BEA Tuxedo®

BEA Tuxedo CORBA プログラミング・リファレンス

BEA Tuxedo 8.1
2003 年 1 月

Copyright

Restricted Rights Legend

Trademarks or Service Marks

目次

4. 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

5. FactoryFinder インターフェイス

11. 共同クライアント / サーバ

12. 開発コマンド

13. OMG IDL 文の C++ へのマッピング

14. CORBA API

15. サーバ側のマッピング

このマニュアルについて

このマニュアルでは、BEA Tuxedo CORBA C++ アプリケーション・プログラミング・インターフェイス (API) 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 「OMG IDL 構文と C++ IDL コンパイラ」では、Object Management Group (OMG) インターフェイス定義言語 (IDL) および OMG IDL の拡張機能について説明します。
- 第 2 章 「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)」では、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) について説明します。
- 第 3 章 「TP フレームワーク」では、BEA Tuxedo TP フレームワーク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) について説明します。
- 第 4 章 「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」では、ブートストラップ処理メカニズムについて説明します。
- 第 5 章 「FactoryFinder インターフェイス」では、FactoryFinder インターフェイスについて説明します。
- 第 6 章 「セキュリティ・サービス」では、セキュリティ・サービスについて説明します。
- 第 7 章 「トランザクション・サービス」では、トランザクション・サービスについて説明します。
- 第 8 章 「ノーティフィケーション・サービス」では、ノーティフィケーション・サービスについて説明します。

- 第 9 章 「要求レベルのインターセプタ」では、要求レベルのインターセプタについて説明します。
- 第 10 章 「CORBA インターフェイス・リポジトリのインターフェイス」では、インターフェイス・リポジトリのインターフェイスについて説明します。
- 第 11 章「共同クライアント / サーバ」では、共同クライアント / サーバ・アプリケーションおよび BEAWrapper Callback API をプログラミングする方法について説明します。
- 第 12 章 「開発コマンド」では、UNIX および Windows プラットフォーム用のビルドおよび管理コマンドについて説明します。
- 第 13 章 「OMG IDL 文の C++ へのマッピング」では、OMG IDL 文の C++ へのマッピングについて説明します。
- 第 14 章 「CORBA API」では、CORBA API について説明します。
- 第 15 章 「サーバ側のマッピング」では、OMG IDL 文の C++ へのサーバ側のマッピングについて説明します。

対象読者

このマニュアルは、BEA Tuxedo CORBA C++ API を使用してクライアント・アプリケーション、共同クライアント / サーバ・アプリケーション、およびオブジェクト・インプリメンテーションを記述するアプリケーション開発者を対象としています。CORBA と、C++ および Java プログラミングに読者が精通していることを前提として書かれています。

e-docs Web サイト

BEA 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。BEA Tuxedo 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、Java 2 Enterprise Edition (J2EE)、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「BEA TuxedoBibliography」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 OMG IDL 構文と C++ IDL コンパイラ

Object Management Group (OMG) インターフェイス定義言語 (IDL) は、クライアント・オブジェクトから呼び出し、オブジェクトのインプリメンテーションで提供するインターフェイスを記述します。OMG IDL インターフェイス定義では、各オペレーションのパラメータをすべて指定して、インターフェイスのオペレーションを使用するクライアント・アプリケーションの開発に必要な情報を提供します。

クライアント・アプリケーションは、OMG IDL 文のマッピング先として定義されている言語で作成されます。OMG IDL 文とクライアント言語の構造体とのマッピング方法は、クライアント言語で利用可能な機能によって異なります。たとえば、OMG IDL の例外を、例外の概念を持たない言語の構造体、または概念を持つ言語の例外にマップすることができます。

OMG IDL 文は、C++ 文と同じ文法規則に従います。ただし、分散概念をサポートするための new キーワードが導入されています。また、OMG IDL 文は、標準的な C++ プリプロセッサ機能および OMG IDL 固有のプラグマをフル・サポートしています。

注記　pragma version 文を使用する場合は、対応するインターフェイス定義の後に配置する必要があります。次の例で正しい使い方を示します。

```
module A
{
    interface B
    {
#pragma version B "3.5"
         void op1();
```

```
    };
};
```

OMG IDL の文法は、オペレーション呼び出しメカニズムをサポートするための構造体が追加された ANSI C++ のサブセットです。OMG IDL は宣言型言語であり、定数、型、およびオペレーションの宣言に関して C++ の構文をサポートしています。ただし、アルゴリズミックな構造体または変数は含まれません。

OMG IDL の文法の詳細については、「Common Object Request Broker: Architecture and Specification Revision 2.4」の第 3 章「OMG IDL Syntax and Semantics」を参照してください。

OMG IDL の文法はサポートされていますが、次の型宣言および関連リテラルを除きます。

- `native`

注記　CORBA 2.4 では `native` 型宣言がオブジェクト・アダプタ用として記述されているので、この型は、コールバックをサポートしているクライアントの `PortableServer` モジュール、つまり共同クライアント / サーバでのみ利用可能です。

- `long double`
- `fixed`

これらのデータ型を IDL 定義で使用しないでください。

注記　BEA Tuxedo CORBA のリリース 8.0 では、`long long`、`unsigned long long`、`wchar`、および `wstring` データ型のサポートが追加されました。

また、OMG IDL ファイルをコンパイルまたはロードする場合にも独自のマクロを定義できます。

表 1-1 では、各プラットフォーム用の定義済みマクロについて説明します。

表 1-1 定義済みのマクロ

マクロ識別子	マクロが定義されているプラットフォーム
__unix__	Sun Solaris、HP-UX、Tru64 UNIX、および IBM AIX
__osf1__	Tru64 UNIX
__sun__	Sun Solaris
__hpux__	HP-UX
__aix__	IBM AIX
__win_nt__	Microsoft Windows 2000 および NT

表 1-2 は、BEA Tuxedo 8.0 の C++ IDL コンパイラの制約について説明し、推奨される対策を示しています。

表 1-2 C++ IDL コンパイラ

制約	**OMG IDL コンテキスト文字列でワイルドカードを使用すると、警告メッセージが表示されます。**
説明	警告メッセージは、ワイルドカード文字を含むコンテキスト文字列をオペレーションの定義で使用した場合に、C++ IDL コンパイラによって表示されます。コンテキスト文字列を OMG IDL オペレーション定義で指定すると、次の警告メッセージが表示されます。

```
void op5() context("*");
                   ^
LIBORBCMD_CAT:131: INFO: ‘*’ is a non-standard
                         context property.
```

表 1-2 C++ IDL コンパイラ (続き)

	対策	OMG CORBA 仕様では、コンテキスト文字列の先頭に英数字を使用しなければならないかどうかについて明確に指定されていません。 この警告メッセージは、解釈によっては OMG CORBA 仕様に準拠していないことを通知するためのものです。上記のように、すべての文字列をコンテキスト文字列として指定する場合、OMG CORBA 仕様では、先頭に英文字を使用した文字列からなるカンマ区切りのリストが必要です。 注記　上記の例は OMG CORBA に準拠していませんが、BEA Tuxedo ソフトウェアによってユーザの意図したとおりに処理されます。
制約	**OMG IDL コンテキスト文字列でワイルドカードを使用すると、警告メッセージが表示されます。**	

	説明	警告メッセージは、ワイルドカード文字を含むコンテキスト文字列をオペレーションの定義で使用した場合に、C++ IDL コンパイラによって表示されます。コンテキスト文字列を OMG IDL オペレーション定義で指定すると、次の警告メッセージが表示されます。
		`void op5() context("*");`
		`^`
		`LIBORBCMD_CAT:131: INFO: ‘*’ is a non-standard context property.`

	対策	OMG CORBA 仕様では、コンテキスト文字列の先頭に英数字を使用しなければならないかどうかについて明確に指定されていません。 この警告メッセージは、解釈によっては OMG CORBA 仕様に準拠していないことを通知するためのものです。上記のように、すべての文字列をコンテキスト文字列として指定する場合、OMG CORBA 仕様では、先頭に英文字を使用した文字列からなるカンマ区切りのリストが必要です。 注記　上記の例は OMG CORBA に準拠していませんが、BEA Tuxedo ソフトウェアによってユーザの意図したとおりに処理されます。
制約	**C++ IDL コンパイラが一部のデータ型をサポートしていません。**	

表 1-2 C++ IDL コンパイラ (続き)

	説明	現在、C++ IDL コンパイラは、CORBA 仕様バージョン 2.4 で定義されている以下のデータ型をサポートしていません。 ■ `native` ■ `fixed` ■ `long double`
	対策	これらのデータ型を IDL 定義で使用しないようにします。
制約	**識別子で特定の文字列を使用すると、C++ IDL コンパイラが不正なコードを生成することがあります。**	

	説明	識別子に以下の文字列を使用すると、コードが正しく生成されず、コードのコンパイル時にエラーが発生します。
		`get_`
		`set_`
		`Impl_`
		`_ptr`
		`_slice`

	対策	これらの文字列を識別子に使用しないようにします。
制約	**大文字と小文字の区別に関して、IDL コンパイラの動作に一貫性がありません。**	
	説明	CORBA 標準によると、大文字と小文字の区別だけが異なる IDL 識別子は競合と見なされ、コンパイル・エラーが発生します。現在、BEA Tuxedo IDL コンパイラには、C++ バインディングに関して、名前の競合を検出して報告しない場合があるという制限があります。
	対策	大文字と小文字の区別だけが異なる IDL 識別子を使用しないようにします。
制約	**C++ IDL typedef の問題があります。**	

表 1-2 C++ IDL コンパイラ (続き)

説明	C++ IDL コンパイラは、次の場合にコンパイルできないコードを生成します。
	■ char または boolean 型の IDL 変数を宣言した場合
	■ かつ上記の型が複数回エイリアスになる場合
	たとえば、次の IDL コードから生成された C++ コードはコンパイルできません。
	`module X`
	`{`
	`typedef boolean a;`
	`typedef a b;`
	`interface Y`
	`{`
	`attribute b Z;`
	`};`
	`};`
	C++ コンパイラは、「operator <<」が不明確であり、char 型の「operator>>」がないことを示すエラーを報告します。これらのエラーは、typedef のレベルが複数になっているために発生します。つまり、X::a という中間の型定義があるため、C++ コンパイラは、型 X::b と CORBA::Boolean を関連付けることができません。

対策	char または boolean 型を定義する場合は、間接レベルを 1 つにします。上記の IDL の例では、属性「X::Z」は、ユーザ型「X::b」ではなく、標準型「boolean」またはユーザ型「X::a」を使用して定義されます。

2 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)

BEA Tuxedo CORBA TP フレームワーク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) には、オブジェクトを活性化および非活性化するためのコールバック・メソッドが用意されています。これらのメソッドを使用すると、アプリケーション・コードに CORBA オブジェクトの柔軟な状態管理スキーマを実装できます。

状態管理とは、オブジェクトの活性化および非活性化時にオブジェクトの状態の保存および復元を制御する方法のことです。状態管理は、サーバの性能およびリソース使用量に影響を与える、活性化されたオブジェクトの有効期間にも影響します。TP フレームワークの外部 API には、`activate_object()` および `deactivate_object()` メソッドが含まれています。これらのメソッドは、状態管理コードを配置可能な場所を示します。また、TP フレームワーク API には、ユーザがオブジェクトを活性化するタイミングを制御できるようにするための `deactivateEnable()` メソッドが含まれています。活性化されたオブジェクトのデフォルトの有効期間は、OMG IDL のコンパイル時にインプリメンテーションに割り当てた方針によって制御されます。

CORBA オブジェクトが活性化されている間、オブジェクトの状態はサーバント内に格納されます。この状態は、オブジェクトが最初に呼び出されたとき、つまりオブジェクト・リファレンスの作成後に CORBA オブジェクトに対してメソッドが最初に呼び出されたときと、オブジェクトが非活性化されて以降の呼び出しで初期化しなければなりません。

CORBA オブジェクトが非活性化されている間、オブジェクトの状態をサーバントが活性化されていたプロセスの外部に保存する必要があります。オブジェクトが活性化されたときには、オブジェクトの状態を復元する必要があります。オブジェクトの状態は、共用メモリ、ファイル、データベースなどに保存できます。プログラマは、オブジェクトの状態の構成要素と、オブジェクトを非活性化する前に何を保存し、オブジェクトを活性化した後に何を復元するかを決定する必要があります。

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) を使用すると、活性化方針を設定して、各インプリメンテーションで活性化されたオブジェクトの有効期間を設定できます。ICF ファイルでは、活性化方針を指定することで、オブジェクトの状態を管理します。活性化方針は、CORBA オブジェクトがメモリ内で活性化している期間を決定します。CORBA オブジェクトがポータブル・オブジェクト・アダプタ (POA) 内で活性化されているのは、POA のアクティブ・オブジェクト・マップにオブジェクト ID と既存のサーバントを関連付けるエントリが入っている場合です。オブジェクトを非活性化すると、オブジェクト ID と活性化されたサーバントとの関連付けが削除されます。

ICF の構文

ICF の構文は次のとおりです。

```
[#pragma activation_policy method|transaction|process]
[#pragma transaction_policy never|ignore|optional|always]
[#pragma concurrency_policy user_controlled|system_controlled]
[Module module-name {]
   implementation [implementation-name]
     {
     implements (module-name::interface-name);
     [activation_policy (method|transaction|process);]
```

```
   [transaction_policy (never|ignore|optional|always);]
   [concurrency_policy (user_controlled|system_controlled);]
   };
[};]
```

pragmas
: 任意の 3 つのプラグマを使用すると、明示的な activation_policy、transaction_policy、または concurrency_policy 文を持たないすべてのインプリメンテーション用の ICF 全体のデフォルト方針として、特定の方針を設定できます。この機能により、プログラマは、インプリメンテーションごとに方針を指定する必要がなくなります。また、デフォルトを上書きすることもできます。

Module module-name
: module-name 変数は、OMG IDL ファイルでオプションの場合はオプションです。この変数は、スコープとグループを指定する場合に使用します。この変数を使用する場合は、OMG IDL ファイル内の使い方と一貫性を持たせる必要があります。

implementation-name
: この変数はオプションで、サーバ名またはサーバ内のクラス名として使用します。プログラマが指定していない場合は、interface-name に _i を追加して名前が作成されます。

implements (module-name::interface-name)
: この変数は、活性化方針およびトランザクション方針の適用対象となるモジュールおよびインターフェイスを識別します。

activation_policy
: 活性化方針の詳細については、「活性化方針」を参照してください。

transaction_policy
: トランザクション方針の詳細については、「トランザクション方針」を参照してください。

concurrency_policy
: 同時実行方針の詳細については、「パラレル・オブジェクト」を参照してください。

ICF ファイルのサンプル

リスト 2-1 は、ICF ファイルの例を示しています。

コード リスト 2-1 ICF の例

```
module POA_University1
    {
    implementation CourseSynopsisEnumerator_i
        {
        activation_policy ( process );
        transaction_policy ( optional );
        implements ( University1::CourseSynopsisEnumerator );
        };

    };

module POA_University1
    {
    implementation Registrar_i
        {
        activation_policy ( method );
        transaction_policy ( optional );
        implements ( University1::Registrar );
        };

   };

module POA_University1
   {
   implementation RegistrarFactory_i
       {
       activation_policy ( process );
       transaction_policy ( optional );
       implements ( University1::RegistrarFactory );
       };

   };
```

ICF ファイルの作成

ICF ファイルの作成には、手動でコーディングする方法と、genicf コマンドを使用して OMG IDL ファイルから作成する方法があります。genicf コマンドの構文およびオプションについては、『BEA Tuxedo コマンド・リファレンス』を参照してください。

章

3 TP フレームワーク

ここでは、以下の内容について説明します。

- 単純なプログラミング・モデル . この節では、次の内容について説明します。
  - 制御フロー
  - オブジェクトの状態管理
  - トランザクションの統合
  - オブジェクトのハウスキーピング
  - 高レベルのサービス
- 状態管理 . この節では、次の内容について説明します。
  - 活性化方針
  - アプリケーション制御の活性化および非活性化
  - サーバントの存続期間
  - オブジェクトの状態の保存と復元
- トランザクション . この節では、次の内容について説明します。
  - トランザクション方針
  - トランザクションの初期化
  - トランザクションの終了
  - トランザクションの一時停止と再開
  - トランザクションに関する制約
  - SQL とグローバル・トランザクション

- トランザクションの結果に関する判断
- トランザクションのタイムアウト

■ パラレル・オブジェクト

■ TP フレームワーク API

■ エラー、例外、およびエラー・メッセージ

BEA Tuxedo CORBA TP フレームワークでは、ユーザが高性能の TP アプリケーション用のサーバを作成することを可能にするプログラミング TP フレームワークを提供します。ここでは、TP フレームワークのプログラミング・モデルおよびアプリケーション・プログラミング・インターフェイス (API) について詳しく説明します。この API については、『BEA Tuxedo CORBA サーバ・アプリケーションの開発方法』でも説明しています。

BEA Tuxedo CORBA サーバを開発する場合、TP フレームワークは必須です。この要件は今後のリリースで緩和される予定ですが、多くの場合、TP フレームワークはアプリケーションの主要な構成部分として使用されると考えられます。

BEA Tuxedo では、ロード・バランシング、トランザクション機能、および管理機能を備えたインフラストラクチャを提供します。TP フレームワークで使用される基本 API は、BEA が拡張した CORBA API です。TP フレームワークの API は、顧客にエクスポーズされます。BEA Tuxedo ATMI は、TP フレームワークの API と混在して利用可能なオプションの API です。この API を使用すると、CORBA サーバと ATMI サーバが混在する環境で分散アプリケーションをデプロイできます。

BEA Tuxedo CORBA 以前、ORB 製品は大規模環境では BEA Tuxedo の性能に及びませんでした。BEA Tuxedo システムは、1 秒あたり数百のトランザクションを処理するアプリケーションをサポートしています。こうしたアプリケーションは、各要求で使用するシステム・リソースを最小限に抑え、したがってスループットおよびコスト・パフォーマンスを最大限に引き出す、BEA Tuxedo のステートレス・サービス・プログラミング・モデルを使用してビルドされます。

現在では、BEA Tuxedo CORBA と TP フレームワークによって、BEA Tuxedo ATMI アプリケーションと同等の性能を持つ CORBA アプリケーションを開発できます。BEA Tuxedo CORBA サーバは、CORBA プログラ

ミング・モデルを使用しながら、BEA Tuxedo ステートレス・サービス・プログラミング・モデルに近いスループット、応答時間、およびプライス・パフォーマンスを実現します。

単純なプログラミング・モデル

TP フレームワークでは、単純で便利な CORBA オブジェクトのさまざまなインプリメンテーションのサブセットを選択できます。使用できるのは、サーバ側オブジェクトのインプリメンテーションを開発する場合のみです。クライアント側 CORBA ORB を使用する場合、クライアントは、TP フレームワークが管理するサーバ側インプリメンテーションを持つ CORBA オブジェクトと対話します。クライアントでは TP フレームワークの存在が意識されません。非 BEA Tuxedo サーバ環境で実行される CORBA オブジェクトにアクセスするように作成されたクライアントは、クライアント・インターフェイスの変更や制約もなく、BEA Tuxedo サーバ環境で実行される同じ CORBA オブジェクトにアクセスできます。

TP フレームワークでは、CORBA ポータブル・オブジェクト・アダプタ (POA) よりも使いやすく、理解も簡単で、特にエンタープライズ・アプリケーション向けに準備されたサーバ環境および API が提供されます。これは、単純なサーバ・プログラミング・モデルと、ORBIX や VisiBroker などの ORB を使用していたプログラマにはなじみのある従来の CORBA モデルのインプリメンテーションです。

TP フレームワークを使用すると、次の方法でサーバ環境の複雑さを抑えて、BEA Tuxedo CORBA サーバのプログラミングが容易になります。

- TP フレームワークは、POA およびネーミング・サービスとのやり取りをすべて処理します。そのため、アプリケーション・プログラマが POA またはネーミング・サービスのインターフェイスの知識を持っている必要はありません。
- TP フレームワークはシングル・スレッド処理されます。つまり、スレッド・セーフなインプリメンテーションを作成せずに済むように、1 つの CORBA オブジェクトに対する要求は一度に 1 つしか処理されません。

- CORBA オブジェクトは、1 つのオブジェクト ID と 1 つのサーバントとの関連付けについて矛盾しないように、一度に 1 つのトランザクションだけに関与できます。

TP フレームワークでは以下の機能を利用できます。

- 制御フロー
- オブジェクトの状態管理
- トランザクションの統合
- オブジェクトのハウスキーピング
- 高レベルのサービス

制御フロー

TP フレームワークは、ORB および POA と連携して、次の方法でアプリケーション・プログラムのフローを制御します。

- サーバを起動またはシャットダウンする場合に、サーバ・マシンを制御し、必要に応じて TP フレームワークで定義されているクラスのコールバック・メソッドを呼び出します。これにより、アプリケーション・プログラマは、ORB および POA の初期化とトランザクションの調整、リソース・マネージャ、およびシャットダウン時のオブジェクトの状態に関連する複雑な対話を考慮する必要がなくなります。
- クライアントの要求を受信して処理した場合のオブジェクトの活性化および非活性化をスケジューリングします。これにより、アプリケーション・プログラマがオブジェクトの活性化および非活性化に関する複雑な管理作業から解放されるだけでなく、ミッション・クリティカルなタスクで重要な役割を果たす、TP モニタ・インフラストラクチャの強力なロード・バランシング機能を使用できるようになります。

オブジェクトの状態管理

TP フレームワーク API には、アプリケーション・コードに CORBA オブジェクトの柔軟な状態管理スキーマを実装するためのコールバック・メソッドが用意されています。状態管理では、オブジェクトを非活性化したり活性化したりするときのオブジェクトの状態を保存および復元する必要があります。状態管理は、サーバの性能およびリソース使用量に影響を与える、活性化されたオブジェクトの有効期間にも関係します。活性化されたオブジェクトのデフォルトの有効期間は、IDL のコンパイル時にインプリメンテーションに割り当てた方針によって制御されます。

トランザクションの統合

TP フレームワークのトランザクション統合には次の機能があります。

- CORBA オブジェクトはグローバル・トランザクションに参加できます。
- トランザクションに参加するオブジェクトを、トランザクション・バウンドの活性化方針で指定したトランザクションの有効期間に合わせてメモリ内に残る状態を持つオブジェクトとして実装すると、クライアントの応答時間を短縮できます。
- トランザクションに参加する CORBA オブジェクトは、トランザクションの処理中、または 2 フェーズ・コミット・アルゴリズムを実行する直前かつすべてのトランザクション処理が完了した後に、トランザクションの結果に関与できます。
- トランザクションは、クライアントに対して透過的にサーバ上で自動的に初期化されます。

オブジェクトのハウスキーピング

TP フレームワークでは、サーバのシャットダウン時に、サーバが関与しているすべてのトランザクションがロールバックされ、活性化されているすべての CORBA オブジェクトが非活性化されます。

高レベルのサービス

TP フレームワーク API の TP インターフェイスには、オブジェクトの登録とユーティリティ関数を実行するメソッドが用意されています。以下のサービスが提供されます。

- オブジェクト・リファレンスの作成
- ファクトリ・ベース・ルーティングのサポート
- ORB などのシステム・オブジェクトのアクセサ
- FactoryFinder でのファクトリの登録および登録の削除
- アプリケーション制御の活性化および非活性化
- ユーザ・ログ機能

こうした高レベル・サービスのメソッドは、開発者が CORBA POA、CORBA ネーミング・サービス、および BEA Tuxedo API を理解しなくても、基となるインプリメンテーションとして使用できるようにすることを目的としています。基となる API 呼び出しを高レベルのメソッドのセットと一緒にカプセル化することで、プログラマは、より複雑な基本機能を理解したり使用したりすることなく、ビジネス・ロジックの実現に集中することが可能となります。

状態管理

状態管理では、オブジェクトを非活性化したり活性化したりするときのオブジェクトの状態を保存および復元する必要があります。状態管理は、サーバの性能およびリソース使用量に影響を与える、活性化されたオブジェクトの有効期間にも関係します。TP フレームワークの外部 API には、`activate_object` および `deactivate_object` メソッドが用意されています。これらのメソッドは、状態管理コードを配置可能な場所を示します。

活性化方針

TP フレームワークでは、状態管理は活性化方針によって提供されます。この方針は、サーバントの作成および破棄ではなく、特定の IDL インターフェイスに対するサーバントの活性化および非活性化を制御します。この方針が適用されるのは、TP フレームワークを使用する CORBA オブジェクトのみです。

活性化方針は、CORBA オブジェクトがメモリ内で活性化しているデフォルトの期間を決定します。CORBA オブジェクトが POA 内で活性化されているのは、POA のアクティブ・オブジェクト・マップにオブジェクト ID と既存のサーバントを関連付けるエントリが入っている場合です。オブジェクトを非活性化すると、オブジェクト ID と活性化されたサーバントとの関連付けが削除されます。選択できる活性化方針は、`method` (デフォルト)、`transaction`、`process` のいずれかです。

注記　活性化方針は、OMG IDL のコンパイル時に設定する ICF ファイルで設定されます。ICF ファイルの詳細については、「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)」を参照してください。

各活性化方針の内容は次のとおりです。

- `method` (デフォルトの活性化方針)

  CORBA オブジェクトの活性化、つまりオブジェクト ID とサーバントとの関連付けは、メソッドが終了するまで維持されます。メソッドが終了すると、オブジェクトは非活性化されます。オブジェクト・リファレンスで次のメソッドが呼び出されると、CORBA オブジェクトは活性化されます。つまり、オブジェクト ID が新しいサーバントに関連付けられます。この動作は、BEA Tuxedo のステートレス・サービスとほぼ同じです。

- transaction

  CORBA オブジェクトの活性化、つまりオブジェクト ID とサーバントとの関連付けは、トランザクションが終了するまで維持されます。トランザクション中は、オブジェクトの複数のメソッドを呼び出すことができます。オブジェクトは最初のメソッド呼び出しの前に活性化され、次のいずれかの方法で非活性化されます。

  - オブジェクトが活性化されたときにユーザが開始したトランザクションが有効な場合、オブジェクトは、トランザクションがコミットまたはロールバックされる、またはサーバが通常の方法でシャットダウンされる、という 2 つの処理のうち、いずれか先に行われた処理に伴って非活性化されます。2 番目の処理には、tmshutdown または tmadmin コマンドが使用されます。これらのコマンドについては、オンライン・マニュアルの『BEA Tuxedo コマンド・リファレンス』を参照してください。
  - TP オブジェクトが活性化されたときにユーザが開始したトランザクションが有効ではない場合、TP オブジェクトはメソッドが終了すると非活性化されます。

  transaction 活性化方針では、2 フェーズ・コミット・アルゴリズムを実行する前に、オブジェクトがトランザクションの結果に関して支持するかどうかを判断できます。オブジェクトがトランザクションのロールバックを支持する場合は、Tobj_ServantBase::deactivate_object メソッドの Current.rollback_only() を呼び出します。トランザクションのコミットを支持する場合は、Current.rollback_only() を呼び出しません。

  注記　これは、BEA Tuxedo の会話型サービスに似たリソース割り当てモデルです。ただし、このモデルの方が、システム・リソースの使用量が少ないという点で、BEA Tuxedo の会話型サービスよりもコストが小さくなります。この理由は、BEA Tuxedo ORB のマルチコンテキスト・ディスパッチ・モデル、つまり、1 つのサーバ用のサーバントがメモリ内に同時に多数存在するモデルを使用しているからです。このモデルでは、多数のクライアントにサービスし、同時に活性化されている多数のサーバントが 1 つのサーバ・プロセスを共有できます。BEA Tuxedo システムでは、プロセスは、会話の有効期間だけ 1 つのクライアント専用となり、1 つのサービスにのみ割り当てられます。

- process

CORBA オブジェクトは、非活性化状態で呼び出されたときに活性化され、デフォルトではプロセスが終了するまでその状態を持続します。

注記　TP フレームワーク API には、activation policy が process に設定されたオブジェクトを非活性化するタイミングをアプリケーションで制御するためのインターフェイス・メソッド (TP::deactivateEnable) が用意されています。このメソッドの説明については、「TP::deactivateEnable()」を参照してください。

アプリケーション制御の活性化および非活性化

通常、活性化と非活性化は、既に説明したように、TP フレームワークによって決定されます。ここで説明するテクニックでは、代替メカニズムの使い方を示します。アプリケーションでは、特定の方針が設定されたオブジェクトを明示的に活性化および非活性化するタイミングを制御できます。

明示的な活性化

アプリケーション・コードでは、process 活性化方針を使用するオブジェクトに関して、TP フレームワークのオン・デマンド活性化機能を無効にすることができます。アプリケーションでは、TP::create_active_object_reference 呼び出しを使用して、オブジェクトを「事前活性化」、つまり呼び出しの前に活性化することができます。

事前活性化のしくみは次のとおりです。アプリケーションは、オブジェクト・リファレンスを作成する前に、サーバントをインスタンス化して、その状態を初期化します。アプリケーションは TP::create_active_object_reference を使用して、オブジェクトをアクティブ・オブジェクト・マップに追加、つまりサーバントと ObjectId を関連付けます。最初の呼び出しが行われると、TP フレームワークが、オブジェクト・リファレンスを作成したプロセスに直ちに要求を転送してから、既存のサーバントに転送します。この際、オブジェクトに対する 2 番目以降の呼び出しと同じように、Server::create_servant に次いでサーバントの activate_object メソッドを呼び出す必要はありません。こうしたオブジェクトのオブジェクト・リファレンスは別のサーバを指さないので、活性化されている限り、オブジェクトがオン・デマンドで活性化されることはありません。

事前活性化されたオブジェクトには process 活性化方針が設定されているので、プロセスの終了または TP::deactivateEnable 呼び出しのいずれかのイベントが発生するまで、オブジェクトは活性化されたままとなります。

使用上の注意

事前活性化は、アプリケーションが共用メモリなどを使用して状態を初期化して、初期状態のサーバントを同じプロセスで確立する必要がある場合に特に有用です。状態にポインタ、オブジェクト・リファレンス、または複雑なデータ構造が含まれている場合、後で別のプロセスで状態が初期化されるまで待機する処理は非常に難しくなるからです。
TP::create_active_object_reference を使用すると、事前活性化されたオブジェクトは、事前活性化を実行したコードと必ず同じプロセスにあります。事前活性化によってリソースがあらかじめ割り当てられるので、便利な手法であっても、事前活性化を多用することは控える必要があります。ただし、必要に応じて適切に使用すれば、事前活性化はその他の方法よりもはるかに効率的です。

適切な使い方の例として、「イテレータ」パターンを使用するオブジェクトがあります。たとえば、database_query メソッドから電話帳の内容のような長い項目リストがアンバウンディッド IDL シーケンスで返される可能性があるとします。メッセージ・サイズも必要なメモリ量も非常に大きくなるので、こうした項目をすべてシーケンスで返すことは非効率的です。

イテレータ・パターンを使用するオブジェクトは、リストを取得する最初の呼び出しで、一定数の項目をシーケンスで返し、さらに要素を取得する場合に呼び出せる「イテレータ」オブジェクトへのリファレンスも返します。イテレータ・オブジェクトは初期オブジェクトによって初期化されます。つまり、初期オブジェクトはサーバントを作成してその状態を設定し、反復処理が長い項目リストのどの位置で止まっているか (データベース、問い合わせパラメータ、カーソルなどを指すポインタ) をトラッキングします。

初期オブジェクトは、TP::create_active_object_reference を使用して、イテレータ・オブジェクトを事前初期化します。また、初期オブジェクトは、そのオブジェクトへのオブジェクト・リファレンスを作成してクライアントに返します。クライアントは、イテレータ・オブジェクトを繰り返し呼び出して、たとえば、呼び出しごとにリスト内の 100 項目を取得します。こうした場合の事前初期化の利点は、複雑な状態を使用できるということで

す。通常は、初期オブジェクトに制御がある間に、すべての情報をコンテキスト (呼び出しフレーム) に持っているメソッドを使用して、そうした状態を最初に設定する方法が最も簡単です。

クライアントがイテレータ・オブジェクトの操作を終了したら、初期オブジェクトで最終メソッドを呼び出して、イテレータ・オブジェクトを非活性化します。初期オブジェクトは、`TP::deactivateEnable` メソッドを呼び出すイテレータ・オブジェクトのメソッドを呼び出すことで、イテレータ・オブジェクトを非活性化します。つまり、イテレータ・オブジェクトは、`TP::deactivateEnable` メソッドを自身に対して呼び出します。

注意事項

通常、この方法で事前活性化されたオブジェクトの場合、クラッシュすると状態を回復することはできません。最初の遅延活性化で設定するには、状態が複雑すぎるか、一貫性を維持できないと考えられるからです。これは、基本的にオブジェクトが 1 回の活性化期間のみ有効であることを示しており、効果的なオブジェクトのテクニックです。

ただし、「1 回」という使い方のために問題が発生する場合があります。クライアントが状態を含むプロセスにつながるオブジェクト・リファレンスを保持しており、クラッシュするとその状態を再度作成することができないので、クライアントの次の呼び出しが自動的にオブジェクトを新しく活性化することがないように注意を払う必要があります。新しく活性化するとオブジェクトが不当な状態を持つことになるからです。

この問題を解決する方法は、オブジェクトが TP フレームワークによって自動的に活性化されないようにすることです。`activate_object` 呼び出しの結果として `TobjS::ActivateObjectFailed` 例外を TP フレームワークに提供した場合、TP フレームワークは活性化を実行せず、`CORBA::OBJECT_NOT_EXIST` 例外をクライアントに返します。クライアントにはイテレータ (または類似) パターンについての知識があるので、こうした状況について想定しています。クライアントは反復を再度開始するように準備する必要があります。

注記　TP フレームワーク自体でオブジェクト・リファレンスが有効ではなくなったことを検出できるようになれば、こうした保護措置は不要になります。特に、`activate_object` メソッドが呼び出される可能性

を考慮せずに済みます。TP フレームワークが変更される場合、activate_object は呼び出されず、フレームワーク自体が OBJECT_NOT_EXIST 例外を生成するようになります。

自己非活性化

process 活性化方針を設定したオブジェクトを事前活性化できるように、process 活性化方針を設定したオブジェクトを非活性化するよう要求することもできます。事前活性化する機能と非活性化を要求する機能は独立しています。つまり、オブジェクトがどのように活性化されたかに関係なく、オブジェクトを明示的に非活性化することができます。

アプリケーションのメソッドは、TP::deactivateEnable 使用して、オブジェクトを非活性化するように要求できます。TP::deactivateEnable を呼び出して、オブジェクトが非活性化された場合、CORBA オブジェクトに対する以降の呼び出しによって、以前に活性化されたように、同じプロセスで再度活性化されるとは限りません。ObjectId とサーバントとの関連付けは、CORBA オブジェクトが活性化されてから、サーバ・プロセスがシャットダウンされるか、アプリケーションが TP::deactivateEnable を呼び出すまで持続します。関連付けが解除された後、オブジェクトをもう一度呼び出すと、BEA Tuxedo コンフィギュレーション・パラメータで許可された場所で再度活性化することができます。

TP::deactivateEnable には 2 つの形式があります。最初の形式 (パラメータなし) では、実行中のオブジェクトは、呼び出しが行われたメソッドの終了後に非活性化されます。非活性化するかどうかはオブジェクト自身が決定します。通常、この非活性化は、「サインオフ」シグナルとして機能するメソッド呼び出しで行われます。

2 番目の形式の TP::deactivateEnable では、サーバは、オブジェクトが実行中かどうかにかかわりなく、活性化されている任意のオブジェクトを非活性化するよう要求できます。つまり、どのサーバもオブジェクトを非活性化するよう要求できます。この形式では、非活性化するオブジェクトを識別するパラメータを指定します。transaction 活性化方針が設定されたオブジェクトを明示的に非活性化することはできません。トランザクションが終了するまで、そうしたオブジェクトを安全に非活性化することができないからです。

TP::deactivateEnable 呼び出しで、TP フレームワークはサーバントの deactivate_object メソッドを呼び出します。TP フレームワークが deactivate_object を呼び出す正確なタイミングは、非活性化されるオブジェクトの状態によって異なります。オブジェクトが実行中でない場合、TP フレームワークは、制御が呼び出し元に戻る前に、オブジェクトを非活性化します。オブジェクトが実行中の場合もあります。これは常にパラメータを持たない TP::deactivateEnable の場合です。この形式では、実行中のオブジェクトが参照されるからです。この場合、TP::deactivateEnable には、オブジェクトが直ちに非活性化されたかどうか通知されません。

注記　TP::deactivateEnable(interface, object id, servant) method を使用すると、オブジェクトを非活性化できます。ただし、オブジェクトがトランザクションに参加している場合、オブジェクトが非活性化されるのは、トランザクションがコミットまたはロールバックしたときです。トランザクションがコミットまたはロールバックする前にオブジェクトに対して呼び出しが行われた場合、オブジェクトは非活性化されません。

必要な動作が確実に実行されるようにするには、オブジェクトがトランザクションに参加していないことを確認するか、TP::deactivateEnable() を呼び出してからトランザクションが終了するまでオブジェクトに対して呼び出しが行われないようにします。

サーバントの存続期間

サーバントとは、IDL インターフェイスのオペレーションを実装するメソッドを格納した C++ クラスのことです。ユーザはサーバントのコードを記述します。TP フレームワークは、サーバント・コードのメソッドを呼び出して要求を満たします。サーバントは、C++ の「new」文で作成され、C++ の「delete」文で破棄されます。ここでは、作成と破棄、および作成と破棄のタイミングについて説明します。

通常のケース

通常、TP フレームワークはサーバントの存続期間を完全に制御します。基本モデルでは、活性化されていないオブジェクトに対する要求を受け取ると、TP フレームワークがサーバントを取得してから、activate_object メ

ソッドを呼び出してサーバントを活性化します。非活性化する場合、TP フレームワークはサーバントの deactivate_object メソッドを呼び出してから、サーバントを破棄します。

「TP フレームワークがサーバントを取得」するフェーズは、TP フレームワークが、サーバントを作成する必要がある場合に、ユーザ作成のサーバ・メソッド、Server::create_servant または ServerBase::create_servant_with_id を呼び出す、ということです。このとき、アプリケーション・コードは要求されたサーバントを指すポインタを返す必要があります。ほとんどの場合、アプリケーションは、サーバントの新しいインスタンスを作成する C++ 「new」文を使用することで、この処理を行います。「サーバントを破棄」するフェーズは、TP フレームワークが、実際に削除するサーバントへのリファレンスを削除する、ということです。

アプリケーションでは、サーバントを作成および削除する動作が将来のバージョンの製品で変更される可能性があることを考慮しておく必要があります。アプリケーションは現在の動作に依存してはなりませんが、サーバントを再利用するサーバント・コードを記述する必要があります。具体的には、サーバント・コードは、サーバントが C++ 「new」文によって作成されていない場合でも、動作しなければなりません。TP フレームワークは、サーバントが非活性化されてからも削除しないで、後で再度活性化することができます。つまり、サーバントは、コンストラクタでサーバントが作成されたときではなく、サーバントの activate_object メソッドでコールバックされたときに自身を初期化しなければならないということです。

特殊なケース

アプリケーションで TP フレームワークの標準的なサーバントの使い方を変更するには、2 つのテクニックがあります。サーバントの取得に関するものとサーバントの破棄に関するものです。

アプリケーションでは、明示的な事前活性化により、「取得」メカニズムを変更できます。この場合、アプリケーションはサーバントを作成して初期化してから、サーバントを活性化された状態として宣言するように TP フレームワークに要求します。こうしたサーバントは、TP::create_active_object_reference 呼び出しによって TP フレームワークに渡されると、その他のサーバントとまったく同じように TP フレームワークによって扱われます。唯一の違いは、作成および初期化の方法です。

アプリケーションは、サーバントの破棄を TP フレームワークに任せる代わりに自身で処理することで、「破棄」メカニズムを変更できます。サーバントが Server::create_servant、ServerBase::create_servant_with_id または TP::create_active_object_reference によって TP フレームワークに通知されると、TP フレームワークのデフォルト動作は、そのサーバント自体を削除することになります。この場合、アプリケーション・コードでは、非活性化後にそのサーバントへのリファレンスを使用してはなりません。

ただし、アプリケーションから TP フレームワークに対して、TP フレームワークがサーバントを非活性化した後にサーバントを破棄しないように指示することはできます。サーバントの処理は、サーバントのクラス全体について行うのではなく、サーバントを識別するパラメータを付けて Tobj_ServantBase::_add_ref を呼び出すことで、個別に行います。

注記　BEA Tuxedo リリース 8.0 以降で作成するアプリケーションでは、TP::application_responsibility() メソッドの代わりに Tobj_ServantBase::_add_ref メソッドを使用します。TP::application_responsibility() メソッドと違い、add_ref() メソッドは引数を取りません。

サーバントの処理を行うアプリケーションの利点は、サーバントを新しく作成せずに済むことです。サーバントの取得に大きなコストが伴う場合、アプリケーションではサーバントを保存しておいて、後でそれを再利用すると便利です。特に便利なのは事前活性化されたオブジェクトのサーバントの場合ですが、一般的にも言えることです。たとえば、TP フレームワークが次に Server::create_servant または ServerBase::create_servant_with_id を呼び出した場合、アプリケーションは以前に保存しておいたサーバントを返すことができます。

また、アプリケーションがサーバントの処理を行うようにすると、アプリケーションは、サーバントが必要でなくなったとき、つまりほかの C++ インスタンスと同じように、リファレンス数がゼロになったときに、Tobj_ServantBase::_remove_ref を使用してサーバントを削除しなければなりません。_remove_ref() メソッドのしくみについては、「Tobj_ServantBase::_remove_ref()」を参照してください。

シングル・スレッドおよびマルチスレッドのサーバ・アプリケーションの作成方法については、『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。

オブジェクトの状態の保存と復元

CORBA オブジェクトが活性化されている間、オブジェクトの状態はサーバント内に格納されます。`TP::create_active_object_reference` を使用するアプリケーションと違い、状態は、オブジェクトが最初に呼び出されたとき、つまりオブジェクト・リファレンスの作成後に CORBA オブジェクトに対してメソッドが最初に呼び出されたときと、オブジェクトが非活性化されて以降の呼び出しで初期化しなければなりません。CORBA オブジェクトが非活性化されている間、オブジェクトの状態をサーバントが活性化されていたプロセスの外部に保存する必要があります。オブジェクトの状態は、共用メモリ、ファイル、データベースなどに保存できます。CORBA オブジェクトを非活性化する前に、オブジェクトの状態を保存して、オブジェクトが活性化されたときに状態を復元する必要があります。

プログラマは、オブジェクトの状態の構成要素と、オブジェクトを非活性化する前に何を保存し、オブジェクトを活性化した後に何を復元するかを決定する必要があります。

CORBA オブジェクトのコンストラクタおよびデストラクタの使い方に関する注記

CORBA オブジェクトの状態をサーバント・クラスのコンストラクタまたはデストラクタで初期化、保存、または復元してはなりません。TP フレームワークが、サーバントのインスタンスを非活性化するときに削除しないで、再利用する可能性があるからです。サーバントのインスタンスの作成および削除のタイミングについては、一切保証されません。

トランザクション

以降の節では、トランザクション方針とトランザクションの使い方について説明します。

トランザクション方針

CORBA オブジェクトがグローバル・トランザクションに参加できるかどうかは、コンパイル時にインプリメンテーションに割り当てられたトランザクション方針によって制御されます。以下の方針を割り当てることができます。

注記　トランザクション方針は、OMG IDL のコンパイル時に設定する ICF ファイルで設定されます。ICF ファイルの詳細については、「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)」を参照してください。

- `never`

  インプリメンテーション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。この方針を設定したインプリメンテーションがトランザクションに関与した場合、例外 (`INVALID_TRANSACTION`) が生成されます。インターフェイスの `UBBCONFIG` ファイルで指定した `AUTOTRAN` 方針は無視されます。

- `ignore`

  インプリメンテーションはトランザクションに関与しません。この方針では、このインプリメンテーションからトランザクション内の要求を送信できます。インターフェイスの `UBBCONFIG` ファイルで指定した `AUTOTRAN` 方針は無視されます。

- `optional` (デフォルトの `transaction_policy`)

  インプリメンテーション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。トランザクションに関与するオブジェクトを含むサーバは、XA 準拠のリソース・マネージャに関連付けられているグループ内で設定する必要があります。`AUTOTRAN` パラメータがインターフェイスの `UBBCONFIG` ファイルで指定されている場合、`AUTOTRAN` は有効になります。

- `always`

  インプリメンテーションはトランザクションに関与します。オブジェクトは、必ずトランザクションに関与する必要があります。トランザク

ション外の要求が送信されると、システムはトランザクションを自動的に開始してからメソッドを呼び出します。トランザクションは、メソッド終了時にコミットされます (これは、optional トランザクション方針を設定したオブジェクトに対して AUTOTRAN を指定した場合とほぼ同じ動作ですが、この動作を有効にするには管理コンフィギュレーションが必要な点が異なります。また、管理コンフィギュレーションによって上書きすることはできません)。トランザクションに関与するオブジェクトを含むサーバは、XA 準拠のリソース・マネージャに関連付けられているグループ内で設定する必要があります。

注記 optional 方針は、管理コンフィギュレーションによって影響を受ける唯一のトランザクション方針です。システム管理者が UBBCONFIG ファイルまたは管理ツールを使用して、AUTOTRAN 属性をインターフェイスに対して設定すると、既にトランザクションに関与している場合を除いて、オブジェクトの呼び出し時にトランザクションが自動的に開始されます。つまり、always 方針を指定した場合と同じ動作になります。

トランザクションの初期化

トランザクションの初期化には、次の 2 つの方法があります。

- CosTransactions::Current::begin() オペレーションを使用した、アプリケーション・コードによる方法。この処理は、クライアントでもサーバでも行うことができます。このオペレーションの説明については、『BEA Tuxedo CORBA トランザクション』を参照してください。
- 次の条件を満たすオブジェクトに対して呼び出しが行われた場合の、システムによる方法
  - トランザクション方針が always に設定されている。
  - トランザクション方針が optional に設定されており、インターフェイスに対して AUTOTRAN が設定されている。

  詳細については、『BEA Tuxedo CORBA トランザクション』を参照してください。

トランザクションの終了

一般に、トランザクションの結果を処理することは、イニシエータの責任です。したがって、次のことが言えます。

- クライアントまたはサーバ・アプリケーション・コードでトランザクションを初期化する場合、TP フレームワークはトランザクションをコミットしません。トランザクションが不当な状態になっているときにサーバ・プロセスがクライアントに復帰しようとすると、BEA Tuxedo システムはトランザクションをロールバックします。
- システムがトランザクションを初期化する場合、必ず BEA Tuxedo システムがコミットまたはロールバックを処理します。

BEA Tuxedo システムによって、以下の動作が強制的に実行されます。

- CORBA オブジェクトのメソッドが呼び出されたときにアクティブなトランザクションがなく、そのメソッドがトランザクションを開始した場合、トランザクションは、メソッド呼び出しが復帰したときにコミット、ロールバック、または一時停止される必要があります。これらのアクションが実行されない場合、トランザクションは TP フレームワークによってロールバックされ、`CORBA::OBJ_ADAPTER` 例外がクライアント・アプリケーションで発生します。この例外はトランザクションがサーバ・アプリケーションで初期化されたために発生します。したがって、クライアント・アプリケーションでは、`TRANSACTION_ROLLEDBACK` のようなトランザクション・エラーを予想していません。

トランザクションの一時停止と再開

CORBA オブジェクトは、メソッド呼び出し内のトランザクションの一時停止および再開に関する規則に厳密に従う必要があります。以下に、これらの規則と規則に違反した場合のエラーについて説明します。

CORBA オブジェクト・メソッドが実行を開始すると、トランザクションに関する状態は次の 3 つのいずれかになります。

- クライアント・アプリケーションがトランザクションを開始しました。

- 正当なサーバ・アプリケーションの動作 : メソッドの実行中にトランザクションを一時停止および再開します。
- 不当なサーバ・アプリケーションの動作 : 一時停止されたトランザクションに関与したメソッドから復帰します。つまり、一時停止された場合、再開しないままメソッドから復帰します。
- エラー処理 : 不当な動作が発生した場合、TP フレームワークは CORBA::TRANSACTION_ROLLEDBACK 例外をクライアント・アプリケーションに生成させ、トランザクションは BEA Tuxedo システムによってロールバックされます。

■ システムはトランザクションを開始して、AUTOTRAN または always トランザクション方針の動作を実行しました。

注記　オペレーション呼び出しを受信したときにトランザクションを自動的に開始する場合、各 CORBA インターフェイスに対して、AUTOTRAN を Yes に設定します。AUTOTRAN を Yes に設定しても、インターフェイスが既にトランザクション・モードにある場合は無効です。AUTOTRAN の詳細については、『BEA Tuxedo CORBA トランザクション』を参照してください。

- 正当なサーバの動作 : メソッドの実行中にトランザクションを一時停止および再開します。

  注記　お勧めしません。メソッドがトランザクションを再開する前に、トランザクションがタイムアウトしたり、アボートされたりする可能性があります。

- 不正なサーバの動作 : 一時停止されたトランザクションに関与したメソッドから復帰します。つまり、一時停止された場合、再開しないままメソッドから復帰します。
- エラー処理 : 不当な動作が発生した場合、TP フレームワークは CORBA::OBJ_ADAPTER 例外をクライアント・アプリケーションに生成させ、トランザクションがロールバックされます。CORBA::OBJ_ADAPTER 例外は、クライアント・アプリケーションがトランザクションを初期化していないために発生します。したがって、クライアント・アプリケーションでは、トランザクション・エラーが発生することを予想していません。

- CORBA オブジェクトは、実行を開始したときにトランザクションに関与していません。
  - 正当なサーバの動作 :
    - メソッドの実行中にトランザクションを開始してコミットします。
    - メソッドの実行中にトランザクションを開始してロールバックします。
    - メソッドの実行中にトランザクションを開始して一時停止します。
  - 不当なサーバの動作 : トランザクションを開始して、トランザクションがアクティブな状態でメソッドから復帰します。
  - エラー処理 : 不当な動作が発生した場合、TP フレームワークは CORBA::OBJ_ADAPTER 例外をクライアント・アプリケーションに生成させ、トランザクションは BEA Tuxedo システムによってロールバックされます。CORBA::OBJ_ADAPTER 例外は、クライアント・アプリケーションがトランザクションを初期化していないために発生します。したがって、クライアント・アプリケーションでは、トランザクション・エラーが発生することを予想していません。

トランザクションに関する制約

BEA Tuxedo CORBA トランザクションには、以下の制約が適用されます。

- BEA Tuxedo システムの CORBA オブジェクトは、メソッド呼び出しから復帰したときに、メソッドが呼び出されたときと同じトランザクション・コンテキストを持つ必要があります。
- CORBA オブジェクトは、一度に 1 つのトランザクションにのみ関与できます。既にトランザクションに関与しているオブジェクトを関与させようとする呼び出しに対しては、CORBA::INVALID_TRANSACTION 例外が返されます。
- CORBA オブジェクトがトランザクションに関与している場合に、そのオブジェクトに対してトランザクションに関与しない要求が送信されると、CORBA::OBJ_ADAPTER 例外が発生します。

- アプリケーションは、Server::initialize(), でトランザクションを開始した場合、メソッドから復帰する前にトランザクションをコミットまたはロールバックしなければなりません。アプリケーションがトランザクションを開始していない場合、TP フレームワークはサーバをシャットダウンします。この理由は、Server::initialize メソッドの終了後に制御をアプリケーションに戻す予測可能な方法がないからです。
- CORBA オブジェクトがトランザクションに関与しており、transaction 活性化方針が設定されている場合、かつメソッドに渡された理由コードが DR_TRANS_COMMITTING または DR_TRANS_ABORTED の場合、Tobj_ServantBase::deactivate_object メソッドからはどの CORBA オブジェクトに対しても呼び出しは行われません。こうした呼び出しに対しては、CORBA::BAD_INV_ORDER 例外が生成されます。

SQL とグローバル・トランザクション

SQL およびグローバル・トランザクションを使用する場合には、次のガイドラインに従ってください。

- グローバル・トランザクションのスコープ外で SQL 文を実行する場合には注意が必要です。SQL 標準では、ローカル・トランザクションは、トランザクション・コンテキストを必要とする SQL 文を実行するときにアクティブなトランザクションがない場合に、データベース・リソース・マネージャによって暗黙的に開始する必要があると指定されています。また、データベース・リソース・マネージャによって暗黙的に開始されたトランザクションは、COMMIT または ROLLBACK SQL 文を実行して、明示的に終了する必要があることも指定されています。これは、TP フレームワークが、リソース・マネージャによって開始されたトランザクションを終了しないからです。

  注記　この点は、アプリケーションが XA ライブラリを使用して Oracle サーバに接続している場合には問題になりません。こうしたアプリケーションはグローバル・トランザクションでしか処理を行えないからです。Oracle サーバでは、XA を使用している場合にローカル・トランザクションが許可されません。

- SQL COMMIT および ROLLBACK 文を使用して、Current.begin() を明示的に使用して開始されたグローバル・トランザクションやシステム

によって暗黙的に開始されたグローバル・トランザクションを終了することはできません。各データベース製品でグローバル・トランザクションを使用する場合のその他の制約については、データベース・ベンダのマニュアルで確認してください。

- SQL カーソルは、トランザクションの終了時にクローズすることができます。カーソルの処理規則については、お使いのデータベース製品のマニュアルを参照してください。アプリケーション・プログラマは、適切な活性化方針を設定した CORBA オブジェクトでのみ、かつ適切なトランザクション境界内で、カーソルを使用するよう注意する必要があります。

トランザクションの結果に関する判断

CORBA オブジェクトは、トランザクション処理の 2 つの段階で、トランザクションの結果に関与することができます。

- トランザクションの処理中

  `Current.rollback_only` メソッドを使用すると、現在のトランザクションをロールバックすることが唯一の結果になることが保証されます。`Current.rollback_only()` は、どの CORBA オブジェクト・メソッドからも呼び出せます。

- トランザクションの処理終了後

  トランザクション・バウンドの活性化方針が設定された CORBA オブジェクトでは、トランザクションの処理終了後にトランザクションをコミットするかロールバックするかを判断できます。これらのオブジェクトには、TP フレームワークが 2 フェーズ・コミット・アルゴリズムを開始する前で、`deactivate_object` メソッドを呼び出すときに、トランザクションの処理が終了したことが通知されます。

  ただしこの動作は、`process` または `method` 活性化方針が設定されたオブジェクトには適用されません。CORBA オブジェクトがトランザクションをロールバックする場合、CORBA オブジェクトは `Current::rollback_only` を呼び出します。トランザクションをコミットすることを支持する場合、CORBA オブジェクトはそのメソッドを呼び出しません。ただし、コミットすることを支持してもトランザクションが

実際にコミットされるとは限りません。その後、ほかのオブジェクトがトランザクションをロールバックすることを支持する可能性があるからです。

注記　SQL カーソルのユーザは、`method` または `process` 活性化方針が設定されたオブジェクトを使用する場合に注意する必要があります。プロセスは、クライアントが開始したトランザクション内で SQL カーソルをオープンします。典型的な SQL データベース製品の場合、クライアントがトランザクションをコミットすると、そのトランザクション内でオープンされていたすべてのカーソルは自動的にクローズされますが、オブジェクトにはカーソルがクローズされたことが通知されません。

トランザクションのタイムアウト

トランザクションのタイムアウトが発生すると、トランザクションをロールすることが唯一の結果になるようにトランザクションにマークされ、`CORBA::TRANSACTION_ROLLEDBACK` 標準例外がクライアントに返されます。新しい要求を送信しようとすると、トランザクションがアボートされるまで、必ず `CORBA::TRANSACTION_ROLLEDBACK` 例外となって失敗します。

パラレル・オブジェクト

リリース 8.0 の BEA Tuxedo CORBA には、性能の拡張を目的としてパラレル・オブジェクトのサポートが追加されています。パラレル・オブジェクト機能を利用すると、特定アプリケーションのすべてのビジネス・オブジェクトを状態を持たないオブジェクトとして指定できます。1 つのドメインの 1 つのサーバでしか実行できない、状態を持つビジネス・オブジェクトと異なり、状態を持たないビジネス・オブジェクトは 1 つのドメインのすべてのサーバで実行できます。パラレル・オブジェクトの利点は以下のとおりです。

- パラレル・オブジェクトは、同じドメインの複数のサーバで同時に実行できます。したがって、すべてのサーバを利用して複数の同時要求を処理できるので性能が向上します。
- BEA Tuxedo システムでパラレル・ビジネス・オブジェクトへの要求が処理されるときには、常に、ローカル・マシン上の利用可能なサーバが最初にチェックされます。ローカル・マシン上のすべてのサーバが要求されたビジネス・オブジェクトの処理で使用されている場合、BEA Tuxedo システムではローカル・ドメイン内のほかのマシンで利用可能なサーバを探します。したがって、ローカル・マシンに複数のサーバがある場合は、ネットワーク・トラフィックが削減され、性能が向上します。

パラレル・オブジェクトの詳細については、『BEA Tuxedo CORBA アプリケーションのスケーリング、分散、およびチューニング』を参照してください。

パラレル・オブジェクトを実装するために、同時実行方針オプションが ICF ファイルに追加されています。特定のアプリケーションに対してパラレル・オブジェクトを選択するには、同時実行方針オプションをユーザ制御に設定します。ユーザ制御の同時実行を選択すると、ビジネス・オブジェクトはアクティブ・オブジェクト・マップ (AOM) に登録されないので状態を持たなくなり、同時に複数のサーバ上で活性化されることができます。このため、こうしたオブジェクトは「パラレル・オブジェクト」と呼ばれます。

ユーザ制御の同時実行を選択した場合、サーバントのインプリメンテーションは以下のいずれかの記述に該当しなければなりません。

- サーバントのインプリメンテーションには、共用リソースへの同時アクセスに関する要件がありません。
- サーバントのインプリメンテーションは、リソースへの同時アクセス中に正しく動作するために、データベースやロッキング機能などのツールを利用する必要があります。

リリース 8.0 の BEA Tuxedo ソフトウェアでは、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) が変更されてユーザ制御の同時実行をサポートするようになりました。リスト 3-1 では、このサポートのための変更箇所が太字で強調されています。ICF の構文の説明については、「ICF の構文」を参照してください。

コードリスト 3-1 ICF の構文

```
[#pragma activation_policy method|transaction|process]
[#pragma transaction_policy never|ignore|optional|always]
[#pragma concurrency_policy user_controlled|system_controlled]
[Module module-name {]
   implementation [implementation-name]
     {
     implements (module-name::interface-name);
     [activation_policy (method|transaction|process);]
     [transaction_policy (never|ignore|optional|always);]
     [concurrency_policy (user_controlled|system_controlled);]
     };
  [};]
```

ユーザ制御の同時実行は、ファクトリ・ベース・ルーティング、すべての活性化方針、およびすべてのトランザクション方針で使用できます。これらの機能との対話は次のとおりです。

- ファクトリ・ベース・ルーティング

  オブジェクトの作成時にユーザがファクトリ・ベース・ルーティングを指定すると、オブジェクトはそのグループ内のサーバにルーティングされます。オブジェクト・キーにはファクトリ・ベース・ルーティングで選択されたグループが含まれていますが、クライアント・ルーティング・コードでは、インターフェイスにユーザ制御の同時実行が設定されていることを認識し、必要なグループを指定します。これは、BEA Tuxedo の通常のルーティングを使用して行われます。

- 活性化方針

  TP フレームワークでは、ユーザ制御の同時実行を設定された活性化されたオブジェクトは、システム制御の同時実行を設定されたオブジェクトと同じように扱われます。TP フレームワークでは、オブジェクトに関する情報はローカルの AOM に格納され、必要に応じて `activate_object` および `deactivate_object` メソッドが呼び出されます。ただし、オブジェクトは AOM 内にエントリを持たないので、TP フレームワークは AOM のルーチンを呼び出しません。たとえば、活性化されたオブジェクトは AOM ハンドルを持たないので、シャットダウン時に AOM からエントリを削除する呼び出しは行われません。

- トランザクション方針

  TP フレームワークでは、ユーザ制御の同時実行を設定された活性化されたオブジェクトは、システム制御の同時実行を設定されたオブジェクトと同じように扱われます。TP フレームワークがトランザクションのイベントに対してコールバックされると、トランザクションに関与するユーザ制御のオブジェクトに関する情報が AOM に格納されます。状態を持つオブジェクトと比べてパラレル・オブジェクトをトランザクションで使用する場合の主な違いは、AOM が GTRID 情報を格納するために使用されず、AOM のルーチンがトランザクション情報を更新または取得するために呼び出されないことです。

注記　ユーザ制御の同時実行に関して次の制約があります。
`TP::create_active_object_reference` は、ユーザ制御の同時実行が設定されたインターフェイスを渡されると、`TobjS::IllegalOperation` 例外をスローします。AOM はユーザ制御の同時実行が設定されている場合に使用されないので、TP フレームワークには活性化されたオブジェクトをこのサーバに接続する方法がありません。

TP フレームワーク API

ここでは、TP フレームワーク API について説明します。この API については、『BEA Tuxedo CORBA サーバ・アプリケーションの開発方法』でも説明しています。

TP フレームワークは以下のコンポーネントで構成されています。

- `Server` C++ クラス。アプリケーション固有のサーバ初期化および終了ロジック用の仮想メソッドを持っています。
- `ServerBase` C++ クラス。マルチスレッド・サーバ・アプリケーション用の仮想メソッドを持っています。
- `Tobj_ServantBase` C++ クラス。オブジェクトの状態管理用の仮想メソッドを持っています。

- TP C++ クラス。以下の処理を実行するためのメソッドを用意しています。
  - CORBA オブジェクトのオブジェクト・リファレンスの作成
  - FactoryFinder オブジェクトへのファクトリの登録および登録解除
  - ユーザ制御によるオブジェクトの事前活性化および非活性化処理の開始
  - ユーザ制御による呼び出し中の CORBA オブジェクトの非活性化処理の開始
  - 呼び出し中の CORBA オブジェクトへのオブジェクト・リファレンスの取得
  - XA リソース・マネージャのオープンおよびクローズ
  - ユーザ・ログ (ULOG) ファイルへのメッセージの記録
  - ORB および Bootstrap オブジェクトへのオブジェクト・リファレンスの取得 (CORBA インターオペラブル・ネーミング・サービス (INS) を使用していない場合)
- 上記クラスのヘッダ・ファイル
- サーバ・アプリケーションが使用するライブラリ

TP フレームワークの可視部分は、2 種類のオペレーションで構成されています。

- ユーザ・コードで呼び出し可能なサービス・メソッド。これらのメソッドは TP インターフェイスに含まれています。
- ユーザが記述し、TP フレームワークで呼び出すコールバック・メソッド。`Tobj_ServantBase` および `Server` クラスのメソッドも含まれます。これらのオペレーションは、TP フレームワークのコードでのみ呼び出すためのものです。アプリケーション・コードでは、これらのクラスのメソッドを呼び出してはなりません。呼び出した場合、予期できない結果となる可能性があります。

Server インターフェイス

Server インターフェイスには、アプリケーション固有のサーバ初期化および終了ロジック用のコールバック・メソッドが用意されています。このインターフェイスには、オブジェクトを活性化するためにサーバントが必要な場合に、サーバントを作成するためのコールバック・メソッドも用意されています。

Server インターフェイスには次の特徴があります。

- Server クラスは、ServerBase クラスを継承します。
- `Server` クラスは C++ ネイティブ・クラスです。
- `Server.h` ファイルには、Server クラスの宣言および定義が格納されています。

Server インターフェイスのメソッドの説明については、「ServerBase インターフェイス」を参照してください。

C++ 宣言

C++ マッピングについては、「ServerBase インターフェイス」を参照してください。

ServerBase インターフェイス

serverBase インターフェイスを使用すると、マルチスレッド処理機能の利点を最大限に活用できます。ServerBase クラスを継承する独自の Server クラスを作成することもできます。このクラスでは以下のものが提供されます。

- サーバントの作成時に対象のオブジェクトの情報を必要とするインプリメンテーションをサポートする create_servant_with_id() メソッド
- ユーザ指定のスレッド初期化ハンドラおよびリリース・ハンドラのサポート

ServerBase クラスでは、以前のリリースの Server クラスで利用可能だったオペレーションを使用できます。Server クラスは、ServerBase クラスを継承します。

これらのオペレーションは、シングル・スレッド・アプリケーションでもマルチスレッド・アプリケーションでも使用できます。

- Server::create_servant()
- Server::initialize()
- Server::release()
- ServerBase::create_servant_with_id()

これらのメソッドは、マルチスレッド・サーバ・アプリケーションでのみ使用できます。

- ServerBase:: thread_initialize()
- ServerBase::thread_release()

注記　プログラマは、Server クラスのメソッドを定義する必要があります。ServerBase クラスのメソッドには、デフォルトのインプリメンテーションがあります。

C++ 宣言 (Server.h 内)

C++ のマッピングは次のとおりです。

```
class OBBEXPDLLUSER ServerBase {
public:
```

```
    virtual CORBA::Boolean
        initialize(int argc, char** argv) = 0;

    virtual void
        release() = 0;

    virtual Tobj_Servant
        create_servant(const char* interfaceName) = 0;

       // デフォルト・インプリメンテーションの指定
       virtual Tobj_Servant
           create_servant_with_id(const char* interfaceName,
                                  const char* stroid);

       virtual CORBA::Boolean
           thread_initialize(int argc, char** argv);

       virtual void
           thread_release();

};

class Server : public ServerBase {
public:

    CORBA::Boolean initialize(int argc, char** argv);
    void release();
    Tobj_Servant create_servant(const char* interfaceName);
};
```

Server::create_servant()

概要 サーバントを作成して C++ オブジェクトをインスタンス化します。

C++ バインディング

```
class Server {
public:
      Tobj_Servant     create_servant(const char* interfaceName);
};
```

引数

`interfaceName`

オブジェクトの完全修飾インターフェイス名を含む文字列を指定します。この名前は、`TP::create_object_reference()` を使用してオブジェクト・リファレンスを作成した場合に指定したインターフェイス名、または `TP::create_active_object_reference()` の呼び出しで使用したオブジェクト・リファレンスのインターフェイス名と同じになります。この名前を使用して、作成する必要があるサーバントを決定できます。

例外 `Server::create_servant()` で例外がスローされた場合、TP フレームワークが例外をキャッチします。活性化は失敗します。
`CORBA::OBJECT_NOT_EXIST()` 例外がクライアントに返されます。また、エラー・メッセージが、次のように例外型ごとにユーザ・ログ (`ULOG`) ファイルに書き込まれます。

`TobjS::CreateServantFailed`

"TPFW_CAT:23: ERROR: Activating object - application raised TobjS::CreateServantFailed. Reason = reason. Interface = interfaceName, OID = *oid*"

reason はユーザ指定の理由を示し、*interfaceName* と *oid* はそれぞれ呼び出された CORBA オブジェクトの インターフェイス ID とオブジェクト ID を示します。

`TobjS::OutOfMemory`

"TPFW_CAT:22: ERROR: Activating object - application raised TobjS::OutOfMemory. Reason = reason. Interface = *interfaceName*, OID = *oid*"

reason はユーザ指定の理由を示し、*interfaceName* と *oid* はそれぞれ呼び出された CORBA オブジェクトの インターフェイス ID とオブジェクト ID を示します。

CORBA::Exception
"TPFW_CAT:28: ERROR: Activating object - CORBA Exception not handled by application. Exception ID = exceptionID. Interface = *interfaceName*, OID = *oid*"

exceptionID は例外のインターフェイス ID を示し、*interfaceName* と *oid* はそれぞれ呼び出された CORBA オブジェクトのインターフェイス ID とオブジェクト ID を示します。

その他の例外
"TPFW_CAT:29: ERROR: Activating object - Unknown Exception not handled by application. Exception ID = exceptionID. Interface = *interfaceName*, OID = *oid*"

exceptionID は例外のインターフェイス ID を示し、*interfaceName* と *oid* はそれぞれ呼び出された CORBA オブジェクトのインターフェイス ID とオブジェクト ID を示します。

説明 create_servant メソッドは、要求がサーバに送信され、その要求を満たすための利用可能なサーバントがない場合に、TP フレームワークによって呼び出されます。TP フレームワークは、作成するサーバントのインターフェイス名を指定して create_servant メソッドを呼び出します。サーバ・アプリケーションでは、適切な C++ オブジェクトをインスタンス化して、そのオブジェクトを指すポインタを返します。通常、このメソッドは、インターフェイス名の switch 文を格納しており、インターフェイス名に従って新しいオブジェクトを作成します。

Caution: サーバ・アプリケーションでは、CORBA オブジェクトが活性化されるたびにこのメソッドが呼び出されることを前提にしてはなりません。また、サーバ・アプリケーションでは、CORBA オブジェクトのサーバント・クラスのコンストラクタまたはデストラクタで、CORBA オブジェクトの状態を処理してはなりません。TP フレームワークが活性化時にサーバントを再利用する場合や、非活性化時にサーバントを破棄しない場合が考えられるからです。

戻り値 Tobj_Servant
指定されたインターフェイスに対して新しく作成されたサーバント (インスタンス) に対するポインタ。認識できないインターフェイスを指定して create_servant() を呼び出した場合、または何らかの理由でサーバントを作成できなかった場合は、NULL 値が返されます。

create_servant メソッドが NULL ポインタを返した場合、活性化は失敗します。CORBA::OBJECT_NOT_EXIST() 例外がクライアントに返されます。また、次のメッセージがユーザ・ログ (ULOG) に書き込まれます。

```
"TPFW_CAT:23: ERROR: Activating object - application raised
TobjS::CreateServantFailed. Reason = Application's
Server::create_servant returned NULL. Interface =
interfaceName, OID = oid"
```

interfaceName は呼び出されたインターフェイスのインターフェイス ID を示し、*oid* は対応するオブジェクト ID を示します。

注記　このリリースでは、ObjectId の長さに関する制約がなくなりました。

ServerBase::create_servant_with_id()

概要 対象オブジェクト用のサーバントを作成します。このメソッドは、シングル・スレッドおよびマルチスレッド・サーバ・アプリケーションの開発をサポートします。

C++ バインディング

```
Tobj_Servant create_servant_with_id (const char* interfaceName,
                                     const char* stroid);
```

引数

interfaceName

オブジェクトの完全修飾インターフェイス名を含む文字列を指定します。このインターフェイス名は、オブジェクト・リファレンスを作成したときに指定した名前と同じでなければなりません。

stroid

オブジェクト ID を文字列形式で指定します。オブジェクト ID は、処理対象の要求に関連付けられるオブジェクトを一意に識別します。このオブジェクト ID は、オブジェクト・リファレンスを作成したときに指定した ID と同じでなければなりません。

説明 TP フレームワークは、要求がサーバに送信されたときに、その要求を満たすための利用可能なサーバントがない場合に create_servant_with_id メソッドを呼び出します。TP フレームワークは、作成するサーバントのインターフェイスとサーバントに関連付けられるオブジェクトのオブジェクト ID を渡します。サーバ・アプリケーションでは、適切な C++ オブジェクトをインスタンス化して、そのオブジェクトを指すポインタを返します。通常、このメソッドは、インターフェイス名の switch 文を格納しており、インターフェイス名に従って新しいオブジェクトを作成します。オブジェクト ID を指定すると、サーバントのインプリメンテーションでは、サーバント・インスタンスの作成時に対象オブジェクトの情報を基にさまざまな決定を行えます。リエントラントのサポートは、サーバントのインプリメンテーションで対象オブジェクトの情報を利用する方法の一例です。

ServerBase クラスには、インターフェイス名を渡す標準の create_servant メソッドを呼び出す create_servant_with_id のデフォルト・インプリメンテーションが用意されています。このデフォルト・インプリメンテーションでは、対象のオブジェクト ID パラメータは無視されます。

Caution: サーバ・アプリケーションでは、CORBA オブジェクトが活性化されるたびにこのメソッドが呼び出されることを前提にしてはな

りません。また、サーバ・アプリケーションでは、CORBA オブジェクトのサーバント・クラスのコンストラクタまたはデストラクタで、CORBA オブジェクトの状態を処理してはなりません。TP フレームワークが活性化時にサーバントを再利用する場合や、非活性化時にサーバントを破棄しない場合が考えられるからです。

戻り値 `Tobj_Servant`

指定されたインターフェイスに対して新しく作成されたサーバント (インスタンス) に対するポインタ。以下のいずれかの条件が満たされる場合は、`NULL` を返します。

- インターフェイス名を認識できない。
- サーバントを作成できない。

例

```
Tobj_Servant simple_per_request_server::create_servant_with_id(
        const char* intf_repos_id, const char* stroid)
{
        TP::userlog("create_servant_with_id called in thread %ld",
                    (unsigned long)SIMPTHR_GETCURRENTTHREADID);

        // このオブジェクト ID に基づいて必要な初期化を
        // 実行する

        return create_servant(intf_repos_id);
}
```

Server::initialize()

概要 アプリケーションが、データベースへのログイン、既知のオブジェクト・ファクトリの作成および登録、グローバル変数の初期化などのアプリケーション固有の初期化手続きを実行できるようにします。

C++ バインディング

```
class Server {
public:

        CORBA::Boolean  initialize(int argc, char** argv);

};
```

引数 argc および argv 引数がコマンド行から渡されます。argc 引数には、サーバ名が格納されます。argv 引数には、アプリケーション固有の最初のコマンド行オプションが格納されます (存在する場合)。

コマンド行オプションは、SERVERS セクションにあるサーバのエントリの CLOPT パラメータを使用して、UBBCONFIG ファイルで指定します。CLOPT パラメータには、システムで認識されるオプション、ダブル・ハイフン (--)、アプリケーション固有のオプションの順に指定します。argc の値は、アプリケーション固有のオプションの数よりも 1 大きい値です。詳細については、『BEA Tuxedo のファイル形式とデータ記述方法』の「ubbconfig(5)」を参照してください。

例外 Server::initialize() で例外が発生すると、TP フレームワークがその例外をキャッチします。TP フレームワークは、initialize() が FALSE を返した場合と同じように動作します。つまり、例外は失敗と見なされます。また、エラー・メッセージが、次のように例外型ごとにユーザ・ログ (ULOG) ファイルに書き込まれます。

```
TobjS::InitializeFailed
       "TPFW_CAT:1: ERROR: Exception in
       Server::initialize():IDL:beasys.com/TobjS/InitializeFailed:
       1.0. Reason = reason"
```

reason は、アプリケーション・コードで指定される文字列です。たとえば、次のように入力します。

```
       Throw TobjS::InitializeFailed(
                        "Couldn't register factory");
```

CORBA::Exception
"TPFW_CAT:1: ERROR: Exception in Server::initialize():
exception. Reason = unknown"

exception は、発生した CORBA 例外のインターフェイス ID です。

その他の例外
"TPFW_CAT:1: ERROR: Exception in Server::initialize():
unknown exception. Reason = unknown"

説明 サーバ初期化の最後のステップとして呼び出される initialize コールバック・メソッドを使用すると、アプリケーションがアプリケーション固有の初期化を実行できます。

通常、サーバ・アプリケーションは、Server::initialize で以下のタスクを実行します。

- サーバ・アプリケーションに実装された CORBA オブジェクト・ファクトリのリファレンスを作成し、TP::register_factory() オペレーション・チェックを使用して FactoryFinder に登録します。
- グローバル変数を初期化します (使用する場合)。
- XA リソース・マネージャをオープンします (サーバ・アプリケーションで使用する場合)。

サーバ・アプリケーションでは、必要な XA リソース・マネージャをオープンする必要があります。この処理には、以下のいずれかのメソッドを呼び出します。

- TP::open_xa_rm()
  処理は静的関数で行い、オブジェクト・リファレンスを取得する必要がないので、これはサーバ・アプリケーションでは有用なテクニックです。

  注記 INS ブートストラップ処理メカニズムを使用して初期オブジェクト・リファレンスを取得する場合は、TP::open_xa_rm() メソッドを使用する必要があります。

- Tobj::TransactionCurrent::open_xa_rm()
  TransactionCurrent オブジェクトのリファレンスは、Bootstrap オブジェクトから取得できます。Bootstrap オブジェクトのリファレンスの取得方法については、「TP::bootstrap()」を参照してください。

TransactionCurrent オブジェクトの詳細については、「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」と『BEA Tuxedo CORBA トランザクション』を参照してください。

- トランザクションは、Tobj::TransactionCurrent::open_xa_rm() または TP::open_xa_rm メソッドの呼び出し後の initialize メソッドで開始できます。ただし、initialize() メソッドで開始したトランザクションは、initialize() が復帰する前に、サーバ・アプリケーションで終了する必要があります。制御が戻ったときにトランザクションがアクティブな場合、サーバ・アプリケーションが起動に失敗し、正常に終了します。これは、Server::initialize() が復帰した後にトランザクションをコミットするのかロールバックするのかを論理的に処理する方法がサーバ・アプリケーションにないからです。この状況はエラーとなります。

戻り値 Boolean の TRUE または FALSE。TRUE は成功を示します。FALSE は失敗を示します。initialize() でエラーが発生した場合、アプリケーション・コードは FALSE を返します。アプリケーション・コードでは、システム・コールの exit() を呼び出してはなりません。exit() を呼び出すと、TP フレームワークが起動時に割り当てられたリソースを解放できないので、予期できない結果が発生する可能性があります。

戻り値が FALSE の場合は、次のように処理されます。

- Server::release() は呼び出されません。
- initialize() メソッドで開始され、終了されていないトランザクションは、最終的にタイムアウトします。自動的にロールバックされるわけではありません。

ServerBase::thread_initialize()

概要 BEA Tuxedo ソフトウェアを使用して作成されたスレッドに対して、必要なアプリケーション固有の初期化を実行します。このメソッドは、マルチスレッド・サーバ・アプリケーションの開発をサポートします。

C++ バインディング

```
CORBA::Boolean thread_initialize(int argc, char** argv)
```

引数

`argc`
: アプリケーションに指定する引数の数。最初に、このカウントは `main` 関数に渡されます。

`argv`
: アプリケーションに指定する引数。最初に、これらの引数は `main` 関数に渡されます。

説明 スレッド・プールを管理する場合、BEA Tuxedo ソフトウェアでは、オペレーティング・システムのスレッド・ライブラリ・サービスを使用してスレッドを作成および解放します。アプリケーションの要件によっては、要求を処理する前にこれらのスレッドを初期化する必要があります。

`thread_initialize` コールバック・メソッドは、スレッドが作成されるたびに、スレッドの初期化を目的として呼び出されます。ただし、BEA Tuxedo ソフトウェアでは、要求をディスパッチするために多数のシステム所有スレッドを管理しています。これらのスレッドも、スレッド・プールに追加されます。状況によっては、ユーザが実装したサーバントのメソッドもこれらのシステム所有スレッドで実行されます。このため、BEA Tuxedo ソフトウェアでは `thread_initialize` メソッドを呼び出して、システム所有のスレッドが初期化されます。

`ServerBase` クラスには、初期化されたスレッドで XA リソース・マネージャをオープンする `thread_initialize` メソッドのデフォルト・インプリメンテーションが用意されています。

戻り値

`CORBA::Boolean`
: スレッドの初期化が成功した場合は `True` です。

例

```
CORBA::Boolean simple_per_request_server::thread_initialize(
                        int argc, char** argv)
{
        TP::userlog("thread_initialize called in thread %ld",
```

```
                        (unsigned long)SIMPTHR_GETCURRENTTHREADID);
        return CORBA_TRUE;
}
```

Server::release()

概要 アプリケーションが、データベースからのログオフ、既知のファクトリの登録の削除、リソースの割り当て解除などのアプリケーション固有のクリーンアップを実行できるようにします。

C++ バインディング

```
typedef Tobj_ServantBase* Tobj_Servant;

class Server {
public:
        void      release();
};
```

