BEA Tuxedo

BEA Tuxedo CORBA クライアント･アプリケーションの開発方法

BEA Tuxedo 8.0
8.0 版
2001 年 10 月 31 日

Copyright

Copyright © 2001, BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement.It is against the law to copy the software except as specifically allowed in the agreement.This document may not, in whole or in part, be copied photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems.THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, WebLogic, Tuxedo, and Jolt are registered trademarks of BEA Systems, Inc. How Business Becomes E-Business, BEA WebLogic E-Business Platform, BEA Builder, BEA Manager, BEA eLink, BEA WebLogic Commerce Server, BEA WebLogic Personalization Server, BEA WebLogic Process Integrator, BEA WebLogic Collaborate, BEA WebLogic Enterprise, and BEA WebLogic Server are trademarks of BEA Systems, Inc.

All other company names may be trademarks of the respective companies with which they are associated.

BEA Tuxedo CORBA クライアント・アプリケーションの開発方法

Document Edition	Date	Software Version
8.0	2001 年 10 月 31 日	BEA Tuxedo 8.0

目次

このマニュアルについて

対象読者 ... vii
e-docs Web サイト ... viii
マニュアルの印刷方法 ... viii
関連情報 ... viii
サポート情報 ... ix
表記上の規則 ... x

1. CORBA クライアント・アプリケーションの開発概念

クライアント・アプリケーションの概要 ... 1-2
OMG IDL ... 1-3
OMG IDL と C++ のマッピング ... 1-3
OMG IDL と Java のマッピング ... 1-3
OMG IDL と COM のマッピング ... 1-4
静的起動と動的起動 ... 1-4
クライアント・スタブ ... 1-6
インターフェイス・リポジトリ ... 1-7
ドメイン ... 1-8
環境オブジェクト ... 1-9
Bootstrap オブジェクト ... 1-11
ファクトリと FactoryFinder オブジェクト ... 1-13
FactoryFinder オブジェクトの命名規則と BEA Tuxedo 拡張 ... 1-14
InterfaceRepository オブジェクト ... 1-16
SecurityCurrent オブジェクト ... 1-17
TransactionCurrent オブジェクト ... 1-19
NotificationService オブジェクトと Tobj_SimpleEventsService オブジェクト ... 1-20
NameService オブジェクト ... 1-21
ActiveX クライアント・アプリケーションの概念 ... 1-22
ActiveX とは ... 1-22
ビューとバインディング ... 1-22

ActiveX ビューの命名規則 1-24

2. CORBA クライアント・アプリケーションの作成

CORBA C++ クライアント・アプリケーションの開発プロセスの概要 2-2
CORBA Java クライアント・アプリケーションの開発プロセスの概要 2-3
ステップ 1: OMG IDL ファイルの取得 2-4
ステップ 2: 呼び出し方式の選択 2-6
ステップ 3: OMG IDL ファイルのコンパイル 2-6
ステップ 4: CORBA クライアント・アプリケーションの記述 2-8
ORB の初期化 2-8
BEA Tuxedo ドメインとの通信の確立 2-9
FactoryFinder オブジェクトへの初期リファレンスの解決 2-11
FactoryFinder オブジェクトを使用したファクトリの取得 2-12
ファクトリを使用した CORBA オブジェクトの取得 2-13
ステップ 5: CORBA クライアント・アプリケーションのビルド 2-14
クライアント・アプリケーションとして動作するサーバ・アプリケーション 2-15
Java2 アプレットの使い方 2-15

3. ActiveX クライアント・アプリケーションの作成

ActiveX クライアント・アプリケーションの開発プロセスの概要 3-2
BEA Application Builder 3-3
ステップ 1: オートメーション環境オブジェクトのインターフェイス・リポジトリへのロード 3-4
ステップ 2: CORBA インターフェイスのインターフェイス・リポジトリへのロード 3-4
ステップ 3: インターフェイス・リポジトリのサーバ・アプリケーションの起動 3-5
ステップ 4: CORBA インターフェイスの ActiveX バインディングの作成 3-6
ステップ 5: ActiveX バインディングのタイプ・ライブラリのロード 3-8
ステップ 6: ActiveX クライアント・アプリケーションの記述 3-8
オートメーション環境オブジェクト、ファクトリ、および CORBA オブジェクトの ActiveX ビューの宣言のインクルード 3-9
BEA Tuxedo ドメインとの通信の確立 3-9
FactoryFinder オブジェクトへの初期リファレンスの取得 3-11

ファクトリを使用した ActiveX ビューの取得 3-11
ActiveX ビューのオペレーションの呼び出し 3-12
ステップ 7: ActiveX クライアント・アプリケーションのデプロイ 3-13

4. 動的起動インターフェイスの使い方

DII の使用 4-2
DII の概念 4-3
リクエスト・オブジェクト 4-3
要求送信オプション 4-4
要求結果受信オプション 4-5
DII の開発プロセスの概要 4-6
ステップ 1: CORBA インターフェイスのインターフェイス・
リポジトリへのロード..4-7
ステップ 2: CORBA オブジェクトのオブジェクト・リファレンスの
取得 4-8
ステップ 3: リクエスト・オブジェクトの作成 4-8
CORBA::Object::_request メンバ関数の使い方 4-9
CORBA::Object::create_request メンバ関数の使い方 4-9
リクエスト・オブジェクトの引数の設定...................................4-10
CORBA::NamedValue メンバ関数による入力引数と出力引数の
設定...4-10
CORBA::Object::create_request メンバ関数の使用例 4-10
ステップ 4: DII 要求の送信と結果の取得 4-11
同期要求 4-12
遅延同期要求...4-12
oneway の要求...4-13
複数の要求...4-13
ステップ 5: 要求の削除 4-17
ステップ 6: DII でのインターフェイス・リポジトリの使い方.................4-17

5. 例外処理

CORBA 例外処理の概念 5-1
CORBA システム例外 5-1
CORBA C++ クライアント・アプリケーション.....................................5-3
システム例外の処理 5-5
ユーザ例外...5-6

CORBA Java クライアント・アプリケーション 5-7
システム例外 5-8
ユーザ例外 5-9
ActiveX クライアント・アプリケーション 5-9

索引

このマニュアルについて

このマニュアルでは、BEA Tuxedo 製品の CORBA 環境で CORBA C++、CORBA Java、および ActiveX クライアント・アプリケーションを作成する方法について説明します。このマニュアルでは、製品の重要な概念とクライアント・アプリケーションを作成する手順について説明します。また、コード例を使用して開発プロセスについて説明します。

このマニュアルでは、以下の内容について説明します。

- 「第 1 章 CORBA クライアント・アプリケーションの開発概念」では、BEA Tuxedo ソフトウェアを使用して CORBA クライアント・アプリケーションを開発する上で知っておく必要のある概念について説明します。
- 「第 2 章 CORBA クライアント・アプリケーションの作成」では、CORBA C++ および CORBA Java クライアント・アプリケーションを作成する手順について説明します。
- 「第 3 章 ActiveX クライアント・アプリケーションの作成」では、ActiveX クライアント・アプリケーションを作成する手順について説明します。
- 「第 4 章 動的起動インターフェイスの使い方」では、CORBA C++ および CORBA Java クライアント・アプリケーションで動的起動インターフェイス (DII) を使用する方法について説明します。
- 「第 5 章 例外処理」では、CORBA C++、CORBA Java、および ActiveX クライアント・アプリケーションで CORBA 例外を処理する方法について説明します。

対象読者

このマニュアルは、BEA Tuxedo ソフトウェアを使用して CORBA クライアント・アプリケーションを開発するプログラマを対象としています。

e-docs Web サイト

BEA Tuxedo 製品のマニュアルは BEA 社の Web サイトで参照することができます。BEA ホーム・ページの [製品のドキュメント] をクリックするか、または http://edocs.beasys.co.jp/e-docs/index.html に直接アクセスしてください。

マニュアルの印刷方法

このマニュアルは、ご使用の Web ブラウザで一度に 1 ファイルずつ印刷できます。Web ブラウザの [ファイル] メニューにある [印刷] オプションを使用してください。

このマニュアルの PDF 版は、Web サイト上にあります。また、マニュアルの CD-ROM にも収められています。この PDF を Adobe Acrobat Reader で開くと、マニュアル全体または一部をブック形式で印刷できます。PDF 形式を利用するには、BEA Tuxedo マニュアルのホーム・ページにある [PDF 版] ボタンをクリックして、印刷するマニュアルを選択します。

Adobe Acrobat Reader をお持ちでない場合は、Adobe 社の Web サイト (http://www.adobe.co.jp/) から無償でダウンロードできます。

関連情報

CORBA、BEA Tuxedo、分散オブジェクト・コンピューティング、トランザクション処理、C++ プログラミング、および Java プログラミングの詳細については、BEA Tuxedo オンライン・マニュアルの「Bibliography」を参照してください。

サポート情報

皆様の BEA Tuxedo マニュアルに対するフィードバックをお待ちしています。ご意見やご質問がありましたら、電子メールで **docsupport-jp@bea.com** までお送りください。お寄せいただきましたご意見は、BEA Tuxedo マニュアルの作成および改訂を担当する BEA 社のスタッフが直接検討いたします。

電子メール メッセージには、BEA Tuxedo 8.0 リリースのマニュアルを使用していることを明記してください。

BEA Tuxedo に関するご質問、または BEA Tuxedo のインストールや使用に際して問題が発生した場合は、www.bea.com の BEA WebSUPPORT を通して BEA カスタマ・サポートにお問い合わせください。カスタマ・サポートへの問い合わせ方法は、製品パッケージに同梱されているカスタマ・サポート・カードにも記載されています。

カスタマ・サポートへお問い合わせの際には、以下の情報をご用意ください。

- お客様のお名前、電子メール・アドレス、電話番号、Fax 番号
- お客様の会社名と会社の住所
- ご使用のマシンの機種と認証コード
- ご使用の製品名とバージョン
- 問題の説明と関連するエラー・メッセージの内容

表記上の規則

このマニュアルでは、以下の表記規則が使用されています。

規則	項目
太字	用語集に定義されている用語を示します。
Ctrl + Tab	2 つ以上のキーを同時に押す操作を示します。
イタリック体	強調またはマニュアルのタイトルを示します。

等幅テキスト	コード・サンプル、コマンドとオプション、データ構造とメンバ、データ型、ディレクトリ、およびファイル名と拡張子を示します。また、キーボードから入力するテキストも等幅テキストで表示します。
	例 :
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

太字の等幅テキスト	コード内の重要な語を示します。 例 : `void` **`commit`** `( )`
斜体の等幅テキスト	コード内の変数を示します。 例 : `String` *`expr`*

規則	項目
大文字のテキスト	デバイス名、環境変数、および論理演算子を示します。 例 : LPT1 SIGNON OR
`{ }`	構文の行で、選択肢の組み合わせを示します。かっこは入力しません。
`[ ]`	構文の行で、オプション項目を示します。かっこは入力しません。 例 : `buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...[-l` *`file-list`*`]...`
`|`	構文の行で、相互に排他的な選択肢の区切りとして使います。記号は入力しません。
`...`	コマンド・ラインで、以下のいずれかの場合を示します。 ■ コマンド・ラインで、同じ引数を繰り返し使用できることを示します。 ■ 文で、追加のオプション引数が省略されていることを示します。 ■ 追加のパラメータ、値、またはその他の情報を入力できることを示します。 記号は入力しません。 例 : `buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...[-l` *`file-list`*`]...`
. . .	コード例または構文の行で、項目が省略されていることを示します。記号は入力しません。

1 CORBA クライアント・アプリケーションの開発概念

この章では、BEA Tuxedo 製品の CORBA 環境でサポートされるクライアント・アプリケーションのタイプと、CORBA クライアント・アプリケーションの開発前に理解しておく必要がある概念について説明します。

ここでは、次の内容について説明します。

- クライアント・アプリケーションの概要
- OMG IDL
- 静的起動と動的起動
- クライアント・スタブ
- インターフェイス・リポジトリ
- ドメイン
- 環境オブジェクト
- ActiveX クライアント・アプリケーションの概念

クライアント・アプリケーションの概要

BEA Tuxedo ソフトウェアは、以下のタイプのクライアント・アプリケーションをサポートしています。

- CORBA C++

  このタイプのクライアント・アプリケーションは、C++ 環境オブジェクトを使用して BEA Tuxedo ドメイン内の CORBA オブジェクトにアクセスし、CORBA C++ Object Request Broker (ORB) を使用して CORBA オブジェクトへの要求を処理します。CORBA C++ クライアント・アプリケーションをビルドするには、BEA Tuxedo 開発コマンドを使用します。CORBA C++ クライアント・アプリケーションは、OBV (Object-by-Value) と CORBA インターオペラブル・ネーミング・サービス (INS) をサポートしています。

- CORBA Java

  このタイプのクライアント・アプリケーションは、Java 環境オブジェクトを使用して BEA Tuxedo ドメイン内の CORBA オブジェクトにアクセスします。しかし、これらのクライアント・アプリケーションは、BEA Tuxedo CORBA ORB 以外の ORB 製品を使用して CORBA オブジェクトへの要求を処理します。CORBA Java クライアント・アプリケーションは、ORB 製品の Java 開発ツールを使用してビルドします。BEA Tuxedo ソフトウェアは、Sun Java Development Kit (JDK) Java クライアントとの相互運用性をサポートしています。

  注記　サポートされるソフトウェアのバージョンについては、『BEA Tuxedo システムのインストール』を参照してください。

- ActiveX

  このタイプのクライアント・アプリケーションは、オートメーション環境オブジェクトを使用して BEA Tuxedo ドメイン内の CORBA オブジェクトにアクセスし、BEA ActiveX Client を使用して CORBA オブジェクトへの要求を処理します。Application Builder を使用すると、ActiveX クライアント・アプリケーションで使用可能な CORBA インターフェイスを選択して、その CORBA インターフェイスの ActiveX ビューと、その ActiveX ビューをクライアント・マシンにデプロイするためのパッケージを作成できます。これらのクライアント・アプリケーションは、Visual Basic や PowerBuilder などのオートメーション開発ツールを使用してビルドします。

OMG IDL

どのような分散アプリケーションでも、クライアント / サーバ・アプリケーションは通信を行うための基本的な情報を必要とします。たとえば、CORBA クライアント・アプリケーションは、要求できるオペレーションとその引数を知る必要があります。

Object Management Group (OMG) インターフェイス定義言語 (IDL) を使用すると、クライアント・アプリケーションへの CORBA インターフェイスを定義できます。OMG IDL で記述されたインターフェイス定義では、CORBA インターフェイスが完全に定義され、各オペレーションの引数が完全に指定されます。OMG IDL は、純粋な宣言型言語です。このため、OMG IDL には実装の詳細は定義されません。OMG IDL で指定されるオペレーションは、CORBA バインディングを提供する任意の言語で記述し、呼び出すことができます。サポートされる言語には、C++ と Java が含まれます。

一般に、アプリケーション設計者が使用可能な CORBA インターフェイスとオペレーション用の OMG IDL ファイルをプログラマに提供し、プログラマがクライアント・アプリケーションを開発します。

OMG IDL と C++ のマッピング

BEA Tuxedo ソフトウェアは、「The Common Object Request Broker:Architecture and Specification, Version 2.3」に準拠しています。OMG IDL と C++ のマッピングの詳細については、「The Common Object Request Broker:Architecture and Specification, Version 2.3」を参照してください。

OMG IDL と Java のマッピング

BEA Tuxedo ソフトウェアは、「The Common Object Request Broker:Architecture and Specification, Version 2.2」に準拠しています。OMG IDL と Java のマッピングの詳細については、「The Common Object Request Broker:Architecture and Specification, Version 2.2」を参照してください。

OMG IDL と COM のマッピング

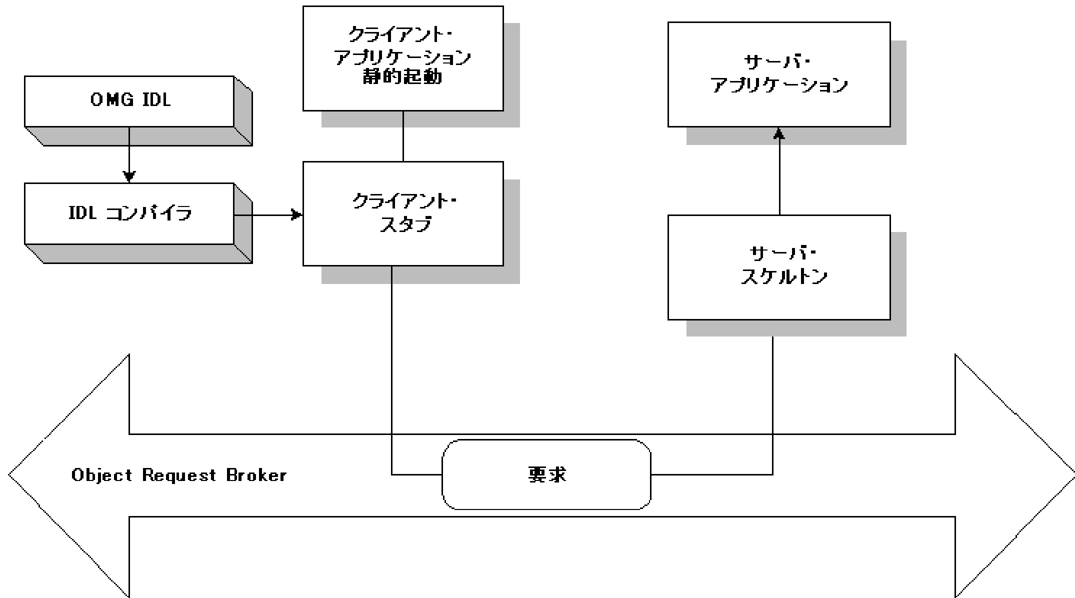
BEA Tuxedo ソフトウェアは、「The Common Object Request Broker:Architecture and Specification, Version 2.3」に定義されている OMG IDL と COM のマッピングに準拠しています。OMG IDL と COM のマッピングの詳細については、「The Common Object Request Broker:Architecture and Specification, Version 2.3」を参照してください。

静的起動と動的起動

BEA Tuxedo 製品の CORBA ORB は、静的と動的という 2 種類のクライアント / サーバ起動をサポートしています。どちらのケースでも、CORBA クライアント・アプリケーションは CORBA オブジェクトのリファレンスへのアクセスを取得し、要求を満たすオペレーションを呼び出すことによってその要求を実行します。CORBA サーバ・アプリケーションは、静的起動と動的起動の違いを区別できません。

