDef，ConstantDef，AliasDef，ExceptionDef，および IDL モジュールで定義できるそのほかの IDL 構成体に相当する IR オブジェクトを入れることができます。
InterfaceDef	IDL インタフェース宣言を示し，OperationDef，ExceptionDef，AliasDef，ConstantDef，および AttributeDef が入っています。
AttributeDef	IDL 属性宣言を示します。
OperationDef	IDL オペレーション（メソッド）宣言を示します。インタフェースのオペレーションを定義します。これには，そのオペレーションに必要なパラメタのリスト，リターン値，そのオペレーションによって発生する可能性がある例外のリスト，およびコンテキストのリストが含まれます。
ConstantDef	IDL 定数宣言を示します。
ExceptionDef	IDL 例外宣言を示します。
ValueDef	定数, 型, 値メンバ, 例外, オペレーション, および属性のリストが入っている valuetype 定義を示します。
ValueBoxDef	ほかの IDL 型の，ボックスに入った単純な valuetype を示します。
ValueMemberDef	valuetype のメンバを示します。
NativeDef	ネイティブ定義を示します。ユーザは独自のネイティブ定義ができます。
StructDef	IDL 構造体宣言を示します。
UnionDef	IDL union 宣言を示します。
EnumDef	IDL 列挙体宣言を示します。
AliasDef	IDL typedef 宣言を示します。IR TypedefDef インタフェースは，StructDef，UnionDef，およびそのほかの共通オペレーションを定義する基本インタフェースであることに注意してください。
StringDef	IDL バウンデッド string 宣言を示します。
SequenceDef	IDL sequence 宣言を示します。
ArrayDef	IDL 配列宣言を示します。
PrimitiveDef	IDL 基本宣言（null，void，long，ushort，ulong，float，double，boolean，char，octet，any，TypeCode，Principal，string，objref，longlong，ulonglong，longdouble，wchar，および wstring）を示します。

16.4.3　継承されるインタフェース

共通メソッドを定義する三つの実体化できない（つまり抽象的な）IDL インタフェースが，IR 内の多数のオブジェクト（表 16-5 参照）に継承されます。表 16-6 に，これらの広く継承されるインタフェースを示します。これらのインタフェースのほかのメソッドの詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「インタフェースリポジトリインタフェースとクラス」の記述を参照してください。

表 16-6　多数の IR オブジェクトが継承するインタフェース

インタフェース	インタフェースを継承するオブジェクト	主要な照会メソッド
IRObject	Repository を含むすべての IR オブジェクト	def_kind() モジュールまたはインタフェースなどの IR オブジェクトの定義の種類を返します。 destroy() IR オブジェクトをデストラクトします。
Container	モジュールまたはインタフェースのような，そのほかの IR オブジェクトを入れることができる IR オブジェクト	lookup() 入っているオブジェクトを名前で検索します。 contents() Container 内のオブジェクトをリスト表示します。 describe_contents() Container 内のオブジェクトを記述します。
Contained	ほかのオブジェクト，つまり Container の中に入れることができる IR オブジェクト	name() このオブジェクトの名前です。 defined_in() オブジェクトを含む Container です。 describe() オブジェクトを記述します。 move() オブジェクトをほかのコンテナに移します。

16.5　IR へのアクセス

クライアントプログラムは，IR の IDL インタフェースを使用して，IR 内のオブジェクトについて情報を取得できます。クライアントプログラムは Repository にバインドして，コードサンプル 16-1 に示すメソッドを呼び出せます。このインタフェースの詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「インタフェースリポジトリインタフェースとクラス」の記述を参照してください。

コードサンプル 16-1　リポジトリクラス（C++）

```
class CORBA {
   class Repository : public Container {
      . . .
      CORBA::Contained_ptr lookup_id(
            const char * search_id);
      CORBA::PrimitiveDef_ptr get_primitive(
            CORBA::PrimitiveKind kind);
      CORBA::StringDef_ptr create_string(
            CORBA::ULong bound);
      CORBA::SequenceDef_ptr create_sequence(
            CORBA::ULong bound,
            CORBA::IDLType_ptr element_type);
      CORBA::ArrayDef_ptr create_array(
            CORBA::ULong length,
            CORBA::IDLType_ptr element_type);
      . . .
   };
   . . .
};
```

注

IR を使用するプログラムは，-D_VIS_INCLUDE_IR フラグでコンパイルする必要があります。

コードサンプル 16-2　リポジトリインタフェース（Java）

```
package org.omg.CORBA;
public interface Repository extends Container {
   . . .
   org.omg.CORBA.Contained lookup_id(string id);
   org.omg.CORBA.PrimitiveDef get_primitive(
                     org.omg.CORBA.PrimitiveKind kind);
   org.omg.CORBA.StringDef create_string(long bound);
   org.omg.CORBA.SequenceDef create_sequence(long bound,
                     org.omg.CORBA.IDLType element_type);
   org.omg.CORBA.ArrayDef create_array(long length,
                     org.omg.CORBA.IDLType element_type);
   . . .
}
```

16.6　サンプルプログラム

ここでは，アカウントを生成して（再び）オープンするための単純な AccountManager インタフェースを含む，IR の簡単なサンプルを示します。コードは examples¥vbe¥ir ディレクトリ内にあります。初期化時に AccountManager インプリメンテーションは，管理されたアカウントインタフェースの IR 定義と接続します。

これは，特定の Account インプリメンテーションがすでにインプリメントした追加オペレーションをクライアントに提供します。ここでクライアントはわかっている（IDL に記述されている）すべてのオペレーションにアクセスでき，さらに，ほかのオペレーションをサポートする IR を検証し，それを呼び出せます。サンプルでは，IR 定義オブジェクトの管理方法と，C++および Java の IR を使用したリモートオブジェクトの検査の方法を示します。

このプログラムをテストするには，次の条件が成立している必要があります。

- osagent が起動され，実行中である
- IR が irep を使用して起動されている
- IR 起動時のコマンドラインによって，または idl2ir を使用して，IR に IDL ファイルがロードされている
- クライアントプログラムが起動する

コードサンプル 16-3　IR 内のインタフェースのオペレーションと属性の検索（C++）

```
/* PrintIR.C */
#ifndef _VIS_INCLUDE_IR
#define _VIS_INCLUDE_IR
#endif

#include "corba.h"
#include "strvar.h"

int main(int argc, char *argv[ ]) {
   try {
      if (argc != 2) {
      cout << "Usage: PrintIR idlName" << endl;
      exit(1);
      }
      CORBA::String_var idlName = (const char *)argv[1];

      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      CORBA::Repository_var rep = CORBA::Repository::_bind();

      CORBA::Contained_var contained = rep->lookup(idlName);
      CORBA::InterfaceDef_var intDef =
         CORBA::InterfaceDef::_narrow(contained);
      if (intDef != CORBA::InterfaceDef::_nil()) {
      CORBA::InterfaceDef::FullInterfaceDescription_var fullDesc =
            intDef->describe_interface();
      cout << "Operations:" << endl;
      for(CORBA::ULong i = 0;
            i < fullDesc->operations.length(); i++)
         cout << " " << fullDesc->operations[i].name << endl;
         cout << "Attributes:" << endl;
         for(i = 0; i < fullDesc->attributes.length(); i++)
            cout << " " << fullDesc->attributes[i].name
                  << endl;
      }else
         cout << "idlName is not an interface: "
               << idlName << endl;

   } catch (const CORBA::Exception& excep) {
      cerr << "Exception occurred ..." << endl;
      cerr << excep << endl;
      exit(1);
```

```
    }
    return 0;
}
```

コードサンプル 16-4　IR 内のインタフェースのオペレーションと属性の検索（Java）

```
// Client.java
import org.omg.CORBA.InterfaceDef;
import org.omg.CORBA.InterfaceDefHelper;
import org.omg.CORBA.Request;
import java.util.Random;

public class Client {
   public static void main(String[ ] args) {
      try {
         // Initialize the ORB.
         org.omg.CORBA.ORB orb =
                        org.omg.CORBA.ORB.init(args,null);
         // Get the manager Id
         byte[ ] managerId = "BankManager".getBytes();
         // Locate an account manager. Give the full POA name
         // and the servant ID.
         Bank.AccountManager manager =
            Bank.AccountManagerHelper.bind(orb,
                               "/bank_ir_poa", managerId);
         // use args[0] as the account name, or a default.
         String name = args.length > 0 ?
                                args[0] : "Jack B. Quick";
         // Request the account manager to open a named
         // account.
         Bank.Account account = manager.open(name);
         // Get the balance of the account.
         float balance = account.balance();
         // Print out the balance.
         System.out.println("The balance in " + name +
                             "'s account is $" + balance);
         // Calculate and set a new balance
         balance = args.length > 1 ?
            Float.parseFloat(args[1]):
            Math.abs(new Random().nextInt()) % 100000 / 100f;
         account.balance(balance);
         // Get the balance description if it is possible
         // and print it
         String desc = getDescription(account);
         System.out.println("Balance description:¥n" + desc);
      }catch (org.omg.CORBA.SystemException e) {
         System.err.println("System exception caught:" + e);
      }catch (Exception e) {
         System.err.println("Unexpected exception caught:");
         e.printStackTrace();
      }
   }

   static String getDescription(Bank.Account account) {
      // Get the interface repository definition for
      // this interface
      InterfaceDef accountDef = InterfaceDefHelper.
         narrow(account._get_interface_def());
      // Check if this *particular* implementation supports
      // "describe" operation
      if (accountDef.lookup("describe") != null) {
         // We cannot use the static skeleton's method here
         // because at the time of its creation this method
         // was not present in the IDL's version of the
         // Account interface. Use DII instead.
         Request request = account._request("describe");
         request.result().value().insert_string("");
         request.invoke();
         return request.result().value().extract_string();
      }else {
         return "<no description>";
      }
   }
}
```

17 動的起動インタフェースの使用

この章では，DII（動的起動インタフェース）の使用方法について説明します。ほとんどのクライアントプログラムの開発者は，自分の CORBA オブジェクトの型を知っていて，それらの型の IDL コンパイラが生成したスタブをコードの中に組み込みます。これに対し, 汎用的なクライアントを作成しようとする開発者は, どの種類のオブジェクトを起動したいのかが事前にわかりません。このような場合，開発者は DII を使用して，ランタイム時に入手した情報から任意の CORBA オブジェクトの任意のメソッドを起動できるクライアントを作成します。

17.1　動的起動インタフェースとは

DII（動的起動インタフェース）を使用すると，クライアントプログラムから，そのクライアントの作成時点では型がわからない CORBA オブジェクトに対してメソッドを起動できます。DII はデフォルトの静的起動とは対照的です。静的起動ではクライアントソースコードに，そのクライアントから起動する CORBA オブジェクトの型ごとに一つずつコンパイラが生成したスタブを組み込む必要があります。つまり，静的起動を使用するクライアントは，起動するオブジェクトの型を事前に宣言します。DII を使用するクライアントは，どのような種類のオブジェクトが起動されるかがプログラマにもわからないので，そのような宣言をしません。DII の利点は柔軟性です。DII を使用すると，クライアントをコンパイルした時点ではインタフェースが存在しなかったオブジェクトも含め，任意のオブジェクトを起動できる汎用的なクライアントを作成できます。DII の短所は次の 2 点です。

- プログラミングが難しい（実質的に，作成するコードはスタブの機能を果たさなければなりません）
- ランタイムに多くの作業が行われるので起動に時間が掛かる

DII は純粋にクライアントインタフェースであり，静的起動と動的起動はオブジェクトインプリメンテーションの観点から見れば同じものです。

DII を使用すると，次のようなクライアントを作成できます。

- スクリプト環境と CORBA オブジェクトの間のブリッジまたはアダプタ。例えば，スクリプトはブリッジを呼び出し，オブジェクトの識別子，メソッドの識別子，およびパラメタ値を引き渡します。ブリッジは動的リクエストを構築して発行し，結果を受け取り，それをスクリプト環境に返します。そのようなブリッジでは，スクリプト環境がどのような種類のオブジェクトを起動したいかが事前に開発者にわからないので，静的起動を使用できません。
- 汎用的なオブジェクトテスタ。例えば，あるクライアントが任意のオブジェクトの識別子を使用し，そのインタフェースを IR（インタフェースリポジトリ）から検索し（「16.　インタフェースリポジトリの使用」参照），個々のメソッドを人為的な引数値を使用して起動するとします。このような汎用的なテスタも，静的起動では作成できません。

注

クライアントは，DII リクエストの中で有効な引数を引き渡さなければなりません。それに失敗すると，サーバのクラッシュも含め，予期できない結果が生じるおそれがあります。IR を使用してパラメタ値の型を動的にチェックすることもできますが，パフォーマンスが低下します。最も効果的な方法は，DII を使用するクライアントを起動するコード（例えば，スクリプト）の信頼性を高め，確実に有効な引数が引き渡されるようにすることです。

17.1.1　DII の主要な概念

動的起動インタフェースを実装しなければいけないオブジェクトは，CORBA オブジェクト全体から見れば実際には少数でしょう。また DII は，多くの場合，一つのタスクを実行するのに複数の方法を提供し，プログラミングの単純性を取るか特殊状況での性能を取るかがその選択基準になります。その結果，DII は理解するのがより難しい CORBA 機能の一つとなります。ここでは，主要な概念を簡単に説明します。コードサンプルも交えての詳細な説明は，以降の節で説明します。

DII を使用するには，最も一般的なことから始めるとして，次の概念を理解しておく必要があります。

- Request オブジェクト
- Any オブジェクトと Typecode オブジェクト
- リクエスト送信オプション

- 応答受信オプション

(1)　Request オブジェクトを使用する

一つの Request オブジェクトは，一つの CORBA オブジェクトの一つのメソッドの一回の起動を示します。同じ CORBA オブジェクトに対して二つのメソッドを起動したい場合，または二つの異なるオブジェクトに対して同じメソッドを起動したい場合は，二つの Request オブジェクトが必要です。メソッドを起動するには，まず，CORBA オブジェクトを表すオブジェクトリファレンス，つまりターゲットのリファレンスが必要です。ターゲットのリファレンスを使用して Request を生成し，それに引数を取り込み，Request を送信し，応答を待ち，Request からの結果を取得します。

Request を生成するには，二つの方法があります。このうちの単純な方法は，ターゲットオブジェクトの _request メソッドを起動することで，このメソッドはすべての CORBA オブジェクトが継承します。実際には，これはターゲットオブジェクトを起動しません。_request には，Request の中で起動したいメソッドの IDL インタフェース名，例えば get_balance などを引き渡します。_request で生成される Request に引数値を追加するには，起動するメソッドに必要な引数ごとに Request の add_value メソッドを起動します。ターゲットに一つ以上の Context オブジェクトを引き渡すには，Request の ctx メソッドを使用してそれらのオブジェクトを Request に追加します。

直感的には気づきませんが，Request の結果の型を Request の result メソッドで指定することも必要です。性能上の理由から，VisiBroker ORB 間で交換されるメッセージには型情報が入っていません。Request 内でプレースホルダ結果型を指定することで，ターゲットオブジェクトが送信する応答メッセージから結果を正しく抽出するために必要な情報を VisiBroker ORB に与えます。同様に，起動するメソッドがユーザ例外を発生させる可能性がある場合は，Request を送信する前にプレースホルダ例外を Request に追加しておかなければなりません。

Request オブジェクトを生成する複雑な方法は，ターゲットオブジェクトの_create_request メソッドを起動することで，このメソッドもすべての CORBA オブジェクトが継承します。このメソッドは幾つかの引数を取り，それらの引数が新しい Request に引数を取り込み，その Request が返す結果とユーザ例外の型があれば，それを指定します。_create_request メソッドを使用するには，このメソッドが引数として取るコンポーネントを事前に作成しておく必要があります。_create_request メソッドを使用すれば性能上の利点が見込まれます。複数のターゲットオブジェクトに対して同じメソッドを起動する場合，複数の _create_request 呼び出しに引数コンポーネントを再利用できます。

注

_create_request メソッドには多重定義された二つの形態があります。一つは ContextList パラメタと ExceptionList パラメタを含み，もう一つはそれらを含んでいません。呼び出しの中で一つ以上の Context オブジェクトを引き渡したい場合や，起動したいメソッドが一つ以上のユーザ例外を発生させる可能性がある場合は，追加パラメタがある_create_request メソッドを使用しなければなりません。

(2)　引数を Any 型でカプセル化する

ターゲットメソッドの引数，結果，および例外は，Any と呼ばれる特殊オブジェクトの中でそれぞれ指定されます。Any は，任意の型の引数をカプセル化する汎用的なオブジェクトです。Any は IDL で記述できるすべての型を保持できます。Request への引数を Any として指定すると，Request に任意の引数型と値を保持させることができ，コンパイラで型の不一致も起きません（同じことが結果と例外にも当てはまります）。

Any は TypeCode と value で構成されます。value は単なる値であり，TypeCode は値の中のビット列をどのように解釈するか（つまり，値の型）を記述したオブジェクトです。long や Object など，単純な IDL 型用の単純な TypeCode 定数は，idl2cpp コンパイラまたは idl2java コンパイラによって生成されるヘッダファイルに組み込まれます。struct，union，typedef など，IDL 構造体の TypeCode は，作成す

る必要があります。そのような TypeCode は，記述する型が再帰的であってもかまいません。long と string から成る struct を考えてみてください。この struct の TypeCode には，long 用の TypeCode と string 用の TypeCode が含まれます。idl2cpp コンパイラは，-type_code_info オプションを指定して起動されると，IDL ファイル内に定義された型用に TypeCode を生成します。ただし，DII を使用している場合，ランタイム時に TypeCode を取得する必要があります。ランタイム時に TypeCode を IR から取得できます（「16. インタフェースリポジトリの使用」参照）。または，ORB::create_struct_tc または ORB::create_exception_tc を起動することによって，VisiBroker ORB に TypeCode を生成させ，TypeCode を取得できます。

_create_request メソッドを使用する場合は，Any にカプセル化したターゲットメソッド引数を NVList という別の特殊なオブジェクトに挿入する必要があります。Request の生成方法に関係なく，Request の結果は NVList としてエンコードされます。ここで引数に関して述べた内容はすべて結果にも当てはまります。NV は名前付きの値を意味し，NVList は項目数と項目番号で構成され，各項目は，名前と値とフラグをそれぞれ一つずつ備えています。名前は引数名であり，値は Any にカプセル化された引数であり，フラグはその引数の IDL モード（例えば，in か out）を示します。

Request の結果は，一つの名前付きの値として表されます。

(3)　リクエストを送信するオプション

Request を生成し，それに引数，結果の型，例外の型を取り込んだあと，その Request をターゲットオブジェクトへ送信します。Request を送信するには，次のような複数の方法があります。

- 最も単純な方法は，Request の invoke メソッドを呼び出すことです。このメソッドは応答メッセージを受信するまで待ちます。
- それより複雑で応答を待たない方法は，Request の send_deferred メソッドです。これは，並列処理にスレッドを使用することに代わる方法です。多くのオペレーティングシステムで，send_deferred メソッドはスレッドを生成するより効率的です。
- send_deferred メソッドを使用する目的が複数のターゲットオブジェクトを並行して起動することなら，代わりに VisiBroker ORB オブジェクトの send_multiple_requests_deferred メソッドを使用できます。このメソッドは Request オブジェクトのシーケンスを取ります。
- ターゲットメソッドが IDL で oneway として定義されている場合，Request の send_oneway メソッドを使用してください。
- VisiBroker ORB の send_multiple_requests_oneway メソッドを使用して，複数の oneway メソッドを並行して起動できます。

(4)　応答を受信するオプション

Request の invoke メソッド呼び出しによってその Request を送信する場合，結果を取得する方法は一つしかありません。つまり，Request オブジェクトの env メソッドを使用して例外の有無をチェックし，例外がなければ，Request の result メソッドを使用して Request から NamedValue を抽出します。send_oneway メソッドを使用した場合，結果はありません。send_deferred メソッドを使用した場合，Request の poll_response メソッドを呼び出すことによって，処理が完了したかどうかを定期的に検査できます。poll_response メソッドは，応答を受信したかどうかを示すコードを返します。しばらくポーリングしたあと，遅延送信の完了を待ち続ける場合は，Request の get_response メソッドを使用します。

send_multiple_requests_deferred メソッドを使用して複数の Request を送信した場合は，該当する Request の get_response メソッドを起動することによって，その Request が完了したかどうかがわかります。Request が完了するまで待つには，VisiBroker ORB の get_next_response メソッドを使用します。待ち続けたくない場合は，VisiBroker ORB の poll_next_response メソッドを使用します。

17.1.2　オブジェクトのオペレーションを動的に起動する手順

DII を使用する場合にクライアントが実行する手順を次に示します。

1. C++の場合，タイプコードが IDL インタフェースとタイプ用に生成されるように，必ず idl コンパイラに-type_code_info オプションが渡されるようにしてください。idl2cpp ツールの完全な説明については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「idl2cpp」の記述を参照してください。
2. 使用したいターゲットオブジェクトの汎用的なリファレンスを取得します。
3. ターゲットオブジェクト用の Request オブジェクトを生成します。
4. request パラメタと返したい結果を初期化します。
5. リクエストを起動し，結果を待ちます。
6. 結果を抽出します。

17.1.3　DII を使用したサンプルプログラムの格納場所

DII の使用方法を示したサンプルプログラムが Borland Enterprise Server VisiBroker をインストールしたディレクトリの examples/vbe/basic/bank_dynamic に入っています。この章では，これらのサンプルプログラムを使って，DII の概念を説明します。C++の場合，これらのサンプルプログラムを VIS_INCLUDE_IR フラグでコンパイルし，タイプコード生成オプションを追加してください。

17.1.4　idl2java コンパイラの使用（Java）

idl2java コンパイラにはフラグ（-dynamic_marshal）があり，このフラグがオンになると DII を使用してスタブコードを生成します。DII の任意の型でこれを行うには，IDL ファイルを作成し，-dynamic_marshal でスタブコードを生成し，調べます。

17.2　汎用的なオブジェクトリファレンスを取得

DII を使用する場合，クライアントプログラムで従来のバインド方法を使用してターゲットオブジェクトのリファレンスを取得する必要はありません。コンパイル時に，ターゲットオブジェクトのクラス定義をクライアントがわからない場合があるためです。

コードサンプル 17-1 に，VisiBroker ORB オブジェクトが提供する bind メソッドをクライアントプログラムが使用して，オブジェクト名を指定することによってオブジェクトにバインドする方法を示します。このメソッドは汎用 CORBA::Object（C++）を返します。

コードサンプル 17-1　汎用的なオブジェクトリファレンスを取得する（C++）

```
. . .
CORBA::Object_var account;
try {
   // initialize the ORB.
   CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
} catch (const CORBA::Exception& e)
   cout << "Failure during ORB_init " << endl;
   cout << e << endl;
}
. . .
try {
   // Request ORB to bind to object supporting the
   // account interface.
   account = orb->bind("IDL:Account:1.0");
} catch (const CORBA::Exception& excep)
   cout << "Error binding to account" << endl;
   cout << excep << endl;
}
cout << "Bound to account object " << endl;
. . .
```

コードサンプル 17-2 に，VisiBroker ORB オブジェクトが提供する bind メソッドをクライアントプログラムが使用して，オブジェクト名を指定することによってオブジェクトにバインドする方法を示します。このメソッドは汎用 org.omg.CORBA.Object（Java）を返します。

コードサンプル 17-2　汎用的なオブジェクトリファレンスを取得する（Java）

```
. . .
org.omg.CORBA.Object account;
try {
   // initialize the ORB.
   com.inprise.vbroker.CORBA.ORB orb =
                                    (com.inprise.vbroker.CORBA.ORB)
                                    org.omg.CORBA.ORB.init(args, null);
}catch(Exception e) {
   System.err.println ("Failure during ORB_init");
   e.printStackTrace();
}
. . .

try {
   // Request ORB to bind to the object supporting
   // the account interface.
   account = orb.bind("IDL:Account:1.0",
                      "BankManager", null, null);
}catch(Exception excep) {
   System.err.println ("Error binding to account" );
   excep.printStackTrace();
}
System.out.println ("Bound to account object");
. . .
```

17.3　Request を生成し初期化

クライアントプログラムがオブジェクトのメソッドを起動した場合，メソッドの起動を表す Request オブジェクトが生成されます。この Request オブジェクトはバッファに書き込まれ（つまりマーシャルされ），オブジェクトインプリメンテーションへ送信されます。クライアントプログラムがクライアントスタブを使用する場合に，この処理は透過的に行われます。DII を使用するクライアントプログラムは，自分自身で Request オブジェクトを生成し，送信する必要があります。

注

このクラスにはコンストラクタがありません。Request オブジェクトの生成には，Object の_request メソッドまたは Object の_create_request メソッドを使用します。

17.3.1　Request クラス（C++）

コードサンプル 17-3 に，Request クラスを示します。リクエストの target は，Request の生成に使用したオブジェクトリファレンスから暗黙的に設定されます。operation 名は Request の生成時に指定しなければなりません。

コードサンプル 17-3　Request クラス（C++）

```
class Request {
   public:
      CORBA::Object_ptr target() const;
      const char* operation() const;
      CORBA::NVList_ptr arguments();
      CORBA::NamedValue_ptr result();
      CORBA::Environment_ptr env();
      void ctx(CORBA::Context_ptr ctx);
      CORBA::Context_ptr ctx() const;
      CORBA::Status invoke();
      CORBA::Status send_oneway();
      CORBA::Status send_deferred();
      CORBA::Status get_response();
      CORBA::Status poll_response();
      . . .
   };
};
```

17.3.2　Request インタフェース（Java）

コードサンプル 17-4 に，Request インタフェースを示します。リクエストの target は，Request の生成に使用したオブジェクトリファレンスから暗黙的に設定されます。operation 名は Request の生成時に指定しなければなりません。

コードサンプル 17-4　Request インタフェース（Java）

```
package org.omg.CORBA;
public abstract class Request {
   public abstract org.omg.CORBA.Object target();
   public abstract java.lang.String operation();
   public abstract org.omg.CORBA.NVList arguments();
   public abstract org.omg.CORBA.NamedValue result();
   public abstract org.omg.CORBA.Environment env();
   public abstract org.omg.CORBA.ExceptionList exceptions();
   public abstract org.omg.CORBA.ContextList contexts();
   public abstract void ctx(org.omg.CORBA.Context ctx);
   public abstract org.omg.CORBA.Context ctx();
   public abstract org.omg.CORBA.Any add_in_arg();
   public abstract org.omg.CORBA.Any add_named_in_arg();
   public abstract org.omg.CORBA.Any add_inout_arg();
   public abstract org.omg.CORBA.Any add_named_inout_arg();
   public abstract org.omg.CORBA.Any add_out_arg();
```

```
    public abstract org.omg.CORBA.Any add_named_out_arg();
    public abstract void set_return_type();
    public abstract org.omg.CORBA.Any return_value();
    public abstract void invoke();
    public abstract void send_oneway();
    public abstract void send_deferred();
    public abstract void get_response();
    public abstract boolean poll_response();
}
```

17.3.3　DII リクエストを生成し初期化する方法

オブジェクトへのバインドを発行し，オブジェクトリファレンスを取得したあと，Request オブジェクトを生成するために二つの方法のどちらかを使用できます。コードサンプル 17-5 に CORBA::Object クラスが提供するメソッドを示します。コードサンプル 17-6 に org.omg.CORBA.Object インタフェースが提供するメソッドを示します。

コードサンプル 17-5　Request オブジェクトを生成する三つのメソッド（C++）

```
class Object {
   . . .
   CORBA::Request_ptr _request(Identifier operation);
   CORBA::Status _create_request(
      CORBA::Context_ptr ctx,
      const char *operation,
      CORBA::NVList_ptr arg_list,
      CORBA::NamedValue_ptr result,
      CORBA::Request_ptr request,
      CORBA::Flags req_flags);
   CORBA::Status _create_request(
      CORBA::Context_ptr ctx,
      const char *operation,
      CORBA::NVList_ptr arg_list,
      CORBA::NamedValue_ptr result,
      CORBA::ExceptionList_ptr eList,
      CORBA::ContextList_ptr ctxList,
      CORBA::Request_out request,
      CORBA::Flags req_flags);
    . . .
};
```

コードサンプル 17-6　Request オブジェクトを生成する三つのメソッド（Java）

```
package org.omg.CORBA;
public interface Object {
. . .
   public org.omg.CORBA.Request _request(
                          java.lang.String operation);

   public org.omg.CORBA.Request _create_request(
      org.omg.CORBA.Context ctx,
      java.lang.String operation,
      org.omg.CORBA.NVList arg_list,
      org.omg.CORBA.NamedValue result
   );

   public org.omg.CORBA.Request _create_request(
      org.omg.CORBA.Context ctx,
      java.lang.String operation,
      org.omg.CORBA.NVList arg_list,
      org.omg.CORBA.NamedValue result,
      org.omg.CORBA.ExceptionList exceptions,
      org.omg.CORBA.ContextList contexts
   );
. . .
}
```

17.3.4　_create_request メソッドを使用

_create_request メソッドを使用して Request オブジェクトを生成し，Context，オペレーション名，引き渡したい引数リスト，および結果を初期化できます。

オプションとして，リクエストの ContextList を設定することもできます。ContextList は，リクエストの IDL で定義された属性に対応するものです。request パラメタは，このオペレーション用に生成された Request オブジェクトをポイントします。

17.3.5　_request メソッドを使用

コードサンプル 17-7 に，_request メソッドを使用して，オペレーション名だけを指定して Request オブジェクトを生成する方法を示します。float リクエストの生成後，float リクエストの add_in_arg メソッドを起動すると，入力パラメタアカウント名を追加して，その結果型が set_return_type メソッドの呼び出しによるオブジェクトリファレンス型となるよう初期化されます。呼び出し後に，メソッド result に対する結果の呼び出しによってリターン値が抽出されます。アカウントマネージャインスタンスで別のメソッドを起動するには，同じ手順が繰り返されます。その場合，in パラメタとリターン型だけが異なります。

req の Any オブジェクトは，希望のアカウント name によって初期化され，入力引数として request の引数リストに加えられます。リクエストの初期化の最後の手順は，float を受信するために result 値を設定することです。

17.3.6　Request オブジェクトの生成例

Request オブジェクトはオペレーション，引数，および結果に対応するすべてのメモリの所有権を保持し，プログラマがそれらの項目を解放しないようにします。コードサンプル 17-7 に Request オブジェクトの生成例（C++），コードサンプル 17-8 に Request オブジェクトの生成例（Java）を示します。

コードサンプル 17-7　Request オブジェクトを生成する（C++）

```
. . .
CORBA::NamedValue_ptr result;
CORBA::Any_ptr resultAny;
CORBA::Request_var req;
CORBA::Any customer;

. . .
try {
   req = account->_request("balance");

   // Create argument to request
   customer <<= (const char *) name;
   CORBA::NVList_ptr arguments = req->arguments();
   arguments->add_value("customer", customer, CORBA::ARG_IN);

   // Set result
   result = req->result();
   resultAny = result->value();
   resultAny->replace(CORBA::_tc_float, &result);

} catch(CORBA::Exception& excep) {
. . .
```

コードサンプル 17-8　Request オブジェクトを生成する（Java）

```
// Client.java
public class Client {
   public static void main(String[ ] args) {
      if (args.length ! = 2) {
         System.out.println(
          "Usage: vbj Client <manager-name> <account-name>¥n");
         return;
```

```
        }
        String managerName = args[0];
        String accountName = args[1];
        org.omg.CORBA.Object accountManager, account;
        org.omg.CORBA.ORB orb =
           org.omg.CORBA.ORB.init(args, null);
        accountManager =
           orb.bind("IDL:Bank/AccountManager:1.0",
                    managerName, null, null);
        org.omg.CORBA.Request request =
           accountManager._request("open");
        request.add_in_arg().insert_string(accountName);
        request.set_return_type(orb.get_primitive_tc(
           org.omg.CORBA.TCKind.tk_objref)
        );
        request.invoke();
        account = request.result().value().extract_Object();
        org.omg.CORBA.Request request =
                                  account._request("balance");
        request.set_return_type(orb.get_primitive_tc(
           org.omg.CORBA.TCKind.tk_float)
        );
        request.invoke();
        float balance =
            request.result().value().extract_float();
        System.out.println("The balance in " + accountName +
           "'s account is $" + balance);
    }
}
```

17.3.7　リクエストのコンテキストを設定（C++）

サンプルプログラムでは使用されていませんが，Request の一部としてオブジェクトインプリメンテーションに渡され，NamedValue オブジェクトとして格納されるプロパティのリストを含めるために Context オブジェクトを使用できます。これらのプロパティは，オブジェクトインプリメンテーションと通信する暗黙的な情報を表します。

コードサンプル 17-9　Context クラス

```
class Context {
   public:
      const char *context_name() const;
      CORBA::Context_ptr parent();
      CORBA::Status create_child(
            const char *name, CORBA::Context_ptr&);
      CORBA::Status set_one_value(
            const char *name, const CORBA::Any&);
      CORBA::Status set_values(CORBA::NVList_ptr);
      CORBA::Status delete_values(const char *name);
      CORBA::Status get_values(
            const char *start_scope,
            CORBA::Flags,
            const char *name,
            CORBA::NVList_ptr&) const;
};
```

17.3.8　リクエストの引数を設定

Request の引数は NVList オブジェクトで表現されます。このオブジェクトは名前・値のペアを NamedValue オブジェクトとして格納します。このリストのポインタを取得するには，arguments メソッドを使用します。そのあと，このポインタを使用して個々の引数の名前と値を設定できます。

注

Request を送信する前に，必ず引数を初期化してください。そうしないと，マーシャルエラーが発生し，サーバの処理が中断されることがあります。

(1)　NVList を使用して引数のリストをインプリメントする

このクラスは，メソッド起動用の引数を表す NamedValue オブジェクトのリストをインプリメントします。リスト内のオブジェクトの追加，削除，および照会を行うメソッドが提供されます。

コードサンプル 17-10　NVList クラス（C++）

```
class NVList {
   public:
      . . .
      CORBA::Long count() const;
      CORBA::NamedValue_ptr add(CORBA::Flags flags);
      CORBA::NamedValue_ptr add_item(
         const char *name, CORBA::Flags flags);
      CORBA::NamedValue_ptr add_value(
         const char *name,
         const CORBA::Any *any,
         CORBA::Flags flags);

   CORBA::NamedValue_ptr add_item_consume(
         char *name, CORBA::Flags flags);
   CORBA::NamedValue_ptr add_value_consume(
         char *name,
         CORBA::Any *any,
         CORBA::Flags flags);
   CORBA::NamedValue_ptr item(CORBA::Long index);
   CORBA::Status remove(CORBA::Long index);
   . . .
};
```

コードサンプル 17-11　NVList クラス（Java）

```
package org.omg.CORBA;

public abstract class NVList {
   public int count();
   public org.omg.CORBA.NamedValue add(int flags);
   public org.omg.CORBA.NamedValue add_item(
                              java.lang.String name, int flags);
   public org.omg.CORBA.NamedValue add_value(
      java.lang.String name,
      org.omg.CORBA.Any value,
      int flags
   );
   public org.omg.CORBA.NamedValue item(int index);
   public void remove(int index);
}
```

(2)　NamedValue クラスを使用して入出力引数を設定する

このクラスは，メソッド起動リクエストの入出力両用の引数を表す名前・値のペアをインプリメントします。NamedValue クラスは，クライアントプログラムへ返すリクエストの結果を表すために使用することもできます。name プロパティは単なる文字列であり，value プロパティは Any クラスによって表されます。コードサンプル 17-12 に NamedValue クラスの例（C++），およびコードサンプル 17-13 に NamedValue クラスの例（Java）を示します。

注

Java の場合，このクラスにコンストラクタはありません。NamedValue オブジェクトのリファレンスを取得するには，ORB.create_named_value メソッドを使用します。

コードサンプル 17-12　NamedValue クラス（C++）

```
class NamedValue{
   public:
      const char *name() const;
      CORBA::Any *value() const;
      CORBA::Flags flags() const;
};
```

コードサンプル 17-13　NamedValue インタフェース（Java）

```
package org.omg.CORBA;
public abstract class NamedValue {
   public java.lang.String name();
   public org.omg.CORBA.Any value();
   public int flags();
}
```

表 17-1 は，NamedValue クラスのメソッドの説明です。

表 17-1　NamedValue のメソッド

メソッド	説明
name	項目名のポインタを返します。このポインタを使用して名前を初期化できます。
value	項目の値を表す Any オブジェクトのポインタを返します。このポインタを使用して値を初期化できます。詳細については，「17.3.9　Any クラスを使用して型を保護した状態で引き渡す」を参照してください。
flags	この項目が入力引数，出力引数，入出力両用の引数のどれであるかを示します。項目が入出力両用の引数である場合は，VisiBroker ORB が引数のコピーを作成して呼び出し側のメモリをそのまま残すように指示するフラグを指定できます。次のフラグがあります。 • ARG_IN • ARG_OUT • ARG_INOUT

17.3.9　Any クラスを使用して型を保護した状態で引き渡す

このクラスは，IDL 指定型を保持し，タイプセーフ方式で引き渡せるようにするために使用します。

(1)　C++の場合

このクラスのオブジェクトは，含まれているオブジェクトの型を定義する TypeCode のポインタと，その含まれたオブジェクトのポインタを持ちます。オブジェクトの値と型を初期化し照会するメソッドだけでなく，オブジェクトの構築，コピー，および解放を行うメソッドも提供されます。さらに，オブジェクトをストリームに書き込んだり，ストリームから読み取ったりするストリームメソッドも提供されます。コードサンプル 17-14 に定義例を示します。

コードサンプル 17-14　Any クラス（C++）

```
class Any {
   public:
      . . .
      CORBA_TypeCode_ptr type();
      void type(CORBA_TypeCode_ptr tc);
      const void *value() const;
      static CORBA::Any_ptr _nil();
      static CORBA::Any_ptr _duplicate(CORBA::Any *ptr);
      static void _release(CORBA::Any *ptr);
      . . .
}
```

(2)　Java の場合

このクラスのオブジェクトは，含まれているオブジェクトの型を定義する TypeCode のリファレンスと，その含まれたオブジェクトのリファレンスを持ちます。オブジェクトの値と型を初期化し照会するメソッドだけでなく，オブジェクトの構築，コピー，および解放を行うメソッドも提供されます。さらに，オブ

ジェクトをストリームに書き込んだり，ストリームから読み取ったりするストリームメソッドも提供されます。コードサンプル 17-15 に定義例を示します。

コードサンプル 17-15　Any クラス（Java）

```
package org.omg.CORBA;
public abstract class Any {
   public abstract TypeCode type();
   public abstract void type(TypeCode type);
   public abstract void read_value(InputStream input,
                                   TypeCode type);
   public abstract void write_value(OutputStream output);
   public abstract boolean equal(Any rhs);
   . . .
}
```

17.3.10　TypeCode クラスを使用して引数または属性の型を表す

このクラスは，IR と IDL コンパイラが引数または属性の型を表すために使用します。Request オブジェクトの中では，引数の型を指定する場合に，Any クラスとともに TypeCode オブジェクトも使用します。

C++の場合，TypeCode オブジェクトは，kind とパラメタリストプロパティを持っています。コードサンプル 17-16 に TypeCode クラスの例（C++）を示します。

Java の場合，TypeCode オブジェクトは，kind とパラメタリストプロパティを持っており，TCKind クラスで定義した値のどれかで表されます。コードサンプル 17-17 に TypeCode クラスの例（Java）を示します。

注

Java の場合，このクラスにはコンストラクタはありません。ORB.get_primitive_tc メソッドか，または ORB.create_*_tc メソッドの一つを使用して，TypeCode オブジェクトを作成してください。詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「ORB」の記述を参照してください。

表 17-2 に，TypeCode オブジェクトの種類とパラメタを示します。

表 17-2　TypeCode の種類とパラメタ

種類	パラメタリスト
tk_abstract_interface	repository_id，interface_name
tk_alias	repository_id，alias_name，TypeCode
tk_any	なし
tk_array	length，TypeCode
tk_boolean	なし
tk_char	なし
tk_double	なし
tk_enum	repository_id，enum-name，enum-id^{1}，enum-id^{2}，…enum-id^{n}
tk_except	repository_id，exception_name，StructMembers
tk_fixed	digits，scale
tk_float	なし

種類	パラメタリスト
tk_long	なし
tk_longdouble	なし
tk_longlong	なし
tk_native	id，name
tk_null	なし
tk_objref	repository_id，interface_id
tk_octet	なし
tk_Principal	なし
tk_sequence	TypeCode，maxlen
tk_short	なし
tk_string	maxlen-integer
tk_struct	repository_id，struct-name，{member[1]，TypeCode[1]}，{member[n]，TypeCode[n]}
tk_TypeCode	なし
tk_ulong	なし
tk_ulonglong	なし
tk_union	repository_id，union-name，switch TypeCode，{label-value[1]，member-name[1]，TypeCode[1]}，{label-value[n]，member-name[n]，TypeCode[n]}
tk_ushort	なし
tk_value	repository_id，value_name，boxType
tk_value_box	repository_id，value_name，typeModifier，concreteBase，members
tk_void	なし
tk_wchar	なし
tk_wstring	なし

コードサンプル 17-16　TypeCode クラス（C++）

```
class _VISEXPORT CORBA_TypeCode {
   public:
      . . .
      // For all CORBA_TypeCode kinds
      CORBA::Boolean equal(CORBA_TypeCode_ptr tc) const;
      CORBA::Boolean equivalent(CORBA_TypeCode_ptr tc) const;
      CORBA_TypeCode_ptr get_compact_typecode() const;
      CORBA::TCKind kind() const // . . .
      // For tk_objref, tk_struct, tk_union,
            tk_enum, tk_alias and tk_except
      virtual const char* id() const; // raises(BadKind);
      virtual const char *name() const; // raises(BadKind);
      // For tk_struct, tk_union, tk_enum and tk_except
      virtual CORBA::ULong member_count() const;
            // raises((BadKind));
      virtual const char *member_name(
            CORBA::ULong index) const;
```

```
            // raises((BadKind, Bounds));
        // For tk_struct, tk_union and tk_except
        virtual CORBA_TypeCode_ptr member_type(
            CORBA::ULong index) const;
            // raises((BadKind, Bounds));
        // For tk_union
        virtual CORBA::Any_ptr member_label(
            CORBA::ULong index) const;
            // raises((BadKind, Bounds));
        virtual CORBA_TypeCode_ptr discriminator_type() const;
            // raises((BadKind));
        virtual CORBA::Long default_index() const;
            // raises((BadKind));
        // For tk_string, tk_sequence and tk_array
        virtual CORBA::ULong length() const; // raises(
            (BadKind));
        // For tk_sequence, tk_array and tk_alias
        virtual CORBA_TypeCode_ptr content_type() const;
            // raises((BadKind));
        // For tk_fixed
        virtual CORBA::UShort fixed_digits() const;
            // raises (BadKind)
        virtual CORBA::Short fixed_scale() const;
            // raises (BadKind)

         // for tk_value
        virtual CORBA::Visibility member_visibility(
            CORBA::ULong index)const;
            //raises(BadKind, Bounds);
        virtual CORBA::ValueModifier type_modifier() const;
            // raises(BadKind);
        virtual CORBA::TypeCode_ptr concrete_base_type() const;
            // raises(BadKind);
};
```

コードサンプル 17-17　TypeCode インタフェース（Java）

```
public abstract class TypeCode extends java.lang.Object
      implements org.omg.CORBA.portable.IDLEntity {
   public abstract boolean equal(org.omg.CORBA.TypeCode tc);
   public boolean equivalent(org.omg.CORBA.TypeCode tc);
   public abstract org.omg.CORBA.TCKind kind();
   public TypeCode get_compact_typecode();
   public abstract java.lang.String id()
      throws org.omg.CORBA.TypeCodePackage.BadKind;
   public abstract java.lang.String name()
      throws org.omg.CORBA.TypeCodePackage.BadKind;
   public abstract int member_count()
      throws org.omg.CORBA.TypeCodePackage.BadKind;
   public abstract java.lang.String member_name(int index)
      throws org.omg.CORBA.TypeCodePackage.BadKind,
         org.omg.CORBA.TypeCodePackage.Bounds;
   public abstract org.omg.CORBA.TypeCode member_type(
                                                  int index)
      throws org.omg.CORBA.TypeCodePackage.BadKind,
         org.omg.CORBA.TypeCodePackage.Bounds;

   public abstract org.omg.CORBA.Any member_label(int index)
      throws org.omg.CORBA.TypeCodePackage.BadKind,
         org.omg.CORBA.TypeCodePackage.Bounds;
   public abstract org.omg.CORBA.TypeCode
                                     discriminator_type()
      throws org.omg.CORBA.TypeCodePackage.BadKind;
   public abstract int default_index()
      throws org.omg.CORBA.TypeCodePackage.BadKind;
   public abstract int length()
      throws org.omg.CORBA.TypeCodePackage.BadKind;
   public abstract org.omg.CORBA.TypeCode content_type()
      throws org.omg.CORBA.TypeCodePackage.BadKind;
   public short fixed_digits()
      throws org.omg.CORBA.TypeCodePackage.BadKind;
   public short fixed_scale()
      throws org.omg.CORBA.TypeCodePackage.BadKind;
   public short member_visibility(int index)
      throws org.omg.CORBA.TypeCodePackage.BadKind,
```

```
            org.omg.CORBA.Bounds;
        public short type_modifier()
            throws org.omg.CORBA.TypeCodePackage.BadKind;
        public TypeCode concrete_base_type()
            throws org.omg.CORBA.TypeCodePackage.BadKind;
    }
```

17.4　DII リクエストを送信し，結果を受信

コードサンプル 17-3 および 17-4 に示した Request クラスは，正しく初期化されたあと，リクエストを送信する複数のメソッドを提供します。

17.4.1　リクエストを起動

リクエストを送信する最も単純な方法は，そのリクエストの invoke メソッドを呼び出すことです。このメソッドはリクエストを送信し，応答を待ってからクライアントプログラムに戻ります。return_value メソッドは，リターン値を表す Any オブジェクトのポインタ（C++）またはリファレンス（Java）を返します。

コードサンプル 17-18　invoke を使用してリクエストを送信する（C++）

```
try {
   . . .
   // Create request that will be sent to the account object
   request = account->_request("balance");
   // Set the result type
   request->set_return_type(CORBA::_tc_float);
   // Execute the request to the account object
   request->invoke();
   // Get the return balance
   CORBA::Float balance;
   CORBA::Any& balance_result = request->return_value();
   balance_result >>= balance;
   // Print out the balance
   cout << "The balance in " << name << "'s account is $"
        << balance << endl;
} catch(const CORBA::Exception& e) {
   cerr << e << endl;
   return 1;
}
return 0;
. . .
```

コードサンプル 17-19　invoke を使用してリクエストを送信する（Java）

```
try {
   . . .
   // Create request that will be sent to the account object
   request = account._request("balance");
   // Set the result type
   request.set_return_type(orb.get_primitive_tc
                (org.omg.CORBA.TCKind.tk_float));
   // Execute the request to the account object
   request.invoke();
   // Get the return balance
   float balance;
   org.omg.CORBA.Any balance_result = request.return_value();
   balance = balance_result.extract_float();
   // Print out the balance
   System.out.println("The balance in " + name +
                      "'s account is $" + balance);
} catch(Exception e) {
   e.printStackTrace();
}
```

17.4.2　send_deferred メソッドを使用して遅延 DII リクエストを送信

オペレーション要求の送信には，応答を待ち続けないメソッドである send_deferred も使用できます。このメソッドを使用したクライアントは，リクエストを送信したあと，poll_response メソッドを使用して応答が返ってきているかどうかを調べることができます。get_response メソッドは，応答を受信するまで待ちます。コードサンプル 17-20 に send_deferred メソッドと poll_response メソッドを使用して遅延

DII リクエストを送信する方法（C++），コードサンプル 17-21 に遅延 DII リクエストを送信する方法（Java）を示します。

コードサンプル 17-20　send_deferred メソッドと poll_response メソッドを使用して遅延 DII リクエストを送信する（C++）

```
. . .
try {
   // Create request that will be sent to the manager object
   CORBA::Request_var request = manager->_request("open");
   // Create argument to request
   CORBA::Any customer;
   customer <<= (const char *) name;
   CORBA::NVList_ptr arguments = request->arguments();
   arguments->add_value( "name" , customer, CORBA::ARG_IN );
   // Set result type
   request->set_return_type(CORBA::_tc_Object);
   // Creation of a new account can take some time
   // Execute the deferred request to the manager object
   request->send_deferred();
   VISPortable::vsleep(1);
   while (!request->poll_response()) {
      cout << " Waiting for response..." << endl;
      VISPortable::vsleep(1); // Wait one second between polls
   }
   request->get_response();
   // Get the return value
   CORBA::Object_var account;
   CORBA::Any& open_result = request->return_value();
   open_result >>= CORBA::Any::to_object(account.out());
. . .
}
```

コードサンプル 17-21　send_deferred メソッドと poll_response メソッドを使用して遅延 DII リクエストを送信する（Java）

```
try {
. . .
   // Create request that will be sent to the manager object
   org.omg.CORBA.Request request = manager._request("open");
   // Create argument to request
   org.omg.CORBA.Any customer = orb.create_any();
   customer.insert_string(name);
   org.omg.CORBA.NVList arguments = request.arguments();
   arguments.add_value("name",
                       customer, org.omg.CORBA.ARG_IN.value);
   // Set result type
   request.set_return_type(orb.get_primitive_tc
      (org.omg.CORBA.TCKind.tk_objref));
   // Creation of a new account can take some time
   // Execute the deferred request to the manager
   // object-plist
   request.send_deferred();
   Thread.currentThread().sleep(1000);
   while (!request.poll_response()) {
      System.out.println(" Waiting for response...");
      Thread.currentThread().sleep(1000);
                            // Wait one second between polls
   }
   request.get_response();
   // Get the return value
   org.omg.CORBA.Object account;
   org.omg.CORBA.Any open_result = request.return_value();
   account = open_result.extract_Object();
   . . .
}catch(Exception e) {
   e.printStackTrace();
}
```

17.4.3　send_oneway メソッドを使用して非同期 DII リクエストを送信

send_oneway メソッドを使用すると，非同期リクエストを送信できます。一方向リクエストは，オブジェクトインプリメンテーションからクライアントへ返される応答はありません。

17.4.4　複数のリクエストを送信

Request オブジェクトの配列を使用すると，DII Request オブジェクトのシーケンスを生成できます。リクエストのシーケンスを送信するには，VisiBroker ORB メソッドの send_multiple_requests_oneway か send_multiple_requests_deferred を使用します。リクエストのシーケンスを一方向リクエストとして送信した場合，どのリクエストにもサーバからの応答は期待できません。

C++の場合，コードサンプル 17-22 に，二つのリクエストがどのように生成され，リクエストのシーケンスを生成するためにどのように使用されるかを示します。シーケンスは，send_multiple_requests_deferred メソッドによって送信されます。

