C++ API リファレンス

Borland VisiBroker® 7.0

Borland®

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

2006 年 5 月 14 日初版発行

著者：Borland Software Corporation

発行：ボーランド株式会社
PDF

目次

第 21 章

第 1 章

Borland VisiBroker の概要

Borland は，CORBA 開発者に向けて，業界最先端の VisiBroker オブジェクトリクエストブローカー（ORB）を活用するために *VisiBroker for Java*，*VisiBroker for C++*，および *VisiBroker for .NET* を提供しています。この 3 つの VisiBroker は CORBA 2.6 仕様の実装です。

VisiBroker の概要

VisiBroker は，CORBA が Java オブジェクトと Java 以外のオブジェクトの間でやり取りする必要がある分散配布で使用されます。幅広いプラットフォーム（ハードウェア，オペレーティングシステム，コンパイラ，および JDK）で使用できます。VisiBroker は，異種環境の分散システムに関連して一般に発生するすべての問題を解決します。

VisiBroker は次のコンポーネントからなります。

- VisiBroker for Java，VisiBroker for C++，および VisiBroker for .NET（業界最先端のオブジェクトリクエストブローカーの 3 つの実装）。
- VisiNaming Service - Interoperable Naming Specification バージョン 1.3 の完全な実装。
- GateKeeper - ファイアウォールの背後の CORBA サーバーとの接続を管理するプロキシサーバー。
- VisiBroker Console - CORBA 環境を簡単に管理できる GUI ツール。
- コモンオブジェクトサービス - VisiNotify（通知サービス仕様の実装），VisiTransact（トランザクションサービス仕様の実装），VisiTelcoLog（Telecom ログサービス仕様の実装），VisiTime（タイムサービス仕様の実装），VisiSecure など。

VisiBroker の機能

VisiBroker には次の機能があります。

- セキュリティと Web 接続性を容易に装備できます。
- J2EE プラットフォームにシームレスに統合できます（CORBA クライアントが EJB に直接アクセスできる）。
- 堅牢なネーミングサービス（VisiNaming）とキャッシュ，永続的ストレージ，および複製によって高可用性を実現します。
- プライマリサーバーにアクセスできない場合に，クライアントをバックアップサーバーに自動的にフェイルオーバーします。
- CORBA サーバークラスタ内で負荷分散を行います。
- OMG CORBA 2.6 仕様に完全に準拠します。
- Borland JBuilder 統合開発環境と統合されます。
- Borland AppServer などの他の Borland 製品と最適に統合されます。

VisiBroker のマニュアル

VisiBroker のマニュアルセットは次のマニュアルで構成されています。

- *Borland VisiBroker インストールガイド* — VisiBroker をネットワークにインストールする方法について説明します。このマニュアルは，Windows または UNIX オペレーティングシステムに精通しているシステム管理者を対象としています。
- *Borland VisiBroker セキュリティガイド* — VisiSecure for VisiBroker for Java および VisiBroker for C++ など，VisiBroker のセキュリティを確保するための Borland のフレームワークについて説明しています。
- *Borland VisiBroker for Java 開発者ガイド* — Java による VisiBroker アプリケーションの開発方法について記載されています。Visibroker ORB の設定と管理，およびプログラミングツールの使用方法について説明します。また, IDL コンパイラ, スマートエージェント，ロケーションサービス，ネーミングサービス，イベントサービス，オブジェクトアクティベーションデーモン (OAD), Quality of Service (QoS), インターフェースリポジトリ，および Web サービスサポートについても説明します。
- *Borland VisiBroker for C++ 開発者ガイド* — C++ による VisiBroker アプリケーションの開発方法について記載されています。Visibroker ORB の設定と管理，およびプログラミングツールの使用方法について説明します。また, IDL コンパイラ, スマートエージェント，ロケーションサービス，ネーミングサービス，イベントサービス，OAD, QoS，プラグイン可能トランスポートインターフェース，RT CORBA 拡張機能，Web サービスサポート，およびインターフェースリポジトリについても説明します。
- *Borland VisiBroker for .NET 開発者ガイド* — .NET 環境による VisiBroker アプリケーションの開発方法について記載されています。
- *Borland VisiBroker for C++ API リファレンス* — VisiBroker for C++ に付属するクラスとインターフェースについて説明します。
- *Borland VisiBroker VisiTime ガイド* — Borland による OMG Time Service 仕様の実装について説明します。
- *Borland VisiBroker VisiNotify ガイド* — Borland による OMG 通知サービス仕様の実装について説明します。通知メッセージフレームワークの主な機能として, 特に Quality of Service (QoS) のプロパティ, フィルタリング, および Publish/Subscribe Adapter（PSA）の使用方法が記載されています。

- *Borland VisiBroker VisiTransact ガイド* — Borland による OMG Object Transaction Service 仕様の実装および Borland Integrated Transaction Service コンポーネントについて説明します。
- *Borland VisiBroker VisiTelcoLog ガイド* — Borland による OMG Telecom Log Service 仕様の実装について説明します。
- *Borland VisiBroker GateKeeper ガイド* — Web ブラウザやファイアウォールによるセキュリティ制約の下で，VisiBroker GateKeeper を使用して，VisiBroker のクライアントがネットワークを介してサーバーとの通信を確立する方法について説明します。

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

重要 Web サイト http://www.borland.com/techpubs には，PDF 版のマニュアルと最新の製品マニュアルがあります。

スタンドアロンヘルプビューアからの VisiBroker オンラインヘルプトピックへのアクセス

製品がインストールされているコンピュータでスタンドアロンのヘルプビューアからオンラインヘルプにアクセスするには，次のいずれかの手順を実行します。

Windows

- ［スタート｜すべてのプログラム｜ Borland VisiBroker ｜ Help Topics］の順に選択します。
- または，コマンドプロンプトを開き，製品のインストールディレクトリの ¥bin ディレクトリに移動し，次のコマンドを入力します。

```
help
```

UNIX コマンドシェルを開き，製品のインストールディレクトリの /bin ディレクトリに移動し，次のコマンドを入力します。

```
help
```

ヒント UNIX システムにインストールするときの指定で，PATH エントリのデフォルトに bin を含まないようにします。カスタムインストールオプションを選択して PATH エントリのデフォルトを変更せず，PATH に現在のディレクトリのエントリがない場合は，./help を使用してヘルプビューアを起動できます。

VisiBroker コンソールからの VisiBroker オンラインヘルプトピックへのアクセス

VisiBroker コンソールから VisiBroker オンラインヘルプトピックにアクセスするには，［Help ｜ Help Topics］を選択します。

［Help］メニューには，オンラインヘルプ内のいくつかの文書へのショートカットもあります。ショートカットの 1 つを選択すると，ヘルプトピックビューアが起動し，［Help］メニューで選択した項目が表示されます。

マニュアルの表記規則

VisiBroker のマニュアルでは，文中の特定の部分を表すために，次の表に示す書体と記号を使用します。

表 1.1　マニュアルの表記規則

表記規則	用途
italic	新規の用語およびマニュアル名に使用されます。
`computer`	ユーザーやアプリケーションが提供する情報，サンプルコマンドライン，およびコードです。
bold computer	本文では，ユーザーが入力する情報を示します。サンプルコードでは，重要なステートメントを強調表示します。
[ ]	省略可能な項目。
...	繰り返しが可能な直前の引数。
|	二者択一の選択。

プラットフォームの表記

VisiBroker マニュアルでは，次の記号を使用してプラットフォーム固有の情報を示します。

表 1.2　プラットフォームの表記

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

Borland サポートへの連絡

ボーランド社は各種のサポートオプションを用意しています。それらにはインターネット上の無償サービスが含まれており，大規模な情報ベースを検索したり，他の Borland 製品ユーザーからの情報を得ることができます。さらに Borland 製品のインストールに関するサポートから有償のコンサルタントレベルのサポートおよび高レベルなアシスタンスに至るまでの複数のカテゴリから，電話サポートの種類を選択できます。

Borland のサポートサービスの詳細や Borland テクニカルサポートへの問い合わせについては，Web サイト `http://support.borland.com` で地域を選択してください。

ボーランド社のサポートへの連絡にあたっては，次の情報を用意してください。

- 名前
- 会社名およびサイト ID
- 電話番号
- ユーザー ID 番号（米国のみ）
- オペレーティングシステムおよびバージョン
- Borland 製品名およびバージョン
- 適用済みのパッチまたはサービスパック
- クライアントの言語とそのバージョン（使用している場合）
- データベースとそのバージョン（使用している場合）

- 発生した問題の詳細な内容と経緯
- 問題を示すログファイル
- 発生したエラーメッセージまたは例外の詳細な内容

オンラインリソース

ネットワーク上の次のサイトから情報を得ることができます。

Web サイト http://www.borland.com/jp/

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

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

http://www.borland.com/products/newsletters

Web サイト

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

特に，次の URL をチェックすることをお勧めします。

- http://www.borland.com/products/downloads/download_visibroker.html（最新の VisiBroker ソフトウェアおよび他のファイル）
- http://www.borland.com/techpubs（マニュアルの更新および PDF）
- http://info.borland.com/devsupport/bdp/faq/（VisiBroker の FAQ）
- http://community.borland.com（英語，開発者向けの弊社 Web ベースニュースマガジン）

Borland ニュースグループ

Borland VisiBroker を対象とした数多くのニュースグループに参加できます。VisiBroker などの Borland 製品のユーザーによるニュースグループへの参加については，http://www.borland.com/newsgroups を参照してください。

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

第 2 章

生成されるインターフェースとクラス

ここでは，VisiBroker for C++ の IDL コンパイラによって生成されるクラス，その使い方，および機能について説明します。

生成されるインターフェースとクラスの概要

VisiBroker の IDL コンパイラは，クライアントアプリケーションやオブジェクトサーバーの開発を容易にするクラスを生成します。生成されるクラスの多くは，次の CORBA クラスとして使用できます。

- スタブクラス
- サーバントクラス
- tie クラス
- var クラス

<Interface_name>

<interface_name> クラスは，特定の IDL インターフェースに対して生成され，クライアントアプリケーションによって使用されます。このクラスは，特定の IDL インターフェースに定義されたメソッドをすべて提供します。クライアントがオブジェクトリファレンスを使用してオブジェクトのメソッドを呼び出した場合，実際にはスタブメソッドが呼び出されます。スタブメソッドにより，クライアントの処理要求がパッケージされ，オブジェクトインプリメンテーションに送信され，その結果が反映されます。このプロセス全体は，クライアントアプリケーションに透過的に行われます。

クライアントがオブジェクトリファレンスを使用してローカルオブジェクトのメソッドを呼び出した場合，スタブメソッドは呼び出されません。

メモ IDL コンパイラによって生成されたスタブクラスの内容は変更しないでください。

<Interface_name>ObjectWrapper

このクラスはローカルインターフェースに適用されません。ローカル以外のインターフェースの場合，idl2cpp コマンドを -obj_wrapper オプションとともに実行すると，型付きオブジェクトラッパーを派生させるのに使用し，ローカル以外のインターフェースに対して生成されます。-obj_wrapper オプションの詳細については，『VisiBroker for C++ 開発者ガイド』の「C++ 対応プログラマツール」を参照してください。

```
static void add(CORBA::ORB_ptr orb,
   CORBA::ObjectFactory factory,
   VISObjectWrapper::Location loc);

static void remove(CORBA::ORB_ptr orb,
      CORBA::ObjectFactory factory,
VISObjectWrapper::Location loc);
```

サーバーアプリケーションから型なしオブジェクトラッパーを 1 つ削除します。

パラメータ	説明
orb	クライアントが使用する ORB。ORB_init メソッドから返されます。
factory	削除するオブジェクトラッパークラスのファクトリメソッド。
loc	削除するオブジェクトラッパーの場所。次のいずれかの値です。 VISObjectWrapper::Client, VISObjectWrapper::Server VISObjectWrapper::Both

POA<class_name>

POA<class_name> クラスは，IDL コンパイラによって生成される抽象ベースクラスであり，オブジェクトインプリメンテーションクラスの派生に使用されます。オブジェクトインプリメンテーションは，通常，サーバントクラスから派生し，サーバントクラスは，クライアントの処理要求を受信したり，解釈するために必要なメソッドを提供します。

tie<class_name>

tie<class_name> クラスは，IDL コンパイラによって生成され，デリゲーションインプリメンテーションの作成に利用されます。tie クラスを使用すると，すべての処理要求を別のオブジェクトにデリゲートするオブジェクトインプリメンテーションを作成できます。これにより，CORBA::Object クラスを継承しないで既存のオブジェクトを使用できます。

<class_name>_var

<class_name>_var クラスは，IDL インターフェースに対して生成され，簡略化されたメモリ管理セマンティクスを提供します。

第 3 章

コアインターフェースとクラス

ここでは，VisiBroker for C++ のコアインターフェースとクラスについて説明します。

PortableServer::AdapterActivator

アダプタアクティベータは，POA（ポータブルオブジェクトアダプタ）に関連付けられています。これにより，POA は子 POA を次の条件で作成できます。

- オンデマンド。
- 子 POA（またはその 1 つ）が指定された要求を受信したときの副作用として。
- アクティブパラメータが `TRUE` に設定されて `find_POA` メソッドが呼び出された場合。

POA の詳細については，『VisiBroker for C++ 開発者ガイド』の「POA の使い方」を参照してください。

IDL 定義

```
interface AdapterActivator {
     boolean unknown_adapter(in POA parent, in string name);
};
```

PortableServer::AdapterActivator のメソッド

```
CORBA::Boolean unknown_adapter(
   PortableServer::POA_ptr parent, const char* name);
```

存在しないターゲット POA を識別するオブジェクトリファレンスに対する要求を VisiBroker ORB が受信すると呼び出されます。VisiBroker ORB は，ターゲット POA を作成するために必要な POA があれば，ルート POA に最も近い上位の POA から順に一度ずつこのメソッドを呼び出します。

パラメータ	説明
parent	このメソッドが呼び出されるアダプタアクティベータに関連付けられている親 POA。
name	この親に関して作成される POA の名前。

BindOptions

VisiBroker 4.x で使用されなくなりました。

```
struct BindOptions
```

この構造体を使用して，_bind メソッドのオプションを指定します。詳細については，18 ページの「Object」を参照してください。各プロセスはグローバルな BindOptions 構造体を 1 つ持ち，バインドオプションが指定されないすべての _bind 呼び出しに使用されます。Object::_default_bind_options メソッドを使用して，デフォルトのバインドオプションを変更できます。

バインドオプションを特定のオブジェクトに対して設定することもでき，オブジェクトとの接続が継続している間有効です。

インクルードファイル

この構造体を使用するには，corba.h ファイルをインクルードする必要があります。

BindOptions のメンバー

```
CORBA::Boolean defer_bind;
```

TRUE に設定した場合は，最初のクライアントオペレーションが発行されるまで，クライアントとオブジェクトインプリメンテーションの間の接続の確立が遅延されます。

FALSE に設定した場合，_bind メソッドはただちに接続を確立します。

```
CORBA::Boolean enable_rebind;
```

TRUE に設定した場合，ネットワーク障害などのエラーによって接続が失われると，VisiBroker ORB は適切なオブジェクトインプリメンテーションとの接続を再び確立しようとします。

FALSE に設定した場合，クライアントをオブジェクトインプリメンテーションに再接続する試みは行われません。

```
CORBA::Long max_bind_tries;
```

OAD がビジーな場合に，バインド要求を再試行する回数を指定します。

```
CORBA::ULong send_timeout;
```

クライアントが処理要求の送信を待機してブロックする最大時間を秒単位で指定します。要求がタイムアウトになると CORBA::NO_RESPONSE 例外が生成され，サーバーとの接続は廃棄されます。

デフォルト値 0 は，クライアントが無制限にブロックすることを示します。

```
CORBA::ULong receive_timeout;
```

クライアントが処理要求に対する応答を待機してブロックする最大時間を秒単位で指定します。要求がタイムアウトになると a CORBA::NO_RESPONSE 例外が生成され，サーバーとの接続は廃棄されます。

デフォルト値 0 は，クライアントが無制限にブロックすることを示します。

```
CORBA::ULong connection_timeout;
```

クライアントが接続を待機する最大時間を秒単位で指定します。指定した時間がすぎた場合は，CORBA::NO_IMPLEMENT 例外が生成されます。

デフォルト値 0 は，接続に対するデフォルトのシステムタイムアウトの使用を指定します。

BOA

VisiBroker 4.0 で使用されなくなりました。

```
class BOA
```

BOA クラスは基本オブジェクトアダプタ（Basic Object Adaptor）を表し，オブジェクトとオブジェクトリファレンスを作成および操作するためのメソッドを提供します。オブジェクトサーバーは，オブジェクトインプリメンテーションをアクティブ化または非アクティブ化したり，使用するスレッドポリシーを指定するために BOA を使用します。

BOA オブジェクトはインスタンス化しないでください。かわりに ORB::BOA_init メソッドを呼び出して，BOA オブジェクトまでのリファレンスを取得してください。

Borland VisiBroker は，CORBA BOA 仕様に対する拡張機能を提供します。詳細については，11 ページの「CORBA::BOA のメソッド」を参照してください。このメソッドは，接続，スレッド，およびサービスのアクティブ化の管理を提供します。

インクルードファイル

この構造体を使用する場合は，**corba.h** ファイルをインクルードする必要があります。

CORBA::BOA のメソッド

```
CORBA::Object_ptr create(const CORBA::ReferenceData& refData,
  extension::CreationImplDef& creationImplDef)
```

指定されたインプリメンテーションを OAD に登録します。

パラメータ	説明
refData	このパラメータは使用されませんが，CORBA 仕様との互換性のために提供されています。
creationImplDef	このポインタの正しい型は CreationImplDef です。このパラメータは，実行可能プログラムのインターフェース名，オブジェクト名，パス名と，アクティブ化ポリシーなどのパラメータを提供します。CreationImplDef クラスの詳細については，111 ページの「アクティベーションのインターフェースとクラス」を参照してください。

void **deactivate_impl**(extension::ImplementationDef_ptr **implDefPtr**)

インプリメンテーションへの要求を破棄します。

このメソッドは，implDefPtr で指定されたインプリメンテーションを非アクティブ化します。このメソッドが呼び出されると，オブジェクトとインプリメンテーションが再びアクティブ化されるまで，クライアントの要求はインプリメンテーション内のオブジェクトに送信されなくなります。インプリメンテーションが再び要求を受け付けるようにするには，impl_is_ready または obj_is_ready メソッドを呼び出す必要があります。

パラメータ	説明
implDefPtr	このポインタの正しい型は CreationImplDef です。このパラメータは，実行可能プログラムのインターフェース名，オブジェクト名，パス名，アクティブ化ポリシーなどのパラメータを提供します。

void **deactivate_obj**(CORBA::Object_ptr **objPtr**)

BOA が指定オブジェクトを非アクティブ化するように要求します。このメソッドが呼び出されると，obj_is_ready または impl_is_ready が呼び出されるまで，BOA は要求をそのオブジェクトに送信しません。

パラメータ	説明
objPtr	非アクティブ化するオブジェクトへのポインタ。

void **dispose**(CORBA::Object_ptr objPtr)

指定されたオブジェクトのインプリメンテーションをオブジェクトアクティベーションデーモンから登録解除します。このメソッドが呼び出されると，指定されたオブジェクトへのすべての参照は無効になり，このオブジェクトインプリメンテーションとの接続はすべて切断されます。オブジェクトにメモリが割り当てられている場合，アプリケーションがそのオブジェクトを削除する必要があります。

メモ このメソッドは VisiBroker 4.x で使用されなくなりました。かわりに OAD のインターフェースを使用してください。

パラメータ	説明
objPtr	登録を解除するオブジェクトへのポインタ。

static CORBA::BOA_ptr **_duplicate**(CORBA::BOA_ptr **ptr**)

この static メソッドは，パラメータとして渡された BOA ポインタをコピーします。

パラメータ	説明
ptr	BOA ポインタ。

void **exit_impl_ready**()

このメソッドは，VisiBroker for C++ の前バージョンとの下位互換性を提供します。このメソッドは BOA::shutdown メソッドを呼び出し，これにより，前の impl_is_ready メソッドの呼び出しが戻ります。詳細については，14 ページの「void shutdown()」を参照してください。このメソッドをアクティブな要求のコンテキストで呼び出すことはできません。

CORBA::ReferenceData_ptr **get_id**(CORBA::Object_ptr **objPtr**)

指定されたオブジェクトのリファレンスデータを取得します。リファレンスデータは，アクティブ化の際にオブジェクトインプリメンテーションによって設定され，オブジェクトの存続期間中は一定であることが保証されます。

パラメータ	説明
objPtr	リファレンスデータを返すオブジェクトへのポインタ。

```
CORBA::Principal_ptr get_principal(CORBA::Object_ptr objPtr,
  CORBA::Environment_ptr env=NULL)
```

指定オブジェクトに関連付けられている Principal オブジェクトを返します。このメソッドは，クライアント要求の処理中に，オブジェクトインプリメンテーションによってのみ呼び出されます。

パラメータ	説明
objPtr	インプリメンテーションを変更するオブジェクトへのポインタ。
env	この Principal に関連付けられている Environment オブジェクトへのポインタ。

```
void impl_is_ready(const char *service_name, extension::Activator_ptr activator,
   CORBA::Boolean block = 1)
```

クライアントがサービスを要求するまで，指定された service_name に関連付けられているオブジェクトインプリメンテーションのアクティブ化を遅延するように，BOA に指示します。クライアントがそのサービスを要求すると，指定された Activator オブジェクトを使用して，オブジェクトインプリメンテーションがアクティブ化されます。block が 0 に設定された場合，このメソッドは exit_impl_ready メソッドが呼び出されるまで呼び出し側をブロックします。

パラメータ	説明
service_name	指定された Activator オブジェクトに関連付けられているサービス名。
activator	オブジェクトインプリメンテーションのアクティブ化に使用される Activator。
block	1 に設定した場合，このメソッドは呼び出し側をブロックします。0 に設定した場合はブロックしません。デフォルトではブロックします。

```
void impl_is_ready(extension::ImplementationDef_ptr impl=NULL)
```

サーバー内の 1 つ以上のオブジェクトがサービス要求を受信する準備を完了していることを BOA に通知します。このメソッドは，exit_impl_ready が呼び出されるまで呼び出し側をブロックします。インプリメンテーションが提供しているすべてのオブジェクトが C++ のインスタンス化を介して作成されており，obj_is_ready メソッドを使用してアクティブ化されている場合は，ImplementationDef_ptr を指定することはできません。

オブジェクトインプリメンテーションは，オブジェクトを 1 つしか提供しないことがあり，クライアントの要求を受信するまで，そのオブジェクトのアクティブ化を遅らせることもあります。そのような場合，オブジェクトインプリメンテーションは，最初に obj_is_ready メソッドを呼び出す必要はありません。そのかわり，このメソッドだけを呼び出し，単一のオブジェクトとして ActivationImplDef ポインタを渡します。

パラメータ	説明
impl	このポインタの正しい型は ActivationImplDef です。このパラメータは，実行可能プログラムのインターフェース名，オブジェクト名，パス名，アクティブ化ポリシーなどのパラメータを提供します。

```
static CORBA::BOA_ptr _nil()
```

この static メソッドは，初期化に使用される NULL BOA ポインタを返します。

void **obj_is_ready**(CORBA::Object_ptr **obj**,
  extension::ImplementationDef_ptr **impl_ptr** = NULL)

指定されたオブジェクトがクライアントから使用される準備を完了していることを BOA に通知します。このメソッドの使い方には，2 とおりあります。

- C++ インスタンス化を使用して作成されたオブジェクトでは，そのオブジェクトへのポインタだけを指定し，ImplementationDef_ptr にはデフォルトの NULL を設定する必要があります。
- 最初のクライアント要求を受信するまで作成を遅延するオブジェクトでは Object_ptr に NULL を指定し，初期化済みの ActivationImplDef オブジェクトへのポインタを提供する必要があります。

パラメータ	説明
obj	アクティブ化するオブジェクトへのポインタ。
impl_ptr	ActivationImplDef オブジェクトへのポインタ。省略可能です。

static void **CORBA::release**(CORBA::BOA_ptr **boa**)

この static メソッドは，指定された BOA ポインタを解放します。オブジェクトの参照カウントが 0 になると，そのオブジェクトは自動的に削除されます。

パラメータ	説明
boa	有効な BOA ポインタ。

static RegistrationScope **scope**()

この static メソッドは，BOA の登録スコープを返します。オブジェクトの有効な登録スコープは，SCOPE_GLOBAL または SCOPE_LOCAL です。グローバルスコープを持つオブジェクトだけが osagent に登録されます。

static void **scope**(RegistrationScope **val**)

この static メソッドは，BOA の登録スコープを指定された値に変更します。

パラメータ	説明
val	この BOA のスコープ。LOCAL_SCOPE（永続的オブジェクト）または GLOBAL_SCOPE（スマートエージェントに登録されるオブジェクト）のいずれかの値になります。

void **shutdown**()

前の impl_is_ready メソッドの呼び出しから戻ります。このメソッドをアクティブな要求のコンテキストで呼び出すことはできません。

CORBA::Object_ptr **string_to_object**(const char * **str**)

文字列化されたオブジェクトリファレンスを変換してオブジェクトリファレンスに戻します。文字列化されたオブジェクトリファレンスは，object_to_string メソッドを使用して作成されます。26 ページの「char *object_to_string(CORBA::Object_ptr obj);」を参照してください。戻されたオブジェクトリファレンスは，そのオブジェクトのメソッドを呼び出すために使用されます。

パラメータ	説明
str	元のオブジェクトリファレンスに変換する文字列。

CORBA::BOA の VisiBroker 拡張機能

CORBA::ULong **connection_max**()

許容される最大接続数を返します。

void **connection_max**(CORBA::ULong **max_conn**)

利用できる最大接続数を設定するためにサーバーによって使用されます。

パラメータ	説明
max_conn	許容する最大接続数。

CORBA::ULong **thread_max**()

TPool スレッドポリシーが選択されている場合に，利用できる最大スレッド数を返します。

void **thread_max**(CORBA::ULong **max**)

TPool スレッドポリシーが選択されている場合，利用できる最大スレッド数を設定します。現在のスレッド数が this number を超えている場合，スレッド数が max に達するまで余分なスレッドは使用されなくなり次第 1 つずつ廃棄されます。

パラメータ	説明
max	許容する最大スレッド数。

CORBA::ULong **thread_stack_size**()

スレッドポリシーとして TPool または TSession が選択されている場合のスレッドの最大スタックサイズをバイト単位で返します。

void **thread_stack_size**(CORBA::ULong **size**)

スレッドポリシーとして TPool または TSession が選択されている場合のスレッドの最大スタックサイズをバイト単位で設定します。

パラメータ	説明
size	新しく設定するスタックサイズ。

CompletionStatus

enum **CompletionStatus**

この列挙体は，処理要求の完了状態を表します。

IDL 定義

```
enum CompletionStatus {
   COMPLETED_YES;
   COMPLETED_NO;
   COMPLETED_MAYBE;};
```

CompletionStatus のメンバー

COMPLETED_YES = 0	処理要求が正常に完了したことを示します。
COMPLETED_NO = 1	何らかの種類の例外またはエラーが原因で，処理要求が完了しなかったことを示します。
COMPLETED_MAYBE = 2	例外またはエラーが発生したが，処理要求が完了していることを示します。

Context

class **CORBA::Context**

Context クラスはクライアントアプリケーションの環境に関する情報が含まれ，静的または動的なメソッドの呼び出しにおいて暗黙的なパラメータとしてサーバーに渡されます。このクラスは，要求に関連して必要となる特殊な情報の伝達に使用されますが，メソッドの引数リストには示されません。

Context クラスは，名前／値の組として格納されるプロパティのリストからなり，それらのプロパティを設定および操作するためのメソッドを提供します。Context には NVList オブジェクトが含まれており，名前／値の組をチェイン化します。

Context_var クラスも使用可能です。このクラスは，より簡単なメモリ管理セマンティクスを提供します。

インクルードファイル

この構造体を使用する場合は，**corba.h** ファイルをインクルードする必要があります。

Context のメソッド

const char ***context_name()** const;

このコンテキストの識別に使用される名前を返します。このオブジェクトの作成時に名前が指定されなかった場合は，NULL 値が返されます。

void **create_child**(const char * **name**, **CORBA::Context_out** context_ptr);

このオブジェクトの子 Context を作成します。

パラメータ	説明
name	新規 Context オブジェクトの名前。
context_ptr&	新しく作成された子 Context へのリファレンス。

void **delete_values**(const char ***name**);

このオブジェクトから 1 つ以上のプロパティを削除します。

パラメータ	説明
name	削除する 1 つ以上のプロパティの名前。名前の末尾に「*」ワイルドカード文字を付けると，一致するプロパティをすべて削除できます。すべてのプロパティを削除するには，1 つのアスタリスクだけを指定します。

static CORBA::Context_ptr **_duplicate**(CORBA::Context_ptr **ctx**);

指定されたオブジェクトをコピーします。

パラメータ	説明
ctx	コピーするオブジェクト。

```
void get_values(const char *start_scope, CORBA::Flags flag, const char *name,
CORBA::NVList_out NVList_ptr);
```

Context オブジェクトの階層を検索し，name パラメータで指定された 1 つ以上の名前／値の組を取得します。name パラメータの末尾にワイルドカード文字（*）を付けると，一致するプロパティとその値がすべて返されます。次に NVList オブジェクトを作成し，名前／値の組を NVList に配置します。name パラメータが空の文字列または NULL 文字列である場合は，標準のシステム例外 BAD_PARAM が生成されます。name パラメータが見つからなかった場合は，標準のシステム例外 BAD_CONTEXT が生成され，プロパティリストは返されません。

start_scope パラメータには，検索を開始する位置のコンテキストの名前を指定します。そのプロパティが見つからなかった場合は，一致するものが見つかるか，検索する Context オブジェクトがなくなるまで，Context オブジェクトの階層を上方に検索を続けます。start_scope パラメータを省略すると，指定されたコンテキストオブジェクトから検索が開始されます。

パラメータ	説明
start_scope	検索を開始する位置の Context オブジェクトの名前。CORBA::Context::_nil() に設定した場合，検索は現在のコンテキストから開始されます。検索範囲を現在のコンテキストに限定するには，CORBA::CTX_RESTRICT_SCOPE を指定します。
flag	一致するコンテキスト名が見つからなかった場合は，例外が生成されます。
name	検索対象のプロパティ名。末尾に「*」ワイルドカード文字を使用すると，name に一致するすべてのプロパティを取得できます。
NVList_ptr	見つかったプロパティのリストへの参照。

```
static CORBA::Context_ptr _nil();
```

初期化の目的に合った NULL Context_ptr を返します。

```
CORBA::Context_ptr parent();
```

親 Context へのポインタを返します。親 Context が見つからなかった場合は，NULL 値が返されます。

```
static void _release(CORBA::Context_ptr ctx);
```

この static メソッドは，指定された Context オブジェクトを解放します。オブジェクトの参照カウントが 0 になると，そのオブジェクトは自動的に削除されます。

パラメータ	説明
ctx	解放するオブジェクト。

```
void set_one_value(const char *name, const CORBA::Any& anAny);
```

指定された名前と値を使用して，このオブジェクトにプロパティを追加します。

パラメータ	説明
name	プロパティの名前。
anAny	プロパティの値。

```
void set_values(CORBA::NVList_ptr _list);
```

NVList で指定された名前／値の組を使用して，このオブジェクトに 1 つ以上のプロパティを追加します。NVList オブジェクトを作成してこのメソッドの入力パラメータとして使用する場合は，Flags フィールドを 0 に設定する必要があります。また，NVList に追加する各 Any オブジェクトは，それぞれの TypeCode が TC_string に設定されている必要があります。NVList クラスの詳細については，第 4 章「動的なインターフェースとクラス」を参照してください。

パラメータ	説明
_list	このオブジェクトに追加する名前／値の組のリスト。

PortableServer::Current

```
class PortableServer::Current : public CORBA::Current
```

このクラスは，あるオブジェクトのメソッドが呼び出されたとき，そのオブジェクト自身にアクセスするためのメソッドを提供します。Current クラスは，複数のオブジェクトを実装するが，POA によってディスパッチされる任意のサーバントに対するメソッド呼び出しのコンテキスト内で使用できるサーバントをサポートします。

IDL 定義

```
interface Current : CORBA::Current {
   exception NoContext {};
   POA get_POA() raises (NoContext);
};
```

PortableServer::Current のメソッド

```
PortableServer::POA *get_POA();
```

このオブジェクトが呼び出されたコンテキスト内でこのオブジェクトを実装する POA への参照を返します。POA によって送られたメソッド呼び出しではない場合は，NoContext 例外が生成されます。

```
PortableServer::ObjectId get_object_id();
```

このオブジェクトが呼び出されたコンテキスト内でこのオブジェクトを識別する ObjectId を返します。POA によって送られたメソッド呼び出しではない場合は，NoContext 例外が生成されます。

Exception

```
class CORBA::Exception
```

Exception クラスは，システム例外クラスとユーザー例外クラスのベースクラスです。詳細については，44 ページの「SystemException」を参照してください。

インクルードファイル

この構造体を使用する場合は，**corba.h** ファイルをインクルードする必要があります。

Object

```
class CORBA::Object
```

すべての ORB オブジェクトは，Object クラスから派生します。このクラスは，オブジェクトの状態を照会および設定するメソッドのほか，クライアントをオブジェクトにバインドしたり，オブジェクトリファレンスを操作するためのメソッドも提供します。Object クラスメソッドは ORB によって実装されます。

VisiBroker for C++ は，CORBA Object 仕様に対する拡張機能を提供します。この拡張機能の詳細については，21 ページの「CORBA::Object の VisiBroker 拡張機能」を参照してください。

インクルードファイル

この構造体を使用する場合は，**corba.h** ファイルをインクルードする必要があります。

CORBA::Object のメソッド

```
void _create_request(CORBA::Context_ptr ctx, const char *operation,
CORBA::NVList_ptr arg_list, CORBA::NamedValue_ptr result, CORBA::Request_out
request, CORBA::Flags req_flags);
```

動的起動インターフェースを使った呼び出しに向けて，オブジェクトインプリメンテーションに対する Request を作成します。

パラメータ	説明
ctx	この要求に関連付けられているコンテキスト。詳細については，15 ページの「CompletionStatus」を参照してください。
operation	オブジェクトインプリメンテーションで実行するオペレーションの名前。
arg_list	オブジェクトインプリメンテーションに渡す引数のリスト。詳細については，「動的なインターフェースとクラス」の第 4 章「動的なインターフェースと クラス」を参照してください。
result	オペレーションの結果。詳細については，「動的なインターフェースとクラス」の第 4 章「動的なインターフェースと クラス」を参照してください。
request	作成された Request へのポインタ。詳細については，「動的なインターフェースとクラス」の第 4 章「動的なインターフェースと クラス」を参照してください。
req_flags	req_flags パラメータに OUT_LIST_MEMORY フラグや IN_COPY_VALUE フラグを設定することはできますが，引数の挿入や抽出は Any 型を介して実行されるため，これは意味を持たず，無視されます。

```
void _create_request(CORBA::Context_ptr ctx, const char *operation, CORBA::NVList_ptr
arg_list,
   CORBA::NamedValue_ptr result,
   CORBA::ExceptionList_ptr eList,
   CORBA::ContextList_ptr ctxList,
   CORBA::Request_out request, CORBA::Flags req_flags);
```

動的起動インターフェースを使った呼び出しに向けて，オブジェクトインプリメンテーションに対する Request を作成します。

パラメータ	説明
ctx	この要求に関連付けられているコンテキスト。詳細については，15 ページの「CompletionStatus」を参照してください。
operation	オブジェクトインプリメンテーションで実行するオペレーションの名前。
arg_list	オブジェクトインプリメンテーションに渡す引数のリスト。詳細については，「動的なインターフェースとクラス」の第 4 章「動的なインターフェースと クラス」を参照してください。

パラメータ	説明
result	オペレーションの結果。詳細については，「動的なインターフェースとクラス」の第 4 章「動的なインターフェースと クラス」を参照してください。
eList	この要求の例外のリスト。
ctxList	この要求の Context オブジェクトのリスト。
request	作成された Request へのポインタ。詳細については，「動的なインターフェースとクラス」の第 4 章「動的なインターフェースと クラス」を参照してください。
req_flags	arg_list 内に出力引数の NamedValue 項目がある場合は，このフラグを OUT_LIST_MEMORY に設定する必要があります。

```
static CORBA::Object_ptr _duplicate(CORBA::Object_ptr obj);
```

この static メソッドは，指定された Object_ptr をコピーし，そのオブジェクトへのポインタを返します。オブジェクトの参照カウントは，1 だけインクリメントされます。

パラメータ	説明
obj	複製するオブジェクトポインタ。

```
CORBA::InterfaceDef_ptr _get_interface();
```

このオブジェクトのインターフェース定義へのポインタを返します。詳細については，「インターフェースリポジトリのインターフェースとクラス」の第 5 章「インターフェースリポジトリのインターフェースとクラス」を参照してください。

```
CORBA::ULong _hash(CORBA::ULong maximum);
```

このオブジェクトのハッシュ値を返します。この値は，このオブジェクトの存続期間中は変化しませんが，必ずしも一意の値ではありません。2 つのオブジェクトが異なるハッシュ値を返した場合，それらのオブジェクトは同じではありません。ハッシュ値の上限を指定できます。 下限は 0 です。

パラメータ	説明
maximum	取得するハッシュ値の上限。

```
CORBA::Boolean _is_a(const char *logical_type_id);
```

このオブジェクトが，指定されたリポジトリ ID に関連付けられているインターフェースを実装している場合は，TRUE を返します。そうでない場合は，False を返します。

パラメータ	説明
logical_type_id	チェックするリポジトリ識別子。

```
CORBA::Boolean _is_equivalent(CORBA::Object_ptr other_object);
```

指定されたオブジェクトポインタとこのオブジェクトが同じオブジェクトインプリメンテーションをポイントしている場合は，TRUE を返します。そうでない場合は，False を返します。

パラメータ	説明
other_object	このオブジェクトと比較するオブジェクトへのポインタ。

```
static CORBA::Object_ptr _nil();
```

この static メソッドは，初期化の目的に合った NULL ポインタを返します。

```
CORBA::Boolean _non_existent();
```

このオブジェクトリファレンスによって表されるオブジェクトがすでに存在しない場合は，TRUE を返します。

```
CORBA::Request_ptr _request(const char* operation);
```

このオブジェクトのメソッドを呼び出すための適切な Request を作成します。作成された Request オブジェクトを返します。詳細については，「動的なインターフェースとクラス」の第 4 章「動的なインターフェースと クラス」を参照してください。

パラメータ	説明
operation	呼び出すメソッドの名前。

```
CORBA::Object_ptr _resolve_reference(const char* id);
```

指定されたサービス識別子を使用してサーバー側インターフェースを解決します。このメソッドは，このオブジェクトリファレンスを使用してクライアントアプリケーションから呼び出されます。指定のサービスを解決するために ORB::_resolve_initial_references メソッドをサーバー側で呼び出します。. オブジェクトリファレンスが返されるので，クライアントはそれを適切なサーバーの型にナローイングします。

このメソッドは，通常，クライアントアプリケーションがサーバーの属性を管理する場合に使用されます。

パラメータ	説明
id	サーバー側で解決されるインターフェースの名前。

CORBA::Object の VisiBroker 拡張機能

```
CORBA::BindOptions* _bind_options();
```

このオブジェクトのためにだけ使用されるバインドオプションへのポインタを返します。詳細については，10 ページの「BindOptions」を参照してください。

```
void _bind_options(const CORBA::BindOptions& opt);
```

このオブジェクトのバインドオプションを設定します。設定したオプションは，プロキシオブジェクトの存続期間中は有効なままです。タイムアウト値に加えた変更は，リバインド処理だけでなく，それ以降のすべての送信および受信オペレーションにも適用されます。詳細については，10 ページの「BindOptions」を参照してください。

パラメータ	説明
opt	このオブジェクトの新しいバインドオプション。

```
static CORBA::Object_ptr _bind_to_object(
    const char *rep_id,
    const char *object_name=NULL,
    const char *host_name=NULL,
    const CORBA::BindOptions *options=NULL,
    CORBA::ORB_ptr orb=NULL);
```

指定された BindOptions と ORB を使用して，指定されたホストの指定された repository_id と object_name を持つオブジェクトにバインドを試みます。

パラメータ	説明
rep_id	目的のオブジェクトのリポジトリ ID。
object_name	目的のオブジェクトの名前。
host_name	オブジェクトインプリメンテーションが実行されている目的のホストの名前。

パラメータ	説明
options	この接続のバインドオプション。詳細については，10 ページの「BindOptions」を参照してください。
orb	使用する ORB。

```
CORBA::BOA _boa() const;
```

このオブジェクトが登録されている基本オブジェクトアダプタへのポインタを返します。

メモ 次のメソッドは VisiBroker 4.0 で使用されなくなりました。

```
static CORBA::Object_ptr _clone(CORBA::Object_ptr obj,
  CORBA::Boolean reset_connection = 1UL);
```

指定されたオブジェクトリファレンスをコピーします。

パラメータ	説明
obj	複製するオブジェクトリファレンス。
reset_connection	このパラメータは使用されません。

```
static const CORBA::BindOptions * _default_bind_options();
```

クライアントプロセスごとのグローバルな BindOptions へのポインタを返します。詳細については，10 ページの「BindOptions」を参照してください。

```
static void _default_bind_options(const CORBA::BindOptions& bindOptions);
```

これは，別にバインドオプションが指定されていないすべての _bind 呼び出しに対して使用されます。詳細については，10 ページの「BindOptions」を参照してください。

```
static const CORBA::TypeInfo *_desc();
```

このオブジェクトの型情報を返します。

```
const char *_interface_name() const;
```

このオブジェクトのインターフェース名を返します。

```
CORBA::Boolean _is_bound() const;
```

クライアントプロセスがオブジェクト実装との接続を確立した（バインドされている）場合，このメソッドは 1 を戻します。オブジェクトがバインドされていない場合は，0 を戻します。

```
CORBA::Boolean _is_local() const;
```

オブジェクトインプリメンテーションがクライアントアプリケーションと同じプロセスまたはアドレス空間内にある場合は，TRUE を返します。

```
CORBA::Boolean _is_persistent() const;
```

このオブジェクトが永続的オブジェクトの場合は，TRUE を返します。FALSE 値が返されても，オブジェクトが永続的ではないとは限りません。

```
CORBA::Boolean _is_remote() const;
```

オブジェクトインプリメンテーションがクライアントアプリケーションと異なるプロセスまたはアドレス空間内にある場合は，TRUE を返します。クライアントとオブジェクトインプリメンテーションが同じホスト上にあるかどうかには関係ありません。

```
const char *_object_name() const;
```

このオブジェクトに関連付けられているオブジェクト名を返します。

```
CORBA::Long _ref_count() const;
```

このオブジェクトの参照カウントを返します。

```
void _release();
```

このオブジェクトの参照カウントをデクリメントし，参照カウントが 0 になった場合は，オブジェクトを解放します。

```
const char *_repository_id() const;
```

このオブジェクトのリポジトリ ID を返します。

ORB

```
class CORBA::ORB
```

ORB クラスは，インターフェースをオブジェクトリクエストブローカーに提供します。このクラスは，特定の Object またはオブジェクトアダプタとは独立して，クライアントオブジェクトにメソッドを提供します。

Borland VisiBroker は，CORBA ORB に対する拡張機能を提供します。詳細については，28 ページの「CORBA::ORB に対する VisiBroker 拡張機能」を参照してください。このメソッドは，接続，スレッド，およびサービスのアクティブ化の管理を目的として提供されています。

インクルードファイル

この構造体を使用する場合は，**corba.h** ファイルをインクルードする必要があります。

CORBA::ORB のメソッド

```
CORBA::Boolean work_pending();
```

ORB が処理待ちの作業を持つ場合は，true を返します。

```
static CORBA::TypeCode_ptr create_alias_tc(
   const char *repository_id,
   const char *type_name,
   CORBA::TypeCode_ptr original_type);
```

この static メソッドは，指定された型と名前を持つエリアスに対する TypeCode を動的に作成します。

パラメータ	説明
repository_id	IDL コンパイラによって生成されたか，動的に構築された識別子。
type_name	エリアスの型の名前。
original_type	作成するエリアスに対する元の型。

```
static CORBA::TypeCode_ptr create_array_tc(CORBA::Ulong length, TypeCode_ptr element_type);
```

この static メソッドは，配列の TypeCode を動的に作成します 。

パラメータ	説明
length	配列要素の最大数。
element_type	この配列に保存される要素の型。

```
static CORBA::TypeCode_ptr create_enum_tc(const char *repository_id, const char *type_name,
const CORBA::EnummemberSeq& members);
```

この static メソッドは，指定された型とメンバーを持つ列挙体に対する TypeCode を動的に作成します。

パラメータ	説明
repository_id	IDL コンパイラによって生成されたか，動的に構築された識別子。
type_name	列挙体の型の名前。
members	列挙メンバーの値のリスト。

```
void create_environment(CORBA::Environment_out env);
```

Environment オブジェクトを作成します。

パラメータ	説明
env	新しく作成された Environment をポイントするように設定されるリファレンス。

```
static CORBA::TypeCode_ptr create_exception_tc(
   const char *repository_id,
   const char *type_name,
   const CORBA::StructMemberSeq& members);
```

この static メソッドは，指定された型とメンバーを持つ例外に対する TypeCode を動的に作成します。

パラメータ	説明
repository_id	IDL コンパイラによって生成されたか，動的に構築された識別子。
type_name	構造体の型の名前。
members	構造体メンバーの値のリスト。

```
static CORBA::TypeCode_ptr create_interface_tc(const char *repository_id, const char
*type_name);
```

この static メソッドは，指定された型と名前を持つインターフェースに対する TypeCode を動的に作成します。

パラメータ	説明
repository_id	IDL コンパイラによって生成されたか，動的に構築された識別子。
type_name	インターフェースの型の名前。

```
void create_list(CORBA::Long num, CORBA::NVList_out nvList);
```

指定された数の要素を持つ NVList を作成し，そのリストへの参照を返します。

パラメータ	説明
num	リスト内の要素の数。
nvlist	新しく作成されたリストをポイントするように初期化されます。

```
void create_named_value(CORBA::NamedValue_out value);
```

NamedValue オブジェクトを作成します。

```
void create_operation_list(CORBA::OperationDef_ptr opDefPtr, CORBA::NVList_out nvList);
```

指定された OperationDef オブジェクトの引数リストを作成します。

```
static CORBA::TypeCode_ptr create_recursive_sequence_tc(CORBA::Ulong bound, CORBA::Ulong
offset);
```

この static メソッドは，再帰シーケンスの TypeCode を動的に作成します。このメソッドの結果をほかの型の作成に利用できます。offset パラメータは，このシーケンスの要素を記述している中身の TypeCode を決定します。

パラメータ	説明
bound	シーケンス要素の最大数。
offset	現在の要素のタイプコードが前に生成されたバッファ内の位置。

```
static CORBA::TypeCode_ptr create_sequence_tc(CORBA::Ulong bound, CORBA::TypeCode_ptr
element_type);
```

この static メソッドは，シーケンスの TypeCode を動的に作成します。

パラメータ	説明
bound	シーケンス要素の最大数。
element_type	このシーケンスに保存される要素の型。

```
static CORBA::TypeCode_ptr create_string_tc(CORBA::Ulong bound);
```

この static メソッドは，文字列の TypeCode を動的に作成します。

パラメータ	説明
bound	文字列の最大の長さ。

```
static CORBA::TypeCode_ptr create_struct_tc(
   const char *repository_id, const char *type_name,
   const ORBA::StructMemberSeq& members);
```

この static メソッドは，指定された型とメンバーを持つ構造体に対する TypeCode を動的に作成します。

パラメータ	説明
repository_id	IDL コンパイラによって生成されたか，動的に構築された識別子。
type_name	構造体の型の名前。
members	構造体メンバーの値のリスト。

```
static CORBA::TypeCode_ptr create_union_tc(
   const char *repository_id,
   const char *type_name,
   CORBA::TypeCode_ptr discriminator_type,
   const CORBA::UnionMemberSeq& members);
```

この static メソッドは，指定された型，ディスクリミネータ，およびメンバーを持つ共用体に対する TypeCode を動的に作成します。

パラメータ	説明
repository_id	IDL コンパイラによって生成されたか，動的に構築された識別子。
type_name	共用体の型の名前。

パラメータ	説明
discriminator_typ	共用体のディスクリミネータの型。
members	共用体メンバーの値のリスト。

```
CORBA::Status get_default_context(CORBA::Context_ptr& contextPtr);
```

VisiBroker によって維持されるデフォルトのプロセスごとの Context を返します。デフォルトのコンテキストは，DII 要求の構築によく使用されます。詳細については，16 ページの「Context」を参照してください。

パラメータ	説明
contextPtr&	プロパティの値。

```
CORBA::Status get_next_response(CORBA::Request_out*& req);
```

遅延要求に関連付けられた応答を待機してブロックします。このメソッドを呼び出す前に，受信されることを待機している応答があるかどうかを判定するには，ORB::poll_next_response メソッドを使用します。

パラメータ	説明
req	受信した要求をポイントするように設定されます。

```
ObjectIdList *list_initial_services();
```

アプリケーションで使用可能なオブジェクトサービスの名前のリストを返します。これらのサービスには，ロケーションサービス，インターフェースリポジトリ，ネーミングサービス，イベントサービスなどがあります。ORB::resolve_initial_references メソッドを持つ返された名前を使用して，サービスに対して最上位のオブジェクトを取得できます。

```
char *object_to_string(CORBA::Object_ptr obj);
```

指定されたオブジェクトリファレンスを文字列に変換します。CORBA 仕様では，この処理を「文字列化」と呼びます。文字列に変換されたオブジェクトリファレンスは，たとえば，ファイルに保存することができます。これが ORB のメソッドであるのは，ORB インプリメンテーションが異なると，オブジェクトリファレンスを文字列として表す方法も異なるからです。

```
CORBA::BOA_ptr ORB::BOA_init(
   int& argc, char *const *argv,
   const char *boa_identifier = (char *)NULL);
```

BOA のハンドルを返し，オプションのネットワークパラメータを指定します。argc パラメータと argv パラメータは，オブジェクトインプリメンテーションプロセスの開始時に渡されたパラメータと同じです。

メモ 次のメソッドは VisiBroker 4.0 から使用されなくなりました。

パラメータ	説明
argc	渡される引数の数。
argv	引数への char ポインタの配列。2 つ以外のすべての引数は，キーワードと値の組という形式になり，以下に示されます。このメソッドは，認識できないキーワードをすべて無視します。
boa_identifier	使用する BOA の型を識別します。複数のスレッドサポートが必要な場合は，TPool を使用します。インプリメンテーションでスレッドを使用しない場合は，TSingle を使用します。

```
static CORBA::ORB_ptr ORB_init(int& argc, char *const *argv, const char *orb_id = NULL);
```

ORB を初期化し，ORB メソッドの呼び出しに使用できる ORB へのポインタを返します。このメソッドは，クライアントとオブジェクトインプリメンテーションの両方によっ

て使用されます。アプリケーションの main 関数に渡すパラメータ argc パラメータと argv パラメータを直接このメソッドに渡すことができます。このメソッドが受け取る引数は，名前／値の組という形式になり，その他のコマンドライン引数と区別されます。

パラメータ	説明
argc	渡される引数の数。
argv	引数への char ポインタの配列。2 つ以外のすべての引数は，キーワードと値の組という形式になり，以下に示されます。このメソッドは，認識できないキーワードをすべて無視します。
boa_identifier	使用する ORB の型を識別します。デフォルトは IIOP です。

```
void perform_work();
```

ORB に何らかの動作を行うように指示します。

```
CORBA::Boolean poll_next_response();
```

遅延要求への応答を受信した場合は TRUE を返し，それ以外の場合は FALSE を返します。この呼び出しはブロックしません。

```
CORBA::Object_ptr resolve_initial_references(const char * identifier);
```

ORB::list_initial_services メソッドから返された名前の 1 つを対応するインプリメンテーションオブジェクトに解決します。このメソッドの詳細については，26 ページの「ObjectIdList *list_initial_services();」を参照してください。解決されたオブジェクトが返されると，適切なサーバーの型にナローイングできます。指定されたサービスが見つからなかった場合は，InvalidName 例外が生成されます。

パラメータ	説明
identifier	最上位のオブジェクトを取得するサービスの名前。この識別子は，返されるオブジェクトの**名前ではありません**。

```
void send_multiple_requests_deferred(const CORBA::RequestSeq& req);
```

指定されたシーケンス内のすべてのクライアント要求を遅延要求として送信します。ORB は，オブジェクトインプリメンテーションからの応答を待ちません。クライアントアプリケーションは，ORB::get_next_response メソッドを使用してそれぞれの要求に対する応答を取得する必要があります。

パラメータ	説明
req	送信する遅延要求のシーケンス。

```
void send_multiple_requests_oneway(const CORBA::RequestSeq& req);
```

指定されたシーケンス内のすべてのクライアント要求を一方向要求として送信します。一方向要求では，オブジェクトインプリメンテーションからの応答は生成されないため，ORB はどの要求からの応答も待ちません。

パラメータ	説明
req	送信する一方向要求のシーケンス。

```
CORBA::Object_ptr string_to_object(const char *str);
```

オブジェクトを表す文字列をオブジェクトポインタに変換します。文字列は，ORB::object_to_string メソッドを使用して作成されている必要があります。

パラメータ	説明
str	オブジェクトを表す文字列へのポインタ。

```
static CORBA::ORB_ptr _duplicate(CORBA::ORB_ptr ptr);
```

この static メソッドは，指定された ORB ポインタをコピーし，コピーされた ORB へのポインタを返します。

パラメータ	説明
ptr	コピーする ORB ポインタ。

```
static CORBA::ORB_ptr _nil();
```

この static メソッドは，初期化の目的に合った NULL ORB ポインタを返します。

```
void run();
```

ORB が処理を開始するようにします。この ORB は，要求を受信してディスパッチします。この呼び出しは，ORB がシャットダウンされるまで，このプロセスをブロックします。

```
static void shutdown(CORBA::Boolean wait_for_completion=0);
```

前の impl_is_ready メソッドの呼び出しから戻ります。すべてのオブジェクトアダプタはシャットダウンされ，関連付けられているメモリは解放されます。wait_for_completion パラメータが TRUE の場合，このオペレーションはシャットダウンが完了するまでブロックされます。現在呼び出しを処理しているスレッドでアプリケーションがこれを実行すると，システム例外 BAD_INV_ORDER が生成されます。wait_for_completion パラメータが FALSE の場合は，メソッドが戻ったときにシャットダウンが完了していない可能性があります。

パラメータ	説明
wait_for_completion	シャットダウンが完了するのを待つかどうかを指定します。

```
static void destroy();
```

このオペレーションは ORB を破棄します。これにより，そのリソースをアプリケーションで再利用できます。シャットダウンしていない ORB に対して destroy を呼び出すと，シャットダウンプロセスが開始されます。destroy は ORB がシャットダウンするまでブロックされ，その後で ORB が破棄されます。これは，wait_for_completion parameter を TRUE に設定して shutdown を呼び出した場合の動作と同じです。現在呼び出しを処理しているスレッドでアプリケーションが Destroy を呼び出すと，システム例外 BAD_INV_ORDER が生成されます。

CORBA::ORB に対する VisiBroker 拡張機能

```
CORBA::Object_ptr bind(
   const char *rep_id,
   const char *object_name = (
      const char*)NULL, const char *host_name = (
      const char*)NULL, CORBA::BindOptions *opt = (
         CORBA::BindOptions*)NULL);
```

オブジェクトのリポジトリ ID を指定して，その共通オブジェクトリファレンスを所得します。オプションで，オブジェクトの名前とオブジェクトが実装されているホストの名前も指定できます。

パラメータ	説明
rep_id	IDL コンパイラによって生成されたか，このオブジェクトのために動的に構築された識別子。
object_name	オブジェクトの名前。このパラメータは省略可能です。

パラメータ	説明
host_name	オブジェクトインプリメンテーションがあるホストの名前。IP アドレスまたは完全なホスト名として指定します。このパラメータは省略可能です。
opt	オブジェクトのバインドオプション。このパラメータは省略可能です。バインドオプションについては，10 ページの「BindOptions」を参照してください。

`CORBA::ULong` **`connection_count`**`()`

クライアントアプリケーションによって使用され，現在アクティブな接続の数を返します。

`void` **`connection_max`**`(CORBA::ULong` **`max_conn`**`)`

利用できる最大接続数を設定するためにサーバーによって使用されます。

パラメータ	説明
max_conn	許容する最大接続数。

`CORBA::ULong` **`connection_max`**`()`

利用できる最大接続数を取得するためにクライアントアプリケーションによって使用されます。

`static CORBA::TypeCode_ptr` **`create_wstring_tc`**`(CORBA::Ulong` **`bound`**`);`

この static メソッドは，Unicode 文字列の TypeCode を動的に作成します。

パラメータ	説明
bound	文字列の最大の長さ。

PortableServer::POA

`class` **`PortableServer::POA`**

POA クラスのオブジェクトは，オブジェクトの集合のインプリメンテーションを管理します。POA は，オブジェクト ID で識別されるこれらのオブジェクトの名前空間をサポートします。POA は既存の POA の子として作成され，ルート POA から始まる階層構造を形成することから，POA はほかの POA にも名前空間を提供することになります。

POA オブジェクトをほかのプロセスにエキスポートしたり，文字列化することはできません。無理に実行しようとすると，MARSHAL 例外が発生します。

PortableServer::POA のメソッド

```
PortableServer::ObjectId* activate_object(PortableServer::Servant _p_servant);
```

オブジェクト ID を生成し，それを返します。そのオブジェクト ID，および指定された _p_servant がアクティブオブジェクトマップに入力されます。POA が UNIQUE_ID ポリシーを持ち，_p_servant がすでにアクティブオブジェクトマップにある場合は，ServantAlreadyActive 例外が生成されます。

このメソッドを使用するには，POA が SYSTEM_ID ポリシーと RETAIN ポリシーを持つ必要があります。そうでない場合は，WrongPolicy 例外が生成されます。

パラメータ	説明
_p_servant	アクティブオブジェクトマップに入れられる Servant。

```
void activate_object_with_id(
      const PortableServer::ObjectId& _oid,
      PortableServer::Servant _p_servant);
```

指定された _oid をアクティブ化し，さらにアクティブオブジェクトマップ内で指定された _p_servant に関連付けます。この _oid にバインドされたサーバントがすでにアクティブオブジェクトマップ内にある場合は，ObjectAlreadyActive 例外が生成されます。POA が UNIQUE_ID ポリシーを持ち，_p_servant がすでにアクティブオブジェクトマップにある場合は，ServantAlreadyActive 例外が生成されます。

POA が SYSTEM_ID ポリシーを持ち，_oid がシステムによって生成されていないか，この POA に対して生成されていないことが検知された場合，このメソッドは，BAD_PARAM 例外を生成します。

このメソッドを使用するには，POA が RETAIN ポリシーを持つ必要があります。そうでない場合は，WrongPolicy 例外が生成されます。

パラメータ	説明
_oid	アクティブ化するオブジェクトの ObjectId。
_p_servant	アクティブオブジェクトマップに入れられる Servant。

```
PortableServer::ImplicitActivationPolicy_ptr create_implicit_activation_policy(
        PortableServer::ImplicitActivationPolicyValue _value);
```

指定された _value を持つ ImplicitActivationPolicy オブジェクトへのポインタを返します。アプリケーションは，Policy オブジェクトが不要になった後で，Policy オブジェクトの継承された destroy メソッドを呼び出す必要があります。

POA の作成時に ImplicitActivationPolicy が指定されなかった場合は，デフォルトの NO_IMPLICIT_ACTIVATION が使用されます。

パラメータ	説明
_value	IMPLICIT_ACTIVATION に設定した場合，POA は，サーバントを暗黙的にアクティブ化します。また，SYSTEM_ID ポリシーと RETAIN ポリシーが必要です。NO_IMPLICIT_ACTIVATION に設定した場合，POA は，サーバントの暗黙的なアクティブ化をサポートしません。

```
CORBA::Object_ptr create_reference(const char* _intf);
```

POA が生成した ObjectId と指定された _intf 値をカプセル化するオブジェクトリファレンスを生成し，それを返します。null になる _intf は生成したオブジェクトリファレンスの type_id になります。このメソッドはアクティブ化を行いません。_intf 値がこのオブジェクトの最下位派生インターフェースまたはそのベースインターフェースの 1 つを指していない場合は，予期しない動作が起こります。ObjectId は，返されたオブジェクトの POA::reference_to_id メソッドを呼び出すことによって取得できます。

このメソッドを使用するには，POA が RETAIN ポリシーを持つ必要があります。そうでない場合は，WrongPolicy 例外が生成されます。

パラメータ	説明
_intf	作成するオブジェクトのクラスのリポジトリインターフェース ID。

```
CORBA::Object_ptr create_reference_with_id (
        const PortableServer::ObjectId& _oid,const char* _intf);
```

指定された _oid と _intf 値をカプセル化するオブジェクトリファレンスを生成し，それを返します。null 文字列になる _intf は生成したオブジェクトリファレンスの type_id になります。_intf 値がこのオブジェクトの最下位派生インターフェースまたはそのベースインターフェースの 1 つを指していない場合は，予期しない動作が起こります。このメソッドはアクティブ化を行いません。返されたオブジェクトリファレンスは，クライアントに渡すことができます。それ以降にこのリファレンスを使った要求があると，適用されているポリシーにしたがい，必要に応じてオブジェクトがアクティブ化されるか，デフォルトのサーバントが使用されます。

POA が SYSTEM_ID ポリシーを持ち，ObjectId がシステムによって生成されていないか，この POA に対して生成されていないことが検知された場合，このメソッドは，BAD_PARAM 例外を生成します。

パラメータ	説明
_oid	参照が作成されるオブジェクトの ID。
_intf	作成するオブジェクトのクラスのリポジトリインターフェース ID。

```
PortableServer::IdAssignmentPolicy_ptr create_id_assignment_policy
(PortableServer::IdAssignmentPolicyValue _value);
```

指定された _value を持つ IdAssignmentPolicy オブジェクトへのポインタを返します。アプリケーションは，Policy オブジェクトが不要になった後で，継承された destroy メソッドを呼び出す必要があります。

POA の作成時に IdAssignmentPolicy が指定されなかった場合は，デフォルトの SYSTEM_ID が使用されます。

パラメータ	説明
_value	USER_ID に設定した場合，POA を使用して作成されたオブジェクトには，アプリケーションによってのみオブジェクト ID が割り当てられます。SYSTEM_ID に設定した場合，POA を使用して作成されたオブジェクトには POA によってのみオブジェクト ID が割り当てられます。

```
PortableServer::IdUniquenessPolicy_ptr create_id_uniqueness_policy
(PortableServer::IdUniquenessPolicyValue _value);
```

指定された _value を持つ IdUniquenessPolicy オブジェクトへのポインタを返します。アプリケーションは，Policy オブジェクトが不要になった後で，継承された destroy メソッドを呼び出す必要があります。

POA の作成時に IdUniquenessPolicy が指定されなかった場合は，デフォルトの UNIQUE_ID が使用されます。

パラメータ	説明
_value	UNIQUE_ID に設定した場合，POA を使用してアクティブ化されたサーバントは，オブジェクト ID を 1 つだけサポートします。MULTIPLE_ID に設定した場合，POA を使用してアクティブ化されたサーバントは，1 つまたは複数のオブジェクト ID をサポートできます。

```
PortableServer::LifespanPolicy_ptr create_lifespan_policy
(PortableServer::LifespanPolicyValue _value);
```

指定された _value を持つ LifespanPolicy オブジェクトへのポインタを返します。アプリケーションは，Policy オブジェクトが不要になった後で，継承された destroy メソッドを呼び出す必要があります。

POA の作成時に LifespanPolicy が指定されなかった場合は，デフォルトの TRANSIENT が使用されます。

パラメータ	説明
_value	TRANSIENT に設定した場合，POA で実装されているオブジェクトは，最初に作成された POA インスタンスの存続期間より長くは存続できません。一時的な POA が非アクティブ化された場合は，それから生成されたオブジェクトリファレンスを使用すると，OBJECT_NOT_EXIST 例外が生成されます。PERSISTENT に設定した場合，POA で実装されているオブジェクトは，最初に作成されたプロセスの存続期間をすぎても存続できます。

```
PortableServer::POA_ptr create_POA(
      const char* _adapter_name,
      PortableServer::POAManager_ptr _a_POAManager,
      const CORBA::PolicyList& _policies);
```

指定された _adapter_name を持つ新しい POA を作成します。この新しい POA は指定された _a_POAManager の子です。その親 POA に同じ名前の子 POA がすでに存在する場合は，PortableServer::AdapterAlreadyExists 例外が生成されます。

指定された _policies は新しい POA に関連付けられ，その動作を制御するために使用されます。

パラメータ	説明
_adapter_name	新しい POA に付ける名前。
_a_POAManager	新しい POA の親 POA オブジェクト。
_policies	新しい POA に適用するポリシーのリスト。ポリシーオブジェクトは，このオペレーションが戻る前にコピーされます。

```
PortableServer::RequestProcessingPolicy_ptr
      create_request_processing_policy(
         PortableServer::RequestProcessingPolicyValue _value);
```

指定された _value を持つ RequestProcessingPolicy オブジェクトへのポインタを返します。アプリケーションは，Policy オブジェクトが不要になった後で，継承された destroy メソッドを呼び出す必要があります。

POA の作成時に RequestProcessingPolicy を指定しない場合は，デフォルトの USE_ACTIVE_OBJECT_MAP_ONLY が使用されます。

パラメータ	説明
_value	USE_ACTIVE_OBJECT_MAP_ONLY に設定した場合は，アクティブオブジェクトマップにオブジェクト ID が見つからないと，OBJECT_NOT_EXIST 例外がクライアントに返されます。RETAIN ポリシーも必要です。USE_DEFAULT_SERVANT に設定した場合は，アクティブオブジェクトマップにオブジェクト ID が見つからないか，NON_RETAIN ポリシーがあると，次のように処理されます。set_servant メソッドを使用して，すでにデフォルトサーバントが POA に登録されている場合，その要求はデフォルトサーバントに送られます。デフォルトサーバントが登録されていない場合は，OBJ_ADAPTER 例外がクライアントに返されます。MULTIPLE_ID ポリシーも必要です。USE_SERVANT_MANAGER に設定した場合は，アクティブオブジェクトマップにオブジェクト ID が見つからないか，NON_RETAIN ポリシーがあると，次のように処理されます。set_servant_manager メソッドを使用して，すでにサーバントマネージャが POA に登録されている場合，そのサーバントマネージャにサーバントを検索するか，例外を生成する機会が与えられます。デフォルトサーバントが登録されていない場合は，OBJ_ADAPTER 例外がクライアントに返されます。

```
PortableServer::ServantRetentionPolicy_ptr
      create_servant_retention_policy (
         PortableServer::ServantRetentionPolicyValue _value);
```

指定された _value を持つ ServantRetentionPolicy オブジェクトへのポインタを返します。アプリケーションは, Policy オブジェクトが不要になった後で, 継承された destroy メソッドを呼び出す必要があります。

POA の作成時に ServantRetentionPolicy が指定されなかった場合は，デフォルトの RETAIN が使用されます。

パラメータ	説明
_value	RETAIN に設定した場合，POA はアクティブなサーバントをアクティブオブジェクトマップに保持します。NON_RETAIN に設定した場合，POA はサーバントを保持しません。

```
PortableServer::ThreadPolicy_ptr create_thread_policy(
        PortableServer::ThreadPolicyValue _value);
```

指定された _value を持つ ThreadPolicy オブジェクトへのポインタを返します。アプリケーションは，Policy オブジェクトが不要になった後で，継承された destroy メソッドを呼び出す必要があります。

POA の作成時に ThreadPolicy が指定されなかった場合は，デフォルトの ORB_CTRL_MODEL が使用されます。

パラメータ	説明
_value	ORB_CTRL_MODEL に設定した場合は，ORB が，ORB の制御下にある POA に対する要求をスレッドに割り当てます。マルチスレッド環境では，同時に複数の要求があった場合，それらに複数のスレッドを提供します。SINGLE_THREAD_MODEL に設定した場合，POA に対する複数の要求は順番に処理されます。マルチスレッド環境では，サーバントやサーバントマネージャに対する POA からの呼び出しは，マルチスレッドに対応していないコードにも安全な方法で行われます。

```
void deactivate_object(const PortableServer::ObjectId& _oid);
```

指定された _oid 値を非アクティブ化します。ObjectId は，非アクティブ化された後も，その ObjectId に対するアクティブな要求がなくなるまで，要求を処理し続けます。その ObjectId に対するすべての要求の実行が完了すると，その ObjectId はアクティブオブジェクトマップから削除されます。

ServantManager が POA に関連付けられている場合は，ObjectId がアクティブオブジェクトマップから削除された後, その ObjectId とそれに関連付けられているサーバントとともに ServantActivator::etherealize メソッドが呼び出されます。その ObjectId を再アクティブ化すると，必要な霊化が完了するまでブロックします。ただし，このメソッドは，要求または霊化の完了を待たず，指定された _oid を非アクティブ化した後，必ずただちに戻ります。

このメソッドを使用するには，POA が RETAIN ポリシーを持つ必要があります。そうでない場合は，WrongPolicy 例外が生成されます。

パラメータ	説明
_oid	非アクティブ化するオブジェクトの ObjectId。

```
void destroy(
      CORBA::Boolean _etherealize_objects,
      CORBA::Boolean _wait_for_completion);
```

この POA オブジェクトとその下位のすべての POA を廃棄します。最初に子が廃棄され，最後に現在のコンテナ POA が廃棄されます。その後，必要に応じて，廃棄した POA と同じ名前の POA を同じプロセス内で作成することができます。

パラメータ	説明
_etherealize_objects	これが TRUE で，POA が RETAIN ポリシーを持ち，サーバントマネージャが POA に登録されている場合は，アクティブオブジェクトマップ内の各アクティブオブジェクトごとに etherealize メソッドが呼び出されます。etherealize メソッドが呼び出される前に POA がはっきりと廃棄されるので，etherealize メソッドが POA のメソッドを呼び出そうとすると，OBJECT_NOT_EXIST 例外が生成されます。
_wait_for_completion	これが TRUE で，現在のスレッドが，この POA と同じ ORB に属する別の POA から送られた呼び出しコンテキスト内にない場合，destroy メソッドは，アクティブな要求および etherealize メソッドの呼び出しがすべて完了してから戻ります。これが FALSE で，現在のスレッドが，この POA と同じ ORB に属する別の POA から送られた呼び出しコンテキスト内にある場合は BAD_INV_ORDER 例外が生成され，POA の廃棄は行われません。

```
PortableServer::POA_ptr find_POA(
     const char* _adapter_name,CORBA::Boolean _activate_it);
```

このメソッドが呼び出される対象の POA オブジェクトが指定された _adapter_name を持つ POA の親である場合は，その子 POA が返されます。

パラメータ	説明
_adapter_name	POA に関連付けられている AdapterActivator の名前。
_activate_it	TRUE に設定し，adapter_name で指定された POA の子 POA が存在しない場合は，_adapter_name が null でなければその POA の AdapterActivator が呼び出されます。子 POA のアクティブ化に成功すると，その POA が返されます。そうでない場合は，AdapterNonExistent 例外が生成されます。

```
PortableServer::Servant get_servant();
```

この POA に関連付けられているデフォルトの Servant を返します。POA に関連付けられた Servant がない場合は，NoServant 例外が生成されます。

このメソッドを使用するには，POA が USE_DEFAULT_SERVANT ポリシーを持つ必要があります。そうでない場合は，WrongPolicy 例外が生成されます。

```
PortableServer::ServantManager_ptr get_servant_manager();
```

この POA に関連付けられている ServantManager オブジェクトへのポインタを返します。POA に関連付けられた ServantManager がない場合，結果は null です。

このメソッドを使用するには，POA が USE_SERVANT_MANAGER ポリシーを持つ必要があります。そうでない場合は，WrongPolicy 例外が生成されます。

```
CORBA::Object_ptr id_to_reference(PortableServer::ObjectId& _oid);
```

指定された _oid 値が現在アクティブな場合は，そのオブジェクトリファレンスを返します。_oid がアクティブでない場合は，ObjectNotActive 例外が生成されます。

このメソッドを使用するには，POA が RETAIN ポリシーを持つ必要があります。そうでない場合は，WrongPolicy 例外が生成されます。

パラメータ	説明
_oid	その参照を取得するオブジェクトの ObjectId。

```
PortableServer::Servant id_to_servant(PortableServer::ObjectId& _oid);
```

このメソッドは，次の 3 つの動作の 1 つを行います。

- POA が RETAIN ポリシーを持ち，指定された _oid がアクティブオブジェクトマップにある場合，このメソッドは，アクティブオブジェクトマップ内でそのオブジェクトに関連付けられているサーバントを返します。
- POA が USE_DEFAULT_SERVANT ポリシーを持ち，POA にデフォルトサーバントが登録されている場合，このメソッドはそのデフォルトサーバントを返します。
- そうでない場合は，ObjectNotActive 例外が生成されます。

このメソッドを使用するには POA が USE_DEFAULT_SERVANT ポリシーを持つ必要があります。ポリシーがない場合は，WrongPolicy 例外が生成されます。

パラメータ	説明
_oid	そのサーバントを取得するオブジェクトの ObjectId。

```
PortableServer::Servant reference_to_servant(CORBA::Object_ptr _reference);
```

このメソッドは，次の 3 つの動作の 1 つを行います。

- POA が RETAIN ポリシーを持ち，指定された _reference がアクティブオブジェクトマップにある場合，このメソッドは，アクティブオブジェクトマップ内でそのオブジェクトに関連付けられているサーバントを返します。
- POA が USE_DEFAULT_SERVANT ポリシーを持ち，POA にデフォルトサーバントが登録されている場合，このメソッドはそのデフォルトサーバントを返します。
- そうでない場合は，ObjectNotActive 例外が生成されます。

このメソッドを使用するには，POA が RETAIN または USE_DEFAULT_SERVANT ポリシーを持つ必要があります。そうでない場合は，WrongPolicy 例外が生成されます。

パラメータ	説明
_reference	そのサーバントを取得するオブジェクト。

```
PortableServer::ObjectId* reference_to_id(CORBA::Object_ptr _reference);
```

指定された _reference でカプセル化された ObjectId 値を返します。_reference がこの POA によって作成されている場合にだけ，このメソッドを有効に呼び出すことができます。_reference がこの POA によって作成されていない場合は，WrongAdapter 例外が生成されます。_reference パラメータが指すオブジェクトがアクティブでなくても，このメソッドは成功します。