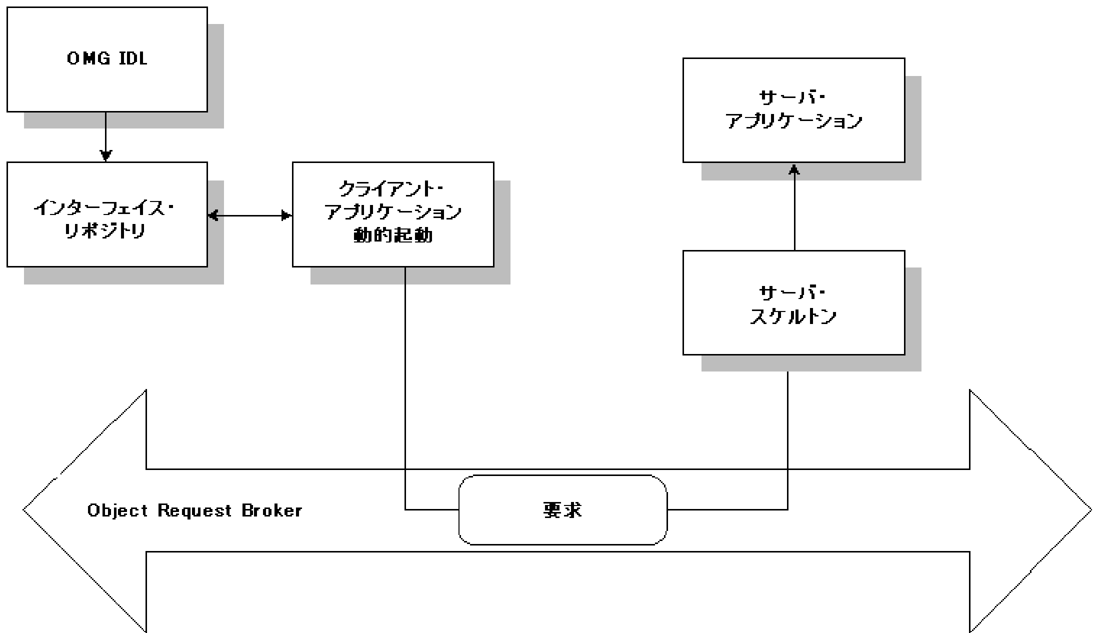
静的起動を使用する場合、CORBA クライアント・アプリケーションはクライアント・スタブ上でオペレーションを直接呼び出します。静的起動は、最も簡単で、最も一般的な呼び出し方式です。スタブは、IDL コンパイラによって生成されます。静的起動は、呼び出す必要があるオペレーションの詳細をコンパイル時に認識し、その呼び出しの同期的性質内で処理できるアプリケーションに適しています。図 1-1 に、静的起動のしくみを示します。

図 1-1　静的起動



動的起動は、より複雑です。ただし、動的起動を使用すると、CORBA クライアント・アプリケーションはコンパイル時に CORBA オブジェクトのインターフェイスを認識しなくても CORBA オブジェクトのオペレーションを呼び出すことができます。図 1-2 に、動的起動のしくみを示します。

図 1-2　動的起動



動的起動を使用すると、CORBA クライアント・アプリケーションは、インターフェイス・リポジトリに格納されている CORBA オブジェクト・インターフェイス用のオペレーション要求を動的に構築できます。CORBA サーバ・アプリケーションは、特別な設計を必要とせずに動的起動要求を受け付けて処理できます。通常、動的起動は CORBA クライアント・アプリケーションで遅延同期通信が必要なときに使用されるか、または対話の性質が未定義の場合に動的クライアント・アプリケーションによって使用されます。動的起動の詳細については、「動的起動インターフェイスの使い方」を参照してください。

クライアント・スタブ

クライアント・スタブは、CORBA オブジェクトが実行できるオペレーションへのプログラミング・インターフェイスを提供します。クライアント・スタブは、CORBA オブジェクトのローカル・プロキシです。クライアント・スタブは、CORBA オブジェクトのオブジェクト・リファレンスの同期呼び出しを実行する

ためのメカニズムを提供します。CORBA クライアント・アプリケーションは、特別なコードを必要とせずに CORBA オブジェクトまたはその引数を処理できます。CORBA クライアント・アプリケーションは、スタブをローカル・オブジェクトとして取り扱います。

CORBA クライアント・アプリケーションは、使用するインターフェイスごとにスタブを持つ必要があります。`idl` コマンド (または Java ORB 製品の同等コマンド) を使用すると、CORBA インターフェイスの OMG IDL 定義からクライアント・スタブを生成できます。このコマンドにより、C++ や Java などのプログラミング言語からクライアント・スタブを使用する場合に必要なすべてのものが定義されたスタブ・ファイルとヘッダ・ファイルが生成されます。このため、CORBA クライアント・アプリケーション内からメソッドを呼び出すだけで、CORBA オブジェクトのオペレーションを要求できます。

インターフェイス・リポジトリ

インターフェイス・リポジトリには、CORBA オブジェクトのインターフェイスとオペレーションの定義が含まれています。インターフェイス・リポジトリに格納される情報は OMG IDL ファイルに定義される情報と同じですが、この情報には実行時にプログラマティックにアクセス可能です。CORBA クライアント・アプリケーションがインターフェイス・リポジトリを使用する理由は以下のとおりです。

- 動的起動を使用する CORBA クライアント・アプリケーションは、インターフェイス・リポジトリを使用して CORBA オブジェクトのインターフェイスについて学習し、そのオブジェクトのオペレーションを呼び出します。
- ActiveX クライアント・アプリケーションは、インターフェイス・リポジトリを使用していることを認識しません。ActiveX クライアント・アプリケーションは、CORBA オペレーションを使用してインターフェイス・リポジトリから CORBA オブジェクトに関する情報を取得します。

静的起動を使用する CORBA クライアント・アプリケーションは、実行時にインターフェイス・リポジトリにアクセスしません。CORBA オブジェクトのインターフェイスに関する情報は、クライアント・スタブに含まれています。

インターフェイス・リポジトリを管理するには、以下の BEA Tuxedo 開発コマンドを使用します。

- `idl2ir` コマンドを使用すると、インターフェイス・リポジトリに CORBA インターフェイスを追加できます。インターフェイス・リポジトリが存在しない場合は、このコマンドでインターフェイス・リポジトリを作成できます。また、インターフェイス・リポジトリ内の CORBA インターフェイスを更新できます。
- `ir2idl` コマンドを使用すると、インターフェイス・リポジトリの内容から OMG IDL ファイルを作成できます。
- `irdel` コマンドを使用すると、インターフェイス・リポジトリから CORBA インターフェイスを削除できます。

インターフェイス・リポジトリの開発コマンドについては、『BEA Tuxedo コマンド・リファレンス』を参照してください。

ドメイン

ドメインは、オブジェクトとサービスを管理エンティティとしてグループ化するための方法です。BEA Tuxedo ドメインは、最低 1 つの IIOP リスナ / ハンドラを持ち、名前によって識別されます。異なる Bootstrap オブジェクトを使用することで、1 つの CORBA クライアント・アプリケーションが複数の BEA Tuxedo ドメインに接続できます。BEA Tuxedo ドメインごとに、CORBA クライアント・アプリケーションはその BEA Tuxedo ドメイン内で提供されるサービス (トランザクション、セキュリティ、ネーミング、イベントなど) に対応するオブジェクトを取得できます。Bootstrap オブジェクトと BEA Tuxedo ドメインで使用可能な CORBA サービスについては、「環境オブジェクト」を参照してください。

注記 サービスごとに 1 つの環境オブジェクトだけが同時に存在でき、環境オブジェクトは同じ Bootstrap オブジェクトに関連付けられる必要があります。

図 1-3 に、BEA Tuxedo ドメインの機能を示します。

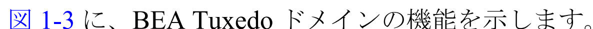
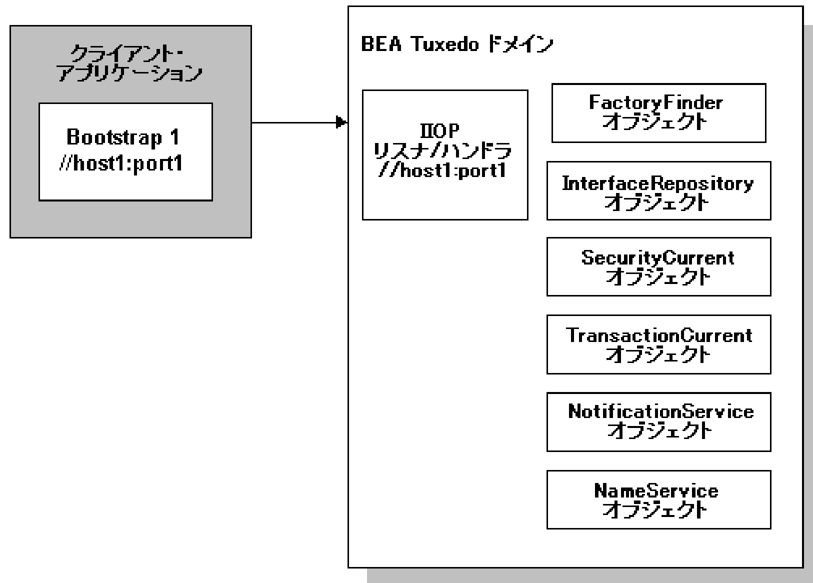


図 1-3　BEA Tuxedo ドメインの機能



環境オブジェクト

BEA Tuxedo ソフトウェアには、CORBA クライアント・アプリケーションと BEA Tuxedo ドメイン内の CORBA サーバ・アプリケーション間の通信を設定し、そのドメインで提供される CORBA サービスへのアクセスを提供する環境オブジェクト・セットが用意されています。BEA Tuxedo ソフトウェアには、以下の環境オブジェクトが用意されています。

■ Bootstrap

  このオブジェクトは、CORBA クライアント・アプリケーションと BEA Tuxedo ドメイン間の通信を確立します。また、BEA Tuxedo ドメイン内のほかの環境オブジェクトのオブジェクト・リファレンスを取得します。

注記 サードパーティ・クライアント ORB は、CORBA インターオペラブル・ネーミング・サービス (INS) を使用して BEA Tuxedo ドメイン内のサービスにアクセスできます。詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』の「CORBA ブートストラップ処理のプログラミング・リファレンス」を参照してください。

- FactoryFinder

  この CORBA オブジェクトは、ファクトリを検索します。このファクトリは、CORBA オブジェクトのオブジェクト・リファレンスを作成できます。

- InterfaceRepository

  この CORBA オブジェクトには、使用可能なすべての CORBA インターフェイスのインターフェイス定義と、CORBA インターフェイスへのオブジェクト・リファレンスを作成するためのファクトリが含まれています。

- SecurityCurrent

  この BEA 固有のオブジェクトは、CORBA クライアント・アプリケーションが適切なセキュリティ・クリデンシャルを使用して BEA Tuxedo ドメインにログインするために使用されます。BEA Tuxedo ソフトウェアは、CORBA サービス・セキュリティ・サービスの実装を提供します。

- TransactionCurrent

  この BEA 固有のオブジェクトは、CORBA クライアント・アプリケーションがトランザクションに参加できるようにします。TransactionCurrent オブジェクトは、CORBA のオブジェクト・トランザクション・サービス (OTS) の実装を提供します。

- NotificationService

  この CORBA オブジェクトを使用すると、CORBA クライアント・アプリケーションは CosNotification サービス内のイベント・チャネル・ファクトリ (`CosNotifyChannelAdmin::EventChannelFactory`) へのリファレンスを取得できます。EventChannelFactory は、ノーティフィケーション・サービス・チャネルの検索に使用されます。

  また、`Tobj_SimpleEventsService` オブジェクトも用意されています。この BEA 固有のオブジェクトを使用すると、CORBA クライアント・アプリケーションは BEA 固有のイベント・インターフェイスへのリファレンスを取得できます。このイベント・インターフェイスは、CosNotification サービスで

定義される標準の構造化されたイベントを受け渡しますが、API は簡単に使用できるよう簡素化されています。

- NameService

  この CORBA オブジェクトを使用すると、CORBA クライアント・アプリケーションは名前空間を使用してオブジェクト・リファレンスを解決できます。BEA Tuxedo ソフトウェアは、CORBA サービス・ネーミング・サービスの実装を提供します。

BEA Tuxedo ソフトウェアには、以下のプログラミング環境用の環境オブジェクトが用意されています。

- C++
- Java
- オートメーション

Bootstrap オブジェクト

CORBA クライアント・アプリケーションは、IIOP リスナ / ハンドラのアドレスを定義する Bootstrap オブジェクトを作成します。IIOP リスナ / ハンドラは、BEA Tuxedo ドメインおよびそのドメインによって提供される CORBA サービスへのアクセス・ポイントです。IIOP リスナ / ハンドラのリストは、パラメータとして提供されるか、`TOBJADDR` 環境変数または Java プロパティを介して提供されます。1 つの IIOP リスナ / ハンドラは、次のように指定されます。

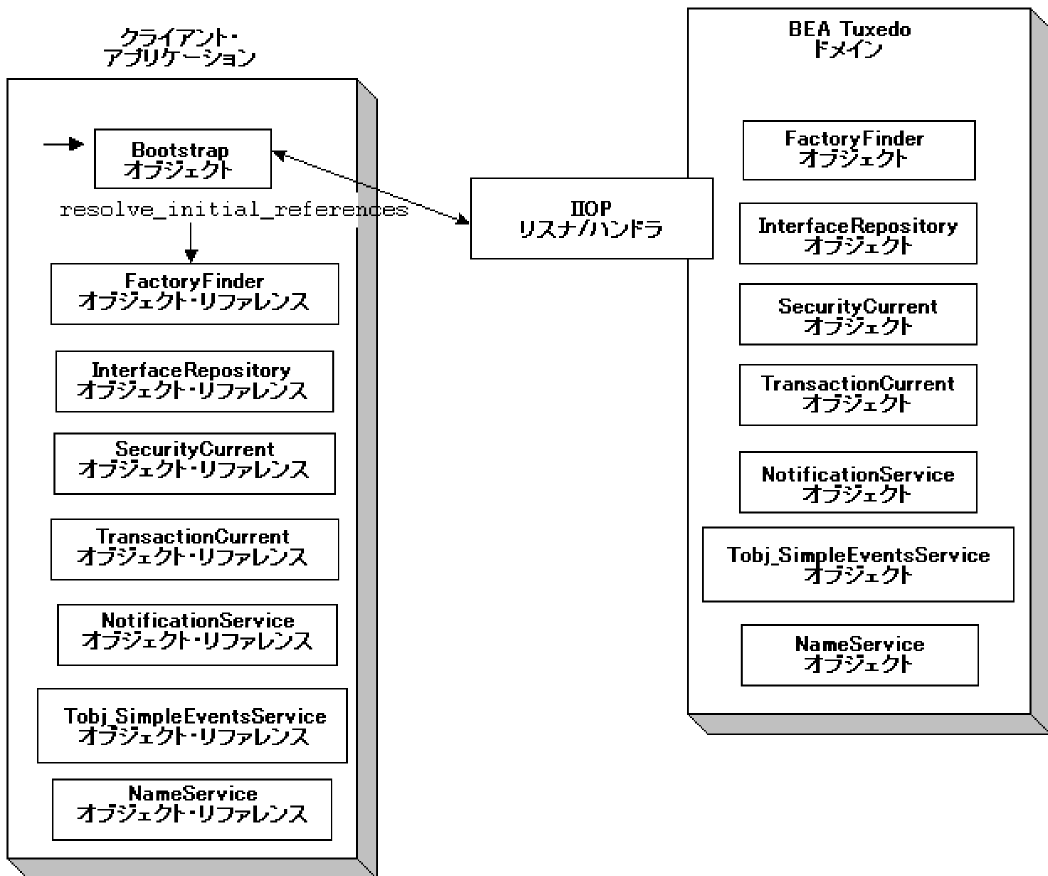
`//`*`host:port`*

例 : `//myserver:4000`

Bootstrap オブジェクトがインスタンス化されると、`resolve_initial_references` メソッドが呼び出され、文字列 ID が受け渡されて使用可能なオブジェクトへのリファレンスが取得されます。文字列 ID の有効値は、FactoryFinder、Interface Repository、SecurityCurrent、TransactionCurrent、NotificationService、TObj_SimpleEventsService、および NameService です。

図 1-4 に、BEA Tuxedo ドメインでの Bootstrap オブジェクトの機能を示します。

図 1-4　Bootstrap オブジェクトの機能



サードパーティ・クライアント ORB は、CORBA インターオペラブル・ネーミング・サービス (INS) メカニズムを使用して BEA Tuxedo ドメインとそのサービスにアクセスできます。インターオペラブル・ネーミング・サービスを使用すると、サードパーティ・クライアント ORB は、自身の `resolve_initial_references()` 関数を使用して BEA Tuxedo ドメインによって提供される CORBA サービスにアクセスし、標準 OMG IDL から生成されたスタブを使用してドメインから返されたインスタンスを処理できます。インターオペラブル・ネーミング・サービスの使い方については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

ファクトリと FactoryFinder オブジェクト

CORBA クライアント・アプリケーションは、CORBA オブジェクトへのリファレンスをファクトリから取得します。ファクトリは、別の CORBA オブジェクトへのリファレンスを返し、自身を FactoryFinder オブジェクトに登録する任意の CORBA オブジェクトです。

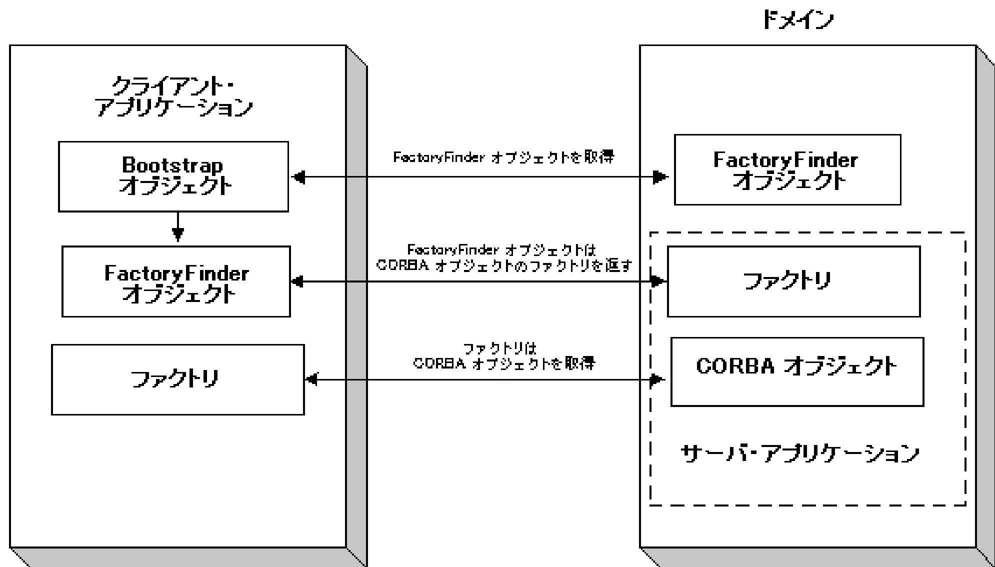
CORBA クライアント・アプリケーションが CORBA オブジェクトを使用するには、その CORBA オブジェクトへのオブジェクト・リファレンスを作成するファクトリを検索する必要があります。BEA Tuxedo ソフトウェアには、そのために FactoryFinder オブジェクトが用意されています。CORBA クライアント・アプリケーションで使用可能なファクトリは、起動時に CORBA サーバ・アプリケーションによって FactoryFinder オブジェクトに登録されたファクトリです。

CORBA クライアント・アプリケーションは、次の一連のステップを使用して CORBA オブジェクトへのリファレンスを取得します。

1. Bootstrap オブジェクトが作成されると、`resolve_initial_references` メソッドが呼び出され、FactoryFinder オブジェクトへのリファレンスが取得されます。
2. CORBA クライアント・アプリケーションは、FactoryFinder オブジェクトに目的のファクトリへのオブジェクト・リファレンスを問い合わせます。
3. 次に、CORBA クライアント・アプリケーションはそのファクトリを呼び出して CORBA オブジェクトへのオブジェクト・リファレンスを取得します。