コードサンプル 17-22　send_multiple_requests_deferred メソッドによる複数の遅延リクエストの送信（C++）

```
. . .
// Create request to balance
try {
   req1 = account->_request("balance");
   // Create argument to request
   customer1 <<= (const char *) "Happy";
   CORBA::NVList_ptr arguments = req1->arguments();
   arguments->add_value("customer", customer1, CORBA::ARG_IN);
   // Set result
   . . .
} catch(const CORBA::Exception& excep) {
   cout << "Error while creating request" << endl;
   cout << excep << endl;
}
// Create request2 to slowBalance
try {
   req2 = account->_request("slowBalance");
   // Create argument to request
   customer2 <<= (const char *) "Sleepy";
   CORBA::NVList_ptr arguments = req2->arguments();
   arguments->add_value("customer", customer2, CORBA::ARG_IN);
   // Set result
   . . .
}catch(const CORBA::Exception& excep) {
   cout << "Error while creating request" << endl;
   cout << excep << endl;
}
// Create request sequence
CORBA::Request_ptr reqs[2];
reqs[0] = (CORBA::Request*) req1;
reqs[1] = (CORBA::Request*) req2;
CORBA::RequestSeq reqseq((CORBA::ULong)2, 2,
      (CORBA::Request_ptr *)reqs);
// Send the request
try {
   orb->send_multiple_requests_deferred(reqseq);
   cout << "Send multiple deferred calls are made..." << endl;
} catch(const CORBA::Exception& excep) {
. . .
```

17.4.5　複数のリクエストを受信

send_multiple_requests_deferred を使用してリクエストのシーケンスを送信した場合は，poll_next_response メソッドと get_next_response メソッドを使用して，サーバから各リクエストについて送信される応答を受信します。

VisiBroker ORB メソッド poll_next_response を使用すると，サーバから応答を受信したかどうかを調べることができます。このメソッドは，一つでも応答があれば true（真）を返し，何も応答がなければ false（偽）を返します。

VisiBroker ORB メソッド get_next_response を使用すると，応答を受信できます。何も応答がなければ，このメソッドは応答を受信するまで待ち続けます。クライアントプログラムを待たせ続けたくない場合は，まず poll_next_response メソッドを使用して応答が入手できるかどうかを調べ，次に get_next_response メソッドを使用して結果を受信します。コードサンプル 17-23 に複数のリクエストを送受信する VisiBroker ORB のメソッド（C++），コードサンプル 17-24 に複数のリクエストを送受信する VisiBroker ORB のメソッド（Java）を示します。

コードサンプル 17-23　複数のリクエストを送信し結果を受信する VisiBroker ORB メソッド（C++）

```
class CORBA {
   class ORB {
      . . .
      typedef sequence <Request_ptr> RequestSeq;
      void send_multiple_requests_oneway(const RequestSeq &);
      void send_multiple_requests_deferred(const RequestSeq &);
      Boolean poll_next_response();
      Status get_next_response();
      . . .
   };
};
```

コードサンプル 17-24　複数のリクエストを送信し結果を受信する VisiBroker ORB メソッド（Java）

```
package org.omg.CORBA;
public abstract class ORB {
   public abstract org.omg.CORBA.Environment
                                       create_environment();
   public abstract void send_multiple_requests_oneway(
                              org.omg.CORBA.Request[ ] reqs);
   public abstract void send_multiple_requests_deferred(
                              org.omg.CORBA.Request[ ] reqs);
   public abstract boolean poll_next_response();
   public abstract org.omg.CORBA.Request get_next_response();
   . . .
}
```

17.5　DII と一緒に IR を使用

DII Request オブジェクトに取り込む必要がある情報ソースの一つは，IR です（「16. インタフェースリポジトリの使用」参照）。次に示すのは，IR を使用してオペレーションのパラメタを取得する場合の例です。この例は，実際の DII アプリケーションでは一般的ではありませんが，リモートオブジェクトの型（Account）とそのメソッドの一つの名前（balance）を組み込み情報として持っています。実際の DII アプリケーションでは，その情報をソースの外部，例えばユーザから取得します。

例

- 任意の Account オブジェクトにバインドします。
- IR の中から Account の balance メソッドを検索し，IR の OperationDef からオペレーションリストを作成します。
- 引数と結果のコンポーネントを生成し，それらを_create_request メソッドに引き渡します。balance メソッドが例外を返さないことに注意してください。
- Request を起動し，結果を抽出して出力します。

コードサンプル 17-25　IR と DII の使用（C++）

```
// acctdii_ir.C
// This example illustrates IR and DII

#include <iostream.h>
#include "corba.h"

int main(int argc, char* const* argv) {
   CORBA::ORB_ptr orb;
   CORBA::Object_var account;

   CORBA::NamedValue_var result;
   CORBA::Any_ptr resultAny;
   CORBA::Request_var req;
   CORBA::NVList_var operation_list;

   CORBA::Any customer;
   CORBA::Float acct_balance;

try {
   // use argv[1] as the account name, or a default.
   CORBA::String_var name;
   if (argc == 2)
      name = (const char *) argv[1];
   else
      name = (const char *) "Default Name";
   try {
      // Initialize the ORB.
      orb = CORBA::ORB_init(argc, argv);
   } catch(const CORBA::Exception& excep) {
      cout << "Failure during ORB_init" << endl;
      cout << excep << endl;
      exit(1);
   }

   cout << "ORB_init succeeded" << endl;

   // Unlike traditional binds, this bind is called off of "orb"
   // and returns a generic object pointer based
   // on the interface name
   try {
      account = orb->bind("IDL:Account:1.0");
   } catch(const CORBA::Exception& excep) {
      cout << "Error binding to account" << endl;
      cout << excep << endl;
      exit(2);
   }
```

```
cout << "Bound to account object" << endl;

// Obtain Operation Description for the "balance" method of
// the Account
try {
   CORBA::InterfaceDef_var intf = account->_get_interface();
   if (intf == CORBA::InterfaceDef::_nil()) {
      cout << "Account returned a nil interface definition."
           << endl;
      cout << "Be sure an Interface Repository is
            running and"
           << endl;
      cout << "properly loaded" << endl;
      exit(3);
   }
   CORBA::Contained_var oper_container =
         intf->lookup("balance");
   CORBA::OperationDef_var oper_def =
   CORBA::OperationDef::_narrow(oper_container);
   orb->create_operation_list(
         oper_def, operation_list.out());

} catch(const CORBA::Exception& excep) {
   cout << "Error while obtaining operation list" << endl;
   cout << excep << endl;
   exit(4);
}

// Create request that will be sent to the account object
try {
   // Create placeholder for result
   orb->create_named_value(result.out());
   resultAny = result->value();
   resultAny->replace( CORBA::_tc_float, &result);

   // Set the argument value within the operation_list
   CORBA::NamedValue_ptr arg = operation_list->item(0);
   CORBA::Any_ptr anyArg = arg->value();
   *anyArg <<= (const char *) name;

   // Create the request
   account->_create_request(CORBA::Context::_nil(),
         "balance",
         operation_list,
         result,
         req.out(),
         0);

} catch(const CORBA::Exception& excep) {
   cout << "Error while creating request" << endl;
   cout << excep << endl;
   exit(5);
}

// Execute the request
try {
   req->invoke();
   CORBA::Environment_ptr env = req->env();
   if ( env->exception() ) {
      cout << "Exception occurred" << endl;
      cout << *(env->exception()) << endl;
      acct_balance = 0;
   } else {
      // Get the return value;
      acct_balance = *(CORBA::Float *)resultAny->value();
   }
} catch(const CORBA::Exception& excep) {
   cout << "Error while invoking request" << endl;
   cout << excep << endl;
   exit(6);
}

// Print out the results
cout << "The balance in " << name << "'s account is $";
```

```
   cout << acct_balance << "." << endl;
} catch ( const CORBA::Exception& excep ) {
   cout << "Error occurred" << endl;
   cout << excep << endl;
}
```

18 動的スケルトンインタフェースの使用

この章では, オブジェクトサーバがランタイム時にどのようにしてオブジェクトインプリメンテーションを動的に生成し, クライアントリクエストにサービスするかについて説明します。

18.1　動的スケルトンインタフェースとは

DSI（動的スケルトンインタフェース）は，生成されたスケルトンインタフェースから何も継承しないオブジェクトインプリメンテーションを生成する方式を提供します。通常, オブジェクトインプリメンテーションは idl2cpp コンパイラ（C++）または idl2java コンパイラ（Java）によって生成されたスケルトンクラスから派生します。DSI を使用すると，オブジェクトは idl2cpp コンパイラ（C++）または idl2java コンパイラ（Java）が生成したスケルトンクラスを継承しないで自分自身を VisiBroker ORB に登録し，クライアントからオペレーション要求を受信し，リクエストを処理し，結果をクライアントに返せます。

注

クライアントプログラムから見た場合，DSI を使用してインプリメントされたオブジェクトは，ほかの VisiBroker ORB オブジェクトとまったく同じように動作します。クライアントは，DSI を使用するオブジェクトインプリメンテーションと通信するために特別な処理を提供する必要はありません。

VisiBroker ORB は，オブジェクトの invoke メソッドを呼び出してそれを ServerRequest オブジェクトに引き渡すことによって, DSI オブジェクトインプリメンテーションにクライアントオペレーション要求を提示します。オブジェクトインプリメンテーションは，リクエストされたオペレーションを判断し，リクエストに対応する引数を解釈し，適切な内部メソッドまたはリクエストを満たすメソッドを呼び出し，適切な値を返します。

DSI を使用してオブジェクトをインプリメントするには，オブジェクトスケルトンが提供する通常の言語マッピングを使用した場合よりは手間の掛かるプログラミング作業が必要です。それでも, DSI を使用してインプリメントしたオブジェクトは，プロトコル間ブリッジを提供する場合に便利です。

18.1.1　idl2java コンパイラの使用（Java）

idl2java コンパイラにはフラグ（-dynamic_marshal）があり，このフラグをオンにすると DSI を使用してスケルトンコードを生成します。任意の型の DSI の使用方法を理解するには，IDL ファイルを生成し，-dynamic_marshal を使用してスケルトンコードを生成して調べます。

18.2　オブジェクトインプリメンテーションの動的生成手順

DSI を使用してオブジェクトインプリメンテーションを動的に生成するには，次の手順を実行します。

1. C++で IDL をコンパイルする場合, -type_code_info フラグを使用します。Java で IDL をコンパイルする場合，-dynamic_marshal フラグを使用します。
2. スケルトンクラスからオブジェクトインプリメンテーションを派生させる代わりに，PortableServer::DynamicImplementation abstract クラス（C++），または org.omg.PortableServer.DynamicImplementation インタフェース（Java）からオブジェクトインプリメンテーションを派生させるようにオブジェクトインプリメンテーションを設計します。
3. invoke メソッドを宣言し，インプリメントします。このメソッドは，VisiBroker ORB がクライアントリクエストをユーザのオブジェクトへ渡すために使用します。
4. デフォルトのサーバントとして，オブジェクトインプリメンテーション（POA サーバント）を POA マネージャに登録します。

18.2.1　DSI を使用したサンプルプログラムの格納場所

DSI の使用方法を示したサンプルプログラムは，Borland Enterprise Server VisiBroker をインストールしたディレクトリの examples/vbe/basic/bank_dynamic に入っています。この章では，このサンプルプログラムを使って，DSI の概念を説明します。IDL サンプル 18-1 に示す Bank.idl ファイルは，このサンプルでインプリメントされるインタフェースを示します。

IDL サンプル 18-1　DSI のサンプルプログラムで使用する Bank.idl ファイル

```
// Bank.idl
module Bank {
   interface Account {
      float balance();
   };
   interface AccountManager {
      Account open(in string name);
   };
};
```

18.3　DynamicImplementation クラスの継承

DSI を使用するには，コードサンプル 18-1 に示した DynamicImplementation ベースクラスからオブジェクトインプリメンテーションを派生させる必要があります。このクラスは，複数のコンストラクタと invoke メソッドを提供しますが，これらは自分でインプリメントしてください。

コードサンプル 18-1　DynamicImplementation ベースクラス（C++）

```
class PortableServer::DynamicImplementation :
      public virtual PortableServer::ServantBase {
   public:
      virtual void invoke(
            PortableServer::ServerRequest_ptr request) = 0;
. . .
};
```

コードサンプル 18-2　DynamicImplementation abstract クラス（Java）

```
package org.omg.CORBA;
public abstract class DynamicImplementation extends Servant {
   public abstract void invoke(ServerRequest request);
. . .
}
```

18.3.1　動的リクエスト用オブジェクトの設計例

コードサンプル 18-3 および 18-4 に，DSI を使用してインプリメントする C++および Java の AccountImpl クラスの宣言を示します。これは，invoke メソッドを宣言する DynamicImplementation クラスから派生します。VisiBroker ORB は，その invoke メソッドを呼び出して，クライアントオペレーション要求を ServerRequest オブジェクトの形でインプリメンテーションに引き渡します。

コードサンプル 18-3 に Account クラスコンストラクタと_primary_interface 関数を示します。

コードサンプル 18-3　dynamic から派生する AccountImpl クラスの例（C++）

```
class AccountImpl :
      public PortableServer::DynamicImplementation ,
               public virtual PortableServer::RefCountServantBase{
   public:
      AccountImpl(PortableServer::Current_ptr current,
            PortableServer::POA_ptr poa)
            : _poa_current(PortableServer::Current::_
                  duplicate(current)), _poa(poa)
      {}

      CORBA::Object_ptr get(const char *name) {
         CORBA::Float balance;
         // Check if account exists
         if (!_registry.get(name, balance)) {
            // simulate delay while creating new account
            VISPortable::vsleep(3);
            // Make up the account's balance,
                  between 0 and 1000 dollars
            balance = abs(rand()) % 100000 / 100.0;
            // Print out the new account
            cout << "Created " << name << "'s account: "
                  << balance << endl;
            _registry.put(name, balance);
         }
         // Return object reference
         PortableServer::ObjectId_var accountId =
         PortableServer::string_to_ObjectId(name);
         return _poa->create_reference_with_id(
               accountId, "IDL:Bank/Account:1.0");
      }
   private:
      AccountRegistry _registry;
      PortableServer::POA_ptr _poa;
```

```
        PortableServer::Current_var _poa_current;

        CORBA::RepositoryId _primary_interface(
              const PortableServer::ObjectId& oid,
                    PortableServer::POA_ptr poa) {
              return CORBA::string_dup(
                    (const char *)"IDL:Bank/Account:1.0");
        };

        void invoke(CORBA::ServerRequest_ptr request) {
        // Get the account name from the object id
        PortableServer::ObjectId_var oid =
              _poa_current->get_object_id();
        CORBA::String_var name;
        try {
           name = PortableServer::ObjectId_to_string(oid);
        } catch (const CORBA::Exception& e) {
           throw CORBA::OBJECT_NOT_EXIST();
        }
           // Ensure that the operation name is correct
           if (strcmp(request->operation(), "balance") != 0) {
              throw CORBA::BAD_OPERATION();
           }
           // Find out balance and fill out the result
           CORBA::NVList_ptr params = new CORBA::NVList(0);
           request->arguments(params);
           CORBA::Float balance;
           if (!_registry.get(name, balance))
              throw CORBA::OBJECT_NOT_EXIST();
           CORBA::Any result;
           result <<= balance;
           request->set_result(result);
           cout << "Checked " << name << "'s balance: "
                 << balance << endl;
        }
};
```

コードサンプル 18-4 に Account クラスコンストラクタの例を示します。

コードサンプル 18-4　Dynamic から派生する AccountImpl クラスの例（Java）

```
import java.util.*;
import org.omg.PortableServer.*;
public class AccountImpl extends DynamicImplementation {
   public AccountImpl(org.omg.CORBA.ORB orb, POA poa) {
      _orb = orb;
      _poa = poa;
   }

   public synchronized org.omg.CORBA.Object get(String name) {
      org.omg.CORBA.Object obj;
      // Check if account exists
      Float balance = (Float)_registry.get(name);
      if (balance == null) {
         // simulate delay while creating new account
         try {
            Thread.currentThread().sleep(3000);
         } catch (Exception e) {
            e.printStackTrace();
         }
         // Make up the account's balance, between 0 and 1000 dollars
         balance = new Float(Math.abs(_random.nextInt())
                                                % 100000 / 100f);
         // Print out the new account
         System.out.println("Created " + name + "'s account: " +
                balance.floatValue());
         _registry.put(name, balance);
      }
      // Return object reference
      byte[ ] accountId = name.getBytes();
      try {
         obj = _poa.create_reference_with_id(accountId,
                                          "IDL:Bank/Account:1.0");
      } catch (org.omg.PortableServer.POAPackage.WrongPolicy e) {
```

```
            throw new org.omg.CORBA.INTERNAL(e.toString());
         }
         return obj;
      }

      public String[ ] _all_interfaces(POA poa, byte[ ] objectId) {
                                                    return null; }

      public void invoke(org.omg.CORBA.ServerRequest request) {
         Float balance;
         // Get the account name from the object id
         String name = new String(_object_id());
         // Ensure that the operation name is correct
         if (!request.operation().equals("balance")) {
            throw new org.omg.CORBA.BAD_OPERATION();
         }
         // Find out balance and fill out the result
         org.omg.CORBA.NVList params = _orb.create_list(0);
         request.arguments(params);
         balance = (Float)_registry.get(name);
         if (balance == null) {
            throw new org.omg.CORBA.OBJECT_NOT_EXIST();
         }
         org.omg.CORBA.Any result = _orb.create_any();
         result.insert_float(balance.floatValue());
         request.set_result(result);
         System.out.println("Checked " + name + "'s balance: " +
                balance.floatValue());
      }
      private Random _random = new Random();
      static private Hashtable _registry = new Hashtable();
      private POA _poa;
      private org.omg.CORBA.ORB _orb;
   }
```

コードサンプル 18-5 および 18-6 に，DSI を使用してインプリメントする必要のある AccountManagerImpl クラスのインプリメンテーションを示します。これは，invoke メソッドを宣言する DynamicImplementation クラスからも派生します。VisiBroker ORB は，その invoke メソッドを呼び出して，クライアントオペレーション要求を ServerRequest オブジェクトの形でインプリメンテーションに引き渡します。

コードサンプル 18-5　Dynamic から派生する AccountManagerImpl クラスの例（C++）

```
class AccountManagerImpl
      : public PortableServer::DynamicImplementation,
        public virtual PortableServer::RefCountServantBase {
   public:
      AccountManagerImpl(AccountImpl* accounts)
            { _accounts = accounts; }

      CORBA::Object_ptr open(const char* name) {
         return _accounts->get(name);
      }

   private:
      AccountImpl* _accounts;
      CORBA::RepositoryId _primary_interface
            (const PortableServer::ObjectId& oid,
            PortableServer::POA_ptr poa) {
         return CORBA::string_dup((const char *)
               "IDL:Bank/AccountManager:1.0");
      };
      void invoke(CORBA::ServerRequest_ptr request) {
         // Ensure that the operation name is correct
         if (strcmp(request->operation(), "open") !=0)
            throw CORBA::BAD_OPERATION();
         // Fetch the input parameter
         char *name = NULL;
         try {
            CORBA::NVList_ptr params = new CORBA::NVList(1);
            CORBA::Any any;
            any <<= (const char*) "";
```

```
            params->add_value("name", any, CORBA::ARG_IN);
            request->arguments(params);
            *(params->item(0)->value()) >>= name;
         } catch (const CORBA::Exception& e) {
            throw CORBA::BAD_PARAM();
         }
         // Invoke the actual implementation and
         // fill out the result
         CORBA::Object_var account = open(name);
         CORBA::Any result;
         result <<= account;
         request->set_result(result);
      }
};
```

コードサンプル 18-6　Dynamic から派生する AccountManagerImpl クラスの例（Java）

```
import org.omg.PortableServer.*;
public class AccountManagerImpl extends DynamicImplementation {
   public AccountManagerImpl(org.omg.CORBA.ORB orb, AccountImpl
      accounts) { _orb =orb; _accounts =accounts;
   }

   public synchronized org.omg.CORBA.Object open(String name) {
      return _accounts.get(name);
   }
   public String[ ] _all_interfaces(POA poa, byte[ ] objectId)
      { return null; }

   public void invoke(org.omg.CORBA.ServerRequest request) {
      // Ensure that the operation name is correct
      if (!request.operation().equals("open")) {
         throw new org.omg.CORBA.BAD_OPERATION();
      }

      // Fetch the input parameter
      String name = null;
      try {
         org.omg.CORBA.NVList params = _orb.create_list(1);
         org.omg.CORBA.Any any = _orb.create_any();
         any.insert_string(new String(""));
         params.add_value("name", any, org.omg.CORBA.ARG_IN.value);
         request.arguments(params);
         name = params.item(0).value().extract_string();
      } catch (Exception e) {
         throw new org.omg.CORBA.BAD_PARAM();
      }
      // Invoke the actual implementation and fill out the result
      org.omg.CORBA.Object account = open(name);
      org.omg.CORBA.Any result = _orb.create_any();
      result.insert_Object(account);
      request.set_result(result);
   }

   private AccountImpl _accounts;
   private org.omg.CORBA.ORB _orb;
}
```

18.3.2　リポジトリ ID の指定

サポートされているリポジトリ ID を返すには，_primary_interface メソッドをインプリメントしなければなりません。正しいリポジトリ ID を指定するには，オブジェクトの IDL インタフェース名を使用して，次の手順に従います。

1. 区切り文字であるスコープ解決演算子「::」をすべて「/」に置換します。
2. 文字列の先頭に「IDL:」を付けます。
3. 文字列の最後に「:1.0」を付けます。

IDL インタフェース名の一例をコードサンプル 18-7 に，生成されるリポジトリ ID 文字列の一例をコードサンプル 18-8 に示します。

コードサンプル 18-7　IDL インタフェース名

```
Bank::AccountManager
```

コードサンプル 18-8　生成されるリポジトリ ID

```
IDL:Bank/AccountManager:1.0
```

18.4　ServerRequest クラスの考察

ServerRequest オブジェクトは，オブジェクトインプリメンテーションの invoke メソッドへパラメタとして引き渡されます。ServerRequest オブジェクトはオペレーション要求を示し，リクエストされたオペレーションの名前，パラメタリスト，およびコンテキストを取得するメソッドを提供します。また，呼び出し側へ返す結果を設定するメソッドと例外を反映させるメソッドも提供します。

コードサンプル 18-9　ServerRequest ベースクラス（C++）

```
class CORBA::ServerRequest {
   public:
   . . .
   CORBA::Context_ptr ctx();
   // POA spec methods
   const char *operation() const ;
   void arguments(CORBA::NVList_ptr param);
   void set_result(const CORBA::Any& a) ;
   void set_exception(const CORBA::Any& a);
};
```

コードサンプル 18-10　ServerRequest abstract クラス（Java）

```
package org.omg.CORBA;
public abstract class ServerRequest {
   public java.lang.String operation();
   public void arguments(org.omg.CORBA.NVList args);
   public void set_result(org.omg.CORBA.Any result);
   public void set_exception(org.omg.CORBA.Any except);
   public abstract org.omg.CORBA.Context ctx();
   // the following methods are deprecated
   public java.lang.String op_name(); // use operation()
   public void params(org.omg.CORBA.NVList params);
                                                  // use arguments()
   public void result(org.omg.CORBA.Any result); // use set_result()
   public abstract void except(org.omg.CORBA.Any except);
                                                // use set_exception()
}
```

C++の場合，arguments，set_result，または set_exception の各メソッドへ引き渡したすべての引数は，VisiBroker ORB の所有になります。これらの引数用のメモリは VisiBroker ORB によって解放されるので，ユーザで解放しないでください。

18.5　Account オブジェクトのインプリメント

Account インタフェースはメソッドを一つしか宣言しないので，AccountImpl クラスの invoke メソッドが実行する処理は非常に単純です。

invoke メソッドは最初に，リクエストされたオペレーションの名前が「balance」であるかどうかを調べます。この名前が一致しない場合は, BAD_OPERATION 例外が発生します。Account オブジェクトが複数のメソッドを提供する場合には，invoke メソッドは可能なすべてのオペレーション名について検査し，適切な内部メソッドを使用してオペレーション要求を処理する必要があります。

balance メソッドにはパラメタを指定できないので，オペレーション要求に対応するパラメタリストはありません。balance メソッドは単純に起動され，結果は Any オブジェクトの中にパッケージされ，その Any オブジェクトが ServerRequest オブジェクトの set_result メソッドを使用して呼び出し側へ返されます。

18.6　AccountManager オブジェクトのインプリメント

Account オブジェクトと同様に，AccountManager インタフェースもメソッドを一つしか宣言しません。しかし，AccountManagerImpl オブジェクトの open メソッドにはアカウント名パラメタを指定できるので，invoke メソッドが実行する処理は少しだけ複雑になります。コードサンプル 18-3（C++）および 18-4（Java）に，AccountManagerImpl オブジェクトの invoke メソッドのインプリメンテーションを示しています。

このメソッドは最初に，リクエストされたオペレーションの名前が「open」であるかどうかを調べます。この名前が一致しない場合は，BAD_OPERATION 例外が発生します。AccountManager オブジェクトが複数のメソッドを提供する場合には，invoke メソッドは可能なすべてのオペレーション名について検査し，適切な内部メソッドを使用してオペレーション要求を処理する必要があります。

(1)　入力パラメタを処理する

AccountManagerImpl オブジェクトの invoke メソッドがオペレーション要求の入力パラメタを処理するために使用する手順は，次のとおりです。

1. オペレーション用のパラメタリストを保持する NVList を生成します。
2. 予期されるパラメタごとに Any オブジェクトを生成し，TypeCode とパラメタ型（ARG_IN，ARG_OUT，ARG_INOUT のどれか）を設定して NVList に追加します。
3. ServerRequest オブジェクトの arguments メソッドを起動して NVList を引き渡し，リスト内のすべてのパラメタの値を更新します。

open メソッドはアカウント名パラメタを予期しているので，ServerRequest 内のパラメタを入れる NVList オブジェクトが生成されます。NVList クラスは，一つ以上の NamedValue オブジェクトが入っているパラメタリストをインプリメントします。NVList クラスと NamedValue クラスについては，「17.　動的起動インタフェースの使用」を参照してください。

アカウント名を入れる Any オブジェクトが生成されます。この Any は，引数名を「name」，パラメタ型を ARG_IN に設定して NVList に追加されます。

NVList が初期化されたあと，リスト内のすべてのパラメタ値を取得するために，ServerRequest オブジェクトの arguments メソッドが呼び出されます。

注

arguments メソッドを呼び出したあと，NVList は VisiBroker ORB に所有されます。したがって，オブジェクトインプリメンテーションが NVList 内の ARG_INOUT パラメタを変更すると，その変更は VisiBroker ORB にも自動的に認識されます。この NVList を呼び出し側で解放してはなりません。

入力引数用に NVList を構築する代わりに，VisiBroker ORB オブジェクトの create_operation_list メソッドを使用することもできます。このメソッドは OperationDef オブジェクトを受け取り，必要なすべての Any オブジェクトを使用して完全に初期化された NVList オブジェクトを返します。適切な OperationDef オブジェクトは，「16.　インタフェースリポジトリの使用」で説明した IR から取得できます。

(2)　リターン値を設定する

ServerRequest オブジェクトの arguments メソッドを起動したあと，name パラメタの値を抽出して新しい Account オブジェクトの生成に使用できます。新規に生成された Account オブジェクトを入れるため

の Any オブジェクトが生成され，ServerRequest オブジェクトの set_result メソッドを起動することによってその Any オブジェクトが呼び出し側へ返されます。

18.7　サーバのインプリメンテーション

main ルーチンのインプリメンテーションは，コードサンプル 18-11 および 18-12 に示すように，「4. Borland Enterprise Server VisiBroker によるサンプルアプリケーションの開発」で示したサンプルとほぼ同じものです。

コードサンプル 18-11　サーバのインプリメンテーション（C++）

```
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

      // Get a reference to the rootPOA
      CORBA::Object_var obj =
            orb->resolve_initial_references("RootPOA");
      PortableServer::POA_var rootPOA =
            PortableServer::POA::_narrow(obj);
      // Get the POA Manager
      PortableServer::POAManager_var poaManager =
            rootPOA->the_POAManager();
      // Create the account POA with the right policies
      CORBA::PolicyList accountPolicies;
      accountPolicies.length(3);
      accountPolicies [(CORBA::ULong)0] =
            rootPOA->create_servant_retention_policy(
            PortableServer::NON_RETAIN);
      accountPolicies[(CORBA::ULong)1] =
            rootPOA->create_request_processing_policy(
            PortableServer::USE_DEFAULT_SERVANT);
      accountPolicies[(CORBA::ULong)2] =
            rootPOA->create_id_uniqueness_policy(
            PortableServer::MULTIPLE_ID);
      PortableServer::POA_var accountPOA = rootPOA->create_POA(
            "bank_account_poa",
            poaManager,
            accountPolicies);
     // Create the account default servant
      PortableServer::Current_var current =
            PortableServer::Current::_instance();
      AccountImpl accountServant(current, accountPOA);
      accountPOA->set_servant(&accountServant);
      // Create the manager POA with the right policies
      CORBA::PolicyList managerPolicies;
      managerPolicies.length(3);
      managerPolicies[(CORBA::ULong)0] =
            rootPOA->create_lifespan_policy(
            PortableServer::PERSISTENT);
      managerPolicies[(CORBA::ULong)1] =
            rootPOA->create_request_processing_policy(
            PortableServer::USE_DEFAULT_SERVANT);
      managerPolicies[(CORBA::ULong)2] =
            rootPOA->create_id_uniqueness_policy(
            PortableServer::MULTIPLE_ID);
      PortableServer::POA_var managerPOA =
            rootPOA->create_POA(
            "bank_agent_poa",
            poaManager,
            managerPolicies);
      // Create the manager default servant
      AccountManagerImpl managerServant(&accountServant);
      managerPOA->set_servant(&managerServant);
      // Activate the POA Manager
      poaManager->activate();
      cout << "AccountManager is ready" << endl;
      // Wait for incoming requests
      orb->run();
   } catch(const CORBA::Exception& e) {
      cerr << e << endl;
      return 1;
   }
```

```
    return 0;
  }
```

コードサンプル 18-12　サーバのインプリメンテーション（Java）

```
  import org.omg.PortableServer.*;
  public class Server {
    public static void main(String[ ] args) {
      try {
        // Initialize the ORB
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
        // Get a reference to the rootPOA
        POA rootPOA = POAHelper.narrow(orb.
                              resolve_initial_references("RootPOA"));
        // Get the POA Manager
        POAManager poaManager = rootPOA.the_POAManager();
        // Create the account POA with the right policies
        org.omg.CORBA.Policy[ ] accountPolicies = {
          rootPOA.create_servant_retention_policy(
            ServantRetentionPolicyValue.NON_RETAIN),
          rootPOA.create_request_processing_policy(
            RequestProcessingPolicyValue.USE_DEFAULT_SERVANT)
        };
        POA accountPOA = rootPOA.create_POA("bank_account_poa",
              poaManager, accountPolicies);
        // Create the account default servant
        AccountImpl accountServant = new AccountImpl(orb,
                                                    accountPOA);
        accountPOA.set_servant(accountServant);
        // Create the manager POA with the right policies
        org.omg.CORBA.Policy[ ] managerPolicies = {
         rootPOA.create_lifespan_policy(
                 LifespanPolicyValue.PERSISTENT),
         rootPOA.create_request_processing_policy(
                 RequestProcessingPolicyValue.USE_DEFAULT_SERVANT)
        };
        POA managerPOA = rootPOA.create_POA("bank_agent_poa",
              poaManager,managerPolicies);
        // Create the manager default servant
        AccountManagerImpl managerServant =
                          new AccountManagerImpl(orb, accountServant);
        managerPOA.set_servant(managerServant);
        // Activate the POA Manager
        poaManager.activate();
        System.out.println("AccountManager is ready");
        // Wait for incoming requests
        orb.run();
      } catch(Exception e) {
        e.printStackTrace();
      }
    }
  }
```

DSI インプリメンテーションは，デフォルトサーバントとして実体化されます。また，POA は該当するポリシーのサポートによって生成されなければなりません。詳細については，「7.　POA の使用」を参照してください。

19 ポータブルインタセプタの使用

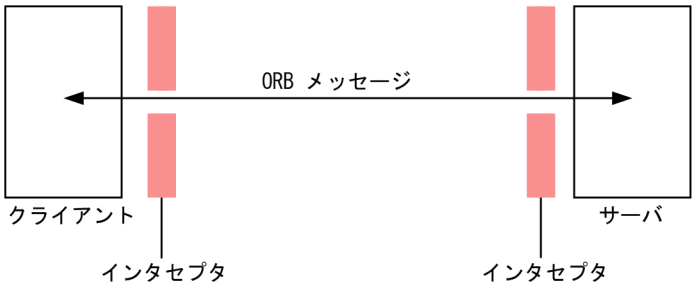
この章では, OMG 規格で定義されている新しいポータブルインタセプタの概要について説明し，ポータブルインタセプタの例を示しながら，ポータブルインタセプタファクトリなどの高度な機能について説明します。ポータブルインタセプタの詳細については, OMG 採用確定規格のポータブルインタセプタについての記述を参照してください。

19.1　概要

Borland Enterprise Server VisiBroker ORB は，セキュリティ，トランザクション，ログなどの機能を ORB へ追加するための一連のインタフェースをインタセプタとして提供しています。これらのインタセプタインタフェースは，コールバック方式に基づいています。例えば，インタセプタを使用するとクライアント／サーバ間の通信を知ることができ，それらの通信を希望に応じて修正し，Borland Enterprise Server VisiBroker ORB の動作を効果的に変更できます。

最も単純な使用方法として，インタセプタはコードのトレースに便利です。クライアント／サーバ間で送信されるメッセージを見られるので，ORB がリクエストをどう処理しているのかを正確に確認できます。インタセプタの機能を図 19-1 に示します。

図 19-1　インタセプタの機能



モニタリングツールやセキュリティレイヤなどの，より高度なアプリケーションを開発する場合は，それらを開発するのに必要な低レベルアプリケーションの情報や制御をインタセプタから得られます。例えば，各種のサーバの動作を監視し，負荷分散を実行するアプリケーションを開発できます。

Borland Enterprise Server VisiBroker ORB がサポートするインタセプタには，ポータブルインタセプタ（単にインタセプタと呼ぶこともあります）および VisiBroker Interceptor の 2 種類があります。ポータブルインタセプタは OMG 標準化機能であり，インタセプタのポータブルコードの書き込みと各種ベンダ ORB での使用が可能となります。VisiBroker 4.x インタセプタは，VisiBroker 4.x で定義され，インプリメントされるインタセプタです（VisiBroker インタセプタの詳細については，「20.　VisiBroker 4.x インタセプタの使用」を参照してください）。

OMG 規格が定義するポータブルインタセプタには次の 2 種類があります。

- リクエストインタセプタを使用すると，クライアントとサーバ間で Borland Enterprise Server VisiBroker ORB サービスがコンテキスト情報を転送できるようになります。リクエストインタセプタは，さらにクライアントリクエストインタセプタとサーバリクエストインタセプタに分けられます。
- IOR インタセプタを使用すると，Borland Enterprise Server VisiBroker ORB サービスがサーバまたはオブジェクトの ORB サービス関連機能を説明する情報を IOR に追加できるようになります。例えば，SSL のようなセキュリティサービスがそのタグの付いたコンポーネントを IOR に追加するので，そのコンポーネントを認識したクライアントは，そのコンポーネントの情報に基づいてサーバとのコネクションを設定できます。

ポータブルインタセプタの詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「ポータブルインタセプタインタフェースとクラス」の記述を参照してください。

ポータブルインタセプタと VisiBroker インタセプタの使用については，「20.　VisiBroker 4.x インタセプタの使用」を参照してください。

19.2　ポータブルインタセプタおよび情報インタフェース

すべてのポータブルインタセプタは次に示すベースインタセプタ API クラスのどれかをインプリメントします。ベースインタセプタ API クラスは Borland Enterprise Server VisiBroker ORB によって定義され，インプリメントされます。

- リクエストインタセプタ
  - ClientRequestInterceptor
  - ServerRequestInterceptor
- IOR インタセプタ

19.2.1　インタセプタ

リクエストインタセプタおよび IOR インタセプタは，共通クラスである Interceptor クラスから派生します。この Interceptor クラスは共通メソッドを定義しており，この共通メソッドは Interceptor クラスの継承クラスにも使用できます。

コードサンプル 19-1　Interceptor クラス（C++）

```
class PortableInterceptor::Interceptor
{
     virtual char* name()=0;
     virtual void destroy()=0;
}
```

コードサンプル 19-2　Interceptor インタフェース（Java）

```
public interface Interceptor
extends org.omg.CORBA.portable.IDLEntity,org.omg.CORBA.LocalInterface
{
     public java.lang.String name();
     public void destroy();
}
```

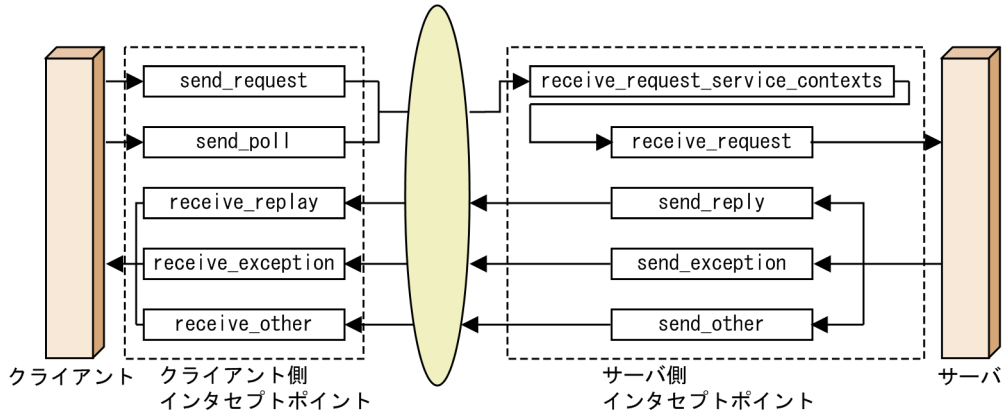
19.2.2　リクエストインタセプタ

リクエストインタセプタは，特定のインタセプトポイントでリクエスト／応答シーケンスのフローを受け取るために使用します。これによってサービスはクライアントとサーバ間でコンテキスト情報を転送できます。各インタセプトポイントでは Borland Enterprise Server VisiBroker ORB はオブジェクトを与えて，このオブジェクトによってインタセプタはリクエスト情報にアクセスできます。リクエストインタセプタには 2 種類あり，そのそれぞれにリクエスト情報インタフェースがあります。

- ClientRequestInterceptor および ClientRequestInfo
- ServerRequestInterceptor および ServerRequestInfo

リクエストのインタセプタポイントを図 19-2 に示します。

図 19-2　リクエストのインタセプタポイント



リクエストインタセプタの詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「ポータブルインタセプタインタフェースとクラス」の記述を参照してください。

(1)　ClientRequestInterceptor

ClientRequestInterceptor には，クライアント側でインプリメントされたインタセプトポイントがあります。

表 19-1 に示すように，OMG が ClientRequestInterceptor で定義しているインタセプトポイントは五つあります。

表 19-1　ClientRequestInterceptor インタセプトポイント

インタセプトポイント	説明
send_request	クライアント側インタセプタが，リクエストがサーバに送信される前にリクエストを照会して，サービスコンテキストを修正します。
send_poll	クライアント側インタセプタが，TII（時間非依存呼び出し）ポーリング取得応答シーケンス※中にリクエストを照会します。
receive_reply	クライアント側インタセプタが，サーバから応答情報が戻されたあと，クライアントに制御が移る前にその応答情報を照会します。
receive_exception	クライアント側インタセプタが，例外発生時に，その例外がクライアントに送信される前に例外情報を照会します。
receive_other	クライアント側インタセプタが，正常応答または例外以外のリクエスト結果を受け取った場合に利用できる情報を照会します。

注※

TII は VisiBroker Edition ORB ではインプリメントされていません。結果として，send_poll()インタセプトポイントが呼び出されることはありません。

各インタセプトポイントの詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「ポータブルインタセプタインタフェースとクラス」の記述を参照してください。

コードサンプル 19-3　ClientRequestInterceptor クラス（C++）

```
class _VISEXPORT ClientRequestInterceptor:public virtualInterceptor {
    public:
    virtual void send_request(ClientRequestInfo_ptr _ri)=0;
    virtual void send_poll(ClientRequestInfo_ptr _ri)=0;
    virtual void receive_reply(ClientRequestInfo_ptr _ri)=0;
    virtual void receive_exception(ClientRequestInfo_ptr _ri)=0;
    virtual void receive_other(ClientRequestInfo_ptr _ri)=0;
}
```

コードサンプル 19-4　ClientRequestInterceptor インタフェース（Java）

```
package org.omg.PortableInterceptor;
public interface ClientRequestInterceptor
      extends Interceptor,org.omg.CORBA.portable.IDLEntity,
               org.omg.CORBA.LocalInterface
{
      public void send_request(ClientRequestInfo ri)
             throws ForwardRequest;
      public void send_poll(ClientRequestInfo ri)
             throws ForwardRequest;
      public void receive_reply(ClientRequestInfo ri);
      public void receive_exception(ClientRequestInfo ri)
             throws ForwardRequest;
      public void receive_other(ClientRequestInfo ri)
             throws ForwardRequest;
}
```

クライアント側の規則を次に，具体例を表 19-2 に示します。

- 開始インタセプトポイントは send_request および send_poll です。どのリクエスト／応答シーケンスでも，このインタセプトポイントのどちらか一方だけが呼び出されます。
- 終了インタセプトポイントは receive_reply，receive_exception，および receive_other です。どのリクエスト／応答シーケンスでも，このインタセプトポイントのどれか一つだけが呼び出されます。
- 中間インタセプトポイントはありません。
- 終了インタセプトポイントは，send_request または send_poll のどちらかの実行が成功した場合だけ呼び出されます。
- receive_exception は，ORB のシャットダウンによってリクエストがキャンセルされて，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 が発生すると呼び出されます。
- receive_exception は，リクエストがそのほかの理由によってキャンセルされて，マイナーコード 3 のシステム例外 TRANSIENT が発生すると呼び出されます。

表 19-2　クライアント側の規則の具体例

クライアント側の規則	説明
呼び出し成功	send_request のあとに receive_reply。開始点のあとに終了点がきます。
リトライ	send_request のあとに receive_other。開始点のあとに終了点がきます。

(2)　ServerRequestInterceptor

ServerRequestInterceptor には，サーバ側でインプリメントされたインタセプトポイントがあります。表 19-3 に示すように，OMG が ServerRequestInterceptor で定義しているインタセプトポイントは五つあります。

表 19-3　ServerRequestInterceptor インタセプトポイント

インタセプトポイント	説明
receive_request_service_contexts	サーバ側インタセプタが，入力リクエストからそのサービスコンテキスト情報を取得して，それを PortableInterceptor::Current のスロットに転送します。
receive_request	サーバ側インタセプタが，オペレーションパラメタのようなすべての情報が利用可能になってから，リクエスト情報を照会します。
send_reply	サーバ側インタセプタが，ターゲットのオペレーションが呼び出されたあと，クライアントに応答が戻される前に応答情報を照会して，応答サービスコンテキストを修正します。
send_exception	サーバ側インタセプタが，例外発生時に，その例外がクライアントに送信される前に例外情報を照会して応答サービスコンテキストを修正します。
send_other	サーバ側インタセプタが，正常応答または例外以外のリクエスト結果を受け取った場合に利用できる情報を照会します。

各インタセプトポイントの詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「ポータブルインタセプタインタフェースとクラス」の記述を参照してください。

コードサンプル 19-5　ServerRequestInterceptor クラス（C++）

```
class _VISEXPORT ServerRequestInterceptor:public virtual Interceptor {
      public:
      virtual void receive_request_service_contexts
         (ServerRequestInfo_ptr _ri)=0;
      virtual void receive_request(ServerRequestInfo_ptr _ri)=0;
      virtual void send_reply(ServerRequestInfo_ptr _ri)=0;
      virtual void send_exception(ServerRequestInfo_ptr _ri)=0;
      virtual void send_other(ServerRequestInfo_ptr _ri)=0;
};
```

コードサンプル 19-6　ServerRequestInterceptor インタフェース（Java）

```
package org.omg.PortableInterceptor;

public interface ServerRequestInterceptor
   extends Interceptor,org.omg.CORBA.portable.IDLEntity,
        org.omg.CORBA.LocalInterface
{
     public void receive_request_service_contexts
         (ServerRequestInfo ri)throws ForwardRequest;
     public void receive_request(ServerRequestInfo ri)
          throws ForwardRequest;
     public void send_reply(ServerRequestInfo ri);
     public void send_exception(ServerRequestInfo ri)
          throws ForwardRequest;
     public void send_other(ServerRequestInfo ri)throws ForwardRequest;
}
```

サーバ側の規則を次に，具体例を表 19-4 に示します。

- 開始インタセプトポイントは receive_request_service_contexts です。どのリクエスト／応答シーケンスでも，このインタセプトポイントが呼び出されます。
- 終了インタセプトポイントは send_reply，send_exception，および send_other です。どのリクエスト／応答シーケンスでも，このインタセプトポイントのどれか一つだけが呼び出されます。
- 中間インタセプトポイントは receive_request です。これは，receive_request_service_contexts のあと，終了インタセプトポイントの前に呼び出されます。

- 例外では，receive_request は呼び出されないことがあります。
- 終了インタセプトポイントは，send_request または send_poll のどちらかの実行が成功した場合だけ呼び出されます。
- send_exception は，ORB のシャットダウンによってリクエストがキャンセルされて，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 が発生すると呼び出されます。
- send_exception は，リクエストがそのほかの理由によってキャンセルされて，マイナーコード 3 のシステム例外 TRANSIENT が発生すると呼び出されます。

表 19-4　サーバ側の規則の具体例

サーバ側の規則	説明
呼び出し成功	インタセプトポイントの順序 receive_request_service_contexts，receive_request，send_reply。開始点，中間点，終了点の順。

19.2.3　IOR インタセプタ

(1)　IORInterceptor

IORInterceptor は，クライアント側の Borland Enterprise Server VisiBroker ORB サービスインプリメンテーションが正しく機能できるように，サーバまたはオブジェクトの ORB サービス関連機能を説明する情報をオブジェクトリファレンスに追加する機能をアプリケーションに提供します。これは，インタセプトポイント establish_components を呼び出すことによって提供されます。IORInfo のインスタンスは，インタセプトポイントに渡されます。IORInfo の詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「IORInfo」の記述を参照してください。

コードサンプル 19-7　IORInterceptor クラス（C++）

```
class _VISEXPORT IORInterceptor:public virtual Interceptor
{
   public:
      virtual void establish_components(IORInfo_ptr _info)=0;
      virtual void components_established(IORInfo_ptr _info)=0;
      virtual void adapter_manager_state_changed(
         CORBA::Long _id,CORBA::Short _state)=0;
      virtual void adapter_state_changed(
         const ObjectReferenceTemplateSeq&_templates,
         CORBA::Short _state)=0;
};
```

コードサンプル 19-8　IORInterceptor インタフェース（Java）

```
package org.omg.PortableInterceptor;

public interface IORInterceptor
   extends Interceptor,org.omg.CORBA.portable.IDLEntity,
        org.omg.CORBA.LocalInterface
{
     public void establish_components(IORInfo info);
     public void components_established(IORInfo info);
     public void adapter_manager_state_changed(int id,short state);
     public void adapter_state_changed(
          ObjectReferenceTemplate[ ] templates, short state);
}
```

19.2.4　Portable Interceptor Current

PortableInterceptor::Current オブジェクト（以降，PICurrent と呼びます）は，スレッドコンテキスト情報をリクエストコンテキストに転送するためにポータブルインタセプタが使用するスロットのテーブルです。PICurrent の使用は必須ではありませんが，インタセプトポイントでクライアントのスレッドコンテキ

スト情報が必要である場合は，この情報を転送するために PICurrent を使用できます。PICurrent は次に示す呼び出しによって取得されます。

Java の場合

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

C++の場合

```
ORB->resolve_initial_references ("PICurrent");
```

コードサンプル 19-9　PortableInterceptor::Current クラス（C++）

```
class _VISEXPORT Current:public virtual CORBA::Current,
      public virtual CORBA_Object
{
   public:
      virtual CORBA::Any*get_slot(CORBA::ULong _id);
      virtual void set_slot(CORBA::ULong _id,
              const CORBA::Any&_data);
};
```

コードサンプル 19-10　PortableInterceptor.Current インタフェース（Java）

```
package org.omg.PortableInterceptor;

public interface Current
   extends org.omg.CORBA.CurrentOperations,
      org.omg.CORBA.portable.IDLEntity
{
   public org.omg.CORBA.Any get_slot(int id)throws InvalidSlot;
   public void set_slot(int id,org.omg.CORBA.Any data)throws
          InvalidSlot;
}
```

19.2.5　Codec

Codec は，コンポーネントの IDL データ型と CDR カプセル化表現の間でコンポーネントを転送する機能をインタセプタに提供します。Codec は，CodecFactory から取得されます。

コードサンプル 19-11　Codec クラス（C++）

```
class _VISEXPORT Codec
{
   public:
      virtual CORBA::OctetSequence* encode(const CORBA::Any& _data)=0;
      virtual CORBA::Any* decode(const CORBA::OctetSequence& _data)=0;
      virtual CORBA::OctetSequence* encode_value
                                        (const CORBA::Any& _data)=0;
      virtual CORBA::Any* decode_value(const
             CORBA::OctetSequence& _data, CORBA::TypeCode_ptr _tc)=0;
};
```

コードサンプル 19-12　Codec インタフェース（Java）

```
package org.omg.IOP;

public interface Codec
   extends org.omg.CORBA.portable.IDLEntity,org.omg.CORBA.LocalInterface
{
   public byte[ ] encode(org.omg.CORBA.Any data)
       throws InvalidTypeForEncoding;
   public org.omg.CORBA.Any decode(byte[ ] data)throws
                                                  FormatMismatch;
   public byte[ ] encode_value(org.omg.CORBA.Any data)
      throws InvalidTypeForEncoding;
   public org.omg.CORBA.Any decode_value(byte[ ] data,
      org.omg.CORBA.TypeCode tc)
      throws FormatMismatch,TypeMismatch;
}
```

19.2.6　CodecFactory

このクラスは，エンコーディングフォーマット，メジャーバージョン，およびマイナーバージョンを指定して Codec オブジェクトを生成するために使用します。CodecFactory は次の呼び出しによって取得されます。

Java の場合

```
ORB.resolve_initial_references ("CodecFactory")
```

C++の場合

```
ORB->resolve_initial_references("CodecFactory")
```

コードサンプル 19-13 CodecFactory クラス（C++）

```
class _VISEXPORT CodecFactory
{
   public:
      virtual Codec_ptr create_codec(const Encoding&_enc)=0;
};
```

コードサンプル 19-14　CodecFactory インタフェース（Java）

```
public interface CodecFactory
     extends org.omg.CORBA.portable.IDLEntity,
                                          org.omg.CORBA.LocalInterface
   {
    public Codec create_codec(Encoding enc) throws
                                          UnknownEncoding;
   }
```