IDL では，このメソッドによって WrongPolicy 例外が生成される可能性があるとされていますが，これは，将来の拡張を考えての記述です。

パラメータ	説明
_reference	その ObjectId を取得するオブジェクト。

```
PortableServer::ObjectId* servant_to_id(
    PortableServer::Servant _p_servant);
```

このメソッドは，次の 4 つの動作のいずれかを実行します。

- POA が UNIQUE_ID ポリシーを持ち，指定された _p_servant がアクティブな場合は，_p_servant に関連付けられた ObjectId が返されます。
- POA が IMPLICIT_ACTIVATION ポリシーを持ち，POA が MULTIPLE_ID ポリシーを持つか指定された _p_servant がアクティブでない場合は，POA によって生成された ObjectId と _p_servant に関連付けられたリポジトリインターフェース ID を使用して _p_servant がアクティブ化され，その ObjectId が返されます。

- POA が USE_DEFAULT_SERVANT ポリシーを持ち，指定された _p_servant がデフォルトサーバントである場合は，現在の呼び出しに関連付けられた ObjectId が返されます。
- そうでない場合は，ServantNotActive 例外が生成されます。

このメソッドを使用するには，USE_DEFAULT_SERVANT ポリシーが必要です。または，UNIQUE_ID と IMPLICIT_ACTIVATION のどちらかのポリシーと RETAIN ポリシーの組み合わせが必要です。そうでない場合は，WrongPolicy 例外が生成されます。

パラメータ	説明
_p_servant	その ObjectId を取得するサーバント。

```
CORBA::Object_ptr servant_to_reference(PortableServer::Servant _p_servant);
```

このメソッドは，次の動作のいずれかを実行します。

- POA が RETAIN と UNIQUE_ID の両方のポリシーを持ち，指定された _p_servant がアクティブな場合は，サーバントのアクティブ化に使用された情報をカプセル化するオブジェクトリファレンスが返されます。
- POA が RETAIN と IMPLICIT_ACTIVATION の両方のポリシーを持ち，POA が MULTIPLE_ID ポリシーを持つか，指定された _p_servant がアクティブでない場合は，POA によって生成された ObjectId と _p_servant に関連付けられたリポジトリインターフェース ID を使用して _p_servant がアクティブ化され，対応するオブジェクトリファレンスが返されます。
- 指定された _p_servant で要求を実行するコンテキスト内でこのメソッドが呼び出された場合は，現在の呼び出しに関連付けられている参照が返されます。
- そうでない場合は，ServantNotActive 例外が生成されます。

この POA によって送られたメソッドのコンテキスト外でこのメソッドを呼び出すには，UNIQUE_ID と IMPLICIT_ACTIVATION のどちらかのポリシーと RETAIN ポリシーが必要です。指定された _p_servant で要求を実行するコンテキスト内でこのメソッドを呼び出さず，上のポリシーもない場合は，WrongPolicy 例外が生成されます。

パラメータ	説明
_p_servant	その参照を取得する Servant。

```
void set_servant(PortableServer::Servant _p_servant);
```

この POA に関連付けられているデフォルトの Servant を設定します。アクティブオブジェクトマップ内に見つからないサーバントに対する要求には，指定されたサーバントが使用されます。

このメソッドを使用するには，POA が USE_DEFAULT_SERVANT ポリシーを持つ必要があります。そうでない場合は，WrongPolicy 例外が生成されます。

パラメータ	説明
_p_servant	この POA のデフォルトとして使用される Servant。

```
void set_servant_manager(PortableServer::ServantManager_ptr _imagr);
```

この POA に関連付けられているデフォルトの ServantManager を設定します。このメソッドは，POA が作成された後でのみ呼び出すことができます。ServantManager がすでに設定された後でもう一度設定しようとすると，BAD_INV_ORDER 例外が生成されます。

このメソッドを使用するには，POA が USE_SERVANT_MANAGER ポリシーを持つ必要があります。そうでない場合は，WrongPolicy 例外が生成されます。

パラメータ	説明
_imagr	この POA のデフォルトとして使用される ServantManager。

```
PortableServer::AdapterActivator_ptr the_activator();
```

この POA に関連付けられている AdapterActivator を返します。POA は，作成時には AdapterActivator を持っていません。つまり，その属性は null です。ルート POA がアクティベータを持つかどうか，およびアプリケーションが必要に応じてアクティベータを割り当てることができるかどうかは，システムに依存します。

```
void the_activator(PortableServer::AdapterActivator_ptr _val);
```

この POA に関連付けられる AdapterActivator オブジェクトとして，指定されたアクティベータを設定します。

パラメータ	説明
_val	この POA に関連付けられる ActivatorAdapter。

```
char* the_name();
```

POA をその親から識別する読み取り専用の属性を返します。この親は POA の作成時に割り当てられます。ルート POA の名前はシステムによって異なるので，アプリケーションはその名前に依存しないようにする必要があります。

```
PortableServer::POA_ptr the_parent();
```

この POA の親 POA を返します。ルート POA の親は null です。

```
Portableserver::POAManager_ptr the_POAManager();
```

POA に関連付けられている POAManager へのポインタである読み取り専用の属性を返します。

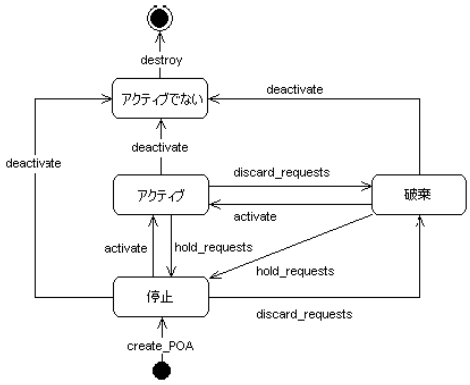
PortableServer::POAManager

各 POA は関連付けられた POA マネージャを 1 つ持ち，一方，POA マネージャは 1 つまたは複数の POA オブジェクトに関連付けられます。POA マネージャは，関連付けられた複数の POA の処理状態をカプセル化します。

POA マネージャの状態は，次の 4 つがあります。

- アクティブ
- アクティブでない
- 停止
- 破棄

POA マネージャは，停止状態で作成されます。次の図は，呼び出されたメソッドにしたがって，POA マネージャの状態が推移していくようすを示したものです。



インクルードファイル

この構造体を使用する場合は，**poa_c.hh** ファイルをインクルードする必要があります。

PortableServer::POAManager のメソッド

```
void activate();
```

POA マネージャを「アクティブな」状態に変更します。これにより，関連付けられている POA は，要求を処理することができます。POA マネージャがアクティブでない状態のときに，このメソッドを呼び出した場合は，AdapterInactive 例外が生成されます。

```
void deactivate(CORBA::Boolean _etherealize_objects,CORBA::Boolean
  _wait_for_completion);
```

POA マネージャを「アクティブでない」状態に変更します。これにより，関連付けられている POA は，まだ実行が開始されていない要求，および新しい要求を拒否します。POA マネージャがアクティブでない状態のときに，このメソッドを呼び出した場合は，AdapterInactive 例外が生成されます。

状態が変更された後の動作は，etherealize_objects パラメータの値によって異なります。

- TRUE - POA マネージャは，RETAIN ポリシーと USE_SERVANT_MANAGER ポリシーを持つすべての関連する POA に，すべてのアクティブなオブジェクトに対して，関連するサーバントマネージャの etherealize オペレーションを実行させます。
- FALSE - etherealize オペレーションは呼び出されません。これは，危機的な状況（回復不可能なエラーなど）にある POA をシャットダウンする手段を開発者に提供します。

wait_for_completion パラメータが FALSE の場合，このオペレーションは状態を変更した後ただちに戻ります。このパラメータが TRUE で，現在のスレッドが，この POA と同じ ORB に属するいずれかの POA によってディスパッチされた呼び出しコンテキスト内にない場合，このオペレーションは，この POA マネージャに関連付けられている任意の POA でアクティブに実行されている要求がなくなり（つまり，状態の変更より前に開始されたすべての要求が完了し），さらに etherealize_objects パラメータが TRUE の場合は，RETAIN ポリシーと USE_SERVANT_MANAGER ポリシーを持つ POA に対するすべての etherealize 呼び出しが完了するまで戻りません。このパラメータが TRUE で，現在のスレッドが，この POA と同じ VisiBroker ORB に属するいずれかの POA によってディスパッチされた呼び出しコンテキスト内にある場合は，BAD_INV_ORDER 例外が生成され，状態は変更されません。

```
void discard_requests(CORBA::Boolean _wait_for_completion);
```

POA マネージャを「破棄」状態に変更します。これにより，関連付けられている POA は着信した要求を破棄します。さらに，キューに入っているが，まだ実行が開始されていない要求がすべて破棄されます。要求が破棄されると，TRANSIENT 例外がクライアントに返されます。POA マネージャがアクティブでない状態のときに，このメソッドを呼び出した場合は，AdapterInactive 例外が生成されます。

wait_for_completion パラメータが FALSE の場合，このオペレーションは状態を変更した後ただちに戻ります。このパラメータが TRUE で，現在のスレッドが，この POA と同じ ORB に属するいずれかの POA によってディスパッチされた呼び出しコンテキスト内にない場合，このオペレーションは，この POA マネージャに関連付けられている任意の POA でアクティブに実行されている要求がなくなり（つまり，状態の変更より前に開始されたすべての要求が完了し），さらに POA マネージャの状態が破棄以外の状態に変更されるまで戻りません。このパラメータが TRUE で，現在のスレッドが，この POA と同じ VisiBroker ORB に属するいずれかの POA によってディスパッチされた呼び出しコンテキスト内にある場合は，BAD_INV_ORDER 例外が生成され，状態は変更されません。

```
void hold_requests(CORBA::Boolean _wait_for_completion);
```

POA マネージャを「停止」状態に変更します。これにより，関連付けられている POA は着信した要求をキューに入れます。キューに入っており実行されていない要求は，停止状態の間そのままキューの中に保持されます。POA マネージャがアクティブでない状態のときに，このメソッドを呼び出した場合は，AdapterInactive 例外が生成されます。

wait_for_completion パラメータが FALSE の場合，このオペレーションは状態を変更した後ただちに戻ります。このパラメータが TRUE で，現在のスレッドが，この POA と同じ ORB に属するいずれかの POA によってディスパッチされた呼び出しコンテキスト内にない場合，このオペレーションは，この POA マネージャに関連付けられている任意の POA でアクティブに実行されている要求がなくなり（つまり，状態の変更より前に開始されたすべての要求が完了し），さらに etherealize_objects パラメータが TRUE の場合は，RETAIN ポリシーと USE_SERVANT_MANAGER ポリシーを持つ POA に対するすべての etherealize 呼び出しが完了するまで戻りません。このパラメータが TRUE で，現在のスレッドが，この POA と同じ VisiBroker ORB に属するいずれかの POA によってディスパッチされた呼び出しコンテキスト内にある場合は，BAD_INV_ORDER 例外が生成され，状態は変更されません。

Principal

```
typedef OctetSequence Principal
```

Principal はクライアントアプリケーションのために要求が行われる場合に，そのアプリケーションを表すために使用されます。オブジェクトインプリメンテーションは，クライアントの Principal の内容に基づいて，バインド要求を受け付けたり拒否することができます。

メモ この機能は VisiBroker 4.0 で使用されなくなりました。

インクルードファイル

この typedef を使用する場合は，**corba.h** ファイルをインクルードする必要があります。

Principal のメソッド

BOA クラスには get_principal メソッドがあり，オブジェクトに関連付けられている Principal へのポインタを返します。Object クラスにも，Principal を取得および設定するためのメソッドがあります。

PortableServer::RefCountServantBase

```
class RefCountServantBase : public ServantBase
```

派生クラスで使用される PortableServer::ServantBase のかわりに，このクラスを標準のサーバント参照カウントミックスインクラスとして使用できます （41 ページの「PortableServer::ServantBase」も参照）。

インクルードファイル

この構造体を使用する場合は，**poa_c.hh** ファイルをインクルードする必要があります。

PortableServer::RefCountServantBase のメソッド

```
void _add_ref();
```

参照カウントを 1 だけインクリメントします。ベースクラスのこのメソッドを使用して，正確な参照カウントを提供できます。

```
void _remove_ref();
```

参照カウントを 1 だけデクリメントします。ベースクラスのこのメソッドをオーバーライドすると，正確な参照カウントを提供できます。

PortableServer::ServantActivator

```
class PortableServer::ServantActivator : public PortableServer::ServantManager
```

POA が RETAIN ポリシーを持つ場合，その POA はサーバントマネージャとして PortableServer::ServantActivator オブジェクトを使用します。

インクルードファイル

この構造体を使用する場合は，**poa_c.hh** ファイルをインクルードする必要があります。

PortableServer::ServantActivator のメソッド

```
void etherealize(PortableServer::ObjectId& oid,
      PortableServer::POA_ptr adapter,
      PortableServer::Servant serv,
      CORBA::Boolean cleanup_in_progress,
      CORBA::Boolean remaining_activations);
```

RETAIN ポリシーと USE_SERVANT_MANAGER ポリシーが設定されているとき，あるオブジェクト（指定された oid）のサーバントが非アクティブ化されるたびに指定された adapter によって呼び出されます。

パラメータ	説明
oid	そのサーバントが非アクティブ化されるオブジェクトのオブジェクト ID。
adapter	そのスコープ内でオブジェクトがアクティブだった POA。
serv	非アクティブ化されるサーバント。
cleanup_in_progress	TRUE の場合，このメソッドが呼び出された理由は，etherealize_objects パラメータが TRUE に設定されて deactivate メソッドまたは destroy メソッドが呼び出されたためです。そうでない場合，このメソッドはその他の理由で呼び出されました。
remaining_activations	指定された serv が，指定された adapter のその他のオブジェクトに関連付けられている場合は，TRUE に設定されます。そうでない場合は，FALSE に設定されます。

```
PortableServer::Servant incarnate(const PortableServer::ObjectId& oid,
        PortableServer::POA_ptr adapter);
```

RETAIN ポリシーと USE_SERVANT_MANAGER ポリシーが設定されているとき，POA がアクティブでないオブジェクト（指定された oid）に対する要求を受信するたびに POA によって呼び出されます。

サーバントマネージャのインプリメンテーションを提供するのはユーザーです。このインプリメンテーションにより，指定された oid 値に対応する適切なサーバントを検索および作成します。このメソッドはサーバントを返し，そのサーバントはアクティブオブジェクトマップにも入力されます。これ以降，このアクティブなオブジェクトに対して要求があると，関連付けられているサーバントに直接要求が渡されます。

このメソッドが別のオブジェクト ID ですでにアクティブ化されているサーバントを返し，POA が UNIQUE_ID ポリシーも持つ場合は OBJ_ADAPTER 例外が生成されます。

パラメータ	説明
oid	そのサーバントがアクティブ化されるオブジェクトのオブジェクト ID。
adapter	そのスコープ内でオブジェクトがアクティブ化される POA。

PortableServer::ServantBase

```
class PortableServer::ServantBase
```

Portable::ServantBase クラスは，サーバーアプリケーションのベースクラスです。

インクルードファイル

この構造体を使用する場合は，**poa_c.hh** ファイルをインクルードする必要があります。

PortableServer::ServantBase のメソッド

```
void _add_ref();
```

このサーバントの参照カウントを追加します。デフォルトのインプリメンテーションは何も実行しないため，派生クラスでこのメソッドをオーバーライドし，参照カウント機能を提供する必要があります。

```
PortableServer::POA_ptr _default_POA();
```

現在のプロセス内にあるデフォルト 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，デフォルト ORB の ORB::resolve_initial_references("RootPOA") を呼び出したときの戻り値と同じです。PortableServer::ServantBase の派生クラスでは，必要に応じてこのメソッドをオーバーライドし，POA を選択して返すこともできます。

```
CORBA::InterfaceDef_ptr _get_interface();
```

このオブジェクトのインターフェース定義へのポインタを返します。詳細については，「インターフェースリポジトリのインターフェースとクラス」の第 5 章「インターフェースリポジトリのインターフェースとクラス」を参照してください。

```
CORBA::Boolean _is_a(const char *rep_id);
```

このサーバントが指定されたリポジトリ ID に関連付けられているインターフェースを実装している場合は，TRUE を返します。そうでない場合は，FALSE を返します。

パラメータ	説明
rep_id	チェック対象のリポジトリ識別子。

```
void _remove_ref();
```

このサーバントの参照カウントを削除します。デフォルトのインプリメンテーションは何も実行しないため，派生クラスでこのメソッドをオーバーライドし，参照カウント機能を提供する必要があります。

PortableServer::ServantLocator

```
class PortableServer::ServantLocator : public
      PortableServer::ServantManager
```

POA が NON_RETAIN ポリシーを持つ場合，POA はサーバントマネージャとして PortableServer::ServantLocator オブジェクトを使用します。このサーバントマネージャから返されるサーバントは，単一の要求に対してのみ使用されます。

POA は，このサーバントマネージャから返されるサーバントが単一の要求に対してのみ使用されることを認識しているので，サーバントマネージャのメソッドに追加情報を提供できます。この 2 つのメソッドは PortableServer::ServantLocator サーバントマネージャとは異なる動作を行います。

インクルードファイル

この構造体を使用する場合は，**poa_c.hh** ファイルをインクルードする必要があります。

PortableServer::ServantLocator のメソッド

```
PortableServer::Servant preinvoke(const PortableServer::ObjectId& oid,
      PortableServer::POA_ptr adapter,const char* operation,
      Cookie& the_cookie);
```

NON_RETAIN ポリシーと USE_SERVANT_MANAGER ポリシーが設定されているとき，POA が現在アクティブでないオブジェクトに対する要求を受信するたびに POA によって呼び出されます。

サーバントマネージャのインプリメンテーションを提供するのはユーザーです。このインプリメンテーションにより，可能な場合は指定された oid 値に対応する適切なサーバントを検索および作成します。

パラメータ	説明
oid	着信した要求に関連付けられている ObjectId 値。
adapter	そのスコープ内でオブジェクトがアクティブ化される POA。
operation	サーバントが戻ったときに POA によって呼び出されるオペレーションの名前。
the_cookie	サーバントマネージャによって設定され，後で postinvoke メソッドで使用される不透過値。

```
void postinvoke(const PortableServer::ObjectId& oid,
      PortableServer::POA_ptr adapter, const char* operation,
      Cookie the_cookie, PortableServer::Servant the_servant)
```

POA が NON_RETAIN ポリシーと USE_SERVANT_MANAGER ポリシーを持つとき，サーバントが要求を完了するたびに呼び出されます。このメソッドは，オブジェクトに対する要求の一部とみなすことができます。つまり，メソッドが正常に終了しても，postinvoke が例外を生成した場合は，メソッドの正常な戻り値が上書きされ，要求は例外付きで完了します。

POA に既知のサーバントを廃棄すると，予期しない結果が生じる可能性があります。

パラメータ	説明
oid	着信した要求に関連付けられている ObjectId 値。
adapter	そのスコープ内でオブジェクトがアクティブ化される POA。
operation	サーバントが戻ったときに POA によって呼び出されるオペレーションの名前。
the_cookie	preinvoke メソッド内でサーバントマネージャによって設定され，このメソッド内で使用される不透過値。
the_servant	オブジェクトに関連付けられているサーバント。

PortableServer::ServantManager

```
class PortableServer::ServantManager
```

サーバントマネージャは，ポータブルオブジェクトアダプタ（POA）に関連付けられます。サーバントマネージャを使用すると，POA は，アクティブでないオブジェクトに向けられた要求を受信したとき，オンデマンドでそのオブジェクトをアクティブ化できます。

PortableServer::ServantManager クラスにメソッドはなく，ほかの 2 つのクラス PortableServer::ServantActivator と Portableserver::ServantLocator のベースクラスになります。詳細については，40 ページの「PortableServer::ServantActivator」および 42 ページの「PortableServer::ServantLocator」を参照してください。どちらのクラスが使用されるかは，POA のポリシーに依存します。RETAIN は PortableServer::ServantActivator に対応し，NON_RETAIN は Portableserver::ServantLocator に対応します。

インクルードファイル

この構造体を使用する場合は，**poa_c.hh** ファイルをインクルードする必要があります。

SystemException

```
class CORBA::SystemException : public CORBA::Exception
```

SystemException クラスは，VisiBroker ORB またはオブジェクトインプリメンテーションが検出した標準システムエラーを報告するために使用されます。このクラスは，Exception クラスの派生クラスです。このクラスは，例外の名前と詳細を出力ストリームに出力するためのメソッドを提供します。18 ページの「Exception」を参照してください。

SystemException オブジェクトは，例外の発生元のオペレーションが完了したかどうかを示す完了状態を保持します。SystemException オブジェクトはマイナーコードも保持します。マイナーコードは設定および取得することができます。

インクルードファイル

このクラスを使用するには，**corba.h** ファイルをインクルードする必要があります。

SystemException のメソッド

```
CORBA::SystemException(CORBA::ULong minor = 0,
        CORBA::CompletionStatus status = CORBA::COMPLETED_NO);
```

指定されたプロパティを持つ SystemException オブジェクトを作成します。

パラメータ	説明
minor	マイナーコード。
status	完了状態。CORBA::COMPLETED_YES，CORBA::COMPLETED_NO，CORBA::COMPLETED_MAYBE のいずれかです。

```
CORBA::CompletionStatus completed() const;
```

このオブジェクトの完了状態が COMPLETED_YES に設定されている場合は，TRUE を返します。

```
void completed(CORBA::CompletionStatus status);
```

このオブジェクトの完了状態を設定します。

パラメータ	説明
status	完了状態。COMPLETED_YES，COMPLETED_NO，COMPLETED_MAYBE のいずれかです。

```
CORBA::ULong minor() const;
```

このオブジェクトのマイナーコードを返します。

```
void minor(CORBA::ULong val);
```

このオブジェクトのマイナーコードを設定します。

パラメータ	説明
val	マイナーコード。

```
static CORBA::SystemException *_downcast(CORBA::Exception *exc);
```

指定された Exception ポインタを SystemException ポインタにダウンキャストします。指定されたポインタが SystemException オブジェクトまたはその派生オブジェクトをポ

イントする場合は，そのオブジェクトへのポインタが返されます。指定されたポインタが SystemException オブジェクトまたはその派生オブジェクトをポイントしない場合は，NULL ポインタが返されます。

パラメータ	説明
exc	ダウンキャストする Exception ポインタ。

メモ このメソッドにより，Exception オブジェクトの参照カウントがインクリメントされることはありません。

例外の名前	説明
BAD_INV_ORDER	ルーチン呼び出し順序の不正。
BAD_OPERATION	無効なオペレーション。
BAD_CONTEXT	コンテキストオブジェクトの処理エラー。
BAD_PARAM	無効なパラメータが渡された。
BAD_TYPECODE	無効なタイプコード。
COMM_FAILURE	通信の障害。
DATA_CONVERSION	データ変換のエラー。
FREE_MEM	メモリを解放できない。
IMP_LIMIT	インプリメンテーションの制限の違反。
INITIALIZE	ORB の初期化エラー。
INTERNAL	ORB の内部エラー。
INTF_REPOS	インターフェースリポジトリへのアクセスエラー。
INV_FLAG	無効なフラグが指定された。
INV_INDENT	識別子の構文が不正。
INV_OBJREF	無効なオブジェクトリファレンスが指定された。
MARSHAL	パラメータまたは結果のマーシャリングエラー。
NO_IMPLEMENT	オペレーションのインプリメンテーションを利用できない。
NO_MEMORY	動的なメモリ割り当てのエラー。
NO_PERMISSION	要求されたオペレーションを実行する権限がない。
NO_RESOURCES	リソース不足のために要求を処理できない。
NO_RESPONSE	要求に対する有効な応答がまだない。
OBJ_ADAPTOR	オブジェクトアダプタによってエラーが検出された。
OBJECT_NOT_EXIST	オブジェクトが利用できない。
PERSIST_STORE	永続的ストレージの障害。
TRANSIENT	一時的な障害。
UNKNOWN	未知の例外。

UserException

```
class CORBA::UserException : public CORBA::Exception
```

UserException は，オブジェクトインプリメンテーションから生成されるユーザー例外を派生させるためのベースクラスとして使用されます。これは，Exception の派生クラスです。

第 4 章

動的なインターフェースとクラス

ここでは，クライアントアプリケーションが使用する動的起動インターフェース，およびオブジェクトサーバーが使用する動的スケルトンインターフェースをサポートするクラスについて説明します。

Any

CORBA::Any クラスは，1 つの IDL 型を表し，その値をタイプセーフな方法で渡すために使用されます。このクラスのオブジェクトは，オブジェクトの型を定義する TypeCode へのポインタと，このオブジェクトに関連付けられている値へのポインタを持ちます。オブジェクトを構築，コピー，および廃棄するメソッドのほか，オブジェクトの値と型を初期化および照会するためのメソッドが提供されます。また，ストリームからオブジェクトを読み込んだり，ストリームにオブジェクトを書き込むための演算子も提供されます。

次のコードサンプルードは，Any オブジェクトを作成および使用する例を示します。

```
// 任意のオブジェクトを作成します
CORBA::Any  anObject;
// typecode 演算子を使用して，
// 'anObject' オブジェクトが long を格納できることを指定します
anObject <<= CORBA::_tc_long;
```

インクルードファイル

この構造体を使用する場合は，**CORBA.h** ファイルをインクルードする必要があります。

Any のメソッド

CORBA::Any();

デフォルトコンストラクタです。空の Any オブジェクトを作成します。

```
CORBA::Any(const CORBA::Any& val);
```

指定されたオブジェクトをコピーして Any オブジェクトを作成するコピーコンストラクタです。

パラメータ	説明
val	コピーするオブジェクト。

```
CORBA::Any(CORBA::TypeCode_ptr tc,
    *value, CORBA::Boolean release = 0);
```

指定された値と TypeCode で初期化した Any オブジェクトを作成するコンストラクタです。

パラメータ	説明
tc	この Any が保持する値の TypeCode。
value	この Any が保持する値。
release	TRUE に設定した場合は，この Any オブジェクトが廃棄されたときに，この Any オブジェクトの値に関連付けられているメモリが解放されます。

```
static CORBA::Any-_ptr _duplicate(CORBA::Any_ptr ptr);
```

この static メソッドは，指定されたオブジェクトの参照カウントをインクリメントし，そのオブジェクトへのポインタを返します。

パラメータ	説明
ptr	コピーする Any。

```
static CORBA::Any_ptr _nil();
```

この static メソッドは，初期化に使用される NULL ポインタを返します。

```
static void _release(CORBA::Any_ptr *ptr);
```

この static メソッドは，指定されたオブジェクトの参照カウントをデクリメントします。参照カウントが 0 になった場合は，そのオブジェクトの管理下にあるすべてのメモリが解放され，オブジェクトは削除されます。

パラメータ	説明
ptr	解放する Any。

挿入用の演算子

```
void operator<<=(CORBA::Short);
void operator<<=(CORBA::UShort);
void operator<<=(CORBA::Long);
void operator<<=(CORBA::ULong);
void operator<<=(CORBA::Float);
void operator<<=(CORBA::Double);
void operator<<=(const CORBA::Any&);
void operator<<=(const char *);
void operator<<=(CORBA::LongLong);
void operator<<=(CORBA::ULongLong);
void operator<<=(CORBA::LongDouble);
```

これらの演算子は，指定された値でこのオブジェクトを初期化し，その値に適した TypeCode を自動的に設定します。この Any オブジェクトの構築時に release フラグが

TRUE に設定された場合は，前回この Any オブジェクトに保存された値が解放されてから，新しい値が代入されます。

```
void operator<<=(CORBA::TypeCode_ptr tc);
```

指定された値を持つ TypeCode オブジェクトを初期化します。

パラメータ	説明
tc	この Any に設定する TypeCode。

抽出用の演算子

```
CORBA::Boolean operator>>=(CORBA::Short&) const;
CORBA::Boolean operator>>=(CORBA::UShort&) const;
CORBA::Boolean operator>>=(CORBA::Long&) const;
CORBA::Boolean operator>>=(CORBA::ULong&) const;
CORBA::Boolean operator>>=(CORBA::Float&) const;
CORBA::Boolean operator>>=(CORBA::Double&) const;
CORBA::Boolean operator>>=(CORBA::Any&) const;
CORBA::Boolean operator>>=(char *&) const;
CORBA::Boolean operator>>=(CORBA::LongLong&) const;
CORBA::Boolean operator>>=(CORBA::ULongLong&) const;
CORBA::Boolean operator>>=(CORBA::LongDouble&) const;
```

これらの演算子は，指定されたターゲットにこのオブジェクトの値を格納します。ターゲットの TypeCode と格納されている値の TypeCode が一致しない場合は，FALSE が返され，値は抽出されません。そうでない場合は，格納されている値がターゲットに代入され，TRUE が返されます。

```
CORBA::Boolean operator>>=(CORBA::TypeCode_ptr& tc) const;
```

オブジェクトに保存されている値の TypeCode を抽出します。

パラメータ	説明
tc	この Any の TypeCode を格納するオブジェクト。

ContextList

```
class CORBA::ContextList
```

このクラスは，処理要求に関連付けられるコンテキストのリストを保持します。詳細については，66 ページの「Request」を参照してください。

ContextList のメソッド

```
CORBA::ContextList();
```

空の Context リストを構築します。

```
~CORBA::ContextList();
```

デフォルトのデストラクタです。

```
void add(const char *ctx);
```

指定されたコンテキストをこのオブジェクトのリストに追加します。

パラメータ	説明
ctx	リストに追加するコンテキスト。

```
void add_consume(char *ctx);
```

指定されたコンテキストコードをこのオブジェクトのリストに追加します。この引数によって指定された所有権は ContextList に移ります。このメソッドを呼び出した後で，この Context にアクセスしたり，解放してはなりません。

パラメータ	説明
ctx	リストに追加するコンテキスト。

```
CORBA::ULong count() const;
```

リスト内に現在格納されている項目の数を返します。

```
const char *item(CORBA::Long index);
```

リスト内の指定されたインデックスに格納されているコンテキストへのポインタを返します。インデックスが無効な場合は，NULL ポインタが返されます。このメソッドを呼び出した後で，返されたコンテキストを解放してはなりません。コンテキストを削除する場合は，かわりに remove メソッドを使用します。

パラメータ	説明
index	返されるコンテキストの 0 ベースのインデックス。

```
void remove(CORBA::long index);
```

指定されたインデックスからコンテキストを削除します。インデックスが無効な場合，削除は行われません。

パラメータ	説明
index	削除するコンテキストの 0 ベースのインデックス。

```
static CORBA::ContextList-_ptr _duplicate(CORBA::ContextList_ptr ptr);
```

この static メソッドは，オブジェクトの参照カウントをインクリメントし，そのオブジェクトへのポインタを返します。

パラメータ	説明
ptr	コピーするオブジェクト。

```
static CORBA::ContextList_ptr _nil();
```

この static メソッドは，初期化に使用される NULL ポインタを返します。

```
static void _release(CORBA::ContextList *ptr);
```

この static メソッドは，このオブジェクトの参照カウントをデクリメントします。参照カウントが 0 になった場合は，そのオブジェクトの管理下にあるすべてのメモリが解放され，オブジェクトは削除されます。

パラメータ	説明
ptr	解放するオブジェクト。

DynamicImplementation

```
class PortableServer::DynamicImplementation : public
      PortableServer::ServantBase
```

このクラスは，IDL コンパイラが生成するスケルトンクラスのかわりに動的スケルトンインターフェースを使用するオブジェクトインプリメンテーションを派生させるためのベースクラスとして使用されます。このクラスの派生クラスでは，invoke メソッドと _primary-interface() メソッドのインプリメンテーションを提供する必要があります。

DynamicImplementation のメソッド

```
virtual void invoke(CORBA::ServerRequest_ptr request) = 0;
```

オブジェクトインプリメンテーションに対するクライアントの処理要求を受信するたびに，POA によって呼び出されます。ユーザーがこのメソッドのインプリメンテーションを提供して，ServerRequest オブジェクトの内容を検証し，要求に応答するための必要な処理を行い，クライアントにその結果を返す必要があります。ServerRequest インターフェースの詳細については，69 ページの「ServerRequest」を参照してください。

パラメータ	説明
request	クライアントの処理要求を表す ServerRequest オブジェクト。

```
virtual CORBA::RepositoryId _primary_interface(
      const PortableServer::ObjectId& oid PortableServer::POA_ptr poa) const;
```

POA によってコールバックとして呼び出されます。DynamicImplementation クラスから継承されるサーバントは，このメソッドをインプリメンテーションする必要があります。このメソッドは直接呼び出す必要があります。そうしないと，予期しない動作が発生します。ほかの条件下でこのメソッドを呼び出した場合は，予期しない結果が生じる可能性があります。_primary_interface メソッドは，入力パラメータとして ObjectId 値と POA_ptr を受け取り，その oid に対する最下位派生インターフェースを表す有効な RepositoryId を返します。

DynAny

```
class DynamicAny::DynAny : public CORBA::Pseudo Object
```

DynAny オブジェクトは，コンパイル時に定義されていなかったデータ型を実行時に作成および解釈するために，クライアントアプリケーションまたはサーバーによって使用されます。DynAny には，基本型（boolean，int，float など）または複合型（struct，union など）が含まれます。DynAny が保持する型は，オブジェクトの作成時に定義され，このオブジェクトの存続期間中は変更できません。

1 つの DynAny オブジェクトは 1 つ以上の要素からなるデータを表し，各要素がそれぞれ独自の値を持ちます。それらの要素間をナビゲートするために，next，seek，rewind，current_component の各メソッドが提供されます。

ORB::resolve_initial_references("DynAnyFactory") を呼び出すことで，DynAnyFactory が作成されます。次に，このファクトリが基本型または複合型の作成に使用されます。DynAnyFactory は DynamicAny モジュールに属します。

基本型に対する DynAny オブジェクトは，DynAnyFactory::create_dyn_any_from_type_code メソッドを使用して作成します。DynAny オブジェクトは，DynAnyFactory::create_dyn_any メソッドを使用して，Any オブジェクトから作成および初期化することもできます。

次のインターフェースは DynAny から派生し，動的に管理される構造型をサポートします。

構造型	インターフェース
配列	55 ページの「DynArray」内の DynArray
列挙体	55 ページの「DynEnum」内の DynEnum
シーケンス	56 ページの「DynSequence」内の DynSequence
構造体	57 ページの「DynStruct」内の DynStruct
共用体	58 ページの「DynUnion」内の DynUnion

インクルードファイル

このクラスを使用するには，**dynany.h** ファイルをインクルードする必要があります。

重要な使用上の制約

DynAny オブジェクトは，処理要求または DII 要求のパラメータとして使用できません。また，ORB::object_to_string メソッドを使用して外部化することもできません。ただし，DynAny::to_any メソッドを使用して DynAny オブジェクトを Any に変換すると，それをパラメータとして使用できます。

DynAny のメソッド

```
void assign(DynamicAny::DynAny_ptr dyn_any);
```

指定された DynAny を使用して，この DynAny オブジェクトの値を初期化します。

Any が保持する型がこのオブジェクトが保持する型と一致しない場合は，TypeMismatch 例外が生成されます。

```
DynamicAny::DynAny_ptr copy();
```

このオブジェクトのコピーを返します。

```
virtual CORBA::ULong component_count( );
```

DynAny 内に格納されている複合型の要素数を符号なし長整数で返します。

```
virtual DynamicAny::DynAny_ptr current_component();
```

このオブジェクト内の現在の要素を返します。

```
virtual void destroy();
```

このオブジェクトを廃棄します。

```
virtual CORBA::Boolean equal(const DynamicAny::DynAny_ptr value);
```

2 つの DynAny 値を比較し，等しいかどうかをチェックします。等しい場合は，TRUE を返します。そうでない場合は，FALSE を返します。

```
virtual void from_any(CORBA::Any& value);
```

指定された Any を使用してこのオブジェクトの現在の要素を初期化します。

Any が保持する値の TypeCode が，このオブジェクトの作成時に定義された TypeCode と一致しない場合は，型の不一致例外が生成されます。

渡された value パラメータが正しくない場合は，InvalidValue 例外が生成されます。

パラメータ	説明
value	このオブジェクトに設定する値を保持する Any オブジェクト。

```
virtual boolean next();
```

次の要素があれば，次の要素に進み，TRUE を返します。次の要素がない場合は，FALSE を返します。

```
virtual void rewind();
```

この DynAny に定義されている最初の要素を現在の要素にします。

このオブジェクトが要素を 1 つも持たない場合は，このメソッドを呼び出しても，何も効果はありません。

```
virtual CORBA::Boolean seek(CORBA::Long index);
```

指定されたインデックスの要素を現在の要素にします。指定されたインデックスに要素がない場合は，FALSE を返します。そうでない場合は，TRUE を返します。

パラメータ	説明
index	目的の要素の 0 ベースのインデックス。

```
virtual CORBA::Any* to_any( );
```

DynAny オブジェクトを Any オブジェクトに変換し，Any オブジェクトへのポインタを返します。

```
CORBA::TypeCode_ptr type();
```

DynAny に保存されている値の TypeCode を返します。

抽出用のメソッド

DynAny の抽出用のメソッドは，この DynAny オブジェクトの現在の要素に保持される型の値を返します。次に，抽出用のメソッドの名前を示します。

この DynAny が保持する値が，使用される抽出用メソッドの戻り値の型と一致しない場合は，TypeMismatch 例外が生成されます。

DynAny クラスにある抽出用のメソッドは次のとおりです。

```
virtual CORBA::Any* get_any();
virtual CORBA::Boolean get_boolean();
virtual CORBA::Char get_char();
virtual CORBA::Double get_double();
virtual DynamicAny::DynAny* get_dyn_any();
virtual CORBA::Float get_float();
virtual CORBA::Long get_long();
virtual CORBA::Long get_longlong();
virtual CORBA::Octet get_octet();
virtual CORBA::Object_ptr get_reference();
virtual CORBA::Short get_short();
virtual char* get_string();
virtual CORBA::TypeCode_ptr get_typecode();
virtual CORBA::ULong get_ulong();
virtual CORBA::UlongLong get_ulonglong();
virtual CORBA::UShort get_ushort();
virtual CORBA::ValueBase* get_val();
virtual CORBA::WChar get_wchar();
```

```
virtual CORBA::WChar* get_wstring();
Solaris   :
virtual CORBA::LongDouble get_longdouble();
```

挿入用のメソッド

挿入用のメソッドは，特定の型の値をこの DynAny オブジェクトの現在の要素にコピーします。次に，さまざまな型の挿入用のメソッドを示します。

挿入されるオブジェクトの型がこの DynAny オブジェクトの型と一致しない場合，これらのメソッドは InvalidValue 例外を生成します。

DynAny クラスにある挿入用のメソッドは次のとおりです。

```
virtual void insert_any(const CORBA:Any& value);
virtual void insert_boolean(CORBA::Boolean value);
virtual void insert_char(CORBA::char value);
virtual void insert_double(CORBA::Double value);
virtual void insert_dyn_any (DynamicAny::DynAny_ph_value);
virtual void insert_float(CORBA::Float value);
virtual void insert_long(CORBA::Long value);
virtual void insert_longlong(CORBA::LongLong value);
virtual void insert_octet(CORBA::Octet value);
virtual void insert_reference(CORBA:Object_ptr value);
virtual void insert_short(CORBA::Short value);
virtual void insert_string(const char* value);
virtual void insert_typecode(CORBA:TypeCode_ptr value);
virtual void insert_ulong(CORBA::ULong value);
virtual void insert_ulonglong(CORBA::ULongLong value);
virtual void insert_ushort(CORBA::UShort value);
virtual void insert_val(count CORBA::ValueBase& value);virtual void
insert_wchar(CORBA::WChar value);
virtual void insert_wstring(const CORBA::WChar* value);
Solaris:
virtual void insert_longdouble(CORBA::LongDouble value); Solaris only
```

DynAnyFactory

```
class DynamicAny::DynAnyFactory : public CORBA::PseudoObject
```

新しい DynAny オブジェクトを作成するのに使用する DynAnyFactory オブジェクト。DynAnyFactory オブジェクトへのリファレンスを取得するには，ORB::resolve_initial_references("DynAnyFactory") を呼び出します。

DynAnyFactory のメソッド

```
DynAny_ptr create_dyn_any (const CORBA::Any& value);
```

指定された値の DynAny オブジェクトを作成します。

パラメータ	説明
value	指定された値の新しい DynAny オブジェクト。

```
DynAny_ptr create_dyn_any_from_type_code (CORBA::TypeCode_ptr type);
```

指定された type の DynAny オブジェクトを作成します。

パラメータ	説明
type	新規 DynAny オブジェクトの型。

DynArray

class **DynamicAny::DynArray** : public VISDynComplex

このクラスのオブジェクトは，コンパイル時に定義されていなかった配列データ型を実行時に作成および解釈するために，クライアントアプリケーションまたはサーバーによって使用されます。DynArray には基本型（boolean，int，float など）または複合型（struct，union など）が含まれています。DynArray が保持する型は，オブジェクトの作成時に定義され，このオブジェクトの存続期間中は変更できません。

DynAny から継承された next，rewind，seek，current_component の各メソッドを使用して，任意の要素にアクセスできます。

VISDynComplex クラスは，ORB が複合 DynAny 型の管理に使用するヘルパークラスです。

重要な使用上の制約

DynArray オブジェクトは，処理要求または DII 要求のパラメータとして使用できません。また，ORB::object_to_string メソッドを使用して外部化することもできません。ただし，DynAny::to_any メソッドを使用して DynArray オブジェクトを一連の Any オブジェクトに変換すると，それをパラメータとして使用できます。

DynArray のメソッド

virtual void **destroy**();

このオブジェクトを廃棄します。

CORBA::AnySeq* **get_elements**();

このオブジェクトに保存されている値を保持する Any オブジェクトのシーケンスを返します。

void **set_elements**(CORBA::AnySeq& **_value**);

DynArray の要素を value パラメータで指定されたシーケンスの要素に代入します。

DynamicAny::DynAnySeq* **get_elements_as_dyn_any**();

DynAny に保持される要素を DynAny シーケンスとして返します。

void **set_elements_as_dyn_any** (const DynamicAny::DynAnySeq& value);

指定された DynAny シーケンスからこのオブジェクトに保持される要素を設定します。

value 内の要素の数がこの DynArray 内の要素の数と異なる場合は，InvalidValue 例外が生成されます。Any の型が DynAny の TypeCode と一致しない場合は，TypeMismatch 例外が生成されます。

パラメータ	説明
_value	この DynArray に設定される値を保持する Any オブジェクトの配列。

DynEnum

class **DynamicAny::DynEnum** : public DynamicAny::DynAny

このクラスのオブジェクトは，コンパイル時に定義されていなかった列挙体の値を実行時に作成および解釈するために，クライアントアプリケーションまたはサーバーによって使用されます。

この型のオブジェクトは 1 つの要素しか保持しないため，DynEnum オブジェクトで DynAn::rewind メソッドや DynAny::next メソッドを呼び出すと，常に FALSE が返されます。

重要な使用上の制約

DynEnum オブジェクトは，処理要求または DII 要求のパラメータとして使用できません。また，ORB::object_to_string メソッドを使用して外部化することもできません。ただし，to_any メソッドを使用して DynEnum オブジェクトを Any に変換すると，それをパラメータとして使用できます。

DynEnum のメソッド

```
void from_any(const CORBA::Any& value);
```

指定された Any を使用してこのオブジェクトの値を初期化します。

Any が保持する値の TypeCode が，このオブジェクトの作成時に定義された TypeCode と一致しない場合は，Invalid 例外が生成されます。

パラメータ	説明
value	Any オブジェクト。

```
CORBA::Any* to_any();
```

現在の要素の値を保持する Any オブジェクトを返します。

```
char* get_as_string();
```

この DynEnum オブジェクトの値を文字列として返します。

```
void set_as_string(const char* value_as_string);
```

指定された文字列を使用して，この DynEnum の値を設定します。

パラメータ	説明
value_as_string	この DynEnum の値を設定するために使用される文字列。

```
CORBA::ULong get_as_ulong()
```

この DynEnum オブジェクトの値を含む符号なし長整数を返します。

```
void set_as_ulong(CORBA::ULong value_as_ulong)
```

指定された CORBA::Ulong を使用して，この DynEnum の値を設定します。

パラメータ	説明
value_as_ulong	この DynEnum の値を設定するために使用される整数。

DynSequence

```
class DynamicAny::DynSequence : public DynamicAny::DynArray
```

このクラスのオブジェクトは，コンパイル時に定義されていなかったシーケンスデータ型を実行時に作成および解釈するために，クライアントアプリケーションまたはサーバーによって使用されます。DynSequence には基本型（boolean，int，float など）または複合型（struct，union など）が含まれています。DynSequence が保持する型は，オブジェクトの作成時に定義され，このオブジェクトの存続期間中は変更できません。

next，rewind，seek，current_component の各メソッドを使用して，任意の要素にアクセスできます。

重要な使用上の制約

DynSequence オブジェクトは，処理要求または DII 要求のパラメータとして使用できません。また，ORB::object_to_string メソッドを使用して外部化することもできません。ただし，to_any メソッドを使用して DynSequence オブジェクトを Any シーケンスに変換することができます。Any オブジェクトのシーケンスをパラメータとして使用できます。

DynSequence のメソッド

```
CORBA::ULong get_length();
```

この DynSequence に保持される要素の数を返します。

```
void set_length(CORBA::ULong length);
```

この DynSequence に保持される要素の数を設定します。

現在の要素数より小さい長さを指定した場合，シーケンスは切り詰められます。

パラメータ	説明
length	この DynSequence に保持される要素の数。

```
CORBA::AnySeq * get_elements();
```

このオブジェクトに保存されている値を保持する Any オブジェクトのシーケンスを返します。

```
void set_elements (const AnySeq& _value)
```

指定された Any オブジェクトのシーケンスを使用して，このオブジェクト内の要素を設定します。

```
set _elements_as_dyn_any();
```

詳細については，55 ページの「DynArray」を参照してください。

```
get_elements_as_dyn_any();
```

詳細については，55 ページの「DynArray」を参照してください。

DynStruct

```
class DynamicAny::DynStruct :public VISDynComplex
```

このクラスのオブジェクトは，コンパイル時に定義されていなかった構造体を実行時に作成および解釈するために，クライアントアプリケーションまたはサーバーによって使用されます。

next，rewind，seek，current_component の各メソッドを使用して，任意の構造体メンバーにアクセスできます。

DynStruct オブジェクトを作成するには，DynAnyFactory::create_dyn_any_from_typecode メソッドを呼び出します。

重要な使用上の制約

DynStruct オブジェクトは，処理要求または DII 要求のパラメータとして使用できません。また，ORB::object_to_string メソッドを使用して外部化することもできません。ただし，to_any メソッドを使用して DynStruct オブジェクトを Any オブジェクトに変換すると，それをパラメータとして使用できます。

DynStruct のメソッド

```
void destroy();
```

このオブジェクトを廃棄します。

```
CORBA::FieldName current_member_name();
```

現在の要素のメンバー名を返します。

```
CORBA::TCKind current_member_kind();
```

現在の要素に関連付けられている TypeCode を返します。

```
DynamicAny::NameValuePairSeq get_members();
```

構造体のメンバーを NameValuePair オブジェクトのシーケンスとして返します。

```
void set_members(const DynamicAny::NameValuePairSeq& value);
```

NameValuePair オブジェクトの配列を使用して構造体のメンバーを設定します。

```
DynamicAny::Name DynAnyPairSeq get_members_as_dyn_any();
```

構造体のメンバーを NameDynAnyPair シーケンスとして返します。

```
void set_members_as_dyn_any(const DynamicAny::nameDynAnyPairSeq value);
```

NameDynAnyPair オブジェクトを使用して構造体のメンバーを設定します。

値シーケンスの長さが DynStruct のメンバーの数に等しくない場合は，InvalidValue 例外が生成されます。要素のタイプコードのいずれかが構造体のタイプコードと一致しない場合は，TypeMismatch 例外が生成されます。

DynUnion

```
class DynamicAny::DynUnion : public VISDynComplex
```

このインターフェースは，コンパイル時に定義されていなかった共用体を実行時に作成および解釈するために，クライアントアプリケーションまたはサーバーによって使用されます。DynUnion は，共用体のディスクリミネータと実メンバーの 2 つの要素のシーケンスを保持します。

next，rewind，seek，current_component の各メソッドを使用して，任意の要素にアクセスできます。

DynUnion オブジェクトを作成するには，DynamicAny::DynAnyFactory::create_dyn_any_from_type_code メソッドを呼び出し，union タイプコードを引数として渡します。

重要な使用上の制約

DynUnion オブジェクトは，処理要求または DII 要求のパラメータとして使用できません。また，ORB::object_to_string メソッドを使用して外部化することもできません。ただし，DynAny::to_any メソッドを使用して DynUnion オブジェクトを Any オブジェクトに変換すると，それをパラメータとして使用できます。

DynUnion のメソッド

```
DynamicAny::DynAny_ptr get_discriminator();
```

この共用体のディスクリミネータを保持する DynAny オブジェクトを返します。

```
CORBA::TCKind discriminator_kind();
```

この共用体のディスクリミネータのタイプコードを返します。

```
DynamicAny::DynAny_ptr member();
```

この共用体の 1 つのメンバーを表し，現在の要素に対応する DynAny オブジェクトを返します。

```
CORBA::TCKind member_kind();
```

この共用体の 1 つのメンバーを表し，現在の要素に対応するタイプコードを返します。

```
CORBA::FieldName member_name();
```

現在の要素のメンバー名を返します。

```
void set_discriminator (DynamicAny::DynAny_ptr value);
```

この DynUnion のディスクリミネータを指定された値に設定します。

```
void set_to_default_member();
```

ディスクリミネータを共用体のデフォルトケースの値に対応する値に設定します。

```
void set_to_no_active_member();
```

ディスクリミネータを共用体のケースラベルのいずれとも一致しない値に設定します。

```
boolean has_no_active_member();
```

共用体がアクティブなメンバーを 1 つも持たない場合，つまりディスクリミネータが明示的なケースラベルとしてリストされていない値を持つために，共用体の値がディスクリミネータだけで構成される場合は，TRUE を返します。

Environment

```
class CORBA::Environment
```

Environment クラスは，C++ 言語の例外がサポートされていないプラットフォームで，例外とユーザー例外の両方を報告したり，アクセスするために使用されます。あるインターフェースで，そのオブジェクトのメソッドからユーザー例外を生成するように指定

されている場合は，そのメソッドの明示的なパラメータとして Environment クラスが使用されます。インターフェースが何も例外を生成しない場合，Environment クラスは暗黙的なパラメータになり，例外を報告するためにだけ使用されます。Environment オブジェクトがクライアントからスタブに渡されなかった場合は，オブジェクトごとにデフォルトの Environment が使用されます。

マルチスレッドのアプリケーションは，作成されるスレッドごとに 1 つのグローバルな Environment オブジェクトを持ちます。マルチスレッドでないアプリケーションは，グローバルな Environment オブジェクトを 1 つだけ持ちます。

インクルードファイル

この構造体を使用する場合は，**corba.h** ファイルをインクルードする必要があります。

Environment のメソッド

```
CORBA::Status ORB::create_environment(COBRA::Environment_ptr& ptr);
```

新しい Environment オブジェクトの作成に使用されます。

メモ このメソッドは，CORBA 仕様に準拠するために提供されています。通常は，このクラスに提供されているコンストラクタ，または C++ の new 演算子を使用する方が簡単です。

パラメータ	説明
ptr	新しく作成されたオブジェクトをポイントするように設定されるポインタ。

```
Environment();
```

Environment オブジェクトを作成します。これは，ORB::create_environment メソッドを呼び出すことと同じです。

```
static COBRA::Environment& CORBA::current_environment();
```

この static メソッドは，アプリケーションプロセスに対するグローバルな Environment オブジェクトへの参照を返します。マルチスレッドのアプリケーションの場合は，スレッドに対するグローバルな Environment オブジェクトを返します。

```
void exception(COBRA::Exception *exp);
```

引数として渡された Exception オブジェクトを記録します。指定された Environment オブジェクトが Exception オブジェクトの所有権を持ち，Environment 自身が削除されるとき，Exception オブジェクトを削除するため，Exception オブジェクトは動的に割り当てる必要があります。このメソッドに NULL ポインタを渡すことは，Environment の clear メソッドを呼び出すことと同じです。

パラメータ	説明
exp	動的に割り当てられ，この Environment に記録される Exception オブジェクトへのポインタ。

```
CORBA::Exception *exception() const;
```

この Environment に現在記録されている Exception のポインタを返します。このメソッドから返された Exception ポインタに対して，delete を呼び出してはなりません。Exception が記録されていない場合は，NULL が返されます。

```
void clear();
```

このメソッドは，このオブジェクトが保持しているすべての Exception を削除します。このオブジェクトが例外を 1 つも保持していない場合，このメソッドの効果は何もありません。

ExceptionList

class **CORBA::ExceptionList**

処理要求によって生成される例外を表すタイプコードのリストを保持します。詳細については，66 ページの「Request」を参照してください。

ExceptionList のメソッド

CORBA::ExceptionList();

空の例外リストを構築します。

CORBA::ExceptionList(CORBA::ExceptionList& list);

コピーコンストラクタです。

パラメータ	説明
list	コピーするリスト。

~CORBA::ExceptionList();

デフォルトのデストラクタです。

void **add**(CORBA::TypeCode_ptr **tc**);

指定された例外のタイプコードをこのオブジェクトのリストに追加します。

パラメータ	説明
tc	リストに追加する例外のタイプコード。

void **add_consume**(CORBA::TypeCode_ptr **tc**);

指定された例外のタイプコードをこのオブジェクトのリストに追加します。渡された引数の所有権は，この ExceptionList に移ります。このメソッドを呼び出した後で，引数にアクセスしたり，引数を解放してはなりません。

パラメータ	説明
tc	リストに追加する例外のタイプコード。

CORBA::ULong **count**() const;

リスト内に現在格納されている項目の数を返します。

CORBA::TypeCode_ptr **item**(CORBA::Long **index**);

リスト内の指定されたインデックスに格納されている TypeCode へのポインタを返します。インデックスが無効な場合は，NULL ポインタが返されます。このメソッドを呼び出

した後で，引数にアクセスしたり，引数を解放してはなりません。リストから TypeCode を削除する場合は，remove メソッドを使用します。

パラメータ	説明
index	返されるタイプコードの 0 ベースのインデックス。

```
void remove(CORBA::long index);
```

このリストから，指定された index の TypeCode を削除します。index が無効な場合，削除は行われません。

パラメータ	説明
index	削除するタイプコードのインデックス。インデックスの先頭は 0 です。

```
static CORBA::ExceptionList-_ptr _duplicate(CORBA::ExceptionList_ptr ptr);
```

この static メソッドは，指定されたオブジェクトの参照カウントをインクリメントし，そのオブジェクトへのポインタを返します。

パラメータ	説明
ptr	コピーするオブジェクト。

```
static CORBA::ExceptionList_ptr _nil();
```

この static メソッドは，初期化に使用される NULL ポインタを返します。

```
static void _release(CORBA::ExceptionList *ptr);
```

この static メソッドは，指定されたオブジェクトの参照カウントをデクリメントします。参照カウントが 0 になった場合は，そのオブジェクトの管理下にあるすべてのメモリが解放され，オブジェクトは削除されます。

パラメータ	説明
ptr	解放するオブジェクト。

NamedValue

```
class CORBA::NamedValue
```

NamedValue クラスは，動的起動インターフェースの要求において，パラメータまたは戻り値として使用される名前と値の組を表すために使用されます。このクラスのオブジェクトは NVList で組にされます。詳細については，63 ページの「NVList」を参照してください。. 名前と値の組の値は，Any オブジェクトを使用して表されます。Request クラスについては，66 ページの「Request」で説明します。

インクルードファイル

この構造体を使用する場合は，**corba.h** ファイルをインクルードする必要があります。

NamedValue のメソッド

```
CORBA::Flags flags() const;
```

この名前／値の組の使い方を定義するフラグを返します。次のいずれかの値になります。

- ARG_IN　名前と値の組は，入力パラメータとして使用されます。
- ARG_OUT　名前と値の組は，出力パラメータとして使用されます。
- ARG_INOUT　名前と値の組は，入力パラメータおよび出力パラメータの両方で使用されます。
- IN_COPY_VALUE　ARG_INOUT フラグと組み合わせて指定され，ORB が出力パラメータをコピーすることを示します。これにより，ORB は，クライアントアプリケーションのメモリには影響を与えずに，このパラメータに関連付けられているメモリを解放できます。

```
const char *name() const;
```

このオブジェクトの名前／値の組のうち，名前部分を返します。戻り値がポイントしている格納領域を解放してはなりません。

```
CORBA::Any *value() const;
```

このオブジェクトの名前／値の組のうち，値部分を返します。戻り値がポイントしている格納領域を解放してはなりません。

```
static CORBA::NamedValue-_ptr _duplicate(CORBA::NamedValue_ptr ptr);
```

この static メソッドは，指定されたオブジェクトの参照カウントをインクリメントし，そのオブジェクトへのポインタを返します。

パラメータ	説明
ptr	コピーするオブジェクト。

```
static CORBA::NamedValue_ptr _nil();
```

この static メソッドは，CORBA::NamedValue_ptr の初期化に使用される NULL ポインタを返します。

```
static void _release(CORBA::NamedValue *ptr);
```

この static メソッドは，指定されたオブジェクトの参照カウントをデクリメントします。参照カウントが 0 になった場合は，そのオブジェクトの管理下にあるすべてのメモリが解放され，オブジェクトは削除されます。

パラメータ	説明
ptr	解放するオブジェクト。

NVList

```
class CORBA::NVList
```

NVList クラスには，NamedValue オブジェクトのリストが含まれています。詳細については，62 ページの「NamedValue」を参照してください。また，動的起動インターフェース要求に関連付けられたパラメータを渡すためにも使用されます。Request クラスについては，66 ページの「Request」で説明します。

リストに項目を追加するためのメソッドがいくつか用意されています。戻り値がポイントしている格納領域を解放してはなりません。このリストから項目を削除する場合は，必ず remove メソッドを使用してください。

インクルードファイル

この構造体を使用する場合は，**corba.h** ファイルをインクルードする必要があります。

NVList のメソッド

CORBA::NamedValue_ptr **add**(CORBA::Flags **flags**);

このリストに NamedValue オブジェクトを追加し，フラグだけを初期化します。追加されたオブジェクトの名前と値は初期化されません。NamedValue の名前と値の属性を初期化するには，返されたポインタを使用します。戻り値に関連付けられている格納領域を解放してはなりません。

パラメータ	説明
flags	NamedValue オブジェクトの使い方を示すフラグ。ARG_IN, ARG_OUT, ARG_INOUT のいずれかを指定できます。

CORBA::NamedValue_ptr **add_item**(const char ***name**, CORBA::Flags **flag**);

このリストに NamedValue オブジェクトを追加し，オブジェクトの flag と name を初期化します。NamedValue 値の属性を初期化するには，返されたポインタを使用します。

注意 戻り値に関連付けられている格納領域を解放してはなりません。

パラメータ	説明
name	名前。
flag	NamedValue オブジェクトの使い方を示すフラグ。ARG_IN, ARG_OUT, ARG_INOUT のいずれかを指定できます。

NamedValue_ptr **add_item_consume**(char ***nm**, CORBA::Flags **flag**);

nm がポイントしている格納領域の管理を NVList が引き継ぐこと以外は，add_item メソッドと同じです。このメソッドを呼び出した後は，このリストが nm をコピーおよび解放している可能性があるため，これにはアクセスできません。この項目が削除されると，それに関連付けられている格納領域は自動的に解放されます。

注意 このメソッドの戻り値に関連付けられているメモリを解放してはなりません。

パラメータ	説明
name	名前。
flag	NamedValue オブジェクトの使い方を示すフラグ。ARG_IN, ARG_OUT, ARG_INOUT のいずれかを指定する必要があります。

CORBA::NamedValue_ptr **add_value**(const char ***name**, const CORBA::Any ***value**, CORBA::Flags **flag**);

このリストに NamedValue オブジェクトを追加し，その名前，値，およびフラグを初期化します。NamedValue オブジェクトへのポインタを返します。

注意 戻り値に関連付けられている格納領域を解放してはなりません。

パラメータ	説明
name	名前。
value	値。
flag	NamedValue オブジェクトの使い方を示すフラグ。ARG_IN, ARG_OUT, ARG_INOUT のいずれかを指定できます。

NamedValue_ptr **add_value_consume**(char ***nm**, CORBA::Any ***value**, CORBA::Flags **flag**);

nm および value がポイントしている格納領域の管理を NVList が引き継ぐこと以外は，add_value メソッドと同じです。このメソッドを呼び出した後は，このリストが nm または value をコピーおよび解放している可能性があるため，これらにはアクセスできません。この項目が削除されると，それに関連付けられている格納領域は自動的に解放されます。

パラメータ	説明
nm	名前。
value	値。
flag	NamedValue オブジェクトの使い方を示すフラグ。ARG_IN, ARG_OUT, ARG_INOUT のいずれかを指定する必要があります。

```
CORBA::Long count() const;
```

リスト内の NamedValue オブジェクトの数を返します。

```
static CORBA::Boolean CORBA::is_nil(NVList_ptr obj);
```

指定された NamedValue ポインタが NULL である場合は，TRUE を返します。

パラメータ	説明
obj	チェックされるオブジェクトへのポインタ。

```
NamedValue_ptr item(CORBA::Long index);
```

このリスト内の指定された index を持つ NamedValue を返します。

注意 戻り値に関連付けられている格納領域を解放してはなりません。

パラメータ	説明
index	目的の NamedValue オブジェクトの 0 ベースのインデックス。

```
static void CORBA::release(CORBA::NVList_ptr obj);
```

この static メソッドは，指定されたオブジェクトを解放します。

パラメータ	説明
obj	解放するオブジェクト。

```
Status remove(CORBA::Long index);
```

このリスト内の指定された index にある NamedValue オブジェクトを削除します。その項目が add_item_consume または add_value_consume メソッドを使用してリストに追加された場合は，その項目が削除される前に，関連する格納領域が解放されます。

パラメータ	説明
index	NamedValue オブジェクトのインデックス。インデックスの先頭は 0 です。

```
static CORBA::NVList-_ptr _duplicate(CORBA::NVList_ptr ptr);
```

この static メソッドは，指定されたオブジェクトの参照カウントをインクリメントし，そのオブジェクトへのポインタを返します。

パラメータ	説明
ptr	コピーするオブジェクト。

```
static CORBA::NVList_ptr _nil();
```

この static メソッドは，NV_List ポインタの初期化に使用される NULL ポインタを返します。たとえば，次のように指定できます。CORBA::NV_List_ptr p = CORBA::NVList::_nil();

```
static void _release(CORBA::NVList *ptr);
```

この static メソッドは，指定されたオブジェクトの参照カウントをデクリメントします。参照カウントが 0 になった場合は，そのオブジェクトの管理下にあるすべてのメモリが解放され，オブジェクトは削除されます。

パラメータ	説明
ptr	解放するオブジェクト。

Request

```
class CORBA::Request
```

Request クラスは，動的起動インターフェースを使用して，ORB オブジェクトのオペレーションを呼び出すために，クライアントアプリケーションによって使用されます。特定の Request オブジェクトには，1 つの ORB オブジェクトが関連付けられます。Request は，その ORB オブジェクトで実行されるオペレーションを表します。Context には，引数として Context と Environment が渡されます。Environment オブジェクトは，ない場合もあります。また，要求を呼び出すためのメソッド，オブジェクトインプリメンテーションから応答を受信するためのメソッド，およびオペレーションの結果を取得するためのメソッドが提供されます。

Request オブジェクトを作成するには，Object::_create_request を使用します。詳細については，「コアインターフェースとクラス」の第 3 章「コアインターフェースとクラス」を参照してください。

Request オブジェクトは，すべての戻り値の所有権を維持するため，それらのパラメータを解放しようとしてはなりません。

インクルードファイル

この構造体を使用する場合は，**corba.h** ファイルをインクルードする必要があります。

Request のメソッド

```
CORBA::Any& add_in_arg();
```

この Request に名前なしの入力引数を追加し，Any オブジェクトへのリファレンスを返します。この Any オブジェクトを使用して，名前，型，および値を設定できます。

```
CORBA::Any& add_in_arg(const char *name);
```

この Request に名前付きの入力引数を追加し，Any オブジェクトへのリファレンスを返します。この Any オブジェクトを使用して，型および値を設定できます。

注意 このメソッドの戻り値に関連付けられているメモリを解放してはなりません。

パラメータ	説明
name	追加する input 引数の名前。

```
CORBA::Any& add_inout_arg();
```

この Request に名前なしの inout 引数を追加し，Any オブジェクトへのリファレンスを返します。この Any オブジェクトを使用して，名前，型，および値を設定できます。

```
CORBA::Any& add_inout_arg(const char *name);
```

この Request に名前付きの inout 引数を追加し，Any オブジェクトへのリファレンスを返します。この Any オブジェクトを使用して，型および値を設定できます。

パラメータ	説明
name	追加する inout 引数の名前。

```
CORBA::Any& add_out_arg();
```

この Request に名前なしの出力引数を追加し，Any オブジェクトへのリファレンスを返します。この Any オブジェクトを使用して，名前，型，および値を設定できます。

```
CORBA::Any& add_out_arg(const char *name);
```

この Request に名前付きの出力引数を追加し，Any オブジェクトへのリファレンスを返します。この Any オブジェクトを使用して，型および値を設定できます。

パラメータ	説明
name	追加する出力引数の名前。

```
CORBA::NVList_ptr arguments();
```

この要求の引数を保持する NVList オブジェクトへのポインタを返します。このポインタは，引数の値を設定したり，取得するために使用できます。NVList の詳細については，63 ページの「NVList」を参照してください。

注意 このメソッドの戻り値に関連付けられているメモリを解放してはなりません。

```
CORBA::ContextList_ptr contexts();
```

この Request に関連付けられているすべての Context オブジェクトのリストへのポインタを返します。Context クラスの詳細については，「コアインターフェースとクラス」の第 3 章「コアインターフェースとクラス」を参照してください。

注意 このメソッドの戻り値に関連付けられているメモリを解放してはなりません。

```
CORBA::Context_ptr ctx() const;
```

この要求に関連付けられている Context へのポインタを返します。

```
void ctx(CORBA::Context_ptr ctx);
```

この要求で使用される Context を設定します。Context クラスの詳細については，「コアインターフェースとクラス」の第 3 章「コアインターフェースとクラス」を参照してください。

パラメータ	説明
ctx	この要求に関連付ける Context オブジェクト。

```
CORBA::Environment_ptr env();
```

この要求に関連付けられている Environment へのポインタを返します。Environment クラスの詳細については，59 ページの「Environment」を参照してください。

```
CORBA::ExceptionList_ptr exceptions();
```

この要求が生成する可能性があるすべての例外のリストへのポインタを返します。

注意 このメソッドの戻り値に関連付けられているメモリを解放してはなりません。

```
void get_response();
```

send_deferred メソッドが呼び出された後，オブジェクトインプリメンテーションから応答を取得するために使用されます。使用可能な応答が存在しない場合，このメソッドは，応答を受信するまでクライアントアプリケーションをブロックします。

```
void invoke();
```

この要求に関連付けられている ORB オブジェクトに対して，この Request を呼び出します。このメソッドは，オブジェクトインプリメンテーションから応答を受信するまで，クライアントをブロックします。この Request は，このメソッドを呼び出す前に，ターゲットオブジェクト，オペレーション名，および引数を使用して初期化されている必要があります。

```
const char* operation() const;
```

この要求が実行するオペレーションの名前を返します。

```
CORBA::Boolean poll_response();
```

応答が受信されたかどうかを判定するために，send_deferred メソッドの後に呼び出され，ブロックしません。遅延要求への応答を受信した場合は TRUE を返し，それ以外の場合は FALSE を返します。

```
CORBA::NamedValue_ptr result();
```

オペレーションの戻り値が格納される NamedValue オブジェクトへのポインタを返します。このポインタは，オブジェクトインプリメンテーションによって要求が処理された後，戻り値を取得するために使用されます。NamedValue クラスの詳細については，62 ページの「NamedValue」を参照してください。

```
CORBA::Any& return_value();
```

この Request オブジェクトの戻り値を表す Any オブジェクトへの参照を返します。

```
void set_return_type(CORBA::TypeCode_ptr tc);
```

予期される戻り値の TypeCode を設定します。invoke メソッド，またはいずれかの send メソッドを使用する前に，戻り値の型を設定する必要があります。

パラメータ	説明
tc	戻り値の型。

```
void send_deferred();
```

invoke メソッドと同様に，このメソッドはこの Request をオブジェクトインプリメンテーションに送ります。invoke メソッドとは異なり，このメソッドは，応答を待機してブロックしません。クライアントアプリケーションは，get_response メソッドを使用して応答を取得できます。

```
void send_oneway();
```

一方向オペレーションとして Request を呼び出します。また，オブジェクトインプリメンテーションからクライアントアプリケーションに応答が送信されることはありません。

```
CORBA::Object_ptr target() const;
```

このオブジェクトがオペレーションを要求するターゲットオブジェクトへの参照を返します。

```
static CORBA::Request-_ptr _duplicate(CORBA::Request_ptr ptr);
```

この static メソッドは，指定されたオブジェクトの参照カウントをインクリメントし，そのオブジェクトへのポインタを返します。

パラメータ	説明
ptr	コピーするオブジェクト。

```
static CORBA::Request_ptr _nil();
```

この static メソッドは，CORBA::Request_ptr の初期化に使用される NULL ポインタを返します。

```
static void _release(CORBA::Request *ptr);
```

この static メソッドは，指定されたオブジェクトの参照カウントをデクリメントします。参照カウントが 0 になった場合は，そのオブジェクトの管理下にあるすべてのメモリが解放され，オブジェクトは削除されます。

パラメータ	説明
ptr	解放するオブジェクト。

ServerRequest

ServerRequest クラスは，動的スケルトンインターフェースを使用しているオブジェクトインプリメンテーションによって受信された処理要求を表すために使用されます。POA は，クライアントの処理要求を受信すると，オブジェクトインプリメンテーションの invoke メソッドを呼び出し，この型のオブジェクトを渡します。

このクラスは，要求されたオペレーションと引数をオブジェクトインプリメンテーションが判定するために必要なメソッドを提供します。また，戻り値を設定するためのメソッドと，例外をクライアントアプリケーションに反映するためのメソッドも提供します。

このクラスのメソッドの戻り値に関連付けられているメモリを解放してはなりません。

インクルードファイル

このクラスを使用するには，**corba.h** ファイルをインクルードする必要があります。

ServerRequest のメソッド

```
void arguments(CORBA::NVList_ptr param);
```

この要求のパラメータのリストを設定します。

パラメータ	説明
params	記入先のパラメータリスト。このリストは，このメソッドを呼び出すより前に，適切な数の Any オブジェクトを使用して初期化し，それらの型とフラグ値を設定する必要があります。

```
CORBA::Context_ptr ctx()
```

要求に関連付けられている Context を返します。

注意 このメソッドの戻り値に関連付けられているメモリを解放してはなりません。

```
void exception(CORBA::Any_ptr exception);
```

指定された例外をクライアントアプリケーションに反映するために使用されます。

パラメータ	説明
exception	生成された例外。ポインタが NULL の場合は，CORBA::UnknownUserException が反映されます。

```
const char *operation() const;
```

要求されるオペレーションの名前を返します。

```
const char* op_name() const
```

この要求に関連付けられているオペレーション名を返します。オブジェクトインプリメンテーションは，この名前を使用して要求が有効かどうかを判定し，要求に応答するために必要な処理を行い，クライアントに適切な値を返します。

```
void params(CORBA::NVList_ptr params);
```

適切な数の Any オブジェクトを使用して初期化された NVList オブジェクトを受け取り，それにクライアントから提供されたパラメータで NVList を記入します。

パラメータ	説明
params	記入先のパラメータリスト。このリストは，このメソッドを呼び出すより前に，適切な数の Any オブジェクトを使用して初期化し，それらの型とフラグ値を設定する必要があります。

```
void result(CORBA::Any_ptr result);
```

クライアントアプリケーションに反映される結果を設定します。

パラメータ	説明
result	戻り値を表す Any オブジェクト。

```
void set_exception(const CORBA::Any& a);
```

クライアントアプリケーションに反映される例外を設定します。

パラメータ	説明
a	例外を表す Any オブジェクト。

```
void set_result(const CORBA::Any& a);
```

クライアントアプリケーションに反映される結果を設定します。

パラメータ	説明
a	戻り値を表す Any オブジェクト。

```
static CORBA::ServerRequest-_ptr _duplicate(CORBA::ServerRequest_ptr ptr);
```

この static メソッドは，指定されたオブジェクトの参照カウントをインクリメントし，そのオブジェクトへのポインタを返します。

パラメータ	説明
ptr	コピーするオブジェクト。

```
static CORBA::ServerRequest_ptr _nil();
```

この static メソッドは，初期化に使用される NULL ポインタを返します。

```
static void _release(CORBA::ServerRequest *ptr);
```

この static メソッドは，指定されたオブジェクトの参照カウントをデクリメントします。参照カウントが 0 になった場合は，そのオブジェクトの管理下にあるすべてのメモリが解放され，オブジェクトは削除されます。

パラメータ	説明
ptr	解放するオブジェクト。

TCKind

```
enum TCKind
```

この列挙体は，TypeCode オブジェクトが表すさまざまな型を記述します。72 ページの「TypeCode」を参照してください。

次の表に値を示します。

名前	意味
tk_abstract_interface	抽象インターフェース
tk_alias	エリアス
tk_any	Any
tk_array	配列
tk_boolean	boolean
tk_char	char
tk_double	double
tk_enum	列挙体
tk_except	例外
tk_fixed	Fixed 型
tk_float	float
tk_long	long
tk_longdouble	long double
tk_longlong	long long
tk_native	ネイティブ型
tk_null	NULL
tk_objref	オブジェクトリファレンス
tk_octet	Octet 文字列
tk_Principal	Principal
tk_sequence	sequence
tk_short	short
tk_string	文字列
tk_struct	構造体
tk_TypeCode	TypeCode
tk_ulonglong	unsigned long long
tk_union	共用体

名前	意味
tk_ulong	unsigned long
tk_ushort	unsigned short
tk_value	値
tk_value_box	値ボックス
tk_void	void
tk_wchar	Unicode 文字
tk_wstring	Unicode 文字列

TypeCode

class **CORBA::TypeCode**

TypeCode クラスは，IDL で定義されるさまざまな型を表します。タイプコードは，通常，Any オブジェクトに保存されている値の型を定義するために使用されます。Any の詳細については，47 ページの「Any」を参照してください。タイプコードをパラメータとしてメソッド呼び出しに渡すこともできます。

TypeCode オブジェクトは，さまざまな CORBA::ORB.create_<type>_tc メソッドを使用して作成できます。詳細については，「*コアインターフェースとクラス*」の第 3 章「コアインターフェースとクラス」を参照してください。以下に示すコンストラクタを使用することもできます。