引数 特にありません。

例外 release() で例外が発生すると、TP フレームワークがその例外をキャッチします。例外が発生するたびに、エラー・メッセージが、次のようにユーザ・ログ (ULOG) ファイルに書き込まれます。

```
TobjS::ReleaseFailed
       "TPFW_CAT:2: WARN: Exception in Server::release():
       IDL:beasys.com/TobjS/ReleaseFailed:1.0. Reason = reason"
```

reason は、アプリケーション・コードで指定される文字列です。たとえば、次のように入力します。

```
       Throw TobjS::ReleaseFailed(
                  "Couldn't unregister factory");
```

```
CORBA::Exception
       "TPFW_CAT:2: WARN: Exception in Server::release():
       exception. Reason = unknown"
```

exception は、発生した CORBA 例外のインターフェイス ID です。

その他の例外

```
       "TPFW_CAT:2: WARN: Exception in Server::release(): unknown
       exception. Reason = unknown"
```

いずれの場合でも、例外の発生に続いてサーバが終了します。

説明 サーバ初期化の最初のステップとして呼び出される release コールバック・メソッドを使用すると、アプリケーションがアプリケーション固有のクリーンアップを実行できます。ユーザは、仮想関数の定義を上書きしなければなりません。

通常、このメソッドでは以下の処理が実行されます。

- XA リソース・マネージャのクローズ
- Server::initialize() で FactoryFinder に登録された CORBA オブジェクト・ファクトリの登録の削除
- 解放されていないサーバ・リソースの割り当て解除

このメソッドは通常、管理者またはオペレータからの tmshutdown コマンドに応答して呼び出されます。

TP フレームワークには、Server::release() のデフォルト・インプリメンテーションが用意されています。デフォルト・インプリメンテーションは、サーバ用の XA リソース・マネージャをクローズします。この処理は、UBBCONFIG ファイルでサーバのグループに対してデフォルト設定されている CLOSEINFO を使用する tx_close() 呼び出しによって行われます。

アプリケーションでは、オープンされている XA リソース・マネージャをクローズする必要があります。この処理には、以下のいずれかのメソッドを呼び出します。

- TP::close_xa_rm()

注記　INS ブートストラップ処理メカニズムを使用して初期オブジェクト・リファレンスを取得する場合は、TP::close_xa_rm() メソッドを使用する必要があります。

- Tobj::TransactionCurrent::close_xa_rm()。TransactionCurrent オブジェクトのリファレンスは、Bootstrap オブジェクトから取得できます。Bootstrap オブジェクトのリファレンスの取得方法については、「TP::bootstrap()」を参照してください。TransactionCurrent オブジェクトの詳細については、「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」と『BEA Tuxedo CORBA トランザクション』を参照してください。

注記　サーバが tmshutdown(1) コマンドからシャットダウン要求を受信すると、ほかのリモート・オブジェクトからの要求を受信できなくなります。サーバをシャットダウンする場合、順序を考慮しなければならないことがあります。たとえば、サーバ 1 の Server::release() メソッドから サーバ 2 にあるオブジェクトのメソッドにアクセスする必要がある場合、サーバ 1 をシャットダウンしてから、サーバ 2 をシャットダウンしなければなりません。特に、

TP::unregister_factory() メソッドは、別のサーバにある FactoryFinder Registrar オブジェクトにアクセスします。通常、TP::unregister_factory() メソッドは release() メソッドから呼び出されるので、FactoryFinder サーバは、Server::release() メソッドで TP::unregister_factory() を呼び出すすべてのサーバの後にシャットダウンする必要があります。

戻り値　特にありません。

ServerBase::thread_release()

概要 BEA Tuxedo ソフトウェアで作成されたスレッドが解放されたときに、アプリケーション固有のクリーンアップを実行します。このメソッドは、マルチスレッド・サーバ・アプリケーションの開発をサポートします。

C++ バインディング

```
void thread_release()
```

引数 特にありません。

説明 thread_release コールバック・メソッドは、スレッドが解放されるたびに呼び出されます。アプリケーション固有のリソース・クリーンアップを実行する場合は、thread_release を実装します。

ServerBase クラスには、解放されたスレッドの XA リソース・マネージャをクローズする thread_release メソッドのデフォルト・インプリメンテーションが用意されています。

戻り値 特にありません。

例

```
void simple_per_request_server::thread_release()
{
        TP::userlog("thread_release called in thread %ld",
                    (unsigned long)SIMPTHR_GETCURRENTTHREADID);
}
```

Tobj_ServantBase インターフェイス

Tobj_ServantBase インターフェイスは、PortableServer::RefCountServantBase クラスを継承し、スレッド・セーフな方法で状態を管理する際に CORBA オブジェクトが役立つようにするオペレーションを定義します。IDL コンパイラによって生成されるインプリメンテーションの各スケルトンは、Tobj_ServantBase クラスを自動的に継承します。Tobj_ServantBase クラスには、プログラマが任意で実装可能な 2 つの仮想メソッド (activate_object() および deactivate_object()) が含まれています。

活性化されたいない CORBA オブジェクトに対する要求が受信されるたびに、オブジェクトが活性化され、activate_object() メソッドがサーバントに対して呼び出されます。CORBA オブジェクトが非活性化されると、deactivate_object() メソッドがサーバントに対して呼び出されます。非活性化のタイミングは、インプリメンテーションの活性化方針によって決まります。deactivate_object() メソッドが呼び出されると、TP フレームワークは呼び出しの理由を示す理由コードを渡します。

これらのメソッドは、マルチスレッド・サーバ・アプリケーションの開発をサポートします。

- TobjServantBase::_add_ref()
- TobjServantBase::_is_reentrant()
- TobjServantBase::_remove_ref()

注記　CORBA オブジェクトの活性化および非活性化時に呼び出すことを TP フレームワークで保証されているメソッドは、Tobj_ServantBase::activate_object() と Tobj_ServantBase::deactivate_object() だけです。活性化および非活性化時に C++ の Server::create_servant 呼び出しによってサーバント・クラスのコンストラクタおよびデストラクタを呼び出すことはできません。したがって、サーバ・アプリケーションでは、サーバント・クラスのコンストラクタまたはデストラクタで CORBA オブジェクトの状態を処理してはなりません。

注記　プログラマは、直接 Tobj_ServantBase をキャストしたり、参照したりする必要がありません。Tobj_ServantBase メソッドはスケルトンに含まれているので、サーバントのインプリメンテーションにも含まれることになります。プログラマは activate_object および deactivate_object メソッドを定義できますが、これらのメソッドを直接呼び出してはなりません。TP フレームワークだけがこれらのメソッドを呼び出します。

C++ 宣言 (Tobj_ServantBase.h 内)

次に、Tobj_servantBase インターフェイスの C++ マッピングを示します。

```
class Tobj_ServantBase : public PortableServer::RefCountServantBase {
public:
```

```
    Tobj_ServantBase& operator=(const Tobj_ServantBase&);
    Tobj_ServantBase() {}
    Tobj_ServantBase(const Tobj_ServantBase& s) :
        PortableServer::RefCountServantBase(s) {}

    virtual void activate_object(const char *) {}

    virtual void deactivate_object(const char*,
        TobjS::DeactivateReasonValue) {}

    virtual CORBA::Boolean _is_reentrant() { return CORBA_FALSE; }
};

typedef Tobj_ServantBase * Tobj_Servant;
```

Tobj_ServantBase:: activate_object()

概要 オブジェクト ID をサーバントに関連付けます。このメソッドによって、アプリケーションでは、オブジェクトが活性化されたときにオブジェクトの状態を復元できます。状態は、共用メモリ、通常のフラット・ファイル、またはデータベース・ファイルから復元できます。

C++ バインディング

```
class Tobj_ServantBase : public PortableServer::ServantBase {
public:
        virtual void activate_object(const char * stroid) {}
};
```

引数

`stroid`

オブジェクト ID を文字列形式で指定します。オブジェクト ID は、クラスのこのインスタンスを一意に識別します。このオブジェクト ID は、`TP::create_object_reference()` を使用してオブジェクト・リファレンスを作成したときに指定した ID、または `TP::create_active_object_reference()` の呼び出しで使用したオブジェクト・リファレンスの ID と同じになります。

注記　このリリースでは、オブジェクト ID の長さに関する制約がなくなりました。

説明 オブジェクトの活性化は、クライアントが活性化されていない CORBA オブジェクトのメソッドを呼び出すことで開始されます。これにより、ポータブル・オブジェクト・アダプタ (POA) がサーバントを CORBA オブジェクトに割り当てます。`activate_object()` メソッドは、クライアントが呼び出したメソッドの前に呼び出されます。`activate_object()` から正常に制御が戻った場合、つまり例外が発生しなかった場合、要求されたメソッドがサーバントで実行されます。

プログラマは、`activate_object()` および `deactivate_object()` メソッドとクライアントが呼び出すメソッドを使用して、オブジェクトの状態を管理できます。これらのメソッドを使用してオブジェクトの状態を管理する方法は、アプリケーションのニーズによって異なります。これらのメソッドの使い方については、『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。

オブジェクトがグローバル・トランザクションに関与している場合、`activate_object()` はそのグローバル・トランザクションのスコープ内で実行されます。

オブジェクトのプログラマは、格納されているオブジェクトの状態が矛盾していないかどうかをチェックする必要があります。つまり、アプリケーション・コードでは、deactivate_object() がオブジェクトの状態を正しく保存したかどうかを示す永続性フラグを保存する必要があります。このフラグを activate_object() でチェックします。

戻り値 特にありません。

例外 activate_object() の実行中にエラーが発生した場合、アプリケーション・コードでは、CORBA 標準例外または TobjS::ActivateObjectFailed 例外を生成する必要があります。例外が発生すると、TP フレームワークは例外をキャッチして、以下のイベントが発生します。

- 活性化が失敗します。
- クライアントが呼び出したメソッドは実行されません。
- activate_object() がクライアントの開始したトランザクション内で実行されている場合、トランザクションはロールバックされません。
- CORBA::OBJECT_NOT_EXIST() 例外がクライアントに返されます。

注記　オペレーション呼び出しを受信したときにトランザクションを自動的に開始する場合、各 CORBA インターフェイスに対して、AUTOTRAN を Yes に設定します。AUTOTRAN を Yes に設定しても、インターフェイスが既にトランザクション・モードにある場合は無効です。AUTOTRAN の詳細については、『BEA Tuxedo CORBA トランザクション』を参照してください。

- 発生した例外に基づいて、次のように、メッセージがユーザ・ログ (ULOG) ファイルに書き込まれます。

```
TobjS::ActivateObjectFailed
       "TPFW_CAT:24: ERROR: Activating object - application raised
       TobjS::ActivateObjectFailed. Reason = reason. Interface =
       interfaceName, OID = oid"
```

reason はユーザ指定の理由を示し、*interfaceName* と *oid* はそれぞれ呼び出された CORBA オブジェクトの インターフェイス ID とオブジェクト ID を示します。

TobjS::OutOfMemory
"TPFW_CAT:22: ERROR: Activating object - application raised TobjS::OutOfMemory. Reason = reason. Interface = *interfaceName*, OID = *oid*"

reason はユーザ指定の理由を示し、*interfaceName* と *oid* はそれぞれ呼び出された CORBA オブジェクトの インターフェイス ID とオブジェクト ID を示します。

CORBA::Exception
"TPFW_CAT:25: ERROR: Activating object - CORBA Exception not handled by application. Exception ID = exceptionID. Interface = *interfaceName*, OID = *oid*"

exceptionID は例外のインターフェイス ID を示し、*interfaceName* と *oid* はそれぞれ呼び出された CORBA オブジェクトのインターフェイス ID とオブジェクト ID を示します。

その他の例外
"TPFW_CAT:26: ERROR: Activating object - Unknown Exception not handled by application. Exception ID = exceptionID. Interface = interfaceName, OID = oid"

exceptionID は例外のインターフェイス ID を示し、*interfaceName* と *oid* はそれぞれ呼び出された CORBA オブジェクトのインターフェイス ID とオブジェクト ID を示します。

Tobj_ServantBase::_add_ref()

概要 サーバントのリファレンスを追加します。このメソッドは、マルチスレッド・サーバ・アプリケーションの開発をサポートします。

注記 BEA Tuxedo リリース 8.0 以降で作成するアプリケーションでは、`TP::application_responsibility()` メソッドの代わりにこのメソッドを使用します。

C++ バインディング

```
void _add_ref()
```

引数 特にありません。

説明 このメソッドは、サーバントのリファレンスが必要な場合に呼び出します。このメソッドを呼び出すと、サーバントのリファレンス数が 1 つ増えます。

戻り値 特にありません。

例

```
myServant * servant = new intf_i();
if(servant != NULL)
   servant->_add_ref();
```

Tobj_ServantBase::deactivate_object()

概要 オブジェクト ID とサーバントとの関連付けを削除します。このメソッドを使用すると、アプリケーションでは、オブジェクトが非活性化される前にオブジェクトの状態のすべてまたは一部を保存できます。状態は、共用メモリ、通常のフラット・ファイル、またはデータベース・ファイルに保存できます。

C++ バインディング

```
class Tobj_ServantBase : public PortableServer::ServantBase {
public:
        virtual void deactivate_object(const char* stroid,
                        TobjS::DeactivateReasonValue  reason) {}
};
```

引数

`stroid`

オブジェクト ID を文字列形式で指定します。オブジェクト ID は、クラスのこのインスタンスを一意に識別します。

注記 このリリースでは、オブジェクト ID の長さに関する制約がなくなりました。

`reason`

このメソッドを呼び出す原因となったイベントを示します。`reason` コードは、以下のいずれかです。

`DR_METHOD_END`

メソッドの終了後にオブジェクトが非活性化されることを示します。オブジェクトの非活性化方針が次の場合に使用されます。

- `method`
- `transaction` (アクティブなトランザクションがない場合のみ)
- `process` (`TP::deactivateEnable()` が呼び出された場合)

`DR_SERVER_SHUTDOWN`

サーバが通常の方法でシャットダウンされるためにオブジェクトが非活性化されることを示します。オブジェクトの非活性化方針が次の場合に使用されます。

- `transaction` (トランザクションがアクティブな場合のみ)
- `process`

サーバを通常の方法でシャットダウンする場合、そのサーバが関与しているすべてのトランザクションがロールバックの対象としてマークされることに注意してください。

DR_TRANS_ABORTED

この reason コードは、transaction 活性化方針が設定されたオブジェクト専用です。これは、トランザクションがクライアントまたはシステムによって自動的に開始された場合に発生します。deactivate_object() メソッドがこの理由コードで呼び出された場合、トランザクションはロールバック対象としてマークされます。

DR_TRANS_COMMITTING

この reason コードは、transaction 活性化方針が設定されたオブジェクト専用です。これは、トランザクションがクライアントまたは TP フレームワークによって開始された場合に発生します。これは、オブジェクトが呼び出されたトランザクションに対して Current.commit() オペレーションが呼び出されたことを示します。deactivate_object() メソッドは、トランザクション・マネージャが 2 フェーズ・コミット・アルゴリズムを開始する直前、つまり prepare がリソース・マネージャに送信される前に呼び出されます。

CORBA オブジェクトでは、DR_TRANS_COMMITTING reason コードで deactivate_object() メソッドが呼び出されたときに、トランザクションの結果に関して判断できます。このメソッドは、Current.rollback_only() メソッドを呼び出すことで、トランザクションをロールバックさせることができます。そうでない場合は、2 フェーズ・コミット・アルゴリズムが続行されます。トランザクションは、Current.rollback_only() がこのメソッドで呼び出されなかったという理由以外でもコミットされないことがあります。トランザクションに関与するその他の CORBA オブジェクトまたはリソース・マネージャも、トランザクションのロールバックを支持できます。

DR_EXPLICIT_DEACTIVATE

アプリケーションがこのオブジェクトに対して TP::deactivateEnable(-,-,-) を実行したためにオブジェクトが非活性化されることを示します。これは、process 活

性化方針を設定されたオブジェクトに関してのみ発生します。

説明 オブジェクトの非活性化は、CORBA オブジェクトのインプリメンテーションに設定されている活性化方針に従って、システムまたはアプリケーションによって開始されます。deactivate_object() メソッドは、CORBA オブジェクトが非活性化される前に呼び出されます。これらの方針および使い方については、「ICF の構文」を参照してください。

CORBA オブジェクトのインプリメンテーションの活性化方針が method の場合にクライアントによって呼び出されるメソッド、または活性化方針が transaction の場合にトランザクションの処理が終わったときに呼び出されるメソッドの実行後に、非活性化が発生する場合もあります。また、活性化方針が transaction または process の場合に、サーバがシャットダウンされると発生します。

さらに、BEA Tuxedo ソフトウェアでは、process または method が設定された CORBA オブジェクトを、TP::deactivateEnable() および TP::deactivateEnable(-,-,-) メソッドによってユーザ側で制御して非活性化することができます。TP::deactivateEnable をオブジェクトのメソッド内で呼び出すと、メソッドの終了時にそのオブジェクトが非活性化されます。transaction 活性化方針が設定されたオブジェクトで TP::deactivateEnable を呼び出すと、例外 (TobjS::IllegalOperation) が発生し、TP フレームワークは何の処理も行いません。process 活性化方針が設定されたオブジェクトに対して TP::deactivateEnable(-,-,-) を呼び出すと、そのオブジェクトは非活性化されます。詳細については、TP::deactivateEnable() を参照してください。

注記 サーバのシャットダウン時には、アクティブ・オブジェクト・マップに残っているすべてのオブジェクトに対して deactivate_object メソッドが呼び出されます。オブジェクトが AOM に追加される方法としては、TP フレームワークによって暗黙的に追加される場合 (オンデマンド活性化手法。TP::create_servant とサーバントの activate_object メソッド) と、ユーザが TP::create_active_object_reference を使用して明示的に追加する方法があります。

プログラマは、activate_object() および deactivate_object() メソッドとクライアントが明示的に呼び出すメソッドを使用して、オブジェクトの状態を管理できます。これらのメソッドを使用してオブジェクトの状態を管理する方法は、アプリケーションのニーズによって異なります。これらのメソッドの使い方については、『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。

transaction 活性化方針が設定された CORBA オブジェクトでは、DR_TRANS_COMMITTING 理由コードで deactivate_object() メソッドが呼び出されたときに、トランザクションの結果に関して判断できます。このメソッドは、Current.rollback_only() メソッドを呼び出すことで、トランザクションをロールバックさせることができます。そうでない場合は、2 フェーズ・コミット・アルゴリズムが続行されます。トランザクションは、Current.rollback_only() がこのメソッドで呼び出されなかったという理由以外でもコミットされないことがあります。トランザクションに関与するその他の CORBA オブジェクトまたはリソース・マネージャも、トランザクションのロールバックを支持できます。

制約　このメソッドが呼び出されたときにオブジェクトがトランザクションに関与している場合、オブジェクトが呼び出された理由に基づいて、実行可能な処理が制約されます。オブジェクトがトランザクションに関与していた場合、活性化方針が transaction で、呼び出しの reason コードは以下のいずれかです。

DR_TRANS_ABORTED
: このメソッドでは、CORBA オブジェクトを呼び出せません。tpcall() は許可されません。トランザクションを一時停止したり開始したりすることはできません。

DR_TRANS_COMMITTING
: このメソッドでは、CORBA オブジェクトを呼び出せません。tpcall() は許可されません。トランザクションを一時停止したり開始したりすることはできません。

こうした制約がある理由は、トランザクション・バウンドの活性化方針を設定されたオブジェクトの非活性化が、トランザクションのトランザクション・マネージャから TP フレームワークに対する呼び出しによって制御されるからです。reason コード DR_TRANS_COMMITTING で呼び出しが行われた場合、トランザクション・マネージャは 2 フェーズ・コミットのフェーズ 1 (準備) を実行しています。この段階では、トランザクションを一時停止する

呼び出し、または新しいトランザクションを開始する呼び出しを行うことはできません。別のプロセス内の CORBA オブジェクトを呼び出すにはそのプロセスがトランザクションに参加する必要があり、トランザクション・マネージャは既に準備フェーズを実行しているので、この呼び出しを行うとエラーが発生します。[1] トランザクションに関与していない CORBA オブジェクトを呼び出すには、そのトランザクションを一時停止する必要があるので、これもエラーの原因となります。同じことは `tpcall()` にもあてはまります。

同じように、`reason` コード `DR_TRANS_ABORTED` での呼び出しが行われた場合、トランザクション・マネージャは既にアボート中です。トランザクション・マネージャがアボート中の場合、トランザクションを一時停止したり、新しいトランザクションを開始したりすることはできません。この制約は、`DR_TRANS_COMMITTING` に関しても適用されます。

戻り値 特にありません。

1. 理論的には、トランザクションに関与する CORBA オブジェクトを同じプロセスで呼び出す場合、新しいトランザクションをトランザクション・マネージャに登録する必要がないので、この呼び出しは有効ということになります。ただし、CORBA オブジェクトに対する呼び出しがプロセスで行われることを保証できないので、この手法を使用することはお勧めしません。

例外 クライアントによって呼び出される CORBA オブジェクト・メソッドで例外が発生した場合、TP フレームワークが例外をキャッチして、最終的にクライアントに返します。これは、deactivate_object() を呼び出して例外が発生した場合にもあてはまります。

クライアントには、deactivate_object() で発生した例外について通知されません。アプリケーション・コードでは、格納されている CORBA オブジェクトの状態が矛盾していないかどうかをチェックする必要があります。たとえば、アプリケーション・コードで、deactivate_object() がオブジェクトの状態を正しく保存したかどうかを示す永続性フラグを保存すると便利です。このフラグを activate_object() でチェックします。

deactivate_object() の実行中にエラーが発生した場合、アプリケーション・コードでは、CORBA 標準例外または DeactivateObjectFailed 例外を生成する必要があります。deactivate_object() が TP フレームワークで呼び出されると、TP フレームワークは例外をキャッチして、以下のイベントが発生します。

- オブジェクトが非活性化されます。
- クライアントがトランザクションを開始した場合、トランザクションはロールバックされません。
- クライアントには、deactivate_object() で発生した例外は通知されません。
- 発生した例外に基づいて、次のように、メッセージがユーザ・ログ (ULOG) ファイルに書き込まれます。

```
TobjS::DeactivateObjectFailed
       "TPFW_CAT:27: ERROR: De-activating object - application
       raised TobjS::DeactivateObjectFailed. Reason = reason.
       Interface = interfaceName, OID = oid"
```

reason はユーザ指定の理由を示し、*interfaceName* と *oid* はそれぞれ呼び出された CORBA オブジェクトの インターフェイス ID とオブジェクト ID を示します。

```
CORBA::Exception
       "TPFW_CAT:28: ERROR: De-activating object - CORBA Exception
       not handled by application. Exception ID = exceptionID.
       Interface = interfaceName, OID = oid"
```

exceptionID は例外のインターフェイス ID を示し、*interfaceName* と *oid* はそれぞれ呼び出された CORBA オブジェクトのインターフェイス ID とオブジェクト ID を示します。

その他の例外

```
"TPFW_CAT:29: ERROR: De-activating object - Unknown
Exception not handled by application. Exception ID =
exceptionID. Interface = interfaceName, OID = oid"
```

exceptionID は例外のインターフェイス ID を示し、*interfaceName* と *oid* はそれぞれ呼び出された CORBA オブジェクトのインターフェイス ID とオブジェクト ID を示します。

Tobj_ServantBase::_is_reentrant()

概要 オブジェクトが同時リエントラント呼び出しをサポートしていることを示します。このメソッドは、マルチスレッド・サーバ・アプリケーションの開発をサポートします。

C++ バインディング

```
CORBA::Boolean _is_reentrant()
```

引数 特にありません。

説明 BEA Tuxedo サーバ・インフラストラクチャでは、このメソッドを使用して、サーバント・インプリメンテーションがリエントラント呼び出しをサポートしているかどうかを判断します。リエントラントをサポートするには、複数のスレッドがオブジェクトと対話する場合に状態の整合性を保護するためのコードをサーバントに含める必要があります。

`Tobj_ServantBase` クラスには、`FALSE` を返す `_is_reentrant` メソッドのデフォルト・インプリメンテーションが用意されています。

戻り値 `CORBA::Boolean`

サーバントがリエントラントをサポートしている場合に `TRUE` を返します。

例

```
CORBA::Boolean Simple_i::_is_reentrant()
{       TP::userlog("_is_reentrant called in thread %ld",
                        (unsigned long)SIMPTHR_GETCURRENTTHREADID);
        return CORBA_TRUE;
}
```

Tobj_ServantBase::_remove_ref()

概要 サーバントのリファレンスを解放します。このメソッドは、マルチスレッド・サーバ・アプリケーションの開発をサポートします。

注記 BEA Tuxedo リリース 8.0 以降で作成するアプリケーションでは、`TP::application_responsibility()` メソッドで使用していた C++ の「delete」文の代わりに、このメソッドを使用します。

C++ バインディング

```
void _remove_ref()
```

パラメータ 特にありません。

説明 このメソッドは、サーバントのリファレンスが必要でなくなった場合に呼び出します。このメソッドを呼び出すと、サーバントのリファレンス数が 1 つ減ります。`_remove_ref()` メソッドによってリファレンス数がゼロになると、このメソッドは自身の `this` ポインタに対して C++ の「delete」文を呼び出し、サーバントを削除します。

戻り値 特にありません。

例

```
if(servant != NULL)
   servant->_remove_ref();
```

TP インターフェイス

TP インターフェイスでは、アプリケーション・コードで呼び出せるサービス・メソッドのセットが提供されます。これは、アプリケーション・コードで安全に呼び出せる TP フレームワーク内の唯一のインターフェイスです。その他のインターフェイスは、システム・コードでのみ呼び出すことを目的としたコールバック・メソッドを持っています。

このインターフェイスの目的は、ポータブル・オブジェクト・アダプタ (POA)、CORBA ネーミング・サービス、および BEA Tuxedo システムで提供される基本 API に対する呼び出しの代わりに、アプリケーション・コードで呼び出し可能な高レベルの呼び出しを行えるようにすることです。これらの呼び出しにより、プログラマはより簡単な API を使用できるので、複雑な基本 API を使用せずに済みます。TP インターフェイスでは、CORBA API を拡張した BEA Tuxedo ソフトウェアの以下の 2 つの機能を暗黙的に使用します。

- ファクトリと FactoryFinder オブジェクト
- ファクトリ・ベース・ルーティング

FactoryFinder オブジェクトの詳細については、「FactoryFinder インターフェイス」を参照してください。ファクトリ・ベース・ルーティングの詳細については、『BEA Tuxedo アプリケーションの設定』を参照してください。

使用上の注意

- サーバ・アプリケーションの初期化時に、アプリケーションはアプリケーション・ファクトリ用のオブジェクト・リファレンスを作成します。作成後、ファクトリのオブジェクト・リファレンスをファクトリ `id` フィールドと一緒に渡して、`register_factory()` メソッドを呼び出します。サーバの解放 (シャットダウン) 時には、アプリケーションは `unregister_factory()` メソッドを使用して、ファクトリの登録を削除します。
- `TP` クラスは C++ ネイティブ・クラスです。
- `TP.h` ファイルには、`TP` クラスの宣言および定義が格納されています。

C++ 宣言 (TP.h 内)

C++ のマッピングは次のとおりです。

```
class TP {
public:
   static CORBA::Object_ptr  create_object_reference(
                                  const char*    interfaceName,
                                  const char*   stroid,
                                  CORBA::NVList_ptr criteria);
   static CORBA::Object_ptr  create_active_object_reference(
                                  const char*  interfaceName,
                                  const char* stroid,
                                  Tobj_Servant servant);
   static CORBA::Object_ptr  get_object_reference();
   static void               register_factory(
                                  CORBA::Object_ptr factory_or,
                                  const char*       factory_id);
   static void               unregister_factory(
                                  CORBA::Object_ptr factory_or,
                                  const char*       factory_id);
   static void               deactivateEnable()
   static void               deactivateEnable(
                                  const char*  interfaceName,
                                  const char* stroid,
                                  Tobj_Servant servant);
   static CORBA::ORB_ptr     orb();
   static Tobj_Bootstrap*    bootstrap();
   static void               open_xa_rm();
   static void               close_xa_rm();
   static int                userlog(char*, ... );
  static char*               get_object_id(CORBA::Object_ptr obj);
   static void               application_responsibility(
                                Tobj_Servant servant);
};
```

TP::application_responsibility()

概要 アプリケーションがサーバントの存続期間を管理することを TP フレームワークに通知します。

注記 BEA Tuxedo リリース 8.0 以上で作成するアプリケーションではこのメソッドを使用しないでください。代わりに、Tobj_ServantBase::_add_ref() メソッドを使用します。

C++ バインディング

```
static void application_responsibility(Tobj_Servant servant);
```

引数

サーバント
TP フレームワークに既に認識されているサーバントに対するポインタ。

例外

TobjS::InvalidServant
指定されたサーバントが NULL であることを示します。

説明 このメソッドは、アプリケーションがサーバントの存続期間を制御することを TP フレームワークに通知します。このメソッドを呼び出すと、TP フレームワークがオブジェクトを非活性化した後に、つまりサーバントの deactivate_object メソッドを呼び出した後に、オブジェクトに対して何の処理も行われません。

サーバントをアプリケーション側で処理する場合、その他の C++ インスタンスと同じように、不要になった時点でサーバントを削除しなければなりません。

サーバントが TP フレームワークに認識されていない (活性化されていない) 場合、この呼び出しは無効です。

戻り値 特にありません。

TP::bootstrap()

概要 `Tobj::Tobj_Bootstrap` オブジェクトに対するポインタを返します。Bootstrap オブジェクトは、FactoryFinder オブジェクト、インターフェイス・リポジトリ、TransactionCurrent オブジェクト、および SecurityCurrent オブジェクトへの初期リファレンスにアクセスする場合に使用します。

C++ バインディング

```
static Tobj_Bootstrap* TP::bootstrap();
```

引数 特にありません。

戻り値 `bootstrap()` は、正常に終了すると、サーバ・アプリケーションの開始時に TP フレームワークで作成された `Tobj::Tobj_Bootstrap` オブジェクトに対するポインタを返します。

例外 特にありません。

説明 TP フレームワークでは、初期化の一部として `Tobj::Tobj_Bootstrap` オブジェクトが作成されます。したがって、アプリケーション・コードでほかの `Tobj::Tobj_Bootstrap` オブジェクトをサーバ内に作成する必要はありません。

Caution: `Tobj::Tobj_Bootstrap` オブジェクトの所有権は TP フレームワークにあるので、サーバ・アプリケーション・コードで Bootstrap オブジェクトを破棄してはなりません。

注記 CORBA INS ブートストラップ処理メカニズムを使用し、セキュリティ用の `SecurityCurrent` またはトランザクション用の `TransactionCurrent` を使用しない場合は、Bootstrap オブジェクトを使用する必要はありません。

TP::close_xa_rm()

概要 呼び出しプロセスのリンク先の XA リソース・マネージャをクローズします。

C++ バインディング

```
static void TP::close_xa_rm ();
```

引数 特にありません。

説明 close_xa_rm() メソッドは、呼び出しプロセスのリンク先の XA リソース・マネージャをクローズします。XA リソース・マネージャは、Oracle や Informix などのデータベース・ベンダから提供されます。

注記 この呼び出しの機能も、Tobj::TransactionCurrent::close_xa_rm() によって提供されます。TransactionCurrent オブジェクトのオブジェクト・リファレンスを取得する必要がないので、サーバ・アプリケーションでリソース・マネージャをクローズする方法としては、TP::close_xa_rm() メソッドを使う方がはるかに便利です。TransactionCurrent オブジェクトのリファレンスは、Bootstrap オブジェクトから取得できます。Bootstrap オブジェクトのリファレンスの取得方法については、「TP::bootstrap()」を参照してください。TransactionCurrent オブジェクトの詳細については、「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」と『BEA Tuxedo CORBA トランザクション』を参照してください。

グローバル・トランザクションに関与するサーバごとに Server::release() メソッドから 1 回、このメソッドを呼び出す必要があります。グローバル・トランザクションに関与しているすべてのサーバだけでなく、XA リソース・マネージャにリンクされたサーバも含まれますが、XA 準拠のリソース・マネージャに実際にはリンクされていません。

close_xa_rm() メソッドは、リソース・マネージャに固有のクローズ呼び出しの代わりに呼び出します。リソース・マネージャの初期化セマンティクスはそれぞれ異なるので、特定のリソース・マネージャをクローズするための情報を、BEA Tuxedo システムの UBBCONFIG ファイルの GROUPS セクションにある CLOSEINFO パラメータに指定します。

CLOSEINFO 文字列の形式は、基となるリソース・マネージャのデータベース・ベンダごとに異なります。CLOSEINFO パラメータの詳細については、『BEA Tuxedo アプリケーションの設定』と『BEA Tuxedo のファイル形式とデータ記述方法』の「ubbconfig(5)」のリファレンス・ページを参照してください。また、XA ライブラリを使用するアプリケーションの開発およびインストール方法については、データベース・ベンダのマニュアルを参照してください。

戻り値 特にありません。

例外 CORBA::BAD_INV_ORDER
アクティブなトランザクションがあります。トランザクションがアクティブになっている場合、リソース・マネージャをクローズすることはできません。

Tobj::RMFailed
tx_close() 呼び出しによって、エラー戻りコードが返されました。

注記 TP フレームワークのその他の例外と違い、Tobj::RMFailed 例外は、TobjS_c.h (TobjS.idl から派生) ではなく、tobj_c.h (tobj.idl から派生) で定義されます。これは、ネイティブ・クライアントでも XA リソース・マネージャをオープンできるからです。したがって、返される例外は、ネイティブ・クライアント・コードおよび Server::release() (ネイティブ・クライアントと共有される代替メカニズムである TransactionCurrent::close_xa_rm を使用している場合) で想定される例外と一致します。

TP::create_active_object_reference()

概要 オブジェクト・リファレンスを作成し、オブジェクトを事前活性化します。

C++ バインディング

```
static CORBA::Object_ptr
    create_active_object_reference(
            const char*       interfaceName,
            const char*      stroid,
            Tobj_Servant     servant);
```

引数

`interfaceName`
オブジェクトの完全修飾インターフェイス名を含む文字列を指定します。

`stroid`
`ObjectId` を文字列形式で指定します。`ObjectId` は、クラスのこのインスタンスを一意に識別します。プログラマは、`ObjectId` に指定する情報を決めます。一例として、データベース・キーの保持に使用する方法があります。オブジェクト識別子の値を選択、つまり一意性のレベルを決定することは、アプリケーション・デザインの一環です。BEA Tuxedo ソフトウェアでは、オブジェクト・リファレンスの一意性は保証されません。オブジェクト・リファレンスを文字列化する場合など、オブジェクト・リファレンスをコピーしたり、BEA Tuxedo 環境の外で共有したりすることがあるからです。

サーバント
アプリケーションで既に作成して初期化したサーバントに対するポインタです。

例外

`TobjS::InvalidInterface`
指定された interfaceName が NULL であることを示します。

`TobjS::InvalidObjectId`
指定された stroid が NULL であることを示します。

`TobjS::ServantAlreadyActive`
サーバントは別の `ObjectId` で使用されているので、オブジェクトを明示的に活性化することができませんでした。サーバントは、1 つの `ObjectId` でのみ使用できます。異なる `ObjectId` を持つオブジェクトを事前活性化するには、アプリケーションで複数のサーバントを作成し、`ObjectId` ごとに個別に事前活性化する必要があります。

`TobjS::ObjectAlreadyActive`
: `ObjectId` が既にアクティブ・オブジェクト・マップで使用されているので、オブジェクトを明示的に活性化することができませんでした。ある特定の `ObjectId` には、1 つのサーバントしか関連付けられません。別のサーバントに変更するには、アプリケーションでまずオブジェクトを事前活性化してからもう一度活性化する必要があります。

`TobjS::IllegalOperation`
: オブジェクトにプロセス・バウンド活性化方針が設定されていないので、オブジェクトを明示的に活性化することができませんでした。

説明 このメソッドでは、オブジェクト・リファレンスを作成し、オブジェクトを事前活性化します。作成されたオブジェクト・リファレンスは、オブジェクトへのアクセスで使用するクライアントに渡されます。

通常、アプリケーションではこのメソッドを以下の 2 か所で呼び出します。

- `Server::initialize()` 内。最初の呼び出しで活性化しなくて済むように、プロセス・オブジェクトを事前活性化します。
- クライアントに返すオブジェクト・リファレンスを作成するすべてのメソッド内。

このメソッドを使用すると、アプリケーションでは最初の呼び出しの前にオブジェクトを活性化できます。この処理が有用となる理由については、「明示的な活性化」を参照してください。ユーザはまずサーバントを作成し、状態を設定してから、`create_active_object_reference` を呼び出します。TP フレームワークでは、サーバントおよび `ObjectId` 文字列をアクティブ・オブジェクト・マップに入れます。その結果、TP フレームワークで既に `Server::create_servant` を呼び出し、サーバント・ポインタを受け取ってから、`servant::activate_object` を呼び出した場合とまったく同じ状態になります。

活性化されたオブジェクトは、プロセス・バウンド活性化方針で宣言されたインターフェイス用でなければなりません。それ以外の場合、例外が発生します。

オブジェクトを非活性化すると、クライアントで保持されていたオブジェクト・リファレンスを使用して、そのオブジェクトをほかのプロセスで活性化することができます。この処理が問題となる状況については、「明示的な活性化」を参照してください。

注記　ユーザ制御の同時実行方針オプションが ICF ファイルで設定されている場合、このメソッドに関する制約が 1 つあります (「パラレル・オブジェクト」を参照してください)。
`TP::create_active_object_reference` メソッドは、ユーザ制御の同時実行が設定されたインターフェイスを渡されると、`TobjS::IllegalOperation` 例外をスローします。AOM はユーザ制御の同時実行が設定されている場合に使用されないので、TP フレームワークには活性化されたオブジェクトをこのサーバに接続する方法がありません。

注意　インターフェイスのオブジェクトを事前活性化する場合、そのインターフェイスの ICF ファイルで `process` 活性化方針を指定する必要があります。ただし、ICF ファイルで `process` 活性化方針を指定すると、次の問題が発生する可能性があります。

問題

1. インターフェイス A のすべてのオブジェクトが事前活性化されるように、SERVER1 を記述します。オブジェクトが TP フレームワークによってオンデマンドで活性化されないようにするために、インターフェイスの `activate_object` メソッドを記述して、`ActivateObjectFailed` 例外がスローされるようにします。
2. SERVER2 もインターフェイス A のオブジェクトを実装します。ただし、SERVER2 では、オブジェクトを事前活性化する代わりに、TP フレームワークでオンデマンド方式でオブジェクトを活性化させます。
3. 管理者は SERVER1 と SERVER2 を同じグループとしてコンフィギュレーションした場合、クライアントでは、SERVER2 からインターフェイス A のオブジェクト・リファレンスを取得して、オブジェクトを呼び出すことができます。ロード・バランシングにより、インターフェイス A のオブジェクトの活性化が SERVER1 に委任されまが、`activate_object` メソッドが `ActivateObjectFailed` 例外をスローするため、SERVER1 ではインターフェイス A のオブジェクトを活性化できません。

対策

この問題を防ぐには、管理者が SERVER1 と SERVER2 をそれぞれ別のグループとしてコンフィギュレーションします。グループを定義するには、UBBCONFIG ファイルの SERVERS セクションを使用します。

戻り値　新しく作成されたオブジェクト・リファレンス。

TP::create_object_reference()

概要 オブジェクト・リファレンスを作成します。作成されたオブジェクト・リファレンスは、オブジェクトへのアクセスで使用するクライアントに渡されます。

C++ バインディング

```
static CORBA::Object_ptr    TP::create_object_reference (
                               const char* interfaceName,
                               const char* stroid,
                               CORBA::NVList_ptr criteria);
```

引数

`interfaceName`

オブジェクトの完全修飾インターフェイス名を含む文字列を指定します。
インターフェイス名を取得するには、次のインターフェイス・タイプ・コード ID 関数を呼び出します。

```
const char* _tc_<CORBA interface name>::id();
```

<CORBA interface name> は任意のオブジェクト・クラス名です。たとえば、次のように入力します。

```
char* idlname = _tc_Simple->id();
```

`stroid`

`ObjectId` を文字列形式で指定します。`ObjectId` は、クラスのこのインスタンスを一意に識別します。プログラマは、`ObjectId` に指定する情報を決めます。一例として、`ObjectId` でデータベース・キーを保持する方法があります。オブジェクト識別子の値を選択、つまり一意性のレベルを決定することは、アプリケーション・デザインの一環です。BEA Tuxedo ソフトウェアでは、オブジェクト・リファレンスの一意性は保証されません。オブジェクト・リファレンスを文字列として渡す場合など、オブジェクト・リファレンスをコピーしたり、BEA Tuxedo ドメインの外で共有したりすることがあるからです。オブジェクト・リファレンスに対して複数の呼び出しを並列実行できるように、`ObjectId` を一意に指定することをお勧めします。

注記 このリリースでは、`ObjectId` の長さに関する制約がなくなりました。

`criteria`
オブジェクト・リファレンスのファクトリ・ベース・ルーティングで使用する名前付き値のリストを指定します。リストはオプションで、`CORBA::NVList` 型です。ファクトリ・ベース・ルーティングを使用するかどうかは任意で、使用する場合はこの引数に依存します。ファクトリ・ベース・ルーティングを使用しない場合は、この引数に 0 (ゼロ) を渡します。
BEA Tuxedo システムの管理者は、`UBBCONFIG` ファイルでルーティング規則を指定して、ファクトリ・ベース・ルーティングを設定します。この機能の詳細については、オンライン・マニュアルの『BEA Tuxedo アプリケーションの設定』を参照してください。

例外 `create_object_reference()` メソッドの例外は以下のとおりです。

`InvalidInterface`
指定された `interfaceName` が NULL であることを示します。

`InvalidObjectId`
指定された `stroid` が NULL であることを示します。

説明 `create_object_reference()` メソッドの呼び出しは、サーバ・アプリケーションの役割です。このメソッドによって、オブジェクト・リファレンスが作成されます。作成されたオブジェクト・リファレンスは、オブジェクトへのアクセスで使用するクライアントに渡されます。

通常、サーバ・アプリケーションではこのメソッドを以下の 2 か所で呼び出します。

- `Server::initialize()` 内。サーバのファクトリを作成します。
- クライアントに返すオブジェクト・リファレンスを作成するファクトリ・メソッド内。

`create_object_reference()` メソッドの呼び出し方法とタイミングの例については、『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。

戻り値 `Object`
新しく作成されたオブジェクト・リファレンス。

例 次のコード例は、criteria 引数の使用方法を示しています。

```
CORBA::NVList_ptr criteria;
CORBA::Long branch_id = 7;
CORBA::Long account_id = 10001;
CORBA::Any any_val;

// リストを作成して  _var に割り当てて、終了時にクリーンアップ
CORBA::ORB::create_list (2, criteria);
CORBA::NVList_var criteria_var(criteria);

// BRANCH_ID を追加
any_val <<= branch_id;
criteria->add_value("BRANCH_ID", any_val, 0);

// ACCOUNT_ID を追加
any_val <<= account_id;
criteria->add_value("ACCOUNT_ID", any_val, 0);

// オブジェクト・リファレンスを作成
TP::create_object_reference ("IDL:BankApp/Teller:1.0",
"Teller_01", criteria);
```

TP::deactivateEnable()

概要 CORBA オブジェクトのアプリケーション制御の非活性化を有効にします。

C++ バインディング current-object 形式

```
static void           TP::deactivateEnable();
```

any-object 形式

```
static void        TP::deactivateEnable(
                           const char*  interfaceName,
                           const char* stroid,
                            Tobj_Servant servant);
```

引数

`interfaceName`
オブジェクトの完全修飾インターフェイス名を含む文字列を指定します。

`stroid`
非活性化するオブジェクトの `ObjectId` を文字列形式で指定します。

サーバント
stroid に関連付けられたサーバントに対するポインタ。

例外 `deactivateEnable()` メソッドの例外は以下のとおりです。

`IllegalOperation`
`TP::deactivateEnable` メソッドが、活性化方針を `transaction` に設定されたオブジェクトによって呼び出されたことを示します。

`TobjS::ObjectNotActive`
any-object 形式では、指定されたオブジェクトは非活性化されていなかった、つまり、`stroid` および `servant` パラメータはアクティブ・オブジェクト・マップ内でオブジェクトを識別できなかったので、活性化できませんでした。

説明 このオブジェクトを使用すると、同時実行されているオブジェクト (オブジェクトを呼び出したメソッドの終了時)、または別のオブジェクトを非活性化できます。このメソッドは、プロセス・バウンド活性化方針が設定されたオブジェクトに対してのみ使用できます。このメソッドにより、`process` 活性化方針が設定されたオブジェクトをさらに柔軟に扱うことができます。

注記　シングル・スレッド・サーバの場合、
`TP::deactivateEnable(interface, object id, servant)` メソッドを使用すると、オブジェクトを非活性化できます。ただし、オブジェクトがトランザクションに参加している場合、オブジェクトが非活性化されるのは、トランザクションがコミットまたはロールバックしたときです。トランザクションがコミットまたはロールバックする前にオブジェクトに対して呼び出しが行われた場合、オブジェクトは非活性化されません。

必要な動作が確実に実行されるようにするには、オブジェクトがトランザクションに参加していないことを確認するか、`TP::deactivateEnable()` を呼び出してからトランザクションが終了するまでオブジェクトに対して呼び出しが行われないようにします。

注記　マルチスレッド・サーバの場合、
`TP::deactivateEnable(interface, object id, servant)` メソッドを使用して、オブジェクトごとのサーバのオブジェクトを非活性化することはできません。このメソッドでは、要求ごとのサーバのオブジェクトを非活性化することができますが、ほかのスレッドがオブジェクトを操作しているので、非活性化は延期されます。

呼び出されるオーバロード関数の種類によって、操作は次のようになります。

current-object 形式
プロセス・バウンド活性化方針が設定されたオブジェクトのメソッド内から呼び出された場合、実行中のオブジェクトは、メソッドの実行が終了した後に非活性化されます。
method 活性化方針が設定されたオブジェクトのメソッド内から呼び出された場合、こうしたオブジェクトの通常の動作 (事実上 NOOP) と同じ効果があります。
オブジェクトが非活性化されると、TP フレームワークではまずアクティブ・オブジェクト・マップからオブジェクトが削除されます。次に、理由コード `DR_METHOD_END` を使用して、オブジェクトに関連付けられたサーバントの `deactivate_object` メソッドが呼び出されます。

any-object 形式
アプリケーションでは、`ObjectId` と関連付けられたサーバントを指定することで、オブジェクトの非活性化を要求します。

オブジェクトが実行中の場合、TP では非活性化対象としてオブジェクトをマークして、current-object 形式と同じように、オブジェクトのメソッドが終了するまで待機してから非活性化します。オブジェクトが実行中でない場合、TP フレームワークでは直ちにオブジェクトを非活性化できます。非活性化のステータスは、呼び出し元に通知されません。オブジェクトが非活性化されると、TP フレームワークではまずアクティブ・オブジェクト・マップからオブジェクトが削除されます。次に、理由コード `DR_EXPLICIT_DEACTIVATE` を使用して、オブジェクトに関連付けられたサーバントの `deactivate_object` メソッドが呼び出されます。

非活性化を要求されたオブジェクトに `transaction` 活性化方針が設定されていた場合、`IllegalOperation` 例外が発生します。これは、こうしたオブジェクトを非活性化すると、BEA Tuxedo トランザクション・マネージャからのトランザクション終了の通知と競合する場合があるからです。

戻り値　特にありません。

TP::get_object_id ()

概要 サーバが、TP フレームワークで作成されたオブジェクト・リファレンスから ObjectId 文字列を取り出せるようにします。

C++ バインディング

```
char* TP::get_object_id(Corba::Object_ptr obj);
```

引数 obj

ObjectId を取り出すオブジェクト・リファレンス。

例外 TobjS::InvalidObject

オブジェクトが nil であるか、TP フレームワークで作成されていません。

説明 このメソッドを使用すると、サーバが、TP フレームワークで作成されたオブジェクト・リファレンスから ObjectId 文字列を取り出せます。オブジェクト・リファレンスがクライアント ORB などによって作成され、TP フレームワークで作成されていない場合、例外が発生します。

呼び出し元では、オブジェクト・リファレンスが不要になった場合に、戻り値に対して CORBA::string_free を呼び出す必要があります。

戻り値 オブジェクト・リファレンスの作成時に TP::create_object_reference または TP::create_active_object_reference に渡された ObjectId 文字列。

TP::get_object_reference()

概要 現在のオブジェクトへのポインタを返します。

C++ バインディング

```
static CORBA::Object_ptr  TP::get_object_reference ();
```

引数 特にありません。

`get_object_reference()` が `Server::initialize()` または `Server::release()` 内から呼び出された場合、アプリケーションの TP オブジェクトの実行スコープ外で呼び出されたものと見なされるので、`TobjS::NilObject` 例外が発生します。

例外 `get_object_reference()` メソッドの例外は次のとおりです。

`NilObject`
: メソッドが、アプリケーションの CORBA オブジェクトの実行スコープ外で呼び出されたことを示します。`reason` 文字列には、`OutOfScope` が格納されます。

説明 このメソッドは、現在のオブジェクトへのポインタを返します。返された `CORBA::Object_ptr` ポインタは、クライアントに渡すことができます。

戻り値 CORBA オブジェクトの実行スコープ内で呼び出された場合、`get_object_reference()` メソッドは、現在のオブジェクトの `CORBA::Object_ptr` を返します。それ以外の場合、`TobjS::NilObject` 例外が発生します。

TP::open_xa_rm()

概要 呼び出しプロセスのリンク先の XA リソース・マネージャをオープンします。

C++ バインディング

```
static void TP::open_xa_rm();
```

引数 特にありません。

例外 `Tobj::RMFailed`
`tx_open()` 呼び出しによって、エラー戻りコードが返されました。

注記 TP フレームワークのその他の例外と違い、この例外は、`TobjS_c.h` (`TobjS.idl` から派生) ではなく、`tobj_c.h` (`tobj.idl` から派生) で定義されます。これは、ネイティブ・クライアントでも XA リソース・マネージャをオープンできるからです。したがって、返される例外は、ネイティブ・クライアント・コードおよび `Server::release()` (ネイティブ・クライアントと共有される代替メカニズムである `TransactionCurrent::close_xa_rm` を使用している場合) で想定される例外と一致します。

説明 `open_xa_rm()` メソッドは、呼び出しプロセスのリンク先の XA リソース・マネージャをオープンします。XA リソース・マネージャは、Oracle や Informix などのデータベース・ベンダから提供されます。

注記 この呼び出しの機能も、`Tobj::TransactionCurrent::close_xa_rm()` によって提供されます。ただし、TransactionCurrent オブジェクトのオブジェクト・リファレンスを取得する必要がないので、サーバ・アプリケーションでリソース・マネージャをオープンする方法としては、`TP::open_xa_rm()` メソッドを使う方がはるかに便利です。TransactionCurrent オブジェクトのリファレンスは、Bootstrap オブジェクトから取得できます。Bootstrap オブジェクトのリファレンスの取得方法については、「TP::bootstrap()」を参照してください。TransactionCurrent オブジェクトの詳細については、「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」と『BEA Tuxedo CORBA トランザクション』を参照してください。

グローバル・トランザクションに関与するサーバごとに Server::initialize() メソッドから 1 回、このメソッドを呼び出す必要があります。グローバル・トランザクションに関与しているすべてのサーバだけでなく、XA リソース・マネージャにリンクされたサーバも含まれますが、XA 準拠のリソース・マネージャに実際にはリンクされていません。

open_xa_rm() メソッドは、リソース・マネージャに固有のオープン呼び出しの代わりに呼び出します。リソース・マネージャの初期化セマンティクスはそれぞれ異なるので、特定のリソース・マネージャをオープンするための情報を、UBBCONFIG ファイルの GROUPS セクションにある OPENINFO パラメータに指定します。

OPENINFO 文字列の形式は、基となるリソース・マネージャのデータベース・ベンダごとに異なります。CLOSEINFO パラメータの詳細については、『BEA Tuxedo アプリケーションの設定』と『BEA Tuxedo のファイル形式とデータ記述方法』の「ubbconfig(5)」のリファレンス・ページを参照してください。また、XA ライブラリを使用するアプリケーションの開発およびインストール方法については、データベース・ベンダのマニュアルを参照してください。

注記　呼び出しプロセスにリンクできるリソース・マネージャは 1 つだけです。

戻り値　特にありません。

TP::orb()

概要 ORB オブジェクトに対するポインタを返します。

C++ バインディング

```
static CORBA::ORB_ptr TP::orb();
```

引数 特にありません。

例外 特にありません。

説明 ORB オブジェクトにアクセスすると、アプリケーションでは、string_to_object() や object_to_string() などの ORB オペレーションを呼び出すことができます。

注記 TP フレームワークが ORB オブジェクトを所有しているので、アプリケーションが削除してはなりません。

戻り値 orb() は、正常に終了すると、サーバ・プログラムの開始時に TP フレームワークで作成された ORB オブジェクトに対するポインタを返します。

TP::register_factory()

概要 BEA Tuxedo FactoryFinder オブジェクトを見つけて、BEA Tuxedo ファクトリに登録します。

C++ バインディング

```
static void TP::register_factory(
            CORBA::Object_ptr factory_or, const char* factory_id);
```

引数

`factory_or`
`TP::create_object_reference()` メソッドを使用して、アプリケーション・ファクトリ用に作成されたオブジェクト・リファレンスを指定します。

`factory_id`
アプリケーション・ファクトリを識別するための文字列識別子を指定します。この文字列を構成する際のヒントについては、『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。

例外 `register_factory()` メソッドの例外は以下のとおりです。

`TobjS::CannotProceed`

`TobjS::InvalidName`
`id` 文字列が空であることを示します。フィールドに空白または制御文字が含まれている場合にも例外が発生します。

`TobjS::InvalidObject`
`factory` 値が nil であることを示します。

`TobjS::RegistrarNotAvailable`

注記 その他、FactoryFinder がトランザクションに関与できない場合にも、この例外が発生します。したがって、`TP::register_factory()` および `TP::unregister_factory()` 呼び出しの前に、現在のトランザクションを一時停止しなければならない場合があります。トランザクションの一時停止と再開方法については、オンライン・マニュアルの『BEA Tuxedo CORBA トランザクション』を参照してください。

`TobjS::OverFlow`
`id` 文字列が 128 バイト (許可されている最大長) より長いことを示します。

説明 このメソッドでは、BEA Tuxedo FactoryFinder オブジェクトを見つけて、BEA Tuxedo ファクトリに登録します。通常、`TP::register_factory()` は、サーバがファクトリを作成するときに、`Server::initialize()` から呼び出します。`register_factory()` メソッドでは、BEA Tuxedo FactoryFinder オブジェクトを見つけて、BEA Tuxedo ファクトリに登録します。

Caution: コールバック・オブジェクト (つまり、POA を介して共同クライアント / サーバによって直接作成されるオブジェクト) を FactoryFinder を登録してはなりません。

戻り値 特にありません。

TP::unregister_factory()

概要 BEA Tuxedo FactoryFinder オブジェクトを見つけて、ファクトリを削除します。

C++ バインディング

```
static void TP::unregister_factory (
            CORBA::Object_ptr factory_or, const char* factory_id);
```

引数

`factory_or`

`TP::create_object_reference()` メソッドを使用して、アプリケーション・ファクトリ用に作成されたオブジェクト・リファレンスを指定します。

`factory_id`

アプリケーション・ファクトリを識別するための文字列識別子を指定します。この文字列を構成する際のヒントについては、『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。

例外 `unregister_factory()` メソッドの例外は以下のとおりです。

`CannotProceed`

`InvalidName`

`id` 文字列が空であることを示します。フィールドに空白または制御文字が含まれている場合にも例外が発生します。

`RegistrarNotAvailable`

注記 その他、FactoryFinder がトランザクションに関与できない場合にも、この例外が発生します。したがって、`TP::register_factory()` および `TP::unregister_factory()` 呼び出しの前に、現在のトランザクションを一時停止しなければならない場合があります。トランザクションの一時停止と再開方法については、オンライン・マニュアルの『BEA Tuxedo CORBA トランザクション』を参照してください。

`TobjS::OverFlow`

`id` 文字列が 128 バイト (許可されている最大長) より長いことを示します。

説明 このメソッドでは、BEA Tuxedo FactoryFinder オブジェクトを見つけて、ファクトリを削除します。通常、`TP::unregister_factory()` は、サービス・ファクトリの登録を削除するために `Server::release()` から呼び出されます。

戻り値 特にありません。

TP::userlog()

概要 ユーザ・ログ (ULOG) ファイルにメッセージを書き込みます。

C++ バインディング

```
static int    TP::userlog(char*, ...);
```

引数 printf(3S) スタイルの形式を指定します。printf(3S) 引数については、C または C++ リファレンス・マニュアルで説明されています。

例外 特にありません。

説明 userlog() メソッドは、ユーザ・ログ (ULOG) ファイルにメッセージを書き込みます。メッセージは、時刻 (hhmmss)、システム名、プロセス名、および呼び出しプロセスのプロセス ID で構成されるタグを付けて、ULOG ファイルに追加されます。タグの最後にはコロンが付けられます。

サーバ・アプリケーションでは、userlog() によるメッセージをアプリケーション・エラーのデバッグで有用となるメッセージに限定することをお勧めします。ULOG が重要度の低いメッセージでいっぱいになると、実際のエラーの特定が難しくなります。

戻り値 userlog() メソッドは、出力された文字数を返し、出力エラーが発生した場合には、負の値を返します。出力エラーには、現在のログ・ファイルのオープン・エラーや書き込みエラーがあります。

例 次のコード例は、TP::userlog() メソッドの使用方法を示しています。

```
userlog (“System exception caught: %s”, e.get_id());
```

CosTransactions::TransactionalObject インターフェイス (任意)

このインターフェイスの使用は避けてください。このインターフェイスの使用は任意となったので、トランザクションに関与するオブジェクトに対して、このインターフェイスの下位インターフェイスを使用する必要はありません。プログラマは、never または ignore トランザクション方針を指定して、オブジェクトがトランザクションに関与しないように指定できます。トランザクションの処理にインターフェイスを使用する必要はありません。指定するのはトランザクション方針だけです。

注記　CORBA のオブジェクト・トランザクション・サービスでは、すべての要求をトランザクションのスコープ内で実行する必要はありません。トランザクションのスコープ外で呼び出された場合の動作は各オブジェクトで決定します。トランザクション・コンテキストを必要とするオブジェクトは、標準例外を生成する場合があります。

エラー、例外、およびエラー・メッセージ

TP フレームワークで生成される例外

TP フレームワークでは以下の例外が発生します。発生した例外は、エラーが発生したクライアントに返されるか、TP で検出されます。

```
CORBA::INTERNAL
CORBA::OBJECT_NOT_EXIST
CORBA::OBJ_ADAPTER
CORBA::INVALID_TRANSACTION
CORBA::TRANSACTION_ROLLEDBACK
```

これらの例外の理由は明確ではないので、TP フレームワークでは、例外が発生するたびに、理由を示すエラー・メッセージをユーザ・ログ・ファイルに書き込まれます。

サーバ・アプリケーション・コード内の例外

クライアントによって呼び出されたメソッド内で発生した例外は、クライアントによって呼び出されたメソッドで発生した例外と同じように、クライアントに返されます。

TP フレームワークの以下のコールバック・メソッドは、オブジェクトに対するクライアントの要求以外のイベントによって開始されます。

```
Tobj_ServantBase::activate_object()
Tobj_ServantBase::deactivate_object()
Server::create_servant()
```

これらのメソッドで例外が発生した場合、まったく同じ例外がクライアントに通知されるわけではありません。ただし、これらの各メソッドは、reason 文字列を含む例外を生成するように定義されています。TP フレームワークでは、コールバックによって生成された例外がキャッチされ、reason 文字列

がユーザ・ログ・ファイルに書き込まれます。TP フレームワークでは、クライアントに例外を返すこともできます。これらの例外の詳細については、TP フレームワークの各コールバック・メソッドの説明を参照してください。

例

`Tobj_ServantBase::deactivate_object()` の場合、次のコードは `DeactivateObjectFailed` 例外をスローします。

```
throw TobjS::DeactivateObjectFailed( “deactivate failed to save
                                        state!”);
```

このメッセージは、時刻 (hhmmss)、システム名、プロセス名、および呼び出しプロセスのプロセス ID で構成されるタグを付けて、ユーザ・ログ・ファイルに追加されます。タグの最後にはコロンが付けられます。上記の throw 文によって、次の行がユーザ・ログ・ファイルに書き込まれます。

```
151104.T1!simpapps.247: APPEXC: deactivate failed to save state!
```

`151104` は時刻 (3:11:04pm)、`T1` はシステム名、`simpapps` はプロセス名、`247` はプロセス ID をそれぞれ示し、`APPEXC` はメッセージがアプリケーション例外メッセージであることを示します。

例外とトランザクション

CORBA オブジェクト・メソッドまたは TP フレームワークのコールバック・メソッドで例外が発生しても、TP フレームワークがトランザクションを開始していない限り、トランザクションは自動的にはロールバックされません。例外が発生した条件に基づいてトランザクションをロールバックする場合は、アプリケーション・コードで `Current.rollback_only()` を呼び出す必要があります。

CORBA オブジェクトに対する入れ子になった呼び出しに関する制約

TP フレームワークには、CORBA オブジェクトに対する入れ子になった呼び出しに関して制約があります。制約の内容は次のとおりです。

- CORBA オブジェクト A のメソッドをクライアントから呼び出す場合、CORBA オブジェクト A を、CORBA オブジェクト A のクライアントとして機能している別の CORBA オブジェクト B によって呼び出すことはできません。

TP フレームワークでは、別の CORBA オブジェクトが既にメソッド呼び出しを処理しているオブジェクトのクライアントとして機能していることを検出すると、呼び出し元に `CORBA::OBJ_ADAPTER` 例外を返します。

注記　アプリケーション・コードではこの動作に依存しないようにしてください。つまり、ユーザはこの動作が発生することを前提に処理を行わないでください。この制限は、今後のリリースで取り払われます。

章

4 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

ここでは、以下の内容について説明します。

- ブートストラップ処理が必要な理由
- サポートされているブートストラップ処理メカニズム
- BEA ブートストラップ処理メカニズム
- Bootstrap オブジェクト API
- Bootstrap オブジェクトのプログラミング例
- インターオペラブル・ネーミング・サービス・ブートストラップ処理メカニズム

ブートストラップ処理が必要な理由

クライアント・アプリケーションが BEA Tuxedo オブジェクトと通信するためには、オブジェクト・リファレンスを取得する必要があります。オブジェクト・リファレンスがないと、通信はできません。この問題を解決するために、クライアント・アプリケーションではブートストラップ処理メカニズムを使用して BEA Tuxedo ドメインの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。

サポートされているブートストラップ処理メカニズム

リリース 8.0 以降の Tuxedo では、2 つのブートストラップ処理メカニズムがサポートされます。

- BEA ブートストラップ処理メカニズム

  BEA クライアント ORB を使用する場合に使用します。

- インターオペラブル・ネーミング・サービス・ブートストラップ処理メカニズム

  別のベンダのクライアント ORB を使用する場合に使用します。

注記　BEA Tuxedo ソフトウェアに付属の CORBA C++ クライアントおよび Java クライアントでは、インターオペラブル・ネーミング・サービスのブートストラップ処理メカニズムを使用できますが、性能上の理由により推奨はできません。

BEA ブートストラップ処理メカニズム

BEA ブートストラップ処理メカニズムでは、Bootstrap オブジェクトを使用します。Bootstrap オブジェクトは、クライアントとサーバ両方の (リモート CORBA オブジェクトではなく) ローカル・プログラミング・オブジェクトです。Bootstrap オブジェクトが作成されるとき、そのコンストラクタは BEA Tuxedo IIOP リスナ / ハンドラのネットワーク・アドレスを必要とします。その情報が提供されると、ブートストラップ処理オブジェクトでは BEA Tuxedo ドメインの主要なリモート・オブジェクトのオブジェクト・リファレンスを生成できます。それらのオブジェクト・リファレンスは、BEA Tuxedo ドメインで利用可能なサービスにアクセスするために使用できます。

Bootstrap オブジェクトの機能

Bootstrap オブジェクトは、次の BEA Tuxedo CORBA インターフェイスのオブジェクト・リファレンスにアクセスする必要のあるクライアントまたはサーバ・アプリケーションによって作成されます。

- FactoryFinder
- セキュリティ
- インターフェイス・リポジトリ
- ネーミング・サービス
- ノーティフィケーション・サービス
- Tobj_SimpleEvents サービス
- トランザクション

Bootstrap オブジェクトは、IIOP リスナ / ハンドラのアドレスの形式によっては特定の BEA Tuxedo ドメインへの最初の接続を表す場合があります。ヌル・スキーマ Universal Resource Locator (URL) 形式が使用される場合に (バージョン 5.1 以前の BEA WebLogic Enterprise リリースと BEA Tuxedo リ

リース 8.0 でサポートされている唯一のアドレス形式)、Bootstrap オブジェクトは最初の接続を表します。ただし、この URL 形式が使用される場合は、Bootstrap オブジェクトが作成されるまで接続は行われません。アドレス形式と接続回数の詳細については、「Tobj_Bootstrap」を参照してください。

BEA Tuxedo CORBA リモート・クライアントについては、Bootstrap オブジェクトは BEA Tuxedo IIOP リスナ / ハンドラのホストとポートを使用して作成されます。しかし、BEA Tuxedo ネイティブのクライアント・アプリケーションとサーバ・アプリケーションでは、ホストとポートを指定する必要はありません (特定の BEA Tuxedo ドメインで実行されるため)。IIOP リスナ / ハンドラのホストとポート ID は、BEA Tuxedo ドメインのコンフィギュレーション情報に含まれています。

Bootstrap オブジェクトは、その作成後に、特定の BEA Tuxedo ドメインにあるオブジェクトのオブジェクト・リファレンスに対する要求を満たします。異なる Bootstrap オブジェクトを使用すると、アプリケーションで複数のドメインを使用できます。

Bootstrap オブジェクトを使用すると、次のオブジェクトのオブジェクト・リファレンスを取得できます。

- SecurityCurrent

  SecurityCurrent オブジェクトは、BEA Tuxedo ドメイン内のセキュリティ・コンテキストを確立するために使用します。クライアントは、SecurityCurrent オブジェクトの `principal_authenticator` 属性から PrincipalAuthenticator を取得できます。

- TransactionCurrent

  TransactionCurrent オブジェクトは、BEA Tuxedo トランザクションに参加するために使用します。基本的なオペレーションは以下の通りです。

  - Begin

    トランザクションを開始します。以降のオペレーションは、このトランザクションのスコープ内で発生します。

  - Commit

    トランザクションを終了します。このクライアント・アプリケーションですべてのオペレーションが正常に終了しています。

  - Roll back

トランザクションをアボートします。ほかのすべてのパーティシパントにロールバックを指示します。

- Suspend

  現在のトランザクションの参加を一時停止します。このオペレーションは、トランザクションを示すオブジェクトを返し、クライアント・アプリケーションが後でトランザクションを再開できるようにします。

- Resume

  指定したトランザクションの参加を再開します。

■ FactoryFinder

FactoryFinder オブジェクトは、ファクトリを取得するために使用します。BEA Tuxedo システムで、ファクトリはアプリケーション・オブジェクトを作成するために使用します。FactoryFinder では、次のような方法でファクトリを検索できます。

- ファクトリ・オブジェクトのオブジェクト・リファレンスと一致する利用可能なすべてのファクトリのリストを取得する (`find_factories`)。
- `id` と種類で構成される名前コンポーネントと一致するファクトリを取得する (`find_one_factory`)。
- 特定の種類の利用可能な最初のファクトリを取得する (`find_one_factory_by_id`)。
- 特定の種類の利用可能なすべてのファクトリのリストを取得する (`find_factories_by_id`)。
- 登録されているすべてのファクトリのリストを取得する (`list_factories`)。

■ InterfaceRepository

インターフェイス・リポジトリには、BEA Tuxedo ドメイン内でインプリメントされる CORBA オブジェクトのインターフェイス記述が含まれています。動的起動インターフェイス (DII) を使用するクライアントでは、インターフェイス・リポジトリのリファレンスがないと CORBA 要求の構造体を構築できません。ActiveX クライアントはこの特殊なケー

スです。COM/IIOP ブリッジのインプリメンテーションでは内部で DII が使用されるので、インターフェイス・リポジトリのリファレンスを取得しなければなりません (ただしこれはデスクトップ・クライアントに対して透過的)。

- NamingService

  NamingService オブジェクトは、ルート名前空間のリファレンスを取得するために使用します。このオブジェクトを使用すると、ORB は名前空間のルートを検索します。

- NotificationService

  NotificationService オブジェクトは、CosNotification サービス内のイベント・チャネル・ファクトリ (CosNotifyChannelAdmin::EventChannelFactory) のリファレンスを取得するために使用します。BEA Tuxedo システムで、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します。

- Tobj_SimpleEventsService

  Tobj_SimpleEventsService オブジェクトは、BEA シンプル・イベント・サービス内のイベント・チャネル・ファクトリ (Tobj_SimpleEvents::ChannelFactory) のリファレンスを取得するために使用します。BEA Tuxedo システムで、ChannelFactory は BEA シンプル・イベント・サービス・チャネルの検索に使用します。

FactoryFinder オブジェクトとインターフェイス・リポジトリ・オブジェクトは、環境オブジェクト・ライブラリでインプリメントされません。しかし、それらのオブジェクトは BEA Tuxedo ドメインに固有であり、したがって概念的には SecurityCurrent オブジェクトおよび TransactionCurrent オブジェクトと似ています。

Bootstrap オブジェクトは、クライアント・アプリケーションと BEA Tuxedo ドメインの間の関連 (セッション) を意味します。この関連のコンテキストで、Bootstrap オブジェクトはほかの Current オブジェクト (SecurityCurrent と TransactionCurrent) との包含関係を強制します。Current オブジェクトは、このドメインの範囲内で、Bootstrap オブジェクトが存在している間のみ有効です。

注記　新しい URL アドレス形式 (corbaloc://*hostname*:*port_number*) を使用している場合の SecurityCurrent の解決はローカルの処理です。つまり、クライアントから IIOP リスナ / ハンドラへの接続は行われません。

また、クライアントでは各 Current オブジェクトにつきインスタンスは 1 つしか利用できません。Current オブジェクトが既に存在する場合でも、新たな Current オブジェクトの作成が失敗することはありません。失敗するのではなく、既存のオブジェクトのもう 1 つのリファレンスが渡されます。つまり、クライアント・アプリケーションは Current オブジェクトの単一インスタンスの複数のリファレンスを持つことになります。

Current オブジェクトの新しいインスタンスを作成するには、まず Bootstrap オブジェクトの destroy_current() メソッドを呼び出す必要があります。この呼び出しですべての Current オブジェクトが無効になりますが、BEA Tuxedo ドメインとのセッションは破棄されません。destroy_current() を呼び出した後は、既存の Bootstrap オブジェクトを使用して BEA Tuxedo ドメイン内で Current オブジェクトの新しいインスタンスを作成できます。

別のドメインの Current オブジェクトを取得するには、別の Bootstrap オブジェクトを作成する必要があります。同時に複数の Bootstrap オブジェクトを持つこともできますが、「アクティブ」にできる (Current オブジェクトを関連付けることができる) のは 1 つの Bootstrap オブジェクトだけです。したがって、アプリケーションでは「アクティブ」な Bootstrap オブジェクトの destroy_current() を呼び出してから別の Bootstrap オブジェクト (アクティブな Bootstrap オブジェクトになる) の新しい Current オブジェクトを取得する必要があります。

注記　複数のドメインのオブジェクトにアクセスする必要がある場合は、オブジェクトをローカル・ドメインにインポートするか、複数のドメインにアクセスするようにアプリケーションをコンフィギュレーションします。マルチ・ドメイン・コンフィギュレーションの詳細については、『BEA Tuxedo Domains コンポーネント』の「Configuring Multiple CORBA Domains」を参照してください。

サーバとネイティブ・クライアントは BEA Tuxedo ドメインの中に存在します。したがって、「セッション」は確立されません。ただし、同じ包含関係が強制されます。サーバとネイティブ・クライアントは、`//host:port` ではなく空の文字列を指定してそれらが含まれているドメインにアクセスします。

注記　Bootstrap オブジェクトを使用する場合、クライアント・アプリケーションとサーバ・アプリケーションでは `ORB::resolve_initial_references()` メソッドではなく `Tobj_Bootstrap::resolve_initial_references()` メソッドを使用する必要があります。

サポートされている BEA リモート・クライアントの種類

表 4-1 は、Bootstrap オブジェクトを使用してほかの環境オブジェクト (FactoryFinder、SecurityCurrent、TransactionCurrent、InterfaceRepository など) にアクセスできるリモート・クライアントの種類を示しています。これらのクライアントは、BEA Tuxedo CORBA ソフトウェアに付属しています。サード・パーティ製のクライアント ORB では、CORBA インターオペラブル・ネーミング・サービスを使用する必要があります。

表 4-1 サポートされている BEA リモート・クライアント

クライアント	説明
CORBA C++	CORBA C++ クライアント・アプリケーションでは、BEA Tuxedo C++ 環境オブジェクトを使用して BEA Tuxedo ドメインの CORBA オブジェクトにアクセスし、BEA Tuxedo オブジェクト・リクエスト・ブローカ (ORB) を使用して CORBA オブジェクトからの要求を処理します。BEA Tuxedo システムの開発コマンドを使用すると、このタイプのクライアント・アプリケーションをビルドできます (『BEA Tuxedo コマンド・リファレンス』を参照)。

表 4-1 サポートされている BEA リモート・クライアント (続き)

クライアント	説明
CORBA Java	CORBA Java クライアント・アプリケーションでは、BEA Tuxedo Java 環境オブジェクトを使用して BEA Tuxedo ドメインの CORBA オブジェクトにアクセスします。ただし、CORBA オブジェクトからの要求の処理には BEA Tuxedo ORB 以外の ORB 製品が使用されます。このタイプのクライアント・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。
ActiveX	BEA Tuxedo オートメーション環境オブジェクトを使用して BEA Tuxedo ドメインの CORBA オブジェクトにアクセスし、ActiveX クライアントを使用して CORBA オブジェクトからの要求を処理します。Application Builder を使用すると、ActiveX クライアント・アプリケーションからアクセスできるように CORBA オブジェクトのバインディングを作成できます。ActiveX クライアント・アプリケーションは、Microsoft Visual Basic、Delphi、PowerBuilder などの開発ツールでビルドします。

機能と制限事項

Bootstrap オブジェクトには、以下の機能と制限があります。

- クライアント・アプリケーションには複数の Bootstrap オブジェクトが共存できますが、Current オブジェクト (Transaction と Security) を所有できるのは 1 つの Bootstrap オブジェクトだけです。クライアント・アプリケーションでは、1 つのドメインと関連付けられている Bootstrap オブジェクトの `destroy_current()` を呼び出してから、別のドメインの Current オブジェクトを取得する必要があります。別々の BEA Tuxedo ドメインと接続を確立する複数の Bootstrap オブジェクトを持つこともできますが、Current オブジェクトは 1 組しか有効になりません。既存の Current オブジェクトを破棄しないと、ほかの Current オブジェクトを取得することはできません。

- 有効な SecurityCurrent オブジェクトを備えるドメイン以外の、セキュリティが有効な BEA Tuxedo ドメインに対してメソッド呼び出しを行うと、その呼び出しは失敗し、CORBA::NO_PERMISSION 例外が返されます。
- 有効な TransactionCurrent オブジェクトを備えるドメイン以外の BEA Tuxedo ドメインに対するメソッド呼び出しは、トランザクションのスコープ内で実行されません。
- Bootstrap オブジェクトから返されるトランザクション・オブジェクトとセキュリティ・オブジェクトは、Current オブジェクトの BEA インプリメンテーションです。ほかの (ネイティブの) Current オブジェクトが環境に存在する場合でも、それらのオブジェクトは無視されます。

Bootstrap オブジェクト API

Bootstrap オブジェクト・アプリケーション・プログラミング・インターフェイス (API) は、まず OMG 定義言語 (IDL) で記述し (移植性のため)、その後に C++、Java、および ActiveX で記述します。C++ および Java の記述では、特定の BEA Tuxedo ドメインの Bootstrap オブジェクトをビルドするために必要なコンストラクタが追加されます。

Tobj モジュール

表 4-2 は、各 ID で返されるオブジェクト・リファレンスを示しています。

表 4-2 返されるオブジェクト・リファレンス

ID	返されるオブジェクト・リファレンス (C++ クライアント)	返されるオブジェクト・リファレンス (Java クライアント)
FactoryFinder	FactoryFinder オブジェクト (Tobj::FactoryFinder)	FactoryFinder オブジェクト (com.beasys.Tobj.FactoryFinder)

表 4-2 返されるオブジェクト・リファレンス (続き)

ID	返されるオブジェクト・リファレンス (C++ クライアント)	返されるオブジェクト・リファレンス (Java クライアント)
InterfaceRepository	InterfaceRepository オブジェクト (`CORBA::Repository`)	InterfaceRepository オブジェクト (`org.omg.CORBA.Repository`)
NameService	CORBA ネーミング・サービス (`Tobj::NameService`)	CORBA ネーミング・サービス (`com.beasys.Tobj.NameService`)
NotificationService	EventChannelFactory オブジェクト (`CosNotifyChannelAdmin::EventChannelFactory`)	EventChannelFactory オブジェクト (`CosNotifyChannelAdmin.EventChannelFactory`)
SecurityCurrent	SecurityCurrent オブジェクト (`SecurityLevel2::Current`)	SecurityCurrent オブジェクト (`org.omg.SecurityLevel2.Current`)
TransactionCurrent	OTS Current オブジェクト (`Tobj::TransactionCurrent`)	OTS Current object (`com.beasys.Tobj.TransactionCurrent`)
Tobj_SimpleEventsService	BEA Simple Events ChannelFactory object (`Tobj_SimpleEvents::ChannelFactory`)	BEA シンプル・イベント ChannelFactory オブジェクト (`Tobj_SimpleEvents.ChannelFactory`)

表 4-3 は、Tobj モジュールの例外を示しています。

表 4-3 Tobj モジュールの例外

C++ の例外	Java の例外	説明
`Tobj::InvalidName`	`com.beasys.Tobj.InvalidName`	`id` が表 4-2 で指定されたどの名前でもない場合に生成されます。サーバでは、`SecurityCurrent` が渡された場合にも `resolve_initial_references` で `InvalidName` が生成されます。
`Tobj::InvalidDomain`	`com.beasys.Tobj.InvalidDomain`	サーバ・アプリケーションで、BEA Tuxedo サーバ環境が起動していない場合に生成されます。

表 4-3 Tobj モジュールの例外 (続き)

C++ の例外	Java の例外	説明
CORBA:: NO_PERMISSION	org.omg.CORBA. NO_PERMISSION	id が TransactionCurrent または SecurityCurrent で、クライアントの別の Bootstrap オブジェクトが Current オブジェクトを所有している場合に生成されます。
BAD_PARAM	org.omg.CORBA. BAD_PARAM	オブジェクトがニルである場合、またはオブジェクトに格納されているホスト名が接続と一致しない場合に生成されます。
IMP_LIMIT	org.omg.CORBA. IMP_LIMIT	register_callback_port メソッドが複数回呼び出された場合に発生します。

C++ のマッピング

リスト 4-1 は、Tobj_bootstrap.h ファイルでの C++ の宣言を示しています。

コード リスト 4-1 Tobj_boostrap.h の宣言

```
#include <CORBA.h>

class Tobj_Bootstrap {
public:
       Tobj_Bootstrap(CORBA::ORB_ptr orb, const char* address);
       CORBA::Object_ptr resolve_initial_references(
                 const char* id);
       void register_callback_port(CORBA::Object_ptr objref);
       void destroy_current( );
};
```