19.2.7　ポータブルインタセプタの作成

ポータブルインタセプタの一般的な作成手順を次に示します。

1. インタセプタは次のインタセプタインタフェースの一つから継承されなければなりません。
   - ClientRequestInterceptor
   - ServerRequestInterceptor
   - IORInterceptor
2. インタセプタは，インタセプタで使用できる一つ以上のインタセプトポイントをインプリメントします。
3. インタセプタは名前付き，または名前なしでもかまいません。すべての名前は同じタイプのインタセプタすべての中では一意である必要がありますが，名前なしのインタセプタは幾つでも Borland Enterprise Server VisiBroker ORB に登録できます。

コードサンプル 19-15　ポータブルインタセプタの作成例（C++）

```
#include "PortableInterceptor_c.hh"

class SampleClientRequestInterceptor:public
   PortableInterceptor::ClientRequestInterceptor
{
   char *name(){
      return "SampleClientRequestInterceptor";
   }

   void send_request(ClientRequestInfo_ptr _ri){
     ......//actual interceptor code here
   }

   void send_poll(ClientRequestInfo_ptr _ri){
     ......//actual interceptor code here
   }
```

```
    void receive_reply(ClientRequestInfo_ptr _ri){
      ......//actual interceptor code here
    }

    void receive_exception(ClientRequestInfo_ptr _ri){
      ......//actual interceptor code here
    }

    void receive_other(ClientRequestInfo_ptr _ri){
      ......//actual interceptor code here
    }
};
```

コードサンプル 19-16　ポータブルインタセプタの作成例（Java）

```
import org.omg.PortableInterceptor.*;

public class SampleClientRequestInterceptor extends
    org.omg.CORBA.LocalObject
    implements ClientRequestInterceptor
{
    public java.lang.String name(){
        return "SampleClientRequestInterceptor";
    }

    public void send_request(ClientRequestInfo ri)
        throws ForwardRequest {
        ......//actual interceptor code here
    }

    public void send_poll(ClientRequestInfo ri)
        throws ForwardRequest {
        ......//actual interceptor code here
    }

    public void receive_reply(ClientRequestInfo ri){
        ......//actual interceptor code here
    }

    public void receive_exception(ClientRequestInfo ri)
        throws ForwardRequest {
        ......//actual interceptor code here
    }

    public void receive_other(ClientRequestInfo ri)
        throws ForwardRequest {
        ......//actual interceptor code here
    }
}
```

19.2.8　ポータブルインタセプタの登録

ポータブルインタセプタは使用前に Borland Enterprise Server VisiBroker ORB に登録する必要があります。ポータブルインタセプタを登録するには，ORBInitializer オブジェクトをインプリメントして，登録する必要があります。ポータブルインタセプタは，pre_init ()メソッドもしくは post_init()メソッド，またはその両方をインプリメントする対応 ORBInitializer オブジェクトを登録することによって，ORB 初期化時に実体化されて登録されます。Borland Enterprise Server VisiBroker ORB は，初期化プロセス時に ORBInitInfo オブジェクトに登録された各 ORBInitializer を呼び出します。

コードサンプル 19-17　ORBInitializer クラス（C++）

```
class _VISEXPORT ORBInitializer
{
    public:
        virtual void pre_init(ORBInitInfo_ptr _info)=0;
        virtual void post_init(ORBInitInfo_ptr _info)=0;
};
```

コードサンプル 19-18　ORBInitInfo クラス（C++）

```
class _VISEXPORT ORBInitInfo
{
   public:
      virtual CORBA::StringSequence*arguments()=0;
      virtual char*orb_id()=0;
      virtual IOP::CodecFactory_ptr codec_factory()=0;
      virtual void register_initial_reference(const char*_id,
         CORBA::Object_ptr _obj)=0;
      virtual CORBA::Object_ptr resolve_initial_references
         (const char*_id)=0;
      virtual void add_client_request_interceptor(
         ClientRequestInterceptor_ptr _interceptor)=0;
      virtual void add_server_request_interceptor(
         ServerRequestInterceptor_ptr _interceptor)=0;
      virtual void add_ior_interceptor(IORInterceptor_ptr
                                          _interceptor)=0;
      virtual CORBA::ULong allocate_slot_id()=0;
      virtual void register_policy_factory(CORBA::ULong _type,
         PolicyFactory_ptr _policy_factory)=0;
};
```

コードサンプル 19-19　ORBInitializer インタフェース（Java）

```
package org.omg.PortableInterceptor;

public interface ORBInitializer
   extends org.omg.CORBA.portable.IDLEntity,
      org.omg.CORBA.LocalInterface
{
   public void pre_init(ORBInitInfo info);
   public void post_init(ORBInitInfo info);
}
```

コードサンプル 19-20　ORBInitInfo インタフェース（Java）

```
package org.omg.PortableInterceptor;

public interface ORBInitInfo
   extends org.omg.CORBA.portable.IDLEntity,org.omg.CORBA.LocalInterface
{
   public java.lang.String[ ] arguments((;
   public java.lang.String orb_id();
   public CodecFactory codec_factory();
   public void register_initial_reference
      (java.lang.String id,org.omg.CORBA.Object obj)
      throws InvalidName;
   public void resolve_initial_references(java.lang.String id)
      throws InvalidName;
   public void add_client_request_interceptor
      (ClientRequestInterceptor interceptor)
      throws DuplicateName;
   public void add_server_request_interceptor
      (ServerRequestInterceptor interceptor)
      throws DuplicateName;
   public void add_ior_interceptor(IORInterceptor interceptor)
      throws DuplicateName;
   public int allocate_slot_id();
   public void register_policy_factory
      (int type,PolicyFactory policy_factory);
}
```

19.2.9　ORBInitializer の登録

ORBInitializer を登録するために，グローバルメソッドである register_orb_initializer が提供されています。インタセプタをインプリメントする各サービスは，ORBInitializer のインスタンスを提供します。サービスを使用するために，アプリケーションは次の手順を行います。

1. サービスの ORBInitializer で register_orb_initializer()メソッドを呼び出します。
2. 新しい ORB 識別子で ORB_Init()メソッド呼び出しを実体化して新しい ORB を生成します。

Java の場合

register_orb_initializer()メソッドは，グローバルメソッドであるため，ORB に関するアプレットセキュリティに違反します。結果として，ORBInitializer は，register_orb_initializer()メソッドを呼び出すのではなく，Java ORB プロパティを使用して VisiBroker Edition ORB に登録されます。

(1)　新しいプロパティセット（Java）

新しいプロパティ名の形式を次に示します。

```
org.omg.PortableInterceptor.ORBInitializerClass.<Service>
```

ここで<Service>は，org.omg.PortableInterceptor.ORBInitializer をインプリメントするクラスの文字列名です。

ORB.init()メソッドの実行中に行われることを次に示します。

1. org.omg.PortableInterceptor.ORBInitializerClass で始まる ORB プロパティが収集されます。
2. 各プロパティの<Service>部分が収集されます。
3. オブジェクトは，そのクラス名として<Service>文字列を使用して実体化されます。
4. pre_init()メソッドおよび post_init()メソッドがそのオブジェクトで呼び出されます。
5. 例外が発生すると，ORB はそれを無視して続行します。

注

名前の衝突を避けるために，DNS 名の入れ替え規則を適用することを推奨します。例えば，ABC 社に二つのイニシャライザがある場合，次のプロパティを定義できます。

```
org.omg.PortableInterceptor.ORBInitializerClass.com.abc.ORBInit1
org.omg.PortableInterceptor.ORBInitializerClass.com.abc.ORBInit2
```

C++の場合

register_orb_initializer メソッドは PortableInterceptor モジュールに次のように定義されます。

```
class _VISEXPORT PortableInterceptor {
    static void register_orb_initializer(ORBInitializer *init);
}
```

(2)　サンプル

ABC 社が書き込んだクライアント側の監視ツールに，次に示す ORBInitializer インプリメンテーションがあるとします。

コードサンプル 19-21　ORBInitializer の登録例（C++）

```
#include "PortableInterceptor_c.hh"

class MonitoringService:public PortableInterceptor::ORBInitializer
{
   void pre_init(ORBInitInfo_ptr _info)
   {
      //instantiate the service ' s Interceptor.
      Interceptor*interceptor =new MonitoringInterceptor();

      //register the Monitoring ' s Intereptor.
      _info->add_client_request_interceptor(interceptor);
   }

   void post_init(ORBInitInfo_ptr _info)
   {
      //This init point is not needed.
   }
};
```

```
MonitoringService *monitoring_service =new MonitoringService();
PortableInterceptor::register_orb_initializer(monitoring_service);
```

コードサンプル 19-22　ORBInitializer の登録例（Java）

```
package com.abc.Monitoring;

import org.omg.PortableInterceptor.Interceptor;
import org.omg.PortableInterceptor.ORBInitializer;
import org.omg.PortableInterceptor.ORBInitInfo;

public class MonitoringService extends org.omg.CORBA.LocalObject
   implements org.omg.PortableInterceptor.ORBInitializer
{
   void pre_init(ORBInitInfo info)
   {
      //instantiate the service ' s Interceptor.
      Interceptor interceptor =new MonitoringInterceptor();

      //register the Monitoring ' s Interceptor.
      info.add_client_request_interceptor(interceptor);
   }

   void post_init(ORBInitInfo info)
   {
      //This init point is not needed.
   }
}
```

この監視サービスを使用して MyApp と呼ばれるプログラムを実行するために次のコマンドが使用されます。

```
java -Dorg.omg.PortableInterceptor.ORBInitializerClass.
     com.abc.Monitoring.MonitoringService MyApp
```

19.2.10　ポータブルインタセプタの Borland Enterprise Server VisiBroker 拡張機能

(1)　POA スコープサーバリクエストインタセプタ

OMG が指定したポータブルインタセプタは，グローバルにスコープされます。Borland Enterprise Server VisiBroker は，PortableInterceptorExt という新しいモジュールを追加することによってポータブルインタセプタのパブリック拡張機能である「POA スコープサーバリクエストインタセプタ」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，POA スコープサーバリクエストインタセプタをインストールするための拡張メソッドを持つ IORInfoExt というローカルインタフェースがあります。

コードサンプル 19-23　IORInfoExt クラス（C++）

```
#include "PortableInterceptorExt_c.hh"

class IORInfoExt:public PortableInterceptor::IORInfo
{
   public:
      virtual void add_server_request_interceptor(
         ServerRequestInterceptor_ptr _interceptor)=0;
      virtual char*full_poa_name();
};
```

コードサンプル 19-24　IORInfoExt インタフェース（Java）

```
package com.inprise.vbroker.PortableInterceptor;

public interface IORInfoExt extends
   org.omg.CORBA.LocalInterface,
   org.omg.PortableInterceptor.IORInfo,
   com.inprise.vbroker.PortableInterceptor.IORInfoExtOperations,
   org.omg.CORBA.portable.IDLEntity
```

```
{
   public void add_server_request_interceptor
      (ServerRequestInterceptor interceptor)
      throws DuplicateName;
   public java.lang.String[ ] full_poa_name();
}
```

19.3　サンプル

この節では，ポータブルインタセプタを使用するための実際のアプリケーションの書き方と，各リクエストインタセプタのインプリメント方法について説明します。それぞれのサンプルはクライアントアプリケーションとサーバアプリケーションのセット，および C++と Java で記述されたそれぞれのインタセプタで構成されています。各インタフェースの定義については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「ポータブルインタセプタインタフェースとクラス」の記述を参照してください。

また，ポータブルインタセプタを使用する開発者には，CORBA の仕様のポータブルインタセプタについての記述を読むことをお勧めします。

19.3.1　サンプルコード

$VBROKERDIR/examples/vbe/pi ディレクトリ（UNIX）または%VBROKERDIR%¥examples¥vbe¥pi ディレクトリ（Windows）内にあるサンプルのリストを次に示します。それぞれのサンプルは，そのサンプルの目的をわかりやすくするためにディレクトリ名に対応しています。

- client_server
- chaining

次にコンパイル手順およびその実行（または配置）について，client_server をサンプルにして詳細に説明します。

19.3.2　サンプル：client_server

(1)　サンプルの目的

このサンプルでは，コードを変更しないで，既存の CORBA アプリケーションにポータブルインタセプタを追加する簡単な方法を実際に説明します。ポータブルインタセプタはクライアント側およびサーバ側の両方のどのアプリケーションにも追加できます。これは，ランタイム時に設定する指定のオプションやプロパティで，関連するアプリケーションを再実行することによって行えます。

使用するクライアントおよびサーバアプリケーションは，$VBROKERDIR/examples/vbe/basic/bank_agent（UNIX）または%VBROKERDIR%¥examples¥vbe¥basic¥bank_agent（Windows）にあるアプリケーションと類似しています。サンプル全体は抜粋であり，ポータブルインタセプタはランタイム構成時に追加されています。その理由は，VisiBroker のインタセプタを熟知している開発者に VisiBroker のインタセプタと OMG 固有ポータブルインタセプタ間のコーディングを早く行える方法を提供するためです。

(2)　コードの説明

(a)　必須パッケージのインポート

ポータブルインタセプタインタフェースを使用するには，関連パッケージまたはヘッダファイルが組み込まれている必要があります。Java ORB で，DuplicateName や InvalidName のようなポータブルインタセプタの例外を使用している場合，org.omg.PortableInterceptor.ORBInitInfoPackage はオプションであることに注意してください。

コードサンプル 19-25　ポータブルインタセプタを使用する際の必須ヘッダファイル（C++）

```
#include "PortableInterceptor_c.hh"
#include "IOP_c.hh"
```

コードサンプル 19-26　ポータブルインタセプタを使用する際の必須パッケージ（Java）

```
import org.omg.PortableInterceptor.*;
import org.omg.PortableInterceptor.ORBInitInfoPackage.*;
```

まずクライアント側で，リクエストインタセプタの各部分の書き方について詳細に説明します。クライアント側のリクエストインタセプタをロードするには，インタフェースをインプリメントするクラスである ORBInitializer をインプリメントする必要があります。また，これは初期化を考慮するとサーバ側のリクエストインタセプタにも該当します。

次にこれをインプリメントするコードを示します。

コードサンプル 19-27　クライアントリクエストインタセプタをロードするための ORBInitializer の正しい継承

C++の場合

```
class SampleClientLoader :
   public PortableInterceptor::ORBInitializer
```

Java の場合

```
public class SampleClientLoader extends
   org.omg.CORBA.LocalObject implements ORBInitializer
```

コードサンプル 19-28　サーバリクエストインタセプタをロードするための ORBInitializer の正しい継承

C++の場合

```
class SampleServerLoader :
   public PortableInterceptor::ORBInitializer
```

Java の場合

```
public class SampleServerLoader extends
   org.omg.CORBA.LocalObject implements ORBInitializer
```

インタフェースをインプリメントするオブジェクトである ORBInitializer それぞれも LocalObject オブジェクトから継承される必要があることに注意してください。これが必要である理由は，ORBInitializer の IDL 定義がキーワードである local を使用するためです。IDL キーワード local の詳細については，「25.　valuetype の使用」を参照してください。

ORB の初期化時に，各リクエストインタセプタは pre_init()インタフェースのインプリメンテーションによって追加されています。このインタフェース内では，クライアントリクエストインタセプタは，add_client_request_interceptor()メソッドによって追加されています。関連するクライアントリクエストインタセプタ自身を ORB に追加する前に，そのインタセプタが実体化される必要があります。

コードサンプル 19-29　クライアント側リクエストインタセプタの初期化および ORB への登録

C++の場合

```
public:
   void pre_init(PortableInterceptor::ORBInitInfo_ptr
    _info){
      SampleClientInterceptor *interceptor =
                                 new SampleClientInterceptor;
         try {
            _info->add_client_request_interceptor(interceptor);
            . . .
```

Java の場合

```
public void pre_init(ORBInitInfo info){
   try {
      info.add_client_request_interceptor(new
```

```
                                        SampleClientInterceptor());
        . . .
```

OMG 規格に従って，必須アプリケーションは register_orb_initializer メソッドによってそれぞれのインタセプタを登録します。詳細については「19.3.2 (2) (f)　クライアントおよびサーバアプリケーションの開発」を参照してください。VisiBroker には，このようなインタセプタの別のオプションの登録方法，(DLL による登録など) があります。この登録方法を使用すると，アプリケーションはコードをまったく変更する必要がなく，その実行方法だけを変更すれば済むという利点があります。実行時の拡張オプションによって，インタセプタが登録され実行されます。オプションは VisiBroker 4.x インタセプタである vbroker.orb.dynamicLibs=<DLL filename>と同様です。ここで，<DLL filename>はダイナミックリンクライブラリのファイル名であり，拡張子は，UNIX の場合は.so，.sl，.a などで，Windows の場合は.dll です。一つ以上の DLL ファイルをロードするには，次のように各ファイル名をコンマ（,）で区切ります。

- `vbroker.orb.dynamicLibs=a.so,b.so,c.so`（UNIX）
- `vbroker.orb.dynamicLibs=a.dll,b.dll,c.dll`（Windows）

動的にインタセプタをロードするには，VISInit インタフェースが使用されます。これは VisiBroker 4.x インタセプタで使用されるものと同様です。詳細については「20.　VisiBroker 4.x インタセプタの使用」を参照してください。ORB_init のインプリメンテーションでは，各インタセプタローダの登録は同様に行われます。

コードサンプル 19-30　DLL のロードによるクライアント側の ORBInitializer の登録（C++）

```
void ORB_init(int& argc,char* const* argv,CORBA::ORB_ptr orb)
{
   if(_bind_interceptors_installed)return;

   SampleClientLoader *client =new SampleClientLoader();
   PortableInterceptor::register_orb_initializer(client);

   . . .
```

クライアント側インタセプタローダの完全なインプリメンテーションを次に示します。

コードサンプル 19-31　クライアント側インタセプタローダの完全なインプリメンテーション（C++）

```
//SampleClientLoader.C

#include "PortableInterceptor_c.hh"
#include "IOP_c.hh"

#include "SampleClientInterceptor.h"

#if !defined(DLL_COMPILE )
#include "vinit.h"
#include "corba.h"
#endif

//USE_STD_NS is a define setup by VisiBroker to use the std namespace
USE_STD_NS

class SampleClientLoader :
   public PortableInterceptor::ORBInitializer
{
private:
   short int _interceptors_installed;

   #if defined(DLL_COMPILE )
   static SampleClientLoader _instance;
   #endif

public:
   SampleClientLoader(){
      _interceptors_installed =0;
```

```
    }

    void pre_init(PortableInterceptor::ORBInitInfo_ptr _info){
       if(_interceptors_installed)return;

       cout <<"=====>SampleClientLoader:Installing ..."<<endl;

       SampleClientInterceptor *interceptor =
                                  new SampleClientInterceptor;

       try {
             _info->add_client_request_interceptor(interceptor);

             _interceptors_installed =1;
             cout <<"=====>SampleClientLoader:Interceptors loaded."
                  <<endl;
       }
       catch(PortableInterceptor::ORBInitInfo::DuplicateName &e){
          cout <<"=====>SampleClientLoader:"
               <<e.name <<"already installed!"<<endl;
       }
       catch(...){
          cout <<"=====>SampleClientLoader:other exception occurred!"
               <<endl;
          }
       }

       void post_init(PortableInterceptor::ORBInitInfo_ptr _info){
    }
};

#if defined(DLL_COMPILE )

class VisiClientLoader :VISInit

{
private:
       static VisiClientLoader _instance;
       short int _bind_interceptors_installed;

public:
    VisiClientLoader():VISInit(1){
    _bind_interceptors_installed =0;
    }

    void ORB_init(int&argc,char*const*argv,CORBA::ORB_ptr orb){
       if(_bind_interceptors_installed)return;

       try {
             SampleClientLoader *client =new SampleClientLoader();
             PortableInterceptor::register_orb_initializer(client);

             _bind_interceptors_installed =1;
       }

       catch(const CORBA::Exception&e)
       {
          cerr <<e <<endl;
       }
    }
};

//static instance
VisiClientLoader VisiClientLoader::_instance;

#endif
```

コードサンプル 19-32　クライアント側インタセプタローダの完全なインプリメンテーション（Java）：SampleClientLoader.java

```
//SampleClientLoader.java

import org.omg.PortableInterceptor.*;
import org.omg.PortableInterceptor.ORBInitInfoPackage.*;
```

```
public class SampleClientLoader extends org.omg.CORBA.LocalObject
implements ORBInitializer
{
   public void pre_init(ORBInitInfo info){
      try {
         System.out.println("=====>SampleClientLoader:
                                                  Installing ...");
         info.add_client_request_interceptor
                                     (new SampleClientInterceptor());
         System.out.println("=====>SampleClientLoader:Interceptors
                                                        loaded.");
      }
      catch(DuplicateName dn){
         System.out.println("=====>SampleClientLoader:"+dn.name
                                               +"already installed.");
      }
      catch(Exception e){
         e.printStackTrace();
         throw new org.omg.CORBA.INITIALIZE(e.toString());
     }
   }

      public void post_init(ORBInitInfo info){
         //We do not do anything here.
      }
}
```

(b)　サーバ側インタセプタでの ORBInitializer のインプリメント

この段階では，クライアントリクエストインタセプタはすでに正しく実体化され，追加されているはずです。これ以降のコードサンプルは例外処理と結果表示だけを提供します。同様に，サーバ側のサーバリクエストインタセプタにも同じことが行われます。ただし，add_server_request_interceptor()メソッドを使用して関連サーバリクエストインタセプタを ORB に追加することが異なります。

コードサンプル 19-33　サーバ側リクエストインタセプタの初期化および ORB への登録

C++の場合

```
public:
   void pre_init(PortableInterceptor::ORBInitInfo_ptr
    _info){
      SampleServerInterceptor *interceptor =
                                  new SampleServerInterceptor;
      try {
         _info->add_server_request_interceptor(interceptor);
         . . .
```

Java の場合

```
public void pre_init(ORBInitInfo info){
   try {
      info.add_server_request_interceptor(
                                new SampleServerInterceptor());
      . . .
```

これは，サーバ側 ORBInitializer クラスの DLL インプリメンテーションによるロードにも同様に適用されます。

コードサンプル 19-34　DLL によってロードされるサーバ側リクエスト ORBInitializer（C++）

```
void ORB_init(int& argc,char* const* argv,CORBA::ORB_ptr orb)
{
   if(_poa_interceptors_installed)return;
   SampleServerLoader *server =new SampleServerLoader();
   PortableInterceptor::register_orb_initializer(server);
   . . .
```

サーバ側インタセプタローダの完全なインプリメンテーションを次に示します。

コードサンプル 19-35 サーバ側インタセプタローダの完全なインプリメンテーション（C++）

```
//SampleServerLoader.C

#include "PortableInterceptor_c.hh"
#include "IOP_c.hh"

#if defined(DLL_COMPILE )
#include "vinit.h"
#include "corba.h"
#endif

#include "SampleServerInterceptor.h"

//USE_STD_NS is a define setup by VisiBroker to use the std namespace
USE_STD_NS

class SampleServerLoader :
   public PortableInterceptor::ORBInitializer
{
private:
   short int _interceptors_installed;

public:
   SampleServerLoader(){
      _interceptors_installed =0;
   }

   void pre_init(PortableInterceptor::ORBInitInfo_ptr _info){
       if(_interceptors_installed)return;

       cout <<"=====>SampleServerLoader:Installing ..."<<endl;

       SampleServerInterceptor *interceptor =
                              new SampleServerInterceptor();

       try {
         _info->add_server_request_interceptor(interceptor);

         _interceptors_installed =1;
         cout <<"=====>SampleServerLoader:Interceptors loaded."
              <<endl;
       }
       catch(PortableInterceptor::ORBInitInfo::DuplicateName &e){
         cout <<"=====>SampleServerLoader:"
           <<e.name <<"already installed!"<<endl;
       }
       catch(...){
         cout <<"=====>SampleServerLoader:other exception occurred!"
           <<endl;
     }
   }
void post_init(PortableInterceptor::ORBInitInfo_ptr _info){}
};


#if defined(DLL_COMPILE )
class VisiServerLoader :VISInit
{
private:
   static VisiServerLoader _instance;
   short int _poa_interceptors_installed;

public:
   VisiServerLoader():VISInit(1){
   _poa_interceptors_installed =0;
}
void ORB_init(int& argc,char* const* argv,CORBA::ORB_ptr orb){
      if(_poa_interceptors_installed)return;
      try {
            SampleServerLoader *server =new SampleServerLoader();
            PortableInterceptor::register_orb_initializer(server);
            _poa_interceptors_installed =1;
      }
```

```
   catch(const CORBA::Exception&e)
      {
      cerr <<e <<endl;
      }
   }
};
//static instance
VisiServerLoader VisiServerLoader::_instance;

#endif
```

コードサンプル 19-36　サーバ側インタセプタローダの完全なインプリメンテーション（Java）：SampleServerLoader.java

```
//SampleServerLoader.java

import org.omg.PortableInterceptor.*;
import org.omg.PortableInterceptor.ORBInitInfoPackage.*;

public class SampleServerLoader extends org.omg.CORBA.LocalObject
implements ORBInitializer

{
   public void pre_init(ORBInitInfo info){
      try {
         info.add_server_request_interceptor(new
                  SampleServerInterceptor());
         System.out.println("=====>SampleServerLoader:
                  Interceptors loaded");
      }
      catch(DuplicateName dn){
         System.out.println("Interceptor:
                           "+dn.name +"already installed.");
      }
      catch(Exception e){
         e.printStackTrace();
         throw new org.omg.CORBA.INITIALIZE(e.toString());
      }
   }
   public void post_init(ORBInitInfo info){
      //We do not do anything here.
   }
}
```

(c)　クライアント側およびサーバ側リクエストインタセプタでの RequestInterceptor のインプリメント

クライアント側またはサーバ側のリクエストインタセプタのインプリメンテーション時に，その両方に共通の別の二つのインタフェースをインプリメントする必要があります。それは，name()と destroy()です。リクエストや応答でロードおよび呼び出しを行うインタセプタを正しく識別するために ORB に名前を提供するので，name()は重要です。CORBA の仕様に従うと，インタセプタはアノニマスでもかまいません。つまり，名前属性として空の文字列でもかまいません。このサンプルでは，SampleClientInterceptor という名前がクライアント側インタセプタに，SampleServerInterceptor という名前がサーバ側インタセプタに割り当てられています。

コードサンプル 19-37　インタフェース属性，read-only 属性名のインプリメンテーション

C++の場合

```
   public:
      char *name(void){
         return _name;
   }
```

Java の場合

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

(d)　クライアントでの ClientRequestInterceptor のインプリメント

クライアントリクエストインタセプタでは，リクエストインタセプタが正しく動作するために ClientRequestInterceptor インタフェースをインプリメントする必要があります。クラスがインタフェースをインプリメントする場合，インプリメンテーションに関係なく五つのリクエストインタセプタメソッドがインプリメントされます。これには send_request()メソッド，send_poll()メソッド，receive_reply()メソッド，receive_exception()メソッド，および receive_other()メソッドがあります。さらに，事前にリクエストインタセプタのインタフェースをインプリメントしておくことが必要です。クライアント側のインタセプタでは，そのイベントに関して次のリクエストインタセプトポイントが起動されます。

- send_request

  インタセプトポイントを提供し，インタセプタがリクエスト情報を照会して，リクエストがサーバに送信される前にサービスコンテキストを修正できるようにします。

コードサンプル 19-38　public void send_request(ClientRequestInfo ri)インタフェースのインプリメンテーション

C++の場合

```
void send_request(PortableInterceptor::ClientRequestInfo_ptr ri){
   ・・・
```

Java の場合

```
public void send_request(ClientRequestInfo ri)
                                          throws ForwardRequest {
   ・・・
```

コードサンプル 19-39　void send_poll(ClientRequestInfo ri)インタフェースのインプリメンテーション

- send_poll

  インタセプトポイントを提供し，TII（時間非依存呼び出し）ポーリング取得応答シーケンス時に情報をインタセプタが照会できるようにします。

C++の場合

```
void send_poll(PortableInterceptor::ClientRequestInfo_ptr ri){
   ・・・
```

Java の場合

```
public void send_poll(ClientRequestInfo ri){
   ・・・
```

コードサンプル 19-40　void receive_reply(ClientRequestInfo ri)インタフェースのインプリメンテーション

- receive_reply

  インタセプトポイントを提供し，応答がサーバから戻されてから，制御がクライアントに戻る前にインタセプタがその応答に関する情報を照会できるようにします。

C++の場合

```
void receive_reply(PortableInterceptor::ClientRequestInfo_ptr
                                                        ri){
   ・・・
```

Java の場合

```
public void receive_reply(ClientRequestInfo ri){
   ・・・
```

コードサンプル 19-41　void receive_exception(ClientRequestInfo ri)インタフェースのインプリメンテーション

- receive_exception

インタセプトポイントを提供し，例外がクライアントに発生する前にインタセプタがその例外の情報を照会できるようにします。

C++の場合

```
void receive_exception
              (PortableInterceptor::ClientRequestInfo_ptr ri){
   . . .
```

Java の場合

```
public void receive_exception(ClientRequestInfo ri)
                                        throws ForwardRequest {
   . . .
```

- receive_other

  インタセプトポイントを提供し，リクエスト結果が正常応答または例外以外の結果になった場合に，インタセプタが利用可能な情報を照会できるようにします。例えば，リクエストの結果がリトライ（例：　LOCATION_FORWARD ステータスの GIOP 応答を受信）になる場合です。非同期呼び出しの場合は，リクエストのあとにすぐに応答は行われませんが，制御はクライアントに戻り，終了インタセプトポイントが呼び出されます。

コードサンプル 19-42　void receive_other(ClientRequestInfo ri)インタフェースのインプリメンテーション

C++の場合

```
void receive_other(PortableInterceptor::ClientRequestInfo_ptr
                                                        ri){
   . . .
```

Java の場合

```
public void receive_other(ClientRequestInfo ri)
                                      throws ForwardRequest {
   . . .
```

クライアント側リクエストインタセプタの完全なインプリメンテーションを次に示します。

コードサンプル 19-43　クライアント側リクエストインタセプタの完全なインプリメンテーション (C++)

```
//SampleClientInterceptor.h

#include "PortableInterceptor_c.hh"
#include "IOP_c.hh"

//USE_STD_NS is a define setup by VisiBroker to use the std namespace
USE_STD_NS

class SampleClientInterceptor :
   public PortableInterceptor::ClientRequestInterceptor
{
private:
   char *_name;

   void init(char *name){
      _name =new char [strlen(name)+1 ];
      strcpy(_name,name);
   }

public:
   SampleClientInterceptor(char *name){
      init(name);
   }

   SampleClientInterceptor(){
      init("SampleClientInterceptor");
   }

char *name(void){
   return _name;
}
```

```
void destroy(void){
   //do nothing here

   cout <<"=====>SampleServerLoader:Interceptors unloaded"<<endl;
}

/**
* This is similar to VisiBroker 4.x ClientRequestInterceptor,
*
* void preinvoke_premarshal(CORBA::Object_ptr target,
*                           const char*operation,
*                           IOP::ServiceContextList&servicecontexts,
*                           VISClosure&closure)=0;
*/
void send_request(PortableInterceptor::ClientRequestInfo_ptr ri){
   cout <<"=====>SampleClientInterceptor id "<<ri->request_id()
      <<"send_request =>"<<ri->operation()
      <<":Target ="<<ri->target()
      <<endl;
}

/**
* There is no equivalent interface for VisiBroker 4.x
* ClientRequestInterceptor.
*/

void send_poll(PortableInterceptor::ClientRequestInfo_ptr ri){
   cout <<"=====>SampleClientInterceptor id "<<ri->request_id()
      <<"send_poll =>"<<ri->operation()
      <<":Target ="<<ri->target()
      <<endl;
}

/**
* This is similar to VisiBroker 4.x ClientRequestInterceptor,
*
* void postinvoke(CORBA::Object_ptr target,
*                 const IOP::ServiceContextList&service_contexts,
*                 CORBA_MarshalInBuffer&payload,
*                 CORBA::Environment_ptr env,
*                 VISClosure&closure)=0;
*
* with env not holding any exception value.
*/
void receive_reply(PortableInterceptor::ClientRequestInfo_ptr ri){
   cout <<"=====>SampleClientInterceptor id "<<ri->request_id()
      <<"receive_reply =>"<<ri->operation()
      <<endl;
}

/**
* This is similar to VisiBroker 4.x ClientRequestInterceptor,
*
* void postinvoke(CORBA::Object_ptr target,
*                 const IOP::ServiceContextList&service_contexts,
*                 CORBA_MarshalInBuffer&payload,
*                 CORBA::Environment_ptr env,
*                 VISClosure&closure)=0;
*
* with env holding the exception value.
*/
void receive_exception(PortableInterceptor::ClientRequestInfo_ptr
                                                              ri){
   cout <<"=====>SampleClientInterceptor id "<<ri->request_id()
      <<"receive_exception =>"<<ri->operation()
      <<":Exception ="<<ri->received_exception()
      <<endl;
}

/**
* This is similar to VisiBroker 4.x ClientRequestInterceptor,
*
* void postinvoke(CORBA::Object_ptr target,
```

```
*                const IOP::ServiceContextList&service_contexts,
*                CORBA_MarshalInBuffer&payload,
*                CORBA::Environment_ptr env,
*                VISClosure&closure)=0;
*
* with env holding the exception value.
*/
void receive_other(PortableInterceptor::ClientRequestInfo_ptr ri){
   cout <<"=====>SampleClientInterceptor id "<<ri->request_id()
      <<"receive_other =>"<<ri->operation()
      <<":Exception ="<<ri->received_exception()
      <<",Reply Status ="<<
     getReplyStatus(ri->reply_status())
     <<endl;
   }

protected:
   char *getReplyStatus(CORBA::Short status){
      if(status ==PortableInterceptor::SUCCESSFUL)
            return "SUCCESSFUL";
      else if(status ==PortableInterceptor::SYSTEM_EXCEPTION)
            return "SYSTEM_EXCEPTION";
      else if(status ==PortableInterceptor::USER_EXCEPTION)
            return "USER_EXCEPTION";
      else if(status ==PortableInterceptor::LOCATION_FORWARD)
            return "LOCATION_FORWARD";
      else if(status ==PortableInterceptor::TRANSPORT_RETRY)
            return "TRANSPORT_RETRY";
      else
            return "invalid reply status id";
      }
};
```

コードサンプル 19-44　クライアント側リクエストインタセプタの完全なインプリメンテーション (Java)：SampleClientInterceptor.java

```
//SampleClientInterceptor.java

import org.omg.PortableInterceptor.*;
import org.omg.Dynamic.*;

public class SampleClientInterceptor extends org.omg.CORBA.LocalObject
implements ClientRequestInterceptor {

   public SampleClientInterceptor(){
      this("SampleClientInterceptor");
}

public SampleClientInterceptor(String name){
_name =name;
}
private String _name =null;
/**
* InterceptorOperations implementation
*/
public String name(){
   return _name;
}

public void destroy(){
   System.out.println("=====>SampleServerLoader:Interceptors unloaded");
}

/**
* ClientRequestInterceptor implementation
*/

/**
* This is similar to VisiBroker 4.x ClientRequestInterceptor,
*
* public void preinvoke_premarshal(org.omg.CORBA.Object target,
      String operation,
*   ServiceContextListHolder service_contexts_holder,Closure
    closure);
```

```
*/

public void send_request(ClientRequestInfo ri)throws ForwardRequest {
   System.out.println("=====>SampleClientInterceptor id "+
      ri.request_id()+
           "send_request =>"+ri.operation()+
           ":target ="+ri.target());
}

/**
* There is no equivalent interface for VisiBroker 4.x
* ClientRequestInterceptor.
*/

public void send_poll(ClientRequestInfo ri){
   System.out.println("=====>SampleClientInterceptor id "+
      ri.request_id()+
           "send_poll => " + ri.operation() +
           " : target = " + ri.target());
}

/**
* This is similar to VisiBroker 4.x ClientRequestInterceptor,
*
* public void postinvoke(org.omg.CORBA.Object target,
*   ServiceContext[ ] service_contexts,InputStream payload,
*   org.omg.CORBA.Environment env,Closure closure);
*
* with env not holding any exception value.
*/
public void receive_reply(ClientRequestInfo ri){
   System.out.println("=====>SampleClientInterceptor id " +
     ri.request_id() +
           " receive_reply => " + ri.operation());
}

/**
* This is similar to VisiBroker 4.x ClientRequestInterceptor,
*
* public void postinvoke(org.omg.CORBA.Object target,
*   ServiceContext[ ] service_contexts,InputStream payload,
*   org.omg.CORBA.Environment env,Closure closure);
*
* with env holding the exception value.
*/
public void receive_exception(ClientRequestInfo ri)throws ForwardRequest {
   System.out.println("=====>SampleClientInterceptor id "+
      ri.request_id() +
            " receive_exception => " + ri.operation() +
            ": exception = " + ri.received_exception());
}
/**
* This is similar to VisiBroker 4.x ClientRequestInterceptor,
*
* public void postinvoke(org.omg.CORBA.Object target,
*   ServiceContext[ ] service_contexts,InputStream payload,
*   org.omg.CORBA.Environment env,Closure closure);
*
* with env holding the exception value.
*/
public void receive_other(ClientRequestInfo ri) throws ForwardRequest {
   System.out.println("=====> SampleClientInterceptor id "+
      ri.request_id() +
           " receive_reply => " + ri.operation() +
           ": exception = " + ri.received_exception() +
           ", reply status = " + getReplyStatus(ri));
}
protected String getReplyStatus(RequestInfo ri){
   switch(ri.reply_status()){
      case SUCCESSFUL.value:
         return "SUCCESSFUL";
      case SYSTEM_EXCEPTION.value:
         return "SYSTEM_EXCEPTION";
      case USER_EXCEPTION.value:
```

```
            return "USER_EXCEPTION";
        case LOCATION_FORWARD.value:
            return "LOCATION_FORWARD";
        case TRANSPORT_RETRY.value:
            return "TRANSPORT_RETRY";
        default:
            return "invalid reply status id";
        }
    }
}
```

(e)　サーバでの ServerRequestInterceptor のインプリメント

サーバリクエストインタセプタでは，リクエストインタセプタが正しく動作するために ServerRequestInterceptor インタフェースをインプリメントする必要があります。サーバ側インタセプタでは，各イベントに関して次のリクエストインタセプトポイントが起動されます。

- receive_request_service_contexts

  インタセプトポイントを提供し，インタセプタが入力リクエストからサービスコンテキスト情報を取得し，それを PortableInterceptor::Current のスロットに転送できるようにします。このインタセプトポイントはサーバントマネージャの前に呼び出されます。

コードサンプル 19-45　void receive_request_service_contexts (ServerRequestInfo ri)インタフェースのインプリメンテーション

C++の場合

```
void receive_request_service_contexts
                (PortableInterceptor::ServerRequestInfo_ptr
                                                        ri){
  . . .
```

Java の場合

```
public void receive_request_service_contexts
       (ServerRequestInfo ri) throws ForwardRequest{
  . . .
```

- receive_request

  インタセプトポイントを提供し，使用できるすべての情報（オペレーションパラメタなど）をインタセプタが照会できるようにします。

コードサンプル 19-46　void receive_request (ServerRequestInfo ri)インタフェースのインプリメンテーション

C++の場合

```
void receive_request(PortableInterceptor::ServerRequestInfo_ptr
                                                      ri){
  . . .
```

Java の場合

```
public void receive_request(ServerRequestInfo ri)
                                        throws ForwardRequest {
  . . .
```

- send_reply

  インタセプトポイントを提供し，ターゲットオペレーションが呼び出されたあと，応答がクライアントに戻される前にインタセプタが応答情報を照会し，応答サービスコンテキストを修正できるようにします。

コードサンプル 19-47 void send_reply (ServerRequestInfo ri)インタフェースのインプリメンテーション

C++の場合

```
void send_reply(PortableInterceptor::ServerRequestInfo_ptr
                                                    ri){
  . . .
```

Java の場合

```
public void send_reply(ServerRequestInfo ri){
  . . .
```

- send_exception

  インタセプトポイントを提供し，例外がクライアントで発生する前にインタセプタが例外情報を照会し，応答サービスコンテキストを修正できるようにします。

コードサンプル 19-48 void send_exception (ServerRequestInfo ri)インタフェースのインプリメンテーション

C++の場合

```
void send_exception(PortableInterceptor::ServerRequestInfo_ptr
                                                      ri) {
  . . .
```

Java の場合

```
public void send_exception(ServerRequestInfo ri)throws
                                            ForwardRequest {
  . . .
```

- send_other

  インタセプトポイントを提供し，リクエスト結果が正常応答または例外以外の結果になった場合に，インタセプタが利用可能な情報を照会できるようにします。例えば，リクエストの結果がリトライ（例: LOCATION_FORWARD ステータスの GIOP 応答を受信）する場合です。非同期呼び出しの場合は，リクエストのあとにすぐに応答は行われませんが，制御はクライアントに戻り，終了インタセプトポイントが呼び出されます。

コードサンプル 19-49 void send_other (ServerRequestInfo ri)インタフェースのインプリメンテーション

C++の場合

```
void send_other(PortableInterceptor::ServerRequestInfo_ptr
                                                    ri){
  . . .
```

Java の場合

```
public void send_other(ServerRequestInfo ri)throws
                                     ForwardRequest {
  . . .
```

すべてのインタセプトポイントによって，クライアントおよびサーバは異なる種類の情報を呼び出しの異なるポイントで取得できます。サンプルでは，このような情報はデバッグの形式で画面に表示されています。

サーバ側リクエストインタセプタの完全なインプリメンテーションを次に示します。

コードサンプル 19-50 サーバ側リクエストインタセプタの完全なインプリメンテーション（C++）：SampleServerInterceptor.cpp

```
// SampleServerInterceptor.h

#include "PortableInterceptor_c.hh"
#include "IOP_c.hh"
```

```
//USE_STD_NS is a define setup by VisiBroker to use the std namespace
USE_STD_NS

class SampleServerInterceptor :
   public PortableInterceptor::ServerRequestInterceptor
{
private:
   char *_name;

   void init(char *name){
      _name =new char [strlen(name)+1 ];
      strcpy(_name,name);
}

public:
   SampleServerInterceptor(char *name){
      init(name);
   }

   SampleServerInterceptor(){
      init("SampleServerInterceptor");
   }

   char *name(void){
      return _name;
}

   void destroy(void){
      //do nothing here
      cout <<"=====>SampleServerLoader:Interceptors unloaded"<<endl;
   }

/**
* This is similar to VisiBroker 4.x ClientRequestInterceptor,
*
* void preinvoke_premarshal(CORBA::Object_ptr target,
*                           const char*operation,
*                           IOP::ServiceContextList&servicecontexts,
*                           VISClosure&closure)=0;
*/
void
receive_request_service_contexts(PortableInterceptor::
   ServerRequestInfo_ptr ri){
         cout <<"=====>SampleServerInterceptor id "<<ri->request_id()
         <<"receive_request_service_contexts =>"<<ri->operation()
         <<endl;
}

/**
* There is no equivalent interface for VisiBroker 4.x
* SeverRequestInterceptor.
*/
void receive_request(PortableInterceptor::ServerRequestInfo_ptr
                                                              ri)
{
   cout <<"=====>SampleServerInterceptor id "<<ri->request_id()
      <<"receive_request =>"<<ri->operation()
      <<":Object ID ="<<ri->object_id()
      <<",Adapter ID ="<<ri->adapter_id()
      <<endl;
}

/**
* There is no equivalent interface for VisiBroker 4.x
* SeverRequestInterceptor.
*/
void send_reply(PortableInterceptor::ServerRequestInfo_ptr ri){
   cout <<"=====>SampleServerInterceptor id "<<ri->request_id()
      <<"send_reply =>"<<ri->operation()
      <<endl;
}

/**
```

```
* This is similar to VisiBroker 4.x ServerRequestInterceptor,
*
* virtual void postinvoke_premarshal(CORBA::Object_ptr _target,
*                      IOP::ServiceContextList&_service_contexts,
*                                    CORBA::Environment_ptr _env,
*                                    VISClosure&_closure)=0;
*
*with env holding the exception value.
*/
void send_exception(PortableInterceptor::ServerRequestInfo_ptr
                                                              ri){
   cout << "=====> SampleServerInterceptor id " << ri->request_id()
        << " send_exception => " << ri ->operation()
        << " :Exception = " << ri ->sending_exception()
        << " ,Reply status = " << getReplyStatus(ri->reply_status())
        <<endl;
}

/**
* This is similar to VisiBroker 4.x ServerRequestInterceptor,
*
* virtual void postinvoke_premarshal(CORBA::Object_ptr _target,
*                                    IOP::ServiceContextList&_service_contexts,
*                                    CORBA::Environment_ptr _env,
*                                    VISClosure&_closure)=0;
*
* with env holding the exception value.
*/
void send_other(PortableInterceptor::ServerRequestInfo_ptr ri){
   cout <<"=====>SampleServerInterceptor id "<<ri->request_id()
      <<"send_other =>"<<ri->operation()
      <<":Exception ="<<ri->sending_exception()
      <<",Reply Status ="<<getReplyStatus(ri->reply_status())
      <<endl;
}

protected:
   char *getReplyStatus(CORBA::Short status){
      if(status ==PortableInterceptor::SUCCESSFUL)
         return "SUCCESSFUL";
            else if(status ==PortableInterceptor::SYSTEM_EXCEPTION)
         return "SYSTEM_EXCEPTION";
            else if(status ==PortableInterceptor::USER_EXCEPTION)
         return "USER_EXCEPTION";
            else if(status ==PortableInterceptor::LOCATION_FORWARD)
         return "LOCATION_FORWARD";
            else if(status ==PortableInterceptor::TRANSPORT_RETRY)
         return "TRANSPORT_RETRY";
            else
         return "invalid reply status id";
      }
};
```

コードサンプル 19-51　サーバ側リクエストインタセプタの完全なインプリメンテーション（Java）：SampleServerInterceptor.java

```
// SampleServerInterceptor.java

import org.omg.PortableInterceptor.*;
import org.omg.Dynamic.*;
import java.io.PrintStream;

public class SampleServerInterceptor extends org.omg.CORBA.LocalObject
implements ServerRequestInterceptor {

   private String _name =null;

   public SampleServerInterceptor(){
      this("SampleServerInterceptor");
   }

   public SampleServerInterceptor(String name){
      _name =name;
   }
```

```
/**
* InterceptorOperations implementation
*/
public String name() {
      return _name;
}

public void destroy(){
   System.out.println("=====>SampleServerLoader:Interceptors unloaded");
}

/**
* ServerRequestInterceptor implementation
*/

/**
* This is similar to VisiBroker 4.x ServerRequestInterceptor,
*
* public void preinvoke(org.omg.CORBA.Object target,
*   String operation,
*   ServiceContext[ ] service__contexts,InputStream payload,
*   Closure closure);
*/

public void receive_request_service_contexts(ServerRequestInfo ri)
      throws ForwardRequest {
   System.out.println("=====>SampleServerInterceptor id " +
           ri.request_id() +
           " receive_request_service_contexts => " + ri.operation());
}

/**
* There is no equivalent interface for VisiBroker 4.x
* SeverRequestInterceptor.
*/
public void receive_request(ServerRequestInfo ri)
      throws ForwardRequest {
   System.out.println("=====>SampleServerInterceptor id "
            + ri.request_id() +
            " receive_request => " + ri.operation() +
            ": object id = " + ri.object_id() +
            ", adapter_id = " + ri.adapter_id());
}

/**
* There is no equivalent interface for VisiBroker 4.x
* SeverRequestInterceptor.
*/
public void send_reply(ServerRequestInfo ri){
   System.out.println("=====> SampleServerInterceptor id " +
            ri.request_id() +
            " send_reply => " + ri.operation());
}

/**
* This is similar to VisiBroker 4.x ServerRequestInterceptor,
*
* public void postinvoke_premarshal(org.omg.CORBA.Object target,
* ServiceContextListHolder service_contexts_holder,
* org.omg.CORBA.Environment env,Closure closure);
*
*with env holding the exception value.
*/
public void send_exception(ServerRequestInfo ri)
      throws ForwardRequest {
   System.out.println("=====>SampleServerInterceptor id " +
            ri.request_id() +
            " send_exception => " + ri.operation() +
            ": exception = " + ri.sending_exception() +
            ", reply status = " + getReplyStatus(ri));
}

/**
* This is similar to VisiBroker 4.x ServerRequestInterceptor,
```

```
*
* public void postinvoke_premarshal(org.omg.CORBA.Object target,
*   ServiceContextListHolder service_contexts_holder,
*   org.omg.CORBA.Environment env,Closure closure);
*
* with env holding the exception value.
*/
public void send_other(ServerRequestInfo ri)throws ForwardRequest {
      System.out.print("=====>SampleServerInterceptor id "+
            ri.request_id() +
            " send_other =>" + ri.operation() +
            ": exception = " + ri.sending_exception() +
            ", reply status = " + getReplyStatus(ri));
}
      protected String getReplyStatus(RequestInfo ri){
         switch(ri.reply_status()){
            case SUCCESSFUL.value:
               return "SUCCESSFUL";
            case SYSTEM_EXCEPTION.value:
               return "SYSTEM_EXCEPTION";
            case USER_EXCEPTION.value:
               return "USER_EXCEPTION";
            case LOCATION_FORWARD.value:
               return "LOCATION_FORWARD";
            case TRANSPORT_RETRY.value:
               return "TRANSPORT_RETRY";
            default:
               return "invalid reply status id";
         }
      }
}
```

(f)　クライアントおよびサーバアプリケーションの開発

インタセプタクラスを書き込んだあと，このクラスをそれぞれクライアントおよびサーバアプリケーションに登録する必要があります。

● C++の場合

C++では，クライアントとサーバは PortableInterceptor::register_orb_initializer(<class name>)メソッドによって，それぞれの ORBInitializer クラスを登録します。

ここで，<class name>は登録するクラスの名前です。サンプルでは，DLL（動的リンクライブラリ）としてインタセプタクラスを登録する別の方法も説明します。この方法の利点は，アプリケーションがコードを変更する必要がなく，実行方法だけを変更すればいいということです。「19.3.2（4）　クライアントおよびサーバアプリケーションの実行または配置」を参照してください。これは VisiBroker の適切な登録方法であることに注意してください。OMG に完全に準拠するには，次に示す方法は使用しないでください。

DLL としてインタセプタクラスをロードすること（すなわち，VisiBroker の適切なメソッドを使用すること）を選択したら，クライアントおよびサーバアプリケーションに拡張コードを追加する必要はありません。サンプルでは，DLL コンパイルとリンク付けが指定されていないと，コードの部分がマクロによる適切な状態ではないことがわかります。