インクルードファイル

この構造体を使用する場合は，**corba.h** ファイルをインクルードする必要があります。

TypeCode のコンストラクタ

CORBA::TypeCode(CORBA::TCKind **kind**, CORBA::Boolean **is_constant**);

追加のパラメータを必要としない型の TypeCode オブジェクトを構築します。kind がこのコンストラクタに対して有効な型でない場合は，BAD_PARAM 例外が生成されます。

パラメータ	説明
kind	表されるオブジェクトの型を記述します。次のいずれかの値です。CORBA::tk_null, CORBA::tk_void, CORBA::tk_short, CORBA::tk_long, CORBA::tk_ushort, CORBA::tk_ulong, CORBA::tk_float, CORBA::tk_double, CORBA::tk_boolean, CORBA::tk_char, CORBA::tk_octet, CORBA::tk_any, CORBA::tk_TypeCode, CORBA::tk_Principal, CORBA::tk_longlong, CORBA::tk_ulonglong, CORBA::tk_longdouble, or CORBA::tk_wchar, CORBA::tk_fixed, CORBA::tk_value, CORBA::tk_value_box, CORBA::native, CORBA::tk_abstract_interface.
is_constant	TRUE の場合は，TypeCode オブジェクトが定数であることを示します。そうでない場合，TypeCode オブジェクトは定数ではありません。

TypeCode のメソッド

CORBA::TypeCode_ptr **content_type**() const;

シーケンスまたは配列内の要素の TypeCode を返します。エリアスの型も返されます。このオブジェクトの種類が CORBA::tk_sequence，CORBA::tk_array，CORBA::tk_alias のいずれでもない場合は，BadKind 例外が生成されます。

CORBA::Long **default_index**() const;

1 つの union を示す TypeCode のデフォルトのインデックスを返します。このオブジェクトの種類が CORBA::tk_union でない場合は，BadKind 例外が生成されます。

```
CORBA::TypeCode_ptr discriminator_type() const;
```

1 つの union を示す TypeCode のディスクリミネータの型を返します。このオブジェクトの種類が CORBA::tk_union でない場合は，BadKind 例外が生成されます。

```
CORBA::Boolean equal(CORBA::TypeCode_ptr tc) const;
```

このオブジェクトと指定された TypeCode を比較します。それらがすべての点で一致する場合は，TRUE が返されます。そうでない場合は，FALSE を返します。

パラメータ	説明
tc	このオブジェクトと比較するオブジェクト。

```
const char* id() const;
```

このオブジェクトが表す型のリポジトリ ID を返します。表される型にリポジトリ ID がない場合は，BadKind 例外が生成されます。リポジトリ ID を持つ型は次のとおりです。

- CORBA::tk_struct
- CORBA::tk_union
- CORBA::tk_enum
- CORBA::tk_alias
- CORBA::tk_except
- CORBA::tk_objref

```
CORBA::TCKind kind() const
```

このオブジェクトの種類を返します。

```
CORBA::ULong length() const;
```

このオブジェクトが表す文字列，シーケンス，または配列の長さを返します。文字列の長さは文字数になります。配列またはシーケンスの長さは要素の数になります。このオブジェクトの種類が CORBA::tk_string，CORBA::tk_sequence，CORBA::tk_array のいずれでもない場合は，BadKind 例外が生成されます。

```
CORBA::ULong member_count() const;
```

この TypeCode オブジェクトが表す型のメンバーの数を返します。表される型にメンバーがない場合は，BadKind 例外が生成されます。メンバーを持つ型は次のとおりです。

- CORBA::tk_struct
- CORBA::tk_union
- CORBA::tk_enum
- CORBA::tk_except

```
CORBA::Any_ptr member_label(CORBA::ULong index) const;
```

union の TypeCode オブジェクトから，指定された index のメンバーのラベルを返します。このオブジェクトの種類が CORBA::tk_union でない場合は，BadKind 例外が生成されます。インデックスが無効な場合は，Bounds 例外が生成されます。

パラメータ	説明
index	そのラベルを取得する共同体メンバーのインデックス。インデックスの先頭は 0 です。

```
const char *member_name(CORBA::ULong index) const;
```

このオブジェクトによって表される型から，指定されたインデックスのメンバーの名前を返します。表される型にメンバーがない場合は，BadKind 例外が生成されます。インデックスが無効な場合は，Bounds 例外が生成されます。

メンバーを持つ型は次のとおりです。

- CORBA::tk_struct
- CORBA::tk_union
- CORBA::tk_enum
- CORBA::tk_except

パラメータ	説明
index	その名前を取得するメンバーの 0 ベースのインデックス。

```
CORBA::TypeCode_ptr member_type(CORBA::ULong index) const;
```

このオブジェクトによって表される型から，指定されたインデックスのメンバーの型を返します。表される型にメンバーがない場合は，BadKind 例外が生成されます。インデックスが無効な場合は，Bounds 例外が生成されます。メンバーを持つ型は次のとおりです。

- CORBA::tk_union
- CORBA::tk_except

パラメータ	説明
index	その名前を取得するメンバーの 0 ベースのインデックス。

```
const char *name() const;
```

このオブジェクトによって表される型の名前を返します。表される型が名前を持たない場合は，BadKind 例外が生成されます。名前を持つ型は次のとおりです。

- CORBA::tk_objref
- CORBA::tk_struct
- CORBA::tk_union
- CORBA::tk_enum
- CORBA::tk_alias
- CORBA::tk_except

```
static CORBA::TypeCode_ptr _duplicate(CORBA::TypeCode_ptr obj);
```

この static メソッドは，指定された TypeCode オブジェクトをコピーします。

パラメータ	説明
obj	コピーするオブジェクト。

```
static CORBA::TypeCode_ptr _nil();
```

この static メソッドは，初期化に使用される NULL TypeCode ポインタを返します。

```
static void _release(CORBA::TypeCode_ptr obj);
```

この static メソッドは，指定されたオブジェクトの参照カウントをデクリメントします。参照カウントが 0 になった場合は，管理しているすべてのメモリを解放し，そのオブジェクトを削除します。

パラメータ	説明
obj	解放するオブジェクト。

```
CORBA::Boolean equivalent (CORBA_TypeCode_ptr tc) const;
```

equivalent オペレーションは，IDL に格納されている値と型が同じであるかどうかを判定するために，ORB によって使用されます。

```
CORBA_TypeCode_ptr get_compact_typecode() const;
```

get_compact_code オペレーションは，省略可能な名前フィールドとメンバー名フィールドをすべて削除します。すべてのエリアスタイプコードはそのまま維持します。

```
virtual CORBA::Visibility member_visibility(CORBA::ULong index) const;
```

インデックスで識別される valuetype の Visibility を返します。

メモ member_visibility オペレーションは，valuebox（またはボックス化された値）ではない valuetype の TypeCode に対してのみ呼び出すことができます。

```
virtual CORBA::ValueModifier type_modifier() const;
```

type_modifier オペレーションは，ボックス化されていない valuetype の TypeCode に対してのみ呼び出すことができます。このメソッドは，ターゲットの TypeCode が表す valuetype に適用される ValueModifier を返します。

```
virtual CORBA::TypeCode_ptr concrete_base_types()
```

concrete_base_types オペレーションは，ボックス化されていない valuetype の TypeCode に対してのみ呼び出すことができます。ターゲットの TypeCode が表す値が具象ベースの valuetype を持つ場合，このメソッドは，その具象ベースの TypeCode を返します。そうでない場合は，nil TypeCode 参照を返します。

第 5 章

インターフェースリポジトリのインターフェースとクラス

ここでは，インターフェースリポジトリへのアクセスに使用されるクラスとインターフェースについて説明します。インターフェースリポジトリは，モジュールに関する情報を保持します。また，それらのモジュールが保持するインターフェースのほか，オペレーション，属性，定数などの型に関する情報も保持します。

AliasDef

```
class CORBA::AliasDef : public CORBA::TypedefDef
```

このクラスは，TypedefDef クラスから派生し，インターフェースリポジトリに保存される typedef のエリアスを表します。このクラスは，元の typedef の IDLType を設定および取得するためのメソッドを提供します。

TypedefDef クラスの詳細については，105 ページの「TypedefDef」を参照してください。IDLType クラスの詳細については，92 ページの「IDLType」を参照してください。

AliasDef のメソッド

```
CORBA::IDLType original_type_def();
```

エリアスを作成する元の typedef の IDLType を返します。

```
void original_type_def(CORBA::IDLType_ptr val);
```

エリアスを作成する元の typedef の IDLType を設定します。

パラメータ	説明
val	このエリアスに設定する IDLType。

ArrayDef

class **CORBA::ArrayDef** : public CORBA::IDLType

このクラスは，IDLType クラスから派生し，インターフェースリポジトリに保存される配列を表します。このクラスは，配列の長さを設定および取得するためのメソッドのほか，配列内の要素の型を設定および取得するためのメソッドを提供します。

ArrayDef のメソッド

CORBA::TypeCode **element_type**();

配列内の要素の TypeCode を返します。

CORBA::IDLType_ptr **element_type_def**();

この配列に保存されている要素の IDLType を返します。

void **element_type_def**(CORBA:IDLType_ptr **element_type_def**);

配列に保存されている要素の IDLType を設定します。

パラメータ	説明
element_type_def	配列内の要素の IDLType。

CORBA::ULong **length**();

配列内の要素の数を返します。

void **length**(CORBA::ULong **length**);

配列内の要素の数を設定します。

パラメータ	説明
length	配列内の要素の数。

AttributeDef

class **CORBA::AttributeDef** : public CORBA::Contained, public CORBA::Object

このクラスは，インターフェースリポジトリに保存されるインターフェースの属性を表すために使用されます。このインターフェースは，属性のモード typedef を設定および取得するためのメソッドを提供します。インターフェースの属性の型を取得するためのメソッドも提供されます。

AttributeDef のメソッド

CORBA::AttributeMode **mode**();

属性のモードを返します。戻り値は，CORBA::AttributeMode ATTR_READONLY（読み取り専用の属性の場合）または CORBA::AttributeMode ATTR_NORMAL（読み取り／書き込みの属性の場合）のいずれかになります。詳細については，79 ページの「AttributeMode」を参照してください。

void **mode**(CORBA::AttributeMode **_val**);

属性のモードを設定します。

パラメータ	説明
_val	設定するモード。

```
CORBA::TypeCode_ptr type();
```

属性の型を示す TypeCode を返します。

```
CORBA::IDLType_ptr type_def();
```

このオブジェクトの IDLType を返します。

```
void type_def(CORBA::IDLType_ptr type_def);
```

このオブジェクトの IDLType を設定します。

パラメータ	説明
type_def	このオブジェクトの IDLType。

AttributeDescription

```
struct CORBA::AttributeDescription
```

AttributeDescription 構造体は，インターフェースリポジトリに保存される属性を記述します。

AttributeDescription のメンバー

```
CORBA::Identifier_var name
```

この属性の名前。

```
CORBA::RepositoryId_var id
```

属性のリポジトリ ID。

```
CORBA::RepositoryId_var defined_in
```

この属性が定義されているインターフェースのリポジトリ ID。

```
CORBA::String_var version
```

属性のバージョン。

```
CORBA::TypeCode_var type
```

属性の IDL 型。

```
CORBA::AttributeMode mode
```

この属性のモード。

AttributeMode

```
enum CORBA::AttributeMode
```

この列挙体を使用して，属性の mode を表します。これは，読み取り専用か標準（読み取り／書き込み）のどちらかです。

AttributeMode の値

定数。	意味
ATTR_NORMAL	読み取り／書き込みの属性です。
ATTR_READONLY	読み取り専用の属性です。

ConstantDef

class **CORBA::ConstantDef** : public CORBA::Contained

このクラスを使用して，インターフェースリポジトリに保存される 1 つの定数定義を表します。このインターフェースは，定数の型と値，および typedef を設定して取得するためのメソッドを提供します。

ConstantDef のメソッド

CORBA::TypeCode_ptr **type**();

オブジェクトの型を示す TypeCode を返します。

CORBA::IDLType_ptr **type_def**();

このオブジェクトの IDLType を返します。

void **type_def**(CORBA::IDLType_ptr **type_def**);

定数の IDLType を設定します。

パラメータ	説明
type_def	この定数の IDLType。

CORBA::Any ***value**();

このオブジェクトの値を表す Any オブジェクトへのポインタを返します。

void **value**(CORBA::Any& **_val**);

この定数の値を設定します。

パラメータ	説明
_val	このオブジェクトの値を示す Any オブジェクト。

ConstantDescription

struct **CORBA::ClassName**

ConstantDescription 構造体は，インターフェースリポジトリに保存される定数を記述します。

ConstantDescription のメンバー

CORBA::Identifier_var **name**

定数の名前。

CORBA::RepositoryId_var **id**

定数のリポジトリ ID。

CORBA::RepositoryId_var **defined_in**

この定数が定義されるモジュールまたはインターフェースの名前。

CORBA::String_var **version**

定数のバージョン。

CORBA::TypeCode_var **type**

定数の IDL 型。

CORBA::Any **value**

この定数の値。

Contained

class **CORBA::Contained** : public CORBA::IRObject, public CORBA::Object

Contained クラスを使用して，それ自身が別のインターフェースリポジトリオブジェクト内に包含されたインターフェースリポジトリオブジェクトをすべて派生します。このクラスは，次の操作を行うためのメソッドを提供します。

- このオブジェクトの名前とバージョンを設定および取り出す。
- このオブジェクトを含む Container を指定する。
- このオブジェクトの絶対名，このオブジェクトを保持するリポジトリ，および記述を取得する。
- コンテナから別のコンテナにオブジェクトを移動する。

インクルードファイル

この構造体を使用する場合は，**corba.h** と **ir_c.hh** ファイルをインクルードする必要があります。

```
interface Contained: IRObject {
  attribute RepositoryId id;
  attribute Identifier name;
  attribute String_var version;

  readonly attribute Container defined_in;
  readonly attribute ScopedName absolute_name;
  readonly attribute Repository containing_Repository;
  struct Description {
   DefinitionKind kind;
    any value;
    };
  Description describe();
```

```
    void move(
     in Container new_Container,
     in Identifier new_name,
     in String_var new_version
   );
};
```

Contained のメソッド

```
CORBA::String_var absolute_name();
```

このオブジェクトを保持する Repository 内でこのオブジェクトを一意に識別する絶対名を返します。オブジェクトの作成時に設定されるオブジェクトの defined_in 属性が Repository を参照している場合，絶対名は，オブジェクト名の前に「::」を付けた名前になります。

```
CORBA::Repository_ptr containing_repository();
```

このオブジェクトを保持するリポジトリへのポインタを返します。

```
CORBA::Container_ptr defined_in();
```

このオブジェクトが定義されている Container へのポインタを返します。

```
Description* describe();
```

このオブジェクトの Description を返します。Description 構造体の詳細については，88 ページの「説明」を参照してください。

```
CORBA::String_var id();
```

このオブジェクトのリポジトリ ID を返します。

```
void id(const char *id);
```

このオブジェクトを一意に識別するリポジトリ ID を設定します。

パラメータ	説明
id	このオブジェクトのリポジトリ ID。

```
CORBA::String_var name();
```

コンテナのスコープ内でこのオブジェクトを一意に識別する名前を返します。

```
void name(const char * val);
```

このメソッドは，保持されるオブジェクトの名前を設定します。

パラメータ	説明
name	このオブジェクトの名前。

```
CORBA::String_var version();
```

このオブジェクトのバージョンを返します。バージョンは，同じ名前を持つほかのオブジェクトとこのオブジェクトを区別します。

```
void version(CORBA::String_var& val);
```

このオブジェクトのバージョンを設定します。

パラメータ	説明
val	このオブジェクトのバージョン。

```
void move(CORBA::Container_ptr new_container,
 const char *new_name,
 CORBA::String_var& new_version);
```

このオブジェクトを現在の Container から new_container に移動します。

パラメータ	説明
new_container	このオブジェクトの移動先の Container。
new_name	このオブジェクトの新しい名前。
new_version	このオブジェクトの新しいバージョン。

Container

```
class CORBA::Container : public CORBA::Container, public CORBA::Object
```

Container クラスを使用して，インターフェースリポジトリ内に包含階層を作成します。Container オブジェクトは，Contained クラスから派生したオブジェクト定義を保持しています。Repository を除く Container クラスから派生されたすべてのオブジェクト定義は，Contained クラスを継承します。

Container は，**orbtypes.h** で定義されている IDL 型（InterfaceDef，ModuleDef，ConstantDef クラスなど。ValueMemberDef クラスを除く）を作成するメソッドを提供します。作成される各定義では，defined_in 属性がこのオブジェクトをポイントするように初期化されます。

インクルードファイル

このクラスを使用するには，corba.h と ir_c.hh ファイルをインクルードする必要があります。

```
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 CORBA::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
);
```

Container のメソッド

```
CORBA::AbstractInterfaceDef_ptr create_abstract_interface(const char* _arg_id, const char*
_arg_name, const char* _arg_name, const char* _arg_version, const
CORBA_AbstractInterfaceDefSeq&_arg_base_interfaces)
```

指定された属性に基づいて，この Container に 1 つの AbstractInterfaceDef オブジェクトを作成し，新しく作成されたオブジェクトへのポインタを返します。

パラメータ	説明
id	インターフェース ID。
name	インターフェース名。
version	インターフェースのバージョン。
base_interfaces	このインターフェースが継承するすべての抽象インターフェースのリスト。

```
CORBA::ContainedSeq * contents(CORBA::DefinitionKind limit_type, CORBA::Boolean
exclude_inherited);
```

このコンテナ内に直接保持されているか，継承されているオブジェクト定義のリストを返します。このメソッドを使用すると，Repository 内のオブジェクト定義の階層間をナビゲートできます。このメソッドは，Repository 内のすべてのモジュールが保持するオブジェクト定義を返し，その後に，その各モジュール内に保持されているすべてのオブジェクト定義を返します。

パラメータ	説明
limit_type	取得するインターフェースオブジェクトの型。dk_all を指定すると，すべての型のオブジェクトが返されます。
exclude_inherited	TRUE に設定した場合，継承されたオブジェクトは返されません。

```
CORBA::AliasDef_ptr create_alias(const char * id,
 const char *name,
 const CORBA::String_var& version,
 CORBA::IDLType_ptr original_type);
```

指定された属性に基づいて，この Container に 1 つの AliasDef オブジェクトを作成し，新しく作成されたオブジェクトへのポインタを返します。

パラメータ	説明
id	エリアスの ID。
name	エリアスの名前。
version	エリアスのバージョン。
original_type	エリアスを作成する元のオブジェクトの型。

```
CORBA::ConstantDef_ptr create_constant(const char * id,
 const char *name,
 const CORBA::String_var& version,
 CORBA::IDLType_ptr type,
 const CORBA::Any& value);
```

指定された属性に基づいて，この Container に 1 つの ConstantDef オブジェクトを作成し，新しく作成されたオブジェクトへのポインタを返します。

パラメータ	説明
id	定数の ID。
name	定数の名前。
version	定数のバージョン。

パラメータ	説明
type	次に指定する値の型。
value	定数の値。

```
CORBA::EnumDef_ptr create_enum(const char * id,
const char *name, const CORBA::String_var& version,
const CORBA::EnumMemberSeq& members);
```

指定された属性に基づいて，この Container に 1 つの EnumDef オブジェクトを作成し，新しく作成されたオブジェクトへのポインタを返します。

パラメータ	説明
id	列挙体の ID。
name	列挙体の名前。
version	列挙体のバージョン。
members	列挙体のフィールドリスト。

```
CORBA::ExceptionDef_ptr create_exception(const char * id,
const char *name,
const CORBA::String_var& version,
const CORBA::StructMemberSeq& members);
```

指定された属性に基づいて，この Container に 1 つの ExceptionDef オブジェクトを作成し，新しく作成されたオブジェクトへのポインタを返します。

パラメータ	説明
id	例外の ID。
name	例外の名前。
version	例外のバージョン。
members	ある場合は，構造体のフィールドのシーケンス。

```
CORBA::InterfaceDef_ptr create_interface(const char * id, const char *name, const
CORBA::String_var& version, const CORBA::InterfaceDefSeq& base_interfaces);
```

指定された属性に基づいて，この Container に 1 つの InterfaceDef オブジェクトを作成し，新しく作成されたオブジェクトへのポインタを返します。

パラメータ	説明
id	インターフェースの ID。
name	インターフェースの名前。
version	インターフェースのバージョン。
base_interfaces	このインターフェースが継承するすべてのインターフェースのリスト。

```
CORBA::ModuleDef_ptr create_module(const char * id,
 const char *name,
 const CORBA::String_var& version);
```

指定された属性に基づいて，この Container に 1 つの ModuleDef オブジェクトを作成し，新しく作成されたオブジェクトへのポインタを返します。

パラメータ	説明
id	モジュールの ID。
name	モジュールの名前。
version	モジュールのバージョン。

```
CORBA::StructDef_ptr create_struct(const char * id,
 const char *name,
 const CORBA::String_var& version,
 const CORBA::StructMemberSeq& members);
```

指定された属性に基づいて，この Container に 1 つの StructureDef オブジェクトを作成し，新しく作成されたオブジェクトへのポインタを返します。

パラメータ	説明
id	構造の ID。
name	構造体の名前。
version	構造体のバージョン。
members	構造体のフィールドのシーケンス。

```
CORBA::UnionDef_ptr create_union(const char * id,
 const char *name,
 const CORBA::String_var& version,
 CORBA::IDLType_ptr discriminator_type,
 const CORBA::UnionMemberSeq& members);
```

指定された属性に基づいて，この Container に 1 つの UnionDef オブジェクトを作成し，新しく作成されたオブジェクトへのポインタを返します。

パラメータ	説明
id	Union の ID。
name	Union の名前。
version	Union のバージョン。
discriminator_type	Union のディスクリミナント値の型。
members	各 Union のフィールドのシーケンス。

```
CORBA::DescriptionSeq * describe_contents(CORBA::DefinitionKind limit_type,
CORBA::Boolean exclude_inherited, CORBA::Long max_returned_objs);
```

このコンテナに直接保持されるか，このコンテナに継承されたすべての定義の記述を返します。

パラメータ	説明
limit_type	その記述を取得するインターフェースオブジェクトの型。dk_all を指定すると，すべての型のオブジェクトの記述が返されます。
exclude_inherited	true に設定した場合，継承されたオブジェクトの記述は返されません。
max_returned_objs	取得する記述の最大数。このパラメータを -1 に設定すると，すべてのオブジェクトが返されます。

```
CORBA::Contained_ptr lookup(const char *search_name);
```

スコープ付きの名前を基に，このコンテナに関する定義を検索します。「::」で始まる絶対スコープ付きの名前を指定すると，このコンテナを含むリポジトリ内で定義が検索されます。オブジェクトが見つからなかった場合は，NULL 値が返されます。

パラメータ	説明
search_name	このオブジェクトのインターフェース名。

```
CORBA::ContainedSeq * lookup_name(const char *search_name, CORBA::Long levels_to_search,
CORBA::DefinitionKind limit_type, CORBA::Boolean exclude_inherited);
```

特定のオブジェクト内で，名前を基にオブジェクトを検索します。この検索では，検索する階層のレベル数，オブジェクトの型，および継承されたオブジェクトを返すという条件を付けることができます。

パラメータ	説明
search_name	検索するオブジェクトの名前。
levels_to_search	階層内で検索するレベルの数。このパラメータを -1 に設定すると，すべてのレベルが検索されます。このパラメータを 1 に設定すると，このオブジェクトだけが検索されます。
limit_type	取得するインターフェースオブジェクトの型。dk_all を指定すると，すべての型のオブジェクトが返されます。
exclude_inherited	true に設定した場合，継承されたオブジェクトは返されません。

```
CORBA::ValueDef_ptr create_value(const char * id, const char *name, const char version,
CORBA::boolean is_custom, CORBA::boolean is_abstract, const CORBA::ValueDef_ptr _base_value,
CORBA::boolean is_truncatable, const CORBA::ValueDefSeq& abstract_base_values, const
CORBA::InterfaceDefSeq& supported _interfaces, const CORBA.InitializerSeq& initializers)
```

指定された属性に基づいて，この Container に 1 つの ValueDef オブジェクトを作成し，新しく作成されたオブジェクトへのリファレンスを返します。

パラメータ	説明
id	構造体のリポジトリ ID。
name	構造体の名前。
version	構造体のバージョン。
is_custom	true に設定した場合は，カスタム valuetype が作成されます。
is_abstract	true に設定した場合は，抽象 valuetype が作成されます。
base_values	サポートされるベース値のリスト。
is_truncatable	true に設定した場合は，truncatable な valuetype が作成されます。
abstract_base_values	サポートされる抽象ベース値のリスト。
supported _interfaces	サポートされるインターフェースのリスト。
initializer	この値型がサポートする初期化子のリスト。

```
CORBA::ValueBoxDef_ptr create_value_box(const char* id, const char* name,
const char* version, CORBA::IDLType_ptr original_type)
```

指定された属性を基に，この Container に 1 つの ValueBoxDef オブジェクトを作成し，新しく作成されたオブジェクトへの参照を返します。

パラメータ	説明
id	構造体のリポジトリ ID。
name	構造体の名前。
version	構造体のバージョン。
original_type	エリアスを作成する元のオブジェクトの IDL 型。

DefinitionKind

```
enum CORBA::DefinitionKind
```

DefinitionKind 列挙体の定数は，インターフェースリポジトリオブジェクトとして有効な型を定義します。

DefinitionKind の値

定数。	意味
dk_none	どれにも該当しない型。リポジトリの検索メソッドで使用されます。
dk_all	すべての有効な型。リポジトリの検索メソッドで使用されます。
dk_Alias	エリアス。
dk_Array	配列
dk_Attribute	エリアス。
dk_Constant	定数。
dk_Enum	列挙体。
dk_Exception	例外
dk_Fixed	Fixed。
dk_Interface	インターフェース
dk_Module	モジュール
dk_Native	ネイティブ。
dk_Operation	インターフェースオペレーション。
dk_Primitive	プリミティブ型（int，long など）。
dk_Repository	リポジトリ
dk_Sequence	シーケンス
dk_String	String
dk_Struct	構造体。
dk_Typedef	Typedef。
dk_Union	共用体
dk_Value	ValueType
dk_ValueBox	ValueBox
dk_ValueMember	ValueMember
dk_Wstring	Unicode 文字列

説明

struct CORBA::Container::**Description**

この構造体は，インターフェースリポジトリ内にあり，Contained クラスから派生された項目に，共通の記述を提供します。

Description のメンバー

CORBA::Contained_var **contained_object**

この構造体に保持されているオブジェクト。

CORBA::DefinitionKind **kind**

このオブジェクトの種類。

CORBA::Any **value**

このオブジェクトの値。

EnumDef

```
class CORBA::EnumDef : public CORBA::TypedefDef, public CORBA::Object
```

このクラスは，インターフェースリポジトリに保存される 1 つの列挙体を記述するために使用されます。このインターフェースは，列挙体のメンバーのリストを設定および取得するためのメソッドを提供します。

EnumDef のメソッド

```
CORBA::EnumMemberSeq *members();
```

この列挙体のメンバーのリストを返します。

```
void members(CORBA::EnumMemberSeq members);
```

この列挙体のメンバーのリストを設定します。

パラメータ	説明
members	メンバーのリスト。

ExceptionDef

```
class ExceptionDef : public CORBA::Contained
```

このクラスは，インターフェースリポジトリに保存される 1 つの例外を記述するために使用されます。このクラスは，例外の TypeCode を設定および取得するメソッドのほか，例外のメンバーのリストを設定および取得するためのメソッドも提供します。

ExceptionDef のメソッド

```
CORBA::StructMemberSeq *members();
```

例外のメンバーのリストを返します。

```
void members(CORBA:StructMemberSeq& members);
```

この例外のメンバーのリストを設定します。

パラメータ	説明
members	メンバーのリスト。

```
CORBA::TypeCode_ptr type();
```

この例外の型を示す TypeCode を返します。

ExceptionDescription

```
struct CORBA::ExceptionDescription
```

この構造体は，インターフェースリポジトリに保存される 1 つの例外を記述するために使用されます。

ExceptionDescription のメンバー

CORBA::String_var **defined_in**

この例外が定義されているモジュールまたはインターフェースのリポジトリ ID。

CORBA::String_var **id**

例外のリポジトリ ID。

CORBA::String_var **name**

例外の名前。

CORBA::TypeCode_var **type**

例外の IDL 型。

CORBA::String_var **version**

例外のバージョン。

FixedDef

CORBA::**FixedDef** public CORBA::IDLType, public CORBA::Object

このインターフェースを使用して，インターフェースリポジトリに保存される 1 つの Fixed 定義を記述します。

メソッド

CORBA::UShort digits();

Fixed 型の桁数を設定します。

void digits(CORBA::UShort _digits);

Fixed 型の属性を設定します。

CORBA::Short scale();

Fixed 型のスケールを設定します。

void scale(CORBA::Short _scale);

Fixed 型の属性を設定します。

FullInterfaceDescription

struct **CORBA::FullInterfaceDescription**

FullInterfaceDescription 構造体は，インターフェースリポジトリに保存されるインターフェースを記述します。

FullInterfaceDescription のメンバー

CORBA::String_var **Name**

インターフェースの名前。

CORBA::String_var **id**

インターフェースのリポジトリ ID。

CORBA::String_var **defined_in**

このインターフェースが定義されるモジュールまたはインターフェースの名前。

CORBA::String_var **version**

インターフェースのバージョン。

CORBA::OpDescriptionSeq **operations**

このインターフェースがサポートするオペレーションのリスト。

CORBA::AttrDescriptionSeq **attributes**

このインターフェースが保持する属性のリスト。

CORBA::RepositoryIdSeq **base_interfaces**

このインターフェースが継承するインターフェース。

CORBA::RepositoryIdSeq derived_interfaces

このインターフェースから派生されたインターフェース。

CORBA::TypeCode_var **type**

インターフェースの TypeCode。

CORBA::Boolean **is_abstract**

このインターフェースが抽象かどうかを指定します。

FullValueDescription

struct CORBA::**FullValueDescription**

この構造体を使用して，インターフェースリポジトリに保存されている完全な値定義を表します。

変数

CORBA::String_var **name**

valuetype の名前。

CORBA::String_var **id**

valuetype のリポジトリ ID。

CORBA::Boolean **is_abstract**

true に設定されている場合は，抽象 valuetype を指定します。

CORBA::Boolean **is_custom**

true に設定されている場合は，valuetype のカスタムマーシャリングを指定します。

CORBA::String_var **defined_in**

この valuetype が定義されているモジュールのリポジトリ ID。

CORBA::String_var **version**

valuetype のバージョン。

CORBA::OpDescriptionSeq **operations**

この valuetype が提供するオペレーションのリスト。

CORBA::AttrDescriptionSeq **attributes**

valuetype のメンバー属性の valuetype のリスト。

CORBA::.ValueMemberSeq **members**

初期化子の値定義。

CORBA::InitializerSeq **initializers**

初期化子の配列。

CORBA::RepositoryIdSeq **supported_interfaces**;

サポートされるインターフェースのリスト。

CORBA::RepositoryIdSeq **abstract_base_values**;

この valuetype が継承する抽象値型のリスト。

CORBA::Boolean **is_truncatable**;

true に設定されている場合は，値をベースの valuetype に安全に切り詰めることができます。

CORBA::String_var **base_values**;

この valuetype が継承する値型の記述。

CORBA::TypeCode_var **type**

この valuetype の IDL タイプコード。

IDLType

class **CORBA::IDLType** : public CORBA::IRObject, public CORBA::Object

IDLType クラスは抽象インターフェースを提供し，IDL 型を表すすべてのインターフェースリポジトリ定義に継承されます。このクラスは，オブジェクトの型を識別する Typecode を返すためのメソッドを提供します。IDLType は一意ですが，Typecode は一意ではありません。

インクルードファイル

この構造体を使用する場合は，**corba.h** と **ir_c.hh** ファイルをインクルードする必要があります。

```
interface IDLType:IRObject {

    readonly attribute TypeCode type;
};
```

IDLType のメソッド

```
CORBA::Typecode_ptr type();
```

現在の IRObject のタイプコードを返します。

InterfaceDef

```
class CORBA::InterfaceDef : public CORBA::Container, public CORBA::Contained,
    public CORBA::IDLType
```

InterfaceDef クラスは，インターフェースリポジトリに保存される ORB オブジェクトのインターフェースを定義するために使用されます。

詳細については，83 ページの「Container」，81 ページの「Contained」，および 92 ページの「IDLType」を参照してください。

インクルードファイル

この構造体を使用する場合は，**corba.h** と **ir_c.hh** ファイルをインクルードする必要があります。

```
interface InterfaceDef: Container, Contained, IDLType {

typedef sequence<RepositoryId> RepositoryIdSeq;
typedef sequence<OperationDescription> OpDescriptionSeq;
typedef sequence<AttributeDescription> AttrDescriptionSeq;
   attribute InterfaceDefSeq base_interfaces;
  attribute boolean is_abstract;
   readonly attribute InterfaceDefSeq
   derived_interfaces
boolean is_a(in RepositoryId interface_id);
   struct FullInterfaceDescription {
Identifier name;
RepositoryId id;
RepositoryId defined_in;
String_var version;
OpDescriptionSeq operations;
AttrDescriptionSeq attributes;
RepositoryIdSeq base_interfaces;
    RepositoryIdSeq derived_interfaces;
TypeCode type;
    boolean is_abstract;
};
FullInterfaceDescription describe_interface();
   AttributeDef create_attribute(
      in RepositoryId id,
      in Identifier name,
      in String_var version,
      in IDLType type,
      in CORBA::AttributeMode mode
   );
   OperationDef create_operation(
      in RepositoryId id,
      in Identifier name,
      in String_var version,
      in IDLType result,
```

```
        in OperationMode mode,
        in ParDescriptionSeq params,
         in ExceptionDefSeq exceptions,
        in ContextIdSeq contexts
    );
    struct InterfaceDescription {
        Identifier name;
        RepositoryId id;
        RepositoryId defined_in;
        String_var version;
        RepositoryIdSeq base_interfaces;
        boolean is_abstract;
    };
};
```

InterfaceDef のメソッド

```
CORBA::InterfaceDefSeq *base_interfaces();
```

このクラスが継承するインターフェースのリストを返します。

```
void base_interfaces(const CORBA::InterfaceDefSeq& val);
```

このクラスが継承するインターフェースのリストを設定します。

パラメータ	説明
val	このインターフェースが継承するインターフェースのリスト。

```
CORBA::AttributeDef_ptr create_attribute(const char * id, const char * name,
const CORBA::String_var& version, CORBA::IDLType_ptr type, CORBA::AttributeMode mode);
```

新しく作成されてこのオブジェクトに保持される AttributeDef へのポインタを返します。id，name，version，type，および mode は，指定された値に設定されます。

パラメータ	説明
id	使用するインターフェース ID。
name	使用するインターフェースの名前。
version	使用するインターフェースのバージョン。
mode	インターフェースのモード。有効な値については，79 ページの「AttributeMode」を参照してください。

```
CORBA::OperationDef_ptr create_operation(const char *id, const char *name,
CORBA::String_var& version, CORBA::IDLType_ptr result, CORBA::OperationMode mode,
const CORBA::ParDescriptionSeq& params, const CORBA::ExceptionDefSeq& exceptions,
const CORBA::ContextIdSeq& contexts);
```

指定されたパラメータを使用して，このオブジェクトに保持される新しい OperationDef を作成します。新しく作成される OperationDef の defined_in 属性は，この InterfaceDef を識別するように設定されます。

パラメータ	説明
id	このオペレーションのインターフェース ID。
name	オペレーションの名前。
version	オペレーションのバージョン。
result	このオペレーションが返す IDL 型。
mode	このオペレーションのモード（一方向または通常）。
params	このオペレーションに渡すパラメータのリスト。

パラメータ	説明
exceptions	このオペレーションが生成する例外のリスト。
contexts	コンテキストリストは，コンテキスト内に予期される値の名前で，要求とともに渡されます。

CORBA::InterfaceDef::FullInterfaceDescription ***describe_interface**();

このオブジェクトのインターフェースを示す FullInterfaceDescription を返します。

CORBA::Boolean **is_a**(const char * **interface_id**);

このインターフェースが，指定されたインターフェースと同じか，指定されたインターフェースを直接または間接に継承している場合は，true を返します。

パラメータ	説明
interface_id	このインターフェースと比較するインターフェースの ID。

InterfaceDescription

struct:CORBA:: **InterfaceDescription**

この構造体は，インターフェースリポジトリに保存されるオブジェクトを記述します。

InterfaceDescription のメンバー

CORBA::String_var **name**

インターフェースの名前。

CORBA::String_var **id**

インターフェースのリポジトリ ID。

CORBA::String_var **defined_in**

インターフェースが定義されているリポジトリ ID の名前。

CORBA::String_var **version**

インターフェースのバージョン。

CORBA::RepositoryIdSeq **base_interfaces**

このインターフェースのベースインターフェースのリスト。

CORBA::Boolean **is_abstract**

このインターフェースが抽象かどうかを指定します。

IRObject

class **IRObject** : CORBA::Object

IRObject クラスは，インターフェースリポジトリオブジェクトによって最も広く使用されるインターフェースを提供します。Container クラス，IDLType，Contained などは，このクラスから派生します。

インクルードファイル

この構造体を使用する場合は，**corba.h** と **ir_c.hh** ファイルをインクルードする必要があります。

```
interface IRObject {
  readonly attribute DefinitionKind def_kind;
  void destroy();
};
```

IRObject のメソッド

`CORBA::DefinitionKind` **`def_kind`**`();`

このインターフェースリポジトリオブジェクトの型を返します。有効な型については，87 ページの「DefinitionKind」を参照してください。

`void` **`destroy()`**`;`

このオブジェクトをインターフェースリポジトリから削除します。このオブジェクトが Container である場合は，その内容もすべて削除されます。このオブジェクトが別のオブジェクトに保持されている場合，このオブジェクトが削除されます。destroy メソッドが PrimitiveDef または Repository オブジェクトに対して呼び出された場合は，Exception(CORBA::BAD_PARAM) が返されます。Repository クラスの詳細については，101 ページの「リポジトリ」を参照してください。

ModuleDef

`class` **`ModuleDef`** `: CORBA::Container,CORBA::Contained`

このクラスを使用して，インターフェースリポジトリ内にある 1 つの IDL モジュールを示します。

ModuleDescription

`struct` **`ModuleDescription`**

ModuleDescription 構造体は，インターフェースリポジトリに保存されるモジュールを記述します。

ModuleDescription のメンバー

`CORBA::String_var` **`name`**

モジュールの名前。

`CORBA::String_var` **`id`**

モジュールのリポジトリ ID。

`CORBA::String_var` **`defined_in`**

このモジュールが定義されているリポジトリ ID の名前。

`CORBA::String_var` **`version`**

モジュールのバージョン。

NativeDef

```
class CORBA::NativeDef
```

このインターフェースを使用して，インターフェースリポジトリに格納されているネイティブ定義を表します。

OperationDef

```
class CORBA::OperationDef : public virtual CORBA::Contained, public CORBA::Object
```

OperationDef クラスは，インターフェースリポジトリに保存されるインターフェースオペレーションに関する情報を保持します。これは，Contained の派生クラスです。詳細については，81 ページの「Contained」を参照してください。継承された describe メソッドは，オペレーションに関する詳細な情報を提供する OperationDescription 構造体を返します。

インクルードファイル

この構造体を使用する場合は，**corba.h** と **ir_c.hh** ファイルをインクルードする必要があります。

```
interface OperationDef: Contained {

  typedef sequence<ParameterDescription> ParDescriptionSeq;
  typedef Identifier ContextIdentifier;
  typedef sequence<ContextIdentifier> ContextIdSeq;
  typedef sequence<ExceptionDef> ExceptionDefSeq;
  typedef sequence<ExceptionDescription> ExcDescriptionSeq;
  readonly attribute TypeCode result;
  attribute IDLType result_def;
  attribute ParDescriptionSeq params;
  attribute CORBA::OperationMode mode;
  attribute ContextIdSeq contexts;
  attribute ExceptionDefSeq exceptions;

readonly attribute OperationKind bind;
};
struct OperationDescription {
  Identifier name;
  RepositoryId id;
  RepositoryId defined_in;
  String_var version;
  TypeCode result;
  OperationMode mode;
  ContextIdSeq contexts;
  ParDescriptionSeq parameters;
  ExcDescriptionSeq exceptions;
};
```

OperationDef のメソッド

```
CORBA::ContextIdSeq * contexts();
```

この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返します。

```
void context(const CORBA::ContextIdSeq& val);
```

この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設定します。

パラメータ	説明
val	コンテキスト識別子のリスト。

```
CORBA::ExceptionDefSeq * exceptions();
```

このオペレーションによって生成される例外の型のリストを返します。

```
void exceptions(const CORBA::ExceptionDefSeq& val);
```

このオペレーションによって生成される例外の型のリストを設定します。

パラメータ	説明
val	このオペレーションが生成する例外のリスト。

```
CORBA::OperationMode mode();
```

この OperationDef が表すこのオペレーションのモードを返します。モードは，通常または一方向です。通常モードのオペレーションは同期的に動作し，クライアントアプリケーションに値を返します。一方向オペレーションはブロックせず，オブジェクトインプリメンテーションからクライアントに送信される応答はありません。

```
void mode(CORBA::OperationMode val);
```

このオペレーションのモードを設定します。

パラメータ	説明
val	このオペレーションのモード（OP_ONEWAY または OP_NORMAL）。詳細については，99 ページの「OperationMode」を参照してください。

```
CORBA::ParDescriptionSeq * params();
```

この OperationDef に対するパラメータを記述する ParameterDescription 構造体のリストへのポインタを返します。

```
void params(const CORBA::ParDescriptionSeq& val);
```

この OperationDef の ParameterDescription のリストを設定します。構造体の順序には意味があり，このオペレーションの IDL 定義内で定義されている順序と一致している必要があります。

パラメータ	説明
val	ParameterDescription 構造体のリスト。

```
CORBA::TypeCode_ptr result();
```

この Operation が返す値の型を表す TypeCode へのポインタを返します。TypeCode は読み取り専用の属性です。

```
CORBA::IDLType_ptr result_def();
```

この OperationDef が返す IDL 型の定義へのポインタを返します。

```
void result_def(CORBA::IDLType_ptr val);
```

この OperationDef が返す型の定義を設定します。

パラメータ	説明
val	使用する型定義へのポインタ。

OperationDescription

struct CORBA::**OperationDescription**

OperationDescription 構造体は，インターフェースリポジトリに保存されるオペレーションを記述します。

OperationDescription のメンバー

CORBA::String_var **name**

このオペレーションの名前。

CORBA::String_var **id**

オペレーションのリポジトリ ID。

CORBA::String_var **defined_in**

このオペレーションが定義されているインターフェースまたは valuetype のリポジトリ ID。

CORBA::String_var **version**

オペレーションのバージョン。

CORBA::TypeCode_var **result**

このオペレーションの結果。

CORBA::OperationMode **mode**

このオペレーションのモード。

CORBA::ContextIdSeq **contexts**

このオペレーションに関連するコンテキストのリスト。

CORBA::ParameterDescriptionSeq **parameters**

オペレーションのパラメータ。

CORBA::ExceptionDescriptionSeq **exceptions**

このオペレーションが生成する例外。

OperationMode

enum CORBA:**OperationMode**

この列挙体は，オペレーションのモードを示すために使用される値を定義します。一方向か通常のどちらかです。一方向のオペレーションは，クライアントアプリケーション

が応答を期待しないオペレーションです。通常要求の場合は，要求の結果を保持する応答がオブジェクトインプリメンテーションからクライアントに送信されます。

OperationMode の値

定数。	意味
OP_NORMAL	通常オペレーションの要求。
OP_ONEWAY	一方向オペレーションの要求。

ParameterDescription

struct CORBA::**ParameterDescription**

ParameterDescription 構造体は，インターフェースリポジトリに保存されるオペレーションのパラメータを記述します。

ParameterDescription のメンバー

CORBA::String_var **name**

パラメータの名前。

CORBA::TypeCode_var **type**

パラメータの TypeCode。

CORBA::IDLType_var **type_def**

パラメータの IDL 型。

CORBA::ParameterMode **mode**

パラメータのモード。

ParameterMode

enum CORBA::**ParameterMode**

オペレーションで利用できるパラメータのモードを表す値。

ParameterMode values

定数。	意味
PARAM_IN	クライアントからサーバーへの入力に使用されます。
PARAM_OUT	サーバーからクライアントへの結果の出力に使用されます。
PARAM_INOUT	クライアントからの入力とサーバーからの出力の両方に使用されます。

PrimitiveDef

```
class PrimitiveDef : public CORBA::IDLType, public CORBA::Object
```

このクラスを使用して，インターフェースリポジトリに保存されるプリミティブ型（int や long など）を表します。このクラスは，表されるプリミティブ型の種類を取得するためのメソッドを提供します。

PrimitiveDef のメソッド

```
CORBA::PrimitiveKind kind();
```

このオブジェクトが表すプリミティブ型の種類を返します。

PrimitiveKind

```
enum CORBA::PrimitiveKind
```

PrimitiveKind 列挙体は，インターフェースリポジトリに保存されるプリミティブ型のオブジェクトを定義する定数を保持します。

PrimitiveKind の値

定数。	意味
pk_null	NULL 値。
pk_void	void。
pk_short	Short
pk_long	Long
pk_ushort	unsigned short。
pk_ulong	unsigned long。
pk_float	Float
pk_double	Double
pk_boolean	Boolean
pk_char	Character
pk_octet	Octet。
pk_any	Any
pk_TypeCode	TypeCode
pk_Principal	Principal
pk_string	String
pk_objref	オブジェクトリファレンス
pk_longlong	long long。
pk_ulonglong	unsigned long long。
pk_longdouble	long double。
pk_wchar	Unicode 文字
pk_wstring	Unicode 文字列

リポジトリ

```
class Repository : public CORBA::Container, public CORBA::Object
```

Repository クラスを使用すると，インターフェースリポジトリにアクセスできます。Repository クラスは，Container クラスから派生します。詳細については，83 ページの「Container」を参照してください。

インクルードファイル

この構造体を使用する場合は，**corba.h** と **ir_c.hh** ファイルをインクルードする必要があります。

```
interface Repository: Container {
  Contained lookup_id(in RepositoryId search_id);
  PrimitiveDef get_primitive(in CORBA::PrimitiveKind kind);
  StringDef create_string(in unsigned long bound);
  WStringDef create_wstring(in unsigned long bound)
  SequenceDef create_sequence(
     in unsigned long bound,
     in IDLType element_type
  };
  ArrayDef create_array(
     in unsigned long length,
     n IDLType element_type
  );
   FixedDef create_fixed(
      in unsigned short digits,
      in short scale
   );
;
```

Repository のメソッド

```
CORBA::ArrayDef_ptr create_array(CORBA::ULong length,
CORBA::IDLType_ptr element_type);
```

新しい ArrayDef オブジェクトを作成し，そのオブジェクトへのポインタを返します。

パラメータ	説明
length	配列内の要素の最大数。0 より大きい数を指定する必要があります。
element_type	配列内の要素の IDLType。

```
CORBA::SequenceDef_ptr create_sequence(CORBA::ULong bound,
CORBA::IDLType_ptr element_type);
```

新しい SequenceDef オブジェクトを作成し，そのオブジェクトへのポインタを返します。

パラメータ	説明
bound	シーケンス内の項目の最大数。0 より大きい数を指定する必要があります。
element_type	シーケンスの項目の IDLType へのポインタ。

```
CORBA::StringDef_ptr create_string(CORBA::Ulong bound);
```

新しい StringDef オブジェクトを作成し，そのオブジェクトへのポインタを返します。

パラメータ	説明
bound	文字列の最大の長さ。0 より大きい数を指定する必要があります。

```
CORBA::WstringDef_ptr create_wstring(CORBA::Ulong bound);
```

新しい WstringDef オブジェクトを作成し，そのオブジェクトへのポインタを返します。

パラメータ	説明
bound	文字列の最大の長さ。0 より大きい数を指定する必要があります。

```
CORBA::PrimitiveDef_ptr get_primitive(CORBA::PrimitiveKind kind);
```

PrimitiveKind への参照を返します。

パラメータ	説明
kind	取得する参照。

```
CORBA::Contained_ptr lookup_id(const char * search_id);
```

指定された検索 ID と一致するオブジェクトをインターフェースリポジトリ内で検索します。一致するものが見つからなかった場合は，NULL 値が返されます。

パラメータ	説明
search_id	検索に使用する識別子。

```
CORBA::FixedDef_ptr create_fixed(CORBA::UShort digits, CORBA::Short scale)
```

Fixed 型の桁数とスケールを設定します。

パラメータ	説明
Ushort digits	Fixed 型の桁数。
short scale	Fixed 型のスケール。

SequenceDef

```
class SequenceDef : public CORBA::IDLType, public CORBA::Object
```

このクラスを使用して，インターフェースリポジトリに保存されるシーケンスを表します。このインターフェースは，シーケンスの範囲を設定および取得するためのメソッドを提供します。

SequenceDef のメソッド

```
CORBA::ULong bound()
```

シーケンスの範囲を返します。

```
void bound(CORBA::ULong bound)
```

シーケンスの範囲を設定します。

パラメータ	説明
members	メンバーのリスト。

```
CORBA::TypeCode_ptr element_type();
```

このシーケンス内の要素の TypeCode を返します。

```
CORBA::IDLType_ptr element_type_def();
```

このシーケンス内の要素の IDL 型を返します。

```
void element_type_def(CORBA::IDLType_ptr element_type_def);
```

このシーケンス内の要素の IDL 型を設定します。

パラメータ	説明
element_type_def	要素に設定する IDL 型。

StringDef

```
class StringDef : public CORBA::IDLType, public CORBA::Object
```

このクラスは，インターフェースリポジトリに保存される 1 つの String を記述するために使用されます。このインターフェースは，文字列の範囲を設定したり，取得するためのメソッドを提供します。

StringDef のメソッド

```
CORBA::ULong bound();
```

String の範囲を返します。

```
void bound(CORBA::ULong bound);
```

String の範囲を設定します。

パラメータ	説明
bound	メンバーのリスト。

StructDef

```
class StructDef : public CORBA::TypedefDef, public CORBA::Container, public CORBA::Object
```

このクラスを使用して，インターフェースリポジトリに保存される構造体を表します。

StructDef のメソッド

```
CORBA::StructMemberSeq *members();
```

この構造体のメンバーのリストを返します。

```
void members(CORBA::StructMemberSeq& members);
```

この構造体のメンバーのリストを設定します。

パラメータ	説明
members	メンバーのリスト。

StructMember

```
struct CORBA::StructMember
```

このインターフェースは，構造体のメンバーを定義するために使用されます。定義には，名前と型を変数として使用します。

StructMember のメソッド

CORBA::String_var **name**

型の名前。

CORBA::TypeCode_var **type**

この型の IDL タイプ。

CORBA::IDLType_var **type_def**

IDL 型の IDL タイプ定義。

TypedefDef

class **TypedefDef** : public CORBA::Contained, public CORBA::IDLType, public CORBA::Object

この抽象ベースクラスは，インターフェースリポジトリに保存される 1 つのユーザー定義の構造を示します。次のインターフェースは，このインターフェースを継承します。

- AliasDef
- EnumDef
- ExceptionDef
- StructDef
- UnionDef
- WstringDef

TypeDescription

structure **TypeDescription**

TypeDescription 構造体は，インターフェースリポジトリに保存されるオペレーションの型を記述します。

TypeDescription のメンバー

CORBA::String_var **name**

型の名前。

CORBA::String_var **id**

型のリポジトリ ID。

CORBA::String_var **defined_in**

この型が定義されるモジュールまたはインターフェースの名前。

CORBA::String_var **version**

型のバージョン。

CORBA::TypeCode_var **type**

この型の IDL タイプ。

UnionDef

class **UnionDef** : public CORBA::TypedefDef, public CORBA::Container, public CORBA::Object

このクラスを使用して，インターフェースリポジトリに保存される Union を表します。このクラスは，共用体のメンバーのリストとディスクリミネータの型を設定および取得するためのメソッドを提供します。

UnionDef のメソッド

CORBA::TypeCode_ptr **discriminator_type**();

Union に対するディスクリミネータの TypeCode を返します。

CORBA::IDLType_ptr **discriminator_type_def**();

この Union のディスクリミネータの IDL 型を返します。

void **discriminator_type_def**(CORBA::IDLType_ptr **discriminator_type_def**);

この Union のディスクリミネータの IDL 型を設定します。

パラメータ	説明
discriminator_type_def	メンバーのリスト。

CORBA::UnionMemberSeq ***members**();

この Union のメンバーのリストを返します。

void **members**(CORBA::UnionMemberSeq& **members**);

この Union のメンバーのリストを設定します。

パラメータ	説明
members	メンバーのリスト。

UnionMember

struct CORBA::**UnionMember**

UnionMember 構造体は，インターフェースリポジトリに保存される Union のメンバーを記述します。

UnionMember のメンバー

CORBA::String_var **name**

Union の名前。

CORBA::Any **label**

Union のラベル。

CORBA::TypeCode_var **type**

Union のタイプコード。

CORBA::IDLType_var **type_def**

Union の IDL 型。

ValueBoxDef

class **ValueBoxDef** public CORBA::Contained, public COBRA::IDLType, public CORBA::Object

このインターフェースは，任意の IDL 型の 1 つの public メンバーを保持する単純な値型として使用されます。ValueBoxDef は，次のような ValueType を簡略化したバージョンです。

public valuetype <IDLType> value;

この宣言は，valuetype のボックス化された型 <IDLType> とほとんど同じですが，ValueBoxDef は単純な ValueTypeDef と同じではありません。

メソッド

CORBA::IDLType_ptr original_type_def();

ボックス化される型を識別します。

void original_type_def(CORBA::IDLType_ptr original_type_def);

ボックス化される型を設定します。

ValueDef

class CORBA::**ValueDef** public CORBA::Container, public CORBA::Contained, public CORBA::IDLType, public CORBA::Object

このインターフェースは，construct と呼ばれる IDL 値型を記述するために使用されます。このインターフェースはクラス型にたいへん似ています。このインターフェースは，インターフェースリポジトリ内に保存される 1 つの値定義を表します。

メソッド

CORBA::InterfaceDefSeq **supported_interfaces**( );

この値の型がサポートするインターフェースをリストします。

void supported_interfaces(const CORBA::interfaceDefSeq& **supported_interfaces**);

サポートされるインターフェースを設定します。

CORBA::InitializerSeq& **initializers( )**;

この共用体の初期化子のリストを返します。

void initializers(const CORBA::InitializerSeq& **initializers**);

初期化子を設定します。

```
CORBA.ValueDef_ptr base_value( );
```

この値が継承する値型を記述します。

```
void base_value(CORBA::ValueDef_ptr base_value);
```

値型を設定します。

```
CORBA.ValueDefSeq& abstract_base_values( );
```

この値が継承する抽象値型のリストを返します。

```
void abstract_base_values(const CORBA::ValueDef[Seq& abstract_base_values);
```

抽象値型のベース値を定義します。

```
CORBA::Boolean is_abstract( );
```

この値が抽象値型である場合は，true です。

```
void is_abstract(CORBA::Boolean is_abstract);
```

valuetype が抽象値型になるように設定します。

```
CORBA::Boolean is_custom( );
```

この値がカスタムマーシャリングを使用する場合は，true です。

```
void is_custom(CORBA::Boolean is_custom);
```

この値にカスタムマーシャリングを設定します。

```
CORBA::Boolean is_truncatable( ):
```

この値をベース値から安全に切り詰めることができる場合は，true です。

```
void is_truncatable(CORBA::Boolean is_truncatable);
```

この値に truncatable の属性を設定します。

```
CORBA::Boolean is_a(const char* value_id);
```

このメソッドを呼び出された値が，value_id パラメータで定義されるインターフェースまたは値と等しいか，それらを直接または間接的に継承する場合は，true を返します。それ以外の場合は，false が返されます。

```
CORBA::ValueDef _ptr FullValueDescription* describe_value();
```

オペレーションや属性など，この値を記述する FullValueDescription を返します。

```
CORBA::ValueMemberDef_ptr create_value_member(const Char* id, const Char* name, const Char*
version, CORBA::IDLType_ptr type_def, CORBA::short access);
```

このメソッドを呼び出された ValueDef に保持される新しい ValueMemberDef を返します。

パラメータ	説明
id	このタイプのリポジトリ ID。
name	この型の名前。
version	このオブジェクトのバージョン。
type_def	この値の IDL 型。
short access	access 値。

```
CORBA::AttributeDef_ptr create_attribute(const Char* id, const Char* name,
const Char* version, CORBA::IDLType_ptr type, CORBA::AttributeMode mode);
```

この valuetype に対する新しい属性定義を作成し，その AttributeDef を返します。

パラメータ	説明
id	このタイプのリポジトリ ID。
name	この型の名前。
version	このオブジェクトのバージョン。
type	この型の IDL タイプ。
mode	このオブジェクトのモード。

```
CORBA::OperationDef_ptr create_operation(const Char* id, const Char* name,
const Char* version, CORBA::IDLType_ptr result, CORBA::OperationMode mode,
const CORBA::ParDescriptionSeq& params, const CORBA::ExceptionDefSeq&
exceptions, const CORBA::ContextIDSeq& contexts);
```

この valuetype に対する新しい Operation を作成し，その OperationDef を返します。

パラメータ	説明
id	このタイプのリポジトリ ID。
name	この型の名前。
version	このオブジェクトのバージョン。
result	オペレーションの IDL 型。
mode	このオブジェクトのモード。
params	このオペレーションのパラメータのリスト。
例外	このオペレーションの例外のリスト。
contexts	このオペレーションのコンテキストのリスト。

ValueDescription

```
struct CORBA::ValueDescription
```

このインターフェースは，インターフェースリポジトリに格納される値型を表します。

値

```
CORBA::String_var name
```

型の名前。

```
CORBA::String_var id
```

型のリポジトリ ID。

```
CORBA::Boolean is_abstract
```

この値が抽象値型である場合，この変数は true です。

```
CORBA::Boolean is_custom
```

この valuetype がカスタムマーシャリングを使用する場合，この変数は true です。

```
CORBA::String_var defined_in.
```

この型が定義されているモジュールのリポジトリ ID。

CORBA::String_var **version**

型のバージョン。

CORBA::RepositoryIdSeq& **supported_interfaces**

この値型がサポートするインターフェースのリスト。

CORBA::RepositoryIdSeq& **abstract_base_values**

この値が継承する抽象値型のリスト。

CORBA::Boolean **is_truncatable**

true に設定されている場合は，値型をベースの値型に安全に切り詰めることができます。

CORBA::String_var **base_value**

この値の継承元の値型。

WstringDef

class **WstringDef** : public CORBA::IDLType, public CORBA::Object

このクラスは，インターフェースリポジトリに保存される 1 つの Unicode 文字列を記述するために使用されます。このクラスは，Unicode 文字列の範囲を設定したり，取得するためのメソッドを提供します。

WStringDef のメソッド

CORBA::ULong **bound**();

Wstring の範囲を返します。

void **members**(CORBA::ULong **bound**);

Wstring の範囲を設定します。

パラメータ	説明
members	メンバーのリスト。

第 6 章

アクティベーションのインターフェースとクラス

ここでは，オブジェクトインプリメンテーションのアクティブ化に使用されるインターフェースとクラスについて説明します。

ImplementationStatus

struct **ImplementationStatus**

ImplementationStatus は，OAD に登録されているサーバーのアクティブ化の状態を追跡するために使用されます。

```
module Activation
{
   . . .
   struct ImplementationStatus {
      extension::CreationImplDef   impl;
      ObjectStatusList      status;
   };
   . . .
};
```

インクルードファイル

このクラスを使用する場合は，oad_c.hh ファイルをインクルードする必要があります。

ImplementationStatus のメンバー

CreationImplDef **impl**;

オブジェクトインプリメンテーションに対する CreationImplDef。

ObjectStatusList **status**;

サーバーから提供される各オブジェクトの状態情報のリストを示します。ObjectStatusList クラスの詳細については，116 ページの「ObjectStatusList」を参照してください。

OAD

OAD インターフェースは，OAD（オブジェクトアクティベーションデーモン）へのアクセスを提供します。このインターフェースは，オブジェクトをリスト，登録，および登録解除するための管理ツールで使用します。また，OAD をプログラムから管理するために，クライアントのコードによっても使用されます。

次のコードサンプルは，OAD IDL を示しています。

```
interface OAD {
   extension::CreationImplDef create_CreationImplDef();

   Object reg_implementation(in extension::CreationImplDef impl)
         raises(DuplicateEntry, InvalidPath);

   extension::CreationImplDef get_implementation(
               in CORBA::RepositoryId repId,
               in COBRA::RepositoryId repId,
               in string object_name)
            raises(NotRegistered);

   void change_implementation(
               in extension::CreationImplDef old_info,
               in extension::CreationImplDef new_info)
            raises(NotRegistered, InvalidPath, IsActive);

   attribute boolean destroy_on_unregister;

   void unreg_implementation(
               in CORBA::RepositoryId repId,
               in string object_name)
            raises(NotRegistered);

   void unreg_interface(in CORBA::RepositoryId repId)
         raises(NotRegistered);

   void unregister_all();

   ImplementationStatus get_status(
            in CORBA::RepositoryId repId,
            in string object_name)
         raises(NotRegistered);

   ImplStatusList get_status_interface(
            in CORBA::RepositoryId repId)
         raises(NotRegistered);

   ImplStatusList get_status_all();

   Object lookup_interface(in CORBA::RepositoryId repId, in long timeout)
        raises(NotRegistered, FailedToExecute,
            NotResponding, Busy);
   Object lookup_implementation(in CORBA::RepositoryId repId, in string object_name, in
long timeout)
         raises(NotRegistered, FailedToExecute,
            NotResponding, Busy);

   extension::CreationImplDef boa_activate_obj(
               in Object obj,
               in string repository_id,
```

```
                in long unique_id)
              raises(NotRegistered);

   void boa_deactivate_obj(in Object obj,
            in string repository_id,
            in long unique_id)
                raises(NotRegistered);

   string generated_command(in extension::CreationImplDef impl);

   string generated_environment(in extension::CreationImplDef impl);

   };
```

IDL ソースコードの完全な記述については，VisiBroker インストールディレクトリの次の場所にある oad.idl ファイルを参照してください。

```
<install_dir>¥idl¥
```

インクルードファイル

このクラスを使用する場合は，oad_c.hh ファイルをインクルードする必要があります。

OAD のメソッド

```
void change_implementation(const extension::CreationImplDef&_old_info, const
extension::CreationImplDef& _new_info);
```

オブジェクトのインプリメンテーションを動的に変更します。このメソッドを使用して，登録済みのアクティブ化ポリシー，パス名，引数，および環境設定を変更できます。

パラメータ	説明
old_info	変更される情報。
new_info	古い情報にかわる情報。

例外	説明
NotRegistered	指定されたオブジェクトは登録されていません。登録されているオブジェクトを指定する必要があります。
IsActive	このオブジェクトインプリメンテーションは現在実行中です。オブジェクトを非アクティブ化してから，その情報の変更をやり直してください。

注意 現在アクティブなインプリメンテーションの情報を変更することはできません。また，このメソッドを使用すると，オブジェクトのインプリメンテーション名とオブジェクト名を変更できますが，その場合は細心の注意が必要です。このような変更を行うと，クライアントアプリケーションは元の名前を使用してオブジェクトを検索できなくなります。

```
CreationImplDef_ptr create_CreationImplDef();
```

extension::CreationImplDef_ptr オブジェクトのインスタンスを返します。その後で，「113 ページの「CreationImplDef_ptr create_CreationImplDef();」」で説明するように，その属性を設定することができます。

```
void destroy_on_unregister(CORBA::Boolean val);
```

この OAD の destroy_on_unregister 属性を設定します。

パラメータ	説明
val	TRUE に設定した場合，アクティブなインプリメンテーションは登録を解除されたときにシャットダウンされます。そうでない場合，アクティブなインプリメンテーションは，登録を解除されたときにシャットダウンされません。

メモ 現在，この属性をプログラムによって設定することはできません。

```
CORBA::Boolean destroy_on_unregister();
```

インプリメンテーションの destroy_on_unregister 属性の設定を返します。この属性が TRUE に設定された場合，アクティブなインプリメンテーションは，登録が解除されたときにシャットダウンされます。

メモ 現在，この属性をプログラムによって設定することはできません。

```
CORBA::CreationImplDef_ptr get_implementation(const char *repId, const char *object_name);
```

指定されたリポジトリ ID とオブジェクト名で登録されているインプリメンテーションに関する情報を取得します。これは，extension::CreationImplDef_ptr オブジェクトを返します。

パラメータ	説明
repId	リポジトリ識別子。
object_name	オブジェクトの名前。

例外	説明
NotRegistered	指定されたオブジェクトは登録されていません。登録されているオブジェクトを指定する必要があります。

```
ImplementationStatus *get_status(const char *repId, const char *object_name);
```

指定されたリポジトリ識別子とオブジェクト名で登録されているインプリメンテーションに関する状態情報を読み取ります。

パラメータ	説明
repId	リポジトリ識別子。
object_name	オブジェクトの名前。

```
ImplStatusList *get_status_all();
```

すべてのインプリメンテーションに関する状態情報を含む ImplStatusList を返します。

```
ImplStatusList *get_status_interface(cost char *repId);
```

指定されたリポジトリ識別子で登録されているインプリメンテーションに関する状態情報を取得します。

パラメータ	説明
repId	リポジトリ識別子。

例外	説明
NotRegistered	指定されたオブジェクトは登録されていません。登録されているオブジェクトを指定する必要があります。

```
::CORBA::Object_ptr reg_implementation(const extension::CreationImplDef& _impl);
```

インプリメンテーションを OAD と VisiBroker Edition ディレクトリサービスに登録します。

パラメータ	説明
impl	CreationImplDef のインスタンス

例外	説明
DuplicateEntry	エントリが重複するオブジェクトが指定されました。登録されていないオブジェクトを指定する必要があります。

```
void unreg_implementation(const char *repId, const char *object_name);
```

リポジトリ識別子とオブジェクト名を基にインプリメンテーションを登録解除します。destroy_on_unregister 属性が true に設定されている場合，このメソッドは，指定されたリポジトリ識別子とオブジェクト名を現在実装しているすべてのプロセスを終了します。

パラメータ	説明
repId	リポジトリ識別子。
object_name	オブジェクトの名前。

例外	説明
NotRegistered	指定されたオブジェクトは登録されていません。登録されているオブジェクトを指定する必要があります。

```
void unreg_interface(const char *repId);
```

リポジトリ識別子を基にインプリメンテーションを登録解除します。destroy_on_unregister 属性が true に設定されている場合，このメソッドは，指定されたリポジトリ識別子を現在実装しているすべてのプロセスを終了します。

パラメータ	説明
repId	リポジトリ識別子。

例外	説明
NotRegistered	指定されたオブジェクトは登録されていません。登録されているオブジェクトを指定する必要があります。

```
void unregister_all();
```

すべてのインプリメンテーションを登録解除します。属性 destroy_on_unregister が true に設定されていない限り，すべてのアクティブなインプリメンテーションは実行を続けます。

ObjectStatus

```
struct ObjectStatus
```

この構造体は，OAD に登録されているオブジェクトインプリメンテーションが提供する特定のオブジェクトに関する情報を格納するために使用され，ObjectStatusList クラスから返されます。詳細については，「116 ページの「ObjectStatusList」」を参照してください。

```
module Activation
{
   . . .
   struct ObjectStatus {
      long         unique_id;
```

```
        State         activation_state;
        Object            objRef;
    };
    . . .
};
```

インクルードファイル

このクラスを使用する場合は，oad_c.hh ファイルをインクルードする必要があります。

ObjectStatus のメンバー

CORBA::Long **unique_id**;

このオブジェクトの一意の識別子。

State **activation_state**;

このオブジェクトの現在のアクティブ化の状態。次のいずれかの値です。

- ACTIVE
- INACTIVE
- WAITING_FOR_ACTIVATION

CORBA::Object **objRef**;

この構造体の中に状態が示されているオブジェクト。

ObjectStatusList

class **ObjectStatusList**

このクラスは，ObjectStatus 構造体のリストを実装し，サーバーが提供するオブジェクトに関する情報を表すために使用されます。

参照

- 115 ページの「ObjectStatus」.