Java マッピング

リスト 4-2 は、Tobj_Bootstrap.java のマッピングを示しています。

コード リスト 4-2　Tobj_Bootstrap.java **のマッピング**

```
package com.beasys;

public class Tobj_Bootstrap {
       public Tobj_Bootstrap(org.omg.CORBA.ORB orb,
                             String address)
              throws org.omg.CORBA.SystemException;
public class Tobj_Bootstrap {
      public Tobj_Bootstrap(org.omg.CORBA.ORB orb, String address,
                             java.applet.Applet applet)
                throws org.omg.CORBA.SystemException;

public void register_callback_port(orb.omg.CORBA.Object objref)
               throws org.omg.CORBA.SystemException;

public org.omg.CORBA.Object
                    resolve_initial_references(String id)
               throws Tobj.InvalidName,
                      org.omg.CORBA.SystemException;
public void destroy_current()
               throws org.omg.CORBA.SystemException;
}
```

Microsoft デスクトップ・クライアントのマッピング

Bootstrap オブジェクトは、Microsoft デスクトップでインプリメントされるクライアントが使用できるように BEA ActiveX クライアント・ソフトウェアで提供されます。デスクトップ・クライアントでは、次の 2 つのインターフェイスを使用できます。

- Visual Basic (VB)、Delphi、または PowerBuilder クライアント向けのオートメーション・インターフェイス

- 動的なクライアント (Visual Basic) で必要なオートメーション・インターフェイスと静的にリンクされたクライアント (C++) で必要な Vtable インターフェイスの両方を備えるデュアル・インターフェイス。ActiveX クライアントの Bootstrap オブジェクトでは、ハイブリッドのデュアル・インターフェイスが提供されます。

オートメーションのマッピング

リスト 4-3 は、オートメーション Bootstrap インターフェイスのマッピングを示しています。

コード リスト 4-3 **オートメーション (デュアル)** Bootstrap **インターフェイスのマッピング**

```
interface DITobj_Bootstrap : IDispatch
{
     HRESULT Initialize(
               [in] BSTR address);

     HRESULT CreateObject(
              [in] BSTR progid,
               [out, retval] IDispatch** rtrn);

     HRESULT destroy_current();
};
```

C++ メンバ関数

この節では、BEA ブートストラップ処理メカニズムでサポートされる C++ メンバ関数について説明します。

Tobj_Bootstrap

概要 Bootstrap オブジェクトのコンストラクタです。

C++ マッピング

```
Tobj_Bootstrap(CORBA::ORB_ptr orb, const char* address);
            throws Tobj::BAD_PARAM
                   org.omg.CORBA.SystemException;
```

パラメータ

`orb`

クライアントの ORB オブジェクトへのポインタ。Bootstrap オブジェクトの内部では、`orb` の `string_to_object` メソッドが使用されます。

`address`

BEA Tuxedo ドメインの IIOP リスナ / ハンドラのアドレス。このアドレスは、クライアントの種類と必要なセキュリティのレベルに応じて異なる形式で指定します。クライアントには次の 3 種類があります。

- リモート・クライアント

  BEA Tuxedo CORBA でサポートされるリモート・クライアントの説明については、「サポートされている BEA リモート・クライアントの種類」という節を参照してください。

  リモート・クライアントの場合、`address` では、クライアント・アプリケーションが BEA Tuxedo ドメインにアクセスするために使用する IIOP リスナ / ハンドラのネットワーク・アドレスを指定します。

  アドレスは、次のいずれかの形式で指定できます。

  “*//hostname*:*port_number*”
  “*//#.#.#.#*:*port_number*”
  “corbaloc:*//hostname*:*port_number*”
  “corbalocs:*//hostname*:*port_number*”

  最初の形式の場合、ドメインではローカルの名前解決機能 (通常は DNS) を使用して hostname のアドレスが検索されます。ホスト名はリモート・マシンでなければならず、ローカルの名前解決機能では hostname がそのリモート・マシンのアドレスに明確に解決されなければなりません。

注記　*hostname* は、先頭を文字で始める必要があります。

2 番目の形式の場合、#.#.#.# にはドット区切りの 10 進数を指定します。ドット区切りの 10 進数形式では、それぞれの # に 0 ～ 255 の数字を指定します。このドット区切りの 10 進数は、リモート・マシンの IP アドレスを表します。

最初と 2 番目の両方の形式で、port_number ではドメイン・プロセスが要求をリッスンする TCP ポート番号を指定します。port_number は、0 ～ 65535 の数字でなければなりません。

TCP/IP アドレスは 1 つまたは複数を指定できます。複数のアドレスは、カンマ区切りのリストを使用して指定します。たとえば、次のように入力します。

```
//m1.acme:3050
//m1.acme:3050,//m2.acme:3050,//m3.acme:3051
```

複数のアドレスを指定すると、BEA Tuxedo ソフトウェアでは接続が確立されるまで左から右の順序でアドレスが試されます。接続が試行されているときにアドレスの構文エラーが検出されると、BAD_PARAM 例外が直ちに呼び出し側に返され、BEA Tuxedo ソフトウェアによって接続の試行がアボートされます。たとえば、上記のカンマ区切りリストの最初のアドレスが //m1.3050 である場合は、構文エラーが検出され、接続の試行がアボートされます。BEA Tuxedo ソフトウェアがアドレス・リストの最後まで達しても有効なアドレスで接続を試行できない場合、つまりリストのどのアドレスにも接続を確立できない場合は、INVALID_DOMAIN 例外が呼び出し側に返されます。INVALID_DOMAIN 以外の例外が生成された場合、その例外は呼び出し側に直ちに返されます。

BEA Tuxedo では、ランダムなアドレスの選択もサポートされています。ランダムなアドレスの選択を使用する場合は、アドレス・リストの任意のメンバをかっこで囲まれたパイプ区切り (|) のネットワーク・アドレス・グループとして指定します。たとえば、次のように入力します。

```
(//m1.acme:3050|//m2.acme:3050),//m1.acme:7000
```

この形式を使用すると、BEA Tuxedo システムではかっこで囲まれたアドレスのいずれか (//m1.acme:3050 または //m2.acme:3050) がランダムに選択されます。INVALID_DOMAIN 以外の例外が生成された場合、その例外は呼び出し側に直ちに返されます。選択したアドレスで接続を確立できない場合は、かっこの後にある次の要素で接続が試行

されます。接続を確立できないうちに文字列の最後に達してしまった場合は、INVALID_DOMAIN 例外が呼び出し側にスローされます。

注記　次の形式でアドレス・リストを指定した場合、

```
(//m1.acme:3050||//m2.acme:3050),//r1.acme:7000
```

パイプ区切りリスト内のヌル・アドレスは無効と判断されます。BEA Tuxedo ソフトウェアで無効なアドレスがランダムに選択された場合は、BAD_PARAM 例外が呼び出し側に返され、BEA Tuxedo ソフトウェアによって接続の試行がアボートされます。

アドレス文字列は、TOBJADDR 環境変数または Tobj_Bootstrap コンストラクタのアドレス・パラメータで指定できます。

TOBJADDR 環境変数の詳細については、『BEA Tuxedo アプリケーションの設定』の「Managing Remote Client Applications」を参照してください。ただし、Tobj_Bootstrap で指定されたアドレスの方が常に TOBJADDR 環境変数よりも優先されます。TOBJADDR 環境変数を使用してアドレス文字列を指定するには、Tobj_Bootstrap の address パラメータで空の文字列を指定する必要があります。

注記　C++ アプリケーションでは TOBJADDR は環境変数であり、Java アプリケーションの場合はプロパティ、Java アプレットの場合は HTML パラメータです。

3 番目と 4 番目の形式は Uniform Resource Locator (URL) アドレス形式と呼ばれ、BEA WebLogic Enterprise バージョン 5.1 で導入されました。ヌル・スキーマ URL アドレス形式 (//hostname:port_number) の場合と同じように、URL アドレス形式を使用して IIOP リスナ / ハンドラの位置を指定します。ただし、corbaloc URL アドレス形式を使用する場合、IIOP リスナ / ハンドラへのクライアント・アプリケーションの初期接続はプリンシパル (クライアント) の ID が認証されるまで、またはユーザが最初のオペレーションを開始するまで遅延されます。corbalocs URL アドレス形式を使用する場合でも corbaloc と同じ接続の遅延がありますが、それに加えて、クライアント・アプリケーションがセキュア・ソケット・レイヤ (SSL) プロトコルを使用して ISL/ISH との初期接続を行います。表 4-4 は、2 つの URL アドレス形式の違いを示しています。

表 4-4 corbaloc **および** corbalocs URL **アドレス形式の違い**

URL **アドレス形式**	**動作モードの違い**
`corbaloc`	IIOP リスナ / ハンドラの呼び出しが保護されません。オプションで、IIOP リスナ / ハンドラに対して SSL プロトコルをコンフィギュレーションします。 注記　プリンシパルは、`SecurityLevel2::Current::authenticate()` オペレーションを使用して証明書ベースの認証の使用を指定することでブートストラップ処理プロセスの安全を確保できます。
`corbalocs`	IIOP リスナ / ハンドラの呼び出しが保護されており、IIOP リスナ / ハンドラまたはサーバ ORB は SSL を使用できるようにコンフィギュレーションする必要があります。

これらの URL アドレス形式は、インターオペラブル・ネーミング・サービスの一部として OMG によって採用されたオブジェクト URL の定義のサブセットです。BEA Tuxedo ソフトウェアでは、BEA WebLogic Enterprise 4.2 で追加されたランダマイズ機能だけでなく、セキュア HTTP の URL に基づいてモデル化される安全な形式をサポートするようにも、OMG 提案のインターオペラブル・ネーミング・サービスで規定された URL 形式が拡張されます。

`corbaloc` および `corbalocs` URL スキーマは、TCP/IP 中心の環境と DNS 中心の環境の両方で簡単に操作される位置を提供します。これらの URL スキーマでは、DNS 形式の *`hostname`* または IP アドレスと *`port_number`* を指定します。次に例を示します。

```
corbaloc://curly:1024,larry:1022,joe:1999
corbalocs://host1:1024,{host2:1022|host3:1999}
```

BEA WebLogic Enterprise バージョン 5.1 ソフトウェアでは、スキーマが異なる複数の URL のリストをサポートするように、OMG 提案のインターオペラブル・ネーミング・サービスで規定された URL 構文が拡張されています。次に例を示します。

```
corbalocs://curly:1024,corbaloc://larry:1111,
corbalocs://ctxobj:3434,mthd:3434,corbaloc://force:1111
```

上の例で、パーサが URL corbaloc://force:1111 に達した場合は、安全な接続が試行されなかったかのようにパーサの内部状態がリセットされ、保護なしの接続の試行が開始されます。

注記　ヌル・スキーマ URL アドレス (//hostname:port_number) と corbaloc および corbalocs URL アドレスは一緒に使用しないでください。

注記　Netscape の埋め込み型 Java ORB および JavaSoft JDK ORB で使用するように提供される Bootstrap オブジェクトでは、corbaloc および corbalocs URL はサポートされていません。

注記　corbaloc および corbalocs URL アドレス形式の使い方の詳細については、『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。

注記　Bootstrap コンストラクタまたは TOBJADDR で指定するネットワーク・アドレスは、サーバ・アプリケーションの UBBCONFIG ファイルのネットワーク・アドレスと (大文字と小文字の区別を含めて) 正確に同じでなければなりません。アドレスが一致しない場合、Bootstrap コンストラクタの呼び出しは失敗し、見たところ関係のなさそうな次のエラー・メッセージが表示されます。

```
ERROR: Unofficial connection from client at
<tcp/ip address>/<port-number>
```

たとえば、ネットワーク・アドレスがサーバ・アプリケーションの UBBCONFIG ファイルの ISL コマンド行オプション文字列で (ヌル URL アドレス形式を使用して) //TRIXIE:3500 として指定されている場合、Bootstrap コンストラクタまたは TOBJADDR で //192.12.4.6:3500 または //trixie:3500 を指定すると接続の試行は失敗します。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。Windows 2000 システムで正確な大文字と小文字の区別を調べるには、コントロール・パネルでホスト・システムのネットワーク設定を確認します。

注記　前の注記のエラーは、URL アドレス形式が使用される場合は遅延されます。つまり、ISL/ISH への接続が遅延させるので Bootstrap オブジェクトの作成時にそのエラーは発生しません。

- ネイティブ・クライアント

  ネイティブ・クライアントの場合、Tobj_Bootstrap コンストラクタの address パラメータは常に (ヌル・ポインタではなく) 空の文字列でなければなりません。ネイティブ・クライアントは、TUXCONFIG 環境変数で指定されたアプリケーションに接続します。アドレスが空でない場合は、コンストラクタによって CORBA::BAD_PARAM が生成されます。

- クライアントとして機能するサーバ

  Bootstrap オブジェクトにアクセスする必要がある場合、サーバでは TP.bootstrap() を呼び出して TP フレームワークを使用してそのオブジェクト・リファレンスを取得しなければなりません。Bootstrap オブジェクトの新しいインスタンスを作成することは避ける必要があります。

applet (Java メソッドのみに適用)
: これは、クライアント・アプレットへのポインタです。クライアント・アプレットから Bootstrap オブジェクトに明示的に ISH ホストとポートが渡されない場合は、この引数を渡すことができます。この引数を渡すと、Bootstrap オブジェクトはアプレットの HTML ファイルで TOBJADDR の定義を検索します。

例外 BAD_PARAM
: オブジェクトがニルである場合、あるいはオブジェクトに格納されているホストが接続と一致しないか、ホスト・アドレス (//hostname:port_number) が有効な形式ではない場合に発生します。

説明 Bootstrap オブジェクトを作成する C++ メンバ関数 (または Java メソッド)。

戻り値 新しく作成された Bootstrap オブジェクトへのポインタ。

Tobj_Bootstrap::register_callback_port

概要 IIOP ハンドラ (ISH) で共同クライアント / サーバの受信ポートを登録します。

C++ マッピング

```
void register_callback_port(CORBA::Object_ptr objref);
```

パラメータ

`objref`
: 共同クライアント / サーバによって作成されたオブジェクト・リファレンス。

例外

`BAD_PARAM`
: オブジェクトがニルである場合、またはオブジェクトに格納されているホストが接続と一致しない場合に発生します。

`IMP_LIMIT`
: `register_callback_port` メソッドが複数回呼び出された場合に発生します。

説明 この C++ メンバ関数 (または Java メソッド) は、共同クライアント / サーバの受信ポートを ISH に通知するために呼び出します。このメソッドは、GIOP 1.2 の双方向機能をサポートしていない共同クライアント / サーバ ORB (つまり GIOP 1.0 および 1.1 のクライアント ORB) でのみ使用します。GIOP 1.0 および 1.1 の場合、ISH では共同クライアント / サーバごとに 1 つの受信ポートのみをサポートします。このため、`register_callback_port` メソッドは接続されている共同クライアント / サーバごとに 1 度だけ呼び出すようにします。

使用上の注意 このメソッドを使用するときには、以下の情報を考慮に入れてください。

- `register_callback_port` メソッドが共同クライアント / サーバから呼び出されない場合、コールバック・ポートは ISH に登録されず、デフォルトで非対称アウトバウンド IIOP が使用されます。この場合は、`-o` オプションを使用してサーバの IIOP リスナ (ISL) を起動する必要があります。`-o` オプションは、非対称アウトバウンド IIOP を有効にします。非対称アウトバウンド IIOP を有効にしないと、サーバからクライアントの呼び出しが ISL/ISH によって許可されません。
- POA の代わりに BEAWrapper Callbacks API を使用し、かつ双方向の振る舞いを使用する必要がある場合は、GIOP 1.2 をサポートする ISH を使

用する場合でも常に register_callback_port メソッドを呼び出す必要があります。

- コールバック・オブジェクトの双方向機能を使用する必要がある場合は、register_callback_port メソッドを呼び出してからコールバック・オブジェクトのオブジェクト・リファレンスをサーバに渡す必要があります。

戻り値　特にありません。

Tobj_Bootstrap::resolve_initial_references

概要 CORBA オブジェクト・リファレンスを取得します。

C++ マッピング

```
CORBA::Object_ptr resolve_initial_references(
                const char* id);
          throws Tobj::InvalidName,
                 org.omg.CORBA.SystemException;
```

パラメータ `id`

このパラメータの値は次のいずれかです。

```
“FactoryFinder”
“InterfaceRepository”
“NameService”
“NotificationService”
“SecurityCurrent”
“TransactionCurrent”
“Tobj_SimpleEventsService”
```

例外 `InvalidName`

`id` が上記のどの名前でもない場合に発生します。サーバでは、SecurityCurrent が渡された場合にも `resolve_initial_references` で `Tobj::InvalidName` が生成されます。

`CORBA::NO_PERMISSION`

`id` が TransactionCurrent または SecurityCurrent で、クライアントの別の Bootstrap オブジェクトが Current オブジェクトを所有している場合に発生します。

説明 この C++ メンバ関数 (または Java メソッド) は、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 InterfaceRepository オブジェクトの CORBA オブジェクト・リファレンスを取得します。特定のオブジェクト・リファレンスについては、`_narrow` 関数を呼び出します。たとえば、FactoryFinder の場合は `Tobj::FactoryFinder::_narrow` を呼び出します。

戻り値 表 4-2 は、各 `id` で返されるオブジェクト・リファレンスを示しています。

Tobj_Bootstrap::destroy_current()

概要 Bootstrap オブジェクトで表されるドメインの Current オブジェクトを破棄します。

C++ マッピング

```
void destroy_current();
```

例外 Bootstrap オブジェクトが Current オブジェクトの所有者でない場合に CORBA::NO_PERMISSION が生成されます。

説明 この C++ メンバ関数は、Bootstrap オブジェクトで表されるドメインの Current オブジェクトを無効にします。destroy_current() メソッドを呼び出した後、Current オブジェクトは無効になります。それ以降に古い Current オブジェクトを使用しようとすると、CORBA::BAD_INV_ORDER 例外がスローされます。プログラミングでは、すべての Current オブジェクトを解放してから destroy_current() を呼び出すようにしてください。

注記 destroy_current() メソッドは、2 つの Current オブジェクト (Transaction と Security) を所有しているドメインの Bootstrap オブジェクトで呼び出す必要があります。このメソッドを呼び出すと、セキュリティの logoff が暗黙的に呼び出され、クライアントによって開始されたトランザクションがすべて暗黙的にロールバックされます。

アプリケーションでは、まず destroy_current() を呼び出してから別のドメインの TransactionCurrent または SecurityCurrent の resolve_initial_references を呼び出す必要があります。そうしないと、resolve_initial_references で CORBA::NO_PERMISSION が生成されます。

戻り値 特にありません。

Java のメソッド

Java BEA ブートストラップ処理 API では、次のメソッドがサポートされています。

- Tobj_Bootstrap

- Tobj_Bootstrap.register_callback_port
- Tobj_Bootstrap.resolve_initial_references
- Tobj_Bootstrap.destroy_current
- Tobj_Bootstrap.GetTransactions
- Tobj_Bootstrap.getUserTransaction
- Tobj_Bootstrap.getNativeProperties
- Tobj_Bootstrap.getRemoteProperties

これらのメソッドの詳細については、Javadoc API を参照してください。

オートメーションのメソッド

この節では、BEA ブートストラップ処理メカニズムでサポートされているオートメーションのメソッドについて説明します。

Initialize

概要 Bootstrap オブジェクトを BEA Tuxedo ドメインに初期化します。

MIDL マッピング

```
HRESULT Initialize(
              [in] BSTR host);
```

オートメーション・マッピング

```
Sub Initialize(address As String)
```

パラメータ

`address`

BEA Tuxedo ドメインの IIOP リスナ / ハンドラのホスト名とポート。1 つまたは複数の TCP/IP アドレスを指定できます。複数のアドレスは、C++ のマッピングと同じようにカンマ区切りのリストで指定します。アドレスを指定しない場合は、`TOBJADDR` 環境変数の値が使用されます。

注記 Bootstrap コンストラクタまたは `TOBJADDR` で指定するネットワーク・アドレスは、アプリケーションの `UBBCONFIG` ファイルのネットワーク・アドレスと (形式および大文字と小文字の区別を含めて) 正確に同じでなければなりません。アドレスが一致しない場合、Bootstrap コンストラクタの呼び出しは失敗し、見たところ関係のなさそうな次のエラー・メッセージが表示されます。

```
ERROR: Unofficial connection from client at
<tcp/ip address>/<port-number>
```

たとえば、ネットワーク・アドレスが ISL コマンド行オプション文字列で `//TRIXIE:3500` として指定された場合、Bootstrap コンストラクタまたは `TOBJADDR` で `//192.12.4.6:3500` または `//trixie:3500` を指定すると接続の試行が失敗します。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。Windows システムで正確な大文字と小文字の区別を調べるには、コントロール・パネルでホスト・システムのネットワーク設定を確認します。

戻り値 特にありません。

例外　表 4-5 は例外を示しています。

表 4-5 Initialize の例外

HRESULT	説明	意味
ITF_E_NO_PERMISSION_YES	Bootstrap が既に初期化されている	Bootstrap オブジェクトが既に初期化されています。新しい BEA Tuxedo ドメインに接続するには、新しい Bootstrap オブジェクトを作成する必要があります。
E_INVALIDARG	無効なアドレス・パラメータ	指定されたアドレスが有効ではありません。
E_OUTOFMEMOY	メモリの割り当てが異常終了した	必要なメモリを割り当てることができませんでした。
E_FAIL	無効なドメイン	指定されたアドレスで BEA Tuxedo ドメインと通信できないか、TOBJADDR が定義されていません。
<SYSTEM ERROR>	初期オブジェクトが取得不能	Bootstrap オブジェクトを初期化できません。障害の原因であるシステム・エラーは、エラー・オブジェクトの「Number」メンバで返されます。

CreateObject

概要 Current 環境オブジェクトのインスタンスを作成します。

MIDL マッピング

```
HRESULT CreateObject(
              [in] BSTR progid,
      [out, retval] IDispatch** rtrn);
```

オートメーション・マッピング

```
Function CreateObject(progid As String) As Object
```

パラメータ

`progid`

作成する環境オブジェクトの `progid`。有効な `progid` は次のとおりです。

```
Tobj.FactoryFinder
Tobj.SecurityCurrent
Tobj.TransactionCurrent
```

戻り値 作成された環境オブジェクトのインターフェイス・ポインタの参照。

例外 表 4-6 は例外を示しています。

表 4-6 CreateObject の例外

例外	説明	意味
`ITF_E_NO_PERMISSION_YES`	Bootstrap オブジェクトの初期化が必要	Bootstrap オブジェクトが初期化されていません。
`ITF_E_NO_PERMISSION_NO`	パーミッションなし	`progid` で TransactionCurrent または SecurityCurrent が指定され、クライアントの別の Bootstrap オブジェクトで Current オブジェクトが所有されている場合。
`E_INVALIDARG`	無効な `progid` パラメータ	指定された `progid` が有効ではありません。
`E_INVALIDARG`	無効な名前	要求された `progid` が上記の有効なパラメータ値ではありません。

表 4-6 CreateObject **の例外 (続き)**

例外	説明	意味
E_INVALIDARG	未知のオブジェクト	要求された progid がシステムに登録されていません。
<SYSTEM ERROR>	CoCreateInstance() が失敗した	Bootstrap オブジェクトが要求されたオブジェクトのインスタンスを作成できませんでした。システム・エラーは、エラー・オブジェクトの「Number」メンバで返されます。

DestroyCurrent

概要 BEA Tuxedo ドメインからログアウトし、TransactionCurrent オブジェクトと SecurityCurrent オブジェクトを無効にします。

MIDL マッピング

```
HRESULT destroy_current();
```

オートメーション・マッピング

```
Sub destroy_current()
```

パラメータ 特にありません。

戻り値 特にありません。

例外 特にありません。

Bootstrap オブジェクトのプログラミング例

この節では、Bootstrap オブジェクトを使用する次のプログラミング例を紹介します。

- Java クライアントの例 : SecurityCurrent オブジェクトの取得
- Visual Basic クライアントの例 : Bootstrap オブジェクトの使用

Java クライアントの例 : SecurityCurrent オブジェクトの取得

リスト 4-4 は、SecurityCurrent オブジェクトを取得する Java クライアントのプログラミング方法を示しています。

コード リスト 4-4　SecurityCurrent **オブジェクトを取得する** Java **クライアントのプログラミング**

```
import java.util.*;
import org.omg.CORBA.*;
import com.beasys.*;
class client {
      public static void main(String[] args)
      {
               Properties prop = null;
               Tobj.PrincipalAuthenticator auth = null;
               String host_port = "//COLORMAGIC:10000";
               // ホストとポートを設定
               if (args.length == 1) host_port = args[0];
               try {
                    // ORB を初期化
                    ORB orb = ORB.init(args, prop);
                    // Bootstrap オブジェクトを作成
                    Tobj_Bootstrap bs=new  Tobj_Bootstrap(orb,host_port);

                    // SecurityCurrent を取得
                    org.omg.CORBA.Object ocur =
                          bs.resolve_initial_references("SecurityCurrent");
                    SecurityLevel2.Current cur =
                          SecurityLevel2.CurrentHelper.narrow(ocur);
               }
               catch (Tobj.InvalidName e) {
                  System.out.println("Invalid name: "+e);
                  System.exit(1);
               }
               catch (Tobj.InvalidDomain e) {
                  System.out.println("Invalid domain address: "+host_port +" "+e);
                  System.exit(1);
               }
               catch (SystemException e) {
                  System.out.println("Exception getting security current: "+e);
                  e.printStackTrace();
                  System.exit(1);
               }
        }
}
```

Visual Basic クライアントの例 : Bootstrap オブジェクトの使用

リスト 4-5 は、Bootstrap オブジェクトを使用する Visual Basic クライアントのプログラミング方法を示しています。

コード リスト 4-5 Visual Basic **クライアントのプログラミング**

```
'Bootstrap オブジェクトを宣言
Public oBootstrap As DITobj_Bootstrap

'FactoryFinder オブジェクトを宣言
Public oBsFactoryFinder As DITobj_FactoryFinder

'Registrar オブジェクトのファクトリを宣言
Public oRegistrarFactory As DIUniversityB_RegistrarFactory

' 実際の Registrar オブジェクトを宣言
Public oRegistrarFactory As DIUniversityB_RegistrarFactory

....

'Bootstrap オブジェクトを作成

Set oBootstrap = CreateObject(“Tobj.Bootstrap”)

'BEA Tuxedo ドメインに接続
oBootstrap.Initialize  “//host:port”

'BEA Tuxedo ドメインの FactoryFinder を取得
Set oBSFactoryFinder =
oBootstrap.CreateObject(“Tobj.FactoryFinder”)

'Registrar オブジェクトのファクトリを取得
‘using the FactoryFinder method find_one_factory_by_id
Set oRegistrarFactory =
oBSFactoryFinder.find_one_factory_by_id(“RegistrarFactoryID”)

'Registrar オブジェクトを作成
Set oRegistrar = oRegistrarFactory.find_registrar(exc)
```

インターオペラブル・ネーミング・サービス・ブートストラップ処理メカニズム

ここでは、次の内容について説明します。

- はじめに
- INS オブジェクト・リファレンス
- INS コマンド行オプション
- INS オブジェクトの URL スキーマ
- INS を使用した FactoryFinder オブジェクト・リファレンスの取得
- INS を使用した PrincipalAuthenticator オブジェクト・リファレンスの取得
- INS を使用した TransactionFactory オブジェクト・リファレンスの取得

はじめに

リリース 8.0 以降の BEA Tuxedo ORB では、CORBA 仕様 2.4.2 の第 4 章と第 13 章で規定されている CORBA ネーミング・サービス (このマニュアルではインターオペラブル・ネーミング・サービスと呼ぶ) ブートストラップ処理メカニズムがサポートされています。

このサポートにより、インターオペラブル・ネーミング・サービス (INS) ブートストラップ処理メカニズムをインプリメントする ORB で、BEA Tuxedo サーバ側 ORB に問い合わせて初期オブジェクト (FactoryFinder など) および BEA Tuxedo 環境に対する PrincipalAuthenticator のオブジェクト・リファレンスを取得できます。このサポートと、インターオペラブル初期オブジェクト・リファレンスのクライアント・サポートにより、クライアントでは BEA ブートストラップ処理メカニズムではなく INS ブートストラップ処理メカニズムを使用できます。

注記　BEA Tuxedo ソフトウェアに付属の CORBA C++ クライアントおよび Java クライアントでは INS ブートストラップ処理メカニズムを使用できますが、性能上の理由により推奨はできません。

INS オブジェクト・リファレンス

表 4-7 は、各 ID で返されるオブジェクト・リファレンスを示しています。

表 4-7 返されるオブジェクト・リファレンス

ID	返されるオブジェクト・リファレンス
FactoryFinder	FactoryFinder オブジェクト (`CORBA::FactoryFinder`)
InterfaceRepository	InterfaceRepository オブジェクト (`CORBA::Repository`)
NameService	CORBA ネーミング・サービス・オブジェクト (`CORBA::NameService`)
NotificationService	EventChannelFactory オブジェクト (`CosNotifyChannelAdmin::EventChannelFactory`)
POACurrent	POACurrent オブジェクト (`CORBA::POACurrent`)
PrincipalAuthenticator	PrincipalAuthenticator オブジェクト (`SecurityLevel2::`PrincipalAuthenticator)
RootPOA	RootPOA オブジェクト (`CORBA::RootPOA`)
Tobj_SimpleEventsService	BEA シンプル・イベント ChannelFactory オブジェクト (`Tobj_SimpleEvents::ChannelFactory`)

INS コマンド行オプション

リリース 8.0 以降の BEA Tuxedo CORBA では、-ORBInitRef および -ORBDefaultInitRef コマンド行オプションがサポートされています。これらのオプションの詳細については、第 14 章の 85 ページ「ORB 初期化メンバ関数」を参照してください。

次の例は、BEA Tuxedo CORBA IIOP クライアントが BEA Tuxedo CORBA IIOP サーバ環境と通信しているものと想定しています。

```
client_app –ORBid BEA_IIOP –ORBInitRef
           FactoryFinder=corbaloc::myhost:2468/FactoryFinder
```

この例で、FactoryFinder の ORB::resolve_initial_references を呼び出すと、インターオペラブル初期リファレンスの要求がポート 2468 を使用して myhost の ISL/ISH に送信されます。myhost は、tuxconfig ファイルで ISL/ISH に指定されたホストと大文字と小文字の区別まで正確に同じでなければなりません。

INS 初期化オペレーション

INS ブートストラップ処理メカニズムを使用するには、アプリケーション・プログラマは以下の要件を満たす必要があります。

- INS 初期リファレンス・メカニズムを使用する必要のある BEA Tuxedo CORBA IIOP クライアントでは、Tobj_Bootstrap::resolve_initial_references 関数ではなく ORB::resolve_initial_references 関数を呼び出す必要があります。ORB::resolve_initial_references の構文の説明については、第 14 章の 79 ページ「CORBA::ORB::resolve_initial_references」を参照してください。

  注記　Tobj_Bootstrap API は依然としてサポートされており、その振る舞いは従来のまま変更されていません。

- INS 初期リファレンス・メカニズムを使用する BEA Tuxedo CORBA IIOP クライアントでは、Tobj_Bootstrap::list_initial_services 関数

ではなく ORB::list_initial_services 関数を使用する必要があります。ORB::list_initial_services の構文の説明については、第 14 章の 75 ページ「CORBA::ORB::list_initial_services」を参照してください。

INS オブジェクトの URL スキーマ

リリース 8.0 以降の BEA Tuxedo CORBA では、BEA Tuxedo CORBA サーバ環境にアクセスし、初期オブジェクトのリファレンスを取得するための位置を指定するために使用する追加の Uniform Resource Locator (URL) 形式がサポートされています。その新しい URL 形式は、INS 仕様の一部として OMG で採用されたオブジェクト URL の定義に従うとともに、それを拡張します。INS 仕様で規定されている URL 形式は、BEA WebLogic Enterprise バージョン 5.1 で導入されたランダマイズ機能だけでなく、セキュア HTTP の URL に基づいてモデル化される安全な形式をサポートするようにも拡張されています。

CORBA 2.4.2 仕様は、仕様に準拠する ORB が 3 つのオブジェクト URL スキーマをサポートすることを要求します。それらのスキーマは、IOR、corbaloc、および corbaname として定義されています。

注記　新しい URL 文字列形式は、ORB::string_to_object 関数にも渡すことができます。

IOR URL スキーマ

IOR スキーマは、IOR:*hex_octets* という形式の文字列です。スキーマ名は IOR で、「:」の後のテキストは CORBA 仕様で定義されます。IOR URL スキーマは堅牢であり、オブジェクトのリファレンスに使用されるカプセル化された転送情報とオブジェクト・キーからクライアントを切り離します。

corbaloc URL スキーマ

その長さとバイナリ情報のテキスト符号化が原因で、IOR を非電子的な手段で人間同士が交換するのは困難です。corbaloc および corbalocs URL スキーマでは、一般に普及している FTP や HTTP の URL スキーマに似た形式の文

字列化されたオブジェクト・リファレンスが提供されます。corbaloc および corbalocs で定義された URL スキーマは、TCP/IP 中心の環境と DNS 中心の環境の両方で簡単に操作されます。corbaloc および corbalocs URL の構成要素は次のとおりです。

- DNS 形式のホスト名または IP アドレスとポート
- 使用する IIOP プロトコルのバージョン (オプション)
- オブジェクト・キー (オプション)

デフォルトでは、corbaloc URL は IIOP 経由でアクセスできるオブジェクトを示し、corbalocs URL は IIOP over SSL を使用してアクセスできるオブジェクトを示します。

表 4-8 は、各 URL 要素の BNF 構文を示しています。

表 4-8 URL 要素の BNF 形式

URL 要素	BNF 形式
<corbaloc>	= “corbaloc::”<obj_addr_list>[“/”<key_string>] [,<corbaloc>|<corbalocs>]
<corbalocs>	= “corbalocs::”<obj_addr_list>[“/”<key_string>] [,<corbaloc>|<corbalocs>]
<obj_addr_list>	= [<obj_addr> “,”]* <obj_addr>
<obj_addr>	= <iiop_prot_addr> | <future_prot_addr>
<iiop_prot_addr	= <iiop_id><iiop_addr>
<iiop_id>	= “//” | <iiop_prot_token>”:”
<iiop_prot_token>	= “iiop”
<iiop_addr>	= [<version> <host> [“:” <port>]]
<host>	= DNS-style Host Name | ip_address
<version>	= <major> “.” <minor> “@” | empty_string
<port>	= number
<major>	= number

表 4-8 URL 要素の BNF 形式 (続き)

URL 要素	BNF 形式
<minor>	= number
<key_string>	= <string> | empty_string

表 4-9 は、各 URL 要素を説明しています。

表 4-9 URL 要素の説明

URL 要素	説明
obj_addr_list	プロトコル ID、バージョン、およびアドレス情報のカンマ区切りのリスト。このリストは、インプリメンテーション定義の方法でオブジェクトのアドレスを指定するために使用します。オブジェクトは、アドレスおよびプロトコルのいずれでもアクセスできます。その要素を使用して障害が発生した場合は、カンマ区切りリストの次の要素が使用されます。
obj_addr	プロトコル ID、バージョン・タグ、およびプロトコル固有のアドレス。右中かっこ「{」、左中かっこ「}」、垂直バー「|」、スラッシュ「/」、およびカンマ「,」は URL のこの構成要素では使用できません。
iiop_prot_addr	IIOP プロトコル ID、バージョン・タグ、および DNS 形式のホスト名と IP アドレスで構成されたアドレス。
iiop_id	IIOP プロトコル corbaloc を示すために認識されるトークン。
iiop_prot_token	IIOP プロトコル・トークン「iiop」。
iiop_addr	単一のアドレス要素。
host	DNS 形式のホスト名または IP アドレス。指定しない場合は、ローカル・ホストと見なされます。
version	「.」で区切られ、「@」が後に続くメジャーおよびマイナーのバージョン番号。バージョンがない場合は、1.0 と見なされます。
ip_address	数値の IP アドレス (ドット区切りの 10 進表記)。
port	IIOP リスナ / ハンドラまたは初期化エージェントがリッスンするポート番号。デフォルト値は 9999 です。

表 4-9 URL 要素の説明

URL 要素	説明
key_string	ヌルで終了しない文字列化されたオブジェクト・キー。key_string では、RFC 2396 で規定されているエスケープ規則を使用して URL の一部として直接使用できないオクテット値からマッピングします。US-ASCII 英数字はエスケープされません。この範囲外の文字は、次の文字を除いてエスケープされます。 “;” | “/” | “:” | “?” | “@” | “&” | “=” | “+” | “$” | ”,” | “-“ | “_” | “.” | “!” | “~” | “*” | “” | “(“ | “)” key_string は、CORBA 仕様で定義されている GIOP 要求または LocateRequest ヘッダの object_key メンバのオクテット・シーケンスに対応します。
string_name	IETF (Internet Engineering Task Force) RFC 2396 で定義されている URL エスケープが含まれる文字列化された名前。エスケープ規則は、変更の必要なく URL をさまざまな方法で転送できるようにします。US-ASCII 英数字はエスケープされません。この範囲外の文字は、次の文字を除いてエスケープされます。 “;” | “/” | “:” | “?” | “@” | “&” | “=” | “+” | “$” | ”,” | “-“ | “_” | “.” | “!” | “~” | “*” | “” | “(“ | “)”

次に、新しい URL 形式の使用例を示します。

```
corbaloc::555xyz.com:1024,555backup.com:1022,555last.com:1999
corbalocs::555xyz.com:1024,{555backup.com:1022|555last.com:1999}
corbaloc::1.2@555xyz.com:1111
corbalocs::1.1@24.128.122.32:1011,1.0@24.128.122.34
```

BEA Tuxedo 8.0 では、スキーマの異なる複数の URL のリストをサポートするように、INS の提案で規定されている URL 構文が拡張されています。次に、拡張例を示します。

```
corbalocs::555xyz.com:1024,corbaloc::1.2@555xyz.com:1111
corbalocs::ctxobj:3434,mthd:3434,corbaloc::force:1111
```

上の例で、パーサが URL corbaloc::force.com:1111 に達した場合は、安全な接続が試行されなかったかのようにパーサの内部状態がリセットされ、保護なしの接続の試行が開始されます。

corbaname URL スキーマ

corbaname URL スキーマは、URL でネーミング・サービスのエントリを示せるように corbaloc スキーマの機能を拡張します。corbaname URL は、ORB コアにネーミング・サービスのインプリメンテーションがなくても解決できます。corbaname URL の例を次に示します。

```
corbaname:555objs.com#a/string/path/to/obj
```

この URL は、ホスト `555objs.com` で NamingContext 型のオブジェクト (オブジェクト・キーが *NamingService*) を見つけることができるか、その位置で動作しているエージェントが NamingContext のリファレンスを返すことを示します。文字列化された名前 `a/string/path/to/obj` は、その NamingContext の `resolve` オペレーションの引数として使用されます。

corbaname URL は corbaloc URL と似ていますが、corbaname URL にはネーミング・コンテキストのバインディングを識別する文字列化された名前が含まれます。# 文字は、文字列化された名前の開始を示します。

URL の BNF 構文は、表 4-10 で示してあります。

表 4-10 URL の BNF 構文

URL 要素	書式	説明
`<corbaname>`	`= “corbaname:”<corbaloc_obj>[ “#”<string_name>]`	`corbaloc_obj` は、corbaname URL のネーミング・コンテキストを識別する部分です。構文は、corbaloc URL の場合と同じです。
`<corbaloc_obj>`	`<obj_addr_list>[“/”<key_string>]`	`obj_addr_list` の説明については、表 4-9 を参照してください。
`<obj_addr_list>`	corbaloc URL の定義と同じ	`obj_addr_list` の説明については、表 4-9 を参照してください。
`<key_string>`	corbaloc URL の定義と同じ	`key_string` の説明については、表 4-9 を参照してください。
`<string_name>`	`Stringified Name | empty string`	`string_name` の説明については、表 4-9 を参照してください。

corbaname URL の解決は、corbaloc URL 処理の単純な拡張としてインプリメントされます。そのインプリメンテーションを説明するために、次の corbaname URL を使用します。

```
corbaname:<corbaloc_obj>[“#”<string_name>]
```

解決は以下のように行われます。

1. corbaname URL から `corbaloc::<corbaloc_obj>` 形式の corbaloc URL を作成します。
2. `CORBA::ORB::string_to_object` を呼び出して `CosNaming::NamingContext` オブジェクトを取得し、corbaloc URL をネーミング・コンテキストのオブジェクト・リファレンスに変換します。
3. `<string_name>` を `CosNaming::Name` に変換します。
4. 作成した `CosNaming::Name` を渡して、`CosNaming::NamingContext` の resolve オペレーションを呼び出します。
5. `CosNaming::NamingContext::resolve` から返されるオブジェクト・リファレンスは、呼び出し側に返されなければなりません。

この解決プロセスに従うことで、ネーミング・サービスに存在しないネーミング・コンテキストのオブジェクト・リファレンスを返すことがなくなります。この手法の 1 つの副作用は、ネーミング・サービスのスタブが ORB コアの一部であるか、`resolve` オペレーションの要求を送信する内部メカニズムがなければならないことです。複雑な手間を避けるため、ネーミング・サービスのスタブを ORB コアに埋め込むことをお勧めします。

INS を使用した FactoryFinder オブジェクト・リファレンスの取得

リスト 4-6 は、クライアント・アプリケーションが INS を使用して FactoryFinder オブジェクトのオブジェクト・リファレンスを取得するしくみを示しています。完全なコード例については、University Sample のクライアント・アプリケーションを参照してください。

コード リスト 4-6 FactoryFinder **オブジェクト取得のコード例**

```
// レジストラを取得するユーティリティ
static UniversityW::Registrar_ptr get_registrar(
    CORBA::ORB_ptr  orb
)
{

    // ORB からファクトリ・ファインダを取得
    CORBA::Object_var v_fact_finder_oref =
        orb->resolve_initial_references("FactoryFinder");

    // ファクトリ・ファインダをナロー変換
    Tobj::FactoryFinder_var v_fact_finder_ref =
        Tobj::FactoryFinder::_narrow(v_fact_finder_oref.in());

    // ファクトリ・ファインダを使用して University の
    // レジストラ・ファクトリを検索する
    CORBA::Object_var v_reg_fact_oref =
        v_fact_finder_ref->find_one_factory_by_id(
            UniversityW::_tc_RegistrarFactory->id()
        );

    // レジストラ・ファクトリをナロー変換
    UniversityW::RegistrarFactory_var v_reg_fact_ref =
        UniversityW::RegistrarFactory::_narrow(
            v_reg_fact_oref.in()
        );

    // University のレジストラを返す
    return v_reg_fact_ref->find_registrar();
}
```

INS を使用した PrincipalAuthenticator オブジェクト・リファレンスの取得

リスト 4-7 は、クライアント・アプリケーションが INS を使用して PrincipalAuthenticator オブジェクトのオブジェクト・リファレンスを取得するしくみを示しています。完全なコード例については、University Sample のクライアント・アプリケーションを参照してください。

コード リスト 4-7　PrincipalAuthenticator **オブジェクト取得のコード例**

```
// セキュリティ・システムにログオンするユーティリティ
static SecurityLevel2::PrincipalAuthenticator_ptr logon(
    CORBA::ORB_ptr          orb,
    const char*             program_name,
    UniversityW::StudentId  stu_id
)
{

    // ORB から直接 Principal Authenticator を取得
    CORBA::Object_var v_pa_obj =
        orb->resolve_initial_references("PrincipalAuthenticator");

    // Principal Authenticator をナロー変換
    SecurityLevel2::PrincipalAuthenticator_var v_pa =
        SecurityLevel2::PrincipalAuthenticator::_narrow(
                                          v_pa_obj.in());
```

INS を使用した TransactionFactory オブジェクト・リファレンスの取得

リリース 8.0 以降の BEA Tuxedo CORBA では、CORBA トランザクション・サービス・インターフェイスを使用してトランザクションを開始できます。クライアントは、`ORB::resolve_initial_references("FactoryFinder")` 関数を使用して FactoryFinder のオブジェクト・リファレンスを取得します。次にクライアントは、FactoryFinder を使用して (トランザクションの開始に使用する) TransactionFactory のリファレンスを取得します。

リスト 4-8 は、クライアント・アプリケーションが INS を使用して TransactionFactory オブジェクトのオブジェクト・リファレンスを取得するしくみを示しています。完全なコード例については、University Sample のクライアント・アプリケーションを参照してください。

コード リスト 4-8　INS **を使用したクライアント・アプリケーションのコー**

ド例

```
// ORB からファクトリ・ファインダを取得
CORBA::Object_var v_fact_finder_oref =
   orb->resolve_initial_references("FactoryFinder");

// ファクトリ・ファインダをナロー変換
Tobj::FactoryFinder_var v_fact_finder_ref =
   Tobj::FactoryFinder::_narrow(v_fact_finder_oref.in());

// FactoryFinder から TransactionFactory を取得
CORBA::Object_var v_txn_fac_oref =
   v_fact_finder_ref->find_one_factory_by_id(
            "IDL:omg.org/CosTransactions/TransactionFactory:1.0");

// TransactionFactory オブジェクト・リファレンスをナロー変換
CosTransactions::TransactionFactory_var v_txn_fac_ref =
   CosTransactions::TransactionFactory::_narrow(
                                        v_txn_fac_oref.in());
```

INS ブートストラップ処理メカニズムを使用したイベントのシーケンスは次のとおりです。

1. `ORB::resolve_initial_references` を使用して FactoryFinder を取得します。
2. FactoryFinder を使用して TransactionFactory を取得します。
3. TransactionFactory の create オペレーションを使用してトランザクションを開始します。
4. create オペレーションで返される Control オブジェクトから、`get_terminator` メソッドを使用してトランザクションの Terminator インターフェイスを取得します。
5. Terminator の commit または rollback オペレーションを使用してトランザクションを終了またはアボートします。

TransactionFactory は、BEA の委譲型インターフェイスではなく、標準の CORBA のトランザクション・サービス・インターフェイスに準拠したオブジェクトを返します。これは、サード・パーティの ORB が、それぞれの ORB の `resolve_initial_references` 関数を使用して BEA Tuxedo CORBA

サーバから TransactionFactory のリファレンスを取得し、標準 OMG IDL で生成されたスタブを使用して、返されたインスタンス上で機能できることを意味します。

制限

BEA Tuxedo 8.0 リリースでは、TransactionFactory とクライアントの Current のアクションが調整されません。つまり、クライアントでは 2 つのメカニズムのいずれかを使用してトランザクションのステータスを管理および取得する必要があります。また、インターフェイスとオペレーションは表 4-11 で示されているものしかサポートされていません。OMG IDL で規定されているほかのオペレーションでは、`CORBA::NO_IMPLEMENT` 例外が返されます。

表 4-11 サポートされている INS インターフェイスとオペレーション

インターフェイス	サポートされているオペレーション
TransactionFactory	create
Control	get_terminator get_coordinator
Terminator	commit rollback
Coordinator	get_status rollback_only get_transaction_name

5 FactoryFinder インターフェイス

FactoryFinder インターフェイスは、BEA Tuxedo ドメインへの唯一のエントリ・ポイントとして作用する 1 つのオブジェクト・リファレンスをクライアントに提供します。BEA Tuxedo NameManager により、FactoryFinder のオブジェクト・リファレンスに対するファクトリ名のマッピングが提供されます。複数の FactoryFinder と NameManager が一緒になると、可用性と信頼性が向上します。このリリースでは、機能性のレベルを拡張して、複数ドメインをサポートしています。

注記　NameManager は CORBA サービス・ネーミング・サービスのようなネーミング・サービスではなく、登録されたファクトリを格納するための単なる手段です。

BEA Tuxedo 環境では、アプリケーション・ファクトリ・オブジェクトを使用して、クライアントがビジネス・オペレーション (TellerFactory、Teller など) の実行のためにやり取りするオブジェクトを作成します。一般に、アプリケーション・ファクトリはサーバの初期化中に作成され、リモート・クライアントと、サーバ・アプリケーション内に置かれたクライアントの双方からアクセスされます。

FactoryFinder インターフェイスと NameManager サービスは、アプリケーション・サーバではない個別のサーバに格納されています。クライアントとサーバの両アプリケーションがファクトリ情報にアクセスしてこれを更新できるように、アプリケーション・プログラミング・インターフェイス (API) のセットが提供されています。

このリリースで複数ドメインをサポートしたことにより、多数のマシンを使用するようスケーリングしたり、アプリケーション環境を分割したりする必要のある顧客にとっての利便性が向上しています。複数ドメインをサポートするために、BEA Tuxedo 環境内でファクトリを見つけるためのメカニズムが拡張され、あるドメインのファクトリを別のドメインでも認識できるようになっています。別のドメインのファクトリの可視性は、システム管理者によって制御されます。

機能、制限事項、および要件

アプリケーション・ファクトリは、サーバ・アプリケーションの初期化中に NameManager で登録する必要があります。それにより、クライアントに FactoryFinder のオブジェクト・リファレンスを提供し、それらがファクトリ登録時に作成された関連する名前に基づくファクトリ・オブジェクト・リファレンスを取得することを可能にします。

以下の機能、制限事項、および要件が、このリリースに適用されます。

- FactoryFinder インターフェイスは、`CosLifeCycle::FactoryFinder` インターフェイスに準拠しています。
- サーバ・アプリケーションでは、CORBA サービス・ネーミング・サービスでアプリケーション・ファクトリを登録したり、登録を削除したりできます。
- クライアントは、唯一のエントリ・ポイントである FactoryFinder を使用してオブジェクトにアクセスできます。
- クライアントは、CORBA サービス・インターフェイスへの BEA Tuxedo の拡張により実現する、簡素化された BEA スキーマ、またはより一般的な CORBA スキーマを使用して、オブジェクト名を構成できます。
- 複数の FactoryFinder と NameManager を使用することで、そのうちの 1 つが失敗した場合に備えて、可用性と信頼性を高めることができます。
- 複数ドメインをサポートしています。あるドメイン内のファクトリを、管理者の制御により別のドメイン内で認識できるようにコンフィギュレーションできます。

- 2 つの NameManager は、少なくとも個別のマシン上にコンフィギュレーションする必要があります。それにより、プロセスのエラーを経てもファクトリからオブジェクトへのリファレンス・マッピングが維持されます。両方の NameManager が失敗した場合は、登録されたファクトリの永続的なジャーナルを残していたマスタ NameManager が、ジャーナルを処理して内部状態を再確立することによって、以前の状況を回復します。
- NameManager の 1 つをマスタに指定し、このマスタ NameManager を、スレーブよりも前に起動する必要があります。マスタ NameManager は、1 つまたは複数のスレーブよりも後に起動された場合には、自身が初期化モードではなくリカバリ・モードに入っていると想定します。

機能説明

BEA Tuxedo CORBA 環境では、クライアントがオブジェクトへのリファレンスを取得するための主な手段として、ファクトリ・デザイン・パターンの使用が促進されます。このデザイン・パターンを使用することにより、クライアント・アプリケーションには、別のオブジェクトのファクトリとして動作するオブジェクトへのリファレンスを取得するメカニズムが必要となります。BEA Tuxedo 環境では、CORBA を可視プログラミング・モデルとして選択しているため、ファクトリをロケートするためのメカニズムは、Object Management Group の CORBA サービス仕様 (1997 年 12 月) の第 6 章「Life Cycle Service」で示されているように、FactoryFinder を モデルとしています。

CORBA FactoryFinder モデルでは、アプリケーション・サーバは活性化されたファクトリを FactoryFinder に登録します。アプリケーション・サーバのファクトリが不活性になると、アプリケーション・サーバは FactoryFinder から対応する登録を削除します。クライアント・アプリケーションは、FactoryFinder に問い合せを行うことによって、ファクトリをロケートします。クライアント・アプリケーションは、1 つまたは複数のリファレンスの選択に使用される基準を指定することによって、返されたファクトリ・オブジェクトへのリファレンスを制御できます。

FactoryFinder のロケート

クライアント・アプリケーションは、適切なファクトリを探し始める前に、FactoryFinder へのリファレンスを取得する必要があります。クライアント・アプリケーションが関連付けられているドメイン内の FactoryFinder へのリファレンスを取得するには、クライアント・アプリケーションは次の 2 つのブートストラップ処理メカニズムのうち、いずれかを使用します。

- `"FactoryFinder"` の値を持つ `Tobj_Bootstrap::resolve_initial_references` オペレーションを呼び出します。このオペレーションは、クライアント・アプリケーションが現在属しているドメイン内の FactoryFinder へのリファレンスを返します。BEA Tuxedo クライアント・ソフトウェアを使用している場合は、このメカニズムを使用します。詳細については、「Tobj_Bootstrap::resolve_initial_references」を参照してください。

- `"FactoryFinder"` の値を持つ `CORBA::ORB::resolve_initial_references` オペレーションを呼び出します。このオペレーションは、クライアント・アプリケーションが現在属しているドメイン内の FactoryFinder へのリファレンスを返します。サード・パーティ製のクライアント ORB を使用している場合は、このメカニズムを使用します。詳細については、CORBA::ORB::resolve_initial_references を参照してください。

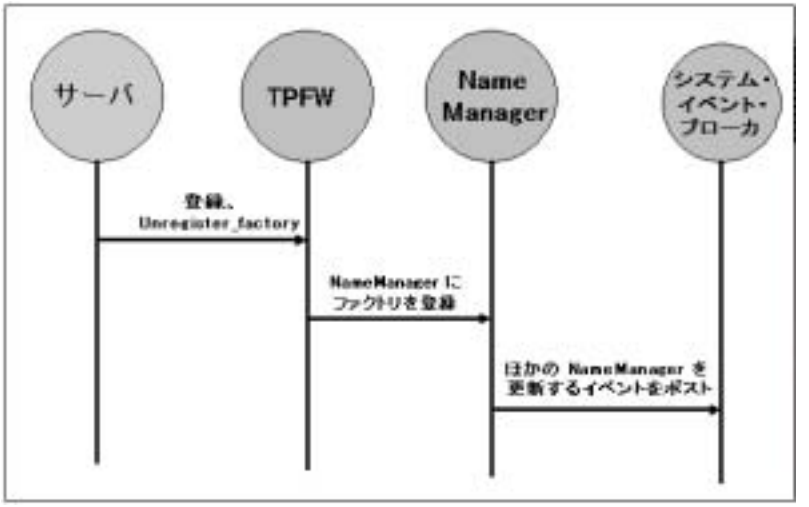
注記　クライアント・アプリケーションに返される FactoryFinder へのリファレンスは、その FactoryFinder と同じマシンに登録されたファクトリ・オブジェクトへのリファレンス、別のマシンに登録されたファクトリ・オブジェクトへのリファレンス、または別のドメインに登録されたファクトリ・オブジェクトへのリファレンスである可能性があります。

ファクトリの登録

クライアント・アプリケーションがファクトリへのリファレンスを取得できるようになるには、アプリケーション・サーバが、FactoryFinder によるインプリメンテーションの提供先となる任意のファクトリ・オブジェクトへのリ

ファレンスを登録する必要があります (図 5-1 を参照)。BEA Tuxedo CORBA TP フレームワークにより、ファクトリ・オブジェクトのリファレンスの登録は、それが作成された後に TP::register_factory オペレーションを使用することにより行われます。ファクトリ・オブジェクトへのリファレンスは、そのファクトリを識別する値と共に、このオペレーションに渡されます。ファクトリ・オブジェクトのリファレンスの登録は、通常はアプリケーションの初期化手順の一部 (普通は、オペレーション Server::initialize のインプリメンテーションの一部) として行われます。

図 5-1 ファクトリ・オブジェクトの登録



サーバ・アプリケーションは、シャットダウン時には以前にアプリケーション・サーバに登録されたファクトリ・オブジェクトへのリファレンスを、すべて登録を削除する必要があります。これは、ファクトリ・オブジェクトへの同じリファレンスと、ファクトリの識別に使用する対応する値を、TP::unregister_factory オペレーションに渡すことによって行われます。登録が削除されると、ファクトリ・オブジェクトへのリファレンスは破棄できるようになります。FactoryFinder でのファクトリの登録を削除するプロセスは、通常は Server::release オペレーションのインプリメンテーションの一環として行われます。これらのオペレーションの詳細については、「Server インターフェイス」を参照してください。

C++ マッピング

リスト 5-1 では、C++ のクラスの (静的) メソッドを示します。これらのメソッドの詳細については、「TP::register_factory()」および「TP::unregister_factory()」を参照してください。

コードリスト 5-1 ファクトリ登録のための C++ マッピングの擬似 OMG IDL

```
#include <TP.h>

static void TP::register_factory(
            CORBA::Object_ptr factory_or, const char* factory_id);

static void TP::unregister_factory (
             CORBA::Object_ptr factory_or, const char* factory_id);
```

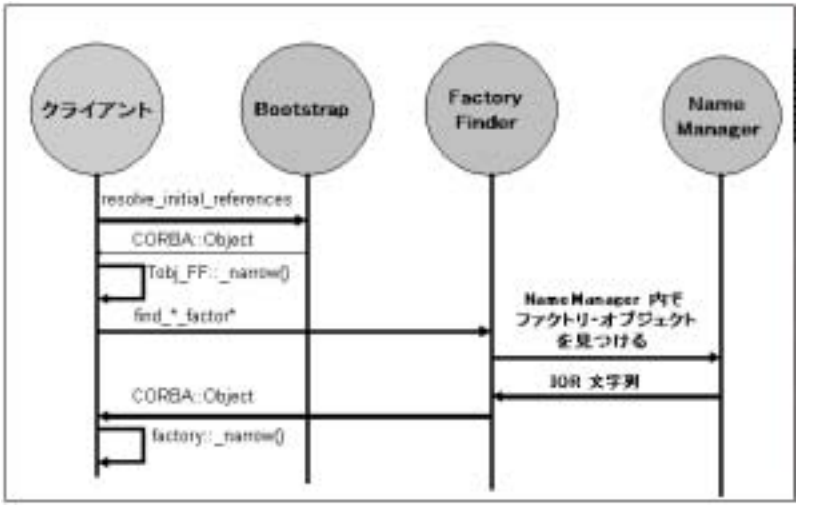
TP.h ヘッダ・ファイルには、2 つのメソッド宣言が含まれています。このファイルは、これらのメソッドを使用するすべてのサーバ・アプリケーションに含める必要があります。

一般に、サーバ・アプリケーションはこのヘッダ・ファイルを、アプリケーション・サーバの初期化と解放のためのメソッドを含むアプリケーション・ファイル内に格納しています。

ファクトリのロケート

クライアントが、オブジェクトへのリファレンスを作成するファクトリを要求するためには、まずファクトリ・オブジェクトへのリファレンスを取得することが必要です。ファクトリ・オブジェクトへのリファレンスは、特定の選択基準で FactoryFinder に問い合せを行うことによって、取得します (図 5-2 参照)。基準は、使用されている特定の FactoryFinder インターフェイスおよびメソッドの形式によって決まります。

図 5-2 ファクトリ・オブジェクトのロケート



BEA Tuxedo CORBA は、FactoryFinder 用に宣言された `find_factories()` メソッドに加えて、4 つのメソッドを導入することによって、`CosLifeCycle::FactoryFinder` インターフェイスを拡張します。したがって、Tobj の拡張により、クライアントは `find_factories()` メソッドまたは `find_factories_by_id()` メソッドのいずれかを使って、アプリケーション・ファクトリのリストを取得できます。またクライアントは、単一のアプリケーション・ファクトリを取得するために `find_one_factory()` メソッドまたは `find_one_factory_by_id()` メソッドを使用し、登録されている全ファクトリのリストを取得するために `list_factories ()` メソッドを使用します。

注記　Tobj_Bootstrap オブジェクトを使用する場合は、`CosLifeCycle::FactoryFinder` インターフェイスに BEA Tuxedo CORBA の拡張を使用できますが、このオブジェクトはファクトリを見つけるために必須のものではありません。CORBA INS を使用していれば、`CosLifeCycle::FactoryFinder` インターフェイスにより提供される `find_factories()` メソッドを使用できます。

`CosLifeCycle::FactoryFinder` インターフェイスは、`factory_key` を定義します。これは、下記の CosNaming 名に準拠した、`id` 文字列と `kind` 文字列のシーケンスです。アプリケーション・ファクトリの登録時に、TP フレーム

ワークによって、全アプリケーション・ファクトリの NameComponent における `kind` フィールドが、文字列 `FactoryInterface` に設定されます。アプリケーションは、`id` フィールドの値を自身で指定します。

次のリストでは、CORBA サービス・ライフサイクル・サービスのモジュールが、自身のファイル (それぞれ `ns.idl` および `lcs.idl`) 内にあると仮定して、BEA Tuxedo FactoryFinder の使用に適した両ファイルのサブセットのための OMG IDL コードのみが示されます。

CORBA サービス・ネーミング・サービス・モジュールの OMG IDL

リスト 5-2 では、FactoryFinder に関連の ns.idl ファイルを部分的に示します。

コードリスト 5-2 CORBA **サービス・ネーミングの** OMG IDL

```
// ------  ns.idl  ------

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

};

// この情報は、「CORBAservices: Common Object Services
// Specification」(1995 年 3 月 31 日改訂版、1997 年 11 月更新 )
// の 6 ～ 10 ページから、OMG の許可を得て転載
```

CORBA サービス・ライフサイクル・サービス・モジュールの OMG IDL

リスト 5-3 では、FactoryFinder に関連の lcs.idl ファイルを部分的に示します。

コードリスト 5-3 **ライフサイクル・サービスの** OMG IDL

```
// -----  lcs.idl  -----

#include “ns.idl”

module CosLifeCycle{
       typedef CosNaming::Name Key;
       typedef Object Factory;
        typedef sequence<Factory> Factories;
```

```
        exception NoFactory{ Key search_key; }

        interface FactoryFinder {
             Factories find_factories(in Key factory_key)
                   raises(NoFactory);

         };
};

// この情報は、「CORBAservices: Common Object Services
// Specification」(1995 年 3 月 31 日改訂版、1997 年 11 月更新 )
// の 6 ～ 10 ページから、OMG の許可を得て転載
```

Tobj モジュールの OMG IDL

リスト 5-4 は、Tobj モジュールの OMG IDL を示します。

コードリスト 5-4 Tobj **モジュールの** OMG IDL

```
// -----  Tobj.idl  -----

module Tobj {

   // 定数

   const string FACTORY_KIND = "FactoryInterface";

   // 例外

   exception CannotProceed { };
   exception InvalidDomain {};
   exception InvalidName { };
   exception RegistrarNotAvailable { };

   // LifeCycle サービスへの拡張

   struct FactoryComponent {
        CosLifeCycle::Key factory_key;
        CosLifeCycle::Factory factory_ior;
   };

   typedef sequence<FactoryComponent> FactoryListing;
```

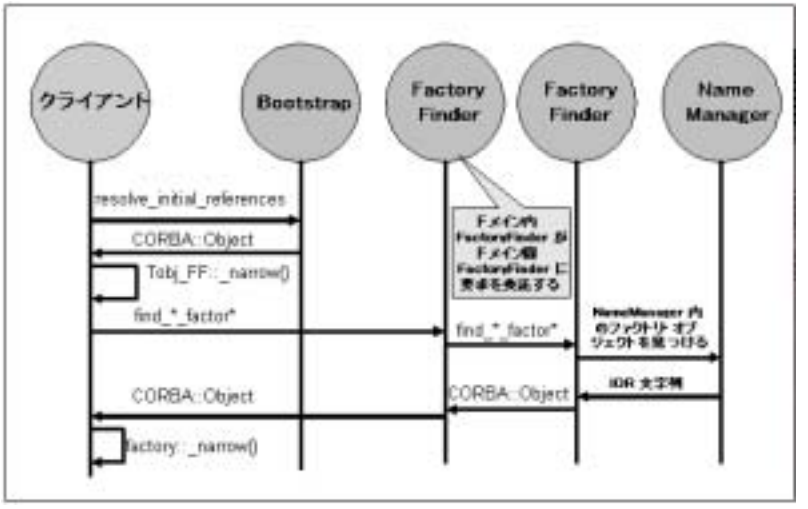
```
   interface FactoryFinder : CosLifeCycle::FactoryFinder {
     CosLifeCycle::Factory find_one_factory(in CosLifeCycle::Key
                                            factory_key)
                    raises (CosLifeCycle::NoFactory,
                            CannotProceed,
                            RegistrarNotAvailable);
      CosLifeCycle::Factory find_one_factory_by_id(in string
                                                   factory_id)
                     raises (CosLifeCycle::NoFactory,
                             CannotProceed,
                             RegistrarNotAvailable);
      CosLifeCycle::Factories find_factories_by_id(in string
                                                   factory_id)
                       raises (CosLifeCycle::NoFactory,
                               CannotProceed,
                               RegistrarNotAvailable);
      FactoryListing list_factories()
                         raises (CannotProceed,
                                 RegistrarNotAvailable);
   };
};
```

別ドメイン内のファクトリのロケート

通常、FactoryFinder は自身と同じドメイン内にあるファクトリ・オブジェクトへのリファレンスを返します。しかし、FactoryFinder が存在しているドメイン以外のドメイン内のファクトリ・オブジェクトへのリファレンスを返すこともできます。これは、FactoryFinder に別ドメイン内のファクトリの情報が含まれている場合に実現可能です (図 5-3 参照)。FactoryFinder は、これらのほかのファクトリ・オブジェクトの場所を説明するコンフィギュレーション情報から、これらドメイン間のファクトリ・オブジェクトについて調べます。

FactoryFinder は、ファクトリ・オブジェクトをロケートする要求を受信すると、まず指定された基準に合致するファクトリ・オブジェクトへのリファレンスが存在するかどうかを判断する必要があります。基準に合致するファクトリ・オブジェクトの登録情報があれば、FactoryFinder はそれが現在のドメインから見てローカルな場所にあるのか、それとも別ドメインからのインポートを必要とするのかを判断する必要があります。ファクトリ・オブジェクトがローカル・ドメインからのものであれば、FactoryFinder はファクトリ・オブジェクトへのリファレンスをクライアントに返します。

図 e 5-3 ドメイン間の FactoryFinder の対話



その一方で、もしその情報が、実際のファクトリ・オブジェクトが別ドメインからのものであることを示していれば、FactoryFinder は要求を適切なドメインにあるドメイン間の FactoryFinder に委譲します。その結果、ファクトリ・オブジェクトと同じドメイン内の FactoryFinder のみが、ファクトリ・オブジェクトへの実際のリファレンスを含むこととなります。ドメイン間の FactoryFinder は、ファクトリ・オブジェクトのリファレンスをローカルの FactoryFinder に返す役割を果たします。そこからその要求は、クライアントに返されます。

BEA Tuxedo CORBA の拡張を使用する理由

BEA Tuxedo ソフトウェアは、Object Management Group の CORBA サービス仕様 (1997 年 12 月) の第 6 章「Life Cycle Service」で定義されているインターフェイスを、以下の理由により拡張しています。

- CORBA 定義の手法は強力で、さまざまな選択基準を使用できますが、FactoryFinder の問い合せを行うためのインターフェイスの使用方法が複雑になる場合があります。
- 加えて、クライアント・アプリケーションによって指定された選択基準が十分な具体性を持たなかった場合、ファクトリ・オブジェクトへのリ

ファレンスが複数、返される可能性があります。その場合、クライアント・アプリケーションが次に行うべき動作が、すぐには判明しません。

- 最後に、CORBA サービスの仕様では、アプリケーション・サーバがファクトリ・オブジェクトを登録する手段となる標準化されたメカニズムが指定されていません。

したがって、BEA Tuxedo は FactoryFinder を使いやすくするために、CORBA サービスの仕様で定義されているインターフェイスを拡張します。拡張は、CORBA サービスの仕様で指定されたインターフェイスから派生する、FactoryFinder の高度化したインターフェイスとなります。

アプリケーション・ファクトリ・キーの作成

FactoryFinder インターフェイスで提供される 5 つのメソッドのうち 2 つが、`CosNaming::Name` に対応する `CosLifeCycle::Keys` を受け付けます。クライアントは、これらのキーを構築できなければなりません。

CosNaming の仕様では、`CosLifeCycle::Keys` の作成と操作に使用可能な名前ライブラリ・インターフェイスを構成する、2 つのインターフェイスが説明されています。これらのインターフェイスの擬似 OMG IDL 文については、次の節で説明します。

名前ライブラリ・インターフェイスの擬似 OMG IDL

注記　この情報は、「CORBAservices: Common Object Services Specification」(1995 年 3 月 31 日改訂版、1997 年 11 月更新) の 3-14 ～ 3-18 ページから、OMG の許可を得て転載しています。

既存のクライアント・アプリケーションに影響を与えることなく名前の表現を生成するには、名前の表現をクライアント・アプリケーションから認識できないようにしたほうがよいでしょう。名前自体がオブジェクトとなることが理想ですが、名前は作成、操作、送信が効率的に行える、ライトウェイトのエンティティでなければなりません。したがって、名前は名前ライブラリからプログラムに提示されます。

名前ライブラリは、名前を擬似オブジェクトとしてインプリメントします。クライアント・アプリケーションは、通常のオブジェクトを呼び出す場合と同じやり方で、擬似オブジェクトに対し呼び出しを行います。ライブラリ名は、擬似 IDL で記述されます。これにより、適切な言語バインディングが示唆されます。C++ クライアント・アプリケーションは、擬似 IDL (PIDL) でも、IDL の場合と同じクライアント言語バインディングを使用します。

擬似オブジェクト・リファレンスは、OMG IDL インターフェイス間で渡すことはできません。「CORBAservices: Common Object Services Specification」の第 3 章の「The CosNaming Module」で説明されるように、CORBA サービス・ネーミング・サービスは、NamingContext OMG IDL インターフェイスをサポートしています。名前ライブラリは、NamingContext インターフェイスを通じてライブラリ名をネーム・サービスに渡せる値に変換するオペレーションをサポートしています。

注記　CORBA サービス・ネーミング・サービスを使用するのに、名前ライブラリの使用は必須ではありません。

名前ライブラリは、リスト 5-5 に示すように LNameComponent インターフェイスおよび LName インターフェイスという 2 つの擬似 IDL インターフェイスで構成されています。

コードリスト 5-5　擬似 IDL による名前ライブラリ・インターフェイス

```
interface LNameComponent { //  PIDL
    const short MAX_LNAME_STRLEN = 128;

    exception NotSet{ };
    exception OverFlow{ };

    string get_id
           raises (NotSet);
    void set_id(in string i)
           raises (OverFlow);
    string get_kind()
           raises(NotSet);
    void set_kind(in string k)
           raises (OverFlow);
    void destroy();
};
```

```
interface LName {// PIDL
       exception NoComponent{ };
       exception OverFlow{ };
       exception InvalidName{ };
       LName insert_component(in unsigned long i,
                     in LNameComponent n)
              raises (NoComponent, OverFlow);
       LNameComponent get_component(in unsigned long i)
              raises (NoComponent);
       LNameComponent delete_component(in unsigned long i)
              raises (NoComponent);
       unsigned long num_components();
       boolean equal(in LName ln);
       boolean less_than(in LName ln);
       Name to_idl_form()
              raises (InvalidName);
       void from_idl_form(in Name n);
       void destroy();
};

LName create_lname();// C/C++
LNameComponent create_lname_component();// C/C++
```

ライブラリ名コンポーネントの作成

ライブラリ名コンポーネントの擬似オブジェクトを作成するには、次の C/C++ 関数を使用します。

```
LNameComponent create_lname_component();    // C/C++
```

返された擬似オブジェクトは、その後 リスト 5-5 に示すオペレーションを使用して、操作できます。

ライブラリ名の作成

ライブラリ名の擬似オブジェクトを作成するには、次の C/C++ 関数を使用します。

```
LName create_lname();        // C/C++
```

返された擬似オブジェクト・リファレンスは、その後 リスト 5-5 に示すオペレーションを使用して、操作できます。

LNameComponent インターフェイス

名前コンポーネントは、identifier および kind という 2 つの属性で構成されています。LNameComponent インターフェイスは、次のように、これらの属性と関連付けられたオペレーションを定義します。

```
string get_id()
raises(NotSet);
void set_id(in string k);
string get_kind()
raises(NotSet);
void set_kind(in string k);
```

get_id
: get_id オペレーションは、identifier 属性の値を返します。属性が設定されていなければ、NotSet 例外が発生します。

set_id
: set_id オペレーションは、identifier 属性を文字列引数に設定します。

get_kind
: get_kind オペレーションは、kind 属性の値を返します。属性が設定されていなければ、NotSet 例外が発生します。

set_kind
: set_kind オペレーションは、kind 属性を文字列引数に設定します。

LName インターフェイス

この節では、以下のオペレーションについて説明します。

- ライブラリ名コンポーネント擬似オブジェクトの破棄
- 名前コンポーネントの挿入
- i 番目の名前コンポーネントの取得
- 名前コンポーネントの削除
- 名前コンポーネントの数
- 等価性のテスト
- 順序のテスト

- OMG IDL 形式の生成
- OMG IDL 形式の変換
- ライブラリ名擬似オブジェクトの破棄

ライブラリ名コンポーネント擬似オブジェクトの破棄

destroy オペレーションは、ライブラリ名コンポーネント擬似オブジェクトを破棄します。

```
void destroy();
```

名前コンポーネントの挿入

名前には、1 つまたは複数のコンポーネントがあります。各コンポーネントは、最後のもの以外は、サブコンテキストの名前を識別するために使用されます。最後のコンポーネントは、バウンド・オブジェクトを示します。insert_component オペレーションは、位置 i の後にコンポーネントを挿入します。

```
LName insert_component(in unsigned long i, in LNameComponent lnc)
raises(NoComponent, OverFlow);
```

コンポーネント i-1 が未定義で、コンポーネント i が 1 より大きい場合、insert_component オペレーションは、NoComponent 例外を生成します。

ライブラリが、挿入されたコンポーネントに資源を割り当てられない場合は、OverFlow 例外が発生します。

i 番目の名前コンポーネントの取得

get_component オペレーションは、i 番目のコンポーネントを返します。最初のコンポーネントの番号を「1」とします。

```
LNameComponent get_component(in unsigned long i)
raises(NoComponent);
```

コンポーネントが存在しない場合は、NoComponent 例外が発生します。

名前コンポーネントの削除

delete_component オペレーションは、i 番目のコンポーネントを削除して返します。

```
LNameComponent delete_component(in unsigned long i)
     raises(NoComponent);
```

コンポーネントが存在しない場合は、NoComponent 例外が発生します。

delete_component オペレーションの実行後は、複合名のコンポーネントが 1 つ減り、それまで i+1...n と認識されてきたコンポーネントが、i...n-1 と認識されるようになります。

名前コンポーネントの数

num_components オペレーションは、ライブラリ名に含まれるコンポーネントの数を返します。

```
unsigned long num_components();
```

等価性のテスト

equal オペレーションは、ライブラリ名 ln との等価性をテストします。

```
boolean equal(in LName ln);
```

順序のテスト

less_than オペレーションは、ライブラリ名 ln に対しての、ライブラリ名の順序をテストします。

```
boolean less_than(in LName ln);
```

このオペレーションは、ライブラリ名が、引数として渡されたライブラリ名 ln よりも小さい値の場合に、TRUE を返します。ライブラリのインプリメンテーションは、名前の順序付けを定義します。

OMG IDL 形式の生成

擬似オブジェクトは、OMG IDL インターフェイス間で渡すことはできません。ライブラリ名は、擬似オブジェクトです。したがって、CORBA サービス・ネーミング・サービスの OMG IDL インターフェイスを越えて渡すことはできません。NamingContext インターフェイスのいくつかのオペレーションには、OMG IDL 定義の構造体 Name の引数があります。ライブラリ名に対する次の PIDL オペレーションは、OMG IDL 要求にわたって渡すことが可能な構造体を生成します。

```
Name to_idl_form()
    raises(InvalidName);
```

名前の長さが 0 (ゼロ) の場合、InvalidName 例外が返されます。

IDL 形式の変換

擬似オブジェクトは、OMG IDL インターフェイス間で渡すことはできません。ライブラリ名は、擬似オブジェクトです。したがって、CORBA サービス・ネーミング・サービスの OMG IDL インターフェイスを越えて渡すことはできません。NamingContext インターフェイスは、Name 型の IDL struct を返すオペレーションを定義します。ライブラリ名に対する次の PIDL オペレーションは、返された OMG IDL 構造体 Name からのライブラリ名のコンポーネントおよび kind 属性を設定します。

```
void from_idl_form(in Name n);
```

ライブラリ名擬似オブジェクトの破棄

destroy オペレーションは、ライブラリ名擬似オブジェクトを破棄します。

```
void destroy();
```

C++ マッピング

名前ライブラリの擬似 OMG IDL インターフェイスは、リスト 5-6 に示す C++ クラスにマッピングします。このリストは、NamesLib.h ヘッダ・ファイルにあります。

スケーラビリティをサポートするために、CORBA に対する 2 つの BEA Tuxedo 拡張が含まれています。具体的には、入力文字列の長さが MAX_LNAME_STRLEN を超えると、LNameComponent::set_id() メソッドおよび LNameComponent::set_kind() メソッドが、OverFlow 例外を生成します。この長さは、BEA Tuxedo オブジェクト ID (OID) およびインターフェイス名の最大長に一致します。ライブラリ名クラスの詳細については、「名前ライブラリ・インターフェイスの擬似 OMG IDL」を参照してください。

コードリスト 5-6 ライブラリ名クラス

```
const short MAX_LNAME_STRLEN = 128;

class LNameComponent {
public:
      class NotSet{ };
      class OverFlow{ };
      static LNameComponent* create_lname_component();
      void   destroy();
      const char*   get_id() const throw (NotSet);
      void          set_id(const char* i) throw (OverFlow);
      const char*   get_kind() const throw (NotSet);
      void          set_kind(const char* k) throw (OverFlow);
};

class LName {
public:
       class NoComponent{ };
       class OverFlow{ };
       class InvalidName{ };
       static LName* create_lname();
       void destroy();
       LName* insert_component(const unsigned long i,
                    LNameComponent* n)
              throw (NoComponent, OverFlow);
       const LNameComponent* get_component(
                    const unsigned long i) const
              throw (NoComponent);
       const LNameComponent* delete_component(
                    const unsigned long i)
              throw (NoComponent);
       unsigned long num_components() const;
       CORBA::Boolean equal(const LName* ln) const;
       CORBA::Boolean less_than(
              const LName* ln) const; // インプリメントされていない
       CosNaming::Name* to_idl_form()
```

```
            throw (InvalidName);
        void from_idl_form(const CosNaming::Name& n);
};
```

Java マッピング

名前ライブラリ擬似 OMG IDL インターフェイスは、リスト 5-7 で示す com.beasys.Tobj パッケージに含まれている Java クラスにマッピングします。例外はすべて、同じパッケージ内に包含されています。

ライブラリ名クラスの詳細については、「*CORBAservices: Common Object Services Specification*」の第 3 章を参照してください。

コードリスト 5-7　LNameComponent **の** Java **マッピング**

```
public class LNameComponent {
    public static LNameComponent create_lname_component();
    public static final short MAX_LNAME_STRING = 128;
    public void destroy();
    public String get_id() throws NotSet;
    public void   set_id(String i) throws OverFlow;
    public String get_kind() throws NotSet;
    public void   set_kind(String k) throws OverFlow;
};

public class LName {

   public static LName create_lname();
   public void destroy();
   public LName insert_component(long i, LNameComponent n)
     throws NoComponent, OverFlow;
   public LNameComponent get_component(long i)
     throws NoComponent;
   public LNameComponent delete_component(long i)
     throws NoComponent;
   public long num_components();
   public boolean equal(LName ln);
   public boolean less_than(LName ln);// インプリメントされていない
   public org.omg.CosNaming.NameComponent[] to_idl_form()
     throws InvalidName;
  public void from_idl_form(org.omg.CosNaming.NameComponent[] nr);
};
```