クライアントアプリケーションの完全なインプリメンテーションを次に示します。

コードサンプル 19-52　クライアントアプリケーションの完全なインプリメンテーション（C++）

```
//Client.C

#include "Bank_c.hh"

//USE_STD_NS is a define setup by VisiBroker to use the std namespace
USE_STD_NS

#if !defined(DLL_COMPILE )
#include "SampleClientLoader.C"
```

```
#endif

int main(int argc, char* const* argv)
{
   try {
      // Instantiate the Loader *before* the orb initialization
      // if chose not to use DLL method of loading
      #if !defined(DLL_COMPILE )
      SampleClientLoader* loader = new SampleClientLoader;
      PortableInterceptor::register_orb_initializer(loader);
      #endif

      // Initialize the ORB.
      CORBA::ORB_var orb =CORBA::ORB_init(argc,argv);

      // Get the manager Id
      PortableServer::ObjectId_var managerId =
               PortableServer::string_to_ObjectId("BankManager");

      // Locate an account manager.Give the full POA name and the
      // servant ID.
      Bank::AccountManager_var manager =
              Bank::AccountManager::_bind("/bank_agent_poa"
                                                  ,managerId);

      // use argv [1 ] as the account name,,or a default.
      const char*name =argc >1 ?argv [1 ] ::"Jack B.Quick";

      // Request the account manager to open a named account.
      Bank::Account_var account =manager->open(name);

      // Get the balance of the account.
      CORBA::Float balance =account->balance();

      // Print out the balance.
      cout <<"The balance in "<<name <<"'s account is $"<<balance
           <<endl;
   }
   catch(const CORBA::Exception&e){
   cerr <<e <<endl;
   return 1;
   }
   return 0;
}
```

サーバアプリケーションの完全なインプリメンテーションを次に示します。

コードサンプル 19-53　サーバアプリケーションの完全なインプリメンテーション（C++）

```
// Server.C

#include "BankImpl.h"

// USE_STD_NS is a define setup by VisiBroker to use the std namespace
USE_STD_NS

#if !defined(DLL_COMPILE )
#include "SampleServerLoader.C"
#endif

int main(int argc,char*const*argv)
{
   try {
       // Instantiate an interceptor loader before initializing
       // the orb:
       #if !defined(DLL_COMPILE )
       SampleServerLoader*loader = new SampleServerLoader();
       PortableInterceptor::register_orb_initializer(loader);
       #endif

       // Initialize the ORB.
       CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);

       // get a reference to the rootPOA
```

```
        CORBA::Object_var obj = orb->
                                resolve_initial_references("RootPOA");
        PortableServer::POA_var rootPOA =
                                PortableServer::POA::_narrow(obj);

        CORBA::PolicyList policies;
        policies.length(1);
        policies [(CORBA::ULong)0 ] = rootPOA->
                  create_lifespan_policy( PortableServer::PERSISTENT );

        // get the POA Manager
        PortableServer::POAManager_var poa_manager =
                                              rootPOA->the_POAManager();

        // Create myPOA with the right policies
        PortableServer::POA_var myPOA =
                               rootPOA->create_POA("bank_agent_poa",
                                                   poa_manager,
                                                   policies);

        // Create the servant
        AccountManagerImpl managerServant;

        // Decide on the ID for the servant
        PortableServer::ObjectId_var managerId =
                    PortableServer::string_to_ObjectId("BankManager");

        // Activate the servant with the ID on myPOA
        myPOA->activate_object_with_id(managerId,&managerServant);

        // Activate the POA Manager
        poa_manager->activate();

        CORBA::Object_var reference =
        myPOA->servant_to_reference(&managerServant);
        cout <<reference <<"is ready"<<endl;

        // Wait for incoming requests
        orb->run();
    }
    catch(const CORBA::Exception&e){
       cerr <<e <<endl;
       return 1;
    }
    return 0;
}
```

● Java の場合

Java では，OMG 規格の register_orb_initializer のマッピングに従います。すなわち，このクラスを Java ORB プロパティによって登録します。サンプルでは，クライアントおよびサーバアプリケーションは実際には，プロパティファイルから，org.omg.PortableInterceptor.ORBInitializerClass.<Service>（<Service>は org.omg.PortableInterceptor.ORBInitializer をインプリメントするクラスの文字列名です）プロパティがある client.properties および server.properties を読み取ります。この場合は，SampleClientLoader および SampleServerLoader という二つのクラスです。

また，ファイルからプロパティを読み取らないで，アプリケーションを書き込むことを選択する場合，コマンドラインオプションを使用することもできます。これには，アプリケーションを次のように実行する必要があります。

```
vbj -Dorg.omg.PortableInterceptor.ORBInitializerClass.
    SampleClientLoader = SampleClientLoaderClient
vbj -Dorg.omg.PortableInterceptor.ORBInitializerClass.
    SampleServerLoader = SampleServerLoaderServer
```

クライアントアプリケーションの完全なインプリメンテーションを次に示します。

コードサンプル 19-54　クライアントアプリケーションの完全なインプリメンテーション（Java）:Client.java

```
// Client.java

import org.omg.PortableServer.*;

import java.util.Properties;
import java.io.FileInputStream;

public class Client {

   private static Properties property =null;

   public static void main(String [ ] args){
      try {
         property = new Properties();
         property.load(new FileInputStream("client.properties"));

      // Initialize the ORB.
      org.omg.CORBA.ORB orb=org.omg.CORBA.ORB.init(args,property);
      // Get the manager Id
      byte[ ] AccountManagerId="BankManager".getBytes();
      // Locate an account manager.Give the full POA name and
      // the servant ID.
      Bank.AccountManager manager =
         Bank.AccountManagerHelper.bind(orb,
            "/bank_client_server_poa", AccountManagerId);
      // use args [0 ]as the account name,or a default.
      String name =null;
      name =args.length >0 ?args [0 ] :"Jack B.Quick";
      // Request the account manager to open a named account.
      Bank.Account account =manager.open(name);
      // Get the balance of the account.
      float balance =account.balance();
      // Print out the balance.
      System.out.println("The balance in "+name +"'s account is $"+
         balance);
      }
      catch(Exception e){
      e.printStackTrace();
      }
  }
}
```

サーバアプリケーションの完全なインプリメンテーションを次に示します。

コードサンプル 19-55　サーバアプリケーションの完全なインプリメンテーション（Java）:Server.java

```
// Server.java

import org.omg.PortableServer.*;
import java.util.Properties;
import java.io.FileInputStream;

public class Server {

   private static Properties property = null;

   public static void main(String[ ] args)){
      try {
         property =new Properties();
         property.load(new FileInputStream("server.properties"));

         // Initialize the ORB.
         org.omg.CORBA.ORB orb =
                           org.omg.CORBA.ORB.init(args,property);
         // get a reference to the rootPOA
         POA rootPOA =POAHelper.narrow(orb.resolve_initial_references
                                                ("RootPOA"));

         // Create policies for our persistent POA
         org.omg.CORBA.Policy[ ] policies ={
```

```
            rootPOA.create_lifespan_policy(
                        LifespanPolicyValue.PERSISTENT)
        };

        // Create myPOA with the right policies
        POA myPOA =rootPOA.create_POA("bank_client_server_poa",
                        rootPOA.the_POAManager(),policies );

        // Create Account servants
        AccountManagerImpl managerServant =
                                        new AccountManagerImpl();
        byte[ ] managerId == "BankManager".getBytes();
        myPOA.activate_object_with_id(managerId,managerServant);
        rootPOA.the_POAManager().activate();

        // Announce Servants are ready
        System.out.println(myPOA.servant_to_reference
                                    (managerServant) + "is ready.");
        //Wait for incoming requests
        orb.run();
      }
      catch (Exception e){
      e.printStackTrace();
     }
   }
}
```

(3)　コンパイルの手順

● C++の場合

VisiBroker の C++サンプルをコンパイルするには，次のコマンドを実行します。

UNIX

```
$VBROKERDIR/examples/vbe/pi/client_server>make -f Makefile.cpp
```

Windows

```
%VBROKERDIR%¥examples¥vbe¥pi¥client_server>nmake -f Makefile.cpp
```

VisiBroker の C++サンプルをコンパイルするには，次のコマンドを実行します。

UNIX

```
$VBROKERDIR/examples/vbe/pi/client_server>make -f Makefile.cpp dll
```

Windows

```
%VBROKERDIR%¥examples¥vbe¥pi¥client_server>nmake -f Makefile.cpp dll
```

● Java の場合

Java のサンプルをコンパイルするには，次のコマンドを実行します。

UNIX

```
$VBROKERDIR/examples/vbe/pi/client_server>make -f Makefile.java
```

Windows

```
%VBROKERDIR%¥examples¥vbe¥pi¥client_server>vbmake
```

または，環境変数%VBROKERDIR%¥bin が環境変数 PATH にすでに追加されている場合は，バッチファイルアイコンをダブルクリックします。

(4)　クライアントおよびサーバアプリケーションの実行または配置

● C++の場合

VisiBroker の C++サンプルを実行するには，サーバとクライアントを次のように起動します。

UNIX（二つのコンソールウィンドウをオープンします）

```
$VBROKERDIR/examples/vbe/pi/client_server>Server（最初のウィンドウ）
$VBROKERDIR/examples/vbe/pi/client_server>Client John（2 番目のウィンドウ：任意の名前を使用）
または，
$VBROKERDIR/examples/vbe/pi/client_server> Client（2 番目のウィンドウ：デフォルトの名前を使用）
```

Windows

```
%VBROKERDIR%¥examples¥vbe¥pi¥client_server> start Server（新しいコマンドプロンプトウィンドウ下で実行）
%VBROKERDIR%¥examples¥vbe¥pi¥client_server> Client John（任意の名前を使用）
または，
%VBROKERDIR%¥examples¥vbe¥pi¥client_server> Client（デフォルトの名前を使用）
```

VisiBroker の C++サンプルを実行するには，サーバとクライアントを次のように起動します。

UNIX（二つのコンソールウィンドウをオープンします）

```
$VBROKERDIR/examples/vbe/pi/client_server>Server
-Dvbroker.orb.dynamicLibs=./SampleServerLoader.so（最初のウィンドウ）
$VBROKERDIR/examples/vbe/pi/client_server>Client
-Dvbroker.orb.dynamicLibs=./SampleClientLoader.so John（2 番目のウィンドウ：任意の名前を使用）
```

Windows

```
%VBROKERDIR%¥examples¥vbe¥pi¥client_server>start Server -
Dvbroker.orb.dynamicLibs=SampleServerLoader.dll（新しいコマンドプロンプトウィンドウ下で実行）
%VBROKERDIR%¥examples¥vbe¥pi¥client_server>Client John -
Dvbroker.orb.dynamicLibs=SampleClientLoader.dll（任意の名前を使用）
または，
%VBROKERDIR%¥examples¥vbe¥pi¥client_server>Client -
Dvbroker.orb.dynamicLibs=SampleClientLoader.dll（デフォルトの名前を使用）
```

● Java の場合

インストール済みのポータブルインタセプタで Java サンプルを実行するには，サーバとクライアントを次のように起動します。

UNIX（二つのコンソールウィンドウをオープンします）

```
$VBROKERDIR/examples/vbe/pi/client_server>vbj Server（最初のウィンドウ）
$VBROKERDIR/examples/vbe/pi/client_server>vbj Client John（2 番目のウィンドウ：任意の名前を使用）
または，
$VBROKERDIR/examples/vbe/pi/client_server>vbj Client（2 番目のウィンドウ：デフォルトの名前を使用）
```

Windows

%VBROKERDIR%¥examples¥vbe¥pi¥client_server>start vbj Server（新しいコマンドプロンプトウィンドウ下で実行）

%VBROKERDIR%¥examples¥vbe¥pi¥client_server>vbj Client John（任意の名前を使用）

20 VisiBroker 4.x インタセプタの使用

この章では，VisiBroker 4.x インタセプタの構成の概要について説明し，VisiBroker 4.x インタセプタの例を示しながら，インタセプタファクトリやインタセプタのチェーン化などの高度な機能について説明します。最後に，この章ではポータブルインタセプタと VisiBroker 4.x インタセプタが同じサービスを使用している場合に予測される動作について説明します。

20.1 概要

VisiBroker 4.x インタセプタは，VisiBroker 4.x で定義され，インプリメントされるインタセプタです。ポータブルインタセプタと同様，VisiBroker 4.x インタセプタは ORB の正常な実行フローを受け取る機能である Borland Enterprise Server VisiBroker ORB サービスを提供します。VisiBroker 4.x インタセプタには次の 2 種類があります。

- クライアントインタセプタは，メソッドがクライアントオブジェクトで起動される場合に呼び出されるシステムレベルのインタセプタです。
- サーバインタセプタは，メソッドがサーバオブジェクトで起動される場合に呼び出されるシステムレベルのインタセプタです。

インタセプタを使用するには，インタセプタインタフェースの一つをインプリメントするクラスを宣言します。インタセプタオブジェクトを実体化したら，そのインタセプタオブジェクトを対応するインタセプタマネージャに登録します。インタセプタオブジェクトに対応するマネージャは，起動されたメソッドやマーシャルされたりデマーシャルされたりしたパラメタの一つがオブジェクトにある場合に，そのインタセプタオブジェクトを認識できるようになります。

注

オペレーション要求がクライアント側でマーシャルされる前，またはサーバ側で処理される前にそのオペレーション要求を受け取りたい場合は，オブジェクトラッパーを使用してください（「21. オブジェクトラッパーの使用」を参照）。

20.2　VisiBroker 4.x インタセプタインタフェースおよびマネージャ

VisiBroker 4.x インタセプタの開発者は，VisiBroker 4.x によって定義され，インプリメントされた次に示す複数のベースインタセプタ API クラスからクラスを派生させます。

- クライアントインタセプタ
  - BindInterceptor
  - ClientRequestInterceptor
- サーバインタセプタ
  - POALifeCycleInterceptor
  - ActiveObjectLifeCycleInterceptor
  - ServerRequestInterceptor
  - IORCreationInterceptor
- ServiceResolver インタセプタ
  - Service Resolver Interceptor

20.2.1　クライアントインタセプタ

次の 2 種類のクライアントインタセプタとそれに対応するマネージャがあります。

- BindInterceptor および BindInterceptorManager
- ClientRequestInterceptor および ClientRequestInterceptorManager

クライアントインタセプタの詳細については，「19.　ポータブルインタセプタの使用」を参照してください。

(1)　BindInterceptor

BindInterceptor オブジェクトは，バインド前後にクライアント側で呼び出されるグローバルインタセプタです。

コードサンプル 20-1　BindInterceptor インタフェース（Java）

```
package com.inprise.vbroker.InterceptorExt;
public interface BindInterceptor {
   public IORValue bind(IORValue ior,
         org.omg.CORBA.Object target,
         boolean rebind,
         Closure closure);
   public IORValue bind_failed(IORValue ior,
         org.omg.CORBA.Object target,
         Closure closure);
   public void bind_succeeded(IORValue ior,
         org.omg.CORBA.Object target,
         int Index,
         InterceptorManagerControl control,
         Closure closure);
   public void exception_occurred(IORValue ior,
         org.omg.CORBA.Object target,
         org.omg.CORBA.Environment env,
         Closure closure);
}
```

(2)　ClientRequestInterceptor

ClientRequestInterceptor オブジェクトは，BindInterceptor オブジェクトの bind_succeeded 呼び出し中に登録できます。またこれは，コネクション期間中アクティブのままです。このメソッドのうちの二つは，クライアントオブジェクトの起動前に呼び出されます。一つ（preinvoke_premarshal）は，パラメタがマーシャルされる前に，もう一つ（preinvoke_postmarshal）は，パラメタがマーシャルされたあとに呼び出されます。三つ目のメソッド（postinvoke）は，リクエストの完了後に呼び出されます。

コードサンプル 20-2　ClientRequestInterceptor インタフェース（Java）

```
package com.inprise.vbroker.InterceptorExt;
public interface ClientRequestInterceptor {
   public void preinvoke_premarshal(org.omg.CORBA.Object target,
         String operation,
         ServiceContextListHolder service_contexts_holder,
         Closure closure);
   public void preinvoke_postmarshal(org.omg.CORBA.Object target,
         OutputStream payload,
         Closure closure);
   public void postinvoke(org.omg.CORBA.Object target,
         ServiceContext [ ] service_contexts,
         InputStream payload,
         org.omg.CORBA.Environment env,
         Closure closure);
   public void exception_occurred(org.omg.CORBA.Object target,
         org.omg.CORBA.Environment env,
         Closure closure);
}
```

20.2.2　サーバインタセプタ

4 種類のサーバインタセプタがあります。

- POALifeCycleInterceptor および POALifeCycleInterceptorManager
- ActiveObjectLifeCycleInterceptor および ActiveObjectLifeCycleInterceptorManager
- ServerRequestInterceptor および ServerRequestInterceptorManager
- IORCreationInterceptor および IORCreationInterceptorManager

サーバインタセプタの詳細については，「19.　ポータブルインタセプタの使用」を参照してください。

(1)　POALifeCycleInterceptor

POALifeCycleInterceptor オブジェクトは POA が（create()メソッドによって）生成されるたび，または（destroy()メソッドによって）デストラクトされるたびに呼び出されるグローバルインタセプタです。

コードサンプル 20-3　POALifeCycleInterceptor インタフェース（Java）

```
package com.inprise.vbroker.InterceptorExt;
public interface POALifeCycleInterceptor {
   public void create(org.omg.PortableServer.POA poa,
      org.omg.CORBA.PolicyListHolder policies_holder,
      IORValueHolder iorTemplate,
      InterceptorManagerControl control);
   public void destroy(org.omg.PortableServer.POA poa);
}
```

(2)　ActiveObjectLifeCycleInterceptor

ActiveObjectLifeCycleInterceptor オブジェクトは，オブジェクトが（create メソッドによって）アクティブオブジェクトマップに追加された場合，またはオブジェクトが（destroy メソッドによって）停止し，エーテライズされたあとに呼び出されます。インタセプタは，POALifeCycleInterceptor によって，

POA 生成時に POA ごとに登録できます。このインタセプタは，POA に RETAIN ポリシーがある場合だけ登録できます。

コードサンプル 20-4　ActiveObjectLifeCycleInterceptor インタフェース（Java）

```
package com.inprise.vbroker.InterceptorExt;
public interface ActiveObjectLifeCycleInterceptor {
   public void create(byte[ ] oid,,
         org.omg.PortableServer.Servant servant,
         org.omg.PortableServer.POA adapter);
   public void destroy (byte[ ] oid,,
         org.omg.PortableServer.Servant servant,
         org.omg.PortableServer.POA adapter);
}
```

(3)　ServerRequestInterceptor

ServerRequestInterceptor オブジェクトは，リモートオブジェクトのサーバインプリメンテーションの起動中のさまざまな段階，例えば（preinvoke メソッドによる）起動前，（postinvoke_premarshal および postinvoke_premarshal メソッドによる）応答のマーシャル前後の起動後に呼び出されます。このインタセプタは，POA ごとに，POA の生成時に POALifeCycleInterceptor オブジェクトによって登録できます。

コードサンプル 20-5　ServerRequestInterceptor インタフェース（Java）

```
package com.inprise.vbroker.InterceptorExt;
public interface ServerRequestInterceptor {
   public void preinvoke(org.omg.CORBA.Object target,
      String operation,
      ServiceContext [ ] service_contexts,
      InputStream payload,
      Closure closure);
   public void postinvoke_premarshal(org.omg.CORBA.Object target,
      ServiceContextListHolder service_contexts_holder,
      org.omg.CORBA.Environment env,
      Closure closure);
   public void postinvoke_postmarshal(org.omg.CORBA.Object target,
      OutputStream payload,
      Closure closure);
   public void exception_occurred(org.omg.CORBA.Object target,
      org.omg.CORBA.Environment env,
      Closure closure);
}
```

(4)　IORCreationInterceptor

IORCreationInterceptor オブジェクトは，POA が（create メソッドによって）オブジェクトリファレンスを生成する場合に呼び出されます。このインタセプタは，POA ごとに，POA の生成時に POALifeCycleInterceptor オブジェクトによって登録できます。

IDL サンプル 20-1　IORCreationInterceptor インタフェース（Java）

```
package com.inprise.vbroker.InterceptorExt;
public interface IORCreationInterceptor {
   public void create(org.omg.PortableServer.POA poa,
         IORValueHolder ior);
}
```

20.2.3　ServiceResolver インタセプタ

このインタセプタはユーザサービスをインストールするために使用し，これによってユーザサービスを動的にロードできるようになります。

コードサンプル 20-6　ServiceResolverInterceptor インタフェース（Java）

```
public interface ServiceResolverInterceptor {
   public org.omg.CORBA.Object resolve (java.lang.String name):
}
public interface ServiceResolverInterceptorManager extends
   com.inprise.vbroker.interceptor.InterceptorManager {
      public void add (java.lang.String name,
      com.inprise.vbroker.interceptor.ServiceResolverInterceptor
                                                ¥interceptor);
   public void remove (java.lang.String name):
}
```

resolve_initial_references()メソッドを実行すると，すべてのユーザのインストールされたサービスの resolve が呼び出されます。resolve は該当するオブジェクトを返せます。

サービスイニシャライザを書き込むには，サービスを追加できるようになる InterceptorManagerControl を取得後に，ServiceResolver を取得する必要があります。

20.2.4　デフォルトのインタセプタクラス（Java）

Borland Enterprise Server VisiBroker は，デフォルトのインタセプタ Java クラスを提供します。このクラスによって継承とインプリメントができます。このデフォルトのインタセプタクラスはインタセプタインタフェースと同じメソッドを提供しますが，デフォルトのインタセプタクラスを継承すると，インプリメントまたは変更するメソッドを選択できます。このクラスを使用する場合，このクラスが提供するデフォルトの動作を受け付けるか，またはそれを変更できます。

- DefaultBindInterceptor クラス
- DefaultClientInterceptor クラス
- DefaultServerInterceptor クラス

20.2.5　Borland Enterprise Server VisiBroker ORB へのインタセプタの登録

それぞれのインタセプタインタフェースには，Borland Enterprise Server VisiBroker ORB にインタセプタオブジェクトを登録する際に使用する，対応するインタセプタマネージャインタフェースがあります。インタセプタを登録するための手順を次に示します。

1. ORB オブジェクトで，パラメタを VisiBrokerInterceptorControl として resolve_initial_references メソッドを呼び出して，InterceptorManagerControl オブジェクトのリファレンスを取得します。
2. 表 20-1 の String 値のどれかを指定した InterceptorManagerControl オブジェクトで get_manager メソッドを呼び出します。表 20-1 は InterceptorManagerControl オブジェクトの get_manager メソッドへ渡す String 値を示します（オブジェクトリファレンスを対応するインタセプタマネージャインタフェースに必ずキャストしてください）。

表 20–1　InterceptorManagerControl オブジェクトの String 値

値	対応するインタセプタインタフェース
ClientRequest	ClientRequestInterceptor
Bind	BindInterceptor
POALifeCycle	POALifeCycleInterceptor
ActiveObjectLifeCycle	ActiveObjectLifeCycleInterceptor

値	対応するインタセプタインタフェース
ServerRequest	ServerRequestInterceptor
IORCreation	IORCreationInterceptor
ServiceResolver	ServiceResolverInterceptor

3. インタセプタのインスタンスを生成します。
4. add メソッドを呼び出して，インタセプタオブジェクトをマネージャオブジェクトに登録します。
5. クライアントプログラムおよびサーバプログラムの実行時にインタセプタオブジェクトをロードします。

20.2.6　インタセプタオブジェクトの生成

ここで，インタセプタのインスタンスを生成し，それを Borland Enterprise Server VisiBroker ORB に登録するファクトリクラスをインプリメントする必要があります。ファクトリクラスは ServiceLoader インタフェース（Java）をインプリメントしなければなりません。

コードサンプル 20-7　ServiceLoader インタフェース（Java）

```
package com.inprise.vbroker.interceptor;
public interface ServiceLoader {
   //This method is called by the ORB when ORB.init()is called.
   public abstract void init(org.omg.CORBA.ORB orb);

   //Called after ORB.init()is done but control hasn't been returned to
   //the user.Can be used to disable certain resources that were only
   //made available to other service inits.
   public abstract void init_complete(org.omg.CORBA.ORB orb);

   //Called when the orb is being shutdown.
   public abstract void shutdown(org.omg.CORBA.ORB orb);
}
```

注

次の例のように，インタセプタの新しいインスタンスを生成し，それを別のインタセプタから Borland Enterprise Server VisiBroker ORB に登録することもできます。

20.2.7　インタセプタのロード

● Java の場合

インタセプタをロードするには，vbroker.orb.dynamicLibs プロパティを設定する必要があります。このプロパティはプロパティファイルで設定するか，または-D オプションを使用して Borland Enterprise Server VisiBroker ORB に渡せます。

20.3　インタセプタのサンプル

次のインタセプタのサンプルでは，（「19.　ポータブルインタセプタの使用」にリストした）すべてのインタセプタ API メソッドを使用しています。そのため，これらのメソッドがどのように使用されているか，またいつ呼び出されるかがわかるようになっています。

20.3.1　コードサンプル

「20.3.2　コード一覧」では，それぞれのインタセプタ API メソッドは単純にインプリメントされており，これは標準出力に情報メッセージを出力します。

Borland Enterprise Server VisiBroker インストレーションの examples/vbe/interceptors ディレクトリには次の四つのアプリケーションの例があります。

- active_object_lifecycle
- client_server
- ior_creation
- encryption（Java）

(1)　クライアントーサーバインタセプタのサンプル

サンプルを実行するには，通常どおりにファイルをコンパイルします。そして，サーバとクライアントを次に示すように起動します。

Java の場合

```
prompt>vbj -Dvbroker.orb.dynamicLibs=SampleServerLoader Server
prompt>vbj -Dvbroker.orb.dynamicLibs=SampleClientLoader Client Kate
```

Borland Enterprise Server VisiBroker ORB サービスに，ServiceLoader インタフェースをインプリメントする二つのクラスを指定します。

注

VisiBroker 3.x で使用した ServiceInit クラスは，ServiceLoader および ServiceResolverInterceptor という二つのインタフェースをインプリメントすることによって置き換えられています。この方法のサンプルについては，「20.3.1（2） ServiceResolverInterceptor のサンプル（Java）」を参照してください。

表 20-2 にインタセプタの例の実行結果を示します。クライアントとサーバによる実行が順を追って示してあります。

表 20-2　インタセプタの例の実行結果

クライアント	サーバ
	========> SampleServerLoader:Interceptors loaded ========> In POA / . Nothing to do. ========> In POA bank_agent_poa,1 ServerRequest interceptor installed Stub [repository_id=IDL:Bank/AccountManager:1.0,key=ServiceId [service=/bank_agent _poa,id={11 bytes: [B][a][n][k][M][a][n][a][g][e][r]}]] is ready.
Bind Interceptors loaded ========> SampleBindInterceptor bind	

クライアント	サーバ
========> SampleBindInterceptor bind_succeeded ========> SampleClientInterceptor id MyClientInterceptor preinvoke_premarshal => open ========> SampleClientInterceptor id MyClientInterceptor preinvoke_postmarshal	
	========> SampleServerInterceptor id MyServerInterceptor preinvoke => open Created john's account: Stub [repository_id=IDL:Bank/ Account:1.0,key=TransientId [poaName=/,id={4 bytes: (0)(0)(0)(0)},sec=0,usec=0]]
========> SampleClientInterceptor id MyClientInterceptor postinvoke ========> SampleBindInterceptor bind ========> SampleBindInterceptor bind_succeeded ========> SampleClientInterceptor id MyClientInterceptor preinvoke_premarshal => balance ========> SampleClientInterceptor id MyClientInterceptor preinvoke_postmarshal	
	========>SampleServerInterceptor id MyServerInterceptor postinvoke_premarshal ========>SampleServerInterceptor id MyServerInterceptor postinvoke_postmarshal
========>SampleClientInterceptor id MyClientInterceptor postinvoke The balance in john's account is $245.64	

OAD は実行されていないので，bind()メソッドの呼び出しは失敗し，サーバが処理を続行します。クライアントはアカウントオブジェクトにバインドしてから，balance()メソッドを起動します。このリクエストはサーバが受信して処理し，結果がクライアントに戻されます。クライアントは結果を出力します。

コードと結果のサンプルに示したように，クライアントとサーバの両方のインタセプタは，それぞれのプロセスの開始時点でインストールされます。インタセプタの登録についての情報については，「20.2.5 Borland Enterprise Server VisiBroker ORB へのインタセプタの登録」を参照してください。

(2)　ServiceResolverInterceptor のサンプル（Java）

次のコードで ServiceLoader インタフェースのインプリメント方法の例を示します。

```
import com.inprise.vbroker.properties.*;
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.InterceptorExt.*;

public final class UtilityServiceLoader implements ServiceLoader,
   ServiceResolverInterceptor {
   private com.inprise.vbroker.orb.ORB _orb = null;
   private String[ ] __serviceNames = {"TimeService",
                                       "WeatherService"};

   public void init(org.omg.CORBA.ORB orb) {
      //Just in case they are needed by resolve()
      _orb =(com.inprise.vbroker.orb.ORB)orb;

      PropertyManager pm =_orb.getPropertyManager();
      //use the PropertyManager to query property settings
      //if needed (not used in this example)

      /****Installing the Initial Reference *****/
      InterceptorManagerControl control =_orb.interceptorManager();
      ServiceResolverInterceptorManager manager =
```

```
            (ServiceResolverInterceptorManager)control.get_manager
                    ("ServiceResolver");
            for (int i =0;i <_serviceNames.length;i++){
               manager.add(_serviceNames [i ],this);
            }
            /****end of installation ***/

            if (_orb.debug)
               _orb.println("UtilityServices package has been
                initialized");
         }

         public void init_complete(org.omg.CORBA.ORB orb){
            //can be used for post-initialization processing if desired
         }

         public void shutdown(org.omg.CORBA.ORB orb){
            _orb =null;
            _serviceNames =null;
         }

         public org.omg.CORBA.Object resolve(java.lang.String service){
            org.omg.CORBA.Object srv =null;
            byte[ ] serviceId =service.getBytes();
            try {
               if (service =="TimeService"){
                  srv =UtilityServices.TimeServiceHelper.bind(_orb,
                     "/time_service_poa",serviceId);
               }
               else if (service =="WeatherService"){
                  srv =UtilityServices.WeatherServiceHelper.bind(_orb,
                     "/weather_service_poa",serviceId);
               }
            }catch (org.omg.CORBA.SystemException e){
               if (_orb.debug)
                  _orb.println("UtilityServices package resolve error:"+e);
               srv =null;
            }

             return srv;
         }
      }
```

20.3.2 コード一覧

Java の場合，SampleServerInterceptorLoader オブジェクトは，POALifeCycleInterceptor クラスのロードとオブジェクトの実体化に責任があります。このクラスは，vbroker.orb.dynamicLibs によって動的に Borland Enterprise Server VisiBroker ORB にリンクされます。SampleServerLoader クラスには init()メソッドがあり，このメソッドは初期化時に Borland Enterprise Server VisiBroker ORB によって起動されます。その唯一の目的は，POALifeCycleInterceptor オブジェクトを生成して，InterceptorManager に登録することによって，POALifeCycleInterceptor オブジェクトをインストールすることです。

コードサンプル 20-8　SampleServerLoader.java（Java）

```
import java.util.*;
import com.inprise.vbroker.orb.*;
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.PortableServerExt.*;

public class SampleServerLoader implements ServiceLoader {
   public void init(org.omg.CORBA.ORB orb){
      try {
         InterceptorManagerControl control =
           InterceptorManagerControlHelper.narrow(
             orb.resolve_initial_references
                ("VisiBrokerInterceptorControl"));
         //Install a POA interceptor
         POALifeCycleInterceptorManager poa_manager =
```

```
                (POALifeCycleInterceptorManager)control.get_manager
                    ("POALifeCycle");
            poa_manager.add(new SamplePOALifeCycleInterceptor());
         }catch(Exception e){
            e.printStackTrace();
            throw new org.omg.CORBA.INITIALIZE(e.toString());
         }
         System.out.println("============>SampleServerLoader:
               Interceptors loaded");
      }
      public void init_complete(org.omg.CORBA.ORB orb){
      }
      public void shutdown(org.omg.CORBA.ORB orb){
      }
   }
```

SamplePOALifeCycleInterceptor オブジェクトは，POA が生成されるたびに，または POA がデストラクトされるたびに起動されます。client_server の例では二つの POA があるので，このインタセプタは 2 回起動されます。1 回目は，rootPOA 生成時，2 回目は myPOA 生成時です。myPOA の生成時だけ，SampleServerInterceptor をインストールします。

コードサンプル 20-9　SamplePOALifeCycleInterceptor.java（Java）

```
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.PortableServerExt.*;
import com.inprise.vbroker.IOP.*;

public class SamplePOALifeCycleInterceptor implements
                                          POALifeCycleInterceptor {
   public void create(org.omg.PortableServer.POA poa,
      org.omg.CORBA.PolicyListHolder policies_holder,
      IORValueHolder iorTemplate,
      InterceptorManagerControl control){
   if(poa.the_name().equals("bank_agent_poa")){
      //Add the Request-level interceptor
      SampleServerInterceptor interceptor =
            new SampleServerInterceptor("MyServerInterceptor");
      //Get the IORCreation interceptor manager
      ServerRequestInterceptorManager manager =
            (ServerRequestInterceptorManager)control.get_manager
                ("ServerRequest");
      //Add the interceptor
      manager.add(interceptor);
      System.out.println("========>In POA " + poa.the_name() +
            ",1 ServerRequest interceptor installed");
   }else
      System.out.println("========>In POA " + poa.the_name() +
                                            ". Nothing to do.");
   }
   public void destroy(org.omg.PortableServer.POA poa){
      //To be a trace!
      System.out.println("========>SamplePOALifeCycleInterceptor
                                                      destroy");
   }
}
```

SampleServerInterceptor オブジェクトは，リクエストを受信するたびに，またはサーバが応答するたびに起動されます。

コードサンプル 20-10　SampleServerInterceptor.java（Java）

```
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.IOP.*;
import com.inprise.vbroker.CORBA.portable.*;

public class SampleServerInterceptor implements ServerRequestInterceptor {
   private String _id;
   public SampleServerInterceptor(String id){
      _id =id;
   }
   public void preinvoke(org.omg.CORBA.Object target,
```

```
            String operation,
            ServiceContext [ ] service_contexts,
            InputStream payload,
            Closure closure){
         //Put the _id of this ServerRequestInterceptor into the
         // closure object
         closure.object =new String(_id);
         System.out.println("========>SampleServerInterceptor id "+
            closure.object + " preinvoke => " + operation);
      }
      public void postinvoke_premarshal(org.omg.CORBA.Object target,
            ServiceContextListHolder service_contexts_holder,
            org.omg.CORBA.Environment env,
            Closure closure){
            System.out.println("========>SampleServerInterceptor id "+
               closure.object + " postinvoke_premarshal");
      }
      public void postinvoke_postmarshal(org.omg.CORBA.Object target,
            OutputStream payload,
            Closure closure){
            System.out.println("========>SampleServerInterceptor id "+
               closure.object + " postinvoke_postmarshal");
      }
      public void exception_occurred(org.omg.CORBA.Object target,
            org.omg.CORBA.Environment env,
            Closure closure){
         System.out.println("============>SampleServerInterceptor id "+
               closure.object +" exception_occurred");
      }
   }
```

SampleClientInterceptor は，リクエストを生成するたびに，またはクライアントが応答を受信するたびに起動されます。

ローダは BindInterceptor オブジェクトのロードに責任があります。SampleClientInterceptorLoader クラスには，bind()メソッドと bind_succeeded()メソッドがあります。これらのメソッドは，オブジェクトのバインド時に Borland Enterprise Server VisiBroker ORB によって起動されます。バインドが成功すると，bind_succeeded()メソッドが ORB によって起動され，BindInterceptor オブジェクトを生成し，それを InterceptorManager に登録することによって，BindInterceptor オブジェクトがインストールされます。

コードサンプル 20-11　SampleClientInterceptor.java（Java）

```
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.IOP.*;
import com.inprise.vbroker.CORBA.portable.*;

public class SampleClientInterceptor implements ClientRequestInterceptor {
   private String _id;
   public SampleClientInterceptor(String id){
      _id =id;
   }
   public void preinvoke_premarshal(org.omg.CORBA.Object target,
         String operation,
         ServiceContextListHolder service_contexts_holder,
         Closure closure){
      // Put the _id of this ClientRequestInterceptor into the
      // closure object
      closure.object =new String(_id);
      System.out.println("============>SampleClientInterceptor id "+
            closure.object +
            "preinvoke_premarshal =>"+operation);
   }
   public void preinvoke_postmarshal(org.omg.CORBA.Object target,
         OutputStream payload,
         Closure closure){
      System.out.println("============>SampleClientInterceptor id "+
         closure.object +"preinvoke_postmarshal");
   }
   public void postinvoke(org.omg.CORBA.Object target,
```

```
            ServiceContext [ ] service_contexts,
            InputStream payload,
            org.omg.CORBA.Environment env,
            Closure closure){
         System.out.println("============>SampleClientInterceptor id "+
               closure.object +"postinvoke");
      }
      public void exception_occurred(org.omg.CORBA.Object target,
            org.omg.CORBA.Environment env,
            Closure closure){
         System.out.println("============>SampleClientInterceptor id "+
               closure.object +"exception_occurred");
      }
   }
```

Java の場合，ローダは BindInterceptor オブジェクトのロードに責任があります。このクラスは，vbroker.orb.dynamicLibs によって動的に Borland Enterprise Server VisiBroker ORB にリンクされます。SampleClientInterceptorLoader クラスには，bind()メソッドと bind_succeeded()メソッドがあります。これらのメソッドは，オブジェクトのバインド時に ORB によって起動されます。バインドが成功すると，bind_succeeded()メソッドが ORB によって起動され，BindInterceptor オブジェクトを生成し，それを InterceptorManager に登録することによって，BindInterceptor オブジェクトがインストールされます。

コードサンプル 20-12　SampleClientLoader.java（Java）

```
import java.util.*;
import com.inprise.vbroker.orb.*;
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.PortableServerExt.*;

public class SampleClientLoader implements ServiceLoader {
   public void init(org.omg.CORBA.ORB orb){
      try {
         InterceptorManagerControl control =
            InterceptorManagerControlHelper.narrow(
               orb.resolve_initial_references("VisiBrokerInterceptorControl"));
         BindInterceptorManager bind_manager =
               (BindInterceptorManager)control.get_manager("Bind");
         bind_manager.add(new SampleBindInterceptor());
      }catch(Exception e){
         e.printStackTrace();
         throw new org.omg.CORBA.INITIALIZE(e.toString());
      }
      System.out.println("Bind Interceptors loaded");
   }
   public void init_complete(org.omg.CORBA.ORB orb){
   }
   public void shutdown(org.omg.CORBA.ORB orb){
   }
}
```

SampleBindInterceptor は，オブジェクトにバインドしようとするクライアントによって起動されます。ORB 初期化後のクライアント側の最初の手順は，AccountManager オブジェクトにバインドすることです。このバインドによって SampleBindInterceptor を起動し，バインドが成功すると，SampleClientInterceptor がインストールされます。

コードサンプル 20-13　SampleBindInterceptor.java（Java）

```
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.IOP.*;

public class SampleBindInterceptor implements BindInterceptor {
   public IORValue bind(IORValue ior,org.omg.CORBA.Object target,
         boolean rebind,Closure closure){
      //To be a trace!
      System.out.println("============>SampleBindInterceptor bind");
      return null;
   }
   public IORValue bind_failed(IORValue ior,org.omg.CORBA.Object target,
```

```
            Closure closure){
         //To be a trace!
         System.out.println("============>SampleBindInterceptor
           bind_failed");
         return null;
      }
      public void bind_succeeded(IORValue ior,org.omg.CORBA.Object target,
            int Index,InterceptorManagerControl control,
            Closure closure){
         //To be a trace!
         System.out.println("============>SampleBindInterceptor
           bind_succeeded");
         //Create the Client Request interceptor:
         SampleClientInterceptor interceptor =
         new SampleClientInterceptor("MyClientInterceptor");
         //Get the manager
         ClientRequestInterceptorManager manager =
               (ClientRequestInterceptorManager)control.
                                       get_manager("ClientRequest");
         //Add CRQ to the list:
         manager.add(interceptor);
      }
      public void exception_occurred(IORValue ior,org.omg.CORBA.Object
         target,
            org.omg.CORBA.Environment env,
            Closure closure){
         //To be a trace!
         System.out.println("============>SampleBindInterceptor
          exception_occurred");
      }
   }
```

20.4　VisiBroker 4.x インタセプタ間での情報の渡し方

Closure（Java）オブジェクトは，インタセプタ呼び出しのシーケンスの始めに ORB によって生成されます。同じ Closure（Java）オブジェクトがその特定のシーケンスのすべての呼び出しで使用されます。Closure（Java）オブジェクトには，一つのパブリックデータフィールドである java.lang.Object タイプ（Java）のオブジェクトがあり，これは状態情報を保存するためにインタセプタが設定します。

Closure オブジェクトが生成されるシーケンスは，インタセプタ型によって異なります。

ClientRequestInterceptor では，preinvoke_premarshal の呼び出し前に新しい Closure（Java）が生成され，リクエストの完了が成功してもしなくても，そのリクエストが完了するまで，そのリクエストで同じ Closure（Java）が使用されます。ServerInterceptor でも同様に，新しい Closure（Java）は preinvoke の呼び出し前に生成され，その Closure（Java）が特定のリクエストの処理に関連するすべてのインタセプタ呼び出しで使用されます。

Closure（Java）の使用方法のサンプルについては，Borland Enterprise Server VisiBroker インストレーションの examples/vbe/interceptors/client_server ディレクトリを参照してください。

response_expected と request_id を取得するために，Closure（Java）オブジェクトは次のように ExtendedClosure（Java）にキャストできます。

```
int my_response_expected
   =((ExtendedClosure)closure).reqInfo.response_expected;
int my_request_id
   =((ExtendedClosure)closure).reqInfo.request_id;
```

20.5 ポータブルインタセプタおよび VisiBroker 4.x インタセプタを同時に使用

ポータブルインタセプタおよび VisiBroker 4.x インタセプタは Borland Enterprise Server VisiBroker ORB で同時にインストールできますが，この二つのインタセプタには異なるインプリメンテーションがあるので, 両方のインタセプタを使用する際に開発者が理解しておかなければならないフローの規則と制約があります。

20.5.1 インタセプトポイントの呼び出し順

インタセプトポイントの呼び出し順は, 開発者が実際に一つ以上のインタセプタのバージョンをインストールするかどうかに関係なく, それぞれのバージョンのインタセプタのインタセプトポイント順序の規則に従います。

20.5.2 クライアント側インタセプタ

クライアント側にポータブルインタセプタおよび VisiBroker インタセプタの両方がインストールされる場合，例外を発生させるインタセプタがないと想定されるイベント順は次のようになります。

1. send_request（ポータブルインタセプタ）の次に preinvoke_premarshal（VisiBroker 4.x インタセプタ）
2. 構成体リクエストメッセージ
3. preinvoke_postmarshal（VisiBroker 4.x インタセプタ）
4. 送信リクエストメッセージおよび応答待ち
5. 応答のタイプに応じて，postinvoke（VisiBroker 4.x インタセプタ）の次に received_reply, receive_exception，または receive_other（ポータブルインタセプタ）

20.5.3 サーバ側インタセプタ

サーバ側にポータブルインタセプタおよび VisiBroker インタセプタの両方がインストールされる場合，例外を発生させるインタセプタがないと想定されるイベント受信順（VisiBroker の動作と同様，リクエストの探索はインタセプタを起動しない）は次のようになります。

1. received_request_service_contexts（ポータブルインタセプタ）の次に preinvoke（VisiBroker 4.x インタセプタ）
2. servantLocator.preinvoke（サーバントロケータを使用している場合）
3. receive_request（ポータブルインタセプタ）
4. サーバントでの起動オペレーション
5. postinvoke_premarshal（VisiBroker 4.x インタセプタ）
6. servantLocator.postinvoke（サーバントロケータを使用している場合）
7. リクエストの結果に応じて，send_reply，send_exception，または send_other
8. postinvoke_postmarshal（VisiBroker 4.x インタセプタ）

20.5.4　POA 生成中の ORB イベント順

POA 生成中の ORB イベント順を次に示します。

1. IOR テンプレートは POA を処理するサーバエンジンのプロファイルに基づいて生成されます。
2. VisiBroker 4.x インタセプタの POA ライフサイクルインタセプタの create()メソッドが呼び出されます。このメソッドは新しいポリシーを追加したり，前述の手順で生成された IOR テンプレートを修正する可能性があります。
3. ポータブルインタセプタの IORInfo オブジェクトが生成され，IORInterceptor の establish_components()メソッドが呼び出されます。このインタセプトポイントによって，インタセプタは create_POA()に渡されたポリシー，および前述の手順で追加されたポリシーを照会し，このポリシーに基づいて IOR テンプレートにコンポーネントを追加できます。
4. POA のオブジェクトリファレンスファクトリとオブジェクトリファレンステンプレートが生成され，ポータブルインタセプタ IORInterceptor の components_established()メソッドが呼び出されます。このインタセプトポイントによって，インタセプタはオブジェクトリファレンスの作成に使用する POA のオブジェクトリファレンスファクトリを変更できます。

20.5.5　オブジェクトリファレンス生成中の ORB イベント順

create_reference()メソッドや create_reference_with_id()メソッドのようなオブジェクトリファレンスを生成する POA の呼び出し中に発生するイベントを次に示します。

1. オブジェクトリファレンス（これは VisiBroker IOR 生成インタセプタを呼び出しません。ファクトリはユーザ提供の場合もあります）を生成するためにオブジェクトリファレンスファクトリの make_object()メソッドを呼び出します。VisiBroker IOR 生成インタセプタがインストールされていなければ，これはアプリケーションに返されるオブジェクトリファレンスであるはずです。インストールされている場合は手順 2.に進みます。
2. 返されたオブジェクトリファレンスのデリゲートから IOR を抽出し，VisiBroker IOR 生成インタセプタの create()メソッドを呼び出します。
3. オブジェクトリファレンスとして create_reference()メソッドおよび create_reference_with_id()メソッドの呼び出し元に手順 2.の IOR が返されます。

21 オブジェクトラッパーの使用

この章では，Borland Enterprise Server VisiBroker のオブジェクトラッパー機能について説明します。この機能を使うと, ユーザのアプリケーションに通知したり，アプリケーションがオブジェクトのオペレーション要求をトラップしたりできます。

21.1　概要

Borland Enterprise Server VisiBroker のオブジェクトラッパー機能を使うと，クライアントアプリケーションがバインドされたオブジェクトのメソッドを呼び出すときに呼び出されるメソッドや，サーバアプリケーションがオペレーション要求を受け取ったときに呼び出されるメソッドを，ユーザが定義できるようになります。前に説明したインタセプタ機能（VisiBroker ORB レベルで起動される）とは異なり，オブジェクトラッパーはオペレーション要求がマーシャルされる前に起動されます。実はオペレーション要求がマーシャルされなくても，ネットワークに送られなくても，または本当にオブジェクトインプリメンテーションに提供されなくても，オブジェクトラッパーが結果を返すように設計できます。

オブジェクトラッパーはクライアント側だけ，サーバ側だけ，または一つのアプリケーションにクライアントとサーバの両方が実装されているものにインストールできます。

ユーザのアプリケーション中でのオブジェクトラッパーの使い方の例を次に示します。

- クライアントが発行した，またはサーバが受け取ったオペレーション要求に関する情報のログを取得する
- オペレーション要求が完了するまでの時間を計る
- 実際には毎回オブジェクトインプリメンテーションにコンタクトすることなく，頻繁に発行されるオペレーション要求の結果がすばやく返されるように，結果をキャッシュする

注

VisiBroker ORB オブジェクトの object_to_string メソッドを使って，オブジェクトラッパーがインストールされているオブジェクトのリファレンスを文字列化しても，文字列化したリファレンスの受信者が異なるプロセスなら，オブジェクトラッパーが伝わることはありません。

21.1.1　タイプドおよびアンタイプドオブジェクトラッパー

Borland Enterprise Server VisiBroker は，タイプドとアンタイプドの 2 種類のオブジェクトラッパーを提供します。一つのアプリケーション内で，両方の型のオブジェクトラッパーを混在して使えます。タイプドオブジェクトラッパーの詳細については，「21.4　タイプドオブジェクトラッパー」を参照してください。表 21-1 に，2 種類のオブジェクトラッパーの相違点を示します。

表 21-1　タイプドおよびアンタイプドオブジェクトラッパーの機能の比較

機能	タイプド	アンタイプド
スタブに渡すすべての引数を受信する	○	×
次のオブジェクトラッパー，スタブ，またはオブジェクトインプリメンテーションを実際に起動しないで，発信元に制御を返すことができる	○	×
すべてのオブジェクトのすべてのオペレーション要求で起動する	×	○

（凡例）○：できる　×：できない

21.1.2　idl2cpp の前提条件（C++）

タイプドオブジェクトラッパーまたはアンタイプドオブジェクトラッパーを使う場合はいつでも，アプリケーションのコード生成の際に，idl2cpp コンパイラに-obj_wrapper オプションを指定しなければなりません。そうすることで次のものが生成されます。

- ユーザのインタフェースごとのオブジェクトラッパーベースクラス

21.1.3　idl2java の前提条件（Java）

タイプドオブジェクトラッパーまたはアンタイプドオブジェクトラッパーを使う場合はいつでも，アプリケーションのコード生成の際に，idl2java コンパイラに-obj_wrapper オプションを指定しなければなりません。そうすることで次のものが生成されます。

- ユーザのインタフェースごとのオブジェクトラッパーベースクラス
- Helper クラスにオブジェクトラッパーの追加，削除のためのメソッドを追加

21.1.4　サンプルアプリケーション

VisiBroker をインストールしたディレクトリの examples/vbe/interceptors/objectWrappers には，クライアントとサーバのサンプルアプリケーションが三つ入っています。この章では，このサンプルを使って，タイプドオブジェクトラッパーおよびアンタイプドオブジェクトラッパーの概念を説明します。

21.2　アンタイプドオブジェクトラッパー

アンタイプドオブジェクトラッパーを使うと，ユーザは，オペレーション要求が処理される前，あと，または前後両方で呼び出されるメソッドを定義できます。アンタイプドオブジェクトラッパーは，クライアントアプリケーション用またはサーバアプリケーション用にインストールでき，また複数のオブジェクトラッパーをインストールできます。