インクルードファイル

このクラスを使用する場合は，oad_c.hh ファイルをインクルードする必要があります。

ObjectStatusList のメソッド

void **length**(CORBA::ULong **len**);

リストの長さを設定します。

パラメータ	説明
len	リストの長さ。

CORBA::ULong **length**() const;

リストの長さを返します。

```
CORBA::ULong maximum() const;
```

リストの最大の長さを返します。

```
ObjectStatus& operator[](CORBA::ULong index);
```

リストにある指定インデックスを持つ ObjectStatus 構造体を返します。

パラメータ	説明
index	リストの項目のインデックス。インデックスの先頭は 0 です。

第 7 章

ネーミングサービス（VisiNaming）のインターフェースとクラス

ここでは，VisiBroker ネーミングサービス（VisiNaming）のインターフェースとクラスについて説明します。

NamingContext

```
class _VISNMEXPORT NamingContext : public virtual CORBA_Object
```

このオブジェクトは，VisiBroker ORB オブジェクトまたはほかの NamingContext オブジェクトにバインドされている名前のリストを保持および操作するために使用されます。クライアントアプリケーションはこのインターフェースを使用して，そのコンテキスト内のすべての名前を解決，または一覧表示します。オブジェクトインプリメンテーションはこのオブジェクトを使用して，オブジェクトインプリメンテーションや NamingContext オブジェクトに名前をバインドします。次のコードサンプルに，NamingContext の IDL 仕様を示します。

```
module CosNaming {
   interface NamingContext {
      void bind(in Name n, in Object obj)
         raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
      void rebind(in Name n, in Object obj)
         raises(NotFound, CannotProceed, InvalidName);
      void bind_context(in Name n, in NamingContext nc)
         raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
      void rebind_context(in Name n, in NamingContext nc)
         raises(NotFound, CannotProceed, InvalidName);
      Object resolve(in Name n)
         raises(NotFound, CannotProceed, InvalidName);
      void unbind(in Name n)
         raises(NotFound, CannotProceed, InvalidName);
      NamingContext new_context();
      NamingContext bind_new_context(in Name n)
         raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
      void destroy()
         raises(NotEmpty);
      void list(in unsigned long how_many,
```

```
                out BindingList bl,
                out BindingIterator bi);
    };
};
```

NamingContext のメソッド

```
virtual void bind(const Name& -n, CORBA::Object _ptr _obj): raises(NotFound,
CannotProceed, InvalidName, AlreadyBound);
```

指定された Object を指定された Name にバインドします。それには，最初の NameComponent に関連付けられているコンテキストを解決し，それに続く Name を使用して新しいコンテキストにそのオブジェクトをバインドします。

```
Name[NameComponent<sub>2</sub>,...,NameComponent<sub>(n-1)</sub>,NameComponent<sub>n</sub>]
```

解決してバインドするというこのプロセスは，NameComponent *(n-1)* に関連付けられているコンテキストを解決し，実際に名前とオブジェクトのバインドを保存して終了するまで，再帰的に行われます。パラメータ n が 1 つの単純な名前である場合，obj は，この NamingContext 内で n にバインドされます。

パラメータ	説明
n	オブジェクトに付ける名前で初期化された Name。
obj	名前を付けるオブジェクト。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
NotFound	Name またはその要素の 1 つが見つかりませんでした。
CannotProceed	シーケンス内の NameComponent オブジェクトの 1 つが解決できませんでした。ただし，クライアントは，返されたネーミングコンテキストでオペレーションを継続できます。
InvalidName	指定された Name に名前要素がないか，名前要素の 1 つの id フィールドが空文字列です。
AlreadyBound	bind または bind_context オペレーションにおいて，Name がすでにこの NamingContext 内の別のオブジェクトにバインドされています。

```
virtual void rebind(const Name& _n, CORBA::Object _ptr _obj) raises(NotFound, CannotProceed,
InvalidName);
```

AlreadyBound 例外が生成されないこと以外は，bind メソッドとまったく同じです。指定された Name がすでに別のオブジェクトにバインドされている場合，そのバインディングは新しいバインディングに置き換えられます。

パラメータ	説明
n	オブジェクトに付ける名前で初期化された Name 構造体。
obj	名前を付けるオブジェクト。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
NotFound	Name またはその要素の 1 つが見つかりませんでした。
CannotProceed	シーケンス内の NameComponent オブジェクトの 1 つが解決できませんでした。ただし，クライアントは，返されたネーミングコンテキストでオペレーションを継続できます。
InvalidName	指定された Name に名前要素がないか，名前要素の 1 つの id フィールドが空文字列です。

```
virtual void bind_context(const Name& _n, NamingContext_ptr _nc) raises(NotFound,
CannotProceed, InvalidName, AlreadyBound);
```

指定された Name が任意の ORB オブジェクトにではなく，NamingContext に関連付けられること以外は，bind メソッドとまったく同じです。

パラメータ	説明
n	ネーミングオブジェクトに付ける名前で初期化された Name 構造体。シーケンスの先頭から *(n-1)* 個の NameComponent 構造体で NamingContext が得られるものとします。
nc	バインドする NamingContext オブジェクト。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
NotFound	Name またはその要素の 1 つが見つかりませんでした。
CannotProceed	シーケンス内の NameComponent オブジェクトの 1 つが解決できませんでした。ただし，クライアントは，返されたネーミングコンテキストでオペレーションを継続できます。
InvalidName	指定された Name に名前要素がないか，名前要素の 1 つの id フィールドが空文字列です。
AlreadyBound	bind または bind_context オペレーションにおいて，Name がすでにこの NamingContext 内の別のオブジェクトにバインドされています。

```
virtual void rebind_context(const Name& _n, NamingContext _ptr _nc) raises(NotFound,
CannotProceed, InvalidName);
```

AlreadyBound 例外が生成されないこと以外は，bind_context メソッドとまったく同じです。指定された Name がすでに別のネーミングコンテキストにバインドされている場合，そのバインディングは新しいバインディングに置き換えられます。

パラメータ	説明
n	オブジェクトに付ける名前で初期化された Name 構造体。
nc	リバインドする NamingContext オブジェクト。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
NotFound	Name またはその要素の 1 つが見つかりませんでした。
CannotProceed	シーケンス内の NameComponent オブジェクトの 1 つが解決できませんでした。ただし，クライアントは，返されたネーミングコンテキストでオペレーションを継続できます。
InvalidName	指定された Name に名前要素がないか，名前要素の 1 つの id フィールドが空文字列です。

```
virtual CORBA::Object _ptr resolve(const Name& _n) raises(NotFound, CannotProceed,
InvalidName);
```

指定された Name を解決し，オブジェクトリファレンスを返します。パラメータ n が単純な名前の場合，この NamingContext を基準に解決します。

n が複雑な名前である場合，最初の NameComponent に関連付けられているコンテキストを使用して解決されます。次に，その新しいコンテキストを使用して続く Name が解決されます。

```
Name[NameComponent<sub>(2)</sub>,...,NameComponent<sub>(n-1)</sub>,NameComponent<sub>n</
sub>]
```

このプロセスは再帰的に継続され，**n** 番めの NameComponent に関連付けられたオブジェクトが返されて終了します。

パラメータ	説明
n	オブジェクトに付ける名前で初期化された Name 構造体。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
NotFound	Name またはその要素の 1 つが見つかりませんでした。
CannotProceed	シーケンス内の NameComponent オブジェクトの 1 つが解決できませんでした。ただし，クライアントは，返されたネーミングコンテキストでオペレーションを継続できます。
InvalidName	指定された Name に名前要素がないか, 名前要素の 1 つの id フィールドが空文字列です。

```
virtual void unbind(const Name& _n) raises(NotFound, CannotProceed, InvalidName);
```

bind メソッドと逆の処理を実行し，指定された Name に関連付けられているバインディングを削除します。

パラメータ	説明
n	リバインドする名前で初期化された Name 構造体。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
NotFound	Name またはその要素の 1 つが見つかりませんでした。
CannotProceed	シーケンス内の NameComponent オブジェクトの 1 つが解決できませんでした。ただし，クライアントは，返されたネーミングコンテキストでオペレーションを継続できます。
InvalidName	指定された Name に名前要素がないか, 名前要素の 1 つの id フィールドが空文字列です。

```
virtual NamingContext_ptr new_context();
```

新しいネーミングコンテキストを作成します。新しく作成されたコンテキストは，このオブジェクトと同じサーバー内で実装されます。新しいコンテキストは，最初はどの Name にもバインドされていません。

```
virtual NamingContext_ptr bind_new_context(const Name& _n) raises(NotFound, CannotProceed,
InvalidName, AlreadyBound);
```

新しいコンテキストを作成し，この Context 内で，指定された Name に新しいコンテキストをバインドします。

パラメータ	説明
n	新しく作成された NamingContext に対する名前で初期化された Name 構造体。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
NotFound	Name またはその要素の 1 つが見つかりませんでした。
CannotProceed	シーケンス内の NameComponent オブジェクトの 1 つが解決できませんでした。ただし，クライアントは，返された NamingContext でオペレーションを継続できます。

例外	説明
InvalidName	指定された Name に名前要素がないか, 名前要素の 1 つの id フィールドが空文字列です。
AlreadyBound	bind または bind_context オペレーションにおいて，Name がすでにこの NamingContext 内の別のオブジェクトにバインドされています。

```
virtual void destroy() raises(NotEmpty);
```

このネーミングコンテキストを非アクティブ化します。その後，このオブジェクトのオペレーションを呼び出そうとすると,CORBA::OBJECT_NOT_EXIST 実行時例外が生成されます。

このメソッドを使用するには，その前に，この NamingContext に関連してバインドされているすべての Name オブジェクトを unbind メソッドを使用してバインド解除しておく必要があります。空でない NamingContext を廃棄しようとすると，NotEmpty 例外が生成されます。

```
virtual void list(CORBA::ULong _how_many, BindingList_out _bl, BindingIterator_out _bi)
```

このコンテキストが保持するすべてのバインディングを返します。最大 how_many 個の Name が BindingList に返されます。残りのバインディングは，BindingIterator を介して返されます。返された BindingList と BindingIterator を使用して，任意の名前にアクセスできます。詳細については，「125 ページの「Binding と BindingList」」を参照してください。

パラメータ	説明
how_many	返される Name の最大数。
bl	呼び出し側に返される Name のリスト。このリスト内の名前の数は how_many を超えることはありません。
bi	残りの Name を検索するための反復子。

NamingContextExt

```
class _VISNMEXPORT NamingContextExt : public virtual NamingContext, public virtual CORBA
Object
```

NamingContext を拡張した NamingContextExt インターフェースは，文字列化された名前と URL の使用に必要な操作を提供します。

このコードサンプルは，NamingContextExt シーケンスの IDL 仕様を示しています。

```
module CosNaming {
   interface NamingContextExt {
      typedef string StringName;
      typedef string Address;
      typedef string URLString;
      StringName to_string(in Name n)
         raises(InvalidName);
      Name to_name(in StringName sn)
         raises(InvalidName);
      exception InvalidAddress {};
      URLString to_url(in Address addr, in StringName sn)
         raises(InvalidAddress, InvalidName);
      Object resolve_str(in StringName n)
         raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
   };
```

NamingContextExt のメソッド

```
virtual char* to_string(const Name& _n) raises(InvalidName);
```

指定された Name の文字列化表現を返します。

パラメータ	説明
n	オブジェクトに付ける名前で初期化された Name 構造体。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
InvalidName	指定された Name に名前要素がないか，名前要素の 1 つの id フィールドが空文字列です。

```
virtual Name* to_name(const char* _sn); raises(InvalidName);
```

文字列化された名前が指定されたとき，その名前に対応する Name オブジェクトを返します。

パラメータ	説明
_sn	目的のオブジェクトの文字列化された名前。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
InvalidName	指定された Name に名前要素がないか，名前要素の 1 つの id フィールドが空文字列です。

```
virtual char* to_url(const char* _addr, const char* _sn); raises(InvalidAddress,
InvalidName);
```

_addr で指定された URL と _sn で文字列化された名前を基に，完全形の文字列 URL を返します。

パラメータ	説明
_addr	myhost.borland.com:800 の形式の URL コンポーネント。このアドレスが空の場合は，ローカルホストになります。
_sn	目的のオブジェクトの文字列化された名前。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
InvalidAddress	指定されたアドレスの形式が不正です。
InvalidName	指定された Name に名前要素がないか，名前要素の 1 つの id フィールドが空文字列です。

```
virtual CORBA::Object _ptr resolve_str(const char* _n); raises(NotFound, CannotProceed,
InvalidName, AlreadyBound);
```

文字列化された名前が指定されたとき，その名前に対応する Name オブジェクトを返します。

パラメータ	説明
_n	目的のオブジェクトの文字列化された名前。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
NotFound	Name またはその要素の 1 つが見つかりませんでした。
CannotProceed	シーケンス内の NameComponent オブジェクトの 1 つが解決できませんでした。ただし，クライアントは，返された NamingContext でオペレーションを継続できます。
InvalidName	指定された Name に名前要素がないか，名前要素の 1 つの id フィールドが空文字列です。
AlreadyBound	bind または bind_context オペレーションにおいて，Name がすでにこの NamingContext 内の別のオブジェクトにバインドされています。

Binding と BindingList

Binding，BindingList，および BindingIterator インターフェースは，NamingContext に含まれている名前オブジェクトバインディングの記述に使用されます。Binding 構造体は，名前とオブジェクトの組を 1 つカプセル化します。binding_name フィールドは Name を表し，binding_type は，その Name が ORB オブジェクトと NamingContext オブジェクトのどちらにバインドされているかを示します。

BindingList は，NamingContext オブジェクトに保持される Binding 構造体のシーケンスです。BindingList を使用するサンプルプログラムは，『Borland VisiBroker 開発者ガイド』の「ネーミングサービス」にあります。

このコードサンプルは，Binding 構造体の IDL 仕様を示しています。

```
module CosNaming {
   enum BindingType {
      nobject,
      ncontext
   }
   struct Binding {
      Name binding_name;
      BindingType binding_type;
   };
   typedef sequence<Binding> BindingList;
};
```

BindingIterator

```
class _VISNMEXPORT BindingIterator : public virtual CORBA_Object
```

クライアントアプリケーションで，NamingContext オペレーションの list から返されたバインディングの可変長の集合内を移動して検索できます。BindingIterator を使用するサンプルプログラムは，『Borland VisiBroker 開発者ガイド』の「ネーミングサービス」にあります。

このコードサンプルは，BindingIterator インターフェースの IDL 仕様を示しています。

```
module CosNaming {
   interface BindingIterator {
      boolean next_one(out Binding b);
      boolean next_n(in unsigned long how_many, out BindingList b);
      void destroy();
   };
};
```

BindingIterator のメソッド

```
virtual CORBA::Boolean next_one(Binding_out b_);
```

集合から次の Binding を返します。リストの最後まで検索しても見つからなかった場合は，CORBA::FALSE が返されます。それ以外の場合は，CORBA::TRUE が返されます。

パラメータ	説明
b_	リスト内の次の Binding オブジェクト。

```
virtual CORBA::Boolean next_n(CORBA::ULong _how_many, BindingList_out _b);
```

要求された数だけリストから Binding オブジェクトを取り出して収めた BindingList を返します。リストの末尾に達した場合，返されるバインディングの数は，要求された数より小さくなる可能性があります。リストの末尾に達した場合は，CORBA::FALSE が返されます。そうでない場合は，CORBA::TRUE を返します。

パラメータ	説明
_how_many	要求する Binding オブジェクトの最大数
_b	要求された数以下の Binding オブジェクトを保持する BindList

```
virtual void destroy();
```

このオブジェクトを廃棄し，関連付けられていたメモリを解放します。このメソッドの呼び出しに失敗すると，メモリの使用量が増加します。

NamingContextFactory

```
class _VISNMEXPORT NamingContextFactory : public virtual CORBA_Object
```

このインターフェースは，初期の NamingContext をインスタンス化するために提供されます。クライアントは，この型のオブジェクトにバインドし，create_context メソッドを使用して初期のコンテキストを作成します。初期のコンテキストを作成した後は，new_context メソッドを使用してほかのコンテキストを作成できます。ネーミングコンテキストファクトリのインスタンスは，ネーミングサービスが起動されたときに作成されます。詳細については，『Borland Enterprise Server 開発者ガイド』の「ネーミングサービス」を参照してください。

自動的に 1 つのルートコンテキストを作成する初期の NamingContextFactory を作成する場合は，「127 ページの「ExtendedNamingContextFactory」」を参照してください。

このコードサンプルは，NamingContextFactory シーケンスの IDL 仕様を示しています。

```
module CosNaming {
   interface NamingContextFactory {
      NamingContext create_context();
      oneway void shutdown();
   };
};
```

メソッド

```
virtual CosNaming::NamingContextEXT_ptr create_context();
```

このメソッドで，クライアントはネーミングコンテキストを作成できます。ネーミングコンテキストの仕様では，ネーミングコンテキストはルートコンテキストを認識しない

とされているので，NamingContextFactory をインスタンス化しただけでは，ネーミングコンテキストは作成されません。

```
virtual ClusterManager_ptr get_cluster_manager();
```

クラスタを返します。

```
virtual NamingContextList* list_all_roots (const char* _password);
```

すべてのルートコンテキストのリストを返します。

```
virtual void remove_stale_contexts (const char* _password);
```

このメソッドを使用すると，クライアントは，クラスタの存続期間中にクラスタからメンバーを削除できます。

```
virtual void shutdown();
```

このメソッドを使用すると,クライアントは,適切な手段でネーミングサービスをシャットダウンできます。同じログファイルを使用してサービスが再起動された場合，ファクトリは，シャットダウンされる前の状態に復元されます。

ExtendedNamingContextFactory

```
class _VISNMEXPORT ExtendedNamingContextFactory : public virtual NamingContextFactory,
public virtual CORBA_Object
```

このインターフェースは NamingContextFactory インターフェースを拡張し，その拡張されたネーミングサービスが起動されたときに，ファクトリ内にデフォルトルートを作成できます。詳細については,『Borland Enterprise Server 開発者ガイド』の「ネーミングサービス」を参照してください。

このコードサンプルは，ExtendedNamingContextFactory シーケンスの IDL 仕様を示しています。

```
module CosNaming {
   interface ExtendedNamingContextFactory : NamingContextFactory{
      NamingContext root_context();
   };
};
```

メソッド

```
virtual CosNaming::NamingContextExt_ptr root_context();
```

このオブジェクトがインスタンス化されたときに自動的に作成されたルートネーミングコンテキストを返します。

第 8 章

イベントサービスのインターフェースとクラス

ここでは，VisiBroker for C++ のイベントサービスに関するインターフェースとクラスについて説明します。

ConsumerAdmin

```
public interface ConsumerAdmin extends ConsumerAdminPOA
```

このインターフェースは，プロキシサプライヤオブジェクトへのリファレンスを取得するために，コンシューマアプリケーションによって使用されます。これは，コンシューマアプリケーションを EventChannel に接続する際の 2 番めの手順です。

IDL 定義

```
module CosEventChannelAdmin {
   interface ConsumerAdmin {
      ProxyPushConsumer obtain_push_supplier();
      ProxyPullConsumer obtain_pull_supplier();
   };
};
```

ConsumerAdmin のメソッド

```
public ProxyPushSupplier obtain_push_supplier();
```

obtain_push_supplier メソッドは，呼び出し側のコンシューマアプリケーションがプッシュモデルを使用して実装されている場合に呼び出されます。プルモデルを使用してアプリケーションを実装する場合は，obtain_pull_supplier メソッドを呼び出す必要があります。

```
public ProxyPullSupplier obtain_pull_supplier();
```

返されたリファレンスは，「ProxyPushConsumer」に説明されている connect_push_consumer メソッド，または 131 ページの「ProxyPullConsumer」に説明されている connect_pull_consumer メソッドのどちらかを呼び出すために使用されます。

EventChannel

public interface **EventChannel**

EventChannel は，チャネルにサプライヤおよびコンシューマを追加したり，チャネルを削除するための管理操作を提供します。イベントチャネルの詳細については，130 ページの「EventChannelFactory」を参照してください。

サプライヤとコンシューマは，ともに _bind メソッドを使用して EventChannel への参照を取得します。すべての _bind 呼び出しと同様に，呼び出し側は，目的の EventChannel のオブジェクト名とバインドオプションを任意に指定できます。これらの引数は，初期のパラメータとしてサプライヤまたはコンシューマに渡したり，ネーミングサービスが使用可能な場合は，そこから取得することができます。オブジェクト名を指定しない場合は，適切な EventChannel が VisiBroker Edition によって検索されます。サプライヤまたはコンシューマは，EventChannel に接続した後で，任意の EventChannel メソッドを呼び出すことができます。

メソッド

次のコードサンプルは，「power」というオブジェクト名で EventChannel にバインドするサプライヤを示します。

```
int main(int argc, char* const* argv)
{
        ...
        CosEventChannelAdmin::EventChannel_var my_channel =
           CosEventChannelAdmin::EventChannel::_bind("power");
        CosEventChannelAdmin::SupplierAdmin_var = channel->for_suppliers();
        ...
}
```

ConsumerAdmin **for_consumers**();

ConsumerAdmin オブジェクトを返します。このオブジェクトは，この EventChannel にコンシューマを追加するために使用されます。

SupplierAdmin **for_suppliers**();

SupplierAdmin オブジェクトを返します。このオブジェクトは，この EventChannel にサプライヤを追加するために使用されます。

void **destroy**();

この EventChannel を廃棄します。

EventChannelFactory

public interface **EventChannelFactory**

EventChannelFactory は，イベントチャネルを作成，検索，および削除するメソッドを提供します。

IDL 定義

```
module CosEventChannelAdmin {
   interface EventCHannelFactory {
      exception AlreadyExists();
      exception ChannelsExist();
      EventChannel create();
      EventChannel create_by_name(in string name)
            raises(AlreadyExists);
      EventChannel lookup_by_name(in string name);
      void destroy()
            raises(ChannelsExist);
   };
};
```

EventChannelFactory のメソッド

`EventChannel create();`

このメソッドは，匿名の一時イベントチャネルを作成します。

`EventChannel create_by_name(in string name) raises(AlreadyExists);`

このメソッドは，名前付きの永続的イベントチャネルを作成します。指定された名前のイベントチャネルがすでに作成されている場合は，AlreadyExists 例外が生成されます。

`EventChannel lookup_by_name(in string name);`

指定した name で EventChannel を返します。指定された名前のチャネルが存在しない場合は，NULL 値が返されます。

`void destroy();`

このイベントチャネルを廃棄します。チャネルが廃棄される前に，このチャネルに接続しているすべてのサプライヤとコンシューマの切断のメソッドが呼び出されます。チャネルが create_by_name メソッドで作成されていた場合，そのチャネルがいったん破棄されてしまると，lookup_by_name メソッドでは見つけることができなくなります。

ProxyPullConsumer

`public interface ProxyPullConsumer`

このインターフェースは，プルサプライヤアプリケーションによって使用され，サプライヤの PullSupplier 派生オブジェクトを EventChannel に接続する connect_pull_supplier メソッドを提供します。同じプロキシに 2 回以上接続しようとすると，AlreadyConnected 例外が生成されます。

IDL 定義

```
module CosEventChannelAdmin {
   exception AlreadyConnected();
   interface ProxyPullConsumer : CosEventComm::PullConsumer {
      void connect_pull_supplier(in CosEventComm::PullSupplier pull_supplier)
            raises(AlreadyConnected);
   };
};
```

ProxyPushConsumer

public interface **ProxyPushConsumer**

このインターフェースは，プッシュサプライヤアプリケーションで使用され，connect_push_supplier メソッドを提供します。このメソッドを使用して，サプライヤの PushSupplier 派生オブジェクトを EventChannel に接続します。同じ proxy に 2 回以上接続しようとすると，AlreadyConnected 例外が生成されます。

IDL 定義

```
module CosEventChannelAdmin {
   exception AlreadyConnected();
   interface ProxyPushConsumer : CosEventComm::PushConsumer {
      void connect_push_supplier(in CosEventComm::PushSupplier push_supplier)
            raises(AlreadyConnected);
   };
};
```

ProxyPullSupplier

public interface **ProxyPullSupplier**

プルコンシューマアプリケーションで使用し，connect_pull_consumer メソッドを提供します。コンシューマの PullConsumer 派生したオブジェクトを EventChannel も提供します。同じ PullConsumer に 2 回以上接続しようとすると，AlreadyConnected 例外が生成されます。

IDL 定義

```
module CosEventChannelAdmin {
   exception AlreadyConnected();
   interface ProxyPullSupplier : CosEventComm::PullSupplier {
      void connect_pull_consumer(in CosEventComm::PullConsumer pull_consumer)
            raises(AlreadyConnected);
   };
};
```

ProxyPushSupplier

public interface **ProxyPushSupplier**

このインターフェースは，プッシュコンシューマアプリケーションで使用され，connect_push_consumer メソッドを提供します。このメソッドを使用して，コンシューマの PushConsumer 派生オブジェクトを EventChannel に接続します。同じ PushConsumer に 2 回以上接続しようとすると，AlreadyConnected 例外が生成されます。

IDL 定義

```
module CosEventChannelAdmin {
   exception AlreadyConnected();
   interface ProxyPushSupplier : CosEventComm::PushSupplier {
      void connect_push_consumer(in CosEventComm::PushConsumer push_consumer)
            raises(AlreadyConnected);
   };
};
```

PullConsumer

public interface **PullConsumer**

このインターフェースは，プルモデルを使用して通信するコンシューマオブジェクトの派生に使用されます。pull メソッドは，サプライヤからのデータが必要な場合にいつでもコンシューマによって呼び出されます。サプライヤが接続されていない場合は，Disconnected 例外が生成されます。

チャネルが廃棄されている場合，このコンシューマを終了するために disconnect_push_consumer メソッドが使用されます。

IDL 定義

```
module CosEventChannelAdmin {
   exception Disconnected {};
   interface PushConsumer {
      void push(in any data) raises(Disconnected);
      void disconnect_push_consumer();
   };
};
```

PushConsumer

public interface **PushConsumer**

このインターフェースは，プッシュモデルを使用して通信するコンシューマオブジェクトの派生に使用されます。コンシューマへのデータがある場合，サプライヤによって push メソッドが使用されます。コンシューマが接続されていない場合は，Disconnected 例外が生成されます。

IDL 定義

```
module CosEventComm {
   exception Disconnected();
   interface PushConsumer {
      void push(in any data) raises(Disconnected);
      void disconnect_push_consumer();
   };
};
```

PullSupplier

public interface **PullSupplier**

このインターフェースは，プルモデルを使用して通信するサプライヤオブジェクトの派生に使用されます。

IDL 定義

```
module CosEventComm {
   interface PullSupplier {
      any pull() raises(Disconnected);
      any try_pull(out boolean has_event) raises(Disconnected);
      void disconnect_pull_supplier();
```

```
    };
};
```

PullSupplier のメソッド

```
any pull();
```

このメソッドは，サプライヤから使用可能なデータがあるまでブロックします。データは Any 型で返されます。コンシューマ接続を解除している場合，このメソッドは Disconnected 例外を生成します。

```
any try_pull(out boolean has_event);
```

このメソッドは，サプライヤからデータを取得しようとしますが，ブロックはしません。このメソッドが戻ったとき，使用可能なデータがあった場合は，has_event が true に設定され，そのデータが Any 型で返されます。has_event が false に設定されている場合は，使用可能なデータがなく，戻り値は NULL です。

```
void disconnect_pull_supplier();
```

チャネルが廃棄された場合に，このプルサーバーを非アクティブ化します。

PushSupplier

```
public interface PushSupplier
```

このインターフェースは，プッシュモデルを使用して通信するサプライヤオブジェクトの派生に使用されます。disconnect_push_supplier メソッドが EventChannel によって使用され，サプライヤが破壊されると切断されます。

IDL 定義

```
module CosEventComm {
   exception AlreadyConnected();
   interface PushSupplier {
      void disconnect_push_supplier();
   };
};
```

SupplierAdmin

public interface **SupplierAdmin**

このインターフェースは，コンシューマプロキシオブジェクトへのリファレンスを取得するために，サプライヤアプリケーションによって使用されます。これは，サプライヤアプリケーションを EventChannel に接続する際の 2 番めの手順です。

IDL 定義

```
module CosEventChannelAdmin {
   interface SupplierAdmin {
      ProxyPushConsumer obtain_push_consumer();
      ProxyPullConsumer obtain_pull_consumer();
   };
};
```

public ProxyPushConsumer **obtain_push_consumer();**

obtain_push_consumer メソッドは，サプライヤアプリケーションがプッシュモデルを使用して実装されている場合に呼び出されます。プルモデルを使用してアプリケーションを実装する場合は，obtain_pull_consumer メソッドを呼び出す必要があります。

public ProxyPullConsumer **obtain_pull_consumer**();

返された参照は，connect_push_supplier を呼び出すために使用されます。

第 9 章

サーバーマネージャのインターフェースとクラス

ここでは，VisiBroker for C++ のサーバーマネージャのインターフェースとクラスについて説明します。サーバーマネージャの詳細については，『VisiBroker for C++ 開発者ガイド』の「VisiBroker サーバーマネージャの使い方」を参照してください。

Container インターフェース

コンテナは，プロパティ，オペレーション，およびほかのコンテナを保持できます。主要な ORB コンポーネントは，コンテナとして表されます。最上位のコンテナは ORB 自体に対応し，いくつかの ORB プロパティ，shutdown メソッド，およびよく使用されるほかのコンテナ（RootPOA や Agent など）を含んでいます。

Container インターフェースのメソッド

ここでは，Container インターフェースで実行できる C++ のメソッドについて説明します。これらのメソッドは次の 4 つのカテゴリに分けられます。

- プロパティの操作とクエリーに関連するメソッド
- オペレーションに関連するメソッド
- 子コンテナに関連するメソッド
- ストレージに関連するメソッド

プロパティの操作とクエリーに関連するメソッド

```
virtual CORBA::StringSequence* list_all_properties();
```

コンテナ内にあるすべてのプロパティの名前を StringSequence として返します。

```
virtual PropertySequence* get_all_properties();
```

コンテナ内にあるすべてのプロパティの名前，値，読み取り／書き込みの状態を含む PropertySequence を返します。

```
virtual Property* get_property(const char * name);
```

入力パラメータとして渡されたプロパティ *name* の値を返します。

パラメータ	説明
name	プロパティの名前。

渡されたパラメータが無効なプロパティ名である場合，*NameInvalid* 例外が生成されます。

```
virtual void set_property(const char* name, CORBA::Any& value);
```

プロパティ *name* の値を要求された *value* に設定します。

パラメータ	説明
name	値を設定するプロパティの名前。
value	*Any* 型のプロパティ値。

例外 *NameInvalid*，*ValueInvalid*，または *ValueNotSettable* が生成されます。

```
virtual void persist_properties(CORBA::Boolean recurse);
```

コンテナは，関連付けられている 140 ページの「Storage インターフェースのメソッド (C++)」にプロパティを実際に保存します。ストレージがコンテナに関連付けられていない場合は，StorageException が生成されます。パラメータ recurse=true を指定して呼び出すと，子コンテナのプロパティもストレージに保存されます。すべてのプロパティを保存するか，変更されたプロパティだけを保存するかは，コンテナによって異なります。

パラメータ	説明
recurse	サブコンテナの *persist_properties* を再帰的に呼び出すかどうかを指定します。

StorageException 例外が生成されることがあります。

```
virtual void restore_properties(CORBA::Boolean recurse);
```

ストレージからプロパティを取得するようにコンテナに指示します。コンテナは，管理しているプロパティを正確に認識しており，それらをストレージから読み取ろうとします。ORB に付属するコンテナは，ストレージからの復元をサポートしていません。この機能をサポートするコンテナは，独自に作成する必要があります。

パラメータ	説明
recurse	サブコンテナの *restore_properties* を再帰的に呼び出すかどうかを指定します。

StorageException 例外が生成されることがあります。

オペレーションに関連するメソッド

```
virtual CORBA::StringSeqence* list_all_operations();
```

コンテナでサポートされているすべてのオペレーションの名前を返します。

```
virtual OperationSequence* get_all_operations();
```

すべてのオペレーション，オペレーションのパラメータ，およびパラメータのタイプコードを返します。これで，適切なパラメータを使用してオペレーションを呼び出すことができます。

```
virtual Operation* get_operation(const char* name);
```

name で指定されたオペレーションのパラメータ情報を返します。この情報を使用して，オペレーションを呼び出すことができます。

パラメータ	説明
name	パラメータ情報を取得するオペレーションの名前。

サポートされていないオペレーションが指定された場合は，*NameInvalid* 例外が生成されることがあります。

```
CORBA::Any* do_operation(const Operation& op);
```

オペレーションのメソッドを呼び出し，その結果を返します。

パラメータ	説明
op	サーバーで実行されるオペレーション。

NameInvalid，*ValueInvalid*，または *OperationFailed* が生成されることがあります。

子コンテナに関連するメソッド

```
virtual CORBA::StringSequence* list_all_containers();
```

現在のコンテナの子コンテナの名前をすべて返します。

```
virtual NamedContainerSequence* get_all_containers();
```

すべての子コンテナを返します。

```
virtual NamedContainer* get_container(const char * name) ;
```

name パラメータで指定された子コンテナを返します。

パラメータ	説明
name	子コンテナを照会するコンテナの名前。

この名前の子コンテナがない場合は，NameInvalid 例外が生成されます。

```
virtual void add_container(const NamedContainer& container);
```

このコンテナの子コンテナとして，*container* を追加します。

パラメータ	説明
container	このコンテナに追加する子コンテナの名前。

例外 *NameAlreadyPresent* または *ValueInvalid* が生成されることがあります。

```
virtual void set_container (const char * name, Container_ptr value);
```

name パラメータで指定された子コンテナを *value* パラメータで指定されたコンテナに変更します。

パラメータ	説明
name	値を置き換えるコンテナの名前。
value	新しい子コンテナ。

例外 *NameInvalid*, *ValueInvalid*, または *ValueNotSettable* が生成されることがあります。

ストレージに関連するメソッド

```
virtual void set_storage(Storage_ptr s, CORBA::Boolean recurse);
```

このコンテナのストレージを設定します。recurse=true の場合は，すべての子コンテナに対してもストレージが設定されます。

パラメータ	説明
s	新しく設定するストレージ。
recurse	子コンテナにもストレージを再帰的に設定するかどうかを指定します。

```
virtual Storage_ptr get_storage();
```

コンテナの現在のストレージを返します。

Storage インターフェース

サーバーマネージャには，任意の形式で実装できる*ストレージ*という抽象概念があります。各コンテナは，プロパティの保存先をデータベースやフラットファイルなどの形式から選択できます。VisiBroker ORB に用意されているストレージのインプリメンテーションでは，フラットファイルが使用されます。

Storage インターフェースのメソッド（C++）

```
virtual void open();
```

ストレージを開き，プロパティを読み書きできるようにします。データベースに基づくインプリメンテーションの場合は，このメソッドによってデータベースにログインします。

何らかの理由でストレージを開くことができなかった場合は，*StorageException* が生成されます。

```
virtual void close();
```

ストレージを閉じます。また，このメソッドは，最後に Container::persist_properties が呼び出されてから変更されたプロパティがあれば，ストレージを更新します。データベースに基づくインプリメンテーションでは，このメソッドによってデータベース接続が閉じられます。

何らかの理由でデータベースを閉じることができなかった場合は，*StorageException* が生成されます。

```
virtual Container::PropertySequence* read_properties();
```

ストレージからすべてのプロパティを読み取ります。ストレージからプロパティを読み取ることができなかった場合は，*StorageException* が生成されることがあります。

```
virtual Container::Property* read_property(const char * propertyName);
```

ストレージから読み取った *propertyName* のプロパティ値を返します。

パラメータ	説明
propertyName	ストレージから読み取るプロパティの名前。

StorageException または *Container::NameInvalid* が生成されることがあります。

```
virtual void write_properties( const Container::PropertySequence& p);
```

ストレージにプロパティシーケンスを保存します。

パラメータ	説明
p	セッション中に変更されたプロパティのシーケンス。

StorageException が生成されることがあります。

```
virtual void write_property( const Container::Property& p);
```

ストレージに 1 つのプロパティを保存します。

パラメータ	説明
p	永続的ストレージに書き込むプロパティ。

StorageException が生成されることがあります。

第 10 章

トランザクションサービスのインターフェースとクラス

ここでは, 次のような VisiBroker VisiTransact トランザクションサービスのモジュール, インターフェース, およびクラスについて説明します。

- CosTransactions モジュールと VISTransactions モジュール
- Current インターフェース
- TransactionalObject インターフェース
- TransactionFactory インターフェース
- Control インターフェース
- Terminator インターフェース
- Coordinator インターフェース
- RecoveryCoordinator インターフェース
- Resource インターフェース
- Synchronization インターフェース
- VISTransactionService クラス
- VISSessionManager モジュール
- ConnectionPool インターフェース
- Connection インターフェース
- ITSDataConnection クラス
- ネイティブハンドル取得インターフェース
- ローカルトランザクション接続／完了インターフェース
- グローバルトランザクション接続／完了インターフェース

CosTransactions モジュールと VISTransactions モジュール

ここでは，CosTransactions モジュールと VisTransactions モジュールを紹介し，CosTransactions モジュールのデータ型，構造体，および例外について説明します。

CosTransactions モジュールの確認

CosTransactions モジュールは，最終的な OMG Transaction Service ドキュメントに準拠するトランザクションサービス IDL です。このモジュールにより，CORBA 準拠のメソッドを使用するように厳しく制限されます。このモジュールの IDL は，ファイル **CosTransactions.idl** に含まれています。

また，VISTransactions モジュールを使用することもできます。このモジュールには，標準の機能に対する VisiBroker VisiTransact による拡張機能の IDL が含まれています。VISTransactions モジュールの IDL は，ファイル **VISTransactions.idl** に含まれています。コードで **VISTransactions.idl** を使用すると，CosTransactions モジュールと VISTransactions モジュールの両方を取得できます。詳細については，147 ページの「VISTransactions モジュールの確認」を参照してください。

データ型

CosTransactions モジュールには，データ型 enum Status および enum Vote が定義されています。

データ型 enum Status の定義は次のとおりです。

```
enum Status
{
  StatusActive,
  StatusMarkedRollback,
  StatusPrepared,
  StatusCommitted,
  StatusRolledBack,
  StatusUnknown,
  StatusNoTransaction
  StatusPreparing,
  StatusCommitting,
  StatusRollingBack,
};
```

各 Status 値の詳細については，154 ページの「ステータス値の定義」を参照してください。

データ型 enum Vote は，CosTransactions::Resource インターフェースのインプリメンテーションによってのみ使用されます。これを使用して，リソースがトランザクションの準備を試みた結果を示すことができます。

データ型 enum Vote の定義は次のとおりです。

```
enum Vote
{
  VoteCommit,
  VoteRollback,
  VoteReadOnly
};
```

次の Vote 値を指定できます。

- VoteCommit - リソースは，トランザクションをコミットするために必要なすべてのデータ，およびトランザクションの準備が整ったことを示すデータを安全なストレージに書き込むことができます（または，すでに書き込みました）。
- VoteRollback - 何らかの理由で，リソースは，トランザクションのコミットを提案できませんでした。たとえば，システムがクラッシュした後など，トランザクションに関する情報がない場合です。

- VoteReadOnly - リソースに関連付けられた永続的データは，トランザクションによって変更されていません。

構造体

CosTransactions モジュールには，次の構造体が定義されています。これらは，トランザクションコンテキストの保存に使用されます。

- otid_t には，オブジェクトトランザクション ID（otid）が格納されます。これは，トランザクションのグローバルに一意な ID です。otid_t 構造体は，X/Open で定義されているトランザクション ID（XID）の効率をよくした OMG IDL のバージョンです。otid_t と X/Open XID は，相互に変換できます。
- TransIdentity には，Coordinator，Terminator（オプション），otid など，トランザクションの主要な情報が格納されます。
- PropagationContext には，トランザクションの TransIdentity とタイムアウトが格納されます。また，親トランザクションと最上位までの各トランザクションの TransIdentity がシーケンス（または配列）の形式で格納されます。VisiTransact にはネストしたトランザクションが実装されていないため，すべてのトランザクションが最上位のトランザクションになり，parents シーケンスは常に空になります。

通常，これらの構造体はバックグラウンドで使用され，直接参照することはありません。

```
struct otid_t
{
  long formatID;
  long bqual_length;
  sequence <octet> tid;
};
struct TransIdentity
{
  Coordinator coordinator;
  Terminator terminator;
  otid_t otid;
};
struct PropagationContext
{
  unsigned long timeout;
  TransIdentity current;
  sequence <TransIdentity> parents;
  any implementation_specific_data;
};
```

トランザクションコンテキストが 1 つのオブジェクトからほかのオブジェクト（通常は TransactionalObject）に渡される場合，これは PropagationContext として渡されるのが普通です。implementation_specific_data フィールドは，VisiTransact Transaction Service のために予約されています。

通常，これらの構造体はバックグラウンドで使用され，直接参照することはありません。ただし，いくつかのメソッドは，PropagationContext を明示的に使用します。

- Coordinator::get_txcontext() は PropagationContext を抽出します。
- TransactionFactory::recreate() は PropagationContext を使用して，新しい Control オブジェクトを作成します。

トランザクションコンテキストは，常にトランザクションオブジェクトに明示的に渡されます。また，プログラムにトランザクションコンテキストをパラメータとして明示的に渡すこともできます。Coordinator::get_txcontext() を使用すると，PropagationContext を取得できます。トランザクションコンテキストの伝達については，161 ページの「TransactionalObject インターフェース」を参照してください。

これらの構造体から情報を取得するメソッドとして，ほかに VISTransactions::Current::get_otid() があります。これは，PropagationContext から otid を抽出します。

例外

例外は，標準，経験則（ヒューリスティック），およびメソッド固有の 3 つのカテゴリに分けられます。

表 10.1 CosTransactions モジュール内の標準の例外

例外	この例外が発生する条件
CORBA::INVALID_TRANSACTION	呼び出し元のスレッドに，無効なトランザクションコンテキストがあります。
CORBA::NO_PERMISSION	呼び出し元のスレッドに，トランザクションを完了する権限がありません。たとえば，トランザクションオリジネータのスレッドだけがこのメソッドを呼び出すことができます。
CORBA::TRANSACTION_REQUIRED	呼び出し元のスレッドに，トランザクションコンテキストがありません。
CORBA::TRANSACTION_ROLLEDBACK	トランザクションがロールバックされました。
CORBA::WrongTransaction	遅延同期要求への応答を返す際に，ORB によって生成されます。この例外は，要求が，Request::get_response() または ORB::get_next_response() を介して応答を要求しているスレッドとは異なるトランザクションに暗黙的に関連付けられていた場合に生成されます。詳細については，動的起動インターフェース（DII）の VisiBroker の節を参照してください。

表 10.2 CosTransactions モジュール内の経験則（ヒューリスティック）の例外

例外	この例外が発生する条件
CosTransactions::HeuristicCommit	リソースのロールバック操作において，経験則による決定が行われ，関連する更新がすべてコミットされたことを通知するために，この例外が生成されます。
CosTransactions::HeuristicMixed	トランザクションのコミットを試みた際に，経験則による決定が行われました。一部の関連する更新がコミットされ，それ以外はロールバックされました。
CosTransactions::HeuristicHazard	トランザクションのコミットを試みた際に，経験則による決定が行われ可能性があります。関連する更新の結果が一部不明です。結果が既知である更新は，すべてコミットされたか，すべてロールバックされています。つまり，HeuristicMixed 例外が HeuristicHazard 例外に優先されます。
CosTransactions::HeuristicRollback	リソースのコミット操作において，経験則による決定が行われ，関連する更新がすべてロールバックされたことを通知するために，この例外が生成されます。

表 10.3 CosTransactions モジュール内のメソッド固有の例外

例外	この例外が発生する条件
CosTransactions::Inactive	トランザクションがすでに準備されています。
CosTransactions::InvalidControl	再開のために渡された Control パラメータが現在の実行時環境で無効です。
CosTransactions::NotPrepared	コミットが発行されたが，リソースが準備できていません。
CosTransactions::NoTransaction	クライアントスレッドに関連付けられたトランザクションがありません。
CosTransactions::NotSubtransaction	VisiTransact では生成されません。
CosTransactions::SubtransactionsUnavailable	VisiBroker VisiTransact はネストしたトランザクションをサポートしていないため，この例外は，このクライアントスレッドのトランザクションがすでに実行されている場合に，トランザクションを開始しようとすると生成されます。
CosTransactions::SynchronizationUnavailable	VisiTransact では生成されません。
CosTransactions::Unavailable	要求されたオブジェクトを提供できません。たとえば，Control オブジェクトが Terminator を提供できませんでした。

VISTransactions モジュールの確認

VISTransactions モジュール内のインターフェースは，CosTransactions のインターフェースを継承および拡張しています。VISTransactions モジュールには，CosTransactions の内容を上書きする新しいデータ型，構造体，または例外は定義されていません。たとえば，Current インターフェースには，特定のプログラミング操作を短時間で簡単に実行できる VisiBroker VisiTransact メソッドが含まれています。このモジュールの IDL は，ファイル **VISTransactions.idl** に含まれています。

関連情報については，147 ページの「Current インターフェースの選択」および 149 ページの「Current オブジェクトリファレンスの取得」を参照してください。

Current インターフェース

Current インターフェースは，次のためのメソッドを定義します。

- プログラムでトランザクションを管理する
- 暗黙的にトランザクションを伝達する
- 現在のトランザクションに関する情報を取得する
- リソースと Synchronization オブジェクトを登録する

VisiBroker VisiTransact は，OMG Transaction Service 仕様に対する拡張機能をサポートしています。複数のメソッドが追加され，使いやすくなっています。Current インターフェースの VisiBroker VisiTransact メソッドを使用すると，ほとんどのプログラムで VisiTransact Transaction Service を簡単に使用できるようになります。これらのメソッドを説明または参照する箇所には，  のアイコンが付けられています。

Current インターフェースの選択

VisiTransact Transaction Service の Current インターフェースは，次の IDL ファイルに含まれています。

- **CosTransactions.idl** には，最終的な OMG Transaction Service ドキュメントに準拠するトランザクションサービス IDL が格納されています。
- **VISTransactions.idl** には，CosTransactions インターフェースと VISTransactions インターフェースの両方が格納されています。これは，CosTransactions::Current インターフェースを継承および拡張しています。このインターフェースには，begin_with_name() や register_resource() など，VisiBroker VisiTransact の拡張機能が含まれています。

これらの IDL ファイルのいずれかを使用する必要があります。CORBA 準拠のメソッドだけを使用する場合は，**CosTransactions.idl** を使用します。VisiTransact の拡張機能を使用する場合は，**VISTransactions.idl** を使用します。

次に，CosTransactions の Current インターフェースを示します。

```
...
interface Current
{
  void begin()
    raises(SubtransactionsUnavailable);
  void commit(in boolean report_heuristics)
    raises (NoTransaction,
         HeuristicMixed,
         HeuristicHazard);
  void rollback()
    raises(NoTransaction);
  void rollback_only()
    raises(NoTransaction);
  Status get_status();
  string get_transaction_name();
  void set_timeout(in unsigned long seconds);
  Control get_control();
  Control suspend();
  void resume(in Control which)
    raises(InvalidControl);
};
...
```

次に，VISTransactions の Current インターフェースを示します。

```
interface Current : CosTransactions::Current
{
  void   begin_with_name(in string user_transaction_name)
         raises(CosTransactions::SubtransactionsUnavailable);
  CosTransactions::RecoveryCoordinator
    register_resource(in CosTransactions::Resource resource)
         raises(CosTransactions::Inactive);
  void   register_synchronization(in CosTransactions::Synchronization synch)
         raises(CosTransactions::NoTransaction,
                CosTransactions::Inactive,
                CosTransactions::SynchronizationUnavailable
                CosTransactions::Unavailable);
  CosTransactions::otid_t get_otid();
         raises(CosTransactions::NoTransaction,
                CosTransactions::Unavailable);
  CosTransactions::PropagationContext get_txcontext()
         raises(CosTransactions::Unavailable,
                CosTransactions::NoTransaction);
  attribute string ots_name;
  attribute string ots_host;
  attribute string ots_factory;
};
```

Current オブジェクトリファレンスの取得

VisiTransact 管理のトランザクションにアクセスするには，Current へのオブジェクトリファレンスを取得する必要があります。Current オブジェクトリファレンスは，プロセスを通じて有効です。

次の例は，resolve_initial_references() メソッドを使用して Current オブジェクトへのリファレンスを取得し，そのオブジェクトを CosTransactions::Current オブジェクトにナローイングする方法を示しています。

```
int main(....)
{
  try
  {
          ... ...
    // ORB 関連の初期化。
   // CosTransactions::Current インスタンスへのリファレンスを取得します
  CORBA::Object_var
      obj = orb->resolve_initial_references("TransactionCurrent");
    CosTransactions::Current_var
      current = CosTransactions::Current::_narrow(obj);
  ...
  }
  catch(...) { }  // すべての例外をキャッチするか，例外を選択してキャッチします
}
```

VisiBroker VisiTransact には，いくつかの操作を簡単に実行するための Current インターフェースの拡張機能が用意されています。この拡張機能を利用するには，VISTransactions::Current オブジェクトにナローイングします。

```
VISTransactions::Current::_narrow(obj)
```

Current オブジェクトリファレンスの使用

Current オブジェクトリファレンスは，それを作成したプロセス全体で有効です。つまり，任意のスレッドで使用できます。Current オブジェクトへのリファレンスを取得するために何度も呼び出しを実行できます。または，プロセス全体でリファレンスを 1 つだけ使用することもできます。通常は，1 つのリファレンスを取得して，resolve_initial_references() を何度も呼び出さないようにします。

インクルードする C++ ヘッダーファイルも，選択したインターフェースに対応している必要があります。

- VISTransactions の場合は，#include <VISTransactions_c.hh> を使用します。
- CosTransactions の場合は，#include <CosTransactions_c.hh> を使用します。

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

VisiTransact Transaction Service の有効性の確認

VisiTransact Transaction Service のインスタンスを使用できるかどうかを確認するには，begin() または begin_with_name() を発行します。インスタンスを使用できない場合，このメソッドは CORBA:NO_IMPLEMENT 例外を生成します。

使用できる VisiTransact Transaction Service のインスタンスがない場合に get_status() を呼び出すと，現在のトランザクションの状態が返されます。したがって，このメソッドを使用して，VisiTransact Transaction Service のインスタンスを使用できるかどうかを確認することはできません。

checked behavior

VisiTransact Transaction Service によってサポートされている checked behavior を使用して，トランザクションの整合性を高めることができます。具体的には，Current::begin() で開始されたトランザクションに対して，checked behavior を使用できます。チェックの目的は，トランザクションがコミットされる前に，アプリケーションによるすべてのトランザクション要求が処理を完了しているようにすることです。これにより，トランザクションにかかわるすべてのトランザクションオブジェクトがトランザクション要求の処理を完了しない限り，コミットが成功しないことが保証されます。コミットプロセスの一環として実行されるチェックについては，151 ページの「commit() void commit(in boolean report_heuristics) raises(NoTransaction, HeuristicMixed, HeuristicHazard);」を参照してください。checked behavior の詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

Current のメソッド

begin()

```
void begin()
raises SubtransactionsUnavailable;
```

このメソッドは，新しいトランザクションを作成します。VisiBroker VisiTransact はネストしたトランザクションを*サポートしていない*ため，これは常に最上位のトランザクションになります。

クライアントスレッドが新しいトランザクションに関連付けられるように，スレッドのトランザクションコンテキストが変更されます。クライアントスレッドがすでにトランザクションに関連付けられている場合は，例外 SubtransactionsUnavailable が発行されます。

CosTransactions.idl の Current インターフェースに含まれています。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::SubtransactionsUnavailable	VisiBroker VisiTransact はネストしたトランザクションをサポートしていないため，この例外は，このクライアントスレッドのトランザクションがすでに実行されている場合に生成されます。

関連するメソッド：

- begin_with_name()
- commit()
- Control インターフェースの get_terminator()
- rollback()
- rollback_only

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

begin_with_name()

```
void begin_with_name(in string user_transaction_name)
raises(CosTransactions::SubtransactionsUnavailable);
```

この VisiBroker VisiTransact メソッドは，呼び出し元からユーザー定義のトランザクション名情報を渡すことができる begin() メソッドです。たとえば，get_transaction_name() メソッドから返される値に，このユーザー定義のトランザクション名が含まれるため，診断が容易になります。また，コンソールは，未処理のトランザクションに関する詳細情報として，この名前を使用するため，管理が容易になります。

このメソッドを使用するには，resolve_initial_references() から返されたオブジェクトを VISTransactions::Current にナローイングします。詳細については，149 ページの「Current オブジェクトリファレンスの取得」を参照してください。

VISTransactions.idl の Current インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
user_transaction_name	このユーザー定義のトランザクション名情報を使用して，トランザクションを追跡したり，プログラムをデバッグすることができます。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::SubtransactionsUnavailable	スレッドにすでにトランザクションコンテキストがある場合に生成されます。

関連するメソッド：

- begin()
- commit()
- Control インターフェースの get_terminator()
- rollback()
- rollback_only()

commit()

```
void commit(in boolean report_heuristics)
raises(NoTransaction,
       HeuristicMixed,
       HeuristicHazard
);
```

このメソッドは，クライアントスレッドに関連付けられているトランザクションをコミットします。このメソッドの効果は，対応する Terminator オブジェクトの commit() メソッドを呼び出した場合と同じです。

このトランザクションにロールバックのマークが付けられているか，ロールバックを提案する Resource がある場合は，このメソッドを呼び出すと，CORBA::TRANSACTION_ROLLEDBACK が生成されます。現在のトランザクションがない場合は，CosTransactions::NoTransaction 例外が生成されます。呼び出し側がトランザクションオリジネータでない場合，commit() は例外 CORBA::NO_PERMISSION を生成します。

checked behavior のためのチェックが行われます。詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

このメソッドから戻った時点で，クライアントスレッドとトランザクションの関連付けが解除されます。トランザクションが存在するときと同様に Current を使用しようとすると，NoTransaction または CORBA::TRANSACTION_REQUIRED などの例外が生成されるか，null オブジェクトリファレンスが返されます。

トランザクションが完了し，関連するすべての Synchronization オブジェクトに通知されるまで，このメソッドは戻りません。

CosTransactions.idl の Current インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
in boolean report_heuristics	true - 経験則による決定が行われる場合にプログラムに通知するように要求します。 false - 経験則に関する情報をプログラムに返さないように要求します。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::NoTransaction	クライアントスレッドに関連付けられたトランザクションがありません。
CosTransactions::HeuristicMixed	経験則による決定が行われ，report_heuristics が true です。一部の関連する更新がコミットされ，それ以外はロールバックされました。
CosTransactions::HeuristicHazard	経験則による決定が行われた可能性があり，report_heuristics が true です。関連する更新の結果が一部不明です。結果が既知である更新は，すべてコミットされたか，すべてロールバックされています。既知の更新にコミットとロールバックの両方が含まれる場合は，HeuristicMixed 例外が生成されます。
CORBA::NO_PERMISSION	トランザクションオリジネータのスレッドだけがこのメソッドを呼び出すことができます。
CORBA::OBJECT_NOT_EXIST	別のスレッドまたはプロセスがすでにトランザクションを終了しているため，トランザクションがコミットされたか，ロールバックされたかが不明です。たとえば，トランザクションがタイムアウトになった場合です。
CORBA::TRANSACTION_ROLLEDBACK	トランザクションがロールバックされました。

関連するメソッド：

- begin()
- begin_with_name()
- Terminator インターフェースの commit()
- Control インターフェースの get_terminator()
- resume()
- rollback()

ヒューリスティックログの詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

get_control()

```
Control get_control();
```

このメソッドは Control オブジェクトリファレンスを返します。これは，現在クライアントスレッドに関連付けられているトランザクションコンテキストを表します。

クライアントスレッドがトランザクションに関連付けられていない場合は，null オブジェクトリファレンスが返されます。

注意 このメソッドを使用する意味と checked behavior の詳細については，『VisiBroker isiTransact ガイド』を参照してください。

CosTransactions.idl の Current インターフェースに含まれています。

ユーザー例外は生成されません。

関連するメソッド：

- resume()
- suspend()

関連資料として，165 ページの「Control インターフェース」と 167 ページの「Terminator インターフェース」を参照してください。詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

get_otid()

```
CosTransactions::otid_t get_otid()
raises(CosTransactions::NoTransaction,
       CosTransactions::Unavailable);
```

通常，ほとんどのアプリケーションではこのメソッドを呼び出しません。

この VISTransactions::Current メソッドは，Current インターフェースを介してオブジェクトトランザクション ID（otid）を提供するので便利です。これにより，Coordinator を介して PropagationContext を調べる必要がなくなります。otid を使用して，回復可能なオブジェクトに対するトランザクションを特定します。クライアントスレッドにトランザクションが関連付けられていない場合，このメソッドは CosTransactions::NoTransaction を生成します。

このメソッドを使用するには，resolve_initial_references() から返されたオブジェクトを VISTransactions::Current にナローイングします。詳細については，149 ページの「Current オブジェクトリファレンスの取得」を参照してください。

VISTransactions.idl の Current インターフェースに含まれています。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::NoTransaction	クライアントスレッドに関連付けられたトランザクションがありません。
CosTransactions::Unavailable	VisiTransact Transaction Service が PropagationContext の利用を制限している場合は，この例外が生成されます。

関連するメソッド：

- get_txcontext()
- Coordinator インターフェースの get_txcontext()
- get_control()

関連資料として，169 ページの「Coordinator インターフェース」および 167 ページの「Terminator インターフェース」を参照してください。

get_status()

```
Status get_status();
```

このメソッドは列挙値（enum Status）を返します。これは，クライアントスレッドに関連付けられているトランザクションの状態を表します。

このメソッドを呼び出すことは，対応する Coordinator オブジェクトの get_status() メソッドを呼び出すことと同じです。現在のスレッドに関連付けられているトランザクションがない場合，このメソッドは CosTransactions::StatusNoTransaction を返します。

戻り値は次のとおりです。

- StatusActive
- StatusMarkedRollback
- StatusPrepared
- StatusCommitted
- StatusRolledBack
- StatusUnknown
- StatusNoTransaction

- StatusPreparing
- StatusCommitting
- StatusRollingBack

CosTransactions.idl の Current インターフェースに含まれています。

ユーザー例外は生成されません。

ステータス値の定義

enum Status 値には，次の意味があります。

- StatusActive - ターゲットオブジェクトに関連付けられているトランザクションの状態がアクティブです。トランザクションにロールバックまたはタイムアウトのマークが付けられていない限り，トランザクションが開始された後で，Coordinator が Prepare 文を発行する前に，VisiTransact Transaction Service はこの状態を返します。
- StatusMarkedRollback - トランザクションはターゲットオブジェクトに関連付けられており，ロールバックのマークが付けられています。これは，通常，rollback_only() メソッドの結果です。
- StatusPrepared - トランザクションはターゲットオブジェクトに関連付けられており，準備が完了しています。
- StatusCommitted - トランザクションはターゲットオブジェクトに関連付けられており，コミットされています。通常は，経験則が存在します。そうでない場合は，トランザクションがすぐに破棄され，StatusNoTransaction が返されています。
- StatusRolledBack - トランザクションはターゲットオブジェクトに関連付けられており，その結果がロールバックと決定されています。通常は，経験則が存在します。そうでない場合は，トランザクションがすぐに破棄され，StatusNoTransaction が返されています。
- StatusUnknown - トランザクションはターゲットオブジェクトに関連付けられていますが，VisiTransact Transaction Service は現在の状態を特定できません。これは一時的な状態です。次の呼び出しでは，異なる状態が返されます。
- StatusNoTransaction - 現在，ターゲットオブジェクトに関連付けられたトランザクションはありません。トランザクションが完了すると，この値が返されます。
- StatusPreparing - ターゲットオブジェクトに関連付けられているトランザクションが準備中です。トランザクションの準備が開始され，そのプロセスが完了していない場合，VisiTransact Transaction Service はこの状態を返します。通常は，1 つ以上のリソースから準備のための応答を待機している状態です。
- StatusCommitting - ターゲットオブジェクトに関連付けられているトランザクションがコミット中です。トランザクションのコミットが開始され，そのプロセスが完了していない場合，VisiTransact Transaction Service はこの状態を返します。通常は，1 つ以上のリソースからの応答を待機している状態です。
- StatusRollingBack - ターゲットオブジェクトに関連付けられているトランザクションがロールバック中です。トランザクションがロールバック中で，そのプロセスが完了していない場合，VisiTransact Transaction Service はこの状態を返します。通常は，1 つ以上のリソースからの応答を待機している状態です。

関連するメソッド：

- Coordinator インターフェースの get_status()

get_transaction_name()

```
string get_transaction_name();
```

このメソッドは，トランザクションの名前をわかりやすく表示する文字列を返します。このメソッドは，診断とデバッグのために使用できます。begin_with_name() メソッド

によってトランザクションが作成された場合，返される文字列は，VisiTransact Transaction Service が生成した名前ではなく，ユーザーが定義してトランザクションに割り当てた名前です。

このメソッドの効果は，対応する Coordinator オブジェクトの get_transaction_name() メソッドを呼び出した場合と同じです。クライアントスレッドに関連付けられているトランザクションがない場合は，空の文字列が返されます。

CosTransactions.idl の Current インターフェースに含まれています。

ユーザー例外は生成されません。

関連するメソッド：

- begin_with_name()
- TransactionFactory インターフェースの create_with_name()
- Coordinator インターフェースの get_transaction_name()

get_txcontext()

```
CosTransactions::PropagationContext get_txcontext()
raises(CosTransactions::Unavailable,
       CosTransactions::NoTransaction);
```

通常，ほとんどのアプリケーションではこのメソッドを呼び出しません。

この VISTransactions::Current メソッドは PropagationContext を返します。この PropagationContext を VisiTransact Transaction Service ドメインで使用して，トランザクションを別の VisiTransact Transaction Service ドメインにエキスポートできます。

このメソッドを使用するには，resolve_initial_references() から返されたオブジェクトを VISTransactions::Current にナローイングします。詳細については，149 ページの「Current オブジェクトリファレンスの取得」を参照してください。

VISTransactions.idl の Current インターフェースに含まれています。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::Unavailable	VisiTransact Transaction Service が PropagationContext の利用を制限している場合は，この例外が生成されます。
CosTransactions::NoTransaction	クライアントスレッドに関連付けられたトランザクションがありません。

関連するメソッド：

- Coordinator インターフェースの get_txcontext()
- Coordinator インターフェース

関連資料として，169 ページの「Coordinator インターフェース」および 167 ページの「Terminator インターフェース」を参照してください。

ots_factory

```
attribute string ots_factory;
```

VISTransactions.idl を使用する場合は，この属性を設定してから VISTransactions::Current::begin() を呼び出すことで，トランザクションの作成に使用する VisiTransact Transaction Service のインスタンスを制御できます。その後の begin() メソッドの呼び出しでは，指定した VisiTransact Transaction Service でトランザクションが作成されます。この属性は，プログラム内のすべてのスレッドに適用されます。一度属性を設定すると，再度設定するまで値が維持されます。

この属性は，VisiTransact Transaction Service のインスタンスを IOR で指定します。VisiTransact は，指定された IOR（CosTransactions::TransactionFactory）を使用して，ネットワーク上で VisiTransact Transaction Service のインスタンスを探します。この引数により，VisiTransact は，スマートエージェント（**osagent**）を使用しなくても操作を実行できます。

ホスト名または VisiTransact Transaction Service 名のどちらかの属性で IOR を指定した場合，スマートエージェントは，VisiTransact Transaction Service のインスタンスを IOR だけで探します。つまり，他の属性は無視されます。この 3 つの属性を null のままにすると，ORB は，VisiBroker スマートエージェントを使用して VisiTransact Transaction Service のインスタンスを選択します。

VISTransactions.idl の Current インターフェースに含まれています。

この属性を設定するには，使用している言語に合わせて自動的に生成される適切なメソッドを使用します。

関連する属性：

- ots_host
- ots_name

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

ots_host

```
attribute string ots_host;
```

VISTransactions.idl を使用する場合は，この属性を設定してから VISTransactions::Current::begin() を呼び出すことで，トランザクションの作成に使用する VisiTransact Transaction Service のインスタンスを制御できます。その後の begin() メソッドの呼び出しでは，指定した VisiTransact Transaction Service でトランザクションが作成されます。この属性は，プログラム内のすべてのスレッドに適用されます。一度属性を設定すると，再度設定するまで値が維持されます。この属性をデフォルトの VisiTransact インスタンスに戻すには，これを空の文字列または null 文字列に設定します。

この属性は，VisiTransact Transaction Service のインスタンスをホスト名で指定します。スマートエージェントは，指定されたホストで，使用できる VisiTransact Transaction Service のインスタンスを探します。

ホスト名と VisiTransact Transaction Service 名の各属性を組み合わせて指定すると，スマートエージェントは，指定されたホストで指定された VisiTransact Transaction Service のインスタンスを探します。この 3 つの属性を null のままにすると，ORB は，VisiBroker スマートエージェントを使用して VisiTransact Transaction Service のインスタンスを選択します。

VISTransactions.idl の Current インターフェースに含まれています。

この属性を設定するには，使用している言語に合わせて自動的に生成される適切なメソッドを使用します。

関連する属性：

- ots_factory
- ots_name

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

ots_name

```
attribute string ots_name;
```

VISTransactions.idl を使用する場合は，この属性を設定してから VISTransactions::Current::begin() を呼び出すことで，トランザクションの作成に使用する VisiTransact Transaction Service のインスタンスを制御できます。その後の begin() メソッドの呼び出しでは，指定した VisiTransact Transaction Service でトランザクションが作成されます。この属性は，プログラム内のすべてのスレッドに適用されます。一度属性を設定すると，再度設定するまで値が維持されます。この属性をデフォルトの VisiTransact インスタンスに戻すには，これを空の文字列または null 文字列に設定します。

この属性は，VisiTransact Transaction Service のインスタンスを名前で指定します。スマートエージェントは，ネットワーク上で，指定された VisiTransact Transaction Service のインスタンスを探します。

ホスト名と VisiTransact Transaction Service 名の各属性を組み合わせて指定すると，スマートエージェントは，指定されたホストで指定された VisiTransact Transaction Service のインスタンスを探します。この 3 つの属性を null のままにすると，ORB は，VisiBroker スマートエージェントを使用して VisiTransact Transaction Service のインスタンスを選択します。

VISTransactions.idl の Current インターフェースに含まれています。

この属性を設定するには，使用している言語に合わせて自動的に生成される適切なメソッドを使用します。

関連する属性：

- ots_factory
- ots_host

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

register_resource()

```
CosTransactions::RecoveryCoordinator
     register_resource(in CosTransactions::Resource resource)
raises(CosTransactions::Inactive);
```

通常，ほとんどのアプリケーションではこのメソッドを呼び出しません。

この VISTransactions::Current メソッドは，回復可能なオブジェクトのリソースを登録します。このメソッドは，Control オブジェクトと Coordinator オブジェクトを使用して，回復可能なオブジェクトのリソースを登録するためのショートカットです。このメソッドは，回復の調整に使用される回復コーディネータオブジェクトを返します。クライアントスレッドに関連付けられているトランザクションがない場合に，このメソッドを呼び出すと，CORBA::TRANSACTION_REQUIRED 例外が生成されます。

このメソッドを使用するには，resolve_initial_references() から返されたオブジェクトを VISTransactions::Current にナローイングします。詳細については，149 ページの「Current オブジェクトリファレンスの取得」を参照してください。

VISTransactions.idl の Current インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
in CosTransactions::Resource resource	回復可能なオブジェクトのリソースオブジェクト。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::Inactive	トランザクションがすでに準備されています。
CORBA::TRANSACTION_ROLLEDBACK	トランザクションにロールバックのマークが付けられています。

関連するメソッド：

- Coordinator インターフェースの register_resource()
- get_control()

関連資料として，169 ページの「Coordinator インターフェース」を参照してください。

register_synchronization()

```
void register_synchronization(in CosTransactions::Synchronization synch)
raises(CosTransactions::NoTransaction,
       CosTransactions::Inactive,
       CosTransactions::SynchronizationUnavailable
       CosTransactions::Unavailable);
```

この VISTransactions::Current メソッドは Synchronization オブジェクトを登録します。このメソッドは，Control オブジェクトと Coordinator オブジェクトを使用して，Synchronization オブジェクトを登録するためのショートカットです。このメソッドを使用するには，resolve_initial_references() から返されたオブジェクトを VISTransactions::Current にナローイングします。詳細については，149 ページの「Current オブジェクトリファレンスの取得」を参照してください。

VISTransactions.idl の Current インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
in CosTransactions::Synchronization synch	登録する Synchronization オブジェクト。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::NoTransaction	クライアントスレッドに関連付けられたトランザクションがありません。
CosTransactions::Inactive	トランザクションがすでに準備されています。
CosTransactions::SynchronizationUnavailable	VisiBroker VisiTransact では，この例外は生成されません。
CosTransactions::Unavailable	VisiTransact Transaction Service が PropagationContext の利用を制限している場合に生成されます。

関連するメソッド：

- Coordinator インターフェースの register_synchronization()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

resume()

```
void resume(in Control which)
raises(InvalidControl);
```

クライアントスレッドを特定のトランザクションに関連付けます。通常は，次のどちらかのために使用されます。

- 暗黙的なトランザクションの伝達に使用するために，トランザクションコンテキストをスレッドに関連付ける
- suspend() メソッドによって一時停止されたトランザクションを再開する

クライアントスレッドは，特定のトランザクションに関連付けられます。クライアントスレッドがすでにほかのトランザクションに関連付けられている場合は，以前のトランザクションコンテキストが破棄されます。NULL コントロールを使用して resume() を呼び出すと，現在のスレッドに関連付けられたトランザクションはなくなり，トランザクションコンテキストは破棄されます。

注意 resume() を介して設定したトランザクションコンテキストがあれば，呼び出し元のオブジェクトに伝達されます。

CosTransactions.idl の Current インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
in Control which	スレッドのトランザクションコンテキストを設定する際に使用される Control オブジェクト。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::InvalidControl	再開のために渡された Control パラメータが現在の実行時環境で無効です。

関連するメソッド：

- get_control()
- suspend()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

rollback()

```
void rollback()
raises(NoTransaction);
```

クライアントスレッドに関連付けられているトランザクションをロールバックします。これは，対応する Terminator オブジェクトの rollback() メソッドを呼び出すことと同じです。トランザクションが完了し，関連するすべての Synchronization オブジェクトに通知されるまで，このメソッドは戻りません。このメソッドから戻った時点で，クライアントスレッドとトランザクションの関連付けが解除されます。トランザクションが存在するときと同様に Current を使用しようとすると，CosTransactions::NoTransaction または CORBA::TRANSACTION_REQUIRED などの例外が生成されるか，null オブジェクトリファレンスが返されます。経験則（ヒューリスティック）が発生すると，このメソッドは，経験則関連の例外を生成します。

呼び出し側がトランザクションオリジネータでない場合，rollback() は例外 CORBA::NO_PERMISSION を生成します。

CosTransactions.idl の Current インターフェースに含まれています。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::NoTransaction	現在のクライアントスレッドに関連付けられたトランザクションがありません。

例外	生成される条件
CORBA::NO_PERMISSION	トランザクションオリジネータのスレッドだけがこのメソッドを呼び出すことができます。
CORBA::OBJECT_NOT_EXIST	別のスレッドまたはプロセスがすでにトランザクションを終了しているため，トランザクションがコミットされたか，ロールバックされたかが不明です。たとえば，トランザクションがタイムアウトになった場合です。

関連するメソッド：

- gcommit()
- Terminator インターフェースの rollback()
- rollback_only()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

rollback_only()

```
void rollback_only()
raises(NoTransaction);
```

このメソッドは，ロールバックだけがトランザクションの結果になるように，クライアントスレッドに関連付けられているトランザクションを変更します。この要求の効果は，対応する Coordinator オブジェクトの rollback_only() メソッドを呼び出した場合と同じです。rollback() オペレーションの実行が制限されているクライアントでも，rollback_only() を呼び出すことができます。

CosTransactions.idl の Current インターフェースに含まれています。

例外

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::NoTransaction	現在のクライアントスレッドに関連付けられたトランザクションがありません。

関連するメソッド：

- rollback()
- Coordinator インターフェースの rollback_only()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

set_timeout()

```
void set_timeout(in unsigned long seconds);
```

このメソッドは，この後このプログラム内のすべてのスレッドで Current::begin() メソッドを呼び出すことによって開始されるトランザクションに対して，新しいタイムアウトを確立します。

新しいタイムアウトを確立するには，次の seconds パラメータの値を使用します。

- **= 0** - この後で開始されるトランザクションのタイムアウトを，そのトランザクションが使用する VisiTransact Transaction Service インスタンスのデフォルトタイムアウトに設定します。
- **> 0** - 新しいタイムアウトを指定した秒数に設定します。seconds パラメータが，使用される VisiTransact Transaction Service インスタンスの最大タイムアウト値を超える場合，新しいタイムアウトはその最大値に設定されて，範囲内に収められます。

メモ この後でプロセス内の任意のスレッドで begin() を呼び出すことによって作成されたトランザクションが，確立されたタイムアウトを過ぎてもトランザクションの完了を開始

できない場合，トランザクションはロールバックされます。トランザクションが完了段階に入る前に（2 フェーズ処理または 1 フェーズ処理を開始する前に）タイムアウトが発生した場合，トランザクションはロールバックされます。それ以外の場合，タイムアウトは無視されます。

CosTransactions.idl の Current インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
in unsigned long seconds	その後の begin() オペレーションで，タイムアウトになるまでの秒数。

ユーザー例外は生成されません。

関連するメソッド：

トランザクションのタイムアウトに影響する他のメソッドについては、下記を参照してください。

- TransactionFactory インターフェースの create()
- TransactionFactory インターフェースの create_with_name()

詳細については，『VisiBroker VisiTransact ガイド』で set_timeout() の説明を参照してください。

suspend()

```
Control suspend();
```

このメソッドは，クライアントスレッドに現在関連付けられているトランザクションを一時停止し，トランザクションの Control オブジェクトを返します。クライアントスレッドがトランザクションに関連付けられていない場合は，null オブジェクトリファレンスが返されます。

この Control オブジェクトを resume() メソッドに渡して，このコンテキストを同じスレッドまたは別のスレッドで再確立できます。

suspend() を呼び出すと，クライアントスレッドに関連付けられたトランザクションはなくなります。トランザクションが存在するときと同様に Current を使用しようとすると，CosTransactions::NoTransaction または CORBA::TRANSACTION_REQUIRED などの例外が生成されるか，null オブジェクトリファレンスが返されます。

CosTransactions.idl の Current インターフェースに含まれています。

ユーザー例外は生成されません。

関連するメソッド：

- get_control()
- resume()

TransactionalObject インターフェース

TransactionalObject は，トランザクションオブジェクトのメソッドを呼び出す際に，トランザクションコンテキストを自動的に伝達するためのインターフェースです。TransactionalObject インターフェースはメソッドを定義しません。

トランザクションに対して動作するメソッドには，トランザクションコンテキストに対するアクセス権が必要です。このメソッドでトランザクションコンテキストを使用できるようにするには，次の 2 つの方法があります。

- **明示的な伝達** - メソッドは，トランザクションコンテキストを Terminator，Control，Coordinator，または PropagationContext 構造体として受け渡します。詳細については，『VisiBroker VisiTransact ガイド』を参照してください。
- **暗黙的な伝達** - トランザクションコンテキストは，メソッドの呼び出し時に自動的（および暗黙的）に渡されます。詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

暗黙的な伝達の方が簡単なので，通常はこの方法が使用されます。これは，TransactionalObject インターフェースによってトランザクションオブジェクトに提供される機能です。

トランザクションコンテキストに含まれる情報の詳細については，145 ページの「構造体」を参照してください。

TransactionalObject のインスタンスは，暗黙的な伝達に参加できます。暗黙的な伝達では，クライアントスレッドに関連付けられているトランザクションコンテキストが，メソッドの呼び出しを介して TransactionalObject インターフェースに自動的に伝達されます。

VisiTransact 管理のトランザクションを使用するには，すべてのトランザクションオブジェクトが TransactionalObject を継承する必要があります。VisiTransact 管理のトランザクションを使用することで，checked behavior の機能を活用できます。

次の例は，**CosTransactions.idl** ファイルの TransactionalObject インターフェースを示しています。

```
interface TransactionalObject
{
};
```

トランザクションコンテキストは，CosTransactions::TransactionalObject を継承するオブジェクトに常に暗黙的に渡されます。また，プログラムにトランザクションコンテキストをパラメータとして明示的に渡すこともできます。

TransactionFactory インターフェース

147 ページの「Current インターフェース」で説明するように，Current インターフェースを使用して，VisiTransact 管理のトランザクションを開始できます。それに対して，ここでは，TransactionFactory インターフェースについて説明します。このインターフェースは，非 VisiTransact 管理のトランザクションを開始するためのメソッドを定義します。TransactionFactory インターフェースを使用すると，トランザクションコンテキストの伝達を直接制御できます。

CosTransactions モジュールの TransactionFactory インターフェースには，次の 3 つのメソッドが用意されています。

- create() - トランザクションを開始します。
- create_with_name() - VisiBroker VisiTransact の拡張機能を含む VISTransactions IDL インターフェースを使用している場合に使用できます。
- recreate() - トランザクションの新しい表現を作成します。

複数の IDL ファイルを使用する方法については，147 ページの「Current インターフェースの選択」を参照してください。

メモ TransactionFactory オブジェクトは，バインディングなどの CORBA オブジェクトと同じ方法で取得します。

VisiBroker VisiTransact の拡張機能に含まれるメソッドを説明または参照する箇所には， のアイコンが付けられています。

次の例は，TransactionFactory の CosTransactions IDL を示しています。

```
...
interface TransactionFactory
```

```
{
  Control create(in unsigned long time_out);
  Control recreate(in PropagationContext ctx);
};
...
```

次の例は，TransactionFactory の VISTransactions IDL を示しています。

```
...
interface TransactionFactory : CosTransactions::TransactionFactory
{
  CosTransactions::Control
    create_with_name(in unsigned long time_out,
                in string user_transaction_name);
};...
...
```

TransactionFactory のメソッド

create()

```
CosTransactions::Control create(in unsigned long time_out);
```

このメソッドは，タイムアウトパラメータ（time_out）を受け取り，新しいトランザクションを作成します。これは，Control オブジェクトを返します。この Control オブジェクトを使用して，新しいトランザクションへの参加を管理または制御できます。Control オブジェクトは任意のスレッドで使用できます。また，ほかの CORBA オブジェクトと同様に，明示的に受け渡しできます。

メモ　このメソッドを使用するトランザクションには，checked behavior を提供できません。

CosTransactions.idl の TransactionFactory インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
in unsigned long time_out	この呼び出しだけに適用されるタイムアウト（秒単位）。

新しいタイムアウトを確立するには，次の time_out パラメータの値を使用します。

- **= 0** - この後で開始されるトランザクションのタイムアウトを，そのトランザクションが使用する VisiTransact Transaction Service インスタンスのデフォルトタイムアウトに設定します。
- **> 0** - 新しいタイムアウトを指定した秒数に設定します。seconds パラメータが，使用される VisiTransact Transaction Service インスタンスの最大タイムアウト値を超える場合，新しいタイムアウトはその最大値に設定されます。

メモ　タイムアウトが発生する前に，トランザクションが完了を開始しなかった場合（2 フェーズ処理または 1 フェーズ処理が開始されなかった場合）は，トランザクションがロールバックされます。

新しいタイムアウトは，この呼び出しで作成されたトランザクションにだけ適用されます。

詳細については，『VisiBroker VisiTransact ガイド』で set_timeout() の説明を参照してください。

ユーザー例外は生成されません。

関連するメソッド：

- create_with_name()
- Control インターフェースの get_terminator()
- Control インターフェースの get_coordinator()

create_with_name()

```
CosTransactions::Control
create_with_name(in unsigned long time_out,
                 in string user_transaction_name);
```

VISTransactions メソッドは，新しいトランザクションを作成し，それにトランザクション名情報を割り当てることができるように，CosTransactions::TransactionFactory::create() メソッドを拡張します。このトランザクション名は，デバッグやエラーの報告に使用できます。ユーザー定義のトランザクション名は，get_transaction_name() から返される値に含まれています。

メモ このメソッドを使用するトランザクションには，checked behavior を提供できません。

このメソッドは，Control オブジェクトを返します。この Control オブジェクトを使用して，新しいトランザクションへの参加を管理または制御できます。Control オブジェクトは任意のスレッドで使用できます。また，ほかの CORBA オブジェクトと同様に，明示的に受け渡しできます。

VISTransactions.idl の TransactionFactory インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
in unsigned long time_out	このトランザクションのタイムアウト（秒単位）。
in string user_transaction_name	このユーザー定義のトランザクション名情報を使用して，トランザクションを追跡したり，アプリケーションをデバッグすることができます。