C++ メンバ関数と Java メソッド

この節では、FactoryFinder の C++ メンバ関数と Java メソッドについて説明します。

注記　LName の `less_than` メンバ関数を除き、FactoryFinder メンバ関数はすべて、C++ でも Java でもインプリメントされます。

ここでは、以下のメソッドについて説明します。

- `CosLifeCycle::FactoryFinder::find_factories`
- `Tobj::Factoryfinder::find_one_factory`
- `Tobj::Factoryfinder::find_one_factory_by_id`
- `Tobj::Factoryfinder::find_factories_by_id`
- `Tobj::Factoryfinder::list_factories`

注記　`CosLifeCycle::FactoryFinder::find_factories` メソッドは、標準的な CORBA CosLifeCycle メソッドです。4 つの Tobj メソッドは、CosLifeCycle インターフェイスに対する拡張であり、したがって、CosLifeCycle インターフェイスの属性を継承しています。

CosLifeCycle::FactoryFinder::find_factories

概要 ファクトリ・オブジェクト・リファレンスのシーケンスを取得します。

C++ マッピング

```
CosLifeCycle::Factories *
CORBA::Object_ptr CosLifeCycle::FactoryFinder::find_factories(
            const CosNaming::Name& factory_key)
      throw (CosLifeCycle::NoFactory);
```

Java マッピング

```
import org.omg.CosLifeCycle.*;

public org.omg.CORBA.Object[] find_factories(
          org.omg.CosNaming.NameComponent[] factory_key)
    throws org.omg.CosLifeCycle.NoFactory;
```

パラメータ

`factory_key`

このパラメータは、ファクトリ・オブジェクト・リファレンスを一意に識別する、NameComponents (<id, kind> ペアのタプル) のアンバウンディッド・シーケンスです。

NameComponent は、`id` および `kind` の 2 つのメンバを持つものとして定義されます。これらのメンバは双方とも、文字列型です。`id` フィールドは、ファクトリ・オブジェクトの ID を表すのに使用されます。`kind` フィールドは、`id` フィールドの値をどのように解釈するかを示すのに使用されます。

オペレーション `TP::register_factory` を使用して登録されたファクトリ・オブジェクトへのリファレンスの kind 値は、`"FactoryInterface"` となります。

例外

`CORBA::BAD_PARAM`

入力パラメータの値が、不適切であるか無効であることを示します。特に重要なのは、この例外が、パラメータ `factory_key` に指定された値が存在しないか、NULL 値であった場合に発生するということです。

`CosLifeCycle::NoFactory`

`factory_key` パラメータの情報に一致するファクトリが登録されていないことを示します。

説明 アプリケーションによって `find_factories` メソッドが呼び出され、ファクトリ・オブジェクト・リファレンスのシーケンスが取得されます。オペレーションには、必要なファクトリを識別するキーが渡されます。このキーは、

CORBA サービス・ネーミング・サービスで定義した名前です。キーに一致するファクトリは複数存在することがあります。そのような場合、FactoryFinder はファクトリのシーケンスを返します。

このキーのスコープは、FactoryFinder です。FactoryFinder は、キーにセマンティクスを割り当てません。単に、キーを照合するだけです。インターフェイスや返されたファクトリのインプリメンテーション、または作成するオブジェクトについての保証も行いません。

キー値が等価であると見なされるのは、長さが等しい、つまりシーケンス中のエレメント数が同じである場合と、キーの中の各 NameComponent 値がすべて、ファクトリ・オブジェクトへのリファレンス登録時に指定されたキーの中のまったく同じ場所にある、対応する NameComponent 値に一致する場合です。

戻り値

`factory_key` パラメータの値として指定された情報に一致する、ファクトリ・オブジェクトへのリファレンスのアンバウンディッド・シーケンスです。C++ では、メソッドは `CosLifeCycle::Factory` 型のオブジェクト・リファレンスのシーケンスを返します。Java では、メソッドは `org.omg.CORBA.Object` 型のオブジェクト・リファレンスのアンバウンディッド配列を返します。

オペレーションが例外を生成した場合には、戻り値は無効であり、呼び出し側によって解放される必要はありません。

単一のファクトリ・オブジェクトへのリファレンスを取得します。

C++ マッピング

```
virtual CosLifeCycle::Factory_ptr
       find_one_factory( const CosNaming::Name& factory_key) = 0;
```

Java マッピング

```
public org.omg.CORBA.Object
  find_one_factory( org.omg.CosNaming.NameComponent[] factory_key)
     throws
        org.omg.CosLifeCycle.NoFactory,
        com.beasys.Tobj.CannotProceed,
        com.beasys.Tobj.RegistrarNotAvailable;
```

パラメータ

`factory_key`

このパラメータは、ファクトリ・オブジェクト・リファレンスを一意に識別する、NameComponents (<id, kind> ペアのタプル) のアンバウンディッド・シーケンスです。

NameComponent は、id および kind の 2 つのメンバを持つものとして定義されます。これらのメンバは双方とも、文字列型です。id フィールドは、ファクトリ・オブジェクトの ID を表すのに使用されます。kind フィールドは、id フィールドの値をどのように解釈するかを示すのに使用されます。
オペレーション TP::register_factory を使用して登録されたファクトリ・オブジェクトへのリファレンスの kind 値は、"FactoryInterface" となります。

例外 CORBA::BAD_PARAM
入力パラメータの値が、不適切であるか無効であることを示します。特に重要なのは、この例外が、パラメータ factory_key に指定された値が存在しないか、NULL 値であった場合に発生するということです。

CosLifeCycle::NoFactory
factory_key パラメータの情報に一致するファクトリが登録されていないことを示します。
FactoryFinder または NameManager が、ファクトリ・オブジェクトのリファレンスを見つけようとしているときに内部エラーに遭遇したことを示します。
エラー情報は、ユーザ・ログに書き込まれます。
FactoryFinder が NameManager と通信できなかったことを示します。
エラー情報は、ユーザ・ログに書き込まれます。

説明 アプリケーションにより find_one_factory メソッドが呼び出され、メソッドへの入力として指定されたキーの値と一致するキーを備えた唯一のファクトリ・オブジェクトへのリファレンスが取得されます。指定されたキーで複数のファクトリ・オブジェクトが登録されている場合、FactoryFinder のロード・バランシング・スキーマに基づき、FactoryFinder はファクトリ・オブジェクトを 1 つ選択します。その結果、同じキーで find_one_factory メソッドを複数回、呼び出すと、さまざまなオブジェクト・リファレンスが返ることがあります。

このキーのスコープは、FactoryFinder です。FactoryFinder は、キーにセマンティクスを割り当てません。単に、キーを照合するだけです。インターフェイスや返されたファクトリのインプリメンテーション、または作成するオブジェクトについての保証も行いません。

キー値が等価であると見なされるのは、長さが等しい、つまりシーケンス中のエレメント数が同じである場合と、キーの中の各 NameComponent 値がすべて、ファクトリ・オブジェクトへのリファレンス登録時に指定されたキーの中のまったく同じ場所にある、対応する NameComponent 値に一致する場合です。

戻り値 ファクトリ・オブジェクトのオブジェクト・リファレンスです。C++ では、メソッドは `CosLifeCycle::Factory` 型のオブジェクト・リファレンスを返します。Java では、メソッドは `org.omg.CORBA.Object` 型のオブジェクト・リファレンスを返します。

オペレーションが例外を生成した場合には、戻り値は無効であり、呼び出し側によって解放される必要はありません。

概要 単一のファクトリ・オブジェクトへのリファレンスを取得します。

C++ マッピング

Java マッピング

```
public org.omg.CORBA.Object
    find_one_factory_by_id( java.lang.String factory_id)
      throws
        org.omg.CosLifeCycle.NoFactory,
        com.beasys.Tobj.CannotProceed,
        com.beasys.Tobj.RegistrarNotAvailable;
```

パラメータ

`factory_id`

探している登録済みファクトリ・オブジェクトを識別するための値を含む、NULL で終了する文字列です。

`factory_id` パラメータの値は、ファクトリ・オブジェクトに対する登録済みリファレンスと対比したときに値が `"FactoryInterface"` となっている、`kind` フィールドを備える NameComponent の、`id` フィールドの値として使用されます。

例外

`CORBA::BAD_PARAM`

入力パラメータの値が、不適切であるか無効であることを示します。特に重要なのは、この例外が、パラメータ `factory_key` に指定された値が存在しないか、NULL 値であった場合に発生するということです。

`CosLifeCycle::NoFactory`

`factory_key` パラメータの情報に一致するファクトリが登録されていないことを示します。

FactoryFinder または NameManager が、ファクトリ・オブジェクトのリファレンスを見つけようとしているときに内部エラーに遭遇したことを示します。

エラー情報は、ユーザ・ログに書き込まれます。

FactoryFinder が NameManager と通信できなかったことを示します。

エラー情報は、ユーザ・ログに書き込まれます。

説明 アプリケーションにより `find_one_factory_by_id` メソッドが呼び出され、メソッドへの入力として指定された ID の値と一致する登録 ID を備えた唯一のファクトリ・オブジェクトへのリファレンスが取得されます。指定された ID で複数のファクトリ・オブジェクトが登録されている場合、

FactoryFinder のロード・バランシング・スキーマに基づき、FactoryFinder はファクトリ・オブジェクトを 1 つ選択します。その結果、同じ ID で `find_one_factory_by_id` オペレーションを複数回呼び出すと、さまざまなオブジェクト・リファレンスが返ることがあります。

`find_one_factory_by_id` メソッドは、`factory_id` パラメータと同じ値を含む `id` フィールド、および値 `"FactoryInterface"` を含む `kind` フィールドを備えた単一の NameComponent が含まれるキーを渡された、`find_one_factory` オペレーションと同じように振る舞います。

ファクトリの登録された識別子が `factory_id` パラメータの値と等価であると見なされるのは、単一の NameComponent を含む `CosLifeCycle::Key` 構造体を構成した結果に、`id` フィールドの値として `factory_id` パラメータが、`kind` フィールドの値として `factory_id` パラメータが含まれる場合です。値は、大文字と小文字、場所など、すべての面で一致している必要があります。

戻り値 ファクトリ・オブジェクトに対するオブジェクト・リファレンスです。C++ では、メソッドは `CosLifeCycle::Factory` 型のオブジェクト・リファレンスを返します。Java では、メソッドは `org.omg.CORBA.Object` 型のオブジェクト・リファレンスを返します。

オペレーションが例外を生成した場合には、戻り値は無効であり、呼び出し側によって解放される必要はありません。

Tobj::FactoryFinder::find_factories_by_id

概要 1 つまたは複数のファクトリ・オブジェクト・リファレンスのシーケンスを取得します。

C++ マッピング

Java マッピング

```
public org.omg.CORBA.Object[]
   find_factories_by_id( java.lang.String factory_id)
     throws
         org.omg.CosLifeCycle.NoFactory,
         com.beasys.Tobj.CannotProceed,
         com.beasys.Tobj.RegistrarNotAvailable;
```

パラメータ

`factory_id`

探している登録済みファクトリ・オブジェクトを識別するための値を含む、NULL で終了する文字列です。

`factory_id` パラメータの値は、ファクトリ・オブジェクトに対する登録済みリファレンスと対比したときに値が `"FactoryInterface"` となっている、`kind` フィールドを備える NameComponent の、`id` フィールドの値として使用されます。

例外

`CORBA::BAD_PARAM`

入力パラメータの値が、不適切であるか無効であることを示します。特に重要なのは、この例外が、パラメータ `factory_key` に指定された値が存在しないか、NULL 値であった場合に発生するということです。

`CosLifeCycle::NoFactory`

`factory_key` パラメータの情報に一致するファクトリが登録されていないことを示します。

FactoryFinder または NameManager が、ファクトリ・オブジェクトのリファレンスを見つけようとしているときに内部エラーに遭遇したことを示します。

エラー情報は、ユーザ・ログに書き込まれます。

FactoryFinder が NameManager と通信できなかったことを示します。

エラー情報は、ユーザ・ログに書き込まれます。

説明 アプリケーションによって `find_factories_by_id` メソッドが呼び出され、1 つまたは複数のファクトリ・オブジェクト・リファレンスのシーケンスが取得されます。メソッドには、探しているファクトリの識別子が含まれた、

NULL で終了する文字列が渡されます。指定した ID で登録されたファクトリ・オブジェクトが複数存在した場合、FactoryFinder は一致する登録済みオブジェクトのオブジェクト・リファレンスのリストを返します。

`find_factories_by_id` メソッドは、`factory_id` パラメータと同じ値を含む `id` フィールドおよび値 `"FactoryInterface"` を含む `kind` フィールドを備えた単一の NameComponent が含まれるキーを渡された、`find_factory` オペレーションと同じように振る舞います。

ファクトリの登録された識別子が `factory_id` パラメータの値と等価であると見なされるのは、単一の NameComponent を含む `CosLifeCycle::Key` 構造体を構成した結果に、`id` フィールドの値として `factory_id` パラメータが、`kind` フィールドの値として `factory_id` パラメータが含まれる場合です。値は、大文字と小文字、場所など、すべての面で一致している必要があります。

戻り値 `factory_key` パラメータの値として指定された情報に一致する、ファクトリ・オブジェクトへのリファレンスのアンバウンディッド・シーケンスです。C++ では、メソッドは `CosLifeCycle::Factory` 型のオブジェクト・リファレンスのシーケンスを返します。Java では、メソッドは `org.omg.CORBA.Object` 型のオブジェクト・リファレンスのアンバウンディッド配列を返します。

オペレーションが例外を生成した場合には、戻り値は無効であり、呼び出し側によって解放される必要はありません。

Tobj::Factoryfinder::list_factories

概要 現在 FactoryFinder で登録されているファクトリ・オブジェクトのリストを取得します。

C++ マッピング

```
virtual FactoryListing * list_factories() = 0;
```

Java マッピング

```
public com.beasys.Tobj.FactoryComponent[] list_factories()
  throws
       com.beasys.Tobj.CannotProceed,
        com.beasys.Tobj.RegistrarNotAvailable;
```

例外

FactoryFinder または NameManager が、ファクトリ・オブジェクトのリファレンスを見つけようとしているときに内部エラーに遭遇したことを示します。
エラー情報は、ユーザ・ログに書き込まれます。

FactoryFinder が NameManager と通信できなかったことを示します。
エラー情報は、ユーザ・ログに書き込まれます。

説明 アプリケーションによって `list_factories` メソッドが呼び出され、FactoryFinder で現在登録されているファクトリ・オブジェクトのリストが取得されます。メソッドは、ファクトリの登録に使用されるキーと、ファクトリ・オブジェクトへのリファレンスの両方を返します。

戻り値 `Tobj::FactoryComponent` のアンバウンディッド・シーケンスです。シーケンス中の `Tobj::FactoryComponent` の各オカレンスには、登録されたファクトリ・オブジェクトへのリファレンス、およびそのファクトリ・オブジェクトの登録に使用された `CosLifeCycle::Key` が含まれます。

オペレーションが例外を生成した場合には、戻り値は無効であり、呼び出し側によって解放される必要はありません。

オートメーションのメソッド

ここでは、DITobj_FactoryFinder オートメーション・メソッドについて説明します。

DIMIDL マッピング

```
HRESULT find_one_factory(
  [in] VARIANT factory_key,
  [in,out,optional] VARIANT* exceptionInfo,
  [out,retval] IDispatch** returnValue);
```

オートメーション・マッピング

```
Function find_one_factory(factory_key, [exceptionInfo]) As Object
```

`factory_key`
: このパラメータは、ファクトリ・オブジェクト・リファレンスを一意に識別する DICosNaming_NameComponents (<id, kind> 値のペア) のセーフ配列を含みます。

`exceptionInfo`
: エラーが発生した場合にアプリケーションが追加の例外データを取得できるようにするオプションの入力引数。

`NoFactory`
: この例外は、入力された `factory_key` に対応するアプリケーション・ファクトリ・オブジェクト・リファレンスを FactoryFinder が見つけられない場合に発生します。

`CannotProceed`
: この例外が発生するのは、FactoryFinder または CORBA サービス・ネーミング・サービスが検索中に内部エラーに遭遇した場合です。エラーは、ユーザ・ログ (`ULOG`) に書き込まれます。この例外が発生した場合は、すぐに作業担当者に通知されます。内部エラーの重要度によっては、FactoryFinder または CORBA サービス・ネーミング・サービスが実行されていたサーバが、終了していることがあります。FactoryFinder サービスが終了した場合は、新しい FactoryFinder サービスを開始します。CORBA サービス・ネーミング・サービスの 1 つが終了しており、別の CORBA サービス・ネーミング・サービス は実行中である場合は、新規の CORBA サービス・ネーミング・サービス を開始します。ネーミング・サービス用

のサーバが実行されていない場合は、アプリケーションを再起動します。

`RegistrarNotAvailable`
この例外は、FactoryFinder が CORBA サービス・ネーミング・サービスのオブジェクトを見つけられない場合に発生します。この例外が発生した場合は、すぐに作業担当者に通知されます。ネーミング・サービス用のサーバが実行されていない場合は、アプリケーションを再起動します。

このメンバ関数は、キーが入力 `factory_key` に一致するアプリケーション・ファクトリ・オブジェクト・リファレンスを 1 つ返すように、FactoryFinder に指示します。これを実現するには、メンバ関数が等価性の照合を行う必要があります。つまり、入力 `factory_key` のすべての NameComponent <id, kind> ペアが、アプリケーション・ファクトリのキーの <id, kind> ペアと完全に一致していなければなりません。入力された `factory_key` を含むファクトリ・キーが複数存在する場合、FactoryFinder は内部で定義されたロード・バランシング・スキーマに基づき、1 つを選択します。同じ `id` で `find_one_factory` を複数回呼び出すと、さまざまなオブジェクト・リファレンスが返ることがあります。

戻り値 アプリケーション・ファクトリのインターフェイス・ポインタへのリファレンスを返します。

DI.find_one_factory_by_id

概要 アプリケーション・ファクトリを 1 つ取得します。

MIDL マッピング

```
HRESULT find_one_factory_by_id(
  [in] BSTR factory_id,
  [in,out,optional] VARIANT* exceptionInfo,
  [out,retval] IDispatch** returnValue);
```

オートメーション・マッピング

```
Function find_one_factory_by_id(factory_id As String,
                                [exceptionInfo]) As Object
```

パラメータ

`factory_id`

このパラメータは、アプリケーション・ファクトリの種類または型を識別するために使用される文字列識別子を表します。この文字列を構成する際のヒントについては、『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。

`exceptionInfo`

エラーが発生した場合にアプリケーションが追加の例外データを取得できるようにするオプションの入力引数。

例外

`NoFactory`

この例外は、入力 `factory_id` に対応するアプリケーション・ファクトリ・オブジェクト・リファレンスを FactoryFinder が見つけられない場合に発生します。

この例外が発生するのは、FactoryFinder または CORBA サービス・ネーミング・サービスが検索中に内部エラーに遭遇した場合です。エラーは、ユーザ・ログ (`ULOG`) に書き込まれます。この例外が発生した場合は、すぐに作業担当者に通知されます。内部エラーの重要度によっては、FactoryFinder または CORBA サービス・ネーミング・サービスが実行されていたサーバが、終了していることがあります。FactoryFinder サービスが終了した場合は、新しい FactoryFinder サービスを開始します。CORBA サービス・ネーミング・サービスの 1 つが終了しており、別の CORBA サービス・ネーミング・サービス は実行中である場合は、新規の CORBA サービス・ネーミング・サービス を開始します。稼動中のネーミング・サービスがない場合は、アプリケーションを再起動します。

`RegistrarNotAvailable`

この例外は、FactoryFinder が CORBA サービス・ネーミング・サービスのオブジェクトを見つけられない場合に発生します。この例外が発生した場合は、すぐに作業担当者に通知されます。ネーミング・サービス用のサーバが実行されていない場合は、アプリケーションを再起動します。

説明 このメンバ関数は、キーの `id` がメソッドの入力 `factory_id` に一致するアプリケーション・ファクトリ・オブジェクト・リファレンスを 1 つ返すように、FactoryFinder に指示します。が等価性の照合を行う必要があります。つまり、入力 `factory_id` が、アプリケーション・ファクトリのキーの <id, kind> ペアの `id` と完全に一致していなければなりません。入力された `factory_id` を含むファクトリ・キーが複数存在する場合、FactoryFinder は内部で定義されたロード・バランシング・スキーマに基づき、1 つを選択します。同じ `id` で `find_one_factory_by_id` を複数回呼び出すと、さまざまなオブジェクト・リファレンスが返ることがあります。

戻り値

DI.find_factories_by_id

概要 アプリケーション・ファクトリのリストを取得します。

MIDL マッピング

```
HRESULT find_factories_by_id(
  [in] BSTR factory_id,
  [in,out,optional] VARIANT* exceptionInfo,
  [out,retval] VARIANT* returnValue);
```

オートメーション・マッピング

```
Function find_factories_by_id(factory_id As String,
                              [exceptionInfo])
```

パラメータ

`factory_id`
このパラメータは、アプリケーション・ファクトリの種類または型を識別するために使用される文字列識別子を表します。この文字列については、オンライン・マニュアル『BEA Tuxedo CORBA クライアント・アプリケーションの開発方法』に構成案をいくつか記載しています。

`exceptionInfo`
エラーが発生した場合にアプリケーションが追加の例外データを取得できるようにするオプションの入力引数。

例外

`NoFactory`
この例外は、入力された `factory_key` または `factory_id` に対応するアプリケーション・ファクトリ・オブジェクト・リファレンスを FactoryFinder が見つけられない場合に発生します。

説明 このメンバ関数は、キーの `id` がメソッドの入力 `factory_id` に一致するアプリケーション・ファクトリ・オブジェクト・リファレンスのリストを返すように、FactoryFinder に指示します。が等価性の照合を行う必要があります。つまり、入力 `factory_id` が、アプリケーション・ファクトリのキーの <id, kind> ペアの各 `id` と完全に一致していなければなりません。

戻り値 アプリケーション・ファクトリへのインターフェイス・ポインタの配列を含むバリアントを返します。

DI.find_factories

概要 アプリケーション・ファクトリのリストを取得します。

MIDL マッピング

```
HRESULT find_factories(
  [in] VARIANT factory_key,
  [in,out,optional] VARIANT* exceptionInfo,
  [out,retval] VARIANT* returnValue);
```

オートメーション・マッピング

```
Function find_factories(factory_key, [exceptionInfo])
```

パラメータ

`factory_key`

このパラメータは、ファクトリ・オブジェクト・リファレンスを一意に識別する DICosNaming_NameComponents (<id, kind> 値のペア) のセーフ配列を含みます。

exceptionInfo

エラーが発生した場合にアプリケーションが追加の例外データを取得できるようにするオプションの入力引数。

例外

`NoFactory`

この例外は、入力された `factory_key` に対応するアプリケーション・ファクトリ・オブジェクト・リファレンスを FactoryFinder が見つけられない場合に発生します。

説明 `find_factories` メソッドは、キーがメソッドの入力キーと一致するサーバ・アプリケーション・ファクトリ・オブジェクト・リファレンスのリストを返すように、FactoryFinder に指示します。BEA Tuxedo システムでは、等価性の照合が行われることを想定しています。つまり、<id,kind> ペアの 2 つのシーケンス (入力キーに対応するものと、アプリケーション・ファクトリのキーに含まれるもの) のそれぞれが同じ長さであり、一方のシーケンスにあるペアはすべて、もう一方のペアにも同一のものが存在します。

戻り値 アプリケーション・ファクトリへのインターフェイス・ポインタの配列を含むバリアントを返します。

DI.list_factories

概要 アプリケーション・ファクトリ名とオブジェクト・リファレンスをすべてリストします。

MIDL マッピング

```
HRESULT list_factories(
  [in,out,optional] VARIANT* exceptionInfo,
  [out,retval] VARIANT* returnValue);
```

オートメーション・マッピング

```
Function list_factories([exceptionInfo])
```

パラメータ

`exceptionInfo`

エラーが発生した場合にアプリケーションが追加の例外データを取得できるようにするオプションの入力引数。

例外

`CannotProceed`

この例外が発生するのは、FactoryFinder または CORBA サービス・ネーミング・サービスが検索中に内部エラーに遭遇した場合です。エラーは、ユーザ・ログ (ULOG) に書き込まれます。この例外が発生した場合は、すぐに作業担当者に通知されます。内部エラーの重要度によっては、FactoryFinder または CORBA サービス・ネーミング・サービスが実行されていたサーバが、終了していることがあります。FactoryFinder サービスが終了した場合は、新しい FactoryFinder サービスを開始します。CORBA サービス・ネーミング・サービスの 1 つが終了しており、別の CORBA サービス・ネーミング・サービス は実行中である場合は、新規の CORBA サービス・ネーミング・サービス を開始します。稼動中のネーミング・サービス・サーバがない場合は、アプリケーションを再起動します。

`RegistrarNotAvailable`

この例外は、FactoryFinder が CORBA サービス・ネーミング・サービスのオブジェクトを見つけられない場合に発生します。この例外が発生した場合は、すぐに作業担当者に通知されます。稼動中のネーミング・サービス・サーバがない場合もあります。その場合は、アプリケーションを再起動します。

説明 このメソッドは、CORBA サービス・ネーミング・サービスで登録されているアプリケーション・ファクトリのファクトリ・キー、および関連するオブジェクト・リファレンスをすべて含むリストを返すよう、FactoryFinder に指示します。

戻り値 DITobj_FactoryComponent オブジェクトの配列を含むバリアントを返します。FactoryComponent オブジェクトは、DICosNaming_NameComponent オブジェクトの配列と、アプリケーション・ファクトリへのインターフェイス・ポインタを含むバリアントで構成されます。

プログラミング例

この節では、FactoryFinder インターフェイスを使用してプログラミングを行う方法について説明します。

注記　コード内の例外を確認するよう留意してください。

FactoryFinder オブジェクトの使用

ファクトリ・オブジェクトへのリファレンスを見つけるには、プログラマは FactoryFinder オブジェクトを使用します。FactoryFinder オブジェクトは、指定した基準に基づき、ファクトリ・オブジェクトへの 1 つまたは複数のリファレンスを取得するオペレーションを提供します。

プロセス・アドレス領域には、複数の FactoryFinder オブジェクトが存在できます。FactoryFinder オブジェクトへの複数リファレンスをサポートする必要があります。FactoryFinder オブジェクトは、ドメイン内の FactoryFinder オブジェクトと、ドメインへのアクセスに使用する特定の IIOP サーバ・リスナ / ハンドラ (ISL/ISH) の間の関係についての状態を維持するという点では、半ば状態を持つオブジェクトです。

Object Management Group の CORBA サービス仕様 (1997 年 12 月) の第 6 章「Life Cycle Service」で定義されているように、FactoryFinder オブジェクトはすべて `CosLifeCycle::FactoryFinder` インターフェイスをサポートしています。このインターフェイスは、指定した基準に合致するファクトリ・オブジェクトへの 1 つまたは複数のリファレンスを取得するために使用されるオペレーションを 1 つ含みます。

ファクトリ・オブジェクトへのリファレンスの登録

次のコードの抜粋 (リスト 5-8) では、TP フレームワーク・インターフェイスを使用して、FactoryFinder でファクトリ・オブジェクトへのリファレンスを登録する方法を示します。

コードリスト 5-8 サーバ・アプリケーション : ファクトリの登録

```
// サーバ・アプリケーション： ファクトリの登録
// C++ の例
TP::register_factory( factory_obj.in( ), “TellerFactory” );
```

CosLifeCycle::FactoryFinder インターフェイスを使用した FactoryFinder オブジェクトへのリファレンスの取得

次のコードの抜粋 (リスト 5-9) は、CORBA 準拠のインターフェイスを使用してファクトリ・オブジェクトへの 1 つまたは複数のリファレンスを取得する方法を示します。

コードリスト 5-9 クライアント・アプリケーション : FactoryFinder オブジェクト・リファレンスの取得

```
// クライアント・アプリケーション： ファクトリ・オブジェクトへの
// オブジェクト・リファレンスの取得
CosLifeCycle::Key_var  factory_key = new CosLifeCycle::Key( );
factory_key ->length(1);
factory_key[0].id = string_dupalloc( “strlen(“TellerFactory”) +1 );
factory_key[0].kind = string_dupalloc(
                                strlen(““FactoryInterface”) + 1);
strcpy( factory_key[0].id, ““TellerFactory” );
strcpy( facory_key[0].kind, “FactoryInterface” );
CosLifeCycle::Factories_var *  flp = ff_np ->
                               find_factories( factory_key.in( ) );
```

拡張 Bootstrap オブジェクトを使用した FactoryFinder オブジェクトへのリファレンスの取得

次のコードの抜粋 (リスト 5-10) は、BEA Tuxedo 拡張 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。

コードリスト 5-10 **クライアント・アプリケーション** : Tobj **の手法を使用して、**1 **つのファクトリを見つける**

```
// クライアント・アプリケーション： Tobj の手法を使用して、
// 1 つのファクトリを見つける

Tobj_Bootstrap *  bsp = new Tobj_Bootstrap(
                                        orb_ptr.in( ), host_port );
CORBA::Object_varptr  ff_op = bsp ->
                    resolve_initial_references( “FactoryFinder” );
Tobj::FactoryFinder_ptrvar   ff_np =
                             Tobj::FactoryFinder::_narrow( ff_op );
```

注記　Tobj_Bootstrap オブジェクトを使用する場合は、CosLifeCycle::FactoryFinder インターフェイスに BEA Tuxedo CORBA の拡張を使用できますが、このオブジェクトはファクトリを見つけるために必須のものではありません。CORBA INS を使用していれば、CosLifeCycle::FactoryFinder インターフェイスにより提供される find_factories() メソッドを使用できます。

FactoryFinder オブジェクトに対する拡張の使用

BEA Tuxedo では、CORBA で定義されるオペレーションと類似の機能をサポートするように、FactoryFinder オブジェクトを拡張します。ただしこちらでは、大幅に簡素化されており、より限定的であるシグニチャが使われます。拡張機能は、Tobj::FactoryFinder インターフェイスを定義することによって提供されます。Tobj::FactoryFinder インターフェイス用に定義されたオペレーションは、CORBA で定義される同等の機能を、焦点を絞って簡素化した形態で提供することを意図しています。アプリケーション開発者は、開発時に、CORBA 定義の拡張を使用するか、BEA Tuxedo の拡張を使用するかを選択できます。インターフェイス Tobj::FactoryFinder は、CosLifeCycle::FactoryFinder インターフェイスから派生しています。

FactoryFinder オブジェクトへの BEA Tuxedo の拡張は、Object Management Group の CORBA サービス仕様 (1997 年 12 月) の第 6 章「Life Cycle Service」で定義されている FactoryFinder オブジェクトとすべて同じ規則に準拠しています。

拡張された FactoryFinder オブジェクトのインプリメンテーションでは、ユーザが CORBA 定義の `CosLifeCycle::FactoryFinder` インターフェイスでのように `CosLifeCycle::Key` を指定するか、または探しているファクトリ・オブジェクトの識別子を含んだ NULL で終了する文字列を指定する必要があります。

Tobj::FactoryFinder を使用しての、1 つのファクトリの取得

次のコードの抜粋 (リスト 5-11) では、BEA Tuxedo 拡張インターフェイスを使用して、識別子に基づき、ファクトリ・オブジェクトへのリファレンスを 1 つ取得する方法を示します。

コードリスト 5-11 **クライアント・アプリケーション** : BEA Tuxedo **拡張の手法をを使用して、**1 **つのファクトリを見つける**

```
CosLifeCycle::Factory_ptrvar  fp_obj = ff_np ->
                        find_one_factory_by_id( “TellerFactory” );
```

Tobj::FactoryFinder を使用しての、1 つまたは複数のファクトリの取得

次のコードの抜粋 (リスト 5-12) では、BEA Tuxedo 拡張を使用して、識別子に基づき、ファクトリ・オブジェクトへの 1 つまたは複数のリファレンスを取得する方法を示します。

コードリスト 5-12 **クライアント・アプリケーション** : BEA Tuxedo **拡張の手法を使用して、**1 **つまたは複数のファクトリを見つける**

```
CosLifeCycle::Factories * _var  flp = ff_np ->
                          find_factories_by_id( “TellerFactory” );
```

章

6 セキュリティ・サービス

セキュリティの詳細については、『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。このマニュアルでは、暗号化技術など BEA Tuxedo のセキュリティ機能に関連する概念を紹介し、このセキュリティ機能によって BEA Tuxedo アプリケーションのセキュリティを高める方法を説明します。また、セキュリティ・サービスで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) の使い方を解説します。

『BEA Tuxedo CORBA アプリケーションのセキュリティ機能』は、オンライン・マニュアルとして PDF でも提供しています。

章

7 トランザクション・サービス

トランザクションの詳細については、『BEA Tuxedo CORBA トランザクション』を参照してください。このマニュアルには、トランザクションの概要、アプリケーション・プログラミング・インターフェイス (API) の説明、およびアプリケーション・プログラミング・インターフェイス (API) を使ってアプリケーションを開発する方法が記載されています。

『BEA Tuxedo CORBA トランザクション』は、オンライン・マニュアルとして PDF でも提供しています。

章

8 ノーティフィケーション・サービス

ノーティフィケーション・サービスの詳細については、『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。このマニュアルには、ノーティフィケーション・サービスの概要、アプリケーション・プログラミング・インターフェイス (API) の説明、および API を使用してアプリケーションを開発する方法が記載されています。

『BEA Tuxedo CORBA ノーティフィケーション・サービス』は、オンライン・マニュアルとして PDF でも提供しています。

9 要求レベルのインターセプタ

要求レベルのインターセプタの詳細については、『BEA Tuxedo CORBA 要求レベルのインターセプタ』を参照してください。このマニュアルには、要求レベルのインターセプタの概要、アプリケーション・プログラミング・インターフェイス (API) の説明、および API を使用して要求レベルのインターセプタをインプリメントする方法が記載されています。

『BEA Tuxedo CORBA 要求レベルのインターセプタ』は、オンライン・マニュアルとして PDF でも提供しています。

章

10 CORBA インターフェイス・リポジトリのインターフェイス

この章では、BEA Tuxedo CORBA インターフェイス・リポジトリのインターフェイスについて説明します。

注記　この章に記載されている情報の大部分は、「Common Object Request Broker: Architecture and Specification, Revision 2.4.2」(2001 年 2 月) の第 10 章からの引用です OMG 情報は、インターフェイス・リポジトリのインターフェイスの BEA Tuxedo CORBA でのインプリメンテーションを説明するために、必要に応じて変更されています。使用にあたっては OMG の許可を得ています。

BEA Tuxedo CORBA インターフェイス・リポジトリには、BEA Tuxedo ドメイン内でインプリメントされる CORBA オブジェクトのインターフェイス記述が含まれています。

インターフェイス・リポジトリは、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。これにより、CORBA で定義されるインターフェイスの適切なサブセットが提供されます。つまり、プログラマが利用できる API は、「Common Object Request Broker: Architecture and Specification Revision 2.4」での定義に従ってインプリメントされます。ただ

し、すべてのインターフェイスがサポートされるわけではありません。一般に、インターフェイス・リポジトリからの読み取りに必要なインターフェイスはサポートされますが、インターフェイス・リポジトリへの書き込みに必要なインターフェイスはサポートされません。また、すべての TypeCode インターフェイスがサポートされるわけではありません。

インターフェイス・リポジトリの管理は、BEA Tuxedo ソフトウェア固有のツールを使用して行われます。システム管理者はこれらのツールを使用して、インターフェイス・リポジトリを作成し、Object Management Group インターフェイス定義言語 (OMG IDL) で指定された定義を挿入し、インターフェイスを削除することができます。加えて、管理者はインターフェイス・リポジトリ・サーバが含まれるようにシステムをコンフィギュレーションしなければならない場合があります。インターフェイス・リポジトリの管理コマンドについては、『BEA Tuxedo コマンド・リファレンス』および『BEA Tuxedo アプリケーションの設定』を参照してください。

いくつかの抽象インターフェイスが、インターフェイス・リポジトリのほかのオブジェクト用の基底インターフェイスとして使用されます。インターフェイス・リポジトリ内のオブジェクトを見つけるには、共通のオペレーションのセットを使用します。これらのオペレーションは、この章で説明する抽象インターフェイス IRObject、Container、および Contained で定義されます。インターフェイス・リポジトリのオブジェクトはすべて IRObject インターフェイスから継承されます。これは、オブジェクトの実際の型を識別するオペレーションを提供するインターフェイスです。コンテナであるオブジェクトは、Container インターフェイスからナビゲーション・オペレーションを継承します。ほかのオブジェクトに包含されるオブジェクトは、ナビゲーション・オペレーションを Contained インターフェイスから継承します。IDLType インターフェイスは、インターフェイス、typedef、および無名の型など OMG IDL の型を表す、すべてのインターフェイス・リポジトリ・オブジェクトから継承されます。TypedefDef インターフェイスは、名前の付いたすべての非インターフェイス型から継承されます。

IRObject、Contained、Container、IDLType、および TypedefDef インターフェイスは、インスタンス化できません。

インターフェイス・リポジトリ内の文字列データはすべて、ISO 8859-1 文字セットの定義に従って符号化されます。

注記　BEA Tuxedo ソフトウェアは、インターフェイス・リポジトリへの読み取り権限のみをサポートしているため、この章では Write インターフェイスには言及していません。インターフェイス・リポジトリに対して Write インターフェイスを使用しようとすると、`CORBA::NO_IMPLEMENT` 例外が発生します。

構造と使用方法

インターフェイス・リポジトリは、データベースとサーバという、2 つの異なったコンポーネントで構成されています。サーバは、データベースに対してオペレーションを実行します。

インターフェイス・リポジトリのデータベースは、`idl2ir` 管理コマンドを使用して作成され、追加されます。このコマンドについては、『BEA Tuxedo コマンド・リファレンス』および『BEA Tuxedo アプリケーションの設定』を参照してください。プログラマ側からは、インターフェイス・リポジトリに対する書き込み権限はありません。CORBA で定義されている書き込みオペレーションや、読み取り専用でない属性に対する設定オペレーションで、サポートされているものはありません。

インターフェイス・リポジトリ・データベースの読み取り権限は、常にインターフェイス・リポジトリ・サーバから与えられます。つまり、クライアントはサーバで実行されたメソッドを呼び出すことによって、データからの読み取りを行います。「CORBA Common Object Request Broker: Architecture and Specification, Revision 2.4」で定義されている読み取りオペレーションについては、この章で説明します。

プログラミング情報

サーバへのインターフェイスは、OMG IDL ファイルで定義します。OMG IDL ファイルへのアクセス方法は、ビルドされているクライアントの種類に応じて変わります。ここではスタブ・ベース、動的起動インターフェイス (DII)、および ActiveX の 3 種類のクライアントについて検討します。

スタブ・スタイルの呼び出しを使用するクライアント・アプリケーションでは、ビルド時に OMG IDL ファイルが必要です。プログラマは OMG IDL ファイルを使用して、スタブなどを生成できます。詳細については、『BEA Tuxedo CORBA クライアント・アプリケーションの開発方法』を参照してください。インターフェイス・リポジトリに対する、これ以外のアクセスは必要ありません。

動的起動インターフェイス (DII) を使用するクライアント・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・リポジトリへのインターフェイスの定義はこの章に記載しています。第 10 章の 6 ページ「クライアント・アプリケーションのビルド」に説明があります。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、クライアントが特定のオブジェクト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。クライアントが特定のオブジェクトに関する情報を得るには、`CORBA::Object::_get_interface` メソッドを使用して InterfaceDef オブジェクトを取得します。このメソッドの記述については、「`CORBA::Object::_get_interface`」を参照してください。InterfaceDef オブジェクトを使用して、クライアントはインターフェイスに関する完全な情報を取得できます。

DII クライアントがインターフェイス・リポジトリを参照できるようになるには、検索を開始するためのインターフェイス・リポジトリのオブジェクト・リファレンスを取得する必要があります。

DII クライアントは、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。このメソッドについては、「Tobj_Bootstrap::register_callback_port」を参照してください。クライアントはオブジェクト・リファレンスを取得すると、インターフェイス・リポジトリ内をルートから参照できるようになります。

クライアント・アプリケーションが関連付けられている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を取得するには、クライアント・アプリケーションは次の 2 つのブートストラップ処理メカニズムのうち、どちらかを使用します。

- `"CORBA::Repository"` の値を持つ `Tobj_Bootstrap::resolve_initial_references` オペレーションを呼び出します。このオペレーションは、クライアント・アプリケーションが現在付属しているドメイン内にある InterfaceRepository オブジェクトへ

のリファレンスを返します。BEA Tuxedo クライアント・ソフトウェアを使用している場合は、このメカニズムを使用します。詳細については、Tobj_Bootstrap::resolve_initial_references を参照してください。

- `"CORBA::Repository"` の値を持つ `CORBA::ORB::resolve_initial_references` オペレーションを呼び出します。このオペレーションは、クライアント・アプリケーションが現在付属しているドメイン内にある InterfaceRepository オブジェクトへのリファレンスを返します。サード・パーティ製のクライアント ORB を使用している場合は、このメカニズムを使用します。詳細については、CORBA::ORB::resolve_initial_references を参照してください。

注記　DII を使用するには、OMG IDL ファイルがインターフェイス・リポジトリ内に格納されている必要があります。

ActiveX を使用するクライアント・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。インターフェイス・リポジトリ側から見れば、Active X クライアントと DII クライアントの間に差はありません。ActiveX クライアントには、Visual Basic コードによる Bootstrap オブジェクトが含まれます。DII クライアントと同様に、ActiveX クライアントは Bootstrap オブジェクトを使用してインターフェイス・リポジトリのオブジェクト・リファレンスを取得します。このメソッドについては、「Tobj_Bootstrap::resolve_initial_references」を参照してください。クライアントはオブジェクト・リファレンスを取得すると、インターフェイス・リポジトリ内をルートから参照できるようになります。

注記　ActiveX クライアントを使用するには、OMG IDL ファイルがインターフェイス・リポジトリ内に格納されている必要があります。

パフォーマンスへの影響

インターフェイス・リポジトリへの実行時アクセスはすべて、インターフェイス・リポジトリ・サーバを介して行われます。リモート・サーバ・アプリケーションの要求では、かなりのオーバーヘッドが生じるので、設計者はこの点を認識しておく必要があります。たとえば、オブジェクト・リファレン

スを使用して、オブジェクト・リファレンスに対する DII 呼び出しを行うのに必要な情報を取得するために要求される対話について以下で考察します。手順は、次のとおりです。

1. クライアント・アプリケーションが、`CORBA::Object` の `_get_interface` オペレーションを呼び出し、問題のオブジェクトに関連付けられた InterfaceDef オブジェクトを取得します。これにより、オブジェクト・リファレンスを作成した ORB にメッセージが送信されます。
2. ORB は、クライアントに InterfaceDef オブジェクトを返します。
3. クライアントはオブジェクトの 1 つまたは複数の `_is_a` オペレーションを呼び出し、このオブジェクトでサポートされているインターフェイスの型を判断します。
4. インターフェイスを識別すると、クライアントは Interface オブジェクトの `describe_interface` オペレーションを呼び出し、そのインターフェイスの完全な記述 (たとえばバージョン番号、属性、およびパラメータ) を取得します。これにより、インターフェイス・リポジトリにメッセージが送信され、応答が返ります。
5. これで、クライアントが DII 要求を構成する準備が整いました。

クライアント・アプリケーションのビルド

インターフェイス・リポジトリを使用するクライアントは、インターフェイス・リポジトリのスタブ内で、リンクしている必要があります。リンク方法は、ベンダごとに異なります。クライアント・アプリケーションが BEA Tuxedo ORB を使用している場合、BEA Tuxedo ソフトウェアはスタブをライブラリ形式で提供します。したがって、プログラマはスタブの構築に、インターフェイス・リポジトリ OMG IDL ファイルを使用する必要がありません。インターフェイス・リポジトリ定義は `CORBA.h` ファイルにありますが、デフォルトでは包含されていません。

注記　インターフェイス・リポジトリ定義を使用するには、クライアント・アプリケーション・コードに CORBA.h を包含する前に、ORB_INCLUDE_REPOSITORY マクロを定義する必要があります (例 : #Define ORB_INCLUDE_REPOSITORY)。

クライアント・アプリケーションがサード・パーティ製の ORB (たとえば ORBIX) を使用している場合、プログラマはそのベンダが提供しているメカニズムを使用しなければなりません。これには、ベンダが提供している IDL コンパイラを使用して OMG IDL ファイルからスタブを生成すること、または単にベンダが提供しているスタブに対するリンクを行うことなどのメカニズムが含まれます。

一部のサード・パーティ製 ORB には、ローカル・インターフェイス・リポジトリ機能が付属しています。この場合、ローカル・インターフェイス・リポジトリはそのベンダによって提供されたものであり、そのクライアントが必要とするインターフェイス定義が追加されています。

InterfaceRepository オブジェクトへの初期リファレンスの取得

InterfaceRepository オブジェクトへの初期リファレンスを取得するには、Bootstrap オブジェクトを使用します。Bootstrap オブジェクト・メソッドについては、「Tobj_Bootstrap::resolve_initial_references」を参照してください。

インターフェイス・リポジトリのインターフェイス

クライアント・アプリケーションは、インターフェイス・リポジトリへのアクセスに、CORBA で定義されたインターフェイスを使用します。この節では、BEA Tuxedo ソフトウェアでインプリメントされる各インターフェイスについて説明します。

注記　インターフェイス・リポジトリの BEA Tuxedo CORBA インプリメンテーションは、インターフェイスに対する読み取りオペレーションしかサポートしません。書き込みオペレーションは、インプリメントされません。

サポートしている型定義

いくつかの型が、インターフェイス・リポジトリのインターフェイス定義全体に使用されています。

```
module CORBA {
    typedef string              Identifier;
    typedef string              ScopedName;
    typedef string              RepositoryId;

    enum DefinitionKind {
       dk_none, dk_all,
       dk_Attribute, dk_Constant, dk_Exception, dk_Interface,
       dk_Module, dk_Operation, dk_Typedef,
       dk_Alias, dk_Struct, dk_Union, dk_Enum,
       dk_Primitive, dk_String, dk_Sequence, dk_Array,
       dk_Repository,
       dk_Wstring, dk_Fixed,
       dk_Value, dk_ValueBox, dk_ValueMember,
       dk_Native
       };
};
```

Identifiers は、モジュール、インターフェイス、値型、値メンバ、値ボックス、定数、typedef、例外、属性、オペレーション、およびネイティブ型を識別する、単純な名前です。ちょうど OMG IDL の識別子に対応します。Identifier は、インターフェイス・リポジトリ全体では必ずしも一意ではありません。特定の Repository、ModuleDef、InterfaceDef、ValueDef、または OperationDef 内でのみ、一意です。

ScopedName は、ダブル・コロン (::) で区切られた、1 つまたは複数の識別子で構成される名前です。OMG IDL のスコープ指定された名前に対応します。絶対 ScopedName とは、先頭にダブル・コロン (::) があり、Repository 内で定義を明確に識別するものを言います。Repository 内の絶対 ScopedName は、OMG IDL ファイルのグローバル名に対応します。相対 ScopedName は、先頭にダブル・コロン (::) がなく、何らかのコンテキストを基準として解決される必要があります。

RepositoryId は、モジュール、インターフェイス、値型、値メンバ、値ボックス、ネイティブ型、定数、typedef、例外、属性、またはオペレーションを一意かつグローバルに識別するために使用される識別子です。RepositoryId は文字列として定義されるため、言語バインディングの文字列操作ルーチンで操作 (たとえばコピーや比較) が可能です。

DefinitionKind は、インターフェイス・リポジトリ・オブジェクトの型を識別します。

IRObject インターフェイス

次に示す基底インターフェイス IRObject は、最も汎用性の高いインターフェイスを表します。ここから、Repository 自体を含む、インターフェイス・リポジトリのほかのすべてのインターフェイスが派正します。

```
module CORBA {
     interface IRObject {
            readonly attribute DefinitionKind def_kind;
      };
};
```

def_kind 属性は、定義の型を識別します。

Contained インターフェイス

次に示す Contained インターフェイスは、ほかのインターフェイス・リポジトリ・オブジェクトに包含されるインターフェイス・リポジトリのすべてのインターフェイスによって継承されます。ルート・オブジェクト (Repository) と無名定義 (ArrayDef、StringDef、SequenceDef) 以外の、インターフェイス・リポジトリ内のすべてのオブジェクト、およびプリミティブ型が、ほかのオブジェクトに包含されます。

```
module CORBA {
    typedef string VersionSpec;

    interface Contained : IRObject {
        readonly attribute RepositoryId    id;
        readonly attribute Identifier      name;
        readonly attribute VersionSpec     version;
        readonly attribute Container       defined_in;
        readonly attribute ScopedName      absolute_name;
        readonly attribute Repository      containing_repository;
        struct Description {
           DefinitionKind              kind;
           any                         value;
         };

        Description describe ();
        };
};
```

別のオブジェクトに包含されるオブジェクトには、それをグローバルに識別する `id` 属性と、包含 Container オブジェクト内で一意に識別する `name` 属性が付いています。また、同じ名前を持つ別バージョンのオブジェクトと区別するための `version` 属性も付いています。BEA Tuxedo CORBA インターフェイス・リポジトリは、同名オブジェクトの同時包含、または複数バージョンの存在をサポートしていません。

包含オブジェクトには、それらが定義される Container を識別する、`defined_in` 属性も付いています。オブジェクトが包含される理由は、包含するオブジェクト内で定義されるから、または包含するオブジェクトによって継承されるからです。前者の例は、インターフェイスがモジュール内で定義される場合などです。後者の例は、あるインターフェイスが別のインターフェイスからオペレーションを継承したことにより、そのオペレーションが

そのインターフェイス内に包含される場合などです。オブジェクトが継承によって包含された場合は、defined_in 属性によって、オブジェクトの継承元である InterfaceDef または ValueDef が識別されます。

absolute_name 属性は、包含 Repository 内で Contained オブジェクトを一意に識別する、絶対 ScopedName です。このオブジェクトの defined_in 属性が Repository を参照している場合、文字列「::」と、このオブジェクトの name 属性を結合することによって、absolute_name が形成されます。それ以外の場合、absolute_name の形成は、このオブジェクトの defined_in 属性によって参照されるオブジェクトの absolute_name 属性、文字列「::」、およびこのオブジェクトの name 属性を結合することによって行われます。

containing_repository 属性は、オブジェクトの defined_in 属性に再帰的に従うことによって最終的に到達される Repository を識別します。

within オペレーションは、オブジェクトを包含するオブジェクトのリストを返します。オブジェクトがインターフェイスまたはモジュールである場合は、それを定義するオブジェクトによってのみ包含可能です。その他のオブジェクトは、それを定義するオブジェクトにも、継承するオブジェクトにも、包含可能です。

describe オペレーションは、インターフェイスの情報が包含された構造体を返します。各インターフェイスと関連付けられた記述構造体は、インターフェイス定義の下に置かれます。返された構造体に記述される定義の種類は、返された構造体と共に提供されます。たとえば、describe オペレーションが属性オブジェクトに対して呼び出されると、kind フィールドに dk_Attribute が包含され、値フィールドに AttributeDescription 構造体を含む any が包含されます。

Container インターフェイス

基底インターフェイス Container は、インターフェイス・リポジトリ内に包含の階層構造を形成するために使用されます。Container は、Contained インターフェイスから派生した任意の数のオブジェクトを包含できます。Repository を除く Container はすべて、Contained からも派生します。

```
module CORBA {
    typedef sequence <Contained> ContainedSeq;
```

```
    interface Container : IRObject {
       Contained lookup (in ScopedName search_name);

       ContainedSeq contents (
           in DefinitionKind        limit_type,
           in boolean               exclude_inherited
            );

       ContainedSeq lookup_name (
           in Identifier            search_name,
           in long                  levels_to_search,
           in DefinitionKind        limit_type,
           in boolean               exclude_inherited
            );

        struct Description {
           Contained                contained_object;
           DefinitionKind           kind;
           any                      value;
         };

          typedef sequence<Description>  DescriptionSeq;

        DescriptionSeq describe_contents (
           in DefinitionKind        limit_type,
           in boolean               exclude_inherited,
           in long                  max_returned_objs
            );
      };
};
```

lookup オペレーションは、スコープ指定された名前を指定されて、このコンテナと関連する定義を見つけます。名前のスコープ指定には、OMG IDL 規則が使用されます。ダブル・コロン (::) が先頭に付く、絶対スコープ指定された名前は、包含 Repository と関連する定義を見つけます。そのようなオブジェクトが見つからなければ、ニル・オブジェクト・リファレンスが返されます。

contents オペレーションは、オブジェクトに直接包含または継承されるオブジェクトのリストを返します。このオペレーションは、オブジェクトの階層構造内を移動するのに使用されます。クライアントはこのオペレーションを使用し、Repository オブジェクトを先頭に、Repository に包含されているすべてのオブジェクト、リポジトリ内のモジュールに包含されているすべてのオブジェクト、特定モジュール内のすべてのインターフェイスおよび値型などを、リストします。

limit_type

dk_all に limit_type が設定されると、すべての型のオブジェクトが返されます。たとえばこれが InterfaceDef であれば、属性、オペレーション、および例外オブジェクトがすべて返されます。
limit_type が特定のインターフェイスに設定されると、その型のオブジェクトのみが返されます。
たとえば、limit_type が dk_Attribute に設定されると、属性オブジェクトのみが返されます。

exclude_inherited

TRUE に設定された継承オブジェクトは、存在したとしても、返されません。FALSE に設定されている場合、継承されたために包含されているオブジェクトも、オブジェクト内で定義されたために包含されているオブジェクトも、すべて返されます。
あるオブジェクトを、特定のオブジェクト内またはそのオブジェクトに包含されているオブジェクト内で名前を指定して見つけるには、lookup_name オペレーションを使用します。describe_contents オペレーションは、contents オペレーションと describe オペレーションを組み合わせます。contents オペレーションから返された各オブジェクトについて、そのオブジェクトの記述が返されます。つまり、そのオブジェクトの describe オペレーションが呼び出され、その結果が返されます。

あるオブジェクトを、特定のオブジェクト内またはそのオブジェクトに包含されているオブジェクト内で名前を指定して見つけるには、lookup_name オペレーションを使用します。

search_name

検索する名前を指定します。

levels_to_search

ルックアップの対象をオペレーションの呼び出し先のオブジェクトに限定するのか、そのオブジェクトに包含されている各オブジェクトを検索するのかを制御します。levels_to_search を -1 に設定すると、現在のオブジェクトと、すべての包含オブジェクトが検索されます。levels_to_search を 1 に設定すると、現在のオブジェクトのみが検索されます。0 または -1 以外の負数とした levels_to_search の値の使用は、未定義です。

describe_contents オペレーションは、contents オペレーションと describe オペレーションを組み合わせます。contents オペレーションから返された各オブジェクトについて、そのオブジェクトの記述が返されます。つまり、そのオブジェクトの describe オペレーションが呼び出され、その結果が返されます。

max_returned_objs
: 呼び出しで返されるオブジェクト数を、指定された数に制限します。パラメータを -1 に設定すると、包含オブジェクトがすべて返されます。

IDLType インターフェイス

次に示す基底インターフェイス IDLType は、OMG IDL の型を表すすべてのインターフェイス・リポジトリ・オブジェクトによって継承されます。これは型を記述する TypeCode へのアクセスを提供し、IDL 型定義への参照が必要な場合には常に、ほかのインターフェイスの定義に使用されます。

```
module CORBA {
       interface IDLType : IRObject {
             readonly attribute TypeCode          type;
       };
};
```

type 属性は、IDLType から派生したオブジェクトによって定義される型を記述します。

Repository インターフェイス

次に示す Repository は、インターフェイス・リポジトリへのグローバル・アクセスを提供するインターフェイスです。Repository オブジェクトには、定数、typedef、例外、インターフェイス、値型、値ボックス、ネイティブ型、およびモジュールを含めることができます。これは Container から継承されるので、name と id のどちらによっても、任意の定義の参照に使用できます。この定義は、グローバルな定義でも、モジュールまたはインターフェイス内での定義でもかまいません。

Repository は、Container からのみ派生するものであって、Contained からは派生しないので、RepositoryId が関連付けられることはありません。これはデフォルトでは、Repository オブジェクトに直接包含されている ModuleDef、InterfaceDef、ValueDef、ValueBoxDef、TypedefDef、ExceptionDef、および ConstantDef の記述構造体における defined_in フィールドに値を割り当てる目的で、RepositoryId "" (空の文字列) を備えていると見なされます。

```
module CORBA {
       interface Repository : Container {
            Contained lookup_id (in RepositoryId search_id);
            TypeCode get_canonical_typecode(in TypeCode tc);
            PrimitiveDef get_primitive (in PrimitiveKind kind);

       };
};
```

lookup_id オペレーションは、RepositoryId を指定して、Repository 内のオブジェクトを検索する際に使用します。Repository に search_id の定義が含まれない場合、ニル・オブジェクト・リファレンスが返されます。

get_canonical_typecode オペレーションは、インターフェイス・リポジトリ内の TypeCode を調べ、すべてのリポジトリの ID、名前、および member_names が含まれる同等な TypeCode を返します。配列およびシーケンスの TypeCode や以前の ORB からの TypeCode の場合など、最上位の TypeCode が RepositoryId を含まない場合、またはターゲット・リポジトリ内に見つからない RepositoryId を含む場合、元の TypeCode の各メンバ TypeCode に対して get_canonical_typecode を再帰的に呼び出すことによって、新規の TypeCode が構築されます。

get_primitive オペレーションは、指定した kind 属性を備えた PrimitiveDef へのリファレンスを返します。PrimitiveDefs はすべて不変のものであり、Repository によって所有されます。

ModuleDef インターフェイス

次に示す ModuleDef は、定数、typedef、例外、インターフェイス、値型、値ボックス、ネイティブ型、およびその他のモジュール・オブジェクトを包含することができます。

```
module CORBA {
       interface ModuleDef : Container, Contained {
       };

        struct ModuleDescription {
               Identifier          name;
               RepositoryId        id;
               RepositoryId        defined_in;
               VersionSpec         version;
         };

};
```

継承される ModuleDef オブジェクトの `describe` オペレーションは、ModuleDescription を返します。

ConstantDef インターフェイス

次に示す ConstantDef オブジェクトは、名前の付いた定数を定義します。

```
module CORBA {
       interface ConstantDef : Contained {
              readonly attribute TypeCode      type;
              readonly attribute IDLType       type_def;
              readonly attribute any           value;
       };

       struct ConstantDescription {
               Identifier        name;
               RepositoryId      id;
               RepositoryId      defined_in;
               VersionSpec       version;
               TypeCode          type;
               any               value;
       };
};
```

`type`
: 定数の型を記述する TypeCode を指定します。定数の型は、単純な型 (long、short、float、char、string、octet など) の 1 つとします。

`type_def`
: 定数の型定義を識別します。

value
: 定数の値が含まれます。値の計算 (たとえば、「1+2」と定義されたという事実) は含みません。

ConstantDef オブジェクトの describe オペレーションは、ConstantDescription を返します。

TypedefDef インターフェイス

次に示す TypedefDef は、すべての名前の付いた非オブジェクト型 (構造体、共用体、列挙、およびエイリアス) の基底インターフェイスとして使用される、抽象インターフェイスです。TypedefDef インターフェイスは、プリミティブまたは無名型の定義オブジェクトによっては継承されません。

```
module CORBA {
      interface TypedefDef : Contained, IDLType {
      };

      struct TypeDescription {
             Identifier              name;
             RepositoryId            id;
             RepositoryId            defined_in;
             VersionSpec             version;
             TypeCode                type;
       };
};
```

継承される TypedefDef から派生したインターフェイスのための describe オペレーションは、TypeDescription を返します。

StructDef

次に示す StructDef は、OMG IDL の構造体定義を表します。これには struct のメンバが包含されます。

```
module CORBA {
       struct StructMember {
              Identifier       name;
              TypeCode         type;
```

```
                IDLType          type_def;
        };
        typedef sequence <StructMember> StructMemberSeq;

        interface StructDef : TypedefDef, Container{
                readonly attribute StructMemberSeq      members;
        };
};
```

members 属性には、各構造体メンバの記述が包含されます。

継承される type 属性は、構造体を記述する tk_struct TypeCode です。

UnionDef

次に示す UnionDef は、OMG IDL の共用体定義を表します。これには共用体のメンバが包含されます。

```
module CORBA {
      struct UnionMember {
            Identifier    name;
            any           label;
            TypeCode      type;
            IDLType       type_def;
      };
      typedef sequence <UnionMember> UnionMemberSeq;

      interface UnionDef : TypedefDef, Container {
          readonly   attribute TypeCode    discriminator_type;
          readonly   attribute IDLType     discriminator_type_def;
          readonly   attribute UnionMemberSeq   members;
      };
};
```

discriminator_type および discriminator_type_def
: 共用体の区別子型を記述および識別します。

members
: 各共用体メンバの記述が包含されます。各 UnionMemberDescription のラベルは、discriminator_type のそれぞれ異なる値です。隣接するメンバは、同じ名前を持つことができます。同名のメンバ同士は、型も同じである必要があります。octet 型で値が 0 のラベルは、デフォルトの共用体メンバであることを示します。

継承される type 属性は、共用体を記述する tk_union TypeCode です。

EnumDef

次に示す EnumDef は、OMG IDL の列挙の定義を表します。

```
module CORBA {
      typedef sequence <Identifier> EnumMemberSeq;

      interface EnumDef : TypedefDef {
             readonly attribute EnumMemberSeq         members;
      };
};
```

members
　列挙値の各々として考えられる個別の名前が包含されます。

継承される type 属性は、列挙を記述する tk_enum TypeCode です。

AliasDef

次に示す AliasDef は、ほかの定義のエイリアスを行う OMG IDL の typedef を表します。

```
module CORBA {
      interface AliasDef : TypedefDef {
            readonly attribute IDLType original_type_def;
      };
};
```

original_type_def
　エイリアスの対象となる型を識別します。

継承される type 属性は、エイリアスを記述する tk_alias TypeCode です。

PrimitiveDef

次に示す PrimitiveDef は、OMG IDL のプリミティブ型の 1 つを表します。プリミティブ型には名前が付かないので、このインターフェイスは TypedefDef からも Contained からも派生しません。

```
module CORBA {
  enum PrimitiveKind {
    pk_null, pk_void, pk_short, pk_long, pk_ushort, pk_ulong,
    pk_float, pk_double, pk_boolean, pk_char, pk_octet,
    pk_any, pk_TypeCode, pk_Principal, pk_string, pk_objref,
    pk_longlong, pk_ulonglong, pk_longdouble, pk_wchar, pk_wstring,
    pk_value_base
  };

  interface PrimitiveDef: IDLType {
      readonly attribute PrimitiveKind        kind;
  };
};
```

kind

PrimitiveDef で表されるプリミティブ型を示します。種類が pk_null の PrimitiveDef はありません。種類が pk_string の PrimitiveDef は、アンバウンディッド文字列を表します。
種類が pk_objref の PrimitiveDef は、OMG IDL の Object 型を表します。種類が pk_value_base の PrimitiveDef は、OMG IDL の ValueBase 型を表します。

継承される type 属性は、プリミティブ型を記述します。

PrimitiveDefs はすべて Repository によって所有されます。これらに対するリファレンスは、Repository::get_primitive を使用して取得されます。

StringDef

StringDef は、IDL のバインドされた文字列型を表します。アンバウンディッド文字列型は、PrimitiveDef として表されます。文字列型は無名なので、このインターフェイスは TypedefDef からも Contained からも派生しません。

```
module CORBA {
   interface StringDef : IDLType {
        attribute unsigned long bound;
   };
};
```

bound 属性は、文字列内の最大文字数を指定します。0 にはできません。

継承される type 属性は、文字列を記述する tk_string TypeCode です。

WstringDef

WstringDef は、IDL のワイド文字列を表します。バインドされていないワイド文字列型は、PrimitiveDef として表されます。ワイド文字列型は無名なので、このインターフェイスは TypedefDef からも Contained からも派生しません。

```
module CORBA {
   interface WstringDef : IDLType {
        attribute unsigned long bound;
   };
};
```

bound 属性は、ワイド文字列内のワイド文字の最大数を指定します。0 にはできません。

継承される type 属性は、ワイド文字列を記述する tk_wstring TypeCode です。

ExceptionDef

次に示す ExceptionDef は、例外定義を表します。構造体、共用体、および enum を包含することができます。

```
module CORBA {
     interface ExceptionDef : Contained, Container {
          readonly   attribute TypeCode              type;
          readonly   attribute StructMemberSeq       members;
     };
```

```
    struct ExceptionDescription {
        Identifier              name;
        RepositoryId            id;
        RepositoryId            defined_in;
        VersionSpec             version;
        TypeCode                type;
    };
};
```

type
　例外を記述する tk_except TypeCode です。

members
　任意の例外メンバを記述します。

ExceptionDef オブジェクトの describe オペレーションは、ExceptionDescription を返します。

AttributeDef

次に示す AttributeDef は、インターフェイスの属性を定義する情報を表します。

```
module CORBA {
      enum AttributeMode {ATTR_NORMAL, ATTR_READONLY};

      interface AttributeDef : Contained {
          readonly      attribute TypeCode          type;
                        attribute IDLType           type_def;
                        attribute AttributeMode     mode;
      };

      struct AttributeDescription {
          Identifier        name;
          RepositoryId      id;
          RepositoryId      defined_in;
          VersionSpec       version;
          TypeCode          type;
          AttributeMode     mode;
      };
};
```

type
　この属性のの型を記述する TypeCode を指定します。

type_def
: この属性の型を定義するオブジェクトを識別します。

mode
: この属性を読み取り専用に指定するか、または読み取り / 書き込みの権限を指定します。

AttributeDef オブジェクトの describe オペレーションは、AttributeDescription を返します。

OperationDef

次に示す OperationDef は、インターフェイスのオペレーションを定義するのに必要な情報を表します。

```
module CORBA {
      enum OperationMode {OP_NORMAL, OP_ONEWAY};

      enum ParameterMode {PARAM_IN, PARAM_OUT, PARAM_INOUT};
      struct ParameterDescription {
            Identifier            name;
            TypeCode              type;
            IDLType               type_def;
            ParameterMode         mode;
      };
      typedef sequence <ParameterDescription> ParDescriptionSeq;

      typedef Identifier ContextIdentifier;
      typedef sequence <ContextIdentifier> ContextIdSeq;

      typedef sequence <ExceptionDef> ExceptionDefSeq;
      typedef sequence <ExceptionDescription> ExcDescriptionSeq;

      interface OperationDef : Contained {
           readonly    attribute TypeCode              result;
           readonly    attribute IDLType               result_def;
           readonly    attribute ParDescriptionSeq     params;
           readonly    attribute OperationMode         mode;
           readonly    attribute ContextIdSeq          contexts;
           readonly    attribute ExceptionDefSeq       exceptions;
      };

      struct OperationDescription {
          Identifier                 name;
```

```
            RepositoryId                id;
            RepositoryId                defined_in;
            VersionSpec                 version;
            TypeCode                    result;
            OperationMode               mode;
            ContextIdSeq                contexts;
            ParDescriptionSeq           parameters;
            ExcDescriptionSeq           exceptions;
        };
};
```

`result`
: オペレーションの戻り値の型を記述する TypeCode です。

`result_def`
: 戻り値の型の定義を識別します。

`params`
: オペレーションのパラメータを記述します。ParameterDescription 構造体のシーケンスです。シーケンス中の ParameterDescriptions の順序は重要です。各構造体の `name` メンバは、パラメータ名を指定します。`type` メンバは、パラメータの型を記述する TypeCode です。`type_def` メンバは、パラメータの型の定義を識別します。`mode` メンバは、パラメータが in パラメータ、out パラメータ、または inout パラメータのうちどれであるのかを示します。

`mode`
: オペレーションの `mode` は、oneway であるか、normal であるかのどちらかです。oneway の場合、出力は返りません。

`contexts`
: オペレーションに適用されるコンテキスト識別子のリストを指定します。

例外
: オペレーションによって生じる可能性のある例外の型のリストを指定します。

継承される OperationDef オブジェクトの `describe` オペレーションは、OperationDescription を返します。

継承される describe_contents オペレーションは、このオペレーションについて定義されている各パラメータの記述をはじめとする、このオペレーションの完全な記述を提供します。

InterfaceDef

次に示す InterfaceDef オブジェクトは、インターフェイス定義を表します。定数、typedef、例外、オペレーション、および属性を包含することができます。

```
module CORBA {
   interface InterfaceDef;
      typedef sequence <InterfaceDef> InterfaceDefSeq;
      typedef sequence <RepositoryId> RepositoryIdSeq;
      typedef sequence <OperationDescription> OpDescriptionSeq;
      typedef sequence <AttributeDescription> AttrDescriptionSeq;

      interface InterfaceDef : Container, Contained, IDLType {

           readonly attribute InterfaceDefSeq   base_interfaces;
           readonly attribute boolean            is_abstract;

           boolean is_a (in RepositoryId interface_id);

          struct FullInterfaceDescription {
              Identifier              name;
              RepositoryId            id;
              RepositoryId            defined_in;
              VersionSpec             version;
              OpDescriptionSeq        operations;
              AttrDescriptionSeq      attributes;
              RepositoryIdSeq         base_interfaces;
              TypeCode                type;
              boolean                 is_abstract;
          };

           FullInterfaceDescription describe_interface();

      };

      struct InterfaceDescription {
           Identifier                name;
           RepositoryId              id;
           RepositoryId              defined_in;
           VersionSpec               version;
```

```
            RepositoryIdSeq             base_interfaces;
            boolean                     is_abstract;
        };
};
```

base_interfaces 属性は、このインターフェイスの継承元であるすべてのインターフェイスをリストします。

インターフェイスが抽象インターフェイス型である場合、is_abstract 属性は TRUE です。

is_a オペレーションは、呼び出し先のインターフェイスが、interface_id パラメータによって識別されたインターフェイスと同一であるか、このインターフェイスから直接的または間接的に継承したものである場合は、TRUE を返します。それ以外の場合には、FALSE を返します。

describe_interface オペレーションは、インターフェイスを記述する FullInterfaceDescription を返します。これには、オペレーションや属性も含まれています。FullInterfaceDescription 構造体のオペレーションおよび属性の各フィールドには、記述されているインターフェイスの継承グラフの推移閉包におけるすべてのオペレーションと属性が含まれます。

継承される InterfaceDef の describe オペレーションは、InterfaceDescription を返します。

継承される contents オペレーションは、この InterfaceDef で定義される定数、typedef、および例外のリストと、この InterfaceDef で定義または継承される属性およびオペレーションのリストを返します。exclude_inherited パラメータが TRUE に設定されていれば、このインターフェイス内で定義された属性とオペレーションのみが返されます。exclude_inherited パラメータが FALSE に設定されていれば、すべての属性とオペレーションが返されます。

章

11 共同クライアント / サーバ

ここでは、以下の内容について説明します。

- 「はじめに」では，次の内容について説明します。
  - メイン・プログラムおよびサーバの初期化
  - サーバント
  - スケルトンからのサーバントの継承
  - サポートされているコールバック・オブジェクト・モデル
  - リモート共同クライアント / サーバ・オブジェクトを呼び出すためのサーバのコンフィギュレーション
  - CORBA を使用してのコールバック・オブジェクトの準備 (C++ 共同クライアント / サーバのみ)
  - BEAWrapper Callbacks を使用してのコールバック・オブジェクトの準備
  - Java 共同クライアント / サーバのプログラミング上の考慮事項
- C++ BEAWrapper Callbacks インターフェイス API
- Java BEAWrapper Callbacks インターフェイス API

この章では、CORBA 共同クライアント / サーバおよび C++ BEAWrapper Callbacks API のプログラミング要件を説明します。Java BEAWrapper パッケージおよび `Java Callbacks` インターフェイス API については、Javadoc API を参照してください。

はじめに

プログラマは、BEA Tuxedo の CORBA クライアントと共同クライアント / サーバ (オブジェクト呼び出しの受信と処理が可能なクライアント) のどちらかに合わせて、クライアントの `main()` を記述します。`main()` は BEA Tuxedo CORBA 環境オブジェクトを使用して接続を確立し、セキュリティを設定し、トランザクションを開始します。

BEA Tuxedo クライアントは、オブジェクトのオペレーションを呼び出します。DII の場合、クライアント・コードによって DII Request オブジェクトが作成され、その DII Request の 2 つのオペレーションのうち一方が呼び出されます。静的呼び出しの場合は、クライアント・コードによって、通常の呼び出しに似た呼び出しが実行されます。これは結果的には、生成されたクライアント・スタブ内のコードを呼び出すことになります。加えて、クライアントのプログラマは OMG によって定義される ORB インターフェイスと、BEA Tuxedo ソフトウェアと共に供給される BEA Tuxedo CORBA 環境オブジェクトを使用して、BEA Tuxedo 固有の機能を実行します。

BEA Tuxedo 共同クライアント / サーバ・アプリケーションでは、クライアント・コードはコールバック BEA Tuxedo オブジェクト用のサーバとして動作できるように構成する必要があります。そのようなクライアントは TP フレームワークを使用しません。また、BEA Tuxedo システムの管理下には置かれません。これはプログラミングに影響を及ぼすほか、CORBA 共同クライアント / サーバには BEA Tuxedo CORBA サーバと同じようなスケーラビリティや信頼性がなく、また TP フレームワークで使用できるような状態管理機能やトランザクション機能もないということを意味します。これらの特性を必要とするユーザは、BEA Tuxedo CORBA のクライアントではなくサーバに、オブジェクトをインプリメントするようアプリケーションを構成する必要があります。

以下の節では、BEA Tuxedo クライアントにコールバックのサポートを追加するために使用するメカニズムを説明します。場合によっては、このメカニズムは TP フレームワークを使用する BEA Tuxedo サーバのメカニズムと対比されます。

メイン・プログラムおよびサーバの初期化

BEA Tuxedo サーバでは、buildobjserver コマンドを使用して、C++ サーバ用のメイン・プログラムを作成します。BEA Tuxedo のリリース 8.0 以降では、Java サーバはサポートされていません。サーバのメイン・プログラムは、サーバ機能における BEA Tuxedo および CORBA 関連の初期化を担当します。しかし、Server オブジェクトのインプリメントを行うので、サーバ・アプリケーションの初期化とシャットダウンの方法をカスタマイズする機会があります。サーバのメイン・プログラムは、Server オブジェクトに対し、適切な時点で自動的にメソッドを呼び出します。

対照的に、BEA Tuxedo CORBA 共同クライアント / サーバでは、BEA Tuxedo CORBA クライアントと同様に、ユーザがメイン・プログラムを作成し、すべての初期化を担当します。メイン・プログラムの完全な制御権があり、都合の良いように初期化およびシャットダウンのコードを指定することができるため、Server オブジェクトをインプリメントする必要はありません。

共同クライアント / サーバのみで必要な初期化については、第 11 章の -3 ページ「サーバント」で説明します。

サーバント

共同クライアント / サーバ用のサーバント (メソッド・コード) は、サーバ用のサーバントに非常に似ています。すべてのビジネス・ロジックが、同じように記述されます。違いは、TP フレームワークを使用しないことによって生じたものです。つまり主な相違点は、CORBA 関数を TP フレームワーク経由で間接的に使用するのではなく、直接使用するということです。

BEA Tuxedo CORBA サーバでは、Server インターフェイスを使用して、ORB がオブジェクトに対する要求を受信したときに、TP フレームワークがそのオブジェクトのためのサーバントの作成をユーザに求められるようにします。しかし、共同クライアント / サーバでは、要求が届く前にサーバントを作成する役割をユーザ・プログラムが担います。したがって、Server インターフェイスは不要です。通常、プログラムはサーバントを作成してから、オブジェクトへのリファレンスを渡す前に、そのオブジェクトを活性化

します。これには、サーバントと ObjectId を使用します。ObjectId はおそらく、システムで生成されています。そのようなオブジェクトは、コールバックの処理に使用されていると考えられます。したがって、サーバントは既に存在しており、オブジェクトは、そのオブジェクトに対する要求が届く前に、活性化されます。

共同クライアント / サーバは、使用されているのが C++ クライアント ORB なのか Java クライアント ORB なのかによって、動作が少し異なります。

- C++ 共同クライアント / サーバの場合、ある特定のオペレーションを実行するには、TP インターフェイスを呼び出すのではなく、TP インターフェイスの内部機能である ORB と POA の呼び出しをクライアント・サーバントから直接行います。また、ORB および POA との対話の多くは、すべてのアプリケーションで同じなので、使いやすくするため、クライアント・ライブラリにより、単一のオペレーションを使用して同じことを行う便利なラッパー・オブジェクトを提供することもできます。便利なラッパー・オブジェクトの使い方については、「サポートされているコールバック・オブジェクト・モデル」および「BEAWrapper Callbacks を使用してのコールバック・オブジェクトの準備」を参照してください。
- Java 共同クライアント / サーバの場合、ある特定のオペレーションを実行するには、TP インターフェイスを呼び出すのではなく、クライアント・サーバが直接、Java JDK 1.2 ORB に基づくクライアントである ORB と BOA の呼び出しを行います。また、ORB および BOA との対話の多くは、すべてのアプリケーションで同じなので、共同クライアント / サーバ・ライブラリ (wleclient.jar) により、単一のオペレーションを使用して同じことを行う、便利なラッパー・オブジェクト (Callbacks) を提供することもできます。加えて、ラッパー・オブジェクトは、ObjectIds 用に POA に類似する追加の寿命方針を提供します。「サポートされているコールバック・オブジェクト・モデル」および「BEAWrapper Callbacks を使用してのコールバック・オブジェクトの準備」を参照してください。Java 共同クライアント / サーバの例については、『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。

スケルトンからのサーバントの継承

コールバックをサポートするクライアントでは、サーバの場合と同様に、IDL コンパイラ (idl) によって生成されたものと同じスケルトン・クラス名から継承したインプリメンテーション・クラスを記述します。

C++ におけるスケルトンからの継承例

以下は、次の IDL を想定した C++ の例です。

```
interface Hospital{ … };
```

idl コマンドによって生成されたスケルトンには、次のように、ユーザ記述クラスの継承元である「スケルトン」クラス POA_Hospital が包含されます。

```
class Hospital_i : public POA_Hospital { ... };
```

サーバでは、スケルトン・クラスは TP フレームワーク・クラス Tobj_ServantBase から継承します。これがさらに、定義済みの PortableServer::ServantBase から継承しています。

共同クライアント / サーバのコールバック・オブジェクトのインプリメンテーションにおける継承ツリーは、サーバにおけるものとは異なります。スケルトン・クラスは TP フレームワーク・クラス Tobj_ServantBase からは継承しませんが、直接 PortableServer::ServantBase から継承します。この振る舞いは、idl コマンドで -P オプションを指定することによって行われます。

サーバントの継承ツリーに Tobj_ServantBase クラスがないということは、そのサーバントに activate_object メソッドと deactivate_object メソッドがないことを意味します。サーバでは、これらのメソッドは TP フレームワークによって呼び出され、サーバントに対してあるメソッドを呼び出す前に、そのサーバントの状態を動的に初期化して保存します。コールバックをサポートするクライアントの場合は、明示的にサーバントを作成し、サーバントの状態を初期化するコードを記述する必要があります。

Java におけるスケルトンからの継承例

以下は、次の IDL を想定した Java の例です。

```
interface Hospital{ … };
```

idltojava によって生成されたスケルトンには、次のようにユーザ記述クラスの継承元であるスケルトン・クラス _HospitalImplBase が包含されます。

```
class HospitalImpl extends _HospitalImplBase {…};
```

BEA Tuxedo サーバ・アプリケーションでは、スケルトン・クラスは TP フレームワーク・クラス com.beasys.Tobj_Servant から継承します。これがさらに、CORBA 定義のクラス org.omg.PortableServer.Servant から継承しています。