同じクライアントまたはサーバアプリケーション内で，タイプドとアンタイプドの両方のオブジェクトラッパーを混在して使えます。

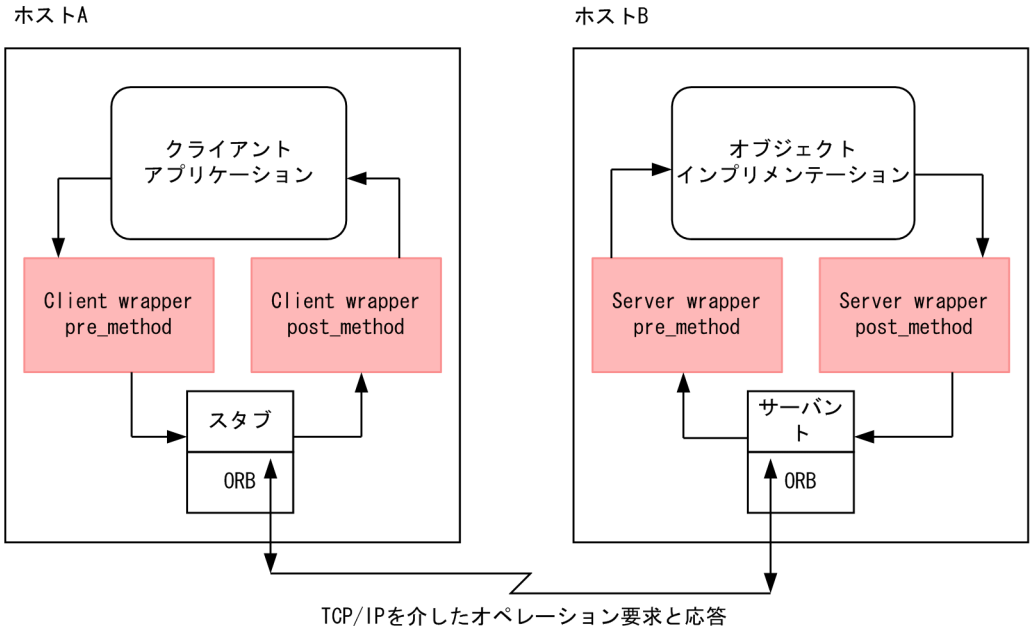
デフォルトでは，アンタイプドオブジェクトラッパーはグローバルスコープを持ち，どのようなオペレーション要求に対しても起動されます。関係ないオブジェクトタイプのオペレーション要求については影響を与えないように，アンタイプドオブジェクトラッパーを設計できます。

注

タイプドオブジェクトラッパーとは異なり，アンタイプドオブジェクトラッパーメソッドは，スタブまたはオブジェクトインプリメンテーションが受け取る引数を受け取りません。また，スタブやオブジェクトインプリメンテーションの起動を防ぐこともできません。

図 21-1 に，クライアントスタブメソッドの前にアンタイプドオブジェクトラッパーの pre_method がどのように起動されるのか，そのあと post_method がどのように起動されるのかを示します。また，オブジェクトインプリメンテーションに関するサーバ側での起動シーケンスも示します。

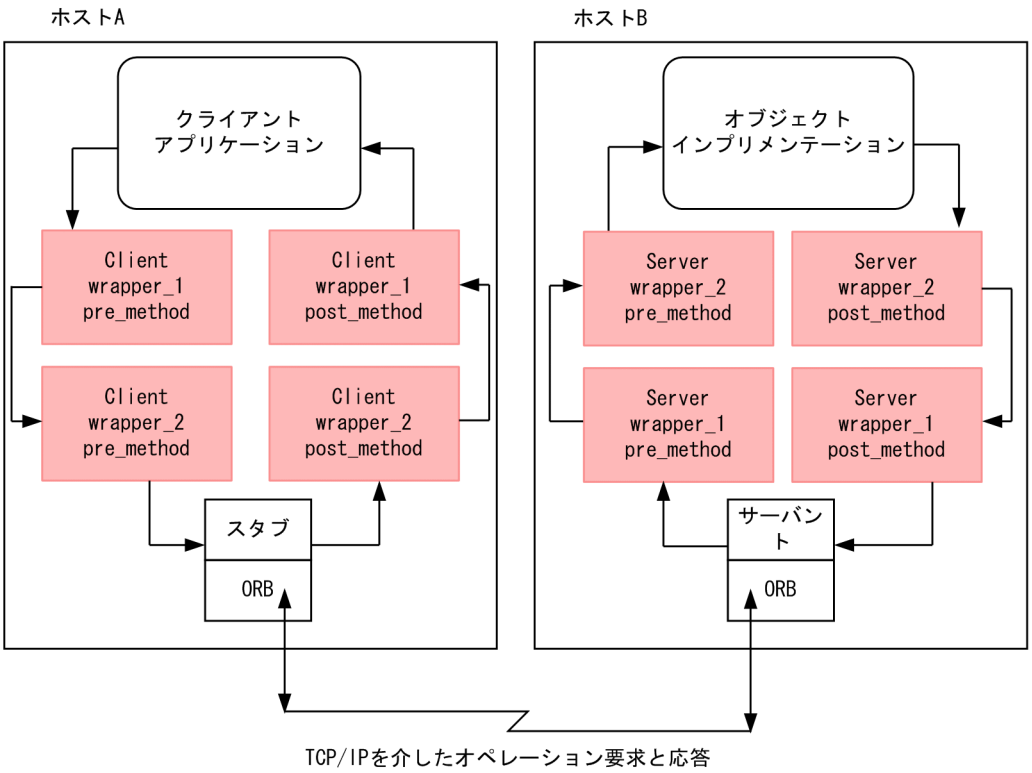
図 21-1　単一のアンタイプドオブジェクトラッパー



21.2.1　複数のアンタイプドオブジェクトラッパーの使用

複数のアンタイプドオブジェクトラッパーを使用した場合の流れを図 21-2 に示します。

図 21-2　複数のアンタイプドオブジェクトラッパー



21.2.2　pre_method 起動の順序

クライアントがバインドされたオブジェクトに対するメソッドを呼び出す場合，各アンタイプドオブジェクトラッパー pre_method は，クライアントのスタブルーチンが呼び出される前に制御を受け取ります。サーバがオペレーション要求を受信する場合，各アンタイプドオブジェクトラッパー pre_method は，オブジェクトインプリメンテーションが制御を受け取る前に起動されます。どちらの場合も，最初に制御を受け取る pre_method は，「最初に登録された」オブジェクトラッパーに属する pre_method です。

21.2.3　post_method 起動の順序

サーバのオブジェクトインプリメンテーションが処理を終了すると，応答がクライアントに送信される前に各 post_method が起動されます。クライアントがオペレーション要求に対する応答を受信すると，制御がクライアントに返される前に各 post_method が起動されます。どちらの場合も，最初に制御を受け取る post_method は，「最後に登録された」オブジェクトラッパーに属する post_method です。

注

タイプドオブジェクトラッパーおよびアンタイプドオブジェクトラッパーの両方を使うことを選択した場合，起動順序については「21.6　タイプドおよびアンタイプドオブジェクトラッパーの混在使用」を参照してください。

21.3　アンタイプドオブジェクトラッパーの使用

アンタイプドオブジェクトラッパーを使う際は次の手順に従ってください。各手順については以降で順に説明します。

1. アンタイプドオブジェクトラッパーを生成したい，一つまたは複数のインタフェースを指定してください。
2. 次のコンパイラに-obj_wrapper オプションを指定して，IDL 指定からコードを生成してください。
   - C++の場合
     idl2cpp
   - Java の場合
     idl2java
3. 次のクラスから派生する，アンタイプドオブジェクトラッパーファクトリのインプリメンテーションを生成してください。
   - C++の場合
     VISObjectWrapper::UntypedObjectWrapperFactory
   - Java の場合
     UntypedObjectWrapperFactory
4. クラスから派生する，アンタイプドオブジェクトラッパーのインプリメンテーションを生成してください。
   - C++の場合
     VISObjectWrapper::UntypedObjectWrapper
   - Java の場合
     UntypedObjectWrapper
5. Java の場合，クライアントまたはサーバアプリケーションを変更して，適切な型の ChainUntypedObjectWrapperFactory にアクセスしてください。
6. アプリケーションを変更して，アンタイプドオブジェクトラッパーファクトリを生成してください。
7. Java の場合，ChainUntypedObjectWrapperFactory の add メソッドを使って，ファクトリをチェーンに加えてください。

21.3.1　アンタイプドオブジェクトラッパーファクトリのインプリメント

C++の場合

objectWrappers サンプルアプリケーションの一部である TimeWrap.h ファイルは，VISObjectWrapper::UntypedObjectWrapperFactory から派生したアンタイプドオブジェクトラッパーファクトリの定義方法を示します。

コードサンプル 21-1 に，TimingObjectWrapperFactory を示します。これはメソッド呼び出しのタイミング情報を表示するアンタイプドオブジェクトラッパーを生成するために使用します。key パラメタを TimingObjectWrapperFactory コンストラクタに追加することに注意してください。また，このパラメタはオブジェクトラッパーを識別するためにサービスイニシャライザが使用します。

コードサンプル 21-1　TimeWrap.h ファイルから派生する TimingObjectWrapperFactory インプリメンテーション（C++）

```
class TimingObjectWrapperFactory
   : public VISObjectWrapper::UntypedObjectWrapperFactory
{
   public:
      TimingObjectWrapperFactory(VISObjectWrapper::
            Location loc, const char* key)
         : VISObjectWrapper::
            UntypedObjectWrapperFactory(loc), _key(key) {}

      // ObjectWrapperFactory operations
      VISObjectWrapper::UntypedObjectWrapper_ptr create (
               CORBA::Object_ptr target,
               VISObjectWrapper::Location loc) {
         if (_owrap == NULL) {
               _owrap = new TimingObjectWrapper(_key);
         }
         return VISObjectWrapper::UntypedObjectWrapper::
               _duplicate(_owrap);
   }

   private:
      CORBA::String_var _key;
      VISObjectWrapper::UntypedObjectWrapper_var _owrap;
};
```

Java の場合

objectWrappers サンプルアプリケーションの一部である TimingUntypedObjectWrapperFactory のインプリメンテーションは，UntypedObjectWrapperFactory から派生したアンタイプドオブジェクトラッパーファクトリの定義方法を示します。

クライアントがオブジェクトにバインドする際，またはサーバがオブジェクトインプリメンテーションでメソッドを起動する際はいつも，ユーザのファクトリの create メソッドが呼び出され，アンタイプドオブジェクトラッパーを生成します。create メソッドは目的のオブジェクトを受け取ります。つまりユーザは，無視したいオブジェクトタイプについて，ファクトリがアンタイプドオブジェクトラッパーを作らないようにファクトリを設計できます。また，生成されたオブジェクトラッパーがサーバ側オブジェクトインプリメンテーションなのか，クライアント側オブジェクトなのかを指定する enum も受け取ります。

コードサンプル 21-2 に，TimingObjectWrapperFactory を示します。これはメソッド呼び出しのタイミング情報を表示するアンタイプドオブジェクトラッパーを生成するために使用されます。

コードサンプル 21-2　TimingUntypedObjectWrapperFactory インプリメンテーション

```
package UtilityObjectWrappers;
import com.inprise.vbroker.interceptor.*;

public class TimingUntypedObjectWrapperFactory implements
      UntypedObjectWrapperFactory {
   public UntypedObjectWrapper create(
         org.omg.CORBA.Object target,
         com.inprise.vbroker.interceptor.Location loc) {
      return new TimingUntypedObjectWrapper();
   }
}
```

21.3.2　アンタイプドオブジェクトラッパーのインプリメント

(1)　C++の場合

コードサンプル 21-3 に，TimeWrap.h ファイルに定義された TimingUntypedObjectWrapper のインプリメンテーションを示します。アンタイプドオブジェクトラッパーは VISObjectWrapper::UntypedObjectWrapper クラスから派生していなければなりません。ユーザはア

ンタイプドオブジェクトラッパー中の pre_method と post_method の両メソッドのインプリメンテーションを提供できます。

ファクトリがインストールされたら，ファクトリのコンストラクタによって自動的にまたは VISObjectWrapper::ChainUntypedObjectWrapperFactory::create メソッドの起動によって手動で，クライアントがオブジェクトにバインドする際，またはサーバがオブジェクトインプリメンテーションでメソッドを起動する際に，アンタイプドオブジェクトラッパーオブジェクトが自動的に生成されます。

コードサンプル 21-3 に示す pre_method は，TimeWrap.C で定義された TimerBegin メソッドを起動します。このメソッドは，現在の時間を取得するために Closure オブジェクトを使用します。同様に，post_method は TimerDelta メソッドを起動し，pre_method が呼び出されてからどれくらい時間がたったかを調べ，経過時間を出力します。

コードサンプル 21-3　TimingUntypedObjectWrapper インプリメンテーション（C++）

```
class TimingObjectWrapper : public VISObjectWrapper::
      UntypedObjectWrapper {
   public:
      TimingObjectWrapper(
            const char* key=NULL) : _key(key) {}

      void pre_method(const char* operation,
            CORBA::Object_ptr target,
            VISClosure& closure) {
         cout << "*Timing: [" << flush;
         if ((char *)_key)
            cout << _key << flush;
         else
            cout << "<no key>" << flush;
         cout << "] pre_method¥t" << operation << "¥t->"
            << endl;
         TimerBegin(closure, operation);
      }

      void post_method(const char* operation,
            CORBA::Object_ptr target,
            CORBA::Environment& env,
            VISClosure& closure) {
         cout << "*Timing: [" << flush;
         if ((char *)_key)
            cout << _key << flush;
         else
            cout << "<no key>" << flush;
         cout << "] post_method¥t" ;
         TimerDelta(closure, operation);
      }
   private:
      CORBA::String_var _key;
};
```

(2)　Java の場合

コードサンプル 21-4 に，TimingUntypedObjectWrapper のインプリメンテーションを示します。アンタイプドオブジェクトラッパーは UntypedObjectWrapper クラスから派生していなければなりません。ユーザはアンタイプドオブジェクトラッパー中の pre_method と post_method の両メソッドのインプリメンテーションを提供できます。

ファクトリがインストールされたら，ファクトリのコンストラクタによって自動的にまたは ChainUntypedObjectWrapperFactory::add メソッドの起動によって手動で，クライアントがオブジェクトにバインドする際，またはサーバがオブジェクトインプリメンテーションでメソッドを起動する際に，アンタイプドオブジェクトラッパーオブジェクトが自動的に生成されます。

コードサンプル 21-4 に示す pre_method は，現在の時間を取得し，それをプライベート変数に保存し，メッセージを出力します。同様に，post_method も現在の時間を取得し，pre_method が呼び出されてからどれくらい時間がたったかを調べ，経過時間を出力します。

コードサンプル 21-4　TimingUntypedObjectWrapper インプリメンテーション（Java）

```
package UtilityObjectWrappers;
import com.inprise.vbroker.interceptor.*;

public class TimingUntypedObjectWrapper implements
                                    UntypedObjectWrapper {
   private long time;
   public void pre_method(String operation,
         org.omg.CORBA.Object target,
         Closure closure) {
      System.out.println("Timing: " +
               ((com.inprise.vbroker.CORBA.Object) target).
               _object_name() + "->" + operation + "()");
      time = System.currentTimeMillis();
   }
   public void post_method(String operation,
         org.omg.CORBA.Object target,
         org.omg.CORBA.Environment env,
         Closure closure) {
      long diff = System.currentTimeMillis() - time;
      System.out.println("Timing: Time for call ¥t" +
         ((com.inprise.vbroker.CORBA.Object)target).
          _object_name() + "->" + operation + "() = " +
          diff + " ms.");
   }
}
```

(3)　pre_method および post_method パラメタ

pre_method と post_method は表 21-2 のパラメタを受け取ります。

表 21-2　pre_method および post_method メソッドの共通引数

パラメタ	説明
operation	目的のオブジェクトでリクエストしたオペレーションの名前
target	目的のオブジェクト
closure	このオブジェクトラッパーのメソッド呼び出し用にデータを保存する領域

post_method は Environment パラメタも受け取ります。Environment パラメタは，メソッド呼び出しの前の手順で起こる可能性のある例外をユーザに通知する際に使えます。

21.3.3　アンタイプドオブジェクトラッパーファクトリの生成と登録

C++の場合

アンタイプドオブジェクトラッパーファクトリは，ロケーションを受け付ける base クラスコンストラクタで生成される場合，アンタイプドオブジェクトラッパーのチェーンに自動的に追加されます。

クライアント側では，オブジェクトがバインドされる前にアンタイプドオブジェクトラッパーファクトリが生成，登録される場合だけ，オブジェクトがラッピングされます。サーバ側では，オブジェクトインプリメンテーションが呼び出される前にアンタイプドオブジェクトラッパーファクトリが生成，登録されます。

コードサンプル 21-5 に，クライアント用の二つのアンタイプドオブジェクトラッパーファクトリの生成と自動登録を示した UntypedClient.C サンプルファイルの一部を示します。ファクトリは

VisiBroker ORB が初期化されたあと，かつクライアントがオブジェクトにバインドする前に生成されます。

コードサンプル 21-5　二つのクライアント側アンタイプドオブジェクトラッパーファクトリの生成と登録 (C++)

```
int main(int argc, char* const* argv) {
   try {
         // Initialize the ORB.
         CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
         // Install untyped object wrappers
         TimingObjectWrapperFactory timingfact(
               VISObjectWrapper::Client, "timeclient");
         TraceObjectWrapperFactory tracingfact(
               VISObjectWrapper::Client, "traceclient");
         // Now locate an account manager.
. . .
```

Java の場合

コードサンプル 21-6 に，クライアント用の二つのアンタイプドオブジェクトラッパーファクトリの生成とインストールを示した UntypedClient.java サンプルファイルの一部を示します。ファクトリは VisiBroker ORB が初期化されたあと，ただしクライアントがオブジェクトにバインドする前に生成されます。

コードサンプル 21-6　二つのクライアント側アンタイプドオブジェクトラッパーファクトリのインストール（Java)

```
// UntypedClient.java
import com.inprise.vbroker.interceptor.*;

public class UntypedClient {
   public static void main(String[ ] args) throws Exception {
      // Initialize the ORB.
      org.omg.CORBA.ORB orb =
                        org.omg.CORBA.ORB.init(args,null);
      doMain (orb, args);
   }
   public static void doMain(org.omg.CORBA.ORB orb,
                          String[ ] args) throws Exception {
      ChainUntypedObjectWrapperFactory Cfactory =
            ChainUntypedObjectWrapperFactoryHelper.narrow(
               orb.resolve_initial_references(
               "ChainUntypedObjectWrapperFactory")
            );
      Cfactory.add(new UtilityObjectWrappers.
            TimingUntypedObjectWrapperFactory(),
            Location.CLIENT);
      Cfactory.add(new UtilityObjectWrappers.
            TracingUntypedObjectWrapperFactory(),
            Location.CLIENT);
      // Locate an account manager. . . .
   }
}
```

C++の場合

コードサンプル 21-7 に UntypedServer.C サンプルファイルを示します。このファイルはサーバ用のアンタイプドオブジェクトラッパーファクトリの生成と登録を示しています。ファクトリは VisiBroker ORB が初期化されたあと，ただしオブジェクトインプリメンテーションが生成される前に生成されます。

コードサンプル 21-7　サーバ側アンタイプドオブジェクトラッパーファクトリの登録（C++)

```
// UntypedServer.C
#include "Bank_s.hh"
#include "BankImpl.h"
#include "TimeWrap.h"
#include "TraceWrap.h"
USE_STD_NS
int main(int argc, char* const* argv) {
```

```
   try {
      // Initialize the ORB.
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      // Initialize the POA.
      CORBA::Object_var obj =
            orb->resolve_initial_references("RootPOA");
      PortableServer::POA_var rootPoa =
            PortableServer::POA::_narrow(obj);
      CORBA::PolicyList policies;
      policies.length(1);
      policies[(CORBA::ULong)0] =
            rootPoa->create_lifespan_policy(
            PortableServer::PERSISTENT);
      // Get the POA Manager.
      PortableServer::POAManager_var poa_manager =
            rootPoa->the_POAManager();
      // Create myPOA With the Right Policies.
      PortableServer::POA_var myPOA =
            rootPoa->create_POA("bank_ow_poa",
            poa_manager,
            policies);
      // Install Untyped Object Wrappers for Account Manager.
      TimingObjectWrapperFactory timingfact(
           VISObjectWrapper::Server, "timingserver");
      TraceObjectWrapperFactory tracingfact(
           VISObjectWrapper::Server, "traceserver");
      // Create the Account Manager Servant.
      AccountManagerImpl managerServant;
      // Decide on ID for Servant.
      PortableServer::ObjectId_var managerId =
      PortableServer::string_to_ObjectId("BankManager");
      // Activate the Servant with the ID on myPOA.
      myPOA->activate_object_with_id(
            managerId, &managerServant);
      // Activate the POA Manager.
      rootPoa->the_POAManager()->activate();
      cout << "Manager is ready." << endl;
      // Wait for Incoming Requests.
      orb->run();
   } catch(const CORBA::Exception& e) {
      cerr << e << endl;
      return 1;
   }
   return 0;
}
```

Java の場合

コードサンプル 21-8 に UntypedServer.java サンプルファイルを示します。このファイルはサーバ用のアンタイプドオブジェクトラッパーファクトリの生成と登録を示しています。ファクトリは VisiBroker ORB が初期化されたあと，ただしオブジェクトインプリメンテーションが生成される前に生成されます。

コードサンプル 21-8　サーバ側アンタイプドオブジェクトラッパーファクトリのインストール（Java）

```
// UntypedServer.java
import com.inprise.vbroker.interceptor.*;
import org.omg.PortableServer.*;
import com.inprise.vbroker.PortableServerExt.
                                BindSupportPolicyValue;
import com.inprise.vbroker.PortableServerExt.
                                BindSupportPolicyValueHelper;
import com.inprise.vbroker.PortableServerExt.
                                BIND_SUPPORT_POLICY_TYPE;

public class UntypedServer {
   public static void main(String[ ] args) throws Exception {
      // Initialize the ORB.
      org.omg.CORBA.ORB orb =
                           org.omg.CORBA.ORB.init(args,null);
      ChainUntypedObjectWrapperFactory Sfactory =
            ChainUntypedObjectWrapperFactoryHelper.narrow
            (orb.resolve_initial_references(
            "ChainUntypedObjectWrapperFactory"));
```

```
        Sfactory.add(new UtilityObjectWrappers.
              TracingUntypedObjectWrapperFactory(),
              Location.SERVER);
        // get a reference to the rootPOA
        POA rootPOA = POAHelper.
             narrow(orb.resolve_initial_references("RootPOA"));
        // Create a BindSupport Policy that makes POA register
        // each servant with osagent
        org.omg.CORBA.Any any = orb.create_any();
        BindSupportPolicyValueHelper.insert(any,
           BindSupportPolicyValue.BY_INSTANCE);
        org.omg.CORBA.Policy bsPolicy =
           orb.create_policy(BIND_SUPPORT_POLICY_TYPE.value,
                             any);
        // Create policies for our testPOA
        org.omg.CORBA.Policy[ ] policies = {
           rootPOA.create_lifespan_policy
              (LifespanPolicyValue.PERSISTENT), bsPolicy
        };
        // Create myPOA with the right policies
        POA myPOA = rootPOA.create_POA( "bank_agent_poa",
           rootPOA.the_POAManager(),
           policies );

        // Create the account manager object.
        AccountManagerImpl managerServant =
                                    new AccountManagerImpl();
        // Decide on the ID for the servant
        byte[ ] managerId = "BankManager".getBytes();
        // Activate the servant with the ID on myPOA
        myPOA.activate_object_with_id(managerId,
                                    managerServant);
        // Activate the POA manager
        rootPOA.the_POAManager().activate();
        System.out.println(
                      "AccountManager: BankManager is ready.");
        for( int i = 0; i < args.length; i++ ) {
           if( args[i].equalsIgnoreCase("-runCoLocated") ) {
              if( args[i+1].equalsIgnoreCase("Client") ) {
                 Client.doMain(orb, new String[0]);
              }else if( args[i+1].
                           equalsIgnoreCase("TypedClient") ) {
                 TypedClient.doMain(orb, new String[0]);
              }
              if( args[i+1].equalsIgnoreCase("UntypedClient") ) {
                 UntypedClient.doMain(orb, new String[0]);
              }
              System.exit(1);
           }
        }
        // Wait for incoming requests
        orb.run();
     }
  }
```

21.3.4　アンタイプドオブジェクトラッパーの削除

VISObjectWrapper::ChainUntypedObjectWrapperFactory クラスの remove メソッド（C++）または ChainUntypedObjectWrapperFactory クラスの remove メソッド（Java）を使って，クライアントまたはサーバアプリケーションからアンタイプドオブジェクトラッパーファクトリを削除できます。ファクトリを削除する場合は，ロケーションを指定しなければなりません。これは，VISObjectWrapper::Both（C++）または Both（Java）のロケーションでファクトリを追加した場合に，Client ロケーション，Server ロケーション，または両方のロケーションからファクトリを削除することを選べるということです。

注

クライアントから一つまたは複数のオブジェクトラッパーファクトリを削除しても，クライアントがすでにバインドしたクラスのオブジェクトは影響を受けません。あとでバインドされるオブジェクトだけが影響を受けます。サーバからオブジェクトラッパーファクトリを削除しても，すでに生成されたオ

ブジェクトインプリメンテーションは影響を受けません。あとで生成されるオブジェクトインプリメンテーションだけが影響を受けます。

21.4　タイプドオブジェクトラッパー

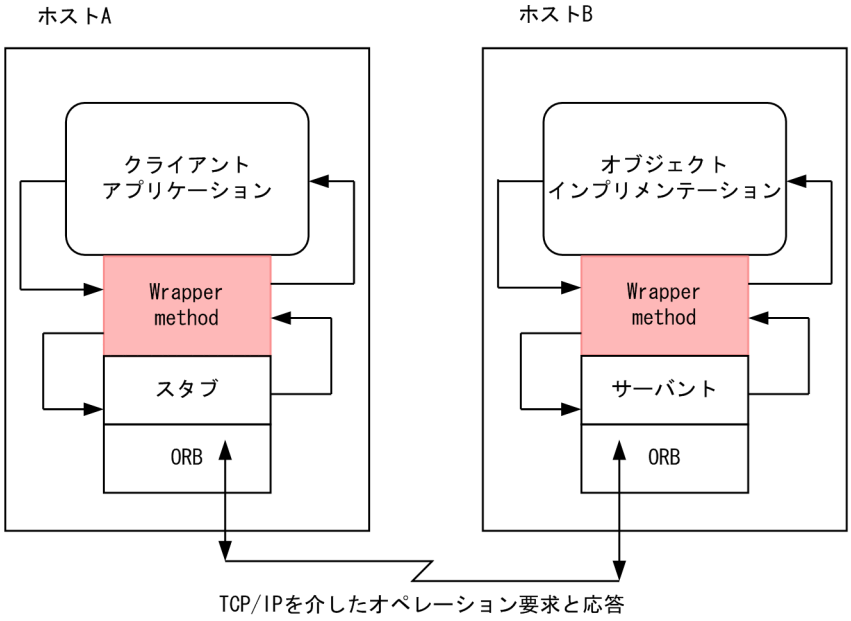
ある特定のクラスのタイプドオブジェクトラッパーをインプリメントする際，バインドされたオブジェクトでメソッドが呼び出されるときに起こる処理を定義してください。図 21-3 にクライアントスタブクラスメソッドの前にクライアントのオブジェクトラッパーメソッドがどのように呼び出されるのか，またサーバのインプリメンテーションメソッドの前にサーバ側のオブジェクトラッパーがどのように起動されるのかを示します。

注

タイプドオブジェクトラッパーインプリメンテーションは，ラップするオブジェクトが提供するメソッドすべてをインプリメントする必要はありません。

同じクライアントまたはサーバアプリケーション内で，タイプドオブジェクトラッパーおよびアンタイプドオブジェクトラッパーの両方を混在させて使用できます。詳細については，「21.6　タイプドおよびアンタイプドオブジェクトラッパーの混在使用」を参照してください。

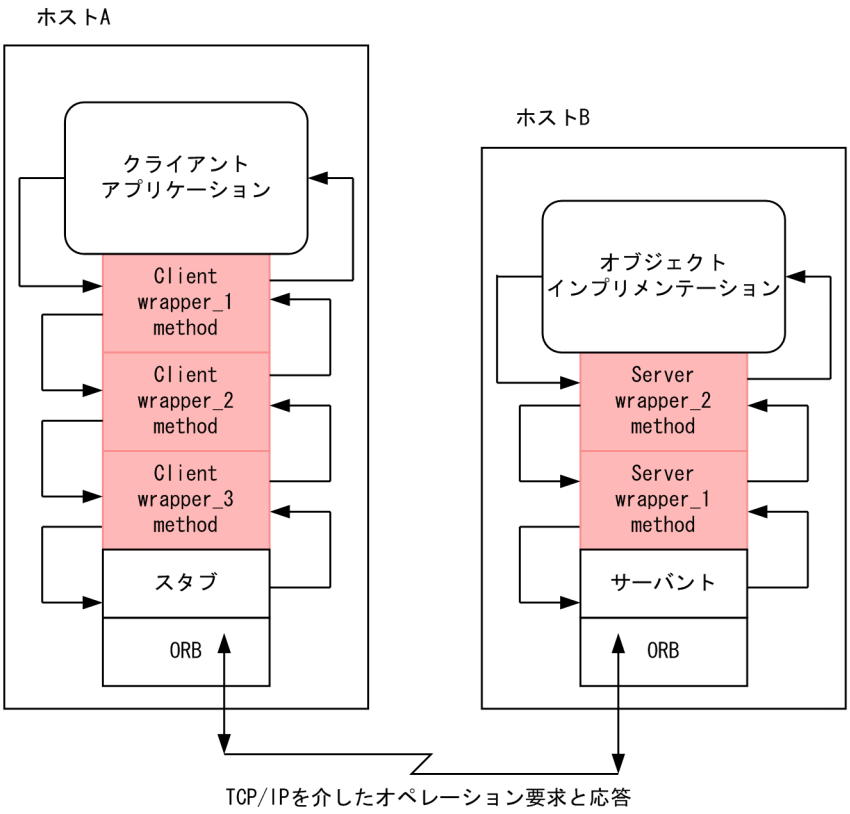
図 21-3　登録された単一のタイプドオブジェクトラッパー



21.4.1　複数のタイプドオブジェクトラッパーの使用

図 21-4 に示すように，ある特定のクラスのオブジェクトについて複数のタイプドオブジェクトラッパーをインプリメントし登録する場合があるでしょう。クライアント側では，最初に登録されたオブジェクトラッパーは client_wrapper_1 だったので，そのメソッドが最初に制御を受け取ります。処理を終えたあと，client_wrapper_1 メソッドはチェーンの中にある次のオブジェクトのメソッドに制御を渡すか，クライアントに制御を返します。サーバ側では，最初に登録されたオブジェクトラッパーは server_wrapper_1 だったので，そのメソッドが最初に制御を受け取ります。処理を終えたあと，server_wrapper_1 メソッドはチェーンの中にある次のオブジェクトのメソッドに制御を渡すか，サーバントに制御を返します。

図 21-4　登録された複数のタイプドオブジェクトラッパー



21.4.2　起動の順序

ある特定のクラス用に登録されたタイプドオブジェクトラッパーのメソッドは，通常，クライアント側のスタブメソッドに渡されるか，サーバ側のサーバントに渡される引数すべてを受け取ります。各オブジェクトラッパーメソッドは，親クラスのメソッド<interface_name>ObjectWrapper::<method_name>（C++）または super.<method_name>（Java）を起動して，チェーンの中にある次のオブジェクトラッパーメソッドに制御を渡します。オブジェクトラッパーがチェーンの中にある次のオブジェクトラッパーメソッドを呼び出さないで制御を返したい場合は，適切なリターン値とともに返されます。

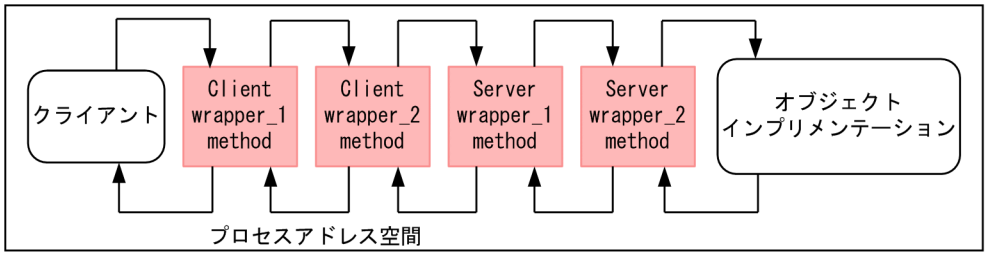
タイプドオブジェクトラッパーメソッドがチェーンの中にある前のメソッドに制御を返せることで，ユーザはクライアントスタブやオブジェクトインプリメンテーションを起動しないオブジェクトラッパーメソッドを作れます。例えば，頻繁に要求されるオペレーションの結果をキャッシュするオブジェクトラッパーメソッドを作れます。この場合，バインドされたオブジェクトに対するメソッドの最初の呼び出しによって，オペレーション要求がオブジェクトインプリメンテーションに送信されます。制御がオブジェクトラッパーメソッドによって戻る場合，結果は保存されます。同じメソッドの後続の呼び出しでは，オブジェクトラッパーメソッドは，オブジェクトインプリメンテーションにオペレーション要求を実際には発行しないで，キャッシュされた結果を返せます。

タイプドオブジェクトラッパーとアンタイプドオブジェクトラッパーの両方を使う選択をした場合，起動順序については「21.6　タイプドおよびアンタイプドオブジェクトラッパーの混在使用」を参照してください。

21.4.3　タイプドオブジェクトラッパーおよび同一プロセスにあるクライアントとサーバ

クライアントとサーバが同じプロセス中にパッケージされている場合，最初に制御を受け取るオブジェクトラッパーメソッドは，最初にインストールされたクライアント側オブジェクトラッパーに属します。図 21-5 に起動順序を示します。

図 21-5　タイプドオブジェクトラッパーの起動順序



21.5　タイプドオブジェクトラッパーの使用

タイプドオブジェクトラッパーを使う場合は次の手順に従ってください。各手順については以降で順に説明します。

1. タイプドオブジェクトラッパーを生成したい，一つまたは複数のインタフェースを指定してください。
2. 次のコンパイラに-obj_wrapper オプションを指定して，IDL からコードを生成してください。
   - C++の場合
     idl2cpp
   - Java の場合
     idl2java
3. タイプドオブジェクトラッパークラスを，idl2cpp コンパイラ（C++）または idl2java コンパイラ（Java）で生成した<interface_name>ObjectWrapper クラスから派生させ，ラッピングしたいメソッドのインプリメンテーションを作成してください。
4. アプリケーションを変更して，タイプドオブジェクトラッパーを登録してください。

21.5.1　タイプドオブジェクトラッパーのインプリメント

タイプドオブジェクトラッパーを，idl2cpp コンパイラ（C++）または idl2java コンパイラ（Java）で生成した<interface_name>ObjectWrapper クラスから派生させてください。コードサンプル 21-10 に Account インタフェース用タイプドオブジェクトラッパーのインプリメンテーション（Java）を示します。このクラスは AccountObjectWrapper インタフェースから派生していて，balance メソッドの単純なキャッシングインプリメンテーションを提供していることに注意してください。balance メソッドは次のような手順で処理を実行します。

1. 次のフラグをチェックし，このメソッドが以前に呼び出されたことがあるかどうかを調べます。
   - C++の場合
     _inited
   - Java の場合
     _initialized
2. 今回が最初の呼び出しの場合，チェーンの中にある次のオブジェクトで balance メソッドを呼び出し，結果を_balance に保存し，_inited（C++）または_initialized（Java）を true に設定し，値を返します。
3. このメソッドがすでに呼び出されたことがある場合，キャッシュされた値を返します。

コードサンプル 21-9　CachingAccountObjectWrapper インプリメンテーションの一部（C++）

```
class CachingAccountObjectWrapper : public Bank::
      AccountObjectWrapper {
   public:
      CachingAccountObjectWrapper() :
            _inited((CORBA::Boolean)0) {}
      CORBA::Float balance() {
         cout << "+CachingAccountObjectWrapper:
               Before Calling Balance" << endl;
      if (! _inited) {
         _balance = Bank::AccountObjectWrapper::balance();
         _inited = 1;
      } else {
         cout << "+ CachingAccountObjectWrapper:
               Returning Cached Value" << endl;
      }
```

```
        cout << "+ CachingAccountObjectWrapper:
              After Calling Balance" << endl;
        return _balance;
    }
    . . .
};
```

コードサンプル 21-10　CachingAccountObjectWrapper インプリメンテーションの一部（Java）

```
package BankWrappers;
public class CachingAccountObjectWrapper extends
                                Bank.AccountObjectWrapper {
   private boolean _initialized = false;
   private float _balance;
   public float balance() {
      System.out.println(
      "+ CachingAccountObjectWrapper: Before calling balance: ");
      try {
         if( !_initialized ) {
            _balance = super.balance();
            _initialized = true;
         } else {
          System.out.println(
          "+ CachingAccountObjectWrapper: Returning Cached value");
         }
         return _balance;
      }finally {
       System.out.println(
       "+ CachingAccountObjectWrapper: After calling balance: ");
      }
   }
}
```

21.5.2　クライアント用タイプドオブジェクトラッパーの登録

C++の場合

タイプドオブジェクトラッパーは，idl2cpp コンパイラがクラス中に生成した<interface_name>::add メソッドを起動して，クライアント側に登録されます。クライアント側オブジェクトラッパーは ORB_init メソッドが呼び出されたあと，ただしオブジェクトがバインドされる前に登録しなければなりません。コードサンプル 21-11 に，タイプドオブジェクトラッパーを生成，登録する TypedClient.C ファイルの一部を示します。

コードサンプル 21-11　クライアント側タイプドオブジェクトラッパーの生成と登録（C++）

```
. . .
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

      //Install Typed Object Wrappers for Account.
      Bank::AccountObjectWrapper::add(orb,
            CachingAccountObjectWrapper::factory,
            VISObjectWrapper::Client);

      // Get the Manager ID.
      PortableServer::ObjectId_var managerId =
            PortableServer::string_to_ObjectId("BankManager");

      // Locate an Account Manager.
      Bank::AccountManager_var manager =
            Bank::AccountManager::_bind(
                  "/bank_ow_poa", managerId);
. . .
```

Java の場合

タイプドオブジェクトラッパーは，idl2java コンパイラが Helper クラス中に生成した addClientObjectWrapperClass メソッドを起動して，クライアント側に登録されます。クライアント側オブジェクトラッパーは ORB.init メソッドが呼び出されたあと，ただしオブジェクトがバインドさ

れる前に登録しなければなりません。コードサンプル 21-12 に，タイプドオブジェクトラッパーを生成，登録する TypedClient.java ファイルの一部を示します。

コードサンプル 21-12　クライアント側タイプドオブジェクトラッパーのインストール（Java）

```
// TypedClient.java
import com.inprise.vbroker.interceptor.*;
public class TypedClient {
   public static void main(String[ ] args) throws Exception {
      // Initialize the ORB.
      org.omg.CORBA.ORB orb =
                          org.omg.CORBA.ORB.init(args,null);
      doMain (orb, args);
   }
   public static void doMain(org.omg.CORBA.ORB orb,
                                              String[ ] args) {
      // Add a typed object wrapper for Account objects
      Bank.AccountHelper.addClientObjectWrapperClass(orb,
         BankWrappers.CachingAccountObjectWrapper.class);
      // Locate an account manager.
      Bank.AccountManager manager =
          Bank.AccountManagerHelper.bind(orb, "BankManager");
      . . .
   }
}
```

VisiBroker ORB は，クライアント側でこのクラスのために登録されたすべてのオブジェクトラッパーの動作の記録を採っています。クライアントが_bind メソッドを起動して，その型のオブジェクトにバインドしようとすると，必要なオブジェクトラッパーが生成されます。クライアントが特定のクラスのオブジェクトの複数のインスタンスにバインドする場合，各インスタンスは，それぞれ専用のオブジェクトラッパーのセットを持ちます。

21.5.3　サーバ用タイプドオブジェクトラッパーの登録

C++の場合

クライアントアプリケーションと同様，タイプドオブジェクトラッパーをサーバ側に登録するには，<interface_name>::add メソッドを呼び出します。サーバ側タイプドオブジェクトラッパーは ORB_init メソッドが呼び出されたあと，ただしオブジェクトインプリメンテーションがリクエストを処理する前に登録されなければなりません。コードサンプル 21-13 にタイプドオブジェクトラッパーを登録する TypedServer.C ファイルの部分を示します。

コードサンプル 21-13　サーバ側タイプドオブジェクトラッパーの登録（C++）

```
// TypedServer.C
#include "Bank_s.hh"
#include "BankImpl.h"
#include "BankWrap.h"
USE_STD_NS
int main(int argc, char* const* argv) {
   try {
      // Initialize the ORB.
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      // Initialize the POA.
      CORBA::Object_var obj =
            orb->resolve_initial_references("RootPOA");
      PortableServer::POA_var rootPoa =
            PortableServer::POA::_narrow(obj);
      CORBA::PolicyList policies;
      policies.length(1);
      policies[(CORBA::ULong)0] =
           rootPoa->create_lifespan_policy(
           PortableServer::PERSISTENT);
      // Get the POA Manager.
      PortableServer::POAManager_var poa_manager =
           rootPoa->the_POAManager();
      // Create myPOA With the Right Policies.
      PortableServer::POA_var myPOA = rootPoa->create_POA(
```

```
                "bank_ow_poa",
            poa_manager,
            policies);
        // Install Typed Object Wrappers for Account Manager.
        Bank::AccountManagerObjectWrapper::add(orb,
            SecureAccountManagerObjectWrapper::factory,
            VISObjectWrapper::Server);
        Bank::AccountManagerObjectWrapper::add(orb,
            CachingAccountManagerObjectWrapper::factory,
                VISObjectWrapper::Server);
        // Create the Account Manager Servant.
        AccountManagerImpl managerServant;
        // Decide on ID for Servant.
        PortableServer::ObjectId_var managerId =
            PortableServer::string_to_ObjectId("BankManager");
        // Activate the Servant with the ID on myPOA.
        myPOA->activate_object_with_id(
            managerId, &managerServant);
        // Activate the POA Manager.
        rootPoa->the_POAManager()->activate();
        cout << "Manager is ready." << endl;
        // Wait for Incoming Requests.
        orb->run();
    } catch(const CORBA::Exception& e) {
        cerr << e << endl;
        return 1;
    }
    return 0;
}
```

Java の場合

クライアントアプリケーションと同様，タイプドオブジェクトラッパーをサーバ側に登録するには，Helper クラスが提供する addServerObjectWrapperClass メソッドを呼び出します。サーバ側タイプドオブジェクトラッパーは ORB.init メソッドが呼び出されたあと，ただしオブジェクトインプリメンテーションがリクエストを処理する前に登録されなければなりません。コードサンプル 21-14 にタイプドオブジェクトラッパーを登録する TypedServer.java ファイルの部分を示します。

コードサンプル 21-14　サーバ側タイプドオブジェクトラッパーのインストール（Java）

```
// TypedServer.java
import org.omg.PortableServer.*;
import com.inprise.vbroker.PortableServerExt.
                                BindSupportPolicyValue;
import com.inprise.vbroker.PortableServerExt.
                                BindSupportPolicyValueHelper;
import com.inprise.vbroker.PortableServerExt.
                                BIND_SUPPORT_POLICY_TYPE;

public class TypedServer {
   public static void main(String[ ] args) throws Exception {
      // Initialize the ORB.
      org.omg.CORBA.ORB orb =
                            org.omg.CORBA.ORB.init(args,null);
      // Add two typed object wrappers for AccountManager
      // objects
      Bank.AccountManagerHelper.addServerObjectWrapperClass
       (orb, BankWrappers.SecureAccountManagerObjectWrapper.
        class);
      Bank.AccountManagerHelper.addServerObjectWrapperClass
       (orb, BankWrappers.CachingAccountManagerObjectWrapper.
        class);
      // get a reference to the rootPOA
      POA rootPOA = POAHelper.narrow
                (orb.resolve_initial_references("RootPOA"));
      // Create a BindSupport Policy that makes POA register
      // each servant with osagent
      org.omg.CORBA.Any any = orb.create_any();
      BindSupportPolicyValueHelper.insert(any,
                          BindSupportPolicyValue.BY_INSTANCE);
      org.omg.CORBA.Policy bsPolicy =
         orb.create_policy(BIND_SUPPORT_POLICY_TYPE.value,
                           any);
```

```
        // Create policies for our testPOA
        org.omg.CORBA.Policy[ ] policies = {
           rootPOA.create_lifespan_policy(LifespanPolicyValue.
           PERSISTENT), bsPolicy
        };
        // Create myPOA with the right policies
        POA myPOA = rootPOA.create_POA("lilo",
                          rootPOA.the_POAManager(), policies);
        // Create the account manager object.
        AccountManagerImpl managerServant =
                                    new AccountManagerImpl();
        // Decide on the ID for the servant
        byte[ ] managerId = "BankManager".getBytes();
        // Activate the servant with the ID on myPOA
        myPOA.activate_object_with_id(managerId, managerServant);
        // Activate the POA manager
        rootPOA.the_POAManager().activate();
        System.out.println(
                        "AccountManager: BankManager is ready.");
        for( int i = 0; i < args.length; i++ ) {
           if ( args[i].equalsIgnoreCase("-runCoLocated") ) {
              if( args[i+1].equalsIgnoreCase("Client") ) {
                 Client.doMain(orb, new String[0]);
              } else if( args[i+1].
                            equalsIgnoreCase("TypedClient") ) {
                 TypedClient.doMain(orb, new String[0]);
              }
              if( args[i+1].equalsIgnoreCase("UntypedClient") ) {
                 UntypedClient.doMain(orb, new String[0]);
              }
              System.exit(1);
           }
        }
        // Wait for incoming requests
        orb.run();
     }
  }
```

サーバが，ある特定のクラスのオブジェクトのインスタンスを複数生成する場合，各インスタンス用にオブジェクトラッパーのセットが生成されます。

21.5.4　タイプドオブジェクトラッパーの削除

C++の場合

idl2cpp コンパイラがクラス用に生成する<interface_name>ObjectWrapper::remove メソッドは，クライアントまたはサーバアプリケーションからタイプドオブジェクトラッパーを削除できるようにします。ファクトリを削除する場合はロケーションを指定しなければなりません。これは，ロケーションが VISObjectWrapper::Both のファクトリを追加したら，クライアントロケーション，サーバロケーション，またはその両方からそのファクトリを削除することを選択できるということです。

Java の場合

Helper クラスは，クライアントまたはサーバアプリケーションからタイプドオブジェクトラッパーを削除するメソッドも提供します。詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「オブジェクトラッパー用に生成されるメソッド」の記述を参照してください。

注

クライアントから一つまたは複数のオブジェクトラッパーを削除しても，クライアントがすでにバインドしたクラスのオブジェクトは影響を受けません。あとでバインドされるオブジェクトだけが影響を受けます。サーバからオブジェクトラッパーを削除しても，すでにリクエストを処理したオブジェクトインプリメンテーションは影響を受けません。あとで生成されるオブジェクトインプリメンテーションだけが影響を受けます。

21.6　タイプドおよびアンタイプドオブジェクトラッパーの混在使用

アプリケーション中でタイプドオブジェクトラッパーとアンタイプドオブジェクトラッパーの両方を使うことを選択した場合，アンタイプドオブジェクトラッパー用に定義された pre_method メソッドはすべて，オブジェクトのために定義されたあらゆるタイプドオブジェクトラッパーメソッドの前に呼び出されます。オブジェクトのために定義されたタイプドオブジェクトラッパーメソッドはすべて，アンタイプドオブジェクトラッパー用に定義されたあらゆる post_method メソッドの前に呼び出されます。

サンプルアプリケーションの Client.C と Server.C（C++）または Client.java と Server.java（Java）は，タイプドオブジェクトラッパーとアンタイプドオブジェクトラッパーのどちらを使用するかを指定するコマンドラインプロパティを使えるようになっています。

21.6.1　タイプドオブジェクトラッパーのコマンドライン引数

C++の場合

表 21-3 に，Client.C および Server.C でインプリメントされた Bank アプリケーションのサンプルでタイプドオブジェクトラッパーを使用できるようにするためのコマンドライン引数を示します。

表 21-3　タイプドオブジェクトラッパー制御用のコマンドライン引数

BankWrappers プロパティ	説明
-BANKaccountCacheClnt <0|1>	クライアントアプリケーション用の balance メソッドの結果をキャッシュするタイプドオブジェクトラッパーを使用できるようにするかどうかを指定します。
-BANKaccountCacheSrvr <0|1>	サーバアプリケーション用の balance メソッドの結果をキャッシュするタイプドオブジェクトラッパーを使用できるようにするかどうかを指定します。
-BANKmanagerCacheClnt <0|1>	クライアントアプリケーション用の open メソッドの結果をキャッシュするタイプドオブジェクトラッパーを使用できるようにするかどうかを指定します。
-BANKmanagerCacheSrvr <0|1>	サーバアプリケーション用の open メソッドの結果をキャッシュするタイプドオブジェクトラッパーを使用できるようにするかどうかを指定します。
-BANKmanagerSecurityClnt <0|1>	クライアントアプリケーション用の open メソッドで渡される無許可のユーザを検出するタイプドオブジェクトラッパーを使用できるようにするかどうかを指定します。
-BANKmanagerSecuritySrvr <0|1>	サーバアプリケーション用の open メソッドで渡される無許可のユーザを検出するタイプドオブジェクトラッパーを使用できるようにするかどうかを指定します。

Java の場合

タイプドオブジェクトラッパーは，コマンドラインに次のように指定することによって使用できます。

1. -Dvbroker.orb.dynamicLibs=BankWrappers.Init
2. 表 21-4 にあるプロパティを一つまたは複数指定する

表 21–4　BankWrappers を使用可能または使用不可能にするコマンドラインプロパティ

BankWrappers プロパティ	説明
-DCachingAccount [=<client|server>]	クライアントまたはサーバ用の balance メソッドの結果をキャッシュするタイプドオブジェクトラッパーをインストールします。サブプロパティに値を指定しないと，クライアントオブジェクトラッパーとサーバオブジェクトラッパーの両方がインストールされます。
-DCachingAccountManager [=<client|server>]	クライアントまたはサーバ用の open メソッドの結果をキャッシュするタイプドオブジェクトラッパーをインストールします。サブプロパティに値を指定しないと，クライアントオブジェクトラッパーとサーバオブジェクトラッパーの両方がインストールされます。
-DSecureAccountManager [=<client|server>]	クライアントまたはサーバ用の open メソッドで渡された無許可のユーザを検出するタイプドオブジェクトラッパーをインストールします。サブプロパティに値を指定しないと，クライアントオブジェクトラッパーとサーバオブジェクトラッパーの両方がインストールされます。

21.6.2　タイプドオブジェクトラッパーのイニシャライザ

C++の場合

タイプドオブジェクトラッパーは objectWrappers/BankWrap.C に定義され，BankInit::update イニシャライザで生成されます。このイニシャライザは，ORB_init が呼び出されるときに起動され，ユーザが指定するコマンドラインプロパティに基づいて，さまざまなタイプドオブジェクトラッパーをインストールできます。

コードサンプル 21-15 に，イニシャライザが PropStruct オブジェクトのセットをどのように使用し，指定されたコマンドラインオプションを把握し，該当するロケーションに AccountObjectWrapper オブジェクトを追加または削除するかを示します。