図 1-5 に、CORBA クライアント・アプリケーションと FactoryFinder オブジェクトの対話を示します。

図 1-5　クライアント・アプリケーションによる FactoryFinder オブジェクトの使用



FactoryFinder オブジェクトの命名規則と BEA Tuxedo 拡張

CORBA クライアント・アプリケーションで使用可能なファクトリは、起動時に CORBA サーバ・アプリケーションによって FactoryFinder オブジェクトに登録されたファクトリです。ファクトリは、以下のフィールドで構成されるキーを使用して登録されます。

- ファクトリのインターフェイスのインターフェイス・リポジトリ
- ファクトリへのオブジェクト・リファレンス

BEA Tuxedo ソフトウェアによって使用される FactoryFinder オブジェクトは、CORBA サービス・ライフサイクル・サービスで定義されます。BEA Tuxedo ソフトウェアは、`COS::LifeCycle::FactoryFinder` の拡張を実装します。この拡張により、クライアント・アプリケーションは FactoryFinder オブジェクトを使用してより簡単にファクトリを検索できるようになります。

CORBA サービス・ライフサイクル・サービスは、CORBA サービス・ネーミング・サービスに定義された名前を使用して、`COS::LifeCycle::FactoryFinder` インターフェイスを介してファクトリを検索するよう指定しています。これらの名前は一連の *`NameComponent`* 構造で構成され、この構造は `ID` フィールドと `kind` フィールドで構成されています。

CORBA 名を使用したファクトリの検索は、クライアント・アプリケーションにとっては面倒です。数多くの呼び出しを行って適切な名前構造を構築し、CORBA ネーム・サービス名を構築して `COS::LifeCycle::FactoryFinder` インターフェイスの `find_factories` メソッドに渡す必要があるからです。また、メソッドは複数のファクトリを返す場合があるため、クライアント・アプリケーションは適切なファクトリの選択と不要なオブジェクト・リファレンスの破棄を行う必要があります。

FactoryFinder オブジェクトは、単純なメソッド呼び出しによってインターフェイスを拡張することで、CORBA クライアント・アプリケーションがファクトリをより簡単に検索できるよう設計されています。

FactoryFinder 拡張の目的は、CORBA クライアント・アプリケーションに対して以下の簡素化を提供することです。

- CORBA クライアント・アプリケーションは ID フィールド用の単純な文字列パラメータを使用して、ID によってファクトリを検索できます。これにより、CORBA クライアント・アプリケーションが名前構造を構築するために必要な作業が削減されます。
- FactoryFinder オブジェクトは、使用可能なファクトリのプールから選択することによってロード・バランシング機構を実装できます。
- 一連のオブジェクト・リファレンスの代わりに 1 つのオブジェクト・リファレンスを返すメソッドを提供します。これにより、CORBA クライアント・アプリケーションはシーケンスから 1 つのファクトリを選択し、不要なリファレンスを破棄するためのコードを提供する必要がなくなります。

最も単純なアプリケーション設計は、CORBA クライアント・アプリケーションで `Tobj::FactoryFinder::find_one_factory_by_id` メソッドを使用することで達成できます。このメソッドは、入力としてファクトリ ID 用の単純な文字列を受け付け、1 つのファクトリを CORBA クライアント・アプリケーションに返します。CORBA クライアント・アプリケーションは、名前コンポーネントを操作し、多くのファクトリから選択する必要がなくなります。

`Tobj::FactoryFinder::find_one_factory_by_id` メソッドを使用するには、アプリケーション設計者は CORBA クライアント・アプリケーションが特定の CORBA オブジェクト・インターフェイス用のファクトリを簡単に検索するために使用できるファクトリの命名規則を定義する必要があります。この命名規則では、特定のタイプの CORBA オブジェクト・インターフェイスのオブジェクト・リファレンスを提供するファクトリの複数のニーモニック型が定義されるのが理想的です。ファクトリは、これらの規則を使用して登録されます。たとえば、Student オブジェクトのオブジェクト・リファレンスを返すファクトリであれば、StudentFactory と呼ばれます。FactoryFinder オブジェクトへのファクトリの登録については、『BEA Tuxedo CORBA サーバ・アプリケーションの開発方法』を参照してください。

OMG IDL ファイルでファクトリの実際のインターフェイス ID を使用するか、OMG IDL ファイルでファクトリ ID を定数として指定することをお勧めします。このテクニックを使用することにより、CORBA クライアント・アプリケーションと CORBA サーバ・アプリケーション間の命名の一貫性が保証されます。

InterfaceRepository オブジェクト

InterfaceRepository オブジェクトは、BEA Tuxedo ドメインのインターフェイス・リポジトリに関する情報を返します。InterfaceRepository オブジェクトは、インターフェイス・リポジトリの CORBA 定義に基づいています。このオブジェクトは、「Common Request Broker Architecture and Specification Version 2.2」で定義されている適切な CORBA インターフェイス・セットを提供します。

動的起動インターフェイス (DII) を使用する CORBA クライアント・アプリケーションは、インターフェイス・リポジトリにプログラマティックにアクセスする必要があります。インターフェイス・リポジトリにアクセスするための正確な手順は、CORBA クライアント・アプリケーションが特定の CORBA インターフェイスに関する情報を検索するのか、またはあるインターフェイスを見つけるため

にリポジトリを参照するのかによって異なります。どちらの場合でも、CORBA クライアント・アプリケーションはインターフェイス・リポジトリへの読み込みだけを行うことができ、書き込みは行うことができません。

DII を使用する CORBA クライアント・アプリケーションが BEA Tuxedo ドメインのインターフェイス・リポジトリを参照するには、CORBA クライアント・アプリケーションがあらかじめそのドメインの InterfaceRepository オブジェクトのオブジェクト・リファレンスを取得しておく必要があります。DII を使用する CORBA クライアント・アプリケーションは、Bootstrap オブジェクトを使用してオブジェクト・リファレンスを取得します。

また、ActiveX クライアント・アプリケーションもインターフェイス・リポジトリを使用します。CORBA クライアント・アプリケーションと同様、ActiveX クライアント・アプリケーションも Bootstrap オブジェクトを使用して InterfaceRepository オブジェクトへのリファレンスを取得します。

DII を使用する CORBA クライアント・アプリケーションで InterfaceRepository オブジェクトを使用する方法については、「動的起動インターフェイスの使い方」を参照してください。InterfaceRepository オブジェクトについては、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

SecurityCurrent オブジェクト

CORBA C++、CORBA Java、および ActiveX クライアント・アプリケーションは、セキュリティを使用して BEA Tuxedo ドメインの認証を受けます。認証とは、クライアント・アプリケーションの ID を検証するプロセスです。正確なログオン情報を入力することによって、クライアント・アプリケーションは BEA Tuxedo ドメインの認証を受けます。BEA Tuxedo ソフトウェアは、CORBA サービス・セキュリティ・サービスで定義された認証を使用し、使い勝手を良くするための拡張を提供します。

CORBA クライアント・アプリケーションは、SecurityCurrent オブジェクトを使用して BEA Tuxedo ドメインにログオンし、セキュリティ・クリデンシャルをドメインに渡します。SecurityCurrent オブジェクトは、CORBA サービス・セキュリティ・サービスの BEA Tuxedo の実装です。BEA Tuxedo 製品の CORBA セキュリティ・モデルは、認証をベースとしています。

SecurityCurrent オブジェクトを使用することによって、ドメインの適切なセキュリティ・レベルを指定します。利用できる認証レベルは以下のとおりです。

- TOBJ_NOAUTH

  認証はまったく必要ありません。ただし、CORBA クライアント・アプリケーションは認証を受け、ユーザ名とクライアント・アプリケーション名を指定することができます。パスワードは不要です。

- TOBJ_SYSAUTH

  CORBA クライアントは BEA Tuxedo ドメインの認証を受け、ユーザ名、クライアント・アプリケーション名、およびアプリケーション・パスワードを指定する必要があります。

- TOBJ_APPAUTH

  TOBJ_SYSAUTH 情報のほかに、CORBA クライアント・アプリケーションはアプリケーション固有の情報を指定する必要があります。デフォルトの BEA Tuxedo 認証サービスがアプリケーション・コンフィギュレーションで使用される場合、CORBA クライアント・アプリケーションはユーザ・パスワードを指定する必要があります。それ以外の場合、CORBA クライアント・アプリケーションはそのアプリケーションのカスタム認証サービスによって解釈される認証データを指定します。

注記　CORBA クライアント・アプリケーションが認証を受けず、セキュリティ・レベルが `TOBJ_NOAUTH` の場合、BEA Tuxedo ドメインの IIOP リスナ / ハンドラはその IIOP リスナ / ハンドラに送信されるユーザ名とクライアント・アプリケーション名に CORBA クライアント・アプリケーションを登録します。

BEA Tuxedo ソフトウェアでは、SecurityCurrent オブジェクトのプロパティとして PrincipalAuthenticator と Credentials だけがサポートされます。

クライアント・アプリケーションでの SecurityCurrent オブジェクトの使い方については、『BEA Tuxedo CORBA アプリケーションのセキュリティ機能』を参照してください。`SecurityLevel1::Current` インターフェイスと `SecurityLevel2::Current` インターフェイスについては、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

TransactionCurrent オブジェクト

TransactionCurrent オブジェクトは、CORBA のオブジェクト・トランザクション・サービスの BEA Tuxedo の実装です。TransactionCurrent オブジェクトは、CORBA クライアント・アプリケーションと CORBA サーバ・アプリケーション間の現行セッションのトランザクション・コンテキストを維持します。TransactionCurrent オブジェクトを使用すると、CORBA クライアント・アプリケーションは、トランザクションの開始と終了や、トランザクションのステータスの取得などのトランザクション・オペレーションを実行できます。

トランザクションは、インターフェイス単位で使用されます。アプリケーション設計者は、設計時に CORBA アプリケーション内のどのインターフェイスでトランザクションを処理するかを決定します。次に、各インターフェイスのトランザクション方針をインプリメンテーション・コンフィギュレーション・ファイル (ICF) に定義します。トランザクション方針は以下のとおりです。

- never

  インターフェイスはトランザクションに関与しません。このインターフェイス用に作成されるオブジェクトは、トランザクションに関与できません。BEA Tuxedo ソフトウェアは、この方針が設定されているインターフェイスがトランザクションに関与した場合に例外 (`INVALID_TRANSACTION`) を生成します。

- optional

  インターフェイスはトランザクションに関与できます。要求がトランザクションに関係する場合、オブジェクトはトランザクションに関与できます。

- always

  インターフェイスは常にトランザクションの一部である必要があります。インターフェイスがトランザクションの一部ではない場合、トランザクションは TP フレームワークによって自動的に開始されます。

- ignore

  インターフェイスはトランザクションに関与しません。インターフェイスはトランザクションの一部になることができますが、UBBCONFIG ファイルでこのインターフェイスに対して指定された AUTOTRAN 方針が無視されます。

CORBA クライアント・アプリケーションでの TransactionCurrent オブジェクトの使い方については、『BEA Tuxedo CORBA トランザクション』を参照してください。TransactionCurrent オブジェクトについては、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

NotificationService オブジェクトと Tobj_SimpleEventsService オブジェクト

NotificationService オブジェクトと Tobj_SimpleEventsService オブジェクトは、CORBA イベント・サービスへのアクセスを提供します。BEA Tuxedo 製品の CORBA 環境のイベント・サービスは、ATMI 環境の EventBroker のイベント・サービスとほぼ同じ機能を提供します。ただし、CORBA イベント・サービスは、プログラミング・モデルと、CORBA プログラマにとって自然なインターフェイスを提供します。

CORBA イベント・サービスは、イベント・ポスト・メッセージを受信し、それらをフィルタして、サブスクライバに配布します。ポスト元は、関心のあるイベントがいつ発生したかを検出し、それをイベント・サービスに報告 (ポスト) する CORBA アプリケーションです。サブスクライバは、関心のあるイベントがポストされたときに実行される通知アクションを要求する CORBA アプリケーションです。

CORBA イベント・サービスは、以下の 2 種類のインターフェイス・セットを提供します。

- NotificationService オブジェクトは、CORBA ベースのノーティフィケーション・サービス・インターフェイス (CosNotification サービス・インターフェイス) のサブセットを提供します。
- Tobj_SimpleEventsService オブジェクトは、使いやすいように設計された BEA 固有のインターフェイスを提供します。

どちらのインターフェイス・セットも、CORBA ノーティフィケーション・サービス仕様で定義される標準の構造化されたイベントを渡します。この 2 つのインターフェイス・セットは、相互に互換性があります。このため、NotificationService インターフェイスを使用してポストされたイベントを `Tobj_SimpleEventsService` インターフェイスでサブスクライブでき、その逆も可能です。

NotificationServer オブジェクトと Tobj_SimpleEventsService オブジェクトの使い方については、『BEA Tuxedo CORBA ノーティフィケーション・サービス』を参照してください。

NameService オブジェクト

NameService オブジェクトは、CORBA ネーム・サービスへのアクセスを提供します。CORBA ネーム・サービスを使用すると、CORBA サーバ・アプリケーションは論理名を使用してオブジェクト・リファレンスを宣言できます。CORBA クライアント・アプリケーションは、CORBA ネーム・サービスに名前のルックアップを依頼することによってオブジェクトをロケートできます。

CORBA ネーム・サービスは、以下のものを提供します。

- Object Management Group (OMG) のインターオペラブル・ネーム・サービス (INS) 仕様の実装
- オブジェクト・リファレンスを階層的な命名構造 (名前空間) にマッピングするためのアプリケーション・プログラミング・インターフェイス (API)
- バインディングを表示し、ネーミング・コンテキスト・オブジェクトとアプリケーション・オブジェクトを名前空間にバインドおよびアンバインドするためのコマンド

CORBA クライアント・アプリケーションでの NameService オブジェクトの使い方については、『BEA Tuxedo CORBA ネーム・サービス』を参照してください。

ActiveX クライアント・アプリケーションの概念

以下の節では、ActiveX クライアント・アプリケーションに固有の概念について説明します。

ActiveX とは

ActiveX は、ネットワーク環境下のソフトウェア・コンポーネントどうしがそれぞれの開発言語に関係なく対話できるようにするための Microsoft の技術です。ActiveX は、Component Object Model (COM) をベースに構築されており、Object Linking and Embedding (OLE) と統合されています。OLE は、ドキュメントの埋め込み用のアーキテクチャを提供します。オートメーションは COM の一部です。オートメーションを使用すると、Visual Basic、Delphi、PowerBuilder などのアプリケーションはオートメーション・オブジェクト、ActiveX コントロール、および ActiveX ドキュメントを処理できます。

BEA ActiveX Client は、BEA Tuxedo と COM オブジェクト・システム間の相互運用性を提供します。ActiveX Client は、BEA Tuxedo ドメインの CORBA オブジェクトのインターフェイスをオートメーション・オブジェクトのメソッドに変換します。

ビューとバインディング

ActiveX クライアント・アプリケーションは、CORBA インターフェイスのビューを使用します。ビューは、BEA Tuxedo ドメインの CORBA インターフェイスをオートメーション・オブジェクトとしてローカルに表します。CORBA オブジェクトの ActiveX ビュー (ActiveX ビュー) を使用するには、ActiveX のバインディングを作成する必要があります。バインディングは、CORBA オブジェクトと ActiveX のインターフェイスを定義します。CORBA オブジェクトのインターフェイスは、インターフェイス・リポジトリにロードされます。次に、BEA Application Builder を使用して、インターフェイスのオートメーション・バインディングを作成します。

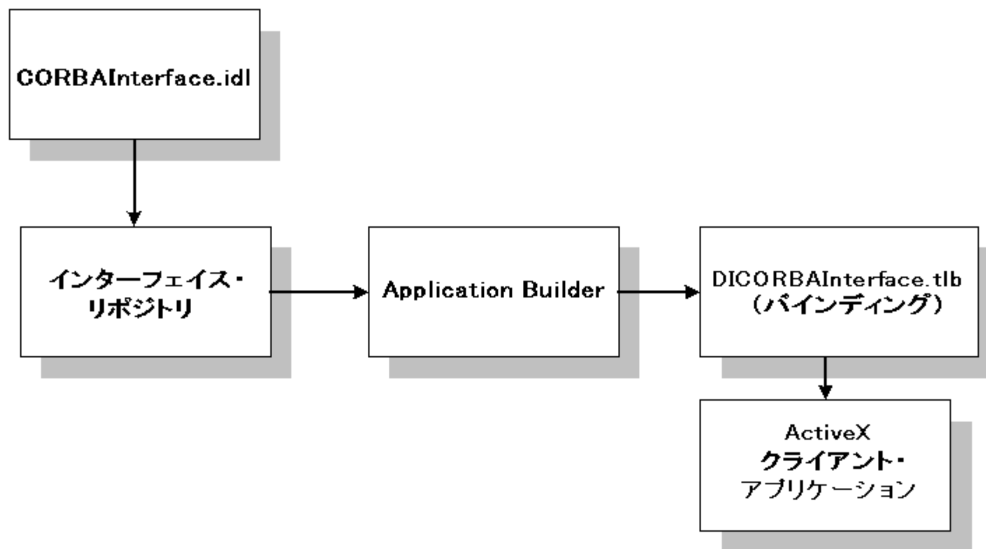
開発ツールの Application Builder は、ActiveX クライアント・アプリケーションの対話先となる BEA Tuxedo ドメイン内の CORBA オブジェクトを選択するためにクライアント開発ツール (Visual Basic など) と一緒に使用します。Application Builder とその機能については、Application Builder のグラフィックファイカル・ユーザ・インターフェイス (GUI) に統合されているオンライン・ヘルプを参照してください。

ActiveX クライアント・アプリケーションと生成されたバインディングの組み合わせにより、オブジェクトの ActiveX ビューが作成されます。

ActiveX クライアント・アプリケーションの作成方法と使い方の詳細については、「ActiveX クライアント・アプリケーションの作成」を参照してください。

図 1-6 に、ActiveX Client の機能を示します。

図 1-6　ActiveX Client の機能



ActiveX ビューの命名規則

命名規則では、CORBA インターフェイスを ActiveX にマッピングして型と変数名の不整合を回避するためのアルゴリズムが定義されます。また、命名規則では、特定のオブジェクトの使い方も指定されます。すべての ActiveX メソッドの名前は、DI で始まります。

ActiveX Client は、CORBA インターフェイスのオートメーション・バインディングを作成するときにこの命名規則を参照します。CORBA インターフェイス名が Account の場合、そのインターフェイスのオートメーション・バインディング名は DIAccount となります。

多くの場合、CORBA インターフェイス名はモジュールと呼ばれる入れ子になったレベルの中でスコープ指定されますが、ActiveX ではスコープ指定は存在しません。名前の不整合を避けるため、ActiveX Client は異なるスコープの名前をインターフェイスの先頭に追加して、CORBA インターフェイスを ActiveX にマッピングします。

たとえば、Account という名前の CORBA インターフェイスが、OMG IDL ファイルで次のように定義されているとします。

```
module University
{
   module Student
     {
       interface Account
           {//Account インターフェイスのオペレーションと属性
           };
     };

};
```

CORBA では、このインターフェイスの名前は University::Student::Account です。ActiveX Client は、この名前を ActiveX 用に DIUniversity_Student_Account に変換します。