共同クライアント / サーバ・アプリケーションのコールバック・オブジェクトのインプリメンテーションにおける継承ツリーは、クライアントにおけるものとは異なります。スケルトン・クラスは TP フレームワーク・クラスからは継承しませんが、org.omg.CORBA.DynamicImplementation クラスから継承します。これがさらに、org.omg.CORBA.portable.ObjectImpl クラスから継承しています。

サーバントの継承ツリーに Tobj_Servant クラスがないということは、そのサーバントに activate_object メソッドと deactivate_object メソッドがないことを意味します。BEA Tuxedo サーバ・アプリケーションでは、これらのメソッドは TP フレームワークによって呼び出され、サーバントに対してあるメソッドを呼び出す前に、そのサーバントの状態を動的に初期化して保存します。共同クライアント / サーバ・アプリケーションでは、ユーザ・コードは明示的にサーバントを作成し、サーバントの状態を初期化する必要があります。したがって、Tobj_Servant オペレーションは不要です。

サポートされているコールバック・オブジェクト・モデル

BEA Tuxedo CORBA は、4 種類のコールバック・オブジェクトをサポートしており、その中で一般的である 3 種類に対してはラッパーを提供しています。これらのオブジェクトは、POA 方針における 3 つの組み合わせに対応しています。POA 方針は、使用可能なオブジェクトの型と、オブジェクト・リファレンスの型の両方を制御します。

適用できる POA 方針は、以下のとおりです。

- LifeSpanPolicy。オブジェクト・リファレンスの有効期間の長さを制御します。
- IdAssignmentPolicy。ユーザとシステムのどちらが ObjectId を割り当てるかを制御します。

これらのオブジェクトについては、ORB と POA による扱われ方の詳細ではなく、主に動作特性との関連で説明します。これらの詳細は、直接的な ORB および POA の呼び出し (CORBA サーバに関して特別な知識が少し必要) を使用するか、または ORB および POA の呼び出しを隠ぺいする BEAWrapper Callbacks インターフェイス (詳細については考慮しないユーザ向け) を使用して、以下の節で説明します。

- Transient/SystemId ― オブジェクト・リファレンスは、クライアント・プロセスの有効期間中のみ有効です。ObjectId はクライアント・アプリケーションによって割り当てられるのではなく、システムによって割り当てられる、一意の値です。この型のオブジェクトは、クライアントが終了するまでの間のみクライアントによって受信される呼び出しに有用です。対応する POA LifeSpanPolicy 値は TRANSIENT で、IdAssignmentPolicy は SYSTEM_ID です。
- Persistent/SystemId ― オブジェクト・リファレンスは、複数回の活性化にわたって有効です。ObjectId はクライアント・アプリケーションによって割り当てられるのではなく、システムによって割り当てられる、一意の値です。この型のオブジェクトおよびオブジェクト・リファレンスは、クライアントがある一定期間内で起動したり終了する場合に有用です。稼動中のクライアントは、その特定のオブジェクト・リファレンスに対するコールバック・オブジェクトを受信できます。

  通常、クライアントは一度オブジェクト・リファレンスを作成し、それを自身の恒久的な記憶域に保存し、そのオブジェクトが復帰するたびに、オブジェクトのサーバントを再活性化します。たとえば BEA Tuxedo CORBA ノーティフィケーション・サービスのアプリケーションと共に使用した場合、これらは永続的なサブスクリプションの概念に対応するコールバックです。つまり、ノーティフィケーション・サービスはコールバック・リファレンスを記憶しており、クライアントが起動して再びイベントを受信する準備が整ったことを宣言すると、イベントを送達します。これにより、ノーティフィケーション・サービスのサブスクリプションは、クライアントの障害やオフライン時間の影響を受けずに済みます。対応する POA 方針の値は、PERSISTENT および SYSTEM_ID です。

- Persistent/UserId — `ObjectId` がクライアント・アプリケーションによって割り当てられなければならない点を除いては、Persistent/SystemId と同じです。該当する `ObjectId` としては、たとえばクライアントに対してのみ意味を持つデータベース・キーなどが考えられます。対応する POA 方針の値は、`PERSISTENT` および `USER_ID` です。

注記　Transient/UserId 方針の組み合わせ は、特に重要なものであるとは見なされません。永続的なケースのどちらかに類似した方式で、ユーザが POA を使用して自給することは可能ですが、BEA Tuxedo ラッパーは、特にそのために有用なわけではありません。

注記　BEA Tuxedo CORBA のネイティブ共同クライアント / サーバでは、永続方針はどちらもサポートされません。一時方針のみがサポートされます。

リモート共同クライアント / サーバ・オブジェクトを呼び出すためのサーバのコンフィギュレーション

BEA Tuxedo サーバがリモート共同クライアント / サーバ・オブジェクト、すなわち BEA Tuxedo ドメイン外部の共同クライアント / サーバ・オブジェクトを呼び出せるようにするには、アウトバウンド IIOP が有効になるようにサーバをコンフィギュレーションする必要があります。この機能は、IIOP サーバ・リスナ (ISL) コマンドで -O (大文字の O) オプションを指定することによって有効化されます。-O オプションを設定すると、IIOP リスナ・ハンドラ (ISH) に接続されていない共同クライアント / サーバ・オブジェクトに対するアウトバウンド呼び出し (アウトバウンド IIOP) が有効になります。

ISL コマンド・オプションの設定は、サーバの UBBCONFIG ファイルの SERVERS セクションで行います。アウトバウンド IIOP のサポートには、少量の予備資源が必要なので、デフォルトではアウトバウンド IIOP は無効になっています。詳細については、『BEA Tuxedo アプリケーションの設定』の「ISL コマンドを使用してアウトバウンド IIOPQ を設定する」と、『BEA Tuxedo コマンド・リファレンス』の「ISL(1)」を参照してください。

CORBA を使用してのコールバック・オブジェクトの準備 (C++ 共同クライアント / サーバのみ)

CORBA を使用して BEA Tuxedo C++ コールバック・オブジェクトを設定するには、クライアントは次の処理を行う必要があります。

1. コールバック・オブジェクト・モデルに適した方針を備える POA との間に接続を確立します。これはデフォルトで利用可能なルート POA になることも、新規 POA の作成が必要となることもあります。
2. サーバント (インターフェイスの C++ インプリメンテーション・クラスのインスタンス) を作成します。

3. サーバントでコールバック・オブジェクトに対する要求を受け付ける準備ができていることを POA に通知します。技術的には、これはクライアントが POA 内のオブジェクトを activate すること、つまりサーバントと ObjectId を POA のアクティブ・オブジェクト・マップに入れることを意味します。
4. ネットワークからの要求の受け付けを開始するように、POA に指示します。これはつまり、POA 自身を活性化するということです。
5. コールバック・オブジェクトのオブジェクト・リファレンスを作成します。
6. オブジェクト・リファレンスを割り当てます。これは通常、パラメータとしてコールバック・オブジェクト・リファレンスを使用して別のオブジェクトに対する呼び出しを行うことによってなされます。この別のオブジェクトは後になってから、コールバック・オブジェクトの呼び出し (コールバック呼び出しの実行) を行います。

クライアントが既に ORB へのリファレンスを取得済みであれば、このタスクの実行に必要な ORB および POA との対話は 4 回です。リスト 11-1 に示すモデルのようになります。このモデルでは、ルート POA のみが必要とされます。

コード リスト 11-1 Transient/SystemId **モデル**

```
// コールバック・オブジェクトのサーバントを作成
Catcher_i* my_catcher_i = new Catcher_i();

// ルート POA リファレンスを取得して、POA を活性化
1   CORBA::Object_var oref =
              orb->resolve_initial_references("RootPOA");
2   PortableServer::POA_var root_poa =
              PortableServer::POA::_narrow(oref);
3   root_poa -> the_POAManager() -> activate();
4   PortableServer::objectId_var temp_Oid =
              root_poa ->activate_object ( my_catcher_i );
5   oref = root_poa->create_reference_with_id(
                                  temp_Oid, _tc_Catcher->id() );
6   Catcher_var my_catcher_ref = Catcher::_narrow( oref );
```

Persistent/UserId モデルを使用するには、POA 作成時に、いくつかの追加手順が必要です。さらに、クライアント側で ObjectId を指定します。これには、さらに別の手順が必要です。これは リスト 11-2 に示すモデルのようになります。

コード リスト 11-2 Persistent/UserId **モデル**

```
    Catcher_i* my_catcher_i = new Catcher_i();
    const char* oid_str = "783";
1   PortableServer::objectId_var oid =
            PortableServer::string_to_objectId(oid_str);
// ルート POA を見つける
2   CORBA::Object_var oref =
            orb->resolve_initial_references("RootPOA");
3   PortableServer::POA_var root_poa =
            PortableServer::POA::_narrow(oref);
// Persistent/UserId POA を作成して活性化
4   CORBA::PolicyList policies(2);
5   policies.length(2);
6   policies[0] = root_poa->create_lifespan_policy(
                                    PortableServer::PERSISTENT);
7   policies[1] = root_poa->create_id_assignment_policy(
                                     PortableServer::USER_ID );
8   PortableServer::POA_var my_poa_ref =
            root_poa->create_POA(
              "my_poa_ref", root_poa->the_POAManager(), policies);
9   root_poa->the_POAmanager()->activate();
// コールバック・オブジェクトのオブジェクト・リファレンスを作成
10 oref =  my_poa_ref -> create_reference_with_id(
                                          oid, _tc_Catcher->id());
11 Catcher_var my_catcher_ref = Catcher::_narrow( oref );
// オブジェクトを活性化
12 my_poa_ref -> activate_object_with_id( oid, my_catcher_i );
// callback ref を渡す呼び出しを行う
    foo -> register_callback ( my_catcher_ref );
```

ここに記載したインターフェイスとオペレーションはすべて、標準的な CORBA インターフェイスおよびオペレーションです。

BEAWrapper Callbacks を使用してのコールバック・オブジェクトの準備

C++ または Java の共同クライアント / サーバのどちらかを記述するには、BEAWrapper Callbacks API を使用します。

C++ での BEAWrapper Callbacks の使用

コールバック・オブジェクトに必要なコードは、コールバックをサポートするどのクライアントについてもほぼ同一であるため、共同クライアント / サーバ用のライブラリで提供されている BEAWrapper を使用すると便利なことがあります。

リスト 11-3 に示すように、BEAWrapper は IDL で記述されます。

コード リスト 11-3 BEAWrapper IDL

```
// ファイル : BEAWrapper
#ifndef _BEA_WRAPPER _IDL_
#define _BEA_WRAPPER _IDL_
#include <orb.idl>
#include <PortableServer.idll>

#pragma prefix “beasys.com”

module BEAWrapper {
   interface Callbacks
   {
       exception ServantAlreadyActive{ };
       exception ObjectAlreadyActive { };
       exception NotInRequest{ };

       // 一時コールバック・オブジェクトを設定
       // POA を準備し、オブジェクトを活性化し、objref を返す
          Object start_transient(
                        in PortableServer::Servant      Servant,
                        in CORBA::RepositoryId          rep_id)
            raises (ServantAlreadyActive);
```

```
        // persistent/systemid コールバック・オブジェクトを設定
        Object start_persistent_systemid(
                in PortableServer::Servant      servant,
                in CORBA::Repository            rep_id,
                out string                      stroid)
           raises (ServantAlreadyActive);

        // persistent/systemid コールバック・オブジェクト
        // の設定に戻す
        Object restart_persistent_systemid(
                in PortableServer::Servant     servant,
                in CORBA::RepositoryId         rep_id,
                in string                      stroid)
           raises (ServantAlreadyActive, ObjectAlreadyActive);

        // persistent/userid コールバック・オブジェクトを設定
        Object start_persistent_userid(
                 in PortableServer::Servant        servant,
                 in CORBA::RepositoryId            rep_id,
                 in string                        stroid)
           raises (ServantAlreadyActive, ObjectAlreadyActive);

        // 所定のサーバントによる、特定のコールバック・オブジェクトの
        // 処理を停止
        void stop_object( in PortableServer::Servant servant);

        // すべてのコールバック・オブジェクト処理を停止
        void stop_all_objects();

       // 現在の要求に対する oid 文字列を取得
       string get_string_oid() raises (NotInRequest);
    };
}
#endif /* _BEA_WRAPPER _IDL_ */
```

リスト 11-4 に示すように、BEAWrapper は C++ で記述されます。

コード リスト 11-4 C++ **宣言** (beawrapper.h **内**)

```
#ifndef _BEAWRAPPER_H_
#define _BEAWRAPPER_H_

#include <PortableServer.h>
class BEAWrapper{
```

```
class Callbacks{
      public:
            Callbacks (CORBA::ORB_ptr init_orb);

            CORBA::Object_ptr start_transient (
                        PortableServer::Servant servant,
                         const char *     rep_id);

            CORBA::Object_ptr start_persistent_systemid (
                          PortableServer::Servant servant,
                          const char *     rep_id,
                           char * & stroid);

            CORBA::Object_ptr restart_persistent_systemid (
                           PortableServer::Servant servant,
                           const char *     rep_id,
                            const char *     stroid);

            CORBA::Object_ptr start_persistent_userid (
                           PortableServer::Servant servant,
                           const char *     rep_id,
                            const char *     stroid);

              void stop_object(PortableServer::Servant servant);

              char* get_string_oid ();

              void stop_all_objects();

             ~Callbacks();
        private:

              static CORBA::ORB_var orb_ptr;

              static PortableServer::POA_var root_poa;
              static PortableServer::POA_var trasys_poa;
              static PortableServer::POA_var persys_poa;
              static PortableServer::POA_var peruser_poa;
    };
};
#endif // _BEAWRAPPER_H_
```

以下に、BEAWrapper::Callbacks インターフェイスの各オペレーションを上記で宣言された順に説明します。

Java での BEAWrapper Callbacks の使用

コールバック・オブジェクトの準備をするためのコードは、すべての共同クライアント / サーバ・アプリケーションについてほぼ同一であり、Java JDK ORB は POA をインプリメントしないため、BEA Tuxedo は C++ で提供されるラッパー・クラスと事実上同一のラッパー・クラスを共同クライアント / サーバ・ライブラリ内で提供します。このラッパー・クラスは、3 種類のコールバック・オブジェクトをサポートするのに必要な POA 方針をエミュレートします。

リスト 11-5 は、Java `Callback` ラッパー・インターフェイスを示します。

コード リスト 11-5 Java Callback **ラッパー・インターフェイス**

```
package com.beasys.BEAWrapper;

       class Callbacks{
             public Callbacks ();

             public Callbacks (org.omg.CORBA.Object init_orb);

             public org.omg.CORBA.Object start_transient (
                                  org.omg.PortableServer.ObjectImpl servant,
                                  java.lang.String rep_id)
                                  throws ServantAlreadyActive,
                                         org.omg.CORBA.BAD_PARAMETER;

             public org.omg.CORBA.Object start_persistent_systemid (
                                  org.omg.PortableServer.ObjectImpl servant,
                                  java.lang.String rep_id,
                                  org.omg.CORBA.StringHolder stroid)
                                  throws ServantAlreadyActive,
                                         org.omg.CORBA.BAD_PARAMETER,
                                         org.omg.CORBA.IMP_LIMIT;

             public org.omg.CORBA.Object restart_persistent_systemid (
                                  org.omg.PortableServer.ObjectImpl servant,
                                  java.lang.String rep_id,
                                  java.lang.String stroid)
                                  throws ServantAlreadyActive,
                                         ObjectAlreadyActive,
                                         org.omg.CORBA.BAD_PARAMETER,
                                         org.omg.CORBA.IMP_LIMIT;
```

```
            public org.omg.CORBA.Object start_persistent_userid (
                                org.omg.PortableServer.ObjectImpl servant,
                                java.lang.String rep_id,
                                java.lang.String stroid)
                                throws ServantAlreadyActive,
                                       ObjectAlreadyActive,
                                       org.omg.CORBA.BAD_PARAMETER,
                                       org.omg.CORBA.IMP_LIMIT;
            public void stop_object(
                                org.omg.PortableServer.ObjectImpl
                                                        servant);

            public String get_string_oid ()
                                throws NotInRequest;

            public void stop_all_objects();
};
```

Java 共同クライアント / サーバのプログラミング上の考慮事項

ここでは、次の Java のプログラミング上のトピックについて説明します。

- メイン・プログラムのスレッドに関する考慮事項
- 複数のスレッドのしくみ
- Java クライアント ORB の初期化
- IIOP のサポート

メイン・プログラムのスレッドに関する考慮事項

プログラムが Java クライアントにおいて、Java 共同クライアント / サーバの場合と同様にクライアントとサーバの両方の機能を持つとき、この 2 つの部分を別々のスレッドで同時に実行できます。実行環境としての Java は本来マルチスレッドなので、Java クライアントから `org.omg.CORBA.orb.work_pending` メソッドおよび `org.omg.CORBA.orb.perform_work` メソッドを呼び出す理由はありません。

実際、Java クライアントがこれらのメソッドを呼び出そうとすると、メソッドからは org.omg.CORBA.NO_IMPLEMENT 例外が発生します。クライアントが org.omg.CORBA.orb.run メソッドを呼び出す必要はありません。すべてのマルチスレッド環境の場合と同じく、クライアント・アプリケーション内で同時に実行できるコード (コールバック用のクライアントとサーバント) はすべて、スレッド・セーフになるようにコーディングする必要があります。

複数のスレッドのしくみ

Java では、クライアントはメイン・スレッドで起動します。次にクライアントは、Callbacks ラッパー・クラスで提供される *(re)*start_*xxxx* メソッドのうちの任意のものに対する呼び出しを通じて、コールバック・オブジェクトを設定します。ラッパー・クラスは、サーバント、および ORB のオブジェクト・マネージャにおける関連の OID の登録を処理します。これでアプリケーションは、*(re)*start_*xxxx* メソッドから返されたオブジェクト・リファレンスを、サーバントのコールバックが必要なアプリケーションへ、自由に渡せるようになります。

注記　ORB は、サーバントを効果的に初期化し、別のアプリケーションに正しくマーシャリングできる有効なオブジェクト・リファレンスを作成するために、*(re)*start_*xxxx* メソッドの 1 つに対する明示的な呼び出しを必要とします。これは、アプリケーションがまだオブジェクト・リファレンスをマーシャリングしていない場合、それを行う際に orb.connect メソッドを内部的に呼び出すことによって、暗黙的なオブジェクト・リファレンス作成を可能とする、基本的な JDK 1.2 ORB の振る舞いからは逸脱しています。

コールバック・オブジェクトに対する呼び出しは、ORB によって処理されます。各要求を受信すると、ORB はその要求をオブジェクト・マネージャに対して評価し、その要求のスレッドを作成します。ORB は各要求について新規スレッドを作成するので、同じオブジェクトに同時に複数の要求を行うことができます。これが、コールバックのサーバント・コードをスレッド・セーフになるように記述する必要がある理由です。各要求が終了すると、サーバントを実行するスレッドも終了します。

メイン・クライアント・スレッドは、クライアント呼び出しを何度でも必要なだけ行うことができます。stop_*(all_)*object メソッドに対する呼び出しは、単にオブジェクト・マネージャのリストからオブジェクトを取り出し、

それに対するこれ以上の呼び出しを防止するだけです。停止されたオブジェクトの呼び出しは、そのオブジェクトがまったく存在しない場合と同様に失敗します。

別のスレッドからコールバックの結果を取得する必要がある場合、クライアント・アプリケーションはそのために通常のスレッド同期化技術を使用する必要があります。

BEA Tuxedo のリモート的なクライアント・アプリケーション内のスレッド (クライアント・メインまたはサーバント) が終了すると、すべてのクライアント・プロセス・アクティビティが停止され、Java 実行環境が終了します。`return` メソッドの呼び出しはスレッドを終了させる場合にのみ行うことをお勧めします。

Java クライアント ORB の初期化

クライアント・アプリケーションは、BEA 提供のプロパティによって ORB を初期化する必要があります。これは、ORB が `Callbacks` ラッパー・クラスおよび Bootstrap オブジェクトをサポートする、BEA 提供のクラスおよびメソッドを利用できるようにするためです。これらのクラスは、`$TUXDIR/udataobj/java/jdk` (Soloaris の場合) または `%TUXDIR%\udataobj\java\jdk` (Windows の場合) にインストールされる `wleclient.jar` にあります。アプリケーションは、これを行うために、リスト 11-6 に示すいくつかのシステム・プロパティを設定する必要があります。

コード リスト 11-6 **システム・プロパティの設定**

```
Properties prop = new Properties(System.getProperties());
prop.put("org.omg.CORBA.ORBClass","com.beasys.CORBA.iiop.ORB");
prop.put("org.omg.CORBA.ORBSingletonClass",
         "com.beasys.CORBA.idl.ORBSingleton");
System.setProperties(prop);
//ORB を初期化
ORB orb = ORB.init(args, prop);
```

IIOP のサポート

IIOP は、ORB 間の通信に使用するプロトコルです。IIOP は、さまざまなベンダ製の ORB の相互運用を可能にします。Java サーバ・アプリケーションの場合、永続オブジェクト・リファレンス方針またはユーザ ID オブジェクト・リファレンス方針について、クライアント側でポート番号を指定する必要があります。

Java アプレットのサポート

コールバックまたはコールアウトを受信するアプレットのための IIOP のサポートは、アプレットのセキュリティ・メカニズムにより制限されています。独自の環境またはプロトコルによるソケットの作成およびリッスンが可能なアプレット実行時環境であれば、BEA Tuxedo 共同クライアント / サーバ・アプリケーションとして動作できます。アプレットの実行時環境がソケット通信を制限している場合、そのアプレットを BEA Tuxedo アプリケーションに対する共同クライアント / サーバ・アプリケーションにはできません。

永続オブジェクト・リファレンスのためのポート番号

BEA Tuxedo Java リモート共同クライアント / サーバ・アプリケーションで IIOP をサポートするには、サーバ・コンポーネント用に作成されたオブジェクト・リファレンスにホストとポートが含まれている必要があります。一時オブジェクト・リファレンスの場合、どのポートでも条件は満たされ、ORB で動的に取得できます。しかし、これは永続オブジェクト・リファレンスには不適当です。

永続リファレンスは、ORB の再起動後に同じポート上で処理する必要があります。つまり、ORB はオブジェクト・リファレンスを作成したのと同じポート上で要求を受け付けるように準備されなければなりません。したがって、特定のポートを使用するように ORB をコンフィギュレーションする方法が必要です。

永続リファレンスのコールバック用サーバとして動作する予定の Java クライアントを、まず指定のポートで起動する必要があります。これは、システム・プロパティ `org.omg.CORBA.ORBPort` を次のコマンドのように設定することによって行います。

Windows:

```
java -DTOBJADDR=//host:port
  -Dorg.omg.CORBA.ORBPort=xxxx
  -classpath=%CLASSPATH% client
```

UNIX:

```
java -DTOBJADDR=//host:port
  -Dorg.omg.CORBA.ORBPort=xxxx
  -classpath=$CLASSPATH client
```

通常、システム管理者は、動的範囲ではなくポート番号のユーザ範囲からクライアントのポート番号を割り当てます。これにより、共同クライアント / サーバ・アプリケーションでポートの競合を防ぐことができます。

BEA Tuxedo リモート共同クライアント / サーバ・アプリケーションが、上記のコマンド行のようにポートの設定をせずに永続オブジェクト・リファレンスを作成しようとすると、オペレーションにより例外 IMP_LIMIT が発生し、真に永続的なオブジェクト・リファレンスが作成できないことをユーザに通知します。

C++ BEAWrapper Callbacks インターフェイス API

この C++ BEAWrapper Callbacks インターフェイス API については、以下の節で説明します。

Callbacks

概要 Callbacks インターフェイスへのリファレンスを返します。

C++ バインディング

```
BEAWrapper::Callbacks( CORBA::ORB_ptr init_orb);
```

Java バインディング

```
public Callbacks(org.omg.CORBA.Object init_orb);
```

引数

`init_orb`
これ以降のすべてのオペレーションに使用する ORB。

例外

`CORBA::IMP_LIMIT`
`BEAWrapper::Callbacks` クラスは既に ORB ポインタでインスタンス化されています。プロセス内で使用できるこのクラスのインスタンスは 1 つだけです。より柔軟性が必要なユーザは、POA を直接使用する必要があります。

説明 コンストラクタは、Callbacks インターフェイスへのリファレンスを返します。複数のスレッドが使用されている場合でも、プロセスのために作成するそのようなオブジェクトは 1 つのみとします。複数のオブジェクトを使用すると、未定義の振る舞いが発生します。

戻り値 Callbacks オブジェクトへのリファレンス。

start_transient

概要 オブジェクトを活性化し、ORB および POA を適切な状態に設定し、活性化したオブジェクトへのオブジェクト・リファレンスを返します。

IDL
```
Object start_transient( in PortableServer::Servant   servant,
                        in CORBA::RepositoryId       rep_id)
          raises ( ServantAlreadyActive );
```

C++ **バインディング**
```
CORBA::Object_ptr start_transient(
                    PortableServer::Servant         servant,
                    const char*                     rep_id);
```

Java **バインディング**
```
org.omg.CORBA.Object start_transient(
                    org.omg.PortableServer.Servant  servant,
                    java.lang.String                rep_id);
```

引数 サーバント

インターフェイスの C++ インプリメンテーション・クラスのインスタンス。

`rep_id`

インターフェイスのリポジトリ `id`。

例外 `ServantAlreadyActive`

サーバントは既にコールバックに使用されています。サーバントは、`ObjectId` が 1 つのコールバックのみに使用できます。複数の `ObjectId` があるオブジェクトに対するコールバックを受信するには、複数のサーバントを作成して、個別に活性化する必要があります。同じサーバントを再利用できるのは、`stop_object` オペレーションがシステムに対して、サーバントを元の `ObjectId` について使用することを止めるように指示する場合のみです。

`CORBA::BAD_PARAM`

リポジトリ ID が NULL 文字列であったか、サーバントが NULL ポインタでした。

説明 このオペレーションは、次の作業を実行します。

- rep_id 型のサービス・オブジェクトに与えられた Servant を使用するオブジェクトを活性化します。これには、システムによって生成された ObjectId を使用します。
- ORB および POA を、このオブジェクトに対する要求を受け付ける状態に設定します。
- 活性化されたオブジェクトへのオブジェクト・リファレンスを返します。返されたオブジェクト・リファレンスが有効なのは、クライアントが終了するまで、または stop_object オペレーションによってユーザがコールバック・サーバントを停止するまでのみです。それ以降は、そのオブジェクト・リファレンスに対する呼び出しは無効となり、絶対に有効化できません。

戻り値 CORBA::Object_ptr

システムが生成した ObjectId と、ユーザが指定した rep_id で作成されたオブジェクトへのリファレンス。オブジェクト・リファレンスは、特定のオブジェクト用に定義された _narrow() オペレーションを呼び出すことによって、特定のオブジェクト型に変換する必要があります。変換が終了したときにオブジェクトを解放するのは、呼び出し側の役割です。

start_persistent_systemid

概要 オブジェクトを活性化し、ORB および POA を適切な状態に設定し、出力パラメータ stroid を設定し、活性化したオブジェクトへのオブジェクト・リファレンスを返します。

IDL
```
Object start_persistent_systemid(
                        in PortableServer::Servant      servant,
                        in CORBA::RepositoryId          rep_id,
                        out string                      stroid)
    raises ( ServantAlreadyActive );
```

C++ バインディング
```
CORBA::Object_ptr start_persistent_systemid(
                         PortableServer::Servant          servant,
                         const char*                      rep_id,
                         char*&                            stroid);
```

Java バインディング
```
org.omg.CORBA.Object start_persistent_systemid(
                         org.omg.PortableServer.Servant   servant,
                         java.lang.String                 rep_id,
                         java.lang.String                 stroid);
```

引数 サーバント
インターフェイスの C++ インプリメンテーション・クラスのインスタンス。

rep_id
インターフェイスのリポジトリ ID。

stroid
この引数はシステムによって設定され、ユーザにとってはオペークです。クライアントは、後になって、おそらくはクライアント・プロセスが終了して再起動してから、この引数が restart_persistent_systemid でオブジェクトを再活性化する場合に、これを使用します。

例外 ServantAlreadyActive
サーバントは既にコールバックに使用されています。サーバントは、ObjectId が 1 つのコールバックのみに使用できます。複数の ObjectId があるコールバックを受信するには、複数のサーバントを作成して、個別に活性化する必要があります。同じサーバントを再利用できるのは、stop オペレーションがシステムに対して、元の

ObjectId についてこのサーバントを使用することを止めるように指示する場合のみです。

CORBA::BAD_PARAMETER
リポジトリ ID が NULL 文字列であったか、サーバントが NULL ポインタでした。

CORBA::IMP_LIMIT
この例外のほかのシステム上の理由に加えて、この状況に特有の理由は、共同クライアント / サーバがポート番号付きでは初期化されておらず、したがって永続オブジェクト・リファレンスが作成できないことです。

説明 このオペレーションは、次の作業を実行します。

- rep_id 型のサービス・オブジェクトに与えられた Servant を使用するオブジェクトを活性化します。これには、システムによって生成された ObjectId を使用します。
- ORB および POA を、このオブジェクトに対する要求を受け付ける状態に設定します。
- 出力パラメータ stroid を、システムによって割り当てられた ObjectId の文字列化されたバージョンに設定します。
- 活性化されたオブジェクトへのオブジェクト・リファレンスを返します。返されたオブジェクト・リファレンスは、クライアントの終了後でも有効です。つまり、クライアントが終了してから再起動され、その後同じ rep_id で、同じ ObjectId についてサーバントを活性化した場合に、サーバントはその同じオブジェクト・リファレンスに対して行われた要求を受け付けます。ObjectId はシステムによって生成されているため、アプリケーションはその ObjectId を保存しておく必要があります。

戻り値 CORBA::Object_ptr
システムが生成した ObjectId と、ユーザが指定した rep_id で作成されたオブジェクト・リファレンス。オブジェクト・リファレンスは、特定のオブジェクト用に定義された _narrow() オペレーションを呼び出すことによって、特定のオブジェクト型に変換する必要があります。変換が終了したときにオブジェクトを解放するのは、呼び出し側の役割です。

restart_persistent_systemid

概要 オブジェクトを活性化し、ORB および POA を適切な状態に設定し、活性化したオブジェクトへのオブジェクト・リファレンスを返します。

IDL

```
Object restart_persistent_systemid(
                in PortableServer::Servant   servant,
                in CORBA::RepositoryId       rep_id,
                in string                    stroid)
    raises (ServantAlreadyActive, ObjectAlreadyActive);
```

C++ バインディング

```
CORBA::Object_ptr restart_persistent_systemid(
                PortableServer::Servant   servant,
                const char*               rep_id
                const char*               stroid);
```

Java バインディング

```
org.omg.CORBA.Object restart_persistent_systemid(
                 org.omg.PortableServer.Servant    servant,
                 java.lang.String                  rep_id,
                 java.lang.String                  stroid);
```

引数

サーバント
: インターフェイスの C++ インプリメンテーション・クラスのインスタンス。

rep_id
: インターフェイスのリポジトリ ID。

stroid
: 作成されているオブジェクト・リファレンス内で設定される、ユーザ指定の ObjectId の文字列化されたバージョン。
start_persistent_systemid に対する以前の呼び出しから返されたものである必要があります。

例外

ServantAlreadyActive
: サーバントは既にコールバックに使用されています。サーバントは、ObjectId が 1 つのコールバックのみに使用できます。複数の ObjectId があるオブジェクトに対するコールバックを受信するには、複数のサーバントを作成して、個別に活性化する必要があります。同じサーバントを再利用できるのは、stop_object オペレーションがシステムに対して、サーバントを元の ObjectId について使用することを止めるように指示する場合のみです。

ObjectAlreadyActive
: 文字列化された ObjectId は既にコールバックに使用されています。ある特定の ObjectId には、1 つのサーバントしか関連付けられません。別のサーバントに変更する場合は、まず現在使用しているサーバントで stop_object を呼び出す必要があります。

CORBA::BAD_PARAM
: リポジトリ ID が NULL 文字列であったか、またはサーバントが NULL ポインタであったか、または指定された ObjectId が事前にシステムによって割り当てられていませんでした。

CORBA::IMP_LIMIT
: この例外のほかのシステム上の理由に加えて、この状況に特有の理由は、共同クライアント / サーバがポート番号付きでは初期化されておらず、したがって永続オブジェクト・リファレンスが作成できないことです。

説明 このオペレーションは、次の作業を実行します。

- rep_id 型のサービス・オブジェクトに与えられた Servant を使用するオブジェクトを活性化します。これには、指定の stroid (文字列化された ObjectId) を使用します。これは、事前に start_persistent_systemid に対する呼び出しによって取得されている必要があります。
- ORB および POA を、このオブジェクトに対する要求を受け付ける状態に設定します。
- 活性化されたオブジェクトへのオブジェクト・リファレンスを返します。
- 再活性化は、restart_persistent_systemid メソッドを使用して行われます。

戻り値 CORBA::Object_ptr
: 文字列化された ObjectId stroid と、ユーザが指定した rep_id で作成されたオブジェクト・リファレンス。オブジェクト・リファレンスは、特定のオブジェクト用に定義された _narrow() オペレーションを呼び出すことによって、特定のオブジェクト型に変換する必要があります。終了したときにオブジェクトを解放するのは、呼び出し側の役割です。

start_persistent_userid

概要 オブジェクトを活性化し、ORB および POA を適切な状態に設定し、活性化したオブジェクトへのオブジェクト・リファレンスを返します。

IDL

```
Object start_persistent_userid(
                portableServer::Servant     a_servant,
                in CORBA::RepositoryId      rep_id,
                in string                   stroid)
    raises ( ServantAlreadyActive, ObjectAlreadyActive );
```

C++ バインディング

```
CORBA::Object_ptr start_persistent_userid (
                 PortableServer::Servant    servant,
                 const char*                rep_id,
                 const char*                stroid);
```

Java バインディング

```
org.omg.CORBA.Object start_persistent_userid(
                  org.omg.PortableServer.Servant     servant,
                  java.lang.String                   rep_id,
                  java.lang.String                   stroid);
```

引数

サーバント
: インターフェイスの C++ インプリメンテーション・クラスのインスタンス。

`rep_id`
: インターフェイスのリポジトリ ID。

`stroid`
: 作成されているオブジェクト・リファレンス内で設定される、ユーザ指定の `ObjectId` の文字列化されたバージョン。`stroid` は、アプリケーション固有のデータを保持しており、ORB 側からはオペークです。

例外

`ServantAlreadyActive`
: サーバントは既にコールバックに使用されています。サーバントは、`ObjectId` が 1 つのコールバックのみに使用できます。複数の `ObjectId` があるオブジェクトに対するコールバックを受信するには、複数のサーバントを作成して、個別に活性化する必要があります。同じサーバントを再利用できるのは、`stop_object` オペレーションがシステムに対して、サーバントを元の `ObjectId` について使用することを止めるように指示する場合のみです。

ObjectAlreadyActive
: 文字列化された ObjectId は既にコールバックに使用されています。ある特定の ObjectId には、1 つのサーバントしか関連付けられません。別のサーバントに変更する場合は、まず現在使用しているサーバントで stop_object を呼び出す必要があります。

CORBA::BAD_PARAM
: リポジトリ ID が NULL 文字列であったか、サーバントが NULL ポインタでした。

CORBA::IMP_LIMIT
: この例外のほかのシステム上の理由に加えて、この状況に特有の理由は、共同クライアント / サーバがポート番号付きでは初期化されておらず、したがって永続オブジェクト・リファレンスが作成できないことです。

説明 このオペレーションは、次の作業を実行します。

- rep_id 型のサービス・オブジェクトに与えられた Servant を使用するオブジェクトを活性化します。これには、オブジェクト ID stroid を使用します。
- ORB および POA を、このオブジェクトに対する要求を受け付ける状態に設定します。
- 活性化されたオブジェクトへのオブジェクト・リファレンスを返します。返されたオブジェクト・リファレンスは、クライアントの終了後でも有効です。つまり、クライアントが終了してから再起動され、その後同じ rep_id で、同じ ObjectId についてサーバントを活性化した場合に、サーバントはその同じオブジェクト・リファレンスに対して行われた要求を受け付けます。

戻り値 CORBA::Object_ptr
: 文字列化された ObjectId stroid と、ユーザが指定した rep_id で作成されたオブジェクト・リファレンス。オブジェクト・リファレンスは、特定のオブジェクト用に定義された _narrow() オペレーションを呼び出すことによって、特定のオブジェクト型に変換する必要があります。変換が終了したときにオブジェクトを解放するのは、呼び出し側の役割です。

stop_object

概要 所定のサーバントを使用しているオブジェクトに対する要求の受け付けを止めるように ORB に指示します。

IDL
```
void stop_object( in PortableServer::Servant servant);
```

C++ **バインディング**
```
void stop_object(PortableServer::Servant servant);
```

Java **バインディング**
```
void stop_object(org.omg.PortableServer.Servant servant);
```

引数

サーバント
: インターフェイスの C++ インプリメンテーション・クラスのインスタンス。このサーバントと、`ObjectId` の間の関連付けは、アクティブ・オブジェクト・マップから削除されます。

例外 特にありません。

説明 このオペレーションは、所定のサーバントに対する要求の受け付けを止めるように ORB に指示します。サーバントの状態は、活性化されていても、不活性化されていてもかまいません。エラーが報告されることはありません。

注記 `stop_object` オペレーションの呼び出し後にコールバック・オブジェクトを呼び出すと、呼び出し側に `OBJECT_NOT_EXIST` 例外が返されます。これは、`stop_object` オペレーションが事実上、オブジェクトを削除するためです。

戻り値 特にありません。

stop_all_objects

概要 すべてのサーバントに対する要求の受け付けを止めるように ORB に指示します。

IDL
```
void stop_all_objects ();
```

C++ **バインディング**
```
void stop_all_objects ();
```

Java **バインディング**
```
void stop_all_objects ();
```

例外 特にありません。

説明 このオペレーションは、このプロセスで設定されたすべてのサーバントに対する要求の受け付けを止めるように ORB に指示します。

使用上の注意 `ORB::shutdown` メソッドを呼び出したクライアントが、その後に `stop_all_objects` を呼び出さないようにしてください。

戻り値 特にありません。

get_string_oid

概要 現在の要求の ObjectId の文字列バージョンを要求します。

IDL
```
string get_string_oid() raises (NotInRequest);
```

C++ **バインディング**
```
char* get_string_oid();
```

Java **バインディング**
```
java.lang.String get_string_oid();
```

例外 NotInRequest

ORB が要求のコンテキスト内に入っていなかったとき、つまり ORB がメソッド・コード内の要求を処理していないときに、関数が呼び出されました。この関数をクライアント・コードから呼び出さないでください。これは、コールバック・オブジェクト (すなわちサーバント) のメソッド実行中のみ有効です。

説明 このオペレーションは、現在の要求の ObjectId の文字列バージョンを返します。

戻り値 char*

現在の要求の ObjectId の文字列バージョン。これは、オブジェクト・リファレンスの作成時に指定された文字列です。この文字列がユーザにとって意味を持つのは、オブジェクト・リファレンスが start_persistent_userid 関数によって作成された場合のみです。つまり、start_transient および start_persistent_systemid によって作成された ObjectId は、ORB によって作成されており、ユーザ・アプリケーションとの間に関係はありません。

~Callbacks

概要 コールバック・オブジェクトを破棄します。

C++ バインディング

```
BEAWrapper::~Callbacks( );
```

Java バインディング

```
public ~Callbacks();
```

引数 特にありません。

例外 特にありません。

説明 このデストラクタは、コールバック・オブジェクトを破棄します。

使用上の注意 ラッパーは破棄するが ORB はシャットダウンしない場合、クライアントは `stop_all_objects` メソッドを呼び出す必要があります。

戻り値 特にありません。

Java BEAWrapper Callbacks インターフェイス API

`BEAWrapper.Callbacks` インターフェイス API の完全な説明については、Javadoc API を参照してください。

章

12 開発コマンド

BEA Tuxedo 開発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。このマニュアルでは、BEA Tuxedo のコマンドとプロセスをすべて説明します。

『BEA Tuxedo コマンド・リファレンス』は、オンライン・マニュアルとして PDF でも提供しています。

章

13 OMG IDL 文の C++ へのマッピング

この章では、OMG IDL 文から C++ へのマッピングについて説明します。

注記　この章の一部の情報は、Object Management Group (OMG) の「Common Object Request Broker: C++ Language Mapping Specification」(1999 年 6 月) から、OMG の許可を得て転載しています。

マッピング

以下では、OMG IDL から C++ へのマッピングの次のトピックについて説明します。

- データ型
- 文字列
- wchar
- wstring
- 定数
- Enum
- 構造体

- 共用体
- シーケンス
- 配列
- 例外
- 擬似オブジェクトの C++ へのマッピング
- 形式
- マッピング規則
- C PIDL マッピングとの関係
- Typedef
- インターフェイスのインプリメント
- オペレーションのインプリメント
- PortableServer 関数
- モジュール
- インターフェイス
- 生成される静的メンバ関数
- オブジェクト・リファレンスの型
- 属性
- Any 型
- 値型

また、次のトピックについても説明します。

- 固定長ユーザ定義型と可変長ユーザ定義型
- var クラスの使い方
- out クラスの使い方
- 引数の受け渡しの考慮事項

データ型

各 OMG IDL のデータ型は、C++ のデータ型またはクラスにマッピングされます。

基本データ型

表 0-1 に示すように、OMG IDL 文の基本データ型は、CORBA モジュールの C++ typedef にマッピングされます。

表 0-1 OMG IDL および C++ の基本データ型

OMG IDL	C++	C++ Out 型
short	CORBA::Short	CORBA::Short_out
long	CORBA::Long	CORBA::Long_out
unsigned short	CORBA::UShort	CORBA::UShort_out
unsigned long	CORBA::ULong	CORBA::ULong_out
float	CORBA::Float	CORBA::Float_out
double	CORBA::Double	CORBA::Double_out
char	CORBA::Char	CORBA::Char_out
boolean	CORBA::Boolean	CORBA::Boolean_out
octet	CORBA::Octet	CORBA::Octet_out
wchar	CORBA::WChar	CORBA::WChart_out

注記　長精度型 (long) が 64 ビットのマシンでも、CORBA::Long の定義は 32 ビット整数を参照します。

複雑なデータ型

表 0-2 に、オブジェクト、擬似オブジェクト、およびユーザ定義型のマッピングを示します。

表 0-2 オブジェクト、擬似オブジェクト、およびユーザ定義の OMG IDL と C++ の型

OMG IDL	C++
`Object`	`CORBA::Object_ptr`
`struct`	`C++ struct`
`union`	`C++ class`
`enum`	`C++ enum`
`string`	`char *`
`wstring`	`CORBA::WChar *`
`sequence`	`C++ class`
`array`	`C++ array`

文字列および UDT のマッピングの詳細については、以降の節を参照してください。

文字列

OMG IDL の文字列は、C++ の `char *` にマッピングされます。`char *` には、バウンディッド文字列とアンバウンディッド文字列の両方がマッピングされます。C++ の CORBA 文字列は、NULL で終了し、char * 型の使用時には常に使用できます。

`struct` など、ほかのユーザ定義の型の中に文字列がある場合は、`CORBA::String_var` 型にマッピングされます。これにより、構造体内の各メンバは自身のメモリを管理できます。

文字列の割り当ておよび割り当て解除には、CORBA クラスの次のメンバ関数を使用する必要があります。

- string_alloc
- string_dup
- string_free

注記　string_alloc 関数によって len+1 文字が割り当てられるので、結果として得られた文字列には後続の NULL 文字を保持するのに十分なスペースを確保できます。

wchar

OMG IDL では、任意の文字セットからワイド文字を符号化する wchar データ型を定義します。文字データと同じく、インプリメンテーションでは任意のコード・セットを使用してワイド文字を符号化します。ただし、転送にはほかの形式への変換が必要になる場合もあります。wchar のサイズは、インプリメンテーションによって異なります。

次に、wchar を定義する構文を示します。

```
<wide_char_type> ::= “wchar”
```

次に、wchar のコード例を示します。

```
wchar_t wmixed[256];
```

注記　wchar および wstring データ型を使用すると、ユーザが記述するネイティブ言語でコンピュータとやり取りできます。日本語や中国語など、言語によっては固有の文字が多数ある言語もあります。このような文字セットは、1 バイトの中に収まりません。マルチ・バイト文字セットをサポートするためにさまざまなスキーマが考案されていますが、まだ実用的なレベルには至っていません。ワイド文字とワイド文字列を使用すると、こういった複雑な文字セットとのやり取りが容易になります。

wstring

wstring データ型は、ワイド文字 NULL を除き wchar のシーケンスを表します。wstring 型は string 型と似ていますが、その要素型が char ではなく wchar である点で異なります。wstring の実際の長さは実行時に設定されます。バウンディッド形式を使用する場合、長さはバウンド以下でなければなりません。

次に、wstring を定義する構文を示します。

```
<wide_string_type> ::= “wstring” “<” <positive_int_const> “>”
                     | “wstring
```

次に、wstring のコード例を示します。

```
CORBA::WString_var v_upper = CORBA::wstring_dup(wmixed);
```

wstring 型は、unsigned long、char、string、double などの型とまったく同様に作成します。これらの型は、パラメータとして直接使用したり、typedef で指定したり、構造体、シーケンス、共用体、配列などの作成に使用したりできます。

注記　wchar および wstring データ型を使用すると、ユーザが記述するネイティブ言語でコンピュータとやり取りできます。日本語や中国語など、言語によっては固有の文字が多数ある言語もあります。このような文字セットは、1 バイトの中に収まりません。マルチ・バイト文字セットをサポートするためにさまざまなスキーマが考案されていますが、まだ実用的なレベルには至っていません。ワイド文字とワイド文字列を使用すると、こういった複雑な文字セットとのやり取りが容易になります。

定数

OMG IDL の定数は、C++ の const 定義にマッピングされます。次に、OMG IDL 定義の例を示します。

```
// OMG IDL

const string CompanyName = “BEA Systems Incorporated”;
```

```
module INVENT
   {
    const string Name = “Inventory Modules”;

    interface Order
        {
        const long MAX_ORDER_NUM = 10000;
        };
    };
```

上の定義は、次のように C++ にマッピングされます。

```
// C++

const  char *const
   CompanyName  = “BEA Systems Incorporated”;
. . .
class INVENT
    {
    static const  char *const  Name;
    . . .

    class Order : public virtual CORBA::Object
        {
        static const CORBA::Long MAX_ORDER_NUM;
        . . .
        };
     };
```

最上位の定数は、生成された .h インクルード・ファイルで初期化されます。ただし、モジュールとインターフェイスの各定数は、生成されたクライアント・スタブ・モジュールで初期化されます。

次に、前の例で定義した MAX_ORDER_NUM 定数への有効なリファレンスの例を示します。

```
CORBA::Long accnt_id = INVENT::Order::MAX_ORDER_NUM;
```

Enum

OMG IDL の enum は、C++ の enum にマッピングされます。次に、OMG IDL 定義の例を示します。

```
// OMG IDL
```

```
module INVENT
    {
    enum Reply {ACCEPT, REFUSE};
     }
```

上の定義は、次のように C++ にマッピングされます。

```
// C++

class INVENT
   {
   . . .

    enum  Reply {ACCEPT, REFUSE};
    };
```

次に、前の例で定義した enum への有効なリファレンスの例を示します。enum へのリファレンスは次のとおりです。

```
INVENT::Reply accept_reply;
accept_reply = INVENT::ACCEPT;
```

構造体

OMG IDL の構造体は、C++ の構造体にマッピングされます。

構造体に対して生成されたコードは、固定長か可変長かによって異なります。固定長型と可変長型の詳細については、「固定長ユーザ定義型と可変長ユーザ定義型」を参照してください。

固定長構造体と可変長構造体

可変長構造体には、代入演算子メンバ関数が別にあり、2 つの可変長構造体間の代入を処理します。

次に、OMG IDL 定義の例を示します。

```
// OMG IDL

module INVENT
   {
    // 固定長
    struct Date
        {
```

```
            long year;
            long month;
            long day;
            };

        // 可変長
        struct Address
            {
            string  aptNum;
            string  streetName;
            string  city;
            string  state;
            string  zipCode;
             };
            };
```

上の定義は、次のように C++ にマッピングされます。

```
// C++

class INVENT
     {
     struct  Date
        {
        CORBA::Long year;
        CORBA::Long month;
        CORBA::Long day;
         };

     struct  Address
        {
         CORBA::String_var aptNum;
         CORBA::String_var streetName;
         CORBA::String_var city;
         CORBA::String_var state;
         CORBA::String_var zipCode;
         Address &operator=(const Address obj);
         };

      };
```

メンバのマッピング

構造体のメンバは、適切な C++ データ型にマッピングされます。long、short などの基本データ型 については、表 0-1 を参照してください。オブジェクト・リファレンス、擬似オブジェクト・リファレンス、および文字列については、メンバは次の適切な var クラスにマッピングされます。

- `CORBA::String_var`
- `CORBA::Object_var`

その他のデータ型については、表 0-2 を参照してください。

生成される構造体にはコンストラクタがないため、メンバは初期化されません。固定長構造体の場合、集約初期化を行うと初期化できます。たとえば、次のように入力します。

```
INVENT::Date a_date = { 1995, 10, 12 };
```

可変長メンバの場合、自己管理型にマッピングします。この型には、メンバを初期化するコンストラクタが含まれています。

Var

構造体に対して var クラスが生成されます。詳細については、var クラスの使い方を参照してください。

Out

構造体に対して out クラスが生成されます。詳細については、out クラスの使い方を参照してください。

共用体

OMG IDL の共用体は、C++ のクラスにマッピングされます。C++ クラスには、次のものが格納されています。

- コンストラクタ
- デストラクタ
- 代入演算子
- 共用体の値のモディファイア
- 共用体の値のアクセサ
- 共用体の区別子のモディファイアおよびアクセサ

次に、OMG IDL 定義の例を示します。

```
// OMG IDL

union OrderItem switch (long)
    {
    case 1: itemStruct itemInfo;
    case 2: orderStruct orderInfo;
    default: ID idInfo;
    };
```

上の定義は、次のように C++ にマッピングされます。

```
// C++

class OrderItem
    {
    public:
       OrderItem();
       OrderItem(const OrderItem &);
       ~OrderItem();

        OrderItem &operator=(const OrderItem&);

       void  _d  (CORBA::Long);
       CORBA::Long  _d  () const;

       void itemInfo (const itemStruct &);
       const itemStruct & itemInfo () const;
       itemStruct & itemInfo ();

       void orderInfo (const orderStruct &);
       const orderStruct & orderInfo () const;
       orderStruct & orderInfo ();

       void idInfo (ID);
       ID idInfo () const;

       . . .
     };
```

デフォルトの共用体コンストラクタでは、共用体のデフォルト区別子の値が設定されていません。そのため、共用体の値を設定するまでは、すべての共用体アクセサ・メンバ関数を呼び出すことができません。区別子は、_d メンバ関数でマッピングされる属性です。

共用体メンバのアクセサ・メンバ関数とモディファイア・メンバ関数のマッピング

共用体のメンバごとに、アクセサ・メンバ関数とモディファイア・メンバ関数が生成されます。

前の例から抜粋した次のコードでは、2 つのメンバ関数は ID メンバ関数に対して生成されています。

```
void idInfo (ID);
ID idInfo () const;
```

この例では、最初の関数 (モディファイア) で区別子をデフォルト値に、共用体の値を指定の ID 値に設定しています。2 番目の関数 (アクセサ) では、共用体の値を返しています。

共用体メンバのデータ型によっては、モディファイア関数が追加生成されます。各データ型に対して生成されるメンバ関数は次のとおりです。

- 基本データ型 —short、long、unsigned short、unsigned long、float、double、char、boolean、および octet

  次の例では、メンバ名 `basictype` で基本データ型の 2 つのメンバ関数を生成しています。

  ```
  void basictype (TYPE);      // モディファイア
  TYPE basictype () const;    // アクセサ
  ```

  OMG IDL のデータ型から C++ のデータ型 `TYPE` へのマッピングについては、表 0-1 を参照してください。

- オブジェクトおよび擬似オブジェクト

  メンバ名が `objtype` のオブジェクト型および Typecode 型の場合、メンバ関数は次のように生成されます。

  ```
  void objtype (TYPE);      // モディファイア
  TYPE objtype () const;    // アクセサ
  ```

  OMG IDL のデータ型から C++ のデータ型 `TYPE` へのマッピングについては、表 0-1 を参照してください。

モディファイア・メンバ関数では、指定されたオブジェクト・リファレンス引数の所有権を想定していません。代わりに、モディファイアはオブジェクト・リファレンスまたは擬似オブジェクト・リファレンスを複製します。リファレンスが不要になったときは、リファレンスを解放してください。

- Enum

  メンバ名が enumtype の enum TYPE の場合、メンバ関数は次のように生成されます。

```
void enumtype (TYPE);       // モディファイア
TYPE enumtype () const;  // アクセサ
```

- 文字列

  文字列の場合、次に示すようにアクセサ関数が 1 つとモディファイア関数 3 つが生成されます。

```
void stringInfo (char *);                          // モディファイア 1
void stringInfo (const char *);                    // モディファイア 2
void stringInfo (const CORBA::String_var &);  // モディファイア 3
const char * stringInfo () const;                 // アクセサ
```

  最初のモディファイアでは、渡された char * パラメータの所有権を想定しています。また、共用体の値に変更があったり、共用体が破棄されたときに、共用体は CORBA::string_free メンバ関数を呼び出します。

  2 番目と 3 番目のモディファイアでは、パラメータに渡されるか、または文字列 var に格納されている指定の文字列をコピーします。

  アクセサ関数は、共用体の内部メモリへのポインタを返します。また、このメモリの解放を試行せず、共用体の値に変更があったり、共用体が破棄されたりした後は、このメモリにアクセスしません。

- 構造体、共用体、シーケンス、および Any

これらのデータ型の場合、次に示すようにモディファイアとアクセサは、type へのリファレンスで生成されます。

```
void reftype (TYPE &);          // モディファイア
const TYPE & reftype () const; // アクセサ
TYPE & reftype ();              // アクセサ
```

モディファイア関数は、入力パラメータ type の所有権を想定していません。代わりに、関数はデータ型をコピーします。

- 配列

配列の場合、次に示すように、モディファイア・メンバ関数は配列ポインタを受け付け、アクセサは配列スライスへのポインタを返します。

```
void arraytype (TYPE);      // モディファイア
TYPE_slice * arraytype () const;  // アクセサ
```

モディファイア関数は、入力パラメータ type の所有権を想定していません。代わりに、関数は配列をコピーします。

Var

共用体に対して var クラスが生成されます。詳細については、var クラスの使い方を参照してください。

Out

共用体に対して out クラスが生成されます。詳細については、out クラスの使い方を参照してください。

メンバ関数

アクセサとモディファイアのほかに、型 TYPE の OMG IDL 共用体に対して switch (long) 区別子で次のメンバ関数が生成されます。

`TYPE();`
: これは、共用体のデフォルト・コンストラクタです。この関数ではデフォルトの区別子が設定されないため、共用体の値を設定するまでは共用体にアクセスできません。

`TYPE( const TYPE & From);`
: このコピー・コンストラクタは、指定の共用体をディープ・コピーします。共用体パラメータの任意の値をコピーします。`From` 引数には、コピー元の共用体を指定します。

`~TYPE();`
: このデストラクタは、共用体に関連付けられたデータを解放します。

`TYPE &operator=(const TYPE & From);`
: この代入演算子は指定の共用体をコピーします。現在の共用体にある既存の値は解放されます。`From` 引数には、コピー元の共用体を指定します。

`void _d (CORBA::Long Descrim);`
: このメンバ関数は、区別値を設定し、現在の値を解放します。`Descrim` 引数には、新しい区別値を指定します。引数のデータ型は、共用体の switch 文で指定した OMG IDL のデータ型で決まります。OMG IDL および C++ の各データ型については、表 0-1 を参照してください。

`CORBA::Long _d () const;`
: この関数は現在の区別値を返します。戻り値のデータ型は、共用体の switch 文で指定した OMG IDL のデータ型で決まります。OMG IDL および C++ の各データ型については、表 0-1 を参照してください。

シーケンス

OMG IDL のシーケンスは、C++ のクラスにマッピングされます。C++ クラスには、次のものが格納されています。

- コンストラクタ

  各シーケンスには、次のものがあります。

  - デフォルト・コンストラクタ

- 各要素を初期化するコンストラクタ
- コピー・コンストラクタ

■ デストラクタ

■ 現在の長さ (およびシーケンスがアンバウンディッドの場合は最大の長さ) のモディファイア

■ 現在の長さのアクセサ

■ シーケンス要素にアクセスしたり、シーケンス要素を変更する Operator[] 関数

■ メンバ関数の割り当ておよび割り当て解除

要素にアクセスする場合、事前に長さを設定しておく必要があります。

次に、OMG IDL 定義の例を示します。

```
// OMG IDL

module INVENT
    {
    . . .
    typedef sequence<LogItem>       LogList;
    }
```

上の定義は、次のように C++ にマッピングされます。

```
// C++

class LogList
    {
    public:
        // デフォルト・コンストラクタ
        LogList();

        // 最大コンストラクタ
        LogList(CORBA::ULong _max);

       // TYPE * データ・コンストラクタ
        LogList
        (
            CORBA::ULong _max,
            CORBA::ULong _length,
            LogItem *_value,
```

```
            CORBA::Boolean _relse = CORBA_FALSE
        );

        // コピー・コンストラクタ
         LogList(const LogList&);

        // デストラクタ
        ~LogList();

        LogList &operator=(const LogList&);

        CORBA::ULong maximum() const;

        void length(CORBA::ULong);
        CORBA::ULong length() const;

        LogItem &operator[](CORBA::ULong _index);
        const LogItem &operator[](CORBA::ULong _index) const;

        static LogItem *allocbuf(CORBA::ULong _nelems);
        static void freebuf(LogItem *);
        };

    };
```

シーケンス要素のマッピング

operator[] 関数は、シーケンス要素へのアクセスまたはシーケンス要素の変更に使用します。この演算子は、シーケンス要素にリファレンスを返します。OMG IDL の基本型は、C++ の適切なデータ型にマッピングされます。

基本データ型については、表 0-1 を参照してください。オブジェクト・リファレンス、TypeCode リファレンス、および文字列の場合、基本型は生成された _ForSeq_var クラスにマッピングされます。_ForSeq_var クラスは、文字列またはシーケンス内に格納されたオブジェクトを更新する機能を備えています。このクラスには、対応する var クラスと同じメンバ関数とシグニチャがあります。ただし、_ForSeq_var クラスは、シーケンス・コンストラクタの release パラメータの設定に従います。ほかのクラスとの相違点は、_ForSeq_var クラスの場合、ユーザが Release フラグを指定できるので、メモリの解放を制御できることです。

その他のデータ型については、表 0-2 を参照してください。

Vars

シーケンスに対して var クラスが生成されます。詳細については、var クラスの使い方を参照してください。

Out

シーケンスに対して out クラスが生成されます。詳細については、out クラスの使い方を参照してください。

メンバ関数

以下では、基本型が TYPE で指定の OMG IDL シーケンスが SEQ の場合に、生成されるシーケンス・クラスのメンバ関数について説明します。

```
SEQ ();
```

これは、シーケンスのデフォルト・コンストラクタです。長さは 0 (ゼロ) に設定されています。シーケンスがアンバウンディッドの場合でも、最大値は 0 (ゼロ) に設定されます。シーケンスがバウンディッドの場合は、最大値は OMG IDL の型で指定され、変更はできません。

```
SEQ (CORBA::ULong Max);
```

このコンストラクタは、シーケンスがアンバウンディッドの場合にのみ指定します。この関数は、シーケンスの長さを 0 (ゼロ) に設定し、バッファの最大値を指定の値に設定します。Max 引数には、シーケンスの最大長を指定します。

```
SEQ (CORBA::ULong Max, CORBA::ULong Length, TYPE * Value,
      CORBA::Boolean Release);
```

このコンストラクタは、シーケンスの最大値、長さ、および要素を設定します。シーケンスが破棄されたときに要素を解放するかどうかは、Release フラグで指定します。各引数について次に説明します。

Max

シーケンスの最大値。バウンディッド・シーケンスでは、この引数はありません。

Length
: シーケンスの現在の長さ。バウンディッド・シーケンスの場合、この値は OMG IDL の型で指定した最大値より小さい値を指定する必要があります。

Value
: シーケンスの要素を格納するバッファへのポインタ。

Release
: 要素を解放するかどうかを指定します。このフラグの値が CORBA_TRUE の場合、シーケンスは Value 引数が指すバッファの所有権を想定します。Release フラグが CORBA_ TRUE の場合、このバッファは allocbuf メンバ関数で割り当てる必要があります。これは、シーケンスの破棄時に、このバッファを freebuf メンバ関数で解放するためです。

```
SEQ(const S& From);
```
このコピー・コンストラクタは、指定の引数からシーケンスをディープ・コピーします。From 引数には、コピー元のシーケンスを指定します。

```
~SEQ();
```
このデストラクタは、シーケンスを解放します。また、Release フラグの値によっては、シーケンス要素を解放する場合もあります。

```
SEQ& operator=(const SEQ& From);
```
この代入演算子は、指定のシーケンス引数からシーケンスをディープ・コピーします。現在のシーケンスの Release フラグが CORBA_TRUE の場合、現在のシーケンスにある既存の要素はすべて解放されます。From 引数には、コピー元のシーケンスを指定します。

```
CORBA::ULong maximum( ) const;
```
この関数は、シーケンスの最大値を返します。バウンディッド・シーケンスの場合、OMG IDL の型で設定した値になります。アンバウンディッド・シーケンスの場合は、シーケンスの現在の最大値になります。

```
void length(CORBA::ULong Length);
```
この関数は、シーケンスの現在の長さを設定します。Length 引数には、シーケンスの新しい長さを指定します。シーケンスがアンバウンディッドで新しい長さが現在の最大値を超える場合、バッファが再割り当てされ、要素が新しいバッファにコピーされます。新しい長さが最大値を超える場合、最大値は新しい長さに設定されます。

バウンディッド・シーケンスの場合は、最大値を超える値には長さを設定できません。

`CORBA::ULong length() const;`
この関数は、シーケンスの現在の長さを返します。

```
TYPE & operator[](CORBA::ULong Index);
const TYPE & operator[](CORBA::ULong Index) const;
```
上記のアクセサ関数は、指定のインデックスでシーケンス要素へのリファレンスを返します。Index 引数には、戻り値となる要素のインデックスを指定します。このインデックスは、現在のシーケンスの長さを超えることはできません。長さは、TYPE * コンストラクタまたは length(CORBA::ULong) モディファイアで設定済みである必要があります。TYPE がオブジェクト・リファレンス、TypeCode リファレンス、または文字列の場合、戻り値の型は ForSeq_var クラスになります。

`static TYPE * allocbuf(CORBA::ULong NumElems);`
この静的関数は、TYPE * コンストラクタで使用されるバッファを割り当てます。NumElems 引数には、バッファ内の割り当てる要素数を指定します。バッファの割り当てができない場合、NULL が返されます。

release が CORBA_TRUE に設定されていて、このバッファが TYPE * に渡されない場合は、freebuf メンバ関数で解放する必要があります。

`static void freebuf(TYPE * Value);`
この静的関数は、allocbuf 関数で割り当てられた TYPE * シーケンス・バッファを解放します。Value 引数には、allocbuf 関数で割り当てられた TYPE * バッファを指定します。0 (ゼロ) ポインタは無視されます。

配列

OMG IDL の配列は、C++ の 配列定義にマッピングされます。次に、OMG IDL 定義の例を示します。

```
// OMG IDL
```

```
module INVENT
    {
    . . .
    typedef LogItem   LogArray[10];
    };
```

上の定義は、次のように C++ にマッピングされます。

```
// C++

module INVENT
    {
    . . .
    typedef LogItem LogArray[10];
    typedef LogItem LogArray_slice;
    static LogArray_slice * LogArray_alloc(void);
    static void LogArray_free(LogArray_slice *data);

     };
```

配列スライス

配列のスライスとは、元の配列の最初のサイズを除いたすべてのサイズを持つ配列のことです。配列で生成されたクラスのメンバ関数は、スライスへのポインタを使用して配列へのポインタを返します。スライスごとに typedef が生成されます。

次に、OMG IDL 定義の例を示します。

```
// OMG IDL
typedef LogItem         LogMultiArray[5][10];
```

上の定義は、次のように C++ にマッピングされます。

```
// C++
typedef LogItem          LogMultiArray[5][10];
typedef LogItem           LogMultiArray_slice[10];
```

配列のサイズが 1 つの場合、配列スライスは型だけになります。たとえば、サイズが 1 つの `long` の配列がある場合、配列スライスは `CORBA::Long` データ型になります。

配列要素のマッピング

OMG IDL の配列の型は、構造体と同じ方法で C++ の配列要素型にマッピングされます。詳細については、メンバのマッピングを参照してください。

Var

配列に対して var クラスが生成されます。詳細については、var クラスの使い方を参照してください。

Out

配列に対して out クラスが生成されます。詳細については、out クラスの使い方を参照してください。

割り当てメンバ関数

各配列には、配列の割り当ておよび割り当て解除を行う 2 つの静的関数があります。指定の OMG IDL の型 TYPE の場合、割り当てルーチンと割り当て解除ルーチンは次のとおりです。

`static TYPE_slice * TYPE_alloc(void);`
: この関数は、TYPE 配列を割り当て、割り当て済みの TYPE 配列へのポインタを返します。配列を動的に割り当てることができない場合、0 (ゼロ) を返します。

`static void TYPE_free(TYPE_slice * Value);`
: この関数は、動的に割り当てられた TYPE 配列を解放します。Value 引数は、解放元の動的に割り当てられた TYPE 配列へのポインタです。

例外

OMG IDL の例外は、C++ のクラスにマッピングされます。C++ クラスには、次のものが格納されています。

- コンストラクタ

- デストラクタ
- 例外型を判別する静的 _narrow 関数

生成されるクラスは可変長構造体と似ています。ただし、このクラスは初期化を簡単にするためのコンストラクタ、および UserException の型を判別するための静的 _narrow メンバ関数が追加されている点で異なります。

次に、OMG IDL 定義の例を示します。

```
// OMG IDL

module INVENT
    {
    exception NonExist
        {
        ID BadId;
        };
     };
```

上の定義は、次のように C++ にマッピングされます。

```
// C++

class INVENT
    {
     . . .

    class NonExist : public CORBA::UserException
        {
        public:
            static NonExist * _narrow(CORBA::Exception_ptr);
            NonExist (ID _BadId);
            NonExist ();
            NonExist (const NonExist &);
            ~NonExist ();
            NonExist & operator=(const NonExist &);
            void _raise ();
            ID BadId;
        };
    };
```

Exception クラスの属性 (データ・メンバ) はパブリックなので、直接アクセスできます。

メンバのマッピング

例外のメンバは、構造体と同じ方法でマッピングされます。詳細については、メンバのマッピングを参照してください。

例外メンバはすべて、C++ ではパブリック・データなので、直接アクセスできます。

Var

例外に対して var クラスが生成されます。詳細については、var クラスの使い方を参照してください。

Out

例外に対して out クラスが生成されます。詳細については、out クラスの使い方を参照してください。

メンバ関数

指定の OMG IDL 例外 TYPE の場合、生成されるメンバ関数は次のとおりです。

```
static TYPE * _narrow(CORBA::Exception_ptr Except);
```

この関数は、例外が TYPE 例外にナロー変換できる場合に、TYPE 例外へのポインタを返します。例外がナロー変換できない場合は、0 (ゼロ) を返します。TYPE ポインタは、新しいクラスへのポインタではなく、元の例外ポインタへの型付きポインタです。これは、Except パラメータが有効な限りにおいて有効です。

```
TYPE ( );
```

これは、例外のデフォルト・コンストラクタです。メンバが固定長の場合、メンバは初期化されません。可変長メンバの場合、自己管理型にマッピングします。この型には、メンバを初期化するコンストラクタが含まれています。

```
TYPE(member-parameters);
```

このコンストラクタには、例外のメンバごとに引数があります。このコンストラクタは、各引数をコピーし、各引数のメモリの所有権

を想定しません。前の例外を作成すると、そのコンストラクタのシグニチャは次のようになります。

```
NonExist (ID _BadId);
```

例外のメンバごとに引数が 1 つあります。型とパラメータ渡しのメカニズムは、Any の挿入演算子と同じです。Any の挿入演算子の詳細については、「Any への挿入」を参照してください。

```
TYPE (const TYPE & From);
```

このコピー・コンストラクタは、指定の TYPE 例外引数からデータをコピーします。From 引数には、コピー元の例外を指定します。

```
~TYPE ();
```

このデストラクタは、例外に関連付けられたデータを解放します。

```
TYPE & operator=(const TYPE & From);
```

この代入演算子は、指定の TYPE 例外引数からデータをコピーします。From 引数には、コピー元の例外を指定します。

```
void  _raise ();
```

この関数によって、例外インスタンスが自己に対してスローされます。catch 句は、
派生した型で例外インスタンスをキャッチできます。

擬似オブジェクトの C++ へのマッピング

CORBA 擬似オブジェクトは、通常の CORBA オブジェクトまたはサーバレス・オブジェクトとしてインプリメントできます。CORBA 仕様における、両者の基本的な違いは次のとおりです。

- サーバレス・オブジェクト型は CORBA::Object を継承しません。
- 個々のサーバレス・オブジェクトは、ORB に登録されません。
- サーバレス・オブジェクトの場合、IDL 型と同じメモリ管理規則に従う必要はありません。

サーバレス・オブジェクトへのリファレンスは、アドレス空間などのコンピュータ間のコンテキストで有効とは限りません。代わりに、パラメータとして渡されるサーバレス・オブジェクトへのリファレンスでは、このリファレンスの受信側が使用するオブジェクトの、非依存的で機能的に同じコピー

が作成されます。これをサポートするには、データ・レイアウトなど、サーバレス・オブジェクトのほかの隠れた表現プロパティを ORB に認識させます。この章では、ORB にサーバレス・オブジェクトを認識させる仕様については、インプリメンテーション上の詳細事項のため説明は割愛します。

この章では、代わりにすべての擬似オブジェクト型の標準的なマッピング・アルゴリズムについて説明します。これにより、9 種類の CORBA 擬似オブジェクト型ごとのマッピングについて、説明が断片的にならずに済み、CORBA の将来のリビジョンで提供される擬似オブジェクト型にも対応できます。また、C マッピングの表現に依存せずに、C 互換の表現に依存したインプリメンテーションを実現することができます。

形式

C PIDL とは異なり、このマッピングでは、完全な OMG IDL の拡張形式を使用してサーバレス・オブジェクト型を記述します。擬似オブジェクト型のインターフェイスでは、通常の OMG IDL インターフェイスと同じ規則に従います。ただし、次の例外があります。

- 先頭にキーワード `pseudo` が付きます。
- 通常の場合では OMG IDL で許可されていない、ほかのサーバレス・オブジェクト型[1] を宣言で参照する場合があります。

`pseudo` 接頭辞は、そのインターフェイスが通常の方法またはサーバレスの方法のどちらかでインプリメントされることを示します。つまり、以降の節で説明する規則か、またはこの節で説明する通常のマッピング規則のどちらかを適用することになります。

マッピング規則

サーバレス・オブジェクトは、この節で概説する相違点を除いて、通常のインターフェイスと同じ方法でマッピングされます。

1. 特に、データ型および関数名として使用する `exception`。

サーバレス・オブジェクト型を表すクラスは、CORBA::Object のサブクラスではありません。また、ほかの C++ クラスのサブクラスである必要はありません。したがって、Object::create_request などのオペレーションを必ずしもサポートしません。

サーバレス・オブジェクト型の T を表す各クラスの場合、次の関数のオーバーロードの各バージョンが CORBA 名前空間で提供されます。

```
// C++
void release(T_ptr);
Boolean is_nil(T_ptr p);
```

サブクラスはインプリメンテーションで提供できます。ただし、マッピングされた C++ クラスは、ユーザが容易にサブクラス化できるという保証はありません。インプリメンテーションでは、サブクラスに適用されない内部表現や転送形式を想定することは可能です。

サーバレス・オブジェクト型を表すクラスのメンバ関数は、通常のメモリ管理規則に従うとは限りません。これは、CORBA::NVList などの一部のサーバレス・オブジェクトが、本質的にほかのサーバレス・オブジェクトのいくつかのレベルのコンテナでしかない場合があるためです。格納されたサーバレス・オブジェクトのアクセサ関数から返された値を、呼び出し側が明示的に解放しなければならないということは、本来の目的とは反対の使い方になってしまいます。

マッピングのその他の要素はすべて、次に示すように同じです。

- サーバレス・オブジェクトへのリファレンスの型である T_ptr は、単に T* の typedef である必要はありません。
- マッピングされた各クラスは、次の静的メンバ関数をサポートします。

  ```
  // C++
  static T_ptr _duplicate(T_ptr p);
  static T_ptr _nil();
  ```

- _duplicate の有効なインプリメンテーションでは、単純に引数を返すか、または新しいインスタンスへのリファレンスを作成します。個別にインプリメンテーションを行うことで、動作の確実性が高くなることがあります。
- 対応する C++ クラスは、直接インスタンス化可能である必要はありません。また、ほかのインスタンス化の制約があってもなくてもかまいませ

ん。移植性を考慮して、ユーザは適切な構成オペレーションを呼び出す必要があります。

- 通常のインターフェイスと同じく、代入演算子はサポートされません。
- 「リファレンス・スタイル」ではなく「コピー・スタイル」を透過的に採用していますが、シグニチャを渡すパラメータ、およびメモリ管理規則を含めた規則は、この節で注記した点を除いて通常のオブジェクトの場合と同じです。

C PIDL マッピングとの関係

サーバレス・オブジェクトのインターフェイスと宣言がこれらに依存する場合、すべて C マッピングに直接類似しています。マッピングされた C++ クラスは、必要に応じて、C マッピング用に選択したクラスと互換性のある表現を使用してインプリメントすることができます。C PIDL と C++ PIDL の擬似オブジェクト仕様の相違点は次のとおりです。

- C++ PIDL では、構造体および typedef ではなくインターフェイスを使用して、表現の依存関係を解消する必要があります。
- C++ PIDL では、擬似オブジェクトのインターフェイスにそのオペレーションを格納する必要があります。場合によっては、この処理に対応するよう機能を再指定することもあります。
- C++ PIDL では、その他の指定がない限り、`release` は、関連付けられた C マッピングの `free` および `delete` オペレーションの役割を果たします。

以降の節では、各擬似インターフェイスとその C++ マッピングの概説と一覧を示します。以下で定義が行われない型も含め、詳細については、このマニュアルの関連する節を参照してください。

Typedef

OMG IDL の typedef は、C++ の typedef にマッピングされます。OMG IDL のデータ型によっては、追加の typedef およびメンバ関数が定義される場合もあります。各データ型に対して生成されるコードは次のとおりです。

- 基本データ型 (short、long、unsigned short、unsigned long、float、double、char、boolean、および octet)

  基本データ型は単純な typedef にマッピングされます。たとえば、次のように入力します。

  ```
  // OMG IDL
  typedef long ID;

  // C++
  typedef CORBA::Long ID;
  ```

- 文字列

  文字列の typedef は単純な typedef にマッピングされます。たとえば、次のように入力します。

  ```
  // OMG IDL
  typedef string IDStr;

  // C++
  typedef char * IDStr;
  ```

- オブジェクト、インターフェイス、TypeCode

  オブジェクト、インターフェイス、および TypeCode は、4 つの typedef にマッピングされます。たとえば、次のように入力します。

  ```
  // OMG IDL
  typedef Item Intf;

  // C++
  typedef Item Intf;
  typedef Item_ptr Intf_ptr;
  typedef Item_var Intf_var;
  typedef Item_ptr & Intf _out;
  ```

- enum、構造体、共用体、シーケンス

  UDT は 3 つの typedef にマッピングされます。たとえば、次のように入力します。

```
// OMG IDL
typedef LogList ListRetType;

// C++
typedef LogList ListRetType;
typedef LogList_var ListRetType_var;
typedef LogList_out & ListRetType_out;
```

- 配列

  配列は、メモリを割り当ておよび解放するために 4 つの typedef と静的メンバ関数にマッピングされます。たとえば、次のように入力します。

```
// OMG IDL
typedef LogArray ArrayRetType;

// C++
typedef LogArray ArrayRetType;
typedef LogArray_var ArrayRetType_var;
typedef LogArray_forany ArrayRetType_forany;
typedef LogArray_slice ArrayRetType_slice;
ArrayRetType_slice * ArrayRetType_alloc();
void ArrayRetType_free(ArrayRetType_slice *);
```

インターフェイスのインプリメント

OMG IDL のオペレーションは、C++ のメンバ関数にマッピングされます。

メンバ関数の名前がオペレーションの名前になります。オペレーションは、インターフェイス・クラスとスタブ・クラスの両方でメンバ関数として定義します。インターフェイス・クラスは仮想クラスです。一方、スタブ・クラスは仮想クラスを継承し、クライアント・アプリケーション・スタブのメンバ関数のコードを格納します。オブジェクト・インターフェイスでオペレーションが呼び出されると、対応するスタブのメンバ関数に格納されているコードが実行されます。

次に、OMG IDL 定義の例を示します。

```
// OMG IDL
```

```
module INVENT
    {
     interface Order
        {
        . . .
        ItemList modifyOrder (in ItemList ModifyList);
        };
    };
```

上の定義は、次のように C++ にマッピングされます。

```
// C++

class INVENT
    {
    . . .

    class Order : public virtual CORBA::Object
        {
        . . .
        virtual ItemList * modifyOrder (
            const ItemList & ModifyList) = 0;
        };
    };

class Stub_Order : public Order
    {
    . . .
    ItemList * modifyOrder (
        const ItemList & ModifyList);
    };
```

生成されたクライアント・アプリケーション・スタブには、スタブ・クラスに対して生成された次のコードが格納されます。

```
// ルーチン名 :    INVENT::Stub_Order::modifyOrder
//
// 関数の説明 :
//
// オペレーション modifyOrder の
// クライアント・アプリケーション・スタブ
// (Interface : Order)

INVENT::ItemList * INVENT::Stub_Order::modifyOrder (
     const INVENT::ItemList & ModifyList)
{
. . .
}
```

引数のマッピング

オペレーションの各引数は、表 0-1 および 表 0-2 で説明した、対応する C++ の型にマッピングされます。

オペレーションでの引数のパラメータ渡しモードについては、表 0-7 および 表 0-8 を参照してください。

オペレーションのインプリメント

インプリメンテーション・メンバ関数のシグニチャは、OMG IDL オペレーションのマッピングされたシグニチャです。クライアント側と異なり、サーバ側のマッピングでは、関数ヘッダに適切な例外 (throw) 指定を含める必要があります。これにより、いつ無効な例外が発生したかをコンパイラで検出できるようになります。これは、ローカル C++ から C++ ライブラリへの呼び出しを行う場合に必要です。この作業を行わない場合、呼び出しでは有効な例外をチェックするラッパーを通過してしまいます。たとえば、次のように入力します。

```
// IDL
interface A
{
exception B {};
void f() raises(B);
};

// C++
class MyA : public virtual POA_A
{
  public:
void f() throw(A::B, CORBA::SystemException);
...
};
```

すべてのオペレーションおよび属性が原因で CORBA システム例外は発生するため、オペレーションに `raises` 句がない場合でも、すべての例外指定で `CORBA::SystemException` を含める必要があります。