コードサンプル 21-15　タイプドオブジェクトラッパーのイニシャライザ（C++）

```
. . .
static const CORBA::ULong kNumTypedAccountProps = 2;
static PropStruct TypedAccountProps[kNumTypedAccountProps] =
{ { "BANKaccountCacheClnt",
      CachingAccountObjectWrapper::factory,
      VISObjectWrapper::Client },
   { "BANKaccountCacheSrvr",
        CachingAccountObjectWrapper::factory,
        VISObjectWrapper::Server }
};
static const CORBA::ULong kNumTypedAccountManagerProps = 4;
static PropStruct TypedAccountManagerProps[
      kNumTypedAccountManagerProps] =
{ { "BANKmanagerCacheClnt",
      CachingAccountManagerObjectWrapper::factory,
      VISObjectWrapper::Client },
{ "BANKmanagerSecurityClnt",
      SecureAccountManagerObjectWrapper::factory,
      VISObjectWrapper::Client },
{ "BANKmanagerCacheSrvr",
      CachingAccountManagerObjectWrapper::factory,
      VISObjectWrapper::Server },
{ "BANKmanagerSecuritySrvr",
      SecureAccountManagerObjectWrapper::factory,
      VISObjectWrapper::Server },
};

void BankInit::update(int& argc, char* const* argv) {
```

```
    if (argc > 0) {
       init(argc, argv, "-BANK");
       CORBA::ULong i;
       for (i=0; i < kNumTypedAccountProps; i++) {
          CORBA::String_var arg(
             getArgValue(TypedAccountProps[i].propname));
          if (arg && strlen(arg) > 0) {
             if (atoi((char*) arg)) {
                Bank::AccountObjectWrapper::add(_orb,
                      TypedAccountProps[i].fact,
                      TypedAccountProps[i].loc);
             } else {
                Bank::AccountObjectWrapper::remove(_orb,
                      TypedAccountProps[i].fact,
                      TypedAccountProps[i].loc);
             }
          }
       }
       for (i=0; i < kNumTypedAccountManagerProps; i++) {
             CORBA::String_var arg(
                getArgValue(
                      TypedAccountManagerProps[i].propname));
             if (arg && strlen(arg) > 0 ) {
                   if (atoi((char*) arg)) {
                         Bank::AccountManagerObjectWrapper
                           ::add(_orb,
                            TypedAccountManagerProps[i].fact,
                            TypedAccountManagerProps[i].loc);
                   } else {
                        Bank::AccountManagerObjectWrapper
                           ::remove(_orb,
                           TypedAccountManagerProps[i].fact,
                           TypedAccountManagerProps[i].loc);
                   }
             }
       }
    }
}
```

Java の場合

タイプドオブジェクトラッパーは BankWrappers パッケージに定義され，コードサンプル 21-16 にあるサービスイニシャライザの BankWrappers/Init.java を含みます。このイニシャライザは，クライアントまたはサーバを vbj で起動するときにコマンドラインで-Dvbroker.orb.dynamicLibs=BankWrappers.Init を指定すると起動されます。ユーザが指定するコマンドラインプロパティに基づいて，さまざまなタイプドオブジェクトラッパーをインストールできます。コマンドラインプロパティについては表 21-4 を参照してください。

コードサンプル 21-16　BankWrappers のサービスイニシャライザ（Java）

```
package BankWrappers;
import java.util.*;
import com.inprise.vbroker.orb.ORB;
import com.inprise.vbroker.properties.PropertyManager;
import com.inprise.vbroker.interceptor.*;
public class Init implements ServiceLoader {
   com.inprise.vbroker.orb.ORB _orb;
   public void init(final org.omg.CORBA.ORB orb) {
      _orb = (ORB) orb;
      PropertyManager pm = _orb.getPropertyManager();
      // install my CachingAccountObjectWrapper
      String val = pm.getString("CachingAccount",
         this.toString());
      Class c = CachingAccountObjectWrapper.class;
      if( !val.equals(this.toString())) {
         if( val.equalsIgnoreCase("client") ) {
            Bank.AccountHelper.
                        addClientObjectWrapperClass(orb, c);
         } else if( val.equalsIgnoreCase("server") ) {
            Bank.AccountHelper.
                        addServerObjectWrapperClass(orb, c);
         } else {
            Bank.AccountHelper.
```

```
                        addClientObjectWrapperClass(orb, c);
            Bank.AccountHelper.
                        addServerObjectWrapperClass(orb, c);
         }
      }
      // install my CachingAccountManagerObjectWrapper
      val = pm.getString("CachingAccountManager",
                                          this.toString());
      c = CachingAccountManagerObjectWrapper.class;
      if( !val.equals(this.toString())) {
         if( val.equalsIgnoreCase("client") ) {
            Bank.AccountManagerHelper.
                        addClientObjectWrapperClass(orb, c);
         } else if( val.equalsIgnoreCase("server") ) {
            Bank.AccountManagerHelper.
                        addServerObjectWrapperClass(orb, c);
         } else {
            Bank.AccountManagerHelper.
                        addClientObjectWrapperClass(orb, c);
            Bank.AccountManagerHelper.
                        addServerObjectWrapperClass(orb, c);
         }
      }
      // install my CachingAccountManagerObjectWrapper
      val = pm.getString("SecureAccountManager",
            this.toString());
      c = SecureAccountManagerObjectWrapper.class;
      if( !val.equals(this.toString())) {
         if( val.equalsIgnoreCase("client") ) {
            Bank.AccountManagerHelper.
                        addClientObjectWrapperClass(orb, c);
         } else if( val.equalsIgnoreCase("server") ) {
            Bank.AccountManagerHelper.
                        addServerObjectWrapperClass(orb, c);
         } else {
            Bank.AccountManagerHelper.
                        addClientObjectWrapperClass(orb, c);
            Bank.AccountManagerHelper.
                        addServerObjectWrapperClass(orb, c);
         }
      }
   }
   public void init_complete(org.omg.CORBA.ORB orb) {}
   public void shutdown(org.omg.CORBA.ORB orb) {}
}
```

21.6.3　アンタイプドオブジェクトラッパー用コマンドライン引数

C++の場合

表 21-5 に Client.C および Server.C でインプリメントされた Bank アプリケーションのサンプルでアンタイプドオブジェクトラッパーを使用できるようにするためのコマンドライン引数を示します。

表 21-5　アンタイプドオブジェクトラッパー制御用のコマンドライン引数

BankWrappers プロパティ	説明
-TRACEWRAPclient <numwraps>	クライアントアプリケーション用のオブジェクトラッパーをトレースするための指定の数のアンタイプドオブジェクトラッパーファクトリを実体化します。
-TRACEWRAPserver <numwraps>	サーバアプリケーションでトレースするための指定の数のアンタイプドオブジェクトラッパーファクトリを実体化します。
-TRACEWRAPboth <numwraps>	クライアントおよびサーバアプリケーションでトレースするための指定の数のアンタイプドオブジェクトラッパーファクトリを実体化します。

BankWrappers プロパティ	説明
-TIMINGWRAPclient <numwraps>	クライアントアプリケーションでタイミングを計るための指定の数のアンタイプドオブジェクトラッパーファクトリを実体化します。
-TIMINGWRAPserver <numwraps>	サーバアプリケーションでタイミングを計るための指定の数のアンタイプドオブジェクトラッパーファクトリを実体化します。
-TIMINGWRAPboth<numwraps>	クライアントおよびサーバアプリケーションでタイミングを計るための指定の数のアンタイプドオブジェクトラッパーファクトリを実体化します。

Java の場合

アンタイプドオブジェクトラッパーは，コマンドラインに次のように指定することによって使用できます。

1. -Dvbroker.orb.dynamicLibs=UtilityObjectWrappers.Init
2. 表 21-6 にあるプロパティを一つまたは複数指定する

表 21-6　UtilityObjectWrappers を使用可能または使用不可能にするコマンドラインプロパティ

UtilityObjectWrappers プロパティ	説明
-DTiming[=<client|server>]	クライアントまたはサーバの情報のタイミングを取るためのアンタイプドオブジェクトラッパーをインストールします。サブプロパティの値が指定されない場合，クライアントオブジェクトラッパーとサーバオブジェクトラッパーの両方がインストールされます。
-DTracing[=<client|server>]	クライアントまたはサーバの情報をトレーシングするためのアンタイプドオブジェクトラッパーをインストールします。サブプロパティの値が指定されない場合，クライアントオブジェクトラッパーとサーバオブジェクトラッパーの両方がインストールされます。

21.6.4　アンタイプドオブジェクトラッパーのイニシャライザ

C++の場合

アンタイプドオブジェクトラッパーは BankWrappers/TraceWrap.C と TimeWrap.C に定義され，TraceWrapInit::update と TimingWrapInit::update メソッド内に作成され，登録されます。これらのイニシャライザは，ORB_init メソッドが起動するときに起動され，さまざまなアンタイプドオブジェクトラッパーをインストールできます。コードサンプル 21-17 に，指定したコマンドラインプロパティに基づいて，該当するアンタイプドオブジェクトラッパーをインストールする TraceWrap.C ファイルの一部を示します。

コードサンプル 21-17　アンタイプドオブジェクトラッパーのイニシャライザ（C++）

```
TraceWrapInit::update(int& argc, char* const* argv) {
   if (argc > 0) {
      init(argc, argv, "-TRACEWRAP");

      VISObjectWrapper::Location loc;
      const char* propname;
      LIST(VISObjectWrapper::
           UntypedObjectWrapperFactory_ptr) *list;

      for (CORBA::ULong i=0; i < 3; i++) {
         switch (i) {
            case 0:
               loc = VISObjectWrapper::Client;
               propname = "TRACEWRAPclient";
```

```
                list = &_clientfacts;
                break;
            case 1:
                loc = VISObjectWrapper::Server;
                propname = "TRACEWRAPserver";
                list = &_serverfacts;
                break;
            case 2:
                loc = VISObjectWrapper::Both;
                propname = "TRACEWRAPboth";
                list = &_bothfacts;
                break;
        }
        CORBA::String_var getArgValue(
                          property_value(propname));
        if (arg && strlen(arg) > 0) {
            int numNew = atoi((char*) arg);
            char key_buf[256];
            for (CORBA::ULong j=0; j < numNew; j++) {
                sprintf(key_buf, "%s-%d", propname,
                      list->size());
                list->add(new TraceObjectWrapperFactory(loc,
                      (const char*) key_buf));
            }
        }
    }
  }
}
```

Java の場合

アンタイプドオブジェクトラッパーは UtilityObjectWrappers パッケージに定義され，コードサンプル 21-18 にあるサービスイニシャライザの UtilityObjectWrappers/Init.java を含みます。このイニシャライザは，クライアントまたはサーバを vbj で起動するときにコマンドラインで-Dvbroker.orb.dynamicLibs=UtilityObjectWrappers.Init を指定すると起動されます。ユーザが指定するコマンドラインプロパティに基づいて，さまざまなアンタイプドオブジェクトラッパーをインストールできます。コマンドラインプロパティについては表 21-6 を参照してください。

コードサンプル 21-18　UtilityObjectWrappers のサービスイニシャライザ（Java）

```
package UtilityObjectWrappers;
import java.util.*;
import com.inprise.vbroker.orb.ORB;
import com.inprise.vbroker.properties.PropertyManager;
import com.inprise.vbroker.interceptor.*;

public class Init implements ServiceLoader {
   com.inprise.vbroker.orb.ORB _orb;
   public void init(final org.omg.CORBA.ORB orb) {
      _orb = (ORB) orb;
      PropertyManager pm= _orb.getPropertyManager();
      try {
         ChainUntypedObjectWrapperFactory factory =
           ChainUntypedObjectWrapperFactoryHelper.narrow(
             orb.resolve_initial_references(
             "ChainUntypedObjectWrapperFactory"));
         // install my Timing ObjectWrapper
         String val = pm.getString("Timing", this.toString());
         if( !val.equals(this.toString())) {
            UntypedObjectWrapperFactory f=
               new TimingUntypedObjectWrapperFactory();
            if( val.equalsIgnoreCase("client") ) {
               factory.add(f, Location.CLIENT);
            } else if( val.equalsIgnoreCase("server") ) {
               factory.add(f, Location.SERVER);
            } else {
               factory.add(f, Location.BOTH);
            }
         }
         // install my Tracing ObjectWrapper
         val = pm.getString("Tracing", this.toString());
         if( !val.equals(this.toString())) {
            UntypedObjectWrapperFactory f=
```

```
                    new TracingUntypedObjectWrapperFactory();
                if( val.equalsIgnoreCase("client") ) {
                    factory.add(f, Location.CLIENT);
                } else if( val.equalsIgnoreCase("server") ) {
                    factory.add(f, Location.SERVER);
                } else {
                    factory.add(f, Location.BOTH);
                }
            }
        } catch( org.omg.CORBA.ORBPackage.InvalidName e ) {
            return;
        }
    }

    public void init_complete(org.omg.CORBA.ORB orb) {}
    public void shutdown(org.omg.CORBA.ORB orb) {}
}
```

21.6.5　サンプルアプリケーションの実行

サンプルアプリケーションを実行する前に，ネットワーク上で osagent が実行中であることを確認してください。そのあと，トレーシングオブジェクトラッパーやタイミングオブジェクトラッパーを使わないで，次のコマンドを使って，サーバアプリケーションを実行できます。

C++の場合

```
prompt> Server
```

Java の場合

```
prompt> vbj Server
```

注

サーバは同一プロセスにあるアプリケーションとして設計されています。これはサーバとクライアントの両方をインプリメントします。

別のウィンドウからは，トレーシングオブジェクトラッパーやタイミングオブジェクトラッパーを使わないで，次のコマンドを使ってクライアントアプリケーションを実行し，ユーザアカウントの残高を問い合わせることができます。

C++の場合

```
prompt> Client John
```

Java の場合

```
prompt> vbj Client John
```

また，デフォルト名を使いたい場合も，このコマンドを実行できます。

C++の場合

```
prompt> Client
```

Java の場合

```
prompt> vbj Client
```

(1)　タイミングおよびトレーシングオブジェクトラッパーを使用可能にする

アンタイプドタイミングオブジェクトラッパーおよびトレーシングオブジェクトラッパーを使用可能にした状態でクライアントを実行するには，次のコマンドを使用してください。

C++の場合

```
prompt> Client -TRACEWRAPclient 1 -TIMINGWRAPclient 1
```

Java の場合

```
prompt> vbj -Dvbroker.orb.dynamicLibs=¥
        UtilityObjectWrappers.Init                          ¥
        -DTiming=client                                     ¥
        -DTracing=client Client John
```

タイミングおよびトレーシング用アンタイプドオブジェクトラッパーを使用可能にした状態でサーバを実行するには，次のコマンドを使用してください。

C++の場合

```
prompt> Server -TRACEWRAPserver 1 -TIMINGWRAPserver 1
```

Java の場合

```
prompt> vbj -Dvbroker.orb.dynamicLibs=¥
        UtilityObjectWrappers.Init                          ¥
        -DTiming=server                                     ¥
        -DTracing=server Server
```

(2)　キャッシングおよびセキュリティオブジェクトラッパーを使用可能にする

キャッシングおよびセキュリティのためのタイプドオブジェクトラッパーを使用可能にした状態でクライアントを実行するには，次のコマンドを使用してください。

C++の場合

```
prompt> Client -BANKaccountCacheClnt 1   ¥
        -BANKmanagerCacheClnt 1          ¥
        -BANKmanagerSecurityClnt 1
```

Java の場合

```
prompt> vbj -Dvbroker.orb.dynamicLibs=BankWrappers.Init   ¥
        -DCachingAccount=client                           ¥
        -DCachingAccountManager=client                    ¥
        -DSecureAccountManager=client                     ¥
        Client John
```

キャッシングおよびセキュリティのためのタイプドオブジェクトラッパーを使用可能にした状態でサーバを実行するには，次のコマンドを使用してください。

C++の場合

```
prompt> Server -BANKaccountCacheSrvr 1  ¥
        -BANKmanagerCacheSrvr 1         ¥
        -BANKmanagerSecuritySrvr 1
```

Java の場合

```
prompt> vbj -Dvbroker.orb.dynamicLibs=BankWrappers.Init    ¥
        -DCachingAccount=server                           ¥
        -DCachingAccountManager=server                    ¥
        -DSecureAccountManager=server                     ¥
        Server
```

(3)　タイプドおよびアンタイプドオブジェクトラッパーを使用可能にする

すべてのタイプドオブジェクトラッパーおよびアンタイプドオブジェクトラッパーを使用可能にした状態でクライアントを実行するには，次のコマンドを使用してください。

C++の場合

```
prompt> Client -BANKaccountCacheClnt 1          ¥
        -BANKmanagerCacheClnt 1                 ¥
        -BANKmanagerSecurityClnt 1              ¥
        -TRACEWRAPclient 1 -TIMINGWRAPclient 1
```

Java の場合

```
prompt> vbj -Dvbroker.orb.dynamicLibs=BankWrappers.Init,¥
        UtilityObjectWrappers.Init                          ¥
        -DCachingAccount=client                             ¥
        -DCachingAccountManager=client                      ¥
        -DSecureAccountManager=client                       ¥
        -DTiming=client                                     ¥
        -DTracing=client                                    ¥
        Client John
```

すべてのタイプドオブジェクトラッパーおよびアンタイプドオブジェクトラッパーを使用可能にした状態でサーバを実行するには，次のコマンドを使用してください。

C++の場合

```
prompt> Server BANKaccountCacheSrvr 1           ¥
        -BANKmanagerCacheSrvr 1                  ¥
        -BANKmanagerSecuritySrvr 1              ¥
        -TRACEWRAPserver 1 -TIMINGWRAPserver 1
```

Java の場合

```
prompt> vbj -Dvbroker.orb.dynamicLibs=BankWrappers.Init,¥
        UtilityObjectWrappers.Init                           ¥
        -DCachingAccount=server                              ¥
        -DCachingAccountManager=server                       ¥
        -DSecureAccountManager=server                        ¥
        -DTiming=server                                      ¥
        -DTracing=server                                     ¥
        Server
```

(4)　同一プロセスにあるクライアントとサーバを実行する

C++の場合

すべてのタイプドオブジェクトラッパーを使用可能にした状態，クライアントだけのアンタイプドオブジェクトラッパーを使用可能にした状態，およびサーバだけのアンタイプドトレーシングオブジェクトラッパーを使用可能にした状態で，同一プロセスにあるサーバとクライアントを実行するには，次のコマンドを使用してください。

例

```
prompt> Server -BANKaccountCacheClnt 1                   ¥
        -BANKaccountCacheSrvr 1                          ¥
        -BANKmanagerCacheClnt 1 -BANKmanagerCacheSrvr 1  ¥
        -BANKmanagerSecurityClnt 1                       ¥
        -BANKmanagerSecuritySrvr 1                       ¥
        -TRACEWRAPboth 1                                 ¥
        -TIMINGWRAPboth 1
```

Java の場合

表 21-7 の-runCoLocated コマンドラインオプションを指定すると，同じプロセス内でクライアントとサーバを実行できます。

表 21-7　-runCoLocated コマンドラインオプション

プロパティ	説明
-runCoLocated Client	同じプロセス内で Server.java と Client.java を実行します。
-runCoLocated TypedClient	同じプロセス内で Server.java と TypedClient.java を実行します。
-runCoLocated UntypedClient	同じプロセス内で Server.java と UntypedClient.java を実行します。

すべてのタイプドオブジェクトラッパーを使用可能にした状態，クライアントだけのアンタイプドタイミングオブジェクトラッパーを使用可能にした状態，およびサーバだけのアンタイプドトレーシングオ

ブジェクトラッパーを使用可能にした状態で，同一プロセスにあるサーバとクライアントを実行するには，次のコマンドを使用してください。

例

```
prompt> vbj -Dvbroker.orb.dynamicLibs=BankWrappers.Init,¥
        UtilityObjectWrappers.Init                          ¥
        -DCachingAccount                                    ¥
        -DSecureAccountManager                              ¥
        -DTiming=client                                     ¥
        -DTracing=server                                    ¥
        Server -runCoLocated Client
```

22 イベントキュー

この章では，イベントキュー機能について説明します。ただし，この機能はサーバ側だけに提供されています。サーバは, サーバが対象とするイベントタイプに基づいてリスナーをイベントキューに登録できるので, サーバが必要なときにこのイベントを処理できます。

22.1　イベントタイプ

コネクションイベントタイプは，生成されているイベントタイプだけです。

22.1.1　コネクションイベント

VisiBroker ORB が生成して，登録されたコネクションイベントにプッシュするコネクションイベントには次の 2 種類があります。

- 設定されたコネクション
  これは，新しいクライアントがサーバへの接続に成功したことを示します。
- クローズしたコネクション
  これは，既存のクライアントがサーバから切断されたことを示します。

22.2　イベントリスナー

サーバは，サーバが処理する必要のあるイベントタイプに基づいて VisiBroker ORB によってリスナーをインプリメントして登録します。サポートされているイベントリスナーは，コネクションイベントリスナーだけです。

ほかのタイプのイベントリスナーは，必要に応じて将来のバージョンで追加される予定です。

22.2.1　IDL 定義

インタフェース定義を次に示します。

```
module EventQueue {
   // Connection event types
   enum EventType {UNDEFINED, CONN_EVENT_TYPE};

   // Peer (Client)connection info
   struct ConnInfo {
      string ipaddress; // in %d.%d.%d.%d format
      long     port;
      long     connID;
   };

   // Marker interface for all types of event listeners
   local interface EventListener {};


   typedef sequence<EventListener> EventListeners;


   // connection event listener interface
   local interface ConnEventListener : EventListener{
         void conn_established(in ConnInfo info);
         void conn_closed(in ConnInfo info);
   };

   // The EventQueue manager
   local interface EventQueueManager :
                           interceptor::InterceptorManager {
        void register_listener(in EventListener listener,
                                          in EventType type);
        void unregister_listener(in EventListener listener,
                                          in EventType type);
        EventListeners get_listeners(in EventType type);
   };

};
```

インタフェース定義の詳細を以降で説明します。

(1)　ConnInfo 構造体

ConnInfo 構造体には次のようなクライアントコネクション情報があります。

- ipaddress：通信相手の IP アドレスを格納します。
- port：通信相手のポート番号を格納します。
- connID：このクライアントコネクションのサーバごとの一意の識別子を格納します。

(2)　EventListener インタフェース

EventListener インタフェースの部分は，すべてのタイプのイベントリスナーのマーカーインタフェースです。

(3)　ConnEventListeners インタフェース

ConnEventListeners インタフェースは次のように二つのオペレーションを定義します。

- void conn_established (in ConnInfo info)
  このオペレーションは VisiBroker ORB によってコールバックされ，コネクション設定イベントをプッシュします。VisiBroker ORB は in ConnInfo info パラメタにクライアントコネクション情報を与えて，この値をコールバックオペレーションに渡します。
- void conn_closed (in ConnInfo info)
  このオペレーションは VisiBroker ORB によってコールバックされ，コネクションクローズイベントをプッシュします。VisiBroker ORB は in ConnInfo info パラメタにクライアントコネクション情報を与えて，この値をコールバックオペレーションに渡します。

サーバ側アプリケーションは，リスナーにプッシュされているイベントの処理と同様，ConnEventListener インタフェースのインプリメンテーションにも責任があります。

(4)　EventQueueManager インタフェース

EventQueueManager インタフェースは，イベントリスナーの登録に関するサーバ側インプリメンテーションによってハンドルとして使用されます。このインタフェースは次のように三つのオペレーションを定義します。

- void register_listener (in EventListener listener, in EventType type)
  このオペレーションは，指定のイベントタイプのイベントリスナーの登録用に用意されています。
- EventListeners get_listeners (in EventType type)
  このオペレーションは指定のタイプの登録済みイベントリスナーのリストを返します。
- void unregister_listener (in EventListener listener, in EventType type)
  このオペレーションは事前に登録された指定のタイプのリスナーを削除します。

22.2.2　EventQueueManager の返し方

EventQueueManager オブジェクトは ORB 初期化時に生成されます。サーバ側のインプリメンテーションは，次に示すコードによって EventQueueManager オブジェクトリファレンスを返します。

コードサンプル 22-1　EventQueueManager（C++）

```
CORBA::Object *object =
       orb->resolve_initial_references("VisiBrokerInterceptorControl");


interceptor::InterceptorManagerControl_var control =
          interceptor::InterceptorManagerControl::_narrow(object);


interceptor::InterceptorManager_var manager =
       control->get_manager("EventQueueManager");
EventQueue::EventQueueManager_var eq_mgr =

EventQueue::EventQueueManager::_narrow(manager);
```

コードサンプル 22-2　EventQueueManager（Java）

```
com.inprise.vbroker.interceptor.InterceptorManagerControl control =
com.inprise.vbroker.interceptor.
            InterceptorManagerControlHelper.narrow(
    orb.resolve_initial_references(
            "VisiBrokerInterceptorControl"));
EventQueueManager manager =
```

```
            (EventQueueManager)control.get_manager("EventQueue");
EventListener theListener =...
manager.register_listeners(theListener);
```

22.2.3　コードサンプル

EventListeners の登録およびコネクション EventListener のインプリメントについて，C++と Java のコードサンプルを示します。

(1)　EventListeners の登録

SampleServerLoader クラスには，初期化時に ORB が呼び出す ORB_init()メソッド（C++）または init()メソッド（Java）があります。ServerLoader の目的は，EventListener を作成して，それを EventQueueManager に登録することです。

コードサンプル 22-3　SampleServerLoader.h（C++）

```
#ifdef _VIS_STD
#include <iostream>
#else
#include <iostream.h>
#endif

#include "vinit.h"
#include "ConnEventListenerImpl.h"

USE_STD_NS

class SampleServerLoader :public VISInit
{
    private:
       short int _conn_event_interceptors_installed;

    public:
    SampleServerLoader(){
      _conn_event_interceptors_installed =0;
    }

    void ORB_init(int& argc, char* const* argv, CORBA::ORB_ptr orb)
    {
      if( _conn_event_interceptors_installed) return;
         cout << "Installing Connection event interceptors" << endl;

       ConnEventListenerImpl *interceptor =
        new ConnEventListenerImpl("ConnEventListener");

       // Get the interceptor manager control
       CORBA::Object *object =
            orb->resolve_initial_references
                          ("VisiBrokerInterceptorControl");

       interceptor::InterceptorManagerControl_var control =
            interceptor::InterceptorManagerControl::_narrow(object);

       // Get the POA manager
       interceptor::InterceptorManager_var manager =
          control->get_manager("EventQueueManager");
       EventQueue::EventQueueManager_var eq_mgr =
          EventQueue::EventQueueManager::_narrow(manager);

       // Add POA interceptor to the list
       eq_mgr->register_listener( (EventQueue::ConnEventListener
             *)interceptor,
          EventQueue::CONN_EVENT_TYPE);

       cout << "Event queue interceptors installed" << endl;
            _conn_event_interceptors_installed = 1;
    }
};
```

コードサンプル 22-4　SampleServerLoader.java（Java）

```
import com.inprise.vbroker.EventQueue.*;
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.PortableServerExt.*;

public class SampleServerLoader implements ServiceLoader
{
   public void init(org.omg.CORBA.ORB orb)
   {
      try {
        InterceptorManagerControl control =
               InterceptorManagerControlHelper.narrow(
                  orb.resolve_initial_references
                     ("VisiBrokerInterceptorControl"));

        EventQueueManager queue_manager =
           (EventQueueManager)control.get_manager("EventQueue");

        queue_manager.register_listener((EventListener)new
                  ConnEventListenerImpl(),EventType.CONN_EVENT_TYPE);
      }
      catch(Exception e){
         e.printStackTrace();
         throw new org.omg.CORBA.INITIALIZE(e.toString());
      }
      System.out.println("=========>SampleServerLoader:
                                    ConnEventListener registered");
   }

   public void init_complete(org.omg.CORBA.ORB orb){

   }

   public void shutdown(org.omg.CORBA.ORB orb){

   }
}
```

(2)　EventListeners のインプリメント

ConnEventListenerImpl には，コネクションイベントリスナーのインプリメンテーションサンプルがあります。ConnEventListener インタフェースは，サーバ側アプリケーションで conn_established および conn_closed オペレーションをインプリメントします。インプリメンテーションによってコネクションは，サーバ側でリクエストを待っている間，30,000 ミリ秒間アイドル状態にできます。このようなオペレーションは，クライアントがコネクションを設定した場合と，コネクションが切断された場合にそれぞれ呼び出されます。

コードサンプル 22-5　ConnEventListenerImpl.h（C++）

```
#ifdef _VIS_STD
#include <iostream>
#else
#include <iostream.h>
#endif

#include "vextclosure.h"
#include "interceptor_c.hh"
#include "IOP_c.hh"
#include "EventQueue_c.hh"
#include "vutil.h"

//USE_STD_NS is a define setup by VisiBroker to use the std namespace
USE_STD_NS

//--------------------------------------------------------------
//defines the server interceptor functionality
//--------------------------------------------------------------
class ConnEventListenerImpl :public EventQueue::ConnEventListener
{
```

```
   private:
      char * _id;
   public:
   ConnEventListenerImpl( const char* id){
      _id =new char [ strlen((id) + 1 ];
      strcpy( _id,id);
   }

   virtual~ConnEventListenerImpl(){
      delete[ ] _id;
      _id =NULL;
   }

   //-------------------------------------------------------------
   // This method gets called when a request arrives at the
   // server end.
   //-------------------------------------------------------------
   void conn_established(const EventQueue::ConnInfo& connInfo){
      cout <<"Processing connection established from" <<endl;
      cout << connInfo;
      cout <<endl;
      VISUtil : sleep(30000);
   }
   void conn_closed(const EventQueue::ConnInfo &connInfo){
      cout <<"Processing connection closed from " <<endl ;
      cout <<connInfo ;
      cout <<endl;
      VISUtil::sleep(30000);
   }
};
```

コードサンプル 22-6　ConnEventListenerImpl.java（Java）

```
import com.inprise.vbroker.EventQueue.*;
import org.omg.CORBA.LocalObject;

public class ConnEventListenerImpl extends LocalObject implements
                                                  ConnEventListener {
   public void conn_established(ConnInfo info){
      System.out.println("Received conn_established:address ="+
                         info.ipaddress + "port ="+info.port +
                         " connID = " + info.connID);
      System.out.println("Processing the event ...");
      try {
        Thread.sleep(30000);
      }catch (Exception e){e.printStackTrace();}
   }

   public void conn_closed(ConnInfo info) {
      System.out.println("Received conn_closed:address = " +
                         info.ipaddress+"port ="+info.port +
                         " connID = " + info.connID);
   }
}
```

23 RMI-IIOP の使用

この章では，RMI-IIOP を使用するための Borland Enterprise Server VisiBroker ツールについて説明します。また, RMI-IIOP を使用する Java アプレットを実行する場合のセットアップ許可要件についても簡単に説明します。

23.1　概要

RMI（Remote Method Invocation）は，オブジェクトを生成し，分散環境で使用できるようにする Java の方式です。この意味では，RMI は言語固有（Java）であり，CORBA に準拠しない VisiBroker ORB です。OMG は，IDL マッピングに対する Java 言語の仕様を規定しており，これによって，IIOP を使用する CORBA オブジェクトと RMI を使用している Java のクラスが相互に通信できるようになります。

23.1.1　RMI-IIOP による Java アプレットの設定

RMI-IIOP を使用するアプレットを実行できますが，Reflect および Runtime で許可を設定する必要があります。このような許可は JRE のインストールディレクトリにある java.policy ファイルで設定します。ここに，java.policy ファイルでの許可の設定方法の例を示します。

```
grant codeBase "http://xxx.xxx.xxx.xxx:8088/-" {
permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
permission java.lang.RuntimePermission "accessDeclaredMembers";
};
```

23.1.2　java2iiop および java2idl ツール

Borland Enterprise Server VisiBroker には二つのコンパイラがあります。このコンパイラによって，ユーザは VisiBroker ORB を使用して既存の Java クラスをほかのオブジェクトで作業させることができるようになります。

- java2iiop コンパイラは，RMI 準拠のクラスを受け取り，適切なスケルトン，スタブ，およびヘルパークラスを生成して IIOP を使用できるようにします。
- java2idl コンパイラは，Java クラスから IDL を生成して，Java 以外の言語でインプリメントできるようにします。

23.2　java2iiop の使用

java2iiop コンパイラは，CORBA で使用できるインタフェースおよびデータ型を定義できますが，このコンパイラのメリットはそれらを IDL ではなく Java で定義できる点にあります。このコンパイラは Java ソースコード（Java ファイル）や IDL を読み込みませんが，Java バイトコード（class ファイル）は読み込みます。そして，コンパイラは CORBA で必要なすべてのマーシャルおよび通信のために IIOP 準拠のスタブおよびスケルトンを生成します。

23.2.1　サポートしているインタフェース

java2iiop コンパイラを実行すると，インタフェースを IDL で記述した場合と同じようにファイルが生成されます。数値型（short，int，long，float，および double）などの基本データ型，文字列，CORBA オブジェクトまたはインタフェースオブジェクト，Any オブジェクト，typecode オブジェクトはすべて java2iiop コンパイラによって認識され，IDL の対応する型にマッピングされます。

Java クラスまたはインタフェースで java2iiop を使用できます。例えば，Java インタフェースが次の規則のどれかを継承する場合，java2iiop はインタフェースを IDL の CORBA インタフェースに変換します。

- java.rmi.Remote を継承して，そのメソッドのすべてで java.rmi.RemoteException が発生する
- org.omg.CORBA.Object を継承する

コードサンプル 23-1 に Java RMI インタフェースを示します。コードサンプルは，Borland Enterprise Server VisiBroker をインストールしたディレクトリの examples/vbe/rmi-iiop に入っています。

コードサンプル 23-1　java.rmi.Remote の継承

```
public interface Account extends java.rmi.Remote {
   String name() throws java.rmi.RemoteException;
   float getBalance() throws java.rmi.RemoteException;
   void setBalance(float bal) throws java.rmi.RemoteException;
}
```

23.2.2　java2iiop の実行

java2iiop コンパイラを使用する前に，Java クラスをコンパイルする必要があります。バイトコードを生成したら，java2iiop を実行してクライアントスタブ，サーバスケルトン，およびその関連補助ファイルを生成できます。例えば，java2iiop を examples/vbe/rmi-iiop/Bank ディレクトリの Account.class ファイルで実行すると，次のファイルができることになります。

- _Account_Stub
- AccountHelper
- AccountHolder
- AccountPOA
- _Account_Tie
- AccountOperations

これらのファイルの詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「生成されるインタフェースとクラス」の記述を参照してください。

(1)　Java クラスの IDL への逆マッピング

idl2java コンパイラを使用して IDL インタフェースを Java クラスにマッピングする場合，インタフェース名は，生成されたクラスの拡張子（例: Helper，Holder，POA など）を使用し，場合によって idl2java ツールはインタフェース名の識別子にプリフィクスとして下線（_）を付けてクラスを生成します。例えば，IDL へ Foo と FooHolder インタフェースの両方を定義する場合，idl2java は Foo.java，FooHolder.java，_FooHolder.java，および_FooHolderHolder.java ファイルを生成します。反対に，java2iiop コンパイラを使用して RMI Java クラスから IIOP 準拠の Java クラスを生成する場合，ツールはプリフィクスとして下線を付けたクラスを生成できません。そのため，予約した拡張子を使用するインタフェースを宣言する場合，同じ名前のインタフェースとして同じパッケージに入れることはできません（例えば，java2iiop コンパイラを使用する場合，Foo と FooHolder クラスを同じパッケージに入れることはできません）。

23.2.3　開発プロセスの完了

インタフェースから対応するファイルを生成したあと，インタフェース用にインプリメンテーションを提供する必要があります。そのためには，次の手順に従ってください。

1. インタフェースクラス用のインプリメンテーションを作成します。
2. サーバクラスをコンパイルします。
3. クライアントコードを書き込み，コンパイルします。
4. サーバプログラムを起動します。
5. クライアントプログラムを実行します。

注

非準拠クラスをマーシャルしようとすると，org.omg.CORBA.MARSHAL: Cannot marshal non-conforming value of class <class name>が発生します。例えば，次の二つのクラスを生成します。

```
// This is a conforming class
public class Value implements java.io.Serializable {
   java.lang.Object any;
   . . .
}

// This is a non-conforming class
public class Something {
. . .
}
```

そして，次を試行します。

```
Value val = new Value();
val.any = new Something();
```

val をマーシャルしようとすると org.omg.CORBA.MARSHAL 例外が発生します。

23.3　RMI-IIOP バンクのサンプル

Account インタフェースは，java.rmi.Remote インタフェースを継承し，AccountImpl クラスによってインプリメントされます（コードサンプル 23-2 参照）。

Client クラス（コードサンプル 23-3 参照）は，まず，アカウントを生成するために各アカウント用に AccountData オブジェクトを生成して，それらを AccountManager に渡すことによって，指定のすべての Account オブジェクトを適切な残高で生成します。そして，生成されたアカウントで残高が正しいかどうかを確認します。クライアントは AccountManager に，アカウントすべてのリストを照会し，各アカウントの貸方に$10.00 を記入します。クライアントはアカウントの新しい残高が正確であるかどうかを検証します。

注

コードサンプルは，Borland Enterprise Server VisiBroker をインストールしたディレクトリの examples/vbe/rmi-iiop に入っています。

コードサンプル 23-2　Account インタフェースのインプリメント

```
public class AccountImpl extends Bank.AccountPOA {
   public AccountImpl(Bank.AccountData data) {
      _name = data.getName();
      _balance = data.getBalance();
   }
   public String name() throws java.rmi.RemoteException {
      return _name;
   }
   public float getBalance() throws java.rmi.RemoteException {
      return _balance;
   }
   public void setBalance(float balance) throws
                                        java.rmi.RemoteException {
      _balance = balance;
   }
   private float _balance;
   private String _name;
}
```

コードサンプル 23-3　Client クラス

```
public class Client {
   public static void main(String[ ] args) {
      try {
         // Initialize the ORB.
         org.omg.CORBA.ORB orb =
                           org.omg.CORBA.ORB.init(args,null);
         // Get the manager Id
         byte[ ] managerId = "RMIBankManager".getBytes();
         // Locate an account manager. Give the full POA
         // name and the servant ID.
         Bank.AccountManager manager =
                        Bank.AccountManagerHelper.bind(orb,
                                "/rmi_bank_poa", managerId);
         // Use any number of argument pairs to indicate
         // name, balance of accounts to create
         if (args.length == 0 || args.length % 2 != 0) {
            args = new String[2];
            args[0] = "Jack B. Quick";
            args[1] = "123.23";
         }
         int i = 0;
         while (i < args.length) {
            String name = args[i++];
            float balance;
            try {
               balance = new Float(args[i++]).floatValue();
            } catch (NumberFormatException n) {
               balance = 0;
            }
```

```
            Bank.AccountData data =
                        new Bank.AccountData(name, balance);
            Bank.Account account = manager.create(data);
            System.out.println("Created account for " +
               name + " with opening balance of $" + balance);
         }
         java.util.Hashtable accounts =
                                     manager.getAccounts();
         for (java.util.Enumeration e = accounts.elements();
                                    e.hasMoreElements();) {
            Bank.Account account =
                  Bank.AccountHelper.narrow(
                      (org.omg.CORBA.Object)e.nextElement());
            String name = account.name();
            float balance = account.getBalance();
            System.out.println("Current balance in " + name +
                               "'s account is $" + balance);
            System.out.println("Crediting $10 to " + name +
                                                "'s account.");
            account.setBalance(balance + (float)10.0);
            balance = account.getBalance();
            System.out.println("New balance in " + name +
                               "'s account is $" + balance);
         }
      } catch (java.rmi.RemoteException e){
         System.err.println(e);
      }
   }
}
```

23.4　サポートされるデータ型

Java 基本データ型に加えて，RMI-IIOP は Java クラスのサブセットをサポートします。

23.4.1　基本データ型のマッピング

java2iiop によって生成されたクライアントスタブは，オペレーション要求を表す Java 基本データ型のマーシャルを処理して，そのデータ型をオブジェクトサーバへ送信できるようにしています。Java 基本データ型のマーシャル時には，そのデータ型を IIOP 互換形式に変換する必要があります。表 23-1 に，Java 基本データ型と IDL/IIOP 型間のマッピングの概略を示します。

表 23-1　Java の型から IDL/IIOP へのマッピング

Java の型	IDL/IIOP の型
void	void
boolean	boolean
byte	octet
char	wchar
short	short
int	long
long	long long
float	float
double	double
java.lang.String	CORBA::WStringValue
java.lang.Object	any
java.io.Serializable	any
java.io.Externalizable	any

23.4.2　複合データ型のマッピング

ここでは，インタフェース，配列，および Java クラスについて説明し，java2iiop コンパイラを使用して複合データ型を処理する方法を示します。

(1)　インタフェース

Java インタフェースは，IDL では CORBA インタフェースとして表され，org.omg.CORBA.Object インタフェースから継承する必要があります。このようなインタフェースをインプリメントするオブジェクトを渡す場合はリファレンス渡しになります。

(2)　配列

クラス内に定義される別の複合データ型には配列があります。配列を使用する定義またはインタフェースがある場合，その配列はボックス型の CORBA ボックス型シーケンスにマッピングします。

24 動的管理型の使用

この章では，Borland Enterprise Server VisiBroker の DynAny 機能について説明します。この機能を使うことで, ユーザはランタイム時にデータ型を作成し，解釈できます。

24.1 概要

DynAny インタフェースは，ランタイム時に動的に基本データ型と構造化データ型を生成できるようにします。またコンパイル時にサーバがオブジェクトに含まれる型を知らなくても，Any オブジェクトの情報を解釈し，抽出できるようにします。DynAny インタフェースを使用すると，ユーザはランタイム時にデータ型を作成，解釈するためのクライアントおよびサーバアプリケーションを作成できます。

DynAny インタフェースの使い方を示したクライアントおよびサーバアプリケーションのサンプルが，Borland Enterprise Server VisiBroker をインストールしたディレクトリの examples/vbe/dynany に入っています。この章では，これらのサンプルプログラムを使って DynAny の概念を説明します。

24.2　DynAny の型

DynAny オブジェクトには対応する値があり，その値は基本データ型（例：boolean，int，float）または構造化データ型です。DynAny インタフェースは，含まれるデータの型を調べるメソッドや，基本データ型の値を設定したり抽出したりするメソッドを提供します。DynAny インタフェースの詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「DynAny」の記述を参照してください。

構造化データ型はすべて，DynAny から派生する表 24-1 に示すインタフェースで表現されます。各インタフェースは，おのおのが含む値を設定または抽出するのに適した独自のメソッドのセットを持ちます。

表 24-1　構造化データ型を表現する DynAny 派生インタフェース

インタフェース	TypeCode	説明
DynArray	_tk_array	固定数の要素を持つ，同じデータ型の値の配列
DynEnum	_tk_enum	単数の列挙体値
DynFixed	_tk_fixed	未サポート
DynSequence	_tk_sequence	同じデータ型の値のシーケンス。要素数は増加または減少します。
DynStruct	_tk_struct	構造体
DynUnion	_tk_union	Union
DynValue	_tk_value	未サポート

24.2.1　使用上の制限事項

DynAny オブジェクトを生成したプロセスだけが，ローカルで DynAny オブジェクトを使用できます。バインドされたオブジェクト用のオペレーション要求で DynAny オブジェクトをパラメタとして使おうとしたり，ORB::object_to_string メソッド（C++）または ORB.object_to_string メソッド（Java）を使って DynAny オブジェクトを文字列化しようとしたりすると，BAD_OPERATION 例外が発生します。

また，DynAny オブジェクトを DII リクエスト中でパラメタとして使おうとすると，BAD_OPERATION 例外が発生します。

このバージョンでは，CORBA 2.5 で指定したように long double および fixed 型はサポートしていません。

24.2.2　DynAny の生成

DynAny オブジェクトは，DynAnyFactory オブジェクトでオペレーションを呼び出して生成します。まず，DynAnyFactory オブジェクトのリファレンスを取得してから，そのオブジェクトを使用して新しい DynAny オブジェクトを生成します。

コードサンプル 24-1　DynAny の生成（C++）

```
CORBA::Object_var obj =
      orb->resolve_initial_references("DynAnyFactory");
DynamicAny::DynAnyFactory_var factory =
      DynamicAny::DynAnyFactory::_narrow(obj);

// Create Dynamic struct
```

```
DynamicAny::DynAny_var dynany =
      factory->create_dyn_any_from_type_code(
            Printer::_tc_StructType);

DynamicAny::DynStruct_var info =
      DynamicAny::DynStruct::_narrow(dynany);

info->set_members(seq);

CORBA::Any_var any = info->to_any();
```

コードサンプル 24-2　DynAny の生成（Java）

```
// Resolve Dynamic Any Factory
   DynAnyFactory factory =
      DynAnyFactoryHelper.narrow(
            orb.resolve_initial_references("DynAnyFactory"));
      byte[ ] oid = "PrinterManager".getBytes();

// Create the printer manager object.
   PrinterManagerImpl manager =
     new PrinterManagerImpl((com.inprise.vbroker.CORBA.ORB)
                            orb, factory, serverPoa, oid);

// Export the newly create object.
   serverPoa.activate_object_with_id(oid, manager);
   System.out.println(manager + " is ready.");
```

24.2.3　DynAny 中の値の初期化とアクセス

DynAny::insert_<type>メソッド（C++）または DynAny.insert_<type>メソッド（Java）を使って，ユーザはさまざまな基本データ型の DynAny オブジェクトを初期化できます。<type>に boolean，octet，char などを指定できます。DynAny に定義された TypeCode と異なる型を挿入すると，TypeMismatch 例外が発生します。

DynAny::get_<type>メソッド（C++）または DynAny.get_<type>メソッド（Java）を使って，ユーザは DynAny オブジェクトに含まれる値にアクセスできます。<type>には boolean，octet，char などを指定できます。DynAny に定義された TypeCode と異なる DynAny コンポーネントから値にアクセスしようとすると，TypeMismatch 例外が発生します。

DynAny インタフェースはまた，Any オブジェクトのコピー，割り当て，および変換（Any オブジェクトへの変換と Any オブジェクトからの変換）のメソッドも提供します。この章に記述されているサンプルプログラムは，幾つかのメソッドの使い方を示します。マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「DynAny」の記述で全メソッドについての説明をしています。

24.3　構造化データ型

ここで説明する型は，DynAny インタフェースから派生したもので，構造化データ型を表現するのに使われます。これらのインタフェースおよびインタフェースが提供するメソッドについては，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「動的インタフェースとクラス」の記述を参照してください。

構造化データ型でコンポーネントを移動する

DynAny から派生したインタフェースの中には，複数のコンポーネントを持つものがあります。DynAny インタフェースは，これらのコンポーネントの中で繰り返しを行うメソッドを提供します。複数のコンポーネントを持つ DynAny 派生オブジェクトは，カレントコンポーネントのポインタを保持します。DynAny メソッドを表 24-2 に示します。

表 24–2　DynAny メソッド

DynAny メソッド	説明
rewind	カレントコンポーネントポインタを最初のコンポーネントに移動します。オブジェクトにコンポーネントが一つしかない場合，効果はありません。
next	ポインタを次のコンポーネントに移動させます。コンポーネントがすでにない場合や，オブジェクトにコンポーネントが一つしかない場合，false が返されます。
current_component	DynAny オブジェクトを返します。DynAny オブジェクトは，コンポーネントの TypeCode に基づいて適切な型にナロウされている場合があります。
seek	カレントコンポーネントポインタを，指定された 0 から始まるインデックス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，false が返されます。負のインデックスで指定されている場合，カレントコンポーネントポインタを-1（コンポーネントなし）に設定します。

24.3.1　DynEnum

このインタフェースは，一つの列挙体定数を表現します。値を文字列または整数値として設定し，取得するメソッドが提供されます。

24.3.2　DynStruct

このインタフェースは，動的に構築された struct 型を表現します。構造体のメンバを検索または設定するには，NameValuePair オブジェクトのシーケンスを使用してください。各 NameValuePair オブジェクトには，メンバの名前と，メンバの型および値を含んだ Any が含まれます。

ユーザは，rewind，next，current_component，および seek メソッドを使って，構造体中のメンバ間を移動できます。構造体のメンバを設定および取得するメソッドが提供されます。

24.3.3　DynUnion

このインタフェースは union を表現し，二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識別子で，二つ目のコンポーネントはメンバの値を表します。

ユーザは，rewind，next，current_component，および seek メソッドを使って，コンポーネント間を移動できます。union の識別子とメンバ値を設定および取得するメソッドが提供されます。

24.3.4　DynSequence と DynArray

DynSequence または DynArray は，基本データ型または構造化データ型のシーケンスを表現します。DynSequence や DynArray を使えば，シーケンスまたは配列の中のコンポーネントごとに個々に DynAny オブジェクトを生成しなくて済みます。DynSequence 中のコンポーネントの数は変化しますが，DynArray 中のコンポーネントの数は固定です。

ユーザは，rewind，next，current_component，および seek メソッドを使って，DynArray または DynSequence のメンバ間を移動できます。

24.4　IDL サンプル

コードサンプル 24-3 に，クライアント／サーバアプリケーションのサンプルで使われている IDL を示します。StructType 構造体には，二つの基本データ型と一つの列挙体の値が含まれています。Any の内容を表示するために PrinterManager インタフェースが使われます。この場合，Any が含むデータ型に関する静的情報は表示されません。

コードサンプル 24-3　DynAny クライアントのサンプルの IDL

```
// Printer.idl
module Printer {
   enum EnumType {first, second, third, fourth};
   struct StructType {
      string str;
      EnumType e;
      float fl;
   };
   interface PrinterManager {
      void printAny(in any info);
      oneway void shutdown();
   };
};
```

24.5　クライアントアプリケーションのサンプル

コードサンプル 24-4 および 24-5 に，Borland Enterprise Server VisiBroker をインストールしたディレクトリの examples/vbe/dynany に入っているクライアントアプリケーションを示します。クライアントアプリケーションは DynStruct インタフェースを使って，動的に StructType 構造体を生成します。

DynStruct インタフェースは，NameValuePair オブジェクトのシーケンスを使って，構造体メンバとメンバに対応する値を表現します。名前・値の各ペアは，構造体のメンバ名を含む文字列と，構造体メンバの値を含む Any オブジェクトで構成されています。

通常の方法で VisiBroker ORB を初期化し，PrintManager オブジェクトにバインドしたあと，クライアントは次の動作をします。

1. 空の DynStruct を適切な型で生成します。
2. 構造体メンバを含むための，NameValuePair オブジェクトのシーケンスを生成します。
3. 構造体メンバの各値用に Any オブジェクトを生成し，初期化します。
4. 各 NameValuePair を，適切なメンバ名と値で初期化します。
5. DynStruct オブジェクトを NameValuePair シーケンスで初期化します。
6. PrinterManager::printAny メソッド（C++）または PrinterManager.printAny メソッド（Java）を呼び出し，通常の Any に変換した DynStruct を渡します。

注

DynAny オブジェクトまたはその派生型の一つをオペレーション要求のパラメタとして渡す前に，DynAny::to_any メソッド（C++）または DynAny.to_any メソッド（Java）を使って，これを Any に変換しておいてください。