2 CORBA クライアント・アプリケーションの作成

ここでは、次の内容について説明します。

- CORBA C++ クライアント・アプリケーションの開発プロセスの概要
- CORBA Java クライアント・アプリケーションの開発プロセスの概要
- ステップ 1: OMG IDL ファイルの取得
- ステップ 2: 呼び出し方式の選択
- ステップ 3: OMG IDL ファイルのコンパイル
- ステップ 4: CORBA クライアント・アプリケーションの記述
- ステップ 5: CORBA クライアント・アプリケーションのビルド
- クライアント・アプリケーションとして動作するサーバ・アプリケーション
- Java2 アプレットの使い方

CORBA C++ クライアント・アプリケーションの開発プロセスの概要

CORBA C++ クライアント・アプリケーションの作成手順は次のとおりです。

手順	説明
1	CORBA C++ クライアント・アプリケーションによって使用される CORBA インターフェイスの OMG IDL ファイルを取得します。
2	呼び出し方式を選択します。
3	IDL コンパイラを使用して OMG IDL ファイルをコンパイルします。クライアント・スタブは、OMG IDL のコンパイルによって生成されます。
4	CORBA C++ クライアント・アプリケーションを記述します。このトピックでは、基本的なクライアント・アプリケーションの作成方法について説明します。
5	CORBA C++ クライアント・アプリケーションをビルドします。

このプロセスの各手順については、以降の節で詳しく説明します。

CORBA C++ クライアント・アプリケーションの BEA Tuxedo 開発環境には、以下のものが含まれます。

- `idl` コマンド。OMG IDL ファイルをコンパイルし、CORBA インターフェイスで必要なクライアント・スタブを生成します。
- `buildobjclient` コマンド。CORBA C++ クライアント・アプリケーションの実行可能ファイルをビルドします。
- C++ 環境オブジェクト。BEA Tuxedo ドメインの CORBA オブジェクト、およびその CORBA オブジェクトによって提供されるサービスへのアクセスを提供します。

CORBA Java クライアント・アプリケーションの開発プロセスの概要

BEA Tuxedo ソフトウェアは､Sun Microsystem 社の Java Development Kit (JDK) Java クライアントとの相互運用性をサポートしています。

注記　サポートされるソフトウェアのバージョンについては、『BEA Tuxedo システムのインストール』を参照してください。

CORBA Java クライアント・アプリケーションの作成手順は次のとおりです。

手順	説明
1	CORBA Java クライアント・アプリケーションによって使用される CORBA インターフェイスの OMG IDL ファイルを取得します。
2	呼び出し方式を選択します。
3	使用する CORBA Java Object Request Broker (ORB) に用意されている開発ツールを使用して、OMG IDL ファイルをコンパイルし、クライアント・スタブを生成します。
4	CORBA Java クライアント・アプリケーションを記述します。このトピックでは、基本的なクライアント・アプリケーションの作成方法について説明します。
5	CORBA Java クライアント・アプリケーションをビルドします。

このプロセスの各手順については、以降の節で詳しく説明します。

OMG IDL ファイルのコンパイル、クライアント・スタブの生成、および CORBA Java クライアント・アプリケーションの実行可能ファイルのビルドを行うには、使用する CORBA Java ORB 製品の開発ツールを使用する必要がありま

す。また、BEA Tuxedo ドメインの CORBA オブジェクト、およびその CORBA オブジェクトによって提供されるサービスへのアクセスを提供する Java 環境オブジェクトを使用します。

ステップ 1: OMG IDL ファイルの取得

一般に、使用可能なインターフェイスとオペレーション用の OMG IDL ファイルは、アプリケーション設計者からクライアント・プログラマに提供されます。この節では、Basic サンプル・アプリケーション用の OMG IDL を示します。リスト 2-1 に、`univb.idl` ファイルを示します。このファイルには、以下のインターフェイスが定義されています。

インターフェイス	説明	オペレーション
`Registrar`	コース・データベースからコース情報を取得します。	`get_courses_synopsis()` `get_courses_details()`
`RegistrarFactory`	Registrar オブジェクトへのオブジェクト・リファレンスを作成します。	`find_registrar()`
`CourseSynopsisEnumerator`	コース・データベースから情報のサブセットを取得し、そのサブセットの一部を CORBA クライアント・アプリケーションに繰り返し返します。	`get_next_n()` `destroy()`

リスト 2-1 Basic サンプル・アプリケーション用の OMG IDL ファイル

```
#pragma prefix "beasys.com"
module UniversityB
{
   typedef unsigned long CourseNumber;
   typedef sequence<CourseNumber> CourseNumberList;
```

```
struct CourseSynopsis
  {
     CourseNumber   course_number;
     string         title;
  };

typedef sequence<CourseSynopsis> CourseSynopsisList;
interface CourseSynopsisEnumerator
{
     CourseSynopsisList get_next_n(
         in  unsigned long number_to_get,
         out unsigned long number_remaining
};
      void destroy();
};

typedef unsigned short Days;
const Days MONDAY    =  1;
const Days TUESDAY   =  2;
const Days WEDNESDAY =  4;
const Days THURSDAY  =  8;
const Days FRIDAY    = 16;

struct ClassSchedule
{
    Days           class_days; // 曜日のビットマスク
    unsigned short start_hour; // 軍用時間による開始時刻
    unsigned short duration;   // 分数
};

struct CourseDetails
{
    CourseNumber   course_number;
    double         cost;
    unsigned short number_of_credits;
    ClassSchedule  class_schedule;
    unsigned short number_of_seats;
    string         title;
    string         professor;
    string         description;
 };

typedef sequence<CourseDetails> CourseDetailsList;

interface Registrar
{
    CourseSynopsisList
    get_courses_synopsis(
    in string                    search_criteria,
    in unsigned long             number_to_get, // 0 = all
    out unsigned long            number_remaining,
```

```
        out CourseSynopsisEnumerator rest
    );

        CourseDetailsList get_courses_details(in CourseNumberList
        courses);

    interface RegistrarFactory
    {
         Registrar find_registrar(
    );

    };
};
```

ステップ 2: 呼び出し方式の選択

CORBA クライアント・アプリケーションでの要求で使用する呼び出し方式 (静的または動的) を選択します。CORBA クライアント・アプリケーションでは、どちらの呼び出し方式も使用できます。

静的起動と動的起動の概要については、「静的起動と動的起動」を参照してください。

以後、このトピックでは、CORBA クライアント・アプリケーションで静的起動を使用することを選択したものとして説明を進めます。動的起動を選択した場合は、「動的起動インターフェイスの使い方」を参照してください。

ステップ 3: OMG IDL ファイルのコンパイル

CORBA C++ クライアント・アプリケーションを作成する場合は、`idl` コマンドを使用して OMG IDL ファイルをコンパイルし、インターフェイスで必要なファイルを生成します。次に、`idl` コマンドの構文を示します。

```
idl idlfilename(s)
```

IDL コンパイラは、クライアント・スタブ (*idlfilename*_c.cpp) と、C++ プログラミング言語からクライアント・スタブを使用するために必要なすべてを記述したヘッダ・ファイル (*idlfilename*_c.h) を生成します。これらのファイルは、CORBA クライアント・アプリケーションにリンクする必要があります。

さらに、IDL コンパイラは CORBA オブジェクトのオペレーションのシグニチャを含んだスケルトンを生成します。生成されるスケルトン情報は、*idlfilename*_s.cpp ファイルと *idlfilename*_s.h ファイルに格納されます。CORBA クライアント・アプリケーションの開発中、この情報はサーバ・ヘッダ・ファイルとスケルトン・ファイルを参照するのに役立ちます。

注記　生成されるクライアント・スタブとスケルトンは変更しないでください。

idl コマンドとオプションの詳細については、『BEA Tuxedo コマンド・リファレンス』を参照してください。

CORBA Java クライアント・アプリケーションを作成する場合は、以下のものを使用します。

- JDK バージョン 1.2 を使用する場合、idltojava コマンドを使用して OMG IDL ファイルをコンパイルできます。idltojava コマンドの詳細については、JDK バージョン 1.2 のマニュアルを参照してください。
- Netscape バージョン 3.0 と Java Development Kit (JDK) バージョン 1.1.5 を使用する場合、その製品の IDL コンパイラを使用して OMG IDL をコンパイルする必要があります。

idltojava コマンドまたは IDL コンパイラは、以下のものを生成します。

- 各インターフェイスのクライアント・スタブ (_*interface*Stub.java)
- Java プログラミング言語からクライアント・スタブを使用するために必要なすべてを記述した CORBA ヘルパ・クラス (*interface*Helper.java) と CORBA ホルダ・クラス (*interface*Holder.java)

OMG IDL で定義された各例外は、例外クラスとそのヘルパ・クラスおよびホルダ・クラスを定義します。コンパイルされた .class ファイルは、CORBA クライアント・アプリケーションの CLASSPATH に存在する必要があります。

また、idltojava コマンドまたは IDL コンパイラは CORBA オブジェクトのオペレーションのシグニチャを含んだスケルトンを生成します。生成されたスケルトン情報は、_*interface*ImplBase ファイルに格納されます。

ステップ 4: CORBA クライアント・アプリケーションの記述

CORBA サーバ・アプリケーションとのセッションに参加するには、CORBA クライアント・アプリケーションは CORBA オブジェクトのオブジェクト・リファレンスを取得してそのオブジェクトのオペレーションを呼び出すことができなければなりません。これを実現するには、CORBA クライアント・アプリケーションは以下のことを行う必要があります。

1. BEA Tuxedo ORB の初期化
2. BEA Tuxedo ドメインとの通信の確立
3. FactoryFinder オブジェクトへの初期リファレンスの解決
4. ファクトリの使用による目的の CORBA オブジェクトのオブジェクト・リファレンスの取得
5. CORBA オブジェクトのオペレーションの呼び出し

以降の節では、Basic サンプル・アプリケーションのクライアント・アプリケーションの一部を使用して、これらのステップについて説明します。Basic サンプル・アプリケーションについては、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。Basic サンプル・アプリケーションは、BEA Tuxedo ソフトウェア・キットの次のディレクトリに格納されています。

```
drive:\tuxdir\samples\corba\university\basic
```

ORB の初期化

すべての CORBA クライアント・アプリケーションは、最初に ORB を初期化する必要があります。

CORBA C++ クライアント・アプリケーションから ORB を初期化するには、次のコードを使用します。

C++

```
CORBA::ORB_var orb=CORBA::ORB_init(argc, argv, ORBid);
```

一般に、ORBid は指定されず、初期化中に指定されたデフォルトの ORBid が使用されます。ただし、CORBA クライアント・アプリケーションが実行されているマシンで CORBA サーバ・アプリケーションも実行されており、CORBA クライアント・アプリケーションが別の BEA Tuxedo ドメインのサーバ・アプリケーションにアクセスする場合は、デフォルト ORBid を上書きする必要があります。そのためには、ORBid を `BEA_IIOP` としてハードコードするか、コマンド・ラインで ORBid を `_ORBid BEA_IIOP` として受け渡します。

CORBA Java クライアント・アプリケーションから ORB を初期化するには、次のコードを使用します。

Java アプリケーション

```
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init (args,props);
```

CORBA Java クライアント・アプレットから ORB を初期化するには、次のコードを使用します。

Java アプレット

```
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init (this,null);
```

ここで *`this`* は Java アプレットの名前です。

BEA Tuxedo ドメインとの通信の確立

CORBA クライアント・アプリケーションは、Bootstrap オブジェクトを作成します。IIOP リスナ / ハンドラのリストは、パラメータとして提供されるか、`TOBJADDR` Java プロパティまたはアプレット・プロパティを介して提供されます。1 つの IIOP リスナ / ハンドラは、次のように指定されます。

```
//host:port
```

IIOP リスナ / ハンドラが `TOBJADDR` を介して提供される場合、コンストラクタの 2 番目の引数は null にできます。

IIOP リスナ / ハンドラのホストとポートの組み合わせは、`UBBCONFIG` ファイルに定義されます。Bootstrap オブジェクトに対して指定されるホストとポートの組み合わせは、BEA Tuxedo ドメインの `UBBCONFIG` ファイルの ISL パラメータ

と完全に一致する必要があります。ホストとポートの組み合わせの形式のほかに、大文字、小文字も一致する必要があります。アドレスが一致しない場合、Bootstrap オブジェクトの呼び出しは失敗し、次のメッセージがログ・ファイルに記録されます。

```
Error: Unofficial connection from client at <tcp/ip
address>/<portnumber>
```

たとえば、`UBBCONFIG` ファイルの ISL パラメータでネットワーク・アドレスが `//TRIXIE::3500` として指定されている場合、Bootstrap オブジェクトで `//192.12.4.6.:3500` または `//trixie:3500` と指定すると接続が失敗します。

UNIX システムで大文字、小文字の区別を調べるには、ホスト・システムで `uname -n` コマンドを使用します。Window 2000 で大文字、小文字の区別を調べるには、コントロール・パネルの [ネットワーク] を使用します。

次の C++ と Java の例では、Bootstrap オブジェクトの使い方を示します。

C++

```
Tobj_Bootstrap* bootstrap = new Tobj_Bootstrap(orb, "//host:port");
```

Java

Java クライアント・アプリケーションの有効な IIOP リスナ / ハンドラを取得するには、次のコマンドを使用します。

```
Properties prop = Tobj_Bootstrap.getRemoteProperties();
```

注記　BEA Tuxedo 8.0 は、リモート Java クライアント・アプリケーションだけをサポートします。`getNativeProperties()` メソッドを使用する場合、ORB を初期化するときにエラーが発生します。これは、BEA Tuxedo 8.0 にはリモート Java クライアント・アプリケーション用の JAR ファイルしか用意されていないからです。

次のコマンドで、IIOP リスナ / ハンドラを使用します。

```
Tobj_Bootstrap bootstrap = new Tobj_Bootstrap(orb, "//host:port");
```

Java アプレット

```
Tobj_Bootstrap bootstrap = new Tobj_Bootstrap(orb, "//host:port",
this);
```

ここで *this* は Java アプレットの名前です。

BEA Tuxedo ドメインには、複数の IIOP リスナ / ハンドラが存在できます。複数の IIOP リスナ / ハンドラを使用して BEA Tuxedo ドメインにアクセスする場合、Host:Port の組み合わせのリストを Bootstrap オブジェクトに提供します。Bootstrap コマンドの 2 番目のパラメータが空の文字列の場合、Bootstrap オブジェクトは BEA Tuxedo ドメインに接続するまでこのリストを参照します。また、IIOP リスナ / ハンドラのリストは、TOBJADDR に指定することもできます。

複数の BEA Tuxedo ドメインにアクセスする場合、アクセスする BEA Tuxedo ドメインごとに Bootstrap オブジェクトを作成する必要があります。

注記 サードパーティ・クライアント ORB は、CORBA インターオペラブル・ネーミング・サービス (INS) メカニズムを使用して BEA Tuxedo ドメインとそのサービスにアクセスできます。CORBA INS を使用すると、サードパーティ・クライアント ORB は、自身の resolve_initial_references() 関数を使用して BEA Tuxedo ドメインによって提供される CORBA サービスにアクセスし、標準 OMG IDL から生成されたスタブを使用してドメインから返されたインスタンスを処理できます。インターオペラブル・ネーミング・サービスの使い方については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

FactoryFinder オブジェクトへの初期リファレンスの解決

CORBA クライアント・アプリケーションは、自身にサービスを提供する環境オブジェクトへの初期リファレンスを取得する必要があります。Bootstrap オブジェクトの resolve_initial_references オペレーションを呼び出すと、FactoryFinder、InterfaceRepository、SecurityCurrent、TransactionCurrent、NotificationService、Tobj_SimpleEventsService、および NameService 環境オブジェクトへのリファレンスを取得できます。このオペレーションに受け渡す引数は、目的のオブジェクト・リファレンスの名前を含む文字列です。CORBA クライアント・アプリケーションで使用する環境オブジェクトの初期リファレンスだけを取得する必要があります。

次の C++ と Java の例では、Bootstrap オブジェクトを使用して FactoryFinder オブジェクトへの初期リファレンスを解決する方法を示します。

C++

```
//Factory Finder の解決
CORBA::Object_var var_factory_finder_oref = bootstrap.resolve_initial_references
       ("FactoryFinder");
Tobj::FactoryFinder_var var_factory_finder_ref = Tobj::FactoryFinder::_narrow
      (factory_finder_oref.in());
```

Java

```
//Factory Finder の解決
org.omg.CORBA.Object off = bootstrap.resolve_initial_references
      ("FactoryFinder");
FactoryFinder ff=FactoryFinderHelper.narrow(off);
```

FactoryFinder オブジェクトを使用したファクトリの取得

CORBA クライアント・アプリケーションは、CORBA オブジェクトへのリファレンスをファクトリから取得します。ファクトリは、別の CORBA オブジェクトへのリファレンスを返し、自身をファクトリとして登録する任意の CORBA オブジェクトです。CORBA クライアント・アプリケーションは、ファクトリのオペレーションを呼び出して特定のタイプの CORBA オブジェクトへのオブジェクト・リファレンスを取得します。ファクトリを使用するには、CORBA クライアント・アプリケーションは必要なファクトリを検索できなければなりません。FactoryFinder オブジェクトは、そのために役立ちます。FactoryFinder オブジェクトの機能の登録については、「CORBA クライアント・アプリケーションの開発概念」を参照してください。

FactoryFinder オブジェクトには以下のメソッドがあります。

- `find_factories()`

  入力キーに完全に一致するファクトリのシーケンスを返します。

- `find_one_factory()`

  入力キーに一致する 1 つのファクトリを返します。

- `find_factories_by_id()`

  名前コンポーネントの ID フィールドが入力引数に一致するファクトリのシーケンスを返します。

- `find_one_factory_by_id()`

  ファクトリの CORBA 名前コンポーネントの ID フィールドが入力引数に一致する 1 つのファクトリを返します。

次の C++ と Java の例では、FactoryFinder `find_one_factory_by_id` メソッドを使用して、Basic サンプル・アプリケーションの CORBA クライアント・アプリケーションで使用される Registrar オブジェクトのファクトリを取得する方法を示します。

C++

```
CORBA::Object_var var_registrar_factory_oref = var_factory_finder_ref->
     find_one_factory_by_id(UniversityB::_tc_RegistrarFactory->id()
);
UniversityB::RegistrarFactory_var var_RegistrarFactory_ref =
     UniversityB::RegistrarFactory::_narrow(
          var_RegistrarFactory_oref.in()
     );
```

Java

```
org.omg.CORBA.Object of = FactoryFinder.find_one_factory_by_id
    (UniversityB.RegistrarFactoryHelper.id());
UniversityB.RegistrarFactory F = UniversityB.RegistrarFactoryHelper.narrow(of);
```

ファクトリを使用した CORBA オブジェクトの取得

CORBA クライアント・アプリケーションは、ファクトリを呼び出して CORBA オブジェクトへのオブジェクト・リファレンスを取得します。次に、CORBA クライアント・アプリケーションは、ファクトリへのポインタとオペレーションで必要な引数を受け渡すことによって、CORBA オブジェクトのオペレーションを呼び出します。

次の C++ と Java の例では、Registrar オブジェクトのファクトリを取得して、そのオブジェクトの `get_courses_details()` メソッドを呼び出す方法を示します。

C++

```
UniversityB::Registrar_var var_Registrar = var_RegistrarFactory->
             find_Registrar();
```

```
UniversityB::CourseDetailsList_var course_details_list = Registrar_oref->
            get_course_details(CourseNumberList);
```

Java

```
UniversityB.Registrar gRegistrarObjRef = F.find_registrar();
gRegistrarObjRef.get_course_details(selected_course_numbers);
```