新しいタイムアウトを確立するには，次の time_out パラメータの値を使用します。

- **= 0** - この後で開始されるトランザクションのタイムアウトを，そのトランザクションが使用する VisiTransact Transaction Service インスタンスのデフォルトタイムアウトに設定します。
- **> 0** - 新しいタイムアウトを指定した秒数に設定します。seconds パラメータが，使用される VisiTransact Transaction Service インスタンスの最大タイムアウト値を超える場合，新しいタイムアウトはその最大値に設定されます。

メモ タイムアウトが発生する前に，トランザクションが完了を開始しなかった場合（2 フェーズ処理または 1 フェーズ処理が開始されなかった場合）は，トランザクションがロールバックされます。

新しいタイムアウトは，この呼び出しで作成されたトランザクションにだけ適用されます。

詳細については，『VisiBroker VisiTransact ガイド』で set_timeout() の説明を参照してください。

ユーザー例外は生成されません。

関連するメソッド：

- create()
- Control インターフェースの get_terminator()
- Control インターフェースの get_coordinator()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

recreate()

```
Control recreate(in PropagationContext context);
```

通常，ほとんどのアプリケーションではこのメソッドを呼び出しません。

このメソッドは，PropagationContext パラメータを使用して新しい Control オブジェクトを作成します。この Control オブジェクトを使用して，トランザクションへの参加を管理または制御できます。

トランザクションの PropagationContext を取得するには，トランザクションの Coordinator オブジェクトの get_txcontext() メソッドを呼び出します。

CosTransactions.idl の TransactionFactory インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
in PropagationContext context	インポートするトランザクションのコンテキスト。

ユーザー例外は生成されません。

次の例は，get_txcontext() と recreate() を使用して，トランザクションを再作成する方法を示しています。

```
⋮
CosTransactions::Coordinator_var coord;
CosTransactions::TransactionFactory_var newDomainFactory;
⋮
propxtxt = coord->get_txcontext();
CosTransactions::Control_var control = newDomainFactory->recreate(propctxt);
⋮
```

関連するメソッド：

- Current インターフェースの get_txcontext()
- Coordinator インターフェースの get_txcontext()

Control インターフェース

Control インターフェースを使用すると，プログラムでトランザクションコンテキストを明示的に管理または伝達できます。Control オブジェクトは，1 つの特定のトランザクションに暗黙的に関連付けられます。

Control インターフェースは，次の 2 つのメソッドを定義します。

- get_coordinator()
- get_terminator()

get_coordinator() メソッドは Coordinator オブジェクトを返します。このオブジェクトは，トランザクションの参加者によって使用されるメソッドをサポートします。get_terminator() メソッドは Terminator オブジェクトを返します。このオブジェクトは，トランザクションを完了するためのメソッドをサポートします。Terminator オブジェクトと Coordinator オブジェクトがサポートするメソッドは，通常，複数の当事者によって実行されます。この 2 つのオブジェクトにより，これらのメソッドを必要とする当事者にだけメソッドを提供できます。

次の例は，**CosTransactions.idl** ファイルから抜粋された Control インターフェースの IDL を示しています。

```
...
interface Control
{
  Terminator get_terminator()
    raises(Unavailable);
  Coordinator get_coordinator()
    raises(Unavailable);
};
...
```

Control オブジェクトを取得するには，TransactionFactory のメソッドを使用します。詳細については，162 ページの「TransactionFactory インターフェース」を参照してください。また，Current オブジェクトのメソッドを使用して，スレッドに関連付けられている現在のトランザクションに対する Control オブジェクトを取得することもできます。

Control のメソッド

get_coordinator()

```
Coordinator get_coordinator()
raises(Unavailable);
```

このメソッドは，Coordinator オブジェクトを返します。Coordinator は，トランザクションの参加者によって呼び出されるメソッドを提供します。通常，これらの参加者は，回復可能なオブジェクトまたは回復可能なオブジェクトのエージェントです。

CosTransactions.idl の Control インターフェースに含まれています。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::Unavailable	Control オブジェクトが，要求された Coordinator オブジェクトを提供できません。

関連するメソッド：

- Current インターフェースの get_control()

Coordinator オブジェクトを取得することによって使用できるメソッドについては，169 ページの「Coordinator インターフェース」を参照してください。

get_terminator()

```
Terminator get_terminator()
raises(Unavailable);
```

このメソッドは，Terminator オブジェクトを返します。Terminator を使用して，Control に関連付けられたトランザクションをロールバックまたはコミットできます。Terminator オブジェクトを送信できないか，ほかの実行環境で使用できないために，Control が要求されたオブジェクトを提供できない場合は，例外 Unavailable が生成されます。

CosTransactions.idl の Control インターフェースに含まれています。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::Unavailable	Control オブジェクトが，要求された Terminator オブジェクトを提供できません。

関連するメソッド：

- Current インターフェースの get_control()
- get_coordinator()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

Terminator インターフェース

Terminator インターフェースは，トランザクションをコミットまたはロールバックするためのメソッドをサポートします。通常，このメソッドは，トランザクションオリジネータによって使用されます。ただし，トランザクションの Terminator オブジェクトにアクセスできる場合は，任意のプログラムがトランザクションをコミットまたはロールバックできます。

次の例は，**CosTransactions.idl** ファイルから抜粋された Terminator インターフェースの IDL を示しています。

```
...
interface Terminator
{
  void commit(in boolean report_heuristics)
    raises (HeuristicMixed,
            HeuristicHazard);
  void rollback();
};
...
```

Terminator のメソッド

commit()

```
void commit(in boolean report_heuristics)
raises(HeuristicMixed,
       HeuristicHazard);
```

トランザクションをコミットする前に，このメソッドはいくつかのチェックを行います。トランザクションにロールバックのみのマークが付けられておらず，トランザクションのすべての参加者がコミットに同意する場合，トランザクションはコミットされ，オペレーションは正常に終了します。それ以外の場合は，トランザクションがロールバックされ，標準の例外 CORBA::TRANSACTION_ROLLEDBACK が生成されます。

report_heuristics パラメータが true の場合，VisiTransact Transaction Service は，必要に応じて例外 CosTransactions::HeuristicMixed と CosTransactions::HeuristicHazard を使用して，矛盾した（またはその可能性がある）結果が生成されることを報告します。経験則出力に関連するリソースについての情報が，VisiTransact Transaction Service のインスタンスに対応する経験則ログファイルに書き込まれます。経験則の詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

トランザクションがコミットされると，このトランザクションの間に回復可能なオブジェクトに加えられた変更がすべて確定され，ほかのトランザクションやクライアントから認識できるようになります。

CosTransactions.idl の Terminator インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
report_heuristics	true - 必要に応じて，HeuristicMixed または HeuristicHazard 例外の生成を要求します。 false - 経験則に関する情報をプログラムに返さないように要求します。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::HeuristicMixed	経験則による決定が行われました。一部の関連する更新がコミットされ，それ以外はロールバックされました。
CosTransactions::HeuristicHazard	経験則による決定が行われた可能性があり，すべての関連する更新の結果が不明です。結果が既知である更新は，すべてコミットされたか，すべてロールバックされています。既知の更新にコミットとロールバックの両方が含まれる場合は，HeuristicMixed 例外が生成されます。
CORBA::TRANSACTION_ROLLEDBACK	トランザクションにロールバックのマークが付けられています。
CORBA::OBJECT_NOT_EXIST	別のスレッドまたはプロセスがすでにトランザクションを終了しているため，トランザクションがコミットされたか，ロールバックされたかが不明です。たとえば，トランザクションがタイムアウトになった場合です。

次の例は，commit() で経験則を使用する方法と使用しない方法を示しています。

```
// commit() で経験則を使用しない場合
try
{
   terminator->commit(0);
}
catch(CORBA::TRANSACTION_ROLLEDBACK&)
{
   cerr << "Transaction failed" << endl;
}
...
// commit() で経験則を使用する場合
try
{
   terminator->commit(1);
}
catch(CORBA::TRANSACTION_ROLLEDBACK&)
{
   cerr << "Transaction failed" << endl;
}
catch(CosTransactions::HeuristicMixed&)
{
   cerr << "HeuristicMixed exception was raised" << endl;
}
catch(CosTransactions::HeuristicHazard&)
{
   cerr << "HeuristicHazard exception was raised" << endl;
}
catch(CORBA::OBJECT_NOT_EXIST&)
{
   cerr << "Transaction no longer exists" << endl;
}
```

関連するメソッド：

- Current インターフェースの commit()
- rollback()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

rollback()

```
void rollback();
```

このメソッドは，トランザクションをロールバックします。トランザクションがロールバックされると，このトランザクションの間に回復可能なオブジェクトに加えられた変更がすべてロールバックされます。リソースによって適用されるアイソレーションの程度に応じて，トランザクションによってロックされたすべてのリソースがほかのトランザクションでも使用できるようになります。

トランザクションが完了し，関連するすべての Synchronization オブジェクトに通知されるまで，このメソッドは戻りません。経験則出力は，コンソールを介して提供されます。

CosTransactions.idl の Terminator インターフェースに含まれています。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CORBA::NO_PERMISSION	トランザクションオリジネータのスレッドだけがこのメソッドを呼び出すことができます。
CORBA::OBJECT_NOT_EXIST	別のスレッドまたはプロセスがすでにトランザクションを終了しているため，トランザクションがコミットされたか，ロールバックされたかが不明です。たとえば，トランザクションがタイムアウトになった場合です。

関連するメソッド：

- commit()
- Current インターフェースの rollback()
- Coordinator インターフェースの rollback_only()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

Coordinator インターフェース

Coordinator インターフェースは，トランザクションの参加者によって使用されるメソッドを提供します。通常，これらの参加者は，回復可能なオブジェクトまたは回復可能なオブジェクトのエージェントです。各 Coordinator は，1 つのトランザクションに暗黙的に関連付けられます。

次の例は，Coordinator インターフェースの CosTransactions IDL を示しています。

```
...
interface Coordinator
{
  Status get_status();
  Status get_parent_status();
  Status get_top_level_status();

  boolean is_same_transaction(in Coordinator coord);
  boolean is_related_transaction(in Coordinator coord);
  boolean is_ancestor_transaction(in Coordinator coord);
  boolean is_descendant_transaction(in Coordinator coord);
  boolean is_top_level_transaction();

  unsigned long hash_transaction();
  unsigned long hash_top_level_tran();

  RecoveryCoordinator register_resource(in Resource resource)
    raises(Inactive);

  void register_synchronization(in Synchronization synch)
```

```
    raises(Inactive, SynchronizationUnavailable);

  void register_subtran_aware(in SubtransactionAwareResource resource)
    raises(Inactive, NotSubtransaction);

  void rollback_only()
    raises(Inactive);

  string get_transaction_name();

  Control create_subtransaction()
    raises(SubtransactionsUnavailable, Inactive);

  PropagationContext get_txcontext()
    raises(Unavailable);
};
```

VisiTransact はネストしたトランザクションをサポートしていません。そのため，Coordinator のいくつかのメソッドは同等になり，同じ結果を返します。メソッドの詳細については，この後で説明します。

次のメソッドは等価です。

- get_status()
- get_top_level_status()
- get_parent_status()

同様に，いくつかのメソッドは，ターゲットオブジェクトとパラメータが同じ Coordinator オブジェクトを参照する場合にだけ true を返します。したがって，次のメソッドも同等です。

- is_same_transaction()
- is_related_transaction()
- is_ancestor_transaction()
- is_descendant_transaction()

また，次のメソッドは同等です。

- hash_transaction()
- hash_top_level_tran()

ネストされたトランザクションがない場合，create_subtransaction() メソッドを使用する意味はありません。そのため，このメソッドは，OMG 仕様には記載されていますが，このバージョンの VisiTransact とマニュアルからは除外されています。

Coordinator のメソッド

get_parent_status()

```
Status get_parent_status();
```

VisiTransact はネストしたトランザクションをサポートしていません。そのため，各トランザクションが最上位のトランザクションになります。また，OMG 定義により，最上位のトランザクションの get_parent_status() は get_status() と同等です。詳細については，get_status() を参照してください。

CosTransactions.idl の Coordinator インターフェースに含まれています。

関連するメソッド：

- get_status()

get_status()

```
Status get_status();
```

このメソッドは，ターゲットオブジェクトに関連付けられているトランザクションの状態を列挙値（enum Status）で返します。ターゲットオブジェクトに関連付けられているトランザクションがない場合，このメソッドは値 StatusNoTransaction を返します。

戻り値は次のとおりです。これらは **CosTransactions.idl** で定義されます。

- StatusActive
- StatusMarkedRollback
- StatusPrepared
- StatusCommitted
- StatusRolledBack
- StatusUnknown
- StatusNoTransaction
- StatusPreparing
- StatusCommitting
- StatusRollingBack

各 Status 値の詳細については，154 ページの「ステータス値の定義」を参照してください。

CosTransactions.idl の Coordinator インターフェースに含まれています。

関連するメソッド：

- get_parent_status()
- Current インターフェースの get_status()
- get_top_level_status()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

get_top_level_status()

```
Status get_top_level_status();
```

VisiTransact はネストしたトランザクションをサポートしていないため，各トランザクションが最上位のトランザクションになります。そのため，このメソッドは get_status() メソッドと同等です。

CosTransactions.idl の Coordinator インターフェースに含まれています。

関連するメソッド：

- get_status()

get_transaction_name()

```
string get_transaction_name();
```

このメソッドは，トランザクションの名前をわかりやすく表示する文字列を返します。このメソッドは，診断とデバッグのために使用できます。VISTransactions::TransactionFactory::create_with_name() メソッドによってトランザクションが作成された場合，返される文字列は，VisiTransact Transaction Service が生成した名前ではなく，ユーザーが説明のために定義したトランザクション名です。クライアントスレッドに関連付けられているトランザクションがない場合は，空の文字列が返されます。

CosTransactions.idl の Coordinator インターフェースに含まれています。

関連するメソッド：

- Current インターフェースの begin_with_name()
- TransactionFactory インターフェースの create_with_name()

- Current インターフェースの get_transaction_name()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

get_txcontext()

```
PropagationContext get_txcontext()
raises(Unavailable);
```

通常，ほとんどのアプリケーションではこのメソッドを呼び出しません。

get_txcontext() メソッドは PropagationContext を返します。この PropagationContext を VisiTransact Transaction Service ドメインで使用して，トランザクションを別の VisiTransact Transaction Service ドメインにエキスポートできます。

CosTransactions.idl の Coordinator インターフェースに含まれています。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::Unavailable	VisiTransact Transaction Service が PropagationContext の利用を制限しています。

関連するメソッド：

- Current インターフェースの get_control()
- TransactionFactory インターフェースの create_with_name()
- TransactionFactory インターフェースの recreate()

hash_top_level_tran()

```
unsigned long hash_top_level_tran();
```

通常，ほとんどのアプリケーションではこのメソッドを呼び出しません。

VisiTransact はネストしたトランザクションをサポートしていないため，各トランザクションが最上位のトランザクションになります。そのため，このメソッドは hash_transaction() メソッドと同等です。

このメソッドは，ターゲットオブジェクトに関連付けられているトランザクションのハッシュコードを返します。各トランザクションは 1 つのハッシュコードを持ちます。このハッシュコードをほかのトランザクションのハッシュコードと照らし合わせることで，Coordinator が同じかどうかを効率よく確認できます。2 つの Coordinator のハッシュコードが異なる場合，これは別のトランザクションを表しています。2 つのハッシュコードが等しい場合は，Coordinator::is_same_transaction() を使用して，それらが等しいかどうかを確認する必要があります。これは，2 つの Coordinator のハッシュコードが等しくても，実際には異なるトランザクションを表すことがあるためです。

CosTransactions.idl の Coordinator インターフェースに含まれています。

関連するメソッド：

- hash_transaction()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

hash_transaction()

```
unsigned long hash_transaction();
```

通常，ほとんどのアプリケーションではこのメソッドを呼び出しません。

このメソッドは，ターゲットオブジェクトに関連付けられているトランザクションのハッシュコードを返します。各トランザクションは 1 つのハッシュコードを持ちます。このハッシュコードをほかのトランザクションのハッシュコードと照らし合わせることで，

Coordinator が同じかどうかを効率よく確認できます。2 つの Coordinator のハッシュコードが異なる場合，これは別のトランザクションを表しています。2 つのハッシュコードが等しい場合は，Coordinator::is_same_transaction() を使用して，それらが等しいかどうかを確認する必要があります。これは，2 つの Coordinator のハッシュコードが等しくても，実際には異なるトランザクションを表すことがあるためです。

CosTransactions.idl の Coordinator インターフェースに含まれています。

次の例は，hash_transaction() を使用して，等しくない Coordinator どうしを効率よく除外するメソッドを示しています。

```
CORBA::Boolean are_same(Coord1, Coord2)
{
  CORBA::ULong hash1 = Coord1->hash_transaction();
  CORBA::ULong hash2 = Coord2->hash_transaction();
  if(hash1 != hash2)
  {
    return 0;
  }
  else
  {
    return Coord1->is_same_transaction(Coord2);
  }
}
```

関連するメソッド：

- hash_top_level_tran()
- is_same_transaction()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

is_ancestor_transaction()

```
boolean is_ancestor_transaction(in Coordinator coord);
```

VisiTransact はネストしたトランザクションをサポートしていないため，ターゲットオブジェクトとパラメータオブジェクトが同じトランザクションを参照する場合にだけ，このメソッドは true を返します。

CosTransactions.idl の Coordinator インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
coord	ターゲット Coordinator と比較される Coordinator。

関連するメソッド：

- is_same_transaction()

is_descendant_transaction()

```
boolean is_descendant_transaction(in Coordinator coord);
```

VisiTransact はネストしたトランザクションをサポートしていないため，ターゲットオブジェクトとパラメータオブジェクトが同じトランザクションを参照する場合にだけ，このメソッドは true を返します。

CosTransactions.idl の Coordinator インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
coord	ターゲット Coordinator と比較される Coordinator。

関連するメソッド：

- is_same_transaction()

is_related_transaction()

```
boolean is_related_transaction(in Coordinator coord);
```

VisiTransact はネストしたトランザクションをサポートしていないため，ターゲットオブジェクトとパラメータオブジェクトが同じトランザクションを参照する場合にだけ，このメソッドは true を返します。

CosTransactions.idl の Coordinator インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
coord	ターゲット Coordinator と比較される Coordinator。

関連するメソッド：

- is_same_transaction()

is_same_transaction()

```
boolean is_same_transaction(in Coordinator coord);
```

ターゲットオブジェクトとパラメータオブジェクトが同じトランザクションを参照する場合にだけ，このメソッドは true を返します。

CosTransactions.idl の Coordinator インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
coord	ターゲット Coordinator と比較される Coordinator。

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

is_top_level_transaction()

```
boolean is_top_level_transaction(in Coordinator coord);
```

VisiTransact ではネストしたトランザクションはサポートされていないため，このメソッドは常に true を返します。

CosTransactions.idl の Coordinator インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
coord	ターゲット Coordinator と比較される Coordinator。

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

register_resource()

```
RecoveryCoordinator register_resource(in Resource resource)
raises(Inactive);
```

このメソッドは，指定されたリソースをターゲットオブジェクトに関連付けられたトランザクションの参加者として登録します。トランザクションが終了すると，リソースは，トランザクションの間に実行された更新を準備，コミット，またはロールバックする要

求を受け取ります。Resource のメソッドの詳細については，177 ページの「Resource インターフェース」を参照してください。

このメソッドは，このリソースによってリカバリ中に使用される RecoveryCoordinator を返します。

CosTransactions.idl の Coordinator インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
resource	登録するリソースオブジェクト。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::Inactive	トランザクションの準備が完了している場合は，この例外が生成されます。
CORBA::TRANSACTION_ROLLEDBACK	トランザクションにロールバックのマークが付けられている場合は，この例外が生成されます。

関連するメソッド：

- Current インターフェースの register_resource()

詳細については，176 ページの「RecoveryCoordinator インターフェース」と 177 ページの「Resource インターフェース」を参照してください。

register_synchronization()

```
void register_synchronization(in Synchronization synch)
raises(Inactive, SynchronizationUnavailable);
```

このメソッドは，トランザクションが完了する前後に，指定された Synchronization オブジェクトが必要な処理を実行するための通知を受けるように，このオブジェクトを登録します。このメソッドについては，Synchronization インターフェースの解説の中で説明されています。181 ページの「Synchronization インターフェース」を参照してください。

CosTransactions.idl の Coordinator インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
synch	登録する Synchronization オブジェクト。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::Inactive	トランザクションの準備が完了している場合は，この例外が生成されます。
CosTransactions::SynchronizationUnavailable	この例外は生成されません。
CORBA::TRANSACTION_ROLLEDBACK	トランザクションにロールバックのマークが付けられている場合は，この例外が生成されます。

関連するメソッド：

- Current インターフェースの register_synchronization()

詳細については，177 ページの「Resource インターフェース」，181 ページの「Synchronization インターフェース」，および『VisiBroker VisiTransact ガイド』を参照してください。

register_subtran_aware()

```
void register_subtran_aware(in SubtransactionAwareResource resource)
raises(Inactive, SubtransactionsUnavailable);
```

VisiTransact ではネストしたトランザクションはサポートされていないため，このメソッドは常に CosTransactions::SubtransactionsUnavailable を生成します。

CosTransactions.idl の Coordinator インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
resource	サブトランザクションに登録されるリソース。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::Inactive	この例外は生成されません。
CosTransactions::SubtransactionsUnavailable	このメソッドを呼び出すたびに生成されます。

関連するメソッド：

- register_resource()

rollback_only()

```
void rollback_only()
raises (Inactive);
```

このメソッドは，ロールバックだけがトランザクションの結果になるように，Coordinator に関連付けられているトランザクションを変更します。

CosTransactions.idl の Coordinator インターフェースに含まれています。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::Inactive	トランザクションの準備が完了している場合は，この例外が生成されます。

関連するメソッド：

- Current インターフェースの get_coordinator()
- Current インターフェースの rollback_only()

rollback_only() を呼び出す方法については，『VisiBroker VisiTransact ガイド』を参照してください。

RecoveryCoordinator インターフェース

Coordinator にリソースを登録すると，RecoveryCoordinator が返されます。RecoveryCoordinator は，暗黙的に 1 つのリソース登録要求に関連付けられ，そのリソースによってのみ使用されます。リカバリが必要な場合，リソースはリカバリ処理で RecoveryCoordinator を使用できます。

また，トランザクションの現在の状態を取得する必要がある場合にも，RecoveryCoordinator を使用できます。たとえば，リソースは独自のタイムアウトを設定できるので，タイムアウトまでにコミットやロールバックが行われなかった場合は，replay_completion() を呼び出してトランザクションの状態を特定できます。

次の例は，**CosTransactions.idl** ファイルの RecoveryCoordinator インターフェースを示しています。

```
...
interface RecoveryCoordinator
{
  Status replay_completion(in Resource resource)
    raises(NotPrepared);
};
...
```

RecoveryCoordinator のメソッド

replay_completion()

```
Status replay_completion(Resource resource)
raises(NotPrepared);
```

このメソッドは，リソースが使用可能であることを VisiTransact Transaction Service に通知します。通常，このメソッドはリカバリ中に使用されます。また，リソースは，このメソッドを使用してトランザクションの状態を特定できます。

メモ　このメソッドは，完了を開始しません。

CosTransactions.idl の RecoveryCoordinator インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
resource	リカバリを実行するリソース。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::NotPrepared	準備が完了していないリソースに対して replay_completion() を呼び出すと，この例外が生成されます。

関連するメソッド：

- Resource インターフェースの commit()
- Current インターフェースの register_resource()
- Coordinator インターフェースの register_resource()
- Resource インターフェースの rollback()

Status 値の詳細については，147 ページの「Current インターフェース」の 154 ページの「ステータス値の定義」を参照してください。

Resource インターフェース

VisiBroker VisiTransact では，2 フェーズコミットプロトコルを使用して，各リソースが登録されている最上位のトランザクションを完了します。つまり，この各リソースは，トランザクションの間に変更される可能性があります。Resource インターフェースは，各リソースで VisiTransact Transaction Service によって呼び出されるメソッドを定義します。Resource インターフェースをサポートする各オブジェクトは，1 つの最上位トランザクションに暗黙的に関連付けられます。

次の例は，**CosTransactions.idl** ファイルの Resource インターフェースを示しています。

```
...
interface Resource
```

```
{
  Vote prepare()
    raises(
        HeuristicMixed,
        HeuristicHazard
    );
  void rollback()
    raises(
        HeuristicCommit,
        HeuristicMixed,
        HeuristicHazard
    );
  void commit()
    raises(
        NotPrepared,
        HeuristicRollback,
        HeuristicMixed,
        HeuristicHazard
    );
  void commit_one_phase()
    raises(
        HeuristicHazard
    );
  void forget();
};
...
```

このインターフェースは VisiBroker VisiTransact から提供されますが，Resource を使用する場合は，そのインプリメンテーションを提供する必要があります。通常のアプリケーションは，Resource を実装しません。

Resource のメソッド

commit()

```
void commit()
raises(NotPrepared
       HeuristicRollback
       HeuristicMixed
       HeuristicHazard
  );
```

このメソッドは，リソースに関連付けられているすべての変更をコミットしようとします。経験則出力の例外が生成された場合，リソースは，forget() メソッドが実行されるまで，経験則による決定を永続的ストレージに保存する必要があります。これにより，リカバリ中に commit() が再度呼び出されても，同じ結果を返すことができます。それ以外の場合，リソースは，トランザクションに関するすべての情報をすぐに破棄できます。

CosTransactions.idl の Resource インターフェースに含まれています。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::NotPrepared	prepare() メソッドが呼び出される前に，commit() メソッドが呼び出されました。
CosTransactions::HeuristicRollback	経験則による決定が行われ，すべての関連する更新がロールバックされました。

例外	生成される条件
CosTransactions::HeuristicMixed	経験則による決定が行われました。一部の関連する更新がコミットされ，それ以外はロールバックされました。
CosTransactions::HeuristicHazard	経験則による決定が行われた可能性があり，すべての関連する更新の結果が不明です。結果が既知である更新は，すべてコミットされたか，すべてロールバックされています。既知の更新にコミットとロールバックの両方が含まれる場合は，HeuristicMixed 例外が生成されます。

関連するメソッド：

- commit_one_phase()
- rollback()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

commit_one_phase()

```
void commit_one_phase()
raises (HeuristicHazard);
```

commit_one_phase() メソッドは，トランザクションの間に加えられたすべての変更をコミットするようにリソースに要求します。このメソッドは，トランザクションに参加するリソースが 1 つしかない場合に使用するように最適化されています。リソースでは，最初に prepare() を呼び出し，次に commit() または rollback() を呼び出すかわりに，commit_one_phase() メソッドを呼び出すことができます。

経験則出力の例外が生成された場合，リソースは，forget() メソッドが実行されるまで，経験則による決定を永続的ストレージに保存する必要があります。これにより，リカバリ中に commit_one_phase() が再度実行されても，同じ結果を返すことができます。それ以外の場合，リソースは，トランザクションに関するすべての情報をすぐに破棄できます。

CosTransactions.idl の Resource インターフェースに含まれています。

commit_one_phase() の処理中にエラーが発生した場合は，エラーが修復されると，同じメソッドが再度呼び出されます。リソースは 1 つしかないため，例外 HeuristicHazard を使用して，そのリソースに関連する経験則による決定が報告されます。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::HeuristicHazard	経験則による決定が行われた可能性があり，すべての関連する更新の結果が不明です。結果が既知である更新は，すべてコミットされたか，すべてロールバックされています。
CORBA::TRANSACTION_ROLLEDBACK	commit_one_phase() メソッドが，トランザクションの間に加えられたすべての変更をコミットできません。

関連するメソッド：

- commit()
- forget()
- prepare()
- rollback()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

forget()

```
void forget();
```

VisiBroker VisiTransact は，経験則の例外を受け取ると，その例外を記録します。VisiTransact Transaction Service は，最終的にリソースの forget() を呼び出します。つまり，リソースは，経験則の例外を生成したトランザクションに関するすべての情報を破棄できます。このメソッドは，経験則の例外が rollback()，commit()，または commit_one_phase() から生成された場合にだけ呼び出されます。

CosTransactions.idl の Resource インターフェースに含まれています。

関連するメソッド：

- commit()
- commit_one_phase()
- rollback()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

prepare()

```
Vote prepare()
raises(HeuristicMixed
       HeuristicHazard
);
```

このメソッドは準備処理を実行します。これは，Resource オブジェクトの 2 フェーズコミットプロトコルにおける最初の手順です。処理が完了すると，このメソッドは次の Vote 値のいずれかを返します。

- VoteReadOnly - リソースに関連付けられた永続的データは，トランザクションによって変更されていません。
- VoteCommit - 次のデータが永続的ストレージに保存されています。
  1. トランザクションの間に変更されたすべてのデータ
  2. RecoveryCoordinator オブジェクトへの参照
  3. リソースの準備が整ったことを示すデータ

VoteRollback - 関連データを保存できない，結果に一貫性がない，トランザクションに関する情報がない（たとえば，クラッシュ後の状態）など，リソースがロールバックを要求する状況が発生しました。

VoteReadOnly または VoteRollback が返された場合，リソースは，トランザクションに関するすべての情報を破棄できます。

経験則出力の例外が生成された場合，リソースは，forget() メソッドが呼び出されるまで，経験則による決定を永続的ストレージに保存する必要があります。これにより，prepare() が再度呼び出されても，同じ結果を返すことができます。

CosTransactions.idl の Resource インターフェースに含まれています。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::HeuristicMixed	経験則による決定が行われました。一部の関連する更新がコミットされ，それ以外はロールバックされました。
CosTransactions::HeuristicHazard	経験則による決定が行われた可能性があり，すべての関連する更新の結果が不明です。結果が既知である更新は，すべてコミットされたか，すべてロールバックされています。既知の更新にコミットとロールバックの両方が含まれる場合は，HeuristicMixed 例外が生成されます。

関連するメソッド：

- commit_one_phase()
- Current インターフェースの register_resource()
- Coordinator インターフェースの register_resource()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

rollback()

```
void rollback()
raises(HeuristicCommit
       HeuristicMixed
       HeuristicHazard
);
```

このメソッドは，リソースオブジェクトに関連付けられているすべての更新をロールバックします。

経験則出力の例外が生成された場合，リソースは，forget() メソッドが呼び出されるまで，経験則による決定を永続的ストレージに保存する必要があります。これにより，リカバリ中に rollback() が再度呼び出されても，同じ結果を返すことができます。それ以外の場合，リソースは，トランザクションに関するすべての情報をすぐに破棄できます。

CosTransactions.idl の Resource インターフェースに含まれています。

このメソッドを呼び出すと，次の例外が生成されることがあります。

例外	生成される条件
CosTransactions::HeuristicCommit	経験則による決定が行われ，すべての関連する更新がコミットされました。
CosTransactions::HeuristicMixed	経験則による決定が行われました。一部の関連する更新がコミットされ，それ以外はロールバックされました。
CosTransactions::HeuristicHazard	経験則による決定が行われた可能性があり，すべての関連する更新の結果が不明です。結果が既知である更新は，すべてコミットされたか，すべてロールバックされています。既知の更新にコミットとロールバックの両方が含まれる場合は，HeuristicMixed 例外が生成されます。

関連するメソッド：

- commit()
- commit_one_phase()
- forget()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

Synchronization インターフェース

Synchronization インターフェースは，2 フェーズコミットプロトコルまたは 1 フェーズコミットプロトコルを開始する前と，それが完了した後に，トランザクションオブジェクトが通知を受けるためのメソッドを定義します。詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

CosTransactions モジュールの Synchronization インターフェースには，次の 2 つのメソッドが用意されています。

- before_completion() - トランザクションのコミットを開始する前に，before_completion() が呼び出されます。
- after_completion() - トランザクションが完了した後で，トランザクションオブジェクトが通知を受けます。これは，トランザクションがコミットされてもロールバックされても，すべてのトランザクションに適用されます。

次の 2 つの制限に注意する必要があります。

- before_completion() を呼び出す際に，VisiTransact Transaction Service が Synchronization オブジェクトにアクセスできない場合，トランザクションはロールバックされます。完了後に Synchronization オブジェクトを使用できない場合，そのオブジェクトは無視されます。
- VisiTransact Transaction Service インスタンスがエラーから修復されても，Synchronization オブジェクトは失われたままです。完了は再開されますが，Synchronization オブジェクトは元に戻りません。エラーが発生した場合，VisiTransact Transaction Service によってトランザクションが完了した状況は，Synchronization オブジェクトに通知されません。

メモ before_completion() が呼び出されなかった場合に，after_completion() が呼び出されることがあります。完了処理の開始時にトランザクションがまだコミットを継続している場合にだけ，before_completion() が呼び出されます。after_completion() は常に呼び出されます。ただし，トランザクションが完了する前に，VisiTransact Transaction Service がクラッシュした場合を除きます。

Synchronization オブジェクトを回復することはできません。VisiTransact Transaction Service のインスタンスにエラーが発生した場合，完了したトランザクションに Synchronization オブジェクトは含まれません。

メモ これらのメソッドのシグニチャは Synchronization インターフェースによって固定されていますが，インプリメンテーションはユーザーによって定義されます。これにより，アプリケーションでは，トランザクションが完了する前および後のトランザクションのキーポイントで処理をカスタマイズできます。

次の例は，Synchronization インターフェースの CosTransactions IDL を示しています。

```
...
interface Synchronization : TransactionalObject
  {
    void before_completion();
    void after_completion(in Status status);
  };
...
```

Synchronization のメソッド

after_completion()

```
void after_completion(in Status status);
```

これは，トランザクションの完了後にカスタマイズされた処理を実行するためのメソッドとして，ユーザーによって記述されます。これは，本質的にコールバックです。

メモ after_completion() メソッドは，通常の処理中に常に呼び出されます。

上記のように，Synchronization インターフェースの IDL は TransactionalObject インターフェースを継承します。プログラマは，この IDL に準拠する after_completion() メソッドのインプリメンテーションを記述する必要があります。

特定のトランザクションを処理する際に after_completion() を呼び出す場合は，次の操作を行う必要があります。

- Synchronization オブジェクトを作成します。これは，トランザクションオリジネータまたはほかのトランザクション参加者が行います。
- Synchronization オブジェクトを登録します。それには，トランザクションの Coordinator を取得し，Coordinator と Current で register_synchronization() メソッドを呼び出します。詳細については，169 ページの「Coordinator インターフェース」の 175 ページの「register_synchronization() void register_synchronization(in Synchronization synch) raises(Inactive, SynchronizationUnavailable);」を参照し

てください。登録は，トランザクションが作成された後で，2 フェーズコミットプロトコルが開始される前に，行う必要があります。

複数の Synchronization オブジェクトを作成し，それらを 1 つのトランザクションに登録することができます。

VisiTransact Transaction Service は，2 フェーズコミットプロトコルの完了後に，このメソッドを呼び出します。たとえば，トランザクションオブジェクトで after_completion() を使用して，トランザクションの結果を検索できます。これは，回復可能なオブジェクトではなく，結果が自動的に通知されないトランザクションオブジェクトの場合に特に便利です。

get_status() を呼び出して，トランザクションにロールバックのマークが付けられているかどうかを確認することもできます。

Synchronization は TransactionalObject を継承するため，Current オブジェクトを介してトランザクションコンテキストを使用できます。

CosTransactions.idl の Synchronization インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
status	トランザクションの結果が決定した後で，Terminator によって Synchronization オブジェクトに渡された Status 値。有効な Status 値については，「Current インターフェース」の「ステータス値の定義」を参照してください。

例外はすべて無視されます。

関連するメソッド：

- before_completion()
- Current インターフェースの get_status()
- Terminator インターフェースの commit()
- Current インターフェースの register_synchronization()
- Coordinator インターフェースの register_synchronization()
- Current インターフェースの rollback_only()
- Coordinator インターフェースの rollback_only()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

before_completion()

```
void before_completion();
```

これは，トランザクションの完了が開始されたときにカスタマイズされた処理を実行するためのメソッドとして，ユーザーによって記述されます。このメソッドは，トランザクションが正常に完了処理を継続している場合にだけ呼び出されます。これは，本質的にコールバックです。

メモ アプリケーションが commit() を呼び出すと，VisiTransact Transaction Service がトランザクションの完了を開始する前に，before_completion() メソッドが呼び出されます。ロールバック要求の場合，before_completion() メソッドは呼び出されません。

この節の最初に示したように，Synchronization インターフェースの IDL は TransactionalObject インターフェースを継承します。プログラマは，この IDL に準拠する before_completion() メソッドのインプリメンテーションを記述する必要があります。

特定のトランザクションを処理する際に before_completion() を呼び出す場合は，Coordinator インターフェースの register_synchronization() メソッドを使用して，Synchronization オブジェクトを登録する必要があります。Synchronization オブジェクトの登録は，トランザクションオブジェクトまたは回復可能なサーバーから行います。詳細については，「Coordinator インターフェース」の「register_synchronization()」を参照してく

ださい。登録は，トランザクションが作成された後で，2 フェーズコミットプロトコルが開始される前に，行う必要があります。

複数の Synchronization オブジェクトを作成し，それらを 1 つのトランザクションに登録することができます。

VisiTransact Transaction Service は，トランザクション作業が完了した後で，2 フェーズコミットプロトコルが開始される前に（参加しているリソースの prepare() が呼び出される前に），このメソッドを呼び出します。完了処理の開始時にトランザクションがまだコミットを継続している場合にだけ，VisiBroker VisiTransact は before_completion() を呼び出します。つまり，Terminator->commit() が呼び出され，トランザクションにロールバックのマークが付けられていない場合です。Terminator->rollback() が呼び出された場合，つまり最初の Synchronization オブジェクトによってトランザクションにロールバックのマークが付けられた場合，またはトランザクションにロールバックのマークがすでに付けられている場合，このトランザクションでは before_completion() は呼び出されません。

トランザクションを確実にロールバックするには，このメソッド内で rollback_only() メソッドを呼び出します。get_status() を呼び出して，トランザクションにロールバックのマークが付けられているかどうかを確認することもできます。ただし，メソッドが呼び出された時点では，その状態に基づいてトランザクションが実際にコミットされるかどうかを示すことはできません。

the Synchronization インターフェースは TransactionalObject を継承するため，Current オブジェクトを介してトランザクションコンテキストを使用できます。つまり，before_completion() は，get_status() や get_control() など，Current オブジェクトのすべてのメソッドを使用できます。

CosTransactions.idl の Synchronization インターフェースに含まれています。

Synchronization オブジェクトによって CORBA の例外が生成されると，トランザクションがロールバックされます。

関連するメソッド：

- after_completion()
- Current インターフェースの get_status()
- Terminator インターフェースの commit()
- Current インターフェースの register_synchronization()
- Coordinator インターフェースの register_synchronization()
- Current インターフェースの rollback_only()
- Coordinator インターフェースの rollback_only()

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

VISTransactionService クラス

VISTransactionService クラスは，VisiTransact Transaction Service のインスタンスをアプリケーションプロセスにリンクするために用意されています。

次のメソッドが **visits.h** ファイルに定義されています。

- init()
- terminate()

次の節で，このメソッドについて説明します。

VISTransactionService メソッド

init()

```
static void init(int &argc, char* const* argv);
```

このメソッドは，アプリケーションプロセスにリンクされている VisiTransact Transaction Service のインスタンスをすべて初期化します。**ots_r** ライブラリと **otsinit** オブジェクトファイルをリンク行に追加してプロセスにリンクした VisiTransact Transaction Service のインスタンスをアクティブ化するには，このメソッドを呼び出す必要があります。

VisiTransact Transaction Service の埋め込みインスタンスを初期化するには，init() メソッドを呼び出す必要があります。ORB_init() が呼び出された後で，認識されたすべての VisiTransact 引数は，ユーザーのクライアントプログラムに必要な他の引数の処理を妨げないように，元のパラメータリストから削除されます。

注意 terminate(), **vshutdown**, またはコンソールを使用して, インプロセスの VisiTransact Transaction Service インスタンスを非アクティブ化した場合は，init() を再度呼び出さないでください。

visits.h の VISTransactionService インターフェースに含まれています。

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
argc	init() メソッドに渡される引数の数。
argv	init() メソッドに渡される実際の引数。

メモ argc パラメータと argv パラメータは，main 関数の argc と argv から取得されます。

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

terminate()

```
static void terminate();
```

このメソッドは, init() を呼び出して初期化された VisiTransact Transaction Service のインスタンスをすべてクリーンアップします。コマンドラインオプション OTSexit_on_shutdown が 1 に設定されていない限り，このメソッドによってアプリケーションプロセスが停止することはありません。

このオプションが設定されていないか，0 に設定されている場合は，スマートエージェントに登録されている VisiTransact Transaction Service オブジェクトが非アクティブ化されますが，アプリケーションプロセスが停止することはありません。

visits.h の VISTransactionService インターフェースに含まれています。

詳細については，『VisiBroker VisiTransact ガイド』を参照してください。

VISSessionManager module

ここでは，VISSessionManager モジュールを紹介し，そのクラス，データ型，構造体，およびメソッドについて説明します。

モジュールの概要

セッションマネージャは，事前設定されたデータベース接続を取得するためにアプリケーションによって使用されるコンポーネントです。セッションマネージャは，アプリケーションをデータベース固有の要件（接続処理，スレッド管理など）から分離します。セッショ

ンマネージャを使って接続が取得されると，VisiTransact トランザクションサービスによってトランザクションが自動的に調整されます。アプリケーション開発者は，トランザクションにデータベースの関与を組み込むコードを記述する必要はありません。アプリケーションコードでは，データベース内の必要なデータにアクセスするための操作を処理するだけで済みます。

セッションマネージャとそれに関連付けられているリソースは，DBMS への完全なトランザクションアクセスを提供します。セッションマネージャの XA 実装と，そのリソース実装（XA リソースディレクタ）により，完全な 2 フェーズコミット機能がサポートされます。分散トランザクションの場合，セッションマネージャは，VisiTransact トランザクションサービスとの組み合わせで，XA インターフェース呼び出しを実行して，データベースに対するアプリケーションの作業を組み込みます。

一方，セッションマネージャの DirectConnect バージョンは，統合化リソースを使用して，最適化されたトランザクションアクセスを提供します。ただし，プログラミングモデルの制約は大きくなります。

アプリケーションでは，セッションマネージャモジュールにある次のインターフェースを使用できます。

- `Connection`— トランザクションをサポートするデータベース接続を表します。
- `ConnectionPool`— クライアントに接続を割り当てるファクトリインターフェースです。

Connection オブジェクトや ConnectionPool オブジェクトはプロセス内でローカルに使用できる必要があるため，これらのインターフェースは擬似 IDL です。真の IDL ではありません。ConnectionPool へのアクセスは，`resolve_initial_references()` 呼び出しを使って取得できます。

現在のところ，C++ セッションマネージャインターフェースだけを使用できます。

次のサンプルコードは，VISSessionManager モジュールの IDL です。

```
#ifndef _vissessionmanager_idl_
#define _vissessionmanager_idl_
#include <CosTransactions.idl>
#pragma prefix "visigenic.com"

module VISSessionManager
{
  struct Attribute
  {
    string name;
    string value;
  };
  typedef sequence<Attribute> Attributes;

  struct ErrorInfo
  {
    string reason;
    string subsystem;
    unsigned long code;
  };
  typedef sequence<ErrorInfo> ErrorInfos;



  exception Error
  {
    ErrorInfos info;
  };

  interface Connection
  {
    enum ReleaseType
```

```
    {
      MarkSuccess,
      MarkForRollback
    };

    typedef unsigned long long NativeConnectionHandle;

    NativeConnectionHandle getNativeConnectionHandle()
      raises(Error);

    Attributes getAttributes()
      raises(Error);

    string getInfo(in string info_type)
      raises(Error);

    boolean isSupported(in string support_type)
      raises(Error);

    void hold(in unsigned long timeout)
      raises(Error);

    void resume()
      raises(Error);

    void release(in ReleaseType type)
      raises(Error);
    void releaseAndDisconnect()
      raises(Error);
  };  // Connection

  interface ConnectionPool
  {
    exception ProfileError
    {
      string reason;
      unsigned long code;
    };

    Connection getConnection(in string profile_name)
      raises(ProfileError, Error);

    Connection getConnectionWithCoordinator(in string profile_name,
              in CosTransactions::Coordinator coord)
      raises(ProfileError, Error);

    Attributes getProfileAttributes(in string profile_name)
      raises(ProfileError);

  }; // ConnectionPool インターフェース
};  // module VISSessionManager

#pragma prefix ""
#endif // _vissessionmanager_idl_
```

構造体

VISSessionManager モジュールは，次の構造体をデータ型として定義します。

- ErrorInfo— この構造体は例外で使用されます。
- Attribute— この構造体は接続属性を記述します。属性値は，常に Attribute 構造体内の文字列として表現されます。

次のサンプルコードは，VISSessionManager の ErrorInfo 構造体です。

```
struct ErrorInfo
{
  string reason;
  string subsystem;
  unsigned long code;
};
```

次の表に，ErrorInfo 構造体のメンバーの定義を示します。.

表 10.4　ErrorInfo 構造体のメンバー

メンバー	説明
reason	エラーの説明を含む文字列。
subsystem	エラーを生成した基本コンポーネント。
code	エラーコード番号。

現在のところ，セッションマネージャに関するエラーを生成するサブシステムは次のとおりです。

- "Session Manager"— セッションマネージャモジュールの本体。
- "XA Native"— データベースに対する XA 呼び出しで発生したエラー。
- "Oracle"— Oracle APIs に対する直接呼び出しで発生したエラー。

次のサンプルコードは，VISSessionManager の Attribute 構造体です。

```
struct Attribute
{
  string name;
  string value;
};
```

T 次の表に，Attribute 構造体のメンバーの定義を示します。.

表 10.5　Attribute 構造体のメンバー

メンバー	説明
name	属性の型を示す文字列。
value	名前付きの属性の値。

次の属性は，すべてのリソースマネージャに対して存在し，すべての接続プロファイルで指定されます。.

表 10.6　接続プロファイルの属性

属性	デフォルトの設定	説明
database_name	なし	データベースの名前を表す文字列。
userid	なし	接続の取得に使用されるユーザー ID を表す文字列。
password	なし	データベースのパスワードを表す文字列。

例外

セッションマネージャのメソッドが例外を受け取る場合は，通常，この例外です。ただし，標準の CORBA 例外を受け取る可能性もあります。

次のサンプルコードは，VISSessionManager モジュールの Error 例外です。

```
exception Error
{
  ErrorInfos info;
};
```

ErrorInfos は，アプリケーションに戻される（エラースタックを表す）構造体のシーケンスです。アプリケーションでは，シーケンス内にあるエラーの数を問い合わせたり，エラーが発生した層を調べることができます。さらに，エラーの情報に 1 つずつアクセスすることができます。この構造体は，次の情報を提供します。

- エラーの理由
- エラーが発生したサブシステム（モジュール）
- この特定のエラーのエラーコード

次のサンプルコードは，エラーを反復処理する例を示します。

```
⋮
try
{
  // プールにデータベース接続を要求します。
  // データベースプロファイル "quickstart" を使用します。
  conn = _pool->getConnection("quickstart");

  // ネイティブの OCI 呼び出しに使用する接続ハンドルを取得します。
  lda_ptr = (Lda_Def*) conn->getNativeConnectionHandle();
}
catch(const VISSessionManager::Error& ex)
{
  cerr << "Session Manager error:\n";
  // すべてのエラーメッセージを出力します。
  for(CORBA::ULong i = 0; i < ex.info.length(); i++)
  {
    cerr << "  " << ex.info[i].subsystem
      << "-"  << ex.info[i].code
      << ": " << ex.info[i].reason
      << endl;
    }
  throw ApplicationException();
  // これは，アプリケーションで定義します。
}
⋮
```

ConnectionPool インターフェース

ConnectionPool インターフェースは，セッションマネージャのデータベース接続プールへのアクセスを提供します。管理する接続のタイプに関係なく，プロセスごとに 1 つの論理 ConnectionPool があります。ConnectionPool は，指定された接続プロファイルを使用して，接続を割り当てます。アプリケーションが接続を要求すると，ConnectionPool オブジェクトは，プール内ですでに開かれている使用可能な接続を探します。適切な接続を検出できない場合，ConnectionPool オブジェクトは新しい接続を開きます。

ConnectionPool インターフェースには，データベース接続を割り当てなくても，アプリケーションが設定プロファイルの属性を調べることができる方法が用意されています。アプリケーションが getProfileAttributes() を使って接続プールに照会すると，プロファイルの属性が戻されます。

選択したインターフェースに対応する C++ ヘッダーファイルをインクルードする必要があります。VISSessionManager_c.hh は，**VISSessionManager.idl** から生成されたクライアントヘッダーファイルです。

メモ 独自のクライアントスタブヘッダーファイルを生成しないでください。セッションマネージャオブジェクトライブラリとの互換性を維持するため，提供されたクライアントヘッダーファイルを使用する必要があります。

ConnectionPool オブジェクトリファレンスの取得

次に，ConnectionPool オブジェクトへのリファレンスを取得するための一般的な手順とサンプルコードを示します。

- ORB resolve_initial_references() メソッドを呼び出し，VISSessionManager::ConnectionPool. オブジェクト型を渡します。
- 戻されたオブジェクトを VISSessionManager::ConnectionPool にナローイングします。

次のサンプルコードは，ConnectionPool オブジェクトリファレンスを取得する C++ の例です。

```
⋮
{
    CORBA::ORB_var orb = CORBA::ORB_init();
    CORBA::Object_var object =
      orb->resolve_initial_references("VISSessionManager::ConnectionPool");
    VISSessionManager::ConnectionPool_var pool =
      VISSessionManager::ConnectionPool::_narrow(object);
    ⋮
}
```

ConnectionPool オブジェクトリファレンスの使用

ConnectionPool オブジェクトリファレンスは，それを作成したプロセス全体で有効です。つまり，任意のスレッドで使用できます。ConnectionPool オブジェクトへのリファレンスを取得するために何度も呼び出しを実行できます。または，プロセス全体でリファレンスを 1 つだけ使用し，膨大な resolve_initial_references() 呼び出しのオーバーヘッドを削減することもできます。

例外

プロファイルを操作しようとしてエラーが発生すると，ConnectionPool が ConnectionPool::ProfileError 例外を生成します。

次のサンプルコードは，ConnectionPool インターフェースの ProfileError 例外を示します。

```
exception ProfileError
{
  string reason;
  unsigned long code
};
```

Error の詳細は，188 ページの「例外」を参照してください。

メソッド

ConnectionPool インターフェースのメソッドは，次のとおりです。

- getConnection()
- getConnectionWithCoordinator()
- getProfileAttributes()

次に，これらのメソッドについて説明します。

getConnection()

このメソッドは VisiBroker 固有です。データベース接続を割り当て，接続とトランザクションを透過的に関連付け，アプリケーションに Connection オブジェクトを戻します。

スレッドが getConnection() を呼び出すには，アクティブなトランザクションコンテキストを保持している必要があります。getConnectionWithCoordinator() メソッドを使用して，明示的に指定されたトランザクションのための接続を取得できます。

このメソッドが Error 例外を生成する場合は，「エラーコード」の情報を参照してください。

インターフェース

VISSessionManager.idl の ConnectionPool

シグニチャ

```
Connection getConnection(in string profile_name)
raises(VISSessionManager::ConnectionPool::ProfileError,
      VISSessionManager::Error);
```

パラメータ

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
profile_name	要求された接続の属性を記述するプロファイルの名前。

例

次のサンプルコードは，getConnection() メソッドの使用例です。

```
⋮
VISSessionManager::ConnectionPool_var pool;
// Ask the pool for a database connection
VISSessionManager::Connection_var conn = pool->getConnection("quickstart");
⋮
```

接続プールの詳細は，「セッションマネージャを使用したデータアクセス」を参照してください。

参照

- 189 ページの「ConnectionPool インターフェース」

getConnectionWithCoordinator()

このメソッドは VisiBroker 固有です。明示的に指定されたトランザクションコーディネータを使って接続を割り当て，アプリケーションに Connection オブジェクトを戻します。接続は透過的にトランザクションに関連付けられます。このメソッドが Error 例外を生成する場合は，「エラーコード」の情報を参照してください。

暗黙的なトランザクションコンテキストの接続を割り当てるには，getConnection() を使用します。

インターフェース

VISSessionManager.idl の ConnectionPool

シグニチャ

```
Connection getConnectionWithCoordinator(in string profile_name,
                        in CosTransactions::Coordinator coord)
raises(VISSessionManager::ConnectionPool::ProfileError,
      VISSessionManager::Error);
```

パラメータ

このメソッドでは，次のパラメータが使用されます。.

パラメータ	説明
profile_name	取得する接続の接続名。
coord	この接続に関連付けられるトランザクションを表す Coordinator オブジェクトリファレンス。

例

次のサンプルコードは，getConnectionWithCoordinator() メソッドの使用例です。

```
⋮
VISSessionManager::ConnectionPool_var pool;
// プールにデータベース接続を要求します。
// データベースプロファイル "quickstart" を使用します。
conn = pool->getConnectionWithCoordinator("quickstart", coordinator);
⋮
```

接続プールの詳細は，「セッションマネージャを使用したデータアクセス」を参照してください。

getProfileAttributes()

このメソッドは VisiBroker 固有です。このメソッドを使用すると，接続を割り当てずに，プロファイルの属性を照会できます。アプリケーションは，このメソッドで接続プロファイルの名前を渡し，ConnectionPool オブジェクトは，接続プロファイルの属性をアプリケーションに戻します。

メモ getAttributes() を使用して，現在開いている接続の属性を参照することもできます。

セッションマネージャは，このメソッド呼び出しに応答して，Attributes オブジェクトのリファレンスを戻します。アプリケーションでは，次の型のオブジェクトを使って Attributes オブジェクトを保持し，適切なメモリ管理を行う必要があります。

```
VISSessionManager::Attributes_var
```

インターフェース

VISSessionManager.idl の ConnectionPool

シグニチャ

```
Attributes getProfileAttributes(in string profile_name)
raises(VISSessionManager::ConnectionPool::ProfileError);
```

接続プロファイルの属性については，188 ページの「接続プロファイルの属性」の表を参照してください。

パラメータ

このインターフェースでは，次のパラメータが使用されます。

パラメータ	説明
profile_name	profile_name 属性値が戻される接続プロファイルの名前。

例

次のサンプルコードは，getProfileAttributes() の使用例です。

```
⋮
VISSessionManager::ConnectionPool pool;
VISSessionManager::Attributes_var attrs;
attrs = pool->getProfileAttributes("quickstart");
CORBA::ULong len = attrs->length();
for (CORBA::ULong i=0; i<len; i++)
{
  cout << "Attribute " << i << ": " << attrs[i].name
      << " = " << attrs[i].value << endl;
}
⋮
```

詳細については，「セッションマネージャを使用したデータアクセス」と 189 ページの「ConnectionPool インターフェース」を参照してください。

Connection インターフェース

Connection インターフェースは，トランザクションが設定されたデータベース接続へのアクセスをアプリケーションに提供します。

Connection オブジェクトを直接作成するかわりに，getConnection() や getConnectionWithCoordinator() を使用して，ConnectionPool から Connection オブジェクトのリファレンスを取得します。ConnectionPool がこれらのオブジェクトの 1 つを戻した場合，データベース接続の割り当てとトランザクションとの関連付けは完了しています。

release() メソッドまたは releaseAndDisconnect() メソッドを一度呼び出した後で，その接続のオペレーションを実行すると，すべて例外になります。

Connection インターフェースは，ネイティブデータベースハンドルを取得するメソッドのほか，接続に関する情報をプログラムで参照するためのメソッドもいくつか提供します。

選択したインターフェースに対応する C++ ヘッダーファイルをインクルードする必要があります。VISSessionManager_c.hh は，**VISSessionManager.idl** から生成されたクライアントヘッダーファイルです。

データ型

Connection インターフェースは，ReleaseType データ型を定義します。ReleaseType の詳細は，197 ページの「release()」を参照してください。

メソッド

Connection インターフェースのメソッドは，次のとおりです。

- getAttributes()
- getInfo()
- getNativeConnectionHandle()
- hold()
- isSupported()
- release()
- releaseAndDisconnect()
- resume()

次に，これらのメソッドについて説明します。

getAttributes()

このメソッドは VisiBroker 固有です。このメソッドは，現在割り当てられている接続の設定プロファイルの属性の値を戻します。セッションマネージャは，Attributes オブジェクトを割り当て，そのオブジェクトへのリファレンスを戻します。アプリケーションでは，次の型のオブジェクトを使って Attributes オブジェクトを保持し，適切なメモリ管理を行う必要があります。

```
VISSessionManager::Attributes_var
```

インターフェース

VISSessionManager.idl の Connection

シグニチャ

```
Attributes getAttributes()
raises (VISSessionManager::Error);
```

接続プロファイルの属性については，188 ページの「接続プロファイルの属性」の表を参照してください。

パラメータ

なし。

例

次のサンプルコードは，getAttributes() の使用例です。

```
⋮
VISSessionManager::Connection_var conn;
VISSessionManager::Attributes_var attrs;
attrs = conn->getAttributes();
CORBA::ULong len = attrs->length();
for (CORBA::ULong i=0; i<len; i++)
{
  cout << "Attribute " << i << ": " << attrs[i].name
      << " = " << attrs[i].value << endl;
}
⋮
```

詳細については,「セッションマネージャを使用したデータアクセス」を参照してください。

getInfo()

このメソッドは VisiBroker 固有です。特定のリソースマネージャに対して，セッションマネージャ実装の特性を照会します。たとえば，セッションマネージャのバージョンを照会できます。アプリケーションでは，次の型のオブジェクトを使って戻された値を保持し，適切なメモリ管理を行う必要があります。

```
CORBA::String_var
```

インターフェース

VISSessionManager.idl の Connection

シグニチャ

```
string getInfo(in string info_type)
raises (VISSessionManager::Error);
```

パラメータ

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
info_type	The information type to be returned.

特定のタイプのセッションマネージャの情報タイプについては，VisiBroker Integrated Transaction Service の『*Data Access Guide*』を参照してください。

次の情報タイプは，すべてのタイプのセッションマネージャで使用できます。

- "version"— 汎用セッションマネージャのバージョン番号を戻します。バージョン番号は，VisiBroker ユーティリティ vbver の標準の 5 フィールド文字列として戻されます。これは，情報提供の目的で使用されます。
- "version_rm"—セッションマネージャのリソースマネージャ専用コンポーネントのバージョン番号を戻します。これは，情報提供の目的で使用されます。

次のサンプルコードは，getInfo() の使用例です。

```
⋮
VISSessionManager::Connection_var conn;
CORBA::String_var info = conn->getInfo("version");
⋮
```

詳細については,「セッションマネージャを使用したデータアクセス」を参照してください。

getNativeConnectionHandle()

このメソッドは VisiBroker 固有です。この Connection オブジェクトによって表されるデータベース接続のリソースマネージャ（通常はデータベース）のネイティブ接続ハンドルを戻します。アプリケーションは，このネイティブハンドルを使用して，リソースマネー

ジャのネイティブ API を呼び出します。アプリケーションは，このハンドルのリソースマネージャ API 呼び出しを使って接続を解除するのではなく，release() メソッドまたは releaseAndDisconnect() メソッドを使って接続を解放する必要があります。

これは，オーバーヘッドの小さなオペレーションなので，ネイティブ接続ハンドルを取得するために必要に応じて何度も呼び出すことができます。

メモ　このメソッドは，接続を割り当てません。詳細については，190 ページの「getConnection()」または 191 ページの「getConnectionWithCoordinator()」を参照してください。

インターフェース

VISSessionManager.idl の Connection

シグニチャ

```
NativeConnectionHandle getNativeConnectionHandle()
raises(VISSessionManager::Error);
```

パラメータ

なし。

例

次のサンプルコードは，getNativeConnectionHandle() メソッドの使用例です。

```
⋮
VISSessionManager::Connection_var conn;
//get a connection handle
lda = (Lda_Def *) conn->getNativeConnectionHandle();
⋮
```

接続プールの詳細は，「セッションマネージャを使用したデータアクセス」を参照してください。

hold()

注意　hold() を使用すると接続が独占されるため，パフォーマンスに影響します。hold() は，必要なときだけに使用してください。

このメソッドは VisiBroker 固有です。制御スレッドが現在のプロセスから離れて戻ろうとしていることをセッションマネージャに通知します。アプリケーションは，resume() が呼び出されるまでこの接続ハンドルを使用しないことを保証します。

セッションマネージャは，この接続に関してアクティブなスレッドが現在のプロセス内にない場合に通知を受けることを要求します。この要件の主な理由は，要求元がエラーなどの理由でこのプロセスに戻ってリソースを解放できない場合に，この接続に使用されているすべてのリソース（データベースロック，リソースなど）をセッションマネージャがクリーンアップする必要があるということです。セッションマネージャは，アプリケーションがまだアクティブに接続を使用しているかどうかがわからない限り，トランザクションの関連付けを解除してクリーンアップを続行することができません。

timeout パラメータは，接続がタイムアウトになり，リソースをクリーンアップするまでセッションマネージャが待機する時間を秒単位で示します。クリーンアッププロセスの一環として，接続は ConnectionPool に戻され，トランザクションにはロールバックのマークが付けられます。

アプリケーションは，resume() 呼び出しを入れずに，複数の hold() 要求を送信できます。hold() が 2 度呼ばれると，呼び出しごとにタイマーが新しい値にリセットされます。たとえば，8:42:30 に hold(60) を呼び出すと，8:43:30 にタイムアウトになります。ただし，続けて 8:42:50 に hold(45) を呼び出すと，2 度めの hold() 呼び出しによってタイマーがリセットされ，8:43:35 にタイムアウトになります。

メモ　このメソッドをサポートしていないデータベースセッションマネージャ実装もあります。アプリケーションは，isSupported() を使用して，セッションマネージャが hold() メソッドをサポートしているかどうかを照会できます。詳細については，VisiBroker Integrated Transaction Service の『*Data Access Guide*』を参照してください。

Connection オブジェクト，または対応するデータベース接続ハンドルをもう一度使用するには，事前に Connection オブジェクトの resume() を呼び出す必要があります。

インターフェース

VISSessionManager.idl の Connection

シグニチャ

```
void hold(in unsigned long timeout)
raises(VISSessionManager::Error);
```

パラメータ

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
timeout	セッションマネージャが接続を解放してプールに戻し，トランザクションにロールバックのマークを付けるまで待機する秒数。0 を渡すと，例外が発生します。

例

```
⋮
VISSessionManager::Connection_var conn;
conn->hold(60);
⋮
```

詳細については，「セッションマネージャを使用したデータアクセス」と 189 ページの「ConnectionPool インターフェース」を参照してください。

isSupported()

このメソッドは VisiBroker 固有です。特定のリソースマネージャに対して，セッションマネージャ実装のサポートされているタイプを照会します。たとえば，アプリケーションは，セッションマネージャが hold() メソッドをサポートしているかどうかを照会できます。

インターフェース

VISSessionManager.idl の Connection

シグニチャ

```
boolean isSupported(in string support_type)
raises (VISSessionManager::Error);
```

パラメータ

このメソッドでは，次のパラメータが使用されます。

パラメータ	説明
support_type	戻されるサポートタイプ。

次のサポートタイプは，すべてのタイプのセッションマネージャで使用できます。

- "hold"— hold() メソッドがサポートされている場合は true，そうでない場合は false を戻します。
- "thread_portable"—接続が作成元のスレッド以外のスレッドでも使用される場合は true，そうでない場合は false を戻します。

```
⋮
VISSessionManager::Connection_var conn;
CORBA::Boolean isPortable = conn->isSupported("thread-portable");
⋮
```

詳細については，「セッションマネージャを使用したデータアクセス」と 189 ページの「ConnectionPool インターフェース」を参照してください。

release()

このメソッドは VisiBroker 固有です。接続を解放して ConnectionPool に戻し，トランザクションにコミットまたはロールバックのマークを付けます。このメソッドを呼び出さない場合は，Connection オブジェクトが破棄されたときに，トランザクションにロールバックのマークが付けられます。

アプリケーションが release() を呼び出すと，接続の状態は消去されます。その後で Connection を呼び出すと，CORBA::BAD_OPERATION が生成されます。

後で再度接続を取得すると，同じトランザクションでさらに作業を実行できます。

メモ release() を呼び出しても，CORBA _release() メソッドのように Connection オブジェクトが解放されるわけではありません。このメソッドは，アプリケーションにとって基底のデータベース接続が必要なくなったことを ConnectionPool に示します。アプリケーションは，さらに Connection オブジェクトを解放する必要があります。それには，Connection_var オブジェクトで Connection オブジェクトを保持すると簡単です。

インターフェース

VISSessionManager.idl の Connection

シグニチャ

```
void release(in ReleaseType type)
raises(VISSessionManager::Error);
```

ReleaseType

ReleaseType の定義は次のとおりです。

```
enum ReleaseType
{
  MarkSuccess,
  MarkForRollback
};
```

ReleaseType 値については，次のパラメータの表を参照してください。

パラメータ

このインターフェースの release() および ReleaseType の値には，次のパラメータが使用されます。

パラメータ	説明
type	このトランザクション部分が成功したかどうかを示すフラグ。MarkSuccess が渡された場合，トランザクションは通常どおり進行します。そうでなく，MarkForRollback が渡された場合，トランザクションにはロールバックのマークが付けられます。
MarkSuccess	このトランザクション部分は成功しました。トランザクションは通常どおり進行します。
MarkForRollback	トランザクションにロールバックのマークが付けられます。ロールバックは VisiTransact Transaction Service によって行われるため，release(MarkForRollback) を呼び出してから実際にロールバックが行われるまで，少し遅延があります。

例

次のサンプルコードは，release() メソッドの使用例です。

```
⋮
VISSessionManager::Connection_var conn;
conn->release(VISSessionManager::Connection::MarkSuccess);
⋮
```

詳細については,「セッションマネージャを使用したデータアクセス」を参照してください。

releaseAndDisconnect()

このメソッドは VisiBroker 固有です。データベース接続を完全に閉じ，トランザクションにロールバックのマークを付けます。接続は ConnectionPool に戻されず，再利用されま

せん。このメソッドは，アプリケーションが接続になんらかの問題を検出した場合に，その接続が再利用されないようにする目的で使用されます。

アプリケーションが releaseAndDisconnect() を呼び出すと，接続の状態は消去されます。その後で Connection を呼び出すと，CORBA::BAD_OPERATION が生成されます。

インターフェース

VISSessionManager.idl の Connection

シグニチャ

```
void releaseAndDisconnect()
raises(VISSessionManager::Error);
```

パラメータ

なし。

例

releaseConnection() メソッドの使用例を次に示します。

```
⋮
VISSessionManager::Connection_var conn;
cerr << "Profile error: " << ex.code << ex.reason << endl;
conn->releaseAndDisconnect();
return balance;
⋮
```

詳細については，「セッションマネージャを使用したデータアクセス」と 189 ページの「ConnectionPool インターフェース」を参照してください。

resume()

このメソッドは VisiBroker 固有です。このメソッドは，hold() の後で使用され，この Connection の制御スレッドがプロセスに戻ったことをセッションマネージャに示します。これは，hold() に関連付けられたタイムアウトをキャンセルし，セッションマネージャが基底の接続を変更してアクティブなアプリケーションと競合しないようにします。Connection が保留状態にないときに resume() を呼び出すと，Error 例外になりますが，トランザクションや接続の状態は変更されません。

メモ hold() の呼び出しと resume() の呼び出しの間に，アプリケーションが Connection オブジェクトまたは関連するネイティブデータベースハンドルに基づいて他の呼び出しを行うことはできません。この間に hold() 呼び出しがタイムアウトになった場合は，セッションマネージャが接続を解放し，トランザクションにロールバックのマークを付ける権利を持ちます。これにより，クライアントが停止したり，再度呼び出しを行わなくなっても，そのトランザクションによってアプリケーションサーバーに保持されているリソースが永久に残ることはなくなります。

インターフェース

VISSessionManager.idl の Connection

シグニチャ

```
void resume()
raises(VISSessionManager::Error)
```

パラメータ

なし。

例

resume() メソッドの使用例を次に示します。

```
⋮
VISSessionManager::Connection_var conn;
conn->resume();
⋮
```

詳細については,「セッションマネージャを使用したデータアクセス」を参照してください。

The ITSDataConnection クラス

プラグイン可能リソースインターフェースは，Borland VisiTransact によって管理されるトランザクション内でトランザクション型アプリケーションがデータベースを永続的ストレージとして使用できるようにする一連の事前定義インターフェースを実装するコンポーネントです。詳細については，「VisiTransact 向けプラグイン可能データベースリソースモジュール」を参照してください。

このクラスは，次のように定義されます。

```
class ITSDataConnection
{
  public:
  virtual void connect() = 0;
  virtual void disconnect() = 0;
  virtual void rollback() = 0;
  virtual void commit() = 0;
  virtual xa_switch_t* xa_switch() { return 0; }
  virtual const char* xa_open_string() { return 0; }
  virtual const char* xa_close_string() { return 0; }
  virtual void* native_handle() { return 0; }
};
```

ITSDataConnection クラスのメソッドは，次の 3 つのグループに分けられます。

- ネイティブハンドル取得インターフェース
- ローカルトランザクション接続／完了インターフェース
- グローバルトランザクション接続／完了インターフェース

ネイティブハンドル取得インターフェース

```
void* native_handle();
```

この関数は，モジュールによってサポートされる，データベースのネイティブ API にアクセスするために使用されます。戻り値は void ポインタです。実装では，データベース内のデータの操作に必要な任意の型を戻すことができます。トランザクション型アプリケーションは，getNativeConnectionHandle() を介してこのポインタを取得します。このメソッド内で，セッションマネージャの接続マネージャは，native_handle() を呼び出し，そのポインタをアプリケーションに戻します。

すべてのプラグイン可能モジュールは，この関数を実装する必要があります。

ローカルトランザクション接続／完了インターフェース

ローカルトランザクションをサポートするプラグイン可能モジュールは，これらの関数を実装する必要があります。

次の 4 つのメソッドは，セッションマネージャの接続マネージャによって使用され，ローカルトランザクションの開始と完了をデータベースに通知します。

void connect();

データベースとの接続を確立し，ローカルトランザクションが開始されたことをデータベースに通知します。

void disconnect();

接続が確立されている場合は，それが不要になったことを示します。したがって，接続を閉じることができます。

void rollback();
トランザクションをコミットするように，データベースに通知します。

void commit();
トランザクションをロールバックするように，データベースに通知します。

グローバルトランザクション接続／完了インターフェース

グローバルトランザクションをサポートするプラグイン可能モジュールは，これらの関数を実装する必要があります。

セッションマネージャは，X-Open の XA インターフェースを使用して，XA 準拠データベースとやり取りします。

xa_switch_t* xa_switch();
セッションマネージャの接続マネージャがプラグイン可能モジュールに要求するものは，xa.h で定義されているすべての XA API を含む xa_switch_t データ構造体へのポインタだけです。xa_switch() 関数は，この目的で使用されます。この関数は，呼び出されるたびに，このデータへの有効なポインタを戻す必要があります。

通常，各データベースは xa_switch_t を実装し，それをクライアントに公開します。このデータ構造体の名前は，データベースによって異なります。たとえば，Oracle9i は，xa_switch_t を xaosw という名前のグローバル変数として実装しています。

この関数は，セッションマネージャの接続マネージャにより，接続のタイプを識別するためにも使用されます。この関数が 0 を戻した場合，セッションマネージャは，接続を DC タイプとして処理します。そうでない場合は，XA タイプとして処理します。

グローバルトランザクションをサポートするプラグイン可能モジュールは，この関数を実装し，0 以外を戻す必要があります。

const char* xa_open_string();
この関数は，`xa_open()` 呼び出しの引数として使用される文字列を戻します。

const char* xa_close_string();
この関数は，`xa_close()` 呼び出しの引数として使用される文字列を戻します。

この 2 つのメソッドは，データベースへの XA 接続を開くまたは閉じるためのデータベース固有のパラメータを取得するために，セッションマネージャによって呼び出されます。xa_open_string() から戻された文字列は xa_open() で使用され，xa_close_string() から戻された文字列は xa_close() で使用されます。

これらの関数が 1 つの XA 接続に対して呼び出されると，セッションマネージャは，それらの戻り値を後で使用できるように保存します。戻されたポインタの有効性を実装で維持し続ける必要はありません。

第 11 章

ネイティブメッセージングのインターフェースとクラス

ここでは，ネイティブメッセージングに関連するインターフェースとクラスについて説明します。

RequestAgent

```
class NativeMessaging::RequestAgent : public virtual CORBA_Object
```

リクエストエージェントインターフェースは，ネイティブメッセージングリクエストエージェントのオペレーションを定義します。

インクルードファイル

このクラスを使用する場合は，NativeMessaging_c.hh ファイルをインクルードする必要があります。

IDL 定義

```
module NativeMessaging {
  interface RequestAgent {
        exception DuplicatedRequestTag {};
        exception PollingGroupIsEmpty {};
        exception RequestNotExist {};

        Request create_request(
                in RequestDesc  desc) raises (DuplicatedRequestTag);
        RequestTagSeq poll(
                in string               polling_group,
                in unsigned long        timeout,
                in boolean              unmask) raises
(PollingGroupIsEmpty);
        void destroy_request(
                in Request      req) raises (RequestNotExist);
```

```
  };
};
```

RequestAgent のメソッド

create_request

```
virtual ::CORBA::Object_ptr create_request(const NativeMessaging::RequestDesc& _desc);
```

このメソッドは，リクエストエージェントで非同期メソッド呼び出し要求オブジェクトを作成して返します。

パラメータ	説明
_desc	ターゲットオブジェクトと非同期要求に関する情報を含む RequestDesc 構造体。

このメソッドは，DuplicatedRequestTag 例外を生成します。

poll

```
virtual NativeMessaging::RequestTagSeq* poll(const char* _polling_group,
    ::CORBA::Ulong _timeout,
    ::CORBA::Boolean _umask);
```

このメソッドは，応答の準備が完了したリクエストタグのシーケンスを返します。

パラメータ	説明
_polling_group	ポーリンググループの名前。
_timeout	ポーリンググループに有効な応答がない場合にタイムアウトになるまで待機する時間（ミリ秒単位）。値の意味は次のとおりです。 • timeout > 0 のポーリングは，指定された時間ブロックします。タイムアウトになった時点で応答がない場合は，空のリクエストタグシーケンスが返されます。 • timeout = 0 のポーリングは，ブロックしません。応答がある場合は，それらのタグが呼び出し側に返されます。応答がない場合は，空のシーケンスが返されます。 • timeout < 0（または timeout = 2^(32-1)）のポーリングは，応答があるまでブロックします。
_unmask	このフラグが false の場合は，同じポーリンググループに対して poll を呼び出すと，前回のポーリングで返されたリクエストタグを手動または自動で破棄していない限り，同じリクエストタグが返されます。このフラグが true の場合，一度ポーリングで返されたリクエストタグは，その後のポーリングで返されなくなります。