コードサンプル 24-4　DynStruct を使用したクライアントアプリケーションのサンプル（C++）

```
// Client.C

#include "Printer_c.hh"
#include "dynany.h"

int main(int argc, char* const* argv) {
   try {
   // Initialize the ORB.
   CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
   DynamicAny::DynAnyFactory_var factory =
      DynamicAny::DynAnyFactory::_narrow(
            orb->resolve_initial_references(
                  "DynAnyFactory"));
   // Get the manager Id
   PortableServer::ObjectId_var managerId =
         PortableServer::string_to_ObjectId("PrinterManager");
   // Locate an account manager. Give the full POA name
   // and the servant ID.
   Printer::PrinterManager_ptr manager =
         Printer::PrinterManager::_bind(
               "/serverPoa", managerId);

   DynamicAny::NameValuePairSeq seq(3);
   seq.length(3);

   CORBA::Any strAny,enumAny,floatAny;
   strAny <<= "String";
   enumAny <<= Printer::second;
   floatAny <<= (CORBA::Float)864.50;

   CORBA::NameValuePair nvpairs[3];

   nvpairs[0].id = CORBA::string_dup("str");
```

```
   nvpairs[0].value = strAny;

   nvpairs[1].id = CORBA::string_dup("e");
   nvpairs[1].value = enumAny;

   nvpairs[2].id = CORBA::string_dup("fl");
   nvpairs[2].value = floatAny;

   seq[0] = nvpairs[0];
   seq[1] = nvpairs[1];
   seq[2] = nvpairs[2];

   // Create Dynamic struct
   DynamicAny::DynStruct_var info =
      DynamicAny::DynStruct::_narrow(
      factory->create_dyn_any_from_type_code(
            Printer::_tc_StructType));

   info->set_members(seq);
      manager->printAny(*(info->to_any()));
      manager->shutdown();
   }
   catch(const CORBA::Exception& e) {
      cerr << "Caught " << e << "Exception" << endl;
   }
}
```

コードサンプル 24-5　DynStruct を使用したクライアントアプリケーションのサンプル（Java）

```
// Client.java

import org.omg.DynamicAny.*;

public class Client {

   public static void main(String[ ] args) {
      try {
         // Initialize the ORB.
         org.omg.CORBA.ORB orb =
                        org.omg.CORBA.ORB.init(args, null);

         DynAnyFactory factory = DynAnyFactoryHelper.narrow(
            orb.resolve_initial_references("DynAnyFactory"));

         // Locate a printer manager.
         Printer.PrinterManager manager =
            Printer.PrinterManagerHelper.bind(orb,
                                        "PrinterManager");

         // Create Dynamic struct
         DynStruct info =
            DynStructHelper.narrow(factory.
               create_dyn_any_from_type_code(
                  Printer.StructTypeHelper.type()));

         // Create our NameValuePair sequence (array)
         NameValuePair[ ] NVPair = new NameValuePair[3];

         // Create and initialize Dynamic Struct data as any's
         org.omg.CORBA.Any str_any = orb.create_any();
         str_any.insert_string("String");
         org.omg.CORBA.Any e_any = orb.create_any();
         Printer.EnumTypeHelper.insert(
                           e_any, Printer.EnumType.second);
         org.omg.CORBA.Any fl_any = orb.create_any();
         fl_any.insert_float((float)864.50);

         NVPair[0] = new NameValuePair("str", str_any);
         NVPair[1] = new NameValuePair("e", e_any);
         NVPair[2] = new NameValuePair("fl", fl_any);

         // Initialize the Dynamic Struct
         info.set_members(NVPair);

         manager.printAny(info.to_any());
```

```
            manager.shutdown();
        }
        catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

24.6　サーバアプリケーションのサンプル

コードサンプル 24-6 に，Borland Enterprise Server VisiBroker をインストールしたディレクトリの examples/vbe/dynany に入っているサーバアプリケーションを示します。サーバアプリケーションは次の動作を行います。

1. VisiBroker ORB を初期化します。
2. POA のポリシーを生成します。
3. PrintManager オブジェクトを生成します。
4. PrintManager オブジェクトをエクスポートします。
5. メッセージを出力し，オペレーション要求が入力されるのを待ちます。

コードサンプル 24-6　サーバアプリケーションのサンプル（C++）

```
. . .
int main(int argc, char* const* argv) {

   try {
      // Initialize the ORB.
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      int Verbose = 0;
      // get a reference to the rootPOA
      PortableServer::POA_var rootPOA =
      PortableServer::POA::_narrow(
            orb->resolve_initial_references("RootPOA"));
      CORBA::PolicyList policies;
      policies.length(1);
      policies[(CORBA::ULong)0] =
            rootPOA->create_lifespan_policy(
            PortableServer::PERSISTENT);
      // Create serverPOA with the right policies
      PortableServer::POA_var serverPOA =
            rootPOA->create_POA("serverPoa",
            rootPOA->the_POAManager(),
            policies );
      // Resolve Dynamic Any Factory
      DynamicAny::DynAnyFactory_var factory =
            orb->resolve_initial_references("DynAnyFactory");
      PortableServer::ObjectId_var managerId =
            PortableServer::string_to_ObjectId(
            "PrinterManager");
      // Create the printer manager object.
      PrinterManagerImpl manager(
            orb, factory, serverPOA, managerId);
      // Export the newly create object.
      serverPOA->activate_object_with_id(managerId,&manager);
      // Activate the POA Manager
      rootPOA->the_POAManager()->activate();
      cout << serverPOA->servant_to_reference(&manager)
            << "is ready" << endl;
      // Wait for incoming requests
      orb->run();
   }
   catch(const CORBA::Exception& e) {
      cerr << e << endl;
   }
}
```

コードサンプル 24-7　サーバアプリケーションのサンプル（Java）

```
// Server.java

import java.util.*;
import org.omg.DynamicAny.*;
import org.omg.PortableServer.*;
import com.inprise.vbroker.PortableServerExt.*;
```

```
public class Server {

   public static void main(String[ ] args) {
      try {
         // Initialize the ORB.
         org.omg.CORBA.ORB orb =
                        org.omg.CORBA.ORB.init(args,null);

         // Resolve RootPOA
         POA rootPoa = POAHelper.narrow(
                  orb.resolve_initial_references("RootPOA"));
         rootPoa.the_POAManager().activate();

         // Create a BindSupport Policy that makes POA
         // register each servant with osagent
         org.omg.CORBA.Any any = orb.create_any();
         BindSupportPolicyValueHelper.insert(any,
                        BindSupportPolicyValue.BY_INSTANCE);

         org.omg.CORBA.Policy bsPolicy = orb.create_policy(
                        BIND_SUPPORT_POLICY_TYPE.value, any);

         // Create policies for our testPOA
         org.omg.CORBA.Policy[ ] policies = {
            rootPoa.create_lifespan_policy(
               LifespanPolicyValue.PERSISTENT),
            bsPolicy
         };

         // Create managerPOA with the right policies
         POA serverPoa = rootPoa.create_POA(
           "serverPoa", rootPoa.the_POAManager(), policies);

         // Resolve Dynamic Any Factory
         DynAnyFactory factory =
            DynAnyFactoryHelper.narrow(
            orb.resolve_initial_references("DynAnyFactory"));

         byte[ ] oid = "PrinterManager".getBytes();

         // Create the printer manager object.
         PrinterManagerImpl manager = new PrinterManagerImpl(
            (com.inprise.vbroker.CORBA.ORB)orb,
            factory, serverPoa, oid);

         // Export the newly create object.

         serverPoa.activate_object_with_id(oid, manager);

         System.out.println(manager + "is ready.");
   // Wait for incoming requests
         orb.run();
      }
      catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

コードサンプル 24-8 は，PrinterManager インプリメンテーションが，次の手順で，コンパイル時に Any が含む型を知らないで，DynAny を使って Any オブジェクトを処理する様子を示します。

1. DynAny オブジェクトを生成し，受け取った Any で初期化します。
2. DynAny オブジェクトタイプに対して switch を実行します。
3. DynAny が基本データ型を含む場合，値を出力します。
4. DynAny が Any 型を含む場合，Any 用に DynAny を生成し，内容を確定し，値を出力します。
5. DynAny が enum を含む場合，enum 用に DynEnum を生成し，文字列値を出力します。

6. DynAny が union を含む場合，union 用に DynUnion を生成し，union の識別子とメンバを出力します。
7. DynAny が struct，array，または sequence を含む場合，含まれるコンポーネント間を移動し，各値を出力します。

コードサンプル 24-8　PrinterManager のインプリメンテーション（C++）

```
// PrinterManager Implementation
class PrinterManagerImpl : public POA_Printer::PrinterManager
{
   CORBA::ORB_var           _orb;
   DynamicAny::DynAnyFactory_var   _factory;
   PortableServer::POA_var         _poa;
   PortableServer::ObjectId_var    _oid;

public:
   PrinterManagerImpl(CORBA::ORB_ptr orb,
      DynamicAny::DynAnyFactory_ptr DynAnyFactory,
      PortableServer::POA_ptr poa,
      PortableServer::ObjectId_ptr oid
      ) : _orb(orb), _factory(DynAnyFactory),
          _poa(poa), _oid(oid) {}

   void printAny(const CORBA::Any& info) {
   try {
      // Create a DynAny object
      DynamicAny::DynAny_var dynAny =
          _factory->create_dyn_any(info); display(dynAny);
   }
   catch (CORBA::Exception& e) {
      cout << "Unable to create Dynamic Any from factory"
           << endl;
   }
}
   void shutdown() {
   try {
      _poa->deactivate_object(_oid);
       cout << "Server shutting down..." << endl;
      _orb->shutdown(0UL);
   }
catch (const CORBA::Exception& e){
   cout << e << endl;
   }
}

void display(DynamicAny::DynAny_var value) {
   switch(value->type()->kind()) {
   case CORBA::tk_null:
    case CORBA::tk_void: {
     break;
    }
    case CORBA::tk_short: {
      cout << value->get_short() << endl;
      break;
   }
   case CORBA::tk_ushort: {
     cout << value->get_ushort() << endl;
     break;
    }
   case CORBA::tk_long: {
   cout << value->get_long() << endl;
   break;
   }
   case CORBA::tk_ulong: {
    cout << value->get_ulong() << endl;
    break;
   }
   case CORBA::tk_float: {
     cout << value->get_float() << endl;
     break;
   }
   case CORBA::tk_double: {
```

```
  cout << value->get_double() << endl;
  break;
}
case CORBA::tk_boolean: {
  cout << value->get_boolean() << endl;
  break;
}
case CORBA::tk_char: {
  cout << value->get_char() << endl;
  break;
}
case CORBA::tk_octet: {
  cout << value->get_octet() << endl;
  break;
}
case CORBA::tk_string: {
  cout << value->get_string() << endl;
  break;
}
case CORBA::tk_any: {
  DynamicAny::DynAny_var dynAny =
        _factory->create_dyn_any(*(value->get_any()));
  display(dynAny);
  break;
}
case CORBA::tk_TypeCode: {
  cout << value->get_typecode() << endl;
  break;
}
case CORBA::tk_objref: {
  cout << value->get_reference() << endl;
  break;
}
case CORBA::tk_enum: {
  DynamicAny::DynEnum_var dynEnum =
        DynamicAny::DynEnum::_narrow(value);
  cout << dynEnum->get_as_string() << endl;
  break;
}
case CORBA::tk_union: {
  DynamicAny::DynUnion_var dynUnion =
        DynamicAny::DynUnion::_narrow(value);
   display(dynUnion->get_discriminator());
   display(dynUnion->member());
   break;
   }
   case CORBA::tk_struct:
   case CORBA::tk_array:
   case CORBA::tk_sequence: {
     value->rewind();
   CORBA::Boolean next = 1UL;
     while(next) {
     DynamicAny::DynAny_var d =
           value->current_component();
       display(d);
       next =value->next();
     }
     break;
   }
   case CORBA::tk_longlong: {
     cout << value->get_longlong() << endl;
     break;
   }
   case CORBA::tk_ulonglong: {
     cout << value->get_ulonglong() << endl;
     break;
   }
   case CORBA::tk_wstring: {
     cout << value->get_wstring() << endl;
     break;
   }
   case CORBA::tk_wchar: {
     cout << value->get_wchar() << endl;
     break;
```

```
        }
        default:
          cout << "Invalid Type" << endl;
        }
    }
};
```

コードサンプル 24-9　PrinterManager のインプリメンテーション（Java）

```
// PrinterManagerImpl.java

import java.util.*;
import org.omg.DynamicAny.*;
import org.omg.PortableServer.*;

public class PrinterManagerImpl extends
                              Printer.PrinterManagerPOA {
   private com.inprise.vbroker.CORBA.ORB _orb;
   private DynAnyFactory _factory;
   private POA _poa;
   private byte[ ] _oid;

   public PrinterManagerImpl(
         com.inprise.vbroker.CORBA.ORB orb,
         DynAnyFactory factory, POA poa, byte[ ] oid) {
      _orb = orb;
      _factory = factory;
      _poa = poa;
      _oid = oid;
   }

   public synchronized void printAny(org.omg.CORBA.Any info) {
      // Display info with the assumption that we don't have
      // any info statically about the type inside the any

   try {
      // Create a DynAny object
      DynAny dynAny = _factory.create_dyn_any(info);
      display(dynAny);
   }
   catch (Exception e) {
      e.printStackTrace();
   }

}

public void shutdown() {
   try {
      _poa.deactivate_object(_oid);
      System.out.println("Server shutting down");
      _orb.shutdown(false);
   }
   catch (Exception e) {
      System.out.println(e);
   }
}

private void display(DynAny value) throws Exception {
   switch(value.type().kind().value()) {
      case org.omg.CORBA.TCKind._tk_null:
      case org.omg.CORBA.TCKind._tk_void: {
         break;
      }
      case org.omg.CORBA.TCKind._tk_short: {
         System.out.println(value.get_short());
         break;
      }
      case org.omg.CORBA.TCKind._tk_ushort: {
         System.out.println(value.get_ushort());
         break;
      }
      case org.omg.CORBA.TCKind._tk_long: {
         System.out.println(value.get_long());
         break;
      }
```

```
case org.omg.CORBA.TCKind._tk_ulong: {
   System.out.println(value.get_ulong());
   break;
}
case org.omg.CORBA.TCKind._tk_float: {
   System.out.println(value.get_float());
   break;
}
case org.omg.CORBA.TCKind._tk_double: {
   System.out.println(value.get_double());
   break;
}
case org.omg.CORBA.TCKind._tk_boolean: {
   System.out.println(value.get_boolean());
   break;
}
case org.omg.CORBA.TCKind._tk_char: {
   System.out.println(value.get_char());
   break;
}
case org.omg.CORBA.TCKind._tk_octet: {
   System.out.println(value.get_octet());
   break;
}
case org.omg.CORBA.TCKind._tk_string: {
   System.out.println(value.get_string());
   break;
}
case org.omg.CORBA.TCKind._tk_any: {
   DynAny dynAny =
          _factory.create_dyn_any(value.get_any());
   display(dynAny);
   break;
}
case org.omg.CORBA.TCKind._tk_TypeCode: {
   System.out.println(value.get_typecode());
   break;
}
case org.omg.CORBA.TCKind._tk_objref: {
   System.out.println(value.get_reference());
   break;
}
case org.omg.CORBA.TCKind._tk_enum: {
   DynEnum dynEnum = DynEnumHelper.narrow(value);
   System.out.println(dynEnum.get_as_string());
   break;
}
case org.omg.CORBA.TCKind._tk_union: {
   DynUnion dynUnion = DynUnionHelper.narrow(value);
   display(dynUnion.get_discriminator());
   display(dynUnion.member());
   break;
}
case org.omg.CORBA.TCKind._tk_struct:
case org.omg.CORBA.TCKind._tk_array:
case org.omg.CORBA.TCKind._tk_sequence: {
   value.rewind();
   boolean next = true;
   while(next) {
      DynAny d = value.current_component();
      display(d);
      next = value.next();
   }
   break;
}
case org.omg.CORBA.TCKind._tk_longlong: {
   System.out.println(value.get_longlong());
   break;
}
case org.omg.CORBA.TCKind._tk_ulonglong: {
   System.out.println(value.get_ulonglong());
   break;
}
case org.omg.CORBA.TCKind._tk_wstring: {
```

```
            System.out.println(value.get_wstring());
            break;
        }
        case org.omg.CORBA.TCKind._tk_wchar: {
            System.out.println(value.get_wchar());
            break;
        }
        default:
            System.out.println("Invalid type");


        }
    }
}
```

25 valuetype の使用

この章では，Borland Enterprise Server VisiBroker の valuetype IDL 型の使用方法を説明します。

25.1　valuetype とは

IDL の valuetype は，ネットワーク越しに状態データを渡す際に使用されます。valuetype は，継承とメソッドを持つ struct と考えることができます。valuetype は通常のインタフェースとは異なり，valuetype の状態を記述するプロパティを含み，インタフェース以外にインプリメンテーションの詳細も含みます。次の IDL コードは単純な valuetype を宣言します。

IDL サンプル 25-1　単純な valuetype IDL

```
module Map {
   valuetype Point {
      public long x;
      public long y;
      private string label;
      factory create ( in long x, in long y, in string z);
      void print();
   };
};
```

valuetype は，常にローカルなものです。valuetype は VisiBroker ORB に登録されることもなく，一意の識別子も必要ありません。それは valuetype の値自体が識別子のためです。valuetype はリモートで呼び出すことはできません。

25.1.1　concrete valuetype

concrete valuetype には，状態データがあります。次のように通常の IDL の表現形式を拡張したものです。

- 一つの concrete valuetype の派生および複数の abstract valuetype の派生
- 複数のインタフェースのサポート（一つの concrete および複数の abstract）
- 任意の再帰的な valuetype の定義
- ヌルセマンティクス
- 共用セマンティクス

(1)　valuetype の派生

concrete valuetype を別の concrete valuetype から派生させることができますが，valuetype は，別の複数の abstract valuetype から派生させることもできます。

(2)　共用セマンティクス

valuetype インスタンスは，別のインスタンスの valuetype と共用できます。struct，union，または sequence のようなほかの IDL データ型は共用できません。共用される valuetype は，送信コンテキストおよび受信コンテキスト間で同じ型です。

さらに，同じ valuetype が複数の引数のオペレーションに渡されると，受信コンテキストは両方の引数の同じ valuetype リファレンスを受け取ります。

(3)　ヌルセマンティクス

ヌル valuetype は，struct，union，sequence のような IDL データ型と異なり，ネットワーク越しに渡せます。例えば，struct をボックス型 valuetype としてボックス化すると，ヌル値 struct を渡せます。詳細については，「25.4　ボックス型 valuetype」を参照してください。

(4)　ファクトリ

ファクトリは，移植性のある valuetype を生成するために，valuetype で宣言できるメソッドです。ファクトリの詳細については，「25.3　ファクトリのインプリメント」を参照してください。

25.1.2　abstract valuetype

abstract valuetype にはメソッドだけがあり，状態はありません。abstract valuetype は実体化できません。abstract valuetype は，単にローカルインプリメンテーションのあるオペレーションのシグニチャの固まりです。

例えば, 次の IDL は状態を含みませんが, 一つのメソッド get_name を含む abstract valuetype Account を定義します。

```
abstract valuetype Account{
   string get_name();
}
```

ここで, 二つの valuetype が定義されます。これは abstract valuetype から get_name メソッドを継承します。

```
valuetype savingsAccount:Account{
   private long balance;
}
valuetype checkingAccount:Account{
   private long balance;
}
```

これら二つの valuetype には変数 balance があり，abstract valuetype Account から get_name メソッドを継承します。

25.2　valuetype のインプリメント

アプリケーションで valuetype をインプリメントするには，次の手順を行います。

1. IDL ファイルで valuetype を定義します。
2. 次のコンパイラを使用して IDL ファイルをコンパイルします。
   - C++の場合
     idl2cpp
   - Java の場合
     idl2java
3. valuetype ベースクラスを継承して valuetype をインプリメントします。
4. IDL で定義したファクトリメソッドをインプリメントするために Factory クラスをインプリメントします。
5. create_for_unmarshal メソッドをインプリメントします。
6. C++の場合，Factory を VisiBroker ORB に登録します。
   Java の場合，必要であれば，Factory を ORB に登録します。
7. _add_ref，_remove_ref，_ref_countvalue メソッドをインプリメントするか，または CORBA::DefaultValueRefCountBase から派生させます。

25.2.1　valuetype の定義

IDL サンプル 25-1 では，Point という名前の valuetype を定義します。この valuetype はグラフのポイントを定義します。これには，二つのパブリック変数（x 座標と y 座標），ポイントのラベルであるプライベート変数一つ，valuetype のファクトリ，およびポイントを出力する出力メソッドがあります。

25.2.2　IDL ファイルのコンパイル

これで IDL を定義できたので，C++ソースファイルを作成するために idl2cpp を使用してコンパイルするか，または C++ソースファイルを作成するために idl2java を使用してコンパイルします。これによって，valuetype をインプリメントするために修正する Java ソースファイルが作成されます。上記の IDL をコンパイルすると，出力は次のファイルで構成されます。

- C++の場合
  - Map_c.cc
  - Map_c.hh
  - Map_s.cc
  - Map_s.hh
- Java の場合
  - Point.java
  - PointDefaultFactory.java
  - PointHelper.java
  - PointHolder.java
  - PointValueFactory.java

25.2.3　valuetype ベースクラスの継承

IDL のコンパイル後，valuetype のインプリメンテーションを作成します。インプリメンテーションクラスは，ベースクラスを継承します。このクラスには，ValueFactory で呼び出されたコンストラクタがあり，IDL で宣言された変数とメソッドのすべてを含んでいます。

例えば obv¥PntImpl.h では，コードサンプル 25-1 で示すように PointImpl クラスは IDL から生成された Point クラスを継承します。

例えば obv¥PointImpl.java では，コードサンプル 25-2 で示すように PointImpl クラスは IDL から生成された Point クラスを継承します。

コードサンプル 25-1　valuetype ベースクラスの継承（C++）

```
class PointImpl : public Map::OBV_Point,
      public CORBA::DefaultValueRefCountBase {
   public:
      PointImpl(){}
      virtual ~PointImpl(){}
      CORBA_ValueBase* _copy_value() {
         return new PointImpl(x(), y(),
               new Map::Label(CORBA::string_dup(label())));
      }
      PointImpl( CORBA::Long x, CORBA::Long y,
            Map::Label_ptr label )
            : OBV_Point( x,y,label->_boxed_in())
      {}
      virtual void print() {
         cout << "Point is [" << label() << ": ("
            << x() << ", " << y() << ")]" << endl << endl;
      }
};
```

コードサンプル 25-2　valuetype ベースクラスの継承（Java）

```
public class PointImpl extends Point {
   public PointImpl() {}
   public PointImpl(int a_x, int a_y, String a_label) {
      x = a_x;
      y = a_y;
      label = a_label;
   }
   public void print() {
      System.out.println("Point is [" + label +
                         ": (" + x + ", " + y + ")]");
   }
}
```

25.2.4　Factory クラスのインプリメント

これでインプリメンテーションクラスを生成したので，valuetype の Factory をインプリメントします。

次に示すサンプルでは，生成された Point_init クラスには IDL で宣言された create メソッドがあります。このクラスは CORBA::ValueFactoryBase（C++）または org.omg.CORBA.portable.ValueFactory（Java）を継承します。コードサンプル 25-3 および 25-4 で示すように PointDefaultFactory クラスは PointValueFactory をインプリメントします。

コードサンプル 25-3　Factory クラスのインプリメント（C++）

```
class PointFactory: public CORBA::ValueFactoryBase {
   public:
      PointFactory(){}
      virtual ~PointFactory(){}
      CORBA::ValueBase* create_for_unmarshal() {
         return new PointImpl();
```

```
        }
};
```

C++の場合，Point_init には，パブリックメソッドである create_for_unmarshal があり，これは Map_c.hh の純仮想メソッドとして出力されます。ユーザは Point_init からクラスを派生させ，create_for_unmarshal メソッドをインプリメントして Factory クラスを生成する必要があります。IDL ファイルをコンパイルする場合，IDL ファイル用のスケルトンクラスを生成しません。

コードサンプル 25-4　Factory クラスのインプリメント（Java）

```
public class PointDefaultFactory implements
                                        PointValueFactory {
   public java.io.Serializable read_value (
                  org.omg.CORBA.portable.InputStream is) {
      java.io.Serializable val =
          new PointImpl(); // Called the implementation class
      // create and initialize value
      val = ((org.omg.CORBA_2_3.portable.InputStream)is).
                                          read_value(val);
      return val;
   }
   // It is up to the user to implement the valuetype
   // however they want:
   public Point create (int x,
         int y,
         java.lang.String z) {
      //IMPLEMENT:
      return null;
   }
}
```

新しい valuetype を作成するために PointImpl()が呼び出されます。新しい valuetype は，read_value によって InputStream から読み込まれます。

注（Java の場合）

ユーザは read_value を呼び出す必要があります。これを呼び出さないと，ファクトリが動作しないで，ほかのどのメソッドも呼び出せません。

25.2.5　VisiBroker ORB への Factory の登録

(1)　C++の場合

ORB::register_value_factory を呼び出して，VisiBroker ORB に Factory を登録します。ファクトリの登録の詳細については，「25.3.2　valuetype の登録」を参照してください。

(2)　Java の場合

ORB.register_value_factory を呼び出して，VisiBroker ORB に Factory を登録します。これは，ファクトリに valuetypenameDefaultFactory という名前を付けない場合だけ必要です。ファクトリの登録の詳細については，「25.3.2　valuetype の登録」を参照してください。

25.3　ファクトリのインプリメント

VisiBroker ORB が valuetype を受け取ると，まずデマーシャルする必要があります。それから，その型の適切なファクトリを，その型の新しいインスタンスを生成するために見つけなければなりません。インスタンスが生成されたら，値データはインスタンスにアンマーシャルされます。その型は呼び出しのときに渡される RepositoryID によって識別されます。型とファクトリ間のマッピングは，言語固有です。

コードサンプル 25-5 および 25-6 に，JDK1.2 を使用した Point valuetype のファクトリのサンプルインプリメンテーションを示します。

コードサンプル 25-5　Point valuetype のファクトリ（C++）

```
class PointFactory: public CORBA::ValueFactoryBase
{
public:
   PointFactory(){}
   virtual ~PointFactory(){}
   CORBA::ValueBase* create_for_unmarshal() {
      return new PointImpl();
   }
};
```

コードサンプル 25-6　Point valuetype のファクトリ（Java）

```
public class PointDefaultFactory implements
                                        PointValueFactory {
   public java.io.Serializable read_value(
                     org.omg.CORBA.portable.InputStream is) {
   java.io.Serializable val = new PointImpl();
   // create and initialize value
   // It is very important that this call is made.
   val = ((org.omg.CORBA_2_3.portable.InputStream)is).
                                           read_value(val);
   return val;
   }
   public Point create (int x, int y, java.lang.String z) {
      // IMPLEMENT:
   return NO_IMPLEMENT;
   }
}
```

Borland Enterprise Server VisiBroker 4.5 以降のバージョンは，JDK 1.2 または JDK 1.3 のデフォルト値ファクトリメソッドの正しいシグニチャを生成します。既存（Borland Enterprise Server VisiBroker 4.0）の生成コードは，次のようにデフォルト値ファクトリメソッドシグニチャを修正しないかぎり，JDK 1.3 で実行するように設計されていません。既存のコードを JDK 1.3 で使用し，デフォルト値ファクトリを修正しないと，コードはコンパイルされません。または NO_IMPLEMENT 例外が発生します。したがって，正しいシグニチャを生成するにはコードを再生成することをお勧めします。

次のコードサンプルに，JDK 1.3 でコンパイルできるようにデフォルト値ファクトリメソッドシグニチャを修正する方法を示します。

コードサンプル 25-7　JDK 1.3 コード生成でのメソッドシグニチャを示すファクトリコード（C++）

```
class PointFactory: public CORBA::ValueFactoryBase
{
public:
   PointFactory(){}
   virtual ~PointFactory(){}
   CORBA::ValueBase* create_for_unmarshal() {
      return new PointImpl();
   }
};
```

コードサンプル 25-8　JDK 1.3 コード生成でのメソッドシグニチャを示すファクトリコード（Java）

```
public class PointDefaultFactory implements
                                        PointValueFactory {
```

```
    public java.io.Serializable read_value (
                org.omg.CORBA_2_3.portable.InputStream is) {
    java.io.Serializable val = new PointImpl();
    // create and initialize value
    // It is very important that this call is made.
    val = ((org.omg.CORBA_2_3.portable.InputStream)is).
                                            read_value(val);
    return val;
    }
    public Point create (int x, int y, java.lang.String z) {
       // IMPLEMENT:
    return NO_IMPLEMENT;
    }
}
```

25.3.1 ファクトリと valuetype

VisiBroker ORB が valuetype を受け取ると，その型のファクトリを探します。ファクトリ名が valuetypeDefaultFactory のファクトリを探します。例えば，Point valuetype のファクトリは PointDefaultFactory になります。正しいファクトリがこのネーミングスキーマ（valuetypeDefaultFactory）に準拠しない場合，正しいファクトリを登録し，VisiBroker ORB が valuetype のインスタンスを生成できるようにする必要があります。

VisiBroker ORB が指定の valuetype の正しいファクトリを見つけられない場合，MARSHAL 例外が発生し，マイナーコードを示します。

25.3.2 valuetype の登録

それぞれの言語マッピングによって，登録の方法とタイミングを指定します。valuetypeDefaultFactory ネーミング規則によってファクトリを生成した場合，そのファクトリは暗黙的に登録されるので，VisiBroker ORB にファクトリを明示的に登録する必要はありません。

valuetypeDefaultFactory ネーミング規則に準拠しないファクトリを登録するには，register_value_factory を呼び出します。ファクトリの登録を解除するには，VisiBroker ORB で unregister_value_factory を呼び出します。また，VisiBroker ORB で lookup_value_factory を呼び出して，登録された valuetype ファクトリを探すこともできます。

25.4　ボックス型 valuetype

ボックス型 valuetype は，値の定義がない IDL データ型を valuetype としてラッピングできるものです。例えば，IDL ボックス型 valuetype 宣言（valuetype Label string;）は，次の IDL valuetype 宣言と同じです。

```
valuetype Label{
   public string name;
}
```

ほかのデータ型を valuetype としてボックス化することによって，ユーザは valuetype のヌルセマンティクスと共用セマンティクスを使用できます。

valuebox は，生成されたコードだけでインプリメントされます。ユーザは特殊なコードを追加する必要はありません。

25.5　abstract インタフェース

abstract インタフェースは，オブジェクトを値渡しするのか参照渡しするのかを実行時に選択できるインタフェースです。

abstract インタフェースは IDL インタフェースとは次の点で異なります。

- 実際のパラメタ型は，オブジェクトがリファレンスによって渡されるか，valuetype が渡されるかを決めます。このパラメタ型は二つのルールに基づいて決められます。パラメタ型が通常のインタフェース型，またはサブタイプの場合，インタフェース型が abstract インタフェース型のシグニチャのサブタイプの場合，およびオブジェクトがすでに VisiBroker ORB に登録されている場合は，オブジェクトリファレンスとして処理されます。また，パラメタ型がオブジェクトリファレンスとして渡すことはできませんが，値として渡せる場合は，値として処理されます。値として渡すのに失敗すると，BAD_PARAM 例外が発生します。
- abstract インタフェースは CORBA::Object（C++）または org.omg.CORBA.Object（Java）から暗黙的に派生しません。それは，abstract インタフェースがオブジェクトリファレンスか valuetype のどちらかを表すためです。valuetype は，必ずしも共通オブジェクトリファレンスオペレーションをサポートする必要はありません。abstract インタフェースのオブジェクトリファレンス型へのナロウに成功したら，CORBA::Object（C++）または org.omg.CORBA.Object（Java）のオペレーションを呼び出せます。
- abstract インタフェースはほかの abstract インタフェースだけから継承できます。
- valuetype は，複数の abstract インタフェースをサポートできます。

例えば，次の abstract インタフェースについて次に示します。

IDL サンプル 25-2　abstract インタフェース IDL

```
abstract interface ai{
};
interface itp : ai{
};
valuetype vtp supports ai{
};
interface x {
   void m(ai aitp);
};
valuetype y {
   void op(ai aitp);
};
```

メソッド m の引数では

- itp は常に，オブジェクトリファレンスとして渡されます。
- vtp は値として渡されます。

25.6　custom valuetype

IDL で custom valuetype を宣言することによって，デフォルトのマーシャルモデルとアンマーシャルモデルをスキップできますが，ユーザはエンコーディングとデコーディングの責任を負います。

IDL サンプル 25-3　custom valuetype IDL

```
custom valuetype customPoint{
   public long x;
   public long y;
   private string label;
   factory create(in long x, in long y, in string z);
};
```

ユーザは, CustomMarshal インタフェースからマーシャルおよびアンマーシャルメソッドをインプリメントする必要があります。

C++の場合

custom valuetype を宣言する場合，正規の valuetype である CORBA::StreamableValue とは対照的に，valuetype は CORBA::CustomValue を継承します。コンパイラは，ユーザの valuetype 用の読み込みメソッド，書き込みメソッドを生成しません。

ユーザは，値の読み込み，書き込みを行うために CORBA::DataInputStream と CORBA::DataOutputStream をそれぞれ使用して自分の読み込みメソッド，書き込みメソッドをインプリメントする必要があります。

Java の場合

custom valuetype を宣言する場合，正規の valuetype である org.omg.CORBA.portable.StreamableValue とは対照的に，valuetype は org.omg.CORBA.portable.CustomValue を継承します。コンパイラは，ユーザの valuetype 用の読み込みメソッド，書き込みメソッドを生成しません。

ユーザは，値の読み込み，書き込みを行うために org.omg.CORBA.portable.DataInputStream と org.omg.CORBA.portable.DataOutputStream をそれぞれ使用して自分の読み込みメソッド, 書き込みメソッドをインプリメントする必要があります。

25.7　truncatable valuetype

truncatable valuetype によって，ユーザは継承した valuetype をその親として処理できます。

次の IDL は，ベースタイプ Account から継承した valuetype checkingAccount を定義しています。また，valuetype checkingAccount は受信オブジェクトを truncate できます。

```
valuetype checkingAccount: truncatable Account{
   private long balance;
}
```

これは，派生した valuetype で受信コンテキストが新しいデータメンバやメソッドを必要としない場合や，受信コンテキストが派生した valuetype を認識していない場合に役立ちます。しかし，親データ型に存在しない valuetype から派生した状態データは，valuetype が受信コンテキストに渡されると失われます。

注

custom valuetype を truncatable にはできません。

26 URL ネーミングの使用

この章では，URL ネーミングサービスを使用して URL をオブジェクトの IOR に対応させる方法について説明します。いったん URL がオブジェクトにバインドされると，クライアントアプリケーションはオブジェクト名の代わりにその URL を文字列として指定することによって，そのオブジェクトのリファレンスを取得できます。osagent も CORBA ネーミングサービスも使用しないでオブジェクトの場所をクライアントアプリケーションで探したい場合は，代わりに URL を指定する方法があります。

なお，詳細の説明は Java についてのものだけです。

26.1　URL ネーミングサービス

URL ネーミングサービスは，サーバオブジェクトにその IOR をファイル内の文字列の形式で URL に対応させる簡単な方式です。クライアントプログラムは Web サーバ上の文字列化された URL を含むファイルを指している URL を使用して，そのオブジェクトの場所を探せます。URL ネーミングサービスは，オブジェクト登録用の http URL 体系や，Java ランタイムがサポートする URL 体系（http:，ftp:，file:のような URL によるオブジェクトの探索用）をサポートします。

この URL ネームサービスは，スマートエージェントまたは CORBA ネーミングサービスを使用しないでオブジェクトの場所を探す方法を提供します。これによって, クライアントアプリケーションはベンダが提供したオブジェクトの場所を探せます。このサービスの IDL 仕様を IDL サンプル 26-1 に示します。

注

Borland Enterprise Server VisiBroker の URL ネーミングサービスは，Java 環境がサポートするすべての URL 処理形式をサポートします。

IDL サンプル 26-1　WebNaming モジュール

```
// WebNaming.idl
#pragma prefix "inprise.com"
module URLNaming {
   exception InvalidURL{string reason;};
   exception CommFailure{string reason;};
   exception ReqFailure{string reason;};
   exception AlreadyExists{string reason;};
   abstract interface Resolver {
   // Read Operations
      Object locate(in string url_s)
         raises (InvalidURL, CommFailure, ReqFailure);
   // Write Operations
      void force_register_url(in string url_s, in Object obj)
         raises (InvalidURL, CommFailure, ReqFailure);
      void register_url(in string url_s, in Object obj)
         raises (InvalidURL, CommFailure, ReqFailure,
                 AlreadyExists);
   };
};
```

26.2　オブジェクトの登録

オブジェクトサーバは，まず Resolver にバインドし，次に register_url メソッドまたは force_register_url メソッドを使用して URL をオブジェクトの IOR に対応させることによってオブジェクトを登録します。まだ URL がオブジェクトの IOR に対応づけがされていなければ対応させるために register_url を使用します。force_register_url メソッドを使用すると，URL がオブジェクトにバインドされていてもいなくても，URL をそのオブジェクトの IOR に対応させます。これに対して同じ状況で register_url メソッドを使用すると，AlreadyExists 例外が発生します。利用できるすべてのメソッドについては，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「URL ネーミングインタフェースとクラス（Java)」の記述を参照してください。

サーバ側でのこの機能の使用例については，コードサンプル 26-1 を参照してください。ここに示すサンプルでは，force_register_url を使用しています。force_register_url が成功するには，Web サーバによって HTTP PUT コマンドの発行が許可されている必要があります。この章で示すサンプルのコードは，Borland Enterprise Server VisiBroker をインストールしたディレクトリの examples/vbe/basic 下の bank_URL に入っています。

注

Resolver のリファレンスを取得するには，サンプルに示すように VisiBroker ORB の resolve_initial_references メソッドを使用します。

コードサンプル 26-1　URL とオブジェクトの IOR の対応

```
. . .
public class Server {
   public static void main(String[ ] args) {
      if (args.length == 0) {
         System.out.println("Usage: vbj Server <URL string>");
         return;
      }
      String url = args[0];
      try {
         // Initialize the ORB.
         org.omg.CORBA.ORB orb =
                        org.omg.CORBA.ORB.init(args,null);
         // get a reference to the rootPOA
         POA rootPOA = POAHelper.narrow(
                  orb.resolve_initial_references("RootPOA"));
         // Create the servant
         AccountManagerImpl managerServant =
                                    new AccountManagerImpl();
         // Decide on the ID for the servant
         byte[ ] managerId = "BankManager".getBytes();
         // Activate the servant with the ID on myPOA
         rootPOA.activate_object_with_id(managerId,
                                         managerServant);

         // Activate the POA manager
         rootPOA.the_POAManager().activate();
         // Create the object reference
         org.omg.CORBA.Object manager =
                rootPOA.servant_to_reference(managerServant);
         // Obtain the URLNaming Resolver
         Resolver resolver = ResolverHelper.narrow(
            orb.resolve_initial_references(
               "URLNamingResolver"));
         // Register the object reference (overwrite
         // if exists)
         resolver.force_register_url(url, manager);
         System.out.println(manager + " is ready.");
         // Wait for incoming requests
         orb.run();
      } catch (Exception e) {
         e.printStackTrace();
      }
```

```
    }
}
```

このコードサンプルでは，args[0]は次の形式になっています。

例

```
http://<host_name>:<http_server_port>/
                        <ior_file_path>/<ior_file_name>
```

ior_file_name は，文字列化されたオブジェクトリファレンスが格納されているユーザ指定のファイル名です。ior_file_name の接尾語は，ゲートキーパーが HTTP サーバの代わりに使用される場合は.ior である必要があります。ゲートキーパーとそのデフォルトのポート番号を使用した例は次のとおりです。

例

```
http://mars:15000/URLNaming/Bank_Manager.ior
```

26.3　URL によるオブジェクトの検索

クライアントアプリケーションは Resolver にバインドする必要はなく，コードサンプル 26-2 に示すように bind()メソッドの呼び出し時に URL を指定するだけです。バインドは URL をオブジェクト名として受け取ります。URL が無効な場合は InvalidURL 例外が発生します。bind()メソッドは透過的に locate()メソッドを呼び出します。locate()メソッドの使用例については，コードサンプル 26-3 を参照してください。

コードサンプル 26-2　URL の指定によるオブジェクトリファレンスの取得

```
// ResolverClient.java
import com.inprise.vbroker.URLNaming.*;
public class ResolverClient {
   public static void main(String[ ] args) {
      if (args.length == 0) {
         System.out.println(
            "Usage: vbj Client <URL string> [Account name]");
         return;
      }
      String url = args[0];
      try {
         // Initialize the ORB.
         org.omg.CORBA.ORB orb =
                           org.omg.CORBA.ORB.init(args,null);
         // Obtain the URLNaming Resolver
         Resolver resolver = ResolverHelper.narrow(
            orb.resolve_initial_references(
               "URLNamingResolver"));
         // Locate the object
         Bank.AccountManager manager =
         Bank.AccountManagerHelper.
                              narrow(resolver.locate(url));
         // use args[0] as the account name, or a default.
         String name = args.length > 1 ?
                                    args[1] : "Jack B. Quick";
         // Request the account manager to open a named
         // account.
         Bank.Account account = manager.open(name);
         // Get the balance of the account.
         float balance = account.balance();
         // Print out the balance.
         System.out.println("The balance in " + name +
                           "'s account is $" + balance);
      } catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

コードサンプル 26-3　Resolver.locate メソッドの使用によるオブジェクトリファレンスの取得

```
// Client.java
public class Client {
   public static void main(String[ ] args) {
      if (args.length == 0) {
         System.out.println(
            "Usage: vbj Client <URL string> [Account name ]");
         return;
      }
      String url = args[0];
      // Initialize the ORB.
      org.omg.CORBA.ORB orb =
                           org.omg.CORBA.ORB.init(args,null);
      // Locate the object
      Bank.AccountManager manager =
                  Bank.AccountManagerHelper.bind(orb, url);
      // use args[0] as the account name, or a default.
      String name = args.length > 1 ?
                                    args[1] : "Jack B. Quick";
      // Request the account manager to open a named account.
      Bank.Account account = manager.open(name);
      // Get the balance of the account.
```

```
            float balance = account.balance();
            // Print out the balance.
            System.out.println("The balance in " + name +
                               "'s account is $" + balance);
        }
    }
```

27 双方向通信

この章では，ゲートキーパーを使用しないで Borland Enterprise Server VisiBroker で双方向コネクションを設定する方法を説明します。ゲートキーパーを使用する双方向通信については，マニュアル「Borland Enterprise Server VisiBroker　ゲートキーパーガイド」を参照してください。

ただし，双方向 IIOP を有効にする前に，「27.6　セキュリティの考慮事項」を参照してください。

27.1 双方向 IIOP の使用

インターネットによって情報交換を行うほとんどのクライアントとサーバは, 一般に共同のファイアウォールによって保護されています。リクエストがクライアントだけによって開始されるシステムでは, 通常クライアントにとってファイアウォールの存在は透過的です。しかし, クライアントが非同期で情報を必要とする場合があります。すなわち，リクエストに対する応答としてではなく情報が到着する必要がある場合です。クライアント側のファイアウォールは，サーバがクライアントにコネクションを戻さないようにします。そのため，クライアントが非同期情報を受け取るためには，通常，拡張構成が必要です。

GIOP および Borland Enterprise Server VisiBroker の従来のバージョンでは，サーバがクライアントに非同期情報を送信できる唯一の方法はクライアント側のゲートキーパーを使用してサーバからのコールバックを処理する方法でした。

非同期情報をクライアントに返送する必要がある場合にクライアントに対して個別のオープンなコネクションをサーバが持つのではなく（どちらにしても，これはクライアント側のファイアウォールに拒否される），双方向 IIOP を使用すれば，サーバはクライアント起動コネクションを使用してクライアントに情報を送信します。また，CORBA の仕様はこの機能をポータブルに制御するための新しいポリシーも追加します。

双方向 IIOP によってゲートキーパーなしでコールバックが設定できるので，クライアントの配置が非常に楽になります。

27.2　双方向 VisiBroker ORB のプロパティ

次に示すの三つのプロパティが双方向をサポートします。

- vbroker.orb.enableBiDir=client|server|both|none
- vbroker.se.<se-name>.scm.<scm-name>.manager.exportBiDir=true|false
- vbroker.se.<se-name>.scm.<scm-name>.manager.importBiDir=true|false

(1)　vbroker.orb.enableBiDir プロパティ

vbroker.orb.enableBiDir プロパティは，サーバおよびクライアントの両方で使用され，双方向通信が可能になります。このプロパティによって，ユーザはコードをまったく変更しないで，既存の一方向アプリケーションを双方向アプリケーションに変更できます。vbroker.orb.enableBiDir プロパティは表 27-1 に示す値に設定できます。

表 27-1　vbroker.orb.enableBiDir プロパティの設定値

値	説明
client	すべての POA およびすべての出力コネクションで双方向 IIOP を有効にします。この設定は，both に対して BiDirectional ポリシーを設定してすべての POA を作成すること，および VisiBroker ORB レベルで both に対して BiDirectional ポリシーの変更を設定することと同じです。さらに，どの SCM でも exportBiDir プロパティが ture に設定されているかのように，生成された SCM のすべては双方向コネクションを許可します。
server	サーバが双方向のコネクションを受け付けて，それを使用できるようになります。これは，すべての SCM で importBiDir プロパティを true に設定するのと同じです。
both	Client と Server の両方にプロパティを設定します。
none	双方向 GIOP を全体的に無効にします。これはデフォルト値です。

(2)　vbroker.se.<se-name>.scm.<scm-name>.manager.exportBiDir プロパティ

vbroker.se.<se-name>.scm.<scm-name>.manager.exportBiDir プロパティは，クライアント側のプロパティです。デフォルトでは，VisiBroker ORB によって何も設定されていません。これを true に設定すると，指定のサーバエンジンで双方向コールバック POA の作成ができるようになります。

false に設定すると，指定のサーバエンジンで双方向 POA の作成ができなくなります。

(3)　vbroker.se.<se-name>.scm.<scm-name>.manager.importBiDir プロパティ

vbroker.se.<se-name>.scm.<scm-name>.manager.importBiDir プロパティは，サーバ側のプロパティです。デフォルトでは，VisiBroker ORB によって何も設定されていません。これを true に設定すると，サーバ側はクライアントにリクエストを送信するためにクライアントがすでに設定したコネクションを再利用できます。false に設定すると，このようなコネクションの再利用はできません。

(4)　注意

これらのプロパティは SCM の作成時に一度だけ評価されます。すべての場合，SCM の exportBiDir プロパティおよび importBiDir プロパティは，enableBiDir プロパティを管理します。言い換えれば，両方のプロパティに，競合する値が設定されていると，SCM 固有のプロパティが有効になります。これによって，ユーザは enableBiDir プロパティをグローバルに設定でき，特に個々の SCM で BiDir をオフにできるようになります。

27.3　サンプルについて

この機能の使用方法のサンプルは，Borland Enterprise Server VisiBroker インストールディレクトリの examples/vbe/bidir-iiop サブディレクトリにあります。すべてのサンプルは，次のような単純な株価情報のコールバックアプリケーションに基づいています。

1. クライアントは，株価情報更新を処理する CORBA オブジェクトを作成します。
2. クライアントは，この CORBA オブジェクトのオブジェクトリファレンスをサーバに送信します。
3. サーバは，株価情報を定期的に更新するためにこのコールバックオブジェクトを呼び出します。

「27.4　既存のアプリケーションで双方向 IIOP を有効にする」および「27.5　双方向 IIOP を明示的に有効にする」では，さまざまな場合の双方向 IIOP 機能について例を使用して説明します。

27.4　既存のアプリケーションで双方向 IIOP を有効にする

ユーザはソースコードをまったく修正しないで，既存の C++および Java の Borland Enterprise Server VisiBroker アプリケーションで双方向通信を有効にできます。双方向 IIOP をまったく使用しない単純なコールバックアプリケーションは，examples/vbe/bidir-iiop/basic ディレクトリに格納されています。

このアプリケーションで双方向 IIOP を有効にするには，vbroker.orb.enableBiDir プロパティを設定します。

1. スマートエージェントが実行中であることを確認します。
2. サーバを起動します。

   C++の場合（UNIX）

   ```
   prompt>Server -Dvbroker.orb.enableBiDir=server &
   ```

   Java の場合（UNIX）

   ```
   prompt>vbj -Dvbroker.orb.enableBiDir=server Server &
   ```

   C++の場合（Windows）

   ```
   prompt>start Server -Dvbroker.orb.enableBiDir=server
   ```

   Java の場合（Windows）

   ```
   prompt>start vbj -Dvbroker.orb.enableBiDir=server Server
   ```

3. クライアントを起動します。

   C++の場合

   ```
   prompt>Client -Dvbroker.orb.enableBiDir=client
   ```

   Java の場合

   ```
   prompt>vbj -Dvbroker.orb.enableBiDir=client RegularClient
   ```

ここで，既存のコールバックアプリケーションは双方向 IIOP を使用して，クライアント側のファイアウォールを越えて動作します。

27.5　双方向 IIOP を明示的に有効にする

examples/vbe/bidir-iiop/basic ディレクトリの Client は，「27.4　既存のアプリケーションで双方向 IIOP を有効にする」で記述した Client から派生します。ただし，このクライアントはプログラムによって双方向 IIOP を有効にします。

クライアントコードだけを変更する必要があります。一方向クライアントを双方向クライアントに変換するには，次に示す操作を行うだけです。

1. BiDirectional ポリシーをコールバック POA のポリシーのリストに入れます。
2. BiDirectional ポリシーを，双方向 IIOP を有効にしたいサーバを表すオブジェクトリファレンスの変更のリストに追加します。
3. クライアントで exportBiDir プロパティを true に設定します。

次に示すコードの抜粋では，双方向 IIOP をインプリメントするコードをボールド体で示します。

コードサンプル 27-1　双方向 IIOP のインプリメント（C++）

```
try {
   CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);

   // Get the manager Id
   PortableServer::ObjectId_var managerId =
      PortableServer::string_to_ObjectId("BankManager");
   PortableServer::ObjectId_var oid =
      PortableServer::string_to_ObjectId("QuoteServer");
   Quote::QuoteServer_var quoter =
      Quote::QuoteServer::_bind("/QuoteServer_poa",oid);

   //set up the callback object...first get the RootPOA
   CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
   PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);
   PortableServer::POAManager_var the_manager = rootPOA->the_POAManager();
   PortableServer::POA_var consumer_poa;

   //Set up a policy.
   CORBA::Any policy_value;
   policy_value <<= BiDirPolicy::BOTH;
   CORBA::Policy_var policy =
      orb->create_policy(BiDirPolicy::BIDIRECTIONAL_POLICY_TYPE,
         policy_value);

   CORBA::PolicyList policies;
   policies.length(1);
   policies [0] = CORBA::Policy::_duplicate(policy);

   consumer_poa = rootPOA->create_POA("QuoteConsumer_poa"
      ,the_manager,policies );

   QuoteConsumerImpl* consumer = new QuoteConsumerImpl;
   oid = PortableServer::string_to_ObjectId("consumer");
   consumer_poa->activate_object_with_id(oid, consumer);

   the_manager->activate();
   CORBA::Object_var obj = quoter->set_policy_overrides(
      policies,CORBA::ADD_OVERRIDE);
   quoter = Quote::QuoteServer::_narrow(_obj);

   obj = consumer_poa->id_to_reference(oid);
   Quote::QuoteConsumer_var quote_consumer =
      Quote::QuoteConsumer::_narrow(obj);
   quoter->registerConsumer(quote_consumer.in());

   cout << "implementation is running" << endl;
   orb->run();
}
catch(const CORBA::Exception& e){
```