ステップ 5: CORBA クライアント・アプリケーションのビルド

CORBA クライアント・アプリケーションの開発の最終ステップは、クライアント・アプリケーションの実行可能ファイルを生成することです。そのためには、コードをコンパイルしてクライアント・スタブに対してリンクする必要があります。

CORBA C++ クライアント・アプリケーションを作成する場合、`buildobjclient` コマンドを使用して CORBA クライアント・アプリケーションの実行可能ファイルを生成します。このコマンドは、静的起動を使用するインターフェイスのクライアント・スタブ、関連付けられるヘッダ・ファイル、および標準の BEA Tuxedo ライブラリを組み合わせてクライアント実行可能ファイルを生成します。`buildobjclient` コマンドの構文については、『BEA Tuxedo コマンド・リファレンス』を参照してください。

CORBA Java クライアント・アプリケーションをコンパイルする場合、BEA Tuxedo 環境オブジェクトの Java クラスを含んだ Java ARchive (JAR) ファイルを `CLASSPATH` に追加する必要があります。JDK バージョン 1.2 を使用する場合、`m3envobj.jar` ファイルは次のディレクトリに格納されています。

```
tuxdir\udataobj\java\jdk
```

クライアント・アプリケーションとして動作するサーバ・アプリケーション

CORBA クライアント・アプリケーションからの要求を処理するには、CORBA サーバ・アプリケーションは別のサーバ・アプリケーションに処理を要求する必要が生じる場合があります。このような場合、CORBA サーバ・アプリケーションは CORBA クライアント・アプリケーションとして動作します。

CORBA クライアント・アプリケーションとして動作するには、CORBA サーバ・アプリケーションは現在の BEA Tuxedo ドメインの Bootstrap オブジェクトを取得する必要があります。CORBA サーバ・アプリケーションの Bootstrap オブジェクトは、`TP::Bootstrap` (CORBA C++ クライアント・アプリケーションの場合) または `TP.Bootstrap` (CORBA Java クライアント・アプリケーションの場合) を介して既に使用可能になっています。CORBA サーバ・アプリケーションは、FactoryFinder オブジェクトを使用して、CORBA クライアント・アプリケーションの要求を満たす CORBA オブジェクトのファクトリを検索します。

Java2 アプレットの使い方

BEA Tuxedo の CORBA 環境では、Java2 アプレットをクライアントとして使用できます。Java2 アプレットを実行するには、Sun Microsystems, Inc. から Java Plug-In 製品をインストールする必要があります。Java Plug-In は、Sun の Java 仮想マシン (JVM) を使用して HTML ページ内で Java アプレットを実行します。

Sun の Web サイトから Java Plug-in キットをダウンロードする前に、Java Plug-In が既にマシンにインストールされているかどうかを確認してください。

Netscape Navigator

Netscape Navigator では、ブラウザ・ウィンドウの [ヘルプ] メニューから [Plug-ins について] オプションを選択します。Java Plug-In がインストールされている場合は、次の文字列が表示されます。

```
application/x-java-applet;version 1.2
```

Internet Explorer

Windows 2000 の [スタート] メニューから、[プログラム] オプションを選択します。Java Plug-In がインストールされている場合は、[Java Plug-In コントロール パネル] オプションが表示されます。

Java Plug-In がインストールされていない場合は、JDK1.2 Plug-in (`jre12-win32.exe`) と HTML Converter ツール (`htmlconv12.zip`) をダウンロードおよびインストールする必要があります。どちらの製品も、`java.sun.com/products/plugin` から入手できます。

また、`java.sun.com/products/plugin/1.2/docs/tags.html` の 「Java Plug-In HTML Specification」を通読する必要があります。この仕様には、Web ページ作成者がブラウザのデフォルト Java 実行時環境ではなく Java Plug-in を使用して既存の JDK 1.2 アプレットを実行するために必要な、既存の HTML コードの変更内容が説明されています。

Java アプレットを記述します。Java アプレットから ORB を初期化するには、次のコードを使用します。

```
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init (this,null);
```

Internet Explorer ブラウザまたは Netscape Navigator ブラウザがアプレットの HTML ページをブラウズしているときに Java Plug-in が自動的に起動するようにするには、HTML 仕様の `OBJECT` タグと `EMBED` タグを使用します。HTML Converter ツールを使用してアプレットを HTML に変換した場合、これらのタグは自動的に挿入されます。`OBJECT` タグと `EMBED` タグの使い方については、`java.sun.com/products/plugin/1.2/docs/tags.html` を参照してください。

3 ActiveX クライアント・アプリケーションの作成

ここでは、次の内容について説明します。

- ActiveX クライアント・アプリケーションの開発プロセスの概要
- BEA Application Builder
- ステップ 1: オートメーション環境オブジェクトのインターフェイス・リポジトリへのロード
- ステップ 2: CORBA インターフェイスのインターフェイス・リポジトリへのロード
- ステップ 3: インターフェイス・リポジトリのサーバ・アプリケーションの起動
- ステップ 4: CORBA インターフェイスの ActiveX バインディングの作成
- ステップ 5: ActiveX バインディングのタイプ・ライブラリのロード
- ステップ 6: ActiveX クライアント・アプリケーションの記述
- ステップ 7: ActiveX クライアント・アプリケーションのデプロイ

ActiveX クライアント・アプリケーションを開発する前に理解しておく必要がある概念については、「ActiveX クライアント・アプリケーションの概念」を参照してください。

ActiveX クライアント・アプリケーションの開発プロセスの概要

ActiveX クライアント・アプリケーションの作成手順は次のとおりです。

手順	説明
1	オートメーション環境オブジェクトをインターフェイス・リポジトリにロードします。
2	ActiveX クライアント・アプリケーションからアクセスする CORBA インターフェイスがインターフェイス・リポジトリにロードされていることを確認します。必要に応じて、CORBA インターフェイス用の Object Management Group (OMG) インターフェイス定義言語 (IDL) をインターフェイス・リポジトリにロードします。
3	インターフェイス・リポジトリのサーバ・プロセスを起動します。
4	BEA Application Builder を使用して、CORBA オブジェクトのインターフェイス用の ActiveX バインディングを作成します。
5	ActiveX バインディング用のタイプ・ライブラリを開発ツールにロードします。
6	ActiveX クライアント・アプリケーションを記述します。このトピックでは、基本的なクライアント・アプリケーションの作成方法について説明します。
7	ActiveX クライアント・アプリケーションのデプロイメント・パッケージを作成します。

このプロセスの各手順については、以降の節で詳しく説明します。

ActiveX クライアント・アプリケーションの BEA Tuxedo 開発環境には、以下のものが含まれます。

- `idl2ir` コマンド。OMG IDL に定義されているインターフェイス定義をインターフェイス・リポジトリにロードします。
- BEA Application Builder。CORBA オブジェクトのインターフェイス用の ActiveX バインディングを作成し、そのインターフェイス用のデプロイメント・パッケージを作成します。
- オートメーション環境オブジェクト。BEA Tuxedo ドメインの CORBA オブジェクトの ActiveX ビュー (ActiveX ビューと呼ばれる)、およびその ActiveX ビューによって提供されるサービスへのアクセスを提供します。

BEA Application Builder

BEA Application Builder は、CORBA オブジェクトの ActiveX ビューを作成する開発ツールです。BEA Application Builder は、BEA ActiveX Client への主要なユーザ・インターフェイスです。Application Builder を使用すると、デスクトップ・アプリケーションで使用可能な CORBA オブジェクトを選択し、その CORBA オブジェクトの ActiveX ビューと、その ActiveX ビューをクライアント・マシンにデプロイするためのパッケージを作成できます。

ActiveX ビューを使用するには、CORBA オブジェクトのインターフェイスをインターフェイス・リポジトリにロードします。次に、CORBA インターフェイスの ActiveX バインディングを作成します。バインディングは、CORBA オブジェクトと ActiveX のインターフェイスを定義します。ActiveX クライアント・アプリケーションと生成されたバインディングの組み合わせにより、オブジェクトの ActiveX ビューが作成されます。

BEA Application Builder とその機能については、Application Builder のグラフィカル・ユーザ・インターフェイス (GUI) に統合されているオンライン・ヘルプを参照してください。

ステップ 1: オートメーション環境オブジェクトのインターフェイス・リポジトリへのロード

オートメーション環境オブジェクトをインターフェイス・リポジトリにロードして、ActiveX クライアント・アプリケーションがそれらのオブジェクトのインターフェイス定義を使用できるようにします。MS-DOS プロンプトで、次のコマンドを入力して OMG IDL ファイル (`TOBJIN.idl`) をインターフェイス・リポジトリにロードします。

```
prompt> idl2ir -D _TOBJ -I drive:\tuxdir\include drive:\tuxdir\include\tobjin.idl
```

ステップ 2: CORBA インターフェイスのインターフェイス・リポジトリへのロード

CORBA オブジェクトの ActiveX ビューを作成する前に、CORBA オブジェクトのインターフェイスをインターフェイス・リポジトリにロードしておく必要があります。CORBA オブジェクトのインターフェイスがインターフェイス・リポジトリにロードされていない場合、BEA Application Builder の [Services] ウィンドウにそれらが表示されません。目的の CORBA インターフェイスが [Services] ウィンドウに表示されない場合は、`idl2ir` コマンドを使用して、CORBA を定義する OMG IDL をインターフェイス・リポジトリにロードします。`idl2ir` コマンドの構文は次のとおりです。

```
idl2ir [repositoryfile.idl] file.idl
```

オプション	説明
repositoryfile	CORBA インターフェイスの OMG IDL ファイルを指定されたインターフェイス・リポジトリにロードします。ActiveX クライアント・アプリケーションがアクセスする BEA Tuxedo ドメインのインターフェイス・リポジトリの名前を指定します。
file.idl	CORBA インターフェイスの定義が記述されている OMG IDL ファイルを指定します。

`idl2ir` コマンドの詳細については、『BEA Tuxedo コマンド・リファレンス』を参照してください。

「CORBA クライアント・アプリケーションの作成」には、すべての CORBA University サンプル・アプリケーションの開始点となるサンプル OMG IDL ファイルが示されています。この OMG IDL ファイルに基づいて、次の CORBA インターフェイスが BEA Application Builder ウィンドウに表示されます。

- `RegistrarFactory`
- `Registrar`
- `CourseSynopsisEnumerator`

University サンプル・アプリケーションの詳細については、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。

ステップ 3: インターフェイス・リポジトリのサーバ・アプリケーションの起動

ActiveX クライアント・アプリケーションは、実行時にインターフェイス・リポジトリから CORBA オブジェクトのインターフェイス定義を読み取り、それらをオートメーション・オブジェクトに変換します。このため、インターフェイ

ス・リポジトリのサーバ・プロセスを起動して、インターフェイス定義を使用可能にする必要があります。インターフェイス・リポジトリのサーバ・プロセスを起動するには、UBBCONFIG ファイルを使用します。

注記 システム管理者が既にこのステップを実行している場合もあります。

BEA Tuxedo ドメインの UBBCONFIG ファイルで、インターフェイス・リポジトリのサーバ・アプリケーション、TMIFRSVR が起動しているかどうかをチェックします。UBBCONFIG ファイルには、以下のエントリが存在します。

```
TMIFRSVR
     SRVGRP = SYS_GRP
     SRVID = 6
     RESTART = Y
     MAXGEN = 5
     GRACE = 3600
```

また、IIOP リスナ / ハンドラを起動する ISL パラメータが指定されていることを確認します。UBBCONFIG ファイルには、以下のエントリが存在します。

```
ISL
     SRVGRP = SYS_GRP
     SRVID  = 5
     CLOPT  = "-A -- -n //TRIXIE:2500"
```

ここで、TRIXIE はホスト (サーバ) システムの名前で、2500 はポート番号です。

サーバ・アプリケーションの起動と ISL パラメータの指定については、『BEA Tuxedo アプリケーションの設定』を参照してください。

ステップ 4: CORBA インターフェイスの ActiveX バインディングの作成

ActiveX クライアント・アプリケーションが CORBA オブジェクトにアクセスするには、CORBA オブジェクトのインターフェイスの ActiveX バインディングを生成する必要があります。CORBA インターフェイス用の ActiveX バインディングを作成するには、BEA Application Builder を使用します。

CORBA インターフェイス用の ActiveX バインディングを作成するには、次の手順に従います。

1. [BEA Tuxedo] プログラム・グループの [BEA Application Builder] オプションをクリックします。

   [Domain] の [Logon] ウィンドウが表示されます。

2. [Logon] ウィンドウで、`UBBCONFIG` ファイルの `ISL` パラメータで指定したホスト名とポート番号を入力します。`UBBCONFIG` ファイルで使用されている大文字、小文字をそのまま入力する必要があります。

   BEA Application Builder の [Logon] ウィンドウが表示されます。

3. 対象の CORBA インターフェイスを [Services] ウィンドウで選択して [Workstation Views] ウィンドウにドラッグするか、またはその CORBA インターフェイスを [Services] ウィンドウから切り取って [Workstation Views] ウィンドウに貼り付けます。

   BEA Application Builder は、以下の処理を行います。

   - タイプ・ライブラリを作成します。デフォルトでは、タイプ・ライブラリは `\tuxdir\TypeLibraries` に格納されます。

     タイプ・ライブラリの名前は、`DI`*`modulename_interfacename`*`.tlb` です。

   - CORBA インターフェイス用の Windows システム・レジストリ・エントリ (各オブジェクト型の一意のプログラム ID を含む) を作成します。

これで、ActiveX クライアント・アプリケーションから ActiveX ビューを使用できます。

Application Builder の機能の詳細については、BEA Application Builder のグラフィカル・ユーザ・インターフェイス (GUI) に統合されているオンライン・ヘルプを参照してください。

ステップ 5: ActiveX バインディングのタイプ・ライブラリのロード

ActiveX クライアント・アプリケーションを記述する前に、CORBA インターフェイスの ActiveX バインディングが定義されているタイプ・ライブラリを開発ツールにロードしておく必要があります。タイプ・ライブラリをロードするには、使用する開発ツールの説明に従います。

たとえば Visual Basic の場合、使用可能なタイプ・ライブラリのリストを取得するには、[プロジェクト] メニューの [参照設定] オプションを選択します。次に、そのリストから目的のタイプ・ライブラリを選択します。

デフォルトでは、BEA Application Builder は生成されたすべてのタイプ・ライブラリを `\tuxdir\TypeLibraries` に格納します。CORBA インターフェイスの ActiveX バインディング用のタイプ・ライブラリは、次の形式を取ります。

```
DImodulename_interfacename.tlb
```

ステップ 6: ActiveX クライアント・アプリケーションの記述

ActiveX クライアント・アプリケーションは、以下の処理を行う必要があります。

1. オートメーション環境オブジェクト、ActiveX ビューのファクトリ、および ActiveX ビューの宣言のインクルード
2. BEA Tuxedo ドメインとの通信の確立
3. Bootstrap オブジェクトの使用による FactoryFinder オブジェクトへのリファレンスの取得
4. ファクトリの使用による ActiveX ビューへのオブジェクト・リファレンスの取得
5. ActiveX ビューのオペレーションの呼び出し

6. ActiveX クライアント・アプリケーションのデプロイ

以下の節では、Basic サンプル・アプリケーションの ActiveX クライアント・アプリケーションの一部を使用して、これらのステップについて説明します。Basic サンプル・アプリケーションについては、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。Basic サンプル・アプリケーションは、BEA Tuxedo ソフトウェア・キットの次のディレクトリに格納されています。

drive:\tuxdir\samples\corba\university\basic

オートメーション環境オブジェクト、ファクトリ、および CORBA オブジェクトの ActiveX ビューの宣言のインクルード

実行時のエラーを防ぐため、以下のオブジェクト型を宣言する必要があります。

- オートメーション環境オブジェクト
- CORBA オブジェクトの ActiveX ビューを作成するファクトリ
- ActiveX ビュー

次の例は、Bootstrap オブジェクトと FactoryFinder オブジェクト、Registrar オブジェクトの ActiveX ビュー用のファクトリ、および Registrar オブジェクトの ActiveX ビューを宣言する Visual Basic コードです。

```
\\Declare Bootstrap object\\
    Public objBootstrap As DITobj_Bootstrap
\\Declare FactoryFinder object\\
    Public objFactoryFinder As DITobj_FactoryFinder
\\Declare factory object for Registrar Object\\
    Public objRegistrarFactory As DIUniversityB_RegistrarFactory
\\Declare the ActiveX view of the Registrar object\\
    Public objRegistrar As DIUniversityB_Registrar
```

BEA Tuxedo ドメインとの通信の確立

ActiveX クライアント・アプリケーションを記述する場合、以下の 2 つのステップで BEA Tuxedo ドメインとの通信を確立します。

1. Bootstrap オブジェクトを作成します。
2. Bootstrap オブジェクトを初期化します。

次の Visual Basic サンプルは、`CreateObject` オペレーションを使用して Bootstrap オブジェクトを作成する方法を示したものです。

```
Set objBootstrap = CreateObject("Tobj.Bootstrap")
```

次に、Bootstrap オブジェクトを初期化します。Bootstrap オブジェクトを初期化する場合、次のように、対象の BEA Tuxedo ドメインの IIOP リスナ / ハンドラのホストとポートを指定します。

```
objBootstrap.Initialize "//host:port"
```

IIOP リスナ / ハンドラのホストとポートの組み合わせは、`UBBCONFIG` ファイルの `ISL` パラメータで定義されます。Bootstrap オブジェクトに対して指定するホストとポートの組み合わせは、この `ISL` パラメータと完全に一致する必要があります。ホストとポートの組み合わせの形式のほかに、大文字、小文字も一致する必要があります。アドレスが一致しない場合、Bootstrap オブジェクトの呼び出しは失敗し、次のメッセージがログ・ファイルに記録されます。

```
Error: Unofficial connection from client at <tcp/ip address/<portnumber>
```

たとえば、`UBBCONFIG` ファイルの `ISL` パラメータでネットワーク・アドレスが `//TRIXIE::3500` として指定されている場合、Bootstrap オブジェクトで `//192.12.4.6.:3500` または `//trixie:3500` と指定すると接続が失敗します。

BEA Tuxedo ドメインには、複数の IIOP リスナ / ハンドラが存在できます。複数の IIOP リスナ / ハンドラを使用して BEA Tuxedo ドメインにアクセスする場合、*`host:port`* の組み合わせのリストを Bootstrap オブジェクトに提供します。Bootstrap オブジェクトは、BEA Tuxedo ドメインに接続するまでこのリストを参照します。また、IIOP リスナ / ヘッダのリストは、`TOBJADDR` 環境変数に指定することもできます。

複数の BEA Tuxedo ドメインにアクセスする場合、アクセスする BEA Tuxedo ドメインごとに Bootstrap オブジェクトを作成する必要があります。

FactoryFinder オブジェクトへの初期リファレンスの取得

ActiveX クライアント・アプリケーションは、自身にサービスを提供するオブジェクトへの初期リファレンスを取得する必要があります。Bootstrap オブジェクトを使用すると、FactoryFinder オブジェクト、SecurityCurrent オブジェクト、TransactionCurrent オブジェクト、NotificationService オブジェクト、Tobj_SimpleEventsService オブジェクト、および NameService オブジェクトへのリファレンスを取得できます。このオペレーションに受け渡す引数は、目的のオブジェクトの `progid` を含む文字列です。ActiveX クライアント・アプリケーションで使用するオブジェクトの初期リファレンスだけを取得する必要があります。