メンバ関数内で、「this」ポインタは、クラスで定義されたインプリメンテーション・オブジェクトのデータを参照します。データにアクセスするだけでなく、メンバ関数は同じクラスで定義されたほかのメンバ関数を暗黙的に呼び出すこともできます。たとえば、次のように入力します。

```
// IDL

interface A
{
void f();
void g();
};



// C++
class MyA : public virtual POA_A
{
  public:

void f() throw(SystemException);
void g() throw(SystemException);
  private:
long x_;
};

void
MyA::f() throw(SystemException)
{
this->x_ = 3;
this->g();
}
```

ただし、この方法でサーバント・メンバ関数が呼び出されるときは、CORBA オブジェクトのオペレーションのインプリメンテーションとしてではなく、単純に C++ のメンバ関数として呼び出されます。このような場合、`POA_Current` オブジェクトで利用可能な情報はすべて、メンバ関数の呼び出し自体ではなく、C++ メンバ関数の呼び出しを実行した CORBA 要求の呼び出しを参照します。

オブジェクトからのスケルトンの派生

一部の既存の ORB インプリメンテーションでは、対応するインターフェイス・クラスから、それぞれスケルトン・クラスが派生します。たとえば、インターフェイスが `Mod::A` の場合、スケルトン・クラス `POA_Mod::A` が

Mod::A から派生します。したがって、これらのシステムでは、標準の C++ 派生ベースの変換規則によって暗黙的にサーバントをオブジェクト・リファレンスで取得できます。

```
// C++
MyImplOfA my_a;           // A のインプリメンテーションを宣言
A_ptr a = &my_a;    // C++ 派生ベースの規則で
                    // そのオブジェクト・リファレンスを取得
```

上のようなコードは、ORB 準拠のインプリメンテーションでサポートしていますが、必須ではありません。そのため、移植性はありません。移植性のある同等のコードでは、インプリメンテーション・オブジェクトで _this() を呼び出し、未登録の場合は暗黙的にそれを登録して、そのオブジェクト・リファレンスを取得します。

```
// C++
MyImplOfA my_a;           // A のインプリメンテーションを宣言
A_ptr a = my_a._this();  // オブジェクト・リファレンスを取得
```

PortableServer 関数

POA に登録されたオブジェクトでは、オブジェクト識別子としてオクテット、特に PortableServer::POA::ObjectId 型のシーケンスを使用します。ただし、C++ のプログラマはオブジェクト識別子として文字列を使用することが多いため、C++ マッピングでは、文字列の ObjectId への変換、またはその逆の変換を行ういくつかの変換関数が用意されています。

```
// C++
namespace PortableServer
{
char* ObjectId_to_string(const ObjectId&);

ObjectId* string_to_ObjectId(const char*);
}
```

上記の関数は、パラメータ渡しおよびメモリ管理について、標準の C++ マッピングの規則に準拠しています。

ObjectId を文字列に変換したときに、文字列に正しくない文字 (NULL など) が生成された場合は、最初の 2 つの関数によって CORBA::BAD_PARAM 例外がスローされます。

モジュール

OMG IDL のモジュールは、C++ のクラスにマッピングされます。モジュールに格納されるオブジェクトは、この C++ クラスの中で定義します。インターフェイスと型もクラスにマッピングされるため、入れ子になった C++ クラスが生成されます。

次に、OMG IDL 定義の例を示します。

```
// OMG IDL

module INVENT
  {
  interface Order
        {
        . . .
        };
   };
```

上の定義は、次のように C++ にマッピングされます。

```
// C++

class INVENT
   {
   . . .
   class Order : public virtual CORBA::Object
        {
        .. .
        };  // Order クラス
    };  // INVENT クラス
```

入れ子になったモジュールが複数ある場合は、複数の入れ子になったクラスが生成されます。モジュール内部にあるものはすべてモジュール・クラスの中にあり、インターフェイス内部にあるものはすべてインターフェイス・クラスの中にあります。

OMG IDL では、モジュール、インターフェイス、および型の名前を同じにすることができます。ただし、C++ 言語用のファイルを生成する場合、名前を同じにすることはできません。この制限が必要になるのは、OMG IDL の名前が同じ名前で入れ子の C++ クラスに生成される際に、これを C++ コンパイラでサポートしていないためです。

注記　現在のモジュール名と同じ名前のインターフェイスまたは型で OMG IDL から C++ コードを生成すると、BEA Tuxedo OMG IDL コンパイラでは、その情報を伝えるメッセージが出力されます。このメッセージを無視して、インターフェイスまたは型とモジュール名とを区別するために一意の名前を付けなかった場合、コンパイラによってファイルの生成時にエラーが発生したことが通知されます。

インターフェイス

OMG IDL のインターフェイスは、C++ のクラスにマッピングされます。このクラスは、OMG IDL インターフェイス内にあるオペレーション、属性、定数、およびユーザ定義の型 (UDT) を格納します。

インターフェイス INTF の場合、生成されるインターフェイス・コードには、次の項目が格納されます。

- オブジェクト・リファレンスの型 (*INTF*_ptr)
- オブジェクト・リファレンスの変数の型 (*INTF*_var)
- _duplicate 静的メンバ関数
- _narrow 静的メンバ関数
- _nil 静的メンバ関数
- UDT
- 属性およびオペレーションのメンバ関数

次に、OMG IDL 定義の例を示します。

```
// OMG IDL

module INVENT
    {
     interface Order
       {
        void cancelOrder ();
        };
      };
```

上の定義は、次のように C++ にマッピングされます。

```
// C++
class INVENT
    {
     . . .
     class   Order;
      typedef Order *            Order_ptr;

     class Order : public virtual CORBA::Object
        {
        . . .
        static Order_ptr _duplicate(Order_ptr obj);
        static Order_ptr _narrow(CORBA::Object_ptr obj);
        static Order_ptr _nil();
        virtual void cancelOrder () = 0;
       . . .
       };
    };
```

オブジェクト・リファレンスの型、静的メンバ関数、UDT、オペレーション、および属性については、以降の節を参照してください。

生成される静的メンバ関数

ここでは、インターフェイス INTF に対して生成される、_duplicate、_narrow、および _nil の各静的メンバ関数について詳細に説明します。

static *INTF*_ptr _duplicate (*INTF*_ptr Obj)
: この静的メンバ関数は、既存の INTF オブジェクト・リファレンスを複製して、新しい INTF オブジェクト・リファレンスを返します。新しい INTF オブジェクト・リファレンスは、CORBA::release メンバ関数を呼び出して解放する必要があります。エラーが発生した場合、ニル INTF オブジェクトへのリファレンスが返されます。引数 Obj には、複製元のオブジェクト・リファレンスを指定します。

static *INTF*_ptr _narrow (CORBA::Object_ptr Obj)
: この静的メンバ関数は、既存の CORBA::Object_ptr オブジェクト・リファレンスに付与される、新しい INTF オブジェクト・リファレンスを返します。Object_ptr オブジェクト・リファレンスは、CORBA::ORB::string_to_object メンバ関数を呼び出して作成され

ているか、またはオペレーションからパラメータとして返されています。

*INTF*_ptr オブジェクト・リファレンスは、INTF オブジェクト、または INTF オブジェクトを継承するオブジェクトと対応していなければなりません。新しい INTF オブジェクト・リファレンスは、CORBA::release メンバ関数を呼び出して解放する必要があります。引数 Obj には、INTF オブジェクト・リファレンスにナロー変換されるオブジェクト・リファレンスを指定します。Obj パラメータは、このメンバ関数では変更されません。また、不要になったときは解放してください。Obj が INTF オブジェクト・リファレンスにナロー変換できない場合は、INTF ニル・オブジェクト・リファレンスが返されます。

static *INTF*_ptr _nil ( )
この静的メンバ関数は、*INTF* インターフェイスの新しいニル・オブジェクト・リファレンスを返します。新しいリファレンスは、CORBA::release メンバ関数を呼び出して解放する必要はありません。

オブジェクト・リファレンスの型

インターフェイス・クラス (INTF) は仮想クラスです。CORBA 標準では、次の処理は許可されていません。

- インターフェイス・クラスのインスタンスの作成または保持
- インターフェイス・クラスへのポインタまたはリファレンスの使用

代わりに、オブジェクト・リファレンスの型、*INTF*_ ptr クラス、または *INTF*_var クラスのいずれかを使用します。
オブジェクト・リファレンスを取得するには、_narrow 静的メンバ関数を使用します。これらのクラスでオペレーションを呼び出すには、矢印演算子 (->) を使用します。

*INTF*_var クラスは、*INTF*_var クラスがスコープをはずれるか、または再割り当てされるときに、オブジェクト・リファレンスを自動的に解放することでメモリ管理を簡単にします。変数の型は、多くの UDT に対して生成されます。「var クラスの使い方」では、変数の型について説明しています。

属性

OMG IDL の読み取り専用の属性は、属性値を返す C++ 関数にマッピングされます。読み書き属性は、2 つのオーバーロード C++ 関数にマッピングされます。2 つの関数で 1 つは属性値を返し、もう 1 つは属性値を設定します。オーバーロードのメンバ関数の名前が属性の名前になります。

属性はオペレーションの生成と同じ方法で生成されます。属性は、仮想クラスとスタブ・クラスの両方で定義します。次に、OMG IDL 定義の例を示します。

```
// OMG IDL

module INVENT
    {
    interface Order
        {
        . . .
        attribute itemStruct   itemInfo;
        };
    };
```

上の定義は、次のように C++ にマッピングされます。

```
// C++

class INVENT
    {
    . . .

    class Item : public virtual CORBA::Object
        {
        . . .
         virtual itemStruct * itemInfo ( ) = 0;

        virtual void itemInfo (
            const itemStruct & itemInfo) = 0;
        };
    };
```

```
class Stub_Item : public Item
    {
. . .
    itemStruct * itemInfo ();

    void itemInfo (
       const itemStruct & itemInfo);
    };
```

生成されたクライアント・アプリケーション・スタブには、スタブ・クラスに対して生成された次のコードが格納されます。

```
// ルーチン名：       INVENT::Stub_Item::itemInfo
//
// 関数の説明：
//
//   属性 INVENT::Stub_Item::itemInfo の
//   クライアント・アプリケーション・スタブ・ルーチン (Interface : Item)

INVENT::itemStruct * INVENT::Stub_Item::itemInfo ( )
{
. . .
}

//
// ルーチン名：       INVENT::Stub_Item::itemInfo
//
// 関数の説明：
//
//   属性 INVENT::Stub_Item::itemInfo の
//   クライアント・アプリケーション・スタブ・ルーチン (Interface : Item)

void INVENT::Stub_Item::itemInfo (
  const INVENT::itemStruct & itemInfo)
{
}
```

引数のマッピング

属性は、属性値を返すオペレーションと属性を設定するオペレーションの 2 つのオペレーションと等価です。たとえば、上記の `itemInfo` 属性は次と等価です。

```
void itemInfo (in itemStruct itemInfo);
itemStruct itemInfo ();
```

属性の引数のマッピングは、オペレーションの引数のマッピングと同じです。属性は、対応する C++ の型にマッピングされます。C++ の型については、表 0-1 および表 0-2 を参照してください。オペレーションでの引数のパラメータ渡しモードについては、表 0-7 および 表 0-8 を参照してください。

Any 型

OMG IDL の any は、CORBA::Any クラスにマッピングされます。CORBA::Any クラスは、型セーフな方法で C++ の型を処理します。

型付き値の処理

一致しない TypeCode や値で any が作成される可能性を低くするには、C++ 関数のオーバーロード機能を使用します。具体的には、OMG IDL 仕様の特定の型ごとに、その型の値を挿入および抽出するためのオーバーロード関数が用意されています。名前空間の純粋さが損なわれるのを完全に防ぐには、これらの関数にオーバーロード演算子を使用します。これらの関数については以下で詳細に説明しますが、特徴としては、any に挿入または any から抽出される値の C++ の型によって適切な TypeCode が示されるという点です。

後述する型セーフな any インターフェイスは C++ 関数のオーバーロードに基づいているため、OMG IDL 仕様から生成される C++ の型は別々のものである必要があります。ただし、この要件が満たされない特別な場合があります。

- OMG IDL の Boolean、octet、および char 型は、特定の C++ の型にマッピングされるとは限りません。したがって、関数のオーバーロードを行うために互いの型を識別する方法が別に必要となります。これらの型を互いに識別する方法については、「Boolean、Octet、Char、およびバウンディッド文字列の識別」を参照してください。
- 文字列がバウンディッドかアンバウンディッドかに関係なく、文字列はすべて char* にマッピングされます。そのため、バウンディッド文字列値で any を作成または設定する方法が別に必要となります。この方法については、「Boolean、Octet、Char、およびバウンディッド文字列の識別」を参照してください。

- C++ では、関数の引数リスト内の配列は、その最初の要素へのポインタに集束されます。したがって、関数のオーバーロードは、サイズが異なる配列の区別には使用できません。配列を扱う際に any を作成または設定する方法については、下記および「配列」を参照してください。

Any への挿入

型セーフな方法で any の値を設定できるようにするには、OMG IDL の型 T ごとに次のオーバーロード演算子関数が個別に用意されています。

```
// C++
void operator<<=(Any&, T);
```

この関数のシグニチャでは、次の型があれば十分です。通常、これら型は値によって渡されます。

- Short、UShort、Long、ULong、Float、Double
- 列挙
- アンバウンディッド文字列 (値によって渡された char*)
- オブジェクト・リファレンス (T_ptr)

型 T の値が大きすぎて完全に値を渡すことができない場合、次の 2 つの形式の挿入関数が用意されています。

```
// C++
void operator<<=(Any&, const T&);     // コピー形式
void operator<<=(Any&, T*);            // 非コピー形式
```

コピー形式は、呼び出し側に関して見れば、最初に示した形式とほぼ同じです。

上記の「左シフト代入」演算子は、次のように型付き値を any に挿入するために使用します。

```
// C++
Long value = 42;
Any a;
a <<= value;
```

この場合、型 Long にオーバーロードされた operator<<= のバージョンは、Any 変数の値と TypeCode プロパティの両方を設定します。

operator<<= を使用して any の値を設定することは、次のことを意味します。

- operator<<= のコピー・バージョンの場合、Any の値の存続期間は、operator<<= に渡された値の存続期間に依存しません。Any のインプリメンテーションでは、operator<<= に渡された値へのリファレンスまたはポインタとして、その値を格納しません。
- operator<<= の非コピー・バージョンの場合、挿入された T* は Any が処理します。挿入後は Any が T* の所有権を想定するため、呼び出し側は T* を使用して指されたデータにアクセスすることができません。また、Any は指されたデータを直ちにコピーして、元のデータを破棄します。
- operator<<= のコピーと非コピーの両バージョンでは、Any が保持していた前の値をすべて適切に割り当て解除します。たとえば、「型付けされていない値の処理」で説明した、Any(TypeCode_ptr,void*,TRUE) コンストラクタを呼び出して Any を作成した場合、Any は void* が指すメモリの割り当てを解除してから、新しい値をコピーします。

文字列型の挿入をコピーすると、次の関数が呼び出されます。

```
// C++
void operator<<=(Any&, const char*);
```

文字列型はすべて char* にマッピングされるので、この挿入関数は、挿入された値がアンバウンディッド文字列であることを想定します。バウンディッド文字列を正しく Any に挿入する方法については、「Boolean、Octet、Char、およびバウンディッド文字列の識別」を参照してください。バウンディッド文字列とアンバウンディッド文字列の挿入をコピーせずに実行するには、Any::from_string ヘルパ型を使用します。このヘルパ型については、「Boolean、Octet、Char、およびバウンディッド文字列の識別」を参照してください。

型セーフで配列を挿入するには、Array_forany 型を使用します。この型については、「配列」を参照してください。ORB では、Array_forany 型ごとに、オーバーロードされた operator<<= のバージョンが用意されています。たとえば、次のように入力します。

```
// IDL
typedef long LongArray[4][5];
```

```
// C++
typedef Long LongArray[4][5];
typedef Long LongArray_slice[5];
class LongArray_forany { ... };

void operator<<=(Any &, const LongArray_forany &);
```

Array_forany 型は、const へのリファレンスによって常に operator<<= に渡されます。Array_forany の nocopy フラグは、挿入された値をコピーするか (nocopy == FALSE)、処理するか (nocopy == TRUE) どうかを制御するために使用します。nocopy フラグはデフォルトで FALSE に指定されているため、デフォルトでは挿入がコピーされます。

T の配列と T* には型にあいまいさがあるため、移植性のあるコードの場合は、配列を Any に挿入するときに適切な Array_forany 型を明示的に使用することをお勧めします。たとえば、次のように入力します。

```
// IDL
struct S {... };
typedef S SA[5];

// C++
struct S { ... };
typedef S SA[5];
typedef S SA_slice;
class SA_forany { ... };

SA s;
// ... s を初期化 ...
Any a;
a <<= s;                              // 行 1
a <<= SA_forany(s);                   // 行 2
```

行 1 の場合、配列型 SA がその最初の要素へのポインタに集束されるため、コピー SA_forany 挿入演算子ではなく、非コピー operator<<=(Any&, S*) が呼び出されます。行 2 の場合、SA_forany 型が明示的に作成されるので、目的の挿入演算子が呼び出されます。

オブジェクト・リファレンスの operator<<= の非コピー・バージョンでは、次のように T_ptr 型のアドレスを取得します。

```
// IDL
interface T { ... };
```

```
// C++
void operator<<=(Any&, T_ptr);                              // コピー
void operator<<=(Any&, T_ptr*);                          // 非コピー
```

非コピーのオブジェクト・リファレンスの挿入では、T_ptr* が指すオブジェクト・リファレンスを処理します。したがって、Any が元のオブジェクト・リファレンスを複製して直ちに解放するため、挿入後に呼び出し側は、T_ptr が参照するオブジェクトにアクセスすることができません。呼び出し側は、T_ptr 自体の記憶域の所有権を保持します。

operator<<= のコピー・バージョンは、Any_var 型でもサポートされます。

Any からの抽出

型セーフな方法で any から値を取得できるようにするために、ORB では OMG IDL の型 T ごとに次の演算子が用意されています。

```
// C++
Boolean operator>>=(const Any&, T&);
```

この関数のシグニチャでは、プリミティブ型があれば十分です。通常、この型は値によって渡されます。型 T の値が大きすぎて完全に値を渡すことができない場合、ORB では次のように別のシグニチャが提供されます。

```
// C++
Boolean operator>>=(const Any&, T*&);
```

この関数の最初の形式は、次の型にのみ使用されます。

- Boolean、Char、Octet、Short、UShort、Long、ULong、Float、Double
- 列挙
- アンバウンディッド文字列 (リファレンスによって渡された char*、つまり char*&)
- オブジェクト・リファレンス (T_ptr)

その他の型については、すべて関数の 2 番目の形式が使用されます。

上記の「右シフト代入」演算子は、次のように型付き値を any から抽出するために使用します。

```
// C++
Long value;
Any a;
a <<= Long(42);
if (a >>= value) {
       // ... 値を使用 ...
}
```

この場合、型 Long の operator>>= のバージョンは、Any が型 Long の値を格納しているかどうかを判別します。格納している場合、呼び出し側によって提供されたリファレンス変数にその値をコピーし、TRUE を返します。Any が型 Long の値を格納していない場合、呼び出し側のリファレンス変数の値は変更されず、operator>>= は FALSE を返します。

型が非プリミティブの場合、ポインタが抽出を行います。次に、OMG IDL 構造体の例を示します。

```
// IDL
struct MyStruct {
    long lmem;
    short smem;
};
```

上記の構造体は、次のように Any から抽出できます。

```
// C++
Any a;
// ...a に型 MyStruct の値を指定 ...
MyStruct *struct_ptr;
if (a >>= struct_ptr) {
       // ... 値を使用 ...
}
```

抽出に成功した場合、呼び出し側のポインタは Any が管理する記憶域を指し、operator>>= は TRUE を返します。呼び出し側は、この記憶域の delete または解放を試行する必要はありません。また、代入、挿入、または replace 関数によって Any 変数が置き換えられた後、または Any 変数が破棄された後に、呼び出し側は記憶域を使用する必要もありません。ただし、これらの抽出演算子で T_var 型を使用しないよう注意する必要があります。これは、Any によって所有される記憶域を抽出演算子が削除しようとするためです。

抽出に失敗した場合、呼び出し側のポインタの値は NULL ポインタと同じに設定され、operator>>= は FALSE を返します。

配列型を正しく抽出するには、Array_forany 型を使用します。この型については、「配列」を参照してください。

次に、OMG IDL の例を示します。

```
// IDL
typedef long A[20];
typedef A B[30][40][50];

// C++
typedef Long A[20];
typedef Long A_slice;
class A_forany { ... };
typedef A B[30][40][50];
typedef A B_slice[40][50];
class B_forany { ... };

Boolean operator>>=(const Any&, A_forany&);
// 型 A 用
Boolean operator>>=(const Any&, B_forany&);                    //
型 B 用
```

Array_forany 型は、リファレンスによって常に operator>>= に渡されます。

文字列および配列の場合、アプリケーションは Any の TypeCode をチェックします。これは、抽出値を使用する際に、アプリケーションが確実に配列オブジェクトまたは文字列オブジェクトのバウンドを超えないようにするためです。

operator>>= は、Any_var 型でもサポートされます。

Boolean、Octet、Char、およびバウンディッド文字列の識別

OMG IDL の型の Boolean、octet、および char は、特定の C++ 型にマッピングする必要はありません。そのため、型セーフな Any インターフェイスで使用できるようにするために、これらの型を互いに識別する方法が別に必要となります。同様に、バウンディッド文字列とアンバウンディッド文字列は共に char* にマッピングされるため、両者を識別する方法が別に必要となります。ここでは、識別を行うために、Any クラス・インターフェイス内で入れ子になった新しいヘルパ型をいくつか導入します。ヘルパ型を使用した識別の例を次に示します。

```
// C++
class Any
```

```
{
  public:
    // 特別なヘルパ型が boolean、octet、char、
    // およびバウンディッド文字列の挿入に必要
    struct from_boolean {
        from_boolean(Boolean b) : val(b) {}
        Boolean val;
    };
    struct from_octet {
        from_octet(Octet o) : val(o) {}
        Octet val;
    };
    struct from_char {
       from_char(Char c) : val(c) {}
       Char val;
    };
    struct from_string {
       from_string(char* s, ULong b,
                 Boolean nocopy = FALSE) :
            val(s), bound(b) {}
       char *val;
       ULong bound;
    };

    void operator<<=(from_boolean);
    void operator<<=(from_char);
    void operator<<=(from_octet);
    void operator<<=(from_string);
    // 特別なヘルパ型が boolean、octet、char、
    // およびバウンディッド文字列の抽出に必要
    struct to_boolean {
        to_boolean(Boolean &b) : ref(b) {}
        Boolean &ref;
    };
    struct to_char {
        to_char(Char &c) : ref(c) {}
        Char &ref;
    };
    struct to_octet {
        to_octet(Octet &o) : ref(o) {}
        Octet &ref;
    };
    struct to_string {
        to_string(char *&s, ULong b) : val(s), bound(b) {}
        char *&val;
        ULong bound;
    };
```

```
    Boolean operator>>=(to_boolean) const;
    Boolean operator>>=(to_char) const;
    Boolean operator>>=(to_octet) const;
    Boolean operator>>=(to_string) const;

    // ほかのパブリック Any の詳細は省略

private:
    // 下記の関数はプライベートでインプリメントされない。
    // 下記の関数を隠ぺいすると、unsigned char の
    // コンパイル・エラーが発生する
    void operator<<=(unsigned char);
    Boolean operator>>=(unsigned char &) const;
};
```

ORB では、これらの特別なヘルパ型用に、オーバーロードの operator<<= および operator>>= 関数を提供します。これらのヘルパ型は、次のように使用します。

```
// C++
Boolean b = TRUE;
Any any;
any <<= Any::from_boolean(b);
// ...
if (any >>= Any::to_boolean(b)) {
     // ... Boolean を格納した any ...
}

char* p = "bounded";
any <<= Any::from_string(p, 8);
// ...
if (any >>= Any::to_string(p, 8)) {
     // ... string<8> を格納した any ...
}
```

バウンド値が 0 (ゼロ) の場合、アンバウンディッド文字列を示します。

Any へのバウンディッド文字列またはアンバウンディッド文字列の挿入をコピーせずに行う場合、from_string コンストラクタで nocopy フラグを TRUE に設定する必要があります。

```
// C++
char* p = string_alloc(8);
// ... 文字列 p を初期化 ...
any <<= Any::from_string(p, 8, 1);      // any は p を処理
```

boolean、char、および octet がすべて C++ 型の unsigned char にマッピングされる場合、boolean、char、または octet 型の any の直接挿入または抽出を試行すると、unsigned char 用のプライベートでインプリメントされていない operator<<= と operator>>= によって、コンパイル・エラーが発生します。

```
// C++
Octet oct = 040;
Any any;
any <<= oct;                          // この行はコンパイルされない。
any <<= Any::from_octet(oct);         // この行はコンパイルされる。
```

Object への型の拡大

Any からオブジェクト・リファレンスを基本 Object 型として抽出する方が望ましい場合があります。これを実行するには、boolean、char、および octet の抽出に必要なヘルパ型と同様のヘルパ型を使用します。

```
// C++
class Any
{
  public:
    ...
    struct to_object {
        to_object(Object_ptr &obj) : ref(obj) {}
        Object_ptr &ref;
    ;
    Boolean operator>>=(to_object) const;
    ...
};
```

to_object ヘルパ型は、Any からオブジェクト・リファレンスを基本 Object 型として抽出するために使用します。Any にその TypeCode として示されたオブジェクト・リファレンス型の値がある場合、抽出関数 operator>>=(to_object) は、その格納されているオブジェクト・リファレンスを明示的に Object 型に拡大し、TRUE を返します。それ以外の場合は、FALSE を返します。抽出されたオブジェクト・リファレンスの型の拡大を実行するのは、オブジェクト・リファレンス抽出関数のみです。通常のオブジェクト・リファレンスの抽出と同じく、to_object 抽出演算子は、オブジェクト・リファレンスを複製しません。

型付けされていない値の処理

状況によっては、Any への型セーフなインターフェイスでは不十分な場合があります。たとえば、データ型がバイナリ形式でファイルから読み取られ、型 Any の値を作成するために使用される状況などです。このような場合、Any クラスは、コンストラクタに明示的な TypeCode とジェネリック・ポインタを提供します。

```
// C++
Any(TypeCode_ptr tc, void *value, Boolean release = FALSE);
```

コンストラクタは、指定の TypeCode 擬似オブジェクト・リファレンスを複製します。release パラメータが TRUE の場合、Any オブジェクトは、value パラメータが指す記憶域の所有権を想定します。value パラメータが release=TRUE の指定で Any に対して処理された後は、呼び出し側はこのパラメータの存続期間を想定できません。これは、Any が value パラメータをコピーして、直ちに元のポインタを解放するためです。release パラメータがデフォルトの FALSE の場合、Any オブジェクトは、value によって指されるメモリを呼び出し側が管理すると想定します。value パラメータは NULL ポインタでもかまいません。

Any クラスでは、3 つの危険なオペレーションも定義します。

```
// C++
void replace(
    TypeCode_ptr,
    void *value,
    Boolean release = FALSE
);
TypeCode_ptr type() const;
const void *value() const;
```

replace 関数は、型セーフな挿入インターフェイスで使用不可能な型で使用します。したがって、この関数は上述のコンストラクタに似ています。既存の TypeCode は解放され、必要に応じて値の記憶域の割り当てが解除されます。TypeCode 関数のパラメータは複製されます。release パラメータが TRUE の場合、Any オブジェクトは、value パラメータが指す記憶域の所有権を想定します。value パラメータが release=TRUE の指定で Any::replace 関数に対して処理された後は、Any はこのパラメータの存続期間を想定できません。これは、Any が value パラメータをコピーして、直ちに元のポインタを解放するためです。release パラメータがデフォルトの FALSE の場合、

Any オブジェクトは、値が入っているメモリを呼び出し側が管理すると想定します。replace 関数の value パラメータは NULL ポインタでもかまいません。

上記のコンストラクタも replace 関数も型セーフではありません。特に、TypeCode と void* 引数の実際の型との一貫性については、実行時にコンパイラは保証しません。一致しない TypeCode と値で Any が作成される場合の、ORB インプリメンテーションの動作は定義されません。

type 関数は、Any に関連付けられた TypeCode に TypeCode_ptr 擬似オブジェクト・リファレンスを返します。すべてのオブジェクト・リファレンスの戻り値と同様に、リファレンスが不要になったときに呼び出し側は、それを解放するか、または自動管理を行うために TypeCode_var 変数を割り当てる必要があります。

value 関数は、Any に格納されているデータへのポインタを返します。Any に関連付けられている値がない場合、value 関数は NULL ポインタを返します。

Any のコンストラクタ、デストラクタ、代入演算子

デフォルト・コンストラクタは、値を作成せずに、型 tk_null の TypeCode で Any を作成します。コピー・コンストラクタは、Any パラメータの TypeCode_ptr で _duplicate を呼び出し、パラメータの値をディープ・コピーします。代入演算子は、それ自身の TypeCode_ptr を解放し、必要に応じて現在の値の記憶域を割り当て解除します。その後、Any パラメータの TypeCode_ptr を複製し、パラメータの値をディープ・コピーします。デストラクタは、TypeCode_ptr で release を呼び出し、必要に応じて値の記憶域を割り当て解除します。

その他のコンストラクタについては、「型付けされていない値の処理」を参照してください。

Any クラス

Any クラスの定義の詳細については、「Any クラスのメンバ関数」を参照してください。

値型

この節は、Object Management Group (OMG) の「Common Object Request Broker: Architecture and Specification, Revision 2.4.2」(2001 年 2 月) の第 3 章、第 5 章、および第 6 章、および「CORBA C++ Language Mapping Specification」(1999 年 6 月) に記載されている情報に基づいています。また、この情報は、OMG の許可を得て転載しています。

概要

特に、オブジェクトがサポートするオブジェクト・インターフェイス型は、IDL インターフェイスで定義するので、任意のインプリメンテーションが可能になります。分散オブジェクト・システムでは、インプリメンテーション上の制約がほとんどないので、大きな値が入ります。ただし、リファレンスよりも値でオブジェクトを渡すことが可能な方が望ましい場合が多くあります。これは、オブジェクトの主要な「目的」がデータをカプセル化する場合や、アプリケーションでオブジェクトの「コピー」を明示的に作成するのが好ましい場合に、特に便利です。

値でオブジェクトを渡すセマンティクスは、標準のプログラミング言語のセマンティクスとほぼ同じです。値で渡されるパラメータの受信側は、オブジェクトの「状態」の記述を受信します。次に、受信側は新しいインタンスをその状態でインスタンス化します。その際、送信側のインスタンスの ID を個別に付けます。パラメータ渡しのオペレーションが完了すると、2 つのインスタンス間には何の関係もないものと見なされます。

受信側はインスタンスをインスタンス化する必要があるため、オブジェクトの状態とインプリメンテーションに関する情報を認識している必要があります。そこで、値型では、CORBA 構造体と CORBA インターフェイスをブリッジする、次のセマンティクスを備えています。

- 再帰やサイクルを伴った任意のグラフなど、複雑な状態の記述をサポートします。
- インスタンスはリモート呼び出しへのパラメータとして渡されるときに常にコピーされるので、使用されるコンテキストに対してインスタンスは常にローカルです。

- インプリメンテーションに対してパブリック・データ・メンバとプライベート・データ・メンバの両方をサポートします。
- オブジェクト・インプリメンテーションの状態の指定に使用できます。つまり、インターフェイスをサポートできます。
- サポートする値型の継承は 1 つのみです。また、インターフェイスをサポートできます。
- 抽象にもなることができます。

アーキテクチャ

値型の基本概念は比較的単純です。ある意味で値型は、「通常の」IDL インターフェイス型と構造体との中間的なものと言えます。アプリケーションのプログラマは、インターフェイス型に関係なく追加のプロパティ (状態) とインプリメンテーションの詳細が指定されることを、値型を使用して通知します。この情報の指定は、インターフェイス型に関係なくインプリメンテーション上の制約として追加されます。これは、ここで指定したセマンティクスと言語マッピングの両方に反映されます。

利点

値型 (値で渡すことが可能なオブジェクト) のサポート以前は、CORBA オブジェクトには、すべてオブジェクト・リファレンスがありました。そのため、複数のクライアントが特定のオブジェクトを呼び出すとき、クライアントは同じリファレンスを使用していました。オブジェクトのインスタンスはサーバ ORB 上に残り、その状態はクライアント ORB ではなく、サーバ ORB が管理していました。

値型は、CORBA アーキテクチャへの重要な追加機能であると言えます。リファレンスで渡されるオブジェクトと同じく、値型には状態とメソッドがありますが、オブジェクト・リファレンスはありません。また、プログラミング言語のオブジェクトと同様に常にローカルで呼び出されます。受信側から要求があると、値型は送信コンテキストでその状態をパッケージ化し、「ネットワーク経由で」状態を受信側に送信します。受信側では、インスタンスが作成され、転送された状態が設定されます。この段階以降、送信側はクライアント側のインスタンスを制御できなくなります。つまり、これ以降は、受信側がインスタンスをローカルで呼び出すことができるようになります。このモデルにより、ネットワーク経由の通信で生じる遅延を短縮するこ

とができます。この遅延は、大規模なネットワークで顕著に生じます。値型を追加することにより、CORBA インプリメンテーションの規模の調整が容易になり、大量のデータ処理要件を満たすことができます。

このように、値型の本質的な特徴は、そのインプリメンテーションが常にローカルである点です。つまり、具象的なプログラミング言語で値型を明示的に使用すると、常にローカルでインプリメンテーションが行われ、リモート呼び出しが不要になることが保証されます。値型の値自体が ID であるため、値型には ID がなく、ORB に「登録」されません。

値型の例

次に、IDL の値型の例を示します。これは、Object Management Group (OMG) の「CORBA C++ Language Mapping Specification」(1999 年 6 月) から引用したものです。

```
// IDL
valuetype Example {
      short op1();
      long op2(in Example x);
      private short val1;
      public long val2;

      private string val3;
      private float val4;
      private Example val5;
};
```

上記の値型の C++ へのマッピングは次のとおりです。

```
// C++
class Example : public virtual ValueBase {
   public:
      virtual Short op1() = 0;
      virtual Long op2(Example*) = 0;

      virtual Long val2() const = 0;
      virtual void val2(Long) = 0;

      static Example* _downcast(ValueBase*);

   protected:
      Example();
      virtual ~Example();
```

```
        virtual Short val1() const = 0;
        virtual void val1(Short) = 0;

        virtual const char* val3() const = 0;
        virtual void val3(char*) = 0;
        virtual void val3(const char*) = 0;
        virtual void val3(const String_var&) = 0;

        virtual Float val4() const = 0;
        virtual void val4(Float) = 0;

        virtual Example* val5() const = 0;
        virtual void val5(Example*) = 0;

    private:
        // プライベートおよび未インプリメント
        void operator=(const Example&);
};

class OBV_Example : public virtual Example {
    public:
        virtual Long val2() const;
        virtual void val2(Long);

    protected:
        OBV_Example();
        OBV_Example(Short init_val1, Long init_val2,
                    const char* init_val3, Float init_val4,
                    Example* init_val5);
        virtual ~OBV_Example();

        virtual Short val1() const;
        virtual void val1(Short);

        virtual const char* val3() const;
        virtual void val3(char*);
        virtual void val3(const char*);
        virtual void val3(const String_var&);

        virtual Float val4() const;
        virtual void val4(Float);

        virtual Example* val5() const;
        virtual void val5(Example*);

        // ...
};
```

固定長ユーザ定義型と可変長ユーザ定義型

ユーザ定義型のメモリ管理規則およびメンバ関数のシグニチャは、その型が固定長か可変長かどうかによって異なります。次のいずれか 1 つに該当する場合、ユーザ定義型は可変長です。

- バウンディッド文字列またはアンバウンディッド文字列
- バウンディッド・シーケンスまたはアンバウンディッド・シーケンス
- 可変長メンバを含む構造体または共用体
- 可変長の要素型を持つ配列
- 可変長型への typedef

上記の一覧に型が該当しない場合、その型は固定長です。

var クラスの使い方

自動変数 var は、メモリ管理を簡単に行うために用意されています。var は var クラスを通じて提供されます。このクラスは、型に必要なメモリの所有権を想定し、var オブジェクトのインスタンスが破棄されたり、新しい値が var オブジェクトに割り当てられたときにメモリを解放します。

BEA Tuxedo では、次の型の var クラスが用意されています。

- String (`CORBA::String_var`)
- Object references (`CORBA::Object_var`)
- ユーザ定義の OMG IDL の型 (`struct`、`union`、`sequence`、`array`、および `interface`)

var クラスのメンバ関数は共通ですが、OMG IDL の型によって演算子を追加でサポートする場合があります。OMG IDL の型 TYPE の場合、TYPE_var クラスは、コンストラクタ、デストラクタ、代入演算子、および基となる TYPE 型にアクセスするための演算子を格納しています。次に、var クラスの例を示します。

```
class TYPE_var
    {
    public:
        // コンストラクタ
        TYPE_var();
        TYPE_var(TYPE *);
        TYPE_var(const TYPE_var &);
        // デストラクタ
        ~TYPE_var();

        // 代入演算子
        TYPE_var &operator=(TYPE *);
        TYPE_var &operator=(const TYPE_var &);

        // アクセサ演算子
        TYPE *operator->();
        TYPE *operator->() const;

        TYPE_var_ptr in() const;
        TYPE_var_ptr& inout();
        TYPE_var_ptr& out();

        TYPE_var_ptr _retn();
        operator const TYPE_ptr&() const;
        operator TYPE_ptr&();
        operator TYPE_ptr;
        };
```

次に、各メンバ関数の詳細について説明します。

`TYPE_var()`

TYPE_var クラスのデフォルトのコンストラクタです。このコンストラクタは 0 (ゼロ) に初期化します。この値は、var クラスが TYPE * を所有することを示します。有効な TYPE * を割り当てていない限り、TYPE_var クラスで operator-> を呼び出すことはできません。

`TYPE_var(TYPE * Value);`

このコンストラクタは、指定の TYPE * パラメータの所有権を想定します。TYPE_var が破棄されると、TYPE は解放されます。Value 引数

は、この var クラスが所有する TYPE へのポインタです。このポインタは 0 (ゼロ) に指定しないでください。

```
TYPE_var(const TYPE_var & From);
```

このコピー・コンストラクタは、新しい TYPE を割り当てて、From パラメータが所有する TYPE に格納されているデータをディープ・コピーします。TYPE_var が破棄されると、TYPE のコピーは解放または削除されます。From パラメータには、コピー元の TYPE を指す var クラスを指定します。

```
~TYPE_var();
```

このデストラクタは、var クラスが所有する TYPE を適切なメカニズムで解放します。これは、文字列の場合は CORBA::string_free ルーチン、オブジェクト・リファレンスの場合は CORBA::release ルーチンです。その他の型の場合は delete、または割り当て済みのメモリを解放するために生成された静的ルーチンです。

```
TYPE_var &operator=(TYPE * NewValue);
```

この代入演算子は、NewValue パラメータが指す TYPE の所有権を想定します。現在、TYPE_var が TYPE を所有している場合、それを解放してから、NewValue パラメータの所有権を想定します。NewValue 引数は、この var クラスが所有する TYPE へのポインタです。このポインタは 0 (ゼロ) に指定しないでください。

```
TYPE_var &operator=(const TYPE_var &From);
```

この代入演算子は、新しい TYPE を割り当てて、From TYPE_var パラメータが所有する TYPE に格納されているデータをディープ・コピーします。現在、TYPE_var が TYPE を所有している場合は解放されます。TYPE_var が破棄されると、TYPE のコピーは解放されます。From パラメータには、コピー元のデータを指す var クラスを指定します。

```
TYPE *operator->();
TYPE *operator->() const;
```

これらの演算子は、var クラスが所有している TYPE へのポインタを返します。var クラスは、引き続き TYPE を所有し、TYPE を解放します。var が有効な TYPE を所有していない場合は、operator-> を使用できません。TYPE_var が破棄された後に、この戻り値の解放またはアクセスを試行しないでください。

```
TYPE_var_ptr in() const;
TYPE_var_ptr& inout();
```

```
TYPE_var_ptr& out();
TYPE_var_ptr _retn();
```

暗黙的な変換を実行すると、一部の C++ コンパイラやコードの可読性に問題が生じることがあります。そのため、TYPE_var 型では、パラメータ渡しのために明示的な変換を実行できるようにするメンバ関数もサポートしています。TYPE_var および in パラメータを渡すには、in() メンバ関数を、inout パラメータを渡す場合は inout() メンバ関数を、out パラメータを渡す場合は out() メンバ関数をそれぞれ呼び出します。TYPE_var から戻り値を取得するには、_return() 関数を呼び出します。各 TYPE_var 型について、これらの関数の戻り値の型はそれぞれ、表 0-7 で示す in、inout、out の型に一致し、基となる型 TYPE のモードを返します。

ユーザ定義のデータ型によって、サポートされる演算子は一部異なります。表 0-3 では、生成された C++ コードにおける各 OMG IDL のデータ型でサポートされるさまざまな演算子について説明しています。表 0-3 で示すように、代入演算子はすべてのデータ型でサポートされるので、比較には含まれていません。

表 0-3 ユーザ定義のデータ型の var クラスでサポートされる演算子の比較

OMG IDL のデータ型	operator ->	operator[ ]
struct	あり	なし
union	あり	なし
sequence	あり	あり (const を除く)
array	なし	あり

表 0-4 では、シグニチャを示します。

表 0-4 _var クラスの演算子のシグニチャ

OMG IDL のデータ型	演算子メンバ関数
struct	TYPE * operator-> () TYPE * operator-> () const

表 0-4 _var クラスの演算子のシグニチャ

OMG IDL のデータ型	演算子メンバ関数
union	TYPE * operator-> () TYPE * operator-> () const
sequence	TYPE * operator-> () TYPE * operator-> () const TYPE & operator[](CORBA::Long index)
array	TYPE_slice & operator[](CORBA::Long index) TYPE_slice & operator[](CORBA::Long index) const

シーケンス var

シーケンス var では、次の operator[] メンバ関数を追加でサポートします。

```
TYPE &operator[](CORBA::ULong Index);
```

この演算子は、var クラスが所有するシーケンスの operator[] を呼び出します。operator[] は、指定のインデックスでシーケンスの適切な要素へのリファレンスを返します。Index 引数には、戻り値となる要素のインデックスを指定します。このインデックスは、現在のシーケンスの長さを超えることはできません。

配列 var

配列 var では、配列要素にアクセスするために operator-> ではなく、次の operator[] メンバ関数を追加でサポートします。

```
TYPE_slice& operator[](CORBA::ULong Index);
const TYPE_slice & operator[](CORBA::ULong Index) const;
```

上記の演算子は、指定のインデックスで配列スライスへのリファレンスを返します。配列のスライスとは、元の配列の最初のサイズを除いたすべてのサイズを持つ配列のことです。配列で生成されたクラスのメンバ関数は、スライスへのポインタを使用して配列へのポ

インタを返します。Index 引数には、戻り値となるスライスのインデックスを指定します。このインデックスは、配列のサイズを超えることはできません。

文字列 var

この節と「シーケンス var」で説明するメンバ関数の文字列 var には、char * の TYPE があります。文字列 var では、次のメンバ関数を追加でサポートします。

```
String_var(char * str)
```
このコンストラクタは、文字列から String_var を作成します。str 引数には、想定される文字列を指定します。str ポインタを使用して、データにアクセスしないでください。

```
String_var(const char * str)
String_var(const String_var & var)
```
このコンストラクタは、const 文字列から String_var を作成します。str 引数には、コピー元となる const 文字列を指定します。var 引数には、コピー元となる文字列へのリファレンスを指定します。

```
String_var & operator=(char * str)
```
この代入演算子は、CORBA::string_free を使用して格納されている文字列を解放してから、入力文字列の所有権を想定します。str 引数には、この String_var オブジェクトが所有権を想定する文字列を指定します。

```
String_var & operator=(const char * str)
String_var & operator=(const String_var & var)
```
この代入演算子は、CORBA::string_free を使用して格納されている文字列を解放してから、入力文字列をコピーします。Data 引数には、この String_var オブジェクトが所有権を想定する文字列を指定します。

```
char operator[] (Ulong Index)
char operator[] (Ulong Index) const
```
上記の配列演算子は、文字列内の文字へのアクセスを提供する添字付き演算子です。Index 引数には、配列内の特定の文字にアクセスするときに使用する配列のインデックスを指定します。インデックスの基数はゼロです。Char operator[] (Ulong Index) 関数の戻り

値は、lvalue として使用できます。
`Char operator[] (Ulong Index) const` 関数の戻り値は、lvalue として使用できません。

out クラス

構造化された型 (構造体、共用体、シーケンス)、配列、およびインターフェイスには、対応する _out クラスが生成されます。out クラスは、可変長型および固定長型へのポインタのメモリ管理を容易にするために提供されます。out クラスと共通のメンバ関数の詳細については、「out クラスの使い方」を参照してください。

ユーザ定義のデータ型によって、サポートされる演算子は一部異なります。表 0-5 では、生成された C++ コードにおける各 OMG IDL のデータ型でサポートされるさまざまな演算子について説明しています。表 0-3 で示すように、代入演算子はすべてのデータ型でサポートされるので、比較には含まれていません。

表 0-5 ユーザ定義のデータ型の out クラスでサポートされる演算子の比較

OMG IDL のデータ型	operator ->	operator[ ]
`struct`	あり	なし
`union`	あり	なし
`sequence`	あり	あり (const を除く)
`array`	なし	あり

表 0-6 では、シグニチャを示します。

表 0-6 _out クラスの演算子のシグニチャ

OMG IDL のデータ型	演算子メンバ関数
`struct`	`TYPE * operator-> ()` `TYPE * operator-> () const`

表 0-6 _out クラスの演算子のシグニチャ

OMG IDL のデータ型	演算子メンバ関数
union	TYPE * operator-> () TYPE * operator-> () const
sequence	TYPE * operator-> () TYPE * operator-> () const TYPE & operator[](CORBA::Long index)
array	TYPE_slice & operator[](CORBA::Long index) TYPE_slice & operator[](CORBA::Long index) const

out クラスの使い方

TYPE_var を out パラメータとして渡すと、参照先だった前の値はすべて暗黙的に削除する必要があります。この要件を満たす ORB フックを付与するために、各 T_var 型には対応する TYPE_out 型があります。この型は、out パラメータ型としてのみ使用します。

注記　プログラマは、_out クラスを直接インスタンス化できません。そのため、_out クラスは関数のシグニチャでのみ指定してください。

可変長型の TYPE_out 型の一般的な形式は、次のとおりです。

```
// C++
class TYPE_out
{
  public:
    TYPE_out(TYPE*& p) : ptr_(p) { ptr_ = 0; }
    TYPE_out(TYPE_var& p) : ptr_(p.ptr_) { delete ptr_; ptr_ = 0;}
    TYPE_out(TYPE_out& p) : ptr_(p.ptr_) {}
    TYPE_out& operator=(TYPE_out& p) { ptr_ = p.ptr_;
                                       return *this;
    }
    Type_out& operator=(Type* p) { ptr_ = p; return *this; }

    operator Type*&() { return ptr_; }
```

```
    Type*& ptr() { return ptr_; }

    Type* operator->() { return ptr_; }

  private:
     Type*& ptr_;

     // 代入演算子 TYPE_var not allowed
     void operator=(const TYPE_var&):
};
```

最初のコンストラクタは、リファレンス・データ・メンバを T*& 引数にバインドし、ポインタをゼロ (0) ポインタ値に設定します。2 番目のコンストラクタは、リファレンス・データ・メンバを TYPE_var 引数が保持するポインタにバインドし、ポインタで delete を呼び出します。その際、型が String_out の場合は string_free() を、配列型 TYPE が TYPE_var の場合は TYPE_free() を呼び出します。3 番目のコンストラクタはコピー・コンストラクタで、コンストラクタ引数のデータ・メンバによって参照される同じポインタにリファレンス・データ・メンバをバインドします。

ほかの TYPE_out から代入すると、TYPE_out 引数によって参照される TYPE* がデータ・メンバにコピーされます。TYPE* のオーバーロードの代入演算子は、ポインタ引数をデータ・メンバに代入するだけです。代入を行っても、以前に保持されていたポインタはまったく削除されません。この意味で、TYPE_out 型の動作は、TYPE* とまったく同じです。TYPE*& 変換演算子は、データ・メンバを返します。ptr() メンバ関数もデータ・メンバを返しますが、暗黙的な変換の実行を避ける場合に使用できます。オーバーロードの矢印演算子 (operator->()) を使用すると、TYPE* データ・メンバが指すデータ構造体のメンバにアクセスできます。アプリケーションに準拠する場合、有効な非 NULL TYPE* で TYPE_out が初期化済みでない限り、オーバーロードの operator->() を呼び出すことはできません。

対応する TYPE_var 型のインスタンスから、TYPE_out に代入することはできません。これは、アプリケーション開発者がコピーを行うかどうか、または TYPE_var が TYPE_out に代入できるよう、それが管理するポインタの所有権を返すかどうかを判定する方法がないためです。TYPE_var を TYPE_out にコピーするには、アプリケーションで次のように new を使用する必要があります。

```
// C++
TYPE_var t = ...;
my_out = new TYPE(t.in());          // コピーのヒープ割り当て
```

通常、t で呼び出される in() 関数は、const TYPE& を返して、新しく割り当てられた T インスタンスのコピー・コンストラクタを呼び出すことができるようにします。

また、TYPE_var を作成すると、T_out パラメータで返されるよう、それが管理するポインタの所有権を返します。アプリケーションでは、次のように TYPE_var::_retn() 関数を使用する必要があります。

```
// C++
TYPE_var t = ...;
my_out = t._retn();        // t は所有権を返す。コピーは行われない
```

TYPE_out 型は、アプリケーションで作成または破棄される汎用目的のデータ型としては機能しません。そのため、この型は必要なメモリ管理を適切に行う目的で、オペレーションのシグニチャの内部でのみ使用する型です。

オブジェクト・リファレンスの out パラメータ

_var を out パラメータとして渡すと、参照先の前の値はすべて暗黙的に解放する必要があります。この要件を満たす C++ マッピング・インプリメンテーションのフックを付与するために、オブジェクト・リファレンス型ごとに _out 型が生成されます。この型は、out パラメータ型としてのみ使用します。たとえば、インターフェイス TYPE の場合、オブジェクト・リファレンス型 TYPE_ptr、ヘルパ型 TYPE_var、および out パラメータ型 TYPE_out が生成されます。オブジェクト・リファレンス _out 型の一般的な形式は次のとおりです。

```
// C++
class TYPE_out
{
  public:
    TYPE_out(TYPE_ptr& p) : ptr_(p) { ptr_ = TYPE::_nil(); }
    TYPE_out(TYPE_var& p) : ptr_(p.ptr_) {
        release(ptr_); ptr_ = TYPE::_nil();
    }
    TYPE_out(TYPE_out& a) : ptr_(a.ptr_) {}
    TYPE_out& operator=(TYPE_out& a) {
        ptr_ = a.ptr_; return *this;
    }
    TYPE_out& operator=(const TYPE_var& a) {
    ptr_ = TYPE::_duplicate(TYPE_ptr(a)); return *this;
```

```
    }
    TYPE_out& operator=(TYPE_ptr p) { ptr_ = p; return *this; }
    operator TYPE_ptr&() { return ptr_; }
    TYPE_ptr& ptr() { return ptr_; }
    TYPE_ptr operator->() { return ptr_; }

  private:
    TYPE_ptr& ptr_;
};
```

シーケンス out

シーケンス out では、次の operator[] メンバ関数を追加でサポートします。

```
TYPE &operator[](CORBA::ULong Index);
```

この演算子は、out クラスが所有するシーケンスの operator[] を呼び出します。operator[] は、指定のインデックスでシーケンスの適切な要素へのリファレンスを返します。Index 引数には、戻り値となる要素のインデックスを指定します。このインデックスは、現在のシーケンスの長さを超えることはできません。

配列 out

配列 out では、配列要素にアクセスするために operator-> ではなく、次の operator[] メンバ関数を追加でサポートします。

```
TYPE_slice& operator[](CORBA::ULong Index);
const TYPE_slice & operator[](CORBA::ULong Index) const;
```

上記の演算子は、指定のインデックスで配列スライスへのリファレンスを返します。配列のスライスとは、元の配列の最初のサイズを除いたすべてのサイズを持つ配列のことです。配列で生成されたクラスのメンバ関数は、スライスへのポインタを使用して配列へのポインタを返します。Index 引数には、戻り値となるスライスのインデックスを指定します。このインデックスは、配列のサイズを超えることはできません。

文字列 out

String_var を out パラメータとして渡すと、参照先の前の値はすべて暗黙的に解放する必要があります。この要件を満たす C++ マッピング・インプリメンテーションのフックを付与するために、文字列型にも CORBA 名前空間で String_out 型が生成されます。この型は、文字列 out パラメータ型としてのみ使用します。String_out 型の一般的な形式は次のとおりです。

```
// C++
class String_out
{
  public:
     String_out(char*& p) : ptr_(p) { ptr_ = 0; }
     String_out(String_var& p) : ptr_(p.ptr_) {
         string_free(ptr_); ptr_ = 0;
     }
     String_out(String_out& s) : ptr_(s.ptr_) {}
            String_out& operator=(String_out& s) {
          ptr_ = s.ptr_; return *this;
     }
     String_out& operator=(char* p) {
          ptr_ = p; return *this;
     }
     String_out& operator=(const char* p) {
          ptr_ = string_dup(p); return *this;
     }
     operator char*&() { return ptr_; }
     char*& ptr() { return ptr_; }

  private:
     char*& ptr_;

     // String_var からの代入は不可
     void operator=(const String_var&);
};
```

最初のコンストラクタは、リファレンス・データ・メンバを char*& 引数にバインドします。2 番目のコンストラクタは、リファレンス・データ・メンバを String_var 引数が保持する char* にバインドし、文字列で string_free() を呼び出します。3 番目のコンストラクタはコピー・コンストラクタで、その引数のデータ・メンバにバインドされる同じ char* に、リファレンス・データ・メンバをバインドします。

ほかの String_out から代入すると、String_out 引数によって参照される char* が、データ・メンバによって参照される char* にコピーされます。char* のオーバーロードの代入演算子は、char* 引数をデータ・メンバに代入するだけです。const char* のオーバーロードの代入演算子は、引数を複製して結果をデータ・メンバに代入します。代入を行っても、以前に保持されていた文字列はまったく削除されません。この意味で、String_out 型の動作は、char* とまったく同じです。char*& 変換演算子は、データ・メンバを返します。ptr() メンバ関数もデータ・メンバを返しますが、暗黙的な変換の実行を避ける場合に使用できます。

String_var から String_out への代入はできません。これは、メモリ管理にあいまいさが生じるためです。特に、String_var によって所有される文字列を String_out が取得するときに、コピーが行われるか、コピーなしで行われるかどうかを判別することは不可能です。String_var からの代入が不可能なため、アプリケーション開発者は次のように明示的に指定しなければなりません。

```
// C++
void
A::op(String_out arg)
{
    String_var s = string_dup("some string");
    ...
    out = s;                // 不可、次のどちらかで指定
    out = string_dup(s);    // 1: コピー、または
    out = s._retn();        // 2: 借用
}
```

コメントで「1」とマークされた行の場合、呼び出し側は String_var が所有する文字列を明示的にコピーし、結果を out 引数に代入しています。また、呼び出し側は、コメントで「2」とマークされた行で示すように、String_var に文字列の所有権を強制的に放棄させるというテクニックも使用できます。これにより、メモリ管理のエラーを発生させずに out 引数に文字列を返すことができます。

引数の受け渡しの考慮事項

パラメータ渡しモードのマッピングでは、効率性と簡便性の両方のバランスを調整します。プリミティブ型、列挙、およびオブジェクト・リファレンスの場合、モードは単純で、プリミティブと列挙用の型 P、およびインターフェイス型 A 用の型 A_ptr を渡します。

集約型は複雑です。これは、いつ、どのようにパラメータのメモリを割り当てるか、または割り当て解除するかという問題があるためです。in パラメータのマッピングは単純です。これは、パラメータの記憶域が呼び出し側によって割り当てられ、読み取り専用であるためです。out パラメータと inout パラメータのマッピングは、ほかと比較して問題が多くあります。可変長型の場合、呼び出し先は記憶域の一部またはすべてを割り当てる必要があります。呼び出し側の割り当てという点では、3 つの浮動小数点値のメンバを格納する構造体として表される Point 型など、固定長型の方がスタック割り当てが可能なので好ましいと言えます。

両方の割り当て方法に対応するために、分割割り当てで生じる可能性のある混同を未然に排除します。また、コピーを実行するときに、マッピングで固定長の集約体 T 用の T& と、可変長の T 用の T*& との間に混同が生じないようにする必要があります。このような手法を採る理由は、構造体が固定長か可変長かによって構造体の形式が異なることにあります。ただし、呼び出し側が管理対象の型 T_var を使用する場合でも、マッピングは T_var& で一貫しています。

out パラメータと inout パラメータのマッピングでは、T_var が渡されるときにパラメータ内の以前の可変長データをすべて割り当て解除する必要があります。これらのパラメータの初期値がオペレーションに送信されない場合でも、BEA Tuxedo には、out パラメータがあります。これは、パラメータに以前の呼び出しの結果が格納されている可能性があるためです。T_out 型が提供される目的は、T_var 型からの変換時にアクセスできない記憶域を解放するのに必要なフックをインプリメンテーションに付与することです。次に、この動作の例を示します。

```
// IDL
struct S { string name; float age; };
void f(out S p);

// C++
S_var s;
f(s);
// use s
f(s);        // 最初の結果は解放

S *sp;       // out に渡す前の初期化は不要
f(sp);
// use sp
delete sp;   // 次の呼び出しで古い値が解放されることは想定不可能
f(sp);
```

out パラメータと inout パラメータの前の値を暗黙的に割り当て解除できるのは、T_var 型だけです。

```
// IDL
void q(out string s);

// C++
char *s;
for (int i = 0; i < 10; i++)
q(s);          // メモリ・リーク
```

ループ内で q 関数を呼び出すたびに、メモリ・リークが発生します。これは、呼び出し側が out 結果で string_free を呼び出していないことが原因です。これを修正するには、次に示す 2 つの方法があります。

```
// C++
char *s;
String_var svar;
for (int i = 0 ; i < 10; i++) {
    q(s);
    string_free(s);    // 明示的な割り当て解除
    // または
    q(svar);           // 暗黙的な割り当て解除
}
```

out パラメータに通常の char* を使用する場合、呼び出し側は変数を毎回 out パラメータとして再利用する前に、明示的にメモリを割り当て解除する必要があります。一方、String_var を使用した場合は、変数を out パラメータとして使用するたびに、割り当てがすべて暗黙的に解除されます。

可変長データは、上書きされる前に明示的に解放しなければなりません。たとえば、inout 文字列パラメータへの割り当ての前に、オペレーションのインプリメンタは最初に古い文字データから削除します。同様に、inout インターフェイス・パラメータは、再割り当てされる前に解放する必要があります。パラメータの記憶域を確実に解放する方法としては、次の例に示すように、記憶域をローカル T_var 変数に割り当てて自動的に解放が行われるようにすることです。

```
// IDL
interface A;
void f(inout string s, inout A obj);

// C++
void Aimpl::f(char *&s, A_ptr &obj) {
    String_var s_tmp = s;
    s = /* new data */;
    A_var obj_tmp = obj;
    obj = /* new reference */
}
```

パラメータがポインタ (T*) またはポインタ (T*&) へのリファレンスとして受け渡しされる場合、アプリケーションは NULL ポインタを受け渡しすることができません。これを実行すると、結果は未定義になります。特に、次のどちらかに該当する場合、呼び出し側は NULL ポインタを渡すことができません。

- in および inout 文字列
- in および inout 配列 (最初の要素へのポインタ)

ただし、呼び出し側は、out パラメータの NULL 値でポインタへのリファレンスを渡すことはできます。これは、呼び出し先が値を確認せずに上書きするためです。呼び出し先は、次のいずれかに該当する場合、NULL ポインタを渡すことができません。

- out および戻り値の可変長の構造体
- out および戻り値の可変長の共用体
- out および戻り値の文字列
- out および戻り値のシーケンス
- out および戻り値の可変長または固定長の配列

- out および戻り値の any

オペレーションのパラメータおよびシグニチャ

表 0-7 には、受け渡しされる型に応じた、基本 OMG IDL パラメータ渡しモードおよび戻り値の型のマッピングを示します。表 0-8 には、T_var 型の場合の同様の情報を示します。表 0-8 は、あくまで参考用に提供されています。これは、T_out 型がすべての out パラメータの実際のパラメータ型として使用される点を除けば、クライアントとサーバのオペレーションのシグニチャが、表 0-7 で示すパラメータ渡しモードでコーディングされるためです。

また、T_var 型では、この型を直接渡すのに必要な変換演算子をサポートします。呼び出し側は、T_var 型または表 0-7 で示す基本型のインスタンスを常に渡す必要があります。呼び出し先は、その T_out パラメータが実際に表 0-7 の対応する、基となる型であるように扱う必要があります。

表 0-7 の中で、固定長の配列が適用対象となるのは、out パラメータの型が戻り値と異なる場合のみです。これが必要となるのは、C++ では関数で配列を返すことができないためです。
マッピングは、配列のスライスへのポインタを返します。
配列のスライスとは、指定された元の配列の最初のサイズを除いたすべてのサイズを持つ配列のことです。

表 0-7 基本的な引数と結果の受け渡し

データ型	In	Inout	Out	戻り値
short	Short	Short&	Short&	Short
long	Long	Long&	Long&	Long
unsigned short	UShort	UShort&	UShort&	UShort
unsigned long	ULong	ULong&	ULong&	ULong
float	Float	Float&	Float&	Float

表 0-7 基本的な引数と結果の受け渡し (続き)

データ型	In	Inout	Out	戻り値
double	Double	Double&	Double&	Double
boolean	Boolean	Boolean&	Boolean&	Boolean
char	Char	Char&	Char&	Char
wchar	WChar	WChar&	WChar	Octet
octet	Octet	Octet&	Octet&	Octet
enum	enum	enum&	enum&	enum
object reference ptr (下記の注記を参照)	objref_ptr	objref_ptr&	objref_ptr&	objref_ptr
struct (固定長)	const struct&	struct&	struct&	struct
struct (可変長)	const struct&	struct&	struct*&	struct*
union (固定長)	const union&	union&	union&	union
union (可変長)	const union&	union&	union*&	union*
string	const char*	char*&	char*&	char*
wstring	const WChar	WChar*&	Wchar*&	WChar*
sequence	const sequence&	sequence&	sequence*&	sequence*
array (固定長)	const array	array	array	array slice* (下記の注記を参照)
array (可変長)	const array	array	array slice*&	array slice*
any	const any&	any&	any*&	any*

注記　データ型 object reference ptr には、擬似オブジェクト・リファレンスも含まれています。配列スライスの戻り値は、元の配列の最初のサイズを除いたすべてのサイズを持つ配列です。

呼び出し側は、in 引数として渡される引数すべてに記憶域を提供する必要があります。

表 0-8 T_var の引数と結果の受け渡し

データ型	In	Inout	Out	戻り値
object reference var (下記の注記を参照)	const objref_var&	objref_var&	objref_var&	objref_var
struct_var	const struct_var&	struct_var&	struct_var&	struct_var
union_var	const union_var&	union_var&	union_var&	union_var
string_var	const string_var&	string_var&	string_var&	string_var
sequence_var	const sequence_var&	sequence_var&	sequence_var&	sequence_var
array_var	const array_var&	array_var&	array_var&	array_var
any_var	const any_var&	any_var&	any_var&	any_var

注記　データ型 object reference var には、擬似オブジェクト・リファレンスも含まれています。

表 0-9 および表 0-10 では、inout パラメータと out パラメータに関連付けられた記憶域、および戻り値の結果に対する呼び出し側の作業について説明します。

表 0-9 引数の記憶域に対する呼び出し側の作業

型	inout パラメータ	out パラメータ	戻り値の結果
short	1	1	1
long	1	1	1
unsigned short	1	1	1
unsigned long	1	1	1
float	1	1	1

表 0-9 引数の記憶域に対する呼び出し側の作業 (続き)

型	inout パラメータ	out パラメータ	戻り値の結果
double	1	1	1
boolean	1	1	1
char	1	1	1
wchar	1	1	1
octet	1	1	1
enum	1	1	1
object reference ptr	2	2	2
struct (固定長)	1	1	1
struct (可変長)	1	3	3
union (固定長)	1	1	1
union (可変長)	1	3	3
string	4	3	3
wstring	4	3	3
sequence	5	3	3
array (固定長)	1	1	6
array (可変長)	1	6	6
any	5	3	3

表 0-10 引数の受け渡しのケース

ケース	
1	呼び出し側は必要な記憶域をすべて割り当てます。ただし、パラメータ自体の中でカプセル化および管理できる記憶域は除きます。inout パラメータの場合、呼び出し側は初期値を指定し、呼び出し先はその値を変更できます。out パラメータの場合、呼び出し側は記憶域を割り当てますが初期化する必要はなく、呼び出し先が値を設定します。関数は値で戻り値を返します。
2	呼び出し側は、オブジェクト・リファレンスの記憶域を割り当てます。inout パラメータの場合、呼び出し側は初期値を指定します。呼び出し先が inout パラメータを再割り当てする場合は、まず、元の入力値で CORBA::release を呼び出します。inout として渡されたオブジェクト・リファレンスを引き続き使用するには、呼び出し側は最初にリファレンスを複製する必要があります。そして、out をすべて解放してから、オブジェクト・リファレンスを返します。ほかの構造体に埋め込まれたオブジェクト・リファレンスはすべて、各構造体自体によって自動的に解放されます。
3	out パラメータの場合、呼び出し側はポインタを割り当て、リファレンスでそれを呼び出し先に渡します。呼び出し先は、パラメータの型の有効なインスタンスを指すようにポインタを設定します。戻り値の場合は、呼び出し先は同様のポインタを返します。どちらの場合でも、呼び出し先は NULL ポインタを返すことはできません。 どちらの場合でも、呼び出し側は戻り値の記憶域を解放する必要があります。ローカルまたはリモートの透過性を維持するには、呼び出し先が呼び出し側と同じアドレス空間にあるか、異なるアドレス空間にあるかどうかに関係なく、呼び出し側は常に戻り値の記憶域を解放しなければなりません。要求が完了した後は、呼び出し側は戻り値の記憶域の値を変更することはできません。これを行う場合は、呼び出し側は最初に戻り値のインスタンスを新しいインスタンスにコピーしてから、新しいインスタンスを変更する必要があります。

表 0-10 引数の受け渡しのケース (続き)

ケース	
4	inout 文字列の場合、呼び出し側は、入力文字列とそれを指す char* の両方に記憶域を割り当てます。呼び出し先が入力文字列の割り当てを解除し、新しい記憶域を指す char* を再割り当てして出力値を保持する可能性があるため、呼び出し側は string_alloc() を使用して入力文字列を割り当てる必要があります。そのため、out 文字列のサイズは in 文字列のサイズに制限されません。呼び出し側は、string_free() を使用して out の記憶域を削除する必要があります。呼び出し先は、inout、out、または戻り値に対して NULL ポインタを返すことはできません。
5	inout シーケンスおよび any の場合、シーケンスまたは any を作成した Boolean 解放パラメータの状態によっては、シーケンスまたは any を割り当てまたは変更することにより、再割り当てが行われる前に、所有している記憶域の割り当てが解除される場合があります。
6	out パラメータの場合、呼び出し側は配列スライス (元の配列の最初のサイズを除いたすべてのサイズを持つ配列) へのポインタを割り当て、リファレンスでポインタを呼び出し先に渡します。呼び出し先は、配列の有効なインスタンスを指すようにポインタを設定します。 戻り値の場合は、呼び出し先は同様のポインタを返します。どちらの場合でも、呼び出し先は NULL ポインタを返すことはできません。どちらの場合でも、呼び出し側は戻り値の記憶域を解放する必要があります。 ローカルまたはリモートの透過性を維持するには、呼び出し先が同じアドレス空間にあるか、異なるアドレス空間にあるかどうかに関係なく、呼び出し側は常に戻り値の記憶域を解放しなければなりません。要求が完了した後は、呼び出し側は戻り値の記憶域の値を変更することはできません。これを行う場合は、呼び出し側は最初に戻り値の配列インスタンスを新しい配列インスタンスにコピーしてから、新しいインスタンスを変更する必要があります。

章

14 CORBA API

この章では、C++ と拡張 C++ でのコア・メンバ関数の BEA Tuxedo インプリメンテーションについて説明します。また、擬似オブジェクトとその C++ クラスとの関係についても説明します。擬似オブジェクトとは、ネットワーク経由で転送不可能なオブジェクト・リファレンスのことです。擬似オブジェクトはその他のオブジェクトと似ていますが、ORB によって所有されるため拡張ができません。

注記　この章の一部の情報は、Object Management Group (OMG) の「Common Object Request Broker: Architecture and Specification, Revision 2.4.2」(2001 年 2 月) から、OMG の許可を得て転載しています。

グローバル・クラス

次の BEA Tuxedo クラスは、スコープ内においてグローバルです。

- CORBA
- Tobj

上記のクラスには、BEA Tuxedo の開発で使用する定義済みの型、クラス、および関数が含まれています。

CORBA クラスには、CORBA で定義されている、オブジェクト・リクエスト・ブローカ (ORB) を使用する際に不可欠なクラス、データ型、およびメンバ関数があります。BEA Tuxedo の CORBA への拡張は、Tobj C++ クラス

に含まれています。Tobj クラスには、BEA Tuxedo で CORBA への拡張として用意されているデータ型、入れ子になったクラス、およびメンバ関数があります。

BEA Tuxedo 製品で CORBA のデータ型とメンバ関数を使用するには、`CORBA::` 接頭辞が必要です。たとえば、`Long` の場合は `CORBA::Long` となります。同様に、BEA Tuxedo 製品で Tobj の入れ子になったクラスとメンバ関数を使用するには、`Tobj::` 接頭辞が必要です。たとえば、FactoryFinder の場合は `Tobj::FactoryFinder` となります。

擬似オブジェクト

擬似オブジェクトは、CORBA クラス内部にあるローカル・クラスとして表されます。擬似オブジェクトおよび対応するメンバ関数の名前は、入れ子になったクラス構造で付けます。たとえば、ORB オブジェクトの場合は `CORBA::ORB`、Current オブジェクトの場合は `CORBA::Current` となります。

Any クラスのメンバ関数

ここでは、`Any` クラスの各メンバ関数について説明します。

ExceptionList メンバ関数の C++ へのマッピングは次のとおりです。

```
class CORBA
{
  class Any
  {
  public:

    Any ();
    Any (const Any&);
    Any (TypeCode_ptr tc, void *value, Boolean release =
                                                CORBA_ FALSE);
    ~Any ();
    Any & operator=(const Any&);
```

```
    void    operator<<=(Short);
    void    operator<<=(UShort);
    void    operator<<=(Long);
    void    operator<<=(ULong);
    void    operator<<=(Float);
    void    operator<<=(Double);
    void    operator<<=(const Any&);
    void    operator<<=(const char*);
    void    operator<<=(Object_ptr);
    void    operator<<=(from_boolean);
    void    operator<<=(from_char);
    void    operator<<=(from_octet);
    void    operator<<=(from_string);
    Boolean    operator>>=(Short&) const;
    Boolean    operator>>=(UShort&) const;
    Boolean    operator>>=(Long&) const;
    Boolean    operator>>=(ULong&) const;
    Boolean    operator>>=(Float&) const;
    Boolean    operator>>=(Double&) const;
    Boolean    operator>>=(Any&) const;
    Boolean    operator>>=(char*&) const;
    Boolean    operator>>=(Object_ptr&) const;
    Boolean    operator>>=(to_boolean) const;
    Boolean    operator>>=(to_char) const;
    Boolean    operator>>=(to_octet) const;
    Boolean    operator>>=(to_object) const;
    Boolean    operator>>=(to_string) const;

    TypeCode_ptr type()const;
    void        replace(TypeCode_ptr, void *, Boolean);
    void        replace(TypeCode_ptr, void *);
    const void * value() const;

  };
}; //CORBA
```

CORBA::Any::Any()

概要 Any オブジェクトを作成します。

C++ バインディング `CORBA::Any::Any()`

引数 特にありません。

説明 `CORBA::Any` クラスのデフォルトのコンストラクタです。`tc_null` 型および値 0 (ゼロ) の TypeCode で `Any` オブジェクトを作成します。

戻り値 特にありません。

CORBA::Any::Any(const CORBA::Any & InitAny)

概要 ほかの Any オブジェクトのコピーである Any オブジェクトを作成します。

C++ バインディング

```
CORBA::Any::Any(const CORBA::Any & InitAny)
```

引数

InitAny

CORBA::Any を参照してコピーします。

説明 CORBA::Any クラスのコピー・コンストラクタです。このコンストラクタは、渡される Any の TypeCode リファレンスを複製します。

コピーする型は、コピー元の Any オブジェクトの release フラグによって決まります。release が CORBA_TRUE と評価された場合、コンストラクタはパラメータの値をディープ・コピーします。release が CORBA_FALSE と評価された場合、コンストラクタはパラメータの値をシャロー・コピーします。シャロー・コピーを使用すると、メモリの割り当てをより高度に制御できます。ただし、呼び出し側は、解放済みのメモリを Any が使用していないことを確認する必要があります。

戻り値 特にありません。

CORBA::Any::Any(TypeCode_ptr TC, void * Value, Boolean Release)

概要 TypeCode および値で Any オブジェクトを作成します。

C++ バインディング

```
CORBA::Any::Any(TypeCode_ptr TC, void * Value, Boolean Release)
```

引数

`TC`

作成する型を指定した、TypeCode 擬似オブジェクト・リファレンスへのポインタ。

`Value`

Any オブジェクトの作成で使用するデータへのポインタ。この引数のデータ型は、指定された TypeCode と一致していなければなりません。

`Release`

Value 引数で指定されたメモリの所有権を Any が想定するかどうかを指定します。Release が CORBA_TRUE の場合、Any は所有権を想定します。Release が CORBA_FALSE の場合、Any は所有権を想定しません。この場合、Value 引数が指すデータは、割り当て時または破棄時に解放されません。

説明 このコンストラクタは、型セーフでない Any インターフェイスで使用されます。指定された TypeCode オブジェクト・リファレンスを複製してから、Any オブジェクト内の値が指すデータを挿入します。

戻り値 特にありません。

CORBA::Any::~Any()

概要 Any のデストラクタ。

C++ バインディング CORBA::Any::~Any()

引数 特にありません。

説明 このデストラクタは、Release フラグが CORBA_TRUE に指定されている場合に、CORBA::Any が保持しているメモリを解放します。また、Any に含まれる TypeCode 擬似オブジェクト・リファレンスも解放します。

戻り値 特にありません。

CORBA::Any & CORBA::Any::operator=(const CORBA::Any & InitAny)

概要 Any の代入演算子。

C++ バインディング

```
CORBA::Any & CORBA::Any::operator=(const CORBA::Any & InitAny)
```

引数

InitAny

割り当てに使用するための Any のリファレンス。割り当てで使用する Any によって、Any が Value でメモリの所有権を想定するかどうかが決まります。Release が CORBA_TRUE の場合、Any は所有権を想定して InitAny 引数の値をディープ・コピーします。Release が CORBA_FALSE の場合、Any は InitAny 引数の値をシャロー・コピーします。

説明 これは、Any クラスの代入演算子です。このメンバ関数のメモリ管理は、Release フラグの現在の値によって決まります。また、Release フラグの現在の値によって、現在のメモリが割り当て前に解放されるかどうかも決まります。現在の Release フラグが CORBA_TRUE の場合、Any は保持していたすべての値を解放します。現在の Release フラグが CORBA_FALSE の場合、Any は保持していた値を解放しません。

戻り値 InitAny のコピーを保持する Any を返します。

void CORBA::any::operator<<=()

概要 型セーフな Any の挿入演算子。

C++ バインディング

```
void CORBA::Any::operator<<=(CORBA::Short Value)
void CORBA::Any::operator<<=(CORBA::UShort Value)
void CORBA::Any::operator<<=(CORBA::Long Value)
void CORBA::Any::operator<<=(CORBA::Ulong Value)
void CORBA::Any::operator<<=(CORBA::Float Value)
void CORBA::Any::operator<<=(CORBA::Double Value)
void CORBA::Any::operator<<=(const CORBA::Any & Value)
void CORBA::Any::operator<<=(const char * Value)
void CORBA::Any::operator<<=(Object_ptr Value)
```

引数

Value

Any に挿入する型固有の値。

説明 この挿入メンバ関数は、型セーフな挿入を実行します。Any に前の値があり、Release フラグが CORBA_TRUE の場合、メモリの割り当てを解除し、前の TypeCode オブジェクトを解放します。次に、Value パラメータで渡された値をコピーして Any に新しい値を挿入します。これにより、適切な TypeCode リファレンスが複製されます。

戻り値 特にありません。

CORBA::Boolean CORBA::Any::operator>>=()

概要 型セーフな Any の抽出演算子。

C++ バインディング

```
CORBA::Boolean CORBA::Any::operator>>=(
                  CORBA::Short & Value) const
CORBA::Boolean CORBA::Any::operator>>=(
                  CORBA::UShort & Value) const
CORBA::Boolean CORBA::Any::operator>>=(
                  CORBA::Long & Value) const
CORBA::Boolean CORBA::Any::operator>>=(
                  CORBA::Ulong & Value) const
CORBA::Boolean CORBA::Any::operator>>=(
                  CORBA::Float & Value) const
CORBA::Boolean CORBA::Any::operator>>=(
                  CORBA::Double & Value) const
CORBA::Boolean CORBA::Any::operator>>=(CORBA::Any & Value) const
CORBA::Boolean CORBA::Any::operator>>=(char * & Value) const
CORBA::Boolean CORBA::Any::operator>>=(Object_ptr & Value) const
```

引数 Value 引数は、Any オブジェクトに格納されている値の出力を受け取る関連オブジェクトのリファレンスです。

説明 この抽出メンバ関数は、型セーフな抽出を実行します。Any オブジェクトに指定の型がある場合、このメンバ関数は、Any のポインタを出力リファレンス値 Value に割り当て、CORBA_TRUE を返します。Any に適切な型がない場合は、CORBA_FALSE を返します。記憶域は Any オブジェクトによって所有および管理されているため、呼び出し側は記憶域の解放または削除を試行しないでください。Value 引数は、Any オブジェクトに格納されている値の出力を受け取る関連オブジェクトのリファレンスです。Any オブジェクトに適切な型がない場合、値は変更されません。

戻り値 特定の型の値が Any にあった場合は CORBA_TRUE。特定の型の値が Any になかった場合は CORBA_FALSE。

CORBA::Any::operator<<=()

概要 Any の型セーフな挿入演算子。

C++ バインディング

```
void CORBA::Any::operator<<=(from_boolean Value)
void CORBA::Any::operator<<=(from_char Value)
void CORBA::Any::operator<<=(from_octet Value)
void CORBA::Any::operator<<=(from_string Value)
```

引数

Value

Any に挿入する値を含んだ関連オブジェクト。

説明 これらの挿入メンバ関数は、Any に CORBA::Boolean、CORBA::Char、または CORBA::Octet リファレンスを型セーフで挿入します。Any に前の値があり、その Release フラグが CORBA_TRUE の場合、メモリの割り当てを解除し、前の TypeCode オブジェクトを解放します。次に、Value パラメータで渡された値をコピーして Any オブジェクトに新しい値を挿入します。これにより、適切な TypeCode リファレンスが複製されます。

戻り値 特にありません。

CORBA::Boolean CORBA::Any::operator>>=()

概要 Any の型セーフな抽出演算子。

C++ バインディング

```
CORBA::Boolean CORBA::Any::operator>>=(to_boolean Value) const
CORBA::Boolean CORBA::Any::operator>>=(to_char Value) const
CORBA::Boolean CORBA::Any::operator>>=(to_octet Value) const
CORBA::Boolean CORBA::Any::operator>>=(to_object Value) const
CORBA::Boolean CORBA::Any::operator>>=(to_string Value) const
```