```
    cout <<e <<endl;
}
```

コードサンプル 27-2　双方向 IIOP のインプリメント（Java）

```
public static void main (String[ ] args)){
      try {
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      org.omg.PortableServer.POA rootPOA =
         org.omg.PortableServer.POAHelper.narrow(
            orb.resolve_initial_references("RootPOA"));
      org.omg.CORBA.Any bidirPolicy = orb.create_any();
      bidirPolicy.insert_short(BOTH.value);
      org.omg.CORBA.Policy[ ] policies = {
         //set bidir policy
         orb.create_policy(BIDIRECTIONAL_POLICY_TYPE.value,
            bidirPolicy)
      };
      org.omg.PortableServer.POA callbackPOA =
         rootPOA.create_POA("bidir", rootPOA.the_POAManager(),
                                                    policies);
      QuoteConsumerImpl c = new QuoteConsumerImpl();
      callbackPOA.activate_object(c);
      callbackPOA.the_POAManager().activate();
      QuoteServer serv = QuoteServerHelper.bind(orb,
                  "/QuoteServer_poa", "QuoteServer".getBytes());
      serv=QuoteServerHelper.narrow(serv._set_policy_override(
            policies,org.omg.CORBA.SetOverrideType.ADD_OVERRIDE));
      serv.registerConsumer(QuoteConsumerHelper.narrow(
            callbackPOA.servant_to_reference(c)));
      System.out.println("Client:consumer registered");
      //sleeping for 60 seconds,receiving message
      try{
         Thread.currentThread().sleep(60*1000);
      }
      catch(java.lang.InterruptedException e){ }
      serv.unregisterConsumer(QuoteConsumerHelper.narrow(
            callbackPOA.servant_to_reference(c)));
      System.out.println("Client: consumer unregistered.Good bye.");
      orb.shutdown(true);
. . .
```

注

ユーザのアプリケーションを調整するためにポリシーを設定する方法については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」の「QoS インタフェースとクラス」の記述を参照してください。

クライアントコネクションは一方向または双方向のどちらでもかまいません。サーバは双方向コネクションを使用して，新しいコネクションをオープンしないでクライアントをコールバックできます。コネクションがこれ以外の場合，一方向コネクションとみなされます。

コールバックオブジェクトのホストとなる POA は，BiDirectional ポリシーを BOTH に設定することによって双方向 IIOP を有効にする必要があります。この POA は，SCM マネージャで vbroker.se.<sename>.scm.<scmname>.manager.exportBiDir プロパティを設定して双方向サポートを有効にしてある SCM で作成する必要があります。POA がこれ以外の場合，POA はクライアント起動コネクションによってサーバからリクエストを受信することはできません。

POA が BiDirectional ポリシーを指定しない場合，POA を出力コネクションで使用しないでください。この要件を満たすために，BiDirectional ポリシーが設定されていない POA は，exportBiDir プロパティが設定されている SCM が一つでもあるサーバエンジン上では作成できません。一方向 SE で POA を作成しようとすると，ServerEnginePolicy がエラーとなる InvalidPolicy 例外が発生します。

注

同じクライアントコネクションを使用している異なるオブジェクトは BiDirectional ポリシーの競合の変更を設定することがあります。しかし，一度コネクションが双方向になったら，あとでポリシーが有効になってもならなくても，常に双方向のままになります。

双方向設定に対してユーザに完全に制御が移ったら，iiop_tp SCM だけで双方向 IIOP を有効にします。

C++の場合

```
prompt>Client -Dvbroker.se.iiop_tp.scm.iiop_tp.manager.exportBiDir¥
       =true
```

Java の場合

```
prompt>vbj -Dvbroker.se.iiop_tp.scm.iiop_tp.manager.¥
       exportBiDir=true Client
```

27.6　セキュリティの考慮事項

双方向 IIOP を使用する場合，重要なセキュリティ問題が発生することがあります。ほかにセキュリティ機能を備えていない場合，悪質なクライアントが選んだホストやポートとコネクションの双方向での使用を要求することがあります。特に，クライアントは，そのホストに常駐していないセキュリティに敏感なオブジェクトのホストやポートを指定することがあります。ほかにセキュリティ機能を備えていない場合，入力コネクションを受け付けたサーバには，アイデンティティを確認したり，コネクションを起動したクライアントの保全性を確認したりする方法がありません。さらに，サーバは双方向コネクションによってアクセスできる別のオブジェクトへのアクセスを確保する可能性があります。これが，個別の双方向 SCM をコールバックオブジェクトに使用した方がよい理由です。クライアントの保全性について心配であれば，双方向 IIOP を使用しないことをお勧めします。

セキュリティの面から，VisiBroker を実行しているサーバは双方向 IIOP が明示的に設定されていなければ双方向 IIOP を使用しません。vbroker.se.<se-name>.scm.<scm-name>.manager.importBiDir プロパティは，双方向性の制御を SCM ごとにユーザに与えます。例えば，クライアントを認証するために SSL を使用するサーバエンジンだけで双方向 IIOP を有効にすること，また，双方向の使用のためにそのほかの，通常の IIOP コネクションを有効にしないことを選択できます（この方法については，「27.2　双方向 VisiBroker ORB のプロパティ」を参照してください）。さらにクライアント側で，クライアントのファイアウォールの外側でコールバックを行うサーバだけで双方向コネクションを有効にします。クライアントとサーバ間で高度なセキュリティを設定するには，相互認証（クライアントとサーバの両方で vbroker.security.peerAuthenticationMode を REQUIRE_AND_TRUST に設定）で SSL を使用しなければなりません。

28 VisiBroker コードの移行

この章では，VisiBroker コードを VisiBroker の以前のバージョンから Borland Enterprise Server VisiBroker へ移行する方法について説明します。Java コードを VisiBroker 3.x から VisiBroker 5.x に移行する方法は二つあります。移行プロセスの重要な部分を自動化しようとするコマンドラインユーティリティであるマイグレータと，手動による移行です。できるだけ VisiBroker 3.x コードは手動で VisiBroker 5.x に移行することをお勧めします。VisiBroker 3.x をアップグレードするより，VisiBroker 5.x の固有の呼び出しを使用する方が多くの利点があります。

28.1　BOA の POA への手動による移行

クラス名は VisiBroker の以前のバージョンから変更されています。ソースファイルを更新して，最新のクラス名を指定してください。表 28-1 および表 28-2 に，クラス名の変更例について示します。

表 28–1　クラス名の変更（C++）

旧クラス名	新クラス名
_sk_Account	POA_Account
_sk_AccountManager	POA_AccountManager
_tie_Account	POA_Account_tie
_tie_AccountManager	POA_AccountManager_tie

表 28–2　クラス名の変更（Java）

旧クラス名	新クラス名
_st_Account	_AccountStub
_st_AccountManager	_AccountManagerStub
_AccountImplBase	AccountPOA
_AccountManagerImplBase	AccountManagerPOA
_tie_Account	AccountPOATie
_tie_AccountManager	AccountManagerPOATie

28.1.1　サンプルについて

Borland Enterprise Server VisiBroker をインストールしたディレクトリの examples/vbe/boa/boa2poa に，BOA をそれに対応する POA コードに更新する場合のサンプルが入っています。

このサンプルでは，Server.C（C++），Server.java（Java）の BOA コードは次の手順によって POA に更新されています。

- BOA を初期化する代わりにルート POA のリファレンスを取得する
- BOA 特性を模造するために適切な POA ポリシーを設定する
- サーバントを定義する（POA には BOA とは異なるサーバントの定義があります）
- POA マネージャを起動する（BOA と異なる手順）
- boa->impl_is_ready()メソッド（C++）または boa.impl_is_ready()メソッド（Java）の代わりに orb->run()メソッド（C++）または orb.run()メソッド（Java）によって入力リクエストを待つ

(1)　ルート POA のリファレンスの取得

C++の場合

BOA を使用する際に，BOA のリファレンスは orb->BOA_init()メソッドによって取得しました。

しかし，POA ではルート POA のリファレンスを取得します。これは，orb->resolve_initial_references("RootPOA")メソッドを使用して行います。resolve_initial_references は，CORBA::object 型の値を返します。そしてこの値を目的のタイプにナロウします。

コードサンプル 28-1　rootPOA のリファレンスの取得（C++）

```
CORBA::object_var obj = resolve_initial_references(”RootPOA”);
PortableServer::POA_var rootPOA =
      PortableServer::POA::_narrow(obj);
```

Java の場合

BOA を使用する際に，BOA のリファレンスは orb.BOA_init()メソッドによって取得しました。

しかし，POA ではルート POA のリファレンスを取得します。これは，orb.resolve_initial_references("RootPOA")メソッドを使用して行います。resolve_initial_references は，CORBA.object 型の値を返します。そしてこの値を目的のタイプにナロウします。

コードサンプル 28-2　rootPOA のリファレンスの取得（Java）

```
POA rootPOA = POAHelper.narrow(orb.resolve_initial_references(”RootPOA”));
```

(2)　POA ポリシーの設定

POA の特性は，その POA のポリシーセットによって定義されます。各 POA には，独自のポリシーのセットがあります。POA は，ほかの POA からのポリシーを継承できません。

ここに示すサンプルでは，パーシステントオブジェクトが使用されています。BOA では，パーシステントオブジェクトには特定のインスタンス名があり，スマートエージェントに登録されるものです。

一つの BOA はパーシステントオブジェクトとトランジェントオブジェクトの両方をサポートできます。POA では，パーシステントオブジェクトはパーシステントオブジェクトを生成するプロセスを残したものです。一つの POA はパーシステントオブジェクトもトランジェントオブジェクトもサポートしますが，両方をサポートすることはできません。サポートされるオブジェクトタイプは，POA ポリシーによって設定されます。ルート POA はトランジェントオブジェクトを（デフォルトで）サポートするので，パーシステントオブジェクトをサポートするために新しい POA を作成する必要があります。

注

POA のポリシーをいったん生成したら変更することはできません。

パーシステントオブジェクトをサポートするには，ライフスパンポリシーを PERSISTENT に設定します。これは，C++と Java の両方に適用されます。

C++または Java の場合

ここに示すサンプルでは，バインドサポートポリシー（Borland Enterprise Server VisiBroker 固有ポリシー）を BY_INSTANCE に設定します。このポリシーは，すべてのアクティブなオブジェクトを POA（デフォルト）だけでなくスマートエージェントに登録しています。

適切なポリシーを設定したら，新しい POA が create_POA()メソッドで生成できます。

コードサンプル 28-3　POA ポリシーの設定（C++）

```
CORBA::PolicyList policies;
   policies.length(1);

   policies [(CORBA::ULong)0] =
         rootPOA->create_lifespan_policy(
         PortableServer::PERSISTENT);

   // Create myPOA with the right policies
   PortableServer::POAManager_var mgr =
         rootPOA->the_POAManager();
```

```
PortableServer::POA_var myPOA =
      rootPOA->create_POA(
      "bank_agent_poa", mgr, policies );
```

コードサンプル 28-4　POA ポリシーの設定（Java）

```
org.omg.CORBA.Any any = orb.create_any();
   BindSupportPolicyValueHelper.insert(any,
                        BindSupportPolicyValue.BY_INSTANCE);
   org.omg.CORBA.Policy bsPolicy =
     orb.create_policy(com.inprise.vbroker.PortableServerExt.
         BIND_SUPPORT_POLICY_TYPE.value, any);

   org.omg.CORBA.Policy[ ] policies ={
      rootPOA.create_lifespan_policy(LifespanPolicyValue.
         PERSISTENT), bsPolicy};
      // Create myPOA with the right policies
      POA myPOA = rootPOA.create_POA("bank_agent_poa",
                        rootPOA.the_POAManager(), policies);
```

(3)　サーバントの定義

BOA では，サーバントが CORBA オブジェクトです。ここに示すサンプルでは，アカウントマネージャオブジェクトが生成されてから，obj_is_ready()メソッドでエクスポートされます。

POA では，サーバントはプログラミングオブジェクトで，これは abstract オブジェクトのインプリメンテーションを提供します。サーバントは CORBA オブジェクトではありません。POA シナリオ下では，サーバントが生成されてから固有 ID で起動されます。この ID を使ってオブジェクトリファレンスを取得できます。

コードサンプル 28-5　サーバントの定義および起動（C++）

```
// Create the servant
   AccountManagerImpl *managerServant =
         new AccountManagerImpl;
// Decide on the ID for the servant
   PortableServer::ObjectId_var managerId =
      PortableServer::string_to_ObjectId("BankManager");
// Activate the servant with the ID on myPOA
   myPOA->activate_object_with_id(managerId,managerServant);
```

コードサンプル 28-6　サーバントの定義および起動（Java）

```
// Create the servant
   AccountManagerImpl managerServant =
                                    new AccountManagerImpl();
// Decide on the ID for the servant
   byte[ ] managerId = "BankManager".getBytes();
// Activate the servant with the ID on myPOA
   myPOA.activate_object_with_id(managerId, managerServant);
```

(4)　POA マネージャの起動

POA マネージャは POA がリクエストをどのように処理するかを制御するオブジェクトです。デフォルトで，POA マネージャは待機状態で生成されます。この状態では，すべてのリクエストは待機キューにルーティングされ，処理はされません。リクエストをディスパッチするには，POA に対応する POA マネージャを待機状態からアクティブ状態に変更する必要があります。

これは，POA で必要な新しい手順です。BOA と同じ手順ではありません。

コードサンプル 28-7　POA マネージャの起動

C++の場合

```
rootPOA->the_POAManager()->activate();
```

Java の場合

```
rootPOA.the_POAManager().activate();
```

(5)　入力リクエスト待ち

BOA では，クライアントからのリクエストを待つために impl_is_ready()メソッドが呼び出されます。

POA では，orb->run()（C++）または orb.run()（Java）を使用します。

コードサンプル 28-8　入力リクエスト待ち

C++の場合

```
orb->run();
```

Java の場合

```
orb.run();
```

(6)　ほかのファイルについて（C++および Java）

AccountImpl および AccountManagerImpl クラスの変更事項は，ほぼ新しいクラスの指定を行うことだけです。

28.1.2　BOA 型の POA ポリシーへのマッピング

表 28-3 は，BOA 動作を模造するためにユーザの POA ポリシーを設定する方法を示します。

表 28-3　BOA 型の POA ポリシーへのマッピング

ポリシー	トランジェント BOA	パーシステント BOA
TPOOL	TPOOL ディスパッチャによるサーバエンジンポリシー	TPOOL ディスパッチャによるサーバエンジンポリシー
		PERSISTENT に設定された LifeCycle プロパティ
	TRANSIENT に設定された LifeCycle プロパティ	USER_ID に設定された IDAssignment ポリシー
		BY_INSTANCE に設定された BindSupport ポリシー
TSESSION	TSESSION ディスパッチャによるサーバエンジンポリシー	TSESSION ディスパッチャによるサーバエンジンポリシー
		PERSISTENT に設定された LifeCycle プロパティ
	TRANSIENT に設定された LifeCycle プロパティ	USER_ID に設定された IDAssignment ポリシー
		BY_INSTANCE に設定された BindSupport ポリシー
サービス起動オブジェクト	TRANSIENT に設定された LifeCycle プロパティ	PERSISTENT に設定された LifeCycle プロパティ
		USE_SERVANT_MANAGER に対するリクエスト処理ポリシー
	USE_SERVANT_MANAGER に対するリクエスト処理ポリシー	IMPLICIT_ACTIVATION に設定された暗黙起動ポリシー
	IMPLICIT_ACTIVATION に設定された暗黙起動ポリシー	

28.2　新しいパッケージ名への移行（Java）

表 28-4 に VisiBroker 3.x パッケージ名プリフィクスの最新バージョンへのマッピング方法を示します。

表 28–4　VisiBroker 3.x パッケージ名プリフィクスのマッピング

VisiBroker 3.x パッケージ名	Borland Enterprise Server VisiBroker 5.x パッケージ名
com.visigenic.vbroker	com.inprise.vbroker
com.visigenic.vbroker.services.CosEvent	com.inprise.vbroker.CosEvent
com.visigenic.vbroker.services.CosNaming	com.inprise.vbroker.naming

28.3　新しい API 呼び出しへの移行（Java）

表 28-5 に VisiBroker 3.x API 呼び出しの最新バージョンへのマッピング方法を示します。

表 28–5　VisiBroker 3.x API 呼び出しのマッピング

VisiBroker 3.x API 呼び出し	Borland Enterprise Server VisiBroker API 呼び出し
com.visigenic.vbroker.services.CosEvent.EventLibrary のインスタンスでの create_channel(boa, name, debug, maxQueueLength)	com.inprise.vbroker.CosEvent.EventLibrary のインスタンスでの create_channel(name, debug, maxQueueLength)
com.visigenic.vbroker.services.CosEvent.EventLibrary のインスタンスでの create_channel(boa, name, debug)	com.inprise.vbroker.CosEvent.EventLibrary のインスタンスでの create_channel(name, debug)
com.visigenic.vbroker.services.CosEvent.EventLibrary のインスタンスでの create_channel(boa, name)	com.inprise.vbroker.CosEvent.EventLibrary のインスタンスでの create_channel(name)
com.visigenic.vbroker.services.CosEvent.EventLibrary のインスタンスでの create_channel(boa)	com.inprise.vbroker.CosEvent.EventLibrary のインスタンスでの create_channel()

新しいパッケージ名，クラス，および API 呼び出しの詳細については，マニュアル「Borland Enterprise Server VisiBroker　プログラマーズリファレンス」を参照してください。

28.4　インタセプタの移行

インタセプタを Borland Enterprise Server VisiBroker に移行する際は，ポータブルインタセプタを使用してください。ポータブルインタセプタの動作方法については，「19.　ポータブルインタセプタの使用」を参照してください。

28.5　イベントループの統合の移行（C++）

VisiBroker 3.x の機能によって，ユーザはオブジェクトのイベントポーリングをネットワークやウィンドウコンポーネントのイベントループに組み込めます。VisiBroker 3.x の機能を次に示します。

- シングルスレッドライブラリによって，サードパーティライブラリに応答する CORBA アプリケーションを構築できます。このようなアプリケーションは再入可能でないシステムライブラリで構築する必要があります。
- Windows でシングルスレッドサーバを構築する WDispatcher クラス。
  このクラスによって，VisiBroker ORB イベントは Windows メッセージイベントと統合できます。
- X Window System でシングルスレッドサーバを構築する XDispatcher クラス。
  このクラスによって，VisiBroker ORB イベントは X Window System の XtMainLoop に直接統合できます。
- Dispatcher クラスによって，VisiBroker ORB イベントと別の環境との統合が可能になります。これは，Dispatcher からカスタムサブクラスを派生させ，すべてのメソッドのインプリメンテーションを提供することによって行うことができます。
  また，Dispatcher クラスを使用して，VisiBroker ORB イベントを直接監視し，特定のファイルのディスクリプタのイベントを処理し，イベントタイマを設定できます。

ただし，再入可能な VisiBroker ORB ライブラリでは，すべての VisiBroker ORB イベントは VisiBroker ORB スレッドによって処理され，通常の状況下では別のイベント処理システムと統合する必要はありません。VisiBroker 5.x では，このリリースにはシングルスレッド VisiBroker ORB ライブラリは組み込まれていません。このため，上記の機能は VisiBroker 5.x 以降のバージョンではサポートされていません。アプリケーションが VisiBroker 3.x の機能を使用するような場合，この機能を除去するためにコーディングを修正する必要があります。以降でこのような変更をするためのガイドラインを説明します。

28.5.1　シングルスレッド VisiBroker ORB の移行

ここでは，シングルスレッド VisiBroker ORB を移行する際のガイドラインを説明します。

- 使用するすべてのインプリメンテーションメソッドがスレッドセーフであること，動作に応じてアプリケーションで SINGLE_THREAD_MODEL，または MAIN_THREAD_MODEL の POAThreadPolicy を使用することを確認します。場合によっては，このようなポリシーを使用している場合でも，インプリメンテーションコードの重要な部分を保護する必要があります。このような場合，アプリケーションの重要部分を統合するグローバルミューテックスを使用して，同期を取れます。
- アプリケーション（場合によってはメソッド）に main()メソッドを戻さないようにする機能がほかにない場合は，ORB run()メソッドを使用します。
  ただし，run()メソッドを呼び出す必要はありません。すべての VisiBroker ORB スレッドは ORB_init()メソッドおよび POA マネージャ起動メソッドによって自動的に作成されます。

28.5.2　XDispatcher クラスまたは WDispatcher クラスによる移行

XDispatcher クラスと WDispatcher クラスは VisiBroker ORB ランタイムおよび X Window System（または Windows）イベントループと透過的に動作します。これは，ORB_init に対する最初の呼び出し前に，XDispatcher または WDispatcher のインスタンス作成時に動作します。このため，「28.5.1　シングルスレッド VisiBroker ORB の移行」で説明したように，アプリケーションがスレッドセーフとなったら，XDispatcher または WDispatcher のリファレンスを単純に削除できます。コードサンプル 28-9 および 28-10 に WDispatcher 変換の例を示します。

コードサンプル 28-9 移行前の WDispatcher

```
//. . .
hwnd = CreateWindow(szAppName, "AccountServer",
                        WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                        CW_USEDEFAULT,200,200,NULL,
                        NULL,hInstance,NULL);

//Create a WDispatcher instance before calling ORB_init
WDispatcher *winDispatcher = new WDispatcher(hwnd);

//Initialize the ORB
CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);

//Initialize the BOA
CORBA::BOA_var orb = orb->BOA_init(__argc, __argv);

//Create the servant
AccountImpl server("BankManager");

//Activate the servant on the BOA
boa->obj_is_ready(&server);

ShowWindow(hwnd,nCmdShow);
UpdateWindow(hwnd);

//Enter message loop
while(GetMessage(&msy, NULL, 0, 0)) {
        TranslateMessage(&msg); DispatchMessage(&msg);
    }
return msg.wParam;

//. . .
```

コードサンプル 28-10 移行後の WDispatcher

```
//. . .
hwnd = CreateWindow(szAppName, "AccountServer",
                        WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                        CW_USEDEFAULT,200,200,NULL,
                        NULL,hInstance,NULL);
//Initialize the ORB
CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);

//Initialize the POA
CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);
CORBA::PolicyList policies;
policies.length(1);
policies [(CORBA::ULong)0 ] = rootPOA->create_lifespan_policy(
                                        PortableServer::PERSISTENT);
PortableServer::POAManager_var poa_mgr =rootPOA->the_POAManager();

PortableServer::POA_var myPOA =rootPOA->create_POA("bank_agent_poa",
                                                    poa_mgr,policies);


//Create the servant
AccountManagerImpl managerServant;

//Activate the servant on the POA
PortableServer::ObjectId_var managerId =
                  PortableServer::string_to_ObjectId("BankManager");

myPOA->activate_object_with_id(managerId, &managerServant);

ShowWindow(hwnd, nCmdShow);
UpdateWindow(hwnd);

//Enter message loop
while(GetMessage(&msy,NULL,0,0)){
                        TranslateMessage(&msg);
                        DispatchMessage(&msg);
}
```

```
return msg.wParam;
// . . .
```

WDispatcher または XDispatcher を削除する場合, POA を使用可能にするためにアプリケーションを変換するための追加手順は必要ありません。

29 オブジェクトアクティベータの使用

この章では，VisiBroker オブジェクトアクティベータの使用方法について説明します。

提供する Borland Enterprise Server VisiBroker では，POA は VisiBroker 3.x で BOA が提供していた機能をサポートします。下位互換性を保証するために, この章で説明するオブジェクトアクティベータをユーザのコードで使用することもできます。BOA アクティベータを提供する Borland Enterprise Server VisiBroker で使用する方法の詳細については，「28. VisiBroker コードの移行」を参照してください。

29.1　オブジェクト活性化の遅延

サーバが多くのオブジェクトに対してインプリメンテーションを提供する必要がある場合に単一の Activator でサービス活性化を使用して，複数のオブジェクトインプリメンテーションの起動を遅延できます。

29.2　アクティベータインタフェース

ユーザは，独自のインタフェースを Activator クラス（C++）または Activator インタフェース（Java）から派生させることができます。これによってユーザは VisiBroker ORB が AccountImpl オブジェクト（C++）または DBObjectImpl オブジェクト（Java）で使用する純仮想（C++）activate および deactivate メソッドをインプリメントできます。次に，ユーザは BOA がオブジェクトに対する要求を受け取るまで，AccountImpl オブジェクトの実体化を遅延させることができます。また，BOA がオブジェクトを非活性化する場合に，クリーンアップ処理を提供できるようになります。

コードサンプル 29-1 に Activator クラス（C++），コードサンプル 29-2 に Activator インタフェース（Java）を示します。これは，VisiBroker ORB オブジェクトを活性化，非活性化するために BOA が呼び出すメソッドを提供します。

コードサンプル 29-1　Activator クラス（C++）

```
class Activator {
   public:
      virtual CORBA::Object_ptr activate(
            extension::ImplementationDef impl) = 0;
      virtual void deactivate(
            Object_ptr, extension::ImplementationDef_ptr impl)
            = 0;
};
```

コードサンプル 29-2　Activator インタフェース（Java）

```
package com.inprise.vbroker.extension;
public interface Activator {
   public abstract org.omg.CORBA.Object
                              activate(ImplementationDef impl);
   public abstract void deactivate(org.omg.CORBA.Object obj,
                                   ImplementationDef impl);
}
```

コードサンプル 29-3 に AccountImpl インタフェースに対して Activator を生成する方法を示します。

コードサンプル 29-3　activate および deactivate メソッドをインプリメントして DBActivator クラス（C++）を派生

```
class extension {
   . . .
   class AccountImplActivator : public extension::Activator {
      public:
         virtual CORBA::Object_ptr activate(
               CORBA::ImplementationDef_ptr impl);
         virtual void deactivate(CORBA::Object_ptr,
               CORBA::ImplementationDef_ptr impl);
   };
   CORBA::Object_ptr AccountImplActivator::activate(
         CORBA::ImplementationDef_ptr impl) {
      // When the BOA needs to activate us,
      // instantiate the AccountImpl object.
      extension::ActivationImplDef* actImplDef =
               extension::ActivationImplDef::_downcast(impl);
      CORBA::Object_var obj =
               new AccountImpl(actImplDef->object_name());
      return CORBA::_duplicate(obj);
   }
   void AccountImplActivator::deactivate(CORBA::Object_ptr obj,
         CORBA::ImplementationDef_ptr impl) {
      // When the BOA deactivates us, release the Account object.
      obj->_release;
   }
}
```

コードサンプル 29-4 に DBObjectImpl インタフェースに対して Activator を生成する方法を示します。

コードサンプル 29-4　activate および deactivate メソッドをインプリメントして DBActivator インタフェース（Java）を派生

```
// Server.java
import com.inprise.vbroker.extension.*;
. . .
class DBActivator implements Activator {
   private static int _count;
   private com.inprise.vbroker.CORBA.BOA _boa;

   public DBActivator(com.inprise.vbroker.CORBA.BOA boa) {
      _boa = boa;
   }
   public org.omg.CORBA.Object activate(
       com.inprise.vbroker.extension.ImplementationDef impl) {
      System.out.println(
          "Activator called " + ++_count + " times");
      byte[ ] ref_data = ((ActivationImplDef) impl) .id();
      DBObjectImpl obj =
                   new DBObjectImpl(new String(ref_data));
      _boa.obj_is_ready(obj);
      return obj;
   }
   public void deactivate(org.omg.CORBA.Object obj,
      com.inprise.vbroker.extension.ImplementationDef impl) {
    // nothing to do here...
   }
}
. . .
```

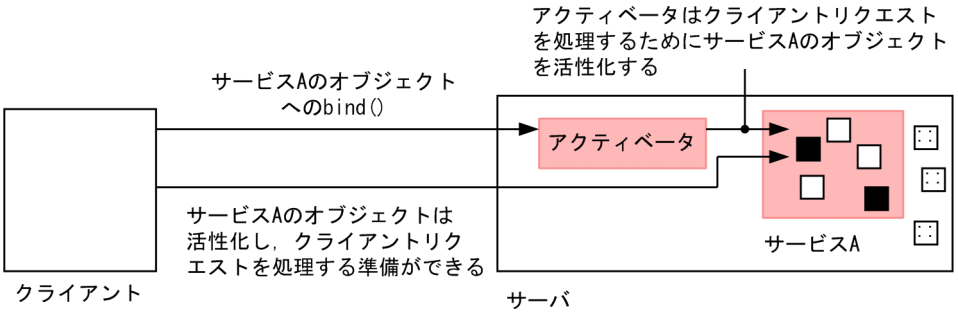
29.3　サービス活性化のアプローチ方法

サーバは，多数のオブジェクト（通常，数千個のオブジェクト。数百万個のオブジェクトの場合もある）に対してインプリメンテーションを提供する必要があります。任意の一時点で活性化する必要のあるインプリメンテーションの数が少ない場合，サービス活性化を使用できます。サーバはこれらの補助オブジェクトのどれかが必要になると通知を受ける単一の Activator を提供できます。サーバはこれらのオブジェクトが使用されていない場合に非活性化することもできます。

例えば，状態がデータベースに格納されるオブジェクトインプリメンテーションをロードするサーバに対してサービス活性化を使用すると仮定します。Activator には所定の型または論理区分のオブジェクトをすべてロードする責任があります。VisiBroker ORB リクエストがこれらのオブジェクトのリファレンスで発行されると，Activator は通知を受け，データベースからロードされる状態を持つ，新しいインプリメンテーションを生成します。Activator はオブジェクトがメモリに存在しなくなったと判断し，オブジェクトが変更されている場合，そのオブジェクトの状態をデータベースに書き込み，インプリメンテーションを解放します。

サービスの活性化の遅延プロセスを図 29-1 に示します。

図 29-1　サービスの活性化の遅延プロセス



（凡例）　■：活性化されたオブジェクト

29.3.1　サービスアクティベータを使用したオブジェクト活性化の遅延

サービスを構成するオブジェクトが生成済みであるということを前提にして，サービス活性化を使用するサーバをインプリメントするには，次の手順に従ってください。

1. Activator によって活性化され，非活性化されるすべてのオブジェクトを記述するサービス名を定義します。
2. パーシステントオブジェクトではなく，サービスオブジェクトになっているインタフェースのインプリメンテーションを提供します。これはオブジェクトが自身をサービスの活性化可能部として構成する場合に行われます。
3. 要求に応じて，オブジェクトインプリメンテーションを生成する Activator をインプリメントします。インプリメンテーション内では，extension::Activator から Activator インタフェースを派生させ，activate メソッドと deactivate メソッドを変更します。
4. サービス名と Activator インタフェースを BOA に登録します。

29.3.2　サービスの遅延オブジェクト活性化のサンプル

ここでは，サービス活性化の odb のサンプルについて説明します。サンプルは Borland Enterprise Server VisiBroker をインストールしたディレクトリの examples/vbe/boa/odb に入っています。このディレクトリには，表 29-1 のファイルも含まれます。

表 29–1　サービス活性化用 odb のサンプルファイル

ファイル名	説明
odb.idl	DB インタフェースと DBObject インタフェースの IDL
Server.C（C++） Server.java（Java）	サービスアクティベータを使用してオブジェクトを生成し，オブジェクトの IOR を返し，オブジェクトを非活性化します。
Creator.C（C++） Creator.java（Java）	100 個のオブジェクトを生成するために DB インタフェースを呼び出し，結果として生じた文字列化オブジェクトリファレンスをファイル（objref.out）に格納します。
Client.C（C++） Client.java（Java）	オブジェクトの文字列化オブジェクトリファレンスをファイルから読み出して，それらに対する呼び出しを行い，サーバ内のアクティベータにオブジェクトを生成させます。
Makefile	make または nmake（Windows の場合）が odb サブディレクトリ内で呼び出されたときに，次に示すクライアントプログラムとサーバプログラムを生成します。 • Server.exe（C++）または Server（Java） • Creator.exe（C++）または Creator（Java） • Client.exe（C++）または Client（Java）

odb のサンプルは，一つのサービスから任意の数のオブジェクトがどのように作成できるかを示します。サービス単体は，IOR の一部として格納された各オブジェクトのリファレンスデータとともに，各オブジェクトではなく BOA に登録されます。これによって，オブジェクトキーをオブジェクトリファレンスの一部として格納できるので，OODB（オブジェクト指向データベース）統合を簡易化できます。未生成のオブジェクトをクライアントが呼び出す場合，BOA はユーザが定義した Activator を呼び出します。すると，アプリケーションは適切なオブジェクトをパーシステント記憶領域からロードできます。

このサンプルでは，「DBService」という名前のサービスに対してオブジェクトを活性化し，非活性化する責任のある Activator が生成されます。この Activator が生成するオブジェクトのリファレンスには，VisiBroker ORB が DBService サービスの Activator を再検索し，Activator が要求に応じてこれらのオブジェクトを再生成するために十分な情報が含まれています。

DBService サービスは DBObject インタフェースをインプリメントするオブジェクトに対して責任があります。インタフェース（odb.idl に含まれる）はこれらのオブジェクトの手動生成をできるようにするために提供されます。

(1)　odb.idl インタフェース

odb.idl インタフェースは DBObject odb インタフェースをインプリメントするオブジェクトの手動生成をできるようにします。

IDL サンプル 29-1　odb.idl インタフェース

```
interface DBObject {
   string get_name();
};

typedef sequence<DBObject> DBObjectSequence;
```

```
interface DB {
   DBObject create_object(in string name);
};
```

DBObject インタフェースは DB インタフェースが生成したオブジェクトを表し，サービスオブジェクトとして取り扱えます。

DBObjectSequence は DBObject のシーケンスです。サーバはこのシーケンスを使って，現在活性化しているオブジェクトを把握します。

DB インタフェースは create_object オペレーションを使って一つ以上の DBObject を生成します。DB インタフェースが生成したオブジェクト群は，サービスとして一つにまとめることができます。

(2)　サービス活性化オブジェクトのインプリメント（C++）

idl2cpp コンパイラは，boa/odb/odb.idl からスケルトンクラス_sk_DBObject 用に 2 種類のコンストラクタを生成します。最初のコンストラクタは手動実体化オブジェクトで使用します。二つ目のコンストラクタはオブジェクトをサービスの一部にします。コードサンプル 29-5 に示すように，DBObject のインプリメンテーションは，手動実体化オブジェクトで一般に使用される object_name コンストラクタではなく，サービスコンストラクタを使用してベースの_sk_DBObject メソッドを構築します。この種のコンストラクタを起動して，DBObject はそれ自身を DBService というサービスの一部として構築します。

コードサンプル 29-5　サービス活性化オブジェクトのインプリメント例

```
class DBObjectImpl: public _sk_DBObject {
   private:
      CORBA::String_var_name;
   public:
      DBObjectImpl(const char *nm,
            const CORBA::ReferenceData& data)
      : _sk_DBObject("DBService", data), _name(nm) {}
   . . .
}
```

注

ベースのコンストラクタは，不明瞭な CORBA::ReferenceData 値と同様にサービス名を必要とします。Activator は，クライアントリクエストによって起動しなければならない場合に，これらのパラメタを使用して該当するオブジェクトを一意に識別します。このサンプルで複数のインスタンスを区別するための参照データは，0 から 99 までの数字で構成されます。

(3)　サービスアクティベータのインプリメント

通常，オブジェクトをインプリメントしている C++または Java クラスをサーバが実体化し，次に BOA::obj_is_ready（C++），BOA::impl_is_ready（C++）または obj_is_ready（Java），impl_is_ready（Java）の順で呼び出すと，オブジェクトが活性化されます。オブジェクトの活性化を遅延させるには，BOA がオブジェクト活性化中に呼び出す activate メソッドの制御を得る必要があります。この制御を得るには，extension::Activator（C++）または com.inprise.vbroker.extension.Activator（Java）から新たなクラスを派生させ，activate メソッドを変更し，変更した activate メソッドを使ってオブジェクト固有の C++クラスまたは Java クラスを実体化します。

odb のサンプルでは，DBActivator クラスが extension::Activator（C++）または com.inprise.vbroker.extension.Activator（Java）から派生し，activate メソッドと deactivate メソッドを変更します。DBObject は activate メソッド内に構築されます。

コードサンプル 29-6　activate メソッドと deactivate メソッドを変更する例（C++）

```
class DBActivator: public extension::Activator {
      virtual CORBA::Object_ptr activate(
            CORBA::ImplementationDef_ptr impl);
      virtual void deactivate(CORBA::Object_ptr,
```

```
            CORBA::ImplementationDef_ptr impl );
   public:
      DBActivator(CORBA::BOA_ptr boa) : _boa(boa) {}
   private:
      CORBA::BOA_ptr _boa;
};
```

コードサンプル 29-7　activate メソッドと deactivate メソッドを変更する例（Java）

```
// Server.java
class DBActivator implements Activator {
   private static int _count;
   private com.inprise.vbroker.CORBA.BOA _boa;
   public DBActivator(com.inprise.vbroker.CORBA.BOA boa) {
      _boa = boa;
   }
   public org.omg.CORBA.Object activate(
         com.inprise.vbroker.extension.ImplementationDef impl) {
      System.out.printIn("Activator called " + ++_count + " times");
      byte[ ] ref_data = ((ActivationImplDef) impl).id();
      DBObjectImpl obj = new DBObjectImpl(new String(ref_data));
      _boa.obj_is_ready(obj);
      return obj;
   }
   public void deactivate(org.omg.CORBA.Object obj, ImplementationDef impl) {
      // nothing to do here...
   }
}
```

コードサンプル 29-8 に示すように，DBActivator クラスはその CORBA::ReferenceData パラメタに基づいてオブジェクトを生成します（C++）。コードサンプル 29-9 では，DBActivator クラスが ReferenceData パラメタに基づいてオブジェクトを生成する方法を示します（Java）。BOA は Activator の責任の下でオブジェクトを求めるクライアントリクエストを受信すると，その Activator に対して activate メソッドを起動します。このメソッドを呼び出すとき，BOA は Activator に ImplementationDef パラメタを引き渡すことによって，活性化されたオブジェクトインプリメンテーションを一意に識別します。このパラメタから，インプリメンテーションはリクエストされたオブジェクトの一意の識別子である CORBA::ReferenceData（C++）または ReferenceData（Java）を取得できます。

コードサンプル 29-8　サービスアクティベータをインプリメントする例（C++）

```
CORBA::Object_ptr DBActivator::activate(
      CORBA::ImplementationDef_ptr impl) {
   extension::ActivationImplDef* actImplDef =
         extension::ActivationImplDef::_downcast(impl);
   CORBA::ReferenceData_var id(actImplDef->id());
   cout << "Activate called for object=[" << (char*) id->data()
         << "]" << endl;
   DBObjectImpl *obj = new DBObjectImpl((char *)id->data(), id);
   _impls.length(_impls.length() +1);
   _impls[_impls.length()-1] = DBObject::_duplicate(obj);
   _boa->obj_is_ready(obj);
   return obj;
}
```

コードサンプル 29-9　サービスアクティベータをインプリメントする例（Java）

```
public org.omg.CORBA.Object activate(ImplementationDef impl) {
   System.out.println("Activator called " + ++_count + " times");
   byte[ ] ref_data = ((ActivationImplDef) impl) .id();
   DBObjectImpl obj = new DBObjectImpl(new String(ref_data));
   _boa.obj_is_ready(obj);
   return obj;
}
```

(4)　サービスアクティベータの実体化

コードサンプル 29-10 に示すように，DBActivator サービスアクティベータは，メインサーバプログラムの BOA::impl_is_ready 呼び出しを使って生成され，BOA に登録されます（C++）。コードサンプル

29-11 は，メインサーバプログラムの impl_is_ready 呼び出しによって DBActivator サービスアクティベータを作成し，登録する Java の例を示します。

DBActivator サービスアクティベータは，DBService サービスに属するすべてのオブジェクトに対して責任を持ちます。DBService サービスのオブジェクトを求めるリクエストはすべて DBActivator サービスアクティベータを通じて指示されます。このサービスアクティベータによって活性化されたオブジェクトはすべてそれらが DBService サービスに属していることを VisiBroker ORB に通知するリファレンスを持っています。

コードサンプル 29-10　サービスアクティベータを実体化する例（C++）

```
int main(int argc, char **argv) {
   CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
   CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);
   MyDB db("Database Manager");
   boa->obj_is_ready(&db);
   DBObjectImplReaper reaper;
   reaper.start();
   cout << "Server is ready to receive requests" << endl;
   boa->impl_is_ready("DBService", new DBActivator(boa));
   return(0);
}
```

コードサンプル 29-11　サービスアクティベータを実体化する例（Java）

```
public static void main(String[ ] args) {
   org.omg.CORBA.ORB orb = ORB.init(args, null);
   com.inprise.vbroker.CORBA.BOA boa
            = ((com.inprise.vbroker.orb.ORB )orb).BOA_init();
   DB db = new DBImpl("Database Manager");
   boa.obj_is_ready(db);
   boa.impl_is_ready("DBService", new DBActivator(boa));
}
```

BOA::impl_is_ready（C++）または impl_is_ready（Java）の呼び出しは，通常の BOA::impl_is_ready（C++）または impl_is_ready（Java）の呼び出しの変形であり，次のように引数を二つ持つことに注意してください。

- サービス名
- BOA がサービスに属するオブジェクトを活性化するために使用する Activator インタフェースのインスタンス

(5)　サービスアクティベータを使用したオブジェクトの活性化（C++）

オブジェクトが構築されるたびに，BOA::obj_is_ready を DBActivator::activate 内で明示的に呼び出す必要があります。サーバプログラムには BOA::obj_is_ready に対して二つの呼び出しがあります。第一の呼び出しはサーバがサービスオブジェクトを生成し，生成元プログラムに IOR を返したときに発生します。

コードサンプル 29-12　BOA::obj_is_ready へのサーバの第一の呼び出し（C++）

```
DBObject_ptr create_object(const char *name) {
   char ref_data[100];
   memset(ref_data,'¥0',100);
   sprintf(ref_data, "%s", name);
   CORBA::ReferenceData id(100, 100, (CORBA::Octet *)ref_data);
   DBObjectImpl *obj = new DBObjectImpl(name, id);
   _boa()->obj_is_ready(obj);
   _impls.length(_impls.length() + 1);
   _impls[_impls.length()-1] = DBObject::_duplicate(obj);
   return obj;
}
```

BOA::obj_is_ready への第二の呼び出しは，DBActivator::activate 内にあり，これは明示的に呼び出す必要があります。コンテキスト内でのこの第二の呼び出しについては，コードサンプル 29-8 を参照してください。

(6)　サービスアクティベータを使用したオブジェクトの活性化（Java）

オブジェクトが構築されるたびに，obj_is_ready を activate()メソッド内で明示的に呼び出す必要があります。サーバプログラムには obj_is_ready に対して二つの呼び出しがあります。第一の呼び出しはサーバがサービスオブジェクトを生成し，生成元プログラムに IOR を返したときに発生します。

コードサンプル 29-13　obj_is_ready へのサーバの第一の呼び出し（Java）

```
public DBObject create_object(String name) {
   System.out.println("Creating: " + name);
   DBObject dbObject = new DBObjectImpl(name);
   _boa().obj_is_ready(dbObject, "DBService", name.getBytes());
   return dbObject;
}
```

obj_is_ready への第二の呼び出しは，activate 内にあり，これは明示的に呼び出す必要があります。コンテキスト内でのこの第二呼び出しについては，コードサンプル 29-9 を参照してください。

29.3.3　サービス活性化オブジェクトインプリメンテーションの非活性化（C++）

サービス活性化の主な用途は，サーバ内で多数のオブジェクトがアクティブであるような錯覚を与えることです。実際には少数のオブジェクトだけがアクティブです。このモデルをサポートするには，サーバは一時的にオブジェクトを使用できないようにする必要があります。マルチスレッド DBActivator のサンプルプログラムには，30 秒ごとにすべての DBObjectImpl を非活性化するリーパスレッドがあります。
DBActivator は，deactivate メソッドが起動されるとオブジェクトリファレンスを解放します。新しいクライアントリクエストが非活性化オブジェクトに到着すると，VisiBroker ORB はオブジェクトを再起動しなければならないことを Activator に通知します。

コードサンプル 29-14　サービス活性化オブジェクトインプリメンテーションを非活性化する例

```
//static sequence of currently active Implementations
static VISMutex         _implMtx;
static DBObjectSequence _impls;

//updated DBActivator to store activated implementations
//in the global sequence.
class DBActivator: public extension::Activator {
   virtual CORBA::Object_ptr activate(
         CORBA::ImplementationDef_ptr impl) {
      extension::ActivationImplDef* actImplDef =
            extension::ActivationImplDef::_downcast(impl);
      CORBA::ReferenceData_var id(actImplDef->id());
      DBObjectImpl *obj = new DBObjectImpl(
            (char *)id->data(), id);
      VISMutex_var lock(_implMtx);
      _impls.length(_impls.length() + 1);
      _impls[_impls.length()-1] = DBObject::_duplicate(obj);
      return obj;
   }
   virtual void deactivate(CORBA::Object_ptr,
      CORBA::ImplementationDef_ptr impl) {
      obj->_release();
   }
};
// Multi-threaded Reaper for destroying all activated
// objects every 30 seconds.
class DBObjectImplReaper : public VISThread {
   public:
      // Reaper methods
```

```
        virtual void start() {
           run();
        }
        virtual CORBA::Boolean startTimer() {
           vsleep(30);
           return 1;
        }
        virtual void begin() {
           while (startTimer()) {
              doOneReaping();
           }
        }
     protected:
        virtual void doOneReaping() {
           VISMutex_var lock(_implMtx);
           for (CORBA::ULong i=0; i < _impls.length(); i++) {
              // assigning nil into each element will release
              // the reference stored in the _var
              DBObject_var obj = DBObject::_duplicate(_impls[i-1];
              _impls[i] = DBObject::_nil();
              CORBA::BOA_var boa = obj->_boa();
              boa->deactivate_obj(obj);
           }
           _impls.length(0);
        }
  };
```

注

マルチスレッドが_impls データ構造にアクセスしているコードの部分は，排他を提供するために VISMutex によって保護されています。

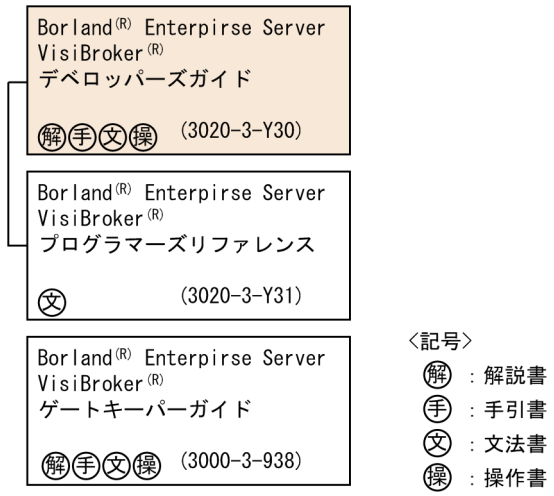
付録

付録 A　このマニュアルの参考情報

このマニュアルを読むに当たっての参考情報を示します。

付録 A.1　関連マニュアル

関連マニュアルを次に示します。必要に応じてお読みください。



なお，CORBA の仕様の詳細については，「The Common Object Request Broker: Architecture and Specification」を参照してください。

付録 A.2　このマニュアルでの表記

このマニュアルでは，製品名を次のように表記しています。

略称	製品名称
AIX	AIX 5L V5.3
	AIX V6.1
	AIX V7.1
Borland Enterprise Server VisiBroker	Borland(R) Enterprise Server VisiBroker(R)
HP-UX	HP-UX 11i V2 (IPF)
	HP-UX 11i V3 (IPF)
IPF	Itanium(R) Processor Family
Java	Java™
JavaServer	JavaServer™
Linux	Red Hat Enterprise Linux(R) AS 4
	Red Hat Enterprise Linux(R) AS 4 (IPF)

略称	製品名称
Linux	Red Hat Enterprise Linux(R) ES 4
	Red Hat Enterprise Linux(R) ES 4 (IPF)
	Red Hat Enterprise Linux(R) 5
	Red Hat Enterprise Linux(R) 5 (IPF)
	Red Hat Enterprise Linux 5.1
	Red Hat Enterprise Linux 6.1
Netscape Communicator	Netscape(R) Communicator
Netscape ディレクトリサーバ	Netscape(R) Directory Server
Solaris	Solaris 9
	Solaris 10
VisiBroker	Borland(R) Enterprise Server VisiBroker(R)
VisiBroker 3.x	VisiBroker Version 3.0（x は 0 以上の整数）
VisiBroker 4.x	VisiBroker Version 4.0（x は 0 以上の整数）
VisiBroker 5.x	VisiBroker Version 5.0（x は 0 以上の整数）

AIX，HP-UX，Linux および Solaris を総称して UNIX と表記しています。

付録 A.3　英略語

このマニュアルで使用する英略語を次に示します。

英略語	英字での表記
ACL	Access Control List
API	Application Programming Interface
BOA	Basic Object Adapter
CA	Certificate Authority
CDE	Common Desktop Environment
CN	Common-Name
CORBA	Common Object Request Broker Architecture
DII	Dynamic Invocation Interface
DLL	Dynamic Linking Library
DNS	Domain Name System
DSI	Dynamic Skeleton Interface
EJB	Enterprise Java Beans(TM)

英略語	英字での表記
GIOP	General Inter - ORB Protocol
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transport Protocol
HTTPS	HTTP-over-SSL
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IOR	Interoperable Object Reference
IR	Interface Repository
J2EE	Java(TM) 2 Platform, Enterprise Edition
JAAS	Java(TM) Authentication and Authorization Service
JavaVM	Java(TM) Virtual Machine
JDBC	Java(TM) Database Connectivity
JDK	Java(TM) Development Kit
JNDI	Java Naming and Directory Interface(TM)
JSSE	Java Secure Sockets Extension
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
NDS	Novell Directory Services
OAD	Object Activation Daemon
OMG	Object Management Group
OODB	Object-Oriented Database
ORB	Object Request Broker
PKC	Public-Key Certificate
POA	Portable Object Adapter
QoP	Quality of Protection
QoS	Quality of Service
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SGML	Standard Generalized Markup Language
SPI	Service Provider Interface

英略語	英字での表記
SSL	Secure Socket Layer
TCP/IP	Transmission Control Protocol/Internet Protocol
TII	Time-Independent Invocation
UDP	User Datagram Protocol
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WWW	World Wide Web
XML	Extensible Markup Language

付録 A.4　KB（キロバイト）などの単位表記について

1KB（キロバイト），1MB（メガバイト），1GB（ギガバイト），1TB（テラバイト）はそれぞれ 1,024 バイト，$1{,}024^2$ バイト，$1{,}024^3$ バイト，$1{,}024^4$ バイトです。

索引

記号

A

B

C

D

E

F

H

I

J

M

N

O

P

Q

R

S

T

U

V

W

X

あ

い

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

て

と

な

に

ぬ

ね

は

ひ

ふ

へ

ほ

ま

め

も

ゆ

ら

り

る

れ

ろ