次の Visual Basic サンプルは、Bootstrap オブジェクトを使用して FactoryFinder オブジェクトへのリファレンスを取得する方法を示したものです。

```
Set objFactoryFinder = objBootstrap.CreateObject("Tobj.FactoryFinder")
```

ファクトリを使用した ActiveX ビューの取得

ActiveX クライアント・アプリケーションは、CORBA オブジェクトの ActiveX ビューへのインターフェイス・ポインタをファクトリから取得します。ファクトリは、別の CORBA オブジェクトへのオブジェクト・リファレンスを返す任意の CORBA オブジェクトです。ActiveX クライアント・アプリケーションは、ファクトリのオペレーションを呼び出して特定のタイプの CORBA オブジェクトへのオブジェクト・リファレンスを取得します。ファクトリを使用するには、ActiveX クライアント・アプリケーションは必要なファクトリを検索できなければなりません。FactoryFinder オブジェクトは、そのために役立ちます。FactoryFinder オブジェクトの機能の登録については、「ファクトリと FactoryFinder オブジェクト」を参照してください。

`CreateObject` 関数を使用して FactoryFinder オブジェクトを作成し、次に FactoryFinder オブジェクトのメソッドの 1 つを使用してファクトリを検索します。FactoryFinder オブジェクトには以下のメソッドがあります。

- `find_factories()`

  入力キーに完全に一致するファクトリのシーケンスを返します。

- `find_one_factory()`

  入力キーに一致する 1 つのファクトリを返します。

- `find_factories_by_id()`

  名前コンポーネントの ID フィールドが入力引数に一致するファクトリのシーケンスを返します。

- `find_one_factory_by_id()`

  ファクトリの CORBA 名前コンポーネントの ID フィールドが入力引数に一致する 1 つのファクトリを返します。

次の Visual Basic のサンプルでは、FactoryFinder `find_one_factory_by_id()` メソッドを使用して、Basic サンプル・アプリケーションのクライアント・アプリケーションで使用される Registrar オブジェクトのファクトリを取得する方法を示します。

```
Set objRegistrarFactory = objBsFactoryFinder.find_one_factory_by_id
    ("RegistrarFactory")
Set objRegistrar = RegistrarFactory.find_registrar
```

ActiveX ビューのオペレーションの呼び出し

ActiveX ビューのオペレーションを呼び出すには、ファクトリへのポインタとオペレーションで必要な引数を受け渡します。

次の Visual Basic のサンプルは、ActiveX ビューのオペレーションを呼び出す方法を示したものです。

```
'Get course details from the Registrar object'
aryCourseDetails = objRegistrar.get_course_details(aryCourseNumbers)
```

ステップ 7: ActiveX クライアント・アプリケーションのデプロイ

ActiveX クライアント・アプリケーションをほかのクライアント・マシンに配布するには、デプロイメント・パッケージを作成する必要があります。デプロイメント・パッケージには、クライアント・アプリケーションが CORBA オブジェクトの ActiveX ビューを使用するために必要なすべてのデータ (バインディング、タイプ・ライブラリ、登録情報など) が含まれています。デプロイメント・パッケージは自動登録を行う ActiveX コントロールで、ファイル拡張子は `.ocx` です。

ActiveX ビューのデプロイメント・パッケージを作成するには、次の手順に従います。

1. [Workstation Views] ウィンドウから、ActiveX ビューを選択します。
2. [Tools] の [Deploy Modules] をクリックするか、または目的のビューを右マウスボタンでクリックしてメニューから [Deploy Modules] を選択します。

   確認ウィンドウが表示されます。
3. [Create] をクリックすると、デプロイメント・パッケージが作成されます。

   デフォルトでは、デプロイメント・パッケージは `\tuxdir\Packages` に格納されます。

4　動的起動インターフェイスの使い方

ここでは、次の内容について説明します。

- DII の使用
- DII の概念
- DII の開発プロセスの概要
- ステップ 1: CORBA インターフェイスのインターフェイス・リポジトリへのロード
- ステップ 2: CORBA オブジェクトのオブジェクト・リファレンスの取得
- ステップ 3: リクエスト・オブジェクトの作成
- ステップ 4: DII 要求の送信と結果の取得
- ステップ 5: 要求の削除
- ステップ 6: DII でのインターフェイス・リポジトリの使い方

このトピックの情報は、CORBA C++ および CORBA Java クライアント・アプリケーションに適用されます。DII は、ActiveX クライアント・アプリケーションではサポートされません。

呼び出し方式と DII の概要については、「静的起動と動的起動」を参照してください。

このトピックで説明する CORBA メンバ関数の詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

DII の使用

静的起動または動的起動のいずれかを使用して CORBA クライアント・アプリケーションから要求を送信するのには、合理的な理由があります。しかし、同じ CORBA アプリケーションで両方の呼び出し方式を使用する必要がある場合もあります。呼び出し方式を選択するには、DII の利点と欠点を理解しておく必要があります。

静的起動と動的起動の大きな違いの 1 つは、どちらも同期 oneway 通信をサポートしているのに対し、動的起動だけが遅延同期通信をサポートしているという点です。

同期通信では、CORBA クライアント・アプリケーションは要求を送信し、応答を取得するまで待機します。応答待ちの間、CORBA クライアント・アプリケーションはほかの作業を行うことができません。遅延同期通信では、CORBA クライアント・アプリケーションは要求を送信し、ほかの作業を自由に行うことができます。CORBA クライアント・アプリケーションは、要求が完了したかどうかを定期的にチェックし、完了した場合は、その要求の結果を利用します。

さらに、DII を使用すると、CORBA クライアント・アプリケーションは、自身が記述された時点で未知であった CORBA オブジェクト型のメソッドを呼び出すことができます。これは、静的起動とは対照的です。静的起動では、CORBA クライアント・アプリケーションには呼び出す CORBA オブジェクトの各型に対するクライアント・スタブが含まれている必要があります。ただし、DII では、コードでクライアント・スタブの作業を実行する必要があるため、プログラミングが難しくなります。

CORBA クライアント・アプリケーションは、DII を使用してより高いパフォーマンスを取得できます。たとえば、CORBA クライアント・アプリケーションは複数の遅延同期要求を同時に送信し、それらの完了を処理できます。要求が異なるサーバ・アプリケーションに送信された場合、この処理を並列に行うことができます。同期クライアント・スタブを使用している場合は、こうした処理を行うことができません。

注記　クライアント・スタブは、最適化機能を備えています。このため、クライアント・スタブは、単一の要求を送信し、即座にブロックしてその要求の応答を取得するときに、DII で達成される応答時間より短い応答を達成できます。

DII は、単なる CORBA クライアント・アプリケーションへのインターフェイスです。CORBA サーバ・アプリケーションからは、静的起動と動的起動はまったく同じに見えます。

DII の概念

多くの場合、DII はあるタスクを達成するための複数の方法を提供します。それには、プログラミングの簡素化と性能のトレード・オフが必要となります。この節では、DII の使い方を理解するのに必要な高度な概念について説明します。このトピックの後半には、コード例などの詳細を示します。

この節で説明する概念は次のとおりです。

- リクエスト・オブジェクト
- 要求送信オプション
- 応答受信オプション

リクエスト・オブジェクト

リクエスト・オブジェクトは、ある CORBA オブジェクトの 1 つのメソッドの 1 つの呼び出しを表します。同じメソッドに対して 2 つの呼び出しを行う場合、2 つのリクエスト・オブジェクトを作成する必要があります。

メソッドを呼び出すには、そのメソッドが含まれている CORBA オブジェクトへのオブジェクト・リファレンスが必要です。オブジェクト・リファレンスを使用して、リクエスト・オブジェクトを作成し、そのリクエスト・オブジェクトに引数を指定し、要求を送信し、応答を待機して、要求の結果を取得します。

リクエスト・オブジェクトは、次の方法で作成できます。

- `CORBA::Object::_request` メンバ関数を使用します。

  `CORBA::Object::_request` メンバ関数を使用して、その要求で呼び出すインターフェイス名だけを指定する空のリクエスト・オブジェクト (`get_course_details` など) を作成します。リクエスト・オブジェクトを作成したら、CORBA サーバ・アプリケーションに送信する前に引数を指定す

る必要があります (存在する場合)。呼び出すメソッドで必要な引数ごとに、`CORBA::NVList::add_value` メンバ関数を呼び出します。

また、`CORBA::Request::result` メンバ関数を使用して、要求の結果の型を指定する必要があります。パフォーマンス上の理由から、Object Request Brokers (ORB) 間で交換されるメッセージには型情報は含まれません。結果の型のプレース・ホルダを指定することによって、応答から結果を適切に抽出するために必要な情報を ORB に提供します。同様に、呼び出すメソッドによって例外が発生する可能性がある場合、リクエスト・オブジェクトを送信する前に例外用のプレース・ホルダを追加する必要があります。

- `CORBA::Object::_create_request` メンバ関数を使用します。

  `CORBA::Object:: _create_request` メンバ関数を使用してリクエスト・オブジェクトを作成する場合、その要求を行うために必要なすべての引数を受け渡し、その結果の型とその要求が返す可能性のあるユーザ例外を指定します。このメンバ関数を使用することで、空の NVList を作成し、一度に 1 つの引数を NVList に追加し、リクエスト・オブジェクトを作成して、完成した NVList をその要求に引数として受け渡します。

  `CORBA::Object::_create_request` メンバ関数には、性能上のメリットがあります。複数のオブジェクトの同じオブジェクトを呼び出す場合、これらの引数を複数の `CORBA::ORB::_create_request` 呼び出しで再利用できます。

CORBA メンバ関数の詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

要求送信オプション

リクエスト・オブジェクトを作成し、引数、結果の型、および例外の型を追加したら、その要求を CORBA オブジェクトに送信します。要求は、以下の方法で送信できます。

- 最も単純な方法は、`CORBA::Request::invoke` メンバ関数を呼び出すことです。この関数は、応答メッセージを取得するまでブロックします。
- 上の方法より複雑で、ブロッキングを行わない方法は、`CORBA::Request::send_deferred` メンバ関数を使用することです。

- 複数の CORBA 要求を並列に呼び出す場合は、`CORBA::ORB::send_multiple_requests_deferred` メンバ関数を使用します。これは、リクエスト・オブジェクトのシーケンスを取ります。
- CORBA メソッドが OMG IDL ファイルで oneway として定義されている場合に限り、`CORBA::Request::send_oneway` メンバ関数を使用します。
- 複数の oneway メソッドを並列に呼び出す場合は、`CORBA::ORB::send_multiple_requests_oneway` メンバ関数を使用します。

注記 `CORBA::Request::send_deferred` メンバ関数を使用する場合、対象オブジェクトが呼び出しを発行する CORBA クライアント・アプリケーションと同じアドレス空間に存在するときには、リクエスト・オブジェクトの呼び出しは同期的に動作します。この動作の結果、`CosTransaction::Current::suspend` オペレーションを呼び出しても、`CORBA::BAD_IN_ORDER` 例外は発生しません。トランザクションが完了したからです。

CORBA メンバ関数の詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

要求結果受信オプション

呼び出しメソッドを使用して要求を送信した場合、結果を取得する方法は 1 つしかありません。その方法とは、リクエスト・オブジェクトの `CORBA::Request::env` メンバ関数を使用して例外をチェックし、例外が存在しない場合は `CORBA::Request::result` メンバ関数を使用してリクエスト・オブジェクトから NVList を抽出することです。

遅延同期メソッドを使用して要求を送信した場合、次のいずれかのメンバ関数を使用して結果を取得できます。

- `CORBA::ORB::poll_response`

  このメンバ関数は、要求が完了し、処理の準備ができているかどうかを調べます。このメンバ関数はブロックしません。要求を処理できる場合、CORBA クライアント・アプリケーションは `get_response()` または `get_next_response()` メンバ関数を使用して応答を処理する必要があります。この関数は、応答の処理順序を考慮しない場合、CORBA クライアント・アプリケーションが特定の応答を待つ間ほかの要求を処理するようにする場合、またはタイムアウトを設定する場合に使用します。

- `CORBA::ORB::poll_next_response`

  このメンバ関数は、未処理の要求が処理できる状態にあるかどうかを調べます。要求を処理できる場合、CORBA クライアント・アプリケーションは `get_response()` または `get_next_response()` メンバ関数を使用して応答を処理する必要があります。このメンバ関数は、要求の処理順序が重要ではなく、CORBA クライアント・アプリケーションが特定の応答を待つ間にほかの要求を処理する場合に使用します。

- `CORBA::ORB::get_response`

  このメンバ関数は、特定の要求に対する応答が完了して処理されるまでブロックします。このメンバ関数は、未処理の要求を特定の順序で処理する場合に使用します。

- `CORBA::ORB::get_next_response`

  このメンバ関数は、未処理の要求に対する応答が完了して処理されるまでブロックします。このメンバ関数は、要求の処理順序が重要でない場合に使用します。

`CORBA::Request::send_oneway` メンバ関数を使用する場合、結果は存在しません。

CORBA メンバ関数の詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

DII の開発プロセスの概要

クライアント・アプリケーションで DII を使用するための手順は次のとおりです。

手順	説明
1	CORBA インターフェイスをインターフェイス・リポジトリにロードします。
2	呼び出すメソッドが含まれる CORBA オブジェクトへのオブジェクト・リファレンスを取得します。

手順	説明
3	CORBA オブジェクト用のリクエスト・オブジェクトを作成します。
4	DII 要求を送信し、結果を取得します。
5	要求を削除します。
6	DII でインターフェイス・リポジトリを使用します。

以降の節では、これらのステップについて詳しく説明し、C++ のコード例を示します。

ステップ 1: CORBA インターフェイスのインターフェイス・リポジトリへのロード

DII を使用する CORBA クライアント・アプリケーションを作成する前に、CORBA オブジェクトのインターフェイスをインターフェイス・リポジトリにロードしておく必要があります。CORBA オブジェクトのインターフェイスがインターフェイス・リポジトリにロードされていない場合、BEA Application Builder にはそれらが表示されません。目的の CORBA インターフェイスが [Services] ウィンドウに表示されない場合、`idl2ir` コマンドを使用して、CORBA オブジェクトを定義する OMG IDL をインターフェイス・リポジトリにロードします。`idl2ir` コマンドの構文は次のとおりです。

```
idl2ir [-f repositoryfile.idl] file.idl
```

オプション	説明
`-f` *repositoryfile*	CORBA インターフェイスの OMG IDL ファイルを指定されたインターフェイス・リポジトリにロードします。CORBA クライアント・アプリケーションがアクセスする BEA Tuxedo ドメインのインターフェイス・リポジトリの名前を指定します。

オプション	説明
file.idl	CORBA インターフェイスの定義が記述されている OMG IDL ファイルを指定します。

`idl2ir` コマンドの詳細については、『BEA Tuxedo コマンド・リファレンス』を参照してください。

ステップ 2: CORBA オブジェクトのオブジェクト・リファレンスの取得

Bootstrap オブジェクトを使用して、FactoryFinder オブジェクトを取得します。次に、FactoryFinder オブジェクトを使用して、DLL 要求からアクセスする CORBA オブジェクトのファクトリを取得します。Boostrap オブジェクトと FactoryFinder オブジェクトを使用してファクトリを取得する例については、「ステップ 4: CORBA クライアント・アプリケーションの記述」を参照してください。

ステップ 3: リクエスト・オブジェクトの作成

CORBA クライアント・アプリケーションが CORBA オブジェクトのメソッドを呼び出す場合、そのメソッド呼び出しのための要求を作成します。この要求はバッファに書き込まれ、CORBA サーバ・アプリケーションに送信されます。CORBA クライアント・アプリケーションがクライアント・スタブを使用する場合、この処理は透過的に発生します。DII を使用する クライアント・アプリケーションは、リクエスト・オブジェクトを作成し、その要求を送信する必要があります。

CORBA::Object::_request メンバ関数の使い方

次の C++ の コード例に、CORBA::Object::_request メンバ関数の使い方を示します。

```
Boolean          aResult;
CORBA::ULong     long1 = 42;
CORBA::Any       in_arg1;
CORBA::Any       &in_arg1_ref = in_arg1;

in_arg1 <<= long1;

// 短縮形式を使用して要求を作成
Request_ptr reqp = anObj->_request(“anOp”);

// 引数操作ヘルパ関数を使用
reqp->add_in_arg() <<= in_arg1_ref;

// boolean 型の結果が必要
reqp->set_return_type(_tc_boolean);

// 結果用の場所を提供
CORBA::Any::from_boolean boolean_return_in(aResult);
reqp->return_value() <<= boolean_return_in;

// invoke を実行
reqp->invoke();

// エラーがないので、戻り値を取得
CORBA::Any::to_boolean boolean_return_out(aResult);
reqp->return_value() >>= boolean_return_out;
```

CORBA::Object::create_request メンバ関数の使い方

CORBA::Object::create_request メンバ関数を使用してリクエスト・オブジェクトを作成する場合、空の NVList を作成し、その NVList に引数を 1 つずつ追加します。次に、リクエスト・オブジェクトを作成し、完成した NVList を引数としてそのリクエストに受け渡します。

リクエスト・オブジェクトの引数の設定

リクエスト・オブジェクトの引数は、名前付き / 値オブジェクトを格納する NVList オブジェクトで表されます。このリスト内のオブジェクトの追加、削除、問い合わせを行うためのメソッドが用意されています。CORBA::NVList の詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

CORBA::NamedValue メンバ関数による入力引数と出力引数の設定

CORBA::NamedValue メンバ関数は、要求の入力引数と出力引数を表すために使用する名前付き / 値オブジェクトを指定します。名前付き / 値オブジェクトは、リクエスト・オブジェクトの引数として使用されます。また、CORBA::NamedValue のペアは、CORBA クライアント・アプリケーションに返される要求の結果を表すためにも使用されます。名前付き / 値オブジェクトの名前プロパティは文字列で、値プロパティは CORBA Any によって表されます。

CORBA::NamedValue メンバ関数の詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

CORBA::Object::create_request メンバ関数の使用例

次の C++ のコード例に、CORBA::Object::create_request メンバ関数の使い方を示します。

```
CORBA::Request_ptr          reqp;
CORBA::Context_ptr          ctx;
CORBA::NamedValue_ptr       boolean_resultp = 0;
Boolean                     boolean_result;
CORBA::Any              boolean_result_any(CORBA::_tc_boolean, & boolean_result);
CORBA::NVList_ptr           arg_list = 0;
CORBA::Any                  arg;

// デフォルト・コンテキストを取得
orbp->get_default_context(ctx);

// 結果の名前付き値ペアを作成
(void) orbp->create_named_value(boolean_resultp);
CORBA::Any *tmpany = boolean_resultp->value();
*tmpany = boolean_result_any;
```

```
arg.replace(CORBA::_tc_long, &long_arg, CORBA_FALSE)

// NVList を作成
orbp->create_list(1, arg_list);

// IN 引数をリストに追加
arg_list->add_value(“arg1”, arg, CORBA::ARG_IN);

// 長い形式を使用して要求を作成
anObj->_create_request (ctx,
                        "anOp",
                        arg_list,
                        boolean_resultp,
                        reqp,
                        CORBA::VALIDATE_REQUEST );
// invoke を実行
reqp->invoke();

CORBA::NamedValue_ptrresult_namedvalue;
Boolean aResult;
CORBA::Any *result_any;
// 結果を取得
result_namedvalue = reqp->result();
result_any = result_namedvalue->value();

// any から Boolean を抽出
*result_any >>= aResult;
```