引数

`Value`

Any オブジェクトに格納されている値の出力を受け取る関連オブジェクトのリファレンス。Any オブジェクトに適切な型がない場合、値は変更されません。

説明 これらの抽出メンバ関数は、Any から CORBA::Boolean、CORBA::Char、CORBA::Octet、CORBA::Object、または String リファレンスを型セーフで抽出します。これらの関数は、Any クラス内で入れ子になったヘルパです。その目的は、C++ では Boolean 型、char 型、octet 型の識別が不要なため、OMG IDL のこれらの型を識別して抽出することです。

戻り値 Any オブジェクトに指定の型がある場合、このメンバ関数は、Any オブジェクト・リファレンスの値を出力変数 Value に割り当て、CORBA_TRUE を返します。Any オブジェクトに適切な型がない場合は、CORBA_FALSE を返します。

CORBA::TypeCode_ptr CORBA::Any::type() const

概要 Any の TypeCode アクセサ。

C++ バインディング

```
CORBA::TypeCode_ptr CORBA::Any::type();
```

引数 特にありません。

説明 この関数は、Any に関連付けられた TypeCode オブジェクトの TypeCode_ptr 擬似オブジェクト・リファレンスを返します。TypeCode_ptr 擬似オブジェクト・リファレンスは、CORBA::release メンバ関数で解放する必要があります。または、TypeCode_var に割り当てて自動的に解放されるようにする必要があります。

戻り値 Any 内にある TypeCode_ptr。

void CORBA::Any::replace()

概要 型セーフでない Any の「挿入」。

C++ バインディング

```
void CORBA::Any::replace(TypeCode_ptr TC, void * Value,
                         Boolean Release = CORBA_FALSE);
```

引数

TC

置き換え元の Any オブジェクトの TypeCode 値を指定する TypeCode 擬似オブジェクト・リファレンス。この引数は複製されます。

Value

Any オブジェクトが指す記憶域を指定する void ポインタ。

Release

Any が指定の Value 引数を管理するかどうかを指定します。Release が CORBA_TRUE の場合、Any は所有権を想定します。Release が CORBA_FALSE の場合、Any は所有権を想定しません。この場合、Value パラメータが指すデータは、割り当て時または破棄時に解放されません。

説明 これらのメンバ関数は、渡された TC および Value 引数の値に、現在 Any に格納されているデータおよび TypeCode 値を置き換えます。この関数による置き換えは型セーフではありません。つまり、TypeCode 値と Value 引数が指す記憶域のデータ型との一貫性は、呼び出し側が維持することになります。

Release が CORBA_TRUE の場合、この関数は Any オブジェクト内の既存の TypeCode 擬似オブジェクト解放し、Any オブジェクト・リファレンスが指す記憶域を解放します。

戻り値 特にありません。

Context メンバ関数

Context は、メソッド呼び出しに関連付けられたオプションのコンテキスト情報を提供します。

ExceptionList メンバ関数の C++ へのマッピングは次のとおりです。

```
class CORBA
{
  class Context
  {
    public:
      const char *context_name() const;
      Context_ptr parent() const;

      void create_child(const char *, Context_out);

      void set_one_value(const char *, const Any &);
      void set_values(NVList_ptr);
      void delete_values(const char *);
      void get_values(
                  const char *,
                  Flags,
                  const char *,
                  NVList_out
      );
  }; // Context
}// CORBA
```

メモリ管理

Context には、次の特別なメモリ管理規則があります。

- `context_name` および `parent` 関数の戻り値の所有権は Context が保持します。したがって、呼び出し側はこれらの戻り値を解放しないでください。

ここでは、各 Context メンバ関数について説明します。

CORBA::Context::context_name

概要 Context オブジェクトの名前を返します。

C++ バインディング

```
Const char * CORBA::Context::context_name () const;
```

引数 特にありません。

説明 このメンバ関数は、Context オブジェクトの名前を返します。Context オブジェクト・リファレンスは、戻り値 char * のメモリを所有します。このメモリは変更できません。

戻り値 メンバ関数が成功した場合、Context オブジェクトの名前を返します。Context オブジェクトが `CORBA::Context::create_child` の呼び出しで作成された子 Context でない場合、値は空になることがあります。

Context オブジェクトに名前がない場合、空の文字列になります。

CORBA::Context::create_child

概要 Context オブジェクトの子を作成します。

C++ バインディング

```
void CORBA::Context::create_child (
       const char *                CtxName,
       CORBA::Context_out           CtxObject);
```

引数

`CtxName`
Context リファレンスに関連付ける名前。

`CtxObject`
新しく作成する Context オブジェクト・リファレンス。

例外 `CORBA::NO_MEMORY`

説明 このメンバ関数は、Context オブジェクトの子を作成します。子 Context オブジェクトの検索は、一致する親コンテキストのプロパティ名や、必要に応じてコンテキスト・ツリーなどを対象に行われます。

戻り値 特にありません。

関連項目
`CORBA::ORB::get_default_context`
`CORBA::release`

CORBA::Context::delete_values

概要 Context オブジェクトの指定された属性の値を削除します。

C++ バインディング

```
void CORBA::Context::delete_values (
      const char *      AttrName);
```

引数

`AttrName`

削除する値の属性名。この引数の後にワイルドカード文字 (*) を付けると、ワイルドカード文字の前にある文字列に一致する名前がすべて削除されます。

例外 属性が空の文字列の場合、`CORBA::BAD_PARAM`。
削除する属性が見つからなかった場合、`CORBA::BAD_CONTEXT`。

説明 このメンバ関数は、Context オブジェクトの属性の名前付きの値を削除します。ただし、親がある場合、再帰的に親に対して同じ処理は行いません。

戻り値 特にありません。

関連項目

`CORBA::Context::create_child`
`CORBA::ORB::get_default_context`

CORBA::Context::get_values

概要 指定されたスコープ内で Context オブジェクトの属性値を取得します。

C++ バインディング

```
void CORBA::Context::get_values (
      const char *              StartScope,
      CORBA::Flags              OpFlags,
      const char *              AttrName,
      CORBA::NVList_out         AttrValues);
```

引数

`StartScope`
: 指定されたプロパティの検索を開始する Context オブジェクト・レベル。レベルは、コンテキスト名または検索が開始される `parent` です。値が 0 (ゼロ) の場合、現在の Context オブジェクトで検索が開始されます。

`OpFlags`
: 有効なオペレーション・フラグは、`CORBA::CTX_RESTRICT_SCOPE` のみです。このフラグを指定すると、オブジェクト・インプリメンテーションによってプロパティの検索が現在のスコープのみに制限されます。したがって、一連の親コンテキストのプロパティの検索は再帰的に実行されません。このフラグを指定しない場合は、スコープを広げて一致が見つかるか、または検索対象がすべてのレベルになるまで検索が続行されます。

`AttrName`
: 戻り値の属性名。この引数の後にワイルドカード文字 (*) を付けると、ワイルドカード文字の前にある文字列に一致する名前がすべて返されます。

`AttrValues`
: 指定された属性値を受け取り、`NVList` オブジェクトを返します。属性値のリスト内の各項目は `NamedValue` です。

例外 属性が空の文字列の場合、`CORBA::BAD_PARAM`。
一致する属性が見つからなかった場合、`CORBA::BAD_CONTEXT`。
動的メモリ割り当てに失敗した場合、`CORBA::NO_MEMORY`。

説明 このメンバ関数は、Context オブジェクトの指定された属性の値を取得します。値は NVList オブジェクトとして返します。このオブジェクトが不要になったときは、`CORBA::release` メンバ関数を使用して解放する必要があります。

戻り値　特にありません。

関連項目　`CORBA::Context::create_child`
`CORBA::ORB::get_default_context`

CORBA::Context::parent

概要 Context オブジェクトの親コンテキストを返します。

C++ バインディング

```
CORBA::Context_ptr CORBA::Context::parent () const;
```

引数 特にありません。

説明 このメンバ関数は、Context オブジェクトの親コンテキストを返します。Context オブジェクトの親は Context が所有する属性で、呼び出し側が変更または解放することはできません。`CORBA::Context::create_child` メンバ関数で Context オブジェクトを作成するまで、親はニルです。

戻り値 メンバ関数が成功した場合、Context オブジェクトの親コンテキストを返します。親コンテキストがニルの場合もあります。`CORBA::is_nil` メンバ関数を使用すると、オブジェクト・リファレンスがニルかどうかをテストできます。

メンバ関数が失敗した場合、例外がスローされます。`CORBA::is_nil` メンバ関数を使用すると、オブジェクト・リファレンスがニルかどうかをテストできます。

CORBA::Context::set_one_value

概要 Context オブジェクトの指定の属性値を設定します。

C++ バインディング

```
void CORBA::Context::set_one_value (
     const char *            AttrName,
     const CORBA::Any &       AttrValue);
```

引数

`AttrName`
設定する属性の名前。

`AttrValue`
属性の値。現在、BEA Tuxedo システムでは文字列型のみをサポートしています。したがって、このパラメータは、`CORBA::Any` オブジェクト内で文字列型で指定する必要があります。

例外 `AttrName` が空の文字列の場合、または `AttrValue` に文字列型がない場合、`CORBA::BAD_PARAM`。
動的メモリ割り当てに失敗した場合、`CORBA::NO_MEMORY`。

説明 このメンバ関数は、Context オブジェクトの指定された属性値を設定します。現在、Context オブジェクトでサポートされているのは文字列値のみです。Context オブジェクトに属性名が設定済みの場合、最初にその属性名が削除されます。

戻り値 特にありません。

関連項目

`CORBA::Context::get_values`
`CORBA::Context::set_values`

CORBA::Context::set_values

概要 Context オブジェクトの指定の属性値を設定します。

C++ バインディング

```
void CORBA::Context::set_values (
     CORBA::NVList_ptr         AttrValue);
```

引数

`AttrValues`
属性の名前および値。現在、BEA Tuxedo システムでは文字列型のみをサポートしています。したがって、リスト内の NamedValue オブジェクトはすべて、`CORBA::Any` オブジェクト内で文字列型で指定する必要があります。

例外 属性値の中に文字列型でない値がある場合、`CORBA::BAD_PARAM`。
動的メモリ割り当てに失敗した場合、`CORBA::NO_MEMORY`。

説明 このメンバ関数は、Context オブジェクトの指定の属性値を設定します。`CORBA::NVList` メンバ関数には、設定するプロパティの名前と値のペアが入ります。

戻り値 特にありません。

関連項目
`CORBA::Context::get_values`
`CORBA::Context::set_one_value`

ContextList メンバ関数

ContextList を使用すると、クライアント・アプリケーションまたはサーバ・アプリケーションでコンテキスト文字列のリストを提供できるようになります。このリストは、Request 呼び出しで指定する必要があります。Request メンバ関数については、「第 14 章の 126 ページ「Request メンバ関数」」を参照してください。

ContextList と Context は、前者が必要に応じて Request 呼び出しで検索または送信されるコンテキスト文字列値のみを提供するのに対し、後者は取得する文字列値を提供する点で異なります。Context メンバ関数については、「Context メンバ関数」を参照してください。

ExceptionList メンバ関数の C++ へのマッピングは次のとおりです。

```
class CORBA
{
  class ContextList
  {
    public:
      Ulong count ();
      void add(const char* ctxt);
      void add_consume(char* ctxt);
      const char* item(Ulong index);
      Status remove(Ulong index);
   }; // ContextList
}// CORBA
```

CORBA::ContextList:: count

概要 リスト内の現在の項目数を取得します。

C++ バインディング

```
Ulong count ();
```

引数 特にありません。

例外 関数が失敗した場合、例外がスローされます。

説明 このメンバ関数は、リスト内の現在の項目数を取得します。

戻り値 関数が成功した場合、戻り値はリスト内の項目数です。リストを作成したばかりで、ContextList オブジェクトを追加していない場合は、0 (ゼロ) が返されます。

関連項目

```
CORBA::ContextList::add
CORBA::ContextList::add_consume
CORBA::ContextList::item
CORBA::ContextList::remove
```

CORBA::ContextList::add

概要 名前の付いていない項目で ContextList オブジェクトを作成します。これは、flags 属性のみを設定したオブジェクトです。

C++ バインディング

```
void add(const char* ctxt);
```

引数

ctxt

char* によって参照されるメモリ位置を定義します。

例外 メンバ関数が失敗した場合、CORBA::NO_MEMORY 例外がスローされます。

説明 このメンバ関数は、名前の付いていない項目で ContextList オブジェクトを作成します。これは、flags 属性のみを設定したオブジェクトです。

ContextList オブジェクトは動的に拡張するので、そのサイズをアプリケーションでトラッキングする必要がありません。

戻り値 関数が成功した場合、戻り値は新しく作成された ContextList オブジェクトへのポインタです。

関連項目

CORBA::ContextList::add_consume
CORBA::ContextList::count
CORBA::ContextList::item
CORBA::ContextList::remove

CORBA::ContextList::add_consume

概要 ContextList オブジェクトを作成します。

C++ バインディング

```
void add_consume(const char* ctxt);
```

引数 ctxt

char* によって参照されるメモリ位置を定義します。

例外 メンバ関数が失敗した場合、例外が発生します。

説明 このメンバ関数は、ContextList オブジェクトを作成します。

ContextList オブジェクトは動的に拡張するので、そのサイズをアプリケーションでトラッキングする必要がありません。

戻り値 関数が成功した場合、戻り値は新しく作成された ContextList オブジェクトへのポインタです。

関連項目

```
CORBA::ContextList::add
CORBA::ContextList::count
CORBA::ContextList::item
CORBA::ContextList::remove
```

CORBA::ContextList::item

概要 渡されたインデックスに基づいて ContextList オブジェクトへのポインタを取得します。

C++ バインディング

```
const char* item(ULong index);
```

引数

`index`

ContextList オブジェクトへのインデックス。インデックスの基数はゼロです。

例外 この関数が失敗した場合、`BAD_PARAM` 例外がスローされます。

説明 このメンバ関数は、渡されたインデックスに基づいて ContextList オブジェクトへのポインタを取得します。関数では、ゼロを基数にしたインデックスを使用します。

戻り値 関数が成功した場合、戻り値は ContextList オブジェクトへのポインタです。

関連項目

```
CORBA::ContextList::add
CORBA::ContextList::add_consume
CORBA::ContextList::count
CORBA::ContextList::remove
```

CORBA::ContextList::remove

概要 指定されたインデックスの項目を削除し、関連付けられたメモリをすべて解放してから、リストの残りの項目を順序付けし直します。

C++ バインディング

```
Status remove(ULong index);
```

引数

`Index`

ContextList オブジェクトへのインデックス。インデックスの基数はゼロです。

例外 この関数が失敗した場合、`BAD_PARAM` 例外がスローされます。

説明 このメンバ関数は、指定されたインデックスの項目を削除し、関連付けられたメモリをすべて解放してから、リストの残りの項目を順序付けし直します。

戻り値 特にありません。

関連項目

```
CORBA::ContextList::add
CORBA::ContextList::add_consume
CORBA::ContextList::count
CORBA::ContextList::item
```

NamedValue メンバ関数

NamedValue メンバ関数は、特に DII で NVList の要素としてのみ使用します。NamedValue は、(オプションの) 名前、any の値、およびラベリング・フラグを保持します。有効なフラグ値は、CORBA::ARG_IN、CORBA::ARG_OUT、および CORBA::ARG_INOUT です。

NamedValue の値は、any の標準オペレーションで操作できます。

ExceptionList メンバ関数の C++ へのマッピングは次のとおりです。

```
// C++
class NamedValue
{
  public:
    Flags        flags() const;
    const char * name() const;
    Any *        value() const;
};
```

メモリ管理

NamedValue には、次の特別なメモリ管理規則があります。

- name() および value() 関数の戻り値の所有権は NamedValue が保持します。したがって、呼び出し側はこれらの戻り値を解放しないでください。

以下の節では、NamedValue の各メンバ関数について説明します。

CORBA::NamedValue::flags

概要 NamedValue オブジェクトの flags 属性を取得します。

C++ バインディング

```
CORBA::Flags CORBA::NamedValue::flags () const;
```

引数 特にありません。

説明 このメンバ関数は、NamedValue オブジェクトの flags 属性を取得します。

戻り値 関数が成功した場合、戻り値は NamedValue オブジェクトの flags 属性です。

関数が失敗した場合、例外がスローされます。

CORBA::NamedValue::name

概要 NamedValue オブジェクトの name 属性を取得します。

C++ バインディング

```
const char * CORBA::NamedValue::name () const;
```

引数 特にありません。

説明 このメンバ関数は、NamedValue オブジェクトの name 属性を取得します。このメンバ関数が返す名前は NamedValue オブジェクトによって所有されます。そのため、変更または解放しないでください。

戻り値 関数が成功した場合、戻り値は NamedValue オブジェクトの name 属性を表す定数 Identifier オブジェクトです。

関数が失敗した場合、例外がスローされます。

CORBA::NamedValue::value

概要 NamedValue オブジェクトの value 属性へのポインタを取得します。

C++ バインディング

```
CORBA::Any * CORBA::NamedValue::value () const;
```

引数 特にありません。

説明 このメンバ関数は、NamedValue オブジェクトの value 属性を表す Any オブジェクトへのポインタを取得します。この属性は NamedValue オブジェクトによって所有されます。そのため、変更または解放しないでください。

戻り値 関数が成功した場合、戻り値は NamedValue オブジェクト内にある Any オブジェクトへのポインタです。

関数が失敗した場合、例外がスローされます。

NVList メンバ関数

NVList は NamedValue のリストです。新しい NVList は、ORB::create_list オペレーション (「CORBA::ORB::create_exception_list」を参照) で作成します。新しい NamedValue は、次のいずれかの方法で NVList の一部として作成できます。

- add ― flags のみを初期化して名前の付いていない値を作成します。
- add_item ― name と flags を初期化します。
- add_value ― name、value、および flags を初期化します。

上記の各オペレーションによって、新しい項目が返されます。

要素は、ゼロを基数にしたインデックスを介してアクセスおよび削除できます。add、add_item、add_value、add_item_consume、および add_value_consume 関数は、呼び出されるたびに NVList を長くして新しい要素を保持できるようにします。既存の要素にアクセスするには、item 関数を使用します。

```
// C++
class NVList
{
  public:
   ULong count() const;
   NamedValue_ptr add(Flags);
   NamedValue_ptr add_item(const char*, Flags);
   NamedValue_ptr add_value(const char*, const Any&, Flags);
   NamedValue_ptr item(ULong);
   void remove(ULong);
};
```

メモリ管理

NVList には、次の特別なメモリ管理規則があります。

- add、add_item、add_value、add_item_consume、add_value_consume、および item 関数の戻り値の所有権は NVList が保持します。したがって、呼び出し側はこれらの戻り値を解放しないでください。
- add_item_consume と add_value_consume 関数の char* パラメータ、および add_value_consume 関数の Any* パラメータは、NVList が使用します。NVList はこれらのパラメータをコピーして、元のパラメータをすぐに破棄します。そのため、パラメータが関数に渡された後は、呼び出し側がこれらのデータにアクセスすることはできません。基となる NamedValue の value 属性を呼び出し側が変更する場合は、NamedValue::value() オペレーションを使用します。
- remove 関数は、削除済みの NamedValue の CORBA::release も呼び出します。

以下の節では、NVList の各メンバ関数について説明します。

CORBA::NVList::add

概要 名前の付いていない項目で NamedValue オブジェクトを作成します。これは、flags 属性のみを設定したオブジェクトです。

C++ バインディング

```
CORBA::NamedValue_ptr CORBA::NVList::add (
     CORBA::Flags            Flags);
```

引数

`Flags`

渡す引数を指定するフラグ。次の値を指定できます。

```
CORBA::ARG_IN
CORBA::ARG_INOUT
CORBA::ARG_OUT
```

説明 このメンバ関数は、名前の付いていない項目で NamedValue オブジェクトを作成します。これは、flags 属性のみを設定したオブジェクトです。NamedValue オブジェクトは、呼び出された NVList オブジェクトに追加されます。

NVList オブジェクトは動的に拡張するので、そのサイズをアプリケーションでトラッキングする必要はありません。

戻り値 関数が成功した場合、戻り値は新しく作成された NamedValue オブジェクトへのポインタです。返された NamedValue オブジェクト・リファレンスは NVList が所有するため、解放しないでください。

メンバ関数が失敗した場合、`CORBA::NO_MEMORY` 例外がスローされます。

関連項目

```
CORBA::NVList::add
CORBA::NVList::add_item
CORBA::NVList::add_value
CORBA::NVList::count
CORBA::NVList::remove
```

CORBA::NVList::add_item

概要 空の value 属性を作成し、name および flags 属性を初期化して NamedValue オブジェクトを作成します。

C++ バインディング

```
CORBA::NamedValue_ptr CORBA::NVList::add_item (
     const char *    Name,
     CORBA::Flags    Flags);
```

引数

`Name`

リスト項目の名前。

`Flags`

渡す引数を指定するフラグ。次の値を指定できます。

```
CORBA::ARG_IN
CORBA::ARG_INOUT
CORBA::ARG_OUT
```

説明 このメンバ関数は、空の value 属性を作成し、パラメータとして渡す name および flags 属性を初期化して NamedValue オブジェクトを作成します。NamedValue オブジェクトは、呼び出された NVList オブジェクトに追加されます。

NVList オブジェクトは動的に拡張するので、そのサイズをアプリケーションでトラッキングする必要はありません。

戻り値 関数が成功した場合、戻り値は新しく作成された NamedValue オブジェクトへのポインタです。返された NamedValue オブジェクト・リファレンスは NVList が所有するため、解放しないでください。

メンバ関数が失敗した場合、例外がスローされます。

関連項目

```
CORBA::NVList::add
CORBA::NVList::add_value
CORBA::NVList::count
CORBA::NVList::item
CORBA::NVList::remove
```

CORBA::NVList::add_value

概要 name、value、および flags 属性を初期化して NamedValue オブジェクトを作成します。

C++ バインディング

```
CORBA::NamedValue_ptr CORBA::NVList::add_value (
    const char *          Name,
    const CORBA::Any &    Value,
    CORBA::Flags           Flags);
```

引数

`Name`

リスト項目の名前。

`Value`

リスト項目の値。

`Flags`

渡す引数を指定するフラグ。次の値を指定できます。

```
CORBA::ARG_IN
CORBA::ARG_INOUT
CORBA::ARG_OUT
```

説明 このメンバ関数は、name、value、および flags 属性を初期化して NamedValue オブジェクトを作成します。NamedValue オブジェクトは、呼び出された NVList オブジェクトに追加されます。

NVList オブジェクトは動的に拡張するので、そのサイズをアプリケーションでトラッキングする必要はありません。

戻り値 関数が成功した場合、戻り値は新しく作成された NamedValue オブジェクトへのポインタです。返された NamedValue オブジェクト・リファレンスは NVList が所有するため、解放しないでください。

メンバ関数が失敗した場合、例外が発生します。

関連項目

```
CORBA::NVList::add
CORBA::NVList::add_item
CORBA::NVList::count
CORBA::NVList::item
CORBA::NVList::remove
```

CORBA::NVList::count

概要 リスト内の現在の項目数を取得します。

C++ バインディング

```
CORBA::ULong CORBA::NVList::count () const;
```

引数 特にありません。

説明 このメンバ関数は、リスト内の現在の項目数を取得します。

戻り値 関数が成功した場合、戻り値はリスト内の項目数です。リストを作成したばかりで、NamedValue オブジェクトを追加していない場合は、0 (ゼロ) が返されます。

関数が失敗した場合、例外がスローされます。

関連項目

```
CORBA::NVList::add
CORBA::NVList::add_item
CORBA::NVList::add_value
CORBA::NVList::item
CORBA::NVList::remove
```

CORBA::NVList::item

概要 渡されたインデックスに基づいて NamedValue オブジェクトへのポインタを取得します。

C++ バインディング

```
CORBA::NamedValue_ptr CORBA::NVList::item (
     CORBA::ULong           Index);
```

引数

`Index`

NVList オブジェクトへのインデックス。インデックスの基数はゼロです。

例外 この関数が失敗した場合、`BAD_PARAM` 例外がスローされます。

説明 このメンバ関数は、渡されたインデックスに基づいて NamedValue オブジェクトへのポインタを取得します。関数では、ゼロを基数にしたインデックスを使用します。

戻り値 関数が成功した場合、戻り値は NamedValue オブジェクトへのポインタです。返された NamedValue オブジェクト・リファレンスは NVList が所有するため、解放しないでください。

関連項目

`CORBA::NVList::add`
`CORBA::NVList::add_item`
`CORBA::NVList::add_value`
`CORBA::NVList::count`
`CORBA::NVList::remove`

CORBA::NVList::remove

概要 指定されたインデックスの項目を削除し、関連付けられたメモリをすべて解放してから、リストの残りの項目を順序付けし直します。

C++ バインディング

```
void CORBA::NVList::remove (
     CORBA::ULong            Index);
```

引数

`Index`

NVList オブジェクトへのインデックス。インデックスの基数はゼロです。

例外 この関数が失敗した場合、`BAD_PARAM` 例外がスローされます。

説明 このメンバ関数は、指定されたインデックスの項目を削除し、関連付けられたメモリをすべて解放してから、リストの残りの項目を順序付けし直します。

戻り値 特にありません。

関連項目

```
CORBA::NVList::add
CORBA::NVList::add_item
CORBA::NVList::add_value
CORBA::NVList::count
CORBA::NVList::item
```

Object メンバ関数

ここでは、OMG IDL インターフェイスの Object に適用される規則について説明します。Object は、OMG IDL インターフェイス階層の基底インターフェイスです。インターフェイス Object では、擬似オブジェクトではなく通常の CORBA オブジェクトを定義します。ただし、ここではほかの擬似オブジェクトも参照するため、これについても触れています。

ほかの規則に加えて、インターフェイス Object のオペレーション名はすべて、マッピングされる C++ クラスの先頭に下線が付きます。また、`create_request` のマッピングは 3 つに分類されます。これらは、「Request メンバ関数」で説明した方式にそれぞれ対応しています。`is_nil` および `release` 関数については、「Object メンバ関数」で説明する CORBA 名前空間を参照してください。

BEA Tuxedo ソフトウェアでは、CORBA Revision 2.2 で定義されているオブジェクト・リファレンス・オペレーションを使用しています。これらのオペレーションが依存するのは、型 `Object` のみです。したがって、オペレーションは CORBA 名前空間の中で通常の関数として記述できます。

注記　BEA Tuxedo ソフトウェアでは、BOA ではなく POA を使用しているため、非推奨の `get_implementation()` メンバ関数は認識されません。このメンバ関数を参照しようとすると、コンパイル・エラーが発生します。

ExceptionList メンバ関数の C++ へのマッピングは次のとおりです。

```
class CORBA
{
  class Object
  {
    public:
       CORBA::Boolean _is_a(const char *)
       CORBA::Boolean _is_equivalent();
       CORBA::Boolean _nonexistent(Object_ptr);

       static Object_ptr _duplicate(Object_ptr obj);
       static Object_ptr _nil();
       InterfaceDef_ptr _get_interface();
       CORBA::ULong _hass(CORBA::ULong);
```

```
        void _create_request(
                   Context_ptr ctx,
                   const char *operation,
                   NVList_ptr arg_list,
                   NamedValue_ptr result,
                   Request_out request,
                   Flags req_flags
        );
        Status _create_request(
                  Context_ptr        ctx,
                  const char *       operation,
                  NVList_ptr         arg_list,
                  NamedValue_ptr     result,
                  ExceptionList_ptr  Except_list,
                  ContextList_ptr    Context_list,
                  Request_out       request,
                  Flags             req_flags
       );
       Request_ptr _request(const char* operation);
  }; //Object
}; // CORBA
```

以下の節では、Object の各メンバ関数について説明します。

CORBA::Object::_create_request

概要 ユーザ指定の情報で要求を作成します。

C++ バインディング

```
Void CORBA::Object::_create_request (
        CORBA::Context_ptr        Ctx,
        const char *              Operation,
        CORBA::NVList_ptr         Arg_list,
        CORBA::NamedValue_ptr     Result,
        CORBA::ExceptionList_ptr  Except_list,
        CORBA::ContextList_ptr    Context_list,
        CORBA::Request_out        Request,
        CORBA::Flags               Req_flags,);
```

引数

`Ctx`
この要求に使用する Context。

`Operation`
この要求のオペレーション名。

`Arg_list`
この要求の引数リスト。

`Result`
正常に呼び出しが行われた後、この要求の戻り値を格納する NamedValue リファレンス。

`Except_list`
この要求の例外リスト。

`Context_list`
この要求のコンテキスト・リスト。

`Request`
新しく作成する要求リファレンス。

`Req_flags`
将来使用するために予約されています。ゼロの値を渡す必要があります。

説明 このメンバ関数は、コンテキスト、オペレーション名、およびその他の情報を提供する要求を作成します (長い形式)。呼び出し時に指定したオペレーション名だけで要求を作成する (短縮形式) には、`CORBA::Object::_request` メンバ関数を使用します。ただし、長い形式で提供される残りの情報については、最終的には指定する必要があります。

戻り値 特にありません。

関連項目 `CORBA::Object::_request`

CORBA::Object::_duplicate

概要 Object オブジェクト・リファレンスを複製します。

C++ バインディング

```
CORBA::Object_ptr CORBA::Object::_duplicate(
                                    Object_ptr Obj);
```

引数

`obj`

複製するオブジェクト・リファレンス。

説明 このメンバ関数は、指定の Object オブジェクト・リファレンス (`Obj`) を複製します。指定のオブジェクト・リファレンスがニルの場合、`_duplicate` 関数はニル・オブジェクト・リファレンスを返します。この呼び出しで返されるオブジェクトは、`CORBA::release` で解放するか、または自動的に破棄されるように `CORBA::Object_var` を割り当てる必要があります。

この関数では、CORBA システム例外がスローされる場合があります。

戻り値 複製オブジェクト・リファレンスを返します。指定のオブジェクト・リファレンスがニルの場合は、ニル・オブジェクト・リファレンスを返します。

例

```
CORBA::Object_ptr op = TP::create_object_reference(
                                "IDL:Teller:1.0","MyTeller");
CORBA::Object_ptr dop = CORBA::Object::_duplicate(op);
```

CORBA::Object::_get_interface

概要 Repository オブジェクトのインターフェイス定義を返します。

C++ バインディング

```
CORBA::InterfaceDef_ptr  CORBA::Object::_get_interface ();
```

引数 特にありません。

説明 Repository オブジェクトのインターフェイス定義を返します。

注記 リポジトリ・インターフェイス API を使用するには、マクロを定義してから CORBA.h をインクルードします。マクロの定義方法については、『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。

戻り値 InterfaceDef_ptr

CORBA::Object::_is_a

概要 オブジェクトが特定のインターフェイスのインスタンスであるかどうかを指定します。

C++ バインディング

```
CORBA::Boolean CORBA::Object::_is_a(const char * interface_id);
```

引数

`interface_id`
インターフェイス・リポジトリ ID を示す文字列。

説明 このメンバ関数は、オブジェクトが `interface_id` パラメータで指定されたインターフェイスのインスタンスであるかどうかを指定するために使用します。この関数を使用すると、ORB のスコープでオブジェクト・リファレンスの型セーフの維持が容易になります。

戻り値 オブジェクトが指定の型のインスタンスである場合、またはオブジェクトがそのオブジェクトの「最終派生」型の上位オブジェクトである場合、`TRUE` を返します。

例

```
CORBA::Object_ptr op = TP::create_object_reference(
                                    "IDL:Teller:1.0", "MyTeller");
CORBA::Boolean b = op->_is_a("IDL:Teller:1.0");
```

CORBA::Object::_is_equivalent

概要 2 つのオブジェクト・リファレンスが等価であるかどうかを判別します。

C++ バインディング

```
CORBA::Boolean CORBA::Object::_is_equivalent (
                            CORBA::Object_ptr other_obj);
```

引数

`other_obj`
対象のオブジェクトとの比較に使用される、他方のオブジェクトのオブジェクト・リファレンス。

例外 標準 CORBA 例外がスローされる場合があります。

説明 このメンバ関数は、2 つのオブジェクト・リファレンスが等価であるかどうかを判別するために使用します。これにより、ORB での判別が容易になります。この関数では、一方のオブジェクト・リファレンスが、パラメータとして渡されたオブジェクト・リファレンスと等価である場合、TRUE を返します。2 つのオブジェクト・リファレンスが同じ場合、両者は等価です。2 つのオブジェクト・リファレンスが異なっても、同一のオブジェクトを参照していれば、両者は等価です。

戻り値 対象のオブジェクト・リファレンスが、パラメータとして渡された他方のオブジェクト・リファレンスと等価であると認識された場合、TRUE を返します。それ以外の場合は、FALSE を返します。

例

```
CORBA::Object_ptr op = TP::create_object_reference(
                                    "IDL:Teller:1.0", "MyTeller");
CORBA::Object_ptr dop = CORBA::Object::_duplicate(op);
CORBA::Boolean b = op->_is_equivalent(dop);
```

CORBA::Object::_nil

概要 ニル・オブジェクトのリファレンスを返します。

C++ バインディング

```
CORBA::Object_ptr CORBA::Object::_nil();
```

引数 特にありません。

説明 このメンバ関数は、ニル・オブジェクト・リファレンスを返します。指定のオブジェクトがニルであるかどうかをテストするには、適切な CORBA::is_nil メンバ関数を使用します。この関数については、「CORBA::release」を参照してください。_nil メンバ関数の CORBA:is_nil ルーチンを呼び出すと、常に CORBA_TRUE が返されます。

戻り値 ニル・オブジェクト・リファレンスを返します。

例

```
CORBA::Object_ptr op = CORBA::Object::_nil();
```

CORBA::Object::_non_existent

概要 オブジェクトが破棄されているかどうかを判別するために使用します。

C++ バインディング

```
CORBA::Boolean CORBA::Object::_non_existent();
```

引数 特にありません。

説明 このメンバ関数は、オブジェクトが破棄されているかどうかを判別するために使用します。この関数では、オブジェクトのアプリケーション・レベルのオペレーションを呼び出さずに判別を行うので、オブジェクト自体に影響は生じません。

戻り値 ORB で正式にオブジェクトが存在しないことが認識された場合、CORBA::OBJECT_NOT_EXIST を引き起こさずに CORBA_TRUE を返します。それ以外の場合は、CORBA_FALSE を返します。

CORBA::Object::_request

概要 オペレーション名を指定する要求を作成します。

C++ バインディング

```
CORBA::Request_ptr CORBA::Object::_request (
     const char  *      Operation);
```

引数

`Operation`
この要求のオペレーション名。

説明 このメンバ関数は、オペレーション名を指定する要求を作成します。引数や結果などのその他の情報はすべて、`CORBA::Request` メンバ関数で設定する必要があります。

戻り値 関数が成功した場合、戻り値は新しく作成された要求へのポインタです。

メンバ関数が失敗した場合、例外がスローされます。

関連項目 `CORBA::Object::_create_request`

CORBA メンバ関数

ここでは、Object および Pseudo-Object Reference メンバ関数について説明します。

ExceptionList メンバ関数の C++ へのマッピングは次のとおりです。

```
class CORBA {
       void release(Object_ptr);
       void release(Environment_ptr);
       void release(NamedValue_ptr);
       void release(NVList_ptr);
       void release(Request_ptr);
       void release(Context_ptr);
       void release(TypeCode_ptr);
       void release(POA_ptr);
       void release(ORB_ptr);
       void release(ExceptionList_ptr);
       void release(ContextList_ptr);

       Boolean is_nil(Object_ptr);
       Boolean is_nil(Environment_ptr);
       Boolean is_nil(NamedValue_ptr);
       Boolean is_nil(NVList_ptr);
       Boolean is_nil(Request_ptr);
       Boolean is_nil(Context_ptr);
       Boolean is_nil(TypeCode_ptr);
       Boolean is_nil(POA_ptr);
       Boolean is_nil(ORB_ptr);
       Boolean is_nil(ExceptionList_ptr);
       Boolean is_nil(ContextList_ptr);

       hash(maximum);

       resolve_initial_references(identifier);
       ...
};
```

CORBA::release

概要 指定のオブジェクト型に対して割り当てられたリソースを解放できるようにします。

C++ バインディング

```
void CORBA::release(spec_object_type obj);
```

引数

`obj`

呼び出し側が今後アクセスしないオブジェクト・リファレンス。指定するオブジェクト型は、「CORBA メンバ関数」の一覧にある型のいずれかでなければなりません。

説明 このメンバ関数は、呼び出し側が今後リファレンスにアクセスしないことを示します。その結果、関連付けられたリソースの割り当てが解除される場合があります。指定したオブジェクト・リファレンスがニルの場合、release オペレーションは何の処理も行いません。ORB の最後のリファレンスである ORB インスタンスを解放する場合、ORB は破棄の前にシャットダウンされます。これは、`CORBA::release` を呼び出す前に `ORB_shutdown` を呼び出すのと同じです。この処理は、ORB で呼び出された `release` メンバ関数にのみ適用されます。

このメンバ関数では、CORBA 例外はスローされない場合があります。

戻り値 特にありません。

例

```
CORBA::Object_ptr op = TP::create_object_reference(
                       "IDL:Teller:1.0", "MyTeller");
CORBA::release(op);
```

CORBA::is_nil

概要 指定のオブジェクト型に対応するオブジェクトが存在するかどうかを判別します。

C++ バインディング

```
CORBA::Boolean CORBA::is_nil(spec_object_type obj);
```

引数

`obj`

オブジェクト・リファレンス。指定するオブジェクト型は、「CORBA メンバ関数」の一覧にある型のいずれかでなければなりません。

説明 このメンバ関数は、指定のオブジェクト・リファレンスがニルかどうかを判別するために使用します。ORB で定義したニル・オブジェクト・リファレンスの特別な値がオブジェクト・リファレンスにある場合、TRUE が返されます。

このオペレーションでは、CORBA 例外はスローされない場合があります。

戻り値 指定のオブジェクトがニルの場合は `TRUE` を、それ以外の場合は `FALSE` を返します。

例

```
CORBA::Object_ptr op = TP::create_object_reference(
                                  "IDL:Teller:1.0", "MyTeller");
CORBA::Boolean b = CORBA::is_nil(op);
```

CORBA::hash

概要 ORB 内部の識別子でオブジェクト・リファレンスへの間接アクセスを提供します。

C++ バインディング

```
CORBA::hash(CORBA::ULong maximum);
```

引数

`maximum`
ORB が返すハッシュ値の上限を指定します。

説明 オブジェクト・リファレンスは、ORB 内部識別子に関連付けられています。アプリケーションは、hash() オペレーションを使用して、この識別子に間接的にアクセスできます。この識別子の値は、オブジェクト・リファレンスが存在している間は変更されません。したがって、識別子のどのハッシュ関数も変更されません。

このオペレーションの値は、確実に一意とは限りません。つまり、ほかのオブジェクト・リファレンスが同じハッシュ値を返す場合もあるということです。ただし、2 つのオブジェクト・リファレンスが別々にハッシュされた場合、アプリケーションでは 2 つのオブジェクト・リファレンスが同じでないことを判別できます。

hash オペレーションの maximum パラメータには、ORB が返すハッシュ値の上限を指定します。この値の下限はゼロです。この機能は通常、オブジェクト・リファレンスの衝突チェーン状のハッシュ・テーブルを作成したり、これにアクセスしたりするために使用します。したがって、その範囲内で値がランダムに分散するほど、計算する値は小さくなり、計算精度も向上します。

戻り値 特にありません。

CORBA::resolve_initial_references

概要 identifier 文字列に対応する初期オブジェクト・リファレンスを返します。

C++ バインディング

```
CORBA::Object_ptr CORBA::resolve_initial_references(
         const CORBA::char *identifier);
```

引数

identifier
リファレンスが必要なオブジェクトを識別する文字列。

例外 InvalidName

説明 identifier 文字列に対応する初期オブジェクト・リファレンスを返します。有効な識別子は、"RootPOA" および "POACurrent" です。

注記　この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値 CORBA::Object_ptr を返します。

例

```
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
CORBA::Object_ptr pfobj =
        orb->resolve_initial_references("RootPOA");
PortableServer::POA_ptr rootPOA;
rootPOA = PortableServer::POA::narrow(pfobj);
```

ORB メンバ関数

ORB メンバ関数は、オブジェクト・リクエスト・ブローカのプログラミング・インターフェイスです。

ORB メンバ関数の C++ へのマッピングは次のとおりです。

```
class CORBA
{
  class ORB
  {
    public:
      char *object_to_string(Object_ptr);
      Object_ptr string_to_object(const char *);
      void create_list(Long, NVList_out);
      void create_operation_list(operationDef_ptr, NVList_out);
      void create_named_value(NamedValue_out);
      void create_exception_list(ExceptionList_out);
      void create_context_list(ContextList_out);
      void get_default_context(Context_out);
      void create_environment(Environment_out);
      void send_multiple_requests_oneway(const requestSeq&);
      void send_multiple_requests_deferred(const requestSeq&);
      Boolean poll_next_response();
      void get_next_response(Request_out);
      Boolean work_pending();
      void perform_work();
      void create_policy (in PolicyType type, in any val);
      // 拡張
      void destroy();
      // スレッド間のコンテキストの共有をサポートする拡張
      void Ctx get_ctx() = 0;
      void set_ctx(Ctx) = 0;
      void clear_ctx() = 0;
      // スレッドの拡張
      void inform_thread_exit(TID) = 0;
 }; //ORB
}; // CORBA
```

スレッド関連のオペレーション

シングル・スレッド ORB、および認識されないマルチスレッド・コードを実行するマルチスレッド ORB をサポートするには、perform_work と work_pending の 2 つのオペレーションを ORB インターフェイスに含めます。これらのオペレーションは、シングル・スレッド・アプリケーションでもマルチスレッド・アプリケーションでも使用できます。アプリケーションが純粋な ORB クライアントの場合は、これらのオペレーションは不要です。

マルチスレッド・サーバ・アプリケーションをサポートするには、get_ctx、set_ctx、clear_ctx、および inform_thread_exit の 4 つのオペレーションを ORB インターフェイスへの拡張として含めます。

以下の節では、ORB の各メンバ関数について説明します。

CORBA::ORB::clear_ctx

概要 このメソッドでコンテキストが不要になったことを示します。このメソッドは、マルチスレッド・サーバ・アプリケーションの開発をサポートします。

C++ バインディング

```
void clear_ctx()
```

パラメータ 特にありません。

戻り値 特にありません。

説明 このメソッドは、スレッドがコンテキストの使用を終了した後、アプリケーション管理のスレッドによって呼び出されます。このメソッドは、当該スレッドとコンテキストとの関連付けを解除します。

注記 `clear_ctx` メソッドは、BEA Tuxedo システムが管理しているスレッド内からは呼び出さないでください。BEA Tuxedo システムでは、適切にコンテキストを伝達し、管理しているスレッドを自動的にクリーンアップします。BEA Tuxedo システムが管理しているスレッドでこのメソッドを呼び出した場合、`BAD_PARAM` 例外がスローされます。

例

```
TP::orb()->clear_ctx();
```

関連項目

```
CORBA::ORB::get_ctx
CORBA::ORB::set_ctx
```

CORBA::ORB::create_context_list

概要 コンテキストのリストを作成して返します。

C++ バインディング

```
void CORBA::ORB::create_context_list(
        CORBA::ContextList_out  List);
```

引数

`List`

新しく作成されるコンテキスト・リストのリファレンスを受け取ります。

説明 このメンバ関数は、コンテキスト文字列のリストを作成して返します。コンテキスト文字列のリストには、動的起動インターフェイス (DII) で使用可能な形式で Request オペレーションを指定する必要があります。このリストが不要になったときは、`CORBA::release` メンバ関数でリストを解放しなければなりません。

戻り値 特にありません。

CORBA::ORB::create_environment

概要 環境を作成します。

C++ バインディング

```
void CORBA::ORB::create_environment (
            CORBA::Environment_out   New_env);
```

引数

`New_env`
新しく作成される環境のリファレンスを受け取ります。

説明 このメンバ関数は、環境を作成します。

戻り値 特にありません。

関連項目

`CORBA::NVList::add`
`CORBA::NVList::add_item`
`CORBA::NVList::add_value`
`CORBA::release`

CORBA::ORB::create_exception_list

概要 例外のリストを返します。

C++ バインディング

```
void CORBA::ORB::create_exception_list(
        CORBA::ExceptionList_out  List);
```

引数

`List`
新しく作成される例外リストのリファレンスを受け取ります。

説明 このメンバ関数は、動的起動インターフェイス (DII) で使用可能な形式で例外のリストを作成して返します。このリストが不要になったときは、`CORBA::release` メンバ関数でリストを解放しなければなりません。

戻り値 特にありません。

CORBA::ORB::create_list

概要 NVList オブジェクト・リファレンスを作成して返します。

C++ バインディング

```
void CORBA::ORB::create_list (
     CORBA::Long               NumItem,
     CORBA::NVList_out          List);
```

引数

`NumItem`
新しく作成されるリストに事前に割り当てる要素数。

`List`
新しく作成されたリストを受け取ります。

説明 このメンバ関数は、指定の項目数を事前に割り当ててリストを作成します。リスト項目は、`CORBA::NVList_add_item` メンバ関数を使用して順番にリストに追加できます。このリストが不要になったときは、`CORBA::release` メンバ関数でリストを解放しなければなりません。

戻り値 特にありません。

関連項目

`CORBA::NVList::add`
`CORBA::NVList::add_item`
`CORBA::NVList::add_value`
`CORBA::release`

CORBA::ORB::create_named_value

概要 NamedValue オブジェクト・リファレンスを作成します。

C++ バインディング

```
void CORBA::ORB::create_named_value (
     NameValue_out         NewNamedVal);
```

引数

`NewNamedVal`

新しく作成される NamedValue オブジェクトのリファレンス。

説明 このメンバ関数は、NamedValue オブジェクトを作成します。これは、NamedValue オブジェクトが必要な要求の結果の引数を取得するために使用します。このメンバ関数を呼び出すと、NVList オブジェクトを作成するのに余計な手順を省くことができます。

NamedValue オブジェクトが不要になったときは、`CORBA::release` メンバ関数で NamedValue オブジェクトを解放しなければなりません。

戻り値 特にありません。

関連項目

`CORBA::NVList::add`
`CORBA::NVList::add_item`
`CORBA::NVList::add_value`
`CORBA::release`

CORBA::ORB::create_operation_list

概要 指定されたオペレーションの引数のリストを作成して返します。

C++ バインディング

```
void CORBA::ORB::create_operation_list (
     CORBA::OperationDef_ptr     Oper,
     CORBA::NVList_out            List);
```

引数

`Oper`

作成中のリストのオペレーション定義。

`List`

新しく作成される引数リストのリファレンスを受け取ります。

説明 このメンバ関数は、動的起動インターフェイス (DII) で使用可能な形式で、指定されたオペレーションの引数のリストを作成して返します。このリストが不要になったときは、`CORBA::release` メンバ関数でリストを解放しなければなりません。

戻り値 特にありません。

関連項目

`CORBA::OBB::create_list`
`CORBA::NVList::add`
`CORBA::NVList::add_item`
`CORBA::NVList::add_value`
`CORBA::release`

CORBA::ORB::create_policy

概要 指定された初期状態で特定の型の方針オブジェクトの新しいインスタンスを作成します。

C++ バインディング

```
void CORBA::ORB::create_policy (
                        in PolicyType type,
                         in any val);
```

引数

`type`

BEA WebLogic Enterprise バージョン 4.2 でサポートされている `PolicyType` 値は、`BiDirPolicy::BIDIRECTIONAL_POLICY_TYPE` のみです。

`val`

BEA WebLogic Enterprise バージョン 4.2 でサポートされている `val` 値は、`BiDirPolicy::BidirectionalPolicyValue` のみです。

例外 `PolicyError`

この例外は、`ORB::create_policy` オペレーションに渡されたパラメータ値に問題があることを示すために発生します。個別の例外と理由を次に示します。

例外	理由
`BAD_POLICY`	要求された方針は ORB で認識されません。
`UNSUPPORTED_POLICY`	要求された方針は ORB で有効と認識されますが、現在サポートされていません。
`BAD_POLICY_TYPE`	方針に要求された値の型は、その PolicyType で無効です。
`BAD_POLICY_VALUE`	方針に要求された値は有効な型ですが、その型の有効な範囲に入っていません。
`UNSUPPORTED_POLICY_VALUE`	方針に要求された値は有効な型で、その型の有効な範囲に入っていますが、現在サポートされていません。

説明 このオペレーションは、指定された初期状態で特定の型の方針オブジェクトの新しいインスタンスを作成するために呼び出します。方針の要求された型と内容を解釈できなかったために、create_policy が新しい方針オブジェクトのインスタンス化に失敗した場合、適切な理由により方針エラー例外が発生します。

リモート・クライアントでは IIOP を使用しているため、コールバックを使用して BidirectionalPolicy 引数をリモート・クライアントに指定します。この引数は、コールバックを使用するネイティブ・クライアント、または BEA Tuxedo サーバには使用しません。これは、BEA Tuxedo ドメイン内部のマシンが個別に通信するためです。

GIOP 1.2 より前では、双方向方針は TCP/IP を使用する IIOP では利用できませんでした。GIOP 1.0 および 1.1 の通信は一方向のため、要求がクライアントからサーバに送信され、応答がサーバからクライアントに送信されるだけでした。サーバが要求をクライアント・マシンに戻す、つまりコールバックする場合、サーバ・マシンでは別の一方向の接続を確立する必要がありました (ここでいう「接続」とは、オペレーティング・システムのリソースのことであり、物理的な個別のワイヤや接続パスのことではありません。接続ではリソースを使用するので、接続数を最小限にすることが求められます)。

最新の BEA Tuxedo C++ ソフトウェアでは、GIOP 1.2 をサポートしています。これにより、要求の送信と受信の両方で TCP/IP 接続を再利用できるようになりました。接続を再利用することにより、リモート・クライアントが BEA Tuxedo ドメインにコールバック・リファレンスを送信するときにリソースを節約できます。共同クライアント / サーバでは、接続で BEA Tuxedo ドメインに要求を送信します。その際、コールバックの要求に接続を再利用することができます。接続を再利用しない場合、コールバック要求では別の接続を確立しなければなりません。

接続の再利用を可能にするために、ORB/POA が用意されています。ORB/POA は、コールバック・オブジェクト・リファレンスを作成します。このオブジェクト・リファレンスのサーバ (特にコールバックの場合は、リファレンスの作成元) では、セキュリティを考慮して再利用を無効にすることができます。つまり、このマシンのコールバック・サーバではセキュリティを必要としますが、リモート・サーバではセキュリティを必要としないため、このマシンからリモート・サーバにクライアントの要求を送信接続する際はセキュリティは不要になります。このように、接続ベースで部分的にセキュリティが確立されるので、別の接続を使用する場合に限って受信時の

セキュリティを確立することができます。リモート・サーバでセキュリティが必要な場合、またはそのセキュリティに相互認証がある場合、通常はローカル・サーバで接続の再利用を有効にしても安全と認識されます。

プロセスがサーバ (この場合、共同クライアント / サーバ) として機能してオブジェクト・リファレンスを作成する場合は、常に接続の再利用はサーバ・エンドで行われます。そのため、接続を再利用することを ORB に通知する必要があります。これは、オブジェクト・リファレンスを作成する POA の方針を設定することによって行います。デフォルトの方針では、再利用は無効になっています。したがって、再利用するための方針オブジェクトを指定しない限り、POA では再利用ができません。

このデフォルトでは、CORBA バージョン 2.3 より前に記述されたコードとの下位互換性が維持されています。このようなコードでは、再利用が可能であることは認識されないので、再利用に関するセキュリティのインプリメンテーションを考慮する必要はありません。ただし、変更が加えられていないコードでは、ユーザがセキュリティを考慮して明示的に方針を転換しない限り、引き続き再利用は無効のままになります。

再利用を有効にするには、`create_policy` オペレーションを使用して再利用を可能にする方針オブジェクトを作成し、その方針オブジェクトを POA 作成用の方針リストの一部として使用します。

戻り値 特にありません。

例

```
#include <BiDirPolicy_c.h>
BiDirPolicy::BidirectionalPolicy_var bd_policy;
CORBA::Any allow_reuse;

allow_reuse <<= BiDirPolicy::BOTH;

CORBA::Policy_var generic_policy =
      orb->create_policy( BiDirPolicy::BIDIRECTIONAL_POLICY_TYPE,
                          allow_reuse );
bd_policy = BiDirPolicy::BidirectionalPolicy::_narrow(
                                                    generic_policy );
```

上記の例では、`create_poa` オペレーションに渡された PolicyList の中に `bd_policy` を挿入しています。

CORBA::ORB::destroy

概要 指定された ORB を破棄します。

C++ バインディング

```
void destroy();
```

パラメータ 特にありません。

戻り値 特にありません。

説明 このメソッドを使用すると、ORB に関連付けられたリソースを再利用できるよう ORB を破棄できます。ORB を破棄した場合、同じ ORB ID で `ORB_init` を別に呼び出すと、新しく作成された ORB のリファレンスが返されます。現在呼び出しているスレッドからアプリケーションで `ORB::destroy` メソッドを呼び出した場合、BEA Tuxedo システムでは、OMG マイナー・コード 3 の `BAD_INV_ORDER` システム例外が発生します。これは、ブロッキングによってデッドロックが発生するためです。

例

```
pOrb->destroy();
```

CORBA::ORB::get_ctx

概要 現在のスレッドと関連付けられているコンテキストを取得します。このメソッドは、マルチスレッド・サーバ・アプリケーションの開発をサポートします。

C++ バインディング

```
CORBA::ORB::Ctx get_ctx()
```

引数 特にありません。

戻り値

```
CORBA::ORB::Ctx
```

現在のスレッドに関連付けられているコンテキスト。

説明 このメソッドを使用すると、現在のスレッドに関連付けられているコンテキストを取得できます。このコンテキストは、アプリケーションによって作成および管理されるほかのスレッドを初期化するのに使用できます。

オブジェクトがスレッドを作成すると、ORB のこのオペレーションを呼び出して、スレッドに渡すことができるシステム・コンテキスト情報を取得します。このオペレーションは、既にコンテキストを持っているスレッドから呼び出される必要があります。たとえば、メソッドがディスパッチされたスレッドには、既にコンテキストがあります。

例

```
thread.context = TP::orb()->get_ctx();
```

関連項目

```
CORBA::ORB::set_ctx
CORBA::ORB::clear_ctx
```

CORBA::ORB::get_default_context

概要 デフォルト・コンテキストへのリファレンスを返します。

C++ バインディング

```
void CORBA::ORB::get_default_context (
     CORBA::Context_out       ContextObj);
```

引数

`ContextObj`
デフォルト・コンテキストへのリファレンス。

説明 このメンバ関数は、デフォルト・コンテキストへのリファレンスを返します。このコンテキスト・リファレンスが不要になったときは、`CORBA::release` メンバ関数でコンテキスト・リファレンスを解放しなければなりません。

戻り値 特にありません。

関連項目

`CORBA::Context::get_one_value`
`CORBA::Context::get_values`

CORBA::ORB::get_next_response

概要 次に完了する遅延同期要求を判別および報告します。

C++ バインディング

```
void CORBA::ORB::get_next_response (
     CORBA::Request_out       RequestObj);
```

引数

`RequestObj`
次に完了する要求へのリファレンス。

説明 このメンバ関数は、次に完了する要求へのリファレンスを返します。完了した要求がない場合、関数は要求が完了するまで待機します。このメンバ関数は、キューの次の要求を返します。これと対照的に `CORBA::Request::get_response` メンバ関数は、特定の要求が完了するまで待機します。この要求が不要になったときは、`CORBA::release` メンバ関数で要求を解放しなければなりません。

戻り値 特にありません。

関連項目

`CORBA::ORB::poll_next_response`
`CORBA::Request::get_reponse`

CORBA::ORB::inform_thread_exit

概要 アプリケーション管理のスレッドに関連付けられたリソースが解放可能であることを、BEA Tuxedo システムに通知します。このメソッドは、マルチスレッド・サーバ・アプリケーションの開発をサポートします。

C++ バインディング

```
void CORBA::ORB::inform_thread_exit(CORBA::TID threadId)
```

パラメータ

threadId
削除中のアプリケーション管理スレッドの論理スレッド ID。

戻り値 特にありません。

説明 このメソッドは、次の状態を BEA Tuxedo システムに通知します。

- 指定されたアプリケーション管理スレッドがサーバント・インプリメンテーションで今後使用されない点。
- そのスレッドに関連付けられたすべてのリソースが解放される点。

注記 このオペレーションは、アプリケーションが作成および管理するスレッドに対してのみ呼び出します。BEA Tuxedo システムによって管理されるディスパッチ・スレッドを指定するときは、このメソッドを呼び出さないでください。

例

```
pOrb->inform_thread_exit(thread.threadId);
```

CORBA::ORB::list_initial_services

概要 初期リファレンス・メカニズムで利用可能なリファレンスがどのオブジェクトにあるかを判別します。

C++ バインディング

```
typedef string ObjectId;
typedef sequence ObjectId ObjectIdList;
ObjectIdList list_initial_services ();
```

引数

ObjectId
オブジェクト ID。

list_initial_services ()
オブジェクト型を定義します。

説明 このオペレーションは、初期リファレンス・メカニズムで利用可能なリファレンスがどのオブジェクトにあるかを判別するために、アプリケーションで使用されます。このオペレーションは、ObjectId のシーケンスである ObjectIdList を返します。ObjectId には文字列型が付きます。

初期化時に利用可能にする必要のある各オブジェクトには、それを表す文字列値が割り当てられます。また、ID の定義に加えて、返されているオブジェクトの型が必ず定義されます。たとえば、InterfaceRepository は 型 Repository のオブジェクトを返し、NameService は CosNamingContext オブジェクトを返します。

戻り値 ObjectId のシーケンス。

関連項目 CORBA::ORB::resolve_initial_references

CORBA::ORB::object_to_string

概要 オブジェクト・リファレンスの文字列表現を生成します。

C++ バインディング

```
char * CORBA::ORB::object_to_string (
     CORBA::Object_ptr   ObjRef);
```

引数

`ObjRef`
文字列として表すオブジェクト・リファレンス。

説明 このメンバ関数は、オブジェクト・リファレンスの文字列表現を生成します。呼び出し元のプログラムでは、文字列のメモリが不要になったら、`CORBA::string_free` メンバ関数で解放する必要があります。

戻り値 指定されたオブジェクト・リファレンスを表す文字列。

例

```
CORBA::Object_ptr op = TP::create_object_reference(
                                    "IDL:Teller:1.0", "MyTeller");
char* objstr = TP::orb()->object_to_string(op);
```

関連項目

`CORBA::ORB::string_to_object`
`CORBA::string_free`

CORBA::ORB::perform_work

概要 ORB がサーバ関連の作業を実行できるようにします。

C++ バインディング

```
void CORBA::ORB::perform_work ();
```

引数 特にありません。

例外 ORB がシャットダウンした後に、work_pending および perform_work() を呼び出すと、BAD_INV_ORDER 例外が発生します。アプリケーションでは、この例外を検出して、いつポーリング・ループを終了するかを判定します。

説明 このオペレーションは、メイン・スレッドによって呼び出された場合に ORB がサーバ関連の作業を実行できるようにします。それ以外の場合は、何の処理も行いません。

work_pending() および perform_work() オペレーションを使用すると、ORB などのアクティビティのメイン・スレッドを多重化する単純なポーリング・ループを記述できます。このようなループは主に、シングル・スレッドのサーバで必要になります。マルチスレッドのサーバでポーリング・ループが必要になるのは、メイン・スレッドの使用が必要なほかのコードと ORB の両方がある場合のみです。このようなポーリング・ループについては、以下の例を参照してください。

戻り値 特にありません。

関連項目 CORBA::ORB::work_pending

例 次に、ポーリング・ループの例を示します。

```
// C++
for (;;) {
    if (orb->work_pending()) {
        orb->perform_work();
    }
    // ほかの作業を実行させるか、
    // またはスリープ状態にする
}
```

CORBA::ORB::poll_next_response

概要 完了した要求が未処理かどうかを判別します。

C++ バインディング

```
CORBA::Boolean  CORBA::ORB::poll_next_response ();
```

引数 特にありません。

説明 このメンバ関数は、完了した要求が未処理 (保留中) かどうかを報告します。これは、報告をするだけで要求を削除するわけではありません。完了した要求が未処理の場合、次に CORBA::ORB::get_next_response メンバ関数を呼び出すと、直ちに要求が確実に返されます。完了した要求に未処理のものがない場合、直ちに CORBA::ORB::poll_next_response メンバ関数が返されます (ブロッキング)。

戻り値 完了した要求が未処理の場合、関数は CORBA_TRUE を返します。

完了した要求に未処理のものがない場合、関数は CORBA_FALSE を返します。

関連項目 CORBA::ORB::get_next_response

CORBA::ORB::resolve_initial_references

概要 初期サービスのオブジェクト・リファレンスを取得します。

C++ バインディング

```
Object resolve_initial_references ( in ObjectId identifier )
    raises (InvalidName);

exception InvalidName {};
```

引数

`identifier`
リファレンスが必要なオブジェクトを識別する文字列。

説明 このオペレーションは、初期サービスのオブジェクト・リファレンスを取得するためにアプリケーションで使用されます。インターフェイスはネーミング・サービスの解決方法によって異なります。つまり、`ObjectId` (文字列) をより複雑なネーミング・サービスの構成 (名前の構成要素の文字列のペアを含んだ構造体のシーケンス) に置き換えるかどうかで異なります。この簡素化の方法を実行すると、名前空間は 1 つのコンテキストに減ります。

`ObjectId` は、リファレンスが必要なオブジェクトを識別する文字列です。初期リファレンスを取得するインターフェイスの簡素さを維持するには、限定したオブジェクトのセットのみに、上記の方法で見つかったリファレンスを格納します。ORB 識別子とは異なり、`ObjectId` 名前空間には慎重な管理が必要になります。そのために ORB では、後にこのインターフェイスを介してアプリケーションで必要になるサービスを定義したり、そのサービスの名前を指定したりできます。

現在、予約されている `ObjectId` は、`RootPOA`、`POACurrent`、`InterfaceRepository`、`NameService`、`TradingService`、`SecurityCurrent`、`TransactionCurrent`、および `DynAnyFactory` です。

アプリケーションでは、`resolve_initial_references` から返されるオブジェクト・リファレンスを `ObjectId` で要求された型に限定します。たとえば、`InterfaceRepository` の場合、返されるオブジェクトは `Repository` 型に限定されます。

戻り値 初期サービスのオブジェクト・リファレンス。

関連項目 `CORBA::ORB::list_initial_services`

CORBA::ORB::send_multiple_requests_deferred

概要 遅延同期要求のシーケンスを送信します。

C++ バインディング

```
void CORBA::ORB::send_multiple_requests_deferred (
     const CORBA::ORB::RequestSeq &    Reqs);
```

引数

Reqs

送信する要求のシーケンス。シーケンスへの要求リファレンスの挿入に関する詳細については、「形式」の CORBA::ORB::RequestSeq を参照してください。

説明 このメンバ関数は、オペレーションが完了するのを待機せずに要求のシーケンスを送信し、呼び出し側に制御を返します。呼び出し側は、CORBA::ORB::poll_ next_response、CORBA::ORB::get_next_response、または CORBA::Rquest::get_response、あるいはこれら 3 つをすべて使用して、オペレーションが完了したかどうか、および出力引数が更新されたかどうかを判別できます。

戻り値 特にありません。

関連項目

CORBA::Request::get_response
CORBA::ORB::get_next_response
CORBA::ORB::send_multiple_requests_oneway

CORBA::ORB::send_multiple_requests_oneway

概要 一方向の遅延同期要求のシーケンスを送信します。

C++ バインディング

```
void CORBA::ORB::send_multiple_requests_oneway (
     const CORBA::RequestSeq &      Reqs);
```

引数

`Reqs`

送信する要求のシーケンス。シーケンスへの要求リファレンスの挿入に関する詳細については、「形式」の `CORBA::ORB::RequestSeq` を参照してください。

説明 このメンバ関数は、オペレーションが完了するのを待機せずに要求のシーケンスを送信し、呼び出し側に制御を返します。呼び出し側は、応答を待機せず、更新する出力引数も指定しません。

戻り値 特にありません。

関連項目 `CORBA::ORB::send_multiple_requests_deferred`

CORBA::ORB::set_ctx

概要 現在のスレッドのコンテキストを設定します。このメソッドは、マルチスレッド・サーバ・アプリケーションの開発をサポートします。

C++ バインディング

```
void set_ctx(CORBA::ORB::Ctx aContext)
```

パラメータ

aContext
現在のスレッドに関連付けるコンテキスト。

戻り値 特にありません。

説明 このメソッドは、現在のアプリケーション管理スレッドのコンテキストを設定します。指定するコンテキスト・パラメータは、BEA Tuxedo システムによって管理されている実行済みのスレッド、または初期化済みのアプリケーション管理スレッドで既に取得されています。

注記 set_ctx メソッドは、BEA Tuxedo システムが管理しているスレッドでは呼び出さないでください。BEA Tuxedo システムでは、管理しているスレッドに対して適切にコンテキストを自動的に伝達します。BEA Tuxedo システムが管理しているスレッドで、このメソッドがアプリケーションによって呼び出された場合、BAD_PARAM 例外がスローされます。

例

```
TP::orb()->set_ctx(thread->context);
```

関連項目

```
CORBA::ORB::get_ctx()
CORBA::ORB::clear_ctx()
```

CORBA::ORB::string_to_object

概要 CORBA::ORB::object_to_string オペレーションによって生成された文字列を変換し、対応するオブジェクト・リファレンスを返します。

C++ バインディング

```
Object string_to_object ( in string str );
```

引数

str

CORBA::ORB::object_to_string オペレーションによって生成された文字列。

説明 このオペレーションは、CORBA::ORB::object_to_string オペレーションによって生成された文字列を変換するためにアプリケーションで使用され、対応するオブジェクト・リファレンスを返します。

確実に ORB でオブジェクト・リファレンスの文字列形式を認識できるようにするには、文字列を生成する際にその ORB の object_to_string オペレーションを使用する必要があります。string_to_object オペレーションでは、IOR の URL、corbaloc、corbalocs、および corbanames の各形式をオブジェクト・リファレンスに変換できます。変換に失敗した場合、string_to_object オペレーションによって、次のいずれかのマイナー・コードの BAD_PARAM 標準例外が発生します。

- BadSchemeName
- BadAddress
- BadSchemeSpecificPart

ORB に完全準拠する場合、obj がオブジェクトの有効なリファレンスであり、2 つのオペレーションが同じ ORB で実行されると、string_to_object(object_to_string(obj)) は同じオブジェクトの有効なリファレンスを返します。ORB がサポートする IOP に完全準拠する場合は、2 つのオペレーションが異なる ORB で実行されても設定は同じままになります。

戻り値 オブジェクト・リファレンスを返します。

関連項目 CORBA::ORB::object_to_string

CORBA::ORB::work_pending

概要 サーバ関連の作業を実行するために ORB でメイン・スレッドが必要かどうかを示す値を返します。

C++ バインディング

```
CORBA::boolean  CORBA::ORB::work_pending ();
```

引数 特にありません。

説明 このオペレーションは、サーバ関連の作業を実行するために ORB でメイン・スレッドが必要かどうかを示す値を返します。

戻り値 結果が TRUE の場合は、サーバ関連の作業を実行するために ORB でメイン・スレッドが必要であることを示します。結果が FALSE の場合は、ORB でメイン・スレッドが不要であることを示します。

関連項目 CORBA::ORB::perform_work

ORB 初期化メンバ関数

ORB 初期化メンバ関数の C++ へのマッピングは次のとおりです。

```
class CORBA {
       static CORBA::ORB_ptr ORB_init(int& argc, char** argv,
                                      const char* orb_identifier = 0,
                                      const char* -ORBport nnn);
       <appl-name> [-ORBid {BEA_IIOP | BEA_TOBJ} \
            [-ORBInitRef <ObjectID>=<ObjectURL> [*]]
            [-ORBDefaultInitRef <ObjectURL>]
            [-ORBport port-number] \
            [-ORBsecurePort port-number] \
            [-ORBminCrypto {0 | 40 | 56 | 128}] \
            [-ORBmaxCrypto {0 | 40 | 56 | 128}] \
            [-ORBmutualAuth] \
            [-ORBpeerValidate {detect | warn | none}] \
            [appl-options]

};
```

CORBA::ORB_init

概要 ORB のオペレーションを初期化します。

C++ バインディング

```
static CORBA::ORB_ptr ORB_init(int& argc, char** argv,
                               const char* orb_identifier = 0);
```

引数

`argc`
: `argv` の文字列数。

`argv`
: この引数は、文字列のアンバウンディッド配列 (`char **`) として定義します。配列内の文字列数は、`argc` パラメータで渡されます。

`orb_identifier`
: `orb_identifier` パラメータを指定すると、リモート・クライアントが `"BEA_IIOP"` で明示的に指定され、「Tobj_Bootstrap」で定義されるネイティブ・クライアントが `"BEA_TOBJ"` で明示的に指定されます。

説明 このメンバ関数は、ORB のオペレーションを初期化して、ORB へのポインタを返します。ORB でプログラミングする場合は、`CORBA::release` メンバ関数を使用して、`CORBA::ORB_ptr ORB_init` から返された ORB ポインタに割り当てられたリソースを解放します。

返された ORB は、クライアントのタイプ (リモートまたはネイティブ) の処理方法とサーバのポート番号の処理方法を指定する 2 つの情報で初期化されています。クライアントのタイプは、`orb_identifier` 引数、argv 引数、またはシステム・レジストリで指定できます。サーバのポート番号は、`argv` 引数で指定できます。

通常、引数 `argc` と `argv` は、メイン・プログラムに渡されたパラメータと同じです。C++ での指定のように、これらのパラメータには、クライアントを起動したコマンド行の文字列トークンが格納されます。2 つの ORB オプションは、以下の例で示すように、トークンのペアをそれぞれ使用してコマンド行で指定できます。

クライアントのタイプ

`ORB_init` 関数は、次の手順で ORB のクライアントのタイプを判別します。

1. orb_identifier 引数が指定されている場合、ORB_init は、文字列が "BEA_IIOP" か "BEA_TOBJ" かによって、クライアントのタイプがネイティブかリモートかを判別します。orb_identifier 文字列がある場合、argv の -ORBid パラメータはすべて無視または削除されます。

2. orb_identifier が指定されていないか、または明示的にゼロに指定されている場合、ORB_init は argc/argv のエントリを確認します。argv に "-ORBid" のエントリがある場合、次のエントリは、リモートまたはネイティブを表す "BEA_IIOP" か "BEA_TOBJ" かのどちらかになります。このエントリのペアが出現するのは、コマンド行に "-ORBid BEA_IIOP" または "-ORBid BEA_TOBJ" のどちらかがある場合です。

3. クライアントのタイプが argc/argv で指定されていない場合、ORB_init は、システム・レジストリ (BEA_IIOP または BEA_TOBJ) のデフォルトのクライアントのタイプを使用します。システム・レジストリは、BEA Tuxedo のインストール時に初期化されています。

サーバのポート

BEA Tuxedo リモート共同クライアント / サーバの場合、IIOP をサポートするために、サーバに対して作成されたオブジェクト・リファレンスにホストとポートを定義する必要があります。一時オブジェクト・リファレンスの場合、ORB で動的に任意のポートを取得できます。しかし、これは永続オブジェクト・リファレンスには適用できません。永続的なリファレンスの場合、ORB の再起動後に同じポートを指定する必要があります。つまり、ORB では、オブジェクト・リファレンスが作成されたのと同じポートで要求を受け付けなければなりません。このように、特定のポートを使用するように ORB をコンフィギュレーションする方法は複数あります。

通常、システム管理者は、動的範囲ではなくポート番号のユーザ範囲からクライアントのポート番号を割り当てます。これにより、共同クライアント / サーバでポートの競合を防ぐことができます。

ポート番号を指定するために ORB_init は、argv パラメータの "-ORBport" トークンとそれに続く数値のトークンを検索します。たとえば、クライアントの実行可能ファイルの名前が sherry の場合、コマンド行では次のようにサーバのポート 937 に指定します。

```
sherry -ORBport 937
```

ARGV パラメータの考慮事項

C++ の場合、argv パラメータの処理順序がアプリケーションでは重要になります。アプリケーションで認識されない argv パラメータは、確実にアプリケーションで処理不要にし、ORB 初期化関数を呼び出してから、残りのパラメータを処理するようにしなければなりません。したがって、ORB_init を呼び出した後は、argv および argc パラメータは、ORB で認識された引数を削除するために変更されています。この際に重要なのは、ORB_init 関数が、argv リストのパラメータのリファレンスの順序変更または削除しかできない点です。この制限が設けられたのは、一部の argv リストの解放を試行したり、パラメータの argv リストの拡張を試行したりすることで発生する可能性のあるメモリ管理上の問題を防ぐためです。このような理由から、argv は、char**& ではなく char** として渡されます。

注記　CORBA::ORB_init から返されたポインタに割り当てられたリソースを解放するには、CORBA::release メンバ関数を使用します。

戻り値　CORBA::ORB へのポインタ。

例外　特にありません。

ORB

概要 BEA Tuxedo の CORBA オブジェクトにアクセスしたり、CORBA オブジェクトを提供するために、BEA Tuxedo CORBA C++ ORB に基づいてアプリケーションをコンフィギュレーションします。

構文

```
<appl-name> [-ORBid {BEA_IIOP | BEA_TOBJ} \
            [-ORBInitRef <ObjectID>=<ObjectURL> [*]]
            [-ORBDefaultInitRef <ObjectURL>]
            [-ORBport port-number] \
            [-ORBsecurePort port-number] \
            [-ORBminCrypto {0 | 40 | 56 | 128}] \
            [-ORBmaxCrypto {0 | 40 | 56 | 128}] \
            [-ORBmutualAuth] \
            [-ORBpeerValidate {detect | warn | none}] \
            [appl-options]
```

説明 BEA Tuxedo CORBA C++ ORB は、BEA Tuxedo に組み込まれているライブラリです。このライブラリを使用すると、IIOP または IIOP-SSL を使用して BEA Tuxedo のオブジェクトにアクセスしたり、このオブジェクトを提供するための CORBA ベースのアプリケーションを開発できます。また、ORB のコマンド行のオプションを使用すると、カスタマイズが可能になります。

パラメータ

`[-ORBid {BEA_IIOP | BEA_TOBJ}]`

値 `BEA_IIOP` は、IIOP または IIOP-SSL プロトコルで通信するクライアント環境とサーバ環境をサポートするように、ORB がコンフィギュレーションされることを明示的に指定します。
値 `BEA_TOBJ` は、BEA Tuxedo ドメイン内で TGIOP プロトコルでのみ通信可能なネイティブ・クライアント環境をサポートするように、ORB がコンフィギュレーションされることを明示的に指定します。
上記のパラメータを指定しない場合、ORB は、ORB 自身がデプロイされ、その環境で使用するようにコンフィギュレーションされる環境を検出します。

`[-ORBInitRef` *`ObjectId=ObjectURL`*`]`

`-ORBInitRef` は、ORB 初期リファレンス引数です。この引数では、初期サービスに対して任意のオブジェクト・リファレンスを指定できます。
`ObjectID` は、CORBA 仕様で定義されるサービスの既知のオブジェクト ID を表します。このメカニズムにより、ORB のインストール時に定義しなかった、新しい初期サービスの Object ID で ORB をコンフィギュレーションできます。

ObjectURL には、CORBA 仕様で定義する CORBA::ORB::string_to_object オペレーションでサポートされている任意の URL スキーマを指定できます。URL の構文に誤りがある場合、またはインプリメンテーション定義方法が無効と判定された場合、CORBA::ORB_init によって CORBA::BAD_PARAM 標準例外が発生します。表 14-1 では、この例外の一覧を示します。

表 14-1 CORBA::BAD_PARAM 標準例外のマイナー・コード

マイナー・コード	説明
BadSchemeName	指定のスキーマは ORB インプリメンテーションによって認識されました。ただし、サポートされているスキーマは、IOR、corbaloc、corbalocs、および corbaname のみです。
BadAddress	アドレスの形式が ORB インプリメンテーションで認識されません。ホスト名は、DNS に従って指定するか、またはドットで区切る形式のクラス C の IP アドレスとして指定する必要があります。
BadSchemeSpecificPart	アドレスの形式が ORB インプリメンテーションで認識されません。ホスト名は、DNS に従って指定するか、またはドットで区切る形式のクラス C の IP アドレスとして指定する必要があります。
BadSchemeSpecificPart	指定されたスキーマでは、URL の一部の形式が正しくありません。

[–ORBDefaultInitRef <ObjectURL>]

-ORBDefaultInitRef は、ORB のデフォルトの初期リファレンス引数です。この引数は、-ORBInitRef で明示的に指定されていない初期リファレンスの解決に役立ちます。また、現在の Tobj_Bootstrap オブジェクトに指定されている IIOP リスナ・アドレスと同様の機能を備えています。

-ORBInitRef 引数とは異なり、-ORBDefaultInitRef では URL が必要です。この URL は、初期オブジェクト・リファレンスを識別するための新しい URL を生成します。この URL の後にはスラッシュ文字 (/) と文字列化されたオブジェクト・キーを付けます。次に、デフォルトの初期リファレンス引数の指定例を示します。

```
-ORBDefaultInitRef corbaloc:555objs.com
```

サービスの初期リファレンスを取得するために、ORB::resolve_initial_references("NotificationService") を呼び出すと、次の新しい URL が生成されます。

```
corbaloc:555objs.com/NotificationService
```

ORB::resolve_initial_references オペレーションをインプリメントすると、新しい URL が生成され、サービスの初期リファレンスを取得するために CORBA::ORB::string_to_object が呼び出されます。-ORBDefaultInitRef 引数の値として指定した URL には、複数の場所を指定できます。これは、Tobj_Bootstrap オブジェクトで使用される場所のリストに提供されている機能とほぼ同じです。この場合、ORB では URL の構文規則に基づいて URL の場所を処理します。次に、デフォルトの初期リファレンス引数の指定例を示します。

```
-ORBDefaultInitRef corbaloc:555objs.com,555Backup.com
```

サービスの初期リファレンスを取得するために、ORB::resolve_initial_references("NameService") を呼び出すと、次の新しい URL のどちらかが生成されます。

```
corbaloc:555objs.com/NameService
```

または

```
corbaloc:555Backup.com/NameService
```

生成された URL は、サービスの初期リファレンスを取得するために、CORBA::ORB::string_to_object に渡されます。

[–ORBminCrypto [0 | 40 | 56 | 128]]

ネットワーク・リンクを確立するときに必要最低限の暗号化レベルです。ゼロ (0) は、暗号化が行われないことを示し、40、56、および 128 は暗号化キーの長さ (ビット単位) を指定します。ここで指定された最低レベルの暗号化が行われないと、リンクの確立は失敗します。
デフォルト値は 0 です。

[–ORBmaxCrypto [0 | 40 | 56 | 128]]

ネットワーク・リンクを確立するときに許容される最高の暗号化レベルです。ゼロ (0) は、暗号化が行われないことを示し、40、56、および 128 は暗号化キーの長さ (ビット単位) を指定します。デフォルトは、ライセンスで指定されている機能すべてです。–ORBmaxCrypto または –ORBmaxCrypto オプションは、国際版または米国 / カナダ版の BEA Tuxedo セキュリティ・アド・オン・パッケージがインストールされている場合にのみ使用できます。

[–ORBmutualAuth]

リモート・アプリケーションから SSL 接続を受け付けるときに証明書ベースの認証を有効にするかどうかを指定します。–ORBmutualAuth オプションは、国際版または米国 / カナダ版の BEA Tuxedo セキュリティ・アド・オン・パッケージがインストールされている場合にのみ使用できます。