このメソッドは，PollingGroupIsEmpty 例外を生成します。

destroy_request

```
virtual void destroy_request(::CORBA::Object_ptr _req);
```

このメソッドは非同期要求を破棄します。

パラメータ	説明
_req	破棄する非同期要求オブジェクトリファレンス。

このメソッドは，RequestNotExist 例外を生成します。

RequestDesc

```
struct NativeMessaging::RequestDesc;
```

非同期要求の処理に必要な情報をすべて含むデスクリプタ構造体。

インクルードファイル

この構造体を使用する場合は，NativeMessaging_c.hh ファイルをインクルードする必要があります。

IDL 定義

```
module NativeMessaging {
  typedef Object               Request;
  typedef sequence<octet>      OctetSeq;
  typedef OctetSeq             RequestTag;
  typedef sequence<RequestTag> RequestTagSeq;
  typedef OctetSeq        Cookie;

  struct RequestDesc {
         Object          target;
         string          repository_id;
         ReplyRecipient  reply_recipient;
    Cookie   the_cookie;
         string          polling_group;
         RequestTag      request_tag;
         PropertySeq     properties;
  };
};
```

RequestDesc のフィールド

フィールド	説明
target	クライアントがオペレーションを呼び出すターゲットオブジェクトのリファレンス。null 値が渡されると，CORBA::BAD_PARAM 例外が生成されます。
repository_id	ターゲットオブジェクトのリポジトリ ID です。これが空の文字列の場合，リクエストエージェントは，ターゲットオブジェクトリファレンスからリポジトリ ID を抽出しようとします。リポジトリ ID が空で，ターゲット IOR リファレンスが null または空の場合は，CORBA::BAD_PARAM 例外が生成されます。また，クライアントはリポジトリ ID として * を使用できます。これはワイルドカードとして機能し，この要求オブジェクトの is_a オペレーションは，任意のリポジトリ ID に対して true を返します。
reply_recipient	コールバックモデルを使用する場合の応答受信者（応答ハンドラ）のリファレンス。このリファレンスが null でない場合，応答の準備が完了すると，リクエストエージェントはこのリファレンスの reply_available メソッドを呼び出します。
the_cookie	ユーザーが指定したオクテットのシーケンス。reply_available が呼び出されると，reply_recipient に送信されます。Cookie 内の情報は，ユーザーが定義します。
polling_group	ユーザーが割り当てたポーリンググループ名。グループ名は，リクエストエージェントのスコープ内に限定されます。グループ名は一意には使用されません。空でない文字列がグループ名として割り当てられ，リクエストエージェントに同じ名前のポーリンググループがない場合は，その名前の新しいグループが暗黙に作成されます。ただし，グループがすでに存在する場合は，作成された要求オブジェクトがそのグループに挿入されます。

フィールド	説明
request_tag	ユーザーが割り当てます。空でない場合は，グループ内の要求を一意に識別します。グループ内の別の要求が同じタグを持つ場合，create_request メソッドは DuplicatedRequestTag exception を生成します。
properties	Property 構造体のシーケンス。現在は，1 つの Property 値だけを定義できます（「Property 構造体」を参照）。

ReplyRecipient

```
class NativeMessaging::ReplyRecipient : public virtual CORBA_Object
```

コールバック応答受信者のインターフェースを定義します。

インクルードファイル

このクラスを使用する場合は，NativeMessaging_c.hh ファイルをインクルードする必要があります。

```
module NativeMessaging {
interface ReplyRecipient {
    void reply_available(
  in Request reply_holder,
  in string  operation,
  in Cookie  the_Cookie);
  };
};
```

ReplyRecipient のメソッド

reply_available

```
virtual void reply_available(::CORBA::Object_ptr _reply_holder, const char* _operation,
const NativeMessaging::OctetSeq& _the_Cookie);
```

パラメータ	説明
_reply_holder	応答を受信する非同期要求。
_operation	クライアントによって呼び出されるオペレーション。
_the_Cookie	要求の作成時にクライアントから渡される Cookie。

REPLY_NOT_AVAILABLE

この定数は，要求の応答がない場合にポーリングを実行したクライアントに対して RequestAgent によって生成される CORBA::NO_RESPONSE 例外のマイナーコード値を定義します。

インクルードファイル

この定数を使用する場合は，NativeMessaging_c.hh ファイルをインクルードする必要があります。

IDL 定義

```
module NativeMessaging {
const unsigned long REPLY_NOT_AVAILABLE = 100;
};
```

Property

```
struct NativeMessaging::Property;
```

シンボル的なプロパティ名とその値を CORBA::Any に保持します。

インクルードファイル

この構造体を使用する場合は，NativeMessaging_c.hh ファイルをインクルードする必要があります。

IDL 定義

```
module NativeMessaging {
  struct Property {
        string  name;
        any     value;
  };
};
```

Property のフィールド

フィールド	説明
name	プロパティの名前。現在，1 つの名前（RequestManualTrash）だけが認識されます。
value	プロパティの値。RequestManualTrash は boolean 型の値を持ちます。true に設定した場合，要求は，destroy_request メソッドを呼び出して手動で破棄されます。false に設定した場合，要求は，応答が読み取られたときに自動的に破棄されます（デフォルト）。

PropertySeq

```
class NativeMessaging::PropertySeq : private VISResource
```

非同期要求の作成時に，RequestDesc 内で渡される Property のシーケンス。

インクルードファイル

このクラスを使用する場合は，NativeMessaging_c.hh ファイルをインクルードする必要があります。

OctetSeq

```
class NativeMessaging::OctetSeq : private VISResource
```

このクラスはオクテットのシーケンスを表します。CORBA::OctetSeq に似ていますが，NativeMessaging.idl を他の IDL から独立させるためにここで定義されます。

インクルードファイル

このクラスを使用する場合は，NativeMessaging_c.hh ファイルをインクルードする必要があります。

RequestTag

```
typedef OctetSeq RequestTag;
```

ポーリンググループ内の要求を識別するオクテットシーケンス。

インクルードファイル

このクラスを使用する場合は，NativeMessaging_c.hh ファイルをインクルードする必要があります。

RequestTagSeq

```
class NativeMessaging::RequestTagSeq : private VISResource
```

このクラスのインスタンスは，グループのポーリングが実行されたときに，RequestAgent の poll メソッドから返されます。シーケンス内の各要素は RequestTag です。つまり，ポーリンググループ内の要求を識別するオクテットシーケンスです。

インクルードファイル

このクラスを使用する場合は，NativeMessaging_c.hh ファイルをインクルードする必要があります。

Cookie

```
typedef OctetSeq Cookie
```

非同期要求の作成時に，RequestDesc 内で渡されるオクテットのシーケンス。Cookie 内の内容は，ユーザーが定義します。コールバックが行われると，リクエストエージェントは，この Cookie を ReplyRecipient の reply_available メソッドに渡します。

インクルードファイル

この型を使用する場合は，NativeMessaging_c.hh ファイルをインクルードする必要があります。

DuplicatedRequestTag

```
class DuplicatedRequestTag : public CORBA_UserException
```

このクラスは，指定されたポーリンググループ名で非同期要求を作成したときに，同じリクエストタグを持つ別の要求がポーリンググループにある場合に生成される UserException を定義します。

インクルードファイル

このクラスを使用する場合は，NativeMessaging_c.hh ファイルをインクルードする必要があります。

PollingGroupIsEmpty

```
class PollingGroupIsEmpty : public CORBA_UserException
```

このクラスは，次の場合に RequestAgent で poll メソッドが呼び出されたときに生成される UserException を定義します。

- 指定された名前のグループがない。
- ポーリンググループは存在するが，応答を待機している要求が含まれていない。

インクルードファイル

このクラスを使用する場合は，NativeMessaging_c.hh ファイルをインクルードする必要があります。

RequestNotExist

```
class RequestNotExist : public CORBA_UserException
```

このクラスは，RequestAgent で destroy_request メソッドが呼び出されたときに，指定した要求が見つからないか，それがすでに破棄されていた場合に生成される UserException を定義します。

インクルードファイル

このクラスを使用する場合は，NativeMessaging_c.hh ファイルをインクルードする必要があります。

第 12 章

ポータブルインターセプタのインターフェースとクラス

ここでは，OMG 仕様で定義されている*ポータブルインターセプタ*のインターフェースとクラスの BES VisiBroker によるインプリメンテーションについて説明します。これらのインターフェースとクラスの詳細については，OMG Final Adopted Specification（最終採用仕様）ptc/2001-04-03 の「Portable Interceptors」を参照してください。

メモ　これらのインターフェースを使用する方法については，『VisiBroker for C++ 開発者ガイド』の「ポータブルインターセプタの使い方」を参照してください。

インターセプタについて

VisiBroker ORB は，インターセプタと呼ばれる API のセットを提供します。インターセプタは，VisiBroker ORB にトランザクションやセキュリティのサポートなどの追加機能を組み込む手段を提供します。VisiBroker ORB の通常の実行フローを VisiBroker ORB サービスがインターセプトできるように，インターセプタは VisiBroker ORB にフックされています。次の表は，VisiBroker でサポートしているインターセプタの種類です。

ポータブルインターセプタの使用方法については，『VisiBroker for C++ 開発者ガイド』の「ポータブルインターセプタの使い方」を参照してください。

表 12.1　VisiBroker がサポートしているインターセプタの種類

インターセプタの種類	説明
ポータブルインターセプタ	ポータブルインターセプタは，インターセプタのポータブルコードを記述して，さまざまなベンダーの ORB と一緒に使用できるようにする OMG による標準化機能です。
インターセプタ	インターセプタは，VisiBroker で定義されている Borland Enterprise Server 固有のインターセプタです。

インターセプタの使用方法については，第 13 章「5.x インターセプタとオブジェクトラッパーのインターフェースとクラス」を参照してください。その他の詳細については，『VisiBroker for C++ 開発者ガイド』の「ポータブルインターセプタの使い方」を参照してください。

次の表に，2 種類のポータブルインターセプタを示します。

表 12.2　ポータブルインターセプタの種類

インターセプタの種類	説明
リクエストインターセプタ	VisiBroker ORB サービスがクライアントとサーバーの間でコンテキスト情報を転送できるようにします。リクエストインターセプタには，クライアントリクエストインターセプタとサーバーリクエストインターセプタがあります。
IOR インターセプタ	VisiBroker ORB サービスがサーバーやオブジェクトの ORB サービス関連機能について記述された情報を IOR に追加できるようにします。たとえば，SSL などのセキュリティサービスでは，タグ付きコンポーネントを IOR に追加することで，そのコンポーネントを認識するクライアントがコンポーネント内の情報に基づいてサーバーとの接続を確立できます。

ポータブルインターセプタの使用方法については，『VisiBroker for C++ 開発者ガイド』の「ポータブルインターセプタの使い方」を参照してください。

ClientRequestInfo

```
class PortableInterceptor::ClientRequestInfo : public virtual RequestInfo
```

このクラスは RequestInfo から派生され，クライアント側のインターセプトポイントに渡されます。

ClientRequestInfo のメソッドのすべてが，すべてのインターセプトポイントで有効であるとは限りません。次の表は，属性とメソッドの有効性をまとめたものです。無効な属性にアクセスしようとすると，標準マイナーコード 14 の BAD_INV_ORDER が生成されます。

表 12.3　ClientRequestInfo の有効性

	send_request	send_poll	receive_reply	receive_exception	receive_other
request_id	はい	はい	はい	はい	はい
operation	はい	はい	はい	はい	はい
arguments	はい[1]	いいえ	はい	いいえ	いいえ
exception	はい	いいえ	はい	はい	はい
contexts	はい	いいえ	はい	はい	はい
operation_context	はい	いいえ	はい	はい	はい
result	いいえ	いいえ	はい	いいえ	いいえ
response_expected	はい	はい	はい	はい	はい
sync_scope	はい	いいえ	はい	はい	はい
reply_status	いいえ	いいえ	はい	はい	はい
forward_reference	いいえ	いいえ	いいえ	いいえ	はい[2]
get_slot	はい	はい	はい	はい	はい
get_request_service_context	はい	いいえ	はい	はい	はい
get_reply_service_context	いいえ	いいえ	はい	はい	はい
target	はい	はい	はい	はい	はい
effective_target	はい	はい	はい	はい	はい
effective_profile	はい	はい	はい	はい	はい
received_exception	いいえ	いいえ	いいえ	はい	いいえ
received_exception_id	いいえ	いいえ	いいえ	はい	いいえ
get_effective_component	はい	いいえ	はい	はい	はい
get_effective_components	はい	いいえ	はい	はい	はい
get_request_policy	はい	いいえ	はい	はい	はい
add_request_service_context	はい	いいえ	いいえ	いいえ	いいえ

[1] ClientRequestInfo が send_request() に渡される場合は，in，inout，out の各引数に入力があります。ただし，有効なのは in 引数と inout 引数です。

[2] reply_status() が LOCATION_FORWARD を返さない場合は，この属性にアクセスすると標準マイナーコード 14 の BAD_INV_ORDER が生成されます。

インクルードファイル

このクラスを使用する場合は，PortableInterceptor_c.hh ファイルをインクルードする必要があります。

ClientRequestInfo のメソッド

```
virtual CORBA::Object_ptr target() = 0;
```

オペレーションを実行する際にクライアントが呼び出したオブジェクトを返します。次の effective_target() を参照してください。

```
virtual CORBA::Object_ptr effective_target() = 0;
```

オペレーションを呼び出す実際のオブジェクトを返します。reply_status() で LOCATION_FORWARD が返されると，それに続く要求で effective_target() に送信された IOR が格納されますが，target は変化しません。

```
virtual IOP::TaggedProfile* effective_profile() = 0;
```

要求を送信するためのプロファイルを IOP::TaggedProfile 形式で返します。このオペレーションのオブジェクトに対してロケーション転送が発生してオブジェクトのプロファイルが変更されると，このプロファイルはそのロケーションのプロファイルになります。

```
virtual CORBA::Any* received_exception() = 0;
```

クライアントに返す例外を含むデータを CORBA::Any 形式で返します。

例外が CORBA::Any に挿入できないユーザー例外の場合（未知の例外の場合や TypeCode が得られないバインディングの場合など），この属性は標準マイナーコード 1 のシステム例外 UNKNOWN を含む CORBA::Any になります。ただし，received_exception_id 属性に例外の RepositoryId があります。

```
virtual char* received_exception_id() = 0;
```

このメソッドはクライアントに返される received_exception の ID を返します。

```
virtual IOP::TaggedComponent* get_effective_component(CORBA::ULong _id) = 0;
```

この要求に対して選択したプロファイルから，IOP::TaggedComponent を返します。

指定コンポーネント ID が複数ある場合，このオペレーションがどのコンポーネントを返すかは確定されていません。指定のコンポーネント ID に当てはまるコンポーネントが複数ある場合，かわりに get_effective_components() が呼び出されます。

指定コンポーネント ID に対応するコンポーネントがない場合，このオペレーションでは標準マイナーコード 28 で BAD_PARAM 例外が生成されます。

パラメータ	説明
_id	返されるコンポーネントの ID。

```
virtual IOP::TaggedComponentSeq* get_effective_components(CORBA::ULong _id) = 0;
```

この要求に対して選択したプロファイルから，指定 ID のタグ付きコンポーネントをすべて返します。このシーケンスは IOP::TaggedComponentSeq の形式です。

指定コンポーネント ID に対応するコンポーネントがない場合，このオペレーションでは標準マイナーコード 28 で BAD_PARAM 例外が生成されます。

パラメータ	説明
_id	返されるコンポーネントの ID。

```
virtual CORBA::Policy_ptr get_request_policy(CORBA::ULong _type) = 0;
```

現在のオペレーションで有効な指定ポリシーを返します。

指定タイプが現在の ORB でサポートされていないか，そのタイプのポリシーオブジェクトが現在の ORB に関連付けられていないためにポリシータイプが有効でない場合，標準マイナーコード 2 で INV_POLICY 例外が生成されます。

パラメータ	説明
_type	返されるポリシーを指定するポリシーのタイプ。

```
virtual void add_request_service_context(const IOP::ServiceContext&
_service_context,CORBA::Boolean _replace) = 0;
```

インターセプタでサービスコンテキストを要求に追加します。

サービスコンテキストの順序の宣言はありません。サービスコンテキストは，必ずしも追加された順序では実行されません。

パラメータ	説明
_service_context	要求に追加される IOP::ServiceContext。
_replace	指定された ID のサービスコンテキストがすでに存在する場合のこのメソッドの動作を示します。false の場合は，標準マイナーコード 15 で BAD_INV_ORDER が生成されます。true の場合，既存のサービスコンテキストが新しいサービスコンテキストに置き換えられます。

ClientRequestInterceptor

```
class PortableInterceptor::ClientRequestInterceptor : public virtual Interceptor
```

ClientRequestInterceptor クラスは，ユーザー定義のクライアント側インターセプタを派生するために使用されます。ClientRequestInterceptor インスタンスは VisiBroker ORB に登録されます（223 ページの「ORBInitializer」を参照）。

インクルードファイル

この構造体を使用する場合は，**PortableInterceptor_c.hh** ファイルをインクルードする必要があります。

ClientRequestInterceptor のメソッド

```
virtual void send_request(ClientRequestInfo_ptr _ri) = 0;
```

send_request() インターセプトポイントは，インターセプタが要求情報を照会し，サーバーに要求を送信する前にサービスコンテキストを変更するためのポイントです。

このインターセプトポイントでは，システム例外が生成される可能性があります。生成された場合，ほかのインターセプタの send_request() インターセプトポイントは呼び出されません。フロースタック上のインターセプタがポップし，それぞれの receive_exception() インターセプトポイントが呼び出されます。

このインターセプトポイントでは，ForwardRequest 例外が生成される可能性があります（218 ページの「ForwardRequest」を参照）。インターセプトによってこの例外が生成された場合，ほかのインターセプタの send_request メソッドは呼び出されません。フロースタック上のインターセプタがポップし，それぞれの receive_other() インターセプトポイントが呼び出されます。

パラメータ	説明
_ri	インターセプタで使用される ClientRequestInfo インスタンス。

```
virtual void send_poll(ClientRequestInfo_ptr _ri) = 0;
```

send_poll() インターセプトポイントは，TII（Time-Independent Invocation）ポーリングが応答シーケンスを取得する間にインターセプタが情報を照会するためのポイントです。

ただし，VisiBroker ORB は TII をサポートしないので，send_poll() インターセプトポイントは呼び出されません。

パラメータ	説明
_ri	インターセプタで使用される ClientRequestInfo インスタンス。

```
virtual void receive_reply(ClientRequestInfo_ptr _ri) = 0;
```

receive_reply() インターセプトポイントはサーバーから応答が返され，制御がクライアントに戻るまでの間にインターセプタが情報を照会するポイントです。

このインターセプトポイントでは，システム例外が生成される可能性があります。生成された場合，ほかのインターセプタの receive_reply() メソッドは呼び出されません。フロースタック上のほかのインターセプタでは，それぞれの receive_exception() インターセプトポイントが呼び出されます。

パラメータ	説明
_ri	インターセプタで使用される ClientRequestInfo インスタンス。

```
virtual void receive_exception(ClientRequestInfo_ptr _ri) = 0;
```

receive_exception() インターセプトポイントは例外の生成時に呼び出されます。これによって，例外がクライアントに生成される前にインターセプタが例外の情報を照会できます。

このインターセプトポイントでは，システム例外が生成される可能性があります。その場合は，フロースタックからポップされる後続のインターセプタが receive_exception() の呼び出しで受け取る例外が変更されます。クライアントには，前回インターセプタが生成した例外が生成されます。インターセプタによる例外の変更がなければ元の例外がクライアントに生成されます。

このインターセプトポイントでは，ForwardRequest 例外が生成される可能性があります（218 ページの「ForwardRequest」を参照）。インターセプタによってこの例外が生成された場合，ほかのインターセプタの receive_exception() インターセプトポイントは呼び出されません。フロースタック上のインターセプタがポップし，それぞれの receive_other() インターセプトポイントが呼び出されます。

パラメータ	説明
_ri	インターセプタで使用される ClientRequestInfo インスタンス。

```
virtual void receive_other(ClientRequestInfo_ptr _ri) = 0;
```

この receive_other() インターセプトポイントでインターセプタは，標準応答や例外以外の要求の結果に関する情報を照会します。LOCATION_FORWARD 状態を受信して返される

GIOP 応答など，要求の結果が再試行になる場合もあれば，非同期呼び出しでは要求に対してただちに応答が返されるかわりに制御がクライアントに戻され，終了インターセプトポイントが呼び出されることもあります。

再試行では，再試行の指示に対して新しい要求が生成される場合とそうでない場合がありますが，これは適用されるポリシーによって異なります。新しい要求が生成される場合，インターセプタに対してこの要求が新しい要求である限り，元の要求と再試行の間には 1 つの相関関係が維持されます。クライアントに制御が戻っておらず，PortableInterceptor::Current が適用される要求が元の要求と再試行要求の両方で同じ要求になるためです（216 ページの「Current」を参照）。

このインターセプトポイントでは，システム例外が生成される可能性があります。生成された場合，ほかのインターセプタの receive_other() インターセプトポイントは呼び出されません。フロースタック上のインターセプタがポップし，それぞれの receive_exception() インターセプトポイントが呼び出されます。

このインターセプトポイントでは，ForwardRequest 例外が生成される可能性があります（218 ページの「ForwardRequest」を参照）。インターセプタがこの例外を生成すると，ForwardRequest 例外から提供された新しい情報を使用して，後続のインターセプタの receive_other() メソッドが呼び出されます。

パラメータ	説明
_ri	インターセプタで使用される ClientRequestInfo インスタンス。

Codec

class IOP::**Codec**

ORB サービスが使用する IOR コンポーネントの形式とサービスコンテキストデータの形式は，通常 IDL 定義のデータ型インスタンスをエンコードした CDR カプセル化で定義します。Codec には，これらのコンポーネントを対応する IDL データ型形式と CDR カプセル化形式の間で転送する機構があります。

Codec は CodecFactory から取得できます。CodecFactory は，ORB::resolve_initial_references ("CodecFactory") の呼び出しを介して取得されます。

インクルードファイル

この構造体を使用する場合は，**IOP_c.hh** ファイルをインクルードする必要があります。

Codec のメンバークラス

class Codec::**InvalidTypeForEncoding** : public CORBA_UserException

エンコーディングに無効な型を指定すると，encode() または encode_value() でこの例外が生成されます。

class Codec::**FormatMismatch** : public CORBA_UserException

Octet シーケンス内のデータを CORBA::Any にデコードできない場合，decode() または decode_value() でこの例外が生成されます。

class Codec::**TypeMismatch** : public CORBA_UserException

TypeCode が Octet シーケンス内のデータと一致しない場合，decode_value() でこの例外が生成されます。

Codec のメソッド

```
virtual CORBA::OctetSequence* encode(const CORBA::Any& _data) = 0;
```

CORBA::Any 形式の指定データを，Codec に対して有効なエンコード形式をベースにした Octet シーケンスに変換します。この Octet シーケンスには，TypeCode と指定データ型のデータが含まれています。

このオペレーションで InvalidTypeForEncoding 例外が生成される場合があります。

パラメータ	説明
_data	Octet シーケンスにエンコードされる CORBA::Any 形式のデータ。

```
virtual CORBA::Any* decode(const CORBA::OctetSequence& _data) = 0;
```

現在の Codec で有効なエンコード形式に基づいて，指定された Octet シーケンスを CORBA::Any オブジェクトにデコードします。

Octet シーケンスを CORBA::Any にデコードできない場合，FormatMismatch 例外が生成されます。

パラメータ	説明
_data	CORBA::Any にデコードされる Octet シーケンス形式のデータ。

```
virtual CORBA::OctetSequence* encode_value(const CORBA::Any& _data) = 0;
```

CORBA::Any オブジェクトを，現在の Codec で有効なエンコード形式をベースにした Octet シーケンスに変換します。TypeCode ではなく CORBA::Any からのデータだけがエンコードされます。

このオペレーションで InvalidTypeForEncoding 例外が生成される場合があります。

パラメータ	説明
_data	エンコードされた CORBA::Any のデータを格納する Octet シーケンス。

```
virtual CORBA::Any* decode_value(const CORBA::OctetSequence& _data,
CORBA::TypeCode_ptr _tc) = 0;
```

指定された TypeCode と現在の Codec で有効なエンコード形式に基づいて，指定された Octet シーケンスを CORBA::Any にデコードします。

Octet シーケンスを CORBA::Any にデコードできない場合，FormatMismatch 例外が生成されます。

パラメータ	説明
_data	CORBA::Any にデコードされる Octet シーケンス形式のデータ。
_tc	データのデコードに使用する TypeCode。

CodecFactory

```
class IOP::CodecFactory
```

Codec を取得するために使用されます。CodecFactory は，ORB::resolve_initial_references ("CodecFactory") の呼び出しを介して取得されます。

インクルードファイル

このクラスを使用する場合は，IOP_c.hh ファイルをインクルードする必要があります。

CodecFactory のメンバー

```
class CodecFactory::UnknownEncoding : public CORBA_UserException
```

CodecFactory が Codec を作成できないときこの例外が生成されます。次の create_codec() 関数を参照してください。

CodecFactory のメソッド

```
virtual Codec_ptr create_codec(const Encoding& _enc) = 0;
```

create_codec() メソッドは，指定エンコードで Codec を作成します。

指定した Codec を現在のファクトリで作成できない場合，UnknownEncoding 例外が生成されます。

パラメータ	説明
_enc	Codec を作成するためのエンコードを指定します。

Current

```
class PortableInterceptor::Current: public virtual CORBA::Current, public virtual
CORBA_Object
```

Current クラスは，単なるスロットテーブルです。ここにあるスロットを利用してサービスは，サービスのコンテキストと要求や応答のサービスコンテキスト間でコンテキストデータを転送します。

Current を使用しようとするすべてのサービスは，初期化（226 ページの「virtual CORBA::ULong allocate_slot_id() = 0;」を参照）の際に 1 つまたは複数のスロットを確保し，要求や応答の処理にこれらのスロットを使用します。

起動する前に，ORB::resolve_initial_references("PICurrent") を呼び出して Current を取得します。

スレッドスコープからリクエストスコープに移動した Current のデータは，RequestInfo オブジェクトの get_slot() メソッドでインターセプトポイントから利用できます。Current は resolve_initial_references() から取得できますが，これはインターセプタのスレッドスコープ Current です。

インクルードファイル

このクラスを使用する場合は，PortableInterceptor_c.hh ファイルをインクルードする必要があります。

Current のメソッド

```
virtual CORBA::Any* get_slot(CORBA::ULong _id);
```

サービスが PICurrent に設定するスロットデータは，get_slot() メソッドで取得します。データは CORBA::Any オブジェクト形式です。

指定されたスロットが設定されていない場合は，TCKind 値が tk_null のタイプコードを保持し，値を保持しない CORBA::Any が返されます。

割り当てられていないスロットで get_slot() が呼び出された場合は，InvalidSlot が生成されます。

ORB 初期化子（223 ページの「ORBInitializer」を参照）から get_slot() を呼び出すと，マイナーコード 14 の BAD_INV_ORDER が生成されます。

パラメータ	説明
_id	データが返されるスロットの SlotId。

```
virtual void set_slot(CORBA::ULong _id, const CORBA::Any& _data);
```

サービスは set_slot(). でスロットにデータを設定します。データは CORBA::Any オブジェクト形式です。

データがすでにスロットにある場合，古いデータを上書きします。

割り当てられていないスロットで set_slot() が呼び出された場合は，InvalidSlot が生成されます。

ORB 初期化子（223 ページの「ORBInitializer」を参照）から set_slot() を呼び出すと，マイナーコード 14 の BAD_INV_ORDER が生成されます。

パラメータ	説明
_id	データが設定されるスロットの SlotId。
_data	CORBA::Any オブジェクト形式のデータです。指定された ID のスロットに設定されます。

Encoding

```
struct IOP::Encoding
```

この構造体は Codec のエンコード形式を定義します。CDR カプセル化エンコーディングなどエンコード形式の詳細や，その形式のメジャーバージョン，マイナーバージョンを記述します。

次のようなエンコーディングがサポートされています。

- ENCODING_CDR_ENCAPS，バージョン 1.0
- ENCODING_CDR_ENCAPS，バージョン 1.1
- ENCODING_CDR_ENCAPS，バージョン 1.2
- ENCODING_CDR_ENCAPS（GIOP の将来のバージョンすべて）

インクルードファイル

この構造体を使用する場合は，IOP_c.hh ファイルをインクルードする必要があります。

メンバー

```
CORBA::Short format;
```

このメンバーは Codec のエンコード形式を持ちます。

```
CORBA::Octet major_version;
```

このメンバーは Codec のメジャーバージョン番号を定義します。

```
CORBA::Octet minor_version;
```

このメンバーは Codec のマイナーバージョン番号を定義します。

ExceptionList

```
class Dynamic::ExceptionList
```

このクラスは，クラス RequestInfo のメソッド exceptions() から返された例外情報を保持するために使用されます。これは，CORBA::TypeCode タイプの変数長配列の実装です。ExceptionList の長さは，実行時にわかります。

詳細については，228 ページの「virtual Dynamic::ExceptionList* exceptions() = 0;」を参照してください。

インクルードファイル

このクラスを使用する場合は，Dynamic_c.hh ファイルをインクルードする必要があります。

ForwardRequest

```
class PortableInterceptor::ForwardRequest : public CORBA_UserException
```

インターセプタは，ForwardRequest 例外で指定された新しいオブジェクトを使用して要求を再試行するように ORB に指示します。再試行を促すこの動作は，ORB がインターセプタから ForwardRequest を受け取らないと実行されません。ForwardRequest がほかの場所で生成されても，ユーザー例外の通常の場所で生成された場合と同様に ORB から伝送されます。

あるインターセプタの呼び出しに対して別のインターセプタが ForwardRequest 例外を生成すると，そのインターセプトポイントにほかのインターセプタは呼び出されません。フロースタック上のほかのインターセプタには，クライアント上の receive_other() またはサーバー上の send_other() のうち，それぞれ適切な終了インターセプトポイントを呼び出します。receive_other() の reply_status() または send_other() が LOCATION_FORWARD を返します。

インクルードファイル

このクラスを使用する場合は，PortableInterceptor_c.hh ファイルをインクルードする必要があります。

Interceptor

```
class PortableInterceptor::Interceptor
```

このクラスは，すべてのインターセプタの派生元になるベースクラスです。

インクルードファイル

このクラスを使用する場合は，PortableInterceptor_c.hh ファイルをインクルードする必要があります。

Interceptor のメソッド

```
virtual char* name() = 0;
```

インターセプタの名前を返します。インターフェースごとに名前を付けることができ，これをもとにインターセプタリストのデータを整列できます。VisiBroker ORB では，各インターセプタについて登録できる名前はインターセプタの種類ごとに 1 つです。インターセプタは匿名でもかまいません。その場合，name 属性に空の文字列を指定します。VisiBroker ORB には，匿名インターセプタをいくつでも登録できます。

```
virtual void destroy() = 0;
```

ORB::destroy() の中で呼び出されます。アプリケーションが ORB::destroy() を呼び出すと，VisiBroker ORB は以下を実行します。

1. 実行中のすべての要求が終了するまで待ちます。
2. 各インターセプタの Interceptor::destroy() メソッドを呼び出します。
3. ORB の廃棄を完了します。

廃棄中の VisiBroker ORB で実装したオブジェクトのオブジェクトリファレンスの Interceptor::destroy() からメソッドを呼び出した場合の結果は予測できません。ただし，廃棄中でない VisiBroker ORB で実装したオブジェクトのメソッドは呼び出すことができます。これは，廃棄中の VisiBroker ORB はサーバーとしての機能はなくてもクライアントとしての機能があることを意味します。

IORInfo

```
class PortableInterceptor::IORInfo
```

IORInfo インターフェースは，IOR の構築時に適用可能なポリシーへのアクセスを持つサーバー側 ORB サービスを提供し，さらにコンポーネントを追加する機能も提供します。ORB　はこのインターフェースインプリメンテーションのインスタンスを IORInterceptor::establish_components() にパラメータとして渡します。

次の表は，IORInterceptor で定義するメソッドにおける IORInfo の各属性やメソッドの有効性をまとめたものです。

表 12.4　IORInfo の有効性

	establish_components	components_established
get_effective_policy	はい	はい
add_component	はい	いいえ
add_component_to_profile	はい	いいえ
manager_id	はい	はい
state	はい	はい
adapter_template	いいえ	はい
current_factory	いいえ	はい

IORInfo の属性やメソッドに無効な呼び出しがあると，標準マイナーコード 14 の BAD_INV_ORDER 例外が生成されます。

インクルードファイル

このクラスを使用する場合は，PortableInterceptor_c.hh ファイルをインクルードする必要があります。

IORInfo のメソッド

```
virtual CORBA::Policy_ptr get_effective_policy(CORBA::ULong _type) = 0;
```

ORB サービスインプリメンテーションは，get_effective_policy() メソッドを呼び出して，構築中の IOR に対して有効なサーバー側ポリシーの型がどれであるかを判定します。POA で実装したオブジェクトに IOR を構築する場合，その POA を作成した PortableServer::POA::create_POA() 呼び出しに渡したすべての Policy オブジェクトは get_effective_policy でアクセスできます。

指定した型のポリシーを ORB が認識できない場合にこのメソッドを実行すると，標準マイナーコード 3 の INV_POLICY 例外が生成されます。

パラメータ	説明
_type	返されるポリシーの型を指定する CORBA::PolicyType。

```
virtual void add_ior_component(const IOP::TaggedComponent& _a_component) = 0;
```

establish_components() から呼び出して，IOR の構築時に組み込むセットにタグ付きコンポーネントを追加します。このセットのコンポーネントは，すべてのプロファイルに組み込まれます。

同じコンポーネント ID のコンポーネントがいくつあってもかまいません。

パラメータ	説明
_a_component	追加する IOP::TaggedComponent。

```
virtual void add_ior_component_to_profile(const IOP::TaggedComponent&
_a_component,CORBA::ULong _profile_id) = 0;
```

establish_components() から呼び出して，IOR の構築時に組み込むセットにタグ付きコンポーネントを追加します。このセットのコンポーネントは，指定プロファイルに組み込まれます。

同じコンポーネント ID のコンポーネントがいくつあってもかまいません。

指定プロファイル ID で既知のプロファイルがない場合，またはそのプロファイルにコンポーネントを追加できない場合，標準マイナーコード 29 の BAD_PARAM 例外が生成されます。

パラメータ	説明
_a_component	追加する IOP::TaggedComponent。
_profile_id	このコンポーネントを追加するプロファイルの IOP::ProfileId。

```
virtual CORBA::Long manager_id() = 0;
```

アダプタのマネージャに不透過ハンドルを提供する属性を返します。同じアダプタマネージャで管理するアダプタの状態変化を報告します。

```
virtual CORBA::Short state() = 0;
```

アダプタの現在の状態を返します。HOLDING, ACTIVE, DISCARDING, INACTIVE, NON_EXISTENT のうちの 1 つでなければなりません。

```
virtual ObjectReferenceTemplate_ptr adapter_template() = 0;
```

IOR インターセプタを呼び出すたびにオブジェクトリファレンステンプレートを取得するための属性を返します。オブジェクトリファレンステンプレートを直接作成することはできません。adapter_template() の戻り値は，アダプタポリシーおよび

add_component() と add_component_to_profile() の IOR インターセプタ呼び出しに対して作成されるテンプレートです。adapter_template() の戻り値は，オブジェクトアダプタが存続する間変更されません。

```
virtual ObjectReferenceFactory_ptr current_factory() = 0;
```

このメソッドは，ファクトリへのアクセスを提供する属性を返します。これは，オブジェクトリファレンスの作成時にアダプタによって使用されます。current_factory() の初期値は adapter_template 属性と同じですが，current_factory を別のファクトリに設定すれば変更できます。オブジェクトアダプタによって作成されるオブジェクトリファレンスは，どれも current_factory の make_object() メソッドを呼び出して作成する必要があります。

```
virtual void current_factory(ObjectReferenceFactory_ptr _current_factory) = 0;
```

属性の current_factory を設定します。アダプタが使用する <current_factory 属性の値は，components_established メソッドの呼び出しの間でないと設定できません。

パラメータ	説明
_current_factory	設定する current_factory オブジェクト。

IORInfoExt

```
class IORInfoExt: public PortableInterceptor::IORInfo
```

ポータブルインターセプタで POA スコープ付きサーバーリクエストインターセプタをインストールするための VisiBroker 拡張モジュールです。IORInfoExt インターフェースは，IORInfo インターフェースを継承します。また，POA スコープ付きサーバーリクエストインターセプタをサポートするメソッドも備えています。

インクルードファイル

このクラスを使用する場合は，PortableInterceptorExt_c.hh ファイルをインクルードする必要があります。

IORInfoExt のメソッド

```
virtual void add_server_request_interceptor (ServerRequestInterceptor_ptr _interceptor) = 0;
```

（ORB スコープ付きではなく）POA スコープ付きサーバー側リクエストインターセプタをサービスに追加するために使用されます。

パラメータ	説明
_interceptor	追加する ServerRequestInterceptor。

```
virtual char* full_poa_name();
```

完全な POA 名を返します。

IORInterceptor

```
class PortableInterceptor::IORInterceptor : public virtual Interceptor
```

クライアントの ORB サービスインプリメンテーションは，サーバーやオブジェクトの ORB サービス関連機能を記述する情報をオブジェクトリファレンスに追加しないと正しく機能しないことがあります。

このインターフェースは，IORInterceptor インターフェースと IORInfo インターフェースを介してサポートされています。

IOR インターセプタは，IOR 内のプロファイルでタグ付きコンポーネントを確立するために使用します。

インクルードファイル

このクラスを使用する場合は，PortableInterceptor_c.hh ファイルをインクルードする必要があります。

IORInterceptor のメソッド

```
virtual void establish_components(IORInfo_ptr _info) = 0;
```

1 つ以上のプロファイルに組み込むコンポーネントのリストを構築するとき，サーバー側 ORB は，すべての登録 IORInterceptor インスタンスで，establish_components() メソッドを呼び出します。このメソッドは，必ずしもオブジェクトリファレンスごとに呼び出す必要はありません。POA の場合，これらの呼び出しは POA::create_POA() の呼び出しごとに行います。ほかのアダプタでは，アダプタを初期化するたびにこれらの呼び出しが行われるのが普通です。この段階では，アダプタテンプレートに必要な情報（コンポーネント）が構築途中なので，アダプタテンプレートを利用することはできません。

パラメータ	説明
_info	適用できるポリシーを照会したり，生成される IOR に組み込むコンポーネントを追加するために，ORB サービスによって使用される IORInfo インスタンス。

```
virtual void components_established(IORInfo_ptr _info) = 0;
```

すべての establish_components() メソッドの呼び出しが完了すると，components_established() メソッドがすべての登録 IOR インターセプタで起動します。アダプタテンプレートが利用できるようになるのはこの段階です。この段階では，current_factory 属性の取得や設定が可能です。

components_established() で発生したすべての例外が，components_established() の呼び出しに返されます。POA の場合，これによって create_POA 呼び出しが失敗し，標準マイナーコード 6 の OBJ_ADAPTER 例外が create_POA() の起動側に返ります。

パラメータ	説明
_info	適用できるポリシーにアクセスするために，ORB サービスによって使用される IORInfo インスタンス。

```
virtual void adapter_manager_state_changed(CORBA::Long _id, CORBA::Short _state) = 0;
```

アダプタマネージャの状態が変化すると，adapter_manager_state_changed() メソッドがすべての登録 IOR インターセプタで起動します。

adapter_manager_state_changed() が状態変化を伝えられる場合，adapter_state_changed() による伝達は行われません。

パラメータ	説明
_id	適用できるポリシーにアクセスするために，ORB サービスによって使用される IORInfo インスタンス。
_state	オブジェクトアダプタの新しい状態。

```
virtual void adapter_state_changed(const ObjectReferenceTemplateSeq& _templates,CORBA::Short
_state) = 0;
```

アダプタマネージャの状態変化に無関係な理由で 1 つ以上のアダプタの状態が変化すると，オブジェクトアダプタの状態変化がこのメソッドに伝えられます。テンプレート引数は，状態が変化したオブジェクトアダプタをテンプレート ID 情報で伝えます。シーケンスには，状態移行が伝えられたすべてのオブジェクトアダプタのアダプタテンプレートが格納されます。

パラメータ	説明
_templates	状態が変化したオブジェクトアダプタをテンプレート ID 情報で識別します。
_state	オブジェクトアダプタの新しい状態。

ORBInitializer

```
class PortableInterceptor::ORBInitializer
```

対応する ORBInitializer オブジェクトを登録するとインターセプタが登録され，これによって ORBInitializer クラスが実装されます。ORB は，初期化時に，登録済みの ORBInitializer を個別に呼び出して，それぞれのインターセプタの登録に必要な ORBInitInfo オブジェクトを渡します。

インクルードファイル

このクラスを使用する場合は，PortableInterceptor_c.hh ファイルをインクルードする必要があります。

ORBInitializer のメソッド

```
virtual void pre_init(ORBInitInfo_ptr _info) = 0;
```

ORB の初期化中に呼び出されます。インターセプタが登録した初期サービスをほかのインターセプタで使用する場合は，その初期サービスは ORBInitInfo::register_initial_reference() への呼び出しの時点で登録されます。

パラメータ	説明
_info	インターセプタの登録に必要な初期化属性とメソッドを渡すオブジェクト。

```
virtual void post_init(ORBInitInfo_ptr _info) = 0;
```

ORB の初期化中に呼び出されます。初期化の一環として，サービスで初期リファレンスを解決する場合，この時点ですべての初期リファレンスが利用できることが条件です。

post_init() メソッドの呼び出しで ORB の初期化がすべて完了するわけではありません。post_init() 呼び出しに続いて登録インターセプタのリストを ORB に登録します。これが最後のタスクです。したがって，post_init() に対する呼び出しの間，ORB にそ

のインターセプタは格納されていません。ORB が仲介する呼び出しを post_init() 内で生成すると，その呼び出しではリクエストインターセプタは起動しません。同様に，IOR を作成するメソッドを実行すると，IOR インターセプタは起動しません。

パラメータ	説明
_info	インターセプタの登録に必要な初期化属性とメソッドを渡すオブジェクト。

ORBInitInfo

class PortableInterceptor::**ORBInitInfo**

インターセプタを登録するには，ORBInitInfo クラスを ORBInitializer オブジェクトに渡す必要があります。

インクルードファイル

このクラスを使用する場合は，PortableInterceptor_c.hh ファイルをインクルードする必要があります。

ORBInitInfo のメンバークラス

class **DuplicateName** : public CORBA_UserException;

ORB に登録できるインターセプタの名前は，インターセプタの種類ごとに 1 つだけです。同じ名前で第 2 のインターセプタを登録しようとすると，DuplicateName 例外が生成されます。

インターセプタは匿名でもかまいません。その場合，name 属性に空の文字列を指定します。匿名インターセプタはいくつでも ORB に登録できるため，登録するインターセプタが匿名の場合，登録オペレーションで DuplicateName 例外は生成されません。

class **InvalidName**: public CORBA_UserException

これは，register_initial_reference() と resolve_initial_references() によって生成される例外です。

次の場合に，register_initial_reference() は InvalidName を生成します。

- このメソッドを空の文字列 ID で呼び出した場合。
- 登録済みの ID でこのメソッドを呼び出した場合。OMG で定義したデフォルトの名前もこの対象です。

解決される名前が無効な場合，resolve_initial_references() は InvalidName を生成します。

ORBInitInfo のメソッド

virtual CORBA::StringSequence* **arguments**() = 0;

ORB_init() に渡された引数を返します。ORB の引数が格納されている場合とそうでない場合があります。

virtual char* **orb_id**() = 0;

初期化中の ORB の ID を返します。

```
virtual IOP::CodecFactory_ptr codec_factory() = 0;
```

このメソッドは，IOP::CodecFactory を返します。CodecFactory を取得するには，通常は ORB::resolve_initial_references("CodecFactory") を呼び出しますが，この段階ではまだ ORB を利用できず，サービスコンテキストの処理でインターセプタに Codec が必要なため，ORB の初期化では Codec を取得する手段が必要です。

```
virtual void register_initial_reference(const char* _id, CORBA::Object_ptr _obj) = 0;
```

ID「Y」，オブジェクト YY でこのメソッドを呼び出すと，register_initial_reference() に対する後続の呼び出しではオブジェクト YY が返されます。

これは，ORB::register_initial_reference() と同じです。同じ機能がここにあるのは，この段階では初期化が不十分なため ORB を利用できないにもかかわらず，初期リファレンスをインターセプタ登録の一環で登録しなければならない場合があるからです。唯一の違いは，このメソッドの ORB バージョンが PIDL（CORBA::ORB::ObjectId と CORBA::ORB::InvalidName）を使用するのに対して，このインターフェースのバージョンは現在のインターフェースで定義された IDL を使用する点にあり，セマンティクスに違いはありません。

次の場合に，register_initial_reference() は InvalidName を生成します。

- このメソッドを空の文字列 ID で呼び出した場合。
- 登録済みの ID でこのメソッドを呼び出した場合。OMG で定義したデフォルトの名前もこの対象です。

パラメータ	説明
_id	初期リファレンスを識別する ID。
_obj	初期リファレンスそのもの。

```
virtual CORBA::Object_ptr resolve_initial_references(const char* _id) = 0;
```

post_init() の間だけ有効なメソッドです。ORB::resolve_initial_references() と同じ働きをします。同じ機能がここにあるのは，この段階では初期化が不十分なため ORB を利用できないにもかかわらず，初期リファレンスがインターセプタ登録で必要になる場合があるからです。

パラメータ	説明
_id	初期リファレンスを識別する ID。

解決される名前が無効な場合，resolve_initial_references() は InvalidName を生成します。

```
virtual void add_client_request_interceptor(ClientRequestInterceptor_ptr _interceptor) = 0;
```

クライアント側リクエストインターセプタのリストに，クライアント側リクエストインターセプタを追加します。

クライアント側リクエストインターセプタが現在のインターセプタの名前で登録済みの場合は，DuplicateName 例外が生成されます。

パラメータ	説明
_interceptor	追加する ClientRequestInterceptor。

```
virtual void add_server_request_interceptor(ServerRequestInterceptor_ptr _interceptor) = 0;
```

サーバー側リクエストインターセプタのリストに，サーバー側リクエストインターセプタを追加します。

サーバー側リクエストインターセプタが現在のインターセプタの名前で登録済みの場合は，DuplicateName 例外が生成されます。

パラメータ	説明
_interceptor	追加する ServerRequestInterceptor。

```
virtual void add_ior_interceptor(IORInterceptor_ptr _interceptor) = 0;
```

IOR インターセプタのリストに，IOR インターセプタを追加します。

IOP インターセプタが現在のインターセプタの名前で登録済みの場合は，DuplicateName 例外が生成されます。

パラメータ	説明
_interceptor	追加する IORInterceptor。

```
virtual CORBA::ULong allocate_slot_id() = 0;
```

割り当て済みのスロットのインデックスを返します。

サービスは，PortableInterceptor::Current のスロットを割り当てるために allocate_slot_id を呼び出します。

メモ　スロット ID は，ORB 初期化子の中で割り当てることができますが，スロット自体は初期化できません。ORB 初期化子で Current（「Current」を参照）の set_slot() または get_slot() を呼び出すと，マイナーコード 14 の BAD_INV_ORDER が生成されます。

```
virtual void register_policy_factory(CORBA::ULong _type, PolicyFactory_ptr _policy_factory) = 0;
```

指定された PolicyType に対する PolicyFactory を作成します。

指定した PolicyType の PolicyFactory がすでに存在する場合は，標準マイナーコード 16 の BAD_INV_ORDER 例外が生成されます。

パラメータ	説明
_type	指定された PolicyFactory がサービスを提供する CORBA::PolicyType。
_policy_factory	指定された CORBA::PolicyType に対するファクトリ。

Parameter

```
struct Dynamic::Parameter
```

この構造体にはパラメータ情報を保存します。この構造体は，ParameterList で使用される要素です（227 ページの「ParameterList」を参照）。

インクルードファイル

この構造体を使用する場合は，Dynamic_c.hh ファイルをインクルードする必要があります。

メンバー

```
CORBA::Any argument;
```

CORBA::Any 形式でパラメータデータを保存します。

```
CORBA::ParameterMode mode;
```

パラメータモードを指定します。列挙値 PARAM_IN, PARAM_OUT, PARAM_INOUT のいずれかになります。

ParameterList

class Dynamic::**ParameterList**

このクラスは，クラス RequestInfo のメソッド arguments() から返された例外情報を渡すのに使用します。これは，Parameter タイプの変数長配列の実装です。ParameterList の長さは，実行時にわかります。

詳細については，228 ページの「virtual Dynamic::ParameterList* arguments() = 0;」を参照してください。

インクルードファイル

このクラスを使用する場合は，Dynamic_c.hh ファイルをインクルードする必要があります。

PolicyFactory

class PortableInterface::**PolicyFactory**

ポータブル ORB サービスインプリメンテーションは，ORB の初期化の際に PolicyFactory インターフェースのインスタンスを登録して，CORBA.ORB.create_policy() で構築するポリシーの型を有効にします。POA では，この方法で ORBInitInfo に登録したポリシーをすべて保存する必要があります。

インクルードファイル

このクラスを使用する場合は，PortableInterceptor_c.hh ファイルをインクルードする必要があります。

PolicyFactory のメソッド

virtual CORBA::Policy_ptr **create_policy**(CORBA::ULong **_type**, const CORBA::Any& **_value**) = 0;

PolicyFactory が登録されたときの PolicyType に対して CORBA::ORB::create_policy() が呼び出されると，ORB は，登録されている PolicyFactory インスタンスの create_policy() を呼び出します。create_policy() メソッドは，指定された CORBA::Any に対応する値を持つ CORBA::Policy から派生したインターフェースのインスタンスを返します。失敗すると，CORBA::ORB::create_policy() に記述した例外が生成されます。

パラメータ	説明
_type	作成されるポリシーの型を指定する CORBA::PolicyType。
_value	CORBA::Policy を構築するデータを含む CORBA::Any。

RequestInfo

class PortableInterceptor::**RequestInfo**

このクラスは，ClientRequestInfo と ServerRequestInfo の派生元になるベースクラスです。各インターセプトポイントにはインターセプタが要求情報にアクセスするためのオ

ブジェクトが与えられます。クライアント側とサーバー側のインターセプトポイントはそれぞれ異なる情報を扱うため，情報オブジェクトは 2 つあります。ClientRequestInfo がクライアント側インターセプトポイントに渡され，ServerRequestInfo がサーバー側インターセプトポイントに渡されます。ただし両方に共通の情報があるので，いずれもこの共通インターフェースである RequestInfo を継承します。

インクルードファイル

このクラスを使用する場合は，PortableInterceptor_c.hh ファイルをインクルードする必要があります。

RequestInfo のメソッド

```
virtual CORBA::ULong request_id() = 0;
```

アクティブ要求／応答シーケンスを一意で識別する ID を返します。要求／応答シーケンスが終了すると，この ID は再利用できます。

メモ この ID は，GIOP request_id とは異なります。使用するトランスポート機構が GIOP の場合，これらの ID が同じになる可能性は高くなりますが，必ずしもそうなるとは限らず，またそれが条件というわけでもありません。

```
virtual char* operation() = 0;
```

呼び出されるオペレーションの名前を返します。

```
virtual Dynamic::ParameterList* arguments() = 0;
```

実行されるオペレーションの引数を含む Dynamic::ParameterList を返します。引数がない場合，この属性は長さゼロのシーケンスになります。

```
virtual Dynamic::ExceptionList* exceptions() = 0;
```

このオペレーション起動で生成されるユーザー例外の TypeCodes を記述する型 Dynamic::ExceptionList の配列を返します。ユーザー例外がない場合，この属性は，長さゼロのシーケンスになります。

```
virtual CORBA::StringSequence* contexts() = 0;
```

このオペレーション起動で渡される可能性のあるコンテキストを記述する CORBA::StringSequence を返します。コンテキストがない場合，この属性は，長さゼロのシーケンスになります。

```
virtual CORBA::StringSequence* operation_context() = 0;
```

要求に送られたコンテキストを含む CORBA::StringSequence の配列を返します。

```
virtual CORBA::Any* result() = 0;
```

オペレーション起動の結果を含むデータを CORBA::Any 形式で返します。オペレーションの戻り型が void の場合，この属性は TCKind 値が tk_void のタイプコードを含む CORBA::Any になります。

```
virtual CORBA::Boolean response_expected() = 0;
```

応答が必要かどうかを示すブール値を返します。

クライアントでは，response_expected() が false の場合は応答が返されないため，receive_reply() を呼び出すことができません。例外が発生しない限り，receive_other() が呼び出され，receive_exception() が呼び出されます。

```
virtual CORBA::Short sync_scope() = 0;
```

response_expected() が false の場合にだけメッセージング仕様で定義した属性を返します。response_expected() が true の場合，sync_scope() の値は定義されていません。このメソッドはクライアントに制御が戻るまでの要求の進捗状況を定義します。この属性には次の値の 1 つを指定する必要があります。

- Messaging::SYNC_NONE
- Messaging::SYNC_WITH_TRANSPORT
- Messaging::SYNC_WITH_SERVER
- Messaging::SYNC_WITH_TARGET

サーバーでは，すべてのスコープでターゲットオペレーションの戻り値から応答が作成されますが，クライアントに応答は返されません。クライアントに応答は返されませんが，応答自体は生成されるため，標準サーバー側インターセプトポイントが生成されます（たとえば，receive_request_service_contexts()，receive_request()，send_reply()，send_exception()）。

SYNC_WITH_SERVER と SYNC_WITH_TARGET の場合，ターゲットが起動する前にサーバーは空の応答をクライアントに返します。サービス側インターセプタは，この応答をインターセプトしません。

```
virtual CORBA::Short reply_status() = 0;
```

オペレーション起動の結果の状態を記述する属性を返します。次の値の 1 つになります。

- PortableInterceptor::SUCCESSFUL = 0
- PortableInterceptor::SYSTEM_EXCEPTION = 1
- PortableInterceptor::USER_EXCEPTION = 2
- PortableInterceptor::LOCATION_FORWARD = 3
- PortableInterceptor::TRANSPORT_RETRY = 4

クライアント側：

- receive_reply インターセプトポイントでこの属性がとる値は SUCCESSFUL だけです。
- receive_exception インターセプトポイントでこの属性がとる値は SYSTEM_EXCEPTION または USER_EXCEPTION です。
- receive_other インターセプトポイントでこの属性がとる値は，SUCCESSFUL，LOCATION_FORWARD，または TRANSPORT_RETRY のいずれかです。SUCCESSFUL は，非同期要求の戻りが成功したことを表します。LOCATION_FORWARD は，返った応答の状態が LOCATION_FORWARD であったことを表します。TRANSPORT_RETRY は，トランスポート機構が再試行を示したことを表します。たとえば，状態 NEEDS_ADDRESSING_MODE の GIOP 応答などです。

サーバー側：

- send_reply インターセプトポイントでこの属性がとる値は SUCCESSFUL だけです。
- send_exception インターセプトポイントでこの属性がとる値は SYSTEM_EXCEPTION または USER_EXCEPTION です。
- send_other インターセプトポイントでこの属性がとる値は，SUCCESSFUL または LOCATION_FORWARD のいずれかです。SUCCESSFUL は，非同期要求の戻りが成功したことを表します。LOCATION_FORWARD は，返った応答の状態が LOCATION_FORWARD であったことを表します。

```
virtual CORBA::Object_ptr forward_reference() = 0;
```

reply_status() が LOCATION_FORWARD を返すと，このメソッドは要求が転送されるオブジェクトを返します。送信した要求が実際に生成されるかどうかは不確定です。

```
virtual CORBA::Any* get_slot(CORBA::ULong _id) = 0;
```

要求のスコープ内にある PortableInterceptor::Current の指定スロットから CORBA::Any の形式でデータを返します。

指定されたスロットが設定されていない場合は，TCKind 値が tk_null のタイプコードを保持する CORBA::Any が返されます。

割り当てたスロットを ID が定義していない場合，InvalidSlot 例外が生成されます。

スロットと PortableInterceptor::Current の詳細については，216 ページの「Current」を参照してください。

パラメータ	説明
_id	返されるスロットの SlotId。

```
virtual IOP::ServiceContext* get_request_service_context(CORBA::ULong _id) = 0;
```

要求に関連付けられた指定 ID でサービスコンテキストのコピーを返します。

要求のサービスコンテキストに，その ID のエントリがない場合は，標準マイナーコード 26 の BAD_PARAM 例外が生成されます。

パラメータ	説明
_id	返されるスロットの IOP::ServiceContext。

```
virtual IOP::ServiceContext* get_reply_service_context(CORBA::ULong _id) = 0;
```

応答に関連付けられた指定 ID でサービスコンテキストのコピーを返します。

要求のサービスコンテキストに，その ID のエントリがない場合は，標準マイナーコード 26 の BAD_PARAM 例外が生成されます。

パラメータ	説明
_id	返されるスロットの IOP::ServiceContext。

ServerRequestInfo

```
class PortableInterceptor::ServerRequestInfo : public virtual RequestInfo
```

このクラスは RequestInfo から派生したもので，サーバーインターセプトポイントに送られます。

ServerRequestInfo のメソッドのすべてが，すべてのインターセプトポイントで有効であるとは限りません。次の表は，属性とメソッドの有効性をまとめたものです。無効なメソッドにアクセスしようとすると，標準マイナーコード 14 の BAD_INV_ORDER が生成されます。

表 12.5 ServerRequestInfo

	receive_request_service_contexts	receive_request	send_reply	send_exception	send_other
request_id	はい	はい	はい	はい	はい
operation	はい	はい	はい	はい	はい
arguments	いいえ	はい[1]	はい	いいえ[2]	いいえ[2]
exception	いいえ	はい	はい	はい	はい

表 12.5 ServerRequestInfo（続き）

	receive_request_service_contexts	receive_request	send_reply	send_exception	send_other
contexts	いいえ	はい	はい	はい	はい
operation_context	いいえ	はい	はい	いいえ	いいえ
result	いいえ	いいえ	はい	いいえ	いいえ
response_expected	はい	はい	はい	はい	はい
sync_scope	はい	はい	はい	はい	はい
reply_status	いいえ	いいえ	はい	はい	はい
forward_reference	いいえ	いいえ	いいえ	いいえ	はい[2]
get_slot	はい	はい	はい	はい	はい
get_request_service_context	はい	はい	はい	はい	はい
get_reply_service_context	いいえ	いいえ	はい	はい	はい
sending_exception	いいえ	いいえ	いいえ	はい	いいえ
object_id	いいえ	はい	はい	はい[3]	はい[3]
adapter_id	いいえ	はい	はい	はい[3]	はい[3]
server_id	いいえ	はい	はい	はい	はい
orb_id	いいえ	はい	はい	はい	はい
adapter_name	いいえ	はい	はい	はい	はい
target_most_derived_interface	いいえ	はい	いいえ[4]	いいえ[4]	いいえ[4]
get_server_policy	はい	はい	はい	はい	はい
set_slot	はい	はい	はい	はい	はい
target_is_a	いいえ	はい	いいえ[4]	いいえ[4]	いいえ[4]
add_reply_service_context	はい	はい	はい	はい	

[1] ServerRequestInfo が receive_request() に渡される場合は，in，inout，out の各引数に入力があります。ただし，有効なのは in 引数と inout 引数です。
[2] reply_status() が LOCATION_FORWARD を返さない場合，この属性にアクセスすると標準マイナーコード 14 で BAD_INV_ORDER が生成されます。
[3] サーバントロケータでロケーション転送を行った場合や，または例外が生成された場合，このインターセプトポイントではこの属性やメソッドを利用できません。標準マイナーコード 1 の NO_RESOURCES 例外が生成されます。
[4] このインターセプトポイントではこのメソッドを使用できません。ターゲットオブジェクトのサーバントにアクセスしないと必要な情報が得られず，ターゲットオブジェクトのサーバントは ORB では使用できないからです。たとえば，オブジェクトのアダプタが ServantLocator を使用する POA の場合，ORB は ServantLocator::postinvoke() の呼び出し後，インターセプトポイントを呼び出します。

インクルードファイル

このクラスを使用する場合は，PortableInterceptor_c.hh ファイルをインクルードする必要があります。

ServerRequestInfo のメソッド

```
virtual CORBA::Any* sending_exception() = 0;
```

クライアントに返す例外を含むデータを CORBA::Any 形式で返します。

例外が CORBA::Any に挿入できないユーザー例外の場合（未知の例外の場合や TypeCode が得られないバインディングの場合など），この属性は標準マイナーコード 1 のシステム例外 UNKNOWN を含む CORBA::Any になります。

```
virtual char* server_id() = 0;
```

ORB の作成時に -ORBServerId 引数で ORB::init 呼び出しに渡された値を返します。

```
virtual char* orb_id() = 0;
```

ORB::init() 呼び出しに渡された値を返します。

Java では，現在のテンプレートを作成したオブジェクトアダプタを含む ORB を作成した ORB.init 呼び出しの -ORBid 引数で対応します。同じサーバーの複数の ORB::init() 呼び出しで同じ ORBid を使用した場合に何が発生するかは，この時点では未確定です。

```
virtual CORBA::StringSequence* adapter_name() = 0;
```

起動オブジェクトの要求をサービスするオブジェクトアダプタの名前を CORBA::StringSequence 形式で返します。POA の場合，adapter_name は，ルート POA から要求をサービスする POA までの名前シーケンスです。このシーケンスでルート POA は指名しません。

```
virtual CORBA::OctetSequence* object_id() = 0;
```

オペレーション起動のターゲットを記述する不透過 object_id を CORBA::OctetSequence 形式で返します。

```
virtual CORBA::OctetSequence* adapter_id() = 0;
```

オブジェクトアダプタの不透過識別子を CORBA::OctetSequence 形式で返します。

```
virtual char* target_most_derived_interface() = 0;
```

サーバントの最下位派生インターフェースの RepositoryID を返します。

```
virtual CORBA::Policy_ptr get_server_policy(CORBA::ULong _type)
  = 0;
```

現在のオペレーションに有効なポリシーを指定のポリシー型で返します。返される CORBA::Policy オブジェクトは register_policy_factory() で型を登録したオブジェクトです。

指定した型のポリシーが register_policy_factory で登録されていない場合，このメソッドを実行すると，標準マイナーコード 3 の INV_POLICY 例外が生成されます。

パラメータ	説明
_type	返されるポリシーを指定する CORBA::PolicyType。

```
virtual void set_slot(CORBA::ULong _id, const CORBA::Any& _data) = 0;
```

要求のスコープに，インターセプタで PortableInterceptor::Current にスロットを設定するためのメソッドです。データがすでにスロットにある場合，古いデータは上書きされます。

割り当てたスロットを ID が定義していない場合，InvalidSlot 例外が生成されます。

スロットと PortableInterceptor::Current の詳細については，216 ページの「Current」を参照してください。

パラメータ	説明
_id	スロットの SlotId。
_data	CORBA::Any 形式でスロットに保存するデータ。

```
virtual CORBA::Boolean target_is_a(const char* _id) = 0;
```

サーバントが指定された RepositoryId の場合は true，そうでない場合は false が返されます。

パラメータ	説明
_id	呼び出し側は，サーバントがこの CORBA::RepositoryId であるかどうかの確認を要求します。

```
virtual void add_reply_service_context(const IOP::ServiceContext&
_service_context,CORBA::Boolean _replace) = 0;
```

インターセプタでサービスコンテキストを要求に追加します。

サービスコンテキストの順序の宣言はありません。サービスコンテキストは，必ずしも追加された順序では実行されません。

パラメータ	説明
_service_context	応答に追加する IOP::ServiceContext。
_replace	指定された ID のサービスコンテキストがすでに存在する場合のこのメソッドの動作を示します。false の場合は，標準マイナーコード 15 で BAD_INV_ORDER が生成されます。true の場合，既存のサービスコンテキストが新しいサービスコンテキストに置き換えられます。

ServerRequestInterceptor

```
class PortableInterceptor::ServerRequestInterceptor : public virtual Interceptor
```

ServerRequestInterceptor クラスは，ユーザー定義のサーバー側インターセプタを派生するために使用されます。ServerRequestInterceptor インスタンスは ORB に登録されます（223 ページの「ORBInitializer」を参照）。

インクルードファイル

このクラスを使用する場合は，PortableInterceptor_c.hh ファイルをインクルードする必要があります。

ServerRequestInterceptor のメソッド

```
virtual void receive_request_service_contexts(ServerRequestInfo_ptr _ri) = 0;
```

この receive_request_service_contexts() インターセプトポイントでインターセプタは入力要求からサービスコンテキスト情報を取得し，PortableInterceptor::Current のスロットに送信します。

このインターセプトポイントはサーバントマネージャの呼び出し前に呼び出されます。この段階ではまだオペレーションパラメータを利用できません。このインターセプトポイントは，ターゲット起動と同じスレッドで実行してもほかのスレッドで実行してもかまいません。

このインターセプトポイントでは，システム例外が生成される可能性があります。生成された場合，ほかのインターセプタの receive_request_service_contexts() インターセプトポイントは呼び出されません。フロースタック上のインターセプタがポップし，それぞれの send_exception() インターセプトポイントが呼び出されます。

このインターセプトポイントでは，ForwardRequest 例外が生成される可能性があります（218 ページの「ForwardRequest」を参照）。インターセプタによってこの例外が生成された場合，ほかのインターセプタの receive_request_service_contexts() メソッドは呼び出されません。フロースタック上のインターセプタがポップし，それぞれの send_other インターセプトポイントが呼び出されます。

パラメータ	説明
_ri	インターセプタで使用する ServerRequestInfo インスタンス。

```
virtual void receive_request(ServerRequestInfo_ptr _ri) = 0;
```

メソッドパラメータなどすべての情報を入手した後，receive_request() インターセプトポイントでは，インターセプタで要求情報の照会ができます。このインターセプトポイントは，ターゲット起動と同じスレッドで実行します。

DSI モデルでは，ユーザーコードが arguments() を呼び出したときに初めてパラメータを利用できるため，receive_request() は arguments() 内で呼び出されます。DSI モデルで arguments() が呼び出されない可能性もあります。arguments() を呼び出す前に，ターゲットが set_exception() を呼び出す場合です。ORB では，arguments() または set_exception() のいずれを介して receive_request() が必ず一度呼び出されます。set_exception() を介して呼び出された場合は，arguments() を要求すると，標準マイナーコード 1 の NO_RESOURCES が生成されます。

このインターセプトポイントでは，システム例外が生成される可能性があります。生成された場合，ほかのインターセプタの receive_request() メソッドは呼び出されません。フロースタック上のインターセプタがポップし，それぞれの send_exception インターセプトポイントが呼び出されます。

このインターセプトポイントでは，ForwardRequest 例外が生成される可能性があります (218 ページの「ForwardRequest」を参照)。インターセプタによってこの例外が生成された場合，ほかのインターセプタの receive_request() メソッドは呼び出されません。フロースタック上のインターセプタがポップし，それぞれの send_other() インターセプトポイントが呼び出されます。

パラメータ	説明
_ri	インターセプタで使用する ServerRequestInfo インスタンス。

```
virtual void send_reply(ServerRequestInfo_ptr _ri) = 0;
```

この send_reply() インターセプトポイントでは，インターセプタで応答情報を照会できます。また，ターゲットオペレーションを起動して応答がクライアントに返るまでの間に，応答サービスコンテキストを変更できます。このインターセプトポイントは，ターゲット起動と同じスレッドで実行します。

このインターセプトポイントでは，システム例外が生成される可能性があります。生成された場合，ほかのインターセプタの send_reply() インターセプトポイントは呼び出されません。フロースタック上のほかのインターセプタでは，それぞれの send_exception() インターセプトポイントが呼び出されます。

パラメータ	説明
_ri	インターセプタで使用する ServerRequestInfo インスタンス。

```
virtual void send_exception(ServerRequestInfo_ptr _ri) = 0;
```

send_exception() インターセプトポイントは例外の生成時に呼び出されます。これにより，インターセプタで例外情報を照会して，例外がクライアントに生成されるまでに応答サービスコンテキストを変更できます。このインターセプトポイントは，ターゲット起動と同じスレッドで実行します。

このインターセプトポイントでは，システム例外が生成される可能性があります。その場合は，フロースタックからポップされる後続のインターセプタが send_exception の呼び出しで受け取る例外が変更されます。クライアントには，前回インターセプタが生成した例外が生成されます。インターセプタによる例外の変更がなければ元の例外がクライアントに生成されます。

このインターセプトポイントでは，ForwardRequest 例外が生成される可能性があります (218 ページの「ForwardRequest」を参照)。インターセプタによってこの例外が生成された場合，ほかのインターセプタの send_exception() インターセプトポイントは呼び出されません。フロースタック上のほかのインターセプタでは，それぞれの send_other インターセプトポイントが呼び出されます。

```
virtual void send_other(ServerRequestInfo_ptr _ri) = 0;
```

この send_other() インターセプトポイントでインターセプタは，標準応答や例外以外の要求の結果に関する情報を照会します。たとえば，要求が再試行になることもあります（たとえば，LOCATION_FORWARD 状態の GIOP 応答を受け取った場合など）。このインターセプトポイントは，ターゲット起動と同じスレッドで実行します。

このインターセプトポイントでは，システム例外が生成される可能性があります。生成された場合，ほかのインターセプタの send_other() メソッドは呼び出されません。フロースタック上のほかのインターセプタでは，それぞれの send_exception インターセプトポイントが呼び出されます。

このインターセプトポイントでは，ForwardRequest 例外が生成される可能性があります。

第 13 章

5.x インターセプタとオブジェクトラッパーのインターフェースとクラス

ここでは，**インターセプタ**および**オブジェクトラッパー**とともに使用するインターフェースとクラスについて説明します。

詳細については，『VisiBroker for C++ 開発者ガイド』の「VisiBroker インターセプタの使い方」と「オブジェクトラッパーの使い方」を参照してください。

はじめに

5.x インターセプタは，VisiBroker バージョン 5.x で定義および実装されているインターセプタです。ポータブルインターセプタと同様に，5.x インターセプタも ORB の通常の実行の流れをインターセプトするメカニズムである VisiBroker ORB サービスを提供します。次の表は，5.x インターセプタの 3 つの形式です。

表 13.1　インターセプタの種類

インターセプタの種類	説明
クライアントインターセプタ	システムレベルのインターセプタです。クライアント側 ORB の処理中に，トランザクションやセキュリティなどの ORB サービスをフックするために使用されます。
サーバーインターセプタ	システムレベルのインターセプタです。サーバー側 ORB の処理中に，トランザクションやセキュリティなどの ORB サービスをフックするために使用されます。
オブジェクトラッパー	ユーザーレベルのインターセプタです。スタブやスケルトンへの呼び出しをインターセプトする簡潔なメカニズムをユーザーに提供します。これにより，簡単なトレーシングやデータのキャッシングなどを可能になります。

InterceptorManagers

インターセプタは，インターセプタマネージャを介してインストールおよび管理されます。InterceptorManager インターフェースは，各インターセプタに固有のすべてのマネージャによって継承される共通のインターセプタマネージャです。InterceptorManager 型は各インターセプタの型に関連付けられています。InterceptorManager は特定の種類のインターセプタ群をリストまたはチェインとして保持し，それらすべてのインターセプタが同じスコープを持ち，同時に起動されます。そのため，POA やオブジェクトのスコープを持つインターセプタがスコープごとの InterceptorManager を持つのと同様に，POALifeCycle や Bind のようなグローバルインターセプタにはグローバルな InterceptorManagers を持ちます。スコープにはグローバル，POA，またはオブジェクトがありますが，各スコープは，複数の型のインターセプタを保持することがあります。特定のインターセプタに対する正しい種類のマネージャを取得するには，InterceptorManagerControl を使用します。

グローバルインターセプタには，ローカライズされたインターセプタをインストールするために，追加のインターセプタマネージャを渡すことができます。たとえば，POA ごとのインターセプタは POAInterceptorManager を使用します。

グローバルインターセプタマネージャ InterceptorManager のインスタンスを取得するには，ORB.resolve_initial_references を呼び出し，引数として文字列 String InterceptorManager を渡します。この値は，ORB が管理モードにあるとき，つまり ORB の初期化中にだけ使用可能です。この値は，POALifeCycle インターセプタや Bind インターセプタなどのグローバルインターセプタをインストールするためにだけ使用できます。

POA インターセプタマネージャは POA 単位のマネージャであり，POALifeCycleInterceptors の create の呼び出し中にだけ使用できます。POALifeCycleInterceptors は，create の呼び出し中に，サーバー側のほかのすべてのインターセプタをセットアップします。Bind インターセプタマネージャはオブジェクト単位のマネージャであり，Bind インターセプタの bind_succeeded( ) 呼び出し中にだけ使用できます。Bind インターセプタは，bind_succeeded 呼び出し中に ClientRequest インターセプタをセットアップします。

IOR テンプレート

インターセプタを使用する以外に，POALifeCycleInterceptor::create( ) の呼び出し中に POAIntercptorManager インターフェースで，インターオペラブルオブジェクトリファレンス（Interoperable Object Reference，IOR）テンプレートを直接変更することもできます。IOR テンプレートは，GIOP::ProfileBodyValues 値のオブジェクトキーがすべて不完全で，type_id が設定されていない完全な IOR 値です。POA は，type_id を設定し，テンプレートのオブジェクトキーを記述してから，IORCreationInterceptors を呼び出します。

InterceptorManager

class Interceptor::**InterceptorManager**

これは，ほかのすべてのインターセプタマネージャの派生元になるベースクラスです。インターセプタマネージャは，システムにインターセプタをインストールしたり，それを削除するための管理に使用されるインターフェースです。

InterceptorManagerControl

class Interceptor::**InterceptorManagerControl** public CORBA::PseudoObject

このクラスは,関連するインターセプタマネージャのセットを制御する役割を持ちます。このクラスは，使用可能なすべてのマネージャを保持し，各マネージャは，管理するイ

ンターセプタの型に対応する文字列によって識別されます。スコープごとに 1 つの InterceptorManagerControl があります。

インクルードファイル

この構造体を使用する場合は，**interceptor_c.hh** ファイルをインクルードする必要があります。

InterceptorManagerControl のメソッド

```
InterceptorManager_ptr get_manager(const char name);
```

InterceptorManager のインスタンスを返します。InterceptorManager のインスタンスは，そのマネージャを識別する文字列を返します。

パラメータ	説明
name	インターセプタの名前。

BindInterceptor

```
class Interceptor::BindInterceptor public VISPseudoInterface
```

このクラスを使用すると，独自のインターセプタを派生して，クライアントまたはサーバーアプリケーションのバインドイベントとリバインドイベントを処理できます。Bind インターセプタは，バインドの前後にクライアント側で呼び出されるグローバルインターセプタです。