ステップ 4: DII 要求の送信と結果の取得

要求は、使用する通信の方式に応じて複数の方法で呼び出すことができます。この節では、CORBA メンバ関数を使用して要求の送信と結果の取得を行う方法について説明します。

同期要求

同期通信を行う場合、CORBA::Request::invoke メンバ関数は要求を送信し、その応答が CORBA クライアント・アプリケーションに返されるまで待機します。CORBA::Request::result メンバ関数は、戻り値を表す名前付き / 値オブジェクトへのリファレンスを返すために使用します。結果を取得したら、要求に格納されている NVList から値を読み出します。

遅延同期要求

要求の送信用として、非ブロッキング・メンバ関数の CORBA::Request::send_deferred も用意されています。このメンバ関数を使用すると、CORBA クライアント・アプリケーションは要求を送信し、CORBA::Request::poll_response メンバ関数を使用して応答が利用可能であるかどうかを調べることができます。CORBA::Request::get_response メンバ関数は、応答が利用可能になるまでブロックします。

次のコード例に、CORBA::Request::send_deferred、CORBA::Request::poll_response、および CORBA::Request::get_response メンバ関数の使い方を示します。

```
request->send_deferred ();

if (poll)
{
     for ( int k = 0 ; k < 10 ; k++ )
     {
         CORBA::Boolean done = request->poll_response();
         if ( done )
           break;

     }
}
request->get_response();
```

oneway の要求

oneway の要求を送信するには、CORBA::Request::send_oneway メンバ関数を使用します。oneway の要求では、CORBA サーバ・アプリケーションから応答が返されません。CORBA::Request::send_oneway メンバ関数の詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

次のコード例に、CORBA::Request::send_oneway メンバ関数の使い方を示します。

```
request->send_oneway();
```

複数の要求

CORBA::Request::send_multiple_requests_deferred メンバ関数を使用してリクエスト・オブジェクトのシーケンスを送信する場合、CORBA::ORB::poll_response、CORBA::ORB::poll_next_response、CORBA::ORB::get_response、および CORBA::ORB::get_next_response メンバ関数を使用すると、CORBA サーバ・アプリケーションが各要求に対して送信する応答を取得できます。

CORBA::ORB::poll_response メンバ関数と CORBA::ORB::poll_next_response メンバ関数を使用すると、CORBA サーバ・アプリケーションから応答を取得したかどうかを調べることができます。これらのメンバ関数は、少なくとも 1 つの応答が利用可能な場合は 1 を返し、利用可能な応答が存在しない場合は 0 を返します。

CORBA::ORB::get_response メンバ関数と CORBA::ORB::get_next_response メンバ関数を使用すると、応答を取得できます。利用可能な応答が存在しない場合、これらのメンバ関数は応答を取得するまでブロックします。CORBA クライアント・アプリケーションがブロックしないようにする場合は、最初に CORBA::ORB::poll_next_response メンバ関数を使用して応答が利用可能かどうかを調べてから、CORBA::ORB::get_next_response メソッドを使用してその結果を取得します。

また、CORBA::Request::send_multiple_requests_oneway メンバ関数を使用すると、複数の oneway 要求を送信できます。

次のコード例に、CORBA::Request::send_multiple_requests_deferred、CORBA::Request::poll_next_response、および CORBA::Request::get_next_response メンバ関数の使い方を示します。

```
CORBA::Context_ptr          ctx;
CORBA::Request_ptr          requests[2];
CORBA::Request_ptr          request;
CORBA::NVList_ptr           arg_list1, arg_list2;
CORBA::ULong                i, nreq;
CORBA::Long                 arg1 = 1;
Boolean                     aResult1 = CORBA_FALSE;
Boolean                     expected_aResult1 = CORBA_TRUE;
CORBA::Long                 arg2 = 3;
Boolean                     aResult2 = CORBA_FALSE;
Boolean                     expected_aResult2 = CORBA_TRUE

try
{
     orbp->get_default_context(ctx);

     populate_arg_list ( &arg_list1, &arg1, &aResult1 );

     nreq = 0;

     anObj->_create_request ( ctx,
                             "Multiply",
                              arg_list1,
                              0,
                              requests[nreq++],
                              0);

     populate_arg_list ( &arg_list2, &arg2, &aResult2 );

     anObj->_create_request ( ctx,
                               "Multiply",
                               arg_list2,
                               0,
                               requests[nreq++],
                               0 );

// 要求シーケンス変数を宣言 ...
CORBA::ORB::RequestSeq rseq ( nreq, nreq, requests, CORBA_FALSE );

orbp->send_multiple_requests_deferred ( rseq );
for ( i = 0 ; i < nreq ; i++ )
```

```
{
      requests[i]->get_response();
}

// ここで結果をチェック

if ( aResult1 != expected_aResult1 )
{
      cout << "aResult1=" << aResult1 << " different than expected: " <<
expected_aResult1;
}

if ( aResult2 != expected_aResult2 )
{
      cout << "aResult2=" << aResult2 << " different than expected: " <<
expected_aResult2;
}

aResult1 = CORBA_FALSE;
aResult2 = CORBA_FALSE;

// 同じ引数リストを使用して、数値を再び乗算
// 今度は応答をポーリング

orbp->send_multiple_requests_deferred ( rseq );

// ここで応答をポーリング

for ( i = 0 ; i < nreq ; i++ )
{

// 最大 10 回ランダムにポーリング
     for ( int j = 0 ; j < 10 ; j++ )
     {
      CORBA::Boolean done = requests[i]->poll_response();

      if ( done ) break;
      }
}
// ここで応答を実際に所得
for ( i = 0 ; i < nreq ; i++ )
{
     requests[i]->get_response();
}

// ここで結果をチェック
if ( aResult1 != expected_aResult1 )
{
      cout << "aResult1=" << aResult1 << " different than expected: " <<
expected_aResult1
```

```
}
if ( aResult2 != expected_aResult2 )
{
       cout << "aResult2=" << aResult2 << " different than expected: " <<
expected_aResult2;
}

aResult1 = CORBA_FALSE;
aResult2 = CORBA_FALSE;

// 同じ引数リストを使用して、数値を再び乗算
// get_next_response を呼び出し、応答を待機
orbp->send_multiple_requests_deferred ( rseq );

// 応答を取得するまでポーリングし、get_next_response を使用して取得
for ( i = 0 ; i < nreq ; i++ )
     {
           CORBA::Boolean res = 0;

           while ( ! res )
           {
             res = orbp->poll_next_response();
           }
           orbp->get_next_response(request);
           CORBA::release(request);
     }
// ここで結果をチェック
if ( aResult1 != expected_aResult1 )
{
      cout << "aResult1=" << aResult1 << " different than expected: " <<
expected_aResult1;
}
if ( aResult2 != expected_aResult2 )
{
      cout << "aResult2=" << aResult2 << " different than expected: " <<
expected_aResult2;
}


static void populate_arg_list (
CORBA::NVList_ptr          ArgList,
CORBA::Long                * Arg1,
CORBA::Long                * Result )
{
CORBA::Any                 any_arg1;
CORBA::Any                 any_result;

(* ArgList) = 0;
```

```
orbp->create_list(3, *ArgList);

any_arg1 <<= *Arg1;
any_result.replace(CORBA::_tc_boolean, Result, CORBA_FALSE);

(*ArgList)->add_value(“arg1”, any_arg1, CORBA::ARG_IN);
(*ArgList)->add_value(“result”, any_result, CORBA::ARG_OUT);

return;

}
```

ステップ 5: 要求の削除

要求が正常に完了したことが通知されたら、既存の要求を削除するか、または次の呼び出しでその一部を再利用するかを決定する必要があります。

要求全体を削除するには、削除する要求に対して `CORBA::Release(request)` メンバ関数を使用します。このオペレーションにより、要求に関連するすべてのメモリが解放されます。遅延同期通信を使用して発行された要求を削除する場合、その要求が完了していないときはその要求がキャンセルされます。

ステップ 6: DII でのインターフェイス・リポジトリの使い方

CORBA クライアント・アプリケーションは、その作成時に未知であったオブジェクトの要求を作成、設定、および送信できます。そのために、CORBA クライアント・アプリケーションはインターフェイス・リポジトリを使用して、要求を作成および設定するために必要な情報を取得します。CORBA クライアント・アプリケーションは、そのインターフェイスのクライアント・スタブを持っていないので、DII を使用して要求を送信します。

このテクニックは未知の型の CORBA オブジェクトのオペレーションを呼び出すのに役立ちますが、インターフェイス・リポジトリとの対話によるオーバーヘッドのため、パフォーマンスが問題となります。このタイプの DII 要求は、オ

ブジェクトを参照する CORBA クライアント・アプリケーションを作成する場合か、または管理ツールの CORBA クライアント・アプリケーションを作成する場合に使用を検討します。

DLL 要求でインターフェイス・リポジトリを使用するための手順は次のとおりです。

1. `CORBA.h` の `ORB_INCLUDE_REPOSITORY` を、BEA Tuxedo システム内でのインターフェイス・リポジトリ・ファイルの格納場所に設定します。
2. Bootstrap オブジェクトを使用して、インターフェイス・リポジトリ・オブジェクトを取得します。このオブジェクトには、特定の BEA Tuxedo ドメイン内のインターフェイス・リポジトリへのリファレンスが含まれています。インターフェイス・リポジトリへのリファレンスを取得したら、そのインターフェイス・リポジトリをルートから参照できます。
3. `CORBA::Object::_get_interface` メンバ関数を使用して、目的の CORBA オブジェクトをインプリメントする CORBA サーバ・アプリケーションと通信します。
4. `CORBA::InterfaceDef_ptr` を使用して、インターフェイス・リポジトリに格納されている CORBA インターフェイスの定義を取得します。
5. `FullInterfaceDescription` オペレーションで、目的のオペレーション用の `OperationDescription` を検索します。
6. `OperationDescription` からリポジトリ ID を取得します。
7. `OperationDescription` で返されたリポジトリ ID を使用して `CORBA::Repository::lookup_id` を呼び出して、インターフェイス・リポジトリ内の `OperationDef` を検索します。この呼び出しにより、包含オブジェクトが返されます。
8. 包含オブジェクトを `OperationDef` にナロー変換します。
9. `OperationDef` 引数と `CORBA::ORB::create_operation_list` メンバ関数を使用して、オペレーションの引数リストを作成します。
10. オペレーション・リスト内の引数値を設定します。
11. ほかの要求と同じように、要求を送信して結果を取得します。このトピックで説明した任意のオプションを使用して、要求を送信して結果を取得できます。

5　例外処理

このトピックでは、CORBA C++、CORBA Java、および ActiveX クライアント・アプリケーションで CORBA 例外を処理する方法について説明します。

CORBA 例外処理の概念

CORBA では、以下の例外が定義されています。

- システム例外。これらは、メモリ不足や通信障害などの一般的なエラーです。システム例外には、Object Request Broker (ORB) によって発生した例外が含まれます。CORBA 仕様には、CORBA クライアント・アプリケーションからの要求の処理でエラーが発生したときに発生する可能性のあるシステム例外が定義されています。
- ユーザ例外。これらは、オブジェクトによって発生する例外です。この例外には、ユーザ定義のデータが含まれています。OMG IDL で CORBA オブジェクトのインターフェイスを定義する場合、オブジェクトによって発生する可能性のあるユーザ例外を指定できます。

以降の節では、各タイプの CORBA クライアント・アプリケーションがどのように例外を処理するかについて説明します。

CORBA システム例外

表 5-1 に、CORBA システム例外のリストを示します。

表 5-1 CORBA システム例外

例外名	説明
BAD_CONTEXT	コンテキスト・オブジェクトの処理中にエラーが発生しました。
BAD_INV_ORDER	ルーチン呼び出しが順不同です。
BAD_OPERATION	無効なオペレーションです。
BAD_PARAM	無効なパラメータが受け渡されました。
BAD_TYPECODE	無効な typecode です。
COMM_FAILURE	通信障害。
DATA_CONVERSION	データ変換エラー。
FREE_MEM	メモリを解放できません。
IMP_LIMIT	インプリメンテーション制限に違反しました。
INITIALIZE	ORB 初期化障害。
INTERNAL	ORB 内部エラー。
INTF_REPOS	インターフェイス・リポジトリへのアクセス中にエラーが発生しました。
INV_FLAG	無効なフラグが指定されています。
INV_IDENT	無効な識別子構文です。
INV_OBJREF	無効なオブジェクト・リファレンスが指定されています。
MARSHAL	パラメータまたは結果のマーシャリング・エラー。
NO_IMPLEMENT	オペレーションのインプリメンテーションが利用できません。
NO_MEMORY	動的メモリ割り当てが異常終了しました。
NO_PERMISSION	試行されたオペレーションのパーミッションが存在しません。

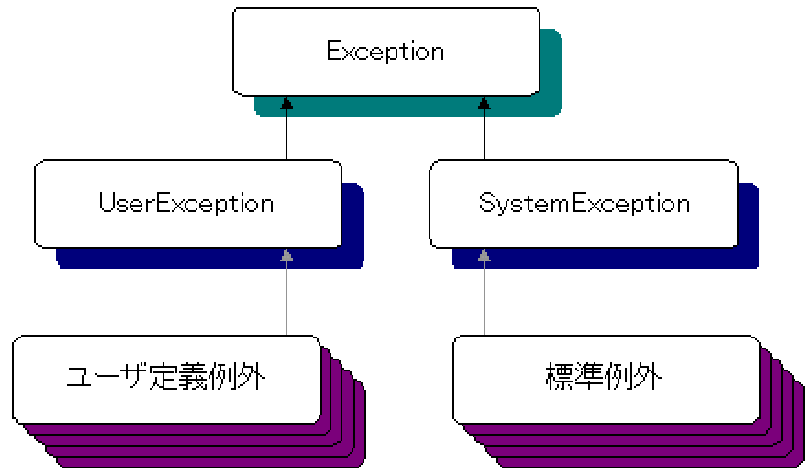
表 5-1 CORBA システム例外 (続き)

例外名	説明
NO_RESOURCES	要求を処理するためのリソースが不足しています。
NO_RESPONSE	要求に対する応答がまだ利用できません。
OBJ_ADAPTER	オブジェクト・アダプタによってエラーが検出されました。
OBJECT_NOT_EXIST	オブジェクトが利用できません。
PERSIST_STORE	永続ストレージの障害。
TRANSIENT	一時障害。
UNKNOWN	未知の結果。

CORBA C++ クライアント・アプリケーション

システム例外とユーザ例外のどちらも類似の機能を必要とするので、SystemException クラスと UserException クラスが共通の Exception クラスから派生されます。例外が発生すると、CORBA クライアント・アプリケーションは Exception クラスから固有の SystemException クラスまたは UserException クラスにナロー変換することができます。図 5-1 に、C++ Exception の継承階層を示します。

図 5-1　C++ Exception の継承階層



Exception クラスは、以下のパブリック・オペレーションを提供します。

- copy constructor
- destructor
- _narrow

copy constructor オペレーションと destructor オペレーションは、例外に関連付けられているストレージを自動的に管理します。

_narrow オペレーションを使用すると、CORBA クライアント・アプリケーションは任意の型の例外をキャッチしてその型を特定できます。_narrow オペレーションに渡される exception 引数は、基底クラスの Exception へのポインタです。_narrow オペレーションは、任意の Exception オブジェクトへのポインタを受け付けます。ポインタが SystemException 型である場合、narrow() オペレーションはその例外へのポインタを返します。ポインタが SystemException 型でない場合、narrow() オペレーションは Null ポインタを返します。

オブジェクト・リファレンスの _narrow オペレーションとは異なり、例外の _narrow オペレーションは新しい例外へのポインタではなく、同じ例外への適切に型付けされたポインタを返します。このため、_narrow オペレーションによって返されるポインタを開放してはなりません。元の例外がスコープをはずれるか破棄された場合、_narrow オペレーションによって返されるポインタは有効でなくなります。

注記　BEA Tuxedo CORBA サンプル・アプリケーションでは、`_narrow` オペレーションは使用されていません。

システム例外の処理

BEA Tuxedo サンプル・アプリケーションの CORBA C++ クライアント・アプリケーションは、エラー状態が発生した場合、ステータス値をチェックしてエラーを検出する代わりに、標準の C++ try-catch 例外処理メカニズムを使用して例外を発生およびキャッチします。この例外処理メカニズムは、CORBA 例外を CORBA クライアント・アプリケーションに統合するためにも使用されます。C++ では、`catch` 句が指定された順序で試行され、最初に一致したハンドラが呼び出されます。

次の Basic サンプル・アプリケーションの CORBA C++ クライアント・アプリケーションの例に、<< 演算子を使用した出力を示します。

注記　このトピックでは、サンプル内の例外コードを太字で表します。

```
try{

//ORB を初期化
CORBA::ORB* orb=CORBA::ORB_init(argc, argv, ORBid);

//Bootstrap オブジェクトを取得
Tobj_Bootstrap* bs= new Tobj_Bootstrap(orb, "//host:port");

// Factory Finder を解決
CORBA::Object_var var_factory_finder_oref = bs->
     resolve_initial_reference("FactoryFinder");
Tobj::FactoryFinder_var var_factory_finder_ref = Tobj::FactoryFinder::_narrow
     (var_factory_finder_oref.in());

catch(CORBA::Exception& e) {
     cerr <<e.get_id() <<endl;
}
```

ユーザ例外

ユーザ例外は、それらが定義されているユーザ記述 OMG IDL ファイルから生成されます。例外を処理する場合、コードは最初にシステム例外をチェックする必要があります。システム例外は CORBA によってあらかじめ定義されており、多くの場合アプリケーションはシステム例外から回復できません。たとえば、システム例外はネットワーク転送の問題や ORB の内部的な問題を示す場合があります。システム例外のチェックが済んだら、特定のユーザ例外をチェックします。

次の C++ サンプルに、`Registar` インターフェイス内部の `TooManyCredits` ユーザ例外を宣言する OMG IDL ファイルを示します。例外はモジュール内またはインターフェイス内のいずれかで宣言できることに注意してください。

```
exception TooManyCredits
{
    unsigned short maximum_credits;
};

interface Registrar

NotRegisteredList register_for_courses(
    in StudentId                    student,
    in CourseNumberList             courses
) raises (
    TooManyCredits
);
```

次の C++ のコード例に、`TooManyCredits` ユーザ例外がクラスの登録用のトランザクションのスコープ内でどのように機能するかを示します。

```
// コースの学生を登録

try {
     pointer_registrar_reference->register_for_courses
         (student_id, course_number_list);

catch (UniversityT::TooManyCredits& e) {
    cout <<"You cannot register for more than"<< e.maximum_credits
    <<"credits."<<endl;
}
```