[–ORBpeerValidate {detect | warn | none}]

BEA Tuxedo ORB によって開始されたアウトバウンド接続のピアのデジタル証明書を、セキュア・ソケット・レイヤ (SSL) プロトコル・ハンドシェークの一部として受信したときに、BEA Tuxedo CORBA ORB がどのように動作するかを指定します。検証は安全な接続のイニシエータのみが行います。検証では、サーバのデジタル証明書にあるドメイン名の指定と同じネットワーク・アドレスにピア・サーバが実際にあるかを確認します。この検証は、技術的には SSL プロトコルの一部ではなく、Web ブラウザで行われているチェックと同じものです。

値が detect の場合、BEA Tuxedo CORBA ORB は、接続に使用されるオブジェクト・リファレンスで指定されたホストがピアのデジタル証明書で指定されたドメイン名と一致するかを確認します。照合に失敗した場合、ORB はピアの認証を拒否し、接続を破棄します。このチェックによって、介在者の攻撃から保護します。

値が warn の場合、BEA Tuxedo CORBA ORB は、接続に使用されるオブジェクト・リファレンスで指定されたホストがピアのデジタル証明書で指定されたドメイン名と一致するかを確認します。照合に失敗した場合、ORB はユーザ・ログにメッセージを書き込み、接続処理を続行します。

値が none の場合、BEA Tuxedo CORBA ORB は、ピアの検証を行わずに接続処理を続行します。

-ORBpeerValidate オプションは、国際版または米国 / カナダ版の BEA Tuxedo セキュリティ・アド・オン・パッケージがインストールされている場合にのみ使用できます。
値を指定しない場合は、デフォルトの detect が指定されます。

[–ORBport port-number]
リモート CORBA クライアントからの接続を受け付けるために、ORB で使用するネットワーク・アドレスを指定します。通常、システム管理者は、動的範囲ではなくポート番号の「ユーザ」範囲からクライアントのポート番号を割り当てます。これにより、共同クライアント / サーバでポートの競合を防ぐことができます。
これは、BEA Tuxedo CORBA ORB が永続オブジェクト・リファレンスを作成するための必須パラメータです。永続オブジェクト・リファレンスでは、ORB が再起動した場合でも、オブジェクト・リファレンス内に格納されているのと同じポート番号を指定する必要があります。一時オブジェクト・リファレンスの場合は、ORB で動的に任意のポートを取得できます。
port-number には、BEA Tuxedo CORBA ORB プロセスが接続要求の受信をリッスンする TCP ポート番号を指定します。port-number には、0 から 65535 までの数字を指定します。

注記 Java Tobj_Bootstrap オブジェクトでは、*port-number* を格納するのに short 型を使用します。そのため、Java クライアントからの接続をサポートする場合は、0 ～ 32767 の範囲内で *port-number* を指定する必要があります。

[–ORBsecurePort port-number]
IIOP リスナ / ハンドラでセキュア・ソケット・レイヤ・プロトコルを使用して安全な接続をリッスンするために使用するポート番号を指定します。ポート番号を指定せずにコマンド行のオプションを指定した場合、OMG で割り当てられたポート番号 684 が SSL 接続に使用されます。
port-number には、BEA Tuxedo CORBA ORB プロセスが接続要求の受信をリッスンする TCP ポート番号を指定します。port-number には、0 から 65535 までの数字を指定します。

注記 Java Tobj_Bootstrap オブジェクトでは、port-number を格納するのに short 型を使用します。そのため、Java クライアントからの接続をサポートする場合は、0 ～ 32767 の範囲内で port-number を指定する必要があります。

BEA Tuxedo CORBA ORB への接続に安全なものだけを有効にするには、管理者は -ORBport と -ORBsecurePort で指定するポート番号を同じ値にコンフィギュレーションします。

–ORBsecurePort オプションは、国際版または米国 / カナダ版の BEA Tuxedo セキュリティ・アド・オン・パッケージがインストールされている場合にのみ使用できます。

移植性 BEA Tuxedo CORBA ORB は、UNIX および Microsoft Windows 2000 オペレーティング・システム上で BEA Tuxedo に組み込まれているクライアントまたはサーバとしてサポートされています。また、Windows 98 オペレーティング上で BEA Tuxedo に組み込まれているクライアントとしてもサポートされています。

相互運用性 BEA Tuxedo CORBA ORB は、TCP/IP 接続で GIOP プロトコル・バージョン 1.0、1.1、または 1.2 をサポートする、ORB 準拠のすべての IIOP と相互運用できます。また、BEA Tuxedo CORBA ORB は、オブジェクト・リファレンスで TAG_SSL_SEC_TRANS タグ付きコンポーネントの使用をサポートし、セキュア・ソケット・レイヤ・プロトコルのバージョン 3 をサポートする、ORB 準拠のすべての IIOP-SSL と相互運用できます。

例 C++ コード例

```
ChatClient –ORBid BEA_IIOP –ORBport 2100
           -ORBDefaultInitRef corbaloc:piglet:1900
           -ORBInitRef TraderService=corbaloc:owl:2530
           –ORBsecurePort 2100
           -ORBminCrypto 40
           –ORBmaxCrypto 128
           TechTopics
```

Java コード例

```
java –DORBDefaultInitRef=corbalocs:piglet:1900
.....-DORBInitRef=TraderService=corbaloc:owl:2530
     -Dorg.omg.CORBA.ORBPort=1948
     -classpath=%CLASSPATH% client
```

関連項目 ISL

Policy メンバ関数

方針は、オペレーションに関連する ORB に特定の選択を通知するオブジェクトです。この情報には、CORBA モジュールで定義された Policy インターフェイスから生成されたインターフェイスを使用する、構造化された方法でアクセスします。

注記　以下の CORBA::Policy オペレーションおよび構造体は、通常の場合、プログラマにとっては必須ではありません。通常、生成されたインターフェイスには、仕様に関連する情報が格納されています。方針オブジェクトは、特定のファクトリ、または CORBA::create_policy オペレーションを使用することで作成できます。

方針オブジェクトの C++ へのマッピングは次のとおりです。

```
class CORBA
{
  class Policy
  {
    public:
         copy();
         void destroy();
  }; //Policy
     typedef sequence<Policy>PolicyList;
}; // CORBA
```

PolicyList には、その他の C++ シーケンス・マッピングと同じものを使用します。シーケンスの使用方法については、「シーケンス」を参照してください。

関連項目 : POA Policy および CORBA::ORB::create_policy

CORBA:Policy::copy

概要 方針オブジェクトをコピーします。

C++ バインディング

```
CORBA::Policy::copy();
```

引数 特にありません。

説明 このオペレーションは、方針オブジェクトをコピーします。コピーでは、方針とドメインまたはオブジェクトとの間にあった関係は保持されません。

注記 この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値 特にありません。

CORBA::Policy::destroy

概要　方針オブジェクトを破棄します。

C++ バインディング

```
void CORBA::Policy::destroy();
```

引数　特にありません。

例外　方針オブジェクトが破棄できないと判定された場合、`CORBA::NO_PERMISSION` 例外が発生します。

説明　このオペレーションは、方針オブジェクトを破棄します。破棄の可否を判定するのは方針オブジェクトです。

注記　この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値　特にありません。

PortableServer メンバ関数

PortableServer メンバ関数の C++ へのマッピングは次のとおりです。

```
// C++
class PortableServer
{
    public:
           class LifespanPolicy;
           class IdAssignmentPolicy;
           class POA::find_POA
           class reference_to_id
           class POAManager;
           class POA;
           class Current;
           class virtual ObjectId
           class ServantBase
};
```

`ObjectId`
: 特定の抽象 CORBA オブジェクトを識別するために、POA およびユーザ指定のインプリメンテーションで使用される値です。`ObjectId` 値は、POA またはインプリメンテーションで割り当ておよび管理できます。`ObjectId` 値はリファレンスによってカプセル化されるので、クライアントからは隠ぺいされます。`ObjectId` には標準の形式がないため、POA では未解釈のオクテット・シーケンスとして管理されます。

以下の節では、これ以外のクラスについて説明します。

PortableServer::POA::activate_object

概要 個別のオブジェクトを明示的に活性化します。

C++ バインディング

```
ObjectId * activate_object (
       Servant p_servant);
```

引数

`p_servant`
インターフェイスの C++ インプリメンテーション・クラスのインスタンス。

例外 指定のサーバントが既にアクティブ・オブジェクト・マップにある場合、`ServantAlreadyActive` 例外が発生します。

注記 POA でサポートされていない方針を使用する場合、ほかの例外も発生する可能性があります。

説明 このオペレーションは、`ObjectId` を生成し、`ObjectId` および指定のサーバントをアクティブ・オブジェクト・マップに入れることにより、個別のオブジェクトを明示的に活性化します。

注記 この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値 関数が成功した場合、`ObjectId` を返します。

例 次の例では、最初の構造体がユーザ定義のコンストラクタでサーバントを作成します。2 番目の構造体は、オブジェクトでの要求の処理にサーバントが使用可能であることを POA に通知します。POA は、そのオブジェクト用に作成した `ObjectId` を返します。3 番目の文は、POA に `IMPLICIT_ACTIVATION` 方針 (BEA Tuxedo ソフトウェアのバージョン 4.2 でのみサポートされている方針) があると想定し、オブジェクトへのリファレンスを返します。これにより、このリファレンスは呼び出し用にクライアントで処理が可能になります。クライアントがリファレンスを呼び出すと、作成されたばかりのサーバントに要求が返されます。

```
MyFooServant* afoo = new MyFooServant(poa,27);
PortableServer::ObjectId_var oid =
        poa->activate_object(afoo);
Foo_var foo = afoo->_this();
```

PortableServer::POA::activate_object_with_id

概要 指定の ObjectId で個別のオブジェクトを活性化します。

C++ バインディング

```
void activate_object_with_id (
        const ObjectId & id,
        Servant p_servant);
```

引数

id
: そのオペレーションが呼び出されたオブジェクトを識別する ObjectId。

p_servant
: インターフェイスの C++ インプリメンテーション・クラスのインスタンス。

例外 ObjectId 値で示された CORBA オブジェクトが POA で既に活性化されている場合、ObjectAlreadyActive 例外が発生します。

サーバントが既にアクティブ・オブジェクト・マップにある場合、ServantAlreadyActive 例外が発生します。

注記　POA でサポートされていない方針を使用する場合、ほかの例外も発生する可能性があります。

POA に SYSTEM_ID 方針があり、ObjectId 値をシステムが生成しなかったり、POA に対して生成されなかったことを検出した場合、BAD_PARAM システム例外が発生する可能性があります。こういった無効の ObjectId 値をすべて検出するのに、ORB は特に必要ではありません。ただし、POA に対してシステムによって以前に生成された ObjectId 値を持つ SYSTEM_ID 方針が POA にある場合、または以前にインスタンス化した同じ POA について PERSISTENT 方針が POA にある場合、移植可能なアプリケーションでは POA の activate_object_with_id を呼び出さないでください。

説明 このオペレーションは、アクティブ・オブジェクト・マップで指定の ObjectId と指定のサーバントとを関連付けます。

注記　この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値 特にありません。

例

```
MyFooServant* afoo = new MyFooServant(poa, 27);
PortableServer::ObjectId_var oid =
           PortableServer::string_to_ObjectId("myLittleFoo");
poa->activate_object_with_id(oid.in(), afoo);
Foo_var foo = afoo->_this();
```

PortableServer::POA::create_id_assignment_policy

概要 IdAssignmentPolicy インターフェイスを持つオブジェクトを取得します。これにより、ユーザは POA::create_POA オペレーションにオブジェクトを渡すことができます。

C++ バインディング

```
IdAssignmentPolicy_ptr
        PortableServer::POA::create_id_assignment_policy (
                PortableServer::IdAssignmentPolicyValue value)
```

引数

value

値は、ObjectId がアプリケーションによってのみ割り当てられることを示す PortableServer::USER_ID か、または ObjectId がシステムによってのみ割り当てられることを示す PortableServer::SYSTEM_ID のどちらかです。

説明 POA::create_id_assignment_policy オペレーションは、IdAssignmentPolicy インターフェイスを持つオブジェクトを取得します。この方針に POA::create_POA オペレーションが渡されると、この方針は作成される POA の ObjectId をアプリケーションが生成するか、ORB が生成するかを指定します。指定可能な値は次のとおりです。

- PortableServer::USER_ID Å\ アプリケーションのみが、その POA で作成されるオブジェクトに ObjectId を割り当てます。
- PortableServer::SYSTEM_ID ― POA のみが、その POA で作成されるオブジェクトに ObjectId を割り当てます。POA に PERSISTENT LifespanPolicy もある場合は、割り当てられた ObjectId は、同じ POA のすべてのインスタンスを通じて一意でなければなりません。

IdAssignmentPolicy が POA 作成時に指定されない場合、SYSTEM_ID がデフォルトに指定されます。

注記　この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値 Id Assignment 方針を返します。

PortableServer::POA::create_lifespan_policy

概要 LifespanPolicy インターフェイスを持つオブジェクトを取得します。これにより、ユーザは POA::create_POA オペレーションにオブジェクトを渡すことができます。

C++ バインディング

```
LifespanPolicy_ptr
       PortableServer::POA::create_lifespan_policy (
              PortableServer::LifespanPolicyPolicyValue value)
```

引数

value

値は、ObjectId がアプリケーションによってのみ割り当てられることを示す PortableServer::USER_ID か、または ObjectId がシステムによってのみ割り当てられることを示す PortableServer::SYSTEM_ID のどちらかです。

説明 POA::create_lifespan_policy オペレーションで LifespanPolicy インターフェイスを持つオブジェクトを取得します。このオブジェクトは、POA::create_POA オペレーションに渡され、作成された POA でインプリメントされるオブジェクトの寿命を指定します。指定可能な値は次のとおりです。

- TRANSIENT—POA でインプリメントされたオブジェクトは、最初に作成されたときのプロセスの存続期間を延ばすことができません。POA が非活性化されると、その POA から生成されたオブジェクト・リファレンスを使用した場合、OBJECT_NOT_EXIST 例外が発生します。
- PERSISTENT — POA でインプリメントされたオブジェクトは、最初に作成されたときのプロセスの存続期間を延ばすことができます。
  - 永続オブジェクトには、それに関連付けられた POA (永続オブジェクトを作成した POA) があります。ORB が永続オブジェクトで要求を受け取ると、まず、POA の名前とそのすべての上位オブジェクトを基準にして、一致する POA を検索します。
  - ここでは取り上げない管理作業が必要になる場合があります。たとえば、ORB の作成のサービスの場所およびこの POA の寿命を通知する作業や、オプションでこの POA をインプリメントするプロセスの活性化を要求に応じて調整する作業などです。

- POA の名前は、その包含スコープ (親 POA) 内で一意でなければなりません。移植可能なプログラムでは、自身の POA の名前とほかのプロセスで使用される POA の名前が衝突しないことを前提にしています。CORBA インプリメンテーションに準拠すると、このプロパティを保証するメソッドが提供されます。

`LifespanPolicy` オブジェクトが `POA::create_POA` に渡されない場合、寿命方針のデフォルトは `TRANSIENT` に設定されます。

注記　この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値　LifespanPolicy を返します。

PortableServer::POA::create_POA

概要 新しい POA を対象の POA の子として作成します。

C++ バインディング

```
POA_ptr PortableServer::create_POA (
              const char * adapter_name,
              POAManager_ptr a_POAManager,
               const CORBA::PolicyList & policies)
```

引数

`adapter_name`
作成する POA の名前。

`a_POAManager`
新しい POAManager が作成されて新しい POA に関連付けられることを示す NULL 値か、または既存の POAManager へのポインタ。

`policies`
新しい POA に関連付ける方針オブジェクト。

例外

`AdapterAlreadyExists`
対象の POA に指定の名前を持つ子 POA が既にある場合に発生します。

`InvalidPolicy`
指定の方針オブジェクトが ORB インプリメンテーションで無効な場合、指定した方針オブジェクトに競合がある場合、または指定の方針オブジェクトで事前の管理作業が必要にもかかわらず実行していない場合に発生します。この例外では、最初に方針オブジェクトと競合が生じた方針パラメータ値のインデックスが記述されます。

`IMP_LIMIT`
CORBA::ORB_init オペレーションで説明したポートの設定を行わずに、`PERSISTENT` の LifespanPolicy で POA の作成をプログラムで試行した場合に発生します。

説明 このオペレーションは、新しい POA を対象の POA の子として作成します。指定する名前は一意でなければなりません。この名前で同じ親 POA を持つほかの POA と新しい POA とを識別します。

a_POAManager パラメータが NULL の場合、新しい PortableServer::POAManager オブジェクトが作成され、新しい POA に関連付けられます。それ以外の場合、指定した POAManager オブジェクトは新しい POA に関連付けられます。POAManager オブジェクトは、属性名 the_POAManager を使用して取得できます。

指定した方針オブジェクトは POA に関連付けられ、その動作の制御に使用されます。方針オブジェクトは、このオペレーションが戻り値を返す前にコピーされます。したがって、アプリケーションでは POA の使用中にいつでも方針オブジェクトを破棄できます。方針は、親 POA から継承されません。

注記　この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値　作成された POA へのポインタを返します。

例　**例 1**

この例では、子 POA は親 POA と同じマネージャを使用します。この場合、子 POA の状態は親と同じになります。たとえば、親が活性化されていれば、子も活性化されます。

```
CORBA::PolicyList policies(2);
policies.length (1);
policies[0] = rootPOA->create_lifespan_policy(
              PortableServer::LifespanPolicy::TRANSIENT);
PortableServer::POA_ptr poa =
       rootPOA->create_POA("my_little_poa",
              rootPOA->the_POAManager, policies);
```

例 2

この例では、新しい POA がルート POA の子として作成されています。

```
CORBA::PolicyList policies(2);
policies.length (1);
policies[0] = rootPOA->create_lifespan_policy(
              PortableServer::LifespanPolicy::TRANSIENT);
PortableServer::POA_ptr poa =
       rootPOA->create_POA("my_little_poa",
              PortableServer::POAManager::_nil(), policies);
```

PortableServer::POA::create_reference

概要 POA 生成の ObjectId 値と指定のインターフェイス・リポジトリ ID をカプセル化するオブジェクト・リファレンスを作成します。

C++ バインディング

```
CORBA::Object_ptr create_reference (
                        const char * intf)
```

引数

intf

インターフェイス・リポジトリ ID。

例外 このオペレーションには、値 SYSTEM_ID を持つ LifespanPolicy が必要です。この LifespanPolicy がない場合、PortableServer::WrongPolicy 例外が発生します。

説明 create_reference オペレーションは、POA 生成の ObjectId 値と指定のインターフェイス・リポジトリ ID をカプセル化するオブジェクト・リファレンスを作成します。このオペレーションは、POA に関連付けられた情報およびオペレーションのパラメータからのリファレンスを構成するために必要な情報を収集します。このオペレーションは、リファレンスを作成するだけで、リファレンスと活性化されたサーバントとを関連付けるわけではありません。作成したリファレンスは、クライアントに渡されます。これにより、生成された ObjectId を使用して、そのリファレンスの以降の要求を POA に返すことができます。生成された ObjectId 値を取得するには、作成したリファレンスで POA::reference_to_id を呼び出します。

注記　この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値 オブジェクトへのポインタを返します。

PortableServer::POA::create_reference_with_id

概要 指定の ObjectId とインターフェイス・リポジトリ ID 値をカプセル化するオブジェクト・リファレンスを作成します。

C++ バインディング

```
CORBA::Object_ptr create_reference_with_id (
             const ObjectId & oid,
              const char * intf)
```

引数

oid

そのオペレーションが呼び出されたオブジェクトを識別する ObjectId。

intf

ÉCÉìÉ^Å[ÉtÉFÉCÉXÅEÉäÉ|ÉWÉgÉä IDÅB

例外 POA に SYSTEM_ID 値を持つ LifespanPolicy があり、ObjectId がシステムまたは POA によって生成されなかったことをその POA が検出した場合、BAD_PARAM システム例外が発生します。

説明 create_reference オペレーションは、指定の ObjectId とインターフェイス・リポジトリ ID 値をカプセル化するオブジェクト・リファレンスを作成します。このオペレーションは、POA に関連付けられた情報およびオペレーションのパラメータからリファレンスを構成するために必要な情報を収集します。このオペレーションは、リファレンスを作成するだけで、リファレンスと活性化されたサーバントとを関連付けるわけではありません。作成したリファレンスは、クライアントに渡されます。これにより、そのリファレンスの以降の要求で、指定の ObjectId を持つ同じ POA に呼び出しを返すことができます。

注記 この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値 Object_ptr を返します。

例

```
PortableServer::ObjectId_var oid =
       PortableServer::string_to_ObjectId("myLittleFoo");
CORBA::Object_var obj = poa->create_reference_with_id(
              oid.in(), "IDL:Foo:1.0");
Foo_var foo = Foo::_narrow(obj);
```

PortableServer::POA::deactivate_object

概要　アクティブ・オブジェクト・マップから ObjectId を削除します。

C++ バインディング

```
void deactivate_object (
                const ObjectId & oid)
```

引数

oid

オブジェクトを識別する ObjectId。

例外　指定の ObjectId に関連付けられた活性化されたオブジェクトがない場合、ObjectNotActive 例外が発生します。

説明　このオペレーションは、oid パラメータで指定された ObjectId およびそのサーバントをアクティブ・オブジェクト・マップから削除します。

注記　この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値　特にありません。

PortableServer::POA::destroy

概要 POA およびすべての下位 POA を破棄します。

C++ バインディング

```
void destroy (
              CORBA::Boolean etherealize_objects,
               CORBA::Boolean wait_for_completion)
```

引数

`etherealize_objects`
このリリースの BEA Tuxedo では、この引数は `FALSE` に指定します。

`wait_for_completion`
この引数は、オペレーションが直ちに戻り値を返すかどうかを示します。

説明 このオペレーションは、POA およびすべての下位 POA を破棄します。POA とその名前は、同じプロセスで後から再作成できます。ただし、`POAManager::deactivate` オペレーションで同じプロセスによる関連付けられた POA の再作成を無効にしている場合は除きます。

POA が破棄された場合、実行を開始した要求については、完了するまで処理が続行されます。実行を開始していない要求については、それが新しく受け取られ、POA がないような状態で処理されます。そのため、要求は拒否され、`OBJECT_NON_EXIST` 例外が発生します。

`wait_for_completion` パラメータが `TRUE` の場合、プロセス内の要求がすべて完了し、`etherealize` の呼び出しがすべて完了した後にのみ、`destroy` オペレーションは戻り値を返します。それ以外の場合、`destroy` オペレーションは POA の破棄後に戻り値を返します。

注記 このリリースの BEA Tuxedo では、マルチスレッドをサポートしていません。そのため、オブジェクト呼び出しのコンテキストで呼び出しを行う場合、`wait_for_completion` は `TRUE` に指定しないでください。この指定を行うと、POA はそれが実行中のときに自身を解放できなくなります。

注記 この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値 特にありません。

PortableServer::POA::find_POA

概要 指定の名前を持つ子 POA へのリファレンスを返します。

C++ バインディング

```
void find_POA( in string adapter_name, in boolean activate_it);
```

引数

`adapter_name`
対象の POA へのリファレンス。

`active_it`
このバージョンの BEA Tuxedo では、このパラメータは `FALSE` に指定しなければなりません。

例外

`AdapterNonExistent`
この例外は、POA が存在しない場合に発生します。

説明 POA に指定の名前を持つ子 POA がある場合、その子 POA を返します。指定の名前を持つ子 POA が存在せず、`activate_it` パラメータの値が `FALSE` の場合、`AdapterNonExistent` 例外が発生します。

戻り値 特にありません。

PortableServer::POA::reference_to_id

概要 指定の reference によってカプセル化された ObjectId 値を返します。

C++ バインディング

```
ObjectId reference_to_id(in Object reference);
```

引数 reference

オブジェクトへのリファレンスを指定します。

例外 WrongAdapter

その POA によってリファレンスが作成されなかった場合に発生します。

説明 このオペレーションは、指定の reference によってカプセル化された ObjectId 値を返します。このオペレーションを実行中の POA によってリファレンスが作成された場合にのみ、このオペレーションは有効です。リファレンスで示されたオブジェクトが活性化されているかどうかは、このオペレーションの成否には関係ありません。

注記 この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値 指定の reference によってカプセル化された ObjectId 値を返します。

PortableServer::POA::the_POAManager

概要 POA に関連付けられた POA マネージャを識別します。

C++ バインディング

```
POAManager_ptr the_POAManager ();
```

引数 特にありません。

説明 この属性は読み取り専用で、POA に関連付けられた POA マネージャを識別します。

注記 この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値 特にありません。

例

```
poa->the_POAManager()->activate();
```

この文では、POA の POAManager の状態を活性化に設定します。POAManager は、POA で要求を受け付けるときに必要になります。POA には親があるので、ルート POA ではありません。POA の親の POAManager もすべて、この文で活性化状態にして有効にする必要があります。

PortableServer::ServantBase::_default_POA

概要 サーバントに関連付けられた POA へのオブジェクト・リファレンスを返します。

C++ バインディング

```
class PortableServer
{
class ServantBase
  {
  public:
  virtual POA_ptr _default_POA();
  }
}
```

引数 特にありません。

説明 C++ サーバントはすべて、PortableServer::ServantBase から継承するので、_default_POA 関数も継承します。このバージョンの BEA Tuxedo では、通常、_default_POA を使用する必要はありません。

この関数のデフォルトのインプリメンテーションでは、このプロセスのデフォルト ORB のルート POA へのオブジェクト・リファレンスを返します。これは、ORB::resolve_initial_references("RootPOA") を呼び出したときの戻り値と同じです。C++ サーバントは、この定義を無効にして目的に応じた POA を返すことができます。

注記　この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値 サーバントに関連付けられたデフォルト POA。

POA Current メンバ関数

CORBA::Current から生成される PortableServer::Current インターフェイスは、メソッドが呼び出されたオブジェクトの ID にアクセスしてメソッド・インプリメンテーションを提供します。

PortableServer::Current::get_object_id

概要 呼び出されるオブジェクトをそのコンテキストで識別する `ObjectId` を返します。

C++ バインディング

```
ObjectId * get_object_id ();
```

引数 特にありません。

例外 POA ディスパッチ・オペレーションのコンテキストの外部で呼び出された場合、`PortableServer::NoContext` 例外が発生します。

説明 このオペレーションは、呼び出されるオブジェクトをそのコンテキストで識別する `PortableServer::ObjectId` を返します。

注記　この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値 このオペレーションは、呼び出されるオブジェクトをそのコンテキストで識別する `ObjectId` を返します。

PortableServer::Current::get_POA

概要 呼び出されるオブジェクトをそのコンテキストでインプリメントする POA へのリファレンスを返します。

C++ バインディング

```
POA_ptr get_POA ();
```

引数 特にありません。

例外 POA ディスパッチ・オペレーションのコンテキストの外部で呼び出された場合、PortableServer::NoContext 例外が発生します。

説明 このオペレーションは、呼び出されるオブジェクトをそのコンテキストでインプリメントする POA へのリファレンスを返します。

注記　この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値 このオペレーションは、呼び出されるオブジェクトをそのコンテキストでインプリメントする POA へのリファレンスを返します。

POAManager メンバ関数

各 POA オブジェクトには、関連付けられた POAManager オブジェクトがあります。POAManager は、1 つまたは複数の POA オブジェクトと関連付けることができます。POAManager は、関連付けられた POA の処理状態をカプセル化します。POA マネージャのオペレーションを使用すると、アプリケーションで POA の要求をキューに登録したり、破棄したり、POA を非活性化したりできます。

POA マネージャは、暗黙的に作成および破棄されます。POA の作成時に POAManager オブジェクトを明示的に指定しない限り、POAManager は POA の作成時に作成され、自動的に POA に関連付けられます。POAManager オブジェクトは、関連付けられた POA がすべて破棄されたときに暗黙的に破棄されます。

POAManager には、活性化、非活性化、保持、破棄の 4 つの処理状態があります。この処理状態によって、関連付けられた POA の機能、および POA が受け取った要求を破棄するかどうかが決まります。

POAManager は保持状態で作成されます。この状態では、POAManager が活性化状態に移行するまで、POA の呼び出しはすべてキューに登録されます。このバージョンの BEA Tuxedo では、活性化または非活性化にだけ状態を移行することができます。したがって、このバージョンでは、保持状態に戻したり、破棄状態に移行したりすることができません。

PortableServer::POAManager::activate

概要 POAManager の状態を活性化に変更します。

C++ バインディング

```
void activate();
```

引数 特にありません。

例外 POAManager の状態が非活性化のときにこのオプションを使用すると、`PortableServer::POAManager::AdapterInactive` 例外が発生します。

説明 このオペレーションは、POAManager の状態を活性化に変更します。状態を活性化に移行すると、関連付けられた POA は要求を処理できるようになります。

注記　POA による要求の処理では、すべての親 POA にも活性化状態の POAManager がなければなりません。

注記　この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値 特にありません。

PortableServer::POAManager::deactivate

概要 POAManager の状態を非活性化に変更します。

C++ バインディング

```
void deactivate (
        CORBA::Boolean etherealize_objects,
        CORBA::Boolean wait_for_completion);
```

引数

`etherealize_objects`

BEA WebLogic Enterprise バージョン 4.2 以降のソフトウェア、および BEA Tuxedo バージョン 8.0 以降のソフトウェアでは、この引数は常に `FALSE` に設定する必要があります。

`wait_for_completion`

この引数を `TRUE` に指定すると、`deactivate` オペレーションは、プロセス内の要求がすべて完了した後にのみ戻り値を返します。この引数を `FALSE` に指定すると、`deactivate` オペレーションは、関連付けられた POA の状態の変更後に戻り値を返します。

例外 POA マネージャの状態が非活性化のときに使用すると、`PortableServer::POAManager::AdapterInactive` 例外が発生します。

説明 このオペレーションは、POAManager の状態を非活性化に変更します。状態を非活性化に移行すると、関連付けられた POA は、未実行の要求およびすべての新しい要求を拒否します。

注記 このリリースの BEA Tuxedo では、マルチスレッドをサポートしていません。そのため、オブジェクト呼び出しのコンテキストで呼び出しを行う場合、`wait_for_completion` は `TRUE` に指定しないでください。したがって、POAManager が現在実行中の場合、POAManager は非活性化に設定できません。

注記 この関数がサポートされるのは、共同クライアント / サーバのみです。

戻り値 特にありません。

POA 方針メンバ・オブジェクト

CORBA::Policy から生成されたインターフェイスは、POA に適用する方針を指定するために POA::create_POA オペレーションで使用されます。方針オブジェクトは、ルート POA と同様に既存の POA のファクトリ・オペレーションを作成します。方針オブジェクトは、POA の作成時に指定します。既存の POA では方針は変更できません。方針は、親 POA から継承されません。

PortableServer::LifespanPolicy

概要 create_POA オペレーションにオブジェクトの寿命を指定します。

説明 POA::create_lifespan_policy オペレーションで LifespanPolicy インターフェイスを持つオブジェクトを取得します。このオブジェクトは、POA::create_POA オペレーションに渡され、作成された POA でインプリメントされるオブジェクトの寿命を指定します。指定可能な値は次のとおりです。

- TRANSIENT ― POA でインプリメントされたオブジェクトは、最初に作成されたときのプロセスの存続期間を延ばすことができません。
- PERSISTENT ― POA でインプリメントされたオブジェクトは、最初に作成されたときのプロセスの存続期間を延ばすことができます。

  永続オブジェクトには、それに関連付けられた POA (永続オブジェクトを作成した POA) があります。ORB が永続オブジェクトで要求を受け取ると、POA の名前とそのすべての上位オブジェクトを基準にして、一致する POA を検索します。

  POA の名前は、その包含スコープ (親 POA) 内で一意でなければなりません。移植可能なプログラムでは、自身の POA の名前とほかのプロセスで使用される POA の名前が衝突しないことを前提にしています。

LifespanPolicy オブジェクトが create_POA に渡されない場合、寿命方針のデフォルトは TRANSIENT に設定されます。

注記　この関数がサポートされるのは、共同クライアント / サーバのみです。

例外 特にありません。

PortableServer::IdAssignmentPolicy

概要 作成された POA の ObjectIds の生成元をアプリケーションにするか、ORB にするかを指定します。

説明 POA::create_id_assignment_policy オペレーションで IdAssignmentPolicy インターフェイスを持つオブジェクトを取得します。このオブジェクトは、POA::create_POA オペレーションに渡され、作成された POA の ObjectId の生成元をアプリケーションにするか、ORB にするかを指定します。指定可能な値は次のとおりです。

- USER_ID Å\ アプリケーションのみが、その POA で作成されるオブジェクトに ObjectId を割り当てます。
- SYSTEM_ID ― POA のみが、その POA で作成されるオブジェクトに ObjectId を割り当てます。POA に PERSISTENT 方針もある場合は、割り当てられた ObjectId は、同じ POA のすべてのインスタンスを通じて一意でなければなりません。

IdAssignmentPolicy が POA 作成時に指定されない場合、SYSTEM_ID がデフォルトに指定されます。

注記 この関数がサポートされるのは、共同クライアント / サーバのみです。

Request メンバ関数

ExceptionList メンバ関数の C++ へのマッピングは次のとおりです。

```
// C++
class Request
{
  public:
    Object_ptr target() const;
    const char *operation() const;
    NamedValue_ptr result();
    NVList_ptr arguments();
    Environment_ptr env();
    ExceptionList_ptr exceptions();
    ContextList_ptr contexts();
    void ctx(Context_ptr);
    Context_ptr ctx() const

    // 引数操作ヘルパ関数
    Any &add_in_arg();
    Any &add_in_arg(const char* name);
    Any &add_inout_arg():
    Any &add_inout_arg(const char* name);
    Any &add_out_arg():
    Any &add_out_arg(const char* name);
    void set_return_type(TypeCode_ptr tc);
    Any &return_value();

    void invoke();
    void send_oneway();
    void send_deferred();
    void get_response();
    Boolean poll_response();
};
```

注記　`add_*_arg`、`set_return_type`、および `set_return_type` メンバ関数は、属性ベースのアクセサの使用を簡略化するものとして追加されます。

以下の節では、TypeCode の各メンバ関数について説明します。

CORBA::Request::arguments

概要 要求の引数リストを取得します。

C++ バインディング

```
CORBA::NVList_ptr CORBA::Request::arguments () const;
```

引数 特にありません。

説明 このメンバ関数は、要求の引数リストを取得します。arguments には、`input` または `output`、あるいはその両方を指定できます。

戻り値 関数が成功した場合、戻り値は要求のオペレーションの引数リストへのポインタです。返された引数リストは Request オブジェクト・リファレンスが所有するため、解放しないでください。

関数が失敗した場合、例外がスローされます。

CORBA::Request::ctx(Context_ptr)

概要 オペレーションの Context オブジェクトを設定します。

C++ バインディング

```
void CORBA::Request::ctx (
          CORBA::Context_ptr       CtxObject);
```

引数

`CtxObject`
Context オブジェクトの設定に使用する新しい値。

説明 このメンバ関数は、オペレーションの Context オブジェクトを設定します。

戻り値 特にありません。

関連項目 `CORBA::Request::ctx()`

CORBA::Request::get_response

概要　特定の遅延同期要求の応答を取得します。

C++ バインディング

```
void CORBA::Request::get_response ();
```

引数　特にありません。

説明　このメンバ関数は、特定の要求の応答を取得します。CORBA::Request::send_deferred 関数または CORBA::Request::send_multiple_requests 関数の呼び出し後に使用されます。要求が完了していない場合、CORBA::Request::get_response 関数によって要求が完了するまでブロックされます。

戻り値　特にありません。

関連項目　CORBA::Request::send_deferred

CORBA::Request::invoke

概要 要求で指定されたオペレーションで呼び出しを実行します。

C++ バインディング

```
void CORBA::Request::invoke ();
```

引数 特にありません。

説明 このメンバ関数は、オブジェクト・リクエスト・ブローカ (ORB) を呼び出して、適切なサーバ・アプリケーションに要求を送信します。

戻り値 特にありません。

CORBA::Request::operation

概要 要求用のオペレーションを取得します。

C++ バインディング

```
const char * CORBA::Request::operation () const;
```

引数 特にありません。

説明 このメンバ関数は、要求用のオペレーションを取得します。

戻り値 関数が成功した場合、戻り値はオブジェクト用のオペレーションへのポインタです。値には 0 (ゼロ) を指定できます。返されたメモリは Request オブジェクトが所有するため、解放しないでください。

関数が失敗した場合、例外がスローされます。

CORBA::Request::poll_response

概要 遅延同期要求が完了したかどうかを判別します。

C++ バインディング

```
CORBA::Boolean CORBA::Request::poll_response ();
```

引数 特にありません。

説明 このメンバ関数は、要求が完了したかどうか、完了後すぐに戻り値が返されたかどうかを判別します。このメンバ関数を使用すると、要求の状態をチェックできます。また、CORBA::Request::get_response の呼び出しがブロックするかどうかの判別にも使用できます。

戻り値 関数が成功した場合、戻り値は、応答が完了済みのときは CORBA_TRUE、応答が未完了のときは CORBA_FALSE になります。

関数が失敗した場合、例外がスローされます。

関連項目

```
CORBA::ORB::get_next_response
CORBA::ORB::poll_next_response
CORBA::ORB::send_multiple_requests
CORBA::Request::get_response
CORBA::Request::send_deferred
```

CORBA::Request::result

概要 要求の結果を取得します。

C++ バインディング

```
CORBA::NamedValue_ptr CORBA::Request::result ();
```

引数 特にありません。

説明 このメンバ関数は、要求の結果を取得します。

戻り値 関数が成功した場合、戻り値はオペレーションの結果へのポインタです。返された結果は Request オブジェクトが所有するため、解放しないでください。

関数が失敗した場合、例外がスローされます。

CORBA::Request::env

概要 要求の環境を取得します。

C++ バインディング

```
CORBA::Environment_ptr CORBA::Request::env ();
```

引数 特にありません。

説明 このメンバ関数は、要求の環境を取得します。

戻り値 関数が成功した場合、戻り値はオペレーションの環境へのポインタです。返された環境は Request オブジェクトが所有するため、解放しないでください。

関数が失敗した場合、例外がスローされます。

CORBA::Request::ctx

概要 要求のコンテキストを取得します。

C++ バインディング

```
CORBA::context_ptr CORBA::Request::ctx ();
```

引数 特にありません。

説明 このメンバ関数は、要求のコンテキストを取得します。

戻り値 関数が成功した場合、戻り値はオペレーションのコンテキストへのポインタです。返されたコンテキストは Request オブジェクトが所有するため、解放しないでください。

関数が失敗した場合、例外がスローされます。

CORBA::Request::contexts

概要 要求のコンテキスト・リストを取得します。

C++ バインディング

```
CORBA::ContextList_ptr CORBA::Request::contexts ();
```

引数 特にありません。

説明 このメンバ関数は、要求のコンテキスト・リストを取得します。

戻り値 関数が成功した場合、戻り値はオペレーションのコンテキスト・リストへのポインタです。返されたコンテキスト・リストは Request オブジェクトが所有するため、解放しないでください。

関数が失敗した場合、例外がスローされます。

CORBA::Request::exceptions

概要 要求の例外リストを取得します。

C++ バインディング

```
CORBA::ExceptionList_ptr CORBA::Request::exceptions ();
```

引数 特にありません。

説明 このメンバ関数は、要求の例外リストを取得します。

戻り値 関数が成功した場合、戻り値は要求の例外リストへのポインタです。返された例外リストは Request オブジェクトが所有するため、解放しないでください。

関数が失敗した場合、例外がスローされます。

CORBA::Request::target

概要 要求の対象のオブジェクト・リファレンスを取得します。

C++ バインディング

```
CORBA::Object_ptr CORBA::Request::target () const;
```

引数 特にありません。

説明 このメンバ関数は、要求の対象のオブジェクト・リファレンスを取得します。

戻り値 関数が成功した場合、戻り値はオペレーションの対象のオブジェクトへのポインタです。戻り値は Request オブジェクトが所有するため、解放しないでください。

関数が失敗した場合、例外がスローされます。

CORBA::Request::send_deferred

概要 遅延同期要求を開始します。

C++ バインディング

```
void CORBA::Request::send_deferred ();
```

引数 特にありません。

説明 このメンバ関数は、遅延同期要求を開始します。この関数は、応答が要求されたときに CORBA::Request::get_response 関数と共に使用します。

戻り値 特にありません。

関連項目

```
CORBA::ORB::get_next_response
CORBA::ORB::poll_next_response
CORBA::ORB::send_multiple_requests
CORBA::Request::get_response
CORBA::Request::poll_response
CORBA::Request::send_oneway
```

CORBA::Request::send_oneway

概要 一方向の要求を開始します。

C++ バインディング

```
void CORBA::Request::send_oneway ();
```

引数 特にありません。

説明 このメンバ関数は一方向の要求を開始します。この関数では、応答は要求されません。

戻り値 特にありません。

関連項目

```
CORBA::ORB::send_multiple_requests
CORBA::Request::send_deferred
```

文字列

文字列の各関数の C++ へのマッピングは次のとおりです。

```
// C++

namespace CORBA {
    static char *  string_alloc(ULong len);
    static char *  string_dup (const char *);
    static void    string_free(char *);
    ...
}
```

注記　C++ では、char の静的配列は char* に移行が進んでいます。したがって、String_var に静的配列を割り当てる際は注意が必要になります。これは、String_var では、string_alloc で割り当てられたデータをポインタが指すことを前提としており、最終的には string_free でデータを解放しようとするためです。

このような動作に対応するために、ANSI/ISO C++ では、文字列リテラルが char* から const char* に変更されています。ただし、ほどんどの C++ コンパイラではこの変更をインプリメントしていないため、移植可能なプログラムを作成する際は、上記の注意事項に留意しなければなりません。

以下の節では、文字列に割り当てられるメモリを管理する各関数について説明します。

CORBA::string_alloc

概要 文字列用のメモリを割り当てます。

C++ バインディング

```
char * CORBA::string_alloc(ULong len);
```

引数

`len`

メモリに割り当てる文字列の長さ。

説明 このメンバ関数は、文字列用のメモリを動的に割り当てます。割り当てが実行できなかった場合は、ニル・ポインタを返します。また、`len+1` 文字を割り当てるので、結果として得られた文字列には後続の NULL 文字を保持するのに十分なスペースを確保できます。このメンバ関数で割り当てられたメモリを解放するには、`CORBA::string_free` メンバ関数を呼び出します。

この関数では、CORBA 例外はスローされません。

戻り値 関数が成功した場合、戻り値は文字列オブジェクト用に新しく割り当てられたメモリへのポインタです。関数が失敗した場合、戻り値はニル・ポインタです。

例

```
char* s = CORBA::string_alloc(10);
```

関連項目

`CORBA::string_free`
`CORBA::string_dup`

CORBA::string_dup

概要 文字列のコピーを作成します。

C++ バインディング

```
char * CORBA::string_dup (const char * Str);
```

引数

`Str`
コピーする文字列のアドレス。

説明 このメンバ関数は、NULL 文字も含め文字列引数のコピーを保持するのに十分なメモリを割り当ててから、そのメモリに文字列引数をコピーし、新しい文字列へのポインタを返します。

この関数では、CORBA 例外はスローされません。

戻り値 関数が成功した場合、戻り値は新しい文字列へのポインタです。関数が失敗した場合、戻り値はニル・ポインタです。

例

```
char* s = CORBA::string_dup("hello world");
```

関連項目

`CORBA::string_free`
`CORBA::string_alloc`

CORBA::string_free

概要 文字列に割り当てられたメモリを解放します。

C++ バインディング

```
void CORBA::string_free(char * Str);
```

引数

`Str`
割り当てを解除するメモリのアドレス。

説明 このメンバ関数は、`CORBA::string_alloc()` または `CORBA::string_dup()` メンバ関数で文字列に割り当てられたメモリの割り当てを解除します。この関数にニル・ポインタを渡すと、すべてのアクションが実行されなくなります。

この関数では、CORBA 例外はスローされない場合があります。

戻り値 特にありません。

例

```
char* s = CORBA::string_dup("hello world");
CORBA::string_free(s);
```

関連項目

```
CORBA::string_alloc
CORBA::string_dup
```

ワイド文字列

C++ では、バウンディッド・ワイド文字列型とアンバウンディッド・ワイド文字列型は共に CORBA::WChar* にマッピングします。また、CORBA モジュールでは、WString_var および WString_out クラスを定義します。これらの各クラスは、同じメンバ関数に、対応する文字列として同じセマンティクスを提供します。ただし、これらのクラスがワイド文字列およびワイド文字を扱う場合は例外です。

ワイド文字列の動的な割り当てまたは割り当て解除は、次の関数で実行します。

```
// C++
namespace CORBA {
   // ...
   WChar *wstring_alloc(ULong len);
   WChar *wstring_dup(const WChar* ws);
   void wstring_free(WChar*);
};
```

上記のメンバ関数は、ワイド文字列を処理する点を除いて、文字列型の同じ関数として同じセマンティクスを持っています。

一般的なマッピング・インプリメンテーションでは、直接 C++ 入出力ストリームで WString_var および WString_out を使用するために、オーバーロードの operator<< (挿入) と operator>> (抽出) が提供されます。

これらのメンバ関数については、「文字列」の対応する関数を参照してください。

リスト 14-1 では、ワイド文字列およびワイド文字を使用したコード例を示します。

コード リスト 14-1　**ワイド文字列の例**

```
// ユーザから文字列を取得
cout << "String?";
char mixed[256]; // これは十分な大きさ
char lower[256];
```

```
char upper[256];
wchar_t wmixed[256];

cin >> mixed;
// 文字列をワイド文字列に変換
// これはサーバの要求によるもの
mbstowcs(wmixed, mixed, 256);

// 文字列を大文字に変換
CORBA::WString_var v_upper = CORBA::wstring_dup(wmixed);
v_simple->to_upper(v_upper.inout());
wcstombs(upper, v_upper.in(), 256);
cout << upper << endl;

// 文字列を小文字に変換
CORBA::WString_var v_lower = v_simple->to_lower(wmixed);
wcstombs(lower, v_lower.in(), 256);
cout << lower << endl;

// すべて正常に完了
return 0;
```

TypeCode メンバ関数

TypeCode は、OMG IDL の型情報を表します。

TypeCode のコンストラクタは定義しません。ただし、マッピング・インターフェイス、および各基本型と各定義済み OMG IDL 型には、インプリメンテーションで TypeCode 擬似オブジェクト・リファレンス (TypeCode_ptr) へのアクセスが提供されます。この擬似オブジェクト・リファレンスには、_tc_<type> という形式が付きます。この形式は、Any での型を設定したり、equal の引数として使用したりできます。これらの TypeCode リファレンス定数の名前では、<type> は定義したスコープの範囲内にある型のローカル名を参照します。各 C++ _tc_<type> 定数は、一致した型と同じスコープ・レベルで定義します。

ほかのサーバレス・オブジェクトと同様に、TypeCode への C++ マッピングには _nil() オペレーションがあります。このオペレーションは、TypeCode へのニル・オブジェクト・リファレンスを返します。また、作成された型に埋め込まれた TypeCode リファレンスの初期化にも使用できます。ただし、ニル TypeCode リファレンスは、引数としてオペレーションに渡すことができません。これは、TypeCodes がオブジェクト・リファレンスではなく値として渡されるためです。

ExceptionList メンバ関数の C++ へのマッピングは次のとおりです。

```
class CORBA
{
  class TypeCode
  {
    public:
      class Bounds { ... };
      class BadKind { ... };

      Boolean equal(TypeCode_ptr) const;
      TCKind kind() const;
      Long param_count() const;
      Any *parameter(Long) const;
      RepositoryId id () const;
  }; // TypeCode
}; // CORBA
```

メモリ管理

TypeCode には、次の特別なメモリ管理規則があります。

- `id` 関数の戻り値の所有権は TypeCode が保持します。したがって、呼び出し側はこれらの戻り値を解放しないでください。

以下の節では、TypeCode の各メンバ関数について説明します。

CORBA::TypeCode::equal

概要 2 つの TypeCode オブジェクトが同じかどうかを判別します。

C++ バインディング

```
CORBA::Boolean CORBA::TypeCode::equal (
       CORBA::TypeCode_ptr       TypeCodeObj) const;
```

引数

`TypeCodeObj`
比較対象の TypeCode オブジェクトへのポインタ。

説明 このメンバ関数は、TypeCode オブジェクトが入力パラメータの `TypeCodeObj` と同じかどうかを判別します。

戻り値 TypeCode オブジェクトが `TypeCodeObj` パラメータと同じ場合、`CORBA_TRUE` が返されます。

TypeCode オブジェクトが `TypeCodeObj` パラメータと同じでない場合、`CORBA_FALSE` が返されます。

関数が失敗した場合、例外がスローされます。

CORBA::TypeCode::id

概要 TypeCode の ID を返します。

C++ バインディング

```
CORBA::RepositoryId CORBA::TypeCode::id () const;
```

引数 特にありません。

説明 このメンバ関数は、TypeCode の ID を返します。

戻り値 TypeCode の リポジトリ ID。

CORBA::TypeCode::kind

概要 TypeCode オブジェクト・リファレンスに格納されているデータの種類を取得します。

C++ バインディング

```
CORBA::TCKind CORBA::TypeCode::kind () const;
```

à¯êî 特にありません。

説明 このメンバ関数は、CORBA::TypeCode クラスの kind 属性を取得します。この属性は、TypeCode オブジェクト・リファレンスに格納されているデータの種類を指定します。

戻り値 メンバ関数が成功した場合、TypeCode オブジェクト・リファレンスに格納されているデータの種類を返します。TypeCode の種類とそのパラメータの一覧については、表 14-2. を参照してください。

メンバ関数が失敗した場合、例外がスローされます。

表 14-2 有効な TypeCode の種類およびパラメータ

TypeCode の種類	パラメータの一覧
CORBA::tk_null	なし
CORBA::tk_void	なし
CORBA::tk_short	なし
CORBA::tk_long	なし
CORBA::tk_long	なし
CORBA::tk_ushort	なし
CORBA::tk_ulong	なし
CORBA::tk_float	なし
CORBA::tk_double	なし
CORBA::tk_boolean	なし
CORBA::tk_char	なし

表 14-2 有効な TypeCode の種類およびパラメータ (続き)

TypeCode の種類	パラメータの一覧
CORBA::tk_wchar	なし
CORBA::tk_octet	なし
CORBA::tk_Typecode	なし
CORBA::tk_Principal	なし
CORBA::tk_objref	{interface_id}
CORBA::tk_struct	{struct-name, member-name, TypeCode, ... (repeat pairs)}
CORBA::tk_union	{union-name, switch-TypeCode, label-value, member-name, enum-id, ...}
CORBA::tk_enum	{enum-name, enum-id, ...}
CORBA::tk_string	{maxlen-integer}
CORBA::tk_wstring	{maxlen-integer}
CORBA::tk_sequence	{TypeCode, maxlen-integer}
CORBA::tk_array	{TypeCode, length-integer}

CORBA::TypeCode::param_count

概要 TypeCode オブジェクト・リファレンスのパラメータ数を取得します。

C++ バインディング

```
CORBA::Long CORBA::TypeCode::param_count () const;
```

引数 特にありません。

説明 このメンバ関数は、CORBA::TypeCode クラスのパラメータ属性を取得します。この属性は、TypeCode オブジェクト・リファレンスのパラメータ数を指定します。各種のパラメータの一覧については、表 14-2 を参照してください。

戻り値 関数が成功した場合、TypeCode オブジェクト・リファレンスに格納されているパラメータ数を返します。

関数が失敗した場合、例外がスローされます。

CORBA::TypeCode::parameter

概要 インデックス入力引数で指定されたパラメータを取得します。

C++ バインディング

```
CORBA::Any * CORBA::TypeCode::parameter (
        CORBA::Long             Index) const;
```

引数

`Index`

パラメータ・リストのインデックス。取得するパラメータを指定するのに使用します。

説明 このメンバ関数は、インデックス入力引数で指定されたパラメータを取得します。各種のパラメータの一覧については、表 14-2 を参照してください。

戻り値 メンバ関数が成功した場合、戻り値はインデックス入力引数で指定されたパラメータへのポインタです。

メンバ関数が失敗した場合、例外がスローされます。

Exception メンバ関数

BEA Tuxedo ソフトウェアでは、例外のスローとキャッチをサポートしています。

Caution: 例外コンストラクタを誤って使用すると、データ・メンバが初期化されなくなります。reason フィールド付きで定義する例外は、データ・メンバを初期化するコンストラクタを使用して作成する必要があります。デフォルトのコンストラクタを使用した場合は、そのデータ・メンバは初期化されず、例外の破棄時にシステムによって存在しないデータの破棄が試行されます。

例外を作成する際は、できる限り完全にデータ・フィールドを初期化するコンストラクタを使用するようにしてください。例外を最も簡単に識別するには、OMG IDL 定義を確認します。この定義には、データ・メンバに関する追加の定義が記述されています。

以下では、例外の各メンバ関数について説明します。

```
CORBA::SystemException::SystemException ()
```

CORBA::SystemException クラスのデフォルトのコンストラクタです。マイナー・コードは 0 (ゼロ) に初期化され、完了ステータスは COMPLETED_NO に設定されています。

```
CORBA::SystemException::SystemException (
            const CORBA::SystemException  & Se)
```

CORBA::SystemException クラスのコピー・コンストラクタです。

```
CORBA::SystemException::SystemException(
        CORBA::ULong Minor, CORBA::CompletionStatus Status)
```

CORBA::SystemException クラスのコンストラクタで、マイナー・コードと完了ステータスを設定します。

各引数について次に説明します。

Minor

Exception オブジェクトのマイナー・コード。minor フィールドはインプリメンテーション固有の値で、ORB が例外を

識別するために使用します。BEA Tuxedo の minor フィールドの定義は、orbminor.h ファイルにあります。

Status

Exception オブジェクトの完了ステータス。値は次のとおりです。

```
CORBA::COMPLETED_YES
CORBA::COMPLETED_NO
CORBA::COMPLETED_MAYBE
```

```
CORBA::SystemException::~SystemException ()
```

CORBA::SystemException クラスのデストラクタです。Exception オブジェクトが使用していたメモリを解放します。

```
CORBA::SystemException CORBA::SystemException::operator =
                    const CORBA::SystemException Se)
```

この代入演算子は、ソース例外から例外情報をコピーします。Se 引数には、コピー元の SystemException オブジェクトを指定します。

```
CORBA::CompletionStatus CORBA::SystemException::completed()
```

この例外の完了ステータスを返します。

```
CORBA::SystemException::completed(
                CORBA::CompletionStatus Completed)
```

この例外の完了ステータスを設定します。Completed 引数には、この例外の完了ステータスを指定します。

```
CORBA::ULong CORBA::SystemException::minor()
```

この例外のマイナー・コードを返します。

```
CORBA::SystemException::minor (CORBA::ULong Minor)
```

この例外のマイナー・コードを設定します。minor 引数には、この例外の新しいマイナー・コードを指定します。minor フィールドはインプリメンテーション固有の値で、アプリケーションが例外を識別するために使用します。

```
CORBA::SystemException * CORBA::SystemException::_narrow (
            CORBA::Exception_ptr Exc)
```

指定の例外がシステム例外に限定可能かどうかを判別します。Exc 引数には、限定する例外を指定します。

指定の例外がシステム例外の場合、システム例外へのポインタが返されます。指定の例外がシステム例外ではない場合、0 (ゼロ) が返されます。

```
CORBA::UserException * CORBA::UserException::_narrow(
                        CORBA::Exception_ptr Exc)
```

指定の例外がユーザ例外に限定可能かどうかを判別します。Exc 引数には、限定する例外を指定します。

指定の例外がユーザ例外の場合、ユーザ例外へのポインタが返されます。指定の例外がユーザ例外ではない場合、0 (ゼロ) が返されます。

標準例外

ここでは、ORB に対して定義される標準例外について説明します。標準例外の例外識別子は、インターフェイス仕様に関係なくオペレーション呼び出しの結果として返されます。標準例外は、raises 式には示されません。

標準例外の処理の複雑さを抑制するには、標準例外のセットを制御可能なサイズに抑えます。この制約により、類似する例外を数多く列挙するのではなく、同等のクラス定義の例外だけに限定することができます。

たとえば、動的なメモリ割り当てができないために、さまざまなポイントでオペレーション呼び出しが失敗することがあります。その際、動的なメモリ割り当ての失敗に対応する 1 つの例外が定義されます。つまり、マーシャルまたはマーシャル解除、クライアント、オブジェクト・インプリメンテーション、ネットワーク・パケットの割り当てなど、メモリ割り当ての失敗で例外が発生するさまざまな要因に対応する、複数の異なる例外を列挙するといったことは行いません。各標準例外には、例外のサブカテゴリを指定するマイナー・コードが含まれています。マイナー・コードの値の割り当ては、各 ORB インプリメンテーションで行います。

また、標準例外には completion_status コードも含まれています。このコードは、次のいずれかの値を取ります。

```
CORBA::COMPLETED_YES
```

例外が発生する前にオブジェクト・インプリメンテーションの処理が完了しています。

CORBA::COMPLETED_NO
: 例外が発生する前にオブジェクト・インプリメンテーションが開始されていませんでした。

CORBA::COMPLETED_MAYBE
: インプリメンテーションの完了ステータスが不明です。

例外の定義

次の表に、標準例外の説明を示します。クライアントは、この一覧にはないシステム例外の処理を準備しておく必要があります。これは、将来のバージョンの仕様で標準例外の定義が追加される可能性があるため、およびORB インプリメンテーションで非標準のシステム例外が発生する可能性があるためです。例外の詳細については、「『システム・メッセージ』」を参照してください。

表 14-3 defines the exceptions.

表 14-3 例外の定義

例外	説明
CORBA::UNKNOWN	未知の例外。
CORBA::BAD_PARAM	無効なパラメータが受け渡されました。
CORBA::NO_MEMORY	動的メモリ割り当てが異常終了しました。
CORBA::IMP_LIMIT	インプリメンテーション制限に違反しました。
CORBA::COMM_FAILURE	通信障害。
CORBA::INV_OBJREF	無効なブジェクト・リファレンスです。
CORBA::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。
CORBA::INTERNAL	ORB 内部エラー。

表 14-3 例外の定義 (続き)

例外	説明
CORBA::MARSHAL	パラメータまたは結果のマーシャル時にエラーが発生しました。
CORBA::INITIALIZE	ORB 初期化障害。
CORBA::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。
CORBA::BAD_TYPECODE	タイプ・コードが間違っています。
CORBA::BAD_OPERATION	無効なオペレーションです。
CORBA::NO_RESOURCES	要求を処理するためのリソースが不足しています。
CORBA::NO_RESPONSE	要求に対する応答がまだ利用できません。
CORBA::PERSIST_STORE	永続ストレージの障害。
CORBA::BAD_INV_ORDER	ルーチン呼び出しが順不同です。
CORBA::TRANSIENT	一時障害のため、要求が再度呼び出されます。
CORBA::FREE_MEM	メモリを解放できません。
CORBA::INV_IDENT	無効な識別子構文です。
CORBA::INV_FLAG	無効なフラグが指定されています。
CORBA::INTF_REPOS	インターフェイス・リポジトリへのアクセス時にエラーが発生しました。
CORBA::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。
CORBA::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。
CORBA::DATA_CONVERSION	データ変換エラー。
CORBA::OBJECT_NOT_EXIST	オブジェクトが存在しません。リファレンスを削除してください。

表 14-3 例外の定義 (続き)

例外	説明
CORBA::TRANSACTION_REQUIRED	トランザクションが必要です。
CORBA::TRANSACTION_ROLLEDBACK	トランザクションがロールバックされました。
CORBA::INVALID_TRANSACTION	無効なトランザクションです。

オブジェクトが存在しない場合

削除済みのオブジェクトを呼び出すと、常に CORBA::OBJECT_NOT_EXIST 例外が発生します。この場合、「困難な」障害として報告されます。この例外を受け取った場合は、このオブジェクト・リファレンスのすべてのコピーの削除、およびほかの適切な「最終回復」手順の実行が許可 (推奨) されます。

トランザクションの例外

CORBA::TRANSACTION_REQUIRED 例外は、要求では NULL トランザクション・コンテキストを登録したにもかかわらず、アクティブなトランザクションが要求されたことを示します。

CORBA::TRANSACTION_ROLLEDBACK 例外は、要求に関連付けられたトランザクションがロールバック済みか、またはロールバックとしてマークされていたことを示します。したがって、要求されたオペレーションは実行できなかったか、または実行されていません。これは、トランザクションでこれ以降の計算が無意味になるためです。

CORBA::INVALID_TRANSACTION は、要求が無効なトランザクション・コンテキストを登録したことを示します。たとえば、リソースを登録しようとしたときにエラーが発生した場合に、この例外が発生します。

ExceptionList メンバ関数

ExceptionList メンバ関数を使用すると、Request が呼び出されたときに発生するユーザ定義例外すべての TypeCode のリストを、クライアント・アプリケーションまたはサーバ・アプリケーションで提供できるようになります。Request メンバ関数については、「Request メンバ関数」を参照してください。

ExceptionList メンバ関数の C++ へのマッピングは次のとおりです。

```
class CORBA
{
  class ExceptionList
  {
    public:
      Ulong count ();
      void add(TypeCode_ptr tc);
      void add_consume(TypeCode_ptr tc);
      TypeCode_ptr item(Ulong index);
      Status remove(Ulong index);
   }; // ExceptionList
}// CORBA
```

CORBA::ExceptionList::count

概要 リスト内の現在の項目数を取得します。

C++ バインディング

```
Ulong count ();
```

引数 特にありません。

例外 関数が失敗した場合、例外がスローされます。

説明 このメンバ関数は、リスト内の現在の項目数を取得します。

戻り値 関数が成功した場合、戻り値はリスト内の項目数です。リストを作成したばかりで、ExceptionList オブジェクトを追加していない場合は、0 (ゼロ) が返されます。

CORBA::ExceptionList::add

概要 名前の付いていない項目で ExceptionList オブジェクトを作成します。これは、flags 属性のみを設定したオブジェクトです。

C++ バインディング

```
void add(TypeCode_ptr tc);
```

引数

tc

TypeCode_ptr によって参照されるメモリ位置を定義します。

例外 メンバ関数が失敗した場合、CORBA::NO_MEMORY 例外がスローされます。

説明 このメンバ関数は、名前の付いていない項目で ExceptionList オブジェクトを作成します。これは、flags 属性のみを設定したオブジェクトです。

ExceptionList オブジェクトは動的に拡張するので、そのサイズをアプリケーションでトラッキングする必要はありません。

戻り値 関数が成功した場合、戻り値は新しく作成された ExceptionList オブジェクトへのポインタです。

関連項目

CORBA::ExceptionList::add_consume
CORBA::ExceptionList::count
CORBA::ExceptionList::item
CORBA::ExceptionList::remove

CORBA::ExceptionList::add_consume

概要 ExceptionList オブジェクトを作成します。

C++ バインディング

```
void add_consume(TypeCode_ptr tc);
```

引数

`tc`

想定されるメモリ位置。

例外 メンバ関数が失敗した場合、例外が発生します。

説明 このメンバ関数は、ExceptionList オブジェクトを作成します。

ExceptionList オブジェクトは動的に拡張するので、そのサイズをアプリケーションでトラッキングする必要はありません。

戻り値 関数が成功した場合、戻り値は新しく作成された ExceptionList オブジェクトへのポインタです。

関連項目

```
CORBA::ExceptionList::add
CORBA::ExceptionList::count
CORBA::ExceptionList::item
CORBA::ExceptionList::remove
```

CORBA::ExceptionList::item

概要 渡されたインデックスに基づいて ExceptionList オブジェクトへのポインタを取得します。

C++ バインディング

```
TypeCode_ptr item(ULong index);
```

引数

`index`

ExceptionList オブジェクトへのインデックス。インデックスの基数はゼロです。

例外 関数が失敗した場合、`BAD_PARAM` 例外がスローされます。

説明 このメンバ関数は、渡されたインデックスに基づいて ExceptionList オブジェクトへのポインタを取得します。関数では、ゼロを基数にしたインデックスを使用します。

戻り値 関数が成功した場合、戻り値は ExceptionList オブジェクトへのポインタです。

関連項目

```
CORBA::ExceptionList::add
CORBA::ExceptionList::add_consume
CORBA::ExceptionList::count
CORBA::ExceptionList::remove
```

CORBA::ExceptionList::remove

概要 指定されたインデックスの項目を削除し、関連付けられたメモリをすべて解放してから、リストの残りの項目を順序付けし直します。

C++ バインディング

```
Status remove(ULong index);
```

引数

`Index`

ContextList オブジェクトへのインデックス。インデックスの基数はゼロです。

例外 `ä÷êîÇ™é³îsÇµÇ¾èÍçáÅABAD_PARAM ó·äOÇ™ÉXÉçÅ[ÇŠÇÍÇ‹Ç²ÅB`

説明 このメンバ関数は、指定されたインデックスの項目を削除し、関連付けられたメモリをすべて解放してから、リストの残りの項目を順序付けし直します。

戻り値 特にありません。

関連項目

```
CORBA::ExceptionList::add
CORBA::ExceptionList::add_consume
CORBA::ExceptionList::count
CORBA::ExceptionList::item
```

章

15 サーバ側のマッピング

サーバ側のマッピングでは、C++ で記述されたオブジェクト・インプリメンテーションの移植性の制約を参照します。ここでは、「サーバ」という用語は、メソッド呼び出しで領域間またはマシン間でアドレス指定する状況に、インプリメンテーションを制限するという意味ではありません。ここでのマッピングは、Object Management Group (OMG) インターフェイス定義言語 (IDL) のインターフェイスのあらゆるインプリメンテーションにアドレス指定します。

注記　この章の情報は、Object Management Group (OMG) の「Common Object Request Broker: Architecture and Specification, Revision 2.4.2」(2001 年 2 月) に基づいています。また、この情報は、OMG の許可を得て転載しています。

インターフェイスのインプリメント

C++ でインプリメンテーションを定義するには、有効な C++ の名前で C++ クラスを定義します。インターフェイスのオペレーションごとに、クラスはオペレーションのマッピングされた名前で非静的なメンバ関数を定義します。マッピングされた名前は、OMG IDL の識別子と同じです。

サーバ・アプリケーションのマッピングでは、アプリケーションで提供されたインプリメンテーション・クラスと、インターフェイスに対して生成されたクラスとの間に別の 2 つの関係を指定します。具体的には、継承ベースの

関係とデレゲーション・ベースの関係の両方のサポートがマッピングで必要になります。標準的なアプリケーションでは、これらの関係の一方または両方を使用できます。BEA Tuxedo CORBA では、継承ベースとデレゲーション・ベースの両方の関係をサポートしています。

継承ベースのインターフェイス・インプリメンテーション

継承ベースのインターフェイス・インプリメンテーションの手法では、インプリメンテーション・クラスは、OMG IDL インターフェイス定義に基づいて生成された基本クラスから派生します。生成された基本クラスはスケルトン・クラスと呼ばれ、派生クラスはインプリメンテーション・クラスと呼ばれます。インターフェイスの各オペレーションには、スケルトン・クラスで宣言された対応するメンバ関数があります。生成されたスケルトン・クラスは、インターフェイスの各オペレーションに対応するメンバ関数を備えていますが、プログラマにとってはオペークな部分もあります。メンバ関数のシグニチャは、生成されたクライアント・スタブ・クラスのシグニチャと同じです。

このインターフェイスを継承でインプリメントするには、プログラマはこのスケルトン・クラスから派生させて、OMG IDL インターフェイスでオペレーションごとにインプリメントする必要があります。ほかの基本インターフェイス用のスケルトン・クラスとインプリメンテーションクラスから、エラーやあいまいさをなくして複数のインターフェイスに移植性のあるインプリメンテーションを可能にするには、スケルトンの仮想基本クラスに `Tobj_ServantBase` クラスを、`Tobj_ServantBase` クラスの仮想基本クラスに `PortableServer::ServantBase` を使用する必要があります。インプリメンテーション・クラス、スケルトン・クラス、`Tobj_ServantBase` クラス、および `PortableServer::ServantBase` クラスの継承は、すべてパブリック仮想でなければなりません。

インプリメンテーション・クラスまたはサーバントは、単一の生成されたスケルトン・クラスからのみ直接派生しなければなりません。複数のスケルトンから直接派生すると、`_this()` オペレーションの定義が複数あるため、あいまいエラーが発生します。ただし、CORBA オブジェクトには単一の最終

派生インターフェイスのみがあるので、これは制限ではありません。C++ サーバントが複数のインターフェイス型をサポートしている場合、デレゲーション・ベースのインターフェイス・インプリメンテーションの手法に利用できます。リスト 15-1 に、インターフェイス継承を使用した OMG IDL の例を示します。

コード リスト 15-1 **インターフェイス継承を使用した** OMG IDL

```
// IDL
interface A
{
    short op1() ;
    void op2(in long val) ;
};
```

コード リスト 15-2 **インターフェイス・クラス** A

```
// C++
class  A : public virtual CORBA::Object
 {
    public:
        virtual CORBA::Short op1 ();
        virtual void op2 (CORBA::Long val);
};
```

サーバ側ではスケルトン・クラスが生成されます。このクラスは、インターフェイスの各オペレーションに対応するメンバ関数を備えていますが、プログラマにとってはオペークな部分もあります。

ポータブル・オブジェクト・アダプタ (POA) の場合、スケルトン・クラスの名前は、対応するインターフェイスの完全にスコープ指定された名前の先頭に文字列「POA_」が付きます。また、このクラスは、サーバントの基本クラス Tobj_ServantBase から直接派生します。次に、Tobj_ServantBase の C++ マッピングを示します。

```
// C++
class Tobj_ServantBase
{
    public:
        virtual void activate_object(const char* stroid);
        virtual void deactivate_object (
            const char* stroid,
            TobjS::DeactivateReasonValue reason
        );
}
```

activate_object() および deactivate_object() メンバ関数の詳細については、「Tobj_ServantBase:: activate_object()」および「Tobj_ServantBase::_add_ref()」を参照してください。

リスト 15-3 に、上で示したインターフェイス A のスケルトン・クラスを示します。

コード リスト 15-3 **インターフェイス** A **のスケルトン・クラス**

```
// C++
class POA_A : public Tobj_ServantBase
{
    public:
// ... サーバ側の ORB インプリメンテーション
        // 固有の記述 ...

        virtual CORBA::Short op1 () = 0;
        virtual void op2 (CORBA::Long val) = 0;
        //...
};
```

グローバル・スコープ (Mod::A など) ではなく、モジュール内部でインターフェイス A が定義された場合、そのスケルトン・クラス名は POA_Mod::A になります。これは、サーバ・アプリケーションのスケルトンの宣言および定義と、クライアントから生成された C++ コードとを区別するのに役立ちます。

このインターフェイスを継承でインプリメントするには、このスケルトン・クラスから派生させて、対応する OMG IDL インターフェイスでオペレーションごとにインプリメントする必要があります。リスト 15-4 では、インターフェイス A のインプリメンテーション・クラスの宣言を示します。

コード リスト 15-4 **インターフェイス** A **のインプリメンテーション・クラスの宣言**

```
// C++
class A_impl : public POA_A
{
  public:
       CORBA::Short op1();
       void op2(CORBA::Long val);
       ...
};
```

デレゲーション・ベースのインターフェイス・インプリメンテーション

デレゲーション・ベースのインターフェイス・インプリメンテーションは、CORBA オブジェクトを継承とは別の手法で CORBA オブジェクトをインプリメントします。この手法は、継承のオーバーヘッドが大きすぎたり、使用できなかったりする場合に使用します。たとえば、一部のグローバル・クラスに継承が必要な場合に既存のレガシー・コードを使用すると、継承の侵襲的な性質が原因で、オブジェクトをインプリメントできないことがあります。デレゲーションは、この種の問題を解決するために使用できます。デレゲーションを使用すると、Process-Entity デザイン・パターンにより、オブジェクトをより自然にインプリメントできます。このパターンでは、Process オブジェクトが 1 つまたは複数のエンティティ・オブジェクトにオペレーションを委譲します。

デレゲーション・ベースの手法では、インプリメンテーションはスケルトン・クラスから継承しません。代わりに、インプリメンテーションはアプリケーションの必要に応じてコーディングでき、ラッパー・オブジェクトがそのインプリメンテーションに呼び出しを委譲します。この「ラッパー・オブジェクト」は「*tie*」と呼ばれ、継承手法で使用される同じスケルトン・クラスと共に IDL コンパイラによって生成されます。生成された *tie* クラスは、スケルトンと同様に、関連付けられたインターフェイスの各 OMG IDL オペ

レーションに対応するメソッドを備えていますが、プログラマにとってはオペークな部分もあります。生成された *tie* クラスの名前は、スケルトン・クラスと同様に、クラス名の末尾に文字列 `_tie` が付け加えられます。

`tie` クラスのインスタンスはサーバントで、tie オブジェクトによって委譲される C++ オブジェクトではありません。このサーバントは、`Servant` 引数を必要とするオペレーションに引数として渡されます。また、関連付けられたオブジェクトは、`_this( )` オペレーションにアクセスしたり、直接データ・メンバにアクセスしたりすることはできません。

型セーフな tie クラスは、C++ テンプレートでインプリメントされます。リスト 15-5 のコードでは、前に例示した OMG IDL の Derived インターフェイスから生成された tie クラスを示します。

コード リスト 15-5　Derived **インターフェイスから生成された** tie **クラス**

```
// C++
template <class T>
class POA_A_tie : public POA_A {
public:
    POA_A_tie(T& t)
        : _ptr(&t), _poa(PortableServer::POA::_nil()), _rel(0) {}
    POA_A_tie(T& t, PortableServer::POA_ptr poa)
        : _ptr(&t), _poa(PortableServer::POA::_duplicate(poa)), _rel(0) {}
    POA_A_tie(T* tp, CORBA::Boolean release = 1)
        : _ptr(tp), _poa(PortableServer::POA::_nil()), _rel(release) {}
    POA_A_tie(T* tp, PortableServer::POA_ptr poa, CORBA::Boolean release = 1)
        : _ptr(tp), _poa(PortableServer::POA::_duplicate(poa)), _rel(release) {}
    ~POA_A_tie()
    { CORBA::release(_poa);
    if (_rel) delete _ptr;
    }

    // tie 固有の関数
    T* _tied_object () {return _ptr;}
    void _tied_object(T& obj)
    { if (_rel) delete _ptr;
    _ptr = &obj;
    _rel = 0;
    }
    void _tied_object(T* obj, CORBA::Boolean release = 1)
    { if (_rel) delete _ptr;
    _ptr = obj;
    _rel = release;
    }

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

    // IDL オペレーション ***********************************
    CORBA::Short op1 ()
    {
        return _ptr->op1 ();
    }



    void op2 (CORBA::Long val)
    {
        _ptr->op2 (val);
    }
    // ************************************************
```

```
    // ServantBase オペレーションを無効化
    PortableServer::POA_ptr _default_POA()
    {
        if (!CORBA::is_nil(_poa))
        {
            return _poa;
        }
        else {
#ifdef WIN32
            return ServantBase::_default_POA();
#else
            return PortableServer::ServantBase::_default_POA();
#endif
        }
    }

private:
    T* _ptr;
    PortableServer::POA_ptr _poa;
    CORBA::Boolean _rel;

    // コピーおよび代入は不可
    POA_A_tie (const POA_A_tie<T> &);
    void operator=(const POA_A_tie<T> &);
};
```

このクラス定義は、IDL コンパイラによって生成されたテンプレートです。通常、このテンプレートを使用するには、レガシー・クラスに対するポインタを取得してから、tie クラスをそのポインタでインスタンス化します。たとえば、次のように入力します。

```
Old::Legacy * legacy = new Old::Legacy( oid);
POA_A_tie<Old::Legacy> * A_servant_ptr =
             new POA_A_tie<Old::Legacy>( legacy );
```

この例からわかるように、tie クラスにはインターフェイスの op1 オペレーションと op2 オペレーションが定義されています。これらのオペレーションは、レガシー・クラスには IDL と同じシグニチャのオペレーションがあることを想定しています。この想定どおりの場合、tie クラス・ファイルをそのまま使用でき、正確に委譲できます。ただし、レガシー・クラスのシグニチャが異なっていたり、単一の関数呼び出しを複数回にわたって行わなければならない場合も多くあります。このような場合は、生成されたコードの

op1 と op2 のコードを置き換えてください。通常、各オペレーションのコードでは、tie クラス変数 _ptr を使用してレガシー・クラスを呼び出します。このクラス変数には、レガシー・クラスに対するポインタが格納されます。たとえば、次の行の場合、

```
    CORBA::Short op1 () {return _ptr->op1 (); }
    void op2 (CORBA::Long val) {_ptr->op2 (val); }
```

次のように変更します。

```
CORBA::Short op1 ()
{
    return _ptr->op37 ();
}

void op2 (CORBA::Long val)
{
    CORBA::Long temp;
    temp = val + 15;
    _ptr->lookup(val, temp, 43);
}
```

このテンプレート・クラスのインスタンスは、デレゲーションのタスクを実行します。Derived インターフェイスのオペレーションを提供するクラス型でテンプレートがインスタンス化された場合、POA_Derived_tie クラスは、そのインプリメンテーション・クラスのインスタンスにオペレーションをすべて委譲します。実際のインプリメンテーション・オブジェクトに対するリファレンスまたはポインタは、POA_Derived_tie クラスが作成されるときに適切な tie コンストラクタに渡されます。そこで要求が呼び出されると、tie サーバントはインプリメンテーション・クラスの対応するメソッドを呼び出して要求を委譲します。

tie クラスのテンプレートを使用すると、アプリケーション開発者は、テンプレートの指定のインスタンス化用にテンプレートのオペレーションの一部またはすべてを特化するすることができます。これにより、アプリケーションでは関連付けられたオブジェクト型のレガシー・クラスを使用できるようになります。この場合、関連付けられたオブジェクトのシグニチャは、tie クラスのシグニチャとは異なります。

オペレーションのインプリメント

インプリメンテーション・メンバ関数のシグニチャは、OMG IDL オペレーションのマッピングされたシグニチャです。クライアント側のマッピングとは異なり、OMG の仕様では、サーバ側のマッピングの関数ヘッダには適切な例外指定が含まれています。リスト 15-6 に、この例を示します。

コード リスト 15-6 **例外の指定**

```
// IDL
interface A
{
     exception B {};
     void f() raises(B);
};

// C++
class MyA : public virtual POA_A
{
  public:
     void f();
     ...
};
```

すべてのオペレーションおよび属性が原因で CORBA システム例外は発生するため、オペレーションに raises 句がない場合でも、すべての例外指定で CORBA::SystemException を含める必要があります。

注記　C++ コンパイラには違いがあるため、メソッド・シグニチャで「スロー宣言」を除外することをお勧めします。スローされる例外が宣言済みのメソッドで未宣言の例外がスローされた場合、システムによってはアプリケーション・サーバがクラッシュすることもあります。

メンバ関数内で、「this」ポインタは、クラスで定義されたインプリメンテーション・オブジェクトのデータを参照します。データにアクセスするだけでなく、メンバ関数は同じクラスで定義されたほかのメンバ関数を暗黙的に呼び出すこともできます。リスト 15-7 に、この例を示します。

コード リスト 15-7 **ほかのメンバ関数の呼び出し**

```
// IDL
interface A
{
     void f();
     void g();
};

// C++
class MyA : public virtual POA_A
{
  public:
       void f();
       void g();
  private:
       long x_;
};

void
MyA::f();
{
       x_ = 3;
       g();
}
```

この方法でサーバント・メンバ関数が呼び出されるときは、CORBA オブジェクトのオペレーションのインプリメンテーションとしてではなく、単純に C++ のメンバ関数として呼び出されています。