バインド中に例外が生成された場合，チェイン内の残りのインターセプタは呼び出されず，チェインは，すでに呼び出されたインターセプタを残して切り捨てられます。bind_succeeded または bind_failed に生成される例外は無視されます。

インクルードファイル

この構造体を使用する場合は，**interceptor_c.hh** ファイルをインクルードする必要があります。

BindInterceptor のメソッド

```
virtual IOP::IORValue_ptr bind(IOP::IORValue_ptr ior, CORBA::Object_ptr obj,CORBA::Boolean
rebind, VISClosure& closure);
```

すべての ORB bind オペレーションの間に呼び出されます。

パラメータ	説明
ior	クライアントがバインドするサーバーオブジェクトのインターオペラブルオブジェクトリファレンス（Interoperable Object Reference, IOR）。
obj	サーバーにバインドを試みたクライアントオブジェクト。このオブジェクトは，この時点では正しく初期化されません。したがって，このオブジェクトのオペレーション呼び出してはなりません。ただし，これをデータ構造に保存して，バインドの完了後に使用することができます。
rebind	そのサーバーにリバインドを試みます。bind() が失敗した後，現在の Quality of Service により，リバインドが試みられます。

パラメータ	説明
closure	このバインド処理に対する新しいClosureオブジェクト。このClosureオブジェクトは，対応するbind_failureまたはbind_succeededの呼び出しの中で使用されます。
return	新しいIORを使用してバインド処理を継続する場合は，その新しいIORを返します。そうでない場合はnull値を返し，バインドは元のIORを使用して継続されます。 パラメータとして渡されたIORと同じIORを返すことは正しくなく，その場合は，バインド時に例外が生成されます。

```
virtual IOP::IORValue_ptr bind_failed(IOP::IORValue_ptr ior, CORBA::Object_ptr
obj,VISClosure& closure);
```

バインド処理が失敗した場合に呼び出されます。

パラメータ	説明
ior	バインド処理が失敗したサーバーオブジェクトのIOR。
obj	サーバーにバインドを試みたクライアントオブジェクト。
closure	前のbind呼び出し内で指定されたClosureオブジェクト。
return	新しいIORを使用してリバインドを試みる場合は，その新しいIORを返します。そうでない場合はnullを返し，リバインドは試みられません。

```
virtual void bind_succeeded(IOP::IORValue_ptr ior, CORBA::Object_ptr obj,CORBA::Long
profileIndex, InterceptorManagerControl_ptr interceptorControl,VISClosure& closure);
```

バインド操作が成功に終わると，このメソッドが呼び出されます。

パラメータ	説明
ior	バインド処理が成功したサーバーオブジェクトのIOR。
obj	サーバーにバインドを試みたクライアントオブジェクト。
profileIndex	接続プロトコルを識別します。
interceptorControl	このマネージャは，マネージャの型のリストを提供します。
closure	前のbind呼び出し内で指定されたClosureオブジェクト。

BindInterceptorManager

```
class Interceptor::BindInterceptorManager public InterceptorManager, public
VISPseudoInterface
```

このクラスは，すべてのグローバルバインドインターセプタを管理します。このクラスにはpublicメソッドが1つだけあり，これを使用してインターセプタを登録できます。

BindInterceptorManagerは常にORB_init( )で使用する必要があります。ORBが初期化された後では，BindInterceptorManager の効果はありません。したがって，BindInterceptorManagerは，VISinitを継承するローダークラスのコンテキスト内でのみ使用する必要があります。

InterceptorManagerControlからBindInterceptorManagerを取得するには，識別文字列Bindとともに InterceptorManagerControl::get_manager( )を使用します。

インクルードファイル

この構造体を使用する場合は，**interceptor_c.hh** ファイルをインクルードする必要があります。

BindInterceptorManager のメソッド

```
void add (BindInterceptor _ptr interceptor)
```

バインド時に起動されるインターセプタのリストに BindInterceptor を追加するために使用されます。

ClientRequestInterceptor

```
class Interceptor::ClientRequestInterceptor public VISPseudoInterface
```

このクラスは，独自のクライアント側インターセプタを派生するために使用されます。ClientRequestInterceptor は，BindInterceptor の bind_succeeded 呼び出しの間にインストールされ，接続が続いている間はアクティブなままになります。派生先のクラスで定義されたメソッドが ORB によって呼び出されるのは，処理要求の準備中か送信中，応答メッセージの受信中，または例外が生成されたときです。

インクルードファイル

この構造体を使用する場合は，**interceptor_c.hh** ファイルをインクルードする必要があります。

ClientRequestInterceptor のメソッド

```
virtual void preinvoke_premarshal (CORBA::Object_ptr target, const char* operation,
IOP::ServiceContextList& service_contexts,VisClosure& closure);
```

要求があるたび，要求がマーシャリングされる前に ORB によって呼び出されます。このインターセプタから例外が生成されると，その要求はただちに完了されます。その場合，チェインは，すでに起動されているインターセプタだけに短縮され，要求は送信されません。チェイン内の残りのインターセプタに対しては，exception_occurred() が呼び出されます。

パラメータ	説明
target	サーバーにバインドを試みたクライアントオブジェクト。
operation	呼び出されるオペレーションの名前。
service_context	ORB が割り当てるサービス。これらのサービスは，OMG に登録しているタグによって識別されます。
closure	前の bind 呼び出し内で指定された Closure オブジェクト。

```
virtual void preinvoke_postmarshal(CORBA::Object_ptr target, CORBA_MarshallOutBuffet&
payload, VISClosure& closure);
```

すべての要求がマーシャリングされた後で，その要求が送信される前に呼び出されます。

このメソッド内で例外が生成された場合の動作は，次のとおりです。

- チェイン内の残りのインターセプタは呼び出されません。
- 要求はサーバーに送信されません。
- インターセプタチェイン全体に対して exception_occurred() が呼び出されます。

パラメータ	説明
target	サーバーにバインドを試みたクライアントオブジェクト。
payload	マーシャリングされたバッファ。
closure	前の bind 呼び出し内で指定された Closure オブジェクト。

```
virtual void postinvoke(CORBA::Object_ptr target, const IOP::ServiceContextList&
Service_contexts, CORBA_MarshallInBuffet& payload, CORBA::Environment_ptr env, VISClosure&
closure);
```

要求が正常に終了した後，または例外が生成されることによって呼び出されます。このメソッドは，ServantLocator が呼び出された後で呼び出されます。チェイン内に例外を生成するインターセプタがあった場合，そのインターセプタは exceptionoccurred() も呼び出し，そのチェイン内の残りすべてのインターセプタは，postinvoke() を呼び出さずに exception() を呼び出します。

すでに引数に例外を記述している双方向呼び出しがある場合でも，CORBA::Environment パラメータは，この例外を反映して変更されます。

パラメータ	説明
target	サーバーにバインドを試みたクライアントオブジェクト。
service_context	サーバーにバインドを試みたクライアントオブジェクト。一方向呼び出しおよび例外発生時のサービスコンテキストの長さは 0 です。
payload	マーシャリングされたバッファ。
env	生成された例外に関する情報を保持します。
closure	前の bind 呼び出し内で指定された Closure オブジェクト。

```
virtual void exception_occurred(CORBA::Object_ptr target,CORBA::Environment_ptr env,
VISClosure& closure);
```

このインターセプタが呼び出される前に例外が生成された場合，ORB によって呼び出されます。呼び出し後に生成されたすべての例外は，postinvoke メソッドの environment パラメータに集められます。

パラメータ	説明
target	サーバーにバインドを試みたクライアントオブジェクト。
env	生成された例外に関する情報を保持します。
closure	前の bind 呼び出し内で指定された Closure オブジェクト。

ClientRequestInterceptorManager

```
class Interceptor::ClientRequestInterceptorManager:public InterceptorManager, public
VISPseudoInterface
```

現在のオブジェクトに対する ClientRequestInterceptors のチェインを保持するクラスです。

ClientRequestInterceptorManager は，BindInterceptor::bind_succeeded( ) メソッドの内部で，bind_succeeded( ) の引数として渡された InteceptorManagerControl によって設定されるスコープにおいて使用する必要があります。

インクルードファイル

この構造体を使用する場合は，**interceptor_c.hh** ファイルをインクルードする必要があります。

ClientRequestInterceptorManager のメソッド

```
virtual void add (ClientRequestInterceptor_ptr interceptor);
```

ローカルチェインに ClientRequestInterceptor を追加するために呼び出されます。

```
virtual void remove (ClientRequestInterceptor_ptr interceptor);
```

ClientRequestInterceptorManager を削除します。

POALifeCycle インターセプタ

```
class InterceptorManager::POALifeCycletInterceptor public VISPseudoInterface
```

POALifeCycleInterceptor は，POA が作成または廃棄されるたびに呼び出されるグローバルインターセプタです。その他のサーバー側インターセプタは，グローバルインターセプタ，または特定の POA のインターセプタとしてインストールされます。POALifeCycleInterceptor をインストールするには，POALifeCycleInterceptorManager インターフェースを使用します。詳細については，243 ページの「POALifeCycleInterceptorManager」を参照してください。POA が作成および廃棄されるときに，POALifeCycleInterceptor が呼び出されます。

インクルードファイル

このクラスを使用する場合は，**PortableServerExt_c.hh** ファイルをインクルードする必要があります。

POALifeCycleInterceptor のメソッド

```
virtual void create(PortableServer::POA_ptr poa, CORBA::PolicyList& policiesIOP::IORValue*&
iorTemplate,interceptor::InterceptorManagerControl_ptr poaAdmin);
```

新しい POA が create_POA 呼び出しを介して明示的に作成されるか，AdapterActivator を介して作成されたときに呼び出されます。AdapterActivator を使用した場合，このインターセプタが呼び出されるのは，unknown_adapter メソッドが成功して，AdapterActivator から戻った後だけです。create メソッドは，最近作成された POA への参照として，また，その POA インスタンスの POAInterceptorManager への参照として渡されます。

パラメータ	説明
poa	作成される現在の POA に関連付けられた ID。
policies	作成される POA のポリシー。
iorTemplate	IOR テンプレートは，GIOP::ProfileBody 値のオブジェクトキーがすべて不完全で，type_id が設定されていない完全な IOR 値です。
poaAdmin	作成される POA のコントロール。詳細については，238 ページの「InterceptorManagerControl」を参照してください。

```
virtual void destroy(PortalServer::POA_ptr poa);
```

POA が廃棄され，そのすべてのオブジェクトが霊化される前に呼び出されます。再び同じ名前の POA を作成する create が呼び出される前に，すべてのインターセプタに対して，必ず destroy が呼び出されます。destroy オペレーションが例外を生成しても，それは無視され，残りのインターセプタへの呼び出しが続けられます。

パラメータ	説明
poa	廃棄されるポータブルオブジェクトアダプタ（POA）。

POALifeCycleInterceptorManager

```
class InterceptorExt::POALifeCycleInterceptorManager public interceptor::InterceptorManager,
public VISPseudoInterface
```

このクラスは，すべての POALifeCycle グローバルインターセプタを管理します。1 つの ORB で定義される POALifeCycleInterceptorManager のインスタンスは 1 つです。

このインターフェースのスコープは，ORB 単位でグローバルです。このクラスは，ORB_init( ) の間だけアクティブです。

インクルードファイル

このクラスを使用する場合は，**PortableServerExt_c.hh** ファイルをインクルードする必要があります。

POALifeCycleInterceptorManager のメソッド

```
virtual void add(POALifeCycleInterceptor_ptr interceptor);
```

POALifeCycle インターセプタのグローバルチェインに POALifeCycleInterceptor を追加するために呼び出されます。

パラメータ	説明
interceptor	追加するインターセプタ。

ActiveObjectLifeCycleInterceptor

```
class PortableServerExt::ActiveObjectLifeCycleInterceptor public VISPseudoInterface
```

オブジェクトを追加してからアクティブオブジェクトマップから削除すると，ActiveObjectLifeCycleInterceptor インターセプタが呼び出されます。このインターフェースは，POA が RETAIN ポリシーを持つ場合にだけ使用されます。このクラスは POA のスコープを持つインターセプタであり，POA が作成されるときに POALifeCycleInterceptor によってインストールされます。

インクルードファイル

このクラスを使用する場合は，**PortableServerExt_c.hh** ファイルをインクルードする必要があります。

ActiveObjectLifeCycleInterceptor のメソッド

```
virtual void create(const PortableServer::ObjectId& oid,PortableServer::ServantBase*
servant, PortableServer::POA_ptr adapter);
```

明示的または暗黙的なアクティブ化を介して，オブジェクトがアクティブオブジェクトマップに追加された後で呼び出されます。直接 API を使用して，または ServantActivator を使用してのどちらの場合もあります。新しいアクティブオブジェクトのオブジェクトリファレンスと POA がパラメータとして渡されます。

パラメータ	説明
oid	現在アクティブ化しているオブジェクトのオブジェクト ID。
servant	関連するサーバント。
adapter	作成または廃棄されるポータブルオブジェクトアダプタ（POA）。

```
virtual void destroy(const PortableServer::ObjectId& oid,PortableServer::ServantBase*
servant, PortableServer::POA_ptr adapter);
```

オブジェクトが非アクティブ化され，霊化された後で呼び出されます。そのオブジェクトのオブジェクトリファレンスと POA がパラメータとして渡されます。

パラメータ	説明
oid	現在アクティブ化しているオブジェクトのオブジェクト ID。
servant	関連するサーバント。
adapter	作成または廃棄されるポータブルオブジェクトアダプタ（POA）。

ActiveObjectLifeCycleInterceptorManager

```
class PortableServerExt::ActiveObjectLifeCycleInterceptorManager public
interceptor::InterceptorManager, public VISPseudoInterface
```

このクラスは，スコープに登録されているすべての ActiveObjectLifeCycleInterceptors を管理します。各 POA が 1 つの ActiveObjectLifeCycleInterceptorManager を持っています。

インクルードファイル

このクラスを使用する場合は，**PortableServer_c.hh** ファイルをインクルードする必要があります。

ActiveObjectLifeCycleInterceptorManager のメソッド

```
virtual void add(ActiveObjectLifeCycleInterceptor interceptor_ptr interceptor);
```

ローカルチェインに ActiveObjectLifeCycleInterceptor を追加するために呼び出されます。

ServerRequestInterceptor

```
class Interceptor::ServerRequestInterceptor public VISPseudoInterface
```

ServerRequestInterceptor インターフェースは POA ごとのインターセプタであり，POA の作成時，POALifeCycleInterceptor によってインストールされます。このクラスは，アクセスコントロールを実行したり，サービスコンテキストを検査および挿入したり，要求の応答状態を変更するために使用されます。

インクルードファイル

この構造体を使用する場合は，**interceptor_c.hh** ファイルをインクルードする必要があります。

ServerRequestInterceptor のメソッド

```
virtual void preinvoke(CORBA::Object_ptr _target, const char* operation, const
IOP::ServiceContextList& service_contexts, CORBA::MarshalInBuffer& payload,
VISClosure& closure) raises (ForwardRequestException);
```

要求ごとに，要求がアンマーシャリングされる前に ORB によって呼び出されます。このインターセプタから例外が生成されると，その要求はただちに完了されます。

ServantLocators が呼び出される前に呼び出されます。そのため，そのサーバントが使用できなくなる可能性があります。

パラメータ	説明
target	サーバーにバインドを試みたクライアントオブジェクト。
operation	呼び出されるオペレーションの名前を識別します。
service_contexts	ORB によって割り当てられるサービスを識別します。これらのサービスは OMG に登録されています。
payload	マーシャリングされたバッファ。
closure	あるインターセプタのメソッドによって保存され，後に別のインターセプタのメソッドによって取得されるデータを保持します。

```
virtual void postinvoke_premarshal(CORBA::Object_ptr target,IOP::ServiceContextList&
ServiceContextList,CORBA::Environment_ptr env,VISClosure& closure);
```

サーバントへのアップコールの後で，応答をマーシャリングする前に呼び出されます。このとき生成された例外は，チェインへの割り込みによって処理されます。この場合，要求はサーバーに送信されず，チェイン内のすべてのインターセプタに対して，exceptionoccurred() が呼び出されます。

パラメータ	説明
target	サーバーにバインドを試みたクライアントオブジェクト。
ServiceContextList	ORB によって割り当てられるサービスを識別します。これらのサービスは OMG に登録されています。
env	生成された例外に関する情報を保持します。
closure	あるインターセプタのメソッドによって保存され，後に別のインターセプタのメソッドによって取得されるデータを保持します。

```
virtual void postinvoke_postmarshal(CORBA::Object_ptr _target,CORBA::MarshalOutBuffer&
_payload, VISClosure& _closure);
```

応答をマーシャリングした後で，応答をクライアントに送信する前に呼び出されます。ここで生成される例外は無視されます。チェイン内のすべてのインターセプタが呼び出されます。

パラメータ	説明
target	アプリケーションがバインドを試みたオブジェクト。
payload	マーシャリングされたバッファ。
closure	あるインターセプタのメソッドによって保存され，後に別のインターセプタのメソッドによって取得されるデータを保持します。

```
virtual void exception_occurred(CORBA::Object_ptr _target,CORBA::Environment_ptr _env,
VISClosure& _closure);
```

prepare_reply インターセプタの 1 つで例外が生成され，チェイン内の残りのすべてのインターセプタに対して exceptionoccurred が呼び出されるときに，ORB によって呼び出されます。この呼び出しの間に生成された例外は，環境内の既存の例外を置き換えます。

パラメータ	説明
target	サーバーにバインドを試みたクライアントオブジェクト。
env	生成された例外に関する情報を保持します。
closure	あるインターセプタのメソッドによって保存され，後に別のインターセプタのメソッドによって取得されるデータを保持します。

ServerRequestInterceptorManager

```
class Interceptor::ServerRequestInterceptorManager public InterceptorManager, public
VISPseudoInterface
```

このクラスは，スコープに登録されているすべての ServerRequestInterceptors を管理します。各 POA が 1 つの ServerRequestInterceptorManager を持っています。

インクルードファイル

この構造体を使用する場合は，**interceptor_c.hh** ファイルをインクルードする必要があります。

ServerRequestInterceptorManager メソッド

```
virtual void add(ServerRequestInterceptor_ptr interceptor);
```

ローカルチェインに ServerRequestInterceptor を追加するために呼び出されます。

IORCreationInterceptor

```
class PortableServerExt::IORInterceptor public VISPseudoInterface
```

IORCreationInterceptor は POA ごとのインターセプタであり，POA の作成時，POALifeCycleInterceptor によってインストールされます。このインターセプタは，プロファイルまたはコンポーネントを追加することにより，IOR を変更するために使用されます。このクラスは，トランザクションやファイアウォールなどのサービスをサポートするためによく使用されます。

このようなインターセプタは，開発時に名前と ID がわからない特定のクラスで IOR テンプレートを自動的に変更するために使用されます。たとえば，トランザクションやファイアウォールなどのサービスで使用されます。

メモ　POA によって作成されたすべての IOR を変更するには，その POA の IORTemplate を変更します。変更は，既存の IOR ではなく，新しく作成された IOR にだけ適用されます。

IOR を大きく変更することはお勧めできません。

インクルードファイル

このクラスを使用する場合は，**PortableServerExt_c.hh** ファイルをインクルードする必要があります。

IORInterceptor のメソッド

```
virtual void create(PortableServer::POA poa, IOP::IORValue*& ior);
```

POA がオブジェクトリファレンスを作成する必要があるときに必ず呼び出されます。このメソッドは，そのオブジェクトリファレンスに対する POA，OID，および IORValue を引数として受け取ります。インターセプタでは，新しいプロファイルまた

はコンポーネントを追加したり，既存のプロファイルまたはコンポーネントを変更することにより，IORValue を変更できます。

パラメータ	説明
poa	作成される現在の POA に関連付けられた ID。
ior	クライアントがバインドするサーバーオブジェクトの IOR。

IORCreationInterceptorManager

```
class PortableServerExt::IORCreationInterceptorManager public
interceptor::InterceptorManager, public VISPseudoInterface
```

このクラスは，IOR インターセプタをローカルチェインに追加するために使用されます。各 POA が 1 つの IORInterceptorManager を持っています。

インクルードファイル

このクラスを使用する場合は，**PortableServerExt_c.hh** ファイルをインクルードする必要があります。

IORCreationInterceptorManager のメソッド

```
virtual void add(IORCreationInterceptor_ptr _interceptor);
```

ローカルチェインに IORInterceptor を追加するために呼び出されます。

Closure

```
public interface Closure extends Object
```

インターセプタによる呼び出しのうち，あるシーケンスの開始時に，ORB によって Closure オブジェクトが作成されます。この特定のシーケンス内のすべての呼び出しに対して，同じ Closure オブジェクトが使用されます。Closure オブジェクトには，java.lang.Object 型の 1 つの public データフィールド object があります。このデータフィールドは，状態情報を保存するためにインターセプタによって設定されます。Closure オブジェクトが作成されるシーケンスは，インターセプタのタイプによって異なります。

このコードサンプルは，Closure クラスを示しています。

```
class Closure {
        java.lang.Object object;
};
```

ExtendedClosure

```
public interface ExtendedClosure extends Closure (
   public RequestInfo reqInfo;
   public InputStream payload;
}
```

このインターフェースは，Closure の派生クラスであり，読み取り専用属性の RequestInfo を保持します。

このコードサンプルは，RequestInfo IDL クラスを示しています。

```
struct RequestInfo {
   boolean response_expected;
   unsigned long request_id;
};
```

ServerRequestInterceptor と ClientRequestInterceptor に渡された Closure オブジェクトをサブクラスの ExtendedClosure にキャストできます。ExtendedClosure を使用して，RequestInfo を抽出できます。ExtendedClosure は，request_id と response_expected を抽出できます。request_id は，要求に割り当てられている一意な ID です。response_expected フラグは，その要求が一方向呼び出しであることを示しています。

```
int my_response_expected = ((ExtendedClosure)closure).reqInfo.response_expected;
int my_request_id = ((ExtendedClosure)closure).reqInfo.request_id;
```

詳細については，examples/interceptor/client_server にあるサンプルを参照してください。

VISClosure

struct **VISClosure**

この構造体は，インターセプタのメソッドが複数回呼び出される場合に，それらの間で共有されるデータを格納するために使用されます。格納されるデータは型なしであり，処理要求，バインド要求，または検索要求に関連する状態情報を表します。この構造体は，VISClosureData クラスとともに使用されます。

インクルードファイル

このクラスを使用する場合は，**vinter.h** ファイルをインクルードする必要があります。

VISClosure のメンバー

CORBA::**ULong id**

このデータメンバーを使用すると，複数の VISClosure オブジェクトを使用している場合に，このオブジェクトを一意に識別できます。

void ***data**

このデータメンバーは，インターセプタメソッドによって格納またはアクセスされる型なしのデータへのポインタです。

VISClosureData ***managedData**

このデータメンバーは，実際のデータを表す VISClosureData クラスへのポインタです。管理しているデータをこの型にキャストします。

VISClosureData

class **VISClosureData**

このクラスは，インターセプタメソッドの複数回の呼び出しで共有され，管理されるデータを表します。

VISClosureData のメソッド

```
virtual void _VisClosureData();
```

デフォルトのデストラクタです。

```
virtual void _release();
```

このオブジェクトを解放し，参照カウントをデクリメントします。参照カウントが 0 になった場合，そのオブジェクトは削除されます。

ChainUntypedObjectWrapperFactory

```
class VISObjectWrapper::ChainUntypedObjectWrapperFactory:public UntypedObjectWrapperFactory
```

このインターフェースは UntypedObjectWrapperFactory オブジェクトを追加または削除するために，クライアントまたはサーバーアプリケーションによって使用されます。UntypedObjectWrapperFactory は，クライアントアプリケーションがバインドする各オブジェクトの UntypedObjectWrapper，またはサーバーアプリケーションによって作成される各オブジェクトインプリメンテーションの UntypedObjectWrapper を作成するために使用されます。

オブジェクトラッパーの使用方法については，『Borland VisiBroker 開発者ガイド』のオブジェクトラッパーのセクションを参照してください。

インクルードファイル

このクラスを使用する場合は，**vobjwrap.h** ファイルをインクルードする必要があります。

ChainUntypedObjectWrapperFactory のメソッド

```
void add(UntypedObjectWrapperFactory_ptr factory,Location loc);
```

指定された型なしオブジェクトラッパーファクトリをクライアント，サーバー，または共用アプリケーションに追加します。

アプリケーションがクライアントアプリケーションとサーバーアプリケーションの両方として動作する共用アプリケーションである場合は，型なしオブジェクトラッパーファクトリをインストールできます。その場合は，バインドされているオブジェクトに対する呼び出しとオブジェクトインプリメンテーションが受け取る処理要求に対する呼び出しの両方について，つまり，アプリケーションのクライアント部分とサーバー部分の両方について，ラッパーのメソッドが呼び出されます。

メモ クライアント側では，オブジェクトがバインドされる前に，型なしオブジェクトラッパーファクトリを定義する必要があります。サーバー側では，オブジェクトインプリメンテーションに対する呼び出しを受け取る前に，型なしオブジェクトラッパーファクトリを定義する必要があります。

パラメータ	説明
factory	登録するファクトリへのポインタ。
loc	追加するファクトリの場所。次のいずれかの値です。VISObjectWrapper::ClientVISObjectWrapper::ServerVISObjectWrapper::Both

```
void remove(UntypedObjectWrapperFactory_ptr factory, Location loc);
```

指定された場所から，指定された型なしオブジェクトラッパーファクトリを削除します。

アプリケーションがクライアントとサーバーの両方として動作している場合は，クライアント側オブジェクト，サーバー側インプリメンテーション，またはその両方に対するオブジェクトラッパーファクトリを削除できます。

メモ クライアントから 1 つ以上のオブジェクトラッパーファクトリを削除しても，すでにそのクライアントによってバインドされているクラスのオブジェクトには影響しません。影響を受けるのは，それ以後にバインドされたオブジェクトだけです。

サーバーからオブジェクトラッパーファクトリを削除しても，すでにサービスされている要求を持つオブジェクトインプリメンテーションには影響しません。影響を受けるのは，それ以後に作成されたオブジェクトインプリメンテーションだけです。

パラメータ	説明
factory	登録するファクトリへのポインタ。
loc	削除するファクトリの場所。次のいずれかの値です。 VISObjectWrapper::Client VISObjectWrapper::Server VISObjectWrapper::Both

```
static CORBA::ULong count(Location loc);
```

この static メソッドは，指定された場所について，インストールされている型なしオブジェクトラッパーファクトリの数を返します。

パラメータ	説明
loc	ファクトリの場所。次のいずれかの値です。 VISObjectWrapper::Client VISObjectWrapper::Server VISObjectWrapper::Both

UntypedObjectWrapper

```
class VISObjectWrapper::UntypedObjectWrapper : public VISResource
```

クライアントアプリケーション, サーバーアプリケーション, または共用アプリケーションのための型なしオブジェクトラッパーを派生および実装するには，このクラスを使用します。このクラスから型なしオブジェクトラッパーを派生させる場合は，pre_method メソッドを定義します。pre_method メソッドは，要求がクライアントアプリケーションによって発行される前，または要求がサーバー側のオブジェクトインプリメンテーションによって処理される前に呼び出されます。post_method メソッドは，処理要求がサーバー側のオブジェクトインプリメンテーションによって処理された後，またはクライアントアプリケーションが応答を受け取った後で呼び出されます。

さらに，型なしラッパーオブジェクト作成するためのファクトリクラスを派生させる必要があります。これは UntypedObjectWrapperFactory クラスから派生させます。詳細については，252 ページの「UntypedObjectWrapperFactory」を参照してください。

オブジェクトラッパーの使用方法については，『Borland VisiBroker 開発者ガイド』を参照してください。

インクルードファイル

このクラスを使用する場合は，**vobjwrap.h** ファイルをインクルードする必要があります。

UntypedObjectWrapper のメソッド

```
virtual void pre_method(const char* operation, CORBA::Object_ptr target, VISClosure&
closure);
```

処理要求がクライアント側から送信される前，または要求がサーバー側のオブジェクトインプリメンテーションによって処される前に呼び出されます。

パラメータ	説明
operation	要求されるオペレーションの名前。
target	要求の対象のオブジェクト。
closure	オブジェクトラッパーのメソッド間でデータの受け渡しに使用される Closure オブジェクト。

```
virtual void post_method(const char* operation, CORBA::Object_ptr target, CORBA::Environment&
env, VISClosure& closure);
```

処理要求がサーバー側のオブジェクトインプリメンテーションによって処理された後，または応答メッセージがクライアント側のスタブによって処理される前に呼び出されます。

パラメータ	説明
operation	要求されるオペレーションの名前。
target	要求の対象のオブジェクト。
env	要求の処理中に発生した例外を反映するために使用される Environment オブジェクト。
closure	オブジェクトラッパーのメソッド間でデータの受け渡しに使用される Closure オブジェクト。

UntypedObjectWrapperFactory

```
class VISObjectWrapper::UntypedObjectWrapperFactory
```

このインターフェースは，独自の型なしオブジェクトラッパーファクトリを派生させるために使用されます。このファクトリは，新しいオブジェクトがバインドされたり，オブジェクトインプリメンテーションがサービスを要求すると，アプリケーションの型なしオブジェクトラッパーのインスタンスを作成するために必ず使用されます。

インクルードファイル

このクラスを使用する場合は，**vobjwrap.h** ファイルをインクルードする必要があります。

UntypedObjectWrapperFactory のコンストラクタ

```
UntypedObjectWrapperFactory(Location loc, CORBA::Boolean doAdd=1);
```

指定された場所に型なしオブジェクトラッパーファクトリを作成し，デフォルトでは，それを ChainUntypedObjectWrapperFactory に登録します。アプリケーションがクライアントアプリケーションとサーバーアプリケーションの両方として動作する場合は，バインドされているオブジェクトに対する呼び出しとオブジェクトインプリメンテーションが受け取るオペレーション要求に対する呼び出しの両方でラッパーのメソッドが呼び出されるように，型なしオブジェクトラッパーファクトリをインストールできます。

デフォルトのパラメータを使用しない場合は，そのように doAdd を指定します。ただし，型なしオブジェクトラッパーを作成するには，ChainUntypedObjectWrapper::add を呼び出す必要があります。

パラメータ	説明
loc	追加するファクトリの場所。次のいずれかの値です。 VISObjectWrapper::Client VISObjectWrapper::Server VISObjectWrapper::Both
doAdd	ファクトリを登録するかどうかを指定するフラグ。

UntypedObjectWrapperFactory のメソッド

```
virtual UntypedObjectWrapper_ptr create(CORBA::Object_ptr target, Location loc);
```

独自の型の UntypedObjectWrapper のインスタンスを作成するために呼び出されます。このメソッドを実装すると，バインドされるオブジェクトまたはオブジェクトインプリメンテーションの型を検査して，そのオブジェクトにオブジェクトラッパーを作成する必要があるかどうかを判定できます。loc パラメータを使用して，クライアントオブジェクトまたはサーバーインプリメンテーションのどちらをラップするために create が呼び出されたのかを指定します。

パラメータ	説明
target	クライアントアプリケーションによって新しくバインドされ，型なしオブジェクトラッパーの作成対象となるオブジェクト。このメソッドがサーバー側で呼び出されている場合，このオブジェクトは，新しく作成されるオブジェクトインプリメンテーションを表します。
loc	追加するファクトリの場所。

第 14 章

QoS のインターフェースとクラス

ここでは，Quality of Service（QoS）API の VisiBroker for C++ によるインプリメンテーションについて説明します。ポリシーの作成方法については，「コアインターフェースとクラス」の第 3 章「コアインターフェースとクラス」を参照してください。

CORBA::PolicyManager

class CORBA::**PolicyManager**

このクラスは，VisiBroker ORB レベルでポリシーオーバーライドを設定したり，アクセスするために使用されます。VisiBroker ORB レベルで定義されたポリシーは，システムデフォルトを上書きします。マネージャスレッドに属するインスタンスは，resolve_initial_reference("PolicyManager") を使用して PolicyManager にナローイングすることでアクセスできます。

IDL 定義

```
module CORBA {
     interface PolicyManager {
      PolicyList get_policy_overrides(in PolicyTypeSeq ts);
      void set_policy_overrides(in PolicyList policies, in SetOverrideType set_add)
         raises (InvalidPolicies);
   };
};
```

メソッド

PolicyList get_policy_overrides (PolicyTypeSeq **ts**);

要求されたポリシータイプのすべてのポリシーのリストを返します。指定されたシーケンスが空（リストの長さが 0）の場合は，このスコープ内のすべてのポリシーが返されます。要求されたポリシータイプが対象の **PolicyManager** にまったく設定されていない場合は，空のシーケンスが返されます。

void set_policy_overrides (PolicyList **policies**,CORBA::SetOverrideType **set_add**)

指定されたポリシーオーバーライドのリストを使用して，現在のポリシーセットを更新します。PolicyManager からすべてのオーバーライドを削除するには，空のポリシーシーケンスと SET_OVERRIDE モードを使用して，set_policy_overrides を呼び出します。

このオペレーションを使用して上書きできるのは，クライアントエンドから呼び出すオペレーションに関連した一定のポリシーだけです。その他のポリシーを上書きしようとすると，CORBA::NO_PERMISSION 例外が生成されます。要求によって対象の PolicyManager のポリシーオーバーライドに矛盾が起こる場合，ポリシーは何も変更または追加されず，InvalidPolicies 例外が生成されます。ほかの PolicyManager 内で設定されたポリシーとの互換性は評価されません。

パラメータ	説明
policies	Policy オブジェクトへの参照のシーケンス。
set_add	指定したポリシーの設定方法を示します。PolicyManager にすでに存在するほかのオーバーライドに追加する場合は ADD_OVERRIDE，ほかにオーバーライドがない空の PolicyManager に追加する場合は SET_OVERRIDE を指定します。

CORBA::Object

class CORBA::**Object**

Quality of Service API の Visibroker によるインプリメンテーションを使用して，オブジェクト，スレッド，および ORB にポリシーを割り当てることができます。オブジェクトに割り当てられたポリシーは，ほかのすべてのポリシーを上書きします。

IDL 定義

```
#pragma prefix "omg.org"
module CORBA {
   interface Object {
      Policy get_client_policy(in PolicyType type);
      Policy get_policy(in PolicyType type);
      PolicyList get_policy_overrides(in PolicyTypeSeq types);
      Object set_policy_overrides(in PolicyList policies,in SetOverrideType
         set_add)
         raises (InvalidPolicies);
      boolean validate_connection(out PolicyList inconsistent_policies);
   };
};
```

メソッド

CORBA::Policy_ptr get_client_policy(CORBA::PolicyType **type);**

このオブジェクトリファレンスの有効なオーバーライドポリシーを返します。有効なオーバーライドを取得するため，指定された PolicyType のオーバーライドがあるかどうかをまずこの Object のスコープでチェックし，次に Current のスコープ，最後に ORB スコープでチェックします。要求された PolicyType のオーバーライドが存在しない場合は，その PolicyType に対応するシステム依存のデフォルト値が使用されます。可搬性のあるアプリケーションでは，デフォルトの Policy 値が指定されないので，ORB のスコープで必要なデフォルト値を設定するのが普通です。

CORBA::Policy_ptr get_policy(CORBA::PolicyType **type**);

このオブジェクトリファレンスの有効なポリシーを返します。有効なポリシーとは，要求が行われた場合に使用されるポリシーです。有効なポリシーを判定するため，まず

get_client_policy を使用して，目的の PolicyType の有効なオーバーライドが取得されます。

次に，その有効なオーバーライドが，IOR で指定されているポリシーと比較されます。有効なポリシーは，有効なオーバーライドおよび IOR で指定されたポリシーの両者によって許可される値の共通部分です。共通部分が空の場合は，INV_POLICY 例外が生成されます。共通部分が空でない場合は，その部分に正しく値を持つポリシーが有効なポリシーとして返されます。IOR にポリシーを示す値がない場合は，暗黙的に，任意の正しい値が使用されます。このオブジェクトリファレンスの non_existent または validate_connection を呼び出してから get_policy を呼び出すと，確実に正しい有効なポリシーを取得できます。オブジェクトリファレンスがバインドされる前に get_policy が呼び出された場合，返される有効なポリシーはインプリメンテーションによって異なります。そのような場合，通常のインプリメンテーションは次のいずれかの処理を行います。つまり，例外 CORBA::BAD_INV_ORDER を生成するか，バインディングが実行されると変化する可能性がある PolicyType に対する何らかの値を返すか，バインディングを試みてから有効なポリシーを返します。RebindPolicy の値が TRANSPARENT である場合は，透過的なリバインディングのため，有効なポリシーが呼び出しごとに異なる可能性がある点に注意してください。

メモ Visibroker によるインプリメンテーションでは，このメソッドは，オブジェクト，スレッド，または ORB に割り当てられているポリシーを取得します。

```
CORBA::Object set_policy_overrides(const PolicyList& _policies, CORBA::SetOverrideType
_set_add);
```

同じ名前の PolicyManager メソッドと同じ働きをします。ただし，このメソッドは，オブジェクト，スレッド，または VisiBroker ORB の現在のポリシーセットを，要求されたポリシーオーバーライドのリストを使用して更新します。さらに，同じ名前を持つほかのメソッドは void を返しますが，このメソッドは CORBA::Object を返します。

```
CORBA::Boolean validate_connection(PolicyList inconsistent_policies);
```

このオブジェクトの現在の有効なポリシーが呼び出しの実行を許可する場合，このメソッドは TRUE を返します。このオブジェクトリファレンスがまだバインドされていない場合は，このオペレーションの一環としてバインディングが行われます。このオブジェクトリファレンスがすでにバインドされているが，現在のポリシーオーバーライドが変更されているか，何らかの理由でバインディングが有効でなくなっている場合は，RebindPolicy オーバーライドの設定値に関係なく，リバインドが試みられます。現在の有効な RebindPolicy によって暗黙的なリバインドが許可されていない場合，リバインドを強制的に実行する手段は validate_connection オペレーションしかありません。バインドまたはリバインドには，VisiBroker ORB による GIOP LocateRequests の処理が伴うことがあります。

呼び出しを行うと，現在の有効なポリシーに基づいて INV_POLICY 例外が生成される場合，このメソッドは FALSE を返します。現在の有効なポリシーどうしが矛盾する場合，出力パラメータ inconsistent_policies にその原因となっているポリシーが格納されます。ここで返されたポリシーのリストが，該当するポリシーをすべて網羅していることは保証されません。ポリシーオーバーライドとは関係のない原因によってバインディングが失敗した場合は，適切な例外が生成されます。

Messaging::RebindPolicy

```
class Messaging::RebindPolicy
```

VisiBroker の RebindPolicy は，メッセージング仕様で定義されている RebindPolicy にフェイルオーバーをサポートする機能を加えた完全な実装です。

ORB の RebindPolicy は，ORB が GIOP ロケーション転送メッセージとオブジェクト障害を処理する方法を決定します。VisiBroker ORB は，CORBA::Object のインスタンスの有効なポリシーを調べることにより，フェイルオーバーとリバインドを処理します。

CORBA::Policy から派生されている OMG のインプリメンテーションでは，VisiBroker ORB がターゲットサーバーに正しくバインドした後，透過的にリバインドするかどうかを判定します。拡張されたインプリメンテーションでは，VisiBroker ORB がターゲットオブジェクト，スレッド，または VisiBroker ORB に正しくバインドした後，透過的にフェイルオーバーするかどうかを判定します。

IDL 定義

```
#pragma prefix "omg.org"
module Messaging {
   typedef short RebindMode;
   const CORBA::PolicyType REBIND_POLICY_TYPE = 23;
   interface RebindPolicy CORBA::Policy {
      readonly attribute RebindMode rebind_mode;
   };
}
```

ポリシーの値

メモ ポリシーは，バインドが成功した後でのみ適用されます。

リバインドポリシーとして設定される OMG のポリシー値は，次のとおりです。

ポリシーの値	説明
TRANSPARENT	このポリシーを使用すると，VisiBroker ORB はリモート要求を行う間に，オブジェクト転送および必要な再接続を暗黙的に処理できます。これは，OMG ポリシーの中で最も制約の少ない値です。
NO_REBIND	このポリシーを使用すると，VisiBroker ORB は，リモート要求を行う間に，閉じた接続の再オープンを暗黙的に処理します。ただし，クライアント側の有効な QoS ポリシーを変更するような透過的なオブジェクト転送は行いません。
NO_RECONNECT	このポリシーを使用すると，VisiBroker ORB は，オブジェクト転送または閉じた接続の再オープンを暗黙的に処理できません。これは，OMG ポリシーの中で最も制約の多い値です。

リバインドポリシーとして設定される VisiBroker 固有の値は，次のとおりです。

ポリシーの値	説明
VB_TRANSPARENT	このポリシーは，TRANSPARENT の動作をオブジェクト，スレッド，および VisiBroker ORB におけるフェイルオーバー状態にまで拡張します。これがデフォルトのポリシーです。このポリシーが設定された場合，サーバーオブジェクトの停止が原因でリモート呼び出しが失敗すると，VisiBroker ORB は osagent を使用して別のサーバーに再接続を試みます。リバインドが成功すると，VisiBroker ORB は通信障害をマスクして，クライアントに例外を生成しません。
VB_NOTIFY_REBIND	VB_NOTIFY_REBIND の動作は VB_TRANSPARENT に似ていますが，通信障害が検出された場合に例外を生成します。呼び出しが再試行された場合は，別のオブジェクトに透過的に再接続を試みます。
VB_NO_REBIND	VB_NO_REBIND はフェイルオーバーを無効にします。クライアントの VisiBroker ORB は，閉じた GIOP 接続を同じサーバーに向けて再オープンできるだけで，どのようなオブジェクト転送も許可されません。

QoSExt::DeferBindPlicy

```
class QoSExt::DeferBindPolicy
```

デフォルトでは，VisiBroker ORB は，bind() の呼び出しを受信したときに，（リモート）オブジェクトに接続します。

TRUE に設定された場合，このポリシーはデフォルトの動作を変更します。つまり，ORB は最初の呼び出しがあるまでオブジェクトとのコンタクトを遅延します。

IDL 定義

```
#pragma prefix "inprise.com"
module QoSExt {
   interface DeferBindPolicy :CORBA::Policy {
      readonly attribute boolean value;
   };
};
```

QoSExt::RelativeConnectionTimeoutPolicy

```
class QoSExt::RelativeConnectionTimoutPolicy
```

RelativeConnectionTimeoutPolicy には，利用可能なエンドポイントの 1 つからオブジェクトへの接続の試行が打ち切られるまでのタイムアウト値を指定できます。このポリシー値は，タイムアウトを 100 ナノ秒（ns）単位で指定する unsigned longlong 型の値です。この値は，VisiBroker ORB が接続を試みるすべてのエンドポイントに適用されます。したがって，複数の接続が試られた場合，経過時間は，設定されたタイムアウトの倍数となります。デフォルト値 0 は，タイムアウト値をオペレーティングシステムのデフォルトのタイムアウト値に設定します。

メモ このポリシーは，ローカル IPC やインプロセス通信には適用されません。デフォルトでは，インプロセス通信やローカル IPC は，他の通信より優先順位が高いため，このポリシーを確実に適用する必要がある場合は，インプロセス通信とローカル IPC をオフにしておく必要があります。Java ／ C++ VisiBroker ORB または他の ORB ベンダーとの相互運用性を保証するためにタイムアウトポリシーが必要な場合は，ローカル IPC をオフにします。

IDL 定義

```
module QoSExt {
   const CORBA::PolicyType RELATIVE_CONN_TIMEOUT_POLICY_TYPE = 0x56495304;
   interface RelativeConnectionTimeoutPolicy :CORBA::Policy {
      readonly attribute TimeBase::TimeT relative_expiry;
   };
};
```

Messaging::RelativeRequestTimeoutPolicy

```
class Messaging::RelativeRequestTimoutPolicy
```

RelativeRequestTimeoutPolicy は，クライアントが処理要求の送信を待機してブロックする最大時間を指定します。要求がタイムアウトになると CORBA::TIMEOUT 例外が生成され，サーバーとの接続は廃棄されます。

第 15 章

IOP および IIOP のインターフェースとクラス

ここでは，CORBA 仕様で定義されている GIOP（General Inter-ORB Protocol）の主要インターフェースとその他の構造体の VisiBroker for C++ によるインプリメンテーションについて説明します。これらのインターフェースの詳細については，『OMG CORBA/IIOP Specification』を参照してください。

GIOP::MessageHeader

```
struct MessageHeader
```

この構造体は，GIOP メッセージに関する情報を表すために使用されます。

MessageHeader のメンバー

```
CORBA::Char magic[4]
```

この文字列は，必ず GIOP にする必要があります。

```
Version GIOP_version
```

使用されるプロトコルのバージョンを示します。次に示すように，この構造体には，メジャーバージョンとマイナーバージョンがあります。メジャーバージョンは 1 に設定し，マイナーバージョンは古いバージョン（VisiBroker 3.x）を使用していない限り，2 に設定する必要があります。VisiBroker 3.x の場合はマイナーバージョンを 0 に設定する必要があります。

```
struct Version {
      CORBA::Octet    major;
      CORBA::Octet    minor;
};
```

```
CORBA::Boolean byte_order
```

メッセージのバイトオーダーにリトルエンディアンを使用するには，TRUE に設定します。ビッグエンディアンを使用するには，FALSE に設定します。

CORBA::Octet **message_type**

このヘッダーの後に続くメッセージの型を指定します。次の値の 1 つを指定する必要があります。

```
enum MsgType {
      Request,
      Reply,
      CancelRequest,
      LocateRequest,
      LocateReply,
      CloseConnection,
      MessageError,
      Fragment
    };
```

CORBA::ULong **message_size**

このヘッダーの後に続くメッセージの長さを指定します。

GIOP::CancelRequestHeader

struct **CancelRequestHeader**

この構造体は，キャンセル要求のメッセージヘッダーに関する情報を表すために使用されます。

CancelRequestHeader のメンバー

CORBA::ULong **request_id**

このデータメンバーは，キャンセルされる要求識別子を示します。

GIOP::LocateReplyHeader

struct **LocateReplyHeader**

この構造体は，検索要求メッセージに対する応答として送信されるメッセージを表すために使用されます。locate_status が OBJECT_FORWARD に設定された場合は，このヘッダーの後に追加データが続きます。

LocateReplyHeader のメンバー

CORBA::ULong **request_id**

元の要求の要求識別子。

LocateStatusType **locate_status**

検索要求の特性を示します。次の値を指定できます。

値	説明
UNKNOWN_OBJECT	要求されたオブジェクトが見つからなかったことを示します。このメッセージに関連付けられるその他のデータはありません。
OBJECT_HERE	オブジェクトがこのサーバーによって実装されていることを示します。このメッセージに関連付けられるその他のデータはありません。
OBJECT_FORWARD	応答が LocateRequest メッセージで指定されたオブジェクトへの要求のターゲットとして使用できるオブジェクトリファレンス（IOR）を保持することを示します。オブジェクトは別のサーバーによって実装されており，このヘッダーの後には，そのサーバーの IOR が続きます。
OBJECT_FORWARD_PERM	応答が LocateRequest メッセージで指定されたオブジェクトへの要求のターゲットとして使用できるオブジェクトリファレンス（IOR）を保持することを示します。
LOC_SYSTEM_EXCEPTION	例外が，マーシャリングされた GIOP::SystemExceptionReplyBody を保持していることを示します。
LOC_NEEDS_ADDRESSING_MODE	LocateRequest が再送信される場合に，要求されたアドレッシングモードが使用されることを示します。

GIOP::LocateRequestHeader

structure **LocateRequestHeader**

この構造体は，オブジェクトの検索要求を保持するメッセージを表します。

LocateRequestHeader のメンバー

CORBA::ULong **request_id**

この要求の要求識別子。複数の未処理メッセージを区別するために使用されます。

GIOP::TargetAddress **target**

検索されるオブジェクトを示します。ターゲットは，オブジェクトキー，プロファイル，IOR の 3 つからなる共用体です。

GIOP::ReplyHeader

struct **ReplyHeader**

この構造体は，要求メッセージに対する応答としてクライアントに送信される応答メッセージの応答ヘッダーを表します。

インクルードファイル

この構造体を使用する場合は，**GIOP_c.hh** ファイルをインクルードする必要があります。このファイルは，installation/include ディレクトリ内の corba.h にすでにインクルードされています。

ReplyHeader のメンバー

CORBA::ULong **request_id**

この応答が関連付けられている要求メッセージと同じ request_id に設定する必要があります。

ReplyStatusType **reply_status**

応答の状態を示します。次のいずれかの enum 値に設定する必要があります。

- NO_EXCEPTION
- USER_EXCEPTION
- SYSTEM_EXCEPTION
- LOCATION_FORWARD
- LOCATION_FORWARD_PERM
- NEEDS_ADDRESSING_MODE

IOP::ServiceContextList **service_info**

サーバーからクライアントに渡されるサービスコンテキスト情報のリストです。

GIOP::RequestHeader

struct **RequestHeader**

この構造体は，オブジェクトインプリメンテーションに送信される要求メッセージのリクエストヘッダーを表します。

インクルードファイル

この構造体を使用する場合は，**GIOP_c.hh** ファイルをインクルードする必要があります。このファイルは，installation/include ディレクトリ内の corba.h にすでにインクルードされています。

RequestHeader のメンバー

CORBA::ULong **request_id**

応答メッセージを特定の要求メッセージに関連付けるために使用される一意の識別子です。

CORBA::Boolean **response_expected**

この要求が，応答の必要がない一方向オペレーションである場合，このメンバーは FALSE です。応答が必要な処理要求およびその他の要求の場合，このメンバーは TRUE です。

GIOP::TargetAddress **_target**

要求の対象のオブジェクトは，オブジェクトキー，プロファイル，および IOR の 3 つからなる共用体です。オブジェクトキーは，ベンダー固有の形式で格納され，IOR の作成時に生成されます。

CORBA::String_var **oper**

ターゲットオブジェクトに要求されるオペレーションを識別します。このメンバーは，管理型であるという点を除いて，operator メンバーと同じです。

const char ***operation**

ターゲットオブジェクトに要求されるオペレーションを識別します。このメンバーは，管理型ではないという点を除いて，oper メンバーと同じです。

IOP::ServiceContextList **service_context**

クライアントからサーバーに渡されるサービスコンテキスト情報のリストです。

IIOP::ProfileBody

struct **ProfileBody**

この構造体は，あるオブジェクトによってサポートされるプロトコルに関する情報を保持します。

```
module IIOP {
   . . .
   struct ProfileBody {
      Version iiop_version;
      string host;
      unsigned short port;
      sequence<octet> object_key;
      sequence<IOP::taggedComponent> components;
   };
};
```

ProfileBody のメンバー

Version **iiop_version**

サポートする IIOP のバージョン。

CORBA::String_var **host**

オブジェクトをホストするサーバーが実行されているホストの名前。

CORBA::UShort **port**

オブジェクトをホストするサーバーへの接続を確立するために使用されるポート番号。

CORBA::OctetSequence **object_key**

オブジェクトキーは，ベンダー固有の形式で格納され，IOR の作成時に生成されます。

IIOP::MultiComponentProfile **components**

サポートされているプロトコルに関する情報を保持する TaggedComponents のシーケンス。

IOP::IOR

struct **IOR**

このクラスは，インターオペラブルオブジェクトリファレンス（Interoperable Object Reference，IOR）を表し，オブジェクトリファレンスに関する重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，ORB::object_to_string メソッドを実行して文字列化した IOR を作成することができます。

インクルードファイル

この構造体を使用するには，IOP_c.hh ファイルをインクルードする必要があります。

IOR のメンバー

CORBA::String_var **type_id**

この IOR によって表されるオブジェクトリファレンスの型を記述します。

`TaggedProfileSequence` **profiles**

サポートされるプロトコルに関する情報を保持する 1 つ以上の `TaggedProfile` 構造体のシーケンスです。

IOP::TaggedProfile

`struct` **TaggedProfile**

この構造体は，インターオペラブルオブジェクトリファレンス（IOR）によってサポートされている特定のプロトコルを表します。

TaggedProfile のメンバー

`ProfileID` **tag**

プロファイルデータの内容。次の値の 1 つになります。

値	説明
TAG_INTERNET_IOP	このプロトコルが標準 IIOP であることを示します。
TAG_MULTIPLE_COMPONENTS	このプロトコルを使用して使用可能な VisiBroker ORB サービスのリストがプロファイルデータに保持されていることを示します。
TAG_VB_LOCATOR	この IOR が一時的な擬似オブジェクトであり，osagent が実際の IOR を受け取るまで使用されるオブジェクトであることを示します。
TAG_VSGN_LIOP	このプロトコルがローカルの IPC メカニズムを介した IOP であることを示します。

`CORBA_OctetSequence` **profile_data**

このデータメンバーは，IOR のオペレーションを呼び出すために必要なすべてのプロトコル情報をカプセル化します。

第 16 章

バッファマーシャリングのインターフェースとクラス

ここでは，処理要求や応答メッセージの作成時に，バッファにデータをマーシャリングするために使用されるバッファクラスについて説明します。また，受信された処理要求や応答メッセージからデータを抽出するために使用されるバッファクラスについても説明します。

CORBA::MarshalInBuffer

```
class CORBA::MarshalInBuffer : public VISistream
```

このクラスは，バッファから IDL 型を読み取ることができるストリームバッファを表します。このクラスでは，ユーザーの実装によるインターセプタメソッドを使用できます。インターセプタインターフェースの詳細については，第 12 章「ポータブルインターセプタの インターフェースとクラス」を参照してください。

CORBA::MarshalInBuffer クラスは，クライアント側で，応答メッセージに関連付けられているデータを抽出するために使用されます。また，サーバー側で，処理要求に関連付けられているデータを抽出するために使用されます。このクラスは，さまざまな型のデータをバッファから取得するためのメソッドを数多く提供します。

このクラスには，CORBA::MarshalInBuffer ポインタをテストしたり操作するための static メソッドもあります。

CORBA::MarshalInBuffer_var クラスも用意されています。このクラスは，保持しているオブジェクトを自動的に管理するラッパーを提供します。

インクルードファイル

このクラスを使用するには，**mbuf.h** ファイルをインクルードする必要があります。このファイルは **corba.h** に含まれています。したがって，**mbuf.h** を別にインクルードする必要はありません。

CORBA::MarshalInBuffer のコンストラクタとデストラクタ

CORBA::MarshalInBuffer(char ***read_buffer**, CORBA::ULong **length**,CORBA::Boolean **release_flag**=0, CORBA::ULong **start_offset=0**, CORBA::Boolean **byte_order** = CORBA::ByteOrder);

これはデフォルトのコンストラクタです。

パラメータ	説明
read_buffer	マーシャリングされたデータが実際に格納されるバッファ。
length	read_buffer に格納されるバイトの最大数。
release_flag	TRUE に設定した場合，read_buffer に関連付けられているすべてのメモリは，このオブジェクトが廃棄されたときに解放されます。デフォルト値は，FALSE です。
start_offset	read_buffer にデータを書き込む際の開始オフセット。デフォルト値は，0 です。
byte_order	リトルエンディアンによるバイトオーダーを使用する場合は，TRUE に設定します。ビッグエンディアンによるバイトオーダーを使用する場合は，FALSE に設定します。

virtual **~CORBA::MarshalInBuffer**();

デフォルトのデストラクタです。release_flag が TRUE に設定されている場合，このオブジェクトに関連付けられているバッファメモリは解放されます。release_flag は，オブジェクトの作成時に設定されますが，release_flag メソッドを呼び出して設定することもできます。詳細については，269 ページの「void release_flag(CORBA::Boolean val);」を参照してください。

CORBA::MarshalInBuffer のメソッド

char ***buffer**() const;

このオブジェクトに関連付けられているバッファへのポインタを返します。

void **byte_order**(CORBA::Boolean **val**) const;

このメッセージバッファのバイトオーダーを設定します。

パラメータ	説明
val	リトルエンディアンによるバイトオーダーを使用する場合は，TRUE に設定します。ビッグエンディアンによるバイトオーダーを使用する場合は，FALSE に設定します。

CORBA::Boolean **byte_order**() const;

バッファでリトルエンディアンによるバイトオーダーを使用する場合は，TRUE を返します。ビッグエンディアンによるバイトオーダーを使用する場合は，FALSE を返します。

CORBA::ULong **curoff**() const;

このオブジェクトに関連付けられているバッファ内の現在のオフセットを返します。

virtual VISistream& **get**(char& **data**); virtual VISistream& **get**(unsigned char& **data**);

バッファから，現在の位置にある 1 文字を取得できます。

このメソッドは，ここで取得したデータの直後に続くバッファ内の位置を示すポインタを返します。

パラメータ	説明
data	取得した char または unsigned char を格納する場所。

```
virtual VISistream& get(<data_type> data, unsigned size);
```

バッファから，現在の位置にあるデータのシーケンスを取得できます。下に示すそれぞれのデータ型について，別にメソッドが用意されています。

このメソッドは，ここで取得したデータの直後に続くバッファ内の位置を示すポインタを返します。

パラメータ	説明	
data	取得したデータを格納する場所。 次のデータ型がサポートされています。	
	char*	float*
	unsigned char*	double*
	short*	long double*
	unsigned short*	VISLongLong*
	int*	VISULongLong*
	unsigned int*	wchar_t*
	long*	
	unsigned long*	
size	取得する特定のデータ型のサイズ。	

```
virtual VISistream& getCString(char* data, unsigned maxlen);
```

バッファから，現在の位置にある文字列を取得できます。ここで取得したデータの直後に続くバッファ内の位置を示すポインタが返されます。

パラメータ	説明
data	取得した文字列を格納する場所。
maxlen	取得する最大文字数。

```
virtual int is_available(unsigned long size);
```

指定された size が，このオブジェクトに関連付けられているバッファのサイズと比較して，小さいか等しい場合は，1 を返します。

パラメータ	説明
size	このバッファ内に収まる必要のあるバイト数。

```
virtual CORBA::ULong length() const;
```

このオブジェクトのバッファ内の合計バイト数を返します。

```
virtual void new_encapsulation() const;
```

バッファ内の開始オフセットを 0 にリセットします。

```
void release_flag(CORBA::Boolean val);
```

このオブジェクトが廃棄されたとき，バッファメモリを自動的に解放することを有効または無効にします。

パラメータ	説明
val	val を TRUE に設定した場合，このオブジェクトが廃棄されたときに，オブジェクトのバッファメモリは解放されます。val を FALSE に設定した場合，このオブジェクトが廃棄されたときに，オブジェクトのバッファメモリは解放されません。

```
CORBA::Boolean release_flag() const;
```

このオブジェクトのバッファメモリが自動的に解放される場合は，TRUE を返します。そうでない場合は，FALSE を返します。

```
void reset();
```

開始オフセット，現在のオフセット，およびシーク位置を 0 にリセットします。

```
void rewind();
```

シーク位置を 0 にリセットします。

```
CORBA::ULong seekpos(CORBA::ULong pos);
```

現在のオフセットを pos の値に設定します。pos に，バッファのサイズより大きいオフセットを指定している場合は，CORBA::BAD_PARAM 例外が生成されます。

```
static CORBA::MarshalInBuffer *_duplicate(CORBA::MarshalInBuffer_ptr ptr);
```

ptr で示されるこのオブジェクトへのポインタをコピーして返し，このオブジェクトの参照カウントをインクリメントします。

```
static CORBA::MarshalInBuffer *_nil();
```

CORBA::MarshalInBuffer 型の NULL ポインタを返します。

```
static void _release(CORBA::MarshalInBuffer_ptr ptr);
```

ptr によって示されるオブジェクトの参照カウントをデクリメントします。参照カウントが 0 になった場合，そのオブジェクトは廃棄されます。オブジェクトの構築時に，その release_flag が TRUE に設定されている場合は，オブジェクトに関連付けられているバッファが解放されます。

CORBA::MarshalInBuffer の演算子

```
virtual VISistream&operator>>(<data_type> data);
```

このストリーム演算子を使用すると，指定されたソース data_type のデータのシーケンスをバッファの現在の位置に取り込めます。

このメソッドは，ここで書き込んだデータの直後に続くバッファ内の位置を示すポインタを返します。

パラメータ	説明
data	バッファに書き込むデータ。 次のデータ型がサポートされています。 char*& long& char& unsigned long& unsigned char& float&

パラメータ	説明	
	short&	double&
	unsigned short&	long double&
	int&	wchar_t*&
	unsigned int&	wchar_t&

CORBA::MarshalOutBuffer

```
class CORBA::MarshalOutBuffer : public VISostream
```

このクラスは，バッファに IDL 型を書き込むことができるストリームバッファを表し，ユーザーの実装によるインターセプタメソッドで使用できます。インターセプタインターフェースの詳細については，第 12 章「ポータブルインターセプタの インターフェースとクラス」を参照してください。

CORBA::MarshalOutBuffer クラスは，クライアント側で，オペレーション要求に関連付けられているデータをマーシャリングするために使用されます。また，サーバー側で，応答メッセージに関連付けられているデータをマーシャリングするために使用されます。このクラスは，さまざまな型のデータをバッファに追加したり，書き込まれたデータをバッファから取得するためのメソッドを数多く提供します。

このクラスには，CORBA::MarshalOutBuffer ポインタをテストしたり操作するための static メソッドがあります。

CORBA::MarshalOutBuffer_var クラスも用意されています。このクラスは，保持しているオブジェクトを自動的に管理するラッパーを提供します。

インクルードファイル

このクラスを使用するには，**mbuf.h** ファイルをインクルードする必要があります。このファイルは **corba.h** に含まれています。したがって，**mbuf.h** を別にインクルードする必要はありません。

CORBA::MarshalOutBuffer のコンストラクタとデストラクタ

```
CORBA::MarshalOutBuffer(CORBA::ULong initial_size = 255, CORBA::Boolean release_flag = 0);
```

サイズ initial_size の marshalOutBuffer を作成します。MarshalOutBuffer には，put オペレーションの間に自分自身のサイズを変更する機能があります。書き込まれるすべてのデータを保持できるだけの十分な領域がない場合は，バッファのサイズが 2 倍になります。

パラメータ	説明
initial_size	このオブジェクトに関連付けられているバッファの初期サイズ。デフォルトのサイズは 255 バイトです。
release_flag	TRUE に設定した場合，read_buffer に関連付けられているすべてのメモリは，このオブジェクトが廃棄されたときに解放されます。デフォルト値は，FALSE です。

```
CORBA::MarshalOutBuffer(char *read_buffer, CORBA::ULong len,CORBA::Boolean release_flag=0);
```

指定されたバッファ，バッファ長，および解放フラグ値を持つオブジェクトを作成します。

パラメータ	説明
read_buffer	マーシャリングされたデータが実際に格納されるバッファ。
length	read_buffer に格納されるバイトの最大数。
release_flag	TRUE に設定した場合，read_buffer に関連付けられているすべてのメモリは，このオブジェクトが廃棄されたときに解放されます。デフォルト値は，FALSE です。

```
virtual ~CORBA::MarshalOutBuffer();
```

デフォルトのデストラクタです。release_flag が TRUE に設定されている場合，このオブジェクトに関連付けられているバッファメモリは解放されます。release_flag は，オブジェクトの作成時に設定されますが，release_flag メソッドを呼び出して設定することもできます。詳細については，270 ページの「CORBA::Boolean release_flag() const;」を参照してください。

CORBA::MarshalOutBuffer のメソッド

```
char *buffer() const;
```

このオブジェクトに関連付けられているバッファへのポインタを返します。

```
CORBA::ULong curoff() const;
```

このオブジェクトに関連付けられているバッファ内の現在のオフセットを返します。

```
virtual CORBA::ULong length() const;
```

このオブジェクトのバッファ内の合計バイト数を返します。

```
virtual void new_encapsulation() const;
```

バッファ内の開始オフセットを 0 にリセットします。

```
virtual VISostream& put(char data);
```

バッファの現在の位置に 1 文字を追加します。

このメソッドは，ここで追加したデータの直後に続くバッファ内の位置を示すポインタを返します。

パラメータ	説明
data	格納する char。

```
virtual VISostream& put(const <data_type> data, unsigned size);
```

バッファ内の現在の位置にデータのシーケンスを格納できます。

このメソッドは，ここで追加したデータの直後に続くバッファ内の位置を示すポインタを返します。

パラメータ	説明
data	格納するデータ。 次のデータ型がサポートされています。 char* float* unsigned char* double* short* long double*

パラメータ	説明	
	unsigned short*	VISLongLong*
	int*	VISULongLong*
	unsigned int*	wchar_t*
	long*	
	unsigned long*	
size	格納する特定のデータ型のサイズ。	

virtual VISostream& **putCString**(const char* **data**);

バッファ内の現在の位置に文字列を格納できます。ここで追加したデータの直後に続くバッファ内の位置を示すポインタが返されます。

パラメータ	説明
data	格納する文字列。

void **release_flag**(CORBA::Boolean **val**);

このオブジェクトが廃棄されたとき，バッファメモリを自動的に解放することを有効または無効にします。

パラメータ	説明
val	val を TRUE に設定した場合，このオブジェクトが廃棄されたときに，オブジェクトのバッファメモリは解放されます。val を FALSE に設定した場合，このオブジェクトが廃棄されたときに，オブジェクトのバッファメモリは解放されません。

CORBA::Boolean **release_flag**() const;

このオブジェクトのバッファメモリが自動的に解放される場合は，TRUE を返します。そうでない場合は，FALSE を返します。

void **reset**();

開始オフセット，現在のオフセット，およびシーク位置を 0 にリセットします。

void **rewind**();

シーク位置を 0 にリセットします。

CORBA::ULong **seekpos**(CORBA::ULong **pos**);

現在のオフセットを pos の値に設定します。pos に，バッファのサイズより大きいオフセットを指定している場合は，CORBA::BAD_PARAM 例外が生成されます。

static CORBA::MarshalOutBuffer *_**duplicate**(CORBA::MarshalOutBuffer_ptr **ptr**);

ptr で示されるこのオブジェクトへのポインタをコピーして返し，このオブジェクトの参照カウントをインクリメントします。

static CORBA::MarshalOutBuffer *_**nil**();

CORBA::MarshalOutBuffer 型の NULL ポインタを返します。

static void _**release**(CORBA::MarshalOutBuffer_ptr **ptr**);

ptr によって示されるオブジェクトの参照カウントをデクリメントします。参照カウントが 0 になった場合，そのオブジェクトは廃棄されます。オブジェクトの構築時に，そ

の release_flag が TRUE に設定されている場合は，オブジェクトに関連付けられているバッファが解放されます。

CORBA::MarshalOutBuffer の演算子

```
virtual VISostream& operator<<(<data_type> data);
```

このストリーム演算子を使用すると，指定された data_type のデータをバッファの現在の位置に追加できます。

このメソッドは，ここで書き込んだデータの直後に続くバッファ内の位置を示すポインタを返します。

パラメータ	説明	
data	バッファに書き込むデータ。	
	次のデータ型がサポートされています。	
	const char*	float
	char	double
	unsigned char	long double
	short	VISLongLong
	unsigned short	VISULongLong
	int	wchar_t*
	unsigned int	wchar_t
	long	
	unsigned long	

第 17 章

ロケーションサービスのインターフェースとクラス

ここでは，スマートエージェントのネットワーク上でオブジェクトインスタンスの検索に使用できるインターフェースについて説明します。ロケーションサービスの詳細については，『VisiBroker for C++ 開発者ガイド』の「ロケーションサービスの使い方」を参照してください。

Agent

```
class Agent : public CORBA::Object
```

このクラスは，スマートエージェントのネットワーク上で特定のオブジェクトのすべてのインスタンスを検索するためのメソッドを提供します。このクラスが提供するメソッドは 2 つのカテゴリに分類されます。1 つはオブジェクトに関するデータをスマートエージェントに照会するメソッドであり，もう 1 つはトリガーを扱うメソッドです。

クライアントアプリケーションは，インターフェースリポジトリ ID に基づいて，または，それとインスタンス名を組み合わせてオブジェクト情報を取得できます。

クライアントアプリケーションでトリガーを使用すると，使用できるオブジェクトインスタンスに変化があった場合に，その通知を受けることができます。

IDL 定義

```
interface Agent {
   HostnameSeq all_agent_locations()
         raises (Fail);
   RepositoryIdSeq all_repository_ids()
         raises (Fail);
   ObjSeqSeq all_available()
         raises (Fail);
   ObjSeq all_instances (in string repository_id)
         raises (Fail);
   ObjSeq all_replica (in string repository_id, in string instance_name)
         raises (Fail);
   DescSeqSeq all_available_descs()
```

```
        raises (Fail);
    DescSeq all_instances_descs (in string repository_id)
        raises (Fail);
    DescSeq all_replica_descs (in string repository_id, in string instance_name)
        raises (Fail);
     void reg_trigger(in TriggerDesc desc, in TriggerHandler handler)
       raises (Fail);
     void unreg_trigger(in TriggerDesc desc, in TriggerHandler handler)
       raises (Fail);
    attribute boolean willRefreshOADs;
};
```

インクルードファイル

この構造体を使用する場合は，**locate_c.hh** ファイルをインクルードする必要があります。

Agent のメソッド

```
ObjLocation::HostnameSeq_ptr all_agent_locations();
```

osagent のプロセスが現在実行されているホストを表すホスト名のシーケンスを返します。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
Fail	表示できる FailReason 値には，NO_AGENT_AVAILABLE，NO_SUCH_TRIGGER，AGENT_ERROR があります。Fail クラスの詳細については，280 ページの「Fail」を参照してください。

参照

- 282 ページの「<type>Seq」

```
ObjLocation::ObjSeqSeq all_available();
```

ネットワーク上のあるスマートエージェントに現在登録されているすべてのオブジェクトのオブジェクトリファレンスのシーケンスを返します。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
Fail	表示できる FailReason 値には，NO_AGENT_AVAILABLE，NO_SUCH_TRIGGER，AGENT_ERROR があります。Fail クラスの詳細については，280 ページの「Fail」を参照してください。

参照

- 282 ページの「<type>Seq」

```
ObjLocation::DescSeqSeq_ptr all_available_descs();
```

ネットワーク上の 1 つのスマートエージェントに現在登録されているすべてのオブジェクトの記述を返します。返される記述情報は，リポジトリ ID ごとに整理されます。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
Fail	有効な FailReason 値には，NO_AGENT_AVAILABLE，NO_SUCH_TRIGGER，AGENT_ERROR があります。Fail クラスの詳細については，280 ページの「Fail」を参照してください。

参照
- 283 ページの「<type>SeqSeq」

ObjLocation::ObjSeq_ptr **all_instances**(const char ***repository_id**);

オブジェクトリファレンスのシーケンスを指定した repository_id を持つすべてのインスタンスに返します。

パラメータ	説明
repository_id	取得するオブジェクトリファレンスのリポジトリ ID。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
Fail	NO_SUCH_TRIGGER 以外の FailReason 値が示されます。Fail クラスの詳細については，280 ページの「Fail」を参照してください。

参照
- 282 ページの「<type>Seq」

ObjLocation::DescSeq_ptr **all_instances_descs**(const char ***repository_id**);

指定された repository_id を持つすべてのオブジェクトインスタンスの記述情報を返します。

パラメータ	説明
repository_id	取得するオブジェクト記述のリポジトリ ID。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
Fail	NO_SUCH_TRIGGER 以外の FailReason 値が示されます。Fail クラスの詳細については，280 ページの「Fail」を参照してください。

参照
- 282 ページの「<type>Seq」

ObjLocation::ObjSeq_ptr **all_replica**(const char ***repository_id**, const char ***instance_name**);

指定された repository_id と instance_name を持つオブジェクトのオブジェクトリファレンスのシーケンスを返します。

パラメータ	説明
repository_id	取得するオブジェクトリファレンスのリポジトリ ID。
instance_name	取得するオブジェクトリファレンスのインスタンス名。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
Fail	NO_SUCH_TRIGGER 以外の FailReason 値が示されます。Fail クラスの詳細については，280 ページの「Fail」を参照してください。

参照

282 ページの「<type>Seq」

```
ObjLocation::DescSeq_ptr all_replica_descs(const char *repository_id,const char *instance_name);
```

指定された repository_id と instance_name を持つすべてのオブジェクトインスタンスの記述情報のシーケンスを返します。

パラメータ	説明
repository_id	取得するオブジェクト記述のリポジトリ ID。
instance_name	取得するオブジェクトのインスタンス名。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
Fail	NO_SUCH_TRIGGER 以外の FailReason 値が示されます。Fail クラスの詳細については，280 ページの「Fail」を参照してください。

参照
- 282 ページの「<type>Seq」

```
CORBA::StringSequence* all_repository_ids();
```

いずれかの osagent に既知のインターフェースをすべて取得します。このメソッドは，次の例外を生成する可能性があります。

例外	説明
Fail	リポジトリ ID が無効です。

```
void reg_trigger(const ObjLocation::TriggerDesc& desc, ObjLocation::TriggerHandler_ptr hdlr);
```

desc で指定された記述情報と一致するオブジェクトインスタンスのトリガーハンドラ hdlr を登録します。

メモ TriggerHandler は，トリガーの記述と一致するオブジェクトが使用可能になるたびに呼び出されます。そのオブジェクトの最初のインスタンスが使用可能になるタイミングだけを調べる場合は，最初の通知を受信した後，unreg_trigger メソッドを使用してトリガーを削除する必要があります。

パラメータ	説明
desc	オブジェクトインスタンスの記述情報。リポジトリ ID，インスタンス名，ホスト名などの情報を組み合わせて格納できます。指定する情報を増減することで，監視するオブジェクトインスタンスの幅を調整できます。
hdlr	登録するトリガーハンドラオブジェクト。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
Fail	NO_SUCH_TRIGGER 以外の FailReason 値が示されます。Fail クラスの詳細については，280 ページの「Fail」を参照してください。

```
void unreg_trigger(const ObjLocation::TriggerDesc& desc, ObjLocation::TriggerHandler_ptr hdlr);
```

desc で指定された記述情報と一致するオブジェクトインスタンスのトリガーハンドラ hdlr の登録を解除します。

パラメータ	説明
desc	オブジェクトの記述情報。
hdlr	登録解除するトリガーハンドラオブジェクト。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
Fail	有効な FailReason 値は，NO_SUCH_TRIGGER です。Fail クラスの詳細については，280 ページの「Fail」を参照してください。

```
CORBA::Boolean willRefreshOADs();
```

このクラスが提供するメソッドが呼び出されるたびに，オブジェクトアクティベーションデーモンのセットが更新される場合は，TRUE を返します。そうでない場合は，FALSE を返します。呼び出しがあるたびにキャッシュが更新されない場合は，次の状態が生じる可能性があります。

- 依然としてすべてのオブジェクトが報告されますが，それらのデスクリプタの activable フラグは正しくない可能性があります。
- OAD キャッシュが最後に更新されてから起動された OAD に登録されているオブジェクトが存在するかどうかを確認しようとすると，それらのオブジェクトはその OAD によってアクティブ化されます。

```
void willRefreshOADs(CORBA::Boolean val);
```