CORBA Java クライアント・アプリケーション

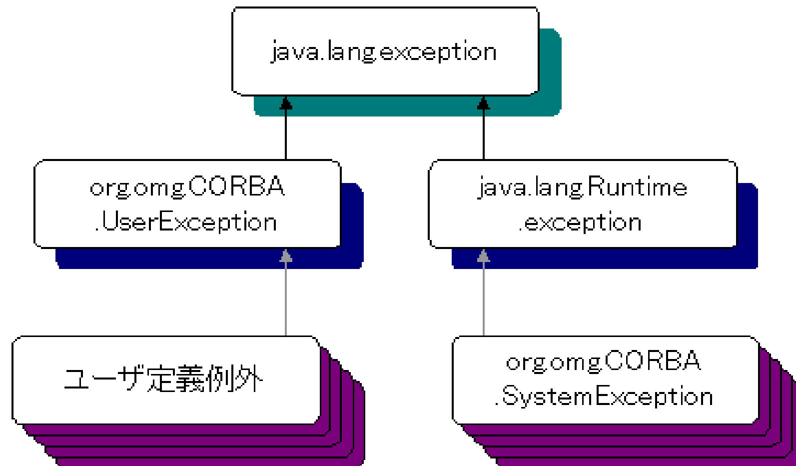
注記 このセクションの情報は、「OMG IDL/Java Language Mapping Specification, orbos/97-03-01」(1997 年 3 月 19 日改訂) をベースにしています。

CORBA Java クライアント・アプリケーションは、CORBA C++ クライアント・アプリケーションと同じような方法で例外を処理します。

- システム例外は `java.lang.RuntimeException` から継承されます。
- ユーザ定義例外は `java.lang.Exception` から継承されます。

図 5-2 に、Java Exception クラスの継承階層を示します。

図 5-2 Java Exception 継承階層



システム例外

標準の OMG IDL システム例外は、org.omg.CORBA.SystemException を拡張し、OMG IDL の大小の例外コード、および例外の理由を定義した文字列へのアクセスを提供する最終的な Java クラスにマップされます。

注記　org.omg.CORBA.SystemException のパブリック・コンストラクタは存在せず、その拡張クラスだけをインスタンス化できます。

各標準 OMG IDL 例外の Java クラス名はその OMG IDL 名と同じで、org.omg.CORBA パッケージで宣言されます。たとえば、CORBA 定義のシステム例外 BAD_CONTEXT は、org.omg.CORBA.BAD_CONTEXT としてマップされます。デフォルト・コンストラクタは、小さいコードの場合は 0、完了コードの場合は COMPLETED_NO、および理由の文字列の場合は "" を提供します。また、理由を取り、ほかのフィールドのデフォルトを使用するコンストラクタ、および 3 つのパラメータをすべて指定する必要があるコンストラクタが存在します。

次の Java コード例に、システム例外の使い方を示します。

```
try
 {
// Factory Finder を解決
  org.omg.CORBA.Object off = bs.resolve_initial_references
      ("FactoryFinder");
  FactoryFinder ff=FactoryFinderHelper.narrow(off);

  org.omg.CORBA.Object of = FactoryFinder.find_one_factory_by_id
     (UniversityT.RegistrarFactoryHelper.id());
  UniversityT.RegistrarFactory F =
      UniversityT.RegistrarFactoryHelper.narrow(of);

catch (org.omg.CORBA.SystemException e)
   {
         System.err.println("System exception " + e);
         System.exit(1);
}
```

ユーザ例外

ユーザ例外は org.omg.CORBA.UserException を拡張する最終 Java クラスにマップされ、それ以外の場合は OMG IDL struct データ型のようにマップされます (Helper クラスと Holder クラスの生成を含む)。

例外が入れ子になった OMG IDL スコープ内 (本質的にインターフェイス内部) で定義される場合、その Java クラス名は特別なスコープ内で定義されます。それ以外の場合、その Java クラス名は例外の包含 OMG IDL モジュールに対応する Java パッケージのスコープ内で定義されます。

次に、ユーザ例外のサンプルを示します。

```
// コースへの登録を行う
try{
   gRegistrarObjRef.register_for_courses(student_id, selected_course_numbers);

catch(UniversityT.TooManyCredits e)
{
        System.err.println("TooManyCredits: " +e);
        System.exit(1);
}
```

ActiveX クライアント・アプリケーション

ActiveX クライアント・アプリケーションは、Visual Basic エラー処理モデルを使用します。このモデルを使用すると、エラー発生時に特別なアクション (特定のエラー処理ルーチンへのジャンプなど) を実行できます。ActiveX クライアント・アプリケーションで例外が発生した場合、標準の Visual Basic エラー処理は期待通りに機能しますが、Visual Basic が例外に対して返すエラー情報の量は非常に限定されます。

Visual Basic は、組み込みの Error オブジェクトの description プロパティを通して発生する例外に関する追加情報を提供します。エラーが発生すると、説明文字列が設定され、発生したエラーの型が示されます。オブジェクトは、例外の定義済みのデータ型を返します。ユーザ例外には、それらを区別するために名前が付けられます。

Visual Basic エラー処理モデルを使用する場合、説明文字列には以下の情報が定義されます。

- 例外がユーザ例外かシステム例外か
- 例外の名前
- 例外の発生前にオペレーションが完了したかどうか

Visual Basic エラー処理モデルは、ユーザ例外のユーザ・データなど、例外固有の情報を返すことができません。

この欠点を補うため、CORBA オブジェクトの ActiveX ビューは、ユーザ例外を返す追加のオプション例外戻りパラメータを備えています。このオプション例外オブジェクトを提供した場合、Visual Basic 例外は発生しません。代わりに、この戻りパラメータは例外情報を返します。

例外が発生した場合、戻りパラメータにはその例外のデータを取得するためのオブジェクトが含まれます。このオブジェクトは、各値のプロパティを備えた構造疑似オブジェクトに似ています。例外の型を調べるには、例外オブジェクト・プロパティ `EX_majorCode` または `EX_minorCode` を使用します。`EX_majorCode` オブジェクト・プロパティは、以下の 3 種類の値を取ることができます。

- 例外が発生しなかった場合は 0
- システム例外が発生した場合は 1
- ユーザ例外が発生した場合は 2

次に、例外を処理する Visual Basic のコード例を示します。

```
Dim exceptType As ExceptionType
Dim exceptInfo As DIForeignException

Set exceptInfo = Exc
exceptType = exceptInfo.EX_majorCode

Select Case exceptType

    Case NO_EXCEPTION

            msg = "No Exception" & vbCrLf

            MsgBox msg

    Case SYSTEM_EXCEPTION
```

```
        ' システム例外の場合、返されるバリアントは
        ' minorCode および completionStatus properties をサポートする

        Dim minorCode As Long
        Dim completionStatus As CORBA_CompletionStatus
        Dim completionMsg As String

        minorCode = exceptInfo.EX_minorCode
        completionStatus = exceptInfo.EX_completionStatus
        Select Case completionStatus
            Case COMPLETION_NO
                 completionMsg = "No"
            Case COMPLETION_YES
                 completionMsg = "Yes"
            Case COMPLETION_MAYBE
                 completionMsg = "Maybe"
        End Select

        msg = "System Exception" & vbCrLf
        msg = msg & "    Minor Code = " & minorCode & vbCrLf
        msg = msg & "    Completion Status = " & completionMsg & vbCrLf

        MsgBox msg

    Case USER_EXCEPTION

        ' ユーザ例外の場合、返されるバリアントは
        ' 定義されたユーザ例外のプロパティをサポートする

        msg = "User Exception" & vbCrLf
        msg = msg & "    Exception: " & exceptInfo.INSTANCE_repositoryId &
        vbCrLf
        MsgBox msg

    End Select
```

索引

A

ActiveX 1-22
概念 1-22
バインディング 1-22
ビュー 1-22
命名規則 1-24
ActiveX クライアント・アプリケーション
ISL パラメータ 3-6
インターフェイスのインターフェイス・リポジトリへのロード 3-4
インターフェイス・リポジトリのサーバ・アプリケーションの起動 3-5
インターフェイス・リポジトリの使用 1-7
オブジェクトのオペレーションの呼び出し 3-11
オブジェクトへの初期リファレンスの解決 3-11
概念 1-22
開発プロセス 3-2
環境オブジェクトのインターフェイス・リポジトリへのロード 3-4
記述 3-8
作成
バインディング 3-6
ビュー 3-6
説明 1-2
ドメインとの通信の確立 3-9
ビューのデプロイ 3-13
ファクトリの使用 3-11
例外処理 5-9
Application Builder
説明 1-22

B

BEA Application Builder
ISL パラメータ 3-7
ウィンドウ 3-3
作成
タイプ・ライブラリ 3-7
デプロイメント・パッケージ 3-13
バインディング 3-6
ビュー 3-6
BEA Tuxedo ドメインとの関係 1-9
Bootstrap オブジェクト
DII での使用 4-8
サーバ・アプリケーションの使用 2-15
初期リファレンスの解決
C++ 2-11
Java 2-11
Visual Basic 3-9
説明 1-11
宣言
Visual Basic 3-9
buildobjclient コマンド 2-2

C

C++
コード例
Bootstrap オブジェクト 2-11
Bootstrap オブジェクトの使い方 2-10
FactoryFinder オブジェクト 2-12
ORB の初期化 2-9
システム例外 5-5
ファクトリ 2-13
ユーザ例外 5-6
例外処理 5-3

CORBA C++ クライアント・アプリケーション 2-2
  DII の使用 4-6
  インターフェイス・リポジトリの使用 1-7
  オブジェクトのオペレーションの呼び出し 2-13
  オブジェクトへの初期リファレンスの解決 2-11
  開発プロセス 2-2
  記述 2-8
  システム例外 5-3
  静的起動の使用 2-6
  説明 1-2
  ビルド 2-14
  ファクトリの使用 2-12
  ユーザ例外 5-6
  呼び出し方式 2-6
  例外処理 5-3
CORBA Java クライアント・アプリケーション
  DII の使用 4-6
  インターフェイス・リポジトリの使用 1-7
  オブジェクトのオペレーションの呼び出し 2-13
  オブジェクトへの初期リファレンスの解決 2-11
  開発プロセス 2-3
  記述 2-8
  システム例外 5-7
  静的起動の使用 2-6
  説明 1-2
  ソフトウェアの要件 2-3
  必須フィールド 2-14
  ビルド 2-14
  ファクトリの使用 2-12
  ユーザ例外 5-9
  呼び出し方式 2-6
  例外処理 5-7
CORBA インターフェイス
  インターフェイス・リポジトリへのロード 3-4
  バインディングの作成 3-6
CORBA サービス・セキュリティ・サービス 1-17
CORBA システム例外
  説明 5-1
CourseSynposisEnumerator インターフェイス
  OMG IDL 2-4

D

DII
  Bootstrap オブジェクトの使い方 4-8
  CORBA インターフェイスのインターフェイス・リポジトリへのロード 4-7
  FactoryFinder オブジェクトの使用 4-8
  NVList の使用 4-10
  インターフェイス・リポジトリの使い方 4-17
  概念
    受信オプション 4-3
    要求の送信 4-3
    リクエスト・オブジェクト 4-3
  選択 4-2
  遅延同期通信 4-2
  要求の削除 4-17
  要求の作成 4-8
  要求の送信
    oneway 4-13
    遅延同期 4-12
    同期 4-11
    複数 4-13

F

FactoryFinder オブジェクト 2-12
  DII での使用 4-8
  コード例
    C++ 2-12
    Java 2-12
    Visual Basic 3-11

サーバ・アプリケーションの使用 2-15
図 1-13
説明 1-13
宣言
Visual Basic 3-9
メソッド 2-12

I

idl コマンド 2-2
CORBA C++ クライアント・アプリケーション 2-6
OMG IDL のコンパイル 2-6
形式 2-6
生成
クライアント・スタブ 2-7
スケルトン 2-7
説明 2-2
IDL コンパイラ
生成されるファイル 2-7
idl2ir コマンド
ActiveX クライアント・アプリケーションでの使用 3-3
インターフェイスのインターフェイス・リポジトリへのロード 3-4
インターフェイス・リポジトリへの追加 1-7
オートメーション環境オブジェクトのインターフェイス・リポジトリへのロード 3-4
構文 3-4
説明 1-8
InterfaceRepository オブジェクト
説明 1-16
ir2idl コマンド
OMG IDL ファイルの作成 1-7
説明 1-8
irdel コマンド
インターフェイス・リポジトリからの CORBA インターフェイスの削除 1-7
説明 1-8
ISL パラメータ 3-6
ActiveX クライアント・アプリケーションでの使用 3-10
BEA Application Builder での使用 3-7
CORBA クライアント・アプリケーションでの使用 2-10

J

JAR ファイル 2-14
Java
コード例
Bootstrap オブジェクト 2-11
Bootstrap オブジェクトの使い方 2-10
FactoryFinder オブジェクト 2-12
ORB の初期化 2-9
システム例外 5-8
ファクトリ 2-13
例外処理 5-7
Java アーカイブ・ファイル 2-14

N

NameService オブジェクト
説明 1-21
NotificationService オブジェクト
説明 1-20
NVList
DII での使用 4-10

O

OMG IDL
Basic サンプル・アプリケーション用 2-4
C++ へのマッピング 1-3
COM へのマッピング 1-4
CourseSynposisEnumerator インターフェイス 2-4
Java へのマッピング 1-3
Registrar インターフェイス 2-4

RegistrarFactory インターフェイス 2-4
コード例 2-4
コンパイル 2-6
説明 1-3
ユーザ例外の定義 5-1
ORB
初期化
C++ コード例 2-8
Java コード例 2-8
ORBid 2-9

R

Registrar インターフェイス
OMG IDL 2-4
RegistrarFactory インターフェイス
OMG IDL 2-4

S

SecurityCurrent オブジェクト
説明 1-17
プロパティ
Credentials 1-17
PrincipalAuthenticator 1-17

T

TOBJ_APPAUTH
説明 1-17
TOBJ_NOAUTH
説明 1-17
Tobj_SimpleEventsService オブジェクト
説明 1-20
TOBJ_SYSAUTH
説明 1-17
TransactionCurrent オブジェクト
トランザクション方針 1-19

U

UBBCONFIG ファイル
インターフェイス・リポジトリのサーバ・アプリケーションの起動 3-5

V

Visual Basic
コード例
Bootstrap オブジェクト 3-9
FactoryFinder オブジェクト 3-11
オペレーションの呼び出し 3-11, 3-12
ファクトリ 3-11
例外 5-10
宣言 3-9
Bootstrap オブジェクト 3-9
FactoryFinder オブジェクト 3-9
バインディングのタイプ・ライブラリのロード 3-8
例外処理 5-9

い

インターフェイス・リポジトリ
DII での使用 4-17
格納される情報 1-7
コマンド
idl2ir 1-7
ir2idl 1-7
irdel 1-7
サーバ・アプリケーションの起動 3-5
説明 1-7
ロード
オートメーション環境オブジェクト 3-4

お

オートメーション環境オブジェクト
TOBJIN.IDL 3-4
インターフェイス・リポジトリへのロード 3-4
宣言の記述 3-9

か

開発コマンド
- buildobjclient 2-2
- idl 2-2
- idl2ir 1-8
- ir2idl 1-8
- irdel 1-8

開発プロセス
- ActiveX クライアント・アプリケーション 3-2
- CORBA C++ クライアント・アプリケーション 2-2
- CORBA Java クライアント・アプリケーション 2-3
- DII 4-6

カスタマ・サポートへのお問い合わせ情報 1-ix

環境オブジェクト 1-9
- Bootstrap 1-9
- C++ 1-9, 2-2
- FactoryFinder 1-9
- Java 1-9
- NameService 1-11
- NotificationService 1-10
- SecurityCurrent 1-9
- Tobj_SimpleEventsService 1-10
- TransactionCurrent 1-9
- インターフェイス・リポジトリ 1-9
- オートメーション 1-9, 3-3
- 説明 1-9

関連情報 1-viii

く

クライアント・アプリケーション
- DII の使用の選択 4-2
- サポートされている 1-2

クライアント・スタブ
- 生成 1-6, 2-7
- 説明 1-6
- 定義 1-4

こ

コード例
- BEA Tuxedo ドメインへのログオン
  - Visual Basic 3-9
- Bootstrap オブジェクト
  - C++ 2-11
  - Java 2-11
  - Visual Basic 3-9
- FactoryFinder オブジェクト
  - C++ 2-12
  - Java 2-12
  - Visual Basic 3-11
- OMG IDL 2-4
- ORB
  - 初期化
    - C++ 2-8
    - Java 2-8
- オペレーションの呼び出し
  - C++ 2-13
  - Java 2-13
  - Visual Basic 3-11, 3-12
- システム例外
  - C++ 5-5
  - Java 5-8
  - Visual Basic 5-10
- 宣言
  - Visual Basic 3-9
- ファクトリ
  - C++ 2-13
  - Java 2-13
  - Visual Basic 3-11
- ユーザ例外
  - C++ 5-6
  - Java 5-9
  - Visual Basic 5-10

コンパイル
- OMG IDL 2-6

さ

サーバ・アプリケーション
- Bootstrap オブジェクトの使用 2-15

FactoryFinder オブジェクトの使用 2-15
クライアント・アプリケーションとして動作 2-15
サポート
テクニカル 1-ix
サンプル・アプリケーション
Basic 2-8

し

システム例外
説明 5-1

す

スケルトン
生成 2-7

せ

静的起動 1-4
クライアント・アプリケーション 2-6
クライアント・スタブの使用 1-4
しくみ 1-4
説明 1-4
製品マニュアルを印刷する 1-viii
セキュリティ
サポートされる認証レベル 1-17
説明 1-22

そ

ソフトウェアの要件
CORBA Java クライアント・アプリケーション 2-3

た

タイプ・ライブラリ
BEA Application Builder での作成 3-7
ディレクトリの場所 3-7, 3-8
バインディングの開発ツールへのロード 3-8
命名規則 3-7

ち

遅延同期通信
DII の使用 4-2

て

ディレクトリの場所
タイプ・ライブラリ 3-7
デプロイメント・パッケージ 3-13
デプロイメント・パッケージ
説明 3-13
ディレクトリの場所 3-13

と

動的起動
しくみ 1-4
図 1-4
説明 1-4
ドメイン
図 1-8
説明 1-8
通信の確立 2-8
ActiveX クライアント・アプリケーション 3-9
トランザクション方針
説明 1-19

に

認証レベル
BEA Tuxedo ソフトウェアでのサポート 1-17
TOBJ_APPAUTH 1-17
TOBJ_NOAUTH 1-17
TOBJ_SYSAUTH 1-17

は

バインディング
- 作成 3-6
- 説明 1-22
- デプロイ 3-13

ひ

ビュー
- オペレーションの呼び出し 3-11, 3-12
- 作成 3-6
- 説明 1-22
- 宣言の記述 3-9
- デプロイ 3-13

ビルド
- CORBA C++ クライアント・アプリケーション 2-14
- CORBA Java クライアント・アプリケーション 2-14

ふ

ファクトリ
- CORBA オブジェクトの作成 1-13
- コード例
  - C++ 2-13
  - Java 2-13
  - Visual Basic 3-11
- 説明 1-13
- 宣言
  - Visual Basic 3-9
- 宣言の記述 3-9
- 命名規則 1-14

ま

マニュアルの場所 1-viii

め

命名規則
- ActiveX 1-24
- ファクトリ 1-14

メソッド
- FactoryFinder オブジェクト 2-12

ゆ

ユーザ例外
- 説明 5-1

よ

呼び出し方式
- CORBA クライアント・アプリケーションでの使用 2-6
- 静的 1-4
- 動的 1-4

り

リクエスト・オブジェクト
- 作成 4-8
- 説明 4-3
- 引数の設定 4-10

れ

例外
- CORBA システム例外 5-1
- 概念 5-1
- システム 5-1
- ユーザ 5-1

例外処理
- C++ 5-3
- Java 5-7
- Visual Basic 5-9

例外のキャッチ
- C++ 5-5
- Java 5-8
- Visual Basic 5-10