このクラスは，オブジェクトアクティベーションデーモンのセットを保持します。このメソッドは，このセット内の OAD の自動更新を有効または無効にします。

パラメータ	説明
val	TRUE に設定した場合，このクラスが提供するメソッドが呼び出されるたびに，OAD セットが更新されます。

Desc

```
struct Desc
```

この構造体は，オブジェクトの特性を記述するための情報を保持します。この章で説明しているロケーションサービスのいくつかのメソッドには，この構造体を引数として渡します。また，ロケーションサービスのいくつかのメソッドからは，この *Desc* 構造体またはそのシーケンスが返されます。

参照
- 282 ページの「<type>Seq」

IDL 定義

```
module ObjLocation {
   struct Desc {
      Object ref;
      IIOP::ProfileBody iiop_locator;
      string repository_id;
      string instance_name;
      boolean activable;
      string agent_hostname;
   };
. . .
};
```

Desc のメンバー

`Object` **ref**

記述されたオブジェクトへのリファレンス。

`IIOP::ProfileBody` **iiop_locator**

オブジェクトのプロファイルデータを表します。詳細については,「IIOP::ProfileBody」を参照してください。

`CORBA::String_var` **repository_id**

オブジェクトのリポジトリ識別子。

`CORBA::String_var` **instance_name**

このオブジェクトのインスタンス名。

`CORBA::Boolean` **activable**

このオブジェクトがオブジェクトアクティベーションデーモン（Object Activation Daemon，OAD）に登録されていることを示す場合は，`TRUE` に設定します。オブジェクトが手動で起動され，osagent に登録されていることを示す場合は，`FALSE` に設定します。

`CORBA::String_var` **agent_hostname**

このオブジェクトが登録されているスマートエージェントが動作するホストの名前。

Fail

`class` **Fail** `: public CORBA::UserException`

この例外クラスは，さまざまなエラーを通知するため，`Agent` クラスによって生成されます。データメンバー `FailReason` は，障害の原因を示すために使用されます。

Fail のメンバー

`FailReason` **reason**

障害の性質を示す次のいずれかの値に設定されます。

```
enum FailReason {
    NO_AGENT_AVAILABLE,
    INVALID_REPOSITORY_ID,
    INVALID_OBJECT_NAME,
    NO_SUCH_TRIGGER,
    AGENT_ERROR
};
```

TriggerDesc

`struct` **TriggerDesc**

この構造体は，`TriggerHandler` の登録対象となる 1 つ以上のオブジェクトの特性を記述する情報を保持します。`TriggerHandler` の詳細については，281 ページの「TriggerHandler」を参照してください。

できる限り広範囲のオブジェクトを監視するには，host_name および instance_name のメンバーを NULL に設定します。指定する情報が多くなるほど，結果のオブジェクトは少なくなります。

IDL 定義

```
module ObjLocation {
   . . .
         struct TriggerDesc {
            string repository_id;
            string instance_name;
            string host_name;
         };
   . . .
};
```

TriggerDesc のメンバー

ORBA::String_var **repository_id**

この TriggerHandler によって監視されるオブジェクトのリポジトリ識別子。有効なリポジトリ識別子をすべて指定する場合は，NULL に設定します。

CORBA::String_var **instance_name**

この TriggerHandler によって監視されるオブジェクトのリポジトリ識別子。有効なインスタンス名をすべて指定する場合は，NULL に設定します。

CORBA::String_var **host_name**;

この TriggerHandler によって監視される 1 つ以上のオブジェクトがあるホストの名前。ネットワークにあるすべてのホストを含むには，NULL に設定します。

TriggerHandler

このベースクラスを使用して，オブジェクトが使用可能または使用不可になるたびに呼び出される独自のコールバックオブジェクトを派生させます。対象となる 1 つ以上のオブジェクトを選択する条件を指定します。作成した TriggerHandler オブジェクトを登録するには，Agent::reg_trigger メソッドを使用します。詳細については，281 ページの「TriggerHandler」を参照してください。

impl_is_ready および impl_is_down メソッドへのインプリメンテーションを提供する必要があります。

IDL 定義

```
interface TriggerHandler {
    void impl_is_ready(in Desc desc);
    void impl_is_down(in Desc desc);
};
```

インクルードファイル

この構造体を使用する場合は，**locate_c.hh** ファイルをインクルードする必要があります。

TriggerHandler のメソッド

```
virtual void impl_is_ready(const Desc& desc);
```

desc で指定される条件に一致するオブジェクトインスタンスにアクセスできるようになると，ロケーションサービスによって呼び出されます。

パラメータ	説明
desc	オブジェクトの記述情報。

```
virtual void impl_is_down(const Desc& desc);
```

desc で指定される条件に一致するオブジェクトインスタンスにアクセス不可になると，ロケーションサービスによって呼び出されます。

パラメータ	説明
desc	オブジェクトの記述情報。

<type>Seq

ロケーションサービスによって使用される次のシーケンスクラスを一般に表したものです。

クラス	説明
DescSeq	Desc 構造体のシーケンス。
HostnameSeq	ホスト名のシーケンス。
ObjSeq	オブジェクトリファレンスのシーケンス。
RepositoryIdSeq	リポジトリ ID のシーケンス。

それぞれのクラスは，特定の <type> のシーケンスを表します。ロケーションサービスは，これらのクラスの 1 つにマッピングされたシーケンスという形で，クライアントアプリケーションに情報のリストを返します。

各クラスには，C++ の配列と同様にシーケンス内の項目に添え字を付けるための演算子が用意されています。また，各クラスは，配列の長さを取得したり，配列の長さを設定するためのメソッドも提供します。

次のコードサンプルに，Agent::all_agent_locations メソッドから返された HostnameSeq に正しく添え字を付ける方法を示します。

```
. . .
ObjLocation::HostnameSeq_var hostnames(myAgent->all_agent_locations());
for (CORBA::ULong i=0; i < hostnames->length(); i++) {
  cout << "Agent host #" << i+1 << ": " << hostnames[i] << endl;
}
. . .
```

参照

- 283 ページの「<type>SeqSeq」

<type>Seq メソッド

```
<type>& operator[](CORBA::ULong index) const;
```

index で識別されるシーケンス内の要素への参照を返します。

注意 インデックスには CORBA::ULong 型を使用する必要があります。int を使用すると，予期しない動作が発生します。

パラメータ	説明
index	返される要素の 0 ベースのインデックス。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
CORBA::BAD_PARAM	指定されたインデックスが，0 より小さいか，シーケンスのサイズより大きい。

```
CORBA::ULong length() const;
```

取得するシーケンス内の要素の数。

```
void length(CORBA::ULong len);
```

シーケンスの最大の長さを len の値に設定します。

パラメータ	説明
len	このシーケンスの新しい長さ。

<type>SeqSeq

ロケーションサービスによって使用される次のクラスを一般に表したものです。

クラス	説明
DescSeqSeq	DescSeq オブジェクトのシーケンス。
ObjSeqSeq	ObjSeq オブジェクトのシーケンス。

それぞれのクラスは，特定の <type>Seq のシーケンスを表します。ロケーションサービスの一部のメソッドは，これらのクラスの 1 つにマッピングされたシーケンスのシーケンスという形で，クライアントアプリケーションに情報のリストを返します。

各クラスには，C++ の配列と同様にシーケンス内の項目に添え字を付けるための演算子が用意されています。また，配列の長さを取得したり，配列の長さを設定するためのメソッドも提供します。

参照

- 282 ページの「<type>Seq」

<type>SeqSeq のメソッド

```
<type>Seq& operator[](CORBA::ULong index) const;
```

index で識別されるシーケンス内の要素への参照を返します。この参照は，282 ページの「<type>Seq」で説明している 1 次元シーケンスへの参照です。

注意 インデックスには CORBA::ULong 型を使用する必要があります。int を使用すると，予期しない動作が発生します。

パラメータ	説明
index	返される要素の 0 ベースのインデックス。

このメソッドは，次の例外を生成する可能性があります。

例外	説明
CORBA::BAD_PARAM	指定されたインデックスが，0 より小さいか，シーケンスのサイズより大きい。

```
CORBA::ULong length() const;
```

取得するシーケンス内の要素の数。

```
void length(CORBA::ULong len);
```

シーケンスの最大の長さを len の値に設定します。

パラメータ	説明
len	このシーケンスの新しい長さ。

第 18 章

初期化のインターフェースとクラス

ここでは，インターセプタなどの VisiBroker ORB サービスを静的に初期化するために提供されるインターフェースとクラスについて説明します。

VISInit

class **VISInit**

この抽象ベースクラスは，VisiBroker ORB と BOA が初期化された後のサービスクラスの静的な初期化のために用意されています。VISInit から独自のサービスクラスを派生させ，それを静的に宣言することにより，そのサービスクラスのインスタンスを正しく初期化できます。

アプリケーションが CORBA::ORB_init または BOA_init メソッドを呼び出すたびに，ORB は VISInit::ORB_init と VISInit::BOA_init を呼び出します。これらのメソッドに独自のインプリメンテーションを提供することにより，目的のサービスに対して必要な初期化を追加できます。

インクルードファイル

このクラスを使用する場合は，vinit.h ファイルをインクルードする必要があります。

VISInit のコンストラクタとデストラクタ

VISInit();

これはデフォルトのコンストラクタです。

VISInit(CORBA::Long **init_priority**);

このコンストラクタは，指定された優先順位にしたがって，VISInit の派生オブジェクトを作成します。優先順位は，ほかの VISInit 派生オブジェクトに対する初期化の時期を決定します。

ユーザー定義のクラスに先立って初期化する必要がある VisiBroker の内部クラスは，負の優先順位値を持ちます。VisiBroker の内部クラスが現在使用している最小の優先順位値は -10 です。

メモ 独自のクラスを VisiBroker の内部クラスより前に初期化する必要がある場合は，-10 より小さい優先順位値を設定する必要があります。

優先順位値を設定しなかった場合のデフォルト値は 0 です。したがって，そのクラスは，VisiBroker の内部クラスより後に初期化されます。

パラメータ	説明
init_priority	このオブジェクトの初期化優先順位。負の優先順位値を指定すると，このクラスはより早く初期化されます。正の優先順位値を指定すると，このクラスはより遅く初期化されます。

```
virtual ~VISInit();
```

デフォルトのデストラクタです。

VISInit のメソッド

```
virtual void ORB_init(int& argc, char * const *argv, CORBA::ORB_ptr orb);
```

VisiBroker ORB の初期化中に呼び出されます。使用するクライアント側インターセプタファクトリの初期化には，独自のインプリメンテーションを用意する必要があります。

パラメータ	説明
argc	引数の数。
argv	引数ポインタの配列。
orb	初期化される VisiBroker ORB。

```
virtual void ORB_initialized(CORBA::ORB_ptr orb);
```

VisiBroker ORB の初期化後に呼び出されます。使用するクライアント側インターセプタファクトリの初期化には，独自のインプリメンテーションを用意する必要があります。

パラメータ	説明
orb	初期化される VisiBroker ORB。

```
virtual void BOA_init(int& argc, char * const *argv, CORBA::BOA_ptr boa);
```

BOA の初期化時に呼び出されます。使用するサーバー側インターセプタファクトリの初期化には，独自のインプリメンテーションを用意する必要があります。

パラメータ	説明
argc	引数の数。
argv	引数ポインタの配列。
boa	初期化される BOA。

```
virtual void ORB_shutdown()
```

VisiBroker ORB がシャットダウンされるときに呼び出されます。

第 19 章

リアルタイム CORBA のインターフェースとクラス

ここでは，VisiBroker for C++ がサポートするリアルタイム CORBA インターフェースについて説明します。

メモ　このインターフェースを使用する前に，サポートされる拡張機能の説明と使い方が記載されている『VisiBroker for C++ 開発者ガイド』の「リアルタイム CORBA 拡張」をお読みください。

はじめに

リアルタイム CORBA は, CORBA 呼び出しの実行で呼び出されるスレッドの数と優先順位を制御し，安定した CORBA ベースのシステムを開発するために役立つ一連の API を提供します。

リアルタイム CORBA API の大部分は，IDL で指定され，CORBA C++ 言語マッピングの規則に基づいて C++ にマップされます。リアルタイム CORBA IDL のスコープはモジュール RTCORBA の範囲内であり，そのため，C++ のすべてのクラス名には `RTCORBA::` というプレフィクスが付きます。

次のセクションでは，以下のリアルタイム CORBA インターフェースとクラスについて説明します。

- `RTCORBA::Current`
- `RTCORBA::Mutex`
- `RTCORBA::NativePriority`
- `RTCORBA::Priority`
- `RTCORBA::PriorityMapping`
- `RTCORBA::PriorityModel`
- `RTCORBA::PriorityModelPolicy`
- `RTCORBA::RTORB`
- `RTCORBA::ThreadpoolId`
- `RTCORBA::ThreadpoolPolicy`

インクルードファイル

この章で説明するリアルタイム CORBA 機能のいずれかを使用するには，アプリケーションでファイル **rtcorba.h** をインクルードする必要があります。このファイルは，VisiBroker for C++ で提供されるインクルードファイルの 1 つです。

RTCORBA::Current

```
class RTCORBA::Current : public CORBA::Object
typedef RTCORBA::Current* Current_ptr
class RTCORBA::Current_var
```

クラス RTCORBA::Current は，リアルタイム CORBA 優先順位値を現在の実行スレッドに関連付けるメソッドや，現在のスレッドに関連付けられているリアルタイム CORBA 優先順位値を読み取ることができるメソッドを提供します。

リアルタイム CORBA 優先順位値を現在のスレッドに関連付けると，その値がただちに基底のスレッドのネイティブ優先順位の設定に使用されます。スレッドに適用されるネイティブ優先順位値は，現在インストールされている優先順位マッピングによって取得されます。

また，クライアント伝搬優先順位モデルが使用されている場合は，スレッドに関連付けられた優先順位でスレッドからの CORBA 呼び出しの優先順位が判断されます。詳細は，『*VisiBroker for C++ 開発者ガイド*』の「リアルタイム CORBA 優先順位モデル」を参照してください。

RTCORBA::Current は，IDL で局所性制約付きのインターフェースとして定義されます。そのため，アプリケーションは，C++ クラス RTCORBA::Current_ptr および RTCORBA::Current_var を使用し，CORBA オブジェクトリファレンスによって RTCORBA::Current を処理します。

詳細は，291 ページの「RTCORBA::Priority」を参照してください。

RTCORBA::Current の作成と破棄

RTCORBA::Current は，特殊なインターフェースです。アプリケーションでは，処理しているインスタンスを考慮する必要がありません。RTCORBA::Current のリファレンスは，RTCORBA::RTORB の resolve_initial_references メソッドから取得され，必要がなくなると通常の方法で解放されます。詳細は，『*VisiBroker for C++ 開発者ガイド*』の「リアルタイム CORBA Current」を参照してください。

IDL 定義

```
// 局所性制約付きオブジェクト
interface Current {
    attribute Priority base_priority;
};
```

RTCORBA::Current のメソッド

```
void base_priority(Priority _val);
```

RTCORBA::Priority 値 _val を実行スレッドに関連付けます。

パラメータ	説明
_val	スレッドに関連付ける優先順位値

```
Priority base_priority();
```

実行スレッドに関連付けられた RTCORBA::Priority 値を取得します。

RTCORBA::Mutex

```
class RTCORBA::Mutex : public CORBA::Object
typedef RTCORBA::Mutex* RTCORBA::Mutex_ptr
class RTCORBA::Mutex_var
class TimeBase {
    typedef unsigned long long TimeT;
};
```

インターフェース RTCORBA::Mutex は，VisiBroker にミューテックス同期プリミティブをアプリケーションに提供します。これは，VisiBroker が ORB リソースを保護するために内部的に使用するミューテックスと同じ優先順位継承プロパティを持つことが保証されています。詳細は,『*VisiBroker for C++ 開発者ガイド*』の「リアルタイム CORBA ミューテックス API」を参照してください。

RTCORBA::Mutex は，IDL で局所性制約付きのインターフェースとして定義されます。そのため，アプリケーションは，C++ クラス RTCORBA::Mutex_ptr および RTCORBA::Mutex_var を使用し，CORBA オブジェクトリファレンスによって RTCORBA::Mutex インスタンスを処理します。

詳細は，294 ページの「RTCORBA::RTORB」を参照してください。

Mutex の作成と破棄

新しい RTCORBA::Mutex は，RTCORBA::RTORB インターフェースの **create_mutex** オペレーションによって取得されます。新しい RTCORBA::Mutex は, unlocked 状態で作成されます。

不要になった RTCORBA::Mutex は，RTCORBA::RTORB の **destroy_mutex** オペレーションを使用して破棄されます。詳細は，294 ページの「RTCORBA::RTORB」を参照してください。

RTCORBA::Mutex_var 型を RTCORBA::Mutex_ptr 型のかわりに使用する場合，リファレンスは，_var インスタンスがスコープ外になると自動的に解放されますが，参照している RTCORBA::Mutex インスタンスは自動的には破棄されません。RTCORBA::Mutex インスタンスは，**destroy_mutex** の呼び出しで破棄する必要があります。

IDL 定義

```
// 局所性制約付きオブジェクト
interface Mutex {
    void lock();
    void unlock();
    boolean try_lock ( in TimeBase::TimeT max_wait );
};

interface RTORB {
    ⋮
    Mutex create_mutex();
    void destroy_mutex( in Mutex the_mutex );
    ⋮
};

// defined in TimeBase.idl
module TimeBase {
```

```
    typedef unsigned long long TimeT;
};
```

RTCORBA::Mutex のメソッド

```
void lock();
```

RTCORBA::Mutex をロックします。RTCORBA::Mutex オブジェクトが unlocked 状態の場合，最初に lock() オペレーションを呼び出したスレッドが Mutex オブジェクトを locked 状態に変更します。Mutex オブジェクトがまだ locked 状態である間は，オーナースレッドがロック解除するまで，後続のスレッドが lock() を呼び出してもブロックされます。

```
void unlock();
```

locked 状態の RTCORBA::Mutex のロックを解除します。

```
CORBA::Boolean try_lock( const TimeBase::TimeT _max_wait );
```

最大で _max_wait の時間まで RTCORBA::Mutex のロックを試行します。時間内にロックに成功した場合は true を返し，時間切れになる前に成功しなかった場合は false を返します。

パラメータ	説明
_max_wait	ロックを試行する最大待機時間（100 ナノ秒単位）。値が 0 の場合は，ロックを待機しません。

RTCORBA::NativePriority

```
typedef CORBA::Short RTCORBA::NativePriority
```

RTCORBA::NativePriority は，リアルタイム ORB が実行されているオペレーティングシステムに固有の優先順位スキームで，優先順位を表します。リアルタイム CORBA アプリケーションは，次の特殊な状況でのみ RTCORBA::NativePriority 値を使用します。

- 優先順位マッピングを定義する場合 詳細は，291 ページの「RTCORBA::PriorityMapping」を参照してください。
- ネイティブ優先順位に基づいて動作するオペレーティングシステムまたは他の非 CORBA サブシステムと直接やり取りする場合 この場合も，インストールされている優先順位マッピングによって実行される必要があります。詳細は，『*VisiBroker for C++ 開発者ガイド*』の「VisiBroker アプリケーションコードでのネイティブ優先順位の使用」を参照してください。

通常，リアルタイム CORBA アプリケーションでは，優先順位は RTCORBA::Priority の値に基づいて表されます。

IDL 定義

```
typedef CORBA::Short NativePriority;
```

RTCORBA::Priority

```
typedef CORBA::Short RTCORBA::Priority
static const Priority RTCORBA::minPriority; // 0
static const Priority RTCORBA::maxPriority; // 32767
```

リアルタイム CORBA アプリケーションで優先順位値を表すためには，型 RTCORBA::Priority を使用する必要があります。この値は，現在インストールされている優先順位マッピングにより，アプリケーションを実行している特定のオペレーティングシステムのネイティブ優先順位スキームにマップされます。リアルタイム CORBA 優先順位の詳細は，『*VisiBroker for C++ 開発者ガイド*』の「リアルタイム CORBA 優先順位」を参照してください。

リアルタイム CORBA アプリケーションでネイティブ優先順位値を使用する必要があるのは，オペレーティングシステムまたは他の非 CORBA サブシステムと直接やり取りする場合だけです。その場合でも，インストールされている優先順位マッピングを使用して実行する必要があります。 詳細は，『*VisiBroker for C++ 開発者ガイド*』の「VisiBroker アプリケーションコードでのネイティブ優先順位の使用」を参照してください。

RTCORBA::Priority 値の範囲は，0 ～ 32767 の範囲です。ただし，優先順位のすべての範囲がリアルタイム CORBA システムで使用されることは想定されていません。かわりに，アプリケーションシステムの設計者は，システムに適切な範囲を決定して，優先順位値がその範囲に限られる優先順位マッピングを実装する必要があります。多くのアプリケーションでは，デフォルトの有効範囲 0 ～ 31 を使用できますが，デフォルトの優先順位マッピングをオーバーライドする必要がある場合もあります。詳細は，291 ページの「RTCORBA::PriorityMapping」を参照してください。

IDL 定義

```
typedef CORBA::Short Priority;
static const Priority minPriority; // 0
static const Priority maxPriority; // 32767
```

RTCORBA::PriorityMapping

```
class RTCORBA::PriorityMapping
```

RTCORBA::PriorityMapping クラスは，RTCORBA::Priority 値とリアルタイム ORB が実行されているオペレーティングシステムのネイティブ優先順位スキーム間のマッピングを行います。ORB は，RTCORBA::Priority 値を RTCORBA::NativePriority 値にマップする必要がある場合，またはその逆の場合に，優先順位マッピングオブジェクトを呼び出します。

リアルタイム CORBA アプリケーションでは，RTCORBA::Priority 値を基準にして優先順位を記述する必要があります。ただし，アプリケーションでオペレーティングシステムまたは他の非 CORBA サブシステムと直接やり取りするためには，インストールされている優先順位マッピングを明示的に使用する必要があります。詳細は，『*VisiBroker for C++ 開発者ガイド*』の「VisiBroker アプリケーションコードでのネイティブ優先順位の使用」を参照してください。

優先順位マッピングでサポートされる RTCORBA::Priority 値の範囲は，常に 0 から開始する必要があります。リアルタイム ORB では，0 の RTCORBA::Priority が有効とみなされます。また，これにより，同じノードにある複数のリアルタイム CORBA システムを簡単に統合できます。

PriorityMapping の作成と破棄

通常のリアルタイム CORBA アプリケーションのコードでは，優先順位マッピングのインスタンスを作成する必要はありません。使用可能な優先順位マッピングが自動的に ORB に使用され，必要に応じてアプリケーションからアクセスできます。

一度に 1 つの優先順位マッピングだけがインストールされます。デフォルトの優先順位マッピングが提供され，デフォルトでインストールされます。このデフォルトの優先順位マッピングは，アプリケーションで実装する優先順位マッピングオブジェクトをインストールすることによってオーバーライドできます。インストールプロセスは，『*VisiBroker for C++ 開発者ガイド*』の「デフォルト以外の優先順位マッピングの使用」に記載されています。

IDL 定義

```
// 「ネイティブ」の IDL 型
native PriorityMapping;
```

RTCORBA::PriorityMapping IDL 型は，「ネイティブ」の IDL 型として定義されます。つまり，異なるプログラミング言語へのマッピングは，言語ごとに定義されます。RTCORBA::PriorityMapping を表す C++ クラスの宣言は，次のとおりです。

```
class PriorityMapping {
    public:
        virtual CORBA::Boolean to_native(
                RTCORBA::Priority corba_priority,
                RTCORBA::NativePriority &native_priority )=0;
        virtual CORBA::Boolean to_CORBA(
                RTCORBA::NativePriority native_priority,
                RTCORBA::Priority &corba_priority )=0;
        virtual RTCORBA::Priority max_priority() = 0;
        PriorityMapping();
        virtual ~PriorityMapping() {}
        static RTCORBA::PriorityMapping * instance();
};
```

各メソッドの目的は，次の「PriorityMapping のメソッド」で説明します。

PriorityMapping のメソッド

```
static RTCORBA::PriorityMapping * instance();
```

この static メソッドは，VisiBroker for C++ に実装され，現在インストールされている優先順位マッピングにリアルタイム CORBA アプリケーションがアクセスするために使用されます。詳細は，『*VisiBroker for C++ 開発者ガイド*』の「VisiBroker アプリケーションコードでのネイティブ優先順位の使用」を参照してください。

```
virtual RTCORBA::Priority max_priority() = 0;
```

このメソッドは，この優先順位マッピングを使用している有効な CORBA 優先順位の最大値を返します。たとえば，インストールされている優先順位マッピングが 0 ～ 31 の範囲でリアルタイム CORBA 優先順位をマップする場合，このメソッドが呼び出されるたびに 31 が返されます。

このメソッドは，新しい優先順位マッピングを実装する場合に実装する必要があります。

```
virtual CORBA::Boolean to_CORBA (
    RTCORBA::NativePriority native_priority,
    RTCORBA::Priority &corba_priority ) = 0;
```

このメソッドは，指定されたネイティブ優先順位値 native_priority をリアルタイム CORBA 優先順位値にマップします。ネイティブ優先順位値がこの優先順位マッピングでサ

ポートされる範囲にある場合は，結果のリアルタイム CORBA 優先順位値が corba_priority に保存され，true 値が返されます。それ以外の場合は，corba_priority は変更されず，false が返されます。

このメソッドは，新しい優先順位マッピングを実装する場合に実装する必要があります。

パラメータ	説明
native_priority	リアルタイム CORBA 優先順位値にマップされるネイティブ優先順位。
corba_priority	マップされるリアルタイム CORBA 優先順位値に割り当てられる変数。

```
virtual CORBA::Boolean to_native (
    RTCORBA::Priority corba_priority,
    RTCORBA::NativePriority &native_priority ) = 0;
```

このメソッドは，指定されたリアルタイム CORBA 優先順位値 corba_priority をネイティブ優先順位値にマップします。リアルタイム CORBA 優先順位値がこの優先順位マッピングでサポートされる範囲にある場合は，結果のネイティブ優先順位値が native_priority に保存され，true 値が返されます。それ以外の場合は，native_priority は変更されず，false 値が返されます。

このメソッドは，新しい優先順位マッピングを実装する場合に実装する必要があります。

パラメータ	説明
corba_priority	ネイティブ優先順位値にマップされるリアルタイム CORBA 優先順位。
native_priority	マップされるネイティブ優先順位値に割り当てられる変数。

RTCORBA::PriorityModel

```
enum RTCORBA::PriorityModel {
    CLIENT_PROPAGATED,
    SERVER_DECLARED
};
```

この列挙体は，*クライアント伝搬優先順位モデル*と *サーバー宣言優先順位モデル*の 2 つの CORBA 優先順位モデルを指定します。これについては，『*VisiBroker for C++ 開発者ガイド*』の「リアルタイム CORBA 優先順位モデル」で説明されています。

この列挙値は，RTCORBA::RTORB の create_priority_model_policy メソッドのパラメータの値として使用されます。詳細は，293 ページの「RTCORBA::PriorityModelPolicy」を参照してください。

RTCORBA::PriorityModelPolicy

```
class RTCORBA::PriorityModelPolicy : CORBA::Policy
```

このリアルタイムポリシー型のインスタンスは，RTCORBA::RTORB の create_priority_model_policy メソッドの呼び出しによって作成されます。次に，ポリシーインスタンスは，ポリシーリストパラメータのメンバーとして create_POA メソッドに渡され，リアルタイム POA の作成時の設定に使用されます。

詳細は，294 ページの「RTCORBA::RTORB」および 293 ページの「RTCORBA::PriorityModel」を参照してください。

IDL 定義

```
interface PriorityModelPolicy : CORBA::Policy {
    readonly attribute PriorityModel priority_model;
```

```
    readonly attribute Priority server_priority;
};
```

RTCORBA::RTORB

```
class RTCORBA::RTORB : public CORBA::Object
typedef RTCORBA::RTORB* RTCORBA::RTORB_ptr
class RTCORBA::RTORB_var
```

インターフェース RTCORBA::RTORB は，リアルタイム CORBA スレッドプールと Mutex を管理するメソッドを提供し，リアルタイム CORBA ポリシーのインスタンスを作成します。

RTCORBA::RTORB は，IDL で局所性制約付きのインターフェースとして定義されます。そのため，アプリケーションは，C++ クラス RTCORBA::RTORB_ptr および RTCORBA::RTORB_var を使用し，CORBA オブジェクトリファレンスによって RTCORBA::RTORB を処理します。

メモ

『*VisiBroker for C++ 開発者ガイド*』で説明されているように，リアルタイム CORBA 拡張をサポートするために，VisiBroker for C++ ORB は特殊な「リアルタイム互換」モードで動作する必要があり，その動作とセマンティクスは通常の動作モードと異なります。"RTORB" リファレンスを取得することで自動的に ORB がこの特殊モードになるため，動作の不整合を防ぐために，アプリケーションでは，**できるだけ早く "RTORB" リファレンスを取得する**必要があります。

289 ページの「RTCORBA::Mutex」，291 ページの「RTCORBA::Priority」，297 ページの「RTCORBA::ThreadpoolId」，および 297 ページの「RTCORBA::ThreadpoolPolicy」を参照してください。リアルタイム CORBA スレッドプールの使い方の詳細は，『*VisiBroker for C++ 開発者ガイド*』の「スレッドプール」を参照してください。

RTORB の作成と破棄

リアルタイム ORB は，明示的に初期化する必要はなく，通常の CORBA::ORB_init 呼び出しの中で暗黙的に初期化されます。

アプリケーションでリアルタイム ORB オペレーションを使用するには，リアルタイム ORB インスタンスへのリファレンスを持つ必要があります。このリファレンスは，ORB_init の呼び出し後はいつでも取得でき，オブジェクト ID 文字列 "RTORB" をパラメータとして CORBA::ORB の resolve_initial_references オペレーションで取得します。詳細は，『*VisiBroker for C++ 開発者ガイド*』の「リアルタイム CORBA ORB」を参照してください。

IDL 定義

```
// 局所性制約付きインターフェース
interface RTORB {
    Mutex create_mutex();
    void destroy_mutex( in Mutex the_mutex );

    exception InvalidThreadpool {};

    ThreadpoolId create_threadpool (
            in unsigned long stacksize,
            in unsigned long static_threads,
            in unsigned long dynamic_threads,
            in Priority default_priority,
            in boolean allow_request_buffering,
            in unsigned long max_buffered_requests,
```

```
            in unsigned long max_request_buffer_size );

        void destroy_threadpool( in ThreadpoolId threadpool )
            raises (InvalidThreadpool);

        void threadpool_idle_time( in ThreadpoolId threadpool,
                in unsigned long seconds )
            raises (InvalidThreadpool);

        PriorityModelPolicy create_priority_model_policy(
                in PriorityModel priority_model,
                in Priority server_priority );

        ThreadpoolPolicy create_threadpool_policy(
                in ThreadpoolId threadpool );
    };
```

RTORB のメソッド

```
Mutex_ptr create_mutex();
```

新しいリアルタイム CORBA Mutex を作成し，リファレンスを返します。

```
void destroy_mutex( Mutex_ptr _the_mutex );
```

リアルタイム CORBA Mutex を破棄します。

パラメータ	説明
_the_mutex	破棄する Mutex のリファレンス

```
ThreadpoolId create_threadpool(
        CORBA::ULong _stacksize,
        CORBA::ULong _static_threads,
        CORBA::ULong _dynamic_threads,
        Priority _default_priority,
        CORBA::Boolean _allow_request_buffering = 0,
        CORBA::ULong _max_buffered_requests = 0,
        CORBA::ULong _max_request_buffer_size = 0 );
```

指定された設定で新しいリアルタイム CORBA スレッドプールを作成し，その RTCORBA::ThreadpoolId を返します。

パラメータ	説明
_stacksize	スレッドプールの各スレッドのスタックサイズ（バイト単位）。
_static_threads	スレッドプールの作成時に作成するスレッドの数。_dynamic_threads が 0 でなければ，この値を 0 にすることができます。
_dynamic_threads	静的に作成されたすべてのスレッドが使用されているときに，スレッドがさらに必要になった場合に，作成できる追加のスレッドの数。_static_threads が 0 でなければ，この値を 0 にする（追加のスレッドが動的に作成されないようにする）ことができます。
_allow_request_buffering	すべてのスレッドが使用されている場合にバッファリングの要求を有効にするブールフラグ。VisiBroker for C++ ではサポートされていません。このパラメータの値は無視されます。
_max_buffered_requests	すべてのスレッドが使用されている場合に要求するバッファの最大数。VisiBroker for C++ ではサポートされていません。このパラメータの値は無視されます。
_max_request_buffer_size	すべてのスレッドが使用されている場合にバッファリングするデータの最大量。VisiBroker for C++ ではサポートされていません。このパラメータの値は無視されます。

```
void destroy_threadpool( ThreadpoolId _threadpool );
```

リアルタイム CORBA スレッドプールを破棄します。スレッドプールは，オブジェクトアダプタからは使用しないでください。使用すると，オペレーションは失敗し，CORBA システム例外が発生します。

パラメータ	説明
_threadpool	破棄するスレッドプールの ThreadpoolId。

```
void threadpool_idle_time(
    ThreadpoolId _threadpool,
    CORBA::ULong _seconds );
```

動的に割り当てられたアイドル状態のスレッドでガベージコレクションが実行されるまでの時間を秒単位で設定します。スレッドプールごとに設定されます。デフォルトでは，動的に割り当てられたスレッドは，300 秒後にガベージコレクションが実行されます。

このメソッドは，VisiBroker 拡張機能専用です。

パラメータ	説明
_threadpool	アイドル時間を設定するスレッドプールの ThreadpoolId。
_seconds	このスレッドプール内の動的に割り当てられたアイドル状態のスレッドを破棄するまでの最大秒。静的に割り当てられたスレッドは破棄されません。

```
PriorityModelPolicy create_priority_model_policy(
    in PriorityModel _priority_model,
    in Priority _server_priority );
```

1 つ以上のリアルタイム POA の設定で使用される RTCORBA::PriorityModelPolicy ポリシーオブジェクトのインスタンスを作成します。293 ページの「RTCORBA::PriorityModel」および 293 ページの「RTCORBA::PriorityModelPolicy」を参照してください。

パラメータ	説明
_priority_model	サーバー宣言優先順位モデルの場合は RTCORBA::SERVER_DECLARED，クライアント伝搬優先順位モデルの場合は RTCORBA::CLIENT_PROPAGATED。
_server_priority	サーバーモデルでは，優先順位値がアクティブ化されるときに個別のオブジェクトに関連付けられていない場合に，この POA でアクティブ化されるオブジェクトでリアルタイム CORBA 優先順位の呼び出しが実行されます。 クライアントモデルでは，リアルタイム CORBA クライアント以外，または呼び出し前に RTCORBA::Current でリアルタイム CORBA 優先順位を指定していないリアルタイム CORBA クライアントからの場合に，この POA でアクティブ化されるオブジェクトでリアルタイム CORBA 優先順位の呼び出しが実行されます。

```
ThreadpoolPolicy create_threadpool_policy(
    in ThreadpoolId _threadpool );
```

1 つ以上のリアルタイム POA の設定で使用される RTCORBA::ThreadpoolPolicy ポリシーオブジェクトのインスタンスを作成します。

パラメータ	説明
_threadpool	POA と関連付けるスレッドプールの ThreadpoolId。

RTCORBA::ThreadpoolId

```
typedef CORBA::ULong RTCORBA::ThreadpoolId
```

リアルタイム CORBA スレッドプールの識別に使用される型 RTCORBA::ThreadpoolId の値。この型の値は，RTCORBA::RTORB の create_threadpool メソッドから返されます。

ID は，スレッドプールポリシーのインスタンスの初期化で使用され，次にリアルタイム POA を設定するための PolicyList パラメータのメンバーとして create_POA の呼び出しで渡されます。詳細は，294 ページの「RTCORBA::RTORB」，297 ページの「RTCORBA::ThreadpoolPolicy」，および『*VisiBroker for C++ 開発者ガイド*』の「オブジェクトアダプタとスレッドプールの関連付け」を参照してください。

IDL 定義

```
typedef unsigned long ThreadpoolId;
```

RTCORBA::ThreadpoolPolicy

```
class RTCORBA::ThreadpoolPolicy : CORBA::Policy
```

このリアルタイムポリシー型のインスタンスは，RTCORBA::RTORB の create_threadpool_policy メソッドの呼び出しによって作成されます。次に，ポリシーインスタンスは，ポリシーリストパラメータのメンバーとして create_POA メソッドに渡され，リアルタイム POA の作成時の設定に使用されます。詳細は，294 ページの「RTCORBA::RTORB」，297 ページの「RTCORBA::ThreadpoolId」，および『*VisiBroker for C++ 開発者ガイド*』の「オブジェクトアダプタとスレッドプールの関連付け」を参照してください。

IDL 定義

```
interface ThreadpoolPolicy : CORBA::Policy {
    readonly attribute ThreadpoolId threadpool;
};
```

第 20 章

プラグイン可能トランスポート インターフェースのクラス

この章では，VisiBroker for C++ が提供するプラグイン可能トランスポートインターフェースのクラスについて説明します。VisiBroker のプラグイン可能トランスポートインターフェースを介してトランスポートプロトコルのサポートを実装する方法については，『開発者ガイド』の「VisiBroker プラグイン可能トランスポートインターフェース」の章を参照してください。

重要 最新のマニュアルについては，www.borland.com/techpubs/bes を参照してください。

VISPTransConnection

このクラスは，VisiBroker が処理するトランスポートプロトコルごとに実装する必要がある接続クラスの抽象ベースクラスです。派生クラスの各インスタンスは，サーバーとクライアント間の単一の接続を表します。新しい接続が必要なときは，ORB のクライアント側とサーバー側の両方で，対応するファクトリクラスを使用してこのクラスのインスタンスを作成する（「virtual CORBA::Boolean waitNextMessage(CORBA::ULong _timeout) = 0;」を参照）必要があります。

インクルードファイル

このクラスを使用するには，**vptrans.h** ファイルをインクルードする必要があります。

VISPTransConnection メソッド

```
virtual void close() = 0;
```

派生接続クラスによって実装されます。このメソッドは，接続を正常に閉じます。接続のクライアント側とサーバー側のどちらからも接続を閉じることができる必要があります。

```
virtual void connect(CORBA::ULongLong _timeout) = 0;
```

派生接続クラスによって実装されます。このメソッドはクライアント側 ORB によって呼び出され，リモートピアの「Listener」インスタンスと通信して，サーバー側に新しい接続を設定します。関数はエラーコードを返しませんが，トランスポート層のエラーが発生した場合は，例外を生成します。例外はクライアント CORBA アプリケーションにスローバックされるため，CORBA ユーザー例外を含むあらゆる例外が生成される可能性があります。生成される可能性がある例外の 1 つとして，CORBA::TRANSIENT があります。

タイムアウト値は，ミリ秒単位で指定します。値 0 はタイムアウトなし（永遠にブロック）を意味し，これがデフォルト値です。VisiBroker ポリシーシステムを介してタイムアウトが設定されない限り，このデフォルト値が使用されます。トランスポートが接続時にタイムアウトをサポートしない場合でも，問題なくトランスポートを使用できます。この場合，接続が確立されるか失敗するまで，接続呼び出しは常にブロックされる必要があります。

パラメータ	説明
_timeout	使用するタイムアウト値（ミリ秒単位）。0 はタイムアウトなし（永遠にブロック）を意味します。

```
virtual void flush() = 0;
```

派生接続クラスによって実装されます。このトランスポートがデータをバッファに入れると，このメソッドは，出力のためにバッファに格納されたすべてのデータをただちに送信し，データの送信が完了するまでブロックします。そうでない場合は，何も行わずに，すぐに戻ることができます。

```
virtual IOP::ProfileValue_ptr getPeerProfile() = 0;
```

派生接続クラスによって実装されます。このメソッドは，この接続で使用されるピアエンドポイントを記述するプロファイルのコピーを返します。コピーはヒープ上に作成され，呼び出し元は使用されたメモリを解放する必要があります。プロファイルは，このインスタンスの実際の接続を示すのではなく，「connect」呼び出し時に使用された「Listener」エンドポイントのプロファイルを示します。

```
virtual CORBA::Long id() = 0;
```

派生接続クラスによって実装されます。このメソッドは，各接続インスタンスに一意な番号を返します。ID は，このトランスポートでのみ一意である必要があります。この ID は，このトランスポートに対する要求のディスパッチ時に，接続インスタンスを参照／検索するために使用されます。

```
virtual CORBA::Boolean isBridgeSignalling() = 0;
```

派生接続クラスによって実装されます。このメソッドは，使用する作業スレッドの「冷却」ストラテジを ORB に示すために使用されます。メソッドが 0（FALSE）を返す場合，これは，要求の読み取り後に，プロトコルプラグイン自身が接続の再読み取りを処理することを意味します。これは，プロトコルのエンドポイントでタイムアウトによるブロッキング読み取りをプラグインが実行できる場合にのみ可能です。ブロッキング読み取りを実行できない，または実行しない場合，このメソッドは 1（TRUE）を返します。この場合，別の要求を利用できる状態になるか，またはタイムアウトに達すると，トランスポートブリッジがスレッドを通知します。スレッドの冷却は，プロトコルインスタンスに冷却時間が設定されている場合のみ発生します。

```
virtual CORBA::Boolean isConnected() = 0;
```

派生接続クラスによって実装されます。リモートピアがまだ接続されている場合，このメソッドは 1（TRUE）を返します。接続がピアによって閉じられた場合や，この接続の使用を妨げるエラー状況が発生した場合は，0（FALSE）を返します。

```
virtual CORBA::Boolean isDataAvailable() = 0;
```

派生接続クラスによって実装されます。接続からデータを読み取り可能な状態の場合，このメソッドは 1（TRUE）を返します。そうでない場合は，0（FALSE）を返す必要があります。

```
virtual CORBA::Boolean no_callback() = 0;
```

派生接続クラスによって実装されます。このメソッドは，このトランスポートの接続を使用することにより，クライアント／サーバー設定を逆にしてクライアントコードでサーバントにコールバックできるかどうかを示します。できない場合，このメソッドは 0（FALSE）を返し，ORB はこの種の呼び出しのために新しい接続を作成します。できる場合は 1（TRUE）を返します。

この機能は，GIOP-1.2 で導入された双方向 IIOP をサポートするために提供されています。詳細は，CORBA 仕様を参照してください。

```
virtual void read(CORBA::Boolean _isFirst, CORBA::Boolean _isLast, char* _data, CORBA::ULong
_offset, CORBA::ULong _length, CORBA::ULongLong _timeout)= 0;
```

派生接続クラスによって実装されます。このメソッドは，接続からデータを読み取ります。エラーコードは返しませんが，トランスポート関連のエラーは例外を生成することによって通知する必要があります。引数は，値を格納する必要がある特定の長さのバイト配列です。この関数は，完全なバイト配列を正常に格納するか，タイムアウトするか，例外を生成します。

timeout パラメータの値は，VisiBroker QoS ポリシーを介してユーザーが設定しない限り，デフォルトでは 0 です。値 0 はタイムアウトなしを意味するため，読み取りはデータを待機する間いつまでもブロックされます。したがって，このトランスポートが読み取り / 書き込み時にタイムアウトをサポートしない場合でも，問題なくトランスポートを使用できます。この場合，read 呼び出しは，すべてのデータが到着するまでブロックされます。

パラメータ	説明
_isFirst	これが接続からの最初のデータ読み取りの場合，TRUE。
_isLast	これが接続からの最後のデータ読み取りの場合，TRUE。
_data	データの読み取り先のバイト配列。
_offset	読み取りデータの格納を開始する配列でのオフセット。
_length	読み取られるデータのバイト数
_timeout	使用するタイムアウト値（ミリ秒単位）。0 はタイムアウトなし（永遠にブロック）を意味します。

```
virtual void setupProfile(const char* prefix, VISPTransProfileBase_ptr peer) = 0;
```

派生接続クラスによって実装されます。このメソッドは，新たに作成されたクライアント側接続オブジェクトに，後の手順で（connect() が呼び出されたとき）どのピアへの接続を試行するかを示すために 使用されます。指定された VISPTransProfileBase_ptr ベースクラスは特定のトランスポートのプロファイルクラス型にキャストされ，接続内のすべてのメンバーデータはそのインスタンスによって初期化されます。追加のプロパティパラメータを読み取る必要がある場合は，プロパティ検索のためのプレフィクス文字列も渡されます。

パラメータ	説明
prefix	「vbroker.se.<SE_name>.scm.<SCM_name>」という形式の文字列プレフィクス。メソッドは，このプレフィクスを使用して，このインスタンスを設定するために使用された可能性があるプロトコル固有の VisiBroker プロパティを読み取ることができます。
peer	この接続が接続する監視エンドポイントのプロファイル。このプロトコルのプロファイルクラスのインスタンスとして指定され，ベース VISPTransProfile クラスへのポインタとして渡されます。

```
virtual CORBA::Boolean waitNextMessage(CORBA::ULong _timeout) = 0;
```

派生接続クラスによって実装されます。このメソッドは，この接続がデータを受け取るか，指定されたタイムアウト（ミリ秒単位）が終了するまで，呼び出し元スレッドをブロックします。データを使用できるようになった場合は 1（TRUE），そうでない場合は 0（FALSE）を返します。_timeout パラメータの値が 0 になることはありません（値が 0 の場合, ORB はこのメソッドを呼び出せないため）。したがって, この値を受け取った場合は，エラーメッセージをログに記録してエラーとして処理します。

パラメータ	説明
_timeout	メッセージを待機する最大時間（秒単位）。0 は，永遠に待機することを意味します。

```
virtual void write(CORBA::Boolean _isFirst, CORBA::Boolean _isLast, char* _data,
CORBA::ULong _offset, CORBA::ULong _length, CORBA::ULongLong _timeout) = 0;
```

派生接続クラスによって実装されます。このメソッドは，接続を通じてデータをリモートピアに送信します。エラーコードは返しませんが，トランスポート関連のエラーは例外を生成することによって通知する必要があります。引数は，送信する必要がある特定の長さのバイト配列です。この関数は，完全なバイト配列を正常に送信するか，タイムアウトするか，例外を生成します。timeout パラメータの値は，VisiBroker QoS ポリシーを介してユーザーが設定しない限り，デフォルトでは 0 です。値 0 はタイムアウトなしを意味するため，書き込みはデータを待機する間いつまでもブロックされます。したがって，このトランスポートが読み取り / 書き込み時にタイムアウトをサポートしない場合でも，問題なくトランスポートを使用できます。この場合，write 呼び出しは，すべてのデータが到着するまでブロックされます。

パラメータ	説明
_isFirst	これが接続を通じた最初のデータ送信の場合，TRUE。
_isLast	これが接続を通じた最後のデータ送信の場合，TRUE。
_data	送信する必要があるデータのバイト配列。
_offset	読み取りデータの格納を開始する配列でのオフセット。
_length	送信されるデータのバイト数
_timeout	使用するタイムアウト値（秒単位）。0 はタイムアウトなし（永遠にブロック）を意味します。

VISPTransConnectionFactory

このクラスは，VisiBroker が処理するトランスポートプロトコルごとに実装する必要がある接続ファクトリクラスの抽象ベースクラスです。派生クラスの Singleton インスタンスが VISPTransRegistrar クラス（後述）を介して VisiBroker に登録されます。ORB は，接続ファクトリオブジェクトを呼び出して，関連付けられたトランスポートの接続クラスのインスタンスを作成します。接続クラスは，VISPTransConnection クラスから派生した対応クラスです。

インクルードファイル

このクラスを使用するには，**vptrans.h** ファイルをインクルードする必要があります。

VISPTransConnectionFactory メソッド

```
VISPTransConnection_ptr create(const char* prefix) = 0;
```

派生接続ファクトリクラスによって実装されます。このメソッドは，対応する接続クラスの新しいインスタンスを作成し，ベースクラス型にキャストされたインスタンスへの

ポインタを返します。インスタンスが不要になった場合，呼び出し元はインスタンスを破棄する必要があります。

パラメータ	説明
prefix	「vbroker.se.<SE_name>.scm.<SCM_name>」という形式の文字列プレフィクス。メソッドは，このプレフィクスを使用して，接続ファクトリを設定するために使用された可能性があるプロトコル固有の VisiBroker プロパティを読み取ることができます。

VISPTransListener

このクラスは，VisiBroker が処理するトランスポートプロトコルごとに実装する必要があるリスナーファクトリの抽象ベースクラスです。特定のトランスポートプロトコルを指定するサーバー接続マネージャ（SCM）を含むサーバーエンジンが作成されるたびに，派生クラスのインスタンスが作成されます。プロトコルを指定する SCM インスタンスごとに 1 つのインスタンスが作成されます。リスナーインスタンスは，クライアントからの着信接続と要求を待機するためにサーバー側 ORB で使用されます。新しい接続と，既存の接続での要求は，プラグイン可能トランスポートインターフェースのブリッジクラス（307 ページの「VISPTransBridge」を参照）を介してリスナーによって ORB に通知されます。既存の接続で要求が受信されると，接続は「ディスパッチサイクル」に入ります。ディスパッチサイクルは，接続がデータをトランスポート層に転送すると開始されます。この初期ステージでは，ブリッジを介して，このデータの着信を ORB に通知する必要があります。リスナーは，ディスパッチプロセスが完了するまで接続を無視します。このとき，接続は「ディスパッチ状態」にあると言います。ORB がリスナーの completedData() メソッドを呼び出すと，接続が初期状態に戻ります。ディスパッチ状態の間，ORB は，ブリッジ／リスナー間通信によるオーバーヘッドを回避するため，要求がなくなるまで接続から直接読み取りを行います。ほとんどの場合，トランスポート層は，ブロック呼び出しを使用して新しい接続を待機します。この状況を処理するために，リスナーは，クラス VISThread のサブクラスを作成し，ORB 全体を停止しないでブロックできる独立した実行スレッドを開始する必要があります。

インクルードファイル

このクラスを使用するには，**vptrans.h** ファイルをインクルードする必要があります。

VISPTransListener メソッド

```
virtual void completedData(CORBA::Long id) = 0;
```

派生リスナークラスによって実装されます。このメソッドは，ORB が指定された ID を持つ接続からの要求の読み取りを完了し，その接続での新しい着信要求を（ブリッジを介して）通知するようにリスナーに再度要求するときに呼び出されます。

パラメータ	説明
id	再度監視される接続の ID。

```
virtual void destroy() = 0;
```

派生リスナークラスによって実装されます。このメソッドは，エンドポイントを破棄し，関連するすべてのアクティブな接続を閉じるようにリスナーインスタンスに指示します。

```
virtual IOP::ProfileValue_ptr getListenerProfile() = 0;
```

派生リスナークラスによって実装されます。このメソッドは，このトランスポートでのリスナーインスタンスのエンドポイントを記述するプロファイルを返します。返された

プロファイルは，ヒープにコピーされ，そのメモリ管理は呼び出し元（ORB）が引き継ぎます。

```
virtual CORBA::Boolean isDataAvailable(CORBA::Long id) = 0;
```

派生接続ファクトリクラスによって実装されます。指定された ID の接続に読み取り可能なデータがある場合，このメソッドは 1（TRUE）を返します。そうでない場合は 0（FALSE）を返します。通常，呼び出しは，検索するトランスポート層に転送されます。

パラメータ	説明
id	使用可能なデータがあるかどうかを確認するために照会する接続の ID。

```
virtual void setBridge(VISPTransBridge* up) = 0;
```

派生リスナークラスによって実装されます。このメソッドは，このリスナーインスタンスが使用するプラグイン可能トランスポートブリッジインスタンスへの「リンク」を確立します。メソッドがリスナーに渡すポインタは，必要に応じて ORB への「アップコール」を行うために保存されます。

パラメータ	説明
up	リスナーインスタンスが ORB との通信に使用するプラグイン可能トランスポートブリッジインスタンスへのポインタ。

VISPTransListenerFactory

このクラスは，VisiBroker が処理するトランスポートプロトコルごとに実装する必要があるリスナーファクトリクラスの抽象ベースクラスです。派生クラスの Singleton インスタンスが VISPTransRegistrar クラスを介して VisiBroker に登録されます。ORB は，このオブジェクトを呼び出して，関連付けられたトランスポートのリスナークラスのインスタンスを作成します。リスナークラスは，「VISPTransListener」で説明されているように，VISPTransListener クラスから派生した対応クラスです。

インクルードファイル

このクラスを使用するには，**vptrans.h** ファイルをインクルードする必要があります。

VISPTransListenerFactory メソッド

```
VISPTransListener_ptr create(const char* propPrefix) = 0;
```

派生リスナーファクトリクラスによって実装されます。このメソッドは，対応するリスナークラスの新しいインスタンスを作成し，ベースクラス型にキャストされたインスタンスへのポインタを返します。インスタンスが不要になった場合，呼び出し元（ORB）はインスタンスを破棄する必要があります。

パラメータ	説明
propPrefix	「vbroker.se.<SE_name>.scm.<SCM_name>」という形式の文字列プレフィクス。メソッドは，このプレフィクスを使用して，リスナーインスタンスまたは作成中の特定のリスナーインスタンスを設定するために使用された可能性があるプロトコル固有の VisiBroker プロパティを読み取ることができます。作成中のリスナーインスタンスのコンストラクタにファクトリがプレフィクスを渡すことにより，コンストラクタ自身がプロパティを読み取ることができます。それには，派生リスナークラスのコンストラクタはプレフィクスをパラメータとしてとる必要があります。

VISPTransProfileBase

```
class VISPTransProfileBase : public GIOP::ProfileBodyValue, public
CORBA_DefaultValueRefCountBase
```

このクラスは，VisiBroker が処理するトランスポートプロトコルごとに実装する必要があるプロファイルクラスの抽象ベースクラスです。このクラスは，トランスポート固有のエンドポイント記述と，他の CORBA インプリメンテーションと交換可能な CORBA IOP ベースの IOR との変換機能を提供します。また，「解析」関数に ProfileValue を渡すことで，クライアントをサーバーにバインドするプロセスでも使用されます。この関数は，特定の IOR がこのトランスポートで使用できるかどうかを判定するために，TRUE または FALSE を返す必要があります。派生プロファイルクラスのインスタンスは，多くの場合，そのベースクラス型のポインタを介して関数に渡されます。どの C++ コンパイラを使用しても実行時に安全にダウンキャストできるように，キャストが適正かどうかをテストする「_downcast」関数を提供する必要があります。

インクルードファイル

このクラスを使用するには，**vptrans.h** ファイルをインクルードする必要があります。

VISPTransProfileBase メソッド

```
static GIOP::ObjectKey* convert(const PortableServer::ObjectId& seq);
```

オブジェクトキーの Octet シーケンス表現をインメモリ表現に変換します。

パラメータ	説明
seq	インメモリ表現に変換するオブジェクトキーの Octet シーケンスバージョン。

```
void object_key(GIOP::ObjectKey_ptr k);
```

このプロファイルインスタンスのオブジェクトキーを設定します。

パラメータ	説明
k	オブジェクトキー

```
const GIOP::ObjectKey_ptr object_key() const;
```

このプロファイルインスタンスのオブジェクトキーを取得します。

```
void version(const GIOP::Version& v);
```

このプロファイルの GIOP バージョンを設定します。

パラメータ	説明
v	GIOP バージョン

```
GIOP::Version& version();
```

このプロファイルの GIOP バージョンを取得します。

```
const GIOP::Version& version() const;
```

このプロファイルの GIOP バージョンを取得します。

```
static const VISValueInfo& _info();
```

このプロファイルタイプの VisiBroker ValueInfo を取得します。

VISPTransProfileBase のメンバー

```
static const VISValueInfo& _stat_info;
```

この特定のプロファイルタイプの VisiBroker ValueInfo を格納します。

VISPTransProfileBase のベースクラスメソッド

```
IOP::ProfileValue_ptr copy()
```

派生リスナーファクトリクラスによって実装されます。このメソッドは，メモリ上に正確なコピーを作成し，コピーへのポインタを返します。コーディングでは，この関数内でコピーコンストラクタを使用することをお勧めします。

```
CORBA::Boolean matchesTemplate(IOP::ProfileValue_ptr body);
```

派生プロファイルクラスによって実装されます。指定されたデータに，このトランスポートを介した接続に使用できる IOR がある場合，このメソッドは 1（TRUE）を返す必要があります。そうでない場合は 0（FALSE）を返します。

パラメータ	説明
body	このトランスポートで使用できるかどうかを確認するためにチェックするプロファイル。

```
IOP::ProfileId tag()
```

派生プロファイルクラスによって実装されます。このメソッドは，このプロファイルの一意のタグ値を返します。

```
IOP::TaggedProfile* toTaggedProfile();
```

派生プロファイルクラスによって実装されます。このメソッドは，このインスタンスのメンバーデータから読み取った値を使用して作成されたタグ付き（文字列化）プロファイルインスタンスを返します。

```
static VISPTransProfileBase* _downcast(CORBA::ValueBase* vbptr);
```

派生プロファイルクラスによって実装されます。このプロファイルクラスのインスタンスにベースクラスポインタをダウンキャストする関数です。

パラメータ	説明
vbptr	ベースの値型ポインタとして渡されるプロファイルインスタンス。

```
virtual void* _safe_downcast(const VISValueInfo &info) const;
```

派生リスナーファクトリクラスによって実装されます。ダウンキャスト時に型情報データをチェックするために呼び出される仮想メソッドです。

パラメータ	説明
info	このプロファイルタイプの VisiBroker ValueInfo。

VISPTransProfileFactory

このクラスは，VisiBroker が処理するトランスポートプロトコルごとに実装する必要があるプロファイルファクトリクラスの抽象ベースクラスです。 派生クラスの Singleton インスタンスが VISPTransRegistrar クラスを介して VisiBroker に登録されます。ORB は，このオブジェクトを呼び出して，関連付けられたトランスポートのプロファイルクラスのインスタンスを作成します。 プロファイルクラスは，「VISPTransProfileBase」で説明されているように，VISPTransProfileBase クラスから派生した対応クラスです。

インクルードファイル

このクラスを使用するには，**vptrans.h** ファイルをインクルードする必要があります。

VISPTransProfileFactory メソッド

```
IOP::ProfileValue_ptr create(const IOP::TaggedProfile& profile)
```

タグ付き IOR を読み取り，リスナーエンドポイントを記述するプロファイルを作成します。

パラメータ	説明
profile	CDR でエンコードされた読み取り対象の IOR。

```
CORBA::ULong hash(VISPTransProfileBase_ptr prof);
```

ハッシュ検索テーブルにプロファイルを格納する機能をサポートするために，指定されたインスタンスのハッシュ値を計算します。ハッシュ値を提供しない場合は 0 を返します。

パラメータ	説明
prof	ハッシュ値を作成するプロファイルインスタンス。

```
IOP::ProfileId getTag();
```

このファクトリによって作成されたプロファイルタイプの一意のプロファイル ID タグを返します。

VISPTransBridge

このクラスは，トランスポートクラスと ORB 間の汎用インターフェースを提供します。トランスポート層で発生するさまざまなイベントを通知するメソッドを提供します。

インクルードファイル

このクラスを使用するには，**vptrans.h** ファイルをインクルードする必要があります。

VISPTransBridge メソッド

```
CORBA::Boolean addInput(VISPTransConnection_ptr con);
```

リスナーエンドポイントを表す接続インスタンスへのポインタを渡すことで，ブリッジを介して接続要求を ORB に送信します。返されるフラグは，ORB が新しい接続を受け

入れたか（1（TRUE）を返す），拒絶したか（0（FALSE）を返す）を通知します。拒絶は，リソースの制約や接続の制限（プロパティシステムで設定）によって発生します。

メンバー	説明
con	接続を要求しているリスナーエンドポイントを表す接続オブジェクト。

```
void signalDataAvailable(CORBA::Long conId);
```

トランスポート層から新しいデータを取得した接続の接続 ID を ORB に渡します。これにより，着信要求のディスパッチサイクルが開始されます。

メンバー	説明
conId	データが使用可能であることを示す接続の接続 ID

```
void closedByPeer(CORBA::Long conId);
```

指定された ID の接続がリモートピアによって閉じられたことを ORB に通知します。

メンバー	説明
conId	リモートピアによって閉じられたことが通知される接続の接続 ID。

VISPTransRegistrar

新しいトランスポートを ORB に登録するには，このクラスを使用する必要があります。登録時に指定されるプロトコル名文字列は，このトランスポートの識別名として使用され，その ORB の範囲内で一意である必要があります。この文字列は，このトランスポートに関連付けられたプロパティの名前文字列のプレフィクスとしても使用されます。

インクルードファイル

このクラスを使用するには，**vptrans.h** ファイルをインクルードする必要があります。

VISPTransRegistrar メソッド

```
static void addTransport(const char* protocolName, VISPTransConnectionFactory* connFac,
VISPTransListenerFactory* listFac, VISPTransProfileFactory* profFac);
```

このトランスポートの特定のクラスの作成に使用されるプロトコル名文字列と，3 つのファクトリインスタンスを登録します。このメソッドは静的であるため，ORB の初期化中にいつでも呼び出すことができます。

メンバー	説明
protocolName	このトランスポートプロトコルの識別に使用される名前。
connFac	接続ファクトリの Singleton インスタンスへのポインタ。
listFac	リスナーファクトリの Singleton インスタンスへのポインタ。
profFac	プロファイルファクトリの Singleton インスタンスへのポインタ。

第 21 章

VisiBroker for C++ のログ

ここでは，VisiBroker for C++ のログをサポートするクラスについて説明します。

VISDLoggerMgr

このクラスは，ログライブラリ vdlog によって提供される機能へのブートストラップクラスです。

インクルードファイル

このクラスを使用する場合は，**vdlog.h** ファイルをインクルードする必要があります。

VISDLoggerMgr メソッド

```
static VISDLoggerMgr_ptr instance();
```

VISDLoggerMgr の Singleton インスタンスにアクセスする静的関数。

```
CORBA::Boolean global_log_enabled();
```

グローバルログスイッチが有効な場合は true を返し，そうでない場合は false を返します。

```
void global_log_enabled(CORBA::Boolean b);
```

グローバルログレベルスイッチのセッターメソッド。

パラメータ	説明
b	グローバルログレベルスイッチを有効または無効にするブール値

```
VISDLogLevel::Level global_log_level();
```

ログマネージャの現在のグローバルログレベル（詳細レベル）設定を返します。

```
void global_log_level(VISDLogLevel::Level l);
```

ログマネージャのグローバルログレベルのセッターメソッド。

パラメータ	説明
l	グローバル詳細レベルの設定

```
VISDLogger_ptr get_default_logger();
```

デフォルトのロガーを返します。ロガーが作成されていない場合は，作成して返します。返されるロガーの名前は「default」です。

```
VISDLogger_ptr get_logger(const char* name, VISDAppender_ptr* apps = NULL, CORBA::Short
num_apps = 0);
```

指定された名前のロガーが作成されていない場合は，作成して返します。

パラメータ	説明
name	ロガーの入力名
apps	ロガーのアペンダの初期リストを示すアペンダポインタの配列へのポインタ
num_apps	アペンダポインタの配列に含まれるアペンダの数

```
void register_app_factory(VISDAppenderFactory* fac);
```

カスタムアペンダファクトリが自分自身をロガーフレームワークに登録するためのAPI。ファクトリは，ファクトリ名のインデックスを持つアペンダファクトリのディクショナリに追加されます。ファクトリがフレームワークに登録されていない場合，そのタイプのインスタンスを作成することはできません。

パラメータ	説明
fac	登録するアペンダファクトリ。

```
VISDAppender_ptr create_app(const char* logger_name, VISDConfig::LogAppenderConfig_ptr p);
```

指定された設定情報を使用して，名前で指定されたロガーのアペンダを作成する API。

パラメータ	説明
logger_name	アペンダインスタンスを作成するロガーの名前
p	ロガーのアペンダインスタンスの設定へのポインタ

```
void register_lyt_factory(VISDLayoutFactory* fact);
```

カスタムレイアウトファクトリが自分自身をロガーフレームワークに登録するためのAPI。

パラメータ	説明
fact	登録する実装済みレイアウトファクトリへのポインタ

```
VISDLayout_ptr create_lyt(const char* logger_name, VISDConfig::LogAppenderConfig_ptr p);
```

レイアウトインスタンスを作成する API。

パラメータ	説明
logger_name	レイアウトを使用する必要があるアペンダインスタンスが関連付けられるロガーの名前
p	ロガーのアペンダインスタンスの設定へのポインタ

VISDLogger

ログインターフェースを提供するクラス。

インクルードファイル

このクラスを使用する場合は，**vdlog.h** ファイルをインクルードする必要があります。

VISDLogger メソッド

```
const char* name() const;
```

ロガーオブジェクトの名前を返します。

```
void log(VISDLogLevel::Level level, const char* message, const char* sourcefile = NULL,
CORBA::ULong linenum = 0, const void* bindata = NULL, size_t binsize = 0);
```

ログメッセージを記録する API。

パラメータ	説明
level	ログメッセージのログレベル
message	ログメッセージのデータ
sourcefile	メッセージをログに記録するソースファイルの名前
linenum	メッセージをログに記録するソースファイルの行番号
bindata	バイナリデータのポインタ
binsize	バイナリデータのサイズ

```
void log(VISDLogLevel::Level level, const char* component, const char* message, const char
*sourcefile = NULL, CORBA::ULong linenum = 0, const void *bindata = NULL, size_t binsize = 0)
```

ログメッセージを記録する API。

パラメータ	説明
level	ログメッセージのログレベル
component	メッセージをログに記録するソース名。ソース名は論理モジュール名で，フィルタリングに使用できます。
message	ログメッセージのデータ
sourcefile	メッセージをログに記録するソースファイルの名前
linenum	メッセージをログに記録するソースファイルの行番号
bindata	バイナリデータのポインタ
binsize	バイナリデータのサイズ

VISDAppenderFactory

実装するアペンダファクトリ実装のインターフェースです。ロガーフレームワークは，アペンダインスタンスを作成するためにこのインターフェースを呼び出します。

インクルードファイル

このクラスを使用する場合は，**vdlog.h** ファイルをインクルードする必要があります。

VISDAppenderFactory メソッド

```
virtual const char* type_name() = 0;
```

このメソッドは,ファクトリのタイプを取得する必要があるときに,ロガーフレームワークによって呼び出されます。たとえば，ファクトリが自分自身をロガーマネージャに登録する際，タイプ名を取得するためにこの API が呼び出されます。タイプ名は，アペンダがロガーを転送する宛先のタイプを識別します。『開発者ガイド』で説明されているように,「stdout」,「rolling」などのタイプ名は使用が制限されています。アペンダタイプとして一意なタイプ名を返す必要があります。

```
virtual VISDAppender_ptr create(const char* logger_name, VISDConfig::LogAppenderConfig_ptr p) = 0;
```

このメソッドは，このファクトリがサポートするアペンダのインスタンスを作成する必要があるときに，ロガーフレームワークによって呼び出されます。戻り値は，目的のアペンダのインスタンスです。

パラメータ	説明
logger_name	アペンダインスタンスを関連付けるロガーの名前
p	ロガーのアペンダインスタンスの設定へのポインタ。

```
virtual void destroy(VISDAppender_ptr p) = 0;
```

このメソッドは，アペンダインスタンスの使用が終了すると，ロガーフレームワークによって呼び出されます。API は，アペンダインスタンスの作成時にアペンダインスタンスに占有されたすべてのリソースを削除します。

パラメータ	説明
p	破棄されるアペンダインスタンスポインタ

VISDAppender

```
class VISDAppender : public VISResource
```

アペンダインターフェースを提供するインターフェース。ロガーオブジェクトは，このインターフェースを使用して特定の宛先にログを記録します。

インクルードファイル

このクラスを使用する場合は，**vdlog.h** ファイルをインクルードする必要があります。

VISDAppender メソッド

```
virtual VISDAppenderFactory* factory() = 0;
```

このアペンダインスタンスを作成した関連するファクトリオブジェクトを返す必要があります。

```
virtual CORBA::Boolean append(const VISDLogRecord& record) = 0;
```

ログメッセージを特定の宛先に転送するためにロガーが使用する API。ログレコードは，ログメッセージ全体を抽象化します。転送が正常に完了すると，API は TRUE を返します。

パラメータ	説明
record	宛先に追加されるログレコード

```
virtual CORBA::Boolean ORB_initialized(void* orb_ptr) = 0;
```

これは，初期化した ORB からの通知です。アペンダが ORB の機能を使用する場合は，この通知を待ってから，TRUE を返す必要があります。そうでない場合は，FALSE を返す必要があります。この通知後，アペンダは ORB インターフェースの使用を開始できます。

パラメータ	説明
orb_ptr	ORB へのリファレンス。

```
virtual void ORB_shutdown() = 0;
```

これは，シャットダウンしようとしている ORB からの通知です。アペンダが ORB の機能を使用していて，この通知を受け取った場合，機能の使用を停止する必要があります。

VISDLayoutFactory

実装するレイアウトファクトリ実装のインターフェースです。ロガーフレームワークは，レイアウトインスタンスを作成するためにこのインターフェースを呼び出します。

インクルードファイル

このクラスを使用する場合は，**vdlog.h** ファイルをインクルードする必要があります。

VISDLayoutFactory メソッド

```
virtual const char* type_name() = 0;
```

このファクトリが作成するレイアウトのタイプ名を返します。

```
virtual VISDLayout_ptr create(const char* logger_name, VISDConfig::LogAppenderConfig_ptr p) = 0;
```

レイアウトインスタンスを作成します。この API は，レイアウトのインスタンスが必要な場合に，ロガーフレームワークによって呼び出されます。

パラメータ	説明
logger_name	このレイアウトインスタンスを必要とするアペンダインスタンスが関連付けられているロガーの名前
p	ロガーのアペンダインスタンスの設定へのポインタ。

```
virtual void destroy(VISDLayout_ptr layout) = 0;
```

フレームワークは，レイアウトの使用が終了し，ファクトリにインスタンスを破棄させる必要が生じると，この API を呼び出します。

パラメータ	説明
layout	破棄する必要があるレイアウトインスタンスへのポインタ

VISDLayout

```
class VISDLayout : public VISResource
```

すべてのレイアウトインスタンスが実装する必要があるインターフェース。アペンダがログメッセージを目的の宛先に出力する前にフォーマットするために，このインターフェースを使用します。

インクルードファイル

このクラスを使用する場合は，**vdlog.h** ファイルをインクルードする必要があります。

VISDLayout メソッド

```
virtual VISDLayoutFactory* factory() = 0;
```

レイアウトインスタンスの作成元のファクトリを返す必要があります。

```
virtual void format(const VISDLogRecord& record, char* buf, CORBA::ULong buf_size,
CORBA::String_var& other_buf) = 0;
```

ログレコードをフォーマットするために，アペンダインスタンスによって呼び出される API。アペンダはバッファを割り当て，バッファをこの API に送信して，レイアウトがメッセージをフォーマットしてこのバッファに設定するように要求します。ただし，アペンダによって送信されたメモリを超える容量が必要な場合，レイアウトは自分自身でメモリを割り当てた other_buffer を使用できます。

パラメータ	説明
record	ログメッセージを含むログレコード
buf	アペンダによって送信されるメモリバッファ。レイアウトは，フォーマットされたメッセージをこのバッファに設定します。
buf_size	アペンダによって送信されるメモリバッファのサイズ
other_buffer	アペンダによって送信されたメモリを超える容量が必要な場合，レイアウトは，このバッファにメモリを割り当て，フォーマットされたテキストをこのバッファに設定できます。

VISDConfig

コンフィグレーション構造体の名前空間クラス。

インクルードファイル

このクラスを使用する場合は，**vdlog.h** ファイルをインクルードする必要があります。

LogAppenderConfig 構造体

```
struct LogAppenderConfig {
  CORBA::String_var appender_name;
  CORBA::String_var appender_type;
  CORBA::String_var layout_type;
};
typedef LogAppenderConfig* LogAppenderConfig_ptr;
```

この構造体は，ロガー上に 1 つのアペンダインスタンス設定を含みます。設定からの読み取りが終わると，この構造体はロガーフレームワークによって値が格納され，ファクトリインターフェースに渡されます。

メンバー	説明
appender_name	ロガーに設定されるアペンダインスタンスの名前
appender_type	アペンダインスタンスのタイプ名。使用する必要があるアペンダファクトリを示します。
layout_type	必要なレイアウトインスタンスのタイプ名。レイアウトインスタンスを取得するために使用するレイアウトファクトリを示します。

VISDLogRecord

ログメッセージを抽象化するクラス。実際のログメッセージのほかに，スレッド ID，タイムスタンプパラメータなど，他のさまざまな状態も取得します。

インクルードファイル

このクラスを使用する場合は，**vdlog.h** ファイルをインクルードする必要があります。

VISDLogRecord メソッド

```
Timestamp get_timestamp() const;
```

ログレコードのタイムスタンプを返します。

```
CORBA::ULong get_seq_number() const;
```

多くのログレコードが同じ時間間隔で記録される場合に，シーケンス番号を返します。

```
CORBA::ULong get_process_id() const;
```

プロセス ID を返します。

```
CORBA::ULong get_thread_id() const;
```

このメッセージを記録したスレッドのスレッド ID を返します。

```
const char* get_thread_name() const;
```

スレッドに名前が付けられている場合は，スレッド名を返します。

```
const char* get_logger_name() const;
```

ロガーオブジェクトの名前を返します。

```
VISDLogLevel::Level get_log_level() const;
```

ログメッセージの詳細レベルを返します。

```
const char* get_component_name() const;
```

メッセージを記録したソースのソース名を返します。

```
const char* get_filename() const;
```

メッセージを記録したファイル名を返します。

```
CORBA::ULong get_line_number() const;
```

ログメッセージの発生元のファイル内の行番号を返します。

```
const char* get_message() const;
```

実際に記録されたメッセージです。

```
const unsigned char* get_bindata() const;
```

ログレコードにあるすべてのバイナリデータを返します。

```
size_t get_binsize() const;
```

バイナリデータのサイズを返します。

VISDLogLevel

詳細列挙体 Level を囲むクラス

インクルードファイル

このクラスを使用する場合は，**vdlog.h** ファイルをインクルードする必要があります。

Level 列挙体

```
enum Level {
  OFF_     = 1000,
  EMERG_   = 800,
  EXCEP_   = 800,
  FATAL_   = 800,
  ALERT_   = 700,
  CRIT_    = 600,
  ERROR_   = 500,
  WARN_    = 400,
  NOTICE_  = 300,
  INFO_    = 200,
  DEBUG_   = 100,
  ALL_     = 0,
  DEFAULT_ = -1
};
```

索引

記号

数値

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

W

あ

い

へ

ほ

ま

む

め

ゆ

り

れ

ろ